8253/8253-5
PROGRAMMABLE INTERVAL TIMER

- MCS-85™ Compatible 8253-5
- 3 Independent 16-Bit Counters
- DC to 2.6 MHz
- Programmable Counter Modes
- Count Binary or BCD
- Single +5V Supply
- Available in EXPRESS
  - —Standard Temperature Range
  - —Extended Temperature Range

The Intel® 8253 is a programmable counter/timer device designed for use as an Intel microcomputer peripheral. It uses NMOS technology with a single +5V supply and is packaged in a 24-pin plastic DIP.

It is organized as 3 independent 16-bit counters, each with a count rate of up to 2.6 MHz. All modes of operation are software programmable.

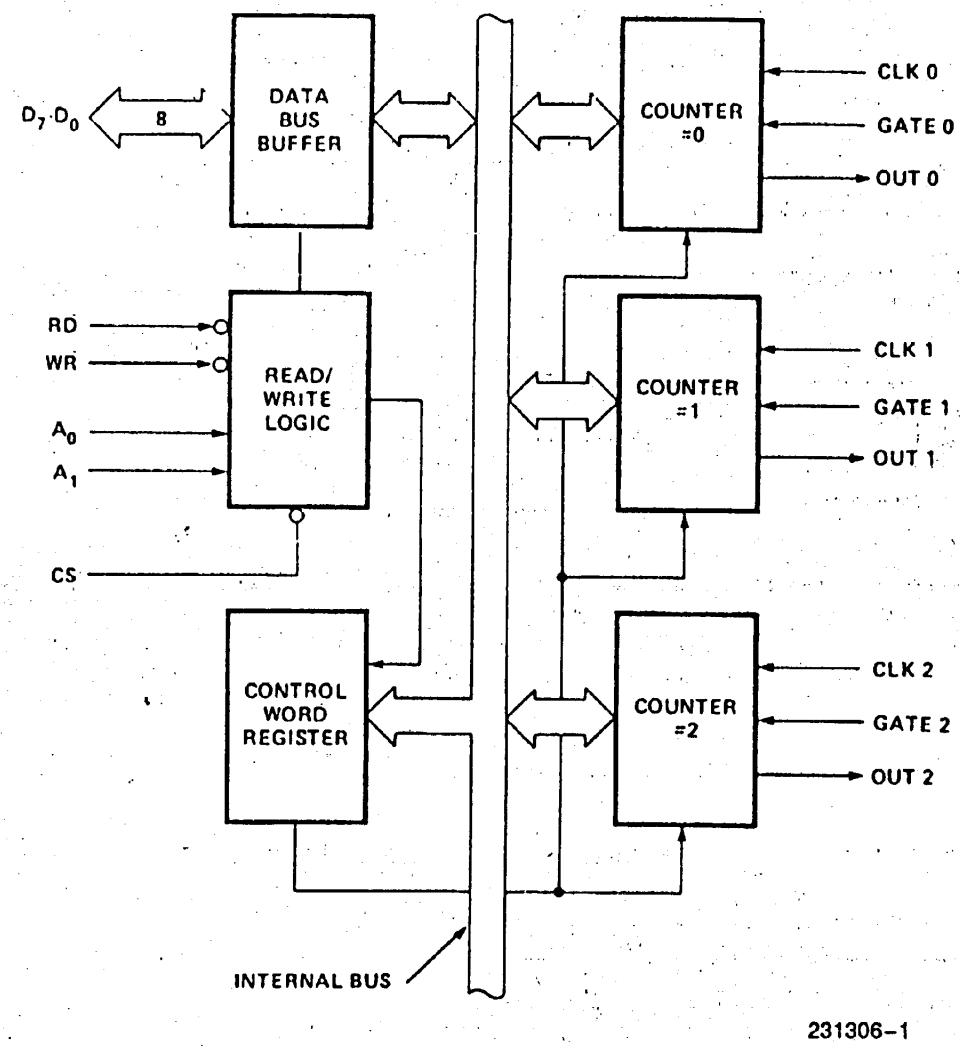


Figure 1. Block Diagram

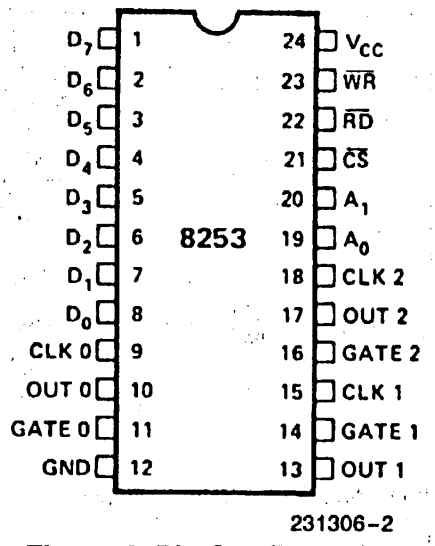


Figure 2. Pin Configuration

$\overline{CS}$	$\overline{RD}$	$\overline{WR}$	A_1	A_0	
0	1	0	0	0	Load Counter No. 0
0	1	0	0	1	Load Counter No. 1
0	1	0	1	0	Load Counter No. 2
0	1	0	1	1	Write Mode Word
0	0	1	0	0	Read Counter No. 0
0	0	1	0	1	Read Counter No. 1
0	0	1	1	0	Read Counter No. 2
0	0	1	1	1	No-Operation 3-State
1	X	X	X	X	Disable 3-State
0	1	1	X	X	No-Operation 3-State

Control Word Register

The Control Word Register is selected when A0, A1 are 11. It then accepts information from the data bus buffer and stores it in a register. The information stored in this register controls the operation MODE of each counter, selection of binary or BCD counting and the loading of each count register.

The Control Word Register can only be written into; no read operation of its contents is available.

Counter #0, Counter #1, Counter #2

These three functional blocks are identical in operation so only a single counter will be described. Each Counter consists of a single, 16-bit, pre-settable, DOWN counter. The counter can operate in either binary or BCD and its input, gate and output are configured by the selection of MODES stored in the Control Word Register.

The counters are fully independent and each can have separate MODE configuration and counting operation, binary or BCD. Also, there are special features in the control word that handle the loading of the count value so that software overhead can be minimized for these functions.

The reading of the contents of each counter is available to the programmer with simple READ operations for event counting applications and special commands and logic are included in the 8253 so that the contents of each counter can be read "on the fly" without having to inhibit the clock input.

8253 SYSTEM INTERFACE

The 8253 is a component of the Intel™ Microcomputer systems and interfaces in the same manner as all other peripherals of the family. It is treated by the systems software as an array of peripheral I/O ports; three are counters and the fourth is a control register for MODE programming.

Basically, the select inputs A0, A1 connect to the A0, A1 address bus signals of the CPU. The $\overline{CS}$ can be derived directly from the address bus using a linear select method. Or it can be connected to the output of a decoder, such as an Intel 8205 for larger systems.

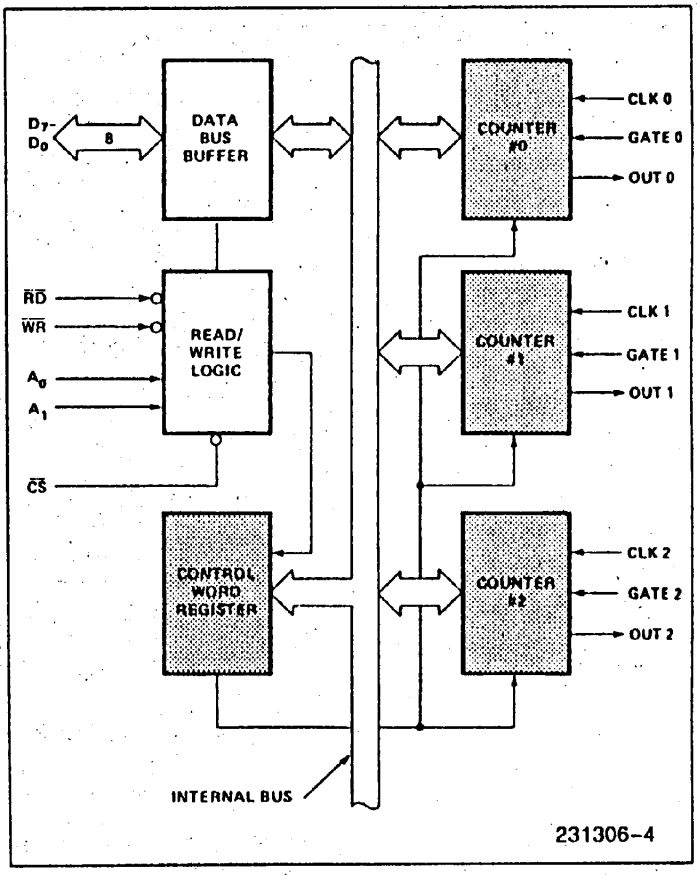


Figure 4. Block Diagram Showing Control Word Register and Counter Functions

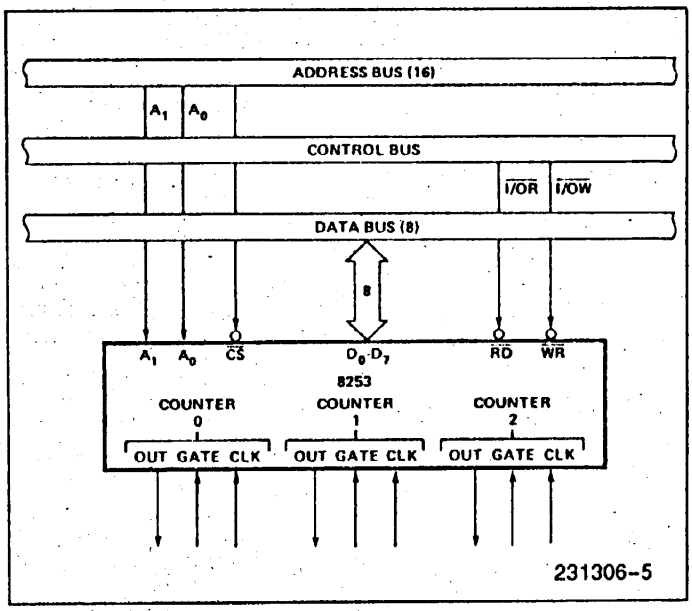


Figure 5. 8253 System Interface

OPERATIONAL DESCRIPTION

General

The complete functional definition of the 8253 is programmed by the systems software. A set of control words *must* be sent out by the CPU to initialize each counter of the 8253 with the desired MODE and quantity information. Prior to initialization, the MODE, count, and output of all counters is undefined. These control words program the MODE, Loading sequence and selection of binary or BCD counting.

Once programmed, the 8253 is ready to perform whatever timing tasks it is assigned to accomplish.

The actual counting operation of each counter is completely independent and additional logic is provided on-chip so that the usual problems associated with efficient monitoring and management of external, asynchronous events or rates to the microcomputer system have been eliminated.

Programming the 8253

All of the MODES for each counter are programmed by the systems software by simple I/O operations.

Each counter of the 8253 is individually programmed by writing a control word into the Control Word Register. (A0, A1 = 11)

Control Word Format

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
SC1	SC0	RL1	RL0	M2	M1	M0	BCD

Definition Of Control

SC—SELECT COUNTER:

SC1	SC0	
0	0	Select Counter 0
0	1	Select Counter 1
1	0	Select Counter 2
1	1	Illegal

RL—READ/LOAD:

RL1	RL0	
0	0	Counter Latching operation (see READ/WRITE Procedure Section).
1	0	Read/Load most significant byte only.
0	1	Read/Load least significant byte only.
1	1	Read/Load least significant byte first, then most significant byte.

M—MODE:

M2	M1	M0	
0	0	0	Mode 0
0	0	1	Mode 1
X	1	0	Mode 2
X	1	1	Mode 3
1	0	0	Mode 4
1	0	1	Mode 5

BCD:

0	Binary Counter 16-Bits
1	Binary Coded Decimal (BCD) Counter (4 Decades)

Counter Loading

The count register is not loaded until the count value is written (one or two bytes, depending on the mode selected by the RL bits), followed by a rising edge and a falling edge of the clock. Any read of the counter prior to that falling clock edge may yield invalid data.

MODE DEFINITION

MODE 0: Interrupt on Terminal Count. The output will be initially low after the mode set operation. After the count is loaded into the selected count register, the output will remain low and the counter will count. When terminal count is reached, the output will go high and remain high until the selected count register is reloaded with the mode or a new count is loaded. The counter continues to decrement after terminal count has been reached.

Rewriting a counter register during counting results in the following:

(1) Write 1st byte stops the current counting.
(2) Write 2nd byte starts the new count.

FUNCTIONAL DESCRIPTION

General

The 8253 is programmable interval timer/counter specifically designed for use with the Intel™ Microcomputer systems. Its function is that of a general purpose, multi-timing element that can be treated as an array of I/O ports in the system software.

The 8253 solves one of the most common problems in any microcomputer system, the generation of accurate time delays under software control. Instead of setting up timing loops in systems software, the programmer configures the 8253 to match his requirements, initializes one of the counters of the 8253 with the desired quantity, then upon command the 8253 will count out the delay and interrupt the CPU when it has completed its tasks. It is easy to see that the software overhead is minimal and that multiple delays can easily be maintained by assignment of priority levels.

Other counter/timer functions that are non-delay in nature but also common to most microcomputers can be implemented with the 8253.

- Programmable Rate Generator
- Event Counter
- Binary Rate Multiplier
- Real Time Clock
- Digital One-Shot
- Complex Motor Controller

Data Bus Buffer

The 3-state, bi-directional, 8-bit buffer is used to interface the 8253 to the system data bus. Data is transmitted or received by the buffer upon execution of INput or OUTput CPU instructions. The Data Bus Buffer has three basic functions.

1. Programming the MODES of the 8253.
2. Loading the count registers.
3. Reading the count values.

Read/Write Logic

The Read/Write Logic accepts inputs from the system bus and in turn generates control signals for overall device operation. It is enabled or disabled by CS so that no operation can occur to change the function unless the device has been selected by the system logic.

$\overline{RD}$ (Read)

A "low" on this input informs the 8253 that the CPU is inputting data in the form of a counters value.

$\overline{WR}$ (Write)

A "low" on this input informs the 8253 that the CPU is outputting data in the form of mode information or loading counters.

A0, A1

These inputs are normally connected to the address bus. Their function is to select one of the three counters to be operated on and to address the control word register for mode selection.

$\overline{CS}$ (Chip Select)

A "low" on this input enables the 8253. No reading or writing will occur unless the device is selected. The $\overline{CS}$ input has no effect upon the actual operation of the counters.

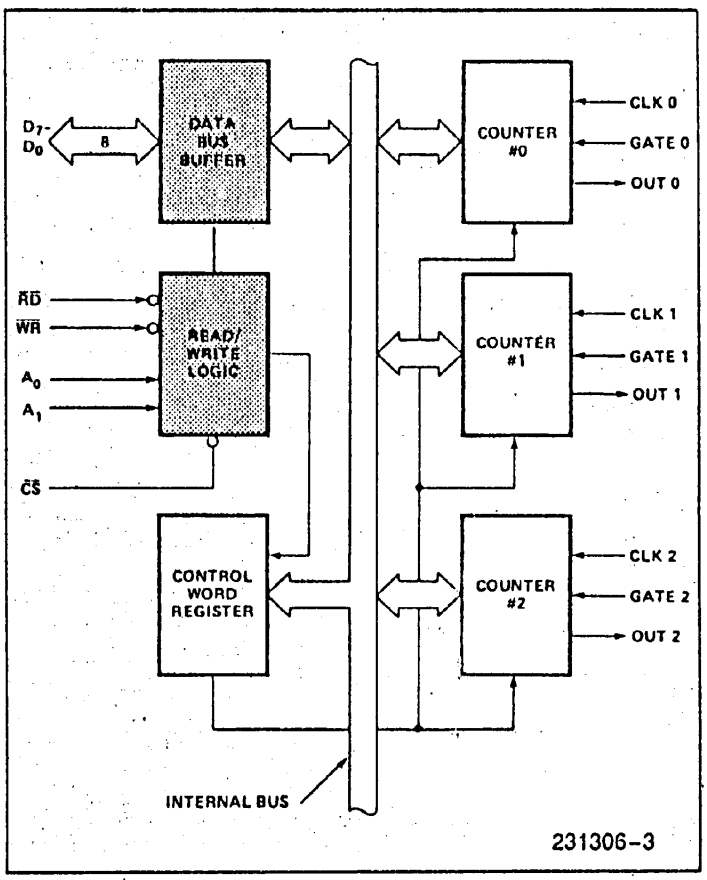


Figure 3. Block Diagram Showing Data Bus Buffer and Read/Write Logic Functions

MODE 1: Programmable One-Shot. The output will go low on the count following the rising edge of the gate input.

The output will go high on the terminal count. If a new count value is loaded while the output is low it will not affect the duration of the one-shot pulse until the succeeding trigger. The current count can be read at any time without affecting the one-shot pulse.

The one-shot is retriggerable, hence the output will remain low for the full count after any rising edge of the gate input.

MODE 2: Rate Generator. Divide by N counter. The output will be low for one period of the input clock. The period from one output pulse to the next equals the number of input counts in the count register. If the count register is reloaded between output pulses the present period will not be affected, but the subsequent period will reflect the new value.

The gate input, when low, will force the output high. When the gate input goes high, the counter will start from the initial count. Thus, the gate input can be used to synchronize the counter.

When this mode is set, the output will remain high until after the count register is loaded. The output then can also be synchronized by software.

MODE 3: Square Wave Rate Generator. Similar to MODE 2 except that the output will remain high until one half the count has been completed (or even numbers) and go low for the other half of the count. This is accomplished by decrementing the counter by two on the falling edge of each clock pulse. When the counter reaches terminal count, the state of the output is changed and the counter is reloaded with the full count and the whole process is repeated.

If the count is odd and the output is high, the first clock pulse (after the count is loaded) decrements the count by 1. Subsequent clock pulses decrement the clock by 2. After timeout, the output goes low and the full count is reloaded. The first clock pulse (following the reload) decrements the counter by 3. Subsequent clock pulses decrement the count by 2 until timeout. Then the whole process is repeated. In this way, if the count is odd, the output will be high for $(N + 1)/2$ counts and low for $(N - 1)/2$ counts.

In Modes 2 and 3, if a CLK source other than the system clock is used, GATE should be pulsed immediately following $\overline{WR}$ of a new count value.

MODE 4: Software Triggered Strobe. After the mode is set, the output will be high. When the count is loaded, the counter will begin counting. On terminal count, the output will go low for one input clock period, then will go high again.

If the count register is reloaded during counting, the new count will be loaded on the next CLK pulse. The count will be inhibited while the GATE input is low.

MODE 5: Hardware Triggered Strobe. The counter will start counting after the rising edge of the trigger input and will go low for one clock period when the terminal count is reached. The counter is retriggerable. The output will not go low until the full count after the rising edge of any trigger.

Signal Status Modes	Low Or Going Low	Rising	High
0	Disables counting	—	Enables counting
1	—	1) Initiates counting 2) Resets output after next clock	—
2	1) Disables counting 2) Sets output immediately high	1) Reloads counter 2) Initiates counting	Enables counting
3	1) Disables counting 2) Sets output immediately high	1) Reloads counter 2) Initiates counting	Enables counting
4	Disables counting	—	Enables counting
5	—	Initiates counting	—

Figure 6. Gate Pin Operations Summary

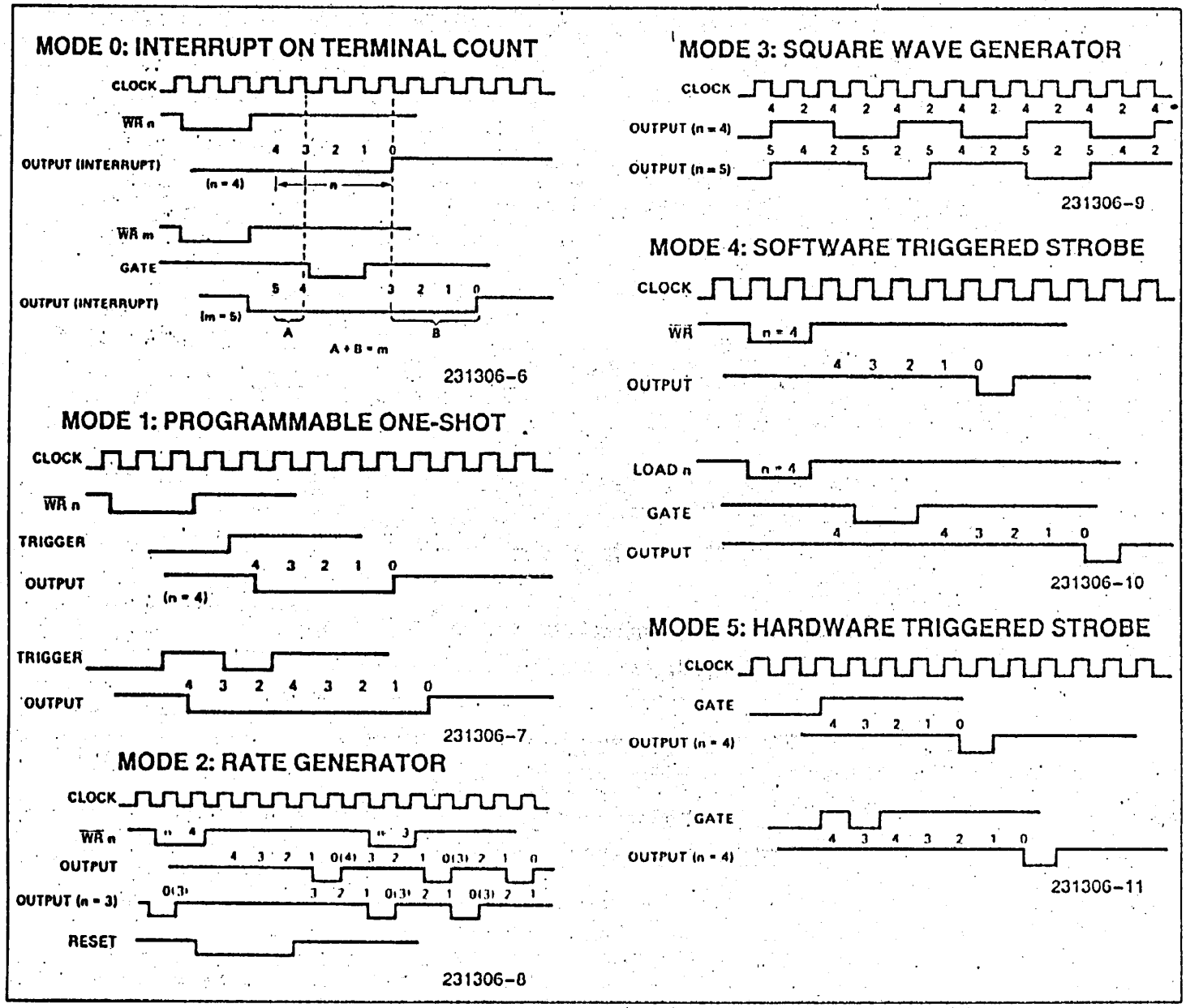


Figure 7. 8253 Timing Diagrams

8253 READ/WRITE PROCEDURE

Write Operations

The systems software must program each counter of the 8253 with the mode and quantity desired. The programmer must write out to the 8253 a MODE control word and the programmed number of count register bytes (1 or 2) prior to actually using the selected counter.

The actual order of the programming is quite flexible. Writing out of the MODE control word can be in any sequence of counter selection, e.g., counter #0 does not have to be first or counter #2 last. Each counter's MODE control word register has a separate address so that its loading is completely sequence independent. (SC0, SC1).

The loading of the Count Register with the actual count value, however, must be done in exactly the sequence programmed in the MODE control word (RL0, RL1). This loading of the counter's count register is still sequence independent like the MODE control word loading, but when a selected count register is to be loaded it *must* be loaded with the number of bytes programmed in the MODE control word (RL0, RL1). The one or two bytes to be loaded in the count register do not have to follow the associated MODE control word. They can be programmed at any time following the MODE control word loading as long as the correct number of bytes is loaded in order.

All counters are down counters. Thus, the value loaded into the count register will actually be decremented. Loading all zeros into a count register will result in the maximum count (2^{16} for Binary or 10^4 for BCD). In MODE 0 the new count will not restart until the load has been completed. It will accept one of two bytes depending on how the MODE control words (RL0, RL1) are programmed. Then proceed with the restart operation.

	MODE Control Word Counter n
LSB	Counter Register byte Counter n
MSB	Counter Register byte Counter n

NOTE:
Format shown is a simple example of loading the 8253 and does not imply that it is the only format that can be used.

Figure 8. Programming Format

			A1	A0
No. 1		MODE Control Word Counter 0	1	1
No. 2		MODE Control Word Counter 1	1	1
No. 3		MODE Control Word Counter 2	1	1
No. 4	LSB	Count Register Byte Counter 1	0	1
No. 5	MSB	Count Register Byte Counter 1	0	1
No. 6	LSB	Count Register Byte Counter 2	1	0
No. 7	MSB	Count Register Byte Counter 2	1	0
No. 8	LSB	Count Register Byte Counter 0	0	0
No. 9	MSB	Count Register Byte Counter 0	0	0

NOTE:
The exclusive addresses of each counter's count register make the task of programming the 8253 a very simple matter, and maximum effective use of the device will result if this feature is fully initilized.

Figure 9. Alternate Programming Formats

Read Operations

In most counter applications it becomes necessary to read the value of the count in progress and make a computational decision based on this quantity. Event counters are probably the most common application that uses this function. The 8253 contains logic that will allow the programmer to easily read the contents of any of the three counters without disturbing the actual count in progress.

There are two methods that the programmer can use to read the value of the counters. The first method involves the use of simple I/O read operations of the selected counter. By controlling the A0, A1 inputs to the 8253 the programmer can select the counter to be read (remember that no read operation of the mode register is allowed A0, A1-11). The only requirement with this method is that in order to assure a stable count reading the actual operation of the selected counter *must be inhibited* either by controlling the Gate input or by external logic that inhibits the clock input. The contents of the counter selected will be available as follows:

First I/O Read contains the least significant byte (LSB).

Second I/O Read contains the most significant byte (MSB).

Due to the internal logic of the 8253 it is absolutely necessary to complete the entire reading procedure, If two bytes are programmed to be read, then two bytes *must* be read before any loading WR command can be sent to the same counter.

Read Operation Chart

A1	A0	RD	
0	0	0	Read Counter No. 0
0	1	0	Read Counter No. 1
1	0	0	Read Counter No. 2
1	1	0	Illegal

Reading While Counting

In order for the programmer to read the contents of any counter without effecting or disturbing the counting operation the 8253 has special internal logic that can be accessed using simple $\overline{WR}$ commands to the MODE register. Basically, when the programmer wishes to read the contents of a selected counter "on the fly" he loads the MODE register with a special code which latches the present count value into a storage register so that its contents contain an accurate, stable quantity. The programmer then issues a normal read command to the selected counter and the contents of the latched register is available.

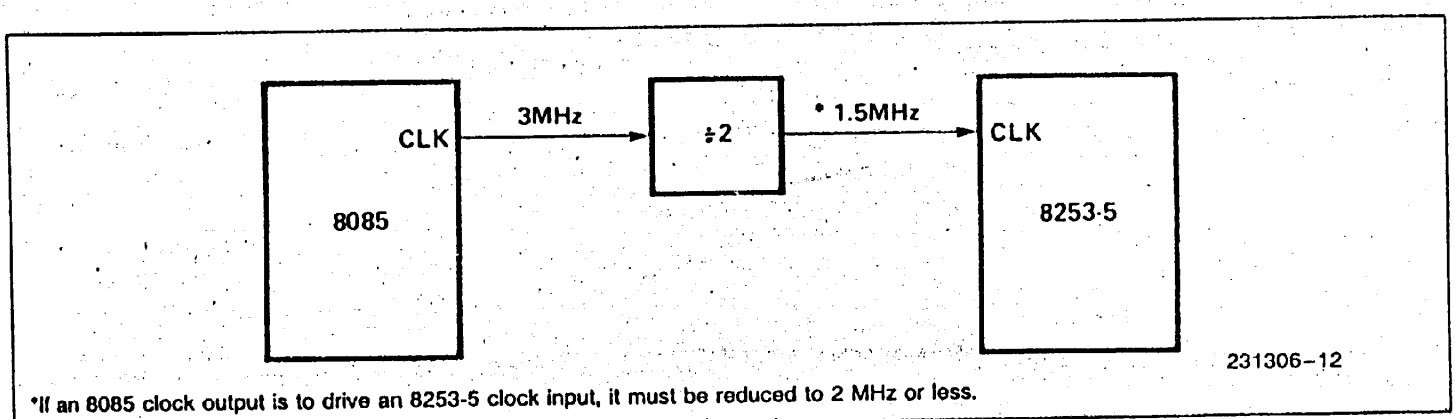
MODE Register for Latching Count

A0, A1 = 11

D7	D6	D5	D4	D3	D2	D1	D0
SC1	SC0	0	0	X	X	X	X

SC1, SC0— specify counter to be latched.
D5, D4 — 00 designates counter latching operation.
X — don't care.

The same limitation applies to this mode of reading the counter as the previous method. That is, it is mandatory to complete the entire read operation as programmed. This command has no effect on the counter's mode.



*If an 8085 clock output is to drive an 8253-5 clock input, it must be reduced to 2 MHz or less.

Figure 10. MCS-85™ Clock Interface*

ABSOLUTE MAXIMUM RATINGS*

Ambient Temperature Under Bias .....0°C to 70°C
Storage Temperature .....−65°C to +150°C
Voltage On Any Pin
with Respect to Ground.....−0.5V to 7V
Power Dissipation .....1 Watt

**Notice: Stresses above those listed under "Absolute Maximum Ratings" may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or any other conditions above those indicated in the operational sections of this specification is not implied. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.*

D.C. CHARACTERISTICS T_A = 0°C to 70°C, V_{CC} = 5V ±10%*

Symbol	Parameter	Min	Max	Unit	Test Conditions
V_{IL}	Input Low Voltage	−0.5	0.8	V	
V_{IH}	Input High Voltage	2.2	V_{CC} + .5V	V	
V_{OL}	Output Low Voltage		0.45	V	(Note 1)
V_{OH}	Output High Voltage	2.4		V	(Note 2)
I_{IL}	Input Load Current		±10	μA	V_{IN} = V_{CC} to 0V
I_{OFL}	Output Float Leakage		±10	μA	V_{OUT} = V_{CC} to 0.45V
I_{CC}	V_{CC} Supply Current		140	mA	

CAPACITANCE T_A = 25°C, V_{CC} = GND = 0V

Symbol	Parameter	Min	Typ	Max	Unit	Test Conditions
C_{IN}	Input Capacitance			10	pF	fc = 1 MHz
$C_{I/O}$	I/O Capacitance			20	pF	Unmeasured pins returned to V_{SS}

A.C. CHARACTERISTICS T_A = 0°C to 70°C, V_{CC} = 5.0V ±10%, GND = 0V*

Bus Parameters(3)

READ CYCLE

Symbol	Parameter	8253		8253-5		Unit
		Min	Max	Min	Max	
t_{AR}	Address Stable before $\overline{READ}$	50		30		ns
t_{RA}	Address Hold Time for $\overline{READ}$	5		5		ns
t_{RR}	$\overline{READ}$ Pulse Width	400		300		ns
t_{RD}	Data Delay from $\overline{READ}$(4)		300		200	ns
t_{DF}	$\overline{READ}$ to Data Floating	25	125	25	100	ns
t_{RV}	Recovery Time between $\overline{READ}$ and Any Other Control Signal	1		1		μs

A.C. CHARACTERISTICS (Continued)

WRITE CYCLE

Symbol	Parameter	8253		8253-5		Unit
		Min	Max	Min	Max	
t_{AW}	Address Stable before $\overline{WRITE}$	50		30		ns
t_{WA}	Address Hold Time for $\overline{WRITE}$	30		30		ns
t_{WW}	$\overline{WRITE}$ Pulse Width	400		300		ns
t_{DW}	Data Set Up Time for $\overline{WRITE}$	300		250		ns
t_{WD}	Data Hold Time for $\overline{WRITE}$	40		30		ns
t_{RV}	Recovery Time between $\overline{WRITE}$ and Any Other Control Signal	1		1		μs

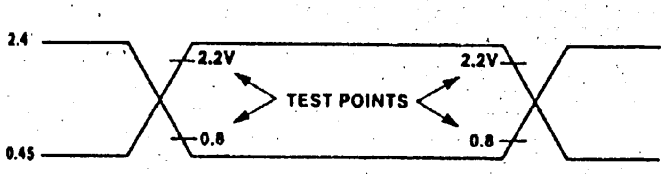
CLOCK AND GATE TIMING

Symbol	Parameter	8253		8253-5		Unit
		Min	Max	Min	Max	
t_{CLK}	Clock Period	380	dc	380	dc	ns
t_{PWH}	High Pulse Width	230		230		ns
t_{PWL}	Low Pulse Width	150		150		ns
t_{GW}	Gate Width High	150		150		ns
t_{GL}	Gate Width Low	100		100		ns
t_{GS}	Gate Set Up Time to CLK ↑	100		100		ns
t_{GH}	Gate Hold Time after CLK ↑	50		50		ns
t_{OD}	Output Delay from CLK ↓ (4)		400		400	ns
t_{ODG}	Output Delay from Gate ↓ (4)		300		300	ns

NOTES:
1. I_{OL} = 2.2 mA.
2. I_{OH} = −400 μA.
3. AC timings measured at V_{OH} 2.2, V_{OL} = 0.8.
4. C_L = 150 pF.

*For Extended Temperature EXPRESS, use M8253 electrical parameters.

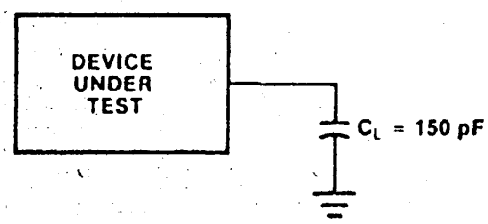
A.C. TESTING INPUT, OUTPUT WAVEFORM



231306-13

A.C. Testing: Inputs are driven at 2.4V for a Logic "1" and 0.45V for a Logic "0". Timing measurements are made at 2.2V for a Logic "1" and 0.8V for a Logic "0".

A.C. TESTING LOAD CIRCUIT

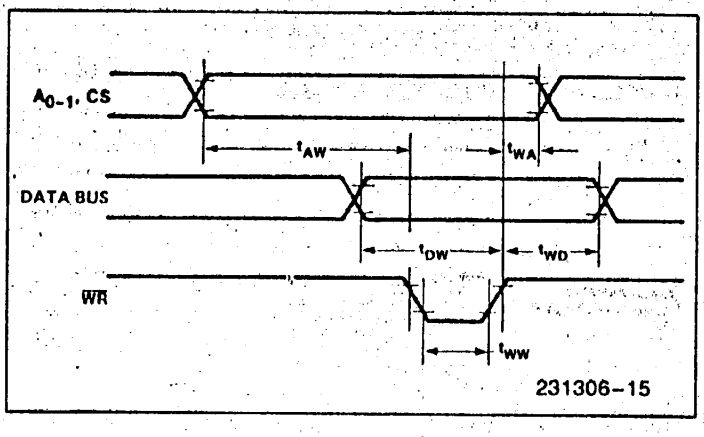


231306-14

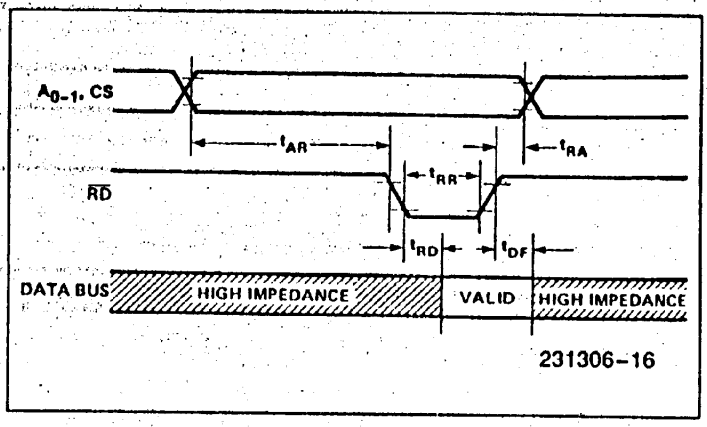
C_L Includes Jig Capacitance

WAVEFORMS

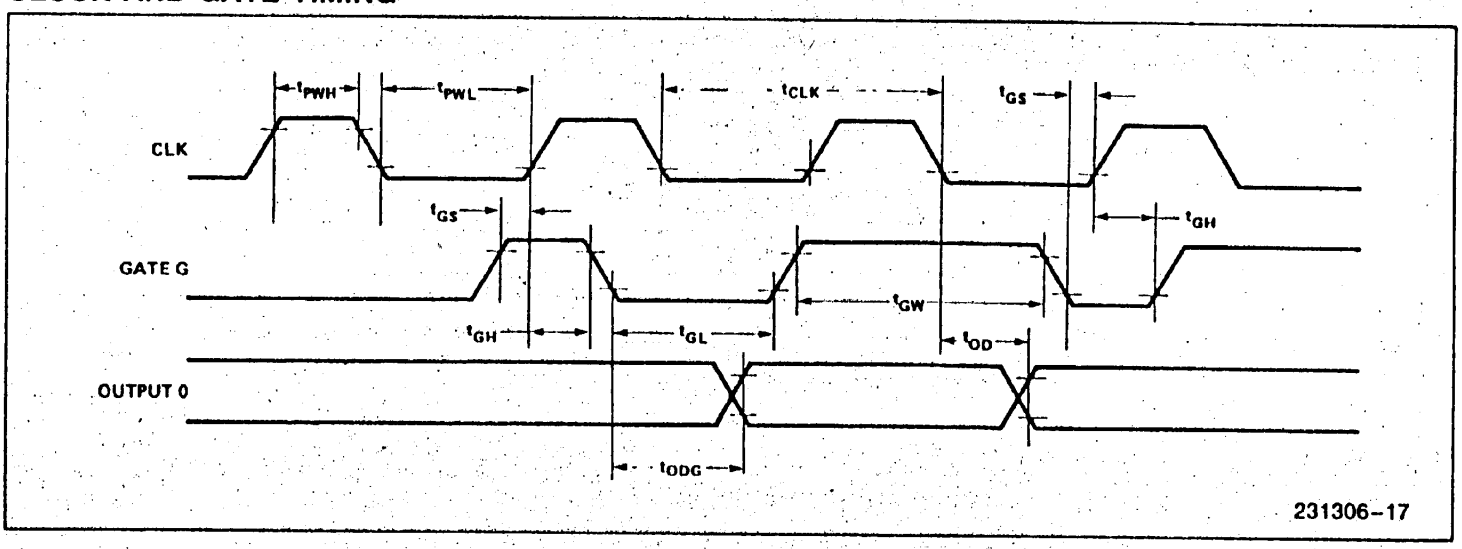
WRITE TIMING



READ TIMING



CLOCK AND GATE TIMING

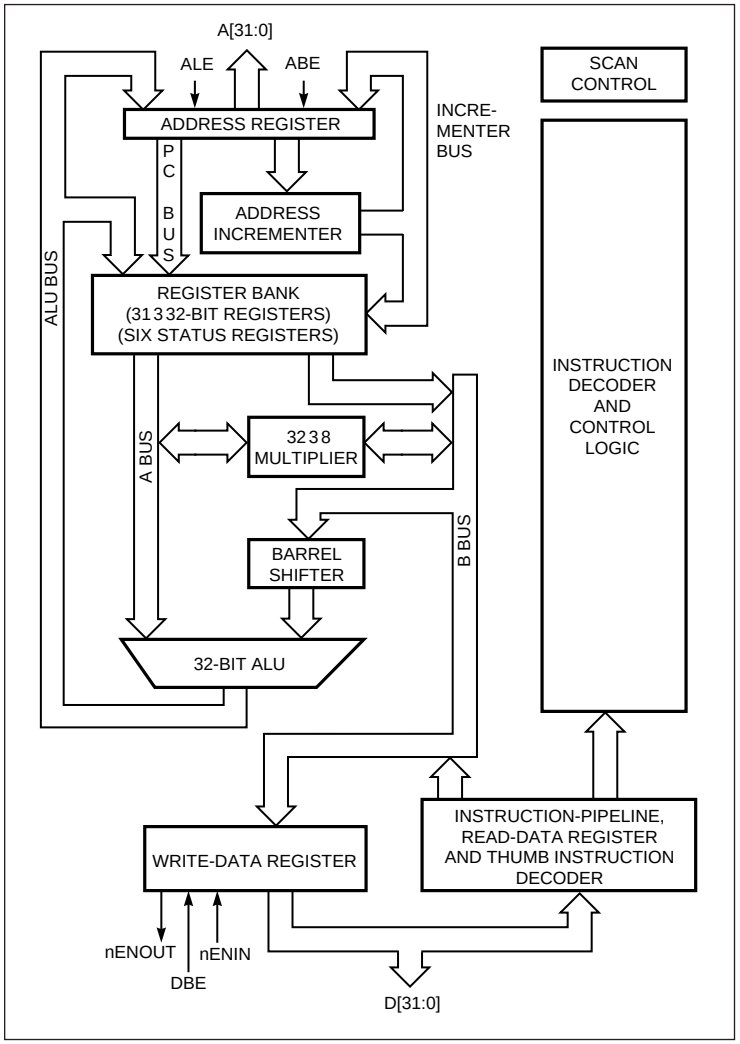


ARM processors

ARM designs mP cores and cached macrocells for its licensees. Partners offering ASICs with embedded ARM cores are Atmel/ES2, Cirrus Logic, Mitel, IBM, LG Semicon (www.lgsemicon.co.kr/), LSI Logic, Lucent (www.lucent.com), National Semiconductor, NEC, Oki, Samsung, Seiko Epson (www.epson.co.jp/), Sharp, Symbios Logic (www.symbios.com), TI, and VLSI. Some partners offer the ARM core in embedded products for vertical markets.

ARM processors comprise the ARM7 Thumb, ARM9 Thumb, and StrongARM product families. (ARM will announce ARM10 in October.) All the processors support the ARM instruction set, providing full software compatibility over a range of performance and cost.

The ARM cores and cached macrocells implement a load/store architecture and have 31 general-purpose registers with 16 simultaneously visible. A fast interrupt has a minimum latency of four processor cycles and uses seven private registers to minimize state-saving overhead. All registers, excluding the program counter, are general-purpose, although a set of conventions, the ARM Procedure Call Standard, governs the registers' use for C compatibility.



The ARM cores and cached macrocells support user and supervisor modes for controlling access; they handle interrupt-request, fast-interrupt-request, abort, and undefined exception-processing modes. Modes use register windows to overlay some of the 16 general-purpose registers.

The Thumb architectural extension is primarily a 16-bit subset of the 32-bit instruction set. On execution, the Thumb module, residing within the instruction pipeline, decompresses the 16-bit instructions back to 32-bit instructions without added delay. The Thumb module adds about 6% to the core's die size but helps increase code density and overcome the waste from using 32-bit fixed-length instructions.

The bus clock for most ARM cached macrocells can be synchronous or asynchronous with respect to the internal cache clock. All ARM cached macrocells contain a write buffer, which lets execution continue while writes are pending. The buffer holds 8 words at four independent addresses.

The ARM7 Thumb family comprises the ARM7TDMI core and ARM7x0T cached macrocells. This architecture, Version 4T, consists of a three-stage—fetch, decode, and execute—pipeline to achieve single-cycle instruction execution. All cores use an 8-bit Booth multiplier, which executes in five or fewer cycles for 32 3 32-bit multiply and offers 64-bit multiplication. The ARM740T integrates a simplified memory-management unit (MMU) that allows you to specify eight memory areas by individually programming their base address, size, cache control, write-buffer control, and access permissions. This approach simplifies the programmer model and reduces the core size to less than that of the ARM710T and ARM720T.

ARM based the ARM9 Thumb family, available as the ARM940T, on the ARM9TDMI core. The core is also an implementation of the ARM Version 4T architecture but with a five-stage—fetch, decode, execute, memory, and write-back—pipeline. The additional pipeline depth and design implementation double the performance over the ARM7 Thumb cores. The bus architecture also differs, using a Harvard approach compared with the ARM7 Thumb core's von Neumann architecture. The ARM940T implements the same MMU as the ARM740T. You can use the cache in write-through and -back modes; write-back mode reduces the number of external transactions from the core.

StrongARM uses a five-stage pipeline and Harvard architecture and supports Version 4 of the ARM architecture. It provides a fourfold increase in performance over the ARM7 Thumb cores. Intel now produces and develops StrongARM, which is available as the standard SA-110 processor and as part of custom logic products.

The cores avoid excess pipeline flushes—StrongARM by using early branch execution and ARM7 by using static branch prediction, always taking the rear branch as in a loop. The SA-110 has separate instruction and data MMUs. The translation-look-aside buffers (TLBs) have 32 entries that can each map a segment, large page, or small page and use a round-robin replacement algorithm. The data TLB supports both

flush-all and flush-single-entry functions, and the instruction TLB supports only the flush-all function.

Power management: All the ARM processor cores and cached macrocells are static designs. Furthermore, the designs use gated clocks and transparent latches, clocking the logic only during an operation (but not during a wait state).

Special instructions: ARM has 11 basic types of fixed-length instructions, which execute conditionally—not just branch—and reduce the need for short pipeline-flushing branches. A not-taken instruction executes in one cycle. Taken branches incur a three-cycle delay. The 16 execution-condition codes include equal, not equal, always, negative, and overflow. The ARM lacks explicit shift instructions; instead, all ALU operations can perform an optional shift operation in one execution cycle. The processors have block-data-transfer instructions to load and store data from any subset of the 16 general-purpose registers.

ARM processors lack an integer-divide instruction; however, the chips have multiply and multiply-accumulate (MAC) instructions. The MAC instruction speeds math-intensive applications. ARM processors can synthesize division and multiplication by a constant using sequences of one or more shift-and-add or shift-and-subtract instructions. (For example, division by 4 and multiplication by 5 each take one cycle.)

Special on-chip peripherals: The ARM7 Thumb and ARM9 Thumb processor cores have integrated EmbeddedICE logic, allowing you to debug the core via a JTAG interface. The ARM Advanced Microcontroller Bus Architecture (AMBA) interface is the standard bus interface to ARM7 Thumb and ARM9 Thumb cached macrocells.

Development tools: ARM offers a variety of software-development tools and hardware-development platforms, including the ARMUlator instruction-set emulator. A range of third-party development tools and operating systems also support the ARM architecture. Cygnus Solutions (www.cygnus.com), Embedded Performance (www.episupport.com), Green Hills Software (www.ghs.com), Metaware (www.metaware.com), Microtec (www.microtec.com), Microware Systems Corp (www.microware.com), and Wind River (www.windriver.com) offer development-tool chains and compilers. Accelerated Technology (www.atinucleus.com), Chorus Systems (www.sun.com), CMX Co (www.cmx.com), Embedded Performance, Etnoteam (www.etnoteam.it/), Geoworks (www.geoworks.com), Integrated Systems (www.isi.com), Microsoft (www.microsoft.com), Microware, Psion Software (www.psion.com), US Software (www.uss.com), and Wind River provide RTOS support. Hewlett-Packard (www.hp.com), Lauterbach (www.lauterbach.com), and Yokogawa Digital Corp (www.yokogawa.com) offer a debugger and an in-circuit emulator.

80386

The 386 has disappeared from the desktop-PC market but has developed a strong presence in embedded-PC applications. Register-based, the 80386 architecture has four general-purpose registers, four index/pointer registers, six 16-bit segment registers, and two 32-bit status and control registers. Intel's 8086 designers used 64-kbyte segments to extend addressing to 1 Mbyte. The 80386 also uses segmentation; however, because the general-purpose and index/pointer registers are now 32 bits, the segment limits extend to the full 4-Gbyte addressing range, and a segment register references a segment descriptor with a 32-bit base address. These descriptors also carry addressing-range and protection limits to prevent data accesses into code, data that executes as code, and access to inner privilege levels by outer levels.

Hardware-descriptor registers hold segment-access rights and segment-base address and size limits. In protected-mode addressing, a 16-bit selector points to a segment descriptor and furnishes a base address. The base address adds to the 32-bit effective address, producing a 32-bit linear address, which the 80386 then uses as a physical or linear-page address.

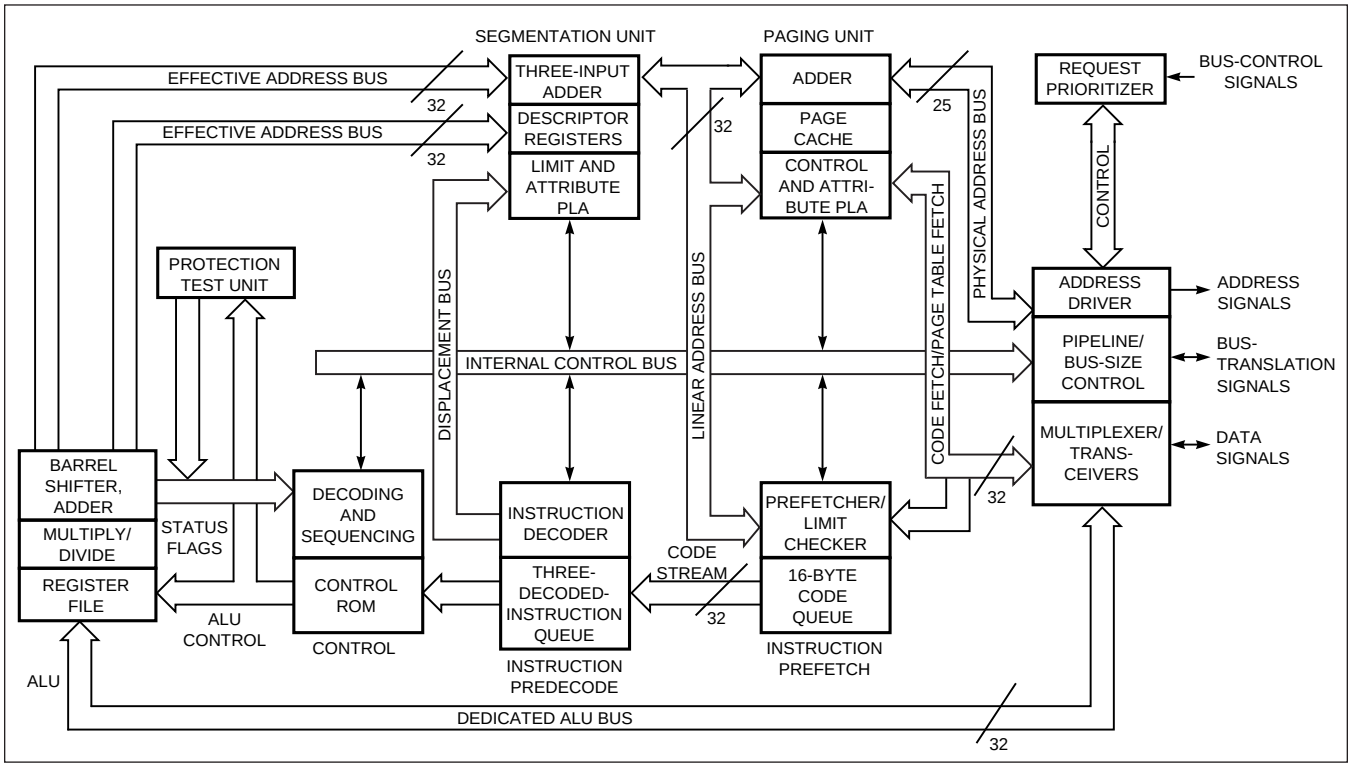
The 386 has four code/data breakpoint registers and two control registers for debugging. You can set the breakpoint registers with addresses for halting execution on a program or data access.

Power management: System-management mode (SMM), a power-management mechanism, enables code to control CPU power without rewriting or revamping operating software. The CPU enters SMM via a hardware interrupt, the system-management interrupt (SMI); the SMI code can set SMMs to reduce chip power dissipation. Integrated versions of the 386, including Intel's 386EX, have idle and power-down modes: Idle discontinues CPU processing but keeps peripherals active, and power-down shuts down the entire chip. AMD's 386SC300 chip has low-speed mode, during which the CPU goes to 0.5 MHz; doze, which stops the CPU, system, and DMA clocks; sleep, which stops additional clocks and peripherals; and suspend, which stops everything except the real-time clock and memory.

Special instructions: The 386 instruction set is a superset of the 8086/186. To support SMM, the 386 has seven additional instructions, such as RSM (resume), which causes the processor to resume from SMM.

Special on-chip peripherals: Intel's 386EX peripherals include a serial-I/O unit, a chip select, a clock generator, a DMA- and bus-arbitrator unit, a DRAM-refresh-control unit, an interrupt-control unit, a memory-management unit, and a parallel-I/O unit. AMD's ElanSC300 combines an Am386 CPU with a PC/AT chip set and essential embedded-PC peripherals. The ElanSC300 also includes mobile-computing peripherals, such as PLL clock generators, PCMCIA-card support, LCD-graphics control, a memory controller, DMA and interrupt controllers, a real-time clock, a serial port, and a parallel port.

Development tools: Numerous third-party vendors support the 386 architecture. They provide tools that include assemblers, compilers, linkers/locators, remote software debuggers, software simulators, and integrated design environments for software development. In addition, several vendors provide utilities, such as flash-programming, device-driver, and flash-translation-layer implementations. Hardware tools include in-circuit emulators, logic analyzers, evaluation platforms, and single-board computers. Operating-system support includes DOS and windowed OSs and a variety of real-time OSs from small, royalty-free microkernels to feature-rich graphical-user-interface RTOSs.



The 486 builds on the 386 architecture by adding a more efficient memory bus; an on-chip floating-point unit; an on-chip, unified, Level 1 cache; and a RISC-like implementation for the core load/store instructions. The 32-bit 80486 implementation retains the i386's complex instruction set but relies on a pipelined RISC-like implementation to speed execution for simple load/store instructions. The standard 486 microarchitecture has a five-stage pipeline and uses two of those stages, decoder stages D1 and D2, to decode the complex instruction set.

The 486 chips use 1- to 15-byte-long instructions for complex operations. The two decoder stages give the hardware time to delineate and decode the instructions waiting in the instruction queue. The instruction or byte-code queue holds 32 bytes for decoding. By fetching 4 words at a time from off-chip or local memory, the hardware minimizes contention between data and instruction accesses of the cache. To speed processing, the hardware loads and writes cache lines in 4-word bursts.

The DX4 has a unified cache that is four-way set-associative and implements a write-through policy: Writes to cache pass through to memory, which raises memory bandwidth. The 486's bus and cache implement a bus-snooping protocol for multiprocessor operation. The bus is more efficient than that of the 386 and has a two-clock single read or write; 4-word read bursts take five cycles and constitute most 486 bus accesses. The processors also support secondary Level 2 cache for both single-processor and multiprocessor operation, as well as write-through/write-back protocols.

The 486 has four code/data breakpoint registers and two control registers for debugging. You can set the breakpoint registers with addresses for halting execution on a program or data access.

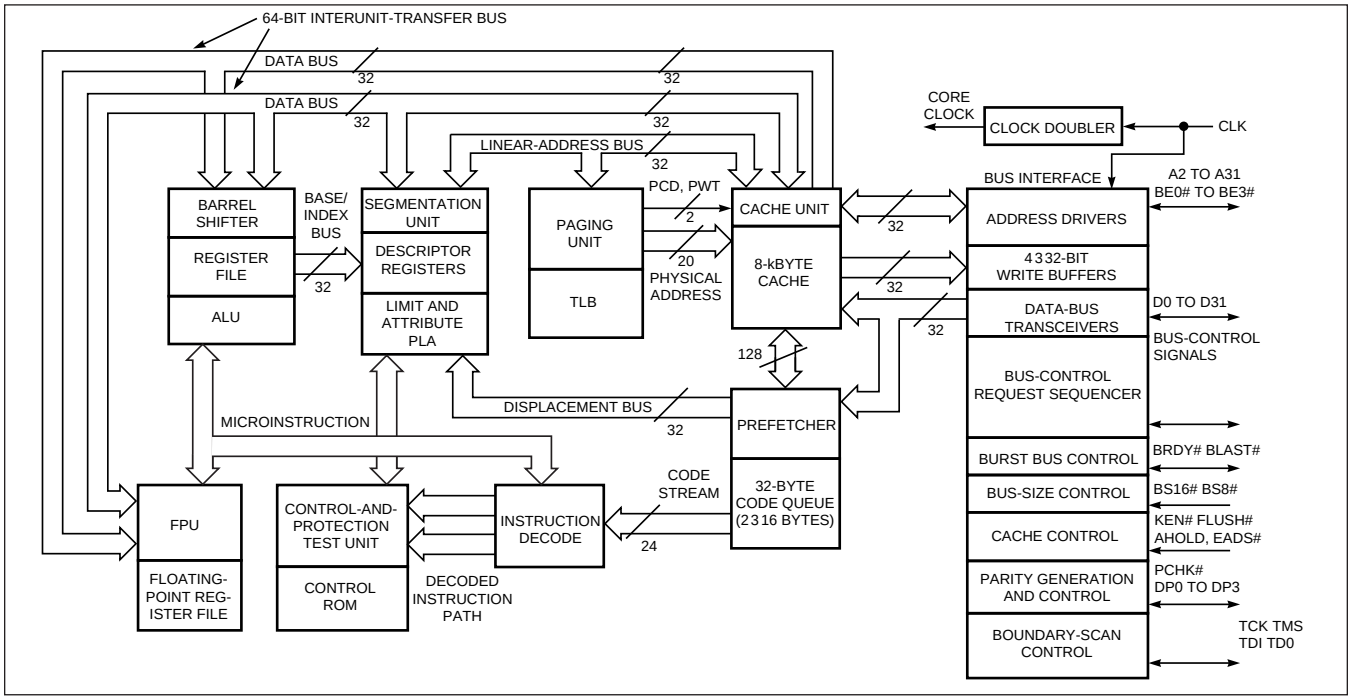
Power management: The standard 486 employs system-management mode (SMM) for power management, which enables code to control CPU power without rewriting or revamping operating software. The CPU enters SMM via a hardware interrupt, the system-management interrupt (SMI); the SMI code can set SMMs to reduce chip power dissipation. A halt instruction powers down most of the CPU's logic.

Special instructions: The 486 instruction set builds upon that of the 80386, adding instructions such as byte swap, exchange and add, compare and exchange, invalidate data cache, write-back and invalidate data cache, invalidate translation-look-aside-buffer entry, processor identification, and SMM resume.

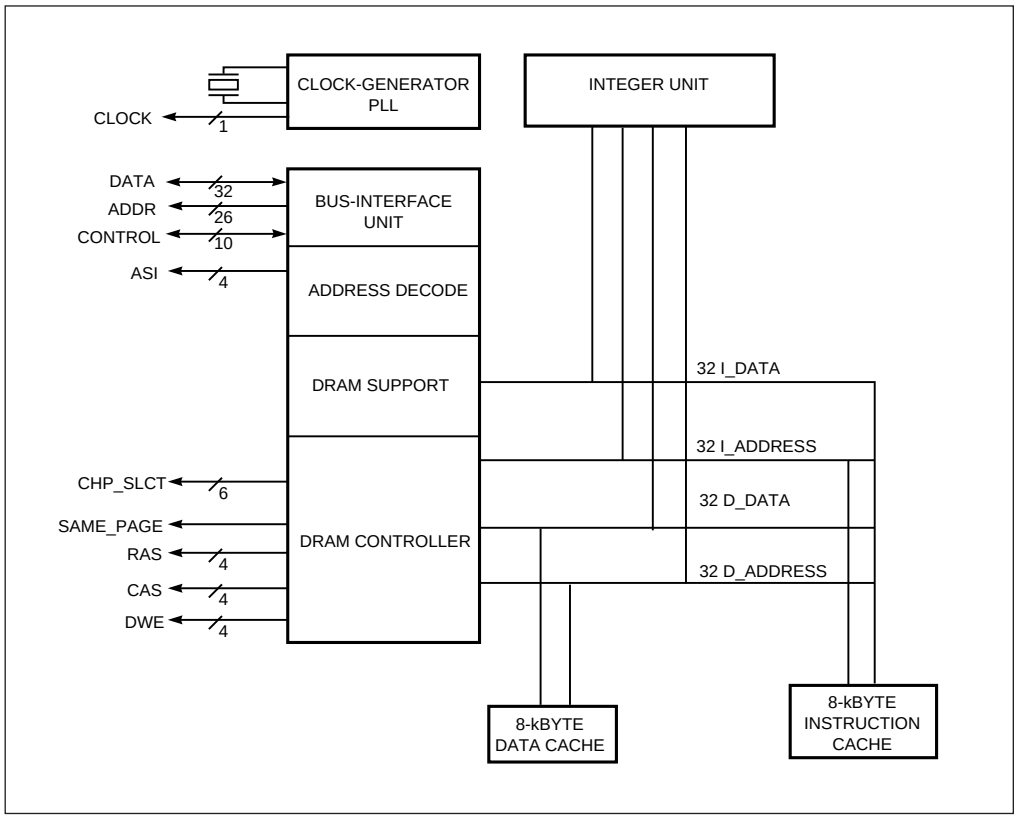
Special on-chip peripherals: AMD's ElanSC400 microcontroller combines an Am486 CPU with a PC/AT chip set and essential embedded-PC peripherals. The ElanSC400 also includes mobile-computing peripherals, such as PLL clock generators, PCMCIA-card support, LCD-graphics control, a memory controller, DMA and interrupt controllers, a real-time clock, a serial port, and a parallel port.

Development tools: Most of the tool support for the 486 is the same as that for the 386. AMD, Intel, and National offer evaluation kits for each of their 486 processors. For example, AMD's $950 mforCE (micro for CE) demonstration system for mobile and embedded product development uses the AMD's ElanSC400 microcontroller and the QNX (www.qnx.com) Realtime OS or Microsoft (www.microsoft.com) Windows CE OS. The board contains an external matrix scan keyboard, a flash minicard slot, a PCMCIA type 2 slot, an IrDA interface, a serial port, a 10-bit digitizer controller for pen input, an audio chip, and 4 Mbytes of ROM and DRAM.

Second sources: There are no pin-compatible second sources for the 80486. AMD, Intel, and STMicroelectronics act as second sources for some implementations.



Fujitsu SPARClite



Fujitsu based its MB8683x, or SPARClite, family on V8E spec, SPARC International's (www.sparc.com) embedded specification. The family features a 32-bit ALU and uses a load/store architecture with a register stack of 136 32-bit registers. (The 86933H chips have 104.) Eight reserved registers hold global values. The remaining registers arrange into eight overlapping register windows, one for each subroutine. This setup speeds procedure calls and interrupt processing. Multiple contexts can be present concurrently by limiting the number of registers for a task.

Fujitsu engineers extended the SPARC pipeline for the SPARClite family to fetch, decode, execute, memory, and write-back stages. The memory stage minimizes the effects of load/store operations and reduces a load/store to one-cycle execution. The stage is idle for nonload/store operations.

All SPARClite mPs have separate data and instruction caches. The caches are two-way set-associative and have 16- or 32-byte cache lines. You can lock and not swap out critical cache lines on chip. The MB86832 also incorporates a debug-support unit and an emulator bus, which makes instruction streams visible even in on-chip cache. Debugging registers hold data values or addresses for individual and range breakpoints.

The SPARClite processors run with DRAM, synchronous DRAM, SRAM, and ROM/EPROM. The memory interface handles page-mode DRAM for low-cost, high-speed access using a 32-byte burst mode. The memory interface includes a refresh generator for DRAMs, programmable wait states for slower memory, and programmable chip selects for memory banking. Boot-up memory interfaces are programmable; SPARClite CPUs can boot from 8-, 16-, or 32-bit ROM/EPROM.

Power management: SPARClite processors incorporate power-down modes, and a power-management register controls shutdown of the floating-point unit.

Special instructions: The SPARClite implements the SPARC V8 specification, which includes a hardware multiply instruction and a software division using divide by 4. Other special instructions include scan word looking for first changed bit or first one or zero, load/store double word, save/restore caller (uses register windows), tagged add/subtract (generates overflow if most significant bits 0 and 1 are not 0), atomic math and swap, and generate trap from conditions.

Special on-chip peripherals: SPARClite processors come with a 24-bit timer that has an 8-bit prescaler and a 16-bit counter. You can program this counter to operate in periodic-interrupt, time-out-interrupt, or square-wave-generator mode. The mP's debug-and-support unit (DSU) comprises two 4-bit emulator buses for data and status and two control signals that enable and set the breakpoint of an in-circuit emulator for hardware debugging and software development. The SPARClite's DSU has six breakpoint-descriptor registers and supports five hardware-monitoring debugging modes.

Development tools: SPARClite shares many of the development tools that support the SPARC architecture, including compilers and debuggers. Fujitsu supplies $89 evaluation kits and full-featured evaluation boards and monitors. Fujitsu works with Wind River Systems (www.windriver.com), Chorus Systems (www.sun.com), Accelerated Technology (www.atinucleus.com), Microtec (www.microtec.com), JMI (www.jmi.com), Integrated Systems (www.isi.com), and Lynx (www.lynx.com) for RTOS support. These vendors also supply system calls and library routines, many device drivers, and network protocols. Cygnus (www.cygnus.com), Wind River, and Green Hills Software (www.ghs.com) development environments also support SPARClite. Orion Instruments (www.yokogawa.com) in-circuit emulators support SPARClite-based system development. US Software (www.ussw.com) and Log Point (www.logpoint.com) offer floating-point libraries for SPARClite.

Second sources: There are no second sources for SPARClite.

Hitachi SuperH Series

The SuperH Series comprises the SH-1, SH-2, SH-3, and SH-4 series of RISC µPs, µCs, and ASIC cores. The SH-1, -2, and -3 employ a fetch, decode, execute, memory-access, and write-back-to-register pipeline. Hitachi built the devices around 25 32-bit registers that you access using load/store instructions. These registers comprise 16 general registers (the SH-3 has eight 32-bit shadow registers for context switching), five control registers, and four system registers. Depending on the chip, the interrupt latency can be as low as seven clock cycles. The chips use 32-bit datapaths to internally move data, but all versions use a flexible external bus width. The SuperH family also has devices with single-cycle mask ROM and one-time-programmable and flash memory with densities as high as 256 kbytes, unlike most RISC families.

Although devices in the SH series have a similar core, significant differences exist. The major differences between SH-1 and SH-2 are that the SH-2 features on-chip cache memory, higher speeds, and a 32×32-bit multiply-accumulate (MAC) unit. (The SH-1's MAC unit is 16×16 bits.) To build the SH-3, Hitachi added to the SH-2 a memory-management unit (MMU), a barrel shifter, and the ability for conditional-branch instructions to enable or disable the pipeline's delay slot. Disabling the delay slot, although decreasing performance, allows the processor to run more deterministically and reduces the effects of pipeline flushes.

The 200-MHz, two-way-superscalar SH-4 µP includes a 3-D graphics accelerator that Hitachi claims can perform at 1.2 Gflops. This µP has four 32×32-bit multipliers fed by two 128-bit buses; it also has four adders. You can load the multipliers with eight operands in one cycle; the µP then adds the results in the next cycle. This hardware performs rotations and transformations on 32-bit, single-precision, floating-point vectors.

SuperH processors use a 16-bit instruction word to achieve compact code. The instruction width limits the number of basic operation codes, handles only 16 general registers, and addresses only two operands. Additionally, only 12 bits are available for an immediate offset; jumps with immediate data must be in 2048-byte hops. However, the SH-3 supports FAR-relative branches to support position-independent code. Although these restrictions lead to more instructions per task, the overall result is significantly smaller code.

The SH-1 µPs can operate from external memory or from on-chip program memory at a CPU frequency of 20 MHz. The 16-bit-wide external-memory bus can supply the CPU with instructions from SRAM or fast DRAM on each cycle. If the processor is operating from external memory, each data access to external memory may take an additional one to two cycles.

Instead of on-chip program memory, the SH-2 and SH-3 have a four-way, set-associative on-chip cache (4 kbytes for the SH-2 and 8 kbytes for the SH-3), a 32-bit-wide memory bus for CPU-memory bandwidth as high as 60 MHz with a synchronous-DRAM interface), and a 32-bit divide unit (replacing the first chip's bit-step-divide function on the SH-2). You can reconfigure the cache as a two-way, set-associative cache and 2 kbytes (SH-2) or 4 kbytes (SH-3) of user-configurable RAM. The external-memory bus supports multiprocessing; it has bus arbitration for multiple masters. The SH-3 also has a unique RTOS feature: If a task or thread crashes, the operating system can gracefully recover and not have the errant task corrupt other tasks or RTOS environments.

Power management: Sleep mode discontinues CPU processing but keeps peripherals active. Standby stops everything but maintains register and cache contents. The SH-2 and -3 provide several clock modes for reducing power; software can adjust the clock rate during program operation. The SH-3's unified cache has a special low-power design that dissipates only 100 mW in operation. The cache sense amps are energized for the cache set that hits while the other three sets stay switched off. The sense amps respond to only a 60-mV differential versus the full 3.3V swing.

Special instructions: A 16×16-bit MAC instruction (42-bit accumulator) in the SH-1 and a 32×32-bit MAC instruction (64-bit accumulator) in the SH-2 and SH-3 provide a fast DSP function. Although Hitachi classifies the architecture as load/store, some instructions reference memory. Delayed branch instructions minimize pipeline disruption. An instruction swaps upper and lower bytes. The SH-4 includes a set of 3-D, floating-point instructions. The SH-DSP, a version of the SH-2, supports 23 32-bit DSP instructions for zero-overhead looping and modulo-addressing support.

Special on-chip peripherals: The SH-DSP contains a DSP as an "on-chip peripheral." This DSP unit shares the five-stage pipeline with the integer unit; the DSP is not a coprocessor. The CPU contains a fetch-and-decode unit, which manages the instruction stream for both the integer and DSP units, routing instructions to the appropriate unit (see *EDN*'s 1998 "DSP-architecture directory," April 23, 1998, pg 54). Other, more conventional peripherals include memory controllers, a real-time clock, smart-card and serial codec interfaces, IrDA support, a floating-point-unit coprocessor, a hardware division unit, complex multifunction timers, a PCMCIA interface, and an LCD controller.

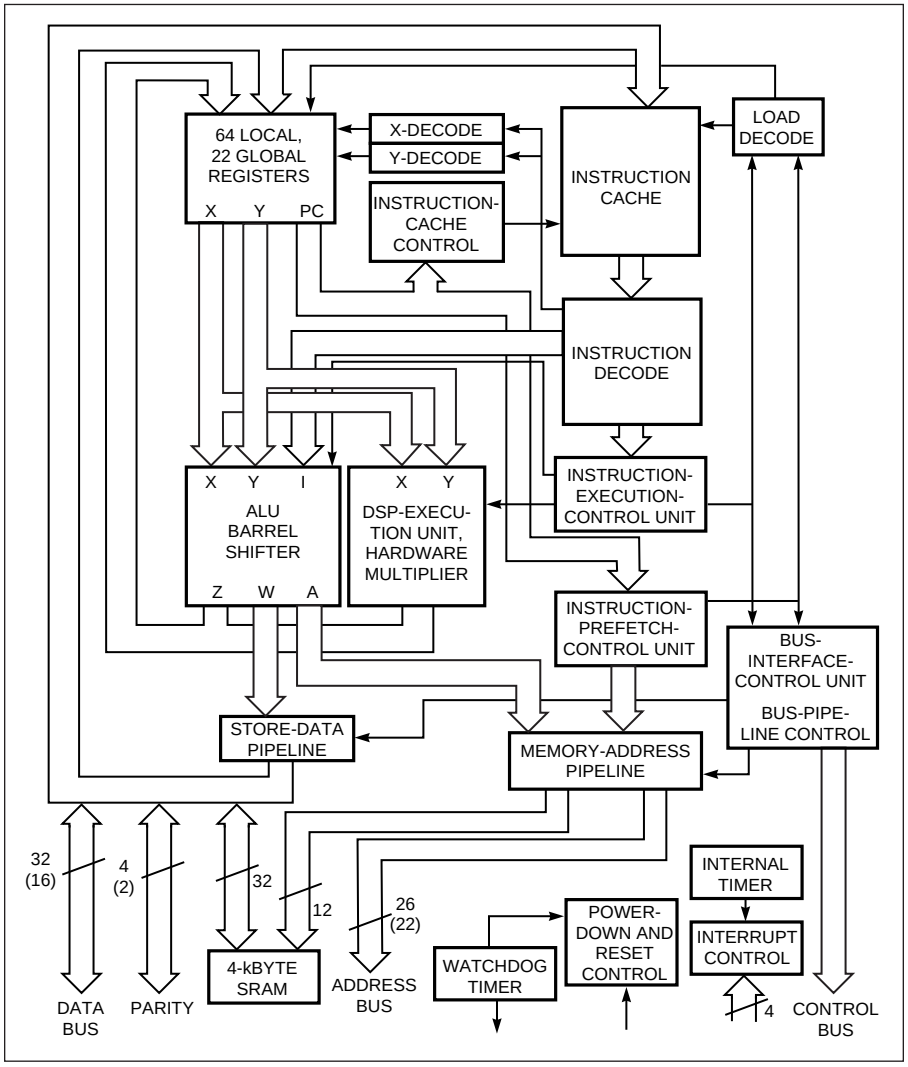
The SH-3 contains an MMU with a 128-entry translation-look-aside buffer (TLB). The TLB caches virtual-to-physical-address translations from user-created page tables to external memory, providing both data protection and virtual memory. Address translation employs a paging system that supports 1- or 4-kbyte pages. The MMU also handles multitasking by providing multiple virtual-memory modes. Thus, each process has its own virtual memory and cannot access the resources of another process or the OS kernel.

Development tools: Hitachi and a number of third-party vendors offer development-tool support for the SuperH. Hitachi, Green Hills Software (www.ghs.com), and Cygnus (www.cygnus.com) provide C and C++ compilers. Hitachi, HP (www.hp.com), Orion Instruments (www.yokogawa.com), and Sophia Systems (www.sophia.com) offer in-circuit emulators. Wind River (www.windriver.com), Accelerated Technology Inc (www.atinucleus.com), and Microsoft (www.microsoft.com) provide RTOSs. Other tools include assemblers, ROM emulators, integrated Windows-based development environments, debuggers, floating-point libraries, and networking libraries. Hitachi supports Windows CE development with the $10,000 D9000, a reconfigurable development platform.

Second sources: Seiko-Epson (www.seiko.com), VLSI, ST-Microelectronics, and Sony (www.sel.sony.com) are licensees.

See *EDN*'s Web-site version, www.ednmag.com, for block diagram.

hyperstone E1-32X



The hyperstone E1-32X combines RISC and DSP technology in one core. The E1-32X has a load/store architecture built around a register set that includes 64 general-purpose local and 22 global registers. Local registers are organized into a 64-word, circular register stack to hold function/subroutine stack frames. The stack is organized into frames of as many as 16 words; the E1-32X keeps current frames on chip and automatically pushes the frame to off-chip memory as the register stack fills and moves frames back on chip as the register stack empties. For fast parameter passing, the current stack frame can overlap the previous one with a variable range. Instructions are 16, 32, or 48 bits wide. The variable-length instructions, which the E1-32X automatically prefetches, provide constants and native addresses as large as 32 bits.

The 4-Gbyte address space divides into four blocks; you can configure each block individually for bus width and timing. You can use these blocks for glueless connection of DRAM, extended-data-out DRAM, SRAM, EPROM, or other memory devices, each with its own timing and bus width. A separate I/O-address space also allows each I/O device to have its own timing.

The integrated DSP unit, working in parallel with the ALU and the load/store unit, can perform DSP calculations while the ALU is performing loop counts, address calculations, or load-and-store operations. The ALU executes its instructions during the latency cycles of DSP instructions. The DSP unit shares all the E1-32X's functional blocks, including the register set; however, it provides dedicated result registers and 32- and 64-bit hardware accumulators. The DSP unit supports 16- and 32-bit data types.

Power management: In automatic power-down mode, only the interrupt logic, clock, and DRAM-refresh logic remain active. Sleep mode also disables DRAM refresh. At 3.3V, current consumption in power-down and sleep modes is less than 2.5 mA and 100 µA, respectively.

Special instructions: DSP instructions include multiply, complex and real multiply-accumulate, multiply-subtract, and complex addition/subtraction. Other special instructions include test-leading zeros.

Special on-chip peripherals: Hyperstone's E1-32X contains a DRAM controller that allows you to program page size, refresh rate, timing, and access parameters with an internal-memory register. The controller supports fast-page-mode and extended-data-out DRAMs. The µP also contains a single-cycle-access, 8-kbyte memory, and an I/O- and peripheral-interface controller. You can use this controller to set the width and timing of the µP's address areas. An integrated PLL allows you to multiply the external clock by a factor as large as 4.

Development tools: The vendor offers a development starter kit, a PC-based development board, and the hyICE serial connector for stand-alone operation. The company also provides an ANSI C compiler and DSP library, a source- and task-level debugger, a multitasking real-time kernel, an assembler, a linker, a library manager, and a profiler. Eonic Systems (www.eonic.com) and Etnoteam (www.etnoteam.com) provide RTOS support. Visual Tools (www.etnoteam.it) offers a JPEG embedded-image-compression/decompression library for the hyperstone E1-32X µP. The library supports user-defined subsampling for image quality and compression to the desired size. Hyperstone provides speech compression/decompression algorithms (G.726, G.729, G.723.1, GSM 06.10) and a complete modem. GAO Research (www.gaoresearch.com) offers V.22 modem code.

Second sources: LG Semicon (www.lgsemicon.co.kr) is a licensee.

IBM/Motorola PowerPC

Serving as a base for a family of RISC chips, the PowerPC derives its core architecture from the performance-optimized-with-enhanced-RISC (POWER) architecture. The instruction set and 32 32-bit, general-purpose registers support multiple microarchitecture implementations that include the 32-bit 603e, 604e, 740, 750, and embedded processors (Motorola's MPC 50x, MPC8x0, MPC82x, and IBM's 400 series).

The PowerPC 750 contains seven parallel-operating execution units: two integer units, a branch-processing unit, a load/store unit (LSU), a floating-point unit (FPU), a condition-register unit, and an L2-cache-interface unit. (The 740 is the lower cost version of the 750 and lacks the L2-cache-interface unit.) This CPU can fetch as many as four instructions per cycle. The 750 processes branches as they enter the instruction buffer and can decode and dispatch two nonbranches in one cycle. Completion logic keeps track of the outstanding instructions and retires them in order.

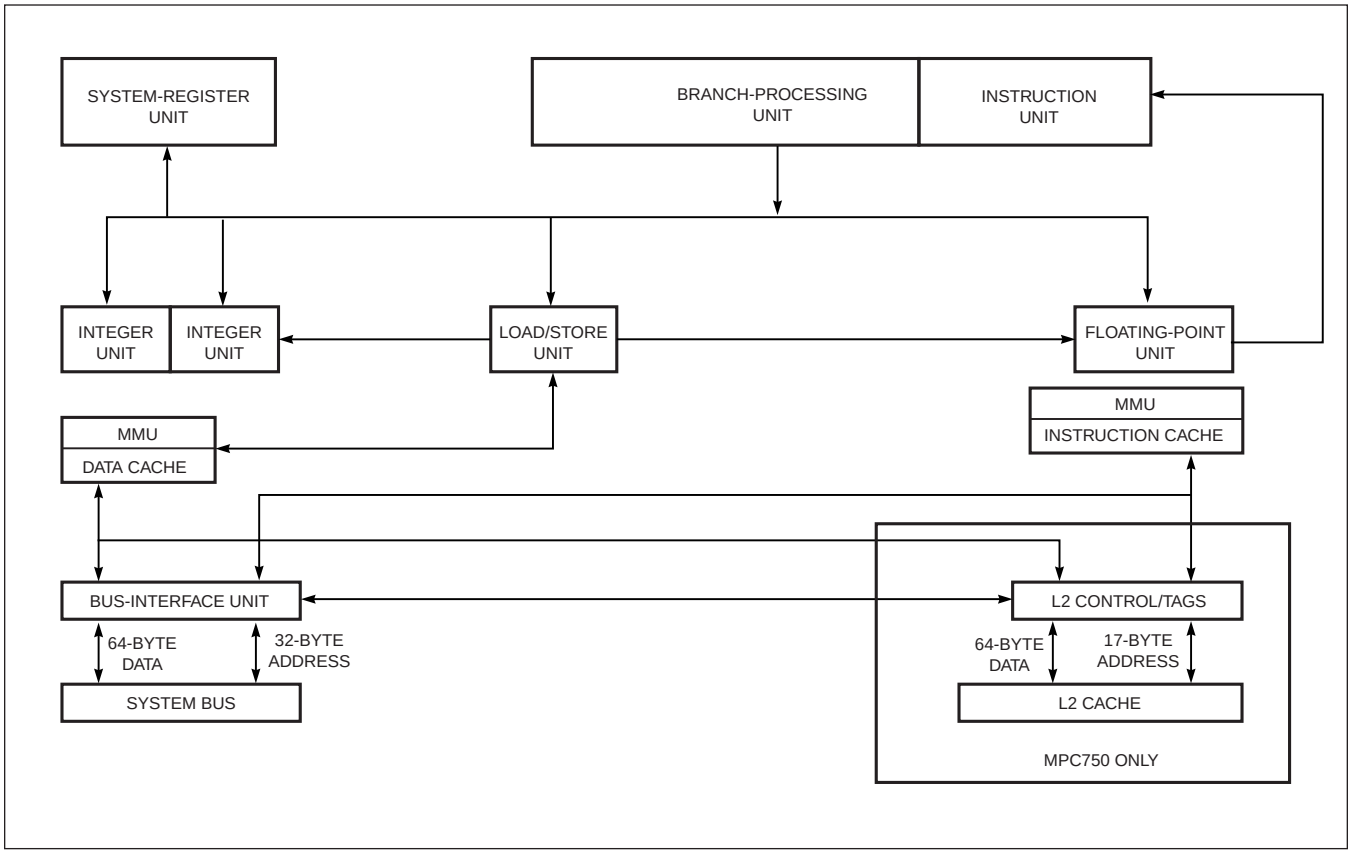
The PowerPC 750 mP uses static or dynamic branch prediction to improve the accuracy of instruction prefetching. For static prediction, the branch-operation codes provide hints to predict whether a branch is taken or not. For dynamic prediction, the CPU uses a 512-entry branch-history table and a 64-entry branch-target instruction. The CPU permits speculative execution down a predicted path beyond one unresolved branch.

The 750 has separate 32-kbyte instruction and data caches. Both eight-way, set-associative, lockable caches provide byte-level parity checking. A locked cache typically supplies data on a hit, but cache lines are not replaced on a miss. The 750 contains an on-chip L2-cache controller and backside L2 bus, which improves system performance by reducing system-bus traffic. The L2-cache controller includes 8196 tag entries, which support 256 kbytes, 512 kbytes, or 1 Mbyte of external, two-way, set-associative, unified L2 cache. The L2 cache uses standard, commodity SRAMs. The nonblocking L2 cache supports hit-under-miss mode and can simultaneously service as many as four requests. The L2-cache bus can operate at various speeds relative to the processor frequency.

The PowerPC 604e contains seven independent execution units: two single-cycle integer units, a multiple-cycle integer unit, a branch-processing unit, an LSU, an FPU, and a condition-register unit. Instructions execute out of order, and execution results can be immediately available to subsequent instructions through the use of rename registers. The completion unit commits, or "retires," results to floating-point or general-purpose registers. The unit retires as many as four instructions per clock cycle in order, ensuring a precise exception model.

The PowerPC 604e mP uses dynamic branch prediction to improve the accuracy of instruction prefetching. This feature and the ability to speculatively execute through two unresolved branches minimize pipeline stalls. The 604e has separate 32-kbyte, four-way, set-associative instruction and data caches, both of which provide byte-level parity checking.

(continued on pg 142)



The 604e and 750 have separate memory-management units (MMUs) for instructions and data. The MMUs support as many as 4 petabytes of virtual memory and 4 Gbytes of physical memory. Access privileges and memory protection are controlled on 128-kbyte to 256-Mbyte blocks and 4-kbyte pages. Translation-look-aside buffers (TLBs) with 128 entries efficiently translate addresses by storing the most recently used page translations.

The 604e and 750 support 64-bit data and 32-bit address buses. The interface protocol allows multiple masters to access system resources through a central arbiter. The PowerPC 604e works in multiprocessor systems and snooping tasks and requires no additional bus cycles. The 604e's on-chip snooping logic maintains cache coherency in multiprocessor systems. The 750 supports snooping but is optimized for uniprocessor systems. It supports no data sharing among caches in different processors. The buses on the 604e and 750 are compatible electrically and in the protocol they use. A common chip set supports both processors.

The 603e comprises five parallel execution units: integer execution, floating point, branch, system, and load/store. With a four-stage pipeline—fetch, dispatch, execute, and complete—the 603 can achieve three instructions per clock cycle. During the fetch stage, the 603 uses a six-instruction prefetch queue to hold pending instructions. Unlike other PowerPC derivatives, the 603 supports only static branch prediction. However, the architecture supports out-of-order execution and in-order retirement, similar to other PowerPC devices.

The embedded PowerPC processors include IBM's 400 series and Motorola's MPC500 and MCP800 families and devices. Compared with other PowerPC devices, these devices have similar—but fewer—execution units. IBM's 403Gx embedded controllers have a five-stage pipeline and can dispatch as many as two instructions per cycle. These devices implement static branch prediction and branch folding and have a four-instruction prefetch queue. Integrated caches of varying sizes are two-way set-associative and are implemented as fetch-through instruction caches and write-back data caches. (The 403GCX data cache does not provide write-through operation.) The 403Gx processors do not provide hardware support for maintaining cache coherency during DMA and external bus-master operations or in a multiprocessor configuration.

The PowerPC 403GC and 403GCX include an MMU featuring a fully associative TLB. Each entry provides translation for a memory page, which can be one of several sizes for efficient system-memory use. Memory components attach directly to the 403 devices with a programmable-memory interface on the processor's bus-interface unit. The DRAM controller includes the address multiplexer, eliminating the need for an external address multiplexer. The DRAM controller supports external bus masters. You can use software programming to tune the timing for the interface control signals.

IBM/Motorola PowerPC (continued)

The PowerPC 401GF implements a three-stage pipeline and supports hardware multiply and divide and unaligned loads and stores. The CPU uses operand forwarding and static branch prediction to increase performance. The 401GF's cache controllers implement critical data forwarding, fill-first handling of cache misses, and nonblocking flush operations.

Motorola's MPC500 and MPC800 families, although targeting different applications, have the same basic CPU architecture. (However, the new MPC8260 PowerQUICC II is an upgrade of the MPC860 and contains a PowerPC EC603e core.) Both families integrate a fixed-point unit (FXU), an LSU, two register files, and a sequencer unit; the MPC500 family also adds an FPU. The FPU includes single- and double-precision multiply-add instructions. The sequencer unit contains a branch processor featuring static branch prediction and branch-folding capability during execution (zero-cycle branch execution time) and runtime reordering of loads and stores.

The MPC500 and MPC800 devices use an InterModule Bus, developed for Motorola's 683xx devices, as a backplane to connect all system modules. Both families include a system-integration unit (SIU) that enables simple integration with external memories, other CPUs, and peripherals. The SIU for the MPC505 and MPC509 differs from the one in the 800 family devices and in the MPC555. The 505 and 509 SIUs have separate data and instruction buses; the 800 and 555 devices combine these buses outside the SIU. The 800 family has both instruction and data caches and an MMU. The caches are two-way set-associative and feature lockability on a line.

Special instructions: Motorola has expanded the PowerPC architecture with its AltiVec technology—162 new instructions along with a 128-bit vector-execution unit that performs single-instruction multiple-data operations concurrently with the integer units and FPUs. AltiVec supports 16-way parallelism for 8-bit integers and characters, eight-way parallelism for 16-bit integers, and four-way parallelism for 32-bit integers and IEEE floating-point numbers. AltiVec also includes a separate register file with 32 128-bit-wide registers.

Development tools: The PowerPC families have a large third-party tool-supplier base. IBM also offers development tools for all its PowerPC embedded processors. These tools include a C/C++ compiler; a RISCWatch debugger with in-circuit emulation; a ROM monitor; RTOS-aware debugging; and real-time, noninvasive trace capability.

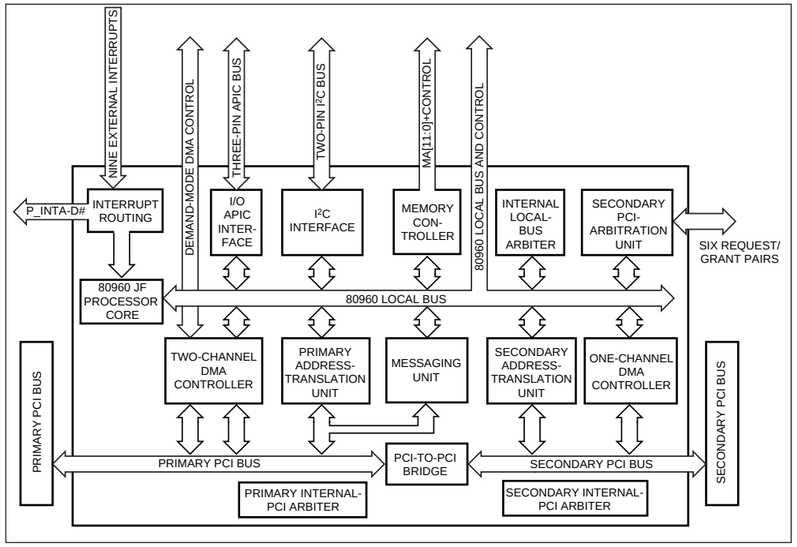
Second sources: Mitsubishi is a second source for IBM's embedded PowerPC MPs.

Intel i960

The range of i960s runs from the new superscalar HA/HD/HT to the 16-bit SA/SB variants, including low-power versions of the i960 Jx series that operate at 3.3V. The i960 combines a von Neumann architecture with a load/store architecture that centers on a core of 32 32-bit general-purpose registers comprising 16 local and 16 global registers. An on-chip register cache automatically caches the local register sets to speed context switching. If the cache is full, the oldest cached set moves to memory, and the latest set caches. All i960s have multistage pipelines and use resource "scoreboarding" to track resource usage.

The i960CA provides superscalar operation and five pipeline stages. The key to the Cx is its four-instruction-wide instruction decoder, which decodes as many as four instructions per cycle. Current implementations dispatch as many as three of these instructions for execution. The i960CF has 128-bit-wide buses to move instructions to the decoder and 128-bit-wide buses to move data between the cache and registers.

Intel built the superscalar i960s around a six-port register file with register or memory-control execution units. These units include an integer unit, a floating-point unit, and an interrupt-control unit on the register side and address-generation and bus-controller units on the memory side. The i960s can cache instructions in a lockable cache; later versions add an instruction cache to supplement the register cache.



Intel based the i960Rx series I/O processors on the i960 Jx series processor core. The i960Rx processors target server-motherboard and adapter-card applications, in which the processors create an "intelligent" I/O subsystem. Intel and others have developed an intelligent I/O (I^2O) specification to speed I/O processing and simplify driver development.

Special instructions: The i960 family has uninterruptible atomic add and modify instructions. Other instructions flush local registers and provide cache-locking control.

Development tools: More than 70 vendors support the i960 with a robust tool suite. These vendors offer a range of compilers, emulators, evaluation boards, debugging monitors, and real-time operating systems for the i960 family.

MIPS R3000

MIPS (www.mips.com) built the MIPS R3000 processors around a set of 32-bit, general-purpose registers in a central register file. To minimize control logic and improve speed, the instruction set has only 73 instructions, and addressing options are few. The chip has a three-address load/store architecture. Similarly, instructions are one 32-bit word to minimize decoding and speed processing. To reduce code size, MIPS and LSI Logic codesigned the MIPS16 application-specific extension. MIPS16 comprises new 16-bit instructions with a corresponding decoding block that the MIPS mP core integrates. Although most applications still need to run 32-bit code (MIPS16 supports a mixture of 32- and 16-bit code), MIPS claims that MIPS16 provides an overall memory savings as large as 40%. LSI Logic, with its TinyRISC TR4101, is the first MIPS licensee to implement the MIPS16 instruction extensions.

MIPS engineers implemented a five-stage pipeline: instruction fetch, read operand and decode instruction, execute, access data memory, and write back results for the R3000. The pipeline lets as many as five instructions execute concurrently—each at a different stage of its instruction cycle. A branch-delay slot minimizes branch effects. The compiler fills the instruction slot, following the branch with a no-operation instruction or an instruction from the current thread that can execute before the branch takes effect. Toshiba's R3900 and Integrated Device Technology's (IDT) RISCore32300, R3000 derivatives, incorporate register "scoreboarding" to enable nonblocking loads and avoid pipeline stalls when there are no data dependencies in subsequent instructions. This feature has a significant benefit in communications applications: It allows programmers to hide main-memory latencies during routing or packet processing. On IDT's 32300, you can also use the nonblocking load for cache prefetch and for performing DMA transfers without performing invalidates and write-backs. IDT implemented this feature as a new hint, called "ignore hint." This feature helps you get around the MIPS instruction-set architecture's lack of "move-multiple" operations. The 32300 also supports a mechanism to minimize pipeline stalls; in the event of a cache miss, the first entering word goes directly to the pipeline.

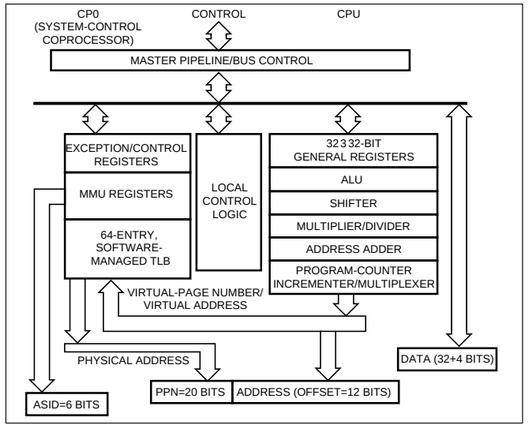
To improve the multiply and divide performance of the standard R3000, IDT built in a dedicated integer multiply/divide unit. In the MIPS instruction-set architecture, multiply and divide use special destination registers, permitting only one multiply at a time. IDT enhances this capability with a three-operand multiply, whereby the operand results go directly to a general register. This feature supports DSP capability and performs atomic multiply adds and multiply subtracts. It also implements count-leading ones and zeros operations. The multiply-add throughput is one cycle faster than the data latency, so if you use two distinct operands, the operation becomes load-bound. Whereas the general MIPS mechanism supports reset, cache/parity error, user translation-look-aside-buffer (TLB) miss, and general interrupts, IDT's 32300 lets you define separate interrupts to support software compatibility with your legacy code.

The standard R3000 memory-management unit includes a fully associative, 64-entry TLB that translates virtual addresses to 32-bit physical addresses. (Note: Not all R3000 derivatives contain the TLB.) The mP uses a write-through cache policy. A small on-chip FIFO buffer enables the CPU to perform instruction "streaming"—refilling the cache and executing instructions even while reading additional instructions from memory.

Special instructions: The R3000 implements the MIPS-I instruction set. IDT's 32300 uses the MIPS II instruction-set architecture but includes some MIPS-IV functions. It implements those MIPS-IV instructions, such as prefetch operations and conditional moves, that are independent of operand size. The 32300 also supports both big- and little-endian data types. Several of the MIPS derivatives add a multiply-accumulate (MAC) instruction. LSI is the first MIPS licensee to implement MIPS16 instruction extensions on the TinyRISC TR4101. Toshiba's TX19 also uses the MIPS16 instruction extensions. (See R4xxx, pg 169, for more details.)

Special on-chip peripherals: Philips offers the TwoChipPIC, which combines the UCB1200 that interfaces with the company's PR31700 MIPS mP. The TwoChipPIC provides a microsystem on a chip for handheld devices. Integrated modules include a MAC unit, an LCD controller, an infrared controller, PCMCIA-card support, touchscreen control, and audio in/out. Toshiba's peripherals include a graphics controller, a PCI controller, and support for Microsoft's (www.microsoft.com)Windows CE.

Development tools: A range of third-party development tools is available for the MIPS RISC architecture. Detailed information is available in the MIPS RISC Resource Catalog from MIPS Technologies Inc or at www.mips.com. Philips supplies the hardware-abstraction layer, device drivers, a reference design, and a development board for Windows CE implementation on the TwoChipPIC. Microsoft's Visual C++ tool chain supports TwoChipPIC development.



LSI Logic offers evaluation boards and kits for its line of TinyRISC and MiniRISC mPs and cores. For example, the company's BDMR4101 evaluation board uses an 81-MHz TR4101 CPU core and features 1 Mbyte of SRAM and an 8-Mbyte plug-in DRAM single-inline-memory module, 512 kbytes of flash, a full-duplex serial port, SCN2681 dual UART with dual RS-232C ports, the DP83934 Sonic Ethernet controller with a 10BaseT interface, and the SerialICE debugging monitor and software in EPROM. It supports both PC and Unix host environments. LSI offers a number of tools, including the MiniSIM and TinySIM architectural simulators for system-on-chip embedded applications, as well as a system-verification environment for silicon-design verification. LSI Logic also provides application-specific evaluation boards, such as the Integra for set-top-box development and the ATMIzer II for communication-product development.

IDT's 33-MHz 79S381 evaluation board allows you to evaluate the 3041, 3052, and 3081 mPs. The board features 2 Mbytes of interleaved DRAM, expandable to 16 Mbytes; 256 kbytes of zero-wait-state SRAM; 512 kbytes of EPROM, expandable to 2 Mbytes; and a 1024-bit serial EEPROM. The company provides the 79S361 evaluation platform for the 79R36100. This board has 1 Mbyte of noninterleaved, zero-wait-state DRAM, expandable to 64 Mbytes. It also contains 2 Mbytes of EPROM and a slot for 1 Mbyte of zero-wait-state SRAM.

IDT offers its kernel-integration tool that includes source- and object-code versions of common routines for CPU design. The company also offers a system-integration monitor that is a ROMable debugging kernel. The monitor includes IDT's micromonitor, which requires only a UART and ROM to perform the initial debugging and integration of new hardware. IDT/C is an ANSI C-compliant Gnu compiler, assembler, linker, and librarian. It includes start-up code, cache, and exception routines.

Toshiba offers evaluation boards for its TX39 products. These boards feature support for serial, SCSI-II, Ethernet, or VMEbus interfaces. Wind River's (www.windriver.com) VxWorks and Tornado RTOS support these boards. Toshiba also offers the TMPR3912 and TMPR3922 reference development systems that support the Microsoft Window CE operating system.

Second sources: MIPS licenses the R3xxx processors to IDT, LSI Logic, NKK, Philips, and Toshiba.

Mitsubishi M32Rx/D

The Mitsubishi M32Rx/D contains a 32-bit RISC CPU; as much as 4 Mbytes of on-chip DRAM, which Mitsubishi calls "eRAM"; a 32 3 16-bit multiply-accumulate (MAC) unit; and a bus-interface unit (BIU). A 128-bit, 66-MHz internal bus connects the CPU, DRAM, cache, and BIU. The M32Rx/D's circuitry automatically refreshes the internal DRAM.

The M32Rx/D family comprises the M32R/D and the new superscalar M32Rx/D architectures. Both architectures are instruction-set-compatible and comprise a combination of 16- and 32-bit-wide instruction formats with six addressing modes. The devices include 16 32-bit, general-purpose registers and two 56-bit accumulators.

The M32R/D CPU executes most instructions in one clock cycle, using an instruction-fetch, decode, execute, memory-access, and write-back pipeline. The decode stage dispatches instructions in order, and the remaining stages execute them out of order to hide memory-access latency. The MAC unit contains a single-cycle, 32 3 16-bit multiplier and a 56-bit adder.

The M32Rx/D contains a dual-issue, six-stage pipeline and performs out-of-order execution; it can execute two 16-bit instructions in parallel. The pipelines are asymmetrical, and instructions have to align properly to keep the pipes full. For example, both pipelines can execute arithmetic and logical operations, but only Pipeline 1 can execute load/store and jump/branch instructions. Additionally, only Pipeline 2 can execute MAC instructions.

Both CPUs have an instruction queue of two 128-bit entries. The cache maps directly to the address space and has caching modes for internal instruction and data, for internal and external instructions, and for cache off. If a cache miss occurs, the CPU fetches one 128-bit data line in five cycles. The BIU has 128-bit data buffers and supports burst transfers on 128-bit boundary data.

A 16.67-MHz bus clock and four digital PLLs generate the internal 66-MHz clock. The PLL contains a digital frequency multiplier. Four cascaded, 64-tap inverter chains generate four timing edges in one-half of a clock cycle. A phase detector and an up/down counter adjust the pulse width to one-fourth of the one-half clock cycle to keep the duty cycle of the four-times clock at 50%. The generated clock then feeds into a digital phase shifter to reduce the phase difference between the external and internal clocks to 400 psec.

Power management: The M32Rx/D supports sleep and standby modes, during which the average power consumption is 170 and 2 mW for the two modes, respectively, for the 2-Mbyte version. In the sleep mode, the CPU and caches stop; in standby mode, only the DRAM is clocked.

Special instructions: The M32Rx/D supports MACs of 32 3 16 and 16 3 16 bits. It also performs data rounding in the accumulator and block moves. The M32R/D and M32Rx/D support 83 and 95 instructions, respectively. The M32Rx/D's additional instructions include five DSP-function instructions for MAC and rounding operations.

Special off-chip peripherals: Mitsubishi's M65439FP peripheral-function "super-I/O" chip performs such functions as M32000D bus control, DRAM control supporting two banks and page-mode burst transfers, and chip-select control for as many as five 64-kbyte to 4-Mbyte blocks with one to eight wait states. The peripheral I/O ASIC also contains a two-channel DMA controller that can transfer as much as 2 Mbytes using cycle-steal, single-transfer, continuous-burst-transfer mode or cycle-steal, continuous-transfer mode. An interrupt controller handles 20 sources with priority resolution for as many as seven levels. Other functions of the M65439FP include timers, a two-channel UART, and a two-slot IC-card controller. The device sells for $8 (10,000).

Development tools: Cygnus (www.cygnus.com) supplies C and C++ compilers and debuggers for the M32Rx/D. Mitsubishi also supplies a C compiler and debugger, an evaluation board, and an in-circuit emulator. Wind River (www.windriver.com) supplies the Tornado development environment, which includes the VxWorks RTOS. Integrated Systems (www.isi.com) supplies the Prism+ development environment, which includes the pSOS RTOS, Diab Data's (www.diabdata.com) C compiler and Light Source debugger.

Second sources: There are no second sources for the M32R/D or M32Rx/D.

See *EDN* Web-site version, www.ednmag.com, for block diagram.

Motorola ColdFire

Motorola's ColdFire, or VL-RISC (variable-length RISC), architecture evolved from the M68000. VL instructions help to attain higher code density. Also, the ColdFire architecture eliminates M68000 instructions, which embedded applications rarely use, and optimizes the pipeline. As a result, it has fewer transistors—approximately 55,000—than the M68000. ColdFire continues to use the M68000 programmer's model.

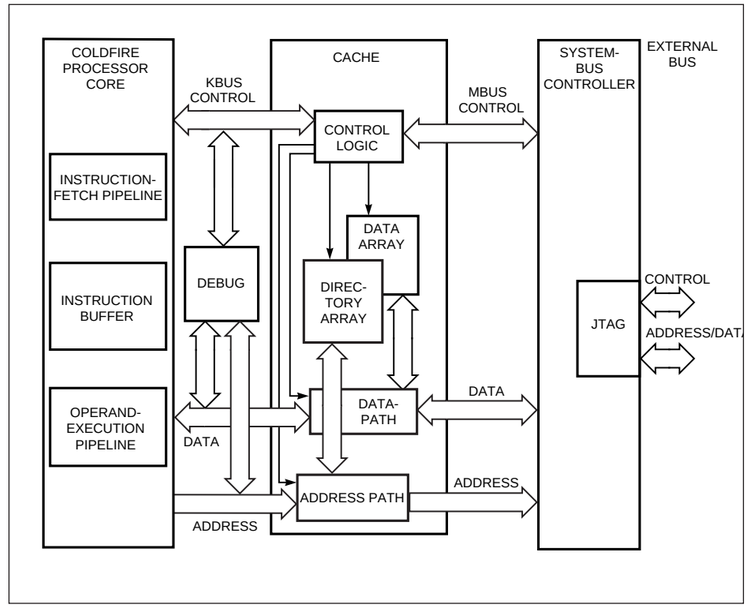
Motorola designed three versions of ColdFire. The first version was relatively short-lived. The version 2 and 3 architectures comprise two decoupled subpipelines: an instruction-fetch pipeline (IFP) and an operand-execution pipeline (OEP). An instruction buffer separates the two pipelines and minimizes pipeline stalls. Motorola's design goals for V2 included making the core as small as possible. Hence, V2's IFP is a simple two-stage pipeline: instruction-address generation and instruction fetch.

Design goals for V3 were to stretch the pipeline to allow the device to operate at clock frequencies of 90 MHz and

Motorola ColdFire (continued)

higher. For starters, the V3 core includes multiple clock domains that allow it to operate at a higher frequency than the remainder of the mP. V3's IFP includes two additional stages to help pipeline the address-generation and instruction-fetch phases. One of the additional stages is an instruction early decode to help reduce decoding time. This concept, borrowed from the 68060, includes some branch-acceleration techniques. For example, the early-decode mechanism considers that backward branches are taken. By default, the mechanism considers forward branches taken, but a condition-code register bit allows you to set up forward branches to be not taken.

In the V2 implementation, the instruction buffer comprises a three-long-word-entry FIFO buffer. The V3 implementation holds three complete instructions, regardless of length. This instruction buffer essentially converts variable- into fixed-length instructions.



A modular, standard bus architecture separates the CPU core from on-chip peripherals. The core communicates with on-chip memories using the tightly coupled Kbus processor. This bus lets the core perform a 32-bit fetch from internal memory in one clock cycle by pipelining the address and data. A controller interface on the Kbus indirectly attaches the core to user-selectable cache, ROM, and RAM modules. V3 uses a two-stage, but pipelined, Kbus that adds one cycle to most operand-read accesses; however, the increased operating frequency offsets the extra cycle. Another bus, the Mbus (master bus), offers centralized arbitration. A special module connects the Mbus to the Kbus. The Sbus (slave bus) interfaces to standard on- and off-chip peripherals and attaches to the Mbus through a system-bus controller.

On-chip debugging supports real-time trace; real-time and non-real-time debugging; and access to control registers to define types of memory regions, such as cacheable copy-back, write-through, and noncacheable. Real-time trace reflects the processor's status and indicates events such as instruction completion and monitoring change-of-flow target addresses. Real-time debugging supports program-counter-relative, operand-address, operand-data, and non-real-time-debugging hardware breakpoints. Non-real-time debugging is similar to background-debugging mode on current 683xx products. You can use a three-pin serial interface in this mode to read register contents, generate an infinite-priority interrupt, and force the CPU to halt.

Power management: A low-power-stop (LPSTOP) instruction shuts down active circuits in the processor and halts instruction execution. Processing resumes via a reset or valid interrupt.

Special instructions: ColdFire added 32 3 32-bit integer-multiply, register-sign-extension, and multiword nonoperation instructions to the 68000 architecture. Compilers use nonoperation instructions to remove branch instructions.

Special on-chip peripherals: The MCF5200M processor, which Motorola designed with its FlexCore methodology, integrates the ColdFire core, debugging module, and misalignment module with a multiply-accumulate (MAC) unit supporting 16- and 32-bit operations. The MCF5202 supports a 32-bit multiplexed bus with dynamic bus sizing that allows access to 8-, 16-, or 32-bit memory and peripherals. It also has a debugging module that provides serial control and visibility of the processor and memory system. Motorola offers the ColdFire2 and ColdFire2M in the FlexCore library for customer design. Both devices integrate the ColdFire core with a debugging module; a misalignment module; and memory controllers that support as much as 32 kbytes each of RAM, ROM, and instruction cache. The ColdFire2M also incorporates the MAC unit.

Development tools: Third-party tools for the ColdFire family include in-circuit emulators from Embedded Support Tools (www.estc.com), Lauterbach (www.lauterbach.com), Microtec International (www.microtec.com), Noral Micrologics (www.noral.com), and Orion (www.yokogawa.com). Cygnus (www.cygnus.com), Diab Data (www.diabdata.com), and Software Development Systems (www.sds.com) offer C compilers. Wind River (www.windriver.com), Integrated Systems Inc (www.isi.com), Embedded System Products (www.esphou.com) offer ColdFire RTOS products. Hewlett-Packard (www.hp.com) offers preprocessor support.

Second sources: There are no second sources for ColdFire.

Motorola MCore

MCore uses a four-stage pipeline to execute two-thirds of its 95 basic instructions in one clock. Similar to Hitachi's 32-bit SuperH architecture, MCore uses 16-bit, fixed-length instructions and a register file with 16 general-purpose registers and an alternative register file for context switches. Unlike SuperH, which shadows only eight of its general-purpose registers, MCore shadows all 16. MCore implements only 86% of its operation-code space, allowing some instruction head room for future generations. The 16-bit instruction size forces MCore to use two operand instructions—4 bits per register. Motorola claims that the lack of three operand instructions causes a 2 to 7% code bloat in key applications.

The architecture supports 8-, 16-, and 32-bit data types, although misaligned data accesses force a misaligned data exception. You can mask the misaligned data exception using a control bit in the process-status register. Although you can't access MCore's registers as bytes and words, instructions can sign-extend non-32-bit data types. A barrel shifter shifts as many as 32 bits in one cycle. The architecture also contains a find-first-one unit and result-feed-forward hardware. The feed-forward hardware allows a subsequent operation to use a result while the CPU writes the result back into the register file. The first version of the MCore architecture offers limited support for hardware multiply and divide; it uses a 2-bit per clock, overlapped-scan, modified Booth algorithm with early-out capability to reduce execution time for operations with small multipliers.

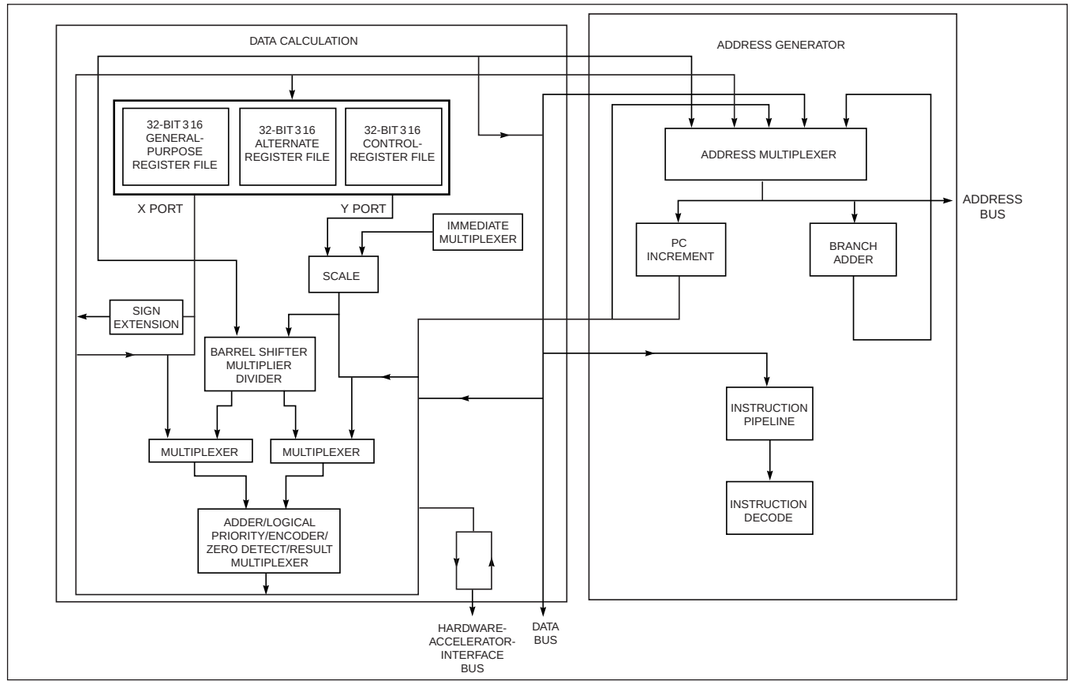
MCore contains a 32-channel interrupt controller. The processor can take in an asynchronous interrupt and get to the first instruction of an interrupt-service routine in six clocks. The CPU determines interrupt prioritization through software, and you can use the find-first-one instruction to scan for the highest priority interrupt and use the resulting value as an offset into a jump table. An interrupt-control bit in the program-status register allows an event to interrupt multicycle instructions, such as the load/store multiple-register instructions. For applications that are less real-time-critical, you can set this bit to prevent the interrupts from breaking into instructions.

MCore's hardware-accelerator interface supports a variety of application-specific functions. You can use one of several interface mechanisms. For example, a register-snooping mechanism reflects updates of MCore's registers across the interface without explicit passing of parameters from the core to the hardware accelerator.

Power management: Besides a 16-bit external interface to minimize power consumption, MCore also implements three software-controlled low-power modes and controls functional-unit clocking. The core runs as low as 1V, although the first products operate from 1.8 to 3.3V.

Development tools: To support the development tools for MCore, Motorola has established a validation center to ensure that third-party vendors comply with the Motorola-defined Application Binary Interface (ABI). ABI ensures that MCore tools will work together in a development environment.

(continued on pg 153)



Although MCore has limited tool support, the tools that are available should handle most development needs. Diab Data (www.diabdata.com) supplies a C/C++ compiler, and Motorola offers a Gnu tool kit. Software Development Systems (SDS, www. sdsi.com) supplies a simulation and debugging tool that offers memory and peripheral simulation. Integrated Systems Inc (www.isi.com) and Microtec (www. microtec.com) have ported their pSOS+ and OS-9000 RTOSs, respectively, to the MCore architecture. Hewlett-Packard (www.hp.com) offers a hardware-based runtime controller that operates through the on-chip emulation circuitry and MCore's JTAG interface.

Motorola also offers the MCore V1 evaluation system (EVS) that comprises a mC-memory board, an I/O-personality board, and a test-interface board. These boards provide 512 kbytes of fast static RAM (zero wait states at 20 MHz); 2 Mbytes of flash; 64 bits of general-purpose input or output; and peripherals, including two controller-access-network interfaces, a queued serial module, a queued ADC, and others; and logic-analyzer connectors. The EVS also includes evaluation copies of the SDS debugger and Diab Data compiler.

Second sources: Lucent Technologies (www.lucent.com) has licensed the MCore technology.

Motorola 68EC000

The 68EC000, a base for the 680x0 and 683xx lines of 32-bit mPs, mixes 16- and 32-bit architectures. It has 32-bit registers for easy addressing, a 16-bit datapath and ALU to conserve silicon, and 16-bit instructions. Programmers get eight general-purpose, 32-bit data registers, which the CPU can address by bit, BCD, byte, word, or double word. In addition to user and supervisor stack pointers, 68EC000 chips have seven address registers. Other registers include the 32-bit program-counter and 16-bit status registers. The status register maintains status for the user and supervisor modes with ser and supervisor bytes. The 68EC000 implements user and supervisor modes in hardware, which eases having a control kernel or OS manage multiple application tasks.

The 68EC000 has microcode and second-level, expanded-nanocode microcode levels. Instruction execution triggers a chain of 10-bit microcode words. Each microcode word can reference another word, such as a jump in microcode or a string of 70-bit nanocode words that directly drive the CPU logic.

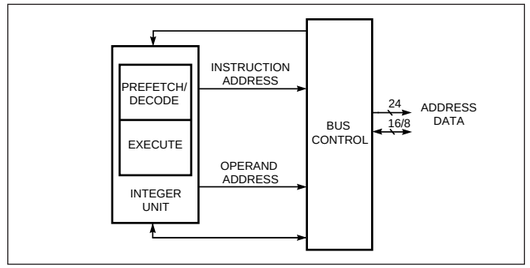
The CPU lacks a memory controller, but the separate address and data buses eliminate the need for buffering addresses. However, the CPU needs logic to generate the required DTACK* signal, which marks the successful completion of a memory cycle. An address decoder is necessary for multiple memory chips, and drivers may be necessary to buffer bus address and data lines. (Integrated versions of the 68EC000 contain this logic.) If DTACK* is late, the CPU generates wait states.

Power management: Only the integrated versions provide variations of sleep and low-power-stop modes.

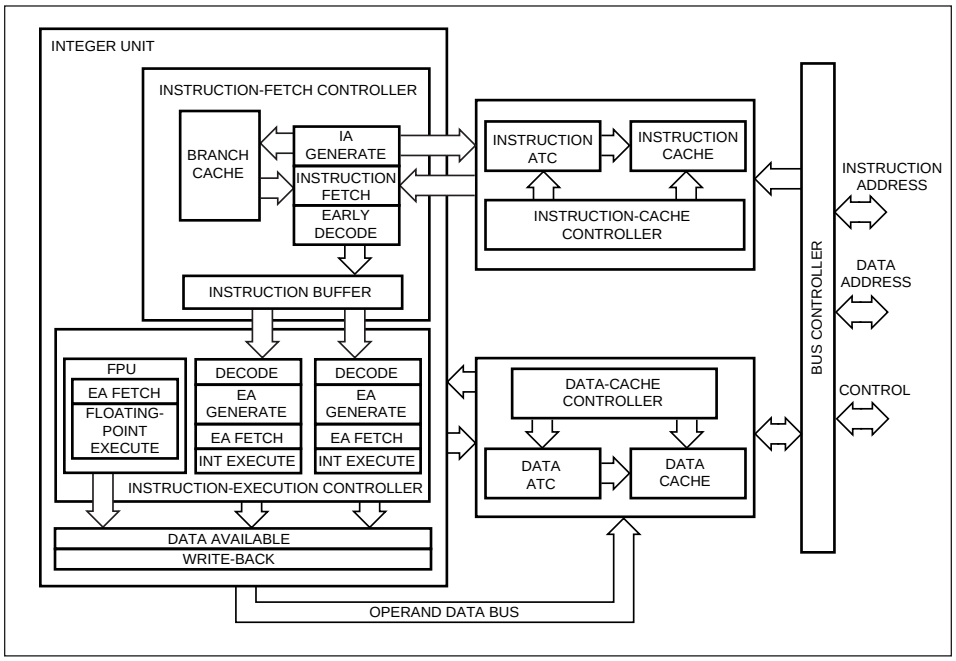
Special instructions: The chip restricts privileged instructions—reset, stop, moves, and operations on the status register—to supervisor mode. To support user and supervisor modes, the hardware implements separate stacks and pushes and pops the program counter and status register onto the stack for exceptions. A link instruction lets you build link lists on private stacks. A special instruction lets you move as many as 16 registers to or from an effective address, including blocks of data registers to or from address registers.

Development tools: Green Hills Software (www.ghs.com) provides C, C++, Fortran, Pascal, and Ada compilers for the 68K architectures. This company also provides its Multi software-development environment for developing programs from these languages and mixing them into a single executable program in almost any combination. Hewlett-Packard (www.hp.com) offers logic analyzers, oscilloscopes, emulators/analyzers, software simulators, debugger/emulator software, a real-time software-performance analyzer, C compilers, assemblers, linkers, and a debugging utility for RTOSs. Huntsville Microsystems (www.hmi.com) supplies emulators, a $199 background-mode debugger (BMD), and simulators for Motorola devices. The company offers its HMI-200 Series and SPS-2000 Series emulators. Integrated Systems (www.isi.com), Microtec (www.microtec.com), and Microware (www.microware.com) provide RTOSs and a variety of other software tools to support hardware and software integration. Intermetrics (www.intermetrics.com) offers compilers, assemblers, utilities, debuggers, and royalty-free real-time kernels. Orion Instruments (www.yokogawa.com) offers in-circuit emulators and high-level-language source debuggers for Windows or Unix hosts. Software Development Systems (www.sdsi.com) provides C and C++ compilers; assemblers; simulators; debuggers for the target monitor, BDM, and JTAG; and interactive development and debugging environments. Wind River Systems (www.windriver.com) provides an RTOS, networking facilities, and a set of cross-development tools. Wind River also provides a diagnostic and analysis tool that provides visibility into the dynamic operation of an embedded system.

Second sources: Second sources of a few NMOS versions of the 68000 are Hitachi, Philips, and Toshiba.



Motorola 680x0



Motorola built the 680x0 architecture around 16 general registers with a 68000-compatible, orthogonal instruction set. The 680x0 has more registers than the original 68000. Motorola added the control registers to control the memory-management unit (MMU) and the floating-point unit (FPU) and to support additional processing capabilities. For example, the 68040 adds eight 80-bit floating-point registers and 12 control registers.These registers include a vector-base register, points to an interrupt-vector table; a cache-control register; user and supervisor root pointers; and translation registers.

The superscalar 68060 heads the 680x0 line-up with its dual integer and floating-point pipelines. As instructions enter the CPU, they flow into a four-stage prefetch pipeline: instruction-address generation, instruction cycle, instruction early decode, and instruction buffer. In this pipeline, the 680x0 converts 68000-compatible, variable-length CISC instructions to a fixed-length instruction. These instructions then enter dual, four-stage, synchronously operating, integer-execution pipelines. The decode, effective-address-calculation, fetch, and integer-execution pipeline dispatches instructions to the FPU and allows for some execution overlap between the integer and FPU engines.

A Harvard architecture allows the 68060 to perform simultaneous instruction fetches and data accesses. The four-way set-associative, four-way-interleaving, on-chip caches support simultaneous read and write operations. You can freeze portions of the caches to prevent reallocation.

The 68040 implements a fetch, decode, effective-address-calculation, effective-address-fetch, execute, and write-back pipeline. To speed processing, the device has two 4-kbyte direct-mapped caches and separate data and instruction MMUs, which allow simultaneous address translations. The 040 includes bus snooping to ensure cache coherency for multiprocessing. The cache supports both write-through and copy-back modes. The 68020 and 68030 CISC implementations have smaller caches; the 030 and 040 versions implement burst mode, moving as much as 16 bytes in an addressing block between registers and memory.

The 040 and 060 deliver apparent single-cycle execution for some instructions, mainly register operations such as memory-to-register moves if the data is in the data cache. A taken branch takes two cycles; a not-taken branch takes three cycles. On the 68060, a 256-entry, four-way, set-associative, on-chip branch cache allows taken and nontaken branches to execute in zero and one clock, respectively. The branch-cache unit contains state bits that provide a history of branch executions, which helps to predict branch direction.

Unlike the 020/030, the 040 and 060 perform no dynamic bus sizing. Instead, they have a highly reliable bus with a high-drive option that can implement a synchronous, two-clock read/write protocol. A 4-word burst takes five clocks. The 040 includes multiprocessor-bus arbitration but requires off-chip logic. Externally, the 68060 bus is a superset of the 68040 bus. Additional signals support higher performance system designs, but the processor can easily operate on a 68040-based bus. An on-chip MMU with separate instruction and data translation-look-aside buffers allows the 68060 to access as much as 4 Gbytes of memory.

Power management: To support power management, the 68060's functional units respond to dynamically controlled clocking; the caches and execution units power down when not accessed. The static design allows you to reduce or stop the external clock, and a low-power-stop (LPSTOP) instruction disconnects most of the chip from the clock pin.

Special instructions: The CPUs have special instructions for variable-length bit fields; moving 16 registers; compare; and swap, which locks memory for multiprocessing. A scaling option addresses data by item size for table-access, FPU, and MMU commands.

The 68040 and 68060 have a special move instruction (MOVE16) to perform a 16-byte block move and a PLPA instruction that loads a physical address by translating a logical address. A table instruction performs a table look-up and interpolates the data.

Special peripherals: The MC68150 allows the 68040, LC040, and EC040 bus to communicate bidirectionally with 32-, 16-, or 8-bit peripherals and memories. The XC68HC901 multifunction peripheral comprises a one-channel USART and an eight-source interrupt controller.

Development tools: The 680x0 shares many of the same tools as the 68EC000.

Second sources: Toshiba acts as a second source for some versions of the 680x0.

Motorola 683xx

For most of the 683xx family, Motorola combined a stripped-down 68020 core with a 16-bit (32-bit for CPU32+) on-chip InterModule Bus, which links the CPU with a device's complex peripherals. The core processor, the CPU32 or CPU32+, is a 68020 CPU for embedded control that lacks memory-management-unit (MMU) or floating-point-unit (FPU) interfaces. The CPU32 and CPU32+ have 16- and 32-bit data buses, respectively. The 32-bit processor has eight general-purpose, 32-bit registers; seven 32-bit address registers; a 32-bit ALU; and separate user and supervisor modes, each with its own stack and separate address and data spaces. The CPU32 is code-compatible with the 68020 but has enhanced addressing modes, including scaled index; address-register indirect with base displacement; and index, program-counter-relative, and 32-bit branch displacements. Postincrement and preincrement/decrement options simplify iterative code. The CPU accesses memory-mapped peripheral-control registers and I/O as addresses in memory.

All 683xxs have a system-integration module featuring system configuration, oscillator and clock dividers, reset and power-down-mode control, chip selects and wait states, parallel I/O with interrupt capability, interrupt configuration/response, and a software watchdog timer. The external-bus interface has as many as 32 address and 16 data lines (32 for CPU32+) and as many as 12 programmable chip-selection lines. The single-chip Integration Module II allows users to select 32-kHz or 4-MHz clock crystals.

Power management: A low-power-stop (LPSTOP) instruction stops the clock. Devices can run at low frequencies.

Special instructions: The 68020 does not support BCD-pack/unpack, bit-field, compare-and-swap, coprocessor, MMU, module-call/return, and memory-indirect-addressing instructions. New instructions include a table look-up and interpolate and the ability to put the chip into a low-power standby mode.

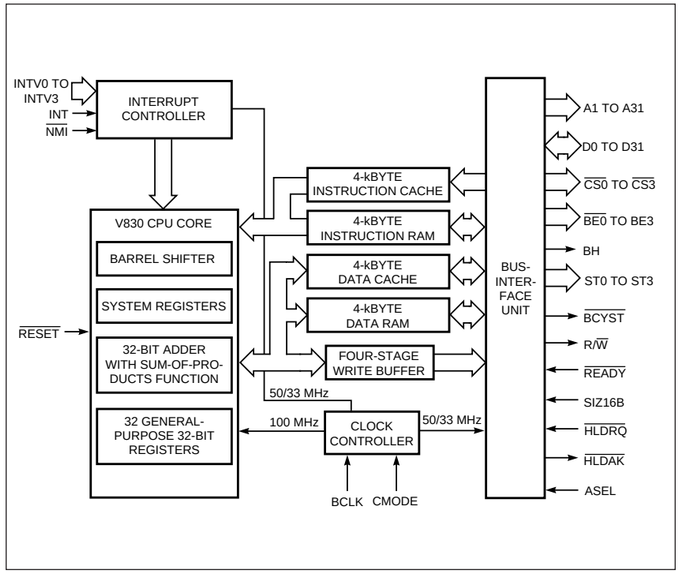
Development tools: The 683xx leverages the extensive development-tool support from the 68xxx architecture. These tools include assemblers, compilers and debuggers, RTOSs, emulators, and evaluation boards.

Second sources: There are no second sources for the 683xx.

NEC V800

NEC's V800 Series µCs are available as cores and standard products. All core versions contain the same peripherals as those in the standard devices. The company based the V800, including the V830 and V850, on a proprietary, 32-bit RISC architecture. NEC designed the V830 for embedded multimedia applications with external-memory support; the family provides on-chip instruction and data caches with demultiplexed address and data buses. The V850 is a µC with integrated RAM, ROM, and flash options.

The V800 architecture comprises a five-stage pipeline: fetch, decode, execute, memory access, and write back; 32 general-purpose registers; a 32-bit barrel shifter; and a hardware multiplier. Most instructions execute in one clock and are 2 bytes long, allowing smaller code. The CPU has a pipeline-stall feature that automatically inserts a bubble in the pipeline to avoid data dependencies and hazards. Instruction and data accesses occur on separate buses. Interrupt latency from an external source or peripherals is a minimum of 11 CPU cycles.



A bus-control unit (BCU) generates a prefetch address to prefetch an instruction code from external memory and store it in the 4-double-word prefetch queue. For accesses from internal ROM, instructions go straight to the CPU; that is, not through the prefetch queue. Instruction fetches from internal ROM consume one cycle; data fetches from ROM require three cycles. Therefore, you should shadow look-up tables and fixed data structures to the CPU's internal RAM, in which the V800 can access data in one clock. The BCU also provides a bus-arbitration function, allowing other devices, such as DMA, to share and take control of the V851's external bus. Programmable wait- and idle-state insertion control facilitates interfacing to slow memory. Although most of the microcontrollers provide as much as 16 Mbytes (24 address bits) of linear addressing, the new V850E also provides dynamic bus sizing. The maximum addressing range of the V800 architecture is 4 Gbytes.

The V800 accesses peripherals as memory-mapped I/O that connects to the CPU through a 16-bit bus. ROM and RAM communicate to the CPU using a 32-bit bus. Although the first member of this family, the V851, has 32 kbytes of ROM and 1 kbyte of RAM, the V850 architecture allows internal expansion to 1 Mbyte of ROM and 4 kbytes of RAM. Similarly, the external bus of the V851 addresses as much as 16 Mbytes. (The architecture allows access to as much as 4 Gbytes on future chips.) The V850's memory space divides into 1-Mbyte unit blocks, and you can insert wait states in a bus cycle for every two blocks.

Power management: In halt mode, the clock generator continues to operate, but the CPU clock stops, allowing the on-chip peripherals to function. Idle mode stops the CPU clock and internal-system clock; however, because the clock generator continues to run, normal operation can resume without waiting for oscillator and PLL stabilization. In stop mode, everything stops, but register and memory contents stay intact.

Special instructions: NEC's V800 devices support a software-trap instruction. The CPUs also perform saturate operations in which the devices store maximum values of additions that result in overflow. For example, if the result exceeds the positive-value 7FFFFFFFh, the CPU stores 7FFFFFFFh in the result registers and then sets the saturation flag. The V850E device also provides single-cycle byte-swapping operations for endian translation of data structures. Also, NEC includes single instructions to assist in C procedure calls for pushing and popping multiple registers. The net effect would be a decrease of code in the prologue and epilogue sections in C and a resulting speed increase.

Special on-chip peripherals: An on-chip DRAM controller, synchronous flash controller, and DMA controllers are available in the latest devices.

Development tools: NEC, Green Hills Software (www.ghs.com), and Cygnus Support (www.cygnus.com) offer C-compiler tool chains for the V800. Accelerated Technology (www.atinucleus.com), Green Hills Software, NEC, JMI Software (www.jmi.com), and Wind River Systems (www.windriver.com) provide RTOSs. A host of stand-alone evaluation boards, PC ISA-bus evaluation boards, and in-circuit emulators is available from NEC and third-party vendors. NEC works jointly with Synopsys (www.synopsys.com) and Mentor (www.mentor.com) to provide simulation tools for the V800 Series embedded core/ASIC development. NEC's OpenCAD environment supports these tools and is compatible with the standard device-development tools.

Second sources: There are no second sources for the V800, but Lucent Technologies (www.lucent.com) licenses the V850 as a core within its cell-based ASCI library.

Siemens Tricore

The Siemens Tricore architecture represents the industry's trend toward a blurring of the distinction between microcontrollers and DSPs (see "Microprocessor and DSP technologies unite for embedded applications," *EDN*, March 2, 1998, pg 73). This architecture's functional units and its unified instruction set target microcontroller- and DSP-specific functionality. Tricore is a superscalar core with two primary four-stage pipelines; the first bit of every instruction identifies which pipeline that instruction follows. One pipeline does loops, loads, and address-generation arithmetic; the other pipeline does all the math and branches. The execute unit comprises a multiply-accumulate (MAC) module, an ALU, and a tightly coupled coprocessor interface. A third pipeline performs loop control for zero-overhead looping. Tricore supports a mixture of 16- and 32-bit-wide instructions to help conserve code space; each operation code includes a size bit to improve the efficiency of instruction decoding.

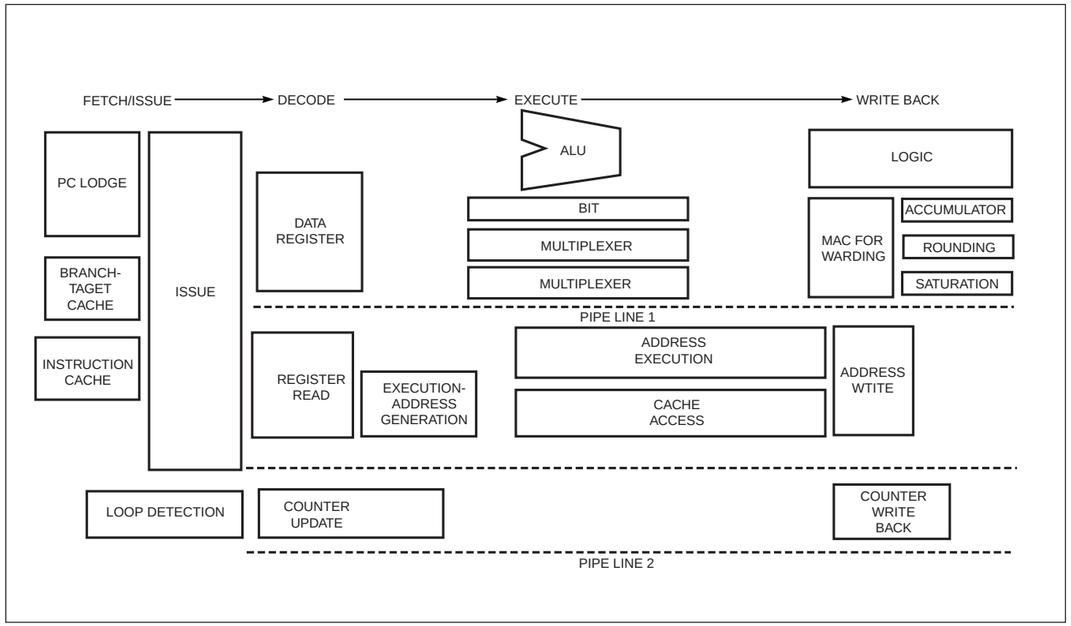
Tricore implements a Harvard architecture with separate address and data buses for program and data memories. Tricore is also a load/store architecture with 16 32-bit general-purpose data registers and 16 32-bit address registers. You can concatenate consecutive even-odd data registers to form eight 64-bit registers for extended precision.

Unlike traditional DSPs, Tricore lacks separate X- and Y-memory spaces, which may require you to perform some loop unrolling to achieve the parallel performance of DSPs. As long as data is available for Tricore's execute unit. The device can perform single-cycle MACs. The data side of the core has a 128-bit-wide bus to on-chip DRAM, which you can use to save two data and two address registers in one cycle to the cache. Tricore supports circular buffers for DSP filters and bit-reversed indexing for FFTs. You must align the start of the circular buffer to a multiple of the data size, which the instruction using the buffer prescribes. The length of the buffer must also be a multiple of the data size the instruction using the buffer references.

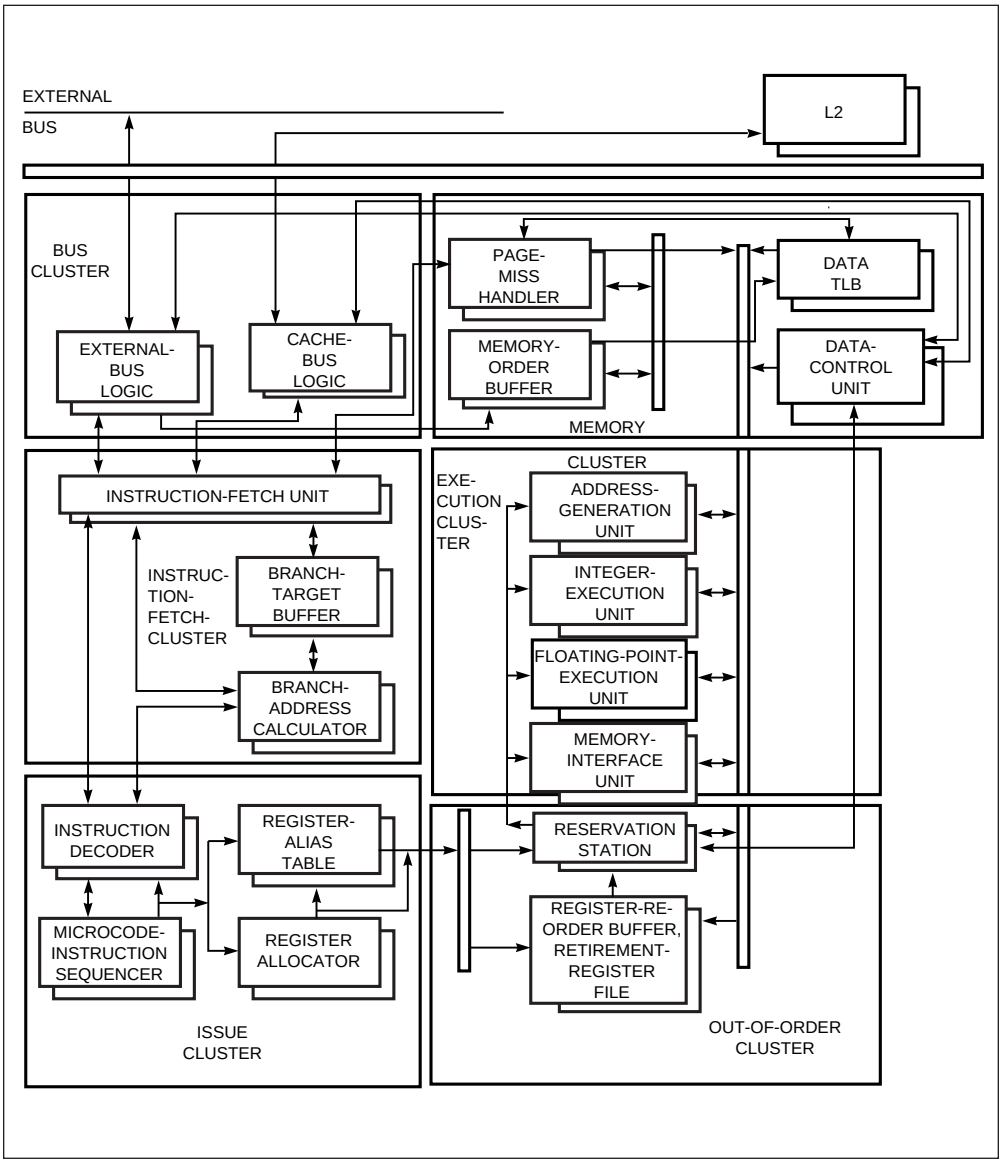
Special instructions: The instruction set supports operations on Booleans, bit strings, characters, signed fractions, addresses, signed and unsigned integers, single-instruction multiple-data, and single-precision floating-point numbers. In addition to a plethora of microcontroller-oriented instructions, such as bit manipulation, Tricore supports the traditional DSP instructions, including multiply and MAC, saturate, scaling, and rounding. Tricore also supports packed arithmetic. Conditional add, subtract, and select instructions let the device avoid using conditional jumps.

Development tools: Tasking (www.tasking.com) and Green Hills Software (www.ghs.com) offer C- and C++-compiler, debugger, simulator, and RTOS support for Tricore. Accelerated Technology (www.atinucleus.com) also supplies a Tricore RTOS. Nohau (www.nohau.com), Ashling (www.ashling.com), Hitex (www.hitex.de), and Lauterbach (www.lauterbach.com) supply Tricore in-circuit emulators. Tasking's instruction-accurate simulator allows you to analyze the basic functionality of your program. Siemens also has a cycle-accurate simulator that implements a flexible cache model that provides options, such as defining start and end addresses, the number of ways and cache lines, and the line size and banks. This simulator also includes branch-prediction logic and determines interrupt latency.

Second sources: There are no second sources for the Tricore.



Slot 1 processors



Because of Intel's patent on the Slot 1 processor's CPU interface, the company is the sole source of these processors, primarily characterized by their dedicated backside L2-cache bus. Separate buses to main memory and L2 cache allow the CPU to access the L2 cache at 200 MHz, one-half the core-processor speed. Slot 1 provides Intel's GTL+ bus protocol, which is helpful in multiprocessor systems.

Intel's Slot 1-compatible Pentium II barely resembles the Pentium or any other x86 processor. With a decoupled 12-stage pipeline, the Pentium II trades less work per pipe stage for more stages. The Pentium II comprises three independent engines: fetch/decode, dispatch/execute, and retire. The fetch/decode engine converts instructions into one or more micro-operations (mops). The mops improve performance by representing fixed-length, fixed-field, easy-to-execute operations. You can individually schedule the mops, facilitating the Pentium II's out-of-order execution of instructions.

After the decoder creates mops, it sends them to a 40-mop-deep reorder buffer (ROB). The mops then await dispatch to the execution portion of the pipeline. The mops are then either ready for execution or waiting for data from a memory access or a result from a previous mop. To avoid register dependencies, the Pentium II performs renaming: Extra registers represent the x86's programmer-visible registers. The dispatch/execute engine queues ready-for-execution mops within a 20-entry, distributed-reservation station. The Pentium II determines the data flow by analyzing which mops depend on each other's results. The processor dispatches mops from anywhere or in any order within the reservation station.

The Pentium II speculatively executes and returns these mops to the ROB, and the retire engine then evaluates them. Although the Pentium II executes mops or instructions out of order, the device completes the instructions in the original program order. Furthermore, speculative execution implies that the device executes some instructions that never retire. This situation occurs if the device mispredicts a program branch. When the Pentium II encounters a mispredicted branch, it flushes its deep pipelines and removes mops from the ROB. To minimize the possibility of a mispredicted branch, Intel designers increased the branch target buffer to 512 entries and added history bits to help the prediction algorithm.

The Pentium II, with the same multimedia-extension instructions as Pentium, comes in a single-edge-contact cartridge with a 512-kbyte L2 cache. This year, Intel introduced its Celeron and Pentium II Xeon processors. Celeron, which comes in a single-edge processor package, is basically a cacheless Pentium II and targets the less-than-$1000 PC market. At the other extreme is Xeon, targeting the midrange to high-end server and workstation market. Xeon is a Pentium II with 512 kbytes or 1 Mbyte of L2 cache. It comes in a Slot 2 module—approximately twice the weight and height of a Slot 1 module—and supports four- or eight-way multiprocessing.

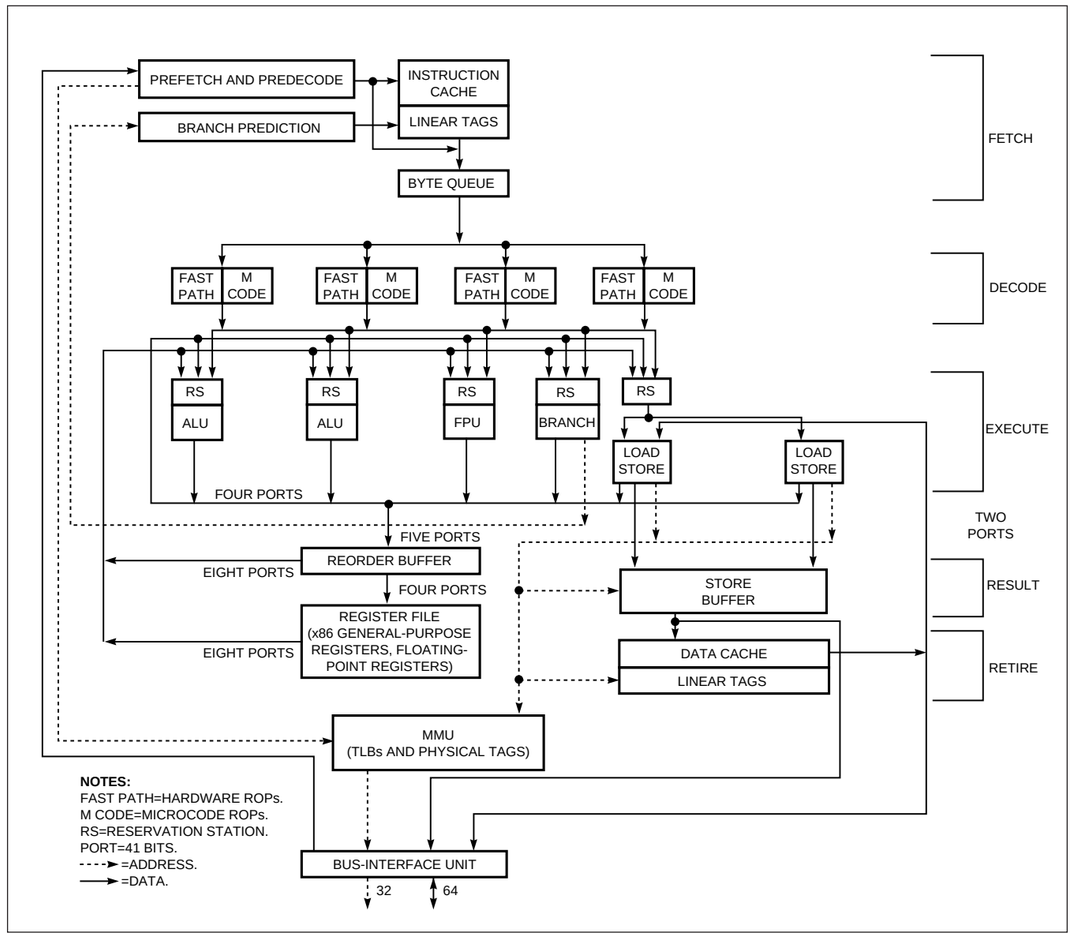
Socket 7 processors

Socket 7 processors, primarily characterized by a common interface between the L2-cache bus and the main system bus, are available from AMD, IBM, Integrated Device Technology (IDT), Intel, Cyrix/National Semiconductor, and Rise Technology (www.rise.com). The common interface typically limits the bus's clock speed; however, some of these vendors recently increased the bus speed from 66 to 100 MHz, effectively boosting the bandwidth by 50%. One other way around the bandwidth limitation is to put the L2 cache on chip; AMD and IDT plan to take this approach this year.

Intel's Pentium processor, the first Socket 7 processor, emphasizes executing simple instructions before complex ones. With Pentium, the simple, RISC-like register-to-register instructions drive the implementation; the microcoded complex instructions are second priority.

Pentium achieves a two-instruction issue peak and has two five-stage pipelines (U and V) for each instruction. A common instruction fetch/align stage, which fetches multiple instructions from the cache, feeds the U and V pipelines. The CPU passes a full 256-bit line to the instruction decoder. Each pipeline has two decoder stages to decode simple and complex instructions. The wide cache-to-decoder path with two-stage decoding enables Pentium to decode the x86's variable-length instructions.

Pentium includes 57 instructions to support multimedia applications, such as image processing and audio synthesis. More fundamentally, these multimedia-extension (MMX) instructions benefit applications with vectorizable code. Eight 64-bit MMX registers, MM0 to MM7, support these instructions and data types; these registers are "aliased"—physical silicon is the same but the registers' names change—with the floating-point registers. Register aliasing eliminates additional silicon for new registers. It also eliminates the need to modify the operating system or system BIOS, which must



Socket 7 processors (continued)

track these registers. However, aliasing inhibits you from performing routines that combine floating-point and MMX instructions; switching from MMX instructions to floating-point instructions can take as many as 50 clock cycles. Before the CPU can execute a floating-point instruction, you must use the empty-MMX-state instruction to set up the floating-point registers.

For superscalar, dual-instruction, load/store operations, the dual-ported Pentium data translation-look-aside buffer (TLB) and cache tags provide concurrent pipeline accesses. The eight-way-interleaved data-cache SRAM allows concurrent accesses to memory banks. (The cache is actually triple-ported with an extra port for snooping.) Cache-hit rates range from 90 to 97%, depending on the code mix. The data cache handles both 4-kbyte and 4-Mbyte pages. It has two four-way, set-associative TLBs, one with 64 entries for 4-kbyte pages and one with eight entries for 4-Mbyte pages. The two-way, set-associative code cache has a four-way, set-associative, 32-entry TLB that handles both 4-kbyte and 4-Mbyte pages.

Dynamic branch prediction allows the CPU to determine which branch to take. Pentium's 256-entry branch-target buffer (BTB) holds branch-target addresses for previously executed branches. The BTB supplies the next instruction address that the last execution of a branch instruction took. Each BTB entry integrates the target address with history and operation bits. Intel claims that a correctly predicted branch takes one pipeline cycle and doesn't cause a pipeline bubble.

Pentium's floating-point unit features an eight-stage pipeline, which shares the first five stages of the U and V pipelines. Data transfers to or from the FPU use a 64-bit-wide datapath to the data cache. Pentium adds a write buffer to each pipeline to avoid write contention.

Pentium uses burst reads to fill its 256-bit-wide cache line. It also has burst write-back writes. The pipelined memory interface allows a second bus cycle to set up while the first bus cycle completes. Pentium reads or writes a 64-bit double word each cycle in burst mode.

AMD's K6-2 with MMX is a six-issue, superscalar mP with a Socket 7-compatible bus interface that runs at 100 MHz. It features a decoupled, decoding/executing, superscalar design that can simultaneously decode multiple x86 instructions. It also performs single-clock RISC operations, out-of-order execution, data forwarding, speculative execution, and register renaming. The K6-2 processor, based on a six-stage pipeline, contains parallel decoders, a centralized RISC86 operation scheduler, and seven execution units.

Similar to the Pentium II, the K6-2 decodes x86 instructions into RISC86 operations that adhere to the RISC-performance principles of fixed-length encoding, regularized instruction fields, and a large register set. The K6-2 implements branch-prediction logic in the form of an 8192-entry branch-history table, a branch-target cache, and a return-address stack. In the K6-2, x86 instruction decoding begins before the CPU fills the on-chip instruction cache. Predecoding logic determines x86-instruction length on a byte-by-byte basis. The K6-2 stores this predecode information, along with x86 instructions, in the instruction cache for later use by the decoders. The decoders translate as many as two x86 instructions per clock into RISC86 operations.

The scheduler contains the logic needed to manage out-of-order execution, data forwarding, register renaming, simultaneous issuing and retirement of multiple RISC86 operations, and speculative execution. The scheduler's RISC86 operation buffer can hold as many as 24 operations. The scheduler can simultaneously issue a RISC86 operation to any available execution unit (store, load, branch, integer, integer/multimedia, or floating point). The scheduler can issue as many as six and retire as many as four RISC86 operations per clock.

Unlike the K6-2 (or Pentium II), Cyrix/National's MII processor directly executes native x86 instructions, rather than converting x86 instructions into RISC-like instructions. The MII achieves a dual-x86 instruction issue/execute rate using dual seven-stage pipelines. The CPU performs register renaming, multilevel dynamic branch prediction, speculative execution, and out-of-order completion. The MII has a dual-ported, 64-kbyte cache and a dual-ported, 384-entry TLB; both support two reads and two writes or one read and one write on every cycle. The processor allows you to turn individual cache lines into scratchpad RAM to provide support for multimedia operations. In addition, the MII fully supports Intel's MMX instruction set.

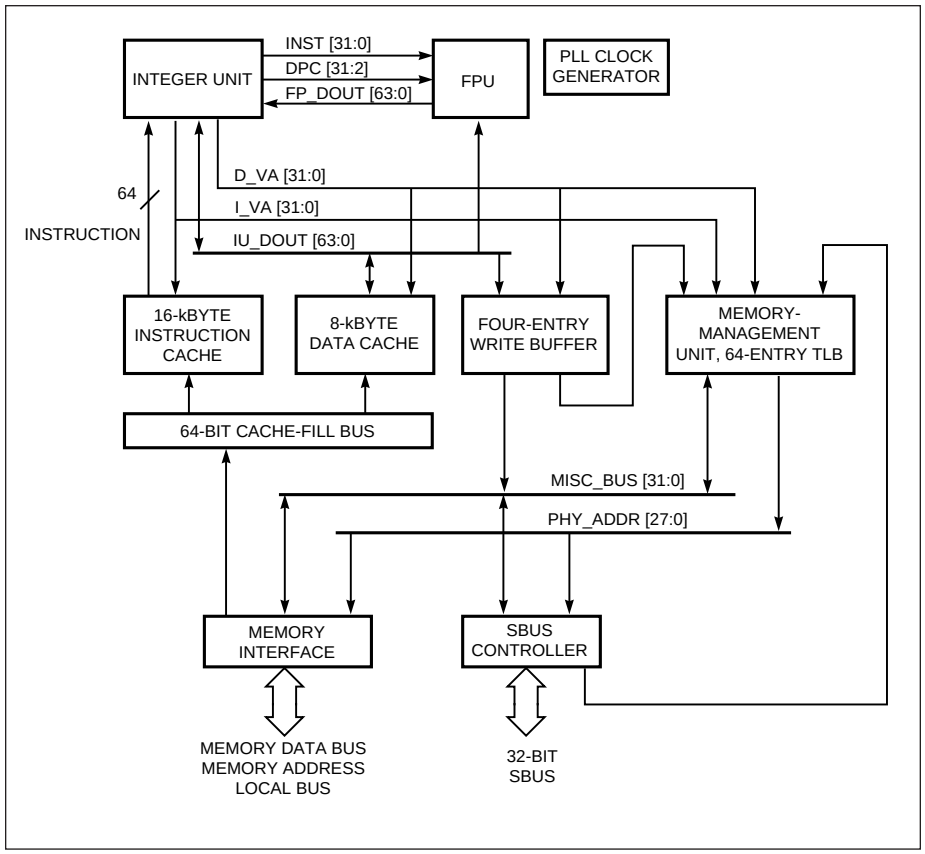
The instruction-fetch stage of the MII's pipeline fetches 16 instruction bytes per cycle from the instruction cache and feeds the instruction-decode stage. The instruction decoder issues as many as two complex x86 instructions per cycle. During decoding, the decoder examines the resource requirements of the two instructions and chooses the optimal pipeline for each instruction. During these stages, the decoder accesses the 512-entry BTB and the 1024-entry branch-history table to avoid pipeline bubbles.

During the access stages of the pipeline, the CPU performs scoreboard checks, renames registers, and accesses the physical register file. The MII also calculates one or two linear addresses per cycle for all addressing modes and accesses the translation-look-aside and cache. The ability to fetch as many as two memory operands from the data cache before the instruction-execution stage allows the MII to execute memory-reference instructions in one cycle.

Cyrix's Media GX processor with MMX performs all standard north-bridge functions of a PC's core logic. It also performs the functions of the PC's graphics controller, audio chip set, memory controller, and CPU-to-PCI bridge. Rather than using only transistors to perform these functions, Cyrix developed its Virtual System Architecture (VSA). VSA supports the graphics- and audio-hardware functions through software. VSA uses the Media GX's system-management interrupt to capture any accesses to the memory- or I/O-address ranges of the graphics and audio functions. Once the processor enters system-management mode, it executes Cyrix-supplied drivers to perform the appropriate function.

Special instructions: MMX instructions operate on single-instruction-multiple-data (SIMD) types. MMX instructions include basic arithmetic operations, including add, subtract, multiply, and divide; logical operations, such as AND, OR, and AND NOT; compare operations; conversion instructions to pack and unpack data elements; shift operations; and data-movement instructions. AMD has developed 3-D instruction extensions known as 3DNow, which will also be implemented by Cyrix and IDT.

Sun microSPARC



Sun built the microSPARC processors around a large, multiported register file that divides into a small set of global registers for holding global variables and sets of overlapping register windows. Each 24-register window has a core of eight registers; eight registers overlapping the previous and next register windows supplement the eight-register core. The overlapping registers eliminate the need to save and restore registers on function calls, returns, or context switches between tasks.

The microSPARC has a five-stage pipeline: fetch, decode, memory access, execute, and write back. It also has a four-entry write buffer to prevent write stalls. A floating-point unit (FPU) contains 32 3 32-bit floating-point registers, a general-purpose execution unit, and a floating-point multiplier. A three-entry queue of floating-point instructions increases concurrency with integer execution.

MicroSPARC includes a SPARC-compliant memory-management unit (MMU). This MMU uses 3 high-order bits of physical address to map eight address spaces. The MMU controls arbitration among I/O, data cache, instruction cache, and translation-look-aside-buffer (TLB) references to memory. The MMU contains a 64-entry, fully associative TLB and supports 256 contexts. The hyperSPARC's MMU uses a context register to identify as many as 4096 contexts.

The microSPARC mPs have a separate 64-bit memory interface that handles as much as 256 Mbytes of 16-Mbit DRAM. An on-chip, 25-MHz, 32-bit, synchronous Sbus (slave-bus) interface and controller handle five Sbus slots.

Special instructions: The microSPARC mPs comply with instructions in the SPARC V8 specification. T.sqware's (www.tsqware.com) HDLC controllers include a microSPARC core with built-in DSP capabilities via an extension to the SPARC instruction set and access to hardware operators using the coprocessor operating code.

Special peripherals: The microSPARC-II has an on-chip Sbus interface. Sun provides peripheral ASICs that attach to the Sbus and provide memory and I/O capabilities, such as Ethernet, serial, keyboard, mouse, SCSI, and parallel ports. One such ASIC, the PCIO chip, links the processor and 10/100-Mbit Ethernet; an 8-bit expansion bus links to standard "super-I/O"-like ASICs for connection to keyboards, mice, serial ports, and the like. The microSPARC-IIep contains a PCI interface for using industry-standard peripherals.

Development tools: A variety of OSs, each with its own set of development tools, supports the microSPARC-IIep. Sun's Solaris OS features the Workshop suite of development tools. Workshop contains a C/C++ compiler and source-code-control, debugging, and profiling tools. Workshop provides a self-hosted development environment allowing programmers to develop software for embedded applications on their desktop development workstations. Wind River's (www.windriver.com) Tornado provides an integrated suite of development tools for a cross-platform, host-target environment. Tornado features graphical host-based tools, a high-performance RTOS, and host-target communication protocols. Sun's Chorus group (www.sun.com) features the ClassiX RTOS. You can compile application code on a Solaris host with the Workshop compiler and debug the code with a Gnu-based source-level debugger.

ClassiX also features a Common Object Request Broker Architecture (CORBA)-compliant Object Request Broker and an Interface Definition Language (IDL) compiler. IDL describes the interface to a routine or function. For example, IDL defines objects in the CORBA distributed-object environment, which describes the services that the object performs and how data passes to the object. IDL stores the definitions in an interface repository that a client application can query to determine which functions, or objects, are available on the object bus. For developers using alternative system software, the Cygnus (www.cygnus.com) GnuPro C/C++ tool kit provides compiling and debugging tools.

Second sources: There are no second sources for microSPARC devices; however, Sun licenses the microSPARC core to C-Cube Microsystems (www.c-cube.com), Hyundai (www.hei.co.kr/), Scientific Atlanta (www.scientific-atlanta.com/), T.sqware, and Xylan (www.xylan.com).

Intel387™ SX
MATH COPROCESSOR

- **New Automatic Power Management**
  - **— Low Power Consumption**
  - **— Typically 100 mA in Dynamic Mode, and 4 mA in Idle Mode**
- **Socket Compatible with Intel387 Family of Math CoProcessors**
  - **— Hardware and Software Compatible**
  - **— Supported by Over 2100 Commercial Software Packages**
  - **— 10% to 15% Performance Increase on Whetstone and Livermore Benchmarks**
- **Compatible with the Intel386™ SX Microprocessor**
  - **— Extends CPU Instruction Set to Include Trigonometric, Logarithmic, and Exponential**
- **High Performance 80-Bit Internal Architecture**
- **Implements ANSI/IEEE Standard 754-1985 for Binary Floating-Point Arithmetic**
- **Available in a 68-Pin PLCC Package**

  See Intel Packaging Specification, Order #231369

The Intel387™ SX Math CoProcessor is an extension to the Intel386™ SX microprocessor architecture. The combination of the Intel387™ SX with the Intel386™ SX microprocessor dramatically increases the processing speed of computer application software that utilizes high performance floating-point operations. An internal Power Management Unit enables the Intel387™ SX to perform these floating-point operations while maintaining very low power consumption for portable and desktop applications. The internal Power Management Unit effectively reduces power consumption by 95% when the device is idle.

The Intel387™ SX Math CoProcessor is available in a 68-pin PLCC package, and is manufactured on Intel's advanced 1.0 micron CHMOS IV technology.

240225–22

Intel386 and Intel387 are trademarks of Intel Corporation.

January 1994 **Order Number: 240225-009**

Intel387™ SX Math CoProcessor

CONTENTS

PAGE

CONTENTS

PAGE

CONTENTS

CONTENTS

CONTENTS

PAGE

FIGURES

CONTENTS

PAGE

TABLES

1.0 PIN ASSIGNMENT

The Intel387 SX Math CoProcessor pinout as viewed from the top side of the component is shown in Figure 1-1. V_{CC} and V_{SS} (GND) connections must be made to multiple pins. The circuit board should include V_{CC} and V_{SS} planes for power distribution and all V_{CC} and V_{SS} pins must be connected to the appropriate plane.

NOTE:
Pins identified as N.C. should remain completely unconnected.

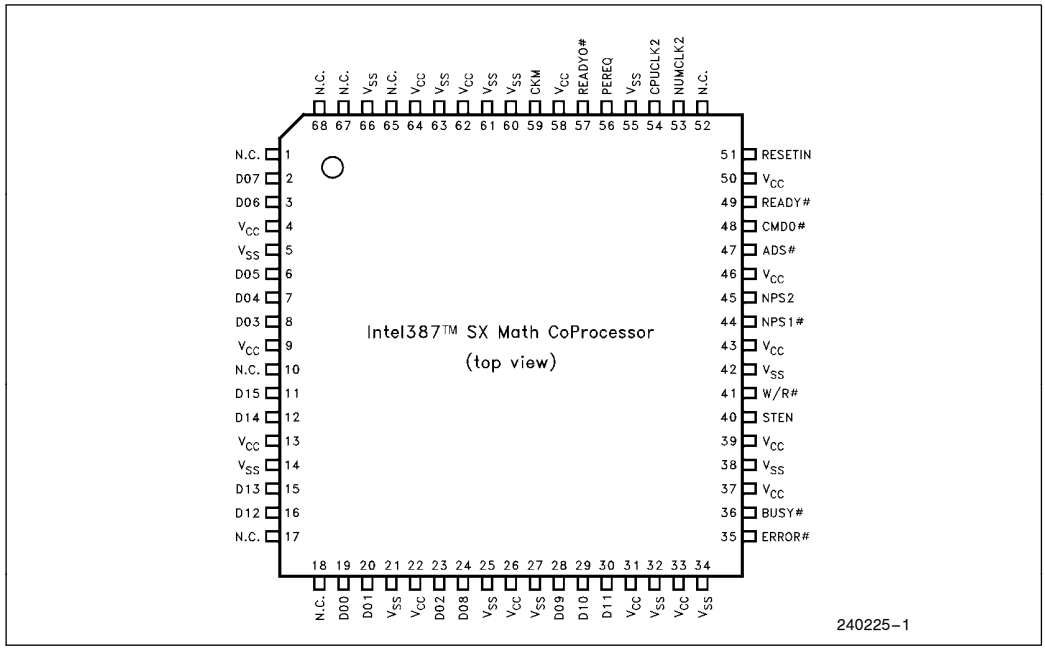


Figure 1-1. Intel387™ SX Math CoProcessor Pinout

Table 1-1. Pin Cross Reference—Functional Grouping

BUSY#	36	D00	19	V_{CC}	4	V_{SS}	5	N.C.	1
PEREQ	56	D01	20		9		14		10
ERROR#	35	D02	23		13		21		17
ADS#	47	D03	8		22		25		18
CMD0#	48	D04	7		26		27		52
NPS1#	44	D05	6		31		32		65
NPS2	45	D06	3		33		34		67
STEN	40	D07	2		37		38		68
W/R#	41	D08	24		39		42		
READY#	49	D09	28		43		55		
READYO#	57	D10	29		46		60		
CKM	59	D11	30		50		61		
CPUCLK2	54	D12	16		58		63		
NUMCLK2	53	D13	15		62		66		
RESETIN	51	D14	12		64				
		D15	11						

1.1 Pin Description Table

The following table lists a brief description of each pin on the Intel387 SX Math CoProcessor. For a more complete description refer to Section 4.1 Signal Description. The following definitions are used in these descriptions:

The signal is active LOW.

I Input Signal

O Output Signal

I/O Input and Output Signal

Symbol	Type	Name and Function
ADS#	I	**ADDRESS STROBE** indicates that the address and bus cycle definition is valid.
BUSY#	O	**BUSY** indicates that the Math CoProcessor is currently executing an instruction.
CKM	I	**CLOCKING MODE** is used to select synchronous or asynchronous clock modes.
CMD0	I	**COMMAND** determines whether an opcode or operand are being sent to the Math CoProcessor. During a read cycle it indicates which register group is being read.
CPUCLK2	I	**CPU CLOCK** input provides the timing for the bus interface unit and the execution unit in synchronous mode.
D15–D0	I/O	**DATA BUS** is used to transfer instructions and data between the Math CoProcessor and CPU.
ERROR#	O	**ERROR** signals that an unmasked exception has occurred.
NC	—	**NO CONNECT** should always remain unconnected. Connection of a N.C. pin may cause the Math CoProcessor to malfunction or be incompatible with future steppings.
NPS1#	I	**NPX SELECT 1** is used to select the Math CoProcessor.
NPS2	I	**NPX SELECT 2** is used to select the Math CoProcessor.
NUMCLK2	I	**NUMERICS CLOCK** is used in asynchronous mode to drive the Floating Point Execution Unit.
PEREQ	O	**PROCESSOR EXTENSION REQUEST** signals the CPU that the Math CoProcessor is ready for data transfer to/from its FIFO.
READY#	I	**READY** indicates that the bus cycle is being terminated.
READYO#	O	**READY OUT** signals the CPU that the Math CoProcessor is terminating the bus cycle.
RESETIN	I	**SYSTEM RESET** terminates any operation in progress and forces the Math CoProcessor to enter a dormant state.
STEN	I	**STATUS ENABLE** serves as a master chip select for the Math CoProcessor. When inactive, this pin forces all outputs and bi-directional pins into a floating state.
W/R#	I	**WRITE/READ** indicates whether the CPU bus cycle in progress is a read or a write cycle.
V_{CC}	I	**SYSTEM POWER** provides the +5V nominal D.C. supply input.
V_{SS}	I	**SYSTEM GROUND** provides the 0V connection from which all inputs and outputs are measured.

2.0 FUNCTIONAL DESCRIPTION

The Intel387 SX Math CoProcessor is designed to support the Intel386 SX Microprocessor and effectively extend the CPU architecture by providing fast execution of arithmetic instructions and transcendental functions. This component contains internal power management circuitry for reduced active power dissipation and an automatic idle mode.

2.1 Feature List

- New power saving design provides low power dissipation in active and idle modes.
- Higher Performance, 10%–25% higher benchmark performance than the original Intel387 SX Math CoProcessor.
- High Performance 84-bit Internal Architecture
- Eight 80-bit Numeric Registers, usable as individually addressable general registers or as a register stack.
- Full-range transcendental operations for SINE, COSINE, TANGENT, ARCTANGENT, and LOGARITHM.
- Programmable rounding modes and notification of rounding effects.
- Exception reporting either by software polling or hardware interrupts.
- Fully compatible with the SX Microprocessors.
- Expands Intel386 SX CPU data types to include 32-bit, 64-bit, and 80-bit Floating Point; 32-bit and 64-bit Integers; and 18 Digit BCD Operands.
- Directly extends the Intel386 SX CPU Instruction Set to trigonometric, logarithmic, exponential, and arithmetic functions for all data types.
- Operates independently of Real, Protected, and Virtual-86 Modes of the Intel386 SX Microprocessors.
- Fully compatible with the Intel387 SL Mobile and DX Math CoProcessors. Implements all Intel387 Math CoProcessor architectural enhancements over 8087 and 80287.
- Implements ANSI/IEEE Standard 754-1985 for binary floating point arithmetic.
- Upward Object Code compatible from 8087 and 80287.

2.2 Math CoProcessor Architecture

As shown in Figure 2-1, the Intel387 SX Math CoProcessor is internally divided into four sections; the Bus Control Logic, the Data Interface and Control Logic, the Floating Point Unit, and the Power Management Unit. The Bus Control Logic is responsible for the CPU bus tracking and interface. The Data Interface and Control Unit latches data and decodes instructions. The Floating Point Unit executes the mathematical instructions. The Power Management Unit is new to the Intel387 family and is the nucleus

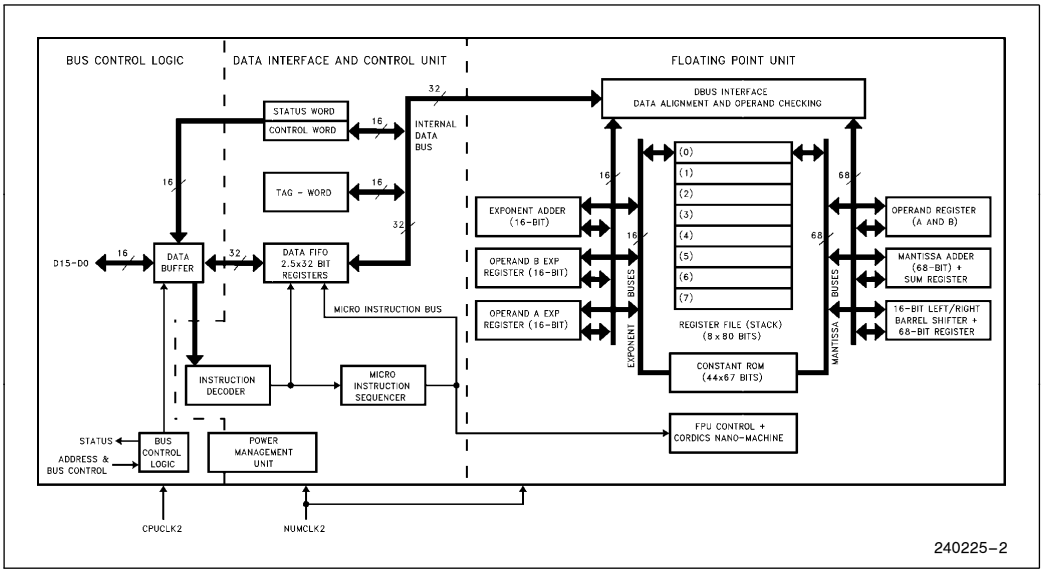


Figure 2-1. Intel387™ SX Math CoProcessor Block Diagram

of the static architecture. It is responsible for shutting down idle sections of the device to save power.

Microprocessor/Math CoProcessor Interface
The Intel386 CPU interprets the pattern 11011B in most significant five bits of an instruction as an opcode intended for a math coprocessor. Instructions thus marked are called ESCAPE or ESC instructions. Upon decoding the instruction as an ESC instruction, the Intel386 CPU transfers the opcode to the math coprocessor through an I/O write cycle at a dedicated address (8000F8H) outside the normal programmed I/O address range. The math coprocessor has dedicated output signals for controlling the data transfer and notifying the CPU if the Math CoProcessor is busy or that a floating point error has occurred.

2.3 Power Management

The Intel387 SX Math CoProcessor offers two modes of power management; dynamic and idle.

2.3.1 DYNAMIC MODE

Dynamic Mode is when the device is executing an instruction. Using Intel's CHMOS IV technology, the Intel387 SX Math CoProcessor draws considerably less power than its predecessor. The active power supply current is reduced to approximately 100 mA at 20 MHz and provides low case temperatures.

2.3.2 IDLE MODE

When an instruction is not being executed, the Intel387 SX Math CoProcessor will automatically change to ***Idle Mode***. Three clocks after completion of the previous instruction, the internal power manager shuts down the floating point execution unit and all non-essential circuitry. Only portions of the Bus Interface Unit remain active to monitor the CPU bus activity and to accept the next instruction when it is transferred. When the CPU transfers the next instruction to the Math CoProcessor, the Intel387 SX Math CoProcessor accepts the instruction and ramps the internal core within one clock so there is no impact to performance or throughput. In idle mode, the Intel387 SX Math CoProcessor draws typically 4 mA of current and reduces case temperature to near ambient.

NOTE:
In asynchronous clock mode (CKM = 0), the internal idle mode is disabled.

2.4 Compatibility

The Intel387 SX Math CoProcessor is compatible with the Intel387 SL Mobile Math CoProcessor. Due to the increased performance and internal pipelining effects, diagnostic programs should never use instruction execution time for test purposes.

2.5 Performance

The increased performance of floating point calculations can be attributed to the 84-bit architecture and floating point processor. For the CPU to execute floating point calculations requires very long software emulation methods with reduced resolution and accuracy. The performance of the Intel387 SX Math CoProcessor has been further enhanced through improvements in the internal microcode and through internal architectural changes. These refinements will increase Whetstone benchmarks by approximately 10% to 25% over the original Intel387 SX Math CoProcessor.

Real performance, however, should be measured with application software. Depending upon software coding, system overhead, and percentage of floating point instructions, performance can vary significantly.

3.0 PROGRAMMING INTERFACE

The Intel387 SX Math CoProcessor effectively extends to an Intel386 Microprocessor system additional instructions, registers, data types, and interrupts specifically designed to facilitate high-speed floating point processing. All communication between the CPU and the Math CoProcessor is transparent to applications software. The CPU automatically controls the Math CoProcessor whenever a numerics instruction is executed. All physical memory and virtual memory of the CPU are available for storage of the instructions and operands of programs that use the Math CoProcessor. All memory addressing modes, including use of displacement, base register, index register, and scaling are available for addressing numerical operands.

The Intel387 SX Math CoProcessor is software compatible with the Intel387 DX Math CoProcessors and supports all applications written for the Intel386 CPU and Intel387 Math CoProcessors.

3.1 Instruction Set

The Intel386 CPU interprets the pattern 11011B in most significant five bits of an instruction as an opcode intended for a math coprocessor. Instructions thus marked are called ESCAPE or ESC instruction.

The typical Math CoProcessor instruction accepts one or two operands and produces one or sometimes two results. In two-operand instructions, one operand is the contents of the Math CoProcessor register, while the other may be a memory location. The operands of some instructions are predefined; for example, FSQRT always takes the square root of the number in the top stack element.

The Intel387 SX Math CoProcessor instruction set can be divided into six groups. The following sections gives a brief description of each instruction. Section 8.0 defines the instruction format and byte fields. Further details can be obtained from the Intel387 User's Manual, Programmer's Reference, Order #231917.

3.1.1 DATA TRANSFER INSTRUCTIONS

The class includes the operations that load, store, and convert operands of any support data types.

Real Transfers

- FLD Load Real (single, double, extended)
- FST Store Real (single, double)
- FSTP Store Real and pop (single, double, extended)
- FXCH Exchange registers

Integer Transfers

- FILD Load (convert from) Integer (word, short, long)
- FIST Store (convert to) Integer (word, short)
- FISTP Store (convert to) Integer and pop (word, short, long)

Packed Decimal Transfers

- FBLD Load (convert from) packed decimal
- FBSTP Store packed decimal and pop

3.1.2 ARITHMETIC INSTRUCTIONS

This class of instructions provide variations on the basic add, subtract, multiply, and divide operations and a number of other basic arithmetic operations. Operands may reside in registers or one operand may reside in memory.

Addition

- FADD Add Real
- FADDP Add Real and pop
- FIADD Add Integer

Subtraction

- FSUB Subtract Real
- FSUBP Subtract Real and pop
- FISUB Subtract Integer
- FSUBR Subtract Real reversed
- FSUBRP Subtract Real reversed and pop
- FISUBR Subtract Integer reversed

Multiplication

- FMUL Multiply Real
- FMULP Multiply Real and pop
- FIMUL Multiply Integer

Division

- FDIV Divide Real
- FDIVP Divide Real and pop
- FIDIV Divide Integer
- FDIVR Divide Real reversed
- FDIVRP Divide Real reversed and pop
- FIDIVR Divide Integer reversed

Other Operations

FSQRT	Square Root
FSCALE	Scale
FPREM	Partial Remainder
FPREM1	IEEE standard partial remainder
FRNDINT	Round to Integer
FXTRACT	Extract Exponent and Significand
FABS	Absolute Value
FCHS	Change sign

3.1.3 COMPARISON INSTRUCTION

Instructions of this class allow comparison of numbers of all supported real and integer data types. Each of these instructions analyzes the top stack element often in relationship to another operand and reports the result as a condition code in the status word.

FCOM	Compare Real
FCOMP	Compare Real and pop
FCOMPP	Compare Real and pop twice
FUCOM	Unordered compare Real
FUCOMP	Unordered compare Real and pop
FUCOMPP	Unordered compare Real and pop twice
FICOM	Compare Integer
FICOMP	Compare Integer and pop
FTST	Test
FXAM	Examine

3.1.4 TRANSCENDENTAL INSTRUCTIONS

This group of the Intel387 operations includes trigonometric, inverse trigonometric, logarithmic and exponential functions. The transcendental operate on the top one or two stack elements, and they return their results to the stack. The trigonometric operations assume their arguments are expressed in radians. The logarithmic and exponential operations work in base 2.

FSIN	Sine
FCOS	Cosine
FSINCOS	Sine and cosine
FPTAN	Tangent
FPATAN	Arctangent of ST(1)/ST
F2XM1	$2^X - 1$
FYL2X	$Y * \log_2 X$
FYL2XP1	$Y * \log_2(X + 1)$

3.1.5 LOAD CONSTANT INSTRUCTIONS

Each of these instructions loads (pushes) a commonly used constant onto the stack. The constants have extended real values nearest to the infinitely precise numbers. The only error that can be generated is an Invalid Exception if a stack overflow occurs.

FLDZ	Load $+0.0$
FLD1	Load $+1.0$
FLDPI	Load π
FLDL2T	Load $\log_2 10$
FLDL2E	Load $\log_2 e$
FLDLG2	Load $\log_{10} 2$
FLDLN2	Load $\log_e 2$

3.1.6 PROCESSOR INSTRUCTIONS (ADMINISTRATIVE)

FINIT	Initialize Math CoProcessor
FLDCW	Load Control Word
FSTCW	Store Control Word
FLDCW	Load Status Word
FSTSW	Store Status Word
FSTSW AX	Store Status Word to AX register
FCLEX	Clear Exceptions
FSTENV	Store Environment
FLDENV	Load Environment
FSAVE	Save State
FRSTOR	Restore State
FINCSTP	Increment Stack pointer
FDECSTP	Decrement Stack pointer
FFREE	Free Register
FNOP	No Operation
FWAIT	Report Math CoProcessor Error

3.2 Register Set

Figure 3-1 shows the Intel387 SX Math CoProcessor register set. When a Math CoProcessor is present in a system, programmers may use these registers in addition to the registers normally available on the CPU.

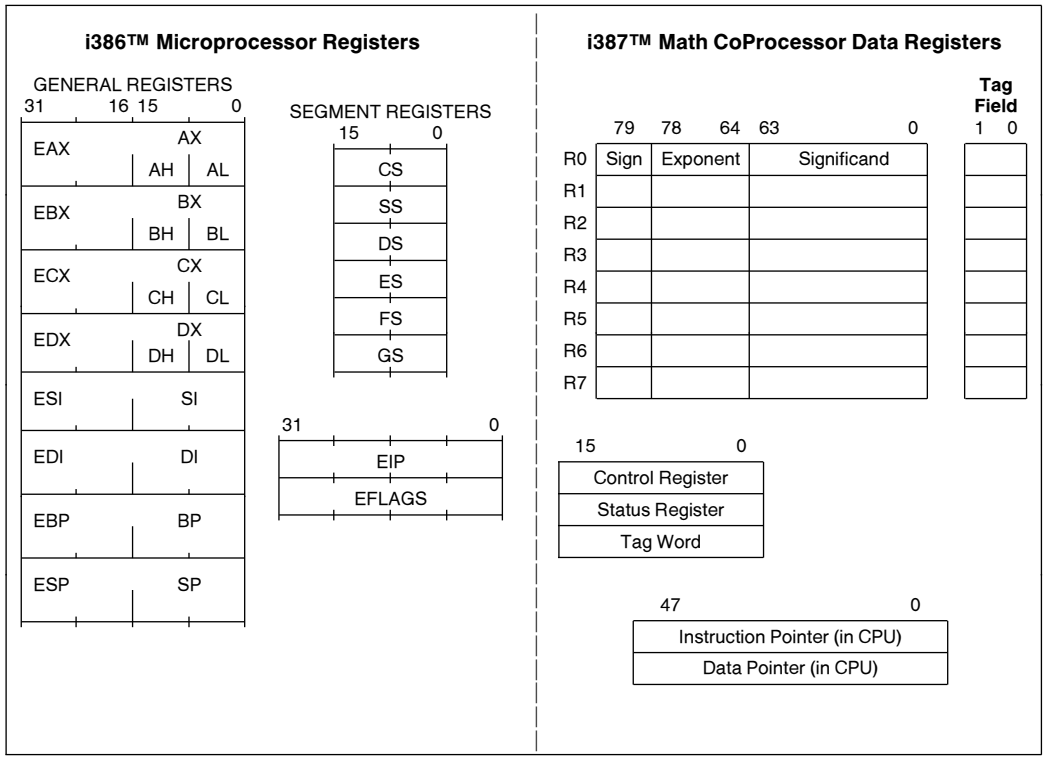


Figure 3-1. Intel386™ CPU and Intel387™ Math CoProcessor Register Set

3.2.1 STATUS WORD (SW) REGISTER

The 16-bit status word (in the status register) shown in Figure 3-2 reflects the overall state of the Math CoProcessor. It can be read and inspected by programs using the FSTSW memory or FSTSW AX instructions.

Bit 15, the Busy bit (B) is included for 8087 compatibility only. It always has the same value as the Error Summary bit (ES, bit 7 of status word); it does not indicate the status of the BUSY# output of the Math CoProcessor.

Bits 13–11 (TOP) serves as the pointer to the Math CoProcessor data register that is the current Top-Of-Stack. The significance of the stack top is described in Section 3.2.5 Data Registers.

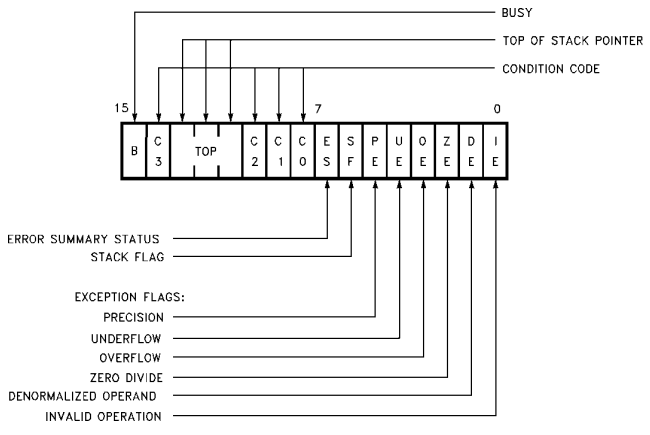
The four numeric condition code bits (C_3–C_0, Bit 14, 10–8) are similar to the flags in a CPU; instructions that perform arithmetic operations update these bits to reflect the outcome. The effects of the instructions on the condition code are summarized in Tables 3-1 through 3-4. These condition code bits are used principally for conditional branching. The FSTSW AX instructions stores the Math CoProcessor status word directly to the CPU AX register, allowing the condition codes to be inspected efficiently by Intel386 CPU code. The Intel386 CPU SAHF instruction can copy C_3–C_0 directly to the flag bits to simplify conditional branching. Table 3-5 shows the mapping of these bits to the Intel386 CPU flag bits.

Bit 7 is the error summary (ES) status bit. This bit is set if any unmasked exception bit is set; it is clear otherwise. If this bit is set, the ERROR# signal is asserted.

Bit 6 is the stack flag (SF). This bit is used to distinguish invalid operations due to stack overflow or underflow from other kinds of invalid operations. When SF is set, bit 9 (C_1) distinguishes between stack overflow (C_1 = 1) or underflow (C_1 = 0).

Bit 5–0 are the six exception flags of the status word and are set to indicate that during an instruction execution the Math CoProcessor has detected one of six possible exception conditions since these status bits were last cleared or reset. Section 3.5 entitled Exception Handling explains how they are set and used.

The exception flags are ''sticky'' bits and can only be cleared by the instructions FINIT, FCLEX, FLDENV, FSAVE, and FRSTOR. Note that when a new value is loaded into the status word by the FLDENV or FRSTOR instruction, the value of ES (bit 7) and B (bit 15) are not derived from the values loaded from memory but rather are dependent upon the values of the exception flags (bits 5–0) in the status word and their corresponding masks in the control word. If ES is set in such a case, the ERROR# output of the Math CoProcessor is activated immediately.



ES is set if any unmasked exception bit is set; cleared otherwise. See Table 2-2 for interpretation of condition code.
TOP values:
000 = Register 0 is Top of Stack
001 = Register 1 is Top of Stack
.
.
.
111 = Register 7 is Top of Stack
For definitions of exceptions, refer to the section entitled ''Exception Handling''

Figure 3-2. Status Word

Table 3-1. Condition Code Interpretation

Instruction	C0 (S)	C3 (Z)	C1 (A)	C2 (C)
FPREM, FPREM1 (see Table 3-2)	Three least significant bits of quotient Q2	Q0	Q1 or O/U#	Reduction 0 = complete 1 = incomplete
FCOM, FCOMP, FCOMPP, FTST, FUCOM, FUCOMP, FUCOMPP, FICOM, FICOMP	Result of comparison (see Table 3-3)		Zero or O/U#	Operand is not comparable (Table 3-3)
FXAM	Operand class (see Table 3-4)		Sign or O/U#	Operand class (Table 3-4)
FCHS, FABS, FXCH, FINCSTP, FDECSTP, Constant loads, FXTRACT, FLD, FILD, FBLD, FSTP (ext real)	UNDEFINED		Zero or O/U#	UNDEFINED
FIST, FBSTP, FRNDINT, FST, FSTP, FADD, FMUL, FDIV, FDIVR, FSUB, FSUBR, FSCALE, FSQRT, FPATAN, F2XM1, FYL2X, FYL2XP1	UNDEFINED		Roundup or O/U#	UNDEFINED
FPTAN, FSIN FCOS, FSINCOS	UNDEFINED		Roundup or O/U#, undefined if C2 = 1	Reduction 0 = complete 1 = incomplete
FLDENV, FRSTOR	Each bit loaded from memory			
FLDCW, FSTENV, FSTCW, FSTSW, FCLEX, FINIT, FSAVE	UNDEFINED			

O/U#	When both IE and SF bits of status word are set, indicating a stack exception, this bit distinguishes between stack overflow (C1 = 1) and underflow (C1 = 0).
Reduction	If FPREM or FPREM1 produces a remainder that is less than the modulus, reduction is complete. When reduction is incomplete the value at the top of the stack is a partial remainder, which can be used as input to further reduction. For FPTAN, FSIN, FCOS, and FSINCOS, the reduction bit is set if the operand at the top of the stack is too large. In this case the original operand remains at the top of the stack.
Roundup	When the PE bit of the status word is set, this bit indicates whether the last rounding in the instruction was upward.
UNDEFINED	Do not rely on finding any specific value in these bits.

Table 3-2. Condition Code Interpretation after FPREM and FPREM1 Instructions

Condition Code				Interpretation after FPREM and FPREM1	
C2	C3	C1	C0		
1	X	X	X	Incomplete Reduction: further interation required for complete reduction	
	Q1	Q0	Q2	Q MOD8	Complete Reduction: C0, C3, C1 contain three least significant bits of quotient
0	0	0	0	0	
	0	1	0	1	
	1	0	0	2	
	1	1	0	3	
	0	0	1	4	
	0	1	1	5	
	1	0	1	6	
	1	1	1	7	

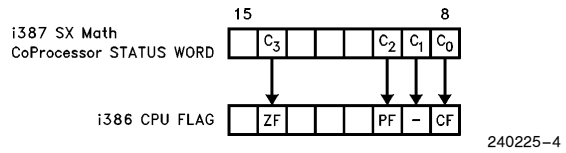
Table 3-3. Condition Code Resulting from Comparison

Order	C3	C2	C0
TOP > Operand	0	0	0
TOP < Operand	0	0	1
TOP = Operand	1	0	0
Unordered	1	1	1

Table 3-4. Condition Code Defining Operand Class

C3	C2	C1	C0	Value at TOP
0	0	0	0	+ Unsupported
0	0	0	1	+ NaN
0	0	1	0	− Unsupported
0	0	1	1	− NaN
0	1	0	0	+ Normal
0	1	0	1	+ Infinity
0	1	1	0	− Normal
0	1	1	1	− Infinity
1	0	0	0	+ 0
1	0	0	1	+ Empty
1	0	1	0	− 0
1	0	1	1	− Empty
1	1	0	0	+ Denormal
1	1	1	0	− Denormal

Table 3-5 Mapping Condition Codes to Intel386™ CPU Flag Bits



3.2.2 CONTROL WORD (CW) REGISTER

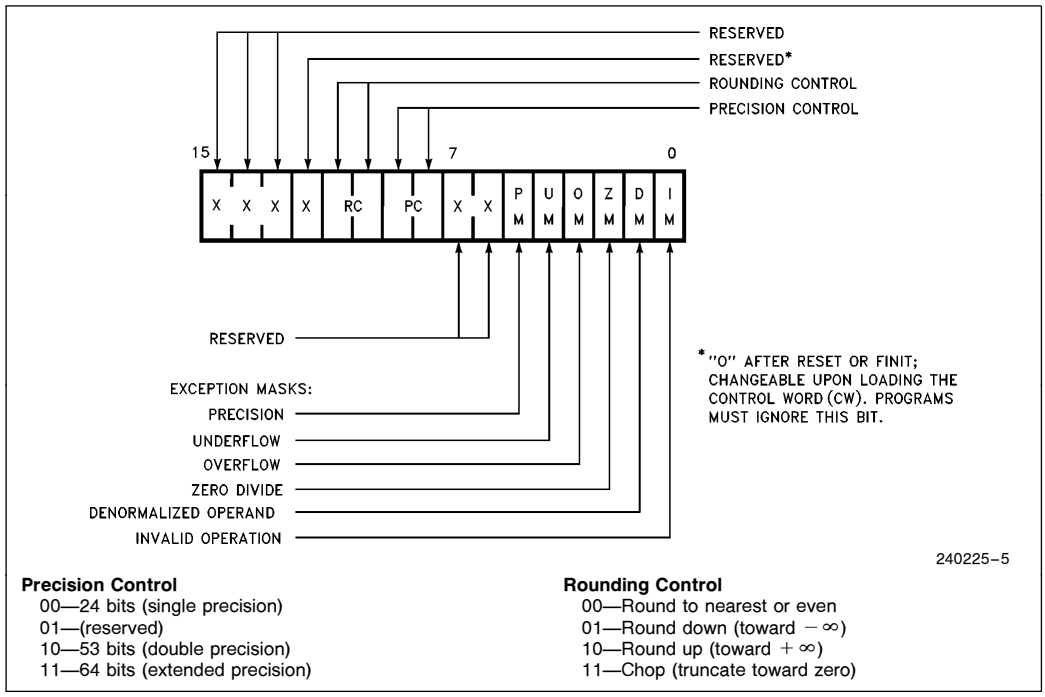
The Math CoProcessor provides the programmer with several processing options that are selected by loading a control word from memory into the control register. Figure 3-3 show the format and encoding of fields in the control word.

The low-order byte of the control word register is used to configure the exception masking. Bits 5–0 of the control word contain individual masks for each of the six exceptions that the Math CoProcessor recognizes. See Section 3.5, Exception Handling, for further explanation on the exception control and definition.

The high-order byte of the control word is used to configure the Math CoProcessor operating mode, including precision, rounding and infinity control.

- The rounding control (RC) field (bits 11–10) provide for directed rounding and true chop, as well as the unbiased round to nearest even mode specified in the IEEE standard. Rounding control affects only those instructions that perform rounding at the end of the operation (and thus can generate a precision exception); namely, FST, FSTP, FIST, all arithmetic instructions (except FPREM, FPREM1, FXTRACT, FABS, and FCHS) and all transcendental instructions.
- The precision control (PC) field (bits 9–8) can be used to set the Math CoProcessor internal operating precision of the significand at less than the default of 64 bits (extended precision). This can be useful in providing compatibility with early generation arithmetic processors of smaller precision. PC affects only the instructions FADD, FSUB(R), FMUL, FDIV(R), and FSQRT. For all other instructions, either the precision is determined by the opcode or extended precision is used.
- The ''infinity control bit'' (bit 12) is not meaningful to the Intel387 SX Math CoProcessor and programs must ignore its value. To maintain compatibility with the 8087 and 80287 (non-387 core), this bit can be programmed, however, regardless of its value the Intel387 SX Math CoProcessor always treats infinity in the affine sense ($-\infty < +\infty$). This bit is initialized to zero both after a hardware reset and after FINIT instruction.

All other bits are reserved and should not be programmed, to assure compatibility with future processors.



Precision Control
00—24 bits (single precision)
01—(reserved)
10—53 bits (double precision)
11—64 bits (extended precision)

Rounding Control
00—Round to nearest or even
01—Round down (toward $-\infty$)
10—Round up (toward $+\infty$)
11—Chop (truncate toward zero)

Figure 3-3. Control Word

3.2.3 DATA REGISTER

Intel387 SX Math CoProcessor data register set consists of eight registers (R0–R7) which are treated as both a stack and a general register file. Each of these data registers in the Math CoProcessor is 80 bits wide and is divided into fields corresponding to the Math CoProcessor's extended-precision real data type, which is used for internal calculations.

The Math CoProcessor register set can be accessed either as a stack, with instructions operating on the top one or two stack elements, or as individually addressable registers. The TOP field in the status word identifies the current top-of-stack register. A ''push'' operation decrements TOP by one and loads a value into the new top register. A ''store and pop'' operation stores the value from the current top register into memory and then increments TOP by one. The Math CoProcessor register stack grows ''down'' toward lower-addressed registers.

Most of the Intel387 SX Math CoProcessor operations use the register stack as the operand(s) and/or as a place to store the result. Instructions may address the data register either implicitly or explicitly. Many instructions operate on the register at the top of the stack. These instructions implicitly address the register at which TOP points. Other instructions allow the programmer to explicitly specify which register to use. Explicit register addressing is also relative to TOP (where ST denotes the current stack top and ST(i) refers to the i'th register from the ST in the stack so the real register address in computed as ST+i).

3.2.4 TAG WORD (TW) REGISTER

The tag word marks the content of each numeric data register, as Figure 3-4 shows. Each two-bit tag represents one of the eight data register. The principal function of the tag word is to optimize the Math CoProcessor's performance and stack handling by making it possible to distinguish between empty and non-empty register locations. It also enables exception handlers to identify special values (e.g. NaNs or denormals) in the contents of a stack location without the need to perform complex decoding of the actual data.

3.2.5 INSTRUCTION AND DATA POINTERS

Because the Math CoProcessor operates in parallel with the CPU, any exceptions detected by the Math CoProcessor may be reported after the CPU has executed the ESC instruction which caused it. To allow identification of the numeric instruction which caused the exception, the Intel386 Microprocessor contains registers that aid in diagnosis. These registers supply the address of the failing instruction and the address of its numeric memory operand (if appropriate).

The instruction and data pointers are provided for user-written exception handlers. These registers are located in the CPU, but appear to be located in the Math CoProcessor because they are accessed by the ESC instructions FLDENV, FSTENV, FSAVE, and FRSTOR; which transfer the values between the registers and memory. Whenever the CPU executes a new ESC instruction (except administrative instructions), it saves the address of the instruction (including any prefixes that may be present), the address of the operand (if present) and the opcode.

The instruction and data pointers appear in one of four formats depending on the operating mode of the CPU (protected mode or real-address mode) and depending on the operand size attribute in effect (32-bit operand or 16-bit operand). (See Figures 3-5, 3-6, 3-7, and 3-8.) Note that the value of the data pointer is *undefined* if the prior ESC instruction did not have a memory operand.

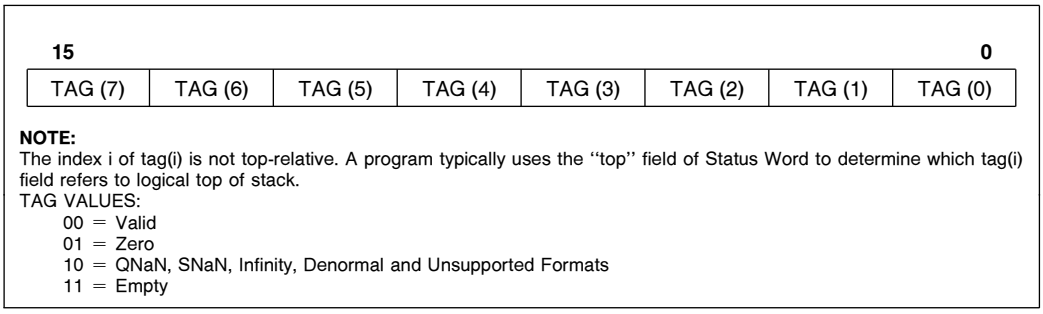


Figure 3-4. Tag Word Register

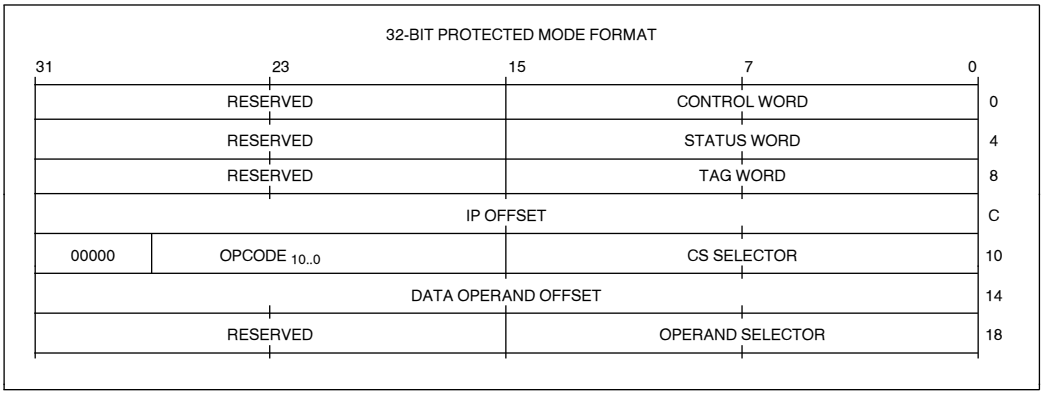


Figure 3-5. Instruction and Data Pointer Image in Memory, 32-Bit Protected-Mode Format

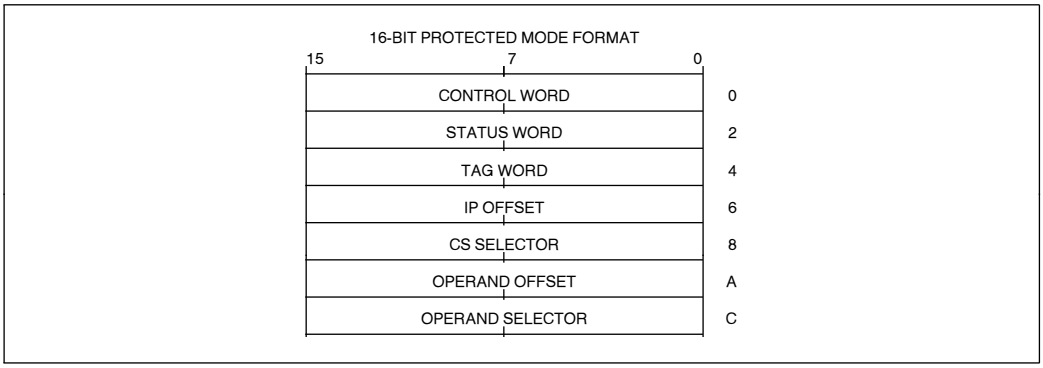


Figure 3-6. Instruction and Data Pointer Image in Memory, 16-Bit Protected-Mode Format

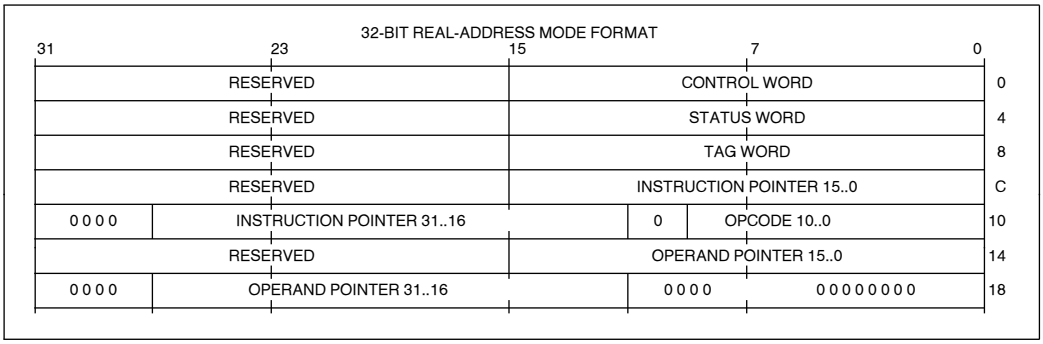


Figure 3-7. Instruction and Data Pointer Image in Memory, 32-Bit Real-Mode Format

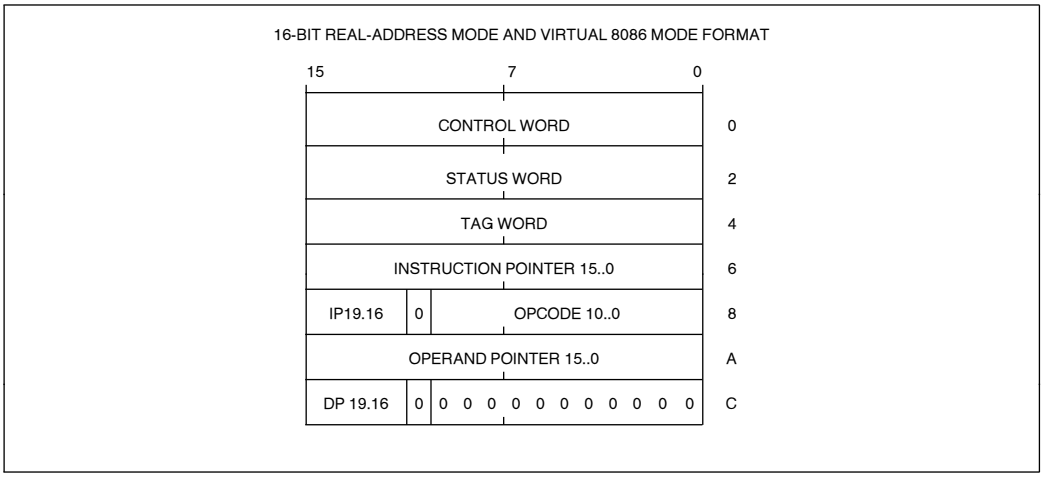


Figure 3-8. Instruction and Data Pointer Image in Memory, 16-Bit Real-Mode Format

3.3 Data Types

Table 3-6 lists the seven data types that the Math CoProcessor supports and presents the format for each type. Operands are stored in memory with the least significant digit at the lowest memory address. Programs retrieve these values by generating the lowest address. For maximum system performance, all operands should start at physical-memory addresses that correspond to the word size of the CPU; operands may begin at any other addresses, but will require extra memory cycles to access the entire operand.

The data type formats can be divided into three classes: binary integer, decimal integer, and binary real. These formats, however, exist in memory only. Internally, the Math CoProcessor holds all numbers in the extended-precision real format. Instructions that load operands from memory automatically convert operands represented in memory as 16, 32, or 64-bit integers, 32 or 64-bit floating point numbers, or 18 digit packed BCD numbers into extended-precision real format. Instructions that store operands in memory perform the inverse type conversion.

In addition to the typical real and integer data values, the Intel387 SX Math CoProcessor data formats encompass encodings for a variety of special values. These special values have significance and can express relevant information about the computations or operations that produced them. The various types of special values are denormal real numbers, zeros, positive and negative infinity, NaNs (Not-a-Number), Indefinite, and unsupported formats. For further information on data types and formats, see the Intel387 Programmer's Reference Manual.

3.4 Interrupt Description

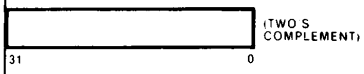
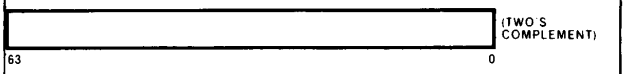
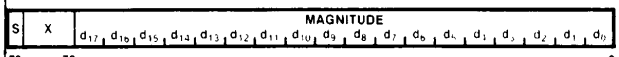
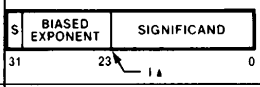
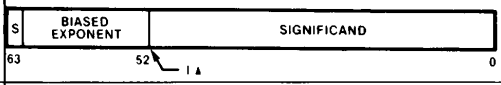
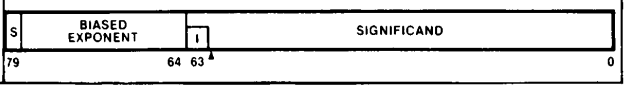
CPU interrupts are used to report errors or exceptional conditions while executing numeric programs in either real or protected mode. Table 3-7 shows these interrupts and their functions.

3.5 Exception Handling

The Math CoProcessor detects six different exception conditions that occur during instruction execution. Table 3-8 lists the exception conditions in order of precedence, showing for each the cause and the default action taken by the Math CoProcessor if the exception is masked by its corresponding mask bit in the control word.

Any exception that is not masked by the control word sets the corresponding exception flag of the status word, sets the ES bit of the status word, and asserts the ERROR# signal. When the CPU attempts to execute another ESC instruction or WAIT, exception 16 occurs. The exception condition must be resolved via an interrupt service routine. The return address pushed onto the CPU stack upon entry to the service routine does not necessarily point to the failing instruction nor to the following instruction. The CPU saves the address of the floating-point instruction that caused the exception and the address of any memory operand required by that instruction.

Table 3-6. Intel387™ SX Math CoProcessor Data Type Representation in Memory

Data Formats	Range	Precision	Most Significant Byte = HIGHEST ADDRESSED BYTE
Word Integer	$\pm 10^4$	16 Bits	
Short Integer	$\pm 10^9$	32 Bits	
Long Integer	$\pm 10^{18}$	64 Bits	
Packed BCD	$\pm 10^{18}$	18 Digits	
Single Precision	$\pm 10^{\pm 38}$	24 Bits	
Double Precision	$\pm 10^{\pm 308}$	53 Bits	
Extended Precision	$\pm 10^{\pm 4932}$	64 Bits	

240225–23

NOTES:
1. S = Sign bit (0 = positive, 1 = negative)
2. d_n = Decimal digit (two per byte)
3. X = Bits have no significance; Math CoProcessor ignores when loading, zeros when storing
4. ▲ = Position of implicit binary point
5. I = Integer bit of significand; stored in temporary real, implicit in single and double precision
6. Exponent Bias (normalized values):
   Single: 127 (7FH)
   Double: 1023 (3FFH)
   Extended REal: 16383 (3FFFH)
7. Packed BCD: $(-1)^S$ $(D_{17}..D_0)$
8. Real: $(-1)^S$ $(2^{E\text{-}BIAS})$ $(F_0$ $F_1...)$

Table 3-7. CPU Interrupt Vectors Reserved for Math CoProcessor

Interrupt Number	Cause of Interrupt
7	An ESC instruction was encountered when EM or TS of CPU control register zero (CR0) was set. EM = 1 indicates that software emulation of the instruction is required. When TS is set, either an ESC or WAIT instruction causes interrupt 7. This indicates that the current Math CoProcessor context may not belong to the current task.
9	In a protected-mode system, an operand of a coprocessor instruction wrapped around an addressing limit (0FFFFH for expand-up segments, zero for expand-down segments) and spanned inaccessible addresses(1). The failing numerics instruction is not restartable. The address of the failing numerics instruction and data operand may be lost; an FSTENV does not return reliable addresses. The segment overrun exception should be handled by executing an FNINIT instruction (i.e., an FINIT without a preceding WAIT). The exception can be avoided by never allowing numerics operands to cross the end of a segment.
13	In a protected-mode system, the first word of a numeric operand is not entirely within the limit of its segment. The return address pushed onto the stack of the exception handler points at the ESC instruction that caused the exception, including any prefixes. The Math CoProcessor has not executed this instruction; the instruction pointer and data pointer register refer to a previous, correctly executed instruction.
16	The previous numerics instruction caused an unmasked exception. The address of the faulty instruction and the address of its operand are stored in the instruction pointer and data pointer registers. Only ESC and WAIT instructions can cause this interrupt. The CPU return address pushed onto the stack of the exception handler points to a WAIT or ESC instruction (including prefixes). This instruction can be restarted after clearing the exception condition in the Math CoProcessor. FNINIT, FNCLEX, FNSTSW, FNSTENV, and FNSAVE cannot cause this interrupt.

NOTE:
1. An operand may wrap around an addressing limit when the segment limit is near an addressing limit and the operand is near the largest valid address in the segment. Because of the wrap-around, the beginning and ending addresses of such an operand will be at opposite ends of the segment. There are two ways that such an operand may also span inaccessible addresses: 1) if the segment limit is not equal to the addressing limit (e.g. addressing limit is FFFFH and segment limit is FFFDH) the operand will span addresses that are not within the segment (e.g. an 8-byte operand that starts at valid offset FFFCH will span addresses FFFC–FFFFH and 0000-0003H; however addresses FFFEH and FFFFH are not valid, because they exceed the limit); 2) if the operand begins and ends in present and accessible segments but intermediate bytes of the operand fall in a not-present page or in a segment or page to which the procedure does not have access rights.

Table 3-8. Intel387™ SX Math CoProcessor Exceptions

Exception	Cause	Default Action (if exception is masked)
Invalid Operation	Operation on a signalling NaN, unsupported format, indeterminate for (0-∞, 0/0, (+∞) + (−∞), etc.), or stack overflow/underflow (SF is also set).	Result is a quiet NaN, integer indefinite, or BCD indefinte
Denormalized Operand	At least one of the operands is denormalized, i.e., it has the smallest exponent but a nonzero significand.	Normal processing continues
Zero Divisor	The divisor is zero while the dividend is a noninfinite, nonzero number.	Result is ∞
Overflow	The result is too large in magnitude to fit in the specified format.	Result is largest finite value or ∞
Underflow	The true result is nonzero but too small to be represented in the specified format, and, if underflow exception is masked, denormalization causes the loss of accuracy.	Result is denormalized or zero
Inexact Result (Precision)	The true result is not exactly representable in the specified format (e.g. 1/3); the result is rounded according to the rounding mode.	Normal processing continues

3.6 Initialization

After FNINIT or RESET, the control word contains the value 037FH (all exceptions masked, precision control 64 bits, rounding to nearest) the same values as in an Intel287 after RESET. For compatibility with the 8087 and Intel287, the bit that used to indicate infinity control (bit 12) is set to zero; however, regardless of its setting, infinity is treated in the affine sense. After FNINIT or RESET, the status word is initialized as follows:

- All exceptions are set to zero.
- Stack TOP is zero, so that after the first push the stack top will be register seven (111B).
- The condition code C_3–C_0 is undefined.
- The B-bit is zero.

The tag word contains FFFFH (all stack locations are empty).

The Intel386 Microprocessor and Intel387 Math CoProcessor initialization software must execute a FNINIT instruction (i.e., FINIT without a preceding WAIT) after RESET. The FNINIT is not strictly required for the Intel386 software, but Intel recommends its use to help ensure upware compatibility with other processors. After a hardware RESET, the ERROR# output is asserted to indicate that an Intel387 Math CoProcessor is present. To accomplish this, the IE (Invalid Exception) and ES (Error Summary) bits of the status word are set, and the IM bit (Invalid Exception Mask) in the control word is cleared. After FNINIT, the status word and the control word have the same values as in an Intel287 Math CoProcessor after RESET.

3.7 Processing Modes

The Intel387 SX Math CoProcessor works the same whether the CPU is executing in real-addressing mode, protected mode, or virtual-8086 mode. All references to memory for numerics data or status information are performed by the CPU, and therefore obey the memory-management and protection rules of the CPU mode currently in effect. The Intel387 SX Math CoProcessor merely operates on instructions and values passed to it by the CPU and therefore is not sensitive to the processing mode of the CPU.

The real-address mode and virtual-8086 mode, the Intel387 SX Math CoProcessor is completely upward compatible with software for the 8086/8087 and 80286/80287 real-address mode systems.

In protected mode, the Intel387 SX Math CoProcessor is completely upward compatible with software for the 80286/80287 protected mode system.

The only differences of operation that may appear when 8086/8087 programs are ported to the protected mode (not using virtual-8086 mode) is in the format of operands for the administrative instructions FLDENV, FSTENV, FRSTOR, and FSAVE.

3.8 Programming Support

Using the Intel387 SX Math CoProcessor requires no special programming tools, because all new instructions and data types are directly supported by the assembler and compilers for high-level languages. All Intel386 Microprocessor development tools that support Intel387 Math CoProcessor programs can also be used to develop software for the Intel386 SX Microprocessors and Intel387 SX Math CoProcessors. All 8086/8088 development tools that support the 8087 can also be used to develop software for the CPU and Math CoProcessor in real-address mode or virtual-8086 mode. All 80286 development tools that support the Intel287 Math CoProcessor can also be used to develop software for the Intel386 CPU and Intel387 Math CoProcessor.

4.0 HARDWARE SYSTEM INTERFACE

In the following description of hardware interface, the # symbol at the end of a signal name indicates that the active or asserted state occurs when the signal is at a low voltage. When no # is present after the signal name, the signal is asserted when at the high voltage level.

4.1 Signal Description

In the following signal descriptions, the Intel387 SX Math CoProcessor pins are grouped by function as shown by Table 4-1. Table 4-1 lists every pin by its identifier, gives a brief description and lists some of its characteristics (Refer to Figure 1-1 and Table 1-1 for pin configuration).

All output signals can be tri-stated by driving STEN inactive. The output buffers of the bi-directional data pins D15–D0 are also tri-state; they only leave the floating state during read cycles when the Math CoProcessor is selected.

4.1.1 Intel386 CPU CLOCK 2 (CPUCLK2)

This input uses the CLK2 signal of the CPU to time the bus control logic. Several other Math CoProcessor signals are referenced to the rising edge of this signal. When CKM = 1 (synchronous mode) this pin also clocks the data interface and control unit and the floating point unit of the Math CoProcessor. This pin requires CMOS-level input. The signal on this pin is divided by two to produce the internal clock signal CLK.

4.1.2 Intel387 MATH COPROCESSOR CLOCK 2 (NUMCLK2)

When CKM = 0 (asynchronous mode), this pin provides the clock for the data interface and control unit and the floating point unit of the Math CoProcessor. In this case, the ratio of the frequency of NUMCLK2 to the frequency of CPUCLK2 must lie within the range 10:16 to 14:10 and the maximum frequency must not exceed the device specifications. When CKM = 1 (synchronous mode), signals on this pin are ignored: CPUCLK2 is used instead for the data interface and control unit and the floating point unit. This pin requires CMOS level input and should be tied low if not used.

Table 4-1. Pin Summary

Pin Name	Function	Active State	Input/ Output	Referenced To . . .
Execution Control				
CPUCLK2	Microprocessor Clock2		I	
NUMCLK2	Math CoProcessor Clock2		I	
CKM	Math CoProcessor Clock Mode		I	
RESETIN	System Reset	High	I	CPUCLK2
Math CoProcessor Handshake				
PEREQ	Processor Request	High	O	CPUCLK2
BUSY#	Busy Status	Low	O	CPUCLK2
ERROR#	Error Status	Low	O	NUMCLK2
Bus Interface				
D15–D0	Data Pins		I/O	CPUCLK2
W/R#	Write/Read Bus Cycle	High/Low	I	CPUCLK2
ADS#	Address Strobe	Low	I	CPUCLK2
READY#	Bus Ready Input	Low	I	CPUCLK2
READYO#	Ready Output	Low	O	CPUCLK2
Chip/Port Select				
STEN	Status Enable	High	I	CPUCLK2
NPS1#	Numerics Select #1	Low	I	CPUCLK2
NPS2	Numerics Select #2	High	I	CPUCLK2
CMD0#	Command	Low	I	CPUCLK2
Power and Ground				
V_{CC}	System Power			
V_{SS}	System Ground			

4.1.3 CLOCKING MODE (CKM)

This pin is strapping option. When it is strapped to V_{CC} (HIGH), the Math CoProcessor operates in synchronous mode; when strapped to V_{SS} (LOW), the Math CoProcessor operates in asynchronous mode. These modes relate to clocking of the internal data interface and control unit and the floating point unit only; the bus control logic always operates synchronously with respect to the CPU.

Synchronous mode requires the use of only one clock, the CPU's CLK2. Use of synchronous mode eliminates one clock generator from the board design and is recommended for all designs. Synchronous mode also allows the internal Power Management Unit to enable the idle and standby power saving modes.

Asynchronous mode can provide higher performance of the floating point unit by running a faster clock on NUMCLK2. (The CPU's CLK2 must still be connected to CPUCLK2 input.) This allows the floating point unit to run up to 40% faster than in synchronous mode. Internal power management is disabled in asynchronous mode.

4.1.4 SYSTEM RESET (RESETIN)

A LOW to HIGH transition on this pin causes the Math CoProcessor to terminate its present activity and to enter a dormant state. RESETIN must remain active (HIGH) for at least 40 CPUCLK2 (NUMCLK2 if CKM = 0) periods.

The HIGH to LOW transitions of RESETIN must be synchronous with CPUCLK2, so that the phase of the internal clock of the bus control logic (which is the CPUCLK2 divided by two) is the same as the phase of the internal clock of the CPU. After RESETIN goes LOW, at least 50 CPUCLK2 (NUMCLK2 if CKM = 0) periods must pass before the first Math CoProcessor instruction is written into the Math CoProcessor. This pin should be connected to the CPU RESET pin. Table 4-2 shows the status of the output pins during the reset sequence. After a reset, all output pins return to their inactive state except for ERROR# which remains active (for CPU recognition) until cleared.

Table 4-2. Output Pin Status during Reset

Pin Value	Pin Name
HIGH	READYO#, BUSY#
LOW	PEREQ, ERROR#
Tri-State OFF	D15–D0

4.1.5 PROCESSOR REQUEST (PEREQ)

When active, this pin signals to the CPU that the Math CoProcessor is ready for data transfer to/from its data FIFO. When all data is written to or read from the data FIFO, PEREQ is deactivated. This signal always goes inactive before BUSY# goes inactive. This signal is reference to CPUCLK2. It should be connected to the CPU PEREQ input pin.

4.1.6 BUSY STATUS (BUSY#)

When active, this pin signals to the CPU that the Math CoProcessor is currently executing an instruction. This signal is referenced to CPUCLK2. It should be connected to the CPU BUSY# input pin.

4.1.7 ERROR STATUS (ERROR#)

This pin reflects the ES bit of the status register. When active, it indicates that an unmasked exception has occurred. This signal can be changed to the inactive state only by the following instructions (without a preceding WAIT); FNINIT, FNCLEX, FNSTENV, FNSAVE, FLDCW, FLDENV, and FRSTOR. ERROR# is driven active during RESET to indicate to the CPU that the Math CoProcessor is present. This pin is referenced to NUMCLK2 (or CPUCLK2 if CKM = 1). It should be connected to the ERROR# pin of the CPU.

4.1.8 DATA PINS (D15–D0)

These bi-directional pins are used to transfer data and opcodes between the CPU and Math CoProcessor. They are normally connected directly to the corresponding CPU data pins. HIGH state indicates a value of one. D0 is the least significant data bit. Timings are referenced to rising edge of CPUCLK2.

4.1.9 WRITE/READ BUS CYCLE (W/R#)

This signal indicates to the Math CoProcessor whether the CPU bus cycle in progress is a read or a write cycle. This pin should be connected directly to the CPU's W/R# pin. HIGH indicates a write cycle to the Math CoProcessor; LOW a read cycle from the Math CoProcessor. This input is ignored if any of the signals STEN, NPS1#, or NPS2 are inactive. Setup and hold times are referenced to CPUCLK2.

4.1.10 ADDRESS STROBE (ADS#)

This input, in conjunction with the READY# input, indicates when the Math CoProcessor bus control logic may sample W/R# and the chip select signals. Setup and hold times are referenced to CPUCLK2. This pin should be connected to the ADS# pin of the CPU.

4.1.11 BUS READY INPUT (READY#)

This input indicates to the Math CoProcessor when a CPU bus cycle is to be terminated. It is used by the bus control logic to trace bus activities. Bus cycles can be extended indefinitely until terminated by READY#. This input should be connected to the same signal that drives the CPU's READY# input. Setup and hold times are referenced to CPUCLK2.

4.1.12 READY OUTPUT (READYO#)

This pin is activated at such a time that write cycles are terminated after two clocks (except FLDENV and FRSTOR) and read cycles after three clocks. In configurations where no extra wait states are required, this pin must directly or indirectly drive the READY# input of the CPU. Refer to the section entitled "BUS OPERATION" for details. This pin is activated only during bus cycles that select the Math CoProcessor. This signal is referenced to CPUCLK2.

(FLDENV and FRSTOR require data transfers larger than the FIFO. Therefore, PEREQ is activated for the duration of transferring 2 words of 32 bits and then deactivated until the FIFO is ready to accept two additional words. The length of the write cycles of the last operand word in each transfer as well as the first operand word transfer of the entire instruction is 3 clocks instead of 2 clocks. This is done to give the Intel386 CPU enough time to sample PEREQ and to notice that the Intel387 is **not** ready for additional transfers.)

4.1.13 STATUS ENABLE (STEN)

This pin serves as a chip select for the Math CoProcessor. When inactive, this pin forces BUSY#, PEREQ, ERROR# and READYO# outputs into a floating state. D15–D0 are normally floating and will leave the floating state only if STEN is active and additional conditions are met (read cycle). STEN also causes the chip to recognize its other chip select inputs. STEN makes it easier to do on-board testing (using the overdrive method) of other chips in systems containing the Math CoProcessor. STEN should be pulled up with a resistor so that it can be pulled down when testing. In boards that do not use on-board testing STEN should be connected to V_{CC}. Setup and hold times are relative to CPUCLK2. Note that STEN must maintain the same setup and hold times as NPS1#, NPS2, and CMD0# (i.e., if STEN changes state during a Math CoProcessor bus cycle, it must change state during the same CLK period as the NPS1#, NPS2, and CMD0# signals).

4.1.14 MATH COPROCESSOR SELECT 1 (NPS1#)

When active (along with STEN and NPS2) in the first period of a CPU bus cycle, this signal indicates that the purpose of the bus cycle is to communicate with the Math CoProcessor. This pin should be connected directly to the M/IO# pin of the CPU, so that the Math CoProcessor is selected only when the CPU performs I/O cycles. Setup and hold times are referenced to the rising edge of CPUCLK2.

4.1.15 MATH COPROCESSOR SELECT 2 (NPS2)

When active (along with STEN and NPS1#) in the first period of a CPU bus cycle, this signal indicates that the purpose of the bus cycle is to communicate with the Math CoProcessor. This pin should be connected directly to the A23 pin of the CPU, so that the Math CoProcessor is selected only when the CPU issues one of the I/O addresses reserved for the Math CoProcessor (8000F8h, 8000FCh, or 8000FEh which is treated as 8000FCh by the Math CoProcessor). Setup and hold times are referenced to the rising edge of CPUCLK2.

4.1.16 COMMAND (CMD0#)

During a write cycle, this signal indicates whether an opcode (CMD0# active low) or data (CMD0# inactive high) is being sent to the Math CoProcessor. During a read cycle, it indicates whether the control or status register (CMD0# active) or a data register (CMD0#) is being read. CMD0# should be connected directly to the A2 output of the CPU. Setup and hold times are referenced to the rising edge of CPUCLK2 at the end of PH2.

4.1.17 SYSTEM POWER (V_{CC})

System power provides the +5V DC supply input. All V_{CC} pins should be tied together on the circuit board and local decoupling capacitors should be used between V_{CC} and V_{SS}.

4.1.18 SYSTEM GROUND (V_{SS})

System ground provides the 0V connection from which all inputs and outputs are measured. All V_{SS} pins should be tied together on the circuit board and local decoupling capacitors should be used between V_{CC} and V_{SS}.

4.2 System Configuration

The Intel387 SX Math CoProcessor is designed to interface with the Intel386 SX Microprocessor as shown by Figure 4-1. A dedicated communication protocol makes possible high-speed transfer of op-codes and operands between the CPU and Math CoProcessor. The Intel387 SX Math CoProcessor is designed so that no additional components are required for interface with the CPU. Most control pins of the Math CoProcessor are connected directly to pins of the CPU.

The interface between the Math CoProcessor and the CPU has these characteristics:

- The Math CoProcessor shares the local bus of the Intel386 SX Microprocessor.
- The CPU and Math CoProcessor share the same reset signals. They may also share the same clock input; however, for greatest performance, an external oscillator may be needed.
- The corresponding Busy#, ERROR#, and PEREQ pins are connected together.
- The Math CoProcessor NPS1# and NPS2 inputs are connected to the latched CPU M/IO# and A23 outputs respectively. For Math CoProcessor cycles, M/IO# is always LOW and A23 always HIGH.
- The Math CoProcessor input CMD0 is connected to the latched A_2 output. The Intel386 SX Microprocessor generates address 8000F8H when writing a command and address 8000FCH or 8000FEH (treated as 8000FCH by the Intel387 SX Math CoProcessor) when writing or reading data. It does not generate any other addresses during Math CoProcessor bus cycles.

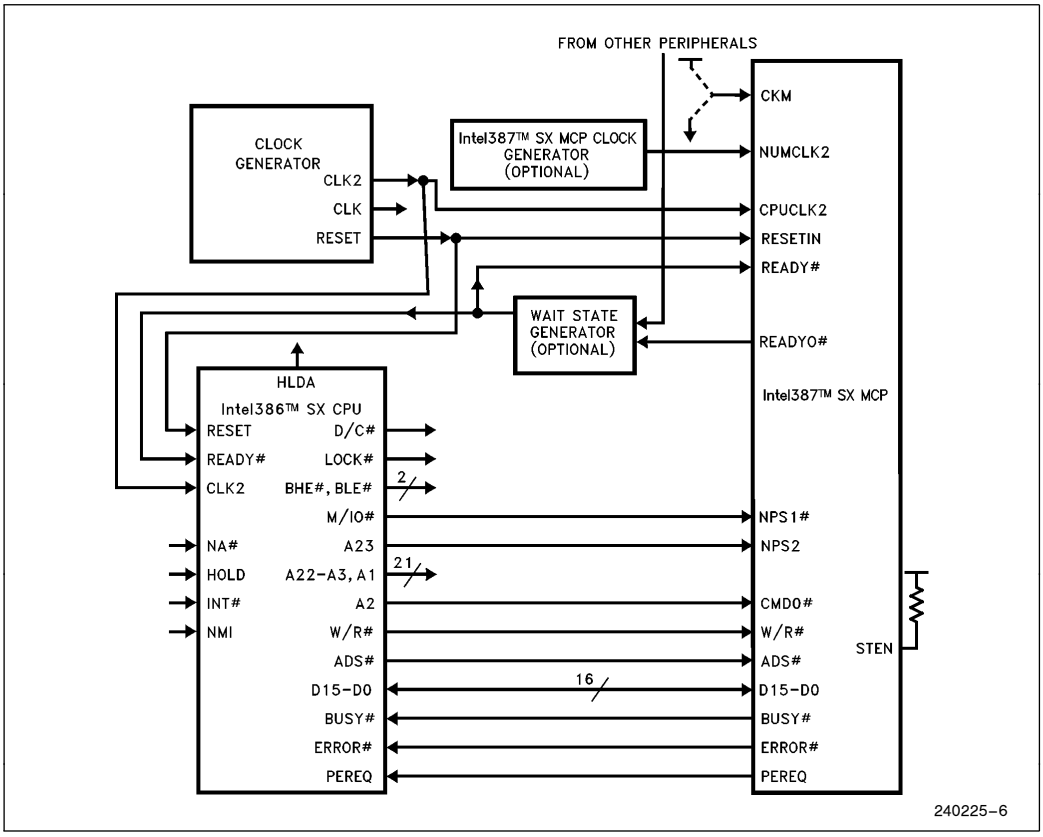


Figure 4-1. Intel386™ SX CPU and Intel387™ SX Math CoProcessor System Configuration

4.3 Math CoProcessor Architecture

As shown in Figure 2-1 Block Diagram, the Intel387 SX Math CoProcessor is internally divided into four sections; the Bus Control Logic (BCL), the Data Interface and Control Logic, the Floating Point Unit (FPU), and the Power Management Unit (PMU). The Bus Control Logic is responsible for the CPU bus tracking and interface. The BCL is the only unit in the Math CoProcessor that must run synchronously with the CPU; the rest of the Math CoProcessor can run asynchronously with respect to the CPU. The Data Interface and Control Unit is responsible for the data flow to and from the FPU and the control registers, for receiving the instructions, decoding them, sequencing the microinstructions, and for handling some of the administrative instructions. The Floating Point Unit (with the support of the control unit which contains the sequencer and other support units) executes the mathematical instructions. The Power Manager is new to the Intel387 family. It is responsible for shutting down idle sections of the device to save power.

4.3.1 BUS CONTROL LOGIC

The BCL communicates solely with the CPU using I/O bus cycles. The BCL appears to the CPU as a special peripheral device. It is special in two respects: the CPU initiates I/O automatically when it encounters ESC instructions, and the CPU uses reserved I/O addresses to communicate with the BCL. The BCL does not communicate directly with memory. The CPU performs all memory access, transferring input operands from the memory to the Math CoProcessor and transferring outputs from the Math CoProcessor to memory.

4.3.2 DATA INTERFACE AND CONTROL UNIT

The data interface and control unit latches the data and, subject to BCL control, directs the data to the FIFO or the instruction decoder. The instruction decoder decodes the ESC instructions sent to it by the CPU and generates controls that direct the data flow in the FIFO. It also triggers the microinstruction sequencer that controls execution of each instruction. If the ESC instruction is FINIT, FCLEX, FSTSW, FSTSW AX, FSTCW, FSETPM, or FRSTPM, the control unit executes it independently of the FPU and the sequencer. The data interface and control unit is the unit that generates the BUSY#, PEREQ, and ERROR# signals that synchronize the Math CoProcessor activities with the CPU.

4.3.3 FLOATING POINT UNIT

The FPU executes all instructions that involve the register stack, including arithmetic, logical, transcendental, constant, and data transfer instructions. The data path in the FPU is 84 bits wide (68 significant bits, 15 exponent bits, and a sign bit) which allows internal operand transfers to be performed at very high speeds.

4.3.4 POWER MANAGEMENT UNIT

The Power Management Unit (PMU) controls all internal power savings circuits. When the Math CoProcessor is not executing an instruction, the PMU disables the internal clock to the FPU, Control Unit, and Data Interface within three clocks. The Bus Control Logic remains enabled to accept the next instruction. Upon decode of a valid Math CoProcessor bus cycle, the PMU enables the internal clock to all circuits. No loss in performance occurs.

4.4 Bus Cycles

All bus cycles are initiated by the CPU. The pins STEN, NPS1#, NPS2, CMD0, and W/R# identify bus cycles for the Math CoProcessor. Table 4-3 defines the types of Math CoProcessor bus cycles.

Table 4-3. Bus Cycle Definition

STEN	NPS1#	NPS2	CMD0#	W/R#	Bus Cycle Type
0	X	X	X	X	Math CoProcessor not selected and all outputs in floating state
1	1	X	X	X	Math CoProcessor not selected
1	X	0	X	X	Math CoProcessor not selected
1	0	1	0	0	CW or SW read from Math CoProcessor
1	0	1	0	1	Opcode write to Math CoProcessor
1	0	1	1	0	Data read from Math CoProcessor
1	0	1	1	1	Data write to Math CoProcessor

4.4.1 INTEL387 SX MATH COPROCESSOR ADDRESSING

The NPS1#, NPS2, and CMD0 signals allow the Math CoProcessor to identify which bus cycles are intended for the Math CoProcessor. The Math CoProcessor responds to I/O cycles when the I/O address is 8000F8h, 8000FCh, and 8000FEh (treated as 8000FCh). The Math CoProcessor responds to I/O cycles when bit 23 of the I/O address is set. In other words, the Math CoProcessor acts as an I/O device in a reserved I/O address space.

Because A23 is used to select the Intel387 SX Math CoProcessor for data transfers, it is not possible for a program running on the CPU to address the Math CoProcessor with an I/O instruction. Only ESC instructions cause the CPU to communicate with the Math CoProcessor.

4.4.2 CPU/MATH COPROCESSOR SYNCHRONIZATION

The pins BUSY#, PEREQ, and ERROR# are used for various aspects of synchronization between the CPU and the Math CoProcessor.

BUSY# is used to synchronize instruction transfer from the CPU to the Math CoProcessor. When the Math CoProcessor recognizes an ESC instruction it asserts BUSY#. For most ESC instructions, the CPU waits for the Math CoProcessor to deassert BUSY# before sending the new opcode.

The Math CoProcessor uses the PEREQ pin of the CPU to signal that the Math CoProcessor is ready for data transfer to or from its data FIFO. The Math CoProcessor does not directly access memory; rather, the CPU provides memory access services for the Math CoProcessor. (For this reason, memory access on behalf of the Math CoProcessor always obeys the protection rules applicable to the current CPU mode.) Once the CPU initiates an Math CoProcessor instruction that has operands, the CPU waits for PEREQ signals that indicate when the Math CoProcessor is ready for operand transfer. Once all operands have been transferred (or if the instruction has no operands) the CPU continues program execution while the Math CoProcessor executes the ESC instruction.

In 8087/8087 systems, WAIT instructions may be required to achieve synchronization of both commands and operands. In the Intel386 Microprocessor and Intel387 Math CoProcessor systems, however, WAIT instructions are required only for operand synchronization; namely, after Math CoProcessor stores to memory (except FSTSW and FSTCW) or load from memory. (In 80286/80287 systems, WAIT is required before FLDENV and FRSTOR.) Used this way, WAIT ensures that the value has already been written or read by the Math CoProcessor before the CPU reads or changes the value.

Once it has started to execute a numerics instruction and has transferred and operands from the CPU, the Math CoProcessor can process the instruction in parallel with and independent of the host CPU. When the Math CoProcessor detects an exception, it asserts the ERROR# signal, which causes a CPU interrupt.

4.4.3 SYNCHRONOUS/ASYNCHRONOUS MODES

The internal logic of the Math CoProcessor can operate either directly from the CPU clock (synchronous mode) or from a separate clock (asynchronous mode). The two configurations are distinguished by the CKM pin. In either case, the bus control logic (BCL) of the Math CoProcessor is synchronized with the CPU clock. Use of asynchronous mode allows the BCL and the FPU section of the Math CoProcessor to run at different speeds. In this case, the ratio of the frequency of NUMCLK2 to the frequency of CPUCLK2 must lie within the range 10:16 to 14:10. Use of synchronous mode eliminates one clock generator from the board design. The internal Power Management Unit of the Intel387 SX Math CoProcessor is disabled in asynchronous mode.

4.4.4 AUTOMATIC BUS CYCLE TERMINATION

In configurations where no extra wait states are required, READYO# can drive the CPU's READY# input and the Math CoProcessors READY# input. If wait states are required, this pin should be connected to the logic that ORs all READY outputs from peripheral devices on the CPU bus. READYO# is asserted by the Math CoProcessor only during I/O cycles that select the Math CoProcessor. Refer to Section 5.0 Bus Operation for details.

5.0 BUS OPERATION

With respect to bus interface, the Intel387 SX Math CoProcessor is fully synchronous with the CPU. Both operate at the same rate because each generates its internal CLK signal by dividing CPUCLK2 by two. Furthermore, both internal CLK signals are in phase, because they are synchronized by the same RESETIN signal.

A bus cycle for the Math CoProcessor starts when the CPU activates ADS# and drives new values on the address and cycle definition lines (W/R#, M/IO#, etc.). The Math CoProcessor examines the address and cycle definition lines in the same CLK period during which ADS# is activated. This CLK period is considered the first CLK of the bus cycle.

During this first CLK period, the Math CoProcessor also examines the W/R# input signal to determine whether the cycle is a read or a write cycle and examines the CMD0# input to determine whether an opcode, operand, or control/status register transfer is to occur.

The Intel387 SX Math CoProcessor supports both pipelined (i.e., overlapped) and non-pipelined bus cycles. A non-pipelined cycle is one for which the CPU asserts ADS# when no other bus cycle is in progress. A pipelined bus cycle is one for which the CPU asserts ADS# and provides valid next address and control signals before the prior Math CoProcessor cycle terminates. The CPU may do this as early as the second CLK period after asserting ADS# for the prior cycle. Pipelining increases the availability of the bus by at least one CLK period. The Intel387 SX Math CoProcessor supports pipelined bus cycles in order to optimize address pipelining by the CPU for memory cycles.

Bus operation is described in terms of an abstract state machine. Figure 5-1 illustrates the states and state transitions for Math CoProcessor bus cycles:

- T_I is the idle state. This is the state of the bus logic after RESET, the state to which bus logic returns after every non-pipelined bus cycle, and the state to which bus logic returns after a series of pipelined cycles.
- T_{RS} is the READY# sensitive state. Different types of bus cycles may require a minimum of one or two successive T_{RS} states. The bus logic remains in T_{RS} state until READY# is sensed, at which point the bus cycle terminates. Any number of wait states may be implemented by delaying READY#, thereby causing additional successive T_{RS} states.
- T_P is the first state for every pipelined bus cycle. This state is not used by non-pipelined cycles.

Note that the bus logic tracks bus state regardless of the values on the chip/port select pins. The

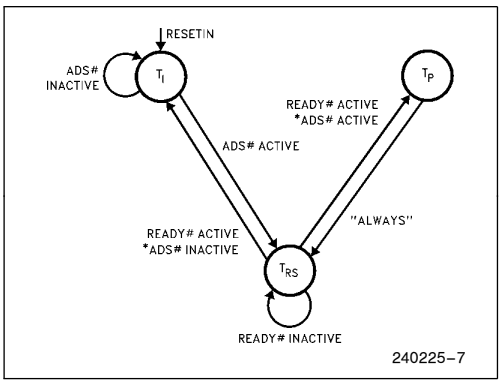


Figure 5-1. Bus State Diagram

READYO# output of the Math CoProcessor indicates when a Math CoProcessor bus cycle may be terminated if no extra wait states are required. For all write cycles (except those for the instructions FLDENV and FRSTOR), READYO# is always asserted during the first T_{RS} state, regardless of the number of wait states. For all read cycles (and write cycles for FLDENV and FRSTOR), READY# is always asserted in the second T_{RS} state, regardless of the number of wait states. These rules apply to both pipelined and non-pipelined cycles. Systems designers may use READYO# in one of the following ways:

1. Connect it (directly or through logic that ORs READY# signals from other devices) to the READY# inputs of the CPU and Math CoProcessor.
2. Use it as one input to a wait-state generator.

The following sections illustrate different types of Intel387 SX Math CoProcessor bus cycles. Because different instructions have different amounts of overhead before, between, and after operand transfer cycles, it is not possible to represent in a few diagrams all of the combinations of successive operand transfer cycles. The following bus cycle diagrams show memory cycles between Math CoProcessor operand transfer cycles. Note however that, during FRSTOR, some consecutive accesses to the Math CoProcessor do not have intervening memory accesses. For the timing relationship between operand transfer cycles and opcode write or other overhead activities, see Figure 7-7 ''Other Parameters''.

5.1 Non-Pipelined Bus Cycles

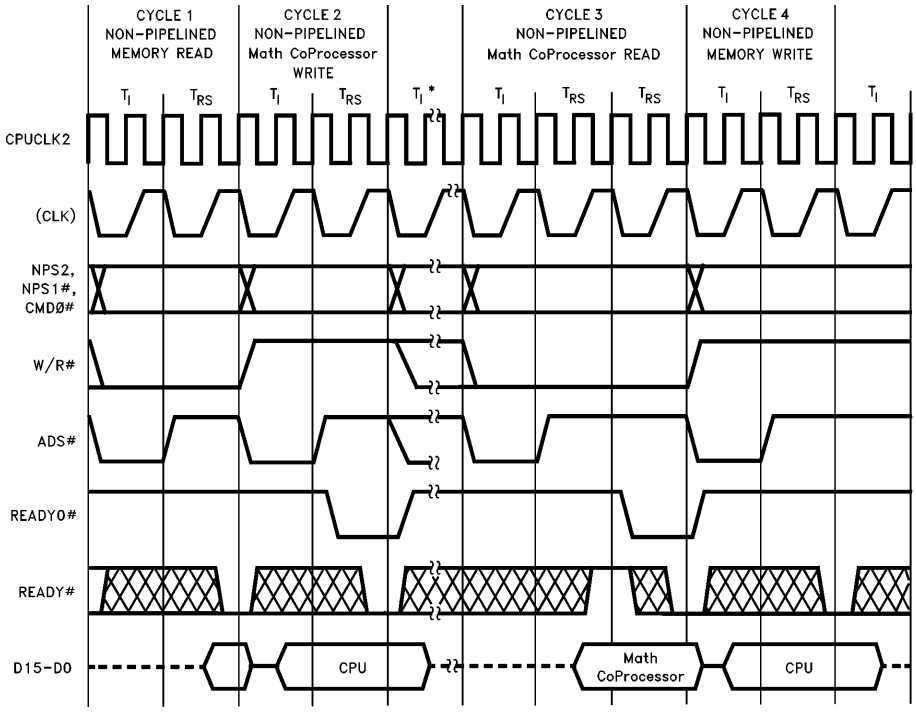
Figure 5-2 illustrates bus activity for consecutive non-pipelined bus cycles.

At the second clock of the bus cycle, the Math CoProcessor enters the T_{RS} state. During this state, it samples the READY# input and stays in this state as long as READY# is inactive.

5.1.1 WRITE CYCLE

In write cycles, the Math CoProcessor drives the READYO# signal for one CLK period during the second CLK period of the cycle (i.e., the first T_{RS} state); therefore, the fastest write cycle takes two CLK periods (see cycle 2 of Figure 5-2). For the instructions FLDENV and FRSTOR, however, the Math CoProcessor forces wait state by delaying the activation of READYO# to the second T_{RS} state (not shown in Figure 5-2).

The Math CoProcessor samples the D15–D0 inputs into data latches at the falling edge of CLK as long as it stays in T_{RS} state.



Cycles 1 & 2 represent part of the operand transfer cycle for instructions involving either 4-byte or 8-byte operand loads.
Cycles 3 & 4 represent part of the operand transfer cycle for a store operation.
*Cycles 1 & 2 could repeat here or T_I states for various non-operand transfer cycles and overhead.

Figure 5-2. Non-Pipelined Read and Write Cycles

When READY# is asserted, the Math CoProcessor returns to the idle state. Simultaneously with the Math CoProcessor entering the idle state, the CPU may assert ADS# again, signaling the beginning of yet another cycle.

5.1.2 READ CYCLE

At the rising edge of CLK in the second CLK period of the cycle (i.e., the first T_{RS} state), the Math CoProcessor starts to drive the D15–D0 outputs and continues to drive them as long as it stays in T_{RS} state.

At least one wait state must be inserted to ensure that the CPU latches the correct data. Because the Math CoProcessor starts driving the data bus only at the rising edge of CLK in the second clock period of the bus cycle, not enough time is left for the data signals to propagate and be latched by the CPU before the next falling edge of CLK. Therefore, the Math CoProcessor does not drive the READYO# signal until the third CLK period of the cycle. Thus, if the READYO# output drives the CPU's READY# input, one wait state is automatically inserted.

Because one wait state is required for Math CoProcessor reads, the minimum length of a Math CoProcessor read cycle is three CLK periods, as cycle 3 of Figure 5-2 shows.

When READY# is asserted, the Math CoProcessor returns to the idle state. Simultaneously with the Math CoProcessor's entering the idle state, the CPU may assert ADS# again, signaling the beginning of yet another cycle. The transition from T_{RS} state to idle state causes the Math CoProcessor to put the D15–D0 outputs into the floating state, allowing another device to drive the data bus.

5.2 Pipelined Bus Cycles

Because all the activities of the Math CoProcessor bus interface occur either during the T_{RS} state or

during the transitions to or from that state, the only difference between a pipelined and a non-pipelined cycle is the manner of changing from one state to another. The exact activities during each state are detailed in the previous section ''Non-pipelined Bus Cycles''.

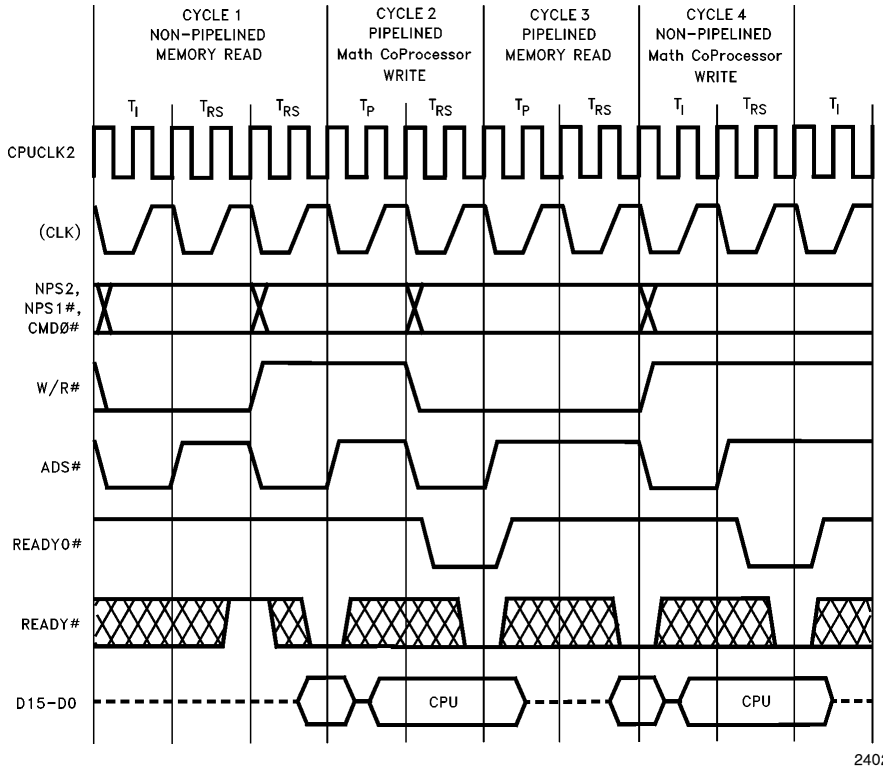
When the CPU asserts ADS# before the end of a bus cycle, both ADS# and READY# are active during a T_{RS} state. This condition causes the Math CoProcessor to change to a different state named T_P. One clock period after a T_P state, the Math CoProcessor always returns to the T_{RS} state. In consecutive pipelined cycles, the Math CoProcessor bus logic uses only the T_{RS} and T_P states.

Figure 5-3 shows the fastest transitions into and out of the pipelined bus cycles. Cycle 1 in the figure represents a non-pipelined cycle. (Non-pipelined write are always followed by another non-pipelined cycle, because READY# is asserted before the earliest possible assertion of ADS# for the next cycle.)

Figure 5-4 shows pipelined write and read cycles with one additional T_{RS} state beyond the minimum required. To delay the assertion of READY# requires external logic.

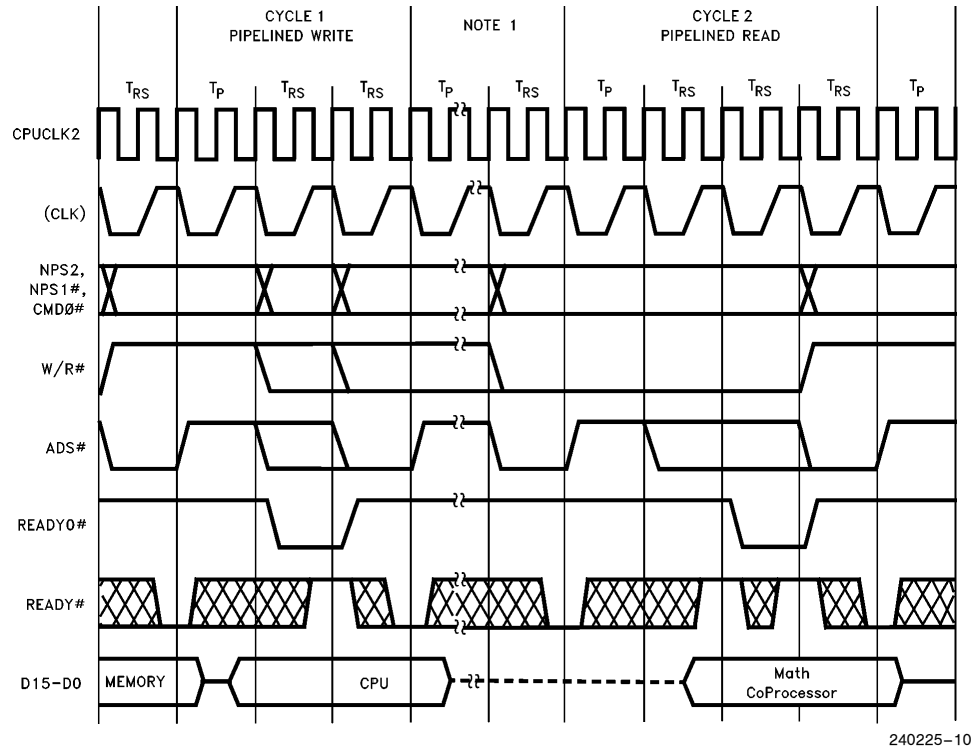
5.3 Mixed Bus Cycles

When the Math CoProcessor bus logic is in the T_{RS} state, it distinguishes between non-pipelined and pipelined cycles according to the behavior of ADS# and READY#. In a non-pipelined cycle, only READY# is activated, and the transition is from the T_{RS} state to the idle state. In a pipelined cycle, both READY# and ADS# are active, and the transition is first from T_{RS} state to T_P state, then, after one clock period, back to T_{RS} state.



Cycle 1–Cycle 4 represent the operand transfer cycle for an instruction involving a transfer of two 32-bit loads in total. The opcode write cycles and other overhead are not shown.
Note that the next cycle will be a pipelined cycle if both READY# and ADS# are sampled active at the end of a T_{RS} state of the current cycle.

Figure 5-3. Fastest Transitions to and from Pipelined Cycles

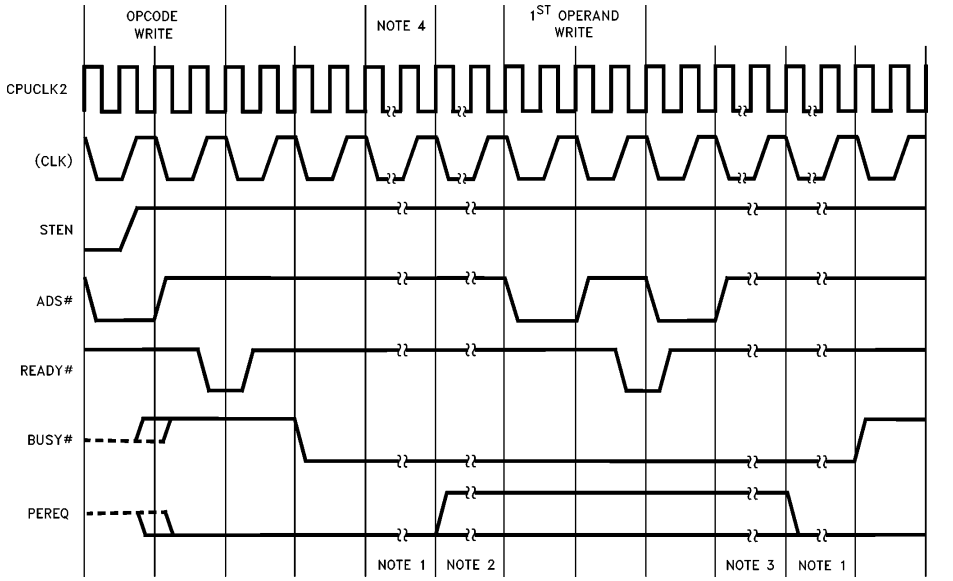


NOTE:
1. Cycles between operand write to the Math CoProcessor and storing result.

Figure 5-4. Pipelined Cycles with Wait States

5.4 BUSY# and PEREQ Timing Relationship

Figure 5-5 shows the activation of BUSY# at the beginning of instruction execution and its deactivation upon completion of the instruction. PEREQ is activated within this interval. If ERROR# is ever asserted, it would be asserted at least six CPUCLK2 periods after the deactivation of PEREQ and would be deasserted at least six CPUCLK2 periods before the deactivation of BUSY#.



NOTES:
1. Instruction dependent.
2. PEREQ is an asynchronous input to the Intel386™ Microprocessor; it may not be asserted (instruction dependent).
3. More operand transfers.
4. Memory read (operand) cycle is not shown.

Figure 5-5. STEN, BUSY#, and PEREQ Timing Relationships

6.0 PACKAGE SPECIFICATIONS

6.1 Mechanical Specifications

The Intel387 SX Math CoProcessor is packaged in a 68-pin PLCC package. Detailed mechanical specifications can be found in the Intel Packaging Specification, Order Number 231369.

6.2 Thermal Specifications

The Intel387 SX Math CoProcessor is specified for operation when the case temperature is within the range of 0°C to 100°C. The case temperature (T_C) may be measured in any environment to determine whether the Intel387 SX Math CoProcessor is within the specified operating range. The case temperature should be measured at the center of the top surface.

The ambient temperature (T_A) is guaranteed as long as T_C is not violated. The ambient temperature can be calculated from the θ_{JC} (thermal resistance constant from the transistor junction to the case) and θ_{JA} (thermal resistance from junction to ambient) from the following calculations:

Junction Temperature $T_J = T_C + P^*\theta_{JC}$

Ambient Temperature $T_A = T_J - P^*\theta_{JA}$

Case Temperature $T_C = T_A + P^* (\theta_{JA} - \theta_{JC})$

Values for θ_{JA} and θ_{JC} are given in Table 6-1 for the 68 pin PLCC package. θ_{JC} is given at various airflows. Table 6-2 shows the maximum T_A allowable without exceeding T_C at various airflows. Note that T_A can be improved further by attaching a heat sink to the package. P is calculated by using the maximum hot I_{CC} and maximum V_{CC}.

Table 6-1. Thermal Resistances (°C/Watt) θ_{JC} and θ_{JA}

Package	θ_{JC}	θ_{JA} versus Airflow - ft/min (m/sec)					
		0 (0)	200 (1.01)	400 (2.03)	600 (3.04)	800 (4.06)	1000 (5.07)
68-Pin PLCC	8	30	25	20	15.5	13	12

Table 6-2. Maximum T_A at Various Airflows

Package	T_A (°C) versus Airflow - ft/min (m/sec)					
	0 (0)	200 (1.01)	400 (2.03)	600 (3.04)	800 (4.06)	1000 (5.07)
68-Pin PLCC	84.9	88.3	91.8	94.8	96.6	97.2

Maximum T_A is calculated at maximum V_{CC} and maximum I_{CC}.

7.0 ELECTRICAL CHARACTERISTICS

The following specifications represent the targets of the design effort. They are subject to change without notice. Contact your Intel representative to get the most up-to-date values.

7.1 Absolute Maximum Ratings*

Case Temperature T_C Under Bias . . . 0°C to +100°C

Storage Temperature . . . . . −65°C to +150°C

Voltage on Any Pin with Respect to Ground . . . . . −0.5 to V_{CC}+0.5

Power Dissipation 0.8W

NOTICE: This is a production data sheet. The specifications are subject to change without notice.

**WARNING: Stressing the device beyond the "Absolute Maximum Ratings" may cause permanent damage. These are stress ratings only. Operation beyond the "Operating Conditions" is not recommended and extended exposure beyond the "Operating Conditions" may affect device reliability.*

7.2 D.C. Characteristics

Table 7-1. D.C. Specifications $T_C = 0°C$ to $+100°C$, $V_{CC} = 5V \pm 10\%$

Symbol	Parameter	Min	Max	Units	Test Conditions
V_{IL}	Input LO Voltage	-0.3	$+0.8$	V	(Note 1)
V_{IH}	Input HI Voltage	2.0	$V_{CC}+0.3$	V	(Note 1)
V_{CL}	CPUCLK2 and NUMCLK2 Input LO Voltage	-0.3	$+0.8$	V	
V_{CH}	CPUCLK2 and NUMCLK2 Input HI Voltage	$V_{CC}-0.8$	$V_{CC}+0.8$	V	
V_{OL}	Output LO Voltage		0.45	V	(Note 2)
V_{OH}	Output HI Voltage	2.4		V	(Note 3)
V_{OH}	Output HI Voltage	$V_{CC}-0.8$		V	(Note 4)
I_{CC}	Power Supply Current Dynamic Mode				
	Freq. = 33 MHz(5)		150	mA	I_{CC} typ. = 135 mA
	Freq. = 25 MHz(5)		150	mA	I_{CC} typ. = 130 mA
	Freq. = 20 MHz(5)		125	mA	I_{CC} typ. = 110 mA
	Freq. = 16 MHz(5)		100	mA	I_{CC} typ. = 90 mA
	Freq. = 1 MHz(5)		20	mA	I_{CC} typ. = 5 mA
	Idle Mode(6)		7	mA	I_{CC} typ. = 4 mA
I_{LI}	Input Leakage Current		± 15	μA	$0V \le V_{IN} \le V_{CC}$
I_{LO}	I/O Leakage Current		± 15	μA	$0.45V \le V_O \le V_{CC}$
C_{IN}	Input Capacitance	7	10	pF	$f_c = 1$ MHz
C_O	I/O Capacitance	7	12	pF	$f_c = 1$ MHz
C_{CLK}	Clock Capacitance	7	20	pF	$f_c = 1$ MHz

NOTES:
1. This parameter is for all inputs, excluding the clock inputs.
2. This parameter is measured at I_{OL} as follows:
   Data = 4.0 mA
   READYO#, ERROR#, BUSY#, PEREQ = 25 mA
3. This parameter is measured at I_{OH} as follows:
   Data = 1.0 mA
   READYO#, ERROR#, BUSY#, PEREQ = 0.6 mA
4. This parameter is measured at I_{OH} as follows:
   Data = 0.2 mA
   READYO#, ERROR#, BUSY# PEREQ = 0.12 mA
5. Synchronous Clock Mode (CKM = 1). I_{CC} is measured at steady state, maximum capacitive loading on the outputs, and worst-case D.C. level at the inputs.
6. Intel387 SX Math CoProcessor Internal Idle Mode. Synchronous clock mode, clock and control inputs are active but the Math CoProcessor is not executing an instruction. Outputs driving CMOS inputs.

7.3 A.C. Characteristics

Table 7-2a. Timing Requirements of the Bus Interface Unit
T_C = 0°C to +100°C, V_{CC} = 5V ±10% (All measurements made at 1.5V unless otherwise specified)

Pin	Symbol	Parameter	16 MHz–25 MHz		33 MHz		Test Conditions	Refer to Figure
			Min (ns)	Max (ns)	Min (ns)	Max (ns)		
CPUCLK2	t1	Period	20	DC	15	DC	2.0V	7.2
CPUCLK2	t2a	High Time	6		6.25		2.0V	
CPUCLK2	t2b	High Time	3		4.5		$V_{CC}-0.8V$	
CPUCLK2	t3a	Low Time	6		6.25		2.0V	
CPUCLK2	t3b	Low Time	4		4.5		0.8V	
CPUCLK2	t4	Fall Time		7		4	From $V_{CC}-0.8V$ to 0.8V	
CPUCLK2	t5	Rise Time		7		4	From 0.8V to $V_{CC}-0.8V$	
READYO#	t7a	Out Delay	4	25	4	17	C_L = 50 pF	7.3
PEREQ	t7b	Out Delay	4	23	4	21	C_L = 50 pF	
BUSY#	t7c	Out Delay	4	23	4	21	C_L = 50 pF	
ERROR#	t7d	Out Delay	4	23	4	23	C_L = 50 pF	
D15–D0	t8	Out Delay	1	45	0	37	C_L = 50 pF	7.4
D15–D0	t10	Setup Time	11		8			
D15–D0	t11	Hold Time	11		8			
D15–D0	t12*	Float Time	6	24	6	19		
READYO#	t13a*	Float Time	1	40	1	30		7.6
PEREQ	t13b*	Float Time	1	40	1	30		
BUSY#	t13c*	Float Time	1	40	1	30		
ERROR#	t13d*	Float Time	1	40	1	30		
ADS#	t14a	Setup Time	15		13			7.4
ADS#	t15a	Hold Time	4		4			
W/R#	t14b	Setup Time	15		13			
W/R#	t15b	Hold Time	4		4			
READY#	t16a	Setup Time	9		7			7.4
READY#	t17a	Hold Time	4		4			
CMD0#	t16b	Setup Time	16		13			
CMD0#	t17b	Hold Time	2		2			
NPS1#, NPS2	t16c	Setup Time	16		13			
NPS1#, NPS2	t17c	Hold Time	2		2			
STEN	t16d	Setup Time	15		13			
STEN	t17d	Hold Time	2		2			
RESETIN	t18	Setup Time	8		5			7.5
RESETIN	t19	Hold Time	3		2			

NOTE:
*Float condition occurs when maximum output current becomes less than I_{LO} in magnitude. Float delay is not tested.

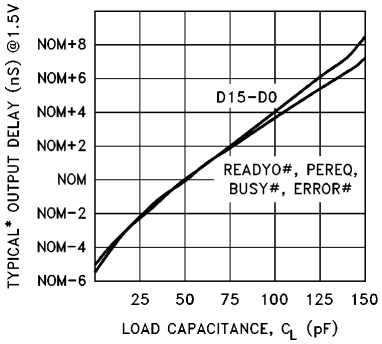
Table 7-2b. Timing Requirements of the Execution Unit (Asynchronous Mode CKM = 0)

Pin	Symbol	Parameter	16 MHz–25 MHz		33 MHz		Test Conditions	Refer to Figure
			Min (ns)	Max (ns)	Min (ns)	Max (ns)		
NUMCLK2	t1	Period	20	500	15	500	2.0V	7.2
NUMCLK2	t2a	High Time	6		6.25		2.0V	
NUMCLK2	t2b	High Time	3		4.5		$V_{CC}-0.8V$	
NUMCLK2	t3a	Low Time	6		6.25		2.0V	
NUMCLK2	t3b	Low Time	4		4.5		0.8V	
NUMCLK2	t4	Fall Time		7		6	From $V_{CC}-0.8V$ to 0.8V	
NUMCLK2	t5	Rise Time		7		6	From 0.8V to $V_{CC}-0.8V$	
NUMCLK2/ CPUCLK2		Ratio	10/16	14/10	10/16	14/10		

NOTE:
If not used (CKM = 1) tie NUMCLK2 low.

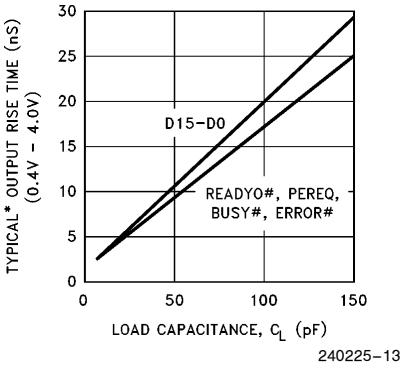
Table 7-2c. Other A.C. Parameters

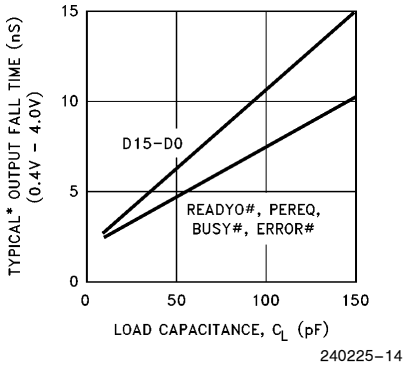
Pin	Symbol	Parameter	Min	Max	Units
RESETIN	t30	Duration	40		NUMCLK2
RESETIN	t31	RESETIN Inactive to 1st Opcode Write	50		NUMCLK2
BUSY#	t32	Duration	6		CPUCLK2
BUSY#, ERROR#	t33	ERROR# (In)Active to BUSY# Inactive	6		CPUCLK2
PEREQ, ERROR#	t34	PEREQ Inactive to ERROR# Active	6		CPUCLK2
READY#, BUSY#	t35	READY# Active to BUSY# Active	0	4	CPUCLK2
READY#	t36	Minimum Time from Opcode Write to Opcode/Operand Write	4		CPUCLK2
READY#	t37	Minimum Time from Operand Write to Operand Write	4		CPUCLK2



NOTE:
*Typical part under worst-case conditions.

Figure 7-1a. Typical Output Valid Delay vs Load Capacitance at Max Operating Temperature





NOTE:
*Typical part under worst-case conditions.

Figure 7-1b. Typical Output Slew Time vs Load Capacitance at Max Operating Temperature

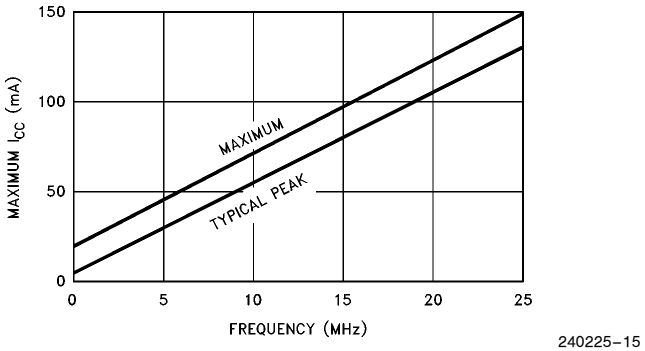


Figure 7-1c. Maximum I_{CC} vs Frequency

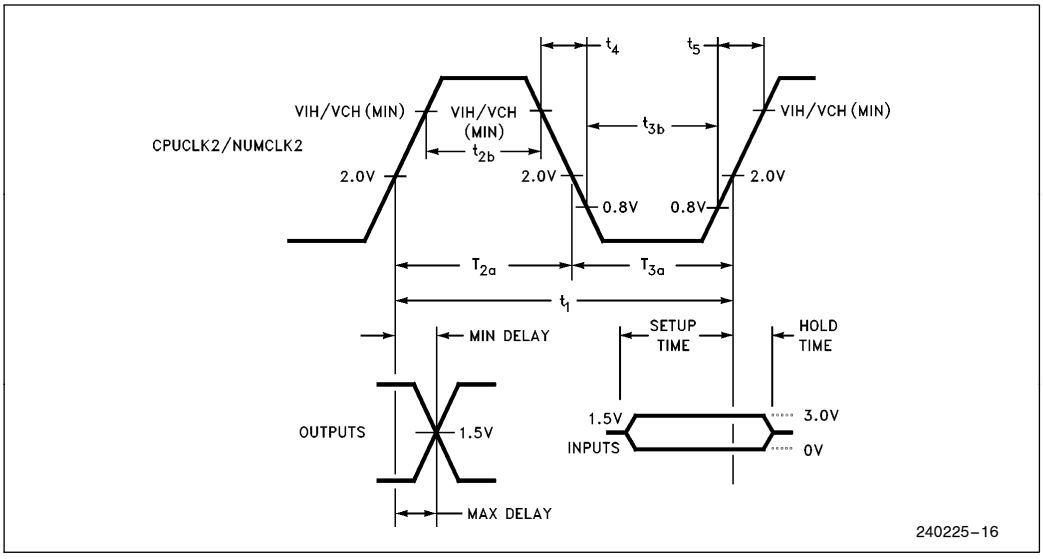


Figure 7-2. CPUCLK2/NUMCLK2 Waveform and Measurement Points for Input/Output

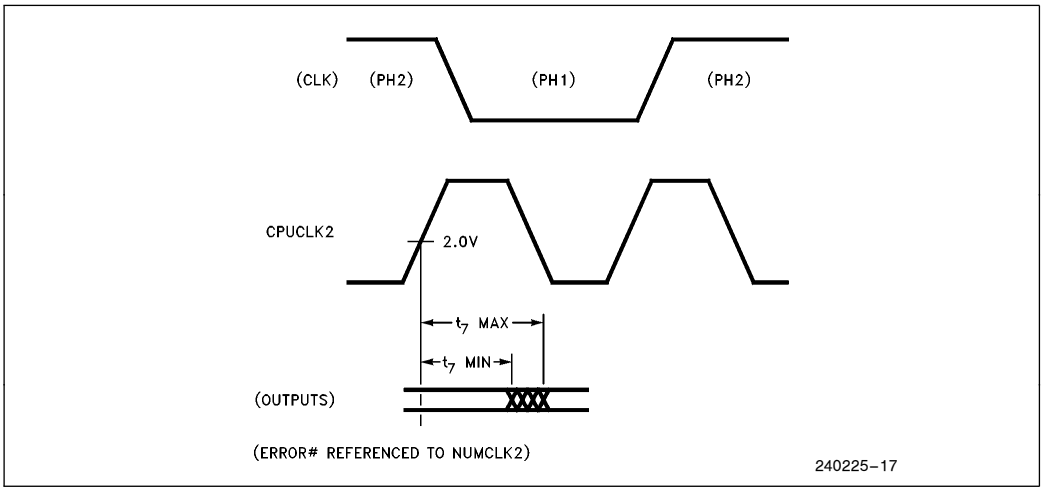


Figure 7-3. Output Signals

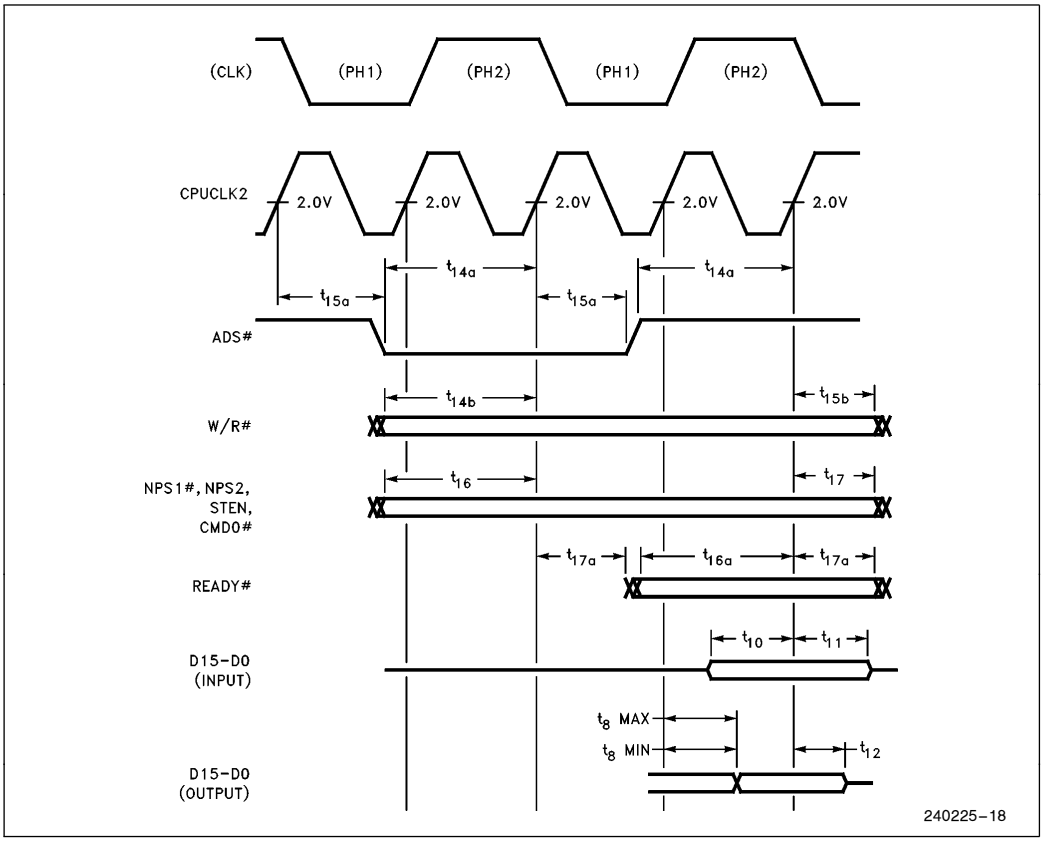
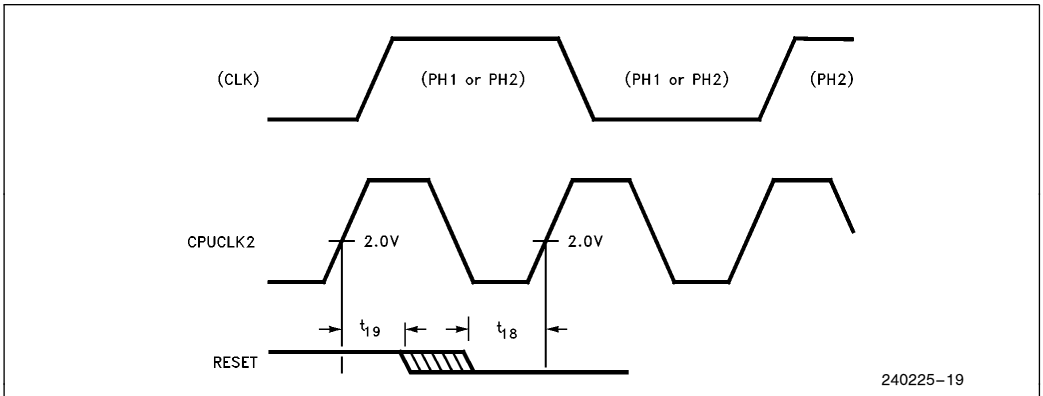


Figure 7-4. Input and I/O Signals



NOTE:
The second internal processor phase following RESET high to low transition is PH2.

