Semester: **IV** Branch: **Computer Science & Engineering**
Subject: **Computer Systems Architecture** Code: **322414 (22)**
Total Theory Periods: **40** Total Tut Periods: **10**
Total Marks in End Semester Exam: **80**
Maximum number of Class Tests to be conducted: **2**

Unit 1: Processor Basics
CPU Organization, Fundamental and features, Data Representation formats, Fixed and Floating
point representation, Instruction Sets, Formats, Types and Programming Considerations.
Unit 2: Datapath Design
Fixed-Point Arithmetic, Combinational ALU and Sequential ALU, Floating point arithmetic and
Advanced topics, Hardware Algorithm – Multiplication, Division.
Unit 3: Control Design
Basic Concepts, Hardwired control, Microprogrammed Control, CPU control unit and Multiplier
control unit, Pipeline Control.

Unit 4: Memory Organization

Memory device characteristics, RAM technology and Serial access memories technology,
Multilevel memory systems, Address translation and Memory allocation systems, Caches
memory.

Unit 5: System Organization

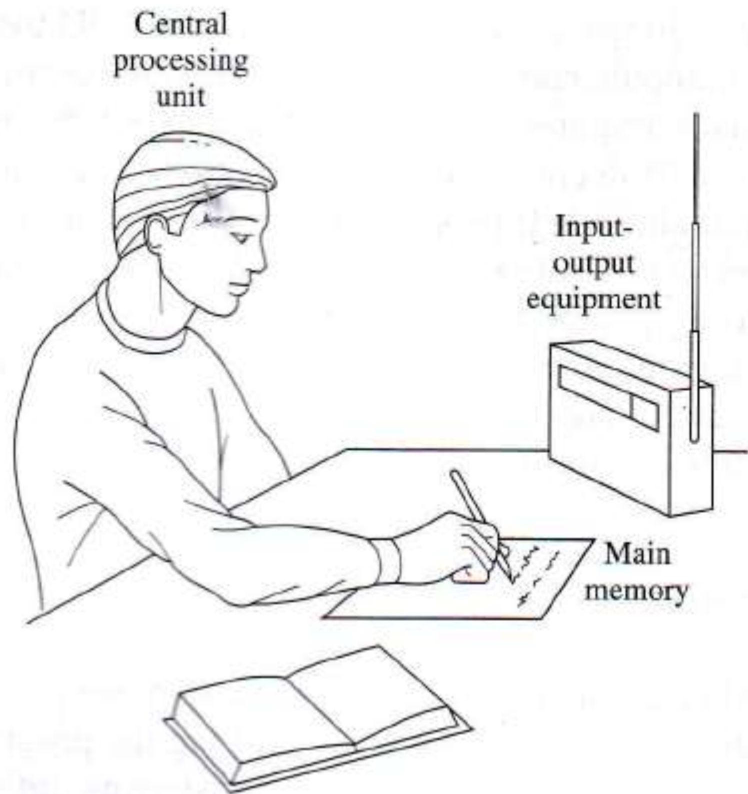
Programmed I/O , DMA, Interrupts and IO Processors, Processor-level Parallelism, Multiprocessor and Fault tolerance system.

Name of Text Books

1. Computer Architecture and organization – John P Hayes, McGraw Hill Publication
2 Computer Organizations and Design- P. Pal Chaudhari, Prentice-Hall of India

Name of reference Books:

1. Computer System Architecture - M. Morris Mano, PHI.
2. Computer Organization and Architecture- William Stallings, Prentice-Hall of India
3. Architecture of Computer Hardware and System Software: An Information Technology Approach,
3rd Edition (Illustrated) – Iry Englander, John Wiley & Sons Inc
4 Structured Computer Organization Andrew S Tanenbaum, Prentice-Hall of India
5 Computer Systems Organization & Architecture – John D Carpinelli, Addison-Wesle


Central
processing
unit
Input-
output
equipment
Main
memory

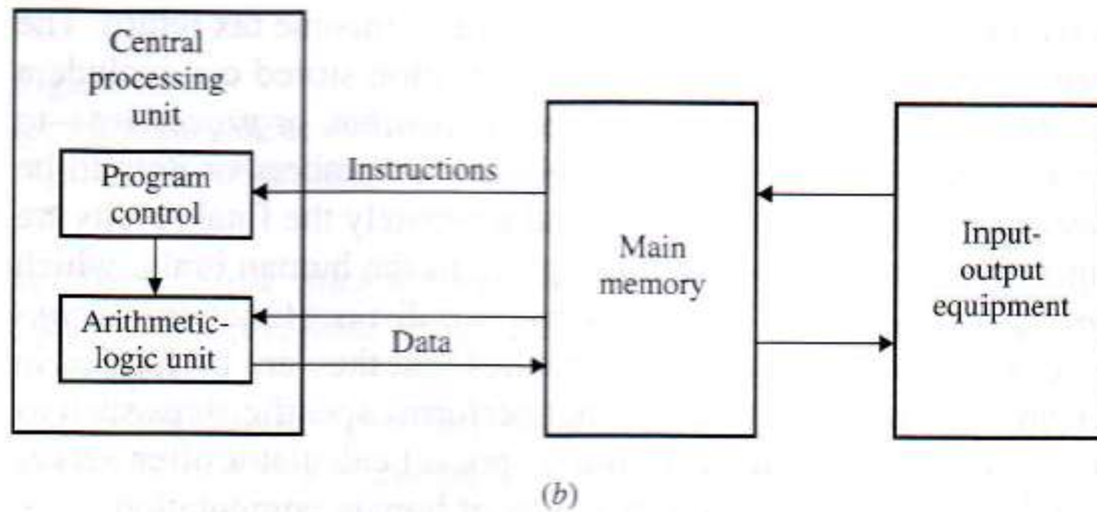


Figure 1.2
Main components of (*a*) human computation and (*b*) machine computation.

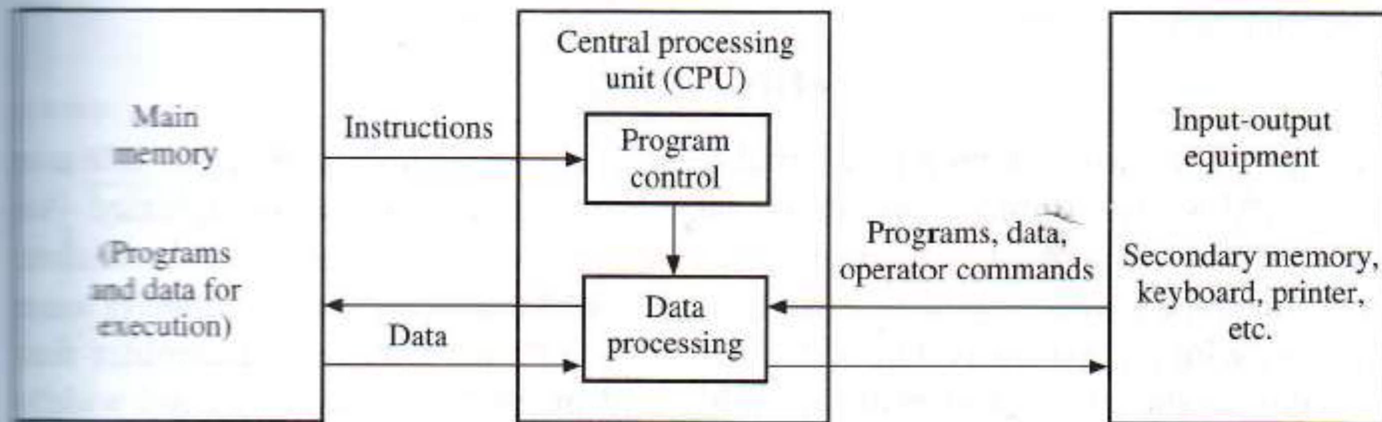
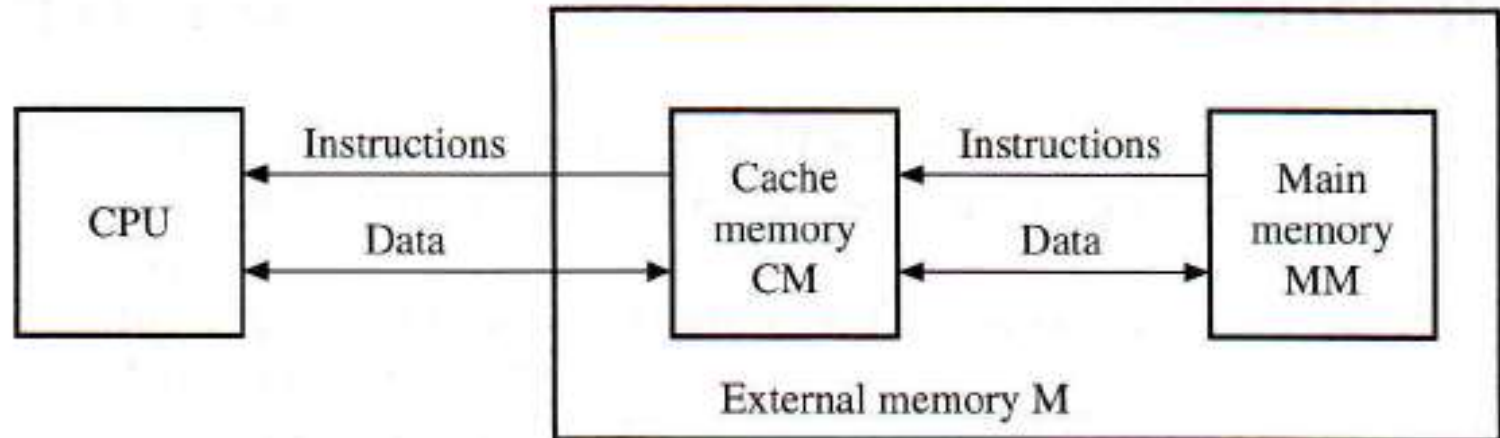


Figure 1.11
Organization of a first-generation computer.

CPU ORGANIZATION

The primary function of the CPU and other instruction-set processors is to execute sequences of instructions, that is, programs, which are stored in an external main memory. Program execution is therefore carried out as follows:

1. The CPU transfers instructions and, when necessary, their input data (operands) from main memory to registers in the CPU.
2. The CPU executes the instructions in their stored sequence except when the execution sequence is explicitly altered by a branch instruction.
3. When necessary, the CPU transfers output data (results) from the CPU registers to main memory.



Processor-memory communication with a cache.

External communication.

To remedy this situation, many computers have a cache memory CM positioned between the CPU and main memory. The cache CM is smaller and faster than main memory and may reside, wholly or in part, on the same chip as the CPU. It typically permits the CPU to perform a memory load or store operation in a single clock cycle, whereas a memory access that bypasses the cache and is handled by main memory takes many clock cycles. The cache is designed to be transparent to the CPU's instructions, which "see" the cache and main memory as forming a single, seamless memory space consisting of 2^m addressable storage locations M(0), M(1), …, M(2^m–1). In this chapter we will take this viewpoint and use M to refer to the *external memory*, whether or not a cache is present. A specific memory location in M with address *adr* is referred to as M(*adr*) or simply as *adr*. When necessary, we will use MM to distinguish the main memory from the cache memory CM, as in Figure 3.1*b*. The structure of caches and their interactions with main memory are further studied in Chapter 6.

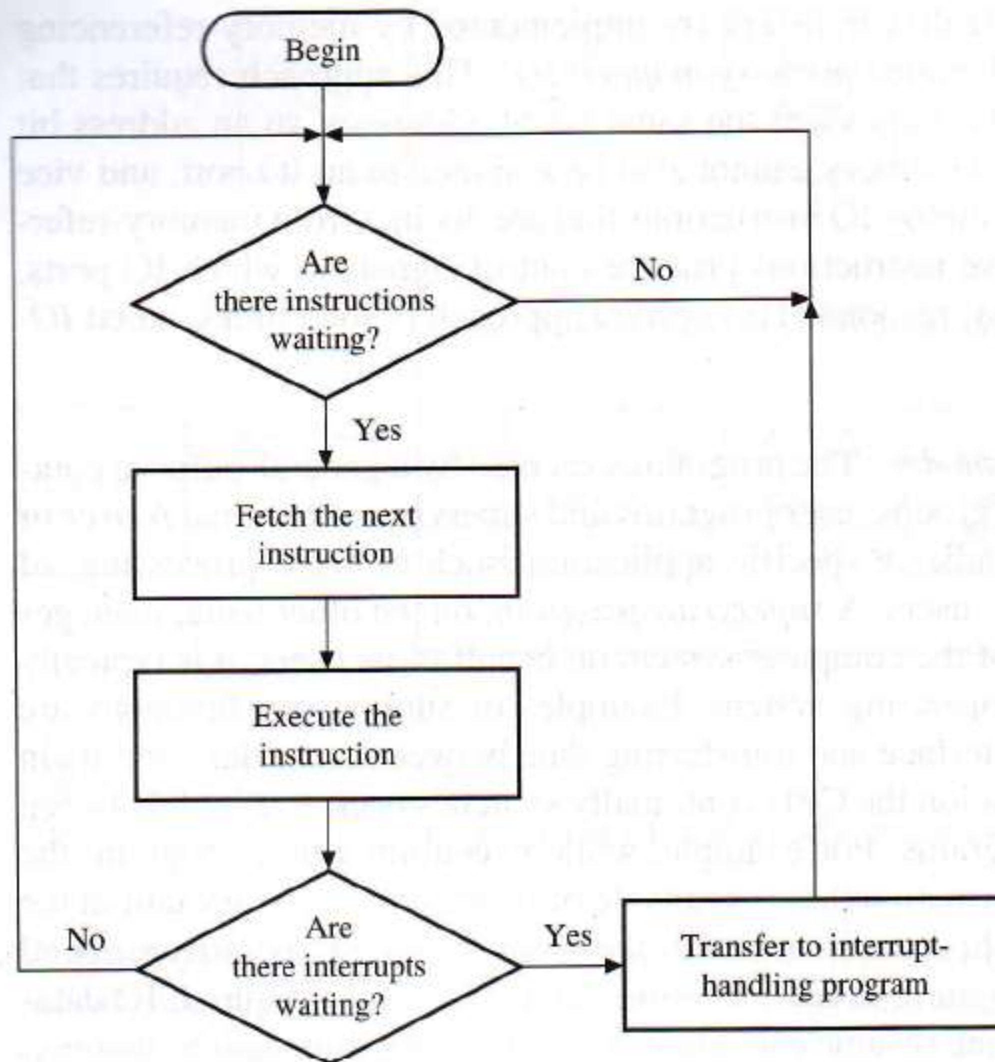


Figure 3.2
Overview of CPU behavior.

Accumulator-based CPU. Despite the improvements in IC technology over the years, CPU design continues to be based on the premise that the CPU should be as fast as the available technology and overall design requirements allow. Since cost generally increases with circuit complexity, the number of components in the CPU must be kept relatively small. The CPU organization proposed by von Neumann and his colleagues for the IAS computer (section 1.2.2) is the basis for most subsequent designs. It comprises a small set of registers and the circuits needed to execute a functionally complete set of instructions. In many early designs, one of the CPU registers, the *accumulator,*[1] played a central role, being used to store an input or output operand (result) in the execution of many instructions.

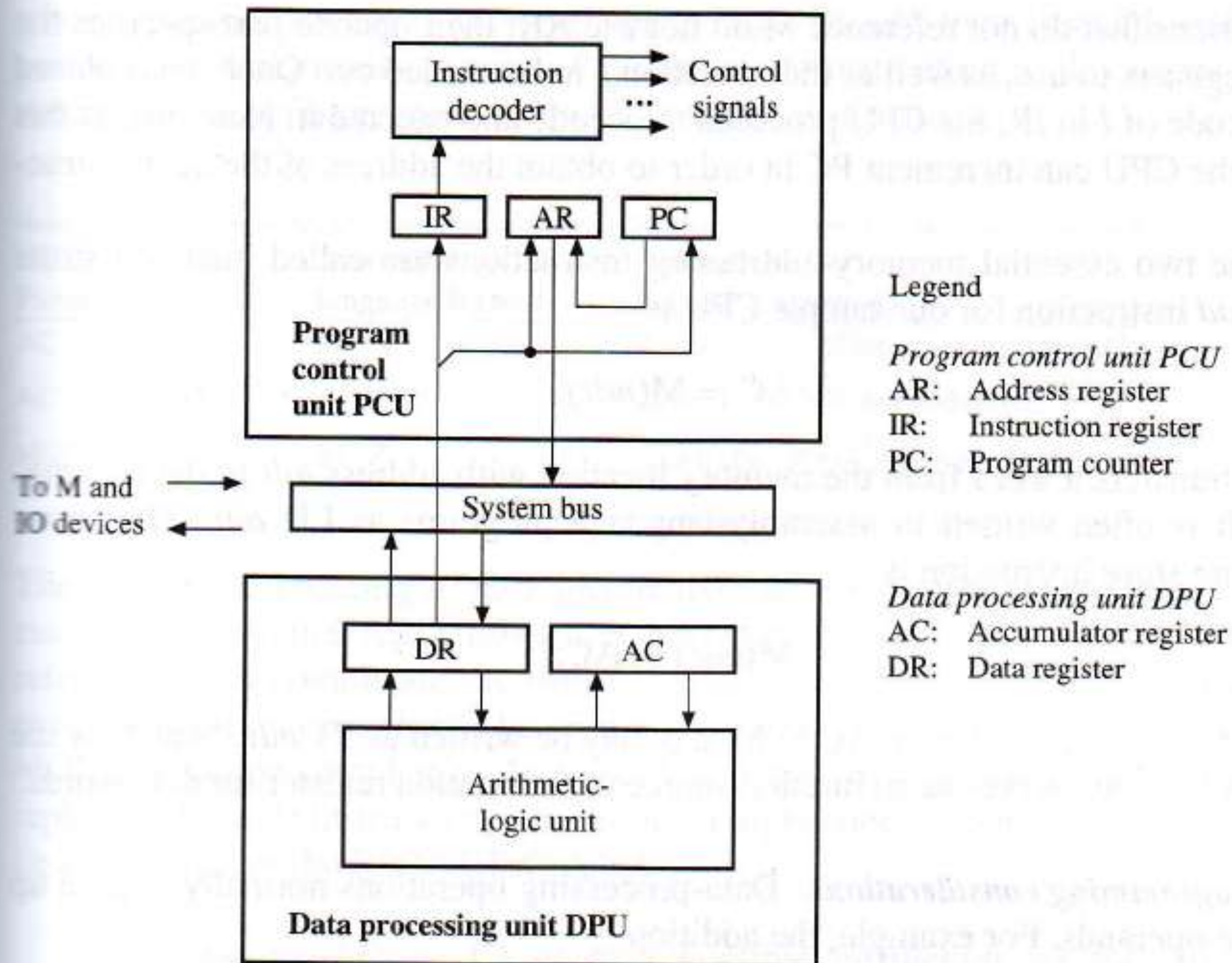


Figure 3.3
A small accumulator-based CPU.

Programming considerations. Data-processing operations normally require up to three operands. For example, the addition

$$Z := X + Y \tag{3.2}$$

has three distinct operands *X*, *Y*, and *Z*. The accumulator-based CPU of Figure 3.3 supports only *single-address* instructions, that is, instructions with one explicit memory address. However, AC and DR can serve as *implicit* operand locations so that multioperand operations can be implemented by executing several instructions in sequence. For example, a program to implement (3.2), assuming that *X*, *Y*, and *Z* all refer to data words in M, can take the following form:

HDL format	Assembly-language format	Narrative format (comment)
AC := M(*X*);	LD *X*	Load *X* from M into accumulator AC.
DR := AC;	MOV DR, AC	Move contents of AC to DR.
AC := M(*Y*);	LD *Y*	Load *Y* into accumulator AC.
AC := AC + DR;	ADD	Add DR to AC.
M(Z) := AC;	ST Z	Store contents of AC in M.

Type	Instruction	HDL format	Assembly-language format	Narrative format (comment)
Data transfer	Load	AC := M(*X*)	LD X	Load *X* from M into AC.
	Store	M(*X*) := AC	ST X	Store contents of AC in M as *X*.
	Move register	DR := AC	MOV DR, AC	Copy contents of AC to DR.
	Move register	AC := DR	MOV AC, DR	Copy contents of DR to AC.
Data processing	Add	AC := AC + DR	ADD	Add DR to AC.
	Subtract	AC := AC – DR	SUB	Subtract DR from AC.
	And	AC := AC ***and*** DR	AND	And bitwise DR to AC.
	Not	AC := ***not*** AC	NOT	Complement contents of AC
Program control	Branch	PC := M(*adr*)	BRA adr	Jump to instruction with address *adr*.
	Branch zero	**if** AC = 0 **then** PC := M(*adr*)	BZ adr	Jump to instruction *adr* if AC = 0.

Figure 3.4
Instruction set for the CPU of Figure 3.3.

3.1.2 Additional Features

Next we examine some more advanced features of CPUs and look at representative commercial microprocessors of the RISC and CISC types.

Architecture extensions. There are many ways in which the basic design of Figure 3.3 can be improved. Most recent CPUs contain the following extensions, which significantly improve their performance and ease of programming.

- *Multipurpose register set for storing data and addresses*: These replace the accumulator AC and the auxiliary registers DR and AR of our basic CPU. The resulting CPU is sometimes said to have the *general register organization* exemplified by the third-generation IBM System/360-370 (Figure 1.17), which has 32 such registers. The set of general registers is now usually referred to as a *register file*.
- *Additional data, instruction, and address types*: Most CPUs have instructions to handle data and addresses with several different word sizes and formats. Although some microprocessors have only add and subtract instructions in the arithmetic category, relatively little extra circuitry is required for (fixed-point) multiply and divide instructions, which simplify many programming tasks. Call and return instructions also simplify program design.
- *Register to indicate computation status*: A *status register* (also called a *condition code* or *flag* register) indicates infrequent or *exceptional conditions* resulting from the instruction execution. Examples are the appearance of an all-zero result or an invalid instruction like divide by zero. A status register can also indicate the user and supervisor states. Conditional branch instructions can test the status register, which simplifies the programming of conditional actions.

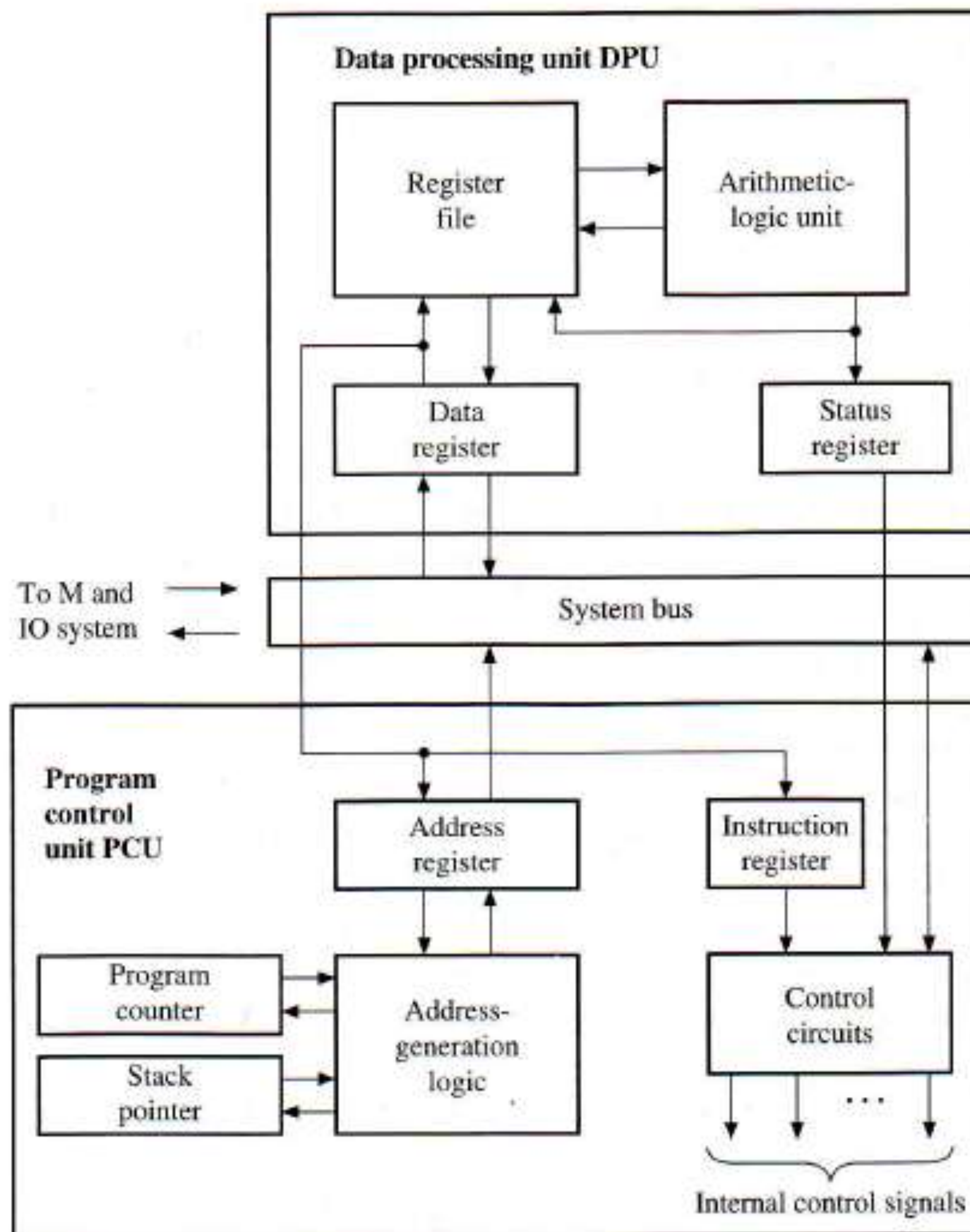


Figure 3.7
A typical CPU with the general register organization.

Coprocessors. The built-in instruction repertoire of the 68020 includes fixed-point multiplication and division and stack-based instructions for transferring control between programs. Hardware-implemented floating-point instructions are not available directly; however, they are provided indirectly by means of an auxiliary IC, the 68881 floating-point coprocessor. (The ARM6 also has provisions for external coprocessors.) In general, a *coprocessor P* is a specialized instruction execution unit that can be coupled to a microprocessor so that instructions to be executed by *P* can be included in programs fetched by the microprocessor. Thus the coprocessor serves as an extension to the microprocessor and forms part of the CPU as indicated in Figure 3.14.