Figure 7-5. RESET Signal

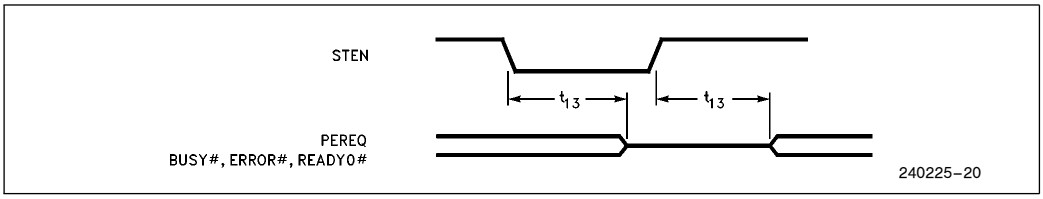
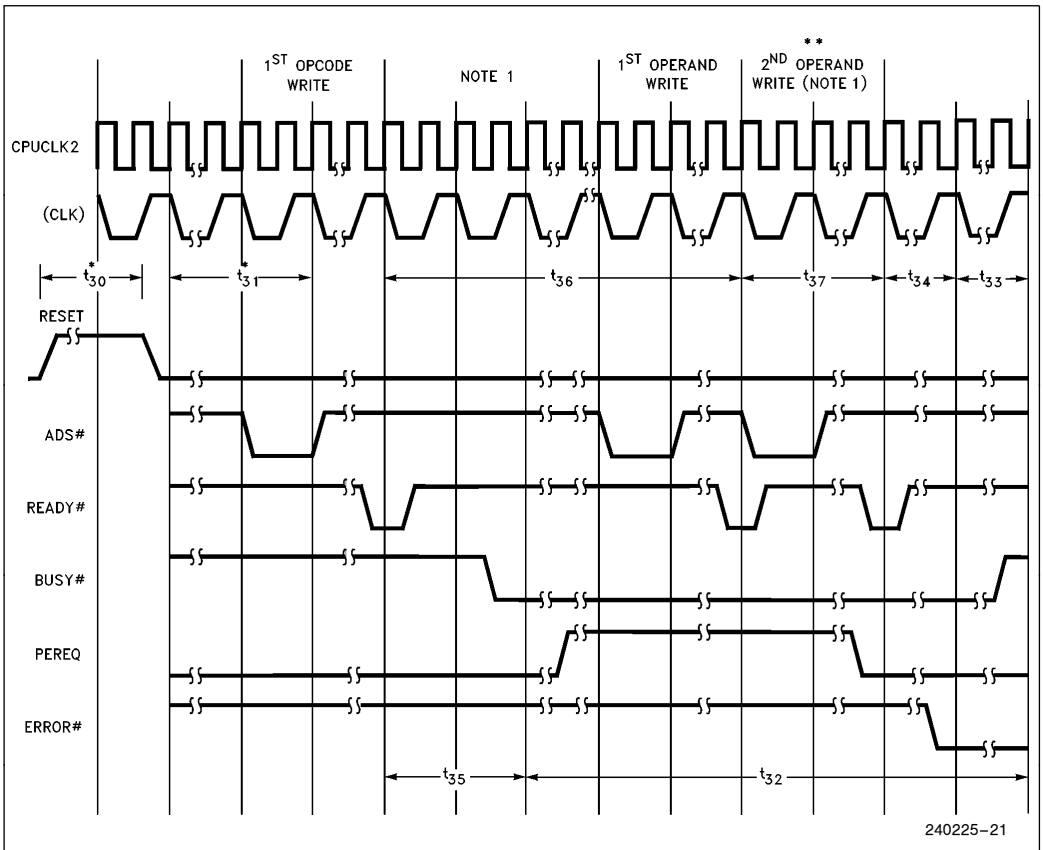


Figure 7-6. Float from STEN



*In NUMCLK2's
**or last operand

NOTE:
1. Memory read (operand) cycle is not shown.

Figure 7-7. Other Parameters

8.0 INTEL387 SX MATH COPROCESSOR INSTRUCTION SET

Instructions for the Intel387 SX Math CoProcessor assume one of the five forms shown in Table 8-1. In all cases, instructions are at least two bytes long and begin with the bit pattern 11011B, which identifies the ESCAPE class of instruction. Instructions that refer to memory operands specify addresses using the CPU's addressing modes.

MOD (Mode field) and R/M (Register/Memory specifier) have the same interpretation as the corresponding fields of CPU instructions (refer to Programmer's Reference Manual for the CPU). SIB (Scale Index Base) byte and DISP (displacement) are optionally present in instructions that have MOD and R/M fields. Their presence depends on the values of MOD and R/M, as for instructions of the CPU.

The instruction summaries that follow in Table 8-2 assume that the instruction has been prefetched, decoded, and is ready for execution; that bus cycles do not require wait states; that there are no local bus HOLD requests delaying processor access to the bus; and that no exceptions are detected during instruction execution. If the instruction has MOD and R/M fields that call for both base and index registers, add one clock.

Table 8-1. Instruction Formats

	Instruction								Optional Fields	
	First Byte				Second Byte					
1	11011	OPA		1	MOD		1	OPB	R/M	SIB	DISP
2	11011	MF		OPA	MOD		OPB*		R/M	SIB	DISP
3	11011	d	P	OPA	1	1	OPB*		ST(i)		
4	11011	0	0	1	1	1	1	OP			
5	11011	0	1	1	1	1	1	OP			
	15–11	10	9	8	7	6	5	4 3 2 1 0			

OP = Instruction opcode, possibly split into two fields OPA and OPB
MF = Memory Format
00 - 32-bit real
01 - 32-bit integer
10 - 64-bit real
11 - 16-bit integer
d = Destination
0 - Destination is ST(0)
1 - Destination is ST(i)
R XOR d = 0 - Destination (op) Source
R XOR d = 1 - Source (op) Destination
*In FSUB and FDIV, the low-order bit of OPB is the R (reversed) bit
P = POP
0 - Do not pop stack
1 - Pop stack after operation
ESC = 11011
ST(i) = Register stack element i
000 = Stack top
001 = Second stack element
•
•
•
111 = Eighth stack element

Instruction	Encoding			Clock Count Range			
	Byte 0	Byte 1	Optional Bytes 2–6	32-Bit Real	32-Bit Integer	64-Bit Real	16-Bit Integer
DATA TRANSFER							
FLD = Load[a]							
Integer/real memory to ST(0)	ESC MF 1	MOD 000 R/M	SIB/DISP	11–20	28–44	20–27	42–53
Long integer memory to ST(0)	ESC 111	MOD 101 R/M	SIB/DISP		30–58		
Extended real memory to ST(0)	ESC 011	MOD 101 R/M	SIB/DISP		16–47		
BCD memory to ST(0)	ESC 111	MOD 100 R/M	SIB/DISP		49–101		
ST(i) to ST(0)	ESC 001	11000 ST(i)			7–12		
FST = Store							
ST(0) to integer/real memory	ESC MF 1	MOD 010 R/M	SIB/DISP	27–45	59–78	59	58–76
ST(0) to ST(i)	ESC 101	11010 ST(i)			7–11		
FSTP = Store and Pop							
ST(0) to integer/real memory	ESC MF 1	MOD 011 R/M	SIB/DISP	27–45	59–78	59	58–76
ST(0) to long integer memory	ESC 111	MOD 111 R/M	SIB/DISP		64–86		
ST(0) to extended real memory	ESC 011	MOD 111 R/M	SIB/DISP		50–56		
ST(0) to BCD memory	ESC 111	MOD 110 R/M	SIB/DISP		116–194		
ST(0) to ST(i)	ESC 101	11011 ST (i)			7–11		
FXCH = Exchange							
ST(i) and ST(0)	ESC 001	11001 ST(i)			10–17		
COMPARISON							
FCOM = Compare							
Integer/real memory to ST(0)	ESC MF 0	MOD 010 R/M	SIB/DISP	15–27	36–54	18–31	39–62
ST(i) to ST(0)	ESC 000	11010 ST(i)			13–21		
FCOMP = Compare and pop							
Integer/real memory to ST(0)	ESC MF 0	MOD 011 R/M	SIB/DISP	15–27	36–54	18–31	39–62
ST(i) to ST(0)	ESC 000	11011 ST(i)			13–21		
FCOMPP = Compare and pop twice							
ST(1) to ST(0)	ESC 110	1101 1001			13–21		
FTST = Test ST(0)	ESC 001	1110 0100			17–25		
FUCOM = Unordered compare	ESC 101	11100 ST(i)			13–21		
FUCOMP = Unordered compare and pop	ESC 101	11101 ST(i)			13–21		
FUCOMPP = Unordered compare and pop twice	ESC 010	1110 1001			13–21		
FXAM = Examine ST(0)	ESC 001	1110 0101			24-37		

Shaded areas indicate instructions not available in 8087/80287.

NOTE:
a. When loading single or double precision zero from memory, add 5 clocks.

Instruction	Encoding			Clock Count Range			
	Byte 0	Byte 1	Optional Bytes 2–6	32-Bit Real	32-Bit Integer	64-Bit Real	16-Bit Integer
ARITHMETIC							
FADD = Add							
Integer/real memory to ST(0)	ESC MF 0	MOD 000 R/M	SIB/DISP	14–31	36–58	19–38	38–64
ST(i) and ST(0)	ESC d P 0	11000 ST(i)	SIB/DISP	12–26[b]			
FSUB = Subtract							
Integer/real memory with ST(0)	ESC MF 0	MOD 10 R R/M	SIB/DISP	14–31	36–58	19–38	38–64[c]
ST(i) to ST(0)	ESC d P 0	1110 R R/M		12–26[d]			
FMUL = Multiply							
Integer/real memory with ST(0)	ESC MF 0	MOD 001 R/M	SIB/DISP	21–33	45–73	27–57	46–74
ST(i) and ST(0)	ESC d P 0	1100 1 R/M		17–50[e]			
FDIV = Divide							
Integer/real memory with ST(0)	ESC MF 0	MOD 11 R R/M	SIB/DISP	79–87	103–116[f]	85–95	105–124[g]
ST(i) and ST(0)	ESC d P 0	1111 R R/M		77–80[h]			
FSQRT[i] = Square root	ESC 001	1111 1010		97–111			
FSCALE = Scale ST(0) by ST(1)	ESC 001	1111 1101		44–82			
FPREM = Partial remainder	ESC 001	1111 1000		56–140			
FPREM1 = Partial remainder (IEEE)	ESC 001	1111 0101		81–168			
FRNDINT = Round ST(0) to integer	ESC 001	1111 1100		41–62			
FXTRACT = Extract components of ST(0)	ESC 001	1111 0100		42–63			
FABS = Absolute value of ST(0)	ESC 001	1110 0001		14–21			
FCHS = Change sign of ST(0)	ESC 001	1110 0000		17–24			
TRANSCENDENTAL							
FCOS[k] = Cosine of ST(0)	ESC 001	1111 1111		122–680			
FPTAN[k] = Partial tangent of ST(0)	ESC 001	1111 0010		162–430[i]			
FPATAN = Partial arctangent of ST(0)	ESC 001	1111 0011		250–420			
FSIN[k] = Sine of ST(0)	ESC 001	1111 1110		121–680			
FSINCOS[k] = Sine and cosine of ST(0)	ESC 001	1111 1011		150–650			
F2XM1[l] = $2^{ST(0)} - 1$	ESC 001	1111 0000		167–410			
FYL2X[m] = ST(1) * $\log_2$ST(0)	ESC 001	1111 0001		99–436			
FYL2XP1[n] = ST(1) * $\log_2$[ST(0) + 1.0]	ESC 001	1111 1001		210–447			

Shaded areas indicate instructions not available in 8087/80287.

NOTES:
b. Add 3 clocks to the range when d = 1.
c. Add 1 clock to each range when R = 1.
d. Add 3 clocks to the range when d = 0.
e. typical = 52 (When d = 0, 46–54, typical = 49).
f. Add 1 clock to the range when R = 1.
g. 135–141 when R = 1.
h. Add 3 clocks to the range when d = 1.
i. $-0 \leq ST(0) \leq +\infty$.
j. These timings hold for operands in the range $|x| < \pi$. For operands not in this range, up to 76 additional clocks may be needed to reduce the operand.
k. $0 \leq ST(0) < 2^{63}$.
l. $-1.0 \leq ST(0) \leq 1.0$.
m. $0 \leq ST(0) < \infty$, $-\infty < ST(1) < +\infty$.
n. $0 \leq |ST(0)| < [2\text{-SQRT}(2)]/2$, $-\infty < ST(1) < +\infty$.

Instruction	Encoding			Clock Count Range			
	Byte 0	Byte 1	Optional Bytes 2–6	32-Bit Real	32-Bit Integer	64-Bit Real	16-Bit Integer
CONSTANTS							
FLDZ = Load +0.0 to ST(0)	ESC 001	1110 1110			10–17		
FLD1 = Load +1.0 to ST(0)	ESC 001	1110 1000			15–22		
FLDPI = Load π to ST(0)	ESC 001	1110 1011			26–36		
FLDL2T = Load $\log_2(10)$ to ST(0)	ESC 001	1110 1001			26–36		
FLDL2E = Load $\log_2(e)$ to ST(0)	ESC 001	1110 1010			26–36		
FLDLG2 = Load $\log_{10}(2)$ to ST(0)	ESC 001	1110 1100			25–35		
FLDLN2 = Load $\log_e(2)$ to ST(0)	ESC 001	1110 1101			26–38		
PROCESSOR CONTROL							
FINIT = Initialize Math CoProcessor	ESC 011	1110 0011			33		
FLDCW = Load control word from memory	ESC 001	MOD 101 R/M	SIB/DISP		19		
FSTCW = Store control word to memory	ESC 001	MOD 111 R/M	SIB/DISP		15		
FSTSW = Store status word to memory	ESC 101	MOD 111 R/M	SIB/DISP		15		
FSTSW AX = Store status word to AX	ESC 111	1110 0000			13		
FCLEX = Clear exceptions	ESC 011	1110 0010			11		
FSTENV = Store environment	ESC 001	MOD 110 R/M	SIB/DISP		117–118		
FLDENV = Load environment	ESC 001	MOD 100 R/M	SIB/DISP		85		
FSAVE = Save state	ESC 101	MOD 110 R/M	SIB/DISP		402–403		
FRSTOR = Restore state	ESC 101	MOD 100 R/M	SIB/DISP		415		
FINCSTP = Increment stack pointer	ESC 001	1111 0111			21		
FDECSTP = Decrement stack pointer	ESC 001	1111 0110			22		
FFREE = Free ST(i)	ESC 101	1100 0 ST(i)			18		
FNOP = No operations	ESC 001	1101 0000			12		

APPENDIX A
INTEL387 SX MATH COPROCESSOR COMPATIBILITY

A.1 8087/80287 Compatibility

This section summarizes the differences between the Intel387 SX Math CoProcessor and the 80287 Math CoProcessor. Any migration from the 8087 directly to the Intel387 SX Math CoProcessor must also take into account the differences between the 8087 and the 80287 Math CoProcessor as listed in Appendix B.

Many changes have been designed into the Intel387 SX Math CoProcessor to directly support the IEEE standard in hardware. These changes result in increased performance by eliminating the need for software that supports the standard.

A.1.1 GENERAL DIFFERENCES

The Intel387 SX Math CoProcessor supports only affine closure for infinity arithmetic, not projective closure.

Operands for FSCALE and FPATAN are no longer restricted in range (except for $\pm\infty$); F2XM1 and FPTAN accept a wider range of operands.

Rounding control is in effect for FLD constant.

Software cannot change entries of the tag word to values (other than empty) that differ from actual register contents.

After reset, FINIT, and incomplete FPREM, the Intel387 SX Math CoProcessor resets to zero the condition code bits C_3–C_0 of the status word.

In conformance with the IEEE standard, the Intel387 SX Math CoProcessor does not support the special data formats pseudo-zero, pseudo-NaN, pseudo-infinity, and unnormal.

The denormal exception has a different purpose on the Intel387 SX Math CoProcessor. A system that uses the denormal exception handler solely to normalize the denormal operands, would better mask the denormal exception on the Intel387 SX Math CoProcessor. The Intel387 SX Math CoProcessor automatically normalizes denormal operands when the denormal exception is masked.

A.1.2 EXCEPTIONS

A number of differences exist due to changes in the IEEE standard and to functional improvements to the architecture of the Intel387 SX Math CoProcessor:

1. When the overflow or underflow exception is masked, the Intel387 SX Math CoProcessor differs from the 80287 in rounding when overflow or underflow occurs. The Intel387 SX Math CoProcessor produces results that are consistent with the rounding mode.
2. When the underflow exception is masked, the Intel387 SX Math CoProcessor sets its underflow flag only if there is also a loss of accuracy during denormalization.
3. Fewer invalid-operations exceptions due to denormal operand, because the instructions FSQRT, FDIV, FPREM, and conversions to BCD or to integer normalize denormal operands before proceeding.
4. The FSQRT, FBSTP, and FPREM instructions may cause underflow, because they support denormal operands.
5. The denormal exception can occur during the transcendental instruction and the FXTRACT instruction.
6. The denormal exception no longer takes precedence over all other exceptions.
7. When the denormal exception is masked, the Intel387 SX Math CoProcessor automatically normalizes denormal operands. The 8087/80287 performs unnormal arithmetic, which might produce an unnormal result.
8. When the operand is zero, the FXTRACT instruction reports a zero-divide exception and leaves $-\infty$ in ST(1).
9. The status word has a new bit (SF) that signals when invalid-operation exceptions are due to stack underflow or overflow.
10. FLD *extended precision* no longer reports denormal exceptions, because the instruction is not numeric.
11. FLD *single/double precision* when the operand is denormal converts the number to extended precision and signals the denormal operand exception. When loading a signaling NaN, FLD *single/double precision* signals an invalid-operation exception.
12. The Intel387 SX Math CoProcessor only generates quiet NaNs (as on the 80287); however, the Intel387 SX Math CoProcessor distinguishes between quiet NaNs and signaling NaNs. Signaling NaNs trigger exceptions when they are used as operands; quiet NaNs do not (except for FCOM, FIST, and FBSTP which also raise IE for quiet NaNs).
13. When stack overflow occurs during FPTAN and overflow is masked, both ST(0) and ST(1) contain quiet NaNs. The 80287/8087 leaves the original operand in ST(1) intact.
14. When the scaling factor is $\pm\infty$, the FSCALE instruction behaves as follows:
    - FSCALE (0, ∞) generates the invalid operation exception.
    - FSCALE (finite, $-\infty$) generates zero with the same sign as the scaled operand.
    - FSCALE (finite, $+\infty$) generates ∞ with the same sign as the scaled operand.

    The 8087/80287 returns zero in the first case and raises the invalid-operation exception in the other cases.
15. The Intel387 SX Math CoProcessor returns signed infinity/zero as the unmasked response to massive overflow/underflow. The 8087 and 80287 support a limited range for the scaling factor; within this range either massive overflow/underflow do not occur or undefined results are produced.

APPENDIX B
COMPATIBILITY BETWEEN THE 80287 AND 8087 MATH COPROCESSOR

The 80286/80287 operating in Real Address mode will execute 8086/8087 programs without major modification. However, because of differences in the handling of numeric exceptions by the 80287 Math CoProcessor and the 8087 Math CoProcessor, exception handling routines *may* need to be changed. This appendix summarizes the differences between the 80287 Math CoProcessor and the 8087 Math CoProcessor, and provides details showing how 8087/8087 programs can be ported to the 80286/80287.

1. The Math CoProcessor signals exceptions through a dedicated ERROR# line to the 80286. The Math CoProcessor error signal does not pass through an interrupt controller (the 8087 INT signal does). Therefore, any interrupt controller oriented instructions in numeric exception handlers for the 8086/8087 should be deleted.
2. The 8087 instructions FENI and FDISI perform no useful function in the 80287. If the 80287 encounters one of these opcodes in its instruction stream, the instruction will effectively be ignored; none of the 80287 internal states will be updated. While 8086/8087 programs containing the instruction may be executed on the 80286/80287, it is unlikely that the exception handling routines containing these instructions will be completely portable to the 80287.
3. Interrupt vector 16 must point to the numeric exception handling routine.
4. The ESC instruction address saved in the 80287 includes any leading prefixes before the ESC opcode. The corresponding address saved in the 8087 does not include leading prefixes.
5. In Protected Address mode, the format of the 80287's saved instruction and address pointers is different than for the 8087. The instruction opcode is not saved in Protected mode; exception handlers will have to retrieve the opcode from memory if needed.
6. Interrupt 7 will occur in the 80286 when executing ESC instructions with either TS (task switched) or EM (emulation) of the 80286 MSW set (TS = 1 or EM = 1). It TS is set, then a WAIT instruction will also cause interrupt 7. An exception handler should be included in 80286/80287 code to handle these situations.
7. Interrupt 9 will occur if the second or subsequent words of a floating point operand fall outside a segment's size. Interrupt 13 will occur if the starting address of a numeric operand falls outside a segment's size. An exception handler should be included in 80286/80287 code to report these programming errors.
8. Except for the processor control instructions, all of the 80287 numeric instructions are automatically synchronized by the 80286 CPU; the 80286 CPU automatically tests the BUSY# line from the 80287 to ensure that the 80287 has completed its previous instruction before executing the next ESC instruction. No explicit WAIT instructions are required to assure this synchronization. For the 8087 used witth 8086 and 8088 processors, explicit WAITs are required before each numeric instruction to ensure synchronization. Although 8086/8087 programs having explicit WAIT instructions will execute perfectly on the 80286/80287 without reassembly, these WAIT instructions are unnecessary.
9. Since the 80287 does not require WAIT instructions before each numeric instruction, the ASM286 assembler does not automatically generate these WAIT instuctions. The ASM86 assembler, however, automatically precedes every ESC instruction with a WAIT instruction. Although numeric routines generated using the ASM86 assembler will generally execute correctly on the 80286/80287, reassembly using ASM286 may result in a more compact code image.

The processor control instructions for the 80287 may be coded using either a WAIT or No-WAIT form of mnemonic. The WAIT forms of these instructions cause ASM286 to precede the ESC instructions with a CPU WAIT instruction, in the identical manner as does ASM86.

80C187
80-BIT MATH COPROCESSOR

- **High Performance 80-Bit Internal Architecture**
- **Implements ANSI/IEEE Standard 754-1985 for Binary Floating-Point Arithmetic**
- **Upward Object-Code Compatible from 8087**
- **Fully Compatible with 387DX and 387SX Math Coprocessors. Implements all 387 Architectural Enhancements over 8087**
- **Directly Interfaces with 80C186 CPU**
- **80C186/80C187 Provide a Software/ Binary Compatible Upgrade from 80186/82188/8087 Systems**
- **Expands 80C186's Data Types to Include 32-, 64-, 80-Bit Floating-Point, 32-, 64-Bit Integers and 18-Digit BCD Operands**
- **Directly Extends 80C186's Instruction Set to Trigonometric, Logarithmic, Exponential, and Arithmetic Instructions for All Data Types**
- **Full-Range Transcendental Operations for SINE, COSINE, TANGENT, ARCTANGENT, and LOGARITHM**
- **Built-In Exception Handling**
- **Eight 80-Bit Numeric Registers, Usable as Individually Addressable General Registers or as a Register Stack**
- **Available in 40-Pin CERDIP and 44-Pin PLCC Package**

  (See Packaging Outlines and Dimensions, Order #231369)

The Intel 80C187 is a high-performance math coprocessor that extends the architecture of the 80C186 with floating-point, extended integer, and BCD data types. A computing system that includes the 80C187 fully conforms to the IEEE Floating-Point Standard. The 80C187 adds over seventy mnemonics to the instruction set of the 80C186, including support for arithmetic, logarithmic, exponential, and trigonometric mathematical operations. The 80C187 is implemented with 1.5 micron, high-speed CHMOS III technology and packaged in both a 40-pin CERDIP and a 44-pin PLCC package. The 80C187 is upward object-code compatible from the 8087 math coprocessor and will execute code written for the 80387DX and 80387SX math coprocessors.

November 1992 **Order Number: 270640-004**

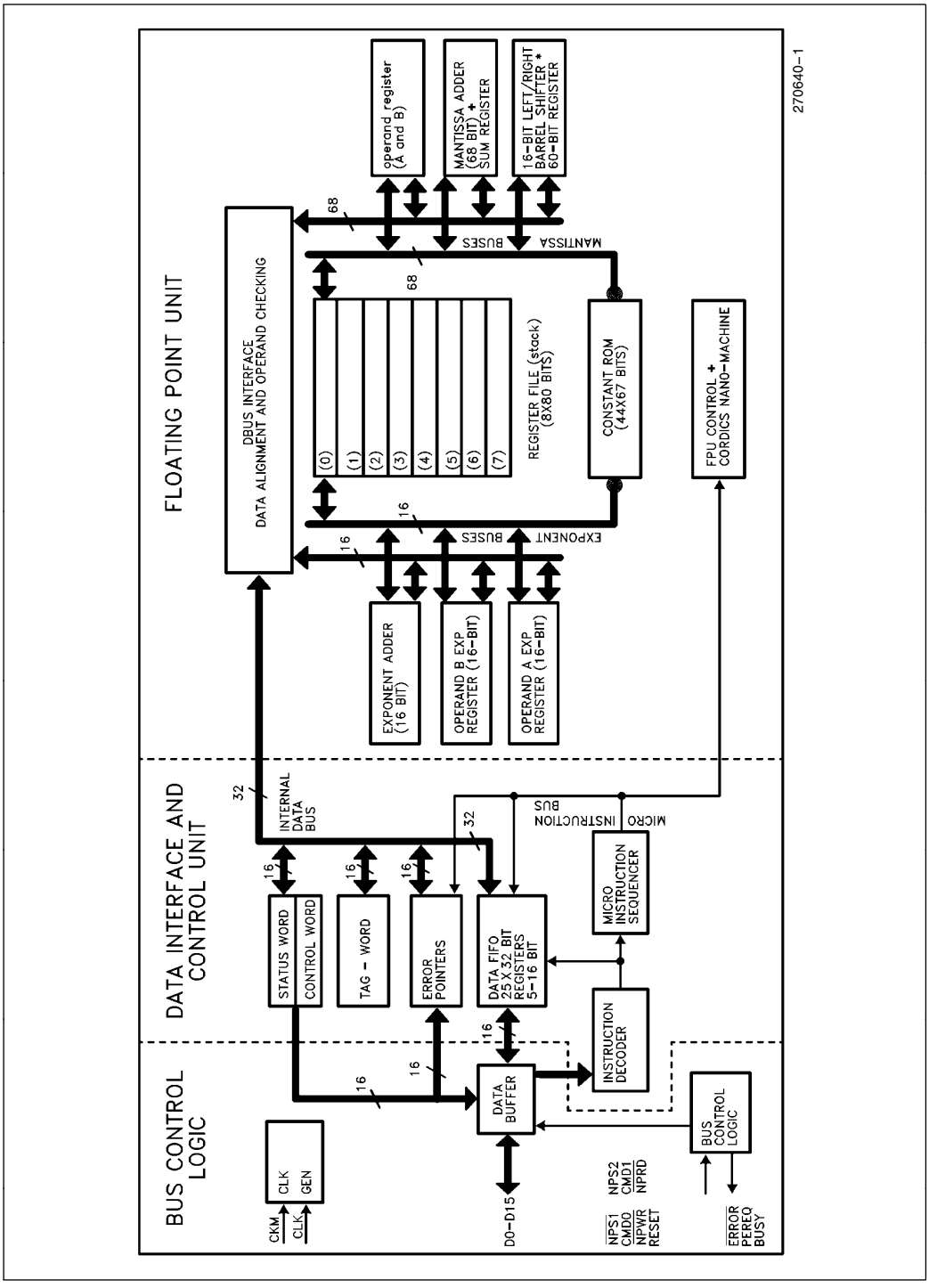


Figure 1. 80C187 Block Diagram

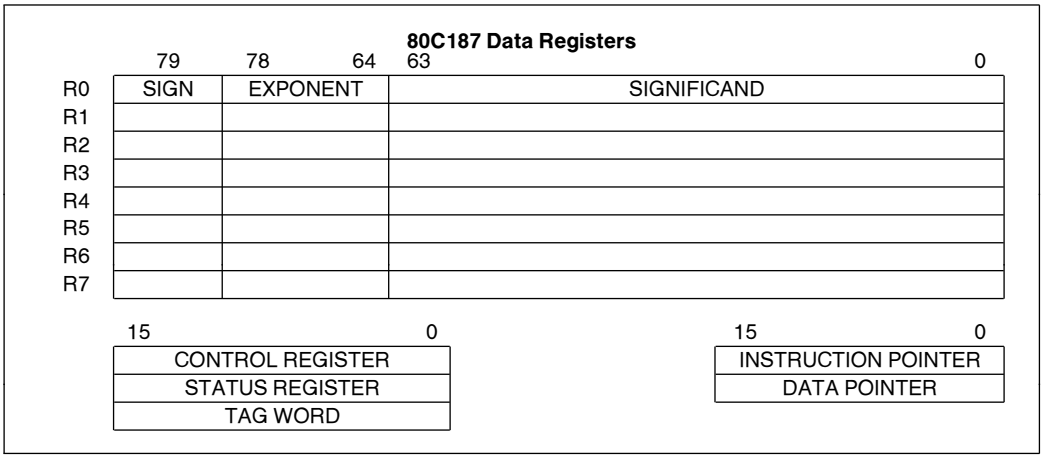


Figure 2. Register Set

FUNCTIONAL DESCRIPTION

The 80C187 Math Coprocessor provides arithmetic instructions for a variety of numeric data types. It also executes numerous built-in transcendental functions (e.g. tangent, sine, cosine, and log functions). The 80C187 effectively extends the register and instruction set of the 80C186 CPU for existing data types and adds several new data types as well. Figure 2 shows the additional registers visible to programs in a system that includes the 80C187. Essentially, the 80C187 can be treated as an additional resource or an extension to the CPU. The 80C186 CPU together with an 80C187 can be used as a single unified system.

A 80C186 system that includes the 80C187 is completely upward compatible with software for the 8086/8087.

The 80C187 interfaces only with the 80C186 CPU. The interface hardware for the 80C187 is not implemented on the 80C188.

PROGRAMMING INTERFACE

The 80C187 adds to the CPU additional data types, registers, instructions, and interrupts specifically designed to facilitate high-speed numerics processing. All new instructions and data types are directly supported by the assembler and compilers for high-level languages. The 80C187 also supports the full 80387DX instruction set.

All communication between the CPU and the 80C187 is transparent to applications software. The CPU automatically controls the 80C187 whenever a numerics instruction is executed. All physical memory and virtual memory of the CPU are available for storage of the instructions and operands of programs that use the 80C187. All memory addressing modes are available for addressing numerics operands.

The end of this data sheet lists by class the instructions that the 80C187 adds to the instruction set.

NOTE:
The 80C187 Math Coprocessor is also referred to as a Numeric Processor Extension (NPX) in this document.

Data Types

Table 1 lists the seven data types that the 80C187 supports and presents the format for each type. Operands are stored in memory with the least significant digit at the lowest memory address. Programs retrieve these values by generating the lowest address. For maximum system performance, all operands should start at even physical-memory addresses; operands may begin at odd addresses, but will require extra memory cycles to access the entire operand.

Internally, the 80C187 holds all numbers in the extended-precision real format. Instructions that load operands from memory automatically convert operands represented in memory as 16-, 32-, or 64-bit integers, 32- or 64-bit floating-point numbers, or 18-digit packed BCD numbers into extended-precision real format. Instructions that store operands in memory perform the inverse type conversion.

Numeric Operands

A typical NPX instruction accepts one or two operands and produces one (or sometimes two) results. In two-operand instructions, one operand is the contents of an NPX register, while the other may be a memory location. The operands of some instructions are predefined; for example, FSQRT always takes the square root of the number in the top stack element (refer to the section on Data Registers).

Register Set

Figure 2 shows the 80C187 register set. When an 80C187 is present in a system, programmers may use these registers in addition to the registers normally available on the CPU.

DATA REGISTERS

80C187 computations use the extended-precision real data type.

Table 1. Data Type Representation in Memory

Data Formats	Range	Precision	Most Significant Byte ... HIGHEST ADDRESSED BYTE
Word Integer	$\pm 10^4$	16 Bits	(TWO'S COMPLEMENT) 15 0
Short Integer	$\pm 10^9$	32 Bits	(TWO'S COMPLEMENT) 31 0
Long Integer	$\pm 10^{18}$	64 Bits	(TWO'S COMPLEMENT) 63 0
Packed BCD	$\pm 10^{18}$	18 Digits	S X MAGNITUDE d_{17} ... d_0 79 72 0
Single Precision	$\pm 10^{\pm 38}$	24 Bits	S BIASED EXPONENT SIGNIFICAND 31 23 ▲ 0
Double Precision	$\pm 10^{\pm 308}$	53 Bits	S BIASED EXPONENT SIGNIFICAND 63 52 ▲ 0
Extended Precision	$\pm 10^{\pm 4932}$	64 Bits	S BIASED EXPONENT I SIGNIFICAND 79 64 63 ▲ 0

270640–2

NOTES:
1. S = Sign bit (0 = Positive, 1 = Negative)
2. d_n = Decimal digit (two per byte)
3. X = Bits have no significance; 80C187 ignores when loading, zeros when storing
4. ▲ = Position of implicit binary point
5. I = Integer bit of significand; stored in temporary real, implicit in single and double precision
6. Exponent Bias (normalized values):
   Single: 127 (7FH)
   Double: 1023 (3FFH)
   Extended Real: 16383 (3FFFH)
7. Packed BCD: $(-1)^S$ $(D_{17} \ldots D_0)$
8. Real: $(-1)^S$ $(2^{E\text{-}BIAS})$ $(F_0, F_1 \ldots)$

The 80C187 register set can be accessed either as a stack, with instructions operating on the top one or two stack elements, or as individually addressable registers. The TOP field in the status word identifies the current top-of-stack register. A ''push'' operation decrements TOP by one and loads a value into the new top register. A ''pop'' operation stores the value from the current top register and then increments TOP by one. The 80C187 register stack grows ''down'' toward lower-addressed registers.

Instructions may address the data registers either implicitly or explicitly. Many instructions operate on the register at the TOP of the stack. These instructions implicitly address the register at which TOP points. Other instructions allow the programmer to explicitly specify which register to use. This explicit addressing is also relative to TOP.

TAG WORD

The tag word marks the content of each numeric data register, as Figure 3 shows. Each two-bit tag represents one of the eight data registers. The principal function of the tag word is to optimize the NPX's performance and stack handling by making it possible to distinguish between empty and nonempty register locations. It also enables exception handlers to identify special values (e.g. NaNs or denormals) in the contents of a stack location without the need to perform complex decoding of the actual data.

STATUS WORD

The 16-bit status word (in the status register) shown in Figure 4 reflects the overall state of the 80C187. It may be read and inspected by programs.

Bit 15, the B-bit (busy bit) is included for 8087 compatibility only. It always has the same value as the ES bit (bit 7 of the status word); it does **not** indicate the status of the BUSY output of 80C187.

Bits 13–11 (TOP) point to the 80C187 register that is the current top-of-stack.

The four numeric condition code bits (C_3–C_0) are similar to the flags in a CPU; instructions that perform arithmetic operations update these bits to reflect the outcome. The effects of these instructions on the condition code are summarized in Tables 2 through 5.

Bit 7 is the error summary (ES) status bit. This bit is set if any unmasked exception bit is set; it is clear otherwise. If this bit is set, the $\overline{\text{ERROR}}$ signal is asserted.

Bit 6 is the stack flag (SF). This bit is used to distinguish invalid operations due to stack overflow or underflow from other kinds of invalid operations. When SF is set, bit 9 (C_1) distinguishes between stack overflow ($C_1 = 1$) and underflow ($C_1 = 0$).

Figure 4 shows the six exception flags in bits 5–0 of the status word. Bits 5–0 are set to indicate that the 80C187 has detected an exception while executing an instruction. A later section entitled ''Exception Handling'' explains how they are set and used.

Note that when a new value is loaded into the status word by the FLDENV or FRSTOR instruction, the value of ES (bit 7) and its reflection in the B-bit (bit 15) are not derived from the values loaded from memory but rather are dependent upon the values of the exception flags (bits 5–0) in the status word and their corresponding masks in the control word. If ES is set in such a case, the $\overline{\text{ERROR}}$ output of the 80C187 is activated immediately.

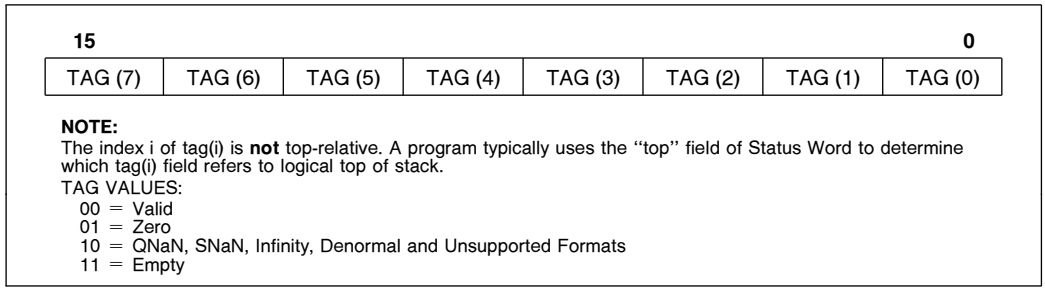


Figure 3. Tag Word

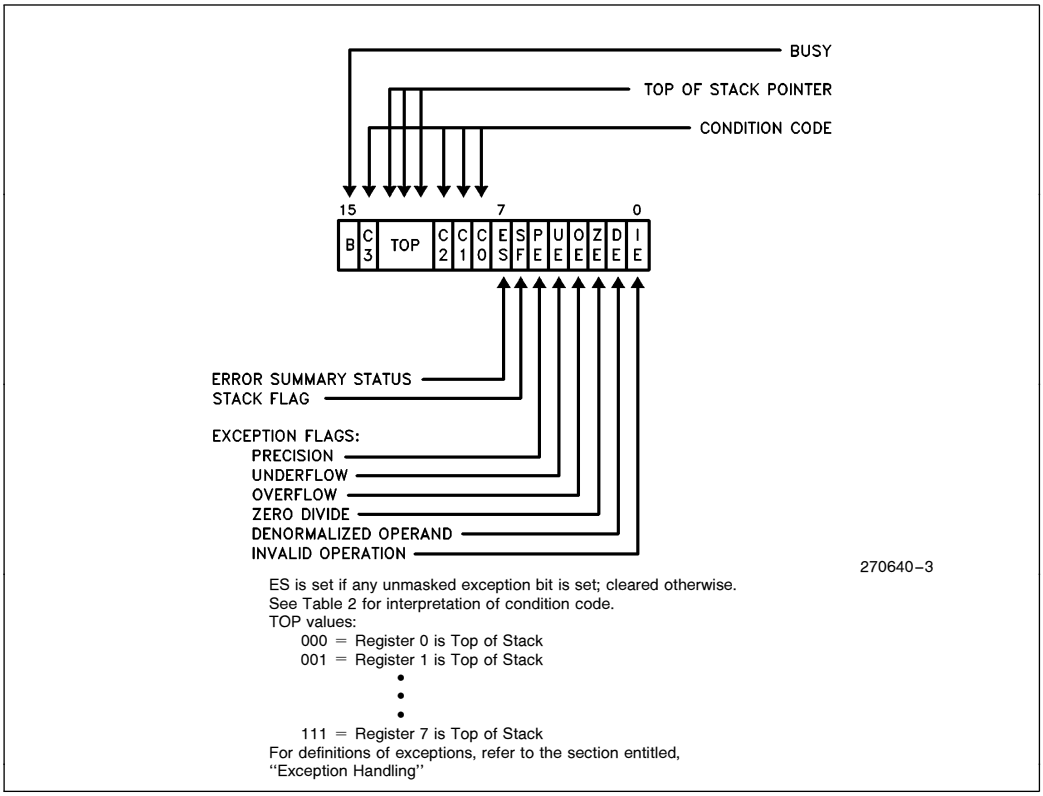


Figure 4. Status Word

CONTROL WORD

The NPX provides several processing options that are selected by loading a control word from memory into the control register. Figure 5 shows the format and encoding of fields in the control word.

Table 2. Condition Code Interpretation

Instruction	C0(S)	C3(Z)	C1(A)	C2(C)
FPREM, FPREM1 (See Table 3)	Three Least Significant Bits of Quotient Q2 Q0		Q1 or O/$\overline{U}$	Reduction 0 = Complete 1 = Incomplete
FCOM, FCOMP, FCOMPP, FTST FUCOM, FUCOMP, FUCOMPP, FICOM, FICOMP	Result of Comparison (See Table 4)		Zero or O/$\overline{U}$	Operand is not Comparable (Table 4)
FXAM	Operand Class (See Table 5)		Sign or O/$\overline{U}$	Operand Class (Table 5)
FCHS, FABS, FXCH, FINCSTP, FDECSTP, Constant Loads, FXTRACT, FLD, FILD, FBLD, FSTP (Ext Real)	UNDEFINED		Zero or O/$\overline{U}$	UNDEFINED
FIST, FBSTP, FRNDINT, FST, FSTP, FADD, FMUL, FDIV, FDIVR, FSUB, FSUBR, FSCALE, FSQRT, FPATAN, F2XM1, FYL2X, FYL2XP1	UNDEFINED		Roundup or O/$\overline{U}$	UNDEFINED
FPTAN, FSIN, FCOS, FSINCOS	UNDEFINED		Roundup or O/$\overline{U}$, Undefined if C2 = 1	Reduction 0 = Complete 1 = Incomplete
FLDENV, FRSTOR	Each Bit Loaded from Memory			
FLDCW, FSTENV, FSTCW, FSTSW, FCLEX, FINIT, FSAVE	UNDEFINED			

O/$\overline{U}$ When both IE and SF bits of status word are set, indicating a stack exception, this bit distinguishes between stack overflow (C1 = 1) and underflow (C1 = 0).

Reduction If FPREM or FPREM1 produces a remainder that is less than the modulus, reduction is complete. When reduction is incomplete the value at the top of the stack is a partial remainder, which can be used as input to further reduction. For FPTAN, FSIN, FCOS, and FSINCOS, the reduction bit is set if the operand at the top of the stack is too large. In this case the original operand remains at the top of the stack.

Roundup When the PE bit of the status word is set, this bit indicates whether one was added to the least significant bit of the result during the last rounding.

UNDEFINED Do not rely on finding any specific value in these bits.

The low-order byte of this control word configures exception masking. Bits 5–0 of the control word contain individual masks for each of the six exceptions that the 80C187 recognizes.

The high-order byte of the control word configures the 80C187 operating mode, including precision, rounding, and infinity control.

- The ''infinity control bit'' (bit 12) is not meaningful to the 80C187, and programs must ignore its value. To maintain compatibility with the 8087, this bit can be programmed; however, regardless of its value, the 80C187 always treats infinity in the affine sense ($-\infty < +\infty$). This bit is initialized to zero both after a hardware reset and after the FINIT instruction.
- The rounding control (RC) bits (bits 11–10) provide for directed rounding and true chop, as well as the unbiased round to nearest even mode specified in the IEEE standard. Rounding control affects only those instructions that perform rounding at the end of the operation (and thus can generate a precision exception); namely, FST, FSTP, FIST, all arithmetic instructions (except FPREM, FPREM1, FXTRACT, FABS, and FCHS), and all transcendental instructions.
- The precision control (PC) bits (bits 9–8) can be used to set the 80C187 internal operating precision of the significand at less than the default of 64 bits (extended precision). This can be useful in providing compatibility with early generation arithmetic processors of smaller precision. PC affects only the instructions ADD, SUB, DIV, MUL, and SQRT. For all other instructions, either the precision is determined by the opcode or extended precision is used.

Table 3. Condition Code Interpretation after FPREM and FPREM1 Instructions

Condition Code				Interpretation after FPREM and FPREM1	
C2	**C3**	**C1**	**C0**		
1	X	X	X	Incomplete Reduction: Further Iteration Required for Complete Reduction	
	Q1	**Q0**	**Q2**	**Q MOD 8**	
0	0	0	0	0	Complete Reduction: C0, C3, C1 Contain Three Least Significant Bits of Quotient
	0	1	0	1	
	1	0	0	2	
	1	1	0	3	
	0	0	1	4	
	0	1	1	5	
	1	0	1	6	
	1	1	1	7	

Table 4. Condition Code Resulting from Comparison

Order	C3	C2	C0
TOP > Operand	0	0	0
TOP < Operand	0	0	1
TOP = Operand	1	0	0
Unordered	1	1	1

Table 5. Condition Code Defining Operand Class

C3	C2	C1	C0	Value at TOP
0	0	0	0	+ Unsupported
0	0	0	1	+ NaN
0	0	1	0	− Unsupported
0	0	1	1	− NaN
0	1	0	0	+ Normal
0	1	0	1	+ Infinity
0	1	1	0	− Normal
0	1	1	1	− Infinity
1	0	0	0	+ 0
1	0	0	1	+ Empty
1	0	1	0	− 0
1	0	1	1	− Empty
1	1	0	0	+ Denormal
1	1	1	1	− Denormal

INSTRUCTION AND DATA POINTERS

Because the NPX operates in parallel with the CPU, any exceptions detected by the NPX may be reported after the CPU has executed the ESC instruction which caused it. To allow identification of the failing numerics instruction, the 80C187 contains registers that aid in diagnosis. These registers supply the opcode of the failing numerics instruction, the address of the instruction, and the address of its numerics memory operand (if appropriate).

The instruction and data pointers are provided for user-written exception handlers. Whenever the 80C187 executes a new ESC instruction, it saves the address of the instruction (including any prefixes that may be present), the address of the operand (if present), and the opcode.

The instruction and data pointers appear in the format shown by Figure 6. The ESC instruction FLDENV, FSTENV, FSAVE and FRSTOR are used to transfer these values between the registers and memory. Note that the value of the data pointer is *undefined* if the prior ESC instruction did not have a memory operand.

Interrupt Description

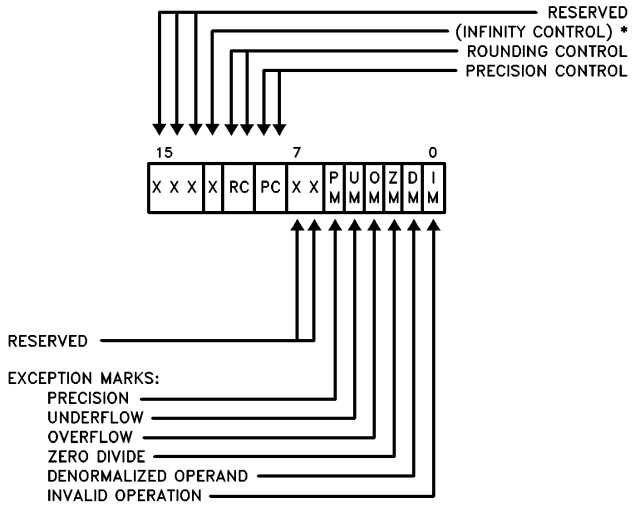
CPU interrupt 16 is used to report exceptional conditions while executing numeric programs. Interrupt 16 indicates that the previous numerics instruction caused an unmasked exception. The address of the faulty instruction and the address of its operand are stored in the instruction pointer and data pointer registers. Only ESC instructions can cause this interrupt. The CPU return address pushed onto the stack of the exception handler points to an ESC instruction (including prefixes). This instruction can be restarted after clearing the exception condition in the NPX. FNINIT, FNCLEX, FNSTSW, FNSTENV, and FNSAVE cannot cause this interrupt.

Exception Handling

The 80C187 detects six different exception conditions that can occur during instruction execution. Table 6 lists the exception conditions in order of precedence, showing for each the cause and the default action taken by the 80C187 if the exception is masked by its corresponding mask bit in the control word.

Any exception that is not masked by the control word sets the corresponding exception flag of the status word, sets the ES bit of the status word, and asserts the $\overline{\text{ERROR}}$ signal. When the CPU attempts to execute another ESC instruction, interrupt 16 occurs. The exception condition must be resolved via an interrupt service routine. The return address pushed onto the CPU stack upon entry to the service routine does not necessarily point to the failing instruction nor to the following instruction. The 80C187 saves the address of the floating-point instruction that caused the exception and the address of any memory operand required by that instruction.

If error trapping is required at the end of a series of numerics instructions (specifically, when the last ESC instruction modifies memory data and that data is used in subsequent nonnumerics instructions), it is necessary to insert the FNOP instruction to force the 80C187 to check its $\overline{\text{ERROR}}$ input.



Precision Control
00— 24 Bits (Single Precision)
01— (Reserved)
10— 53 Bits (Double Precision)
11— 64 Bits (Extended Precision)

Rounding Control
00— Round to Nearest or Even
01— Round Down (toward $-\infty$)
10— Round Up (toward $+\infty$)
11— Chop (Truncate toward Zero)

*The ''infinity control'' bit is not meaningful to the 80C187. To maintain compatibility with the 8087, this bit can be programmed; however, regardless of its value, the 80C187 treats infinity in the affine sense ($-\infty < +\infty$).

Figure 5. Control Word

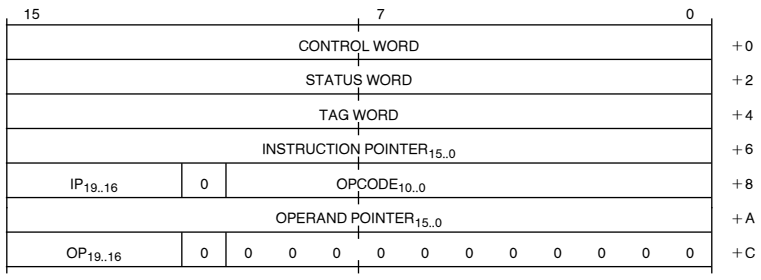


Figure 6. Instruction and Data Pointer Image in Memory

Table 6. Exceptions

Exception	Cause	Default Action (If Exception is Masked)
Invalid Operation	Operation on a signalling NaN, unsupported format, indeterminate form (0*∞, 0/0), (+∞) + (−∞), etc.), or stack overflow/underflow (SF is also set)	Result is a quiet NaN, integer indefinite, or BCD indefinite
Denormalized Operand	At least one of the operands is denormalized, i.e. it has the smallest exponent but a nonzero significand	The operand is normalized, and normal processing continues
Zero Divisor	The divisor is zero while the dividend is a noninfinite, nonzero number	Result is ∞
Overflow	The result is too large in magnitude to fit in the specified format	Result is largest finite value or ∞
Underflow	The true result is nonzero but too small to be represented in the specified format, and, if underflow exception is masked, denormalization causes loss of accuracy	Result is denormalized or zero
Inexact Result (Precision)	The true result is not exactly representable in the specified format (e.g. 1/3); the result is rounded according to the rounding mode	Normal processing continues

Initialization

After FNINIT or RESET, the control word contains the value 037FH (all exceptions masked, precision control 64 bits, rounding to nearest) the same values as in an 8087 after RESET. For compatibility with the 8087, the bit that used to indicate infinity control (bit 12) is set to zero; however, regardless of its setting, infinity is treated in the affine sense. After FNINIT or RESET, the status word is initialized as follows:

- All exceptions are set to zero.
- Stack TOP is zero, so that after the first push the stack top will be register seven (111B).
- The condition code C_3–C_0 is **undefined**.
- The B-bit is zero.

The tag word contains FFFFH (all stack locations are empty).

80C186/80C187 initialization software should execute an FNINIT instruction (i.e. an FINIT without a preceding WAIT) after RESET. The FNINIT is not strictly required for 80C187 software, but Intel recommends its use to help ensure upward compatibility with other processors.

8087 Compatibility

This section summarizes the differences between the 80C187 and the 8087. Many changes have been designed into the 80C187 to directly support the IEEE standard in hardware. These changes result in increased performance by elminating the need for software that supports the standard.

GENERAL DIFFERENCES

The 8087 instructions FENI/FNENI and FDISI/FNDISI perform no useful function in the 80C187 Numeric Processor Extension. They do not alter the state of the 80C187 Numeric Processor Extension. (They are treated similarly to FNOP, except that $\overline{\text{ERROR}}$ is not checked.) While 8086/8087 code containing these instructions can be executed on the 80C186/80C187, it is unlikely that the exception-handling routines containing these instructions will be completely portable to the 80C187 Numeric Processor Extension.

The 80C187 differs from the 8087 with respect to instruction, data, and exception synchronization. Except for the processor control instructions, all of the 80C187 numeric instructions are automatically synchronized by the 80C186 CPU. When necessary, the

80C186 automatically tests the BUSY line from the 80C187 Numeric Processor Extension to ensure that the 80C187 Numeric Processor Extension has completed its previous instruction before executing the next ESC instruction. No explicit WAIT instructions are required to assure this synchronization. For the 8087 used with 8086 and 8088 CPUs, explicit WAITs are required before each numeric instruction to ensure synchronization. Although 8086/8087 programs having explicit WAIT instructions will execute on the 80C186/80C187, these WAIT instructions are unnecessary.

The 80C187 supports only affine closure for infinity arithmetic, not projective closure.

Operands for FSCALE and FPATAN are no longer restricted in range (except for $\pm\infty$); F2XM1 and FPTAN accept a wider range of operands.

Rounding control is in effect for FLD *constant*.

Software cannot change entries of the tag word to values (other than empty) that differ from actual register contents.

After reset, FINIT, and incomplete FPREM, the 80C187 resets to zero the condition code bits C_3–C_0 of the status word.

In conformance with the IEEE standard, the 80C187 does not support the special data formats pseudozero, pseudo-NaN, pseudoinfinity, and unnormal.

The denormal exception has a different purpose on the 80C187. A system that uses the denormal-exception handler solely to normalize the denormal operands, would better mask the denormal exception on the 80C187. The 80C187 automatically normalizes denormal operands when the denormal exception is masked.

EXCEPTIONS

A number of differences exist due to changes in the IEEE standard and to functional improvements to the architecture of the 80C186/80C187:

1. The 80C186/80C187 traps exceptions only on the next ESC instruction; i.e. the 80C186 does not notice unmasked 80C187 exceptions on the 80C186 $\overline{\text{ERROR}}$ input line until a later numerics instruction is executed. Because the 80C186 does not sample $\overline{\text{ERROR}}$ on WAIT and FWAIT instructions, programmers should place an FNOP instruction at the end of a sequence of numerics instructions to force the 80C186 to sample its $\overline{\text{ERROR}}$ input.
2. The 80C187 Numeric Processor Extension signals exceptions through a dedicated $\overline{\text{ERROR}}$ line to the CPU. The 80C187 error signal does not pass through an interrupt controller (the 8087 INT signal does). Therefore, any interrupt-controller-oriented instructions in numerics exception handlers for the 8086/8087 should be deleted.
3. Interrupt vector 16 must point to the numerics exception handling routine.
4. The ESC instruction address saved in the 80C187 Numeric Processor Extension includes any leading prefixes before the ESC opcode. The corresponding address saved in the 8087 does not include leading prefixes.
5. When the overflow or underflow exception is masked, the 80C187 differs from the 8087 in rounding when overflow or underflow occurs. The 80C187 produces results that are consistent with the rounding mode.
6. When the underflow exception is masked, the 80C187 sets its underflow flag only if there is also a loss of accuracy during denormalization.
7. Fewer invalid-operation exceptions due to denormal operands, because the instructions FSQRT, FDIV, FPREM, and conversions to BCD or to integer normalize denormal operands before proceeding.
8. The FSQRT, FBSTP, and FPREM instructions may cause underflow, because they support denormal operands.
9. The denormal exception can occur during the transcendental instructions and the FXTRACT instruction.
10. The denormal exception no longer takes precedence over all other exceptions.
11. When the denormal exception is masked, the 80C187 automatically normalizes denormal operands. The 8087 performs unnormal arithmetic, which might produce an unnormal result.
12. When the operand is zero, the FXTRACT instruction reports a zero-divide exception and leaves $-\infty$ in ST(1).
13. The status word has a new bit (SF) that signals when invalid-operation exceptions are due to stack underflow or overflow.
14. FLD *extended precision* no longer reports denormal exceptions, because the instruction is not numeric.
15. FLD *single/double precision* when the operand is denormal converts the number to extended precision and signals the denormalized oper-

and exception. When loading a signalling NaN, FLD *single/double precision* signals an invalid-operand exception.

16. The 80C187 only generates quiet NaNs (as on the 8087); however, the 80C187 distinguishes between quiet NaNs and signalling NaNs. Signalling NaNs trigger exceptions when they are used as operands; quiet NaNs do not (except for FCOM, FIST, and FBSTP which also raise IE for quiet NaNs).
17. When stack overflow occurs during FPTAN and overflow is masked, both ST(0) and ST(1) contain quiet NaNs. The 8087 leaves the original operand in ST(1) intact.
18. When the scaling factor is $\pm\infty$, the FSCALE (ST(0), ST(1) instruction behaves as follows (ST(0) and ST(1) contain the scaled and scaling operands respectively):
    - FSCALE (0, ∞) generates the invalid operation exception.
    - FSCALE (finite, $-\infty$) generates zero with the same sign as the scaled operand.
    - FSCALE (finite, $+\infty$) generates ∞ with the same sign as the scaled operand.

    The 8087 returns zero in the first case and raises the invalid-operation exception in the other cases.
19. The 80C187 returns signed infinity/zero as the unmasked response to massive overflow/underflow. The 8087 supports a limited range for the scaling factor; within this range either massive overflow/underflow do not occur or undefined results are produced.

Table 7. Pin Summary

Pin Name	Function	Active State	Input/ Output
CLK	CLocK		I
CKM	ClocKing Mode		I
RESET	System reset	High	I
PEREQ	Processor Extension REQuest	High	O
BUSY	Busy status	High	O
$\overline{\text{ERROR}}$	Error status	Low	O
$D_{15}-D_0$	Data pins	High	I/O
$\overline{\text{NPRD}}$	Numeric Processor ReaD	Low	I
$\overline{\text{NPWR}}$	Numeric Processor WRite	Low	I
$\overline{\text{NPS1}}$	NPX select #1	Low	I
NPS2	NPX select #2	High	I
CMD0	CoMmanD 0	High	I
CMD1	CoMmanD 1	High	I
V_{CC}	System power		I
V_{SS}	System ground		I

HARDWARE INTERFACE

In the following description of hardware interface, an overbar above a signal name indicates that the active or asserted state occurs when the signal is at a low voltage. When no overbar is present above the signal name, the signal is asserted when at the high voltage level.

Signal Description

In the following signal descriptions, the 80C187 pins are grouped by function as follows:

1. Execution Control— CLK, CKM, RESET
2. NPX Handshake— PEREQ, BUSY, $\overline{\text{ERROR}}$
3. Bus Interface Pins— D_{15}–D_0, $\overline{\text{NPWR}}$, $\overline{\text{NPRD}}$
4. Chip/Port Select— $\overline{\text{NPS1}}$, NPS2, CMD0, CMD1
5. Power Supplies— V_{CC}, V_{SS}

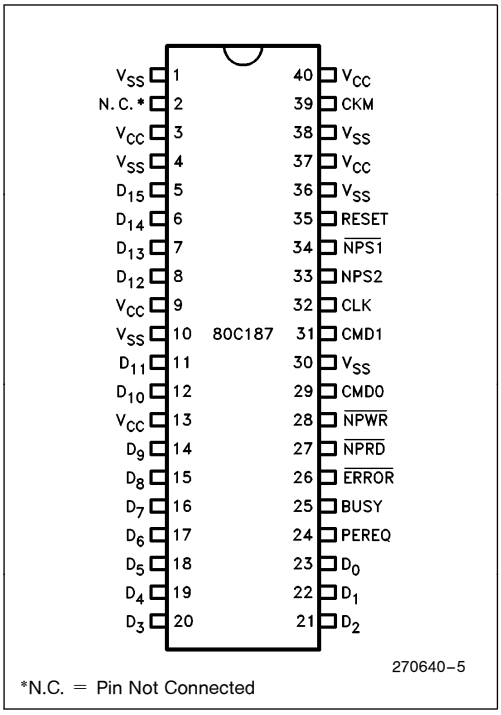
Table 7 lists every pin by its identifier, gives a brief description of its function, and lists some of its characteristics. Figure 7 shows the locations of pins on the CERDIP package, while Figure 8 shows the locations of pins on the PLCC package. Table 8 helps to locate pin identifiers in Figures 7 and 8.

Clock (CLK)

This input provides the basic timing for internal operation. This pin does not require MOS-level input; it will operate at either TTL or MOS levels up to the maximum allowed frequency. A minimum frequency must be provided to keep the internal logic properly functioning. Depending on the signal on CKM, the signal on CLK can be divided by two to produce the internal clock signal (in which case CLK may be up to 32 MHz in frequency), or can be used directly (in which case CLK may be up to 12.5 MHz).

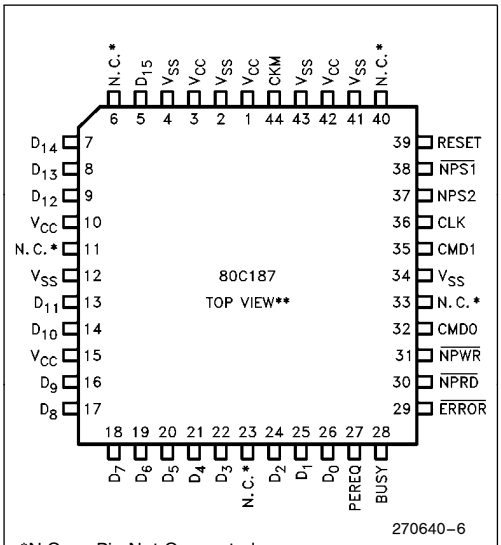
Clocking Mode (CKM)

This pin is a strapping option. When it is strapped to V_{CC} (HIGH), the CLK input is used directly; when strapped to V_{SS} (LOW), the CLK input is divided by two to produce the internal clock signal. During the RESET sequence, this input must be stable at least four internal clock cycles (i.e. CLK clocks when CKM is HIGH; 2 × CLK clocks when CKM is LOW) before RESET goes LOW.



*N.C. = Pin Not Connected

Figure 7. CERDIP Pin Configuration



*N.C. = Pin Not Connected
**"Top View" means as the package is seen from the component side of the board.

Figure 8. PLCC Pin Configuration

Table 8. PLCC Pin Cross-Reference

Pin Name	CERDIP Package	PLCC Package
BUSY	25	28
CKM	39	44
CLK	32	36
CMD0	29	32
CMD1	31	35
D_0	23	26
D_1	22	25
D_2	21	24
D_3	20	22
D_4	19	21
D_5	18	20
D_6	17	19
D_7	16	18
D_8	15	17
D_9	14	16
D_{10}	12	14
D_{11}	11	13
D_{12}	8	9
D_{13}	7	8
D_{14}	6	7
D_{15}	5	5
$\overline{\text{ERROR}}$	26	29
No Connect	2	6, 11, 23, 33, 40
$\overline{\text{NPRD}}$	27	30
$\overline{\text{NPS1}}$	34	38
NPS2	33	37
$\overline{\text{NPWR}}$	28	31
PEREQ	24	27
RESET	35	39
V_{CC}	3, 9, 13, 37, 40	1, 3, 10, 15, 42
V_{SS}	1, 4, 10, 30, 36, 38	2, 4, 12, 34, 41, 43

System Reset (RESET)

A LOW to HIGH transition on this pin causes the 80C187 to terminate its present activity and to enter a dormant state. RESET must remain active (HIGH) for at least four internal clock periods. (The relation of the internal clock period to CLK depends on CLKM; the internal clock may be different from that of the CPU.) Note that the 80C187 is active internally for 25 clock periods after the termination of the RESET signal (the HIGH to LOW transition of RESET); therefore, the first instruction should not be written to the 80C187 until 25 internal clocks after the falling edge of RESET. Table 9 shows the status of the output pins during the reset sequence. After a reset, all output pins return to their inactive states.

Table 9. Output Pin Status during Reset

Output Pin Name	Value during Reset
BUSY	HIGH
$\overline{\text{ERROR}}$	HIGH
PEREQ	LOW
D_{15}–D_0	TRI-STATE OFF

Processor Extension Request (PEREQ)

When active, this pin signals to the CPU that the 80C187 is ready for data transfer to/from its data FIFO. When there are more than five data transfers,

PEREQ is deactivated after the first three transfers and subsequently after every four transfers. This signal always goes inactive before BUSY goes inactive.

Busy Status (BUSY)

When active, this pin signals to the CPU that the 80C187 is currently executing an instruction. This pin is active HIGH. It should be connected to the 80C186's $\overline{TEST}$/BUSY pin. During the RESET sequence this pin is HIGH. The 80C186 uses this HIGH state to detect the presence of an 80C187.

Error Status ($\overline{ERROR}$)

This pin reflects the ES bit of the status register. When active, it indicates that an unmasked exception has occurred. This signal can be changed to inactive state only by the following instructions (without a preceding WAIT): FNINIT, FNCLEX, FNSTENV, FNSAVE, FLDCW, FLDENV, and FRSTOR. This pin should be connected to the $\overline{ERROR}$ pin of the CPU. $\overline{ERROR}$ can change state only when BUSY is active.

Data Pins (D_{15}–D_0)

These bidirectional pins are used to transfer data and opcodes between the CPU and 80C187. They are normally connected directly to the corresponding CPU data pins. Other buffers/drivers driving the local data bus must be disabled when the CPU reads from the NPX. High state indicates a value of one. D_0 is the least significant data bit.

Numeric Processor Write ($\overline{NPWR}$)

A signal on this pin enables transfers of data from the CPU to the NPX. This input is valid only when $\overline{NPS1}$ and NPS2 are both active.

Numeric Processor Read ($\overline{NPRD}$)

A signal on this pin enables transfers of data from the NPX to the CPU. This input is valid only when $\overline{NPS1}$ and NPS2 are both active.

Numeric Processor Selects ($\overline{NPS1}$ and NPS2)

Concurrent assertion of these signals indicates that the CPU is performing an escape instruction and enables the 80C187 to execute that instruction. No data transfer involving the 80C187 occurs unless the device is selected by these lines.

Command Selects (CMD0 and CMD1)

These pins along with the select pins allow the CPU to direct the operation of the 80C187.

System Power (V_{CC})

System power provides the $+5V \pm 10\%$ DC supply input. All V_{CC} pins should be tied together on the circuit board and local decoupling capacitors should be used between V_{CC} and V_{SS}.

System Ground (V_{SS})

All V_{SS} pins should be tied together on the circuit board and local decoupling capacitors should be used between V_{CC} and V_{SS}.

Processor Architecture

As shown by the block diagram (Figure 1), the 80C187 NPX is internally divided into three sections: the bus control logic (BCL), the data interface and control unit, and the floating-point unit (FPU). The FPU (with the support of the control unit which contains the sequencer and other support units) executes all numerics instructions. The data interface and control unit is responsible for the data flow to and from the FPU and the control registers, for receiving the instructions, decoding them, and sequencing the microinstructions, and for handling some of the administrative instructions. The BCL is responsible for CPU bus tracking and interface.

BUS CONTROL LOGIC

The BCL communicates solely with the CPU using I/O bus cycles. The BCL appears to the CPU as a special peripheral device. It is special in two respects: the CPU initiates I/O automatically when it encounters ESC instructions, and the CPU uses reserved I/O addresses to communicate with the BCL. The BCL does not communicate directly with memory. The CPU performs all memory access, transferring input operands from memory to the 80C187 and transferring outputs from the 80C187 to memory. A dedicated communication protocol makes possible high-speed transfer of opcodes and operands between the CPU and 80C187.

Table 10. Bus Cycles Definition

$\overline{NPS1}$	NPS2	CMD0	CMD1	$\overline{NPRD}$	$\overline{NPWR}$	Bus Cycle Type
x	0	x	x	x	x	80C187 Not Selected
1	x	x	x	x	x	80C187 Not Selected
0	1	0	0	1	0	Opcode Write to 80C187
0	1	0	0	0	1	CW or SW Read from 80C187
0	1	1	0	0	1	Read Data from 80C187
0	1	1	0	1	0	Write Data to 80C187
0	1	0	1	1	0	Write Exception Pointers
0	1	0	1	0	1	Reserved
0	1	1	1	0	1	Read Opcode Status
0	1	1	1	1	0	Reserved

DATA INTERFACE AND CONTROL UNIT

The data interface and control unit latches the data and, subject to BCL control, directs the data to the FIFO or the instruction decoder. The instruction decoder decodes the ESC instructions sent to it by the CPU and generates controls that direct the data flow in the FIFO. It also triggers the microinstruction sequencer that controls execution of each instruction. If the ESC instruction is FINIT, FCLEX, FSTSW, FSTSW AX, FSTCW, FSETPM, or FRSTPM, the control executes it independently of the FPU and the sequencer. The data interface and control unit is the one that generates the BUSY, PEREQ, and $\overline{ERROR}$ signals that synchronize 80C187 activities with the CPU.

FLOATING-POINT UNIT

The FPU executes all instructions that involve the register stack, including arithmetic, logical, transcendental, constant, and data transfer instructions. The data path in the FPU is 84 bits wide (68 significant bits, 15 exponent bits, and a sign bit) which allows internal operand transfers to be performed at very high speeds.

Bus Cycles

The pins $\overline{NPS1}$, NPS2, CMD0, CMD1, $\overline{NPRD}$ and $\overline{NPWR}$ identify bus cycles for the NPX. Table 10 defines the types of 80C187 bus cycles.

80C187 ADDRESSING

The $\overline{NPS1}$, NPS2, CMD0, and CMD1 signals allow the NPX to identify which bus cycles are intended for the NPX. The NPX responds to I/O cycles when the I/O address is 00F8H, 00FAH, 00FCH, or 00FEH. The correspondence betwen I/O addresses and control signals is defined by Table 11. To guarantee correct operation of the NPX, programs must not perform any I/O operations to these reserved port addresses.

Table 11. I/O Address Decoding

I/O Address (Hexadecimal)	80C187 Select and Command Inputs			
	NPS2	$\overline{NPS1}$	CMD1	CMD0
00F8	1	0	0	0
00FA	1	0	0	1
00FC	1	0	1	0
00FE	1	0	1	1

CPU/NPX SYNCHRONIZATION

The pins BUSY, PEREQ, and $\overline{\text{ERROR}}$ are used for various aspects of synchronization between the CPU and the NPX.

BUSY is used to synchronize instruction transfer from the CPU to the 80C187. When the 80C187 recognizes an ESC instruction, it asserts BUSY. For most ESC instructions, the CPU waits for the 80C187 to deassert BUSY before sending the new opcode.

The NPX uses the PEREQ pin of the CPU to signal that the NPX is ready for data transfer to or from its data FIFO. The NPX does not directly access memory; rather, the CPU provides memory access services for the NPX.

Once the CPU initiates an 80C187 instruction that has operands, the CPU waits for PEREQ signals that indicate when the 80C187 is ready for operand transfer. Once all operands have been transferred (or if the instruction has no operands) the CPU continues program execution while the 80C187 executes the ESC instruction.

In 8086/8087 systems, WAIT instructions are required to achieve synchronization of both commands and operands. The 80C187, however, does not require WAIT instructions. The WAIT or FWAIT instruction commonly inserted by high-level compilers and assembly-language programmers for exception synchronization is not treated as an instruction by the 80C186 and does not provide exception trapping. (Refer to the section ''System Configuration for 8087-Compatible Exception Trapping''.)

Once it has started to execute a numerics instruction and has transferred the operands from the CPU, the 80C187 can process the instruction in parallel with and independent of the host CPU. When the NPX detects an exception, it asserts the $\overline{\text{ERROR}}$ signal, which causes a CPU interrupt.

OPCODE INTERPRETATION

The CPU and the NPX use a bus protocol that adapts to the numerics opcode being executed. Only the NPX directly interprets the opcode. Some of the results of this interpretation are relevant to the CPU. The NPX records these results (opcode status information) in an internal 16-bit register. The 80C186 accesses this register only via reads from NPX port 00FEH. Tables 10 and 11 define the signal combinations that correspond to each of the following steps.

1. The CPU writes the opcode to NPX port 00F8H. This write can occur even when the NPX is busy or is signalling an exception. The NPX does not necessarily begin executing the opcode immediately.
2. The CPU reads the opcode status information from NPX port 00FEH.
3. The CPU initiates subsequent bus cycles according to the opcode status information. The opcode status information specifies whether to wait until the NPX is not busy, when to transfer exception pointers to port 00FCH, when to read or write operands and results at port 00FAH, etc.

For most instructions, the NPX does not start executing the previously transferred opcode until the CPU (guided by the opcode status information) first writes exception pointer information to port 00FCH of the NPX. This protocol is completely transparent to programmers.

Bus Operation

With respect to bus interface, the 80C187 is fully asynchronous with the CPU, even when it operates from the same clock source as the CPU. The CPU initiates a bus cycle for the NPX by activating both $\overline{\text{NPS1}}$ and NPS2, the NPX select signals. During the CLK period in which $\overline{\text{NPS1}}$ and NPS2 are activated, the 80C187 also examines the $\overline{\text{NPRD}}$ and $\overline{\text{NPRW}}$

input signals to determine whether the cycle is a read or a write cycle and examines the CMD0 and CMD1 inputs to determine whether an opcode, operand, or control/status register transfer is to occur. The 80C187 activates its BUSY output some time after the leading edge of the NPRD or NPRW signal. Input and ouput data are referenced to the trailing edges of the NPRD and NPRW signals.

The 80C187 activates the PEREQ signal when it is ready for data transfer. The 80C187 deactivates PEREQ automatically.

System Configuration

The 80C187 can be connected to the 80C186 CPU as shown by Figure 9. (Refer to the 80C186 Data Sheet for an explanation of the 80C186's signals.) This interface has the following characteristics:

- The 80C187's NPS1, ERROR, PEREQ, and BUSY pins are connected directly to the corresponding pins of the 80C186.
- The 80C186 pin MCS3/NPS is connected to NPS1; NPS2 is connected to V_{CC}. Note that if the 80C186 CPU's DEN signal is used to gate external data buffers, it must be combined with the NPS signal to insure numeric accesses will not activate these buffers.
- The NPRD and NPRW pins are connected to the RD and WR pins of the 80C186.
- CMD1 and CMD0 come from the latched A_2 and A_1 of the 80C186, respectively.
- The 80C187 BUSY output connects to the 80C186 TEST/BUSY input. During RESET, the signal at the 80C187 BUSY output automatically programs the 80C186 to use the 80C187.
- The 80C187 can use the CLKOUT signal of the 80C186 to conserve board space when operating at 12.5 MHz or less. In this case, the 80C187 CKM input must be pulled HIGH. For operation in excess of 12.5 MHz, a double-frequency external oscillator for CLK input is needed. In this case, CKM must be pulled LOW.

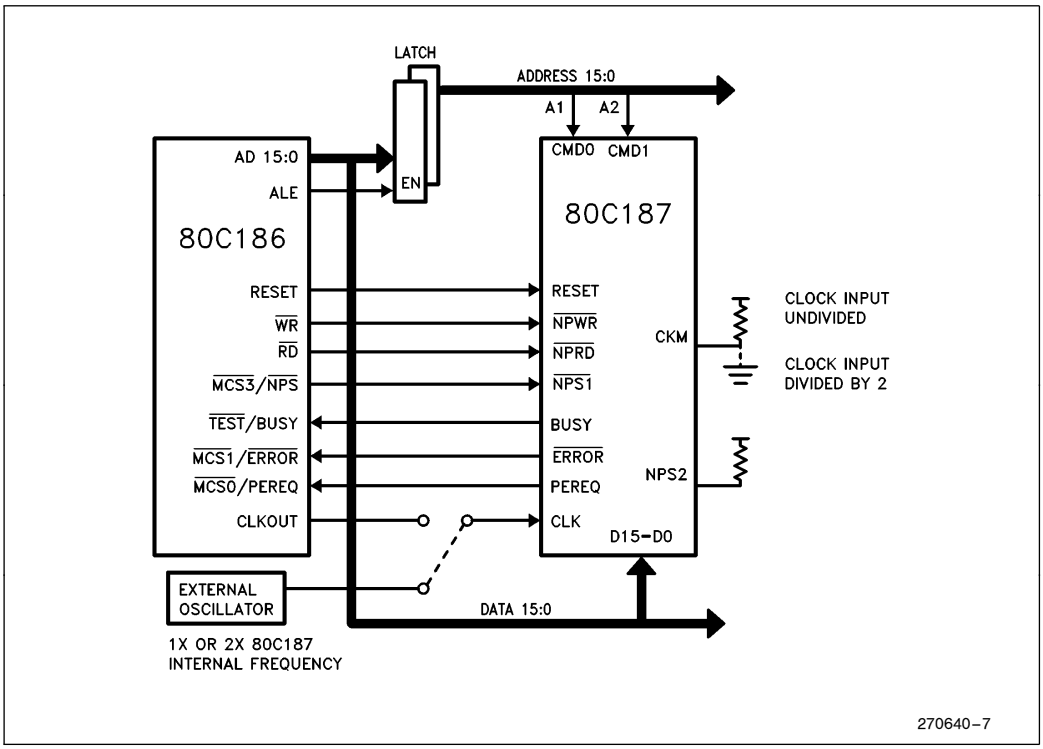
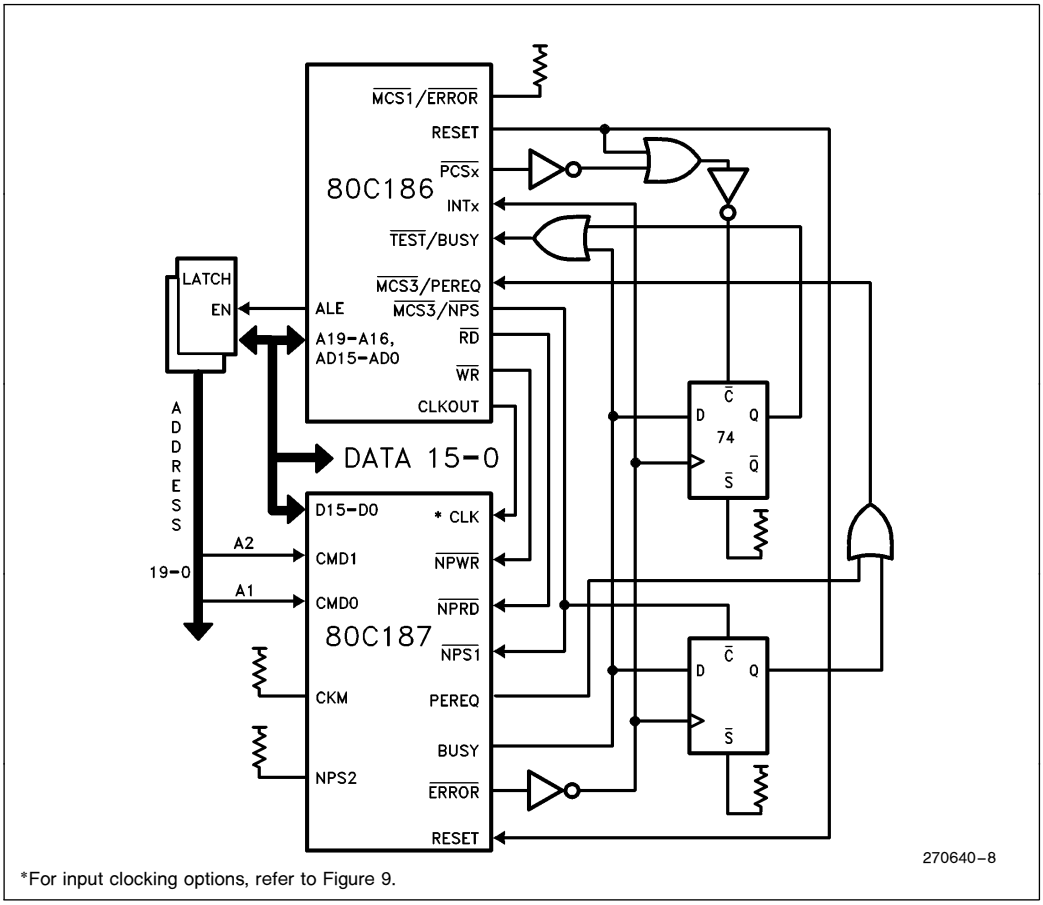


Figure 9. 80C186/80C187 System Configuration

System Configuration for 80186/ 80187-Compatible Exception Trapping

When the 80C187 $\overline{\text{ERROR}}$ output signal is connected directly to the 80C186 $\overline{\text{ERROR}}$ input, floating-point exceptions cause interrupt #16. However, existing software may be programmed to expect floating-point exceptions to be signalled over an external interrupt pin via an interrupt controller.

For exception handling compatible with the 80186/ 82188/8087, the 80C186 can be wired to recognize exceptions through an external interrupt pin, as Figure 10 shows. (Refer to the 80C186 Data Sheet for an explanation of the 80C186's signals.) With this arrangement, a flip-flop is needed to latch BUSY upon assertion of $\overline{\text{ERROR}}$. The latch can then be cleared during the exception-handler routine by forcing a $\overline{\text{PCS}}$ pin active. The latch must also be cleared at RESET in order for the 80C186 to work with the 80C187.



*For input clocking options, refer to Figure 9.

Figure 10. System Configuration for 8087-Compatible Exception Trapping

ELECTRICAL DATA

NOTICE: This is a production data sheet. The specifications are subject to change without notice.

Absolute Maximum Ratings*

Case Temperature Under Bias (T_C) . . . 0°C to +85°C

Storage Temperature . . . . . −65°C to +150°C

Voltage on Any Pin
with Respect to Ground . . . . −0.5V to V_{CC} +0.5V

Power Dissipation 1.5W

WARNING: Stressing the device beyond the "Absolute Maximum Ratings" may cause permanent damage. These are stress ratings only. Operation beyond the "Operating Conditions" is not recommended and extended exposure beyond the "Operating Conditions" may affect device reliability.

Power and Frequency Requirements

The typical relationship between I_{CC} and the frequency of operation F is as follows:

$I_{CC_{typ}} = 55 + 5 * F$ mA where F is in MHz.

When the frequency is reduced below the minimum operating frequency specified in the AC Characteristics table, the internal states of the 80C187 may become indeterminate. The 80C187 clock cannot be stopped; otherwise, I_{CC} would increase significantly beyond what the equation above indicates.

DC Characteristics T_C = 0°C to +85°C, V_{CC} = +5V ±10%

Symbol	Parameter	Min	Max	Units	Test Conditions
V_{IL}	Input LOW Voltage	−0.5	+0.8	V	
V_{IH}	Input HIGH Voltage	2.0	V_{CC} +0.5	V	
V_{ICL}	Clock Input LOW Voltage	−0.5	+0.8	V	
V_{ICH}	Clock Input HIGH Voltage	2.0	V_{CC} +0.5	V	
V_{OL}	Output LOW Voltage		0.45	V	I_{OL} = 3.0 mA
V_{OH}	Output HIGH Voltage	2.4		V	I_{OH} = −0.4 mA
I_{CC}	Power Supply Current		156 135	mA mA	16 MHz 12.5 MHz
I_{LI}	Input Leakage Current		±10	μA	$0V \le V_{IN} \le V_{CC}$
I_{LO}	I/O Leakage Current		±10	μA	$0.45V \le V_{OUT} \le V_{CC} - 0.45V$
C_{IN}	Input Capacitance		10	pF	F_C = 1 MHz
C_O	I/O or Output Capacitance		12	pF	F_C = 1 MHz
C_{CLK}	Clock Capacitance		20	pF	F_C = 1 MHz

AC Characteristics

T_C = 0°C to +85°C, V_{CC} = 5V ±10%
All timings are measured at 1.5V unless otherwise specified

Symbol	Parameter	12.5 MHz		16 MHz		Test Conditions
		Min (ns)	Max (ns)	Min (ns)	Max (ns)	
T_{dvwh} (t6)	Data Setup to $\overline{NPWR}$	43		33		
T_{whdx} (t7)	Data Hold from $\overline{NPWR}$	14		14		
T_{rlrh} (t8)	$\overline{NPRD}$ Active Time	59		54		
T_{wlwh} (t9)	$\overline{NPWR}$ Active Time	59		54		
T_{avwl} (t10)	Command Valid to $\overline{NPWR}$	0		0		
T_{avrl} (t11)	Command Valid to $\overline{NPRD}$	0		0		
T_{mhrl} (t12)	Min Delay from PEREQ Active to $\overline{NPRD}$ Active	40		30		
T_{whax} (t18)	Command Hold from $\overline{NPWR}$	12		8		
T_{rhax} (t19)	Command Hold from $\overline{NPRD}$	12		8		
T_{ivcl} (t20)	$\overline{NPRD}$, $\overline{NPWR}$, RESET to CLK Setup Time	46		38		Note 1
T_{clih} (t21)	$\overline{NPRD}$, $\overline{NPWR}$, RESET from CLK Hold Time	26		18		Note 1
T_{rscl} (t24)	RESET to CLK Setup	21		19		Note 1
T_{clrs} (t25)	RESET from CLK Hold	14		9		Note 1
T_{cmdi} (t26)	Command Inactive Time					
	Write to Write	69		59		
	Read to Read	69		59		
	Read to Write	69		59		
	Write to Read	69		59		

NOTE:
1. This is an asynchronous input. This specification is given for testing purposes only, to assure recognition at a specific CLK edge.

Timing Responses

All timings are measured at 1.5V unless otherwise specified

Symbol	Parameter	12.5 MHz		16 MHz		Test Conditions
		Min (ns)	Max (ns)	Min (ns)	Max (ns)	
T_{rhqz} (t27) T_{rlqv} (t28)	$\overline{NPRD}$ Inactive to Data Float* $\overline{NPRD}$ Active to Data Valid		18 50		18 45	Note 2 Note 3
T_{ilbh} (t29)	$\overline{ERROR}$ Active to Busy Inactive	104		104		Note 4
T_{wlbv} (t30)	$\overline{NPWR}$ Active to Busy Active		80		60	Note 4
T_{klml} (t31)	$\overline{NPRD}$ or $\overline{NPWR}$ Active to PEREQ Inactive		80		60	Note 5
T_{rhqh} (t32)	Data Hold from $\overline{NPRD}$ Inactive	2		2		Note 3
T_{rlbh} (t33)	RESET Inactive to BUSY Inactive		80		60	

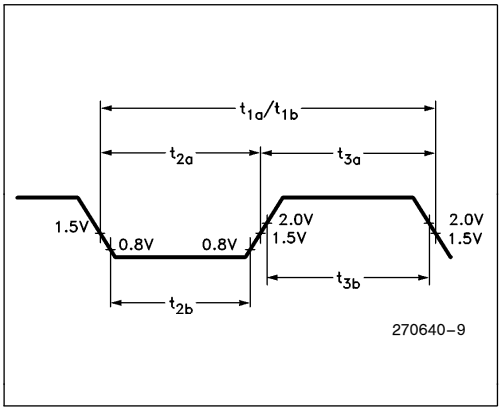
NOTES:
*The data float delay is not tested.
2. The float condition occurs when the measured output current is less than I_{OL} on $D_{15}-D_0$.
3. $D_{15}-D_0$ loading: $C_L = 100$ pF.
4. BUSY loading: $C_L = 100$ pF.
5. On last data transfer of numeric instruction.

Clock Timings

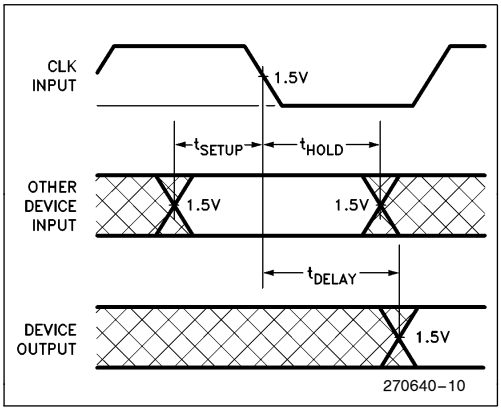
Symbol	Parameter		12.5 MHz		16 MHz*		Test Conditions
			Min (ns)	Max (ns)	Min (ns)	Max (ns)	
T_{clcl} (t1a) (t1B)	CLK Period	CKM = 1 CKM = 0	80 40	250 125	N/A 31.25	N/A 125	Note 6 Note 6
T_{clch} (t2a) (t2b)	CLK Low Time	CKM = 1 CKM = 0	35 9		N/A 7		Note 6 Note 7
T_{chcl} (t3a) (t3b)	CLK High Time	CKM = 1 CKM = 0	35 13		N/A 9		Note 6 Note 8
T_{ch2ch1}(t4)				10		8	Note 9
T_{ch1ch2}(t5)				10		8	Note 10

NOTES:
*16 MHz operation is available only in divide-by-2 mode (CKM strapped LOW).
6. At 1.5V
7. At 0.8V
8. At 2.0V
9. CKM = 1: 3.7V to 0.8V at 16 MHz, 3.5V to 1.0V at 12.5 MHz
10. CKM = 1: 0.8V to 3.7V at 16 MHz, 1.0V to 3.5V at 12.5 MHz

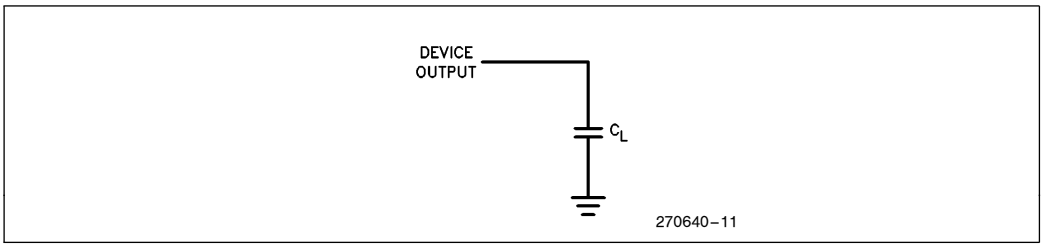
AC DRIVE AND MEASUREMENT POINTS—CLK INPUT



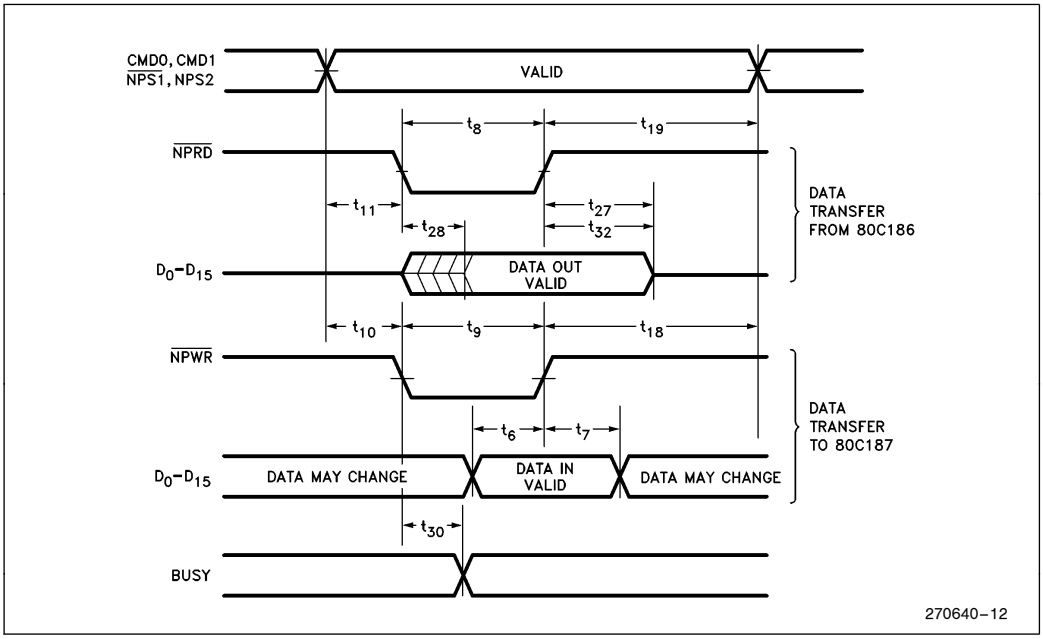
AC SETUP, HOLD, AND DELAY TIME MEASUREMENTS—GENERAL



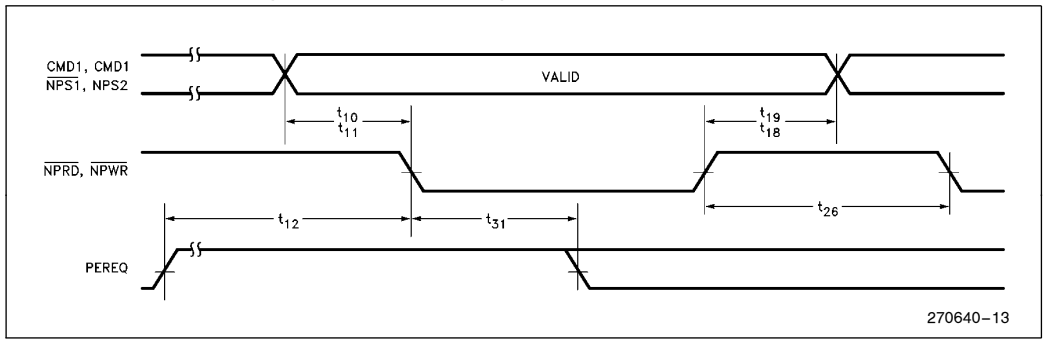
AC TEST LOADING ON OUTPUTS



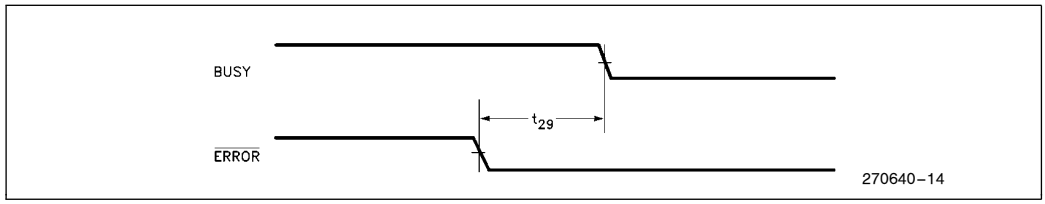
DATA TRANSFER TIMING (INITIATED BY CPU)



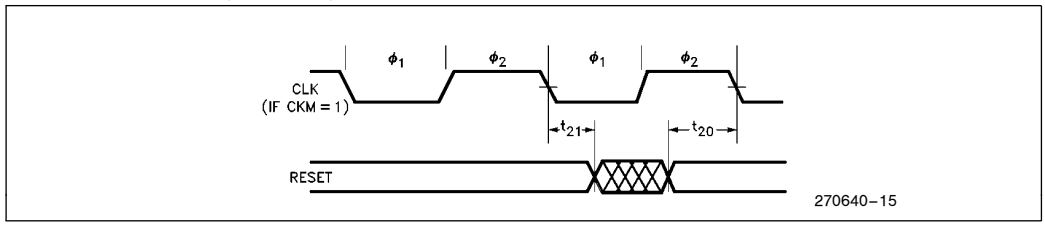
DATA CHANNEL TIMING (INITIATED BY 80C187)



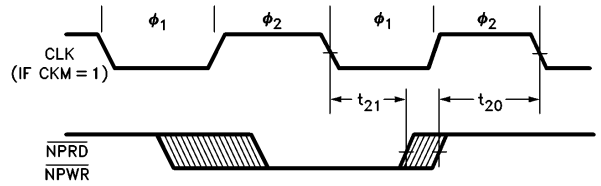
ERROR OUTPUT TIMING



CLK, RESET TIMING (CKM = 1)

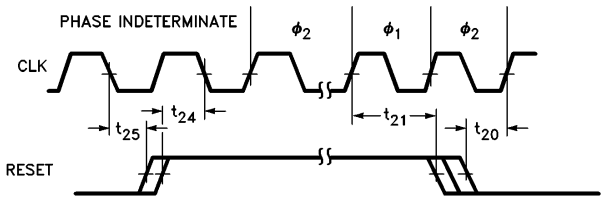


CLK, $\overline{\text{NPRD}}$, $\overline{\text{NPWR}}$ TIMING (CKM = 1)



270640–16

CLK, RESET TIMING (CKM = 0)

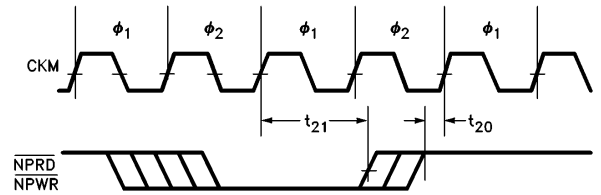


270640–17

RESET must meet timing shown to guarantee known phase of internal divide by 2 circuits.

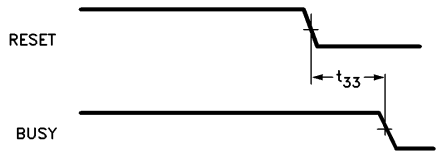
NOTE:
RESET, $\overline{\text{NPWR}}$, $\overline{\text{NPRD}}$ inputs are asynchronous to CLK. Timing requirements are given for testing purposes only, to assure recognition at a specific CLK edge.

CLK, $\overline{\text{NPRD}}$, $\overline{\text{NPWR}}$ TIMING (CKM = 0)



270640–18

RESET, BUSY TIMING



270640–19

80C187 EXTENSIONS TO THE CPU's INSTRUCTION SET

Instructions for the 80C187 assume one of the five forms shown in Table 12. In all cases, instructions are at least two bytes long and begin with the bit pattern 11011B, which identifies the ESCAPE class of instruction. Instructions that refer to memory operands specify addresses using the CPU's addressing modes.

MOD (Mode field) and R/M (Register/Memory specifier) have the same interpretation as the corresponding fields of CPU instructions (refer to Programmer's Reference Manual for the CPU). The DISP (displacement) is optionally present in instructions that have MOD and R/M fields. Its presence depends on the values of MOD and R/M, as for instructions of the CPU.

The instruction summaries that follow assume that the instruction has been prefetched, decoded, and is ready for execution; that bus cycles do not require wait states; that there are no local bus HOLD requests delaying processor access to the bus; and that no exceptions are detected during instruction execution. Timings are given in internal 80C187 clocks and include the time for opcode and data transfer between the CPU and the NPX. If the instruction has MOD and R/M fields that call for both base and index registers, add one clock.

Table 12. Instruction Formats

	Instruction									Optional Field
	First Byte				Second Byte					
1	11011	OPA		1	MOD		1	OPB	R/M	DISP
2	11011	MF		OPA	MOD		OPB *		R/M	DISP
3	11011	d	P	OPA	1	1	OPB *		ST (i)	
4	11011	0	0	1	1	1	1	OP		
5	11011	0	1	1	1	1	1	OP		
	15–11	10	9	8	7	6	5	4 3	2 1 0	

NOTES:

OP = Instruction opcode, possibly split into two fields OPA and OPB

MF = Memory Format
- 00— 32-Bit Real
- 01— 32-Bit Integer
- 10— 64-Bit Real
- 11— 16-Bit Integer

d = Destination
- 0— Destination is ST(0)
- 0— Destination is ST(i)

R XOR d = 0— Destination (op) Source
R XOR d = 1— Source (op) Destination

*In FSUB and FDIV, the low-order bit of OPB is the R (reversed) bit

P = Pop
- 0— Do not pop stack
- 1— Pop stack after operation

ESC = 11011

ST(i) = Register Stack Element *i*
- 000 = Stack Top
- 001 = Second Stack Element
- •
- •
- •
- 111 = Eighth Stack Element

80C187 Extensions to the 80C186 Instruction Set

Instruction	Encoding			Clock Count Range			
	Byte 0	Byte 1	Optional Bytes 2–3	32-Bit Real	32-Bit Integer	64-Bit Real	16-Bit Integer
DATA TRANSFER							
FLD = Load[a]							
Integer/real memory to ST(0)	ESC MF 1	MOD 000 R/M	DISP	40	65–72	59	67–71
Long integer memory to ST(0)	ESC 111	MOD 101 R/M	DISP		90–101		
Extended real memory to ST(0)	ESC 011	MOD 101 R/M	DISP		74		
BCD memory to ST(0)	ESC 111	MOD 100 R/M	DISP		296–305		
ST(i) to ST(0)	ESC 001	11000 ST(i)			16		
FST = Store							
ST(0) to integer/real memory	ESC MF 1	MOD 010 R/M	DISP	58	93–107	73	80–93
ST(0) to ST(i)	ESC 101	11010 ST(i)			13		
FSTP = Store and Pop							
ST(0) to integer/real memory	ESC MF 1	MOD 011 R/M	DISP	58	93–107	73	80–93
ST(0) to long integer memory	ESC 111	MOD 111 R/M	DISP		116–133		
ST(0) to extended real	ESC 011	MOD 111 R/M	DISP		83		
ST(0) to BCD memory	ESC 111	MOD 110 R/M	DISP		542–564		
ST(0) to ST(i)	ESC 101	11001 ST (i)			14		
FXCH = Exchange							
ST(i) and ST(0)	ESC 001	11001 ST(i)			20		
COMPARISON							
FCOM = Compare							
Integer/real memory to ST(0)	ESC MF 0	MOD 010 R/M	DISP	48	78–85	67	77–81
ST(i) to ST(0)	ESC 000	11010 ST(i)			26		
FCOMP = Compare and pop							
Integer/real memory to ST	ESC MF 0	MOD 011 R/M	DISP	48	78–85	67	77–81
ST(i) to ST(0)	ESC 000	11011 ST(i)			28		
FCOMPP = Compare and pop twice							
ST(1) to ST(0)	ESC 110	1101 1001			28		
FTST = Test ST(0)	ESC 001	1110 0100			30		
FUCOM = Unordered compare	ESC 101	11100 ST(i)			26		
FUCOMP = Unordered compare and pop	ESC 101	11101 ST(i)			28		
FUCOMPP = Unordered compare and pop twice	ESC 010	1110 1001			28		
FXAM = Examine ST(0)	ESC 001	11100101			32-40		
CONSTANTS							
FLDZ = Load +0.0 into ST(0)	ESC 001	1110 1110			22		
FLD1 = Load +1.0 into ST(0)	ESC 001	1110 1000			26		
FLDPI = Load pi into ST(0)	ESC 001	1110 1011			42		
FLDL2T = Load $\log_2(10)$ into ST(0)	ESC 001	1110 1001			42		

Shaded areas indicate instructions not available in 8087.

NOTE:
a. When loading single- or double-precision zero from memory, add 5 clocks.

80C187 Extensions to the 80C186 Instruction Set (Continued)

Instruction	Encoding			Clock Count Range			
	Byte 0	Byte 1	Optional Bytes 2–3	32-Bit Real	32-Bit Integer	64-Bit Real	16-Bit Integer
CONSTANTS (Continued)							
FLDL2E = Load $\log_2(e)$ into ST(0)	ESC 001	1110 1010			42		
FLDLG2 = Load $\log_{10}(2)$ into ST(0)	ESC 001	1110 1100			43		
FLDLN2 = Load $\log_e(2)$ into ST(0)	ESC 001	1110 1101			43		
ARITHMETIC							
FADD = Add							
Integer/real memory with ST(0)	ESC MF 0	MOD 000 R/M	DISP	44–52	77–92	65–73	77–91
ST(i) and ST(0)	ESC d P 0	11000 ST(i)			25–33[b]		
FSUB = Subtract							
Integer/real memory with ST(0)	ESC MF 0	MOD 10 R R/M	DISP	44–52	77–92	65–73	77–91[c]
ST(i) and ST(0)	ESC d P 0	1110 R R/M			28–36[d]		
FMUL = Multiply							
Integer/real memory with ST(0)	ESC MF 0	MOD 001 R/M	DISP	47–57	81–102	68–93	82–93
ST(i) and ST(0)	ESC d P 0	1100 1 R/M			31–59[e]		
FDIV = Divide							
Integer/real memory with ST(0)	ESC MF 0	MOD 11 R R/M	DISP	108	140–147[f]	128	142–146[g]
ST(i) and ST(0)	ESC d P 0	1111 R R/M			90[h]		
FSQRT[i] = Square root	ESC 001	1111 1010			124–131		
FSCALE = Scale ST(0) by ST(1)	ESC 001	1111 1101			69–88		
FPREM = Partial remainder of ST(0) ÷ ST(1)	ESC 001	1111 1000			76–157		
FPREM1 = Partial remainder (IEEE)	ESC 001	1111 0101			97–187		
FRNDINT = Round ST(0) to integer	ESC 001	1111 1100			68–82		
FXTRACT = Extract components of ST(0)	ESC 001	1111 0100			72–78		
FABS = Absolute value of ST(0)	ESC 001	1110 0001			24		
FCHS = Change sign of ST(0)	ESC 001	1110 0000			26–27		

Shaded areas indicate instructions not available in 8087.

NOTES:
b. Add 3 clocks to the range when d = 1.
c. Add 1 clock to **each** range when R = 1.
d. Add 3 clocks to the range when d = 0.
e. typical = 54 (When d = 0, 48–56, typical = 51).
f. Add 1 clock to the range when R = 1.
g. 153–159 when R = 1.
h. Add 3 clocks to the range when d = 1.
i. $-0 \leq$ ST(0) $\leq +\infty$.

80C187 Extensions to the 80C186 Instruction Set (Continued)

Instruction	Encoding			Clock Count Range
	Byte 0	Byte 1	Optional Bytes 2–3	
TRANSCENDENTAL				
FCOS = Cosine of ST(0)	ESC 001	1111 1111		125–774[j]
FPTAN[k] = Partial tangent of ST(0)	ESC 001	1111 0010		193–499[j]
FPATAN = Partial arctangent	ESC 001	1111 0011		316–489
FSIN = Sine of ST(0)	ESC 001	1111 1110		124–773[j]
FSINCOS = Sine and cosine of ST(0)	ESC 001	1111 1011		196–811[j]
F2XM1[l] = $2^{ST(0)} - 1$	ESC 001	1111 0000		213–478
FYL2X[m] = ST(1) * $\log_2(ST(0))$	ESC 001	1111 0001		122–540
FYL2XP1[n] = ST(1) * $\log_2(ST(0) + 1.0)$	ESC 001	1111 1001		259–549
PROCESSOR CONTROL				
FINIT = Initialize NPX	ESC 011	1110 0011		35
FSTSW AX = Store status word	ESC 111	1110 0000		17
FLDCW = Load control word	ESC 001	MOD 101 R/M	DISP	23
FSTCW = Store control word	ESC 001	MOD 111 R/M	DISP	21
FSTSW = Store status word	ESC 101	MOD 111 R/M	DISP	21
FCLEX = Clear exceptions	ESC 011	1110 0010		13
FSTENV = Store environment	ESC 001	MOD 110 R/M	DISP	146
FLDENV = Load environment	ESC 001	MOD 100 R/M	DISP	113
FSAVE = Save state	ESC 101	MOD 110 R/M	DISP	550
FRSTOR = Restore state	ESC 101	MOD 100 R/M	DISP	482
FINCSTP = Increment stack pointer	ESC 001	1111 0111		23
FDECSTP = Decrement stack pointer	ESC 001	1111 0110		24
FFREE = Free ST(i)	ESC 101	1100 0 ST(i)		20
FNOP = No operations	ESC 001	1101 0000		14

Shaded areas indicate instructions not available in 8087.

NOTES:
j. These timings hold for operands in the range $|x| < \pi/4$. For operands not in this range, up to 78 clocks may be needed to reduce the operand.
k. $0 \leq |ST(0)| < 2^{63}$.
l. $-1.0 \leq ST(0) \leq 1.0$.
m. $0 \leq ST(0) < \infty$, $-\infty < ST(1) < +\infty$.
n. $0 \leq |ST(0)| < (2 - \sqrt{}(2))/2$, $-\infty < ST(1) < +\infty$.

DATA SHEET REVISION REVIEW

The following list represents the key differences between the -002 and the -001 version of the 80C187 data sheet. Please review this summary carefully.

1. Figure 10, titled ‘‘System Configuration for 8087—Compatible Exception Trapping’’, was replaced with a revised schematic. The previous configuration was faulty. Updated timing diagrams on Data Transfer Timing, Error Output, and RESET/BUSY.

DATA SHEET

80C31/80C32

80C51 8-bit microcontroller family

128/256 byte RAM ROMless low voltage (2.7 V–5.5 V), low power, high speed (33 MHz)

Product specification
IC28 Data Handbook

2000 Aug 07

Philips
Semiconductors

PHILIPS

80C51 8-bit microcontroller family
128/256 byte RAM ROMless low voltage (2.7V–5.5V), low power, high speed (33 MHz)

80C31/80C32

DESCRIPTION

The Philips 80C31/32 is a high-performance static 80C51 design fabricated with Philips high-density CMOS technology with operation from 2.7 V to 5.5 V.

The 80C31/32 ROMless devices contain a 128 × 8 RAM/256 × 8 RAM, 32 I/O lines, three 16-bit counter/timers, a six-source, four-priority level nested interrupt structure, a serial I/O port for either multi-processor communications, I/O expansion or full duplex UART, and on-chip oscillator and clock circuits.

In addition, the device is a low power static design which offers a wide range of operating frequencies down to zero. Two software selectable modes of power reduction—idle mode and power-down mode are available. The idle mode freezes the CPU while allowing the RAM, timers, serial port, and interrupt system to continue functioning. The power-down mode saves the RAM contents but freezes the oscillator, causing all other chip functions to be inoperative. Since the design is static, the clock can be stopped without loss of user data and then the execution resumed from the point the clock was stopped.

SELECTION TABLE

For applications requiring more ROM and RAM, see the 8XC54/58 and 8XC51RA+/RB+/RC+/80C51RA+ data sheet.

ROM/EPROM Memory Size (X by 8)	RAM Size (X by 8)	Programmable Timer Counter (PCA)	Hardware Watch Dog Timer
80C31/8XC51			
0K/4K	128	No	No
80C32/8XC52/54/58			
0K/8K/16K/32K	256	No	No
80C51RA+/8XC51RA+/RB+/RC+			
0K/8K/16K/32K	512	Yes	Yes
8XC51RD+			
64K	1024	Yes	Yes

FEATURES

- 8051 Central Processing Unit
  - 128 × 8 RAM (80C31)
  - 256 × 8 RAM (80C32)
  - Three 16-bit counter/timers
  - Boolean processor
  - Full static operation
  - Low voltage (2.7 V to 5.5 V@ 16 MHz) operation
- Memory addressing capability
  - 64k ROM and 64k RAM
- Power control modes:
  - Clock can be stopped and resumed
  - Idle mode
  - Power-down mode
- CMOS and TTL compatible
- TWO speed ranges at V_{CC} = 5 V
  - 0 to 16 MHz
  - 0 to 33 MHz
- Three package styles
- Extended temperature ranges
- Dual Data Pointers
- 4 level priority interrupt
- 6 interrupt sources
- Four 8-bit I/O ports
- Full–duplex enhanced UART
  - Framing error detection
  - Automatic address recognition
- Programmable clock out
- Asynchronous port reset
- Low EMI (inhibit ALE)
- Wake-up from Power Down by an external interrupt

80C51/87C51 AND 80C31 ORDERING INFORMATION

ROMless	TEMPERATURE RANGE °C AND PACKAGE	VOLTAGE RANGE	FREQ. (MHz)	DRAWING NUMBER
P80C31SBPN	0 to +70, Plastic Dual In-line Package	2.7 V to 5.5 V	0 to 16	SOT129-1
P80C31SBAA	0 to +70, Plastic Leaded Chip Carrier	2.7 V to 5.5 V	0 to 16	SOT187-2
P80C31SBBB	0 to +70, Plastic Quad Flat Pack	2.7 V to 5.5 V	0 to 16	SOT307-2
P80C31SFPN	–40 to +85, Plastic Dual In-line Package	2.7 V to 5.5 V	0 to 16	SOT129-1
P80C31SFAA	–40 to +85, Plastic Leaded Chip Carrier	2.7 V to 5.5 V	0 to 16	SOT187-2
P80C31SFBB	–40 to +85, Plastic Quad Flat Pack	2.7 V to 5.5 V	0 to 16	SOT307-2

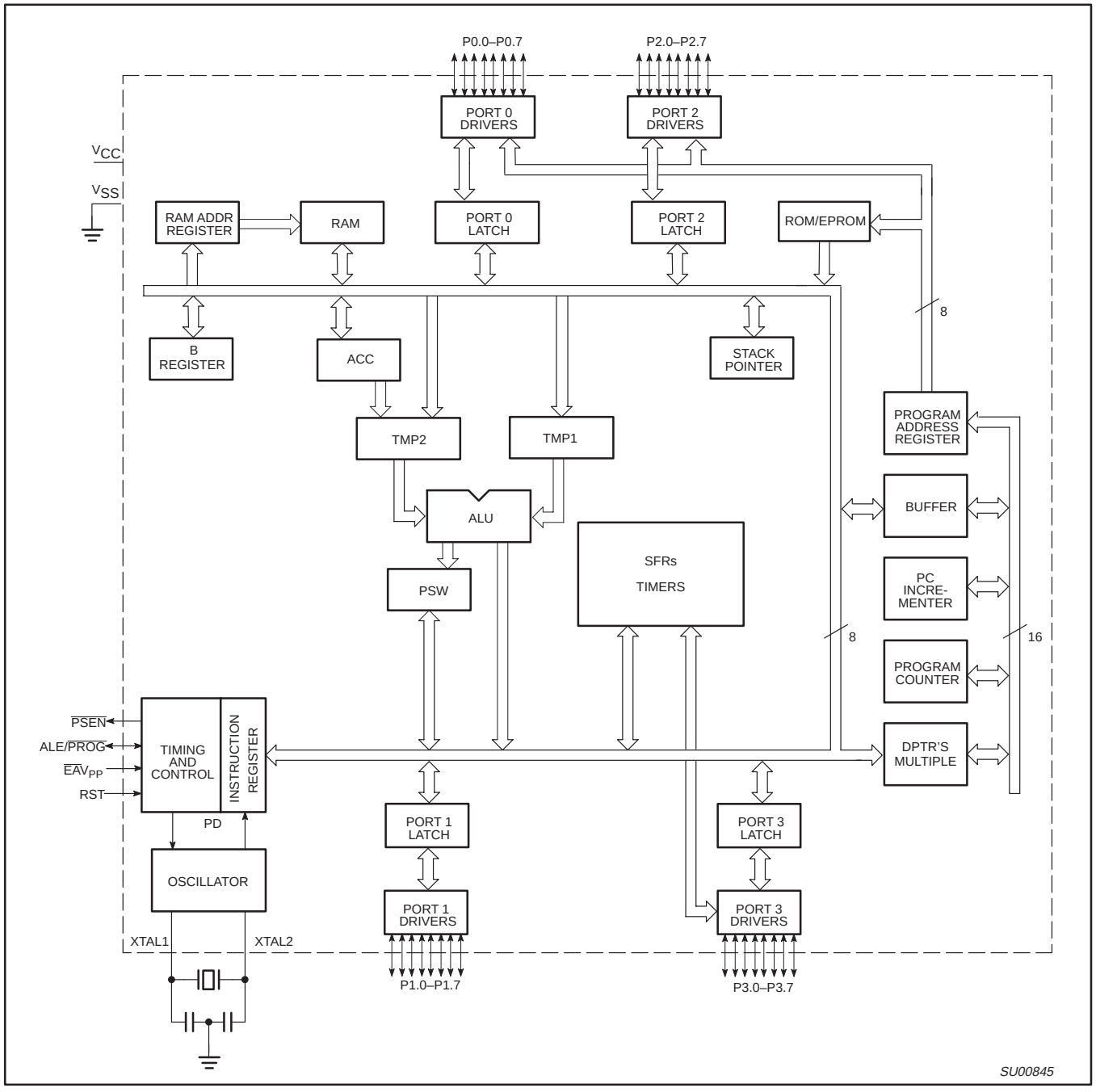
PART NUMBER DERIVATION

DEVICE NUMBER	OPERATING FREQUENCY, MAX (S)	TEMPERATURE RANGE (B)	PACKAGE (AA)
P80C31	S = 16 MHz	B = 0° to +70°C	AA = PLCC
P80C32	U = 33 MHz	F = –40°C to +85°C	BB = PQFP
			PN = PDIP

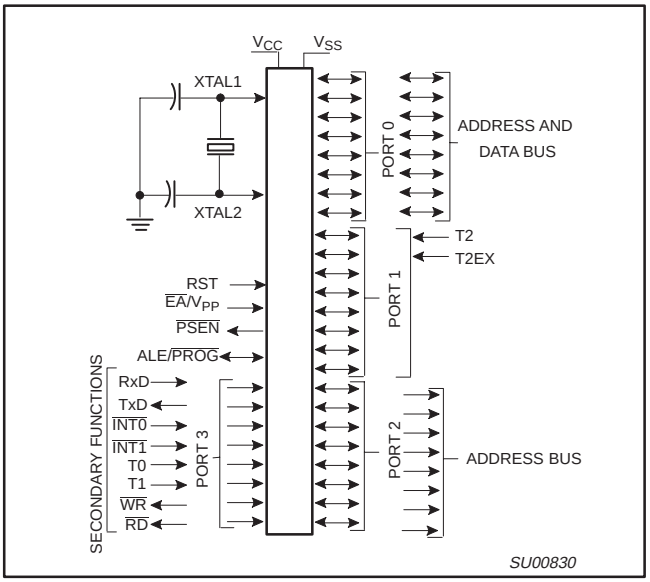
80C32 ORDERING INFORMATION

ROMless	TEMPERATURE RANGE °C AND PACKAGE	FREQ MHz	DRAWING NUMBER
P80C32SBP N	0 to +70, Plastic Dual In-line Package	16	SOT129-1
P80C32SBA A	0 to +70, Plastic Leaded Chip Carrier	16	SOT187-2
P80C32SBB B	0 to +70, Plastic Quad Flat Pack	16	SOT307-2
P80C32SFP N	–40 to +85, Plastic Dual In-line Package	16	SOT129-1
P80C32SFA A	–40 to +85, Plastic Leaded Chip Carrier	16	SOT187-2
P80C32SFB B	–40 to +85, Plastic Quad Flat Pack	16	SOT307-2
P80C32UBA A	0 to +70, Plastic Leaded Chip Carrier	33	SOT187-2
P80C32UBP N	0 to +70, Plastic Dual In-line Package	33	SOT129-1
P80C32UBB B	0 to +70, Plastic Quad Flat Pack	33	SOT307-2
P80C32UFA A	–40 to +85, Plastic Leaded Chip Carrier	33	SOT187-2
P80C32UFP N	–40 to +85, Plastic Dual In-line Package	33	SOT129-1
P80C32UFB B	–40 to +85, Plastic Quad Flat Pack	33	SOT307-2

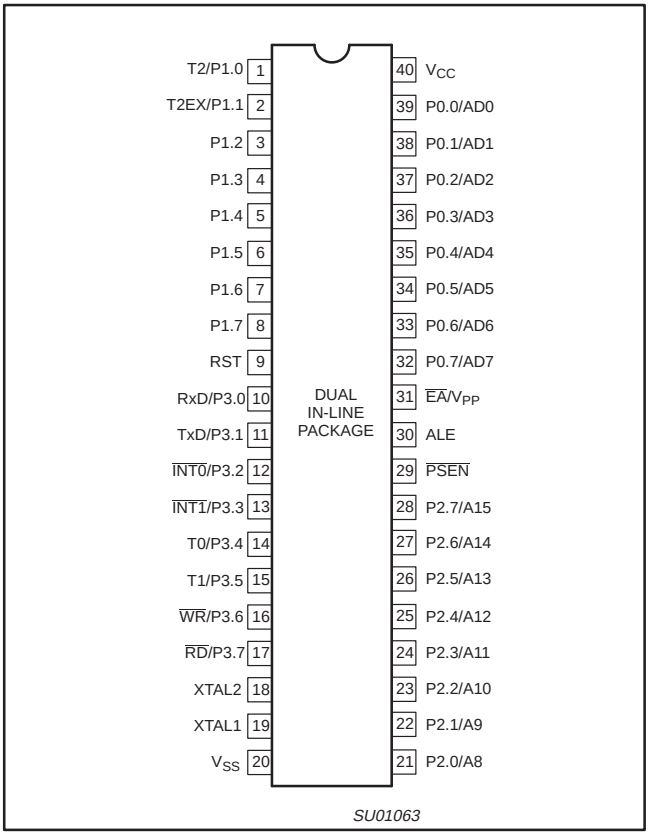
BLOCK DIAGRAM



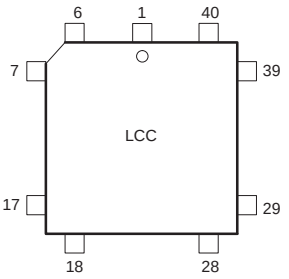
LOGIC SYMBOL



PIN CONFIGURATIONS



PLASTIC LEADED CHIP CARRIER PIN FUNCTIONS

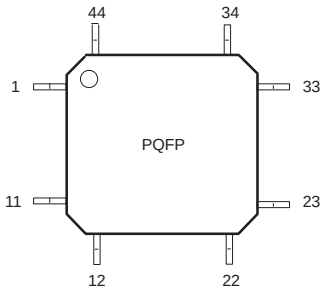


Pin	Function	Pin	Function	Pin	Function
1	NIC*	16	P3.4/T0	31	P2.7/A15
2	P1.0/T2	17	P3.5/T1	32	PSEN
3	P1.1/T2EX	18	P3.6/WR	33	ALE
4	P1.2	19	P3.7/RD	34	NIC*
5	P1.3	20	XTAL2	35	EA/VPP
6	P1.4	21	XTAL1	36	P0.7/AD7
7	P1.5	22	VSS	37	P0.6/AD6
8	P1.6	23	NIC*	38	P0.5/AD5
9	P1.7	24	P2.0/A8	39	P0.4/AD4
10	RST	25	P2.1/A9	40	P0.3/AD3
11	P3.0/RxD	26	P2.2/A10	41	P0.2/AD2
12	NIC*	27	P2.3/A11	42	P0.1/AD1
13	P3.1/TxD	28	P2.4/A12	43	P0.0/AD0
14	P3.2/INT0	29	P2.5/A13	44	VCC
15	P3.3/INT1	30	P2.6/A14		

* NO INTERNAL CONNECTION

SU01062

PLASTIC QUAD FLAT PACK PIN FUNCTIONS



Pin	Function	Pin	Function	Pin	Function
1	P1.5	16	VSS	31	P0.6/AD6
2	P1.6	17	NIC*	32	P0.5/AD5
3	P1.7	18	P2.0/A8	33	P0.4/AD4
4	RST	19	P2.1/A9	34	P0.3/AD3
5	P3.0/RxD	20	P2.2/A10	35	P0.2/AD2
6	NIC*	21	P2.3/A11	36	P0.1/AD1
7	P3.1/TxD	22	P2.4/A12	37	P0.0/AD0
8	P3.2/INT0	23	P2.5/A13	38	VCC
9	P3.3/INT1	24	P2.6/A14	39	NIC*
10	P3.4/T0	25	P2.7/A15	40	P1.0/T2
11	P3.5/T1	26	PSEN	41	P1.1/T2EX
12	P3.6/WR	27	ALE	42	P1.2
13	P3.7/RD	28	NIC*	43	P1.3
14	XTAL2	29	EA/VPP	44	P1.4
15	XTAL1	30	P0.7/AD7		

* NO INTERNAL CONNECTION

SU01064

PIN DESCRIPTIONS

MNEMONIC	PIN NUMBER DIP	LCC	QFP	TYPE	NAME AND FUNCTION
V_{SS}	20	22	16	I	**Ground:** 0 V reference.
V_{CC}	40	44	38	I	**Power Supply:** This is the power supply voltage for normal, idle, and power-down operation.
P0.0–0.7	39–32	43–36	37–30	I/O	**Port 0:** Port 0 is an open-drain, bidirectional I/O port with Schmitt trigger inputs. Port 0 pins that have 1s written to them float and can be used as high-impedance inputs. Port 0 is also the multiplexed low-order address and data bus during accesses to external program and data memory. In this application, it uses strong internal pull-ups when emitting 1s.
P1.0–P1.7	1–8	2–9	40–44, 1–3	I/O	**Port 1:** Port 1 is an 8-bit bidirectional I/O port with internal pull-ups and Schmitt trigger inputs. Port 1 pins that have 1s written to them are pulled high by the internal pull-ups and can be used as inputs. As inputs, port 1 pins that are externally pulled low will source current because of the internal pull-ups. (See DC Electrical Characteristics: I_{IL}). Alternate functions for Port 1 include:
	1	2	40	I/O	**T2 (P1.0):** Timer/Counter 2 external count input/clockout (see Programmable Clock-Out)
	2	3	41	I	**T2EX (P1.1):** Timer/Counter 2 Reload/Capture/Direction control
P2.0–P2.7	21–28	24–31	18–25	I/O	**Port 2:** Port 2 is an 8-bit bidirectional I/O port with internal pull-ups and Schmitt trigger inputs. Port 2 pins that have 1s written to them are pulled high by the internal pull-ups and can be used as inputs. As inputs, port 2 pins that are externally being pulled low will source current because of the internal pull-ups. (See DC Electrical Characteristics: I_{IL}). Port 2 emits the high-order address byte during fetches from external program memory and during accesses to external data memory that use 16-bit addresses (MOVX @DPTR). In this application, it uses strong internal pull-ups when emitting 1s. During accesses to external data memory that use 8-bit addresses (MOV @Ri), port 2 emits the contents of the P2 special function register.
P3.0–P3.7	10–17	11, 13–19	5, 7–13	I/O	**Port 3:** Port 3 is an 8-bit bidirectional I/O port with internal pull-ups and Schmitt trigger inputs. Port 3 pins that have 1s written to them are pulled high by the internal pull-ups and can be used as inputs. As inputs, port 3 pins that are externally being pulled low will source current because of the pull-ups. (See DC Electrical Characteristics: I_{IL}). Port 3 also serves the special features of the 80C51 family, as listed below:
	10	11	5	I	**RxD (P3.0):** Serial input port
	11	13	7	O	**TxD (P3.1):** Serial output port
	12	14	8	I	**$\overline{\text{INT0}}$ (P3.2):** External interrupt
	13	15	9	I	**$\overline{\text{INT1}}$ (P3.3):** External interrupt
	14	16	10	I	**T0 (P3.4):** Timer 0 external input
	15	17	11	I	**T1 (P3.5):** Timer 1 external input
	16	18	12	O	**$\overline{\text{WR}}$ (P3.6):** External data memory write strobe
	17	19	13	O	**$\overline{\text{RD}}$ (P3.7):** External data memory read strobe
RST	9	10	4	I	**Reset:** A high on this pin for two machine cycles while the oscillator is running, resets the device. An internal diffused resistor to V_{SS} permits a power-on reset using only an external capacitor to V_{CC}.
ALE	30	33	27	O	**Address Latch Enable:** Output pulse for latching the low byte of the address during an access to external memory. In normal operation, ALE is emitted at a constant rate of 1/6 the oscillator frequency, and can be used for external timing or clocking. Note that one ALE pulse is skipped during each access to external data memory. ALE can be disabled by setting SFR auxiliary.0. With this bit set, ALE will be active only during a MOVX instruction.
$\overline{\text{PSEN}}$	29	32	26	O	**Program Store Enable:** The read strobe to external program memory. When the 80C31/32 is executing code from the external program memory, $\overline{\text{PSEN}}$ is activated twice each machine cycle, except that two $\overline{\text{PSEN}}$ activations are skipped during each access to external data memory. $\overline{\text{PSEN}}$ is not activated during fetches from internal program memory.
$\overline{\text{EA}}/V_{PP}$	31	35	29	I	**External Access Enable/Programming Supply Voltage:** $\overline{\text{EA}}$ must be externally held low to enable the device to fetch code from external program memory locations 0000H to 0FFFH.
XTAL1	19	21	15	I	**Crystal 1:** Input to the inverting oscillator amplifier and input to the internal clock generator circuits.
XTAL2	18	20	14	O	**Crystal 2:** Output from the inverting oscillator amplifier.

NOTE:
To avoid “latch-up” effect at power-on, the voltage on any pin at any time must not be higher than V_{CC} + 0.5 V or V_{SS} – 0.5 V, respectively.

Table 1. 8XC51/80C31 Special Function Registers

SYMBOL	DESCRIPTION	DIRECT ADDRESS	BIT ADDRESS, SYMBOL, OR ALTERNATIVE PORT FUNCTION MSB							LSB	RESET VALUE
ACC*	Accumulator	E0H	E7	E6	E5	E4	E3	E2	E1	E0	00H
AUXR#	Auxiliary	8EH	–	–	–	–	–	–	–	AO	xxxxxxx0B
AUXR1#	Auxiliary 1	A2H	–	–	–	–	WUPD[2]	0	–	DPS	xxx000x0B
B*	B register	F0H	F7	F6	F5	F4	F3	F2	F1	F0	00H
DPTR:	Data Pointer (2 bytes)										
DPH	Data Pointer High	83H									00H
DPL	Data Pointer Low	82H									00H
			AF	AE	AD	AC	AB	AA	A9	A8	
IE*	Interrupt Enable	A8H	EA	–	ET2	ES	ET1	EX1	ET0	EX0	0x000000B
			BF	BE	BD	BC	BB	BA	B9	B8	
IP*	Interrupt Priority	B8H	–	–	PT2	PS	PT1	PX1	PT0	PX0	xx000000B
			B7	B6	B5	B4	B3	B2	B1	B0	
IPH#	Interrupt Priority High	B7H	–	–	PT2H	PSH	PT1H	PX1H	PT0H	PX0H	xx000000B
			87	86	85	84	83	82	81	80	
P0*	Port 0	80H	AD7	AD6	AD5	AD4	AD3	AD2	AD1	AD0	FFH
			97	96	95	94	93	92	91	90	
P1*	Port 1	90H	–	–	–	–	–	–	T2EX	T2	FFH
			A7	A6	A5	A4	A3	A2	A1	A0	
P2*	Port 2	A0H	AD15	AD14	AD13	AD12	AD11	AD10	AD9	AD8	FFH
			B7	B6	B5	B4	B3	B2	B1	B0	
P3*	Port 3	B0H	$\overline{RD}$	$\overline{WR}$	T1	T0	$\overline{INT1}$	$\overline{INT0}$	TxD	RxD	FFH
PCON#[1]	Power Control	87H	SMOD1	SMOD0	–	POF	GF1	GF0	PD	IDL	00xx0000B
			D7	D6	D5	D4	D3	D2	D1	D0	
PSW*	Program Status Word	D0H	CY	AC	F0	RS1	RS0	OV	–	P	000000x0B
RACAP2H#	Timer 2 Capture High	CBH									00H
RACAP2L#	Timer 2 Capture Low	CAH									00H
SADDR#	Slave Address	A9H									00H
SADEN#	Slave Address Mask	B9H									00H
SBUF	Serial Data Buffer	99H									xxxxxxxxB
			9F	9E	9D	9C	9B	9A	99	98	
SCON*	Serial Control	98H	SM0/FE	SM1	SM2	REN	TB8	RB8	TI	RI	00H
SP	Stack Pointer	81H									07H
			8F	8E	8D	8C	8B	8A	89	88	
TCON*	Timer Control	88H	TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0	00H
			CF	CE	CD	CC	CB	CA	C9	C8	
T2CON*	Timer 2 Control	C8H	TF2	EXF2	RCLK	TCLK	EXEN2	TR2	C/$\overline{T2}$	CP/$\overline{RL2}$	00H
T2MOD#	Timer 2 Mode Control	C9H	–	–	–	–	–	–	T2OE	DCEN	xxxxxx00B
TH0	Timer High 0	8CH									00H
TH1	Timer High 1	8DH									00H
TH2#	Timer High 2	CDH									00H
TL0	Timer Low 0	8AH									00H
TL1	Timer Low 1	8BH									00H
TL2#	Timer Low 2	CCH									00H
TMOD	Timer Mode	89H	GATE	C/$\overline{T}$	M1	M0	GATE	C/$\overline{T}$	M1	M0	00H

NOTE:
Unused register bits that are not defined should not be set by the user's program. If violated, the device could function incorrectly.
* SFRs are bit addressable.
\# SFRs are modified from or added to the 80C51 SFRs.
– Reserved bits.
1. Reset value depends on reset source.
2. Not available on 80C31.

OSCILLATOR CHARACTERISTICS

XTAL1 and XTAL2 are the input and output, respectively, of an inverting amplifier. The pins can be configured for use as an on-chip oscillator, as shown in the logic symbol.

To drive the device from an external clock source, XTAL1 should be driven while XTAL2 is left unconnected. There are no requirements on the duty cycle of the external clock signal, because the input to the internal clock circuitry is through a divide-by-two flip-flop. However, minimum and maximum high and low times specified in the data sheet must be observed.

Reset

A reset is accomplished by holding the RST pin high for at least two machine cycles (24 oscillator periods), while the oscillator is running. To insure a good power-up reset, the RST pin must be high long enough to allow the oscillator time to start up (normally a few milliseconds) plus two machine cycles.

Stop Clock Mode

The static design enables the clock speed to be reduced down to 0 MHz (stopped). When the oscillator is stopped, the RAM and Special Function Registers retain their values. This mode allows step-by-step utilization and permits reduced system power consumption by lowering the clock frequency down to any value. For lowest power consumption the Power Down mode is suggested.

Idle Mode

In idle mode (see Table 2), the CPU puts itself to sleep while all of the on-chip peripherals stay active. The instruction to invoke the idle mode is the last instruction executed in the normal operating mode before the idle mode is activated. The CPU contents, the on-chip RAM, and all of the special function registers remain intact during this mode. The idle mode can be terminated either by any enabled interrupt (at which time the process is picked up at the interrupt service routine and continued), or by a hardware reset which starts the processor in the same manner as a power-on reset.

Power-Down Mode

To save even more power, a Power Down mode (see Table 2) can be invoked by software. In this mode, the oscillator is stopped and the instruction that invoked Power Down is the last instruction executed. The on-chip RAM and Special Function Registers retain their values down to 2.0 V and care must be taken to return V_{CC} to the minimum specified operating voltages before the Power Down Mode is terminated.

For the 80C31 or 80C32, either a hardware reset or external interrupt can be used to exit from Power Down. Reset redefines all the SFRs but does not change the on-chip RAM. An external interrupt allows both the SFRs and the on-chip RAM to retain their values. WUPD (AUXR1.3–Wakeup from Power Down) enables or disables the wakeup from power down with external interrupt. Where:

WUPD = 0 Disable
WUPD = 1 Enable

To properly terminate Power Down the reset or external interrupt should not be executed before V_{CC} is restored to its normal operating level and must be held active long enough for the oscillator to restart and stabilize (normally less than 10 ms).

With an external interrupt, INT0 or INT1 must be enabled and configured as level-sensitive. Holding the pin low restarts the oscillator but bringing the pin back high completes the exit. Once the interrupt is serviced, the next instruction to be executed after RETI will be the one following the instruction that put the device into Power Down.

For the 80C31, wakeup from power down is always enabled.

Design Consideration

- When the idle mode is terminated by a hardware reset, the device normally resumes program execution, from where it left off, up to two machine cycles before the internal reset algorithm takes control. On-chip hardware inhibits access to internal RAM in this event, but access to the port pins is not inhibited. To eliminate the possibility of an unexpected write when Idle is terminated by reset, the instruction following the one that invokes Idle should not be one that writes to a port pin or to external memory.

ONCE™ Mode

The ONCE (“On-Circuit Emulation”) Mode facilitates testing and debugging of systems without the device having to be removed from the circuit. The ONCE Mode is invoked by:

1. Pull ALE low while the device is in reset and $\overline{\text{PSEN}}$ is high;
2. Hold ALE low as RST is deactivated.

While the device is in ONCE Mode, the Port 0 pins go into a float state, and the other port pins and ALE and $\overline{\text{PSEN}}$ are weakly pulled high. The oscillator circuit remains active. While the 80C31/32 is in this mode, an emulator or test CPU can be used to drive the circuit. Normal operation is restored when a normal reset is applied.

Table 2. External Pin Status During Idle and Power-Down Modes

MODE	PROGRAM MEMORY	ALE	$\overline{\text{PSEN}}$	PORT 0	PORT 1	PORT 2	PORT 3
Idle	Internal	1	1	Data	Data	Data	Data
Idle	External	1	1	Float	Data	Address	Data
Power-down	Internal	0	0	Data	Data	Data	Data
Power-down	External	0	0	Float	Data	Data	Data

Programmable Clock-Out

A 50% duty cycle clock can be programmed to come out on P1.0. This pin, besides being a regular I/O pin, has two alternate functions. It can be programmed:

1. to input the external clock for Timer/Counter 2, or
2. to output a 50% duty cycle clock ranging from 61 Hz to 4 MHz at a 16 MHz operating frequency.

To configure the Timer/Counter 2 as a clock generator, bit C/T2 (in T2CON) must be cleared and bit T20E in T2MOD must be set. Bit TR2 (T2CON.2) also must be set to start the timer.

The Clock-Out frequency depends on the oscillator frequency and the reload value of Timer 2 capture registers (RCAP2H, RCAP2L) as shown in this equation:

$$\frac{\text{Oscillator Frequency}}{4 \times (65536 - \text{RCAP2H, RCAP2L})}$$

Where:

(RCAP2H,RCAP2L) = the content of RCAP2H and RCAP2L taken as a 16-bit unsigned integer.

In the Clock-Out mode Timer 2 roll-overs will not generate an interrupt. This is similar to when it is used as a baud-rate generator. It is possible to use Timer 2 as a baud-rate generator and a clock generator simultaneously. Note, however, that the baud-rate and the Clock-Out frequency will be the same.

TIMER 2 OPERATION

Timer 2

Timer 2 is a 16-bit Timer/Counter which can operate as either an event timer or an event counter, as selected by C/$\overline{T}$2* in the special function register T2CON (see Figure 1). Timer 2 has three operating modes:Capture, Auto-reload (up or down counting) ,and Baud Rate Generator, which are selected by bits in the T2CON as shown in Table 3.

Capture Mode

In the capture mode there are two options which are selected by bit EXEN2 in T2CON. If EXEN2=0, then timer 2 is a 16-bit timer or counter (as selected by C/T2* in T2CON) which, upon overflowing sets bit TF2, the timer 2 overflow bit. This bit can be used to generate an interrupt (by enabling the Timer 2 interrupt bit in the IE register). If EXEN2= 1, Timer 2 operates as described above, but with the added feature that a 1- to -0 transition at external input T2EX causes the current value in the Timer 2 registers, TL2 and TH2, to be captured into registers RCAP2L and RCAP2H, respectively. In addition, the transition at T2EX causes bit EXF2 in T2CON to be set, and EXF2 like TF2 can generate an interrupt (which vectors to the same location as Timer 2 overflow interrupt. The Timer 2 interrupt service routine can interrogate TF2 and EXF2 to determine which event caused the interrupt). The capture mode is illustrated in Figure 2 (There is no reload value for TL2 and TH2 in this mode. Even when a capture event occurs from T2EX, the counter keeps on counting T2EX pin transitions or osc/12 pulses.).

Auto-Reload Mode (Up or Down Counter)

In the 16-bit auto-reload mode, Timer 2 can be configured (as either a timer or counter (C/T2* in T2CON)) then programmed to count up or down. The counting direction is determined by bit DCEN (Down Counter Enable) which is located in the T2MOD register (see Figure 3). When reset is applied the DCEN=0 which means Timer 2 will default to counting up. If DCEN bit is set, Timer 2 can count up or down depending on the value of the T2EX pin.

Figure 4 shows Timer 2 which will count up automatically since DCEN=0. In this mode there are two options selected by bit EXEN2 in T2CON register. If EXEN2=0, then Timer 2 counts up to 0FFFFH and sets the TF2 (Overflow Flag) bit upon overflow. This causes the Timer 2 registers to be reloaded with the 16-bit value in RCAP2L and RCAP2H. The values in RCAP2L and RCAP2H are preset by software means.

If EXEN2=1, then a 16-bit reload can be triggered either by an overflow or by a 1-to-0 transition at input T2EX. This transition also sets the EXF2 bit. The Timer 2 interrupt, if enabled, can be generated when either TF2 or EXF2 are 1.

In Figure 5 DCEN=1 which enables Timer 2 to count up or down. This mode allows pin T2EX to control the direction of count. When a logic 1 is applied at pin T2EX Timer 2 will count up. Timer 2 will overflow at 0FFFFH and set the TF2 flag, which can then generate an interrupt, if the interrupt is enabled. This timer overflow also causes the 16–bit value in RCAP2L and RCAP2H to be reloaded into the timer registers TL2 and TH2.

When a logic 0 is applied at pin T2EX this causes Timer 2 to count down. The timer will underflow when TL2 and TH2 become equal to the value stored in RCAP2L and RCAP2H. Timer 2 underflow sets the TF2 flag and causes 0FFFFH to be reloaded into the timer registers TL2 and TH2.

The external flag EXF2 toggles when Timer 2 underflows or overflows. This EXF2 bit can be used as a 17th bit of resolution if needed. The EXF2 flag does not generate an interrupt in this mode of operation.

Table 3. Timer 2 Operating Modes

RCLK + TCLK	CP/$\overline{RL2}$	TR2	MODE
0	0	1	16-bit Auto-reload
0	1	1	16-bit Capture
1	X	1	Baud rate generator
X	X	0	(off)

(MSB)							(LSB)
TF2	EXF2	RCLK	TCLK	EXEN2	TR2	C/$\overline{T2}$	CP/$\overline{RL2}$

Symbol	Position	Name and Significance
TF2	T2CON.7	Timer 2 overflow flag set by a Timer 2 overflow and must be cleared by software. TF2 will not be set when either RCLK or TCLK = 1.
EXF2	T2CON.6	Timer 2 external flag set when either a capture or reload is caused by a negative transition on T2EX and EXEN2 = 1. When Timer 2 interrupt is enabled, EXF2 = 1 will cause the CPU to vector to the Timer 2 interrupt routine. EXF2 must be cleared by software. EXF2 does not cause an interrupt in up/down counter mode (DCEN = 1).
RCLK	T2CON.5	Receive clock flag. When set, causes the serial port to use Timer 2 overflow pulses for its receive clock in modes 1 and 3. RCLK = 0 causes Timer 1 overflow to be used for the receive clock.
TCLK	T2CON.4	Transmit clock flag. When set, causes the serial port to use Timer 2 overflow pulses for its transmit clock in modes 1 and 3. TCLK = 0 causes Timer 1 overflows to be used for the transmit clock.
EXEN2	T2CON.3	Timer 2 external enable flag. When set, allows a capture or reload to occur as a result of a negative transition on T2EX if Timer 2 is not being used to clock the serial port. EXEN2 = 0 causes Timer 2 to ignore events at T2EX.
TR2	T2CON.2	Start/stop control for Timer 2. A logic 1 starts the timer.
C/$\overline{T2}$	T2CON.1	Timer or counter select. (Timer 2) 0 = Internal timer (OSC/12) 1 = External event counter (falling edge triggered).
CP/$\overline{RL2}$	T2CON.0	Capture/Reload flag. When set, captures will occur on negative transitions at T2EX if EXEN2 = 1. When cleared, auto-reloads will occur either with Timer 2 overflows or negative transitions at T2EX when EXEN2 = 1. When either RCLK = 1 or TCLK = 1, this bit is ignored and the timer is forced to auto-reload on Timer 2 overflow.

SU00728

Figure 1. Timer/Counter 2 (T2CON) Control Register

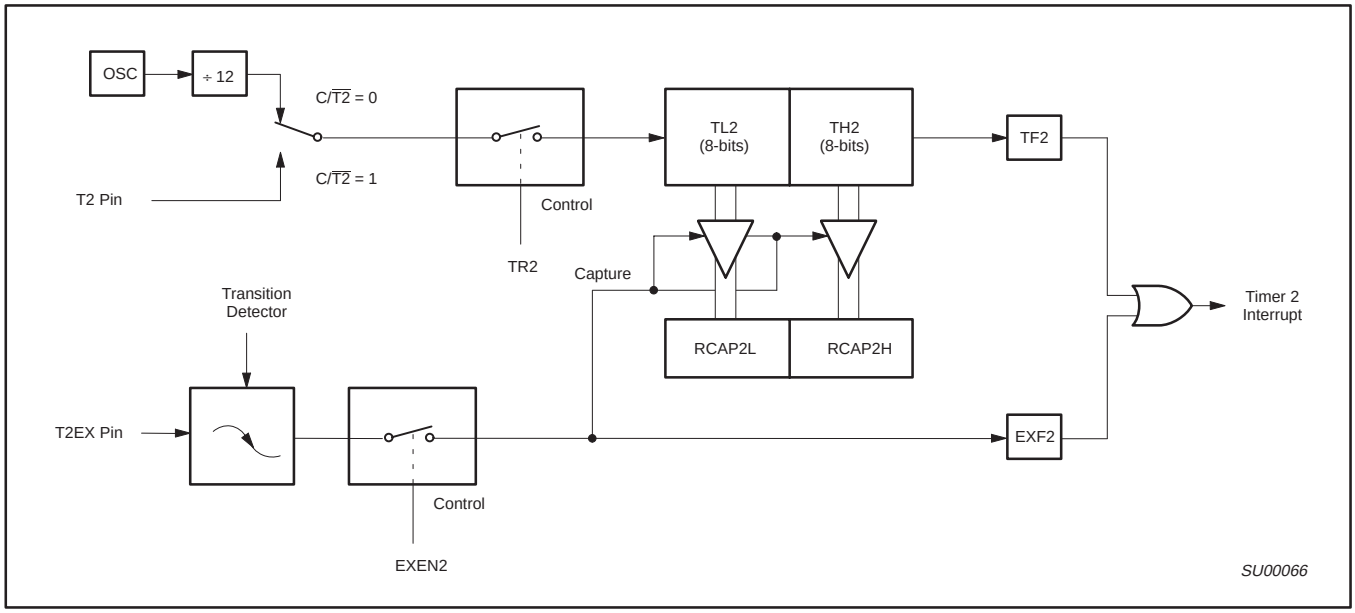


Figure 2. Timer 2 in Capture Mode

T2MOD Address = 0C9H Reset Value = XXXX XX00B

Not Bit Addressable

—	—	—	—	—	—	T2OE	DCEN
Bit 7	6	5	4	3	2	1	0

Symbol	Function
—	Not implemented, reserved for future use.*
T2OE	Timer 2 Output Enable bit.
DCEN	Down Count Enable bit. When set, this allows Timer 2 to be configured as an up/down counter.

* User software should not write 1s to reserved bits. These bits may be used in future 8051 family products to invoke new features. In that case, the reset or inactive value of the new bit will be 0, and its active value will be 1. The value read from a reserved bit is indeterminate.

SU00729

Figure 3. Timer 2 Mode (T2MOD) Control Register

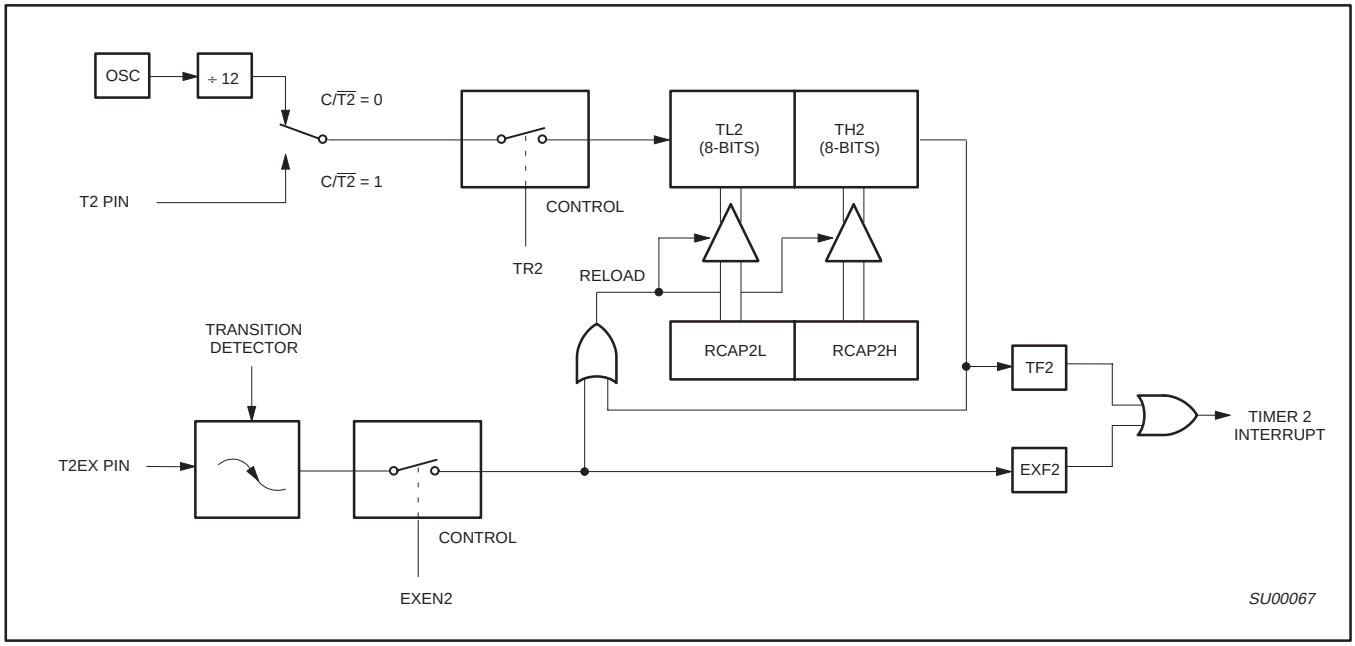


Figure 4. Timer 2 in Auto-Reload Mode (DCEN = 0)

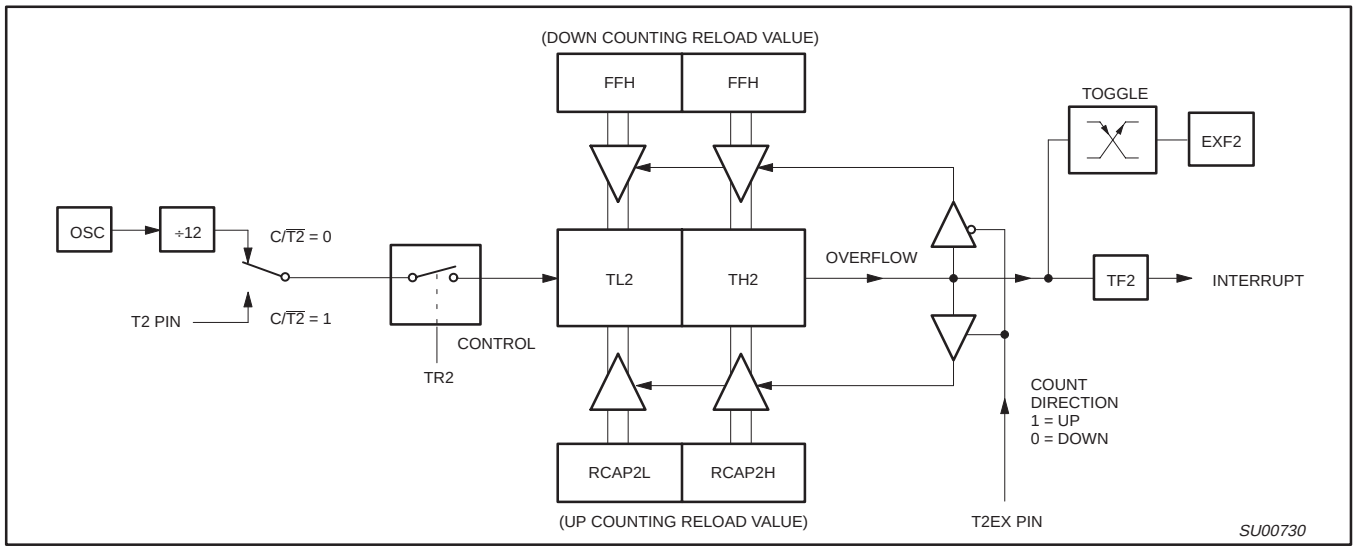


Figure 5. Timer 2 Auto Reload Mode (DCEN = 1)

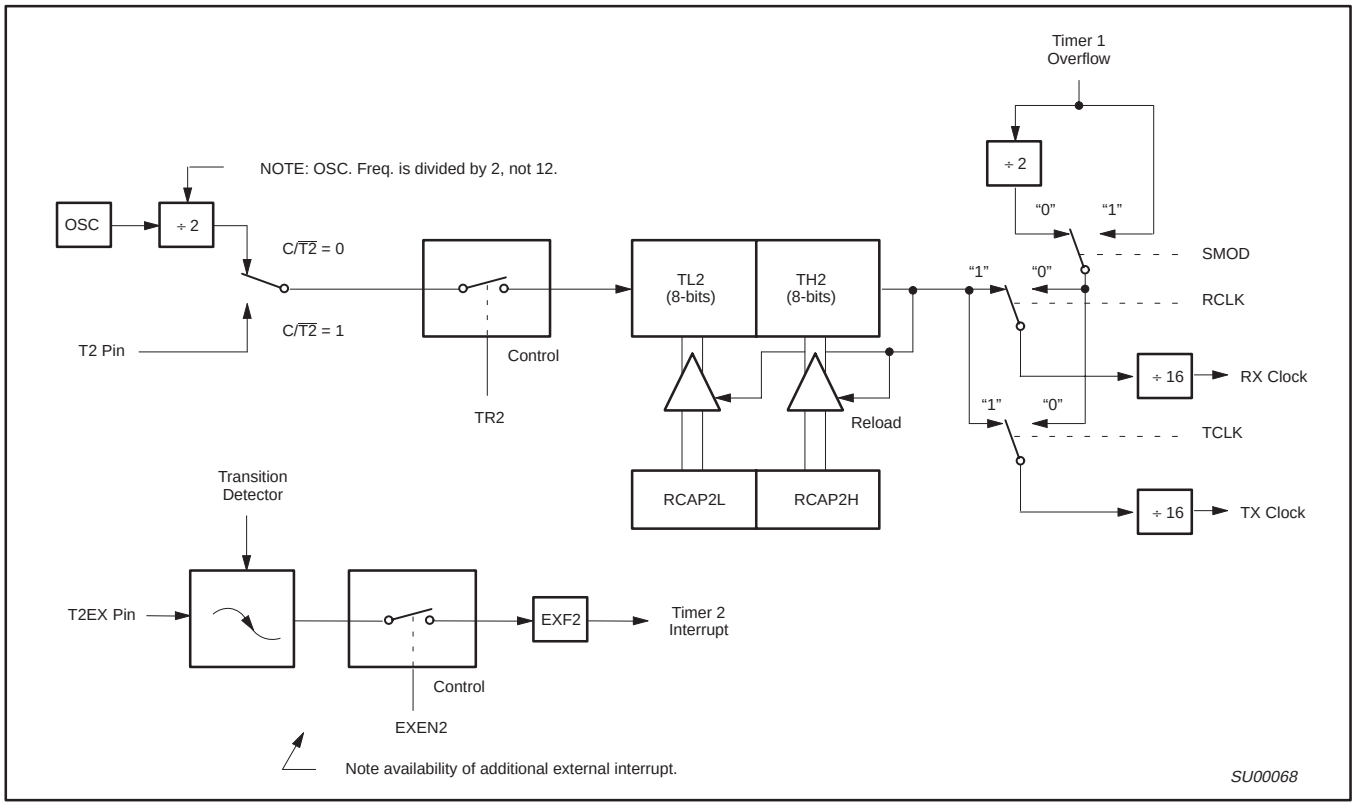


Figure 6. Timer 2 in Baud Rate Generator Mode

Baud Rate Generator Mode

Bits TCLK and/or RCLK in T2CON (Table 3) allow the serial port transmit and receive baud rates to be derived from either Timer 1 or Timer 2. When TCLK= 0, Timer 1 is used as the serial port transmit baud rate generator. When TCLK= 1, Timer 2 is used as the serial port transmit baud rate generator. RCLK has the same effect for the serial port receive baud rate. With these two bits, the serial port can have different receive and transmit baud rates – one generated by Timer 1, the other by Timer 2.

Figure 6 shows the Timer 2 in baud rate generation mode. The baud rate generation mode is like the auto-reload mode, in that a rollover in TH2 causes the Timer 2 registers to be reloaded with the 16-bit value in registers RCAP2H and RCAP2L, which are preset by software.

The baud rates in modes 1 and 3 are determined by Timer 2's overflow rate given below:

$$\text{Modes 1 and 3 Baud Rates} = \frac{\text{Timer 2 Overflow Rate}}{16}$$

The timer can be configured for either "timer" or "counter" operation. In many applications, it is configured for "timer" operation (C/$\overline{T}$2*=0). Timer operation is different for Timer 2 when it is being used as a baud rate generator.

Usually, as a timer it would increment every machine cycle (i.e., 1/12 the oscillator frequency). As a baud rate generator, it increments every state time (i.e., 1/2 the oscillator frequency). Thus the baud rate formula is as follows:

$$\text{Modes 1 and 3 Baud Rates} = \frac{\text{Oscillator Frequency}}{[32 \times [65536 - (\text{RCAP2H}, \text{RCAP2L})]]}$$

Where: (RCAP2H, RCAP2L)= The content of RCAP2H and RCAP2L taken as a 16-bit unsigned integer.

The Timer 2 as a baud rate generator mode shown in Figure 6, is valid only if RCLK and/or TCLK = 1 in T2CON register. Note that a rollover in TH2 does not set TF2, and will not generate an interrupt. Thus, the Timer 2 interrupt does not have to be disabled when Timer 2 is in the baud rate generator mode. Also if the EXEN2 (T2 external enable flag) is set, a 1-to-0 transition in T2EX (Timer/counter 2 trigger input) will set EXF2 (T2 external flag) but will not cause a reload from (RCAP2H, RCAP2L) to (TH2,TL2). Therefore when Timer 2 is in use as a baud rate generator, T2EX can be used as an additional external interrupt, if needed.

When Timer 2 is in the baud rate generator mode, one should not try to read or write TH2 and TL2. As a baud rate generator, Timer 2 is incremented every state time (osc/2) or asynchronously from pin T2; under these conditions, a read or write of TH2 or TL2 may not be accurate. The RCAP2 registers may be read, but should not be written to, because a write might overlap a reload and cause write and/or reload errors. The timer should be turned off (clear TR2) before accessing the Timer 2 or RCAP2 registers.

Table 4 shows commonly used baud rates and how they can be obtained from Timer 2.

Table 4. Timer 2 Generated Commonly Used Baud Rates

Baud Rate	Osc Freq	Timer 2	
		RCAP2H	RCAP2L
375 K	12 MHz	FF	FF
9.6 K	12 MHz	FF	D9
2.8 K	12 MHz	FF	B2
2.4 K	12 MHz	FF	64
1.2 K	12 MHz	FE	C8
300	12 MHz	FB	1E
110	12 MHz	F2	AF
300	6 MHz	FD	8F
110	6 MHz	F9	57

Summary Of Baud Rate Equations

Timer 2 is in baud rate generating mode. If Timer 2 is being clocked through pin T2(P1.0) the baud rate is:

$$\text{Baud Rate} = \frac{\text{Timer 2 Overflow Rate}}{16}$$

If Timer 2 is being clocked internally, the baud rate is:

$$\text{Baud Rate} = \frac{f_{OSC}}{[32 \times [65536 - (\text{RCAP2H}, \text{RCAP2L})]]}$$

Where f_{OSC}= Oscillator Frequency

To obtain the reload value for RCAP2H and RCAP2L, the above equation can be rewritten as:

$$\text{RCAP2H}, \text{RCAP2L} = 65536 - \left(\frac{f_{OSC}}{32 \times \text{Baud Rate}}\right)$$

Timer/Counter 2 Set-up

Except for the baud rate generator mode, the values given for T2CON do not include the setting of the TR2 bit. Therefore, bit TR2 must be set, separately, to turn the timer on. See Table 5 for set-up of Timer 2 as a timer. Also see Table 6 for set-up of Timer 2 as a counter.

Table 5. Timer 2 as a Timer

MODE	T2CON	
	INTERNAL CONTROL (Note 1)	EXTERNAL CONTROL (Note 2)
16-bit Auto-Reload	00H	08H
16-bit Capture	01H	09H
Baud rate generator receive and transmit same baud rate	34H	36H
Receive only	24H	26H
Transmit only	14H	16H

Table 6. Timer 2 as a Counter

MODE	TMOD	
	INTERNAL CONTROL (Note 1)	EXTERNAL CONTROL (Note 2)
16-bit	02H	0AH
Auto-Reload	03H	0BH

NOTES:
1. Capture/reload occurs only on timer/counter overflow.
2. Capture/reload occurs on timer/counter overflow and a 1-to-0 transition on T2EX (P1.1) pin except when Timer 2 is used in the baud rate generator mode.

Enhanced UART

The UART operates in all of the usual modes that are described in the first section of *Data Handbook IC20, 80C51-Based 8-Bit Microcontrollers*. In addition the UART can perform framing error detect by looking for missing stop bits, and automatic address recognition. The 80C31/32 UART also fully supports multiprocessor communication.

When used for framing error detect the UART looks for missing stop bits in the communication. A missing bit will set the FE bit in the SCON register. The FE bit shares the SCON.7 bit with SM0 and the function of SCON.7 is determined by PCON.6 (SMOD0) (see Figure 7). If SMOD0 is set then SCON.7 functions as FE. SCON.7 functions as SM0 when SMOD0 is cleared. When used as FE SCON.7 can only be cleared by software. Refer to Figure 8.

Automatic Address Recognition

Automatic Address Recognition is a feature which allows the UART to recognize certain addresses in the serial bit stream by using hardware to make the comparisons. This feature saves a great deal of software overhead by eliminating the need for the software to examine every serial address which passes by the serial port. This feature is enabled by setting the SM2 bit in SCON. In the 9 bit UART modes, mode 2 and mode 3, the Receive Interrupt flag (RI) will be automatically set when the received byte contains either the “Given” address or the “Broadcast” address. The 9 bit mode requires that the 9th information bit is a 1 to indicate that the received information is an address and not data. Automatic address recognition is shown in Figure 9.

The 8 bit mode is called Mode 1. In this mode the RI flag will be set if SM2 is enabled and the information received has a valid stop bit following the 8 address bits and the information is either a Given or Broadcast address.

Mode 0 is the Shift Register mode and SM2 is ignored.

Using the Automatic Address Recognition feature allows a master to selectively communicate with one or more slaves by invoking the Given slave address or addresses. All of the slaves may be contacted by using the Broadcast address. Two special Function Registers are used to define the slave’s address, SADDR, and the address mask, SADEN. SADEN is used to define which bits in the SADDR are to b used and which bits are “don’t care”. The SADEN mask can be logically ANDed with the SADDR to create the “Given” address which the master will use for addressing each of the slaves. Use of the Given address allows multiple slaves to be recognized while excluding others. The following examples will help to show the versatility of this scheme:

Slave 0	SADDR	=	1100 0000
	SADEN	=	1111 1101
	Given	=	1100 00X0
Slave 1	SADDR	=	1100 0000
	SADEN	=	1111 1110
	Given	=	1100 000X

In the above example SADDR is the same and the SADEN data is used to differentiate between the two slaves. Slave 0 requires a 0 in bit 0 and it ignores bit 1. Slave 1 requires a 0 in bit 1 and bit 0 is ignored. A unique address for Slave 0 would be 1100 0010 since slave 1 requires a 0 in bit 1. A unique address for slave 1 would be 1100 0001 since a 1 in bit 0 will exclude slave 0. Both slaves can be selected at the same time by an address which has bit 0 = 0 (for slave 0) and bit 1 = 0 (for slave 1). Thus, both could be addressed with 1100 0000.

In a more complex system the following could be used to select slaves 1 and 2 while excluding slave 0:

Slave 0	SADDR	=	1100 0000
	SADEN	=	1111 1001
	Given	=	1100 0XX0
Slave 1	SADDR	=	1110 0000
	SADEN	=	1111 1010
	Given	=	1110 0X0X
Slave 2	SADDR	=	1110 0000
	SADEN	=	1111 1100
	Given	=	1110 00XX

In the above example the differentiation among the 3 slaves is in the lower 3 address bits. Slave 0 requires that bit 0 = 0 and it can be uniquely addressed by 1110 0110. Slave 1 requires that bit 1 = 0 and it can be uniquely addressed by 1110 and 0101. Slave 2 requires that bit 2 = 0 and its unique address is 1110 0011. To select Slaves 0

and 1 and exclude Slave 2 use address 1110 0100, since it is necessary to make bit 2 = 1 to exclude slave 2.

The Broadcast Address for each slave is created by taking the logical OR of SADDR and SADEN. Zeros in this result are trended as don't-cares. In most cases, interpreting the don't-cares as ones, the broadcast address will be FF hexadecimal.

Upon reset SADDR (SFR address 0A9H) and SADEN (SFR address 0B9H) are leaded with 0s. This produces a given address of all "don't cares" as well as a Broadcast address of all "don't cares". This effectively disables the Automatic Addressing mode and allows the microcontroller to use standard 80C51 type UART drivers which do not make use of this feature.

SCON Address = 98H — Reset Value = 0000 0000B

Bit Addressable

	SM0/FE	SM1	SM2	REN	TB8	RB8	TI	RI
Bit:	7	6	5	4	3	2	1	0
	(SMOD0 = 0/1)*							

Symbol	Function
FE	Framing Error bit. This bit is set by the receiver when an invalid stop bit is detected. The FE bit is not cleared by valid frames but should be cleared by software. The SMOD0 bit must be set to enable access to the FE bit.
SM0	Serial Port Mode Bit 0, (SMOD0 must = 0 to access bit SM0)
SM1	Serial Port Mode Bit 1

SM0	SM1	Mode	Description	Baud Rate**
0	0	0	shift register	$f_{OSC}/12$
0	1	1	8-bit UART	variable
1	0	2	9-bit UART	$f_{OSC}/64$ or $f_{OSC}/32$
1	1	3	9-bit UART	variable

SM2	Enables the Automatic Address Recognition feature in Modes 2 or 3. If SM2 = 1 then RI will not be set unless the received 9th data bit (RB8) is 1, indicating an address, and the received byte is a Given or Broadcast Address. In Mode 1, if SM2 = 1 then RI will not be activated unless a valid stop bit was received, and the received byte is a Given or Broadcast Address. In Mode 0, SM2 should be 0.
REN	Enables serial reception. Set by software to enable reception. Clear by software to disable reception.
TB8	The 9th data bit that will be transmitted in Modes 2 and 3. Set or clear by software as desired.
RB8	In modes 2 and 3, the 9th data bit that was received. In Mode 1, if SM2 = 0, RB8 is the stop bit that was received. In Mode 0, RB8 is not used.
TI	Transmit interrupt flag. Set by hardware at the end of the 8th bit time in Mode 0, or at the beginning of the stop bit in the other modes, in any serial transmission. Must be cleared by software.
RI	Receive interrupt flag. Set by hardware at the end of the 8th bit time in Mode 0, or halfway through the stop bit time in the other modes, in any serial reception (except see SM2). Must be cleared by software.

NOTE:
*SMOD0 is located at PCON6.
**f_{OSC} = oscillator frequency

SU00043

Figure 7. SCON: Serial Port Control Register

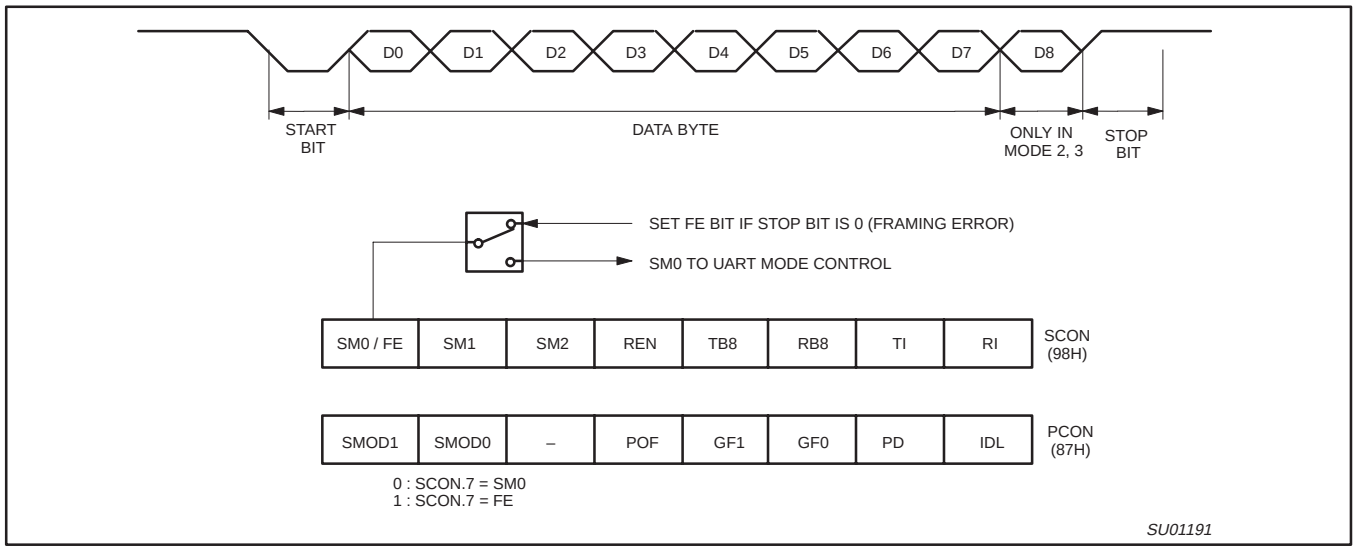


Figure 8. UART Framing Error Detection

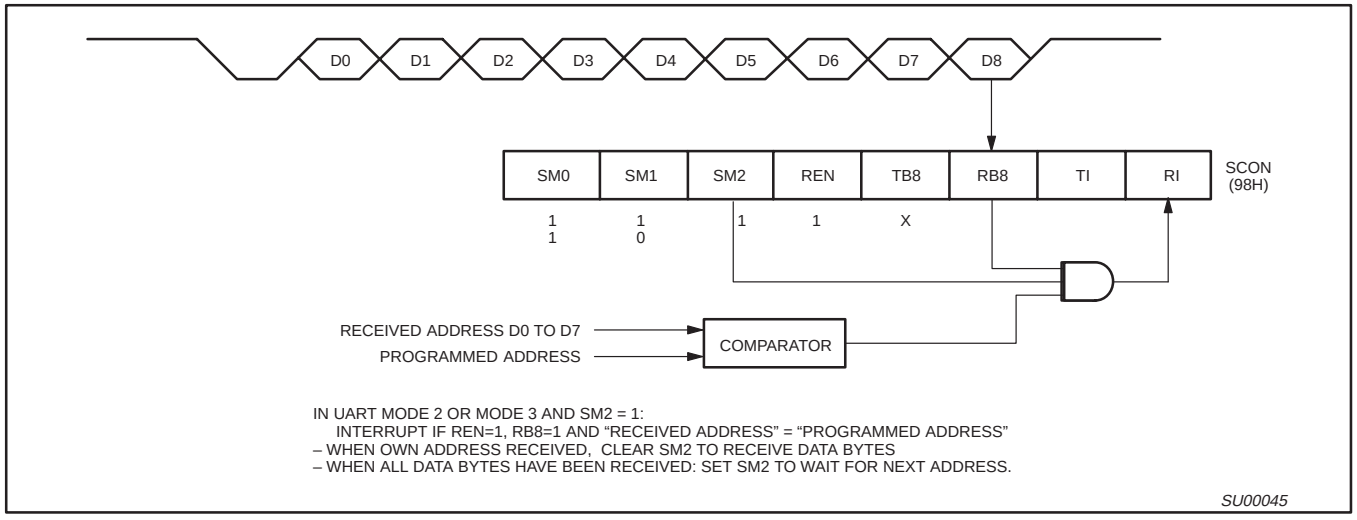


Figure 9. UART Multiprocessor Communication, Automatic Address Recognition

Interrupt Priority Structure

The 80C31 and 80C32 have a 6-source four-level interrupt structure. They are the IE, IP and IPH. (See Figures 10, 11, and 12.) The IPH (Interrupt Priority High) register that makes the four-level interrupt structure possible. The IPH is located at SFR address B7H. The structure of the IPH register and a description of its bits is shown in Figure 12.

The function of the IPH SFR is simple and when combined with the IP SFR determines the priority of each interrupt. The priority of each interrupt is determined as shown in the following table:

PRIORITY BITS		INTERRUPT PRIORITY LEVEL
IPH.x	IP.x	
0	0	Level 0 (lowest priority)
0	1	Level 1
1	0	Level 2
1	1	Level 3 (highest priority)

An interrupt will be serviced as long as an interrupt of equal or higher priority is not already being serviced. If an interrupt of equal or higher level priority is being serviced, the new interrupt will wait until it is finished before being serviced. If a lower priority level interrupt is being serviced, it will be stopped and the new interrupt serviced. When the new interrupt is finished, the lower priority level interrupt that was stopped will be completed.

Table 7. Interrupt Table

SOURCE	POLLING PRIORITY	REQUEST BITS	HARDWARE CLEAR?	VECTOR ADDRESS
X0	1	IE0	N (L)[1] Y (T)[2]	03H
T0	2	TP0	Y	0BH
X1	3	IE1	N (L) Y (T)	13H
T1	4	TF1	Y	1BH
SP	5	RI, TI	N	23H
T2	6	TF2, EXF2	N	2BH

NOTES:
1. L = Level activated
2. T = Transition activated

	7	6	5	4	3	2	1	0
IE (0A8H)	EA	—	ET2	ES	ET1	EX1	ET0	EX0

Enable Bit = 1 enables the interrupt.
Enable Bit = 0 disables it.

BIT	SYMBOL	FUNCTION
IE.7	EA	Global disable bit. If EA = 0, all interrupts are disabled. If EA = 1, each interrupt can be individually enabled or disabled by setting or clearing its enable bit.
IE.6	—	Not implemented. Reserved for future use.
IE.5	ET2	Timer 2 interrupt enable bit.
IE.4	ES	Serial Port interrupt enable bit.
IE.3	ET1	Timer 1 interrupt enable bit.
IE.2	EX1	External interrupt 1 enable bit.
IE.1	ET0	Timer 0 interrupt enable bit.
IE.0	EX0	External interrupt 0 enable bit.

SU00571

Figure 10. IE Registers

	7	6	5	4	3	2	1	0
IP (0B8H)	—	—	PT2	PS	PT1	PX1	PT0	PX0

Priority Bit = 1 assigns higher priority
Priority Bit = 0 assigns lower priority

BIT	SYMBOL	FUNCTION
IP.7	—	Not implemented, reserved for future use.
IP.6	—	Not implemented, reserved for future use.
IP.5	PT2	Timer 2 interrupt priority bit.
IP.4	PS	Serial Port interrupt priority bit.
IP.3	PT1	Timer 1 interrupt priority bit.
IP.2	PX1	External interrupt 1 priority bit.
IP.1	PT0	Timer 0 interrupt priority bit.
IP.0	PX0	External interrupt 0 priority bit.

SU00572

Figure 11. IP Registers

	7	6	5	4	3	2	1	0
IPH (B7H)	—	—	PT2H	PSH	PT1H	PX1H	PT0H	PX0H

Priority Bit = 1 assigns higher priority
Priority Bit = 0 assigns lower priority

BIT	SYMBOL	FUNCTION
IPH.7	—	Not implemented, reserved for future use.
IPH.6	—	Not implemented, reserved for future use.
IPH.5	PT2H	Timer 2 interrupt priority bit high.
IPH.4	PSH	Serial Port interrupt priority bit high.
IPH.3	PT1H	Timer 1 interrupt priority bit high.
IPH.2	PX1H	External interrupt 1 priority bit high.
IPH.1	PT0H	Timer 0 interrupt priority bit high.
IPH.0	PX0H	External interrupt 0 priority bit high.

SU01058

Figure 12. IPH Registers

Reduced EMI Mode

The AO bit (AUXR.0) in the AUXR register when set disables the ALE output.

Reduced EMI Mode

AUXR (8EH)

7	6	5	4	3	2	1	0
–	–	–	–	–	–	–	AO

AUXR.0 AO Turns off ALE output.

Dual DPTR

The dual DPTR structure (see Figure 13) enables a way to specify the address of an external data memory location. There are two 16-bit DPTR registers that address the external memory, and a single bit called DPS = AUXR1/bit0 that allows the program code to switch between them.

- New Register Name: AUXR1#
- SFR Address: A2H
- Reset Value: xxx000x0B

AUXR1 (A2H)

7	6	5	4	3	2	1	0
–	–	–	–	WUPD	0	–	DPS

Where:
DPS = AUXR1/bit0 = Switches between DPTR0 and DPTR1.

Select Reg	DPS
DPTR0	0
DPTR1	1

The DPS bit status should be saved by software when switching between DPTR0 and DPTR1.

Note that bit 2 is not writable and is always read as a zero. This allows the DPS bit to be quickly toggled simply by executing an INC DPTR instruction without affecting the WOPD or LPEP bits.

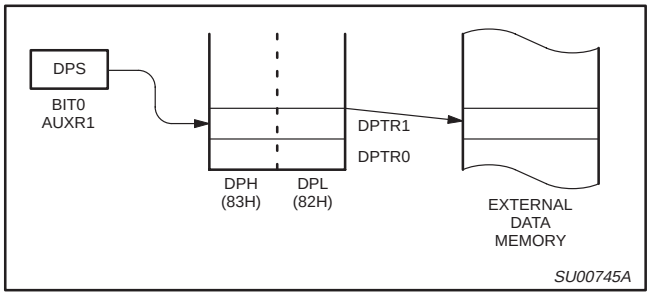


Figure 13.

DPTR Instructions

The instructions that refer to DPTR refer to the data pointer that is currently selected using the AUXR1/bit 0 register. The six instructions that use the DPTR are as follows:

INC DPTR	Increments the data pointer by 1
MOV DPTR, #data16	Loads the DPTR with a 16-bit constant
MOV A, @ A+DPTR	Move code byte relative to DPTR to ACC
MOVX A, @ DPTR	Move external RAM (16-bit address) to ACC
MOVX @ DPTR , A	Move ACC to external RAM (16-bit address)
JMP @ A + DPTR	Jump indirect relative to DPTR

The data pointer can be accessed on a byte-by-byte basis by specifying the low or high byte in an instruction which accesses the SFRs. See application note AN458 for more details.

ABSOLUTE MAXIMUM RATINGS[1, 2, 3]

PARAMETER	RATING	UNIT
Operating temperature under bias	0 to +70 or –40 to +85	°C
Storage temperature range	–65 to +150	°C
Voltage on $\overline{EA}$ pin to V_{SS}	0 to +13.0	V
Voltage on any other pin to V_{SS}	–0.5 to +6.5	V
Maximum I_{OL} per I/O pin	15	mA
Power dissipation (based on package heat transfer limitations, not device power consumption)	1.5	W

NOTES:
1. Stresses above those listed under Absolute Maximum Ratings may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or any conditions other than those described in the AC and DC Electrical Characteristics section of this specification is not implied.
2. This product includes circuitry specifically designed for the protection of its internal devices from the damaging effects of excessive static charge. Nonetheless, it is suggested that conventional precautions be taken to avoid applying greater than the rated maximum.
3. Parameters are valid over operating temperature range unless otherwise specified. All voltages are with respect to V_{SS} unless otherwise noted.

AC ELECTRICAL CHARACTERISTICS

T_{amb} = 0°C to +70°C or –40°C to +85°C

SYMBOL	FIGURE	PARAMETER	CLOCK FREQUENCY RANGE –f		UNIT
			MIN	MAX	
$1/t_{CLCL}$	29	Oscillator frequency			
		Speed versions : S (16 MHz)	0	16	MHz
		U (33 MHz)	0	33	MHz

DC ELECTRICAL CHARACTERISTICS

T_{amb} = 0°C to +70°C or –40°C to +85°C, V_{CC} = 2.7 V to 5.5 V, V_{SS} = 0 V (16 MHz devices)

SYMBOL	PARAMETER	TEST CONDITIONS	LIMITS			UNIT
			MIN	TYP[1]	MAX	
V_{IL}	Input low voltage	4.0 V < V_{CC} < 5.5 V	–0.5		0.2 V_{CC}–0.1	V
		2.7 V<V_{CC}< 4.0 V	–0.5		0.7	V
V_{IH}	Input high voltage (ports 0, 1, 2, 3, $\overline{EA}$)		0.2 V_{CC}+0.9		V_{CC}+0.5	V
V_{IH1}	Input high voltage, XTAL1, RST		0.7 V_{CC}		V_{CC}+0.5	V
V_{OL}	Output low voltage, ports 1, 2, [8]	V_{CC} = 2.7 V I_{OL} = 1.6 mA[2]			0.4	V
V_{OL1}	Output low voltage, port 0, ALE, $\overline{PSEN}$[8, 7]	V_{CC} = 2.7 V I_{OL} = 3.2 mA[2]			0.4	V
V_{OH}	Output high voltage, ports 1, 2, 3 [3]	V_{CC} = 2.7 V I_{OH} = –20 µA	V_{CC} – 0.7			V
		V_{CC} = 4.5 V I_{OH} = –30 µA	V_{CC} – 0.7			V
V_{OH1}	Output high voltage (port 0 in external bus mode), ALE[9], $\overline{PSEN}$[3]	V_{CC} = 2.7 V I_{OH} = –3.2 mA	V_{CC} – 0.7			V
I_{IL}	Logical 0 input current, ports 1, 2, 3	V_{IN} = 0.4 V	–1		–50	µA
I_{TL}	Logical 1-to-0 transition current, ports 1, 2, 3[6]	V_{IN} = 2.0 V See note 4			–650	µA
I_{LI}	Input leakage current, port 0	0.45 < V_{IN} < V_{CC} – 0.3			±10	µA
I_{CC}	Power supply current (see Figure 21): Active mode @ 16 MHz Idle mode @ 16 MHz Power-down mode or clock stopped (see Figure 25 for conditions)	See note 5 T_{amb} = 0°C to 70°C T_{amb} = –40°C to +85°C		 3	 50 75	 µA µA µA µA
R_{RST}	Internal reset pull-down resistor		40		225	kΩ
C_{IO}	Pin capacitance[10] (except $\overline{EA}$)				15	pF

NOTES:

1. Typical ratings are not guaranteed. The values listed are at room temperature, 5 V.
2. Capacitive loading on ports 0 and 2 may cause spurious noise to be superimposed on the V_{OL}s of ALE and ports 1 and 3. The noise is due to external bus capacitance discharging into the port 0 and port 2 pins when these pins make 1-to-0 transitions during bus operations. In the worst cases (capacitive loading > 100 pF), the noise pulse on the ALE pin may exceed 0.8 V. In such cases, it may be desirable to qualify ALE with a Schmitt Trigger, or use an address latch with a Schmitt Trigger STROBE input. I_{OL} can exceed these conditions provided that no single output sinks more than 5 mA and no more than two outputs exceed the test conditions.
3. Capacitive loading on ports 0 and 2 may cause the V_{OH} on ALE and $\overline{PSEN}$ to momentarily fall below the V_{CC}–0.7 specification when the address bits are stabilizing.
4. Pins of ports 1, 2 and 3 source a transition current when they are being externally driven from 1 to 0. The transition current reaches its maximum value when V_{IN} is approximately 2 V.
5. See Figures 22 through 25 for I_{CC} test conditions.
   Active mode: I_{CC} = 0.9 × FREQ. + 1.1 mA
   Idle mode: I_{CC} = 0.18 × FREQ. +1.01 mA; See Figure 21.
6. This value applies to T_{amb} = 0°C to +70°C. For T_{amb} = –40°C to +85°C, I_{TL} = –750 µA.
7. Load capacitance for port 0, ALE, and $\overline{PSEN}$ = 100 pF, load capacitance for all other outputs = 80 pF.
8. Under steady state (non-transient) conditions, I_{OL} must be externally limited as follows:
   Maximum I_{OL} per port pin: 15 mA (*NOTE: This is 85°C specification.)
   Maximum I_{OL} per 8-bit port: 26 mA
   Maximum total I_{OL} for all outputs: 71 mA
   If I_{OL} exceeds the test condition, V_{OL} may exceed the related specification. Pins are not guaranteed to sink current greater than the listed test conditions.
9. ALE is tested to V_{OH1}, except when ALE is off then V_{OH} is the voltage specification.
10. Pin capacitance is characterized but not tested. Pin capacitance is less than 25 pF.

DC ELECTRICAL CHARACTERISTICS

T_{amb} = 0°C to +70°C or –40°C to +85°C, 33 MHz devices; 5 V ±10%; V_{SS} = 0 V

SYMBOL	PARAMETER	TEST CONDITIONS	LIMITS			UNIT
			MIN	TYP[1]	MAX	
V_{IL}	Input low voltage	4.5 V < V_{CC} < 5.5 V	–0.5		0.2 V_{CC}–0.1	V
V_{IH}	Input high voltage (ports 0, 1, 2, 3, $\overline{EA}$)		0.2 V_{CC}+0.9		V_{CC}+0.5	V
V_{IH1}	Input high voltage, XTAL1, RST		0.7 V_{CC}		V_{CC}+0.5	V
V_{OL}	Output low voltage, ports 1, 2, 3 [8]	V_{CC} = 4.5 V I_{OL} = 1.6mA[2]			0.4	V
V_{OL1}	Output low voltage, port 0, ALE, $\overline{PSEN}$ [7, 8]	V_{CC} = 4.5 V I_{OL} = 3.2mA[2]			0.4	V
V_{OH}	Output high voltage, ports 1, 2, 3 [3]	V_{CC} = 4.5 V I_{OH} = –30µA	V_{CC} – 0.7			V
V_{OH1}	Output high voltage (port 0 in external bus mode), ALE[9], $\overline{PSEN}$[3]	V_{CC} = 4.5 V I_{OH} = –3.2mA	V_{CC} – 0.7			V
I_{IL}	Logical 0 input current, ports 1, 2, 3	V_{IN} = 0.4 V	–1		–50	µA
I_{TL}	Logical 1-to-0 transition current, ports 1, 2, 3[6]	V_{IN} = 2.0 V See note 4			–650	µA
I_{LI}	Input leakage current, port 0	0.45 < V_{IN} < V_{CC} – 0.3			±10	µA
I_{CC}	Power supply current (see Figure 21): Active mode (see Note 5) Idle mode (see Note 5) Power-down mode or clock stopped (see Figure 25 for conditions)	See note 5 T_{amb} = 0°C to 70°C T_{amb} = –40°C to +85°C		 3	 50 75	 µA µA
R_{RST}	Internal reset pull-down resistor		40		225	kΩ
C_{IO}	Pin capacitance[10] (except $\overline{EA}$)				15	pF

NOTES:

1. Typical ratings are not guaranteed. The values listed are at room temperature, 5 V.
2. Capacitive loading on ports 0 and 2 may cause spurious noise to be superimposed on the V_{OL}s of ALE and ports 1 and 3. The noise is due to external bus capacitance discharging into the port 0 and port 2 pins when these pins make 1-to-0 transitions during bus operations. In the worst cases (capacitive loading > 100 pF), the noise pulse on the ALE pin may exceed 0.8 V. In such cases, it may be desirable to qualify ALE with a Schmitt Trigger, or use an address latch with a Schmitt Trigger STROBE input. I_{OL} can exceed these conditions provided that no single output sinks more than 5mA and no more than two outputs exceed the test conditions.
3. Capacitive loading on ports 0 and 2 may cause the V_{OH} on ALE and $\overline{PSEN}$ to momentarily fall below the V_{CC}–0.7 specification when the address bits are stabilizing.
4. Pins of ports 1, 2 and 3 source a transition current when they are being externally driven from 1 to 0. The transition current reaches its maximum value when V_{IN} is approximately 2 V.
5. See Figures 22 through 25 for I_{CC} test conditions.
   Active mode: $I_{CC(MAX)}$ = 0.9 × FREQ. + 1.1 mA
   Idle mode: $I_{CC(MAX)}$ = 0.18 × FREQ. +1.0 mA; See Figure 21.
6. This value applies to T_{amb} = 0°C to +70°C. For T_{amb} = –40°C to +85°C, I_{TL} = –750 µA.
7. Load capacitance for port 0, ALE, and $\overline{PSEN}$ = 100 pF, load capacitance for all other outputs = 80 pF.
8. Under steady state (non-transient) conditions, I_{OL} must be externally limited as follows:
   Maximum I_{OL} per port pin: 15 mA (*NOTE: This is 85°C specification.)
   Maximum I_{OL} per 8-bit port: 26 mA
   Maximum total I_{OL} for all outputs: 71 mA
   If I_{OL} exceeds the test condition, V_{OL} may exceed the related specification. Pins are not guaranteed to sink current greater than the listed test conditions.
9. ALE is tested to V_{OH1}, except when ALE is off then V_{OH} is the voltage specification.
10. Pin capacitance is characterized but not tested. Pin capacitance is less than 25 pF. Pin capacitance of ceramic package is less than 15 pF (except $\overline{EA}$ is 25 pF).

AC ELECTRICAL CHARACTERISTICS

T_{amb} = 0°C to +70°C or –40°C to +85°C, V_{CC} = +2.7 V to +5.5 V, V_{SS} = 0 V[1, 2, 3]

SYMBOL	FIGURE	PARAMETER	16 MHz CLOCK		VARIABLE CLOCK		UNIT
			MIN	MAX	MIN	MAX	
$1/t_{CLCL}$	14	Oscillator frequency[5] Speed versions :S			3.5	16	MHz
t_{LHLL}	14	ALE pulse width	85		$2t_{CLCL}$–40		ns
t_{AVLL}	14	Address valid to ALE low	22		t_{CLCL}–40		ns
t_{LLAX}	14	Address hold after ALE low	32		t_{CLCL}–30		ns
t_{LLIV}	14	ALE low to valid instruction in		150		$4t_{CLCL}$–100	ns
t_{LLPL}	14	ALE low to $\overline{PSEN}$ low	32		t_{CLCL}–30		ns
t_{PLPH}	14	$\overline{PSEN}$ pulse width	142		$3t_{CLCL}$–45		ns
t_{PLIV}	14	$\overline{PSEN}$ low to valid instruction in		82		$3t_{CLCL}$–105	ns
t_{PXIX}	14	Input instruction hold after $\overline{PSEN}$	0		0		ns
t_{PXIZ}	14	Input instruction float after $\overline{PSEN}$		37		t_{CLCL}–25	ns
t_{AVIV} [4]	14	Address to valid instruction in		207		$5t_{CLCL}$–105	ns
t_{PLAZ}	14	$\overline{PSEN}$ low to address float		10		10	ns
Data Memory							
t_{RLRH}	15, 16	$\overline{RD}$ pulse width	275		$6t_{CLCL}$–100		ns
t_{WLWH}	15, 16	$\overline{WR}$ pulse width	275		$6t_{CLCL}$–100		ns
t_{RLDV}	15, 16	$\overline{RD}$ low to valid data in		147		$5t_{CLCL}$–165	ns
t_{RHDX}	15, 16	Data hold after $\overline{RD}$	0		0		ns
t_{RHDZ}	15, 16	Data float after $\overline{RD}$		65		$2t_{CLCL}$–60	ns
t_{LLDV}	15, 16	ALE low to valid data in		350		$8t_{CLCL}$–150	ns
t_{AVDV}	15, 16	Address to valid data in		397		$9t_{CLCL}$–165	ns
t_{LLWL}	15, 16	ALE low to $\overline{RD}$ or $\overline{WR}$ low	137	239	$3t_{CLCL}$–50	$3t_{CLCL}$+50	ns
t_{AVWL}	15, 16	Address valid to $\overline{WR}$ low or $\overline{RD}$ low	122		$4t_{CLCL}$–130		ns
t_{QVWX}	15, 16	Data valid to $\overline{WR}$ transition	13		t_{CLCL}–50		ns
t_{WHQX}	15, 16	Data hold after $\overline{WR}$	13		t_{CLCL}–50		ns
t_{QVWH}	16	Data valid to $\overline{WR}$ high	287		$7t_{CLCL}$–150		ns
t_{RLAZ}	15, 16	$\overline{RD}$ low to address float		0		0	ns
t_{WHLH}	15, 16	$\overline{RD}$ or $\overline{WR}$ high to ALE high	23	103	t_{CLCL}–40	t_{CLCL}+40	ns
External Clock							
t_{CHCX}	18	High time	20		20	t_{CLCL}–t_{CLCX}	ns
t_{CLCX}	18	Low time	20		20	t_{CLCL}–t_{CHCX}	ns
t_{CLCH}	18	Rise time		20		20	ns
t_{CHCL}	18	Fall time		20		20	ns
Shift Register							
t_{XLXL}	17	Serial port clock cycle time	750		$12t_{CLCL}$		ns
t_{QVXH}	17	Output data setup to clock rising edge	492		$10t_{CLCL}$–133		ns
t_{XHQX}	17	Output data hold after clock rising edge	8		$2t_{CLCL}$–117		ns
t_{XHDX}	17	Input data hold after clock rising edge	0		0		ns
t_{XHDV}	17	Clock rising edge to input data valid		492		$10t_{CLCL}$–133	ns

NOTES:
1. Parameters are valid over operating temperature range unless otherwise specified.
2. Load capacitance for port 0, ALE, and $\overline{PSEN}$ = 100 pF, load capacitance for all other outputs = 80 pF.
3. Interfacing the 80C31 and 80C32 to devices with float times up to 45ns is permitted. This limited bus contention will not cause damage to Port 0 drivers.
4. See application note AN457 for external memory interface.
5. Parts are guaranteed to operate down to 0 Hz. When an external clock source is used, the RST pin should be held high for a minimum of 20 μs for power-on or wakeup from power down.

AC ELECTRICAL CHARACTERISTICS

T_{amb} = 0°C to +70°C or –40°C to +85°C, V_{CC} = 5 V ±10%, V_{SS} = 0 V[1, 2, 3]

SYMBOL	FIGURE	PARAMETER	VARIABLE CLOCK[4] 16 MHz to f_{max}		33 MHz CLOCK		UNIT
			MIN	MAX	MIN	MAX	
t_{LHLL}	14	ALE pulse width	$2t_{CLCL}$–40		21		ns
t_{AVLL}	14	Address valid to ALE low	t_{CLCL}–25		5		ns
t_{LLAX}	14	Address hold after ALE low	t_{CLCL}–25				ns
t_{LLIV}	14	ALE low to valid instruction in		$4t_{CLCL}$–65		55	ns
t_{LLPL}	14	ALE low to $\overline{PSEN}$ low	t_{CLCL}–25		5		ns
t_{PLPH}	14	$\overline{PSEN}$ pulse width	$3t_{CLCL}$–45		45		ns
t_{PLIV}	14	$\overline{PSEN}$ low to valid instruction in		$3t_{CLCL}$–60		30	ns
t_{PXIX}	14	Input instruction hold after $\overline{PSEN}$	0		0		ns
t_{PXIZ}	14	Input instruction float after $\overline{PSEN}$		t_{CLCL}–25		5	ns
t_{AVIV}	14	Address to valid instruction in		$5t_{CLCL}$–80		70	ns
t_{PLAZ}	14	$\overline{PSEN}$ low to address float		10		10	ns
Data Memory							
t_{RLRH}	15, 16	$\overline{RD}$ pulse width	$6t_{CLCL}$–100		82		ns
t_{WLWH}	15, 16	$\overline{WR}$ pulse width	$6t_{CLCL}$–100		82		ns
t_{RLDV}	15, 16	$\overline{RD}$ low to valid data in		$5t_{CLCL}$–90		60	ns
t_{RHDX}	15, 16	Data hold after $\overline{RD}$	0		0		ns
t_{RHDZ}	15, 16	Data float after $\overline{RD}$		$2t_{CLCL}$–28		32	ns
t_{LLDV}	15, 16	ALE low to valid data in		$8t_{CLCL}$–150		90	ns
t_{AVDV}	15, 16	Address to valid data in		$9t_{CLCL}$–165		105	ns
t_{LLWL}	15, 16	ALE low to $\overline{RD}$ or $\overline{WR}$ low	$3t_{CLCL}$–50	$3t_{CLCL}$+50	40	140	ns
t_{AVWL}	15, 16	Address valid to $\overline{WR}$ low or $\overline{RD}$ low	$4t_{CLCL}$–75		45		ns
t_{QVWX}	15, 16	Data valid to $\overline{WR}$ transition	t_{CLCL}–30		0		ns
t_{WHQX}	15, 16	Data hold after $\overline{WR}$	t_{CLCL}–25		5		ns
t_{QVWH}	16	Data valid to $\overline{WR}$ high	$7t_{CLCL}$–130		80		ns
t_{RLAZ}	15, 16	$\overline{RD}$ low to address float		0		0	ns
t_{WHLH}	15, 16	$\overline{RD}$ or $\overline{WR}$ high to ALE high	t_{CLCL}–25	t_{CLCL}+25	5	55	ns
External Clock							
t_{CHCX}	18	High time	$0.38t_{CLCL}$	t_{CLCL}–t_{CLCX}			ns
t_{CLCX}	18	Low time	$0.38t_{CLCL}$	t_{CLCL}–t_{CHCX}			ns
t_{CLCH}	18	Rise time		5			ns
t_{CHCL}	18	Fall time		5			ns
Shift Register							
t_{XLXL}	17	Serial port clock cycle time	$12t_{CLCL}$		360		ns
t_{QVXH}	17	Output data setup to clock rising edge	$10t_{CLCL}$–133		167		ns
t_{XHQX}	17	Output data hold after clock rising edge	$2t_{CLCL}$–80				ns
t_{XHDX}	17	Input data hold after clock rising edge	0		0		ns
t_{XHDV}	17	Clock rising edge to input data valid		$10t_{CLCL}$–133		167	ns

NOTES:
1. Parameters are valid over operating temperature range unless otherwise specified.
2. Load capacitance for port 0, ALE, and $\overline{PSEN}$ = 100 pF, load capacitance for all other outputs = 80 pF.
3. Interfacing the 80C31 and 80C32 to devices with float times up to 45ns is permitted. This limited bus contention will not cause damage to Port 0 drivers.
4. Variable clock is specified for oscillator frequencies greater than 16 MHz to 33 MHz. For frequencies equal or less than 16 MHz, see 16 MHz "AC Electrical Characteristics", page 23.
5. Parts are guaranteed to operate down to 0 Hz. When an external clock source is used, the RST pin should be held high for a minimum of 20 μs for power-on or wakeup from power down.

EXPLANATION OF THE AC SYMBOLS

Each timing symbol has five characters. The first character is always 't' (= time). The other characters, depending on their positions, indicate the name of a signal or the logical status of that signal. The designations are:

A – Address
C – Clock
D – Input data
H – Logic level high
I – Instruction (program memory contents)
L – Logic level low, or ALE
P – $\overline{\text{PSEN}}$
Q – Output data
R – $\overline{\text{RD}}$ signal
t – Time
V – Valid
W – $\overline{\text{WR}}$ signal
X – No longer a valid logic level
Z – Float

Examples: t_{AVLL} = Time for address valid to ALE low.
t_{LLPL} =Time for ALE low to $\overline{\text{PSEN}}$ low.

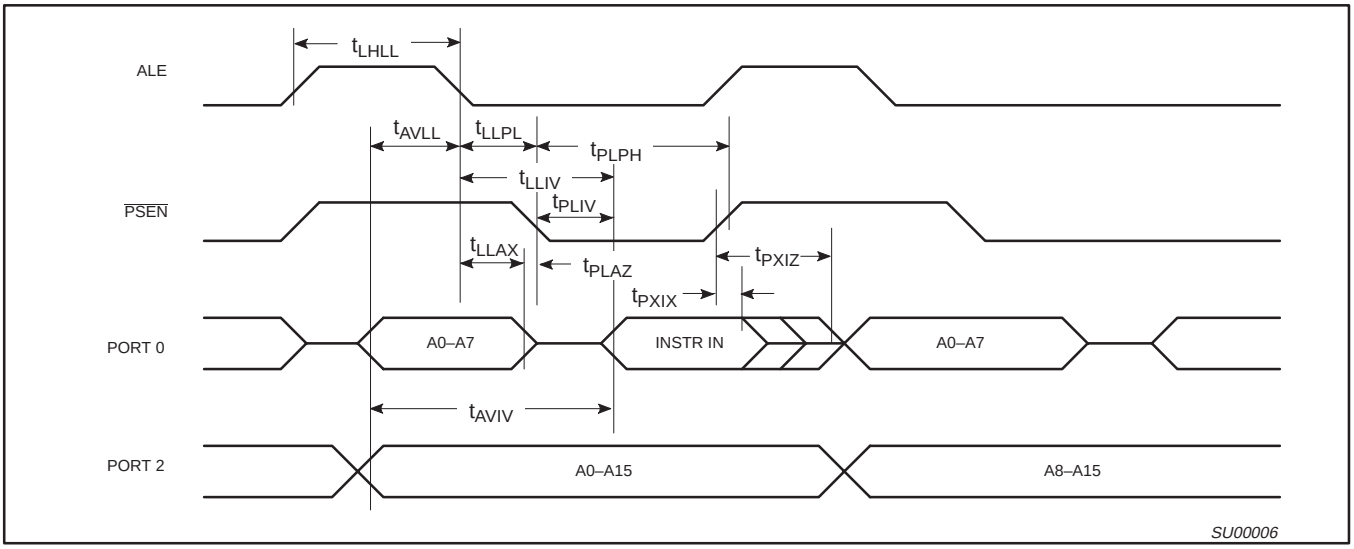


Figure 14. External Program Memory Read Cycle

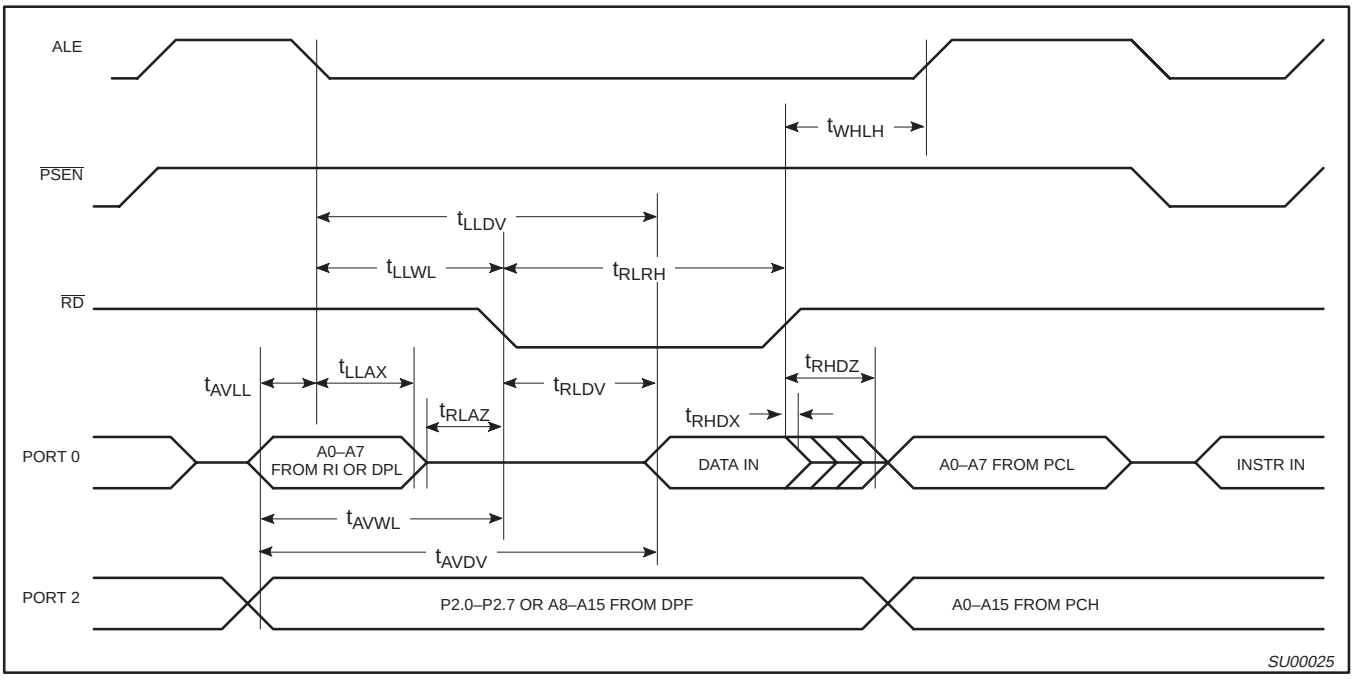


Figure 15. External Data Memory Read Cycle

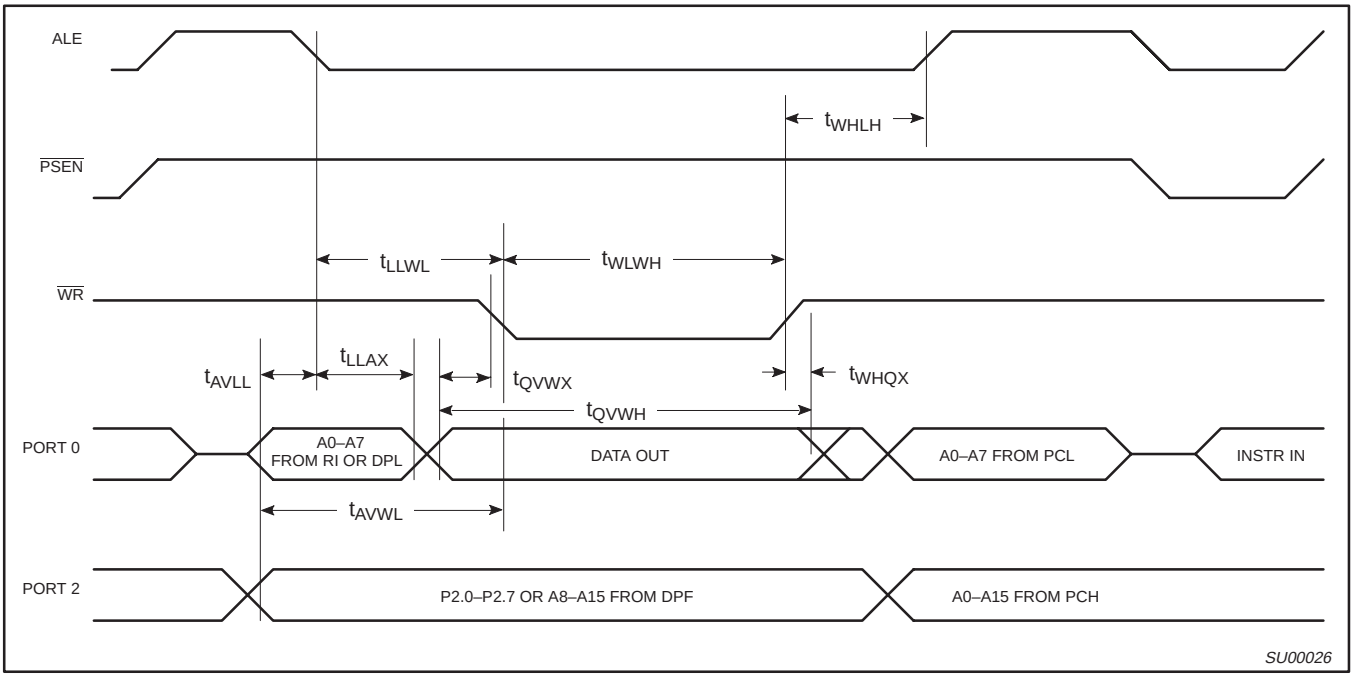


Figure 16. External Data Memory Write Cycle

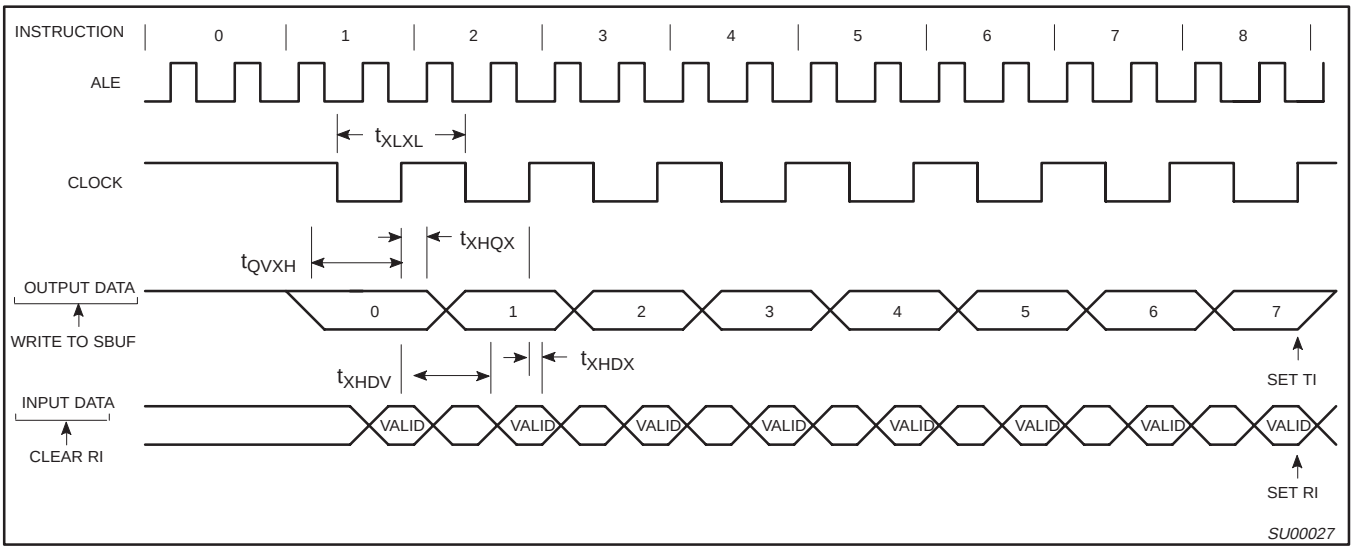


Figure 17. Shift Register Mode Timing

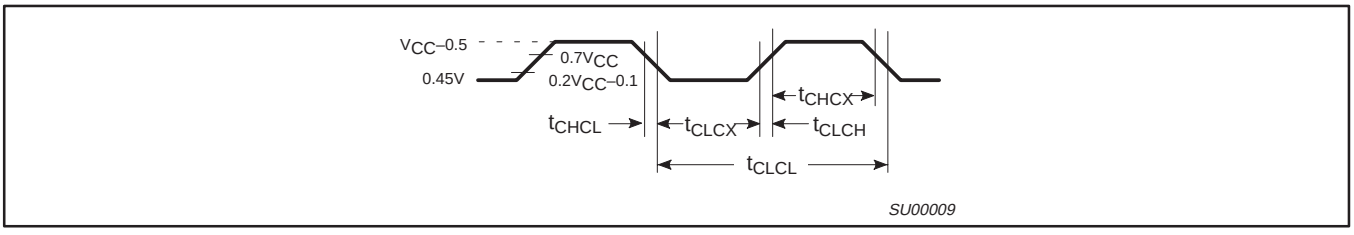
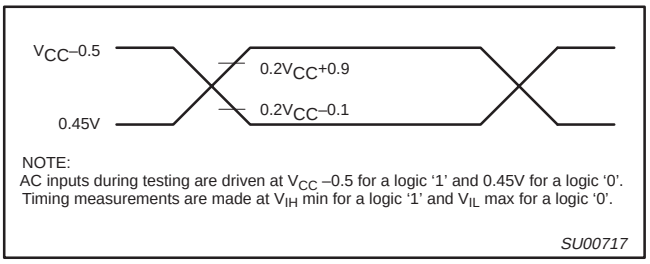


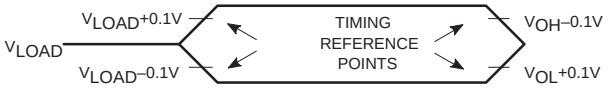
Figure 18. External Clock Drive



NOTE:
AC inputs during testing are driven at V_{CC} –0.5 for a logic '1' and 0.45V for a logic '0'. Timing measurements are made at V_{IH} min for a logic '1' and V_{IL} max for a logic '0'.

SU00717

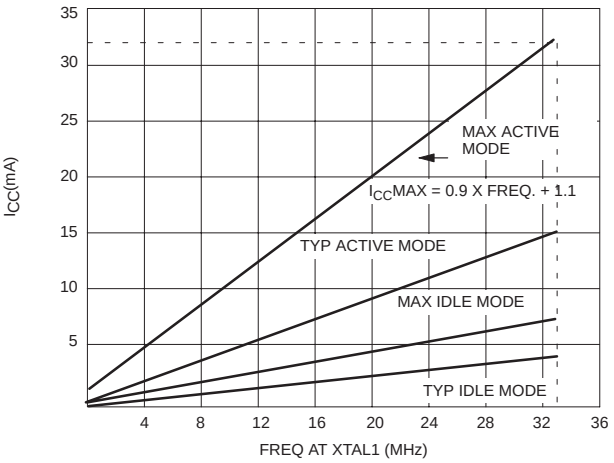
Figure 19. AC Testing Input/Output



NOTE:
For timing purposes, a port is no longer floating when a 100mV change from load voltage occurs, and begins to float when a 100mV change from the loaded V_{OH}/V_{OL} level occurs. $I_{OH}/I_{OL} \geq \pm 20$mA.

SU00718

Figure 20. Float Waveform



SU01413

Figure 21. I_{CC} vs. FREQ
Valid only within frequency specifications of the device under test

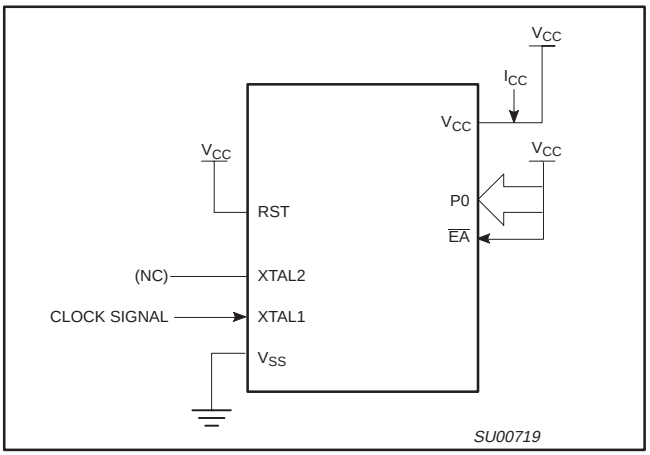


Figure 22. I_{CC} Test Condition, Active Mode
All other pins are disconnected

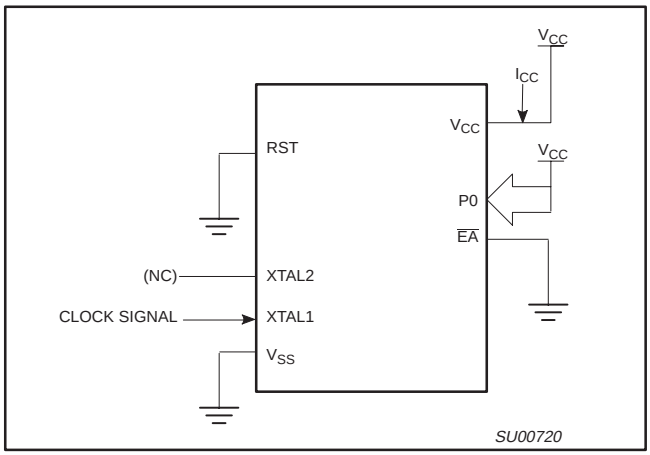


Figure 23. I_{CC} Test Condition, Idle Mode
All other pins are disconnected

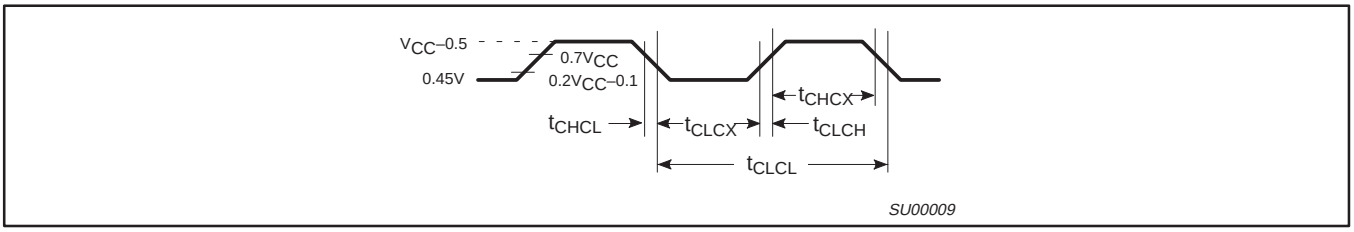


Figure 24. Clock Signal Waveform for I_{CC} Tests in Active and Idle Modes
$t_{CLCH} = t_{CHCL} = 5ns$

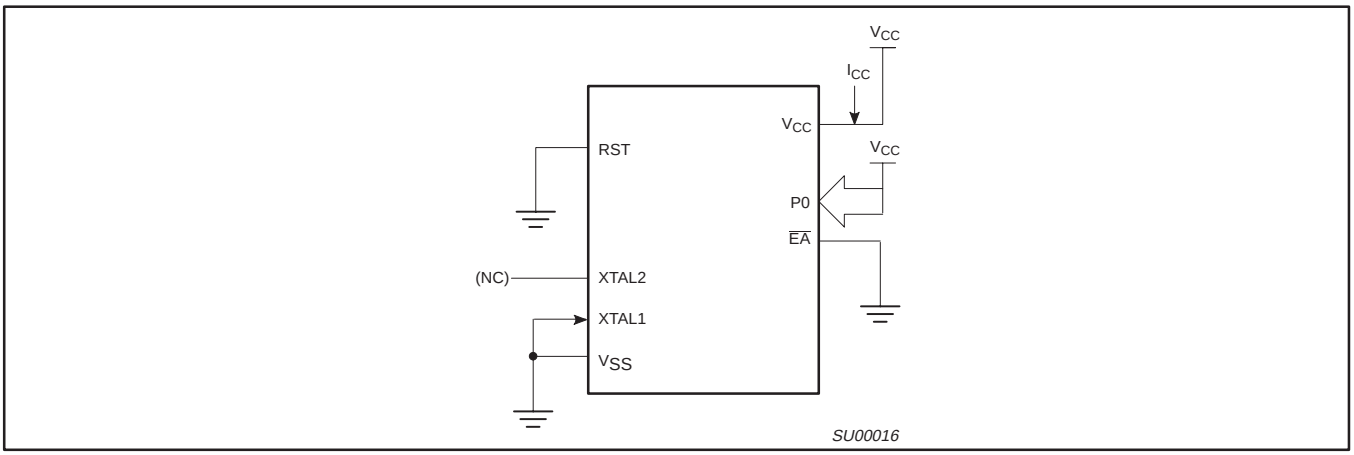
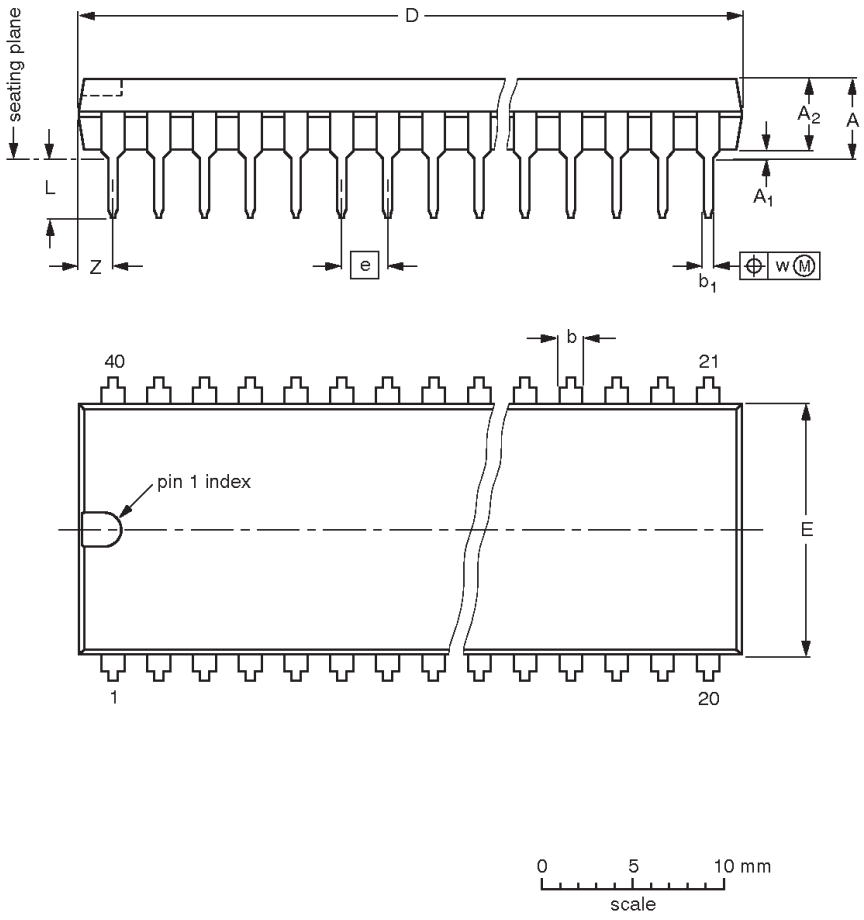


Figure 25. I_{CC} Test Condition, Power Down Mode
All other pins are disconnected. V_{CC} = 2 V to 5.5 V

DIP40: plastic dual in-line package; 40 leads (600 mil) SOT129-1



DIMENSIONS (inch dimensions are derived from the original mm dimensions)

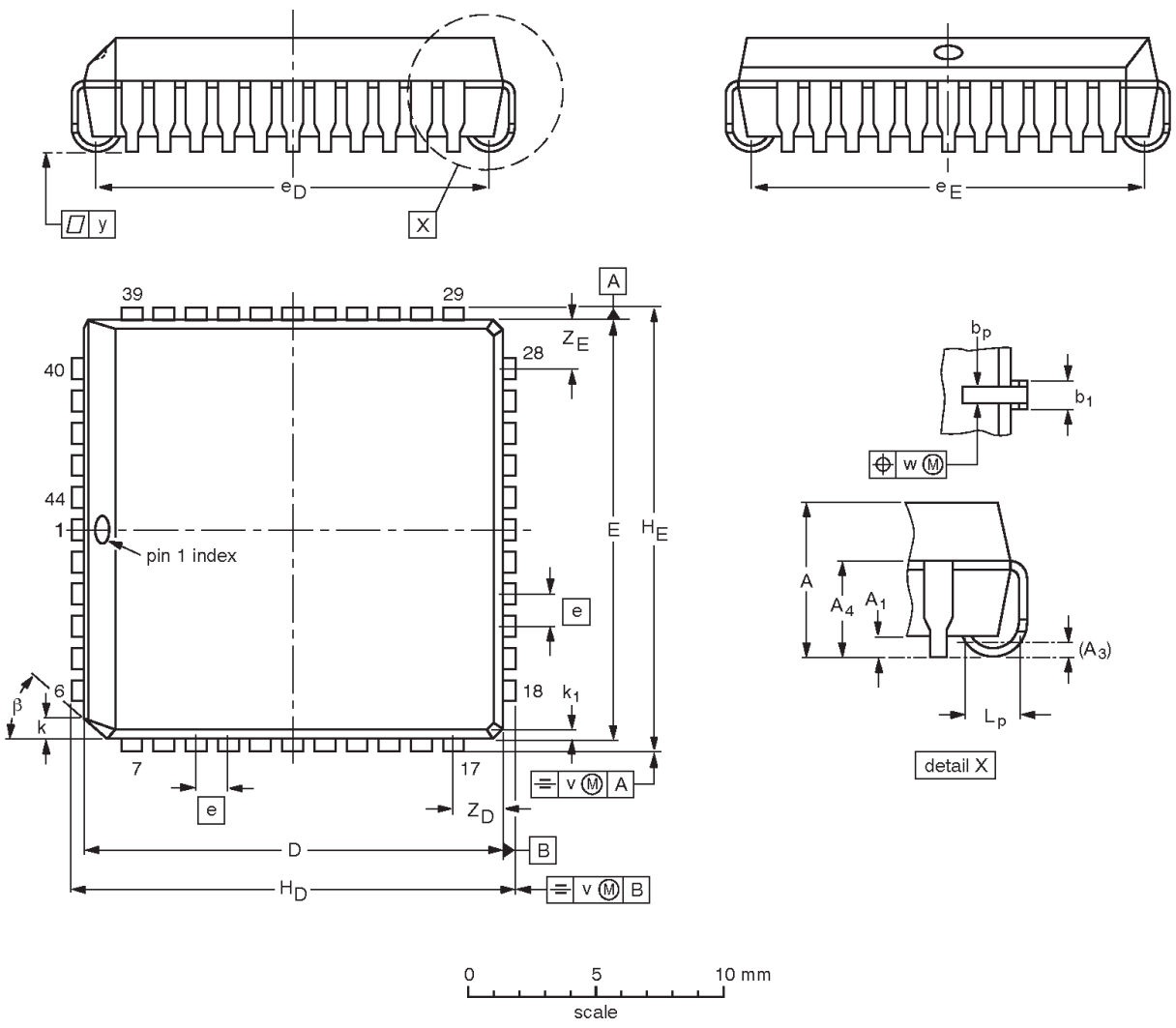
UNIT	A max.	A_1 min.	A_2 max.	b	b_1	c	D (1)	E (1)	e	e_1	L	M_E	M_H	w	Z (1) max.
mm	4.7	0.51	4.0	1.70 1.14	0.53 0.38	0.36 0.23	52.50 51.50	14.1 13.7	2.54	15.24	3.60 3.05	15.80 15.24	17.42 15.90	0.254	2.25
inches	0.19	0.020	0.16	0.067 0.045	0.021 0.015	0.014 0.009	2.067 2.028	0.56 0.54	0.10	0.60	0.14 0.12	0.62 0.60	0.69 0.63	0.01	0.089

Note

1. Plastic or metal protrusions of 0.25 mm maximum per side are not included.

OUTLINE VERSION	REFERENCES				EUROPEAN PROJECTION	ISSUE DATE
	IEC	JEDEC	EIAJ			
SOT129-1	051G08	MO-015	SC-511-40			~~95-01-14~~ 99-12-27

PLCC44: plastic leaded chip carrier; 44 leads SOT187-2



DIMENSIONS (millimetre dimensions are derived from the original inch dimensions)

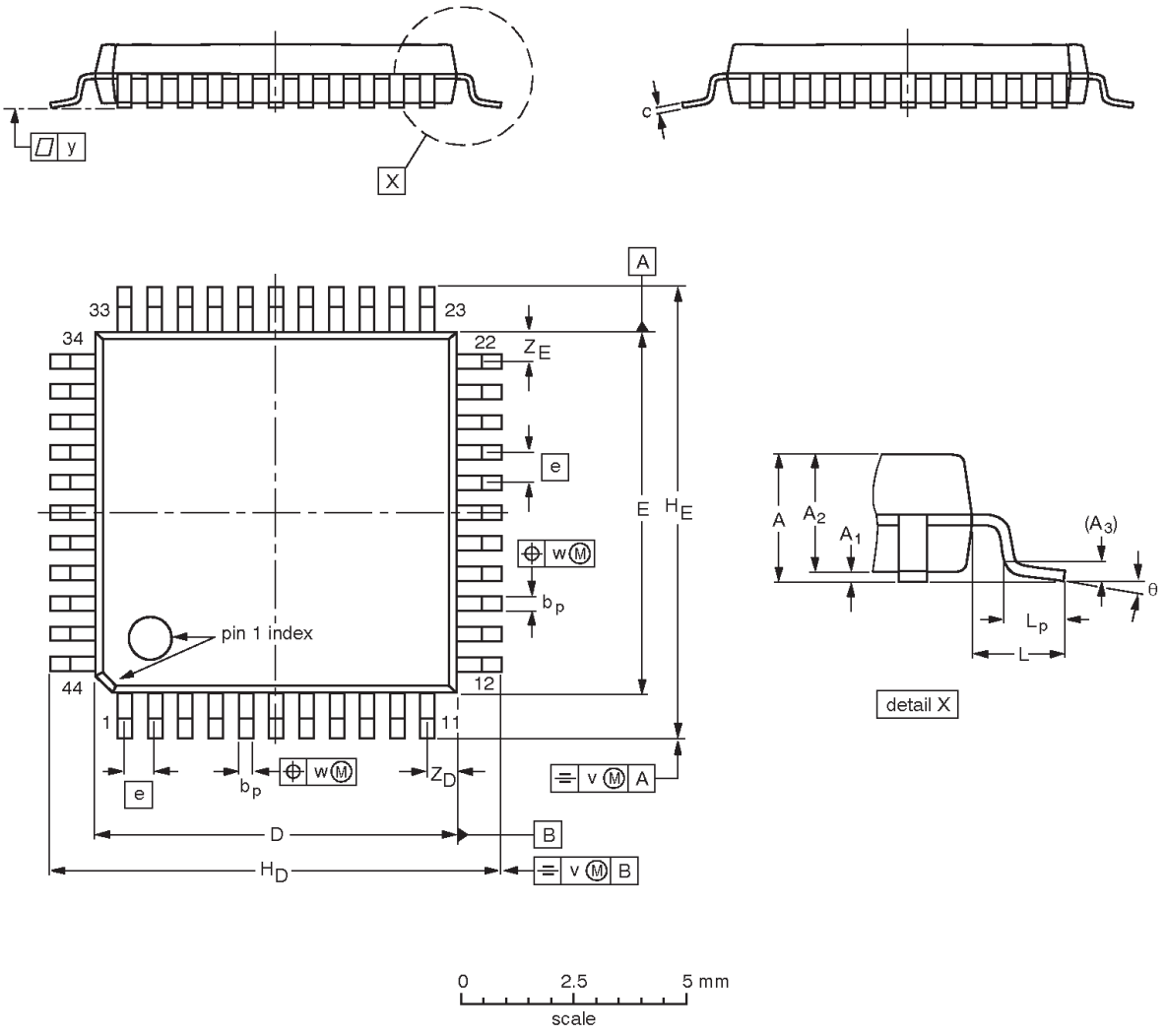
UNIT	A	A_1 min.	A_3	A_4 max.	b_p	b_1	$D^{(1)}$	$E^{(1)}$	e	e_D	e_E	H_D	H_E	k	k_1 max.	L_p	v	w	y	$Z_D^{(1)}$ max.	$Z_E^{(1)}$ max.	β
mm	4.57 4.19	0.51	0.25	3.05	0.53 0.33	0.81 0.66	16.66 16.51	16.66 16.51	1.27	16.00 14.99	16.00 14.99	17.65 17.40	17.65 17.40	1.22 1.07	0.51	1.44 1.02	0.18	0.18	0.10	2.16	2.16	45^o
inches	0.180 0.165	0.020	0.01	0.12	0.021 0.013	0.032 0.026	0.656 0.650	0.656 0.650	0.05	0.630 0.590	0.630 0.590	0.695 0.685	0.695 0.685	0.048 0.042	0.020	0.057 0.040	0.007	0.007	0.004	0.085	0.085	

Note

1. Plastic or metal protrusions of 0.01 inches maximum per side are not included.

OUTLINE VERSION	REFERENCES				EUROPEAN PROJECTION	ISSUE DATE
	IEC	JEDEC	EIAJ			
SOT187-2	112E10	MO-047				~~97-12-16~~ 99-12-27

QFP44: plastic quad flat package; 44 leads (lead length 1.3 mm); body 10 x 10 x 1.75 mm SOT307-2



DIMENSIONS (mm are the original dimensions)

UNIT	A max.	A_1	A_2	A_3	b_p	c	$D^{(1)}$	$E^{(1)}$	e	H_D	H_E	L	L_p	v	w	y	$Z_D^{(1)}$	$Z_E^{(1)}$	θ
mm	2.10	0.25 0.05	1.85 1.65	0.25	0.40 0.20	0.25 0.14	10.1 9.9	10.1 9.9	0.8	12.9 12.3	12.9 12.3	1.3	0.95 0.55	0.15	0.15	0.1	1.2 0.8	1.2 0.8	10^o 0^o

Note

1. Plastic or metal protrusions of 0.25 mm maximum per side are not included.

OUTLINE VERSION	REFERENCES				EUROPEAN PROJECTION	ISSUE DATE
	IEC	JEDEC	EIAJ			
SOT307-2						~~95-02-04~~ 97-08-01

Data sheet status

Data sheet status	Product status	Definition [1]
Objective specification	Development	This data sheet contains the design target or goal specifications for product development. Specification may change in any manner without notice.
Preliminary specification	Qualification	This data sheet contains preliminary data, and supplementary data will be published at a later date. Philips Semiconductors reserves the right to make changes at any time without notice in order to improve design and supply the best possible product.
Product specification	Production	This data sheet contains final specifications. Philips Semiconductors reserves the right to make changes at any time without notice in order to improve design and supply the best possible product.

[1] Please consult the most recently issued datasheet before initiating or completing a design.

Definitions

Short-form specification — The data in a short-form specification is extracted from a full data sheet with the same type number and title. For detailed information see the relevant data sheet or data handbook.

Limiting values definition — Limiting values given are in accordance with the Absolute Maximum Rating System (IEC 134). Stress above one or more of the limiting values may cause permanent damage to the device. These are stress ratings only and operation of the device at these or at any other conditions above those given in the Characteristics sections of the specification is not implied. Exposure to limiting values for extended periods may affect device reliability.

Application information — Applications that are described herein for any of these products are for illustrative purposes only. Philips Semiconductors make no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Disclaimers

Life support — These products are not designed for use in life support appliances, devices or systems where malfunction of these products can reasonably be expected to result in personal injury. Philips Semiconductors customers using or selling these products for use in such applications do so at their own risk and agree to fully indemnify Philips Semiconductors for any damages resulting from such application.

Right to make changes — Philips Semiconductors reserves the right to make changes, without notice, in the products, including circuits, standard cells, and/or software, described or contained herein in order to improve design and/or performance. Philips Semiconductors assumes no responsibility or liability for the use of any of these products, conveys no license or title under any patent, copyright, or mask work right to these products, and makes no representations or warranties that these products are free from patent, copyright, or mask work right infringement, unless otherwise specified.

Philips Semiconductors
811 East Arques Avenue
P.O. Box 3409
Sunnyvale, California 94088–3409
Telephone 800-234-7381

Date of release: 08-00

Document order number: 9397 750 07403

Let's make things better.

Philips Semiconductors

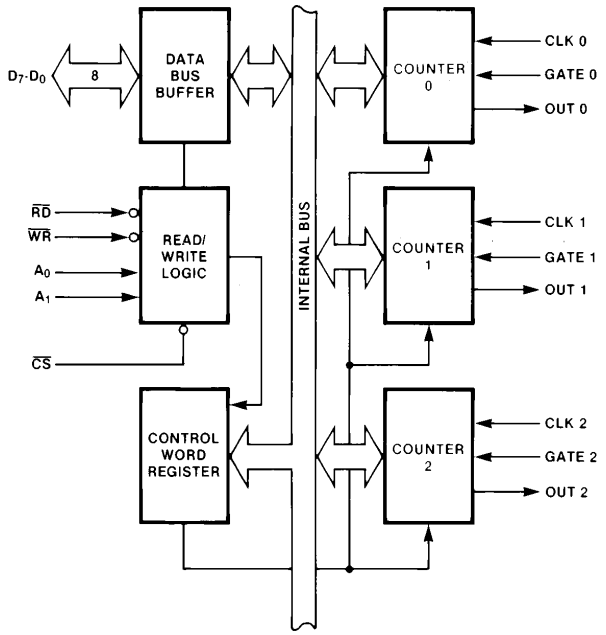
PHILIPS

8254
PROGRAMMABLE INTERVAL TIMER

- **Compatible with All Intel and Most Other Microprocessors**
- **Handles Inputs from DC to 10 MHz**
  — 8 MHz 8254
  — 10 MHz 8254-2
- **Status Read-Back Command**
- **Six Programmable Counter Modes**
- **Three Independent 16-Bit Counters**
- **Binary or BCD Counting**
- **Single +5V Supply**
- **Available in EXPRESS**
  — Standard Temperature Range

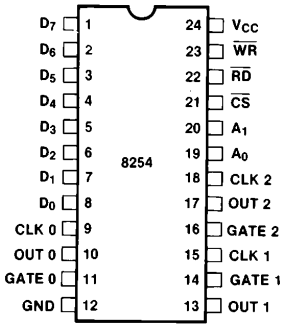
The Intel 8254 is a counter/timer device designed to solve the common timing control problems in microcomputer system design. It provides three independent 16-bit counters, each capable of handling clock inputs up to 10 MHz. All modes are software programmable. The 8254 is a superset of the 8253.

The 8254 uses HMOS technology and comes in a 24-pin plastic or CERDIP package.



231164–1

Figure 1. 8254 Block Diagram



231164–2

Figure 2. Pin Configuration

Table 1. Pin Description

Symbol	Pin No.	Type	Name and Function
D_7–D_0	1–8	I/O	**DATA:** Bi-directional three state data bus lines, connected to system data bus.
CLK 0	9	I	**CLOCK 0:** Clock input of Counter 0.
OUT 0	10	O	**OUTPUT 0:** Output of Counter 0.
GATE 0	11	I	**GATE 0:** Gate input of Counter 0.
GND	12		**GROUND:** Power supply connection.
V_{CC}	24		**POWER:** +5V power supply connection.
$\overline{WR}$	23	I	**WRITE CONTROL:** This input is low during CPU write operations.
$\overline{RD}$	22	I	**READ CONTROL:** This input is low during CPU read operations.
$\overline{CS}$	21	I	**CHIP SELECT:** A low on this input enables the 8254 to respond to $\overline{RD}$ and $\overline{WR}$ signals. $\overline{RD}$ and $\overline{WR}$ are ignored otherwise.
A_1, A_0	20–19	I	**ADDRESS:** Used to select one of the three Counters or the Control Word Register for read or write operations. Normally connected to the system address bus.

A_1	A_0	Selects
0	0	Counter 0
0	1	Counter 1
1	0	Counter 2
1	1	Control Word Register

CLK 2	18	I	**CLOCK 2:** Clock input of Counter 2.
OUT 2	17	O	**OUT 2:** Output of Counter 2.
GATE 2	16	I	**GATE 2:** Gate input of Counter 2.
CLK 1	15	I	**CLOCK 1:** Clock input of Counter 1.
GATE 1	14	I	**GATE 1:** Gate input of Counter 1.
OUT 1	13	O	**OUT 1:** Output of Counter 1.

FUNCTIONAL DESCRIPTION

General

The 8254 is a programmable interval timer/counter designed for use with Intel microcomputer systems. It is a general purpose, multi-timing element that can be treated as an array of I/O ports in the system software.

The 8254 solves one of the most common problems in any microcomputer system, the generation of accurate time delays under software control. Instead of setting up timing loops in software, the programmer configures the 8254 to match his requirements and programs one of the counters for the desired delay. After the desired delay, the 8254 will interrupt the CPU. Software overhead is minimal and variable length delays can easily be accommodated.

Some of the other counter/timer functions common to microcomputers which can be implemented with the 8254 are:

- Real time clock
- Event-counter
- Digital one-shot
- Programmable rate generator
- Square wave generator
- Binary rate multiplier
- Complex waveform generator
- Complex motor controller

Block Diagram

DATA BUS BUFFER

This 3-state, bi-directional, 8-bit buffer is used to interface the 8254 to the system bus (see Figure 3).

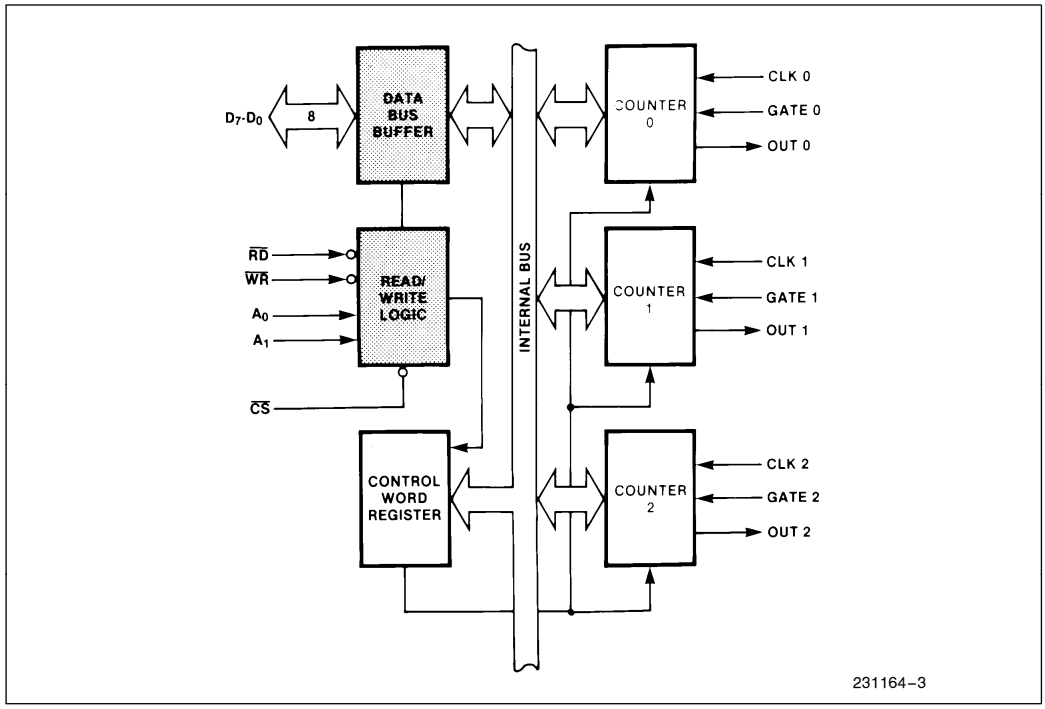


Figure 3. Block Diagram Showing Data Bus Buffer and Read/Write Logic Functions

READ/WRITE LOGIC

The Read/Write Logic accepts inputs from the system bus and generates control signals for the other functional blocks of the 8254. A_1 and A_0 select one of the three counters or the Control Word Register to be read from/written into. A ''low'' on the $\overline{RD}$ input tells the 8254 that the CPU is reading one of the counters. A ''low'' on the $\overline{WR}$ input tells the 8254 that the CPU is writing either a Control Word or an initial count. Both $\overline{RD}$ and $\overline{WR}$ are qualified by $\overline{CS}$; $\overline{RD}$ and $\overline{WR}$ are ignored unless the 8254 has been selected by holding $\overline{CS}$ low.

CONTROL WORD REGISTER

The Control Word Register (see Figure 4) is selected by the Read/Write Logic when $A_1,A_0 = 11$. If the CPU then does a write operation to the 8254, the data is stored in the Control Word Register and is interpreted as a Control Word used to define the operation of the Counters.

The Control Word Register can only be written to; status information is available with the Read-Back Command.

COUNTER 0, COUNTER 1, COUNTER 2

These three functional blocks are identical in operation, so only a single Counter will be described. The internal block diagram of a single counter is shown in Figure 5.

The Counters are fully independent. Each Counter may operate in a different Mode.

The Control Word Register is shown in the figure; it is not part of the Counter itself, but its contents determine how the Counter operates.

The status register, shown in Figure 5, when latched, contains the current contents of the Control Word Register and status of the output and null count flag. (See detailed explanation of the Read-Back command.)

The actual counter is labelled CE (for ''Counting Element''). It is a 16-bit presettable synchronous down counter.

OL_M and OL_L are two 8-bit latches. OL stands for ''Output Latch''; the subscripts M and L stand for ''Most significant byte'' and ''Least significant byte''

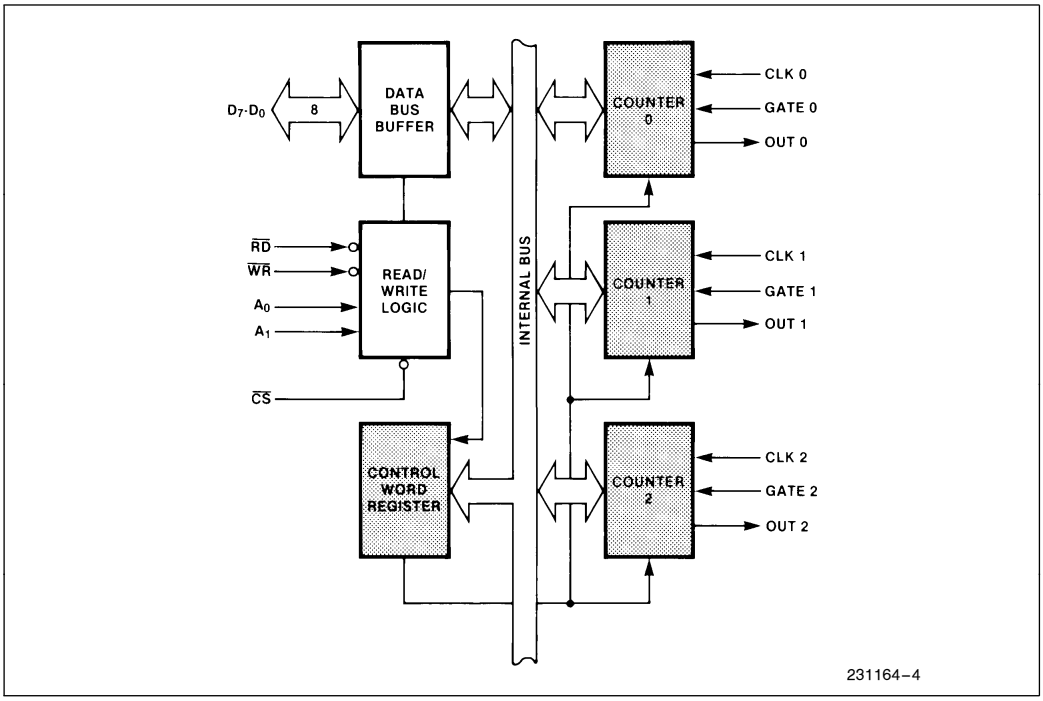


Figure 4. Block Diagram Showing Control Word Register and Counter Functions

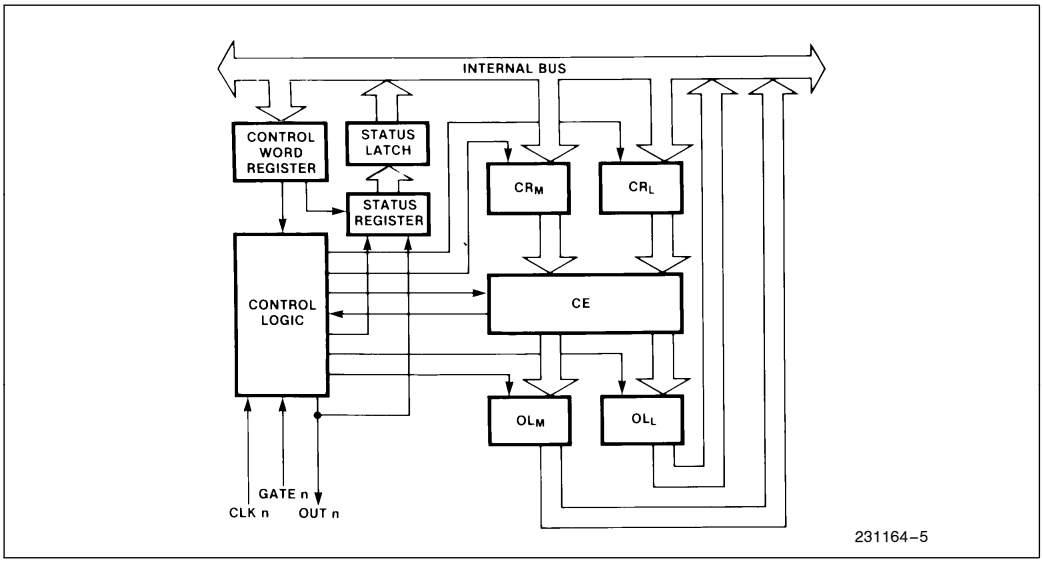


Figure 5. Internal Block Diagram of a Counter

respectively. Both are normally referred to as one unit and called just OL. These latches normally ''follow'' the CE, but if a suitable Counter Latch Command is sent to the 8254, the latches ''latch'' the present count until read by the CPU and then return to ''following'' the CE. One latch at a time is enabled by the counter's Control Logic to drive the internal bus. This is how the 16-bit Counter communicates over the 8-bit internal bus. Note that the CE itself cannot be read; whenever you read the count, it is the OL that is being read.

Similarly, there are two 8-bit registers called CR_M and CR_L (for ''Count Register''). Both are normally referred to as one unit and called just CR. When a new count is written to the Counter, the count is stored in the CR and later transferred to the CE. The Control Logic allows one register at a time to be loaded from the internal bus. Both bytes are transferred to the CE simultaneously. CR_M and CR_L are cleared when the Counter is programmed. In this way, if the Counter has been programmed for one byte counts (either most significant byte only or least significant byte only) the other byte will be zero. Note that the CE cannot be written into; whenever a count is written, it is written into the CR.

The Control Logic is also shown in the diagram. CLK n, GATE n, and OUT n are all connected to the outside world through the Control Logic.

8254 SYSTEM INTERFACE

The 8254 is a component of the Intel Microcomputer Systems and interfaces in the same manner as all other peripherals of the family. It is treated by the system's software as an array of peripheral I/O ports; three are counters and the fourth is a control register for MODE programming.

Basically, the select inputs A_0,A_1 connect to the A_0, A_1 address bus signals of the CPU. The CS can be derived directly from the address bus using a linear select method. Or it can be connected to the output of a decoder, such as an Intel 8205 for larger systems.

OPERATIONAL DESCRIPTION

General

After power-up, the state of the 8254 is undefined. The Mode, count value, and output of all Counters are undefined.

How each Counter operates is determined when it is programmed. Each Counter must be programmed before it can be used. Unused counters need not be programmed.

Programming the 8254

Counters are programmed by writing a Control Word and then an initial count.

The Control Words are written into the Control Word Register, which is selected when A_1,A_0 = 11. The Control Word itself specifies which Counter is being programmed.

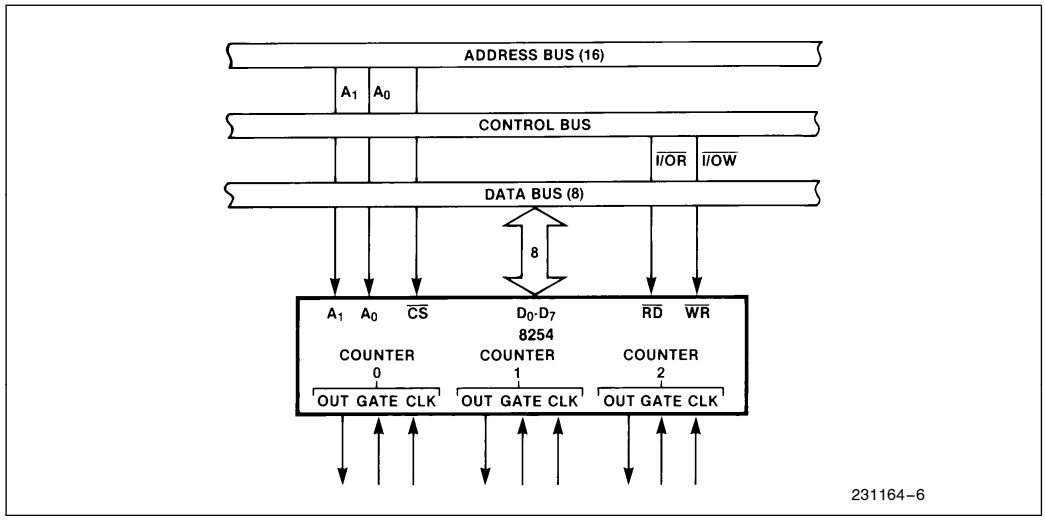


Figure 6. 8254 System Interface

Control Word Format

$A_1,A_0 = 11$ $\overline{CS} = 0$ $\overline{RD} = 1$ $\overline{WR} = 0$

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
SC1	SC0	RW1	RW0	M2	M1	M0	BCD

SC—Select Counter

SC1	SC0	
0	0	Select Counter 0
0	1	Select Counter 1
1	0	Select Counter 2
1	1	Read-Back Command (see Read Operations)

M—Mode

M2	M1	M0	
0	0	0	Mode 0
0	0	1	Mode 1
X	1	0	Mode 2
X	1	1	Mode 3
1	0	0	Mode 4
1	0	1	Mode 5

RW—Read/Write

RW1	RW0	
0	0	Counter Latch Command (see Read Operations)
0	1	Read/Write least significant byte only
1	0	Read/Write most significant byte only
1	1	Read/Write least significant byte first, then most significant byte

BCD

0	Binary Counter 16-bits
1	Binary Coded Decimal (BCD) Counter (4 Decades)

NOTE:
Don't care bits (X) should be 0 to insure compatibility with future Intel products.

Figure 7. Control Word Format

By contrast, initial counts are written into the Counters, not the Control Word Register. The A_1,A_0 inputs are used to select the Counter to be written into. The format of the initial count is determined by the Control Word used.

Write Operations

The programming procedure for the 8254 is very flexible. Only two conventions need to be remembered:

1) For each Counter, the Control Word must be written before the initial count is written.
2) The initial count must follow the count format specified in the Control Word (least significant byte only, most significant byte only, or least significant byte and then most significant byte).

Since the Control Word Register and the three Counters have separate addresses (selected by the A_1,A_0 inputs), and each Control Word specifies the Counter it applies to (SC0,SC1 bits), no special instruction sequence is required. Any programming sequence that follows the conventions in Figure 7 is acceptable.

A new initial count may be written to a Counter at any time without affecting the Counter's programmed Mode in any way. Counting will be affected as described in the Mode definitions. The new count must follow the programmed count format.

If a Counter is programmed to read/write two-byte counts, the following precaution applies: A program must not transfer control between writing the first and second byte to another routine which also writes into that same Counter. Otherwise, the Counter will be loaded with an incorrect count.

	A_1	A_0		A_1	A_0
Control Word—Counter 0	1	1	Control Word—Counter 2	1	1
LSB of count—Counter 0	0	0	Control Word—Counter 1	1	1
MSB of count—Counter 0	0	0	Control Word—Counter 0	1	1
Control Word—Counter 1	1	1	LSB of count—Counter 2	1	0
LSB of count—Counter 1	0	1	MSB of count—Counter 2	1	0
MSB of count—Counter 1	0	1	LSB of count—Counter 1	0	1
Control Word—Counter 2	1	1	MSB of count—Counter 1	0	1
LSB of count—Counter 2	1	0	LSB of count—Counter 0	0	0
MSB of count—Counter 2	1	0	MSB of count—Counter 0	0	0

	A_1	A_0		A_1	A_0
Control Word—Counter 0	1	1	Control Word—Counter 1	1	1
Control Word—Counter 1	1	1	Control Word—Counter 0	1	1
Control Word—Counter 2	1	1	LSB of count—Counter 1	0	1
LSB of count—Counter 2	1	0	Control Word—Counter 2	1	1
LSB of count—Counter 1	0	1	LSB of count—Counter 0	0	0
LSB of count—Counter 0	0	0	MSB of count—Counter 1	0	1
MSB of count—Counter 0	0	0	LSB of count—Counter 2	1	0
MSB of count—Counter 1	0	1	MSB of count—Counter 0	0	0
MSB of count—Counter 2	1	0	MSB of count—Counter 2	1	0

NOTE:
In all four examples, all Counters are programmed to read/write two-byte counts. These are only four of many possible programming sequences.

Figure 8. A Few Possible Programming Sequences

Read Operations

It is often desirable to read the value of a Counter without disturbing the count in progress. This is easily done in the 8254.

There are three possible methods for reading the counters: a simple read operation, the Counter Latch Command, and the Read-Back Command. Each is explained below. The first method is to perform a simple read operation. To read the Counter, which is selected with the A1, A0 inputs, the CLK input of the selected Counter must be inhibited by using either the GATE input or external logic. Otherwise, the count may be in the process of changing when it is read, giving an undefined result.

COUNTER LATCH COMMAND

The second method uses the ''Counter Latch Command''. Like a Control Word, this command is written to the Control Word Register, which is selected when $A_1,A_0 = 11$. Also like a Control Word, the SC0, SC1 bits select one of the three Counters, but two other bits, D5 and D4, distinguish this command from a Control Word.

A_1,A_0 = 11; CS = 0; RD = 1; WR = 0

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
SC1	SC0	0	0	X	X	X	X

SC1,SC0—specify counter to be latched

SC1	SC0	Counter
0	0	0
0	1	1
1	0	2
1	1	Read-Back Command

D5,D4—00 designates Counter Latch Command

X—don't care

NOTE:
Don't care bits (X) should be 0 to insure compatibility with future Intel products.

Figure 9. Counter Latching Command Format

The selected Counter's output latch (OL) latches the count at the time the Counter Latch Command is received. This count is held in the latch until it is read by the CPU (or until the Counter is reprogrammed). The count is then unlatched automatically and the OL returns to ''following'' the counting element (CE). This allows reading the contents of the Counters ''on the fly'' without affecting counting in progress. Multiple Counter Latch Commands may be used to latch more than one Counter. Each latched Counter's OL holds its count until it is read. Counter Latch Commands do not affect the programmed Mode of the Counter in any way.

If a Counter is latched and then, some time later, latched again before the count is read, the second Counter Latch Command is ignored. The count read will be the count at the time the first Counter Latch Command was issued.

With either method, the count must be read according to the programmed format; specifically, if the Counter is programmed for two byte counts, two bytes must be read. The two bytes do not have to be read one right after the other; read or write or programming operations of other Counters may be inserted between them.

Another feature of the 8254 is that reads and writes of the same Counter may be interleaved; for example, if the Counter is programmed for two byte counts, the following sequence is valid.

1) Read least significant byte.
2) Write new least significant byte.
3) Read most significant byte.
4) Write new most significant byte.

If a Counter is programmed to read/write two-byte counts, the following precaution applies: A program must not transfer control between reading the first and second byte to another routine which also reads from that same Counter. Otherwise, an incorrect count will be read.

READ-BACK COMMAND

The third method uses the Read-Back Command. This command allows the user to check the count value, programmed Mode, and current states of the OUT pin and Null Count flag of the selected counter(s).

The command is written into the Control Word Register and has the format shown in Figure 10. The command applies to the counters selected by setting their corresponding bits D3, D2, D1 = 1.

A0, A1 = 11 $\overline{CS}$ = 0 $\overline{RD}$ = 1 $\overline{WR}$ = 0

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
1	1	$\overline{COUNT}$	$\overline{STATUS}$	CNT 2	CNT 1	CNT 0	0

D_5: 0 = Latch count of selected counter(s)
D_4: 0 = Latch status of selected counters(s)
D_3: 1 = Select Counter 2
D_2: 1 = Select Counter 1
D_1: 1 = Select Counter 0
D_0: Reserved for future expansion; Must be 0

Figure 10. Read-Back Command Format

The read-back command may be used to latch multiple counter output latches (OL) by setting the $\overline{COUNT}$ bit D5 = 0 and selecting the desired counter(s). This single command is functionally equivalent to several counter latch commands, one for each counter latched. Each counter's latched count is held until it is read (or the counter is reprogrammed). The counter is automatically unlatched when read, but other counters remain latched until they are read. If multiple count read-back commands are issued to the same counter without reading the count, all but the first are ignored; i.e., the count which will be read is the count at the time the first read-back command was issued.

The read-back command may also be used to latch status information of selected counter(s) by setting $\overline{STATUS}$ bit D4 = 0. Status must be latched to be read; status of a counter is accessed by a read from that counter.

The counter status format is shown in Figure 11. Bits D5 through D0 contain the counter's programmed Mode exactly as written in the last Mode Control Word. OUTPUT bit D7 contains the current state of the OUT pin. This allows the user to monitor the counter's output via software, possibly eliminating some hardware from a system.

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
Output	Null Count	RW1	RW0	M2	M1	M0	BCD

D_7 1 = OUT Pin is 1
0 = OUT Pin is 0

D_6 1 = Null Count
0 = Count available for reading

D_5–D_0 Counter programmed mode (see Figure 7)

Figure 11. Status Byte

NULL COUNT bit D6 indicates when the last count written to the counter register (CR) has been loaded into the counting element (CE). The exact time this happens depends on the Mode of the counter and is described in the Mode Definitions, but until the count is loaded into the counting element (CE), it can't be read from the counter. If the count is latched or read before this time, the count value will not reflect the new count just written. The operation of Null Count is shown in Figure 12.

This Action	Causes
A. Write to the control word register;(1)	Null Count = 1
B. Write to the count register (CR);(2)	Null Count = 1
C. New Count is loaded into CE (CR → CE);	Null Count = 0

NOTE:
1. Only the counter specified by the control word will have its Null Count set to 1. Null count bits of other counters are unaffected.
2. If the counter is programmed for two-byte counts (least significant byte then most significant byte) Null Count goes to 1 when the second byte is written.

Figure 12. Null Count Operation

If multiple status latch operations of the counter(s) are performed without reading the status, all but the first are ignored; i.e., the status that will be read is the status of the counter at the time the first status read-back command was issued.

Both count and status of the selected counter(s) may be latched simultaneously by setting both $\overline{\text{COUNT}}$ and $\overline{\text{STATUS}}$ bits D5,D4 = 0. This is functionally the same as issuing two separate read-back commands at once, and the above discussions apply here also. Specifically, if multiple count and/or status read-back commands are issued to the same counter(s) without any intervening reads, all but the first are ignored. This is illustrated in Figure 13.

If both count and status of a counter are latched, the first read operation of that counter will return latched status, regardless of which was latched first. The next one or two reads (depending on whether the counter is programmed for one or two type counts) return latched count. Subsequent reads return unlatched count.

$\overline{CS}$	$\overline{RD}$	$\overline{WR}$	A_1	A_0	
0	1	0	0	0	Write into Counter 0
0	1	0	0	1	Write into Counter 1
0	1	0	1	0	Write into Counter 2
0	1	0	1	1	Write Control Word
0	0	1	0	0	Read from Counter 0
0	0	1	0	1	Read from Counter 1
0	0	1	1	0	Read from Counter 2
0	0	1	1	1	No-Operation (3-State)
1	X	X	X	X	No-Operation (3-State)
0	1	1	X	X	No-Operation (3-State)

Figure 14. Read/Write Operations Summary

Command								Description	Result
D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0		
1	1	0	0	0	0	1	0	Read back count and status of Counter 0	Count and status latched for Counter 0
1	1	1	0	0	1	0	0	Read back status of Counter 1	Status latched for Counter 1
1	1	1	0	1	1	0	0	Read back status of Counters 2, 1	Status latched for Counter 2, but not Counter 1
1	1	0	1	1	0	0	0	Read back count of Counter 2	Count latched for Counter 2
1	1	0	0	0	1	0	0	Read back count and status of Counter 1	Count latched for Counter 1, but not status
1	1	1	0	0	0	1	0	Read back status of Counter 1	Command ignored, status already latched for Counter 1

Figure 13. Read-Back Command Example

Mode Definitions

The following are defined for use in describing the operation of the 8254.

CLK Pulse: a rising edge, then a falling edge, in that order, of a Counter's CLK input.

Trigger: a rising edge of a Counter's GATE input.

Counter loading: the transfer of a count from the CR to the CE (refer to the ''Functional Description'')

MODE 0: INTERRUPT ON TERMINAL COUNT

Mode 0 is typically used for event counting. After the Control Word is written, OUT is initially low, and will remain low until the Counter reaches zero. OUT then goes high and remains high until a new count or a new Mode 0 Control Word is written into the Counter.

GATE = 1 enables counting; GATE = 0 disables counting. GATE has no effect on OUT.

After the Control Word and initial count are written to a Counter, the initial count will be loaded on the next CLK pulse. This CLK pulse does not decrement the count, so for an initial count of N, OUT does not go high until N + 1 CLK pulses after the initial count is written.

If a new count is written to the Counter, it will be loaded on the next CLK pulse and counting will continue from the new count. If a two-byte count is written, the following happens:

1) Writing the first byte disables counting. OUT is set low immediately (no clock pulse required)
2) Writing the second byte allows the new count to be loaded on the next CLK pulse.

This allows the counting sequence to be synchronized by software. Again, OUT does not go high until N+1 CLK pulses after the new count of N is written.

If an initial count is written while GATE = 0, it will still be loaded on the next CLK pulse. When GATE goes high, OUT will go high N CLK pulses later; no CLK pulse is needed to load the Counter as this has already been done.

MODE 1: HARDWARE RETRIGGERABLE ONE-SHOT

OUT will be initially high. OUT will go low on the CLK pulse following a trigger to begin the one-shot pulse, and will remain low until the Counter reaches zero. OUT will then go high and remain high until the CLK pulse after the next trigger.

After writing the Control Word and initial count, the Counter is armed. A trigger results in loading the Counter and setting OUT low on the next CLK pulse, thus starting the one-shot pulse. An initial count of N will result in a one-shot pulse N CLK cycles in duration. The one-shot is retriggerable, hence OUT will remain low for N CLK pulses after any trigger. The one-shot pulse can be repeated without rewriting the same count into the counter. GATE has no effect on OUT.

If a new count is written to the Counter during a one-shot pulse, the current one-shot is not affected unless the counter is retriggered. In that case, the Counter is loaded with the new count and the one-shot pulse continues until the new count expires.

MODE 2: RATE GENERATOR

This Mode functions like a divide-by-N counter. It is typically used to generate a Real Time Clock interrupt. OUT will initially be high. When the initial count has decremented to 1, OUT goes low for one CLK pulse. OUT then goes high again, the Counter reloads the initial count and the process is repeated. Mode 2 is periodic; the same sequence is repeated indefinitely. For an initial count of N, the sequence repeats every N CLK cycles.