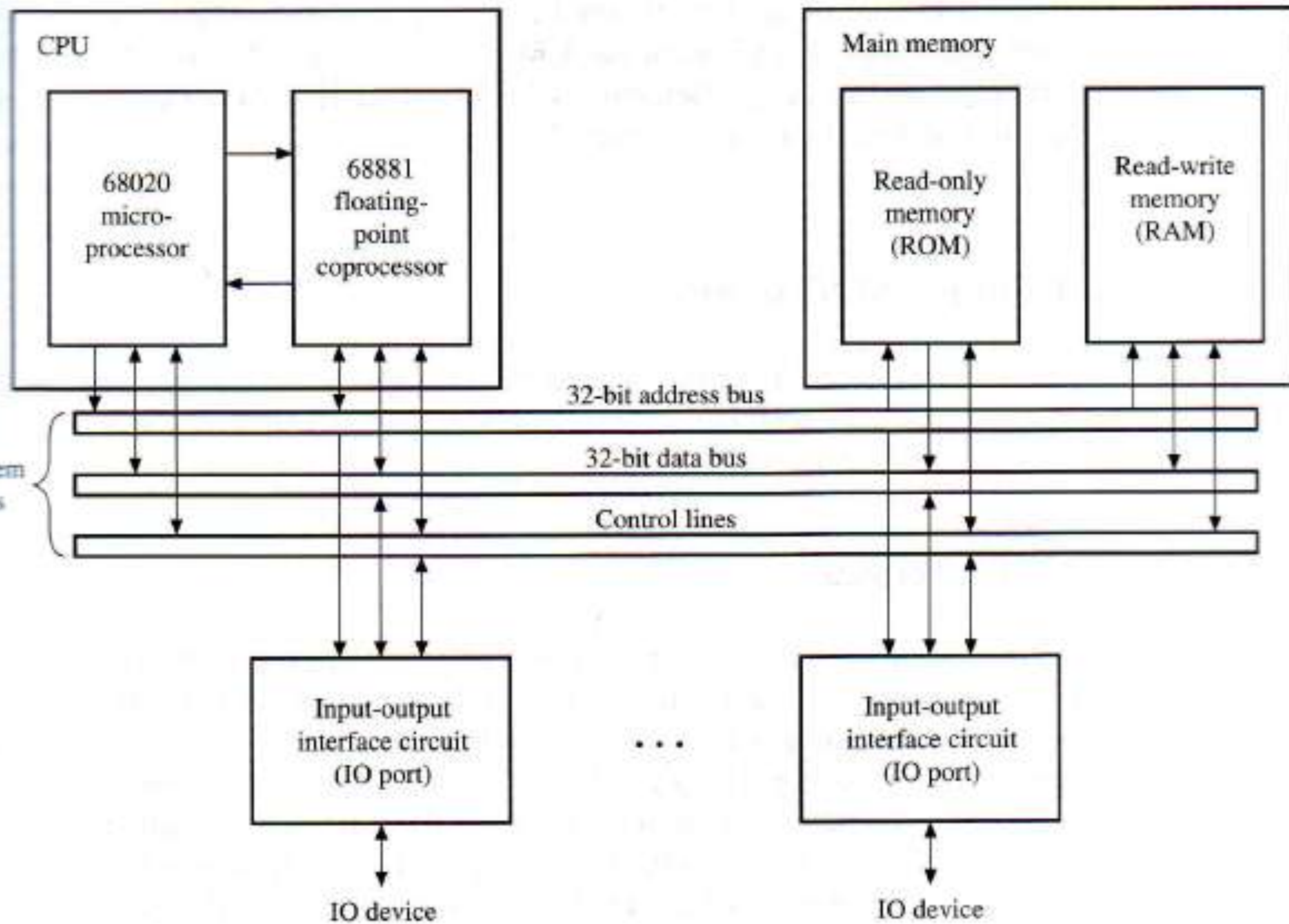


Figure 3.14
68020-based microcomputer with floating-point coprocessor.

INSTRUCTION SETS

Next we turn to the representation, selection, and application of instruction sets. This topic embraces opcode and operand formats, the design of the instruction types to include in a processor's instruction set, and the use of instructions in executable programs.

3.3.1 Instruction Formats

The purpose of an instruction is to specify both an operation to be carried out by a CPU or other processor and the set of operands or data to be used in the operation. The operands include the input data or arguments of the operation and the results that are produced.

Introduction. Most instructions specify a register-transfer operation of the form

$$X_1 := op(X_1, X_2, \ldots, X_n)$$

which applies the operation op to n operands $X_1, X_2, \ldots, X_n$, where n ranges from zero to four or so. We can write the same instruction in the assembly-language notation

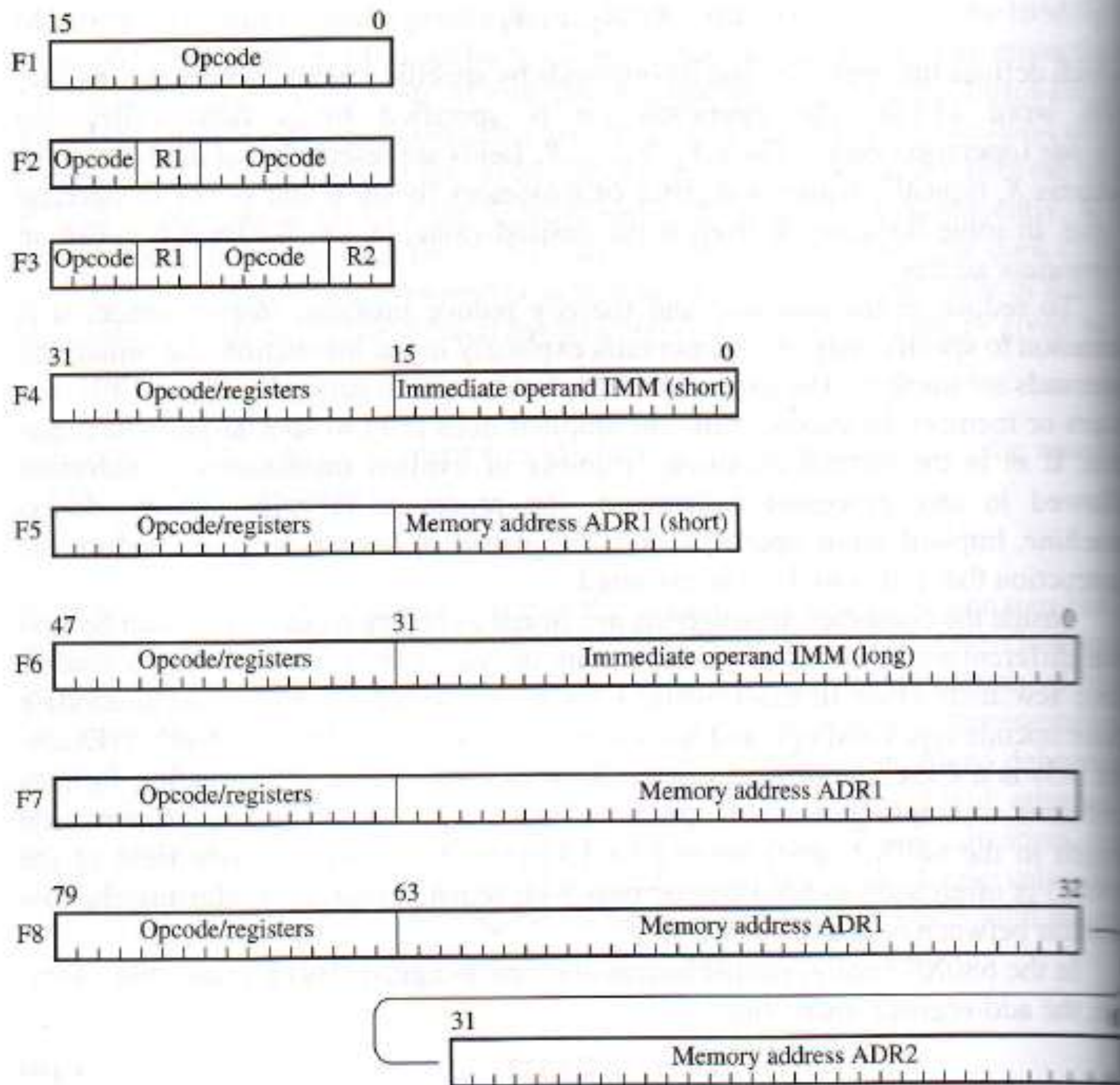


Figure 3.27
A selection of instruction formats of the Motorola 680X0.

Instructions are conveniently divided into the following five types:

1. *Data-transfer* instructions, which copy information from one location to another either in the processor's internal register set or in the external main memory.
2. *Arithmetic* instructions, which perform operations on numerical data.
3. *Logical* instructions, which include Boolean and other nonnumerical operations.
4. *Program-control* instructions, such as branch instructions, which change the sequence in which programs are executed.
5. *Input-output* (IO) instructions, which cause information to be transferred between the processor or its main memory and external IO devices.

Type	Operation name(s)	Description
Data transfer	MOVE	Copy word or block from source to destination.
	LOAD	Copy word from memory to processor register.
	STORE	Copy word from processor register to memory.
	SWAP (EXCHANGE)	Swap contents of source and destination.
	CLEAR	Transfer word of 0s to destination.
	SET	Transfer word of 1s to destination.
	PUSH	Transfer word from source to top of stack.
	POP	Transfer word from top of stack to destination.
Arithmetic	ADD	Compute sum of two operands.
	ADD WITH CARRY	Compute sum of two operands and a carry bit.
	SUBTRACT	Compute difference of two operands.
	MULTIPLY	Compute product of two operands.
	DIVIDE	Compute quotient (and remainder) of two operands.
	MULITPLY AND ADD	Compute product of two operands; add it to a third operand.
	ABSOLUTE	Replace operand by its absolute value.
	NEGATE	Change sign of operand.
	INCREMENT	Add 1 to operand.
	DECREMENT	Subtract 1 from operand.
	ARITHMETIC SHIFT	Shift operand left (right) with sign extension.
Logical	AND	
	OR	Perform the specified logical operation bitwise.
	NOT	
	EXCLUSIVE-OR	
	LOGICAL SHIFT	Shift operand left (right) introducing 0s at end.
	ROTATE	Left- (right-) shift operand around closed path.
	CONVERT (EDIT)	Change data format, for example, from binary to decimal.

Program control	JUMP (BRANCH)	Unconditional transfer; load PC with specified address.
	JUMP CONDITIONAL	Test specified conditions; if true, load PC with specified address.
	JUMP TO SUBROUTINE (BRANCH-AND-LINK)	Place current program control information including PC in known location, for example, top of stack; jump to specified address.
	RETURN	Restore current program control information including PC from known location, for example, from top of stack.
	EXECUTE	Fetch operand from specified location and execute as instruction; note that PC is not modified.
	SKIP CONDITIONAL	Test specified condition; if true, increment PC to skip next instruction.
	TRAP (SOFTWARE INTERRUPT)	Enter supervisor mode.
	TEST	Test specified condition; set flag(s) based on outcome.
	COMPARE	Make logical or arithmetic comparison of two or more operands; set flag(s) based on outcome.

Figure 3.34
List of common instruction types.

Type	Operation name(s)	Description
Program control	SET CONTROL VARIABLES	Large class of instructions to set controls for protection purposes, interrupt handling, timer control, and so forth (often privileged).
	WAIT (HOLD)	Stop program execution; test a specified condition continuously; when the condition is satisified, resume instruction execution.
	NO OPERATION	No operation is performed, but program execution continues.
Input-output	INPUT (READ)	Copy data from specified IO port to destination, for example, output contents of a memory location or processor register.
	OUTPUT (WRITE)	Copy data from specified source to IO port.
	START IO	Transfer instuctions to IOP to initiate an IO operation.
	TEST IO	Transfer status information from IO system to specified destination.
	HALT IO	Transfer instructions to IOP to terminate an IO operation.

Unit - 02

Principles of Computer design

Register Organization

Digital System Overview

- Each module is built from digital components
  - Registers
  - Decoders
  - Arithmetic elements
  - Control logic
- Modules connected by common data and control paths
- Collection of modules is a digital system

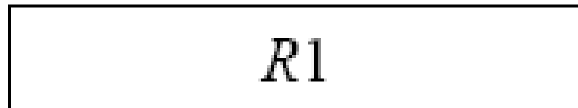
Internal Hardware Organization

- Can be defined by specifying
  - Set of registers and their functions
  - Sequence of microoperations performed on register data
  - Control that initiates the sequence of microoperations
- Can use words to express sequence of microoperations, but it's better to use a notation and symbols
  - Register Transfer Language

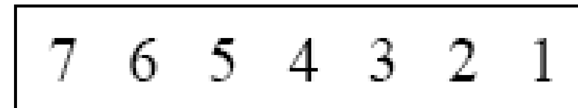
Registers

- Capital letter sometimes followed by a number
- *MAR* – memory address register
- *PC* – program counter
- *IR* – instruction register
- *R*1, *R*2 – processor register 1, process register 2
- Each flip-flop in a *n*-bit register is numbered from *n*-1 to 0 from left to right

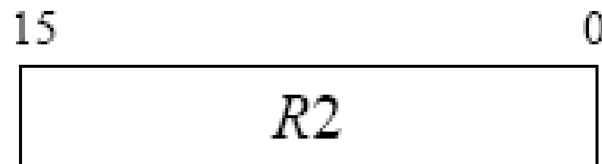
Register Block Diagram



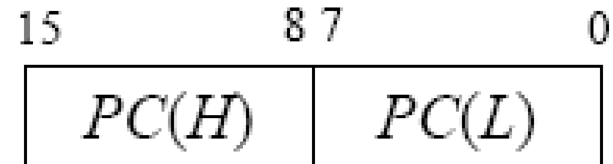
Register $R1$



Showing individual bits



Bit numbering



Divided into two parts

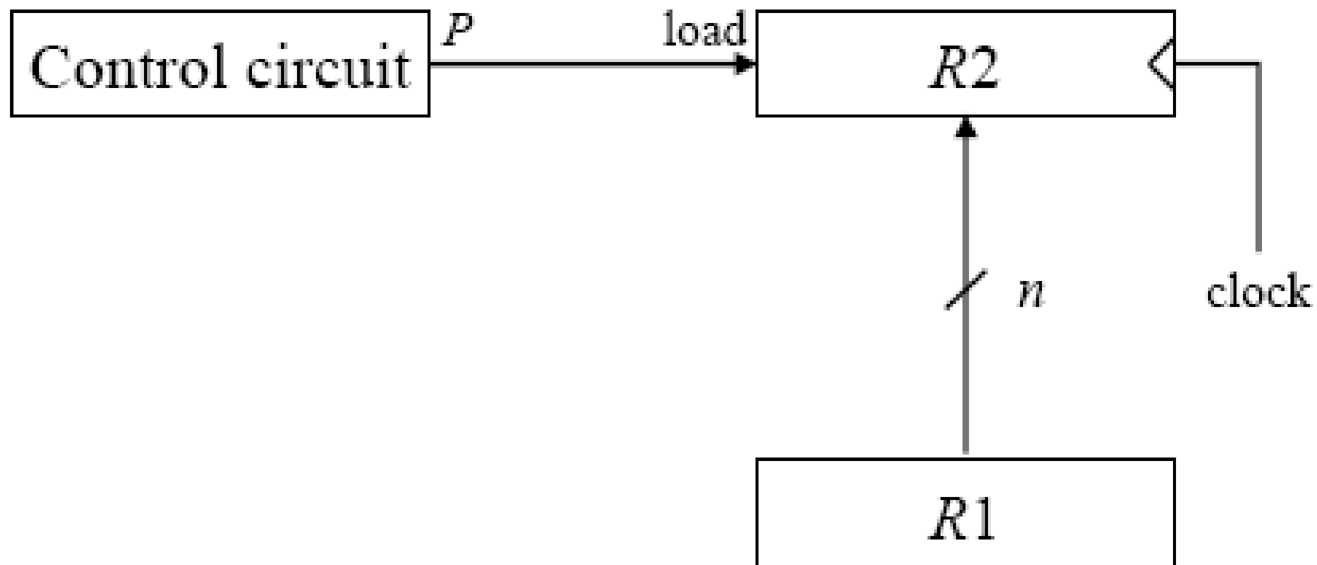
Register Transfer

- Use replacement operator: $\leftarrow$
- $R2 \leftarrow R1$
  - Transfer contents of $R1$ to $R2$ at next clock pulse
  - Contents of $R1$ unchanged
- Circuits are available from outputs of source register to inputs of destination register
- Destination register has parallel load capability

Control Function

- Control condition
  - Boolean variable (equal to 0/false or 1/true)
  - Terminated with colon
  - Prefix to register transfer statement
- $P: R2 \leftarrow R1$
  - Transfer happens at next clock pulse while $P = 1$

P: $R2 \leftarrow R1$ Block Diagram



2nd Register Transfer Example

- $T: R2 \leftarrow R1, R1 \leftarrow R2$
  - Comma used to separate multiple register transfers that happen at the same time
  - This register swap is possible using edge-triggered flip-flops

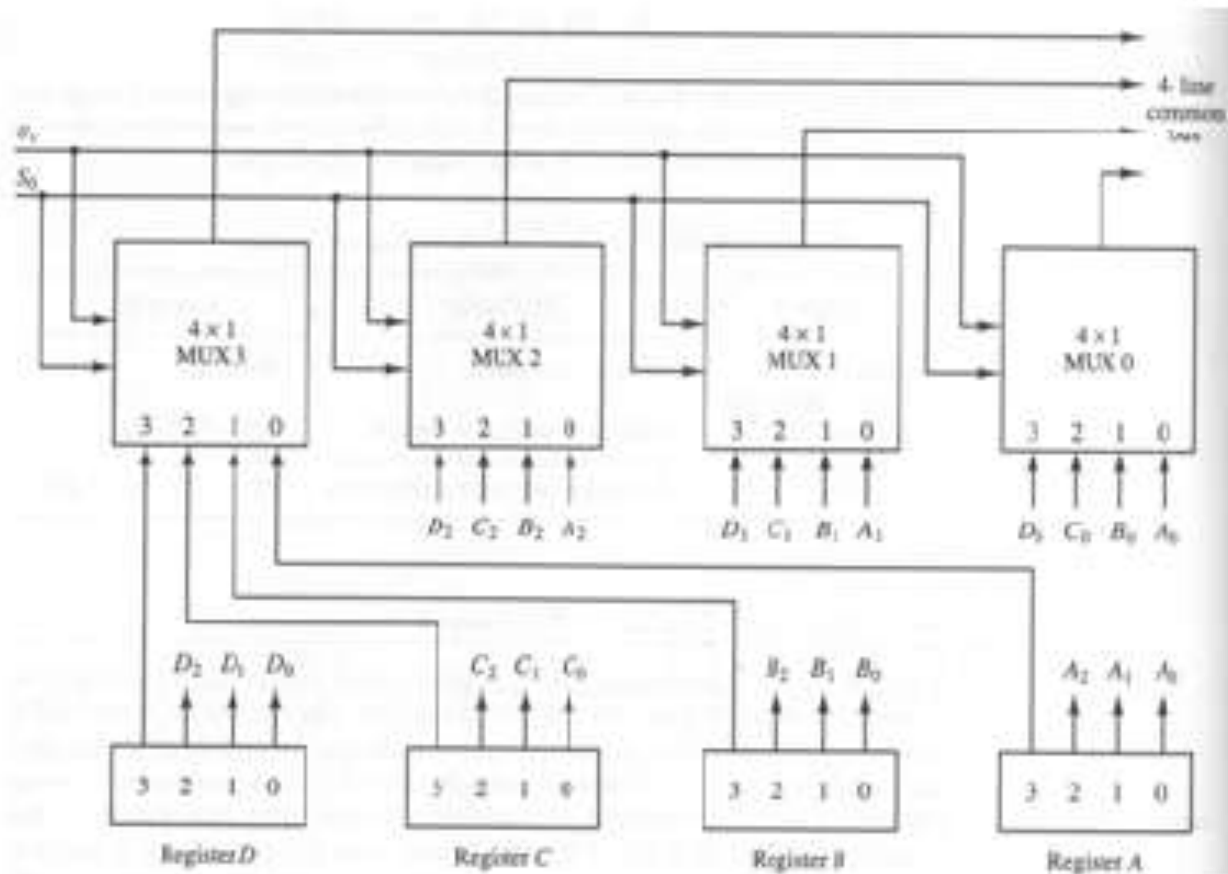
Using Parentheses

- $R1(8\text{–}15)$ or $R1(H)$
- if $R1$ is 16-bit register
  - Indicates high-order byte, leftmost 8 bits

Bus Transfers

- Wiring each register to every other register requires an excessive number of data lines
- Use common bus system instead
  - One data line for each register bit
  - Control signal lines used for register selection
  - Use multiplexers to select source register to put data on common bus
  - Activate load control of destination register to load data from common bus

Bus System For 4 Registers



Bus Details

- Multiplex k registers with n bits each to build n-line common bus
  - Need n multiplexers
  - Each multiplexer has k input lines
    - One for each register
- $BUS \leftarrow C, R1 \leftarrow BUS$ can be rewritten as $R1 \leftarrow C$

Memory Transfer

- Read: from memory to outside world
- Write: from outside world into memory
- Memory word is called M
- Address Register is called AR
- Data Register is called DR
- Read: $DR \leftarrow M[AR]$
- Write: $M[AR] \leftarrow DR$

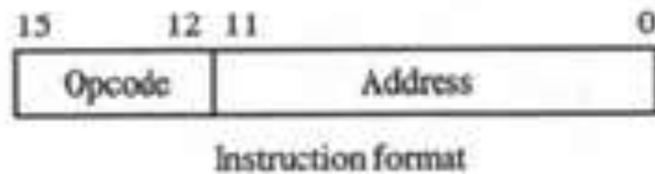
Instruction Code

- Computer instruction is binary code that specifies a sequence of microoperations
- Operation code + Address
  - Op code must have n bits for $\leq 2^n$ operations
  - Op code sometimes called a macrooperation
  - Address is register or memory location
    - Memory location is operand address
- Shorten "instruction code" to "instruction"
- Instructions and data in memory

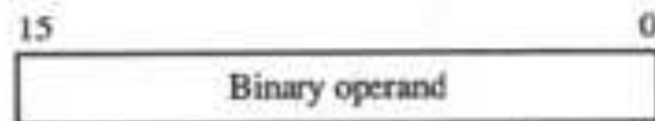
Stored Program Organization

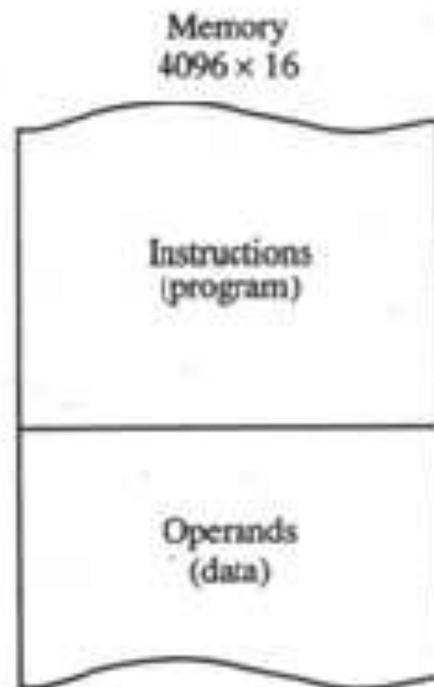
- One processor register
  - *AC* – accumulator
- Instruction format
  - 4-bit op code
  - 12-bit address (for $2^{12} = 4096$ memory words)
- Instruction execution cycle
  - Read 16-bit instruction from memory
  - Use 12-bit address to fetch operand from memory
  - Execute 4-bit op code

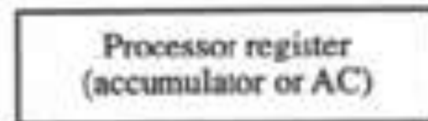
Stored Program Organization



Instruction format



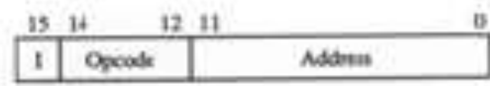


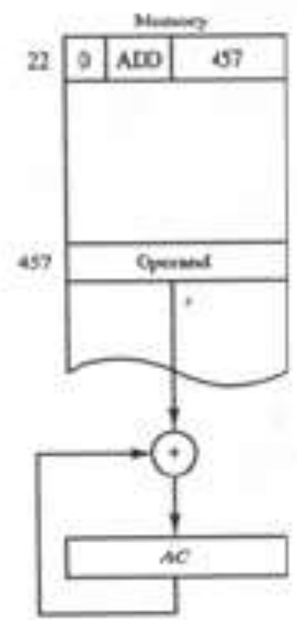


Address Types

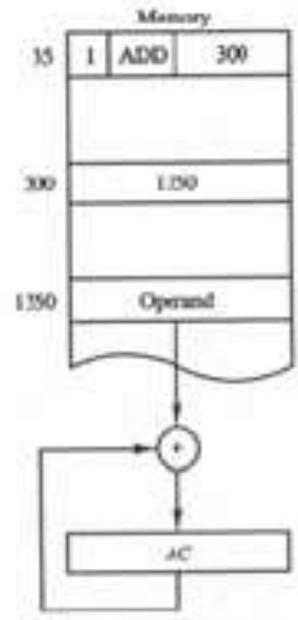
- 12-bit instruction address
  - Immediate
    - Actual data value
  - Direct
    - Memory address where data (operand) resides
  - Indirect
    - Memory address where memory address of data (operand) resides
- Effective address is the address of the operand
- Lead bit of instruction used as indirect flag