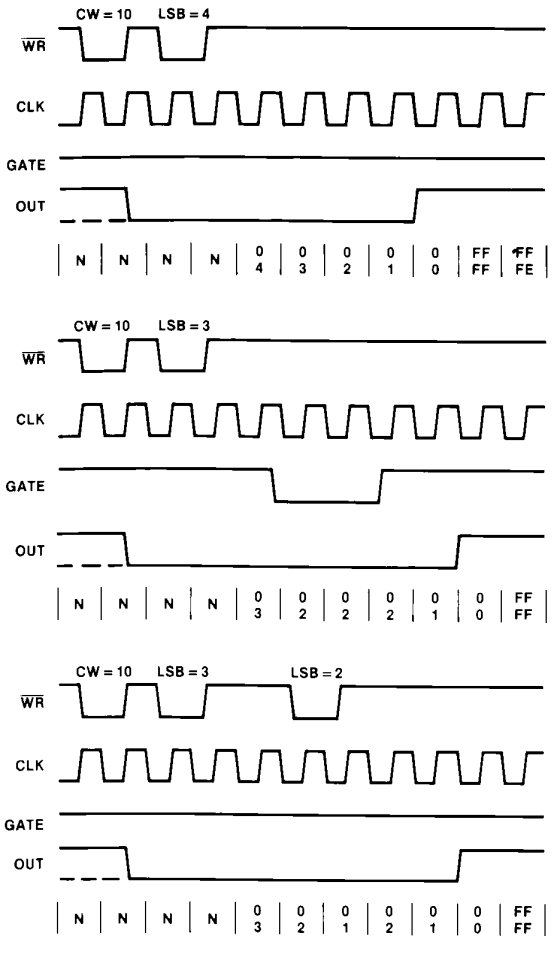
GATE = 1 enables counting; GATE = 0 disables counting. If GATE goes low during an output pulse, OUT is set high immediately. A trigger reloads the Counter with the initial count on the next CLK pulse; OUT goes low N CLK pulses after the trigger. Thus the GATE input can be used to synchronize the Counter.

After writing a Control Word and initial count, the Counter will be loaded on the next CLK pulse. OUT goes low N CLK Pulses after the initial count is written. This allows the Counter to be synchronized by software also.

Writing a new count while counting does not affect the current counting sequence. If a trigger is received after writing a new count but before the end of the current period, the Counter will be loaded with the new count on the next CLK pulse and counting will continue from the new count. Otherwise, the new count will be loaded at the end of the current counting cycle. In mode 2, a COUNT of 1 is illegal.

MODE 3: SQUARE WAVE MODE

Mode 3 is typically used for Baud rate generation. Mode 3 is similar to Mode 2 except for the duty cycle of OUT. OUT will initially be high. When half the



231164–7

NOTE:
The following conventions apply to all mode timing diagrams:
1. Counters are programmed for binary (not BCD) counting and for reading/writing least significant byte (LSB) only.
2. The counter is always selected ($\overline{CS}$ always low).
3. CW stands for ''Control Word''; CW = 10 means a control word of 10 HEX is written to the counter.
4. LSB stands for ''Least Significant Byte'' of count.
5. Numbers below diagrams are count values. The lower number is the least significant byte. The upper number is the most significant byte. Since the counter is programmed to read/write LSB only, the most significant byte cannot be read.
   N stands for an undefined count.
   Vertical lines show transitions between count values.

Figure 15. Mode 0

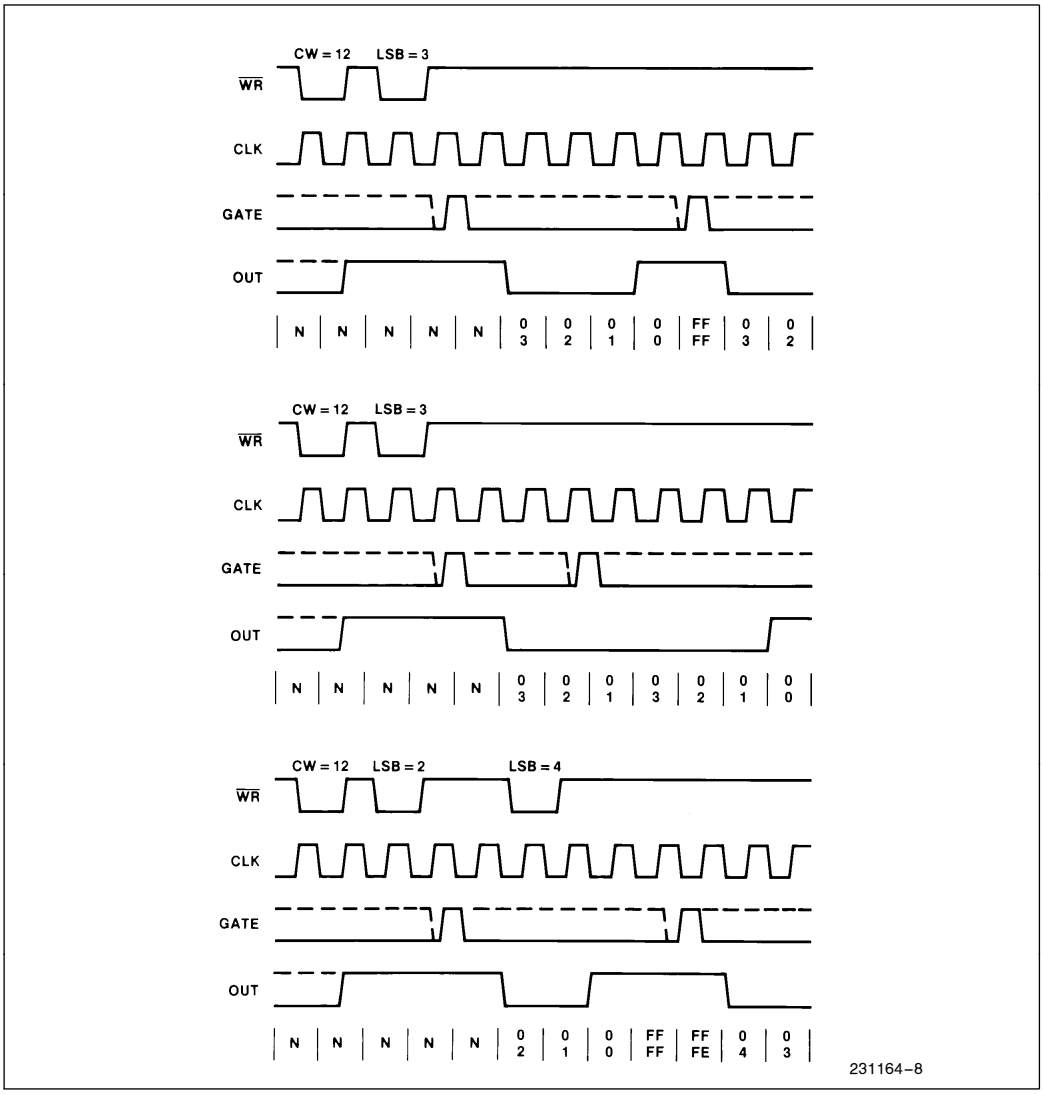


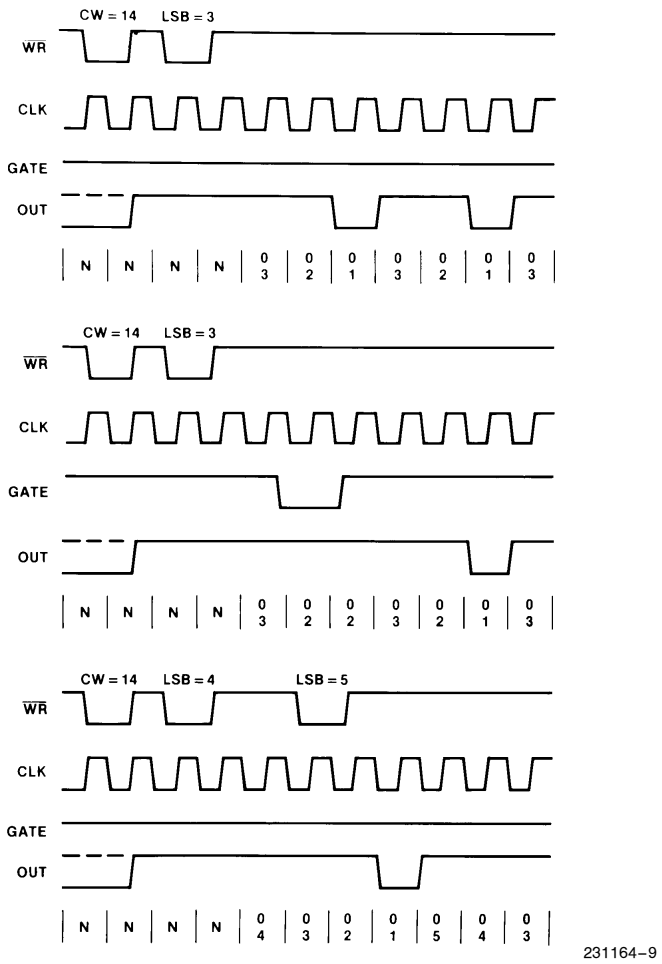
Figure 16. Mode 1

initial count has expired, OUT goes low for the remainder of the count. Mode 3 is periodic; the sequence above is repeated indefinitely. An initial count of N results in a square wave with a period of N CLK cycles.

GATE = 1 enables counting; GATE = 0 disables counting. If GATE goes low while OUT is low, OUT is set high immediately; no CLK pulse is required. A trigger reloads the Counter with the initial count on the next CLK pulse. Thus the GATE input can be used to synchronize the Counter.

After writing a Control Word and initial count, the Counter will be loaded on the next CLK pulse. This allows the Counter to be synchronized by software also.

Writing a new count while counting does not affect the current counting sequence. If a trigger is received after writing a new count but before the end of the current half-cycle of the square wave, the Counter will be loaded with the new count on the next CLK pulse and counting will continue from the



NOTE:
A GATE transition should not occur one clock prior to terminal count.

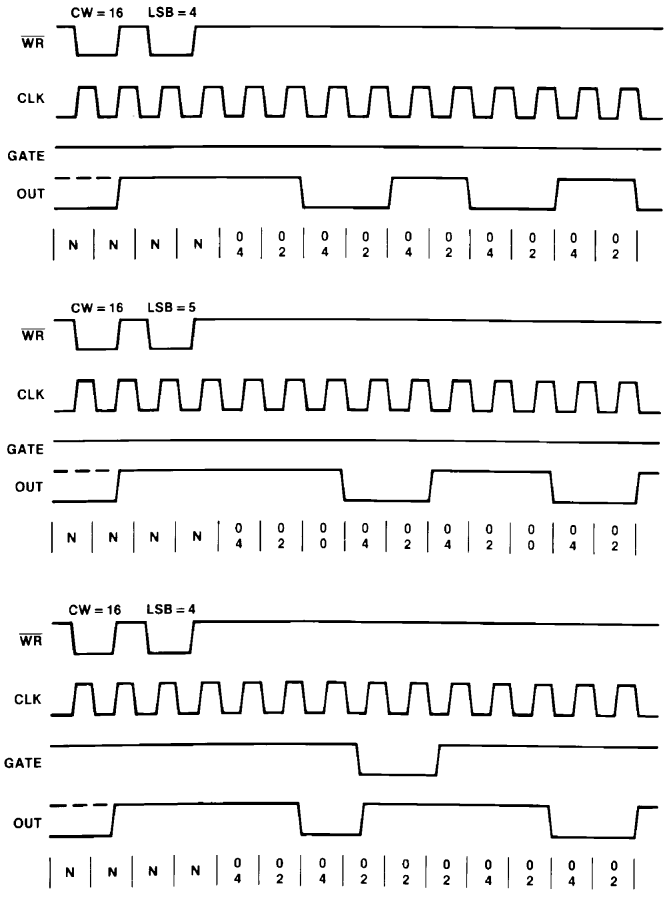
Figure 17. Mode 2

new count. Otherwise, the new count will be loaded at the end of the current half-cycle.

Mode 3 is implemented as follows:

Even counts: OUT is initially high. The initial count is loaded on one CLK pulse and then is decremented by two on succeeding CLK pulses. When the count expires OUT changes value and the Counter is reloaded with the initial count. The above process is repeated indefinitely.

Odd counts: OUT is initially high. The initial count minus one (an even number) is loaded on one CLK pulse and then is decremented by two on succeeding CLK pulses. One CLK pulse *after* the count expires, OUT goes low and the Counter is reloaded with the initial count minus one. Succeeding CLK pulses decrement the count by two. When the count expires, OUT goes high again and the Counter is reloaded with the initial count minus one. The above process is repeated indefinitely. So for odd counts, OUT will be high for $(N + 1)/2$ counts and low for $(N - 1)/2$ counts.



NOTE:
A GATE transition should not occur one clock prior to terminal count.

Figure 18. Mode 3

MODE 4: SOFTWARE TRIGGERED STROBE

OUT will be initially high. When the initial count expires, OUT will go low for one CLK pulse and then go high again. The counting sequence is ''triggered'' by writing the initial count.

GATE = 1 enables counting; GATE = 0 disables counting. GATE has no effect on OUT.

After writing a Control Word and initial count, the Counter will be loaded on the next CLK pulse. This CLK pulse does not decrement the count, so for an initial count of N, OUT does not strobe low until N + 1 CLK pulses after the initial count is written.

If a new count is written during counting, it will be loaded on the next CLK pulse and counting will continue from the new count. If a two-byte count is written, the following happens:

1) Writing the first byte has no effect on counting.
2) Writing the second byte allows the new count to be loaded on the next CLK pulse.

This allows the sequence to be ''retriggered'' by software. OUT strobes low N + 1 CLK pulses after the new count of N is written.

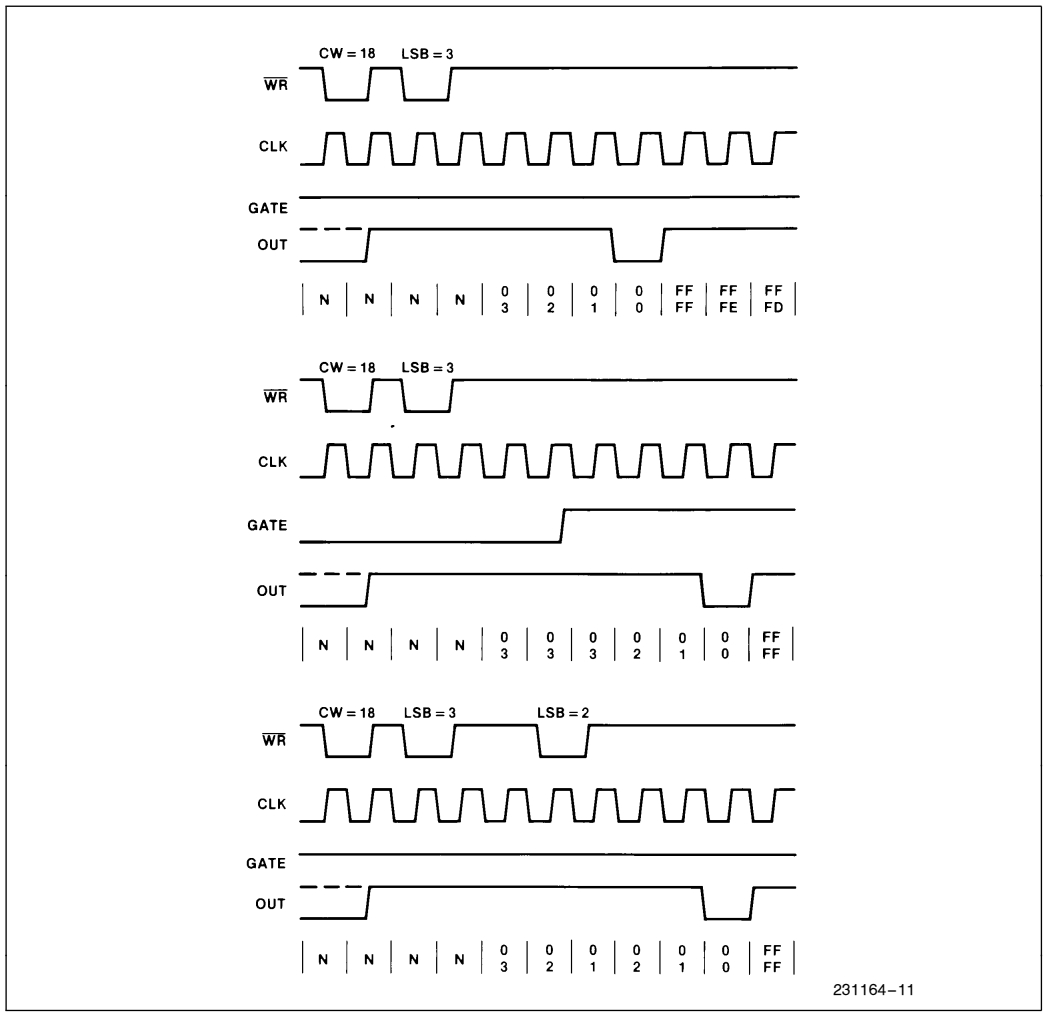


Figure 19. Mode 4

MODE 5: HARDWARE TRIGGERED STROBE (RETRIGGERABLE)

OUT will initially be high. Counting is triggered by a rising edge of GATE. When the initial count has expired, OUT will go low for one CLK pulse and then go high again.

After writing the Control Word and initial count, the counter will not be loaded until the CLK pulse after a trigger. This CLK pulse does not decrement the count, so for an initial count of N, OUT does not strobe low until N + 1 CLK pulses after a trigger.

A trigger results in the Counter being loaded with the initial count on the next CLK pulse. The counting sequence is retriggerable. OUT will not strobe low for N + 1 CLK pulses after any trigger. GATE has no effect on OUT.

If a new count is written during counting, the current counting sequence will not be affected. If a trigger occurs after the new count is written but before the current count expires, the Counter will be loaded with the new count on the next CLK pulse and counting will continue from there.

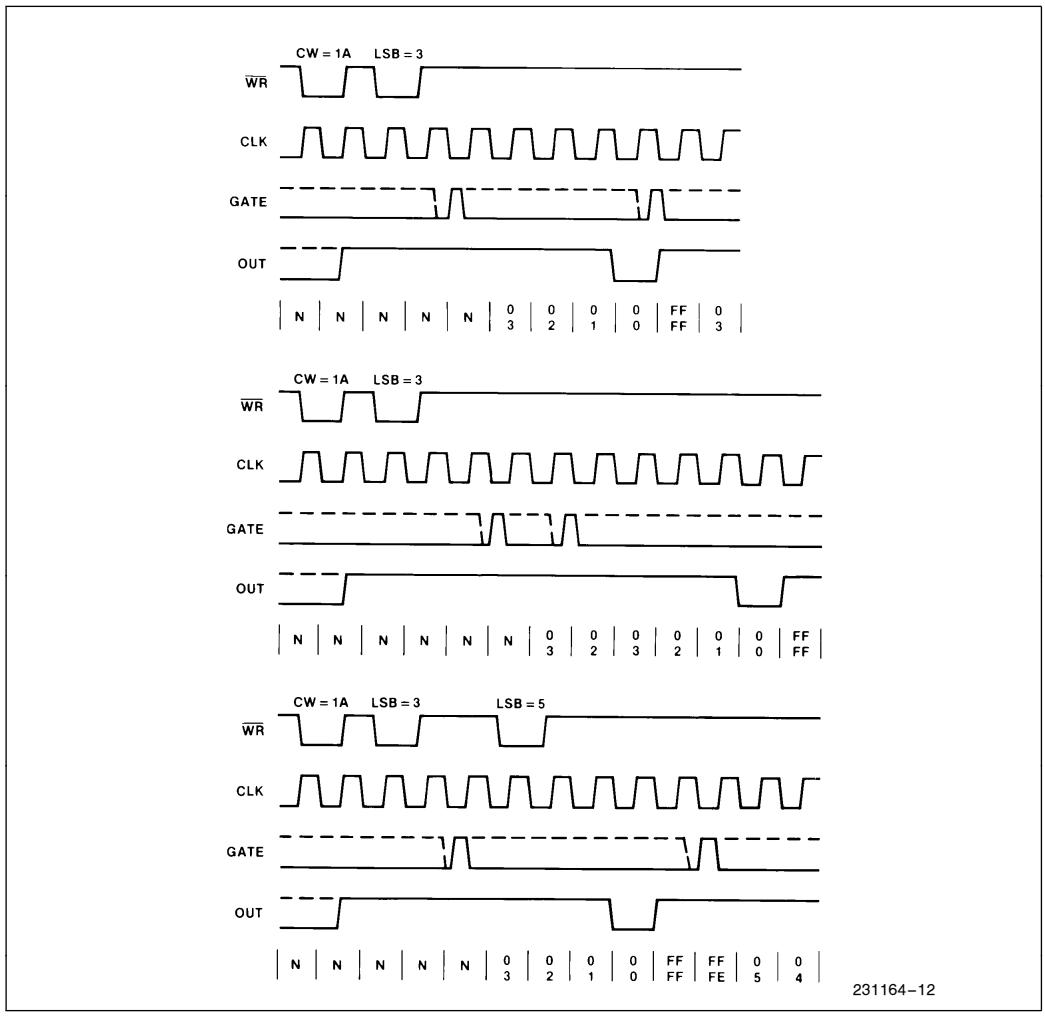


Figure 20. Mode 5

Signal Status Modes	Low Or Going Low	Rising	High
0	Disables Counting	— —	Enables Counting
1	— —	1) Initiates Counting 2) Resets Output after Next Clock	— —
2	1) Disables Counting 2) Sets Output Immediately High	Initiates Counting	Enables Counting
3	1) Disables Counting 2) Sets Output Immediately High	Initiates Counting	Enables Counting
4	Disables Counting	— —	Enables Counting
5	— —	Initiates Counting	— —

Figure 21. Gate Pin Operations Summary

Mode	Min Count	Max Count
0	1	0
1	1	0
2	2	0
3	2	0
4	1	0
5	1	0

NOTE:
0 is equivalent to 2^{16} for binary counting and 10^4 for BCD counting.

Figure 22. Minimum and Maximum Initial Counts

Operation Common to All Modes

PROGRAMMING

When a Control Word is written to a Counter, all Control Logic is immediately reset and OUT goes to a known initial state; no CLK pulses are required for this.

GATE

The GATE input is always sampled on the rising edge of CLK. In Modes 0, 2, 3, and 4 the GATE input is level sensitive, and the logic level is sampled on the rising edge of CLK. In Modes 1, 2, 3, and 5 the GATE input is rising-edge sensitive. In these Modes, a rising edge of GATE (trigger) sets an edge-sensitive flip-flop in the Counter. This flip-flop is then sampled on the next rising edge of CLK; the flip-flop is reset immediately after it is sampled. In this way, a trigger will be detected no matter when it occurs—a high logic level does not have to be maintained until the next rising edge of CLK. Note that in Modes 2 and 3, the GATE input is both edge- and level-sensitive. In Modes 2 and 3, if a CLK source other than the system clock is used, GATE should be pulsed immediately following $\overline{WR}$ of a new count value.

COUNTER

New counts are loaded and Counters are decremented on the falling edge of CLK.

The largest possible initial count is 0; this is equivalent to 2^{16} for binary counting and 10^4 for BCD counting.

The Counter does not stop when it reaches zero. In Modes 0, 1, 4, and 5 the Counter ‘‘wraps around’’ to the highest count, either FFFF hex for binary counting or 9999 for BCD counting, and continues counting. Modes 2 and 3 are periodic; the Counter reloads itself with the initial count and continues counting from there.

ABSOLUTE MAXIMUM RATINGS*

Ambient Temperature Under Bias . . . . . 0°C to 70°C
Storage Temperature . . . . . −65°C to +150°C
Voltage on Any Pin with Respect to Ground . . . . . −0.5V to +7V
Power Dissipation 1W

NOTICE: This is a production data sheet. The specifications are subject to change without notice.

**WARNING: Stressing the device beyond the "Absolute Maximum Ratings" may cause permanent damage. These are stress ratings only. Operation beyond the "Operating Conditions" is not recommended and extended exposure beyond the "Operating Conditions" may affect device reliability.*

D.C. CHARACTERISTICS $T_A = 0°C$ to 70°C, $V_{CC} = 5V \pm 10\%$

Symbol	Parameter	Min	Max	Units	Test Conditions
V_{IL}	Input Low Voltage	−0.5	0.8	V	
V_{IH}	Input High Voltage	2.0	V_{CC} +0.5V	V	
V_{OL}	Output Low Voltage		0.45	V	$I_{OL} = 2.0$ mA
V_{OH}	Output High Voltage	2.4		V	$I_{OH} = -400\ \mu A$
I_{IL}	Input Load Current		±10	μA	$V_{IN} = V_{CC}$ to 0V
I_{OFL}	Output Float Leakage		±10	μA	$V_{OUT} = V_{CC}$ to 0.45V
I_{CC}	V_{CC} Supply Current		170	mA	
C_{IN}	Input Capacitance		10	pF	$f_c = 1$ MHz
$C_{I/O}$	I/O Capacitance		20	pF	Unmeasured pins returned to V_{SS}(4)

A.C. CHARACTERISTICS $T_A = 0°C$ to 70°C, $V_{CC} = 5V \pm 10\%$, GND = 0V

Bus Parameters(1)

READ CYCLE

Symbol	Parameter	8254		8254-2		Unit
		Min	Max	Min	Max	
t_{AR}	Address Stable Before $\overline{RD}\downarrow$	45		30		ns
t_{SR}	$\overline{CS}$ Stable Before $\overline{RD}\downarrow$	0		0		ns
t_{RA}	Address Hold Time After $\overline{RD}\uparrow$	0		0		ns
t_{RR}	$\overline{RD}$ Pulse Width	150		95		ns
t_{RD}	Data Delay from $\overline{RD}\downarrow$		120		85	ns
t_{AD}	Data Delay from Address		220		185	ns
t_{DF}	$\overline{RD}\uparrow$ to Data Floating	5	90	5	65	ns
t_{RV}	Command Recovery Time	200		165		ns

NOTE:
1. AC timings measured at $V_{OH} = 2.0V$, $V_{OL} = 0.8V$.

A.C. CHARACTERISTICS $T_A = 0°C$ to $70°C$, $V_{CC} = 5V \pm 10\%$, GND = 0V (Continued)

WRITE CYCLE

Symbol	Parameter	8254		8254-2		Unit
		Min	Max	Min	Max	
t_{AW}	Address Stable Before $\overline{WR}$ ↓	0		0		ns
t_{SW}	$\overline{CS}$ Stable Before $\overline{WR}$ ↓	0		0		ns
t_{WA}	Address Hold Time After $\overline{WR}$ ↓	0		0		ns
t_{WW}	$\overline{WR}$ Pulse Width	150		95		ns
t_{DW}	Data Setup Time Before $\overline{WR}$ ↑	120		95		ns
t_{WD}	Data Hold Time After $\overline{WR}$ ↑	0		0		ns
t_{RV}	Command Recovery Time	200		165		ns

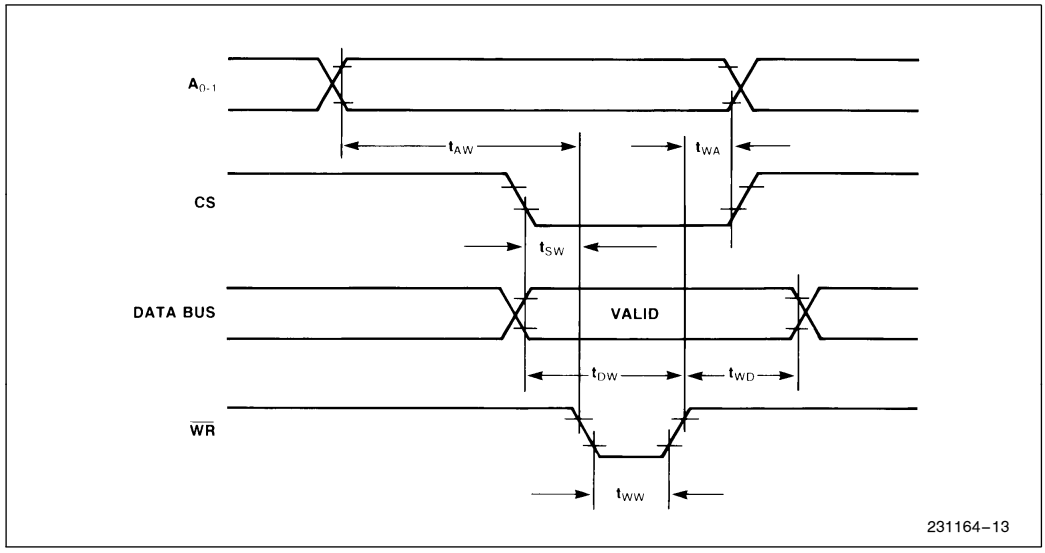
CLOCK AND GATE

Symbol	Parameter	8254		8254-2		Unit
		Min	Max	Min	Max	
t_{CLK}	Clock Period	125	DC	100	DC	ns
t_{PWH}	High Pulse Width	60(3)		30(3)		ns
t_{PWL}	Low Pulse Width	60(3)		50(3)		ns
t_R	Clock Rise Time		25		25	ns
t_F	Clock Fall Time		25		25	ns
t_{GW}	Gate Width High	50		50		ns
t_{GL}	Gate Width Low	50		50		ns
t_{GS}	Gate Setup Time to CLK ↑	50		40		ns
t_{GH}	Gate Setup Time After CLK ↑	50(2)		50(2)		ns
t_{OD}	Output Delay from CLK ↓		150		100	ns
t_{ODG}	Output Delay from Gate ↓		120		100	ns
t_{WC}	CLK Delay for Loading ↓	0	55	0	55	ns
t_{WG}	Gate Delay for Sampling	−5	50	−5	40	ns
t_{WO}	OUT Delay from Mode Write		260		240	ns
t_{CL}	CLK Set Up for Count Latch	−40	45	−40	40	ns

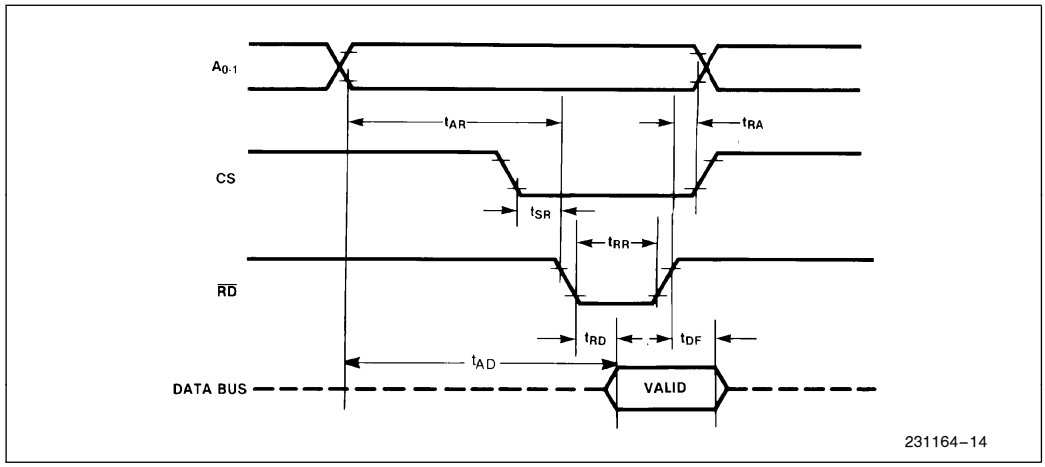
NOTES:
2. In Modes 1 and 5 triggers are sampled on each rising clock edge. A second trigger within 120 ns (70 ns for the 8254-2) of the rising clock edge may not be detected.
3. Low-going glitches that violate t_{PWH}, t_{PWL} may cause errors requiring counter reprogramming.
4. Sampled, not 100% tested. $T_A = 25°C$.
5. If CLK present at TWC min then Count equals N+2 CLK pulses, TWC max equals Count N+1 CLK pulse. TWC min to TWC max, count will be either N+1 or N+2 CLK pulses.
6. In Modes 1 and 5, if GATE is present when writing a new Count value, at TWG min Counter will not be triggered, at TWG max Counter will be triggered.
7. If CLK present when writing a Counter Latch or ReadBack Command, at TCL min CLK will be reflected in count value latched, at TCL max CLK will not be reflected in the count value latched.

WAVEFORMS

WRITE

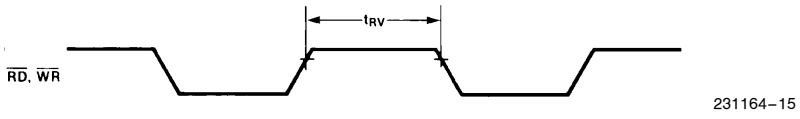


READ

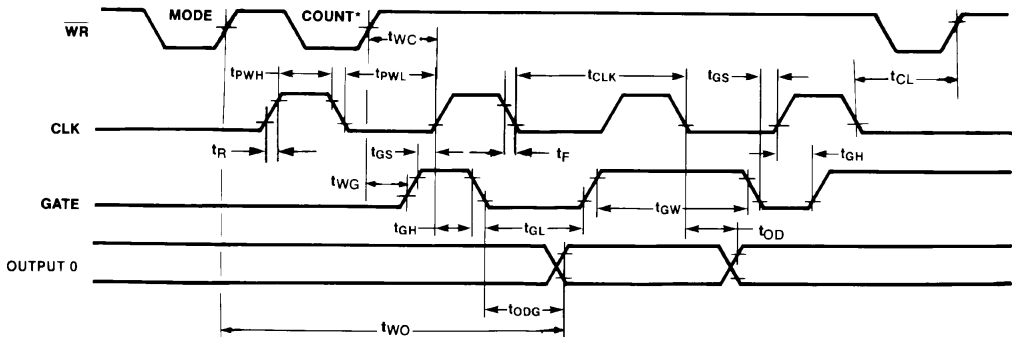


WAVEFORMS (Continued)

RECOVERY

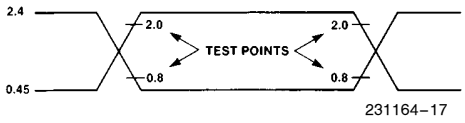


CLOCK AND GATE



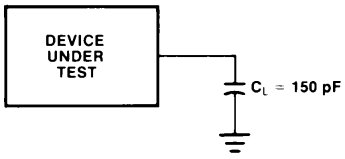
*Last byte of count being written.

A.C. TESTING INPUT, OUTPUT WAVEFORM



A.C. Testing: Inputs are driven at 2.4V for a Logic "1" and 0.45V for a Logic "0." Timing measurements are made at 2.0V for a Logic "1" and 0.8V for a Logic "0".

A.C. TESTING LOAD CIRCUIT



C_L = 150 pF
C_L Includes Jig Capacitance

REVISION SUMMARY

The following list represents the key differences between Rev. 004 and Rev. 005 of the 8254 Data Sheet.

1. References to and specifications for the 5 MHz 8254-5 are removed. Only the 8 MHz 8254 and the 10 MHz 8254-2 remain in production.

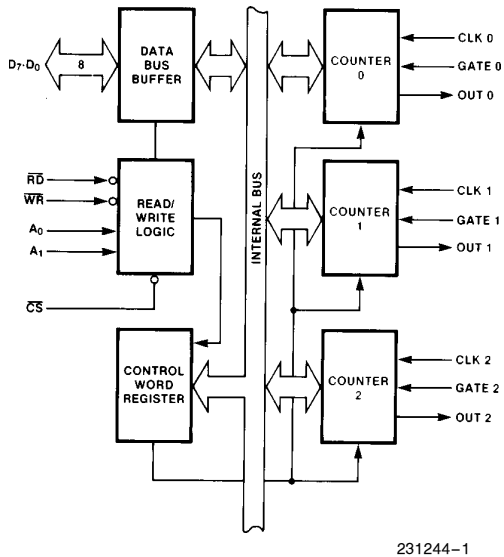
82C54
CHMOS PROGRAMMABLE INTERVAL TIMER

- **Compatible with all Intel and most other microprocessors**
- **High Speed, "Zero Wait State" Operation with 8 MHz 8086/88 and 80186/188**
- **Handles Inputs from DC — 10 MHz for 82C54-2**
- **Available in EXPRESS — Standard Temperature Range — Extended Temperature Range**
- **Three independent 16-bit counters**
- **Low Power CHMOS — I_{CC} = 10 mA @ 8 MHz Count frequency**
- **Completely TTL Compatible**
- **Six Programmable Counter Modes**
- **Binary or BCD counting**
- **Status Read Back Command**
- **Available in 24-Pin DIP and 28-Pin PLCC**

The Intel 82C54 is a high-performance, CHMOS version of the industry standard 8254 counter/timer which is designed to solve the timing control problems common in microcomputer system design. It provides three independent 16-bit counters, each capable of handling clock inputs up to 10 MHz. All modes are software programmable. The 82C54 is pin compatible with the HMOS 8254, and is a superset of the 8253.

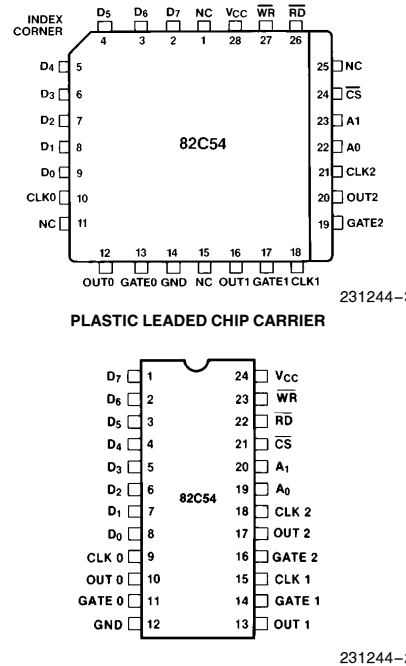
Six programmable timer modes allow the 82C54 to be used as an event counter, elapsed time indicator, programmable one-shot, and in many other applications.

The 82C54 is fabricated on Intel's advanced CHMOS III technology which provides low power consumption with performance equal to or greater than the equivalent HMOS product. The 82C54 is available in 24-pin DIP and 28-pin plastic leaded chip carrier (PLCC) packages.



231244–1

Figure 1. 82C54 Block Diagram



231244–3

PLASTIC LEADED CHIP CARRIER

231244–2

Diagrams are for pin reference only.
Package sizes are not to scale.

Figure 2. 82C54 Pinout

Table 1. Pin Description

Symbol	Pin Number		Type	Function
	DIP	PLCC		
D_7-D_0	1-8	2-9	I/O	Data: Bidirectional tri-state data bus lines, connected to system data bus.
CLK 0	9	10	I	Clock 0: Clock input of Counter 0.
OUT 0	10	12	O	Output 0: Output of Counter 0.
GATE 0	11	13	I	Gate 0: Gate input of Counter 0.
GND	12	14		Ground: Power supply connection.
OUT 1	13	16	O	Out 1: Output of Counter 1.
GATE 1	14	17	I	Gate 1: Gate input of Counter 1.
CLK 1	15	18	I	Clock 1: Clock input of Counter 1.
GATE 2	16	19	I	Gate 2: Gate input of Counter 2.
OUT 2	17	20	O	Out 2: Output of Counter 2.
CLK 2	18	21	I	Clock 2: Clock input of Counter 2.
A_1, A_0	20-19	23-22	I	Address: Used to select one of the three Counters or the Control Word Register for read or write operations. Normally connected to the system address bus.
				A_1 / **A_0** / **Selects** 0 / 0 / Counter 0 0 / 1 / Counter 1 1 / 0 / Counter 2 1 / 1 / Control Word Register
$\overline{CS}$	21	24	I	Chip Select: A low on this input enables the 82C54 to respond to $\overline{RD}$ and $\overline{WR}$ signals. $\overline{RD}$ and $\overline{WR}$ are ignored otherwise.
$\overline{RD}$	22	26	I	Read Control: This input is low during CPU read operations.
$\overline{WR}$	23	27	I	Write Control: This input is low during CPU write operations.
V_{CC}	24	28		Power: +5V power supply connection.
NC		1, 11, 15, 25		No Connect

FUNCTIONAL DESCRIPTION

General

The 82C54 is a programmable interval timer/counter designed for use with Intel microcomputer systems. It is a general purpose, multi-timing element that can be treated as an array of I/O ports in the system software.

The 82C54 solves one of the most common problems in any microcomputer system, the generation of accurate time delays under software control. Instead of setting up timing loops in software, the programmer configures the 82C54 to match his requirements and programs one of the counters for the desired delay. After the desired delay, the 82C54 will interrupt the CPU. Software overhead is minimal and variable length delays can easily be accommodated.

Some of the other counter/timer functions common to microcomputers which can be implemented with the 82C54 are:

- Real time clock
- Even counter
- Digital one-shot
- Programmable rate generator
- Square wave generator
- Binary rate multiplier
- Complex waveform generator
- Complex motor controller

Block Diagram

DATA BUS BUFFER

This 3-state, bi-directional, 8-bit buffer is used to interface the 82C54 to the system bus (see Figure 3).

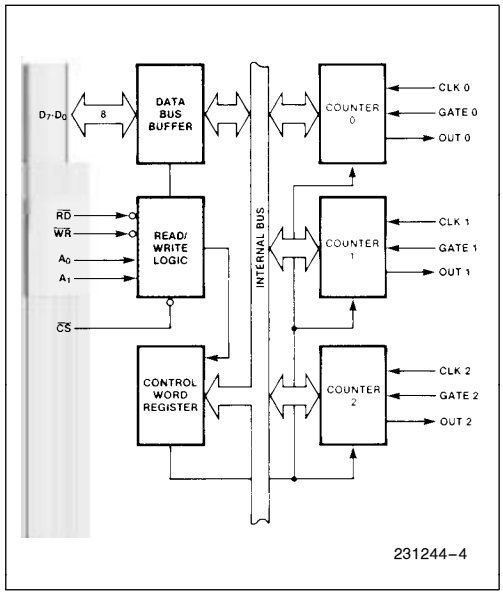


Figure 3. Block Diagram Showing Data Bus Buffer and Read/Write Logic Functions

READ/WRITE LOGIC

The Read/Write Logic accepts inputs from the system bus and generates control signals for the other functional blocks of the 82C54. A_1 and A_0 select one of the three counters or the Control Word Register to be read from/written into. A "low" on the $\overline{RD}$ input tells the 82C54 that the CPU is reading one of the counters. A "low" on the $\overline{WR}$ input tells the 82C54 that the CPU is writing either a Control Word or an initial count. Both $\overline{RD}$ and $\overline{WR}$ are qualified by $\overline{CS}$; $\overline{RD}$ and $\overline{WR}$ are ignored unless the 82C54 has been selected by holding $\overline{CS}$ low.

The WR# and CLK signals should be synchronous. This is accomplished by using a CLK input signal to the 82C54 counters which is a derivative of the system clock source. Another technique is to externally synchronize the WR# and CLK input signals. This is done by gating WR# with CLK.

CONTROL WORD REGISTER

The Control Word Register (see Figure 4) is selected by the Read/Write Logic when A_1, A_0 = 11. If the CPU then does a write operation to the 82C54, the data is stored in the Control Word Register and is interpreted as a Control Word used to define the operation of the Counters.

The Control Word Register can only be written to; status information is available with the Read-Back Command.

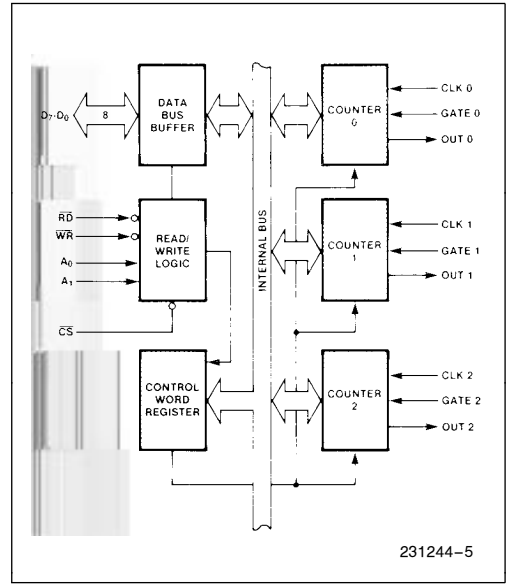


Figure 4. Block Diagram Showing Control Word Register and Counter Functions

COUNTER 0, COUNTER 1, COUNTER 2

These three functional blocks are identical in operation, so only a single Counter will be described. The internal block diagram of a single counter is shown in Figure 5.

The Counters are fully independent. Each Counter may operate in a different Mode.

The Control Word Register is shown in the figure; it is not part of the Counter itself, but its contents determine how the Counter operates.

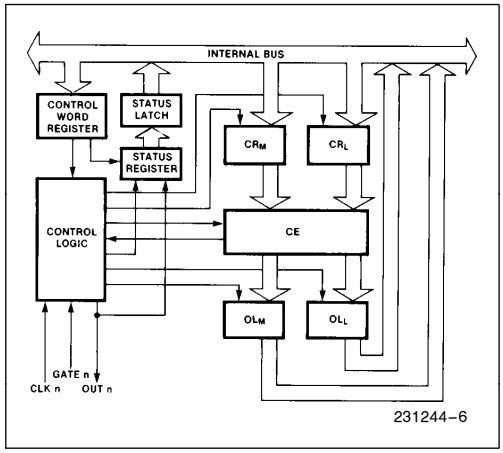


Figure 5. Internal Block Diagram of a Counter

The status register, shown in the Figure, when latched, contains the current contents of the Control Word Register and status of the output and null count flag. (See detailed explanation of the Read-Back command.)

The actual counter is labelled CE (for ''Counting Element''). It is a 16-bit presettable synchronous down counter.

OL_M and OL_L are two 8-bit latches. OL stands for ''Output Latch''; the subscripts M and L stand for ''Most significant byte'' and ''Least significant byte'' respectively. Both are normally referred to as one unit and called just OL. These latches normally ''follow'' the CE, but if a suitable Counter Latch Command is sent to the 82C54, the latches ''latch'' the present count until read by the CPU and then return to ''following'' the CE. One latch at a time is enabled by the counter's Control Logic to drive the internal bus. This is how the 16-bit Counter communicates over the 8-bit internal bus. Note that the CE itself cannot be read; whenever you read the count, it is the OL that is being read.

Similarly, there are two 8-bit registers called CR_M and CR_L (for ''Count Register''). Both are normally referred to as one unit and called just CR. When a new count is written to the Counter, the count is stored in the CR and later transferred to the CE. The Control Logic allows one register at a time to be loaded from the internal bus. Both bytes are transferred to the CE simultaneously. CR_M and CR_L are cleared when the Counter is programmed. In this way, if the Counter has been programmed for one byte counts (either most significant byte only or least significant byte only) the other byte will be zero. Note that the CE cannot be written into; whenever a count is written, it is written into the CR.

The Control Logic is also shown in the diagram. CLK n, GATE n, and OUT n are all connected to the outside world through the Control Logic.

82C54 SYSTEM INTERFACE

The 82C54 is treated by the systems software as an array of peripheral I/O ports; three are counters and the fourth is a control register for MODE programming.

Basically, the select inputs A_0, A_1 connect to the A_0, A_1 address bus signals of the CPU. The $\overline{CS}$ can be derived directly from the address bus using a linear select method. Or it can be connected to the output of a decoder, such as an Intel 8205 for larger systems.

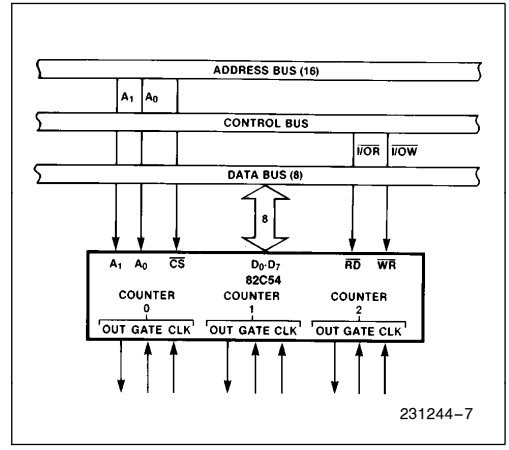


Figure 6. 82C54 System Interface

OPERATIONAL DESCRIPTION

General

After power-up, the state of the 82C54 is undefined. The Mode, count value, and output of all Counters are undefined.

How each Counter operates is determined when it is programmed. Each Counter must be programmed before it can be used. Unused counters need not be programmed.

Programming the 82C54

Counters are programmed by writing a Control Word and then an initial count. The control word format is shown in Figure 7.

All Control Words are written into the Control Word Register, which is selected when A_1, A_0 = 11. The Control Word itself specifies which Counter is being programmed.

By contrast, initial counts are written into the Counters, not the Control Word Register. The A_1, A_0 inputs are used to select the Counter to be written into. The format of the initial count is determined by the Control Word used.

Control Word Format

A_1, A_0 = 11 $\overline{CS}$ = 0 $\overline{RD}$ = 1 $\overline{WR}$ = 0

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
SC1	SC0	RW1	RW0	M2	M1	M0	BCD

SC — Select Counter:

SC1	SC0	
0	0	Select Counter 0
0	1	Select Counter 1
1	0	Select Counter 2
1	1	Read-Back Command (See Read Operations)

RW — Read/Write:

RW1	RW0	
0	0	Counter Latch Command (see Read Operations)
0	1	Read/Write least significant byte only.
1	0	Read/Write most significant byte only.
1	1	Read/Write least significant byte first, then most significant byte.

NOTE: Don't care bits (X) should be 0 to insure compatibility with future Intel products.

M — MODE:

M2	M1	M0	
0	0	0	Mode 0
0	0	1	Mode 1
X	1	0	Mode 2
X	1	1	Mode 3
1	0	0	Mode 4
1	0	1	Mode 5

BCD:

0	Binary Counter 16-bits
1	Binary Coded Decimal (BCD) Counter (4 Decades)

Figure 7. Control Word Format

Write Operations

The programming procedure for the 82C54 is very flexible. Only two conventions need to be remembered:

1) For each Counter, the Control Word must be written before the initial count is written.
2) The initial count must follow the count format specified in the Control Word (least significant byte only, most significant byte only, or least significant byte and then most significant byte).

Since the Control Word Register and the three Counters have separate addresses (selected by the A_1, A_0 inputs), and each Control Word specifies the Counter it applies to (SC0, SC1 bits), no special instruction sequence is required. Any programming sequence that follows the conventions above is acceptable.

A new initial count may be written to a Counter at any time without affecting the Counter's programmed Mode in any way. Counting will be affected as described in the Mode definitions. The new count must follow the programmed count format.

If a Counter is programmed to read/write two-byte counts, the following precaution applies: A program must not transfer control between writing the first and second byte to another routine which also writes into that same Counter. Otherwise, the Counter will be loaded with an incorrect count.

		A_1	A_0
Control Word —	Counter 0	1	1
LSB of count —	Counter 0	0	0
MSB of count —	Counter 0	0	0
Control Word —	Counter 1	1	1
LSB of count —	Counter 1	0	1
MSB of count —	Counter 1	0	1
Control Word —	Counter 2	1	1
LSB of count —	Counter 2	1	0
MSB of count —	Counter 2	1	0

		A_1	A_0
Control Word —	Counter 2	1	1
Control Word —	Counter 1	1	1
Control Word —	Counter 0	1	1
LSB of count —	Counter 2	1	0
MSB of count —	Counter 2	1	0
LSB of count —	Counter 1	0	1
MSB of count —	Counter 1	0	1
LSB of count —	Counter 0	0	0
MSB of count —	Counter 0	0	0

		A_1	A_0
Control Word —	Counter 0	1	1
Counter Word —	Counter 1	1	1
Control Word —	Counter 2	1	1
LSB of count —	Counter 2	1	0
LSB of count —	Counter 1	0	1
LSB of count —	Counter 0	0	0
MSB of count —	Counter 0	0	0
MSB of count —	Counter 1	0	1
MSB of count —	Counter 2	1	0

		A_1	A_0
Control Word —	Counter 1	1	1
Control Word —	Counter 0	1	1
LSB of count —	Counter 1	0	1
Control Word —	Counter 2	1	1
LSB of count —	Counter 0	0	0
MSB of count —	Counter 1	0	1
LSB of count —	Counter 2	1	0
MSB of count —	Counter 0	0	0
MSB of count —	Counter 2	1	0

NOTE:
In all four examples, all counters are programmed to read/write two-byte counts.
These are only four of many possible programming sequences.

Figure 8. A Few Possible Programming Sequences

Read Operations

It is often desirable to read the value of a Counter without disturbing the count in progress. This is easily done in the 82C54.

There are three possible methods for reading the counters: a simple read operation, the Counter Latch Command, and the Read-Back Command. Each is explained below. The first method is to perform a simple read operation. To read the Counter, which is selected with the A1, A0 inputs, the CLK input of the selected Counter must be inhibited by using either the GATE input or external logic. Otherwise, the count may be in the process of changing when it is read, giving an undefined result.

COUNTER LATCH COMMAND

The second method uses the ''Counter Latch Command''. Like a Control Word, this command is written to the Control Word Register, which is selected when A_1, A_0 = 11. Also like a Control Word, the SC0, SC1 bits select one of the three Counters, but two other bits, D5 and D4, distinguish this command from a Control Word.

A_1, A_0 = 11; $\overline{CS}$ = 0; $\overline{RD}$ = 1; $\overline{WR}$ = 0

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
SC1	SC0	0	0	X	X	X	X

SC1, SC0 - specify counter to be latched

SC1	SC0	Counter
0	0	0
0	1	1
1	0	2
1	1	Read-Back Command

D5,D4 - 00 designates Counter Latch Command

X - don't care

NOTE:
Don't care bits (X) should be 0 to insure compatibility with future Intel products.

Figure 9. Counter Latching Command Format

The selected Counter's output latch (OL) latches the count at the time the Counter Latch Command is received. This count is held in the latch until it is read by the CPU (or until the Counter is reprogrammed). The count is then unlatched automatically and the OL returns to ''following'' the counting element (CE). This allows reading the contents of the Counters ''on the fly'' without affecting counting in progress. Multiple Counter Latch Commands may be used to latch more than one Counter. Each latched Counter's OL holds its count until it is read. Counter Latch Commands do not affect the programmed Mode of the Counter in any way.

If a Counter is latched and then, some time later, latched again before the count is read, the second Counter Latch Command is ignored. The count read will be the count at the time the first Counter Latch Command was issued.

With either method, the count must be read according to the programmed format; specifically, if the Counter is programmed for two byte counts, two bytes must be read. The two bytes do not have to be read one right after the other; read or write or programming operations of other Counters may be inserted between them.

Another feature of the 82C54 is that reads and writes of the same Counter may be interleaved; for example, if the Counter is programmed for two byte counts, the following sequence is valid.

1. Read least significant byte.
2. Write new least significant byte.
3. Read most significant byte.
4. Write new most significant byte.

If a Counter is programmed to read/write two-byte counts, the following precaution applies; A program must not transfer control between reading the first and second byte to another routine which also reads from that same Counter. Otherwise, an incorrect count will be read.

READ-BACK COMMAND

The third method uses the Read-Back command. This command allows the user to check the count value, programmed Mode, and current state of the OUT pin and Null Count flag of the selected counter(s).

The command is written into the Control Word Register and has the format shown in Figure 10. The command applies to the counters selected by setting their corresponding bits D3,D2,D1 = 1.

A0, A1 = 11 $\overline{CS}$ = 0 $\overline{RD}$ = 1 $\overline{WR}$ = 0

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
1	1	$\overline{COUNT}$	$\overline{STATUS}$	CNT 2	CNT 1	CNT 0	0

D_5: 0 = Latch count of selected counter(s)
D_4: 0 = Latch status of selected counter(s)
D_3: 1 = Select counter 2
D_2: 1 = Select counter 1
D_1: 1 = Select counter 0
D_0: Reserved for future expansion; must be 0

Figure 10. Read-Back Command Format

The read-back command may be used to latch multiple counter output latches (OL) by setting the $\overline{COUNT}$ bit D5 = 0 and selecting the desired counter(s). This single command is functionally equivalent to several counter latch commands, one for each counter latched. Each counter's latched count is held until it is read (or the counter is reprogrammed). That counter is automatically unlatched when read, but other counters remain latched until they are read. If multiple count read-back commands are issued to the same counter without reading the

count, all but the first are ignored; i.e., the count which will be read is the count at the time the first read-back command was issued.

The read-back command may also be used to latch status information of selected counter(s) by setting $\overline{\text{STATUS}}$ bit D4 = 0. Status must be latched to be read; status of a counter is accessed by a read from that counter.

The counter status format is shown in Figure 11. Bits D5 through D0 contain the counter's programmed Mode exactly as written in the last Mode Control Word. OUTPUT bit D7 contains the current state of the OUT pin. This allows the user to monitor the counter's output via software, possibly eliminating some hardware from a system.

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
OUTPUT	NULL COUNT	RW1	RW0	M2	M1	M0	BCD

D_7 1 = Out Pin is 1
0 = Out Pin is 0
D_6 1 = Null count
0 = Count available for reading
D_5-D_0 Counter Programmed Mode (See Figure 7)

Figure 11. Status Byte

NULL COUNT bit D6 indicates when the last count written to the counter register (CR) has been loaded into the counting element (CE). The exact time this happens depends on the Mode of the counter and is described in the Mode Definitions, but until the count is loaded into the counting element (CE), it can't be read from the counter. If the count is latched or read before this time, the count value will not reflect the new count just written. The operation of Null Count is shown in Figure 12.

THIS ACTION:	CAUSES:
A. Write to the control word register:[1]	Null count = 1
B. Write to the count register (CR);[2]	Null count = 1
C. New count is loaded into CE (CR → CE);	Null count = 0

[1] Only the counter specified by the control word will have its null count set to 1. Null count bits of other counters are unaffected.

[2] If the counter is programmed for two-byte counts (least significant byte then most significant byte) null count goes to 1 when the second byte is written.

Figure 12. Null Count Operation

If multiple status latch operations of the counter(s) are performed without reading the status, all but the first are ignored; i.e., the status that will be read is the status of the counter at the time the first status read-back command was issued.

Both count and status of the selected counter(s) may be latched simultaneously by setting both $\overline{\text{COUNT}}$ and $\overline{\text{STATUS}}$ bits D5,D4 = 0. This is functionally the same as issuing two separate read-back commands at once, and the above discussions apply here also. Specifically, if multiple count and/or status read-back commands are issued to the same counter(s) without any intervening reads, all but the first are ignored. This is illustrated in Figure 13.

If both count and status of a counter are latched, the first read operation of that counter will return latched status, regardless of which was latched first. The next one or two reads (depending on whether the counter is programmed for one or two type counts) return latched count. Subsequent reads return unlatched count.

Command								Description	Results
D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0		
1	1	0	0	0	0	1	0	Read back count and status of Counter 0	Count and status latched for Counter 0
1	1	1	0	0	1	0	0	Read back status of Counter 1	Status latched for Counter 1
1	1	1	0	1	1	0	0	Read back status of Counters 2, 1	Status latched for Counter 2, but not Counter 1
1	1	0	1	1	0	0	0	Read back count of Counter 2	Count latched for Counter 2
1	1	0	0	0	1	0	0	Read back count and status of Counter 1	Count latched for Counter 1, but not status
1	1	1	0	0	0	1	0	Read back status of Counter 1	Command ignored, status already latched for Counter 1

Figure 13. Read-Back Command Example

$\overline{CS}$	$\overline{RD}$	$\overline{WR}$	A_1	A_0	
0	1	0	0	0	Write into Counter 0
0	1	0	0	1	Write into Counter 1
0	1	0	1	0	Write into Counter 2
0	1	0	1	1	Write Control Word
0	0	1	0	0	Read from Counter 0
0	0	1	0	1	Read from Counter 1
0	0	1	1	0	Read from Counter 2
0	0	1	1	1	No-Operation (3-State)
1	X	X	X	X	No-Operation (3-State)
0	1	1	X	X	No-Operation (3-State)

Figure 14. Read/Write Operations Summary

Mode Definitions

The following are defined for use in describing the operation of the 82C54.

CLK PULSE: a rising edge, then a falling edge, in that order, of a Counter's CLK input.

TRIGGER: a rising edge of a Counter's GATE input.

COUNTER LOADING: the transfer of a count from the CR to the CE (refer to the ''Functional Description'')

MODE 0: INTERRUPT ON TERMINAL COUNT

Mode 0 is typically used for event counting. After the Control Word is written, OUT is initially low, and will remain low until the Counter reaches zero. OUT then goes high and remains high until a new count or a new Mode 0 Control Word is written into the Counter.

GATE = 1 enables counting; GATE = 0 disables counting. GATE has no effect on OUT.

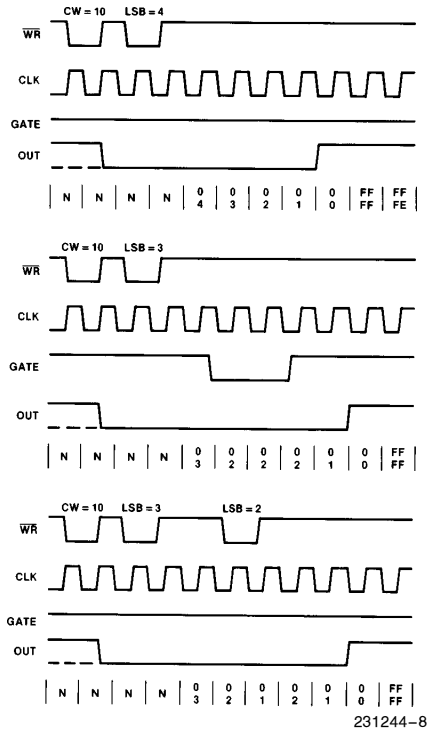
After the Control Word and initial count are written to a Counter, the initial count will be loaded on the next CLK pulse. This CLK pulse does not decrement the count, so for an initial count of N, OUT does not go high until N + 1 CLK pulses after the initial count is written.

If a new count is written to the Counter, it will be loaded on the next CLK pulse and counting will continue from the new count. If a two-byte count is written, the following happens:

1) Writing the first byte does not disable counting. OUT is set low immediately (no clock pulse required).
2) Writing the second byte allows the new count to be loaded on the next CLK pulse.
3) When there is a count in progress, writing a new LSB before the counter has counted down to 0 and rolled over to FFFFh, WILL stop the counter. However, if the LSB is loaded AFTER the counter has rolled over to FFFFh, so that an MSB now exists in the counter, then the counter WILL NOT stop.

This allows the counting sequence to be synchronized by software. Again, OUT does not go high until N + 1 CLK pulses after the new count of N is written.

If an initial count is written while GATE = 0, it will still be loaded on the next CLK pulse. When GATE goes high, OUT will go high N CLK pulses later; no CLK pulse is needed to load the Counter as this has already been done.



NOTE:
The Following Conventions Apply To All Mode Timing Diagrams:
1. Counters are programmed for binary (not BCD) counting and for Reading/Writing least significant byte (LSB) only.
2. The counter is always selected ($\overline{CS}$ always low).
3. CW stands for "Control Word"; CW = 10 means a control word of 10, hex is written to the counter.
4. LSB stands for "Least Significant Byte" of count.
5. Numbers below diagrams are count values.
The lower number is the least significant byte.
The upper number is the most significant byte. Since the counter is programmed to Read/Write LSB only, the most significant byte cannot be read.
N stands for an undefined count.
Vertical lines show transitions between count values.

Figure 15. Mode 0

MODE 1: HARDWARE RETRIGGERABLE ONE-SHOT

OUT will be initially high. OUT will go low on the CLK pulse following a trigger to begin the one-shot pulse, and will remain low until the Counter reaches zero. OUT will then go high and remain high until the CLK pulse after the next trigger.

After writing the Control Word and initial count, the Counter is armed. A trigger results in loading the Counter and setting OUT low on the next CLK pulse, thus starting the one-shot pulse. An initial count of N will result in a one-shot pulse N CLK cycles in duration. The one-shot is retriggerable, hence OUT will remain low for N CLK pulses after any trigger. The one-shot pulse can be repeated without rewriting the same count into the counter. GATE has no effect on OUT.

If a new count is written to the Counter during a one-shot pulse, the current one-shot is not affected unless the Counter is retriggered. In that case, the Counter is loaded with the new count and the one-shot pulse continues until the new count expires.

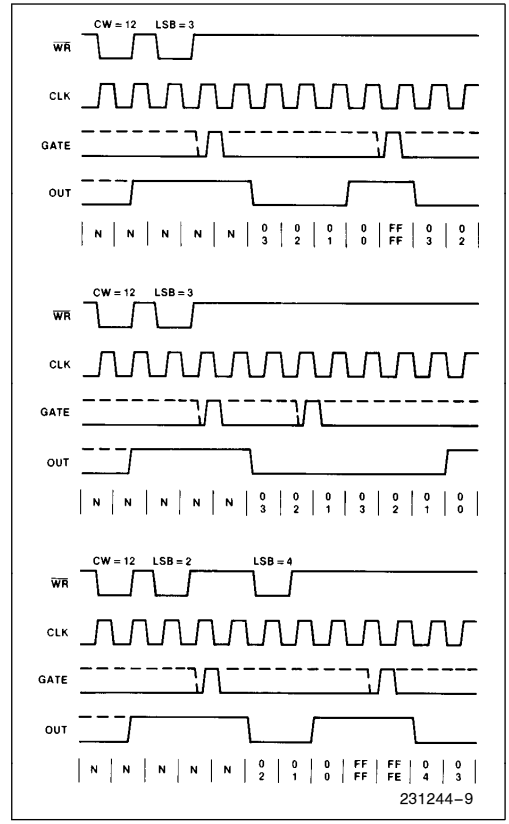


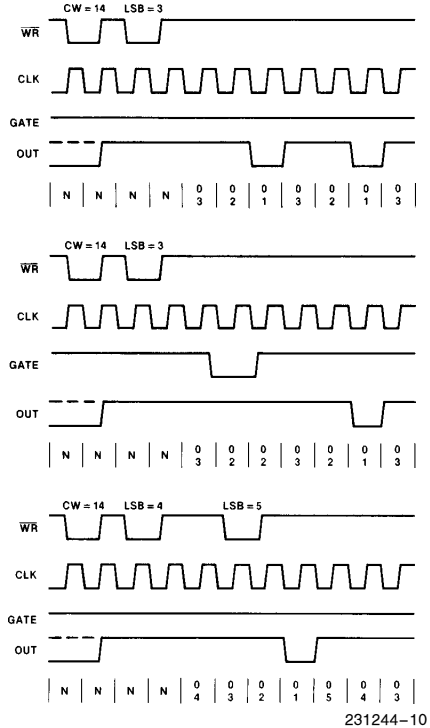
Figure 16. Mode 1

MODE 2: RATE GENERATOR

This Mode functions like a divide-by-N counter. It is typicially used to generate a Real Time Clock interrupt. OUT will initially be high. When the initial count has decremented to 1, OUT goes low for one CLK pulse. OUT then goes high again, the Counter reloads the initial count and the process is repeated. Mode 2 is periodic; the same sequence is repeated indefinitely. For an initial count of N, the sequence repeats every N CLK cycles.

GATE = 1 enables counting; GATE = 0 disables counting. If GATE goes low during an output pulse, OUT is set high immediately. A trigger reloads the Counter with the initial count on the next CLK pulse; OUT goes low N CLK pulses after the trigger. Thus the GATE input can be used to synchronize the Counter.

After writing a Control Word and initial count, the Counter will be loaded on the next CLK pulse. OUT goes low N CLK Pulses after the initial count is written. This allows the Counter to be synchronized by software also.



NOTE:
A GATE transition should not occur one clock prior to terminal count.

Figure 17. Mode 2

Writing a new count while counting does not affect the current counting sequence. If a trigger is received after writing a new count but before the end of the current period, the Counter will be loaded with the new count on the next CLK pulse and counting will continue from the new count. Otherwise, the new count will be loaded at the end of the current counting cycle. In mode 2, a COUNT of 1 is illegal.

MODE 3: SQUARE WAVE MODE

Mode 3 is typically used for Baud rate generation. Mode 3 is similar to Mode 2 except for the duty cycle of OUT. OUT will initially be high. When half the initial count has expired, OUT goes low for the remainder of the count. Mode 3 is periodic; the sequence above is repeated indefinitely. An initial count of N results in a square wave with a period of N CLK cycles.

GATE = 1 enables counting; GATE = 0 disables counting. If GATE goes low while OUT is low, OUT is set high immediately; no CLK pulse is required. A trigger reloads the Counter with the initial count on the next CLK pulse. Thus the GATE input can be used to synchronize the Counter.

After writing a Control Word and initial count, the Counter will be loaded on the next CLK pulse. This allows the Counter to be synchronized by software also.

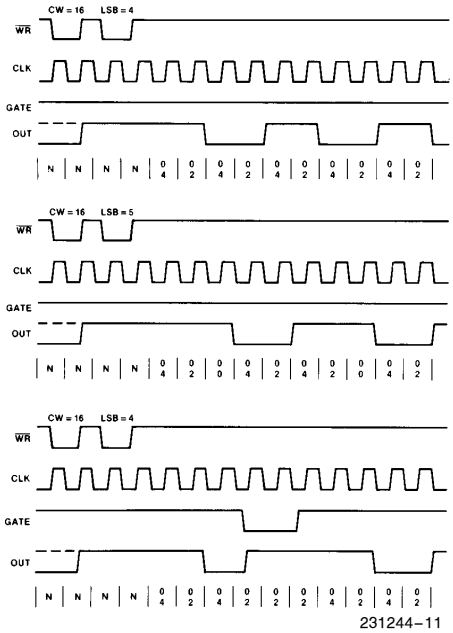
Writing a new count while counting does not affect the current counting sequence. If a trigger is received after writing a new count but before the end of the current half-cycle of the square wave, the Counter will be loaded with the new count on the next CLK pulse and counting will continue from the new count. Otherwise, the new count will be loaded at the end of the current half-cycle.

Mode 3 is implemented as follows:

Even counts: OUT is initially high. The initial count is loaded on one CLK pulse and then is decremented by two on succeeding CLK pulses. When the count expires OUT changes value and the Counter is reloaded with the initial count. The above process is repeated indefinitely.

Odd counts: OUT is initially high. The initial count minus one (an even number) is loaded on one CLK pulse and then is decremented by two on succeeding CLK pulses. One CLK pulse *after* the count expires, OUT goes low and the Counter is reloaded with the initial count minus one. Succeeding CLK pulses decrement the count by two. When the count expires, OUT goes high again and the Counter is reloaded with the initial count minus one. The above process is repeated indefinitely. So for odd counts,

OUT will be high for (N + 1)/2 counts and low for (N −1)/2 counts.



NOTE:
A GATE transition should not occur one clock prior to terminal count.

Figure 18. Mode 3

MODE 4: SOFTWARE TRIGGERED STROBE

OUT will be initially high. When the initial count expires, OUT will go low for one CLK pulse and then go high again. The counting sequence is "triggered" by writing the initial count.

GATE = 1 enables counting; GATE = 0 disables counting. GATE has no effect on OUT.

After writing a Control Word and initial count, the Counter will be loaded on the next CLK pulse. This CLK pulse does not decrement the count, so for an initial count of N, OUT does not strobe low until N + 1 CLK pulses after the initial count is written.

If a new count is written during counting, it will be loaded on the next CLK pulse and counting will continue from the new count. If a two-byte count is written, the following happens:

1) Writing the first byte has no effect on counting.
2) Writing the second byte allows the new count to be loaded on the next CLK pulse.

This allows the sequence to be "retriggered" by software. OUT strobes low N+1 CLK pulses after the new count of N is written.

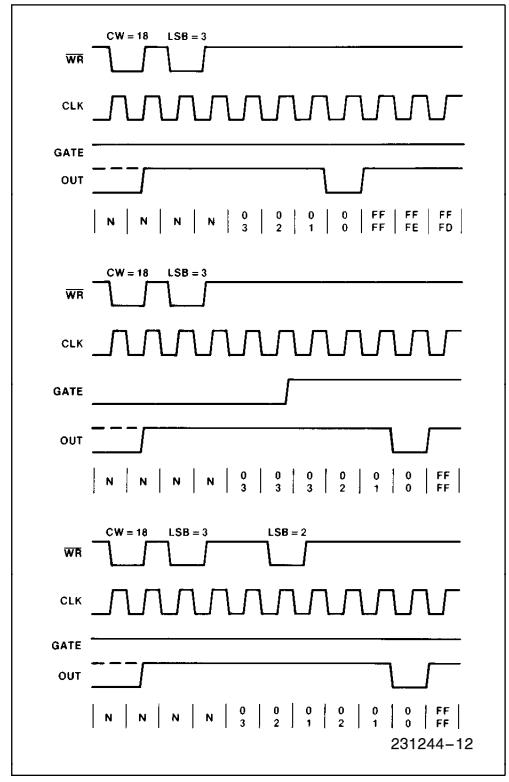


Figure 19. Mode 4

MODE 5: HARDWARE TRIGGERED STROBE (RETRIGGERABLE)

OUT will initially be high. Counting is triggered by a rising edge of GATE. When the initial count has expired, OUT will go low for one CLK pulse and then go high again.

After writing the Control Word and initial count, the counter will not be loaded until the CLK pulse after a trigger. This CLK pulse does not decrement the count, so for an initial count of N, OUT does not strobe low until N + 1 CLK pulses after a trigger.

A trigger results in the Counter being loaded with the initial count on the next CLK pulse. The counting sequence is retriggerable. OUT will not strobe low for N + 1 CLK pulses after any trigger. GATE has no effect on OUT.

If a new count is written during counting, the current counting sequence will not be affected. If a trigger occurs after the new count is written but before the current count expires, the Counter will be loaded with the new count on the next CLK pulse and counting will continue from there.

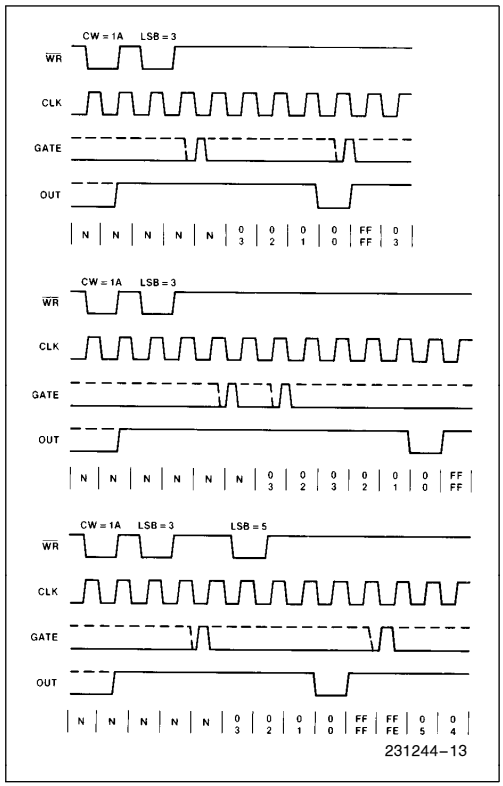


Figure 20. Mode 5

Signal Status Modes	Low Or Going Low	Rising	High
0	Disables counting	—	Enables counting
1	—	1) Initiates counting 2) Resets output after next clock	—
2	1) Disables counting 2) Sets output immediately high	Initiates counting	Enables counting
3	1) Disables counting 2) Sets output immediately high	Initiates counting	Enables counting
4	Disables counting	—	Enables counting
5	—	Initiates counting	—

Figure 21. Gate Pin Operations Summary

MODE	MIN COUNT	MAX COUNT
0	1	0
1	1	0
2	2	0
3	2	0
4	1	0

NOTE:
0 is equivalent to 2^{16} for binary counting and 10^4 for BCD counting

Figure 22. Minimum and Maximum initial Counts

Operation Common to All Modes

Programming

When a Control Word is written to a Counter, all Control Logic is immediately reset and OUT goes to a known initial state; no CLK pulses are required for this.

GATE

The GATE input is always sampled on the rising edge of CLK. In Modes 0, 2, 3, and 4 the GATE input is level sensitive, and the logic level is sampled on the rising edge of CLK. In Modes 1, 2, 3, and 5 the GATE input is rising-edge sensitive. In these Modes, a rising edge of GATE (trigger) sets an edge-sensitive flip-flop in the Counter. This flip-flop is then sampled on the next rising edge of CLK; the flip-flop is reset immediately after it is sampled. In this way, a trigger will be detected no matter when it occurs—a high logic level does not have to be maintained until the next rising edge of CLK. Note that in Modes 2 and 3, the GATE input is both edge- and level-sensitive. In Modes 2 and 3, if a CLK source other than the system clock is used, GATE should be pulsed immediately following $\overline{WR}$ of a new count value.

COUNTER

New counts are loaded and Counters are decremented on the falling edge of CLK.

The largest possible initial count is 0; this is equivalent to 2^{16} for binary counting and 10^4 for BCD counting.

The Counter does not stop when it reaches zero. In Modes 0, 1, 4, and 5 the Counter ‘‘wraps around’’ to the highest count, either FFFF hex for binary counting or 9999 for BCD counting, and continues counting. Modes 2 and 3 are periodic; the Counter reloads itself with the initial count and continues counting from there.

ABSOLUTE MAXIMUM RATINGS*

Ambient Temperature Under Bias. . . . . 0°C to 70°C
Storage Temperature . . . . . −65° to +150°C
Supply Voltage . . . . . −0.5 to +8.0V
Operating Voltage . . . . . +4V to +7V
Voltage on any Input . . . . . GND −2V to +6.5V
Voltage on any Output . . GND−0.5V to V_{CC} + 0.5V
Power Dissipation . . . . . 1 Watt

NOTICE: This is a production data sheet. The specifications are subject to change without notice.

**WARNING: Stressing the device beyond the "Absolute Maximum Ratings" may cause permanent damage. These are stress ratings only. Operation beyond the "Operating Conditions" is not recommended and extended exposure beyond the "Operating Conditions" may affect device reliability.*

D.C. CHARACTERISTICS

(T_A=0°C to 70°C, V_{CC}=5V± 10%, GND=0V) (T_A = −40°C to +85°C for Extended Temperature)

Symbol	Parameter	Min	Max	Units	Test Conditions
V_{IL}	Input Low Voltage	−0.5	0.8	V	
V_{IH}	Input High Voltage	2.0	V_{CC} + 0.5	V	
V_{OL}	Output Low Voltage		0.4	V	I_{OL} = 2.5 mA
V_{OH}	Output High Voltage	3.0 V_{CC} − 0.4		V V	I_{OH} = −2.5 mA I_{OH} = −100 μA
I_{IL}	Input Load Current		±2.0	μA	V_{IN}=V_{CC} to 0V
I_{OFL}	Output Float Leakage Current		±10	μA	V_{OUT}=V_{CC} to 0.0V
I_{CC}	V_{CC} Supply Current		20	mA	Clk Freq= 8MHz 82C54 10MHz 82C54-2
I_{CCSB}	V_{CC} Supply Current-Standby		10	μA	CLK Freq = DC $\overline{CS}$ = V_{CC}. All Inputs/Data Bus V_{CC} All Outputs Floating
I_{CCSB1}	V_{CC} Supply Current-Standby		150	μA	CLK Freq = DC $\overline{CS}$ = V_{CC}. All Other Inputs, I/O Pins = V_{GND}, Outputs Open
C_{IN}	Input Capacitance		10	pF	f_c = 1 MHz
$C_{I/O}$	I/O Capacitance		20	pF	Unmeasured pins returned to GND(5)
C_{OUT}	Output Capacitance		20	pF	

A.C. CHARACTERISTICS

(T_A = 0°C to 70°C, V_{CC} = 5V ±10%, GND =0V) (T_A = −40°C to +85°C for Extended Temperature)

BUS PARAMETERS (Note 1)

READ CYCLE

Symbol	Parameter	82C54-2		Units
		Min	Max	
t_{AR}	Address Stable Before $\overline{RD}$ ↓	30		ns
t_{SR}	$\overline{CS}$ Stable Before $\overline{RD}$ ↓	0		ns
t_{RA}	Address Hold Time After $\overline{RD}$ ↑	0		ns
t_{RR}	$\overline{RD}$ Pulse Width	95		ns
t_{RD}	Data Delay from $\overline{RD}$ ↓		85	ns
t_{AD}	Data Delay from Address		185	ns
t_{DF}	$\overline{RD}$ ↑ to Data Floating	5	65	ns
t_{RV}	Command Recovery Time	165		ns

NOTE:
1. AC timings measured at V_{OH} = 2.0V, V_{OL} = 0.8V.

A.C. CHARACTERISTICS (Continued)

WRITE CYCLE

Symbol	Parameter	82C54-2		Units
		Min	Max	
t_{AW}	Address Stable Before $\overline{WR}\downarrow$	0		ns
t_{SW}	$\overline{CS}$ Stable Before $\overline{WR}\downarrow$	0		ns
t_{WA}	Address Hold Time After $\overline{WR}\uparrow$	0		ns
t_{WW}	$\overline{WR}$ Pulse Width	95		ns
t_{DW}	Data Setup Time Before $\overline{WR}\uparrow$	95		ns
t_{WD}	Data Hold Time After $\overline{WR}\uparrow$	0		ns
t_{RV}	Command Recovery Time	165		ns

CLOCK AND GATE

Symbol	Parameter	82C54-2		Units
		Min	Max	
t_{CLK}	Clock Period	100	DC	ns
t_{PWH}	High Pulse Width	30(3)		ns
t_{PWL}	Low Pulse Width	50(3)		ns
T_R	Clock Rise Time		25	ns
t_F	Clock Fall Time		25	ns
t_{GW}	Gate Width High	50		ns
t_{GL}	Gate Width Low	50		ns
t_{GS}	Gate Setup Time to CLK $\uparrow$	40		ns
t_{GH}	Gate Hold Time After CLK $\uparrow$	50(2)		ns
T_{OD}	Output Delay from CLK $\downarrow$		100	ns
t_{ODG}	Output Delay from Gate $\downarrow$		100	ns
t_{WC}	CLK Delay for Loading(4)	0	55	ns
t_{WG}	Gate Delay for Sampling(4)	−5	40	ns
t_{WO}	OUT Delay from Mode Write		240	ns
t_{CL}	CLK Set Up for Count Latch	−40	40	ns

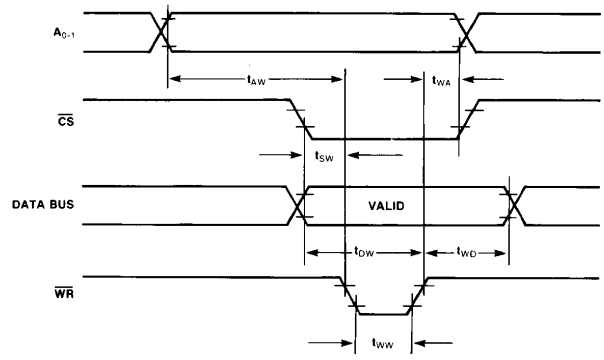
NOTES:
2. In Modes 1 and 5 triggers are sampled on each rising clock edge. A second trigger within 70 ns for the 82C54-2 of the rising clock edge may not be detected.
3. Low-going glitches that violate t_{PWH}, t_{PWL} may cause errors requiring counter reprogramming.
4. Except for Extended Temp., See Extended Temp. A.C. Characteristics below.
5. Sampled not 100% tested. $T_A = 25°C$.
6. If CLK present at T_{WC} min then Count equals N+2 CLK pulses, T_{WC} max equals Count N+1 CLK pulse. T_{WC} min to T_{WC} max, count will be either N+1 or N+2 CLK pulses.
7. In Modes 1 and 5, if GATE is present when writing a new Count value, at T_{WG} min Counter will not be triggered, at T_{WG} max Counter will be triggered.
8. If CLK present when writing a Counter Latch or ReadBack Command, at T_{CL} min CLK will be reflected in count value latched, at T_{CL} max CLK will not be reflected in the count value latched. Writing a Counter Latch or ReadBack Command between T_{CL} min and T_{WL} max will result in a latched count vallue which is ± one least significant bit.

EXTENDED TEMPERATURE ($T_A = -40°C$ to $+85°C$ for Extended Temperature)

Symbol	Parameter	82C54-2		Units
		Min	Max	
t_{WC}	CLK Delay for Loading	−25	25	ns
t_{WG}	Gate Delay for Sampling	−25	25	ns

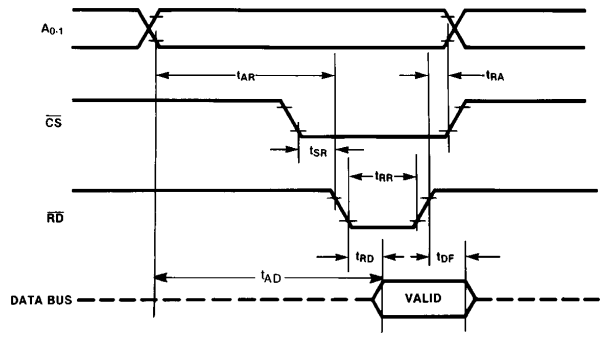
WAVEFORMS

WRITE



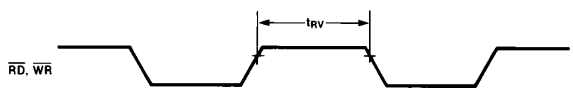
231244-14

READ



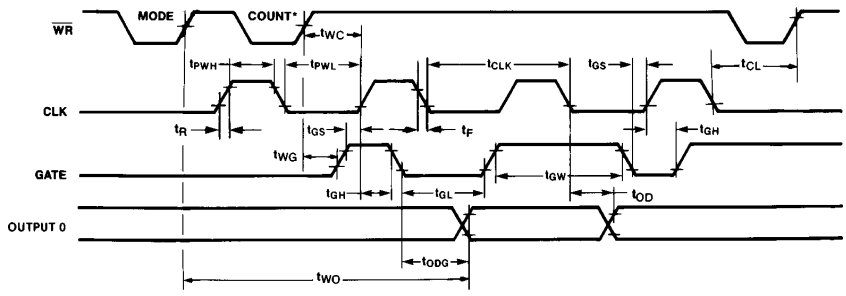
231244-15

RECOVERY



231244-16

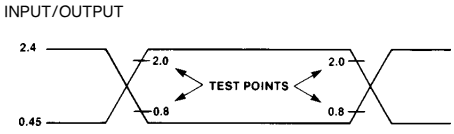
CLOCK AND GATE



231244–17

* Last byte of count being written

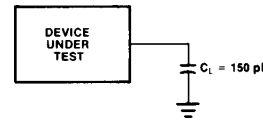
A.C. TESTING INPUT, OUTPUT WAVEFORM



231244–18

A.C. Testing: Inputs are driven at 2.4V for a logic "1" and 0.45V for a logic "0." Timing measurements are made at 2.0V for a logic "1" and 0.8V for a logic "0."

A.C. TESTING LOAD CIRCUIT



231244–19

C_L = 150 pF

C_L includes jig capacitance

REVISION SUMMARY

The following list represents the key differences between Rev. 005 and 006 of the 82C54 Data Sheet.

1. References to and specifications for the 8 MHz 82C54 are removed. Only the 10 MHz 82C52-2 remains in production.

HARRIS Semiconductor

82C54

March 1997

CMOS Programmable Interval Timer

Features

- **8MHz to 12MHz Clock Input Frequency**
- **Compatible with NMOS 8254**
  - **Enhanced Version of NMOS 8253**
- **Three Independent 16-Bit Counters**
- **Six Programmable Counter Modes**
- **Status Read Back Command**
- **Binary or BCD Counting**
- **Fully TTL Compatible**
- **Single 5V Power Supply**
- **Low Power**
  - **ICCSB 10μA**
  - **ICCOP . . . . . .10mA at 8MHz**
- **Operating Temperature Ranges**
  - **C82C54 . . . . . .0°C to +70°C**
  - **I82C54 . . . . . -40°C to +85°C**
  - **M82C54 . . . . . -55°C to +125°C**

Description

The Harris 82C54 is a high performance CMOS Programmable Interval Timer manufactured using an advanced 2 micron CMOS process.

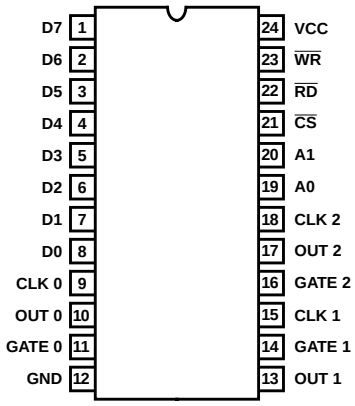
The 82C54 has three independently programmable and functional 16-bit counters, each capable of handling clock input frequencies of up to 8MHz (82C54) or 10MHz (82C54-10) or 12MHz (82C54-12).

The high speed and industry standard configuration of the 82C54 make it compatible with the Harris 80C86, 80C88, and 80C286 CMOS microprocessors along with many other industry standard processors. Six programmable timer modes allow the 82C54 to be used as an event counter, elapsed time indicator, programmable one-shot, and many other applications. Static CMOS circuit design insures low power operation.

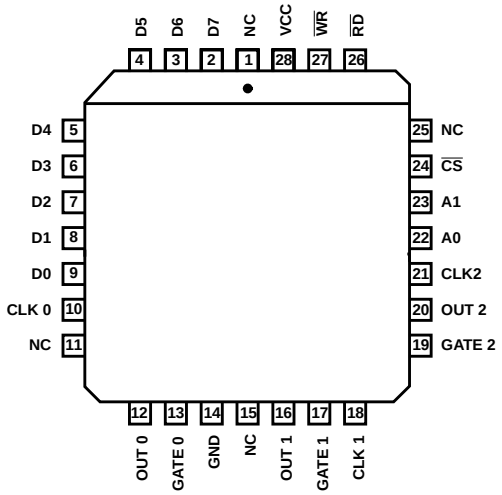
The Harris advanced CMOS process results in a significant reduction in power with performance equal to or greater than existing equivalent products.

Pinouts

82C54 (PDIP, CERDIP, SOIC)
TOP VIEW



82C54 (PLCC/CLCC)
TOP VIEW



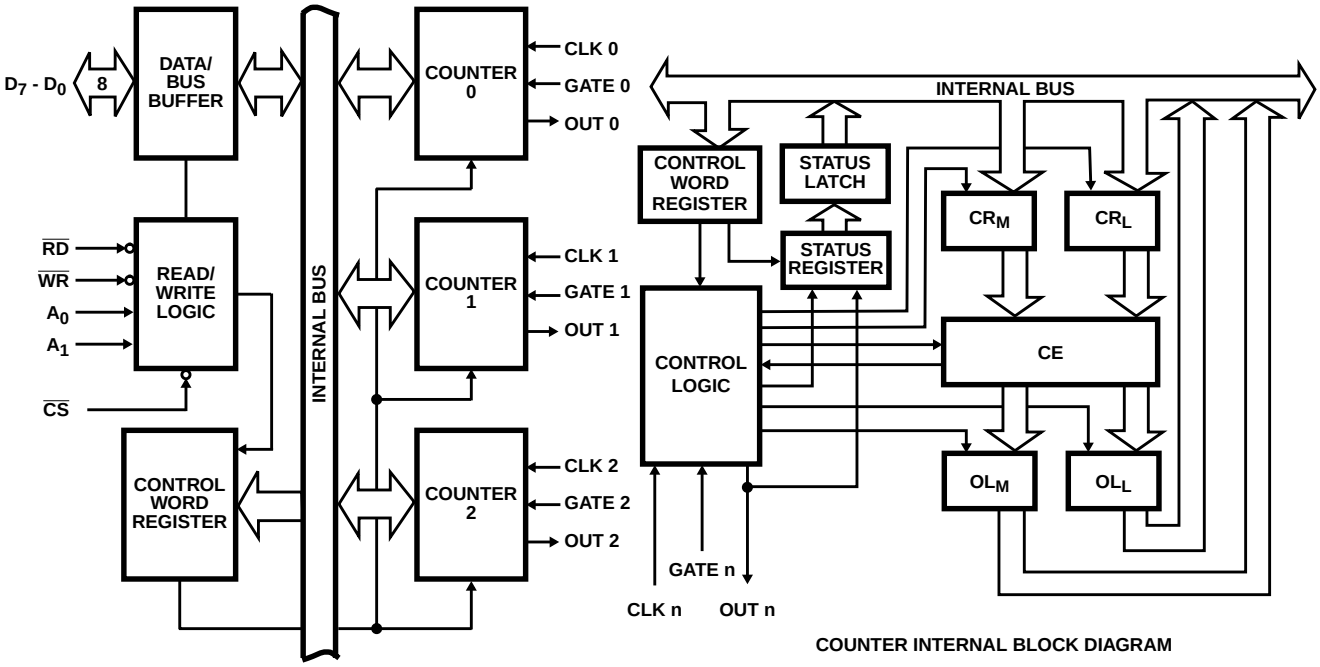
CAUTION: These devices are sensitive to electrostatic discharge. Users should follow proper IC Handling Procedures.

File Number **2970.1**

Ordering Information

PART NUMBERS			TEMPERATURE RANGE	PACKAGE	PKG. NO.
8MHz	10MHz	12MHz			
CP82C54	CP82C54-10	CP82C54-12	0^oC to $+70^oC$	24 Lead PDIP	E24.6
IP82C54	IP82C54-10	IP82C54-12	-40^oC to $+85^oC$	24 Lead PDIP	E24.6
CS82C54	CS82C54-10	CS82C54-12	0^oC to $+70^oC$	28 Lead PLCC	N28.45
IS82C54	IS82C54-10	IS82C54-12	-40^oC to $+85^oC$	28 Lead PLCC	N28.45
CD82C54	CD82C54-10	CD82C54-12	0^oC to $+70^oC$	24 Lead CERDIP	F24.6
ID82C54	ID82C54-10	ID82C54-12	-40^oC to $+85^oC$	24 Lead CERDIP	F24.6
MD82C54/B	MD82C54-10/B	MD82C54-12/B	-55^oC to $+125^oC$	24 Lead CERDIP	F24.6
MR82C54/B	MR82C54-10/B	MR82C54-12/B	-55^oC to $+125^oC$	28 Lead CLCC	J28.A
SMD # 8406501JA	-	8406502JA	-55^oC to $+125^oC$	24 Lead CERDIP	F24.6
SMD# 84065013A	-	84065023A	-55^oC to $+125^oC$	28 Lead CLCC	J28.A
CM82C54	CM82C54-10	CM82C54-12	0^oC to $+70^oC$	24 Lead SOIC	M24.3

Functional Diagram



COUNTER INTERNAL BLOCK DIAGRAM

Pin Description

SYMBOL	DIP PIN NUMBER	TYPE	DEFINITION
D7 - D0	1 - 8	I/O	DATA: Bi-directional three-state data bus lines, connected to system data bus.
CLK 0	9	I	CLOCK 0: Clock input of Counter 0.
OUT 0	10	O	OUT 0: Output of Counter 0.
GATE 0	11	I	GATE 0: Gate input of Counter 0.
GND	12		GROUND: Power supply connection.
OUT 1	13	O	OUT 1: Output of Counter 1.
GATE 1	14	I	GATE 1: Gate input of Counter 1.
CLK 1	15	I	CLOCK 1: Clock input of Counter 1.
GATE 2	16	I	GATE 2: Gate input of Counter 2.
OUT 2	17	O	OUT 2: Output of Counter 2.

Pin Description (Continued)

SYMBOL	DIP PIN NUMBER	TYPE	DEFINITION
CLK 2	18	I	CLOCK 2: Clock input of Counter 2.
A0, A1	19 - 20	I	ADDRESS: Select inputs for one of the three counters or Control Word Register for read/write operations. Normally connected to the system address bus.

A1	A0	SELECTS
0	0	Counter 0
0	1	Counter 1
1	0	Counter 2
1	1	Control Word Register

$\overline{CS}$	21	I	CHIP SELECT: A low on this input enables the 82C54 to respond to $\overline{RD}$ and $\overline{WR}$ signals. $\overline{RD}$ and $\overline{WR}$ are ignored otherwise.
$\overline{RD}$	22	I	READ: This input is low during CPU read operations.
$\overline{WR}$	23	I	WRITE: This input is low during CPU write operations.
V_{CC}	24		V_{CC}: The +5V power supply pin. A 0.1µF capacitor between pins VCC and GND is recommended for decoupling.

Functional Description

General

The 82C54 is a programmable interval timer/counter designed for use with microcomputer systems. It is a general purpose, multi-timing element that can be treated as an array of I/O ports in the system software.

The 82C54 solves one of the most common problems in any microcomputer system, the generation of accurate time delays under software control. Instead of setting up timing loops in software, the programmer configures the 82C54 to match his requirements and programs one of the counters for the desired delay. After the desired delay, the 82C54 will interrupt the CPU. Software overhead is minimal and variable length delays can easily be accommodated.

Some of the other computer/timer functions common to microcomputers which can be implemented with the 82C54 are:

- Real time clock
- Event counter
- Digital one-shot
- Programmable rate generator
- Square wave generator
- Binary rate multiplier
- Complex waveform generator
- Complex motor controller

Data Bus Buffer

This three-state, bi-directional, 8-bit buffer is used to interface the 82C54 to the system bus (see Figure 1).

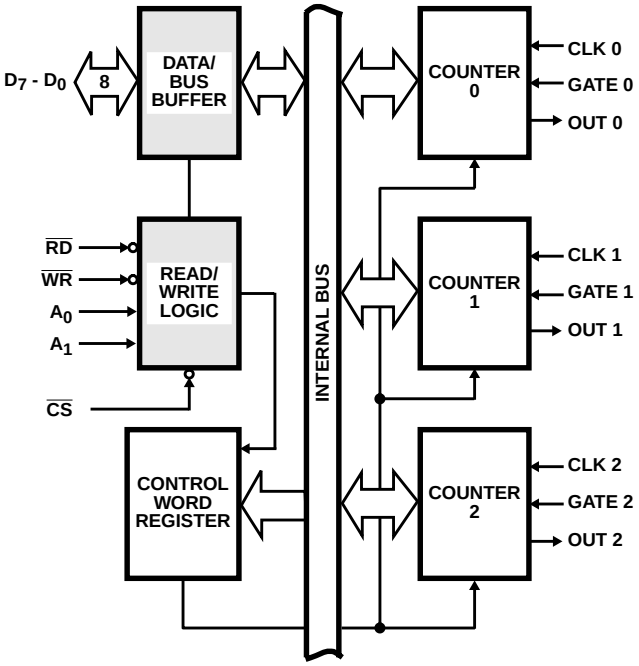


FIGURE 1. DATA BUS BUFFER AND READ/WRITE LOGIC FUNCTIONS

Read/Write Logic

The Read/Write Logic accepts inputs from the system bus and generates control signals for the other functional blocks of the 82C54. A1 and A0 select one of the three counters or the Control Word Register to be read from/written into. A “low” on the $\overline{RD}$ input tells the 82C54 that the CPU is reading one of the counters. A “low” on the $\overline{WR}$ input tells the 82C54 that the CPU is writing either a Control Word or an initial count. Both $\overline{RD}$ and $\overline{WR}$ are qualified by $\overline{CS}$; $\overline{RD}$ and $\overline{WR}$ are ignored unless the 82C54 has been selected by holding $\overline{CS}$ low.

Control Word Register

The Control Word Register (Figure 2) is selected by the Read/Write Logic when A1, A0 = 11. If the CPU then does a write operation to the 82C54, the data is stored in the Control Word Register and is interpreted as a Control Word used to define the Counter operation.

The Control Word Register can only be written to; status information is available with the Read-Back Command.

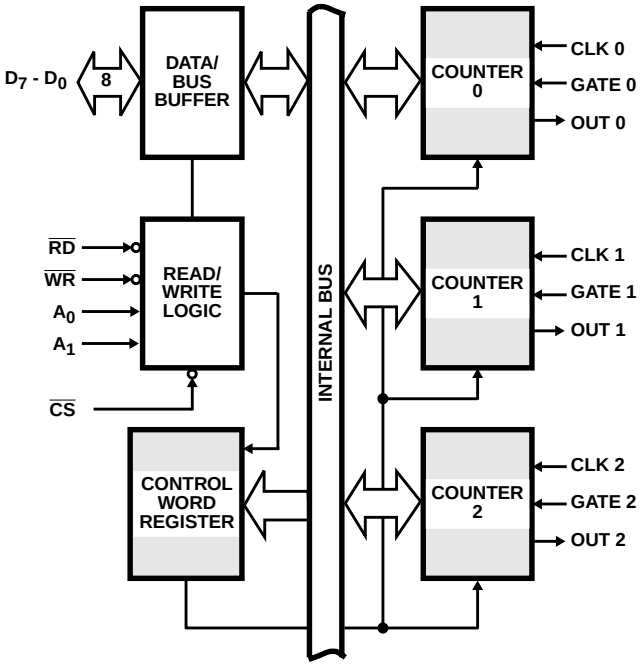


FIGURE 2. CONTROL WORD REGISTER AND COUNTER FUNCTIONS

Counter 0, Counter 1, Counter 2

These three functional blocks are identical in operation, so only a single Counter will be described. The internal block diagram of a signal counter is shown in Figure 3. The counters are fully independent. Each Counter may operate in a different Mode.

The Control Word Register is shown in the figure; it is not part of the Counter itself, but its contents determine how the Counter operates.

The status register, shown in the figure, when latched, contains the current contents of the Control Word Register and status of the output and null count flag. (See detailed explanation of the Read-Back command.)

The actual counter is labeled CE (for Counting Element). It is a 16-bit presettable synchronous down counter.

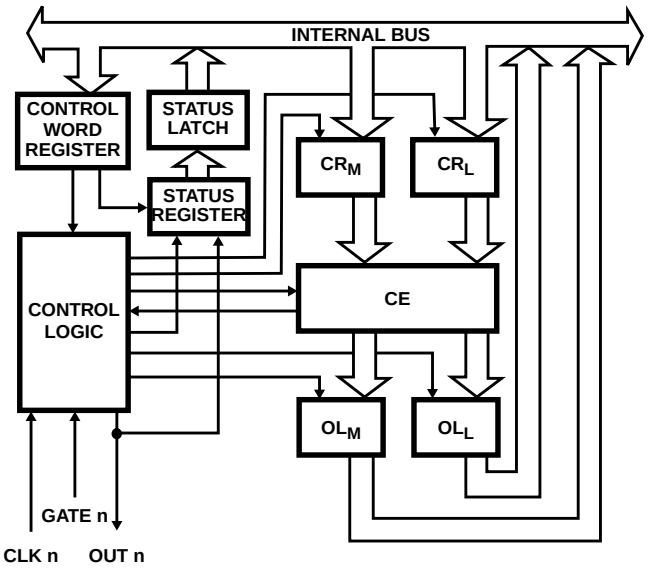


FIGURE 3. COUNTER INTERNAL BLOCK DIAGRAM

OLM and OLL are two 8-bit latches. OL stands for “Output Latch”; the subscripts M and L for “Most significant byte” and “Least significant byte”, respectively. Both are normally referred to as one unit and called just OL. These latches normally “follow” the CE, but if a suitable Counter Latch Command is sent to the 82C54, the latches “latch” the present count until read by the CPU and then return to “following” the CE. One latch at a time is enabled by the counter’s Control Logic to drive the internal bus. This is how the 16-bit Counter communicates over the 8-bit internal bus. Note that the CE itself cannot be read; whenever you read the count, it is the OL that is being read.

Similarly, there are two 8-bit registers called CRM and CRL (for “Count Register”). Both are normally referred to as one unit and called just CR. When a new count is written to the Counter, the count is stored in the CR and later transferred to the CE. The Control Logic allows one register at a time to be loaded from the internal bus. Both bytes are transferred to the CE simultaneously. CRM and CRL are cleared when the Counter is programmed for one byte counts (either most significant byte only or least significant byte only) the other byte will be zero. Note that the CE cannot be written into; whenever a count is written, it is written into the CR.

The Control Logic is also shown in the diagram. CLK n, GATE n, and OUT n are all connected to the outside world through the Control Logic.

82C54 System Interface

The 82C54 is treated by the system software as an array of peripheral I/O ports; three are counters and the fourth is a control register for MODE programming.

Basically, the select inputs A0, A1 connect to the A0, A1 address bus signals of the CPU. The $\overline{CS}$ can be derived directly from the address bus using a linear select method or it can be connected to the output of a decoder.

Operational Description

General

After power-up, the state of the 82C54 is undefined. The Mode, count value, and output of all Counters are undefined.

How each Counter operates is determined when it is programmed. Each Counter must be programmed before it can be used. Unused counters need not be programmed.

Programming the 82C54

Counters are programmed by writing a Control Word and then an initial count.

All Control Words are written into the Control Word Register, which is selected when A1, A0 = 11. The Control Word specifies which Counter is being programmed.

By contrast, initial counts are written into the Counters, not the Control Word Register. The A1, A0 inputs are used to select the Counter to be written into. The format of the initial count is determined by the Control Word used.

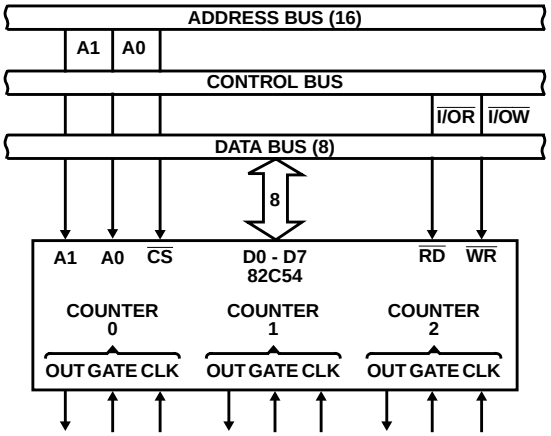


FIGURE 4. 82C54 SYSTEM INTERFACE

Write Operations

The programming procedure for the 82C54 is very flexible. Only two conventions need to be remembered:

1. For Each Counter, the Control Word must be written before the initial count is written.
2. The initial count must follow the count format specified in the Control Word (least significant byte only, most significant byte only, or least significant byte and then most significant byte).

Since the Control Word Register and the three Counters have separate addresses (selected by the A1, A0 inputs), and each Control Word specifies the Counter it applies to (SC0, SC1 bits), no special instruction sequence is required. Any programming sequence that follows the conventions above is acceptable.

Control Word Format

A1, A0 = 11; $\overline{CS}$ = 0; $\overline{RD}$ = 1; $\overline{WR}$ = 0

D7	D6	D5	D4	D3	D2	D1	D0
SC1	SC0	RW1	RW0	M2	M1	M0	BCD

SC - Select Counter

SC1	SC0	
0	0	Select Counter 0
0	1	Select Counter 1
1	0	Select Counter 2
1	1	Read-Back Command (See Read Operations)

RW - Read/Write

RW1	RW0	
0	0	Counter Latch Command (See Read Operations)
0	1	Read/Write least significant byte only.
1	0	Read/Write most significant byte only.
1	1	Read/Write least significant byte first, then most significant byte.

M - Mode

M2	M1	M0	
0	0	0	Mode 0
0	0	1	Mode 1
X	1	0	Mode 2
X	1	1	Mode 3
1	0	0	Mode 4
1	0	1	Mode 5

BCD - Binary Coded Decimal

0	Binary Counter 16-bit
1	Binary Coded Decimal (BCD) Counter (4 Decades)

NOTE: Don't Care bits (X) should be 0 to insure compatibility with future products.

Possible Programming Sequence

	A1	A0
Control Word - Counter 0	1	1
LSB of Count - Counter 0	0	0
MSB of Count - Counter 0	0	0
Control Word - Counter 1	1	1
LSB of Count - Counter 1	0	1
MSB of Count - Counter 1	0	1
Control Word - Counter 2	1	1
LSB of Count - Counter 2	1	0
MSB of Count - Counter 2	1	0

Possible Programming Sequence

	A1	A0
Control Word - Counter 0	1	1
Control Word - Counter 1	1	1
Control Word - Counter 2	1	1
LSB of Count - Counter 2	1	0

Possible Programming Sequence (Continued)

	A1	A0
LSB of Count - Counter 1	0	1
LSB of Count - Counter 0	0	0
MSB of Count - Counter 0	0	0
MSB of Count - Counter 1	0	1
MSB of Count - Counter 2	1	0

Possible Programming Sequence

	A1	A0
Control Word - Counter 2	1	1
Control Word - Counter 1	1	1
Control Word - Counter 0	1	1
LSB of Count - Counter 2	1	0
MSB of Count - Counter 2	1	0
LSB of Count - Counter 1	0	1
MSB of Count - Counter 1	0	1
LSB of Count - Counter 0	0	0
MSB of Count - Counter 0	0	0

Possible Programming Sequence

	A1	A0
Control Word - Counter 1	1	1
Control Word - Counter 0	1	1
LSB of Count - Counter 1	0	1
Control Word - Counter 2	1	1
LSB of Count - Counter 0	0	0
MSB of Count - Counter 1	0	1
LSB of Count - Counter 2	1	0
MSB of Count - Counter 0	0	0
MSB of Count - Counter 2	1	0

NOTE: In all four examples, all counters are programmed to Read/Write two-byte counts. These are only four of many programming sequences.

A new initial count may be written to a Counter at any time without affecting the Counter's programmed Mode in any way. Counting will be affected as described in the Mode definitions. The new count must follow the programmed count format.

If a Counter is programmed to read/write two-byte counts, the following precaution applies. A program must not transfer control between writing the first and second byte to another routine which also writes into that same Counter. Otherwise, the Counter will be loaded with an incorrect count.

Read Operations

It is often desirable to read the value of a Counter without disturbing the count in progress. This is easily done in the 82C54.

There are three possible methods for reading the Counters. The first is through the Read-Back command, which is explained later. The second is a simple read operation of the Counter, which is selected with the A1, A0 inputs. The only requirement is that the CLK input of the selected Counter must be inhibited by using either the GATE input or external logic. Otherwise, the count may be in process of changing when it is read, giving an undefined result.

Counter Latch Command

The other method for reading the Counters involves a special software command called the "Counter Latch Command". Like a Control Word, this command is written to the Control Word Register, which is selected when A1, A0 = 11. Also, like a Control Word, the SC0, SC1 bits select one of the three Counters, but two other bits, D5 and D4, distinguish this command from a Control Word.

A1, A0 = 11; $\overline{CS}$ = 0; $\overline{RD}$ = 1; $\overline{WR}$ = 0

D7	D6	D5	D4	D3	D2	D1	D0
SC1	SC0	0	0	X	X	X	X

SC1, SC0 - specify counter to be latched

SC1	SC0	COUNTER
0	0	0
0	1	1
1	0	2
1	1	Read-Back Command

D5, D4 - 00 designates Counter Latch Command, X - Don't Care.
NOTE: Don't Care bits (X) should be 0 to insure compatibility with future products.

The selected Counter's output latch (OL) latches the count when the Counter Latch Command is received. This count is held in the latch until it is read by the CPU (or until the Counter is reprogrammed). The count is then unlatched automatically and the OL returns to "following" the counting element (CE). This allows reading the contents of the Counters "on the fly" without affecting counting in progress. Multiple Counter Latch Commands may be used to latch more than one Counter. Each latched Counter's OL holds its count until read. Counter Latch Commands do not affect the programmed Mode of the Counter in any way.

If a Counter is latched and then, some time later, latched again before the count is read, the second Counter Latch Command is ignored. The count read will be the count at the time the first Counter Latch Command was issued.

With either method, the count must be read according to the programmed format; specifically, if the Counter is programmed for two byte counts, two bytes must be read. The two bytes do not have to be read one right after the other; read or write or programming operations of other Counters may be inserted between them.

Another feature of the 82C54 is that reads and writes of the same Counter may be interleaved; for example, if the Counter is programmed for two byte counts, the following sequence is valid.

1. Read least significant byte.
2. Write new least significant byte.
3. Read most significant byte.
4. Write new most significant byte.

If a counter is programmed to read or write two-byte counts, the following precaution applies: A program MUST NOT transfer control between reading the first and second byte to another routine which also reads from that same Counter. Otherwise, an incorrect count will be read.

Read-Back Command

The read-back command allows the user to check the count value, programmed Mode, and current state of the OUT pin and Null Count flag of the selected counter(s).

The command is written into the Control Word Register and has the format shown in Figure 5. The command applies to the counters selected by setting their corresponding bits D3, D2, D1 = 1.

A0, A1 = 11; $\overline{CS}$ = 0; $\overline{RD}$ = 1; $\overline{WR}$ = 0

D7	D6	D5	D4	D3	D2	D1	D0
1	1	$\overline{COUNT}$	$\overline{STATUS}$	CNT 2	CNT 1	CNT 0	0

D5: 0 = Latch count of selected Counter (s)
D4: 0 = Latch status of selected Counter(s)
D3: 1 = Select Counter 2
D2: 1 = Select Counter 1
D1: 1 = Select Counter 0
D0: Reserved for future expansion; Must be 0

FIGURE 5. READ-BACK COMMAND FORMAT

The read-back command may be used to latch multiple counter output latches (OL) by setting the COUNT bit D5 = 0 and selecting the desired counter(s). This signal command is functionally equivalent to several counter latch commands, one for each counter latched. Each counter's latched count is held until it is read (or the counter is reprogrammed). That counter is automatically unlatched when read, but other counters remain latched until they are read. If multiple count read-back commands are issued to the same counter without reading the count, all but the first are ignored; i.e., the count which will be read is the count at the time the first read-back command was issued.

The read-back command may also be used to latch status information of selected counter(s) by setting STATUS bit D4 = 0. Status must be latched to be read; status of a counter is accessed by a read from that counter.

The counter status format is shown in Figure 6. Bits D5 through D0 contain the counter's programmed Mode exactly as written in the last Mode Control Word. OUTPUT bit D7 contains the current state of the OUT pin. This allows the user to monitor the counter's output via software, possibly eliminating some hardware from a system.

D7	D6	D5	D4	D3	D2	D1	D0
OUTPUT	NULL COUNT	RW1	RW0	M2	M1	M0	BCD

D7: 1 = Out pin is 1
0 = Out pin is 0
D6: 1 = Null count
0 = Count available for reading
D5 - D0 = Counter programmed mode (See Control Word Formats)

FIGURE 6. STATUS BYTE

NULL COUNT bit D6 indicates when the last count written to the counter register (CR) has been loaded into the counting element (CE). The exact time this happens depends on the Mode of the counter and is described in the Mode Definitions, but until the counter is loaded into the counting element (CE), it can't be read from the counter. If the count is latched or read before this time, the count value will not reflect the new count just written. The operation of Null Count is shown below.

THIS ACTION: CAUSES:

A. Write to the control word register:(1) . . . . . Null Count = 1
B. Write to the count register (CR):(2) . . . . . Null Count = 1
C. New count is loaded into CE (CR - CE) . . . . . Null Count = 0

(1) Only the counter specified by the control word will have its null count set to 1. Null count bits of other counters are unaffected.

(2) If the counter is programmed for two-byte counts (least significant byte then most significant byte) null count goes to 1 when the second byte is written.

If multiple status latch operations of the counter(s) are performed without reading the status, all but the first are ignored; i.e., the status that will be read is the status of the counter at the time the first status read-back command was issued.

COMMANDS								DESCRIPTION	RESULT
D7	D6	D5	D4	D3	D2	D1	D0		
1	1	0	0	0	0	1	0	Read-Back Count and Status of Counter 0	Count and Status Latched for Counter 0
1	1	1	0	0	1	0	0	Read-Back Status of Counter 1	Status Latched for Counter 1
1	1	1	0	1	1	0	0	Read-Back Status of Counters 2, 1	Status Latched for Counter 2, But Not Counter 1
1	1	0	1	1	0	0	0	Read-Back Count of Counter 2	Count Latched for Counter 2
1	1	0	0	0	1	0	0	Read-Back Count and Status of Counter 1	Count Latched for Counter 1, But Not Status
1	1	1	0	0	0	1	0	Read-Back Status of Counter 1	Command Ignored, Status Already Latched for Counter 1

FIGURE 7. READ-BACK COMMAND EXAMPLE

Both count and status of the selected counter(s) may be latched simultaneously by setting both COUNT and STATUS bits D5, D4 = 0. This is functionally the same as issuing two separate read-back commands at once, and the above discussions apply here also. Specifically, if multiple count and/or status read-back commands are issued to the same counter(s) without any intervening reads, all but the first are ignored. This is illustrated in Figure 7.

If both count and status of a counter are latched, the first read operation of that counter will return latched status, regardless of which was latched first. The next one or two reads (depending on whether the counter is programmed for one or two type counts) return latched count. Subsequent reads return unlatched count.

$\overline{CS}$	$\overline{RD}$	$\overline{WR}$	A1	A0	
0	1	0	0	0	Write into Counter 0
0	1	0	0	1	Write into Counter 1
0	1	0	1	0	Write into Counter 2
0	1	0	1	1	Write Control Word
0	0	1	0	0	Read from Counter 0
0	0	1	0	1	Read from Counter 1
0	0	1	1	0	Read from Counter 2
0	0	1	1	1	No-Operation (Three-State)
1	X	X	X	X	No-Operation (Three-State)
0	1	1	X	X	No-Operation (Three-State)

FIGURE 8. READ/WRITE OPERATIONS SUMMARY

Mode Definitions

The following are defined for use in describing the operation of the 82C54.

CLK PULSE:

A rising edge, then a falling edge, in that order, of a Counter's CLK input.

TRIGGER:

A rising edge of a Counter's Gate input.

COUNTER LOADING:

The transfer of a count from the CR to the CE (See "Functional Description")

Mode 0: Interrupt on Terminal Count

Mode 0 is typically used for event counting. After the Control Word is written, OUT is initially low, and will remain low until the Counter reaches zero. OUT then goes high and remains high until a new count or a new Mode 0 Control Word is written to the Counter.

GATE = 1 enables counting; GATE = 0 disables counting. GATE has no effect on OUT.

After the Control Word and initial count are written to a Counter, the initial count will be loaded on the next CLK pulse. This CLK pulse does not decrement the count, so for an initial count of N, OUT does not go high until N + 1 CLK pulses after the initial count is written.

If a new count is written to the Counter it will be loaded on the next CLK pulse and counting will continue from the new count. If a two-byte count is written, the following happens:

(1) Writing the first byte disables counting. Out is set low immediately (no clock pulse required).

(2) Writing the second byte allows the new count to be loaded on the next CLK pulse.

This allows the counting sequence to be synchronized by software. Again OUT does not go high until N + 1 CLK pulses after the new count of N is written.

If an initial count is written while GATE = 0, it will still be loaded on the next CLK pulse. When GATE goes high, OUT will go high N CLK pulses later; no CLK pulse is needed to load the counter as this has already been done.

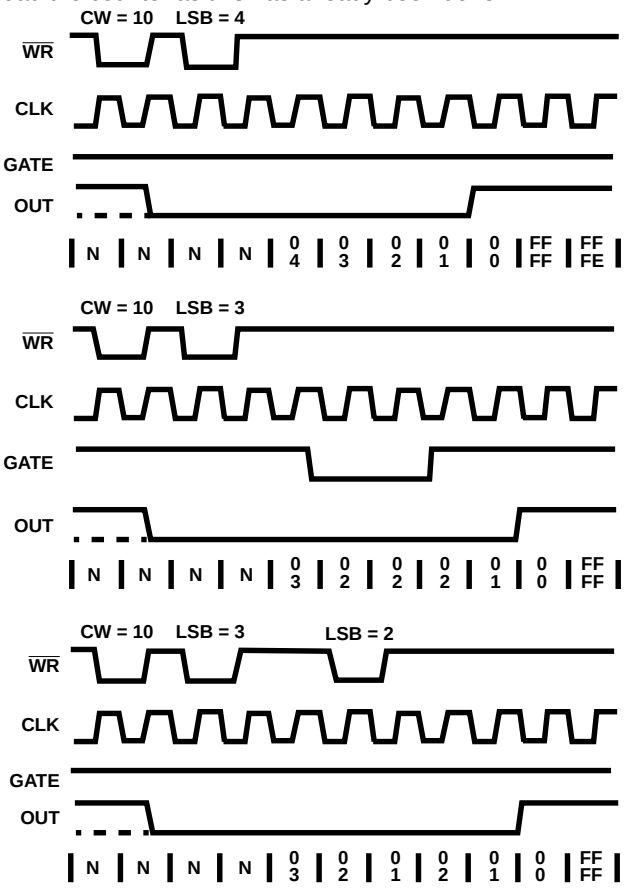


FIGURE 9. MODE 0

NOTES: The following conventions apply to all mode timing diagrams.

1. Counters are programmed for binary (not BCD) counting and for reading/writing least significant byte (LSB) only.
2. The counter is always selected ($\overline{CS}$ always low).
3. CW stands for "Control Word"; CW = 10 means a control word of 10, Hex is written to the counter.
4. LSB stands for Least significant "byte" of count.
5. Numbers below diagrams are count values. The lower number is the least significant byte. The upper number is the most significant byte. Since the counter is programmed to read/write LSB only, the most significant byte cannot be read.
6. N stands for an undefined count.
7. Vertical lines show transitions between count values.

Mode 1: Hardware Retriggerable One-Shot

OUT will be initially high. OUT will go low on the CLK pulse following a trigger to begin the one-shot pulse, and will remain low until the Counter reaches zero. OUT will then go high and remain high until the CLK pulse after the next trigger.

After writing the Control Word and initial count, the Counter is armed. A trigger results in loading the Counter and setting OUT low on the next CLK pulse, thus starting the one-shot pulse N CLK cycles in duration. The one-shot is retriggerable, hence OUT will remain low for N CLK pulses after any trigger. The one-shot pulse can be repeated without rewriting the same count into the counter. GATE has no effect on OUT.

If a new count is written to the Counter during a one-shot pulse, the current one-shot is not affected unless the Counter is retriggerable. In that case, the Counter is loaded with the new count and the one-shot pulse continues until the new count expires.

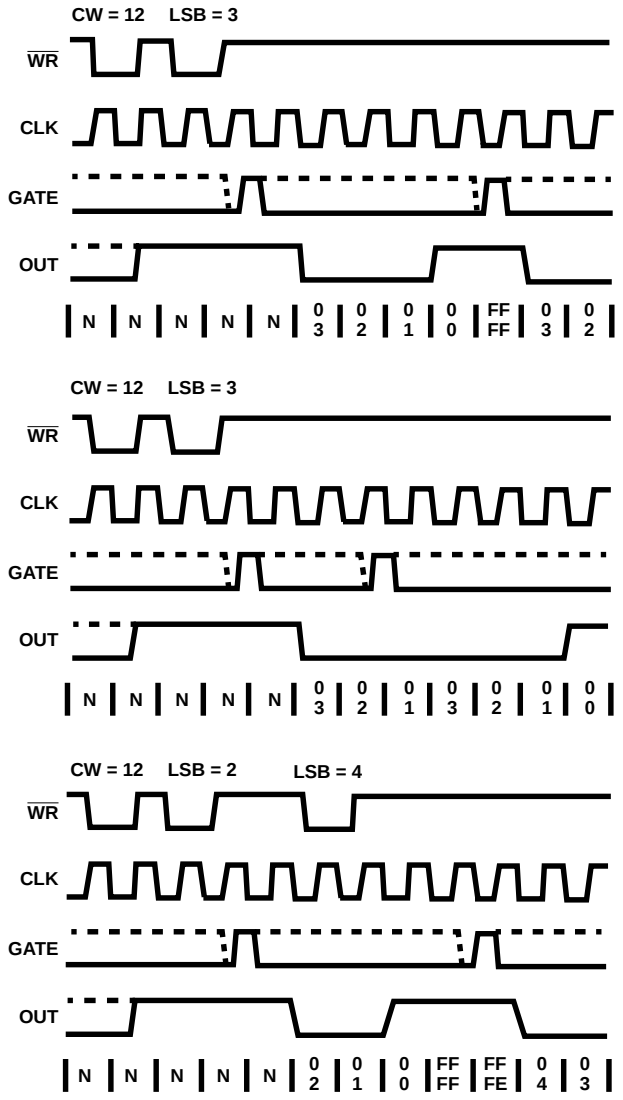


FIGURE 10. MODE 1

Mode 2: Rate Generator

This Mode functions like a divide-by-N counter. It is typically used to generate a Real Time Clock Interrupt. OUT will initially be high. When the initial count has decremented to 1, OUT goes low for one CLK pulse. OUT then goes high again, the Counter reloads the initial count and the process is repeated. Mode 2 is periodic; the same sequence is repeated indefinitely. For an initial count of N, the sequence repeats every N CLK cycles.

GATE = 1 enables counting; GATE = 0 disables counting. If GATE goes low during an output pulse, OUT is set high immediately. A trigger reloads the Counter with the initial count on the next CLK pulse; OUT goes low N CLK pulses after the trigger. Thus the GATE input can be used to synchronize the Counter.

After writing a Control Word and initial count, the Counter will be loaded on the next CLK pulse. OUT goes low N CLK pulses after the initial count is written. This allows the Counter to be synchronized by software also.

Writing a new count while counting does not affect the current counting sequence. If a trigger is received after writing a new count but before the end of the current period, the Counter will be loaded with the new count on the next CLK pulse and counting will continue from the end of the current counting cycle.

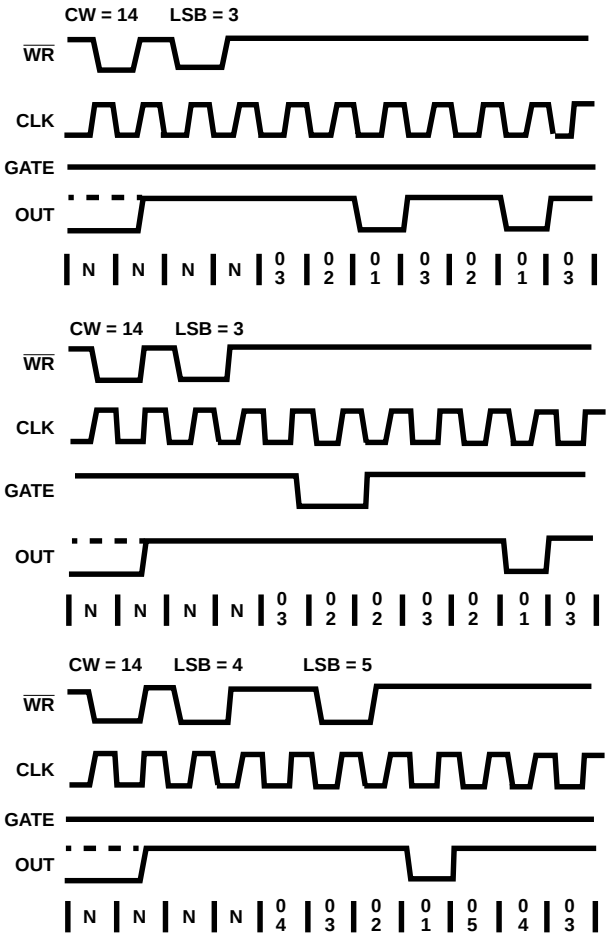


FIGURE 11. MODE 2

Mode 3: Square Wave Mode

Mode 3 is typically used for Baud rate generation. Mode 3 is similar to Mode 2 except for the duty cycle of OUT. OUT will initially be high. When half the initial count has expired, OUT goes low for the remainder of the count. Mode 3 is periodic; the sequence above is repeated indefinitely. An initial count of N results in a square wave with a period of N CLK cycles.

GATE = 1 enables counting; GATE = 0 disables counting. If GATE goes low while OUT is low, OUT is set high immediately; no CLK pulse is required. A trigger reloads the Counter with the initial count on the next CLK pulse. Thus the GATE input can be used to synchronize the Counter.

After writing a Control Word and initial count, the Counter will be loaded on the next CLK pulse. This allows the Counter to be synchronized by software also.

Writing a new count while counting does not affect the current counting sequence. If a trigger is received after writing a new count but before the end of the current half-cycle of the square wave, the Counter will be loaded with the new count on the next CLK pulse and counting will continue from the new count. Otherwise, the new count will be loaded at the end of the current half-cycle.

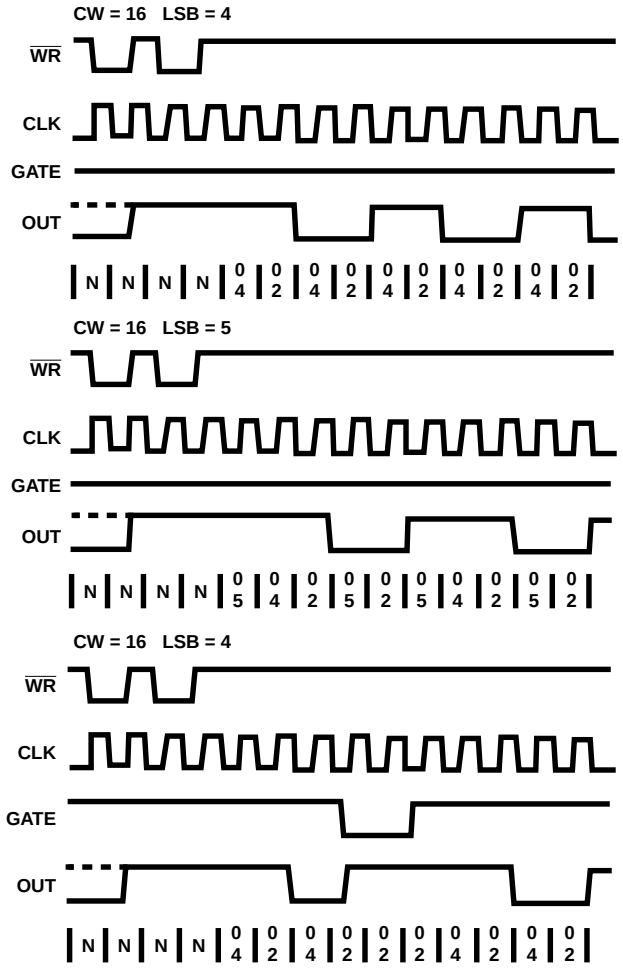


FIGURE 12. MODE 3

Mode 3 is Implemented as Follows:

EVEN COUNTS: OUT is initially high. The initial count is loaded on one CLK pulse and then is decremented by two on succeeding CLK pulses. When the count expires, OUT changes value and the Counter is reloaded with the initial count. The above process is repeated indefinitely.

ODD COUNTS: OUT is initially high. The initial count is loaded on one CLK pulse, decremented by one on the next CLK pulse, and then decremented by two on succeeding CLK pulses. When the count expires, OUT goes low and the Counter is reloaded with the initial count. The count is decremented by three on the next CLK pulse, and then by two on succeeding CLK pulses. When the count expires, OUT goes high again and the Counter is reloaded with the initial count. The above process is repeated indefinitely. So for odd counts, OUT will be high for (N + 1)/2 counts and low for (N - 1)/2 counts.

Mode 4: Software Triggered Mode

OUT will be initially high. When the initial count expires, OUT will go low for one CLK pulse then go high again. The counting sequence is “Triggered” by writing the initial count.

GATE = 1 enables counting; GATE = 0 disables counting. GATE has no effect on OUT.

After writing a Control Word and initial count, the Counter will be loaded on the next CLK pulse. This CLK pulse does not decrement the count, so for an initial count of N, OUT does not strobe low until N + 1 CLK pulses after the initial count is written.

If a new count is written during counting, it will be loaded on the next CLK pulse and counting will continue from the new count. If a two-byte count is written, the following happens:

(1) Writing the first byte has no effect on counting.

(2) Writing the second byte allows the new count to be loaded on the next CLK pulse.

This allows the sequence to be “retriggered” by software. OUT strobes low N + 1 CLK pulses after the new count of N is written.

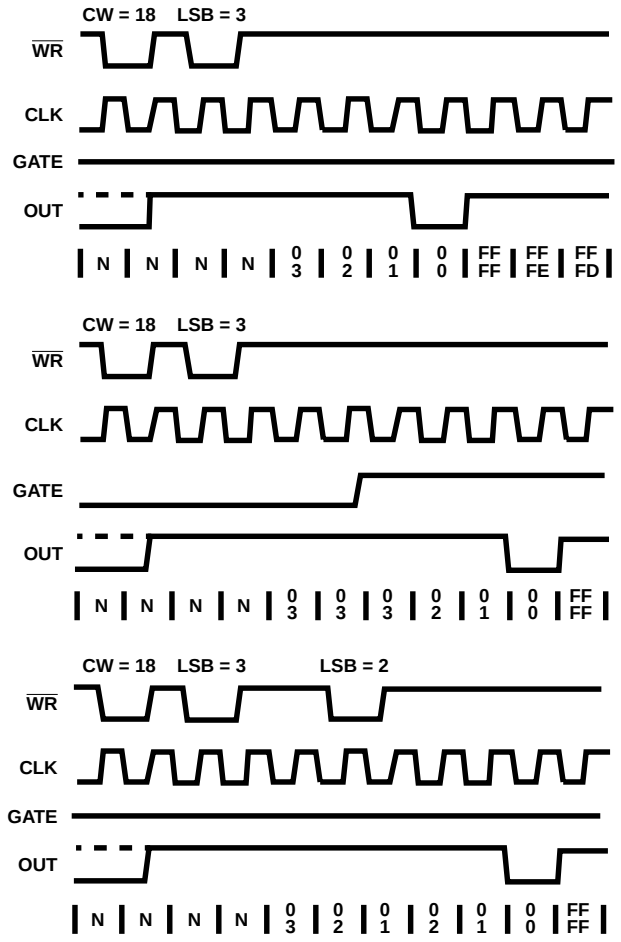


FIGURE 13. MODE 4

Mode 5: Hardware Triggered Strobe (Retriggerable)

OUT will initially be high. Counting is triggered by a rising edge of GATE. When the initial count has expired, OUT will go low for one CLK pulse and then go high again.

After writing the Control Word and initial count, the counter will not be loaded until the CLK pulse after a trigger. This CLK pulse does not decrement the count, so for an initial count of N, OUT does not strobe low until N + 1 CLK pulses after trigger.

A trigger results in the Counter being loaded with the initial count on the next CLK pulse. The counting sequence is triggerable. OUT will not strobe low for N + 1 CLK pulses after any trigger GATE has no effect on OUT.

If a new count is written during counting, the current counting sequence will not be affected. If a trigger occurs after the new count is written but before the current count expires, the Counter will be loaded with new count on the next CLK pulse and counting will continue from there.

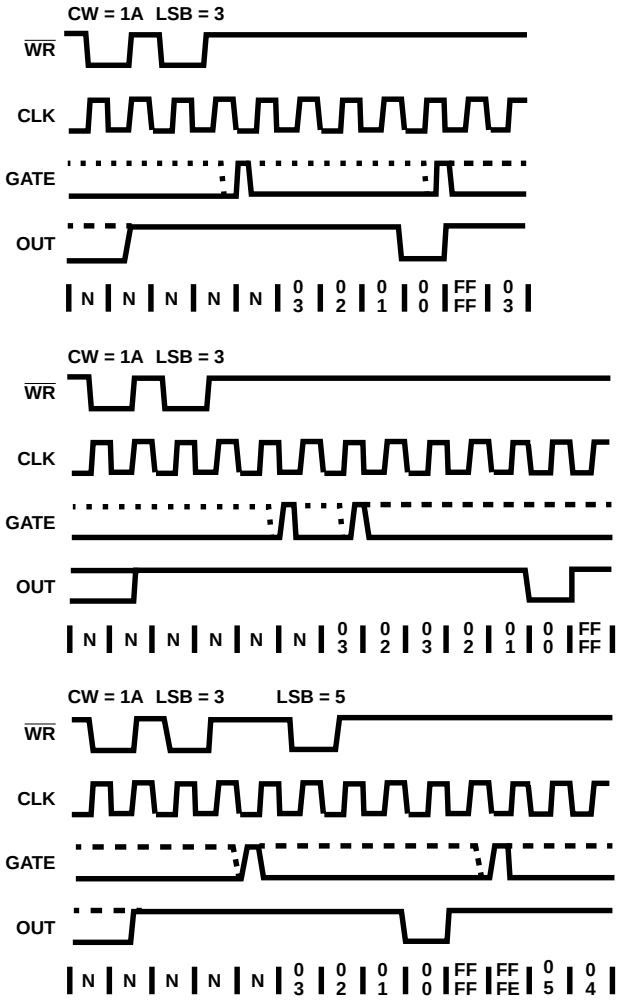


FIGURE 14. MODE 5

Operation Common to All Modes

Programming

When a Control Word is written to a Counter, all Control Logic, is immediately reset and OUT goes to a known initial state; no CLK pulses are required for this.

Gate

The GATE input is always sampled on the rising edge of CLK. In Modes 0, 2, 3 and 4 the GATE input is level sensitive, and logic level is sampled on the rising edge of CLK. In modes 1, 2, 3 and 5 the GATE input is rising-edge sensitive. In these Modes, a rising edge of Gate (trigger) sets an edge-sensitive flip-flop in the Counter. This flip-flop is then sampled on the next rising edge of CLK. The flip-flop is reset immediately after it is sampled. In this way, a trigger will be detected no matter when it occurs - a high logic level does not have to be maintained until the next rising edge of CLK. Note that in Modes 2 and 3, the GATE input is both edge- and level-sensitive.

Counter

New counts are loaded and Counters are decremented on the falling edge of CLK.

The largest possible initial count is 0; this is equivalent to 2^{16} for binary counting and 10^4 for BCD counting.

The counter does not stop when it reaches zero. In Modes 0, 1, 4, and 5 the Counter “wraps around” to the highest count, either FFFF hex for binary counting or 9999 for BCD counting, and continues counting. Modes 2 and 3 are periodic; the Counter reloads itself with the initial count and continues counting from there.

SIGNAL STATUS MODES	LOW OR GOING LOW	RISING	HIGH
0	Disables Counting	-	Enables Counting
1	-	1) Initiates Counting 2) Resets output after next clock	-
2	1) Disables counting 2) Sets output immediately high	Initiates Counting	Enables Counting
3	1) Disables counting 2) Sets output immediately high	Initiates Counting	Enables Counting
4	1) Disables Counting	-	Enables Counting
5	-	Initiates Counting	-

FIGURE 15. GATE PIN OPERATIONS SUMMARY

MODE	MIN COUNT	MAX COUNT
0	1	0
1	1	0
2	2	0
3	2	0
4	1	0
5	1	0

NOTE: 0 is equivalent to 2^{16} for binary counting and 10^4 for BCD counting.

FIGURE 16. MINIMUM AND MAXIMUM INITIAL COUNTS

Absolute Maximum Ratings

Supply Voltage . . . . . +8.0V
Input, Output or I/O Voltage . . . . . GND-0.5V to V_{CC} +0.5V
ESD Classification . . . . . Class 1

Operating Conditions

Operating Voltage Range . . . . . +4.5V to +5.5V
Operating Temperature Range
C82C54, C82C54-10, -12 . . . . . 0^oC to $+70^oC$
I82C54, I82C54-10, -12 . . . . . -40^oC to $+85^oC$
M82C54, M82C54-10, -12 . . . . . -55^oC to $+125^oC$

Thermal Information

Thermal Resistance (Typical)	θ_{JA} ($^oC/W$)	θ_{JC} ($^oC/W$)
CERDIP Package	55	12
CLCC Package	65	14
PDIP Package	60	N/A
PLCC Package	65	N/A
SOIC Package	75	N/A

Storage Temperature Range . . . . . -65^oC to $+150^oC$
Maximum Junction Temperature Ceramic Package . . . . . $+175^oC$
Maximum Junction Temperature Plastic Package . . . . . $+150^oC$
Maximum Lead Temperature Package (Soldering 10s) . . . . $+300^oC$
(PLCC and SOIC - Lean Tips Only)

Die Characteristics

Gate Count . . . . . 2250 Gates

CAUTION: Stresses above those listed in "Absolute Maximum Ratings" may cause permanent damage to the device. This is a stress only rating and operation of the device at these or any other conditions above those indicated in the operational sections of this specification is not implied.

DC Electrical Specifications

V_{CC} = +5.0V ± 10%, $T_A = 0^oC$ to $+70^oC$ (C82C54, C82C54-10, C82C54-12)
$T_A = -40^oC$ to $+85^oC$ (I82C54, I82C54-10, I82C54-12)
$T_A = -55^oC$ to $+125^oC$ (M82C54, M82C54-10, M82C54-12

SYMBOL	PARAMETER	MIN	MAX	UNITS	TEST CONDITIONS
VIH	Logical One Input Voltage	2.0	-	V	C82C54, I82C54
		2.2	-	V	M82C54
VIL	Logical Zero Input Voltage	-	0.8	V	
VOH	Output HIGH Voltage	3.0	-	V	IOH = -2.5mA
		V_{CC} -0.4	-	V	IOH = -100µA
VOL	Output LOW Voltage	-	0.4	V	IOL = +2.5mA
II	Input Leakage Current	-1	+1	µA	VIN = GND or V_{CC} DIP Pins 9,11,14-16,18-23
IO	Output Leakage Current	-10	+10	µA	VOUT = GND or V_{CC} DIP Pins 1-8
ICCSB	Standby Power Supply Current	-	10	µA	V_{CC} = 5.5V, VIN = GND or V_{CC}, Outputs Open, Counters Programmed
ICCOP	Operating Power Supply Current	-	10	mA	V_{CC} = 5.5V, CLK0 = CLK1 = CLK2 = 8MHz, VIN = GND or V_{CC}, Outputs Open

Capacitance

$T_A = +25^oC$; All Measurements Referenced to Device GND, Note 1

SYMBOL	PARAMETER	TYP	UNITS	TEST CONDITIONS
CIN	Input Capacitance	20	pF	FREQ = 1MHz
COUT	Output Capacitance	20	pF	FREQ = 1MHz
CI/O	I/O Capacitance	20	pF	FREQ = 1MHz

NOTE:

1. Not tested, but characterized at initial design and at major process/design changes.

AC Electrical Specifications V_{CC} = +5.0V ± 10%, T_A = 0°C to +70°C (C82C54, C82C54-10, C82C54-12)
T_A = -40°C to +85°C (I82C54, I82C54-10, I82C54-12)
T_A = -55°C to +125°C (M82C54, M82C54-10, M82C54-12)

SYMBOL		PARAMETER	82C54		82C54-10		82C54-12		UNITS	TEST CONDITIONS
			MIN	MAX	MIN	MAX	MIN	MAX		
READ CYCLE										
(1)	TAR	Address Stable Before $\overline{RD}$	30	-	25	-	25	-	ns	1
(2)	TSR	$\overline{CS}$ Stable Before $\overline{RD}$	0	-	0	-	0	-	ns	1
(3)	TRA	Address Hold Time After $\overline{RD}$	0	-	0	-	0	-	ns	1
(4)	TRR	$\overline{RD}$ Pulse Width	150	-	95	-	95	-	ns	1
(5)	TRD	Data Delay from $\overline{RD}$	-	120	-	85	-	85	ns	1
(6)	TAD	Data Delay from Address	-	210	-	185	-	185	ns	1
(7)	TDF	$\overline{RD}$ to Data Floating	5	85	5	65	5	65	ns	2, Note 1
(8)	TRV	Command Recovery Time	200	-	165	-	165	-	ns	
WRITE CYCLE										
(9)	TAW	Address Stable Before $\overline{WR}$	0	-	0	-	0	-	ns	
(10)	TSW	$\overline{CS}$ Stable Before $\overline{WR}$	0	-	0	-	0	-	ns	
(11)	TWA	Address Hold Time After $\overline{WR}$	0	-	0	-	0	-	ns	
(12)	TWW	$\overline{WR}$ Pulse Width	95	-	95	-	95	-	ns	
(13)	TDW	Data Setup Time Before $\overline{WR}$	140	-	95	-	95	-	ns	
(14)	TWD	Data Hold Time After $\overline{WR}$	25	-	0	-	0	-	ns	
(15)	TRV	Command Recovery Time	200	-	165	-	165	-	ns	
CLOCK AND GATE										
(16)	TCLK	Clock Period	125	DC	100	DC	80	DC	ns	1
(17)	TPWH	High Pulse Width	60	-	30	-	30	-	ns	1
(18)	TPWL	Low Pulse Width	60	-	40	-	30	-	ns	1
(19)	TR	Clock Rise Time	-	25	-	25	-	25	ns	
(20)	TF	Clock Fall Time	-	25	-	25	-	25	ns	
(21)	TGW	Gate Width High	50	-	50	-	50	-	ns	1
(22)	TGL	Gate Width Low	50	-	50	-	50	-	ns	1
(23)	TGS	Gate Setup Time to CLK	50	-	40	-	40	-	ns	1
(24)	TGH	Gate Hold Time After CLK	50	-	50	-	50	-	ns	1
(25)	TOD	Output Delay from CLK	-	150	-	100	-	100	ns	1
(26)	TODG	Output Delay from Gate	-	120	-	100	-	100	ns	1
(27)	TWO	OUT Delay from Mode Write	-	260	-	240	-	240	ns	1
(28)	TWC	CLK Delay for Loading	0	55	0	55	0	55	ns	1
(29)	TWG	Gate Delay for Sampling	-5	40	-5	40	-5	40	ns	1
(30)	TCL	CLK Setup for Count Latch	-40	40	-40	40	-40	40	ns	1

NOTE:

1. Not tested, but characterized at initial design and at major process/design changes.

Timing Waveforms

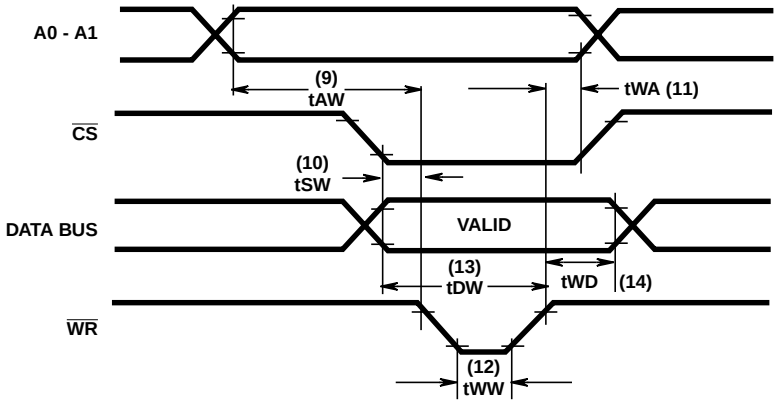


FIGURE 17. WRITE

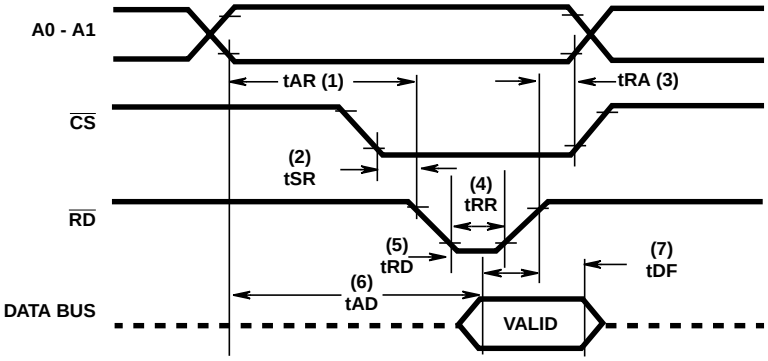


FIGURE 18. READ

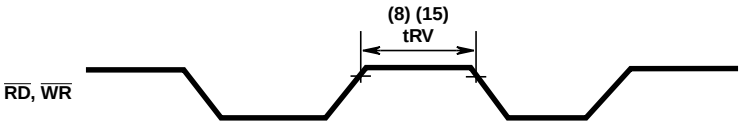


FIGURE 19. RECOVERY

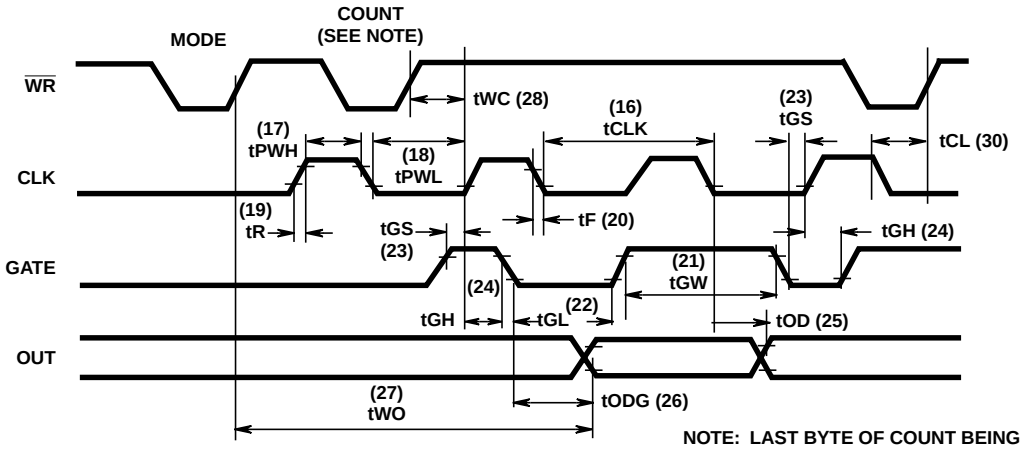
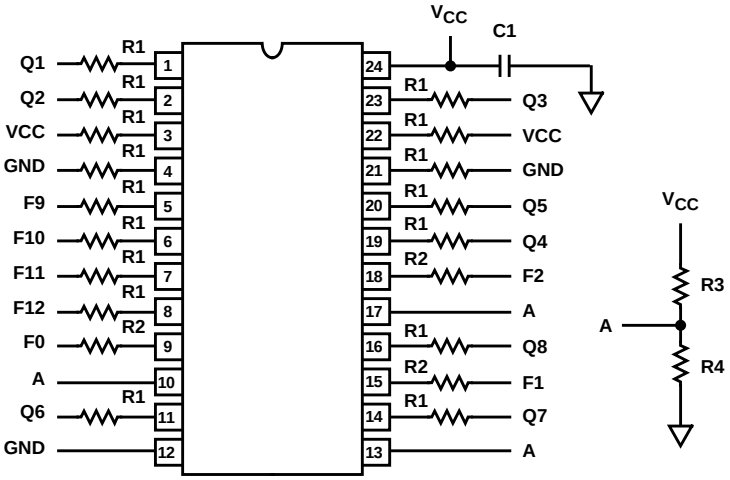


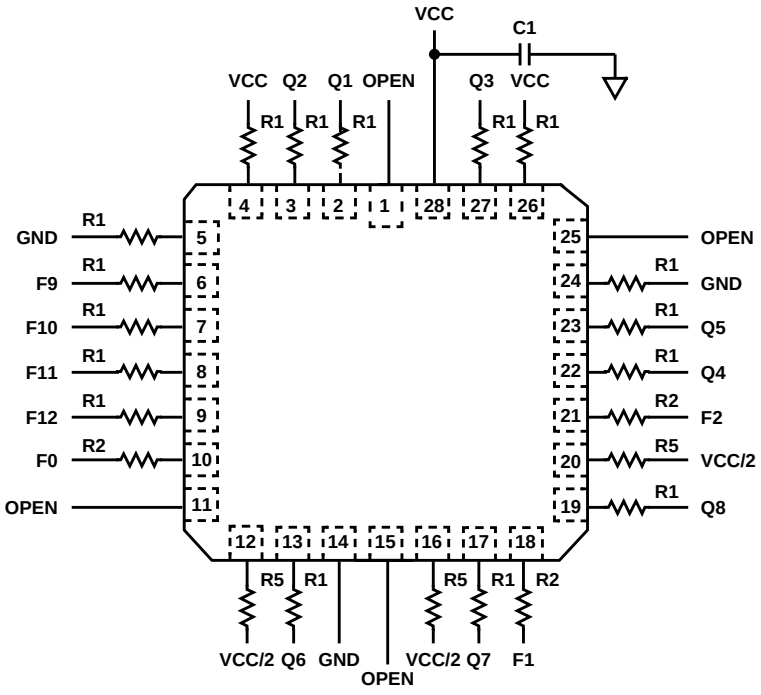
FIGURE 20. CLOCK AND GATE

Burn-In Circuits

MD 82C54 CERDIP



MR 82C54 CLCC



NOTES:

1. V_{CC} = 5.5V ± 0.5V
2. GND = 0V
3. VIH = 4.5V ±10%
4. VIL = -0.2V to 0.4V
5. R1 = 47kΩ ±5%
6. R2 = 1.0kΩ ±5%
7. R3 = 2.7kΩ ±5%
8. R4 = 1.8kΩ ±5%
9. R5 = 1.2kΩ ±5%
10. C1 = 0.01µF Min
11. F0 = 100kHz ±10%
12. F1 = F0/2, F2 = F1/2, ...F12 = F11/2

Die Characteristics

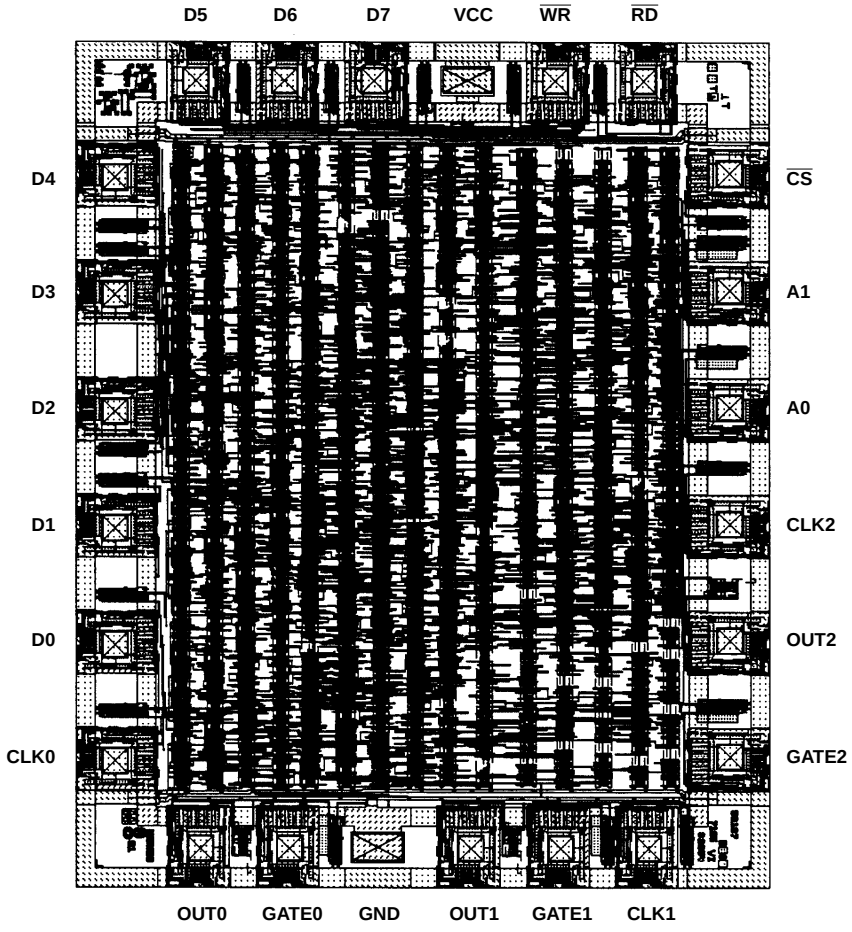
DIE DIMENSIONS:
129mils x 155mils x 19mils
(3270µm x 3940µm x 483µm)

METALLIZATION:
Type: Si-Al-Cu
Thickness: Metal 1: 8kÅ ± 0.75kÅ
Metal 2: 12kÅ ± 1.0kÅ

GLASSIVATION:
Type: Nitrox
Thickness: 10kÅ ± 3.0kÅ

Metallization Mask Layout

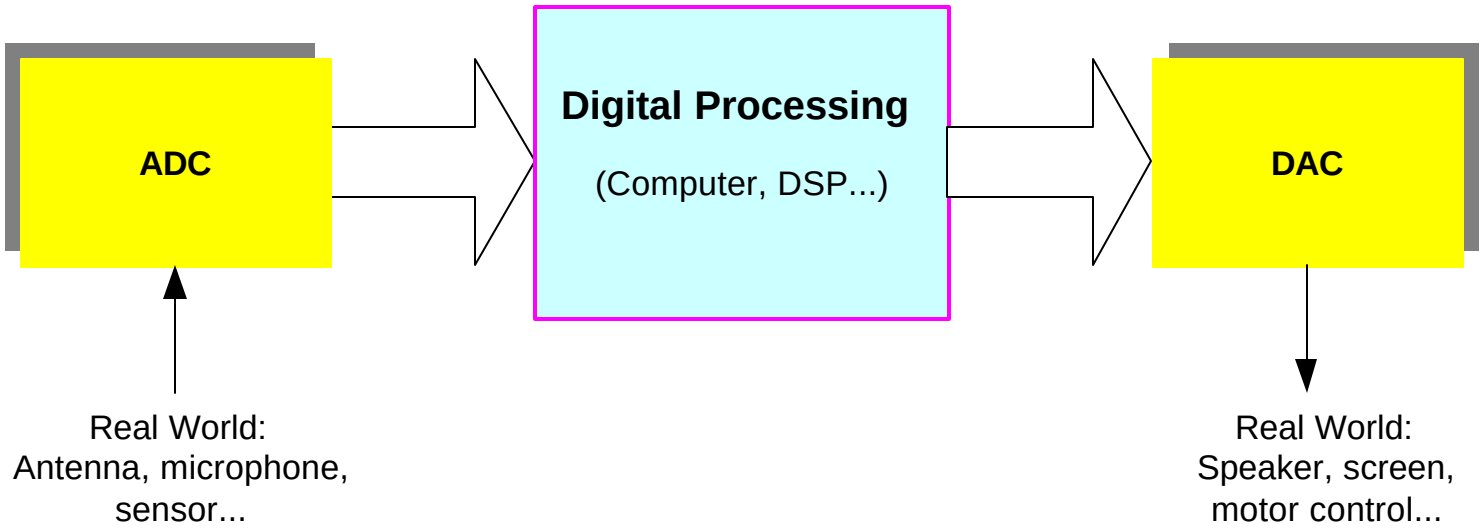
82C54



Data Converters

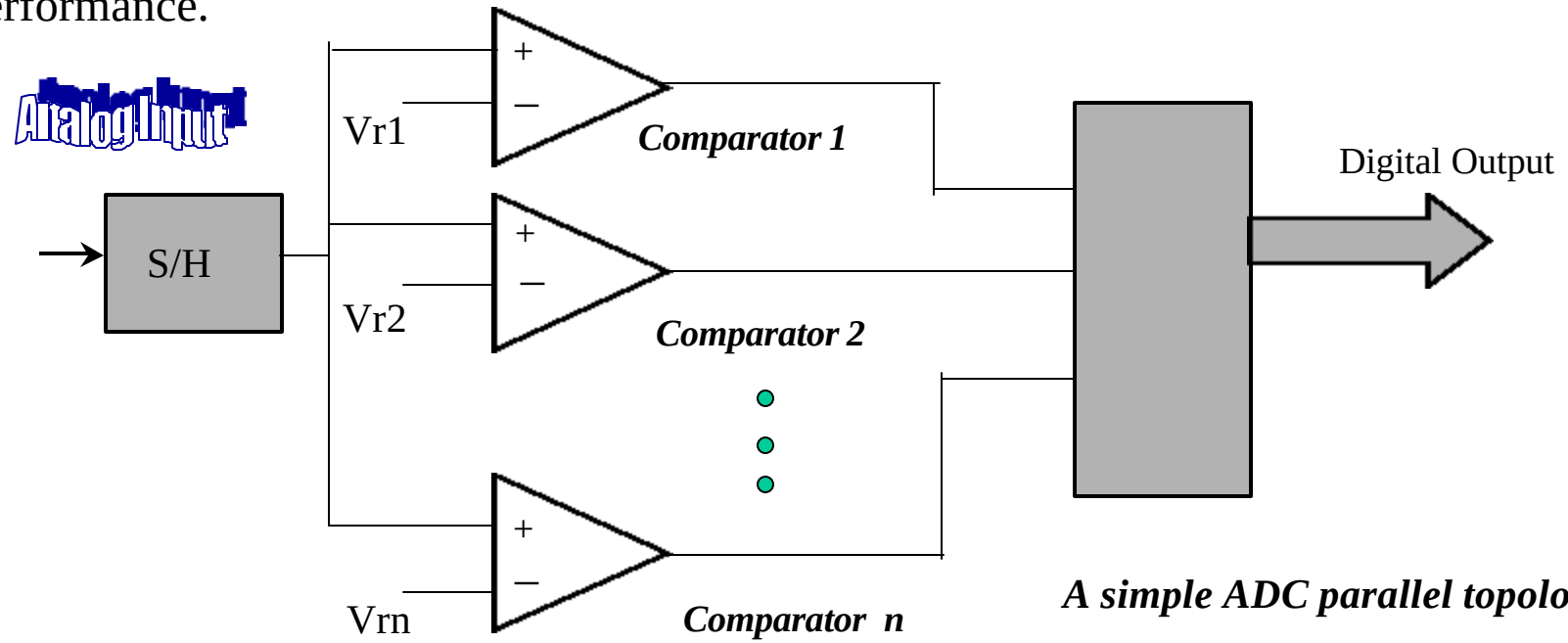
- The Real World is Analog

- ADC are necessary to convert the real world signals (analog) into the digital form for easy processing



Basic Concepts

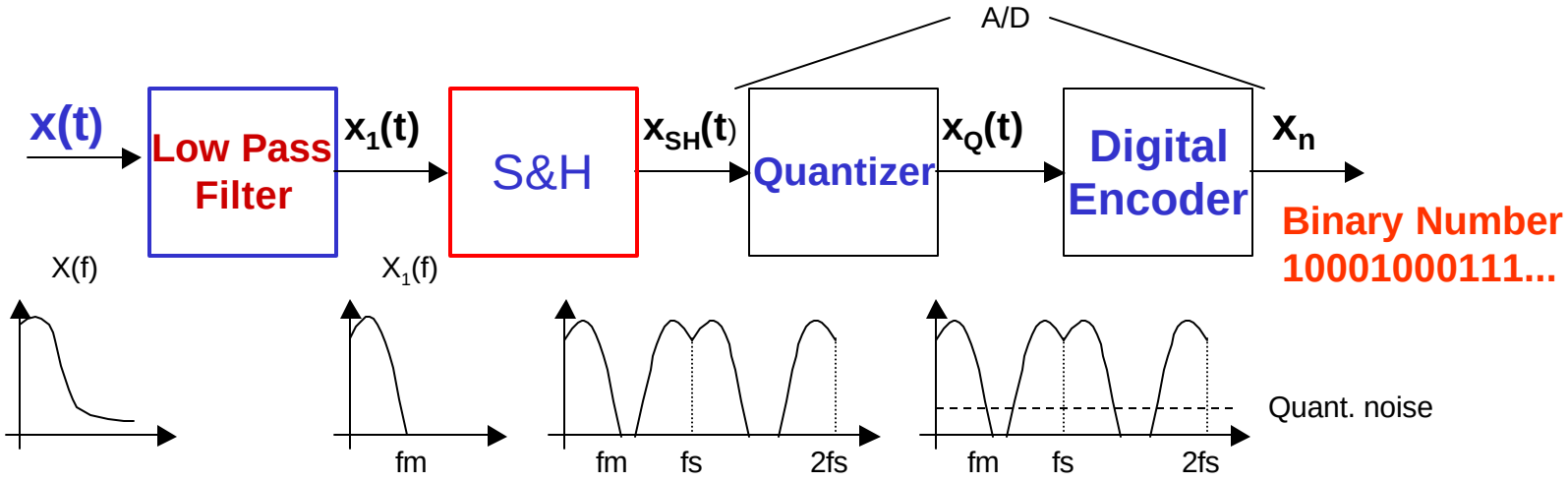
- The goal of an ADC is to determine the output digital word corresponding to an analog input signal.
- The basic internal structures of ADC rely heavily on DACs structures.
- ADCs can be seen as low speed (serial type), medium speed, high-speed and high-performance.



A simple ADC parallel topology

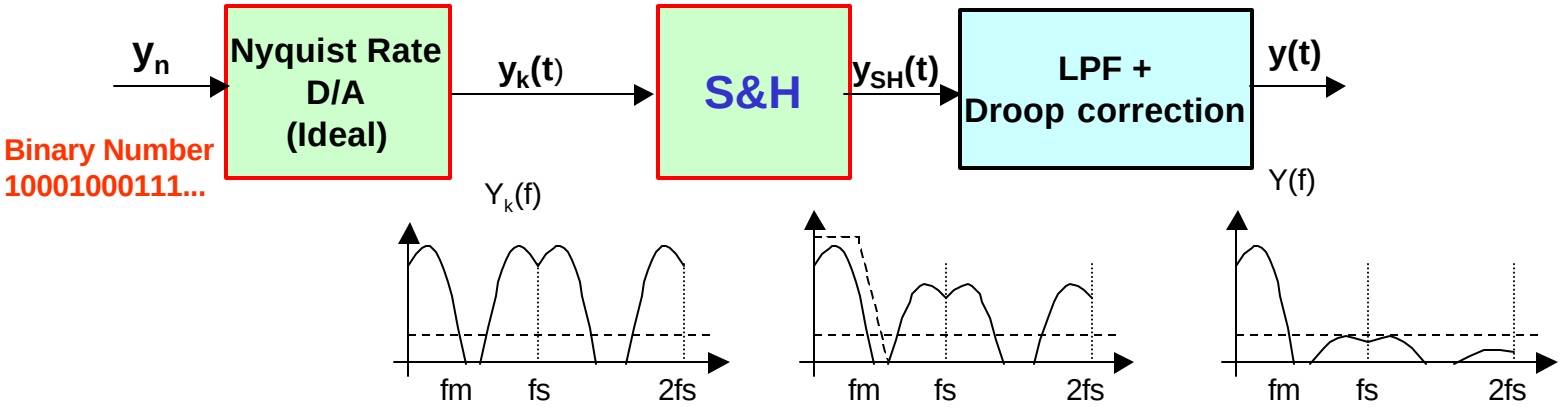
Fundamentals

- Traditional Data Converters at Nyquist Rate (fs>2fm)
  - A/D Converter Details:



Fundamentals

- Traditional Data converters at Nyquist Rate (fs>2fm)
  - D/A Converter:

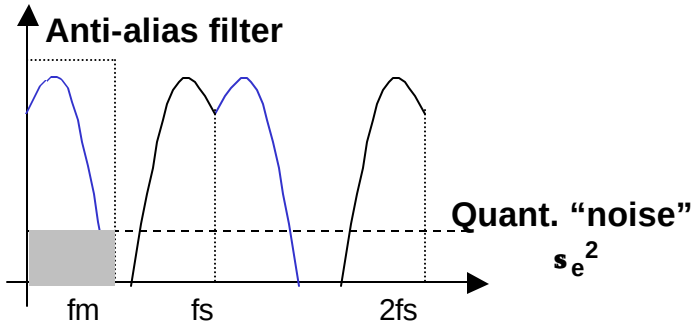


• Droop correction means inverse Sinc

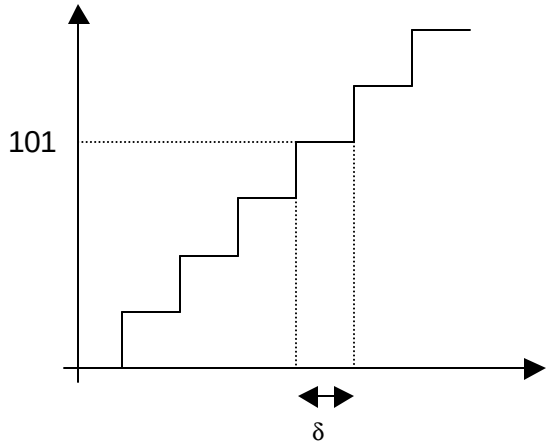
• The S/H is a “deglitching” circuit and could be eliminated for small glitches

Fundamentals

- A/D: Sampled Signal Spectrum:



$$DR = 1.5\,(K-1)^2 = \Big|_{dB}\ 1.76 + n*6.02$$



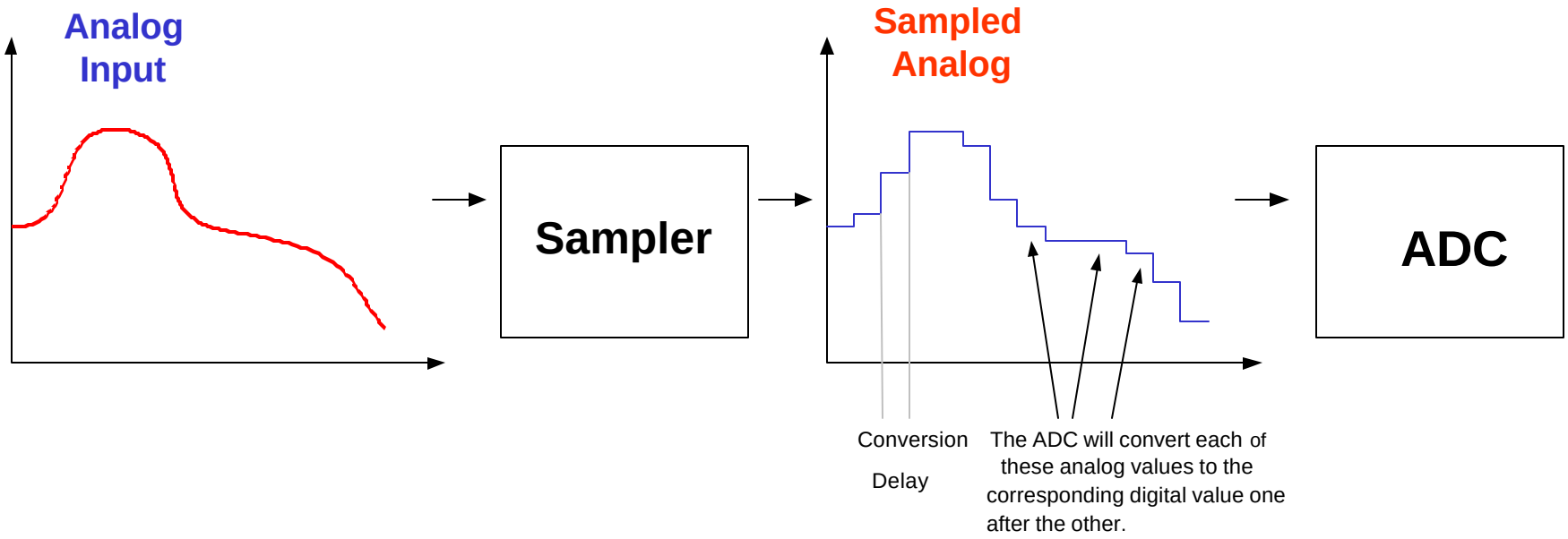
$$s_e^2 = d^2/12$$

K: # Quantizer levels

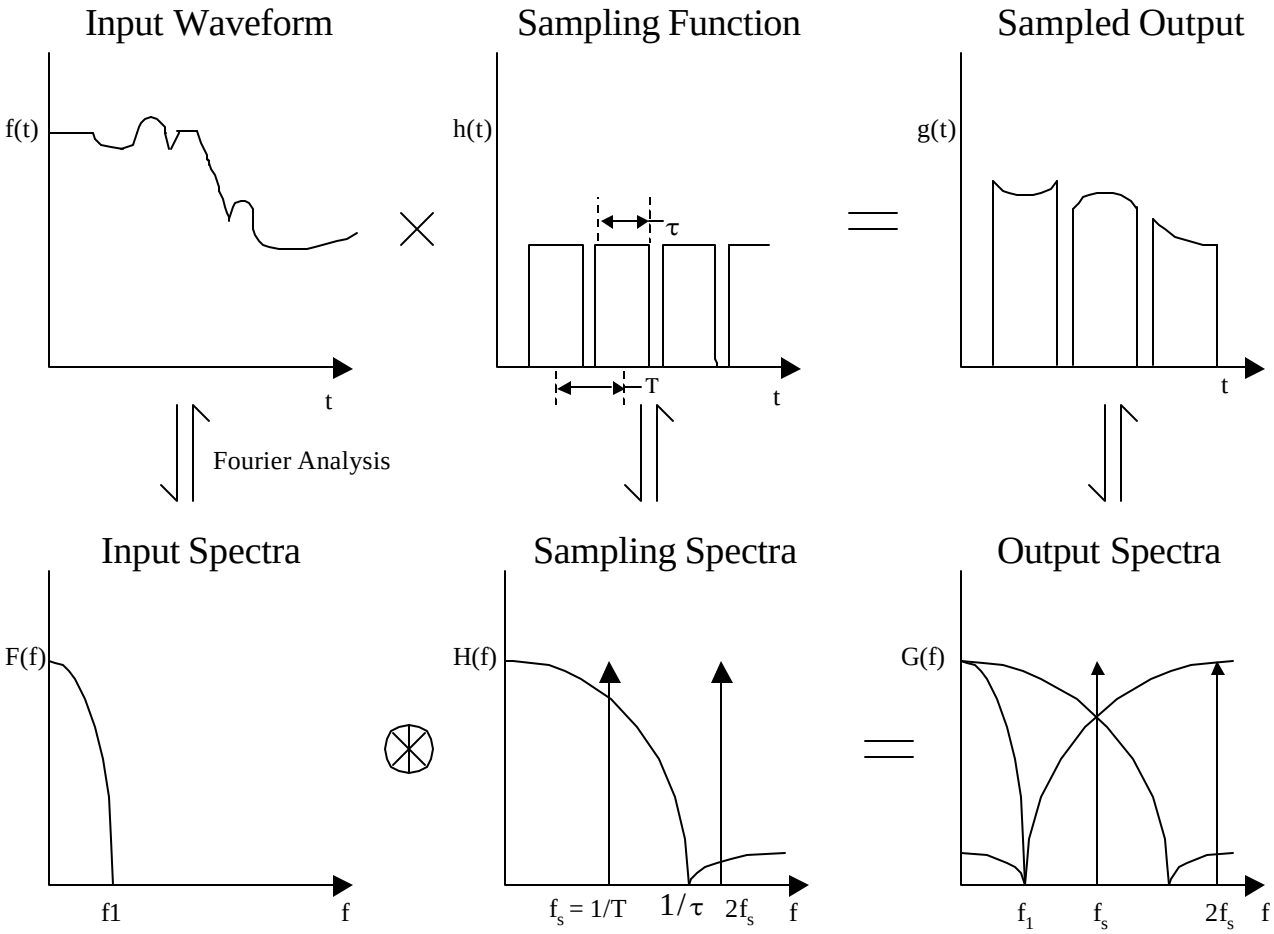
n: Equivalent # Bits

Sampling

- The process of converting to digital can not be instantaneous
- The input has to be stabilized while a conversion is performed.



REAL SAMPLING



Input signals are not truly band limited

$$f(s) \gtrless 2f_1$$

Sampling cannot be done with impulses so, amplitude of signal is modulated by

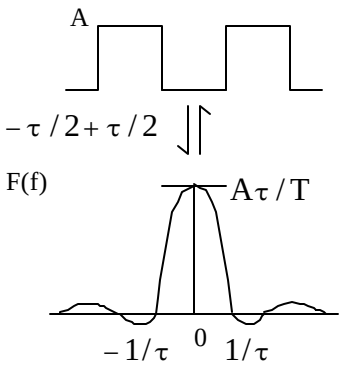
$$\frac{\sin x}{x}$$ envelope

Because of input spectra and sampling there is aliasing and distortion

Square Wave $\rightleftharpoons \frac{\sin x}{x}$

f(t)

Period T



Envelope has the form

$$E = \frac{A\tau}{T}\left(\frac{\sin \pi f \tau}{\pi f \tau}\right)$$

Performance Metrics

Since real Data Converters have a number of non-idealities we need to use a Performance Metrics to evaluate and compare them. In what follows we will attempt to define it.

The number of bits of the digital code is finite: for n-bit we have 2^n codes and each code represents a given *quantization level*.
The error due to the quantization is called quantization error and ranges between plus or minus half quantization level (LSB). This error is a consequence (and a measure) of the finite A/D converter resolution.
Furthermore, the quantization error can be considered as a noise if all quantization levels are exercised with equal probability, the quantization steps are uniform; a large number of quantization levels are used, and thev quantization error is not correlated with the input signal

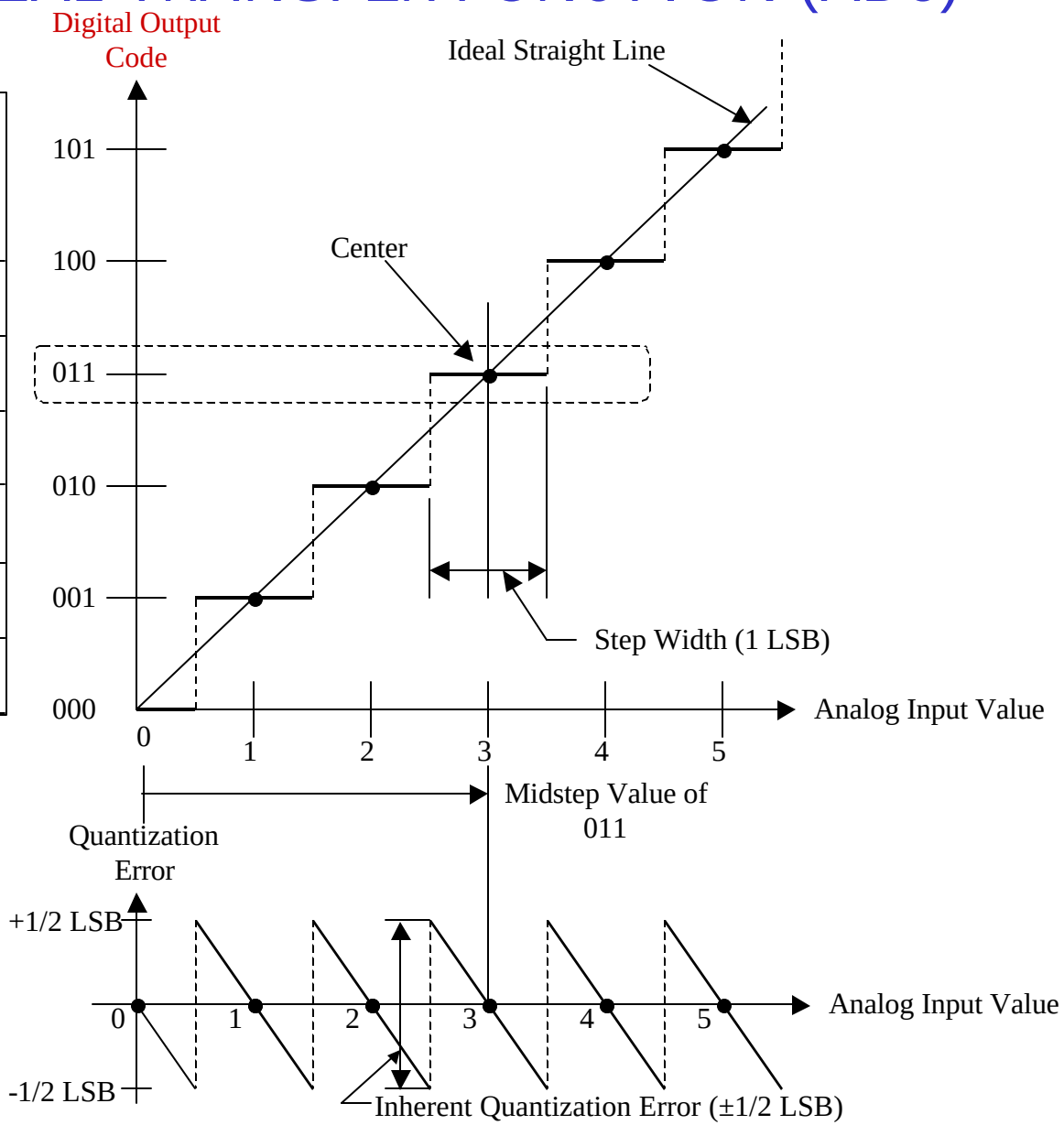
Definitions

- Differential Nonlinearity:
  Deviation in the width of a certain code from the value of 1LSB.
- Integral Non-Linearity:
  Deviation in the midpoint of the code from the best straight line in LSBs.

THE IDEAL TRANSFER FUNCTION (ADC)

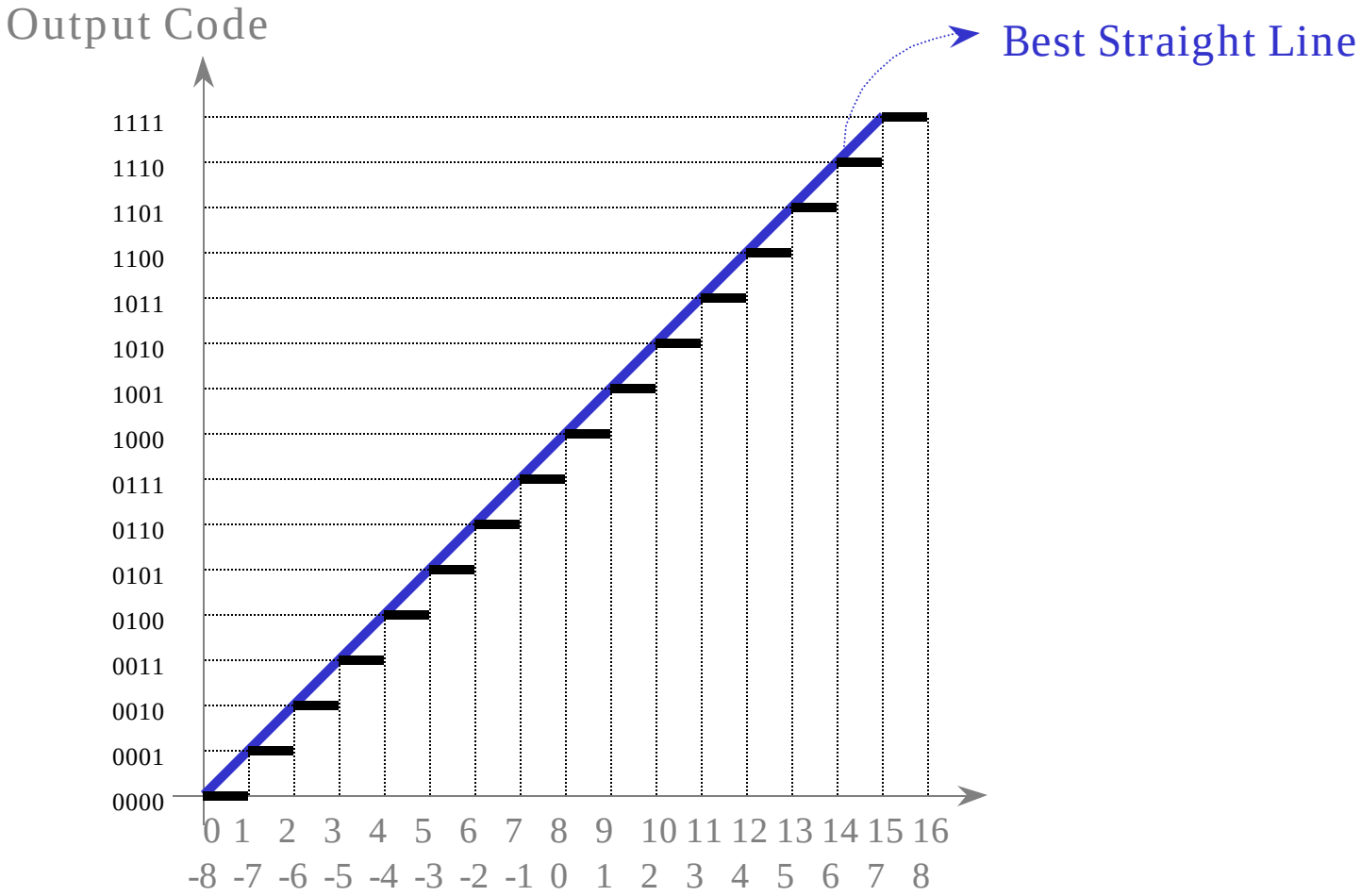
Conversion Code

Range of Analog Input Values	Digital Output Code
4.5 • 5.5	101
3.5 • 4.5	100
2.5 • 3.5	011
1.5 • 2.5	010
0.5 • 1.5	001
0 • 0.5	000

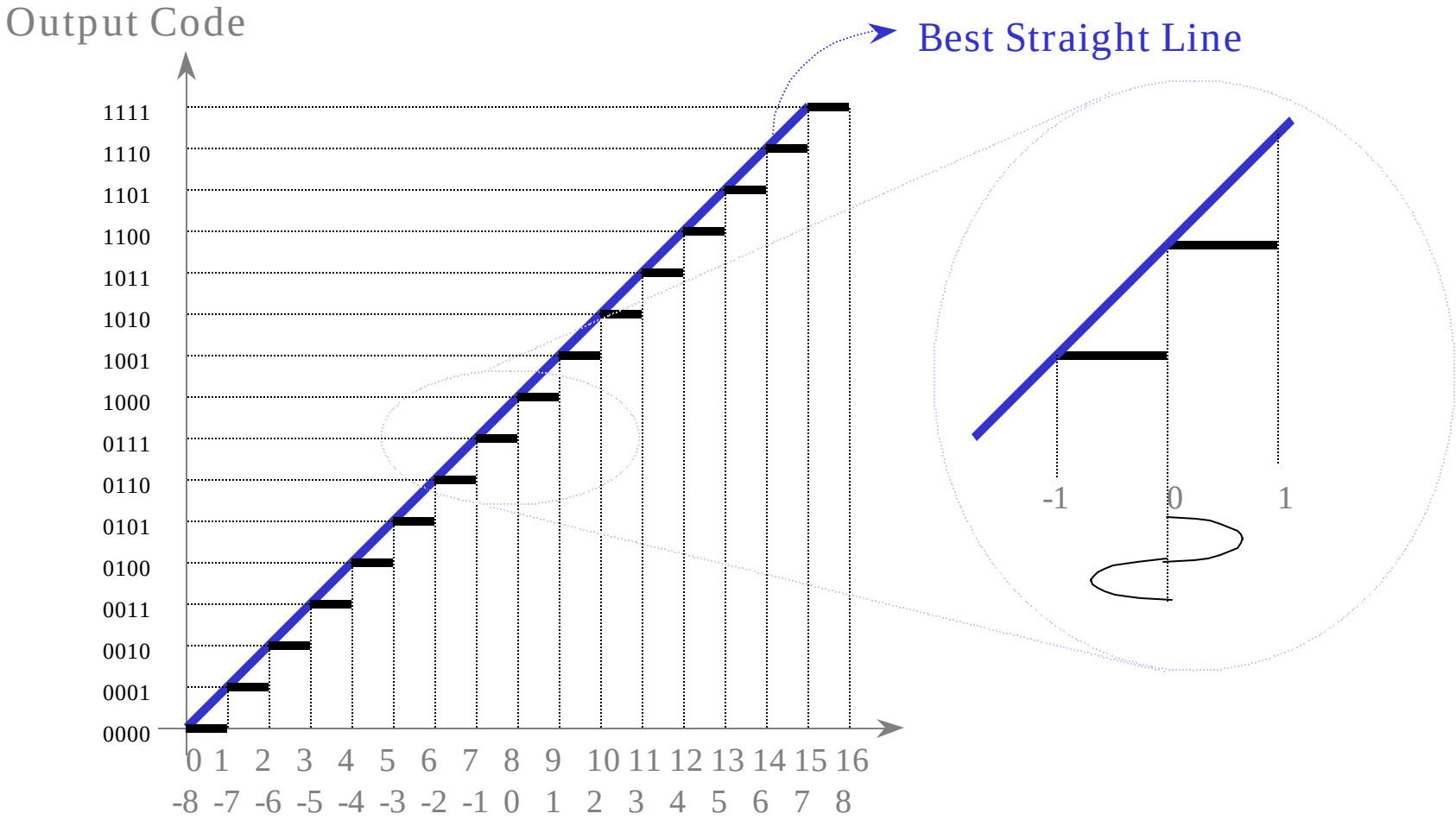


Elements of Transfer Diagram for an Ideal Linear ADC

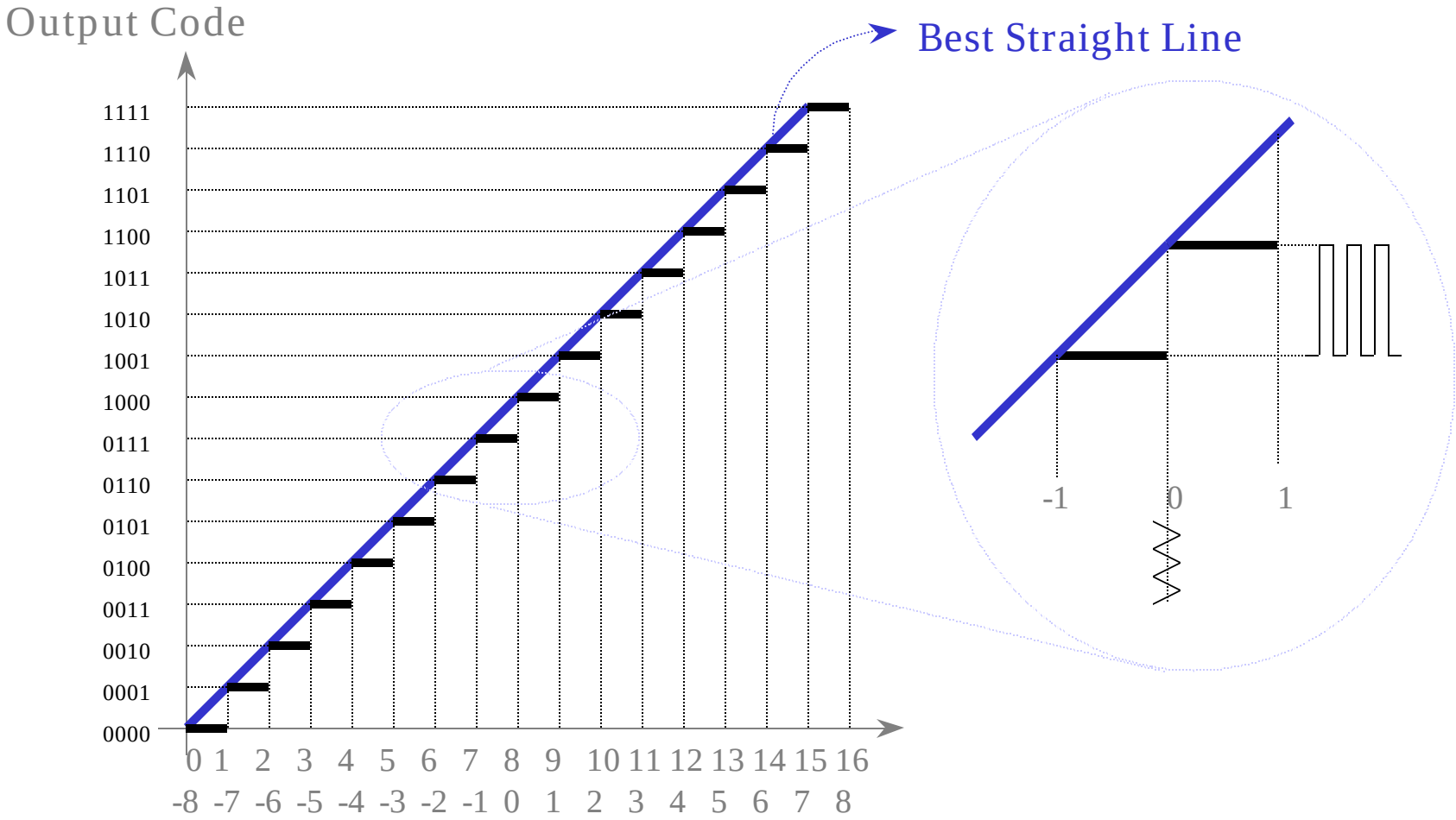
Ideal Transfer Characteristic



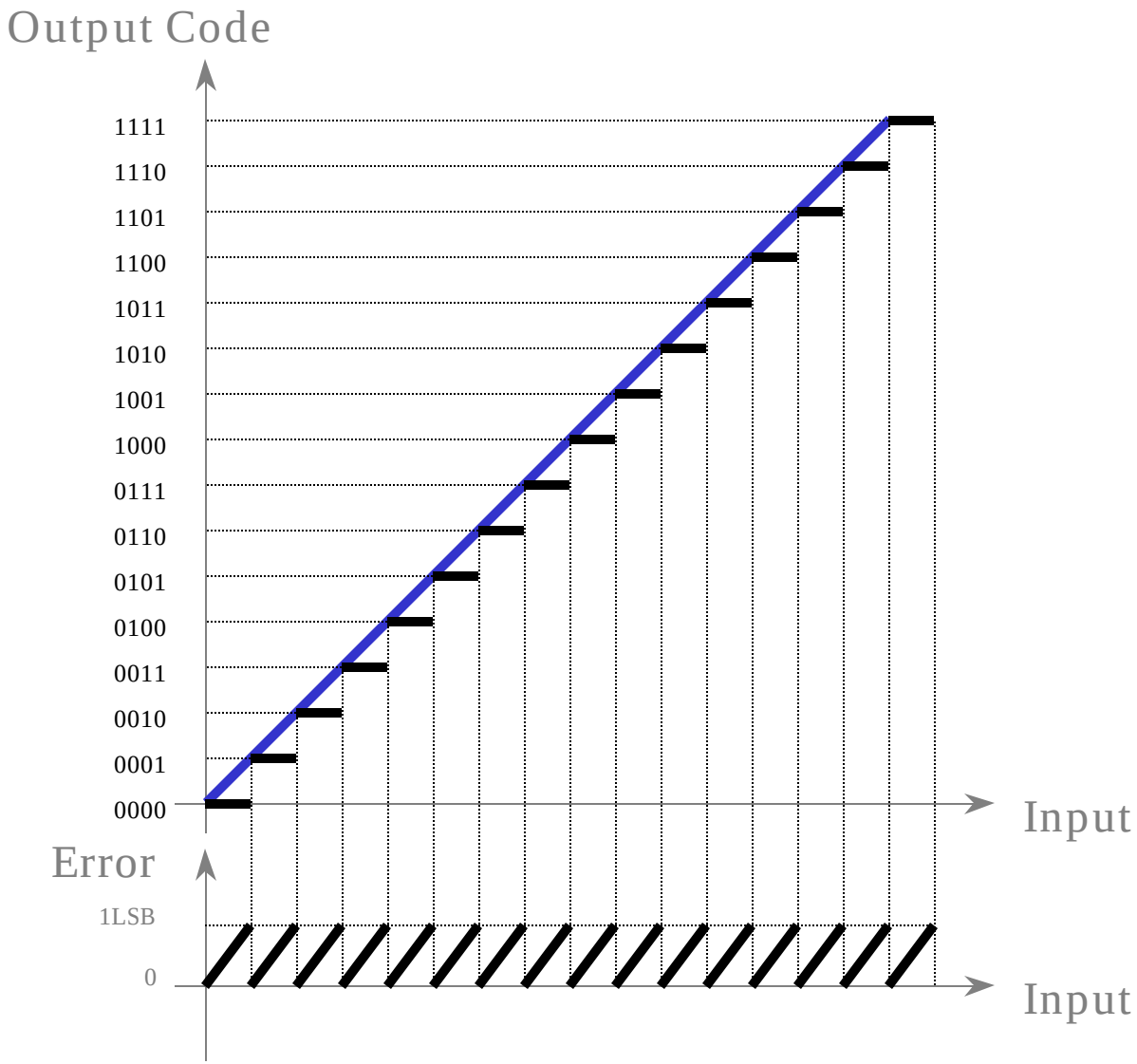
Ideal Transfer Characteristic



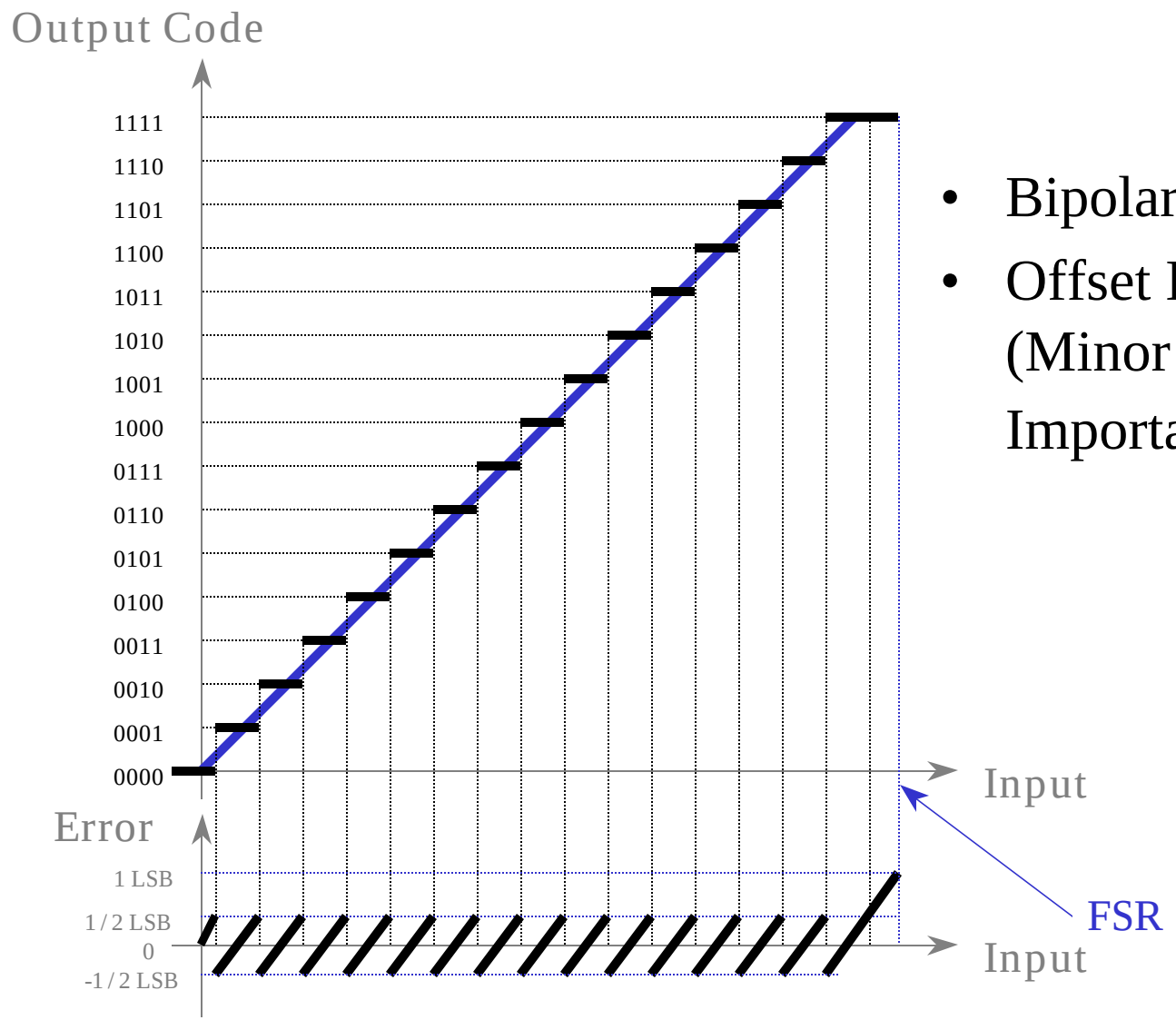
Ideal Transfer Characteristic



Unipolar Quantization Error



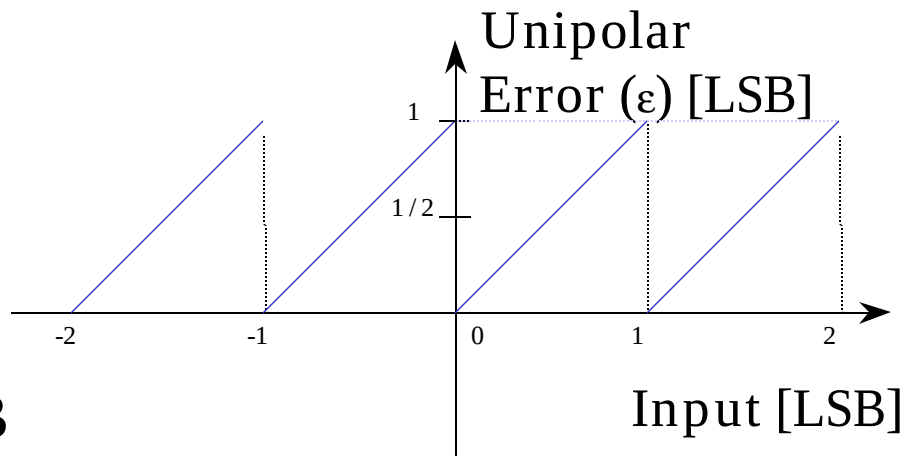
Bipolar Quantization Error



- Bipolar Error
- Offset Error (Minor Importance)

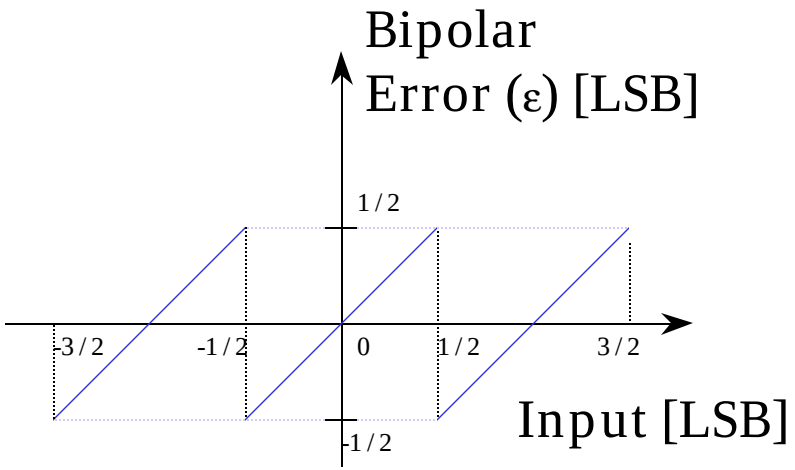
Unipolar vs. Bipolar

- Quantization Noise Power:
  $LSB^2/3$
- RMS Value of Quantization Noise Power:
  $LSB/1.73$
- More Than Half an LSB error.



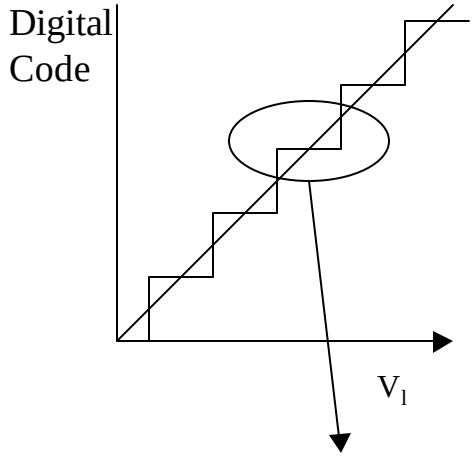
Unipolar vs. Bipolar

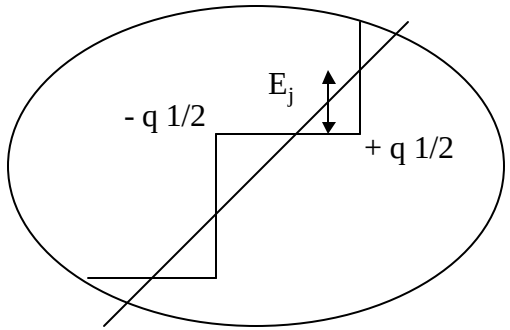
- Quantization Noise Power:
  $LSB^2/12$
- RMS Value of Quantization Noise Power:
  $LSB/3.46$
- Approximately One Third of an LSB.

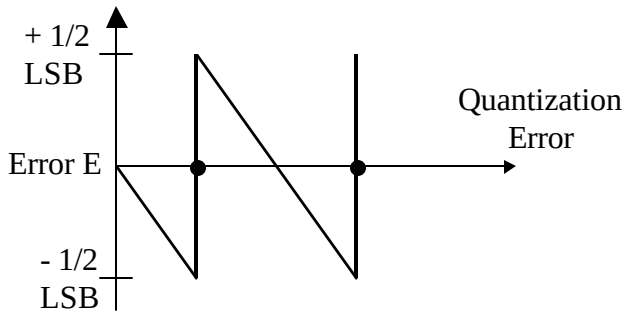


Reference: Spectra of Quantized Signals, W.R.Bennett, BSTJ, July 1948.

QUANTIZATION EFFECTS







Error at the jth step:

$$E_j = (V_j - V_l)$$

The mean square error over the step is:

$$\overline{E_j}^2 = \frac{1}{q_1}\int_{-q/2}^{+q/2} E_j^2 dE = \frac{q^2}{12}$$

Assuming equal steps, the
total error is:

$$\overline{N}^2 = q^2/12$$

(Mean square quantization noise)

QUANTIZATION EFFECTS

Considering a sine wave input F(t) of amplitude A so that $F(t) = A \sin \omega t$

which has a mean square value of $F^2(t)$, where $F^2(t) = \frac{1}{2\pi} \int_0^{2\pi} A^2 \sin^2(\omega t) dt$

which is the signal power. Therefore the signal to noise ratio SNR is given by

$$SNR(dB) = 10 Log \left[\left(\frac{A^2}{2} \right) \Big/ \left(\frac{q^2}{12} \right) \right] \quad \text{but} \quad q = 1\,LSB = \frac{2A}{2^n} = \frac{A}{2^{n-1}}$$

Substituting for q gives

$$SNR(dB) = 10\ Log \left[\left(\frac{A^2}{2} \right) \Big/ \left(\frac{A^2}{3 * 2^{2n}} \right) \right] = 10\ Log \left(\frac{3 * 2^{2n}}{2} \right)$$

$$\Rightarrow \underline{\underline{6.02n + 1.76dB}}$$

This gives the ideal value for an n bit converter and shows that each extra 1 bit of resolution provide approximately 6 dB improvement in the SNR.

In practice, integral and differential non-linearity (discussed later in this presentation) introduce errors that lead to a reduction of this value. The limit of a 1/2 LSB differential linearity error is a missing code condition which is equivalent to a reduction of 1 bit of resolution and hence a reduction of 6 dB in the SNR. This then gives a worst case value of SNR for an n-bit converter with 1/2 LSB linearity error

$$SNR\ (\text{worst case}) = 6.02n + 1.76 - 6 = 6.02n - 4.24dB$$

Thus, *we can established the boundary conditions for the choice of the resolution of the converter based upon a desired level of SNR.*

Signal to Noise Ratio (SNR)

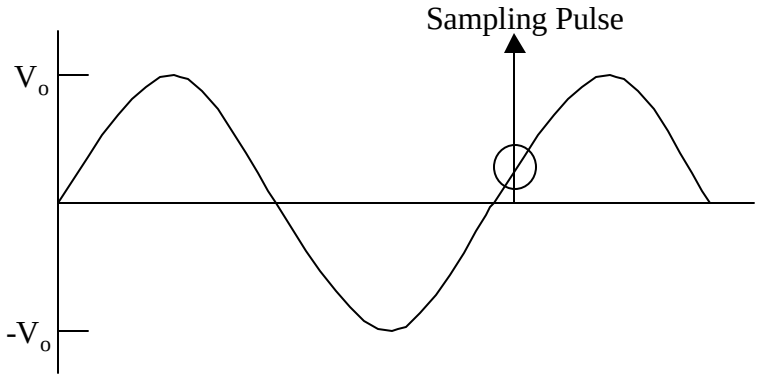
- $V_{in}=A\ \mathrm{Sin}(\omega T)$; $A = V_{FSR} / 2$
- Signal Power:

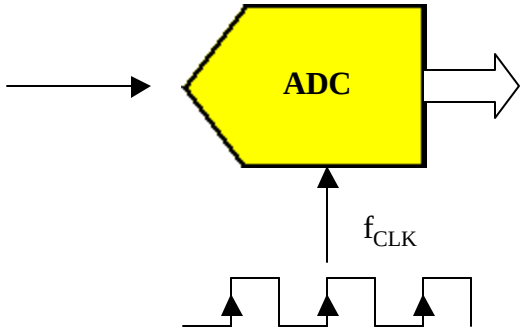
  $V_S^2 = (V_{FSR} / 2.8)^2 = V_{FSR}^2/8$
- Noise Power:

  $V_N^2 = LSB^2/12$
- SNR = $(1.5)2^{2N}$

  $1.8 + 6.02\ N\ [dB]$
- Example: SNR(10bit) = 62dB

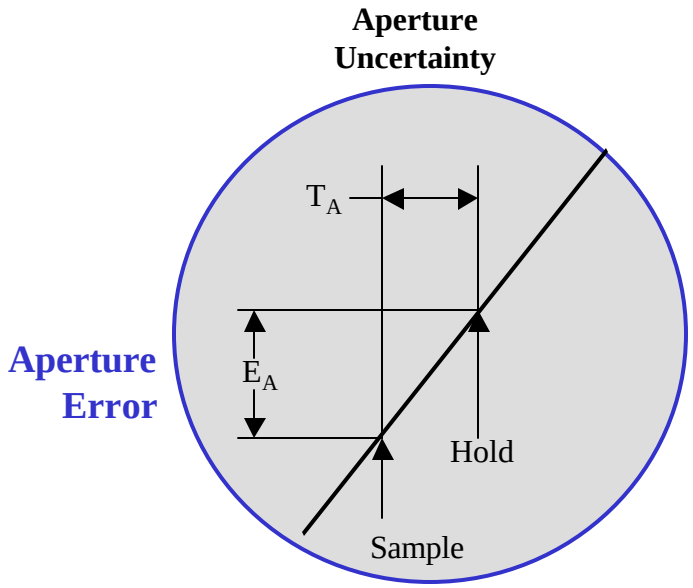
APERTURE ERROR





The aperture error comes from the fact that there is a delay between the clock signal and the effective holding time.

$$V = V_O \sin 2\pi ft$$



$$\frac{dV}{dt} = 2\pi fV_O \cos 2\pi ft \qquad \left.\frac{dV}{dt}\right|_{max} = 2\pi fV_O$$

$$E_A = T_A \frac{dV}{dt} = 1/2\ LSB = \frac{2V_O}{2^{n+1}}$$

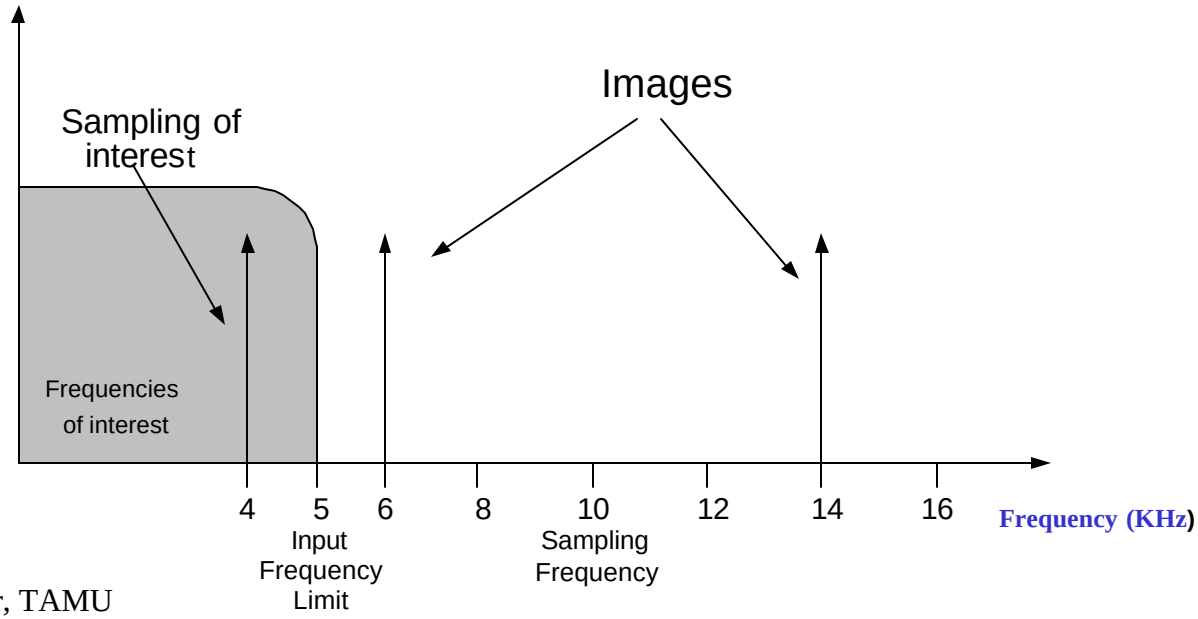
$$\frac{2V_O}{2^{n+1}} = 2\pi fV_O T_A \Rightarrow$$

$$T_A < \frac{V_{LSB}}{\pi f V_{ref}} = \frac{1}{2^N \pi f_{in}}$$

Nyquist Rate

- According to signal processing theory, the sampling process generates images of the input signal around the sampling frequency

- It can be seen that if the input frequency is higher than half the sampling frequency, there will be corruption of the information by the image.



Oversampling

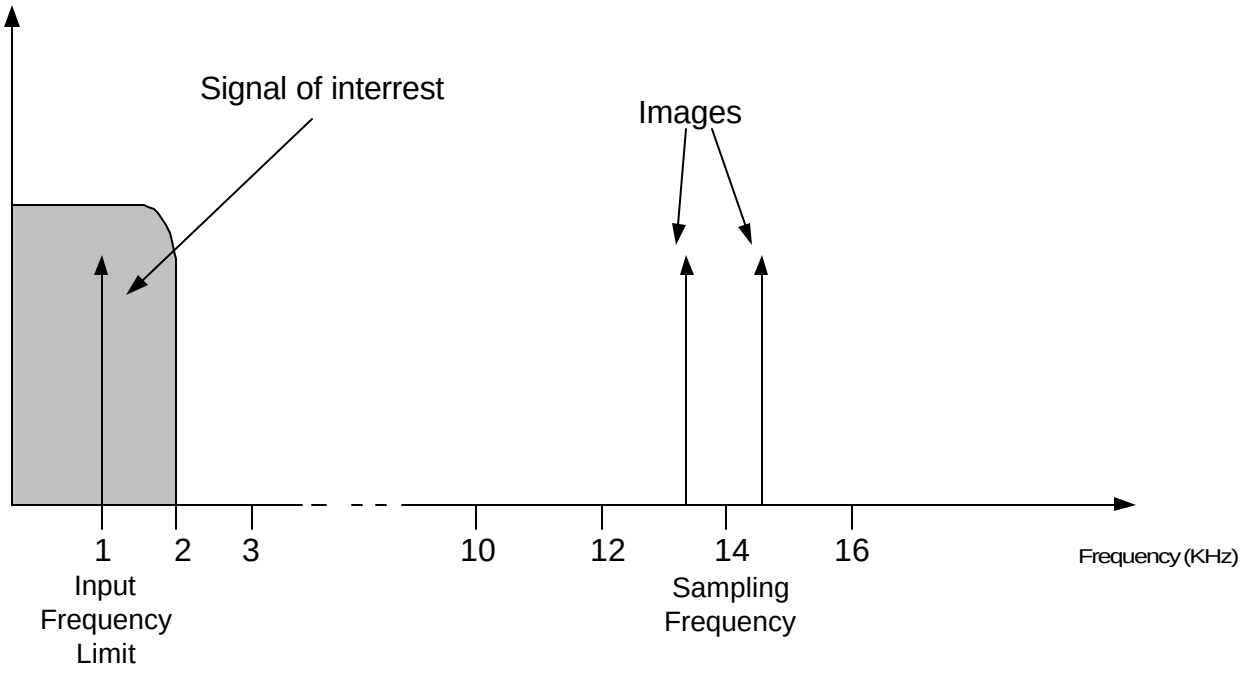
- As we have seen earlier, the SNR of a typical ADC is:

$$6.02n + 1.76dB$$

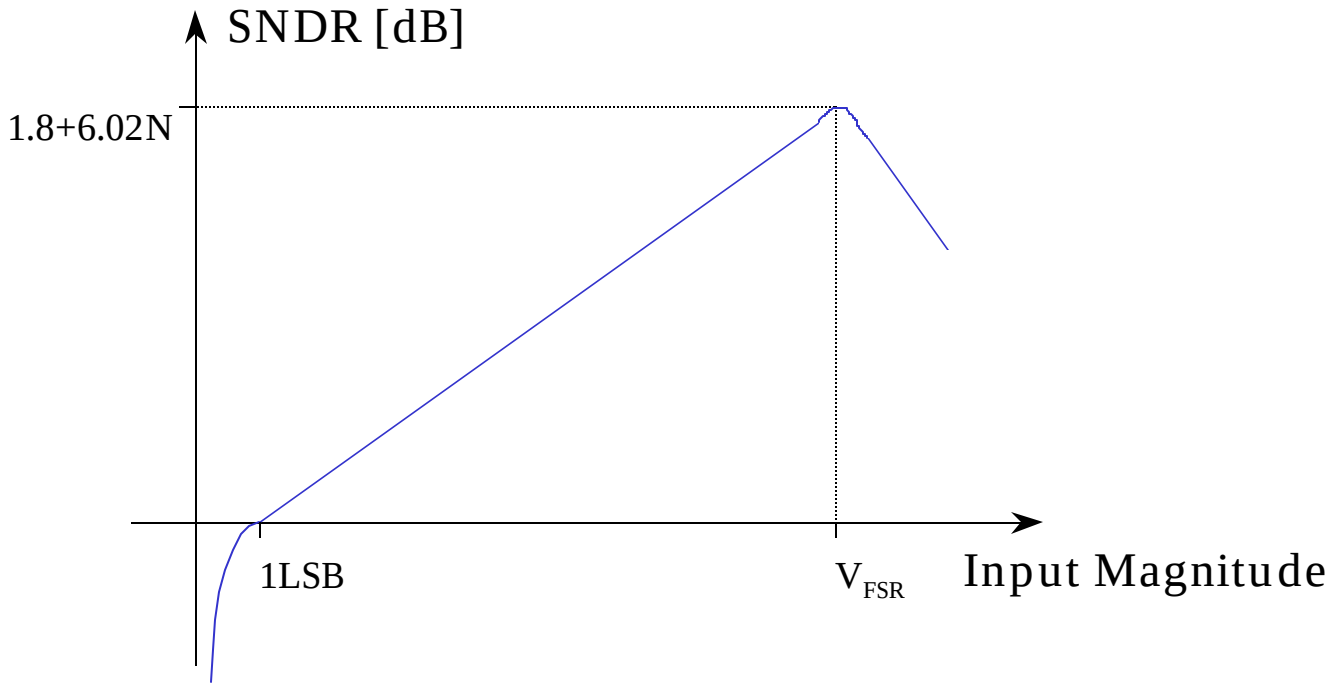
- If the sampling rate is increased, we get the following SNR:

$$6.02n+1.76dB+10\log(OSR)$$

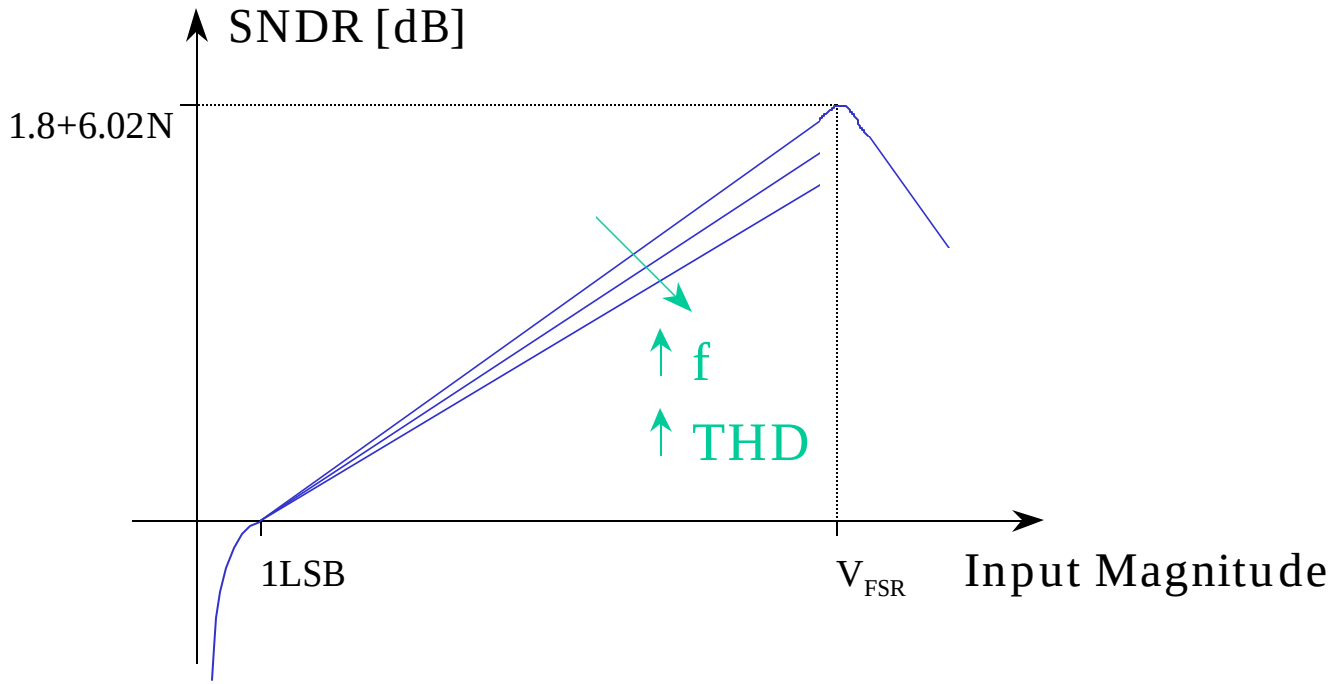
where OSR stands for "***oversampling ratio***".



Signal to Noise + Distortion Ratio (SNDR)

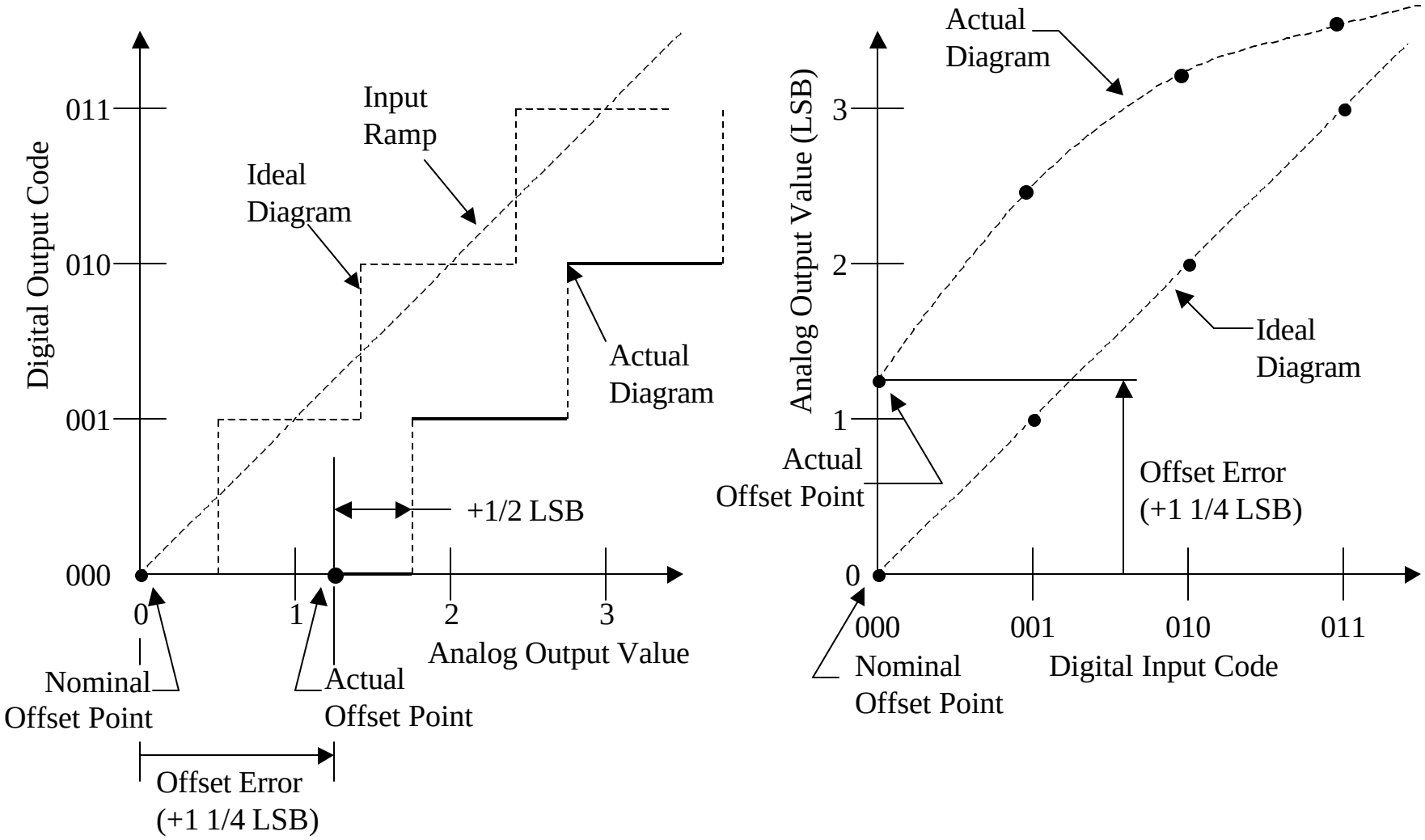


Signal to Noise + Distortion Ratio (SNDR)



Performance Evaluation of ADCs

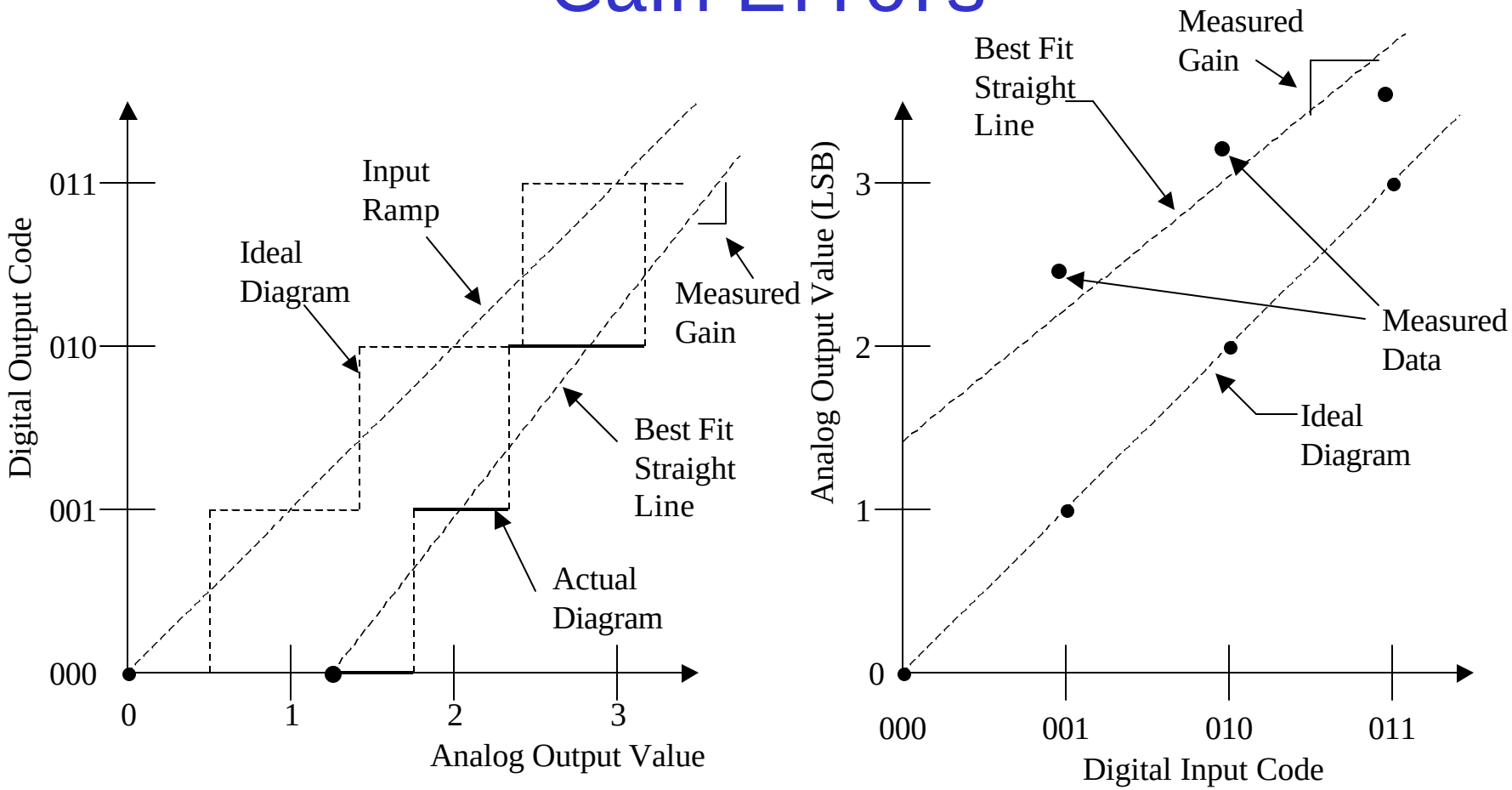
Offset Errors



(a) ADC

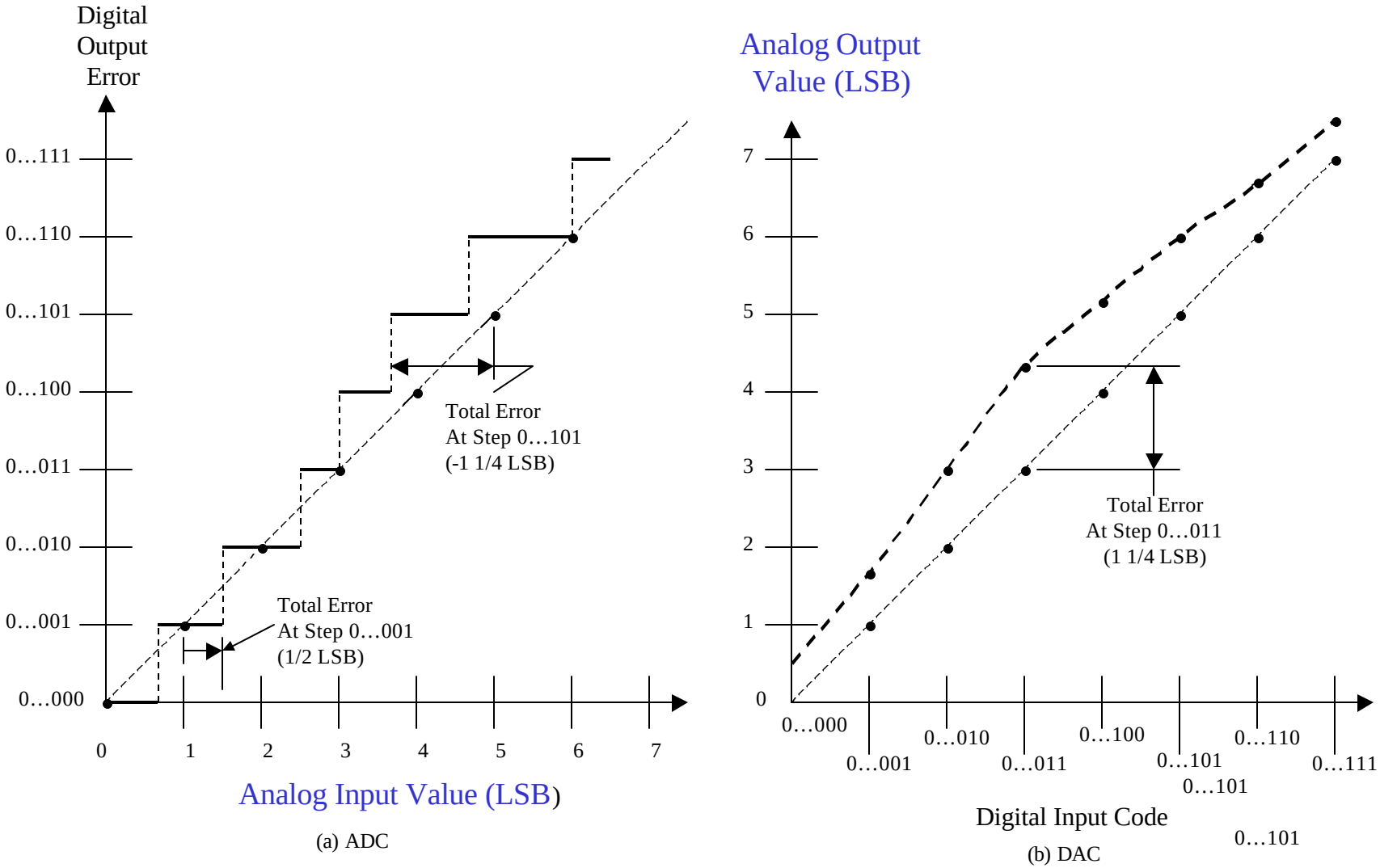
(b) *DAC*

Gain Errors



(a) ADC (b) *DAC*

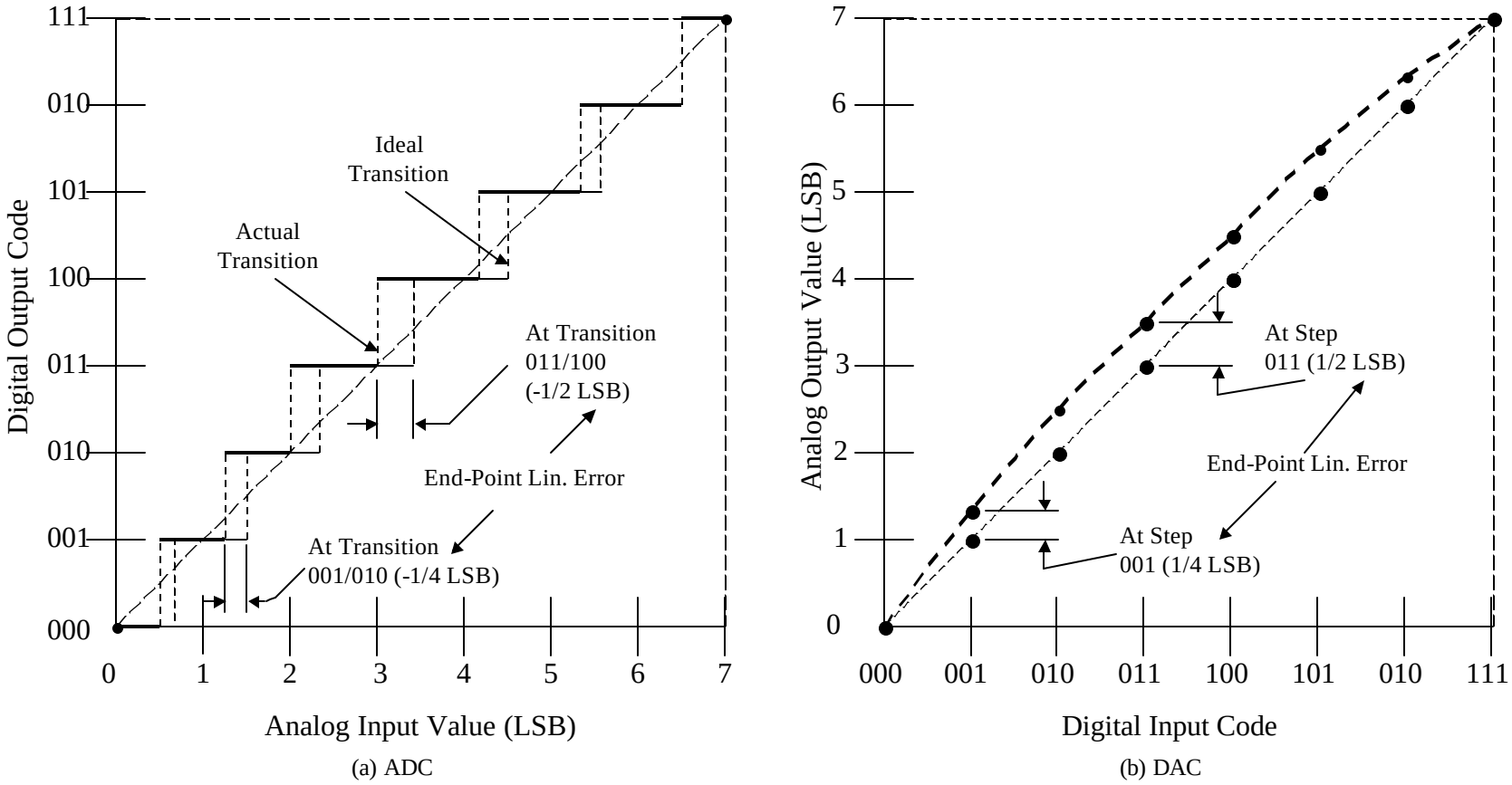
Absolute Accuracy (Total) Error



The absolute accuracy or total error of an ADC as shown in Figure is the maximum value of the difference between an analog value and the ideal midstep value. It includes offset, gain, and integral linearity errors and also the quantization error in the case of an ADC.

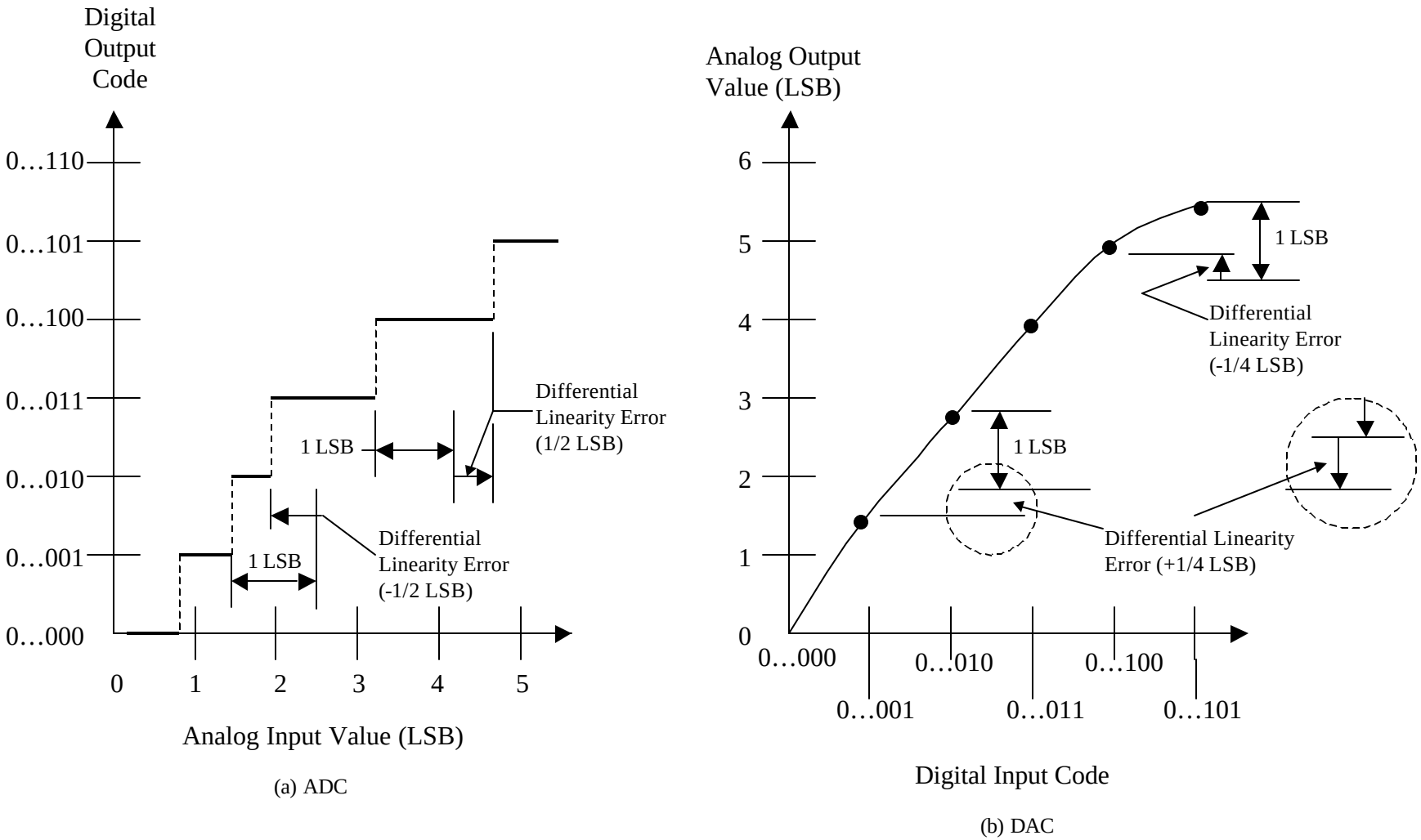
Integral Nonlinearity (INL) Error

The integral non-linearity depicts a possible distortion of the input-output transfer characteristic and leads to harmonic distortion.



End-Point Linearity Error of a Linear 3-bit Natural Binary-Coded ADC or DAC
(Offset Error and Gain Error are Adjusted to the Value Zero)

Differential Nonlinearity (DNL)



Differential Linearity Error of a Linear ADC or DAC

Numerical Examples [1]

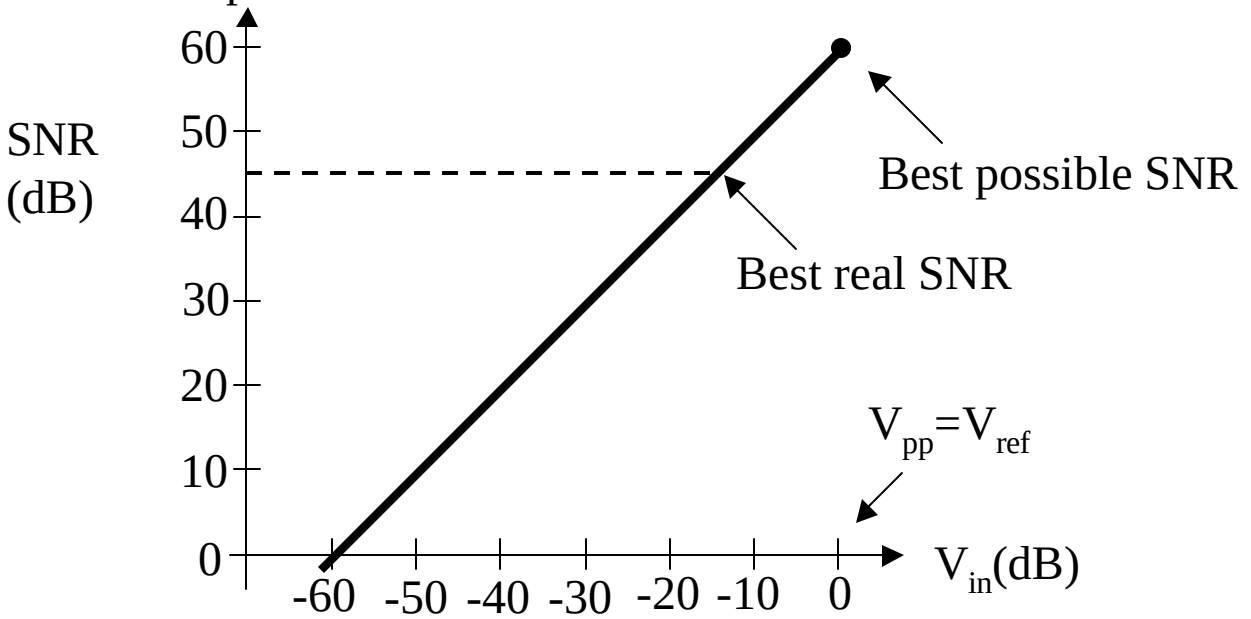
Example 1.

A 100-mV_{pp} sinusoidal signal is applied to an ideal 12-bit A/D converter for which V_{ref}= 5 V. Find the SNR of the digitized output signal.

$$\text{SNR} = 20\log\left(\frac{V_{in}(\text{rms})}{V_Q(\text{rms})}\right) = 20\log\left(\frac{V_{ref}/(2\sqrt{2})}{V_{LSB}/\sqrt{12}}\right) = 20\log\left(\sqrt{\frac{3}{2}}2^N\right)\pm$$

Solution

First, we use to find the maximum SNR if a full-scale sinusoidal wave-form of 2.5 V were applied to the input.



Idealized SNR versus sinusoidal input signal amplitude for a 10-bit A/D converter. The 0-dB input signal amplitude corresponds to a peak-to-peak voltage equaling V_{ref}.

$$\text{SNR}_{max} = 6.02N + 1.76\,\text{dB} = 6.02\times 12 + 1.76 = 74\,\text{dB}$$

However, since the input is only a $\pm$100-mV sinusoidal waveform that is 28 dB below full scale, the SNR of the digitized output is

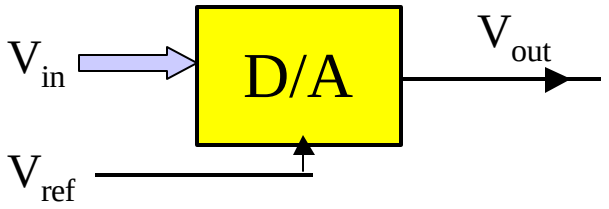
$$\text{SNR} = 74 - 28 = 46\,\text{dB}$$

Example 2. $V_{LSB} = \frac{V_{ref}}{2^N}$, $1\,\text{LSB} = \frac{1}{2^N}$

Consider a 3-bit D/A converter in which V_{ref} = 4 V, with the following measured voltage values:

{ 0.011 : 0.507 : 1.002 : 1.501 : 1.996 : 2.495 : 2.996 : 3.491 }

1. Find the offset and gain errors in units of LSBs.
2. Find the INL (endpoint) and DNL errors (in units of LSBs).
3. Find the effective number of bits of absolute accuracy.
4. Find the effective number of bits of relative accuracy.



Solution

We first note that 1 LSB corresponds to $V_{ref}/2^3 = 0.5$ V.

1. Since that offset voltage is 11 mV, and since 0.5 V corresponds to 1 LSB, we see that the offset error is given by

$$E_{off\ (D/A)} = \left.\frac{V_{out}}{V_{LSB}}\right|_{0\ldots0} = \frac{0.011}{0.5} = 0.022\ \text{LSB}$$

For the gain error, from (11.25) we have

$$E_{gain\,(D/A)} = \left(\left.\frac{V_{out}}{V_{LSB}}\right|_{1\ldots1} - \left.\frac{V_{out}}{V_{LSB}}\right|_{0\ldots0}\right) - (2^N - 1) = \left(\frac{3.491 - 0.011}{0.5}\right) - (2^3 - 1) = -0.04\ \text{LSB}$$

2. For INL and DNL errors, we first need to remove both offset and gain errors in the measured D/A values. The offset error is removed by subtracting 0.022 LSB off each value, whereas the gain error is eliminated by subtracting off scaled values of the gain error. For example, the new value for 1.002 (scaled to 1 LSB) is given by

$$\frac{1.002}{0.5} - 0.022 + \left(\frac{2}{7}\right)(0.04) = 1.993$$

Thus, the offset-free, gain-free, scaled values are given by

{ 0.0 : 0.998 : 1.993 : 2.997 : 3.993 : 4.997 : 6.004 : 7.0 }

Since these results are in units of LSBs, we calculate the INL errors as the difference between these values and the ideal values, giving us

INL errors : { 0 : -0.002 : -0.007 : -0.003 : -0.007 : -0.003 : 0.004 : 0 }

For DNL errors, we find the difference between adjacent offset-free, gain-free, scaled values to give

DNL errors: { -0.002 : -0.005 : 0.004 : -0.004 : 0.004 : 0.007 : -0.004 }

3. For absolute accuracy, we find the largest deviation between the measured values and the ideal values, which, in this case, occurs at 0 V and is 11 mV. To relate this 11-mV value to effective bits, 11 mV should correspond to 1 LSB when V_{ref} = 4 V. In other words, we have the relationship

$$\frac{4\text{V}}{2^{N_{eff}}} = 11\text{mV}$$

which results in an absolute accuracy of N_{abs} = 8.5 bits.

4. For relative accuracy, we use the INL errors found in part 2, whose maximum magnitude is 0.007 LSB, or equivalently, 3.5 mV. We relate this 3.5-mV value to effective bits in the same manner as in part 3, resulting in a relative accuracy of N_{rel} = 10.2 nits.

Example 3 [Johns & Martin]

A full-scale sinusoidal waveform is applied to a 12-bit A/D converter, and the output is digitally analyzed. If the fundamental has a normalized power of 1 W while the remaining power is 0.5 μW, what is the effective number of bits for the converter?

Solution

Using the expression for the SNR, we have

$$\text{SNR} = 6.02\,\text{N}_{\text{eff}} + 1.76$$

In this case, the signal-to-noise ration is found to be

$$\text{SNR} = 10\log\left(\frac{\text{P}_{\text{fund.}}}{\text{P}_{\text{rema}}}\right) = 10\log\left(\frac{1}{0.5\times 10^{-6}}\right) = 63\,\text{dB}$$

Substituting this SNR value into the SNR expression yields

$$\text{N}_{\text{eff}} = \frac{63 - 1.76}{6.02} = 10.2 \;\; \text{effective} \;\; \text{bits}$$

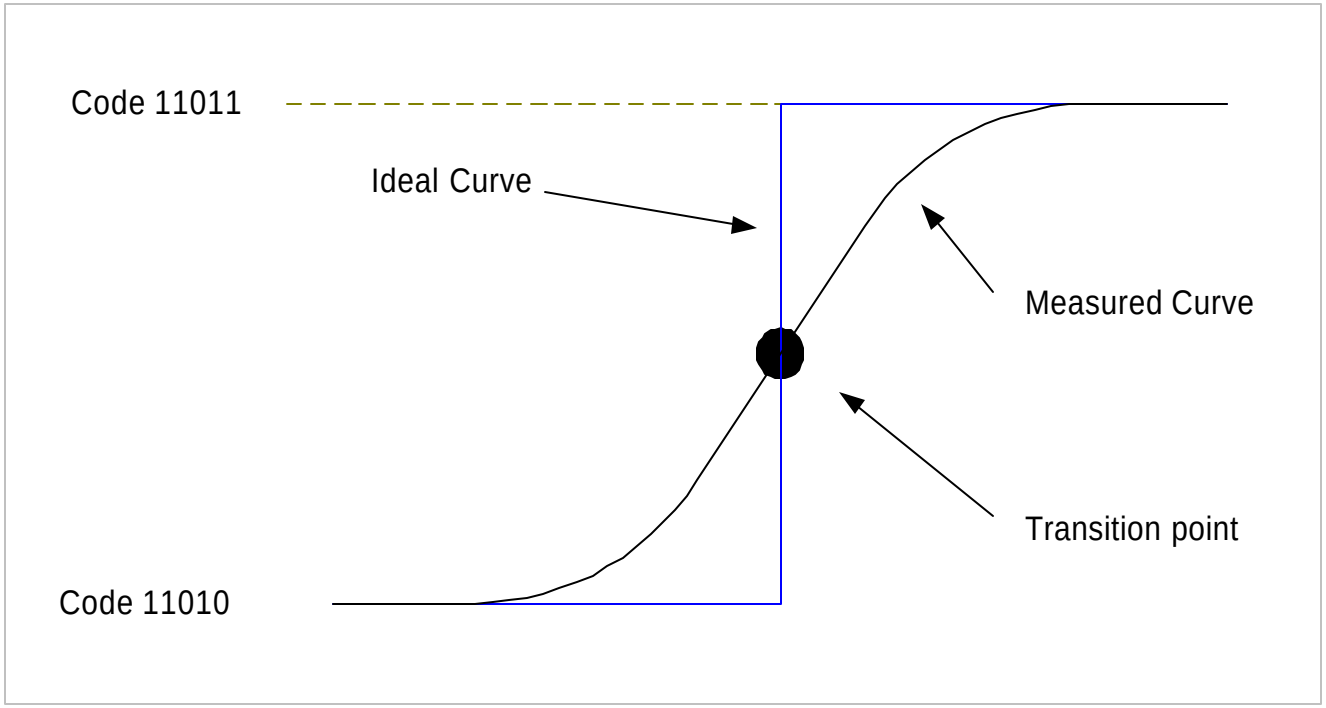
Measure Static Performance

The first step is to find each transition point;

- The real transition is not instantaneous. The transition point is half way between 2 consecutive codes
- Once all the transition points are recorded, the static parameters can be computed by a set of equations
- three different static performance measurement methods are described

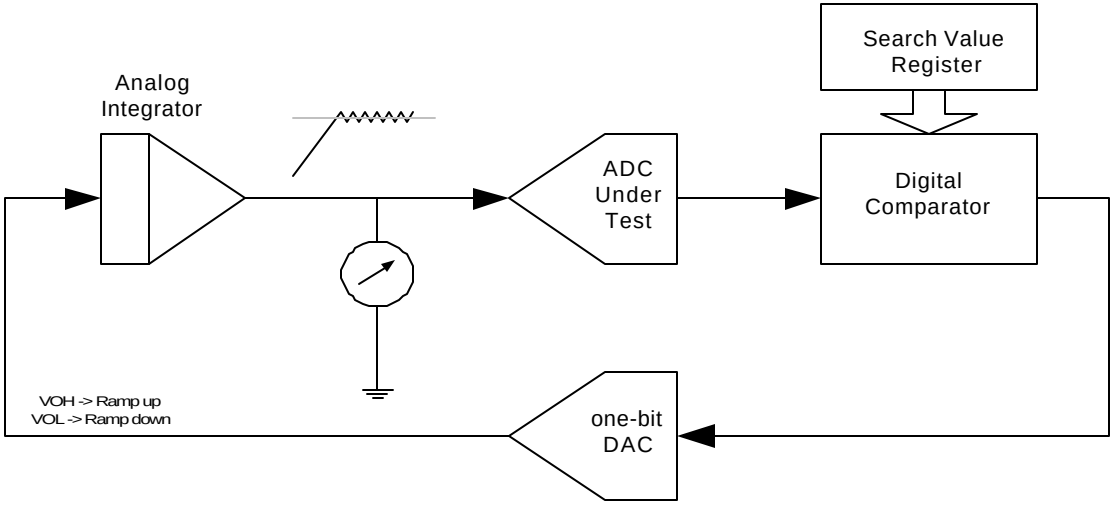
Method 1:manual measurement

- Increase the analog input to the ADC slowly until we can make the digital output 1 LSB more,from 11010 to 11011
- write down the correspond analog input value V1
- decrease the analog input to the ADC slowly until we can make the digital output return to the 11010
- write down the correspond analog input value V2
- The transition point between 11010 and 11011 is 0.5(V1+V2)



Method 2:The Servo Method

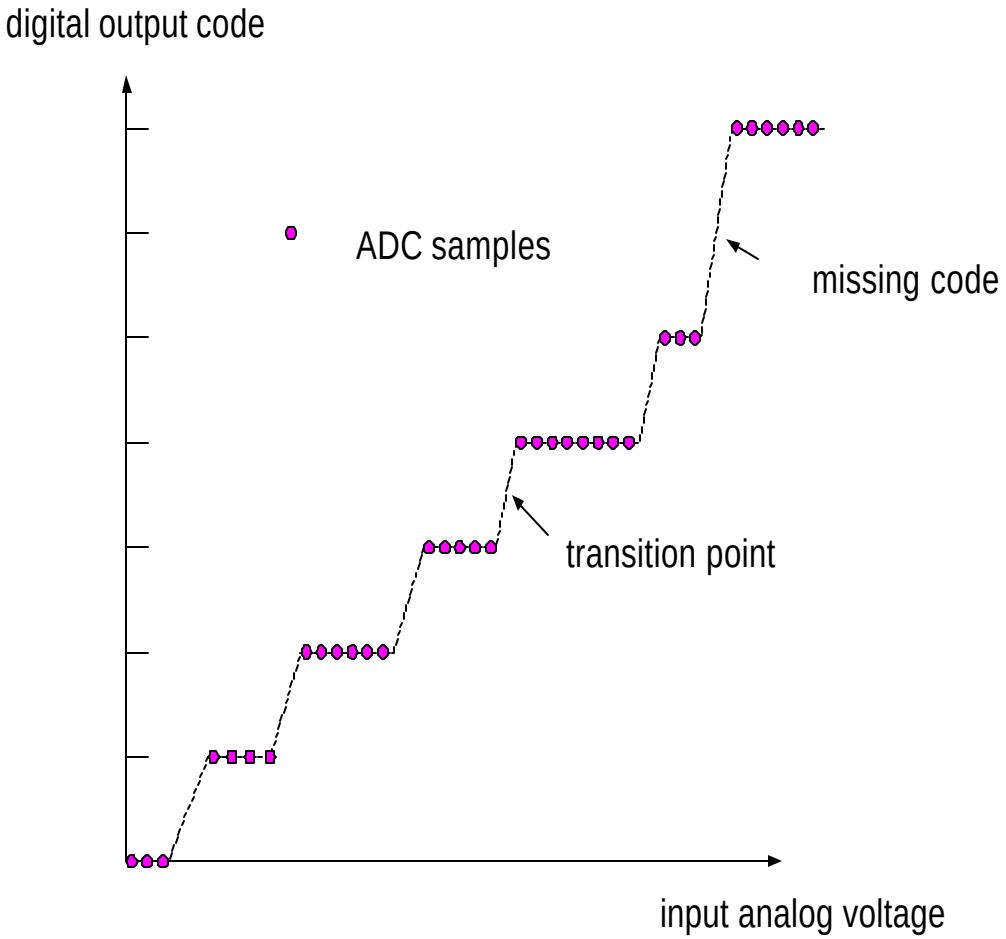
- The Servo method is an automated technique to easily find the transition points
- for example,we need the transition point between 11010 and 11011.We should set search value register as 11010
- close the loop;when the circuit is stable,use a DC voltmeter to measure the analog value at the output of the integrator.It is the transition point between 11010 and 11011



Method 3:The Linear Ramp Histogram

- The histogram is best suited for automated testing of ADCs in the industry
- A linear ramp is sent to the ADC under test and the output codes are sampled and recorded
- The input ramp must be very slow, such that we get at least 16 samples per output code
- This allows a precise evaluation of the static performances.

Method 3:The linear Ramp Histogram (continued)



- Record the samples per digital code
- use the recorded values to compute the transition point one by one

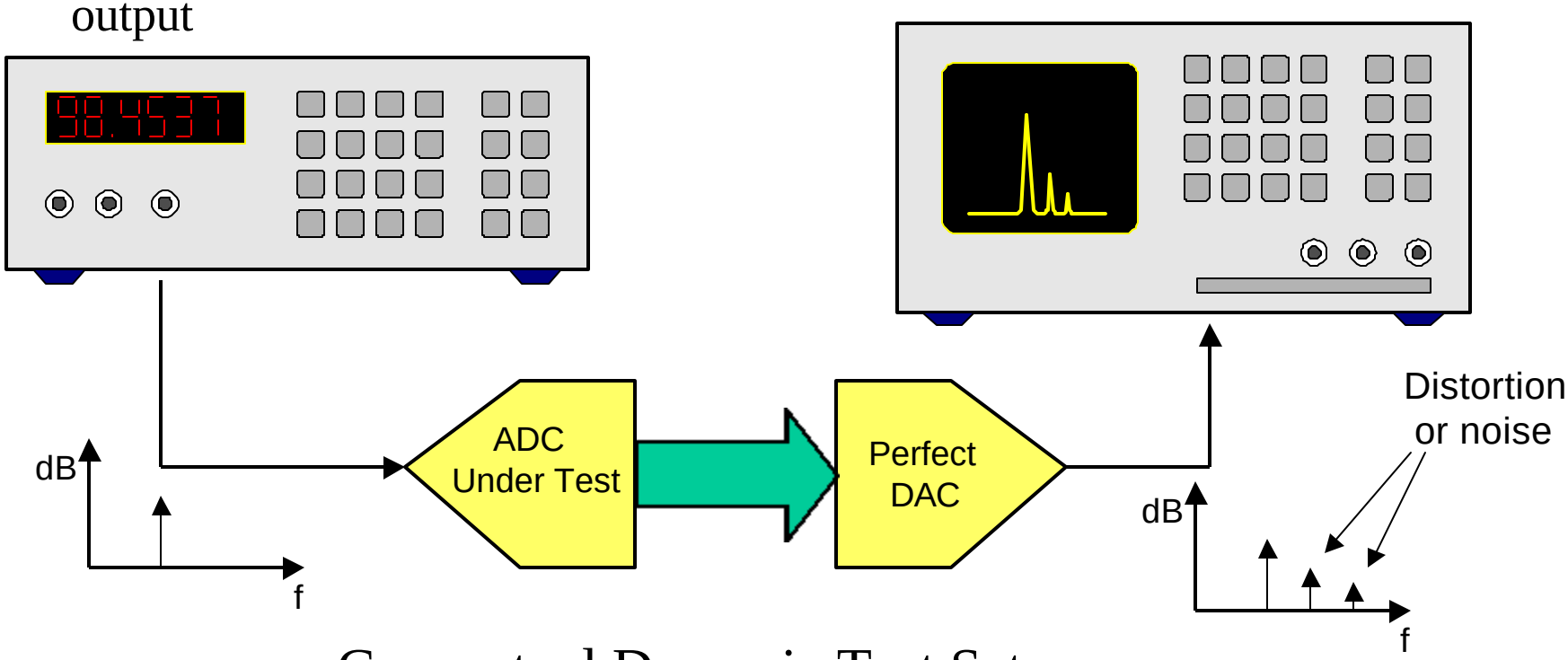
Calculate static parameters from transition points

- Use TP as the symbol of the transition point,and assume TP[i] is the transition point between code i-1 and code I
- Offset = TP[1]-0.5 $\left(\frac{FSR}{2^N}\right)$ FSR is the full scale input range; N is ADC resolution
- Gain Error= $\left|\left(\frac{TP[2\text{^}N-1]-TP[1]}{FSR\times\left[\frac{2\text{^}N-2}{2\text{^}N}\right]}\right)-1\right|\times 100$
- Differential Non-linearity
- DNL[i]= $\frac{TP[i+1]-TP[i]}{LSB}$ -1
- Integral Non-linearity

INL[i]= $\frac{TP[i]-(LSB\times(i-1)+TP[1])}{LSB}$

Dynamic Performance Measurement

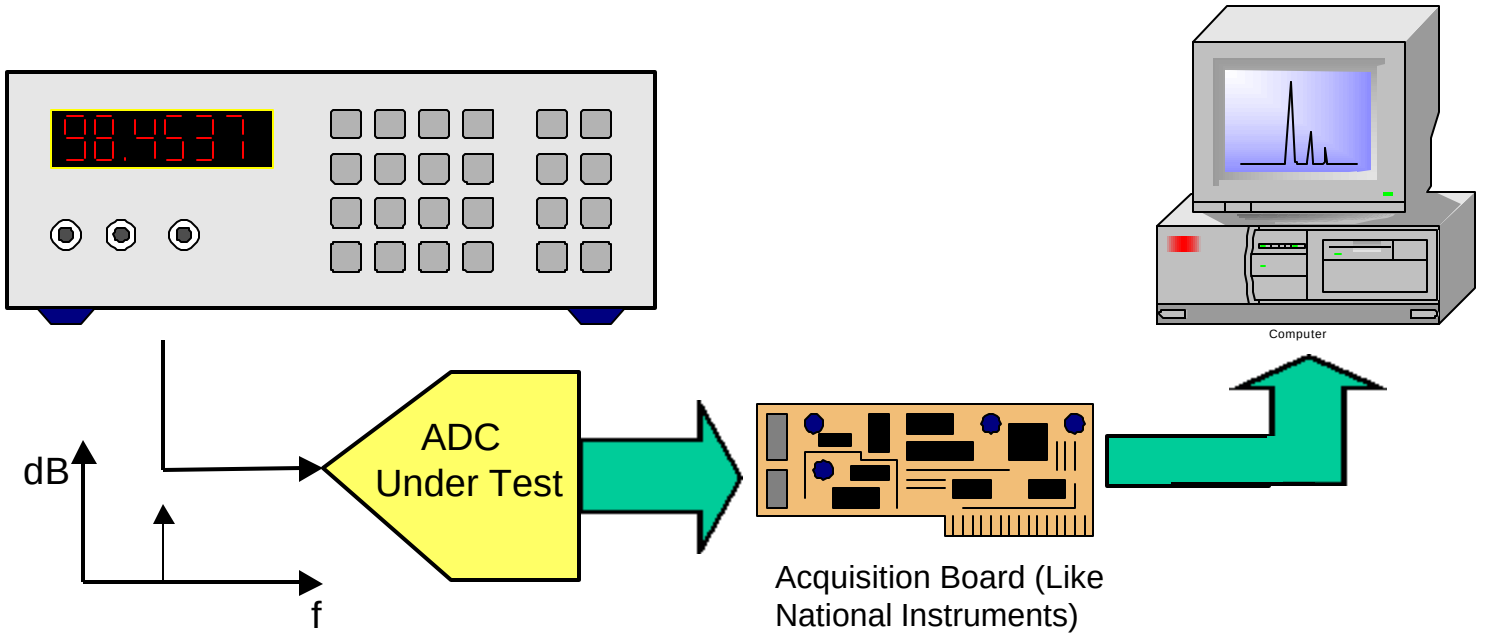
- The most typical dynamic performance measurement consists of looking for distortion in the frequency domain
- This is done by sending a pure sinusoidal input and looking at the output



Conceptual Dynamic Test Setup

Real Measurement of Dynamic Performance

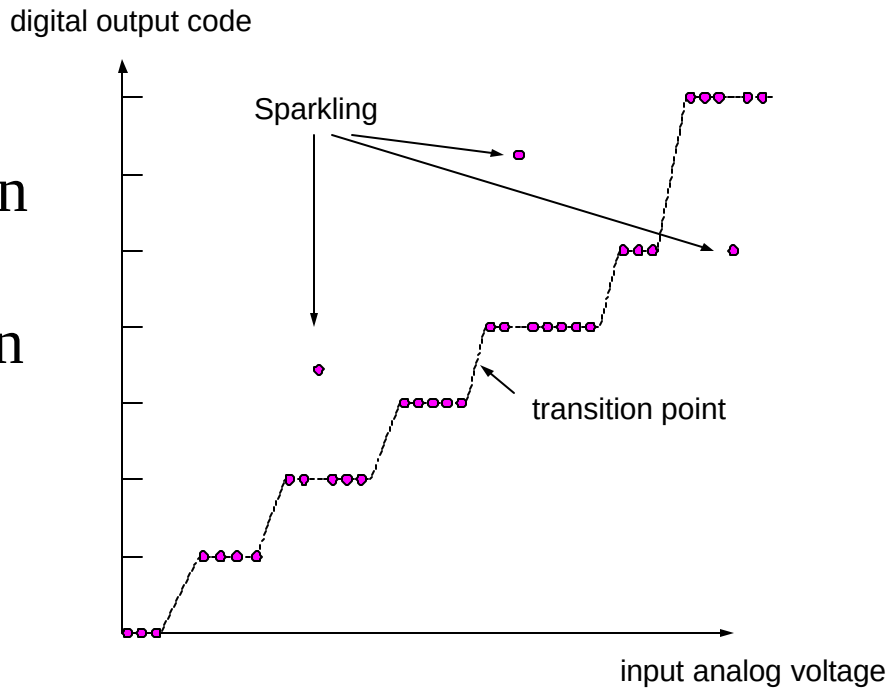
- There is no such thing as a perfect DAC…
- To improve the precision and simplify the post-processing, all the spectrum analysis is done in digital form and in software.



Other Dynamic Measurements

- All the timing and control signals (i.e. Convert, Data_Ready, Read, Data, …) must be tested at full speed to ensure their functionality,

- The output from a sinewave input can also be observed in time-domain to make sure there is no sparkling (sudden out-of-range samples).