Direct / Indirect Address





(b) Direct address



(c) Indirect address

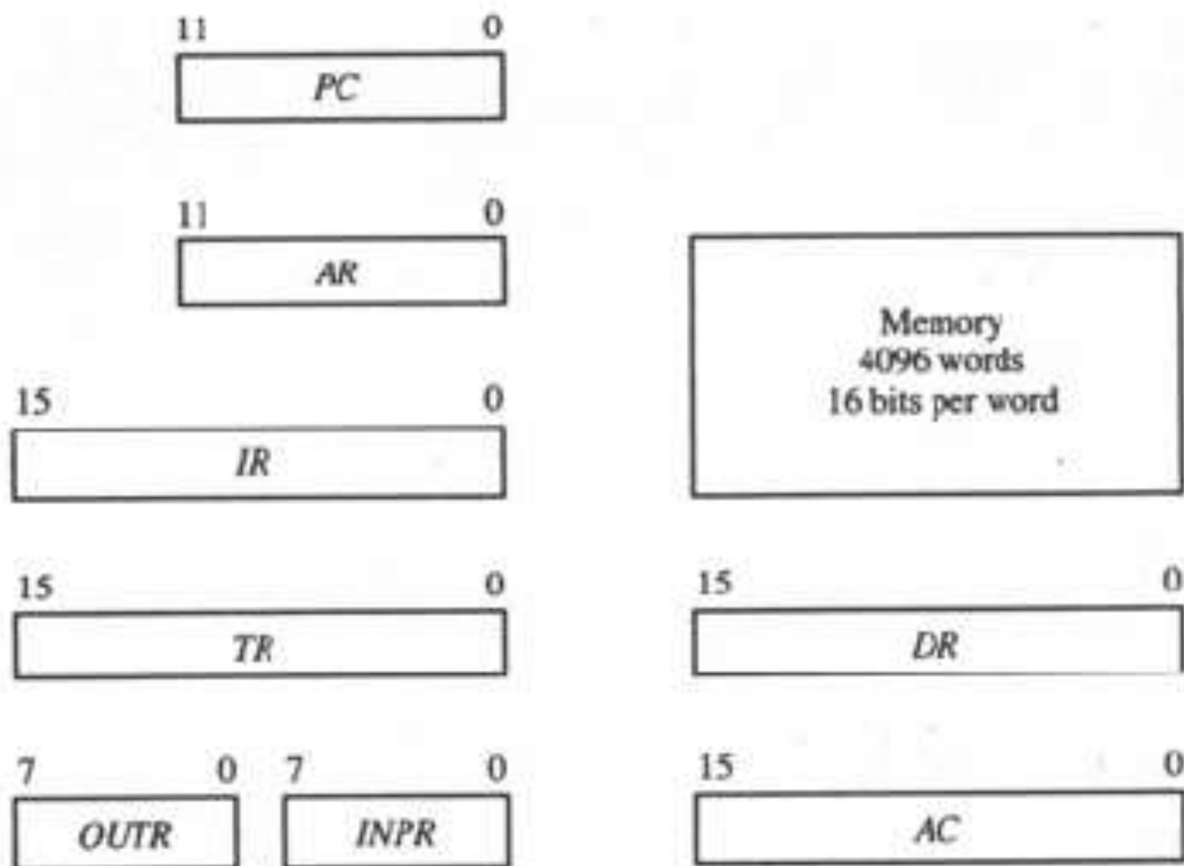
Basic Computer Registers

Register symbol	Number of bits	Register name	Function
DR	16	Data register	Holds memory operand
AR	12	Address register	Holds address for memory
AC	16	Accumulator	Processor register
IR	16	Instruction register	Holds instruction code
PC	12	Program counter	Holds address of instruction
TR	16	Temporary register	Holds temporary data
INPR	8	Input register	Holds input character
OUTR	8	Output register	Holds output character

Program Counter (*PC*)

- Holds memory address of next instruction
- Next instruction is fetched after current instruction completes execution cycle
- *PC* is incremented right after instruction is fetched from memory
- *PC* value can be replaced by new address when executing a branch instruction

Registers + Memory Layout



Register Control Inputs

- Load (LD)
- Increment (INR)
- Clear (CLR)

Common Bus

- Connects registers and memory
- Specific output selected by $S_2S_1S_0$
  - When register has < 16 bits, high-order bus bits are set to 0
- Register with LD enabled reads data from bus
- Memory with Write enabled reads bus
- Memory with Read enabled puts data on bus
  - When $S_2S_1S_0 = 111$

Address Register (*AR*)

- Always used to specify address within memory unit
- Dedicated register eliminates need for separate address bus
- Content of any register output connected to the bus can be written to memory
- Any register input connected to bus can be target of memory read
  - As long as its LD is enabled

Accumulator (*AC*)

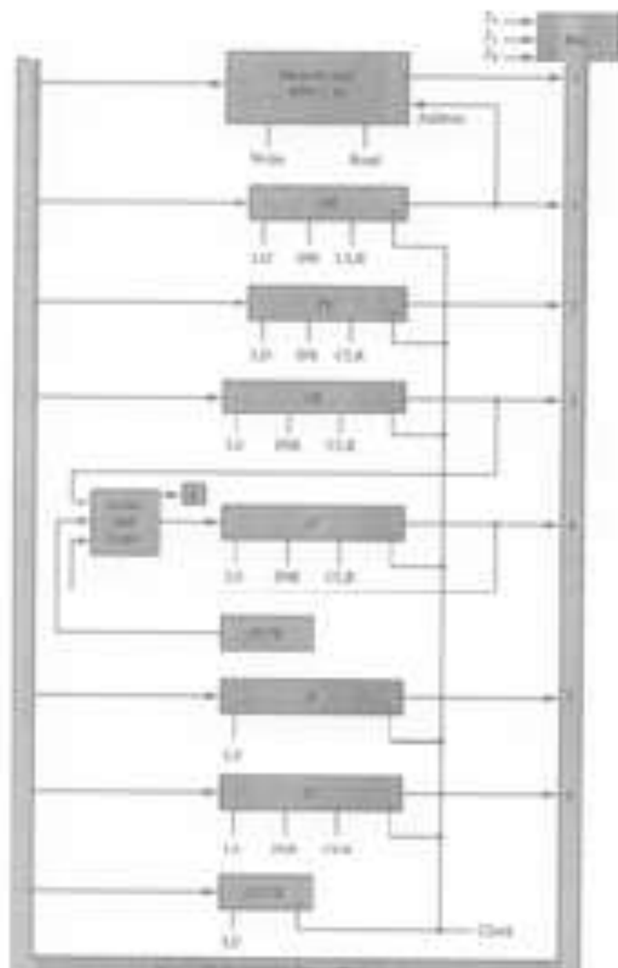
- Input comes from adder and logic circuit
- Adder and logic circuit
  - Input
    - 16-bit output of *AC*
    - 16-bit data register (*DR*)
    - 8-bit input register (*INPR*)
  - Output
    - 16-bit input of *AC*
    - *E* flip-flop (extended *AC* bit, aka overflow)
- *DR* and *AC* input used for arithmetic and logic microoperations

Timing Is Everything

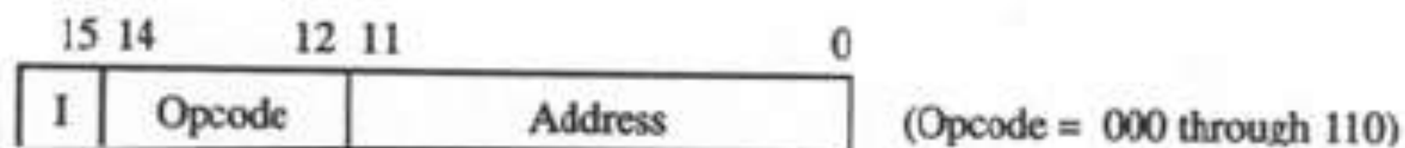
- Content of any register output connected to the bus can be applied to the bus and content of any register input connected to the bus can be loaded from the bus during the same clock cycle
- These 2 microoperations can be executed at the same time

$$DR \leftarrow AC \text{ and } AC \leftarrow DR$$

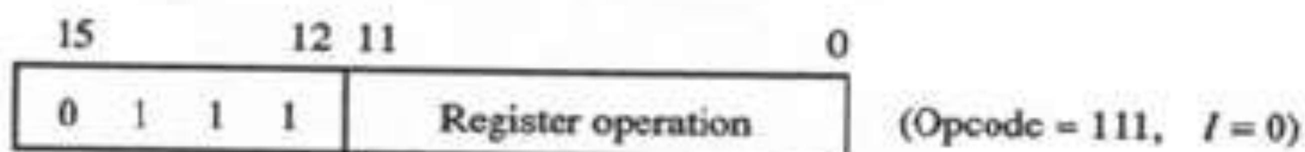
Bus Connections



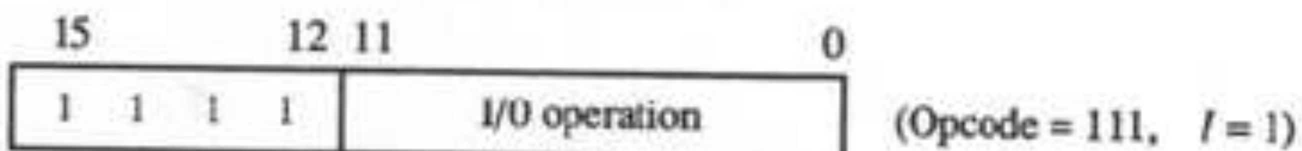
Basic Instruction Formats



(a) Memory – reference instruction



(b) Register – reference instruction



(c) Input – output instruction

Instruction Format

- Only 3 bits used for op code
- Looks like only 8 different op codes are possible
- Wrong!
- For op code 111, one of the low-order 12 bits is turned on to extend the op code definition

Basic Instructions

Symbol	Hexadecimal code I = 0	Hexadecimal code I = 1	Description
AND	0xxx	8xxx	AND memory word to *AC*
ADD	1xxx	9xxx	Add memory word to *AC*
LDA	2xxx	Axxx	Load memory word to *AC*
STA	3xxx	Bxxx	Store content of *AC* in memory
BUN	4xxx	Cxxx	Branch unconditionally
BSA	5xxx	Dxxx	Branch and save return address
ISZ	6xxx	Exxx	Increment and skip if zero
CLA	7800		Clear *AC*
CLE	7400		Clear *E*
CMA	7200		Complement *AC*
CME	7100		Complement *E*
CIR	7080		Circulate right *AC* and *E*
CIL	7040		Circulate left *AC* and *E*
INC	7020		Increment *AC*
SPA	7010		Skip next instruction if *AC* positive
SNA	7008		Skip next instruction if *AC* negative
SZA	7004		Skip next instruction if *AC* zero
SZE	7002		Skip next instruction if *E* is 0
HLT	7001		Halt computer
INP	F800		Input character to *AC*
OUT	F400		Output character from *AC*
SKI	F200		Skip on input flag
SKO	F100		Skip on output flag
ION	F080		Interrupt on
IOF	F040		Interrupt off

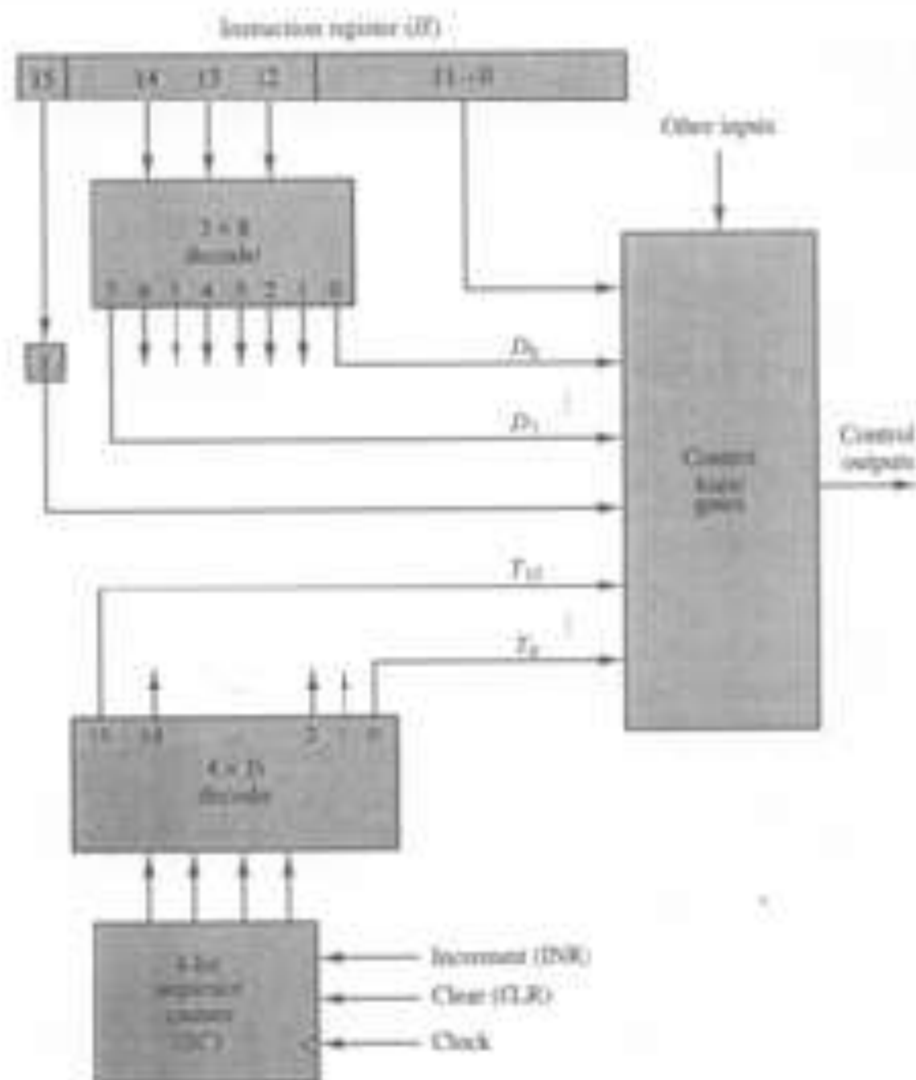
Instruction Set Completeness

- Arithmetic, logical, and shift
- Move data from and to memory and registers
- Program control and status check
- Input and output
  - (I/O, I/O, it's off to the bus we go…)

Control Unit

- Instruction read from memory and put in *IR*
- Leftmost bit put in *I* flip-flop
- 3-bit op code decoded with 3 x 8 decoder into D_0 to D_7
- 4-bit sequence counter (*SC*) decoded with 4 x 16 decoder into T_0 to T_{15} (timing signals)
- *I*, D_0 to D_7, T_0 to T_{15}, rightmost 12 bits of *IR*, and other inputs are fed into control and logic gates

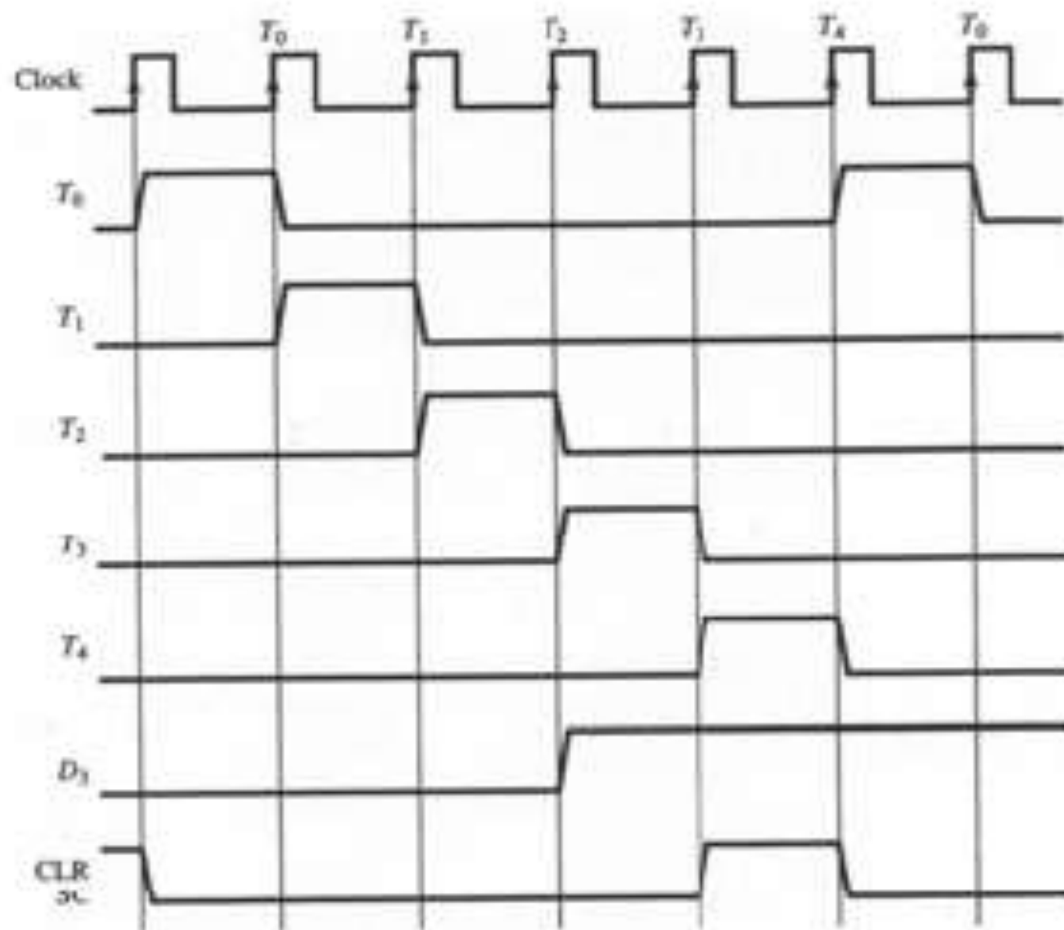
Control Unit



Sequence Counter (*SC*)

- Inputs are increment (INR) and clear (CLR)
- Example
  - *SC* incremented to provide T_0, T_1, T_2, T_3, and T_4
  - At time T_4, *SC* is cleared to 0 if D_3 is active
  - Written as: $D_3T_4: SC \leftarrow 0$

Timing Diagram



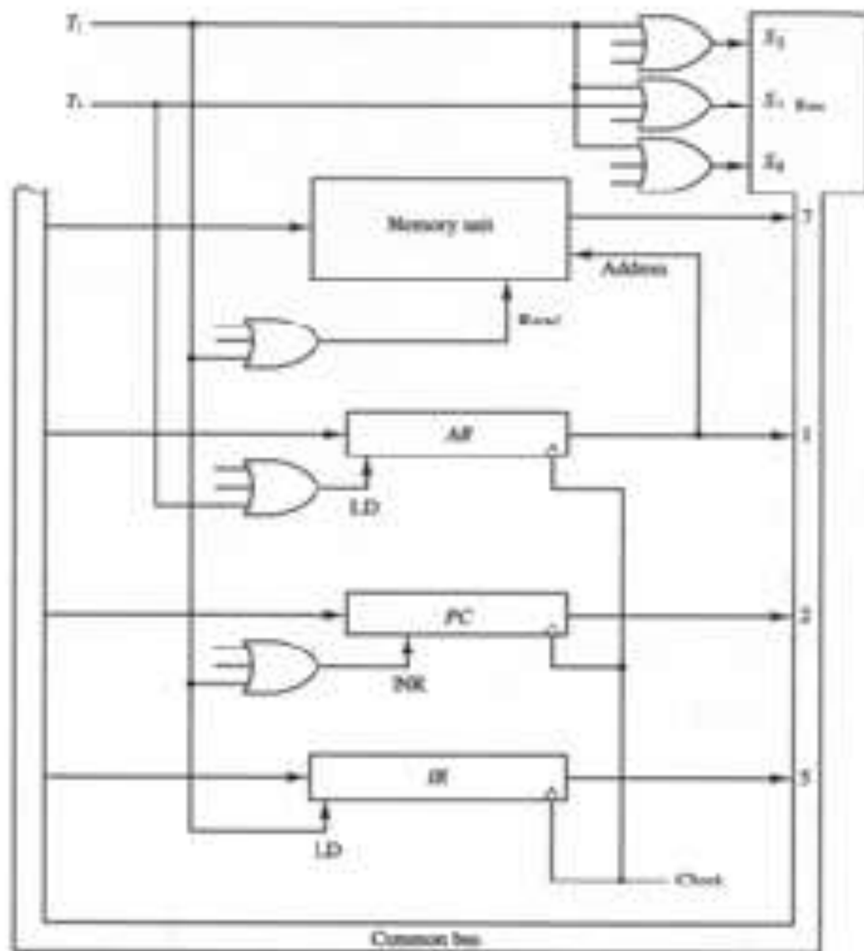
Instruction Cycle

- Fetch instruction from memory
- Decode the instruction
- Read effective address from memory if indirect address
- Execute the instruction

Fetch And Decode

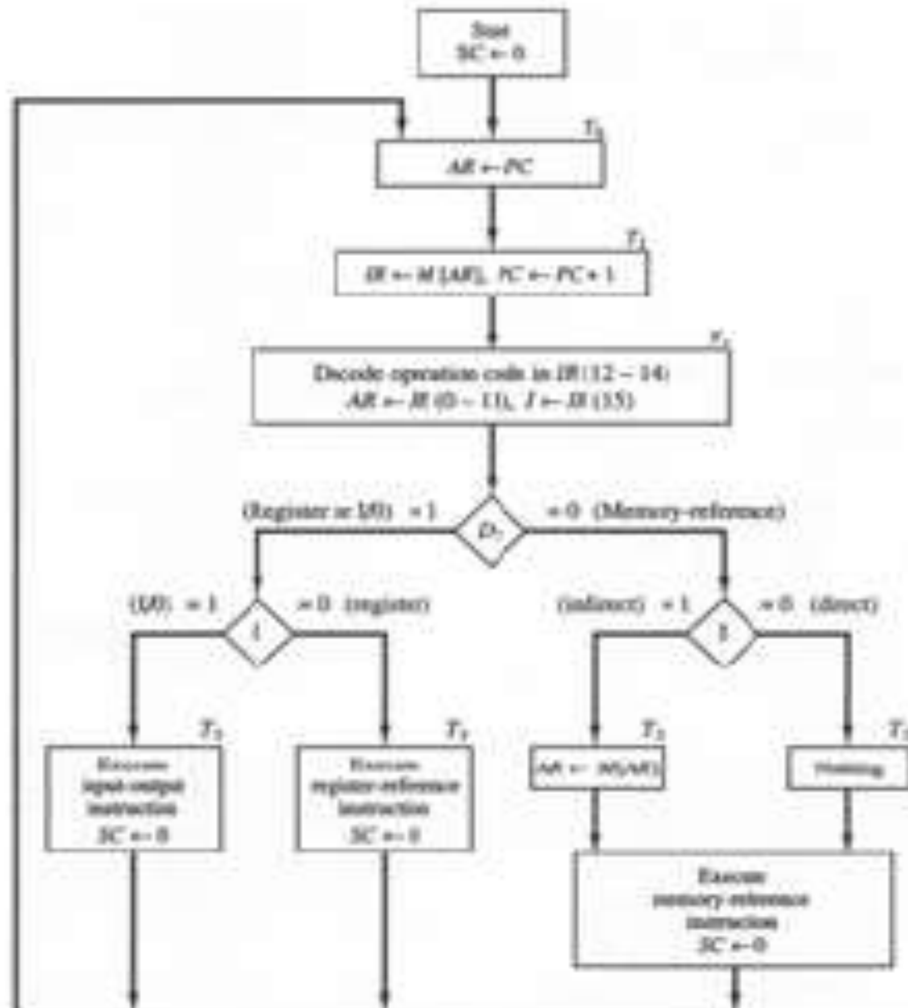
- SC cleared to 0, generating timing signal T_0
- After each clock pulse, SC is incremented
- Fetch and decode microoperations
  - T_0: $AR \leftarrow PC$
  - T_1: $IR \leftarrow M[AR]$,
    $PC \leftarrow PC + 1$
  - T_2: $D_0, \ldots D_7 \leftarrow$ decode $IR(12\text{-}14)$,
    $AR \leftarrow IR(0\text{-}11)$,
    $I \leftarrow IR(15)$

Fetch Phase



Determining The Type of Instruction

Instruction Cycle Flowchart



Instruction Paths

- $D'_7 I T_3$: $AR \leftarrow M[AR]$
- $D'_7 I' T_3$: Do nothing
- $D_7 I' T_3$: Execute a register-reference instruction
- $D_7 I T_3$: Execute an I/O instruction

Register-Reference Instructions

$D_7I'T_3 = r$ (common to all register-reference instructions)
$IR(i) = B_i$ [bit in $IR(0-11)$ that specifies the operation]

	r:	$SC \leftarrow 0$	Clear SC
CLA	rB_{11}:	$AC \leftarrow 0$	Clear AC
CLE	rB_{10}:	$E \leftarrow 0$	Clear E
CMA	rB_9:	$AC \leftarrow \overline{AC}$	Complement AC
CME	rB_8:	$E \leftarrow \overline{E}$	Complement E
CIR	rB_7:	$AC \leftarrow \text{shr } AC,\ AC(15) \leftarrow E,\ E \leftarrow AC(0)$	Circulate right
CIL	rB_6:	$AC \leftarrow \text{shl } AC,\ AC(0) \leftarrow E,\ E \leftarrow AC(15)$	Circulate left
INC	rB_5:	$AC \leftarrow AC + 1$	Increment AC
SPA	rB_4:	If $(AC(15) = 0)$ then $(PC \leftarrow PC + 1)$	Skip if positive
SNA	rB_3:	If $(AC(15) = 1)$ then $(PC \leftarrow PC + 1)$	Skip if negative
SZA	rB_2:	If $(AC = 0)$ then $PC \leftarrow PC + 1)$	Skip if AC zero
SZE	rB_1:	If $(E = 0)$ then $(PC \leftarrow PC + 1)$	Skip if E zero
HLT	rB_0:	$S \leftarrow 0$ (S is a start–stop flip-flop)	Halt computer

Memory-Reference Instructions

Symbol	Operation decoder	Symbolic description
AND	D_0	$AC \leftarrow AC \wedge M[AR]$
ADD	D_1	$AC \leftarrow AC + M[AR], \quad E \leftarrow C_{out}$
LDA	D_2	$AC \leftarrow M[AR]$
STA	D_3	$M[AR] \leftarrow AC$
BUN	D_4	$PC \leftarrow AR$
BSA	D_5	$M[AR] \leftarrow PC, \quad PC \leftarrow AR + 1$
ISZ	D_6	$M[AR] \leftarrow M[AR] + 1$, If $M[AR] + 1 = 0$ then $PC \leftarrow PC + 1$

AND to *AC*

- D_0T_4: $DR \leftarrow M[AR]$
- D_0T_5: $AC \leftarrow AC \wedge DR, \;\; SC \leftarrow 0$

ADD to *AC*

- D_1T_4: $DR \leftarrow M[AR]$
- D_1T_5: $AC \leftarrow AC + DR,\ E \leftarrow C_{out},\ SC \leftarrow 0$

LDA: Load AC

- D_2T_4: $DR \leftarrow M[AR]$
- D_2T_5: $AC \leftarrow DR,\ \ SC \leftarrow 0$

STA: Store *AC*

- D_3T_4: $M[AR] \leftarrow AC,\ SC \leftarrow 0$

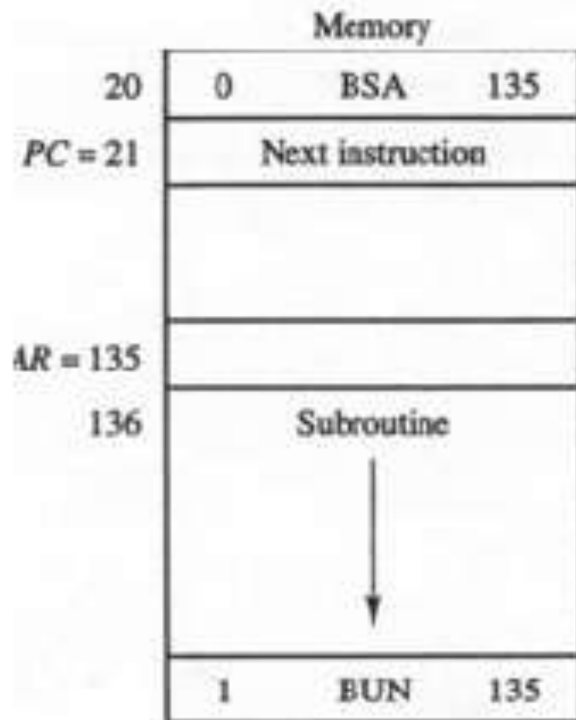
BUN: Branch Unconditionally

- $D_4T_4\!: PC \leftarrow AR,\ SC \leftarrow 0$

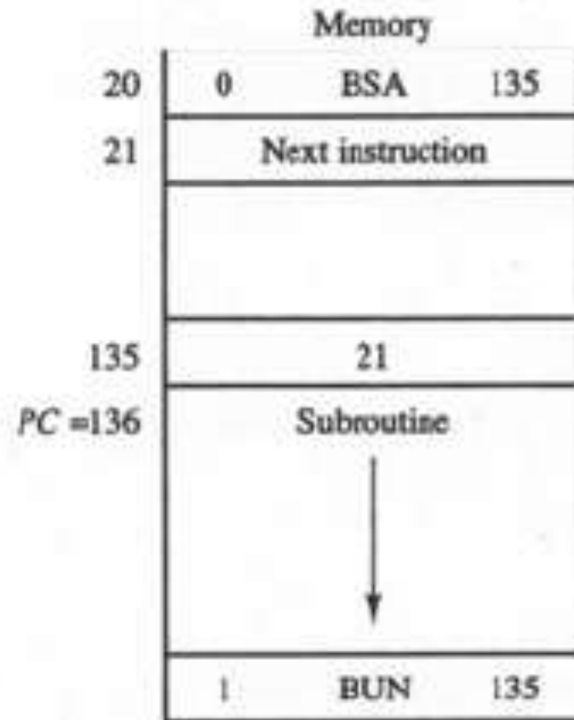
BSA: Branch & Save Return Address

- $D_5T_4: M[AR] \leftarrow PC,\ AR \leftarrow AR + 1$
- $D_5T_5: PC \leftarrow AR,\ SC \leftarrow 0$

BSA Example



(a) Memory, *PC*, and *AR* at time T_4

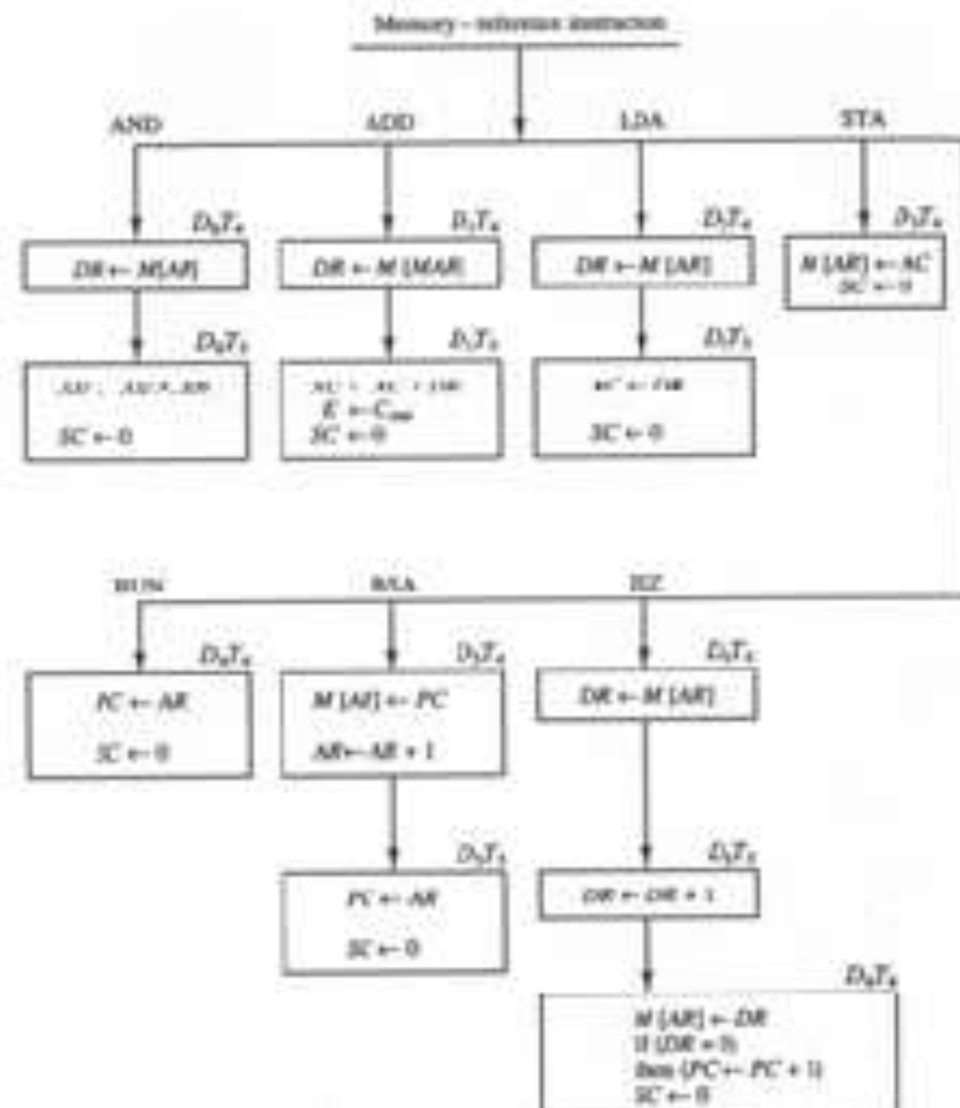


(b) Memory and *PC* after execution

ISZ: Increment & Skip if Zero

- Increment word specified by effective address
  - If value = 0, increment *PC*
- D_6T_4: $DR \leftarrow M[AR]$
- D_6T_5: $DR \leftarrow DR + 1$
- D_6T_6: $M[AR] \leftarrow DR,\ SC \leftarrow 0,$
  if $(DR = 0)$ then $(PC \leftarrow PC + 1)$

Memory-Reference Instructions



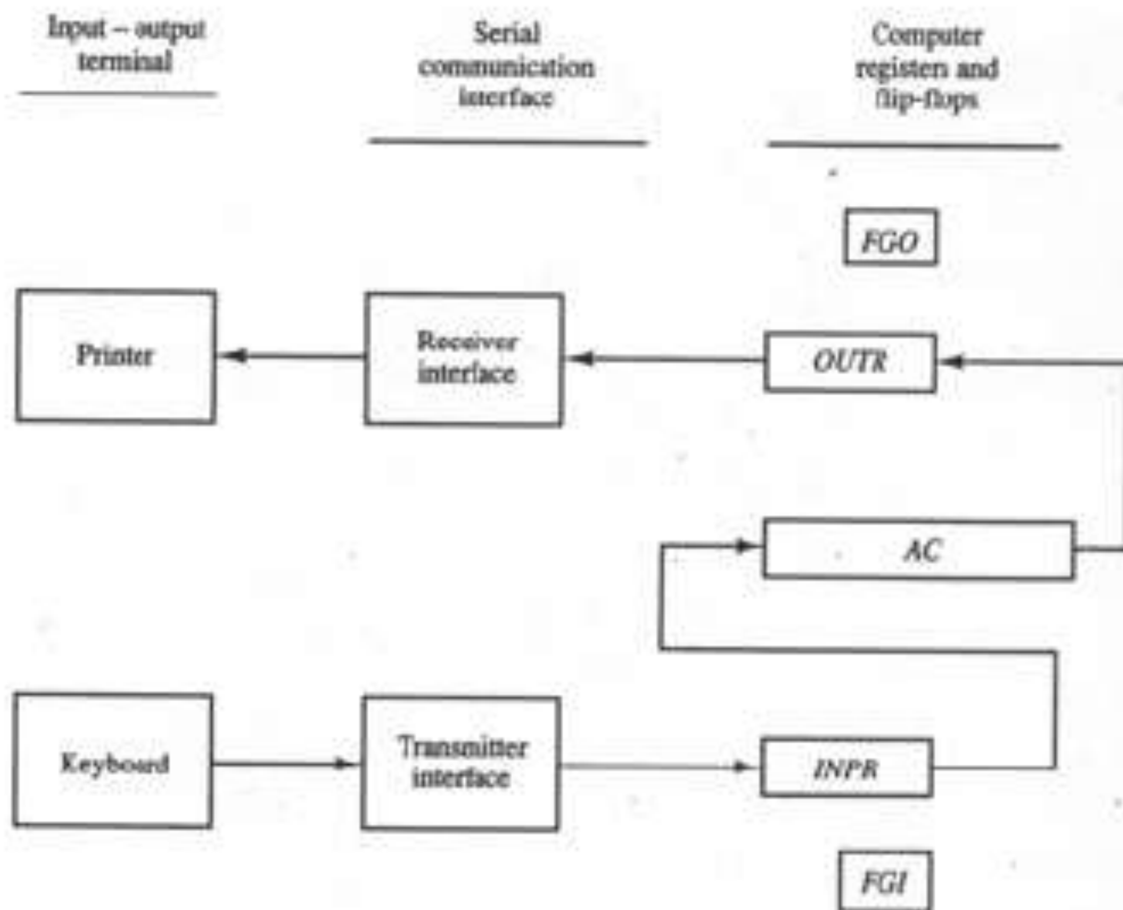
Input Register *INPR*

- 1-bit input flip-flop *FGI*
  - Initially cleared to 0
- When key hit on keyboard
  - 8-bit alphanumeric code is shifted into *INPR*
  - Input flag *FGI* set to 1
  - No more input can be accepted from keyboard
- Computer checks *FGI*, when set to 1
  - Parallel transfer from *INPR* to *AC*
  - *FGI* cleared to 0
  - More input can now be accepted from keyboard

Output Register *OUTR*

- 1-bit output flip-flop *FGO*
  - Initially set to 1
- Computer checks *FGO*, when set to 1
  - Parallel transfer from *AC* to *OUTR*
  - *FGO* cleared to 0
  - No more output can be sent from computer
- Output device accepts 8-bit character
  - *FGO* set to 1
  - More output can now be sent from computer

Input-Output Configuration



Input-Output Instructions

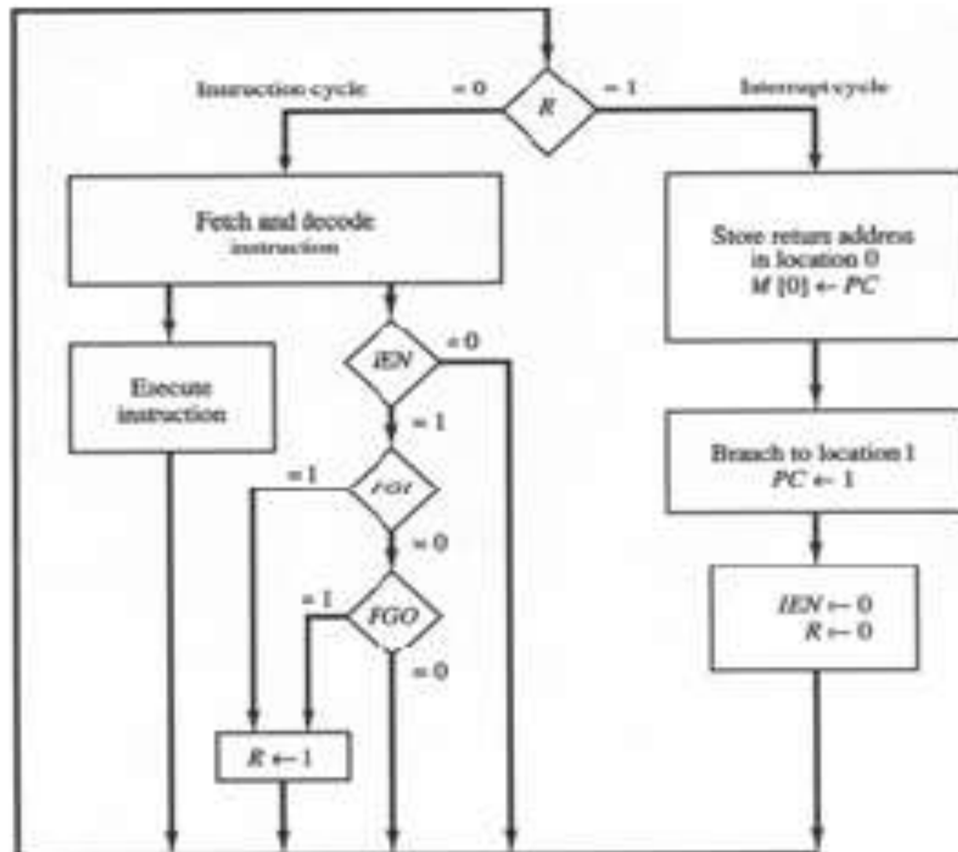
$D_7 I T_3 = p$ (common to all input–output instructions)
$IR(i) = B_i$ [bit in $IR(6\text{–}11)$ that specifies the instruction]

	p:	$SC \leftarrow 0$	Clear SC
INP	pB_{11}:	$AC(0\text{–}7) \leftarrow INPR, \quad FGI \leftarrow 0$	Input character
OUT	pB_{10}:	$OUTR \leftarrow AC(0\text{–}7), \quad FGO \leftarrow 0$	Output character
SKI	pB_9:	If $(FGI = 1)$ then $(PC \leftarrow PC + 1)$	Skip on input flag
SKO	pB_8:	If $(FGO = 1)$ then $(PC \leftarrow PC + 1)$	Skip on output flag
ION	pB_7:	$IEN \leftarrow 1$	Interrupt enable on
IOF	pB_6:	$IEN \leftarrow 0$	Interrupt enable off

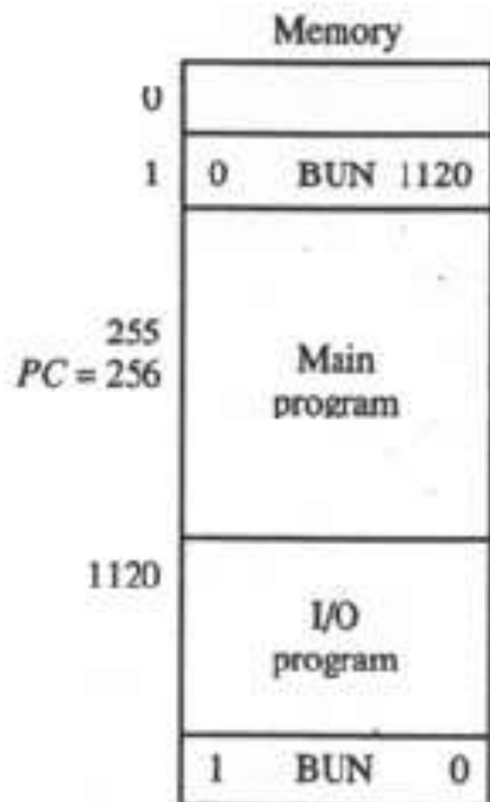
Interrupt Enable *IEN*

- Having computer constantly check *FGI* and *FGO* via an executable instruction is a waste of time
- Instead, *IEN* is programmatically set, effectively saying "let me know if you need me"
  - Meanwhile, it keeps executing instructions
- During each execution cycle, if computer detects *FGI* or *FGO* is set, then R is set to 1
- The interrupt happens when the computer is ready to fetch the next instruction
  - $R = 0$ means go through instruction cycle
  - $R = 1$ means go through interrupt cycle

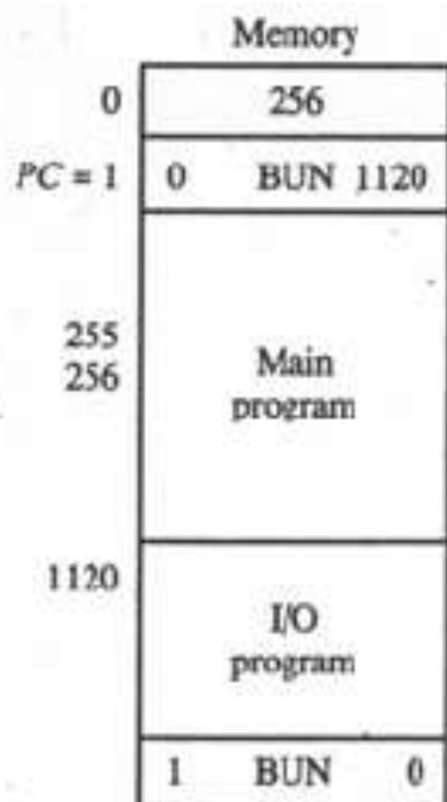
Interrupt Flowchart



Interrupt Cycle Example



(a) Before interrupt



(b) After interrupt cycle

Interrupt Cycle

- Condition for setting R to 1

  $T'_0 T'_1 T'_2 (IEN)(FGI + FGO): R \leftarrow 1$

- Fetch phase modified to service interrupt

  $RT_0: AR \leftarrow 0,\ TR \leftarrow PC$

  $RT_1: M[AR] \leftarrow TR,\ PC \leftarrow 0$

  $RT_2: PC \leftarrow PC + 1,\ IEN \leftarrow 0,\ R \leftarrow 0,\ SC \leftarrow 0$

Microoperation

- Elementary operation performed on data within one or more registers
- Operation result could update same register or another register
- Examples: shift, count, clear, and load
  - Counter with parallel load can perform count and load
  - Bidirectional shift can shift left or shift right

Microoperation Summary

- Transfer data from one register to another (already covered)
- Perform arithmetic operations on numeric data stored in registers
- Manipulate bits on non-numeric data in registers
- Shift bits on data stored in registers

Arithmetic Microoperations

- Addition
- Subtraction
- Increment
- Decrement
- Shift

Add and Subtract

- Add: $R3 \leftarrow R1 + R2$
- Subtract: $R3 \leftarrow R1 - R2$

  $R3 \leftarrow R1 + \overline{R2} + 1$
  - Add 2's complement of $R2$

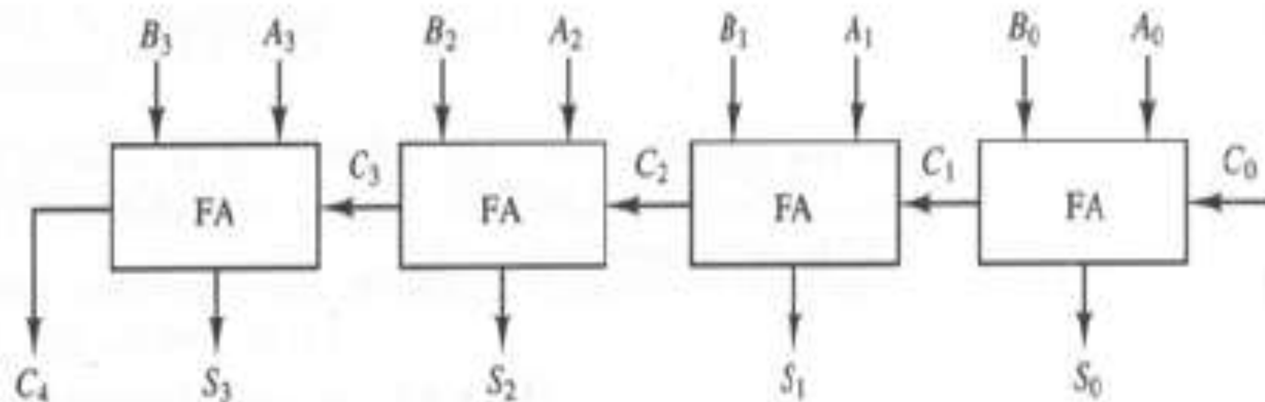
Arithmetic Microoperations

Symbols	Description
$R3 \leftarrow R1 + R2$	Contents $R1$ plus $R2$ put in $R3$
$R3 \leftarrow R1 - R2$	Contents $R1$ minus $R2$ put in $R3$
$R2 \leftarrow \overline{R2}$	1's complement what's in $R2$
$R2 \leftarrow \overline{R2} + 1$	2's complement what's in $R2$
$R3 \leftarrow R1 + \overline{R2} + 1$	$R1$ plus 2's complement of $R2$
$R1 \leftarrow R1 + 1$	Increment $R1$ by 1
$R1 \leftarrow R1 - 1$	Decrement $R1$ by 1

Binary Adder

- Full-adder is a digital circuit that generates the arithmetic sum of two bits and a previous carry
- Binary-adder is a digital circuit that generates the arithmetic sum of two binary numbers of any length
  - Constructed with full-adder circuits
    - Output carry of one FA connected to input carry of next
  - n-bit binary adder requires n full-adders

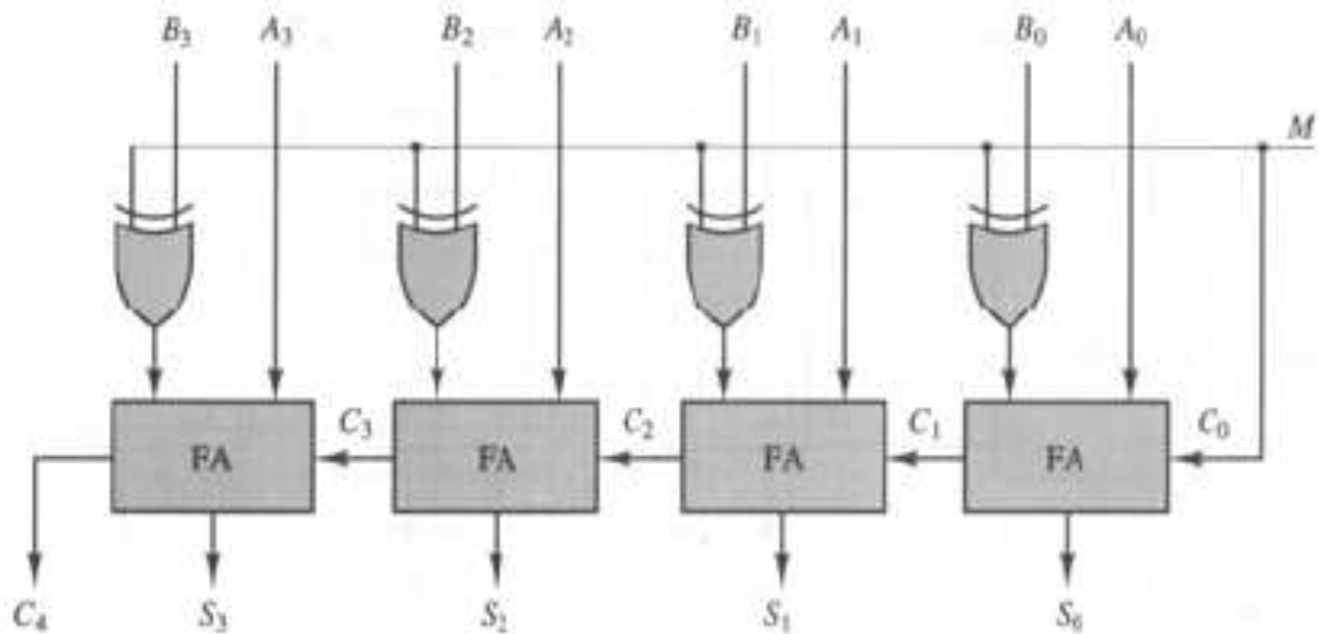
4-Bit Binary Adder



Binary Adder-Subtractor

- Addition and subtraction combined into one common circuit by including an XOR gate with each FA
- Mode input M controls the operation
  - Adder: $M = 0$
  - Subtractor: $M = 1$