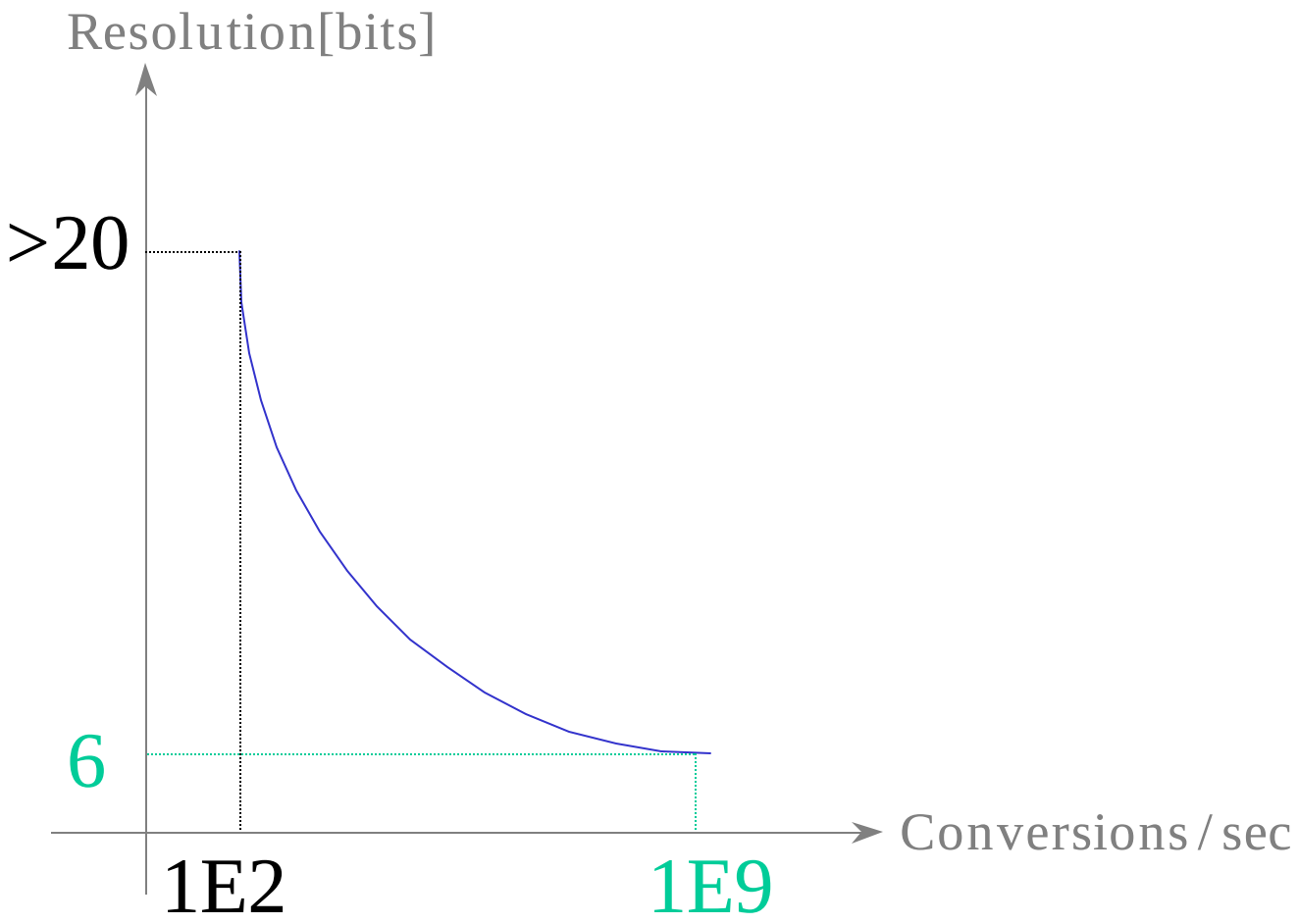
An illustrative example/comparison:

A/D Conversion : Practical Techniques

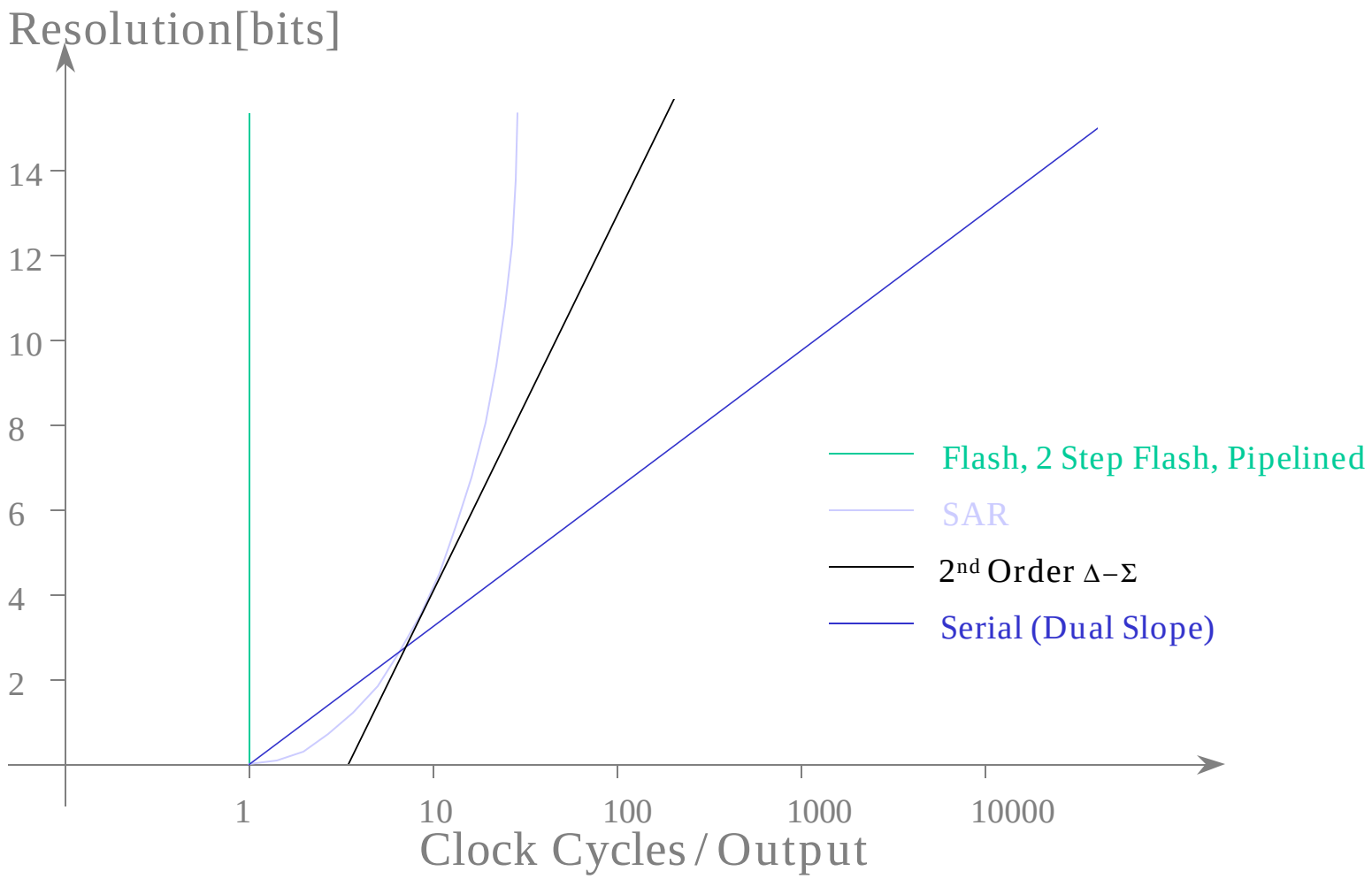
- Serial Conversion:
  Dual Slope
- Successive
  Approximation

- Parallel Conversion:
  Flash
- Quantized Feedforward:
  Pipelined
- Quantized Feedback:
  Delta-Sigma

Speed vs. Resolution



Throughput Rate Comparison of ADCs



An A/D Conversion Classification

Multiplexing

- Time Interweave
- Pipeline

Parallel

- Flash
- Stacking Flash
- Neural Network

Series-Parallel

- Subranging
- Subranging with Folding Amps
- Ripple

Serial

- Non-Algorithmic
  - Successive Approximation
  - Charge Redistribution
  - Variable Ref. Serial Ripple
- Algorithmic Iterative
  - Cyclic
  - Sample/Hold
- Algorithmic Replicative
  - Straight Binary
  - Gray

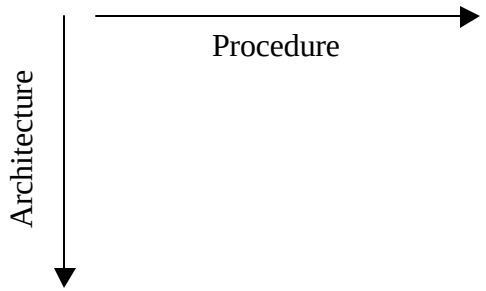
Coarse-Fine

- Coarse-Fine

Counting

- Pulse Width
  - Ramp Comparison
  - Dual Slopes
  - Constant Slope
- Pulse Rate
  - Ramp Comparison
  - Quantized Feedback
- Tracking Feedback
  - Servo
  - Delta Modulation

Note: The procedure and architecture are shown as



Coarse List of A/D Converter Architectures

Low-to-Medium Speed High-Accuracy	Medium Speed Medium Accuracy	High-Speed Low-to-Medium Accuracy
Integrating	Successive Approximation	Flash
Oversampling	Algorithmic	Two-Step
		Interpolating
		Folding
		Pipelined
		Time-Interleaved

References

[1] D.A. Johns and K. Martin, *Analog Integrated Circuit Design*, Chapters 11 and 12, John Wiley & Sons, Inc., New York 1997,.

[2] A.B. Grebene. *Bipolar and MOS Analog Integrated Circuit Design*, John Wiley & Sons, Inc., New York 1984.

[3] B. Razavi, *Principles of Data Conversion System Design*, The IEEE Press, New York 1995.

[4] A.M.J. Daanen, Classification of DA and AD Conversion Techniques, *Report of the Graduation Work*, Technical University, Eindhoven, Dec. 1986.

[5] P.E. Allen and E. Sánchez-Sinencio, *Switched Capacitor Circuits*, Van Nostrand Reinhold, New York 1984.

[6] J.A.Shoeff, "An Inherently Monotonic 12 Bit DAC," *IEEE J. Solid-State Circuits*, vol. SC-14, pp 904-911, Dec. 1979.

[7] R.H. Charles and D.A. Hodges, "Charge Circuits for Analog Circuits for Analog LSI," *IEEE Trans. Circuit and Systems*, vol. CAS-25, No. 7, pp 490-497, July 1978.

[8] J. Doernberg, H.S. Lee and D. Hodges, "Full Speed Testing of A/D Converters," *IEEE J. Solid-State Circuits*, vol. SC-19, No. 6, pp 820-827, Dec. 1984.

[8] Texas Instruments Application Report," *Understanding Data Converters"*, SLAA013, July 1995.

[9] J.C. Candy, and G. C. Temes, Editors. "*Oversampling Delta-Sigma Data Converters: Theory, Design and Simulation"* IEEE Press, New York 1992.

[10] J. E. Franca and Y. Tsividis, Editors, *Design of Analog-Digital VLSI Circuits for Telecommunications and Signal Processing*", Chapters 9 and 10,Prentice Hall, Englewood Cliffs,1994

[11] S. Franco, "*Design with Operational Amplifiers and Analog Integrated Circuits*", McGraw-Hill, Boston, 1998.

[12] M. Burns and G. Roberts, "*Introduction to Mixed-Signal Test and Measurement"*, to be published.

[13] G.J. Gomez, "*Introduction to the Design of Sigma-Delta Modulators*", Seminar document.

November 1999

National Semiconductor

ADC0801/ADC0802/ADC0803/ADC0804/ADC0805 8-Bit µP Compatible A/D Converters

General Description

The ADC0801, ADC0802, ADC0803, ADC0804 and ADC0805 are CMOS 8-bit successive approximation A/D converters that use a differential potentiometric ladder — similar to the 256R products. These converters are designed to allow operation with the NSC800 and INS8080A derivative control bus with TRI-STATE® output latches directly driving the data bus. These A/Ds appear like memory locations or I/O ports to the microprocessor and no interfacing logic is needed.

Differential analog voltage inputs allow increasing the common-mode rejection and offsetting the analog zero input voltage value. In addition, the voltage reference input can be adjusted to allow encoding any smaller analog voltage span to the full 8 bits of resolution.

Features

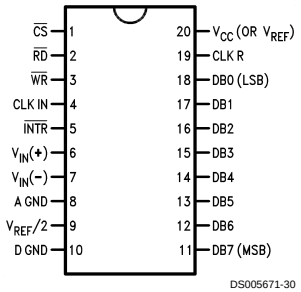
- Compatible with 8080 µP derivatives — no interfacing logic needed - access time - 135 ns
- Easy interface to all microprocessors, or operates "stand alone"
- Differential analog voltage inputs
- Logic inputs and outputs meet both MOS and TTL voltage level specifications
- Works with 2.5V (LM336) voltage reference
- On-chip clock generator
- 0V to 5V analog input voltage range with single 5V supply
- No zero adjust required
- 0.3" standard width 20-pin DIP package
- 20-pin molded chip carrier or small outline package
- Operates ratiometrically or with 5 V_{DC}, 2.5 V_{DC}, or analog span adjusted voltage reference

Key Specifications

- Resolution 8 bits
- Total error ±¼ LSB, ±½ LSB and ±1 LSB
- Conversion time 100 µs

Connection Diagram

ADC080X
Dual-In-Line and Small Outline (SO) Packages



DS005671-30

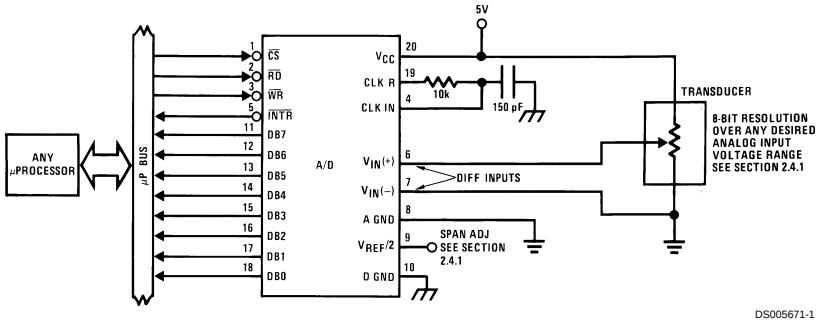
See Ordering Information

Ordering Information

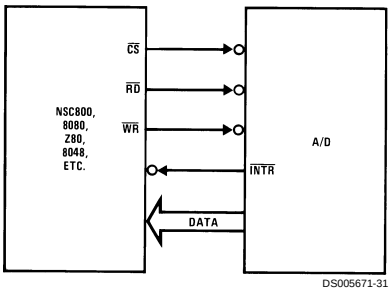
TEMP RANGE		0˚C TO 70˚C	0˚C TO 70˚C	−40˚C TO +85˚C
ERROR	±¼ Bit Adjusted			ADC0801LCN
	±½ Bit Unadjusted	ADC0802LCWM		ADC0802LCN
	±½ Bit Adjusted			ADC0803LCN
	±1Bit Unadjusted	ADC0804LCWM	ADC0804LCN	ADC0805LCN/ADC0804LCJ
PACKAGE OUTLINE		M20B — Small Outline	N20A — Molded DIP	

TRI-STATE® is a registered trademark of National Semiconductor Corp.
Z-80® is a registered trademark of Zilog Corp.

Typical Applications



8080 Interface



Error Specification (Includes Full-Scale, Zero Error, and Non-Linearity)			
Part Number	**Full-Scale Adjusted**	**$V_{REF}/2$=2.500 V_{DC} (No Adjustments)**	**$V_{REF}/2$=No Connection (No Adjustments)**
ADC0801	±¼ LSB		
ADC0802		±½ LSB	
ADC0803	±½ LSB		
ADC0804		±1 LSB	
ADC0805			±1 LSB

Absolute Maximum Ratings (Notes 1, 2)

If Military/Aerospace specified devices are required, please contact the National Semiconductor Sales Office/ Distributors for availability and specifications.

Supply Voltage (V_{CC}) (Note 3)	6.5V
Voltage	
Logic Control Inputs	−0.3V to +18V
At Other Input and Outputs	−0.3V to (V_{CC}+0.3V)
Lead Temp. (Soldering, 10 seconds)	
Dual-In-Line Package (plastic)	260˚C
Dual-In-Line Package (ceramic)	300˚C
Surface Mount Package	
Vapor Phase (60 seconds)	215˚C
Infrared (15 seconds)	220˚C
Storage Temperature Range	−65˚C to +150˚C
Package Dissipation at T_A=25˚C	875 mW
ESD Susceptibility (Note 10)	800V

Operating Ratings (Notes 1, 2)

Temperature Range	$T_{MIN} \le T_A \le T_{MAX}$
ADC0804LCJ	−40˚C≤T_A≤+85˚C
ADC0801/02/03/05LCN	−40˚C≤T_A≤+85˚C
ADC0804LCN	0˚C≤T_A≤+70˚C
ADC0802/04LCWM	0˚C≤T_A≤+70˚C
Range of V_{CC}	4.5 V_{DC} to 6.3 V_{DC}

Electrical Characteristics

The following specifications apply for V_{CC}=5 V_{DC}, $T_{MIN} \le T_A \le T_{MAX}$ and f_{CLK}=640 kHz unless otherwise specified.

Parameter	Conditions	Min	Typ	Max	Units
ADC0801: Total Adjusted Error (Note 8)	With Full-Scale Adj. (See Section 2.5.2)			±¼	LSB
ADC0802: Total Unadjusted Error (Note 8)	V_{REF}/2=2.500 V_{DC}			±½	LSB
ADC0803: Total Adjusted Error (Note 8)	With Full-Scale Adj. (See Section 2.5.2)			±½	LSB
ADC0804: Total Unadjusted Error (Note 8)	V_{REF}/2=2.500 V_{DC}			±1	LSB
ADC0805: Total Unadjusted Error (Note 8)	V_{REF}/2-No Connection			±1	LSB
V_{REF}/2 Input Resistance (Pin 9)	ADC0801/02/03/05 ADC0804 (Note 9)	2.5 0.75	8.0 1.1		kΩ kΩ
Analog Input Voltage Range	(Note 4) V(+) or V(−)	Gnd–0.05		V_{CC}+0.05	V_{DC}
DC Common-Mode Error	Over Analog Input Voltage Range		±1/16	±⅛	LSB
Power Supply Sensitivity	V_{CC}=5 V_{DC} ±10% Over Allowed V_{IN}(+) and V_{IN}(−) Voltage Range (Note 4)		±1/16	±⅛	LSB

AC Electrical Characteristics

The following specifications apply for V_{CC}=5 V_{DC} and $T_{MIN} \le T_A \le T_{MAX}$ unless otherwise specified.

Symbol	Parameter	Conditions	Min	Typ	Max	Units
T_C	Conversion Time	f_{CLK}=640 kHz (Note 6)	103		114	µs
T_C	Conversion Time	(Notes 5, 6)	66		73	1/f_{CLK}
f_{CLK}	Clock Frequency Clock Duty Cycle	V_{CC}=5V, (Note 5)	100 40	640	1460 60	kHz %
CR	Conversion Rate in Free-Running Mode	$\overline{INTR}$ tied to $\overline{WR}$ with $\overline{CS}$ =0 V_{DC}, f_{CLK}=640 kHz	8770		9708	conv/s
$t_{W(\overline{WR})L}$	Width of $\overline{WR}$ Input (Start Pulse Width)	$\overline{CS}$ =0 V_{DC} (Note 7)	100			ns
t_{ACC}	Access Time (Delay from Falling Edge of $\overline{RD}$ to Output Data Valid)	C_L=100 pF		135	200	ns
t_{1H}, t_{0H}	TRI-STATE Control (Delay from Rising Edge of $\overline{RD}$ to Hi-Z State)	C_L=10 pF, R_L=10k (See TRI-STATE Test Circuits)		125	200	ns
t_{WI}, t_{RI}	Delay from Falling Edge of $\overline{WR}$ or $\overline{RD}$ to Reset of $\overline{INTR}$			300	450	ns
C_{IN}	Input Capacitance of Logic Control Inputs			5	7.5	pF

AC Electrical Characteristics (Continued)

The following specifications apply for V_{CC}=5 V_{DC} and $T_{MIN} \leq T_A \leq T_{MAX}$ unless otherwise specified.

Symbol	Parameter	Conditions	Min	Typ	Max	Units
C_{OUT}	TRI-STATE Output Capacitance (Data Buffers)			5	7.5	pF
CONTROL INPUTS [Note: CLK IN (Pin 4) is the input of a Schmitt trigger circuit and is therefore specified separately]						
V_{IN} (1)	Logical "1" Input Voltage (Except Pin 4 CLK IN)	V_{CC}=5.25 V_{DC}	2.0		15	V_{DC}
V_{IN} (0)	Logical "0" Input Voltage (Except Pin 4 CLK IN)	V_{CC}=4.75 V_{DC}			0.8	V_{DC}
I_{IN} (1)	Logical "1" Input Current (All Inputs)	V_{IN}=5 V_{DC}		0.005	1	μA_{DC}
I_{IN} (0)	Logical "0" Input Current (All Inputs)	V_{IN}=0 V_{DC}	−1	−0.005		μA_{DC}
CLOCK IN AND CLOCK R						
V_T+	CLK IN (Pin 4) Positive Going Threshold Voltage		2.7	3.1	3.5	V_{DC}
V_T-	CLK IN (Pin 4) Negative Going Threshold Voltage		1.5	1.8	2.1	V_{DC}
V_H	CLK IN (Pin 4) Hysteresis $(V_T+)-(V_T-)$		0.6	1.3	2.0	V_{DC}
V_{OUT} (0)	Logical "0" CLK R Output Voltage	I_O=360 µA V_{CC}=4.75 V_{DC}			0.4	V_{DC}
V_{OUT} (1)	Logical "1" CLK R Output Voltage	I_O=−360 µA V_{CC}=4.75 V_{DC}	2.4			V_{DC}
DATA OUTPUTS AND $\overline{INTR}$						
V_{OUT} (0)	Logical "0" Output Voltage					
	Data Outputs	I_{OUT}=1.6 mA, V_{CC}=4.75 V_{DC}			0.4	V_{DC}
	$\overline{INTR}$ Output	I_{OUT}=1.0 mA, V_{CC}=4.75 V_{DC}			0.4	V_{DC}
V_{OUT} (1)	Logical "1" Output Voltage	I_O=−360 µA, V_{CC}=4.75 V_{DC}	2.4			V_{DC}
V_{OUT} (1)	Logical "1" Output Voltage	I_O=−10 µA, V_{CC}=4.75 V_{DC}	4.5			V_{DC}
I_{OUT}	TRI-STATE Disabled Output Leakage (All Data Buffers)	V_{OUT}=0 V_{DC}	−3			μA_{DC}
		V_{OUT}=5 V_{DC}			3	μA_{DC}
I_{SOURCE}		V_{OUT} Short to Gnd, T_A=25˚C	4.5	6		mA_{DC}
I_{SINK}		V_{OUT} Short to V_{CC}, T_A=25˚C	9.0	16		mA_{DC}
POWER SUPPLY						
I_{CC}	Supply Current (Includes Ladder Current)	f_{CLK}=640 kHz, $V_{REF}/2$=NC, T_A=25˚C and $\overline{CS}$ =5V				
	ADC0801/02/03/04LCJ/05			1.1	1.8	mA
	ADC0804LCN/LCWM			1.9	2.5	mA

Note 1: Absolute Maximum Ratings indicate limits beyond which damage to the device may occur. DC and AC electrical specifications do not apply when operating the device beyond its specified operating conditions.

Note 2: All voltages are measured with respect to Gnd, unless otherwise specified. The separate A Gnd point should always be wired to the D Gnd.

Note 3: A zener diode exists, internally, from V_{CC} to Gnd and has a typical breakdown voltage of 7 V_{DC}.

Note 4: For $V_{IN}(-) \geq V_{IN}(+)$ the digital output code will be 0000 0000. Two on-chip diodes are tied to each analog input (see block diagram) which will forward conduct for analog input voltages one diode drop below ground or one diode drop greater than the V_{CC} supply. Be careful, during testing at low V_{CC} levels (4.5V), as high level analog inputs (5V) can cause this input diode to conduct–especially at elevated temperatures, and cause errors for analog inputs near full-scale. The spec allows 50 mV forward bias of either diode. This means that as long as the analog V_{IN} does not exceed the supply voltage by more than 50 mV, the output code will be correct. To achieve an absolute 0 V_{DC} to 5 V_{DC} input voltage range will therefore require a minimum supply voltage of 4.950 V_{DC} over temperature variations, initial tolerance and loading.

Note 5: Accuracy is guaranteed at f_{CLK} = 640 kHz. At higher clock frequencies accuracy can degrade. For lower clock frequencies, the duty cycle limits can be extended so long as the minimum clock high time interval or minimum clock low time interval is no less than 275 ns.

Note 6: With an asynchronous start pulse, up to 8 clock periods may be required before the internal clock phases are proper to start the conversion process. The start request is internally latched, see *Figure 4* and section 2.0.

AC Electrical Characteristics (Continued)

Note 7: The $\overline{CS}$ input is assumed to bracket the $\overline{WR}$ strobe input and therefore timing is dependent on the $\overline{WR}$ pulse width. An arbitrarily wide pulse width will hold the converter in a reset mode and the start of conversion is initiated by the low to high transition of the $\overline{WR}$ pulse (see timing diagrams).

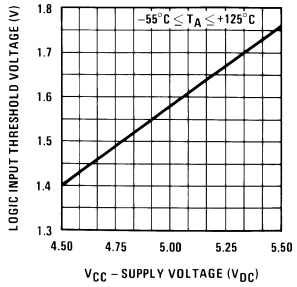
Note 8: None of these A/Ds requires a zero adjust (see section 2.5.1). To obtain zero code at other analog input voltages see section 2.5 and *Figure 7*.

Note 9: The $V_{REF}/2$ pin is the center point of a two-resistor divider connected from V_{CC} to ground. In all versions of the ADC0801, ADC0802, ADC0803, and ADC0805, and in the ADC0804LCJ, each resistor is typically 16 kΩ. In all versions of the ADC0804 except the ADC0804LCJ, each resistor is typically 2.2 kΩ.

Note 10: Human body model, 100 pF discharged through a 1.5 kΩ resistor.

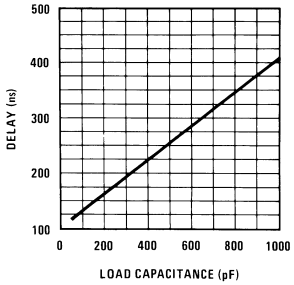
Typical Performance Characteristics

Logic Input Threshold Voltage vs. Supply Voltage



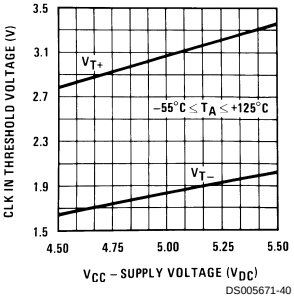
DS005671-38

Delay From Falling Edge of RD to Output Data Valid vs. Load Capacitance



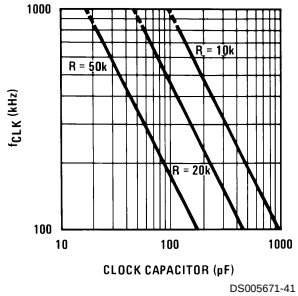
DS005671-39

CLK IN Schmitt Trip Levels vs. Supply Voltage



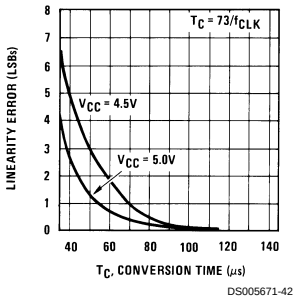
DS005671-40

f_{CLK} vs. Clock Capacitor



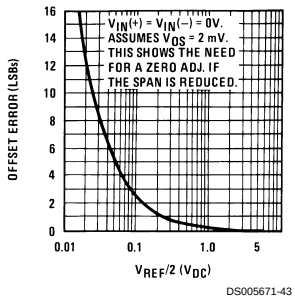
DS005671-41

Full-Scale Error vs Conversion Time



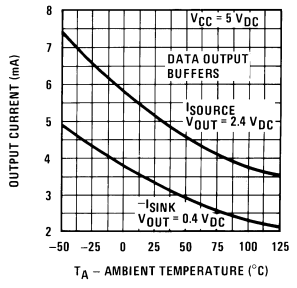
DS005671-42

Effect of Unadjusted Offset Error vs. $V_{REF}/2$ Voltage



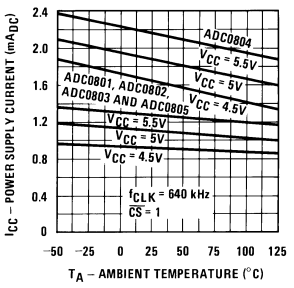
DS005671-43

Output Current vs Temperature



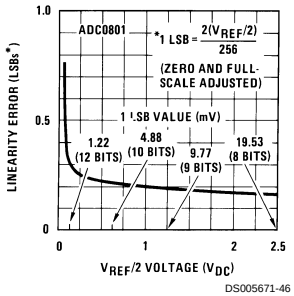
DS005671-44

Power Supply Current vs Temperature (Note 9)



DS005671-45

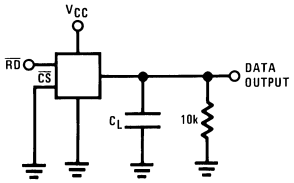
Linearity Error at Low $V_{REF}/2$ Voltages



DS005671-46

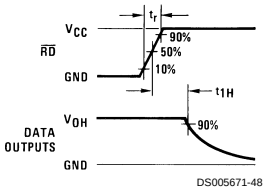
TRI-STATE Test Circuits and Waveforms

t_{1H}



DS005671-47

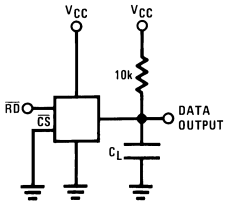
t_{1H}, C_L=10 pF



DS005671-48

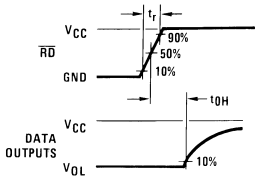
t_r=20 ns

t_{0H}



DS005671-49

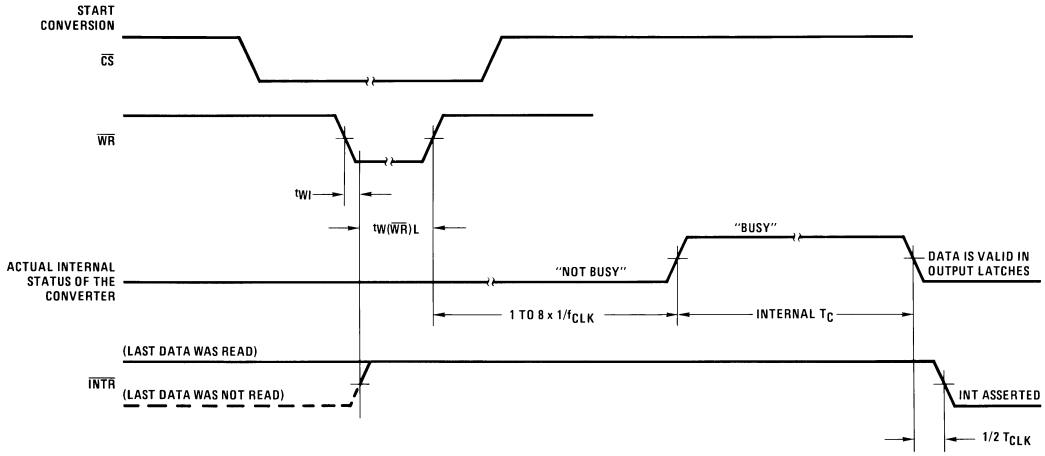
t_{0H}, C_L=10 pF



DS005671-50

t_r=20 ns

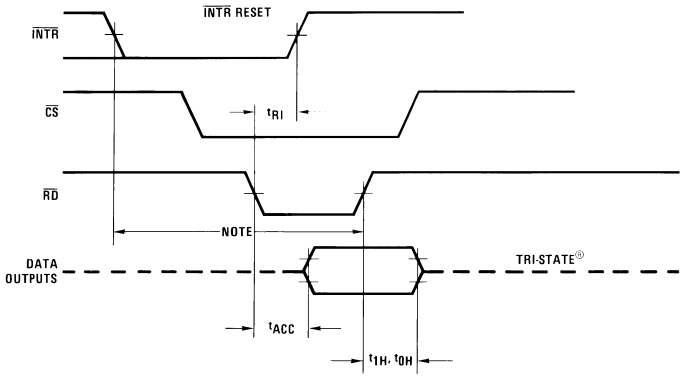
Timing Diagrams (All timing is measured from the 50% voltage points)



DS005671-51

Timing Diagrams (All timing is measured from the 50% voltage points) (Continued)

Output Enable and Reset with $\overline{\text{INTR}}$

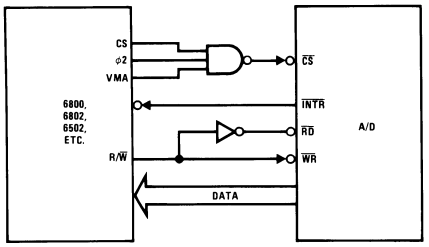


DS005671-52

Note: Read strobe must occur 8 clock periods ($8/f_{CLK}$) after assertion of interrupt to guarantee reset of $\overline{\text{INTR}}$.

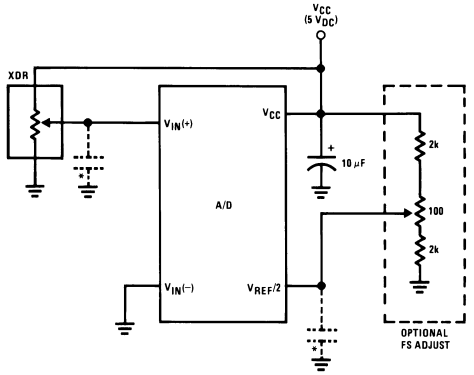
Typical Applications

6800 Interface



DS005671-53

Ratiometeric with Full-Scale Adjust

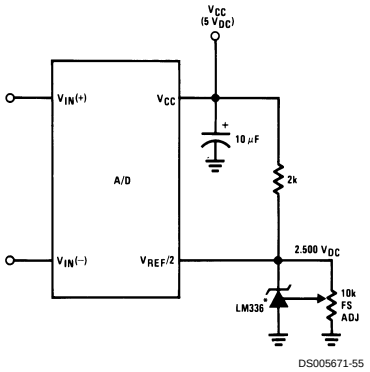


DS005671-54

Note: before using caps at V_{IN} or $V_{REF}/2$, see section 2.3.2 Input Bypass Capacitors.

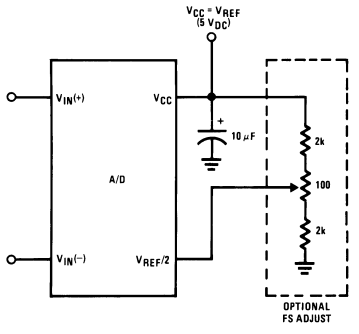
Typical Applications (Continued)

Absolute with a 2.500V Reference



DS005671-55

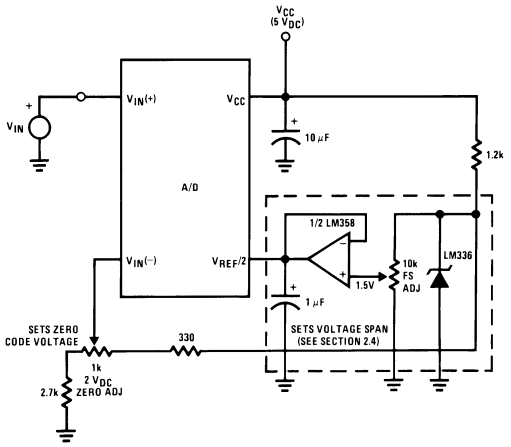
Absolute with a 5V Reference



DS005671-56

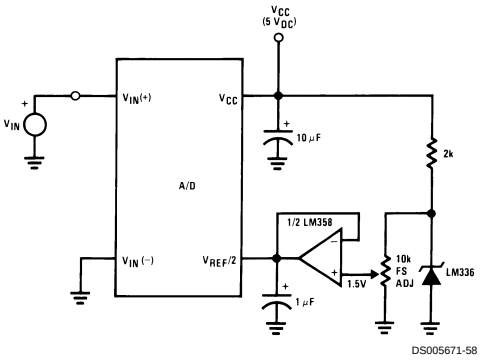
*For low power, see also LM385–2.5

Zero-Shift and Span Adjust: 2V ≤ V_{IN} ≤ 5V



DS005671-57

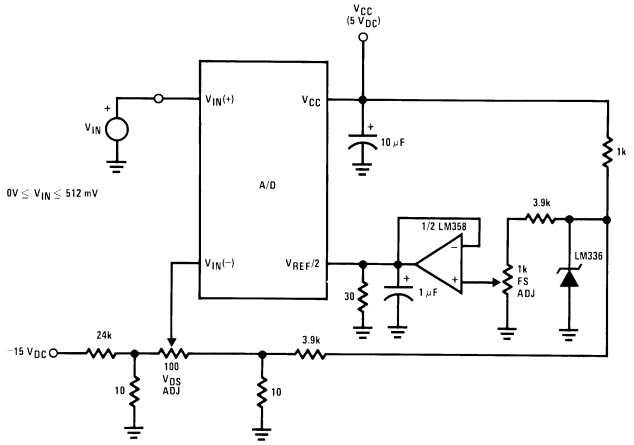
Span Adjust: 0V ≤ V_{IN} ≤ 3V



DS005671-58

Typical Applications (Continued)

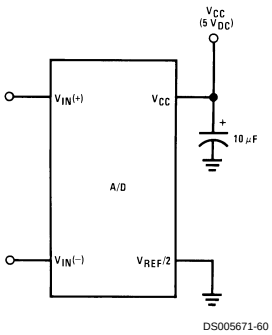
Directly Converting a Low-Level Signal



DS005671-59

$V_{REF}/2$=256 mV

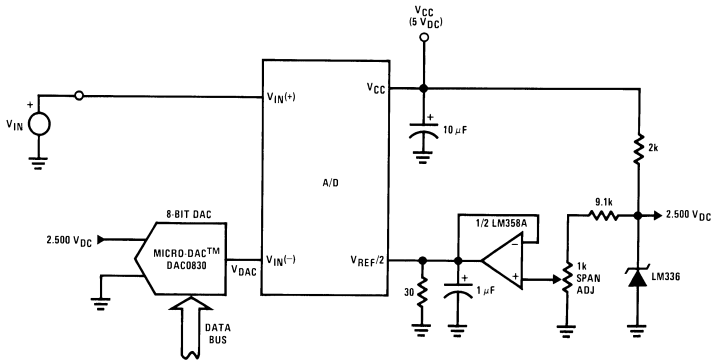
A µP Interfaced Comparator



DS005671-60

For:
$V_{IN}(+) > V_{IN}(-)$
Output=FF_{HEX}
For:
$V_{IN}(+) < V_{IN}(-)$
Output=00_{HEX}

1 mV Resolution with µP Controlled Range

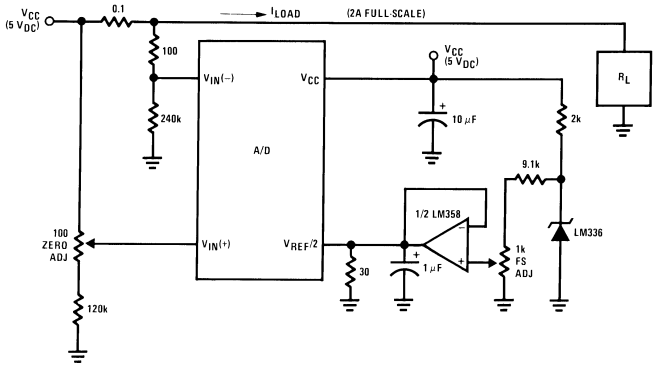


DS005671-61

$V_{REF}/2$=128 mV
1 LSB=1 mV
$V_{DAC} \leq V_{IN} \leq (V_{DAC}+256\ mV)$
$0 \leq V_{DAC} < 2.5V$

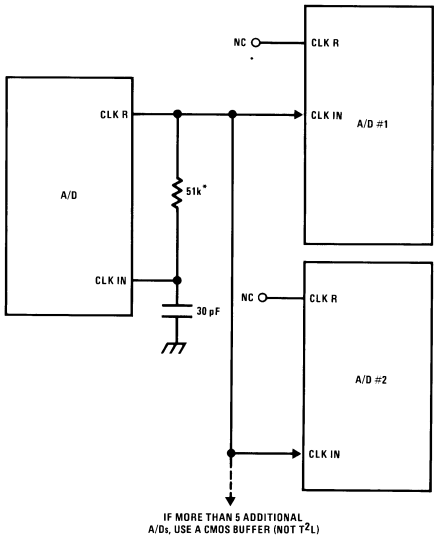
Typical Applications (Continued)

Digitizing a Current Flow



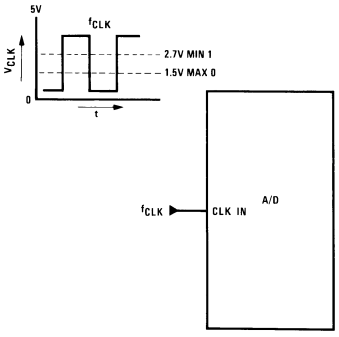
DS005671-62

Self-Clocking Multiple A/Ds



DS005671-63

External Clocking



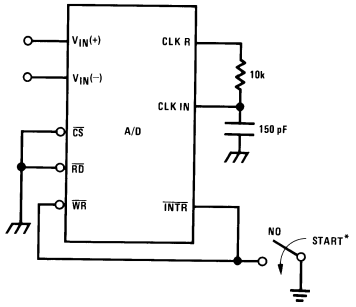
DS005671-64

100 kHz≤f_{CLK}≤1460 kHz

* Use a large R value to reduce loading at CLK R output.

Typical Applications (Continued)

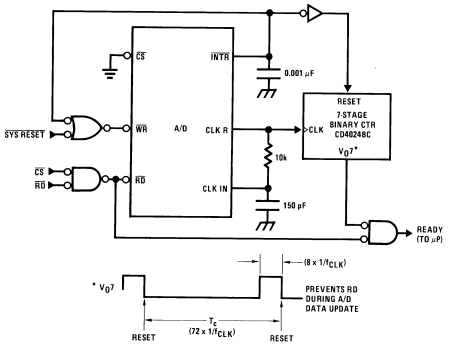
Self-Clocking in Free-Running Mode



DS005671-65

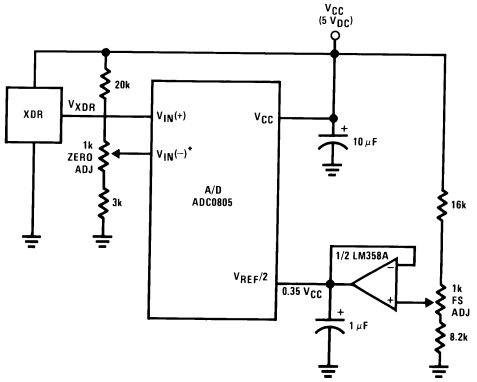
*After power-up, a momentary grounding of the $\overline{WR}$ input is needed to guarantee operation.

µP Interface for Free-Running A/D



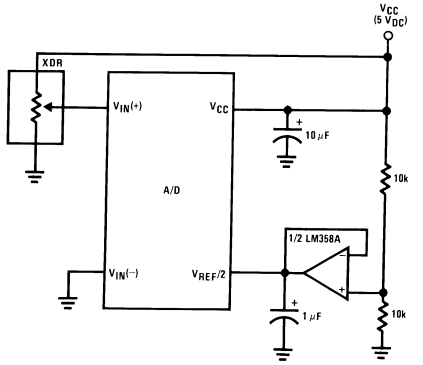
DS005671-66

Operating with "Automotive" Ratiometric Transducers



DS005671-67

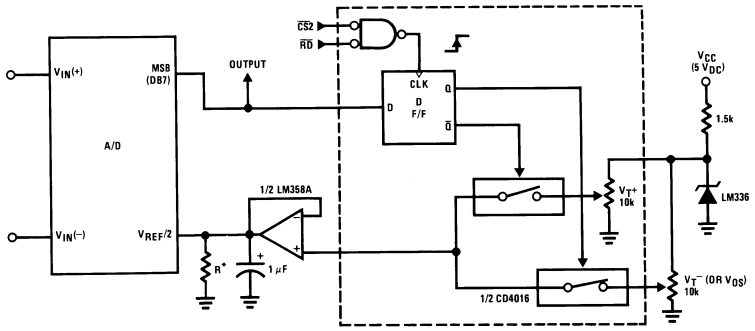
Ratiometric with $V_{REF}/2$ Forced



DS005671-68

*$V_{IN}(-)$=0.15 V_{CC}
15% of $V_{CC} \leq V_{XDR} \leq 85\%$ of V_{CC}

µP Compatible Differential-Input Comparator with Pre-Set V_{OS} (with or without Hysteresis)

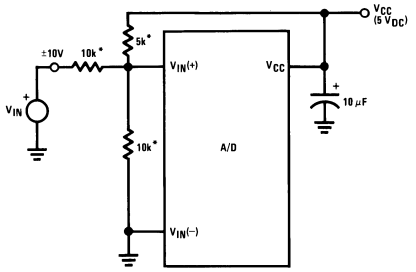


DS005671-69

*See *Figure 5* to select R value
DB7="1" for $V_{IN}(+) > V_{IN}(-) + (V_{REF}/2)$
Omit circuitry within the dotted area if hysteresis is not needed

Typical Applications (Continued)

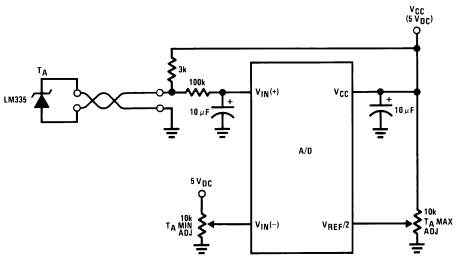
Handling ±10V Analog Inputs



DS005671-70

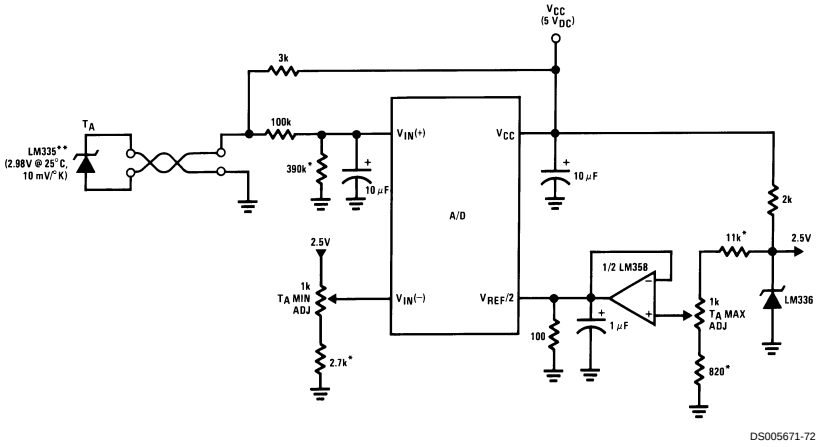
*Beckman Instruments #694-3-R10K resistor array

Low-Cost, µP Interfaced, Temperature-to-Digital Converter



DS005671-71

µP Interfaced Temperature-to-Digital Converter



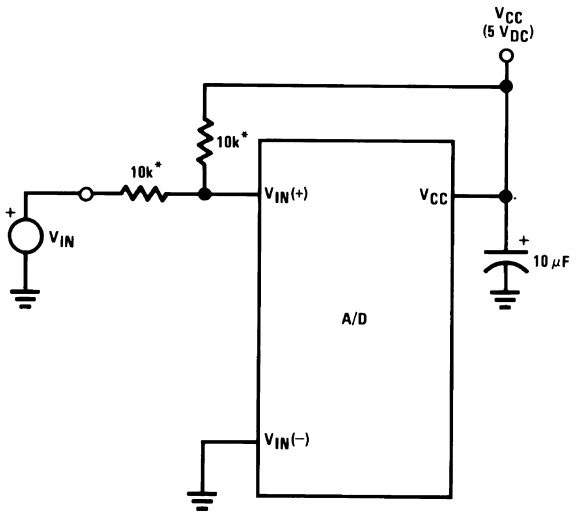
DS005671-72

*Circuit values shown are for $0^{\circ}C \leq T_A \leq +128^{\circ}C$

***Can calibrate each sensor to allow easy replacement, then A/D can be calibrated with a pre-set input voltage.

Typical Applications (Continued)

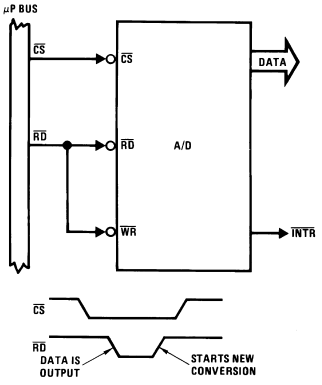
Handling ±5V Analog Inputs



DS005671-33

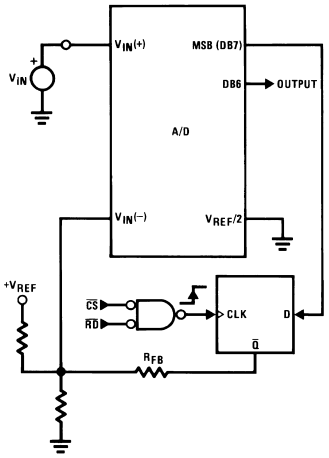
*Beckman Instruments #694-3-R10K resistor array

Read-Only Interface



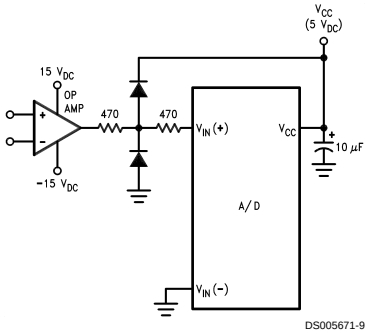
DS005671-34

µP Interfaced Comparator with Hysteresis



DS005671-35

Protecting the Input

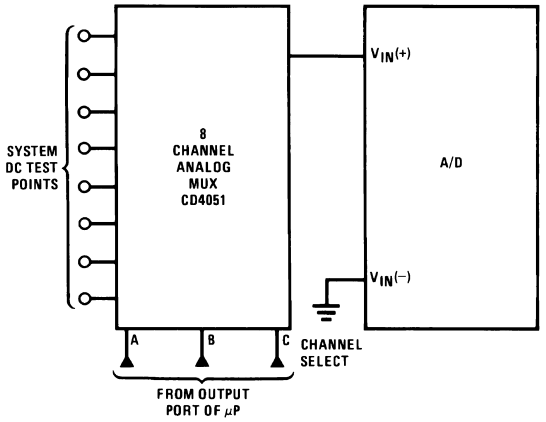


DS005671-9

Diodes are 1N914

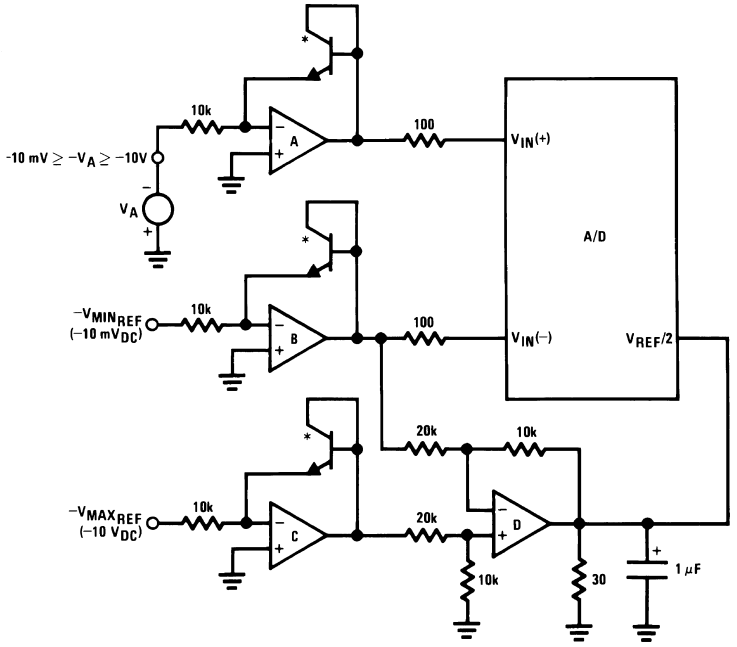
Typical Applications (Continued)

Analog Self-Test for a System



DS005671-36

A Low-Cost, 3-Decade Logarithmic Converter

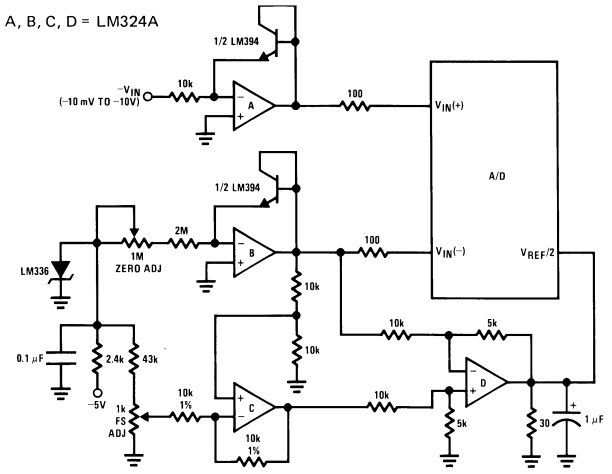


DS005671-37

*LM389 transistors
A, B, C, D = LM324A quad op amp

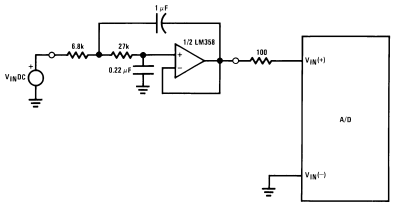
Typical Applications (Continued)

3-Decade Logarithmic A/D Converter



DS005671-73

Noise Filtering the Analog Input



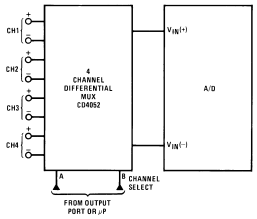
DS005671-74

f_C=20 Hz

Uses Chebyshev implementation for steeper roll-off unity-gain, 2nd order, low-pass filter

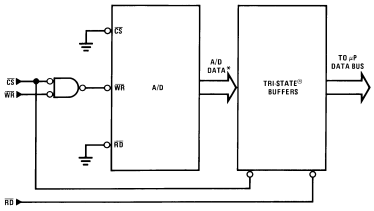
Adding a separate filter for each channel increases system response time if an analog multiplexer is used

Multiplexing Differential Inputs



DS005671-75

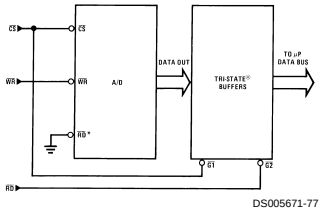
Output Buffers with A/D Data Enabled



DS005671-76

*A/D output data is updated 1 CLK period prior to assertion of $\overline{INTR}$

Increasing Bus Drive and/or Reducing Time on Bus

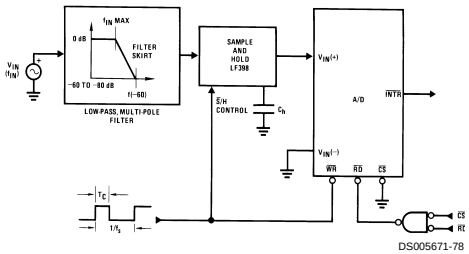


DS005671-77

*Allows output data to set-up at falling edge of $\overline{CS}$

Typical Applications (Continued)

Sampling an AC Input Signal

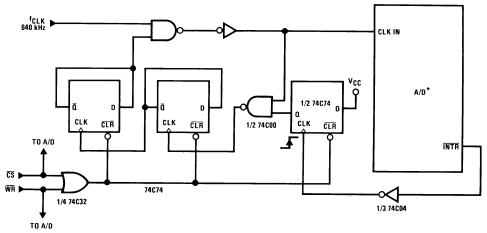


DS005671-78

Note 11: Oversample whenever possible [keep fs > 2f(−60)] to eliminate input frequency folding (aliasing) and to allow for the skirt response of the filter.

Note 12: Consider the amplitude errors which are introduced within the passband of the filter.

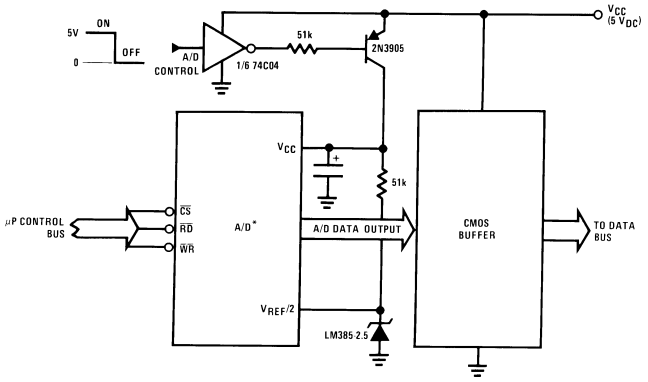
70% Power Savings by Clock Gating



DS005671-79

(Complete shutdown takes ≈ 30 seconds.)

Power Savings by A/D and V_{REF} Shutdown



DS005671-80

*Use ADC0801, 02, 03 or 05 for lowest power consumption.

Note: Logic inputs can be driven to V_{CC} with A/D supply at zero volts.

Buffer prevents data bus from overdriving output of A/D when in shutdown mode.

Functional Description

1.0 UNDERSTANDING A/D ERROR SPECS

A perfect A/D transfer characteristic (staircase waveform) is shown in *Figure 1*. The horizontal scale is analog input voltage and the particular points labeled are in steps of 1 LSB (19.53 mV with 2.5V tied to the $V_{REF}/2$ pin). The digital output codes that correspond to these inputs are shown as D−1, D, and D+1. For the perfect A/D, not only will center-value (A−1, A, A+1, ...) analog inputs produce the correct output digital codes, but also each riser (the transitions between adjacent output codes) will be located ±½ LSB away from each center-value. As shown, the risers are ideal and have no width. Correct digital output codes will be provided for a range of analog input voltages that extend ±½ LSB from the ideal center-values. Each tread (the range of analog input voltage that provides the same digital output code) is therefore 1 LSB wide.

Functional Description (Continued)

Figure 2 shows a worst case error plot for the ADC0801. All center-valued inputs are guaranteed to produce the correct output codes and the adjacent risers are guaranteed to be no closer to the center-value points than ±¼ LSB. In other words, if we apply an analog input equal to the center-value ±¼ LSB, *we guarantee* that the A/D will produce the correct digital code. The maximum range of the position of the code transition is indicated by the horizontal arrow and it is guaranteed to be no more than ½ LSB.

The error curve of *Figure 3* shows a worst case error plot for the ADC0802. Here we guarantee that if we apply an analog input equal to the LSB analog voltage center-value the A/D will produce the correct digital code.

Next to each transfer function is shown the corresponding error plot. Many people may be more familiar with error plots than transfer functions. The analog input voltage to the A/D is provided by either a linear ramp or by the discrete output steps of a high resolution DAC. Notice that the error is continuously displayed and includes the quantization uncertainty of the A/D. For example the error at point 1 of *Figure 1* is +½ LSB because the digital code appeared ½ LSB in advance of the center-value of the tread. The error plots always have a constant negative slope and the abrupt upside steps are always 1 LSB in magnitude.

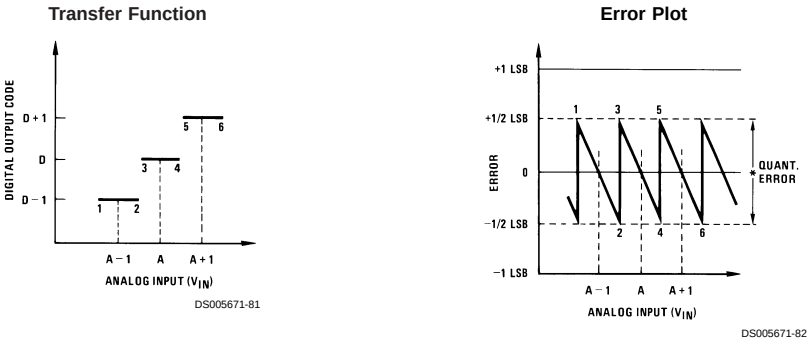


FIGURE 1. Clarifying the Error Specs of an A/D Converter Accuracy=±0 LSB: A Perfect A/D

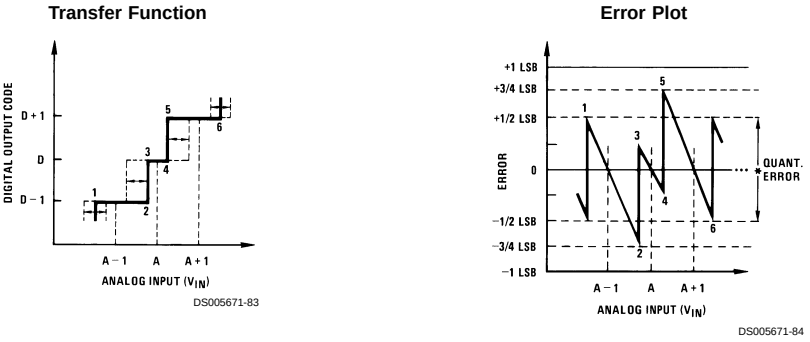


FIGURE 2. Clarifying the Error Specs of an A/D Converter Accuracy=±¼ LSB

Functional Description (Continued)

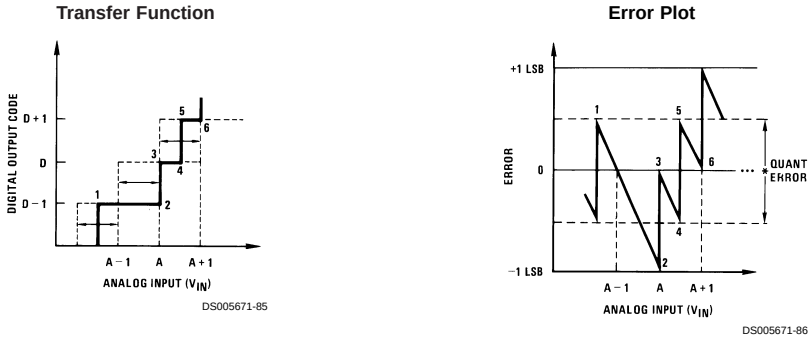


FIGURE 3. Clarifying the Error Specs of an A/D Converter
Accuracy=±½ LSB

2.0 FUNCTIONAL DESCRIPTION

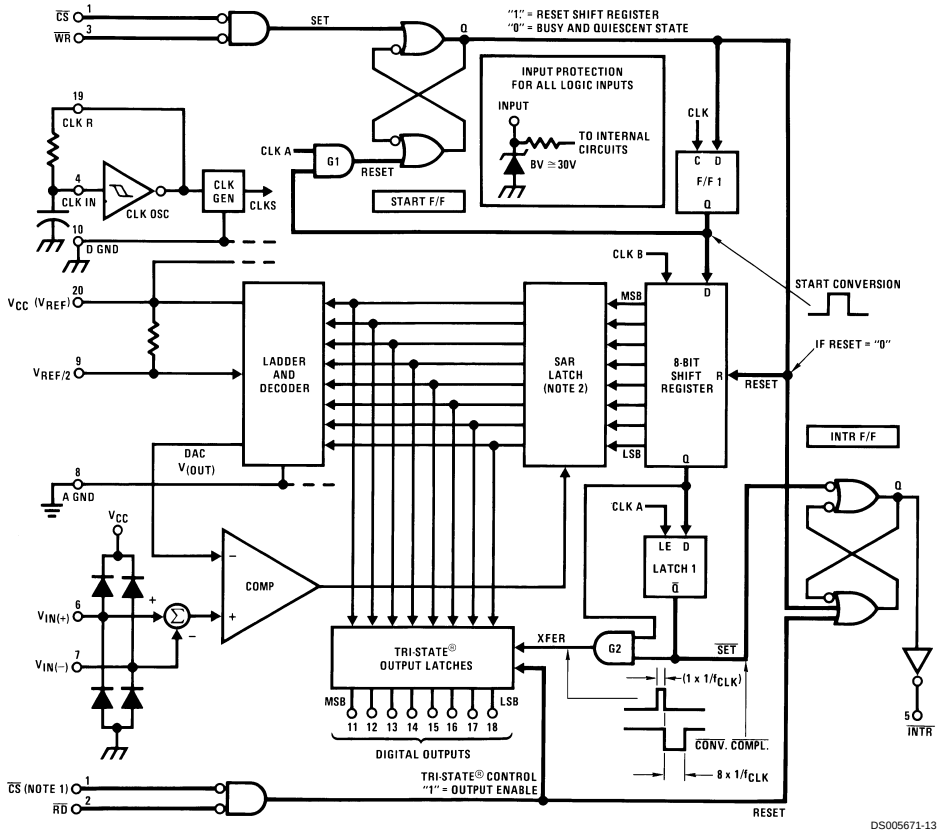
The ADC0801 series contains a circuit equivalent of the 256R network. Analog switches are sequenced by successive approximation logic to match the analog difference input voltage [$V_{IN}(+) - V_{IN}(-)$] to a corresponding tap on the R network. The most significant bit is tested first and after 8 comparisons (64 clock cycles) a digital 8-bit binary code (1111 1111 = full-scale) is transferred to an output latch and then an interrupt is asserted ($\overline{INTR}$ makes a high-to-low transition). A conversion in process can be interrupted by issuing a second start command. The device may be operated in the free-running mode by connecting $\overline{INTR}$ to the $\overline{WR}$ input with $\overline{CS}$ =0. To ensure start-up under all possible conditions, an external $\overline{WR}$ pulse is required during the first power-up cycle.

On the high-to-low transition of the $\overline{WR}$ input the internal SAR latches and the shift register stages are reset. As long as the $\overline{CS}$ input and $\overline{WR}$ input remain low, the A/D will remain in a reset state. *Conversion will start from 1 to 8 clock periods after at least one of these inputs makes a low-to-high transition.*

A functional diagram of the A/D converter is shown in *Figure 4.* All of the package pinouts are shown and the major logic control paths are drawn in heavier weight lines.

The converter is started by having $\overline{CS}$ and $\overline{WR}$ simultaneously low. This sets the start flip-flop (F/F) and the resulting "1" level resets the 8-bit shift register, resets the Interrupt (INTR) F/F and inputs a "1" to the D flop, F/F1, which is at the input end of the 8-bit shift register. Internal clock signals then transfer this "1" to the Q output of F/F1. The AND gate, G1, combines this "1" output with a clock signal to provide a reset signal to the start F/F. If the set signal is no longer present (either $\overline{WR}$ or $\overline{CS}$ is a "1") the start F/F is reset and the 8-bit shift register then can have the "1" clocked in, which starts the conversion process. If the set signal were to still be present, this reset pulse would have no effect (both outputs of the start F/F would momentarily be at a "1" level) and the 8-bit shift register would continue to be held in the reset mode. This logic therefore allows for wide $\overline{CS}$ and $\overline{WR}$ signals and the converter will start after at least one of these signals returns high and the internal clocks again provide a reset signal for the start F/F.

Functional Description (Continued)



Note 13: $\overline{CS}$ shown twice for clarity.

Note 14: SAR = Successive Approximation Register.

FIGURE 4. Block Diagram

After the "1" is clocked through the 8-bit shift register (which completes the SAR search) it appears as the input to the D-type latch, LATCH 1. As soon as this "1" is output from the shift register, the AND gate, G2, causes the new digital word to transfer to the TRI-STATE output latches. When LATCH 1 is subsequently enabled, the Q output makes a high-to-low transition which causes the INTR F/F to set. An inverting buffer then supplies the $\overline{INTR}$ input signal.

Note that this $\overline{SET}$ control of the INTR F/F remains low for 8 of the external clock periods (as the internal clocks run at ⅛ of the frequency of the external clock). If the data output is continuously enabled ($\overline{CS}$ and $\overline{RD}$ both held low), the $\overline{INTR}$ output will still signal the end of conversion (by a high-to-low transition), because the $\overline{SET}$ input can control the Q output of the INTR F/F even though the RESET input is constantly at a "1" level in this operating mode. This $\overline{INTR}$ output will therefore stay low for the duration of the $\overline{SET}$ signal, which is 8 periods of the external clock frequency (assuming the A/D is not started during this interval).

When operating in the free-running or continuous conversion mode ($\overline{INTR}$ pin tied to $\overline{WR}$ and $\overline{CS}$ wired low — see also section 2.8), the START F/F is SET by the high-to-low transition of the $\overline{INTR}$ signal. This resets the SHIFT REGISTER which causes the input to the D-type latch, LATCH 1, to go low. As the latch enable input is still present, the $\overline{Q}$ output will go high, which then allows the INTR F/F to be RESET. This reduces the width of the resulting $\overline{INTR}$ output pulse to only a few propagation delays (approximately 300 ns).

When data is to be read, the combination of both $\overline{CS}$ and $\overline{RD}$ being low will cause the INTR F/F to be reset and the TRI-STATE output latches will be enabled to provide the 8-bit digital outputs.

2.1 Digital Control Inputs

The digital control inputs ($\overline{CS}$, $\overline{RD}$, and $\overline{WR}$) meet standard T^2L logic voltage levels. These signals have been renamed when compared to the standard A/D Start and Output Enable labels. In addition, these inputs are active low to allow an easy interface to microprocessor control busses. For non-microprocessor based applications, the $\overline{CS}$ input (pin 1) can be grounded and the standard A/D Start function is obtained by an active low pulse applied at the $\overline{WR}$ input (pin 3) and the Output Enable function is caused by an active low pulse at the $\overline{RD}$ input (pin 2).

Functional Description (Continued)

2.2 Analog Differential Voltage Inputs and Common-Mode Rejection

This A/D has additional applications flexibility due to the analog differential voltage input. The $V_{IN}(-)$ input (pin 7) can be used to automatically subtract a fixed voltage value from the input reading (tare correction). This is also useful in 4 mA–20 mA current loop conversion. In addition, common-mode noise can be reduced by use of the differential input.

The time interval between sampling $V_{IN}(+)$ and $V_{IN}(-)$ is 4-½ clock periods. The maximum error voltage due to this slight time difference between the input voltage samples is given by:

$$\Delta V_e(MAX) = (V_P)(2\pi f_{cm})\left(\frac{4.5}{f_{CLK}}\right)$$

where:

ΔV_e is the error voltage due to sampling delay

V_P is the peak value of the common-mode voltage

f_{cm} is the common-mode frequency

As an example, to keep this error to ¼ LSB (~5 mV) when operating with a 60 Hz common-mode frequency, f_{cm}, and using a 640 kHz A/D clock, f_{CLK}, would allow a peak value of the common-mode voltage, V_P, which is given by:

$$V_P = \frac{[\Delta V_{e(MAX)}(f_{CLK})]}{(2\pi f_{cm})(4.5)}$$

or

$$V_P = \frac{(5 \times 10^{-3})(640 \times 10^3)}{(6.28)(60)(4.5)}$$

which gives

$V_P \cong 1.9V$.

The allowed range of analog input voltages usually places more severe restrictions on input common-mode noise levels.

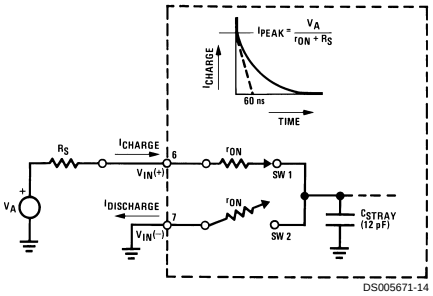
An analog input voltage with a reduced span and a relatively large zero offset can be handled easily by making use of the differential input (see section 2.4 Reference Voltage).

2.3 Analog Inputs

2.3 1 Input Current

Normal Mode

Due to the internal switching action, displacement currents will flow at the analog inputs. This is due to on-chip stray capacitance to ground as shown in *Figure 5*.



r_{ON} of SW 1 and SW 2 ≅ 5 kΩ

$r = r_{ON} C_{STRAY} \cong 5\ k\Omega \times 12\ pF = 60\ ns$

FIGURE 5. Analog Input Impedance

The voltage on this capacitance is switched and will result in currents entering the $V_{IN}(+)$ input pin and leaving the $V_{IN}(-)$ input which will depend on the analog differential input voltage levels. These current transients occur at the leading edge of the internal clocks. They rapidly decay and *do not cause errors* as the on-chip comparator is strobed at the end of the clock period.

Fault Mode

If the voltage source applied to the $V_{IN}(+)$ or $V_{IN}(-)$ pin exceeds the allowed operating range of V_{CC}+50 mV, large input currents can flow through a parasitic diode to the V_{CC} pin. If these currents can exceed the 1 mA max allowed spec, an external diode (1N914) should be added to bypass this current to the V_{CC} pin (with the current bypassed with this diode, the voltage at the $V_{IN}(+)$ pin can exceed the V_{CC} voltage by the forward voltage of this diode).

2.3.2 Input Bypass Capacitors

Bypass capacitors at the inputs will average these charges and cause a DC current to flow through the output resistances of the analog signal sources. This charge pumping action is worse for continuous conversions with the $V_{IN}(+)$ input voltage at full-scale. For continuous conversions with a 640 kHz clock frequency with the $V_{IN}(+)$ input at 5V, this DC current is at a maximum of approximately 5 µA. Therefore, *bypass capacitors should not be used at the analog inputs or the $V_{REF}/2$ pin* for high resistance sources (> 1 kΩ). If input bypass capacitors are necessary for noise filtering and high source resistance is desirable to minimize capacitor size, the detrimental effects of the voltage drop across this input resistance, which is due to the average value of the input current, can be eliminated with a full-scale adjustment while the given source resistor and input bypass capacitor are both in place. This is possible because the average value of the input current is a precise linear function of the differential input voltage.

2.3.3 Input Source Resistance

Large values of source resistance where an input bypass capacitor is not used, *will not cause errors* as the input currents settle out prior to the comparison time. If a low pass filter is required in the system, use a low valued series resistor (≤ 1 kΩ) for a passive RC section or add an op amp RC active low pass filter. For low source resistance applications, (≤ 1 kΩ), a 0.1 µF bypass capacitor at the inputs will prevent noise pickup due to series lead inductance of a long wire. A

Functional Description (Continued)

100Ω series resistor can be used to isolate this capacitor — both the R and C are placed outside the feedback loop — from the output of an op amp, if used.

2.3.4 Noise

The leads to the analog inputs (pins 6 and 7) should be kept as short as possible to minimize input noise coupling. Both noise and undesired digital clock coupling to these inputs can cause system errors. The source resistance for these inputs should, in general, be kept below 5 kΩ. Larger values of source resistance can cause undesired system noise pickup. Input bypass capacitors, placed from the analog inputs to ground, will eliminate system noise pickup but can create analog scale errors as these capacitors will average the transient input switching currents of the A/D (see section 2.3.1.). This scale error depends on both a large source resistance and the use of an input bypass capacitor. This error can be eliminated by doing a full-scale adjustment of the A/D (adjust $V_{REF}/2$ for a proper full-scale reading — see section 2.5.2 on Full-Scale Adjustment) with the source resistance and input bypass capacitor in place.

2.4 Reference Voltage

2.4.1 Span Adjust

For maximum applications flexibility, these A/Ds have been designed to accommodate a 5 V_{DC}, 2.5 V_{DC} or an adjusted voltage reference. This has been achieved in the design of the IC as shown in *Figure 6*.

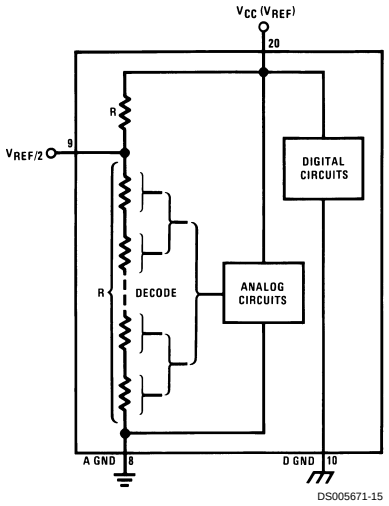


FIGURE 6. The $V_{REFERENCE}$ Design on the IC

Notice that the reference voltage for the IC is either ½ of the voltage applied to the V_{CC} supply pin, or is equal to the voltage that is externally forced at the $V_{REF}/2$ pin. This allows for a ratiometric voltage reference using the V_{CC} supply, a 5 V_{DC} reference voltage can be used for the V_{CC} supply or a voltage less than 2.5 V_{DC} can be applied to the $V_{REF}/2$ input for increased application flexibility. The internal gain to the $V_{REF}/2$ input is 2, making the full-scale differential input voltage twice the voltage at pin 9.

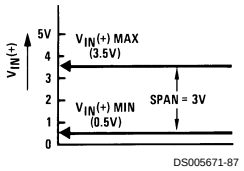
An example of the use of an adjusted reference voltage is to accommodate a reduced span — or dynamic voltage range of the analog input voltage. If the analog input voltage were to range from 0.5 V_{DC} to 3.5 V_{DC}, instead of 0V to 5 V_{DC}, the span would be 3V as shown in *Figure 7*. With 0.5 V_{DC} applied to the $V_{IN}(-)$ pin to absorb the offset, the reference voltage can be made equal to ½ of the 3V span or 1.5 V_{DC}. The A/D now will encode the $V_{IN}(+)$ signal from 0.5V to 3.5 V with the 0.5V input corresponding to zero and the 3.5 V_{DC} input corresponding to full-scale. The full 8 bits of resolution are therefore applied over this reduced analog input voltage range.

2.4.2 Reference Accuracy Requirements

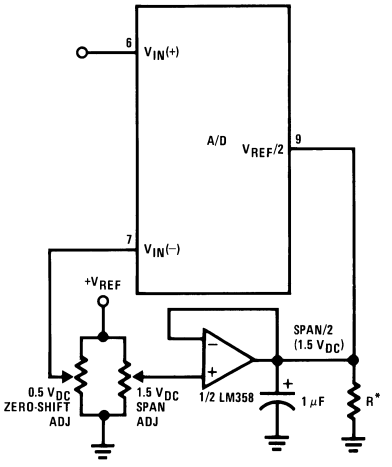
The converter can be operated in a ratiometric mode or an absolute mode. In ratiometric converter applications, the magnitude of the reference voltage is a factor in both the output of the source transducer and the output of the A/D converter and therefore cancels out in the final digital output code. The ADC0805 is specified particularly for use in ratiometric applications with no adjustments required. In absolute conversion applications, both the initial value and the temperature stability of the reference voltage are important factors in the accuracy of the A/D converter. For $V_{REF}/2$ voltages of 2.4 V_{DC} nominal value, initial errors of ±10 mV_{DC} will cause conversion errors of ±1 LSB due to the gain of 2 of the $V_{REF}/2$ input. In reduced span applications, the initial value and the stability of the $V_{REF}/2$ input voltage become even more important. For example, if the span is reduced to 2.5V, the analog input LSB voltage value is correspondingly reduced from 20 mV (5V span) to 10 mV and 1 LSB at the $V_{REF}/2$ input becomes 5 mV. As can be seen, this reduces the allowed initial tolerance of the reference voltage and requires correspondingly less absolute change with temperature variations. Note that spans smaller than 2.5V place even tighter requirements on the initial accuracy and stability of the reference source.

In general, the magnitude of the reference voltage will require an initial adjustment. Errors due to an improper value of reference voltage appear as full-scale errors in the A/D transfer function. IC voltage regulators may be used for references if the ambient temperature changes are not excessive. The LM336B 2.5V IC reference diode (from National Semiconductor) has a temperature stability of 1.8 mV typ (6 mV max) over $0^{\circ}C \leq T_A \leq +70^{\circ}C$. Other temperature range parts are also available.

Functional Description (Continued)



a) Analog Input Signal Example



*Add if $V_{REF}/2 \leq 1\ V_{DC}$ with LM358 to draw 3 mA to ground.

b) Accommodating an Analog Input from 0.5V (Digital Out = 00_{HEX}) to 3.5V (Digital Out=FF_{HEX})

FIGURE 7. Adapting the A/D Analog Input Voltages to Match an Arbitrary Input Signal Range

2.5 Errors and Reference Voltage Adjustments

2.5.1 Zero Error

The zero of the A/D does not require adjustment. If the minimum analog input voltage value, $V_{IN(MIN)}$, is not ground, a zero offset can be done. The converter can be made to output 0000 0000 digital code for this minimum input voltage by biasing the A/D $V_{IN}(-)$ input at this $V_{IN(MIN)}$ value (see Applications section). This utilizes the differential mode operation of the A/D.

The zero error of the A/D converter relates to the location of the first riser of the transfer function and can be measured by grounding the V_{IN} (−) input and applying a small magnitude positive voltage to the V_{IN} (+) input. Zero error is the difference between the actual DC input voltage that is necessary to just cause an output digital code transition from 0000 0000 to 0000 0001 and the ideal ½ LSB value (½ LSB = 9.8 mV for $V_{REF}/2$=2.500 V_{DC}).

2.5.2 Full-Scale

The full-scale adjustment can be made by applying a differential input voltage that is 1½ LSB less than the desired analog full-scale voltage range and then adjusting the magnitude of the $V_{REF}/2$ input (pin 9 or the V_{CC} supply if pin 9 is not used) for a digital output code that is just changing from 1111 1110 to 1111 1111.

2.5.3 Adjusting for an Arbitrary Analog Input Voltage Range

If the analog zero voltage of the A/D is shifted away from ground (for example, to accommodate an analog input signal that does not go to ground) this new zero reference should be properly adjusted first. A $V_{IN}(+)$ voltage that equals this desired zero reference plus ½ LSB (where the LSB is calculated for the desired analog span, 1 LSB=analog span/256) is applied to pin 6 and the zero reference voltage at pin 7 should then be adjusted to just obtain the 00_{HEX} to 01_{HEX} code transition.

The full-scale adjustment should then be made (with the proper $V_{IN}(-)$ voltage applied) by forcing a voltage to the $V_{IN}(+)$ input which is given by:

$$V_{IN}(+)\ \text{fs adj} = V_{MAX} - 1.5\left[\frac{(V_{MAX} - V_{MIN})}{256}\right]$$

where:

V_{MAX}=The high end of the analog input range

and

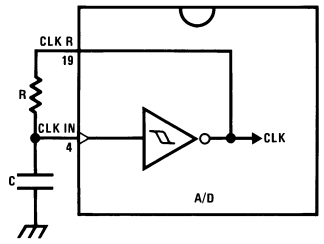
V_{MIN}=the low end (the offset zero) of the analog range. (Both are ground referenced.)

The $V_{REF}/2$ (or V_{CC}) voltage is then adjusted to provide a code change from FE_{HEX} to FF_{HEX}. This completes the adjustment procedure.

2.6 Clocking Option

The clock for the A/D can be derived from the CPU clock or an external RC can be added to provide self-clocking. The CLK IN (pin 4) makes use of a Schmitt trigger as shown in *Figure 8*.

Functional Description (Continued)



DS005671-17

$$f_{CLK} \cong \frac{1}{1.1\ RC}$$

$R \cong 10\ k\Omega$

FIGURE 8. Self-Clocking the A/D

Heavy capacitive or DC loading of the clock R pin should be avoided as this will disturb normal converter operation. Loads less than 50 pF, such as driving up to 7 A/D converter clock inputs from a single clock R pin of 1 converter, are allowed. For larger clock line loading, a CMOS or low power TTL buffer or PNP input logic should be used to minimize the loading on the clock R pin (do not use a standard TTL buffer).

2.7 Restart During a Conversion

If the A/D is restarted ($\overline{CS}$ and $\overline{WR}$ go low and return high) during a conversion, the converter is reset and a new conversion is started. The output data latch is not updated if the conversion in process is not allowed to be completed, therefore the data of the previous conversion remains in this latch. The $\overline{INTR}$ output simply remains at the "1" level.

2.8 Continuous Conversions

For operation in the free-running mode an initializing pulse should be used, following power-up, to ensure circuit operation. In this application, the $\overline{CS}$ input is grounded and the $\overline{WR}$ input is tied to the $\overline{INTR}$ output. This $\overline{WR}$ and $\overline{INTR}$ node should be momentarily forced to logic low following a power-up cycle to guarantee operation.

2.9 Driving the Data Bus

This MOS A/D, like MOS microprocessors and memories, will require a bus driver when the total capacitance of the data bus gets large. Other circuitry, which is tied to the data bus, will add to the total capacitive loading, even in TRI-STATE (high impedance mode). Backplane bussing also greatly adds to the stray capacitance of the data bus.

There are some alternatives available to the designer to handle this problem. Basically, the capacitive loading of the data bus slows down the response time, even though DC specifications are still met. For systems operating with a relatively slow CPU clock frequency, more time is available in which to establish proper logic levels on the bus and therefore higher capacitive loads can be driven (see typical characteristics curves).

At higher CPU clock frequencies time can be extended for I/O reads (and/or writes) by inserting wait states (8080) or using clock extending circuits (6800).

Finally, if time is short and capacitive loading is high, external bus drivers must be used. These can be TRI-STATE buffers (low power Schottky such as the DM74LS240 series is recommended) or special higher drive current products which are designed as bus drivers. High current bipolar bus drivers with PNP inputs are recommended.

2.10 Power Supplies

Noise spikes on the V_{CC} supply line can cause conversion errors as the comparator will respond to this noise. A low inductance tantalum filter capacitor should be used close to the converter V_{CC} pin and values of 1 µF or greater are recommended. If an unregulated voltage is available in the system, a separate LM340LAZ-5.0, TO-92, 5V voltage regulator for the converter (and other analog circuitry) will greatly reduce digital noise on the V_{CC} supply.

2.11 Wiring and Hook-Up Precautions

Standard digital wire wrap sockets are not satisfactory for breadboarding this A/D converter. Sockets on PC boards can be used and all logic signal wires and leads should be grouped and kept as far away as possible from the analog signal leads. Exposed leads to the analog inputs can cause undesired digital noise and hum pickup, therefore shielded leads may be necessary in many applications.

A single point analog ground that is separate from the logic ground points should be used. The power supply bypass capacitor and the self-clocking capacitor (if used) should both be returned to digital ground. Any $V_{REF}/2$ bypass capacitors, analog input filter capacitors, or input signal shielding should be returned to the analog ground point. A test for proper grounding is to measure the zero error of the A/D converter. Zero errors in excess of ¼ LSB can usually be traced to improper board layout and wiring (see section 2.5.1 for measuring the zero error).

3.0 TESTING THE A/D CONVERTER

There are many degrees of complexity associated with testing an A/D converter. One of the simplest tests is to apply a known analog input voltage to the converter and use LEDs to display the resulting digital output code as shown in *Figure 9*.

For ease of testing, the $V_{REF}/2$ (pin 9) should be supplied with 2.560 V_{DC} and a V_{CC} supply voltage of 5.12 V_{DC} should be used. This provides an LSB value of 20 mV.

If a full-scale adjustment is to be made, an analog input voltage of 5.090 V_{DC} (5.120–1½ LSB) should be applied to the $V_{IN}(+)$ pin with the $V_{IN}(-)$ pin grounded. The value of the $V_{REF}/2$ input voltage should then be adjusted until the digital output code is just changing from 1111 1110 to 1111 1111. This value of $V_{REF}/2$ should then be used for all the tests.

The digital output LED display can be decoded by dividing the 8 bits into 2 hex characters, the 4 most significant (MS) and the 4 least significant (LS). *Table 1* shows the fractional binary equivalent of these two 4-bit groups. By adding the voltages obtained from the "VMS" and "VLS" columns in *Table 1*, the nominal value of the digital display (when $V_{REF}/2$ = 2.560V) can be determined. For example, for an output LED display of 1011 0110 or B6 (in hex), the voltage values from the table are 3.520 + 0.120 or 3.640 V_{DC}. These voltage values represent the center-values of a perfect A/D converter. The effects of quantization error have to be accounted for in the interpretation of the test results.

Functional Description (Continued)

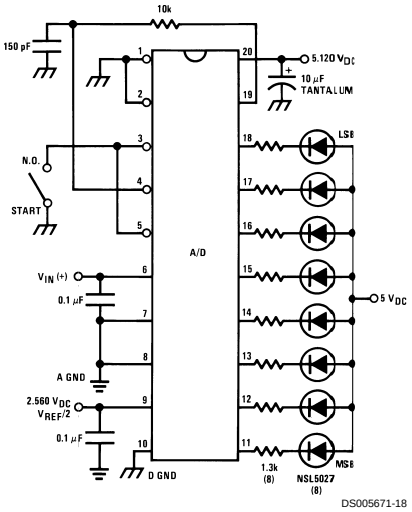


FIGURE 9. Basic A/D Tester

For a higher speed test system, or to obtain plotted data, a digital-to-analog converter is needed for the test set-up. An accurate 10-bit DAC can serve as the precision voltage source for the A/D. Errors of the A/D under test can be expressed as either analog voltages or differences in 2 digital words.

A basic A/D tester that uses a DAC and provides the error as an analog output voltage is shown in *Figure 8*. The 2 op amps can be eliminated if a lab DVM with a numerical subtraction feature is available to read the difference voltage, "A–C", directly. The analog input voltage can be supplied by a low frequency ramp generator and an X-Y plotter can be used to provide analog error (Y axis) versus analog input (X axis).

For operation with a microprocessor or a computer-based test system, it is more convenient to present the errors digitally. This can be done with the circuit of *Figure 11*, where the output code transitions can be detected as the 10-bit DAC is incremented. This provides ¼ LSB steps for the 8-bit A/D under test. If the results of this test are automatically plotted with the analog input on the X axis and the error (in LSB's) as the Y axis, a useful transfer function of the A/D under test results. For acceptance testing, the plot is not necessary and the testing speed can be increased by establishing internal limits on the allowed error for each code.

4.0 MICROPROCESSOR INTERFACING

To dicuss the interface with 8080A and 6800 microprocessors, a common sample subroutine structure is used. The microprocessor starts the A/D, reads and stores the results of 16 successive conversions, then returns to the user's program. The 16 data bytes are stored in 16 successive memory locations. All Data and Addresses will be given in hexadecimal form. Software and hardware details are provided separately for each type of microprocessor.

4.1 Interfacing 8080 Microprocessor Derivatives (8048, 8085)

This converter has been designed to directly interface with derivatives of the 8080 microprocessor. The A/D can be mapped into memory space (using standard memory address decoding for $\overline{CS}$ and the $\overline{MEMR}$ and $\overline{MEMW}$ strobes) or it can be controlled as an I/O device by using the $\overline{I/O\ R}$ and $\overline{I/O\ W}$ strobes and decoding the address bits A0 → A7 (or address bits A8 → A15 as they will contain the same 8-bit address information) to obtain the $\overline{CS}$ input. Using the I/O space provides 256 additional addresses and may allow a simpler 8-bit address decoder but the data can only be input to the accumulator. To make use of the additional memory reference instructions, the A/D should be mapped into memory space. An example of an A/D in I/O space is shown in *Figure 12*.

Functional Description (Continued)

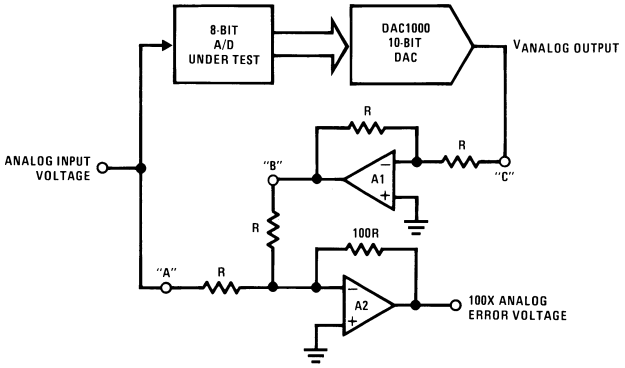


FIGURE 10. A/D Tester with Analog Error Output

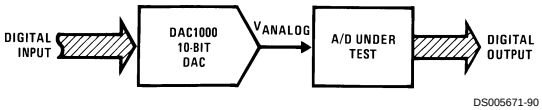


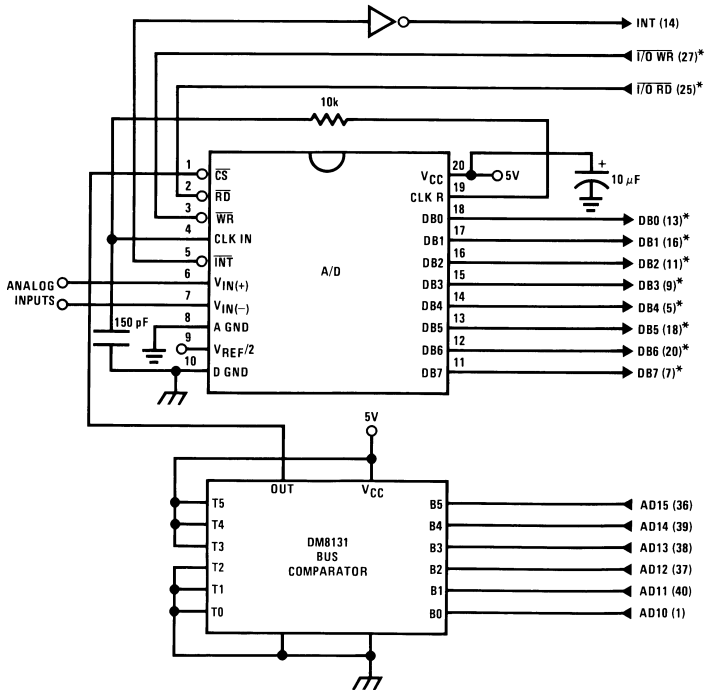
FIGURE 11. Basic "Digital" A/D Tester

TABLE 1. DECODING THE DIGITAL OUTPUT LEDs

HEX	BINARY				FRACTIONAL BINARY VALUE FOR		OUTPUT VOLTAGE CENTER VALUES WITH $V_{REF}/2=2.560\ V_{DC}$	
					MS GROUP	LS GROUP	VMS GROUP (Note 15)	VLS GROUP (Note 15)
F	1	1	1	1	15/16	15/256	4.800	0.300
E	1	1	1	0	7/8	7/128	4.480	0.280
D	1	1	0	1	13/16	13/256	4.160	0.260
C	1	1	0	0	3/4	3/64	3.840	0.240
B	1	0	1	1	11/16	11/256	3.520	0.220
A	1	0	1	0	5/8	5/128	3.200	0.200
9	1	0	0	1	9/16	9/256	2.880	0.180
8	1	0	0	0	1/2	1/32	2.560	0.160
7	0	1	1	1	7/16	7/256	2.240	0.140
6	0	1	1	0	3/8	3/128	1.920	0.120
5	0	1	0	1	5/16	2/256	1.600	0.100
4	0	1	0	0	1/4	1/64	1.280	0.080
3	0	0	1	1	3/16	3/256	0.960	0.060
2	0	0	1	0	1/8	1/128	0.640	0.040
1	0	0	0	1	1/16	1/256	0.320	0.020
0	0	0	0	0			0	0

Note 15: Display Output=VMS Group + VLS Group

Functional Description (Continued)



DS005671-20

Note 16: *Pin numbers for the DP8228 system controller, others are INS8080A.

Note 17: Pin 23 of the INS8228 must be tied to +12V through a 1 kΩ resistor to generate the RST 7 instruction when an interrupt is acknowledged as required by the accompanying sample program.

FIGURE 12. ADC0801_INS8080A CPU Interface

Functional Description (Continued)

SAMPLE PROGRAM FOR *Figure 12* ADC0801–INS8080A CPU INTERFACE

```
0038    C3 00 03    RST 7:          JMP     LD DATA
  •       •           •
  •       •           •
0100    21 00 02    START:          LXI H 0200H         ;HL pair will point to
                                                        ;data storage locations
0103    31 00 04    RETURN:         LXI SP 0400H        ;Initialize stack pointer (Note 1)
0106    7D                          MOV A, L            ;Test # of bytes entered
0107    FE 0F                       CPI 0F H            ;If # = 16. JMP to
0109    CA 13 01                    JZ CONT             ;user program
010C    D3 E0                       OUT E0 H            ;Start A/D
010E    FB                          EI                  ;Enable interrupt
010F    00          LOOP:           NOP                 ;Loop until end of
0110    C3 0F 01                    JMP LOOP            ;conversion
0113      •         CONT:             •
  •       •           •               •
  •       •         (User program to  •
  •       •         process data)     •
  •       •           •               •
  •       •           •               •
0300    DB E0       LD DATA:        IN E0 H             ;Load data into accumulator
0302    77                          MOV M, A            ;Store data
0303    23                          INX H               ;Increment storage pointer
0304    C3 03 01                    JMP RETURN
```

DS005671-99

Note 18: The stack pointer must be dimensioned because a RST 7 instruction pushes the PC onto the stack.

Note 19: All address used were arbitrarily chosen.

The standard control bus signals of the 8080 $\overline{CS}$, $\overline{RD}$ and $\overline{WR}$) can be directly wired to the digital control inputs of the A/D and the bus timing requirements are met to allow both starting the converter and outputting the data onto the data bus. A bus driver should be used for larger microprocessor systems where the data bus leaves the PC board and/or must drive capacitive loads larger than 100 pF.

4.1.1 Sample 8080A CPU Interfacing Circuitry and Program

The following sample program and associated hardware shown in *Figure 12* may be used to input data from the converter to the INS8080A CPU chip set (comprised of the INS8080A microprocessor, the INS8228 system controller and the INS8224 clock generator). For simplicity, the A/D is controlled as an I/O device, specifically an 8-bit bi-directional port located at an arbitrarily chosen port address, E0. The TRI-STATE output capability of the A/D eliminates the need for a peripheral interface device, however address decoding is still required to generate the appropriate $\overline{CS}$ for the converter.

It is important to note that in systems where the A/D converter is 1-of-8 or less I/O mapped devices, no address decoding circuitry is necessary. Each of the 8 address bits (A0 to A7) can be directly used as $\overline{CS}$ inputs — one for each I/O device.

4.1.2 INS8048 Interface

The INS8048 interface technique with the ADC0801 series (see *Figure 13*) is simpler than the 8080A CPU interface. There are 24 I/O lines and three test input lines in the 8048. With these extra I/O lines available, one of the I/O lines (bit 0 of port 1) is used as the chip select signal to the A/D, thus eliminating the use of an external address decoder. Bus control signals $\overline{RD}$, $\overline{WR}$ and $\overline{INT}$ of the 8048 are tied directly to the A/D. The 16 converted data words are stored at on-chip RAM locations from 20 to 2F (Hex). The $\overline{RD}$ and $\overline{WR}$ signals are generated by reading from and writing into a dummy address, respectively. A sample interface program is shown below.

Functional Description (Continued)

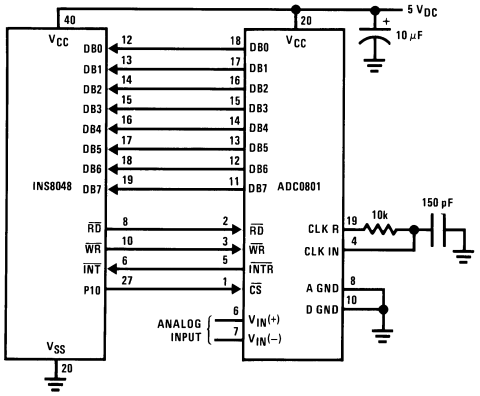


FIGURE 13. INS8048 Interface

SAMPLE PROGRAM FOR *Figure 13* **INS8048 INTERFACE**

```
04 10                  JMP     10H          ;Program starts at addr 10
                       ORG     3H
04 50                  JMP     50H          ;Interrupt jump vector
                       ORG     10H          ;Main program
99 FE                  ANL     P1, #0FEH    ;Chip select
81                     MOVX    A, @R1       ;Read in the 1st data
                                            ;to reset the intr
89 01        START:    ORL     P1, #1       ;Set port pin high
B8 20                  MOV     R0, #20H     ;Data address
B9 FF                  MOV     R1, #0FFH    ;Dummy address
BA 10                  MOV     R2, #10H     ;Counter for 16 bytes
23 FF        AGAIN:    MOV     A, #0FFH     ;Set ACC for intr loop
99 FE                  ANL     P1, #0FEH    ;Send CS (bit 0 of P1)
91                     MOVX    @R1, A       ;Send WR out
05                     EN      I            ;Enable interrupt
96 21        LOOP:     JNZ     LOOP         ;Wait for interrupt
EA 1B                  DJNZ    R2, AGAIN    ;If 16 bytes are read
00                     NOP                  ;go to user's program
00                     NOP
                       ORG     50H
81           INDATA:   MOVX    A, @R1       ;Input data, CS still low
A0                     MOV     @R0, A       ;Store in memory
18                     INC     R0           ;Increment storage counter
89 01                  ORL     P1, #1       ;Reset CS signal
27                     CLR     A            ;Clear ACC to get out of
93                     RETR                 ;the interrupt loop
```

DS005671-A0

4.2 Interfacing the Z-80

The Z-80 control bus is slightly different from that of the 8080. General $\overline{RD}$ and $\overline{WR}$ strobes are provided and separate memory request, $\overline{MREQ}$, and I/O request, $\overline{IORQ}$, signals are used which have to be combined with the generalized strobes to provide the equivalent 8080 signals. An advantage of operating the A/D in I/O space with the Z-80 is that the CPU will automatically insert one wait state (the $\overline{RD}$ and $\overline{WR}$ strobes are extended one clock period) to allow more time for the I/O devices to respond. Logic to map the A/D in I/O space is shown in *Figure 14*.

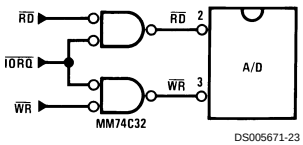


FIGURE 14. Mapping the A/D as an I/O Device for Use with the Z-80 CPU

Additional I/O advantages exist as software DMA routines are available and use can be made of the output data transfer which exists on the upper 8 address lines (A8 to A15) dur-

Functional Description (Continued)

ing I/O input instructions. For example, MUX channel selection for the A/D can be accomplished with this operating mode.

4.3 Interfacing 6800 Microprocessor Derivatives (6502, etc.)

The control bus for the 6800 microprocessor derivatives does not use the $\overline{RD}$ and $\overline{WR}$ strobe signals. Instead it employs a single R/$\overline{W}$ line and additional timing, if needed, can be derived fom the $\phi 2$ clock. All I/O devices are memory mapped in the 6800 system, and a special signal, VMA, indicates that the current address is valid. *Figure 15* shows an interface schematic where the A/D is memory mapped in the 6800 system. For simplicity, the $\overline{CS}$ decoding is shown using ½ DM8092. Note that in many 6800 systems, an already decoded $\overline{4/5}$ line is brought out to the common bus at pin 21. This can be tied directly to the $\overline{CS}$ pin of the A/D, provided that no other devices are addressed at HX ADDR: 4XXX or 5XXX.

The following subroutine performs essentially the same function as in the case of the 8080A interface and it can be called from anywhere in the user's program.

In *Figure 16* the ADC0801 series is interfaced to the M6800 microprocessor through (the arbitrarily chosen) Port B of the MC6820 or MC6821 Peripheral Interface Adapter, (PIA). Here the $\overline{CS}$ pin of the A/D is grounded since the PIA is already memory mapped in the M6800 system and no $\overline{CS}$ decoding is necessary. Also notice that the A/D output data lines are connected to the microprocessor bus under program control through the PIA and therefore the A/D $\overline{RD}$ pin can be grounded.

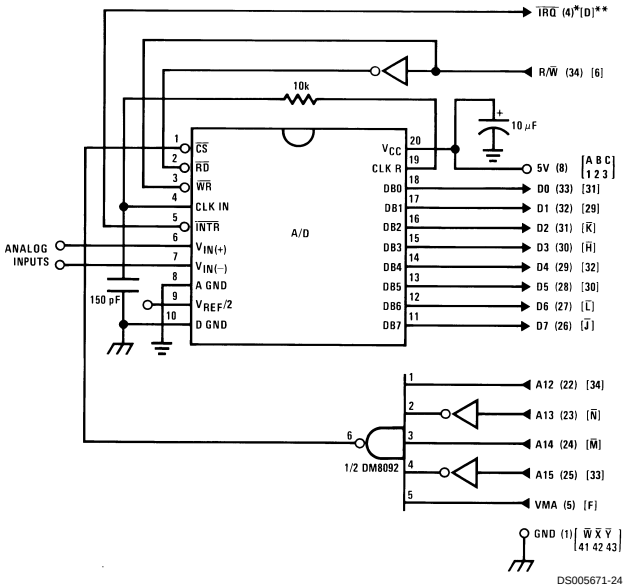
A sample interface program equivalent to the previous one is shown below *Figure 16*. The PIA Data and Control Registers of Port B are located at HEX addresses 8006 and 8007, respectively.

5.0 GENERAL APPLICATIONS

The following applications show some interesting uses for the A/D. The fact that one particular microprocessor is used is not meant to be restrictive. Each of these application circuits would have its counterpart using any microprocessor that is desired.

5.1 Multiple ADC0801 Series to MC6800 CPU Interface

To transfer analog data from several channels to a single microprocessor system, a multiple converter scheme presents several advantages over the conventional multiplexer single-converter approach. With the ADC0801 series, the differential inputs allow individual span adjustment for each channel. Furthermore, all analog input channels are sensed simultaneously, which essentially divides the microprocessor's total system servicing time by the number of channels, since all conversions occur simultaneously. This scheme is shown in *Figure 17*.



Note 20: Numbers in parentheses refer to MC6800 CPU pin out.

Note 21: Number or letters in brackets refer to standard M6800 system common bus code.

FIGURE 15. ADC0801-MC6800 CPU Interface

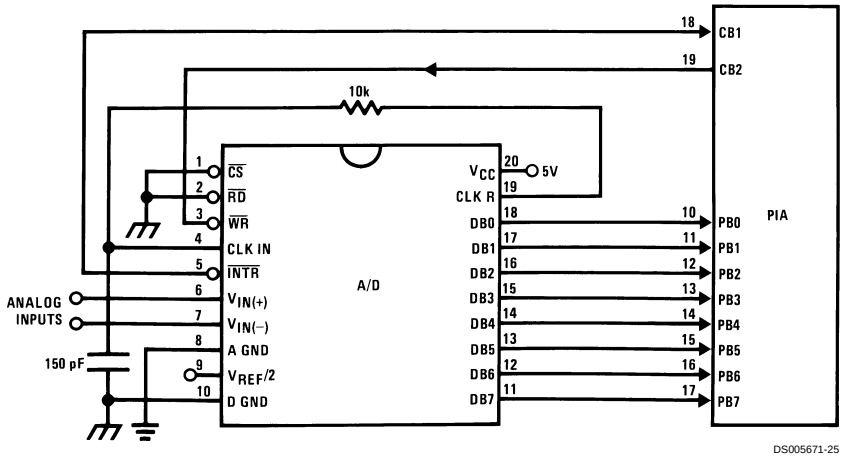
Functional Description (Continued)

SAMPLE PROGRAM FOR *Figure 15* ADC0801-MC6800 CPU INTERFACE

```
0010    DF 36        DATAIN    STX     TEMP2     ; Save contents of X
0012    CE 00 2C               LDX     #$002C    ; Upon IRQ low CPU
0015    FF FF F8               STX     $FFF8     ; jumps to 002C
0018    B7 50 00               STAA    $5000     ; Start ADC0801
001B    0E                     CLI
001C    3E           CONVRT    WAI               ; Wait for interrupt
001D    DE 34                  LDX     TEMP1
001F    8C 02 0F               CPX     #$020F    ; Is final data stored?
0022    27 14                  BEQ     ENDP
0024    B7 50 00               STAA    $5000     ; Restarts ADC0801
0027    08                     INX
0028    DF 34                  STX     TEMP1
002A    20 F0                  BRA     CONVRT
002C    DE 34        INTRPT    LDX     TEMP1
002E    B6 50 00               LDAA    $5000     ; Read data
0031    A7 00                  STAA    X         ; Store it at X
0033    3B                     RTI
0034    02 00        TEMP1     FDB     $0200     ; Starting address for
                                                 ; data storage
0036    00 00        TEMP2     FDB     $0000
0038    CE 02 00     ENDP      LDX     #$0200    ; Reinitialize TEMP1
003B    DF 34                  STX     TEMP1
003D    DE 36                  LDX     TEMP2
003F    39                     RTS               ; Return from subroutine
                                                 ; To user's program
```

DS005671-A1

Note 22: In order for the microprocessor to service subroutines and interrupts, the stack pointer must be dimensioned in the user's program.



DS005671-25

FIGURE 16. ADC0801–MC6820 PIA Interface

Functional Description (Continued)

SAMPLE PROGRAM FOR *Figure 16* ADC0801–MC6820 PIA INTERFACE

```
0010    CE 00 38    DATAIN    LDX     #$0038     ; Upon IRQ low CPU
0013    FF FF F8              STX     $FFF8      ; jumps to 0038
0016    B6 80 06              LDAA    PIAORB     ; Clear possible IRQ flags
0019    4F                    CLRA
001A    B7 80 07              STAA    PIACRB
001D    B7 80 06              STAA    PIAORB     ; Set Port B as input
0020    0E                    CLI
0021    C6 34                 LDAB    #$34
0023    86 3D                 LDAA    #$3D
0025    F7 80 07    CONVRT    STAB    PIACRB     ; Starts ADC0801
0028    B7 80 07              STAA    PIACRB
002B    3E                    WAI                ; Wait for interrupt
002C    DE 40                 LDX     TEMP1
002E    8C 02 0F              CPX     #$020F     ; Is final data stored?
0031    27 0F                 BEQ     ENDP
0033    08                    INX
0034    DF 40                 STX     TEMP1
0036    20 ED                 BRA     CONVRT
0038    DE 40       INTRPT    LDX     TEMP1
003A    B6 80 06              LDAA    PIAORB     ; Read data in
003D    A7 00                 STAA    X          ; Store it at X
003F    3B                    RTI
0040    02 00       TEMP1     FDB     $0200      ; Starting address for
                                                 ; data storage
0042    CE 02 00    ENDP      LDX     #$0200     ; Reinitialize TEMP1
0045    DF 40                 STX     TEMP1
0047    39                    RTS                ; Return from subroutine
                    PIAORB    EQU     $8006      ; To user's program
                    PIACRB    EQU     $8007
```

DS005671-A2

The following schematic and sample subroutine (DATA IN) may be used to interface (up to) 8 ADC0801's directly to the MC6800 CPU. This scheme can easily be extended to allow the interface of more converters. In this configuration the converters are (arbitrarily) located at HEX address 5000 in the MC6800 memory space. To save components, the clock signal is derived from just one RC pair on the first converter. This output drives the other A/Ds.

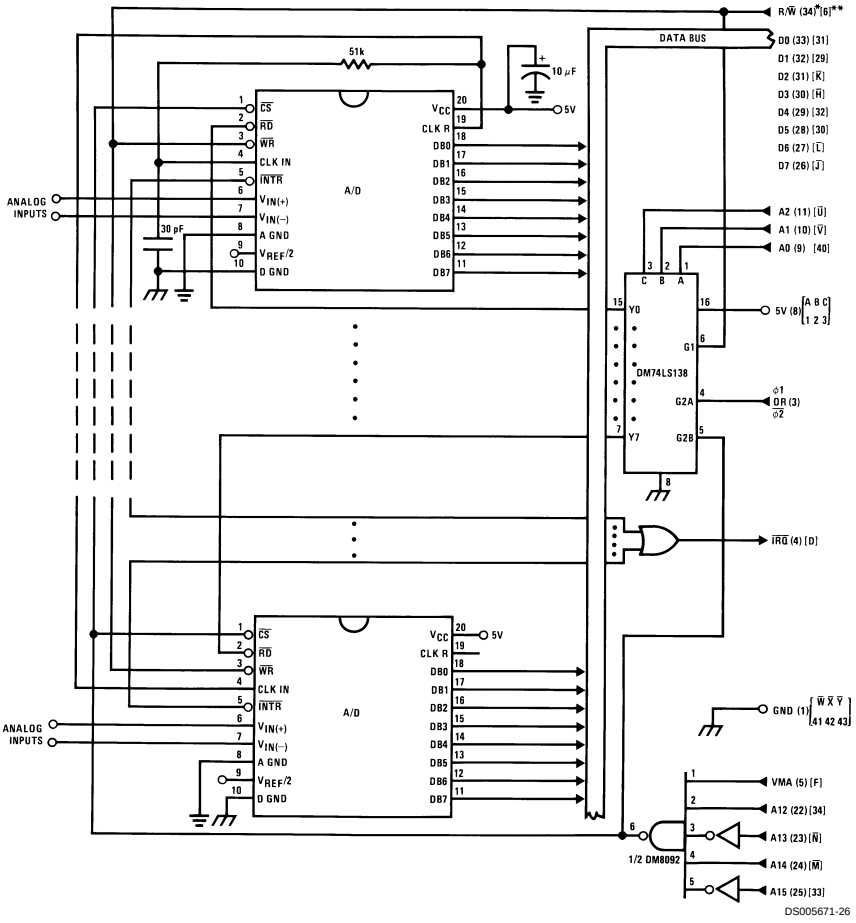
All the converters are started simultaneously with a STORE instruction at HEX address 5000. Note that any other HEX address of the form 5XXX will be decoded by the circuit, pulling all the $\overline{CS}$ inputs low. This can easily be avoided by using a more definitive address decoding scheme. All the interrupts are ORed together to insure that all A/Ds have completed their conversion before the microprocessor is interrupted.

The subroutine, DATA IN, may be called from anywhere in the user's program. Once called, this routine initializes the CPU, starts all the converters simultaneously and waits for the interrupt signal. Upon receiving the interrupt, it reads the converters (from HEX addresses 5000 through 5007) and stores the data successively at (arbitrarily chosen) HEX addresses 0200 to 0207, before returning to the user's program. All CPU registers then recover the original data they had before servicing DATA IN.

5.2 Auto-Zeroed Differential Transducer Amplifier and A/D Converter

The differential inputs of the ADC0801 series eliminate the need to perform a differential to single ended conversion for a differential transducer. Thus, one op amp can be eliminated since the differential to single ended conversion is provided by the differential input of the ADC0801 series. In general, a transducer preamp is required to take advantage of the full A/D converter input dynamic range.

Functional Description (Continued)



Note 23: Numbers in parentheses refer to MC6800 CPU pin out.

Note 24: Numbers of letters in brackets refer to standard M6800 system common bus code.

FIGURE 17. Interfacing Multiple A/Ds in an MC6800 System

Functional Description (Continued)

SAMPLE PROGRAM FOR *Figure 17* INTERFACING MULTIPLE A/D's IN AN MC6800 SYSTEM

ADDRESS	HEX CODE		MNEMONICS		COMMENTS
0010	DF 44	DATAIN	STX	TEMP	; Save Contents of X
0012	CE 00 2A		LDX	#$002A	; Upon $\overline{IRQ}$ LOW CPU
0015	FF FF F8		STX	$FFF8	; Jumps to 002A
0018	B7 50 00		STAA	$5000	; Starts all A/D's
001B	0E		CLI		
001C	3E		WAI		; Wait for interrupt
001D	CE 50 00		LDX	#$5000	
0020	DF 40		STX	INDEX1	; Reset both INDEX
0022	CE 02 00		LDX	#$0200	; 1 and 2 to starting
0025	DF 42		STX	INDEX2	; addresses
0027	DE 44		LDX	TEMP	
0029	39		RTS		; Return from subroutine
002A	DE 40	INTRPT	LDX	INDEX1	; INDEX1 → X
002C	A6 00		LDAA	X	; Read data in from A/D at X
002E	08		INX		; Increment X by one
002F	DF 40		STX	INDEX1	; X → INDEX1
0031	DE 42		LDX	INDEX2	; INDEX2 → X

DS005671-A3

SAMPLE PROGRAM FOR *Figure 17* INTERFACING MULTIPLE A/D's IN AN MC6800 SYSTEM

ADDRESS	HEX CODE		MNEMONICS		COMMENTS
0033	A7 00		STAA	X	; Store data at X
0035	8C 02 07		CPX	#$0207	; Have all A/D's been read?
0038	27 05		BEQ	RETURN	; Yes: branch to RETURN
003A	08		INX		; No: increment X by one
003B	DF 42		STX	INDEX2	; X → INDEX2
003D	20 EB		BRA	INTRPT	; Branch to 002A
003F	3B	RETURN	RTI		
0040	50 00	INDEX1	FDB	$5000	; Starting address for A/D
0042	02 00	INDEX2	FDB	$0200	; Starting address for data storage
0044	00 00	TEMP	FDB	$0000	

DS005671-A4

Note 25: In order for the microprocessor to service subroutines and interrupts, the stack pointer must be dimensioned in the user's program.

For amplification of DC input signals, a major system error is the input offset voltage of the amplifiers used for the preamp. *Figure 18* is a gain of 100 differential preamp whose offset voltage errors will be cancelled by a zeroing subroutine which is performed by the INS8080A microprocessor system. The total allowable input offset voltage error for this preamp is only 50 μV for ¼ LSB error. This would obviously require very precise amplifiers. The expression for the differential output voltage of the preamp is:

$$V_O = \underbrace{[V_{IN}(+) - V_{IN}(-)]}_{\text{SIGNAL}} \underbrace{\left[1 + \frac{2R2}{R1}\right]}_{\text{GAIN}} +$$

$$\underbrace{(V_{OS_2} - V_{OS_1} - V_{OS_3} \pm I_X R_X)}_{\text{DC ERROR TERM}} \underbrace{\left(1 + \frac{2R2}{R1}\right)}_{\text{GAIN}}$$

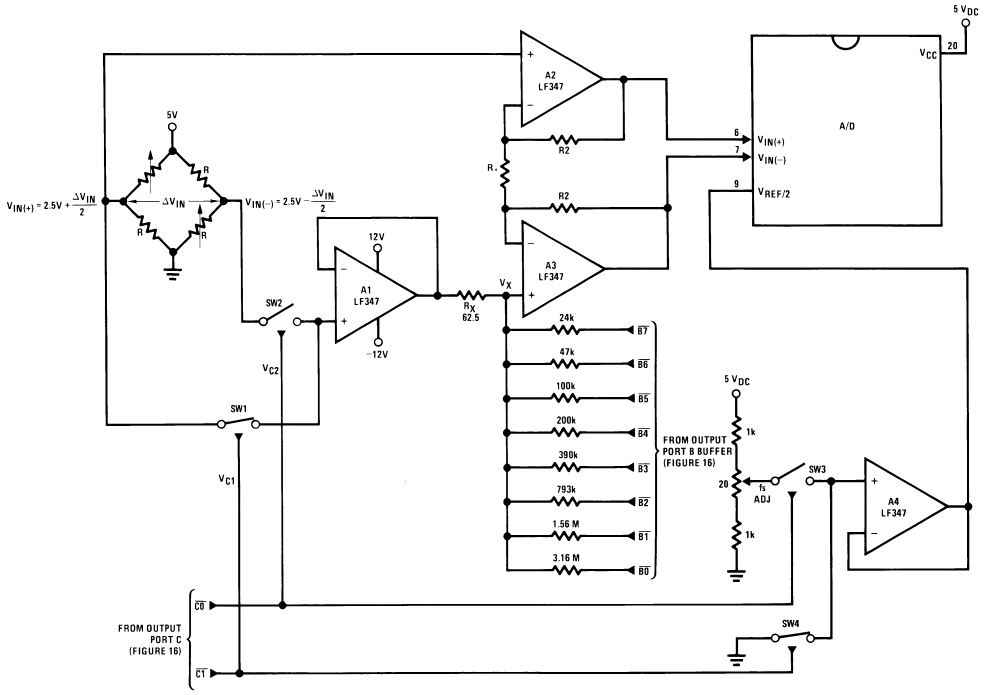
where I_X is the current through resistor R_X. All of the offset error terms can be cancelled by making $\pm I_X R_X = V_{OS1} + V_{OS3} - V_{OS2}$. This is the principle of this auto-zeroing scheme.

The INS8080A uses the 3 I/O ports of an INS8255 Programable Peripheral Interface (PPI) to control the auto zeroing and input data from the ADC0801 as shown in *Figure 19*. The PPI is programmed for basic I/O operation (mode 0) with Port A being an input port and Ports B and C being output ports. Two bits of Port C are used to alternately open or close the 2 switches at the input of the preamp. Switch SW1 is closed to force the preamp's differential input to be zero during the zeroing subroutine and then opened and SW2 is then closed for conversion of the actual differential input signal. Using 2 switches in this manner eliminates concern for the ON resistance of the switches as they must conduct only the input bias current of the input amplifiers.

Output Port B is used as a successive approximation register by the 8080 and the binary scaled resistors in series with each output bit create a D/A converter. During the zeroing subroutine, the voltage at V_x increases or decreases as required to make the differential output voltage equal to zero. This is accomplished by ensuring that the voltage at the output of A1 is approximately 2.5V so that a logic "1" (5V) on

Functional Description (Continued)

any output of Port B will source current into node V_X thus raising the voltage at V_X and making the output differential more negative. Conversely, a logic "0" (0V) will pull current out of node V_X and decrease the voltage, causing the differential output to become more positive. For the resistor values shown, V_X can move ±12 mV with a resolution of 50 µV, which will null the offset error term to ¼ LSB of full-scale for the ADC0801. It is important that the voltage levels that drive the auto-zero resistors be constant. Also, for symmetry, a logic swing of 0V to 5V is convenient. To achieve this, a CMOS buffer is used for the logic output signals of Port B and this CMOS package is powered with a stable 5V source. Buffer amplifier A1 is necessary so that it can source or sink the D/A output current.



DS005671-91

Note 26: R2 = 49.5 R1

Note 27: Switches are LMC13334 CMOS analog switches.

Note 28: The 9 resistors used in the auto-zero section can be ±5% tolerance.

FIGURE 18. Gain of 100 Differential Transducer Preamp

Functional Description (Continued)

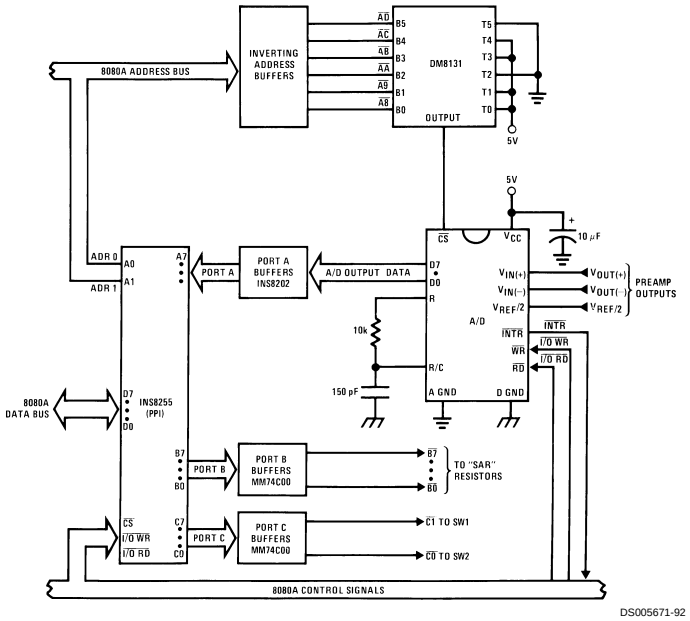


FIGURE 19. Microprocessor Interface Circuitry for Differential Preamp

A flow chart for the zeroing subroutine is shown in *Figure 20*. It must be noted that the ADC0801 series will output an all zero code when it converts a negative input [$V_{IN}(-) \geq V_{IN}(+)$]. Also, a logic inversion exists as all of the I/O ports are buffered with inverting gates.

Basically, if the data read is zero, the differential output voltage is negative, so a bit in Port B is cleared to pull V_X more negative which will make the output more positive for the next conversion. If the data read is not zero, the output voltage is positive so a bit in Port B is set to make V_X more positive and the output more negative. This continues for 8 approximations and the differential output eventually converges to within 5 mV of zero.

The actual program is given in *Figure 21*. All addresses used are compatible with the BLC 80/10 microcomputer system. In particular:

Port A and the ADC0801 are at port address E4

Port B is at port address E5

Port C is at port address E6

PPI control word port is at port address E7

Program Counter automatically goes to ADDR:3C3D upon acknowledgement of an interrupt from the ADC0801

5.3 Multiple A/D Converters in a Z-80 Interrupt Driven Mode

In data acquisition systems where more than one A/D converter (or other peripheral device) will be interrupting program execution of a microprocessor, there is obviously a need for the CPU to determine which device requires servicing. *Figure 22* and the accompanying software is a method of determining which of 7 ADC0801 converters has completed a conversion ($\overline{INTR}$ asserted) and is requesting an interrupt. This circuit allows starting the A/D converters in any sequence, but will input and store valid data from the converters with a priority sequence of A/D 1 being read first, A/D 2 second, etc., through A/D 7 which would have the lowest priority for data being read. Only the converters whose INT is asserted will be read.

The key to decoding circuitry is the DM74LS373, 8-bit D type flip-flop. When the Z-80 acknowledges the interrupt, the program is vectored to a data input Z-80 subroutine. This subroutine will read a peripheral status word from the DM74LS373 which contains the logic state of the $\overline{INTR}$ outputs of all the converters. Each converter which initiates an interrupt will place a logic "0" in a unique bit position in the status word and the subroutine will determine the identity of the converter and execute a data read. An identifier word (which indicates which A/D the data came from) is stored in the next sequential memory location above the location of the data so the program can keep track of the identity of the data entered.

Functional Description (Continued)

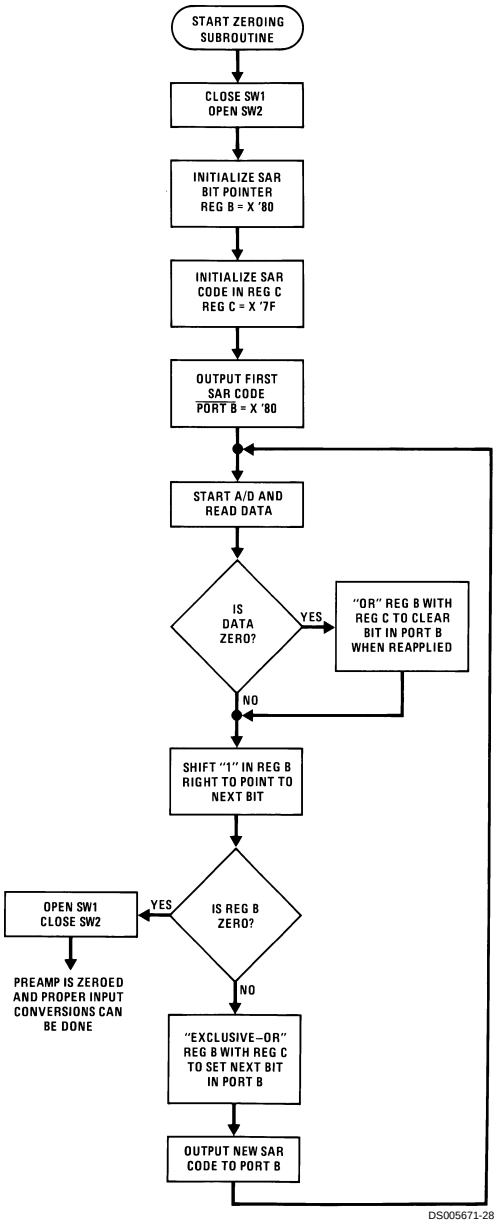


FIGURE 20. Flow Chart for Auto-Zero Routine

Functional Description (Continued)

```
3D00   3E90     MVI 90
3D02   D3E7     Out Control Port                          ;Program PPI
3D04   2601     MVI H 01          Auto-Zero Subroutine
3D06   7C       MOV A,H
3D07   D3E6     OUT C                                    ;Close SW1 open SW2
3D09   0680     MVI B 80                                 ;Initialize SAR bit pointer
3D0B   3E7F     MVI A 7F                                 ;Initialize SAR code
3D0D   4F       MOV C,A           Return
3D0E   D3E5     OUT B                                    ;Port B = SAR code
3D10   31AA3D   LXI SP 3DAA       Start                  ;Dimension stack pointer
3D13   D3E4     OUT A                                    ;Start A/D
3D15   FB       IE
3D16   00       NOP               Loop                   ;Loop until INT asserted
3D17   C3163D   JMP Loop
3D1A   7A       MOV A,D           Auto-Zero
3D1B   C600     ADI 00
3D1D   CA2D3D   JZ Set C                                 ;Test A/D output data for zero
3D20   78       MOV A,B           Shift B
3D21   F600     ORI 00                                   ;Clear carry
3D23   1F       RAR                                      ;Shift "1" in B right one place
3D24   FE00     CPI 00                                   ;Is B zero? If yes last
3D26   CA373D   JZ Done                                  ;approximation has been made
3D29   47       MOV B,A
3D2A   C3333D   JMP New C
3D2D   79       MOV A,C           Set C
3D2E   B0       ORA B                                    ;Set bit in C that is in same
3D2F   4F       MOV C,A                                  ;position as "1" in B
3D30   C3203D   JMP Shift B
3D33   A9       XRA C             New C                  ;Clear bit in C that is in
3D34   C30D3D   JMP Return                               ;same position as "1" in B
3D37   47       MOV B,A           Done                   ;then output new SAR code.
3D38   7C       MOV A,H                                  ;Open SW1, close SW2 then
3D39   EE03     XRI 03                                   ;proceed with program. Preamp
3D3B   D3E6     OUT C                                    ;is now zeroed.
3D3D            •                 Normal
                •
                •
                Program for processing
                proper data values
3C3D   DBE4     IN A              Read A/D Subroutine    ;Read A/D data
3C3F   EEFF     XRI FF                                   ;Invert data
3C41   57       MOV D,A
3C42   78       MOV A,B                                  ;Is B Reg = 0? If not stay
3C43   E6FF     ANI FF                                   ;in auto zero subroutine
3C45   C21A3D   JNZ Auto-Zero
3C48   C33D3D   JMP Normal
```

DS005671-A5

Note 29: All numerical values are hexadecimal representations.

FIGURE 21. Software for Auto-Zeroed Differential A/D

5.3 Multiple A/D Converters in a Z-80 Interrupt Driven Mode (Continued)

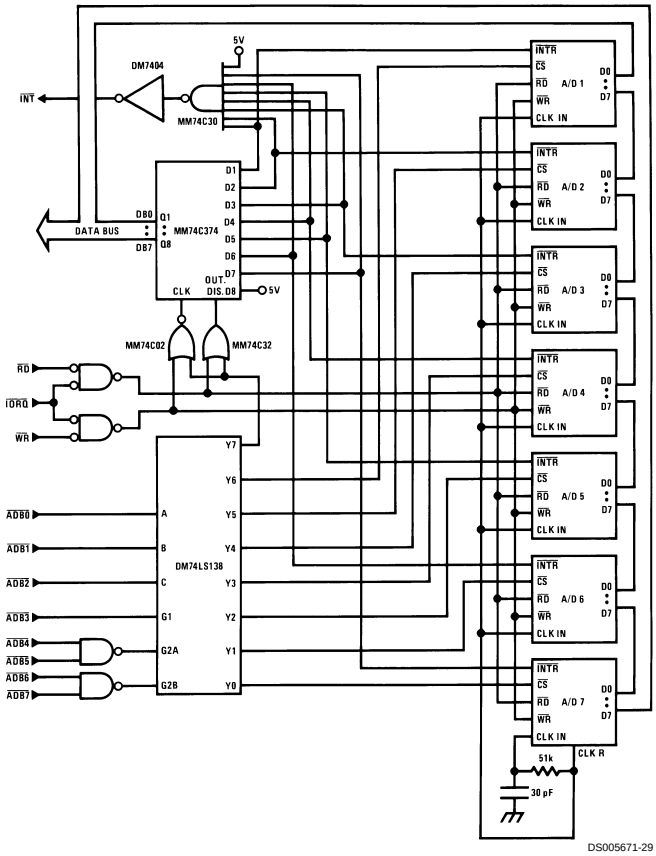
The following notes apply:

- It is assumed that the CPU automatically performs a RST 7 instruction when a valid interrupt is acknowledged (CPU is in interrupt mode 1). Hence, the subroutine starting address of X0038.
- The address bus from the Z-80 and the data bus to the Z-80 are assumed to be inverted by bus drivers.
- A/D data and identifying words will be stored in sequential memory locations starting at the arbitrarily chosen address X 3E00.
- The stack pointer must be dimensioned in the main program as the RST 7 instruction automatically pushes the PC onto the stack and the subroutine uses an additional 6 stack addresses.
- The peripherals of concern are mapped into I/O space with the following port assignments:

Functional Description (Continued)

HEX PORT ADDRESS	PERIPHERAL
00	MM74C374 8-bit flip-flop
01	A/D 1
02	A/D 2
03	A/D 3
04	A/D 4
05	A/D 5
06	A/D 6
07	A/D 7

This port address also serves as the A/D identifying word in the program.



DS005671-29

FIGURE 22. Multiple A/Ds with Z-80 Type Microprocessor

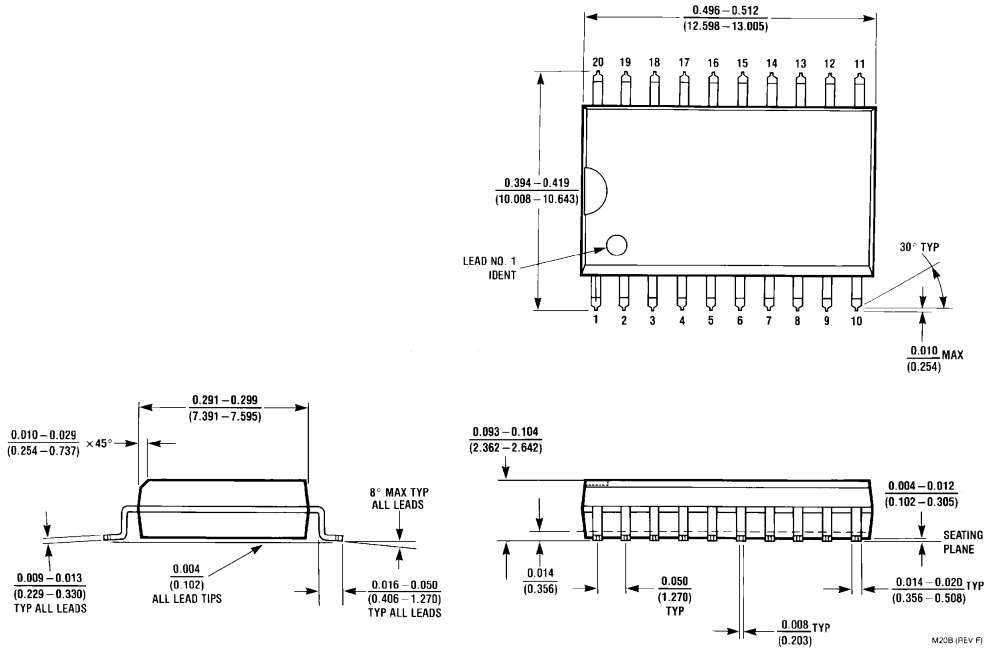
Functional Description (Continued)

INTERRUPT SERVICING SUBROUTINE

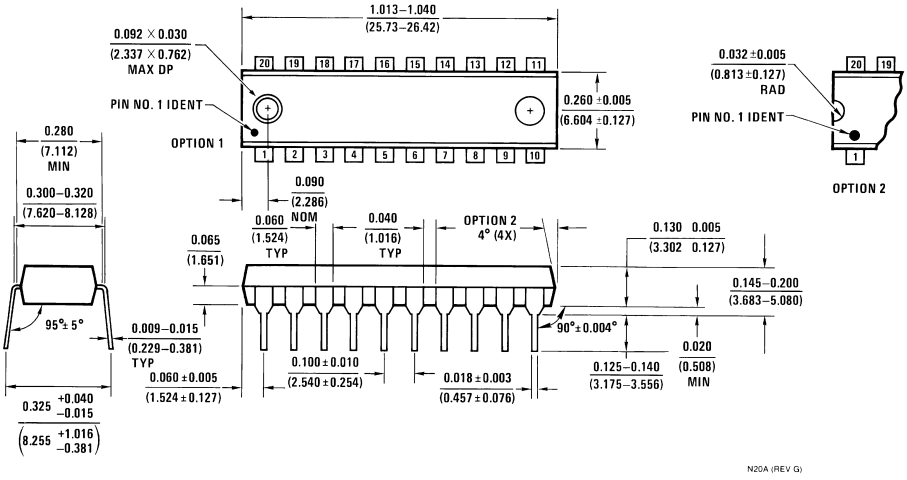
LOC	OBJ CODE		SOURCE STATEMENT	COMMENT
0038	E5		PUSH HL	; Save contents of all registers affected by
0039	C5		PUSH BC	; this subroutine.
003A	F5		PUSH AF	; Assumed INT mode 1 earlier set.
003B	21 00 3E		LD (HL),X3E00	; Initialize memory pointer where data will be stored.
003E	0E 01		LD C, X01	; C register will be port ADDR of A/D converters.
0040	D300		OUT X00, A	; Load peripheral status word into 8-bit latch.
0042	DB00		IN A, X00	; Load status word into accumulator.
0044	47		LD B,A	; Save the status word.
0045	79	TEST	LD A,C	; Test to see if the status of all A/D's have
0046	FE 08		CP, X08	; been checked. If so, exit subroutine
0048	CA 60 00		JPZ, DONE	
004B	78		LD A,B	; Test a single bit in status word by looking for
004C	1F		RRA	; a "1" to be rotated into the CARRY (an $\overline{INT}$
004D	47		LD B,A	; is loaded as a "1"). If CARRY is set then load
004E	DA 5500		JPC, LOAD	; contents of A/D at port ADDR in C register.
0051	0C	NEXT	INC C	; If CARRY is not set, increment C register to point
0052	C3 4500		JP,TEST	; to next A/D, then test next bit in status word.
0055	ED 78	LOAD	IN A, (C)	; Read data from interrupting A/D and invert
0057	EE FF		XOR FF	; the data.
0059	77		LD (HL),A	; Store the data
005A	2C		INC L	
005B	71		LD (HL),C	; Store A/D identifier (A/D port ADDR).
005C	2C		INC L	
005D	C3 51 00		JP,NEXT	; Test next bit in status word.
0060	F1	DONE	POP AF	; Re-establish all registers as they were
0061	C1		POP BC	; before the interrupt.
0062	E1		POP HL	
0063	C9		RET	; Return to original program

DS005671-A6

Physical Dimensions inches (millimeters) unless otherwise noted



SO Package (M)
Order Number ADC0802LCWM or ADC0804LCWM
NS Package Number M20B



Molded Dual-In-Line Package (N)
Order Number ADC0801LCN, ADC0802LCN,
ADC0803LCN, ADC0804LCN or ADC0805LCN
NS Package Number N20A

Notes

LIFE SUPPORT POLICY

NATIONAL'S PRODUCTS ARE NOT AUTHORIZED FOR USE AS CRITICAL COMPONENTS IN LIFE SUPPORT DEVICES OR SYSTEMS WITHOUT THE EXPRESS WRITTEN APPROVAL OF THE PRESIDENT AND GENERAL COUNSEL OF NATIONAL SEMICONDUCTOR CORPORATION. As used herein:

1. Life support devices or systems are devices or systems which, (a) are intended for surgical implant into the body, or (b) support or sustain life, and whose failure to perform when properly used in accordance with instructions for use provided in the labeling, can be reasonably expected to result in a significant injury to the user.
2. A critical component is any component of a life support device or system whose failure to perform can be reasonably expected to cause the failure of the life support device or system, or to affect its safety or effectiveness.

National Semiconductor Corporation
Americas
Tel: 1-800-272-9959
Fax: 1-800-737-7018
Email: support@nsc.com

www.national.com

National Semiconductor Europe
Fax: +49 (0) 1 80-530 85 86
Email: europe.support@nsc.com
Deutsch Tel: +49 (0) 1 80-530 85 85
English Tel: +49 (0) 1 80-532 78 32
Français Tel: +49 (0) 1 80-532 93 58
Italiano Tel: +49 (0) 1 80-534 16 80

National Semiconductor Asia Pacific Customer Response Group
Tel: 65-2544466
Fax: 65-2504466
Email: sea.support@nsc.com

National Semiconductor Japan Ltd.
Tel: 81-3-5639-7560
Fax: 81-3-5639-7507

CMOS 8-bit A/D converters — ADC0803/4-1

DESCRIPTION

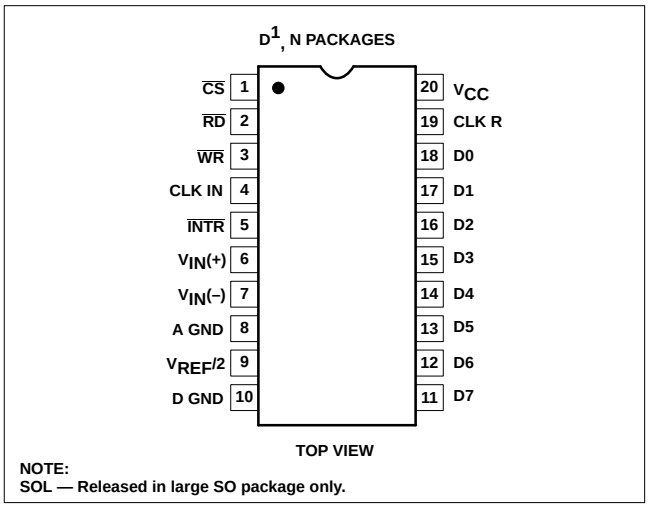
The ADC0803 family is a series of three CMOS 8-bit successive approximation A/D converters using a resistive ladder and capacitive array together with an auto-zero comparator. These converters are designed to operate with microprocessor-controlled buses using a minimum of external circuitry. The 3-State output data lines can be connected directly to the data bus.

The differential analog voltage input allows for increased common-mode rejection and provides a means to adjust the zero-scale offset. Additionally, the voltage reference input provides a means of encoding small analog voltages to the full 8 bits of resolution.

FEATURES

- Compatible with most microprocessors
- Differential inputs
- 3-State outputs
- Logic levels TTL and MOS compatible
- Can be used with internal or external clock
- Analog input range 0V to V_{CC}
- Single 5V supply
- Guaranteed specification with 1MHz clock

PIN CONFIGURATION



NOTE:
SOL — Released in large SO package only.

APPLICATIONS

- Transducer-to-microprocessor interface
- Digital thermometer
- Digitally-controlled thermostat
- Microprocessor-based monitoring and control systems

ORDERING INFORMATION

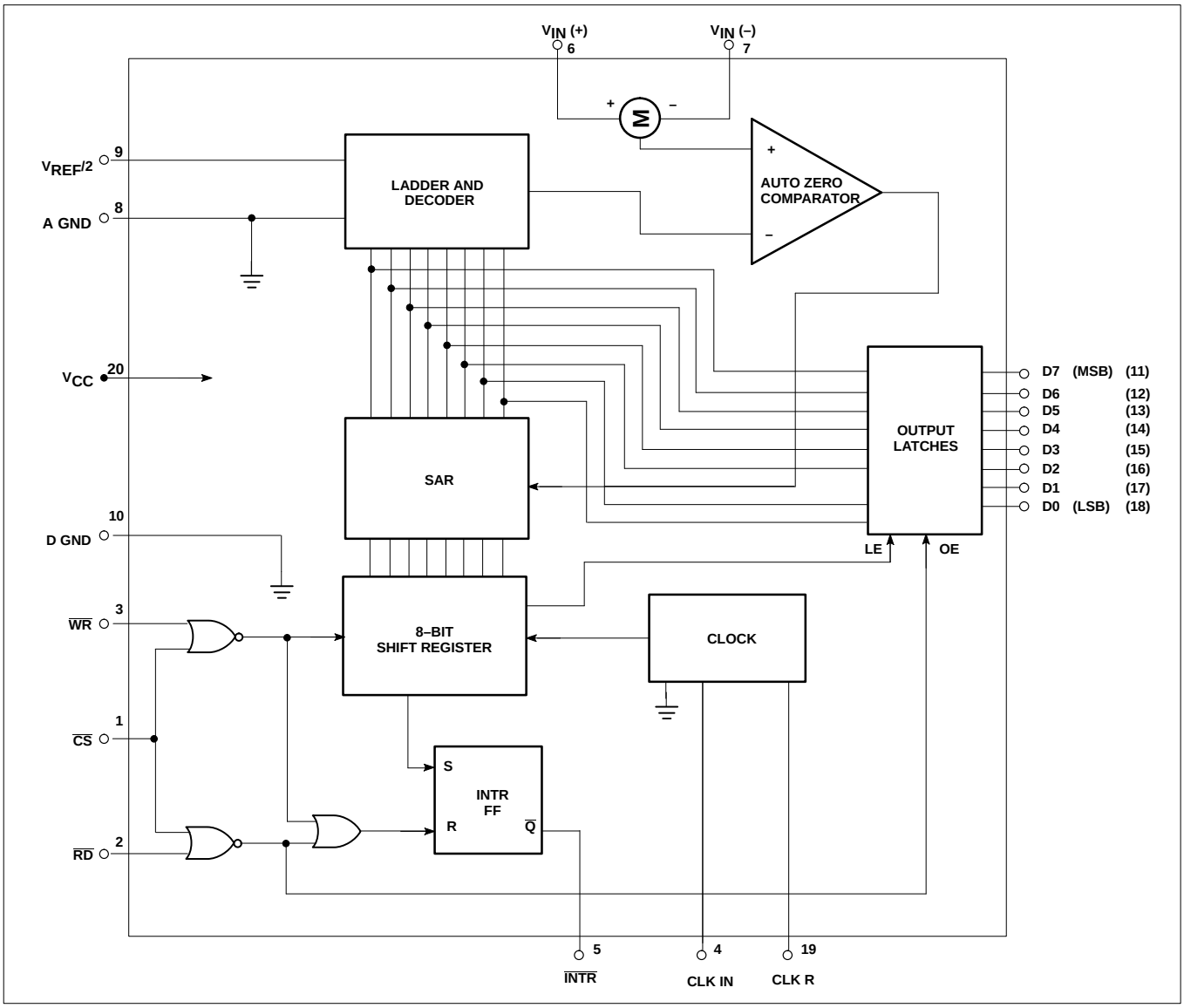
DESCRIPTION	TEMPERATURE RANGE	ORDER CODE	DWG #
20-Pin Plastic Dual In-Line Package (DIP)	-40 to +85°C	ADC0803/04-1 LCN	0408B
20-Pin Plastic Dual In-Line Package (DIP)	0 to 70°C	ADC0803/04-1 CN	0408B
20-Pin Plastic Small Outline (SO) Package	0 to 70°C	ADC0803/04-1 CD	1021B
20-Pin Plastic Small Outline (SO) Package	-40 to 85°C	ADC0803/04-1 LCD	1021B

ABSOLUTE MAXIMUM RATINGS

SYMBOL	PARAMETER	RATING	UNIT
V_{CC}	Supply voltage	6.5	V
	Logic control input voltages	-0.3 to +16	V
	All other input voltages	-0.3 to (V_{CC} +0.3)	V
T_A	Operating temperature range		
	ADC0803/04-1 LCD	-40 to +85	°C
	ADC0803/04-1 LCN	-40 to +85	°C
	ADC0803/04-1 CD	0 to +70	°C
	ADC0803/04-1 CN	0 to +70	°C
T_{STG}	Storage temperature	-65 to +150	°C
T_{SOLD}	Lead soldering temperature (10 seconds)	300	°C
P_D	Maximum power dissipation T_A=25°C (still air)[1]		
	N package	1690	mW
	D package	1390	mW

NOTES:
1. Derate above 25°C, at the following rates: N package at 13.5mW/°C; D package at 11.1mW/°C

BLOCK DIAGRAM



DC ELECTRICAL CHARACTERISTICS

V_{CC} = 5.0V, f_{CLK} = 1MHz, $T_{MIN} \le T_A \le T_{MAX}$, unless otherwise specified.

SYMBOL	PARAMETER	TEST CONDITIONS	ADC0803/4			UNIT
			Min	Typ	Max	
	ADC0803 relative accuracy error (adjusted)	Full-Scale adjusted			0.50	LSB
	ADC0804 relative accuracy error (unadjusted)	$V_{REF}/2 = 2.500V_{DC}$			1	LSB
R_{IN}	$V_{REF}/2$ input resistance[3]	V_{CC} = 0V[2]	400	680		Ω
	Analog input voltage range[3]		–0.05		V_{CC}+0.05	V
	DC common-mode error	Over analog input voltage range		1/16	1/8	LSB
	Power supply sensitivity	V_{CC} = 5V ±10%[1]		1/16		LSB
Control inputs						
V_{IH}	Logical “1” input voltage	$V_{CC} = 5.25V_{DC}$	2.0		15	V_{DC}
V_{IL}	Logical “0” input voltage	$V_{CC} = 4.75V_{DC}$			0.8	V_{DC}
I_{IH}	Logical “1” input current	$V_{IN} = 5V_{DC}$		0.005	1	μA_{DC}
I_{IL}	Logical “0” input current	$V_{IN} = 0V_{DC}$	–1	–0.005		μA_{DC}
Clock in and clock R						
V_T+	Clock in positive-going threshold voltage		2.7	3.1	3.5	V_{DC}
V_T–	Clock in negative-going threshold voltage		1.5	1.8	2.1	V_{DC}
V_H	Clock in hysteresis (V_T+)–(V_T–)		0.6	1.3	2.0	V_{DC}
V_{OL}	Logical “0” clock R output voltage	I_{OL} = 360µA, $V_{CC} = 4.75V_{DC}$			0.4	V_{DC}
V_{OH}	Logical “1” clock R output voltage	I_{OH} = –360µA, $V_{CC} = 4.75V_{DC}$	2.4			V_{DC}
Data output and $\overline{INTR}$						
V_{OL}	Logical “0” output voltage					
	Data outputs	I_{OL} = 1.6mA, $V_{CC} = 4.75V_{DC}$			0.4	V_{DC}
	$\overline{INTR}$ outputs	I_{OL} = 1.0mA, $V_{CC} = 4.75V_{DC}$			0.4	V_{DC}
V_{OH}	Logical “1” output voltage	I_{OH} = –360µA, $V_{CC} = 4.75V_{DC}$	2.4			V_{DC}
		I_{OH} = –10µA, $V_{CC} = 4.75V_{DC}$	4.5			
I_{OZL}	3-state output leakage	$V_{OUT} = 0V_{DC}$, $\overline{CS}$ = logical “1”	–3			μA_{DC}
I_{OZH}	3-state output leakage	$V_{OUT} = 5V_{DC}$, $\overline{CS}$ = logical “1”			3	μA_{DC}
I_{SC}	+Output short-circuit current	V_{OUT} = 0V, T_A = 25°C	4.5	12		mA_{DC}
I_{SC}	–Output short-circuit current	$V_{OUT} = V_{CC}$, T_A = 25°C	9.0	30		mA_{DC}
I_{CC}	Power supply current	f_{CLK} = 1MHz, $V_{REF}/2$ = OPEN, $\overline{CS}$ = Logical “1”, T_A = 25°C		3.0	3.5	mA

NOTES:
1. Analog inputs must remain within the range: $–0.05 \le V_{IN} \le V_{CC} + 0.05V$.
2. See typical performance characteristics for input resistance at V_{CC} = 5V.
3. $V_{REF}/2$ and V_{IN} must be applied after the V_{CC} has been turned on to prevent the possibility of latching.

AC ELECTRICAL CHARACTERISTICS

SYMBOL	PARAMETER	TO	FROM	TEST CONDITIONS	ADC0803/4			UNIT
					Min	Typ	Max	
	Conversion time			f_{CLK}=1MHz[1]	66		73	µs
f_{CLK}	Clock frequency[1]				0.1	1.0	3.0	MHz
	Clock duty cycle[1]				40		60	%
CR	Free-running conversion rate			$\overline{CS}$=0, f_{CLK}=1MHz $\overline{INTR}$ tied to $\overline{WR}$			13690	conv/s
$t_{W(\overline{WR})L}$	Start pulse width			$\overline{CS}$=0	30			ns
t_{ACC}	Access time	Output	$\overline{RD}$	$\overline{CS}$=0, C_L=100pF		75	100	ns
t_{1H}, t_{0H}	3-State control	Output	$\overline{RD}$	CL=10pF, R_L=10kΩ See 3-State test circuit		70	100	ns
t_{W1}, t_{R1}	$\overline{INTR}$ delay	$\overline{INTR}$	$\overline{WD}$ or $\overline{RD}$			100	150	ns
C_{IN}	Logic input=capacitance					5	7.5	pF
C_{OUT}	3-State output capacitance					5	7.5	pF

NOTES:
1. Accuracy is guaranteed at f_{CLK}=1MHz. Accuracy may degrade at higher clock frequencies.

FUNCTIONAL DESCRIPTION

These devices operate on the Successive Approximation principle. Analog switches are closed sequentially by successive approximation logic until the input to the auto-zero comparator [$V_{IN}(+)$-$V_{IN}(-)$] matches the voltage from the decoder. After all bits are tested and determined, the 8-bit binary code corresponding to the input voltage is transferred to an output latch. Conversion begins with the arrival of a pulse at the $\overline{WR}$ input if the CS input is low. On the High-to-Low transition of the signal at the WR or the CS input, the SAR is initialized, the shift register is reset, and the $\overline{INTR}$ output is set high. The A/D will remain in the reset state as long as the $\overline{CS}$ and $\overline{WR}$ inputs remain low. Conversion will start from one to eight clock periods after one or both of these inputs makes a Low-to-High transition. After the conversion is complete, the $\overline{INTR}$ pin will make a High-to-Low transition. This can be used to interrupt a processor, or otherwise signal the availability of a new conversion result. A read ($\overline{RD}$) operation (with $\overline{CS}$ low) will clear the $\overline{INTR}$ line and enable the output latches. The device may be run in the free-running mode as described later. A conversion in progress can be interrupted by issuing another start command.

Digital Control Inputs

The digital control inputs (CS, WR, $\overline{RD}$) are compatible with standard TTL logic voltage levels. The required signals at these inputs correspond to Chip Select, START Conversion, and Output Enable control signals, respectively. They are active-Low for easy interface to microprocessor and microcontroller control buses. For applications not using microprocessors, the $\overline{CS}$ input (Pin 1) can be grounded and the A/D START function is achieved by a negative-going pulse to the $\overline{WR}$ input (Pin 3). The Output Enable function is achieved by a logic low signal at the $\overline{RD}$ input (Pin 2), which may be grounded to constantly have the latest conversion present at the output.

ANALOG OPERATION

Analog Input Current

The analog comparisons are performed by a capacitive charge summing circuit. The input capacitor is switched between $V_{IN(+)}$4 and $V_{IN(-)}$, while reference capacitors are switched between taps on the reference voltage divider string. The net charge corresponds to the weighted difference between the input and the most recent total value set by the successive approximation register.

The internal switching action causes displacement currents to flow at the analog inputs. The voltage on the on-chip capacitance is switched through the analog differential input voltage, resulting in proportional currents entering the $V_{IN(+)}$ input and leaving the $V_{IN(-)}$ input. These transient currents occur at the leading edge of the internal clock pulses. They decay rapidly so do not inherently cause errors as the on-chip comparator is strobed at the end of the clock period.

Input Bypass Capacitors and Source Resistance

Bypass capacitors at the input will average the charges mentioned above, causing a DC and an AC current to flow through the output resistance of the analog signal sources. This charge pumping action is worse for continuous conversions with the $V_{IN(+)}$ input at full scale. This current can be a few microamps, so bypass capacitors should NOT be used at the analog inputs of the V_{REF}/2 input for high resistance sources (> 1kΩ). If input bypass capacitors are desired for noise filtering and a high source resistance is desired to minimize capacitor size, detrimental effects of the voltage drop across the input resistance can be eliminated by adjusting the full scale with both the input resistance and the input bypass capacitor in place. This is possible because the magnitude of the input current is a precise linear function of the differential voltage.

Large values of source resistance where an input bypass capacitor is not used will not cause errors as the input currents settle out prior to the comparison time. If a low pass filter is required in the system, use a low valued series resistor (< 1kΩ) for a passive RC section or add an op amp active filter (low pass). For applications with source resistances at or below 1kΩ, a 0.1µF bypass capacitor at the inputs will prevent pickup due to series lead inductance or a long wire. A 100Ω series resistor can be used to isolate this capacitor (both the resistor and capacitor should be placed out of the feedback loop) from the output of the op amp, if used.

Analog Differential Voltage Inputs and Common-Mode Rejection

These A/D converters have additional flexibility due to the analog differential voltage input. The $V_{IN(-)}$ input (Pin 7) can be used to subtract a fixed voltage from the input reading (tare correction). This is also useful in a 4/20mA current loop conversion. Common-mode noise can also be reduced by the use of the differential input.

The time interval between sampling $V_{IN(+)}$ and $V_{IN(-)}$ is 4.5 clock periods. The maximum error due to this time difference is given by:

$V(max)=(V_P)\ (2f_{CM})\ (4.5/f_{CLK})$,

where:

V=error voltage due to sampling delay

V_P=peak value of common-mode voltage

f_{CM}=common mode frequency

For example, with a 60Hz common-mode frequency, f_{cm}, and a 1MHz A/D clock, F_{CLK}, keeping this error to 1/4 LSB (about 5mV) would allow a common-mode voltage, V_P, which is given by:

$$V_P = \frac{[V(max)\ (f_{CLK})}{(2f_{CM})(4.5)}$$

or

$$V_P = \frac{(5 \times 10^{-3})\ (10^4)}{(6.28)\ (60)\ (4.5)} = 2.95V$$

The allowed range of analog input voltages usually places more severe restrictions on input common-mode voltage levels than this, however.

An analog input span less than the full 5V capability of the device, together with a relatively large zero offset, can be easily handled by use of the differential input. (See Reference Voltage Span Adjust).

Noise and Stray Pickup

The leads of the analog inputs (Pins 6 and 7) should be kept as short as possible to minimize input noise coupling and stray signal pick-up. Both EMI and undesired digital signal coupling to these inputs can cause system errors. The source resistance for these inputs should generally be below 5kΩ to help avoid undesired noise pickup. Input bypass capacitors at the analog inputs can create errors as described previously. Full scale adjustment with any input bypass capacitors in place will eliminate these errors.

Reference Voltage

For application flexibility, these A/D converters have been designed to accommodate fixed reference voltages of 5V to Pin 20 or 2.5V to Pin 9, or an adjusted reference voltage at Pin 9. The reference can be set by forcing it at $V_{REF}/2$ input, or can be determined by the supply voltage (Pin 20). Figure 1 indicates how this is accomplished.

Reference Voltage Span Adjust

Note that the Pin 9 ($V_{REF}/2$) voltage is either 1/2 the voltage applied to the V_{CC} supply pin, or is equal to the voltage which is externally forced at the $V_{REF}/2$ pin. In addition to allowing for flexible references and full span voltages, this also allows for a ratiometric voltage reference. The internal gain of the $V_{REF}/2$ input is 2, making the full-scale differential input voltage twice the voltage at Pin 9.

For example, a dynamic voltage range of the analog input voltage that extends from 0 to 4V gives a span of 4V (4-0), so the $V_{REF}/2$ voltage can be made equal to 2V (half of the 4V span) and full scale output would correspond to 4V at the input.

On the other hand, if the dynamic input voltage had a range of 0.5 to 3.5V, the span or dynamic input range is 3V (3.5-0.5). To encode this 3V span with 0.5V yielding a code of zero, the minimum expected input (0.5V, in this case) is applied to the $V_{IN}(-)$ pin to account for the offset, and the $V_{REF}/2$ pin is set to 1/2 the 3V span, or 1.5V. The A/D converter will now encode the $V_{IN}(+)$ signal between 0.5 and 3.5V with 0.5V at the input corresponding to a code of zero and 3.5V at the input producing a full scale output code. The full 8 bits of resolution are thus applied over this reduced input voltage range. The required connections are shown in Figure 2.

Operating Mode

These converters can be operated in two modes:

1) absolute mode
2) ratiometric mode

In absolute mode applications, both the initial accuracy and the temperature stability of the reference voltage are important factors in the accuracy of the conversion. For $V_{REF}/2$ voltages of 2.5V, initial errors of ±10mV will cause conversion errors of ±1 LSB due to the gain of 2 at the $V_{REF}/2$ input. In reduced span applications, the initial value and stability of the $V_{REF}/2$ input voltage become even more important as the same error is a larger percentage of the $V_{REF}/2$ nominal value. See Figure 3.

In ratiometric converter applications, the magnitude of the reference voltage is a factor in both the output of the source transducer and the output of the A/D converter, and, therefore, cancels out in the final digital code. See Figure 4.

Generally, the reference voltage will require an initial adjustment. Errors due to an improper reference voltage value appear as full-scale errors in the A/D transfer function.

ERRORS AND INPUT SPAN ADJUSTMENTS

There are many sources of error in any data converter, some of which can be adjusted out. Inherent errors, such as relative accuracy, cannot be eliminated, but such errors as full-scale and zero scale offset errors can be eliminated quite easily. See Figure 2.

Zero Scale Error

Zero scale error of an A/D is the difference of potential between the ideal 1/2 LSB value (9.8mV for $V_{REF}/2$=2.500V) and that input voltage which just causes an output transition from code 0000 0000 to a code of 0000 0001.

If the minimum input value is not ground potential, a zero offset can be made. The converter can be made to output a digital code of 0000 0000 for the minimum expected input voltage by biasing the $V_{IN}(-)$ input to that minimum value expected at the $V_{IN}(-)$ input to that minimum value expected at the $V_{IN}(+)$ input. This uses the

differential mode of the converter. Any offset adjustment should be done prior to full scale adjustment.

Full Scale Adjustment

Full scale gain is adjusted by applying any desired offset voltage to $V_{IN}(-)$, then applying the $V_{IN}(+)$ a voltage that is 1-1/2 LSB less than the desired analog full-scale voltage range and then adjusting the magnitude of $V_{REF}/2$ input voltage (or the V_{CC} supply if there is no $V_{REF}/2$ input connection) for a digital output code which just changes from 1111 1110 to 1111 1111. The ideal $V_{IN}(+)$ voltage for this full-scale adjustment is given by:

$$V_{IN}(\ \ ) = V_{IN}(-) - 1.5 \times \frac{V_{MAX} - V_{MIN}}{255}$$

where:

V_{MAX}=high end of analog input range (ground referenced)

V_{MIN}=low end (zero offset) of analog input (ground referenced)

CLOCKING OPTION

The clock signal for these A/Ds can be derived from external sources, such as a system clock, or self-clocking can be accomplished by adding an external resistor and capacitor, as shown in Figure 6.

Heavy capacitive or DC loading of the CLK R pin should be avoided as this will disturb normal converter operation. Loads less than 50pF are allowed. This permits driving up to seven A/D converter CLK IN pins of this family from a single CLK R pin of one converter. For larger loading of the clock line, a CMOS or low power TTL buffer or PNP input logic should be used to minimize the loading on the CLK R pin.

Restart During a Conversion

A conversion in process can be halted and a new conversion began by bringing the $\overline{CS}$ and $\overline{WR}$ inputs low and allowing at least one of them to go high again. The output data latch is not updated if the conversion in progress is not completed; the data from the previously completed conversion will remain in the output data latches until a subsequent conversion is completed.

Continuous Conversion

To provide continuous conversion of input data, the $\overline{CS}$ and $\overline{RD}$ inputs are grounded and $\overline{INTR}$ output is tied to the $\overline{WR}$ input. This $\overline{INTR}/\overline{WR}$ connection should be momentarily forced to a logic low upon power-up to insure circuit operation. See Figure 5 for one way to accomplish this.

DRIVING THE DATA BUS

This CMOS A/D converter, like MOS microprocessors and memories, will require a bus driver when the total capacitance of the data bus gets large. Other circuitry tied to the data bus will add to the total capacitive loading, even in the high impedance mode.

There are alternatives in handling this problem. The capacitive loading of the data bus slows down the response time, although DC specifications are still met. For systems with a relatively low CPU clock frequency, more time is available in which to establish proper logic levels on the bus, allowing higher capacitive loads to be driven (see Typical Performance Characteristics).

At higher CPU clock frequencies, time can be extended for I/O reads (and/or writes) by inserting wait states (8880) or using clock-extending circuits (6800, 8035).

Finally, if time is critical and capacitive loading is high, external bus drivers must be used. These can be 3-State buffers (low power Schottky is recommended, such as the N74LS240 series) or special higher current drive products designed as bus drivers. High current bipolar bus drivers with PNP inputs are recommended as the PNP input offers low loading of the A/D output, allowing better response time.

POWER SUPPLIES

Noise spikes on the V_{CC} line can cause conversion errors as the internal comparator will respond to them. A low inductance filter capacitor should be used close to the converter V_{CC} pin and values of 1µF or greater are recommended. A separate 5V regulator for the converter (and other 5V linear circuitry) will greatly reduce digital noise on the V_{CC} supply and the attendant problems.

WIRING AND LAYOUT PRECAUTIONS

Digital wire-wrap sockets and connections are not satisfactory for breadboarding this (or any) A/D converter. Sockets on PC boards can be used. All logic signal wires and leads should be grouped or kept as far as possible from the analog signal leads. Single wire analog input leads may pick up undesired hum and noise, requiring the use of shielded leads to the analog inputs in many applications.

A single-point analog ground separate from the logic or digital ground points should be used. The power supply bypass capacitor and the self-clocking capacitor, if used, should be returned to digital ground. Any $V_{REF}/2$ bypass capacitor, analog input filter capacitors, and any input shielding should be returned to the analog ground point. Proper grounding will minimize zero-scale errors which are present in every code. Zero-scale errors can usually be traced to improper board layout and wiring.

APPLICATIONS

Microprocessor Interfacing

This family of A/D converters was designed for easy microprocessor interfacing. These converters can be memory mapped with appropriate memory address decoding for $\overline{CS}$ (read) input. The active-Low write pulse from the processor is then connected to the $\overline{WR}$ input of the A/D converter, while the processor active-Low read pulse is fed to the converter $\overline{RD}$ input to read the converted data. If the clock signal is derived from the microprocessor system clock, the designer/programmer should be sure that there is no attempt to read the converter until 74 converter clock pulses after the start pulse goes high. Alternatively, the $\overline{INTR}$ pin may be used to interrupt the processor to cause reading of the converted data. Of course, the converter can be connected and addressed as a peripheral (in I/O space), as shown in Figure 7. A bus driver should be used as a buffer to the A/D output in large microprocessor systems where the data leaves the PC board and/or must drive capacitive loads in excess of 100pF. See Figure 9.

Interfacing the SCN8048 microcomputer family is pretty simple, as shown in Figure 8. Since the SCN8048 family has 24 I/O lines, one of these (shown here as bit 0 or port 1) can be used as the chip select signal to the converter, eliminating the need for an address

decoder. The $\overline{RD}$ and $\overline{WR}$ signals are generated by reading from and writing to a dummy address.

Digitizing a Transducer Interface Output

Circuit Description

Figure 10 shows an example of digitizing transducer interface output voltage. In this case, the transducer interface is the NE5521, an LVDT (Linear Variable Differential Transformer) Signal Conditioner. The diode at the A/D input is used to insure that the input to the A/D does not go excessively beyond the supply voltage of the A/D. See the NE5521 data sheet for a complete description of the operation of that part.

Circuit Adjustment

To adjust the full scale and zero scale of the A/D, determine the range of voltages that the transducer interface output will take on. Set the LVDT core for null and set the Zero Scale Scale Adjust Potentiometer for a digital output from the A/D of 1000 000. Set the LVDT core for maximum voltage from the interface and set the Full Scale Adjust potentiometer so the A/D output is just barely 1111 1111.

A Digital Thermostat

Circuit Description

The schematic of a Digital Thermostat is shown in Figure 11. The A/D digitizes the output of the LM35, a temperature transducer IC with an output of 10mV per °C. With $V_{REF}/2$ set for 2.56V, this 10mV corresponds to 1/2 LSB and the circuit resolution is 2°C. Reducing $V_{REF}/2$ to 1.28 yields a resolution of 1°C. Of course, the lower $V_{REF}/2$ is, the more sensitive the A/D will be to noise.

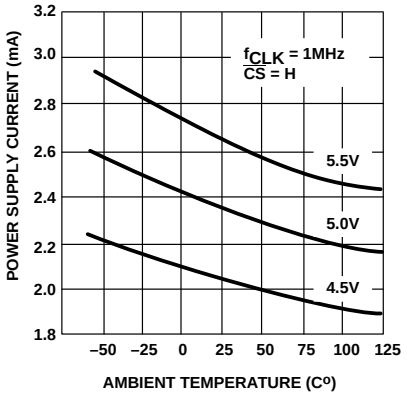
The desired temperature is set by holding either of the set buttons closed. The SCC80C451 programming could cause the desired (set) temperature to be displayed while either button is depressed and for a short time after it is released. At other times the ambient temperature could be displayed.

The set temperature is stored in an SCN8051 internal register. The A/D conversion is started by writing anything at all to the A/D with port pin P10 set high. The desired temperature is compared with the digitized actual temperature, and the heater is turned on or off by clearing setting port pin P12. If desired, another port pin could be used to turn on or off an air conditioner.

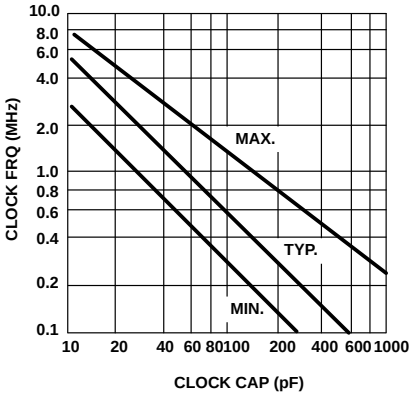
The display drivers are NE587s if common anode LED displays are used. Of course, it is possible to interface to LCD displays as well.

TYPICAL PERFORMANCE CHARACTERISTICS

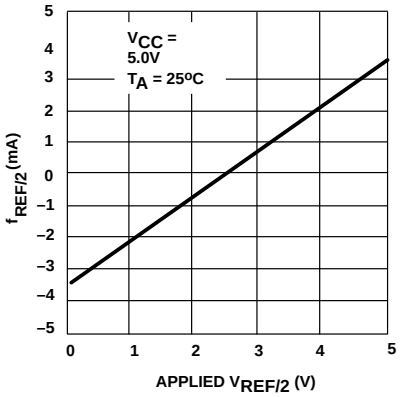
Power Supply Current vs Temperature



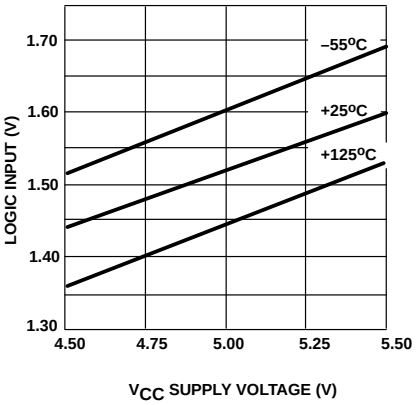
Clock Frequency vs Clock Capacitor



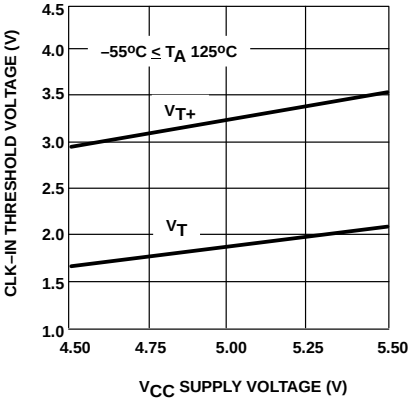
Input Current vs Applied Voltage at $V_{REF/2}$ Pin



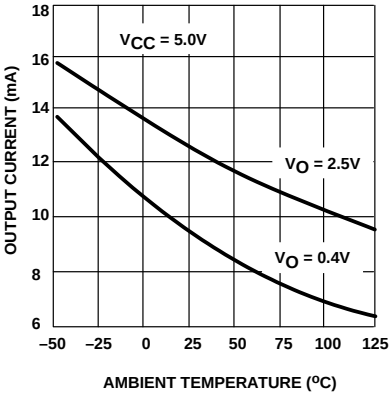
Logic Input Threshold Voltage vs Supply Voltage



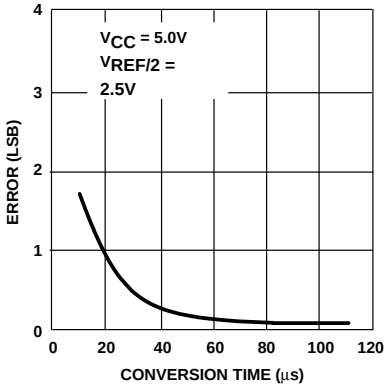
CLK–IN Threshold Voltage vs Supply Voltage



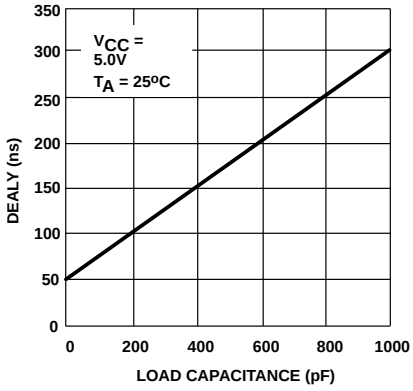
Output Current vs Temperature



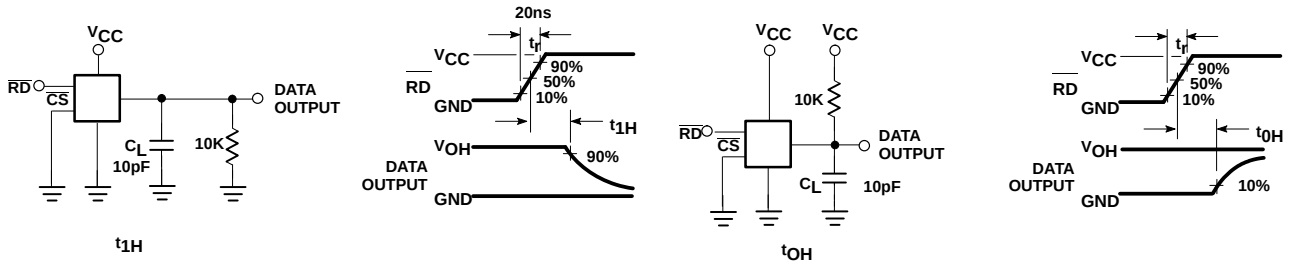
Full Scale Error vs Conversion Time



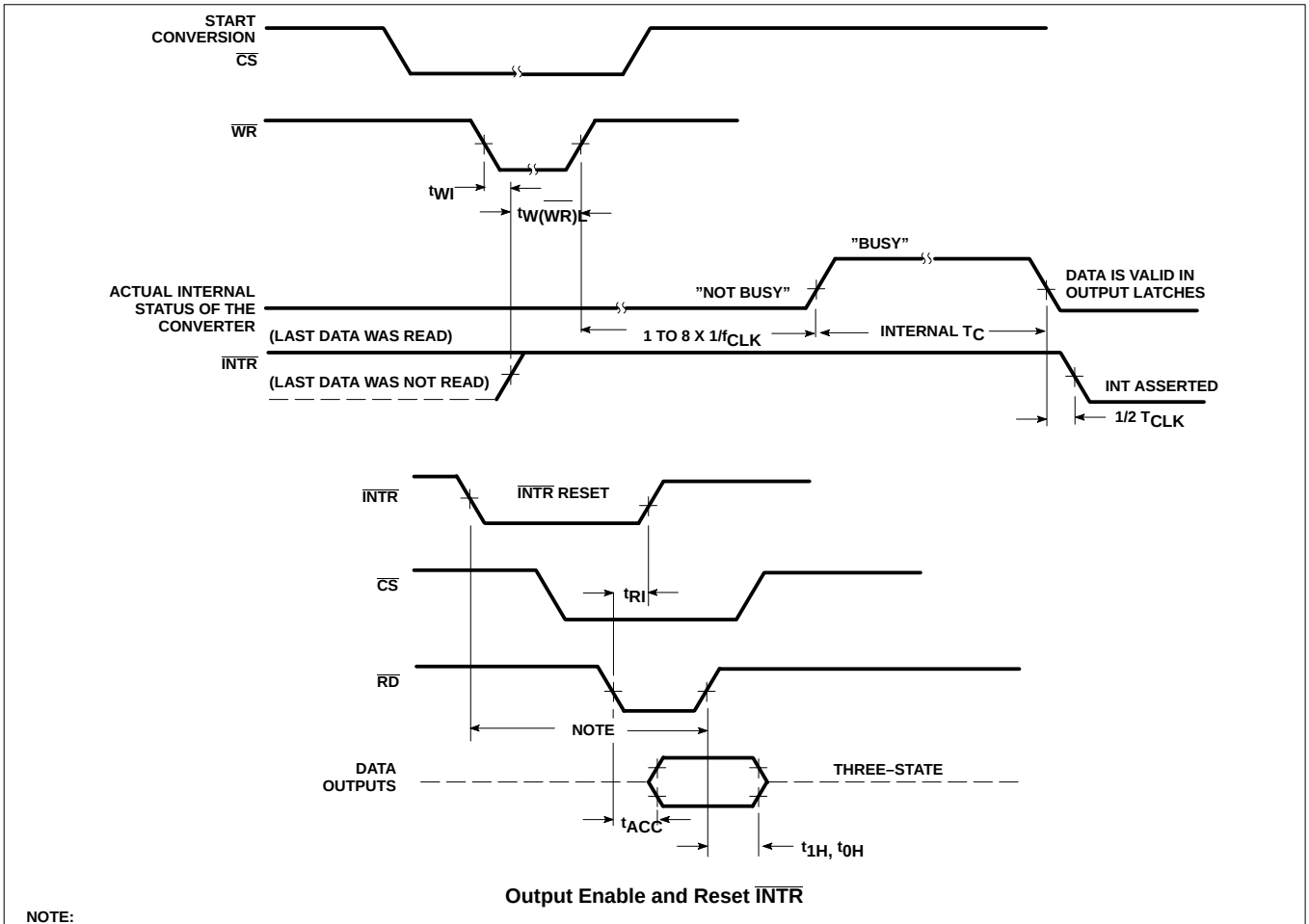
Delay From $\overline{RD}$ Falling Edge to Data Valid vs Load Capacitance



3-STATE TEST CIRCUITS AND WAVEFORMS (ADC0801-1)

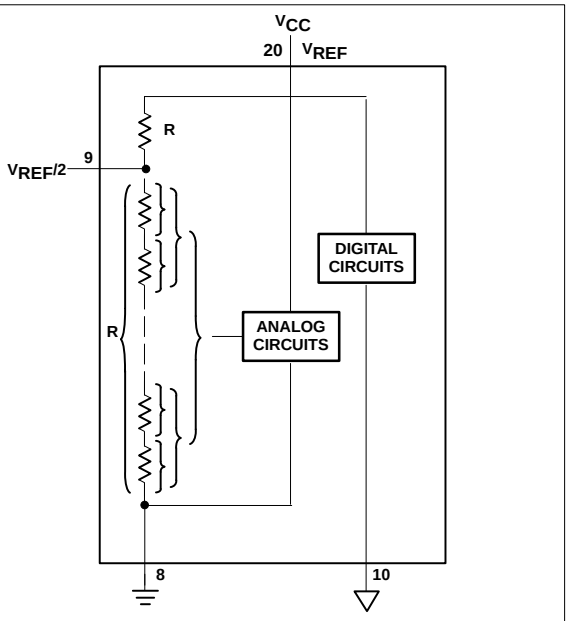


TIMING DIAGRAMS (All timing is measured from the 50% voltage points)



Output Enable and Reset $\overline{INTR}$

NOTE:
Read strobe must occur 8 clock periods ($8/f_{CLK}$) after assertion of interrupt to guarantee reset of $\overline{INTR}$.



NOTE:
The $V_{REF}/2$ voltage is either 1/2 the V_{CC} voltage or is that which is forced at Pin 9.

Figure 1. Internal Reference Design

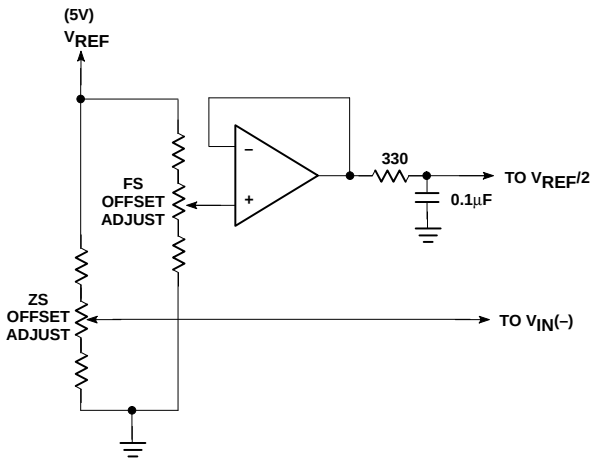
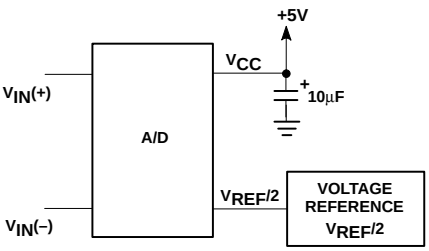
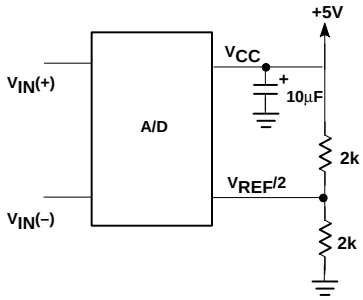


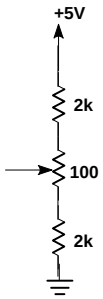
Figure 2. Offsetting the Zero Scale and Adjusting the Input Range (Span)



a. Fixed Reference



b. Fixed Reference Derived from V_{CC}



c. Optional Full Scale Adjustment

Figure 3. Absolute Mode of Operation

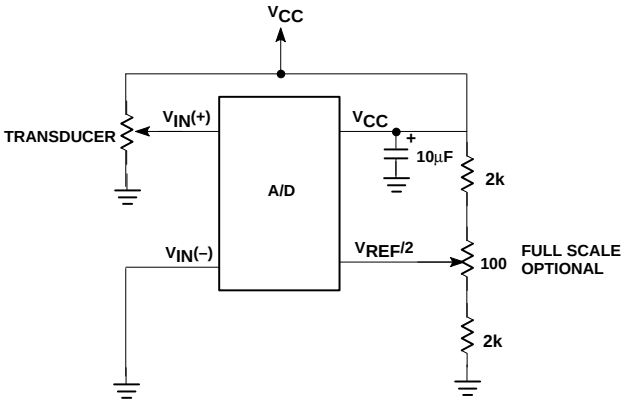


Figure 4. Ratiometric Mode of Operation with Optional Full Scale Adjustment

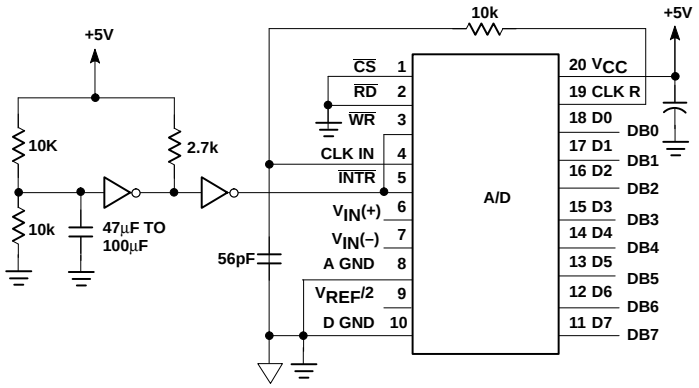


Figure 5. Connection for Continuous Conversion

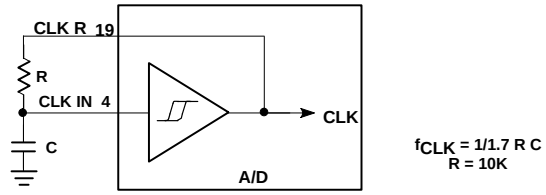


Figure 6. Self-Clocking the Converter

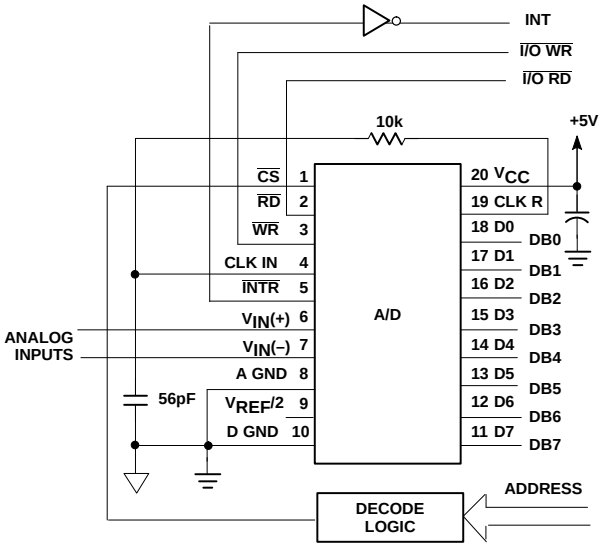


Figure 7. Interfacing to 8080A Microprocessor

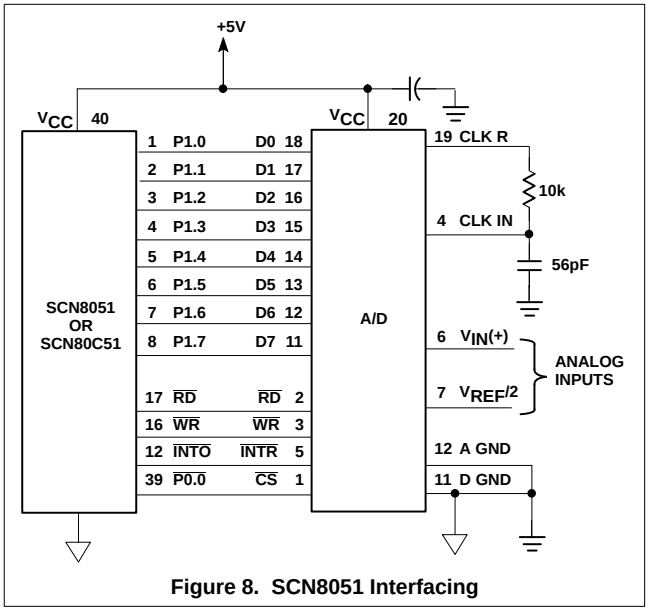


Figure 8. SCN8051 Interfacing

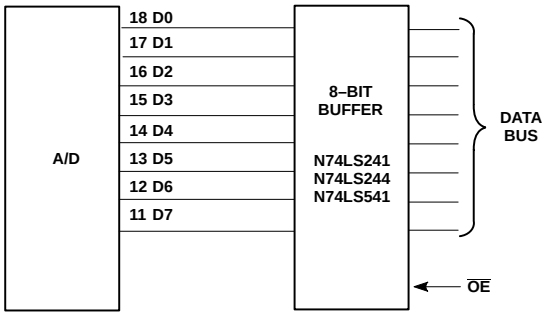


Figure 9. Buffering the A/D Output to Drive High Capacitance Loads and for Driving Off-Board Loads

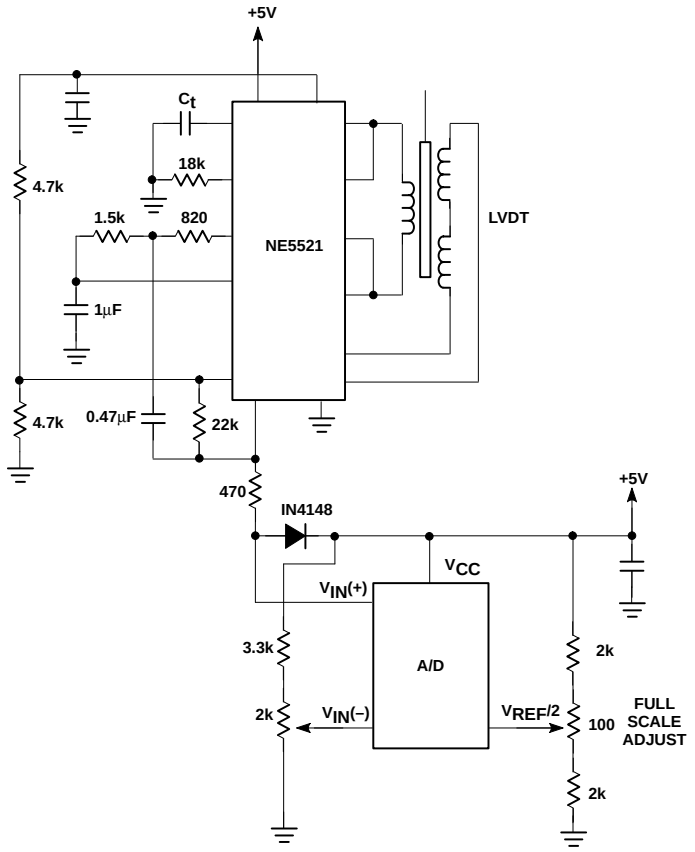


Figure 10. Digitizing a Transducer Interface Output

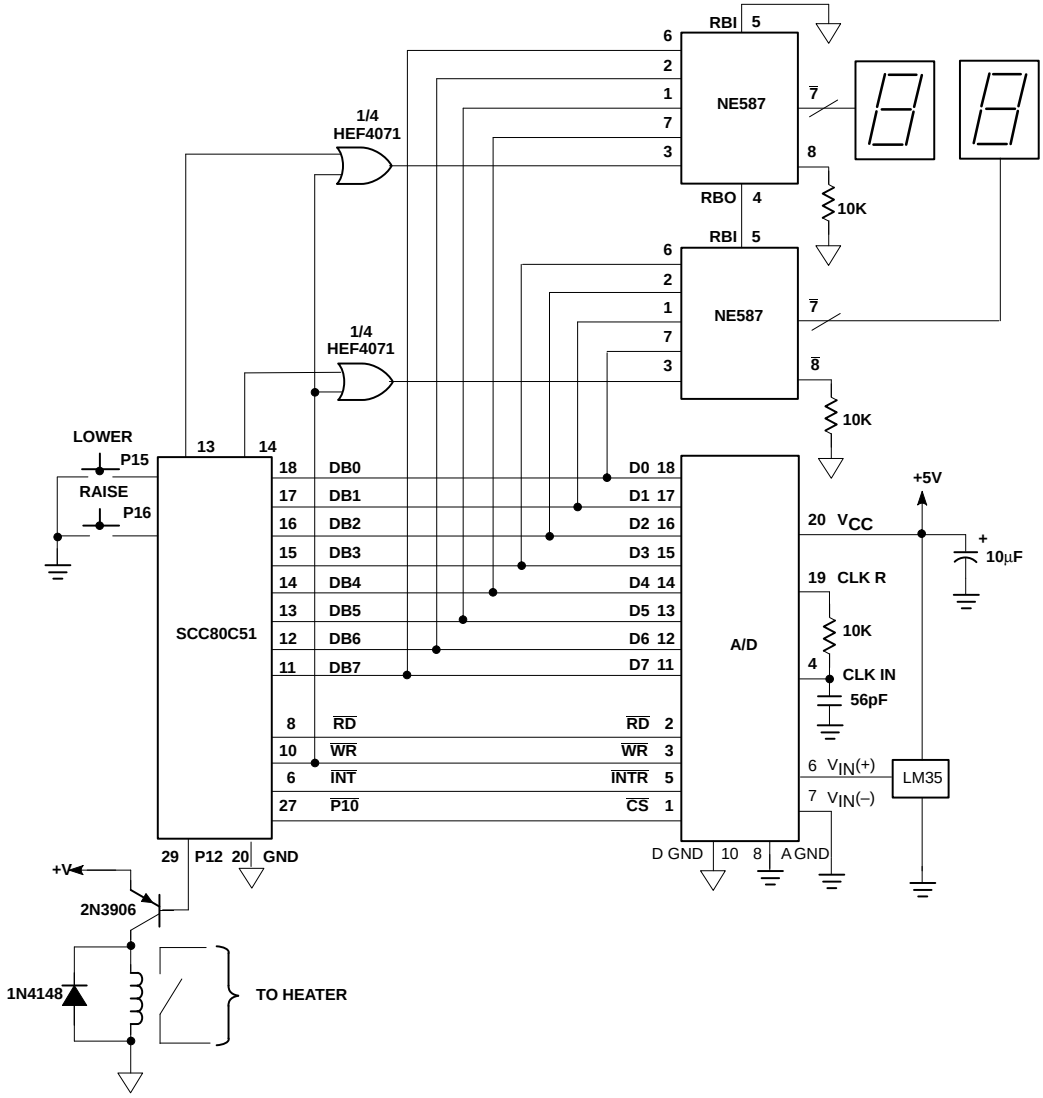


Figure 11. Digital Thermostat

AP-578

APPLICATION NOTE

Software and Hardware Considerations for FPU Exception Handlers for Intel Architecture Processors

February 1997

Order Number: 243291-002

CONTENTS

1.0 INTRODUCTION AND READING GUIDE

The primary purpose of this application note is to provide information to help software engineers write the most robust Floating-Point Unit (FPU) exception handlers possible. This note also provides the basic hardware information needed to design the MS-DOS* compatible interface[1] for the most recent generations of Intel Architecture processors, starting with the Intel486™ processor. (Because of the small amount of new design activity, the hardware interfaces for the 8086 through the Intel386™ processors are treated only briefly.) The third purpose is to provide a compendium of the history of the development and variations of the Intel Architecture Floating-Point Units (FPUs) as relevant to their exception handling. Following is a list of Intel Architecture processors and math coprocessors in chronological order.

- 8086 processor
- 8087 math coprocessor
- 80286 processor
- 80287 math coprocessor
- Intel386™ processor
- Intel387 math coprocessor
- Intel486™ DX processor (with integrated FPU)
- Intel486 SX processor
- Intel487 math coprocessor
- Pentium® processor (with integrated FPU)
- Pentium Pro processor (with integrated FPU)

Much of this material is in various sections of the *Pentium® Processor Family Developer's Manual,* Volume 3. There is also some material in this application note that is not published elsewhere. On the other hand, there is much additional material on the FPU from the *Pentium® Processor Family Developer's Manual,* Volume 3 which has not been reproduced here, including the details on each of its specific exceptions. Much of this will be useful in writing FPU exception handlers, so Volume 3 should be used as an essential reference along with this appliction note.

NOTE

The following manuals referenced in this document are archived and are available on Intel's web site at http://www.intel.com: *Pentium® Processor Family Developer's Manual, Volume 1: Pentium Processor* (Order Number 241428-004) and the *Pentium® Processor Family Developer's Manual, Volume 3: Architecture and Programming Manual* (Order Number 241430-004).

The materials are presented in a mostly chronological order, which supports the history preservation purpose, and also minimizes forward references. Thus the main body of this application note begins with Section 2 which covers the six presently available generations of Intel Architecture FPUs in chronological order starting with the 8087. The history of the FPU exception handling has been complicated both by Intel's successful efforts to improve the performance and flexibility of the FPU through the generations, and by the decision to support upward compatibility for a large customer base which was implementing FPU exception handling in a way compatible with the first 8088 Personal Computers (PCs) and major Operating Systems (OSs). This second complication has resulted in two different systems or modes for FPU exception handling starting with the 80286 and 80287.

Beginning with the 80286 and 80287, Intel provided a dedicated input pin (ERROR#) on the 80286, to be connected to the ERROR# output pin on the 80287, for the FPU exceptions. When asserted, the ERROR# input triggers interrupt 16. The use of this dedicated interrupt for the FPU exception handler is referred to as the "native mode", and is recommended by Intel. However, for reasons explained in Sections 2.1 and 2.2, the majority of the Intel Architecture (IA) customer base has not been using the native mode, but rather the "MS-DOS compatible mode" for FPU exception handling. Since the MS-DOS compatible mode has the largest customer base, is the more complicated mode, and has changed the most between generations, it is the main focus of Section 2. In addition to the history of the

Footnotes

[1] WINDOWS* 95 and WINDOWS 3.1 (and earlier versions) use almost the same interface as MS-DOS*, and the recommendations herein for an MS-DOS compatable system apply to all three operating systems.

architecture and interfaces for FPU exception handling, Section 2 provides the basic hardware information needed to design the MS-DOS compatible interface for the most recent generations of IA processors, and discusses in detail several important system implications.

Section 3 describes the recommended protocol for writing MS-DOS compatible FPU exception handlers, with various options, along with discussions of several problems and how to avoid them. Most of the material is also applicable to native mode handlers.

Although the native mode of FPU exception handling was available from the second generation of the six presently available generations of the Intel Architecture FPUs (and brief discussions of it are provided in Section 2), we give the main presentation of it last, in Section 4. This is more chronologically consistent than it would seem because it has not become widely used until recently.

A software engineer who needs to write an MS-DOS compatible FPU exception handler but does not want to review the FPU history (or read any more about hardware than necessary) may skip Section 2 and begin reading Section 3. Then some subsections of Section 2 should be read as needed when referenced in Section 3. Someone writing a native mode exception handler that wants to read only what's necessary should start with Section 4, but then should also read Section 3, as most of the recommended protocol for FPU exception handling is the same for MS-DOS compatible and native modes and is not repeated in Section 4. Studying Section 4 first will allow this reader more easily to skip references back into Section 2 which are not relevant to the native mode.

A note on TERMINOLOGY: There are many variations of the words which are used to label an (unmasked) FPU error condition, and also the code which handles it. "Error", "exception" and "fault" are used to refer to the condition. Such a condition results in an interrupt, if no mask or block is in effect along the interrupt pathway. The code which handles the interrupt can be referred to as an error or exception or fault handler, or an interrupt or exception service routine, etc. The phrase "**exception handler**" has been used consistently (as much as possible) in this application note, for several reasons: "Exception" is less general than interrupt (which includes external hardware interrupts and software interrupts, as well as the processor problem conditions called exceptions or faults), but correctly **more** general than error or fault (because e.g. a precision exception caused by the fact that the number 1/3 cannot be exactly represented in the 80 bit FPU format is not really due to any mistake or error!). However, the reader should be aware that a number of the variations given above can be found in the literature, and that when applied to the FPU, they all mean the same thing.

2.0 MS-DOS* COMPATIBLE HANDLERS AND THEIR ISSUES OVER GENERATIONS

2.1 Origin of MS-DOS* Mode: 8088 and 8087

The 8087 has an output pin, INT, which it asserts when an unmasked exception occurs. There is no dedicated pin or interrupt vector number in the 8088 or 8086 specific for an FPU error assertion. Intel recommended that the FPU INT be routed to the 8088 or 8086 INTR pin through an 8259A Programmable Interrupt Controller (PIC), and not to the NMI input. However, the original PC design attached INT to NMI anyway, because by the time the 8087 was available, the original PC had already assigned other functions to the 8 inputs of the single PIC used in that design.

2.2 Development of MS-DOS* Mode with 80286 and 80287; Intel386ô Processor and Intel387 Math Coprocessor

The 80286 and 80287 and Intel386 processor and Intel387 math coprocessor pairs are each provided with ERROR# pins that are recommended to be connected between the processor and FPU. If this is done, when an unmasked FPU exception occurs, the FPU records the exception, and asserts its ERROR# pin. The processor recognizes this active condition of the ERROR# status line at the next WAIT or ESC instruction in its instruction stream, and branches to the FPU exception handler at interrupt vector 16. This is the native mode.

However, it was important to maintain maximum compatibility with the already significant 8088 and 8086 PC software base, where the NMI vector (#2) was used for FPU exceptions and vector 16 was

used for the BIOS video software interrupt. So the original IBM PC-AT* design for the 80286 and 80287 maintained Vector #16 for the BIOS video, and vector 2 was shared between the FPU exception and the new parity checking feature. A parity error detected by external hardware directly triggered vector 2 through the NMI pin. The FPU exception was handled by tying the 80286 RROR# input permanently high, and the 80287 ERROR# output was tied to the IRQ13 interrupt input on the second (cascaded) PIC in the PC-AT design. The PIC was programmed to issue vector 75H when IRQ13 was triggered.[2] But to maintain compatibility with older PC software that expected to access its own FPU exception handler by changing vector 2, the BIOS routine activated by INT 75H branches to INT 2. The standard INT 2 routine tests to see if the signal is due to the NMI pin (in which case it branches to the Parity Error handler) or an FPU exception.

2.2.1 SPECIAL HARDWARE FOR THE 80287 INTERFACE

It is necessary to guarantee, in the case of an 80287 exception, that the exception will be handled through the external loop using IRQ13 in the cascaded PIC before other 80287 instructions are sent over from the 80286. This is done by asserting BUSY# to the 80286, which normally means that the 80287 is still busy with a previous instruction, and so blocks the 80286 from sending another until BUSY# is de-asserted. This additional use of BUSY# is implemented by an edge triggered flip-flop which latches BUSY# using ERROR# from the 80287 as a clock. The output of this latch is OR'ed with the BUSY# output of the 80287 and drives the BUSY# input of the 80286. This PC-AT scheme effectively delays

deactivation of BUSY# at the 80286 whenever an 80287 ERROR# is signaled.

Since the BUSY# signal to the 80286 remains active after an exception, the IRQ13 interrupt (exception) handler (accessed through interrupt vector 75H) is guaranteed to execute before any other 80287 instruction can begin (except for some special control instructions).The IRQ13 handler clears the BUSY# latch (by writing to a special I/O port defined at 0F0H), thus allowing execution of 80287 instructions to proceed. The handler then branches to the NMI handler (interrupt vector 2), where the user defined numeric exception handler resides in PC compatible systems. Thus the PC-AT scheme approximates the exception reporting scheme between the 8087 and 8088 in the original PC.

2.2.2 SPECIAL HARDWARE FOR THE INTEL387™ MATH COPROCESSOR INTERFACE

The Intel386 processor can use a PC-AT compatible interface to communicate with an Intel387 math coprocessor, that is similar to the one in the 80286 and 80287 system above. As with the 80286, the Intel386 processor ERROR# pin should be tied permanently inactive (high), and the Intel387 ERROR# output used both to drive IRQ13, and to latch BUSY# in a flip-flop. The IRQ13 handler (vector 75H) should clear the BUSY# latch and branch to the NMI handler, as in the 80286 case.

However, an additional hardware feature is needed to manage the PEREQ signal to the Intel386 processor. After the Intel387 math coprocessor asserts ERROR#, and then its BUSY# signal has gone inactive, external hardware must re-assert the PEREQ signal to the Intel386 processor. This is needed for store instructions (for example, FST *mem*) because the Intel387 math coprocessor drops PEREQ once it

Footnotes

[2] WINDOWS 95 and WINDOWS 3.1 (and earlier versions) use interrupt 5DH instead of 75H, but the recommendations herein apply to systems using these WINDOWS operating systems, as well as MS-DOS.

signals an exception. While the Intel386 processor has not yet recognized the occurrence of the exception, it still expects the data transfers to complete via PEREQ re-activation. It is permissible for the Intel386 processor to receive undefined data during such I/O read cycles. Disabling the Intel387 math coprocessor is not necessary, because the dummy data transfer cycles directed to the Intel387 math co processor when PEREQ is externally reactivated for the Intel386 processor will not disturb the operation of the Intel387math coprocessor. The IRQ13 interrupt handler should remove the extension of BUSY# and also the re-activation of PEREQ via a write to PC/AT compatible hardware at I/O port 0F0H.

An Intel387 math coprocessoroffers significant performance improvements over the 80287, but because the Intel386 processor was ready for production before the Intel387math coprocessor, the Intel386 processor was designed to work with either the 80287 or Intel387 math coprcoessors. The Intel386 processor automatically configures itself for the attached FPU on reset by testing the ERROR# pin, and setting or clearing bit 4 in CR0 (see Section 10.1.3 in the *Pentium® Processor Family Developer's Manual,* Volume 3). This bit is the ET (Extension Type) bit, and it will be set if ERROR# is low (meaning an Intel387 is attached) and cleared if ERROR# is high (meaning there is an 80287 or no FPU attached). The MS-DOS compatible hardware interface is similar to that for the Intel386 processor and Intel387 math coprocessor combination.

2.3 FERR# & IGNNE# with Intel486™ and Pentium® Processors with CR0.NE=0

In the Intel486 and Pentium® processors, more enhancements and speedup features have been added to the corresponding FPUs. Also, the FPU is built into the same chip as the processor, which allows further increases in speed. MS-DOS compatibility for exception handling has also been built in, with the NE bit in control register CR0 selecting the MS-DOS compatible mode if made zero. (NE=1 selects the native or internal mode, which generates Interrupt 16, which is the same as the native version of exception handling for the 80286 and 80287 and the Intel386 processors and Intel 387 math coprocessor.)

In MS-DOS compatible mode, the FERR# (Floating-point ERRor) output replaces the ERROR# signal from the previous generations, and is connected to a PIC. A new input signal, IGNNE# (**IGN**ore **N**umeric **E**rror), is provided to allow the FPU exception handler to execute FPU instructions, if desired, without first clearing the error condition and without triggering the interrupt a second time. This IGNNE# feature is needed to replicate the capability that was provided on MS-DOS compatible Intel 80286 and 80287 and the Intel386 processors and INtel 387 math coprocessor-based systems by turning off the BUSY# signal, when inside the FPU exception handler, before clearing the error condition.

Note that Intel, in order to provide Intel486 processors for market segments which had no need for an FPU, created the "SX" versions. These Intel486 SX processors did not contain the floating-point unit. Intel also produced Intel487 SX math coprocessors for end users who later decided to upgrade to a system with an FPU. These Intel487 SX math coprocessors are similar to standard Intel486 processors with a working FPU on board. Thus the external circuitry necessary to support the MS-DOS compatible mode for Intel487 SX math coprocessors is the same as for standard Intel486 DX processors.

Note that the special DP (Dual Processing) mode for Pentium processors, and also the more general Intel MultiProcessor Specification for systems with multiple Pentium or Pentium Pro processors, support FPU exception handling only in the native mode. Intel does not recommend using the MS-DOS compatible FPU mode for systems using more than one processor.

2.3.1 BASIC RULES: WHEN FERR# IS GENERATED

- Assume the following conditions: NE=0, the IGNNE# input is de-asserted, and then an FPU instruction causes an unmasked FPU exception. Then in most cases, deferred error reporting occurs. This means that the processor does not respond immediately, but rather freezes just before executing the *next* WAIT or FPU instruction (except for "No-Wait" instructions, which the FPU executes regardless of an error condition).
- At the same time that the processor freezes, it also asserts the FERR# output.

- The frozen processor waits for an external interrupt, which must be supplied by external hardware in response to the FERR# assertion.
- In MS-DOS compatible systems, FERR# is fed to the IRQ13 input in the cascaded PIC, which generates interrupt 75H, which then branches to interrupt 2, as described above for the 80286and 80287 and Intel386 processor and Intel387 processor-based systems.

These cases in which FERR# is not asserted at the time of the error, but rather at the next FPU or WAIT instruction, include all exceptions caused by the basic arithmetic instructions (including FADD, FSUB, FMUL, FDIV, FSQRT, FCOM and FUCOM), precision exceptions caused by all types of FPU instructions, and numeric underflow and overflow on all types of FPU instructions except stores to memory. We will refer to these cases as deferred (error reporting).

On the other hand, there are some exceptions, which when caused by some instructions, drive FERR# at the time that the exception occurs. These include FPU stack fault, invalid operation and denormal exceptions caused by all transcendental instructions, FSCALE, FXTRACT, FPREM and others, and all exceptions (except precision) when caused by FPU store instructions. These cases are called immediate (error reporting). (These cases will, like the deferred, cause the processor to freeze just before executing the next WAIT or FPU instruction if the error condition has not been cleared by that time.) Note that in general, whether an FPU exception case is deferred or immediate depends both on which exception occurred, and which instruction caused that exception. A complete specification of these cases, which applies also to the Intel486, is given in Section 5.1.21 in the *Pentium® Processor Family Developer's Manual,* Volume 1.

If NE=0 but the IGNNE# input is active while an unmasked FPU exception is in effect, the processor disregards the exception, does not assert FERR#, and continues. If IGNNE# is then de-asserted and the FPU exception has not been cleared, the processor will respond as described

above. (That is, an immediate exception case will assert FERR# immediately. A deferred exception case will assert FERR# and freeze just before the next FPU or WAIT instruction.) The assertion of IGNNE# is intended for use only inside the FPU exception handler, where it is needed if one wants to execute non-control FPU instructions for diagnosis, before clearing the exception condition. When IGNNE# is asserted inside the exception

handler, a preceding FPU exception has already caused FERR# to be asserted, and the external interrupt hardware has responded, but IGNNE# assertion still prevents the freeze at FPU instructions. Note that if IGNNE# is left active outside of the FPU exception handler, additional FPU instructions may be executed after a given instruction has caused an FPU exception. In this case, if the FPU exception handler ever did get invoked, it could not determine which instruction caused the exception.

To properly manage the interface between the processor's FERR# output, its IGNNE# input, and the IRQ13 input of the PIC, additional external hardware is needed. A recommended configuration is described below.

2.3.2 RECOMMENDED EXTERNAL HARDWARE TO SUPPORT MS-DOS* COMPATIBILITY

Figure 1 below provides an external circuit which will assure proper handling of FERR# and IGNNE# when an FPU exception occurs. In particular, it assures that IGNNE# will be active only inside the FPU exception handler without depending on the order of actions by the exception handler. Some hardware implementations have been less robust because they have depended on the exception handler to clear the FPU exception interrupt request to the PIC (FP_IRQ signal) *before* the handler causes FERR# to be de-asserted by clearing the exception from the FPU itself. Figure 2 below shows the details of how IGNNE# will behave when the circuit in Figure 1 is implemented. The temporal regions within the FPU exception handler activity are described as follows:

1. The FERR# signal is activated by an FPU exception and sends an interrupt request through the PIC to the processor's INTR pin.
2. During the FPU interrupt service routine (exception handler) the processor will need to clear the interrupt request latch (Flip Flop #1). It may also want to execute non-control FPU instructions before the exception is cleared

from the FPU. For this purpose the IGNNE# must be driven low. Typically in the PC environment an I/O access to Port 0F0H clears the external FPU exception interrupt request (FP_IRQ). In the recommended circuit, this access also is used to activate IGNNE#. With IGNNE# active the FPU exception handler may execute any FPU instruction without being blocked by an active FPU exception.

3. Clearing the exception within the FPU will cause the FERR# signal to be deactivated and then there is no further need for IGNNE# to be active. In the recommended circuit, the deactivation of FERR# is used to deactivate IGNNE#. If another circuit is used, the software and circuit together must assure that IGNNE# is deactivated no later than the exit from the FPU exception handler.
4. In the circuit in Figure 1 when the FPU exception handler accesses I/O port 0F0H it clears the IRQ13 interrupt request output from Flip Flop #1 and also clocks out the IGNNE# signal (active) from Flip Flop #2. So the handler can activate IGNNE#, if needed, by doing this 0F0H access before clearing

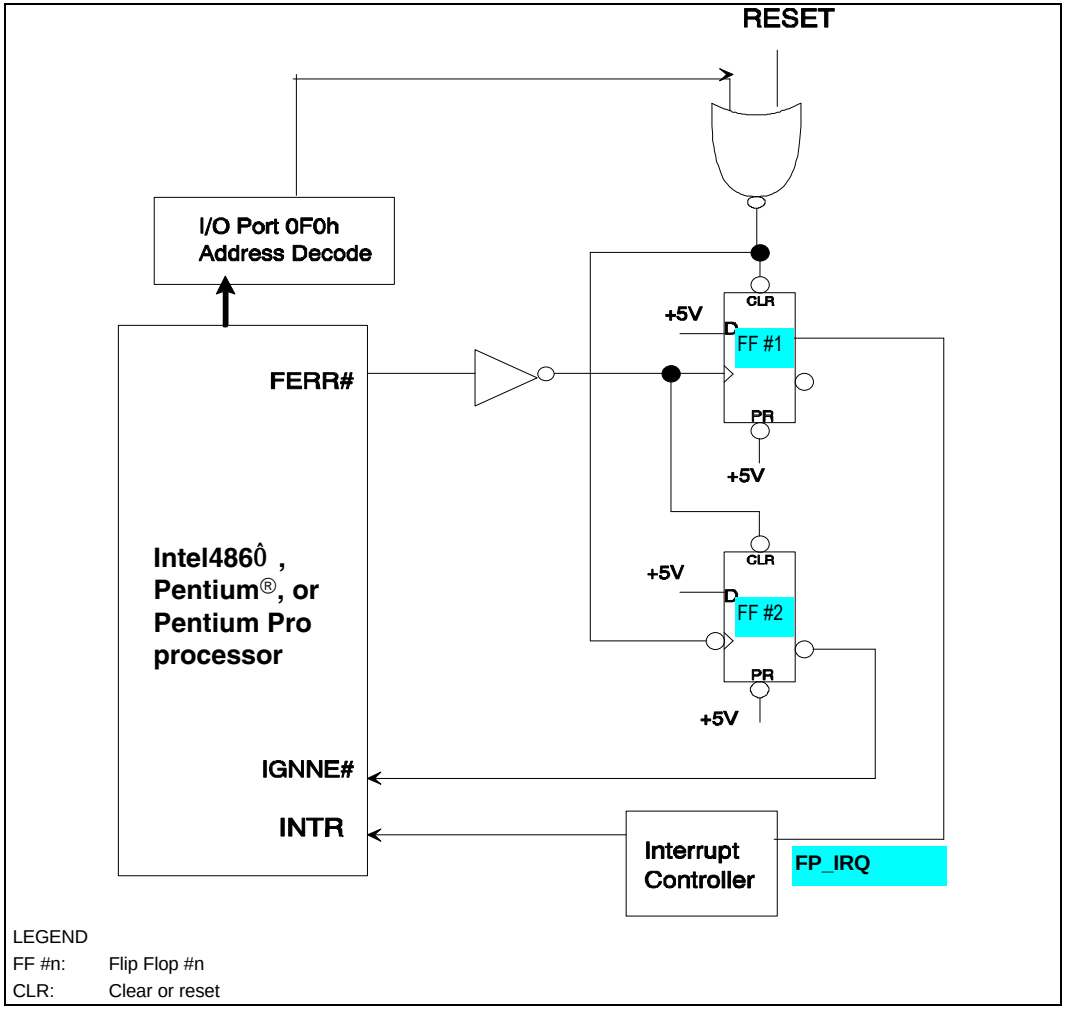


Figure 1. Recommended Circuit for MS-DOS* Compatible FPU Exception Handling

the FPU exception condition (which de-asserts FERR#). However, the circuit does not depend on the order of actions by the FPU exception handler to guarantee the correct hardware state upon exit from the handler. The flip flop which drives IGNNE# to the processor has its CLEAR input attached to the inverted FERR#. This ensures that IGNNE# can never be active when FERR# is inactive. So if the handler clears the FPU exception condition *before* the 0F0H access, IGNNE# does not get activated and left on after exit from the handler.

2.3.3 "NO-WAIT" FPU INSTRUCTIONS CAN GET FPU INTERRUPT IN WINDOW

The Pentium and the Intel486 processors implement the "No-Wait" Floating-Point instructions (FNINIT, FNCLEX, FNSTENV, FNSAVE, FNSTSW, FNSTCW, FNENI, FNDISI or FNSETPM - See Section 6.3.7 in the *Pentium® Processor Family Developer's Manual*, Volume 3) in the MS-DOS Compatibility mode (CR0.NE = 0) in the following manner:

If an unmasked numeric exception is pending from a preceding FPU instruction, a member of the “No-Wait” class of instructions will, at the beginning of its execution, assert the FERR# pin in response to that exception just like other FPU instructions, but then, unlike the other FPU instructions, FERR# will be de-asserted. This de-assertion was implemented to allow the “No-Wait” class of instructions to proceed without an interrupt due to any pending numeric exception. However, the brief assertion of FERR# is sufficient to latch the FPU exception request into most hardware interface implementations (including Intel’s recommended circuit).

All the FPU instructions are implemented such that during their execution, there is a window in which the processor will sample and accept external interrupts. If there is a pending interrupt, the processor services the interrupt first before resuming the execution of the instruction. Consequently, it is possible that the “No-Wait” Floating-Point instruction may accept the external interrupt caused by it’s own assertion of the FERR# pin in the event of a pending unmasked numeric exception, which is not an explicitly documented behavior of a “No-Wait” instruction. This process is illustrated by Figure 3, which is followed by a detailed description of the several cases possible.

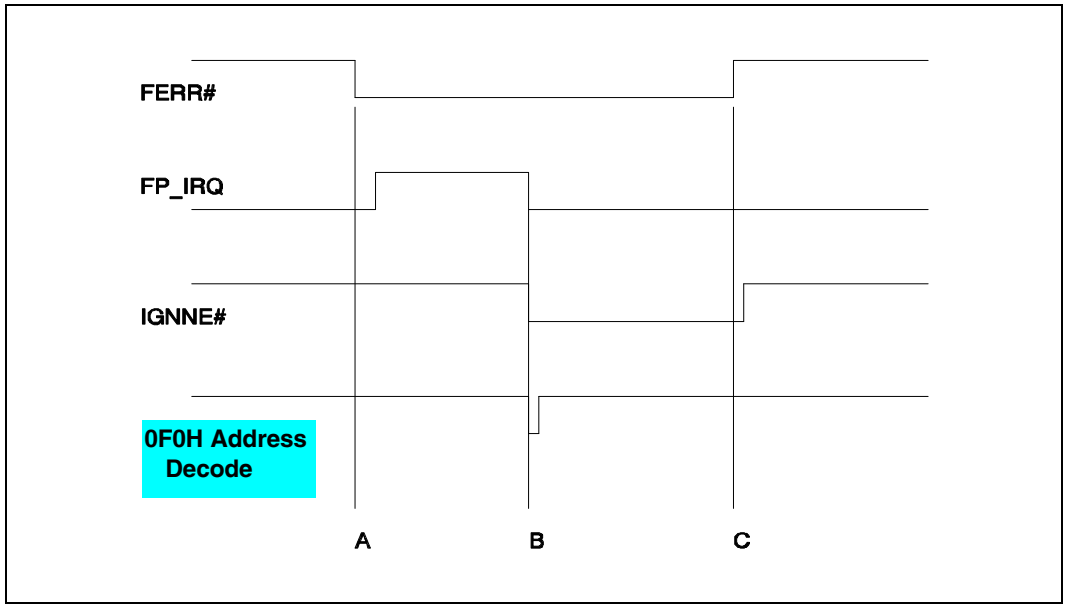


Figure 2. Behavior of Signals During FPU Exception Handling

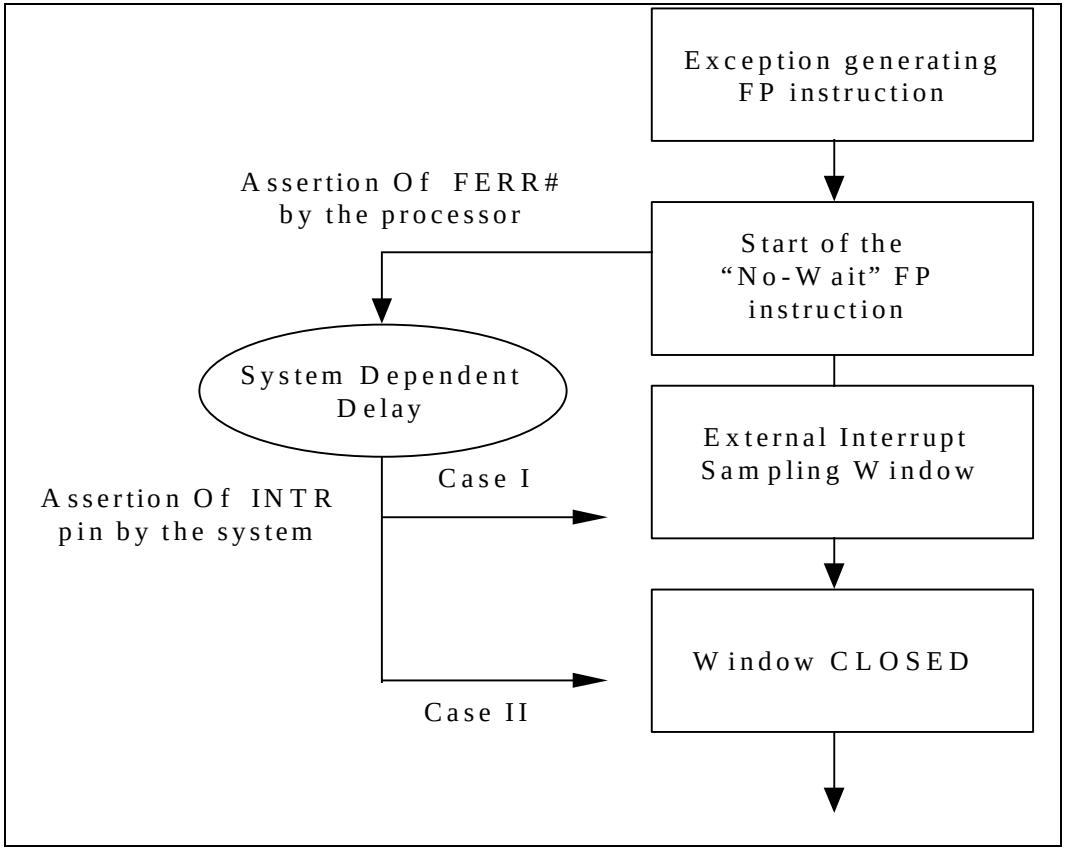


Figure 3: Timing of Receipt of External Interrupt

Figure 3 assumes that a floating-point instruction which generates a "deferred" error (as defined above in the Section 2.3.1), which asserts the FERR# pin only on encountering the next floating-point instruction, causes an unmasked numeric exception. Assume that the next floating-point instruction following this instruction is one of the "No-Wait" floating-point instructions. The FERR# pin is asserted by the processor to indicate the pending exception on encountering the "No-Wait" floating-point instruction. After the assertion of the FERR# pin the "No-Wait" floating-point instruction opens a window where the pending external interrupts are sampled.

Then there are two cases possible depending on the timing of the receipt of the interrupt via the INTR pin (asserted by the system in response to the FERR# pin) by the processor.

Case 1
If the system responds to the assertion of FERR# pin by the "No-Wait" floating-point instruction via the INTR pin during this window then the interrupt is serviced first, before resuming the execution of the "No-Wait" floating-point instruction.

Case 2
If the system responds via the INTR pin after the window has closed then the interrupt is recognized only at the next instruction boundary.

There are two other ways, in addition to Case I above, in which a "No-Wait" floating-point instruction can service a numeric exception inside its interrupt window. First, the first floating-point error condition could be of the "immediate" category (as defined in Section 2.3.1) that assert FERR#

immediately. If the system delay before asserting INTR is long enough, relative to the time elapsed before the “No-Wait” floating-point instruction, INTR can be asserted inside the interrupt window for the latter. Second, consider two “No-Wait” FPU instructions in close sequence, and assume that a previous FPU instruction has caused an unmasked numeric exception. Then if the INTR timing is too long for an FERR# signal triggered by the first “No-Wait” instruction to hit the first instruction’s interrupt window, it could catch the interrupt window of the second.

The possible malfunction of a “No-Wait” FPU instruction explained above cannot happen if the instruction is being used in the manner for which Intel originally designed it. The “No-Wait instructions were intended to be used inside the FPU exception handler, to allow manipulation of the FPU before the error condition is cleared, without hanging the processor because of the FPU error condition, and without the need to assert IGNNE#. They will perform this function correctly, since before the error condition is cleared, the assertion of FERR# that caused the FPU error handler to be invoked is still active. Thus the logic that would assert FERR# briefly at a “No-Wait” instruction causes no change since FERR# is already asserted. The “No-Wait” instructions may also be used without problem in the handler after the error condition is cleared, since now they will not cause FERR# to be asserted at all.

If a “No-Wait” instruction is used outside of the FPU exception handler, it may malfunction as explained above, depending on the details of the hardware interface implementation and which particular processor is involved. The actual interrupt inside the window in the “No-Wait” instruction may be blocked by surrounding it with the instructions: PUSHFD, CLI, “No-Wait”, then POPFD. (CLI blocks interrupts, and the push and pop of flags preserves and restores the original value of the interrupt flag.) However, if FERR# was triggered by the “No-Wait”, its latched value and the PIC response will still be in effect. Further code can be used to check for and correct such a condition, if needed. Section 3.6 (Considerations When FPU Shared Between Tasks) discusses an important example of this type of problem and gives a solution.

2.4 Pentium® Pro Processor with CR0.NE=0

When bit NE=0 in CR0, the MS-DOS* compatible mode of the Pentium Pro processor provides FERR# and IGNNE# functionality that is almost identical to the Intel486 and Pentium processors. The same external hardware, as described in Section 2.3.2 above, is recommended for the Pentium Pro processor as well as the two previous generations. The only change to MS-DOS compatible FPU exception handling with the Pentium Pro processor is that all exceptions for all FPU instructions cause immediate error reporting. That is, FERR# is asserted as soon as the FPU detects an unmasked exception; there are no cases in which error reporting is deferred to the next FPU or WAIT instruction. (As is discussed in Section 2.3.1, most exception cases in the Intel486 and Pentium processors are of the deferred type.)

Although FERR# is asserted immediately upon detection of an unmasked FPU error, this certainly does not mean that therequested interrupt will always be serviced before the next instruction in the code sequence is executed. To begin with, the Pentium Pro processor executes several instructions simultaneously. There also will be a delay, which depends on the external hardware implementation, between the FERR# assertion from the processor and the responding INTR assertion to the processor. Further, the interrupt request to the PICs (IRQ13) may be temporarily blocked by the OS, or delayed by higher priority interrupts, and processor response to INTR itself is blocked if the OS has cleared the IF bit in EFLAGS.

However, just as with the Intel486 and Pentium processors, if the IGNNE# input is inactive, a floating-point exception which occurred in the previous FPU instruction and is unmasked causes the processor to freeze immediately when encountering the next WAIT or FPU instruction (except for “No-Wait” instructions). This means that if the FPU exception handler has not already been invoked due to the earlier exception (and therefore the handler has not cleared that exception state from the FPU), the processor is forced to wait for the handler to be invoked and handle the exception, before the processor can execute another WAIT or FPU instruction.

As explained in Section 2.3.3, if a “No-Wait” instruction is used outside of the FPU exception handler, in the Intel486 and Pentium processors, it may accept an unmasked exception from a previous FPU instruction which happens to fall within the external interrupt sampling window that is opened near the beginning of execution of all FPU instructions. This will not happen in the Pentium Pro processor, because this sampling window has been removed from the “No-Wait” group of FPU instructions.

3.0 RECOMMENDED PROTOCOL FOR MS-DOS® AND WINDOWS* 95 COMPATIBLE HANDLERS[3]

The activities of numeric programs can be split into two major areas: program control and arithmetic. The program control part performs activities such as deciding what functions to perform, calculating addresses of numeric operands, and loop control. The arithmetic part simply adds, subtracts, multiplies, and performs other operations on the numeric operands. The processor is designed to handle these two parts separately and efficiently. An FPU exception handler, if a system chooses to implement one, is often one of the most complicated parts of the program control code.

3.1 Numeric Exceptions and their Defaults

The FPU can recognize six classes of numeric exception conditions while executing numeric instructions:

1. #I — Invalid operation
   #IS—Stack fault
   #IA—IEEE standard invalid operation
2. #Z—Divide-by-zero
3. #D—Denormalized operand
4. #O—Numeric overflow
5. #U—Numeric underflow
6. #P—Inexact result (precision)

For complete details on these exceptions and their defaults, see the *Pentium® Processor Family Developer’s Manual,* Volume 3, Sections 7.1.7 through 7.1.13.

3.1.1 TWO OPTIONS FOR HANDLING NUMERIC EXCEPTIONS

Depending on options determined by the software system designer, the processor takes one of two possible courses of action when a numeric exception occurs:

1. The FPU can handle selected exceptions itself, producing a default fix-up that is reasonable in most situations. This allows the numeric program execution to continue undisturbed. Programs can mask individual exception types to indicate that the FPU should generate this safe, reasonable result whenever the exception occurs. The default exception fix-up activity is treated by the FPU as part of the instruction causing the exception; no external indication of the exception is given (except that the instruction takes longer to execute when it handles a masked exception.) When masked exceptions are detected, a flag is set in the numeric status register, but no information is preserved regarding where or when it was set.

Footnotes

[3] Although there are some differences in the way FPU exceptions are handled between MS-DOS, and WINDOWS 95 and WINDOWS 3.1 (and earlier versions), the WINDOWS operating systems operate the processor in the MS-DOS compatable mode, and the recommended protocol given here applies to all these systems. On the other hand, current versions of WINDOWS NT use the FPU native mode.

2. Alternatively, a software exception handler can be invoked to handle the exception. When a numeric exception is unmasked and the exception occurs, the FPU stops further execution of the numeric instruction and causes a branch to a software exception handler. The exception handler can then implement any sort of recovery procedures desired for any numeric exception detectable by the FPU.

3.1.2 AUTOMATIC EXCEPTION HANDLING: USING MASKED EXCEPTIONS

Each of the six exception conditions described above has a corresponding flag bit in the FPU status word and a mask bit in the FPU control word. If an exception is masked (the corresponding mask bit in the control word = 1), the processor takes an appropriate default action and continues with the computation. The processor has a default fix-up activity for every possible exception condition it may encounter. These masked-exception responses are designed to be safe and are generally acceptable for most numeric applications.

For example, if the Inexact result (Precision) exception is masked, the system can specify whether the FPU should handle a result that cannot be represented exactly by one of four modes of rounding: rounding it normally, chopping it toward zero, always rounding it up, or always down. If the Underflow exception is masked, the FPU will store a number that is too small to be represented in normalized form as a denormal (or zero if it's smaller than the smallest denormal). Note that when exceptions are masked, the FPU may detect multiple exceptions in a single instruction, because it continues executing the instruction after performing its masked response. For example, the FPU could detect a denormalized operand, perform its masked response to this exception, and then detect an underflow.

As an example of how even severe exceptions can be handled safely and automatically using the default exception responses, consider a calculation of the parallel resistance of several values using only the standard formula (Figure 4). If R1 becomes zero, the circuit resistance becomes zero. With the divide-by-zero and precision exceptions masked, the processor will produce the correct result. FDIV of R1 into 1 gives infinity, and then FDIV of (infinity +R2 +R3) into 1 gives zero.

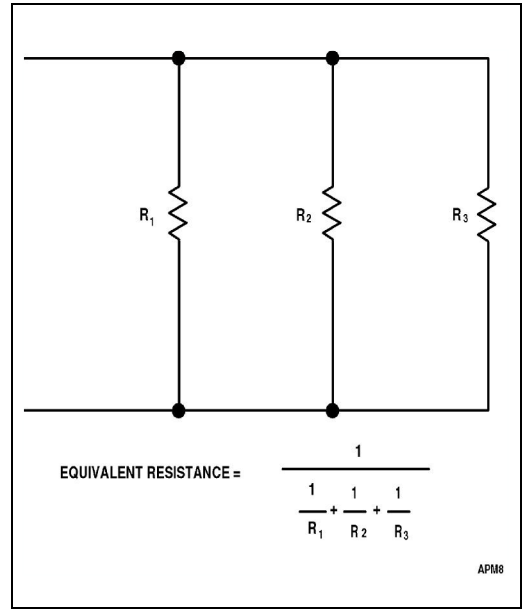


Figure 4. Arithmetic Example Using Infinity

By masking or unmasking specific numeric exceptions in the FPU control word, programmers can delegate responsibility for most exceptions to the processor, reserving the most severe exceptions for programmed exception handlers. Exception-handling software is often difficult to write, and the masked responses have been tailored to deliver the most reasonable result for each condition. For the majority of applications, masking all exceptions yields satisfactory results with the least programming effort. Certain exceptions can usefully be left unmasked during the debugging phase of software development, and then masked when the clean software is actually run. An invalid-operation exception for example, typically indicates a program error that must be corrected.

The exception flags in the FPU status word provide a cumulative record of exceptions that have occurred since these flags were last cleared. Once set, these flags can be cleared only by executing the FCLEX/FNCLEX (clear exceptions) instruction, by reinitializing the FPU with FINIT/FNINIT or FSAVE/FNSAVE, or by overwriting the flags with an FRSTOR or FLDENV instruction. This allows a programmer to mask all exceptions, run a calculation, and then inspect the status word to see

if any exceptions were detected at any point in the calculation.

3.2 Software Exception Handling

If the FPU in or with an Intel family processor (80286 and onwards) encounters an unmasked exception condition, with the system operated in the MS-DOS compatible mode and with IGNNE# not asserted, a software exception handler is invoked through a PIC and the processor's INTR pin. The FERR# (or ERROR#) output from the FPU that begins the process of invoking the exception handler may occur when the error condition is first detected, or when the processor encounters the next WAIT or FPU instruction. Which of these two cases occurs depends on the processor generation and also on which exception and which FPU instruction triggered it, as discussed earlier in Section 2. The elapsed time between the initial error signal and the invocation of the FPU exception handler depends of course on the external hardware interface, and also on whether the external interrupt for FPU errors is enabled. But the architecture ensures that the handler will be invoked before execution of the next WAIT or floating-point instruction since an unmasked floating-point exception causes the processor to freeze just before executing such an instruction (unless the IGNNE# input is active, or it is a "No-Wait" FPU instruction).

The frozen processor waits for an external interrupt, which must be supplied by external hardware in response to the FERR# (or ERROR#) output of the processor (or coprocessor), usually through IRQ13 on the "slave" PIC, and then through INTR. Then the external interrupt invokes the exception handling routine. Note that if the external interrupt for FPU errors is disabled when the processor executes an FPU instruction, the processor will freeze until some other (enabled) interrupt occurs if an unmasked FPU exception condition is in effect. If NE = 0 but the IGNNE# input is active, the processor disregards the exception and continues. Error reporting via an external interrupt is supported for MS-DOS compatibility. Chapter 23 of the *Pentium® Processor Family Developer's Manual,* Volume 3 contains further discussion of compatibility issues.

The references above to the ERROR# output from the FPU apply to the Intel387 and 80287 math coprocessors (NPX chips). If one of these coprocessors encounters an unmasked exception condition, it signals the exception to the 80286 or Intel386 processor using the ERROR# status line between the processor and the coprocessor. See Section 2.2 above, and Chapter 23 of the *Pentium® Processor Family Developer's Manual,* Volume 3 for differences in FPU exception handling.

The exception-handling routine is normally a part of the systems software. The routine must clear (or disable) the active exception flags in the FPU status word before executing any FP instructions that cannot complete execution when there is a pending FP exception. Otherwise, the FP instruction will trigger the FPU interrupt again, and the system will be caught in an endless loop of nested FP exceptions, and hang. In any event, the routine must clear (or disable) the active exception flags in the FPU status word after handling them, and before IRET(D). Typical exception responses may include:

- Incrementing an exception counter for later display or printing
- Printing or displaying diagnostic information (e.g., the FPU environment and registers)
- Aborting further execution, or using the exception pointers to build an instruction that will run without exception and executing it

Applications programmers should consult their operating system's reference manuals for the appropriate system response to numerical exceptions. For systems programmers, some details on writing software exception handlers are provided in Chapter 14 of the *Pentium® Processor Family Developer's Manual,* Volume 3*,* as well as in this application note.

As discussed in Section 2.3.2, some early FERR# to INTR hardware interface implementations are less robust than the recommended circuit. This is because they depended on the exception handler to clear the FPU exception interrupt request to the PIC (by accessing port 0F0H) *before* the handler causes FERR# to be de-asserted by clearing the exception from the FPU itself. To eliminate the chance of a

problem with this early hardware, Intel recommends that FPU exception handlers always access port 0F0H before clearing the error condition from the FPU.

3.3 Synchronization Required for Use of FPU Exception Handlers

Concurrency or synchronization management requires a check for exceptions before letting the processor change a value just used by the FPU. It is important to remember that almost any numeric instruction can, under the wrong circumstances, produce a numeric exception.

3.3.1 EXCEPTION SYNCHRONIZATION: WHAT, WHY AND WHEN

Exception synchronization means that the exception handler inspects and deals with the exception in the context in which it occurred. If concurrent execution is allowed, the state of the processor when it recognizes the exception is often *not* in the context in which it occurred. The processor may have changed many of its internal registers and be executing a totally different program by the time the exception occurs. If the exception handler cannot recapture the original context, it cannot reliably determine the cause of the exception or to recover successfully from the exception. To handle this situation, the FPU has special registers updated at the start of each numeric instruction to describe the state of the numeric program when the failed instruction was attempted. This provides tools to help the exception handler recapture the original context, but the application code must also be written with synchronization in mind. Overall, exception synchronization must ensure that the FPU and other relevant parts of the context are in a well defined state when the handler is invoked after an unmasked numeric exception occurs.

When the FPU signals an unmasked exception condition, it is requesting help. The fact that the exception was unmasked indicates that further numeric program execution under the arithmetic and programming rules of the FPU will probably yield invalid results. Thus the exception must be handled, and with proper synchronization, or the program will not operate reliably.

For programmers in higher-level languages, all required synchronization is automatically provided by the appropriate compiler. However, for assembly language programmers exception synchronization remains the responsibility of the programmer. It is not uncommon for a programmer to expect that their numeric program will not cause numeric exceptions after it has been tested and debugged, but in a different system or numeric environment, exceptions may occur regularly nonetheless. An obvious example would be use of the program with some numbers beyond the range for which it was designed and tested. The example in Section 3.3.2 shows a more subtle way in which unexpected exceptions can occur.

As described in Section 3.1.1, depending on options determined by the software system designer, the processor can perform one of two possible courses of action when a numeric exception occurs.

- The FPU can provide a default fix-up for selected numeric exceptions. If the FPU performs its default action for all exceptions, then the need for exception synchronization is not manifest. However, code is often ported to contexts and operating systems for which it was not originally designed. The example below illustrates that it is safest to always consider exception synchronization when designing code that uses the FPU.
- Alternatively, a software exception handler can be invoked to handle the exception. When a numeric exception is unmasked and the exception occurs, the FPU stops further execution of the numeric instruction and causes a branch to a software exception handler. When an FPU exception handler will be invoked, synchronization must always be considered to assure reliable performance.

The following examples illustrate the need to always consider exception synchronization when writing numeric code, even when the code is initially intended for execution with exceptions masked.

3.3.2 EXCEPTION SYNCHRONIZATION: EXAMPLES

In the following examples, three instructions are shown to load an integer, calculate its square root, then increment the integer. The synchronous execution of the FPU will allow both of these

Incorrect Error Synchronization:

```
FILD     COUNT        ; FPU instruction
INC      COUNT        ; integer instruction alters operand
FSQRT                 ; subsequent FPU instruction -- error
         ; from previous FPU instruction detected here
```

Proper Error Synchronization:

```
FILD     COUNT              ; FPU instruction
FSQRT                 ; subsequent FPU instruction -- error from
                ; previous FPU instruction detected here
INC COUNT              ; integer instruction alters operand
```

programs to execute correctly, with INC COUNT being executed in parallel in the processor, as long as no exceptions occur on the FILD instruction. However, if the code is later moved to an environment where exceptions are unmasked, the code in the first example will not work correctly:

In some operating systems supporting the FPU, the numeric register stack is extended to memory. To extend the FPU stack to memory, the invalid exception is unmasked. A push to a full register or pop from an empty register sets SF (Stack Fault flag) and causes an invalid operation exception. The recovery routine for the exception must recognize this situation, fix up the stack, then perform the original operation. The recovery routine will not work correctly in the first example shown in the figure. The problem is that the value of COUNT is incremented before the exception handler is invoked, so that the recovery routine will load an incorrect value of COUNT, causing the program to fail or behave unreliably.

3.3.3 PROPER EXCEPTION SYNCHRONIZATION IN GENERAL

As explained before (see Section 3.2), if the FPU encounters an unmasked exception condition a software exception handler is invoked *before* execution of the *next* WAIT or floating-point instruction. This is because an unmasked floating-point exception causes the processor to freeze immediately before executing such an instruction (unless the IGNNE# input is active, or it is a “No-Wait” FPU instruction). Exactly when the exception handler will be invoked (in the interval between when the exception is detected and the next WAIT or FPU instruction) is dependent on the processor generation, the system, and which FPU instruction and exception is involved.

To be safe in exception synchronization, one should assume the handler will be invoked at the end of the interval. Thus the program should not change any value that might be needed by the handler (such as COUNT in the above example) until *after* the *next* FPU instruction following an FPU instruction that could cause an error. If the program needs to modify such a value before the next FPU instruction (or if the next FPU instruction could also cause an error), then a WAIT instruction should be inserted before the value is modified. This will force the handling of any exception before the value is modified. A WAIT instruction should also be placed after the last floating-point instruction in an application so that any unmasked exceptions will be serviced before the task completes.

3.4 FPU Exception Handling Examples

There are many approaches to writing exception handlers. One useful technique is to consider the exception handler procedure as consisting of "prologue," "body," and "epilogue" sections of code.

In the transfer of control to the exception handler due to an INTR, NMI, or SMI, external interrupts have been disabled by hardware. The prologue performs all functions that must be protected from possible interruption by higher-priority sources. Typically, this involves saving registers and transferring diagnostic information from the FPU to memory. When the critical processing has been completed, the prologue may re-enable interrupts to allow higher-priority interrupt handlers to preempt the exception handler. The standard "prologue" not only saves the registers and transfers diagnostic information from the FPU to memory but also clears the FP exception flags in the status word. Alternatively, when it is not necessary for the

handler to be re-entrant, another technique may also be used. In this technique, the exception flags are not cleared in the "prologue" and the body of the handler must not contain any FP instructions that cannot complete execution when there is a pending FP exception. (The “No-Wait” instructions are discussed in Section 6.3.7 of the *Pentium® Processor Family Developer’s Manual,* Volume 3). Note that the handler must still clear the exception flag(s) before executing the IRET. If the exception handler uses neither of these techniques the system will be caught in an endless loop of nested FP exceptions, and hang.

The body of the exception handler examines the diagnostic information and makes a response that is necessarily application-dependent. This response may range from halting execution, to displaying a message, to attempting to repair the problem and proceed with normal execution. The epilogue essentially reverses the actions of the prologue, restoring the processor so that normal execution can be resumed. The epilogue must not load an unmasked exception flag into the FPU or another exception will be requested immediately.

The following code examples show the ASM386 and ASM486 coding of three skeleton exception handlers, with the save spaces given as correct for 32 bit protected mode. They show how prologues and epilogues can be written for various situations, but the application dependent exception handling body is just indicated by comments showing where it should be placed.

The first two are very similar; their only substantial difference is their choice of instructions to save and restore the FPU. The tradeoff here is between the increased diagnostic information provided by FNSAVE and the faster execution of FNSTENV. (Also, after saving the original contents, FNSAVE re-initializes the FPU, while FNSTENV only masks all FPU exceptions.) For applications that are sensitive to interrupt latency or that do not need to examine register contents, FNSTENV reduces the duration of the "critical region," during which the processor does not recognize another interrupt request. (See the *Pentium® Processor Family Developer’s Manual*, Volume 3, Section 6.2.1.6 for a complete description of the FPU save image.)

After the exception handler body, the epilogues prepare the processor to resume execution from the point of interruption (i.e., the instruction following the one that generated the unmasked exception). Notice that the exception flags in the memory image that is loaded into the FPU are cleared to zero prior to reloading (in fact, in these examples, the entire status word image is cleared).

Example 1 and Example 2 assume that the exception handler itself will not cause an unmasked exception. Where this is a possibility, the general approach shown in 3 can be employed. The basic technique is to save the full FPU state and then to load a new control word in the prologue. Note that considerable care should be taken when designing an exception handler of this type to prevent the handler from being reentered endlessly.

Example 1. Full-State Exception Handler

```
SAVE_ALL    PROC
;
; SAVE REGISTERS, ALLOCATE STACK SPACE FOR FPU STATE IMAGE
      PUSH  EBP
      .
      .
      MOV   EBP, ESP
      SUB   ESP, 108 ; ALLOCATES 108 BYTES (32-bit PROTECTED MODE SIZE)
;SAVE FULL FPU STATE, RESTORE INTERRUPT ENABLE FLAG (IF)
      FNSAVE      [EBP-108]
      PUSH [EBP + OFFSET_TO_EFLAGS] ; COPY OLD EFLAGS TO STACK TOP
      POPFD       ; RESTORE IF TO VALUE BEFORE FPU EXCEPTION

;
; APPLICATION-DEPENDENT EXCEPTION HANDLING CODE GOES HERE
;
; CLEAR EXCEPTION FLAGS IN STATUS WORD (WHICH IS IN MEMORY)
```

```
; RESTORE MODIFIED STATE IMAGE
      MOV         BYTE PTR [EBP-104], 0H
      FRSTOR      [EBP-108]
; DE-ALLOCATE STACK SPACE, RESTORE REGISTERS
      MOV         ESP, EBP
      .
      .
      POP         EBP
;
; RETURN TO INTERRUPTED CALCULATION
      IRETD
SAVE_ALL    ENDP
```

Example 2. Reduced-Latency Exception Handler

```
SAVE_ENVIRONMENT  PROC
;
; SAVE REGISTERS, ALLOCATE STACK SPACE FOR FPU ENVIRONMENT
      PUSH        EBP
      .
      .
      MOV         EBP, ESP
      SUB         ESP, 28 ; ALLOCATES 28 BYTES (32-bit PROTECTED MODE SIZE)
;SAVE ENVIRONMENT, RESTORE INTERRUPT ENABLE FLAG (IF)
      FNSTENV     [EBP-28]
      PUSH [EBP + OFFSET_TO_EFLAGS] ; COPY OLD EFLAGS TO STACK TOP
      POPFD       ; RESTORE IF TO VALUE BEFORE FPU EXCEPTION

;
; APPLICATION-DEPENDENT EXCEPTION HANDLING CODE GOES HERE
;
; CLEAR EXCEPTION FLAGS IN STATUS WORD (WHICH IS IN MEMORY)
; RESTORE MODIFIED ENVIRONMENT IMAGE
      MOV         BYTE PTR [EBP-24], 0H
      FLDENV      [EBP-28]
; DE-ALLOCATE STACK SPACE, RESTORE REGISTERS
      MOV         ESP, EBP
      .
      .
      POP         EBP
;
; RETURN TO INTERRUPTED CALCULATION
      IRETD
SAVE_ENVIRONMENT  ENDP
```

Example 3. Reentrant Exception Handler

```
LOCAL_CONTROL DW ?      ; ASSUME INITIALIZED
      .
      .
REENTRANT   PROC
;
; SAVE REGISTERS, ALLOCATE STACK SPACE FOR FPU STATE IMAGE
```

```
        PUSH  EBP
        .
        .
        MOV         EBP, ESP
        SUB         ESP, 108 ; ALLOCATES 108 BYTES (32-bit PROTECTED MODE SIZE)

; SAVE STATE, LOAD NEW CONTROL WORD, RESTORE INTERRUPT ENABLE FLAG (IF)
        FNSAVE      [EBP-108]
        FLDCW       LOCAL_CONTROL
        PUSH [EBP + OFFSET_TO_EFLAGS] ; COPY OLD EFLAGS TO STACK TOP
        POPFD       ; RESTORE IF TO VALUE BEFORE FPU EXCEPTION

        .
        .
;
; APPLICATION-DEPENDENT EXCEPTION HANDLING CODE GOES HERE. AN UNMASKED EXCEPTION
; GENERATED HERE WILL CAUSE THE EXCEPTION HANDLER TO BE REENTERED.
; IF LOCAL STORAGE IS NEEDED, IT MUST BE ALLOCATED ON THE STACK.
;
        .
        .
; CLEAR EXCEPTION FLAGS IN STATUS WORD (WHICH IS IN MEMORY)
; RESTORE MODIFIED STATE IMAGE
        MOV         BYTE PTR [EBP-104], 0H
        FRSTOR      [EBP-108]
; DE-ALLOCATE STACK SPACE, RESTORE REGISTERS
        MOV         ESP, EBP
        .
        .
        POP         EBP
;
; RETURN TO POINT OF INTERRUPTION
        IRETD
REENTRANT   ENDP
```

3.5 Need for Preserving the State of IGNNE# Circuit if Use FPU and SMM

In Section 2.3.2 the recommended circuit (Figure 2) for MS-DOS compatible FPU exception handling for Intel486 processors and beyond contains two flip flops. When the FPU exception handler accesses I/O port 0F0H it clears the IRQ13 interrupt request output from Flip Flop #1 and also clocks out the IGNNE# signal (active) from Flip Flop #2. The assertion of IGNNE# may be used by the handler if needed to execute any FPU instruction while ignoring the pending FPU errors. The problem here is that the state of Flip Flop #2 is effectively an additional (but hidden) status bit that can affect processor behavior, and so ideally should be saved upon entering SMM, and restored before resuming to normal operation. If this is not done, and also the SMM code saves the FPU state, AND an FPU error handler is being used which relies on IGNNE# assertion, then (very rarely) the FPU handler will nest inside itself and malfunction. The following example shows how this can happen.

The problem will only occur if the processor enters SMM between the OUT and the FLDCW instructions. But if that happens, AND the SMM code saves the FPU state using FNSAVE, then the IGNNE# Flip Flop will be cleared (because FNSAVE clears the FPU errors and thus de-asserts FERR#). When the processor returns from SMM it will restore the FPU state with FRSTOR, which will re-assert FERR#, but the IGNNE# Flip Flop will not get set. Then when the FPU error handler executes the FLDCW instruction, the active error condition will cause the processor to re-enter the FPU error handler from the beginning. This may cause the handler to malfunction.

Note that NMI (or any interrupt through INTR that is enabled inside the FPU exception handler) will cause this same problem, if the interrupt routine saves and restores the FPU state, and it happens to occur between the OUT and the FLDCW instructions. SMI is the main focus here because it is much more likely to invoke FNSAVE/FRSTOR than other interrupts because of 0V suspend (see below). The problem can easily be eliminated from all interrupts besides SMI and NMI by not enabling INTR inside the FPU exception handler.

To avoid this problem, Intel recommends two measures:

1. Do not use the FPU for calculations inside SMM code (or code for NMI, or any other interrupts enabled inside the FPU exception handler). (The normal power management, and sometimes security, functions provided by SMM have no need for FPU calculations; if they are needed for some special case, use scaling or emulation instead.) This eliminates the need to do FNSAVE/FRSTOR inside SMM code, *except* when going into an 0V suspend state (in which, in order to save power, the processor is turned off completely, requiring its complete state to be saved).
2. The system should not call upon SMM code to put the processor into 0V suspend while the processor is running FPU calculations, or just after an interrupt has occurred. Normal power management protocol avoids this by going into power down states only after timed intervals in which no system activity occurs.

3.6 Considerations When FPU Shared Between Tasks

The Intel Architecture allows speculative deferral of floating-point state swaps on task switches. This feature allows postponing an FPU state swap until an FPU instruction is actually encountered in another task. Since kernel tasks rarely use floating-point, and some applications do not use floating-point or use it infrequently, the amount of time saved by avoiding unnecessary stores of the floating-point state is significant. Speculative deferral of FPU saves does, however, place an extra burden on the kernel in three key ways:

1. The kernel must keep track of which thread owns the FPU, which may be different from the currently executing thread.
2. The kernel must associate any floating-point exceptions with the generating task. This requires special handling since floating-point exceptions are delivered asynchronous with other system activity.
3. There are conditions under which spurious floating-point exception interrupts are generated, which the kernel must recognize and discard.

Suppose that the FPU exception handler includes the following sequence:

```
FNSTSW save_sw          ; save the FPU status word using a “No-Wait” FPU instruction
OUT 0F0H, AL            ; clears IRQ13 & activates IGNNE#
 . . . .
FLDCW new_cw ; loads new CW ignoring FPU errors, since IGNNE# is assumed active; or any
                        ; other FPU instruction that is not a “No-Wait” type will cause the
same problem
 . . . .
FCLEX                   ; clear the FPU error conditions & thus turn off FERR# & reset the
IGNNE# FF
```

3.6.1 SPECULATIVELY DEFERRING FPU SAVES, GENERAL OVERVIEW

In order to support multi-tasking, each thread in the system needs a save area for the general purpose registers, and each task that is allowed to use floating-point needs an FPU save area large enough to hold the entire FPU stack and associated FPU state such as the control word and status word. (See the *Pentium® Processor Family Developer's Manual*, Volume 3, Section 6.2.1.6 for a complete description of the FPU save image.)

On a task switch, the general purpose registers are swapped out to their save area for the suspending thread, and the registers of the resuming thread are loaded. The FPU state does not need to be saved at this point. If the resuming thread does not use the FPU before it is itself suspended, then both a save and a load of the FPU state has been avoided. It is often the case that several threads may be executed without any usage of the FPU.

The processor supports speculative deferral of FPU saves via interrupt 7 "Device Not Available" (DNA), used in conjunction with CR0 bit 3, the "Task Switched" bit (TS). (See the *Pentium® Processor Family Developer's Manual,* Volume 3, Sections 10.1.3 & 14.9.7) Every task switch via the hardware supported task switching mechanism (see Section 13.5 of the *Pentium® Processor Family Developer's Manual,* Volume 3) sets TS. Multi-threaded kernels that use software task switching[4] can set the TS bit by reading CR0, ORing a '1' into bit 3, and writing back CR0[5]. Any subsequent floating-point instructions (now being executed in a new thread context) will fault via interrupt 7 before execution.

Footnotes

[4] In a software task switch, the operating system uses a sequence of instructions to save the suspending thread's state and restore the resuming thread's state instead of the single long, noninterruptable task switch operation provided by the Intel Architecture.

[5] Although CR0, bit 2, the emulation flag (EM), also causes a DNA exception, *do not* use the EM bit as a surrogate for TS. EM means that no floating-point unit is available and that FP instructions must be emulated. Using EM to trap on task switches is not compatible with Intel Architecture MMX™ Technology. If the EM flag is set, MMX instructions raise the invalid opcode exception.

This allows the DNA handler to save the old floating-point context and reload the FPU state for the current thread. The handler should clear the TS bit before exit using the CLTS instruction. On return from the handler the faulting thread will proceed with its floating-point computation.

Some operating systems save the FPU context on every task switch, typically because they also change the linear address space between tasks. The problem and its solution discussed below apply to these operating systems also.

3.6.2 TRACKING FPU OWNERSHIP

Since the contents of the FPU may not belong to the currently executing thread, the thread identifier for the last FPU user needs to be tracked separately. This is not complicated -- the kernel should simply provide a variable to store the thread identifier of the FPU owner, separate from the variable that stores the identifier for the currently executing thread. This variable is updated in the DNA exception handler, and is used by the DNA exception handler to find the FPU save areas of the old and new threads. A simplified flow for a DNA exception handler is then:

1. Use the 'FPU Owner' variable to find the FPU save area of the last thread to use the FPU.
2. Save the FPU contents to the old thread's save area, typically using an FNSAVE instruction.
3. Set the 'FPU Owner' variable to the identify the currently executing thread.
4. Reload the FPU contents from the new thread's save area, typically using an FRSTOR instruction.
5. Clear TS using the CLTS instruction and exit the DNA exception handler.

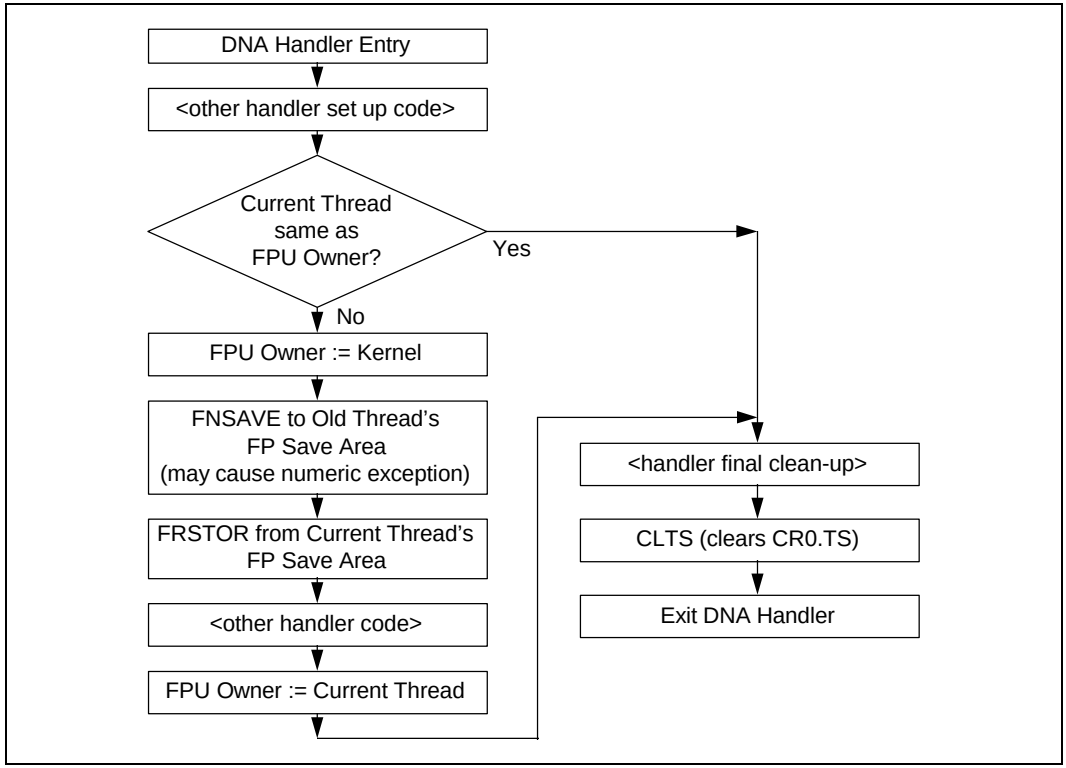
While this flow covers the basic requirements for speculatively deferred FPU state swaps, there are some additional subtleties that need to be handled in a robust implementation.

3.6.3 INTERACTION OF FPU STATE SAVES AND FP EXCEPTION ASSOCIATION

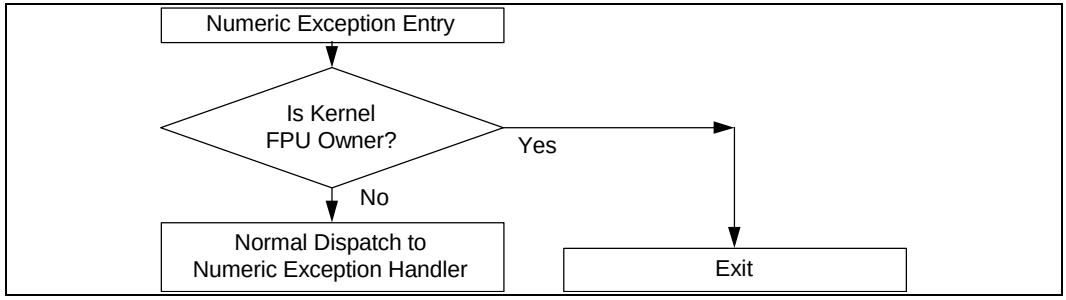
Recall these key points from earlier in this document: When considering FP exceptions across all implementations of the Intel Architecture, and across all FP instructions, an FP exception can be

initiated from any time during the excepting FP instruction, up to just before the next FP instruction. The 'next' FP instruction may be the FNSAVE used to save the FPU state for a task switch. In the case of "no-wait:" instructions such as FNSAVE, the interrupt from a previously excepting instruction (NE=0 case) may arrive just before the "no-wait" instruction, during, or shortly thereafter with a system dependent delay. Note that this implies that an FP exception might be registered during the state swap process itself, and the kernel and FP exception interrupt handler must be prepared for this case.

A simple way to handle the case of exceptions arriving during FPU state swaps is to allow the kernel to be one of the FPU owning threads. A reserved thread identifier is used to indicate kernel ownership of the FPU. During an FP state swap, the 'FPU owner' variable should be set to indicate the kernel as the current owner. At the completion of the state swap, the variable should be set to indicate the new owning thread. The numeric exception handler needs to check the FPU owner and *discard* any numeric exceptions that occur while the kernel is the FPU owner. A more general flow for a DNA exception handler that handles this case is shown next:



Numeric exceptions received while the kernel owns the FPU for a state swap must be discarded in the kernel without being dispatched to a handler. A flow for a numeric exception dispatch routine is shown below:



It may at first glance seem that there is a possibility of FP exceptions being lost because of exceptions that are discarded during state swaps. This is not the case, as the exception will be re-issued when the FP state is reloaded. Walking through state swaps both with and without pending numeric exceptions will clarify the operation of these two handlers.

Case 1: FPU State Swap Without Numeric Exception

Assume two threads 'A' and 'B', both using the floating-point unit. Let A be the thread to have most recently executed a FP instruction, with no pending numeric exceptions. Let B be the currently executing thread. CR0.TS was set when thread A was suspended. When B starts to execute a FP instruction the instruction will fault with the DNA exception because TS is set.

At this point the handler is entered, and eventually it finds that the current FPU Owner is not the currently executing thread. To guard the FPU state swap from extraneous numeric exceptions, the FPU Owner is set to be the kernel. The old owner's FPU state is saved with FNSAVE, and the current thread's FPU state is restored with FRSTOR. Before exiting, the FPU owner is set to thread B, and the TS bit is cleared.

On exit, thread B resumes execution of the faulting FP instruction and continues.

Case 2: FPU State Swap with Discarded Numeric Exception

Again, assume two threads 'A' and 'B', both using the floating-point unit. Let A be the thread to have most recently executed a FP instruction, but this time let there be a pending numeric exception. Let B be the currently executing thread. When B starts to execute a FP instruction the instruction will fault with the DNA exception and enter the DNA handler. (If both numeric and DNA exceptions are pending, the DNA exception takes precedence, in order to support handling the numeric exception in its own context.)

When the FNSAVE starts, it will trigger an interrupt via FERR# because of the pending numeric exception. After some system dependent delay, the numeric exception handler is entered. It may be entered before the FNSAVE starts to execute, or it may be entered shortly after execution of the FNSAVE. Since the FPU Owner is the kernel, the numeric exception handler simply exits, discarding the exception. The DNA handler resumes execution, completing the FNSAVE of the old FP context of thread A and the FRSTOR of the FP context for thread B.

Thread A eventually gets an opportunity to handle the exception that was discarded during the task switch. After some time, thread B is suspended, and thread A resumes execution. When thread A starts to execute an FP instruction, once again the DNA exception handler is entered. B's FPU state is FNSAVE'ed, and A's FPU state is FRSTOR'ed. Note that in restoring the FPU state from A's save area, the pending numeric exception flags are reloaded in to the FP status word. Now when the DNA exception handler returns, thread A resumes execution of the faulting FP instruction just long enough to immediately generate a numeric exception, which now gets handled in the normal way. The net result is that the task switch and resulting FPU state swap via the DNA exception handler causes an 'extra' numeric exception which can be safely discarded.

3.6.4 INTERRUPT ROUTING FROM THE KERNEL

In MS-DOS, an application that wishes to handle numeric exceptions hooks interrupt 2 by placing its handler address in the interrupt vector table, and exiting via a jump to the previous interrupt 2 handler. Protected mode systems that run MS-DOS programs under a subsystem can emulate this exception delivery mechanism. For example, assume a protected mode O.S. that runs with CR.NE = 1, and that runs MS-DOS programs in a virtual machine subsystem. The MS-DOS program is set up in a virtual machine that provides a virtualized interrupt table. The MS-DOS application hooks interrupt 2 in the virtual machine in the normal way. A numeric exception will trap to the kernel via the real INT 16 residing in the kernel at ring 0. The INT 16 handler in the kernel then locates the correct MS-DOS virtual machine, and reflects the interrupt to the virtual machine monitor. The virtual machine monitor then emulates an interrupt by jumping through the address in the virtualized interrupt table, eventually reaching the application's numeric exception handler.

4.0 DIFFERENCES FOR HANDLERS USING NATIVE MODE

The 8087 has a pin INT which it asserts when an unmasked exception occurs. But there is no interrupt input pin in the 8086 or 8088 dedicated to its attachment, nor an interrupt vector number in the 8086 or 8088 specific for an FPU error assertion. But beginning with the Intel 80286 and 80287, hardware connections were dedicated to support the FPU exception, and interrupt vector 16 assigned to it.

4.1 Origin with 80286 and 80287; Intel386™ Processor and Intel387 Math Coprocessor

The 80286 and 80287 and Intel386 processor and Intel387 math coprocessor pairs are each provided with ERROR# pins that are recommended to be connected between the processor and FPU. If this is done, when an unmasked FPU exception occurs, the FPU records the exception, and asserts its ERROR# pin. The processor recognizes this active condition of the ERROR# status line immediately before execution of the next WAIT or FPU instruction (except for the "N0-Wait" type) in its instruction stream, and branches to the routine at interrupt vector 16. Thus an FPU exception will be handled before any other FPU instruction (after the one causing the error) is executed (except for "No-Wait" instructions, which will be executed without triggering the FPU exception interrupt, but it will remain pending).

Using the dedicated interrupt 16 for FPU exception handling is referred to as the native mode. It is the simplest approach, and the one recommended most highly by Intel.

4.2 Changes with Intel486Ô, Pentium® and Pentium Pro Processors with CR0.NE=1

With these latest three generations of the Intel Architecture, more enhancements and speedup features have been added to the corresponding FPUs. Also, the FPU is now built into the same chip as the processor, which allows further increases in the speed at which the FPU can operate as part of the integrated system. This also means that the native mode of FPU exception handling, selected by setting bit NE of register CR0 to 1, is now entirely internal.

If an unmasked exception occurs during an FPU instruction, the FPU records the exception internally, and triggers the exception handler through interrupt 16 immediately before execution of the next WAIT or FPU instruction (except for "No-Wait" instructions, which will be executed as described in Section 4.1 above).

An unmasked numerical exception causes the FERR# output to be activated even with NE=1, and at exactly the same point in the program flow as it would have been asserted if NE were zero. However, the system would not connect FERR# to a PIC to generate INTR when operating in the native, internal mode. (If the hardware of a system has FERR# connected to trigger IRQ13 in order to support MS-DOS, but an OS using the native mode is actually running the system, it is the OSs responsibility to make sure that IRQ13 is not enabled in the slave PIC.) With this configuration a system is immune to the problem discussed in Section 2.3.3, where for Intel486 and Pentium processors a "No-Wait" FPU instruction can get an FPU exception.

4.3 Considerations When FPU Shared Between Tasks Using Native Mode

The protocols recommended in Section 3.6 for MS-DOS compatible FPU exception handlers that are shared between tasks may be used without change with the native mode. However, the protocols for a handler written specifically for native mode can be simplified, because the problem of a spurious floating-point exception interrupt occurring while the kernel is executing cannot happen in native mode.

The problem as actually found in practical code in a MS-DOS compatible system happens when the DNA handler uses FNSAVE to switch FPU contexts. If an FPU exception is active, then FNSAVE triggers FERR# briefly, which usually will cause the FPU exception handler to be invoked inside the DNA handler. In native mode, neither FNSAVE nor any other "No-Wait" instructions can trigger interrupt 16. (As discussed above, FERR# gets asserted independent of the value of the NE bit, but when NE=1, the OS should not enable its path through the PIC.) Another possible (very rare) way a floating-point exception interrupt could occur while the kernel is executing is by an FPU immediate exception case having its interrupt delayed by the external hardware until execution has switched to the kernel. This also cannot

happen in native mode because there is no delay through external hardware.

Thus the native mode FPU exception handler can omit the test to see if the kernel is the FPU owner, and the DNA handler for a native mode system can omit the step of setting the kernel as the FPU owner at the handler's beginning. Since however these simplifications are minor and save little code, it would be a reasonable and conservative habit (as long as the MS-DOS compatible mode is widely used) to include these steps in all systems.

Note that the special DP (Dual Processing) mode for Pentium processors, and also the more general Intel MultiProcessor Specification for systems with multiple Pentium or Pentium Pro processors, support FPU exception handling only in the native mode. Intel does not recommend using the MS-DOS compatible FPU mode for systems using more than one processor.

XA User Guide

Philips Semiconductors and Philips Electronics North America Corporation reserve the right to make changes, without notice, in the products, including circuits, standard cells, and/or software, described or contained herein in order to improve design and/or performance. Philips Semiconductors assumes no responsibility or liability for the use of any of these products, conveys no license or title under any patent, copyright, or mask work right to these products, and makes no representations or warranties that these products are free from patent, copyright, or mask work right infringement, unless otherwise specified. Applications that are de scribed herein for any of these products are for illustrative purposes only. Philips Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

LIFE SUPPORT APPLICATIONS

Philips Semiconductors and Philips Electronics North America Corporation Products are not designed for use in life support appliances, devices, or systems where malfunction of a Philips Semiconductors and Philips Electronics North America Corporation Product can reasonably be expected to result in a personal injury. Philips Semiconductors and Philips Electronics North America Corporation customers using or selling Philips Semiconductors and Philips Electronics North America Corporation Products for use in such applications do so at their own risk and agree to fully indemnify Philips Semiconductors and Philips Electronics North America Corporation for any damages resulting from such improper use or sale.

Printed in U.S.A.

1 The XA Family - High Performance, Enhanced Architecture 80C51-Compatible 16-Bit CMOS Microcontrollers

1.1 Introduction

The role of the microcontroller is becoming increasingly important in the world of electronics as systems which in the past relied on mechanical or simple analog electrical control systems have microcontrollers embedded in them that dramatically improve functionality and reliability, while reducing size and cost. Microcontrollers also provide the general purpose solutions needed so that common software and hardware can be shared among multiple designs to reduce overall design-in time and costs.

The requirements of systems using microcontrollers are also much more demanding now than a few years ago. Whether called by the name “microcontrollers”, “embedded controllers” or “single-chip microcomputers”, the systems that use these devices require a much higher level of performance and on-chip integration.

As microcontrollers begin to enter into more complex control environments, the demand for increased throughput, increased addressing capability, and higher level of on-chip integration has led to the development of 16-bit microcontrollers that are capable of processing much more information than 8-bit microcontrollers. However, simply integrating more bits or more peripheral functions does not solve the demand of the control systems being developed today. New microcontrollers must provide high-level-language support, powerful debugging environments, and advanced methods of real time control in order to meet the more stringent functionality and cost requirements of these systems.

To meet the above goals The XA or “eXtended Architecture” family of general-purpose microcontrollers from Philips is being introduced to provide the highest performance/cost ratio for a variety of high performance embedded-systems-control applications including real-time, multi-tasking environments. The XA family members add to the CPU core a specific complement of on-chip memory, I/Os, and peripherals aimed at meeting the requirements of different application areas. The core-based architecture allows easy expansion of the family according to a wide variety of customer requirements. The powerful instruction set supports faster computing power, faster data transfer, multi-tasking, improved response to external events and efficient high-level language programming.

Upward (assembly-level) code compatibility with the Philips 80C51 family of controllers provides a smooth design transition for system upgrades by providing tremendously enhanced performance.