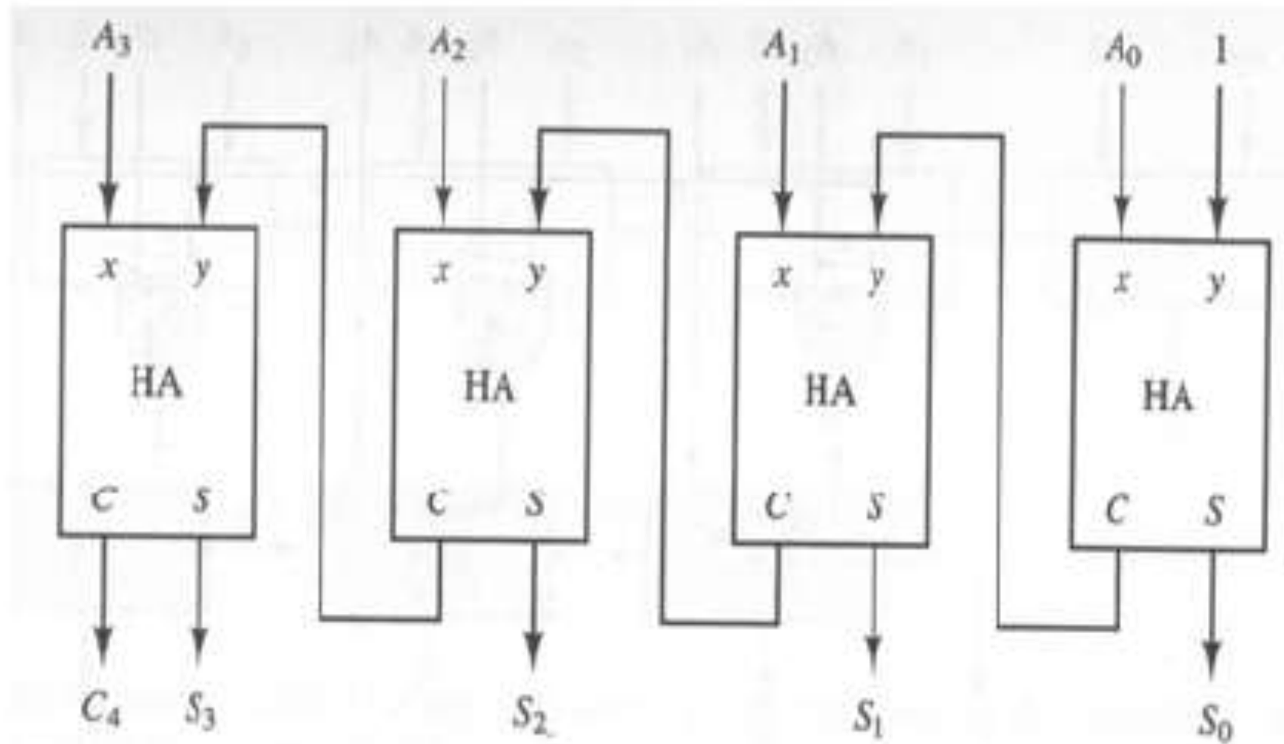
4-Bit Adder-Subtractor



Binary Incrementer

- Implemented by a binary counter
- Can use half-adders
- n-bit binary incrementer uses n half-adders

4-Bit Binary Incrementer



Arithmetic Circuit

- Arithmetic microoperations on slide #19 implemented in one circuit
  - Base component is parallel adder
  - Based on inputs to adder, can do different arithmetic operations

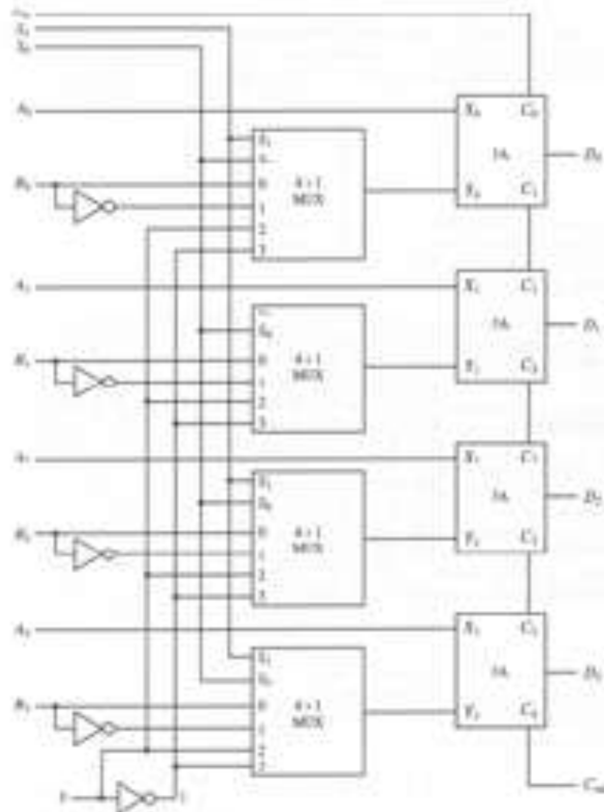
4-Bit Arithmetic Circuit

- Two 4-bit inputs A and B
- One 4-bit output D
- Four inputs A go directly to X inputs of FA
- Input to 4x1 Multiplexers
  - B, B'
  - 0, 1
  - S_0, S_1
- Multiplexer output goes to Y input of FA

Arithmetic Circuit Function Table

Select			In	Output	
S_1	S_0	C_{in}	Y	$D = A + Y + C_{in}$	Microoperation
0	0	0	B	$D = A + B$	Add
0	0	1	B	$D = A + B + 1$	Add w/carry
0	1	0	$\overline{B}$	$D = A + \overline{B}$	Subtract w/borrow
0	1	1	$\overline{B}$	$D = A + \overline{B} + 1$	Subtract
1	0	0	0	$D = A$	Transfer A
1	0	1	0	$D = A + 1$	Increment A
1	1	0	1	$D = A - 1$	Decrement A
1	1	1	1	$D = A$	Transfer A

4-Bit Arithmetic Circuit



Logic Microoperations

- Binary operations on strings of bits stored in registers
- Each bit is dealt with separately
- Exclusive-OR example
  - $P: R1 \leftarrow R1 \oplus R2$
  - Contents of $R1$ 0011
  - Contents of $R2$ $\underline{0101}$
  - Contents of $R1$ after $P = 1$ 0110

Special Symbols

- Logic microoperations
  - OR $\vee$
  - AND $\wedge$
  - Complement bar on top of register symbol
- Distinguish logic microoperation from Boolean function

$$P + Q:\ R1 \leftarrow R2 + R3,\ \ R4 \leftarrow R5 \vee R6$$

- $P + Q$ — "+": OR op between two control function binary variables
- $R2 + R3$ — "+": add microop
- $R5 \vee R6$ — "$\vee$": OR microop

2 Variable/16 Function Truth Table

x	y	F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

16 Logic Microoperations – Part 1

Boolean function	Microoperation	Name
$F_0 = 0$	$F \leftarrow 0$	Clear
$F_1 = xy$	$F \leftarrow A \wedge B$	AND
$F_2 = xy'$	$F \leftarrow A \wedge \overline{B}$	
$F_3 = x$	$F \leftarrow A$	Transfer A
$F_4 = x'y$	$F \leftarrow \overline{A} \wedge B$	
$F_5 = \mathrm{y}$	$F \leftarrow B$	Transfer B
$F_6 = x \oplus y$	$F \leftarrow A \oplus B$	Exclusive-OR
$F_7 = x + y$	$F \leftarrow A \vee B$	OR

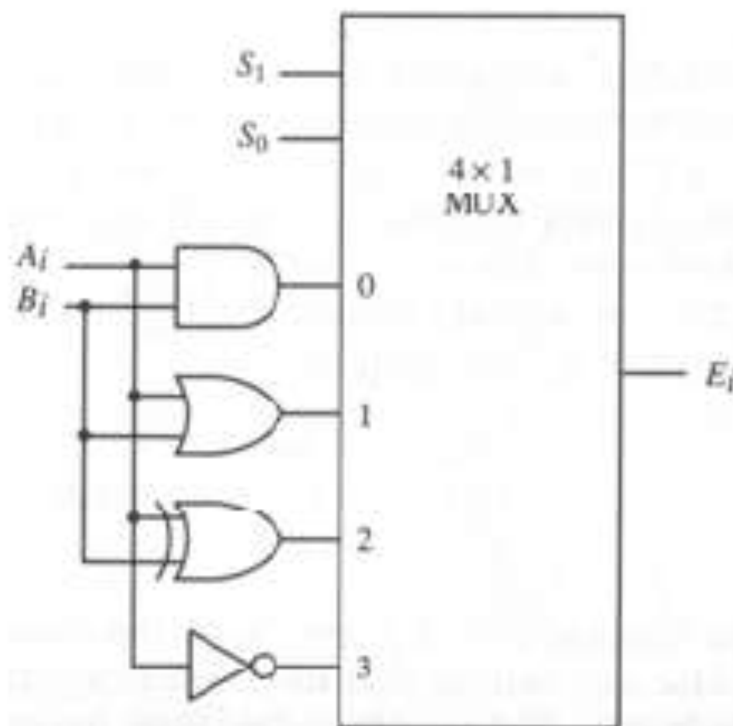
16 Logic Microoperations – Part 2

Boolean function	Microoperation	Name
$F_8 = (x + y)'$	$F \leftarrow \overline{A \vee B}$	NOR
$F_9 = (x \oplus y)'$	$F \leftarrow \overline{A \oplus B}$	Exclusive-NOR
$F_{10} = y'$	$F \leftarrow \overline{B}$	Complement B
$F_{11} = x + y'$	$F \leftarrow A \vee \overline{B}$	
$F_{12} = x'$	$F \leftarrow \overline{A}$	Complement A
$F_{13} = x' + y$	$F \leftarrow \overline{A} \vee B$	
$F_{14} = (xy)'$	$F \leftarrow \overline{A \wedge B}$	NAND
$F_{15} = 1$	$F \leftarrow$ all 1's	Set to all 1's

Hardware Implementation

- Logic gates inserted for each bit or pair of bits in the registers to perform needed logic function
- Most computers use only four microoperations and derive the rest
  - AND, OR, XOR, complement

One Stage Of Logic Circuit



(a) Logic diagram

S_1	S_0	Output	Operation
0	0	$E = A \wedge B$	AND
0	1	$E = A \vee B$	OR
1	0	$E = A \oplus B$	XOR
1	1	$E = \overline{A}$	Complement

(b) Function table

Examples

- Following slides are examples of logic microoperations that are used to manipulate individual bits of register A by the bits contained in another register, B

Selective Set

- Register A bits set to 1 where register B has a 1, otherwise A bits left unchanged
- 0011 A before x
- 0101 B (logic operand) y
- 0111 A after F_7
- OR microoperation

Selective Complement

- Register A bits complemented where register B has a 1, otherwise A bits left unchanged
- 0011 A before x
- 0101 B (logic operand) y
- 0110 A after F_6
- Exclusive-OR microoperation

Selective Clear

- Register A bits cleared to 0 where register B has a 1, otherwise A bits left unchanged
- 0011 A before x
- 0101 B (logic operand) y
- 0010 A after F_2
- $A \leftarrow A \wedge \overline{B}$ logic microoperation

Mask

- Register A bits cleared to 0 where register B has a 0, otherwise A bits left unchanged
- 0011 A before x
- <u>0101</u> B (logic operand) y
- 0001 A after F_1
- AND microoperation

Insert

- Inserts a new value into a group of bits
  - First mask the bits (AND)
  - Next selective-set them with target bit string (OR)
- Insert 1110 into leftmost 4 bits of A
- 1001 0101 A before
- 0000 1111 B (mask)
- 0000 0101 A after masking
- 1110 0000 B (selective-set)
- 1110 0101 A after insert completes

Clear

- Compares bits in A and B and produces all zeros if the two registers have equal values
- Accomplished with exclusive-OR, then all bits are checked for being 0

Shift Microoperations

- First flip-flop gets binary data from serial input
  - For shift left, “first” is rightmost flip-flop
  - For shift right, “first” is leftmost flip-flop
- Serial input source determined by type of shift
  - 0 for logical
  - Other end for circular
  - 0 fill on right and sign bit on left for arithmetic
    - Overflow when sign bit changes

Logical Shift Example

- Original value: 11010011
- Value after shift right: 01101001
- Or, value after shift left: 10100110

Circular Shift Example

- Original value: 11010011
- Value after shift right: 11101001
- Or, value after shift left: 10100111

Arithmetic Shift Example

- Original value: 11010011
- Value after shift right: 11101001
- Or, value after shift left: 10100110

- Second original value: 10011010
- Value after shift right: 11001101
- Or, value after shift left: 00110100 (overflow)

Overflow Flip-Flop

- Bits in register: $R_{n-1}\ R_{n-2}\ \ldots\ R_1\ R_0$
- V_s detects arithmetic shift left overflow
  - = 1 indicates overflow condition
- $V_s = R_{n-1} \oplus R_{n-2}$

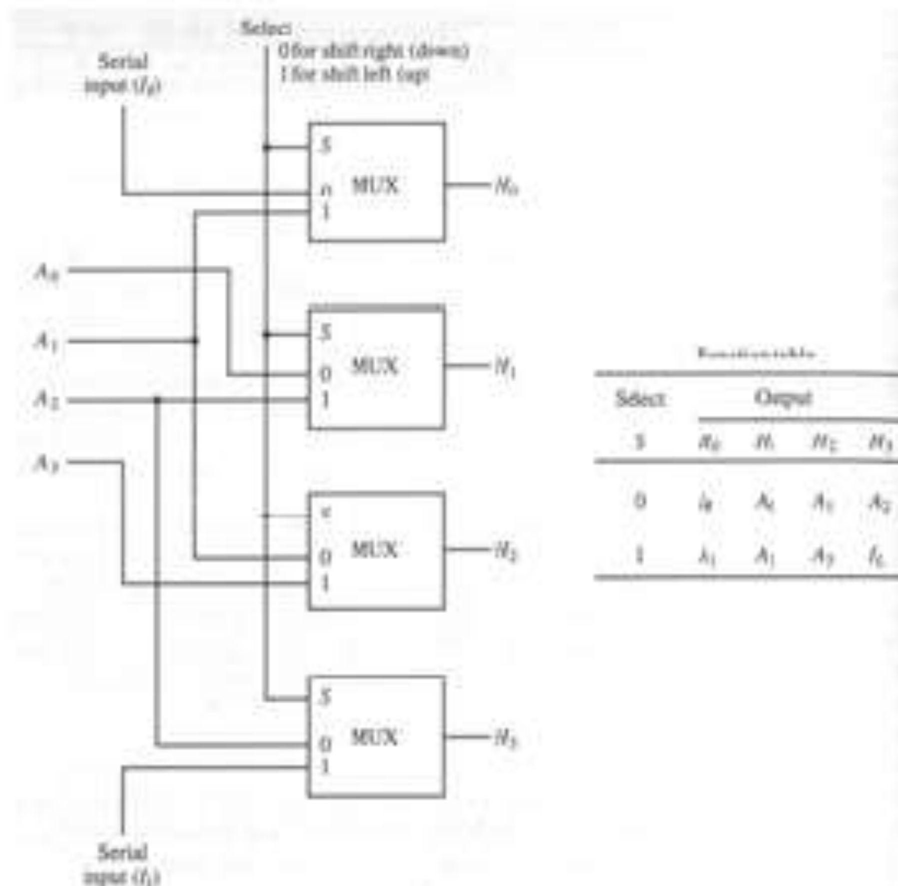
Shift Microoperations

Symbolic designation	Description
$R \leftarrow \text{shl } R$	Shift-left R
$R \leftarrow \text{shr } R$	Shift-right R
$R \leftarrow \text{cil } R$	Circular shift-left R
$R \leftarrow \text{cir } R$	Circular shift-right R
$R \leftarrow \text{ashl } R$	Arithmetic shift-left R
$R \leftarrow \text{ashr } R$	Arithmetic shift-right R

Shift Hardware Implementation

- Register content is placed on bus
- Bus connected to combination circuit shifter
- Shifted value loaded back into same register
- All done in one clock pulse

4-Bit Combinational Circuit Shifter



Function table

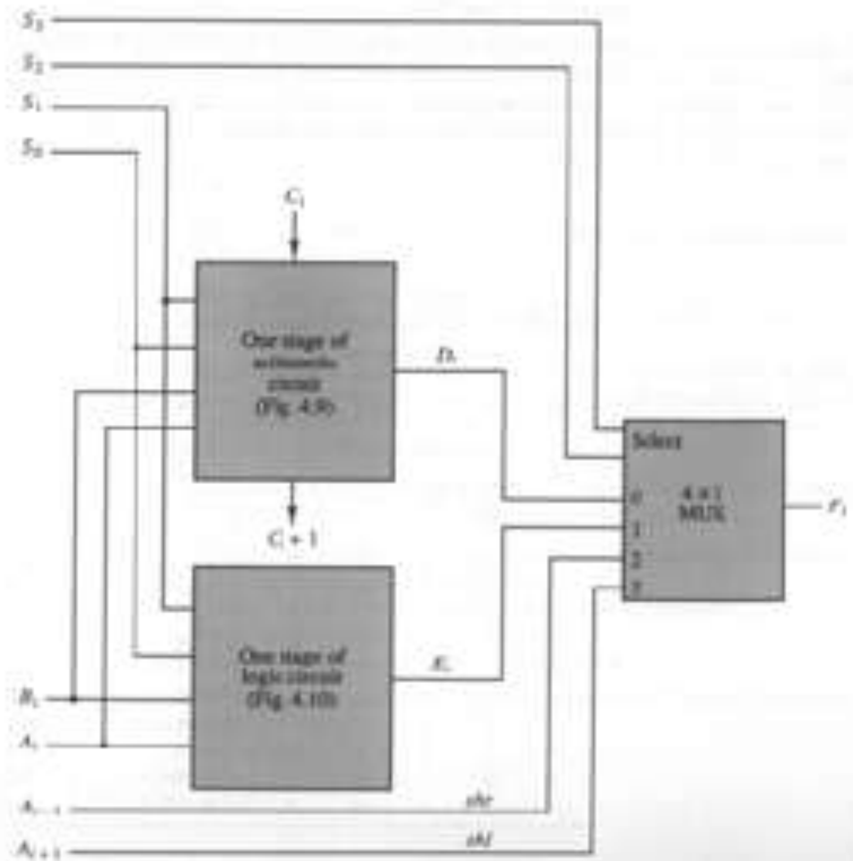
Select	Output			
S	H_0	H_1	H_2	H_3
0	I_R	A_0	A_1	A_2
1	A_1	A_2	A_3	I_L

Arithmetic Logic Unit (ALU)

- One or more source registers provide input
- ALU performs operation
- Result transferred into destination register
- All done in one clock pulse

- Arithmetic, logic, and shift circuits can be combined into single ALU with common selection variables

One Stage Of ALU



ALU Function Table

TABLE 4-8 Function Table for Arithmetic Logic Shift Unit

Operation select						
S_3	S_2	S_1	S_0	C_{in}	Operation	Function
0	0	0	0	0	$F = A$	Transfer A
0	0	0	0	1	$F = A + 1$	Increment A
0	0	0	1	0	$F = A + B$	Addition
0	0	0	1	1	$F = A + B + 1$	Add with carry
0	0	1	0	0	$F = A + \overline{B}$	Subtract with borrow
0	0	1	0	1	$F = A + \overline{B} + 1$	Subtraction
0	0	1	1	0	$F = A - 1$	Decrement A
0	0	1	1	1	$F = A$	Transfer A
0	1	0	0	×	$F = A \wedge B$	AND
0	1	0	1	×	$F = A \vee B$	OR
0	1	1	0	×	$F = A \oplus B$	XOR
0	1	1	1	×	$F = \overline{A}$	Complement A
1	0	×	×	×	$F = \text{shr } A$	Shift right A into F
1	1	×	×	×	$F = \text{shl } A$	Shift left A into F

Elements of an Instruction

- Operation code (Op code)
  - Do this
- Source Operand reference
  - To this
- Result Operand reference
  - Put the answer here
- Next Instruction Reference
  - When you have done that, do this...

Where have all the Operands gone?

- Main memory (or virtual memory or cache)
- CPU register
- I/O device

Instruction Representation

Instruction Types

- Data processing
- Data storage (main memory)
- Data movement (I/O)
- Program flow control

Number of Addresses (a)

- 3 addresses
  - Operand 1, Operand 2, Result
  - a = b + c;
  - May be a forth - next instruction (usually implicit)
  - Not common
  - Needs very long words to hold everything

Number of Addresses (b)

- 2 addresses
  - One address doubles as operand and result
  - a = a + b
  - Reduces length of instruction
  - Requires some extra work
    - Temporary storage to hold some results

Number of Addresses (c)

- 1 address
  - Implicit second address
  - Usually a register (accumulator)
  - Common on early machines

Number of Addresses (d)

- 0 (zero) addresses
  - All addresses implicit
  - Uses a stack
  - e.g. push a
  - push b
  - add
  - pop c
  - c = a + b

How Many Addresses

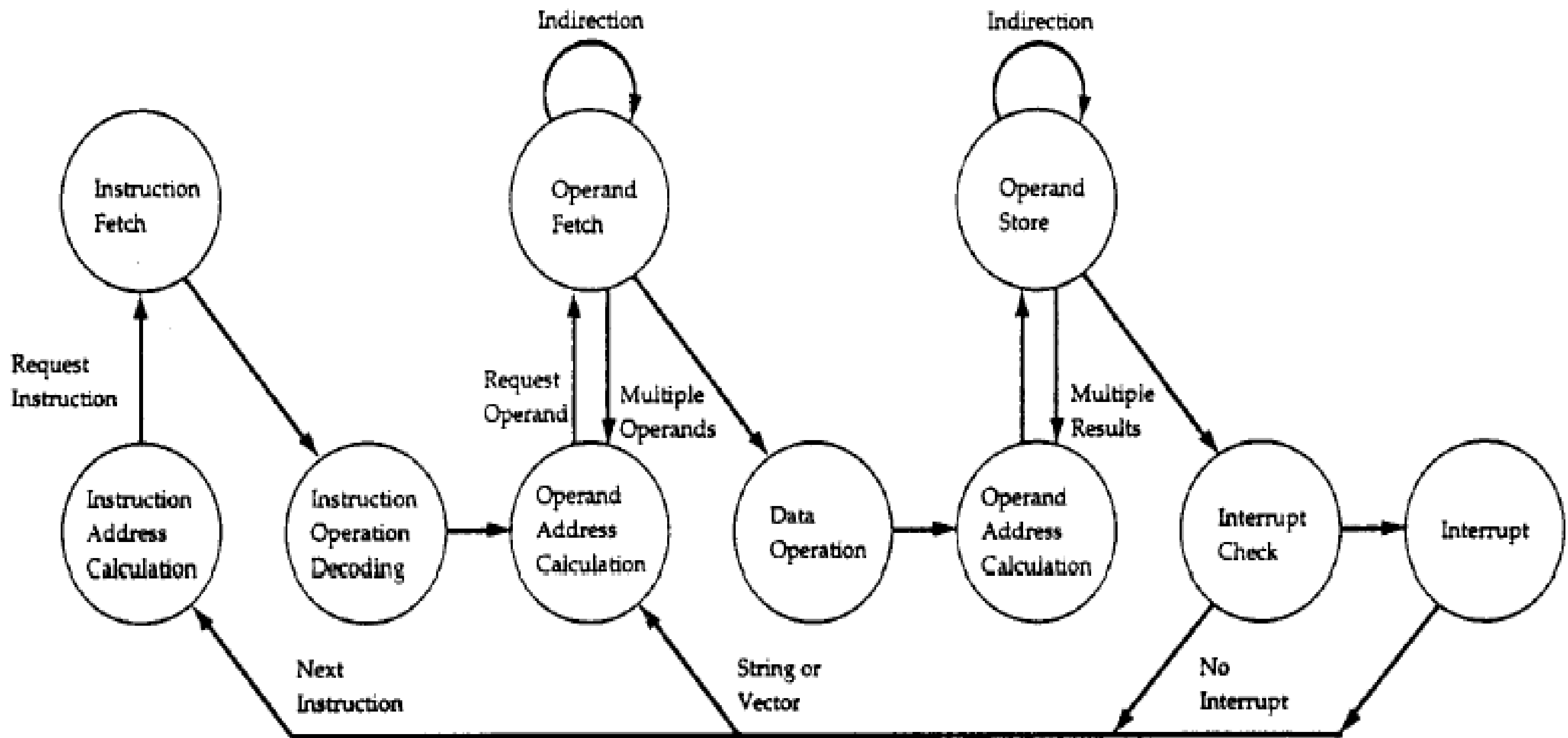
- More addresses
  - More complex (powerful?) instructions
  - More registers
    - Inter-register operations are quicker
  - Fewer instructions per program
- Fewer addresses
  - Less complex (powerful?) instructions
  - More instructions per program
  - Faster fetch/execution of instructions

Design Decisions (1)

- Operation repertoire
  - How many ops?
  - What can they do?
  - How complex are they?
- Data types
- Instruction formats
  - Length of op code field
  - Number of addresses

Design Decisions (2)

- Registers
  - Number of CPU registers available
  - Which operations can be performed on which registers?
- Addressing modes (later…)
- RISC v CISC



Instruction cycle with interrupt

Interrupts - In Summary

- An interruption of normal processing
- Improves processing efficiency
- Allows the processor to execute other instructions while an I/O operation is in progress
- A suspension of a process caused by an event external to that process and performed in such a way that the process can be resumed

Classes of Interrupts

- **Program**
  - arithmetic overflow
  - division by zero
  - execute illegal instruction
  - reference outside user's memory space
- **Timer**
- **I/O**
- **Hardware failure**

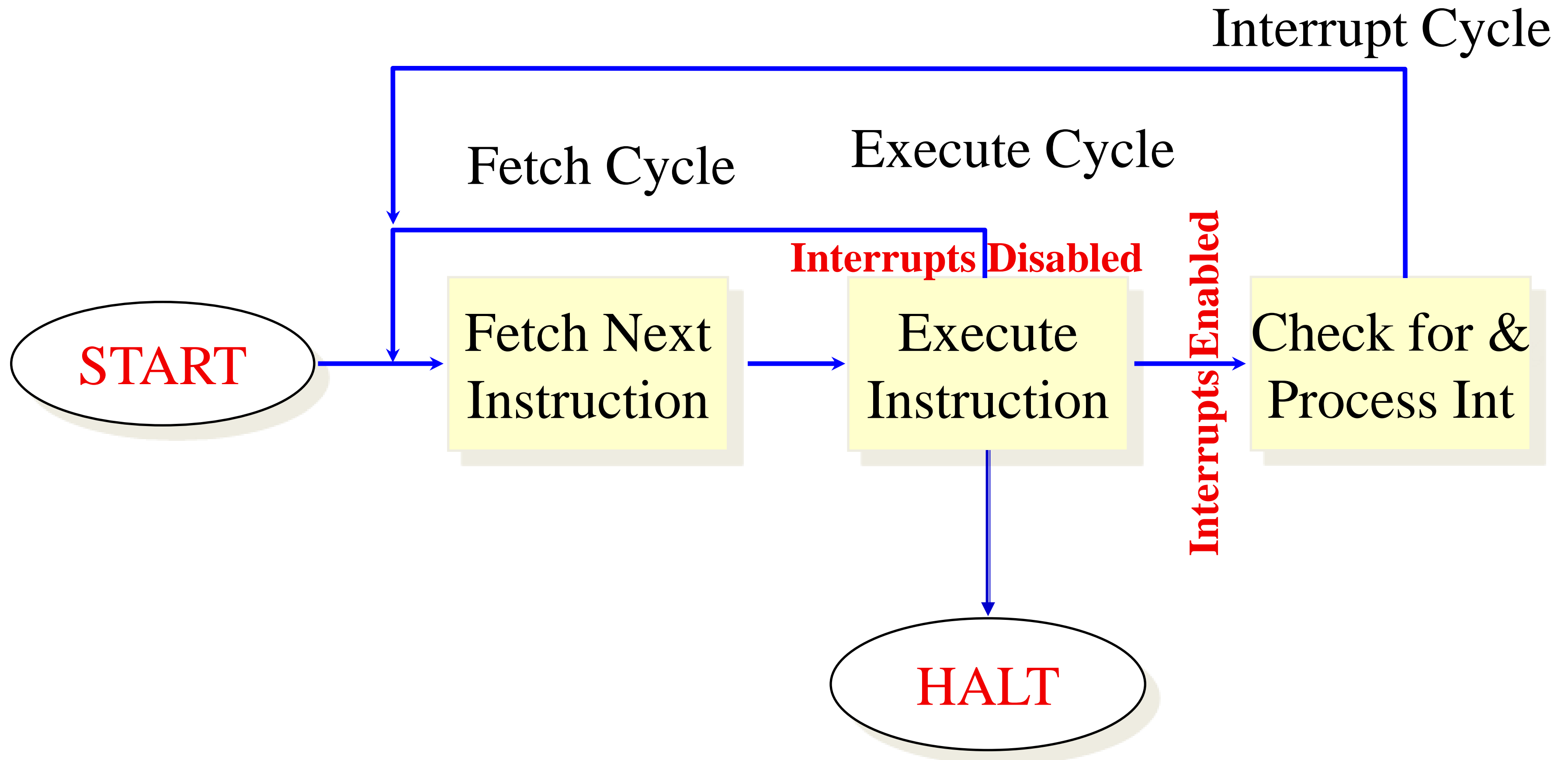
Common Functions of Interrupts

- Interrupts transfer control to the interrupt service routine generally, through the ***interrupt vector***
- Interrupt architecture must save the address of the interrupted instruction.
- interrupts are *disabled* while another interrupt is being processed to prevent a ***lost interrupt***.
- A ***trap*** is a software-generated interrupt caused either by an error or a user request.
- *An operating system is interrupt driven*.

Interrupt Handling

- The operating system preserves the state of the CPU by storing registers and the program counter.
- Determines which type of interrupt has occurred:
  - ***polling***
  - ***vectored interrupt system***
- Separate segments of code determine what action should be taken for each type of interrupt

Instruction Cycle with Interrupts



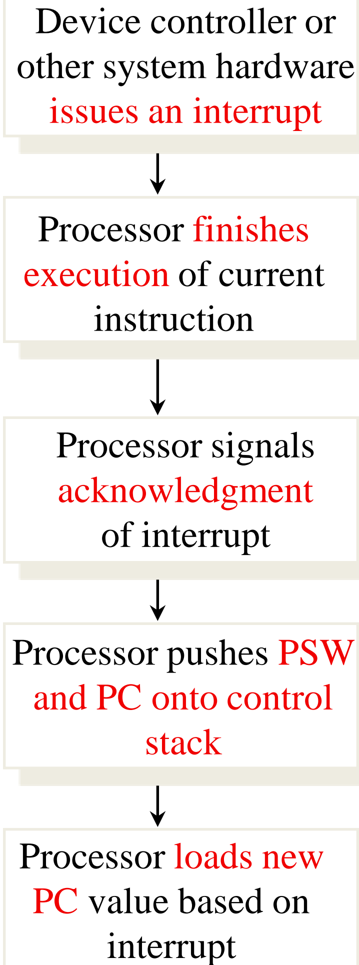
Interrupt Cycle

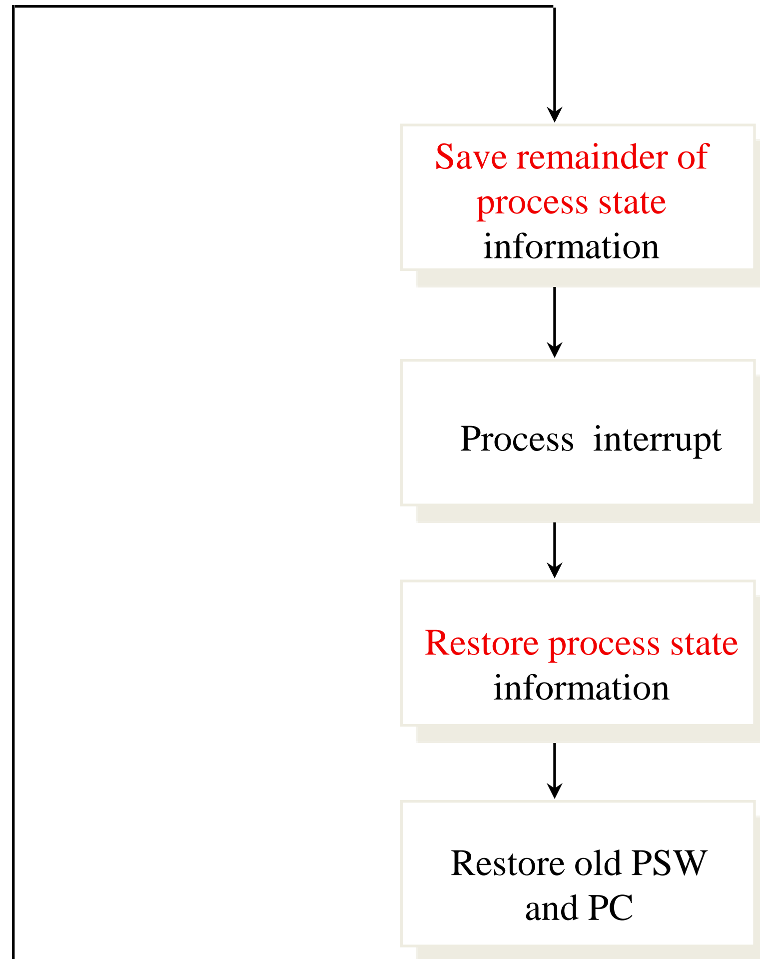
- Processor checks for interrupts
- If no interrupts fetch the next instruction for the current program
- If an interrupt is pending, suspend execution of the current program, and execute the interrupt handler

Interrupt Service Routine (aka handler)

- A program that determines nature of the interrupt and performs whatever actions are needed
- Control is transferred to this program
- Generally part of the operating system

Simple Interrupt Processing





What about Multiple Interrupts

- Simple Approach - disable interrupts
- Use Priorities to differentiate between interrupt classes

Multiple Interrupts Sequential Order

- Disable interrupts so processor can complete task
- Interrupts remain pending until the processor enables interrupts
- After interrupt handler routine completes, the processor checks for additional interrupts

Multiple Interrupts Priorities

- Higher priority interrupts cause lower-priority interrupts to wait
- Causes a lower-priority interrupt handler to be interrupted
- Example when input arrives from communication line, it needs to be absorbed quickly to make room for more input

Unit – 03

CPU

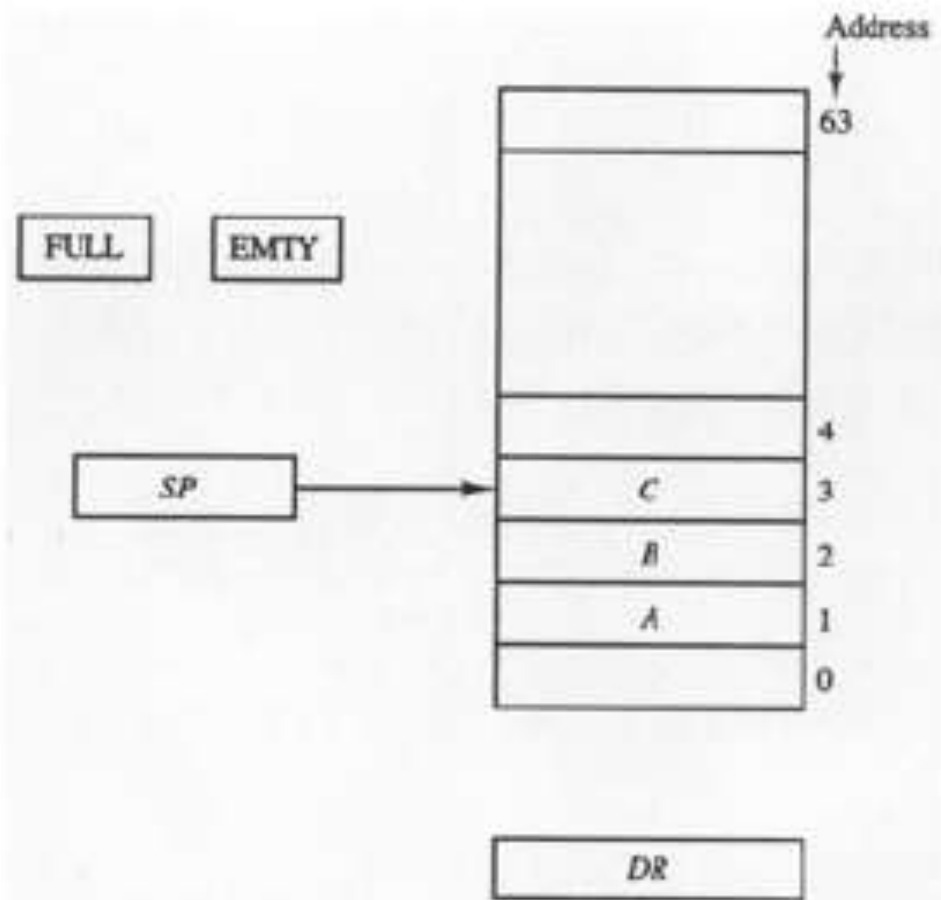
&

Control Unit

Stack

- Last-in first-out (LIFO) list
- Stack pointer (*SP*)
  - Always points to top item in stack
  - Register that holds stack address
- Operations
  - Push – put new item in top of stack
  - Pop – remove item from top of stack

64-Word Stack Block Diagram



Registers:

FULL – one bit

EMTY – one bit

SP – six bit

DR – stack I/O

Stack Initialization

- *SP* cleared to 0
- *EMTY* set to 1
- *FULL* cleared to 0

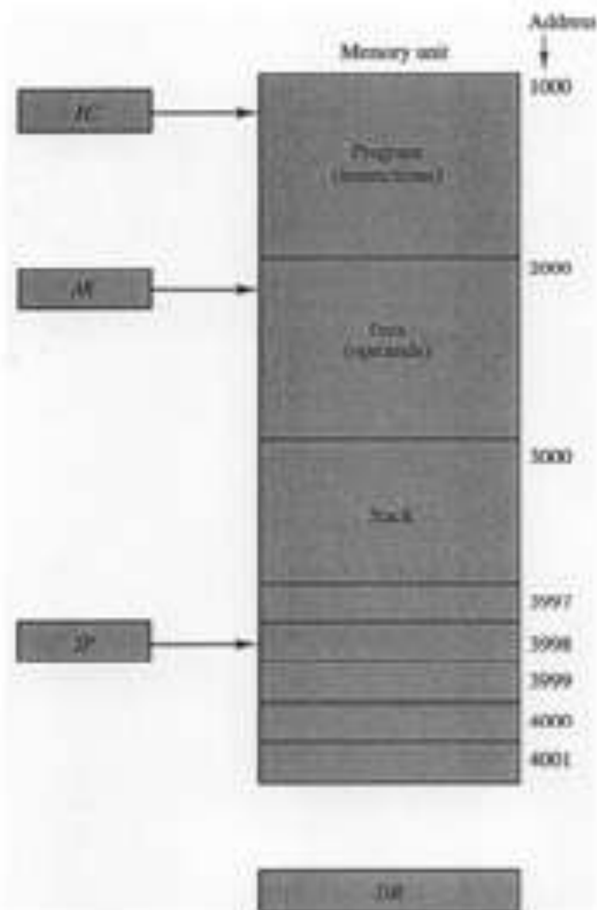
Push

- $SP \leftarrow SP + 1$
- $M[SP] \leftarrow DR$
- If $(SP = 0)$ then $(FULL \leftarrow 1)$
- $EMTY \leftarrow 0$

Pop

- $DR \leftarrow M[SP]$
- $SP \leftarrow SP - 1$
- If $(SP = 0)$ then $(EMTY \leftarrow 1)$
- $FULL \leftarrow 0$

Memory Layout Example



In this example,
PUSH decrements SP
POP increments SP

Arithmetic Expression Notations

A + B	Infix
+AB	Prefix or Polish
AB+	Postfix or reverse Polish

RPN Processing Algorithm

- Scan expression from left to right
- When you find an operator
  - Apply it to the two previous operands
  - Replace operator and two operands just used with result
- Resume left to right scan, repeat above steps until no more operators
- Works well with a stack

RPN Example

- A * B + C * D becomes A B * C D * +
- Stepwise evaluation
  - A B * C D * +
  - (A * B) C D * + where (A * B) is a single value
  - (A * B) (C * D) + where (C * D) is a single value
  - ((A * B) + (C * D)) which is a single value

Another RPN Example

- 8 * 2 + 5 * 3 becomes 8 2 * 5 3 * +
- Stepwise evaluation
  - 8 2 * 5 3 * +
  - 16 5 3 * +
  - 16 15 +
  - 31

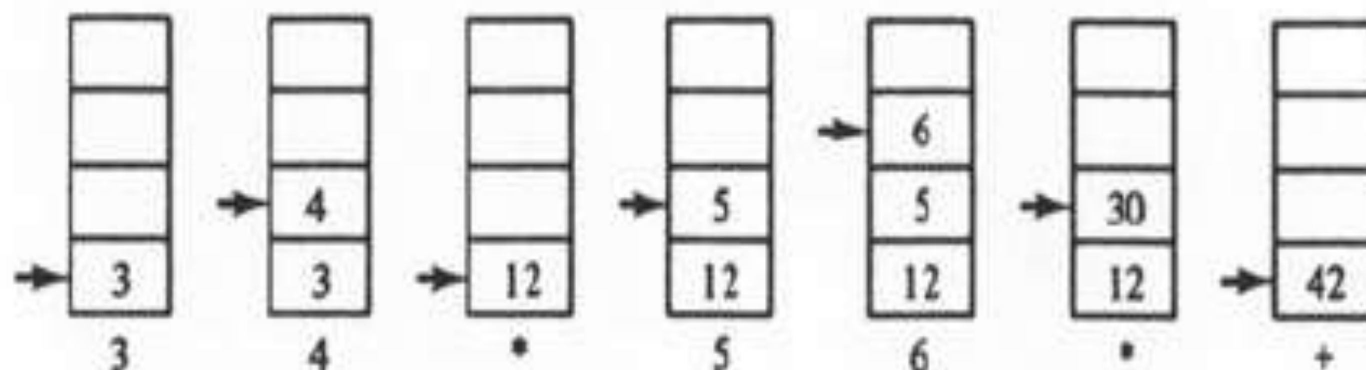
Another Infix to RP

- Infix (A + B) * (C * (D + E) + F)
- RP A B + D E + C * F + *
- RPN doesn't need or use parentheses

Stack Operations

- Infix: 3 * 4 + 5 * 6
- RP: 3 4 * 5 6 * +

Figure 8-5 Stack operations to evaluate 3 • 4 + 5 • 6.



Instruction Formats

The format of an instruction is usually depicted in a rectangular box symbolizing the bits of the instruction as they appear in memory words or in a control register. The bits of the instructions are divided into groups called fields. The most common fields found in instruction format are:-

1. An Operation code field that specifies the operation to be performed.
2. An address field that designates a memory address or a processor register.
3. A mode field that specifies the way the operand or the effective address is determined.

Computer may have instructions of several different length containing varying number of addresses. The no. of address field in the instruction format of a computer depends on the internal organization of its registers.

Types of CPU Organizations

- Single accumulator
- General register
- Stack
- Some CPUs combine features from more than one organization type

Single Accumulator

- ADD X
  - $AC \leftarrow AC + M[X]$

General Register

- ADD R1, R2, R3
  - $R1 \leftarrow R2 + R3$
- ADD R1, R2
  - $R1 \leftarrow R1 + R2$
- MOV R1, R2
  - $R1 \leftarrow R2$
- ADD R1, X
  - $R1 \leftarrow R1 + M[X]$

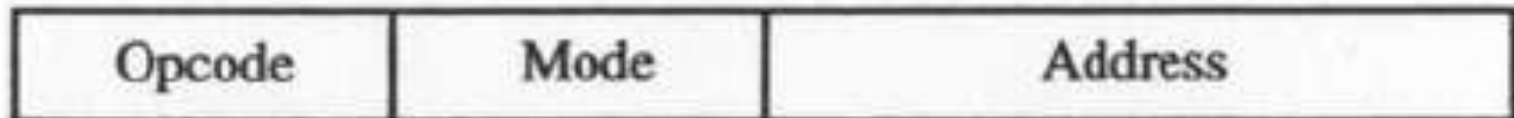
Stack

- PUSH X
- ADD
  - Zero address
  - Pop two numbers off stack
  - Add them
  - Push result back on stack

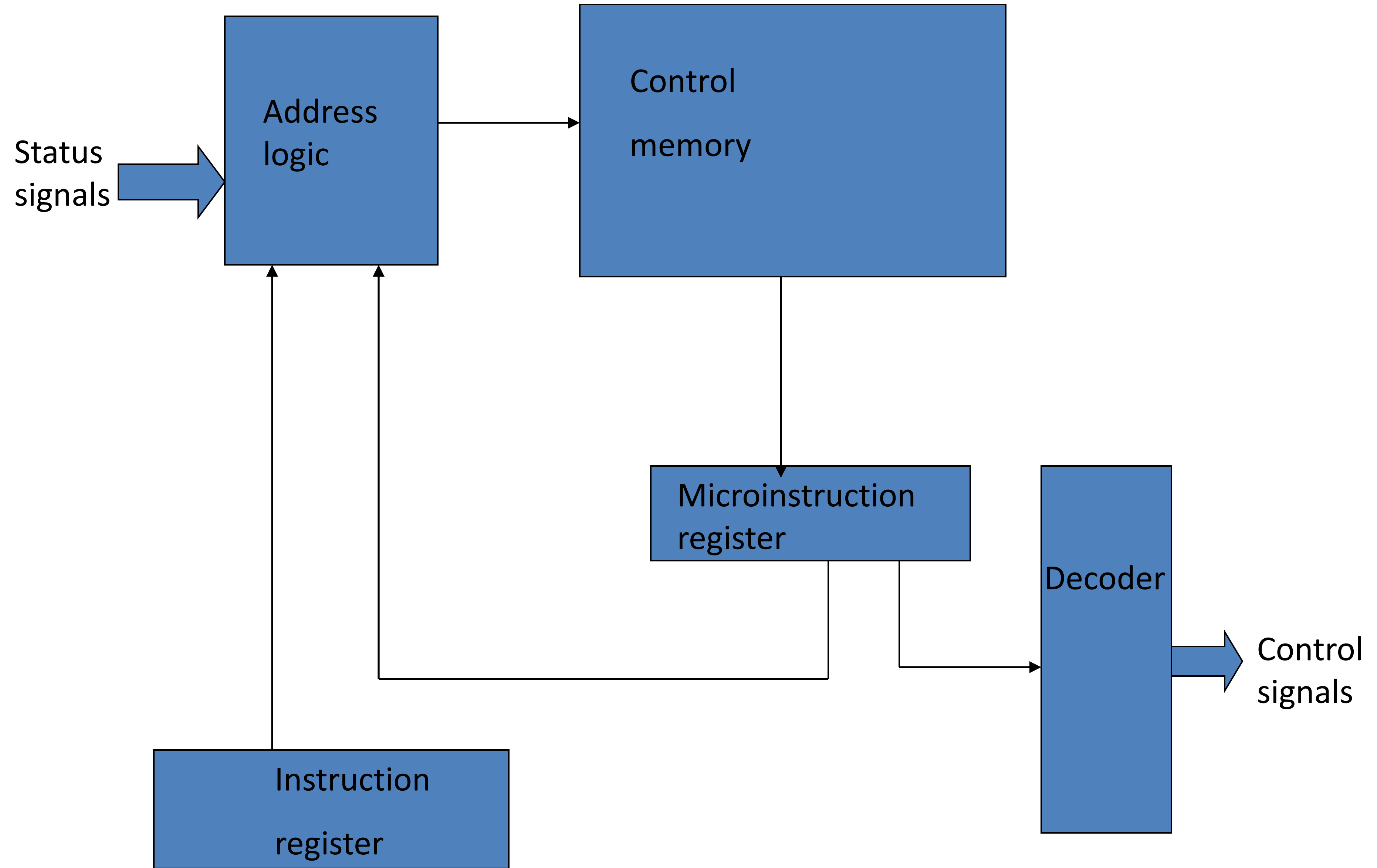
Three Address Instructions

- $X = (A + B) * (C + D)$
- ADD R1, A, B $R1 \leftarrow M[A] + M[B]$
  ADD R2, C, D $R2 \leftarrow M[C] + M[D]$
  MUL X, R1, R2 $M[X] \leftarrow R1 * R2$

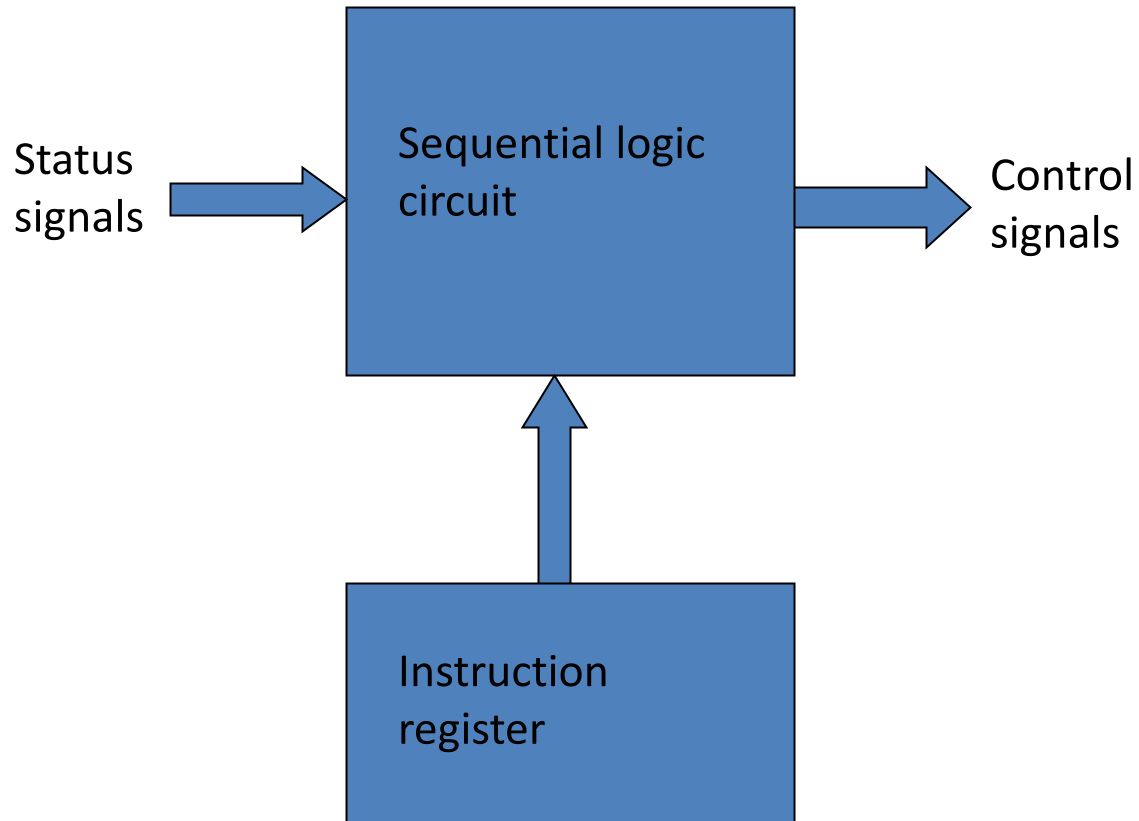
Two Address Instructions

- $X = (A + B) * (C + D)$
- ```
  MOV    R1, A
  ADD    R1, B
  MOV    R2, C
  ADD    R2, D
  MUL    R1, R2
  MOV    X, R1
  ```
  ```

One Address Instructions

- $X = (A + B) * (C + D)$
- LOAD A
  ADD B
  STORE T
  LOAD C
  ADD D
  MUL T
  STORE X

Zero Address Instructions

- $X = (A + B) * (C + D)$
- PUSH A
  PUSH B
  ADD
  PUSH C
  PUSH D
  ADD
  MUL
  POP X

Addressing Modes

Addressing Mode Techniques

- Useful for
  - Reducing the number of bits in the address field of the instruction
  - Writing programming loops, indexing data, using memory pointers, relocating programs in memory

Figure 8-6 Instruction format with mode field.



Addressing Modes

- Implied – operands specified implicitly
  - E.g., "complement accumulator"
- Immediate – operand value in address field
- Register – operand in register specified in register field
- Register Indirect – register contains indirect address
- Autoincrement or Autodecrement – like register indirect, except register value is incremented or decremented after it is used

More Addressing Modes

- Direct Address – effective address is in address part of the instruction
- Indirect Address – effective address is stored in memory location specified in address part of the instruction
- Relative Address – program counter added to address part of the instruction
- Indexed Addressing – value of index register added to address part of the instruction to yield effective address
- Base Register Addressing – similar to Indexed

Addressing Modes Example

Register	Value
PC	200
$R1$	400
XR	100
AC	

Address	Memory	
200	Load to AC	Mode
201	Address = 500	
202	Next instruction	
399	450	
400	700	
500	800	
600	900	
702	325	
800	300	

- Direct
- Immediate
- Indirect
- Relative
- Indexed
- Register
- Register Indirect
- Autoincrement
- Autodecrement

Numerical Example

Addressing Mode	Effective Address	Content of *AC*
Direct address	500	800
Immediate operand	201	500
Indirect address	800	300
Relative address	702	325
Indexed address	600	900
Register	—	400
Register indirect	400	700
Autoincrement	400	700
Autodecrement	399	450

RISC vs CISC

RISC	CISC
➢ Emphasis on hardware	➢ Emphasis on software
➢ Includes multi-clock complex instructions	➢ Single-clock, reduced instruction only
➢ Memory-to-memory: "LOAD" and "STORE“ incorporated in instructions	➢ Register to register: "LOAD" and "STORE“ are independent instructions
➢ Small code sizes, high cycles per second	➢ Low cycles per second, large code sizes
➢ Transistors used for storing complex instructions	➢ Spends more transistors on memory registers

CISC Characteristics

- The instructions in a typical CISC processor provide direct manipulation of operands residing in a memory. The major characteristics of CISC architecture are:-

1. A large number of instructions – typically from 100 to 250 instructions
2. Some instruction that perform specialized tasks and are used infrequently
3. A large variety of addressing modes – typically from 5 to 20 different modes
4. Variable – length instruction format
5. Instruction that manipulate operands in memory
6. Instructions are complex
7. Example - Pentium processors.

RISC Characteristics

- The concept of RISC architecture involves an attempt to reduce execution time by simplifying the instruction set of the computer. The major characteristics of a RISC processor are:

1. Relatively few instructions
2. Relatively few addressing modes
3. Memory access limited to load and store instructions
4. All operation done within the register of the CPU
5. Fixed length, easily decoded instruction format.
6. Single cycle instruction execution
7. Hardwired rather than Micro programmed control
8. Instructions are simple
9. Example:- Power PC.

RISC Instructions

- Only load and store instructions can reference memory
- All other instructions can only reference registers

RISC Instructions

- $X = (A + B) * (C + D)$
- ```
  LOAD   R1, A
  LOAD   R2, B
  LOAD   R3, C
  LOAD   R4, D
  ADD    R1, R1, R2
  ADD    R3, R3, R4
  MUL    R1, R1, R3
  STORE  X, R1
  ```
  ```

General structure for Microprogrammed control unit



General structure for hardwired control unit



- The hardwired approach views the controller as a sequential logic circuit or finite state machine that generates specific sequences of control signals

- **Advantage**: 1. reduces the number of components

  2. speed is fast

- **Disadvantage** : Once the unit is constructed the only way to implement changes in control unit behaviour is by redesigning the entire unit

What is pipelining?