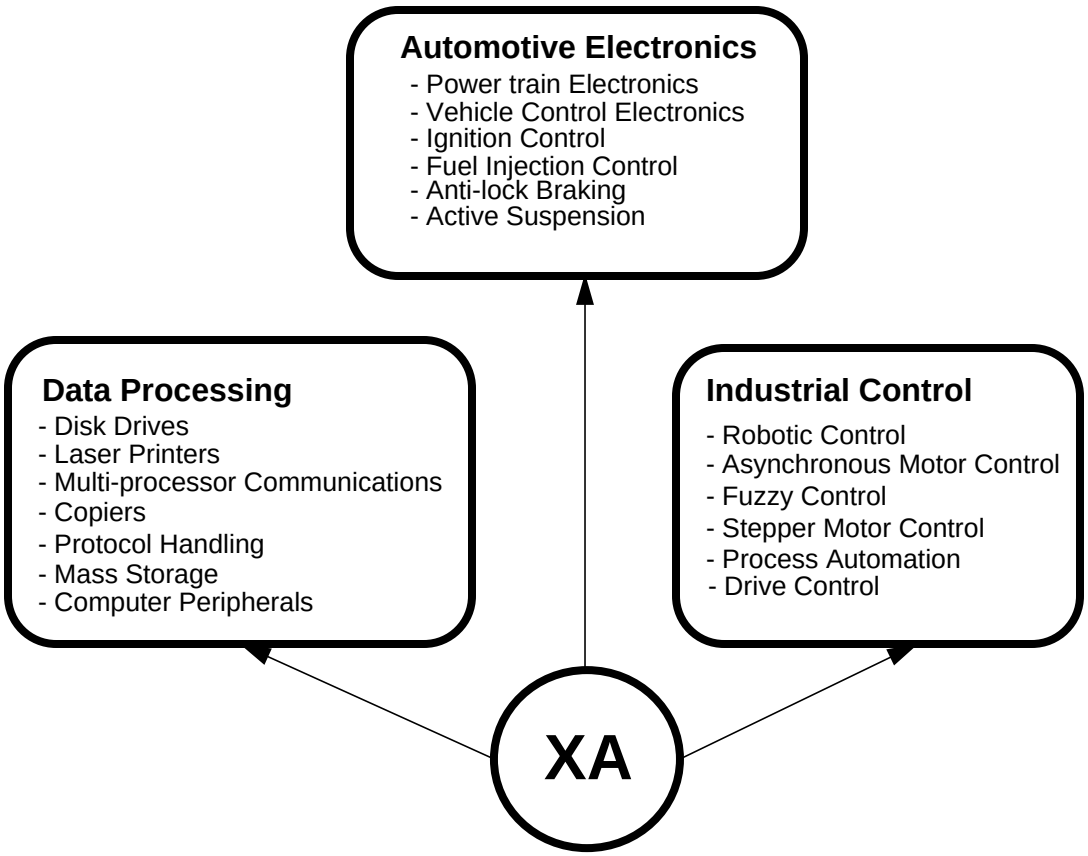


Figure 1. Applications of Philips XA microcontrollers

1.2 Architectural Features of XA

- Upward compatibility with the standard 8XC51 core (assembly source level)
- 24-bit address range (16 Megabytes code and data space)
- 16-bit static CPU
- Enhanced architecture using both 16-bit words and 8-bit bytes
- Enhanced instruction set
- High code efficiency; most of the instructions are 2-4 bytes in length
- Fast 16X16 Multiply and 32x16 Divide Instructions
- 16-bit Stack Pointers and general pointer registers
- Capability to support 32 vectored interrupts - 31 maskable and 1 NMI
- Supports 16 hardware and 16 software traps
- Power Down and Idle power reduction modes
- Hardware support for multi-tasking software

2 Architectural Overview

2.1 Introduction

The Philips XA (eXtended Architecture) has a general purpose register-register architecture to provide the best cost-to-performance trade-off available for a high speed microcontroller using today's technology. Intended as both an upward compatibility path for 80C51 users who need greater performance or more memory, and as a powerful, general-purpose 16-bit controller, the XA also incorporates support for multi-tasking operating systems and high-level languages such as C, while retaining the comprehensive bit-oriented operations that are the hallmark of the 80C51.

This overview introduces the concepts and terminology of the XA architecture in preparation for the detailed descriptions in the following sections of this manual.

2.2 Memory Organization

The XA architecture has several distinct memory spaces. The architecture and the instruction encoding are optimized for register based operations; in addition, arithmetic and logical operations may be done directly on data memory as well. Thus, the XA architecture avoids the bottleneck of having a single accumulator register.

2.2.1 Register File

The register file (Figure 2.1) allows access to 8 words of data at any one time; the eight words are also addressable as 16 bytes. The bottom 4 word registers are "banked". That is, there are four groups of registers, any one of which may occupy the bottom 4 words of the register file at any one time. This feature may be used to minimize the time required for context switching during interrupt service, and to provide more register space for complicated algorithms.

For some instructions –32-bit shifts, multiplies, and divides– adjacent pairs of word registers are referenced as double words.

The upper four words of the register file are not banked. The topmost word register is the stack pointer, while any other word register may be used as a general purpose pointer to data memory.

The entire register file is bit addressable. That is, any bit in the register file (except the 3 unselected banks of the bottom 4 words) may be operated on by bit manipulation instructions.

The XA instruction encoding allows for future expansion of the register file by the addition of 8 word registers. If implemented, these additional registers will be word data registers only and cannot be used as pointers or addressed as bytes.

The overall XA register file structure provides a superset of the 80C51 register structure. For details, refer to the section on 80C51 compatibility.

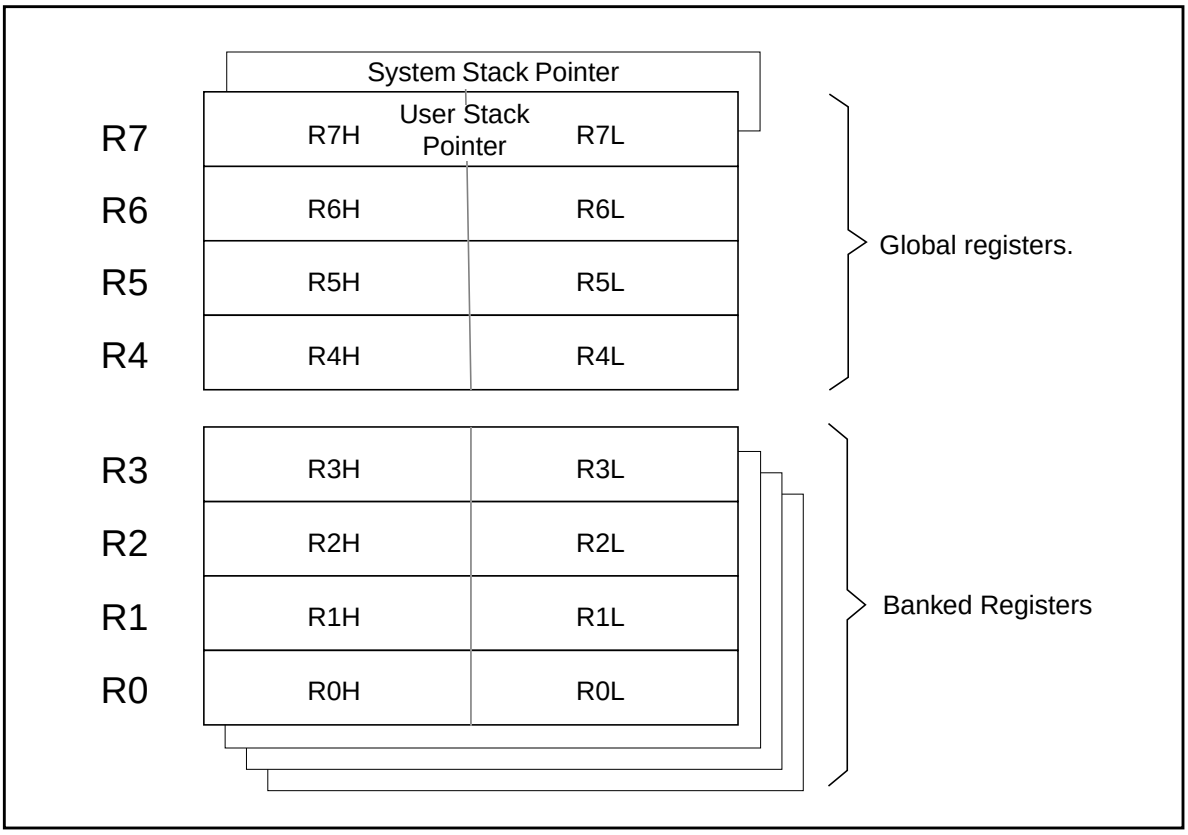


Figure 2.1 XA register file diagram

2.2.2 Data Memory

The XA architecture supports a 16 megabyte data memory space with a full 24-bit address. Some derivative parts may implement fewer address lines for a smaller range. The data space beginning at address 0 is normally on-chip and extends to the limit of the RAM size of a particular XA derivative. For addresses above that on a derivative, the XA will automatically roll over to external data memory.

Data memory in the XA is divided into 64K byte segments (Figure 2.2) to provide an intrinsic protection mechanism for multi-tasking systems and to improve performance. Segment registers provide the upper 8 address bits needed to obtain a complete 24-bit address in applications that require large data memories (Figure 2.3).

The XA provides 2 segment registers used to access data memory, the Data Segment register (DS) and the Extra Segment register (ES). Each pointer register is associated with one of the segment registers via the Segment Select (SSEL) register. Pointer registers retain this association until it is changed under program control.

The XA provides flexible data addressing modes. Most arithmetic, logic, and data movement instructions support the following modes of addressing data memory:

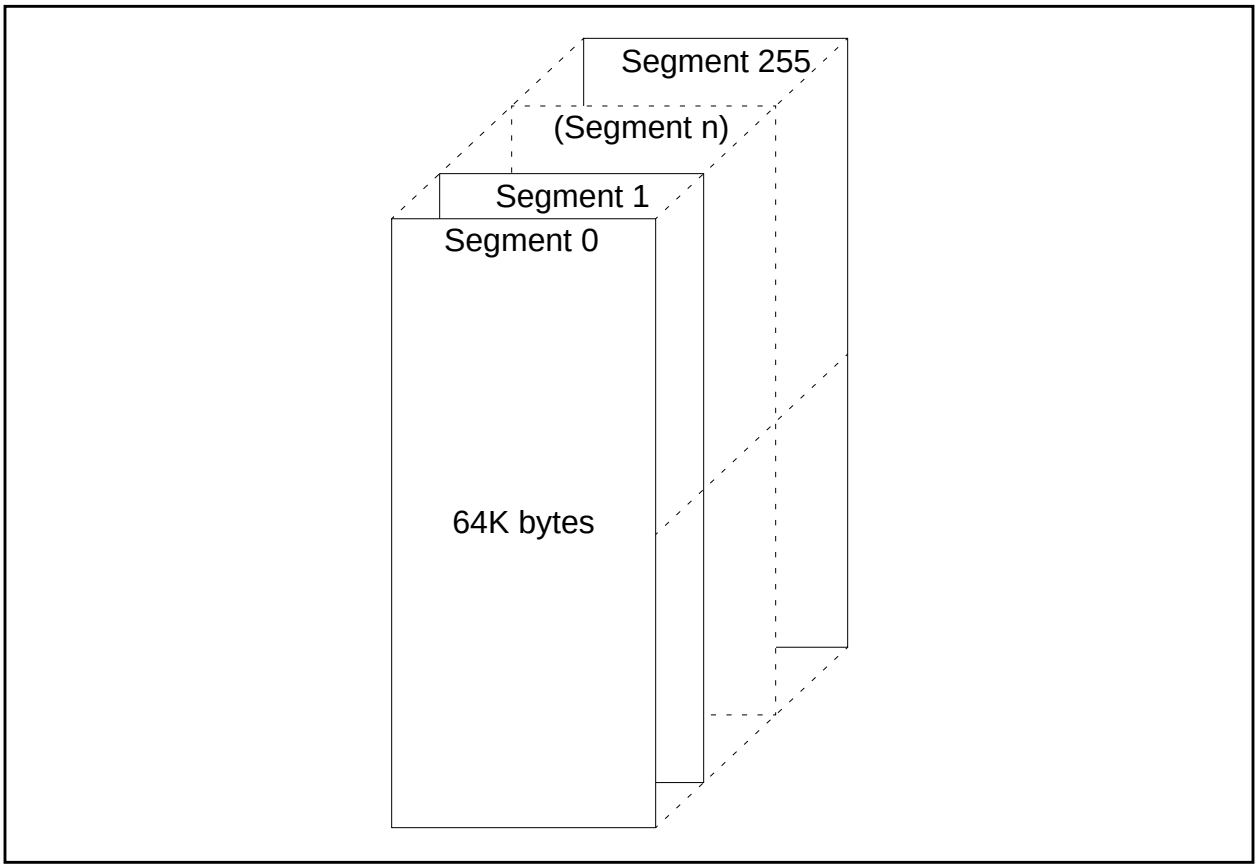


Figure 2.2 XA data memory segments

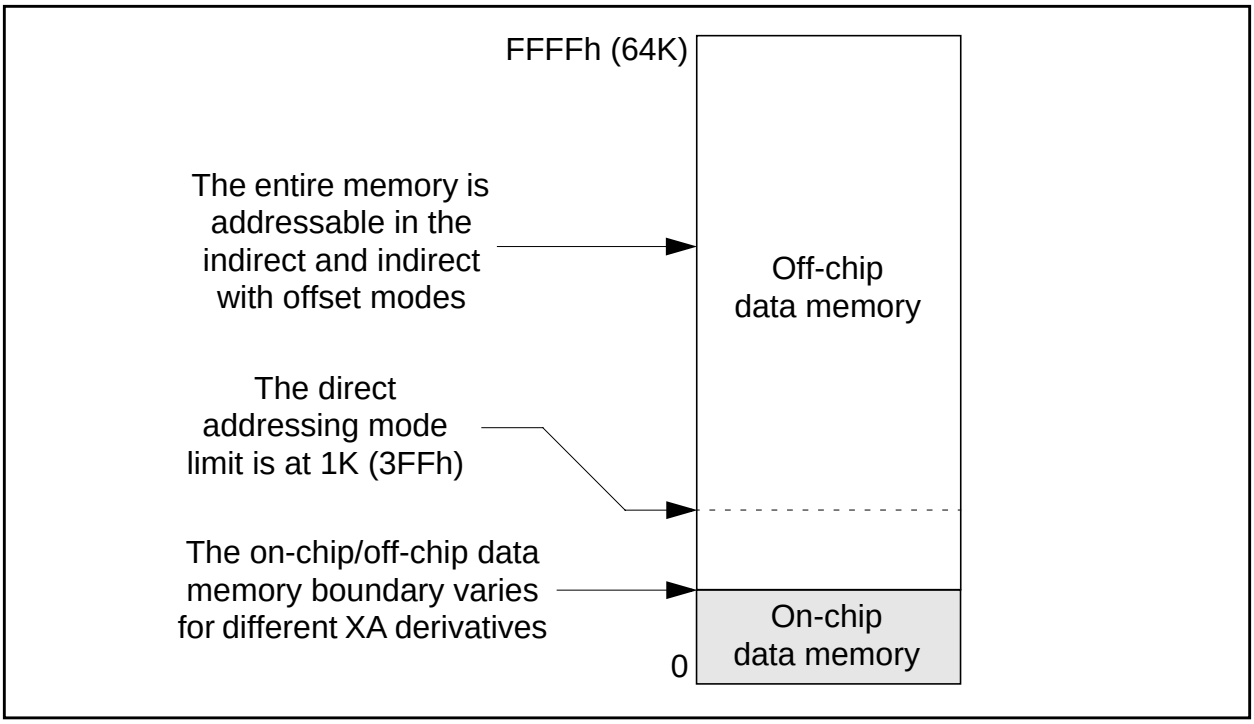


Figure 2.3 Simplified XA data memory diagram

Direct. The first 1K bytes of data on each segment may be accessed by an address contained within the instruction.

Indirect. A complete 24-bit data memory address is formed by an 8-bit segment register concatenated with 16-bits from a pointer register.

Indirect with offset. An 8-bit or 16-bit signed offset contained within the instruction is added to the contents of a pointer register, then concatenated with an 8-bit segment register to produce a complete address. This mode allows access into a data structure when a pointer register contains the starting address of the structure. It also allows subroutines to access parameters passed on the stack.

Indirect with auto-increment. The address is formed in the same manner as plain indirect, but the pointer register contents are automatically incremented following the operation.

Data movement instructions and some special purpose instructions also have additional data addressing modes.

The XA data memory addressing scheme provides for upward compatibility with the 80C51. For details, refer to Chapter 9.

2.2.3 Code Memory

The XA is a Harvard architecture device, meaning that the code and data spaces are separate. The XA provides a continuous, unsegmented linear code space that may be as large as 16 megabytes (Figure 2.4). In XA derivatives with on-chip ROM or EPROM code memory, the on-

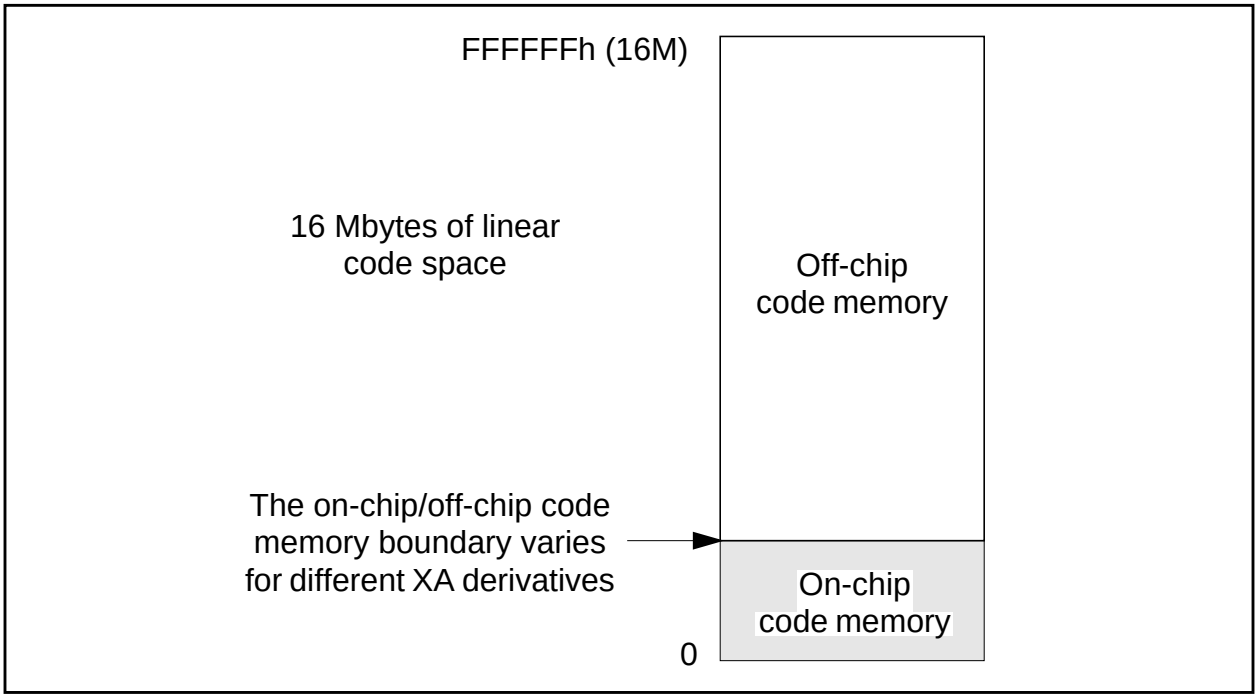


Figure 2.4 XA code memory map

chip space always begins at code address 0 and extends to the limit of the on-chip code memory. Above that, code will be fetched from off-chip. Most XA derivatives will support an external bus for off-chip data and code memory, and may also be used in a ROM-less mode, with no code memory used on-chip.

In some cases, code memory may be addressed as data. Special instructions provide access to the entire code space via pointers. Either a special segment register (CS or Code Segment) or the upper 8-bits of the Program Counter (PC) may be used to identify the portion of code memory referenced by the pointer.

2.2.4 Special Function Registers

Special Function Registers (SFRs) provide a means for the XA to access Core registers, internal control registers, peripheral devices, and I/O ports. Any SFR may be accessed by a program at any time without regard to any pointer or segment. An SFR address is always contained entirely within an instruction. See Figure 2.5.

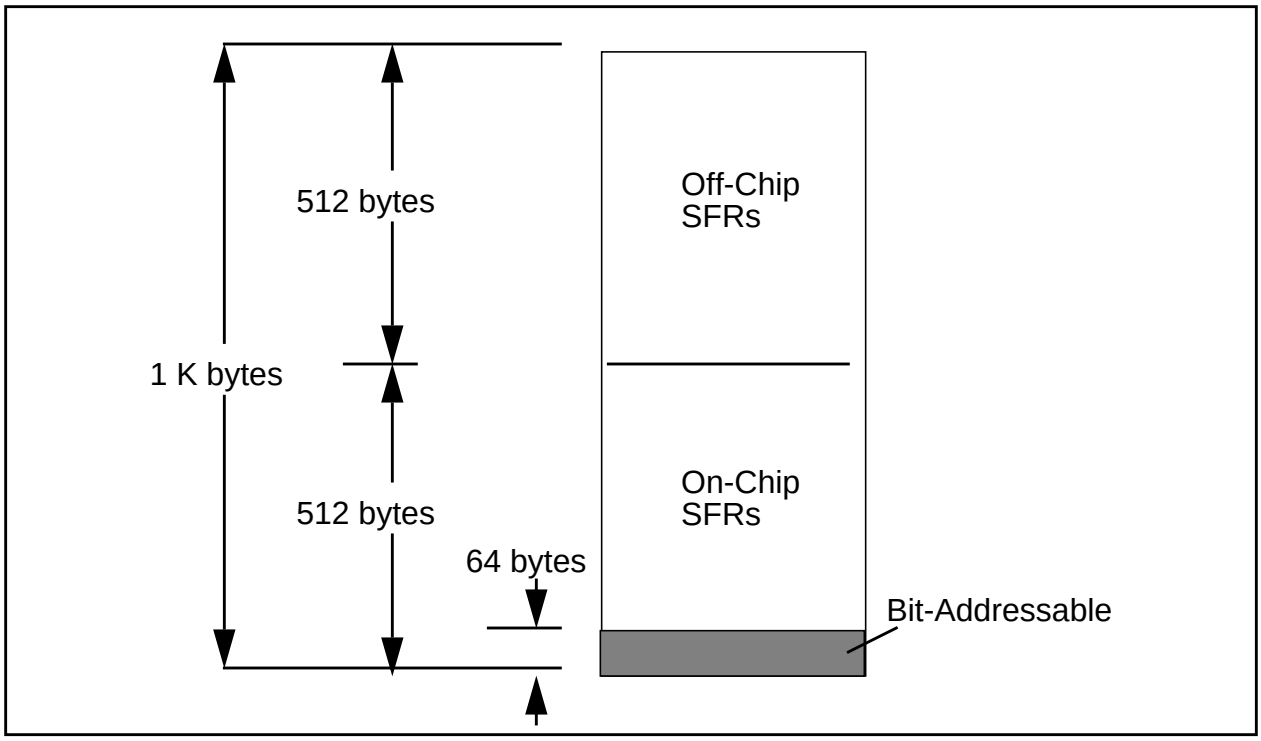


Figure 2.5 SFR Address Space

The total SFR space is 1K bytes in size. This is further divided into two 512 byte regions. The lower half is assigned to on-chip SFRs, while the second half is reserved for off-chip SFRs. This allows provides a means to add off-chip I/O devices mapped into the XA as SFRs. Off-chip SFR access is not implemented on all XA derivatives.

On-chip SFRs are implemented as needed to provide control for peripherals or access to CPU features and functions. Each XA derivative may have a different number of SFRs implemented

because each has a different set of peripheral functions. Many SFR addresses will be unused on any particular XA derivative.

The first 64 bytes of on-chip SFR space are bit-addressable. Any CPU or peripheral register that allows bit access will be allocated an address within that range.

2.3 CPU

Figure 2.6 shows the XA architecture as a whole. Each of the blocks shown are described in this section.

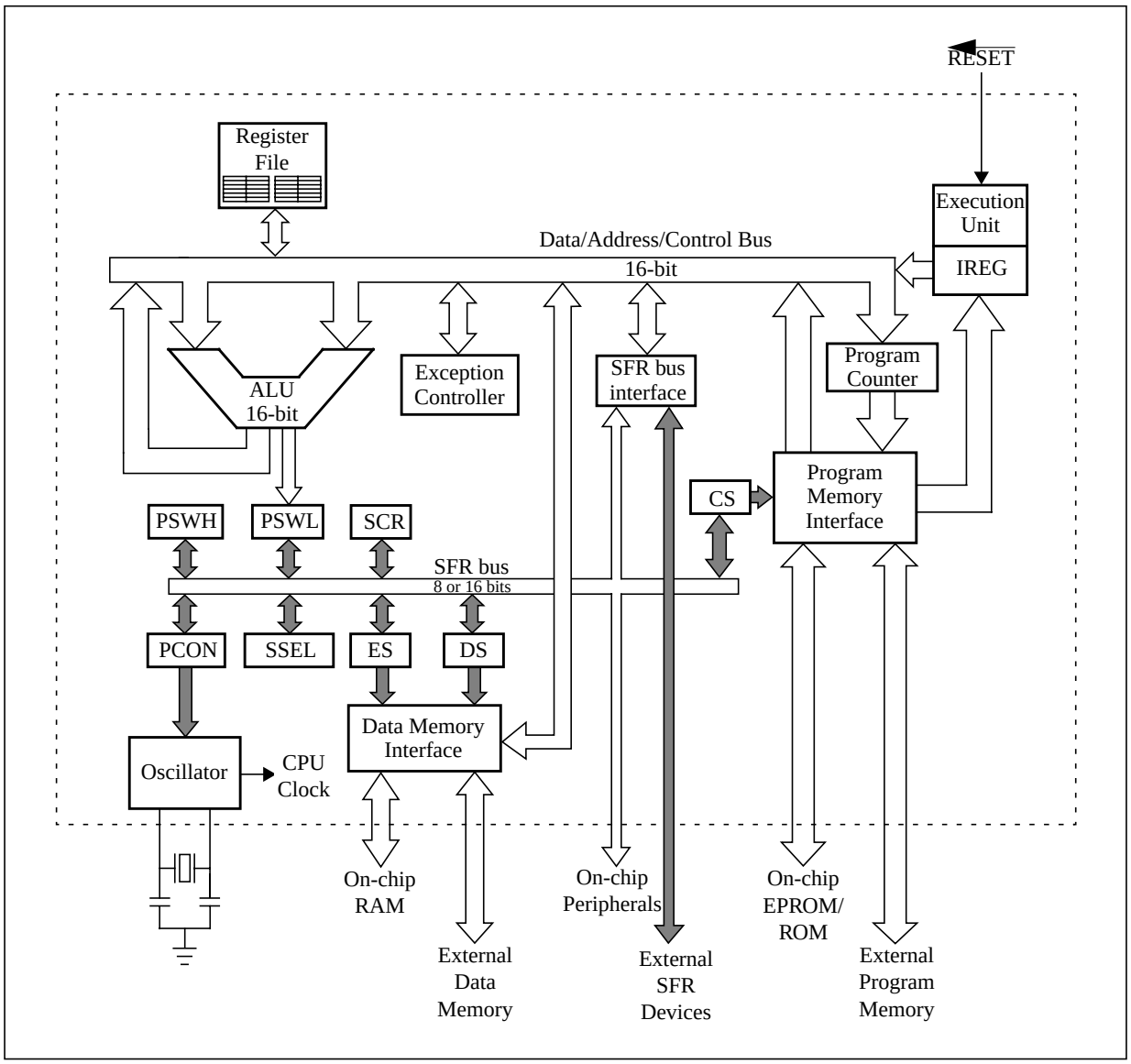


Figure 2.6 The XA Architecture

2.3.1 CPU Blocks

The XA processor is composed of several functional blocks: Instruction fetch and decode; Execution unit; ALU; Exception controller; Interrupt controller; Register File and core registers; Program memory (ROM or EPROM), Data memory (RAM); SFR and external bus interface; Oscillator; and on-chip peripherals and I/O ports.

Certain functional blocks that exist on most XA derivatives are not part of the CPU core and may vary in each derivative. These are: the external bus interface, the Special Function Register bus (SFR bus) interface, specific peripherals, I/O ports, code and data memories, and the interrupt controller.

CPU Performance Features

The XA core is partially pipelined and performs some CPU functions in parallel. For instance, instruction fetch and decode, and in some cases data write-back, are done in parallel with instruction execution. This partial pipelining gives very fast instruction execution at a very low cost. For instance, the instruction execution time for most register-to-register operations on the XA is 3 CPU clocks, or 100 nanoseconds with a 30 MHz oscillator.

ALU

Data operations in the XA core are accomplished with a 16-bit ALU, providing both 8-bit and 16-bit functions. Special circuitry has been included to allow some 32-bit functions, such as shifts, multiply, and divide.

Core Registers

The XA core includes several key Special Function Registers which are accessed by programs.

The System Configuration Register (SCR) sets up the basic operating modes of the XA. The Program Status Word (PSW) contains status flags that show the result of ALU operations, the register select bits for the four register file banks, the interrupt mask bit, and other system flags. The Data Segment (DS), Extra Segment (ES), and Code Segment (CS) registers contain the segment numbers of active data memory segments. The Segment Select register (SSEL), contains bits that determine which segment register is used by each pointer register in the register file. Bits in the Power Control register (PCON) control the reduced power modes of the processor.

Execution and Control

The Execution and Control block fetches instructions from the code memory and decodes the instructions prior to execution. The XA normally attempts to fetch instructions from the code memory ahead of what is immediately needed by the execution unit. These pre-fetched instructions are stored in a 7 byte queue contained in the fetch and decode unit.

If the fetch unit has instructions in the queue, the execution unit will not have to wait for a fetch to occur when it is ready to begin execution of a new instruction. If a program branch is taken, the queue is flushed and instructions are fetched from the new location. This block also decides whether to attempt instruction fetches from on or off-chip code memory.

The instruction at the head of the queue is decoded into separate functional fields that tell the other CPU blocks what to do when the instruction is executed. These fields are stored in staging registers that hold the information until the next instruction begins executing.

Execution Unit

The execution unit controls many of the other CPU blocks during instruction execution. It routes addressing information, sends read and write commands to the register file and memory control blocks, tells the fetch and decode unit when to branch, controls the stack, and ensures that all of these operations are performed in the proper sequence. The execution unit obtains control information for each instruction from a microcode ROM.

Interrupt Controller

The interrupt controller can receive an interrupt request from any of the sources on a particular XA derivative. It prioritizes these based on user programmable registers containing a priority for each interrupt source. It then compares the priority of the highest pending interrupt (if any) to the interrupt mask bits from the PSW. If the interrupt has a higher priority than the currently running code, the interrupt controller issues a request to the execution unit.

The interrupt controller also contains extra registers for processing software interrupts. These are used to run non-critical portions of interrupt service routines at a decreased priority without risking “priority inversion.”

While the interrupt controller is not part of the XA core, it is present in some form on all XA derivatives.

Exception Controller

The exception controller is similar to the interrupt controller except that it processes CPU exceptions rather than hardware and software interrupt requests. Sources of exceptions are: stack overflow; divide by zero; user execution of an RETI instruction; hardware breakpoint; trace mode; and non-maskable interrupt (NMI).

Exceptions are serviced according to a fixed priority ranking. Generally, exceptions must be serviced immediately since each represents some important event or problem that must be dealt with before normal operation can resume.

The Exception Controller is part of the XA core and is always present.

Interrupt and Exception Processing

Interrupt and exception processing both make use of a vector table that resides in the low addresses of the code memory. Each interrupt and exception has an entry in the vector table that includes the starting address of the service routine and a new PSW value to be used at the beginning of the service routine. The starting address of a service routine must be within the first 64K of code memory.

When the XA services an exception or interrupt, it first saves the return address on the stack, followed by the PSW contents. Next, the PC and the PSW are loaded with the starting address of the appropriate service routine and the new PSW contents, respectively, from the vector table.

When the service routine completes, it returns to the interrupted code by executing the RETI (return from interrupt) instruction. This instruction loads first the PSW and then the Program Counter from the stack, resuming operation at the point of interruption. If more than the PC and PSW are used by the service routine, it is up to that routine to save and restore those registers or other portions of the machine state, normally by using the stack, and often by switching register banks.

Reset

Power up reset and any other external reset of the XA is accomplished via an active low reset pin. A simple resistor and capacitor reset circuit is typically used to provide the power-on reset pulse. the reset pin is a Schmitt trigger input, in order to prevent noise on the reset pin from causing spurious or incomplete resets.

The XA may be reset under program control by executing the RESET instruction. This instruction has the effect of resetting the processor as if an external reset occurred, except that some hardware features that are latched following a hardware reset (such as the state of the EA pin and bus width programming) are not re-latched by a software reset. This distinction is necessary because external circuitry driving those inputs cannot determine that a reset is in progress.

Some XA derivatives also have a hardware watchdog timer peripheral that will trigger an equivalent chip reset if it is allowed to time out.

Oscillator and Power Saving Modes

XA derivatives have an on-chip oscillator that may be used with crystals or ceramic resonators to provide a clock source for the processor.

The XA supports two power saving modes of operation: Idle mode and Power Down mode. Either mode is activated by setting a bit in the Power Control (PCON) register. The Idle mode shuts down all processor functions, but leaves most of the on-chip peripherals and the external interrupts functioning. The oscillator continues to run. An interrupt from any operating source will cause the XA to resume operation where it left off.

The Power Down mode goes one step further and shuts down everything, including the on-chip oscillator. This reduces power consumption to a tiny amount of CMOS leakage plus whatever loads are placed on chip pins. Resuming operation from the power down mode requires the oscillator to be restarted, which takes about 10 milliseconds. Power down mode can be terminated either by resetting the XA or by asserting one of the external interrupts, if one was left enabled when power down mode was entered. In Power Down mode, data in on-board RAM is retained. Further power savings may be made by reducing Vdd in Power Down mode; see the device data sheet for details.

Stack

The processor stack provides a means to store interrupt and subroutine return addresses, as well as temporary data. The XA includes 2 stack pointers, the System Stack Pointer (SSP) and the User Stack Pointer (USP), which correspond to 2 different stacks: the system stack and the user stack. See Figure 2.7. The system stack always resides in the first data memory segment,

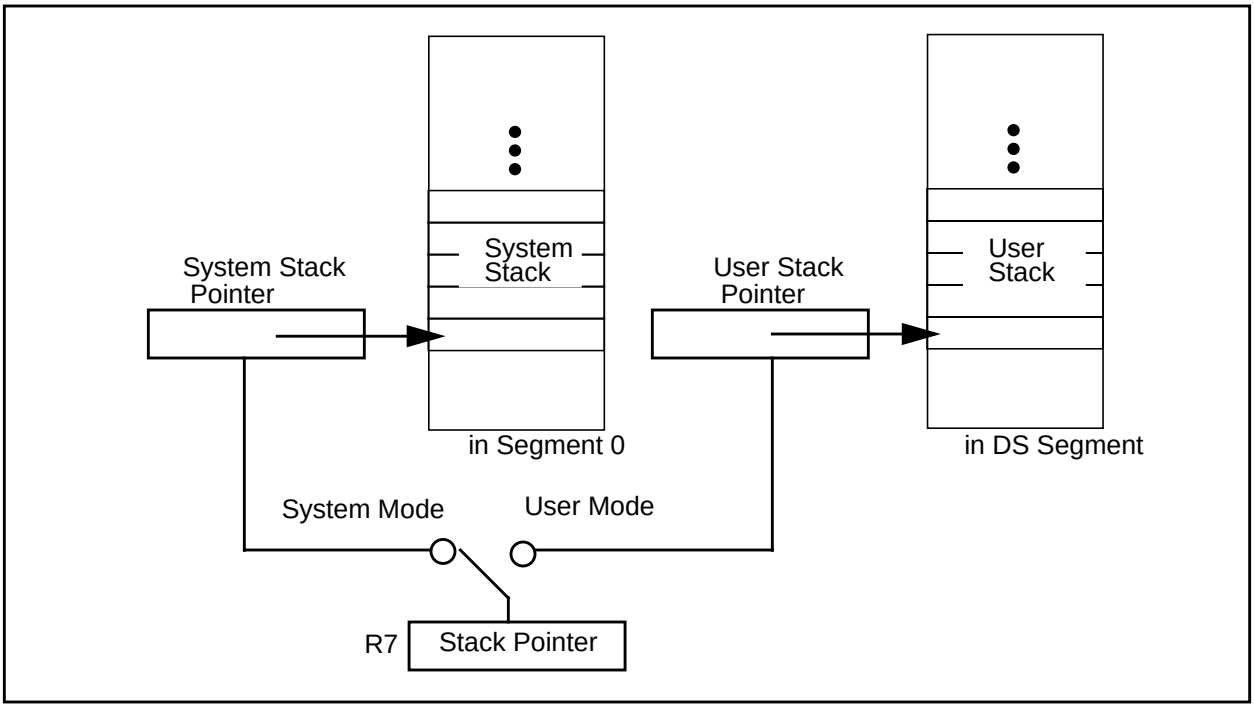


Figure 2.7 XA Stacks

segment 0. The user stack resides in the data memory segment identified by the current value of the data segment (DS) register. Executing code has access to only one of these stacks at a time, via the Stack Pointer, R7. Since each stack resides in a single data memory segment, its maximum size is 64K bytes. The purpose of having two stack pointers will be discussed in more detail in the section on Task Management below.

The XA stack grows downwards, from higher addresses to lower addresses within data memory. The current stack pointer always points to the last item pushed on the stack, unless the stack is empty. Prior to a push operation, the stack pointer is decremented by 2, then data is written to memory. When the stack is popped, the reverse procedure is used. First, data is read from memory, then the stack pointer is incremented by 2. Data on the stack always occupies an even number of bytes and is word aligned in data memory.

Debugging Features

The XA incorporates some special features designed to aid in program and system debugging. There is a software breakpoint instruction that may be inserted in a user's program by a debugger program, causing the user program to break at that point and go to the breakpoint service routine, which can transmit the CPU state so that it can be viewed by the user.

The trace mode is similar to a breakpoint, but is forced by hardware in the XA after the execution of every instruction. The trace service routine can then keep track of every instruction executed by a user program and transmit information about the CPU state to a serial port or other peripheral for display or storage. Trace mode is controlled by a bit in the PSW. The XA is able to alter the trace mode bit whenever an interrupt or exception vector is taken. This gives very flexible use of trace mode, for instance by allowing all interrupts to run at full speed to comply with system hardware requirements, while single stepping through mainline code.

With these two features, a simple monitor debugger routine can allow a user to single step through a program, or to run a program at full speed, stopping only when execution reaches a breakpoint, in either case viewing the CPU state before continuing.

2.4 Task Management

Several features of the XA have been included to facilitate multi-tasking. Multi-tasking can be thought of as running several programs at once on the same processor, with a supervisory program determining when each program, or task, runs, and for how long. Since each task shares the same CPU, the system resources required by each must be kept separate and the CPU state restored when switching execution from one task to another. The problem is much simpler for a microcontroller than it is for a microprocessor, because the code executed by a microcontroller always comes from the same source: the designers of the system it runs on. Thus, this code can be considered to be basically trustworthy and extreme measures to prevent misbehavior are not necessary. The emphasis in the XA design is to protect against simple accidents.

The first step in supporting multi-tasking is to provide two execution contexts, one for the basic tasks –on the XA termed “user mode”– and one for the supervisory program –"system mode.". A program running in system mode has access to all of the processor’s resources and can set up and launch tasks.

Code running in system and user mode use different stack pointers, the System Stack Pointer (SSP) and the User Stack Pointer (USP) respectively. The system stack is always located in the first 64K data memory segment, where it can take advantage of the fast on-chip RAM. The user stack is located within each task’s local data segment, identified by the DS register. The fact that user mode code uses a different stack than system mode code prevents tasks from accidentally destroying data on the system stack and in other task spaces.

Additional protection mechanisms are provided in the form of control bits and registers that are only writable by system mode code. For instance the DS register, that identifies the local data segment for user mode code, is only writable in the system mode. While tasks can still write to the other segment register, the ES register, they cannot write to memory via the ES register unless specifically allowed to do so by the system. The data memory segmentation scheme thus prevents tasks from accessing data memory in unpredictable ways.

Other protected features include enabling of the Trace Mode and alteration of the Interrupt Mask.

The 4 register banks are a feature that can be useful in small multi-tasking systems by using each bank for a different task, including one for system code. This means less CPU state that must be saved during task switching.

2.5 Instruction Set

The XA instruction set is designed to support common control applications. The instruction encoding is optimized for the most commonly used instructions: register to register or register with indirect arithmetic and logic operations; and short conditional and unconditional branches. These instructions are all encoded as 2 bytes. The bulk of XA instructions are encoded as either 2 or 3 bytes, although there are a few 1 byte instructions as well as 4, 5, and 6 byte instructions.

The execution of instructions normally overlaps instruction fetch, and sometimes write-back operations, in order to further speed processing.

2.5.1 Instruction Syntax

The instruction syntax chosen for the XA is similar in many ways to that of the 80C51. A typical XA instruction has a basic mnemonic, such as "ADD", followed by the operands that the operation is to be performed on. The basic syntax is illustrated in Figure 2.8. The direction of operation flow is determined by the order in which operands occur in the source line. For instance, the instruction: "ADD R1, R2" would cause the contents of R1 and R2 to be added together and the result stored in R1. Since R1 and R2 are word registers in the XA, this is a 16-bit operation.

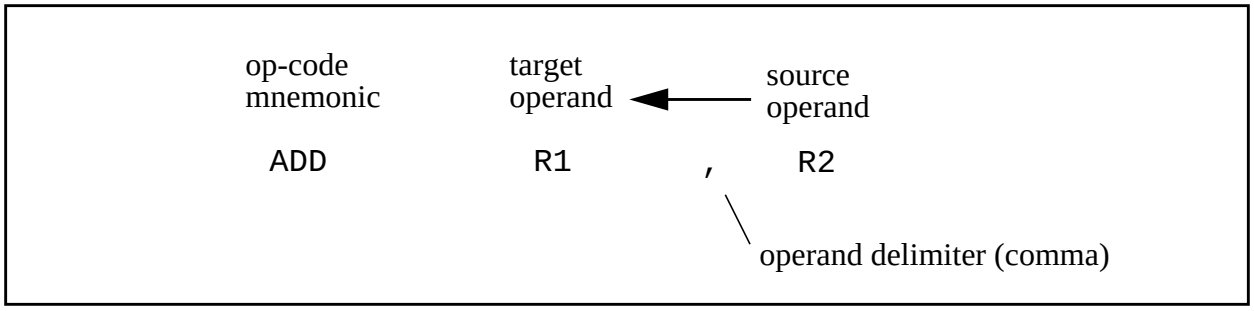


Figure 2.8 Basic Instruction Syntax

An indirect reference (a reference to data memory using the contents of a register as an address) is specified by enclosing the operand in square brackets, as in: "ADD R1, [R2]". See Figure 2.9. This instruction causes the contents of R1 and the data memory location pointed to by R2 (appended to its associated segment register) to be added together and the result stored in R1. Reversing the operand order ("ADD [R2], R1") causes the result to be stored in data memory, as shown in Figure 2.10.

Most instructions support an additional feature called auto-increment that causes the register used to supply the indirect memory address to be automatically incremented after the memory access takes place. The source line for such an operation is written as follows: "ADD R1, [R2+]". As illustrated in Figure 2.11, the auto-increment amount always matches the data size used in the instruction. In the previous example, R2 will have 2 added to it because this was a word operation.

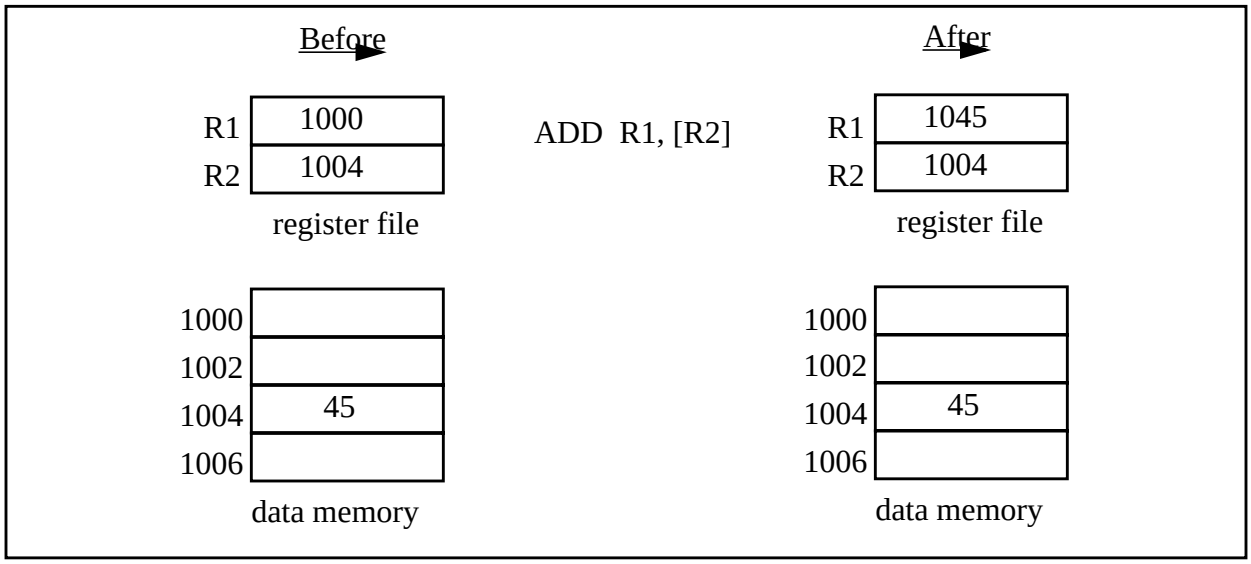


Figure 2.9 Basic Indirect Addressing Syntax, to register

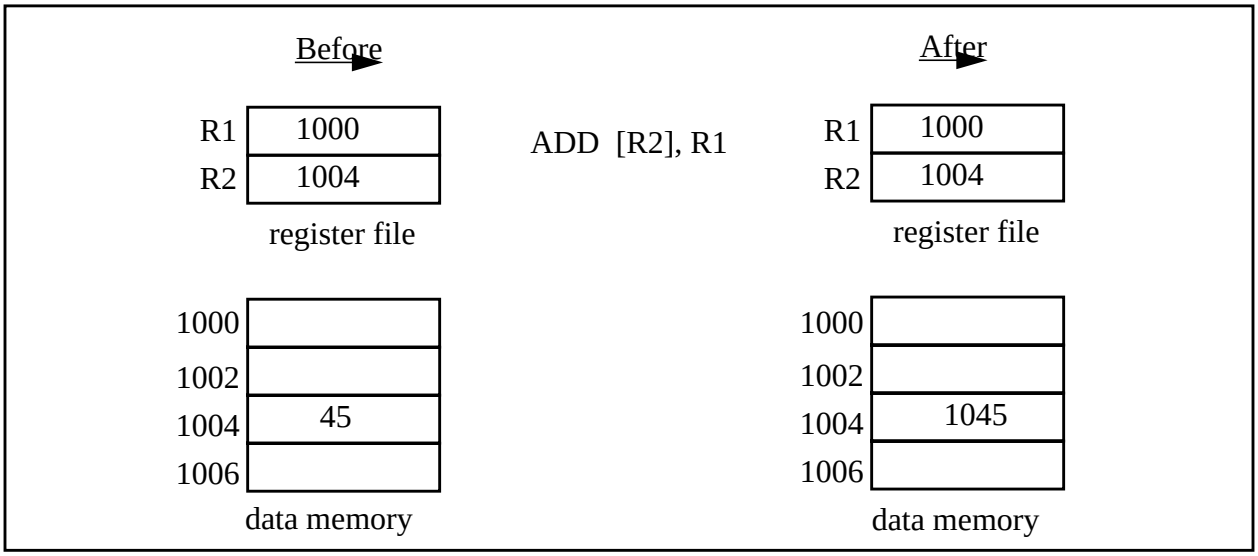


Figure 2.10 Basic Indirect Addressing Syntax, from Register

Another version of indirect addressing is called indirect with offset mode. In this version, an immediate value from the instruction word is added to the contents of the indirect register in order to form the actual address. This result of the add is 16 bits in size, which is then appended to the segment register for that pointer register. If the offset calculation overflows 16 bits, the overflow is ignored, so the indirect reference always remains on the same segment. The immediate data from the instruction is a signed 8-bit or 16-bit offset. Thus, the range is +127 bytes to -128 bytes for an 8-bit offset, and +32,767 to -32,768 bytes for a 16-bit offset. Note that since the address calculation is limited to 16-bits, the 16-bit offset mode allows access to an entire data segment.

When an instruction requires an immediate data value (a value stored within the instruction itself), it is written using the "#" symbol. For example: "ADD R1, #12" says to add the value 12 to register R1.

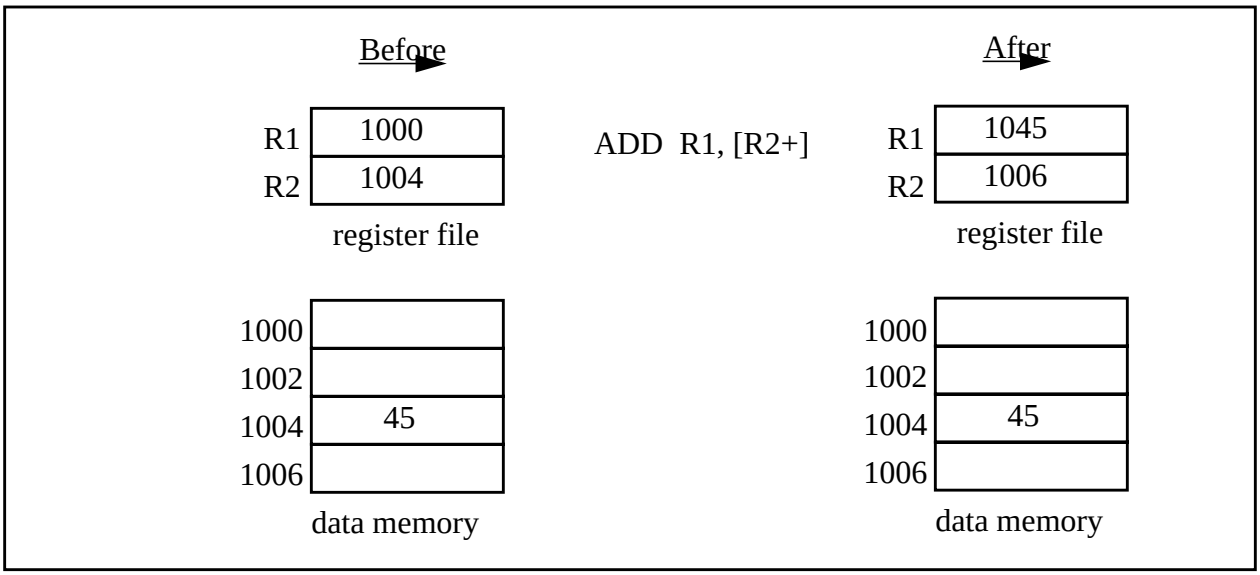


Figure 2.11 Indirect Addressing with Auto-Increment

Since indirect memory references and immediate data values do not implicitly identify the size of the operation to be performed, a few XA instructions must have an operation size explicitly called out. An example would be the instruction: "MOV [R1], #1". The immediate data value does not specify the operation size, and the value stored in memory at the location pointed to by R1 could be either a byte or a word. To clarify the intent of such an instruction, a size identifier is added to the mnemonic as follows: "MOV.b [R1], #1". This tells us that the operation should be performed on a byte. If the line read "MOV.w [R1], #1", it would be a word operation.

If a direct data address is used in an instruction, the address is simply written into the instruction: "ADD 123, R1", meaning to add the contents of register R1 to the data memory value stored at direct address 123. In an actual program, the direct data address could be given a name to make the program more readable, such as "ADD Count, R1".

Operations using Special Function Registers (SFRs) are written in a way similar to direct addresses, except that they are normally called out by their names only: "MOV PSW,#12". Using actual SFR addresses rather than their names in instructions makes the code both harder to read and less transportable between XA derivatives.

Bit addresses within instructions may be specified in one of several ways. A bit may be given a unique name, or it may be referred to by its position within some larger register or entity. An example of a bit name would be one of the status flags in the PSW, for instance the carry ("C") flag. To clear the carry flag, the following instruction could be used: "CLR C". The same bit could be addressed by its position within the PSW as follows: "CLR PSWL.7", where the period (".") character indicates that this is a bit reference. A program may use its own names to identify bits that are defined as part of the application program.

Finally, code addresses are written within instructions either by name or by value. Again, a program is more readable and easier to modify if addresses are called out by name. Examples are: "JMP Loop" and "JMP 124".

2.5.2 Instruction Set Summary

The following pages give a summary of the XA instruction set. For full details, consult Chapter 6.

Basic Arithmetic, Logic, and Data Movement Instructions

The most used operations in most programs are likely to be the basic arithmetic and logic instructions, plus the MOV (move data) instruction. The XA supports the following basic operations:

ADD	Simple addition.
ADDC	Add with carry.
SUB	Subtract.
SUBB	Subtract with borrow.
CMP	Compare.
AND	Logical AND.
OR	Logical OR.
XOR	Exclusive-OR.

These instructions support all of the following standard XA data addressing mode combinations::

Operands	Description
R, R	The source and destination operands are both registers.
R, [R]	The source operand is indirect, the destination operand is a register.
[R], R	The source operand is a register, the destination operand is indirect.
R, [R+]	The source operand is indirect with auto-increment, the destination operand is a register.
[R+], R	The source operand is a register, the destination operand is indirect with auto-increment.
R, [R+offset]	The source operand is indirect with an 8 or 16-bit offset, the destination operand is a register.
[R+offset], R	The source operand is a register, the destination operand is indirect with an 8 or 16-bit offset.
direct, R	The source operand is a register, the destination operand is a direct address.
R, direct	The source operand is a direct address, the destination operand is a register.
R, #data	The source operand is an 8 or 16-bit immediate value, the destination operand is a register.
[R], #data	The source operand is an 8 or 16-bit immediate value, the destination operand is indirect.

Operands	Description
[R+], #data	The source operand is an 8 or 16-bit immediate value, the destination operand is indirect with auto-increment.
[R+offset], #data	The source operand is an 8 or 16-bit immediate value, the destination operand is indirect with an 8 or 16-bit offset.
direct, #data	The source operand is an 8 or 16 bit immediate value, the destination operand is a direct address.

Other instructions on the XA use different operand combinations. All XA instructions are covered in detail in the Instruction Set section. Following is a summary of other instruction types:Additional arithmetic instructions

Additional arithmetic instructions

ADDS	Add short immediate (4-bit signed value).
NEG	Negate (twos complement).
SEXT	Sign extend.
MUL	Multiply.
DIV	Divide.
DA	Decimal adjust.
ASL	Arithmetic shift left.
ASR	Arithmetic shift right.
LEA	Load effective address.

Additional logic instructions

CPL	Complement (ones complement or logical inverse).
LSR	Logical shift right.
NORM	Normalize.
RL	Rotate left.
RLC	Rotate left through carry.
RR	Rotate right.
RRC	Rotate right through carry.

Other data movement instructions

MOVS	Move short immediate (4-bit signed value).
MOVC	Move to or from code memory.
MOVX	Move to or from external data memory.
PUSH	Push data onto the stack.
POP	Pop data from the stack.
XCH	Exchange data in two locations.

Bit manipulation instructions

SETB	Set (write a 1 to) a bit.
CLR	Clear (write a 0 to) a bit.
MOV	Move a bit to or from the carry flag.
ANL	Logical AND a bit (or its inverse) to the carry flag.
ORL	Logical OR a bit (or its inverse) to the carry flag.

Jump, branch, and call instructions

BR	Branch to code address (plus or minus 256 byte range).
JMP	Jump to code address (range depends on specific JMP variation).
CALL	Call subroutine (range depends on specific CALL variation).
RET	Return from subroutine or interrupt.
Bcc	Conditional branches with 15 possible condition variations.
JB, JNB	Jump if a bit set or not set.
CJNE	Compare two operands and jump if they not equal.
DJNZ	Decrement and jump if the result is not zero.
JZ, JNZ	Jump on zero or not zero (included for 80C51 compatibility).

Other instructions

NOP	No operation (used mainly to align branch targets).
BKPT	Breakpoint (used for debugging).
TRAP	Software trap (used to call system services in a multitasking system).
RESET	Reset the entire chip.

2.6 External Bus

Most XA derivatives have the capability of accessing external code and/or data memory through the use of an external bus. The external bus provides address information to external devices, and initiates code read, data read, or data write strobes. The standard XA external bus is designed to provide flexibility, simplicity of connection, and optimization for external code fetches.

As described in section 4.4.4, the initial external bus width is hardware settable, and the XA determines its value (8 or 16 bits) during the reset sequence.

2.6.1 External Bus Signals

The standard XA external bus supports 8 or 16-bit data transfers and up to 24 address lines. The precise number of address lines varies by derivative. The standard control signals and their functions for the external bus are as follows:

Signal name	Function
ALE	Address Latch Enable. This signal directs an external address latch to store a portion of the address for the next bus operation. This may be a data address or a code address.
$\overline{\text{PSEN}}$	Program Store Enable. Indicates that the XA is reading code information over the bus. Typically connected to the Output Enable pin of external EPROMs.
$\overline{\text{RD}}$	Read. The external data read strobe. Typically connected to the $\overline{\text{RD}}$ pin of external peripheral devices.
$\overline{\text{WRL}}$	Write. The low byte write strobe for external data. Typically connected to the $\overline{\text{WR}}$ pin of external peripheral devices. For an 8-bit data bus, this is the only write strobe. For a 16-bit data bus, this strobe applies only to the lower data byte.
$\overline{\text{WRH}}$	Write High. This is the upper byte write strobe for external data when using a 16-bit data bus.
WAIT	Wait. Allows slowing down any type external bus cycle. When asserted during a bus operation, that operation waits for this signal to be de-asserted before it is completed.

2.6.2 Bus Configuration

The standard XA bus is user configurable in several ways. First, the bus size may be configured to either 8 bits or 16 bits. This may be configured by the logic level on a pin at reset, or under firmware control (if code is initially executed from on-chip code memory) prior to any actual external bus operations. As on the 80C51, the $\overline{\text{EA}}$ pin determines whether or not on-chip code memory is used for initial code fetches.

Second, the number of address lines may be configured in order to make optimal use of I/O ports. Since external bus functions are typically shared with I/O ports and/or peripheral I/O functions, it is advantageous to set the number of address lines to only what is needed for a particular application, freeing I/O pins for other uses.

2.6.3 Bus Timing

The standard XA bus also provides a high degree of bus timing configurability. There are separate controls for ALE width, $\overline{\text{PSEN}}$ width, $\overline{\text{RD}}$ and $\overline{\text{WRL}}$/$\overline{\text{WRH}}$ width, and data hold time from $\overline{\text{WRL}}$/$\overline{\text{WRH}}$. These times are programmable in a range that will support most RAMs, ROMs, EPROMs, and peripheral devices over a wide range of oscillator frequencies without the need for additional external latches, buffers, or WAIT state generators.

The following figures show the basic sequence of events and timing of typical XA bus accesses. For more detailed information, consult Section 7 and the device data sheet.

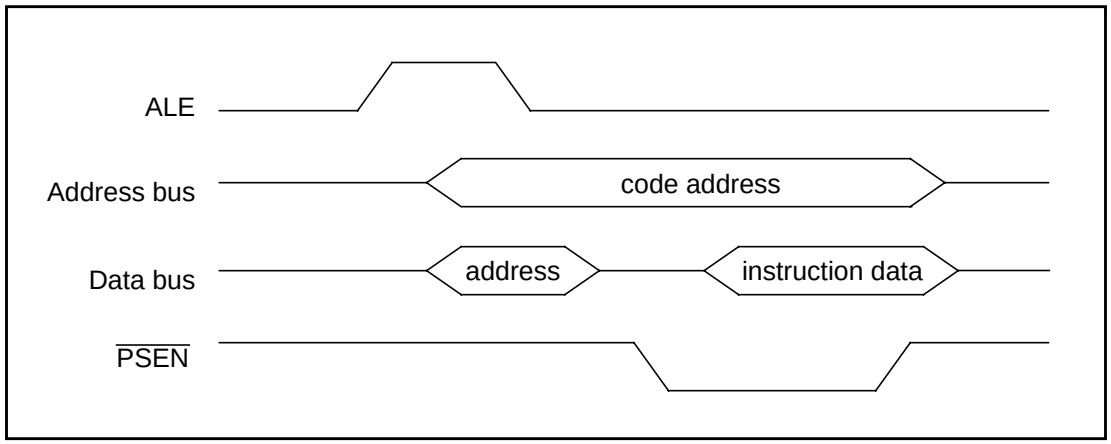


Figure 2.12 Typical External Code Read.

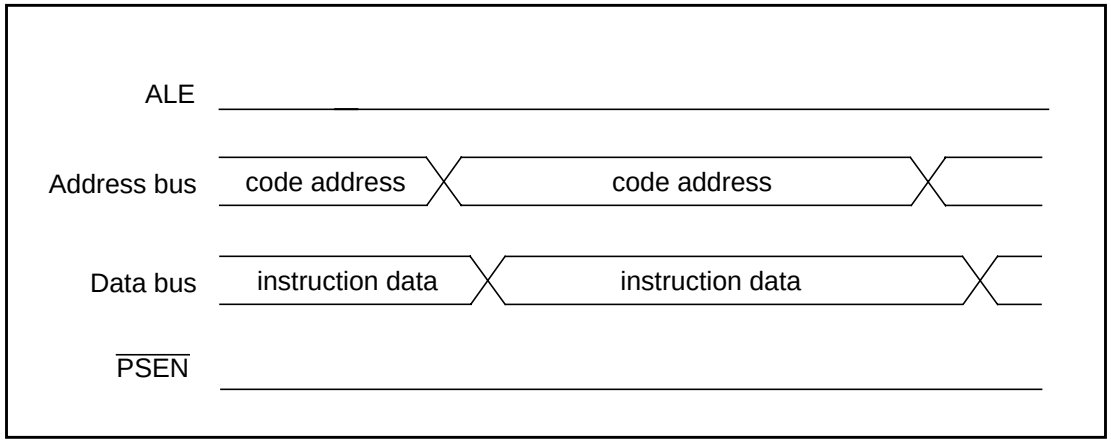


Figure 2.13 Optimized (Sequential Burst) External Code Read.

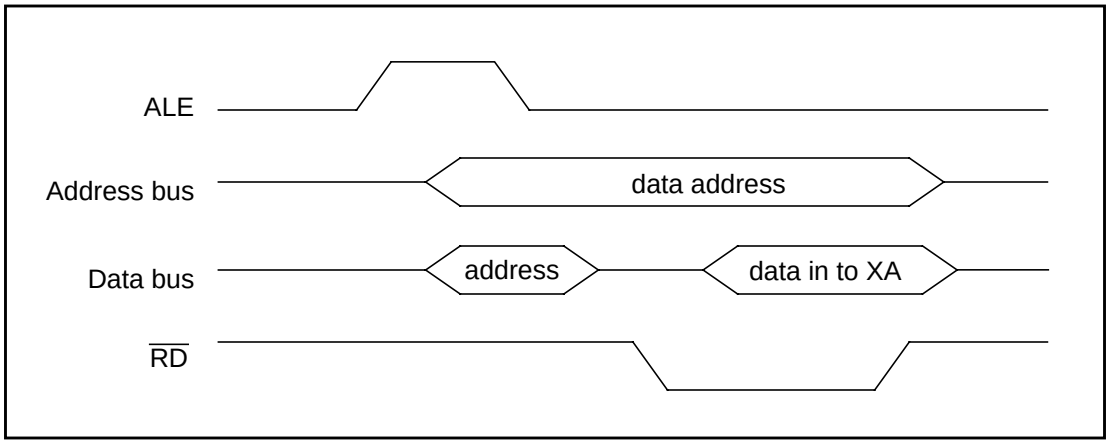


Figure 2.14 Typical External Data Read.

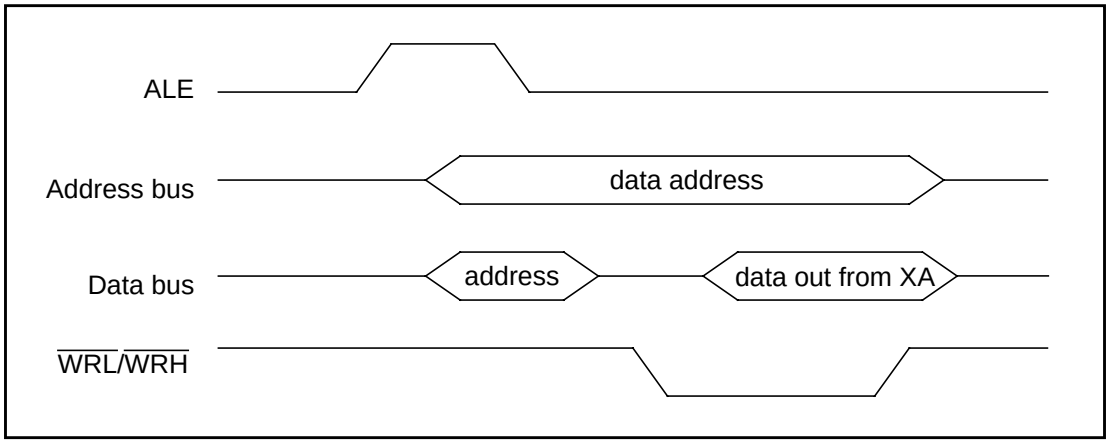


Figure 2.15 Typical External Data Write.

2.7 Ports

Standard I/O ports on the XA have been enhanced to provide better versatility and programmability than was previously available in the 80C51 and most of its derivatives. Access to the I/O ports from a program is through SFR addresses assigned to those ports. Ports may be read and written is the same manner as any other SFR.

The XA provides more flexibility in the use of I/O ports by allowing different output configurations. See Figure 2.16. Port outputs may be programmed to be quasi-bidirectional (80C51 style ports), open drain, push-pull, and high impedance (input only).

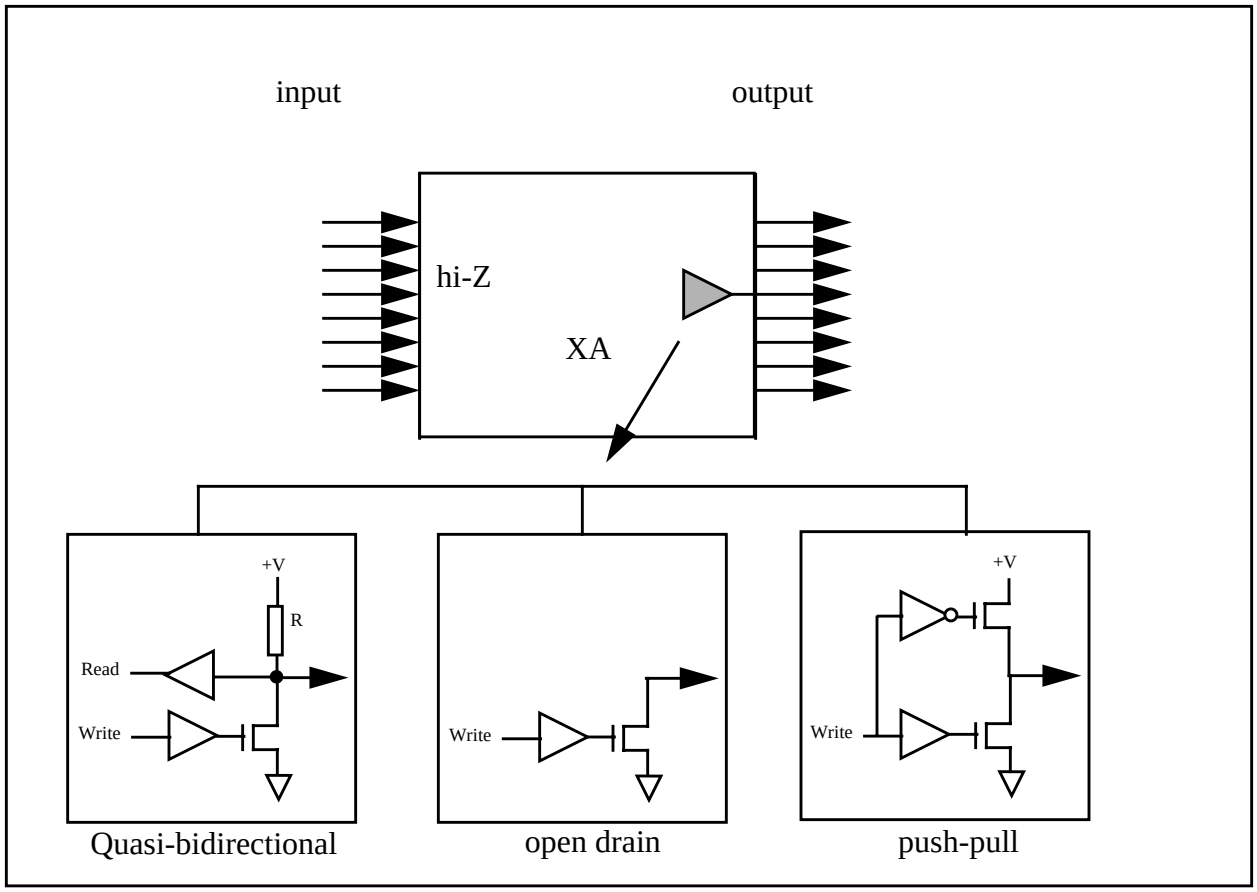


Figure 2.16 XA Port Pins with Driver Option Detail

2.8 Peripherals

The XA CPU core is designed to make derivative design fast and easy. Peripheral devices are not part of the core, but are attached by means of a Special Function Register bus, called the SFR bus, which is distinct from the CPU internal buses. So, a new XA derivative may be made by designing a new SFR bus compatible peripheral function block, if one does not already exist, then attaching it to the XA core.

2.9 80C51 Compatibility

The 80C51 is the most extensively designed-in 8-bit microcontroller architecture in the world, and a vast amount of public and private code exists for this device family. For customers who use the 80C51 or one of its derivatives, preservation of their investment in code development is an important consideration. By permitting simple translation of source code, the XA allows existing 80C51 code to be re-used with this higher-performance 16-bit controller. At the same time, the XA hardware was designed with the clear goal of upward compatibility. 80C51 designs may be migrated to the XA with very few changes necessary to software source or hardware.

The XA provides an 80C51 Compatibility Mode, which essentially replicates the 80C51 register architecture for the best possible upward compatibility. In the alternative Native Mode, the XA operates as an optimized 16-bit microcontroller incorporating the best conceptual features of the original 80C51 architecture.

Many trade-offs and considerations were taken into account in the creation of the XA architecture. The most important goal was to make it possible for a software translator to convert 80C51 assembler source code to XA source code on a 1:1 basis, i.e., one XA instruction for one 80C51 instruction.

Some specific compatibility issues are summarized in the following two sections. See Chapter 9 for a complete description of compatibility.

2.9.1 Software Compatibility

Several basic goals were observed in order to design 80C51 software compatibility for the XA, while avoiding over-complicating the XA design. Following are some key issues for XA software:

• Instruction mapping. Each 80C51 instruction translates into one XA instruction. Multi-instruction combinations that could result in problems if split by an interrupt were avoided as much as possible. Only one 80C51 instruction does not have a one-to-one direct replacement with an XA instruction (this instruction, XCHD, is extremely rarely used).

• "As-is" instructions. Most XA instructions are more powerful supersets of 80C51 instructions. Where this was not possible, the original 80C51 instruction is included "as-is" in the XA instruction set.

• Timing. Instruction timing must necessarily change in order to improve performance. The XA does not attempt to retain timing compatibility with the 80C51; rather, the design simply maximizes instruction execution speed. When 80C51 code that is timing critical is translated to the XA, the user must re-analyze the timing and make adjustments.

• SFR Access. Translation of SFR accesses is usually simple, since SFRs are normally referenced by name. Such references are simply retained in the translated XA code. If program source code from a specific 80C51 derivative references an SFR by its address, the translator can directly substitute the appropriate XA SFR, provided both the 80C51 and the XA derivative are correctly identified to the translator.

2.9.2 Hardware Compatibility

The key goal for hardware was to produce a familiar architecture with a good deal of upward compatibility.

• Memory Map. A major consideration in hardware compatibility of the XA with the 80C51 is the memory map. The XA approaches this issue by having each memory area (registers, data memory, code memory, stack, SFRs) be a superset of the corresponding 80C51 area.

• Stack. One area where a functional change could not be avoided is in the use of the processor stack. Due to the fact that the XA supports 16-bit operations in memory, it was necessary to change the direction of stack growth to downward –the standard for 16-bit processors– in order to match stack usage with efficient access of 16-bit variables in memory. This is an important consideration for support of high-level language compilers such as C.

• Pin-for-pin compatibility. XA derivatives are not intended to be exactly pin-compatible with other 80C51 derivatives that have similar features. Many on-chip XA peripherals, for example, have improved capabilities, and maintaining pin-for-pin compatibility would limit access to these capabilities. In general, peripherals have been made upward compatible with the original 80C51 devices, and most enhancements are added transparently. In these cases, 80C51 code will operate correctly on the 80C51 functional subset.

• Bus Interface. The external bus on the XA is an example of a trade-off between 80C51 compatibility and performance. In order to provide more flexibility and maximum performance, the 80C51 bus had to be changed somewhat. The differences are described in detail in the section on the external bus.

3 XA Memory Organization

3.1 Introduction

The memory space of XA is configured in a Harvard architecture which means that code and data memory (including sfrs) are organized in separate address spaces. The XA architecture supports 16 Megabytes (24-bit address) of both code and data space. The size and type of memory are specific to an XA derivative.

The XA supports different types of both code and data memory e.g.,code memory could be Eprom, EEProm, OTP ROM, Flash, and Masked ROM whereas data memory could be RAM, EEProm or Flash.

This chapter describes the XA Memory Organization of register, code, and data spaces; how each of these spaces are accessed, and how the spaces are related.

3.2 The XA Register File

The XA architecture is optimized for arithmetic, logical, and address-computation operations on the contents of one or more registers in the XA Register File.

3.2.1 Register File Overview

The XA architecture defines a total of 16 word registers in the Register File:
In the baseline XA core, only R0 through R7 are implemented. These registers are available for unrestricted use except R7– which is the XA stack pointer, as illustrated in Figure 3.1. In effect, the XA registers provide users with at least 7 distinct “accumulators” which may be used for all operations. As will be seen below, the XA registers are accessible at the bit, byte, word, and doubleword level.

Additional global registers, R8 through R15, are reserved and may be implemented in specific XA derivatives. These registers, when available, are equivalent to R0 through R7 except byte access and use as pointers will not be possible (only word, double-word, and bit-addressable). The Register File is independent of all other XA memory spaces (except in Compatibility Mode; see chapter 9).

Register File Detail

Figure 3.2 describes R0 through R7 in greater detail.

Byte, Word, and Doubleword Registers

All registers are accessible as bits, bytes, words, and –in a few cases– doublewords. Bit access to registers is described in the next section. As for byte and word accesses, R1 –for example– is a word register that can be word referenced simply as “R1”. The more significant byte is labeled as “R1H” and the less significant byte of R1 is referenced as “R1L”. Double-word registers are always formed by adjacent pairs of registers and are used for 32 bit shifts, multiplies, and divides. The pair is referenced by the name of the lower-numbered register (which contains the

less significant word), and this must have an even number. Thus valid double-register pairs are (R0,R1), (R2,R3), (R4,R5) and (R6, R7).

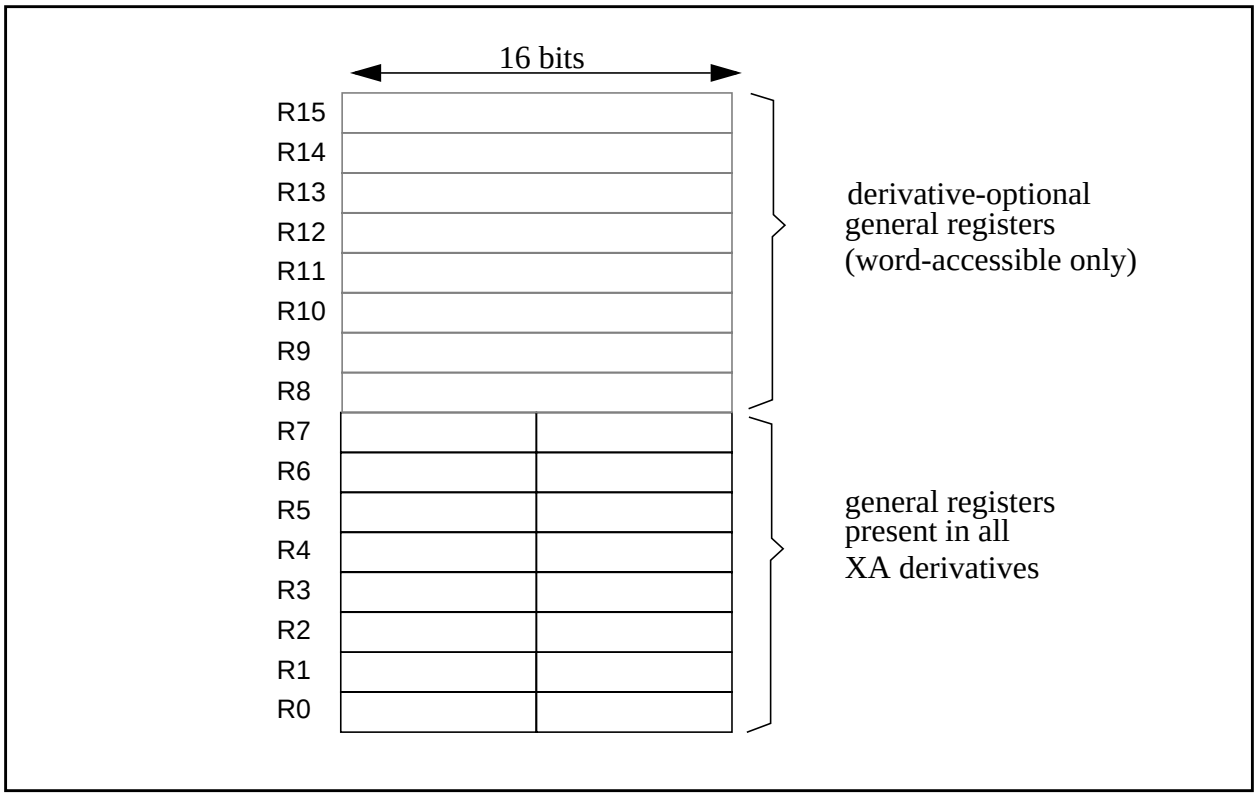


Figure 3.1 XA Register File Overview

As described in section 4.7, there are two stack pointers, one for user mode and another for system mode. At any given instant only one stack pointer is accessible and its value is in R7. When PSW.SM is 0, user mode is active and the USP is accessible via R7. When PSW.SM is 1, the XA is operating in system mode, and SSP is in SP (R7). (Note however, as described in Chapter 4, all interrupts save stack frames on the system stack, using the SSP, regardless of the current operating mode.)

There are four distinct instances of registers R0 through R3. At any given time, only 1 set of the 4 banks is active, referenced as R0 through R3, and the contents of the other banks are inaccessible. This allows high-speed context-switching, for example, for interrupt service routines. **PSW** bits **RS1** and **RS0** select the active register bank:

RS1	RS0	visible register bank
0	0	bank 0
0	1	bank 1
1	0	bank 2
1	1	bank 3

PSW.RSn are writable when the XA is operating in system or user mode, and programs running in either mode may explicitly change these bits to make selected banks visible one at a time. More commonly, the interrupt mechanism, as described in Chapter 4, provides automatic implicit register bank switching so interrupt handlers may immediately begin operating in a reserved register context.

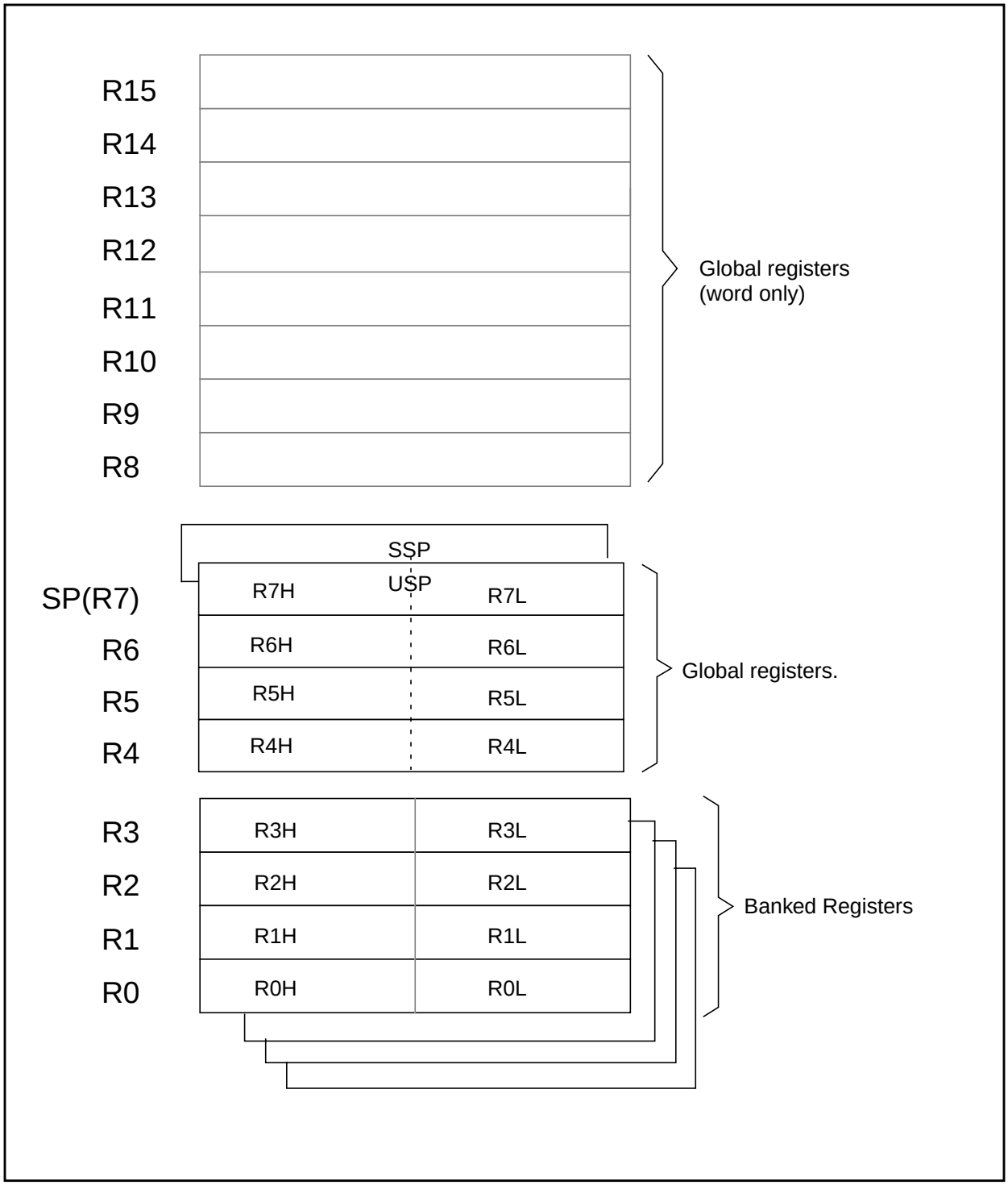


Figure 3.2 XA Register File

Bit Access to Registers

The XA Registers are all bit addressable. Figure 3.3 shows how bit addresses overlie the basic register file map. In general, absolute bit references as given in this map are unnecessary. XA software development tools provide symbolic access to bits in registers. For example, bit 7 may be designated as “R0.7” with no ambiguity

Bit references to banked registers R0 through R3 access the currently accessible register bank, as set by **PSW** bits **RS1**, **RS0** and the currently selected stack pointer USP or SSP. The unselected registers are inaccessible..

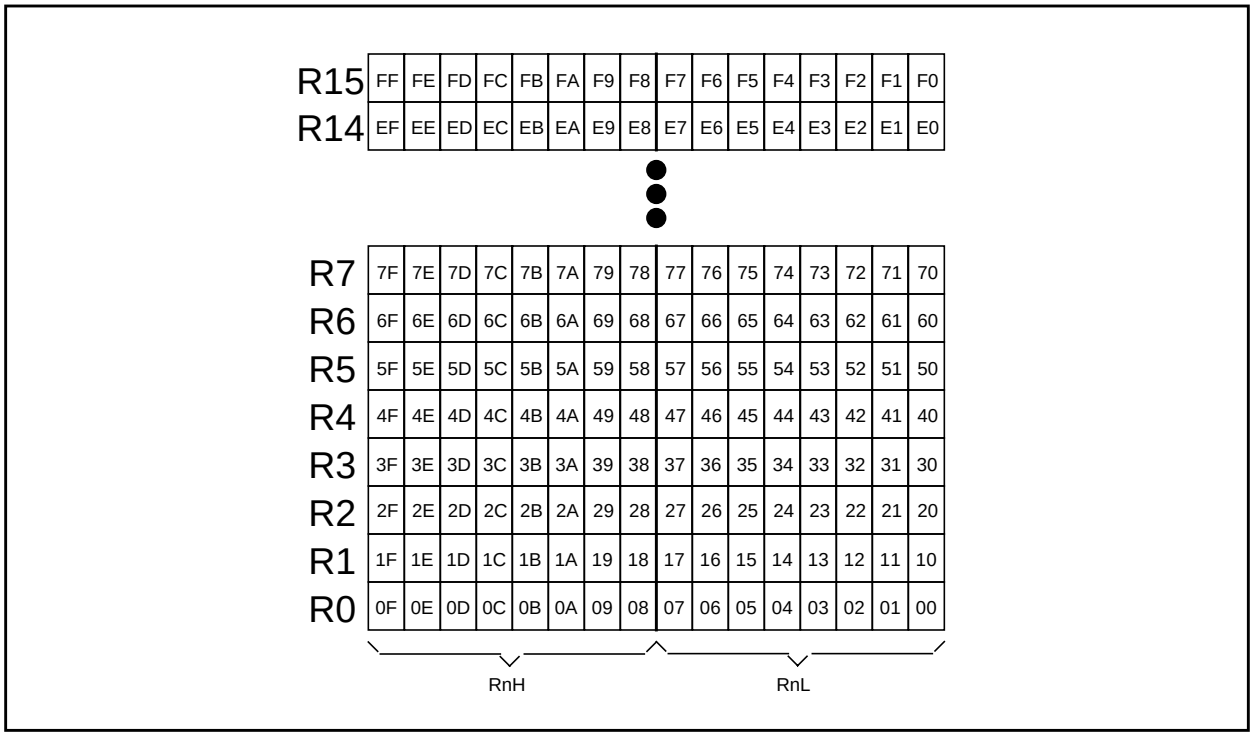


Figure 3.3 Bit Address to Registers

3.3 The XA Memory Spaces

The XA divides physical memory into program and data memory spaces. Twenty-four address bits, corresponding to a 16MB address space, are defined in the XA architecture. In any given XA implementation, fewer than all twenty-four address bits may actually be used, and there is provision for a small-memory mode which uses only 16-bit addresses; see Chapter 4.

Code and data memory may be on-chip or external, depending on the XA variant and the user implementation. Whether a specific region is on-chip or external does not, in general, affect access to the memory.

3.3.1 Bytes, Words, and Alignment

XA memory is addressed in units of *bytes*, where each byte consists of 8 bits. A *word* consists of two bytes, and the word storage order is “Little-Endian”, that is, the less significant byte of word data is located at a lower memory address. See Figure 3.4.

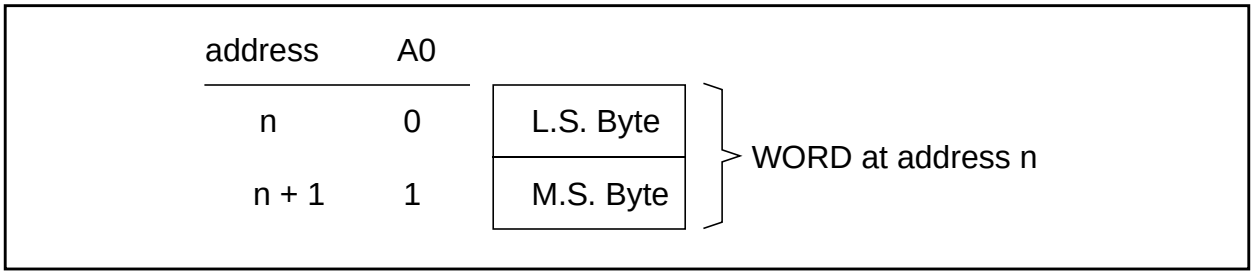


Figure 3.4 Memory byte order

Any word access must be aligned at an even address (Address bit A0=0). If an odd-aligned word access is attempted the word at the next-smallest even address will be accessed, that is, A0 will be set to 0.

The external XA memory spaces may be accessed in byte or word units but the hardware access method does not affect the even alignment restriction on word accesses.

3.4 Data Memory

The data memory space starts at address 0 and extends to the highest valid address in the implementation, at maximum, FFFFFFh. As will be described below, the data memory space is segmented into 256 segments of 64K bytes each. *External Data Memory* starts at the first address following the highest *Internal Data Memory* location. In general, at least 512 bytes of Internal Data Memory, starting at location 0, will be provided in all XA implementations; however, there is no inherent minimum or maximum architectural limitation on Internal Data Memory.

The upper 16 segments of data memory (addresses F0:0000 through FF:FFFF hexadecimal) are reserved for special functions in XA derivatives. A similar range is reserved in the code memory space, see section 3.5.

3.4.1 Alignment in Data Memory

There are no data memory alignment restrictions except that placed on word accesses to all memory: Words must be fetched from even addresses. An attempt to fetch a word at an odd address will fetch a word from the preceding even address.

3.4.2 External and Internal Overlap

If External Data Memory is placed by external logic at addresses that overlaps Internal Data Memory, the Internal Data Memory generally takes precedence. The overlapped portion of the External memory may be accessed only by using a form of the MOVX instruction; see Chapter 6. The use of MOVX always forces external data memory fetch in XA. For non-overlapped portion of external data memory, no MOVX is required.

3.4.3 Use and Read/Write Access

Data memory is defined as read-write, and is intended to contain read/write data. It is logically impossible to execute instructions from XA Data Memory. It is possible, and a common practice, to add logic to overlap external code and data memory spaces. In this case it is important to understand that the memory spaces are logically separate. In such a modified Harvard architecture, implemented with external logic, it is possible –but not recommended– to write self-modifying XA code. No such overlap is possible for internal data memory.

3.4.4 Data Memory Addressing

XA data memory addressing is optimized for the needs of embedded processing. Data memory in the XA is divided into 64K byte segments. This provides an intrinsic protection mechanism for multitasking applications and improves performance by requiring fewer address bits for localized accesses.

Addressing through Segment Registers

Segment registers provide the upper 8 address bits needed to obtain a complete 24-bit address in applications that require full use of the XA 16 Mbyte address space. Two segment registers are defined in the XA architecture for use in accessing data memory, the Data Segment Register (**DS**), and the Extra Segment Register **(ES**). As user stacks are located in the segment specified by **DS**, it is probably most convenient to address user data structures through **ES**. Each pointer register, namely R0 through R6, is associated with one of the segment registers via the Segment Select (**SSEL**) register as illustrated in Figure 3.5.

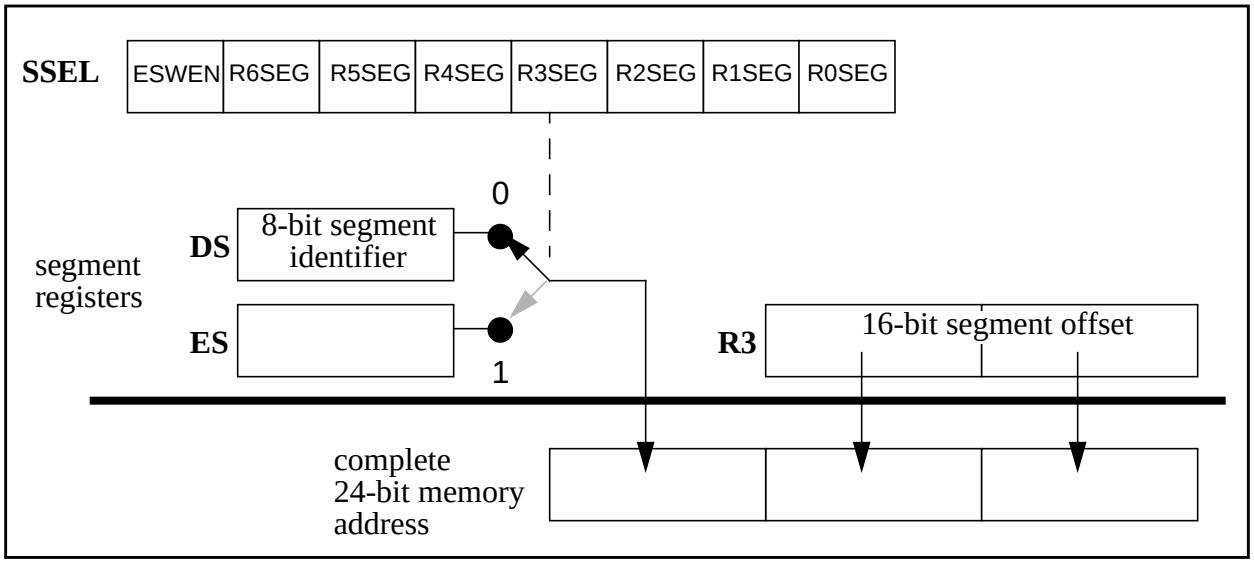


Figure 3.5 Address generation

A 0 in the SSEL bit corresponding to the pointer register selects DS (default on RESET) and 1 selects the ES. For example, when R3 contains a pointer value, the full 24 bit address is formed by concatenating DS or ES, as determined by the state of SSEL bit 3, as the most significant 8 bits. As a consequence of segmented addressing, the XA data memory space may be viewed as 256 segments of 64K bytes each (Figure 3.6).

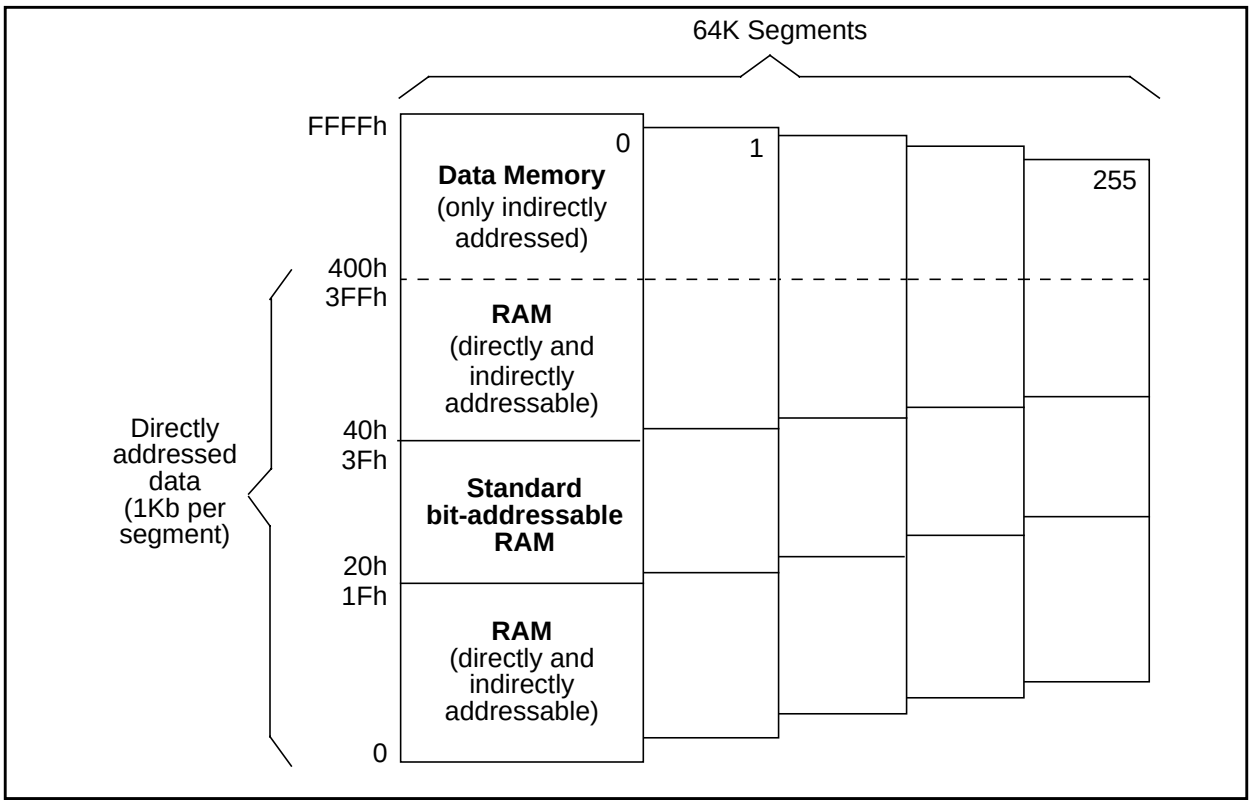


Figure 3.6 Data memory segmentation

If R7 (the stack pointer) is used as a normal indirect pointer, the data segment addressed will always be segment 0 in System Mode and the DS segment in User Mode. More information about the System and User modes may be found in sections 4 and 5.

The ESWEN (bit 7 of SSEL) can be programmed only in the System Mode to enable (1) or disable (0) write privileges to data segment via ES register in the User Mode. This bit defaults to the disabled (0) state after reset.

Addressing Modes

The XA provides flexible data addressing modes. Arithmetic, logic, and data movement instructions generally support the following data memory access:

Indirect. A complete 24-bit data memory address is formed by an 8-bit segment register concatenated with a 16-bit pointer in a register.

Direct. The first 1K bytes of data in each segment may be accessed by an address contained within the instruction. *Indirect with offset.* A signed byte/word offset contained within the instruction is added to the contents of a pointer register, and the result is concatenated with the 8-bit segment register DS to produce a complete 24-bit address.

Indirect with auto-increment. Indirect addresses are formed as above and the pointer register contents are automatically incremented.

Bit-level. Bit-level addresses are absolute references to specific bits.

Data move instructions and some special purpose instructions also have additional data addressing modes as described in Chapter 6.

Indirect Addressing

The entire 16 MByte address space is accessible via register-indirect addressing with a segment register, as illustrated by Figure 3.7 (Note that for simplicity, this figure omits showing how the Extra Segment or Data Segment Register is chosen using **SSEL**.).

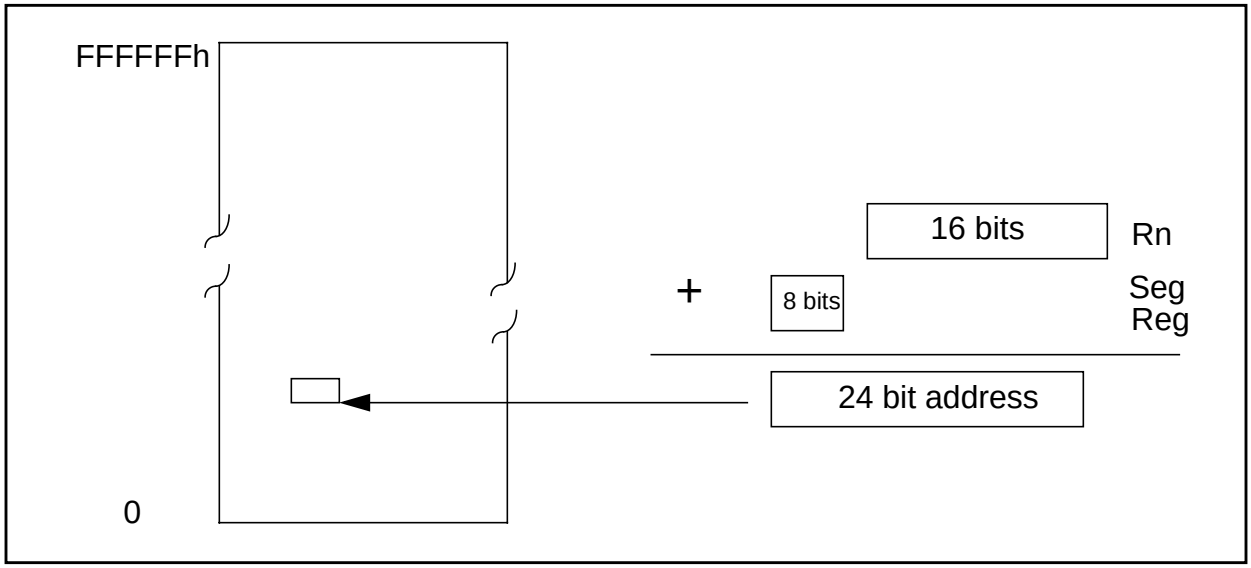


Figure 3.7 Indirect Access to 24 Bit Address Space

Indirect addressing with an offset is a variant of general indirect addressing in which an 8-bit or 16-bit signed offset contained within the instruction is added to the contents of a pointer register, then concatenated with an 8-bit segment register to produce a complete address. This mode gives access to data structures when a pointer register contains the starting address of the structure. It also supports stack-based parameter passing.

Indirect addressing with autoincrement is another variant of indirect addressing in which the pointer register contents are automatically incremented following the operation. When the operand is a byte, the increment is one; when the operand is a word, the increment is 2. Using indirect addressing with auto-increment provides a convenient method of traversing data structures smaller than 64K bytes. For data structures exceeding 64K bytes in length, the program code must explicitly adjust the segment register at page boundaries.

Address generation in these two modes of indirect addressing is illustrated inFigures 3.8 and 3.9. When using indirect addressing care is necessary to avoid accessing a word quantity at an odd address. This will result in an access using the next-lower even address, which is generally not desirable. Note that the indirect addressing with an offset will be successful in this case as long as the final, effective address is even. That is, both the base address and the offset may be odd.

Direct Addressing

The first 1K of each segment is directly addressable. Address generation for the direct address mode is summarized in Figure 3.10. Segment register DS is always used.

Direct data-reference instructions encode a maximum of 10 address bits, which are zero extended to sixteen bits and concatenated with DS to form an absolute 24 bit address. In all segments, direct addressing can be used to access any byte in the first 1K bytes of the segment.

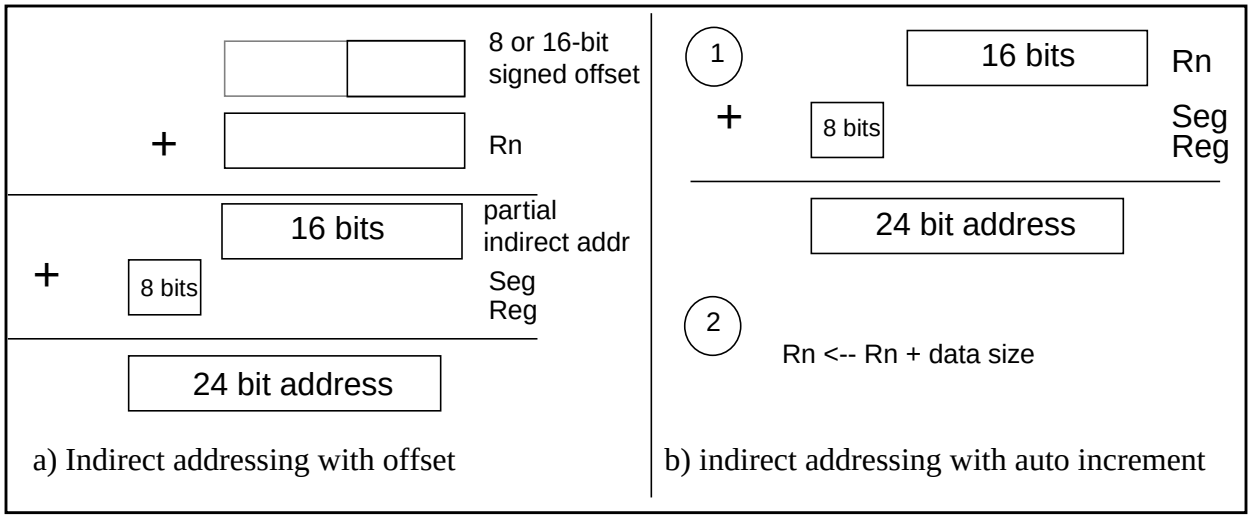


Figure 3.8 Indirect Addressing

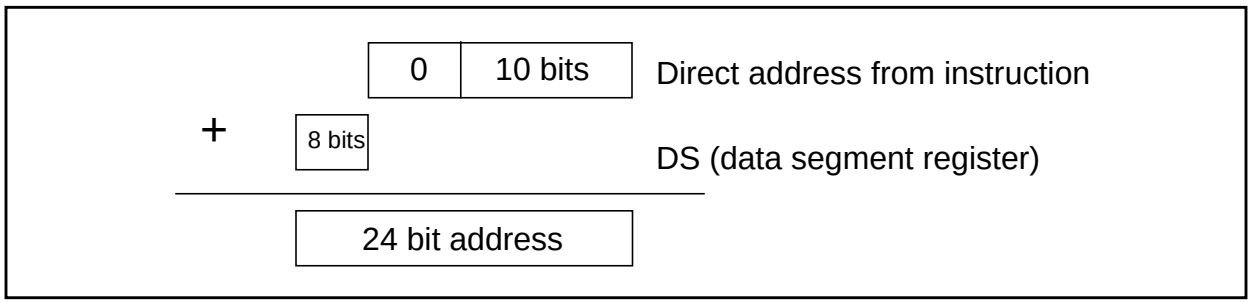


Figure 3.9 Direct address generation

SFR Addressing

A 1K portion of the direct address space, addresses 400h through 7FFh, is reserved for SFR addresses. The SFR address space uses a portion of the direct address space, but represents a completely distinct logical area that is not related to the data memory segmentation scheme. See section 3.6 for a complete description of SFR access.

Bit Addressing

Thirty-two bytes of each segment of data memory are also bit-addressable, starting at offset 20h in the segment addressed by the DS register. Address generation for bit addressing in the data memory space is shown in Figure 3.10. As described in chapter 6, bits are encoded in instructions as 10 bits. Figure 3.11 shows the bit addresses as they appear in memory .

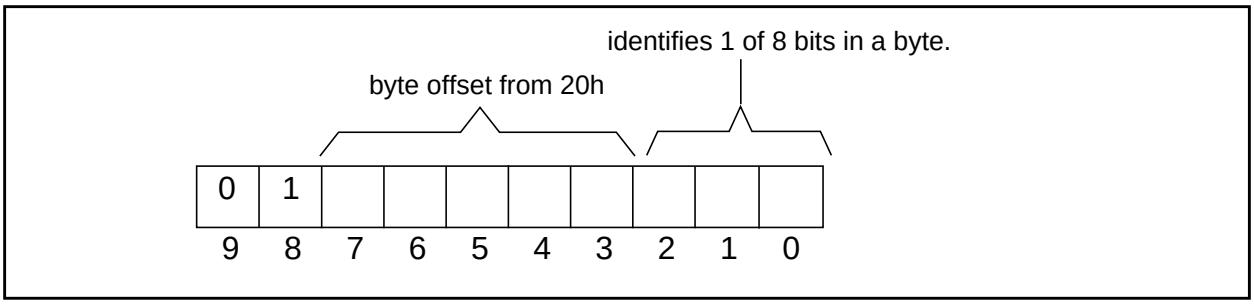


Figure 3.10 Bit address generation in direct memory space

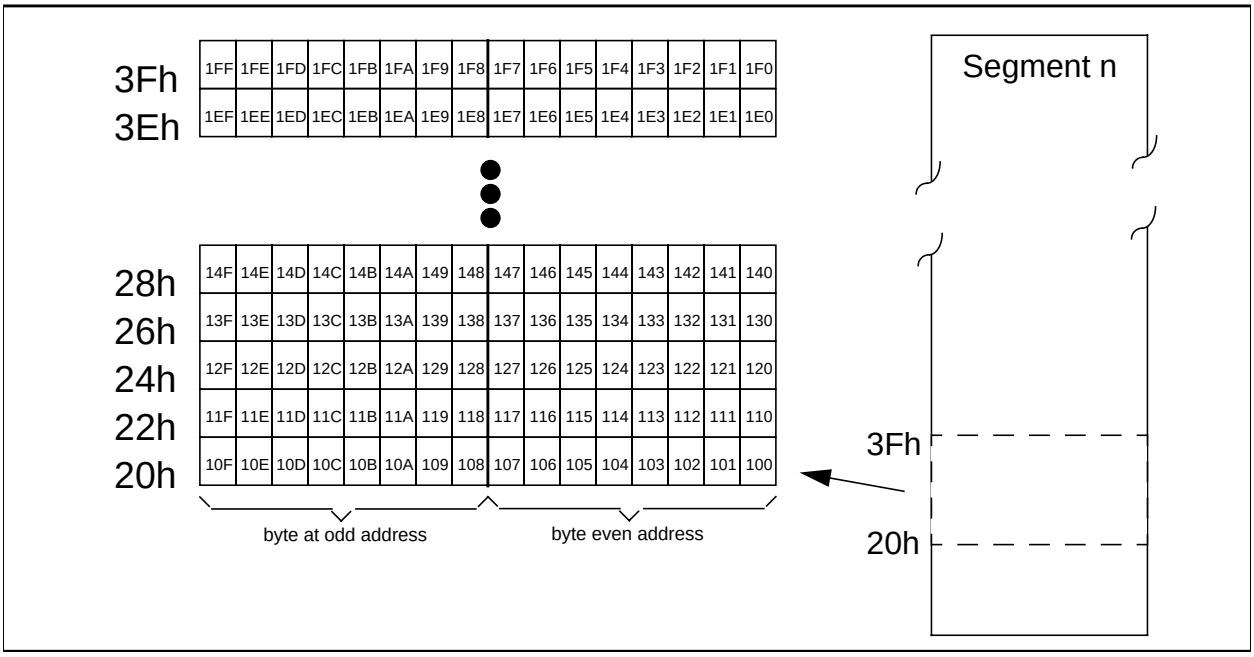


Figure 3.11 Direct memory bit addressing

3.5 Code Memory

Code memory starts at address 0 and extends to the highest valid address in the implementation, at maximum, FFFFFFh. *External Code Memory* (off-chip) starts at the first address following the highest *Internal Code Memory* (on-chip) location, if any. If code memory is present on-chip, it always starts at location 0.

The upper sixteen 64K byte code pages (addresses F00000 through FFFFFF hexadecimal) are reserved for special functions in XA derivatives. The same address range is reserved in the data memory space, see section 3.4.

3.5.1 Alignment in Code Memory

As instructions are variable in length, from 1 to 6 bytes (see Chapter 6), instructions in code memory can be located at odd addresses. As described in Chapter 6, instruction branch targets, i.e., targets of jumps, calls, branches, traps, and interrupts must be aligned on an even address.

3.5.2 External and Internal Overlap

If External Code Memory is placed by external logic at locations that overlap Internal Code Memory, the Internal Code Memory takes precedence, and the overlapped portion of the External memory will in not be accessed. However, on XA implementations that provide an External Address ($\overline{EA}$) hardware input, setting EA low will cause external program memory to be used.

3.5.3 Access

Code memory is intended to contain executable XA instructions**.** The XA architecture supports storing constant data in Code Memory and provides special access modes for retrieving this information. Constant data is implicitly stored within the instruction of many data manipulation instructions when immediate operands are specified.

It is possible, and a common practice, to overlap external code and data memory spaces. In this case it is important to understand that the memory spaces are logically separate. In such an architecture, implemented with external logic, code memory is logically read-only memory that is writable when accessed as external data memory. No such overlap is possible for internal code memory.

MOVC addressing in Code Memory

A special instruction, MOVC, is defined in the XA for accessing constant data (e.g lookup tables, string constants etc.) stored in code memory. There is a standard form of MOVC that reflects the native XA architecture, and there are two variations that reflect 80C51 compatibility; see Chapter 9 for details of 80C51 compatibility. The standard form of MOVC uses a 16-bit register value as a pointer, appended to either the top 8 bits of the Program Counter (PC) or the Code Segment register (CS) to form a 24-bit address, as shown in Figure 3.12. The source for the upper 8 address bits is determined by the setting of the segment selection bit (0 = PC and 1= CS) in the SSEL register that corresponds to the operand register.

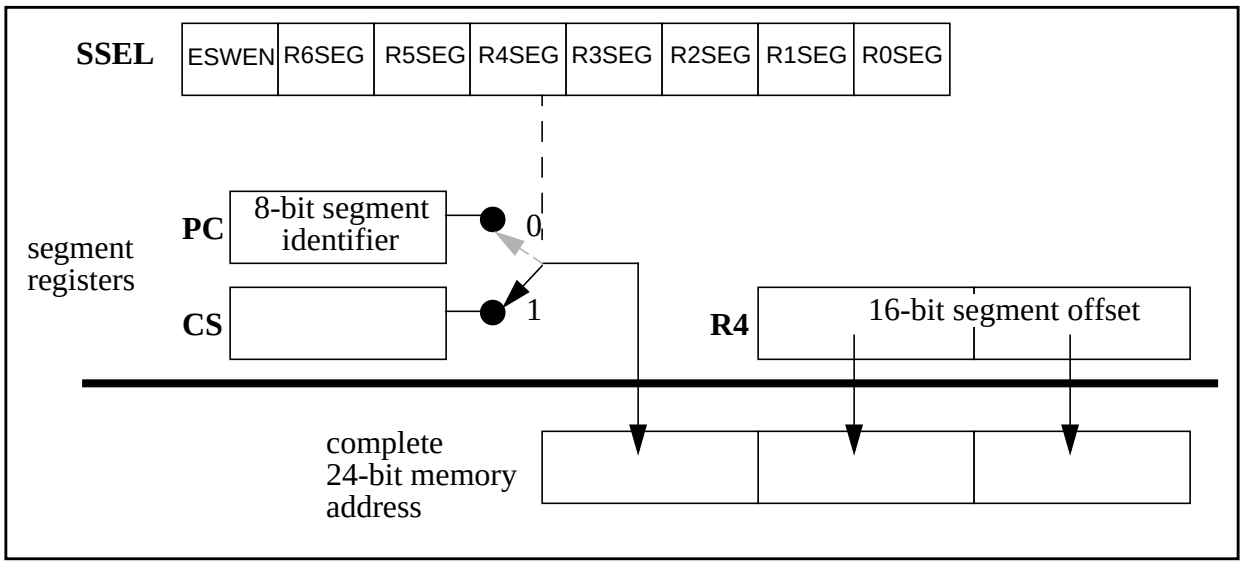


Figure 3.12 MOVC addressing in code memory

3.6 Special Function Registers (SFRs)

Special Function Registers (SFRs) provide a means for programs to access CPU control and status registers, peripheral devices, and I/O ports. The SFR mechanism provides a consistent mechanism for accessing standard portions of the XA core, peripheral functions added to the core within each XA derivative, and external devices as implemented in future derivatives.

Figure 3.13 highlights the core registers that are accessed as SFRs: **PCON**, **SCR**, **SSEL**, **PSWH**, **PSW**L, **CS**, **ES**, **DS**. Communication with these registers as well as on-chip peripheral devices is performed via the dedicated Special Function Register Bus (see section 8).

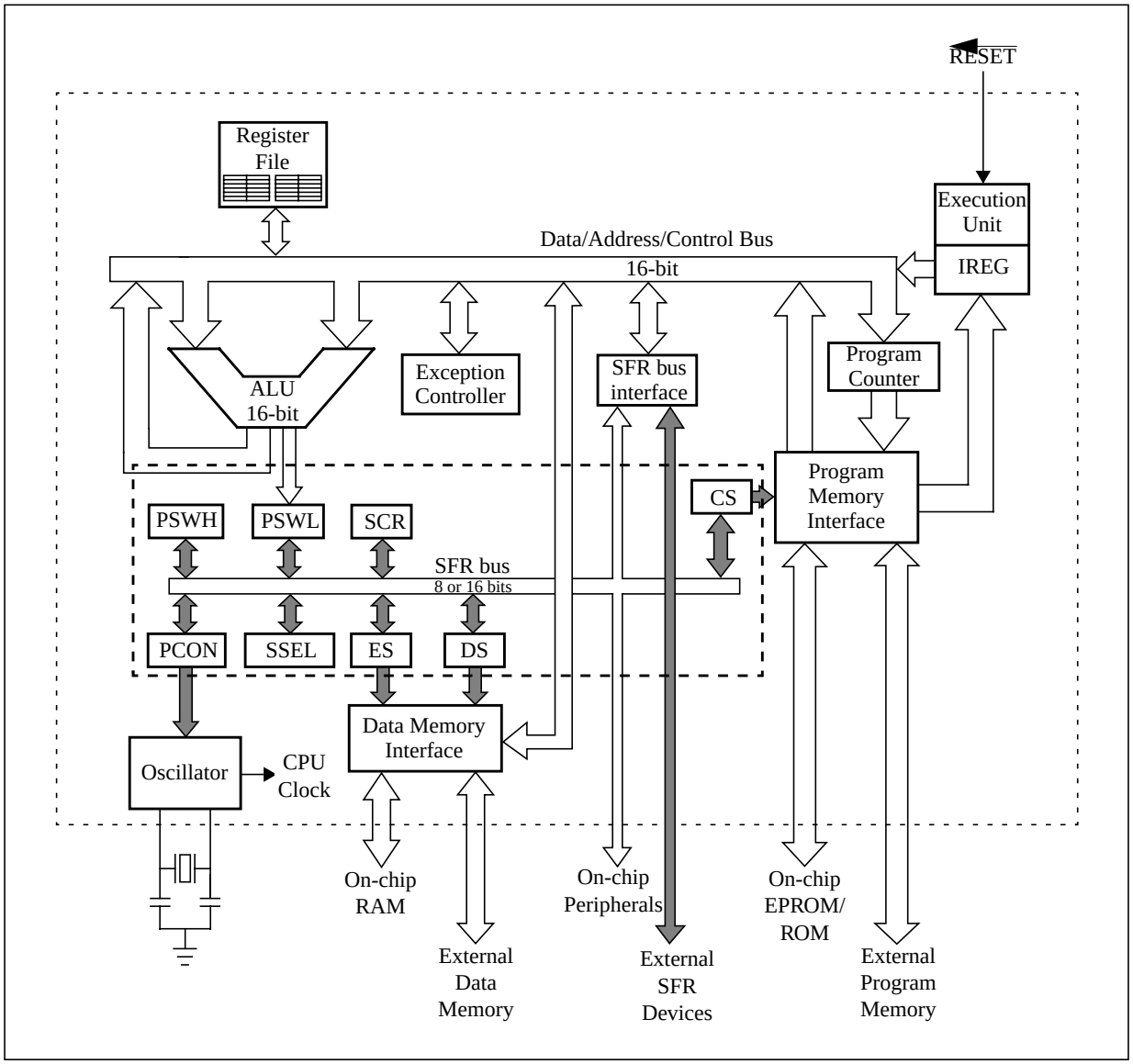


Figure 3.13 XA Core with SFRs highlighted

The SFR address space is 1K bytes (Figure 3.14). The first half of this space (400h through 5FFh) is dedicated to accessing core registers and on-chip peripherals outside the XA core. SFRs

assigned addresses in the range 400h through 43Fh are both byte and bit-addressable. The second half (600h through 7FFh) of the SFR space is reserved for providing access to off-chip SFRs. The off-chip sfr space is provided to allow faster access of off-chip memory mapped I/O devices without having to create a pointer for each access.

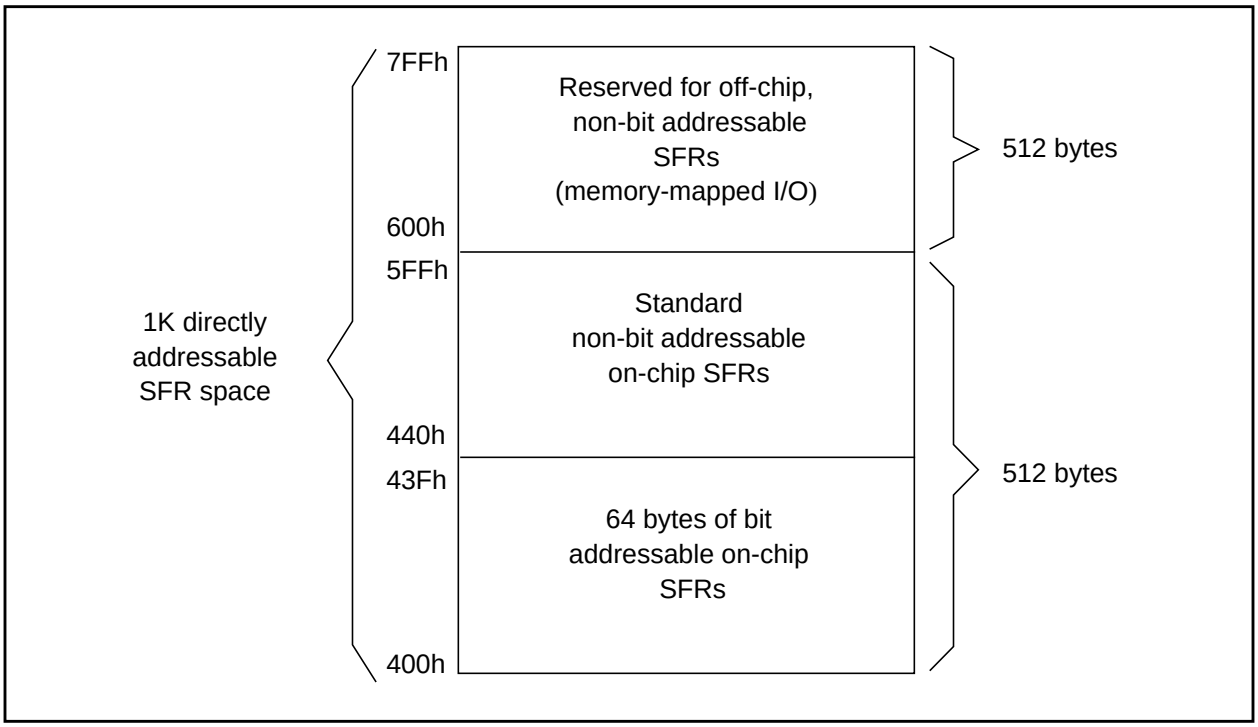


Figure 3.14 SFR address space

Following are some key points to remember when using SFRs:

SFRs should be symbolically addressed. Because SFR assignments may vary from derivative to derivative, it is important to always use symbolic references to SFRs. XA software development tools provide symbolic constants for all SFRs in the form of header/include files and the tools will be updated as new SFRs are added with each added XA derivative.

Verify that your application uses the right header/include files. Although baseline SFRs are likely to retain their addresses in future XA derivatives, this is not guaranteed. SFRs used for optional peripherals may well have different addresses on different derivatives, and the same address on one derivative may access a different peripheral SFR.

Any SFR may be accessed at any time without reference to a pointer or segment. SFR access is independent of any segment register, so SFRs are always accessible with the 10 bit address encoded in instructions accessing SFRs.

SFRs may not be accessed via indirect address. Any time indirection is used, data memory is accessed. If an SFR address is referenced as an indirect address, physical RAM at that address – if it exists– is accessed.

An SFR address is always contained entirely within an instruction. The SFR address is always encoded in the instruction providing the access, and there is no other way of addressing an SFR.

Details of access to external SFRs is determined by derivative implementation. Access to off-chip SFRs is a reserved feature not implemented in the baseline XA. Consult derivative product datasheets for details of external SFR access, e.g., timing.

3.7 Summary of Bit Addressing

Several sections of this chapter have described portions of the XA that are bit-addressable. There are a total of 1024 addressable bits distributed in the XA architecture, chosen to make important data structures immediately accessible via XA bit-processing instructions, specifically, all registers in the register file, R0 through R7 (and R8 through R15 if implemented); directly addressable RAM addresses 20h through 3Fh in the page currently specified by DS, and a portion of the on-chip SFRs. Figure 3.15 summarizes all the bit-addressable portions of the XA.5

bit space			overlaps bytes...		
start		end	type	start	end
0	◄►	0FFh	registers	R0 ◄►	R15
100h	◄►	1FFh	direct RAM	20h ◄►	3Fh
200h	◄►	3FFh	on-chip SFRs	400h ◄►	43Fh

Figure 3.15 Bit addressing summary

4 CPU Organization

This chapter describes the Central Processing Unit (CPU) of the XA Core. The CPU contains all status and control logic for the XA architecture. The XA reset sequence and the system oscillator interface with the CPU, and power control is handled here. The CPU performs interrupt and exception handling. The XA CPU is equipped with special functions to support debugging.

4.1 Introduction

Figure 4.1 is a block diagram of the XA Core.

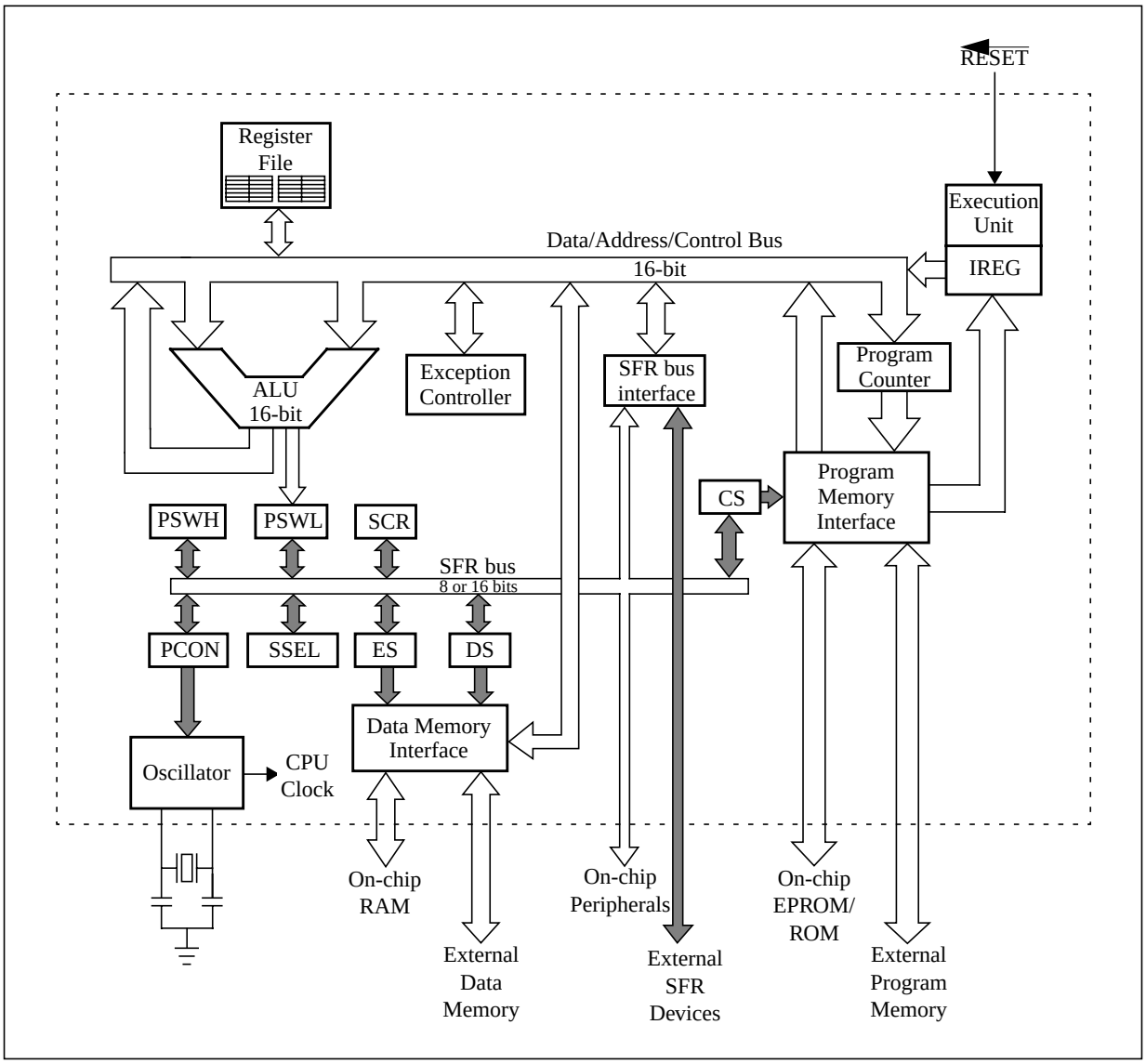


Figure 4.1 The XA Core

Here is an overview of core elements: The XA Core oscillator provides a basic system clock. Timing and control logic are initialized by an external reset signal; once initialized, this logic

provides internal and external timing for program and data memory access. This logic supervises loading the Program Counter and storing instructions fetched by the Program Memory Interface into the Instruction Register. The timing and control logic sequences data transfers to and from the Data Memory Interface. Under the same control, the ALU performs Arithmetic and Logical operations. The ALU stores status information in the low byte of the Program Status Word (**PSWL**). The on-board register file is used for intermediate storage and contains the current value of the Stack Pointer (**SP**). The high byte of the Program Status Word (**PSWH**) chooses between a privileged System Mode and a restricted User Mode; controls a Trace Mode used for single-step debugging, chooses the active register bank, and records the priority of the currently executing process. The System Configuration Register (**SCR**) is initialized to choose native XA mode execution or an 80C51 family compatibility mode. The Segment Selection Register (**SSL**) controls the use of the Code Segment (**CS**), Data Segment (**DS**), and the Extra Segment (**ES)** registers. The XA Core architecture supports interfaces to on- and off-chip RAM, ROM/ EPROM, and Special Function Registers (SFRs).

This chapter describes all these core elements in detail.

4.2 Program Status Word

The Program Status Word (**PSW**) is a two-byte SFR register that is a focal point of XA operations. The least significant byte contains the CPU status flags, which generally reflect the result of each XA instruction execution. This byte is readable and writable by programs running in both User and System modes.

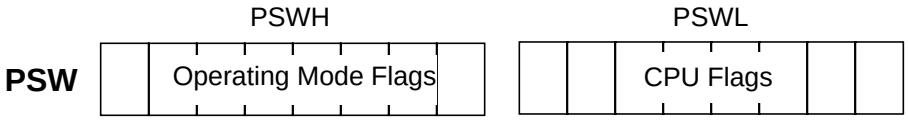


Figure 4.2 XA PSW

The most significant byte of **PSW** is written by programs to set important XA operating modes and parameters: system/user mode, trace mode, register bank select bits, and task execution priority. **PSWH** is readable by any process but only the register select bits may be modified by User mode code. All of the flags may be modified by code running in System Mode.

It should be noted that the XA includes a special SFR that mimics the original 80C51 PSW register. This register, called PSW51, allows complete compatibility with 80C51 code that manipulates bits in the PSW. See Chapter 9 for details of 80C51 compatibility.

4.2.1 CPU Status Flags

The PSW CPU flags (Figure 4.3) signify Carry, Auxiliary Carry, Overflow, Negative, and Zero. Some instructions affect all these flags, others only some of them, and a few XA instructions have no effect on the PSW status flags. In general, these flags are read by programs in order to make logical decisions about program flow. Chapter 6 describes comprehensively how CPU

Status Flags are affected by each instruction type. Consult reference pages in Chapter 6 for details about how individual instructions affect the PSW Status Flags.

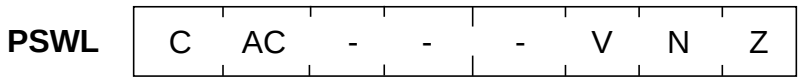


Figure 4.3 PSW CPU status flags

C, the Carry Flag, generally reflects the results of arithmetic and logical operations. It contains the carry out of the most significant bit of an arithmetic operation, if any, for the instructions ADD, ADDC, CMP, CJNE, DA, SUB, and SUBB.The carry flag is also used as an intermediate bit for shift and rotate instructions ASL, ASR, LSR, RLC, and RRC.

The multiply and divide instructions (MUL16, MULU8, MULU16, DIV16, DIV32, DIVU8, DIVU16, and DIVU32) unconditionally clear the carry flag.

AC, the auxiliary carry flag, is updated to reflect the result of arithmetic instructions ADD, ADDC, CMP, SUB, and SUBB with the carry out of the least significant nibble of the ALU. This flag is used primarily to support BCD arithmetic using the decimal adjust instruction (DA).

V is the overflow flag. It is set by an arithmetic overflow condition during signed arithmetic using instructions ADD, ADDC, CMP, NEG, SUB, and SUBB.

V is also set when the result of a divide instruction (DIV16, DIV32, DIVU8, DIVU16, DIVU32) exceeds the size of the specified destination register and when a divide-by-zero has occurred. For multiply instructions (MUL16, MULU8, MULU16) this flag is set when the result of a multiply instruction exceeds the source operand size. In this case “overflow” provides an indication to the program that the result is a larger data type than the source, such as a long integer product resulting from the multiply of two integers).

N reflects the twos complement sign (the high-order or “negative” bit) of the result of arithmetic operations and the value transferred by data moves. This flag is unaffected by PUSH, POP, SEXT, LEA, and XCH instructions.

Z (“zero”) reflects the value of the result of arithmetic operations and the value transferred by data moves. This flag is set if the result or value is zero, otherwise it is cleared. The flag is unaffected by PUSH, POP, SEXT, LEA, and XCH instructions.

Other bits (marked with “-” in the register diagram) are reserved for possible future use. Programs should take care when writing to registers with reserved bits that those bits are given the value 0. This will prevent accidental activation of any function those bits may acquire in future XA CPU implementations.

4.2.2 Operating Mode Flags

The PSW operating mode flags (Figure 4.4) set several aspects of the XA operating mode. All of the flags in the upper byte of the PSW (PSWH) except the bits RS1 and RS0 may be modified only by code running in system mode.

Figure 4.4 PSW operating mode flags

The System Mode bit, **SM**, when asserted, allows the currently running program full System Mode access to all XA registers, instructions, and memories. (For example, most of PSWH can only be modified when **SM** is asserted.) When this bit is cleared, the XA is running in User Mode and some privileges are denied to the currently running program.

The Trace Mode bit, **TM**, when set to 1, enables the built-in XA debugging facilities described in section 4.9. When **TM** is cleared, the XA debugging features are disabled.

The bits **RS1** and **RS0** identify one of the four banks of word registers R0 through R3 as the active register set. The other three banks are not accessible as registers (but also see the Compatibility Mode description in the System Configuration Register section).

The 4 bits **IM3** through **IM0** (Interrupt Mask bits) identify the execution priority of the current executing program. The event interrupt controller compares the setting of the IM bits to the priority of any pending interrupts to decide whether to initiate an interrupt sequence. The value 0 in the IM bits indicates the lowest priority, or fully interruptible code. The value 15 (or F hexadecimal) indicates the highest priority, not interruptible by event interrupts. Note that priority 15 does not inhibit servicing of exception interrupts or NMI.

The value of the IM bits may be written only by code operating in the system mode. Their value may be read by interrupt handler code to implement software-based interrupt priorities. Note that simply writing a new value to the interrupt mask bits can sometimes cause what is called a priority inversion, that is, the currently executing code may have a lower priority than previously interrupted code. The Software Interrupt mechanism is included on some XA derivatives specifically to avoid priority inversion in complex systems. Refer to the section on Software Interrupts for details.

4.2.3 Program Writes to PSW

The bytes comprising the PSW, namely PSWH and PSWL, are accessible as SFRs, and there is a potential ambiguity when a write to the PSW is performed by an instruction whose execution also modifies one or more PSW bits. The XA resolves this by giving full precedence to explicit writes to the PSW.

For example, executing

```
MOV.b R0L,#81h
```

sets PSW bit **N** to 1, since the byte value transferred is a twos complement negative number. However, executing

```
MOV.b PSWL, #81h
```

will set PSW bits **C** and **Z** and leave bit **N** cleared, since the value explicitly written to PSW takes precedence.

This precedence rule suppresses *all* PSW flag updates. When a value is written to the PSW, for example when executing

```
OR.b PSWH, #30
```

the contents of PSWL are unaffected.

4.2.4 PSW Initialization

As described below, at XA reset, the initial PSW value is loaded from the reset vector located at program memory address 0. Philips recommends that the PSW initialization value in the reset vector sets **IM3** through **IM0** to all 1's so that XA initialization is marked as the highest priority process (and therefore cannot be interrupted except by an exception or NMI). At the conclusion of the initialization code, the execution priority is typically reduced, often to 0, to allow all other tasks to run. It is also recommended that the reset vector set the **SM** bit to 1, so that execution begins in System Mode.

4.3 System Configuration Register

The System Configuration Register (**SCR**), described in Figure 4.5, sets XA global operating mode. **SCR** is intended to be written once during system start-up and left alone thereafter. Four bits are currently defined:

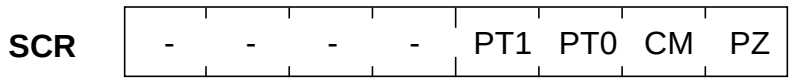


Figure 4.5 System Configuration Register (SCR)

PZ set to 0 (the default) puts the XA in the Large-Memory mode that uses full 24-bit XA addressing. When **PZ** = 1 the XA uses a small-memory "Page 0" mode that uses 16 bit addresses. The intent of Page 0 mode is to save stack space and improve interrupt latency in systems with less than 64K bytes of code and data memory. See the following sections for details.

CM chooses between standard "native" mode XA operation and 80C51 compatibility mode. When 80C51 compatibility mode is enabled, two things happen. First, the bottom 32 bytes of data memory in each data segment are replaced by the four banks of R0 through R3 from the register file. R0L of bank 0 will appear at data address 0, R0H of bank 0 will appear at data address 1, etc. Second, the use of R0 and R1 as indirect pointers is altered. To mimic 80C51 indirect addressing, indirect references to R0 use the byte R0L (zero extended to 16-bits) as the actual pointer value. References to R1 similarly use the byte R0H (zero extended to 16-bits) as the actual pointer value. Note that R0L and R0H on the XA are the same registers as R0 and R1 on the 80C51. No other XA features are altered or affected by compatibility mode. Operation of the XA with compatibility mode off (CM = 0) is reflected in descriptions found in the first 8 chapters of this User Guide. Operation with compatibility mode on (CM = 1) is discussed in Chapter 9.

PT1 and **PT0** select a submultiple of the oscillator clock as a Peripheral Timing clock source, in particular for timers but possibly for other peripherals in XA derivatives. Here are the values for these bits and the resulting clock frequency:

PT1	PT0	Peripheral Clock
0	0	oscillator/4
0	1	oscillator/16
1	0	oscillator/64
1	1	reserved

Other bits (marked with "-" in the register diagram) are reserved for possible future use. Programs should take care when writing to registers with reserved bits that those bits are given the value 0. This will prevent accidental activation of any function those bits may acquire in future XA CPU implementations.

4.3.1 XA Large-Memory Model Description

When the default XA operation is chosen via the **SCR** (**CM** = 0 and **PZ** = 0), all addresses are maintained by the core as 24 bit values, providing a full 16 MByte address space. On a specific XA derivative, fewer than 24 bits may be available at the external bus interface. All 24 address bits are pushed on the stack during calls and interrupts and 24 bits are popped by RETs and RETIs.

4.3.2 XA Page 0 Memory Model Description

When XA Page 0 mode is chosen, only 16 address bits are maintained by the XA core. This operating mode supports XA applications for which a 64K byte address space is sufficient. The external memory interface port used for the upper 8 address bits, if present, is available for other uses. A single 16-bit word is pushed on the stack during calls and interrupts and 16 bits are, in turn popped by RETs and RETIs. Using Page 0 mode when only a small memory model is needed saves stack space and speeds up address PUSH and POP operations on the stack.

Switching into or out of Page 0 mode after the original initialization is not recommended. First, switching into Page 0 mode can only be done by code running on Page 0, since the code address will be truncated to 16-bits as soon as Page 0 mode takes effect. Instructions already in the XA pre-fetch queue would have been fetched prior to Page 0 mode taking effect. Any addresses that may have been pushed onto the stack previously also become invalid when Page 0 mode is changed. Thus Page 0 mode could not be changed while in an interrupt service routine, or any subroutine.

4.4 Reset

The term “reset” refers specifically to the hardware input required when power is first applied to the XA device, and generally to the sequence of initialization that follows a hardware reset, which may occur at any time. The term also refers to the effect of the RESET instruction (see Chapter 6); in addition, an overflowing Watchdog timer (if this peripheral is present) has an identical effect.

This section describes the XA reset sequence and its implications for user hardware and software.

4.4.1 Reset Sequence Overview

A specific hardware reset sequence must be initiated by external hardware when the XA device is powered-up, before execution of a program may begin. If a proper reset at power up is not done, the XA may fail wholly or in part. The XA reset sequence includes the following sequential components:

- Reset signal generated by external hardware
- Internal Reset Sequence occurs
- $\overline{\text{RST}}$ line goes high
- External bus width and memory configuration determined
- Reset exception interrupt generated
- Startup Code executed

Figure 4.6 illustrates this process.

4.4.2 Power-up Reset

This section describes the reset sequence for powering up an XA device.

The XA $\overline{\text{RST}}$ input must be held low for a minimum reset period after Vdd has been applied to the XA device and has stabilized within specifications. The minimum reset period for a typical system with a reasonably fast power supply ramp-up time is 10 milliseconds. This reset period provides sufficient time for the XA oscillator to start and stabilize and for the CPU to detect the reset condition. At this point, the CPU initiates an internal reset sequence. $\overline{\text{RST}}$ must continue to be low for a sufficient time for the internal reset sequence to complete.

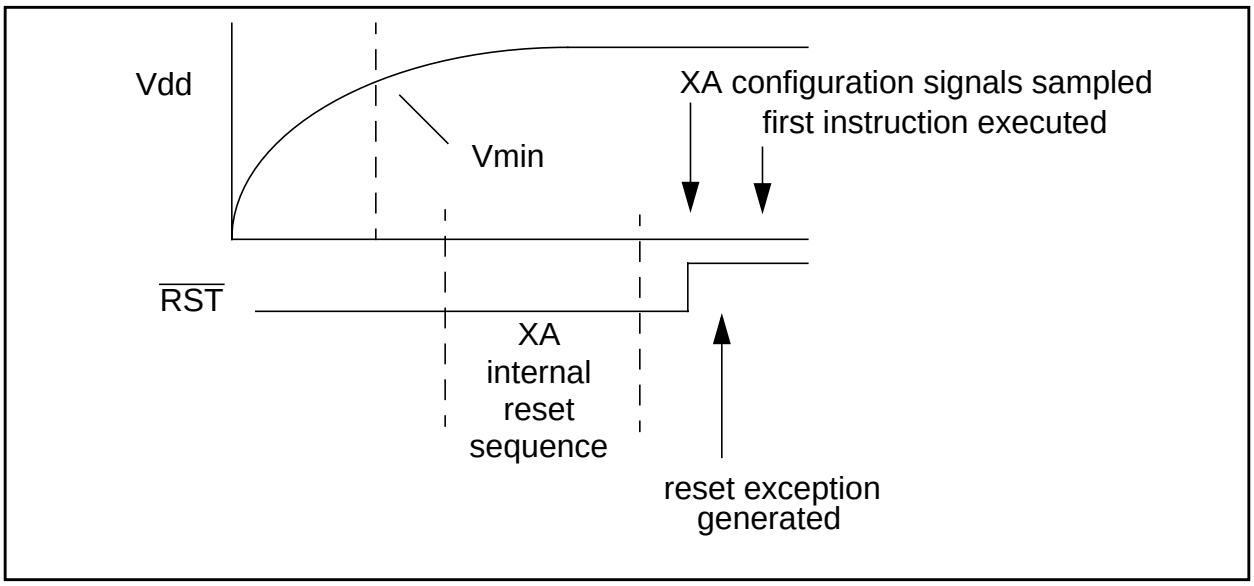


Figure 4.6 XA power-up sequence

4.4.3 Internal Reset Sequence

The XA internal reset sequence occurs after power-up or any time a sufficiently long reset pulse is applied to the $\overline{\text{RST}}$ input while the XA is operating. This sequence requires a minimum of a 10 microseconds (or 10 clocks, whichever is greater) to complete, and $\overline{\text{RST}}$ must remain low for at least this long.

The internal reset sequence does the following:

- Writes a 00 to most core and many peripheral SFRs. Other values are written to some peripheral SFRs. Consult the data sheet of a specific device for details.
- Sets **CS**, **DS**, and **ES** to 0.
- Sets **SSEL** = 0, i.e., sets all accesses through DS.
- Sets all registers in the Register File to 0.
- Sets the user and the system stack pointers (**USP** and **SSP**) to 0100h.
- Clears SCR bit **PZ**, i.e., 24-bit memory addresses will be used by default.
- Clears SCR bit **CM**, i.e., starts execution in XA Native Mode.
- Clears IE bit **EA**, disabling all maskable interrupts.

Note that the internal reset sequence does not initialize internal or external RAM. Note also that the contents of **PSW** at this point is not important, as it will immediately be replaced as described further below.

The effect of the internal reset sequence on components outside the XA core depends on the peripheral complement and configuration of the specific XA derivative. In general, the internal reset sequence has the following effects:

- Sets all port pins to inputs (quasi-bidirectional output configuration with port value = FF hex)
- Clears most SFRs to 0
- Initializes most other SFRs to appropriate non-zero values

Note that serial port buffers, PCA capture registers, and WatchDog feed registers (if present) are unaffected. Consult the XA derivative data sheet for more information.

After the XA internal reset sequence has been completed, the device is quiescent until the $\overline{RST}$ line goes high.

4.4.4 XA Configuration at Reset

As the $\overline{RST}$ line goes high, the value on two input pins is sampled to determine the XA memory and bus configuration. The $\overline{EA}$ and BUSW pins (if present on a specific XA derivative) have special function during the reset sequence, to allow external hardware to determine the use of internal or external program memory, and to select the default external bus width.

Immediately after the $\overline{RST}$ line goes high, the CPU triggers a reset exception interrupt, as described in the next section.

Selecting Internal or External Program Memory

The XA is capable of reading instructions from internal or external memory, both of which may be present. The XA $\overline{EA}$ input pin determines whether internal or external program memory will be used. The $\overline{EA}$ pin is sampled on the rising edge of the $\overline{RST}$ pulse. If $\overline{EA}$ = 0, the XA will operate out of external program memory, otherwise it will use internal code memory. The selection of external or internal code memory is fixed until the next time $\overline{RST}$ is asserted and released; until then all code fetches will access the selected code memory.

The XA cannot detect inconsistencies between the setting detected on the $\overline{EA}$ input and the hardware memory configuration. For example, setting $\overline{EA}$ = 1 on a ROMless XA variant will cause the XA to attempt to execute internal code memory, which is undefined on a ROMless device, typically resulting in a system failure.

Selecting External Bus Width

The XA is capable of accessing an 8 or 16 bit external data bus. The BUSW pin tells the XA the external data bus configuration. BUSW=0 selects an 8-bit bus and BUSW=1 selects an 16-bit bus. On power-up, the XA defaults to the 16-bit bus (due to an on-chip weak pull-up on BUSW). The BUSW pin is sampled on the rising edge of the $\overline{RST}$ pulse. If BUSW is low, the XA operates its external bus interface in 8 bit mode, otherwise, the XA uses 16 bit bus operation. The bus width may also be set under software control on derivatives equipped with the **BCR** (“Bus Configuration Register”) SFR.

After $\overline{RST}$ is released, the BUSW pin may be used an alternate function on some XA derivatives. Consult derivative data sheets for exact pinouts and details of how pins such as these may be shared to keep package size small.

4.4.5 The Reset Exception Interrupt

Immediately after the $\overline{\text{RST}}$ line goes high, the CPU generates a Reset Exception Interrupt. As a result, the initial PSW and address of the first instruction (the "start-up code") is fetched from the reset vector in code memory at location 0. Here's an example in generalized assembler format of the setup for the Reset Exception:

```
          code_seg            ; establish code segment
          org  0h             ; start at address 0

; reset_vector
          dw   initial_PSW    ; define a word constant
          dw   startup_code   ; define a word constant

          org  120h           ; move to address 120h
                              ; (above vector table)

startup_code:
          ...                 ; put startup code here
```

The initial value of **PSWL** set in the Reset Vector is generally of no special system-wide importance and may be set to zero or some other value to meet special needs of the XA application. The initial **PSWH** value sets the stage for system software initialization and its value requires more attention. Here's an example set of declarations that create the recommended initial value of **PSWH**:

```
system_mode    equ  8000h
max_priority   equ  0F00h
initial_PSW    equ  system_mode + max_priority
```

It is generally appropriate to initialize the XA in System Mode so that the start-up code has unrestricted access to the entire architecture. This is done by using a initial value that sets the PSWH bit **SM**.

Philips recommends initializing the execution priority of the start-up code to the highest possible value of 15 (that is, IM0 through IM3 to all ones) so that the start-up code is recognizable as the highest priority process. As described above, the hardware initialization sequence turns off all possible interrupts, so the only potential interrupting process would arise from a non-maskable interrupt (NMI). It is generally a good idea to prevent NMI generation with a hardware lock-out until XA start-up procedures are completed.

The **PSWH** initialization value given in this example sets System Mode (**SM**), selects register bank 0 (any register bank could be used) and clears **TM** so that Trace Mode is inactive.

4.4.6 Startup Code

Philips recommends that the first instruction of start-up code set the value of the System Configuration Register (**SCR**), described in section 4.3, to reflect the system architecture.

The next recommended step is explicitly initializing the stack pointers. The default values (section 4.7) are usually insufficient for application needs.

The start-up code sequence may be concluded by a simple branch or jump to application code. A RETI may not be used at the conclusion of a Reset Exception Interrupt handler (which causes the start-up code to run) because a reset initializes the SP and does not leave an interrupt stack frame.

4.4.7 Reset Interactions with XA Subsystems

The following describes how the reset process interacts with some key subsystems:

- Trace Exception. The trace exception is aborted by an external reset; see section 4.9.
- WatchDog. In XA derivatives equipped with a WatchDog timer feature, an internal reset will be asserted for a derivative-defined number of clocks.
- Resets while in Idle Mode or during normal code execution. Since the XA oscillator is running in Idle Mode, the $\overline{\text{RST}}$ input must be kept low for only 10 microseconds (or 10 clocks, whichever is greater) to achieve a complete reset.
- Resets while in Power-Down Mode. The XA oscillator is stopped in Power-Down mode, so the $\overline{\text{RST}}$ input must be low for at least 10 milliseconds. An exception to this is when an external oscillator is used and the XA is in Power-Down mode. In this case, if the external oscillator is running, a reset during Power-Down mode may be the same as a reset in Idle Mode.

4.4.8 An External Reset Circuit

The $\overline{\text{RST}}$ pin is a high-impedance Schmitt trigger input pin. For applications that have no special start-up requirements, it is practical to generate a reset period known to be much longer than that required by the power supply rise time and by the XA under all foreseeable conditions. One simple way to build a reset circuit is illustrated in Figure 4.7.

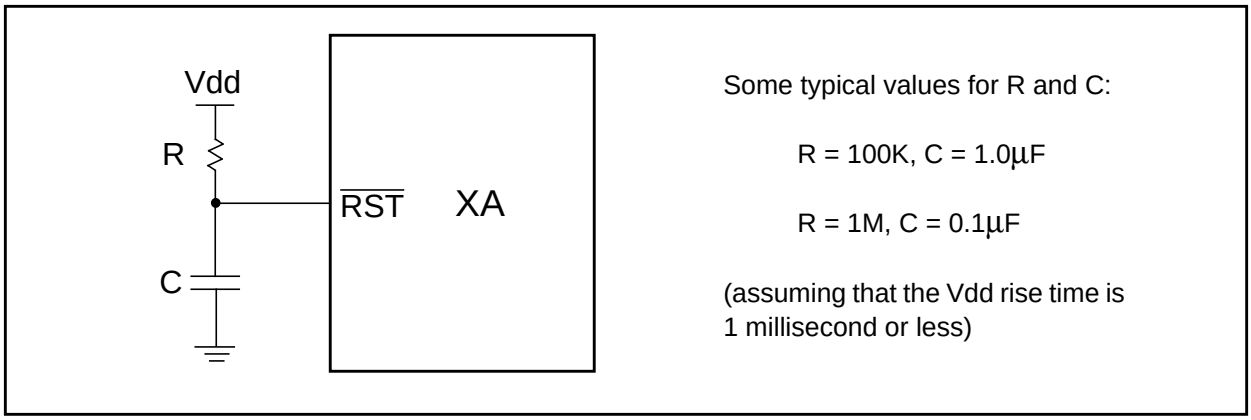


Figure 4.7 An external reset circuit

4.5 Oscillator

The XA contains an on-chip oscillator which may be used as the clock source for the XA CPU, or an external clock source may be used. A quartz crystal or ceramic resonator may be connected as shown in Figure 4.8a to use the internal oscillator. To use an external clock, connect the source to pin XTAL1 and leave pin XTAL2 open, as shown in Figure 4.8b.

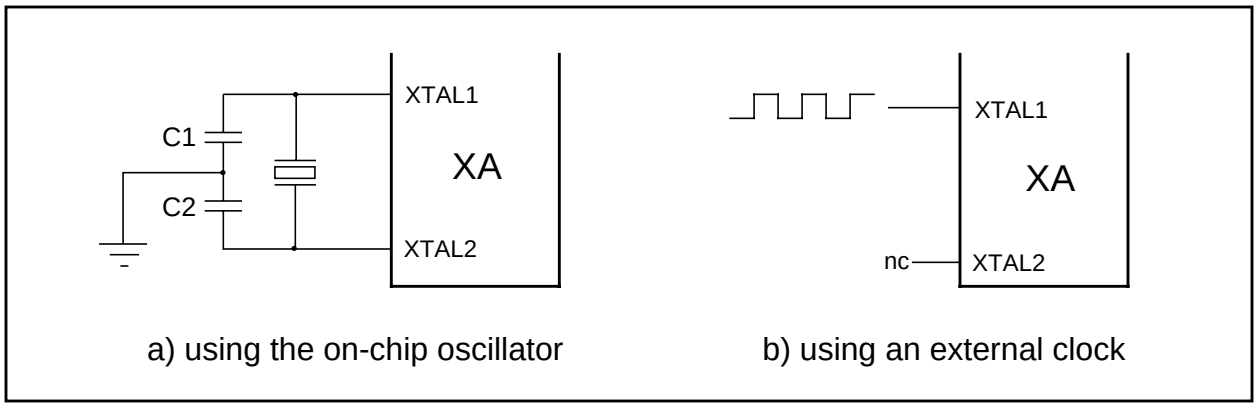


Figure 4.8 XA clock sources

The on-chip oscillator of the XA consists of a single stage linear inverter intended for use as a positive reactance oscillator. In this application, the crystal is operated in its fundamental response mode as an inductive reactance in parallel resonance with capacitance external to the crystal.

A quartz crystal or ceramic resonator is connected between the XTAL1 and XTAL2 pins, capacitors ar connected from both pins to ground. In the case of a quartz crystal, a parallel resonant crystal must be used in order to obtain reliable operation. The capacitor values used in the oscillator circuit should normally be those recommended by the crystal or resonator manufacturer. For crystals, the values may generally be from 18 to 24 pF for frequencies above 25 MHz and 28 to 34 pF for lower frequencies. Too large or too small capacitor values may prevent oscillator start-up or adversely affect oscillator start-up time.

4.6 Power Control

The XA CPU implements two modes of reduced power consumption: Idle mode, for moderate power savings, and Power-Down mode. Power-Down reduces XA consumption to a bare minimum. These modes are initiated by writing SFR **PCON**, as illustrated in Figure 4.9.

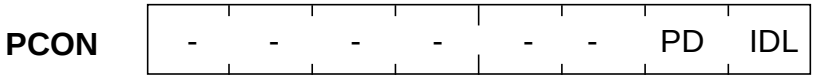


Figure 4.9 PCON

Idle Mode is activated by setting the PCON bit **IDL**. This stops CPU execution while leaving the oscillator and some peripherals running.

Power-Down mode is activated set by setting the PCON bit **PD**. This shuts down the XA entirely, stopping the oscillator.

The reset values of **IDL** and **PD** are 0. If a 1 is written to both bits simultaneously, **PD** takes precedence and the XA goes into Power-Down mode.

Other bits (marked with "-" in the register diagram) are reserved for possible future use. Programs should take care when writing to registers with reserved bits that those bits are given the value 0. This will prevent accidental activation of any function those bits may acquire in future XA CPU implementations.

4.6.1 Idle Mode

Idle mode stops program execution while leaving the oscillator and selected peripherals active. This greatly reduces XA power consumption. Those peripheral functions may cause interrupts (if the interrupt is enabled) that will cause the processor to resume execution where it was stopped.

In the Idle mode, the port pins retains their logical states from their pre-idle mode. Any port pins that may have been acting as a portion of the external bus revert to the port latch and configuration value (normally push-pull outputs with data equal to 1 for bus related pins). ALE and $\overline{\text{PSEN}}$ are held in their respective non-asserted states. When Idle is exited normally (via an active interrupt), port values and configurations will remain in their original state.

4.6.2 Power-Down Mode

Power-Down mode stops program execution and shuts down the on-chip oscillator. This stops all XA activity. The contents of internal registers, SFRs and internal RAM are preserved. Further power savings may be gained by reducing XA Vdd to the RAM retention voltage in Power Down mode; see the device data sheet for the applicable Vdd value. The processor may be re-activated by the assertion of $\overline{\text{RST}}$ or by assertion of one of an external interrupt, if enabled. When the processor is re-activated, the oscillator will be restarted and program execution will resume where it left off.

In Power-Down mode, the ALE and $\overline{\text{PSEN}}$ outputs are held in their respective non-asserted states. The port pins output the values held by their respective SFRs. Thus, port pins that are not configured to be part of an external bus retain their state. Any port pins that may have been acting as a portion of the external bus revert to the port latch and configuration value (normally push-pull outputs with data equal to 1 for bus related pins). If Power-Down mode is exited via Reset, all port values and configurations will be set to the default Reset state.

In order to use an external interrupt to re-activate the XA while in Power-Down mode, the external interrupt must be enabled and be configured to level sensitive mode. When Power-Down mode is exited via an external interrupt, port values and configurations will remain in their original state. Since the XA oscillator is stopped when the XA leaves Power-Down mode via an interrupt, time must be allowed for the oscillator to re-start. Rather than force the external logic asserting the interrupt to remain active during the oscillator start-up time, the XA implements its own timer to insure proper wake-up. This timer counts 9,892 oscillator clocks before allowing the XA to resume program execution, thus insuring that the oscillator is running and stable at

that time. Once the oscillator counter times out, the XA will execute the interrupt that woke it up, if that interrupt is of a higher priority than the currently executing code.

Note that if an external oscillator is used, power supply current reduction in the Power-Down mode is reduced from what would be obtained when using the XA on-chip oscillator. In this case, full power savings may be gained by turning off the external clock source or stopping it from reaching the XTAL1 pin of the XA. If the clock source may be turned off, it may be advantageous to use Idle mode rather than Power-Down mode, to allow more ways of terminating the power reduction mode and to avoid the 9,892 clock waiting period for exiting Power-Down mode.

4.7 XA Stacks

The XA stacks are word-aligned LIFO data structures that grow downward in data memory, from high to low address. This and some other details of the XA stack implementation differ from 80C51 stack operation. Refer to the chapter on 8051 compatibility for a detailed discussion of this topic.

The XA implements two distinct stacks, one for User Mode and one for System Mode. The User Stack may be placed anywhere in data memory, while the System Stack must be located in the first 64K bytes, i.e., segment 0.

4.7.1 The Stack Pointers

The XA has two stacks, the system stack and the user stack. Each stack has an associated stack pointer, the System Stack Pointer (SSP) and the User Stack Pointer (USP), respectively. Only one of these stacks is active at a given time. The current stack pointer at any instant (which may be the SSP or the USP) appears as word register SP (R7) in the register file; the other stack pointer will not be visible. The value of the PSW bit **SM** determines which stack is active (and whose stack pointer therefore appears as R7). In User Mode (**SM** = 0), SP (R7) contains the User Stack Pointer. In System Mode (**SM** =1), SP (R7) contains the System Stack Pointer. The XA automatically switches SSP and USP values when the operating mode is changed. Note that the terms “USP” and “SSP” are logical terms, denoting the value of SP (R7) in each mode.

Segments and Protection

The User stack is always addressed relative to the current data segment (DS) value. This is consistent with each user task being associated with a specific data segment. Moreover, code running in User Mode cannot modify **DS**, so there is no possibility of changing the segment in which the stack resides within the User context. The System Stack must always be located in segment 0, that is, the first 64K of data memory.

4.7.2 PUSH and POP

The PUSH operation is illustrated by Figure 4.10. The stack pointer always points to an existing data item at the top of the stack, and is decremented by 2 prior to writing data.

The POP operation copies the data at the top of the stack and then adds two to the stack pointer, as follows shown in Figure 4.11.

All stack pushes and pops occur in word multiples. If a byte quantity is pushed on the stack it is stored as the least significant byte of a word and the high byte is left unwritten;
see Figure 4.12. A POP to a byte register removes a word from the stack and the byte register receives the least significant 8 bits of the word, as shown in Figure 4.13.

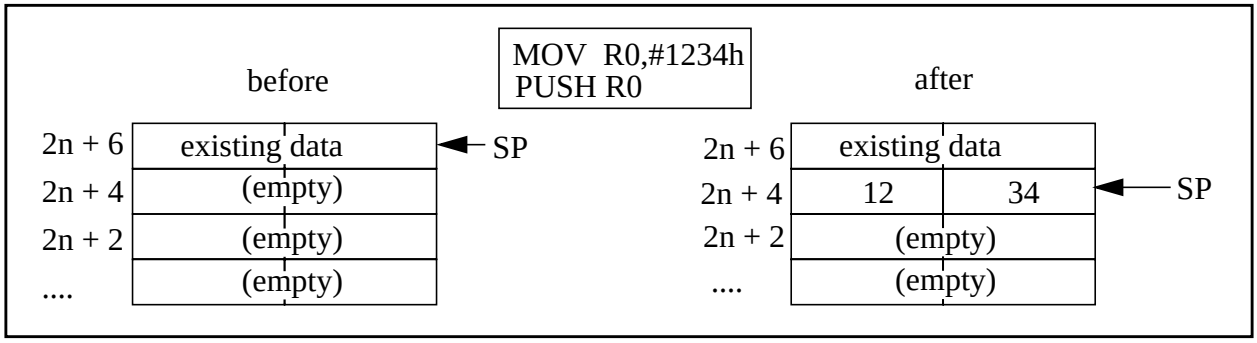


Figure 4.10 PUSH operation

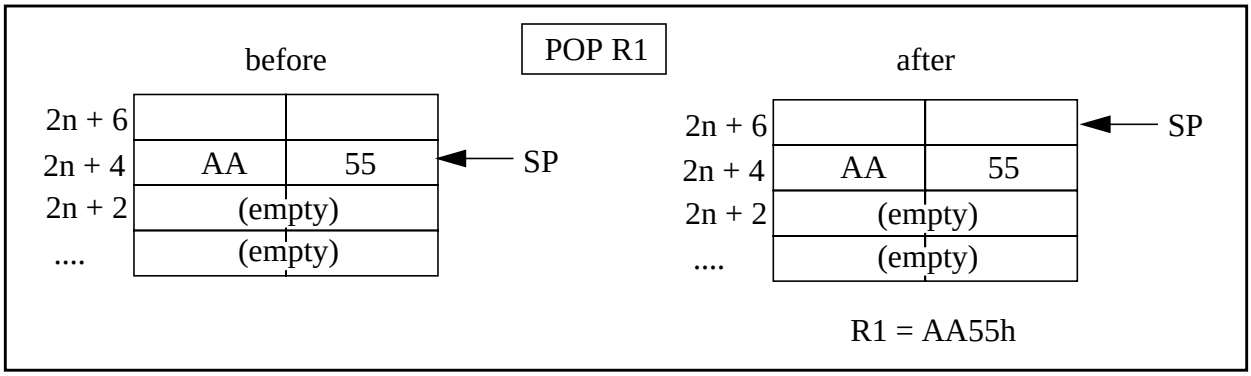


Figure 4.11 POP operation

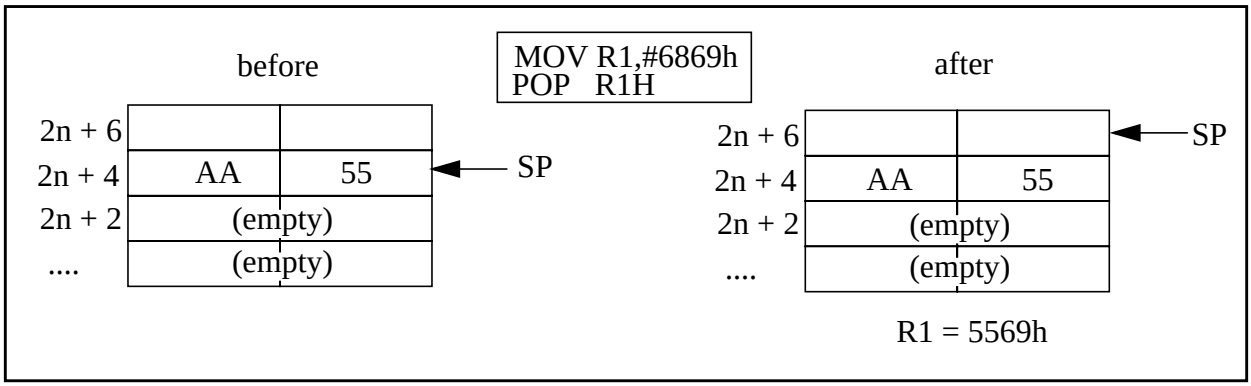


Figure 4.12 POP a byte

The stack should always be word-aligned. If the SP (R7) is modified to an odd value, the offending LSB of the stack pointer is ignored and the word at the next-lower even address is accessed.

Note that neither PUSH or POP operations have any effect on the PSW flags.

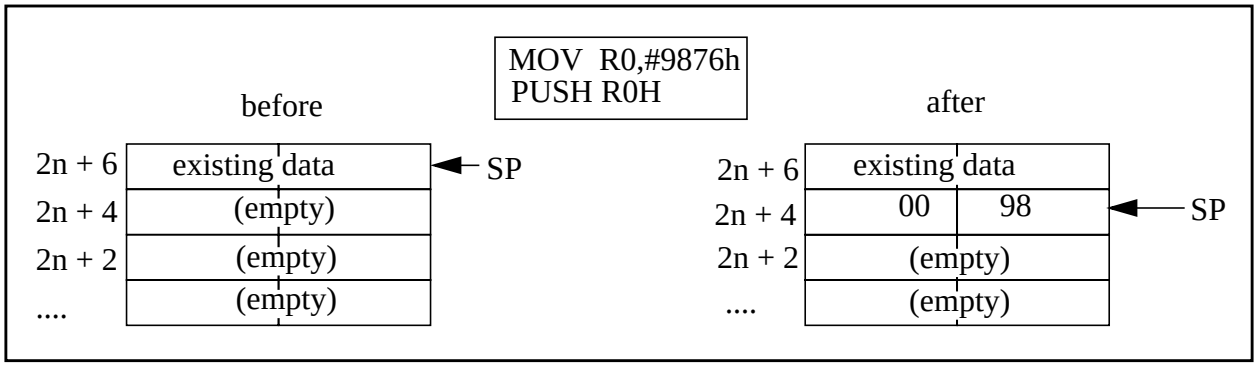


Figure 4.13 PUSH a byte

4.7.3 Stack-Based Addressing

Stack-based data addressing is fully supported by the XA. R0 through R7 may be used in all indexed address modes; the stack pointer in R7 is equally valid as an index.
Figure 4.14 illustrates an example of stack-based addressing. The segment used for stack relative addressing is always the same as for other stack operations (Segment 0 for System mode code and DS for User mode code).

Note that the precautions mentioned in section 3.3.4 apply here: when referencing a word quantity, the final (effective) address must be even, otherwise incorrect data will be accessed. This topic is discussed further in the section Stack Pointer Misalignment.

4.7.4 Stack Errors

Special attention is required to avoid problems due to stack overflow, stack underflow, and stack pointer misalignment

Stack Overflow

Stack overflow occurs when too many items are pushed, either explicitly or as the result of interrupts. As items are pushed on to the stack, it may grow downward past the memory allocated to it. It is not always possible for programs to detect stack overflow, so the XA triggers a Stack Overflow Exception Interrupt whenever the value of the *current* stack pointer (SSP or USP) decrements from 80h to 7Eh (simply setting SP to a value lower than 80h would NOT cause a stack overflow). This value was chosen so that stack space sufficient to handle a stack overflow exception interrupt is always guaranteed, as follows:

The 80h limit leaves 64 bytes available for stack overflow processing. A worst case might be occurs when the Stack Pointer is at 80h and a program executes an 8 word push; this generates a stack overflow. If an NMI occurs at the same time, 3 additional words are pushed. The balance

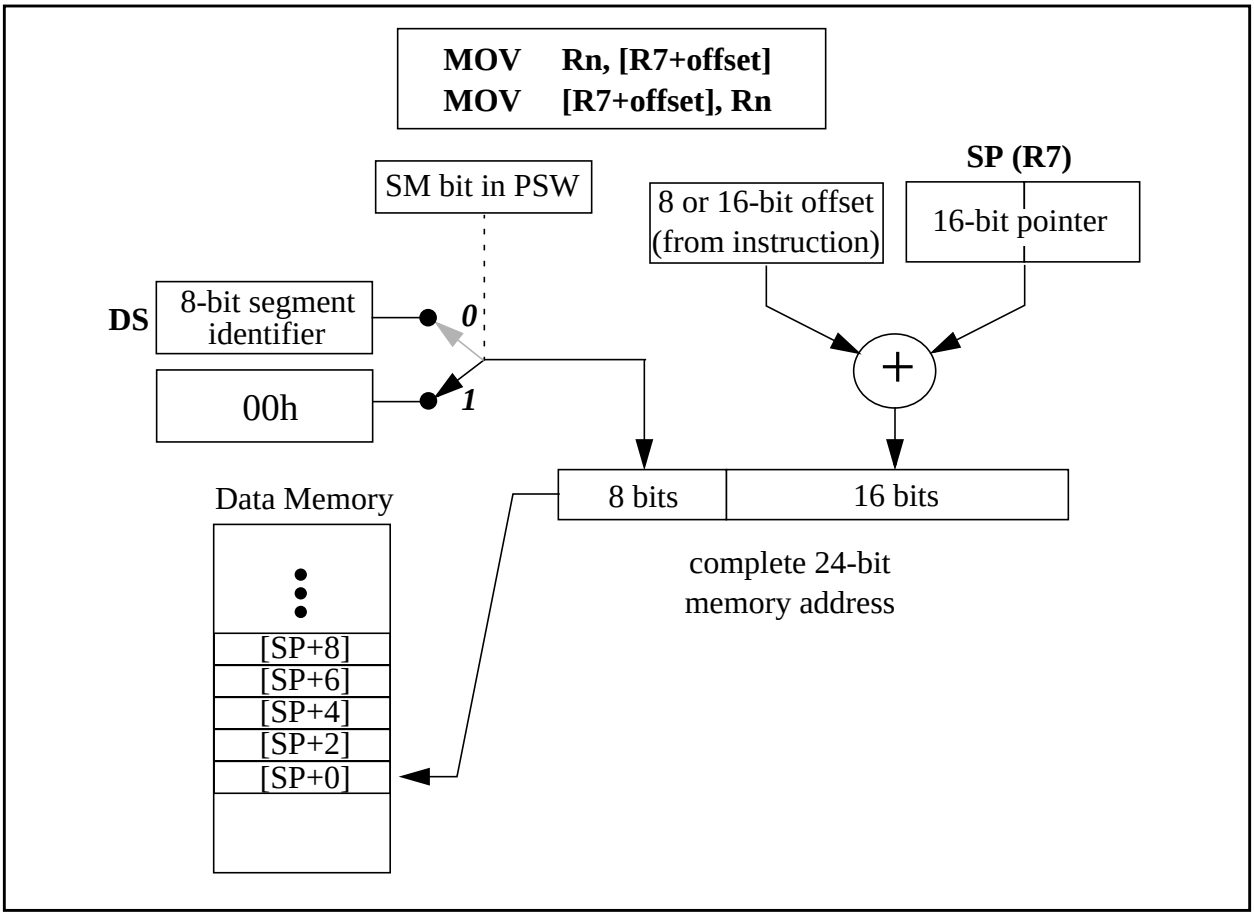


Figure 4.14 Stack-based addressing

of the 64 bytes on the stack is available for handler processing, which should carefully limit further use of the stack.

Stack Underflow

Stack underflow occurs when too many items are popped and the stack pointer value becomes greater than its initial value, i.e., the stack top. The XA does not support stack underflow detection.

Stack Pointer Misalignment

Pointer misalignment occurs when a pointer contains an odd value and is used by an instruction to access a word value in memory. The same situation could occur if some program action forced the stack pointer to an odd value. In these cases, the XA ignores the bottom bit of the pointer and continues with a word memory access.

4.7.5 Stack Initialization

At power-on reset, *both* USP and SSP in all XA derivatives are initialized to 100h. Since SP is pre-decremented, the first PUSH operation will store a word at location FEh and the stack will grow downwards from there.

These default stack pointer start-up values overlap the System and User stacks and are applicable only when one of these stacks will never be used.

Since the System stack is used for all exception and interrupt processing, this may not be appropriate in all XA applications. The startup code should normally set new and different values of both USP and SSP.

4.8 XA Interrupts

The XA architecture defines four kinds of interrupts. These are listed below in order of intrinsic priority:

- Exception Interrupts
- Event Interrupts
- Software Interrupts
- Trap Interrupts

Exception interrupts reflect system events of overriding importance. Examples are stack overflow, divide-by-zero, and Non-Maskable Interrupt. Exceptions are always processed immediately as they occur, regardless of the priority of currently executing code.

Event interrupts reflect less critical hardware events, such as a UART needing service or a timer overflow. Event interrupts may be associated with some on-chip device or an external interrupt input. Event interrupts are processed only when their priority is higher than that of currently executing code. Event interrupt priorities are settable by software.

Software interrupts are an extension of event interrupts, but are caused by software setting a request bit in an SFR. Software interrupts are also processed only when their priority is higher than that of currently executing code. Software interrupt priorities are fixed at levels from 1 through 7.

Trap interrupts are processed as part of the execution of a TRAP instruction. So, the interrupt vector is always taken when the instruction is executed.

All forms of interrupts trigger the same sequence: First, a *stack frame* containing the address of the next instruction and then the current value of the PSW is pushed on the System Stack. A vector containing a new PSW value and a new execution address is fetched from code memory. The new PSW value entirely replaces the old, and execution continues at the new address, i.e., at the specific interrupt handler.

The new PSW value may include a new setting of PSW bit **SM**, allowing handler routines to be executed in System or User mode, and a new value of PSW bits **IM3** through **IM0**, reflecting the execution *priority* of the new task. These capabilities are basic to multi-tasking support on the XA. See Chapter 5 for more details.

Returns from all interrupts should in most cases be accomplished by the RETI instruction, which pops the System Stack and continues execution with the restored PSW context. Since RETI executed while in User Mode will result in an exception trap, as described further below, interrupt service routines will normally be executed in System Mode.

The XA architecture contains sophisticated mechanisms for deciding when and if an interrupt sequence actually occurs. As described below, Exception Interrupts are always serviced as soon as they are triggered. Event Interrupts are deferred until their execution priority is higher than that of the currently executing code. For both exception and event interrupts, there is a systematic way of handling multiple simultaneous interrupts. Software and trap interrupts occur only when program instructions generating them are executed so there is no need for conflict resolution.

The Non-Maskable Interrupt requires special consideration. It is generated outside the XA core, and in that respect is an event interrupt. However, it shares many characteristics of exception interrupts, since it is not maskable. Note that NMI, while part of the XA CPU core, may not always be connected to a pin or other event source on all XA derivatives.

4.8.1 Interrupt Type Detailed Descriptions

This section describes the four kinds of interrupts in detail.

Exception Interrupts

Exception interrupts reflect events of overriding importance and are always serviced when they occur. Exceptions currently defined in the XA core include: Reset, Breakpoint, Divide-by-0, Stack overflow, Return from Interrupt (RETI) executed in User Mode, and Trace. Nine additional exception interrupts are reserved. NMI is listed in the table of exception interrupts (Table 4.1) below because NMI is handled by the XA core in same manner as exceptions, and factors into the precedence order of exception processing.

Since exception interrupts are by definition not maskable, they must always be serviced immediately regardless of the priority level of currently executing code, as defined by the IM bits in the PSW. In the unusual case that more than one exception is triggered at the same time, there is a hard-wired *service precedence* ranking. This determines which exception vector is taken first if multiple exceptions occur. In these cases, the exception vector taken *last* may be considered the highest priority, since its code will execute first. Of course, being non-maskable, any exception occurring during execution of the ISR for another exception will still be serviced immediately.

Programmers should be aware of the following when writing exception handlers:

1. Since another exception could interrupt a stack overflow exception handler routine, care should be taken in all exception handler code to minimize the possibility of a destructive stack overflow. Remember that stack overflow exceptions only occur once as the stack crosses the bottom address limit, 80h.

2. The breakpoint (caused by execution of the BKPT instruction, or a hardware breakpoint in an emulation system) and Trace exceptions are intended to be mutually exclusive. In both cases, the handler code will want to know the address in user code where the exception occurred. If a breakpoint occurs during trace mode, or if trace mode is activated during execution of the breakpoint handler code, one of the handlers will see a return address on the stack that points within the other handler code.

Table 4.1: Exception interrupts, vectors, and precedence

Exception Interrupt	Vector Address	Service Precedence
Breakpoint	0004h:0007h	0
Trace	0008h:000Bh	1
Stack Overflow	000Ch:000Fh	2
Divide-by-zero	0010h:0013h	3
User RETI	0014h:0017h	4
<reserved>	0018h - 003Fh	5
NMI	009Ch:009Fh	6
Reset	0000h:0003h	7 (always serviced immediately, aborts other exceptions)

Event Interrupts

Event Interrupts are typically related to on-chip or off-chip peripheral devices and so occur asynchronously with respect to XA core activities. The XA core contains no inherent event interrupt sources, so event interrupts are handled by an interrupt control unit that resides on-chip but outside of the processor core.

On typical XA derivatives, event interrupts will arise from on-chip peripherals and from events detected on interrupt input pins. Event interrupts may be globally disabled via the **EA** bit in the Interrupt Enable register (IE) and individually masked by specific bits the IE register or its extension. When an event interrupt for a peripheral device is disabled but the peripheral is not turned off, the peripheral interrupt flag can still be set by the peripheral and an interrupt will occur if the peripheral is re-enabled. An event interrupt that is enabled is serviced when its priority is higher than that of the currently executing code. Each event interrupt is assigned a priority level in the Interrupt Priority register(s). If more than one event interrupt occurs at the same time, the priority setting will determine which one is serviced first. If more than one interrupt is pending at the same level priority, a hardwares precedence scheme is used to choose the first to service. The XA architecture defines 15 interrupt occurrence priorities that may be programmed into the Interrupt Priority registers for Event Interrupts. Note that some XA implementations may not support all 15 levels of occurrence priority. Consult the data sheet for a specific XA derivative for details.

Note that, like all other forms of interrupts, the PSW (including the Interrupt Mask bits) is loaded from the interrupt vector table when an event interrupt is serviced. Thus, the priority at which the interrupt service routine executes could be different than the priority at which the interrupt occurred (since that was determined not by the PSW image in the vector table, but by the Interrupt Priority register setting for that interrupt). Normally, it is advisable to set the execution priority in the interrupt vector to be the same as the Interrupt Priority register setting that will be used in the program.

Furthermore, the occurrence priority of an interrupt should never be set higher than the execution priority. This could lead to infinite interrupt nesting where the interrupt service routine is re-interrupted immediately upon entry by the same interrupt source.

Software Interrupts

Software Interrupts act just like event interrupts, except that they are caused by software writing to an interrupt request bit in an SFR. The standard implementation of the software interrupt mechanism provides 7 interrupts which are associated with 2 Special Function Registers. One SFR, the software interrupt request register (SWR), contains 7 request bits: one for each software interrupt. The second SFR is an enable register (SWE), containing one enable bit matching each software interrupt request bit.

Software interrupts are initiated by setting one of the request bits in the SWR register. If the corresponding enable bit in the SWE register is also set, the software interrupt will occur when it becomes the highest priority pending interrupt and its priority is higher than the current execution level. The software interrupt request bit in SWR must be cleared by software prior to returning from the software interrupt service routine.

Software interrupts have fixed interrupt priorities, one each at priorities 1 through 7. These are shown in Table 4.2 below. Software Interrupts are defined outside the XA core and may not be present on all XA derivatives; consult the specific XA derivative data sheet for details.

Table 4.2: Software interrupts, vectors, and fixed priorities

Software Interrupt	Vector Address	Fixed Priority
SWI1	0100h:0103h	1
SWI2	0104h:0107h	2
SWI3	0108h:010Bh	3
SWI4	010Ch:010Fh	4
SWI5	0110h:0113h	5
SWI6	0114h:0117h	6
SWI7	0118h:011Bh	7

The primary purpose of the software interrupt mechanism is to provide an organized way in which portions of event interrupt routines may be executed at a lower priority level than the one

at which the service routine began. An example of this would be an event Interrupt Service Routine that has been given a very high priority in order to respond quickly to some critical external event. This ISR has a relatively small portion of code that must be executed immediately, and a larger portion of follow-up or “clean-up” code which does not need to be completed right away. Overall system performance may be improved if the lower priority portion of the ISR is actually executed at a lower priority level, allowing other more important interrupts to be serviced.

If the high priority ISR simply lowers its execution priority at the point where it enters the follow-up code, by writing a lower value to the IM bits in the PSW, a situation called “priority inversion” could occur. Priority inversion describes a case where code at a lower priority is executing while a higher priority routine is kept waiting. An example of how this could occur by writing to the IM bits follows, and is illustrated in Figure 4.15.

Suppose code is executing at level 0 and is interrupted by an event interrupt that runs at level 10. This is again interrupted by a level 12 interrupt. The level 12 ISR completes a time-critical portion of its code and wants to lower the priority of the remainder of its code (the non-time critical portion) in order to allow more important interrupts to occur. So, it writes to the IM bits, setting the execution priority to 5. The ISR continues executing at level 5 until a level 8 event interrupt occurs. The level 8 ISR runs to completion and returns to the level 5 ISR, which also runs to completion. When the level 5 ISR returns, the previously interrupted level 10 ISR is re-activated and eventually competes.

It can be seen in this example that lower priority ISR code executed and completed while higher priority code was kept waiting on the stack. This is priority inversion.

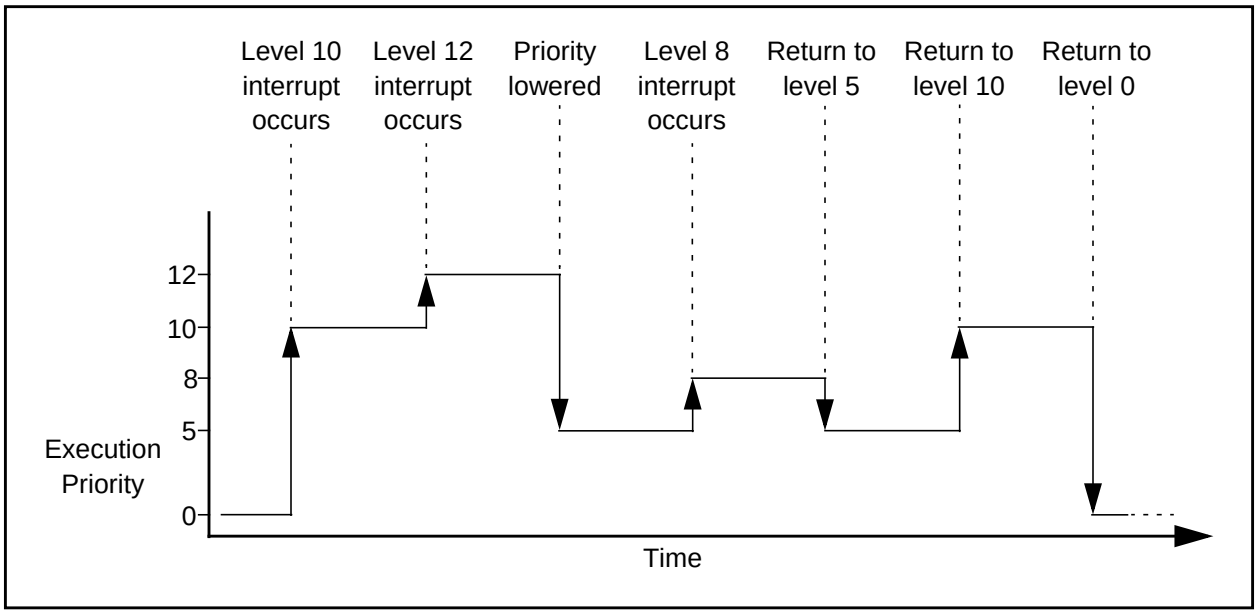


Figure 4.15 Example of priority inversion (see text)

In those cases where it is desirable to alter the priority level of part of an ISR, a software interrupt may be used to accomplish this without risk of priority inversion. The ISR must first be

split into 2 pieces: the high priority portion, and the lower priority portion. The high priority portion remains associated with the original interrupt vector. The lower priority portion is associated with the interrupt vector for software interrupt 5. At the completion of the high priority portion of the ISR, the code sets the request bit for software interrupt 5, then returns. the remainder of the ISR, now actually the ISR for software interrupt 5, executes when it becomes the highest priority pending interrupt.

The diagram in Figure 4.16 shows the same sequence of events as in the example of priority inversion, except using software interrupt 5 as just described. Note that the code now executes in the correct order (higher priority first).

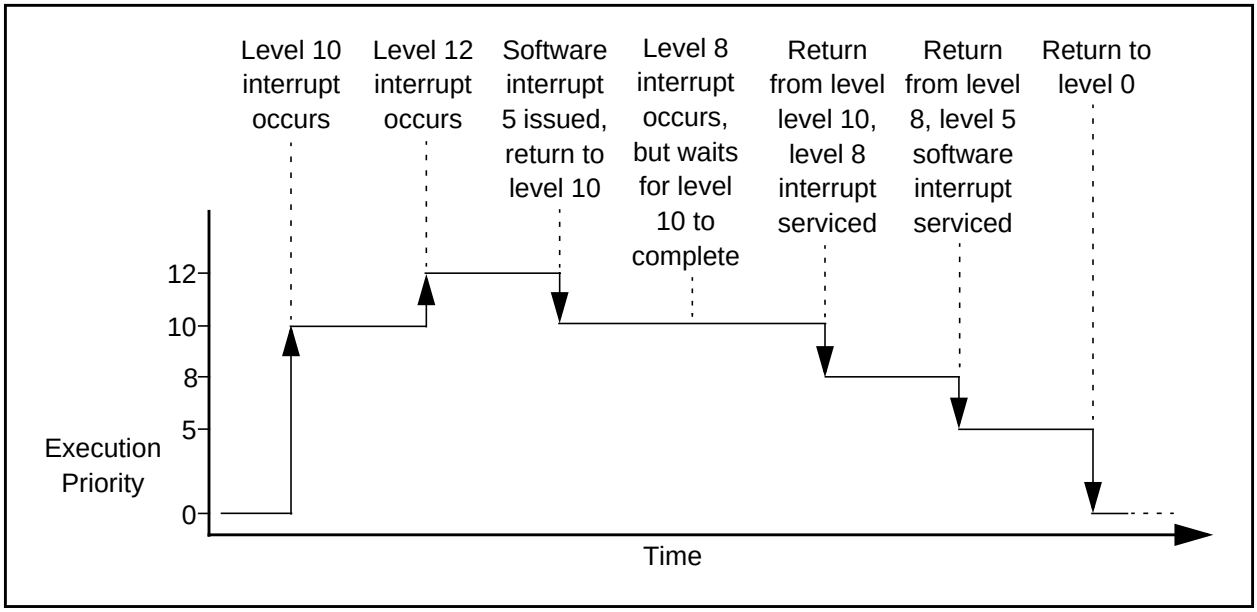


Figure 4.16 Example use of software interrupt (see text)

Trap Interrupts

Trap Interrupts are generated by the TRAP instruction. TRAP 0 through TRAP 15 are defined and may be used as required by applications. Trap Interrupts are intended to support application-specific requirements, as a convenient mechanism to enter globally used routines, and to allow transitions between user mode and system mode. A trap interrupt will occur if and only if the instruction is executed, so there is no need for a precedence scheme with respect to simultaneous traps.

The effect of a TRAP is immediate, the corresponding TRAP service routine is entered upon completion of the TRAP instruction.

See Chapter 6 for a detailed description of the TRAP instruction.

4.8.2 Interrupt Service Data Elements

There are two data elements associated with XA interrupts. The first is the stack frame created when each interrupt is serviced. The second is the interrupt vector table located at the beginning

of code memory. Understanding the structure and contents of each is essential to the understanding of how XA interrupts are processed.

Interrupt Stack Frame

A stack frame is generated, always on the System Stack, for each XA interrupt. With one exception, the stack frame is stored for the duration of interrupt service and used to return to and restore the CPU state of the interrupted code. (The exception is an Exception Interrupt triggered by a Reset event. Since Reset re-initializes the stack pointers, no stack frame is preserved. See section 4.4 for details.) The stack frame in the native 24-bit XA operating mode is illustrated in Figure 4.17. Three words are stored on the stack in this case. The first word pushed is the low-order 16 bits of the current PC, i.e., the address of the next instruction to be executed. The next word contains the high-order byte of the current PC. A zero byte is used as a pad. In sum, a complete 24-bit address is stored in the stack frame. The third word contains a copy of the PSW at the instant the interrupt was serviced.

When the XA is operating in Page 0 Mode (SCR bit **PZ** = 1) the stack frame is smaller because, in this mode, only 16 address bits are used throughout the XA. The stack frame in Page 0 Mode is illustrated in Figure 4.18. Obviously it is very important that stack frames of both sizes not be mixed; this is one reason for the admonition in section 4.3 to set the System Configuration Register once during XA initialization and leave it unchanged thereafter.

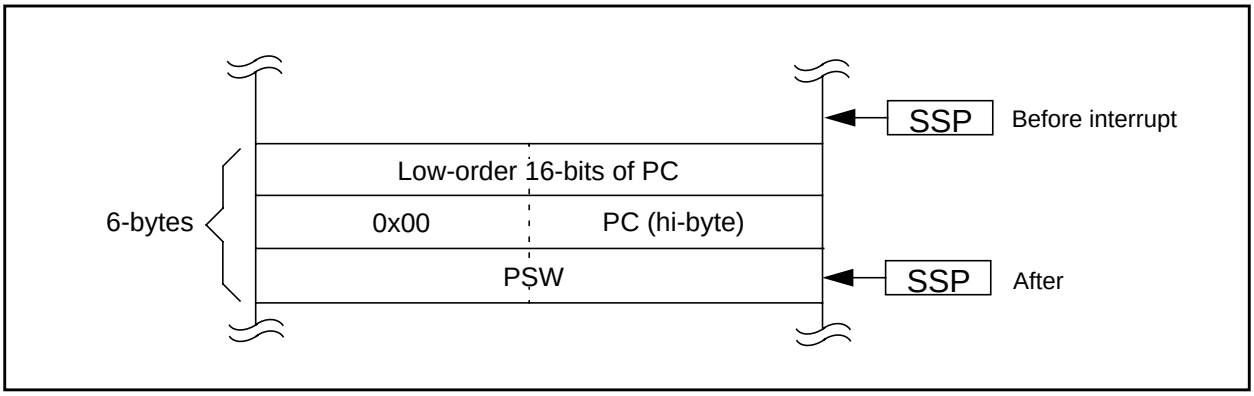


Figure 4.17 Interrupt stack frame (non- page zero mode)

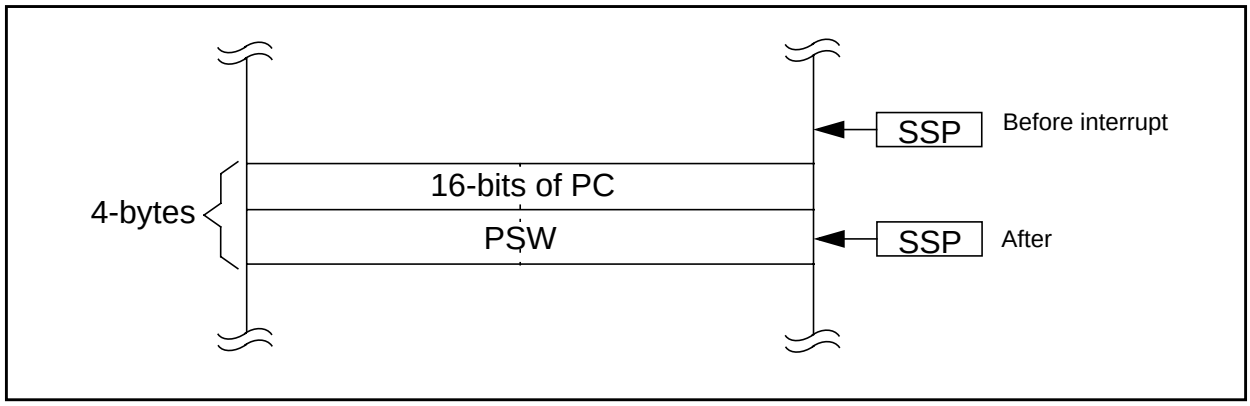


Figure 4.18 Interrupt stack frame (page 0 mode)

Interrupt Vector Table

The XA uses the first 284 bytes of code memory (addresses 0 through 11B hex) for an interrupt vector table. The table may contain up to 71 double-word entries, each corresponding to a particular interrupt event.

The double-word entries each consist of a 16 bit address of an interrupt service routine address and a 16 bit PSW replacement value. Because vector addresses are 16-bit, the first instruction of service routines must be located in the first 64K bytes of XA memory. The first instruction of all service routines must be word-aligned. Key elements of the replacement PSW value are the choice of System or User mode for the service routine, the Register Bank selection, and an Execution Priority setting. For more details on PSW elements, see section 4.2.2.

The first 16 vectors, starting at code memory address 0 are reserved for Exception Interrupt vectors. The second 16 vectors are reserved for Trap Interrupts. The following 32 vectors in the table are reserved for Event Interrupts. The final 7 vectors are used for Software Interrupts. Figure 4.19 illustrates the XA vector table and the structure of each component vector. Of the vectors assigned to Exceptions, 6 are assigned to events specific to the XA CPU and 10 are reserved. All 16 Trap Interrupts may be used freely. Assignments of Event Interrupt vectors are derivative-independent and vary with the peripheral device complement and pinout of each XA derivative.

Unused interrupt vectors should normally be set to point to a dummy service routine. The dummy service routine should clear the interrupt flag (if it is not self-clearing) and execute an RETI to return to the user program. This is especially true of the exception interrupts and NMI, since these could conceivably occur in a system where the designer did not expect them. If these vectors are routed to a dummy service routine, the system can essentially ignore the unexpected exception or interrupt condition and continue operation.

Note that when using some hardware development tools, it may be preferable not to initialize unused vector locations, allowing the development tool to flag unexpected occurrences of these conditions.

4.9 Trace Mode Debugging

The XA has an optional Trace Mode in which a special trace exception is generated at the conclusion of each instruction. Trace Mode supports user-supplied debugger/monitor programs which can single-step through any code, even code in ROM.

4.9.1 Trace Mode Operation

Trace Mode is initiated by asserting **PSW.TM** in the context of the program to be traced.

Using Trace Mode requires a detailed understanding of the XA instruction execution sequence because when and if a trace exception occurs depends on events within the execution sequence of a single instruction. Figure 4.20 illustrates the XA instruction sequence in overview.

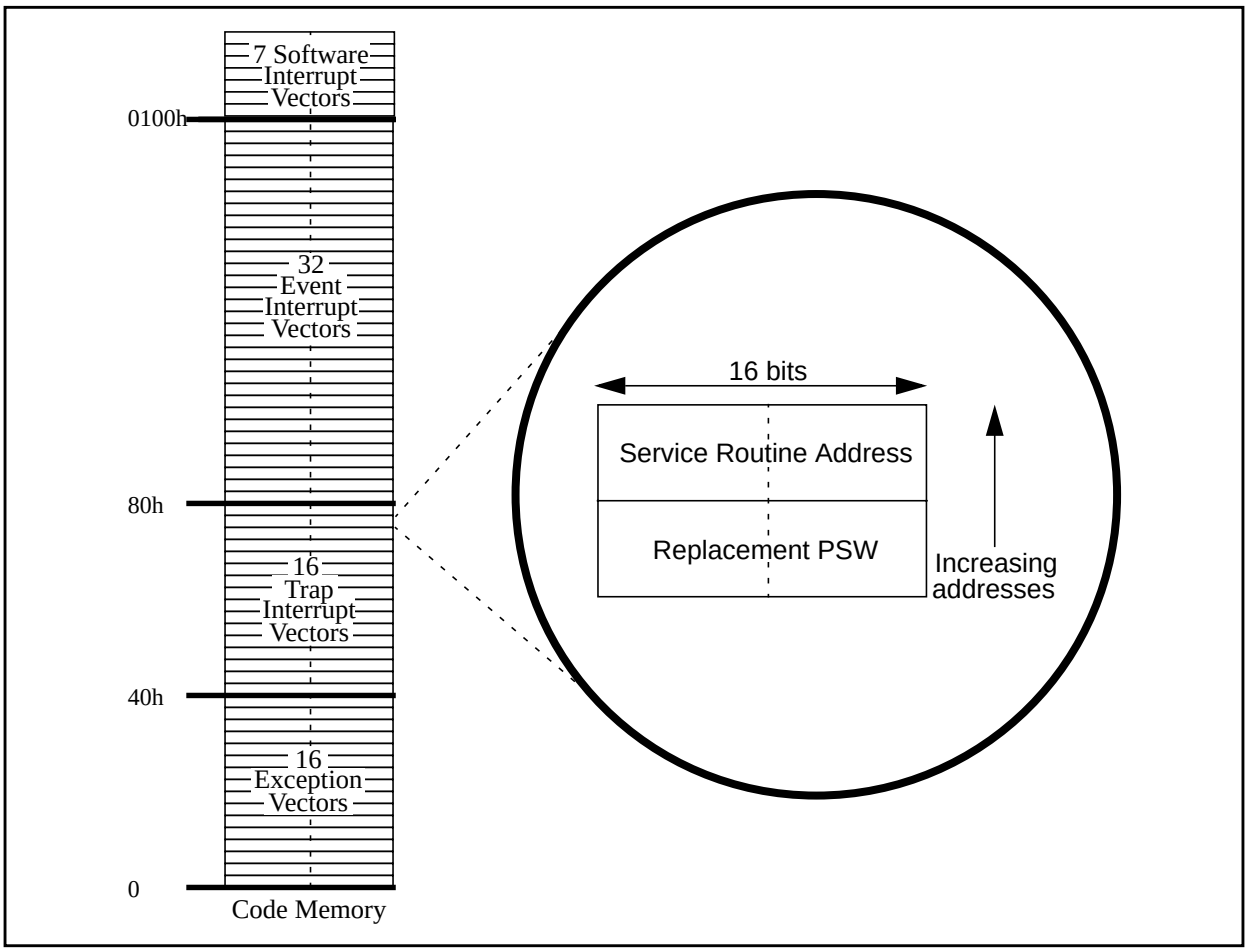


Figure 4.19 Interrupt vectors

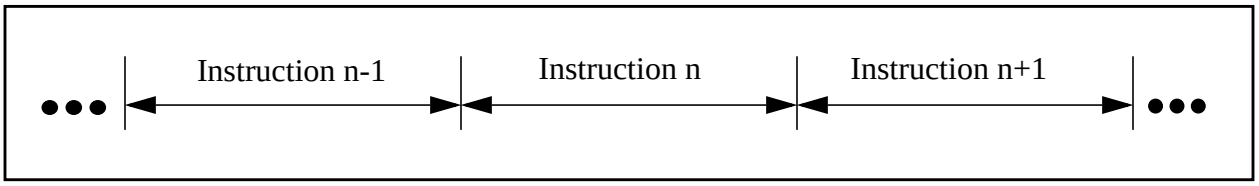


Figure 4.20 XA Instruction Sequence Overview

A detailed model of this sequence is shown in Figure 4.21: First, at the beginning of the instruction cycle, the state of the TM flag is latched. Next, the instruction is checked to see if it is valid; undefined instructions or disallowed operations (like a write through ES in User Mode) are simply not executed, and there is no chance for a trace to occur. The sequence then checks for instructions illegal in the current context (currently only an IRET while in User Mode is detected here) and services an exception if one is found. If, and only if, none of these special conditions occur, the instruction is actually executed. Just after execution, if the Trace Mode bit had been latched TRUE at the beginning of the instruction cycle, the Trace is serviced. Finally, the cycle checks for a pending interrupt and performs interrupt service if one is found
Note that an external reset may occur at any point during the cycle illustrated in Figure 4.21. This will abort processing when it occurs.

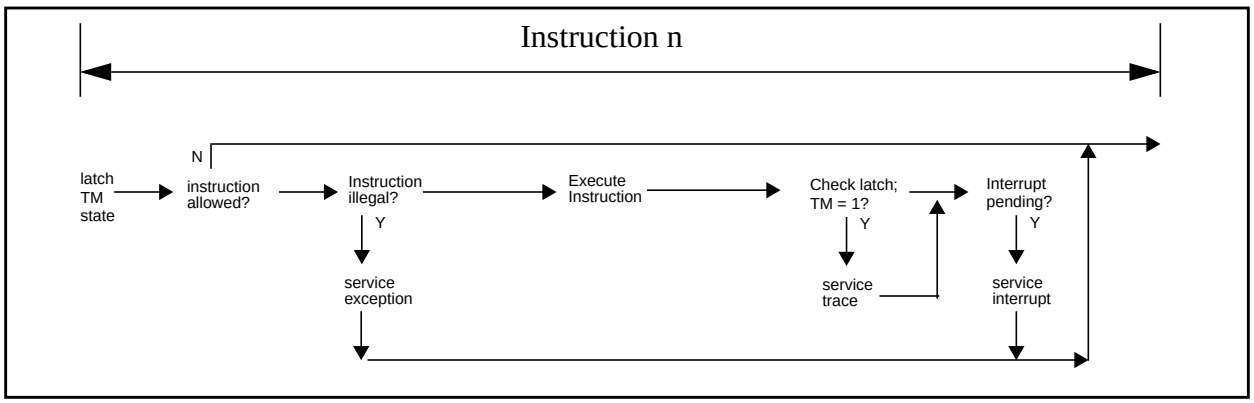
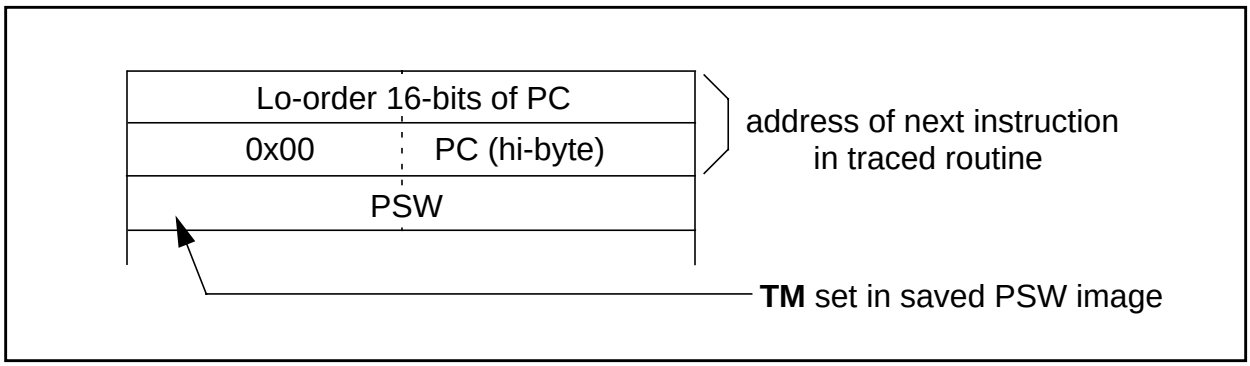


Figure 4.21 Instruction Execution Clock Detail

One consequence of this sequence is that the instruction that sets TM = 1 cannot generate a Trace, since TM is not latched when the instruction is actually executed. Another consequence is that an instruction that generates an exception will never be traced. Finally if an event interrupt occurs during an instruction clock when the instruction being executed is a TRAP, the TRAP will be executed, then the trace service, and finally the interrupt will be serviced.

4.9.2 Trace Mode Initialization and Deactivation

Since **PSW.TM** is in the protected portion of the PSW (i.e., in PSWH), only code executing in System Mode can initiate or turn off Trace Mode. In practice, this may be done by invoking a trap whose replacement PSW clears this bit, or by executing a RETI instruction with a synthetic Exception/Interrupt stack frame explicitly pushed on the top of the System Stack, as follows:



Tracing will continue until the PSW bit **TM** is cleared. This may be done by the trace service routine by examining the stack frame at the top of the system stack and clearing the TM bit prior to returning to the currently traced process. A similar method may be used to initiate trace mode. Note that stack frames generated by exception interrupts are always placed on the System stack. It is probably a good idea for the trace service routine to verify that the item in the stack frame is consistent with the traced process before modifying the TM bit.

5 Real-time Multi-tasking

Multi-tasking as the name suggests, allows tasks, which are pieces of code that do specific duties, to run in an apparently concurrent manner. This means that tasks will seem to all run at the same time, doing many specific jobs simultaneously.

High end applications (like automotive) require instantaneous responses when dealing with high speed events, such as engine management, traction control and adaptive braking system (ABS) and hence there is a trend towards multi-tasking in a wide variety of high performance embedded control applications.

Real-time application programs are often comprised of multiple tasks. Each task manages a specific facet of application program. Building a real-time application from individual tasks allows subdividing a complicated application program into independent and manageable modules. Each task shares the processor with other tasks in the application program according to an assigned priority level.

In real-time multi-tasking, the main concern is the *system overhead*. Switching tasks involve moving lots of data of the terminated and initiated tasks, and extensive book-keeping to be able to restore dormant tasks when required. Thus it is extremely crucial to minimize the system overhead as much as possible. In some cases, some of the tasks may be associated with real-time response, which further complicates the requirements from the system.

The following section analyzes the requirements and the XA suitability to these applications.

5.1 Multi-tasking Support in XA

The XA has numerous provisions to support multi-tasking systems. The architecture provides direct support for the concept of a multi-tasking OS by providing two (System/User) privilege levels for isolation between tasks. High performance, interrupt driven, multi-tasking applications systems requiring protection are feasible with the XA.

The XA architecture offers the following features which will appeal to multi-tasking implementations.

5.1.1 Dual stack approach

The architecture defines a System Stack Pointer (SSP) as well as an User Stack Pointer (USP). The dual stack feature supports fast task switching, and ease the creation of a multi-tasking monitor kernel. The separation of the two offers a reduction is storing and retrieving stack pointers or using a single stack, when switching to the kernel and back to an application. It also serves to speed up interrupt processing in large systems with external data memory. User stacks can be allocated in the slower external memory, while system memory is in internal SRAM, allowing for fast interrupt latency in this environment. The dual stack approach also adds the benefit of a better potential to recover from an ill-behaved task, since the system stack is still intact when an error is sensed.

5.1.2 Register Banks

The XA also supports 4 banks of 8 byte/4 word registers, in addition to 12 shared registers. In some applications, the register banks can be designated statically to tasks, cutting significantly on the overhead for saving and restoring registers on context switching.

5.1.3 Interrupt Latency and Overhead

Interrupt latency is extremely critical in a multitasking environment. For a real-time multitasking environment, a fast interrupt response is crucial for switching between tasks. The XA is designed to provide such fast task switching environment through improved interrupt latency time.

The interrupt service mechanism saves the PC (1 or 2 words, depending on the Page0 mode flag PZ) and the PSW (1 word) on the stack. The interrupt stack normally resides in the internal data memory, and interrupt call including saving of three words takes 23 clocks. Prefetching the service routine takes 3 additional clocks.

When interrupt or an exception/trap occurs, the current instruction in progress always gets executed prior servicing the interrupt. This present an overhead, while increasing the effective interrupt latency, since the event that interrupted the machine cannot be dealt with before the book-keeping is completed. In XA, the longest uninterrupted instruction is the signed 32x16 Divide, which takes 24 clocks.

This puts the worst case interrupt latency at [24 + 23 + 3] = 50 clocks (3.125 microseconds at 16.0 MHz, 2.5 microseconds at 20.0 MHz, and 1.67 microseconds at 30.0 MHz). Saving the state of the lower registers can be done by simply switching the register bank.

In the general case, up to 16 registers would be saved on the stack, which takes 32 clocks. The total latency+overhead at start of an interrupt is a maximum of 68 clocks (4.25 microsecond at 16 MHz, 3.4 at 20 MHz and 2.27 at 30 MHz). This allows for extremely fast context switching for multitasking environments.

5.1.4 Protection

The issue is mentioned here simply to clarify what is and what is not supported by the XA architecture. Dual stack pointer and minor privileges to what looks like a supervisor mode do not mean full protection. It is assumed that code in a microcontroller does not require guarding from intentional system break-in by a lower privilege task. A table of the protected features in XA is given below. Note that features marked “disallowed” are simply not completed if attempted in the User mode. There are no exceptions or flags associated with these occurrences.

Protected Features in the XA

Table 5.1: Segment and Stack Register Protection

Mode	Write to DS	Write through DS	Write to ES	Write through ES	Read through DS	Read through ES	Read through SSP	Write to SSP	Write to SSEL bit 7
System	Allowed	Allowed	Allowed	Allowed	Allowed	Allowed	Allowed	Allowed	Allowed
User	Dis-allowed	Allowed	Allowed	Select-able [1]	Allowed	Allowed	Not possible	Not possible	Dis-allowed

Note 1: The MSB of SSEL (bit 7) selects whether write through ES is allowed in User mode. However, this bit is accessible only in System mode.

Table 5.2: PSW bit protection

Mode	Write to SM bit	Write to RS0:1 bits	Write to TM bit	Write to IM0:3 bits
System	Allowed	Allowed	Allowed	Allowed
User	Disallowed	Allowed	Disallowed	Disallowed

In addition to the above, the System Stack is protected from corruption by User Mode execution of the RETI instruction. If User Mode code attempts to execute that instruction, it causes an exception interrupt. If it is necessary to run TRAP routines, for instance, in User Mode, the User RETI exception handler can perform the return for the User Mode code. To accomplish this, the User RETI exception handler may pop the topmost return address from the stack (2 or 3 words, depending on whether the XA is in Page Zero mode) and then execute the RETI.

Protection Via Data Memory Segmentation

In User/Application mode, each task is protected from all others via the separation of data spaces (unless explicit sharing is planned in advance). If the address spaces of two tasks include no shared data, one task cannot affect the data of another, but it can read any data in the full address space. Code sharing is always safe since code memory may never be written[1]. An application mode program is prohibited from writing the segment registers, thus confining the writable area per an ill-behaved task to its dedicated segment. Most applications, which are not expected to utilize multi-tasking or use external memory, do not require any protection. They will remain after reset in system mode, and could access all system resources.

At any given instant, two segments of memory are immediately accessible to an executing XA program. These are the data segment DS, where the stack and local variables reside, and the extra segment ES, which may be used to read remote data structures. Restricting the addressability of task modules helps gain complete control of system resources for efficient, reliable operation in a multi-tasking environment.

---

1. True for non-writable code memory only like EPROM, ROM, OTP. This might change for FLASH parts.

Protection Via Dual Stack Pointers

The XA provides a two-level user/supervisor protection mechanism. These are the *user* or *application* mode and the *system* or *supervisor* mode. In a multitasking environment, tasks in a supervisor level are protected from tasks in the application level.

The XA has two stack pointers (in the register file) called the System Stack Pointer (SSP) and the User Stack Pointer (USP). In multitasking systems one stack pointer is used for the supervisory system and another for the currently active task. This helps in the protection mechanism by providing isolation of system software from user applications. The two stack pointers also help to improve the performance of interrupts. If the stack for a particular application would exceed the space available in the on-chip RAM, or on-chip RAM is needed for other time critical purposes (since on-chip RAM is accessed more quickly than off-chip memory), the main stack can be put off-chip and the interrupt stack (using the System SP) may be put in on-chip RAM.

These features of the XA place it well above the competition in suitability to multi-tasking applications.

6 Instruction Set and Addressing

This section contains information about the addressing modes and data types used in the XA. The intent is to help the user become familiar with the programming capabilities of the processor.

6.1 Addressing Modes

Addressing modes are ways to form effective addresses of the operands. The XA provides seven *basic* powerful addressing modes for access on word, byte, and bit data, or to specify the target address of a branch instruction. These *basic* addressing modes are uniformly available on a large number of instructions. Table 6.1 includes the basic addressing modes in the XA. An instruction could use a combination of these basic addressing modes, e.g., ADD R0, #020 is a combination of Register and Immediate addressing modes.

All modes (non-register) generate ADDR[15:0]. This address is combined with DS/ES[23:16] for data and PC/CS[23:16] for code to form a 24-bit address[1].

An XA instruction can have zero, one, two, or three operands, whose locations are defined by the addressing mode. A *destination* operand is one that is replaced by a result, or is in some way affected by the instruction. The destination operand is listed first in an addressing mode expression. A *source* operand is a value that is moved or manipulated by the instruction, but is not altered. The source is listed second in an addressing mode expression.

Table 6.1 Basic Addressing Modes

MODE	MNEMONIC	OPERANDS
Register	R	operand(s) in register (in Register file)
Indirect	[R]	Byte/Word whose 16-bit address is in R
Indirect-Offset	[R+off 8/16]	Byte or Word data whose address (16-bit) contained in R, is offset by 8/16-bit signed integer “off 8/16’
Direct	mem_addr	Byte/Word at given memory “mem_addr’
SFR [1]	sfr_addr	Byte/Word at “sfr_addr’ address
Immediate	#data 4/5 #data 8/16	Immediate 4/5 and 8/16-bit integer constants “data8/16”
Bit	bit	10-bit address field specifying Register File, Data Memory or SFR bit address space

1. This is a special case of direct addressing mode but separately identified, as SFR space is separate from data memory.

1. Exception is Page 0 mode, where all addresses are 16-bit.

6.2 Description of the Modes

6.2.1 Register Addressing

Instructions using this addressing mode contain a field that addresses the Register File that contains an operand. The Register file is byte[2], word, double-word or bit addressable.

Example: ADD R6, R4

Before: R4 contains 005Ah
R6 contains A5A5h

After: R4 contains 005Ah
R6 contains A5FFh

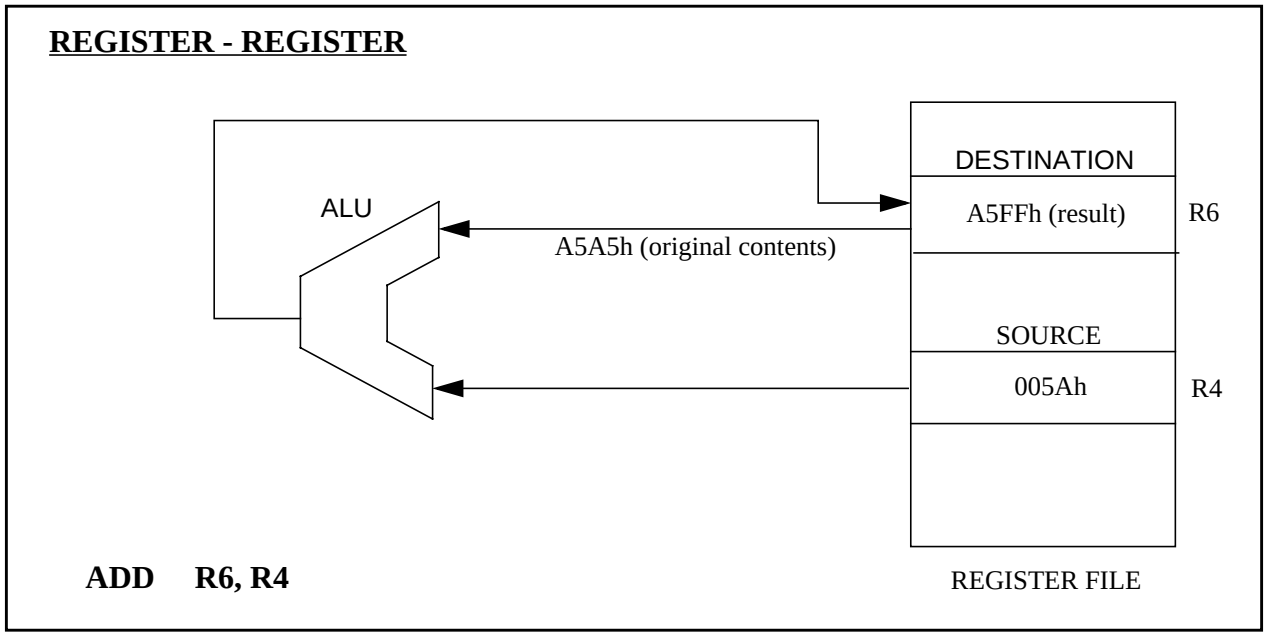


Figure 6.1

2. The unimplemented 8 word registers are not Byte addressable

6.2.2 Indirect Addressing

Instructions using this addressing mode contain a 16-bit address field. This field is contained in 1 out of 8 pointer registers in the Register File (that contain the 16-bit address of the operand in any 64K data segment). For data, the segment is identified by the 8-bit contents of DS or the ES and for code by the 8-bit contents of PC23-16 or CS as selected by the appropriate bit (SSEL.bit n = 0 selects DS and 1 selects ES for data and SSEL.bitn = 0 selects PC and 1 selects CS for code) in the segment select register SSEL corresponding to the indirect register number. The address of the pointer word for word operands should be even

Example:	ADD R6, [R4]	Before**:**	R6 contains 1005h
	SSEL.4 = 1		R4 contains A000h
	i.e., the operand is in		Word at A000h contains A5A5h
	segment determined		
	by the contents of ES	After:	R4 contains A000h
	So, if ES = 08, the		R6 contains B5AAh
	operand is in		Word at A000h in segment 8
	segment 8 of data memory.		of data memory contains A5A5h

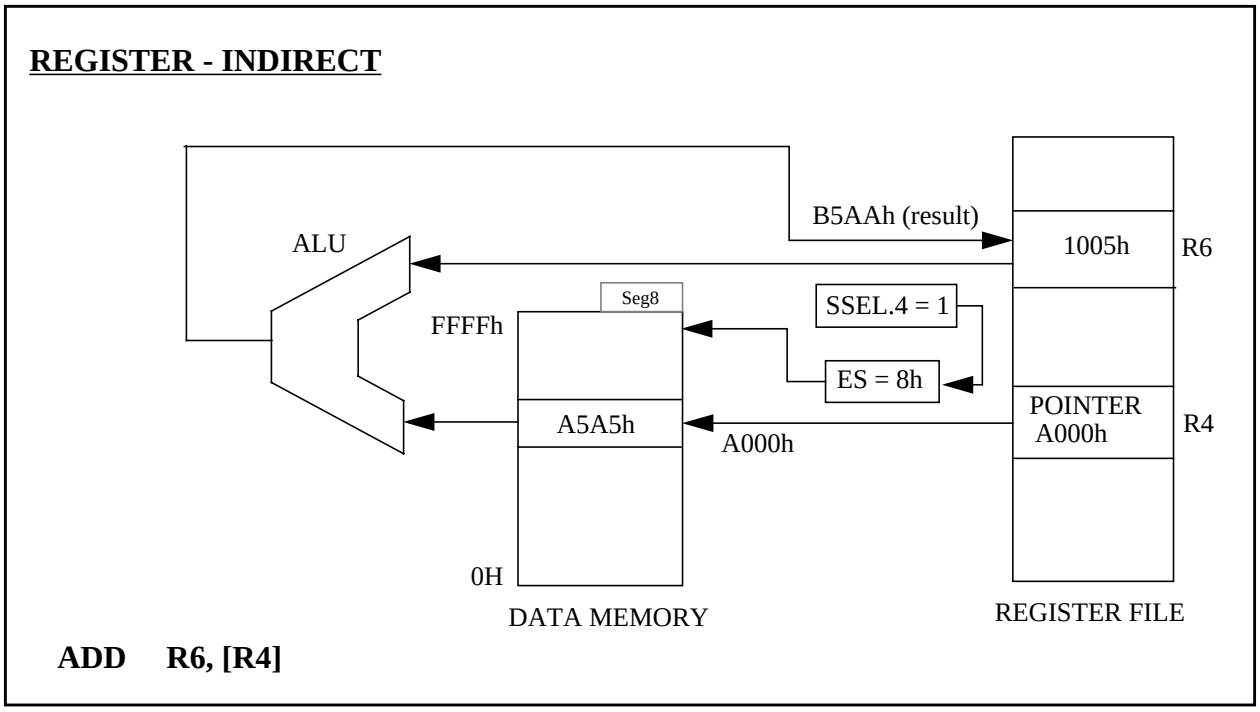


Figure 6.2

6.2.3 Indirect-Offset Addressing

This addressing mode is just like the Register-Indirect addressing mode above except that an additional displacement value is added to obtain the final effective address. Instructions using this addressing mode contain a 16-bit address field and an 8 or 16-bit signed displacement field. This field addresses 1 out of 8 pointer registers in the Register File that contains the 16-bit address of the operand in any 64K data segment. The contents of the pointer register are added to the signed displacement to obtain the effective address[3] (which *must* be even) of the operand. For data the segment is identified by the 8-bit contents of DS or the ES and for code, by the 8-bit contents of PC23-16 or CS as selected by the appropriate bit (SSEL.bit n = 0 selects DS and 1 selects ES for data and SSEL.bitn = 0 selects PC and 1 selects CS for code) in the segment select register SSEL.

Example:	ADD R5, [R3 +30h]	Before:	R3 contains C000h
	SSEL.3 = 1		R5 contains 0065h
	i.e., the operand is in		Word at C030h = A540h
	segment determined		
	by the contents of ES	After:	R3 contains C000h
	So, if ES = 04, the		R5 contains A5A5h
	operand is in segment		Word at C030h = A540h
	4 of data memory.		

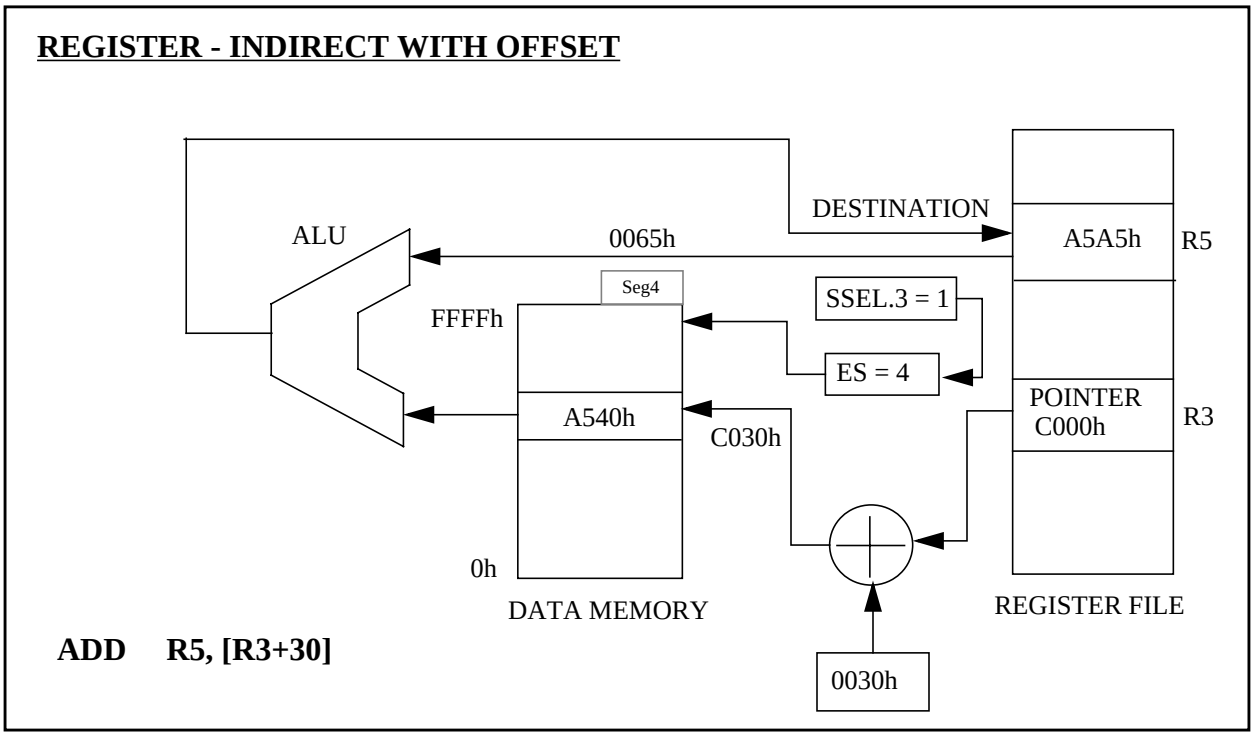


Figure 6.3

3. In case of an odd address, the XA forces the operand fetch from the next lower even boundary (address.bit0 = 0)

6.2.4 Direct Addressing

Instructions using this addressing mode contain an 10-bit address field, which contains the actual address of the operand in any 64K data memory segment or sfr space.The direct address data memory space is always the bottom 1K byte (0:3FFh) of any segment. The associated data segment is always identified by the 8-bit contents of DS.

Example: SUB R0, 200h
If DS = 02, the operand is in segment 2 of data memory.

Before: R0 contains A5FFh
200H contains 5555h

After: R0 contains 50AAh
200h contains 5555h

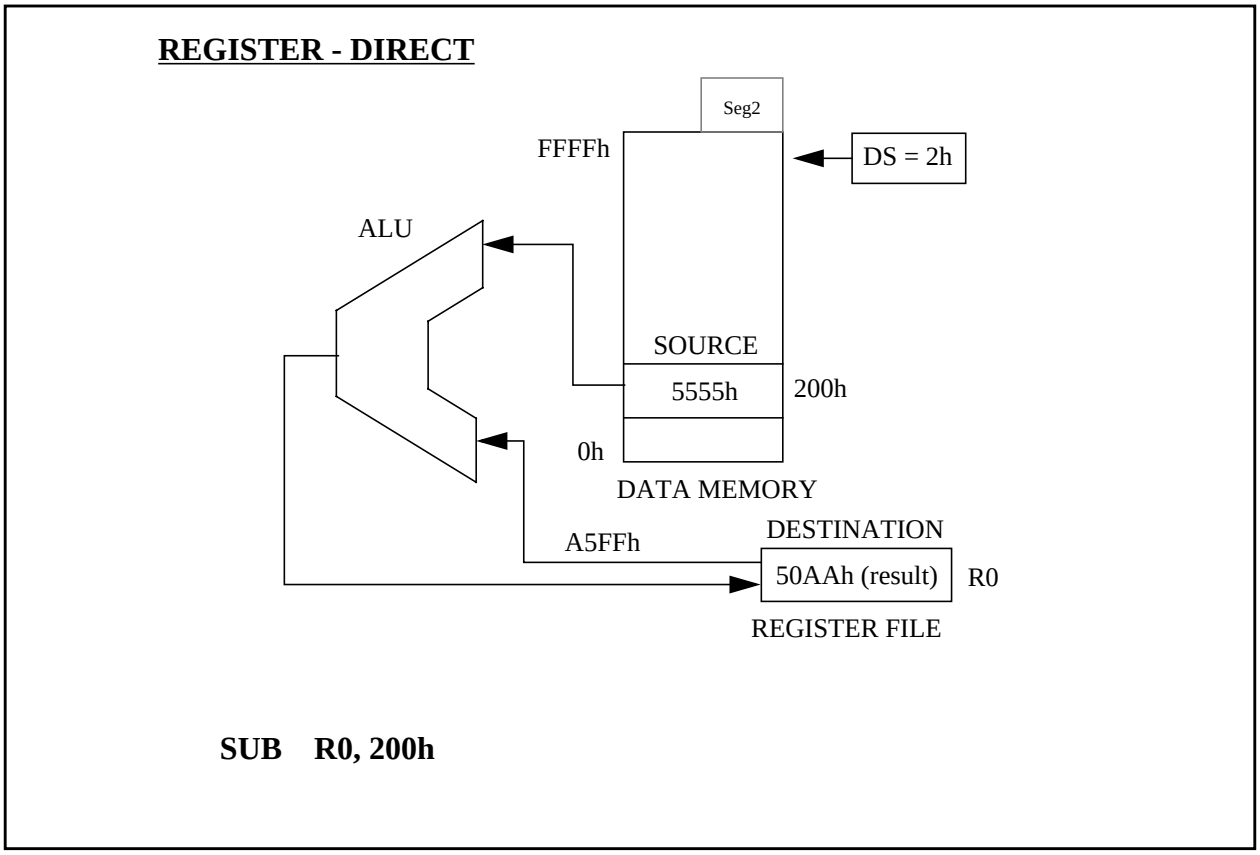


Figure 6.4

6.2.5 SFR Addressing

This is identical to the direct addressing mode described before, except it addresses the 1K SFR space. Although encoded into the same instruction field as the direct addressing described above, this is actually a separate space. Instructions using this addressing mode contain an 10-bit SFR address. The 1K SFR space is always directly addressed (400:7FFh) and is mapped directly above the 1K direct-addressed RAM space.

Example:	MOV R0H, 406h[4]	Before:	R0H contains 05h 406h contains A5h
		After:	R0H contains A5h 406h contains A5h

6.2.6 Immediate Addressing

In immediate addressing, the actual operand is given explicitly in the instruction.The immediate operand is either an 4/5, 8 or 16-bit integer which constitutes the source operand. 4-bit short immediate operands used with instructions ADDS and MOVS are sign extended.

Example:	ADD R0L,#0B9h	Before:	R0 contains 13h
		After:	R0L contains CCh

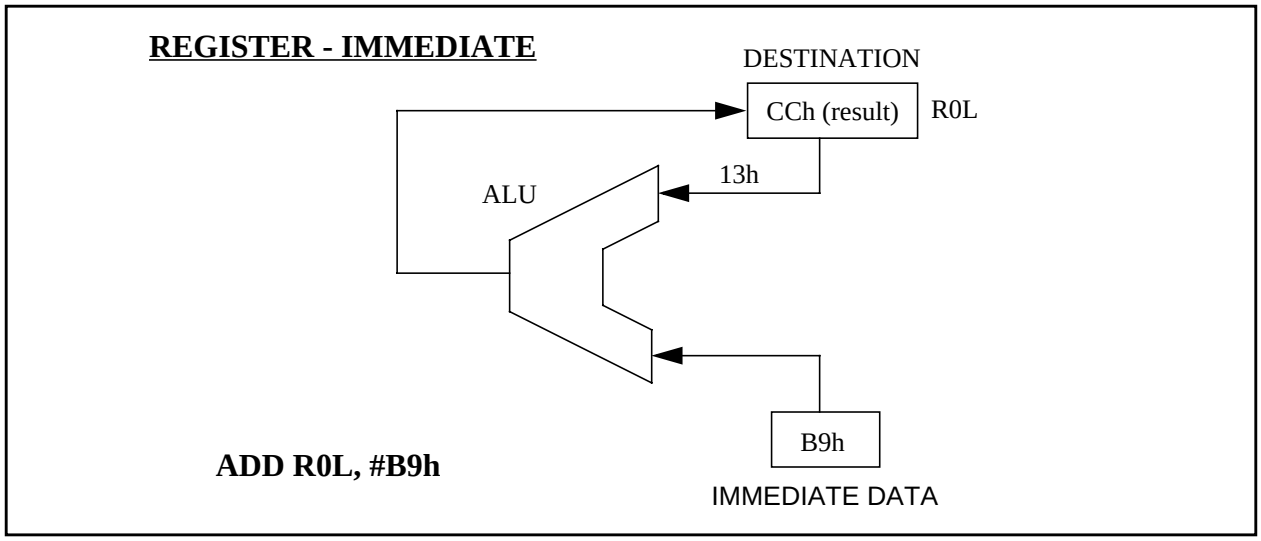


Figure 6.5

4. The syntax always refers to the SFR address starting from the base address of 400H.

6.2.7 Bit Addressing

Instructions using the bit addressing mode contain a 10-bit field containing the address of the bit operand. The XA supports three bit address spaces, which are encoded into the same format. The spaces are: 256 bits in the register file (the entire active register file); 256 bits in the data memory (byte addresses 20 through 3F hex on the current data segment); and 512 bits in the SFR space (byte addresses 400 through 43F hex).

Bit addresses 0 to FF hex map to the register file, bit addresses 100 to 1FF hex map to data memory, and bit addresses 200 to 3FF map to the SFR space.

A separate bit-addressable space (20-3F hex) in the direct-address data memory, exists for each segment. The current working segment for the direct-address space being always identified by the DS register.

The encoding of the 10-bit field for bit addresses is as follows:

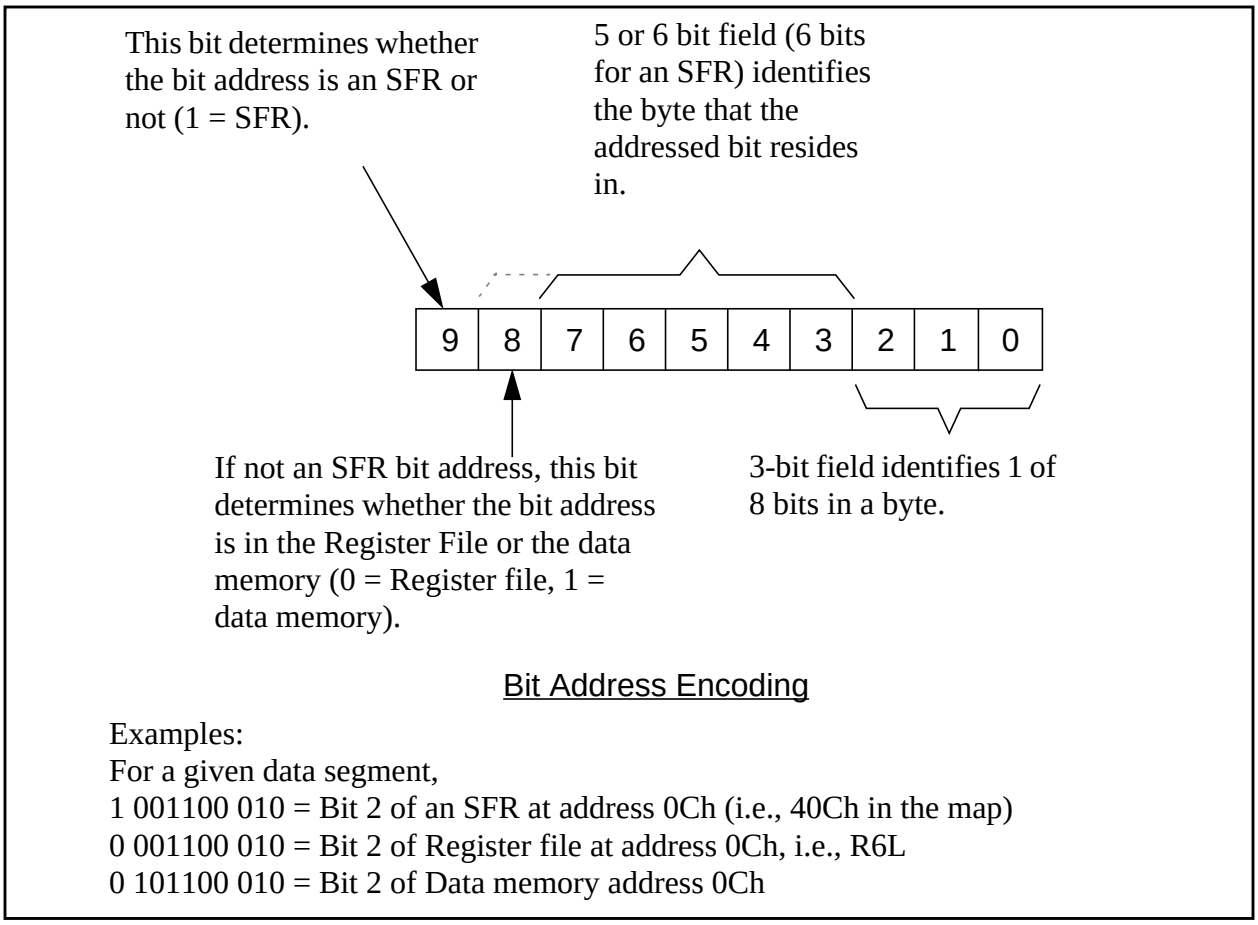


Figure 6.6

6.3 Relative Branching and Jumps

Program memory addresses as referenced by Jumps, Calls, and Branch instructions must be word aligned in XA. For instance, a branch instruction may occur at any code address, but it may only branch to an even address. This forced alignment to even address provides three benefits:

- Branch ranges are doubled without providing an extra bit in the instruction and
- Faster execution as XA always fetches first two byte of an instruction simultaneously.
- Allows translated 8051 code to have branches extended over intervening code that will tend to grow when translated and generally increase the chances of a branch target being in that range.

The *rel8* displacement is a 9-bit two's complement integer which is encoded as 8-bits that represents the relative distance in words from the current PC to the destination PC. Similarly, the *rel16* displacement is a 17-bit twos complement integer which is encoded as 16-bits. The value of the PC used in the target address calculation is the address of the instruction following the Branch, Jump or Call instruction.

The 8-bit signed displacement is between -128 to +127. The branch range for rel8 is (sample calculation shown below) is really +254 bytes to -256 bytes for instructions located at an *even* address, and +253 to -257 for the same located at an *odd* address, with the limitation that the target address is word aligned in code memory.

The 16-bit signed displacement is -32,768 to +32,767. The branch range is therefore +65,534 bytes to -65,536 bytes for instructions located at an *even* address, and +65,533 to -65,537 for the same located at an *odd* address, with the limitation that the target address is word aligned in code memory.

Sample calculation for rel8 range:

Assuming word aligned branch target, forward range as measured from current PC is:

Branch Target Address - Current PC
Now, maximum positive signed 8-bit displacement = +127; So, rel8 << 1 is +254

If Current PC = ODD, then
Range = 254 - 1 = +253 as PC is forced to an even location, else
If current PC = EVEN, then
Range = +254

Similarly, reverse range as measured from current PC is:

Branch Target Address - Current PC
Now, maximum positive signed 8-bit displacement = -128; So, rel8 << 1 is -256

If Current PC = ODD, then
Range = -257
Else if current PC = EVEN, then
Range = -256

6.4 Data Types in XA

The XA uses the following types of data:

- Bits
- 4/5-bit signed integers
- 8-bit (byte) signed and unsigned integers
- 8-bit, two digit BCD numbers
- 16-bit (word) signed and unsigned integers
- 10-bit address for bit-addressing in data memory and SFR space
- 24-bit effective address comprising of 16-bit address and 8-bit segment select. See addressing modes for more information.

A byte consists of 8-bits. A word is a 16-bit value consisting of two contiguous bytes. A double word consists of two 16-bit words packed in two contiguous words in memory.

Negative integers are represented in twos complement form. 4-bit signed integers (sign extended to byte/word) are used as immediate operands in MOVS and ADDS instructions.

Binary coded decimal numbers are packed, 2 digits per byte. BCD operations use byte operands.

6.5 Instruction Set Overview

The XA uses a powerful and efficient instruction set, offering several different types of addressing modes. A versatile set of “branch” and “jump” instructions are available for controlling program flow based on register or memory contents. Special emphasis has been placed on the instruction support of structured high-level languages and real-time multi-tasking operating systems.

This section discusses the set of instructions provided in the XA microcontroller, and also shows how to use them. It includes descriptions of the instruction format and the operands used by the instructions. After a summary of the instructions by category, the section provides a detailed description of the operation of each instruction, in alphabetical order.

Five summary tables are provided that describes the available instructions. The first table is a summary of instructions available in the XA along with their common usage. The second and third table are tables of addressing modes and operands, and the instruction type they pertain to. A fourth table that lists the summary of status flags update by different instructions. A fifth table lists the available instructions with their different addressing modes and briefly describes what each instruction does along with the number of bytes, and number of clocks required for each instruction.

The formats have been chosen to optimize the length and execution speed of those instructions that would be used the most often in critical code. Only the first and sometimes the second byte of an instruction are used for operation encoding. The length of the instruction and the first execution cycle activity are determined from the first byte. Instruction bytes following the first two bytes (if any) are always immediate operands, such as addresses, relative displacements, offsets, bit addresses, and immediate data.

Glossary of mnemonics, notations used

General:

offset8	An 8-bit signed offset (immediate data in the instruction) that is added to a register to produce an absolute address.
offset16	A 16-bit signed offset (immediate data in the instruction) that is added to a register to produce an absolute address.
direct	An 11-bit immediate address contained in the instruction.
#data4	4 bits of immediate data contained in the instruction. (range +7 to -8 for signed immediate data and 0-15 for shifts)
#data5	5 bits of immediate data contained in the instruction. (0-31 for shifts)
#data8	8 bits of immediate data contained in the instruction. (+127 to -128)
#data16	16 bits of immediate data contained in the instruction. (+32,767 to -32,768)
bit	The 10-bit address of an addressable bit.
rel8	An 8-bit relative displacement for branches. (+254 to -256)
rel16	An 16-bit relative displacement for branches.(+65,534 to -65,536)
addr16	A 16-bit absolute branch address within a 64K code page.
addr24	A 24-bit absolute branch address, able to access the entire XA address space.
SP	The current Stack Pointer (User or System) depending on the operation mode.
USP	The User Stack Pointer.
SSP	The System Stack Pointer
C	Carry flag from the PSW.
AC	Auxiliary Carry flag from the PSW.
V	Overflow flag from the PSW.
N	Negative flag from the PSW.
Z	Zero flag from the PSW.
DS	Data segment register. Holds the upper byte of the 24-bit data address space of the XA. Used mainly for local data segments.
ES	Extra segment register. Holds the upper byte of the 24-bit data address space of the XA. Used mainly for addressing remote data structures.

direct	Uses the current DS for data memory for the upper byte of the 24-bit address or none (uses only the low 16-bit address) for accessing the special functions register (SFR) space. The interpretation should be as below:
	if (data range)
	then (direct = (DS:direct)
	if (sfr range)
	then (direct) = (sfr)

Operation encoding fields:

SZ	Data Size. This field encodes whether the operation is byte, word or double-word.
IND	This field flags indirect operation in some instructions.
H/L	This field selects whether PUSH and POP Rlist use the upper or lower half of the available registers.
dddd	Destination register field, specifies one of 16 registers in the register file.
ddd	Destination register field for indirect references, specifies one of 8 pointer registers in the register file.
ssss	Source register field, specifies one of 16 registers in the register file.
sss	Source register field for indirect references, specifies one of 8 pointer registers in the register file.

Mnemonic text:

Rs	Source register.
Rd	Destination register.
[ ]	In the instruction mnemonic, indicates an indirect reference (e.g.: [R4] refers to the memory address pointed to by the contents of register 4).
[R+]	Used to indicate an automatic increment of the pointer register in some indirect addressing modes.
[WS:R]	Indicates that the pointer register (R) is extended to a 24-bit pointer by the selected segment register (either DS or ES for all instructions except MOVC, which uses either PC_{23-16} or CS).
Rlist	A bitmap that represents each register in the register file during a PUSH or POP operation. These registers are R0-R7 for word or R0L-R7H for byte.

Pseudocode:

( )	Used to indicate "contents of" in the instruction operation pseudocode (e.g.: (R4) refers to the contents of register 4).
<---	Pseudocode assignment operator. Occasionally used as <--> to indicate assignment in both directions (interchange of data).
((SP))	Data memory contents at the location pointed to by the current stack pointer. In system mode, the current SP is the SSP, and the segment used is always segment 0. In user mode, the current SP is the USP, and the segment used is the Data Segment (DS). This segment apply to the uses of the SP, not just PUSH and POP. In a few cases, “((SSP))” or “((USP))” indicate that a specific SP is used, regardless of the operating mode.
Rn.x	Indicates bit x of register n.
Rn.x-y	Indicates a range of bits from bit x to bit y of register n.

Note: all indirect addressing is accomplished using the contents of the data segment register as the upper 8 address bits unless otherwise specified. Example: [ES:Rs] indicates that the extra segment register generates the upper 8 bits of the address in this case.

Execution time:

PZ	- In Page 0
nt	- Not Taken
t	- Taken

Syntax For Operand size:
.w = For word operands
.b = byte operands
.d = double-word operands

Default operand size is dependant on the operands used e.g MOV R0,R1 is always word-size whereas MOV R0L, R0H is always byte etc. For INDIRECT_IMMEDIATE, DIRECT_IMMEDIATE, DIRECT_DIRECT, etc., user must specify operand size.

Others

0x = prefix for Hex values
[] = For indirect addressing
[[]] = For Double-indirect addressing
dest = destination
src = source

Table 6.2 Instruction Set in XA

Mnemonic	Usage
MOV, MOVC, MOVS, MOVX, LEA, XCH, PUSH, POP, PUSHU, POPU	Data Movement
ADD, ADDS, ADDC, SUB, SUBB	Add and Subtract
MULU.b, MULU.w, MUL.w DIVU.b, DIVU.w, DIVU.d, DIV.w, DIV.d	Multiply and Divide
RR, RRC, RL, RLC, LSR, ASR, ASL, NORM	Shifts and Rotates
CLR, SETB, MOV, ANL, ORL	Bit Operations
JB, JBC, JNB, JNZ, JZ, DJNZ, CJNE,	Conditional Jumps/Calls
BOV, BNV, BPL, BCC, BCS, BEQ, BNE, BG, BGE, BGT, BL, BLE, BLT, BMI	Conditional Branches
AND, OR, XOR	Boolean Functions
JMP, FJMP, CALL, FCALL, BR	Unconditional Jumps/Calls/Branches
RET, RETI	Return from subroutines, interrupts
SEXT, NEG, CPL, DA	Sign Extend, Negate, Complement, Decimal Adjust
BKPT, TRAP#, RESET	Exceptions
NOP	No Operation

Table 6.3 shows a summary of the basic addressing modes available for data transfer and calculation related instructions.

Table 6.3 Instruction Addressing Modes

Modes/ Operands	MOVX	MOV	CMP	ADD ADDC	SUB SUBB	AND OR XOR	ADDS MOVS	MUL DIV	Shift	XCH	bytes
R, R		•	•	•	•	•		•	•	•	2
R, [R]	•	•	•	•	•	•				•	2
[R], R	•	•	•	•	•	•					2
R, [R+off8]		•	•	•	•	•					3
[R+off8], R		•	•	•	•	•					3
R, [R+off16]		•	•	•	•	•					4
[R+off16], R		•	•	•	•	•					4
R, [R+]		•	•	•	•	•					2
[R+], R		•	•	•	•	•					2
[R+], [R+]		•									2
dir, R		•	•	•	•	•					3
R, dir		•	•	•	•	•				•	3
dir, [R]		•									3
[R], dir		•									3
R, #data		•	•	•	•	•	•	•	•		2*/3/4
[R], #data		•	•	•	•	•	•				2*/3/4
[R+], #data		•	•	•	•	•	•				2*/3/4
[R+off8], #data		•	•	•	•	•	•				3*/4/5
[R+off16], #data		•	•	•	•	•	•				4*/5/6
dir, #data		•	•	•	•	•	•				3*/4/5
dir, dir		•									4
R, USP		•									2

Notes:
- Shift class includes rotates, shifts, and normalize.
- USP = User stack pointer.
* : ADDS and MOVS uses a short immediate field (4 bits).
** instructions with no operands include: BKPT, NOP, RESET, RET, RETI.

Modes/ Operands	**MOVC**	PUSH POP	DA, SEXT CPL, NEG	JUMP CALL	DJNZ	CJNE	BIT OPS	MISC	bytes
R, [R+]	•								2
[R+], R	•								2
A, [A+DPTR]	•								2
A, [A+PC]	•								2
direct		•							3
Rlist		•							2
R			•						2
addr24				•					4
[R]				•					2
[A+DPTR]				JMP					2
R, rel					•				3
direct, rel					•				4
R, direct, rel						•			4
R, #data, rel						•			4/5
[R], #data, rel						•			4/5
bit							•		3
bit, C; C, bit							•		3
C, /bit							•		3
rel				•				Cond. Branch	2
bit, rel								Cond. Branch	4
#data4								TRAP	2
R, R+off8								LEA	3
r, R+off16								LEA	4
<none> **								•	1/2

Notes:
- Shift class includes rotates, shifts, and normalize.
- USP = User stack pointer.
* : ADDS and MOVS uses a short immediate field (4 bits).
** instructions with no operands include: BKPT, NOP, RESET, RET, RETI.

Table 6.4 summarizes the status flag updates for the various XA instruction types.

Table 6.4 Status Flag Updates

Instruction Type	Flags Updated				
	C	AC	V	N	Z
ADD, ADDC, CMP, SUB, SUBB	X	X	X	X	X
ADDS, MOVS	-	-	-	X	X
AND, OR, XOR	-	-	-	X	X
ASR, LSR	*	-	-	X	X
branches, all bit operations, NOP	-	-	-	-	-
Calls, Jumps, and Returns	-	-	-	-	-
CJNE	X	-	-	X	X
CPL	-	-	-	X	X
DA	*	-	-	X	X
DIV, MUL	*	-	*	X	X
DJNZ	-	-	-	X	X
LEA	-	-	-	-	-
MOV, MOVC, MOVX	-	-	-	X	X
NEG	-	-	X	X	X
NORM	-	-	-	X	X
PUSH, POP	-	-	-	-	-
RESET	*	*	*	*	*
RL, RR	-	-	-	X	X
RLC, RRC	*	-	-	X	X
SEXT	-	-	-	-	-
TRAP, BKPT	-	-	-	-	-
XCH	-	-	-	-	-
ASL	*	-	X	X	X

Notes:
-: flag not updated.
X: flag updated according to the standard definition.
*: flag update is non-standard, refer to the individual instruction description.
Note: Explicit writes to PSW flags takes precedence over flag updates.

Instruction Set Summary

Table 6.5 lists the entire XA instruction set by instruction type. This can be used as a quick reference to find specific instructions that may be looked up in the detailed alphabetical description section.

Instruction timing data given in this table and in the following detailed instruction description section are based on code execution from internal code memory and data accesses to internal RAM and registers only. Due to the highly programmable timing of accesses to external code and data memory on the XA and the interaction of pipelined functions, detailed timing for all conditions cannot be documented in a concise fashion. The instruction timing data given here also assumes that the CPU does not need to stall while the instruction is read into the pre-fetch queue.

In the case of branches, one on-chip code fetch (16 bits) is built into the timing numbers. The time given will be valid if the instruction that is branched to is not longer than two bytes. For longer instructions, the CPU will wait until the entire instruction is contained in the pre-fetch queue before resuming execution. This may take one or two additional fetches since the XA has instructions up to six bytes in length.

Following is a summary of events or conditions that may cause timing differences from the given data. These are generally stalls that occur when the CPU must wait for some information to become available.

— Instruction fetch. Execution stalls if the pre-fetch queue does not contains a complete instruction when it is needed. Except following branches, the state of the queue depends upon the history of instructions that have previously executed.

— Instruction sequence dependencies. This typically occurs when an instruction that reads data from a resource such as the SFR bus or the external bus follows an instruction that caused a write to the same resource. The CPU must stall while the write completes (which otherwise requires no CPU time) before the read can begin. Execution cannot resume until the read is complete.

— Internal data memory versus SFR accesses. SFR reads require an additional 2 clocks to complete. Because XA peripherals run from the CPU clock divided by 2, there may be one clock used to synchronize the CPU and the SFR bus.

— Program flow changes. When any change occurs in the program flow, the XA must flush the pre-fetch queue and begin loading it from the new execution address. The published timing values include one internal code fetch for all branches, jumps, calls, etc. If the instruction at the new address is longer than two bytes, additional fetch cycles must occur to obtain a complete instruction in the queue. In the case of a return from subroutine or interrupt, the first code fetch may only obtain one byte of the next instruction since returns may resume execution at odd code addresses.

— Internal versus external code execution. Programmable bus timing and other bus considerations result in a different timing for internal and external code accesses. Use of the 8-bit bus width for external code access has a substantial effect on overall performance. Possible use of the WAIT signal adds an additional variable to this effect. The external bus requirement for an ALE cycle at 16-byte address boundaries, during program flow changes, and after external bus data accesses also adds to the variability.

— Internal versus external data access. Programmable bus timing again causes different timing for internal and external data accesses. The 8-bit data bus setting contributes to the differences. Use of the WAIT signal may vary the timing still further.

— Collision of external code fetch and external data access. When an externally executing program accesses data on the external bus, the pre-fetch queue tends to starve more often that for internal execution.

Table 6.5

Mnemonic		Description	Bytes	Clocks
Arithmetic Operations				
ADD	Rd, Rs	Add registers direct	2	3
ADD	Rd, [Rs]	Add register-indirect to register	2	4
ADD	[Rd], Rs	Add register to register-indirect	2	4
ADD	Rd, [Rs+offset8]	Add register-indirect with 8-bit offset to register	3	6
ADD	[Rd+offset8], Rs	Add register to register-indirect with 8-bit offset	3	6
ADD	Rd, [Rs+offset16]	Add register-indirect with 16-bit offset to register	4	6
ADD	[Rd+offset16], Rs	Add register to register-indirect with 16-bit offset	4	6
ADD	Rd, [Rs+]	Add register-indirect with auto increment to register	2	5
ADD	[Rd+], Rs	Add register-indirect with auto increment to register	2	5
ADD	direct, Rs	Add register to memory	3	4
ADD	Rd, direct	Add memory to register	3	4
ADD	Rd, #data8	Add 8-bit immediate data to register	3	3
ADD	Rd, #data16	Add 16-bit immediate data to register	4	3
ADD	[Rd], #data8	Add 8-bit immediate data to register-indirect	3	4
ADD	[Rd], #data16	Add 16-bit immediate data to register-indirect	4	4
ADD	[Rd+], #data8	Add 8-bit immediate data to register-indirect with auto-increment	3	5
ADD	[Rd+], #data16	Add 16-bit immediate data to register-indirect with auto-increment	4	5
ADD	[Rd+offset8], #data8	Add 8-bit immediate data to register-indirect with 8-bit offset	4	6
ADD	[Rd+offset8], #data16	Add 16-bit immediate data to register-indirect with 8-bit offset	5	6
ADD	[Rd+offset16], #data8	Add 8-bit immediate data to register-indirect with 16-bit offset	5	6

Table 6.5

Mnemonic		Description	Bytes	Clocks
ADD	[Rd+offset16], #data16	Add 16-bit immediate data to register-indirect with 16-bit offset	6	6
ADD	direct, #data8	Add 8-bit immediate data to memory	4	4
ADD	direct, #data16	Add 16-bit immediate data to memory	5	4
ADDC	Rd, Rs	Add registers direct with carry	2	3
ADDC	Rd, [Rs]	Add register-indirect to register with carry	2	4
ADDC	[Rd], Rs	Add register to register-indirect with carry	2	4
ADDC	Rd, [Rs+offset8]	Add register-indirect with 8-bit offset to register with carry	3	6
ADDC	[Rd+offset8], Rs	Add register to register-indirect with 8-bit offset with carry	3	6
ADDC	Rd, [Rs+offset16]	Add register-indirect with 16-bit offset to register with carry	4	6
ADDC	[Rd+offset16], Rs	Add register to register-indirect with 16-bit offset with carry	4	6
ADDC	Rd, [Rs+]	Add register-indirect with auto increment to register with carry	2	5
ADDC	[Rd+], Rs	Add register-indirect with auto increment to register with carry	2	5
ADDC	direct, Rs	Add register to memory with carry	3	4
ADDC	Rd, direct	Add memory to register with carry	3	4
ADDC	Rd, #data8	Add 8-bit immediate data to register with carry	3	3
ADDC	Rd, #data16	Add 16-bit immediate data to register with carry	4	3
ADDC	[Rd], #data8	Add 16-bit immediate data to register-indirect with carry	3	4
ADDC	[Rd], #data16	Add 16-bit immediate data to register-indirect with carry	4	4
ADDC	[Rd+], #data8	Add 8-bit immediate data to register-indirect and auto-increment with carry	3	5
ADDC	[Rd+], #data16	Add 16-bit immediate data to register-indirect and auto-increment with carry	4	5
ADDC	[Rd+offset8], #data8	Add 8-bit immediate data to register-indirect with 8-bit offset and carry	4	6
ADDC	[Rd+offset8], #data16	Add 16-bit immediate data to register-indirect with 8-bit offset and carry	5	6

Table 6.5

Mnemonic		Description	Bytes	Clocks
ADDC	[Rd+offset16], #data8	Add 8-bit immediate data to register-indirect with 16-bit offset and carry	5	6
ADDC	[Rd+offset16], #data16	Add 16-bit immediate data to register-indirect with 16-bit offset and carry	6	6
ADDC	direct, #data8	Add 8-bit immediate data to memory with carry	4	4
ADDC	direct, #data16	Add 16-bit immediate data to memory with carry	5	4
ADDS	Rd, #data4	Add 4-bit signed immediate data to register	2	3
ADDS	[Rd], #data4	Add 4-bit signed immediate data to register-indirect	2	4
ADDS	[Rd+], #data4	Add 4-bit signed immediate data to register-indirect with auto-increment	2	5
ADDS	[Rd+offset8], #data4	Add register-indirect with 8-bit offset to 4-bit signed immediate data	3	6
ADDS	[Rd+offset16], #data4	Add register-indirect with 16-bit offset to 4-bit signed immediate data	4	6
ADDS	direct, #data4	Add 4-bit signed immediate data to memory	3	4
ASL	Rd, Rs	Logical left shift destination register by the value in the source register	2	See Note1
ASL	Rd, #data4	Logical left shift register by the 4-bit immediate value	2	See Note1
ASR	Rd, Rs	Arithmetic shift right destination register by the count in the source	2	See Note1
ASR	Rd, #data4	Arithmetic shift right register by the 4-bit immediate count	2	See Note1
CMP	Rd, Rs	Compare destination and source registers	2	3
CMP	[Rd], Rs	Compare register with register-indirect	2	4
CMP	Rd, [Rs]	Compare register-indirect with register	2	4
CMP	[Rd+offset8], Rs	Compare register with register-indirect with 8-bit offset	3	6
CMP	[Rd+offset16], Rs	Compare register with register-indirect with 16-bit offset	4	6
CMP	Rd, [Rs+offset8]	Compare register-indirect with 8-bit offset with register	3	6
CMP	Rd,[Rs+offset16]	Compare register-indirect with 16-bit offset with register	4	6

Table 6.5

Mnemonic		Description	Bytes	Clocks
CMP	Rd, [Rs+]	Compare auto-increment register-indirect with register	2	5
CMP	[Rd+], Rs	Compare register with auto-increment register-indirect	2	5
CMP	direct, Rs	Compare register with memory	3	4
CMP	Rd, direct	Compare memory with register	3	4
CMP	Rd, #data8	Compare 8-bit immediate data to register	3	3
CMP	Rd, #data16	Compare 16-bit immediate data to register	4	3
CMP	[Rd], #data8	Compare 8 -bit immediate data to register-indirect	3	4
CMP	[Rd], #data16	Compare 16-bit immediate data to register-indirect	4	4
CMP	[Rd+], #data8	Compare 8-bit immediate data to register-indirect with auto-increment	3	5
CMP	[Rd+], #data16	Compare 16-bit immediate data to register-indirect with auto-increment	4	5
CMP	[Rd+offset8], #data8	Compare 8-bit immediate data to register-indirect with 8-bit offset	4	6
CMP	[Rd+offset8], #data16	Compare 16-bit immediate data to register-indirect with 8-bit offset	5	6
CMP	[Rd+offset16], #data8	Compare 8-bit immediate data to register-indirect with 16-bit offset	5	6
CMP	[Rd+offset16], #data16	Compare 16-bit immediate data to register-indirect with 16-bit offset	6	6
CMP	direct, #data8	Compare 8-bit immediate data to memory	4	4
CMP	direct, #data16	Compare 16-bit immediate data to memory	5	4
DA	Rd	Decimal Adjust byte register	2	4
DIV.w	Rd, Rs	16x8 signed register divide	2	14
DIV.w	Rd, #data8	16x8 signed divide register with immediate word	3	14
DIV.d	Rd, Rs	32x16 signed double register divide	2	24
DIV.d	Rd, #data16	32x16 signed double register divide with immediate word	4	24
DIVU.b	Rd, Rs	8x8 unsigned register divide	2	12
DIVU.b	Rd, #data8	8X8 unsigned register divide with immediate byte	3	12

Table 6.5

Mnemonic		Description	Bytes	Clocks
DIVU.w	Rd, Rs	16X8 unsigned register divide	2	12
DIVU.w	Rd, #data8	16X8 unsigned register divide with immediate byte	3	12
DIVU.d	Rd, Rs	32X16 unsigned double register divide	2	22
DIVU.d	Rd, #data16	32X16 unsigned double register divide with immediate word	4	22
LEA	Rd, Rs+offset8	Load 16-bit effective address with 8-bit offset to register	3	3
LEA	Rd, Rs+offset16	Load 16-bit effective address with 16-bit offset to register	4	3
MUL.w	Rd, Rs	16X16 signed multiply of register contents	2	12
MUL.w	Rd, #data16	16X16 signed multiply 16-bit immediate data with register	4	12
MULU.b	Rd, Rs	8X8 unsigned multiply of register contents	2	12
MULU.b	Rd, #data8	8X8 unsigned multiply of 8-bit immediate data with register	3	12
MULU.w	Rd, Rs	16X16 unsigned register multiply	2	12
MULU.w	Rd, #data16	16X16 unsigned multiply 16-bit immediate data with register	4	12
NEG	Rd	Negate (twos complement) register	2	3
SEXT	Rd	Sign extend last operation to register	2	3
SUB	Rd, Rs	Subtract registers direct	2	3
SUB	Rd, [Rs]	Subtract register-indirect to register	2	4
SUB	[Rd], Rs	Subtract register to register-indirect	2	4
SUB	Rd, [Rs+offset8]	Subtract register-indirect with 8-bit offset to register	3	6
SUB	[Rd+offset8], Rs	Subtract register to register-indirect with 8-bit offset	3	6
SUB	Rd, [Rs+offset16]	Subtract register-indirect with 16-bit offset to register	4	6
SUB	[Rd+offset16], Rs	Subtract register to register-indirect with 16-bit offset	4	6
SUB	Rd, [Rs+]	Subtract register-indirect with auto increment to register	2	5
SUB	[Rd+], Rs	Subtract register-indirect with auto increment to register	2	5

Table 6.5

Mnemonic		Description	Bytes	Clocks
SUB	direct, Rs	Subtract register to memory	3	4
SUB	Rd, direct	Subtract memory to register	3	4
SUB	Rd, #data8	Subtract 8-bit immediate data to register	3	3
SUB	Rd, #data16	Subtract 16-bit immediate data to register	4	3
SUB	[Rd], #data8	Subtract 8-bit immediate data to register-indirect	3	4
SUB	[Rd], #data16	Subtract 16-bit immediate data to register-indirect	4	4
SUB	[Rd+], #data8	Subtract 8-bit immediate data to register-indirect with auto-increment	3	5
SUB	[Rd+], #data16	Subtract 16-bit immediate data to register-indirect with auto-increment	4	5
SUB	[Rd+offset8], #data8	Subtract 8-bit immediate data to register-indirect with 8-bit offset	4	6
SUB	[Rd+offset8], #data16	Subtract 16-bit immediate data to register-indirect with 8-bit offset	5	6
SUB	[Rd+offset16], #data8	Subtract 8-bit immediate data to register-indirect with 16-bit offset	5	6
SUB	[Rd+offset16], #data16	Subtract 16-bit immediate data to register-indirect with 16-bit offset	6	6
SUB	direct, #data8	Subtract 8-bit immediate data to memory	4	4
SUB	direct, #data16	Subtract 16-bit immediate data to memory	5	4
SUBB	Rd, Rs	Subtract with borrow registers direct	2	3
SUBB	Rd, [Rs]	Subtract with borrow register-indirect to register	2	4
SUBB	[Rd], Rs	Subtract with borrow register to register-indirect	2	4
SUBB	Rd, [Rs+offset8]	Subtract with borrow register-indirect with 8-bit offset to register	3	6
SUBB	[Rd+offset8], Rs	Subtract with borrow register to register-indirect with 8-bit offset	3	6
SUBB	Rd, [Rs+offset16]	Subtract with borrow register-indirect with 16-bit offset to register	4	6
SUBB	[Rd+offset16], Rs	Subtract with borrow register to register-indirect with 16-bit offset	4	6
SUBB	Rd, [Rs+]	Subtract with borrow register-indirect with auto increment to register	2	5

Table 6.5

Mnemonic		Description	Bytes	Clocks
SUBB	[Rd+], Rs	Subtract with borrow register-indirect with auto increment to register	2	5
SUBB	direct, Rs	Subtract with borrow register to memory	3	4
SUBB	Rd, direct	Subtract with borrow memory to register	3	4
SUBB	Rd, #data8	Subtract with borrow 8-bit immediate data to register	3	3
SUBB	Rd, #data16	Subtract with borrow 16-bit immediate data to register	4	3
SUBB	[Rd], #data8	Subtract with borrow 8-bit immediate data to register-indirect	3	4
SUBB	[Rd], #data16	Subtract with borrow 16-bit immediate data to register-indirect	4	4
SUBB	[Rd+], #data8	Subtract with borrow 8-bit immediate data to register-indirect with auto-increment	3	5
SUBB	[Rd+], #data16	Subtract with borrow 16-bit immediate data to register-indirect with auto-increment	4	5
SUBB	[Rd+offset8], #data8	Subtract with borrow 8-bit immediate data to register-indirect with 8-bit offset	4	6
SUBB	[Rd+offset8], #data16	Subtract with borrow 16-bit immediate data to register-indirect with 8-bit offset	5	6
SUBB	[Rd+offset16], #data8	Subtract with borrow 8-bit immediate data to register-indirect with 16-bit offset	5	6
SUBB	[Rd+offset16], #data16	Subtract with borrow 16-bit immediate data to register-indirect with 16-bit offset	6	6
SUBB	direct, #data8	Subtract with borrow 8-bit immediate data to memory	4	4
SUBB	direct, #data16	Subtract with borrow 16-bit immediate data to memory	5	4
Logical Operations				
AND	Rd, Rs	Logical AND registers direct	2	3
AND	Rd, [Rs]	Logical AND register-indirect to register	2	4
AND	[Rd], Rs	Logical AND register to register-indirect	2	4
AND	Rd, [Rs+offset8]	Logical AND register-indirect with 8-bit offset to register	3	6
AND	[Rd+offset8], Rs	Logical AND register to register-indirect with 8-bit offset	3	6

Table 6.5

Mnemonic		Description	Bytes	Clocks
AND	Rd, [Rs+offset16]	Logical AND register-indirect with 16-bit offset to register	4	6
AND	[Rd+offset16], Rs	Logical AND register to register-indirect with 16-bit offset	4	6
AND	Rd, [Rs+]	Logical AND register-indirect with auto increment to register	2	5
AND	[Rd+], Rs	Logical AND register-indirect with auto increment to register	2	5
AND	direct, Rs	Logical AND register to memory	3	4
AND	Rd, direct	Logical AND memory to register	3	4
AND	Rd, #data8	Logical AND 8-bit immediate data to register	3	3
AND	Rd, #data16	Logical AND 16-bit immediate data to register	4	3
AND	[Rd], #data8	Logical AND 8-bit immediate data to register-indirect	3	4
AND	[Rd], #data16	Logical AND16-bit immediate data to register-indirect	4	4
AND	[Rd+], #data8	Logical AND 8-bit immediate data to register-indirect and auto-increment	3	5
AND	[Rd+], #data16	Logical AND16-bit immediate data to register-indirect and auto-increment	4	5
AND	[Rd+offset8], #data8	Logical AND8-bit immediate data to register-indirect with 8-bit offset	4	6
AND	[Rd+offset8], #data16	Logical AND16-bit immediate data to register-indirect with 8-bit offset	5	6
AND	[Rd+offset16], #data8	Logical AND8-bit immediate data to register-indirect with 16-bit offset	5	6
AND	[Rd+offset16], #data16	Logical AND16-bit immediate data to register-indirect with 16-bit offset	6	6
AND	direct, #data8	Logical AND 8-bit immediate data to memory	4	4
AND	direct, #data16	Logical AND16-bit immediate data to memory	5	4
CPL	Rd	Complement (ones complement) register	2	3
LSR	Rd, Rs	Logical right shift destination register by the value in the source register	2	See Note 1
LSR	Rd, #data4	Logical right shift register by the 4-bit immediate value	2	See Note 1
NORM	Rd, Rs	Logical shift left destination register by the value in the source register until MSB set	2	See Note 1

Table 6.5

Mnemonic		Description	Bytes	Clocks
OR	Rd, Rs	Logical OR registers	2	3
OR	Rd, [Rs]	Logical OR register-indirect to register	2	4
OR	[Rd], Rs	Logical OR register to register-indirect	2	4
OR	Rd, [Rs+offset8]	Logical OR register-indirect with 8-bit offset to register	3	6
OR	[Rd+offset8], Rs	Logical OR register to register-indirect with 8-bit offset	3	6
OR	Rd, [Rs+offset16]	Logical OR register-indirect with 16-bit offset to register	4	6
OR	[Rd+offset16], Rs	Logical OR register to register-indirect with 16-bit offset	4	6
OR	Rd, [Rs+]	Logical OR register-indirect with auto increment to register	2	5
OR	[Rd+], Rs	Logical OR register-indirect with auto increment to register	2	5
OR	direct, Rs	Logical OR register to memory	3	4
OR	Rd, direct	Logical OR memory to register	3	4
OR	Rd, #data8	Logical OR 8-bit immediate data to register	3	3
OR	Rd, #data16	Logical OR 16-bit immediate data to register	4	3
OR	[Rd], #data8	Logical OR 8-bit immediate data to register-indirect	3	4
OR	[Rd], #data16	Logical OR 16-bit immediate data to register-indirect	4	4
OR	[Rd+], #data8	Logical OR 8-bit immediate data to register-indirect with auto-increment	3	5
OR	[Rd+], #data16	Logical OR 16-bit immediate data to register-indirect with auto-increment	4	5
OR	[Rd+offset8], #data8	Logical OR 8-bit immediate data to register-indirect with 8-bit offset	4	6
OR	[Rd+offset8], #data16	Logical OR 16-bit immediate data to register-indirect with 8-bit offset	5	6
OR	[Rd+offset16], #data8	Logical OR 8-bit immediate data to register-indirect with 16-bit offset	5	6
OR	[Rd+offset16], #data16	Logical OR 16-bit immediate data to register-indirect with 16-bit offset	6	6
OR	direct, #data8	Logical OR 8-bit immediate data to memory	4	4

Table 6.5

Mnemonic		Description	Bytes	Clocks
OR	direct, #data16	Logical OR16-bit immediate data to memory	5	4
RL	Rd, #data4	Rotate left register by the 4-bit immediate value	2	See Note 1
RLC	Rd, #data4	Rotate left register though carry by the 4-bit immediate value	2	See Note 1
RR	Rd, #data4	Rotate right register by the 4-bit immediate value	2	See Note 1
RRC	Rd, #data4	Rotate right register though carry by the 4-bit immediate value	2	See Note 1
XOR	Rd, Rs	Logical XOR registers	2	3
XOR	Rd, [Rs]	Logical XOR register-indirect to register	2	4
XOR	[Rd], Rs	Logical XOR register to register-indirect	2	4
XOR	Rd, [Rs+offset8]	Logical XOR register-indirect with 8-bit offset to register	3	6
XOR	[Rd+offset8], Rs	Logical XOR register to register-indirect with 8-bit offset	3	6
XOR	Rd, [Rs+offset16]	Logical XOR register-indirect with 16-bit offset to register	4	6
XOR	[Rd+offset16], Rs	Logical XOR register to register-indirect with 16-bit offset	4	6
XOR	Rd, [Rs+]	Logical XOR register-indirect with auto increment to register	2	5
XOR	[Rd+], Rs	Logical XOR register-indirect with auto increment to register	2	5
XOR	direct, Rs	Logical XOR register to memory	3	4
XOR	Rd, direct	Logical XOR memory to register	3	4
XOR	Rd, #data8	Logical XOR 8-bit immediate data to register	3	3
XOR	Rd, #data16	Logical XOR 16-bit immediate data to register	4	3
XOR	[Rd], #data8	Logical XOR 8-bit immediate data to register-indirect	3	4
XOR	[Rd], #data16	Logical XOR 16-bit immediate data to register-indirect	4	4
XOR	[Rd+], #data8	Logical XOR 8-bit immediate data to register-indirect with auto-increment	3	5
XOR	[Rd+], #data16	Logical XOR 16-bit immediate data to register-indirect with auto-increment	4	5

Table 6.5

Mnemonic		Description	Bytes	Clocks
XOR	[Rd+offset8], #data8	Logical XOR 8-bit immediate data to register-indirect with 8-bit offset	4	6
XOR	[Rd+offset8], #data16	Logical XOR 16-bit immediate data to register-indirect with 8-bit offset	5	6
XOR	[Rd+offset16], #data8	Logical XOR 8-bit immediate data to register-indirect with 16-bit offset	5	6
XOR	[Rd+offset16], #data16	Logical XOR 16-bit immediate data to register-indirect with 16-bit offset	6	6
XOR	direct, #data8	Logical XOR 8-bit immediate data to memory	4	4
XOR	direct, #data16	Logical XOR16-bit immediate data to memory	5	4
Data transfer				
MOV	Rd, Rs	Move register to register	2	3
MOV	Rd, [Rs]	Move register-indirect to register	2	3
MOV	[Rd], Rs	Move register to register-indirect	2	3
MOV	Rd, [Rs+offset8]	Move register-indirect with 8-bit offset to register	3	5
MOV	[Rd+offset8], Rs	Move register to register-indirect with 8-bit offset	3	5
MOV	Rd, [Rs+offset16]	Move register-indirect with 16-bit offset to register	4	5
MOV	[Rd+offset16], Rs	Move register to register-indirect with 16-bit offset	4	5
MOV	Rd, [Rs+]	Move register-indirect with auto increment to register	2	4
MOV	[Rd+], Rs	Move register-indirect with auto increment to register	2	4
MOV	direct, Rs	Move register to memory	3	4
MOV	Rd, direct	Move memory to register	3	4
MOV	[Rd+], [Rs+]	Move register-indirect to register-indirect, both pointers auto-incremented	2	6
MOV	direct, [Rs]	Move register-indirect to memory	3	4
MOV	[Rd], direct	Move memory to register-indirect	3	4
MOV	Rd, #data8	Move 8-bit immediate data to register	3	3
MOV	Rd, #data16	Move 16-bit immediate data to register	4	3

Table 6.5

Mnemonic		Description	Bytes	Clocks
MOV	[Rd], #data8	Move 16-bit immediate data to register-indirect	3	3
MOV	[Rd], #data16	Move 16-bit immediate data to register-indirect	4	3
MOV	[Rd+], #data8	Move 8-bit immediate data to register-indirect with auto-increment	3	4
MOV	[Rd+], #data16	Move 16-bit immediate data to register-indirect with auto-increment	4	4
MOV	[Rd+offset8], #data8	Move 8-bit immediate data to register-indirect with 8-bit offset	4	5
MOV	[Rd+offset8], #data16	Move 16-bit immediate data to register-indirect with 8-bit offset	5	5
MOV	[Rd+offset16], #data8	Move 8-bit immediate data to register-indirect with 16-bit offset	5	5
MOV	[Rd+offset16], #data16	Move 16-bit immediate data to register-indirect with 16-bit offset	6	5
MOV	direct, #data8	Move 8-bit immediate data to memory	4	3
MOV	direct, #data16	Move 16-bit immediate data to memory	5	3
MOV	direct, direct	Move memory to memory	4	4
MOV	Rd, USP	Move User Stack Pointer to register (system mode only)	2	3
MOV	USP, Rs	Move register to User Stack Pointer (system mode only)	2	3
MOVC	Rd, [Rs+]	Move data from WS:Rs address of code memory to register with auto-increment	2	4
MOVC	A, [A+DPTR]	Move data from code memory to the accumulator indirect with DPTR	2	6
MOVC	A, [A+PC]	Move data from code memory to the accumulator indirect with PC	2	6
MOVS	Rd, #data4	Move 4-bit sign-extended immediate data to register	2	3
MOVS	[Rd], #data4	Move 4-bit sign-extended immediate data to register-indirect	2	3
MOVS	[Rd+], #data4	Move 4-bit sign-extended immediate data to register-indirect with auto-increment	2	4
MOVS	[Rd+offset8], #data4	Move register-indirect with 8-bit offset to 4-bit sign-extended immediate data	3	5

Table 6.5

Mnemonic		Description	Bytes	Clocks
MOVS	[Rd+offset16], #data4	Move register-indirect with 16-bit offset to 4-bit sign-extended immediate data	4	5
MOVS	direct, #data4	Move 4-bit sign-extended immediate data to memory	3	3
MOVX	Rd, [Rs]	Move external data from memory to register	2	6
MOVX	[Rd], Rs	Move external data from register to memory	2	6
PUSH	direct	Push the memory content (byte/word) onto the current stack	3	5
PUSHU	direct	Push the memory content (byte/word) onto the user stack	3	5
PUSH	Rlist	Push multiple registers (byte/word) onto the current stack	2	See Note 2
PUSHU	Rlist	Push multiple registers (byte/word)from the user stack	2	See Note 2
POP	direct	Pop the memory content (byte/word) from the current stack	3	5
POPU	direct	Pop the memory content (byte/word) from the user stack	3	5
POP	Rlist	Pop multiple registers (byte/word) from the current stack	2	See Note 3
POPU	Rlist	Pop multiple registers (byte/word) from the user stack	2	See Note 3
XCH	Rd, Rs	Exchange contents of two registers	2	5
XCH	Rd, [Rs]	Exchange contents of a register-indirect address with a register	2	6
XCH	Rd, direct	Exchange contents of memory with a register	3	6
Program Branching				
BCC	rel8	Branch if the carry flag is clear	2	6t/3nt
BCS	rel8	Branch if the carry flag is set	2	6t/3nt
BEQ	rel8	Branch if the zero flag is set	2	6t/3nt
BNE	rel8	Branch if the zero flag is not set	2	6t/3nt
BG	rel8	Branch if greater than (unsigned)	2	6t/3nt
BGE	rel8	Branch if greater than or equal to (signed)	2	6t/3nt
BGT	rel8	Branch if greater than (signed)	2	6t/3nt

Table 6.5

Mnemonic		Description	Bytes	Clocks
BL	rel8	Branch if less than or equal to (unsigned)	2	6t/3nt
BLE	rel8	Branch if less than or equal to (signed)	2	6t/3nt
BLT	rel8	Branch if less than (signed)	2	6t/3nt
BMI	rel8	Branch if the negative flag is set	2	6t/3nt
BPL	rel8	Branch if the negative flag is clear	2	6t/3nt
BNV	rel8	Branch if overflow flag is clear	2	6t/3nt
BOV	rel8	Branch if overflow flag is set	2	6t/3nt
BR	rel8	Short unconditional branch	2	6
CALL	[Rs]	Subroutine call indirect with a register	2	8/5(PZ)
CALL	rel16	Relative call (+/- 64K)	3	7/4(PZ)
CJNE	Rd,direct,rel8	Compare direct byte to register and jump if not equal	4	10t/7nt
CJNE	Rd,#data8,rel8	Compare immediate byte to register and jump if not equal	4	9t/6nt
CJNE	Rd,#data16,rel8	Compare immediate word to register and jump if not equal	5	9t/6nt
CJNE	[Rd],#data8,rel8	Compare immediate word to register-indirect and jump if not equal	4	10t/7nt
CJNE	[Rd],#data16,rel8	Compare immediate word to register-indirect and jump if not equal	5	10t/7nt
DJNZ	Rd,rel8	Decrement register and jump if not zero	3	8t/5nt
DJNZ	direct,rel8	Decrement memory and jump if not zero	4	9t/5nt
FCALL	addr24	Far call (anywhere in the 24-bit address space)	4	12/8 (PZ)
FJMP	addr24	Far jump (anywhere in the 24-bit address space)	4	6
JB	bit,rel8	Jump if bit set	4	10t/6nt
JBC	bit,rel8	Jump if bit set and then clear the bit	4	11t/7nt
JMP	rel16	Long unconditional branch	3	6
JMP	[Rs]	Jump indirect to the address in the register (64K)	2	7
JMP	[A+DPTR]	Jump indirect relative to the DPTR	2	5
JMP	[[Rs+]]	Jump double-indirect to the address (pointer to a pointer)	2	8

Table 6.5

Mnemonic		Description	Bytes	Clocks
JNB	bit,rel8	Jump if bit not set	4	10t/6nt
JNZ	rel8	Jump if accumulator not equal zero	2	6t/3nt
JZ	rel8	Jump if accumulator equals zero	2	6t/3nt
NOP		No operation	1	3
RET		Return from subroutine	2	8/6(PZ)
RETI		Return from interrupt	2	10/ 8(PZ)
Bit Manipulation				
ANL	C, bit	Logical AND bit to carry	3	4
ANL	C, /bit	Logical AND complement of a bit to carry	3	4
CLR	bit	Clear bit	3	4
MOV	C, bit	Move bit to the carry flag	3	4
MOV	bit, C	Move carry to bit	3	4
ORL	C, bit	Logical OR a bit to carry	3	4
ORL	C, /bit	Logical OR complement of a bit to carry	3	4
SETB	bit	Sets the bit specified	3	4
Exception / Trap				
BKPT		Cause the breakpoint trap to be executed.	1	23/ 19(PZ)
RESET		Causes a hardware Reset, identical to an external Reset	2	18
TRAP	#data4	Causes 1 of 16 hardware traps to be executed	2	23/ 19(PZ)

Note 1: For 8 and 16 bit shifts, it is 4+1 per additional two bits. For 32-bit shifts, it is 6+1 per additional two bits.
Note 2: 3 clocks per register pushed.
Note 3: 4 clocks for the first register and two clocks for each additional register.