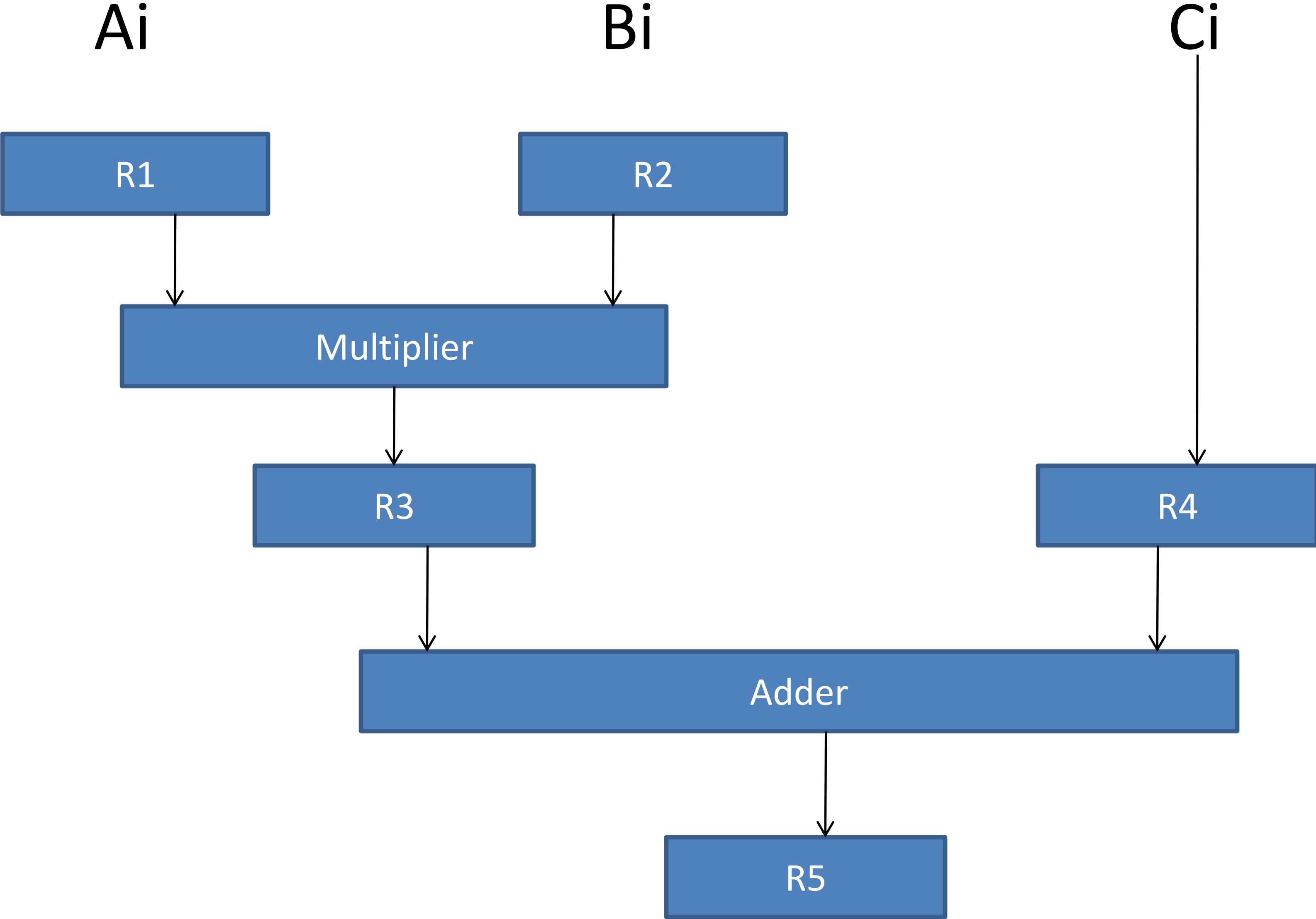
Pipelining is a technique of decomposing a sequential process into sub-operations, with each sub-process being executed in a special dedicated segments that operates concurrently with all other segments. A pipeline can be visualized as a collection of processing segments through which binary information flows. The name pipeline implies a flow of information analogous to an industrial assembly line.

Pipelining Example

- Sub-operation performed in each segment of the pipeline are as follows:-

$R1 \leftarrow Ai$, $R2 \leftarrow Bi$	Input Ai and Bi
$R3 \leftarrow R1 * R2$, $R4 \leftarrow Ci$	Multiply and input Ci
$R5 \leftarrow R3 + R4$	Add Ci to product

Pipelining Processing



Different Types of Pipelining

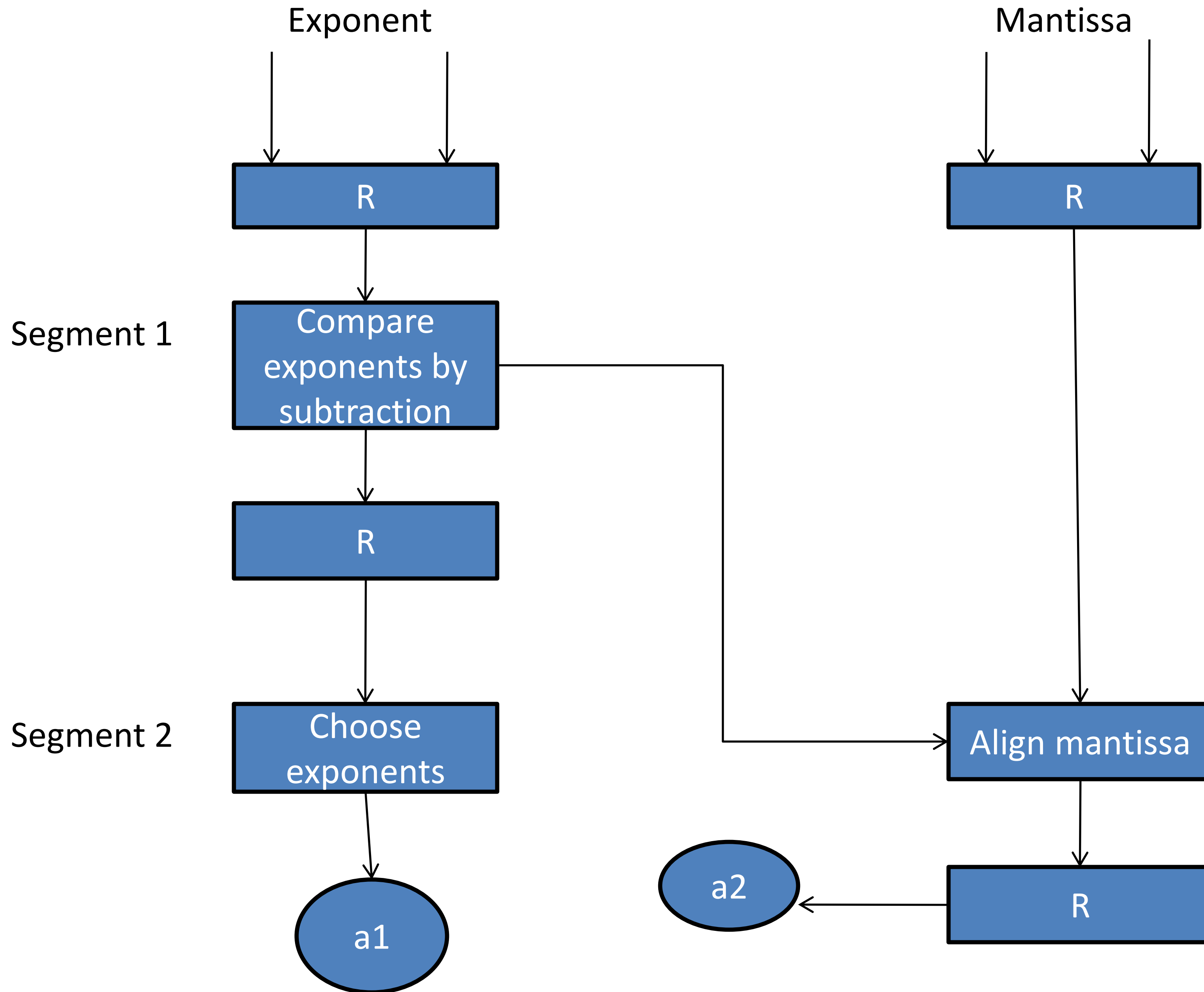
- 1 Arithmetic pipeline
- 2 Instruction pipeline
- 3 RISC pipeline
- 4 Vector processing

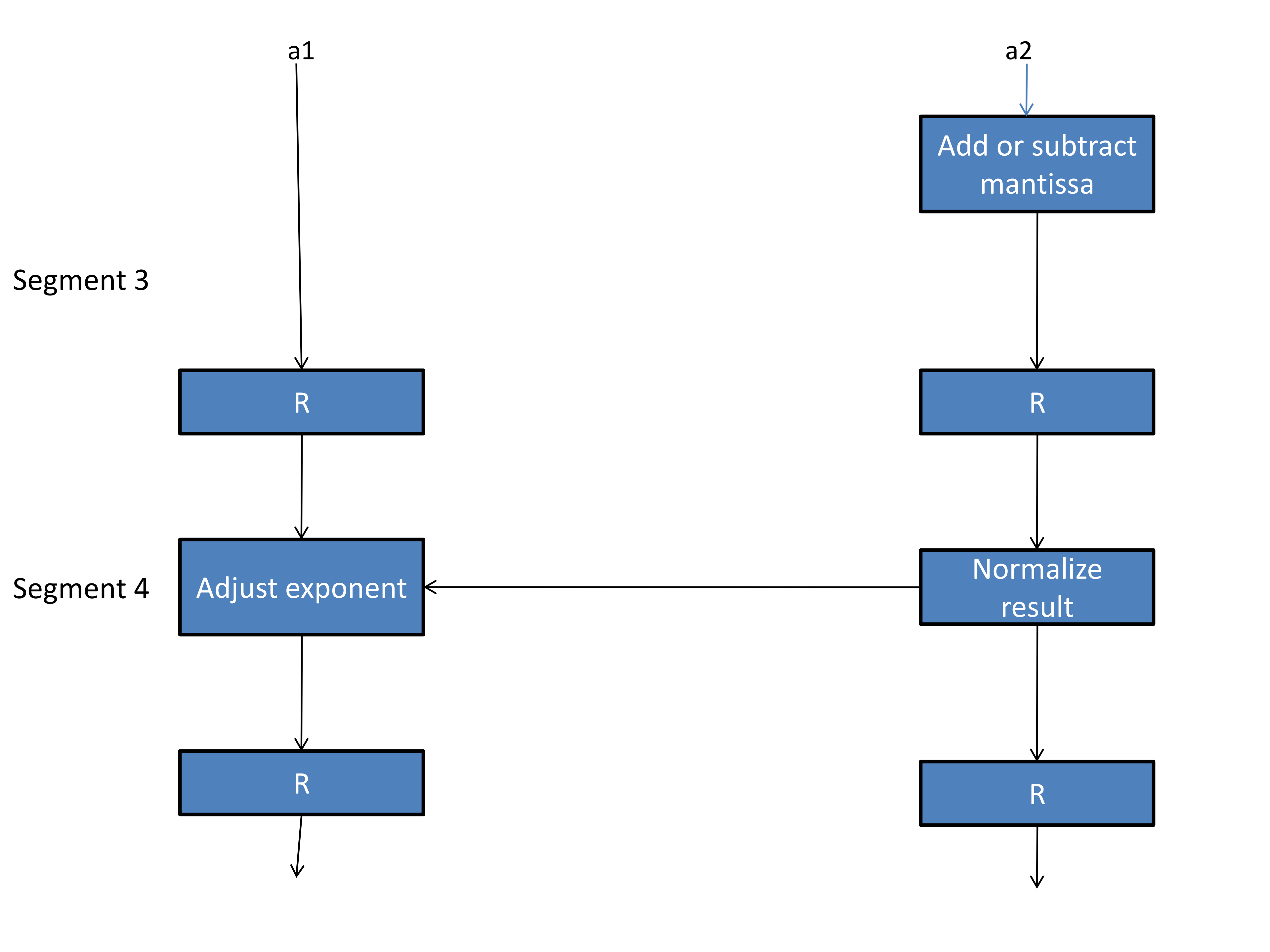
Arithmetic Pipeline

- An arithmetic pipeline divides an arithmetic operation into sub-operation for execution in the pipeline segments.

  Pipeline arithmetic units are usually found in very high speed computers. They are used to implement floating-point operations, multiplication of fixed point numbers, and similar computations encountered in specific problems.

Pipeline for floating point Addition and Subtraction




a1
a2
Add or subtract mantissa
Segment 3
R
R
Segment 4
Adjust exponent
Normalize result
R
R

Unit - 04

Computer Arithmetic
&
I/O Techniques

Digital Hardware Algorithms

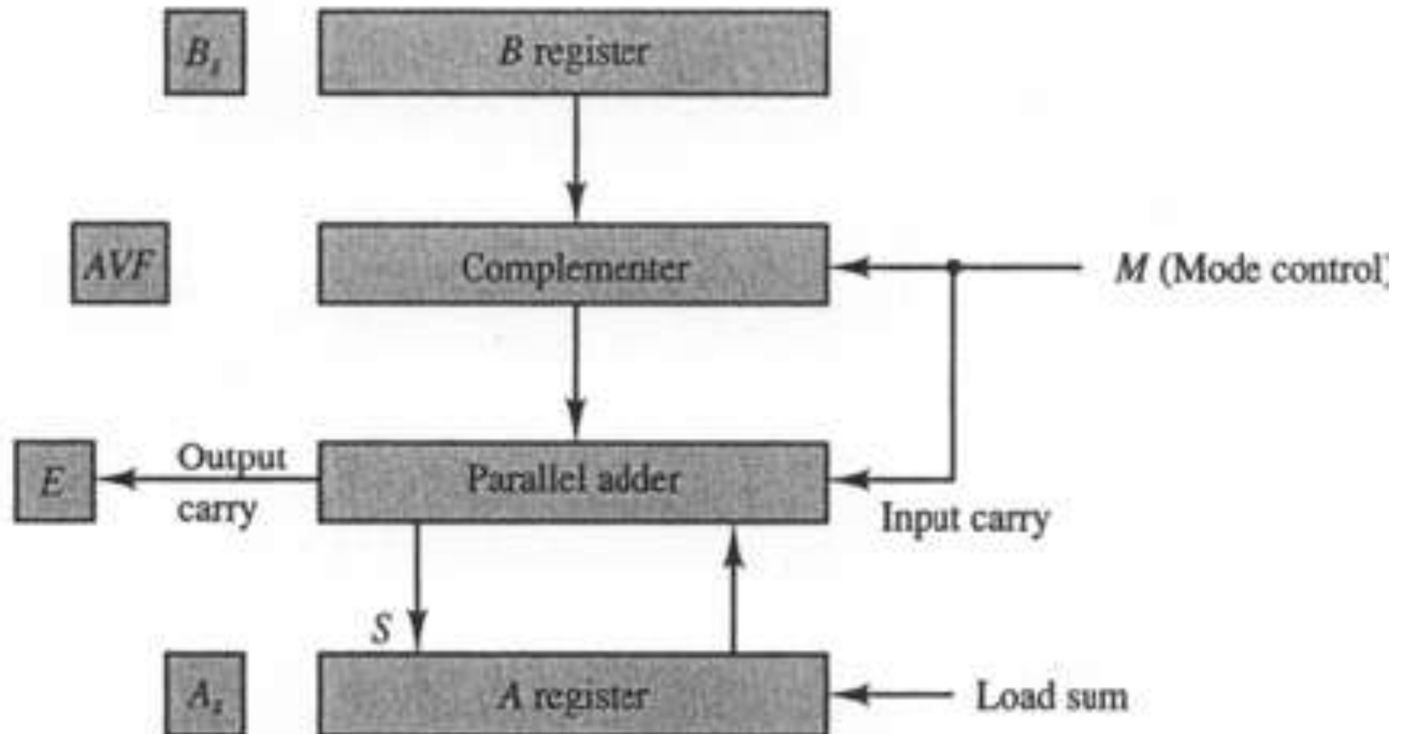
- Arithmetic operations
  - Addition, subtraction, multiplication, division
- Data types
  - Fixed-point binary
    - Signed-magnitude representation
    - Signed-2's complement representation
  - Floating-point binary
  - Binary-coded decimal (BCD)

Add / Subtract Signed-Magnitude

Operation	Add Magnitudes	Subtract Magnitudes		
		When $A > B$	When $A < B$	When $A = B$
$(+A) + (+B)$	$+(A + B)$			
$(+A) + (-B)$		$+(A - B)$	$-(B - A)$	$+(A - B)$
$(-A) + (+B)$		$-(A - B)$	$+(B - A)$	$+(A - B)$
$(-A) + (-B)$	$-(A + B)$			
$(+A) - (+B)$		$+(A - B)$	$-(B - A)$	$+(A - B)$
$(+A) - (-B)$	$+(A + B)$			
$(-A) - (+B)$	$-(A + B)$			
$(-A) - (-B)$		$-(A - B)$	$+(B - A)$	$+(A - B)$

Forces zero to be positive

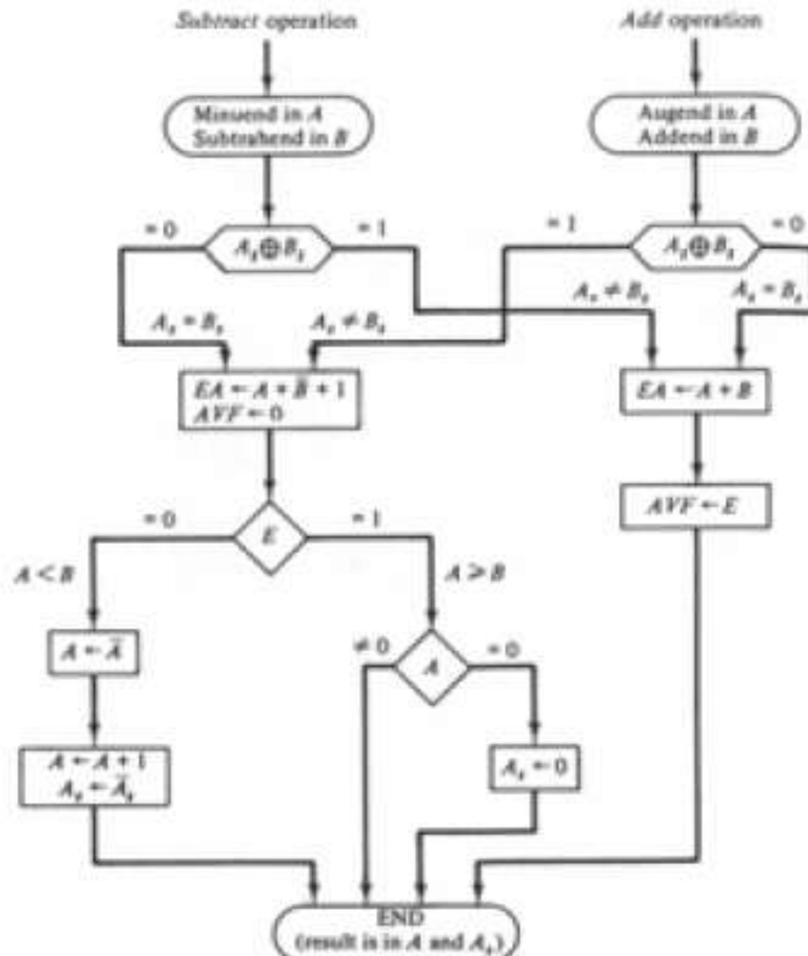
Hardware



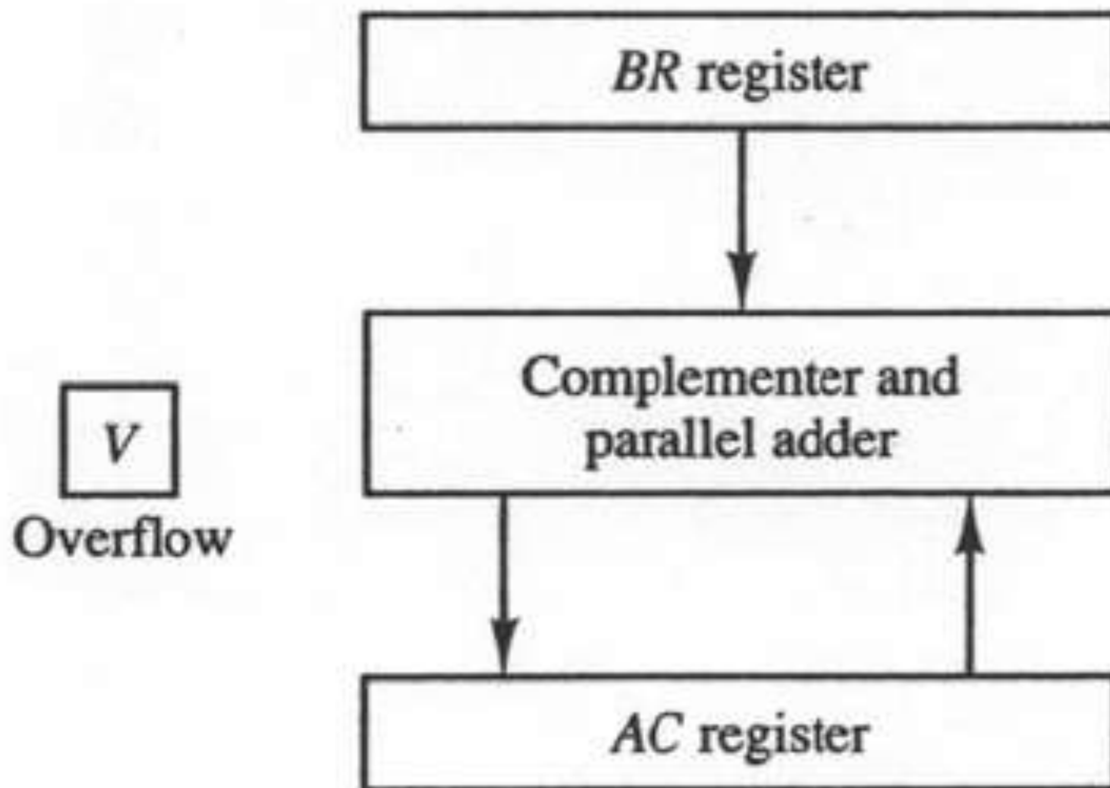
Description

- A_S Sign of A
- B_S Sign of B
- A_S & A Accumulator
- AVF Overflow bit for $A + B$
- E Output carry for parallel adder