ADD **Integer Addition**

Syntax: ADD dest, source

Operation: dest <- src + dest

Description: Performs a twos complement binary addition of the source and destination operands, and the result is placed in the destination operand. The source data is not affected by the operation.

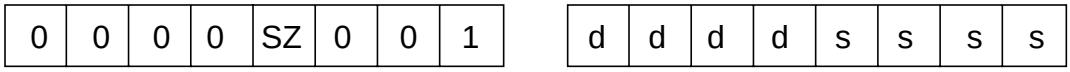
Note: If used with write to PSWL, takes precedence to flag updates

Sizes: Byte-Byte, Word-Word

Flags Updated: C, AC, V, N, Z

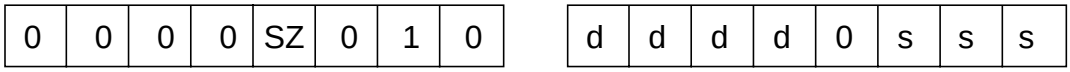
ADD Rd, Rs

Bytes**:** 2
Clocks**:** 3
Operation**:** (Rd) <-- (Rd) + (Rs)
Encoding:



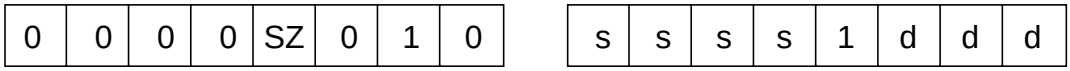
ADD Rd, [Rs]

Bytes: 2
Clocks: 4
Operation: (Rd) <-- (Rd) + ((WS:Rs))
Encoding:



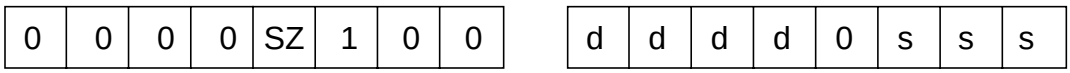
ADD [Rd], Rs

Bytes: 2
Clocks: 4
Operation**:** (WS:Rd) <-- (WS:Rd) + (Rs)
Encoding:



ADD Rd, [Rs+offset8]

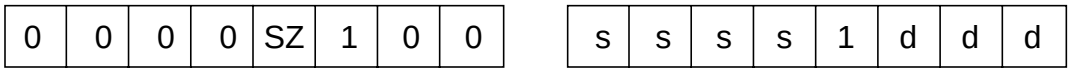
Bytes: 3
Clocks: 6
Operation**:** (Rd) <-- (Rd) + ((WS:Rs)+offset8)
Encoding:



byte 3: offset8

ADD [Rd+offset8], Rs

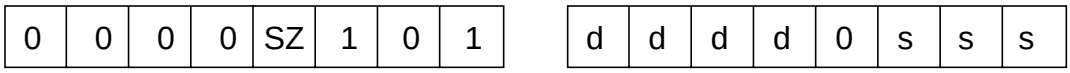
Bytes: 3
Clocks: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) + (Rs)
Encoding:



byte 3: offset8

ADD Rd, [Rs+offset16]

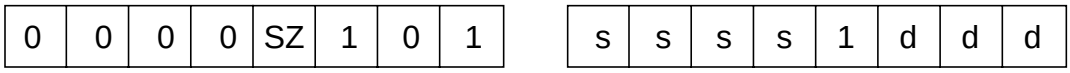
Bytes: 4
Clocks: 6
Operation: (Rd) <-- (Rd) + ((WS:Rs)+offset16)
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

ADD [Rd+offset16], Rs

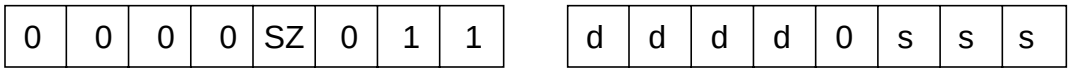
Bytes: 4
Clocks: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) + (Rs)
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

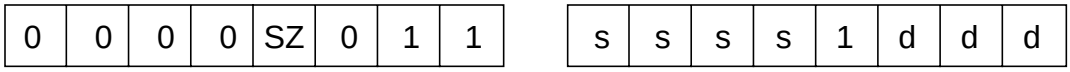
ADD Rd, [Rs+]

Bytes: 2
Clocks: 5
Operation: (Rd) <-- (Rd) + ((WS:Rs))
(Rs) <-- (Rs) + 1 (byte operation) or 2 (word operation)
Encoding:



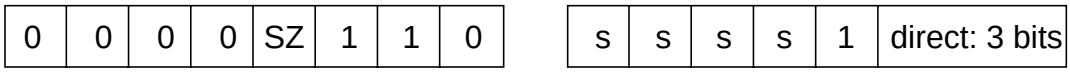
ADD [Rd+], Rs

Bytes: 2
Clocks: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) + (Rs)
(Rd) <-- (Rd) + 1 (byte operation) or 2 (word operation)
Encoding:



ADD direct, Rs

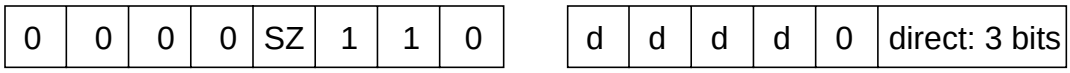
Bytes: 3
Clocks: 4
Operation: (direct) <-- (direct) + (Rs)
Encoding:



byte 3: lower 8 bits of direct

ADD Rd, direct

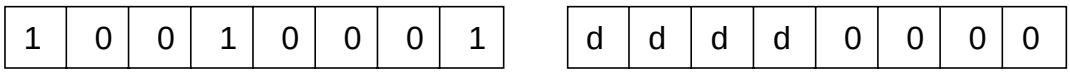
Bytes: 3
Clocks: 4
Operation: (Rd) <-- (Rd) + (direct)
Encoding:



byte 3: lower 8 bits of direct

ADD Rd, #data8

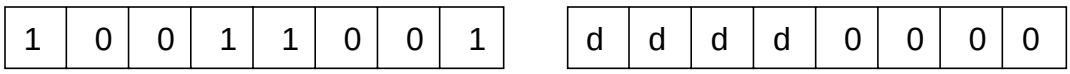
Bytes: 3
Clocks: 3
Operation: (Rd) <-- (Rd) + #data8
Encoding:



byte 3: #data8

ADD Rd, #data16

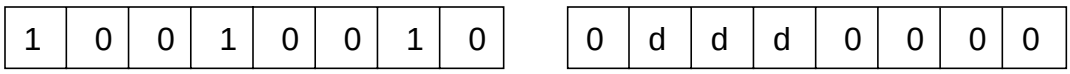
Bytes: 4
Clocks: 3
Operation: (Rd) <-- (Rd) + #data16
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

ADD [Rd], #data8

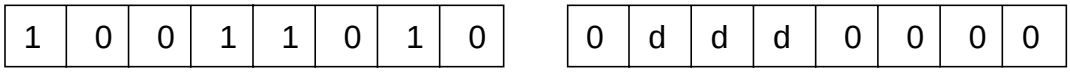
Bytes: 3
Clocks: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) + #data8
Encoding:



byte 3: #data8

ADD [Rd], #data16

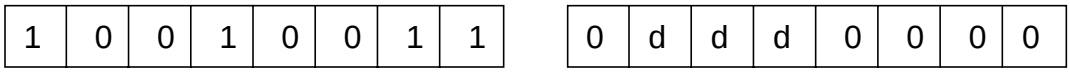
Bytes: 4
Clocks: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) + #data16
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

ADD [Rd+], #data8

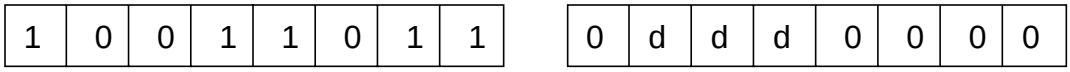
Bytes: 3
Clocks: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) + #data8
(Rd) <-- (Rd) + 1
Encoding:



byte 3: #data8

ADD [Rd+], #data16

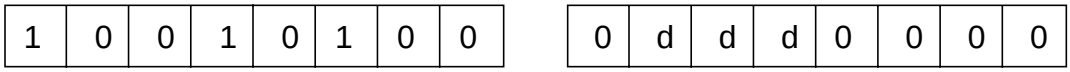
Bytes: 4
Clocks: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) + #data16
(Rd) <-- (Rd) + 2
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

ADD [Rd+offset8], #data8

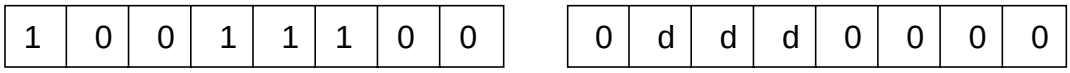
Bytes: 4
Clocks: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) + #data8
Encoding:



byte 3: offset8
byte 4: #data8

ADD [Rd+offset8], #data16

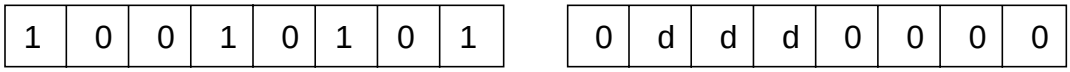
Bytes: 5
Clocks: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) + #data16
Encoding:



byte 3: offset8
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

ADD [Rd+offset16], #data8

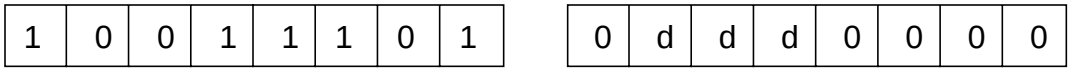
Bytes: 5
Clocks: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) + #data8
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: #data8

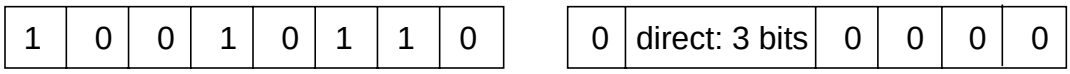
ADD [Rd+offset16], #data16

Bytes: 6
Clocks: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) + #data16
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: upper 8 bits of #data16
byte 6: lower 8 bits of #data16

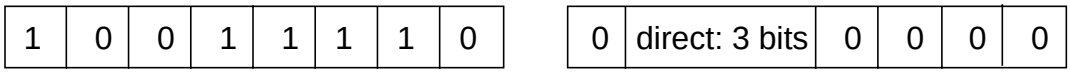
ADD direct, #data8
Bytes: 4
Clocks: 4
Operation: (direct) <-- (direct) + #data8
Encoding:



byte 3: lower 8 bits of direct
byte 4: #data8

ADD direct, #data16

Bytes: 5
Clocks: 4
Operation: (direct) <-- (direct) + #data16
Encoding:



byte 3: lower 8 bits of direct
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

Syntax: ADDC dest**,** source

Operation: dest <- dest + src + C

Description: Performs a two's complement binary addition of the source operand and the previously generated carry bit with the destination operand. The result is stored in the destination operand.The source data is not affected by the operation.

If the carry from previous operation is one (C=1), the result is greater than the sum of the operands; if it is zero (C=0), the result is the exact sum.

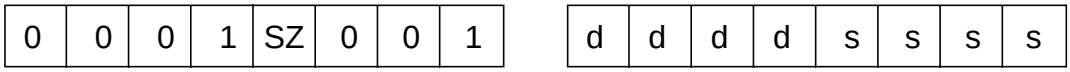
This form of addition is intended to support multiple-precision arithmetic. For this use, the carry bit is first reset, then ADDC is used to add the portions of the multiple-precision values from least-significant to most-significant.

Size: Byte-Byte, Word-Word

Flags Updated: C, AC, V, N, Z

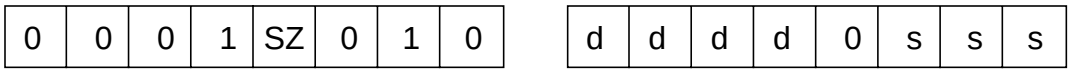
ADDC Rd, Rs

Bytes: 2
Clocks: 3
Operation: (Rd) <-- (Rd) + (Rs) + (C)
Encoding:



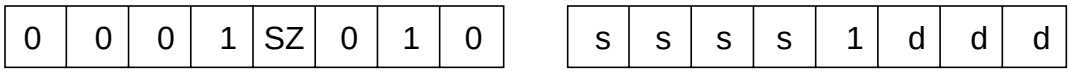
ADDC Rd, [Rs]

Bytes: 2
Clocks: 4
Operation: (Rd) <-- (Rd) + ((WS:Rs)) + (C)
Encoding:



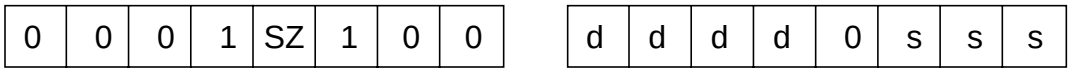
ADDC [Rd], Rs

Bytes: 2
Clocks: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) + (Rs) + (C)
Encoding:



ADDC Rd, [Rs+offset8]

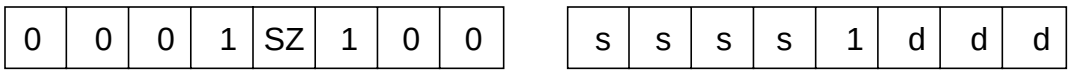
Bytes: 3
Clocks: 6
Operation: (Rd) <-- (Rd) + ((WS:Rs)+offset8) + (C)
Encoding:



byte 3: offset8

ADDC [Rd+offset8], Rs

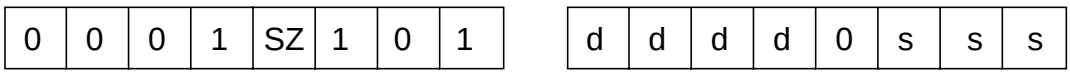
Bytes: 3
Clocks: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) + (Rs) + (C)
Encoding:



byte 3: offset8

ADDC Rd, [Rs+offset16]

Bytes: 4
Clocks: 6
Operation: (Rd) <-- (Rd) + ((WS:Rs)+offset16) + (C)
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

ADDC [Rd+offset16], Rs

Bytes: 4
Clocks: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) + (Rs) + (C)
Encoding:

0	0	0	1	SZ	1	0	1	s	s	s	s	1	d	d	d

byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

ADDC Rd, [Rs+]

Bytes: 2
Clocks: 5
Operation: (Rd) <-- (Rd) + ((WS:Rs)) + (C)
(Rs) <-- (Rs) + 1 (byte operation) or 2 (word operation)
Encoding:

0	0	0	1	SZ	0	1	1	d	d	d	d	0	s	s	s

ADDC [Rd+], Rs

Bytes: 2
Clocks: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) + (Rs) + (C)
(Rd) <-- (Rd) + 1 (byte operation) or 2 (word operation)
Encoding:

0	0	0	1	SZ	0	1	1	s	s	s	s	1	d	d	d

ADDC direct, Rs

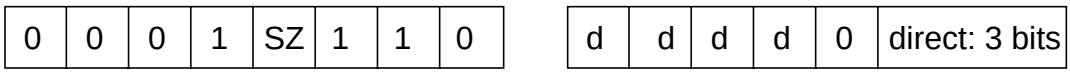
Bytes: 3
Clocks: 4
Operation: (direct) <-- (direct) + (Rs) + (C)
Encoding:

0	0	0	1	SZ	1	1	0	s	s	s	s	1	direct: 3 bits

byte 3: lower 8 bits of direct

ADDC Rd, direct

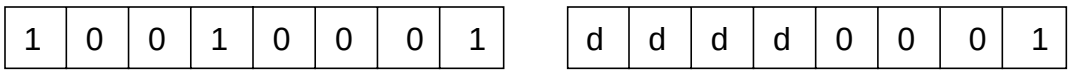
Bytes: 3
Clocks: 4
Operation: (Rd) <-- (Rd) + (direct) + (C)
Encoding:



byte 3: lower 8 bits of direct

ADDC Rd, #data8

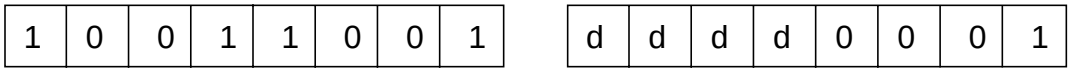
Bytes: 3
Clocks: 3
Operation: (Rd) <-- (Rd) + #data8 + (C)
Encoding:



byte 3: #data8

ADDC Rd, #data16

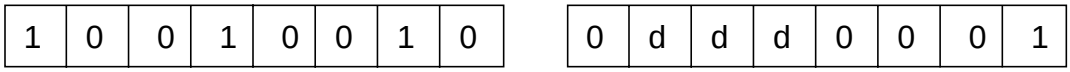
Bytes: 4
Clocks: 3
Operation: (Rd) <-- (Rd) + #data16 + (C)
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

ADDC [Rd], #data8

Bytes: 3
Clocks : 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) + #data8 + (C)
Encoding:

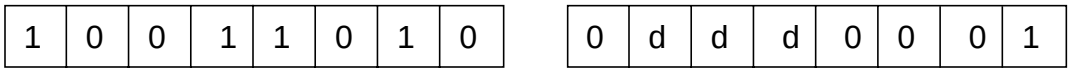


byte 3: #data8

ADDC [Rd], #data16

Bytes: 4
Clocks: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) + #data16 + (C)

Encoding:

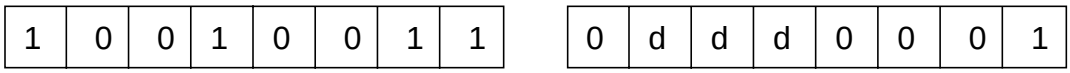


byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

ADDC [Rd+], #data8

Bytes: 3
Clocks: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) + #data8 + (C)
(Rd) <-- (Rd) + 1

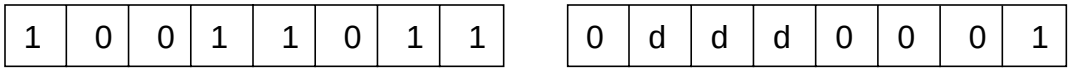
Encoding:



byte 3: #data8

ADDC [Rd+], #data16

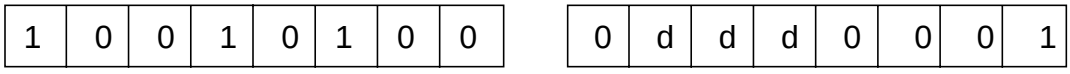
Bytes: 4
Clocks: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) + #data16 + (C)
(Rd) <-- (Rd) + 2
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

ADDC [Rd+offset8], #data8

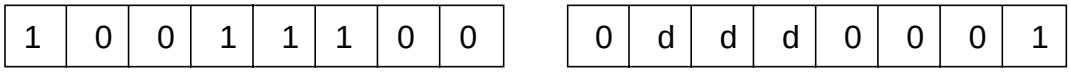
Bytes: 4
Clocks: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) + #data8 + (C)
Encoding:



byte 3: offset8
byte 4: #data8

ADDC [Rd+offset8], #data16

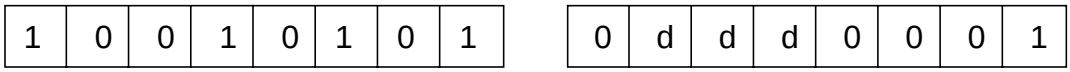
Bytes: 5
Clocks: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) + #data16 + (C)
Encoding:



byte 3: offset8
byte 4: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

ADDC [Rd+offset16], #data8

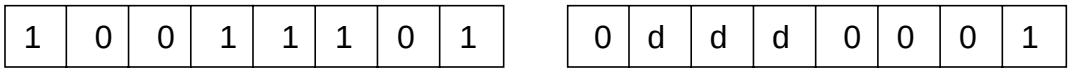
Bytes: 5
Clocks: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) + #data8 + (C)
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: #data8

ADDC [Rd+offset16], #data16

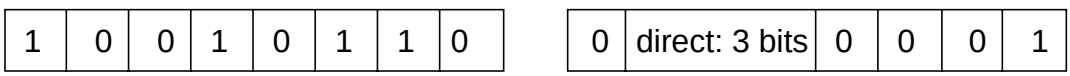
Bytes: 6
Clocks: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) + #data16 + (C)
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: upper 8 bits of #data16
byte 6: lower 8 bits of #data16

ADDC direct, #data8

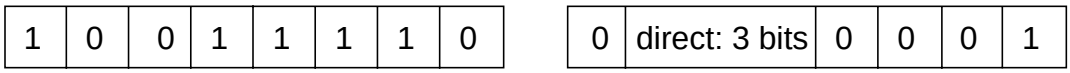
Bytes: 4
Clocks: 4
Operation: (direct) <-- (direct) + #data8 + (C)
Encoding:



byte 3: lower 8 bits of direct
byte 4: #data8

ADDC direct, #data16

Bytes: 5
Clocks: 4
Operation: (direct) <-- (direct) + #data16 + (C)
Encoding:



byte 3: lower 8 bits of direct
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

ADDS Add Short

Syntax: ADDS dest, #value

Operation: dest <- dest + #data4

Description: Four bits of signed immediate data are added to the destination. The immediate data is sign-extended to the proper size, then added to the variable specified by the destination operand, which may be either a byte or a word. The immediate data range is +7 to -8. This instruction is used primarily to increment or decrement pointers and counters.

Size: Byte-Byte, Word-Word

Flags Updated: N, Z

(Note: the C and AC flags must not be updated by ADDS since this instruction is used to replace the 80C51 INC and DEC instructions, which do not update the flags.)

ADDS Rd, #data4

Bytes: 2
Clocks: 3
Operation: (Rd) <-- (Rd) + #data4
Encoding:

1	0	1	0	SZ	0	0	1	d	d	d	d	#data4			

ADDS [Rd], #data4

Bytes: 2
Clocks: 4
Operation:((WS:Rd)) <-- ((WS:Rd)) + #data4
Encoding:

1	0	1	0	SZ	0	1	0	0	d	d	d	#data4			

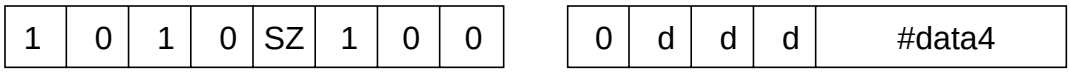
ADDS [Rd+], #data4

Bytes: 2
Clocks: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) + #data4
(Rd) <-- (Rd) + 1 (byte operation) or 2 (word operation)
Encoding:

1	0	1	0	SZ	0	1	1	0	d	d	d	#data4			

ADDS [Rd+offset8], #data4

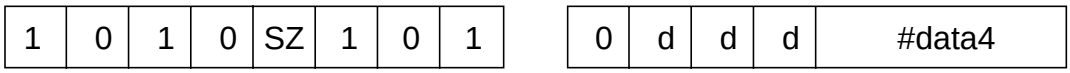
Bytes: 3
Clocks: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) + #data4
Encoding:



byte 3: offset8

ADDS [Rd+offset16], #data4

Bytes: 4
Clocks: 6
Operation:((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) + #data4
Encoding:

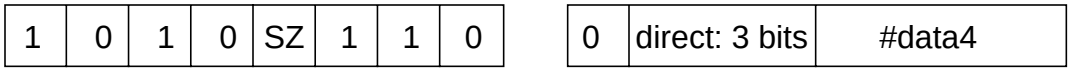


byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

ADDS direct, #data4

Bytes: 3
Clocks: 4
Operation:(direct) <-- (direct) + #data4

Encoding:



byte 3: lower 8 bits of direct

Syntax: AND dest, src

Operation: dest <- dest AND src

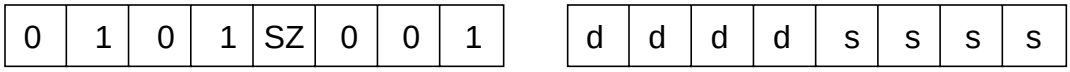
Description: Bitwise logical AND the contents of the source to the destination. The byte or word specified by the source operand is logically ANDed to the variable specified by the destination operand. The source data is not affected by the operation.

Size: Byte-Byte, Word-Word

Flags Updated: N, Z

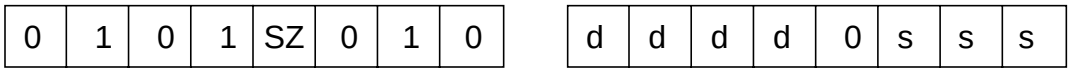
AND Rd, Rs

Bytes: 2
Clocks: 3
Operation: (Rd) <-- (Rd) • (Rs)
Encoding:



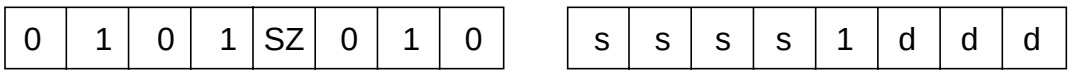
AND Rd, [Rs]

Bytes: 2
Clocks: 4
Operation: (Rd) <-- (Rd) • ((WS:Rs))
Encoding:



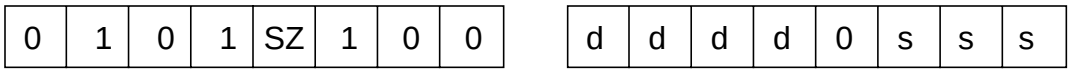
AND [Rd], Rs

Bytes: 2
Clocks: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) • (Rs)
Encoding:



AND Rd, [Rs+offset8]

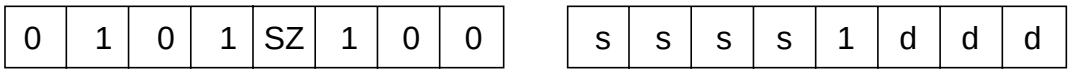
Bytes: 3
Clocks: 6
Operation: (Rd) <-- (Rd) • ((WS:Rs)+offset8)
Encoding:



byte 3: offset8

AND [Rd+offset8], Rs

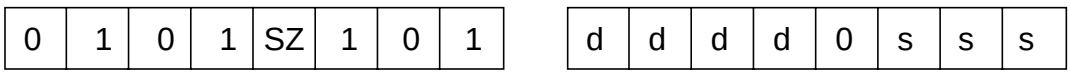
Bytes: 3
Clocks: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) • (Rs)
Encoding:



byte 3: offset8

AND Rd, [Rs+offset16]

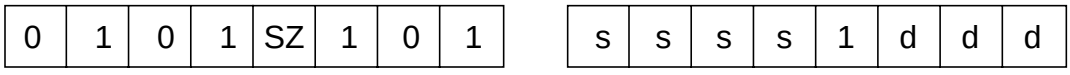
Bytes: 4
Clocks: 6
Operation: (Rd) <-- (Rd) • ((WS:Rs)+offset16)
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

AND [Rd+offset16], Rs

Bytes: 4
Clocks: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) • (Rs)
Encoding:

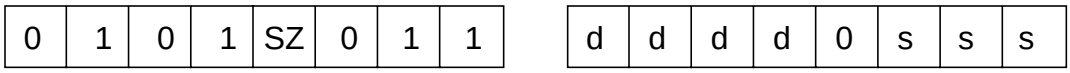


byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

AND Rd, [Rs+]

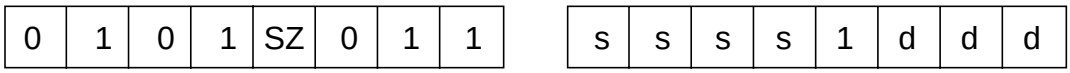
Bytes: 2
Clocks: 5
Operation: (Rd) <-- (Rd) • ((WS:Rs))
(Rs) <-- (Rs) + 1 (byte operation) or 2 (word operation)

Encoding:



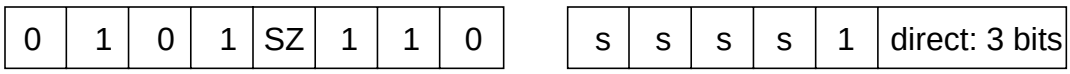
AND [Rd+], Rs

Bytes: 2
Clocks: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) • (Rs)
(Rd) <-- (Rd) + 1 (byte operation) or 2 (word operation)
Encoding:



AND direct, Rs

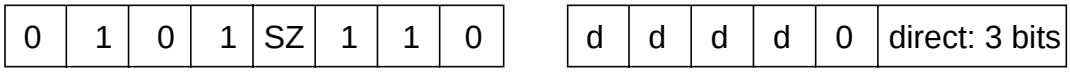
Bytes: 3
Clocks: 4
Operation: (direct) <-- (direct) • (Rs)
Encoding:



byte 3: lower 8 bits of direct

AND Rd, direct

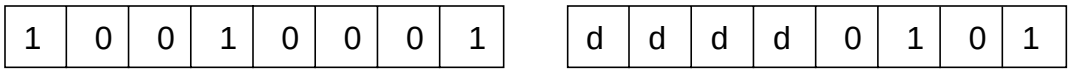
Bytes: 3
Clocks: 4
Operation: (Rd) <-- (Rd) • (direct)
Encoding:



byte 3: lower 8 bits of direct

AND Rd, #data8

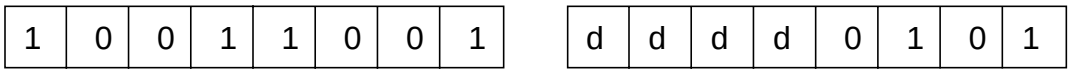
Bytes: 3
Clocks: 3
Operation: (Rd) <-- (Rd) • #data8
Encoding:



byte 3: #data8

AND Rd, #data16

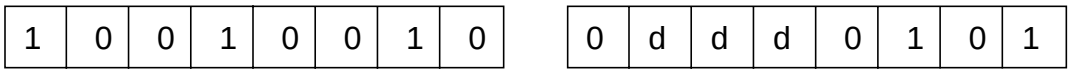
Bytes: 4
Clocks: 3
Operation: (Rd) <-- (Rd) • #data16
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

AND [Rd], #data8

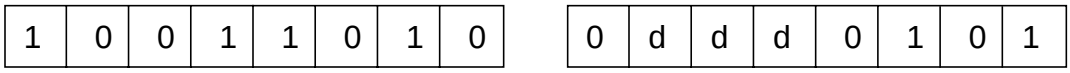
Bytes: 3
Clocks: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) • #data8
Encoding:



byte 3: #data8

AND [Rd], #data16

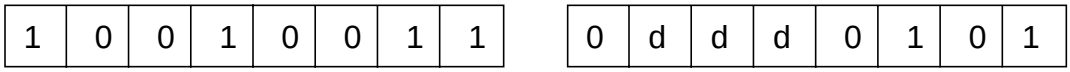
Bytes: 4
Clocks: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) • #data16
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

AND [Rd+], #data8

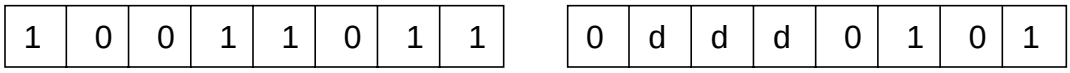
Bytes: 3
Clocks: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) • #data8
(Rd) <-- (Rd) + 1
Encoding:



byte 3: #data8

AND [Rd+], #data16

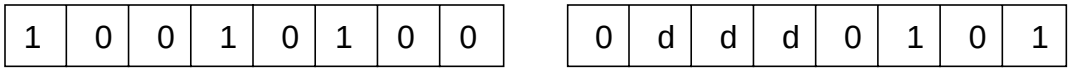
Bytes: 4
Clocks: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) • #data16
(Rd) <-- (Rd) + 2
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

AND [Rd+offset8], #data8

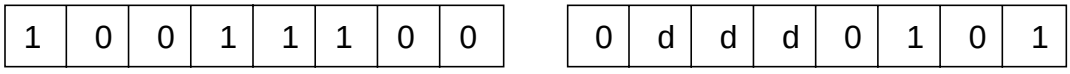
Bytes: 4
Clocks: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) • #data8
Encoding:



byte 3: offset8
byte 4: #data8

AND [Rd+offset8], #data16

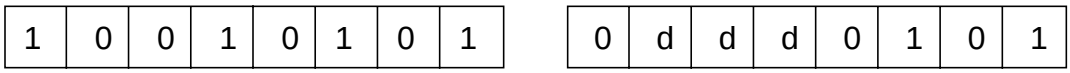
Bytes: 5
Clocks: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) • #data16
Encoding:



byte 3: offset8
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

AND [Rd+offset16], #data8

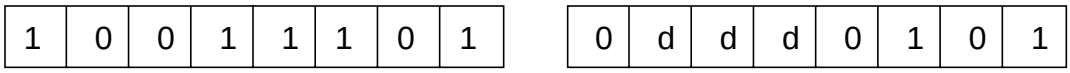
Bytes: 5
Clocks: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) • #data8
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: #data8

AND [Rd+offset16], #data16

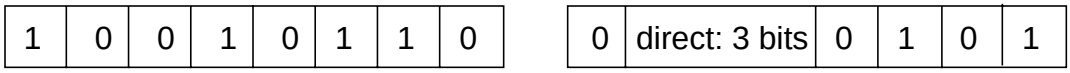
Bytes: 6
Clocks: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) • #data16
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: upper 8 bits of #data16
byte 6: lower 8 bits of #data16

AND direct, #data8

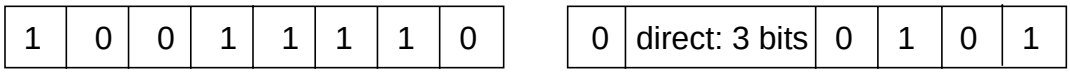
Bytes: 4
Clocks: 4
Operation: (direct) <-- (direct) • #data8
Encoding:



byte 3: lower 8 bits of direct
byte 4: #data8

AND direct, #data16

Bytes: 5
Clocks: 4
Operation: (direct) <-- (direct) • #data16
Encoding:



byte 3: lower 8 bits of direct
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

Syntax: ANL C, bit

Operation: C <- C (AND) Bit

Description: Read the specified bit and logically AND it to the Carry flag.

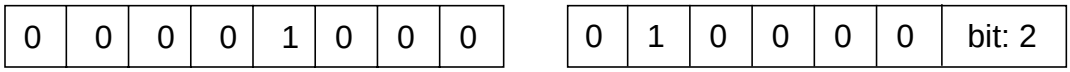
Size: Bit

Flags Updated: none

Note: Here the Carry bit is implicitly written by the instruction, and not to be confused with carry affected by the result of an ALU operation

Bytes: 3
Clocks: 4

Encoding:



byte 3: lower 8 bits of bit address

Syntax: ANL C, /bit

Operation: Carry <- C (AND) $\overline{\text{bit}}$

Description: Read the specified bit, complement it, and logically AND it to the Carry flag.

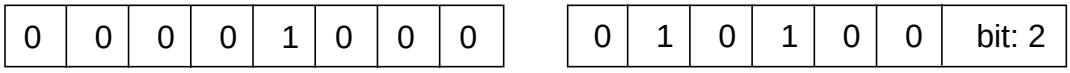
Size: Bit

Flags Updated: none

Note: Here the Carry bit is implicitly written by the instruction, and not to be confused with carry affected by the result of an ALU operation

Bytes: 3
Clocks: 4

Encoding:



byte 3: lower 8 bits of bit address

Syntax: ASL dest, count

Operation:

Do While (count not equal to 0)
(C) <- (dest.msb)
(dest.bit n+1) <- (dest.bit n)
count = count-1
if sign change during shift,
(V) <- 1
End While

Description:
If the count operand is greater than 0, the destination operand is logically shifted left by the number of bits specified by the count operand. The Low-order bits shifted in are zero-filled and the high-order bits are shifted out through the C (carry) bit. If the count operand is 0, no shift is performed.

The count operand could be:
- An immediate value (#data4 or #data5)
- A Register (Only 5 bits are used to implement up to 31 bit shifts)

The count is a positive value which may be from 1 to 31 and the destination operand is a signed integer (twos complement form).The destination operand (data size) may be 8, 16, or 32 bits. In the case of 32-bit shifts, the destination operand must be the least significant half of a double word register.The count operand is not affected by the operation.

Note:
- a double word register is double-word aligned in the register file (R1:R0, R3:R2, R5:R4, or R7:R6).
- If shift count (count in Rs) exceeds data size, the count value is truncated to 5 bits, else for immediate shift count, shifting is continued until count is 0.

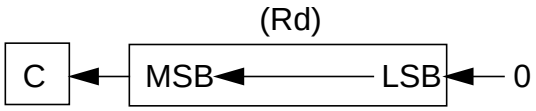
Size: Byte, word, and double word

Flags Updated: C, V, N, Z

Note: The V flag is set if the sign changes at any time during the shift operation and remains set until the end of the shift operation i.e., the V flag does not get cleared even if the sign reverts to its original state because of continued shifts within the same instruction. ASL clears the V flag if the condition to set it does not occur.

ASL Rd, Rs

Operation:

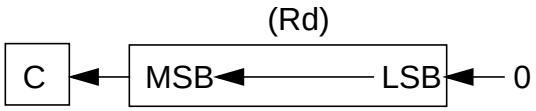


Bytes**:** 2
Clocks**:** For 8/16 bit shifts -> 4 + 1 for each 2 bits of shift
For 32 bit shifts -> 6 + 1 for each 2 bits of shift
Encoding**:**

1	1	0	0	SZ1	SZ0	0	1	d	d	d	d	s	s	s	s

ASL Rd, #data4
Rd,#data5

Bytes: 2
Clocks: For 8/16 bit shifts -> 4 + 1 for each 2 bits of shift
For 32 bit shifts -> 6 + 1 for each 2 bits of shift
Operation:



Encoding: (for byte and word data sizes)

1	1	0	1	SZ1	SZ0	0	1	d	d	d	d	#data4

(for double word data size)

1	1	0	1	1	1	0	1	d	d	d	#data5

Note: SZ1/SZ0 = 00 : byte operation; SZ1/SZ0 = 10 : word operation; SZ1/SZ0 = 11 : double word operation.

Syntax: ASR dest, count

Operation:

Do While (count not equal to 0)
(C) <- (dest.0)
(dest.bit n) <- (dest.bit n+1)
dest.msb <- Sign bit
count = count-1
End While

Description:
If the count operand is greater than 0, the destination operand is logically shifted right by the number of bits specified by the count operand. The low-order bits are shifted out through the C (carry) bit. If the count operand is 0, no shift is performed. To preserve the sign of the original operand, the MSBs of the result are sign-extended with the sign bit.

The count operand could be:
- An immediate value (#data4/5)
- A Register (Only 5 bits are used to implement up to 31 bit shifts)

The count operand could be an immediate value or a register. The count is a positive value which may be from 0 to 31 and the destination operand is a signed integer. The count operand is not affected by the operation. The data size may be 8, 16, or 32 bits. In the case of 32-bit shifts, the destination operand must be the least significant half of a double word register.

Note:
- a double word register is double-word aligned in the register file (R1:R0, R3:R2, R5:R4, or R7:R6).
- If shift count (count in Rs) exceeds data size, the count value is truncated to 5 bits, else for immediate shift count, shifting is continued until count is 0.
- a double word register is double-word aligned in the register file (R1:R0, R3:R2, R5:R4, or R7:R6).

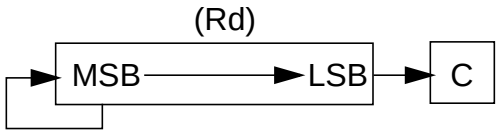
Size: Byte, Word, Double Word

Flags Updated: C, N, Z

ASR Rd, Rs

Bytes: 2
Clocks: For 8/16 bit shifts -> 4 + 1 for each 2 bits of shift
For 32 bit shifts -> 6 + 1 for each 2 bits of shift

Operation:

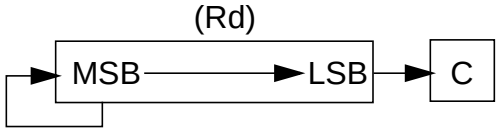


Encoding:

1	1	0	0	SZ1	SZ0	1	0	d	d	d	d	s	s	s	s

ASR Rd, #data4
Rd,#data5

Operation:



Bytes: 2
Clocks: For 8/16 bit shifts -> 4 + 1 for each 2 bits of shift
For 32 bit shifts -> 6 + 1 for each 2 bits of shift

Encoding: (for byte and word data sizes)

1	1	0	1	SZ1	SZ0	1	0	d	d	d	d	#data4			

(for double word data size)

1	1	0	1	SZ1	SZ0	1	0	d	d	d	#data5				

Note: SZ1/SZ0 = 00: byte operation; SZ1/SZ0 = 10: word operation; SZ1/SZ0 = 11: double word operation.

BCC **Branch if carry clear**

Syntax: BCC rel8

Operation:

(PC) <-- (PC) + 2
if (C) = 0 then
(PC) <-- (PC + rel8*2)
(PC.0) <-- 0

Description: The branch is taken if the last arithmetic instruction (or other instruction that updates the C flag) did not generate a carry (the carry flag contains a 0). If Carry is clear, the program execution branches at the location of the PC, plus the specified displacement, rel8. The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

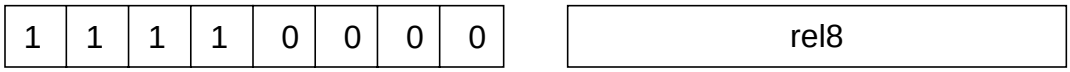
Note: Refer to section 6.3 for details of branch range

Size: Bit

Flags Updated: none

Bytes: 2
Clocks: 6 (t) / 3 (nt)

Encoding:



BCS **Branch if carry set**

Syntax: BCS rel8

Operation:

(PC) <-- (PC) + 2
if (C) = 1 then
(PC) <-- (PC + rel8*2)
(PC.0) <-- 0

Description: The branch is taken if the last arithmetic instruction (or other instruction that updates the C flag) generated a carry (the carry flag contains a 1). The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

Note: Refer to section 6.3 for details of branch range

Size: Bit

Flags Updated: none

Bytes: 2
Clocks: 6t/3nt

Encoding:

1	1	1	1	0	0	0	1	rel8

BEQ Branch if zero

Syntax: BEQ rel8

Operation:

(PC) <-- (PC) + 2
if (Z) = 1 then
(PC) <-- (PC + rel8*2)
(PC.0) <-- 0

Description: The branch is taken if the last arithmetic/logic instruction (or other instruction that updates the Z flag) had a result of zero (the Z flag contains a 1). The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

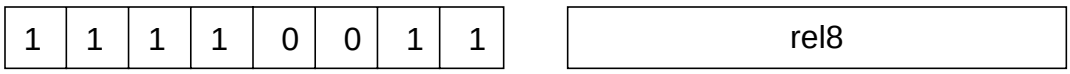
Note: Refer to section 6.3 for details of branch range

Size: Bit

Flags Updated: none

Bytes: 2
Clocks: 6t/3nt

Encoding:



BG **Branch if greater than (unsigned)**

Syntax: BG rel8

Operation: (PC) <-- (PC) + 2
if (Z) OR (C) = 0 then
(PC) <-- (PC + rel8*2)
(PC.0) <-- 0

Description: The branch is taken if the last compare instruction had a destination value that was greater than the source value, in an unsigned operation. The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

Note: Refer to section 6.3 for details of branch range

Size: Bit

Flags Updated: none

Bytes: 2
Clocks: 6t/3nt

Encoding:

1	1	1	1	1	0	0	0	rel8

BGE **Branch if greater than or equal to (signed)**

Syntax: BGE rel8

Operation: (PC) <-- (PC) + 2
if (N) XOR (V) = 0 then
(PC) <-- (PC + rel8*2)
(PC.0) <-- 0

Description: The branch is taken if the last compare instruction had a destination value that was greater than or equal to the source value, in a signed operation. The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

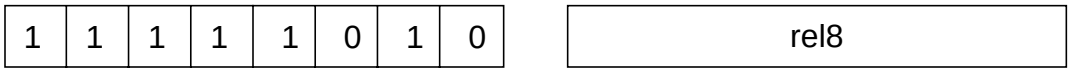
Note: Refer to section 6.3 for details of branch range

Size: Bit

Flags Updated: none

Bytes: 2
Clocks: 6t/3nt

Encoding:



BGT **Branch if greater than (signed)**

Syntax: BGT rel8

Operation: (PC) <-- (PC) + 2
if ((Z) OR (N)) XOR (V) = 0 then
(PC) <-- (PC + rel8*2)
(PC.0) <-- 0

Description: The branch is taken if the last compare instruction had a destination value that was greater than the source value, in a signed operation. The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

Note: Refer to section 6.3 for details of branch range

Size: Bit

Flags Updated: none

Bytes: 2
Clocks: 6t/3nt

Encoding:

1	1	1	1	1	1	0	0	rel8

Syntax: BKPT

Operation:
(PC) <-- (PC) + 1
(SSP) <-- (SSP) - 6
((SSP)) <-- (PC)
((SSP)) <-- (PSW)
(PSW) <-- code memory (bkpt vector)
(PC.15-0) <-- code memory (bkpt vector)
(PC.23-16) <-- 0; (PC.0) <-- 0

Description: Causes a breakpoint trap. The breakpoint trap acts like an immediate interrupt, using a vector to call a specific piece of code that will be executed in system mode. This instruction is intended for use in emulator systems to provide a simple method of implementing hardware breakpoints.

For a breakpoint to work properly under all conditions, it must have an instruction length no greater than the smallest other instruction on the processor, in this case the one byte NOP. This requirement exists because a breakpoint may be inserted in place of a NOP that is followed by another instruction that is branched to or otherwise executed without going through the breakpoint. If the breakpoint instruction were longer than the NOP, it would corrupt the next instruction in sequence if that instruction were executed.

The opcode for the breakpoint instruction is specifically assigned to be all ones (FFh). This is so that un-programmed EPROM code memory will contain breakpoints. Similarly, the NOP instruction is assigned to opcode 00 so that both "blank" code states map to innocuous instructions.

Size: None

Flags Updated: none[5]

Bytes: 1
Clocks: 23/19 (PZ)

Encoding:

1	1	1	1	1	1	1	1

5. All flags are affected during the PSW load from the vector table. It is possible that these flags are restored by the debugger, but does not have to be the case.

BL **Branch if less than or equal to (unsigned)**

Syntax: BL rel8

Operation: (PC) <-- (PC) + 2
if (Z) OR (C) = 1 then
(PC) <-- (PC + rel8*2)
(PC.0) <-- 0

Description: The branch is taken if the last compare instruction had a destination value that was less than or equal to the source value, in an unsigned operation. The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

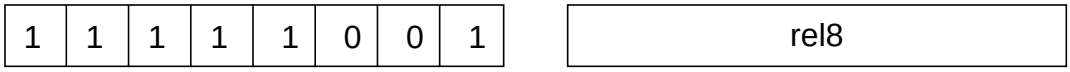
Note: Refer to section 6.3 for details of branch range

Size: Bit

Flags Updated: none

Bytes: 2
Clocks: 6t/3nt

Encoding:



Syntax: BLE rel8

Operation: (PC) <-- (PC) + 2
if ((Z) OR (N)) XOR (V) = 1 then
(PC) <-- (PC + rel8*2)
(PC.0) <-- 0

Description: The branch is taken if the last compare instruction had a destination value that was less than or equal to the source value, in a signed operation. The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

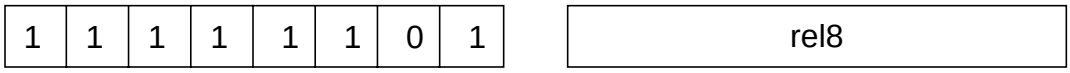
Note: Refer to section 6.3 for details of branch range

Size: Bit

Flags Updated: none

Bytes: 2
Clocks: 6t/3nt

Encoding:



BLT **Branch if less than (signed)**

Syntax: BLT rel8

Operation: (PC) <-- (PC) + 2
if (N) XOR (V) = 1 then
(PC) <-- (PC + rel8*2)
(PC.0) <-- 0

Description: The branch is taken if the last compare instruction had a destination value that was less than the source value, in a signed operation. The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

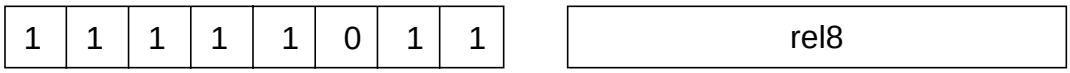
Note: Refer to section 6.3 for details of branch range

Size: Bit

Flags Updated: none

Bytes: 2
Clocks: 6t/3nt

Encoding:



BMI **Branch if negative**

Syntax: BMI rel8

Operation: (PC) <-- (PC) + 2
if (N) = 1 then
(PC) <-- (PC + rel8*2)
(PC.0) <-- 0

Description: The branch is taken if the last arithmetic/logic instruction (or other instruction that updates the N flag) had a result that is less than 0 (the N flag contains a 1). The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

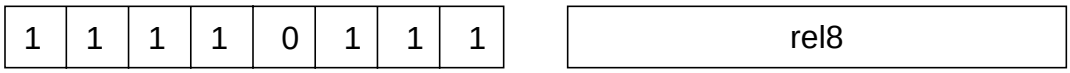
Note: Refer to section 6.3 for details of branch range

Size: Bit

Flags Updated: none

Bytes: 2
Clocks: 6t/3nt

Encoding:



BNE **Branch if not equal**

Syntax: BNE rel8

Operation: (PC) <-- (PC) + 2
if (Z) = 0 then
(PC) <-- (PC + rel8*2)
(PC.0) <-- 0

Description: The branch is taken if the last arithmetic/logic instruction (or other instruction that updates the Z flag) had a non-zero result (the Z flag contains a 0). The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

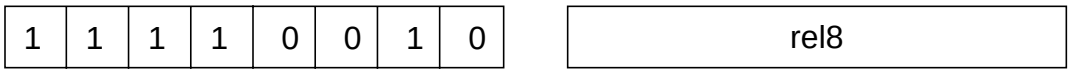
Note: Refer to section 6.3 for details of branch range

Size: Bit

Flags Updated: none

Bytes: 2
Clocks: 6t/3nt

Encoding:



Syntax: BNV rel8

Operation: (PC) <-- (PC) + 2
if (V) = 0 then
(PC) <-- (PC + rel8*2)
(PC.0) <-- 0

Description: The branch is taken if the last arithmetic/logic instruction (or other instruction that updates the V flag) did not generate an overflow (The V flag contains a 0). The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

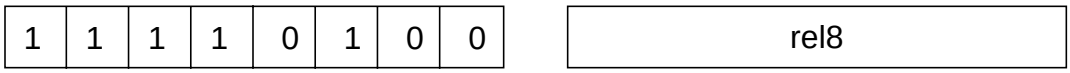
Note: Refer to section 6.3 for details of branch range

Size: Bit

Flags Updated: none

Bytes: 2
Clocks: 6t/3nt

Encoding:



Syntax: BOV rel8

Operation: (PC) <-- (PC) + 2
if (V) = 1 then
(PC) <-- (PC + rel8*2)
(PC.0) <-- 0

Description: The branch is taken if the last arithmetic/logic instruction (or other instruction that updates the V flag) generated an overflow (the V flag contains a 1). The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

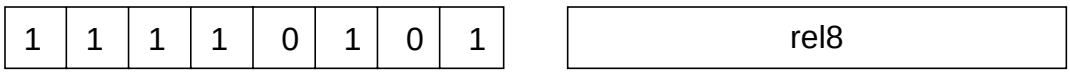
Note: Refer to section 6.3 for details of branch range

Size: Bit

Flags Updated: none

Bytes: 2
Clocks: 6t/3nt

Encoding:



Syntax: BPL rel8

Operation: (PC) <-- (PC) + 2
if (N) = 0 then
(PC) <-- (PC + rel8*2)
(PC.0) <-- 0

Description: The branch is taken if the last arithmetic/logic instruction (or other instruction that updates the N flag) had a result that is greater than 0 (the N flag contains a 0). The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

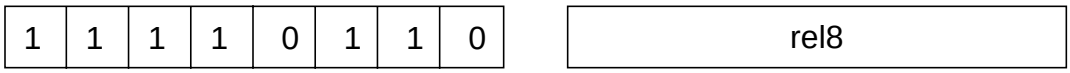
Note: Refer to section 6.3 for details of branch range

Size: Bit

Flags Updated: none

Bytes: 2
Clocks: 6t/3nt

Encoding:



Syntax: BR rel8

Operation: (PC) <-- (PC) + 2
(PC) <-- (PC + rel8*2)
(PC.0) <-- 0

Description: Branches unconditionally in the range of +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

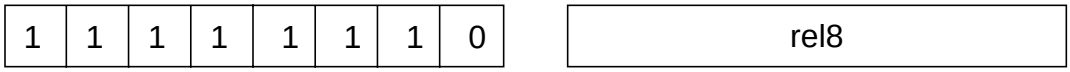
Note: Refer to section 6.3 for details of branch range

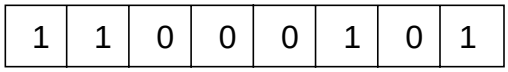
Size: None

Flags Updated: none

Bytes: 2
Clocks: 6

Encoding:





Syntax: CALL rel16

Operation: (PC) <-- (PC) + 3
(SP) <-- (SP) - 4
((SP)) <-- (PC.23-0)
(PC) <-- (PC + rel16*2)
(PC.0) <-- 0

Description: Branches unconditionally in the range of +65,534 bytes to -65,536 bytes, with the limitation that the target address is word aligned in code memory. The 24-bit return address is saved on the stack.

Note: if the XA is in page 0 mode, only a 16-bit address will be pushed to the stack.

Note: Refer to section 6.3 for details of branch range

Size: None

Flags Updated: none

Bytes: 3
Clocks: 7/4(PZ)

Encoding:

byte 2: upper 8 bits of rel16
byte 3: lower 8 bits of rel16

Syntax: CALL [Rs]

Operation: (PC) <-- (PC) + 2
(SP) <-- (SP) - 4
((SP)) <-- (PC.23-0)
(PC.15-1) <-- (Rs.15-1)
(PC.0) <-- 0

Description: Causes an unconditional branch to the address contained in the operand register, anywhere within the 64K page following the CALL instruction.The return address (the address following the CALL instruction) of the calling routine is saved on the stack. The target address must be word aligned, as CALL or branch will force PC.bit0 to 0.

Note:
(1) Since the PC always points to the instruction following the CALL instruction and if that happens to be on a different page, then the called routine should be located in that page (64K)

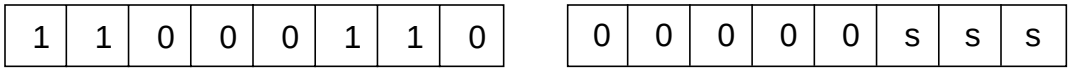
(2) if the XA is in page 0 mode, only a 16-bit address will be pushed to the stack.

Size: None

Flags Updated: none

Bytes: 2
Clocks: 8/5(PZ)

Encoding:



CJNE **Compare and jump if not equal**

Syntax: CJNE dest, src, rel8

Operation: (PC) <-- (PC) + # of instruction bytes
(dest) - (direct) (result not stored)
if (Z) = 0 then
(PC) <-- (PC + rel8*2); (PC.0) <-- 0

Description: The byte or word specified by the source operand is compared to the variable specified by the destination operand and the status flags are updated. Jump to the specified address if the values are not equal. The source and destination data are not affected by the operation. The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

Note: Refer to section 6.3 for details of jump range

Size: Byte-Byte, Word-Word

Flags Updated: C, N, Z

(Note: this particular type of compare must not update the V or AC flags to duplicate the 80C51 function.)

CJNE Rd, direct, rel8

Bytes: 4
Clocks: 10t/7nt
Encoding:

1	1	1	0	SZ	0	1	0

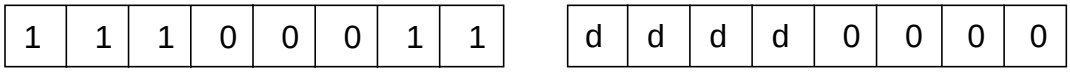
d	d	d	d	0	direct: 3 bits

byte 3: lower 8 bits of direct
byte 4: rel8

CJNE Rd, #data8, rel8

Bytes: 4
Clocks: 9t/6nt

Encoding:

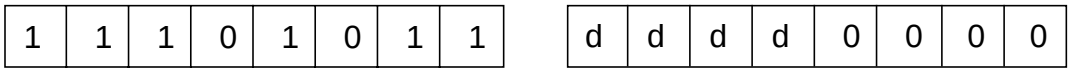


byte 3: rel8
byte 4: data#8

CJNE Rd, #data16, rel8

Bytes: 5
Clocks: 9t/6nt

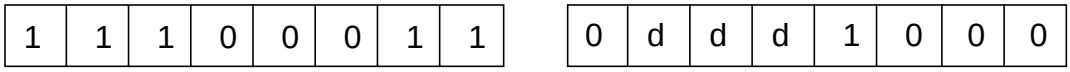
Encoding:



byte 3: rel8
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

CJNE [Rd], #data8, rel8

Bytes: 4
Clocks: 10t/7nt
Encoding:

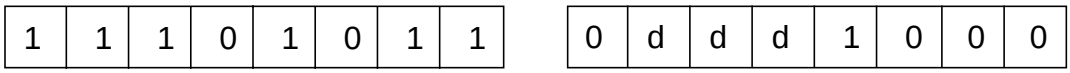


byte 3: rel8
byte 4: #data8

CJNE [Rd], #data16, rel8

Bytes: 5
Clocks: 10t/7nt

Encoding:



byte 3: rel8
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

Syntax: CLR bit

Operation: (bit) <-- 0

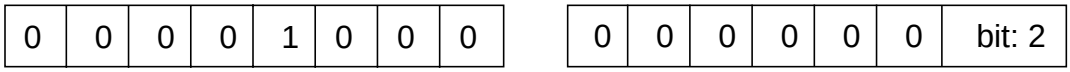
Description: Writes a 0 (clears) to the specified bit.

Size: Bit

Flags Updated: none

Bytes: 3
Clocks: 4

Encoding:



byte 3: lower 8 bits of bit address

Syntax: CMP dest, src

Operation: dest - src

Description: The byte or word specified by the source operand is compared to the specified destination operand by performing a twos complement binary subtraction of src from dest. The flags are set according to the rules of subtraction. The source and destination data are not affected by the operation.

Size: byte-byte, word-word

Flags Updated: C, AC, V, N, Z

CMP Rd, Rs

Operation**:** (Rd) - (Rs)

Bytes**:** 2
Clocks**:** 3

Encoding:

0	1	0	0	SZ	0	0	1	d	d	d	d	s	s	s	s

CMP Rd, [Rs]

Operation: (Rd) - ((WS:Rs))

Bytes: 2
Clocks: 4

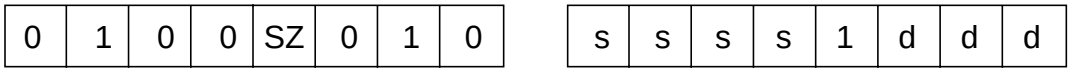
Encoding:

0	1	0	0	SZ	0	1	0	d	d	d	d	0	s	s	s

CMP [Rd], Rs

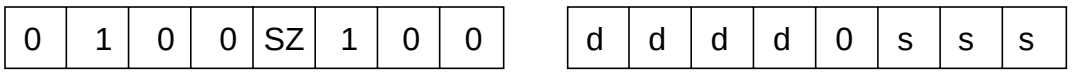
Operation: ((WS:Rd)) - (Rs)

Bytes: 2
Clocks: 4
Encoding**:**



CMP Rd, [Rs+offset8]

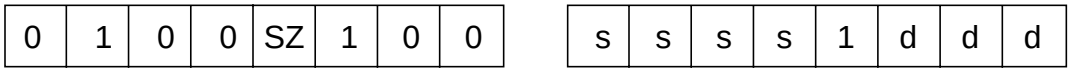
Bytes: 3
Clocks: 6
Operation: (Rd) - ((WS:Rs)+offset8)
Encoding:



byte 3: offset8

CMP [Rd+offset8], Rs

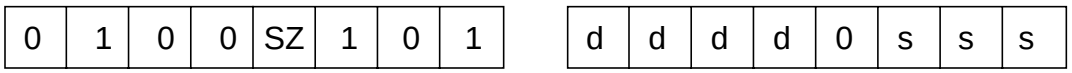
Bytes: 3
Clocks: 6
Operation: ((WS:Rd)+offset8) - (Rs)
Encoding:



byte 3: offset8

CMP Rd, [Rs+offset16]

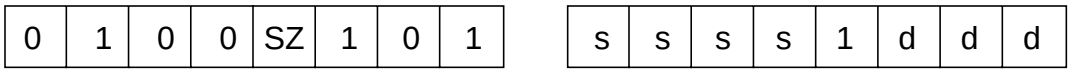
Bytes: 4
Clocks: 6
Operation: (Rd) - ((WS:Rs)+offset16)
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

CMP [Rd+offset16], Rs

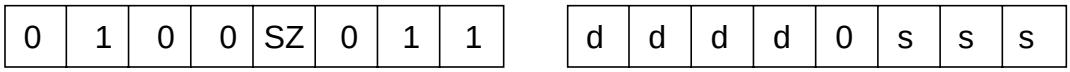
Bytes: 4
Clocks: 6
Operation: ((WS:Rd)+offset16) - (Rs)
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

CMP Rd, [Rs+]

Bytes: 2
Clocks: 5
Operation: (Rd) - ((WS:Rs))
(Rs) <-- (Rs) + 1 (byte operation) or 2 (word operation)
Encoding:

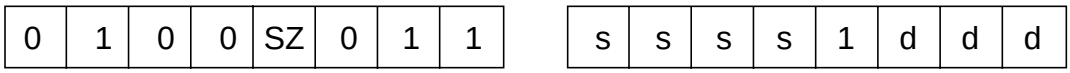


CMP [Rd+], Rs

Bytes: 2
Clocks: 5

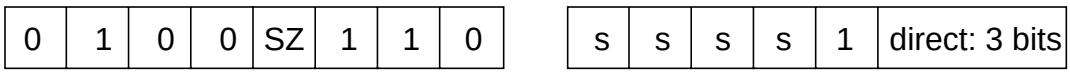
Operation: ((WS:Rd)) - (Rs)
(Rd) <-- (Rd) + 1 (byte operation) or 2 (word operation)

Encoding:



CMP direct, Rs

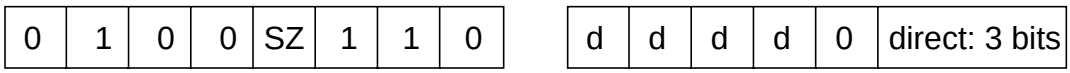
Bytes: 3
Clocks: 4
Operation: (direct) - (Rs)
Encoding:



byte 3: lower 8 bits of direct

CMP Rd, direct

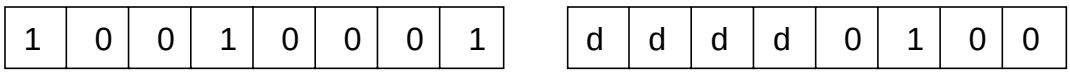
Bytes: 3
Clocks: 4
Operation: (Rd) - (direct)
Encoding:



byte 3: lower 8 bits of direct

CMP Rd, #data8

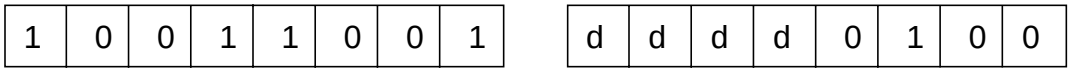
Bytes: 3
Clocks: 3
Operation: (Rd) - #data8
Encoding:



byte 3: #data8

CMP Rd, #data16

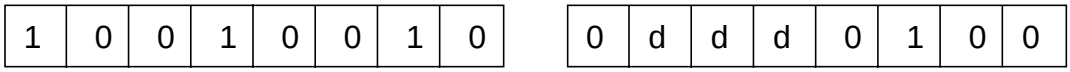
Bytes: 4
Clocks: 3
Operation: (Rd) - #data16
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

CMP [Rd], #data8

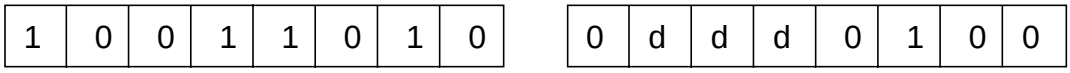
Bytes: 3
Clocks: 4
Operation: ((WS:Rd)) - #data8
Encoding:



byte 3: #data8

CMP [Rd], #data16

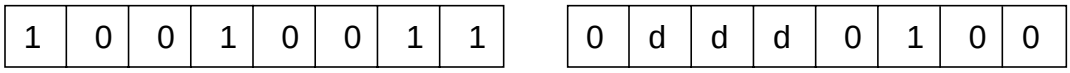
Bytes: 4
Clocks: 4
Operation: ((WS:Rd)) - #data16
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

CMP [Rd+], #data8

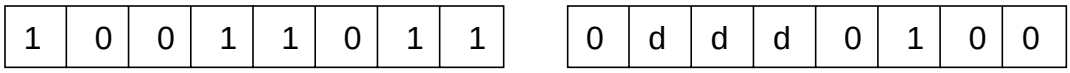
Bytes: 3
Clocks: 5
Operation: ((WS:Rd)) - #data8
(Rd) <-- (Rd) + 1
Encoding:



byte 3: #data8

CMP [Rd+], #data16

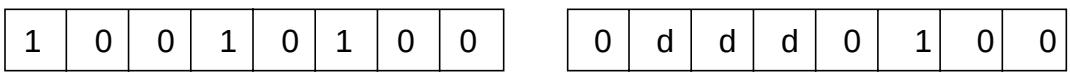
Bytes: 4
Clocks: 5
Operation: ((WS:Rd)) - #data16
(Rd) <-- (Rd) + 2
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

CMP [Rd+offset8], #data8

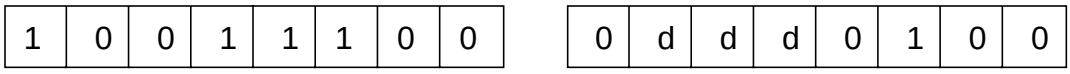
Bytes: 4
Clocks: 6
Operation: ((WS:Rd)+offset8) - #data8
Encoding:



byte 3: offset8
byte 4: #data8

CMP [Rd+offset8], #data16

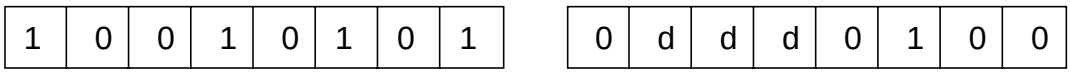
Bytes: 5
Clocks: 6
Operation: ((WS:Rd)+offset8) - #data16
Encoding:



byte 3: offset8
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

CMP [Rd+offset16], #data8

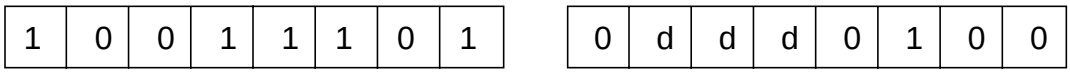
Bytes: 5
Clocks: 6
Operation: ((WS:Rd)+offset16) - #data8
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: #data8

CMP [Rd+offset16], #data16

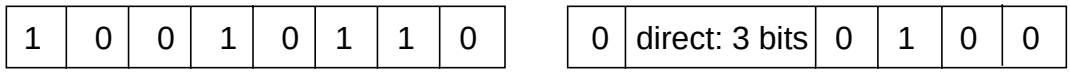
Bytes: 6
Clocks: 6
Operation: ((WS:Rd)+offset16) - #data16
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: upper 8 bits of #data16
byte 6: lower 8 bits of #data16

CMP direct, #data8

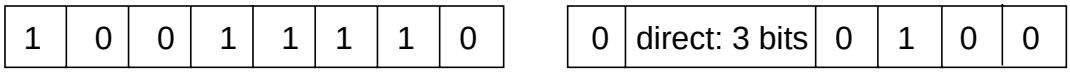
Bytes: 4
Clocks: 4
Operation: (direct) - #data8
Encoding:



byte 3: lower 8 bits of direct
byte 4: #data8

CMP direct, #data16

Bytes: 5
Clocks: 4
Operation: (direct) - #data16
Encoding:



byte 3: lower 8 bits of direct
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

Syntax: CPL Rd

Operation: Rd <-- ($\overline{Rd}$)

Description: Performs a ones complement of the destination operand specified by the register Rd. The result is stored back into Rd. The destination may be either a byte or a word.

Size: Byte, Word

Flags Updated: N, Z

Bytes: 2
Clocks: 3

Encoding:

1	0	0	1	SZ	0	0	0

d	d	d	d	1	0	1	0

Syntax: DA Rd

Operation: if (Rd.3-0) > 9 or (AC) = 1
then (Rd.3-0) <-- (Rd.3-0) + 6
if (Rd.7-4) > 9 or (C) = 1
then (Rd.7-4) <-- (Rd.7-4) + 6

Description: Adjusts the destination register to BCD format (binary-coded decimal) following an ADD or ADDC operation on BCD values. This operation may only be done on a byte register.

If the lower 4 bits of the destination value are greater than 9, or if the AC flag is set, 6 is added to the value. This may cause the carry flag to be set if this addition caused a carry out of the upper 4 bits of the value.

If the upper 4 bits of the destination value are greater than 9, or if the carry flag was set by the add to the lower bits, 60 hex is added to the value. This may cause the carry flag to be set if this addition caused a carry out of the upper 4 bits of the value. Carry will never be cleared by the DA instruction if it was already set.

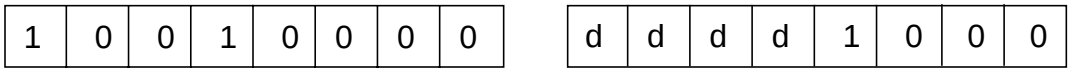
Size: Byte

Flags Updated: C, N, Z

The carry flag may be set but not cleared. See the description of the carry flag update above.

Bytes: 2
Clocks: 4

Encoding:



Note: Please refer to the table on the next page.

The following table shows the possible actions that may occur during the DA instruction, related to the input conditions.

Table 6.6

Low nibble (bits 3-0)	AC	Carry to high nibble	High nibble (bits 7-4)	Initial C flag	Number added to value	Resulting C flag
0 - 9	0	0	0 - 9	0	00	0
A - F	0	1	0 - 8	0	06	0
0 - 3 *	1	0	0 - 9	0	06	0
0 - 9	0	0	A - F	0	60	1
A - F	0	1	9 - F	0	66	1
0 - 3 *	1	0	A - F	0	66	1
0 - 9	0	0	0 - 2 **	1	60	1
A - F	0	1	0 - 2 **	1	66	1
0 - 3 *	1	0	0 - 3 ***	1	66	1

: The largest digit that could result from adding two BCD digits that caused the AC flag to be set is 3. This is with an ADDC instruction where 9 + 9 + 1 (the carry flag) = 13 hex.

** : The largest digit that could result in the upper nibble of a value by adding two BCD bytes, with no carry from the bottom nibble (the AC flag = 0) is 2. For instance, 98 hex + 97 hex = 12F hex.

*** : The largest digit that could result in the upper nibble of a value by adding two BCD bytes, with a carry from the bottom nibble (the AC flag = 1) is 3. For instance, 99 hex + 99 hex = 132 hex.

DIV.w	**16x8**	**Signed Division**
DIV.d	**32x16**	**Signed Division**
DIVU.b	**8x8**	**Unsigned Division**
DIVU.w	**16x8**	**Unsigned Division**
DIVU.d	**32x16**	**Unsigned Division**

Description: The byte or word specified by the source operand is divided into the variable specified by the destination operand.

For DIVU.b, the destination operand can be any byte register that is the least significant byte of a word register. For DIV.w and DIVU.w, the destination operand must be a word register, and for DIV.d and DIVU.d, the destination operand must identify a word register that is the low-word of a double-word register (see note below). The result is stored in the destination register as the quotient (8 bits for DIVU.b, DIVU.w, DIV.w, and DIVU.w, and 16-bits for DIV.d and DIVU.d) in the least significant half and the remainder (same size as the quotient), in the most significant half (except for DIVU.b which stores the quotient in the destination as identified by the lower half of a word register and the remainder at upper half of the same word register).

Note: a double word register is double-word aligned in the register file (R1:R0, R3:R2, R5:R4, or R7:R6).

Size: Byte-Byte, Word-Byte, Double word-Word

Flags Updated: C, V, N, Z

The carry flag is always cleared. The V flag is set in the following cases, otherwise it is cleared:

- DIVU.b: V is set if a divide by 0 occurred. A divide by 0 also causes a hardware trap to be generated.
- DIV.w, DIVU.w: V is set if the result of the divide is larger than 8 bits (the result does not fit in the destination).
- DIV.d, DIVU.d: V is set if the result of the divide is larger than 16 bits (the result does not fit in the destination).

The Z, and N flags are set based on the quotient (integer) portion of the result only and not on the remainder.

Examples:

a) DIVU.b R4L, R4H - will store the result of the division of R4L by R4H in R4L and R4H (quotient in register R4L, remainder in register R4H).

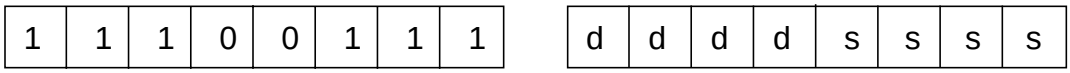
b) DIV.w R0, R2L - will store the result of word register R0 divided by byte register R2L in word register R0 (quotient in register R0L, remainder in register R0H).

c) DIV.d R4,R2 - will store the result of double-word register R5:R4 divided by word register R2 in double-word register R5:R4 (quotient in R4, remainder in R5)

Note: For all divides except DIVU.b, the destination register size is the same as indicated by the instruction (by the ".b", ".w", or ".d") and the source register is half that size.

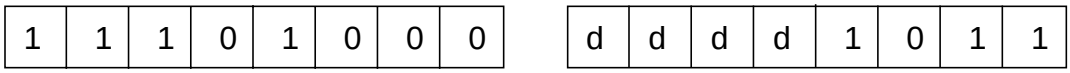
DIV.w Rd, Rs
(signed 16 bits / 8 bits --> 8 bit quotient, 8 bit remainder)

Bytes: 2
Clocks: 14
Operation: (RdL) <-- 8-bit integer portion of (Rd) / (Rs) (signed divide)
(RdH) <-- 8-bit remainder of (Rd) / (Rs)
Encoding:



DIV.w Rd, #data8
(signed 16 bits / 8 bits --> 8 bit quotient, 8 bit remainder)

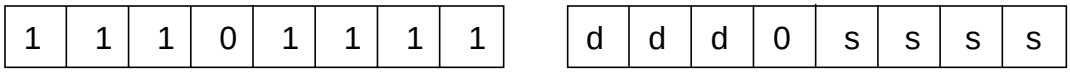
Bytes: 3
Clocks: 14
Operation: (RdL) <-- 8-bit integer portion of (Rd) / #data8 (signed divide)
(RdH) <-- 8-bit remainder of (Rd) / #data8
Encoding:



byte 3: #data8

DIV.d Rd, Rs
(signed 32 bits / 16 bits --> 16 bit quotient, 16 bit remainder)

Bytes: 2
Clocks: 24
Operation: (Rd) <-- 16-bit integer portion of (Rd) / (Rs) (signed divide)
(Rd+1) <-- 16-bit remainder of (Rd) / (Rs)
Encoding:



DIV.d Rd, #data16
(signed 32 bits / 16 bits --> 16 bit quotient, 16 bit remainder)

Bytes: 4
Clocks: 24
Operation: (Rd) <-- 16-bit integer portion of (Rd) / #data16 (signed divide)
(Rd+1) <-- 16-bit remainder of (Rd) / #data16
Encoding:

1	1	1	0	1	0	0	1	d	d	d	0	1	0	0	1

byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

DIVU.b Rd, Rs
(unsigned 8 bits / 8 bits --> 8 bit quotient, 8 bit remainder)

Bytes: 2
Clocks: 12
Operation: (RdL) <-- 8-bit integer portion of (RdL) / (Rs) (unsigned divide)
(RdH) <-- 8-bit remainder of (RdL) / (Rs)
Encoding:

1	1	1	0	0	0	0	1	d	d	d	d	s	s	s	s

DIVU.b Rd, #data8
(unsigned 8 bits / 8 bits --> 8 bit quotient, 8 bit remainder)

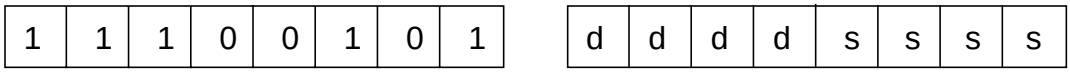
Bytes: 3
Clocks: 12
Operation: (RdL) <-- 8-bit integer portion of (RdL) / #data8 (unsigned divide)
(RdH) <-- 8-bit remainder of (RdL) / #data8
Encoding:

1	1	1	0	1	0	0	0	d	d	d	d	0	0	0	1

byte 3: #data8

DIVU.w Rd, Rs
(unsigned 16 bits / 8 bits --> 8 bit quotient, 8 bit remainder)

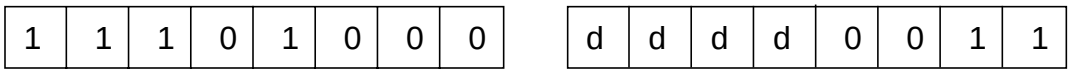
Bytes: 2
Clocks: 12
Operation: (RdL) <-- 8-bit integer portion of (Rd) / (Rs) (unsigned divide)
(RdH) <-- 8-bit remainder of (Rd) / (Rs)
Encoding:



DIVU.w Rd, #data8
(unsigned 16 bits / 8 bits --> 8 bit quotient, 8 bit remainder)

Bytes: 3
Clocks: 12

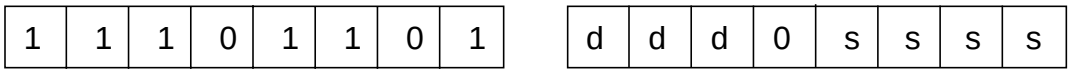
Operation: (RdL) <-- 8-bit integer portion of (Rd) / #data8 (unsigned divide)
(RdH) <-- 8-bit remainder of (Rd) / #data8
Encoding:



byte 3: #data8

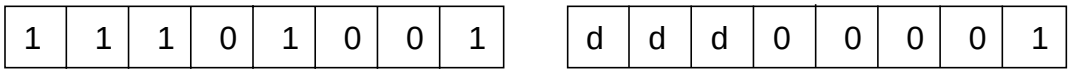
DIVU.d Rd, Rs
(unsigned 32 bits / 16 bits --> 16 bit quotient, 16 bit remainder)

Bytes: 2
Clocks: 22
Operation: (Rd) <-- 16-bit integer portion of (Rd) / (Rs) (unsigned divide)
(Rd+1) <-- 16-bit remainder of (Rd) / (Rs)
Encoding:



DIVU.d Rd, #data16
(unsigned 32 bits / 16 bits --> 16 bit quotient, 16 bit remainder)

Bytes: 4
Clocks: 22
Operation: (Rd) <-- 16-bit integer portion of (Rd) / #data16 (unsigned divide)
(Rd+1) <-- 16-bit remainder of (Rd) / #data16
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

DJNZ Decrement and jump if not zero

Syntax: DJNZ dest, rel8

Operation: (PC) <-- (PC) + 3
(dest) <-- (dest) - 1
if (Z) = 0 then
(PC) <-- (PC + rel8*2); (PC.0) <-- 0

Description: Controls a loop of instructions. The parameters are: a condition code (Z), a counter (register or memory), and a displacement value. The instruction first decrements the counter by one, tests the condition if the result of decrement is 0 (for termination of the loop); if it is false, execution continues with the next instruction. If true, execution branches to the location indicated by the current value of the PC plus the sign extended displacement. The value in the PC is the address of the instruction following DJNZ.

The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.The destination operand could be byte or word.

Note: Refer to section 6.3 for details of jump range

Size: Byte, Word

Flags Updated: N, Z

DJNZ Rd, rel8

Bytes: 3
Clocks: 8t/5nt
Encoding:

1	0	0	0	SZ	1	1	1	d	d	d	d	1	0	0	0

byte 3: rel8

DJNZ direct, rel8

Bytes: 4
Clocks: 9t/5nt
Encoding:

1	1	1	0	SZ	0	1	0	0	0	0	0	1	direct: 3 bits

byte 3: lower 8 bits of direct
byte 4: rel8

Syntax: FCALL addr24

Operation: (PC) <-- (PC) + 4
(SP) <-- (SP) - 4
((SP)) <-- (PC)
(PC.23-0) <-- addr24
(PC.0) <-- 0

Description: Causes an unconditional branch to the absolute memory location specified by the second operand, anywhere in the 16 megabytes XA address space. The 24-bit return address (the address following the CALL instruction) of the calling routine is saved on the stack. The target address must be word aligned as CALL or branch will force PC.bit0 to 0.

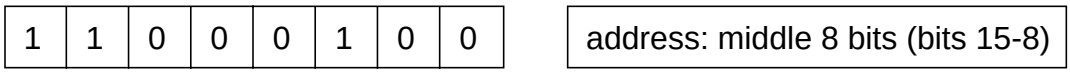
Note: if the XA is in page 0 mode, only a 16-bit address will be pushed to the stack.

Size: None

Flags Updated: none

Bytes: 4
Clocks: 12/8(PZ)

Encoding:



byte 3: lower 8 bits of address (bits 7-0)
byte 4: upper 8 bits of address (bits 23-16)

Syntax: FJMP addr24

Operation: (PC.23-0) <-- addr24
(PC.0) <-- 0

Description: Causes an unconditional branch to the absolute memory location specified by the second operand, anywhere in the 16 megabytes XA address space.
Note: The target address must be word aligned as JMP always forces PC to an even address.

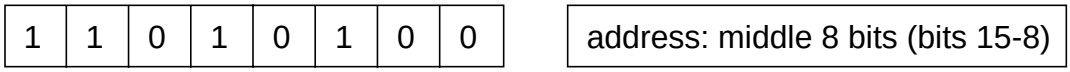
Note: if the XA is in page 0 mode, only 16-bits of the address will be used.

Size: None

Flags Updated: none

Bytes: 4
Clocks: 6

Encoding:



byte 3: lower 8 bits of address (bits 7-0)
byte 4: upper 8 bits of address (bits 23-16)

Syntax: JB bit, rel8

Operation: (PC) <-- (PC) + 4
if (bit) = 1 then
(PC) <-- (PC + rel8*2);
(PC.0) <-- 0

Description: If the specified bit is a one, program execution jumps at the location of the PC, plus the specified displacement. If the specified bit is clear, the instruction following JB is executed.The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

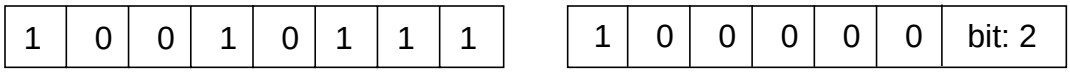
Note: Refer to section 6.3 for details of jump range

Size: Bit

Flags Updated: none

Bytes: 4
Clocks: 10t/6nt

Encoding:



byte 3: lower 8 bits of bit address
byte 4: rel8

Syntax: JBC bit, rel8

Operation: (PC) <-- (PC) + 4
if (bit) = 1 then
(PC) <-- (PC + rel8*2);
(PC.0) <-- 0; (bit) <-- 0

Description: If the bit specified is set, branch to the address pointed to by the PC plus the specified displacement. The specified bit is then cleared allowing implementation of semaphore operations. If the specified bit is clear, the instruction following JBC is executed. The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

Note: Refer to section 6.3 for details of jump range

Size: Bit

Flags Updated: none

Bytes: 4
Clocks: 11t/7nt

Encoding:

1	0	0	1	0	1	1	1

1	1	0	0	0	0	bit: 2

byte 3: lower 8 bits of bit address
byte 4: rel8

Syntax: JMP rel16

Operation: (PC) <-- (PC) + 3
(PC) <-- (PC + rel16*2)
(PC.0) <-- 0

Description: Jumps unconditionally. The branch range is +65,535 bytes to -65,536 bytes, with the limitation that the target address is word aligned in code memory.

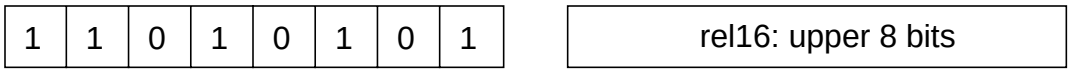
Note: Refer to section 6.3 for details of jump range

Size: None

Flags Updated: none

Bytes: 3
Clocks: 6

Encoding:



byte 3: lower 8 bits of rel16

JMP **Jump Indirect through Register**

Syntax: JMP [Rs]

Operation: (PC) <-- (PC) + 2
(PC.15-1) <-- (Rs.15-1) (note that PC.23-16 is not affected)
(PC.0) <-- 0

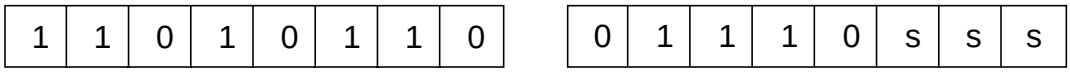
Description: Causes an unconditional branch to the address contained in the operand word register, anywhere within the 64K code page following the JMP instruction.The value of the PC used in the target address calculation is the address of the instruction following the JMP instruction.
The target address must be word aligned as JMP will force PC.bit0 to 0.

Size: none

Flags Updated: none

Bytes: 2
Clocks: 7

Encoding:



Syntax: JMP [A+DPTR]

Operation: (PC) <-- (PC) + 2
(PC15-1) <-- (A) + (DPTR)
(PC.0) <-- 0

Description: Causes an unconditional branch to the address formed by the sum of the 80C51 compatibility registers A and DPTR, anywhere within the 64K code page following the JMP instruction. This instruction is included for 80C51 compatibility. See Chapter 9 for details of 80C51 compatibility features.

Note: The target address must be word aligned as JMP will force PC.bit0 to 0.

Flags Updated: none

Bytes: 2
Clocks: 5

Note: A and DPTR are pre-defined registers used for 80C51 code translation.

Encoding:

1	1	0	1	0	1	1	0

0	1	0	0	0	1	1	0

JMP **Jump double indirect**

Syntax: JMP [[Rs+]]

Operation: (PC) <-- (PC) + 2
(PC.15-0) <-- code memory ((WS:Rs))
(PC.0) <-- 0
(Rs) <-- (Rs) + 2

Description: Causes an unconditional branch to the address contained in memory at the address pointed to by the register specified in the instruction. The specified register is post-incremented.

This 2-byte instruction may be used to compress code size by using it to index through a table of procedure addresses that are accessed in sequence. Each procedure would end with another JMP [[R+]] that would immediately go to the next procedure whose address is in the table.

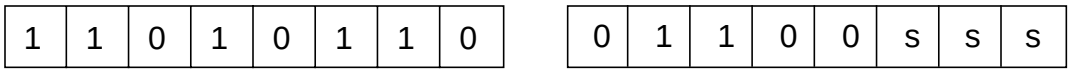
The procedures must be located in the same 64K address page of the executed "Jump Double-indirect" instruction (although the table could be in any page). This instruction can result in substantial code compression and hence cost reduction through smaller memory requirements. The register pointer (index to the table) being automatically post-incremented after the execution of the instruction. The 24-bit address is identified by combining the low order 16-bit of the PC and either of high 8-bits of PC or the contents of a byte-size CS register as chosen by the program through a segment select Special Function Register (SFR).

Note: The subroutine addresses must be word aligned as JMP will force PC.bit0 to 0.

Flags Updated: none

Bytes: 2
Clocks: 8

Encoding:



Syntax: JNB bit, rel8

Operation: (PC) <-- (PC) + 4
if (bit) = 0 then
(PC.15-0) <-- (PC + rel8*2); (PC.0) <-- 0

Description: If the specified bit is a zero, program execution jumps at the location of the PC, plus the specified displacement. If the specified bit is set, the instruction following JB is executed. The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

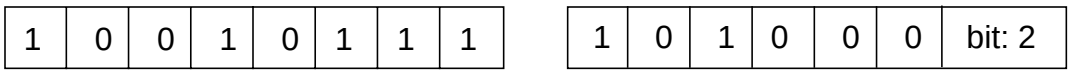
Note: Refer to section 6.3 for details of jump range

Size: Bit

Flags Updated: none

Bytes: 4
Clocks: 10t/6nt

Encoding:



byte 3: lower 8 bits of bit address
byte 4: rel8

JNZ **Jump if the A register is not zero**

Syntax: JNZ rel8

Operation: (PC) <-- (PC) + 2
if (A) not equal to 0, then
(PC.15-0) <-- (PC + rel8*2); (PC.0) <-- 0

Description: A relative branch is taken if the contents of the 80C51 Accumulator are not zero. The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

The contents of the accumulator remain unaffected. This instruction is included for 80C51 compatibility. See Chapter 9 for details of 80C51 compatibility features.

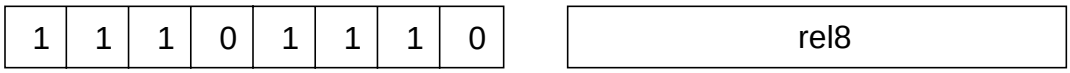
Note: Refer to section 6.3 for details of jump range

Size: Bit

Flags Updated: none

Bytes: 2
Clocks: 6t/3nt

Encoding:



JZ **Jump if the A register is zero**

Syntax: JZ rel8

Operation: (PC) <-- (PC) + 2
If (A) = 0 then
(PC.15-0) <-- (PC + rel8*2);
(PC.0) <-- 0

Description: A relative branch is taken if the contents of the 80C51 Accumulator are zero. The branch range is +254 bytes to -256 bytes, with the limitation that the target address is word aligned in code memory.

The contents of the accumulator remain unaffected. This instruction is included for 80C51 compatibility. See Chapter 9 for details of 80C51 compatibility features.

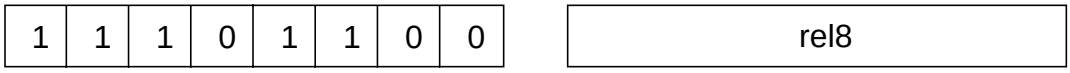
Note: Refer to section 6.3 for details of jump range

Size: Bit

Flags Updated: none

Bytes: 2
Clocks: 6t/3nt

Encoding:



Syntax: LEA Rd, Rs+offset8/16

Operation: (Rd) <-- (Rs)+offset8/16

Description: The word specified by the source operand is added to the offset value and the result is stored into the register specified by the destination operand. The source and destination operands are both registers. The offset value is an immediate data field of either 8 or 16 bits in length. The source data is not affected by the operation.

This instruction mimics the address calculation done during other instructions when the register indirect with offset addressing mode is used, allowing the resulting address to be saved for other purposes.

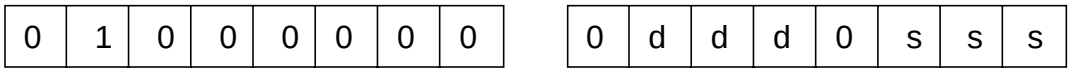
Note: The result of this operation is always a word since it duplicates the calculation of the indirect with offset addressing mode.

Size: Word-Word

Flags Updated: none

LEA Rd, Rs+offset8

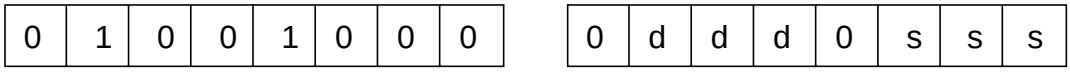
Bytes: 3
Clocks: 3
Encoding:



byte 3: offset8

LEA Rd, Rs+offset16

Bytes: 4
Clocks: 3
Operation: (Rd) <-- (Rs)+offset16
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

Syntax: LSR dest, count

Operation:

Do While (count not equal to 0)
(C) <- (dest.0)
(dest.bit n) <- (dest.bit n+1)
(dest.msb) <- 0
count = count-1
End While

Description: If the count operand is greater than the variable specified by the destination operand is logically shifted right by the number of bits specified by the count operand. The MSBs of the result are filled with zeroes.The low-order bits are shifted out through the C (carry) bit. If the count operand is 0, no shift is performed.The count operand is a positive value which may be from 0 to 31. The data size may be 8, 16, or 32 bits. In the case of 32-bit shifts, the destination operand must be the least significant half of a double word register. The count is not affected by the operation.

Note:
- For Logical Shift Left, use ASL ignoring the N flag.
- If shift count (count in Rs) exceeds data size, the count value is truncated to 5 bits, else for immediate shift count, shifting is continued until count is 0.
- a double word register is double-word aligned in the register file (R1:R0, R3:R2, R5:R4, or R7:R6).

Size: Byte, Word, Double Word

Flags Updated: C, N, Z (N = 0 after an LSR unless count = 0, then it is unchanged)

LSR Rd, Rs (Rs = Byte-register)

Operation**:**

Bytes**:** 2
Clocks**:** For 8/16 bit shifts --> 4+1 for each 2 bits of shift
For 32 bit shifts --> 6+1 for each 2 bits of shift

Encoding**:**

1	1	0	0	SZ1	SZ0	0	0

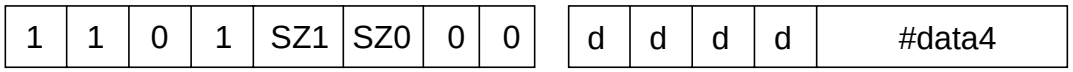
d	d	d	d	s	s	s	s

LSR Rd, #data4
Rd, #data5

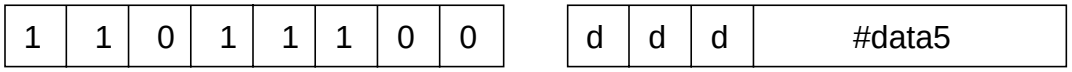
Operation:

Bytes: 2
Clocks: For 8/16 bit shifts --> 4+1 for each 2 bits of shift
For 32 bit shifts --> 6+1 for each 2 bits of shift

Encoding: (for byte and word data sizes)



(for double word data size)



Note: SZ1/SZ0 = 00: byte operation; SZ1/SZ0 = 01: reserved; SZ1/SZ0 = 10: word operation; SZ1/SZ0 = 11: double word operation.

Syntax: MOV dest, src

Operation: dest <- src

Description: The byte or word specified by the source operand is copied into the variable specified by the destination operand. The source data is not affected by the operation.

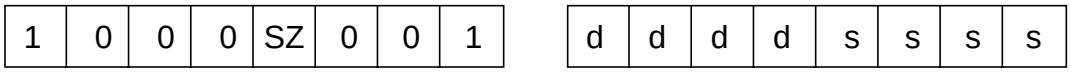
Source and destination operands may be a register in the register file, an indirect address specified by a pointer register, an indirect address specified by a pointer register added to an immediate offset of 8 or 16 bits, or a direct address. Source operands may also be specified as immediate data contained within the instruction. Auto-increment of the indirect pointers is available for simple indirect (not offset) addressing.

Size: Byte-Byte, Word-Word

Flags Updated: N, Z

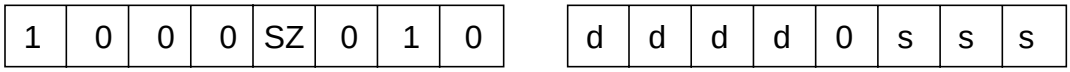
MOV Rd, Rs

Bytes: 2
Clocks: 3
Operation: (Rd) <-- (Rs)
Encoding:



MOV Rd, [Rs]

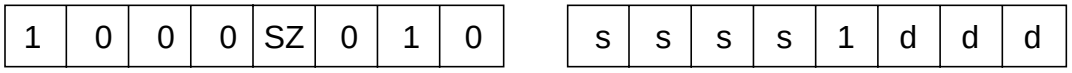
Bytes: 2
Clocks: 3
Operation: (Rd) <-- ((WS:Rs))
Encoding:



MOV [Rd], Rs

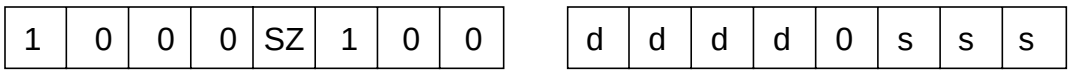
Bytes: 2
Clocks: 3

Operation: ((WS:Rd)) <-- (Rs)
Encoding:



MOV Rd, [Rs+offset8]

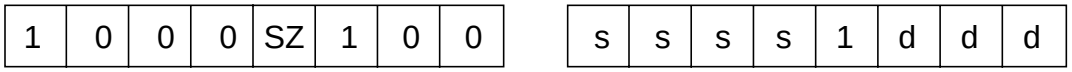
Bytes: 3
Clocks: 5
Operation: (Rd) <-- ((WS:Rs)+offset8)
Encoding:



byte 3: offset8

MOV [Rd+offset8], Rs

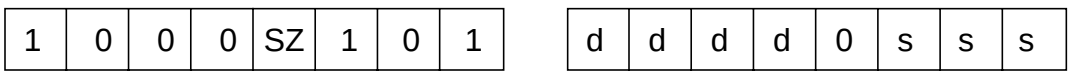
Bytes: 3
Clocks: 5
Operation: ((WS:Rd)+offset8) <-- (Rs)
Encoding:



byte 3: offset8

MOV Rd, [Rs+offset16]

Bytes: 4
Clocks: 5
Operation: (Rd) <-- ((WS:Rs)+offset16)
Encoding:

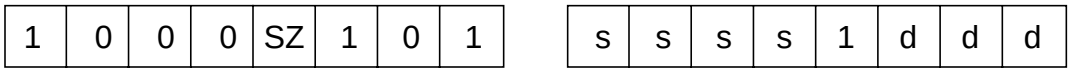


byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

MOV [Rd+offset16], Rs

Bytes: 4
Clocks: 5
Operation: ((WS:Rd)+offset16) <-- (Rs)

Encoding:

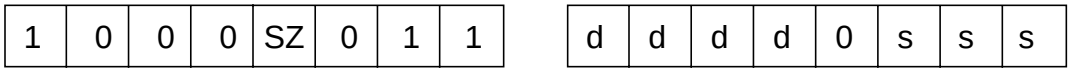


byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

MOV Rd, [Rs+]

Bytes: 2
Clocks: 4
Operation: (Rd) <-- ((WS:Rs))
(Rs) <-- (Rs) + 1 (byte operation) or 2 (word operation)

Encoding:

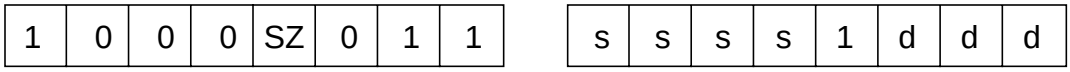


MOV [Rd+], Rs

Bytes: 2
Clocks: 4

Operation: ((WS:Rd)) <-- (Rs)
(Rd) <-- (Rd) + 1 (byte operation) or 2 (word operation)

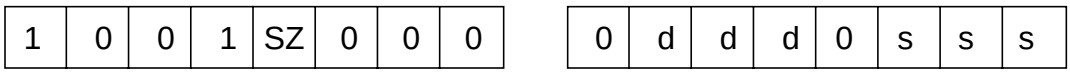
Encoding:



MOV [Rd+], [Rs+]

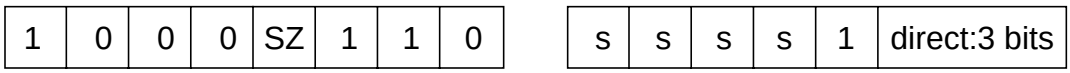
Bytes: 2
Clocks: 6
Operation: ((WS:Rd)) <-- ((WS:Rs))
(Rs) <-- (Rs) + 1 (byte operation) or 2 (word operation)
(Rd) <-- (Rd) + 1 (byte operation) or 2 (word operation)

Encoding:



MOV direct, Rs

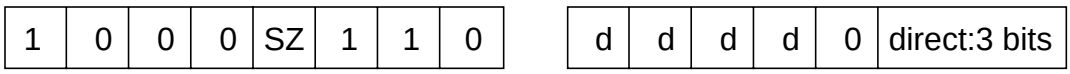
Bytes: 3
Clocks: 4
Operation: (direct) <-- (Rs)
Encoding:



byte 3: lower 8 bits of direct

MOV Rd, direct

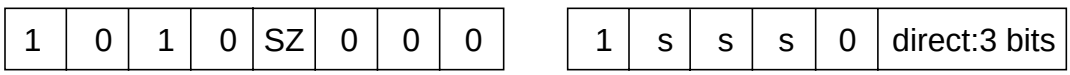
Bytes: 3
Clocks: 4
Operation: (Rd) <-- (direct)
Encoding:



byte 3: lower 8 bits of direct

MOV direct, [Rs]

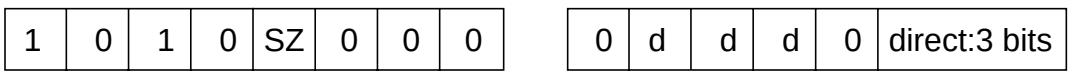
Bytes: 3
Clocks: 4
Operation: (direct) <-- ((WS:Rs))
Encoding:



byte 3: lower 8 bits of direct

MOV [Rd], direct

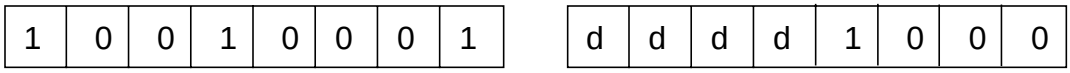
Bytes: 3
Clocks: 4
Operation: ((WS:Rd)) <-- (direct)
Encoding:



byte 3: lower 8 bits of direct

MOV Rd, #data8

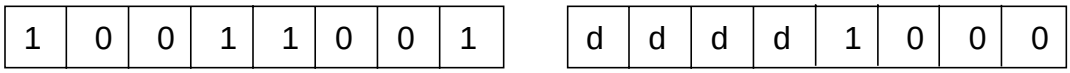
Bytes: 3
Clocks: 3
Operation: (Rd) <-- #data8
Encoding:



byte 3: #data8

MOV Rd, #data16

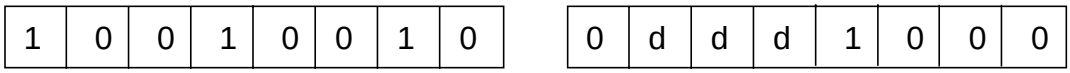
Bytes: 4
Clocks: 3
Operation: (Rd) <-- #data16
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

MOV [Rd], #data8

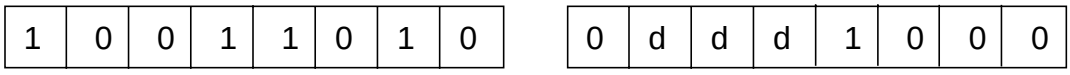
Bytes: 3
Clocks: 3
Operation: ((WS:Rd)) <-- #data8
Encoding:



byte 3: #data8

MOV [Rd], #data16

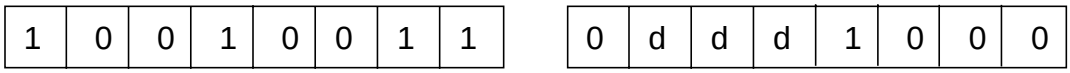
Bytes: 4
Clocks: 3
Operation: ((WS:Rd)) <-- #data16
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

MOV [Rd+], #data8

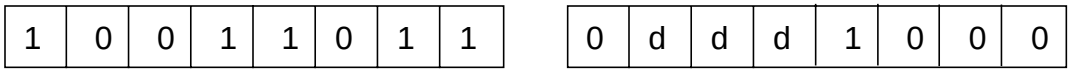
Bytes: 3
Clocks: 4
Operation: ((WS:Rd)) <-- #data8
(Rd) <-- (Rd) + 1
Encoding:



byte 3: #data8

MOV [Rd+], #data16

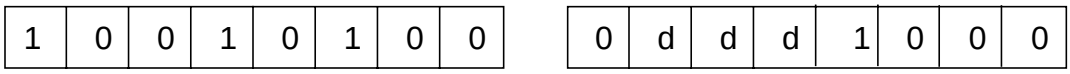
Bytes: 4
Clocks: 4
Operation: ((WS:Rd)) <-- #data16
(Rd) <-- (Rd) + 2
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

MOV [Rd+offset8], #data8

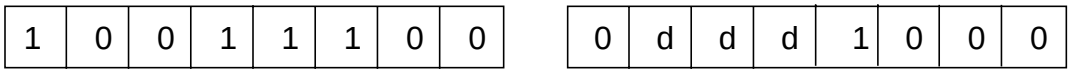
Bytes: 4
Clocks: 5
Operation: ((WS:Rd)+offset8) <-- #data8
Encoding:



byte 3: offset8
byte 4: #data8

MOV [Rd+offset8], #data16

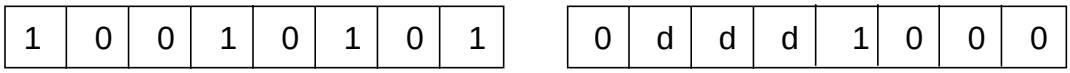
Bytes: 5
Clocks: 5
Operation: ((WS:Rd)+offset8) <-- #data16
Encoding:



byte 3: offset8
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

MOV [Rd+offset16], #data8

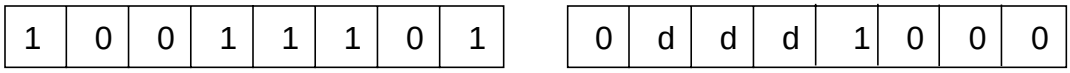
Bytes: 5
Clocks: 5
Operation: ((WS:Rd)+offset16) <-- #data8
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: #data8

MOV [Rd+offset16], #data16

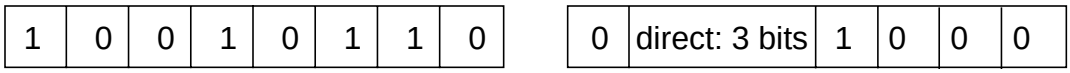
Bytes: 6
Clocks: 5
Operation: ((WS:Rd)+offset16) <-- #data16
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: upper 8 bits of #data16
byte 6: lower 8 bits of #data16

MOV direct, #data8

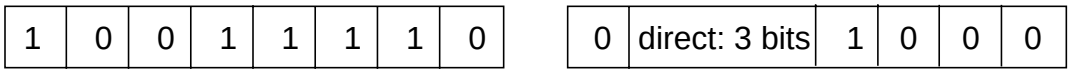
Bytes: 4
Clocks: 3
Operation: (direct) <-- #data8
Encoding:



byte 3: lower 8 bits of direct
byte 4: #data8

MOV direct, #data16

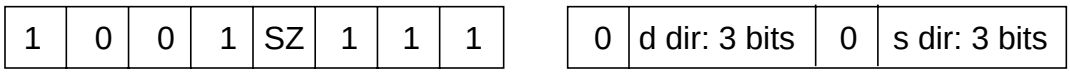
Bytes: 5
Clocks: 3
Operation: (direct) <-- #data16
Encoding:



byte 3: lower 8 bits of direct
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

MOV direct, direct

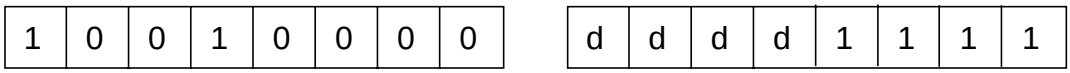
Bytes: 4
Clocks: 4
Operation: (direct) <-- (direct)
Encoding:



byte 3: lower 8 bits of direct (dest)
byte 4: lower 8 bits of direct (src)

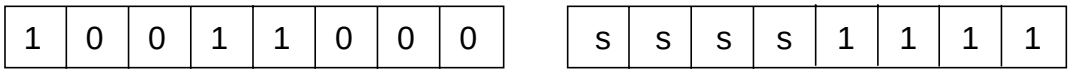
MOV Rd, USP (move from user stack pointer)

Bytes: 2
Clocks: 3
Operation: (Rd) <-- (USP)
Encoding:



MOV USP, Rs (move to user stack pointer)

Bytes: 2
Clocks: 3
Operation: (USP) <-- (Rs)
Encoding:



Syntax: MOV C, bit

Operation: (C) <-- (bit)

Description: Copies the specified bit to the carry flag.

Size: Bit

Flags Updated: none

Note: C is written as the destination of the move, not as a status flag

Bytes: 3
Clocks: 4

Encoding:

0	0	0	0	1	0	0	0

0	0	1	0	0	0	bit: 2

byte 3: lower 8 bits of bit address

Syntax: MOV bit, C

Operation: (bit) <-- (C)

Description: Copies the carry flag to the specified bit.

Size: Bit

Flags Updated: none

Bytes: 3
Clocks: 4

Encoding:

0	0	0	0	1	0	0	0

0	0	1	1	0	0	bit: 2

byte 3: lower 8 bits of bit address

Syntax: MOVC Rd, [Rs+]

Operation: (Rd) <-- code memory ((WS:Rs))
(Rs) <-- (Rs) + 1 (byte operation) or 2 (word operation)

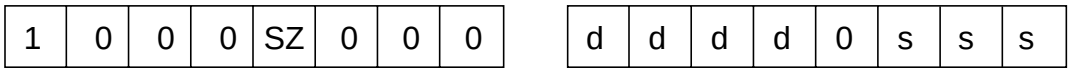
Description: Contents of code memory are copied to an internal register. The byte or word specified by the source operand is copied to the variable specified by the destination operand. In the case of MOVC, the pointer segment selection gives the choices of PC_{23-16} or CS segment (current *working segment* referred here as WS), rather than DS or ES as is used for all other instructions.

Size: Byte-Byte, Word-Word

Flags Updated: N, Z

Bytes: 2
Clocks: 4

Encoding:



Syntax: MOVC A, [A+DPTR]

Operation: PC <- PC+2
(A) <-- code memory (PC.23-16:(A) + (DPTR))

Description: The byte located at the code memory address formed by the sum of A and the DPTR is copied to the A register. The A and DPTR registers are pre-defined registers used for 80C51 compatibility. This instruction is included for 80C51 compatibility. See Chapter 9 for details of 80C51 compatibility features.

Size: Byte-Byte

Flags Updated: N, Z

Bytes: 2
Clocks: 6

Encoding:

1	0	0	1	0	0	0	0

0	1	0	0	1	1	1	0

.

Syntax: MOVC A, [A+PC]

Operation: PC <- PC+2
(A) <-- code memory [PC.23-16: (A +PC.15-0)]

Note: Only 16-bits of A+PC are used

Description: The byte located at the code memory address formed by the sum of A and the current Program Counter value is copied to the A register. The A register is a pre-defined register used for 80C51 compatibility. This instruction is included for 80C51 compatibility. See Chapter 9 for details of 80C51 compatibility features.

Size: Byte-Byte

Flags Updated: N, Z

Bytes: 2
Clocks: 6

Encoding:

1	0	0	1	0	0	0	0

0	1	0	0	1	1	0	0

Syntax: MOVS dest, #data

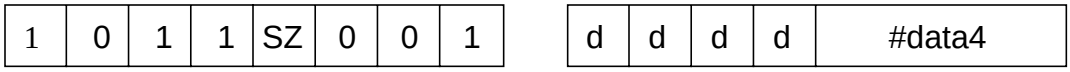
Description: Four bits of signed immediate data are moved to the destination. The immediate data is sign-extended to the proper size, then moved to the variable specified by the destination operand, which may be a byte or a word. The immediate data range is +7 to -8. This instruction is used to save time and code space for the many instances where a small data constant is moved to a destination.

Size: Byte-Byte, Word-Word

Flags Updated: N, Z

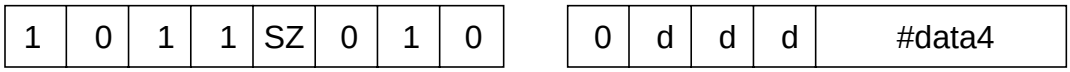
MOVS Rd, #data4

Bytes: 2
Clocks: 3
Operation: (Rd) <-- sign-extended #data4
Encoding:



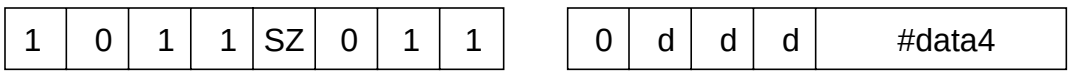
MOVS [Rd], #data4

Bytes: 2
Clocks: 3
Operation: ((WS:Rd)) <-- sign-extended #data4
Encoding:



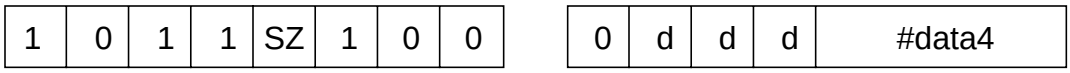
MOVS [Rd+], #data4

Bytes: 2
Clocks: 4
Operation: ((WS:Rd)) <-- sign-extended #data4
(Rd) <-- (Rd) + 1 (byte operation) or 2 (word operation)
Encoding:



MOVS [Rd+offset8], #data4

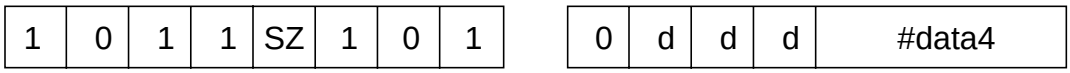
Bytes: 3
Clocks: 5
Operation: ((WS:Rd)+offset8) <-- sign-extended #data4
Encoding:



byte 3: offset8

MOVS [Rd+offset16], #data4

Bytes: 4
Clocks: 5
Operation: ((WS:Rd)+offset16) <-- sign-extended #data4
Encoding:

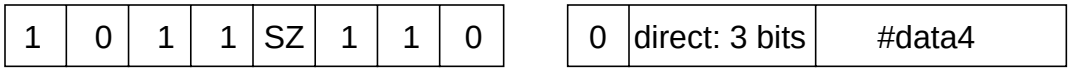


byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

MOVS direct, #data4

Bytes: 3
Clocks: 3
Operation: (direct) <-- sign-extended #data4

Encoding:



byte 3: lower 8 bits of direct

Syntax: MOVX dest, src

Description: Move external data to or from an internal register. The byte or word specified by the source operand is copied into the variable specified by the destination operand. This instruction allows access to data external to the microcontroller in the address range of 0 to 64K. The standard indirect move may access external data only above the boundary where internal data RAM ends, whereas MOVX always forces an external access. MOVX only operates on the first 64K of external data memory. This instruction is included to allow compatibility with 80C51 code.

Note that in the 80C51 MOVX instruction using @Ri as a pointer (where i could be 0 or 1), the pointer was eight bits in length and the upper address lines were not driven on the external bus. The XA always drives all of the enabled external bus address lines. The use of the pointer depends on whether compatibility mode is in use. If CM = 0 (compatibility mode off, the default), 16 bits of R0 or R1 are used as the address within data segment 0. If CM = 1 (compatibility mode on), 8 bits of R0L or R0H are used as the bottom eight bits of the address, while the remainder of the address bits, including those corresponding to the data segment are 0.

Size: Byte-Byte, Word-Word

Flags Updated: N, Z

MOVX Rd, [Rs]

Bytes: 2
Clocks: 6
Operation: (Rd) <-- external data memory ((Rs))

Encoding:

1	0	1	0	SZ	1	1	1		d	d	d	d	0	s	s	s

MOVX [Rd], Rs

Bytes: 2
Clocks: 6
Operation: external data memory ((Rd)) <-- (Rs)
Encoding:

1	0	1	0	SZ	1	1	1		s	s	s	s	1	d	d	d

MUL.w 16x16 Signed Multiply
MULU.b 8x8 Unsigned Multiply
MULU.w 16x16 Unsigned Multiply

Description: The byte or word specified by the source operand is multiplied by the variable specified by the destination operand.

The destination operand must be the first half of a double size register (word for a byte multiply and double word for a word multiply). The result is stored in the double size register.

Note: a double word register is double-word aligned in the register file (R1:R0, R3:R2, R5:R4, and R7:R6).

Size: Byte-Byte, Word-Word

Flags Updated: C, V, N, Z

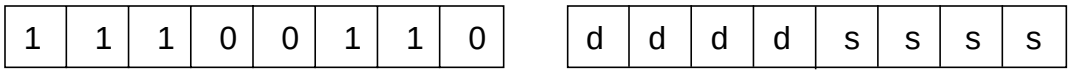
The carry flag is always cleared by a multiply instruction. The V flag is set in the following cases, otherwise it is cleared:
- MULU.b: V is set if the result of the multiply is greater than FFh (the upper byte is not equal to 0).
- MULU.w: V is set if the result of the multiply is greater than FFFFh (the upper word is not equal to 0).
- MUL.w: V is set if the absolute value of the result of the multiply is greater than 7FFFh (the upper word is not a sign extension of the lower word).

Examples:
a) MUL.w R0,R5 stores the product of word register 0 and word register 5 in double word register 0 (least significant word in word register R0, most significant word in word register R1).

b) MULU.b R4L, R4H will store the MS byte of the product of R4L and R4H in R4H and the LS byte in R4L.

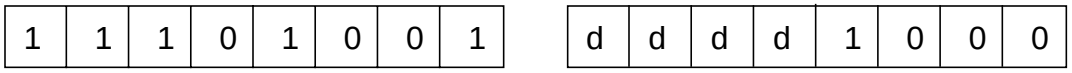
MUL.w Rd, Rs
(signed 16 bits * 16 bits --> 32 bits)

Bytes: 2
Clocks: 12
Operation: (Rd+1) <-- Most significant word of (Rd) * (Rs) (signed multiply)
(Rd) <-- Least significant word of (Rd) * (Rs)
Encoding:



MUL.w Rd, #data16
(signed 16 bits * 16 bits --> 32 bits)

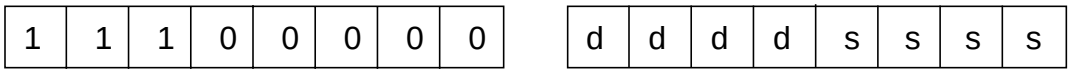
Bytes: 4
Clocks: 12
Operation: (Rd+1) <-- Most significant word of (Rd) * #data16 (signed multiply)
(Rd) <-- Least significant word of (Rd) * #data16
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

MULU.b Rd, Rs
(unsigned 8 bits * 8 bits --> 16 bits)

Bytes: 2
Clocks: 12
Operation: (RdH) <-- Most significant byte of (RdL) * (Rs) (unsigned multiply)
(RdL) <-- Least significant byte of (RdL) * (Rs)
Encoding:



MULU.b Rd, #data8
(unsigned 8 bits * 8 bits --> 16 bits)

Bytes: 3
Clocks: 12
Operation: (RdH) <-- Most significant byte of (RdL) * #data8 (unsigned multiply)
(RdL) <-- Least significant byte of (RdL) * #data8
Encoding:

1	1	1	0	1	0	0	0	d	d	d	d	0	0	0	0

byte 3: #data8

MULU.w Rd, Rs
(unsigned 16 bits * 16 bits --> 32 bits)

Bytes: 2
Clocks: 12
Operation: (Rd+1) <-- Most significant word of (Rd) * (Rs) (unsigned multiply)
(Rd) <-- Least significant word of (Rd) * (Rs)
Encoding:

1	1	1	0	0	1	0	0	d	d	d	d	s	s	s	s

MULU.w Rd, #data16
(unsigned 16 bits * 16 bits --> 32 bits)

Bytes: 4
Clocks: 12
Operation: (Rd+1) <-- Most significant word of (Rd) * #data16 (unsigned multiply)
(Rd) <-- Least significant word of (Rd) * #data16
Encoding:

1	1	1	0	1	0	0	1	d	d	d	d	0	0	0	0

byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

NEG **Negate**

Syntax: NEG Rd

Operation: Rd <-- ($\overline{Rd}$) + 1

Description: The destination register is negated (twos complement). The destination may be a byte or a word.

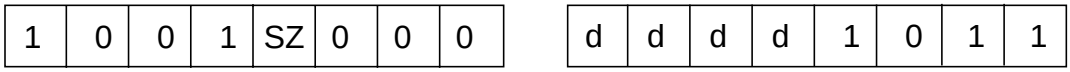
Size: Byte, Word

Flags Updated: V, N, Z

The V flag is set if a twos complement overflow occurred: the original value = result = 8000 hex for a word operation or 80 hex for a byte operation.

Bytes: 2
Clocks: 3

Encoding:



NOP **No Operation**

Syntax: NOP

Operation: PC <- PC + 1

Description: Execution resumes at the following instruction. This instruction is defined as being one byte in length in order to allow it to be used to force word alignment of instructions that are branch targets, or for any other purpose. It may also be used to as a delay for a predictable amount of time.

Size: None

Flags Updated: none

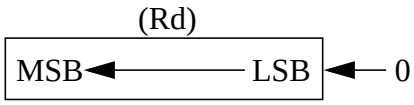
Bytes: 1
Clocks: 3

Encoding:

0	0	0	0	0	0	0	0

Syntax: NORM Rd, Rs

Operation:



Description: Logically shifts left the contents of the destination until the MSB is set, storing the number of shifts performed in the count (source) register. The data size may be 8, 16, or 32 bits.

If the destination value already has the MSB set, the count returned will be 0. If the destination value is 0, the count returned will be 0, the N flag will be cleared, and the Z flag will be set. For all other conditions, the N flag will be 1 and the Z flag will be 0.

Note: a double word register is double-word aligned in the register file (R1:R0, R3:R2, R5:R4, or R7:R6).
The last pair, i.e, R7:R6 is probably not a good idea as R7 is the current stack pointer.

Size: Byte, Word, Double Word

Flags Updated: N, Z

Bytes: 2
Clocks: For 8 or 16 bit shifts -> 4 + 1 for each 2 bits of shift
For 32 bit shifts -> 6 + 1 for each 2 bits of shift

Encoding:

1	1	0	0	SZ1	SZ0	1	1

d	d	d	d	s	s	s	s

Note: SZ1/SZ0 = 00: byte operation; SZ1/SZ0 = 01: reserved; SZ1/SZ0 = 10: word operation; SZ1/SZ0 = 11: double word operation.

Syntax: OR dest, src

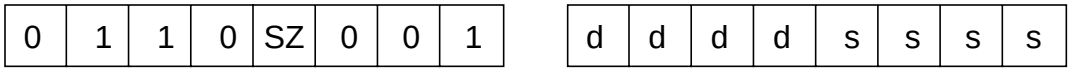
Description: Bitwise logical OR the contents of the source to the destination. The byte or word specified by the source operand is logically ORed to the variable specified by the destination operand. The source data is not affected by the operation.

Size: Byte-Byte, Word-Word

Flags Updated: N, Z

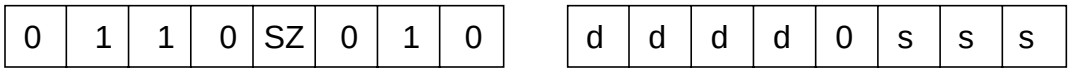
OR Rd, Rs

Bytes: 2
Clocks: 3
Operation: (Rd) <-- (Rd) + (Rs)
Encoding:



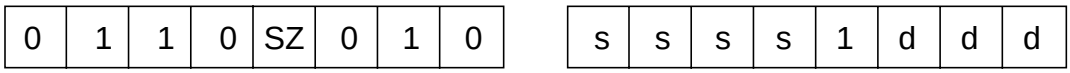
OR Rd, [Rs]

Bytes: 2
Clocks: 4
Operation: (Rd) <-- (Rd) + ((WS:Rs))
Encoding:



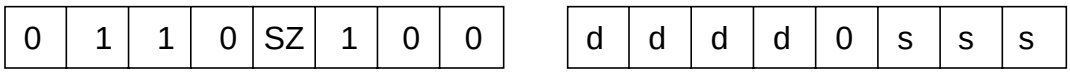
OR [Rd], Rs

Bytes: 2
Clocks: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) + (Rs)
Encoding:



OR Rd, [Rs+offset8]

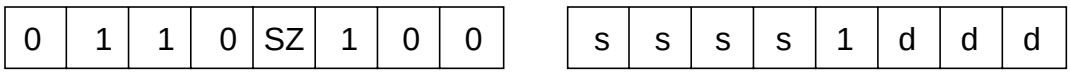
Bytes: 3
Clocks: 6
Operation: (Rd) <-- (Rd) + ((WS:Rs)+offset8)
Encoding:



byte 3: offset8

OR [Rd+offset8], Rs

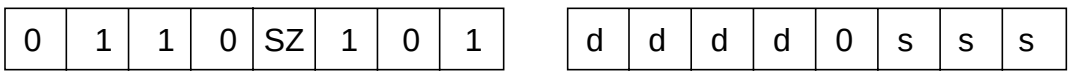
Bytes: 3
Clocks: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) + (Rs)
Encoding:



byte 3: offset8

OR Rd, [Rs+offset16]

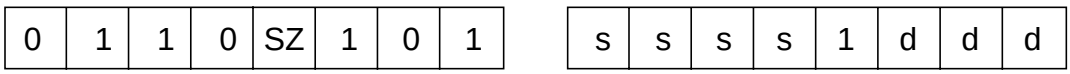
Bytes: 4
Clocks: 6
Operation: (Rd) <-- (Rd) + ((WS:Rs)+offset16)
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

OR [Rd+offset16], Rs

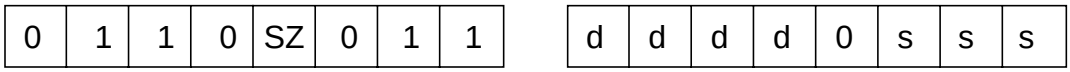
Bytes: 4
Clocks: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) + (Rs)
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

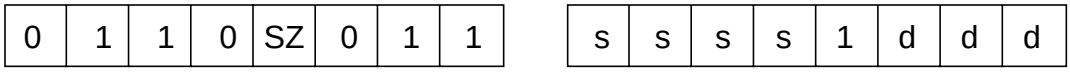
OR Rd, [Rs+]

Bytes: 2
Clocks: 5
Operation: (Rd) <-- (Rd) + ((WS:Rs))
(Rs) <-- (Rs) + 1 (byte operation) or 2 (word operation)
Encoding:



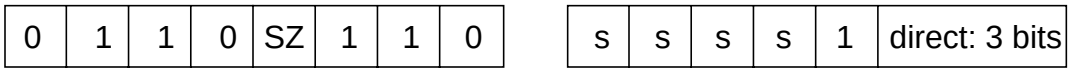
OR [Rd+], Rs

Bytes: 2
Clocks: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) + (Rs)
(Rd) <-- (Rd) + 1 (byte operation) or 2 (word operation)
Encoding:



OR direct, Rs

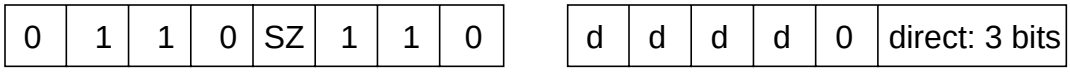
Bytes: 3
Clocks: 4
Operation: (direct) <-- (direct) + (Rs)
Encoding:



byte 3: lower 8 bits of direct

OR Rd, direct

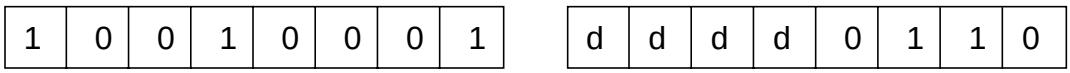
Bytes: 3
Clocks: 4
Operation: (Rd) <-- (Rd) + (direct)
Encoding:



byte 3: lower 8 bits of direct

OR Rd, #data8

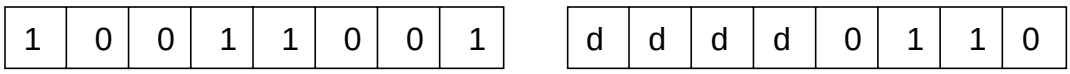
Bytes: 3
Clocks: 3
Operation: (Rd) <-- (Rd) + #data8
Encoding:



byte 3: #data8

OR Rd, #data16

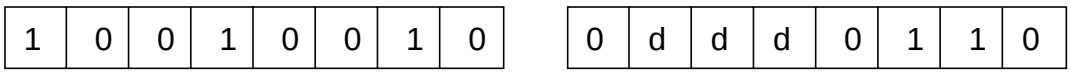
Bytes: 4
Clocks: 3
Operation: (Rd) <-- (Rd) + #data16
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

OR [Rd], #data8

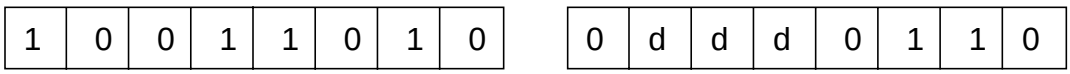
Bytes: 3
Clocks: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) + #data8
Encoding:



byte 3: #data8

OR [Rd], #data16

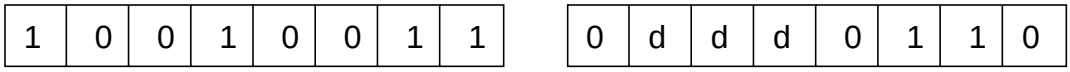
Bytes: 4
Clocks: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) + #data16
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

OR [Rd+], #data8

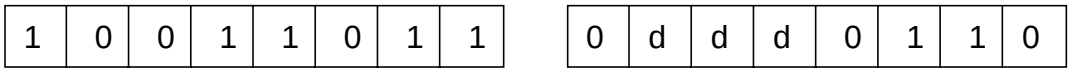
Bytes: 3
Clocks: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) + #data8
(Rd) <-- (Rd) + 1
Encoding:



byte 3: #data8

OR [Rd+], #data16

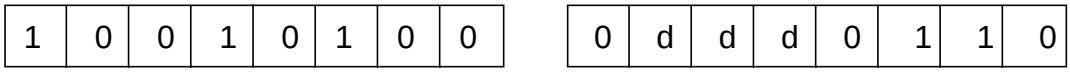
Bytes: 4
Clocks: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) + #data16
(Rd) <-- (Rd) + 2
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

OR [Rd+offset8], #data8

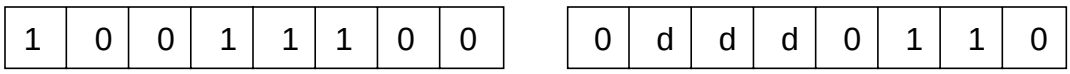
Bytes: 4
Clocks: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) + #data8
Encoding:



byte 3: offset8
byte 4: #data8

OR [Rd+offset8], #data16

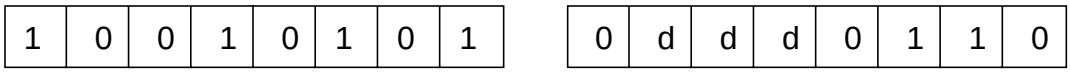
Bytes: 5
Clocks: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) + #data16
Encoding:



byte 3: offset8
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

OR [Rd+offset16], #data8

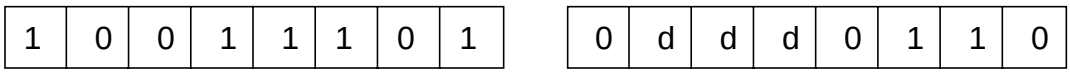
Bytes: 5
Clocks: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) + #data8
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: #data8

OR [Rd+offset16], #data16

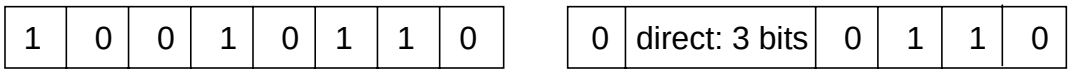
Bytes: 6
Clocks: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) + #data16
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: upper 8 bits of #data16
byte 6: lower 8 bits of #data16

OR direct, #data8

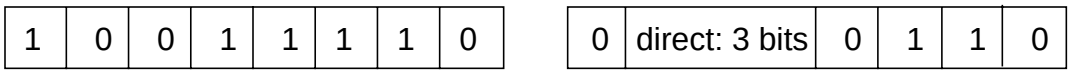
Bytes: 4
Clocks: 4
Operation: (direct) <-- (direct) + #data8
Encoding:



byte 3: lower 8 bits of direct
byte 4: #data8

OR direct, #data16

Bytes: 5
Clocks: 4
Operation: (direct) <-- (direct) + #data16
Encoding:



byte 3: lower 8 bits of direct
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

Syntax: ORL C, bit

Operation:(C) <-- (C) + (bit)

Description: Logical (inclusive) OR a bit to the Carry flag. Read the specified bit and logically OR it to the Carry flag.
(C is written as the destination of the ORL, not as a status flag)

Size: Bit

Flags Updated: none

Bytes: 3
Clocks: 4

Encoding:

0	0	0	0	1	0	0	0

0	1	1	0	0	0	bit: 2

byte 3: lower 8 bits of bit address

ORL **Logical OR complement of bit**

Syntax: ORL C, /bit

Operation: (C) <-- (C) + ($\overline{bit}$)

Description: Logically OR the complement of a bit to the Carry flag. Read the specified bit, complement it, and logically OR it to the Carry flag.
(C is written as the destination of the move, not as a status flag)

Flags Updated: none

Bytes: 3
Clocks: 4

Encoding:

0	0	0	0	1	0	0	0

0	1	1	1	0	0	bit: 2

byte 3: lower 8 bits of bit address

POP **Pop**
POPU **Pop User**

Syntax: POP dest

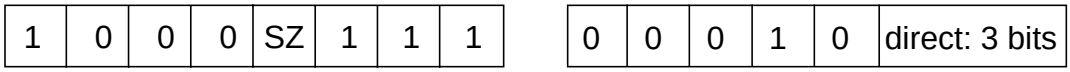
Description: The stack is popped and the data written to the specified directly addressed location. The data size may be byte or word. POP uses the current stack pointer, while POPU forces an access to the user stack.

Size: Byte, Word

Flags Updated: none

POP direct

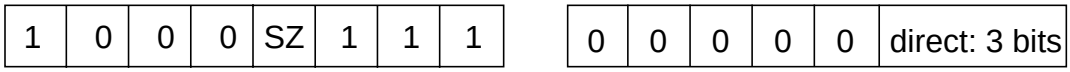
Bytes: 3
Clocks: 5
Operation: (direct) <-- ((SP))
(SP) <-- (SP) + 2
Encoding:



byte 3: 8 bits of direct

POPU direct

Bytes: 3
Clocks: 5
Operation: (direct) <-- ((USP))
(USP) <-- (USP) + 2
Encoding:



byte 3: 8 bits of direct

POP **Pop Multiple**
POPU **Pop User Multiple**

Syntax: POP Rlist
POPU Rlist

Description: Pop the specified registers (one or more) from the stack. The stack is popped (from 1 to 8 times) and the data stored in the specified registers. Any combination of word registers in the group R0 to R7 may be popped in a single instruction in a word operation. Or, any combination of byte registers in the group R0L to R3H or the group R4L to R7H may be popped in a single instruction in a byte operation. POP uses the current stack pointer, while POPU forces an access to the user stack.

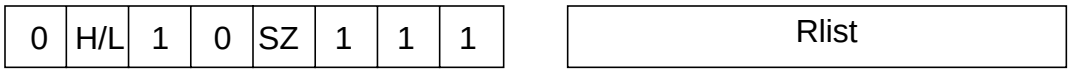
Note: Rlist is a bit map that represents each register to be popped. The registers are in the order R7, R6, R5,....., R0, for word registers or R3H.... R0L, or R7H... R4L for byte registers. The pop order is from right to left, i.e., the register specified by the rightmost one in Rlist will be popped first, etc. The order must be the reverse of that used by the preceding PUSH instruction. Note that if the same register list is used first with a PUSH, then with a POP, the original register contents will be restored. The order in which the registers are called out in the source code is not important because the Rlist operand is encoded as a fixed order bit map (see below).

Size: Byte, Word

Flags Updated: none

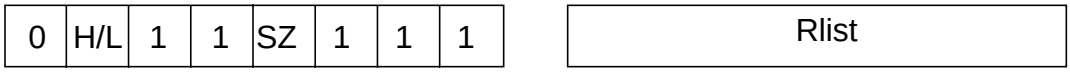
POP Rlist

Bytes: 2
Clocks: 4 + 2 per additional register
Operation: Repeat for all selected registers (Ri):
(Ri) <-- ((SP))
(SP) <-- (SP) + 2
Encoding:



POPU Rlist

Bytes: 2
Clocks: 4 + 2 per additional register
Operation: Repeat for all selected registers (Ri):
(Ri) <-- ((USP))
(USP) <-- (USP) + 2
Encoding:



Rlist bit definitions for a byte POP from register(s) in the upper register group (R4L through R7H):

R7H	R7L	R6H	R6L	R5H	R5L	R4H	R4L

Rlist bit definitions for a byte POP from register(s) in the lower register group (R0L through R3H):

R3H	R3L	R2H	R2L	R1H	R1L	R0H	R0L

Rlist bit definitions for a word POP from any register(s) (R0 through R7):

R7	R6	R5	R4	R3	R2	R1	R0

PUSH **Push**
PUSHU **Push User**

Syntax: PUSH src
PUSHU src

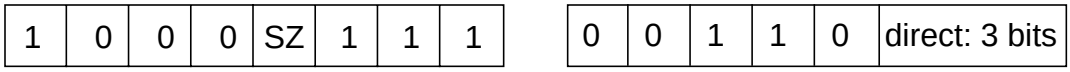
Description: The specified directly addressed data is pushed onto the stack. The data size may be byte or word. PUSH uses the current stack pointer, while PUSHU forces an access to the user stack.

Size: Byte, Word

Flags Updated: none

PUSH direct

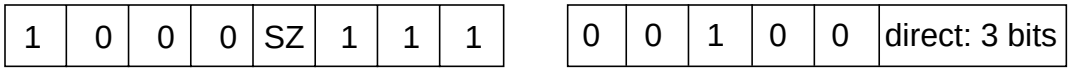
Bytes: 3
Clocks: 5
Operation: (SP) <-- (SP) - 2
((SP)) <-- (direct)
Encoding**:**



byte 3: 8 bits of direct

PUSHU direct

Bytes: 3
Clocks: 5
Operation: (USP) <-- (USP) - 2
((USP)) <-- (direct)
Encoding:



byte 3: 8 bits of direct

PUSH Push Multiple
PUSHU Push User Multiple

Syntax: PUSH Rlist
PUSHU Rlist

Description: Push the specified registers (one or more) onto the stack. The specified registers are pushed onto the stack. Any combination of word registers in the group R0 to R7 may be pushed in a single instruction in a word operation. Or, any combination of byte registers in the group R0L to R3H or the group R4L to R7H may be pushed in a single instruction in a byte operation. The data size may be byte or word. PUSH uses the current stack pointer, while PUSHU forces an access to the user stack.

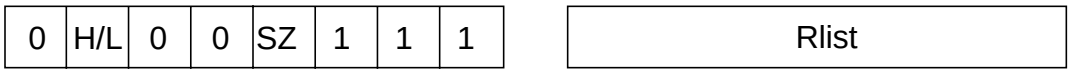
Note: Rlist is a bit map that represents each register to be pushed. The registers are in the order R7, R6, R5,....., R0, for word registers or R3H.... R0L, or R7H... R4L for byte registers. The push order is from left to right, i.e., the register specified by the leftmost one in Rlist will be pushed first, etc. The order must be the reverse of that used by the corresponding POP instruction. Note that if the same register list is used first with a PUSH, then with a POP, the original register contents will be restored. The order in which the registers are called out in the source code is not important because the Rlist operand is encoded as a fixed order bit map (see below).

Size: Byte, Word

Flags Updated: none

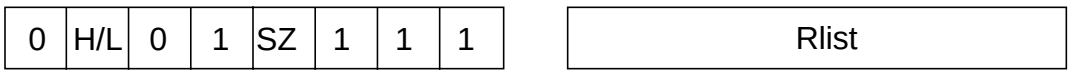
PUSH Rlist

Bytes: 2
Clocks: 3 + 3 per additional register
Operation: Repeat for all selected registers (Ri):
(SP) <-- (SP) - 2
((SP)) <-- (Ri)
Encoding:



PUSHU Rlist

Bytes: 2
Clocks: 3 + 3 per additional register
Operation: Repeat for all selected registers (Ri):
(USP) <-- (USP) - 2
((USP)) <-- (Ri)
Encoding:



Rlist bit definitions for a byte PUSH from register(s) in the upper register group (R4L through R7H):

R7H	R7L	R6H	R6L	R5H	R5L	R4H	R4L

Rlist bit definitions for a byte PUSH from register(s) in the lower register group (R0L through R3H):

R3H	R3L	R2H	R2L	R1H	R1L	R0H	R0L

Rlist bit definitions for a word PUSH from any register(s) (R0 through R7):

R7	R6	R5	R4	R3	R2	R1	R0

RESET Software Reset

Syntax: RESET

Operation: (PC) <-- vector(0)
(PSW) <-- vector(0)
(SFRs) <-- reset values (refer to the description of reset for details)

Description: The chip is reset exactly as if the external hardware reset has been asserted with the exception that it does not sample inputs for configuration, e.g., $\overline{EA}$, BUSW, etc. When a RESET instruction is executed, the chip is internally reset, but no external $\overline{RESET}$ pulse is generated. The above inputs which are latched during rising edge of a $\overline{RESET}$ pulse, hence does not affect the chip configuration.

Flags Updated: The entire PSW is set to the value specified in the reset vector.

Bytes: 2
Clocks: 18

Encoding:

1	1	0	1	0	1	1	0

0	0	0	1	0	0	0	0

Syntax: RET

Operation: (PC) <-- ((SP))
(SP) <-- (SP) + 4

Description: A 24-bit return address is popped from the stack and used to replace the entire program counter value (PC_{23-0}). This instruction is used to return from a subroutine that was called with a CALL or Far Call (FCALL).

Note: if the XA is in page 0 mode, only a 16-bit address will be popped from the stack.

Size: None

Flags Updated: none

Bytes: 2
Clocks: 8/6 (PZ)

Encoding:

1	1	0	1	0	1	1	0

1	0	0	0	0	0	0	0

Syntax: RETI

Operation: (PSW) <-- ((SSP))
(PC.23-0) <-- ((SSP))
(SSP) <-- (SSP) + 6

Description: A 24-bit return address is popped from the stack and used to replace the entire program counter value. The Program Status Word is also restored by being popped from the stack.

This instruction is a privileged instruction (limited to system mode) and is used to return from an interrupt/exception. An attempt to use RETI in user mode will generate a trap.

Note: if the XA is in page 0 mode, only a 16-bit address will be popped from the stack.

Size: None

Flags Updated: All PSW bits are written by the POP of the PSW value in System mode.

Bytes: 2
Clocks: 10/8 (PZ)

Encoding:

1	1	0	1	0	1	1	0

1	0	0	1	0	0	0	0

Syntax: RL Rd, #data4

Operation:

count <- #data4
Do While (count not equal to 0)
($dest_0$) <- ($dest_{msb}$)
($dest_n$) <- ($dest_{n-1}$)
(count) <- count -1
End While

Description: The variable specified by the destination operand is rotated left by the number of bits specified in the immediate data operand. The data size may be 8 or 16 bits. The number of bit positions shifted may be from 0 to 15.

Size: Byte, Word

Flags Updated: N, Z

Bytes: 2
Clocks: 4 + 1 for each 2 bits of shift

Encoding:

1	1	0	1	SZ	0	1	1

d	d	d	d	#data4

Syntax: RLC Rd, #data4

Operation:

count <- #data4
Do While (count not equal to 0)
(temp) <- (C)
(C) <- ($dest_{msb}$)
($dest_n$) <- ($dest_{n-1}$)
($dest_0$) <- (temp)
(count) <- count -1
End While

Description: The variable specified by the destination operand is rotated left through the carry flag by the number of bits specified in the immediate data operand. The data size may be 8 or 16 bits. The number of bit positions shifted may be from 0 to 15.

Size: Byte, Word

Flags Updated: C, N, Z

Bytes: 2
Clocks: 4 + 1 for each 2 bits of shift

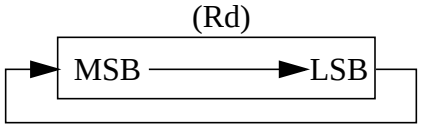
Encoding:

1	1	0	1	SZ	1	1	1

d	d	d	d	#data4

Syntax: RR Rd, #data4

Operation:



count <- #data4
Do While (count not equal to 0)
($dest_{msb}$) <- ($dest_0$)
($dest_{n-1}$) <- ($dest_n$)
(count) <- count -1
End While

Description: If the count operand is greater than 0, the destination operand is rotated right by the number of bits specified in the immediate data operand. The data size may be 8 or 16 bits. The number of bit positions shifted may be from 0 to 15. If the count operand is 0, no rotate is performed.

Size: Byte, Word

Flags Updated: N, Z

Bytes: 2
Clocks: 4 + 1 for each 2 bits of shift

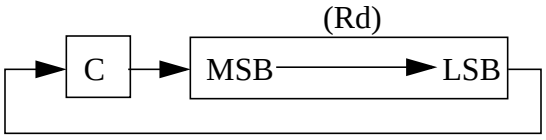
Encoding:

1	0	1	1	SZ	0	0	0

d	d	d	d	#data4

Syntax: RRC Rd, #data4

Operation:



count <- #data4
Do While (count not equal to 0)
(temp) <- (C)
(C) <- ($dest_0$)
($dest_n$) <- ($dest_{n+1}$)
($dest_{msb}$) <- (temp)
(count) <- count -1
End While

Description: If the count operand is greater than 0, the destination operand is rotated right through the carry flag by the number of bits specified in the immediate data operand. The data size may be 8 or 16 bits. The number of bit positions shifted may be from 0 to 15.
If the count operand is 0, no rotate is performed.

Size: Byte, Word

Flags Updated: C, N, Z

Bytes: 2
Clocks: 4 + 1 for each 2 bits of shift

Encoding:

1	0	1	1	SZ	1	1	1

d	d	d	d	#data4

Syntax: SETB bit

Operation: (bit) <-- 1

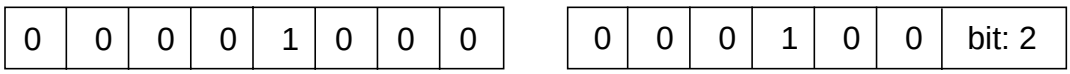
Description: Writes (sets) a 1 to the specified bit.

Size: Bit

Flags Updated:none

Bytes: 3
Clocks: 4

Encoding:



byte 3: lower 8 bits of bit address

Syntax: SEXT Rd

Operation: if N = 1
then (Rd) <-- FF in byte mode or FFFF in word mode
if N = 0
then (Rd) <-- 00 in byte mode or 0000 in word mode

Description: Copies the N flag (the sign bit of the last ALU operation) into the destination register. The destination register may be a byte or word register.

Example:
SEXT.b R1
if the result of the previous operation left the N flag set, then R1 <-- FF

Size: Byte, word

Flags Updated: none

Bytes: 2
Clocks: 3

Encoding:

1	0	0	1	SZ	0	0	0

d	d	d	d	1	0	0	1

Syntax: SUB dest, src

Operation: dest <- dest - src

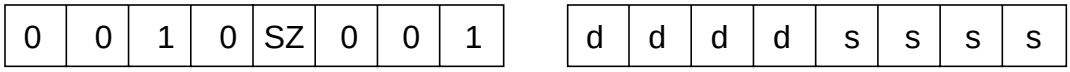
Description: Performs a twos complement binary subtraction of the source and destination operands, and the result is placed in the destination operand. The source data is not affected by the operation.

Size: Byte-Byte, Word-Word

Flags Updated: C, AC, V, N, Z

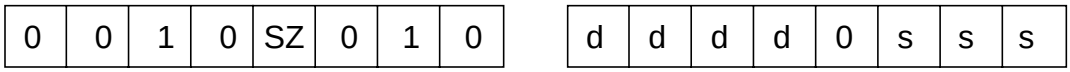
SUB Rd, Rs

Bytes: 2
Clocks: 3
Operation: (Rd) <-- (Rd) - (Rs)
Encoding:



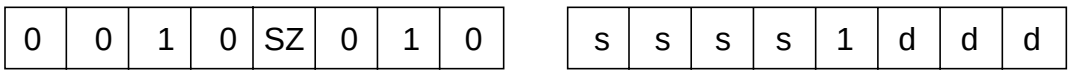
SUB Rd, [Rs]

Bytes: 2
Clocks: 4
Operation: (Rd) <-- (Rd) - ((WS:Rs))
Encoding:



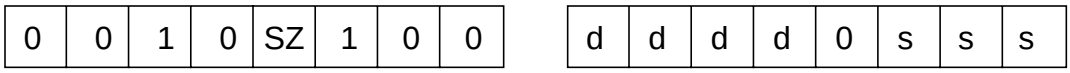
SUB [Rd], Rs

Bytes: 2
Clocks: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) - (Rs)
Encoding:



SUB Rd, [Rs+offset8]

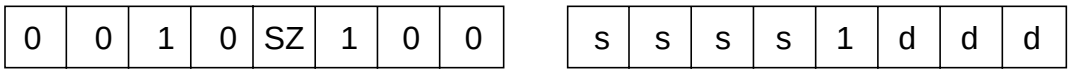
Bytes: 3
Clocks: 6
Operation: (Rd) <-- (Rd) - ((WS:Rs)+offset8)
Encoding:



byte 3: offset8

SUB [Rd+offset8], Rs

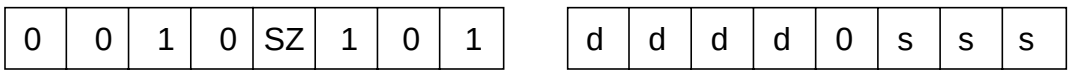
Bytes: 3
Clocks: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) - (Rs)
Encoding:



byte 3: offset8

SUB Rd, [Rs+offset16]

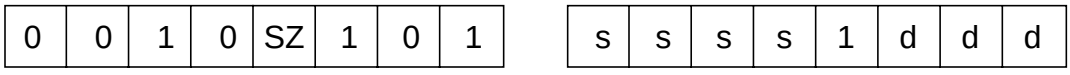
Bytes: 4
Clocks: 6
Operation: (Rd) <-- (Rd) - ((WS:Rs)+offset16)
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

SUB [Rd+offset16], Rs

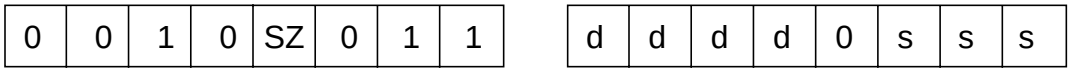
Bytes: 4
Clocks: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) - (Rs)
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

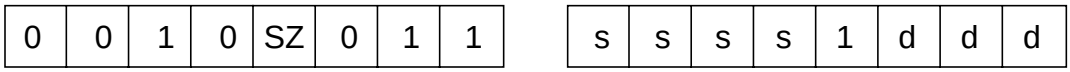
SUB Rd, [Rs+]

Bytes: 2
Clocks: 5
Operation: (Rd) <-- (Rd) - ((WS:Rs))
(Rs) <-- (Rs) + 1 (byte operation) or 2 (word operation)
Encoding:



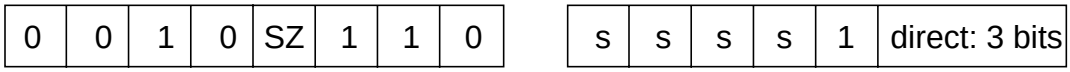
SUB [Rd+], Rs

Bytes: 2
Clocks: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) - (Rs)
(Rd) <-- (Rd) + 1 (byte operation) or 2 (word operation)
Encoding:



SUB direct, Rs

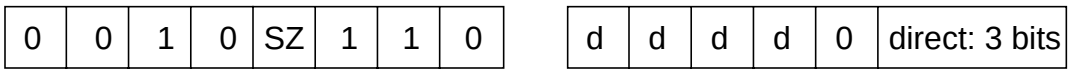
Bytes: 3
Clocks: 4
Operation: (direct) <-- (direct) - (Rs)
Encoding:



byte 3: lower 8 bits of direct

SUB Rd, direct

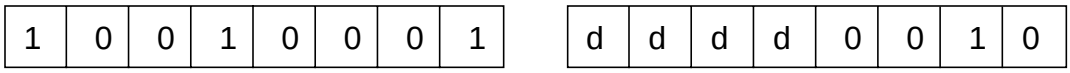
Bytes: 3
Clocks: 4
Operation: (Rd) <-- (Rd) - (direct)
Encoding:



byte 3: lower 8 bits of direct

SUB Rd, #data8

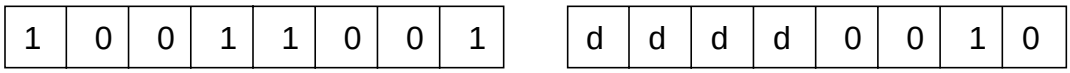
Bytes: 3
Clocks: 3
Operation: (Rd) <-- (Rd) - #data8
Encoding:



byte 3: #data8

SUB Rd, #data16

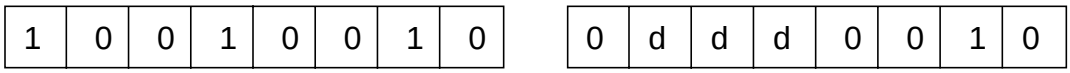
Bytes: 4
Clocks: 3
Operation: (Rd) <-- (Rd) - #data16
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

SUB [Rd], #data8

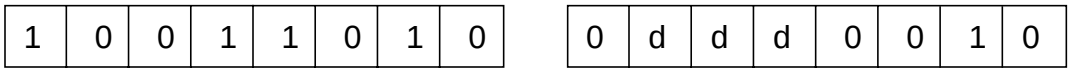
Bytes: 3
Clocks: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) - #data8
Encoding:



byte 3: #data8

SUB [Rd], #data16

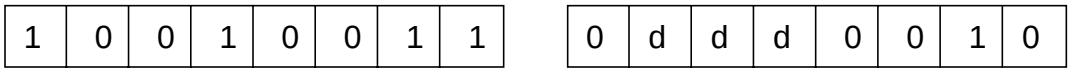
Bytes: 4
Clocks: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) - #data16
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

SUB [Rd+], #data8

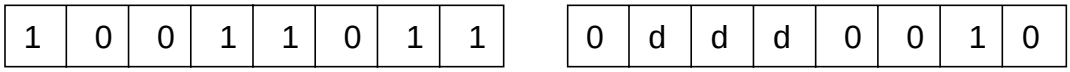
Bytes: 3
Clocks: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) - #data8
(Rd) <-- (Rd) + 1
Encoding:



byte 3: #data8

SUB [Rd+], #data16

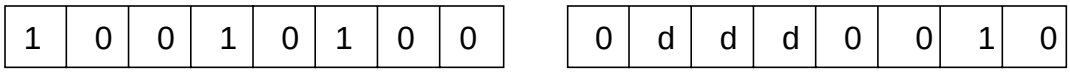
Bytes: 4
Clocks: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) - #data16
(Rd) <-- (Rd) + 2
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

SUB [Rd+offset8], #data8

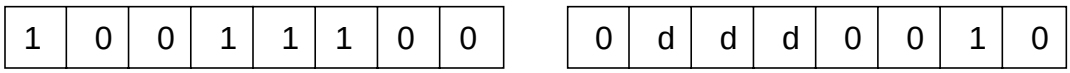
Bytes: 4
Clocks: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) - #data8
Encoding:



byte 3: offset8
byte 4: #data8

SUB [Rd+offset8], #data16

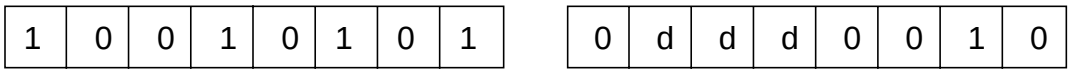
Bytes: 5
Clocks: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) - #data16
Encoding:



byte 3: offset8
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

SUB [Rd+offset16], #data8

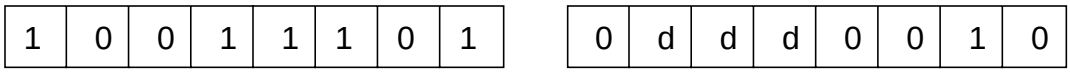
Bytes: 5
Clocks: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) - #data8
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: #data8

SUB [Rd+offset16], #data16

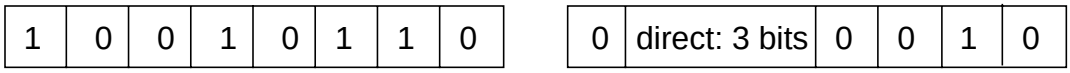
Bytes: 6
Clocks: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) - #data16
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: upper 8 bits of #data16
byte 6: lower 8 bits of #data16

SUB direct, #data8

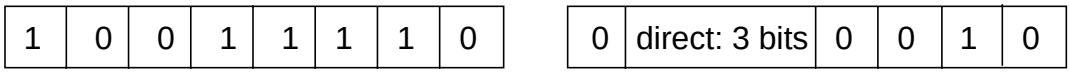
Bytes: 4
Clocks: 4
Operation: (direct) <-- (direct) - #data8
Encoding:



byte 3: lower 8 bits of direct
byte 4: #data8

SUB direct, #data16

Bytes: 5
Clocks: 4
Operation: (direct) <-- (direct) - #data16
Encoding:



byte 3: lower 8 bits of direct
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

Syntax: SUBB dest, src

Operation: dest <- dest - src - C

Description: Performs a twos complement binary addition of the source operand and the previously generated carry bit (borrow) with the destination operand. The result is stored in the destination operand.The source data is not affected by the operation.

If the carry from previous operation is zero (C = 0, i.e., Borrow = 1), the result is exact difference of the operands; if it is one (C = 1, i.e., Borrow = 0), the result is 1 less than the difference in operands.

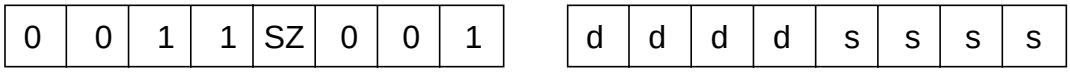
This form of subtraction is intended to support multiple-precision arithmetic. For this use, the carry bit is first reset, then SUBB is used to add the portions of the multiple-precision values from least-significant to most-significant.

Size: Byte-Byte, Word-Word

Flags Updated: C, AC, V, N, Z

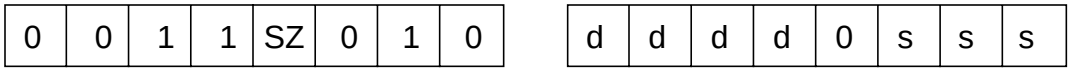
SUBB Rd, Rs

Bytes: 2
Clocks: 3
Operation: (Rd) <-- (Rd) - (Rs) - (C)
Encoding:



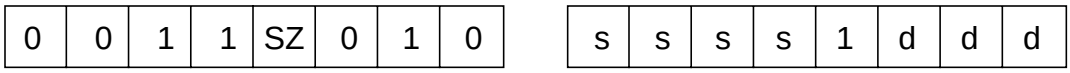
SUBB Rd, [Rs]

Bytes: 2
Clocks: 4
Operation: (Rd) <-- (Rd) - ((WS:Rs)) - (C)
Encoding:



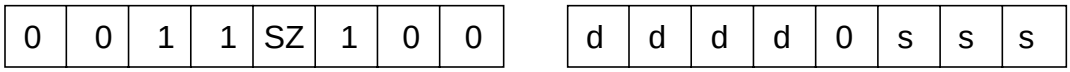
SUBB [Rd], Rs

Bytes: 2
Clocks: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) - (Rs) - (C)
Encoding:



SUBB Rd, [Rs+offset8]

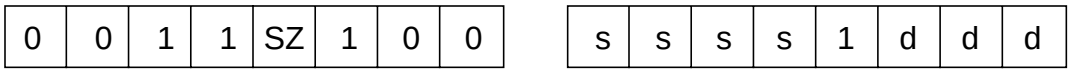
Bytes: 3
Clocks: 6
Operation: (Rd) <-- (Rd) - ((WS:Rs)+offset8) - (C)
Encoding:



byte 3: offset8

SUBB [Rd+offset8], Rs

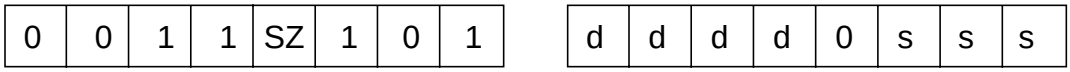
Bytes: 3
Clocks: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) - (Rs) - (C)
Encoding:



byte 3: offset8

SUBB Rd, [Rs+offset16]

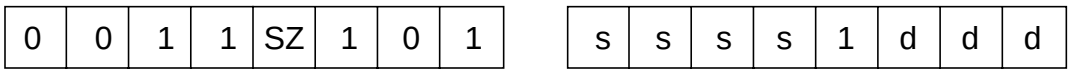
Bytes: 4
Clocks: 6
Operation: (Rd) <-- (Rd) - ((WS:Rs)+offset16) - (C)
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

SUBB [Rd+offset16], Rs

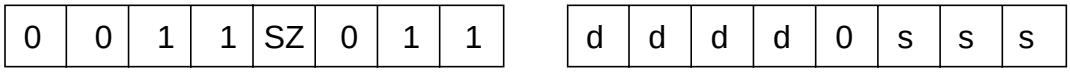
Bytes: 4
Clocks: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) - (Rs) - (C)
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

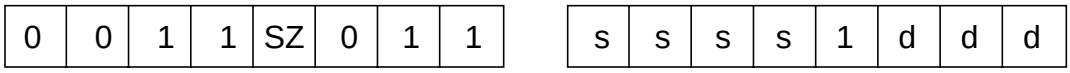
SUBB Rd, [Rs+]

Bytes: 2
Clocks: 5
Operation: (Rd) <-- (Rd) - ((WS:Rs)) - (C)
(Rs) <-- (Rs) + 1 (byte operation) or 2 (word operation)
Encoding:



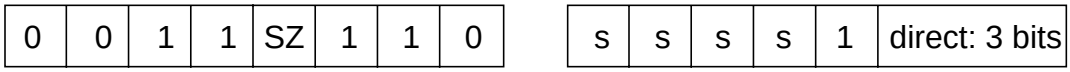
SUBB [Rd+], Rs

Bytes: 2
Clocks: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) - (Rs) - (C)
(Rd) <-- (Rd) + 1 (byte operation) or 2 (word operation)
Encoding:



SUBB direct, Rs

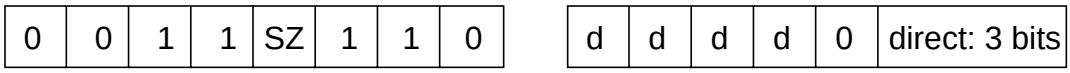
Bytes: 3
Clocks: 4
Operation: (direct) <-- (direct) - (Rs) - (C)
Encoding:



byte 3: lower 8 bits of direct

SUBB Rd, direct

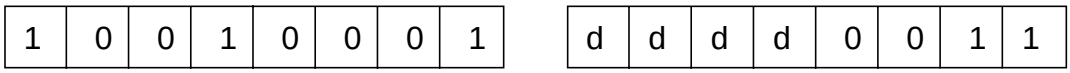
Bytes: 3
Clocks: 4
Operation: (Rd) <-- (Rd) - (direct) - (C)
Encoding:



byte 3: lower 8 bits of direct

SUBB Rd, #data8

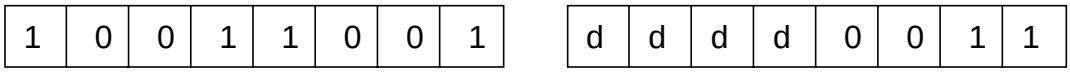
Bytes: 3
Clocks: 3
Operation: (Rd) <-- (Rd) - #data8 - (C)
Encoding:



byte 3: #data8

SUBB Rd, #data16

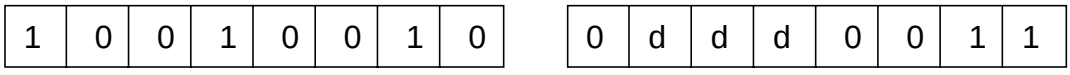
Bytes: 4
Clocks: 3
Operation: (Rd) <-- (Rd) - #data16 - (C)
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

SUBB [Rd], #data8

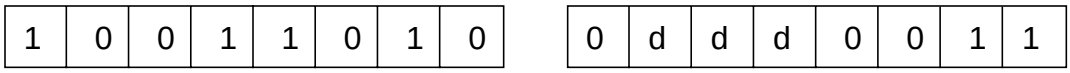
Bytes: 3
Clocks: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) - #data8 - (C)
Encoding:



byte 3: #data8

SUBB [Rd], #data16

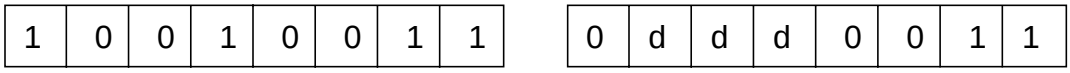
Bytes: 4
Clocks: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) - #data16 - (C)
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

SUBB [Rd+], #data8

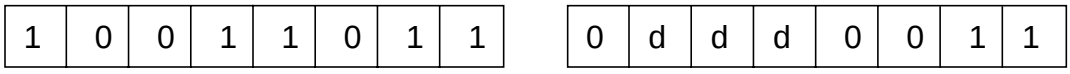
Bytes: 3
Clocks: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) - #data8 - (C)
(Rd) <-- (Rd) + 1
Encoding:



byte 3: #data8

SUBB [Rd+], #data16

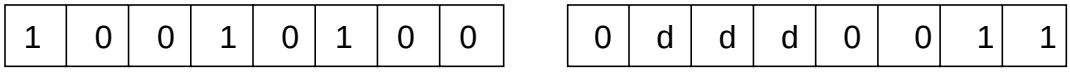
Bytes: 4
Clocks: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) - #data16 - (C)
(Rd) <-- (Rd) + 2
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

SUBB [Rd+offset8], #data8

Bytes: 4
Clocks: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) - #data8 - (C)
Encoding:



byte 3: offset8
byte 4: #data8

SUBB [Rd+offset8], #data16

Bytes: 5
Clocks: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) - #data16 - (C)
Encoding:

1	0	0	1	1	1	0	0		0	d	d	d	0	0	1	1

byte 3: offset8
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

SUBB [Rd+offset16], #data8

Bytes: 5
Clocks: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) - #data8 - (C)
Encoding:

1	0	0	1	0	1	0	1		0	d	d	d	0	0	1	1

byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: #data8

SUBB [Rd+offset16], #data16

Bytes: 6
Clocks: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) - #data16 - (C)
Encoding:

1	0	0	1	1	1	0	1		0	d	d	d	0	0	1	1

byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: upper 8 bits of #data16
byte 6: lower 8 bits of #data16

SUBB direct, #data8

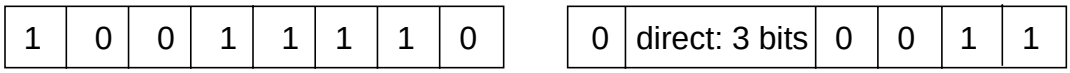
Bytes: 4
Clocks: 4
Operation: (direct) <-- (direct) - #data8 - (C)
Encoding:

1	0	0	1	0	1	1	0		0	direct: 3 bits			0	0	1	1

byte 3: lower 8 bits of direct
byte 4: #data8

SUBB direct, #data16

Bytes: 5
Clocks: 4
Operation: (direct) <-- (direct) - #data16 - (C)
Encoding:



byte 3: lower 8 bits of direct
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

Syntax: TRAP #data4

Operation: (PC) <-- (PC) + 2
(SSP) <-- (SSP) - 6
((SSP)) <-- (PC)
((SSP)) <-- (PSW)
(PSW) <-- code memory (trap vector (#data4))
(PC.15-0) <-- code memory (trap vector (#data4))
(PC.23-16) <-- 0; (PC.0) <-- 0

Description: Causes the specified software trap. The invoked routine is determined by branching to the specified vector table entry point. The RETI, return from interrupt, instruction is used to resume execution after the trap routine has been completed. A trap acts like an immediate interrupt, using a vector to call one of several pieces of code that will be executed in system mode. This may be used to obtain system services for application code, such as altering the data segment register. This is described in more detail in the section on interrupts and exceptions.

Note: The address of the exception handling routine must be word aligned as the PC is forced to an even address before vectoring to the service routine.

Size: None

Flags Updated: none

Bytes: 2
Clocks: 23/19 (PZ)

Encoding:

1	1	0	1	0	1	1	0

0	0	1	1	#data4

Syntax: XCH dest, src

Operation: dest <--> src

Description: The data specified by the source and destination operands is exchanged.

Size: Byte-Byte, word-word.

Flags Updated: none

XCH Rd, Rs

Bytes: 2
Clocks: 5
Operation: (Rd) <--> (Rs)
Encoding:

0	1	1	0	SZ	0	0	0	d	d	d	d	s	s	s	s

XCH Rd, [Rs]

Bytes: 2
Clocks: 6
Operation: (Rd) <--> ((WS:Rs))
Encoding:

0	1	0	1	SZ	0	0	0	d	d	d	d	0	s	s	s

XCH Rd, direct

Bytes: 3
Clocks: 6
Operation: (Rd) <--> (direct)
Encoding:

1	0	1	0	SZ	0	0	0	d	d	d	d	1	direct: 3 bits

byte 3: lower 8 bits of direct

Syntax: XOR dest, src

Operation: dest <- dest (XOR) src

Description: The byte or word specified by the source operand is bitwise logically XORed to the variable specified by the destination operand. The source data is not affected by the operation.

Size: Byte-Byte, Word-Word

Flags Updated: N, Z

XOR Rd, Rs

Bytes: 2
Clocks: 3
Operation: (Rd) <-- (Rd) (XOR) (Rs)
Encoding:

0	1	1	1	SZ	0	0	1	d	d	d	d	s	s	s	s

XOR Rd, [Rs]

Bytes: 2
Clocks: 4
Operation: (Rd) <-- (Rd) (XOR) ((WS:Rs))
Encoding:

0	1	1	1	SZ	0	1	0	d	d	d	d	0	s	s	s

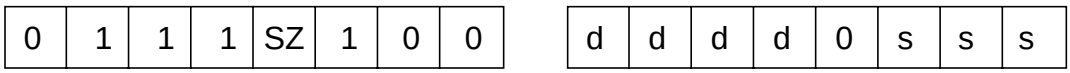
XOR [Rd], Rs

Bytes: 2
Clocks: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) (XOR) (Rs)
Encoding:

0	1	1	1	SZ	0	1	0	s	s	s	s	1	d	d	d

XOR Rd, [Rs+offset8]

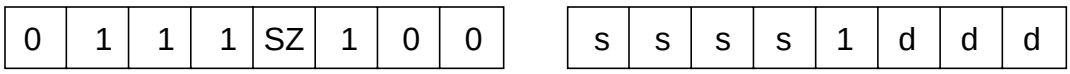
Bytes: 3
Clocks: 6
Operation: (Rd) <-- (Rd) (XOR) ((WS:Rs)+offset8)
Encoding:



byte 3: offset8

XOR [Rd+offset8], Rs

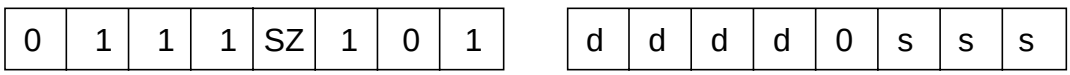
Bytes: 3
Clocks: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) (XOR) (Rs)
Encoding:



byte 3: offset8

XOR Rd, [Rs+offset16]

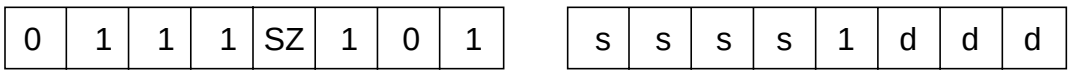
Bytes: 4
Clocks: 6
Operation: (Rd) <-- (Rd) (XOR) ((WS:Rs)+offset16)
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

XOR [Rd+offset16], Rs

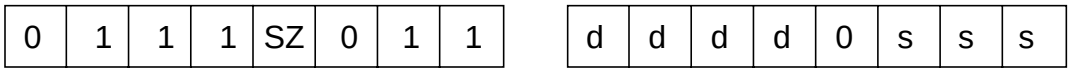
Bytes: 4
Clocks: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) (XOR) (Rs)
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16

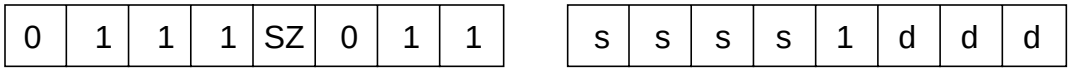
XOR Rd, [Rs+]

Bytes: 2
Clocks: 5
Operation: (Rd) <-- (Rd) (XOR) ((WS:Rs))
(Rs) <-- (Rs) + 1 (byte operation) or 2 (word operation)
Encoding:



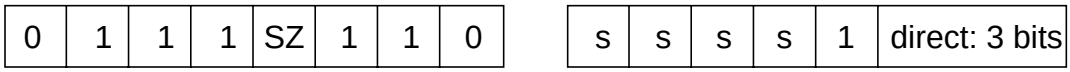
XOR [Rd+], Rs

Bytes: 2
Clocks: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) (XOR) (Rs)
(Rd) <-- (Rd) + 1 (byte operation) or 2 (word operation)
Encoding:



XOR direct, Rs

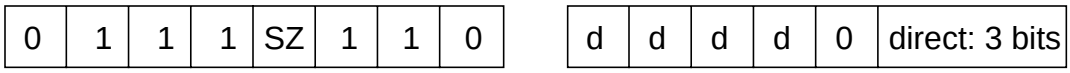
Bytes: 3
Clocks: 4
Operation: (direct) <-- (direct) (XOR) (Rs)
Encoding:



byte 3: lower 8 bits of direct

XOR Rd, direct

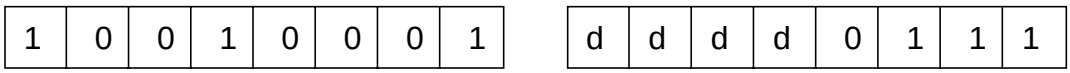
Bytes: 3
Clocks: 4
Operation: (Rd) <-- (Rd) (XOR) (direct)
Encoding:



byte 3: lower 8 bits of direct

XOR Rd, #data8

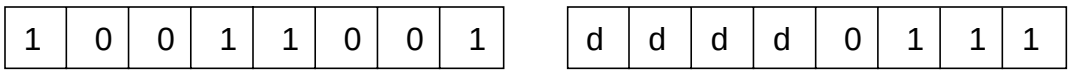
Bytes: 3
Clocks: 3
Operation: (Rd) <-- (Rd) (XOR) #data8
Encoding:



byte 3: #data8

XOR Rd, #data16

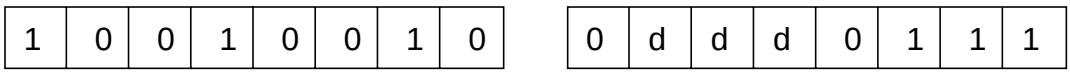
Bytes: 4
Clocks: 3
Operation: (Rd) <-- (Rd) (XOR) #data16
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

XOR [Rd], #data8

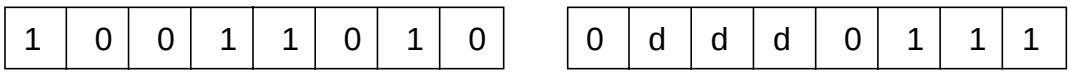
Bytes: 3
Clocks: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) (XOR) #data8
Encoding:



byte 3: #data8

XOR [Rd], #data16

Bytes: 4
Clocks: 4
Operation: ((WS:Rd)) <-- ((WS:Rd)) (XOR) #data16
Encoding:

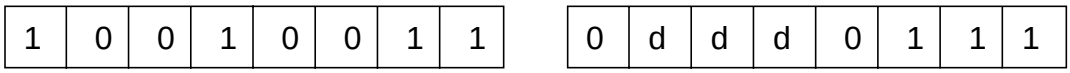


byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

XOR [Rd+], #data8

Bytes: 3
Clocks: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) (XOR) #data8
(Rd) <-- (Rd) + 1

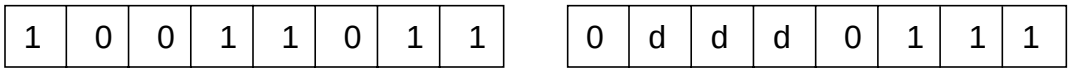
Encoding:



byte 3: #data8

XOR [Rd+], #data16

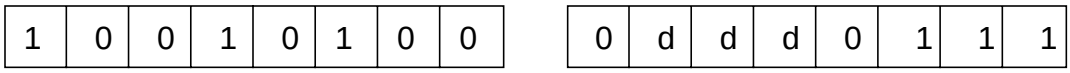
Bytes: 4
Clocks: 5
Operation: ((WS:Rd)) <-- ((WS:Rd)) (XOR) #data16
(Rd) <-- (Rd) + 2
Encoding:



byte 3: upper 8 bits of #data16
byte 4: lower 8 bits of #data16

XOR [Rd+offset8], #data8

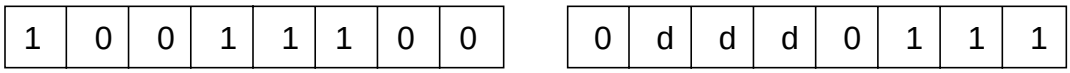
Bytes: 4
Clocks: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) (XOR) #data8
Encoding:



byte 3: offset8
byte 4: #data8

XOR [Rd+offset8], #data16

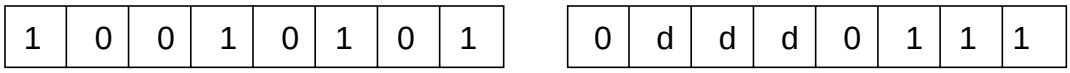
Bytes: 5
Clocks: 6
Operation: ((WS:Rd)+offset8) <-- ((WS:Rd)+offset8) (XOR) #data16
Encoding:



byte 3: offset8
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

XOR [Rd+offset16], #data8

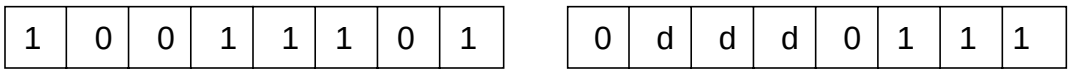
Bytes: 5
Clocks: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) (XOR) #data8
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: #data8

XOR [Rd+offset16], #data16

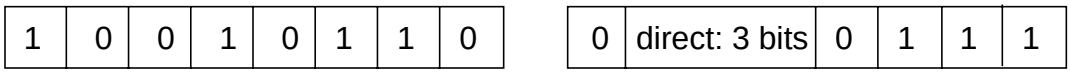
Bytes: 6
Clocks: 6
Operation: ((WS:Rd)+offset16) <-- ((WS:Rd)+offset16) (XOR) #data16
Encoding:



byte 3: upper 8 bits of offset16
byte 4: lower 8 bits of offset16
byte 5: upper 8 bits of #data16
byte 6: lower 8 bits of #data16

XOR direct, #data8

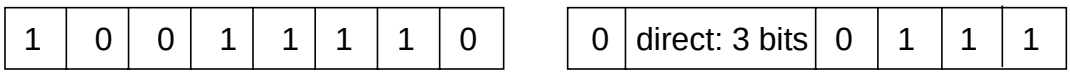
Bytes: 4
Clocks: 4
Operation: (direct) <-- (direct) (XOR) #data8
Encoding:



byte 3: lower 8 bits of direct
byte 4: #data8

XOR direct, #data16

Bytes: 5
Clocks: 4
Operation: (direct) <-- (direct) (XOR) #data16
Encoding:



byte 3: lower 8 bits of direct
byte 4: upper 8 bits of #data16
byte 5: lower 8 bits of #data16

6.6 Summary Of Illegal Operand Combinations On The XA

All but one case are instructions that specify or imply 2 write operations to a single register file location within a single instruction. The other case is a possible corruption of the source register data by an auto-increment before it is read. These conditions are not detected by XA hardware. The instruction/operand combinations indicated should not be used when writing XA code.

Instruction(s) affected	Reason for illegal combination
(any op) Rx, [Rx+]	Auto-increment plus explicit write [1]
mov [Rx+], [Rx+]	Double auto-increment of one register [2]
(any op) [Rx+], Rx	Auto-increment write may corrupt the source register before it is read [3]
NORM Rx, Rx	Result and shift count stored in the same register [4]
XCH Rx, Rx	Double write of a single register [4]
(any op) [Rx+], Ry	Auto-increment plus indirect write to same register [5]
(any op) [Rx+], [Ry+]	Auto-increment plus indirect write to same register [5]
(any op) [Rx+], #data	Auto-increment plus indirect write to same register [5]
XCH Rx, [Rx]	Indirect write plus explicit write to the same register [6]
XCH Rx, direct	Direct write plus explicit write to the same register [7]
POP R7	Stack pointer auto-increment plus explicit write to R7/SP [8]

NOTES:

1 This addressing mode is illegal when the source and destination are the same register. This would cause both a data write and an auto-increment operation to the same register.

2 This instruction is illegal when the source and destination pointer registers are the same register. This would cause two auto-increment operations to the same register.

3 This instruction is illegal when the source and destination are the same register. The source register would be auto-incremented and read at the same time, with an undefined result.

4 This instruction is illegal when the source and destination are the same register. This would cause two writes to the same register.

5 This addressing mode is illegal when the indirect address of the destination points to the pointer register itself in the register file. This is possible only when 8051 compatibility mode is enabled. This would cause both a data write and an auto-increment operation to the same register.

6 This instruction is illegal when the indirect address of the source operand points to the destination register itself in the register file. This is possible only when 8051 compatibility mode is enabled. This would cause two writes to the same register.

7 This instruction is illegal when the direct address of the source operand points to the destination register itself in the register file. This is possible only when 8051 compatibility mode is enabled. This would cause two writes to the same register.

8 A POP to R7 (the stack pointer) would cause both a data write and an auto-increment operation to the same register.

7 External Bus

Most XA derivatives have the capability of accessing external code and/or data memory through the use of an external bus. The external bus provides address information to external devices that are to be accessed, then generates a strobe for the required operation, with data passing in or out on the data bus. Typical bus operations are code read, data read, and data write. The standard XA external bus is designed to provide flexibility, simplicity of connection, and optimization for external code fetches.

The following discussion is based on the standard version of the XA external bus. Some specific XA derivatives may have a different implementation of the external bus, or no external bus at all.

7.1 External Bus Signals

For flexibility, the standard XA external bus supports 8 or 16-bit data transfers and a user selectable number of address bits. The maximum number of address lines varies by derivative but may be up to 24. A standard set of bus control signals coordinates activity on the bus. These are described in the following sections.

7.1.1 $\overline{\text{PSEN}}$ - Program Store Enable

The program store enable signal is used to activate an external code memory, such as an EPROM. This active low signal is typically connected to the Output Enable ($\overline{\text{OE}}$) pin of an external EPROM. $\overline{\text{PSEN}}$ remains high when a code read is not in progress.

7.1.2 $\overline{\text{RD}}$ - Read

The bus read signal is also active low. Activity of this signal indicates data read operations on the external bus. $\overline{\text{RD}}$ is typically connected to the pin of the same name on an external peripheral device.

7.1.3 $\overline{\text{WRL}}$ - Write Low Byte

$\overline{\text{WRL}}$ is the external bus data write strobe. It is typically connected to the $\overline{\text{WR}}$ pin of an external peripheral device. When the XA external bus is used in the 16-bit mode, this strobe applies only to the lower data byte, allowing byte writes on the 16-bit bus. The $\overline{\text{WRL}}$ signal is active low.

7.1.4 $\overline{\text{WRH}}$ - Write High Byte

For a 16-bit data bus, a signal similar to $\overline{\text{WRL}}$, but for the upper data byte is needed. The active low signal $\overline{\text{WRH}}$ serves this purpose.

7.1.5 ALE - Address Latch Enable

Since a portion of the XA external bus is used for multiplexed address and data information, that part of the address must be latched outside of the XA so that it will remain constant during the

subsequent read or write operation. The active high ALE signal directs the external latch to allow information to be stored for a data address or a code address. The external latch must close and retain this address when the ALE signal ends, by going low (inactive).

7.1.6 Address Lines

Some of the address lines used by the external bus interface are driven during a complete bus operation and do not need to be latched. In the standard XA bus interface, the lower four address lines are always driven and unlatched in this manner. This is done specifically as part of the optimization of the bus for fetching instructions from external code memory at high speed. This feature will be explained in detail in a later section.

7.1.7 Multiplexed Address and Data Lines

The part of the bus that is used for data transfer is also used for address output from the XA. Prior to asserting the strobe for the bus operation about to be performed, the XA outputs the address for the operation. On the multiplexed portion of the bus, this address is captured by an external latch, as commanded by the ALE signal. After that is done, this part of the bus is free to be used for data transfer either into or out of the XA. The control signals $\overline{\text{PSEN}}$, $\overline{\text{RD}}$, $\overline{\text{WRL}}$, and $\overline{\text{WRH}}$ determine what type of bus operation takes place.

7.1.8 WAIT - Wait

The WAIT input allows wait states to be inserted into any external bus operation. If WAIT is asserted (high) after a bus control strobe ($\overline{\text{PSEN}}$, $\overline{\text{RD}}$, $\overline{\text{WRL}}$, or $\overline{\text{WRH}}$) is driven by the XA, that bus operation is stretched, and that control strobe continues to be driven by the XA until WAIT goes low again. For this feature to be used, an external circuit must be present to generate the WAIT signal at the appropriate times.

The XA has an internal bus configuration feature that allows programming the various types of external bus cycles to different lengths, so that in most applications the WAIT line will not be needed. This feature will be explained in detail in a later section.

7.1.9 $\overline{\text{EA}}$ - External Access

The $\overline{\text{EA}}$ input determines whether the XA operates in single-chip mode, or begins running code from the internal program memory after reset. If $\overline{\text{EA}}$ is low as Reset goes high, the first code fetch (and all others after that) is made off-chip. If $\overline{\text{EA}}$ is high as Reset goes high, the XA will execute the on-chip code first, but will still attempt to execute instructions from external memory at addresses above the limit of on-chip code. The level on the $\overline{\text{EA}}$ pin is latched as reset goes high, so whatever mode is selected remains valid until the next reset.

On some XA derivatives, the pin used for the $\overline{\text{EA}}$ function may be shared with another function that becomes active after the XA begins code execution.

7.1.10 BUSW - Bus Width

The external XA bus may be configured to be 8 or 16 bits in width. The XA allows the bus width to be programmed in 2 ways. In a system where instructions are initially fetched from on-chip code memory, the user program can configure the external bus size (and many other aspects of the bus) prior to the bus actually being used.

When the initial code fetches must be done using off-chip code memory, however, the XA must know the bus width before the first off-chip code fetch can begin.

On some XA derivatives, the BUSW function may share a pin with some other function. In this case, the level on the BUSW pin is latched as Reset is released and that selection is kept until the next Reset. The secondary function on that pin will be active after Reset when the processor begins executing code normally.

Unlike the $\overline{EA}$ function, the bus width set by the BUSW pin at reset may be over-ridden by a user program, making setting by use of the BUSW pin unnecessary in most systems. Settings in the Bus Configuration Register allow changing the bus size under program control. This feature is covered in more detail in the next section.

7.2 Bus Configuration

The standard XA external bus has a number of configuration options. In addition to the data bus width selection discussed previously, the number of address lines used for external accesses is programmable, as is the bus timing.

7.2.1 8-Bit and 16-Bit Data Bus Widths

The standard XA external bus allows both 8-bit and 16-bit bus widths. BUSW=0 selects an 8-bit bus and BUSW=1 selects a 16-bit bus. On power-up, the XA defaults to the 16-bit bus (due to an on-chip weak pull-up on BUSW). The bus width is determined by the value of the BUSW pin as Reset is released, unless a user program overrides that setting by writing to the Bus Configuration Register (BCR), shown in Figure7.1.

BCR	-	-	-	WAITD	BUSD	BC2	BC1	BC0

WAITD: WAIT disable. Causes the XA external bus interface to ignore the value on the WAIT input. This allows tying the WAIT input high for applications that use internal code and do not need the WAIT function.

BUSD: Bus disable. Causes XA external bus functions to be disabled permanently. The primary purpose of this is to allow prevention of inadvertent activation of the bus by an instruction pre-fetch when the XA is executing code near the end of the on-chip code memory.

BC2 - BC0: These bits select the XA external bus configuration, specifically the number of data bits and the number of address lines. The supported combinations are shown below.

000 : 8-bit data bus, 12 address lines
001 : 8-bit data bus, 16 address lines
010 : 8-bit data bus, 20 address lines
011 : 8-bit data bus, 24 address lines
100 : < function reserved >
101 : < function reserved >
110 : 16-bit data bus, 20 address lines
111 : 16-bit data bus, 24 address lines

"-" Reserved for possible future use. Programs should take care when writing to registers with reserved bits that those bits are given the value 0. This will prevent accidental activation of any function those bits may acquire in future XA CPU implementations.

Figure 7.1 Bus Configuration Register (BCR)

Figures 7.2 and 7.3 show the address and data functions present on XA bus related pins when used with each available bus width.

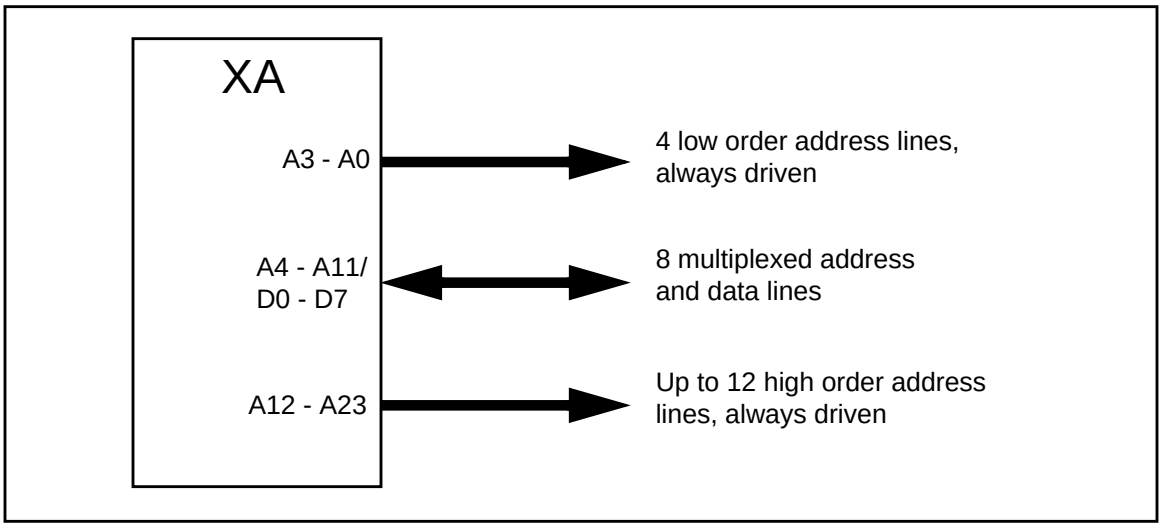


Figure 7.2 8-Bit External Bus Configuration

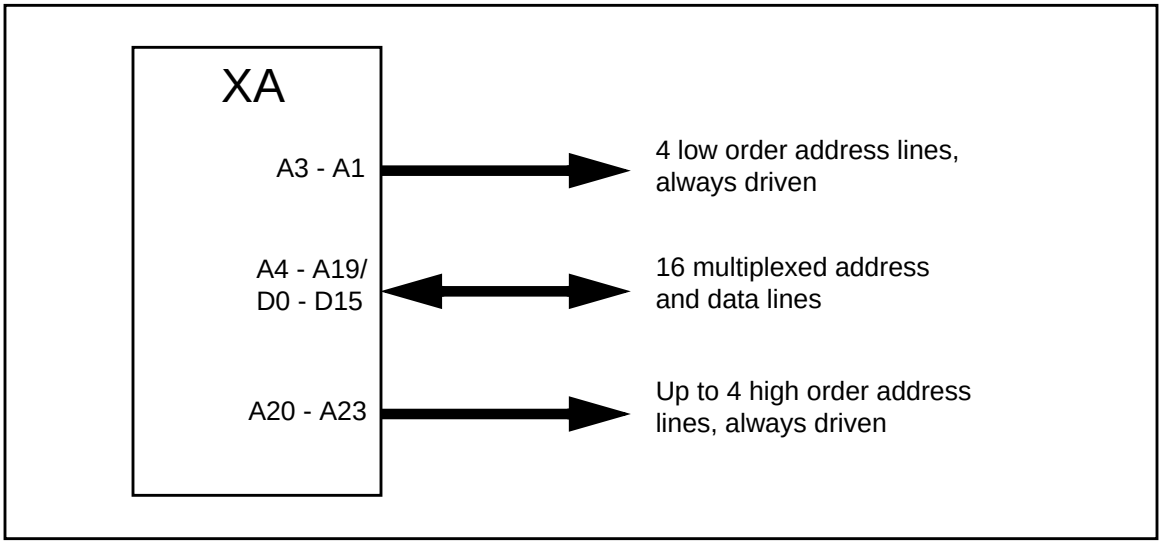


Figure 7.3 16-Bit External Bus Configuration

7.2.2 Typical External Device Connections

Many possibilities exist for connecting and using external devices with the XA bus. The bus will support EPROMs, RAMs, and other memory devices, as well as peripheral devices such as UARTs, and parallel port expanders. The following diagrams show a generalized connection of devices for 8-bit and 16-bit XA bus modes.

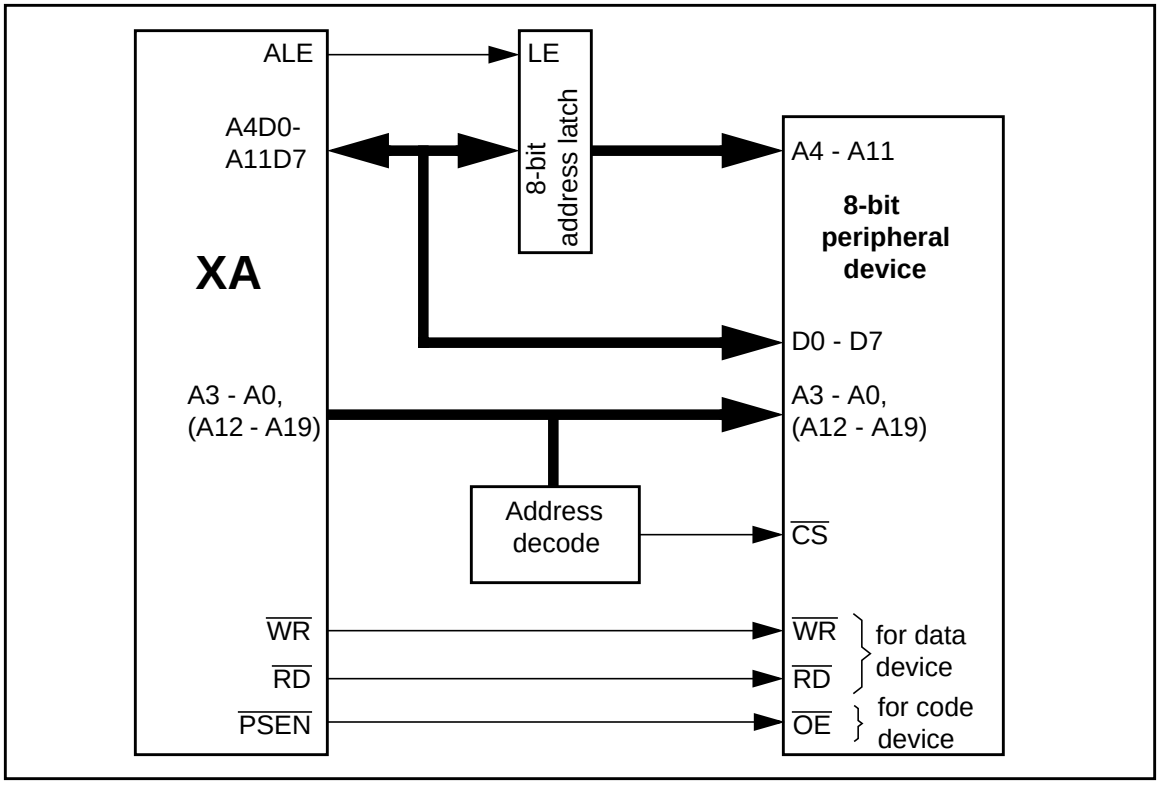


Figure 7.4 Typical XA External Bus Connections for 8-Bit Peripheral Devices

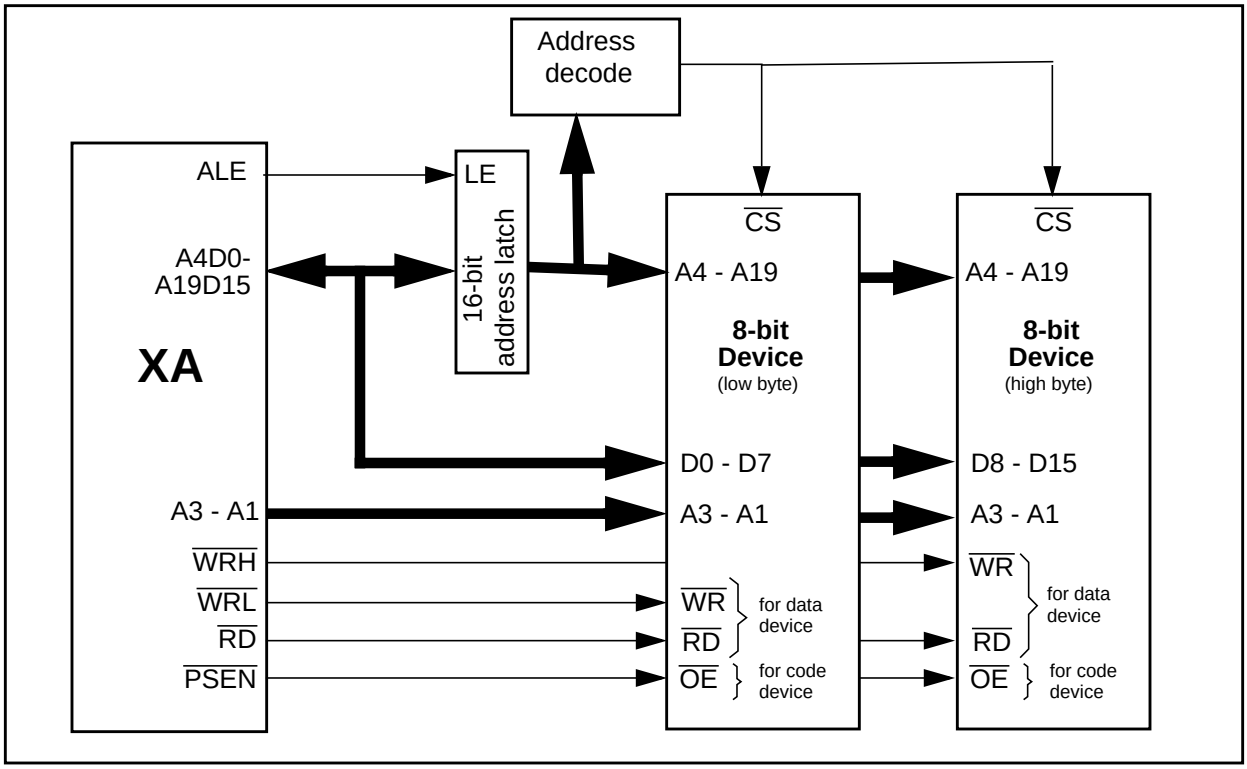


Figure 7.5 Typical XA External Bus Connections for 16-Bit Peripheral Devices

7.3 Bus Timing and Sequences

The standard XA external bus allows programming the widths of the bus control signals ALE, PSEN, WRL, WRH, and RD. There is also an option to extend the data hold time after a write operation. The combinations available will allow interfacing most devices to the XA directly without the need for special buffers or a WAIT state generator. Note that there is always a "rest clock" after any type of bus cycle except part of a burst mode code read. That is, when a bus cycle is completed and the bus strobe de-asserted, no new bus cycle will be begun until one clock has passed with no bus activity.

7.3.1 Code Memory

Interfacing with external code memory, typically in the form of EPROMs, is enabled by the $\overline{\text{PSEN}}$ control signal. If the XA is configured to execute internal code memory at reset, by the setting of the $\overline{\text{EA}}$ pin, it will automatically begin to fetch external code if the program crosses the boundary from internal to external code space. The location of this boundary varies for different XA derivatives, depending on the size of the internal code memory for each part.

Since the XA employs a pre-fetch queue in order to optimize instruction execution times, fetching of external instructions may begin before program execution actually crosses the on/off-chip code memory boundary. If a branch or subroutine return is located near the end of on-chip code memory, the off-chip fetch would be unnecessary, and may in fact cause problems if the XA ports that implement the external bus are being used for other purposes. For this reason, the BUSD (bus disable) bit in the Bus Configuration Register (BCR) is provided to prevent the XA from using the external bus for code or data operations.

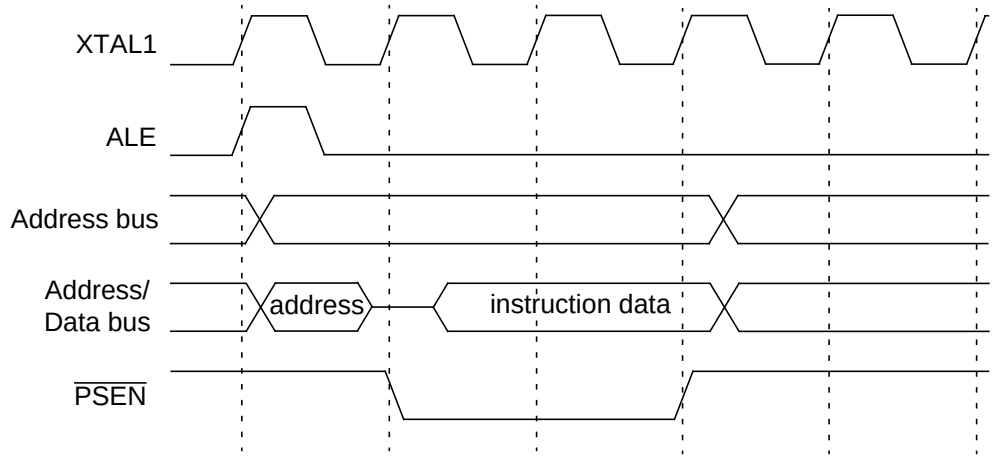
Note also that external code read cycles may sometimes be aborted by the XA. This happens when a code pre-fetch is occurring on the bus and the XA must execute a branch. The instruction data from the code pre-fetch will not be needed, so the bus cycle will be terminated immediately. This may appear as an ALE with no subsequent PSEN strobe, or a PSEN strobe that is shorter than that specified by the bus timing registers.

Code Read with ALE

The classic operation of a multiplexed address and data bus involves the issuance of an address, along with its associated control signal, for every bus cycle. The XA uses the bus control signal ALE to indicate that an address is on the bus that must be latched through the following code or data operation. The following diagram shows a code memory fetch in a cycle using ALE.

Burst Code Read (No ALE)

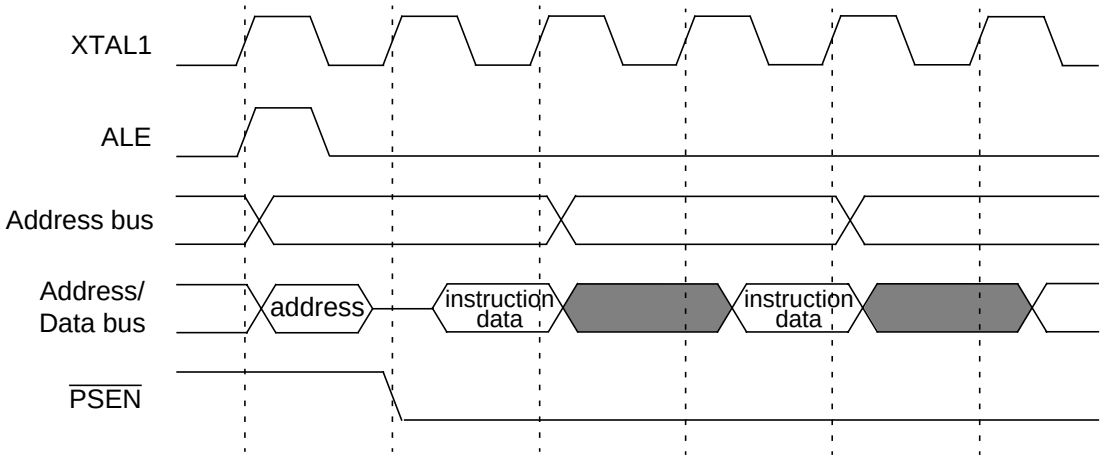
The XA does not always require an ALE cycle for every code fetch. This feature is included specifically to improve performance when the XA executes code from external memory, while increasing the access time available for the external memory device. Because the lower four address lines of the external bus are always driven, not multiplexed, the XA can access up to 16 bytes (or 8 words) of sequential code memory each time an ALE is issued. This type of fast sequential code read is called a burst read. Of course, any type of jump, branch, interrupt, or other change in sequential program flow will require an ALE in order to change the code fetch address in a non-sequential manner. Any data operation (read or write) on the XA external bus also requires an ALE cycle and will cause any subsequent external code fetch to begin with an ALE cycle also.



Note: the timing of this type of bus operation is user programmable. The timing shown here is generated by the Bus Timing Register setup: ALEW = 0, CRA1/0 = 01.

Figure 7.6 Typical External Code Read Using ALE

The following diagram shows a typical sequential code fetch where no ALE is issued between code reads. Also note that the $\overline{\text{PSEN}}$ bus control signal does not toggle, but remains asserted throughout the burst code read



Note: the timing of this type of bus operation is user programmable. The timing shown here is generated by the Bus Timing Register setup: ALEW = 0, CR1/0 = 01, CRA1/0 = 00.

Figure 7.7 Burst Mode (Sequential) External Code Read

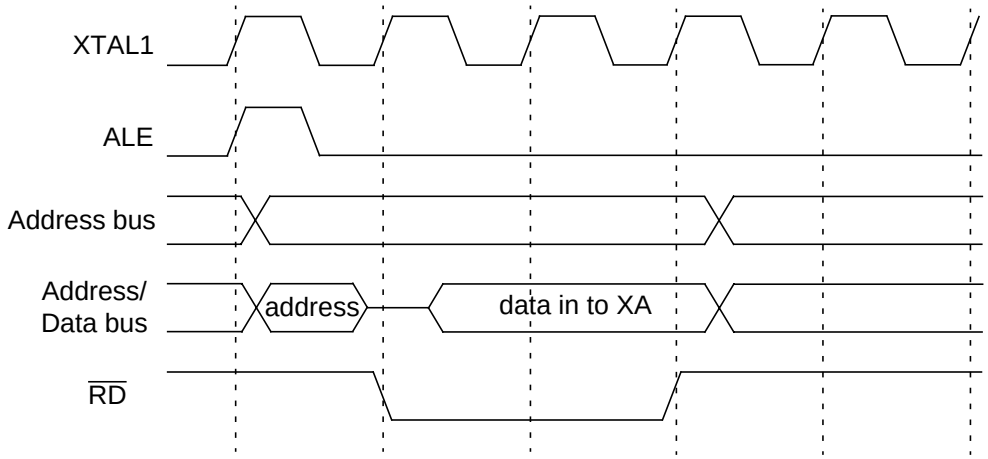
7.3.2 Data Memory

Reads and writes on the XA external bus are controlled through the use of the $\overline{RD}$, $\overline{WRL}$, and $\overline{WRH}$ signals. Since the XA bus supports both 8-bit and 16-bit widths, as well as byte and word read and write operations, several different versions of the basic bus cycles are possible. These are described in the following sections.

Data memory, like code memory, has a boundary where the internal data memory ends, and above which the XA will switch to the external bus in order to act on data memory. This on/off-chip data memory boundary may be in a different place for various XA derivatives, depending upon the amount of internal data memory built into a specific derivative.

Typical Data Read

A simple byte read on an 8-bit bus or any read on a 16-bit bus both begin with an ALE cycle, where the XA presents the address of the data location that is to be read on the bus. This is followed by the assertion of the $\overline{RD}$ strobe, that causes the external device to present its data on the bus. This process is shown in the diagram below.

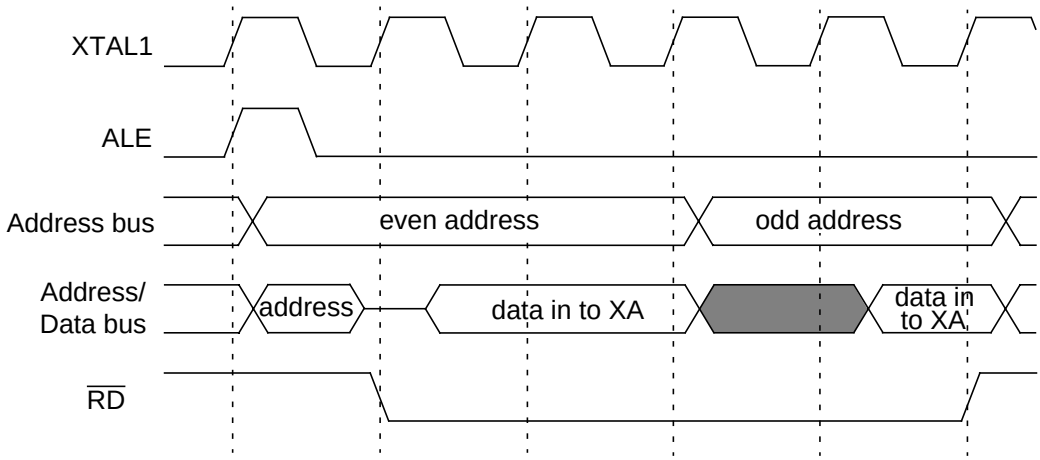


Note: the timing of this type of bus operation is user programmable. The timing shown here is generated by the Bus Timing Register setup: ALEW = 0, DRA1/0 = 01.

Figure 7.8 Typical External Data Read

Word Read on an 8-Bit Data Bus

When the XA external bus is configured for an 8-bit data width, a word read operation is automatically performed as two byte reads at sequential addresses. Since the XA CPU requires word operations to be performed at even addresses, the second half of any word read on a byte-wide bus always uses the same upper address latched by ALE. for this operation, the low order byte first is read at the even byte address, then the high order byte is read at the next (odd) address. So, only one ALE is required in this case. The diagram below shows this sequence.



Note: the timing of this type of bus operation is user programmable. The timing shown here is generated by the Bus Timing Register setup: ALEW = 0, DRA1/0 = 01, DR1/0 = 01.

Figure 7.9 Word Read on 8-Bit Data Bus

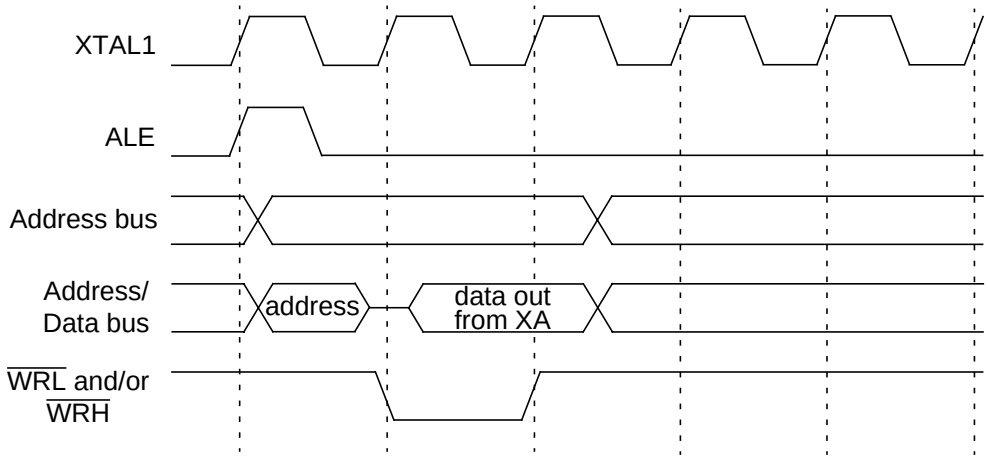
Byte Read on a 16-Bit Data Bus

When an instruction causes a read of one byte of data from the external bus, when it is configured for 16-bit width, a simple read operation is performed. This results in 16 bits of data being received by the XA, which uses only the byte that was requested by the program. There is no way to distinguish a byte read from a word read on the external bus when it is configured for a 16-bit width.

Typical Data Write

A data write operation begins with an ALE cycle, like a read operation, followed by the assertion of one or both of the write strobes, $\overline{WRL}$ and $\overline{WRH}$. This simple bus cycle applies to byte writes on an 8-bit data bus and all writes on a 16-bit data bus.

A byte write on an 8-bit data bus will always use only the $\overline{WRL}$ strobe. A byte write on a 16-bit data bus will always use either the $\overline{WRL}$ or $\overline{WRH}$ strobe, depending on whether the byte is at an even or odd address. A word write on a 16-bit bus requires the assertion of both the $\overline{WRL}$ and $\overline{WRH}$ strobes. The simple data write cycle is shown below.

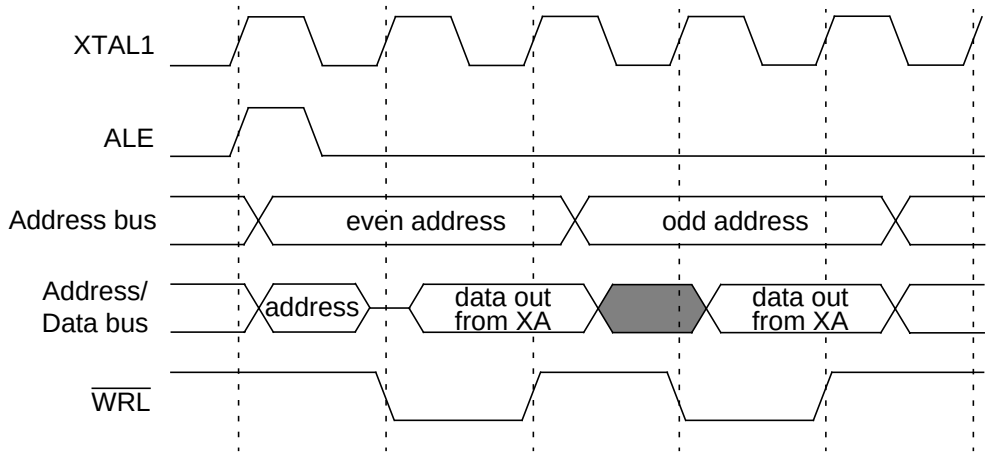


Note: the timing of this type of bus operation is user programmable. The timing shown here is generated by the Bus Timing Register setup: ALEW = 0, DWA1/0 = 00, WM0 = 0, WM1 = 0.

Figure 7.10 Typical External Data Write

Word Write on an 8-Bit Data Bus

When a word write operation is done with the bus configured to an 8-bit width, the XA automatically performs two byte writes. First, the low order byte is written (at the even byte address), then the high order byte is written at the next (odd) address. As with a word read on an 8-bit bus, this requires only a single ALE cycle at the beginning of the process. This sequence is shown in the following diagram.



Note: the timing of this type of bus operation is user programmable. The timing shown here is generated by the Bus Timing Register setup: ALEW = 0, DWA1/0 = 00, DW1/0 = 00, WM0 = 0, WM1 = 0.

Figure 7.11 Word Write on 8-Bit Data Bus

External Bus Signal Timing Configuration

The standard XA bus also provides a high degree of bus timing configurability. There are separate controls for ALE width, data read and write cycle lengths, and data hold time. These times are programmable in a range that will support most RAMs, ROMs, EPROMs, and peripheral devices over a wide range of oscillator frequencies without the need for additional external latches, buffers, or WAIT state generators.

Programmable bus timing is controlled by settings found in the Bus Timing Register SFRs, named BTRH, and BTRL, shown in Figures 7.12 and 7.13.

BTRH	DW1	DW0	DWA1	DWA0	DR1	DR0	DRA1	DRA0

DW1, DW0: Data Write without ALE. Applies only to the second half of a 16-bit write operation when the bus is configured to 8 bits.
00 : Data write cycle is 2 clock in duration.
01 : Data write cycle is 3 clocks in duration.
10 : Data write cycle is 4 clocks in duration.
11 : Data write cycle is 5 clocks in duration.

DWA1, DWA0: Data Write with ALE. Selects the length (in CPU clocks) of the entire data write cycle, including ALE.
00 : Data write cycle is 2 clocks in duration.
01 : Data write cycle is 3 clocks in duration.
10 : Data write cycle is 4 clocks in duration.
11 : Data write cycle is 5 clocks in duration.

DR1, DR0: Data Read without ALE. Applies only to the second half of a 16-bit read operation when the bus is configured to 8 bits.
00 : Data read cycle is 1 clock in duration.
01 : Data read cycle is 2 clocks in duration.
10 : Data read cycle is 3 clocks in duration.
11 : Data read cycle is 4 clocks in duration.

DRA1, DRA0: Data Read with ALE. Selects the length (in CPU clocks) of the entire data read cycle, including ALE.
00 : Data read cycle is 2 clocks in duration.
01 : Data read cycle is 3 clocks in duration.
10 : Data read cycle is 4 clocks in duration.
11 : Data read cycle is 5 clocks in duration.

Notes:
- See text regarding disallowed bus timing combinations.
- The bit pairs DW1:0, DWA1:0, DR1:0, DRA1:0, CR1:0, and CRA1:0 determine the length of entire bus cycles of different types. Bus cycles with an ALE begin when ALE is asserted. Bus cycles without an ALE begin when the bus strobe is asserted or when the address changes (in the case of burst mode code reads). Bus cycles end either when the bus strobe is de-asserted or when data hold time is completed (in the case of a data write with extra hold time, see bit WM0).

Figure 7.12 Bus Timing Register High Byte (BTRH)

BTRL	WM1	WM0	ALEW	-	CR1	CR0	CRA1	CRA0

WM1: Write Mode 1. Selects the width of the write pulse.
0 : Write pulse (WR) width is 1 CPU clock.
1 : Write pulse (WR) width is 2 CPU clocks.

WM0: Write Mode 0. Selects the data hold time.
0 : Data hold time is minimum (0 clocks).
1 : Data hold time is 1 CPU clock.

ALEW: ALE width selection. Determines the duration of ALE pulses.
0 : ALE width is one half of one CPU clock.
1 : ALE width is one and a half CPU clocks.

CR1, CR0: Code Read. Selects the length of a code read cycle when ALE is not used.
00 : Code read cycle is 1 clocks in duration.
01 : Code read cycle is 2 clocks in duration.
10 : Code read cycle is 3 clocks in duration.
11 : Code read cycle is 4 clocks in duration.

CRA1, CRA0: Code Read with ALE. Selects the length of a code read cycle when ALE is used prior to $\overline{\text{PSEN}}$ being asserted.
00 : Code read cycle is 2 clocks in duration.
01 : Code read cycle is 3 clocks in duration.
10 : Code read cycle is 4 clocks in duration.
11 : Code read cycle is 5 clocks in duration.

"-" Reserved for possible future use. Programs should take care when writing to registers with reserved bits that those bits are given the value 0. This will prevent accidental activation of any function those bits may acquire in future XA CPU implementations.

Notes:
- See text regarding disallowed bus timing combinations.
- The bit pairs DW1:0, DWA1:0, DR1:0, DRA1:0, CR1:0, and CRA1:0 determine the length of entire bus cycles of different types. Bus cycles with an ALE begin when ALE is asserted. Bus cycles without an ALE begin when the bus strobe is asserted or when the address changes (in the case of burst mode code reads). Bus cycles end either when the bus strobe is de-asserted or when data hold time is completed (in the case of a data write with extra hold time, see bit WM0).

Figure 7.13 Bus Timing Register Low Byte (BTRL)

Disallowed Bus Timing Configurations

Some possible combinations of bus timing register settings do not make sense and the XA cannot produce working bus signals that match those settings. The disallowed combinations occur where the sum of the specified components of a bus cycle exceed the specified length of the entire cycle. Two simple rules define the allowed/disallowed combinations. Violating these rules may result in incomplete bus cycles, for example a data read cycle in which an address and ALE pulse are output, but no read strobe ($\overline{RD}$) is produced.

For data write cycles on the external bus there are two conditions that must be met. The first applies to data write cycles with no ALE:

$$WM1 + WM0 \leq DW1:0$$

This says that the sum of the timing values defined by the WM1 and WM0 fields must be less than or equal to the timing value defined by the DW field. Note that this is the value of the timing durations that they specify. For example, if the WM1 field specifies a 2 clock write pulse and the WM0 field specifies a 1 clock data hold time, those two times together (3 clocks) must be less than or equal to the timing specified by the DW1:0 field. In this case the DW1:0 field must specify a total bus cycle duration of at least 3 clocks. The other rule uses the same structure, as follows.

A second requirement applies to write cycles with ALE:

$$ALEW + WM1 + WM0 \leq DWA1:0$$

The configuration for data read has only one requirement, which applies to data read cycles with ALE:

$$ALEW + 1 \leq DRA1:0$$

The configuration for code read also has only one requirement, which applies to code read cycles with ALE:

$$ALEW + 1 \leq CRA1:0$$

7.3.3 Reset Configuration

Upon reset, at the time of power up or later, the XA bus is initially configured in certain ways. As previously discussed, the pins $\overline{EA}$ and BUSW select whether the XA will begin operation from internal code, and whether the bus will be 8-bits or 16-bits.

The values for the programmable bus timing are also set to a default value at reset. All of the timing values are set to their maximum, providing the slowest bus cycles. This setting allows for the slowest external devices that may be sued with the XA without WAIT generation logic. The user program should set the bus timing to the correct values for the specific application in the system initialization code. Refer to the data sheet for a particular XA derivative for details of the values found in registers and SFRs after reset.

7.4 Ports

I/O ports on any microcontroller provide a connection to the outside world. The capabilities of those I/O ports determine how easily the microcontroller can be interfaced to the various external devices that make up a complete application. The standard XA I/O ports provide a high degree of versatility through the use of programmable output modes and allow easy connection to a wide variety of hardware.

7.4.1 I/O Port Access

The standard on-chip I/O ports of the XA are accessed as SFRs. The SFR names used for these ports begin with port 0, called P0. Port numbers and names go up in sequence from there, to the number of ports on a specific XA derivative. Ports are normally identified by their names in assembler source code, such as: "MOV P1,#0". This instruction causes the value 0 to be written to port 1.

XA I/O ports are typically bit addressable, meaning that individual port bits are readable, writable, and testable. An instruction using a port bit looks like this: "SETB P2.1". This particular example would result in the second lowest bit in port 2 (bit 1) having a 1 written to it.

Reading of a Port Pin Versus the Port Latch

Each I/O port has two important logic values associated with it. The first is the contents of the port latch. When data is written to a port, it is stored in the port latch. The second value is the logic level of the actual port pin, which may be different than the port latch value, especially if a port pin is being used as an input.

When a port is explicitly read by an instruction, the value returned is that from the pin. When a port is read intrinsically, in order to perform some operation and store the value back to the port, the port latch is read. This type of operation is called a read-modify-write.

1) The following instructions cause read-modify-write operations, and read the port latch when a port or port bit is specified as the destination:

```
ADD     Px, ...
ADDC    Px, ...
ADDS    Px, ...
AND     Px, ...
DJNZ    Px, ...
OR      Px, ...
SUB     Px, ...
SUBB    Px, ...
XOR     Px, ...

CLR     Px.y
JBC     Px.y, rel8
MOV     Px.y, C
SETB    Px.y
```

2) The following instruction reads the port pins when a port is specified as the destination operand:

```
CMP     Px, ...
```

3) When a port or port bit is specified as a source in any instruction, the port pin is always read.

Figure 7.14 How ports are read.

7.4.2 Port Output Configurations

Standard XA I/O ports provide several different output configurations. One is the 80C51 type quasi-bidirectional port output. Others are open drain, push-pull, and high impedance (input only). It is important to note that the port configuration applies to a pin even if that pin is part of the external bus. Bus pins should normally be configured to push-pull mode. Also, the port latches for pins that are to be used as part of the external bus must be set to one (which is the reset state). A zero in a port latch will override bus operations and force a zero on the corresponding bus position.

The port configuration is controlled by settings in two SFRs for each port. One bit in each port configuration register is associated with a port pin in the corresponding bit position. These port configuration SFRs are called: PnCFGA and PnCFGB, where "n" is the port number. So, the configuration registers for port 1 are named P1CFGA and P1CFGB. The table below shows the port control bit combinations and the associated port output modes.

Table 7.1

PnCFGB	PnCFGA	Port Output Mode
0	0	Open drain.
0	1	Quasi-bidirectional (default).
1	0	High impedance.
1	1	Push-pull.

7.4.3 Quasi-Bidirectional Output

The default port output configuration for standard XA I/O ports is the quasi-bidirectional output that is common on the 80C51 and most of its derivatives. This output type can be used as both an input and output without the need to reconfigure the port. This is possible because when the port outputs a logic high, it is weakly driven, allowing an external device to pull the pin low. When the pin is pulled low, it is driven strongly and able to sink a fairly large current. These features are somewhat similar to an open drain output except that there are three pullup transistors in the quasi-bidirectional output that serve different purposes.

One of these pullups, called the "very weak" pullup, is turned on whenever the port latch for a particular pin contains a logic 1. The very weak pullup sources a very small current that will pull the pin high if it is left floating.

A second pullup, called the "weak" pullup, is turned on when the port latch for its associated pin contains a logic 1 and the pin itself is a logic 1. This pullup provides the primary source current for a pin that is outputting a 1, and can drive several TTL loads. If a pin that has a logic 1 on it is pulled low by an external device, the weak pullup turns off, and only the very weak pullup remains on. In order to pull the pin low under these conditions, the external device has to sink enough current to overpower the weak pullup and pull the voltage on the port pin below its input threshold.

The third (and final) pullup is referred to as the "strong" pullup. This pullup is included to speed up low-to-high transitions on a port pin when the port latch changes from 0 to 1. When this occurs, the strong pullup turns on for a brief time, two CPU clocks, pulling the port pin high quickly, then turning off again.

The quasi-bidirectional output structure normally provides a means to have mixed inputs and outputs on port pins without the need for special configurations. However, it has several drawbacks that can be problems in certain situations. For one thing, quasi-bidirectional outputs have a very small source current and are therefore not well suited to driving certain types of loads. They are especially unsuited to directly drive the bases of external NPN transistors, a common method of boosting the current of I/O pins.

Also, since the weak pullup turns off when a port pin is actually low, and the strong pullup turns on only for a brief time, it is possible that under certain port loading conditions, the port pin will get "stuck" low and cannot be driven high. This tends to happen when an external device being driven by the port pin has some leakage to ground that is larger than the current supplied by the very weak pullup of the quasi-bidirectional port output. If there is also a fairly large capacitance on the pin, from a combination of the wiring itself and the pin capacitance of the device(s) connected to the pin, the strong pullup may not succeed in pulling the pin high enough while it is turned on. When the strong pullup is then turned off, the leakage of the external device pulls the pin low again, since only the very weak pullup is turned on at that point and the leakage is greater than the very weak pullup source current. These issues are the reason for enhancing the port configurations of the XA.

A diagram of the quasi-bidirectional output structure is shown in the figure below.

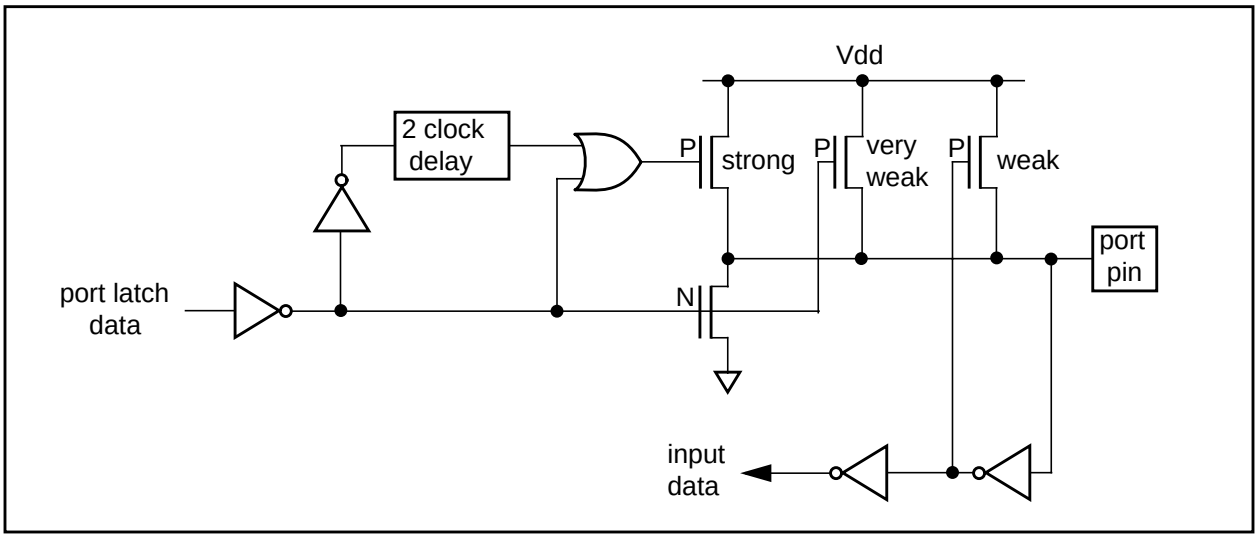


Figure 7.15 Structure of the Quasi-Bidirectional Output Configuration

Open Drain Output

Another port output configuration provided by the standard XA I/O ports is open drain. This configuration turns off all pullups and only drives the pulldown transistor of the port driver when the port latch contains a logic 0. To be used as a logic output, a port configured in this manner must have an external pullup, typically a resistor tied to Vdd. The pulldown for this mode is the same as for the quasi-bidirectional mode.

An advantage of the open drain output is that is may be used to create wired AND logic. Several open drain outputs of various devices can be tied together, and any one of them can drive the wire low, creating a logical AND function without using a logic gate. The figure below show the structure of the open drain output.

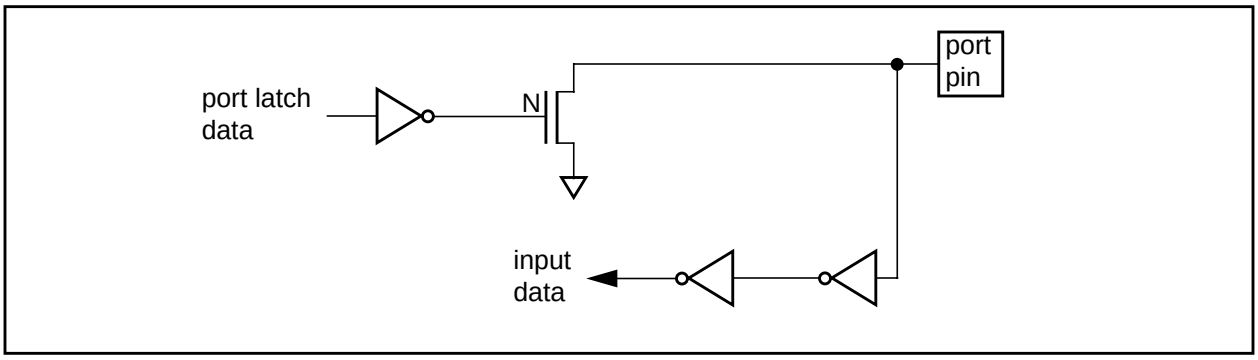


Figure 7.16 Structure of the Open Drain Output Configuration

Push-Pull Output

The push-pull output mode has the same pulldown structure as both the open drain and the quasi-bidirectional output modes, but provides a continuous strong pullup when the port latch contains a logic 1. This mode uses the same pullup as the strong pullup for the quasi-bidirectional mode. The push-pull mode may be used when more source current is needed from a port output. The output structure for this mode is shown below.

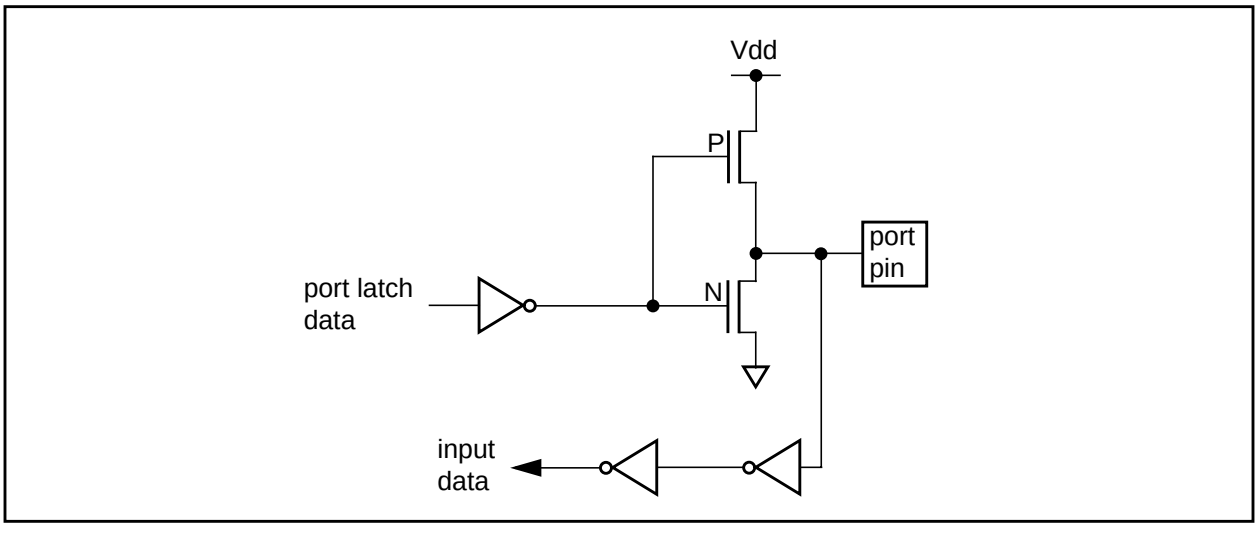


Figure 7.17 Structure of the Push-Pull Output Configuration

High Impedance Output

The final XA port output configuration is called high impedance mode. This mode simply turns all output drivers on a port pin off. Thus, the pin will not source or sink current and may be used effectively as an input-only pin with no internal drivers for an external device to overcome.

7.4.4 Reset State and Initialization

Upon chip reset, all of the port output configurations are set to quasi-bidirectional, and the port latches are written with all ones. The quasi-bidirectional output type is a good default at power-up or reset because it does not source a large amount of current if it is driven by an external device, yet it does not allow the port pin to float. A floating input pin on a CMOS device can cause excess current to flow in the pin's input circuitry, and of course all port pins have input circuits in addition to outputs.

7.4.5 Sharing of I/O Ports with On-Chip Peripherals

Since XA on-chip peripheral devices share device pins with port functions, some care must be taken not to accidentally disable a desired pin function by inadvertently activating another function on the same pin. A peripheral that has an output on a pin will use the I/O port output configuration for that pin (quasi-bidirectional, open drain, push-pull, or high impedance).

The method of sharing multiple functions on a single pin involves a logic AND of all of the functions on a pin. So, if a port latch contains a zero, it will drive that port pin low, and any peripheral output function on that pin is overridden. Conversely, an on-chip peripheral outputting a zero on a pin prevents the contents of the port latch from controlling the output level. It is usually not an issue to avoid turning on an alternate peripheral function on a pin accidentally, since most peripherals must be either explicitly turned on or activated by a write to one of their SFRs. It is more likely that a user program could erroneously write a zero to a port latch bit corresponding to a pin with a peripheral function that is being used and therefore disable that function. The simple rule to follow is: never write a zero to a port bit that is associated with an active on-chip peripheral, or that should only be used an input.

When an XA I/O port pin is used as an input for a peripheral function, it is sampled at the oscillator rate divided by 2. For example, if an XA is running at a 20 MHz clock (giving a 50 ns clock period), an external timer input would have to remain in the same state for at least 100 ns in order to guarantee that it is sampled correctly. This gives a maximum frequency for such inputs as the oscillator rate divided by 4. In this example, the maximum external timer input rate would be 5 MHz.

8 Special Function Register Bus

The Special Function Register Bus or SFR Bus is the means by which all Special Function Registers are connected to the XA CPU so that they may be read and written by user programs. This includes all of the registers contained in peripherals such as Timers and UARTs, as well as some CPU registers such as the PSW. CPU registers communicate functionally with the CPU via direct connections, but read and write operations performed on them are routed through the SFR bus.

The SFR bus provides a common interface for the addition of any new functions to the XA core, thus supplying the means for building a large and varied microcontroller derivative family. This is illustrated in Figure 8.1.

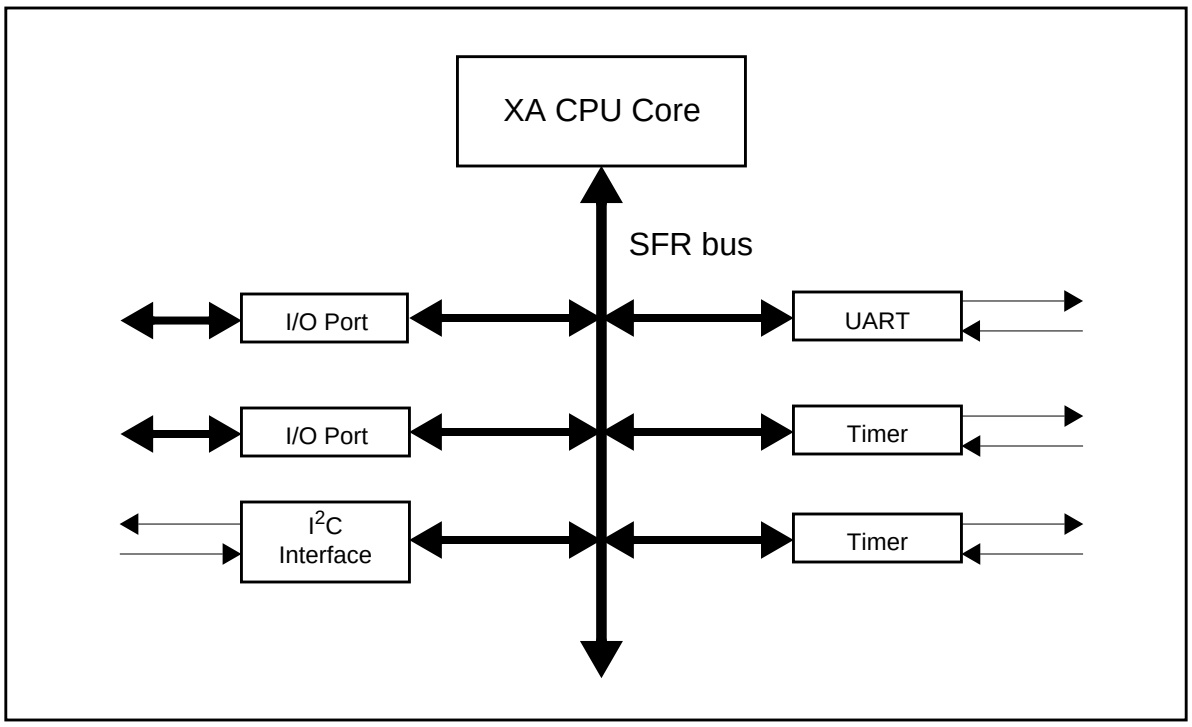


Figure 8.1. Example of peripheral functions connected to the XA SFR bus.

8.1 Implementation and Possible Enhancements

The SFR bus interface is itself not part of the XA CPU core, but a separate functional block. Since the SFR bus controller is a separate block, writes to SFRs may occur simultaneously with the beginning of execution of the next instruction. If the next instruction attempts to access the SFR bus while it is still busy, the instruction execution will stall until the SFR bus becomes available. SFR bus read and write clocks each take 2 CPU clocks to complete. However, the starting time of those 2 clocks has a one clock uncertainty, so the time from the SFR bus controller receiving a request until it is completed can be either 2 or 3 clocks.

The SFR bus implementation on initial XA derivatives is an 8-bit interface. This means that word reads and writes are not allowed. In the future, higher performance XA architecture implementations may expand the capabilities of the SFR bus by supporting 16-bit accesses.

One enhancement to the SFR bus would be to have it divide 16-bit access requests into two 8-bit accesses. This leaves the actual SFR bus width at 8 bits, but allows a user program to act as if it was 16-bits. The highest performance alternative is a full 16-bit SFR bus. This would require extra hardware in the XA to implement, but may eventually become necessary on order to achieve very high performance with some future enhanced XA core implementation.

8.2 Read-Modify-Write Lockout

Some of the SFRs that are accessed via the SFR bus contain interrupt flags and other status bits that are set directly by the peripheral device. When a read-modify-write operation is done on such an SFR, there is a possibility that a peripheral write to a flag bit in the same SFR could occur in the middle of this process. A standard mechanism is defined for the XA to deal with such cases, which is called Read-Modify-Write lockout. A read- modify-write is defined as an operation where a particular SFR is read, altered and written during the execution of a single XA instruction.

The instructions that fit this description are those that write to bits in SFRs and those that modify an entire SFR, except for the MOV instruction. This happens to be the same operations as those that read port latches rather than port pins as specified in Chapter 7, only the SFRs involved are different.

The mechanism used throughout XA peripherals to avoid losing status flags during a read-modify-write operation first involves detecting that such an operation is in progress. A signal from the CPU to the peripherals indicates such a condition. When a peripheral detects this, it prevents the CPU write to just those status flags that the peripheral has already updated since the beginning of the read-modify-write operation. This basically makes it look as if the peripheral flag update happened just after the read-modify-write operation completed, rather than during it. Once the read-modify-write operation is completed, a CPU write may affect all bits in these SFRs.

9 80C51 Compatibility

Many architectural decisions and features were guided by the goal of 80C51 compatibility when the XA core specification was written. The processor's memory configuration, memory addressing modes, instruction set, and many other things had to be taken into account.

9.1 Compatibility Considerations

Source code compatibility of the XA to the 80C51 was chosen as a goal for many reasons. Complete compatibility with an existing processor is not possible if the new processor is to have substantially higher performance.

The XA architecture makes use of a number of rules for 80C51 compatibility. An 80C51 to XA source code translator program is intended to be the means of providing compatibility between the architectures. For the translator software to be fairly simple, a one-to-one translation for all 80C51 instructions is a major consideration. The XA instruction set includes many instructions that are more powerful than 80C51 instructions and yet perform roughly the same function. 80C51 instruction can therefore be translated into those XA instructions. When this is not the case, an 80C51 instruction may be included in its original form in the XA. The XA memory map and memory addressing modes are also a superset of the 80C51, making source code translation easy to accomplish. Other CPU features are made compatible to the extent that such is possible. In rare cases, when this compatibility could not be provided for some important reason, the changes were kept to the minimum while maintaining the XA goals of high performance and low cost.

9.1.1 Compatibility Mode, Memory Map, and Addressing

Specific XA registers are reserved for use as 80C51 registers when translating code. The A register, the B register, and the data pointer all map to a pre-determined place in the XA register file (see figure 9.1). The accumulator (A) is the only one of these that required special hardware support in the XA, because the accumulator can be read or tested directly by certain instructions and in order to generate the parity flag.

The 4 banks of 8 byte registers that are found in the 80C51 are duplicated in the XA. The only difference is that in the XA, these registers do not normally overlap the lower 32 bytes of data memory space as they do in the 80C51. To allow code translation, a special 80C51 compatibility mode causes the XA register file to copy the 80C51 mapping to data memory. This mode is activated by the CM bit in the System Configuration Register (SCR).

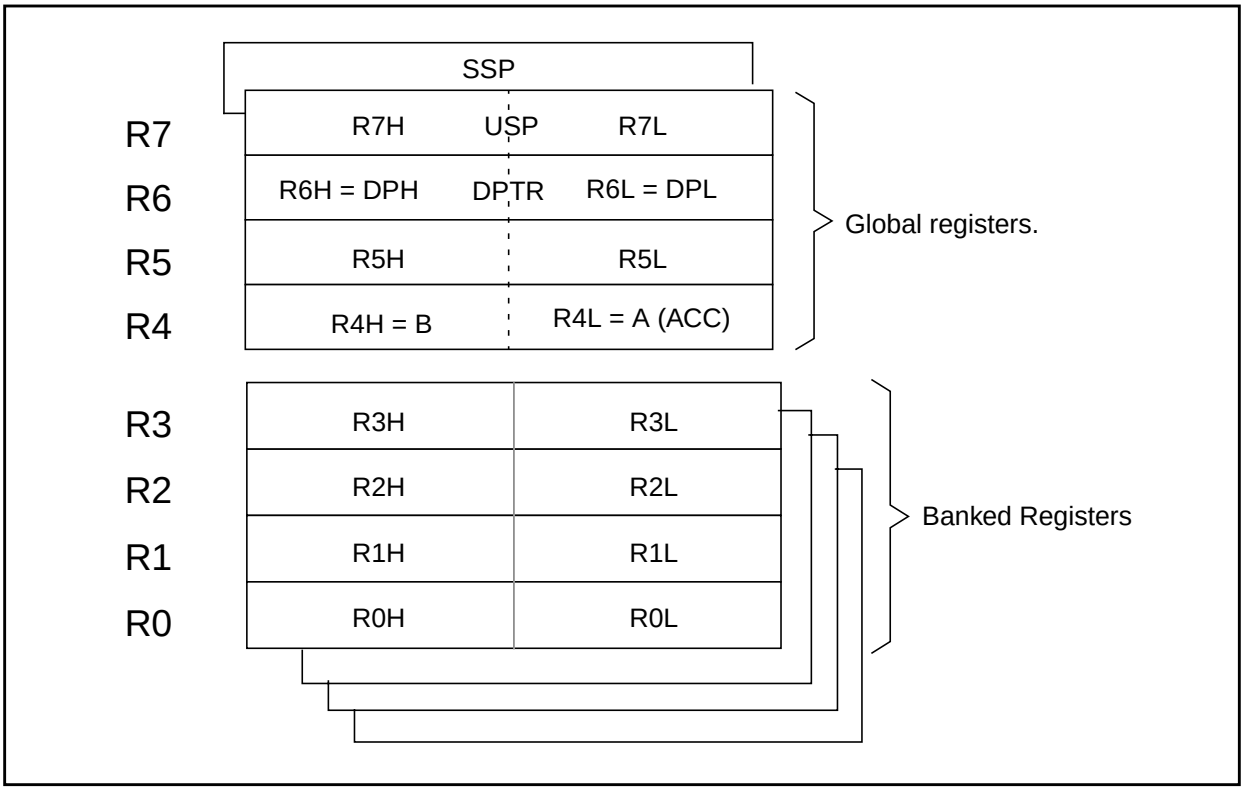


Figure 9.1. XA Register File

Other important registers of the 80C51 are provided in other ways. The program status word (PSW) of the XA is slightly different than the 80C51 PSW, so a special SFR address is reserved to provide an 80C51 compatible "view" of the PSW for use by translated code. This alternate PSW, called PSW51, is shown in the figure 9.2. The F0 flag and the F1 flag are simply readable

Figure 9.2. PSW CPU status flags

and writable bits. The P flag provides an even parity bit for the 80C51 A register and always reflects the current contents of that register. Note that the P flag, the F0 flag, and the F1 flag only appear in the PSW51 register.

The 80C51 indirect data memory access mode, using R0 or R1 as pointers, requires special support on the XA, where pointers are normally 16 bits in length. The 80C51 compatibility mode also causes the XA to mimic the 80C51 indirect scheme, using the first two bytes of the register file as indirect pointers, each zero extended to make a 16-bit address. Due to this and the previously mentioned register overlap to memory feature, the compatibility mode must be turned on in order to execute most translated 80C51 code on the XA. Other than the two aforementioned effects, nothing else about XA functioning is affected by the compatibility mode.

The 80C51 mapped the special function registers (SFRs) into the direct address space, from address 80 hex to FF hex. SFRs were only accessed by instruction that contain the entire SFR address, so translation to the XA is fairly simple. Since references to SFRs are normally done by their name in 80C51 source code, the translation just copies the name into the XA code output. If an SFR happened to be referred to by its address, its name must be found so that it can be inserted into the XA code. This would require that an SFR table be available for the 80C51 derivative for which the code was originally written.

The XA has another mode which may be useful for translated 80C51 code. In order to save stack space as well as speed up execution, a Page Zero (PZ) mode causes return addresses on the stack to be saved as 16 bits only, instead of the usual 24 bits (which occupy 32 bits due to word alignment on the XA stack). All other program and data addresses are also forced to be 16-bits. If an entire 80C51 application program is translated to the XA, it will very likely fit within this 64K limit, allowing the use of this mode.

Other aspects of the processor stack have been altered on the XA. For one, the standard direction of stack growth for 16 bit processors has been adopted. So, the XA stack grows downward, from higher to lower addresses in data memory. The stack can now be nearly 64K in size if necessary, and begin anywhere in its data segment so may be easily moved to a new location for translated 80C51 applications. This stack direction change is important to match the stack contents to normal data memory accesses on the XA.

80C51 code translated to run on the XA will also tend to use more stack space for two reasons. First, the PSW is automatically saved during interrupt and exception processing on the XA. The original 80C51 code should have also saved the PSW explicitly, but the XA PSW is 16 bits in length. Secondly, the initial implementation of the XA allows only word writes to the stack. Both byte and word operations may be performed, but both types of operations use 16 bits of stack space.

The tendency for stack size increase, in addition to the stack growth direction will require some changes to be made if a complete 80C51 application program is translated to run on the XA.

9.1.2 Interrupt and Exception Processing

Interrupt handling on the XA is inherently much more powerful than it was on the 80C51. Along with this added power and flexibility comes some difference that must be taken into account for 80C51 code conversion.

Previously noted was the fact that the XA automatically saves the PSW during interrupt processing. If an 80C51 program relied on this not being the case somehow, it would not work without alteration. This type of reliance is not found in code using common programming practices and should be very rare.

The XA allows up to 15 interrupt priority levels, compared to only 2 in the standard 80C51, although up to 4 levels are available in a few of the newer 80C51 variations. These priorities are stored as 4-bit values, with the priority for 2 interrupts found in the same SFR byte. This is

different (and much more powerful) than any 80C51 derivative, and will require minor changes to code that is translated.

The method of entering an interrupt routine in the XA uses a vector table stored in low addresses of the code memory. Each interrupt or exception source has a vector which consists of the address of the handler routine for that event and a new PSW value that is loaded when the vector is taken. This differs from the 80C51 approach of fixed addresses for the interrupt service routines, and again is a much more flexible and powerful method. So, if a complete 80C51 application program is converted for the XA, the interrupt service routines must be re-located above the XA vector table and the new address stored in the table, a very simple process.

9.1.3 On-Chip Peripherals

Compatibility with standard on-chip peripherals found in the 80C51 has been kept in the XA whenever possible and reasonable, but not to the extent that some enhancements are not made. The set of standard peripheral devices includes the UART, Timers 0 and 1, and Timer 2 from the 80C52.

The XA UART has been enhanced in a way that does not affect translated 80C51 code. Some additional features are added through the use of a new SFR, such as framing error detection, overrun detection, and break detection.

Timers 0 and 1 remain the same except for one difference in the function, and a difference in timing. The functional change was to remove the 8048 timer mode (mode 0) and replace it with something much more useful: a 16-bit auto-reload mode. Sixteen bit reload registers (formed by RTHn and RTLn) had to be added to Timers 0 and 1 to support the new mode 0. In mode 2, RTLn also replaces THn as the 8-bit reload register.

The relationship of all timer count rates to the microcontroller oscillator has also been changed. This adds flexibility since this is now a programmable feature, allowing oscillator divided by 4, 16, or 64 to be used as the base count rate for all of the timers. Since XA performance is much higher (on a clock-by clock basis), an application converted to the XA from the 80C51 would likely not use the same oscillator frequency anyway.

9.1.4 Bus Interface

The customary 80C51 bus control signals are all found on the standard external XA bus. To provide the best performance, the details of some of these signals have changed somewhat, and a few new ones have been added. In addition to the well known ALE, $\overline{\text{PSEN}}$, $\overline{\text{RD}}$, $\overline{\text{WR}}$, and $\overline{\text{EA}}$, there are now also WAIT and $\overline{\text{WRH}}$. The WAIT signal causes wait states to be inserted into any XA bus clock as long as it is asserted. The $\overline{\text{WRH}}$ signal is used to distinguish writes to the high order byte when the XA bus is configured to be 16 bits wide.

The multiplexed address/data bus has undergone some renovations on the XA as well. To get the most performance in a system executing code from the external bus, the XA separates the 4 least significant address lines on to their own pins. Since these lines normally change the most often, an ALE clock would be required on every external code fetch if these lines were multiplexed as they are on the 80C51. The 80C51 had time to do this since its performance was not that high.

The XA, however, uses only as many clocks as are needed to execute each instruction, so an ALE for every fetch would slow things down considerably. With this change, up to 16 bytes (or 8 words) of code may be accessed without the need to insert an ALE cycle on the XA bus.

The number of XA clocks used for each type of bus cycle (code read, data read, or data write) can also be programmed, so that slower peripheral devices can work with the XA without the need for an external WAIT state generator.

Due to the various changes to the bus just mentioned, an XA device cannot be completely pin compatible with an 80C51 derivative if the external bus is used. The changes to application hardware needed are relatively small and easy to make.

9.1.5 Instruction Set

The simplest goal of the XA for instruction set compatibility was to have every 80C51 instruction translate to one XA instruction. That has been achieved but for a single exception. The 80C51 instruction, XCHD or exchange digits, cannot be translated in that manner. XCHD is an instruction that is rarely used on the 80C51 and could not be implemented on the XA, due to its internal architecture, without adding a great deal of extra circuitry. So, if this instruction is encountered when 80C51 source code is being translated, a sequence of XA instructions is used to duplicate the function:

```
PUSH    R4H         ; Save temporary register.
MOV     R4H,(Ri)    ; Get second operand.
RR      R4H,#4      ; Swap one byte.
RR      R4L,#4      ; Swap second byte (the "A" register).
RL      R4,#4       ; Swap word.
                    ; Result is swapped nibbles in A and R4H.
MOV     (Ri),R4H    ; Store result.
POP     R4H         ; Restore temporary register.
```

If the application requires this sequence to not be interruptible, some additional instruction must be added in order to disable and re-enable interrupts. The table at the end of this section shows all of the other XA code replacements for 80C51 instructions.

The XA instruction set is much more powerful than the 80C51 instruction set, and as a direct consequence, the average number of bytes in an instruction is higher on the XA. In code written for the XA, the capability of a single instruction is high, so the size of an entire XA program will normally be smaller than the same program written for an 80C51. Of course, this depends on how much the application can take advantage of XA features. When code is translated from 80C51 source, however, the size change can be an issue.

In the case of a jump table, where the JMP @A+DPTR instruction is used to jump into a table of other jumps composed of the 80C51 AJMP instruction, the XA cannot always duplicate the function of the jumps in the table with instructions that are 2 bytes in length, as in the case of the AJMP instruction. An adjustment to the calculation of the table index will be required to make the translated code work properly. For a data table, accessed using MOVC @A+PC, the distance to the table may change, requiring a similar index adjustment.

Since the XA optimizes the timing of each instruction, there will be very little correspondence to the original 80C51 timing for the same code prior to translation to the XA. If the exact timing of a sequence of instructions is important to the application, the translated code must be altered, perhaps by adding NOPs or delay loops, to provide the necessary timing.

To show how a simple 80C51 to XA source code translator might work, a subroutine was extracted from a working 80C51 program and translated using the table at the end of this document and the other rules presented here. The original 80C51 source code was:

```
;StepCal - Calculates a trip point value for motor movement based on
;  a percent of pointer full scale (0 - 100%).
;  Call with target value in A. Returns result in A and "StepResult".

StepCal:  MOV     Temp2,A          ; Save step target for later use.
          MOV     B,#Steplow       ; Get low byte of step increment.
          MUL     AB               ; Multiply this by the step target.
          MOV     StepResult,B     ; Save high byte as partial result.
          MOV     Temp1,A          ; Save low byte to use for rounding.

          MOV     A,Temp2          ; Get back the step target.
          MOV     B,#StepHigh      ; Get high byte of step increment,
          MUL     AB               ; and multiply the two.

          ADD     A,StepResult     ; Add the two partial results.
          JNB     Temp1.7,Exit     ; Least significant byte > 80h?
          INC     A                ; If so, round up the final result.
Exit:     ADD     A,#MotorBot      ; Add in the 0 step displacement.
          MOV     StepResult,A     ; Save final step target.
          RET
```

The same code as translated for the XA is as follows:

```
;StepCal - Calculates a trip point value for motor movement based on
;  a percent of pointer full scale (0 - 100%).
;  Call with target value in A. Returns result in A and "StepResult".

StepCal:  MOV     Temp2,R4L        ; Save step target for later use.
          MOV     R4H,#Steplow     ; Get low byte of step increment.
          MULU.b  R4,R4H           ; Multiply this by the step target.
          MOV     StepResult,R4H   ; Save high byte as partial result.
          MOV     Temp1,R4L        ; Save low byte to use for rounding.

          MOV     R4L,Temp2        ; Get back the step target.
          MOV     R4H,#StepHigh    ; Get high byte of step increment,
          MULU.b  R4,R4H           ; and multiply the two.

          ADD     R4L,StepResult   ; Add the two partial results.
          JNB     Temp1.7,Exit     ; Least significant byte > 80h?
          ADDS    R4L,#1           ; If so, round up the final result.
Exit:     ADD     R4L,#MotorBot    ; Add in the 0 step displacement.
          MOV     StepResult,R4    ; Save final step target.
          RET
```

In this case, the translated code actually changed very little. Primarily, the 80C51 register names have been replaced by the new ones reserved for them in the XA. The increment (INC) instruction became a short add (ADDS), and the mnemonic for multiply (MUL) changed to MULU8.

Some basic statistical information about these code samples may be found in table 9.1. These statistics show a large performance increase for the XA code. This is significant because the code is only simple translated 80C51 code and therefore does not take any advantage of the XA's unique features.

Table 9.1: 80C51 to XA Code Translation Statistics

Statistic	80C51 code	XA translation	Comments
Code bytes	28	40	- one NOP added for branch alignment on XA
Clocks to execute	300	78	- includes XA pre-fetch queue analysis, raw execution is 66 clocks
Time to execute @ 20MHz	15 µsec	3.9 µsec	- a nearly 4x improvement without any optimization

9.2 Code Translation

Table 9.2 shows every 80C51 instruction type and the XA instruction that replaces it. An actual 80C51 to XA source code translator can make use of this table, but must also flag the compatibility exceptions noted in this section, so that any necessary adjustments may be made to the resulting XA source code.

Table 9.2: 80C51 to XA Instruction Translations

80C51 Instruction	XA Translation
Arithmetic operations	

ADD A, Rn
ADD A, #data8
ADD A,dir8
ADD A, @Ri
ADDC A, Rn
ADDC A, #data8
ADDC A,dir8
ADDC A, @Ri
	ADD.b R, R
	ADD.b R, #data8
	ADD.b R, direct
	ADD.b R, [R]
	ADDC.bR, R
	ADDC.bR, #data8
	ADDC.bR, direct
	ADDC.bR, [R]

SUBB A, Rn
SUBB A, #data8
SUBB A, dir8
SUBB A, @Ri
	SUBB.bR, R
	SUBB.bR, #data8
	SUBB.bR, direct
	SUBB.bR, [R]

INC Rn INC dir8 INC @Ri INC A INC DPTR	ADDS.bR, #1 ADDS.bdirect, #1 ADDS.b[R], #1 ADDS.bR, #1 ADDS.wR, #1
DEC Rn DEC dir8 DEC @Ri DEC A	ADDS.bR, #-1 ADDS.bdirect, #-1 ADDS.b[R], #-1 ADDS.bR, #-1
MUL AB DIV AB DA A	MULU.bR, R DIVU.b R, R DA R

Table 9.2: 80C51 to XA Instruction Translations

80C51 Instruction	XA Translation
Logical operations	
ANL A, Rn ANL A, #data8 ANL A, dir8 ANL A, @Ri ANL dir8, A ANL dir8, #data8	AND.b R, R AND.b R, #data8 AND.b R, direct AND.b R, [R] AND.b direct, R AND.b direct, #data8
ORL A, Rn ORL A, #data8 ORL A, dir8 ORL A, @Ri ORL dir8, A ORL dir8, #data8	OR.b R, R OR.b R, #data8 OR.b R, direct OR.b R, [R] OR.b direct, R OR.b direct, #data8
XRL A, Rn XRL A, #data8 XRL A, dir8 XRL A, @Ri XRL dir8, A XRL dir8, #data8	XOR.b R, R XOR.b R, #data8 XOR.b R, direct XOR.b R, [R] XOR.b direct, R XOR.b direct, #data8
CLR A CPL A SWAP A	MOVS R, #0 CPL.b R RL.b R, #4
RL A RLC A RR A RRC A	RL.b R, #1 RLC.b R, #1 RR.b R, #1 RRC.b R, #1

CLR C
CLR bit
SETB C
SETB bit
CPL C
CPL bit
ANL C, bit
ANL C, /bit
ORL C, bit
ORL C, /bit
MOV C, bit
MOV bit, C
	CLR bit
	CLR bit
	SETB bit
	SETB bit
	XOR.b PSWL, #data8
	XOR.b direct, #data8
	AND C, bit
	AND C, /bit
	OR C, bit
	OR C, /bit
	MOV C, bit
	MOV bit, C

Table 9.2: 80C51 to XA Instruction Translations

80C51 Instruction	XA Translation
Data transfer	
MOV A, Rn	MOV.b R, R
MOV A, #data8	MOV.b R, #data8
MOV A, dir8	MOV.b R, direct
MOV A, @Ri	MOV.b R, [R]
MOV Rn, A	MOV.b R, R
MOV Rn, #data8	MOV.b R, #data8
MOV Rn, dir8	MOV.b R, direct
MOV dir8, A	MOV.b direct, R
MOV dir8, #data8	MOV.b direct, #data8
MOV dir8, Rn	MOV.b direct, R
MOV dir8, dir8	MOV.b direct, direct
MOV dir8, @Ri	MOV.b direct, [R]
MOV @Ri, A	MOV.b [R], R
MOV @Ri, dir8	MOV.b [R], direct
MOV @Ri, #data8	MOV.b [R], #data8
MOV DPTR, #data16	MOV.w R, #data16
XCH A, Rn	XCH.b R, R
XCH A, dir8	XCH.b R, direct
XCH A, @Ri	XCH.b R, R
XCHD A, @Ri	a sequence (see text)
PUSH dir8	PUSH.bdirect
POP dir8	POP.b direct
MOVX A, @Ri	MOVX.bR, [R]
MOVX A, @DPTR	MOVX.bR, [R]
MOVX @Ri, A	MOVX.b[R], R
MOVX @DPTR, A	MOVX.b[R], R
MOVC A, @A+DPTR	MOVC.bA, [A+DPTR]
MOVC A, @A+PC	MOVC.bA, [A+PC]

Table 9.2: 80C51 to XA Instruction Translations

80C51 Instruction	XA Translation
Relative branches	
SJMP rel8	BR rel8
CJNE A, dir8, rel CJNE A, #data8, rel CJNE Rn, #data8, rel CJNE @Ri, #data8, rel	CJNE.b R, direct, rel CJNE.b R, #data8, rel CJNE.b R, #data8, rel CJNE.b [R], #data8, rel
DJNZ Rn, rel DJNZ dir8, rel	DJNZ.b R, rel DJNZ.b direct, rel
JZ rel JNZ rel JC rel JNC rel	JZ rel JNZ rel BCS rel BCC rel
Jumps, Calls, Returns, and Misc.	
NOP	NOP
AJMP addr11 LJMP addr16 JMP @A+DPTR	JMP rel16 JMP rel16 JUMP [A+DPTR]
ACALL addr11 LCALL addr16	CALL rel16 CALL rel16
RET RETI	RET RETI

9.3 New Instructions on the XA

While the XA instructions that are similar to 80C51 instructions have a larger addressing range, more status flags, etc., the XA also has many entirely new instructions and addressing modes that make writing new code for the XA much easier and more efficient. The new addressing modes also make the XA work very well with high level language compilers. A complete list of the new XA instructions and addressing modes is shown in Table 9.3.

Table 9.3: Instructions and addressing modes new to the XA

New Instructions and Addressing Modes	
alu.w ..., ...	All of the 80C51 arithmetic and logic instructions with a 16-bit data size.
SUBB R,...	Subtract (without borrow), all addressing modes.
alu [R], R	Arithmetic and logic operations (ADD, ADDC, SUB, SUBB, CMPAND, OR, XOR, and MOV) from a register to an indirect address.
alu R, [R+]	Arithmetic and logic operations from an indirect address to a register, with the indirect pointer automatically incremented.
alu R,[R+offset8/16]	Arith/Logic operations from an indirect offset address (with 8 or 16-bit offset) to a register.
alu direct, R	The 80C51 has only MOV direct, R.
alu [R], R	The 80C51 has only MOV [R], R.
alu [R+], R	Arith/Logic operations from a register to an indirect address, with the indirect pointer automatically incremented.
alu [R+offset8/16], R	Arith/Logic operations from a register to an indirect offset address (with 8 or 16-bit offset).
alu direct, #data8/16	Arith/Logic operations to a direct address with 8 or 16-bit immediate data.
alu [R], #data8/16	Arith/Logic operations to an indirect address with 8 or 16-bit immediate data.
alu [R+], #data8/16	Arith/Logic operations to an indirect address with 8 or 16-bit immediate data with the indirect pointer automatically incremented.
alu [R+offset8/16], #data8/16	Arith/Logic operations to an indirect offset address (with 8 or 16-bit offset), with 8 or 16-bit immediate data.
MOV direct, [R]	Move data from an indirect to a direct address.
ADDS R, #data4	The 80C51 can only increment or decrement a register by 1. ADDS has a range of +7 to -8.
ADDS [R], #data4	Add a short value to an indirect address.

Table 9.3: Instructions and addressing modes new to the XA

New Instructions and Addressing Modes	
ADDS [R+], #data4	Add a short value to an indirect offset address, with the indirect pointer automatically incremented.
ADDS [R+offset8/16], #data4	Add a short value to an indirect offset address (with 8 or 16-bit offset).
ADDS direct, #data4	Add a short value to a direct address.
MOVS ..., #data4	Move short data to destination using any of the same addressing modes as ADDS.
ASL R, R	Arithmetic shift left a byte, word, or double word, up to 31 places, shift count read from register.
ASR R, R	Arithmetic shift right a byte, word, or double word, up to 31 places, shift count read from register.
LSR R, R	Logical shift right a byte, word, or double word, up to 31 places, shift count read from register.
ASL R, #DATA4/5	Arithmetic shift left a byte, word, or double word, up to 31 places, shift count read from instruction.
ASR R, #DATA4/5	Arithmetic shift right a byte, word, or double word, up to 31 places, shift count read from instruction.
LSR R, #DATA4/5	Logical shift right a byte, word, or double word, up to 31 places, shift count read from instruction.
DIV R, R	Signed divide of 32 bits register by 16 bit register, or 16 bit register by 8 bit register.
DIVU R, R	Unsigned divide of 32 bit register by 16 bit register, or 16 bit register by 8 bit register.
MUL R, R	Signed multiply of 16 bit register by 16 bit register, or 8 bit register by 8 bit register.
MULU R, R	Unsigned multiply of 16 bit register by 16 bit register.
DIV R, #data8/16	Signed divide of 32 bits register by 16 bit immediate, or 16 bit register by 8 bit immediate.
DIVU R, #data8/16	Unsigned divide of 32 bit register by 16 bit immediate, or 16 bit register by 8 bit immediate.
MUL R, #data8/16	Signed multiply of 16 bit register by 16 bit immediate, or 8 bit register by 8 bit immediate.

Table 9.3: Instructions and addressing modes new to the XA

New Instructions and Addressing Modes	
MULU R, #data8/16	Unsigned multiply of 16 bit register by 16 bit immediate, or 8 bit register by 8 bit immediate.
LEA R, R+offset8/16	Load effective address, duplicates the offset8 or 16-bit addressing mode calculation but saves the address in a register.
NEG R	Negate, performs a twos complement operation on a register.
SEXT R	Sign extend, copies the sign flag from the last operation into an 8 or 16-bit register.
NORM R, R	Normalize. Shifts a byte, word, or double word register left until the MSB becomes a 1. The number of shifts used is stored in a register.
RL, RR, RLC, RRC R,#data4	All of the 80C51 rotate modes with 16-bit data size and a variable number of bit positions (up to 15 places).
MOV [R+], [R+]	Block move. Move data from an indirect address to another indirect address, incrementing both pointers.
MOV R, USP and USP, R	Allows system code to move a value to or from the user stack pointer. Handy in multi-tasking applications.
MOVC R, [R+]	Move data from an indirect address in the code space to a register, with the indirect pointer automatically incremented.
PUSH and POP Rlist	PUSH and POP up to 8 word registers in one instruction.
PUSHU and POPU Rlist or direct	Allows system code to write to or read the user stack. Handy in multi-tasking applications.
conditional branches	A complete set of conditional branches, including BEQ, BNE, BG, BGE, BGT, BL, BLE, BMI, BPL, BNV, and BOV.
CALL [R]	Call indirect, to an address contained in a register.
CALL rel16	Call anywhere in a +/- 64K range.

Table 9.3: Instructions and addressing modes new to the XA

New Instructions and Addressing Modes	
FCALL addr24	Far call, anywhere within the XA 16Mbyte code address space.
JMP [R]	Jump indirect, to an address contained in a register.
JMP rel16	Jump anywhere in a +/- 64K range.
FJMP addr24	Far jump, anywhere within the XA 16Mbyte code address space.
JMP [[R+]]	Jump double indirect with auto-increment. Used to branch to a sequence of addresses contained in a table.
BKPT	Breakpoint, a debugging feature.
RESET	Allows software to completely reset the XA in one instruction.
TRAP #data4	Call one of up to 16 system services. Acts like an immediate interrupt.

CMOS 0 to 44 MHz Single Chip 8–bit Microntroller

Description

TEMIC's 80C52 and 80C32 are high performance CMOS versions of the 8052/8032 NMOS single chip 8 bit µC.

The fully static design of the TEMIC 80C52/80C32 allows to reduce system power consumption by bringing the clock frequency down to any value, even DC, without loss of data.

The 80C52 retains all the features of the 8052 : 8 K bytes of ROM ; 256 bytes of RAM ; 32 I/O lines ; three 16 bit timers ; a 6-source, 2-level interrupt structure ; a full duplex serial port ; and on-chip oscillator and clock circuits. In addition, the 80C52 has 2 software-selectable modes of reduced activity for further reduction in power consumption. In the idle mode the CPU is frozen while the RAM, the timers, the serial port and the interrupt system continue to function. In the power down mode the RAM is saved and all other functions are inoperative.

The 80C32 is identical to the 80C52 except that it has no on-chip ROM. TEMIC's 80C52/80C32 are manufactured using SCMOS process which allows them to run from 0 up to 44 MHz with Vcc = 5 V.

TEMIC's 80C52 and 80C32 are also available at 16 MHz with 2.7 V < V_{CC} < 5.5 V.

- 80C32 : Romless version of the 80C52
- 80C32/80C52-L16 : Low power version
  Vcc : 2.7 – 5.5 V Freq : 0-16 MHz
- 80C32/80C52-12 : 0 to 12 MHz
- 80C32/80C52-16 : 0 to 16 MHz
- 80C32/80C52-20 : 0 to 20 MHz
- 80C32/80C52-25 : 0 to 25 MHz
- 80C32/80C52-30 : 0 to 30 MHz
- 80C32/80C52-36 : 0 to 36 MHz
- 80C32-40 : 0 to 40 MHz*
- 80C32-42 : 0 to 42 MHz*
- 80C32-44 : 0 to 44 MHz*

* 0 to 70°C temperature range.
For other speed and temperature range availability please consult your sales office.

Features

- Power control modes
- 256 bytes of RAM
- 8 Kbytes of ROM (80C52)
- 32 programmable I/O lines
- Three 16 bit timer/counters
- 64 K program memory space
- 64 K data memory space
- Fully static design
- 0.8µ CMOS process
- Boolean processor
- 6 interrupt sources
- Programmable serial port
- Temperature range : commercial, industrial, automotive, military

Optional

- Secret ROM : Encryption
- Secret TAG : Identification number

Interface

Figure 1. Block Diagram

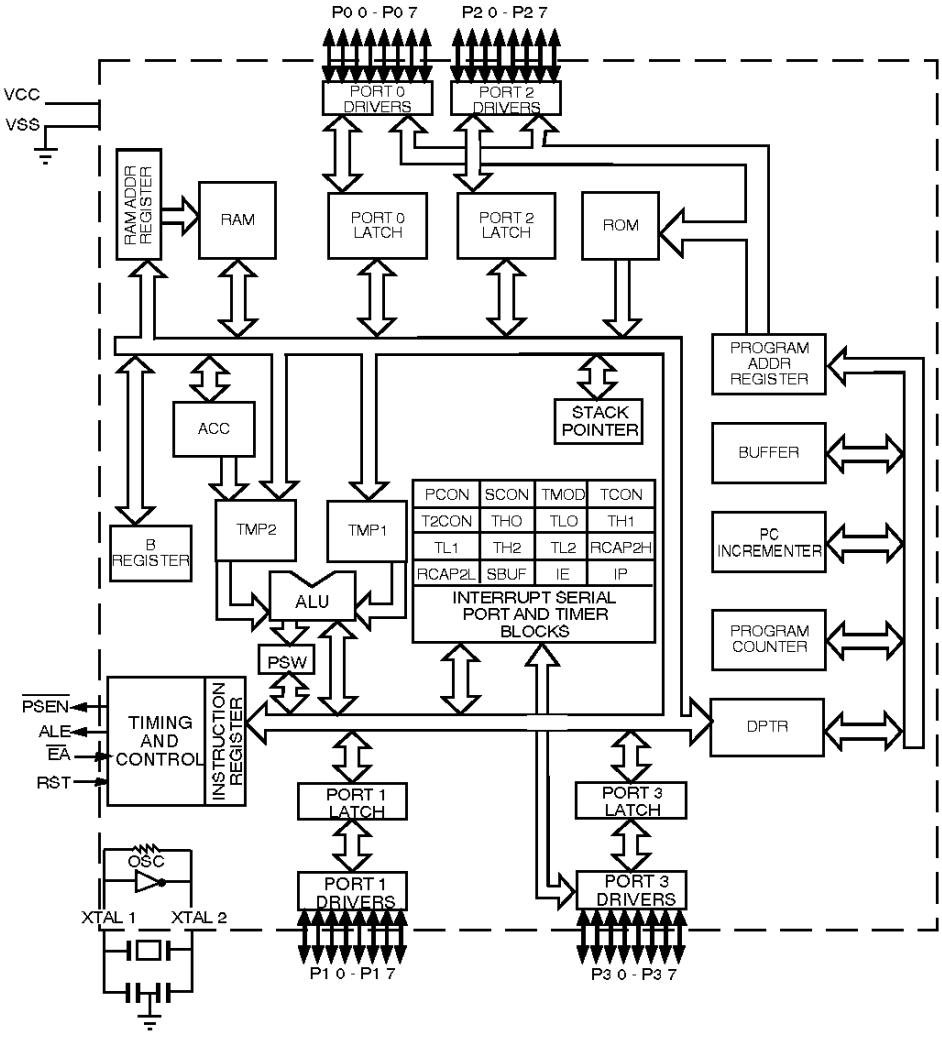
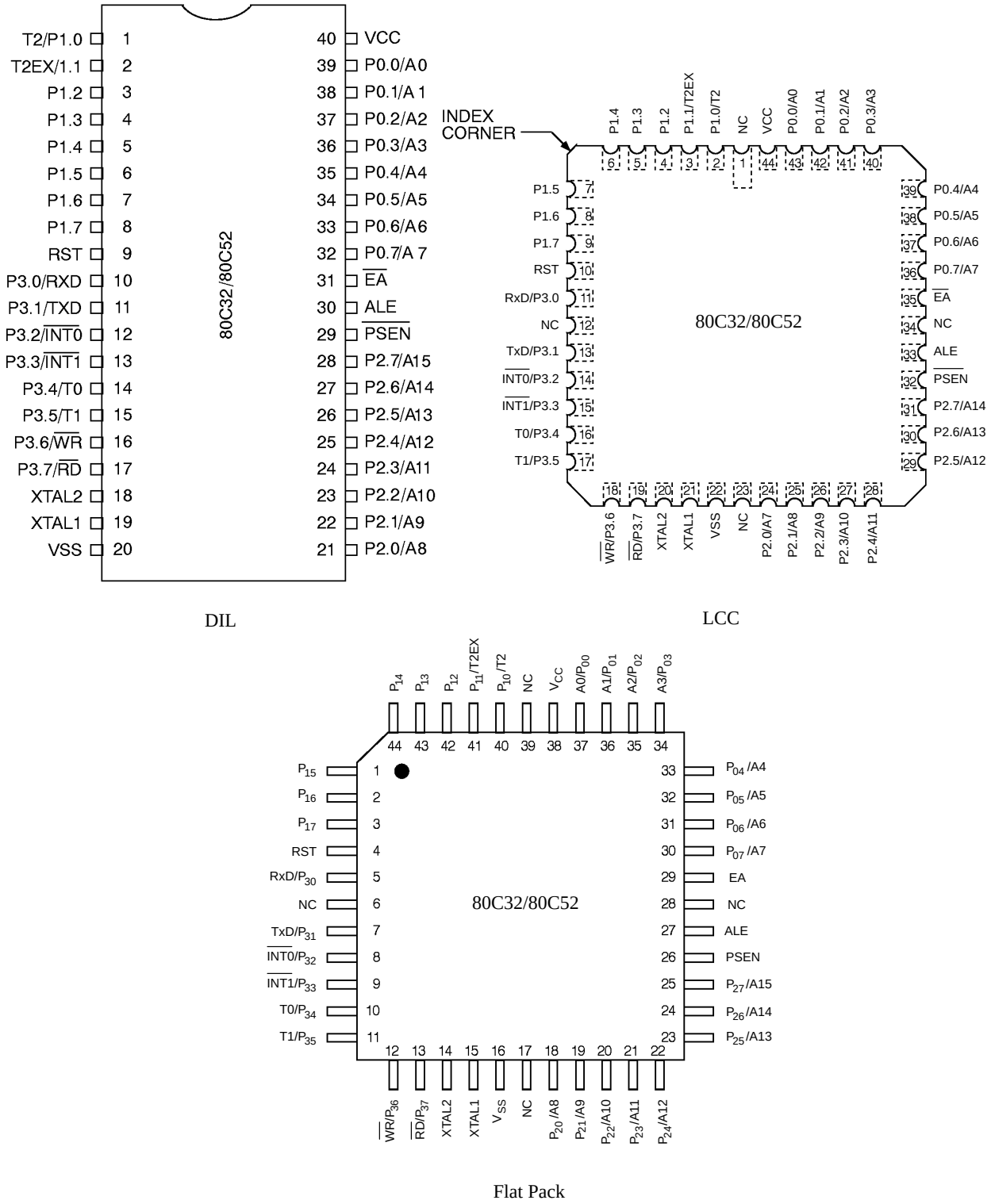


Figure 2. Pin Configuration



Diagrams are for reference only. Package sizes are not to scale.

Pin Description

VSS

Circuit ground potential.

VCC

Supply voltage during normal, Idle, and Power Down operation.

Port 0

Port 0 is an 8 bit open drain bi-directional I/O port. Port 0 pins that have 1's written to them float, and in that state can be used as high-impedance inputs.

Port 0 is also the multiplexed low-order address and data bus during accesses to external Program and Data Memory. In this application it uses strong internal pullups when emitting 1's. Port 0 also outputs the code bytes during program verification in the 80C52. External pullups are required during program verification. Port 0 can sink eight LS TTL inputs.

Port 1

Port 1 is an 8 bit bi-directional I/O port with internal pullups. Port 1 pins that have 1's written to them are pulled high by the internal pullups, and in that state can be used as inputs. As inputs, Port 1 pins that are externally being pulled low will source current (IIL, on the data sheet) because of the internal pullups.

Port 1 also receives the low-order address byte during program verification. In the 80C52, Port 1 can sink/ source three LS TTL inputs. It can drive CMOS inputs without external pullups.

2 inputs of PORT 1 are also used for timer/counter 2 :

P1.0 [T2] : External clock input for timer/counter 2. P1.1 [T2EX] : A trigger input for timer/counter 2, to be reloaded or captured causing the timer/counter 2 interrupt.

Port 2

Port 2 is an 8 bit bi-directional I/O port with internal pullups. Port 2 pins that have 1's written to them are pulled high by the internal pullups, and in that state can be used as inputs. As inputs, Port 2 pins that are externally being pulled low will source current (ILL, on the data sheet) because of the internal pullups. Port 2 emits the high-order address byte during fetches from external Program Memory and during accesses to external Data Memory that use 16 bit addresses (MOVX @DPTR). In this application, it uses strong internal pullups when emitting 1's. During accesses to external Data Memory that use 8 bit addresses (MOVX @Ri), Port 2 emits the contents of the P2 Special Function Register.

It also receives the high-order address bits and control signals during program verification in the 80C52. Port 2 can sink/source three LS TTL inputs. It can drive CMOS inputs without external pullups.

Port 3

Port 3 is an 8 bit bi-directional I/O port with internal pullups. Port 3 pins that have 1's written to them are pulled high by the internal pullups, and in that state can be used as inputs. As inputs, Port 3 pins that are externally being pulled low will source current (ILL, on the data sheet) because of the pullups. It also serves the functions of various special features of the TEMIC 51 Family, as listed below.

Port Pin	Alternate Function
P3.0	RXD (serial input port)
P3.1	TXD (serial output port)
P3.2	$\overline{INT0}$ (external interrupt 0)
P3.3	$\overline{INT1}$ (external interrupt 1)
P3.4	TD (Timer 0 external input)
P3.5	T1 (Timer 1 external input)
P3.6	$\overline{WR}$ (external Data Memory write strobe)
P3.7	RD (external Data Memory read strobe)

Port 3 can sink/source three LS TTL inputs. It can drive CMOS inputs without external pullups.

RST

A high level on this for two machine cycles while the oscillator is running resets the device. An internal pull-down resistor permits Power-On reset using only a capacitor connected to V_{CC}. As soon as the Reset is applied (Vin), PORT 1, 2 and 3 are tied to one. This operation is achieved asynchronously even if the oscillator does not start-up.

ALE

Address Latch Enable output for latching the low byte of the address during accesses to external memory. ALE is activated as though for this purpose at a constant rate of 1/6 the oscillator frequency except during an external data memory access at which time one ALE pulse is skipped. ALE can sink/source 8 LS TTL inputs. It can drive CMOS inputs without an external pullup.

$\overline{\text{PSEN}}$

Program Store Enable output is the read strobe to external Program Memory. PSEN is activated twice each machine cycle during fetches from external Program Memory. (However, when executing out of external Program Memory, two activations of PSEN are skipped during each access to external Data Memory). PSEN is not activated during fetches from internal Program Memory. PSEN can sink/source 8 LS TTL inputs. It can drive CMOS inputs without an external pullup.

$\overline{\text{EA}}$

When EA is held high, the CPU executes out of internal Program Memory (unless the Program Counter exceeds 1 FFFH). When EA is held low, the CPU executes only out of external Program Memory. EA must not be floated.

XTAL1

Input to the inverting amplifier that forms the oscillator. Receives the external oscillator signal when an external oscillator is used.

XTAL2

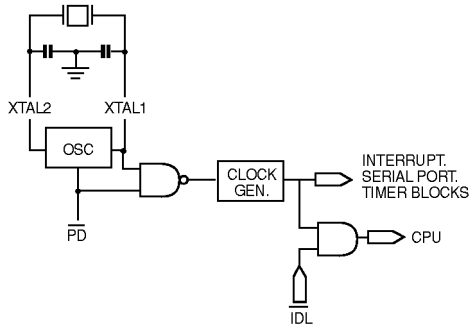
Output of the inverting amplifier that forms the oscillator. This pin should be floated when an external oscillator is used.

Idle And Power Down Operation

Figure 3 shows the internal Idle and Power Down clock configuration. As illustrated, Power Down operation stops the oscillator. Idle mode operation allows the interrupt, serial port, and timer blocks to continue to function, while the clock to the CPU is gated off.

These special modes are activated by software via the Special Function Register, PCON. Its hardware address is 87H. PCON is not bit addressable.

Figure 3.Idle and Power Down Hardware.



PCON : Power Control Register

(MSB) (LSB)

SMOD	–	–	–	GF1	GF0	PD	IDL

Symbol	Position	Name and Function
SMOD	PCON.7	Double Baud rate bit. When set to a 1, the baud rate is doubled when the serial port is being used in either modes 1, 2 or 3.
–	PCON.6	(Reserved)
–	PCON.5	(Reserved)
–	PCON.4	(Reserved)
GF1	PCON.3	General-purpose flag bit.
GF0	PCON.2	General-purpose flag bit.
PD	PCON.1	Power Down bit. Setting this bit activates power down operation.
IDL	PCON.0	Idle mode bit. Setting this bit activates idle mode operation.

If 1's are written to PD and IDL at the same time. PD takes, precedence. The reset value of PCON is (000X0000).

Idle Mode

The instruction that sets PCON.0 is the last instruction executed before the Idle mode is activated. Once in the Idle mode the CPU status is preserved in its entirety : the Stack Pointer, Program Counter, Program Status Word, Accumulator, RAM and all other registers maintain their data during idle. *Table 1* describes the status of the external pins during Idle mode.

There are three ways to terminate the Idle mode. Activation of any enabled interrupt will cause PCON.0 to be cleared by hardware, terminating Idle mode. The interrupt is serviced, and following RETI, the next instruction to be executed will be the one following the instruction that wrote 1 to PCON.0.

The flag bits GF0 and GF1 may be used to determine whether the interrupt was received during normal execution or during the Idle mode. For example, the instruction that writes to PCON.0 can also set or clear one or both flag bits. When Idle mode is terminated by an enabled interrupt, the service routine can examine the status of the flag bits.

The second way of terminating the Idle mode is with a hardware reset. Since the oscillator is still running, the hardware reset needs to be active for only 2 machine cycles (24 oscillator periods) to complete the reset operation.

Power Down Mode

The instruction that sets PCON.1 is the last executed prior to entering power down. Once in power down, the oscillator is stopped. The contents of the onchip RAM and the Special Function Register is saved during power down mode. The hardware reset initiates the Special Fucntion Register. In the Power Down mode, VCC may be lowered to minimize circuit power consumption. Care must be taken to ensure the voltage is not reduced until the power down mode is entered, and that the voltage is restored before the hardware reset is applied which freezes the oscillator. Reset should not be released until the oscillator has restarted and stabilized.

Table 1 describes the status of the external pins while in the power down mode. It should be noted that if the power down mode is activated while in external program memory, the port data that is held in the Special Function Register P2 is restored to Port 2. If the data is a 1, the port pin is held high during the power down mode by the strong pullup, T1, shown in *Figure 4*.

Table 1. Status of the external pins during idle and power down modes.

MODE	PROGRAM MEMORY	ALE	$\overline{\text{PSEN}}$	PORT0	PORT1	PORT2	PORT3
Idle	Internal	1	1	Port Data	Port Data	Port Data	Port Data
Idle	External	1	1	Floating	Port Data	Address	Port Data
Power Down	Internal	0	0	Port Data	Port Data	Port Data	Port Data
Power Down	External	0	0	Floating	Port Data	Port Data	Port Data

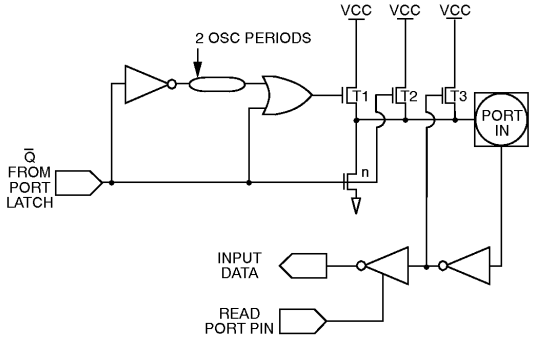
Stop Clock Mode

Due to static design, the TEMIC 80C32/C52 clock speed can be reduced until 0 MHz without any data loss in memory or registers. This mode allows step by step utilization, and permits to reduce system power consumption by bringing the clock frequency down to any value. At 0 MHz, the power consumption is the same as in the Power Down Mode.

I/O Ports

The I/O buffers for Ports 1, 2 and 3 are implemented as shown in *figure 4*.

Figure 4.I/O Buffers in the 80C52 (Ports 1, 2, 3).



When the port latch contains a 0, all pFETS in figure 4 are off while the nFET is turned on. When the port latch makes a 0-to-1 transition, the nFET turns off. The strong pFET, T1, turns on for two oscillator periods, pulling the output high very rapidly. As the output line is drawn high, pFET T3 turns on through the inverter to supply the IOH source current. This inverter and T form a latch which holds the 1 and is supported by T2.

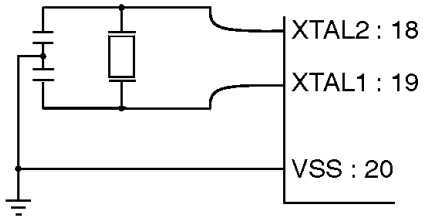
When Port 2 is used as an address port, for access to external program of data memory, any address bit that contains a 1 will have his strong pullup turned on for the entire duration of the external memory access.

When an I/O pin on Ports 1, 2, or 3 is used as an input, the user should be aware that the external circuit must sink current during the logical 1-to-0 transition. The maximum sink current is specified as ITL under the D.C. Specifications. When the input goes below approximately 2 V, T3 turns off to save ICC current. Note, when returning to a logical 1, T2 is the only internal pullup that is on. This will result in a slow rise time if the user's circuit does not force the input line high.

Oscillator Characteristics

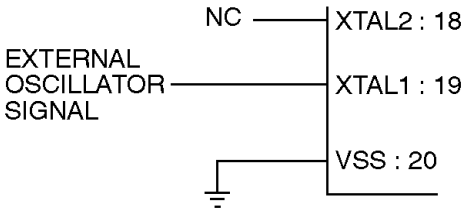
XTAL1 and XTAL2 are the input and output respectively, of an inverting amplifier which is configured for use as an on-chip oscillator, as shown in figure 5. Either a quartz crystal or ceramic resonator may be used.

Figure 5. Crystal Oscillator.



To drive the device from an external clock source, XTAL1 should be driven while XTAL2 is left unconnected as shown in *figure 6*. There are no requirements on the duty cycle of the external clock signal, since the input to the internal clocking circuitry is through a divide-by-two flip-flop, but minimum and maximum high and low times specified on the Data Sheet must be observed.

Figure 6. External Drive Configuration.



Hardware Description

Same as for the 80C51, plus a third timer/counter :

Timer/Event Counter 2

Timer 2 is a 16 bit timer/counter like Timers 0 and 1, it can operate either as a timer or as an event counter. This is selected by bit C/$\overline{T2}$ in the Special Function Register T2CON (Figure 1). It has three operating modes : "capture", "autoload" and "baud rate generator", which are selected by bits in T2CON as shown in *Table 2*.

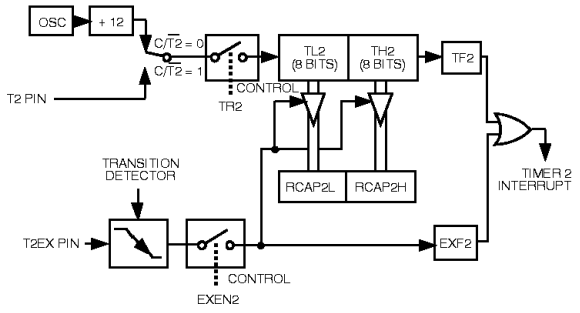
In the capture mode there are two options which are selected by bit EXEN2 in T2CON; If EXEN2 = 0, then Timer 2 is a 16 bit timer or counter which upon overflowing sets bit TF2, the Timer 2 overflow bit, which can be used to generate an interrupt. If EXEN2 = 1, then Timer 2 still does the above, but with the added feature that a 1-to-0 transition at external input T2EX causes the current value in the Timer 2 registers, TL2 and TH2, to be captured into registers RCAP2L and RCAP2H, respectively, (RCAP2L and RCAP2H are new Special Function Register in the 80C52). In addition, the transition at T2EX causes bit EXF2 in T2CON to be set, and EXF2, like TF2, can generate an interrupt.

Table 2. Timer 2 Operating Modes.

RCLK + TCLK	CP/$\overline{RL2}$	TR2	MODE
0	0	1	16 bit auto-reload
0	1	1	16 bit capture
1	X	1	baud rate generator
X	X	0	(off)

The capture mode is illustrated in *Figure 7*.

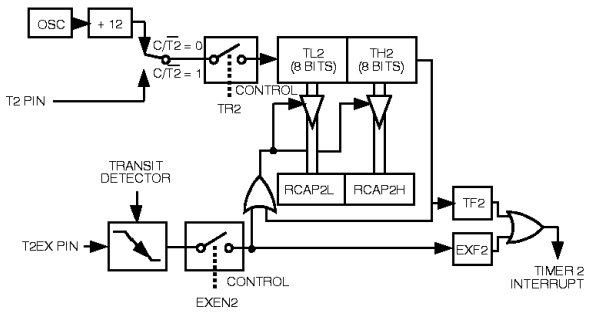
Figure 7. Timer 2 in Capture Mode.



In the auto-reload mode there are again two options, which are selected by bit EXEN2 in T2CON.If EXEN2 = 0, then when Timer 2 rolls over it does not only set TF2 but also causes the Timer 2 register to be reloaded with the 16 bit value in registers RCAP2L and RCAP2H, which are preset by software. If EXEN2 = 1, then Timer 2 still does the above, but with the added feature that a 1-to-0 transition at external input T2EX will also trigger the 16 bit reload and set EXF2.

The auto-reload mode is illustrated in *Figure 8.*

Figure 8. Timer in Auto-Reload Mode.



(MSB) (LSB)

TF2	EXF2	RCLK	TCLK	EXEN2	TR2	C/$\overline{T2}$	CP/$\overline{RL2}$

The baud rate generator mode is selected by : RCLK = 1 and/or TCLK = 1.

Symbol	Position	Name and Significance
TF2	T2CON.7	Timer 2 overflow flag set by a Timer 2 overflow and must be cleared by software. TF2 will not be set when either RCLK = 1 OR TCLK = 1.
EXF2	T2CON.6	Timer 2 external flag set when either a capture or reload is caused by a negative transition on T2EX and EXEN2 = 1. When Timer 2 interrupt is enabled, EXF2 = 1 will cause the CPU to vector to the Timer 2 interrupt routine. EXF2 must be cleared by software.
RCLK	T2CON.5	Receive clock flag. When set, causes the serial port to use Timer2 overflow pulses for its receive clock in modes 1 and 3. RCLK = 0 causes Timer 1 overflow to be used for the receive clock.
TCLK	T2CON.4	Transmit clock flag. When set, causes the serial port to use Timer 2 overflow pulses for its transmit clock in modes 1 and 3. TCLK = 0 causes Timer 1 overflows to be used for the transmit clock.
EXEN2	T2CON.3	Timer 2 external enable flag. When set, allows capture or reload to occur as a result of a negative transition on T2EX if Timer 2 is not being used to clock the serial port. EXEN2 = 0 causes Timer 2 to ignore events at T2EX.
TR2	T2CON.2	Start/stop control for Timer 2. A logic 1 starts the timer.
C/$\overline{T2}$	T2CON.1	Timer or counter select. (Timer 2) 0 = Internal timer (OSC/12) 1 = External event counter (falling edge triggered).
CP/$\overline{RL2}$	T2CON.0	Capture/Reload flag. When set, captures will occur on negative transitions at T2EX if EXEN 2 = 1. When cleared, auto reloads will occur either with Timer 2 overflows or negative transition at T2EX when EXEN2 = 1. When either RCLK = 1 or TCLK = 1, this bit is ignored and the timer is forced to auto-reload on Timer 2 overflow.

80C52 with Secret ROM

TEMIC offers 80C52 with the encrypted secret ROM option to secure the ROM code contained in the 80C52 microcontrollers.

The clear reading of the program contained in the ROM is made impossible due to an encryption through several random keys implemented during the manufacturing process.

The keys used to do such encryption are selected randomwise and are definitely different from one microcontroller to another.

This encryption is activated during the following phases :

– Everytime a byte is addressed during a verify of the ROM content, a byte of the encryption array is selected.
– MOVC instructions executed from external program memory are disabled when fetching code bytes from internal memory.
– EA is sampled and latched on reset, thus all state modification are disabled.

For further information please refer to the application note (ANM053) available upon request.

80C52 with Secret TAG

TEMIC offers special 64-bit identifier called “SECRET TAG” on the microcontroller chip.

The Secret Tag option is available on both ROMless and masked microcontrollers.

The Secret Tag feature allows serialization of each microcontroller for identification of a specific equipment. A unique number per device is implemented in the chip during manufacturing process. The serial number is a 64-bit binary value which is contained and addressable in the Special Function Registers (SFR) area.

This Secret Tag option can be read-out by a software routine and thus enables the user to do an individual identity check per device. This routine is implemented inside the microcontroller ROM memory in case of masked version which can be kept secret (and then the value of the Secret Tag also) by using a ROM Encryption.

For further information, please refer to the application note (ANM031) available upon request.

Electrical Characteristics

Absolute Maximum Ratings*

Ambiant Temperature Under Bias :
C = commercial ..... 0˚C to 70˚C
I = industrial ..... –40˚C to 85˚C
Storage Temperature ..... –65˚C to + 150˚C
Voltage on VCC to VSS ..... –0.5 V to + 7 V
Voltage on Any Pin to VSS ..... –0.5 V to V_{CC} + 0.5 V
Power Dissipation ..... 1 W
* This value is based on the maximum allowable die temperature and the thermal resistance of the package

**** Notice***

Stresses at or above those listed under " Absolute Maximum Ratings" may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or any other conditions above those indicated in the operational sections of this specification is not implied. Exposure to absolute maximum rating conditions may affect device reliability.

DC Parameters

TA = 0°C to 70°C ; VSS = 0 V ; VCC = 5 V ± 10 % ; F = 0 to 44 MHz
TA = –40°C + 85°C ; VSS = 0 V ; VCC = 5 V ± 10 % ; F = 0 to 36 MHz

SYMBOL	PARAMETER	MIN	MAX	UNIT	TEST CONDITIONS
VIL	Input Low Voltage	– 0.5	0.2 Vcc – 0.1	V	
VIH	Input High Voltage (Except XTAL and RST)	0.2 Vcc + 1.4	Vcc + 0.5	V	
VIH1	Input High Voltage (for XTAL and RST)	0.7 Vcc	Vcc + 0.5	V	
VOL	Output Low Voltage (Port 1, 2 and 3)		0.3 0.45 1.0	V V V	IOL = 100 µA IOL = 1.6 mA (note 2) IOL = 3.5 mA
VOL1	Output Low Voltage (Port 0, ALE, $\overline{PSEN}$)		0.3 0.45 1.0	V V V	IOL = 200 µA IOL = 3.2 mA (note 2) IOL = 7.0 mA
VOH	Output High Voltage Port 1, 2, 3	Vcc – 0.3		V	IOH = – 10 µA
		Vcc – 0.7		V	IOH = – 30 µA
		Vcc – 1.5		V	IOH = – 60 µA VCC = 5 V ± 10 %
VOH1	Output High Voltage (Port 0, ALE, $\overline{PSEN}$)	Vcc – 0.3		V	IOH = – 200 µA
		Vcc – 0.7		V	IOH = – 3.2 mA
		Vcc – 1.5		V	IOH = – 7.0 mA VCC = 5 V ± 10 %
IIL	Logical 0 Input Current (Ports 1, 2 and 3)		– 50	µA	Vin = 0.45 V
ILI	Input leakage Current		± 10	µA	0.45 < Vin < Vcc
ITL	Logical 1 to 0 Transition Current (Ports 1, 2 and 3)		– 650	µA	Vin = 2.0 V
IPD	Power Down Current		50	µA	Vcc = 2.0 V to 5.5 V (note 1)
RRST	RST Pulldown Resistor	50	200	KOhm	
CIO	Capacitance of I/O Buffer		10	pF	fc = 1 MHz, Ta = 25°C
ICC	Power Supply Current Freq = 1 MHz Icc op Icc idle Freq = 6 MHz Icc op Icc idle Freq ≥ 12 MHz Icc op = 1.25 Freq (MHz) + 5 mA Icc idle = 0.36 Freq (MHz) + 2.7 mA		 1.8 1 10 4	 mA mA mA mA	Vcc = 5.5 V

Absolute Maximum Ratings*

Ambient Temperature Under Bias :
A = Automotive . . . . . –40˚C to +125˚C
Storage Temperature . . . . . –65˚C to + 150˚C
Voltage on VCC to VSS . . . . . –0.5 V to + 7 V
Voltage on Any Pin to VSS . . . . . –0.5 V to VCC + 0.5 V
Power Dissipation . . . . . 1 W
* This value is based on the maximum allowable die temperature and the thermal resistance of the package

**** Notice***

Stresses above those listed under " Absolute Maximum Ratings" may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or any other conditions above those indicated in the operational sections of this specification is not implied. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.

DC Parameters

TA = –40°C + 125°C ; VSS = 0 V ; VCC = 5 V ± 10 % ; F = 0 to 36 MHz

SYMBOL	PARAMETER	MIN	MAX	UNIT	TEST CONDITIONS
VIL	Input Low Voltage	– 0.5	0.2 Vcc – 0.1	V	
VIH	Input High Voltage (Except XTAL and RST)	0.2 Vcc + 1.4	Vcc + 0.5	V	
VIH1	Input High Voltage (for XTAL and RST)	0.7 Vcc	Vcc + 0.5	V	
VOL	Output Low Voltage (Port 1, 2 and 3)		0.3 0.45 1.0	V V V	IOL = 100 µA IOL = 1.6 mA (note 2) IOL = 3.5 mA
VOL1	Output Low Voltage (Port 0, ALE, $\overline{PSEN}$)		0.3 0.45 1.0	V V V	IOL = 200 µA IOL = 3.2 mA (note 2) IOL = 7.0 mA
VOH	Output High Voltage Port 1, 2 and 3	Vcc – 0.3		V	IOH = – 10 µA
		Vcc – 0.7		V	IOH = – 30 µA
		Vcc – 1.5		V	IOH = – 60 µA VCC = 5 V ± 10 %
VOH1	Output High Voltage (Port 0, ALE, $\overline{PSEN}$)	Vcc – 0.3		V	IOH = – 200 µA
		Vcc – 0.7		V	IOH = – 3.2 mA
		Vcc – 1.5		V	IOH = – 7.0 mA VCC = 5 V ± 10 %
IIL	Logical 0 Input Current (Ports 1, 2 and 3)		– 75	µA	Vin = 0.45 V
ILI	Input leakage Current		±10	µA	0.45 < Vin < Vcc
ITL	Logical 1 to 0 Transition Current (Ports 1, 2 and 3)		– 750	µA	Vin = 2.0 V
IPD	Power Down Current		75	µA	Vcc = 2.0 V to 5.5 V (note 1)
RRST	RST Pulldown Resistor	50	200	KOhm	
CIO	Capacitance of I/O Buffer		10	pF	fc = 1 MHz, Ta = 25°C
ICC	Power Supply Current Freq = 1 MHz Icc op Icc idle Freq = 6 MHz Icc op Icc idle Freq ≥ 12 MHz Icc op = 1.25 Freq (MHz) + 5 mA Icc idle = 0.36 Freq (MHz) + 2.7 mA		 1.8 1 10 4	 mA mA mA mA	Vcc = 5.5 V

Absolute Maximum Ratings*

Ambient Temperature Under Bias :
M = Military . . . . . –55˚C to +125˚C
Storage Temperature . . . . . –65˚C to + 150˚C
Voltage on VCC to VSS . . . . . –0.5 V to + 7 V
Voltage on Any Pin to VSS . . . . . –0.5 V to VCC + 0.5 V
Power Dissipation . . . . . 1 W
* This value is based on the maximum allowable die temperature and the thermal resistance of the package

**** Notice***

Stresses at or above those listed under " Absolute Maximum Ratings" may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or any other conditions above those indicated in the operational sections of this specification is not implied. Exposure to absolute maximum rating conditions may affect device reliability.

DC Parameters

TA = –55°C + 125°C ; Vss = 0 V ; Vcc = 5 V ± 10 % ; F = 0 to 36 MHz

SYMBOL	PARAMETER	MIN	MAX	UNIT	TEST CONDITIONS
VIL	Input Low Voltage	– 0.5	0.2 Vcc – 0.1	V	
VIH	Input High Voltage (Except XTAL and RST)	0.2 Vcc + 1.4	Vcc + 0.5	V	
VIH1	Input High Voltage (for XTAL and RST)	0.7 Vcc	Vcc + 0.5	V	
VOL	Output Low Voltage (Port 1, 2 and 3)		0.45	V	IOL = 1.6 mA (note 2)
VOL1	Output Low Voltage (Port 0, ALE, PSEN)		0.45	V	IOL = 3.2 mA (note 2)
VOH	Output High Voltage (Port 1, 2 and 3)	2.4		V	IOH = – 60 µA Vcc = 5 V ± 10 %
		0.75 Vcc		V	IOH = – 25 µA
		0.9 Vcc		V	IOH = – 10 µA
VOH1	Output High Voltage (Port 0 in External Bus Mode, ALE, PEN)	2.4		V	IOH = – 400 µA Vcc = 5 V ± 10 %
		0.75 Vcc		V	IOH = – 150 µA
		0.9 Vcc		V	IOH = – 40 µA
IIL	Logical 0 Input Current (Ports 1, 2 and 3)		– 75	µA	Vin = 0.45 V
ILI	Input leakage Current		+/– 10	µA	0.45 < Vin < Vcc
ITL	Logical 1 to 0 Transition Current (Ports 1, 2 and 3)		– 750	µA	Vin = 2.0 V
IPD	Power Down Current		75	µA	Vcc = 2.0 V to 5.5 V (note 1)
RRST	RST Pulldown Resistor	50	200	KΩ	
CIO	Capacitance of I/O Buffer		10	pF	fc = 1 MHz, Ta = 25°C
ICC	Power Supply Current Freq = 1 MHz Icc op Icc idle Freq = 6 MHz Icc op Icc idle Freq ≥ 12 MHz Icc op = 1.25 Freq (MHz) + 5 mA Icc idle = 0.36 Freq (MHz) + 2.7 mA		 1.8 1 10 4	 mA mA mA mA	Vcc = 5.5 V

Absolute Maximum Ratings*

Ambient Temperature Under Bias :
C = Commercial . . . . . 0˚C to 70˚C
I = Industrial . . . . . −40 to 85˚C
Storage Temperature . . . . . –65˚C to + 150˚C
Voltage on VCC to VSS . . . . . –0.5 V to + 7 V
Voltage on Any Pin to VSS . . . . . –0.5 V to VCC + 0.5 V
Power Dissipation . . . . . 1 W**
** This value is based on the maximum allowable die temperature and the thermal resistance of the package

**** Notice***

Stresses at or above those listed under " Absolute Maximum Ratings" may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or any other conditions above those indicated in the operational sections of this specification is not implied. Exposure to absolute maximum rating conditions may affect device reliability.

DC Characteristics

TA = 0°C to 70°C ; Vcc = 2.7 V to 5.5 V ; Vss = 0 V ; F = 0 to 16 MHz
TA = –40°C to 85°C ; Vcc = 2.7 V to 5.5 V

SYMBOL	PARAMETER	MIN	MAX	UNIT	TEST CONDITIONS
VIL	Input Low Voltage	– 0.5	0.2 V_{CC} – 0.1	V	
VIH	Input High Voltage (Except XTAL and RST)	0.2 V_{CC} + 1.4	V_{CC} + 0.5	V	
VIH2	Input High Voltage to RST for Reset	0.7 V_{CC}	V_{CC} + 0.5	V	
VIH1	Input High Voltage to XTAL1	0.7 V_{CC}	V_{CC} + 0.5	V	
VPD	Power Down Voltage to Vcc in PD Mode	2.0	5.5	V	
VOL	Output Low Voltage (Ports 1, 2, 3)		0.45	V	IOL = 0.8 mA (note 2)
VOL1	Output Low Voltage Port 0, ALE, $\overline{PSEN}$		0.45	V	IOL = 1.6 mA (note 2)
VOH	Output High Voltage Ports 1, 2, 3	0.9 Vcc		V	IOH = – 10 µA
VOH1	Output High Voltage (Port 0 in External Bus Mode), ALE, $\overline{PSEN}$	0.9 Vcc		V	IOH = – 40 µA
IIL	Logical 0 Input Current Ports 1, 2, 3		– 50	µA	Vin = 0.45 V
ILI	Input Leakage Current		± 10	µA	0.45 < Vin < V_{CC}
ITL	Logical 1 to 0 Transition Current (Ports 1, 2, 3)		– 650	µA	Vin = 2.0 V
IPD	Power Down Current		50	µA	V_{CC} = 2.0 V to 5.5 V (note 1)
RRST	RST Pulldown Resistor	50	200	kΩ	
CIO	Capacitance of I/O Buffer		10	pF	fc = 1 MHz, T_A = 25°C

Maximum Icc (mA)

	OPERATING (NOTE 1)				IDLE (NOTE 1)			
FREQUENCY/Vcc	2.7 V	3 V	3.3 V	5.5 V	2.7 V	3 V	3.3 V	5.5 V
1 MHz	0.8 mA	1 mA	1.1 mA	1.8 mA	400 µA	500 µA	600 µA	1 mA
6 MHz	4 mA	5 mA	6 mA	10 mA	1.5 mA	1.7 mA	2 mA	4 mA
12 MHz	8 mA	10 mA	12 mA		2.5 mA	3 mA	3.5 mA	
16 MHz	10 mA	12 mA	14 mA		3 mA	3.8 mA	4.5 mA	
Freq > 12 MHz (Vcc = 5.5 V)	Icc (mA) = 1.25 × Freq (MHz) + 5 Icc Idle (mA) = 0.36 × Freq (MHz) + 2.7							

Note 1 : ICC is measured with all output pins disconnected ; XTAL1 driven with TCLCH, TCHCL = 5 ns, VIL = VSS + .5 V, VIH = VCC –.5 V ; XTAL2 N.C. ; EA = RST = Port 0 = VCC. ICC would be slighty higher if a crystal oscillator used.

Idle ICC is measured with all output pins disconnected ; XTAL1 driven with TCLCH, TCHCL = 5 ns, VIL = VSS + 5 V, VIH = VCC –.5 V ; XTAL2 N.C ; Port 0 = VCC ; EA = RST = VSS.

Power Down ICC is measured with all output pins disconnected ; EA = PORT 0 = VCC ; XTAL2 N.C. ; RST = VSS.

Note 2 : Capacitance loading on Ports 0 and 2 may cause spurious noise pulses to be superimposed on the VOLS of ALE and Ports 1 and 3. The noise is due to external bus capacitance discharging into the Port 0 and Port 2 pins when these pins make 1 to 0 transitions during bus operations. In the worst cases (capacitive loading 100 pF), the noise pulse on the ALE line may exceed 0.45 V may exceed 0,45 V with maxi VOL peak 0.6 V. A Schmitt Trigger use is not necessary.

Figure 9. ICC Test Condition, Idle Mode. All other pins are disconnected.

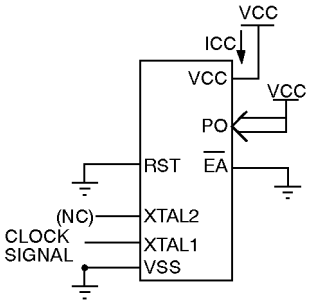


Figure 10. ICC Test Condition, Active Mode. All other pins are disconnected.

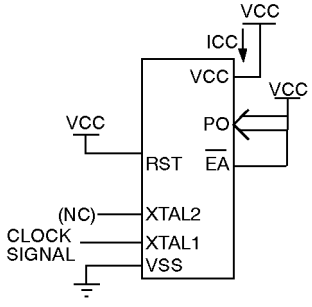


Figure 11. ICC Test Condition, Power Down Mode. All other pins are disconnected.

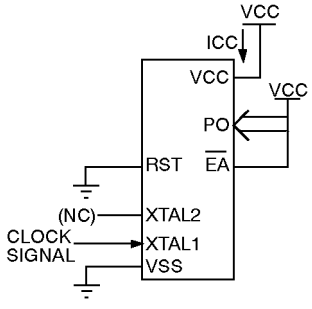
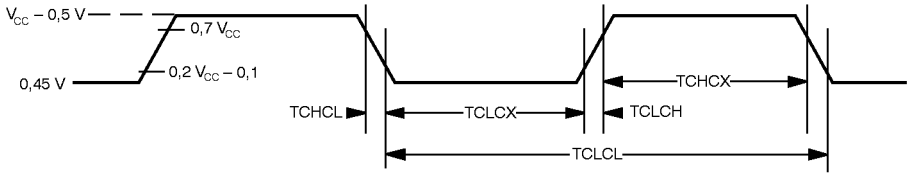


Figure 12. Clock Signal Waveform for ICC Tests in Active and Idle Modes. TCLCH = TCHCL = 5 ns.



Explanation of the AC Symbol

Each timing symbol has 5 characters. The first character is always a “T” (stands for time). The other characters, depending on their positions, stand for the name of a signal or the logical status of that signal. The following is a list of all the characters and what they stand for.

Example :

TAVLL = Time for Address Valid to ALE low.

TLLPL = Time for ALE low to $\overline{\text{PSEN}}$ low.

A : Address.	Q : Output data.
C : Clock.	R : READ signal.
D : Input data.	T : Time.
H : Logic level HIGH	V : Valid.
I : Instruction (program memory contents).	W : WRITE signal.
L : Logic level LOW, or ALE.	X : No longer a valid logic level.
P : PSEN.	Z : Float.

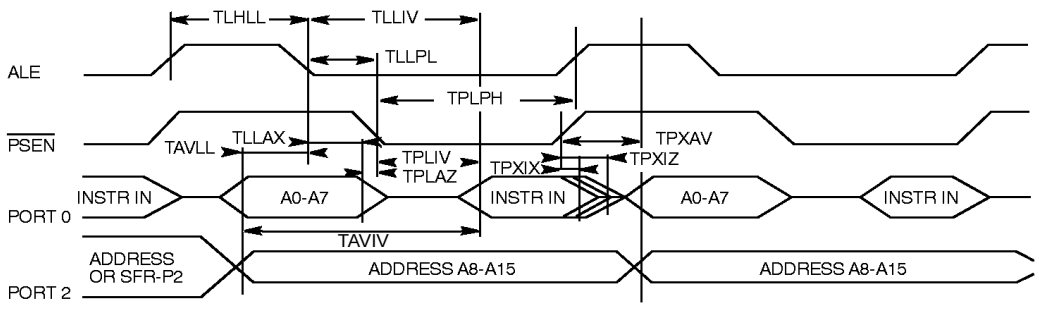
AC Parameters

TA = 0 to + 70°C ; Vss = 0 V ; Vcc = 5 V ± 10 % ; F = 0 to 44 MHz
TA = 0 to +70°C ; Vss = 0 V ; 2.7 V < Vcc < 5.5 V ; F = 0 to 16 MHz
TA = –40° to + 85°C ; Vss = 0 V ; 2.7 V < Vcc < 5.5 V ; F = 0 to 16 MHz
TA = –55° + 125°C ; Vss = 0 V ; Vcc = 5 V ± 10 % ; F = 0 to 36 MHz
(Load Capacitance for PORT 0, ALE and PSEN = 100 pF ; Load Capacitance for all other outputs = 80 pF)

External Program Memory Characteristics (values in ns)

		16 MHz		20 MHz		25 MHz		30 MHz		36 MHz		40 MHz		42 MHz		44 MHz	
SYM-BOL	**PARAMETER**	**min**	**max**	**min**	**max**	**min**	**max**	**min**	**max**	**min**	**max**	**min**	**max**	**min**	**max**	**min**	**max**
TLHLL	ALE Pulse Width	110		90		70		60		50		40		35		30	
TAVLL	Address valid to ALE	40		30		20		15		10		9		8		7	
TLLAX	Address Hold After ALE	35		35		35		35		35		30		25		17	
TLLIV	ALE to valid instr in		185		170		130		100		80		70		65		65
TLLPL	ALE to PSEN	45		40		30		25		20		15		13		12	
TPLPH	PSEN pulse Width	165		130		100		80		75		65		60		54	
TPLIV	PSEN to valid instr in		125		110		85		65		50		45		40		35
TPXIX	Input instr Hold After PSEN	0		0		0		0		0		0		0		0	
TPXIZ	Input instr Float After PSEN		50		45		35		30		25		20		15		10
TPXAV	PSEN to Address Valid	55		50		40		35		30		25		20		15	
TAVIV	Address to Valid instr in		230		210		170		130		90		80		75		70
TPLAZ	PSEN low to Address Float		10		10		8		6		5		5		5		5

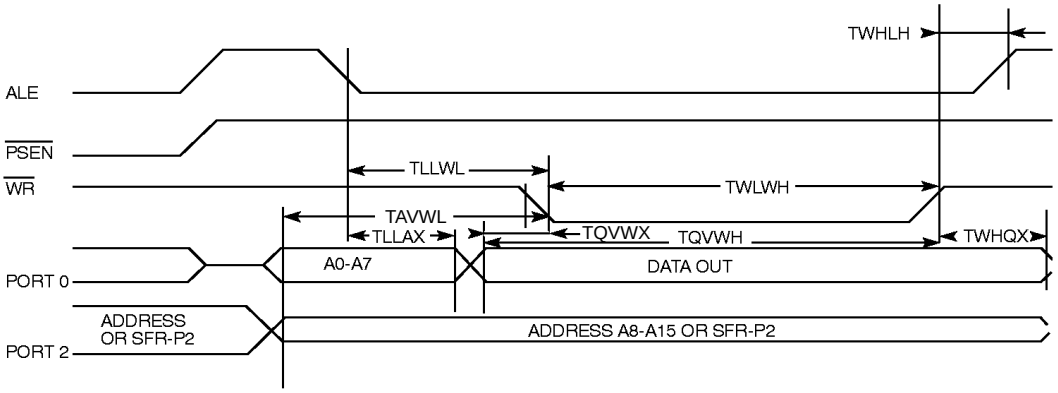
External Program Memory Read Cycle



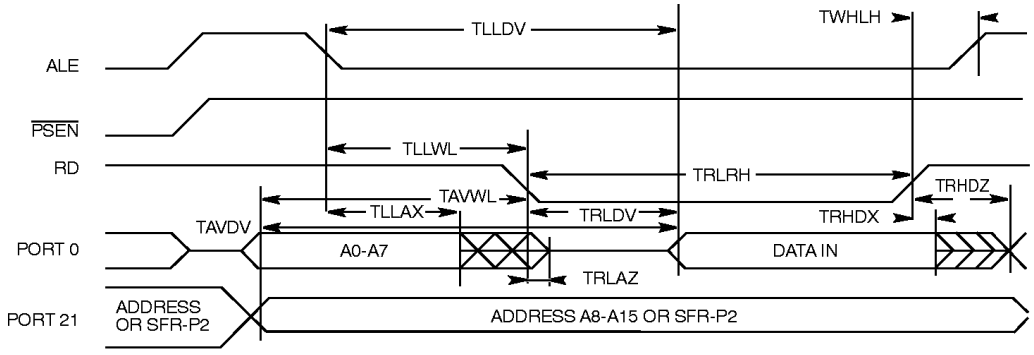
External Data Memory Characteristics (values in ns)

		16 MHz		20 MHz		25 MHz		30 MHz		36 MHz		40 MHz		42 MHz		44 MHz	
SYM-BOL	**PARAMETER**	**min**	**max**	**min**	**max**	**min**	**max**	**min**	**max**	**min**	**max**	**min**	**max**	**min**	**max**	**min**	**max**
TRLRH	RD pulse Width	340		270		210		180		120		100		90		80	
TWLWH	WR pulse Width	340		270		210		180		120		100		90		80	
TLLAX	Address Hold After ALE	85		85		70		55		35		30		25		25	
TRLDV	RD to Valid Data in		240		210		175		135		110		90		80		70
TRHDX	Data hold after RD	0		0		0		0		0		0		0		0	
TRHDZ	Data float after RD		90		90		80		70		50		45		40		35
TLLDV	ALE to Valid Data In		435		370		290		235		170		150		140		130
TAVDV	Address to Valid Data IN		480		400		320		260		190		180		175		170
TLLWL	ALE to WR or RD	150	250	135	170	120	130	90	115	70	100	60	95	55	90	50	85
TAVWL	Address to WR or RD	180		180		140		115		75		65		60		55	
TQVWX	Data valid to WR transition	35		35		30		20		15		10		8		6	
TQVWH	Data Setup to WR transition	380		325		250		215		170		160		150		140	
TWHQX	Data Hold after WR	40		35		30		20		15		10		8		6	
TRLAZ	RD low to Address Float		0		0		0		0		0		0		0		0
TWHLH	RD or WR high to ALE high	35	90	35	60	25	45	20	40	20	40	15	35	13	33	13	33

External Data Memory Write Cycle



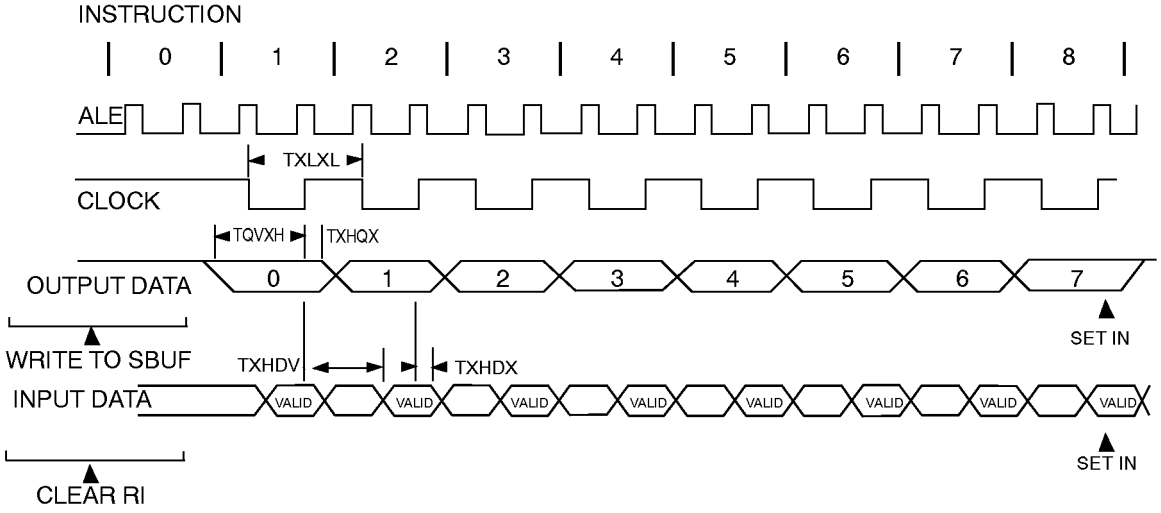
External Data Memory Read Cycle



Serial Port Timing – Shift Register Mode (values in ns)

		16 MHz		20 MHz		25 MHz		30 MHz		36 MHz		40 MHz		42 MHz		44 MHz	
SYM-BOL	**PARAMETER**	**min**	**max**	**min**	**max**	**min**	**max**	**min**	**max**	**min**	**max**	**min**	**max**	**min**	**max**	**min**	**max**
TXLXL	Serial Port Clock Cycle Time	750		600		480		400		330		250		230		227	
TQVXH	Output Data Setup to Clock Rising Edge	563		480		380		300		220		170		150		140	
TXHQX	Output Data Hold after Clock Rising Edge	63		90		65		50		45		35		30		25	
TXHDX	Input Data Hold after Clock Rising Edge	0		0		0		0		0		0		0		0	
TXHDV	Clock Rising Edge to Input Data Valid		563		450		350		300		250		200		180		160

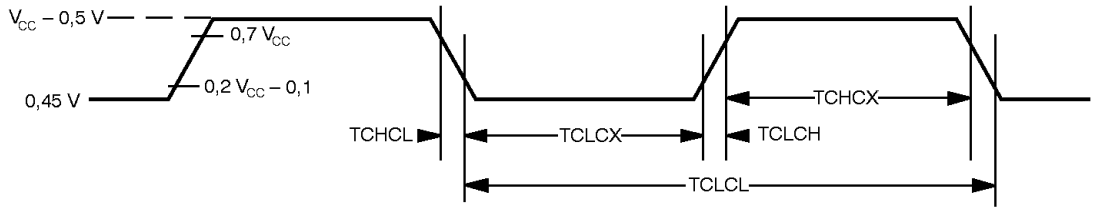
Shift Register Timing Waveforms



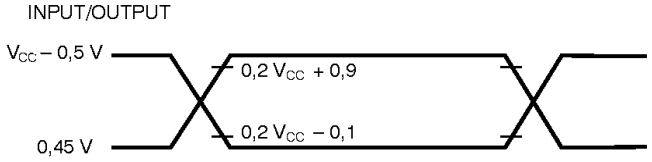
External Clock Drive Characteristics (XTAL1)

SYMBOL	PARAMETER	MIN	MAX	UNIT
FCLCL	Oscillator Frequency		44	MHz
TCLCL	Oscillator period	22.7		ns
TCHCX	High Time	5		ns
TCLCX	Low Time	5		ns
TCLCH	Rise Time		5	ns
TCHCL	Fall Time		5	ns

External Clock Drive Waveforms

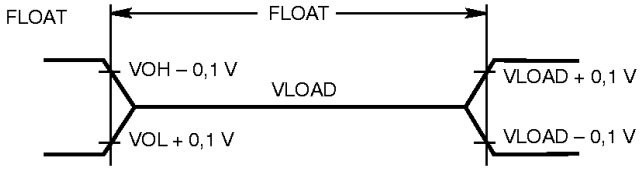


AC Testing Input/Output Waveforms



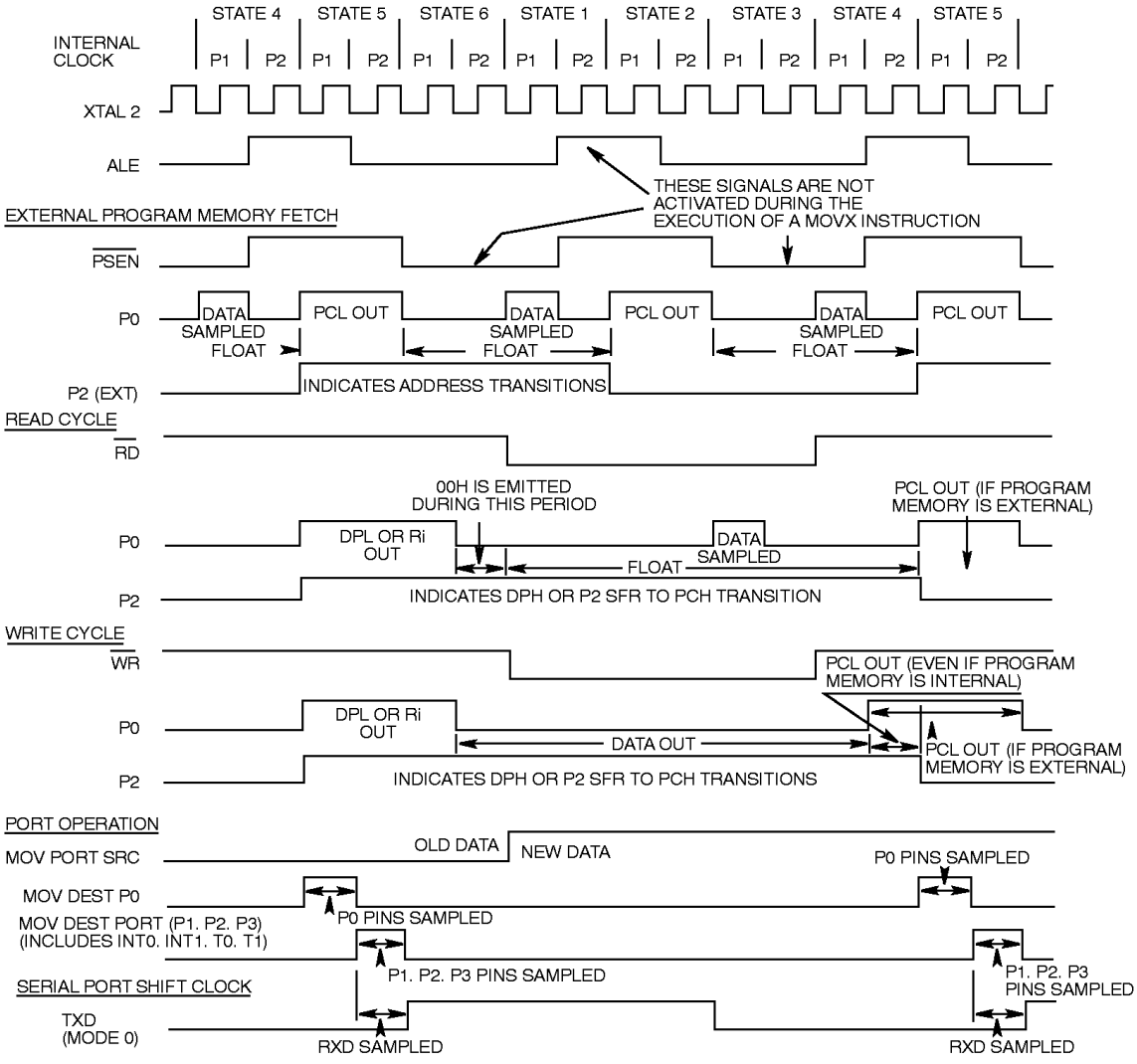
AC inputs during testing are driven at Vcc – 0.5 for a logic “1” and 0.45 V for a logic “0”. Timing measurements are made at VIH min for a logic “1” and VIL max for a logic “0”.

Float Waveforms



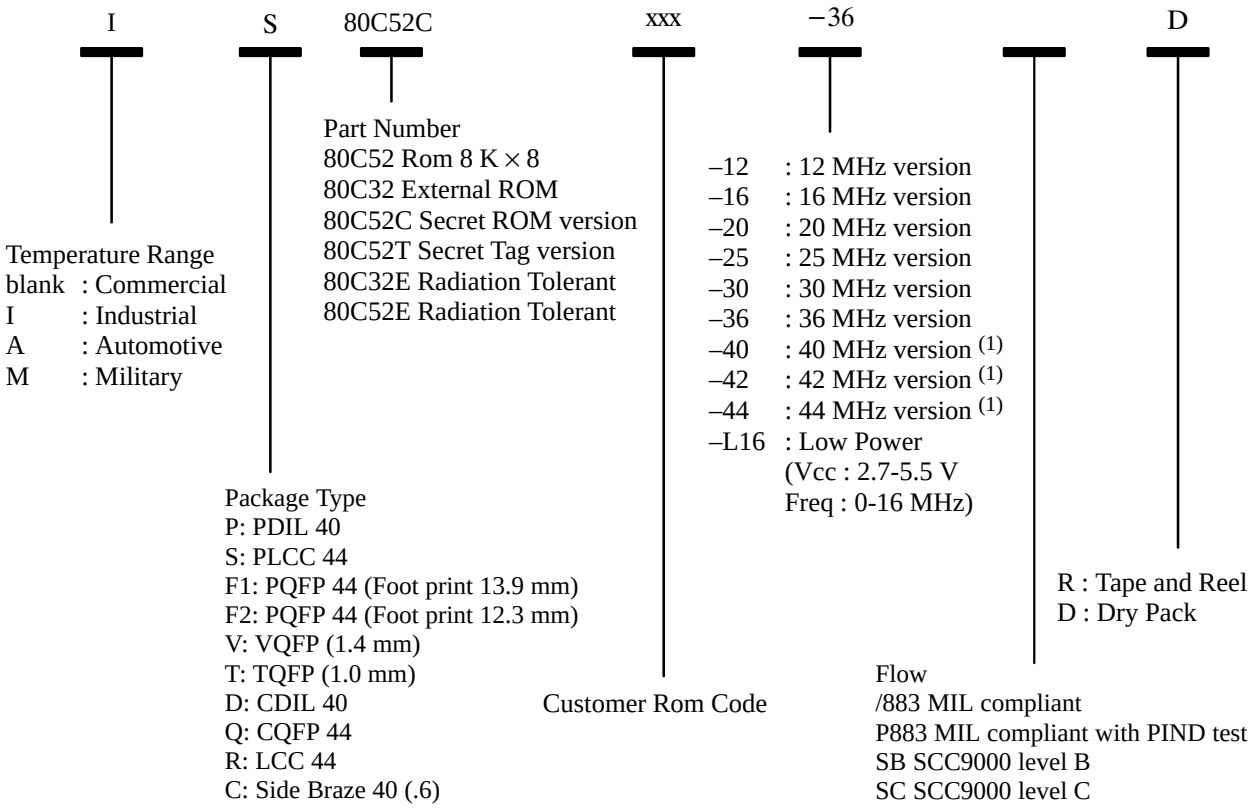
For timing purposes as port pin is no longer floating when a 100 mV change from load voltage occurs and begins to float when a 100 mV change from the loaded VOH/VOL level occurs. Iol/IoH ≥ ± 20 mA.

Clock Waveforms



This diagram indicates when signals are clocked internally. The time it takes the signals to propagate to the pins, however, ranges from 25 to 125 ns. This propagation delay is dependent on variables such as temperature and pin loading. Propagation also varies from output to output and component. Typically though (T_A = 25°C fully loaded) RD and WR propagation delays are approximately 50 ns. The other signals are typically 85 ns. Propagation delays are incorporated in the AC specifications.

Ordering Information



(1) Only for 80C31 at commercial range.

82C55A

CMOS Programmable Peripheral Interface

June 1998

Features

- **Pin Compatible with NMOS 8255A**
- **24 Programmable I/O Pins**
- **Fully TTL Compatible**
- **High Speed, No "Wait State" Operation with 5MHz and 8MHz 80C86 and 80C88**
- **Direct Bit Set/Reset Capability**
- **Enhanced Control Word Read Capability**
- **L7 Process**
- **2.5mA Drive Capability on All I/O Ports**
- **Low Standby Power (ICCSB) 10μA**

Ordering Information

PART NUMBERS		PACKAGE	TEMPERATURE RANGE	PKG. NO.
5MHz	8MHz			
CP82C55A-5	CP82C55A	40 Ld PDIP	0^oC to 70^oC	E40.6
IP82C55A-5	IP82C55A		-40^oC to 85^oC	E40.6
CS82C55A-5	CS82C55A	44 Ld PLCC	0^oC to 70^oC	N44.65
IS82C55A-5	IS82C55A		-40^oC to 85^oC	N44.65
CD82C55A-5	CD82C55A	40 Ld CERDIP	0^oC to 70^oC	F40.6
ID82C55A-5	ID82C55A		-40^oC to 85^oC	F40.6
MD82C55A-5/B	MD82C55A/B		-55^oC to 125^oC	F40.6
8406601QA	8406602QA	SMD#		F40.6
MR82C55A-5/B	MR82C55A/B	44 Pad CLCC	-55^oC to 125^oC	J44.A
8406601XA	8406602XA	SMD#		J44.A

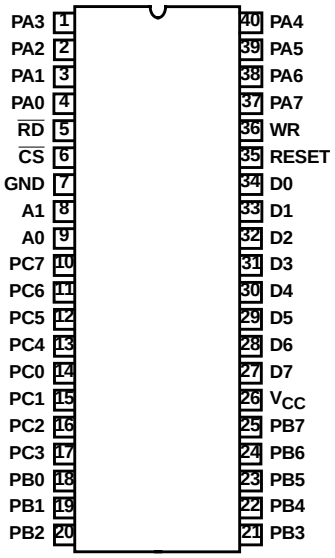
Description

The Intersil 82C55A is a high performance CMOS version of the industry standard 8255A and is manufactured using a self-aligned silicon gate CMOS process (Scaled SAJI IV). It is a general purpose programmable I/O device which may be used with many different microprocessors. There are 24 I/O pins which may be individually programmed in 2 groups of 12 and used in 3 major modes of operation. The high performance and industry standard configuration of the 82C55A make it compatible with the 80C86, 80C88 and other microprocessors.

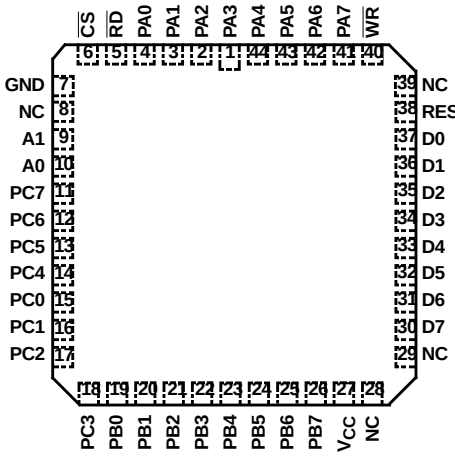
Static CMOS circuit design insures low operating power. TTL compatibility over the full military temperature range and bus hold circuitry eliminate the need for pull-up resistors. The Intersil advanced SAJI process results in performance equal to or greater than existing functionally equivalent products at a fraction of the power.

Pinouts

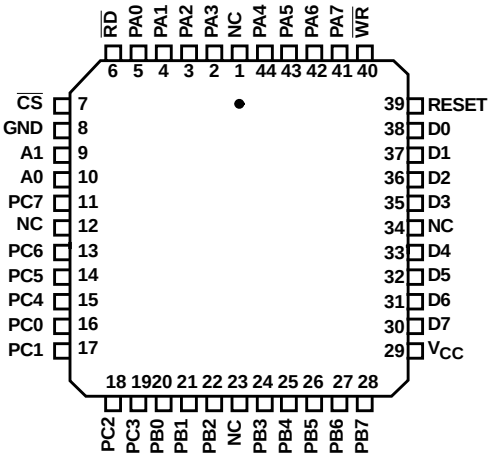
82C55A (DIP)
TOP VIEW



82C55A (CLCC)
TOP VIEW



82C55A (PLCC)
TOP VIEW

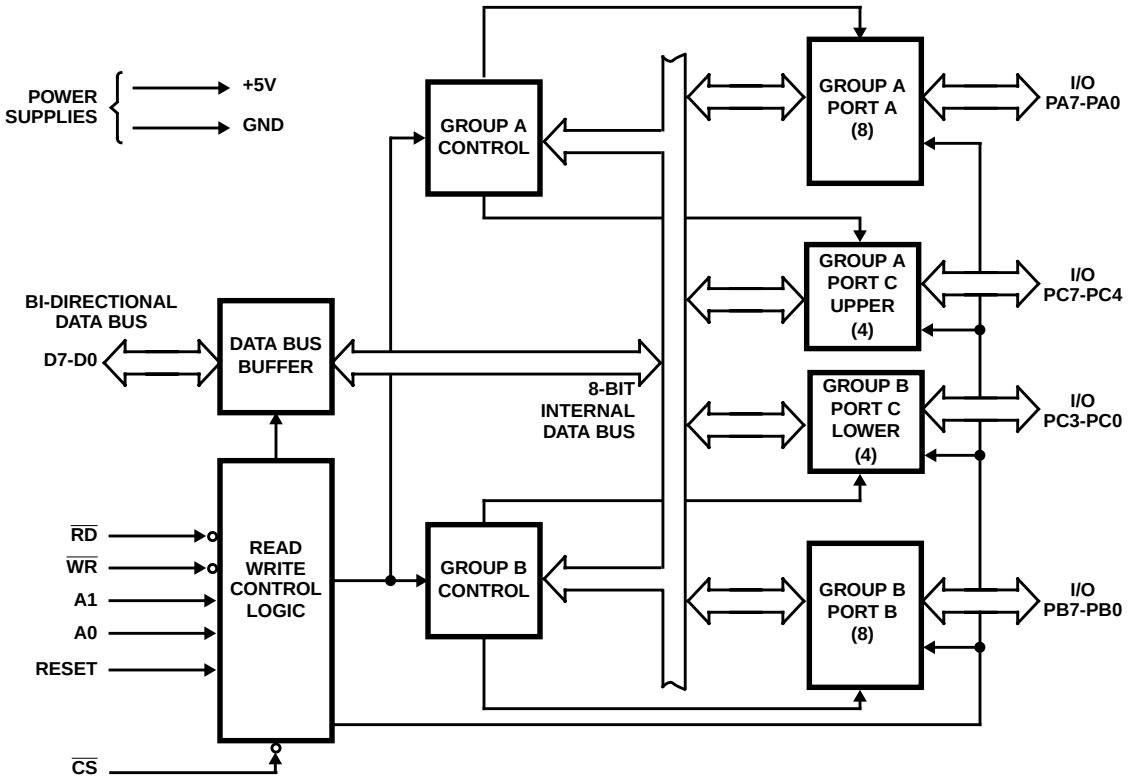


File Number **2969.2**

Pin Description

SYMBOL	PIN NUMBER	TYPE	DESCRIPTION
V_{CC}	26		V_{CC}: The +5V power supply pin. A 0.1µF capacitor between pins 26 and 7 is recommended for decoupling.
GND	7		GROUND
D0-D7	27-34	I/O	DATA BUS: The Data Bus lines are bidirectional three-state pins connected to the system data bus.
RESET	35	I	RESET: A high on this input clears the control register and all ports (A, B, C) are set to the input mode with the “Bus Hold” circuitry turned on.
$\overline{CS}$	6	I	CHIP SELECT: Chip select is an active low input used to enable the 82C55A onto the Data Bus for CPU communications.
$\overline{RD}$	5	I	READ: Read is an active low input control signal used by the CPU to read status information or data via the data bus.
$\overline{WR}$	36	I	WRITE: Write is an active low input control signal used by the CPU to load control words and data into the 82C55A.
A0-A1	8, 9	I	ADDRESS: These input signals, in conjunction with the $\overline{RD}$ and $\overline{WR}$ inputs, control the selection of one of the three ports or the control word register. A0 and A1 are normally connected to the least significant bits of the Address Bus A0, A1.
PA0-PA7	1-4, 37-40	I/O	PORT A: 8-bit input and output port. Both bus hold high and bus hold low circuitry are present on this port.
PB0-PB7	18-25	I/O	PORT B: 8-bit input and output port. Bus hold high circuitry is present on this port.
PC0-PC7	10-17	I/O	PORT C: 8-bit input and output port. Bus hold circuitry is present on this port.

Functional Diagram



Functional Description

Data Bus Buffer

This three-state bi-directional 8-bit buffer is used to interface the 82C55A to the system data bus. Data is transmitted or received by the buffer upon execution of input or output instructions by the CPU. Control words and status information are also transferred through the data bus buffer.

Read/Write and Control Logic

The function of this block is to manage all of the internal and external transfers of both Data and Control or Status words. It accepts inputs from the CPU Address and Control busses and in turn, issues commands to both of the Control Groups.

(CS) Chip Select. A “low” on this input pin enables the communcation between the 82C55A and the CPU.

(RD) Read. A “low” on this input pin enables 82C55A to send the data or status information to the CPU on the data bus. In essence, it allows the CPU to “read from” the 82C55A.

(WR) Write. A “low” on this input pin enables the CPU to write data or control words into the 82C55A.

(A0 and A1) Port Select 0 and Port Select 1. These input signals, in conjunction with the RD and WR inputs, control the selection of one of the three ports or the control word register. They are normally connected to the least significant bits of the address bus (A0 and A1).

82C55A BASIC OPERATION

A1	A0	RD	WR	CS	INPUT OPERATION (READ)
0	0	0	1	0	Port A → Data Bus
0	1	0	1	0	Port B → Data Bus
1	0	0	1	0	Port C → Data Bus
1	1	0	1	0	Control Word → Data Bus
					OUTPUT OPERATION (WRITE)
0	0	1	0	0	Data Bus → Port A
0	1	1	0	0	Data Bus → Port B
1	0	1	0	0	Data Bus → Port C
1	1	1	0	0	Data Bus → Control
					DISABLE FUNCTION
X	X	X	X	1	Data Bus → Three-State
X	X	1	1	0	Data Bus → Three-State

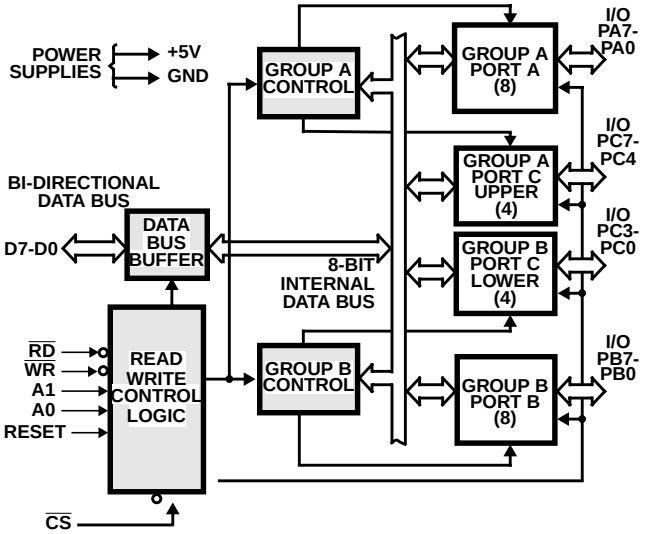


FIGURE 1. 82C55A BLOCK DIAGRAM. DATA BUS BUFFER, READ/WRITE, GROUP A & B CONTROL LOGIC FUNCTIONS

(RESET) Reset. A “high” on this input initializes the control register to 9Bh and all ports (A, B, C) are set to the input mode. “Bus hold” devices internal to the 82C55A will hold the I/O port inputs to a logic “1” state with a maximum hold current of 400µA.

Group A and Group B Controls

The functional configuration of each port is programmed by the systems software. In essence, the CPU “outputs” a control word to the 82C55A. The control word contains information such as “mode”, “bit set”, “bit reset”, etc., that initializes the functional configuration of the 82C55A.

Each of the Control blocks (Group A and Group B) accepts “commands” from the Read/Write Control logic, receives “control words” from the internal data bus and issues the proper commands to its associated ports.

Control Group A - Port A and Port C upper (C7 - C4)

Control Group B - Port B and Port C lower (C3 - C0)

The control word register can be both written and read as shown in the “Basic Operation” table. Figure 4 shows the control word format for both Read and Write operations. When the control word is read, bit D7 will always be a logic “1”, as this implies control word mode information.

Ports A, B, and C

The 82C55A contains three 8-bit ports (A, B, and C). All can be configured to a wide variety of functional characteristics by the system software but each has its own special features or "personality" to further enhance the power and flexibility of the 82C55A.

Port A One 8-bit data output latch/buffer and one 8-bit data input latch. Both "pull-up" and "pull-down" bus-hold devices are present on Port A. See Figure 2A.

Port B One 8-bit data input/output latch/buffer and one 8-bit data input buffer. See Figure 2B.

Port C One 8-bit data output latch/buffer and one 8-bit data input buffer (no latch for input). This port can be divided into two 4-bit ports under the mode control. Each 4-bit port contains a 4-bit latch and it can be used for the control signal output and status signal inputs in conjunction with ports A and B. See Figure 2B.

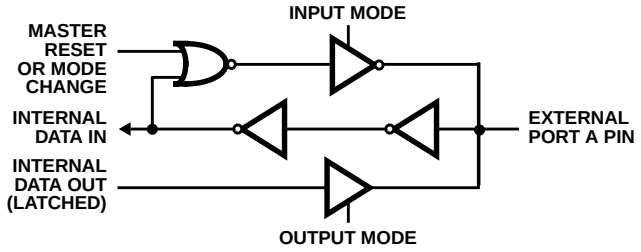


FIGURE 2A. PORT A BUS-HOLD CONFIGURATION

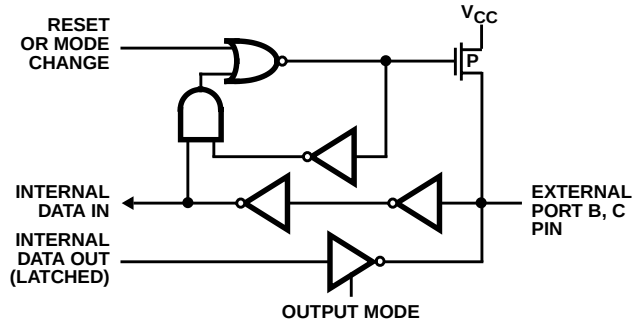


FIGURE 2B. PORT B AND C BUS-HOLD CONFIGURATION

FIGURE 2. BUS-HOLD CONFIGURATION

Operational Description

Mode Selection

There are three basic modes of operation than can be selected by the system software:

Mode 0 - Basic Input/Output
Mode 1 - Strobed Input/Output
Mode 2 - Bi-directional Bus

When the reset input goes "high", all ports will be set to the input mode with all 24 port lines held at a logic "one" level by internal bus hold devices. After the reset is removed, the 82C55A can remain in the input mode with no additional initialization required. This eliminates the need to pullup or pull-down resistors in all-CMOS designs. The control word register will contain 9Bh. During the execution of the system program, any of the other modes may be selected using a single output instruction. This allows a single 82C55A to service a variety of peripheral devices with a simple software maintenance routine. Any port programmed as an output port is initialized to all zeros when the control word is written.

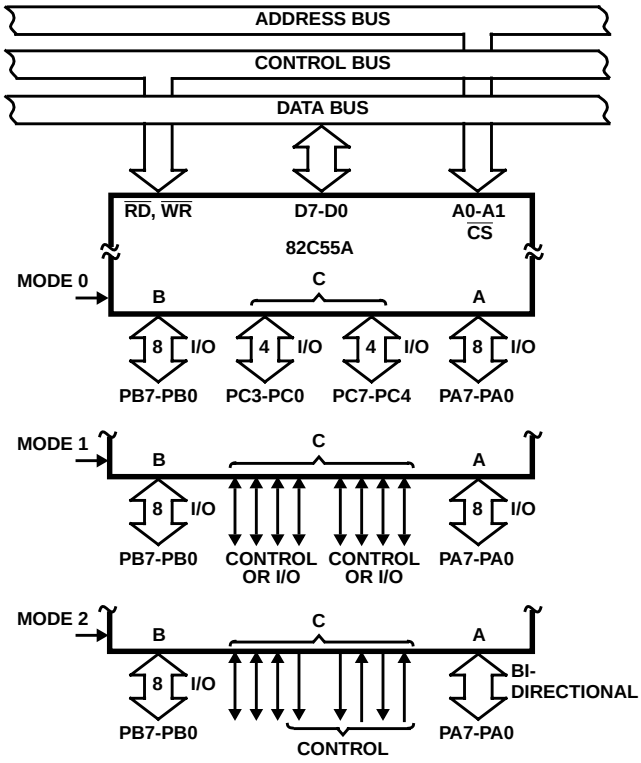


FIGURE 3. BASIC MODE DEFINITIONS AND BUS INTERFACE

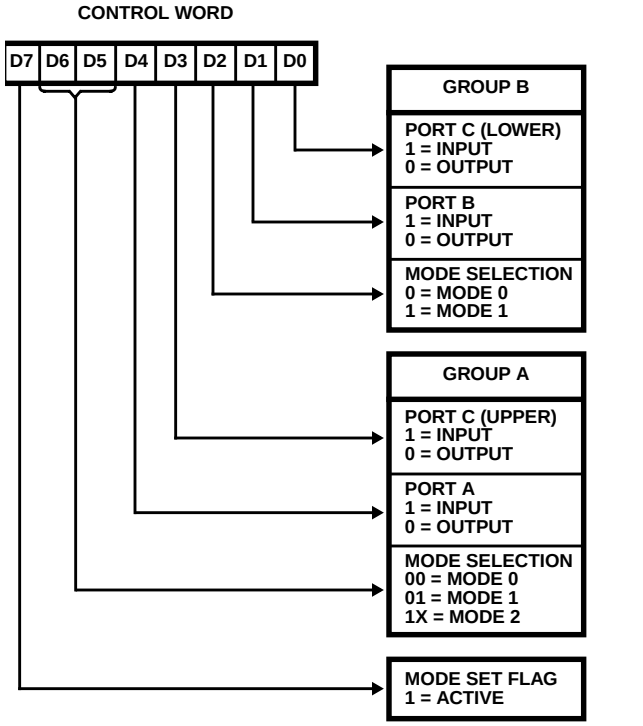


FIGURE 4. MODE DEFINITION FORMAT

The modes for Port A and Port B can be separately defined, while Port C is divided into two portions as required by the Port A and Port B definitions. All of the output registers, including the status flip-flops, will be reset whenever the mode is changed. Modes may be combined so that their functional definition can be “tailored” to almost any I/O structure. For instance: Group B can be programmed in Mode 0 to monitor simple switch closings or display computational results, Group A could be programmed in Mode 1 to monitor a keyboard or tape reader on an interrupt-driven basis.

The mode definitions and possible mode combinations may seem confusing at first, but after a cursory review of the complete device operation a simple, logical I/O approach will surface. The design of the 82C55A has taken into account things such as efficient PC board layout, control signal definition vs. PC layout and complete functional flexibility to support almost any peripheral device with no external logic. Such design represents the maximum use of the available pins.

Single Bit Set/Reset Feature (Figure 5)

Any of the eight bits of Port C can be Set or Reset using a single Output instruction. This feature reduces software requirements in control-based applications.

When Port C is being used as status/control for Port A or B, these bits can be set or reset by using the Bit Set/Reset operation just as if they were output ports.

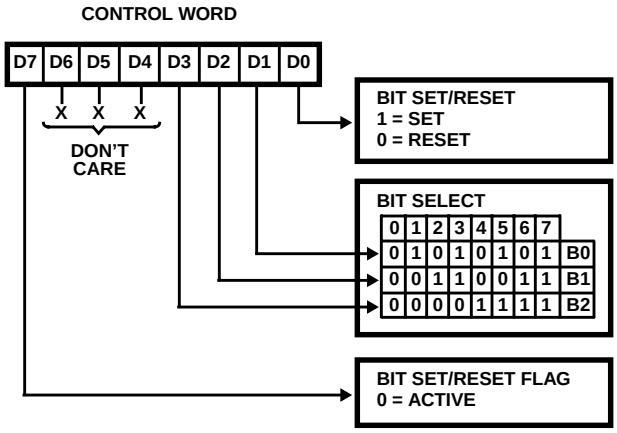


FIGURE 5. BIT SET/RESET FORMAT

Interrupt Control Functions

When the 82C55A is programmed to operate in mode 1 or mode 2, control signals are provided that can be used as interrupt request inputs to the CPU. The interrupt request signals, generated from port C, can be inhibited or enabled by setting or resetting the associated INTE flip-flop, using the bit set/reset function of port C.

This function allows the programmer to enable or disable a CPU interrupt by a specific I/O device without affecting any other device in the interrupt structure.

INTE Flip-Flop Definition

(BIT-SET)-INTE is SET - Interrupt Enable

(BIT-RESET)-INTE is Reset - Interrupt Disable

NOTE: All Mask flip-flops are automatically reset during mode selection and device Reset.

Operating Modes

Mode 0 (Basic Input/Output). This functional configuration provides simple input and output operations for each of the three ports. No handshaking is required, data is simply written to or read from a specific port.

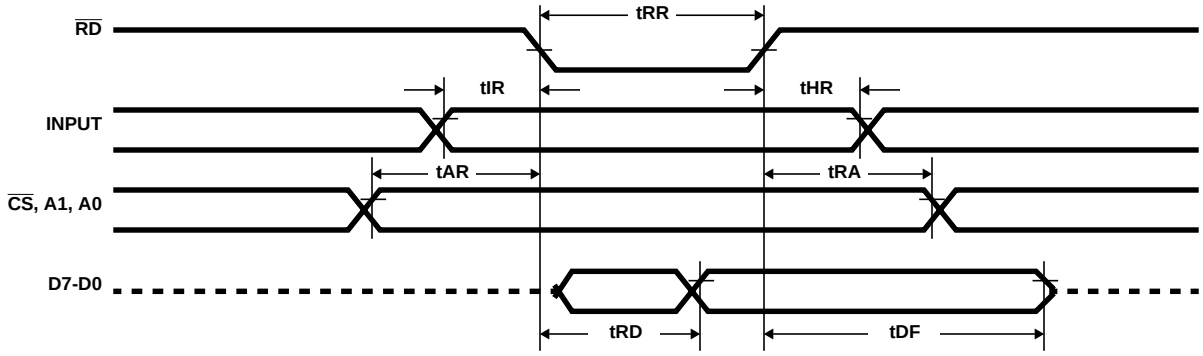
Mode 0 Basic Functional Definitions:

- Two 8-bit ports and two 4-bit ports
- Any Port can be input or output
- Outputs are latched
- Input are not latched
- 16 different Input/Output configurations possible

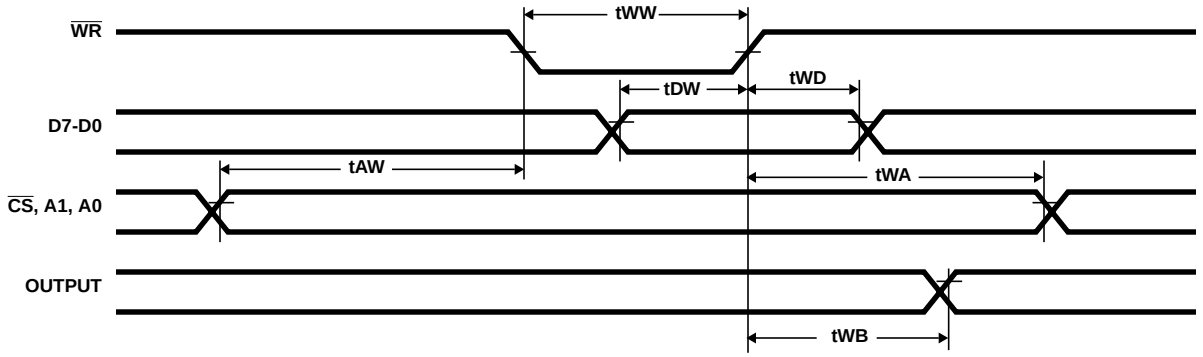
MODE 0 PORT DEFINITION

A		B		GROUP A			GROUP B	
D4	D3	D1	D0	PORT A	PORT C (Upper)	#	PORT B	PORT C (Lower)
0	0	0	0	Output	Output	0	Output	Output
0	0	0	1	Output	Output	1	Output	Input
0	0	1	0	Output	Output	2	Input	Output
0	0	1	1	Output	Output	3	Input	Input
0	1	0	0	Output	Input	4	Output	Output
0	1	0	1	Output	Input	5	Output	Input
0	1	1	0	Output	Input	6	Input	Output
0	1	1	1	Output	Input	7	Input	Input
1	0	0	0	Input	Output	8	Output	Output
1	0	0	1	Input	Output	9	Output	Input
1	0	1	0	Input	Output	10	Input	Output
1	0	1	1	Input	Output	11	Input	Input
1	1	0	0	Input	Input	12	Output	Output
1	1	0	1	Input	Input	13	Output	Input
1	1	1	0	Input	Input	14	Input	Output
1	1	1	1	Input	Input	15	Input	Input

Mode 0 (Basic Input)



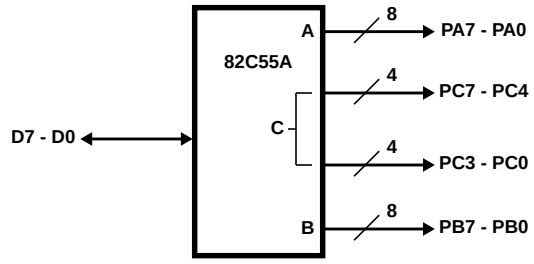
Mode 0 (Basic Output)



Mode 0 Configurations

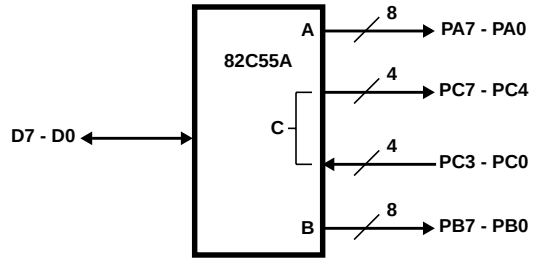
CONTROL WORD #0

D7	D6	D5	D4	D3	D2	D1	D0
1	0	0	0	0	0	0	0



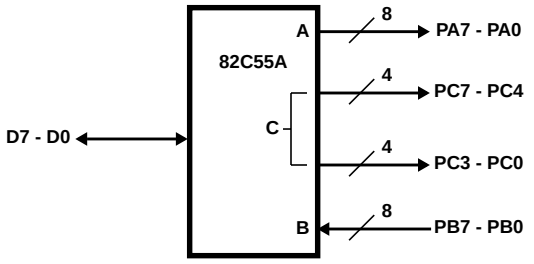
CONTROL WORD #1

D7	D6	D5	D4	D3	D2	D1	D0
1	0	0	0	0	0	0	1



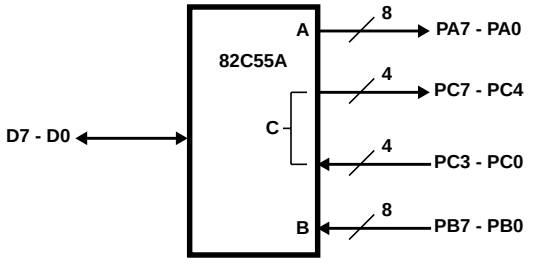
CONTROL WORD #2

D7	D6	D5	D4	D3	D2	D1	D0
1	0	0	0	0	0	1	0



CONTROL WORD #3

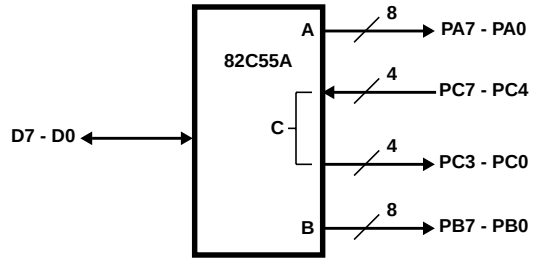
D7	D6	D5	D4	D3	D2	D1	D0
1	0	0	0	0	0	1	1



Mode 0 Configurations (Continued)

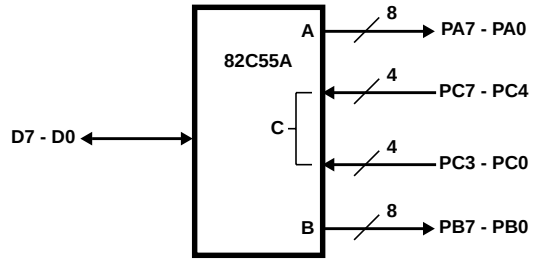
CONTROL WORD #4

D7	D6	D5	D4	D3	D2	D1	D0
1	0	0	0	1	0	0	0



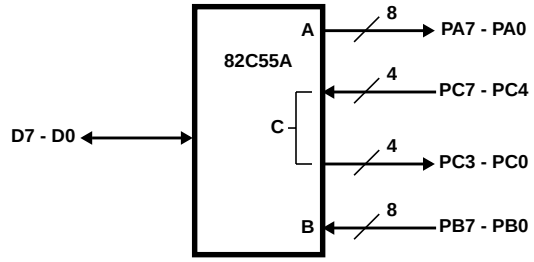
CONTROL WORD #5

D7	D6	D5	D4	D3	D2	D1	D0
1	0	0	0	1	0	0	1



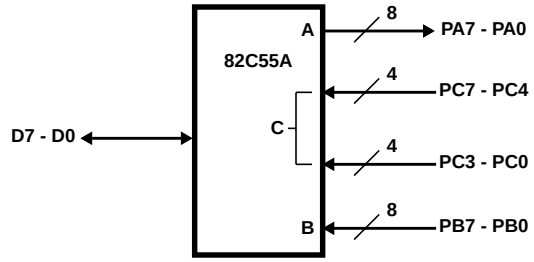
CONTROL WORD #6

D7	D6	D5	D4	D3	D2	D1	D0
1	0	0	0	1	0	1	0



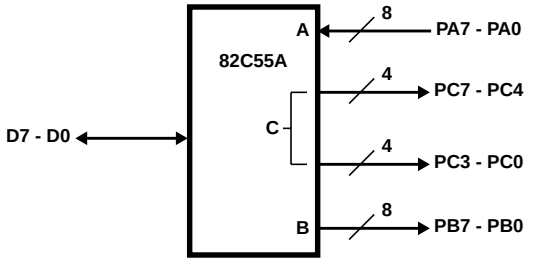
CONTROL WORD #7

D7	D6	D5	D4	D3	D2	D1	D0
1	0	0	0	1	0	1	1



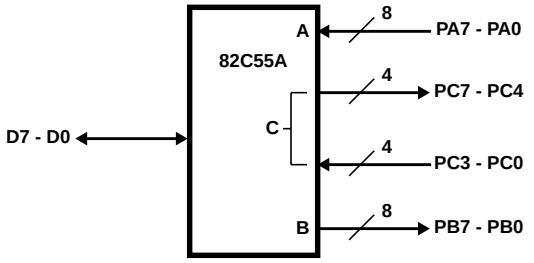
CONTROL WORD #8

D7	D6	D5	D4	D3	D2	D1	D0
1	0	0	1	0	0	0	0



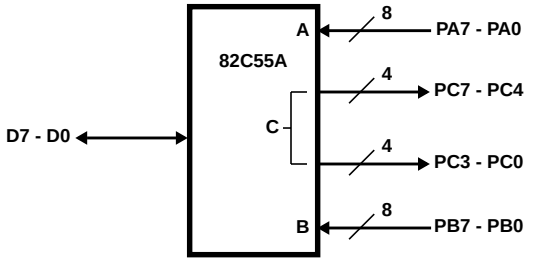
CONTROL WORD #9

D7	D6	D5	D4	D3	D2	D1	D0
1	0	0	1	0	0	0	1



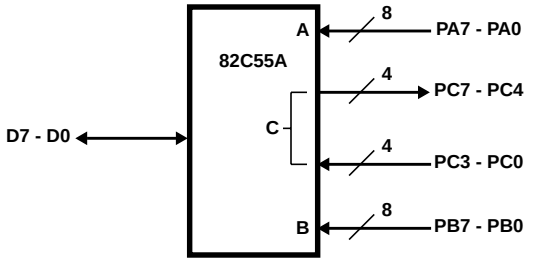
CONTROL WORD #10

D7	D6	D5	D4	D3	D2	D1	D0
1	0	0	1	0	0	1	0



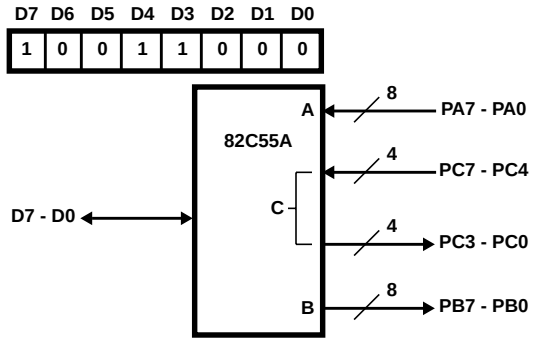
CONTROL WORD #11

D7	D6	D5	D4	D3	D2	D1	D0
1	0	0	1	0	0	1	1

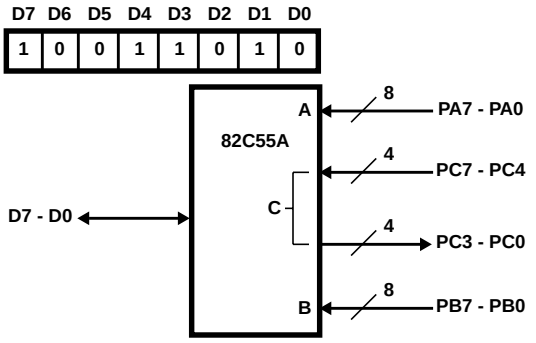


Mode 0 Configurations (Continued)

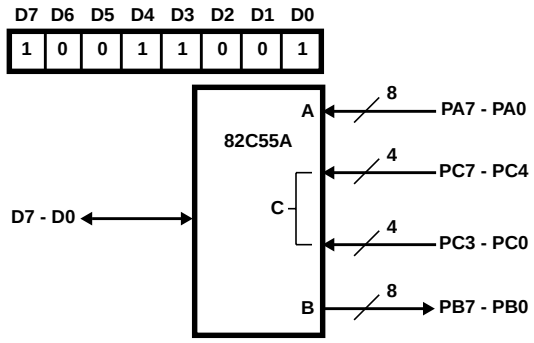
CONTROL WORD #12



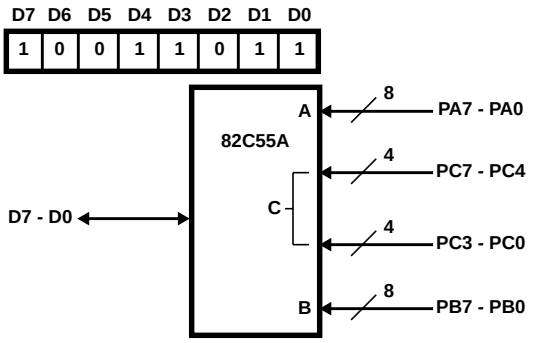
CONTROL WORD #14



CONTROL WORD #13



CONTROL WORD #15



Operating Modes

Mode 1 - (Strobed Input/Output). This functional configuration provides a means for transferring I/O data to or from a specified port in conjunction with strobes or “hand shaking” signals. In mode 1, port A and port B use the lines on port C to generate or accept these “hand shaking” signals.

Mode 1 Basic Function Definitions:

- Two Groups (Group A and Group B)
- Each group contains one 8-bit port and one 4-bit control/data port
- The 8-bit data port can be either input or output. Both inputs and outputs are latched.
- The 4-bit port is used for control and status of the 8-bit port.

Input Control Signal Definition

(Figures 6 and 7)

STB (Strobe Input)

A “low” on this input loads data into the input latch.

IBF (Input Buffer Full F/F)

A “high” on this output indicates that the data has been loaded into the input latch: in essence, and acknowledgment. IBF is set by $\overline{STB}$ input being low and is reset by the rising edge of the $\overline{RD}$ input.

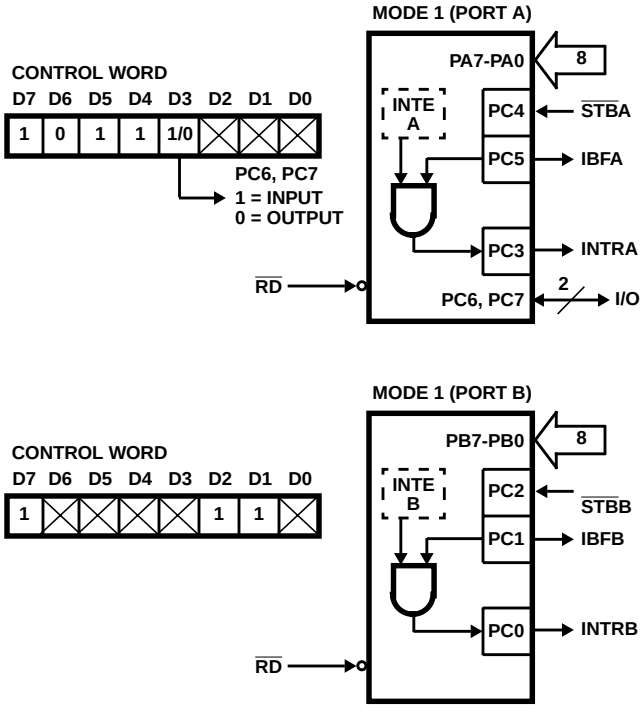


FIGURE 6. MODE 1 INPUT

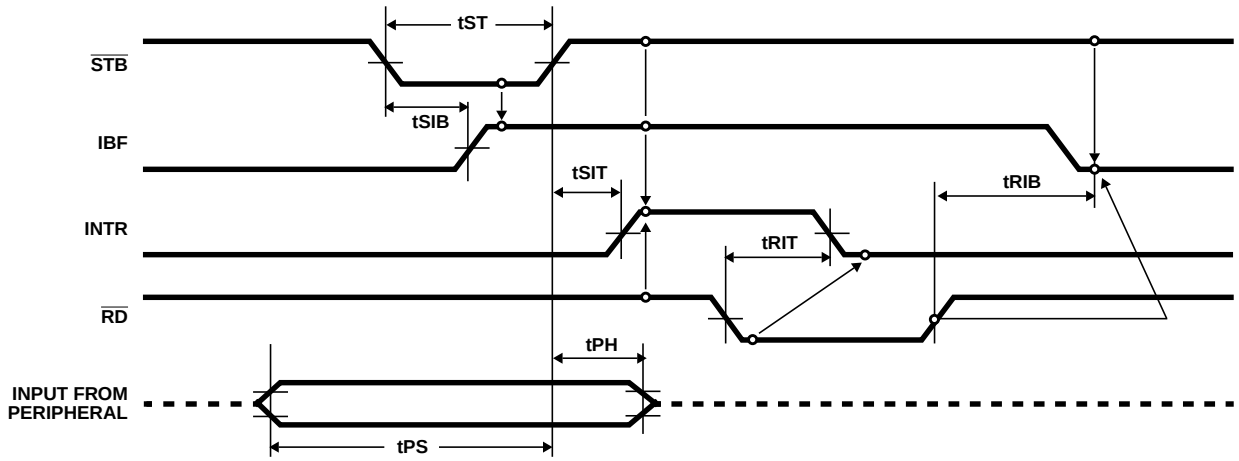


FIGURE 7. MODE 1 (STROBED INPUT)

INTR (Interrupt Request)

A “high” on this output can be used to interrupt the CPU when and input device is requesting service. INTR is set by the condition: $\overline{STB}$ is a “one”, IBF is a “one” and INTE is a “one”. It is reset by the falling edge of $\overline{RD}$. This procedure allows an input device to request service from the CPU by simply strobing its data into the port.

INTE A

Controlled by bit set/reset of PC4.

INTE B

Controlled by bit set/reset of PC2.

Output Control Signal Definition

(Figure 8 and 9)

$\overline{OBF}$ - Output Buffer Full F/F). The $\overline{OBF}$ output will go “low” to indicate that the CPU has written data out to be specified port. This does not mean valid data is sent out of the part at this time since $\overline{OBF}$ can go true before data is available. Data is guaranteed valid at the rising edge of $\overline{OBF}$, (See Note 1). The $\overline{OBF}$ F/F will be set by the rising edge of the $\overline{WR}$ input and reset by $\overline{ACK}$ input being low.

$\overline{ACK}$ - Acknowledge Input). A “low” on this input informs the 82C55A that the data from Port A or Port B is ready to be accepted. In essence, a response from the peripheral device indicating that it is ready to accept data, (See Note 1).

INTR - (Interrupt Request). A “high” on this output can be used to interrupt the CPU when an output device has accepted data transmitted by the CPU. INTR is set when $\overline{ACK}$ is a “one”, OBF is a “one” and INTE is a “one”. It is reset by the falling edge of $\overline{WR}$.

INTE A

Controlled by Bit Set/Reset of PC6.

INTE B

Controlled by Bit Set/Reset of PC2.

NOTE:

1. To strobe data into the peripheral device, the user must operate the strobe line in a hand shaking mode. The user needs to send $\overline{OBF}$ to the peripheral device, generates an $\overline{ACK}$ from the peripheral device and then latch data into the peripheral device on the rising edge of $\overline{OBF}$.

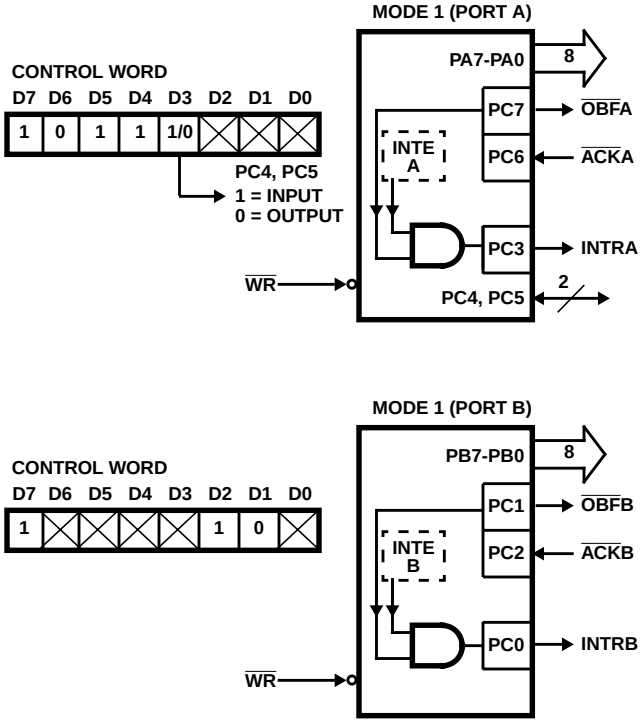


FIGURE 8. MODE 1 OUTPUT

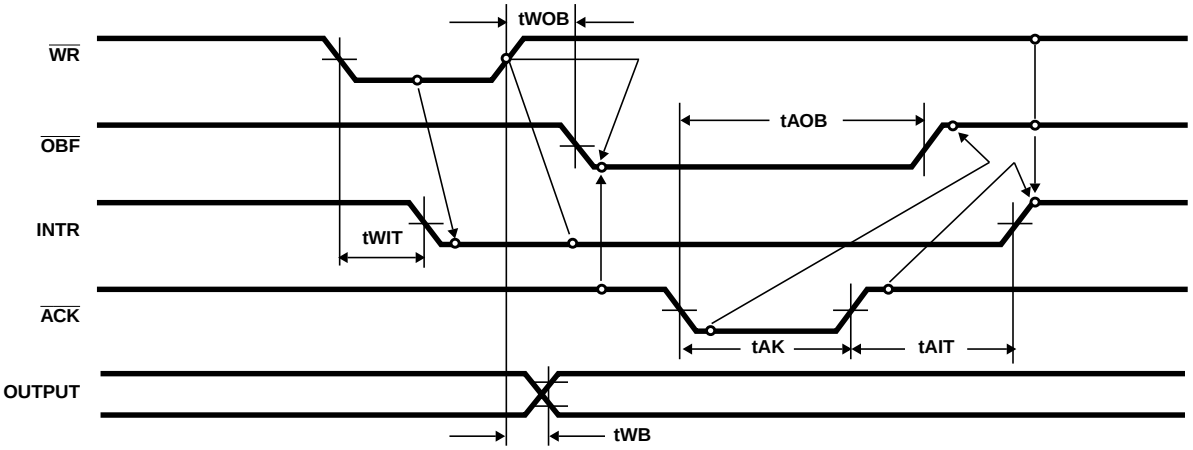
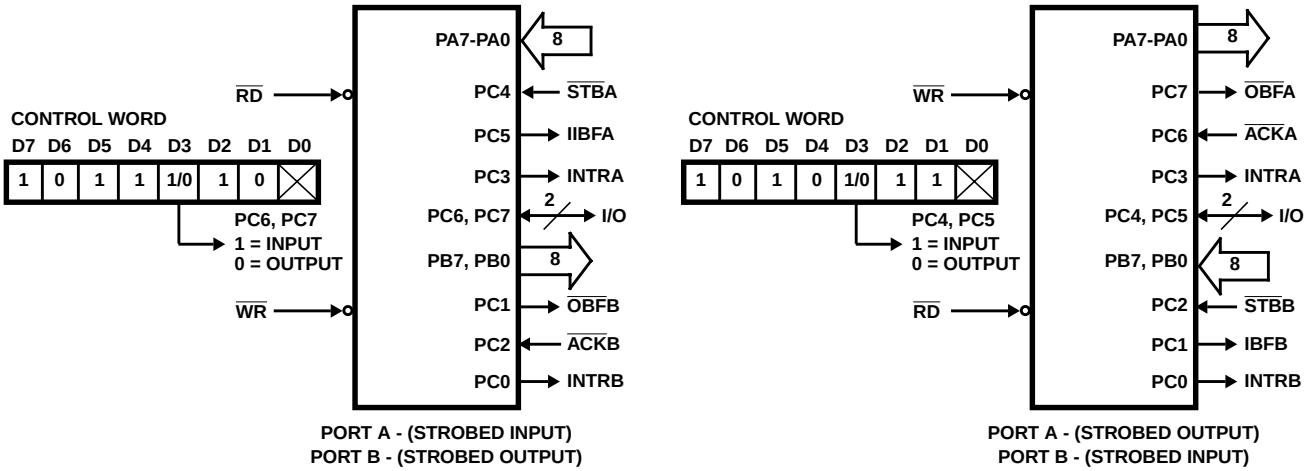


FIGURE 9. MODE 1 (STROBED OUTPUT)



Combinations of Mode 1: Port A and Port B can be individually defined as input or output in Mode 1 to support a wide variety of strobed I/O applications.

FIGURE 10. COMBINATIONS OF MODE 1

Operating Modes

Mode 2 (Strobed Bi-Directional Bus I/O)

The functional configuration provides a means for communicating with a peripheral device or structure on a single 8-bit bus for both transmitting and receiving data (bi-directional bus I/O). “Hand shaking” signals are provided to maintain proper bus flow discipline similar to Mode 1. Interrupt generation and enable/disable functions are also available.

Mode 2 Basic Functional Definitions:

- Used in Group A only
- One 8-bit, bi-directional bus Port (Port A) and a 5-bit control Port (Port C)
- Both inputs and outputs are latched
- The 5-bit control port (Port C) is used for control and status for the 8-bit, bi-directional bus port (Port A)

Bi-Directional Bus I/O Control Signal Definition

(Figures 11, 12, 13, 14)

INTR - (Interrupt Request). A high on this output can be used to interrupt the CPU for both input or output operations.

Output Operations

$\overline{\text{OBF}}$ - (Output Buffer Full). The $\overline{\text{OBF}}$ output will go “low” to indicate that the CPU has written data out to port A.

$\overline{\text{ACK}}$ - (Acknowledge). A “low” on this input enables the three-state output buffer of port A to send out the data. Otherwise, the output buffer will be in the high impedance state.

INTE 1 - (The INTE flip-flop associated with $\overline{\text{OBF}}$). Controlled by bit set/reset of PC4.

Input Operations

$\overline{\text{STB}}$ - (Strobe Input). A “low” on this input loads data into the input latch.

IBF - (Input Buffer Full F/F). A “high” on this output indicates that data has been loaded into the input latch.

INTE 2 - (The INTE flip-flop associated with IBF). Controlled by bit set/reset of PC4.

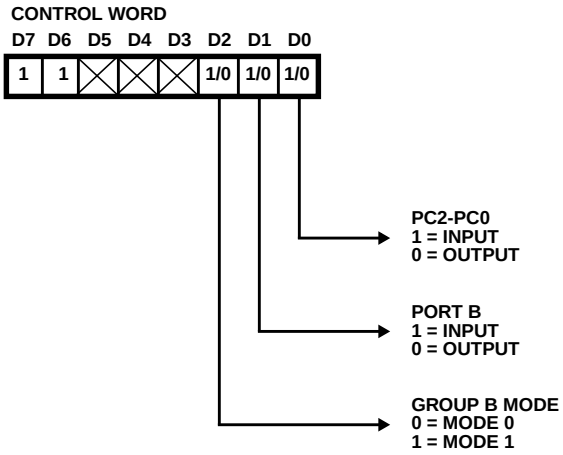


FIGURE 11. MODE CONTROL WORD

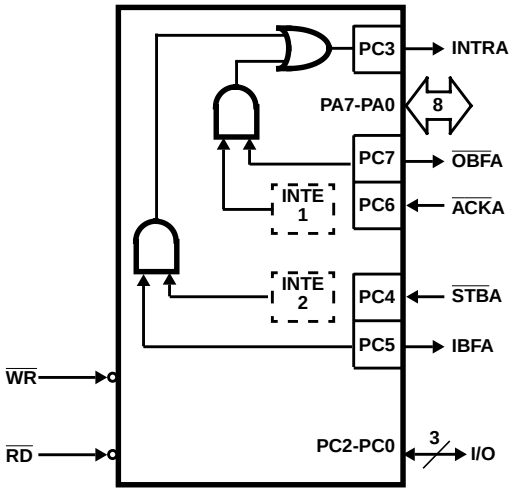
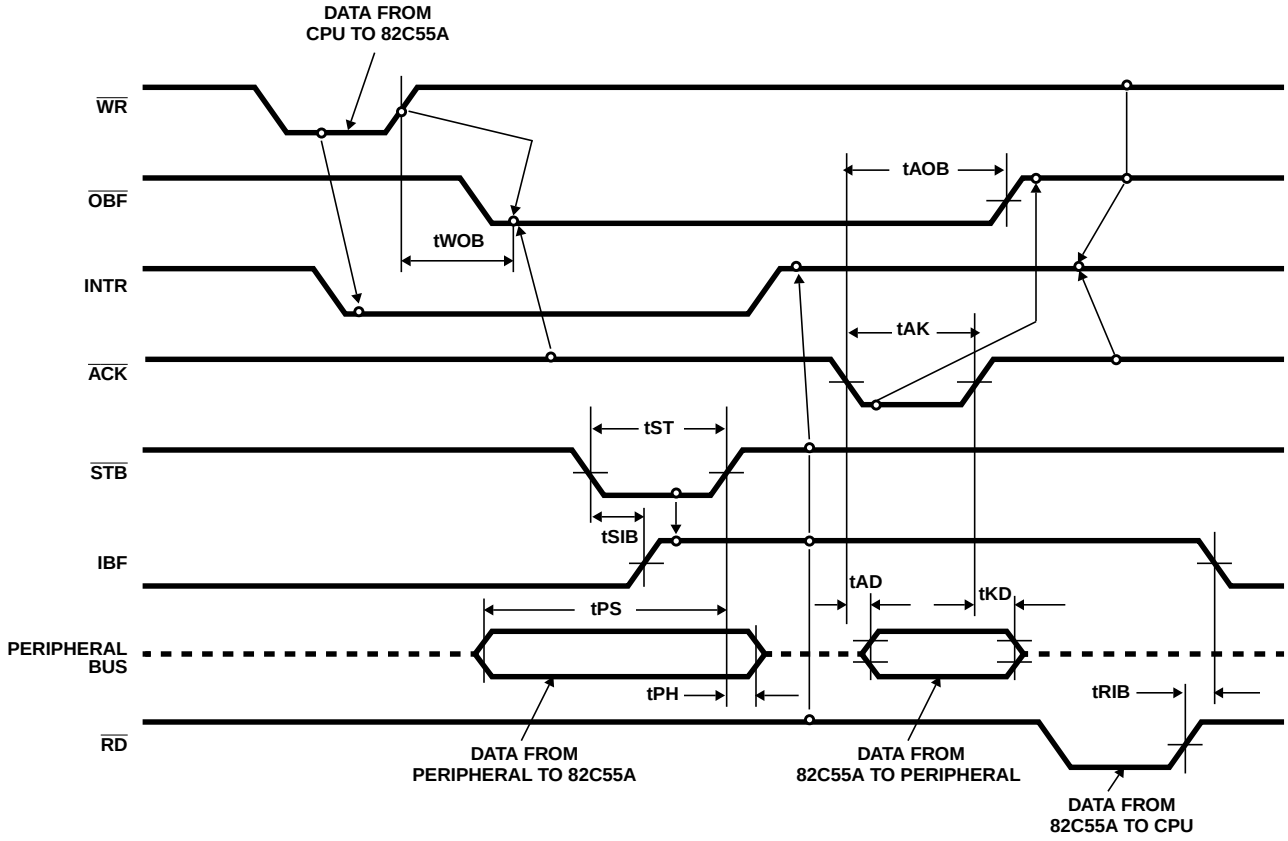


FIGURE 12. MODE 2



NOTE: Any sequence where $\overline{WR}$ occurs before $\overline{ACK}$ and $\overline{STB}$ occurs before RD is permissible. (INTR = IBF • MASK • $\overline{STB}$ • $\overline{RD}$ ÷ $\overline{OBF}$ • MASK • $\overline{ACK}$ • $\overline{WR}$)

FIGURE 13. MODE 2 (BI-DIRECTIONAL)

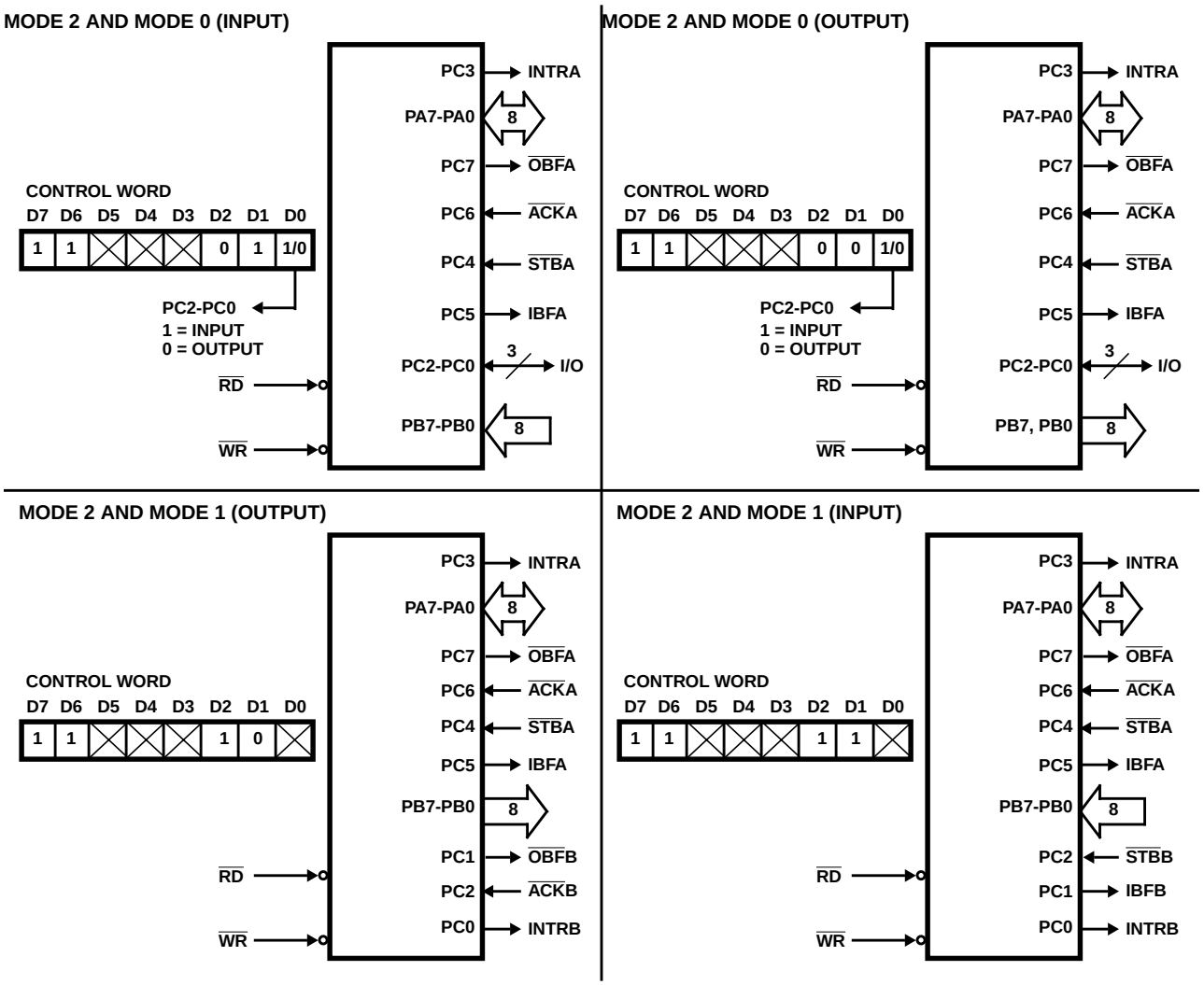


FIGURE 14. MODE 2 COMBINATIONS

MODE DEFINITION SUMMARY

	MODE 0		MODE 1		MODE 2	
	IN	OUT	IN	OUT	GROUP A ONLY	
PA0	In	Out	In	Out	↔	
PA1	In	Out	In	Out	↔	
PA2	In	Out	In	Out	↔	
PA3	In	Out	In	Out	↔	
PA4	In	Out	In	Out	↔	
PA5	In	Out	In	Out	↔	
PA6	In	Out	In	Out	↔	
PA7	In	Out	In	Out	↔	
PB0	In	Out	In	Out		Mode 0 or Mode 1 Only
PB1	In	Out	In	Out		
PB2	In	Out	In	Out		
PB3	In	Out	In	Out		
PB4	In	Out	In	Out		
PB5	In	Out	In	Out		
PB6	In	Out	In	Out		
PB7	In	Out	In	Out		
PC0	In	Out	INTRB	INTRB	I/O	
PC1	In	Out	IBFB	$\overline{OBF}$B	I/O	
PC2	In	Out	$\overline{STB}$B	$\overline{ACK}$B	I/O	
PC3	In	Out	INTRA	INTRA	INTRA	
PC4	In	Out	$\overline{STB}$A	I/O	$\overline{STB}$A	
PC5	In	Out	IBFA	I/O	IBFA	
PC6	In	Out	I/O	$\overline{ACK}$A	$\overline{ACK}$A	
PC7	In	Out	I/O	$\overline{OBF}$A	$\overline{OBF}$A	

Special Mode Combination Considerations

There are several combinations of modes possible. For any combination, some or all of Port C lines are used for control or status. The remaining bits are either inputs or outputs as defined by a “Set Mode” command.

During a read of Port C, the state of all the Port C lines, except the $\overline{ACK}$ and $\overline{STB}$ lines, will be placed on the data bus. In place of the $\overline{ACK}$ and $\overline{STB}$ line states, flag status will appear on the data bus in the PC2, PC4, and PC6 bit positions as illustrated by Figure 17.

Through a “Write Port C” command, only the Port C pins programmed as outputs in a Mode 0 group can be written. No other pins can be affected by a “Write Port C” command, nor can the interrupt enable flags be accessed. To write to any Port C output programmed as an output in Mode 1 group or to change an interrupt enable flag, the “Set/Reset Port C Bit” command must be used.

With a “Set/Reset Port Cea Bit” command, any Port C line programmed as an output (including IBF and $\overline{OBF}$) can be written, or an interrupt enable flag can be either set or reset. Port C lines programmed as inputs, including $\overline{ACK}$ and $\overline{STB}$ lines, associated with Port C fare not affected by a “Set/Reset Port C Bit” command. Writing to the corresponding Port C bit positions of the $\overline{ACK}$ and $\overline{STB}$ lines with the “Set Reset Port C Bit” command will affect the Group A and Group B interrupt enable flags, as illustrated in Figure 17.

INPUT CONFIGURATION

D7	D6	D5	D4	D3	D2	D1	D0
I/O	I/O	IBFA	INTEA	INTRA	INTEB	IBFB	INTRB
GROUP A					GROUP B		

OUTPUT CONFIGURATION

D7	D6	D5	D4	D3	D2	D1	D0
OBFA	INTEA	I/O	I/O	INTRA	INTEB	OBFB	INTRB
GROUP A					GROUP B		

FIGURE 15. MODE 1 STATUS WORD FORMAT

D7	D6	D5	D4	D3	D2	D1	D0
OBFA	INTE1	IBFA	INTE2	INTRA	X	X	X
GROUP A					GROUP B		

(Defined by Mode 0 or Mode 1 Selection)

FIGURE 16. MODE 2 STATUS WORD FORMAT

Current Drive Capability

Any output on Port A, B or C can sink or source 2.5mA. This feature allows the 82C55A to directly drive Darlington type drivers and high-voltage displays that require such sink or source current.

Reading Port C Status (Figures 15 and 16)

In Mode 0, Port C transfers data to or from the peripheral device. When the 82C55A is programmed to function in Modes 1 or 2, Port C generates or accepts “hand shaking” signals with the peripheral device. Reading the contents of Port C allows the programmer to test or verify the “status” of each peripheral device and change the program flow accordingly.

There is not special instruction to read the status information from Port C. A normal read operation of Port C is executed to perform this function.

INTERRUPT ENABLE FLAG	POSITION	ALTERNATE PORT C PIN SIGNAL (MODE)
INTE B	PC2	$\overline{ACK}$B (Output Mode 1) or $\overline{STB}$B (Input Mode 1)
INTE A2	PC4	$\overline{STB}$A (Input Mode 1 or Mode 2)
INTE A1	PC6	$\overline{ACK}$A (Output Mode 1 or Mode 2)

FIGURE 17. INTERRUPT ENABLE FLAGS IN MODES 1 AND 2

Applications of the 82C55A

The 82C55A is a very powerful tool for interfacing peripheral equipment to the microcomputer system. It represents the optimum use of available pins and flexible enough to interface almost any I/O device without the need for additional external logic.

Each peripheral device in a microcomputer system usually has a “service routine” associated with it. The routine manages the software interface between the device and the CPU. The functional definition of the 82C55A is programmed by the I/O service routine and becomes an extension of the system software. By examining the I/O devices interface characteristics for both data transfer and timing, and matching this information to the examples and tables in the detailed operational description, a control word can easily be developed to initialize the 82C55A to exactly “fit” the application. Figures 18 through 24 present a few examples of typical applications of the 82C55A.

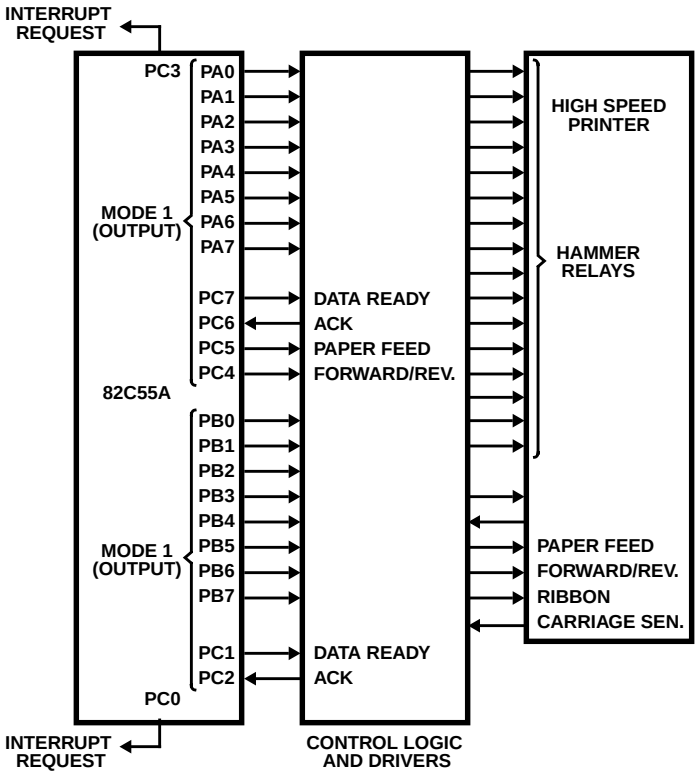


FIGURE 18. PRINTER INTERFACE

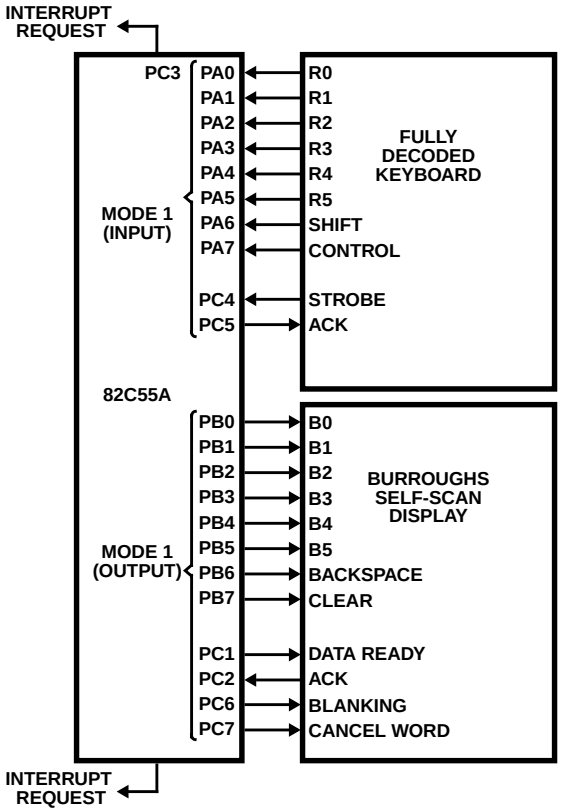


FIGURE 19. KEYBOARD AND DISPLAY INTERFACE

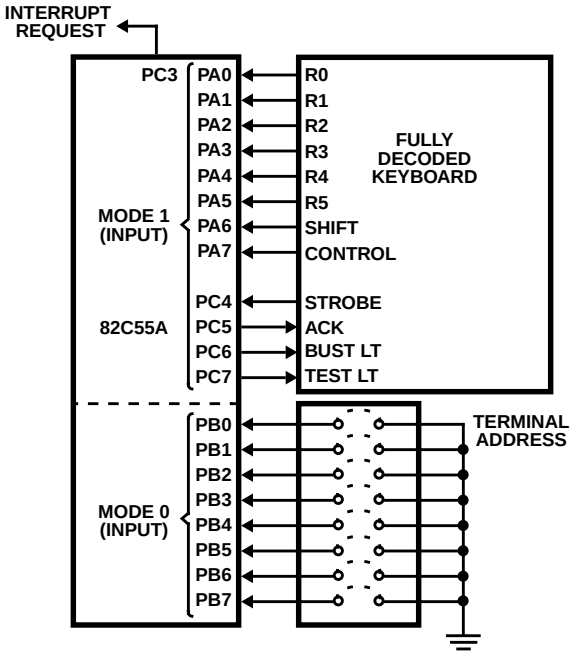


FIGURE 20. KEYBOARD AND TERMINAL ADDRESS INTERFACE

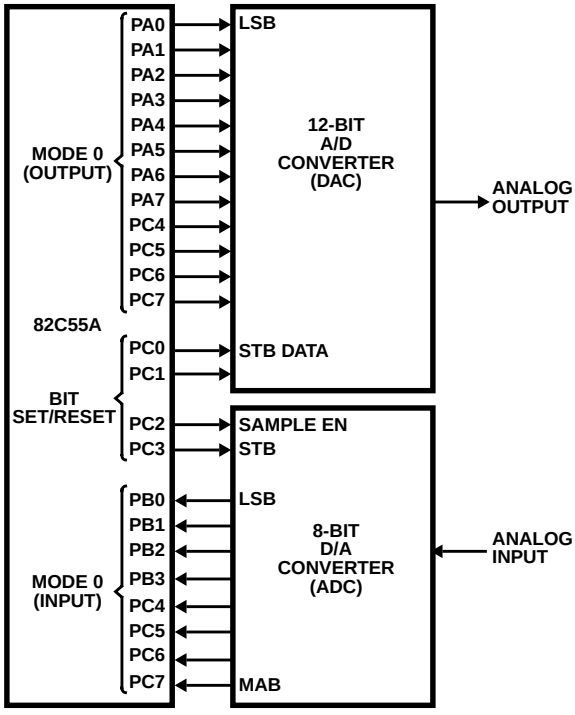


FIGURE 21. DIGITAL TO ANALOG, ANALOG TO DIGITAL

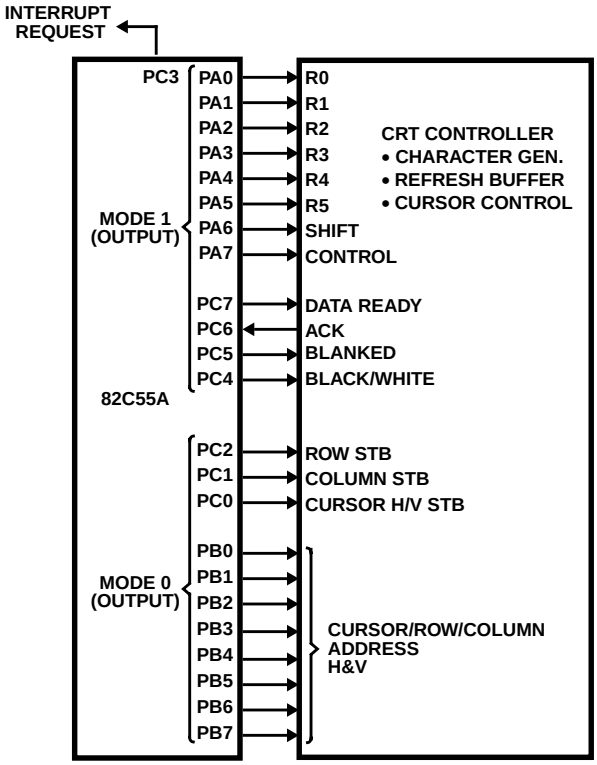


FIGURE 22. BASIC CRT CONTROLLER INTERFACE

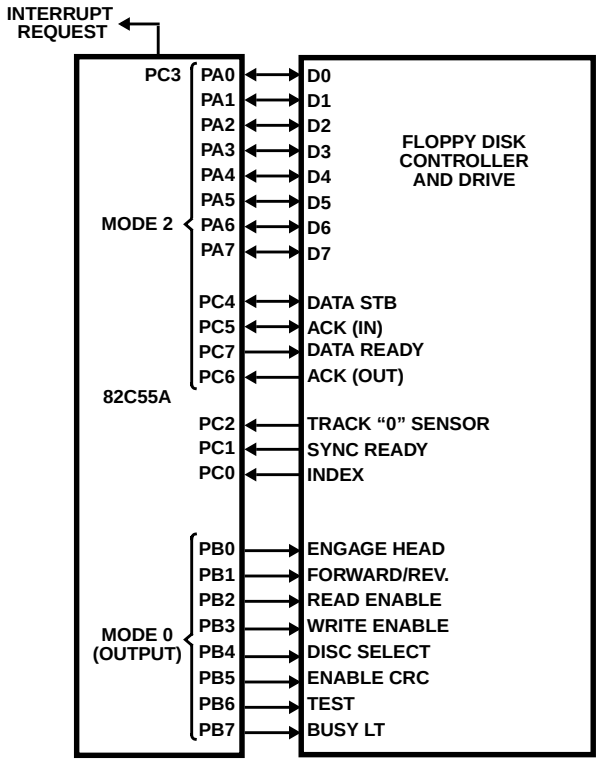


FIGURE 23. BASIC FLOPPY DISC INTERFACE

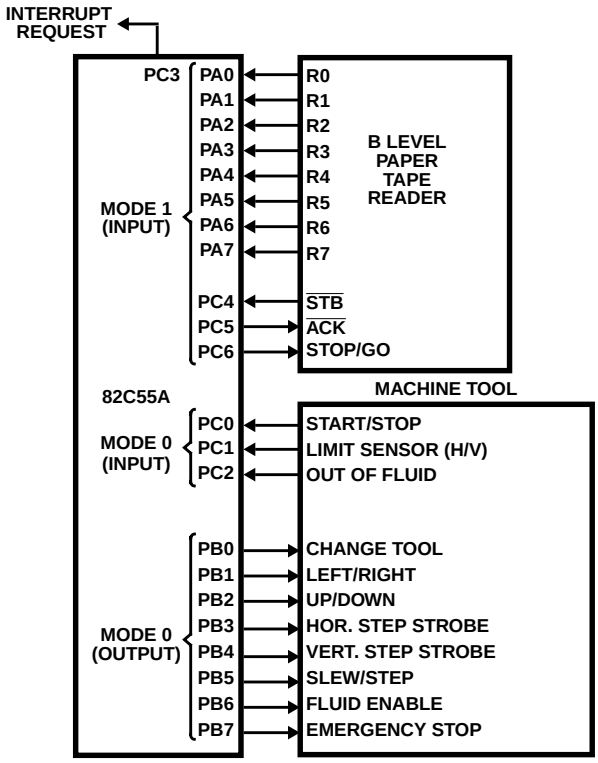


FIGURE 24. MACHINE TOOL CONTROLLER INTERFACE

Absolute Maximum Ratings $T_A = 25^oC$

Supply Voltage . . . . . +8.0V
Input, Output or I/O Voltage . . . . . GND-0.5V to V_{CC}+0.5V
ESD Classification . . . . . Class 1

Operating Conditions

Voltage Range . . . . . +4.5V to 5.5V
Operating Temperature Range
C82C55A . . . . . 0^oC to 70^oC
I82C55A . . . . . -40^oC to 85^oC
M82C55A . . . . . -55^oC to 125^oC

Thermal Information

Thermal Resistance (Typical, Note 1)	θ_{JA}	θ_{JC}
CERDIP Package	$50^oC/W$	$10^oC/W$
CLCC Package	$65^oC/W$	$14^oC/W$
PDIP Package	$50^oC/W$	N/A
PLCC Package	$46^oC/W$	N/A

Maximum Storage Temperature Range . . . . . -65^oC to 150^oC
Maximum Junction Temperature
CDIP Package . . . . . 175^oC
PDIP Package . . . . . 150^oC
Maximum Lead Temperature (Soldering 10s) . . . . . 300^oC
(PLCC Lead Tips Only)

Die Characteristics

Gate Count . . . . . 1000 Gates

CAUTION: Stresses above those listed in "Absolute Maximum Ratings" may cause permanent damage to the device. This is a stress only rating and operation of the device at these or any other conditions above those indicated in the operational sections of this specification is not implied.

NOTE:

1. θ_{JA} is measured with the component mounted on an evaluation PC board in free air.

Electrical Specifications

V_{CC} = 5.0V ±10%; $T_A = 0^oC$ to $+70^oC$ (C82C55A);
$T_A = -40^oC$ to $+85^oC$ (I82C55A);
$T_A = -55^oC$ to $+125^oC$ (M82C55A)

SYMBOL	PARAMETER	LIMITS		UNITS	TEST CONDITIONS
		MIN	MAX		
V_{IH}	Logical One Input Voltage	2.0 2.2	-	V	I82C55A, C82C55A, M82C55A
V_{IL}	Logical Zero Input Voltage	-	0.8	V	
V_{OH}	Logical One Output Voltage	3.0 V_{CC} -0.4	-	V	I_{OH} = -2.5mA, I_{OH} = -100µA
V_{OL}	Logical Zero Output Voltage	-	0.4	V	I_{OL} +2.5mA
I_I	Input Leakage Current	-1.0	+1.0	µA	$V_{IN} = V_{CC}$ or GND, DIP Pins: 5, 6, 8, 9, 35, 36
IO	I/O Pin Leakage Current	-10	+10	µA	VO = V_{CC} or GND DIP Pins: 27 - 34
IBHH	Bus Hold High Current	-50	-400	µA	VO = 3.0V. Ports A, B, C
IBHL	Bus Hold Low Current	50	400	µA	VO = 1.0V. Port A ONLY
IDAR	Darlington Drive Current	-2.5	Note 2, 4	mA	Ports A, B, C. Test Condition 3
ICCSB	Standby Power Supply Current	-	10	µA	V_{CC} = 5.5V, $V_{IN} = V_{CC}$ or GND. Output Open
ICCOP	Operating Power Supply Current	-	1	mA/MHz	$T_A = +25^oC$, V_{CC} = 5.0V, Typical (See Note 3)

NOTES:

2. No internal current limiting exists on Port Outputs. A resistor must be added externally to limit the current.
3. ICCOP = 1mA/MHz of Peripheral Read/Write cycle time. (Example: 1.0µs I/O Read/Write cycle time = 1mA).
4. Tested as V_{OH} at -2.5mA.

Capacitance $T_A = 25^oC$

SYMBOL	PARAMETER	TYPICAL	UNITS	TEST CONDITIONS
CIN	Input Capacitance	10	pF	FREQ = 1MHz, All Measurements are referenced to device GND
CI/O	I/O Capacitance	20	pF	

AC Electrical Specifications

V_{CC} = +5V± 10%, GND = 0V; T_A = -55°C to +125°C (M82C55A) (M82C55A-5);
T_A = -40°C to +85°C (I82C55A) (I82C55A-5);
T_A = 0°C to +70°C (C82C55A) (C82C55A-5)

SYMBOL	PARAMETER	82C55A-5		82C55A		UNITS	TEST CONDITIONS
		MIN	MAX	MIN	MAX		
READ TIMING							
(1) tAR	Address Stable Before $\overline{RD}$	0	-	0	-	ns	
(2) tRA	Address Stable After $\overline{RD}$	0	-	0	-	ns	
(3) tRR	$\overline{RD}$ Pulse Width	250	-	150	-	ns	
(4) tRD	Data Valid From $\overline{RD}$	-	200	-	120	ns	1
(5) tDF	Data Float After $\overline{RD}$	10	75	10	75	ns	2
(6) tRV	Time Between $\overline{RD}$s and/or $\overline{WR}$s	300	-	300	-	ns	
WRITE TIMING							
(7) tAW	Address Stable Before $\overline{WR}$	0	-	0	-	ns	
(8) tWA	Address Stable After $\overline{WR}$	20	-	20	-	ns	
(9) tWW	$\overline{WR}$ Pulse Width	100	-	100	-	ns	
(10) tDW	Data Valid to $\overline{WR}$ High	100	-	100	-	ns	
(11) tWD	Data Valid After $\overline{WR}$ High	30	-	30	-	ns	
OTHER TIMING							
(12) tWB	$\overline{WR}$ = 1 to Output	-	350	-	350	ns	1
(13) tIR	Peripheral Data Before $\overline{RD}$	0	-	0	-	ns	
(14) tHR	Peripheral Data After $\overline{RD}$	0	-	0	-	ns	
(15) tAK	ACK Pulse Width	200	-	200	-	ns	
(16) tST	STB Pulse Width	100	-	100	-	ns	
(17) tPS	Peripheral Data Before STB High	20	-	20	-	ns	
(18) tPH	Peripheral Data After STB High	50	-	50	-	ns	
(19) tAD	ACK = 0 to Output	-	175	-	175	ns	1
(20) tKD	ACK = 1 to Output Float	20	250	20	250	ns	2
(21) tWOB	$\overline{WR}$ = 1 to OBF = 0	-	150	-	150	ns	1
(22) tAOB	ACK = 0 to OBF = 1	-	150	-	150	ns	1
(23) tSIB	STB = 0 to IBF = 1	-	150	-	150	ns	1
(24) tRIB	$\overline{RD}$ = 1 to IBF = 0	-	150	-	150	ns	1
(25) tRIT	$\overline{RD}$ = 0 to INTR = 0	-	200	-	200	ns	1
(26) tSIT	STB = 1 to INTR = 1	-	150	-	150	ns	1
(27) tAIT	ACK = 1 to INTR = 1	-	150	-	150	ns	1
(28) tWIT	$\overline{WR}$ = 0 to INTR = 0	-	200	-	200	ns	1
(29) tRES	Reset Pulse Width	500	-	500	-	ns	1, (Note)

NOTE: Period of initial Reset pulse after power-on must be at least 50µsec. Subsequent Reset pulses may be 500ns minimum.

Timing Waveforms

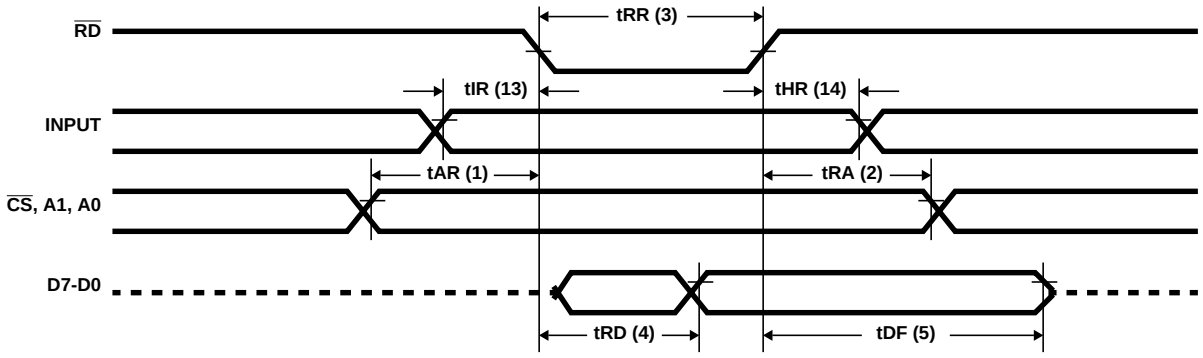


FIGURE 25. MODE 0 (BASIC INPUT)

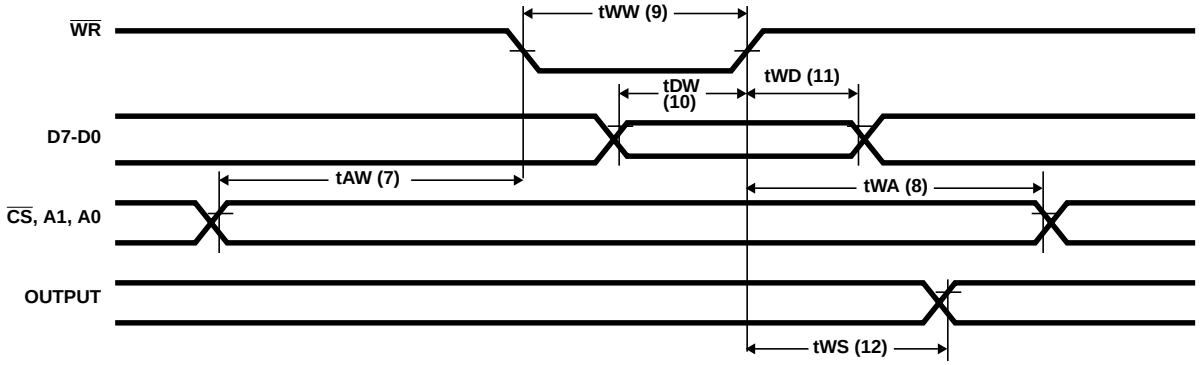


FIGURE 26. MODE 0 (BASIC OUTPUT)

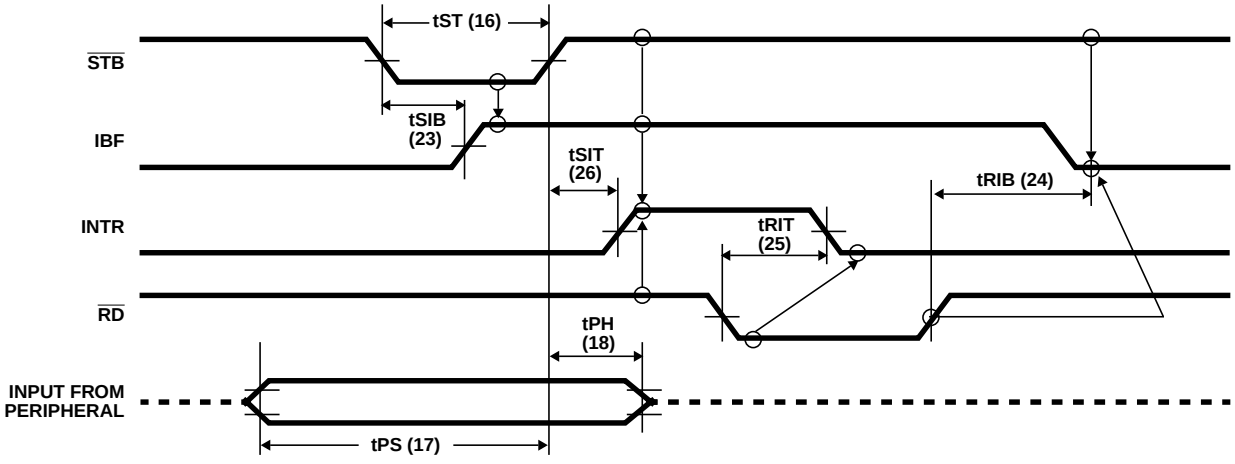


FIGURE 27. MODE 1 (STROBED INPUT)

Timing Waveforms (Continued)

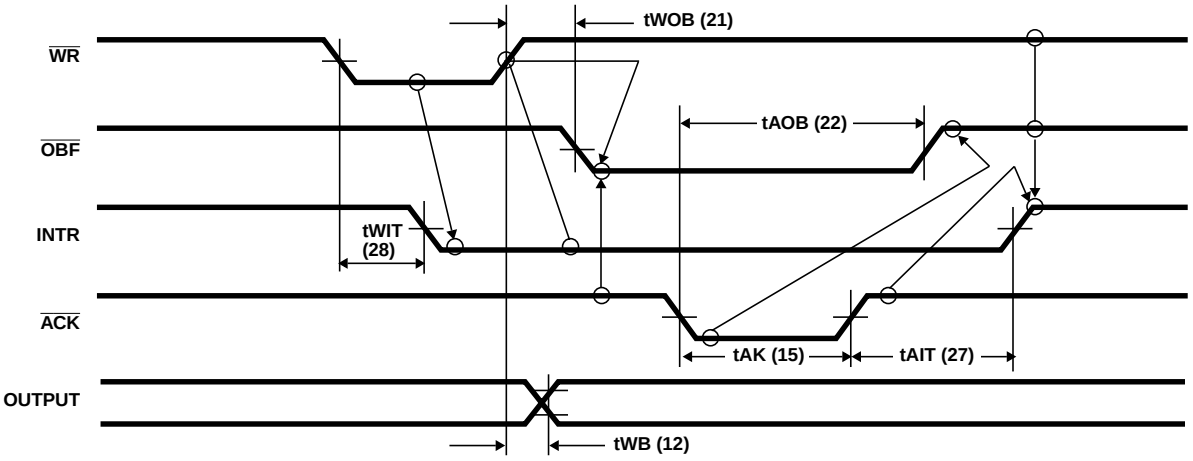


FIGURE 28. MODE 1 (STROBED OUTPUT)

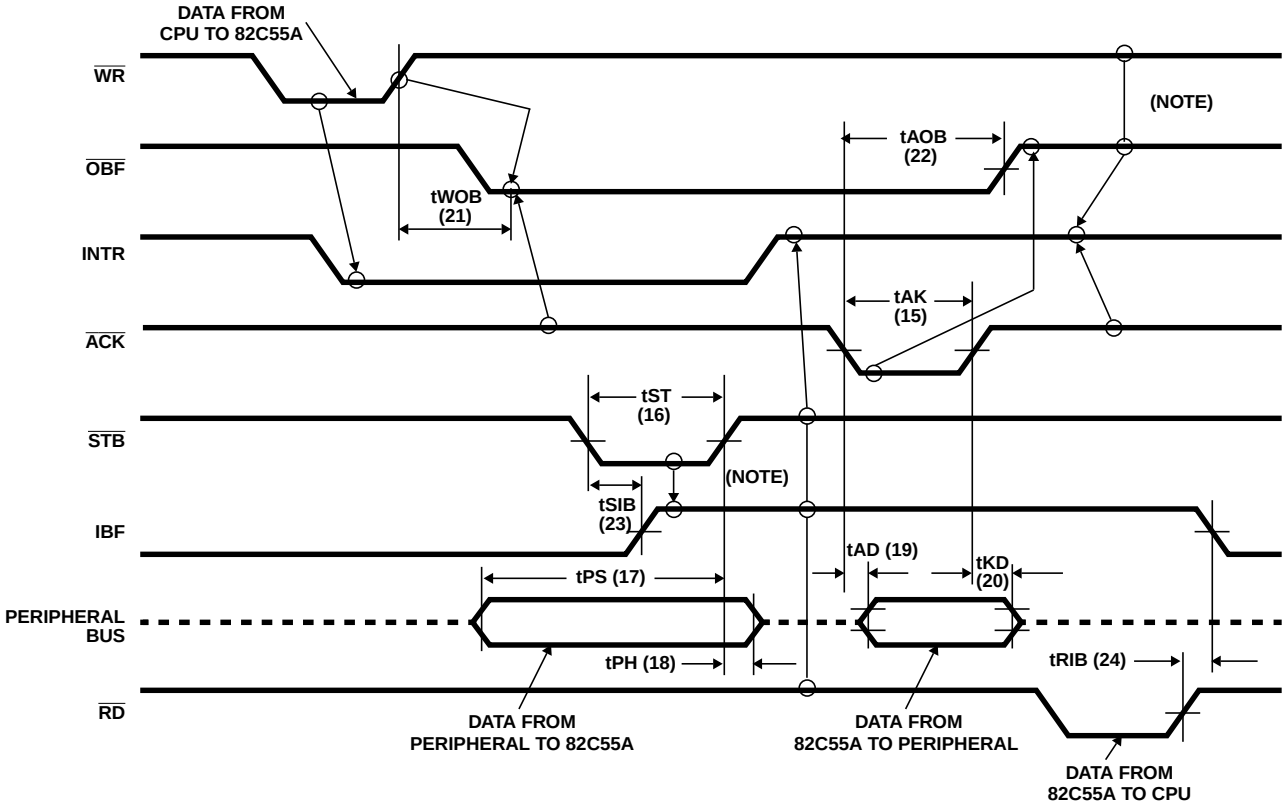


FIGURE 29. MODE 2 (BI-DIRECTIONAL)

NOTE: Any sequence where $\overline{WR}$ occurs before $\overline{ACK}$ and $\overline{STB}$ occurs before $\overline{RD}$ is permissible. (INTR = IBF • $\overline{MASK}$ • $\overline{STB}$ • $\overline{RD}$ • $\overline{OBF}$ • $\overline{MASK}$ • $\overline{ACK}$ • $\overline{WR}$)

Timing Waveforms (Continued)

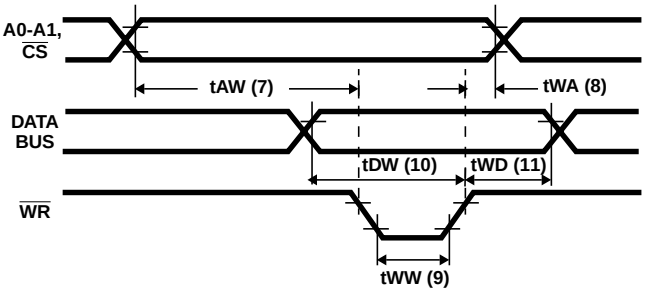


FIGURE 30. WRITE TIMING

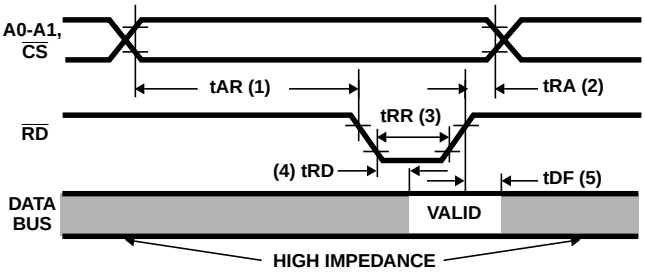
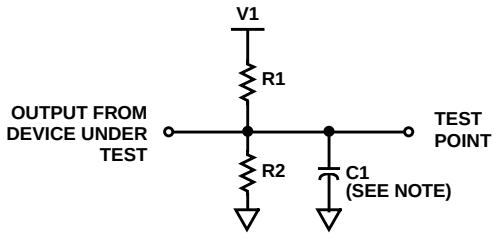


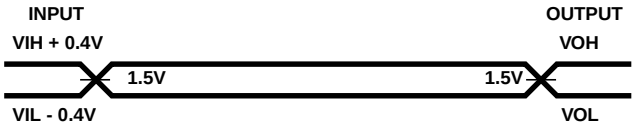
FIGURE 31. READ TIMING

AC Test Circuit



NOTE: Includes STRAY and JIG Capacitance

AC Testing Input, Output Waveforms



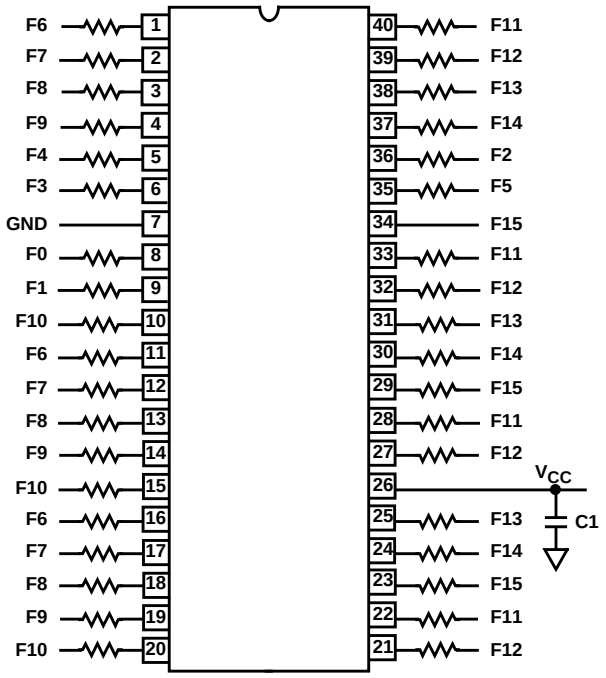
AC Testing: All AC Parameters tested as per test circuits. Input RISE and FALL times are driven at 1ns/V.

TEST CONDITION DEFINITION TABLE

TEST CONDITION	V1	R1	R2	C1
1	1.7V	523Ω	Open	150pF
2	V_{CC}	2kΩ	1.7kΩ	50pF
3	1.5V	750Ω	Open	50pF

Burn-In Circuits

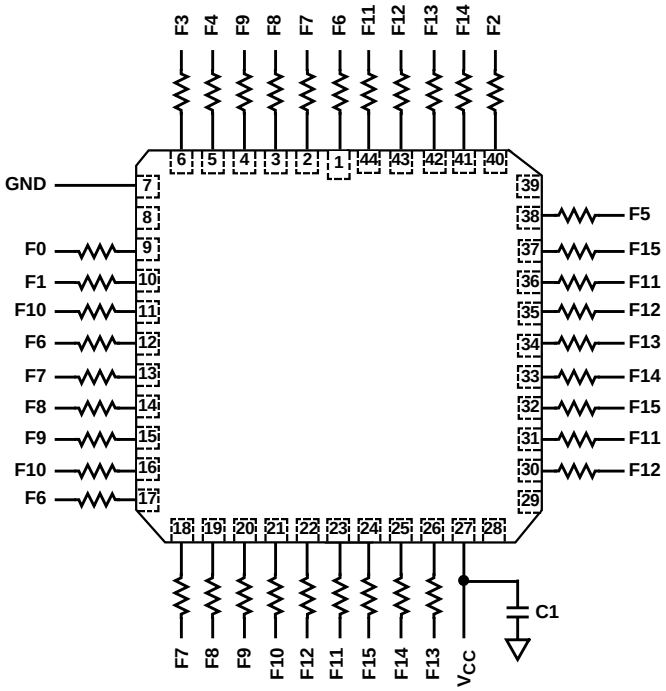
MD82C55A CERDIP



NOTES:

1. V_{CC} = 5.5V ± 0.5V
2. VIH = 4.5V ± 10%
3. VIL = -0.2V to 0.4V
4. GND = 0V

MR82C55A CLCC



NOTES:

1. C1 = 0.01µF minimum
2. All resistors are 47kΩ ± 5%
3. f0 = 100kHz ± 10%
4. f1 = f0 ÷ 2; f2 = f1 ÷ 2; . . . ; f15 = f14 ÷ 2

Die Characteristics

DIE DIMENSIONS:
95 x 100 x 19 ±1mils

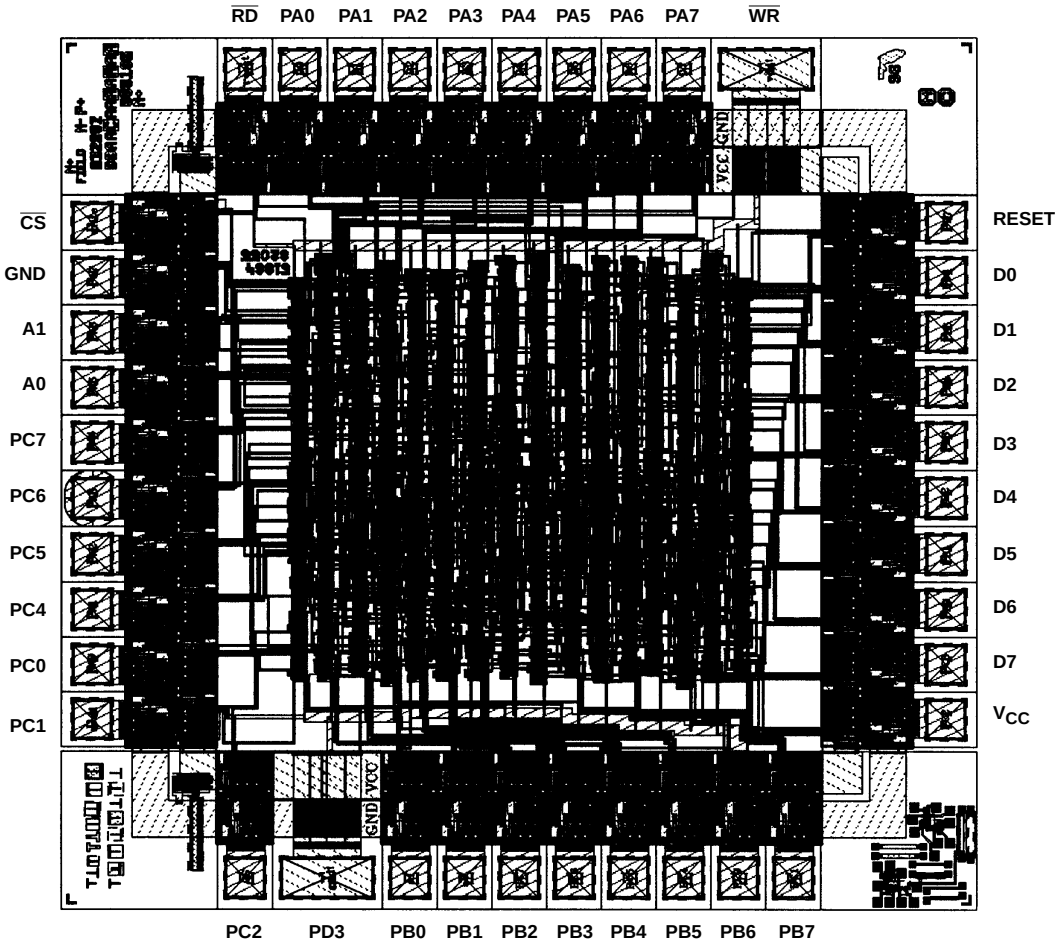
METALLIZATION:
Type: Silicon - Aluminum
Thickness: 11kÅ ±1kÅ

GLASSIVATION:
Type: SiO_2
Thickness: 8kÅ ±1kÅ

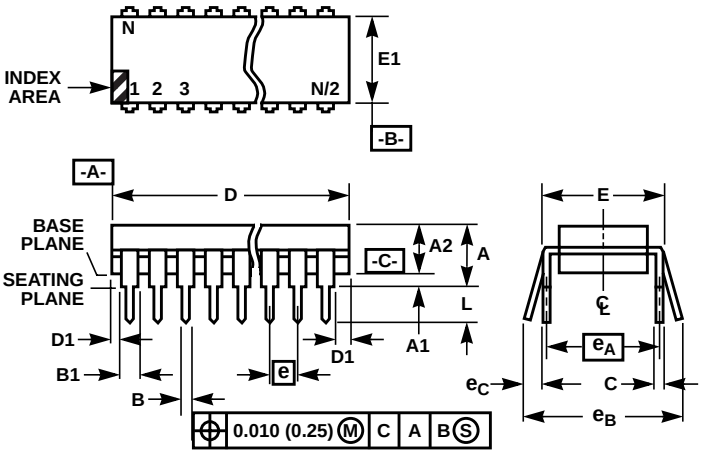
WORST CASE CURRENT DENSITY:
0.78×10^5 A/cm^2

Metallization Mask Layout

82C55A



Dual-In-Line Plastic Packages (PDIP)



NOTES:

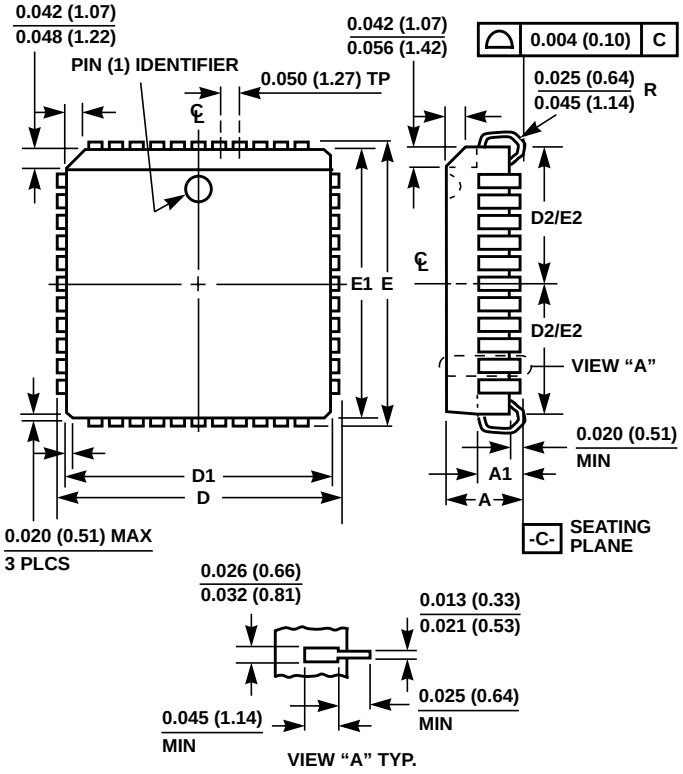
1. Controlling Dimensions: INCH. In case of conflict between English and Metric dimensions, the inch dimensions control.
2. Dimensioning and tolerancing per ANSI Y14.5M-1982.
3. Symbols are defined in the “MO Series Symbol List” in Section 2.2 of Publication No. 95.
4. Dimensions A, A1 and L are measured with the package seated in JEDEC seating plane gauge GS-3.
5. D, D1, and E1 dimensions do not include mold flash or protrusions. Mold flash or protrusions shall not exceed 0.010 inch (0.25mm).
6. E and e_A are measured with the leads constrained to be perpendicular to datum -C-.
7. e_B and e_C are measured at the lead tips with the leads unconstrained. e_C must be zero or greater.
8. B1 maximum dimensions do not include dambar protrusions. Dambar protrusions shall not exceed 0.010 inch (0.25mm).
9. N is the maximum number of terminal positions.
10. Corner leads (1, N, N/2 and N/2 + 1) for E8.3, E16.3, E18.3, E28.3, E42.6 will have a B1 dimension of 0.030 - 0.045 inch (0.76 - 1.14mm).

E40.6 (JEDEC MS-011-AC ISSUE B)
40 LEAD DUAL-IN-LINE PLASTIC PACKAGE

SYMBOL	INCHES		MILLIMETERS		NOTES
	MIN	MAX	MIN	MAX	
A	-	0.250	-	6.35	4
A1	0.015	-	0.39	-	4
A2	0.125	0.195	3.18	4.95	-
B	0.014	0.022	0.356	0.558	-
B1	0.030	0.070	0.77	1.77	8
C	0.008	0.015	0.204	0.381	-
D	1.980	2.095	50.3	53.2	5
D1	0.005	-	0.13	-	5
E	0.600	0.625	15.24	15.87	6
E1	0.485	0.580	12.32	14.73	5
e	0.100 BSC		2.54 BSC		-
e_A	0.600 BSC		15.24 BSC		6
e_B	-	0.700	-	17.78	7
L	0.115	0.200	2.93	5.08	4
N	40		40		9

Rev. 0 12/93

Plastic Leaded Chip Carrier Packages (PLCC)



N44.65 **(JEDEC MS-018AC ISSUE A)**
44 LEAD PLASTIC LEADED CHIP CARRIER PACKAGE

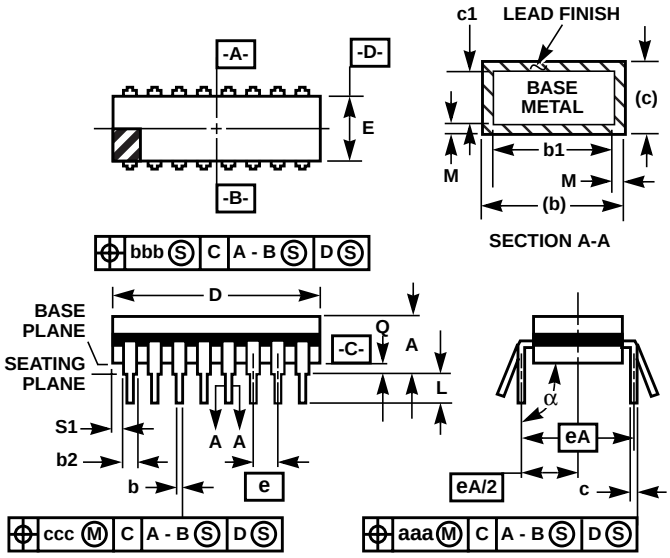
SYMBOL	INCHES		MILLIMETERS		NOTES
	MIN	MAX	MIN	MAX	
A	0.165	0.180	4.20	4.57	-
A1	0.090	0.120	2.29	3.04	-
D	0.685	0.695	17.40	17.65	-
D1	0.650	0.656	16.51	16.66	3
D2	0.291	0.319	7.40	8.10	4, 5
E	0.685	0.695	17.40	17.65	-
E1	0.650	0.656	16.51	16.66	3
E2	0.291	0.319	7.40	8.10	4, 5
N	44		44		6

Rev. 2 11/97

NOTES:

1. Controlling dimension: INCH. Converted millimeter dimensions are not necessarily exact.
2. Dimensions and tolerancing per ANSI Y14.5M-1982.
3. Dimensions D1 and E1 do not include mold protrusions. Allowable mold protrusion is 0.010 inch (0.25mm) per side. Dimensions D1 and E1 include mold mismatch and are measured at the extreme material condition at the body parting line.
4. To be measured at seating plane -C- contact point.
5. Centerline to be determined where center leads exit plastic body.
6. "N" is the number of terminal positions.

Ceramic Dual-In-Line Frit Seal Packages (CERDIP)



F40.6 MIL-STD-1835 GDIP1-T40 (D-5, CONFIGURATION A)
40 LEAD CERAMIC DUAL-IN-LINE FRIT SEAL PACKAGE

SYMBOL	INCHES		MILLIMETERS		NOTES
	MIN	MAX	MIN	MAX	
A	-	0.225	-	5.72	-
b	0.014	0.026	0.36	0.66	2
b1	0.014	0.023	0.36	0.58	3
b2	0.045	0.065	1.14	1.65	-
b3	0.023	0.045	0.58	1.14	4
c	0.008	0.018	0.20	0.46	2
c1	0.008	0.015	0.20	0.38	3
D	-	2.096	-	53.24	5
E	0.510	0.620	12.95	15.75	5
e	0.100 BSC		2.54 BSC		-
eA	0.600 BSC		15.24 BSC		-
eA/2	0.300 BSC		7.62 BSC		-
L	0.125	0.200	3.18	5.08	-
Q	0.015	0.070	0.38	1.78	6
S1	0.005	-	0.13	-	7
α	90^o	105^o	90^o	105^o	-
aaa	-	0.015	-	0.38	-
bbb	-	0.030	-	0.76	-
ccc	-	0.010	-	0.25	-
M	-	0.0015	-	0.038	2, 3
N	40		40		8

Rev. 0 4/94

NOTES:

1. Index area: A notch or a pin one identification mark shall be located adjacent to pin one and shall be located within the shaded area shown. The manufacturer's identification shall not be used as a pin one identification mark.
2. The maximum limits of lead dimensions b and c or M shall be measured at the centroid of the finished lead surfaces, when solder dip or tin plate lead finish is applied.
3. Dimensions b1 and c1 apply to lead base metal only. Dimension M applies to lead plating and finish thickness.
4. Corner leads (1, N, N/2, and N/2+1) may be configured with a partial lead paddle. For this configuration dimension b3 replaces dimension b2.
5. This dimension allows for off-center lid, meniscus, and glass overrun.
6. Dimension Q shall be measured from the seating plane to the base plane.
7. Measure dimension S1 at all four corners.
8. N is the maximum number of terminal positions.
9. Dimensioning and tolerancing per ANSI Y14.5M - 1982.
10. Controlling dimension: INCH.

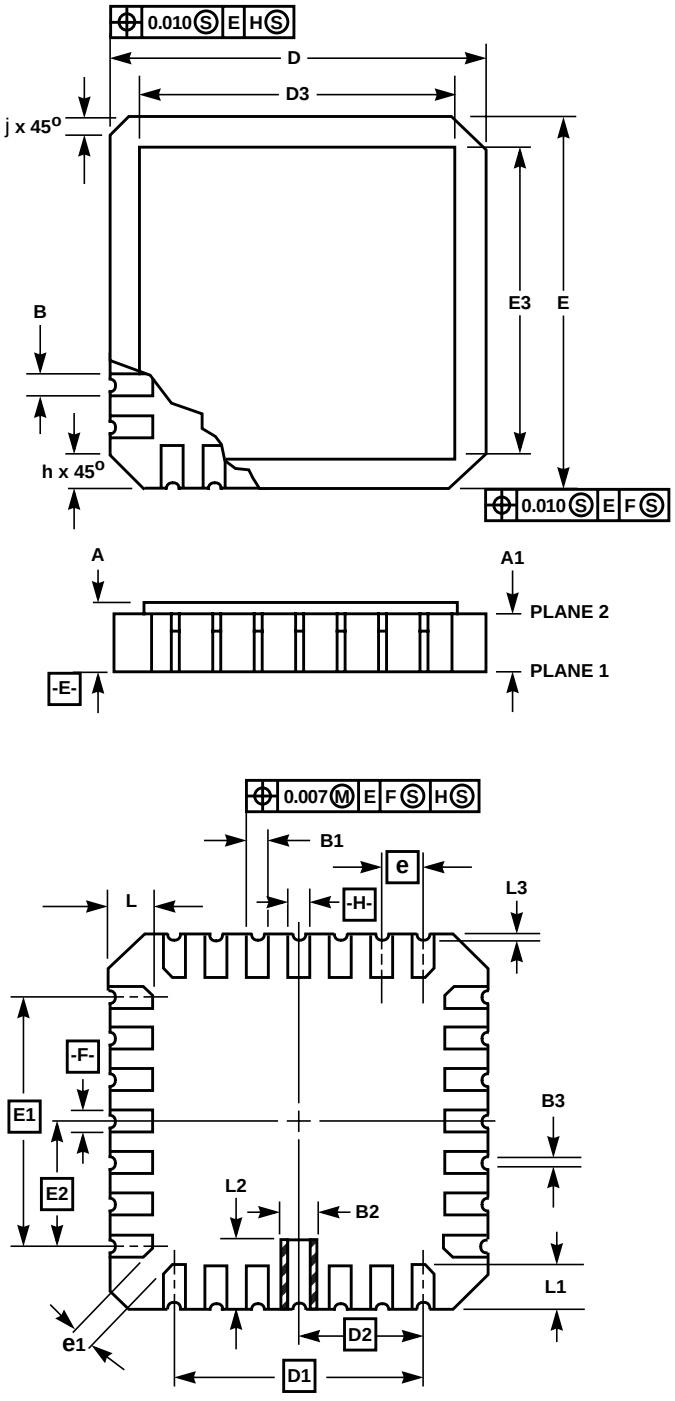
Sales Office Headquarters

NORTH AMERICA
Intersil Corporation
P. O. Box 883, Mail Stop 53-204
Melbourne, FL 32902
TEL: (407) 724-7000
FAX: (407) 724-7240

EUROPE
Intersil SA
Mercure Center
100, Rue de la Fusee
1130 Brussels, Belgium
TEL: (32) 2.724.2111
FAX: (32) 2.724.22.05

ASIA
Intersil (Taiwan) Ltd.
Taiwan Limited
7F-6, No. 101 Fu Hsing North Road
Taipei, Taiwan
Republic of China
TEL: (886) 2 2716 9310
FAX: (886) 2 2715 3029

Ceramic Leadless Chip Carrier Packages (CLCC)



J44.A **MIL-STD-1835 CQCC1-N44 (C-5)**
44 PAD CERAMIC LEADLESS CHIP CARRIER PACKAGE

SYMBOL	INCHES		MILLIMETERS		NOTES
	MIN	MAX	MIN	MAX	
A	0.064	0.120	1.63	3.05	6, 7
A1	0.054	0.088	1.37	2.24	-
B	0.033	0.039	0.84	0.99	4
B1	0.022	0.028	0.56	0.71	2, 4
B2	0.072 REF		1.83 REF		-
B3	0.006	0.022	0.15	0.56	-
D	0.640	0.662	16.26	16.81	-
D1	0.500 BSC		12.70 BSC		-
D2	0.250 BSC		6.35 BSC		-
D3	-	0.662	-	16.81	2
E	0.640	0.662	16.26	16.81	-
E1	0.500 BSC		12.70 BSC		-
E2	0.250 BSC		6.35 BSC		-
E3	-	0.662	-	16.81	2
e	0.050 BSC		1.27 BSC		-
e1	0.015	-	0.38	-	2
h	0.040 REF		1.02 REF		5
j	0.020 REF		0.51 REF		5
L	0.045	0.055	1.14	1.40	-
L1	0.045	0.055	1.14	1.40	-
L2	0.075	0.095	1.90	2.41	-
L3	0.003	0.015	0.08	0.38	-
ND	11		11		3
NE	11		11		3
N	44		44		3

Rev. 0 5/18/94

NOTES:

1. Metallized castellations shall be connected to plane 1 terminals and extend toward plane 2 across at least two layers of ceramic or completely across all of the ceramic layers to make electrical connection with the optional plane 2 terminals.
2. Unless otherwise specified, a minimum clearance of 0.015 inch (0.38mm) shall be maintained between all metallized features (e.g., lid, castellations, terminals, thermal pads, etc.)
3. Symbol “N” is the maximum number of terminals. Symbols “ND” and “NE” are the number of terminals along the sides of length “D” and “E”, respectively.
4. The required plane 1 terminals and optional plane 2 terminals (if used) shall be electrically connected.
5. The corner shape (square, notch, radius, etc.) may vary at the manufacturer’s option, from that shown on the drawing.
6. Chip carriers shall be constructed of a minimum of two ceramic layers.
7. Dimension “A” controls the overall package thickness. The maximum “A” dimension is package height before being solder dipped.
8. Dimensioning and tolerancing per ANSI Y14.5M-1982.
9. Controlling dimension: INCH.

82C59A

March 1997

CMOS Priority Interrupt Controller

Features

- **12.5MHz, 8MHz and 5MHz Versions Available**
  - **12.5MHz Operation 82C59A-12**
  - **8MHz Operation 82C59A**
  - **5MHz Operation 82C59A-5**
- **High Speed, "No Wait-State" Operation with 12.5MHz 80C286 and 8MHz 80C86/88**
- **Pin Compatible with NMOS 8259A**
- **80C86/88/286 and 8080/85/86/88/286 Compatible**
- **Eight-Level Priority Controller, Expandable to 64 Levels**
- **Programmable Interrupt Modes**
- **Individual Request Mask Capability**
- **Fully Static Design**
- **Fully TTL Compatible**
- **Low Power Operation**
  - **ICCSB . . . . . 10μA Maximum**
  - **ICCOP . . . . . 1mA/MHz Maximum**
- **Single 5V Power Supply**
- **Operating Temperature Ranges**
  - **C82C59A . . . . . 0^oC to $+70^oC$**
  - **I82C59A . . . . . -40^oC to $+85^oC$**
  - **M82C59A . . . . . -55^oC to $+125^oC$**

Description

The Intersil 82C59A is a high performance CMOS Priority Interrupt Controller manufactured using an advanced 2μm CMOS process. The 82C59A is designed to relieve the system CPU from the task of polling in a multilevel priority system. The high speed and industry standard configuration of the 82C59A make it compatible with microprocessors such as 80C286, 80286, 80C86/88, 8086/88, 8080/85 and NSC800.

The 82C59A can handle up to eight vectored priority interrupting sources and is cascadable to 64 without additional circuitry. Individual interrupting sources can be masked or prioritized to allow custom system configuration. Two modes of operation make the 82C59A compatible with both 8080/85 and 80C86/88/286 formats.

Static CMOS circuit design ensures low operating power. The Intersil advanced CMOS process results in performance equal to or greater than existing equivalent products at a fraction of the power.

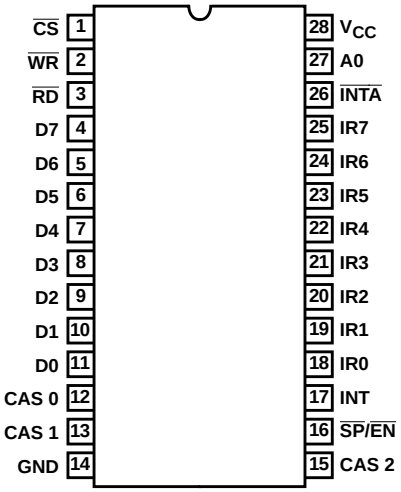
Ordering Information

PART NUMBER			PACKAGE	TEMPERATURE RANGE	PKG. NO.
5MHz	8MHz	12.5MHz			
CP82C59A-5	CP82C59A	CP82C59A-12	28 Ld PDIP	0^oC to $+70^oC$	E28.6
IP82C59A-5	IP82C59A	IP82C59A-12		-40^oC to $+85^oC$	E28.6
CS82C59A-5	CS82C59A	CS82C59A-12	28 Ld PLCC	0^oC to $+70^oC$	N28.45
IS82C59A-5	IS82C59A	IS82C59A-12		-40^oC to $+85^oC$	N28.45
CD82C59A-5	CD82C59A	CD82C59A-12	CERDIP	0^oC to $+70^oC$	F28.6
ID82C59A-5	ID82C59A	ID82C59A-12		-40^oC to $+85^oC$	F28.6
MD82C59A-5/B	MD82C59A/B	MD82C59A-12/B		-55^oC to $+125^oC$	F28.6
5962-8501601YA	5962-8501602YA	-	SMD#		F28.6
MR82C59A-5/B	MR82C59A/B	MR82C59A-12/B	28 Pad CLCC	-55^oC to $+125^oC$	J28.A
5962-85016013A	5962-85016023A	-	SMD#		J28.A
CM82C59A-5	CM82C59A	CM82C59A-12	28 Ld SOIC	0^oC to $+70^oC$	M28.3

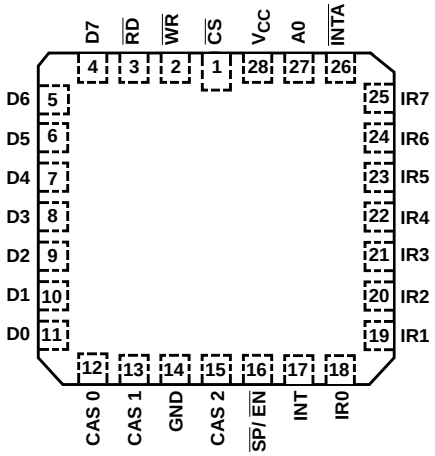
File Number **2784.2**

Pinouts

82C59A (PDIP, CERDIP, SOIC)
TOP VIEW



82C59A (PLCC, CLCC)
TOP VIEW



PIN	DESCRIPTION
D7 - D0	Data Bus (Bidirectional)
$\overline{RD}$	Read Input
$\overline{WR}$	Write Input
A0	Command Select Address
$\overline{CS}$	Chip Select
CAS 2 - CAS 0	Cascade Lines
$\overline{SP}/\overline{EN}$	Slave Program Input Enable
INT	Interrupt Output
$\overline{INTA}$	Interrupt Acknowledge Input
IR0 - IR7	Interrupt Request Inputs

Functional Diagram

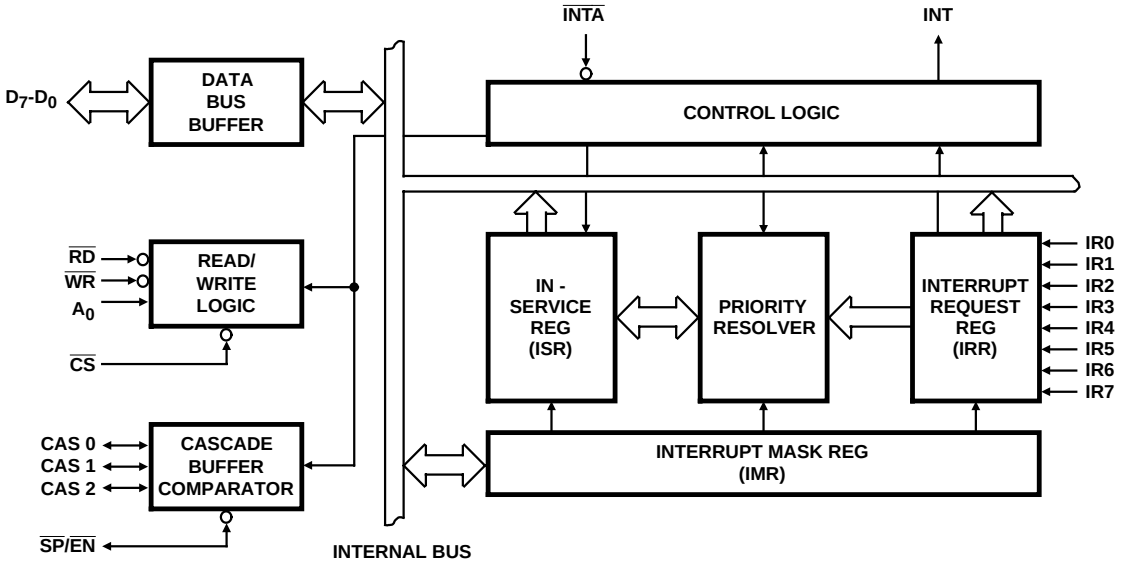


FIGURE 1.

Pin Description

SYMBOL	PIN NUMBER	TYPE	DESCRIPTION
V_{CC}	28	I	V_{CC}: The +5V power supply pin. A 0.1µF capacitor between pins 28 and 14 is recommended for decoupling.
GND	14	I	GROUND
$\overline{CS}$	1	I	CHIP SELECT: A low on this pin enables $\overline{RD}$ and $\overline{WR}$ communications between the CPU and the 82C59A. $\overline{INTA}$ functions are independent of $\overline{CS}$.
$\overline{WR}$	2	I	WRITE: A low on this pin when $\overline{CS}$ is low enables the 82C59A to accept command words from the CPU.
$\overline{RD}$	3	I	READ: A low on this pin when $\overline{CS}$ is low enables the 82C59A to release status onto the data bus for the CPU.
D7 - D0	4 - 11	I/O	BIDIRECTIONAL DATA BUS: Control, status, and interrupt-vector information is transferred via this bus.
CAS0 - CAS2	12, 13, 15	I/O	CASCADE LINES: The CAS lines form a private 82C59A bus to control a multiple 82C59A structure. These pins are outputs for a master 82C59A and inputs for a slave 82C59A.
$\overline{SP}/\overline{EN}$	16	I/O	SLAVE PROGRAM/ENABLE BUFFER: This is a dual function pin. When in the Buffered Mode it can be used as an output to control buffer transceivers ($\overline{EN}$). When not in the Buffered Mode it is used as an input to designate a master ($\overline{SP}$ = 1) or slave ($\overline{SP}$ = 0).
INT	17	O	INTERRUPT: This pin goes high whenever a valid interrupt request is asserted. It is used to interrupt the CPU, thus, it is connected to the CPU's interrupt pin.
IR0 - IR7	18 - 25	I	INTERRUPT REQUESTS: Asynchronous inputs. An interrupt request is executed by raising an IR input (low to high), and holding it high until it is acknowledged (Edge Triggered Mode), or just by a high level on an IR input (Level Triggered Mode). Internal pull-up resistors are implemented on IR0 - 7.
$\overline{INTA}$	26	I	INTERRUPT ACKNOWLEDGE: This pin is used to enable 82C59A interrupt-vector data onto the data bus by a sequence of interrupt acknowledge pulses issued by the CPU.
A0	27	I	ADDRESS LINE: This pin acts in conjunction with the $\overline{CS}$, $\overline{WR}$, and $\overline{RD}$ pins. It is used by the 82C59A to decipher various Command Words the CPU writes and status the CPU wishes to read. It is typically connected to the CPU A0 address line (A1 for 80C86/88/286).

Functional Description

Interrupts in Microcomputer Systems

Microcomputer system design requires that I/O devices such as keyboards, displays, sensors and other components receive servicing in an efficient manner so that large amounts of the total system tasks can be assumed by the microcomputer with little or no effect on throughput.

The most common method of servicing such devices is the Polled approach. This is where the processor must test each device in sequence and in effect “ask” each one if it needs servicing. It is easy to see that a large portion of the main program is looping through this continuous polling cycle and that such a method would have a serious, detrimental effect on system throughput, thus, limiting the tasks that could be assumed by the microcomputer and reducing the cost effectiveness of using such devices.

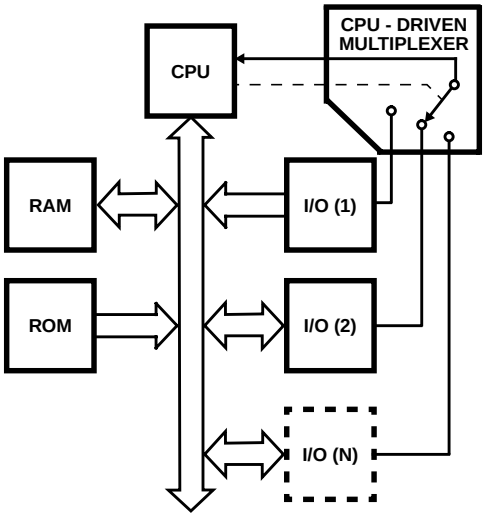


FIGURE 2. POLLED METHOD

A more desirable method would be one that would allow the microprocessor to be executing its main program and only stop to service peripheral devices when it is told to do so by the device itself. In effect, the method would provide an external asynchronous input that would inform the processor that it should complete whatever instruction that is currently being executed and fetch a new routine that will service the requesting device. Once this servicing is complete, however, the processor would resume exactly where it left off.

This is the Interrupt-driven method. It is easy to see that system throughput would drastically increase, and thus, more tasks could be assumed by the microcomputer to further enhance its cost effectiveness.

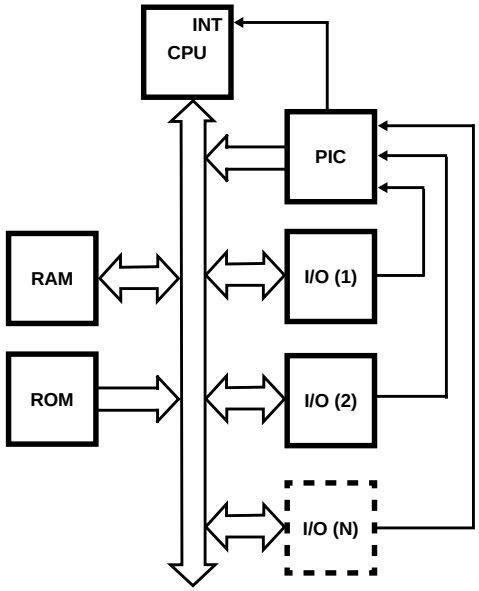


FIGURE 3. INTERRUPT METHOD

The Programmable Interrupt Controller (PIC) functions as an overall manager in an Interrupt-Driven system. It accepts requests from the peripheral equipment, determines which of the incoming requests is of the highest importance (priority), ascertains whether the incoming request has a higher priority value than the level currently being serviced, and issues an interrupt to the CPU based on this determination.

Each peripheral device or structure usually has a special program or "routine" that is associated with its specific functional or operational requirements; this is referred to as a "service routine". The PIC, after issuing an interrupt to the CPU, must somehow input information into the CPU that can "point" the Program Counter to the service routine associated with the requesting device. This "pointer" is an address in a vectoring table and will often be referred to, in this document, as vectoring data.

82C59A Functional Description

The 82C59A is a device specifically designed for use in real time, interrupt driven microcomputer systems. It manages eight levels of requests and has built-in features for expandability to other 82C59As (up to 64 levels). It is programmed by system software as an I/O peripheral. A selection of priority modes is available to the programmer so that the manner in which the requests are processed by the 82C59A can be configured to match system requirements. The priority modes can be changed or reconfigured dynamically at any time during main program operation. This means that the complete interrupt structure can be defined as required, based on the total system environment.

Interrupt Request Register (IRR) and In-Service Register (ISR)

The interrupts at the IR input lines are handled by two registers in cascade, the Interrupt Request Register (IRR) and the In-Service Register (ISR). The IRR is used to indicate all the interrupt levels which are requesting service, and the ISR is used to store all the interrupt levels which are currently being serviced.

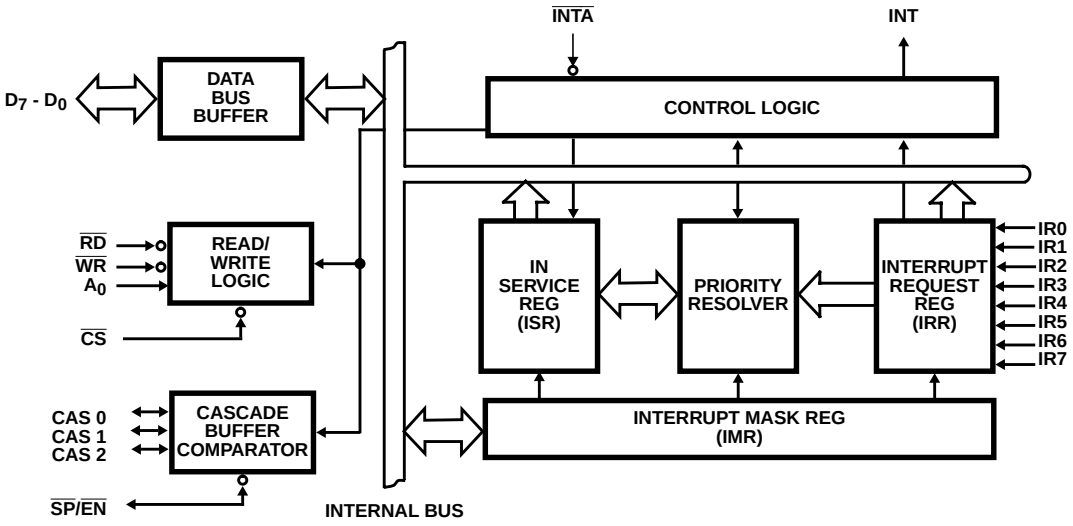


FIGURE 4. 82C59A FUNCTIONAL DIAGRAM

Priority Resolver

This logic block determines the priorities of the bits set in the IRR. The highest priority is selected and strobed into the corresponding bit of the ISR during the $\overline{INTA}$ sequence.

Interrupt Mask Register (IMR)

The IMR stores the bits which disable the interrupt lines to be masked. The IMR operates on the output of the IRR. Masking of a higher priority input will not affect the interrupt request lines of lower priority.

Interrupt (INT)

This output goes directly to the CPU interrupt input. The VOH level on this line is designed to be fully compatible with the 8080, 8085, 8086/88, 80C86/88, 80286, and 80C286 input levels.

Interrupt Acknowledge ($\overline{INTA}$)

$\overline{INTA}$ pulses will cause the 82C59A to release vectoring information onto the data bus. The format of this data depends on the system mode (μPM) of the 82C59A.

Data Bus Buffer

This 3-state, bidirectional 8-bit buffer is used to interface the 82C59A to the System Data Bus. Control words and status information are transferred through the Data Bus Buffer.

Read/Write Control Logic

The function of this block is to accept output commands from the CPU. It contains the Initialization Command Word (ICW) registers and Operation Command Word (OCW) registers which store the various control formats for device operation. This function block also allows the status of the 82C59A to be transferred onto the Data Bus.

Chip Select ($\overline{CS}$)

A LOW on this input enables the 82C59A. No reading or writing of the device will occur unless the device is selected.

Write ($\overline{WR}$)

A LOW on this input enables the CPU to write control words (ICWs and OCWs) to the 82C59A.

Read ($\overline{RD}$)

A LOW on this input enables the 82C59A to send the status of the Interrupt Request Register (IRR), In-Service Register (ISR), the Interrupt Mask Register (IMR), or the interrupt level (in the poll mode) onto the Data Bus.

A0

This input signal is used in conjunction with $\overline{WR}$ and $\overline{RD}$ signals to write commands into the various command registers, as well as to read the various status registers of the chip. This line can be tied directly to one of the system address lines.

The Cascade Buffer/Comparator

This function block stores and compares the IDs of all 82C59As used in the system. The associated three I/O pins (CAS0 - 2) are outputs when the 82C59A is used as a master and are inputs when the 82C59A is used as a slave. As a master, the 82C59A sends the ID of the interrupting slave device onto the CAS0 - 2 lines. The slave, thus selected will send its preprogrammed subroutine address onto the Data Bus during the next one or two consecutive $\overline{INTA}$ pulses. (See section “Cascading the 82C59A”.)

Interrupt Sequence

The powerful features of the 82C59A in a microcomputer system are its programmability and the interrupt routine addressing capability. The latter allows direct or indirect jumping to the specified interrupt routine requested without any polling of the interrupting devices. The normal sequence of events during an interrupt depends on the type of CPU being used.

These events occur in an 8080/8085 system:

1. One or more of the INTERRUPT REQUEST lines (IR0 - IR7) are raised high, setting the corresponding IRR bit(s).
2. The 82C59A evaluates those requests in the priority resolver and sends an interrupt (INT) to the CPU, if appropriate.
3. The CPU acknowledges the INT and responds with an $\overline{INTA}$ pulse.
4. Upon receiving an $\overline{INTA}$ from the CPU group, the highest priority ISR bit is set, and the corresponding IRR bit is reset. The 82C59A will also release a CALL instruction code (11001101) onto the 8-bit data bus through D0 - D7.
5. This CALL instruction will initiate two additional $\overline{INTA}$ pulses to be sent to 82C59A from the CPU group.
6. These two $\overline{INTA}$ pulses allow the 82C59A to release its preprogrammed subroutine address onto the data bus. The lower 8-bit address is released at the first $\overline{INTA}$ pulse and the higher 8-bit address is released at the second $\overline{INTA}$ pulse.
7. This completes the 3-byte CALL instruction released by the 82C59A. In the AEOI mode, the ISR bit is reset at the end of the third $\overline{INTA}$ pulse. Otherwise, the ISR bit remains set until an appropriate EOI command is issued at the end of the interrupt sequence.

The events occurring in an 80C86/88/286 system are the same until step 4.

4. The 82C59A does not drive the data bus during the first $\overline{INTA}$ pulse.
5. The 80C86/88/286 CPU will initiate a second $\overline{INTA}$ pulse. During this $\overline{INTA}$ pulse, the appropriate ISR bit is set and the corresponding bit in the IRR is reset. The 82C59A outputs the 8-bit pointer onto the data bus to be read by the CPU.

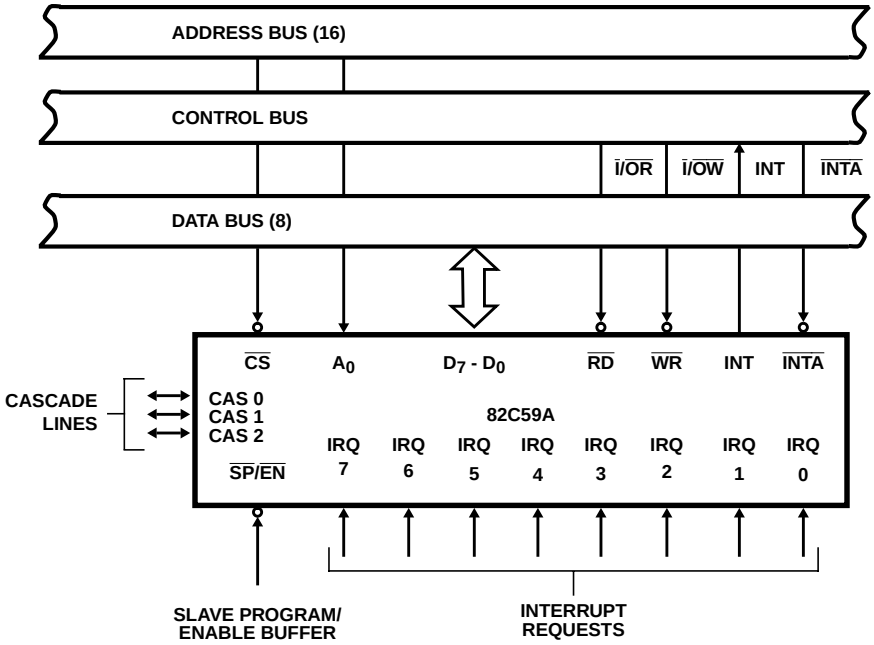


FIGURE 5. 82C59A STANDARD SYSTEM BUS INTERFACE

6. This completes the interrupt cycle. In the AEOI mode, the ISR bit is reset at the end of the second $\overline{INTA}$ pulse. Otherwise, the ISR bit remains set until an appropriate EOI command is issued at the end of the interrupt subroutine.

If no interrupt request is present at step 4 of either sequence (i.e., the request was too short in duration), the 82C59A will issue an interrupt level 7. If a slave is programmed on IR bit 7, the CAS lines remain inactive and vector addresses are output from the master 82C59A.

Interrupt Sequence Outputs

8080, 8085 Interrupt Response Mode

This sequence is timed by three $\overline{INTA}$ pulses. During the first $\overline{INTA}$ pulse, the CALL opcode is enabled onto the data bus.

First Interrupt Vector Byte Data: Hex CD

	D7	D6	D5	D4	D3	D2	D1	D0
Call Code	1	1	0	0	1	1	0	1

During the second $\overline{INTA}$ pulse, the lower address of the appropriate service routine is enabled onto the data bus. When interval = 4 bits, A5 - A7 are programmed, while A0 - A4 are automatically inserted by the 82C59A. When interval = 8, only A6 and A7 are programmed, while A0 - A5 are automatically inserted.

CONTENT OF SECOND INTERRUPT VECTOR BYTE

IR	INTERVAL = 4							
	D7	D6	D5	D4	D3	D2	D1	D0
7	A7	A6	A5	1	1	1	0	0
6	A7	A6	A5	1	1	0	0	0
5	A7	A6	A5	1	0	1	0	0
4	A7	A6	A5	1	0	0	0	0
3	A7	A6	A5	0	1	1	0	0
2	A7	A6	A5	0	1	0	0	0
1	A7	A6	A5	0	0	1	0	0
0	A7	A6	A5	0	0	0	0	0

IR	INTERVAL = 8							
	D7	D6	DS	D4	D3	D2	D1	D0
7	A7	A6	1	1	1	0	0	0
6	A7	A6	1	1	0	0	0	0
5	A7	A6	1	0	1	0	0	0
4	A7	A6	1	0	0	0	0	0
3	A7	A6	0	1	1	0	0	0
2	A7	A6	0	1	0	0	0	0
1	A7	A6	0	0	1	0	0	0
0	A7	A6	0	0	0	0	0	0

During the third $\overline{INTA}$ pulse, the higher address of the appropriate service routine, which was programmed as byte 2 of the initialization sequence (A8 - A15), is enabled onto the bus.

CONTENT OF THIRD INTERRUPT VECTOR BYTE

D7	D6	D5	D4	D3	D2	D1	D0
A15	A14	A13	A12	A11	A10	A9	A8

80C86, 8OC88, 80C286 Interrupt Response Mode

80C86/88/286 mode is similar to 8080/85 mode except that only two Interrupt Acknowledge cycles are issued by the processor and no CALL opcode is sent to the processor. The first interrupt acknowledge cycle is similar to that of 8080/85 systems in that the 82C59A uses it to internally freeze the state of the interrupts for priority resolution and, as a master, it issues the interrupt code on the cascade lines. On this first cycle, it does not issue any data to the processor and leaves its data bus buffers disabled. On the second interrupt acknowledge cycle in the 86/88/286 mode, the master (or slave if so programmed) will send a byte of data to the processor with the acknowledged interrupt code composed as follows (note the state of the ADI mode control is ignored and A5 - A11 are unused in the 86/88/286 mode).

CONTENT OF INTERRUPT VECTOR BYTE FOR 80C86/88/286 SYSTEM MODE

	D7	D6	D5	D4	D3	D2	D1	D0
IR7	T7	T6	T5	T4	T3	1	1	1
IR6	T7	T6	T5	T4	T3	1	1	0
IR5	T7	T6	T5	T4	T3	1	0	1
IR4	T7	T6	T5	T4	T3	1	0	0
IR3	T7	T6	T5	T4	T3	0	1	1
IR2	T7	T6	T5	T4	T3	0	1	0
IR1	T7	T6	T5	T4	T3	0	0	1
IR0	T7	T6	T5	T4	T3	0	0	0

Programming the 82C59A

The 82C59A accepts two types of command words generated by the CPU:

1. **Initialization Command Words (ICWs):** Before normal operation can begin, each 82C59A in the system must be brought to a starting point - by a sequence of 2 to 4 bytes timed by $\overline{WR}$ pulses.
2. **Operation Command Words (OCWs):** These are the command words which command the 82C59A to operate in various interrupt modes. Among these modes are:
   a. Fully nested mode.
   b. Rotating priority mode.
   c. Special mask mode.
   d. Polled mode.

The OCWs can be written into the 82C59A anytime after initialization.

Initialization Command Words (ICWs)

General

Whenever a command is issued with A0 = 0 and D4 = 1, this is interpreted as Initialization Command Word 1 (lCW1). lCW1 starts the initialization sequence during which the following automatically occur:

a. The edge sense circuit is reset, which means that following initialization, an interrupt request (IR) input must make a low-to-high transition to generate an interrupt.

b. The Interrupt Mask Register is cleared.

c. IR7 input is assigned priority 7.

d. Special Mask Mode is cleared and Status Read is set to lRR.

e. If lC4 = 0, then all functions selected in lCW4 are set to zero. (Non-Buffered mode (see note), no Auto-EOI, 8080/85 system).

NOTE: Master/Slave in ICW4 is only used in the buffered mode.

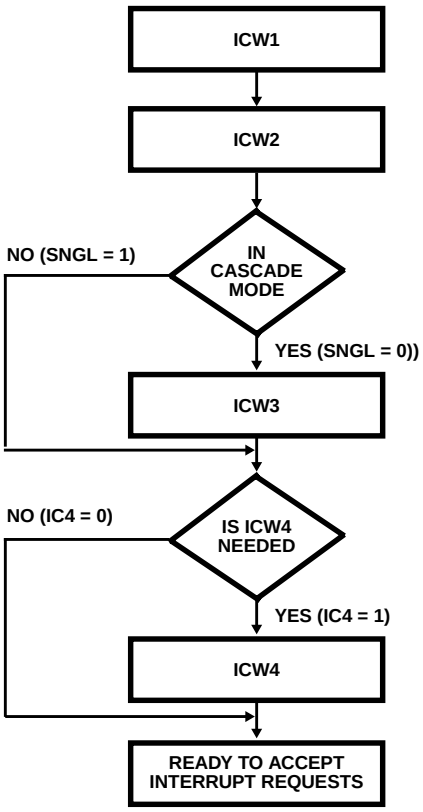


FIGURE 6. 82C59A INITIALIZATION SEQUENCE

Initialization Command Words 1 and 2 (ICW1, ICW2)

A5 - A15: Page starting address of service routines. In an 8080/85 system the 8 request levels will generate CALLS to 8 locations equally spaced in memory. These can be programmed to be spaced at intervals of 4 or 8 memory locations, thus, the 8 routines will occupy a page of 32 or 64 bytes, respectively.

ICW1

A_0	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	A_7	A_6	A_5	1	LTIM	ADI	SNGL	IC4

- IC4: 1 = ICW4 needed; 0 = No ICW4 needed
- SNGL: 1 = Single; 0 = Cascade Mode
- ADI: CALL address interval; 1 = Interval of 4; 0 = Interval of 8
- LTIM: 1 = Level triggered mode; 0 = Edge triggered mode
- A_7 - A_5: A_7 - A_5 of Interrupt vector address (MCS-80/85 mode only)

ICW2

A_0	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
1	A_{15} / T_7	A_{14} / T_6	A_{13} / T_5	A_{12} / T_4	A_{11} / T_3	A_{10}	A_9	A_8

- A_{15} - A_8: A_{15} - A_8 of interrupt vector address (MCS80/85 mode)
- T_7 - T_3: T_7 - T_3 of interrupt vector address (8086/8088 mode)

ICW3 (MASTER DEVICE)

A_0	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
1	S_7	S_6	S_5	S_4	S_3	S_2	S_1	S_0

- S_7 - S_0: 1 = IR input has a slave; 0 = IR input does not have a slave

ICW3 (SLAVE DEVICE)

A_0	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
1	0	0	0	0	0	ID_2	ID_1	ID_0

SLAVE ID (NOTE)

0	1	2	3	4	5	6	7
0	1	0	1	0	1	0	1
0	0	1	1	0	0	1	1
0	0	0	0	1	1	1	1

ICW4

A_0	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
1	0	0	0	SFNM	BUF	M/S	AEOI	µPM

- µPM: 1 = 8086/8088 mode; 0 = MCS-80/85 mode
- AEOI: 1 = Auto EOI; 0 = Normal EOI
- BUF / M/S:

BUF	M/S	
0	X	- Non buffered mode
1	0	- Buffered mode slave
1	1	- Buffered mode master

- SFNM: 1 = Special fully nested moded; 0 = Not special fully nested mode

NOTE: Slave ID is equal to the corresponding master IR input.

FIGURE 7. 82C59A INITIALIZATION COMMAND WORD FORMAT

The address format is 2 bytes long (A0 - A15). When the routine interval is 4, A0 - A4 are automatically inserted by the 82C59A, while A5 - A15 are programmed externally. When the routine interval is 8, A0 - A5 are automatically inserted by the 82C59A while A6 - A15 are programmed externally.

The 8-byte interval will maintain compatibility with current software, while the 4-byte interval is best for a compact jump table.

In an 80C86/88/286 system, A15 - A11 are inserted in the five most significant bits of the vectoring byte and the 82C59A sets the three least significant bits according to the interrupt level. A10 - A5 are ignored and ADI (Address interval) has no effect.

LTIM: If LTIM = 1, then the 82C59A will operate in the level interrupt mode. Edge detect logic on the interrupt inputs will be disabled.

ADI: ALL address interval. ADI = 1 then interval = 4; ADI = 0 then interval = 8.

SNGL: Single. Means that this is the only 82C59A in the system. If SNGL = 1, no ICW3 will be issued.

IC4: If this bit is set - ICW4 has to be issued. If ICW4 is not needed, set IC4 = 0.

Initialization Command Word 3 (ICW3)

This word is read only when there is more than one 82C59A in the system and cascading is used, in which case SNGL = 0. It will load the 8-bit slave register. The functions of this register are:

a. In the master mode (either when $\overline{SP}$ = 1, or in buffered mode when M/S = 1 in ICW4) a "1" is set for each slave in the bit corresponding to the appropriate IR line for the slave. The master then will release byte 1 of the call sequence (for 8080/85 system) and will enable the corresponding slave to release bytes 2 and 3 (for 80C86/88/286, only byte 2) through the cascade lines.

b. In the slave mode (either when $\overline{SP}$ = 0, or if BUF = 1 and M/S = 0 in ICW4), bits 2 - 0 identify the slave. The slave compares its cascade input with these bits and if they are equal, bytes 2 and 3 of the call sequence (or just byte 2 for 80C86/88/286) are released by it on the Data Bus.

NOTE: (The slave address must correspond to the IR line it is connected to in the master ID).

Initialization Command Word 4 (ICW4)

SFNM: If SFNM = 1, the special fully nested mode is programmed.

BUF: If BUF = 1, the buffered mode is programmed. In buffered mode, $\overline{SP}/\overline{EN}$ becomes an enable output and the master/slave determination is by M/S.

M/S: If buffered mode is selected: M/S = 1 means the 82C59A is programmed to be a master, M/S = 0 means the 82C59A is programmed to be a slave. If BUF = 0, M/S has no function.

AEOI: If AEOI = 1, the automatic end of interrupt mode is programmed.

µPM: Microprocessor mode: µPM = 0 sets the 82C59A for 8080/85 system operation, µPM = 1 sets the 82C59A for 80C86/88/286 system operation.

Operation Command Words (OCWs)

After the Initialization Command Words (ICWs) are programmed into the 82C59A, the device is ready to accept interrupt requests at its input lines. However, during the 82C59A operation, a selection of algorithms can command the 82C59A to operate in various modes through the Operation Command Words (OCWs).

OPERATION COMMAND WORDS (OCWs)

A0	D7	D6	D5	D4	D3	D2	D1	D0
OCW1								
1	M7	M6	M5	M4	M3	M2	M1	M0
OCW2								
0	R	SL	EOI	0	0	L2	L1	L0
OCW3								
0	0	ESMM	SMM	0	1	P	RR	RIS

Operation Command Word 1 (OCW1)

OCW1 sets and clears the mask bits in the Interrupt Mask Register (IMR) M7 - M0 represent the eight mask bits. M = 1 indicates the channel is masked (inhibited), M = 0 indicates the channel is enabled.

Operation Command Word 2 (OCW2)

R, SL, EOI - These three bits control the Rotate and End of Interrupt modes and combinations of the two. A chart of these combinations can be found on the Operation Command Word Format.

L2, L1, L0 - These bits determine the interrupt level acted upon when the SL bit is active.

Operation Command Word 3 (OCW3)

ESMM - Enable Special Mask Mode. When this bit is set to 1 it enables the SMM bit to set or reset the Special Mask Mode. When ESMM = 0, the SMM bit becomes a "don't care".

SMM - Special Mask Mode. If ESMM = 1 and SMM = 1, the 82C59A will enter Special Mask Mode. If ESMM = 1 and SMM = 0, the 82C59A will revert to normal mask mode. When ESMM = 0, SMM has no effect.

Fully Nested Mode

This mode is entered after initialization unless another mode is programmed. The interrupt requests are ordered in priority from 0 through 7 (0 highest). When an interrupt is acknowledged the highest priority request is determined and its vector placed on the bus. Additionally, a bit of the Interrupt Service register (IS0 - 7) is set. This bit remains set until the microprocessor issues an End of Interrupt (EOI) command

immediately before returning from the service routine, or if the AEOI (Automatic End of Interrupt) bit is set, until the trailing edge of the last $\overline{INTA}$. While the IS bit is set, all further interrupts of the same or lower priority are inhibited, while higher levels will generate an interrupt (which will be acknowledged only if the microprocessor internal interrupt enable flip-flop has been re-enabled through software). After the initialization sequence, IR0 has the highest priority and IR7 the lowest. Priorities can be changed, as will be explained in the rotating priority mode or via the set priority command.

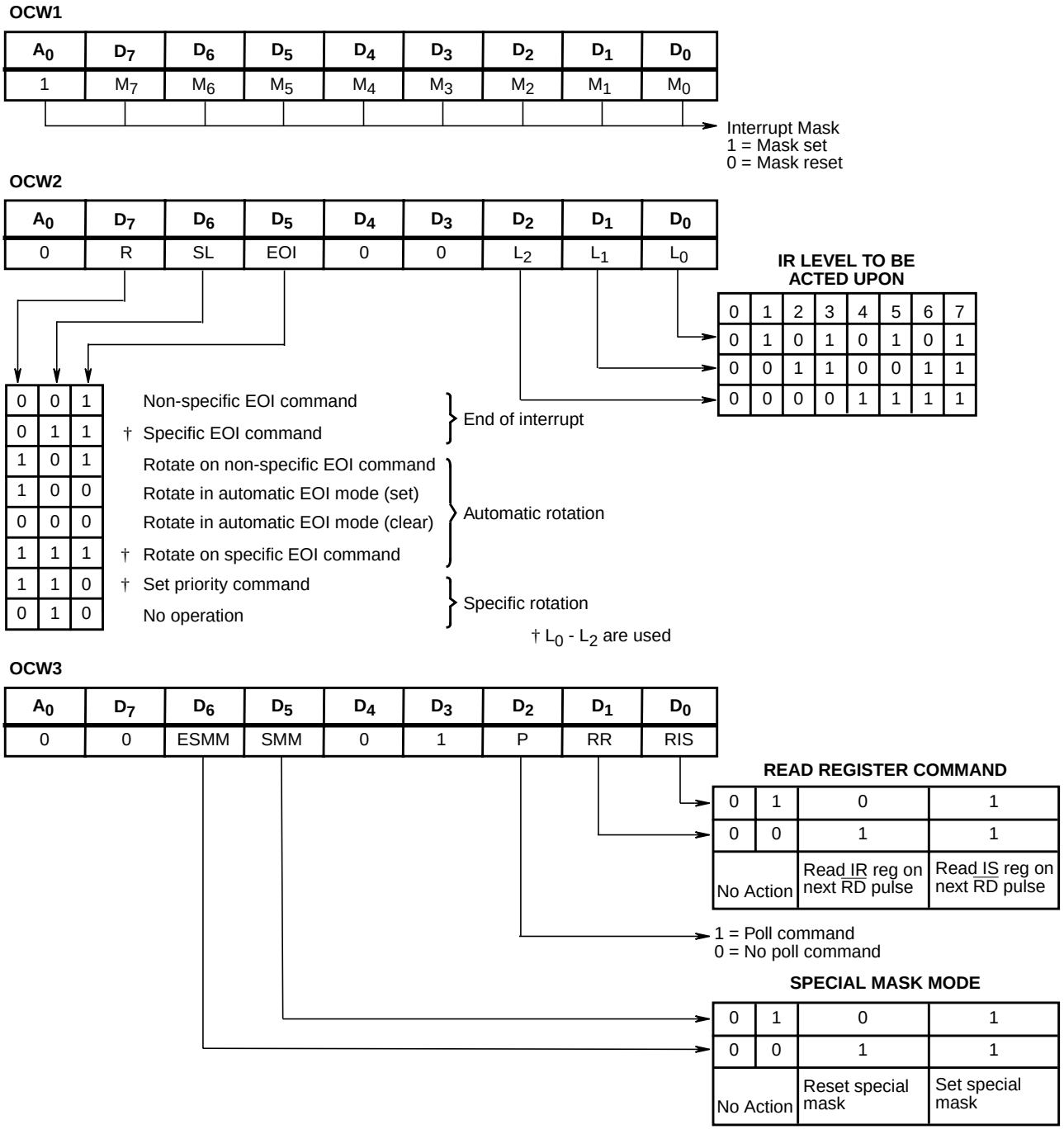


FIGURE 8. 82C59A OPERATION COMMAND WORD FORMAT

End of Interrupt (EOI)

The In-Service (IS) bit can be reset either automatically following the trailing edge of the last in sequence $\overline{INTA}$ pulse (when AEOI bit in lCW1 is set) or by a command word that must be issued to the 82C59A before returning from a service routine (EOI Command). An EOI command must be issued twice if servicing a slave in the Cascade mode, once for the master and once for the corresponding slave.

There are two forms of EOI command: Specific and Non-Specific. When the 82C59A is operated in modes which preserve the fully nested structure, it can determine which IS bit to reset on EOI. When a Non-Specific command is issued the 82C59A will automatically reset the highest IS bit of those that are set, since in the fully nested mode the highest IS level was necessarily the last level acknowledged and serviced. A non-specific EOI can be issued with OCW2 (EOI = 1, SL = 0, R = 0).

When a mode is used which may disturb the fully nested structure, the 82C59A may no longer be able to determine the last level acknowledged. In this case a Specific End of Interrupt must be issued which includes as part of the command the IS level to be reset. A specific EOI can be issued with OCW2 (EOI = 1, SL = 1, R = 0, and L0 - L2 is the binary level of the IS bit to be reset).

An lRR bit that is masked by an lMR bit will not be cleared by a non-specific EOI if the 82C59A is in the Special Mask Mode.

Automatic End of Interrupt (AEOI) Mode

If AEOI = 1 in lCW4, then the 82C59A will operate in AEOI mode continuously until reprogrammed by lCW4. In this mode the 82C59A will automatically perform a non-specific EOI operation at the trailing edge of the last interrupt acknowledge pulse (third pulse in 8080/85, second in 80C86/88/286). Note that from a system standpoint, this mode should be used only when a nested multilevel interrupt structure is not required within a single 82C59A.

Automatic Rotation (Equal Priority Devices)

In some applications there are a number of interrupting devices of equal priority. In this mode a device, after being serviced, receives the lowest priority, so a device requesting an interrupt will have to wait, in the worst case until each of 7 other devices are serviced at most once. For example, if the priority and "in service" status is:

Before Rotate (lR4 the highest priority requiring service)

	IS7	IS6	IS5	IS4	IS3	IS2	IS1	IS0
"IS" Status	0	1	0	1	0	0	0	0
Priority Status	7	6	5	4	3	2	1	0
	↑ lowest							↑ highest

After Rotate (lR4 was serviced, all other priorities rotated correspondingly)

	IS7	IS6	IS5	IS4	IS3	IS2	IS1	IS0
"IS" Status	0	1	0	0	0	0	0	0
Priority Status	2	1	0	7	6	5	4	3
	↑ highest							↑ lowest

There are two ways to accomplish Automatic Rotation using OCW2, the Rotation on Non-Specific EOI Command (R = 1, SL = 0, EOI = 1) and the Rotate in Automatic EOI Mode which is set by (R = 1, SL = 0, EOI = 0) and cleared by (R = 0, SL = 0, EOl = 0).

Specific Rotation (Specific Priority)

The programmer can change priorities by programming the lowest priority and thus, fixing all other priorities; i.e., if IR5 is programmed as the lowest priority device, then IR6 will have the highest one.

The Set Priority command is issued in OCW2 where: R = 1, SL = 1, L0 - L2 is the binary priority level code of the lowest priority device.

Observe that in this mode internal status is updated by software control during OCW2. However, it is independent of the End of Interrupt (EOI) command (also executed by OCW2). Priority changes can be executed during an EOI command by using the Rotate on Specific EOl command in OCW2 (R = 1, SL = 1, EOI = 1, and L0 - L2 = IR level to receive lowest priority).

Interrupt Masks

Each Interrupt Request input can be masked individually by the Interrupt Mask Register (IMR) programmed through OCW1. Each bit in the lMR masks one interrupt channel if it is set (1). Bit 0 masks IR0, Bit 1 masks IR1 and so forth. Masking an IR channel does not affect the operation of other channels.

Special Mask Mode

Some applications may require an interrupt service routine to dynamically alter the system priority structure during its execution under software control. For example, the routine may wish to inhibit lower priority requests for a portion of its execution but enable some of them for another portion.

The difficulty here is that if an Interrupt Request is acknowledged and an End of Interrupt command did not reset its IS bit (i.e., while executing a service routine), the 82C59A would have inhibited all lower priority requests with no easy way for the routine to enable them.

That is where the Special Mask Mode comes in. In the Special Mask Mode, when a mask bit is set in OCW1, it inhibits further interrupts at that level and enables interrupts from all other levels (lower as well as higher) that are not masked.

Thus, any interrupts may be selectively enabled by loading the mask register.

The Special Mask Mode is set by OCW3 where: ESMM = 1, SMM = 1, and cleared where ESMM = 1, SMM = 0.

Poll Command

In this mode, the INT output is not used or the microprocessor internal Interrupt Enable flip flop is reset, disabling its interrupt input. Service to devices is achieved by software using a Poll command.

The Poll command is issued by setting P = 1 in OCW3. The 82C59A treats the next $\overline{RD}$ pulse to the 82C59A (i.e., $\overline{RD}$ = 0, $\overline{CS}$ = 0) as an interrupt acknowledge, sets the appropriate IS bit if there is a request, and reads the priority level. Interrupt is frozen from $\overline{WR}$ to $\overline{RD}$.

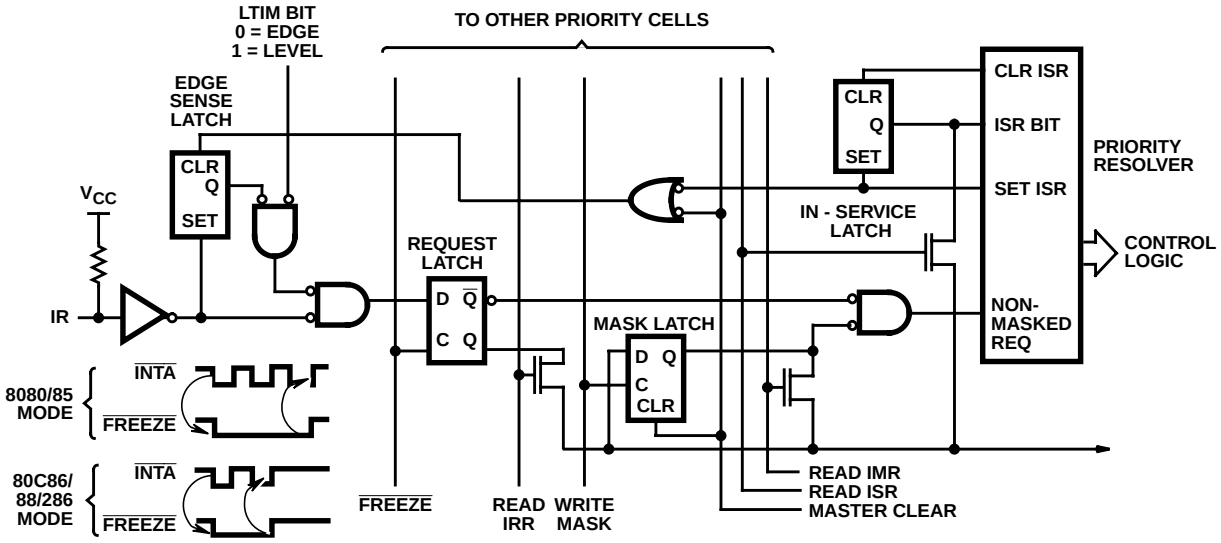
The word enabled onto the data bus during $\overline{RD}$ is:

D7	D6	D5	D4	D3	D2	D1	D0
I	-	-	-	-	W2	W1	W0

W0 - W2: Binary code of the highest priority level requesting service.

I: Equal to a “1” if there is an interrupt.

This mode is useful if there is a routine command common to several levels so that the $\overline{INTA}$ sequence is not needed (saves ROM space). Another application is to use the poll mode to expand the number of priority levels to more than 64.



NOTES:

1. Master clear active only during ICW1.
2. $\overline{FREEZE}$ is active during $\overline{INTA}$ and poll sequence only.
3. Truth Table for D-latch.

C	D	Q	Operation
1	D1	D1	Follow
0	X	Qn-1	Hold

FIGURE 9. PRIORITY CELL - SIMPLIFIED LOGIC DIAGRAM

Reading the 82C59A Status

The input status of several internal registers can be read to update the user information on the system. The following registers can be read via OCW3 (IRR and ISR) or OCW1 (IMR).

Interrupt Request Register (IRR): 8-bit register which contains the levels requesting an interrupt to be acknowledged. The highest request level is reset from the IRR when an interrupt is acknowledged. IRR is not affected by IMR.

In-Service Register (ISR): 8-bit register which contains the priority levels that are being serviced. The ISR is updated when an End of Interrupt Command is issued.

Interrupt Mask Register: 8-bit register which contains the interrupt request lines which are masked.

The IRR can be read when, prior to the $\overline{RD}$ pulse, a Read Register Command is issued with OCW3 (RR = 1, RIS = 0).

The ISR can be read when, prior to the $\overline{RD}$ pulse, a Read Register Command is issued with OCW3 (RR = 1, RIS = 1).

There is no need to write an OCW3 before every status read operation, as long as the status read corresponds with the previous one: i.e., the 82C59A “remembers” whether the IRR or ISR has been previously selected by the OCW3. This is not true when poll is used. In the poll mode, the 82C59A

treats the $\overline{RD}$ following a "poll write" operation as an $\overline{INTA}$. After initialization, the 82C59A is set to IRR.

For reading the IMR, no OCW3 is needed. The output data bus will contain the IMR whenever $\overline{RD}$ is active and A0 = 1 (OCW1). Polling overrides status read when P = 1, RR = 1 in OCW3.

Edge and Level Triggered Modes

This mode is programmed using bit 3 in ICW1.

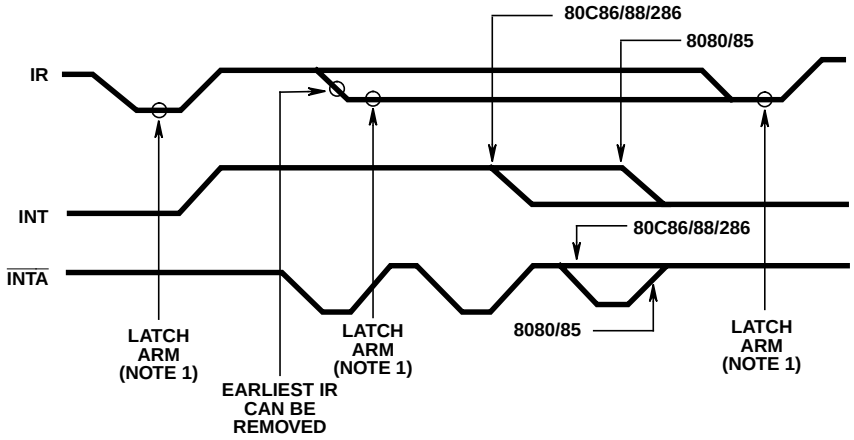
If LTIM = "0", an interrupt request will be recognized by a low to high transition on an IR input. The IR input can remain high without generating another interrupt.

If LTIM = "1", an interrupt request will be recognized by a "high" level on an IR input, and there is no need for an edge detection. The interrupt request must be removed before the EOI command is issued or the CPU interrupt is enabled to prevent a second interrupt from occurring.

The priority cell diagram shows a conceptual circuit of the level sensitive and edge sensitive input circuitry of the 82C59A. Be sure to note that the request latch is a transparent D type latch.

In both the edge and level triggered modes the IR inputs must remain high until after the falling edge of the first $\overline{INTA}$. If the IR input goes low before this time a DEFAULT IR7 will occur when the CPU acknowledges the interrupt. This can be a useful safeguard for detecting interrupts caused by spurious noise glitches on the IR inputs. To implement this feature the IR7 routine is used for "clean up" simply executing a return instruction, thus, ignoring the interrupt. If IR7 is needed for other purposes a default IR7 can still be detected by reading the ISR. A normal IR7 interrupt will set the corresponding ISR bit, a default IR7 won't. If a default IR7 routine occurs during a normal IR7 routine, however, the ISR will remain set. In this case it is necessary to keep track of whether or not the IR7 routine was previously entered. If another IR7 occurs it is a default.

In power sensitive applications, it is advisable to place the 82C59A in the edge-triggered mode with the IR lines normally high. This will minimize the current through the internal pull-up resistors on the IR pins.



NOTE:

1. Edge triggered mode only.

FIGURE 10. IR TRIGGERING TIMING REQUIREMENTS

The Special Fully Nested Mode

This mode will be used in the case of a big system where cascading is used, and the priority has to be conserved within each slave. In this case the special fully nested mode will be programmed to the master (using ICW4). This mode is similar to the normal nested mode with the following exceptions:

a. When an interrupt request from a certain slave is in service, this slave is not locked out from the master's priority logic and further interrupt requests from higher priority IRs within the slave will be recognized by the master and will initiate interrupts to the processor. (In the normal nested mode a slave is masked out when its request is in service and no higher requests from the same slave can be serviced.

b. When exiting the Interrupt Service routine the software has to check whether the interrupt serviced was the only one from that slave. This is done by sending a non-specific End of Interrupt (EOI) command to the slave and then reading its In-Service register and checking for zero. If it is empty, a non-specified EOI can be sent to the master, too. If not, no EOI should be sent.

Buffered Mode

When the 82C59A is used in a large system where bus driving buffers are required on the data bus and the cascading mode is used, there exists the problem of enabling buffers

The buffered mode will structure the 82C59A to send an enable signal on $\overline{SP}/\overline{EN}$ to enable the buffers. In this mode, whenever the 82C59A's data bus outputs are enabled, the $\overline{SP}/\overline{EN}$ output becomes active.

This modification forces the use of software programming to determine whether the 82C59A is a master or a slave. Bit 3 in ICW4 programs the buffered mode, and bit 2 in ICW4 determines whether it is a master or a slave.

Cascade Mode

The 82C59A can be easily interconnected in a system of one master with up to eight slaves to handle up to 64 priority levels.

The master controls the slaves through the 3 line cascade bus (CAS2 - 0). The cascade bus acts like chip selects to the slaves during the $\overline{INTA}$ sequence.

In a cascade configuration, the slave interrupt outputs (INT) are connected to the master interrupt request inputs. When a slave request line is activated and afterwards acknowledged, the master will enable the corresponding slave to release the device routine address during bytes 2 and 3 of $\overline{INTA}$. (Byte 2 only for 80C86/88/286).

The cascade bus lines are normally low and will contain the slave address code from the leading edge of the first $\overline{INTA}$ pulse to the trailing edge of the last $\overline{INTA}$ pulse. Each 82C59A in the system must follow a separate initialization sequence and can be programmed to work in a different mode. An EOI command must be issued twice: once for the master and once for the corresponding slave. Chip select decoding is required to activate each 82C59A.

NOTE: Auto EOI is supported in the slave mode for the 82C59A.

The cascade lines of the Master 82C59A are activated only for slave inputs, non-slave inputs leave the cascade line inactive (low). Therefore, it is necessary to use a slave address of 0 (zero) only after all other addresses are used.

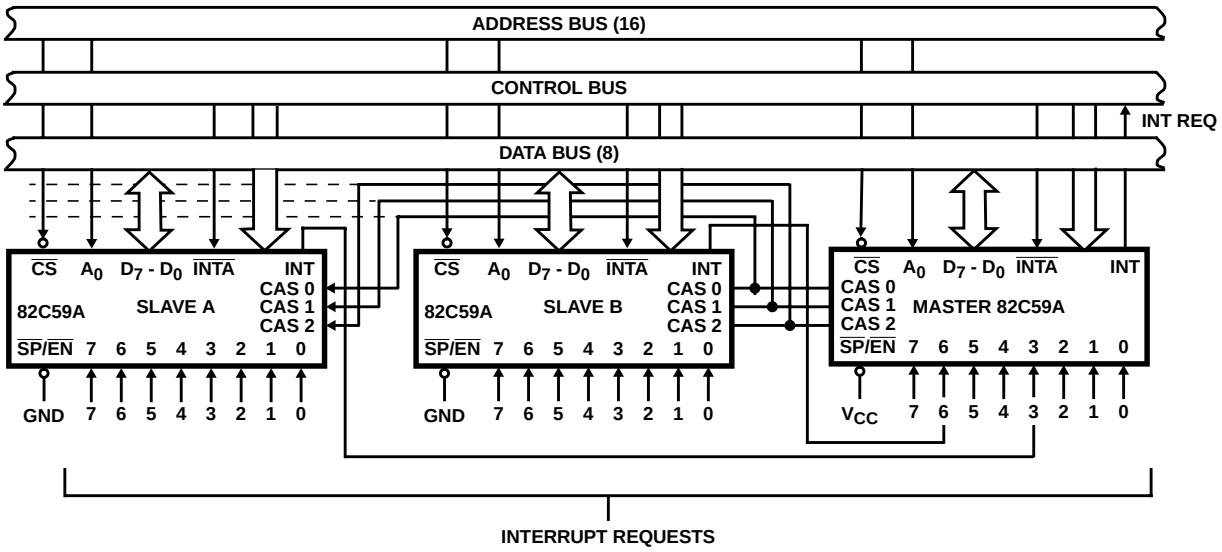


FIGURE 11. CASCADING THE 82C59A

Absolute Maximum Ratings

Supply Voltage . . . . . +8.0V
Input, Output or I/O Voltage . . . . . GND-0.5V to V_{CC}+0.5V
ESD Classification . . . . . Class I

Operating Conditions

Operating Voltage Range . . . . . +4.5V to +5.5V
Operating Temperature Range . . . . . -55^{o}C to +125^{o}C
Input Low Voltage . . . . . 0V to +0.8V

Thermal Information

Thermal Resistance (Typical)	θ_{JA} (^{o}C/W)	θ_{JC} (^{o}C/W)
CERDIP Package	55	12
CLCC Package	65	14
PDIP Package	55	N/A
PLCC Package	65	N/A
SOIC Package	75	N/A

Storage Temperature Range . . . . . -65^{o}C to +150^{o}C
Maximum Junction Temperature Ceramic Package . . . . . +175^{o}C
Maximum Junction Temperature Plastic Package . . . . . +150^{o}C
Maximum Lead Temperature Package (Soldering 10s) . . . . +300^{o}C
(PLCC and SOIC - Lead Tips Only)

Die Characteristics

Gate Count . . . . . 1250 Gates

CAUTION: Stresses above those listed in "Absolute Maximum Ratings" may cause permanent damage to the device. This is a stress only rating and operation of the device at these or any other conditions above those indicated in the operational sections of this specification is not implied.

DC Electrical Specifications

V_{CC} = +5.0V ±10%, T_A = 0^{o}C to +70^{o}C (C82C59A), T_A = -40^{o}C to +85^{o}C (I82C59A), T_A = -55^{o}C to +125^{o}C (M82C59A)

SYMBOL	PARAMETER	MIN	MAX	UNITS	TEST CONDITIONS
V_{IH}	Logical One Input Voltage	2.0 2.2	-	V V	C82C59A, I82C59A M82C59A
V_{IL}	Logical Zero Input Voltage	-	0.8	V	
V_{OH}	Output HIGH Voltage	3.0 VCC -0.4	-	V V	I_{OH} = -2.5mA I_{OH} = -100µA
V_{OL}	Output LOW Voltage	-	0.4	V	I_{OL} = +2.5mA
II	Input Leakage Current	-1.0	+1.0	µA	V_{IN} = GND or V_{CC}, Pins 1-3, 26-27
IO	Output Leakage Current	-10.0	+10.0	µA	V_{OUT} = GND or V_{CC}, Pins 4-13, 15-16
ILIR	IR Input Load Current	- -	-200 10	µA µA	V_{IN} = 0V V_{IN} = V_{CC}
ICCSB	Standby Power Supply Current	-	10	µA	V_{CC} = 5.5V, V_{IN} = V_{CC} or GND Outputs Open, (Note 1)
ICCOP	Operating Power Supply Current	-	1	mA/MHz	V_{CC} = 5.0V, V_{IN} = V_{CC} or GND, Outputs Open, T_A = 25^{o}C, (Note 2)

NOTES:

1. Except for IR0 - IR7 where V_{IN} = V_{CC} or open.
2. ICCOP = 1mA/MHz of peripheral read/write cycle time. (ex: 1.0µs I/O read/write cycle time = 1mA).

Capacitance T_A = +25^{o}C

SYMBOL	PARAMETER	TYP	UNITS	TEST CONDITIONS
CIN	Input Capacitance	15	pF	FREQ = 1MHz, all measurements reference to device GND.
COUT	Output Capacitance	15	pF	
CI/O	I/O Capacitance	15	pF	

AC Electrical Specifications

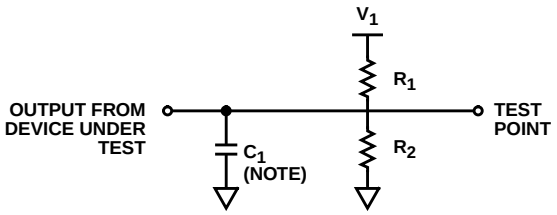
V_{CC} = +5.0V ±10%, GND = 0V, T_A = 0°C to +70°C (C82C59A), T_A -40°C to +85°C (I82C59A), T_A = -55°C to +125°C (M82C59A)

SYMBOL	PARAMETER	82C59A-5		82C59A		82C59A-12		UNITS	TEST CONDITIONS
		MIN	MAX	MIN	MAX	MIN	MAX		
TIMING REQUIREMENTS									
(1) TAHRL	A0/$\overline{CS}$ Setup to $\overline{RD}$/$\overline{INTA}$	10	-	10	-	5	-	ns	
(2) TRHAX	A0/$\overline{CS}$ Hold after $\overline{RD}$/$\overline{INTA}$	5	-	5	-	0	-	ns	
(3) TRLRH	$\overline{RD}$/$\overline{INTA}$ Pulse Width	235	-	160	-	60	-	ns	
(4) TAHWL	A0/$\overline{CS}$ Setup to $\overline{WR}$	0	-	0	-	0	-	ns	
(5) TWHAX	A0/$\overline{CS}$ Hold after $\overline{WR}$	5	-	5	-	0	-	ns	
(6) TWLWH	$\overline{WR}$ Pulse Width	165	-	95	-	60	-	ns	
(7) TDVWH	Data Setup to $\overline{WR}$	240	-	160	-	70	-	ns	
(8) TWHDX	Data Hold after $\overline{WR}$	5	-	5	-	0	-	ns	
(9) TJLJH	Interrupt Request Width Low	100	-	100	-	40	-	ns	
(10) TCVlAL	Cascade Setup to Second or Third $\overline{INTA}$ (Slave Only)	55	-	40	-	30	-	ns	
(11) TRHRL	End of $\overline{RD}$ to next $\overline{RD}$, End of $\overline{INTA}$ (within an $\overline{INTA}$ sequence only)	160	-	160	-	90	-	ns	
(12) TWHWL	End of $\overline{WR}$ to next $\overline{WR}$	190	-	190	-	60	-	ns	
(13) TCHCL (Note 1)	End of Command to next command (not same command type), End of $\overline{INTA}$ sequence to next $\overline{INTA}$ sequence	500	-	400	-	90	-	ns	
TIMING RESPONSES									
(14) TRLDV	Data Valid from $\overline{RD}$/$\overline{INTA}$	-	160	-	120	-	40	ns	1
(15) TRHDZ	Data Float after $\overline{RD}$/$\overline{INTA}$	5	100	5	85	5	22	ns	2
(16) TJHlH	Interrupt Output Delay	-	350	-	300	-	90	ns	1
(17) TlALCV	Cascade Valid from First $\overline{INTA}$ (Master Only)	-	565	-	360	-	50	ns	1
(18) TRLEL	Enable Active from $\overline{RD}$ or $\overline{INTA}$	-	125	-	100	-	40	ns	1
(19) TRHEH	Enable Inactive from $\overline{RD}$ or $\overline{INTA}$	-	60	-	50	-	22	ns	1
(20) TAHDV	Data Valid from Stable Address	-	210	-	200	-	60	ns	1
(21) TCVDV	Cascade Valid to Valid Data	-	300	-	200	-	70	ns	1

NOTE:

1. Worst case timing for TCHCL in an actual microprocessor system is typically greater than the values specified for the 82C59A, (i.e. 8085A = 1.6µs, 8085A -2 = 1µs, 80C86 = 1µs, 80C286 -10 = 131ns, 80C286 -12 = 98ns).

AC Test Circuit

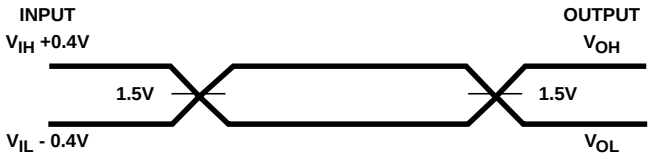


NOTE: Includes stray and jig capacitance.

TEST CONDITION DEFINITION TABLE

TEST CONDITION	V_1	R_1	R_2	C_1
1	1.7V	523Ω	Open	100pF
2	V_{CC}	1.8kΩ	1.8kΩ	50pF

AC Testing Input, Output Waveform



NOTE: AC Testing: All input signals must switch between V_{IL} - 0.4V and V_{IH} + 0.4V. Input rise and fall times are driven at 1ns/V.

Timing Waveforms

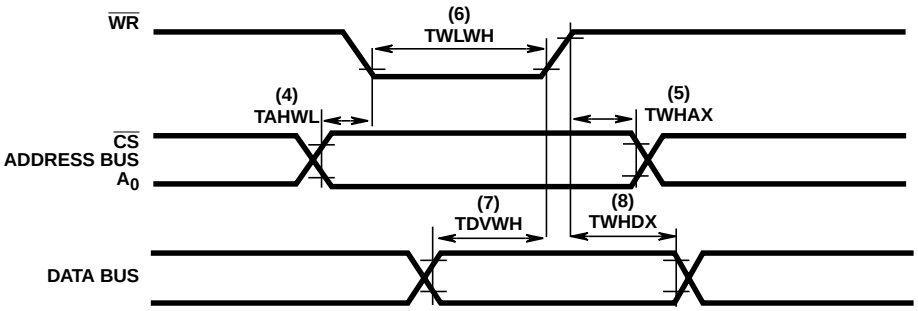


FIGURE 12. WRITE

Timing Waveforms (Continued)

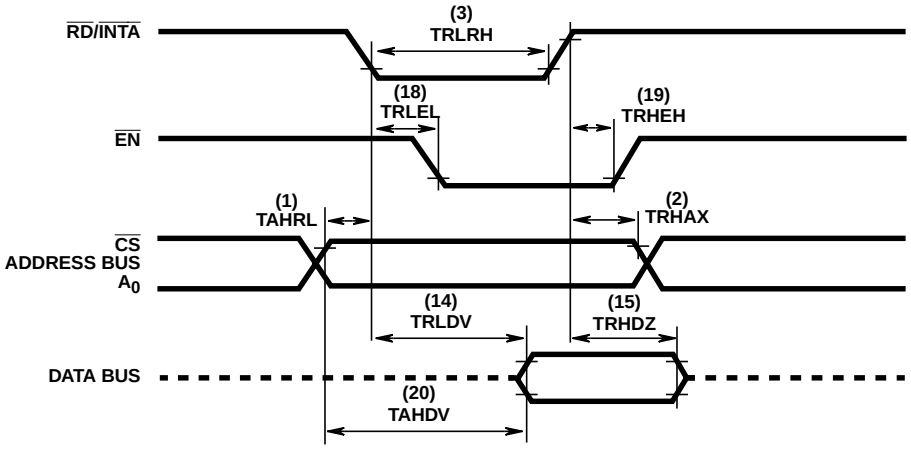


FIGURE 13. RD/INTA

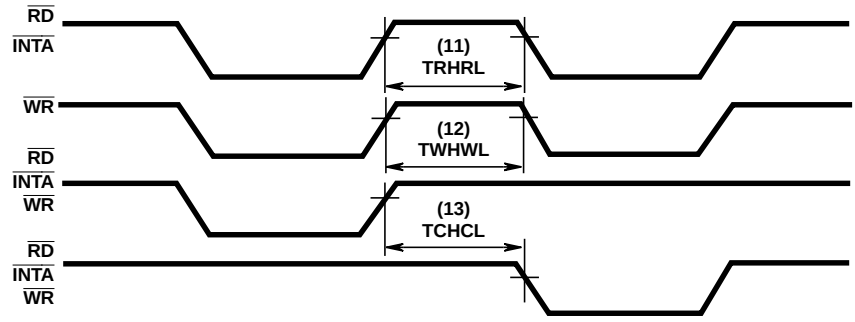
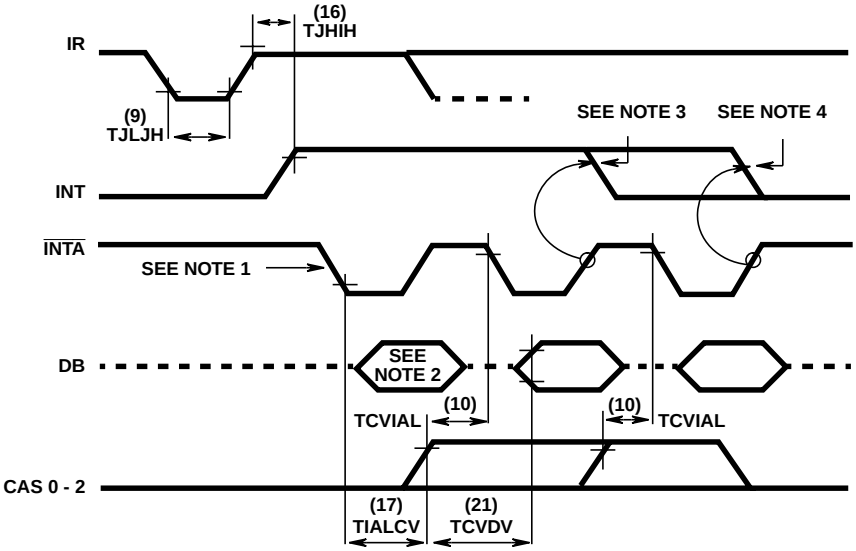


FIGURE 14. OTHER TIMING



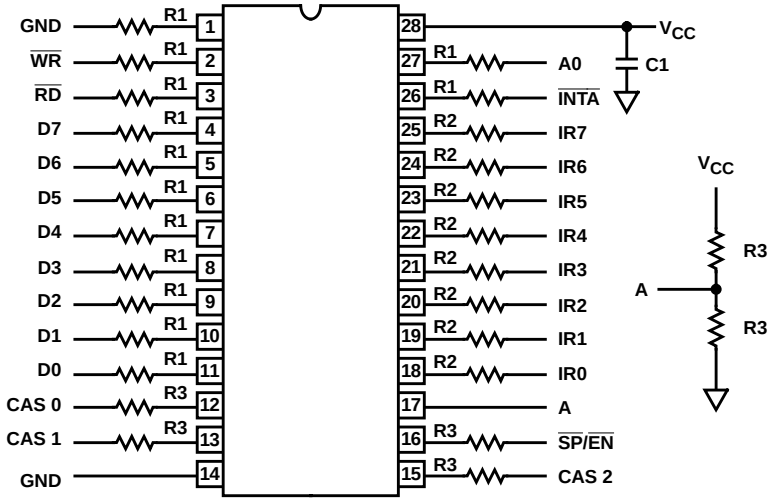
NOTES:

1. Interrupt Request (IR) must remain HIGH until leading edge of first INTA.
2. During first INTA the Data Bus is not active in 80C86/88/286 mode.
3. 80C86/88/286 mode.
4. 8080/8085 mode.

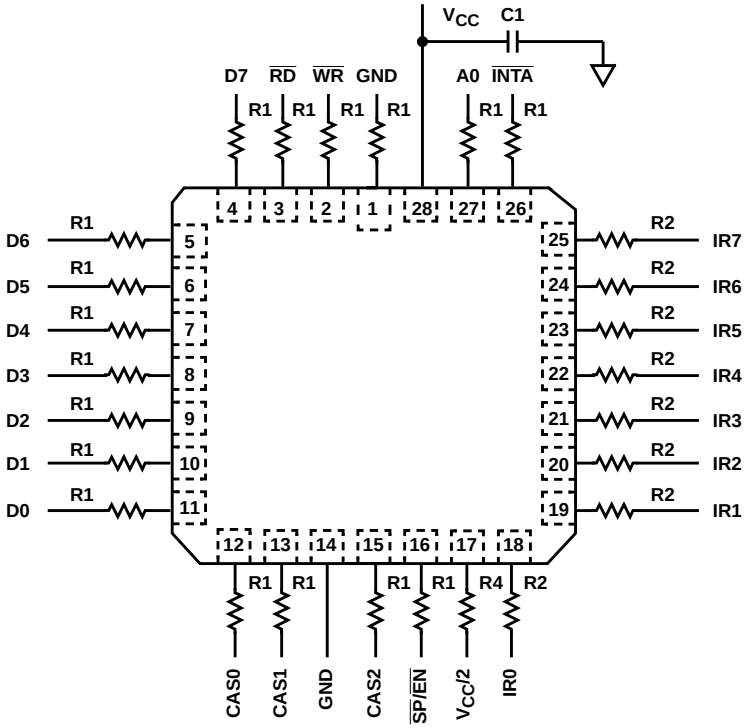
FIGURE 15. INTA SEQUENCE

Burn-In Circuits

MD82C59A CERDIP



MR82C59A CLCC



NOTES:

1. V_{CC} = 5.5V ±0.5V.
2. V_{IH} = 4.5V ±10%.
3. V_{IL} = -0.2V to 0.4V.
4. GND = 0V.
5. R1 = 47kΩ ±5%.
6. R2 = 510Ω ±5%.
7. R3 = 10kΩ ±5%.
8. R4 = 1.2kΩ ±5%.
9. C1 = 0.01µF min.
10. F0 = 100kHz ±10%.
11. F1 = F0/2, F2 = F1/2, ...F8 = F7/2.

Die Characteristics

DIE DIMENSIONS:

143 x 130 x 19 ±1mils
(3630 x 3310 x 525µm)

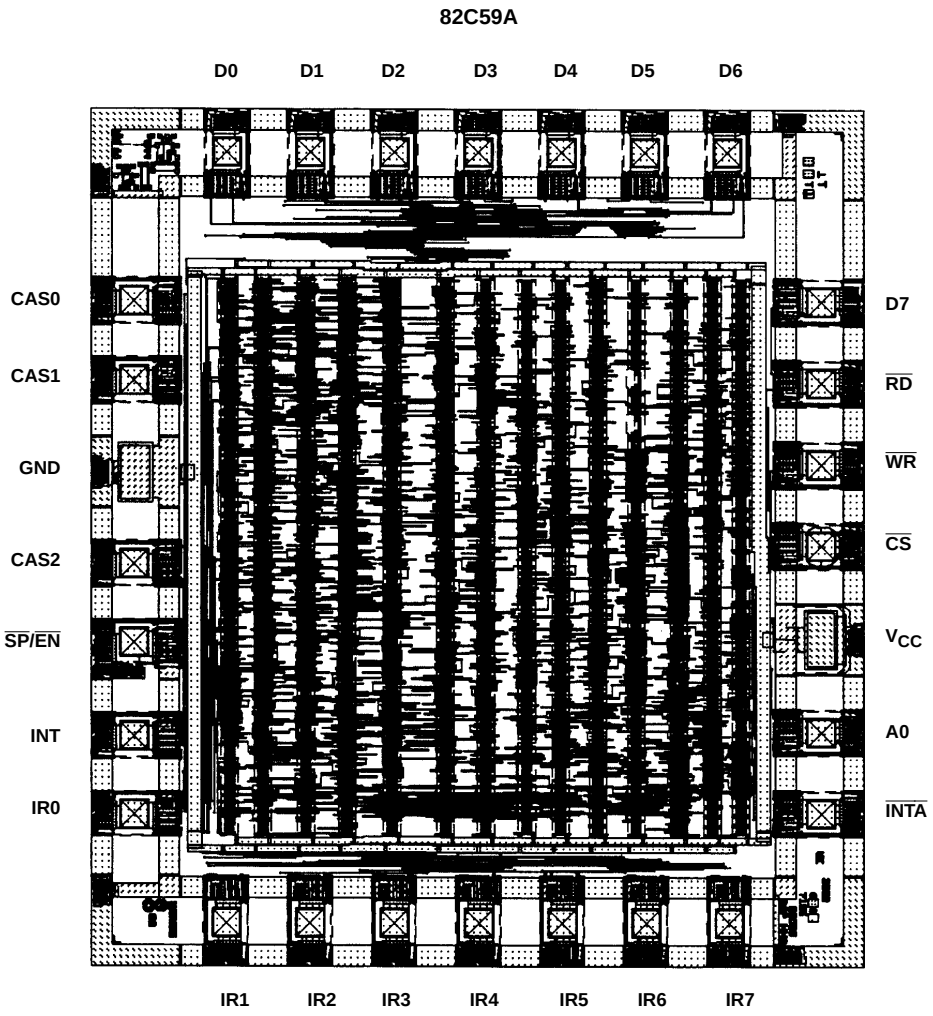
METALLIZATION:

Type: Si-Al-Cu
Thickness: Metal 1: 8kÅ ± 0.75kÅ
Metal 2: 12kÅ ± 1.0kÅ

GLASSIVATION:

Type: Nitrox
Thickness: 10kÅ ± 3.0kÅ

Metallization Mask Layout



PAGE

Introduction

The Intersil 82C59A is a CMOS Priority Interrupt Controller, designed to relieve the system CPU from the task of polling in a multi-level priority interrupt system. The 82C59A is compatible with microprocessors such as the 80C86, 80C88, 8086, 8088, 8080/85 and NSC800.

In the following discussion, we will look at the initialization and operation process for the 82C59A. We will focus our attention on 80C86/80C88-based systems. However, the information presented will also be applicable to use of the 82C59A in 8080 and 8085-based systems as well.

Let us look at the sequence of events that occur with the 82C59A during an interrupt request and service. In an 8080/85 based system:

1. One or more of the INTERRUPT REQUEST lines (IR0 - IR7) are raised high, setting the corresponding bits in the Interrupt Request Register (IRR).
2. The interrupt is evaluated in the priority resolver. If appropriate, an interrupt is sent to the CPU via the INT line (pin 17).
3. The CPU acknowledges the interrupt by sending a pulse on the $\overline{INTA}$ line. Upon reception of this pulse, the 82C59A responds by forcing the opcode for a call instruction (0CDH) onto the data bus.
4. A second $\overline{INTA}$ pulse is sent from the CPU. At this time, the device will respond by placing the lower byte of the address of the appropriate service routine onto the data bus. This address is derived from ICW1.
5. A final (third) pulse of $\overline{INTA}$ occurs, and the 82C59A responds by placing the upper byte of the address onto the data bus. This address is taken from ICW2.
6. The three byte call instruction is then complete. If the AEOI mode has been chosen, the bit set during the first $\overline{INTA}$ pulse in the ISR is reset at the end of the third $\overline{INTA}$ pulse. Otherwise, it will not get reset until an appropriate EOI command is issued to the 82C59A.

For 80C86- and 80C88-based systems:

1. and 2. same as above.
2. The CPU responds to the interrupt request by pulsing the $\overline{INTA}$ line twice. The first pulse sets the appropriate ISR bit and resets the IRR bit while the second pulse causes the interrupt vector to be placed on the data bus. This byte is composed of the interrupt number in bits 0 through 2, and bits 3 through 7 are taken from bits 3 - 7 of ICW2.
3. The interrupt sequence is complete. If using the AEOI mode, the bit set earlier in the ISR will be reset. Otherwise, the interrupt controller will await an appropriate EOI command at the end of the interrupt service routine.

1.0 Glossary of Terms for the 82C59A

1.1 Automatic End of Interrupt (AEOI)

When the 82C59A is programmed to operate in the Automatic EOI mode, the device will produce its own End-of-Interrupt (EOI) at the trailing edge of the last Interrupt Acknowledge pulse ($\overline{INTA}$) from the CPU. Using this mode of operation frees the software (service routines) from needing to send an EOI manually to the 82C59A.

However, using the Automatic EOI mode will upset the priority structure of the 82C59A. When the AEOI is generated, the bit that was set in the In-Service Register (ISR) to indicate which interrupt is being serviced, will be cleared. Because of this, while an interrupt is being serviced there will be no record in the ISR that it is being serviced. Unless interrupts are disabled by the CPU, there is a risk that interrupt requests of lower or equal priority will interrupt the current request being serviced. If this mode of operation is not desired, interrupts should not be re-enabled by the CPU when executing interrupt service routines.

1.2 Automatic Rotation

During normal operation of the 82C59A, we have an assigned order of priorities for the IR lines. There are however, instances when it might be useful to assign equal priorities to all interrupts. Once a particular interrupt has been serviced, all other equal priority interrupts should have an opportunity to be serviced before the original peripheral can be serviced again. This priority equalization can be achieved through Automatic Rotation of priorities.

Assume, for example, that the assigned priorities of interrupts has IR0 as the highest priority interrupt and IR7 as the lowest. Figure 1A shows interrupt requests occurring on IR7 as well as IR3. Because IR3 is of higher priority, it will be serviced first. Upon completion of the servicing of IR3, rotation occurs and IR3 then becomes the lowest priority interrupt. IR4 will now have the highest priority (See Figure 1B).

There are two methods in which Automatic Rotation can be implemented. First, if the 82C59A is operating in the AEOI mode as described above, the 82C59A can be programmed for “Rotate in Automatic EOI mode”. This is done by writing a command word to OCW2. The second method occurs when using normal EOIs. When an EOI is issued by the service routine, the software can specify that rotation be performed.

	IR7	IR6	IR5	IR4	IR3	IR2	IR1	IR0
IRR STATUS	1	0	0	0	1	0	0	0
PRIORITY	7	6	5	4	3	2	1	0
	LOWEST PRIORITY						HIGHEST PRIORITY	

FIGURE 1A. IR PRIORITIES (BEFORE ROTATION)

	IR7	IR6	IR5	IR4	IR3	IR2	IR1	IR0
IRR STATUS	1	0	0	0	0	0	0	0
PRIORITY	3	2	1	0	7	6	5	4
	HIGHEST PRIORITY						LOWEST PRIORITY	

FIGURE 1B. IR PRIORITIES (AFTER ROTATION)

1.3 Buffered Mode

When using the 82C59A in a large system, it may be necessary to use bus buffers to guarantee data integrity and guard against bus contention.

By selecting buffered mode when initializing the device, the $\overline{SP}/\overline{EN}$ pin (pin 16) will generate an enable signal for the buffers whenever the data outputs from the 82C59A are active. In this mode, the dual function $\overline{SP}/\overline{EN}$ pin can no longer be used for specifying whether a particular 82C59A is being used as a master or a slave in the system. This specification must be made through setting the proper bit in ICW4 during the device initialization.

1.4 Cascade Mode

More than one 82C59A can be used in a system to expand the number of priority interrupts to a maximum of 64 levels without adding any additional hardware. This method of expansion is known as “cascading”. An example of cascading 82C59A is shown in Figure 2.

In a cascaded interrupt scheme, a single 82C59A is utilized as the “master” interrupt controller. As many as 8 “slave” 82C59As can be connected to the IR inputs of the “master” 82C59A. Each of these slaves can support up to 8 interrupt inputs, yielding 64 possible prioritized interrupts.

When in cascade mode, the determination of whether a device is a master or a slave can take either of two forms. The state of the $\overline{SP}/\overline{EN}$ pin will select “master” or “slave” mode for a device when the buffered mode is not being used. Should buffered mode be used, then it is necessary that bit D2 (M/S) of ICW4 be set to indicate if the particular 82C59A is being used as a “master” or “slave” interrupt controller in the system.

The CAS0-2 pins on the interrupt controllers serve to provide a private bus for the cascaded 82C59As. These lines allow the “master” to inform the slaves which is to be serviced for a particular interrupt.

1.5 End of Interrupt (EOI)

When an interrupt is recognized and acknowledged by the CPU, its corresponding bit will be set in the In-Service Register (ISR). If the AEOI mode is in use, the bit will be cleared automatically through the interrupt acknowledge signal from the CPU. However, if AEOI is not in effect, it is the task of software to notify the 82C59A when servicing of an interrupt is completed. This is done by issuing an End-of-Interrupt (EOI).

There are 2 different types of EOIs that can be issued to the device; non-specific EOI and specific EOI. In most cases, when the device is operating in a mode that does not disturb the fully nested mode such as Special Fully Nested Mode, we will issue a non-specific EOI. This form of the EOI will automatically reset the highest priority bit set in the ISR. This is because for full nested operation, the highest priority IS bit set is the last interrupt level acknowledged and serviced.

The “specific” EOI is used when the fully nested structure has not been preserved. The 82C59A may not be able to determine the last level acknowledged. Thus, the software must specify which interrupt level is to be reset. This is done by issuing a “specific” EOI.

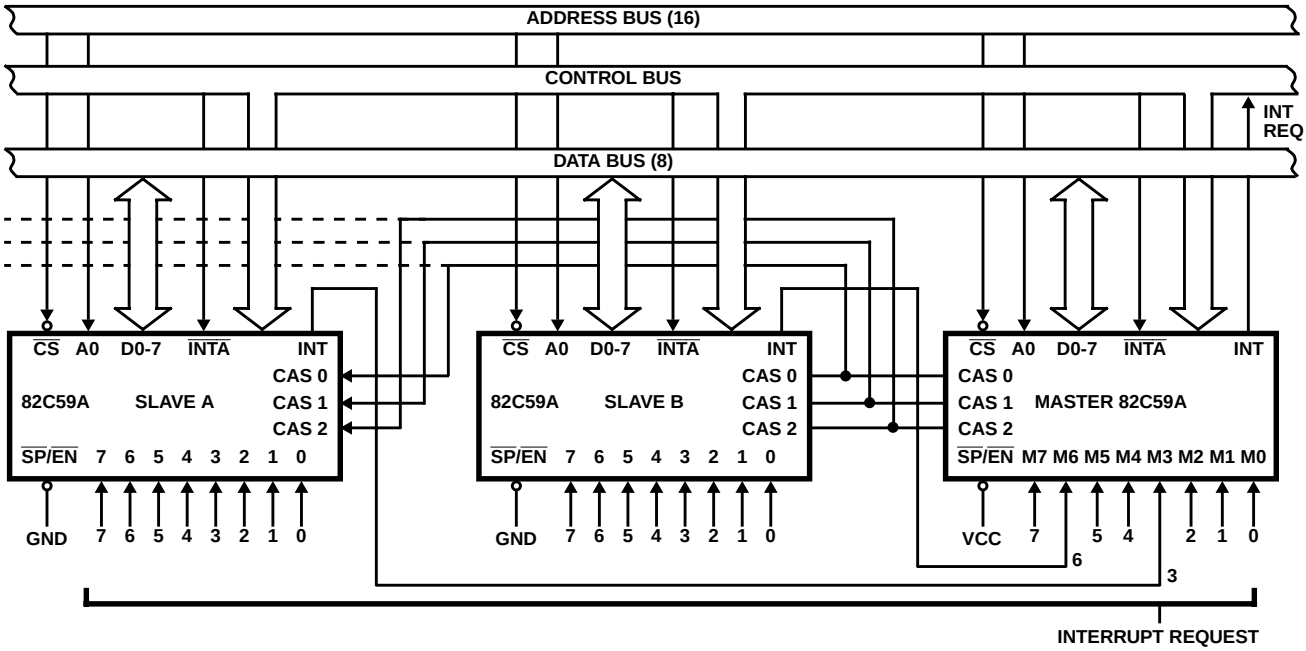


FIGURE 2. CASCADING THE 82C59A

1.6 Fully Nested Mode

By default, the 82C59A operates in the Fully Nested Mode. It will remain in this mode until it is programmed otherwise. In the Fully Nested Mode, interrupts are ordered by priority from highest to lowest. Initially, the highest priority level is IR0 with IR7 having the lowest. This ordering can be changed through the use of priority rotation (see 1.2).

In the Fully Nested Mode, when an interrupt occurs, its corresponding bit will get set in the Interrupt Request Register (IRR). When the processor acknowledges the interrupt, the 82C59A will look to the IRR to determine the highest priority interrupt requesting service. The bit in the In-service Register (ISR) corresponding to this interrupt will then be set. This bit remains set until an EOI is sent to the 82C59A.

While an interrupt is being serviced, only higher priority interrupts will be allowed to interrupt the current interrupt being serviced. However, lower priority interrupts can be allowed to interrupt higher priority requests if the 82C59A is programmed for operation in the Special Mask Mode.

When using the 82C59A in an 80C86- or 80C88-based system, interrupts will automatically be disabled when the processor begins servicing an interrupt request. The current address and the state of the flags in the processor will be pushed onto the stack. The interrupt-enable flag is then cleared. To allow interrupts to occur at this point, the STI instruction can be used. Upon exiting the service routine using the IRET instruction, execution of the program is resumed at the point where the interrupt occurred, and the flags are restored to their original values, thus re-enabling interrupts.

A configuration in which the Fully Nested structure is not preserved occurs when one or more of the following conditions occur:

(a) The Automatic EOI mode is being used.

(b) The Special Mask Mode is in use.

(c) A slave 82C59A has a master that is not programmed to the Special Fully Nested Mode.

Cases (a) and (b) differ from case (c) in that the 82C59A would allow lower priority interrupt requests the opportunity to be serviced before higher priority interrupt requests.

1.7 Master

When using multiple 82C59As in a system, one 82C59A has control over all other 82C59As. This is known as the “master” interrupt controller. Communication between the master and the other (slave) 82C59As occurs via the CAS0 - 2 lines. These lines form a private bus between the multiple 82C59As. Also, the INT lines from the slaves are routed to the master’s IR input pin(s). See Figure 2.

1.8 Slave

A “slave” 82C59A in a system is controlled by a master 82C59A. There is but one “master” in the system, but there can be up to 8 slave 82C59As. The INT outputs from the slaves act as inputs to the master through it’s IR inputs. Communications between the master and slaves occurs via the CAS0 - 2 lines. See Figure 2.

1.9 Special Fully Nested Mode

The Special Fully Nested Mode (SFNM) is used in a system having multiple 82C59As where it is necessary to preserve the priority of interrupts within a slave 82C59A. Only the master is programmed for the Special Fully Nested Mode through ICW4. This mode is similar to the Fully Nested Mode with the following exceptions:

(a) When an interrupt from a particular slave is being serviced, additional higher priority interrupts from that slave can cause an interrupt to the master. Normally, a slave is masked out when its request is in service.

(b) When exiting the Interrupt Service routine, the software should first issue a non-specific EOI to the slave. The Inservice Register (ISR) should then be read and checked to see if its contents are zero. If the register is empty, the software should then write a non-specific EOI to the master. Otherwise, a second EOI need not be written because there are interrupts from that slave still being processed.

NOTE: Because the Master 82C59A and its slave 82C59As must be in Fully Nested Mode for this mode to be functional, we could not utilize Automatic EOIs. These would disturb the Fully Nested structure, as described in section 1.6.

1.10 Special Mask Mode

The Special Mask Mode is utilized in order to allow interrupts from all other levels (higher and lower as well) to interrupt the IR level that is currently being serviced. Invoking this mode of operation will disturb the fully nested priority structure.

Generally, the Special Mask Mode is selected during the servicing of an interrupt. The software should first set the bit corresponding to the IR level being serviced, in the Interrupt Mask Register (OCW1). The Special Mask Mode and interrupts should then be enabled. This will allow any of the IR levels except for those masked off by OCW1 to interrupt the IR level currently being serviced.

Because this disturbs the Fully Nested Structure, it is required that a Specific EOI be issued when servicing interrupts while the Special Mask Mode is in effect. Before exiting the original interrupt routine, the Special Mask Mode should be disabled.

1.11 Specific Rotation

By issuing the proper command word to OCW2, the priority structure of the 82C59A can be dynamically altered. The command word written to OCW2 would specify which is to be the lowest priority IR level.

This specific rotation can be accomplished one of two ways. The first is through a specific EOI. The software can specify that rotation is to be applied to the IR level provided with the EOI. The second method is a simple “set priority” command, in which the lowest priority level is specified with the command word.

2.0 Initialization Control Words

The following section gives a description of the Initialization Control Words (ICW) used for configuring the 82C59A Interrupt controller. There are four (4) control words used for initialization of the 82C59A. These ICWs must be programmed in the proper sequence beginning with ICW1. If at any time during the course of operation the configuration of the 82C59A needs to be changed, the user must again write out the control words to the device in their proper order. The initialization sequence is shown in Figure 3.

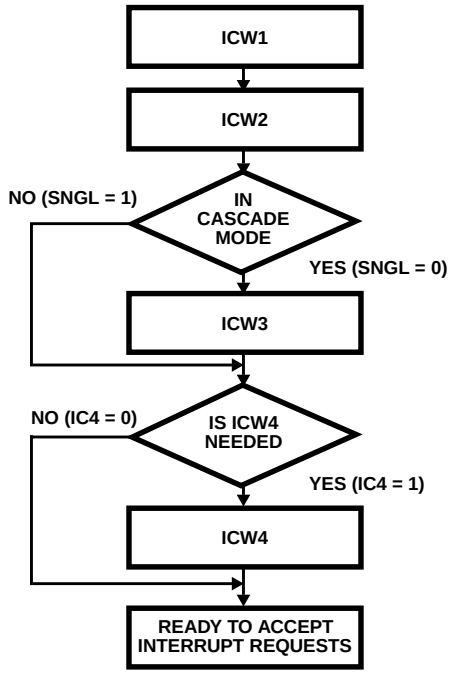


FIGURE 3. 82C59A INITIALIZATION SEQUENCE

ICW1: The 82C59A recognizes the first Initialization Control Word (ICW) written to it based on two criteria: (1) the A0 line from the address bus must be a zero, and (2) the D4 bit must be a one. If the D4 bit is set to a zero, we would be programming either OCW2 or OCW3 (these are explained later). The function of ICW1 is to tell the 82C59A how it is being used in the system (i.e. Single or cascaded, edge or level triggered interrupts etc.).

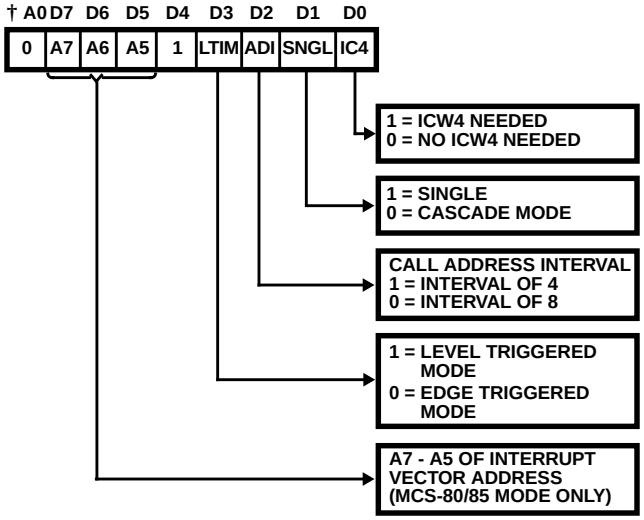
ICW2: This control word is always issued directly after ICW1. When addressing this ICW, the A0 line from the address bus must be a one (high). ICW2 is utilized in providing the CPU with information on where to vector to in memory when servicing an interrupt.

ICW3: This control word is issued only if the SNGL (D1) bit of the ICW1 has been programmed with a zero. When addressing this word, the A0 line from the CPU must be high (1). This control word is for cascaded 82C59As. It allows the master and slave 82C59As to communicate via the CAS0-2 lines. With the master, this word indicates which IR lines have slaves connected to them. For the slave 82C59A(s), this word indicates to which IR line on the master it is connected.

ICW4: Issuance of this ICW is selectable through the IC4 (D0) bit of ICW1. If ICW4 is to be written to the 82C59A, A0 from the CPU must be high (1) when writing to it. This word needs to be written only when the 82C59A is operating in modes other than the default modes. Instances when we would want to write to ICW4 are one or more of the following: An 80C86(80C88) processor is being used, buffered outputs (D0 - D7) are to be used, Automatic EOIs are desired, or the Special Fully Nested mode is to be used.

2.1 ICW1

ICW1 is the first control word that is written to the 82C59A during the initialization process. To access this word, the value of A0 must be a zero (0) in the addressing, and bit D4 of ICW1 must be a one (1). The format of the command word is as follows:



† A0 is an address bit, and not part of the ICW.

FIGURE 4. ICW1 FORMAT

D7 thru D5 - A7, A6, A5: These bits are used in the 8080/85 mode to form a portion of the low byte call address. When using the 4 byte address interval, all 3 bits are utilized. When using the 8 byte interval, only bits A7 and A6 are used. Bit A5 becomes a “don’t care” bit. If using an 80C86(80C88) system, the value of these bits can be set to either a one or zero.

D3 - LTIM:

0: The 82C59A will operate in an edge triggered mode. An interrupt request on one of the IR lines (IR0 - IR7) is recognized by a low to high transition on the pin. The IR signal must remain high at least until the falling edge of the first $\overline{\text{INTA}}$ pulse. Subsequent interrupts on the IR pin(s) will not occur until another low-to-high transition occurs.

1: Sets up the 82C59A to operate in the level triggered mode. Interrupts occur when a “high” level is detected on one or more of the IR pins. The interrupt request must be removed from this pin before the EOI command is issued by the CPU. Otherwise, the 82C59A will see the IR line still in a high state, and consider this to be another interrupt request.

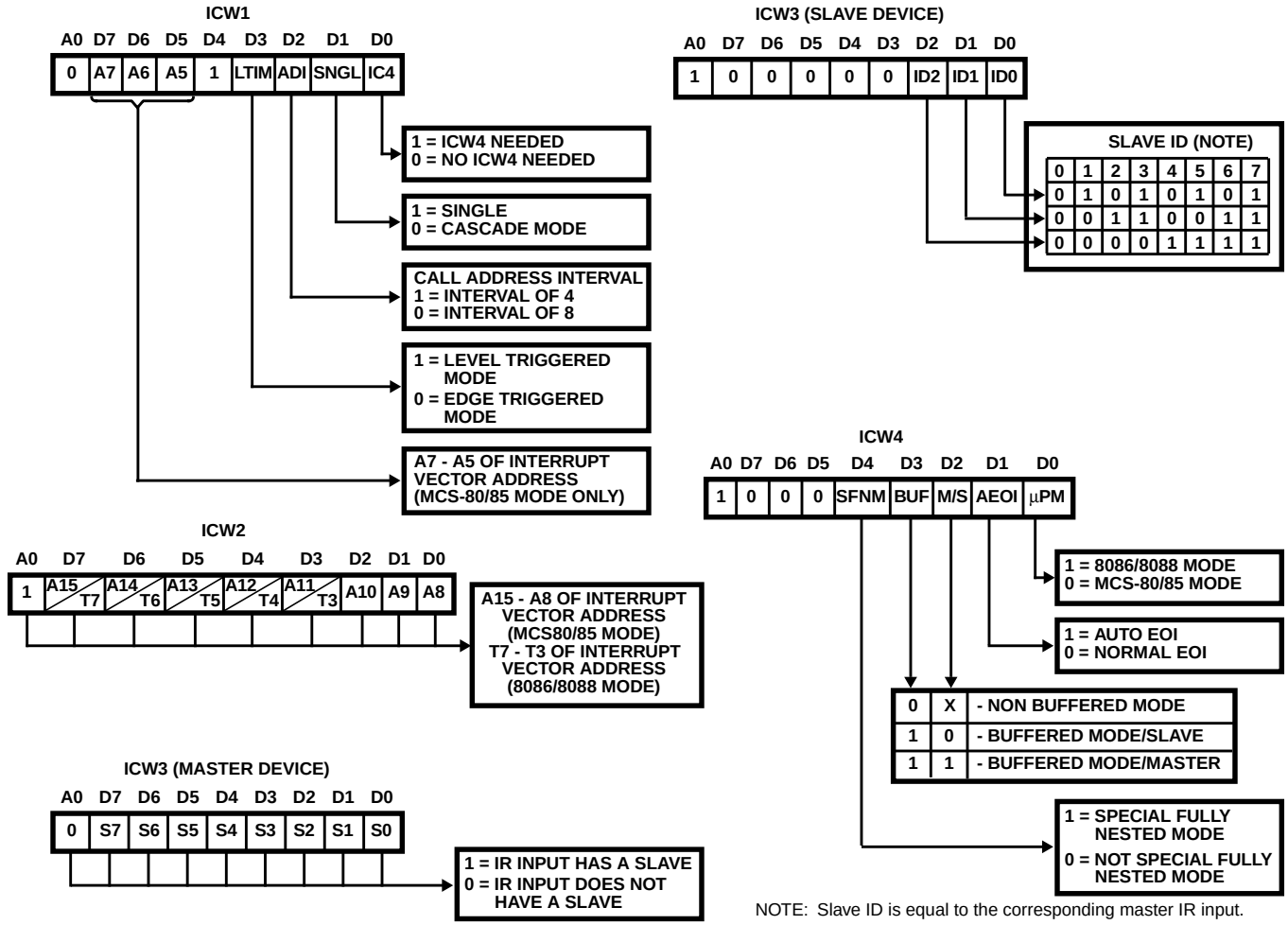


FIGURE 5. 82C59A INITIALIZATION COMMAND WORD FORMAT

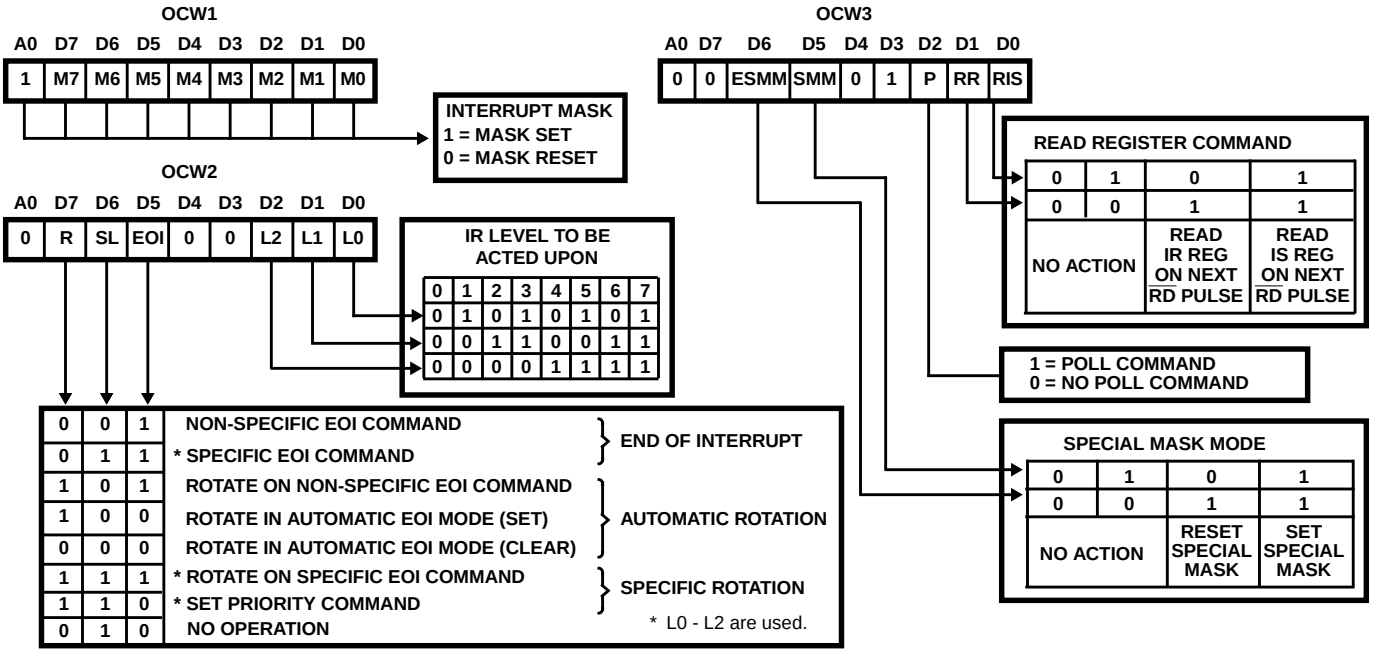


FIGURE 6. 82C59A OPERATION COMMAND WORD FORMAT

D2 - ADI: Call Address Interval (for 8080/8085 use only). If using the 82C59A in an 80C86/88 based system, the value of this bit can be either a 0 or 1.

0: The address interval generated by the 82C59A is 8 bytes. This option provides compatibility with the RST interrupt vectoring in 8080/8085 systems since the vector locations are 8 bytes apart. This vector will be combined with the values specified in bits D7 and D6 of ICW1. The addresses generated are shown in Table 1.

TABLE 1. ADDRESS INTERVAL (8 BYTES)

D7	D6	D5	D4	D3	D2	D1	D0	
A7	A6	1	1	1	0	0	0	**IR7**
A7	A6	1	1	0	0	0	0	**IR6**
A7	A6	1	0	1	0	0	0	**IR5**
A7	A6	1	0	0	0	0	0	**IR4**
A7	A6	0	1	1	0	0	0	**IR3**
A7	A6	0	1	0	0	0	0	**IR2**
A7	A6	0	0	1	0	0	0	**IR1**
A7	A6	0	0	0	0	0	0	**IR0**

1: The address interval generated by the interrupt controller will be 4 bytes. This provides the user with a compact jump table for 8080/8085 systems. The interrupt number is effectively multiplied by four and combined with bits D7, D6, and D5 to form the lower byte of the call instruction generated and sent to the 8080 and 8085. Table 2 shows how these addresses are generated for the various interrupt request (IR) levels.

TABLE 2. ADDRESS INTERVAL (4 BYTES)

D7	D6	D5	D4	D3	D2	D1	D0	
A7	A6	A5	1	1	1	0	0	**IR7**
A7	A6	A5	1	1	0	0	0	**IR6**
A7	A6	A5	1	0	1	0	0	**IR5**
A7	A6	A5	1	0	0	0	0	**IR4**
A7	A6	A5	0	1	1	0	0	**IR3**
A7	A6	A5	0	1	0	0	0	**IR2**
A7	A6	A5	0	0	1	0	0	**IR1**
A7	A6	A5	0	0	0	0	0	**IR0**

D1 - SNGL:

0: This tells the 82C59A that more than one 82C59A is being used in the system, and it should expect to receive ICW3 following ICW2. How the particular 82C59A is being used in the system will be determined either through ICW4 for buffered mode, or through the $\overline{SP}/\overline{EN}$ pin for non-buffered mode operation.

1: Tells the 82C59A that it is being used alone in the system. Therefore, there will be no need to issue ICW3 to the device.

D0 - IC4: Specifies to the 82C59A whether or not it can expect to receive ICW4. If this device is being used in an 80C86/80C88 system, ICW4 must be issued.

0: ICW4 will not be issued. Therefore, all of the parameters associated with ICW4 will default to the zero (0) state. This should only be done when using the 82C59A in an 8080 or 8085 based system.

1: ICW4 will be issued to the 82C59A.

2.2 ICW2

ICW2 is the second control word that must be sent to the 82C59A. This byte is used in one of two ways by the 82C59A, depending on whether it is being used in an 8080/85 or an 80C86/88 based system.

When used in conjunction with the 8080/85 microprocessor, the value given to this register is taken as being the high byte of the address in the CALL instruction sent to the CPU.

D7	D6	D5	D4	D3	D2	D1	D0
A15	A14	A13	A12	A11	A10	A9	A8

FIGURE 7. ICW2 FORMAT

In an 80C86- or 80C88-based system, ICW2 is used to send the processor an interrupt vector. This vector is formed by taking the value of bits D7 through D3 and combining them with the interrupt request level to get an eight bit number. The processor will multiply this number by four and go to that absolute location in memory to find a starting address for the interrupt service routine corresponding to the interrupt request.

For example, if we set ICW2 to “00011000” and an interrupt is recognized on IR1, the vector sent to the 80C86(80C88) will be 00011001 (19H). The processor will then look to the memory location 64H to find the starting address of the corresponding interrupt service routine. It is the responsibility of the software to provide this address in the interrupt table.

D7	D6	D5	D4	D3	D2	D1	D0
A7	A6	A5	A4	A3	X	X	X

FIGURE 8. ICW2 FORMAT (80C86 MODE)

2.3 ICW3

ICW3 is only issued when the SNGL bit in ICW1 has been set to zero. If not set, the next word written to the 82C59A will be interpreted as ICW4 if A0 = 1 and IC4 from ICW4 was set to one, or it could see it as one of the Operation Command Words based upon the state of the A0 line.

Like ICW2, this control word can be interpreted in two ways by the 82C59A. However the interpretation of this word depends on whether the 82C59A is being used as a “master” or “slave” in the system. The definition of the particular device’s role in the system is assigned through

ICW4 (which will be discussed later), or through the state of the $\overline{SP}/\overline{EN}$ pin (pin 16).

82C59A AS A MASTER

If a given 82C59A is being used as a master, the eight (8) bits in this command word are used to indicate which of the IR lines are being driven by a slave 82C59A.

D7	D6	D5	D4	D3	D2	D1	D0
S7	S6	S5	S4	S3	S2	S1	S0

FIGURE 9. ICW3 FORMAT (MASTER)

D7 THRU D0:

0: The corresponding IR line to this bit is not being driven by a slave 82C59A. This line can however then be connected to the interrupt output of another interrupting device such as a UART. If there are unused bits in this byte because not all eight of the IR lines are used, set them to zero.

1: The corresponding IR line to this bit is being driven by a slave 82C59A.

The bits in this command word are directly related to the IR lines. For example, to tell the 82C59A that there is a slave device connected to IR5 (pin 23), bit D5 of the command word should be set to a one (1).

82C59A AS A SLAVE DEVICE

When the device is being used as a slave device, we must use ICW3 to inform itself as to which IR line it will be connected to in the master. Therefore, only the three (3) least significant bits of ICW3 will be used to specify this value.

D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	0	0	ID2	ID1	ID0

FIGURE 10. ICW3 FORMAT (SLAVE)

These bits are coded as follows:

TABLE 3. SLAVE IDENTIFICATION WITH ICW3

MASTER IR NUMBER	ID2	ID1	ID0
IR7	1	1	1
IR6	1	1	0
IR5	1	0	1
IR4	1	0	0
IR3	0	1	1
IR2	0	1	0
IR1	0	0	1
IR0	0	0	0

For example, if the INT output of a “slave” 82C59A is connected to the input pin IR5 on the “master” 82C59A, ICW3 of the “slave” would be programmed with the value 00000101b, or 05H. This informs the “slave” as to which priority level it holds with the “master”.

D7 thru D3: These bits must be set to zeros (0) for proper operation of the device.

2.4 ICW4

This control register is written to only when the IC4 bit is set in ICW1. The purpose of this command word is to set up the 82C59A to operate in a mode other than the default mode of operation. The default mode of operation is the same as if a value of 00H were to be written to ICW4 (i.e. all bits set to zero).

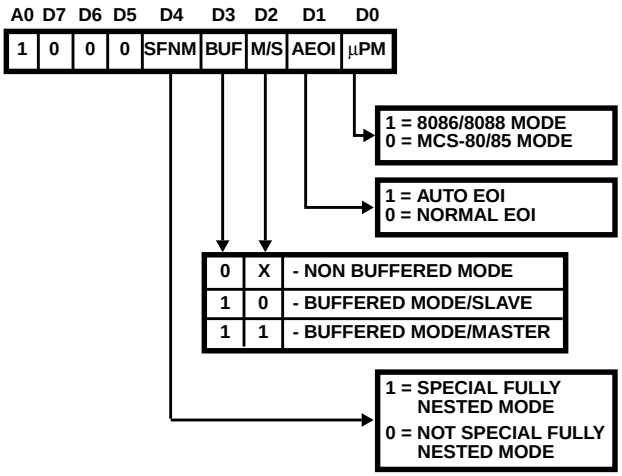


FIGURE 11. ICW4 FORMAT

D7 thru D5: These bits must be set to zero for proper operation.

D4-SFNM: This bit is used in the selection of the Special Fully Nested Mode (SFNM) of operation. This mode should only be used when multiple 82C59As are cascaded in a system. It needs only to be programmed in the Master 82C59A in the system.

0: Special Fully Nested Mode is not selected.

1: Special Fully Nested Mode is selected.

D3 - BUF: This bit tells the 82C59A whether or not the outputs from the data pins (D0 - D7) will be buffered. If they are buffered, this bit will cause the $\overline{SP}/\overline{EN}$ pin to become an output signal that can be used to control the “enable” pin on a buffering device(s).

0: The device will be used in a non-buffered mode. Therefore, (1) the M/S bit in ICW4 is a don’t care, and (2) the $\overline{SP}/\overline{EN}$ pin becomes an input pin telling the device if it is being used as a master (pin 16 = High) or a slave (pin 16 = low). For systems using a single 82C59A, the $\overline{SP}/\overline{EN}$ input should be tied high.

1: The device is used in buffered mode. An enable output signal will be generated on pin 16, and the M/S bit will be

used for determining whether the particular 82C59A is a “master” or a “slave”.

D2 - M/S: This bit is of significance only when the BUF bit is set (BUF =1). The purpose of this bit is to determine whether the particular 82C59A is being used as a “master” or a “slave” in the target system.

0: The 82C59A is being used as slave.

1: The 82C59A is the master interrupt controller in the system.

D1 - AEOI: This bit is used to tell the 82C59A to automatically perform a non-specific End-of-Interrupt on the trailing edge of the last Interrupt Acknowledge pulse. Users should note that when this is selected, the nested priority interrupt structure is lost.

0: Automatic End-of-Interrupt will not be generated.

1: Automatic End-of-Interrupt will be generated on the trailing edge of the last Interrupt Acknowledge pulse.

D0 - μPM: This bit tells the Interrupt Controller which microprocessor is being used in the system. An 8080/8085, or an 80C86/80C88.

0: The 82C59A will be used in an 8080/8085 based system.

1: 82C59A to be used in the 80C86/88 mode of operation.

3.0 Operation Command Words

Once the Initialization Command Words, described in the previous section, have been written to the 82C59A, the device is ready to accept interrupt requests. While the 82C59A is operating, we have the ability to select various options that will put the device in different operating modes, by writing Operation Command Words (OCWs) to the 82C59A. These OCWs can be sent at any time after the device has been initialized and in any order. These words can be changed at any time as well. Note: If A0 = 0 and D4 of the command word = 1, the 82C59A will begin the ICW initialization sequence.

There are three different OCWs for the 82C59A. Each has a different purpose. The first control word (OCW1) is used for masking out interrupt lines that are to be inactive or ignored during operation. OCW2 is used to select from various priority resolution algorithms in the device. Finally, OCW3 is used for (1) controlling the Special Mask Mode, and (2) telling the 82C59A which Register will be read on the next RD pulse; the ISR (In-service Register) or the IRR (Interrupt Request Register).

3.1 OCW1

This control word is used to set or clear the masking of the eight (8) interrupt lines input to the 82C59A. This control word performs this function via the Interrupt Mask Register (IMR). In it’s initial state, the value of this register is 00H. In other words, all of the interrupt lines are enabled. Therefore, we need only write this control word when we wish to disable specific interrupt lines.

A direct mapping occurs between the bits in this control word and the actual interrupt pins on the device. For example bit 7 (D7) controls interrupt line IR7 (pin 25), bit 6 controls IR6, and so on.

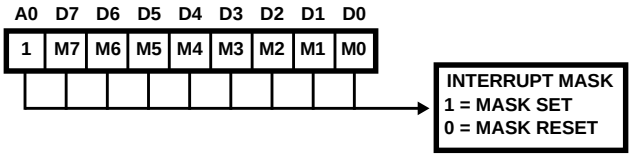


FIGURE 12. OCW1 FORMAT

Even though the user can mask off any of the IR lines, any interrupt occurring during that time will not be lost. The request for an interrupt is retained in the IRR; therefore when that IR is unmasked by issuing a new mask value to OCW1, the interrupt will be generated when it becomes the highest requesting priority.

D7 THRU D0:

0: When any of the bits in the control word are reset (0), the corresponding interrupt is enabled.

1: By setting a bit(s) to a one in the control word, the corresponding interrupt line(s) is disabled.

For example, if the value 34H (00110100b) were written to OCW1, interrupts would be disabled from being serviced on lines IR2, IR4, and IR5.

3.2 OCW2

In ICW4 bit D1 was used to specify whether the 82C59A should wait for an EOI (End of Interrupt) from the CPU, or generate its own EOI (Automatic EOI). If bit D1 of ICW4 had been programmed to be zero, OCW2 would be used for sending the EOI to the 82C59A. Conversely, if this bit had been set to a one, OCW2 would be used for specifying whether or not the 82C59A should perform a priority rotation on the interrupts when the AEOI is detected.

OCW2 has several EOI options. The EOI issued can be either specific or non-specific. For each of these EOIs, the user can specify whether or not priority rotation should be performed.

D7	D6	D5	D4	D3	D2	D1	D0
R	SL	EOI	0	0	L2	L1	L0

FIGURE 13.

R, SL, AND EOI:

These three bits are used for specifying how the device should handle AEOIs, or for issuing one of several different EOIs. They are programmed as shown in the following table.

TABLE 4. ROTATE AND EOI MODES

R	SL	EOI	
0	0	1	Non-specific EOI command
0	1	1	*Specific EOI command
1	0	1	Rotate on non-specific EOI command
1	0	0	Rotate in Automatic EOI mode (set)
0	0	0	Rotate in Automatic EOI mode (clear)
1	1	1	*Rotate on specific EOI command
1	1	0	*Set priority command
0	1	0	No operation

* L0 - L2 are used.

L2, L1, AND L0:

These three bits of the control word are used in conjunction with the issuance of specific EOIs or when specifically establishing a different priority structure. The bits tell the 82C59A which interrupt level is to be acted upon. Therefore, the software needs to know which interrupt is being serviced by the 82C59A

TABLE 5. INTERRUPT LEVEL TO ACT UPON

L2	L1	L0	
0	0	0	IR level 0
0	0	1	IR level 1
0	1	0	IR level 2
0	1	1	IR level 3
1	0	0	IR level 4
1	0	1	IR level 5
1	1	0	IR level 6
1	1	1	IR level 7

3.3 OCW3

There are two main functions that OCW3 controls: (1) Interrupt Status, and (2) Interrupt Masking. Interrupt status can be checked by looking at the ISR or IRR registers, or by issuing a Poll Command to manually identify the highest priority interrupt requesting service.

D7	D6	D5	D4	D3	D2	D1	D0
0	ESMM	SMM	0	1	P	RR	RIS

FIGURE 14.

D7: Must be set to zero for proper operation of the 82C59A.

D6 - ESMM: Enable Special Mask Mode - The ESMM bit when enabled allows the SMM bit to set or clear the Special Mask Mode. When disabled this bit causes the SMM bit to have no effect on the 82C59A.

0: Disables the effect of the SMM bit.

1: Enable the SMM bit to control the Special Mask Mode.

D5 - SMM: Special Mask Mode - The SMM bit is used to enable or disable the Special Mask Mode. This bit will only affect the 82C59A when the ESMM bit is set to 1.

0: Disable the Special Mask Mode.

1: Put the 82C59A into the Special Mask Mode.

D4, D3: These bits are used to differentiate between OCW2, OCW3, and ICW1. To properly select OCW3, D4 must be set to zero and D3 must be set to one.

D2 - P: Poll Command - This bit is used to issue the poll command to the 82C59A. The next read of the 82C59A will cause a poll word to be returned which tells if an interrupt is pending, and if so, which is the highest requesting level.

NOTE: The poll command must be issued each time the poll operation is desired.

0: No poll command issued to the 82C59A.

1: Issue the poll command.

D1 - RR: Read Register - This bit is used to execute the “read register” command. When this bit is set, the 82C59A will look at the RIS bit to determine whether the ISR or IRR register is to be read. When issuing this command, the next instruction executed by the CPU should be an input from this same port to get the contents of the specified register.

0: No “Read Register” command will be performed.

1: The next input instruction by the CPU will read either the contents of the ISR or the IRR as specified by the RIS bit.

D0 - RIS: This bit is used in conjunction with the RR bit to select which register is to be read when the “Read Register” command is issued.

0: The next input instruction will read the contents of the Interrupt Request Register (IRR).

1: The next input instruction will read the contents of the In-Service Register (ISR).

The two registers that can be accessed through the Read Register command are used to determine which interrupts are requesting service, and which one(s) are currently being serviced.

The IRR bits get set when corresponding Interrupt requests are received. For instance, when IR4 is detected, bit D4 of the IRR will get set. When an interrupt acknowledge comes back from the CPU, the priority resolution logic will determine which interrupt request will be serviced. The corresponding bit in the In-service Register (ISR) will then be set. Clearing of the correct bits in the ISR occurs through out use of the AEOI, or by issuing an EOI to the device.

4.0 Addressing the 82C59A

There are two factors that must be taken into account when addressing the 82C59A in a system. To begin with, the 82C59A is accessed only when the $\overline{CS}$ pin (chip select) sees an active signal (low). This signal is generated using control circuitry in the system. Secondly, the various registers within the 82C59A are selected based upon the state of the A0 (address pin) as well as specific bits in the command words (i.e. for ICW1, OCW2, and OCW3 A0 must be a zero).

The circuit in Figure 15 shows that the $\overline{CS}$ signal is generated using an HPL-82C338 Programmable Chip Select Decoder (PCSD). This device is being used as a 3-to-8 decoder. Note that the G1 input is active high and G2 thru G5 have been programmed to be active low. The A, B, and C inputs to the 82C338 correspond to address lines AD2, AD3, and AD4 respectively, from the 80C88. The A0 input to the 82C59A is also taken from the CPUs address bus: AD0 is used. It should be noted that address line AD1 from the 80C88 is not being used in the addressing of this particular peripheral. This is done to allow other peripheral devices that require two address inputs for internal register selection, to use address lines AD0 and AD1 from the processor.

Because the AD1 address line from the 80C88 is not being used, the 82C59A will be addressed regardless of whether AD1 is high or low (1 or 0). The remainder of the address lines from the 80C88 can either be a zero or one when addressing the 82C59A. For the examples to be presented, it can be assumed that all unused address lines will be set to zero when addressing the 82C59A.

In Figure 15, output $\overline{Y6}$ from the HPL-82C338 is being used as the $\overline{CS}$ input to the 82C59A. This line is enabled when the inputs on A, B, and C are: A = 0, B = 1, and C = 1. Combining this with the A0 input to the 82C59A, we get the addresses 18H and 19H for accessing the 82C59A.

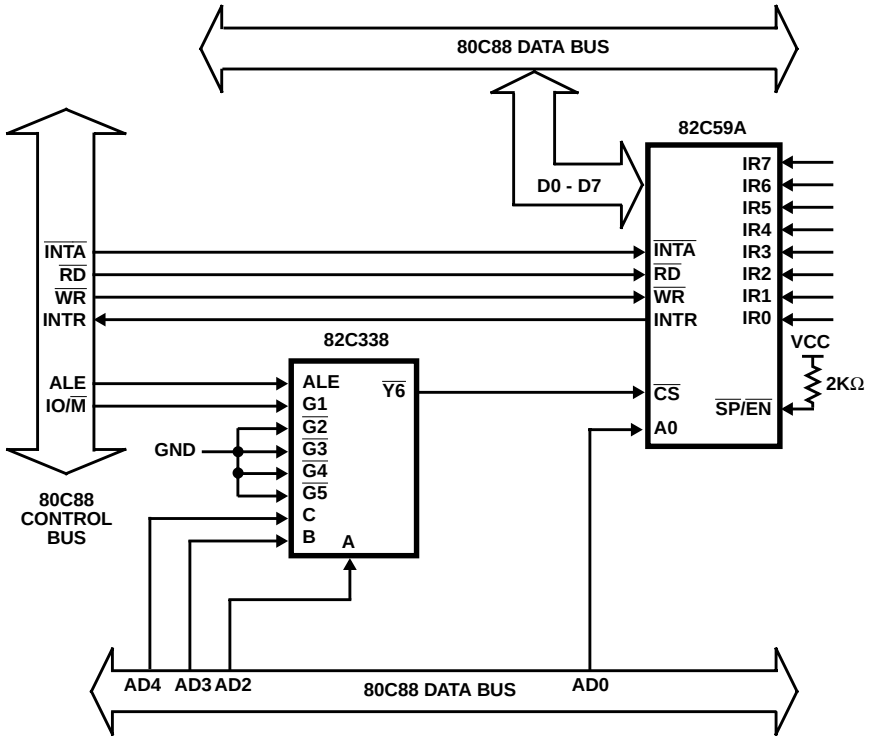


FIGURE 15. ADDRESSING THE 82C59A

5.0 Programming the 82C59A

As described earlier, there are two different types of command words that are used for controlling 82C59A operation; the Initialization Command Words (ICWs) and the Operation Command Words (OCWs). To properly program the 82C59A, it is essential that the ICWs be written first. When writing the ICWs to the 82C59A, they must be written in the following sequence:

1. Write ICW1 to the 82C59A, A0 = 0.
2. Write ICW2 to the 82C59A, A0 = 1.
3. If using cascaded 82C59As in system, write ICW3 to the 82C59A, A0 = 1.
4. If IC4 bit was set in ICW1, write ICW4 to the 82C59A.

NOTE: When using multiple 82C59As in the system (cascaded), each one must be initialized following the above sequence.

Once the 82C59A(s) has been configured through the ICWs, the OCWs can be used to select from the various operation mode options. These include: masking of interrupt lines, selection of priority rotation, issuance of EOIs, reading of the ISR and/or IRR etc. These OCWs can be written to the 82C59A at any time during operation of the 82C59A. The various command words are identified by the state of selected bits in the words, rather than by the sequence that they are written to the 82C59A; as with the ICWs. Therefore, it is imperative that the fixed bit values in the command words be written as such to insure proper operation of the device(s).

5.1 Example 1: Single 82C59A

In Example 1, we are using a single 82C59A in a system to handle the interrupts caused by an 82C52 Programmable UART. The system is driven using a 80C86 microprocessor. The system configuration is shown in Figure 16.

Interrupts are initiated by the 82C52 anytime it receives data on its Serial Data in pin (SDI), or when it is ready to transmit more data via its Serial Data Out pin (SDO).

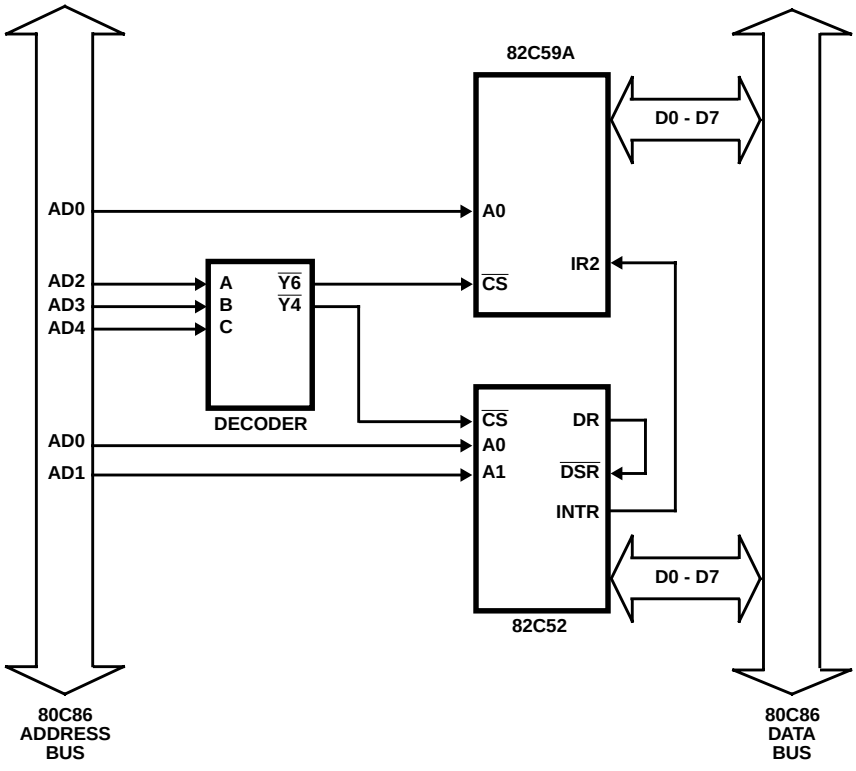


FIGURE 16. EXAMPLE 1: SINGLE 82C59A

5.2 Example 2: Cascaded 82C59As

Example 2 illustrates how we can use multiple 82C59As in Cascade Mode. Figure 17 shows the interconnections between the master and slave interrupt controllers. In this example, only one interrupt can occur. This is generated by the 82C52 UART. Except for the fact that this system is configured with a Master-Slave interrupt scheme, it is the same as that in Example 1.

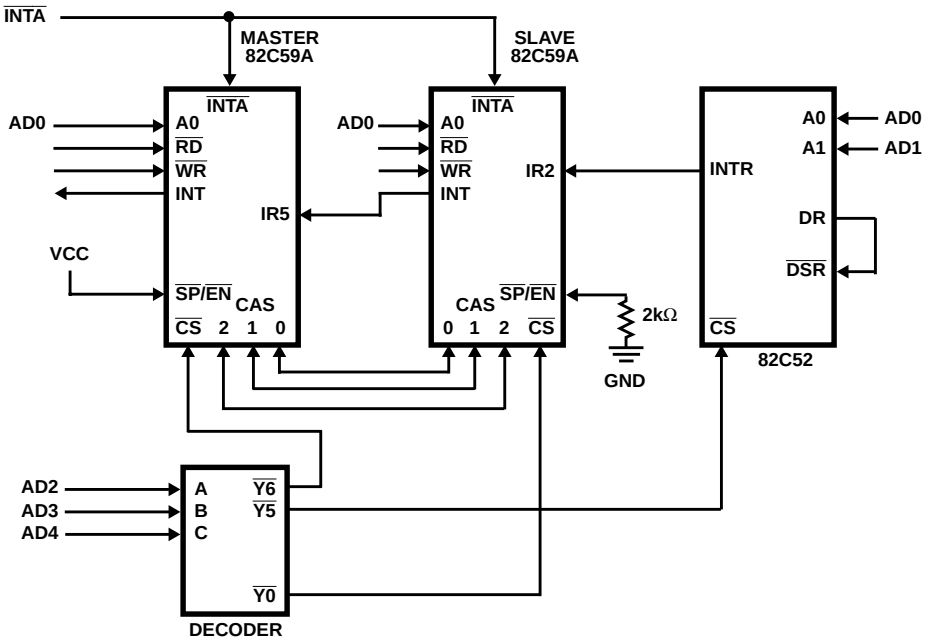


FIGURE 17. EXAMPLE 2: CASCADED 82C59As

6.0 Expansion Past 64 Interrupts

In some instances, it may be desirable to expand the number of available interrupts in a system past the maximum of 64 imposed when using cascaded 82C59As. The easiest way to accomplish this is through the use of the Poll command with the 82C59A. Figure 18 illustrates one example of how this expansion can be accomplished. Notice that we are using two 3 - to -8 decoders to address up to 16 82C59As. Selection of which decoder is active takes place using the $\overline{OE}$ pin driven by AD5 from the CPU's address bus.

With this type of interrupt structure, we are not using the INT and $\overline{INTA}$ lines from our processor (80C88 for this example). Because of this, no interrupts will break execution of the system software. Therefore, it is the task of the software to poll the various 82C59As in the system to see if any interrupts are pending. Once it has been established which interrupt requires servicing, the software can take appropriate action.

There are disadvantages to using the poll mode for the systems interrupt structure: (1) the overhead of polling each of the 82C59As reduces the systems efficiency, and (2) realtime interrupt servicing cannot be guaranteed.

There are several advantages to using the poll mode in this manner: (1) there can be more than 64 priority interrupts in the system, and (2) memory in the system is freed because no interrupt vector table is required.

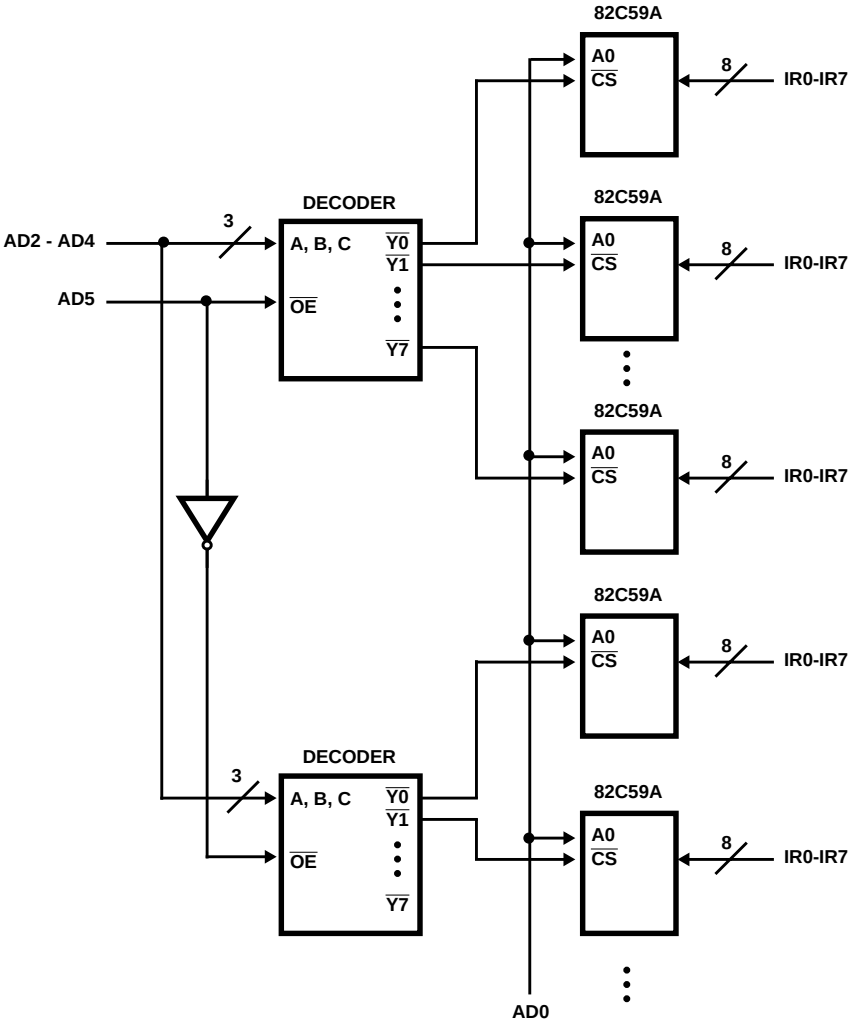


FIGURE 18. EXPANDING PAST 64 INTERRUPTS

CMOS 8-bit A/D converters — ADC0803/4-1

DESCRIPTION

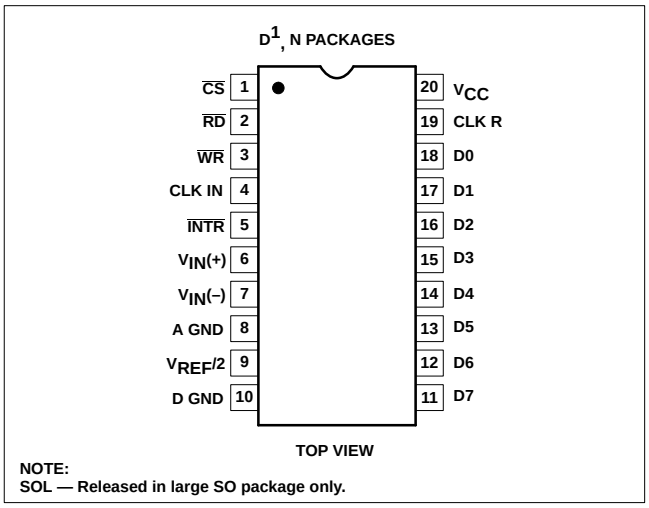
The ADC0803 family is a series of three CMOS 8-bit successive approximation A/D converters using a resistive ladder and capacitive array together with an auto-zero comparator. These converters are designed to operate with microprocessor-controlled buses using a minimum of external circuitry. The 3-State output data lines can be connected directly to the data bus.

The differential analog voltage input allows for increased common-mode rejection and provides a means to adjust the zero-scale offset. Additionally, the voltage reference input provides a means of encoding small analog voltages to the full 8 bits of resolution.

FEATURES

- Compatible with most microprocessors
- Differential inputs
- 3-State outputs
- Logic levels TTL and MOS compatible
- Can be used with internal or external clock
- Analog input range 0V to V_{CC}
- Single 5V supply
- Guaranteed specification with 1MHz clock

PIN CONFIGURATION



NOTE:
SOL — Released in large SO package only.

APPLICATIONS

- Transducer-to-microprocessor interface
- Digital thermometer
- Digitally-controlled thermostat
- Microprocessor-based monitoring and control systems

ORDERING INFORMATION

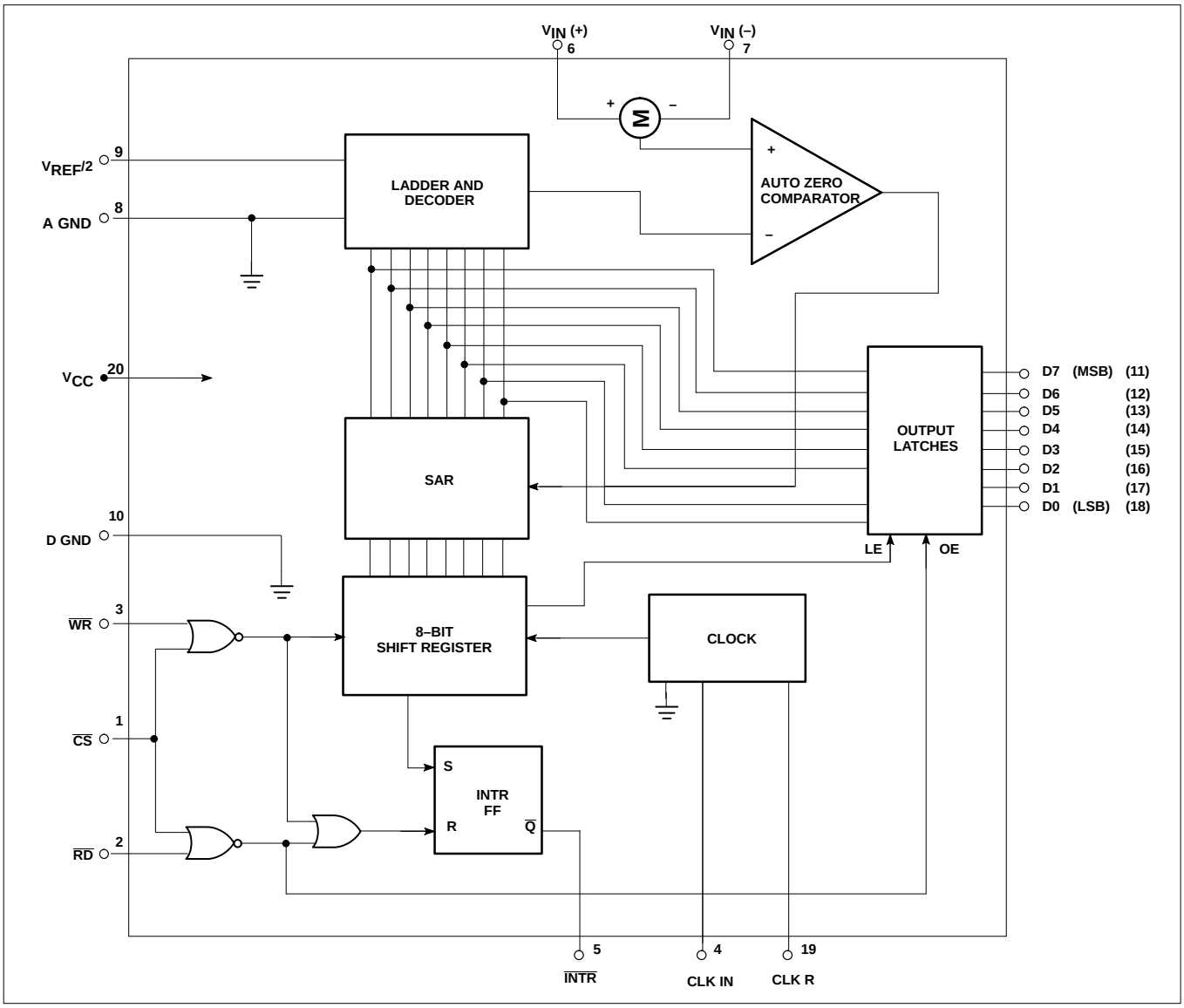
DESCRIPTION	TEMPERATURE RANGE	ORDER CODE	DWG #
20-Pin Plastic Dual In-Line Package (DIP)	-40 to +85°C	ADC0803/04-1 LCN	0408B
20-Pin Plastic Dual In-Line Package (DIP)	0 to 70°C	ADC0803/04-1 CN	0408B
20-Pin Plastic Small Outline (SO) Package	0 to 70°C	ADC0803/04-1 CD	1021B
20-Pin Plastic Small Outline (SO) Package	-40 to 85°C	ADC0803/04-1 LCD	1021B

ABSOLUTE MAXIMUM RATINGS

SYMBOL	PARAMETER	RATING	UNIT
V_{CC}	Supply voltage	6.5	V
	Logic control input voltages	-0.3 to +16	V
	All other input voltages	-0.3 to (V_{CC} +0.3)	V
T_A	Operating temperature range		
	ADC0803/04-1 LCD	-40 to +85	°C
	ADC0803/04-1 LCN	-40 to +85	°C
	ADC0803/04-1 CD	0 to +70	°C
	ADC0803/04-1 CN	0 to +70	°C
T_{STG}	Storage temperature	-65 to +150	°C
T_{SOLD}	Lead soldering temperature (10 seconds)	300	°C
P_D	Maximum power dissipation T_A=25°C (still air)[1]		
	N package	1690	mW
	D package	1390	mW

NOTES:
1. Derate above 25°C, at the following rates: N package at 13.5mW/°C; D package at 11.1mW/°C

BLOCK DIAGRAM



DC ELECTRICAL CHARACTERISTICS

V_{CC} = 5.0V, f_{CLK} = 1MHz, $T_{MIN} \le T_A \le T_{MAX}$, unless otherwise specified.

SYMBOL	PARAMETER	TEST CONDITIONS	ADC0803/4			UNIT
			Min	Typ	Max	
	ADC0803 relative accuracy error (adjusted)	Full-Scale adjusted			0.50	LSB
	ADC0804 relative accuracy error (unadjusted)	$V_{REF}/2 = 2.500V_{DC}$			1	LSB
R_{IN}	$V_{REF}/2$ input resistance[3]	V_{CC} = 0V[2]	400	680		Ω
	Analog input voltage range[3]		–0.05		V_{CC}+0.05	V
	DC common-mode error	Over analog input voltage range		1/16	1/8	LSB
	Power supply sensitivity	V_{CC} = 5V ±10%[1]		1/16		LSB
Control inputs						
V_{IH}	Logical “1” input voltage	$V_{CC} = 5.25V_{DC}$	2.0		15	V_{DC}
V_{IL}	Logical “0” input voltage	$V_{CC} = 4.75V_{DC}$			0.8	V_{DC}
I_{IH}	Logical “1” input current	$V_{IN} = 5V_{DC}$		0.005	1	μA_{DC}
I_{IL}	Logical “0” input current	$V_{IN} = 0V_{DC}$	–1	–0.005		μA_{DC}
Clock in and clock R						
V_T+	Clock in positive-going threshold voltage		2.7	3.1	3.5	V_{DC}
V_T–	Clock in negative-going threshold voltage		1.5	1.8	2.1	V_{DC}
V_H	Clock in hysteresis (V_T+)–(V_T–)		0.6	1.3	2.0	V_{DC}
V_{OL}	Logical “0” clock R output voltage	I_{OL} = 360µA, $V_{CC} = 4.75V_{DC}$			0.4	V_{DC}
V_{OH}	Logical “1” clock R output voltage	I_{OH} = –360µA, $V_{CC} = 4.75V_{DC}$	2.4			V_{DC}
Data output and $\overline{INTR}$						
V_{OL}	Logical “0” output voltage					
	Data outputs	I_{OL} = 1.6mA, $V_{CC} = 4.75V_{DC}$			0.4	V_{DC}
	$\overline{INTR}$ outputs	I_{OL} = 1.0mA, $V_{CC} = 4.75V_{DC}$			0.4	V_{DC}
V_{OH}	Logical “1” output voltage	I_{OH} = –360µA, $V_{CC} = 4.75V_{DC}$	2.4			V_{DC}
		I_{OH} = –10µA, $V_{CC} = 4.75V_{DC}$	4.5			
I_{OZL}	3-state output leakage	$V_{OUT} = 0V_{DC}$, $\overline{CS}$ = logical “1”	–3			μA_{DC}
I_{OZH}	3-state output leakage	$V_{OUT} = 5V_{DC}$, $\overline{CS}$ = logical “1”			3	μA_{DC}
I_{SC}	+Output short-circuit current	V_{OUT} = 0V, T_A = 25°C	4.5	12		mA_{DC}
I_{SC}	–Output short-circuit current	$V_{OUT} = V_{CC}$, T_A = 25°C	9.0	30		mA_{DC}
I_{CC}	Power supply current	f_{CLK} = 1MHz, $V_{REF}/2$ = OPEN, $\overline{CS}$ = Logical “1”, T_A = 25°C		3.0	3.5	mA

NOTES:
1. Analog inputs must remain within the range: $–0.05 \le V_{IN} \le V_{CC} + 0.05V$.
2. See typical performance characteristics for input resistance at V_{CC} = 5V.
3. $V_{REF}/2$ and V_{IN} must be applied after the V_{CC} has been turned on to prevent the possibility of latching.

AC ELECTRICAL CHARACTERISTICS

SYMBOL	PARAMETER	TO	FROM	TEST CONDITIONS	ADC0803/4			UNIT
					Min	Typ	Max	
	Conversion time			f_{CLK}=1MHz[1]	66		73	µs
f_{CLK}	Clock frequency[1]				0.1	1.0	3.0	MHz
	Clock duty cycle[1]				40		60	%
CR	Free-running conversion rate			$\overline{CS}$=0, f_{CLK}=1MHz $\overline{INTR}$ tied to $\overline{WR}$			13690	conv/s
$t_{W(\overline{WR})L}$	Start pulse width			$\overline{CS}$=0	30			ns
t_{ACC}	Access time	Output	$\overline{RD}$	$\overline{CS}$=0, C_L=100pF		75	100	ns
t_{1H}, t_{0H}	3-State control	Output	$\overline{RD}$	CL=10pF, R_L=10kΩ See 3-State test circuit		70	100	ns
t_{W1}, t_{R1}	$\overline{INTR}$ delay	$\overline{INTR}$	$\overline{WD}$ or $\overline{RD}$			100	150	ns
C_{IN}	Logic input=capacitance					5	7.5	pF
C_{OUT}	3-State output capacitance					5	7.5	pF

NOTES:
1. Accuracy is guaranteed at f_{CLK}=1MHz. Accuracy may degrade at higher clock frequencies.

FUNCTIONAL DESCRIPTION

These devices operate on the Successive Approximation principle. Analog switches are closed sequentially by successive approximation logic until the input to the auto-zero comparator [$V_{IN}(+)$-$V_{IN}(-)$] matches the voltage from the decoder. After all bits are tested and determined, the 8-bit binary code corresponding to the input voltage is transferred to an output latch. Conversion begins with the arrival of a pulse at the $\overline{WR}$ input if the CS input is low. On the High-to-Low transition of the signal at the WR or the CS input, the SAR is initialized, the shift register is reset, and the $\overline{INTR}$ output is set high. The A/D will remain in the reset state as long as the $\overline{CS}$ and $\overline{WR}$ inputs remain low. Conversion will start from one to eight clock periods after one or both of these inputs makes a Low-to-High transition. After the conversion is complete, the $\overline{INTR}$ pin will make a High-to-Low transition. This can be used to interrupt a processor, or otherwise signal the availability of a new conversion result. A read ($\overline{RD}$) operation (with $\overline{CS}$ low) will clear the $\overline{INTR}$ line and enable the output latches. The device may be run in the free-running mode as described later. A conversion in progress can be interrupted by issuing another start command.

Digital Control Inputs

The digital control inputs (CS, WR, $\overline{RD}$) are compatible with standard TTL logic voltage levels. The required signals at these inputs correspond to Chip Select, START Conversion, and Output Enable control signals, respectively. They are active-Low for easy interface to microprocessor and microcontroller control buses. For applications not using microprocessors, the $\overline{CS}$ input (Pin 1) can be grounded and the A/D START function is achieved by a negative-going pulse to the $\overline{WR}$ input (Pin 3). The Output Enable function is achieved by a logic low signal at the $\overline{RD}$ input (Pin 2), which may be grounded to constantly have the latest conversion present at the output.

ANALOG OPERATION

Analog Input Current

The analog comparisons are performed by a capacitive charge summing circuit. The input capacitor is switched between $V_{IN(+)}$4 and $V_{IN(-)}$, while reference capacitors are switched between taps on the reference voltage divider string. The net charge corresponds to the weighted difference between the input and the most recent total value set by the successive approximation register.

The internal switching action causes displacement currents to flow at the analog inputs. The voltage on the on-chip capacitance is switched through the analog differential input voltage, resulting in proportional currents entering the $V_{IN(+)}$ input and leaving the $V_{IN(-)}$ input. These transient currents occur at the leading edge of the internal clock pulses. They decay rapidly so do not inherently cause errors as the on-chip comparator is strobed at the end of the clock period.

Input Bypass Capacitors and Source Resistance

Bypass capacitors at the input will average the charges mentioned above, causing a DC and an AC current to flow through the output resistance of the analog signal sources. This charge pumping action is worse for continuous conversions with the $V_{IN(+)}$ input at full scale. This current can be a few microamps, so bypass capacitors should NOT be used at the analog inputs of the V_{REF}/2 input for high resistance sources (> 1kΩ). If input bypass capacitors are desired for noise filtering and a high source resistance is desired to minimize capacitor size, detrimental effects of the voltage drop across the input resistance can be eliminated by adjusting the full scale with both the input resistance and the input bypass capacitor in place. This is possible because the magnitude of the input current is a precise linear function of the differential voltage.

Large values of source resistance where an input bypass capacitor is not used will not cause errors as the input currents settle out prior to the comparison time. If a low pass filter is required in the system, use a low valued series resistor (< 1kΩ) for a passive RC section or add an op amp active filter (low pass). For applications with source resistances at or below 1kΩ, a 0.1µF bypass capacitor at the inputs will prevent pickup due to series lead inductance or a long wire. A 100Ω series resistor can be used to isolate this capacitor (both the resistor and capacitor should be placed out of the feedback loop) from the output of the op amp, if used.

Analog Differential Voltage Inputs and Common-Mode Rejection

These A/D converters have additional flexibility due to the analog differential voltage input. The $V_{IN(-)}$ input (Pin 7) can be used to subtract a fixed voltage from the input reading (tare correction). This is also useful in a 4/20mA current loop conversion. Common-mode noise can also be reduced by the use of the differential input.

The time interval between sampling $V_{IN(+)}$ and $V_{IN(-)}$ is 4.5 clock periods. The maximum error due to this time difference is given by:

$V(max)=(V_P)\ (2f_{CM})\ (4.5/f_{CLK})$,

where:

V=error voltage due to sampling delay

V_P=peak value of common-mode voltage

f_{CM}=common mode frequency

For example, with a 60Hz common-mode frequency, f_{cm}, and a 1MHz A/D clock, F_{CLK}, keeping this error to 1/4 LSB (about 5mV) would allow a common-mode voltage, V_P, which is given by:

$$V_P = \frac{[V(max)\ (f_{CLK})}{(2f_{CM})(4.5)}$$

or

$$V_P = \frac{(5 \times 10^{-3})\ (10^4)}{(6.28)\ (60)\ (4.5)} = 2.95V$$

The allowed range of analog input voltages usually places more severe restrictions on input common-mode voltage levels than this, however.

An analog input span less than the full 5V capability of the device, together with a relatively large zero offset, can be easily handled by use of the differential input. (See Reference Voltage Span Adjust).

Noise and Stray Pickup

The leads of the analog inputs (Pins 6 and 7) should be kept as short as possible to minimize input noise coupling and stray signal pick-up. Both EMI and undesired digital signal coupling to these inputs can cause system errors. The source resistance for these inputs should generally be below 5kΩ to help avoid undesired noise pickup. Input bypass capacitors at the analog inputs can create errors as described previously. Full scale adjustment with any input bypass capacitors in place will eliminate these errors.

Reference Voltage

For application flexibility, these A/D converters have been designed to accommodate fixed reference voltages of 5V to Pin 20 or 2.5V to Pin 9, or an adjusted reference voltage at Pin 9. The reference can be set by forcing it at $V_{REF}/2$ input, or can be determined by the supply voltage (Pin 20). Figure 1 indicates how this is accomplished.

Reference Voltage Span Adjust

Note that the Pin 9 ($V_{REF}/2$) voltage is either 1/2 the voltage applied to the V_{CC} supply pin, or is equal to the voltage which is externally forced at the $V_{REF}/2$ pin. In addition to allowing for flexible references and full span voltages, this also allows for a ratiometric voltage reference. The internal gain of the $V_{REF}/2$ input is 2, making the full-scale differential input voltage twice the voltage at Pin 9.

For example, a dynamic voltage range of the analog input voltage that extends from 0 to 4V gives a span of 4V (4-0), so the $V_{REF}/2$ voltage can be made equal to 2V (half of the 4V span) and full scale output would correspond to 4V at the input.

On the other hand, if the dynamic input voltage had a range of 0.5 to 3.5V, the span or dynamic input range is 3V (3.5-0.5). To encode this 3V span with 0.5V yielding a code of zero, the minimum expected input (0.5V, in this case) is applied to the $V_{IN}(-)$ pin to account for the offset, and the $V_{REF}/2$ pin is set to 1/2 the 3V span, or 1.5V. The A/D converter will now encode the $V_{IN}(+)$ signal between 0.5 and 3.5V with 0.5V at the input corresponding to a code of zero and 3.5V at the input producing a full scale output code. The full 8 bits of resolution are thus applied over this reduced input voltage range. The required connections are shown in Figure 2.

Operating Mode

These converters can be operated in two modes:

1) absolute mode
2) ratiometric mode

In absolute mode applications, both the initial accuracy and the temperature stability of the reference voltage are important factors in the accuracy of the conversion. For $V_{REF}/2$ voltages of 2.5V, initial errors of ±10mV will cause conversion errors of ±1 LSB due to the gain of 2 at the $V_{REF}/2$ input. In reduced span applications, the initial value and stability of the $V_{REF}/2$ input voltage become even more important as the same error is a larger percentage of the $V_{REF}/2$ nominal value. See Figure 3.

In ratiometric converter applications, the magnitude of the reference voltage is a factor in both the output of the source transducer and the output of the A/D converter, and, therefore, cancels out in the final digital code. See Figure 4.

Generally, the reference voltage will require an initial adjustment. Errors due to an improper reference voltage value appear as full-scale errors in the A/D transfer function.

ERRORS AND INPUT SPAN ADJUSTMENTS

There are many sources of error in any data converter, some of which can be adjusted out. Inherent errors, such as relative accuracy, cannot be eliminated, but such errors as full-scale and zero scale offset errors can be eliminated quite easily. See Figure 2.

Zero Scale Error

Zero scale error of an A/D is the difference of potential between the ideal 1/2 LSB value (9.8mV for $V_{REF}/2$=2.500V) and that input voltage which just causes an output transition from code 0000 0000 to a code of 0000 0001.

If the minimum input value is not ground potential, a zero offset can be made. The converter can be made to output a digital code of 0000 0000 for the minimum expected input voltage by biasing the $V_{IN}(-)$ input to that minimum value expected at the $V_{IN}(-)$ input to that minimum value expected at the $V_{IN}(+)$ input. This uses the

differential mode of the converter. Any offset adjustment should be done prior to full scale adjustment.

Full Scale Adjustment

Full scale gain is adjusted by applying any desired offset voltage to $V_{IN}(-)$, then applying the $V_{IN}(+)$ a voltage that is 1-1/2 LSB less than the desired analog full-scale voltage range and then adjusting the magnitude of $V_{REF}/2$ input voltage (or the V_{CC} supply if there is no $V_{REF}/2$ input connection) for a digital output code which just changes from 1111 1110 to 1111 1111. The ideal $V_{IN}(+)$ voltage for this full-scale adjustment is given by:

$$V_{IN}(\ \ ) = V_{IN}(-) - 1.5 \times \frac{V_{MAX} - V_{MIN}}{255}$$

where:

V_{MAX}=high end of analog input range (ground referenced)

V_{MIN}=low end (zero offset) of analog input (ground referenced)

CLOCKING OPTION

The clock signal for these A/Ds can be derived from external sources, such as a system clock, or self-clocking can be accomplished by adding an external resistor and capacitor, as shown in Figure 6.

Heavy capacitive or DC loading of the CLK R pin should be avoided as this will disturb normal converter operation. Loads less than 50pF are allowed. This permits driving up to seven A/D converter CLK IN pins of this family from a single CLK R pin of one converter. For larger loading of the clock line, a CMOS or low power TTL buffer or PNP input logic should be used to minimize the loading on the CLK R pin.

Restart During a Conversion

A conversion in process can be halted and a new conversion began by bringing the $\overline{CS}$ and $\overline{WR}$ inputs low and allowing at least one of them to go high again. The output data latch is not updated if the conversion in progress is not completed; the data from the previously completed conversion will remain in the output data latches until a subsequent conversion is completed.

Continuous Conversion

To provide continuous conversion of input data, the $\overline{CS}$ and $\overline{RD}$ inputs are grounded and $\overline{INTR}$ output is tied to the $\overline{WR}$ input. This $\overline{INTR}/\overline{WR}$ connection should be momentarily forced to a logic low upon power-up to insure circuit operation. See Figure 5 for one way to accomplish this.

DRIVING THE DATA BUS

This CMOS A/D converter, like MOS microprocessors and memories, will require a bus driver when the total capacitance of the data bus gets large. Other circuitry tied to the data bus will add to the total capacitive loading, even in the high impedance mode.

There are alternatives in handling this problem. The capacitive loading of the data bus slows down the response time, although DC specifications are still met. For systems with a relatively low CPU clock frequency, more time is available in which to establish proper logic levels on the bus, allowing higher capacitive loads to be driven (see Typical Performance Characteristics).

At higher CPU clock frequencies, time can be extended for I/O reads (and/or writes) by inserting wait states (8880) or using clock-extending circuits (6800, 8035).

Finally, if time is critical and capacitive loading is high, external bus drivers must be used. These can be 3-State buffers (low power Schottky is recommended, such as the N74LS240 series) or special higher current drive products designed as bus drivers. High current bipolar bus drivers with PNP inputs are recommended as the PNP input offers low loading of the A/D output, allowing better response time.

POWER SUPPLIES

Noise spikes on the V_{CC} line can cause conversion errors as the internal comparator will respond to them. A low inductance filter capacitor should be used close to the converter V_{CC} pin and values of 1µF or greater are recommended. A separate 5V regulator for the converter (and other 5V linear circuitry) will greatly reduce digital noise on the V_{CC} supply and the attendant problems.

WIRING AND LAYOUT PRECAUTIONS

Digital wire-wrap sockets and connections are not satisfactory for breadboarding this (or any) A/D converter. Sockets on PC boards can be used. All logic signal wires and leads should be grouped or kept as far as possible from the analog signal leads. Single wire analog input leads may pick up undesired hum and noise, requiring the use of shielded leads to the analog inputs in many applications.

A single-point analog ground separate from the logic or digital ground points should be used. The power supply bypass capacitor and the self-clocking capacitor, if used, should be returned to digital ground. Any $V_{REF}/2$ bypass capacitor, analog input filter capacitors, and any input shielding should be returned to the analog ground point. Proper grounding will minimize zero-scale errors which are present in every code. Zero-scale errors can usually be traced to improper board layout and wiring.

APPLICATIONS

Microprocessor Interfacing

This family of A/D converters was designed for easy microprocessor interfacing. These converters can be memory mapped with appropriate memory address decoding for $\overline{CS}$ (read) input. The active-Low write pulse from the processor is then connected to the $\overline{WR}$ input of the A/D converter, while the processor active-Low read pulse is fed to the converter $\overline{RD}$ input to read the converted data. If the clock signal is derived from the microprocessor system clock, the designer/programmer should be sure that there is no attempt to read the converter until 74 converter clock pulses after the start pulse goes high. Alternatively, the $\overline{INTR}$ pin may be used to interrupt the processor to cause reading of the converted data. Of course, the converter can be connected and addressed as a peripheral (in I/O space), as shown in Figure 7. A bus driver should be used as a buffer to the A/D output in large microprocessor systems where the data leaves the PC board and/or must drive capacitive loads in excess of 100pF. See Figure 9.

Interfacing the SCN8048 microcomputer family is pretty simple, as shown in Figure 8. Since the SCN8048 family has 24 I/O lines, one of these (shown here as bit 0 or port 1) can be used as the chip select signal to the converter, eliminating the need for an address

decoder. The $\overline{RD}$ and $\overline{WR}$ signals are generated by reading from and writing to a dummy address.

Digitizing a Transducer Interface Output

Circuit Description

Figure 10 shows an example of digitizing transducer interface output voltage. In this case, the transducer interface is the NE5521, an LVDT (Linear Variable Differential Transformer) Signal Conditioner. The diode at the A/D input is used to insure that the input to the A/D does not go excessively beyond the supply voltage of the A/D. See the NE5521 data sheet for a complete description of the operation of that part.

Circuit Adjustment

To adjust the full scale and zero scale of the A/D, determine the range of voltages that the transducer interface output will take on. Set the LVDT core for null and set the Zero Scale Scale Adjust Potentiometer for a digital output from the A/D of 1000 000. Set the LVDT core for maximum voltage from the interface and set the Full Scale Adjust potentiometer so the A/D output is just barely 1111 1111.

A Digital Thermostat

Circuit Description

The schematic of a Digital Thermostat is shown in Figure 11. The A/D digitizes the output of the LM35, a temperature transducer IC with an output of 10mV per °C. With $V_{REF}/2$ set for 2.56V, this 10mV corresponds to 1/2 LSB and the circuit resolution is 2°C. Reducing $V_{REF}/2$ to 1.28 yields a resolution of 1°C. Of course, the lower $V_{REF}/2$ is, the more sensitive the A/D will be to noise.

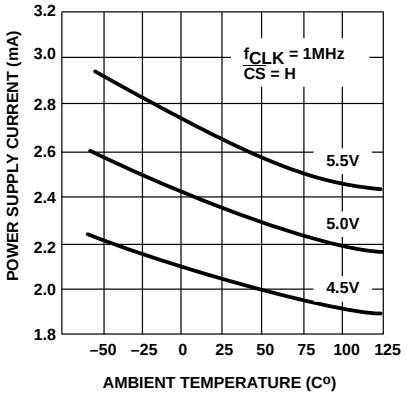
The desired temperature is set by holding either of the set buttons closed. The SCC80C451 programming could cause the desired (set) temperature to be displayed while either button is depressed and for a short time after it is released. At other times the ambient temperature could be displayed.

The set temperature is stored in an SCN8051 internal register. The A/D conversion is started by writing anything at all to the A/D with port pin P10 set high. The desired temperature is compared with the digitized actual temperature, and the heater is turned on or off by clearing setting port pin P12. If desired, another port pin could be used to turn on or off an air conditioner.

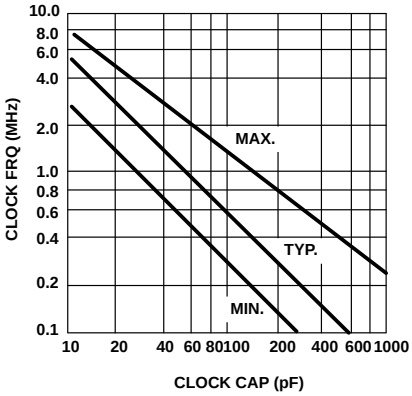
The display drivers are NE587s if common anode LED displays are used. Of course, it is possible to interface to LCD displays as well.

TYPICAL PERFORMANCE CHARACTERISTICS

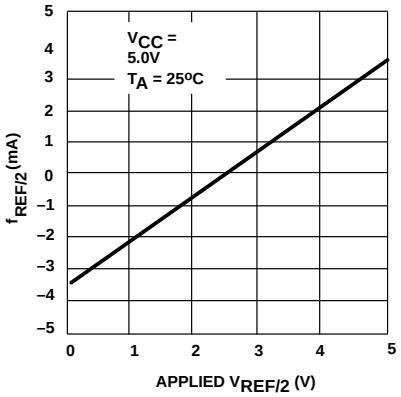
Power Supply Current vs Temperature



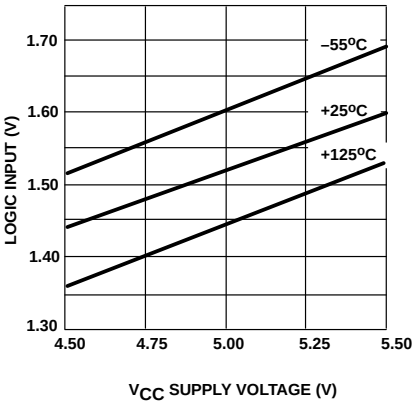
Clock Frequency vs Clock Capacitor



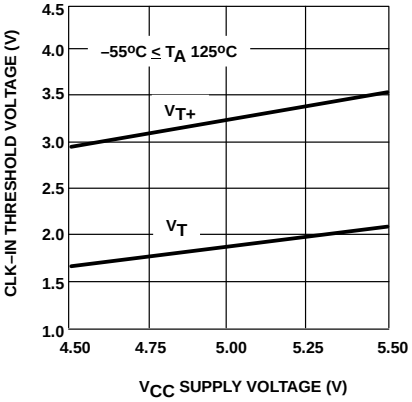
Input Current vs Applied Voltage at $V_{REF/2}$ Pin



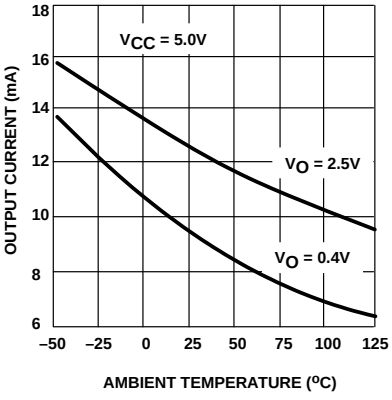
Logic Input Threshold Voltage vs Supply Voltage



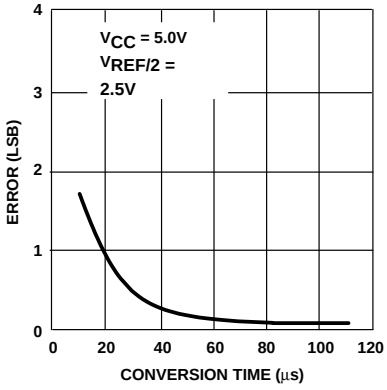
CLK–IN Threshold Voltage vs Supply Voltage



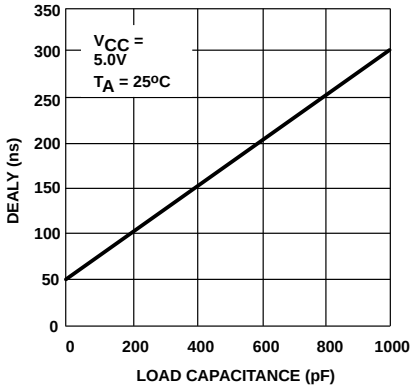
Output Current vs Temperature



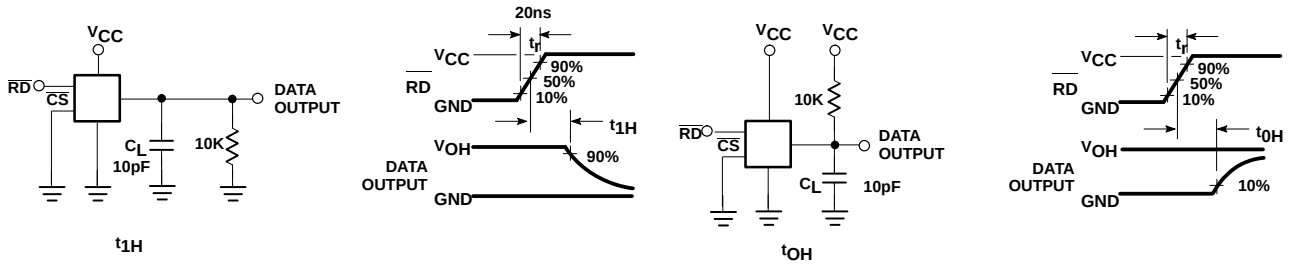
Full Scale Error vs Conversion Time



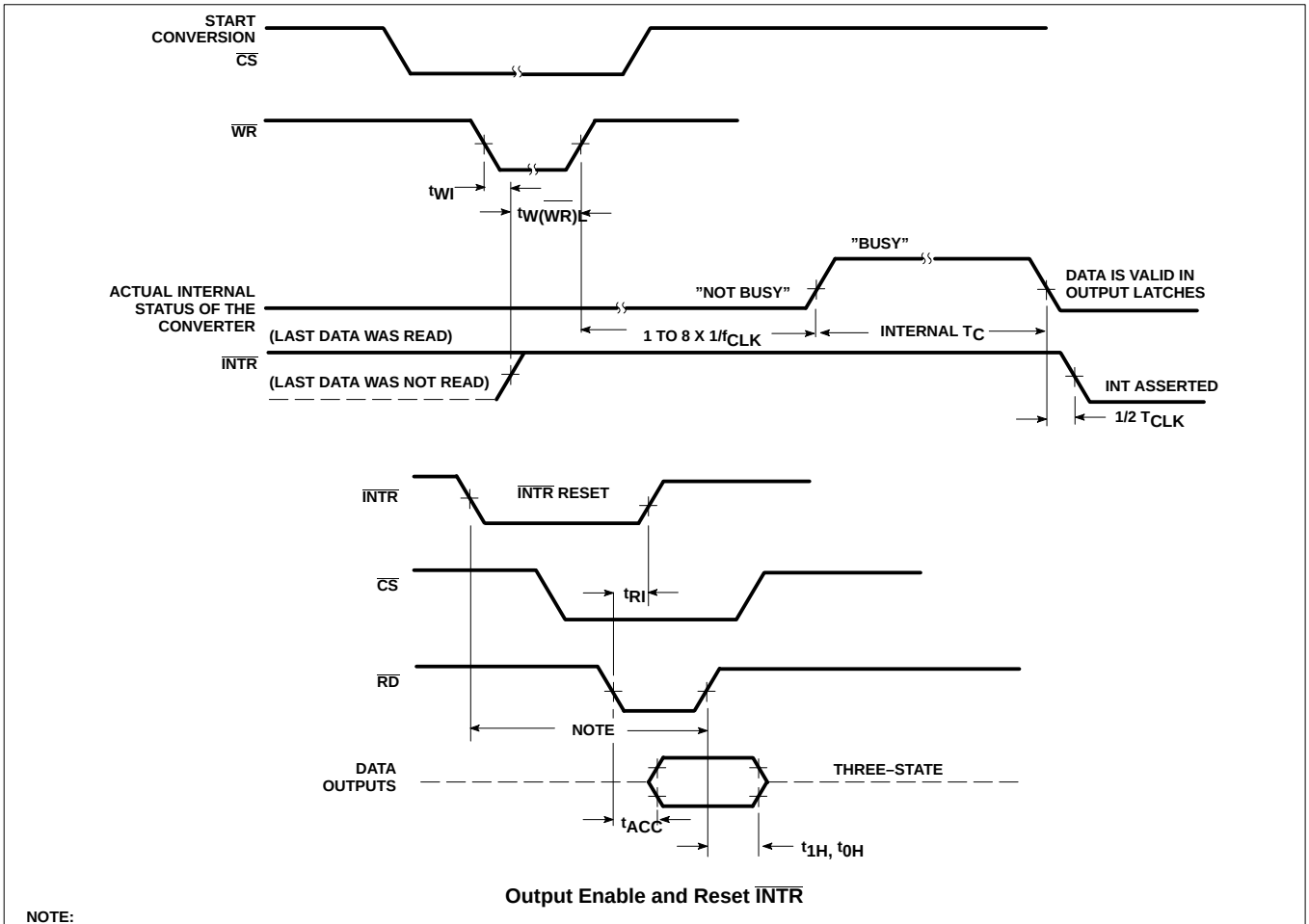
Delay From $\overline{RD}$ Falling Edge to Data Valid vs Load Capacitance



3-STATE TEST CIRCUITS AND WAVEFORMS (ADC0801-1)

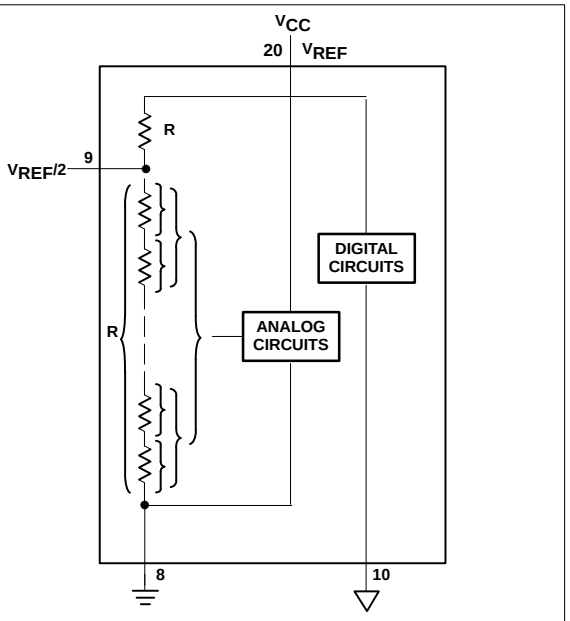


TIMING DIAGRAMS (All timing is measured from the 50% voltage points)



Output Enable and Reset $\overline{INTR}$

NOTE:
Read strobe must occur 8 clock periods ($8/f_{CLK}$) after assertion of interrupt to guarantee reset of $\overline{INTR}$.



NOTE:
The $V_{REF}/2$ voltage is either 1/2 the V_{CC} voltage or is that which is forced at Pin 9.

Figure 1. Internal Reference Design

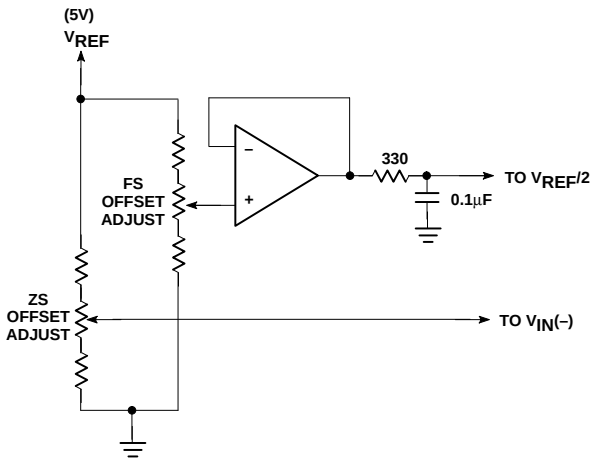
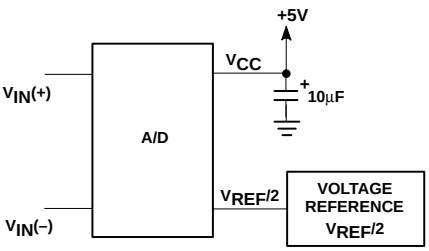
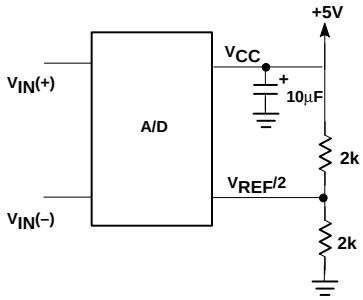


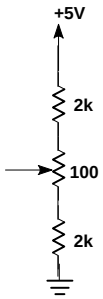
Figure 2. Offsetting the Zero Scale and Adjusting the Input Range (Span)



a. Fixed Reference



b. Fixed Reference Derived from V_{CC}



c. Optional Full Scale Adjustment

Figure 3. Absolute Mode of Operation

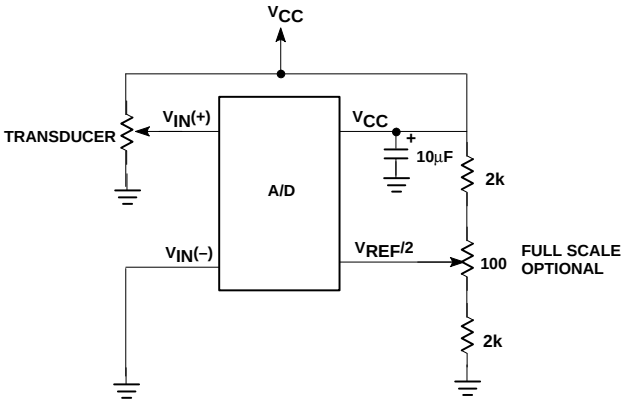


Figure 4. Ratiometric Mode of Operation with Optional Full Scale Adjustment

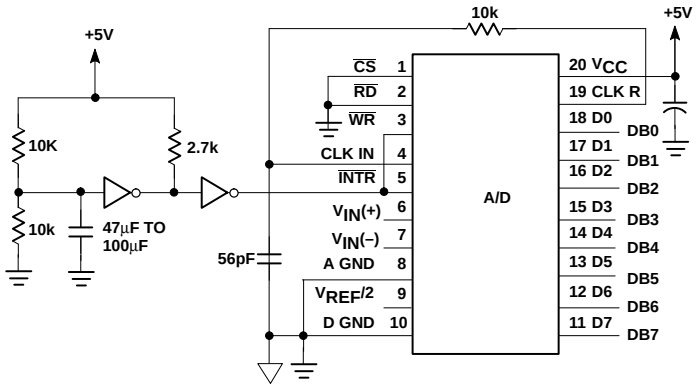


Figure 5. Connection for Continuous Conversion

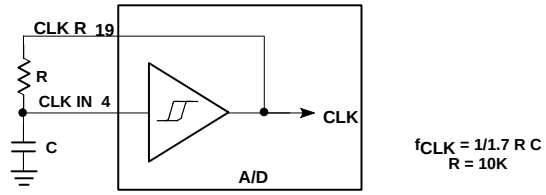


Figure 6. Self-Clocking the Converter

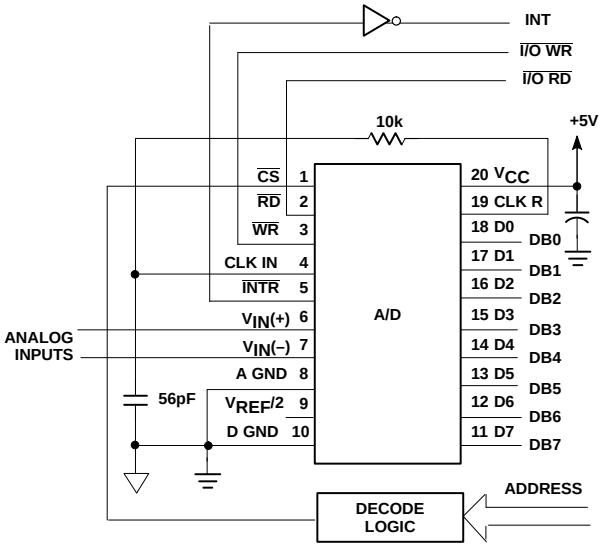


Figure 7. Interfacing to 8080A Microprocessor

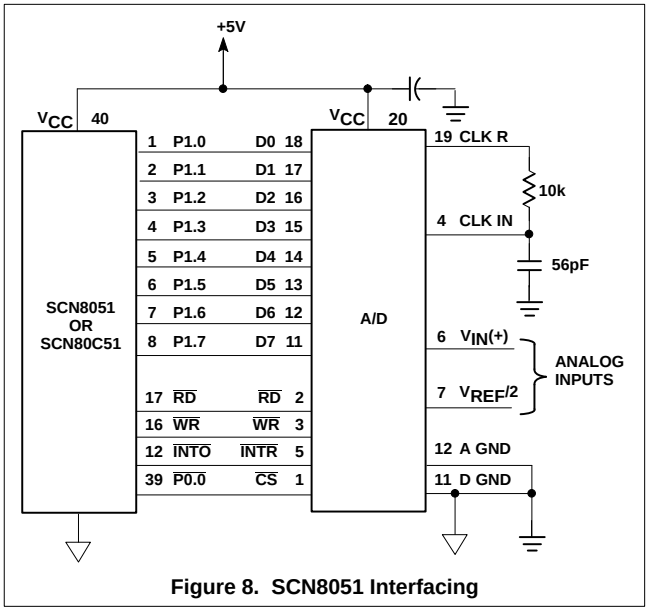


Figure 8. SCN8051 Interfacing

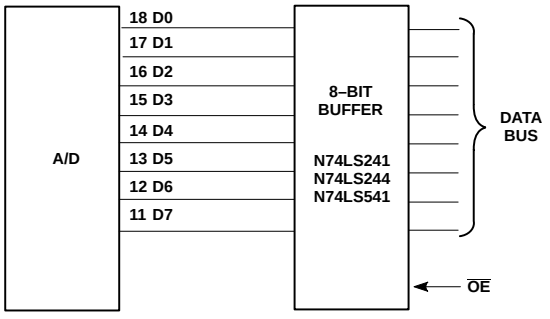


Figure 9. Buffering the A/D Output to Drive High Capacitance Loads and for Driving Off-Board Loads

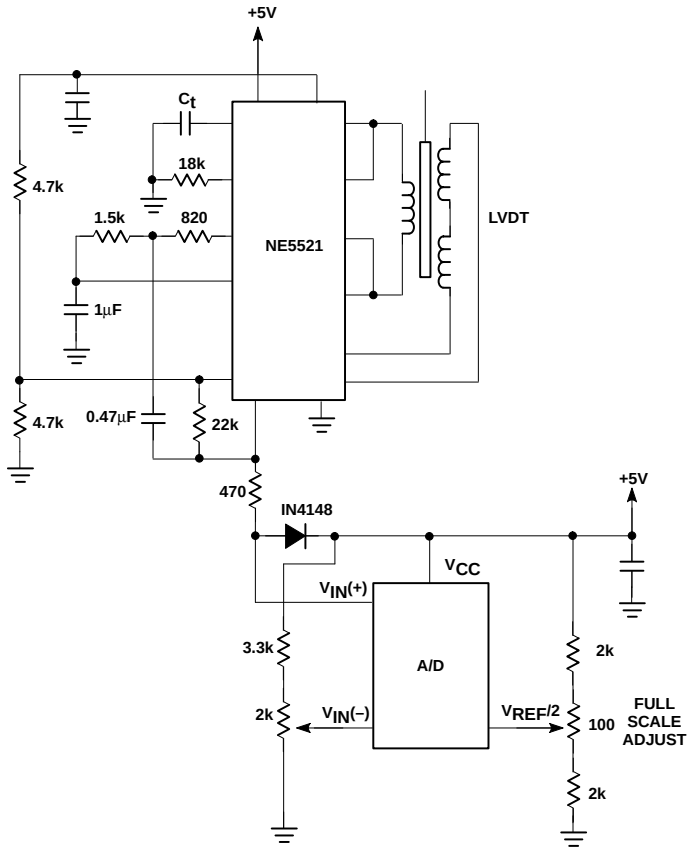


Figure 10. Digitizing a Transducer Interface Output

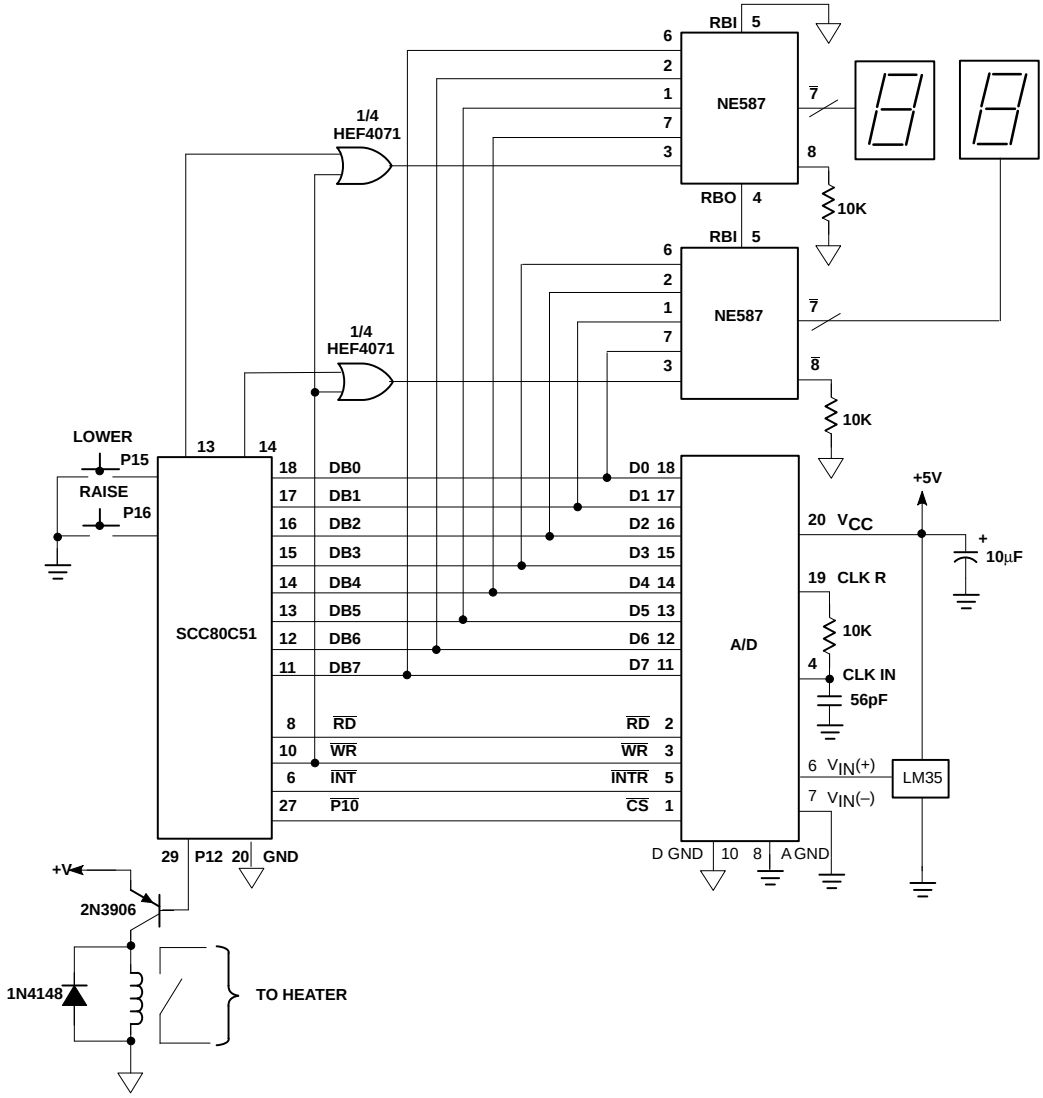


Figure 11. Digital Thermostat

8-Bit high-speed multiplying D/A converter — DAC08 Series

DESCRIPTION

The DAC08 series of 8-bit monolithic multiplying Digital-to-Analog Converters provide very high-speed performance coupled with low cost and outstanding applications flexibility.

Advanced circuit design achieves 70ns settling times with very low glitch and at low power consumption. Monotonic multiplying performance is attained over a wide 20-to-1 reference current range. Matching to within 1 LSB between reference and full-scale currents eliminates the need for full-scale trimming in most applications. Direct interface to all popular logic families with full noise immunity is provided by the high swing, adjustable threshold logic inputs.

Dual complementary outputs are provided, increasing versatility and enabling differential operation to effectively double the peak-to-peak output swing. True high voltage compliance outputs allow direct output voltage conversion and eliminate output op amps in many applications.

All DAC08 series models guarantee full 8-bit monotonicity and linearities as tight as 0.1% over the entire operating temperature range. Device performance is essentially unchanged over the ±4.5V to ±18V power supply range, with 37mW power consumption attainable at ±5V supplies.

The compact size and low power consumption make the DAC08 attractive for portable and military aerospace applications.

FEATURES

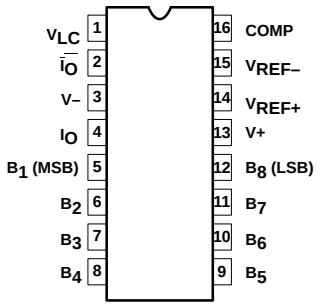
- Fast settling output current—70ns
- Full-scale current prematched to ±1 LSB
- Direct interface to TTL, CMOS, ECL, HTL, PMOS
- Relative accuracy to 0.1% maximum over temperature range
- High output compliance -10V to +18V
- True and complemented outputs
- Wide range multiplying capability
- Low FS current drift — ±10ppm/°C
- Wide power supply range—±4.5V to ±18V
- Low power consumption—37mW at ±5V

APPLICATIONS

- 8-bit, 1µs A-to-D converters
- Servo-motor and pen drivers

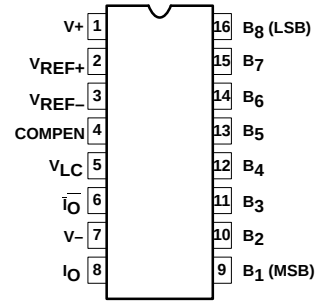
PIN CONFIGURATIONS

F, N Packages



TOP VIEW

D[1] Package



TOP VIEW

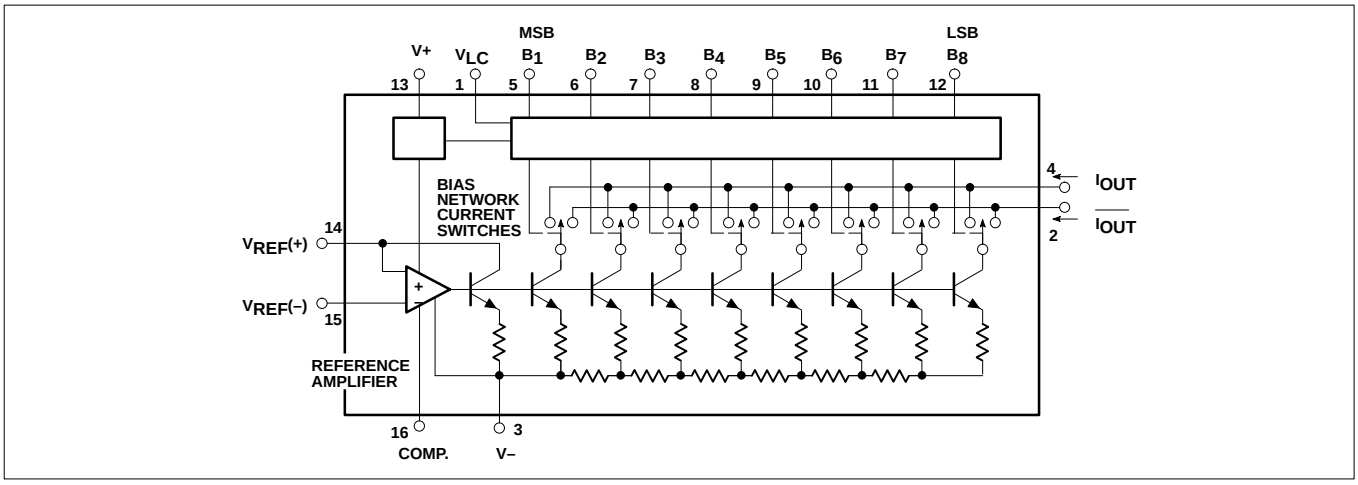
NOTE:
1. SO and non-standard pinouts.

- Waveform generators
- Audio encoders and attenuators
- Analog meter drivers
- Programmable power supplies
- CRT display drivers
- High-speed modems
- Other applications where low cost, high speed and complete input/output versatility are required
- Programmable gain and attenuation
- Analog-Digital multiplication

ORDERING INFORMATION

DESCRIPTION	TEMPERATURE RANGE	ORDER CODE	DWG #
16-Pin Hermetic Ceramic Dual In-Line Package (Cerdip)	-55°C to +125°C	DAC08F	0582B
16-Pin Hermetic Ceramic Dual In-Line Package (Cerdip)	-55°C to +125°C	DAC08AF	0582B
16-Pin Plastic Dual In-Line Package (DIP)	0 to +70°C	DAC08CN	0406C
16-Pin Hermetic Ceramic Dual In-Line Package (Cerdip)	0 to +70°C	DAC08CF	0582B
16-Pin Plastic Dual In-Line Package (DIP)	0 to +70°C	DAC08EN	0406C
16-Pin Hermetic Ceramic Dual In-Line Package (Cerdip)	0 to +70°C	DAC08EF	0582B
16-Pin Plastic Small Outline (SO) Package	0 to +70°C	DAC08ED	0005D
16-Pin Plastic Dual In-Line Package (DIP)	0 to +70°C	DAC08HN	0406C

BLOCK DIAGRAM



ABSOLUTE MAXIMUM RATINGS

SYMBOL	PARAMETER	RATING	UNIT
V+ to V-	Power supply voltage	36	V
V_5-V_{12}	Digital input voltage	V- to V- plus 36V	
V_{LC}	Logic threshold control	V- to V+	
V_0	Applied output voltage	V- to +18	V
I_{14}	Reference current	5.0	mA
V_{14}, V_{15}	Reference amplifier inputs	V_{EE} to V_{CC}	
P_D	Maximum power dissipation T_A=25°C (still-air)[1]		
	F package	1190	mW
	N package	1450	mW
	D package	1090	mW
T_{SOLD}	Lead soldering temperature (10sec max)	300	°C
T_A	Operating temperature range		
	DAC08, DAC08A	-55 to +125	°C
	DAC08C, E, H	0 to +70	°C
T_{STG}	Storage temperature range	-65 to +150	°C

NOTES:
1. Derate above 25°C, at the following rates:
   F package at 9.5mW/°C
   N package at 11.6mW/°C
   D package at 8.7mW/°C

DC ELECTRICAL CHARACTERISTICS

Pin 3 must be at least 3V more negative than the potential to which R_{15} is returned. V_{CC}=±15V, I_{REF}=2.0mA. Output characteristics refer to both I_{OUT} and $\overline{I_{OUT}}$ unless otherwise noted. DAC08C, E, H: T_A=0°C to 70°C DAC08/08A: T_A=-55°C to 125°C

SYMBOL	PARAMETER	TEST CONDITIONS	DAC08C			DAC08E DAC08			UNIT
			Min	Typ	Max	Min	Typ	Max	
	Resolution		8	8	8	8	8	8	Bits
	Monotonicity		8	8	8	8	8	8	Bits
	Relative accuracy	Over temperature range			±0.39			±0.19	%FS
	Differential non-linearity				±0.78			±0.39	%FS
TCI_{FS}	Full-scale tempco			±10			±10		ppm/°C
V_{OC}	Output voltage compliance	Full-scale current change< 1/2LSB	-10		+18	-10		+18	V
I_{FS4}	Full-scale current	V_{REF}=10.000V, R_{14}, R_{15}=5.000kΩ	1.94	1.99	2.04	1.94	1.99	2.04	mA
I_{FSS}	Full-scale symmetry	I_{FS4}-I_{FS2}		±2.0	±16		±1.0	±8.0	µA
I_{ZS}	Zero-scale current			0.2	4.0		0.2	2.0	µA
I_{FSR}	Full-scale output current range	R_{14}, R_{15}=5.000kΩ V_{REF}=+15.0V, V-=-10V V_{REF}=+25.0V, V-=-12V	2.1 4.2			2.1 4.2			mA
V_{IL} V_{IH}	Logic input levels Low High	V_{LC}=0V	 2.0		0.8	 2.0		0.8	V
I_{IL} I_{IH}	Logic input current Low High	V_{LC}=0V V_{IN}=-10V to +0.8V V_{IN}=2.0V to 18V		-2.0 0.002	-10 10		-2.0 0.002	-10 10	µA
V_{IS}	Logic input swing	V-=-15V	-10		+18	-10		+18	V
V_{THR}	Logic threshold range	V_S=±15V	-10		+13.5	-10		+13.5	V
I_{15}	Reference bias current			-1.0	-3.0		-1.0	-3.0	µA
dI/dt	Reference input slew rate		4.0	8.0		4.0	8.0		mA/µs
$PSSI_{FS+}$ PSI_{FS-}	Power supply sensitivity Positive Negative	I_{REF}=1mA V+=4.5 to 5.5V, V-=-15V; V+=13.5 to 16.5V, V-=-15V V-=-4.5 to -5.5V, V+=+15V; V-=-13.5 to -16.5, V+=+15V		0.0003 0.002	0.01 0.01		0.0003 0.002	0.01 0.01	%FS/%VS
I+ I-	Power supply current Positive Negative	V_S=±5V, I_{REF}=1.0mA		3.1 -4.3	3.8 -5.8		3.1 -4.3	3.8 -5.8	mA
I+ I-	Positive Negative	V_S=+5V, -15V, I_{REF}=2.0mA		3.1 -7.1	3.8 -7.8		3.1 -7.1	3.8 -7.8	
I+ I-	Positive Negative	V_S=±15V, I_{REF}=2.0mA		3.2 -7.2	3.8 -7.8		3.2 -7.2	3.8 -7.8	
P_D	Power dissipation	±5V, I_{REF}=1.0mA +5V, -15V, I_{REF}=2.0mA ±15V, I_{REF}=2.0mA		37 122 156	48 136 174		37 122 156	48 136 174	mW

DC ELECTRICAL CHARACTERISTICS (Continued)

Pin 3 must be at least 3V more negative than the potential to which R15 is returned. V_{CC} = +15V, I_{REF} = 2.0mA, Output characteristics refer to both I_{OUT} and $I_{\overline{OUT}}$, unless otherwise noted. DAC08C, E, H: T_A = 0°C to 70°C. DAC08/08A: T_A = -55°C to 125°C.

SYMBOL	PARAMETER	TEST CONDITIONS	DAC08H DAC08A			UNIT
			Min	Typ	Max	
	Resolution		8	8	8	Bits
	Monotonicity		8	8	8	Bits
	Relative accuracy	Over temperature range			±0.1	%FS
	Differential non-linearity				±0.19	%FS
TCI_{FS}	Full-scale tempco			±10	±50	ppm/°C
V_{OC}	Output voltage compliance	Full-scale current change 1/2LSB	-10		+18	V
I_{FS4}	Full-scale current	V_{REF}=10.000V, R_{14}, R_{15}=5.000kΩ	1.984	1.992	2.000	mA
I_{FSS}	Full-scale symmetry	I_{FS4}-I_{FS2}		±1.0	±4.0	µA
I_{ZS}	Zero-scale current			0.2	1.0	µA
I_{FSR}	Full-scale output current range	R_{14}, R_{15}=5.000kΩ V_{REF}=+15.0V, V-=-10V V_{REF}=+25.0V, V-=-12V	 2.1 4.2			mA
	Logic input levels					
V_{IL}	Low	V_{LC}=0V			0.8	V
V_{IH}	High		2.0			
	Logic input current	V_{LC}=0V				
I_{IL}	Low	V_{IN}=-10V to +0.8V		-2.0	-10	µA
I_{IH}	High	V_{IN}=2.0V to 18V		0.002	10	
V_{IS}	Logic input swing	V-=-15V	-10		+18	V
V_{THR}	Logic threshold range	V_S=±15V	-10		+13.5	V
I_{15}	Reference bias current			-1.0	-3.0	µA
dI/dt	Reference input slew rate		4.0	8.0		mA/µs
	Power supply sensitivity	I_{REF}=1mA				
$PSSI_{FS+}$	Positive	V+=4.5 to 5.5V, V-=-15V; V+=13.5 to 16.5V, V-=-15V		0.0003	0.01	%FS/%VS
PSI_{FS-}	Negative	V-=-4.5 to -5.5V, V+=+15V; V-=-13.5 to -16.5, V+=+15V		0.002	0.01	
	Power supply current					
I+	Positive	V_S=±5V, I_{REF}=1.0mA		3.1	3.8	
I-	Negative			-4.3	-5.8	
I+	Positive	V_S=+5V, -15V, I_{REF}=2.0mA		3.1	3.8	mA
I-	Negative			-7.1	-7.8	
I+	Positive	V_S=±15V, I_{REF}=2.0mA		3.2	3.8	
I-	Negative			-7.2	-7.8	
		±5V, I_{REF}=1.0mA		37	48	
P_D	Power dissipation	+5V, -15V, I_{REF}=2.0mA		122	136	mW
		±15V, I_{REF}=2.0mA		156	174	

AC ELECTRICAL CHARACTERISTICS

SYMBOL	PARAMETER	TEST CONDITIONS	DAC08C			DAC08E DAC08			DAC08H DAC08A			UNIT
			Min	Typ	Max	Min	Typ	Max	Min	Typ	Max	
t_S	Settling time	To ± 1/2LSB, all bits switched on or off, T_A=25°C		70	135		70	135		70	135	ns
	Propagation delay											
t_{PLH}	Low-to-High	T_A=25°C, each bit.										ns
t_{PHL}	High-to-Low	All bits switched		35	60		35	60		35	60	

TEST CIRCUITS

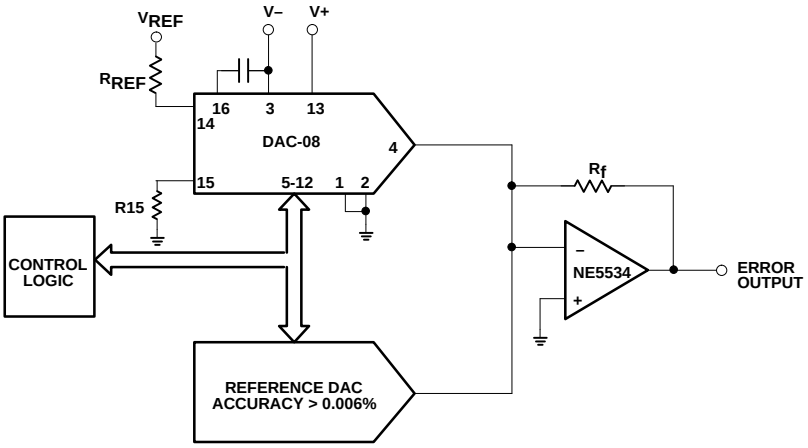


Figure 1. Relative Accuracy Test Circuit

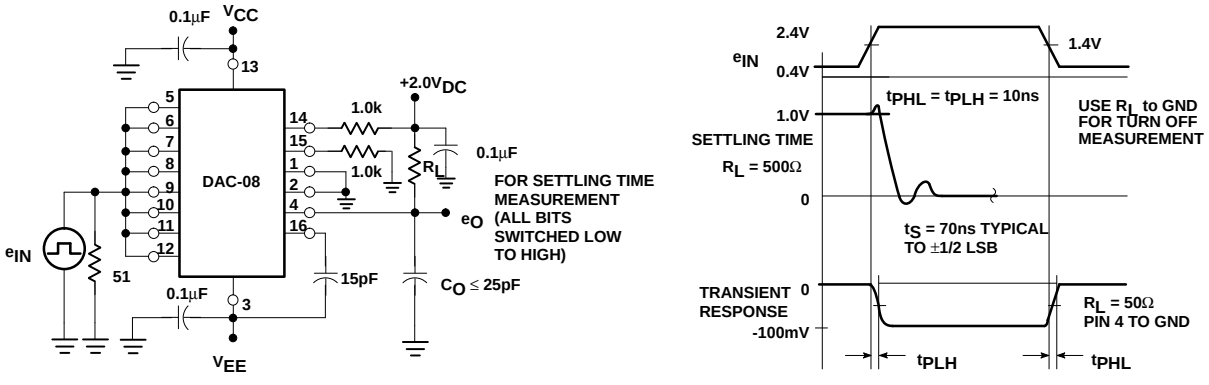


Figure 2. Transient Response and Settling Time

TEST CIRCUITS (Continued)

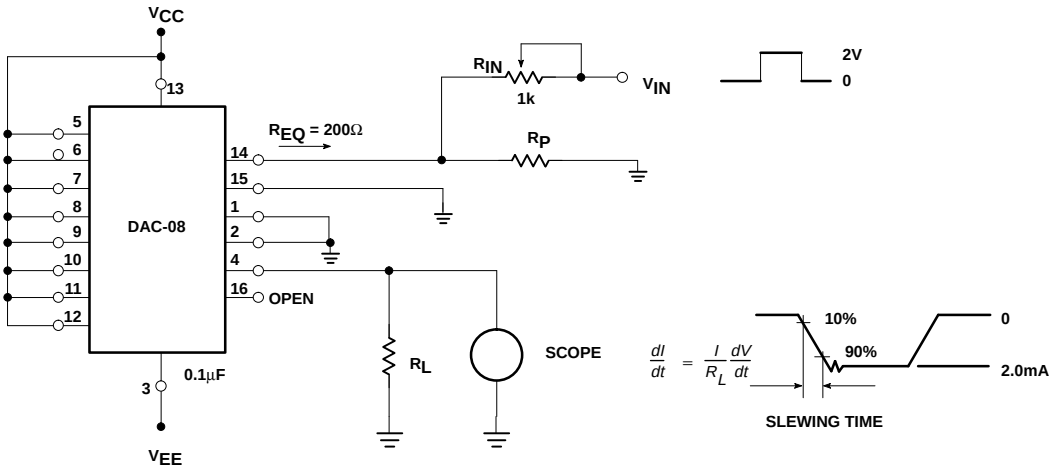
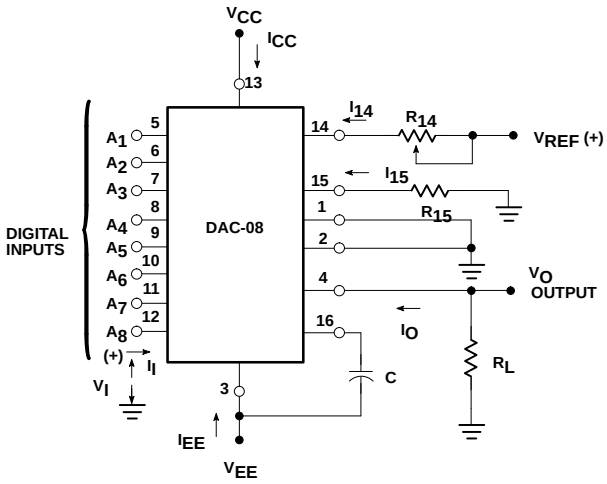


Figure 3. Reference Current Slew Rate Measurement



NOTES:
(See text for values of C.)
Typical values of $R_{14} = R_{15} = 1k$
$V_{REF} = +2.0V$
C = 15pF

V_I and I_I apply to inputs A_1 through A_8
The resistor tied to Pin 15 is to temperature compensate the bias current and may not be necessary for all applications.

$$I_O = K\left|\frac{A_1}{2} + \frac{A_2}{4} + \frac{A_3}{8} + \frac{A_4}{16} + \frac{A_5}{32} + \frac{A_6}{64} + \frac{A_7}{128} + \frac{A_8}{256}\right|$$

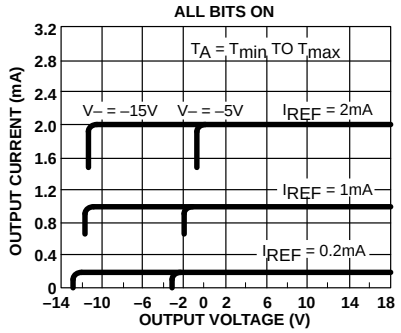
$$\text{where } K \approx \frac{V_{REF}}{R_{14}}$$

and A_N = '1' if A_N is at High Level
A_N = '0' if A_N is at Low Level

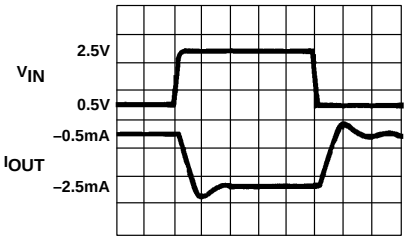
Figure 4. Notation Definitions

TYPICAL PERFORMANCE CHARACTERISTICS

Output Current vs Output Voltage (Output Voltage Compliance)

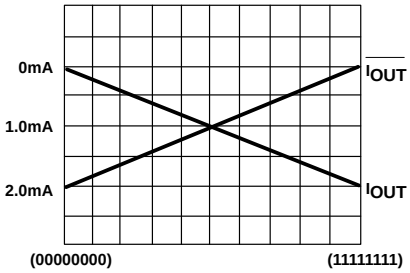


Fast Pulsed Reference Operation

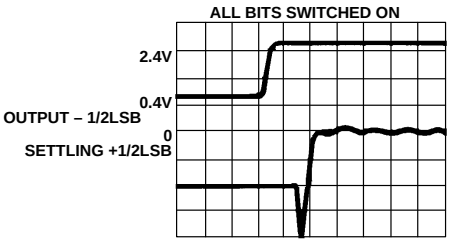


R_{EQ} = 200Ω, R_L = 100Ω, CC = 0

True and Complementary Output Operation

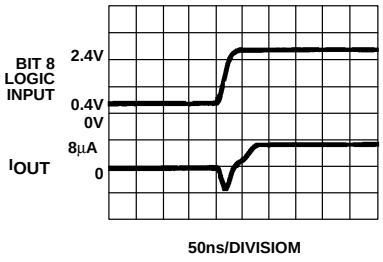


Full-Scale Settling Time

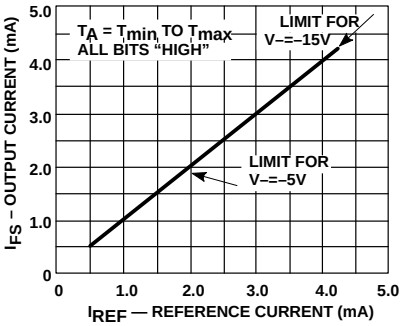


I_{FS}=2mA, R_L=1kΩ 1/2LSB=4μA

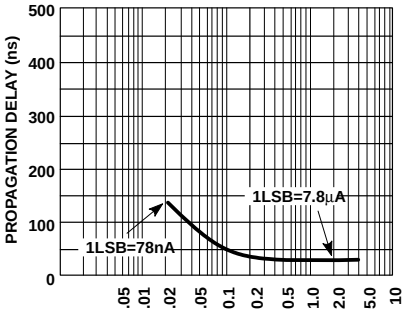
LSB Switching



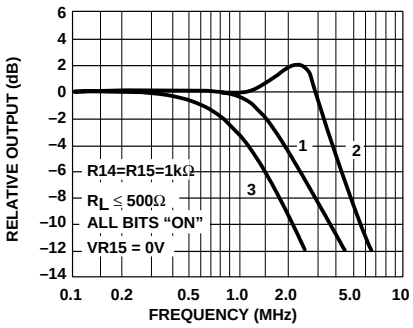
Full-Scale Current vs Reference Current



LSB Propagation Delay vs IFS



Reference Input Frequency Response

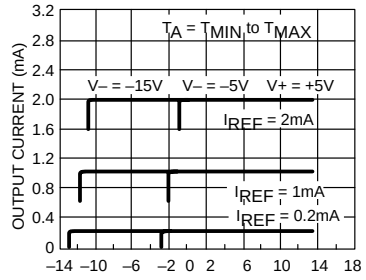


NOTES:

Curve 1: CC = 15pF, V_{IN} = 2.0V_{P-P} centered at +1.0V
Curve 1: CC = 15pF, V_{IN} = 5m0V_{P-P} centered at +200mV
Curve 1: CC = 15pF, V_{IN} = 100m0V_{P-P} centered at 0V and applied through 50Ω connected to Pin 14. +2.0V applied to R_{14}.

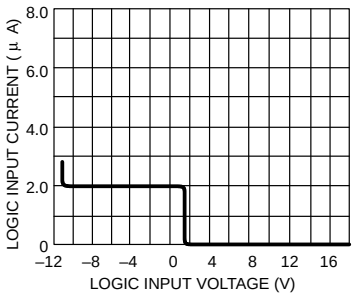
TYPICAL PERFORMANCE CHARACTERISTICS (Continued)

Reference AMP Common-Mode Range All Bits On

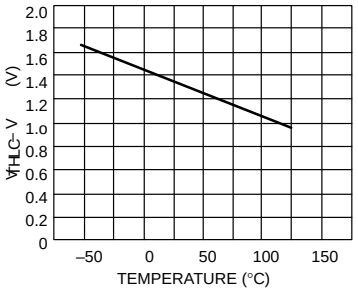


V_{15} — REFERENCE COMMON MODE VOLTAGE (V)
POSITIVE COMMON-MODE RANGE IS ALWAYS (V+) –1.5V.

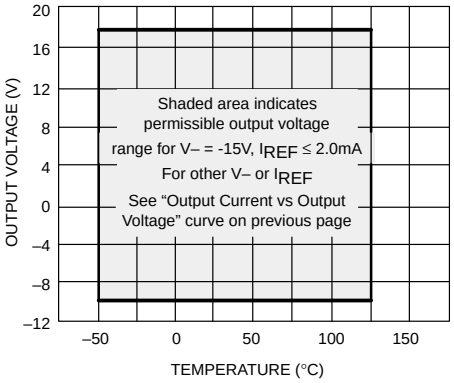
Logic Input Current vs Input Voltage



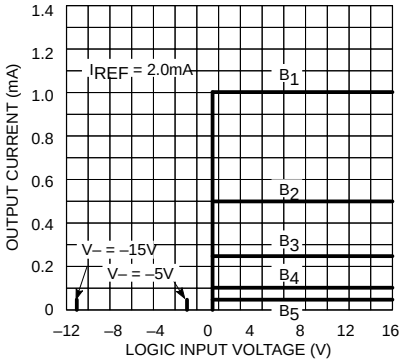
$V_{TH} - V_{LC}$ vs Temperature



Output Voltage Compliance vs Temperature



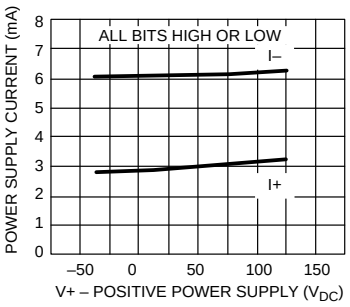
Bit Transfer Characteristics



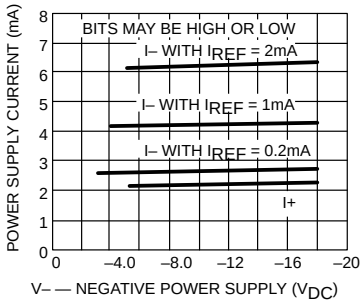
NOTES:

B_1 through B_8 have identical transfer characteristics. Bits are fully switched, with less than 1/2LSB error, at less than ±100mV from actual threshold. These switching points are guaranteed to lie between 0.8 and 2.0V over the operating temperature range (V_{LC} = 0.0V).

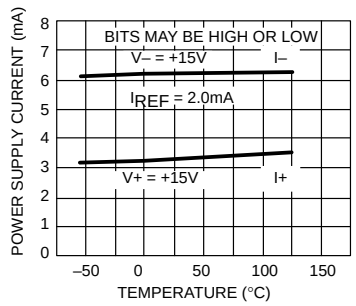
Power Supply Current vs V+



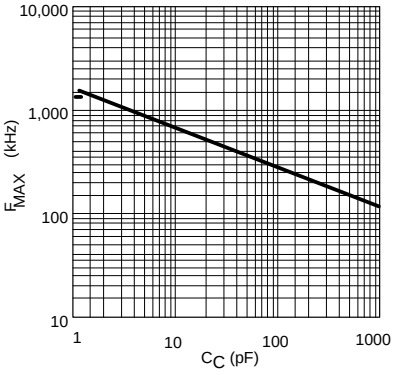
Power Supply Current vs V–



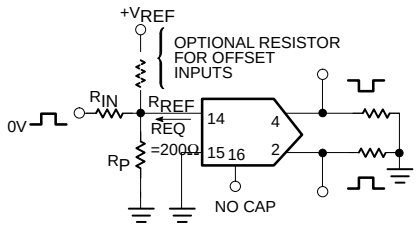
Power Supply Current vs Temperature



Maximum Reference Input Frequency vs Compensation Capacitor Value



TYPICAL APPLICATION



NOTES:
REQ = R_{IN} || R_P
Typical Values
R_{IN} = 5kΩ
$+V_{IN}$ = 10V

Pulsed Referenced Operation

FUNCTIONAL DESCRIPTION

Reference Amplifier Drive and Compensation

The reference amplifier input current must always flow into Pin 14 regardless of the setup method or reference supply voltage polarity.

Connections for a positive reference voltage are shown in Figure 1. The reference voltage source supplies the full reference current. For bipolar reference signals, as in the multiplying mode, R_{15} can be tied to a negative voltage corresponding to the minimum input level. R_{15} may be eliminated with only a small sacrifice in accuracy and temperature drift.

The compensation capacitor value must be increased as R_{14} value is increased. This is in order to maintain proper phase margin. For R_{14} values of 1.0, 2.5, and 5.0kΩ, minimum capacitor values are 15, 37, and 75pF, respectively. The capacitor may be tied to either V_{EE} or ground, but using V_{EE} increases negative supply rejection. (Fluctuations in the negative supply have more effect on accuracy than do any changes in the positive supply.)

A negative reference voltage may be used if R_{14} is grounded and the reference voltage is applied to R_{15} as shown. A high input impedance is the main advantage of this method. The negative reference voltage must be at least 3.0V above the V_{EE} supply. Bipolar input signals may be handled by connecting R_{14} to a positive reference voltage equal to the peak positive input level at Pin 15.

When using a DC reference voltage, capacitive bypass to ground is recommended. The 5.0V logic supply is not recommended as a reference voltage, but if a well regulated 5.0V supply which drives logic is to be used as the reference, R_{14} should be formed of two series resistors with the junction of the two resistors bypassed with 0.1μF to ground. For reference voltages greater than 5.0V, a clamp diode is recommended between Pin 14 and ground.

If Pin 14 is driven by a high impedance such as a transistor current source, none of the above compensation methods applies and the amplifier must be heavily compensated, decreasing the overall bandwidth.

Output Voltage Range

The voltage at Pin 4 must always be at least 4.5V more positive than the voltage of the negative supply (Pin 3) when the reference current is 2mA or less, and at least 8V more positive than the negative supply when the reference current is between 2mA and 4mA. This is necessary to avoid saturation of the output transistors, which would cause serious accuracy degradation.

Output Current Range

Any time the full-scale current exceeds 2mA, the negative supply must be at least 8V more negative than the output voltage. This is due to the increased internal voltage drops between the negative supply and the outputs with higher reference currents.

Accuracy

Absolute accuracy is the measure of each output current level with respect to its intended value, and is dependent upon relative accuracy, full-scale accuracy and full-scale current drift. Relative accuracy is the measure of each output current level as a fraction of the full-scale current after zero-scale current has been nulled out. The relative accuracy of the DAC08 series is essentially constant over the operating temperature range due to the excellent temperature tracking of the monolithic resistor ladder. The reference current may drift with temperature, causing a change in the absolute accuracy of output current. However, the DAC08 series has a very low full-scale current drift over the operating temperature range.

The DAC08 series is guaranteed accurate to within ± LSB at +25°C at a full-scale output current of 1.992mA. The relative accuracy test circuit is shown in Figure 1. The 12-bit converter is calibrated to a full-scale output current of 1.99219mA, then the DAC08 full-scale current is trimmed to the same value with R_{14} so that a zero value appears at the error amplifier output. The counter is activated and the error band may be displayed on the oscilloscope, detected by comparators, or stored in a peak detector.

Two 8-bit D-to-A converters may not be used to construct a 16-bit accurate D-to-A converter. 16-bit accuracy implies a total of ± part in 65,536, or ±0.00076%, which is much more accurate than the ±0.19% specification of the DAC08 series.

Monotonicity

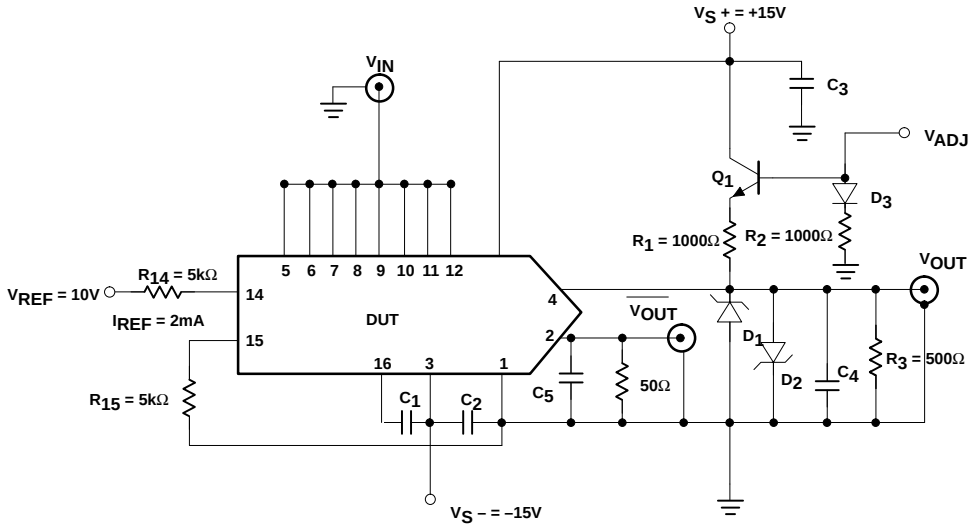
A monotonic converter is one which always provides analog output greater than or equal to the preceding value for a corresponding increment in the digital input code. The DAC08 series is monotonic for all values of reference current above 0.5mA. The recommended range for operation is a DC reference current between 0.5mA and 4.0mA.

Settling Time

The worst-case switching condition occurs when all bits are switched on, which corresponds to a low-to-high transition for all input bits. This time is typically 70ns for settling to within LSB for 8-bit accuracy. This time applies when R_L<500Ω and C_O<25pF. The slowest single switch is the least significant bit, which typically turns on and settles in 65ns. In applications where the DAC functions in a positive-going ramp mode, the worst-case condition does not occur and settling times less than 70ns may be realized.

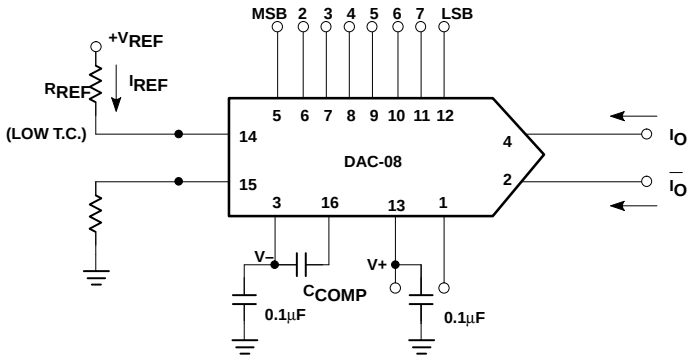
Extra care must be taken in board layout since this usually is the dominant factor in satisfactory test results when measuring settling time. Short leads, 100μF supply bypassing for low frequencies, minimum scope lead length, and avoidance of ground loops are all mandatory.

SETTLING TIME AND PROPAGATION DELAY



NOTES:
D_1, D_2 = IN6263 or equivalent
D_3 = IN914 or equivalent
C_1 = 0.01µF
C_2, C_3 = 0.1µF
Q_1 = 2N3904
C_4, C_5 = 15pF and includes all probe and fixturing capacitance.

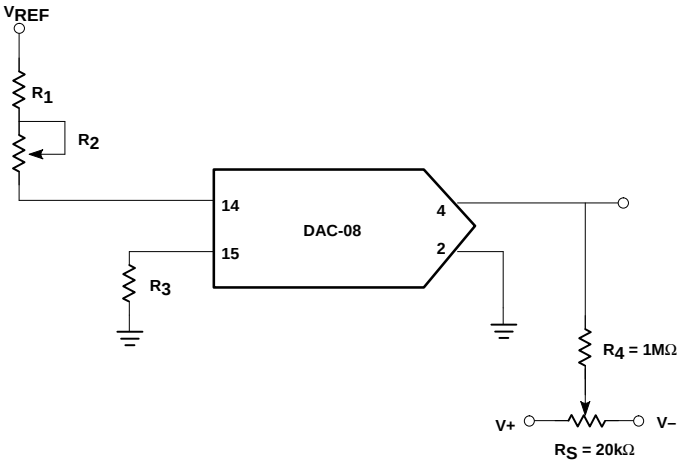
BASIC DAC08 CONFIGURATION



NOTES:

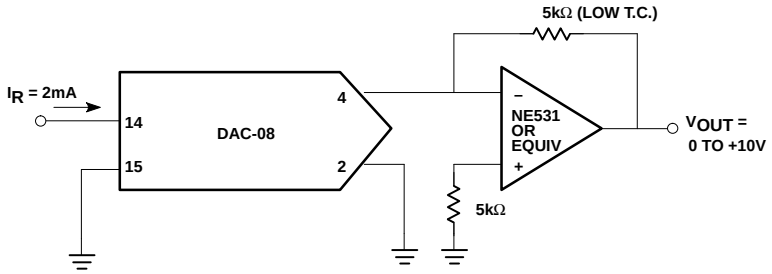
$$I_{FS} \approx \frac{+V_{REF}}{R_{REF}} \times \frac{255}{256} \; ; \; I_O + \overline{I}_O = I_{FS}$$ for all logic states

RECOMMENDED FULL-SCALE AND ZERO-SCALE ADJUST

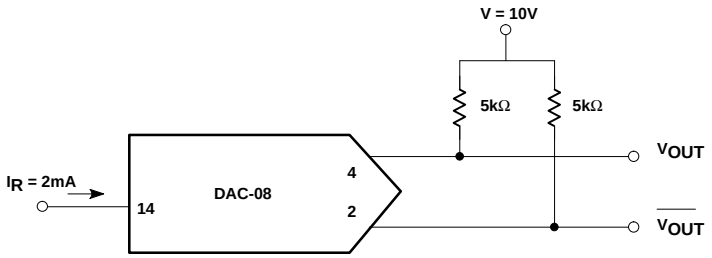


NOTES:
R_1 = low T.C.
$R_3 = R_1 + R_2$
$R_2 \approx 0.1\ R_1$ to minimize pot. contribution to full-scale drift

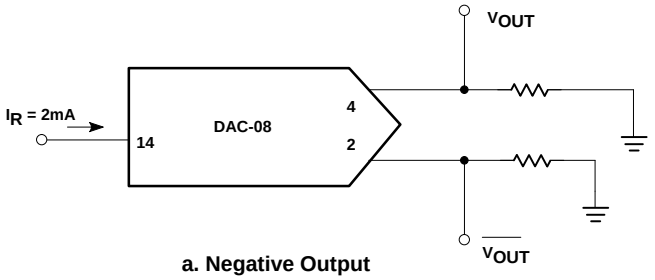
UNIPOLAR VOLTAGE OUTPUT FOR LOW IMPEDANCE OUTPUT



UNIPOLAR VOLTAGE OUTPUT FOR HIGH IMPEDANCE OUTPUT

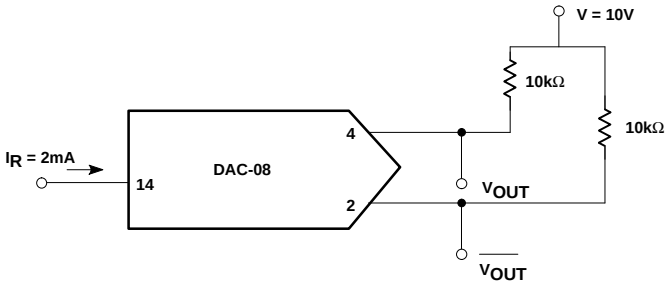


a. Positive Output



a. Negative Output

BASIC BIPOLAR OUTPUT OPERATION (OFFSET BINARY)



	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8	V_{OUT}	$\overline{V}_{\overline{OUT}}$
Positive full-scale	1	1	1	1	1	1	1	1	–9.920V	+10.000
Positive FS – 1LSB	1	1	1	1	1	1	1	0	–9.840V	+9.920
+ Zero-scale + 1LSB	1	0	0	0	0	0	0	1	–0.080V	+0.160
Zero-scale	1	0	0	0	0	0	0	0	0.000	+0.080
Zero-scale – 1LSB	0	1	1	1	1	1	1	1	0.080	0.000
Negative full scale – 1LSB	0	0	0	0	0	0	0	1	+9.920	–9.840
Negative full scale	0	0	0	0	0	0	0	0	+10.000	–9.920

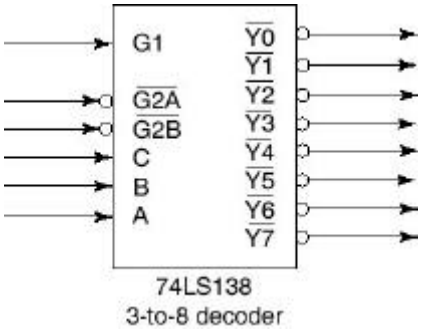
Interfacing I/O Devices

Connecting a peripheral device to a computer is called *interfacing* the device. This requires three major steps. The presentation of the steps here necessarily follows a certain order, but as you will see, the ordering of the steps for a particular device may need to be different. The first step is the design of a hardware connection scheme that will allow the programmer to address the device. Second, the programmer must write an interrupt service routine (ISR) to handle interrupts from the device and install its address in the exception vector table. Finally, the programmer must initialize, or program, the device to work in an appropriate way and enable the device. To perform these steps correctly, the person interfacing the device must have the appropriate documentation or *data sheet* for the device.

Hardware Interfacing

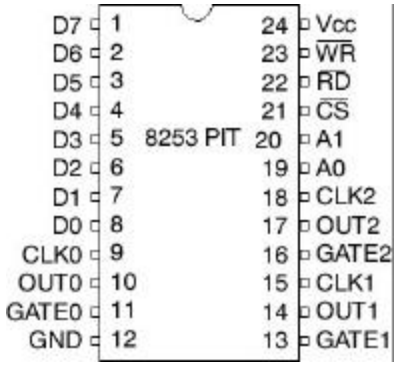
The hardware connection scheme for an I/O device includes appropriate connections to all the buses. A connection to the address bus is necessary to allow the programmer to address the ports. Assuming an I/O address space of 256 ports (the standard for older Intel processors which is still compatible with newer members of the Intel family), we need to be able to decode I/O addresses of just eight bits, since $2^8 = 256$ unique locations. The relevant address lines will therefore be A0-A7. To be able to send data to the device or receive data from it, a connection to the data bus is also essential. Some devices now have 16-bit ports, but many still have 8-bit ports. For simplicity, we will assume that the devices we are interfacing have 8-bit ports, so again, we only need to worry about the low order eight bits of the data bus: D0-D7. In addition, each device generally needs to have connections to some of the control bus lines. The task of hardware interfacing is to connect the device to these lines appropriately. The data sheet for a particular device will include the pin diagram for the device. This diagram gives the purpose of each pin and allows the interfacer to connect the bus lines to the correct pins.

Typically, it is desirable to have the ability to connect more than one device to the CPU. To do this, we can use a commercially available decoder chip, the 74LS138. This chip is a 3-to-8 decoder. It has three input lines which encode a 3-bit binary number. It also has eight output lines, logically numbered from zero to eight. Given a particular 3-bit pattern as input, the 74LS138 will activate the output line with the corresponding number. The input pin for the low order bit of the pattern is A, with pins B and C serving as the inputs for the next bits. For example, an input on pins A, B, and C of 001, respectively, will activate output line Y4#. This chip allows interfacing up to eight different devices from a single address bus connection. In addition the decoder has three "gating signals" that allow a particular set of addresses to activate the decoder. The output lines connect to various devices, while the input lines connect to the address and control bus.



As you know, devices (or their controllers) contain various registers or ports that allow for communication between the CPU and the device. The ports have a specific ordering, just as memory locations do. To be able to read from or write to these ports, the designer must ensure that the device is addressable. Each device must have a unique base address. This address actually provides access to the first port on the device. Subsequent ports are accessible through subsequent addresses. For example, assume that a particular device, such as the ACIA described in Peripheral Devices and I/O, has four 8-bit ports. Suppose that the base address of the device is 0040h. The four ports on the device are then at 0040h, 0041h, 0042h, and 0043h. The ordering of the ports (that is, which one is first, second, third, etc.) is part of the design of the device. The data sheet for the device will give the relative port addresses (i.e., the offset for each port on the chip).

We will work an example showing how to interface the Intel 8253 Programmable Interval Timer (PIT) to an Intel 8088 chip. The timer is a device that we could use to count off the length of a time slice for the kernel of a multitasking operating system. The timer has other uses as well, but we will assume this particular application. The 8253 PIT includes three separate counters, each of which is independently programmable. These are both input and output ports, that is the CPU can both read from and write to these ports. The counters divide the input frequency by a given number and counts down. When a counter reaches 0, it generates a pulse on the corresponding output. If the GATE input (see below) for the counter is high, that is, the counter is still enabled, it automatically restarts the count. The timer has a total of four ports, one for each counter (Counter 0, Counter 1, and Counter 2) and a write-only control register. It also has some internal registers that are not accessible to the programmer. The value in the control register determines the mode of operation of the timer. The following figure shows the pin diagram for the 8253.



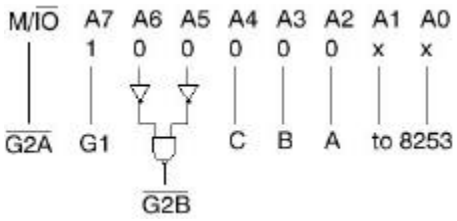
The general technique is to start with a known base address. For example, suppose you want to connect the CS# pin of the 8253 timer to the Y0# line of the 74LS138 decoder so that the timer

has a base address of 80h. That is, we want the CS# pin of the timer to be active (low voltage) *only* when the given address is 80h, 81h, 82h, or 83h. It should *not* be active for any other address. The two low order bits of the address will give the addresses for the four ports on the controller. These lines will connect directly to the timer, rather than to the decoder. This means that we will route address bus lines A0 and A1 to the devices connected to the decoder. We can use the next three higher order address lines (A2, A3, and A4) as the inputs to pins A, B, and C of the decoder. Since the base address has all zeros in these bits (80h = 10000000_2), connecting these address lines directly to the input pins of the 74LS138 will have exactly the right effect. We now have to account for address lines A5-A7. We have three gating signals on the decoder that must be active if the decoder itself is to be active.

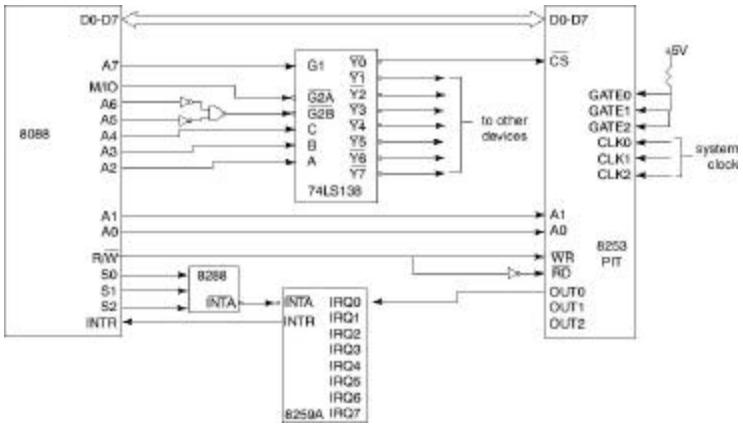
Since the 74LS138 is to serve as a connection strictly to I/O devices, one of the gating pins must connect to the M/IO# control bus signal of the CPU, so that it is active only when an I/O instruction (IN or OUT) is executing. Since this signal selects a chip when the line is low, it will be simplest if we connect this line to one of the negative logic gating pins. We can arbitrarily choose G2A#. In this way, if M/IO# is low (logic 0), G2A# will also be low and, provided the other gating signals are also active, the decoder will be selected. Still remaining are pins G2B# and G1. We will use address lines A5-A7 to activate these signals. When the address is 80h, the values for these lines will be A7 = 1, A6 = 0, and A5 = 0. Of the two gating signals, G1 is active high, so again, it makes sense to connect A7 directly to this pin. The last step is to combine lines A6 and A5 so that only when both are zero, they will send a logic 0 to pin G2B#. To do this, we invert both signals and connect them to a two-input NAND gate. Notice that the double inversion (we first invert A5 and A6 and then get a second inversion from the NAND gate) is necessary. Consider that we try to simplify this connection using no inversion at all. This means we would connect lines A5 and A6 to a two-input AND gate. In this situation, we would get "false lows" for certain addresses that would incorrectly activate the decoder and timer. Examine the following truth table to see why (the starred entries show an incorrect result):

A5	A6	A5&A6	not A5	not A6	not (not A5 & not A6)
0	0	0 (G2B# active)	1	1	0 (G2B# active)
0	1	0 (G2B# active)*	1	0	1 (G2B# inactive)
1	0	0 (G2B# active)*	0	1	1 (G2B# inactive)
1	1	1 (G2B# inactive)	0	0	1 (G2B# inactive)

The following diagram shows the complete connection for the address lines (the x's represent "don't cares," that is, bits for which the value is unimportant). You should try some other 8-bit binary addresses with this connection scheme to convince yourself that no other address will ever activate the Y0# line of the 74LS138 and thus the timer.



Before the hardware connection is complete, we must also connect the other pins of the timer. The data sheet of the 8253 gives the purpose of these pins. Simplest are the data bus pins: D0-D7. These obviously must connect to the corresponding lines of the data bus. The Vcc pin is for power, so it must connect to the power supply of the computer. The GND pin is for electrical ground. This leaves the RD#, WR#, CLK*n*, GATE*n* and OUT*n* pins. RD# and WR# are for enabling reading from the timer's ports and writing to them, respectively. The CPU has a single pin, R/W#, so to get the proper behavior, we must "split" this signal. We will invert the R/W# signal to feed to the RD# pin of the 8253 and connect the R/W# signal directly (not inverted) to the WR# pin. The CLK*n* pins receive the input clock pulse, so we can connect these to the system clock to provide input for the timer. The OUT*n* pins provide the output signal from the timer. Since we will use the timer to generate timer interrupts for the operating system, we will connect these pins to the interrupt controller. Finally, the GATE*n* pins serve to individually enable or disable the three counters on the 8253 chip. Since we want these to be permanently enabled, we can connect them to a pullup resistor to keep them always at +5 volts. The figure below shows the completed connection diagram.



The Interrupt Service Routine

The exact nature of the interrupt service routine and the tasks that it performs depend on the the device itself and the job that it must do. You have already seen an introduction to interrupt handlers for peripheral devices and you know how to install their starting addresses in the exception vector table. You also know that exception and interrupt handlers must end with the IRET instruction. Beyond those general considerations, a few more points deserve special mention. It is important in many instances to disable interrupts before installing a handler's address in the Exception Vector, because it is essential that the full address (both segment and offset) are both correct before the exception of interrupt occurs. If the service routine is an exception handler, this may not be a critical concern, although it can be. For example, consider what would happen in a multitasking operating system if the program installing the handler

reaches the end of its time slice after writing the segment but before writing the offset to the Exception Vector. If the next process to gain the use of the CPU generates the software exception with the inconsistent address in the vector table, the handler will not execute. In addition, if a system includes pipelining, interrupts can occur between any two stages of the fetch execute cycle. This situation increases the chance that an interrupt could occur when the Exception Vector is in an inconsistent state. For these reasons, disabling interrupts with the CLI instruction before altering the Exception Vector is always a good idea. If the service routine is for a hardware device, disabling interrupts before beginning to write to the Exception Vector is absolutely necessary. If the device for which you are installing a handler's address generates an interrupt before the installation is complete, the results will be unpredictable at best and perhaps even catastrophic at worst. Of course, equally important is remembering to reenable interrupts with the STI instruction as soon as the installation of the address is finished.

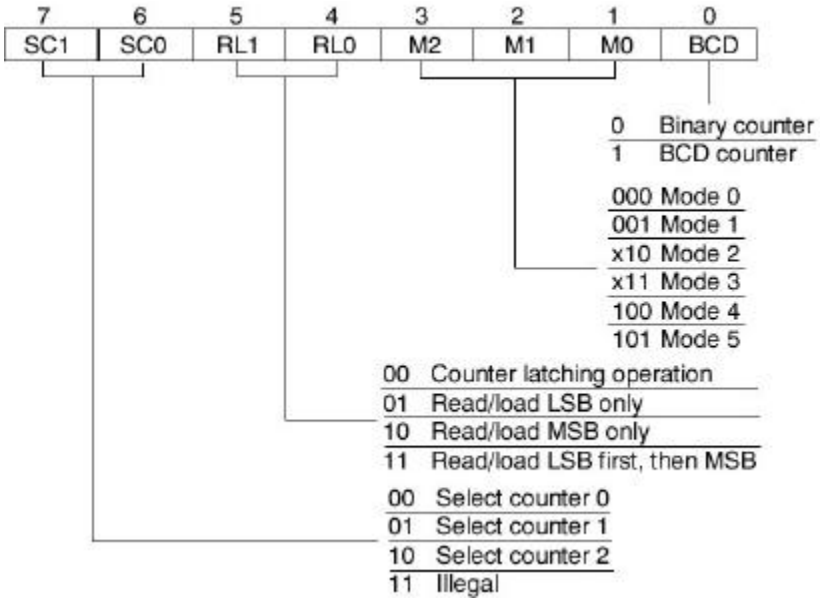
Because many I/O devices perform multiple functions and therefore require different kinds of services from the CPU depending on the reason for the interrupt, another common scenario with interrupt service routines is for the handler to be a *dispatch* routine. This means that the main task of the handler is to determine what the device needs and then to call a subroutine to perform the service. Sometimes, the most efficient implementation for this is a jump table. Other times, the interrupt service routine will need to check several conditions and call an appropriate subroutine based on the results of those checks. Although the 8253 timer is not a good example of this type of device, since it really only performs one function, the ACIA discussed in Peripheral Devices and I/O is. Typically, when a device can generate an interrupt for more than one reason, it includes a status register. This is one of the ports on the device. The service routine begins by reading this status register. Depending on the value of bits in this register, the handler calls the proper subroutine to service the particular request. The meaning of the bits of the status register of a device is available in the data sheet for the device.

The handler for our timer example (using the timer to generate an interrupt at the end of each time slice) would need to perform a context switch between the currently running process and the next process in line waiting for the CPU. Although other tasks would also be necessary, the major job of our interrupt service routine would be to save the values of *all* the system registers and then to write the previously saved values into the registers for the next process in line.

Finally, it is important to ensure that the device is disabled until the installation of the service routine is complete. Many devices include an enable/disable bit in the control register, that allows a programmer to disable a device through a software instruction. This is particularly desirable when changing the service routine for a device that is already installed. If the device does not include this capability, then the service routine installation must be complete before connecting the device. This is the case with the 8253 timer.

Programming Devices

The data sheet for each peripheral device will include its programming requirements. Programming a device generally involves writing one or more values to its control register, but may also require writing values to other ports on the device as well. The 8253 timer is fairly representative and has the advantage of also being quite simple. The first consideration for programming virtually any device is to determine the operation required and the meaning of the bits in the control register. The following figure shows the control register for the 8253 PIT.



Mode 0:	Interrupt on terminal count
Mode 1:	Programmable one-shot ("alarm" type pulse)
Mode 2:	Rate generator (used to generate a regular wave signal of a given period)
Mode 3:	Square wave rate generator (used to generate a regular square wave signal of a given period)
Mode 4:	Software triggered strobe
Mode 5:	Hardware triggered strobe

The above figure gives us most of the information we need to write the correct control word to the timer's control register to make it operate in the way we want. A few additional words of explanation are also necessary, however. Some of the operation modes given in the table above are fairly self-explanatory, but others are not so obvious. The various modes actually determine the shape of the OUT signals for a particular timer. The 8253 PIT can generate various types of square waves by holding the OUT line alternately high and low for given amounts of time. Such output signals can be useful for determining timing for various kinds of applications, such as delays. The data sheet for the timer gives timing diagrams for each of the modes. For our application, we will only need to use Mode 0, since we want the timer to generate an interrupt when it reaches zero.

The low order bit of the control word gives us the ability to specify the initial count value (the frequency divisor) as a 16-bit binary number (0001h-10000h) or as a BCD (binary coded decimal) number in the range of 1-10000. The data sheet also specifies that a value of all zeros written to the counter signifies the largest number in the range of possible count values. This explains how it is possible to represent 10000h with only 16 bits. Binary coded decimal representation uses 4-bit groups to encode each digit of a decimal number. For example, the binary coded decimal representation of 937 is 1001 0011 0111. To specify a binary coded decimal number in a program, you should be able to see that a hex representation is appropriate. That is, if you wanted to use 937 BCD as an immediate operand in an assembly language instruction, you would use 937h. In some programs such as financial applications, using BCD representation is more natural than other representations.

The RL bits allow control over the reading and writing of the counter value for a particular counter. The "counter latching operation" code transfers the current count to an internal storage buffer so that the count is stable for reading (otherwise, the continued countdown could change the value of the counter during the read). The other modes specify which part of the initial count value is to be written or read. It is important to realize that the initial count value is a 16-bit number, whereas the counter registers are only eight bits wide. This means that the programmer must read or write the counter register in two instructions. The usual method is to use RL mode 11. In this case, the programmer reads or writes the least significant byte first. If the operation is a write, the internal logic of the timer transfers this byte to an internal register, so that the next write does not destroy its value. After reading or writing the least significant byte, the programmer uses a second instruction to read or write the most significant byte. Finally, the SC bits allow the programmer to specify *which* counter he or she is reading or writing.

With this information, we are ready to program the timer to function in a manner that will be appropriate for our example application. As we mentioned above, we will want to use mode 0, so we already know that the control byte bits 1, 2, and 3 must be cleared. We will use a binary number as the initial counter value, so bit 0 must also be 0. We will use a two-byte counter, so this determines that the RL bits must be 11. Finally, we arbitrarily decide to use counter 1, so this gives us 01 as the value for bits 6 and 7. Putting these bits together gives us 01110000b (70h) as the value we must write to the control register.

The next step is to determine the initial count value. If we assume that we are interfacing with a clock speed of 8 KHz, then that gives us the frequency on the CLK*n* inputs. Suppose we decide that we want a time slice to be 500 milliseconds (this is quite long for a time slice, but consider that we are working with a slow processor!). Recall that one hertz is one cycle per second, so a period of 500 milliseconds gives a frequency of 0.5 hertz. Of course 8KHz is 8000 hertz. The problem is to determine the appropriate divisor so that the OUT signal will go high to generate an interrupt every 500 milliseconds: 8000/0.5 = 16000 = 3E80h.

We now have the values necessary to program the timer. The only remaining step is to determine the addresses for each of the ports on the timer. We know that the base address is 80h, so we really only need the relative ordering of the ports. The data sheet will give us this information:

A1	A0	Port
0	0	Counter 0
0	1	Counter 1
1	0	Counter 2
1	1	Control register

The following code shows what we need to do:

```
Init8253  proc
          push  ax
          mov   al,70h  ;select mode 0, counter 1, binary rep.
          out   80h,al  ;write to control register
          mov   al,80h  ;set up LSB of count divisor
          out   81h,al  ;write it to counter 1
          mov   al,3eh  ;set up MSB of count divisor
          out   81h,al  ;write it to counter 1
          pop   ax
          ret
Init8253  endp
```

Of course, the programming requirements for different devices and for different processors will vary. For example, some devices have a vector number register that allows the programmer to specify a vector number for the device. When the device generates an interrupt, it sends this number to the CPU during the exception processing state. In general, programming a device always requires writing a specific value to its control register(s).

References

Intel. *Microprocessor and Peripheral Handbook*. Intel Corporation, 1983.

Mazidi, A.M. and J.G. Mazidi. *The 80x86 IBM PC and Compatible Computers, Volumes I and II: Assembly Language, Design and Interfacing*. Englewood Cliffs, NJ: Prentice Hall, 1995.

Orejel, Jorge. Personal Communication.

Acknowledgement

Thanks go to Jorge Orejel for checking the correctness of my hardware interface design.

Introduction

An important requirement in many systems is the ability to off-load numeric data processing. In an 80C286 system, this can be accomplished with an 80287 numeric co-processor. However, as processor speeds increase, it may become necessary to interface a high speed 80C286 processor with a lower speed 80287. This Document will briefly describe the interface between a 16MHz 80C286 (80C286-16) and a 10MHz 80287 (80287-10).

Interfacing the 80C286 with an 80287 can be broken down into three main areas:

(1) Bus control lines and data lines which coordinate and implement the flow of data between the two processors (i.e. the data lines, chip select lines, and read/write lines).

(2) The clock line(s), which drive the two processors.

(3) The four status lines through which the 80C286 and 80287 directly communicate status information to one another - comprised of the $\overline{BUSY}$, $\overline{ERROR}$, Peripheral Request (PEREQ), and Peripheral Acknowledge ($\overline{PEACK}$) lines.

Bus Control Lines

The various bus control and data lines in most systems would be coordinated by either a bus controller (such as the 82C288), or a bus controller subsection of an 80C286 oriented chip set. All requisite bus control timing between a 16MHz 80C286, and a 10MHz 80287 would then be handled by these devices (typically with one wait-state inserted to allow for the slower 80287-10).

Clock Lines

A system using a 16MHz 80C286 with a 10MHz 80287 requires separate clock lines for the two processors. The 32MHz system clock used by the 80C286-16 is too fast for the 80287 ±10, necessitating a dedicated clock driver for the 80287. This clock driver should supply a 10MHz clock to the 80287 with a 1/3 duty cycle to allow the 80287-10 to run at it's full 10MHz capability. One solution for providing this clock is the 82C84A-1, which meets this specification with either a 30MHz crystal at it's crystal inputs, or a 30MHz external frequency input to it's EFI pin. In either case, a 10MHz 1/3 duty cycle clock is output to the 80287. Note that when using a dedicated clock driver such as this, the CKM pin of the 80287 must be pulled up.

Status Lines

The 80C286 and 80287 communicate status information with one another through four signals; the $\overline{BUSY}$ line, the $\overline{ERROR}$ line, the peripheral request line (PEREQ), and the $\overline{PEACK}$ line.

The $\overline{BUSY}$ and $\overline{ERROR}$ lines can be connected from the 80287 to a 80C286-oriented chipset, or from the 80287 directly to a 80C286. In the case of the chipset interface, the signal timing between the 80287 and 80C286 is coordinated by the chipset. In the case of the direct 80287 to 80C286 interface, the signal timing is handled by the 80C286, and, since the signal flow direction is from the 80287 to the 80C286 (i.e. from the slower device to the faster device), no additional hardware is required to achieve proper timing.

The peripheral request (PEREQ) line should be connected directly from the 80287 to the 80C286, and again, since the signal flow direction is from the 80287 to the 80C286, no additional hardware is required.

The peripheral acknowledge ($\overline{PEACK}$) line is normally connected directly from the 80C286 to the 80287. In this case the signal flow direction is from the 80C286 to the 80287 (i.e. faster device to slower device), and the $\overline{PEACK}$ active time is not guaranteed to meet the requirements of the slower 80287-10. Worst case timing for the 80C286-16 reveals that $\overline{PEACK}$ output could be as short as 45.5ns (i.e. $\overline{PEACK}$ (min) = 45.5ns). The 80287-10 input requirement is $\overline{PEACK}$ (min) = 60ns. (min) = 45.5ns). The 80287-10 input require-

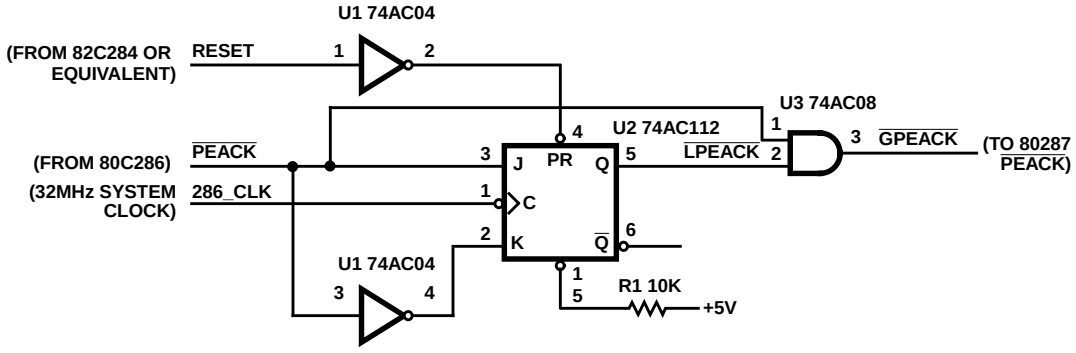


FIGURE 1. PEACK STRETCH CIRCUIT

The proper $\overline{\text{PEACK}}$ timing can be achieved using the circuit shown in Figure 1 comprised of a 74AC04, 74AC08, and a 74AC112. Referring to the timing diagram shown in Figure 2, it can be seen that this circuit effectively "stretches" the 80C286's $\overline{\text{PEACK}}$ output (in the form of $\overline{\text{GPEACK}}$) to 72.7ns, which satisfies the 80287-10 requirement.

The operation of the circuit shown in Figure 1 is as follows:

(1) The RESET signal (which is also applied to the 80C286) is used to initialize the 'AC112 to a known inactive state (Q = 1).

(2) When the 80C286 asserts the $\overline{\text{PEACK}}$ signal, the gated version of this signal ($\overline{\text{GPEACK}}$) is asserted with minimal delay (7.9ns through the 'AC08).

(3) On the falling edge of the 80C286 CLK at the beginning of Phase 2 of the T_S cycle, the low state of $\overline{\text{PEACK}}$ is clocked into the 'AC112. This effectively holds $\overline{\text{GPEACK}}$ low for an additional clock cycle longer than standard $\overline{\text{PEACK}}$ timing.

(4) On the falling edge of the 80C286 CLK at the beginning of phase 2 of the first T_C cycle, the high state of $\overline{\text{PEACK}}$ is clocked into the 'AC112, which then causes $\overline{\text{GPEACK}}$ to go inactive.

The net effect of this circuit operation is to extend the 80C286's Peripheral Acknowledge signal to the 80287-10 sufficiently to meet it's requirements.

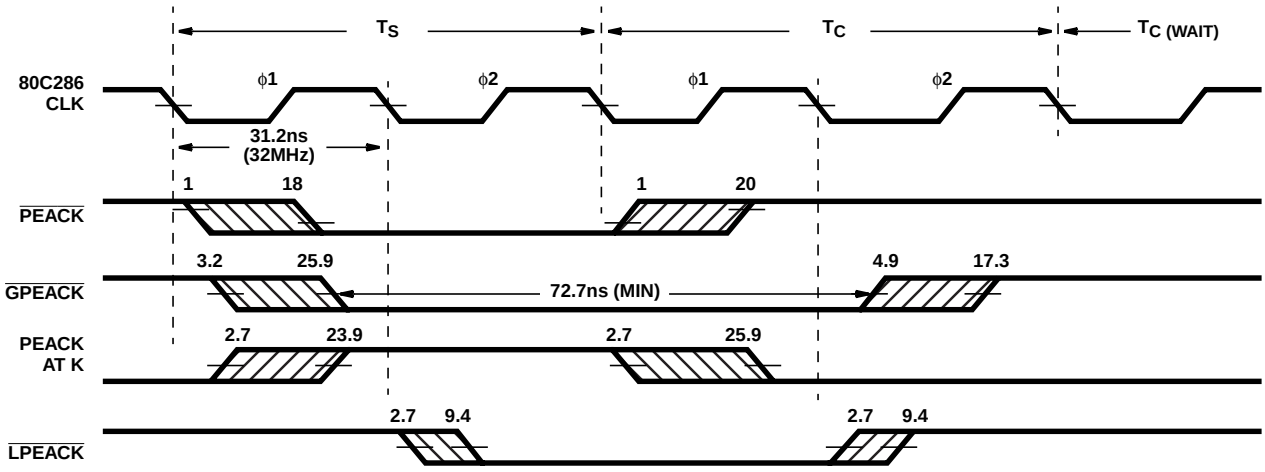


FIGURE 2. PEACK CYCLE TIMING

Sales Office Headquarters

NORTH AMERICA
Intersil Corporation
P. O. Box 883, Mail Stop 53-204
Melbourne, FL 32902
TEL: (407) 724-7000
FAX: (407) 724-7240

EUROPE
Intersil SA
Mercure Center
100, Rue de la Fusee
1130 Brussels, Belgium
TEL: (32) 2.724.2111
FAX: (32) 2.724.22.05

ASIA
Intersil (Taiwan) Ltd.
7F-6, No. 101 Fu Hsing North Road
Taipei, Taiwan
Republic of China
TEL: (886) 2 2716 9310
FAX: (886) 2 2715 3029

Peripheral Devices and I/O

Peripheral devices are the means of communication between the CPU and the outside world. As human beings, we are accustomed to the interface between ourselves and the devices. For instance, we know that to operate a keyboard, we press the keys; to operate a mouse, we place a hand over it and move it or press its buttons. These devices must also communicate with the CPU. The information we enter via an input device (such as a keyboard or mouse) must somehow get past the device and into the registers on the CPU or into memory where the CPU can gain access to it. Similarly, the information that the CPU has must make its way from the CPU to output devices such as a printer or the monitor. For this to happen, the CPU must be able to detect when an input device has information for it. An output device must be able to signal to the CPU when it is ready to process more information so that the CPU can send more information to it. In other words, these devices need the CPU to perform *services* for them.

A primitive, but simple means for the CPU to determine when a device needs a service is to *poll* the device. In essence, the CPU must periodically check the status of each device to see if it needs attention. Typically, the CPU checks these devices in a particular order. For example, the CPU may check a timer to see if it is time to switch programs in a multitasking system. If the timer does not need service, it may check the monitor next, then the keyboard, then the mouse, etc. This check is time consuming and often useless. Many times, there is no device that needs service. The CPU has taken time away from program execution to check on devices that are idle. Furthermore, it may be that a device does need service, but if it is one of the last in the ordering, the CPU wastes time checking on all the devices that are "ahead" of it before it gets to the one that needs attention.

Hardware Interrupts

A more efficient means of servicing the needs of peripheral devices is by having a device itself signal to the CPU that it needs service. This allows the CPU to continue executing instructions until a device needs service. In other words, the device *interrupts* the fetch/execute cycle when it needs the attention of the CPU. This is the idea behind *hardware interrupts*. Hardware interrupts are exceptions caused by hardware external to the CPU, such as peripheral devices or a reset switch. Hardware interrupts occur because the external device needs the attention of the CPU in order to carry out some task. For example, imagine that a communications program requests information from a modem. The modem receives one bit at a time on an incoming telephone line, since it is a serial device. These bits are stored in a buffer (an 8-bit register) until the modem receives a complete character, rather than sending the bits to the CPU one bit at a time. Since the data bus can carry 16 bits at once, it would be wasteful to make eight trips, using only one bit of the data bus for each trip. When the buffer is full, then, the CPU must assist in transferring the buffer contents to memory. That is, once the buffer fills up, the modem's controller must generate an interrupt to alert the CPU and let it know that it needs a service (in this case, emptying the data buffer).

Interrupts are *asynchronous*. That is, they occur at any arbitrary time (whenever a device actually needs the intervention of the CPU); they are unpredictable. Rather than have the CPU continually check each device to determine whether the device needs the CPU to perform some service, the CPU can continue its work until a device calls for its attention.

Although a device can assert an interrupt at any time, the CPU usually only services (or recognizes) interrupts between instructions. In other words, it does not stop in the middle of the fetch/execute cycle to service an interrupt. This means there may be a delay between the time a device generates an interrupt and the time the CPU processes it. We can expand the normal fetch/execute cycle as follows:

Repeat

fetch instruction
decode instruction
fetch any memory operand necessary
execute instruction operation
write result to memory if necessary
check for hardware interrupt

Until Halt

Prioritizing Interrupts

The order in which the CPU checks devices with the polling scheme imposes a *priority* scheme on the devices. Devices that the CPU checks first have a higher priority (they are more likely to receive service). Imagine that two or more devices require service at the same time. The CPU will service the device that it checks first before it services a device that it checks later. There is also a priority scheme for interrupts, that is, the CPU considers some interrupt requests to be more important than others.

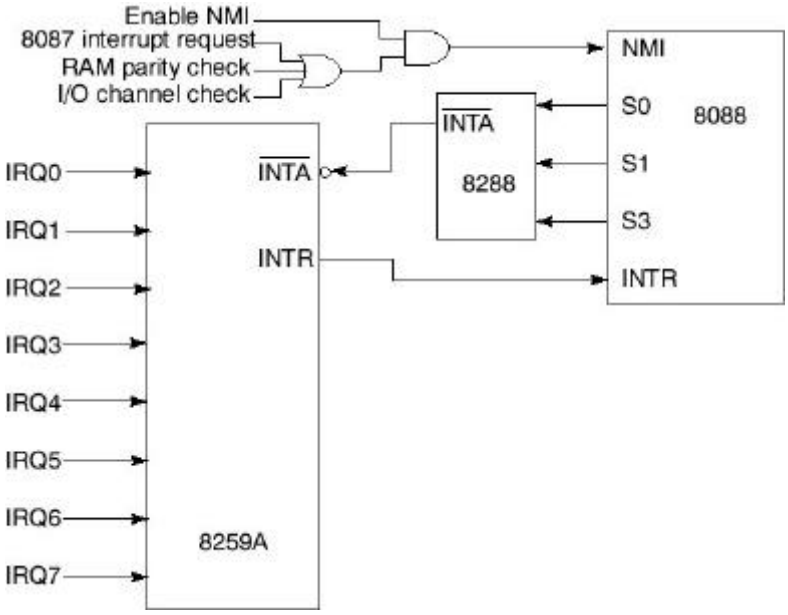
Intel Interrupt Prioritization

The implementation of this prioritization is as follows. First, it is important to understand that peripheral devices do not connect directly to the CPU. Instead, an *interrupt controller* (the 8259A chip) stands between the devices and the processor. This controller is a hardware device that receives signals from up to eight different external devices. The chip has eight input lines, each of which can connect to a peripheral device. The names of these input lines, IRQ0-IRQ7 (Interrupt ReQuest), correspond to the relative priorities of the devices attached to them, with the device attached to line IRQ0 having the highest priority. The table below shows the standard IRQ assignments. A device that requires the attention of the processor sends a signal on its line to the interrupt controller. At any given moment, all, some, or none of the devices may require the services of the CPU. If more than one device is sending a signal at the same time, the controller selects the line with the highest priority (lowest numbered) of all the lines currently active. Only the device connected to this line will actually be able to get the attention of the CPU. The 8259A encodes the vector number corresponding to the selected signal and stores this value in an internal register of the controller itself.

Priority	**IRQ**	**Device**
highest	0	System timer
	1	Keyboard

	2	Available/Secondary Interrupt Controller
	3	Serial Communications Port (COM2)
	4	Serial Communications Port (COM1)
	5	Parallel Communications Port (LPT1)
	6	Standard Floppy Disk Controller
lowest	7	Parallel Communications Port (LPT2)

The combination of the separate, prioritized, interrupt request lines and the operation of the 8259A controller chip allows the CPU to ignore at least temporarily some relatively unimportant interrupts so that it can process more important ones instead. In addition, a programmer can ensure that the CPU ignores even the interrupt requests that the 8259A sends. The FLAGS register includes an "interrupt enable" bit. If this bit is zero, the CPU will ignore any incoming device interrupts and we say that interrupts are disabled. If the bit is one, interrupts are enabled and the CPU will process the interrupt. The CLI (clear interrupt flag) and STI (set interrupt flag) instructions give programmers control over when the CPU should ignore interrupts.



8259A Interrupt Controller ("IBM BIOS Technical Reference", 1984, IBM Corp.)

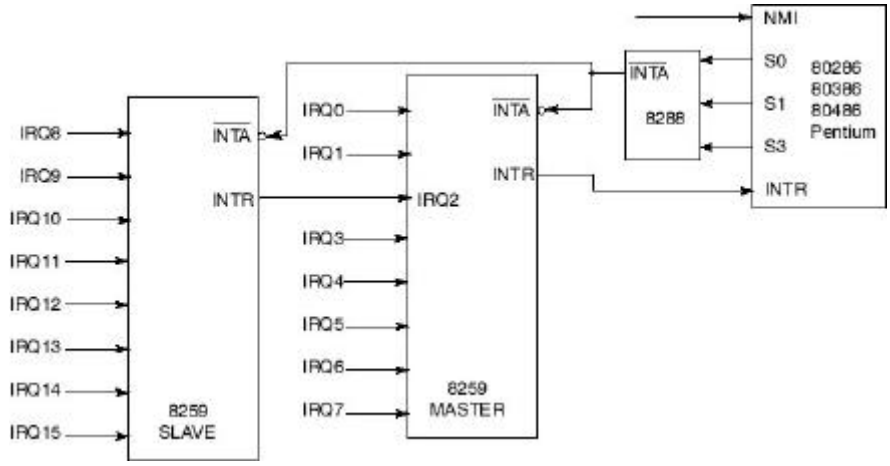
The interrupt flag in the FLAGs register gives programmers the option of enabling or disabling *all* interrupts. Often this is sufficient, but in other cases, the programmer may need to disable only certain interrupts. The 8259A chip has an 8-bit internal register called the IMR (interrupt mask register). Each bit in the register corresponds to one of the interrupt request levels. If bit *n* in the mask is zero, IRQ line *n* is disabled.

In addition to the eight input lines of the 8259A controller, the microprocessor has one other input line for interrupts. This is the NMI (NonMaskable Interrupt) line. As you can see in the

diagram above, several different types of hardware error conditions can trigger this interrupt request. This line, unlike the IRQ lines, signal a hardware failure of some sort or a request from the 8087 floating point processor and therefore it would be inappropriate to attach a normal peripheral device to the NMI line. In a normal situation, the NMI enable line is always set, meaning that if a device asserts an interrupt on one of these lines, the CPU will always receive it. The NMI enable line is inactive only for diagnostic purposes.

The interrupt controller connects to the CPU directly via the INTR (interrupt) line. When one or more devices are requesting an interrupt, the 8259A selects the highest priority device, as mentioned, and encodes the corresponding vector number in one of its internal registers. It then activates the INTR line. When the currently executing instruction finishes, the CPU checks the status of the INTR line. If the line is active and the interrupt flag in the FLAGS register is set, the CPU will process the interrupt. To signal to the interrupt controller that the CPU has recognized the interrupt, it activates the INTA# (interrupt acknowledge) line. The bar over the top of the signal name in the diagram (or the pound sign following it in the text) indicates that this signal uses *negative logic*. This means that a value of zero (no voltage) means "true." As you see from the diagram above, the CPU itself has no INTA# line. Instead, it negates (sets to negative logic "true") three status lines, S0, S1, and S2. The lines connect to additional circuitry in the form of the 8288 chip. When all three lines are zero, the 8388 sets its output line to zero as well, thus indicating to the 8259A chip that the CPU has recognized the interrupt request. At this moment, the 8259A "freezes" the priority by setting the ISR (In Service Register), so that if a device at the same or lower priority device requests an interrupt before the CPU finishes processing the current interrupt, the controller will not pass the new request on to the CPU until the end of the exception processing state. Since the controller will normally activate the INTR line whenever one of the IRQ lines is active, the CPU would continue to receive the interrupt request of the device that it is already processing. The priority "freeze" ensures that the CPU recognizes an interrupt only once.

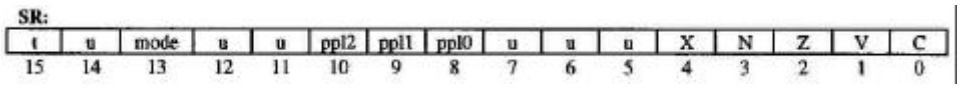
An interesting feature of the 8259 family of interrupt controllers is that it is possible to *cascade* multiple controllers to allow adding more interrupt priority levels and thus more devices. Beginning with the 80286 family of processors, the machines have come standard with a pair of controllers, thus allowing 15 different levels of interrupts. The diagram below shows the basic connection scheme. Note that all of the interrupt levels from the slave controller have lower priorities than that of the device connected to IRQ1, but higher than that of the device connected to IRQ3.



Cascaded Interrupt Controllers ("IBM BIOS Technical Reference", 1984, IBM Corp.)

Motorola 680x0 Family

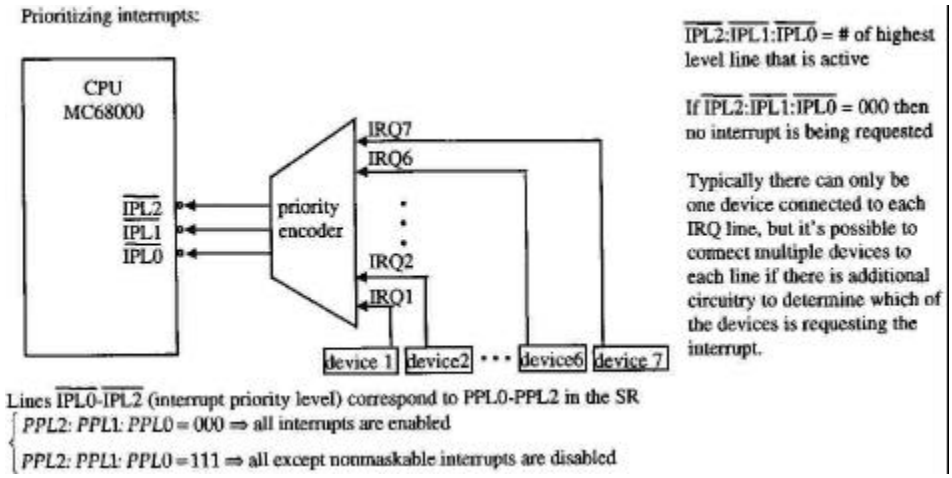
The Motorola family uses some of the same general concepts that we have seen with the Intel family, but interrupt prioritization and recognition also includes some important differences. The Motorola chips have a *status register* (SR) that corresponds in many ways to the Intel FLAGS register. One important difference is that, instead of having a single "interrupt flag" bit, the SR includes a set of three bits. These are the "PPL" bits, PPL0, PPL1 and PPL2, where PPL is an abbreviation for "Processor Priority Level."



Motorola 68000 Status Register

These three bits determine how important an interrupt must be before the CPU will attend to it. This allows the CPU to ignore at least temporarily some relatively unimportant interrupts so that it can process more important ones instead. Each type of peripheral device has an assigned priority level (encoded with a number between 1 and 7). In general, the priority level of an interrupt must be greater than the value in the interrupt mask before the CPU will process it. Thus, if the value of the interrupt mask is 0, all interrupts are "enabled". In other words, no matter what the priority level of an interrupt is, the CPU will recognize it. On the other hand, if the value of the interrupt mask is 7, we say that interrupts are "disabled", meaning that the CPU will process (almost) no interrupt.

These three bits correspond to three signal lines (wires) on the control bus, called $\overline{IPL0}$, $\overline{IPL1}$and $\overline{IPL2}$, where IPL is an abbreviation for "Interrupt Priority Level." These lines bear incoming signals (high or low voltages) from a *priority encoder*. This encoder is a hardware device that receives signals from up to seven different external devices. The encoder selects the highest numbered incoming signal (just as with the Intel processor, devices may be requesting an interrupt on more than one line). It then encodes the number of the selected signal as a 3-bit binary number and outputs this number on the three IPL lines. In general, whenever the priority encoder outputs a number that is greater than the current value of the interrupt mask, the CPU will service the interrupt.



Motorola Interrupt Priority Encoder (from Orejel)

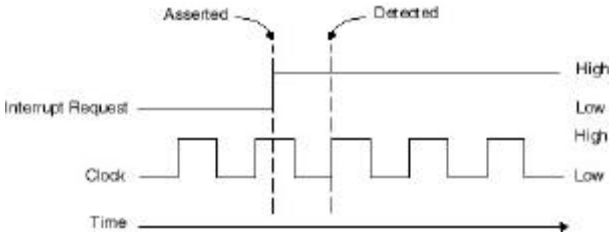
There is one exception to this. An interrupt with priority level 7 is a *non-maskable interrupt* (NMI). Just as for Intel processors, NMIs are typically reserved for only a few situations, generally entailing such catastrophic events as power failure and hardware error. Unlike Intel NMIs, it is impossible to disable an NMI on this chip. Thus, even if the PPL bits are set to seven, the CPU will always recognize a level 7 interrupt. We can express this relationship with the following pseudo-code:

```
if (IPL2:IPL1:IPL0 PPL2:PPL1:PPL0) or (IPL2:IPL1:IPL0 = 7)
   process interrupt
else ignore any pending interrupt request
```

On the Motorola chip, once the CPU recognizes an interrupt request, it must set the interrupt mask to the same level as the interrupt it is processing. Until the CPU can send a signal to the device that is generating the interrupt to let it know that it is receiving the attention it is requesting, it continues to assert the interrupt request. The CPU would accept the interrupt, complete the actions for the exception processing state, then would detect the (same) interrupt request. Since the priority level for the device is (still) higher than the processor priority level, the CPU would *again* accept the interrupt. Obviously, this process would continue until there was a supervisor stack overflow and the device would never receive service. To put it more simply, the continuing interrupt request signal causes the device to "interrupt itself". The Intel 8259A controller solved this problem with the ISR. Since the Motorola machines have a priority encoder, rather than a controller, the ISR register does not exist. If the Motorola CPU did not reset the PPL bits, the requesting device would continue to request an interrupt. Setting the PPL bits to the same level as the interrupt that the CPU is accepting means that the device's priorty level is no longer high enough to merit a response from the CPU. This in turn allows the CPU to service the device's request without further interruptions (unless a higher-priority device requests an interrupt at this moment).

This scheme presents a problem in terms of NMIs, both for Intel and Motorola processors, since it is impossible to set priority high enough to ignore them. For this reason, NMIs are *edge-sensitive* or *edge-triggered*. This means that the CPU recognizes them only at a specific moment in relationship to the signals that the system clock generates. This clock emits signal pulses, alternating between high and low voltages at regular intervals. These transitions can be seen as

the "ticks" of the clock. The CPU recognizes an edge-sensitive interrupt only at the moment of the transition from low to high voltage. Thus, the detection of this type of interrupt occurs within one clock pulse of the moment it is asserted and does not occur again. This scheme prevents an NMI from interrupting itself.



Edge-triggered interrupts

Multiple Simultaneous Exceptions

The discussion of interrupt requests and the response to them of the CPU is explains how a processor can handle situations where more than one hardware device asserts an interrupt at the same time. We also need to look at situations when other kinds of exceptions occur simultaneously. First, we need to specify when the CPU recognizes the various types of exceptions, since this determines what kinds of exceptions *can* occur at at the same time.

When the CPU Recognizes Various Kinds of Exceptions

Aside from remembering when the CPU detects hardware interrupts, you should not need to memorize the times that the CPU recognizes software exceptions. If you realize that the CPU will recognize an exception as soon as possible, you should be able to determine when it will recognize a particular exception, just by reasoning about it. As a result, you should look at the software exceptions in the following list as a means of testing your own reasoning, rather than as a rote-learning task. It is also important to remember that ultimately, the CPU will recognize any exception or interrupt at an *instruction boundary* (that is, between instructions). This is because the occurrence of a software exception aborts any remaining steps in the fetch/execute cycle.

1. The CPU recognizes nonmaskable interrupts as soon as it detects the signal, as detailed above.
2. The CPU recognizes peripheral device interrupts only between instructions.
3. The CPU recognizes bus and address errors such as faults or nonexistent addresses immediately. Since they usually occur during (and as a result of) instruction execution and since they make it impossible to complete execution of the current instruction, this makes perfect sense. These exceptions can occur any time the CPU reads from or writes to memory, so they may occur when the microprocessor fetches an instruction or an operand or when it writes a result back to memory.
4. The CPU recognizes privilege violations, the INT and INTO instructions and illegal or unimplemented instructions as soon as the it decodes the instruction, which is the same as saying that it recognizes them as soon as it detects them.
5. The CPU recognizes a zero divide exception during the execute phase of the divide instruction.
6. The CPU detects trace and other debug exceptions between instructions.
7. The CPU detects bounds violations during the execute phase of the BOUND instruction.

Understanding when the CPU recognizes different sorts of exceptions and interrupts will also help you understand that there are actually very few exceptions that can occur truly simultaneously. For instance, although there are a number of exceptions that the CPU detects when it decodes an instruction, there is only one instruction at a time (ignoring pipelining). Thus, you could never have a privilege violation and an unimplemented instruction at the same time. Although bounds violations and zero divide exceptions both occur during the execute phase of the fetch/execute cycle, it is impossible to be be both dividing and checking bounds simultaneously. If we ignore pipelining, in fact, we can never have two software exceptions occur at the same time unless one is a debug exception such as a trace. Finally, even though two or more peripheral devices may assert an interrupt request simultaneously, the CPU will detect only one of them, since the priority encoder ensures that only one interrupt request at a time will reach the microprocessor.

Prioritizing exceptions

Even with the varying detection times for the different exceptions, certain exceptions can occur simultaneously. For example, a trace exception and a peripheral device interrupt could occur together, since the CPU detects both between instructions. In such a case, it is important to understand what happens. First, not only is there a priority scheme that determines which peripheral device interrupt will receive the CPU's attention first, there is also a priority scheme that encompasses all exceptions and interrupts. You can find the priority scheme that appears below on your reference card. You should also be aware that the INT instruction (not shown with the other priorities) has higher priority than any other. This priority scheme is not so much meant to rank exceptions in terms of "importance". Instead, the priorities are meant to ensure that the behavior of the CPU is appropriate. In fact, it is probably more productive to imagine that a higher priority means *less* important in most cases.

Priority	Exception
Highest Lowest	• trace • breakpoint • segment not present or general protection fault (for instructions) • NMI • maskable interrupt • illegal instruction, privilege violation • coprocessor not available • segment not present, stack fault, general protection fault (for operands) • alignment faults • page faults

Note: These groups and priorities are listed on the back of your reference card just below the Hardware Interrupt Assignment table on the center panel.

Examples of how two exceptions can occur at once:

1. a segment not present fault occurs during the operand fetch of an INT instruction (the INT instruction has priority)

2. a hardware interrupt occurs during execution of an instruction while the trace bit is set (the trace exception has priority)

When multiple exceptions occur simultaneously, the CPU enters the exception processing state for the exception with the highest priority *first*. It completes the exception processing and then, immediately, without executing any of the instructions for this handler, the CPU reenters the exception processing state for the exception with the lower priority level. Once it has obtained the vector number, it fetches the address of the exception handler for *this* exception and places it in the CS and IP registers. Then it executes this handler. Once this routine completes, control returns to the handler for the exception with the higher priority level. This process seems counter-intuitive at first glance. You would expect the CPU to execute the handler for the higher priority exception first. An examination of the second example above (the trace/interrupt example) shows that scheme just outlined is the right way to handle this situation. If the CPU executed the trace routine first, you would end up tracing the first instruction of the interrupt handler, instead of the next instruction of the user program. A programmer using the debugger (the most common use of the trace exception) would certainly be surprised to see a "foreign" instruction that has nothing to do with his or her program.

Processing and Handling Interrupts

Although we will need to fill in details later, the following steps will give you a broad overview of what happens when the CPU recognizes and processes an interrupt. Notice that these steps include the normal exception processing state, but also include other actions that are not part of the exception processing state.

1. The CPU saves state (the FLAGS register or SR and the return address)
2. The CPU (Motorola) or interrupt controller (Intel) disables interrupts of the same or lower priority
3. The CPU clears the trace bit of the SR or FLAGS register (on the Motorola processor, SR also contains a "mode" bit which the CPU sets to 1. This sets the processor operation to *supervisor* mode to allow execution of "privileged" instructions, not available in normal or *user* mode)
4. The CPU obtains the vector number for the interrupt, retrieves the address of the interrupt service routine from the exception vector and places this address in the IP, thus transferring control to the interrupt handler. At this point, we return to normal execution state.
5. The handler identifies and services the condition that caused the device to request an interrupt. The execution of the service routine will include the following (the programmer of the handler must write code to perform these tasks):

a. Optionally disable interrupts. In some situations, a particular service routine must be able to perform at least part of its task without interruption, in which case it would set the interrupt mask or IMR to disable all interrupts.

b. Save scratch registers.

c. Identify the condition that caused the interrupt. A single device often generates interrupts for various different reasons. For example, either moving a mouse or pressing one of its buttons will cause it to generate an interrupt. Similarly, the interface for a modem may generate an interrupt either when its buffer for incoming data is full or when it has transmitted all the data in its buffer for outgoing data and is ready to have the buffer

refilled. These different situations require different actions on the part of the CPU.

d. Service the condition. Often, the interrupt service routine calls subroutines to actually handle the cause of the interrupt. In this case the handler itself is a *dispatch* routine that evaluates the cause of the interrupt and "delegates" the actual work to an appropriate subroutine. On Motorola machines, since the ISR itself is running in supervisor mode, any subroutines it calls will also run in supervisor mode and thus can include instructions that would be illegal in user mode.
e. Enable interrupts if the handler disabled them in step a.
f. Restore scratch registers.
g. Execute an IRET (RTE, return from exception, on Motorola machines) instruction. This pops the return address and FLAGS or SR back off the stack. By popping the FLAGS or SR value, the CPU restores the condition codes that existed before it recognized the interrupt. It also restores the original values of the control flags, such as the trace, mode, and interrupt enable bits that processing or handling the exception might have altered.

Obtaining the Vector Number

There are two methods of determining the vector number for software exceptions. The vector number is a logical index into the vector table that allows the CPU to retrieve the starting address of the exception handler. For most software-generated exceptions that are the result of error conditions (such as divide-by-zero, overflow, address errors, privilege violations, etc.) the circuitry that detects the error condition is also responsible for determining the vector number. The mapping from exception to vector number is automatic, because it is hard-wired. The control unit uses the bits of the instruction opcode itself as inputs to the circuitry that selects the proper vector number.

The second method of obtaining the vector number also involves the bits of the instruction, but in this case, it is not the opcode, but the operand that provides the vector number. The INT instruction includes the interrupt number as an 8-bit immediate operand. This number is the index into the vector table.

There are also two methods of obtaining the vector number for hardware interrupts, although Intel chips use only one of them. Devices are either *vectored*, meaning the device supplies a vector number to the CPU, or they are *autovectored*. Intel uses only vectored devices. We have already seen that the 8259A selects the highest priority IRQ line currently active and places the vector number for the attached device in an internal register. We saw earlier that the CPU sends an interrupt acknowledge signal to the interrupt controller when it recognizes an interrupt request. In fact, it actually sends two "pulses" on this line. The first pulse indicates that the CPU has recognized the interrupt. The second pulse is a request for the 8259A to place the vector number on the data bus to send it to the processor. We can summarize the Intel *interrupt acknowledge cycle* (a special bus cycle, different from the normal read and write cycles) that occurs as a response to an interrupt request as follows:

1. The CPU sets the status lines S0-S2 low (no voltage) to indicate a an interrupt acknowledge cycle. This activates the INTA# line to the 8259A for its first pulse.
2. The 8259A sets the highest priority bit of the ISR and sends a "call" code (CDh) on the data bus to signal that it has received the INTA# pulse and is ready to send the vector

number.

3. The CPU responds with another INTA# pulse (S0-S2 low) to indicate its readiness to receive the vector number.
4. The 8259A sends the preprogrammed vector number to the CPU on the data bus
5. The CPU latches the vector number and completes the exception processing state.

Motorola chips use both vectored and autovectored devices. If a device is autovectored, this means that it uses a default vector number which depends on the priority level of the interrupt. A special section of the Exception vector table contains the starting addresses of handlers for autovectored devices (vector numbers 25-31). When a hardware interrupt occurs, the MC680x0 also executes an . Unlike the Intel interrupt acknowledge cycle, the Motorola chip will also need to determine which sort of interrupt is occurring and thus how it should obtain the vector number. This cycle consists of the following steps:

1. The CPU first sets the function code lines, FC0-FC2, high to indicate what is called a "CPU space cycle", i.e., a special function CPU cycle, as opposed to a normal instruction execution. These lines are the counterpart to Intel's S0-S2 lines.
2. The CPU sets the address lines A16-A19 high to indicate that this space cycle is more specifically an interrupt acknowledge cycle. The function code lines alone are not enough to determine the exact type of bus cycle, as they are on the Intel chip.
3. The CPU places $\overline{IPL2}$:$\overline{IPL1}$:$\overline{IPL0}$ on lines A1-A3 of the address bus and sets the remaining address lines high
4. The CPU asserts $\overline{AS}$, $\overline{UDS}$and $\overline{LDS}$($\overline{UDS}$ is not actually used, since vector numbers are one byte, but it is set for consistency with other types of CPU cycles (review the Motorola read and write bus cycle description in Machine Overview if you have forgotten the meaning of these signals).
5. The CPU sets the R/$\overline{W}$ signal to read
   a. If the interrupt is vectored, the device places the vector number on the data bus and asserts $\overline{DTACK}$
   b. If the device is autovectored, the device asserts $\overline{VPA}$(Valid Peripheral Address). This signals the CPU to calculate the vector number from the priority level of the device (that is, the current value of $\overline{IPL1}$:$\overline{IPL1}$:$\overline{IPL0}$).

Context Switches and Interrupt Latency

It should be clear that processing interrupts takes time away from processing the instructions of user programs. Conversely, we might also contend that processing user programs takes time away from attending to peripheral devices! In either case, it is to our best interest to make interrupt processing as fast as possible. We also would like to have peripheral device requests to receive service as promptly as possible. This latter goal leads us to examine what factors affect the amount of time it will take for a peripheral device to receive the attention of the CPU and thus affect the performance characteristics of I/O operations. We learned about context switches when we talked about multi-user or multitasking systems. At that time, we learned that context switches entail saving the current CPU register values and restoring a set of values that belonged to the process that was gaining access to the CPU. When we discuss context switches in terms of interrupt processing, we are talking about a similar process. Context switches are again the action of saving a state. This time, however, we are not changing from one user or program to another, but from a normal execution state (typically a user program running in user mode) to another

normal execution state, but this time with the processor executing an interrupt handler.

We can define context switch time as the time it takes to perform a state save. Part of the state save occurs before entering the handler, the rest occurs as a result of the first instruction(s) of the service routine itself. More explicitly, context switch includes the following steps:

1. push the FLAGS or SR and the return address to the stack
2. set PPL2:PPL1:PPL0 to $\overline{IPL1}$:$\overline{IPL1}$:$\overline{IPL0}$ (this keeps the request from causing further interrupts until the device actually removes the source of the interrupt) or set the ISR
3. clear the trace bit and, on the Motorola chip, set the mode bit of the SR to 1
4. obtain the vector number so that the CPU can retrieve the address of the service routine
5. execute the first instruction(s) of the service routine to save scratch registers and (optionally), to disable interrupts

Context switch time is one component necessary to understanding another important concept for exception processing: *interrupt latency*. We can define interrupt latency as the time elapsed from the appearance of the interrupt to the time that the processor starts to execute the handler. It should be plain to you that the shorter interrupt latency is, the better performance will be. The maximum interrupt latency that can occur depends on at least two factors. First and most obvious is context switch time itself. The second factor is also fairly obvious if we realize that the microprocessor only recognizes hardware interrupts (with the exception of NMIs) *between* instructions. That means that if a device generates an interrupt just as the execution of an instruction is beginning, it will have to wait until execution is complete before the CPU will recognize it. Thus, in the simplest case, we can define the maximum interrupt latency possible (and thus the worst performance possible) as:

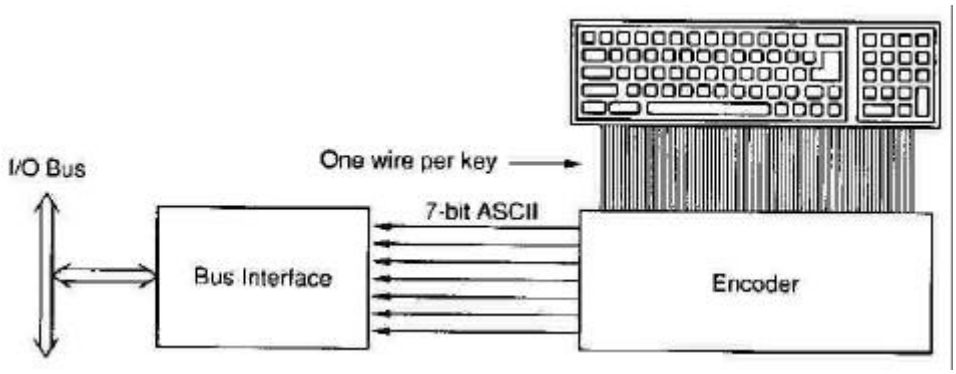
Max Latency = execution time of slowest machine instruction + context switch time

It may not be quite this simple to define maximum interrupt latency, however. If we allow processes or service routines to temporarily disable interrupts, the latency times may stretch out considerably, since we also have to add the time for which interrupts are disabled. Thus a more complete definition of maximum interrupt latency would be:

Max Latency = execution time of slowest machine instruction + context switch time + maximum amount of time that interrupts are disabled

Communication Between I/O Devices and the CPU

Typically, I/O devices require additional hardware in the form of a *controller* or *interface* (often an expansion card in the machine or a chip on the device itself or on the motherboard) to make up the peripheral interface to resolve timing and format differences between the device and CPU. For example, a keyboard has one line connected to each key. When you press a key, the wire connected to that key carries voltage (the line is activated). To be useful to the CPU, this voltage must be translated to an ASCII code. The translation is the task of the encoder/controller.



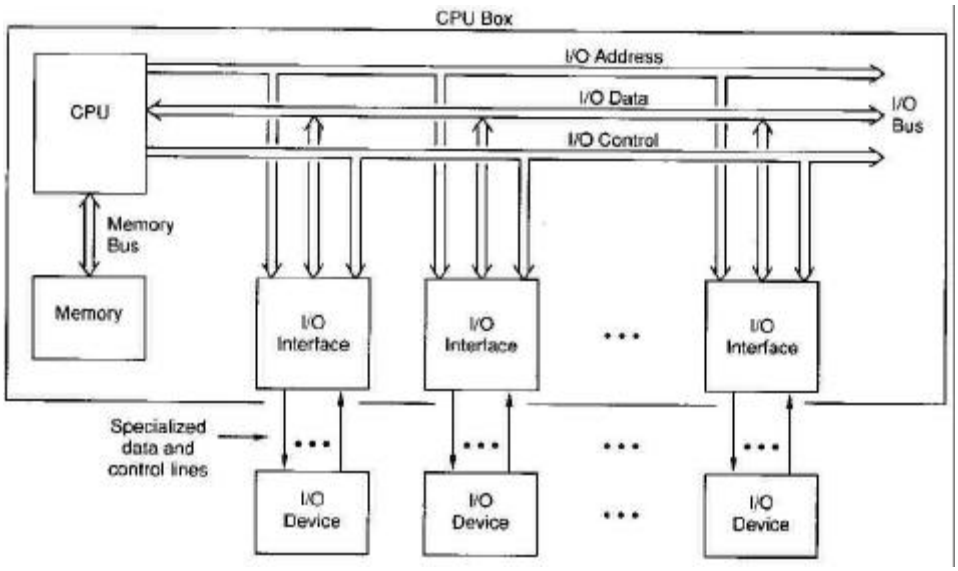
Keyboard and encoder/controller (from Wakerly)

Interfaces usually have their own sets of registers or *ports*. We have already mentioned some of the registers on the 8259A interrupt controller. In general, controllers include one or more status and control registers. A particular controller may also have more "special purpose" registers, such as the IMR, ISR, and vector number registers we saw on the 8259A. Other common registers are transmit and/or receive registers (buffers). The exact number and kind of registers depends on the device. For example, a video controller may have a cursor register for coordinates of the cursor.

Communication between the device and the CPU depends on two different (but equally necessary) factors. First, are the control lines between the device and the CPU. Second, are the *I/O ports*, which you should think of as memory locations, but which are typically registers, to which both the device and the CPU have access. The two basic types of interface for these registers or ports are *isolated I/O* and *memory mapped I/O*.

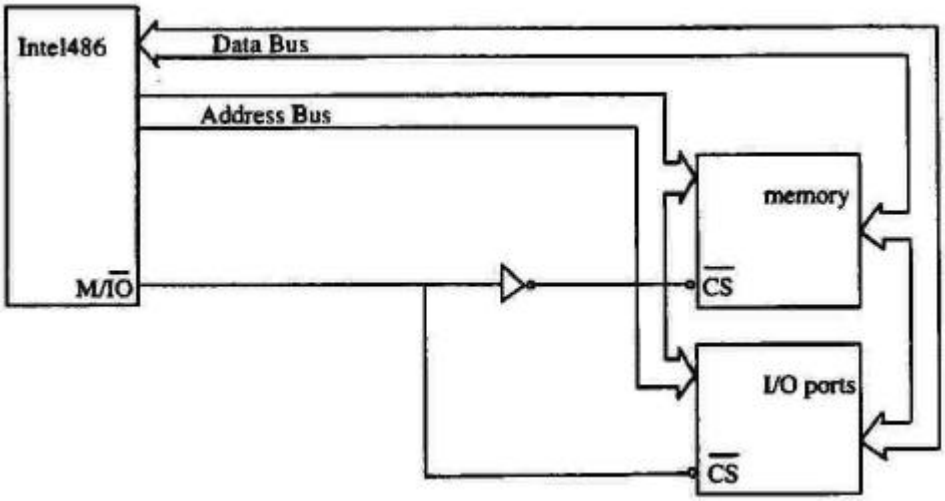
Isolated I/O

Isolated I/O requires special I/O instructions. Since they have less general requirements than normal MOVE instructions and they require fewer memory accessing modes, they require less circuitry to execute. As a result, they are faster than standard MOVE instructions. We can say they are *optimized* for I/O. Typically, the mnemonics for these instructions are IN and OUT, as they are for the Intel family of chips. True isolated I/O also requires a dedicated I/O bus, separate from normal data and address buses. Having separate buses for I/O means that bus cycles can occur simultaneously for conventional memory and I/O devices. This reduces bus contention and provides better performance.



Isolated I/O (from Wakerly)

Intel machines incorporate a less expensive form of isolated I/O. They use a control line to select between conventional memory and I/O ports rather than a dedicated I/O bus. The bus connects to a bank of 8-bit I/O ports to which both the device and the CPU have access. The same bus also connects to conventional memory. The ports are logically an extra bank of memory used strictly for interfacing with external devices. In the Intel scheme, the M/$\overline{IO}$ line selects which chip(s) will be active, thus determining whether a particular address maps to standard memory or to the bank of I/O ports. This means that there can be two separate *address spaces*. An address space is the range of addresses possible given a particular width for the address bus. Having two separate address spaces means that there are actually two different memory locations with address $0000, two with address $0001, etc.; one in standard memory and another in the set of I/O ports.

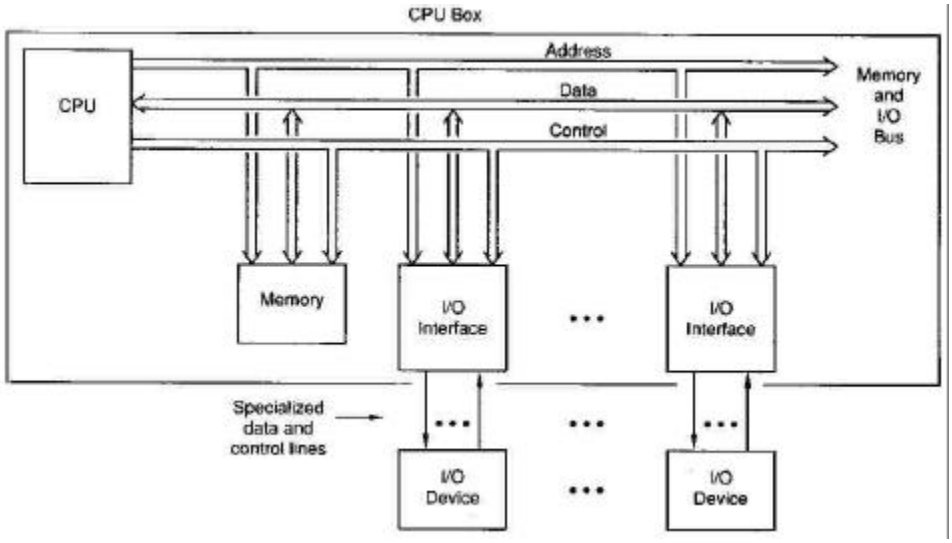


Intel's Version of Isolated I/O, Without Dedicated Buses (from Orejel)

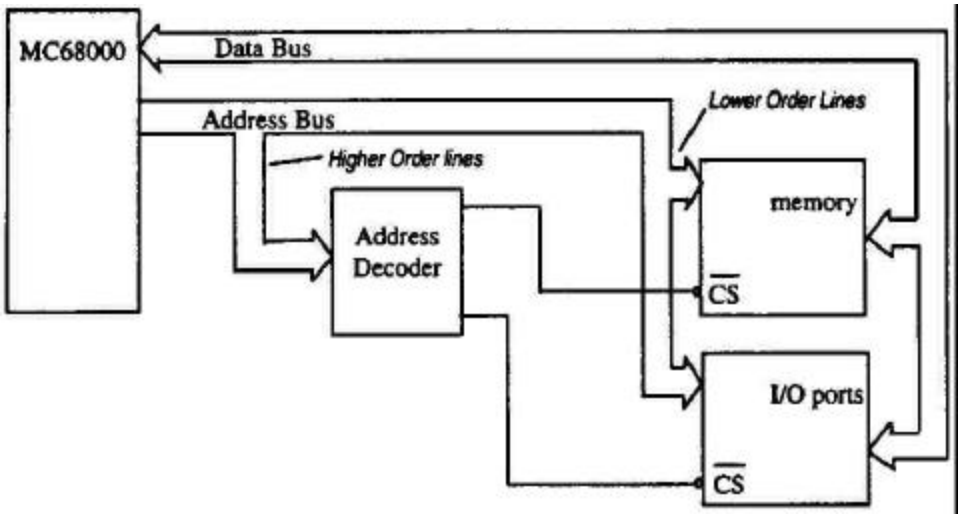
Memory Mapped I/O

Memory mapped I/O uses existing instructions. This implies that all the standard addressing modes available for MOVE instructions are also available for accessing I/O ports. Memory

mapped I/O also uses the existing address and data buses. I/O ports in this scheme occupy part of standard memory address space. A block of memory addresses is reserved for I/O; usually this block is in the highest part of the address space. The conventional memory chips installed in the computer do not include these addresses.



Memory-Mapped I/O (from Wakerly)



Memory-Mapped I/O on the MC68000 (contrast with Intel) (from Orejel)

Advantages of memory mapped I/O:

1. No special opcodes are necessary, making instruction set design simpler (the fewer the opcode bit patterns, the easier it is to avoid ambiguity in these patterns).
2. No special circuitry is necessary. That is, there is no need for a dedicated bus. This means it is cheaper.
3. There are more addressing modes available for accessing I/O ports.

Advantages of isolated I/O:

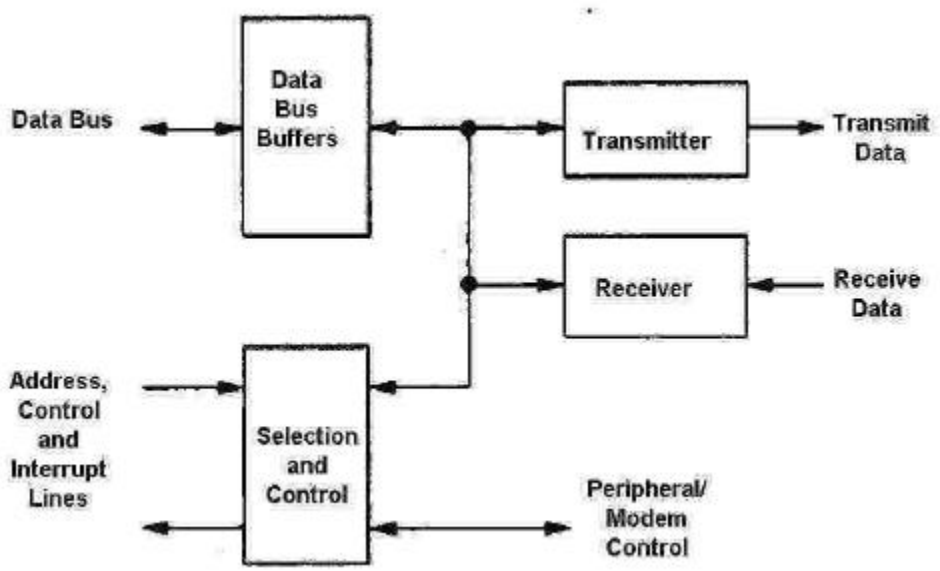
1. I/O ports occupy none of the memory address space. This provides more memory addresses for user programs.

2. Interfaces (controllers) need less circuitry since they do not have to decode 24-bit addresses (only one byte is needed to designate a unique I/O port).
3. I/O instructions are quicker because they are optimized for a single special purpose. As a rule, the more general an operation must be, the more complex and thus the slower it will be.
4. In the case of true isolated I/O with a dedicated bus, there is less bus contention, which also leads to increased performance.
5. A machine with isolated I/O can emulate memory-mapped I/O if desired, with no loss of performance over a true memory mapped system. Note that a system with memory mapped I/O cannot emulate a machine with isolated I/O without a loss of performance over a true isolated I/O machine because it inherently lacks the additional hardware and the "extra" address space of a machine designed to use isolated I/O.

Caution: I/O ports (even in the case of memory mapped I/O) are not part of the physical RAM. Ports are registers on an I/O interface chip. Unlike the CPU registers, however, they have no names, but instead have addresses. With memory-mapped I/O, they are *logically* part of memory, and their addresses are an extension of the addresses for standard RAM chips, but physically they are on separate chips. With isolated I/O, they occupy a separate address space from standard RAM, so obviously they *must* be on separate chips. In either case, it is well to remember that standard RAM itself is essentially a large collection of slow "registers," accessed via an address, rather than a name. If you look at a memory add-on card, you will notice that it includes several separate chips. Each chip contains a portion of the total memory. If you think about it this way, the notion of I/O ports should seem more intuitive.

Case study: ACIA (Asynchronous Communications Interface Adapter)

The ACIA is an interface for various serial communications devices (typically a modem), that can both transmit and receive data, similar in function to the UART (Universal Asynchronous Receiver Transmitter). Serial devices transmit or receive data one bit at a time over a single wire (such as a telephone line). Several data bits in succession (usually seven or eight of them, to form a single character) make up a single data item. If the ACIA is receiving data, it must collect the bits until there are enough to send to the CPU as a character. If the ACIA is transmitting data, it receives an entire character from the CPU and it must store the bits until it can send all of them (one at a time). For this reason, it has a set of *buffers*, or memory locations (ports), connected to the data bus that store the incoming or outgoing data bits. The ACIA is also connected to the address bus so that the CPU can gain access to particular registers on the interface. Finally, the ACIA is connected to an interrupt line and to the device itself.



Block Diagram of the ACIA (from Motorola MC6850 data sheet)

The ACIA has four onboard registers (ports):

1. the Transmit Data Register (TDR), which is a buffer for data that the ACIA must transmit
2. the Receive data register (RDR), which is a buffer for data that the ACIA is receiving
3. the 8-bit Control register (CR), which is a write-only register that the CPU uses to direct the permitted actions of the interface*
4. 8-bit Status register (SR), which is a read-only register that the CPU uses to determine the current status of the interface*

   * write-only and read-only are always from the point of view of the CPU

Since the device both receives and transmits data, as programmers, we must solve the problem of determining what sort of service the device is requesting when an interrupt occurs. The interrupt service routine must use the control and status registers as a means of both controlling what we want to allow the device to do and of determining the meaning of an interrupt issuing from the ACIA. This will allow the interrupt service routine to determine what actions to perform. The bits of these registers have particular meanings and, when taken collectively, give all the information necessary for the ISR to perform the correct actions. The meanings of the bits appear below.

CR:	CR0-CR1	used to reset the device to its start-up state (among other purposes)
	CR2-CR4	used to select the word length (7 or 8-bits), parity (odd or even) and stop bits (1 or 2)
	CR5-CR6	used as transmitter control bits. Via these bits, the CPU can either enable or disable transmit interrupts
	CR7	used as a receive control bit. Via this bit, the CPU can either enable or disable receive interrupts
SR:	SR0 (RDRF)	Receive Data Register Full--cleared when CPU reads data from the RDR
	SR1 (TDRE)	Transmit Data Register Empty (that is, it is ready to send more data)--cleared when the CPU writes data written into the TDR
	SR2 (DCD)	Data Carrier Detect--set when there is no carrier (that is, the device is

	disconnected)
SR3 (CTS)	Clear to Send--set by modem
SR4 (FE)	Framing Error--set when a character was not properly preceded and followed by a start and/or stop bits, which indicates an error in transmission of some kind
SR5 (OVRN)	OVerRuN--data was lost, that is, it was received but the CPU did not read it before it was overwritten
SR6 (PE)	parity error
SR7 (IRQ)	Interrupt ReQuest pending--cleared by a read from the RDR or a write to the TDR

The interrupt service routine uses information from the status register to determine what to do. For example, if the ACIA has set SR0, then an interrupt is a request for the interrupt service routine to read data. If the ACIA has set SR1, then an interrupt is a request for the handler to write data. Note that other services may be necessary for error recovery, as indicated by other bits in the SR.

References

Eggebrecht, L.C. *Interfacing to the IBM Personal Computer.* Indianapolis: Howard W. Sams and Co. 1983.

Ford, William and William Topp. *Assembly Language and Systems Programming for the M68000 Family* (Second Ed.). Lexington, MA: D.C. Heath, 1992.

Intel. *i486 Microprocessor: Hardware Reference Manual.* Intel Corporation, 1990.

86/88, 186/188 User's Manual: Programmer's Reference. Intel Corporation, 1985.

Intel486™ Microprocessor Family: Programmer's Reference Manual. Intel Corporation, 1992.

Intel. *Microprocessor and Peripheral Handbook.* Intel Corporation, 1983.

IBM. *IBM BIOS Technical Reference.* International Business Machines Corporation, 1984.

Motorola. *M68000 8-/16-/32-Bit Microprocessors: User's Manual* (Sixth Ed.). Englewood Cliffs, NJ: Prentice Hall, 1989.

Motorola. Data Sheet for MC6850, MC68A50 and MC68B50, 1978.

Orejel, Jorge. Personal Communication.

Wakerly, John. *Microcomputer Architecture and Programming: The 68000 Family*. NY: Wiley, 1989.

Dell™ PowerEdge™ Systems

MICROPROCESSOR UPGRADE GUIDE

www.dell.com

Notes, Notices, Cautions, and Warnings

Throughout this guide, blocks of text may be accompanied by an icon and printed in bold type or in italic type. These blocks are notes, notices, cautions, and warnings, and they are used as follows:

NOTE: A NOTE indicates important information that helps you make better use of your computer system.

NOTICE: A NOTICE indicates either potential damage to hardware or loss of data and tells you how to avoid the problem.

CAUTION: A CAUTION indicates a potentially hazardous situation which, if not avoided, may result in minor or moderate injury.

WARNING: A WARNING indicates a potentially hazardous situation which, if not avoided, could result in death or serious bodily injury.

January 2000 P/N 9661P Rev. A08

Contents

Figures

Dell™ PowerEdge™ Systems — Microprocessor Upgrade Guide

WARNING: The power supplies in a PowerEdge system may produce high voltages and energy hazards, which can cause bodily harm. Only trained service technicians are authorized to remove the computer cover and access any of the components inside the computer. For more information, see "Safety Instructions" in your system's *Installation and Troubleshooting Guide*.

This document provides procedures for upgrading the Intel® Pentium® II or III microprocessors with either Intel Pentium II or Pentium III microprocessors in the following Dell PowerEdge systems:

- PowerEdge 1300
- PowerEdge 2300
- PowerEdge 2400
- PowerEdge 4300
- PowerEdge 4350

Installing one or more microprocessors in your server may involve the following activities:

- Verifying the basic input/output system (BIOS) revision and saving the current configuration data
- Accessing the system board (see your system's *Installation and Troubleshooting Guide*)
- Replacing the system board mounting plate or tray (PowerEdge 2300 and 4300)
- Installing the upgrade microprocessor
- Installing the cooling shroud (if applicable)
- Reassembling and checking the system

The upgrade procedure requires a #2 Phillips screwdriver. In addition, you should use a wrist grounding strap for electrostatic discharge (ESD) protection. Read the safety instructions in the following section.

The contents of your kit will vary, depending on the PowerEdge system and the number of microprocessors you are installing. Each kit will have one or more new Pentium II or Pentium III microprocessor(s), diskettes containing the Resource Configuration Utility (RCU), BIOS, embedded server management (ESM) firmware, and diagnostics. Also included, for some system models, is a cooling shroud and mounting hardware or a replacement system board mounting plate or tray.

Precautionary Measures

Before you perform any of the procedures in this document, take a few moments to read the following warning for your personal safety and to prevent damage to the computer system from ESD.

WARNING FOR YOUR PERSONAL SAFETY AND PROTECTION OF THE EQUIPMENT

Before you start to work on the computer, perform the following steps in the sequence listed:

1. Turn off your computer and any devices.
2. Ground yourself by touching an unpainted metal surface on the chassis, such as the metal around the card-slot openings at the back of the computer, before touching anything inside your computer.

   While you work, periodically touch an unpainted metal surface on the computer chassis to dissipate any static electricity that might harm internal components.
3. Disconnect your computer and devices from their power sources. Also, disconnect any telephone or telecommunication lines from the computer.

   Doing so reduces the potential for personal injury or shock.

In addition, take note of these safety guidelines when appropriate:

- When you disconnect a cable, pull on its connector or on its strain-relief loop, not on the cable itself. Some cables have a connector with locking tabs; if you are disconnecting this type of cable, press in on the locking tabs before disconnecting the cable. As you pull connectors apart, keep them evenly aligned to avoid bending any connector pins. Also, before you connect a cable, make sure both connectors are correctly oriented and aligned.
- Handle components and cards with care. Don't touch the components or contacts on a card. Hold a card by its edges or by its metal mounting bracket. Hold a component such as a microprocessor chip by its edges, not by its pins.

If your system has two microprocessors installed, the secondary microprocessor must be the same type and have the same operating frequency and cache size as the primary microprocessor. For example, if the system you are installing has a 500-megahertz (MHz) Pentium III primary microprocessor, the secondary microprocessor must also be a 500-MHz Pentium III microprocessor.

NOTICE: Do not attempt to operate a system with one Pentium II microprocessor and one Pentium III microprocessor. Damage to one or both of the microprocessors and/or the system board may occur.

NOTICE: All empty microprocessor connectors must be populated with a terminator card. If your system supports more than one microprocessor and you are not installing the maximum number of microprocessors, the remaining microprocessor connectors must have a terminator card.

Before You Begin

Before shutting down your system, perform these preliminary steps:

- Record the system configuration screens.
- Update the ESM firmware.
- Update the BIOS (if necessary).
- Use the RCU diskette (provided in the kit) to save the RCU configuration settings (see your *User's Guide* for complete information).

Recording the System Configuration

View the system configuration screens in the System Setup program and make a record of the settings.

Updating the ESM Firmware

If an ESM firmware diskette is included with your kit, update your ESM firmware with the version contained on that diskette by performing the following steps. The latest version of the ESM firmware is available online at **http://support.dell.com**.

1. Insert the ESM firmware diskette provided in the upgrade kit into the diskette drive.
2. Reboot the system.
3. After the system completes the boot routine, follow the instructions on the screen.
4. After the `Successfully completed Done` message appears on the screen, remove the ESM firmware diskette from the diskette drive and follow the instructions on the screen to reboot the system.

Updating the BIOS

If a BIOS diskette is included with your kit, update your BIOS with the version contained on that diskette by performing the following steps. The latest version of the BIOS is available online at **http://support.dell.com**.

1. Insert the BIOS diskette provided in the upgrade kit into the diskette drive.
2. Reboot the system.

3. After the system completes the boot routine, follow the instructions on the screen.

4. After the `BIOS has been successfully installed` message appears on the screen, remove the BIOS diskette from the diskette drive and follow the instructions on the screen to reboot the system.

Saving RCU Configuration Settings

Use the RCU to save the current system configuration settings by performing the following steps:

1. Insert the RCU diskette into the diskette drive and reboot the system.

2. When the welcome screen appears, press <Enter>.

   The main menu appears.

3. Select **Step 5: Save and Exit**, and then follow the online instructions to save the current system configuration information.

NOTE: The RCU recognizes microprocessors operating at 450 MHz and faster. The latest version of the RCU is available online at ***http://support.dell.com****.*

Installing Upgrade Microprocessors in the PowerEdge 1300

To upgrade to Pentium II or Pentium III microprocessors in the PowerEdge 1300, perform the following steps:

1. Access the system board, which involves the following steps:

   a. Disconnecting power and peripheral cables.

   b. Removing the covers.

   c. Rotating the power supply.

   See your system *Installation and Troubleshooting Guide* for instructions.

2. Remove the microprocessors.

   See "Removing the Microprocessors," found later in this document.

3. Remove and replace the guide brackets.

   See "Removing and Replacing the Guide Brackets," found later in this document.

4. Remove the cooling shroud.

   See "Removing and Replacing the Cooling Shroud," found later in this document.

5. Install the upgrade microprocessor.

   See "Installing the Upgrade Microprocessors" found later in this document.

6. Replace the cooling shroud.

   See "Removing and Replacing the Cooling Shroud," found later in this document.

   If the upgrade kit comes with a new cooling shroud, you must install the new cooling shroud. See "Installing a New Cooling Shroud," found later in this document.

7. Reassemble and check the system, as follows:

   a. Rotate the power supply back into position, making sure that the securing tab snaps into place.

   b. Replace the covers and front bezel. Reconnect your computer and peripherals to their power sources and turn them on.

      As the system boots, it detects the presence of the new microprocessor and automatically changes the system configuration information in the System Setup program.

NOTE: After you remove and replace the cover of a PowerEdge 1300 system, the chassis intrusion detector causes the following message to be displayed at the next system start-up:

```
ALERT! Cover was previously removed.
```

   c. Enter the System Setup program and confirm that the top line in the system data area correctly identifies the installed microprocessor(s). By default, the serial numbers of Pentium III microprocessors are not displayed. See the procedures in "Using the System Setup Program" in your *User's Guide* for accessing and modifying entries in the System Setup screens.

   d. Reset the chassis intrusion detector while in the System Setup program by changing **Chassis Intrusion** to **Not Detected**.

NOTE: If a setup password has been assigned by someone else, contact your network administrator for information on resetting the chassis intrusion detector.

   e. Run the Dell Diagnostics to verify that the new microprocessor is operating correctly.

      See your *User's Guide* and your computer *Installation and Troubleshooting Guide* for additional information on running the Dell Diagnostics and troubleshooting any problems that may occur.

Installing Upgrade Microprocessors in the PowerEdge 2300

To upgrade to Pentium II or Pentium III microprocessors in the PowerEdge 2300, perform the following steps.

WARNING: The power supplies in this computer system produce high voltages and energy hazards, which can cause bodily harm. Only trained service technicians are authorized to remove the computer cover and access any of the components inside the computer.

NOTICE: Observe the safety information given in "Precautionary Measures" and "Before You Begin" to ensure that the system configuration screens are recorded and the system is properly shut down and disconnected from all power sources.

NOTES: If you are installing a 600 MHz (or greater) microprocessor in your PowerEdge 2300, the system board mounting plate must be replaced. A replacement mounting plate is provided with the upgrade kit.

If you are upgrading a PowerEdge 2300 with one or two microprocessors with an operating frequency less than 600 MHz, you do not need to replace the system board mounting plate.

1. Access the system board, which involves the following steps:

   a. Disconnecting power and peripheral cables.

   b. Removing the covers.

   c. Removing the front bezel.

   d. Removing the cooling fan shroud.

   e. Replacing the system board mounting plate or tray. Applies only when installing 600 MHz or greater microprocessors.

   See your system *Installation and Troubleshooting Guide* for specific instructions, if needed.

2. If you are installing a 600 MHz or greater microprocessor, replace the system board mounting plate as follows.

   a. Unlock and remove the front bezel.

   b. Remove the right- and left-side computer covers.

   c. Record the location and disconnect all external peripheral cables from their connectors on the back of the computer.

   d. Record the location of any internal expansion card cables, and then record the slot number assignments and remove all the expansion cards.

      Place the expansion cards in antistatic bags or on a grounded antistatic mat or other antistatic surface.

e. Record the locations and disconnect all internal cables attached to the system board.

f. At the left side of the system, locate and remove the three screws that secure the system board and mounting plate to the chassis (see Figure 1-1).

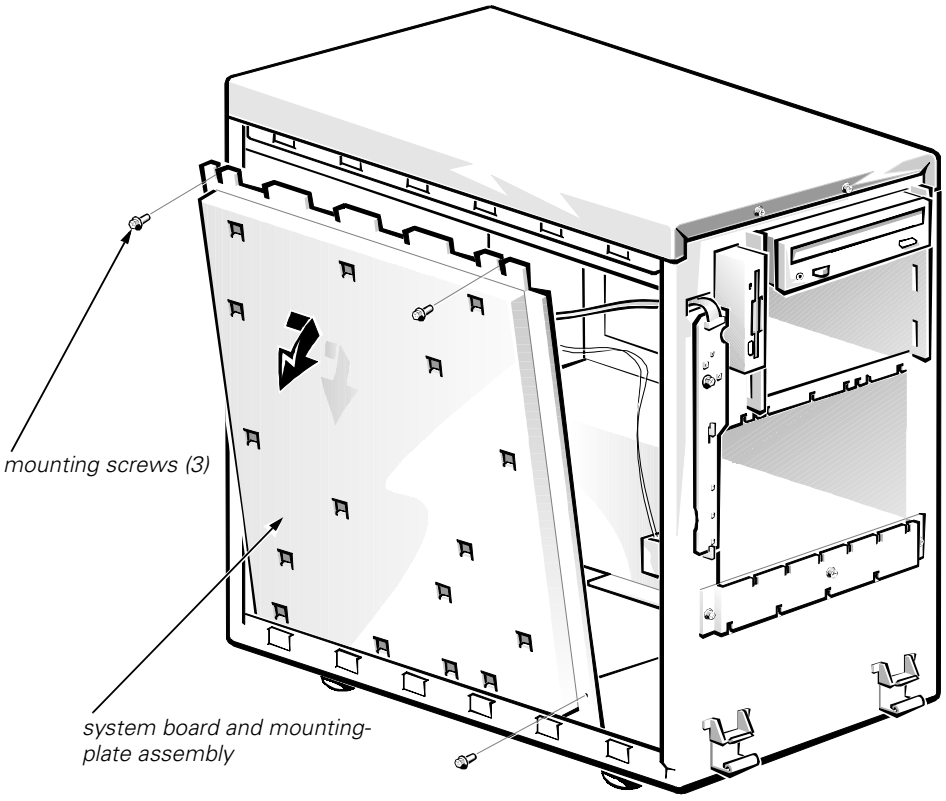


Figure 1-1. Removing the System Board Assembly

g. Slide the system board and mounting plate assembly forward about 12.7 millimeters (mm) (0.5 inch), rotate it away from the top of the chassis, and lift it out of the chassis.

h. Lay the assembly with the system board facing up on a flat surface.

i. Remove the mounting screw from the mounting plate.

j. Slide the system board about 6.3 mm (0.25 inch) toward the front (left edge of the mounting plate) and lift the system board up and away from the mounting plate.

3. To install the replacement mounting plate assembly (provided in your upgrade kit) in your PowerEdge 2300 system, perform the preceding steps in reverse order.

4. Remove the microprocessor.

   See "Removing the Microprocessors," found later in this document.

5. Remove the cooling shroud.

   See "Removing and Replacing the Cooling Shroud," found later in this document.

6. Install the upgrade microprocessor.

   See "Installing the Upgrade Microprocessor," found later in this document.

7. Replace the cooling shroud.

   See "Removing and Replacing the Cooling Shroud," found later in this document.

   If the upgrade kit comes with a new cooling shroud, you must install the new cooling shroud. See "Installing a New Cooling Shroud," found later in this document.

8. Reassemble and check the system.

   See "Reassembling and Checking the System," found later in this document.

Installing Upgrade Microprocessors in the PowerEdge 2400

To upgrade to a faster speed Pentium III microprocessor in the PowerEdge 2400, perform the following steps.

NOTE: The microprocessor is contained within a single-edge contact (SEC) cartridge and heat sink assembly. The system board has two guide bracket assemblies, which hold the SEC cartridge and heat sink assemblies. If your system has only one microprocessor, the secondary guide bracket assembly connector must contain a terminator card.

NOTICE: The terminator card must be rated to run at 133 MHz.

NOTE: If you are adding a microprocessor, the secondary microprocessor must have the same operating frequency as the first. For example, if the system has a 500-MHz primary microprocessor, your secondary microprocessor must also be a 500-MHz microprocessor.

1. Remove the right-side computer cover.

   See your system *Installation and Troubleshooting Guide* for specific instructions, if needed.

2. Remove the SEC cartridge and heat sink assembly, as follows.

CAUTION: The SEC cartridge and heat sink assembly can get extremely hot during system operation. Be sure the assembly has had sufficient time to cool before you touch it.

CAUTION: When handling the SEC cartridge and heat sink assembly, take care to avoid sharp edges on the heat sink.

NOTICE: See "Precautionary Measures," found earlier in this document for important information to prevent damage to the system from ESD.

a. Remove the cooling shroud.

   See "Removing and Replacing the Cooling Shroud," found later in this document.

b. Pull the tab on one side of the guide bracket away from the end of the heat sink and pull up slightly on the cartridge.

c. Deflect the tab on the other end of the guide bracket to disengage the tab on the heat sink, and then lift the cartridge and heat sink assembly away from the guide bracket assembly (see Figure 1-2).

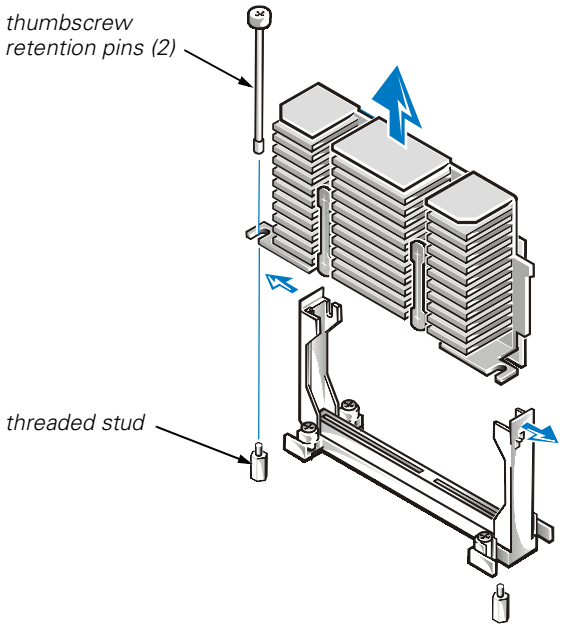


Figure 1-2. Removing an SEC Cartridge and Heat Sink

3. Install the replacement SEC cartridge and heat sink assembly as follows:

   a. Remove the terminator card or old SEC cartridge from the guide bracket assembly.

   b. Slide the SEC cartridge into the guide bracket assembly, and firmly seat the assembly until the tabs on the guide bracket assembly snap into place over the ends of the heat sink (see Figure 1-3).

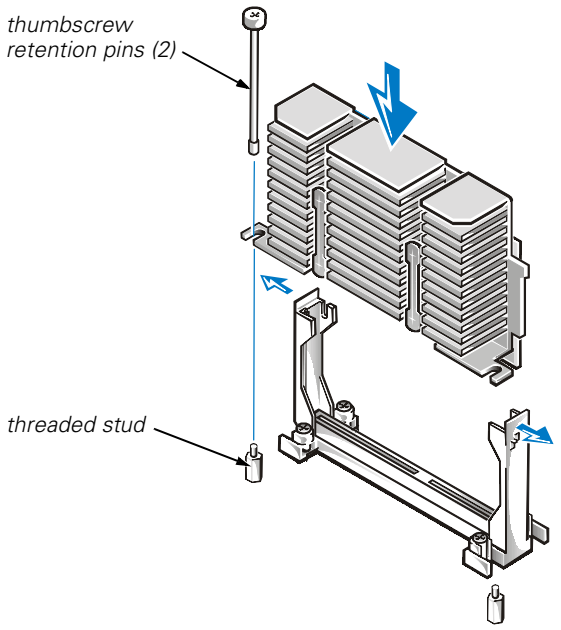


Figure 1-3. Installing an SEC Cartridge and Heat Sink Assembly

4. Reassemble and check the system.

   See "Reassembling and Checking the System," found later in this document.

Installing Upgrade Microprocessors in the PowerEdge 4300

To upgrade to Pentium II or Pentium III microprocessors in the PowerEdge 4300, perform the following steps.

NOTES: If you are installing a 600-MHz (or greater) microprocessor in your PowerEdge 4300, the system board mounting tray must be replaced. A replacement mounting tray is provided with the upgrade kit.

If you are upgrading a PowerEdge 4300 with a microprocessor with an operating frequency less than 600 MHz, you do not need to replace the system board mounting tray.

1. Remove the system board mounting tray as follows.

WARNING: The power supplies in this computer system produce high voltages and energy hazards, which can cause bodily harm. Only trained service technicians are authorized to remove the computer cover and access any of the components inside the computer.

NOTICE: Observe the information given in "Precautionary Measures" and "Before You Begin" to ensure that the system configuration screens are recorded and the system is properly shut down and disconnected from all power sources.

   a. Unlock and remove the computer cover.

      For instructions, see the *Installation and Troubleshooting Guide*.

   b. To access the system board, release the system-board tray latch at the back lower corner of the tray (see Figure 1-4) and pull the tray open to the first stop position (or *service position*).

NOTE: From the service position, if you depress and release the tray latch and pull the tray out again, you will come to a second stop position that is used by manufacturing. To remove the tray completely from any position, depress and hold the latch, and pull the tray out of the chassis.

   c. Record the location of any internal expansion card cables, record the slot number assignments, and then remove all the expansion cards.

      Place the expansion cards in antistatic bags or on a grounded antistatic mat or other antistatic surface.

   d. Record the locations and disconnect all internal cables attached to the system board.

   e. Press and hold the tray release latch as you slide the system board and mounting tray assembly completely out of the chassis.

   f. Lay the tray assembly with the system board facing up on a flat surface.

   g. Remove the mounting screw from the mounting tray.

   h. Slide the system board 6.3 mm (0.25 inch) toward the front (left edge of the mounting tray, as shown in Figure 1-5) and lift the system board away from the mounting tray.

2. To install the upgrade mounting tray assembly (provided in your upgrade kit) in your PowerEdge 4300 system, perform the preceding steps in reverse order.

3. Remove the microprocessors.

   See "Removing the Microprocessors," found later in this document.

4. Replace the guide brackets.

   See "Removing and Replacing the Guide Brackets," found later in this document.

5. Install the upgrade microprocessor.

   See "Installing the Upgrade Microprocessor," found later in this document.

6. Reassemble and check the system.

   See "Reassembling and Checking the System," found later in this document.

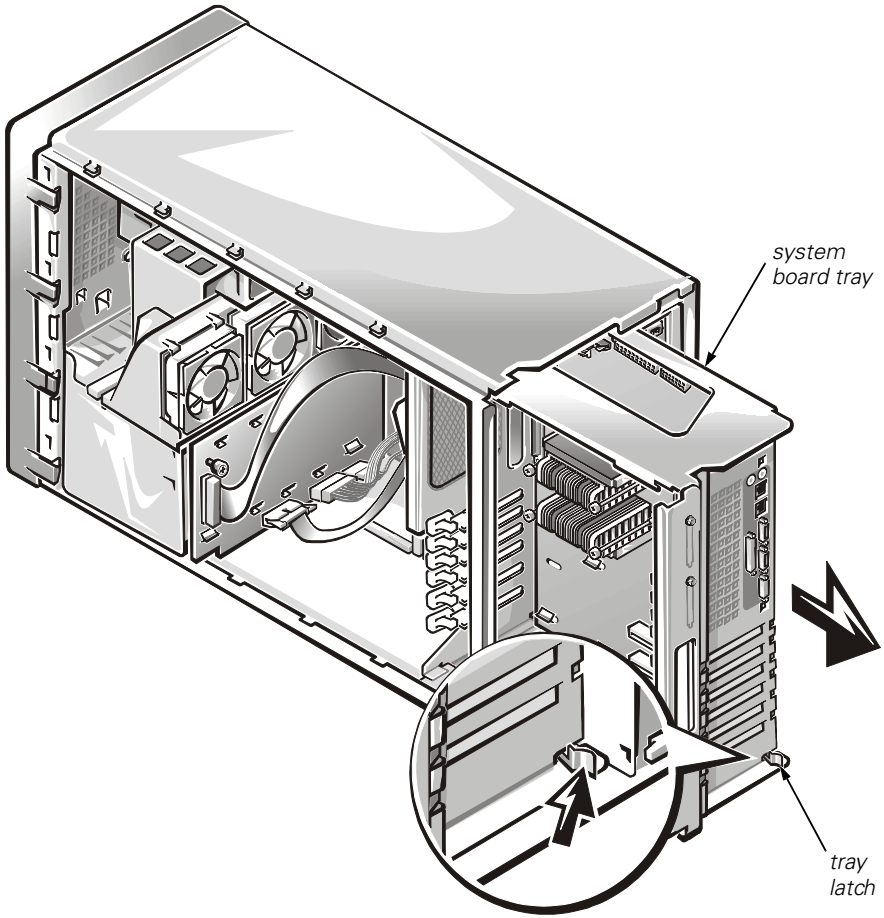


Figure 1-4. Removing the System Board Mounting Tray

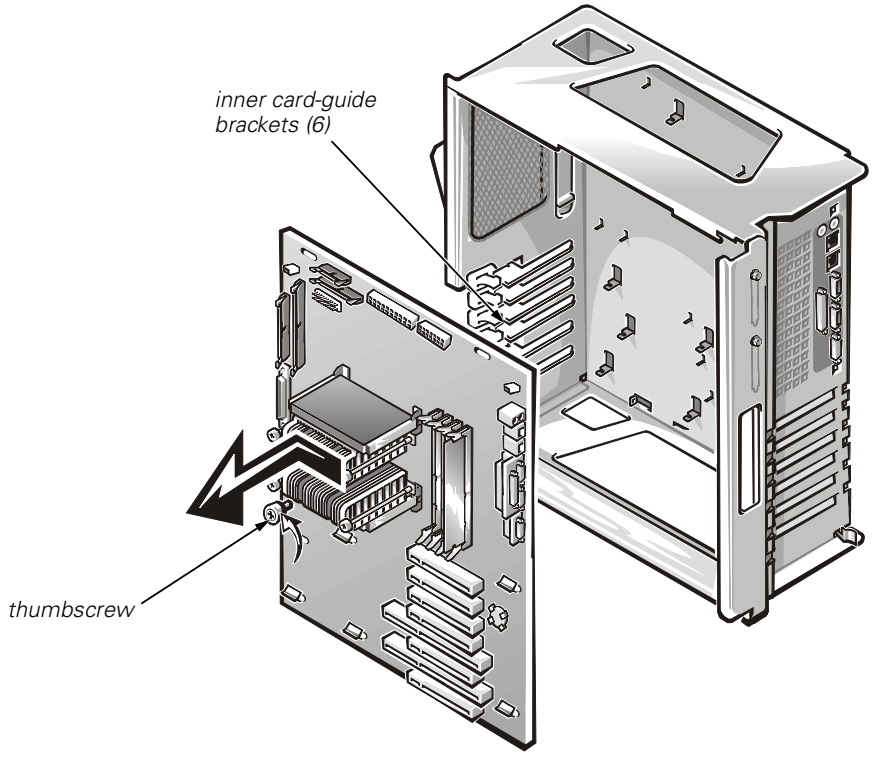


Figure 1-5. Removing the System Board

Installing Upgrade Microprocessors in the PowerEdge 4350

To upgrade to Pentium II or Pentium III microprocessors in the PowerEdge 4350, perform the following steps:

1. Access the system board, which involves the following steps:

   a. Disconnecting power and peripheral cables.

   b. Removing the covers.

   c. Removing the front bezel.

   d. Removing the cooling fan shroud.

   See your system *Installation and Troubleshooting Guide* for specific instructions, if needed.

2. Remove the microprocessors.

   See "Removing the Microprocessors," found later in this document.

3. Remove and replace the guide brackets.

   See "Removing and Replacing the Guide Brackets," found later in this document.

4. Install the upgrade microprocessor.

   See "Installing the Upgrade Microprocessor," found later in this document.

5. Reassemble and check the system.

   See "Reassembling and Checking the System," found later in this document.

Removing the Microprocessors

To remove the current microprocessors from the system board, perform the following steps.

CAUTION: The microprocessor and heat sink assembly can get extremely hot during system operations. Be sure that it has had sufficient time to cool before touching it.

CAUTION: When handling the microprocessor and heat sink assembly, take care to avoid sharp edges on the heat sink.

1. Unscrew and remove the two large thumbscrew retention pins that secure the microprocessor to the system board.

2. Press the microprocessor's release latches inward until they snap into position (see Figure 1-6).

NOTE: Figure 1-6 illustrates the SEC cartridge. The heat sink on the single-edge connector cartridge 2 (SECC2) package is different.

3. Grasp the microprocessor assembly firmly and pull it away from the microprocessor guide bracket assembly.

   You must use up to 15 pounds (lb) of force to disengage the microprocessor from the connector.

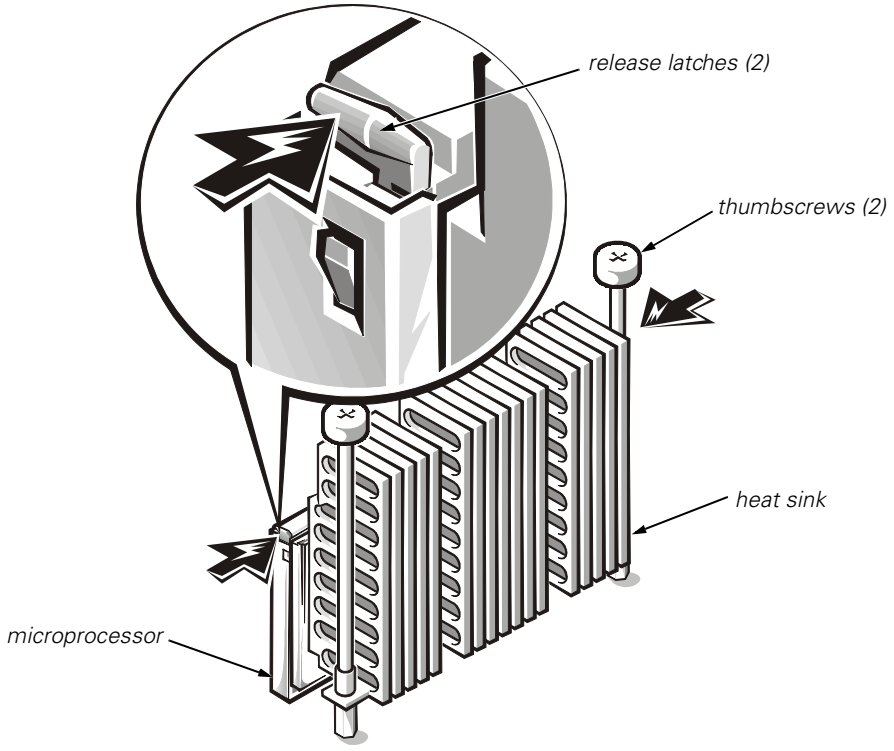


Figure 1-6. Removing the Microprocessor

Removing and Replacing the Guide Brackets

To remove the guide bracket assembly, perform the following steps:

1. Remove any terminator card installed in the guide bracket.
2. Remove any microprocessor assembly installed in the guide bracket.
3. Use a #2 Phillips screwdriver to loosen the four captive nuts that secure the guide bracket assembly to the system board (see Figure 1-7).
4. Lift up the assembly to remove it from the four threaded posts.

To install the new guide bracket assembly, perform the following steps:

1. Position the guide bracket over the four threaded posts (see Figure 1-8).

   You can install the guide bracket only one way (the captive nuts will not align with the threaded posts if installed incorrectly).
2. Tighten the four captive nuts using a #2 Phillips screwdriver.

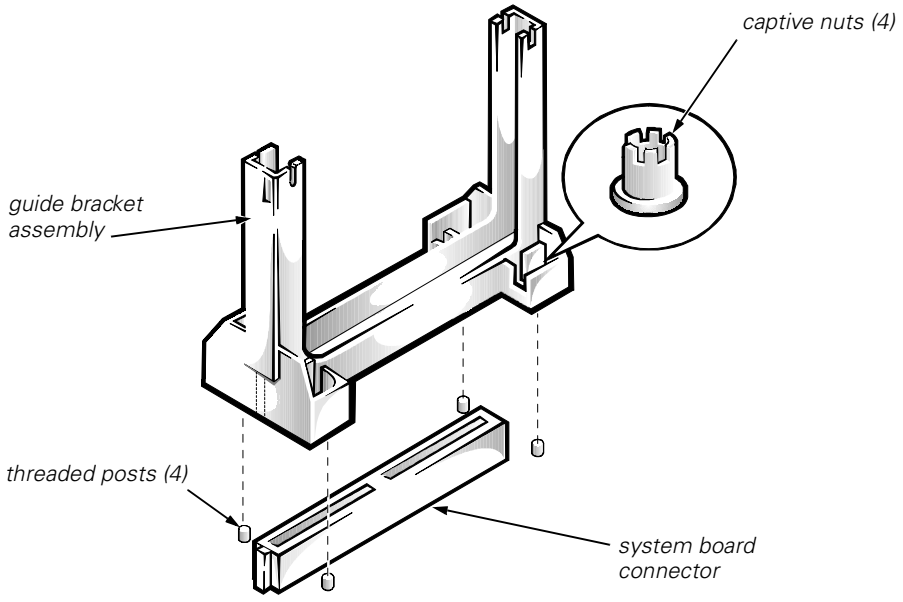


Figure 1-7. Removing the Old Guide Bracket Assembly

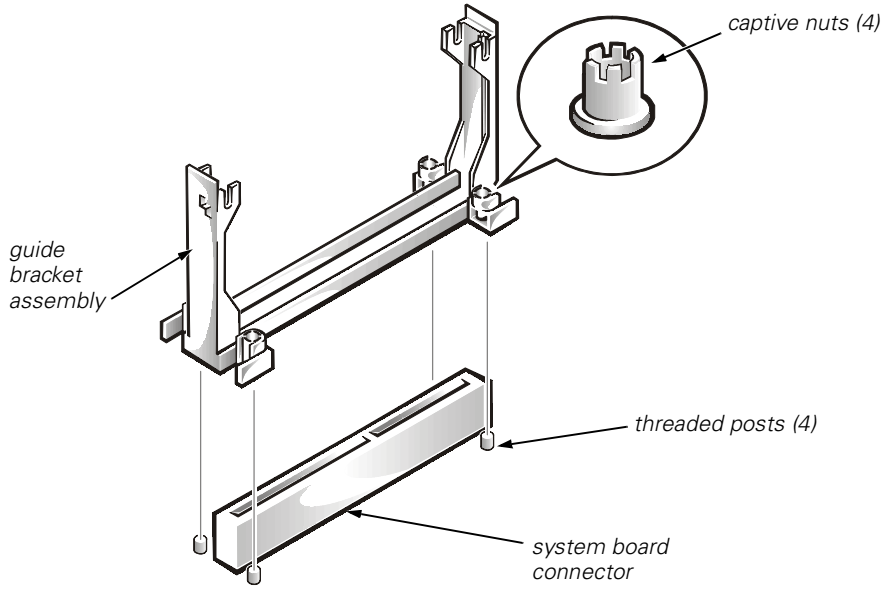


Figure 1-8. Installing the New Guide Bracket Assembly

Removing and Replacing the Cooling Shroud

The plastic cooling shroud inside the system is used to improve airflow over the microprocessors. You may need to remove this shroud to access certain components on the system board.

Removing the Cooling Shroud

To remove the cooling shroud, perform the following steps.

NOTICE: Observe the information given in "Precautionary Measures" and "Before You Begin" to ensure that the system configuration screens are recorded and the system is properly shut down and disconnected from all power sources.

1. Turn off the system, including any attached peripherals, and disconnect the power cable from the electrical outlet.
2. Remove the right-side computer cover.
3. Unscrew and remove the two retention pins (see Figure 1-9).
4. Remove the shroud by lifting the end of the shroud closest to the microprocessor(s) until the opposite end of the shroud disengages from the cooling fan on the system back panel.

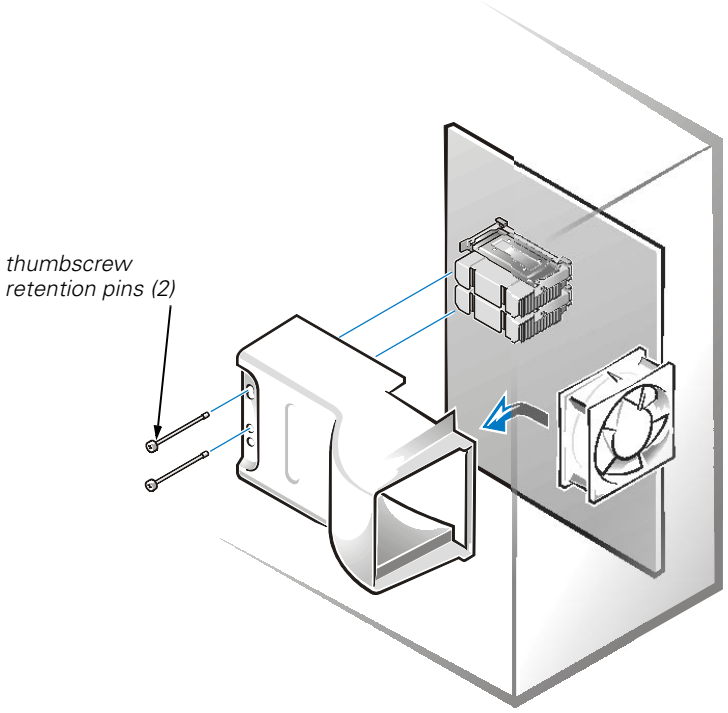


Figure 1-9. Removing the Cooling Shroud

Replacing the Cooling Shroud

To replace the cooling shroud, perform the following steps:

1. Hook the upper edge of the large opening on the end of the cooling shroud over the top of the cooling fan on the system back panel.
2. Lower the other end of the shroud into place over the microprocessor(s).
3. Secure the shroud by reinstalling the two retention pins.

Installing the Upgrade Microprocessor

NOTICE: Do not attempt to operate a system with one Pentium II microprocessor and one Pentium III microprocessor. Damage to one or both of the microprocessors and/or the system board may occur.

NOTICE: All empty microprocessor connectors must be populated with a terminator card. If your system supports more than one microprocessor and you are not installing the maximum number of microprocessors, the remaining microprocessor connectors must have a terminator card.

To install an upgrade microprocessor, perform the following steps:

1. Insert the new microprocessor into the system board connector (see Figure 1-10).

2. Press the microprocessor firmly into its connector until it is fully seated and the latches snap into place.

   You must use up to 25 lb of force to fully seat the microprocessor.

   For Pentium III microprocessors, you do not need to change the jumper settings on the system board.

   For Pentium II microprocessors, set the speed jumper to the speed of the microprocessor.

   Upgrade microprocessors have a heat sink with a notch for engaging a threaded stud on the system board, as shown in Figure 1-10.

3. Repeat steps 1 and 2 for your second upgrade microprocessor or to install a terminator card.

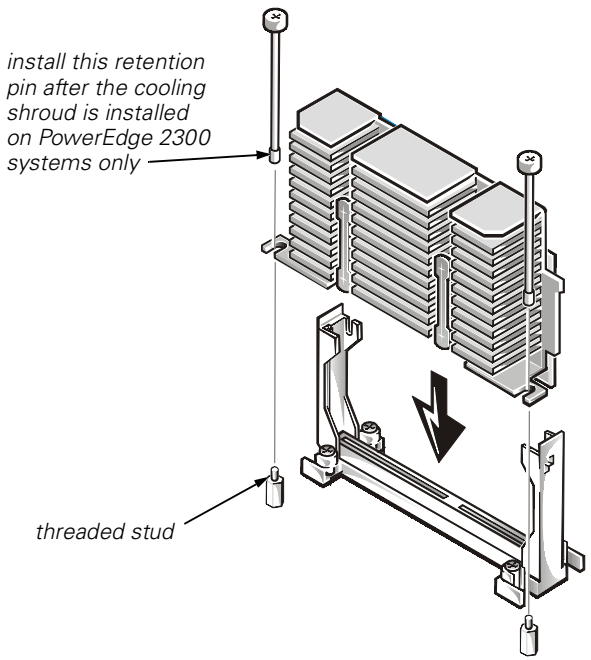


Figure 1-10. Installing the Microprocessor

Installing a New Cooling Shroud

If a cooling shroud came with your microprocessor upgrade kit and your system is a PowerEdge 1300 or PowerEdge 2300, you *must* install the cooling shroud provided in the upgrade kit.

To install a cooling shroud, perform the following steps:

1. Carefully position the shroud into place with the square opening over the bulkhead fan and the top of the shroud's other end resting over the microprocessors, as shown in Figure 1-11.

2. Gently squeeze the tabs to compress the latch as you lower the shroud on the microprocessor's heat sink and allow it to snap securely into place on the heat sink(s).

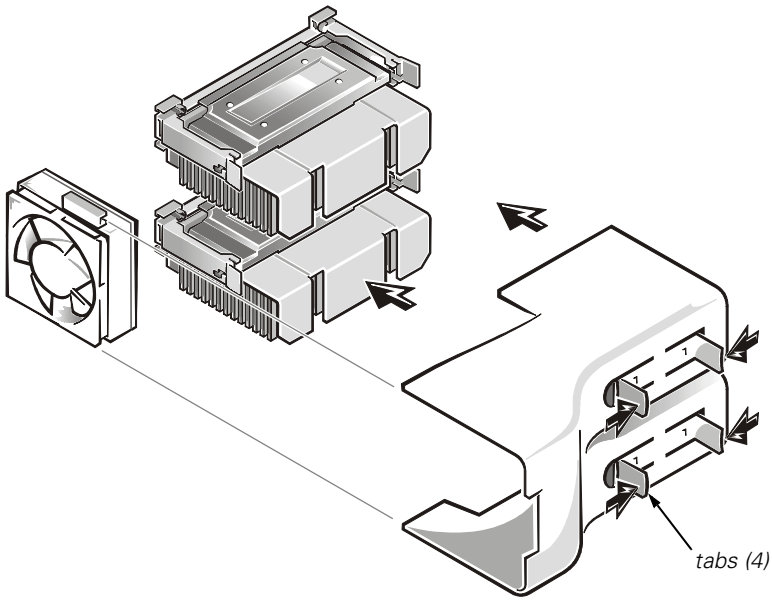


Figure 1-11. Installing a New Cooling Shroud Assembly

Reassembling and Checking the System

To reassemble the system and perform verification checks, perform the following steps.

NOTE: The following procedures apply to PowerEdge 2300, 2400, 4300, and 4350 systems. For PowerEdge 1300 systems, see step 7 under "Installing Upgrade Microprocessors in the PowerEdge 1300," found earlier in this document.

1. Close the computer panel doors (for PowerEdge 4350 systems only) or replace the covers and front bezel, and reconnect your computer and peripherals to their power sources and turn them on.

   As the system boots, it detects the presence of the new microprocessor and automatically changes the system configuration information in the System Setup program. The following message appears:

   ```
   Second processor detected
   ```

2. Enter the System Setup program and confirm that the top line in the system data area correctly identifies the installed microprocessor(s). By default, the serial numbers of Pentium III microprocessors are not displayed. See the procedures in "Using the System Setup Program" in your *User's Guide* for accessing and modifying entries in the System Setup screens.

   Reset the chassis intrusion detector while in the System Setup program by changing **Chassis Intrusion** to **Not Detected**.

   NOTE: If a setup password has been assigned by someone else, contact your network administrator for information on resetting the chassis intrusion detector.

3. Run the Dell Diagnostics to verify that the new microprocessor is operating correctly.

   See your *User's Guide* and your *Installation and Troubleshooting Guide* for additional information on running the Dell Diagnostics and troubleshooting any problems that may occur.