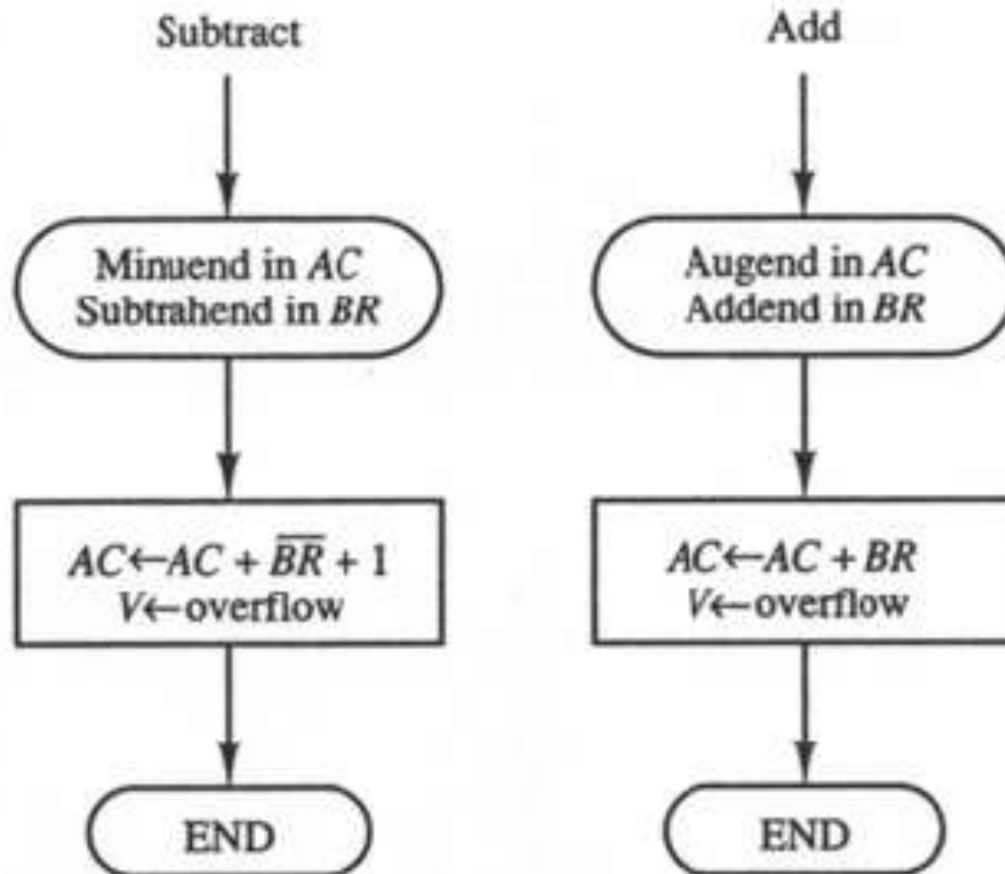
Flowchart



Add / Sub Signed-2's Complement



Algorithm

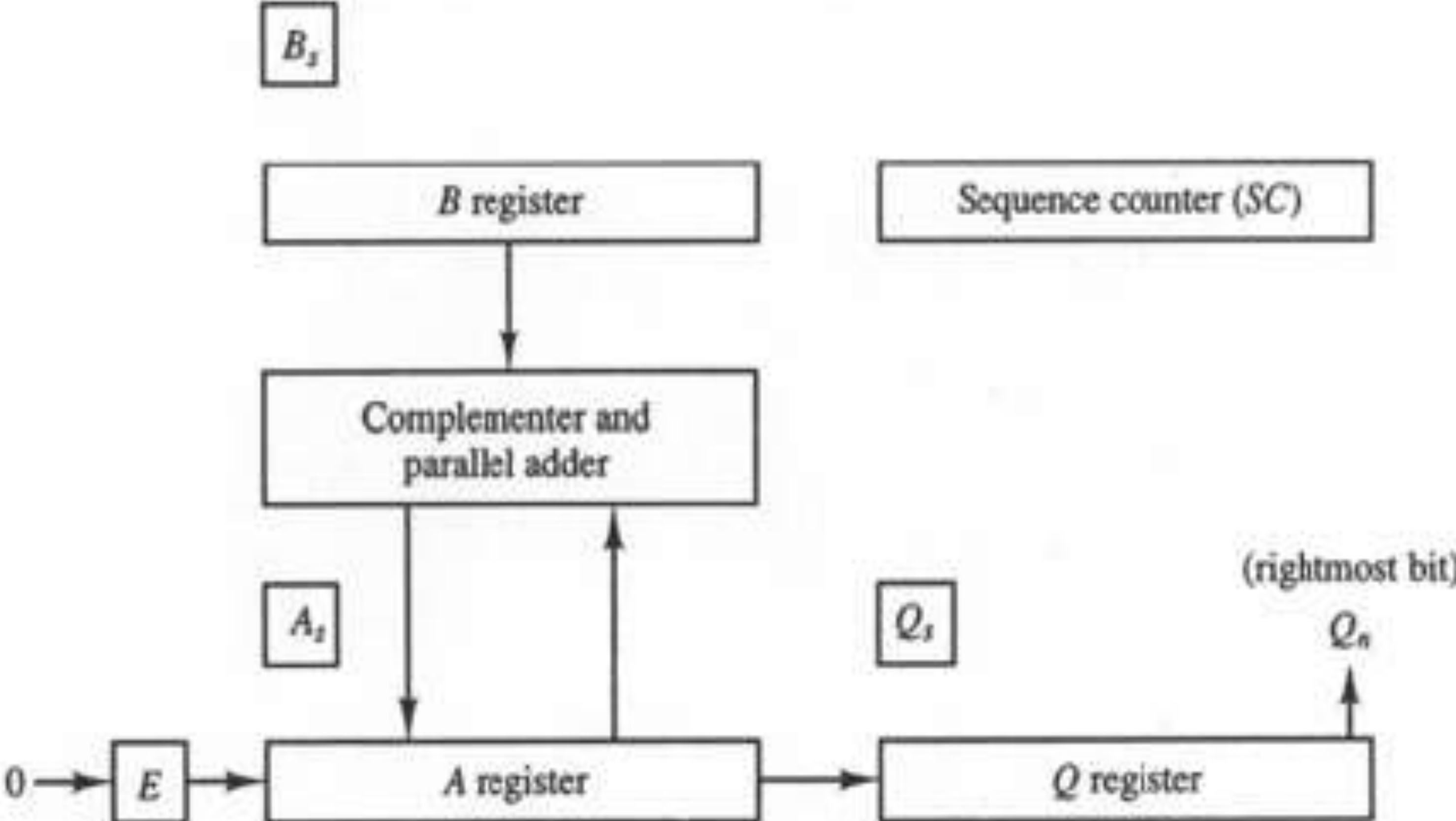


Multiply Signed-Magnitude

- Series of successive shift and add operations
- 23   10111 Multiplicand

23	10111	Multiplicand
19	x 10011	Multiplier
	10111	
	10111	
	00000	
	00000	
	10111 +	
437	110110101	Product

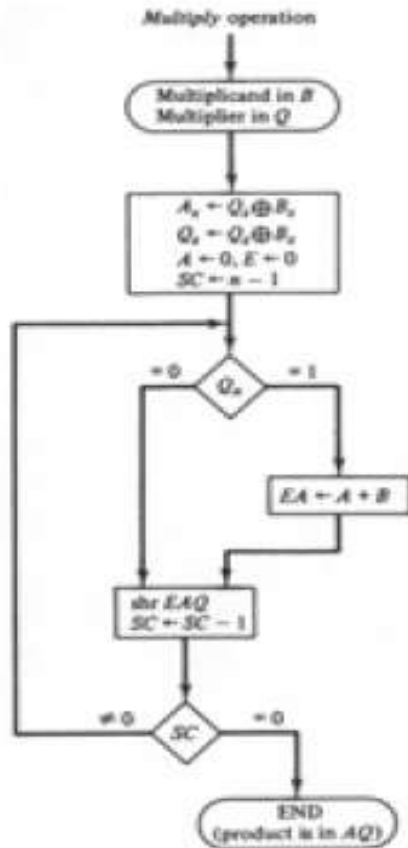
Hardware



Description

- Q multiplier
- B multiplicand
- A 0
- SC number of bits in multiplier
- E overflow bit for A
- Do SC times
  - If low-order bit of Q is 1
    - $A \leftarrow A + B$
  - Shift right EAQ
- Product is in AQ

Flowchart



Example: 23 x 19 = 437

Multiplicand $B = 10111$	E	A	Q	SC
Multiplier in Q	0	00000	10011	101
$Q_n = 1$; add B		10111		
First partial product	0	10111		
Shift right EAQ	0	01011	11001	100
$Q_n = 1$; add B		10111		
Second partial product	1	00010		
Shift right EAQ	0	10001	01100	011
$Q_n = 0$; shift right EAQ	0	01000	10110	010
$Q_n = 0$; shift right EAQ	0	00100	01011	001
$Q_n = 1$; add B		10111		
Fifth partial product	0	11011		
Shift right EAQ	0	01101	10101	000
Final product in $AQ = 0110110101$				

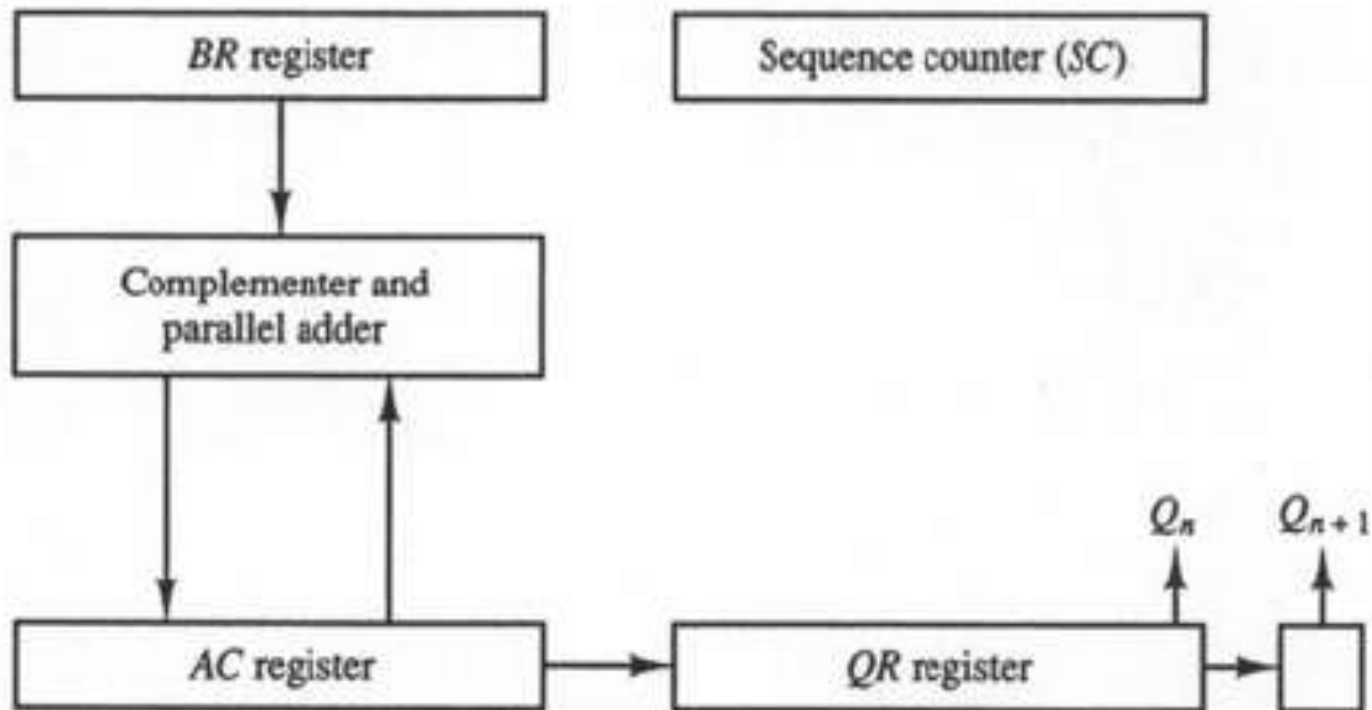
Multiply Signed-2's Complement

- Booth algorithm
- QR multiplier
- Q_n least significant bit of QR
- Q_{n+1} previous least significant bit of QR
- BR multiplicand
- AC 0
- SC number of bits in multiplier

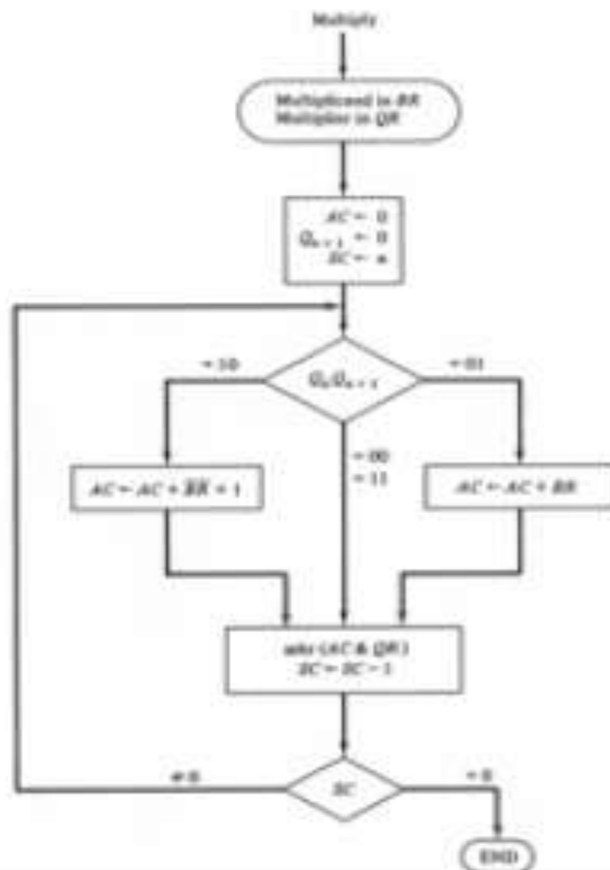
Algorithm

- Do $SC + 1$ times
  - $Q_n Q_{n+1} = 10$
    - $AC \leftarrow AC + \overline{BR} + 1$
  - $Q_n Q_{n+1} = 01$
    - $AC \leftarrow AC + BR$
  - Arithmetic shift right AC & QR
  - $SC \leftarrow SC - 1$

Hardware



Flowchart



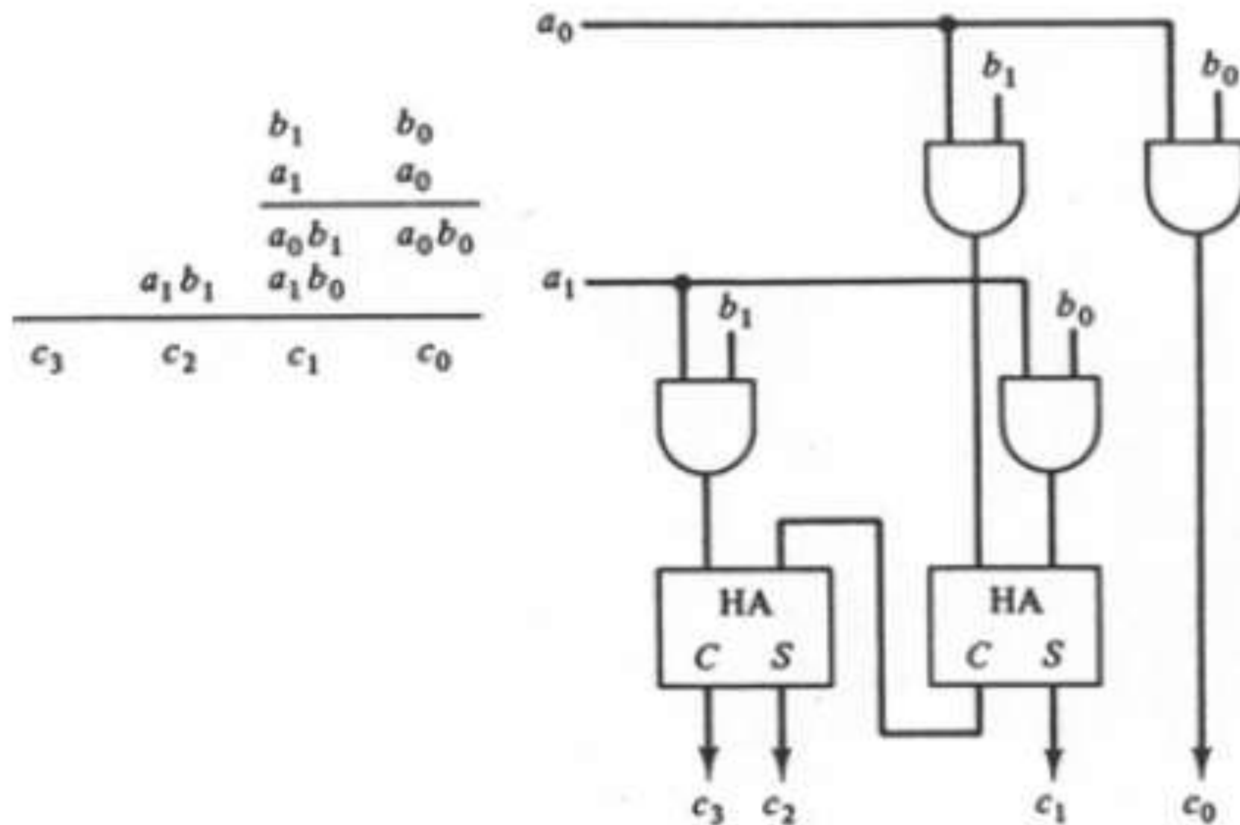
Example: -9 x -13 = 117

$Q_n Q_{n+1}$	$BR = 10111$ $\overline{BR} + 1 = 01001$	AC	QR	Q_{n+1}	SC
	Initial	00000	10011	0	101
1 0	Subtract BR	01001			
		01001			
	ashr	00100	11001	1	100
1 1	ashr	00010	01100	1	011
0 1	Add BR	10111			
		11001			
	ashr	11100	10110	0	010
0 0	ashr	11110	01011	0	001
1 0	Subtract BR	01001			
		00111			
	ashr	00011	10101	1	000

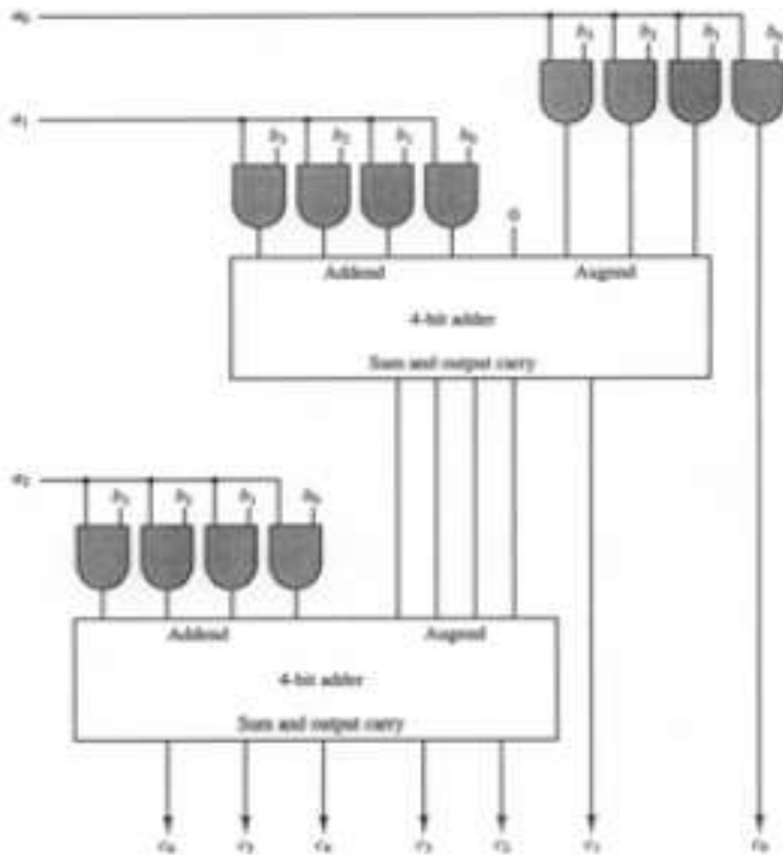
Array Multiplier

- Combination circuit
- Product generated in one microoperation
- Requires large number of gates
- Became feasible after integrated circuits developed
- Needed for j multiplier and k multiplicand bits
  - $j \times k$ AND gates
  - $j - 1$ k-bit adders to produce product of $j + k$ bits

2-bit by 2-bit Array Multiplier



4-bit by 3-bit Array Multiplier



Divide Fixed-Point Signed-Mag

- Series of successive compare, shift, and subtract operations

Divisor:	11010	Quotient = Q
B = 10001	)0111000000	Dividend = A
	01110	5 bits of $A < B$, quotient has 5 bits
	011100	6 bits of $A \geqslant B$
	-10001	Shift right B and subtract; enter 1 in Q
	-010110	7 bits of remainder $\geqslant B$
	--10001	Shift right B and subtract; enter 1 in Q
	--001010	Remainder $< B$; enter 0 in Q; shift right B
	---010100	Remainder $\geqslant B$
	----10001	Shift right B and subtract; enter 1 in Q
	----000110	Remainder $< B$; enter 0 in Q
	-----00110	Final remainder

Example: 448 / 17 = 26 r 6

Divisor $B = 10001$, $\bar{B} + 1 = 01111$

	E	A	Q	SC
Dividend:		01110	00000	5
shl EAQ	0	11100	00000	
add $\bar{B} + 1$		01111		
$E = 1$	1	01011		
Set $Q_n = 1$	1	01011	00001	4
shl EAQ	0	10110	00010	
Add $\bar{B} + 1$		01111		
$E = 1$	1	00101		
Set $Q_n = 1$	1	00101	00011	3
shl EAQ	0	01010	00110	
Add $\bar{B} + 1$		01111		
$E = 0$; leave $Q_n = 0$	0	11001	00110	
Add B		10001		
Restore remainder	1	01010		2
shl EAQ	0	10100	01100	
Add $\bar{B} + 1$		01111		
$E = 1$	1	00011		
Set $Q_n = 1$	1	00011	01101	1
shl EAQ	0	00110	11010	
Add $\bar{B} + 1$		01111		
$E = 0$; leave $Q_n = 0$	0	10101	11010	
Add B		10001		
Restore remainder	1	00110	11010	0
Neglect E				
Remainder in A:		00110		
Quotient in Q:			11010	

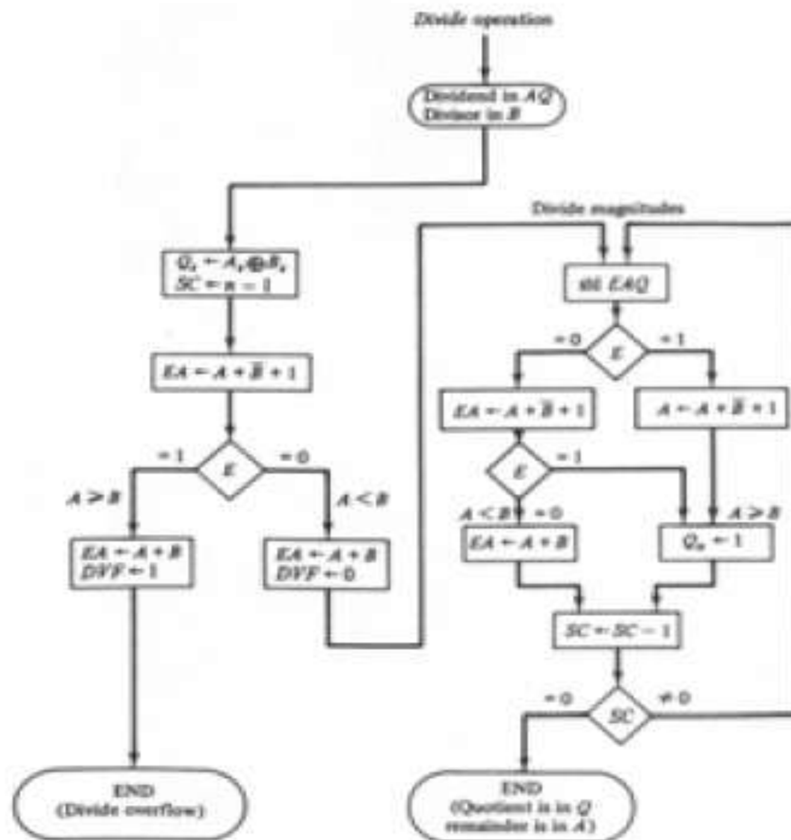
Initially,

AQ	dividend
B	divisor

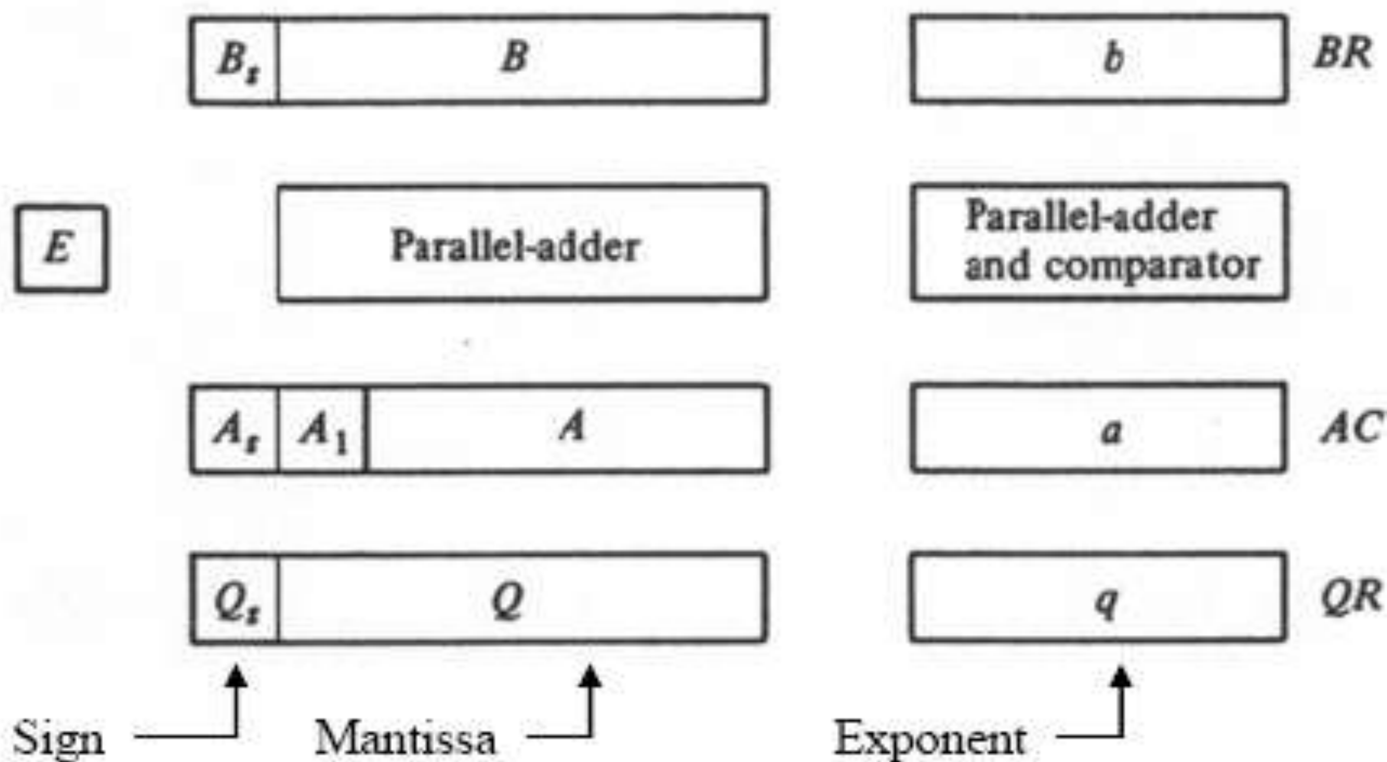
At end of operation,

Q	quotient
A	remainder
DVF	divide overflow

Algorithm



Floating-Point Registers



Signed-magnitude mantissa & biased exponent

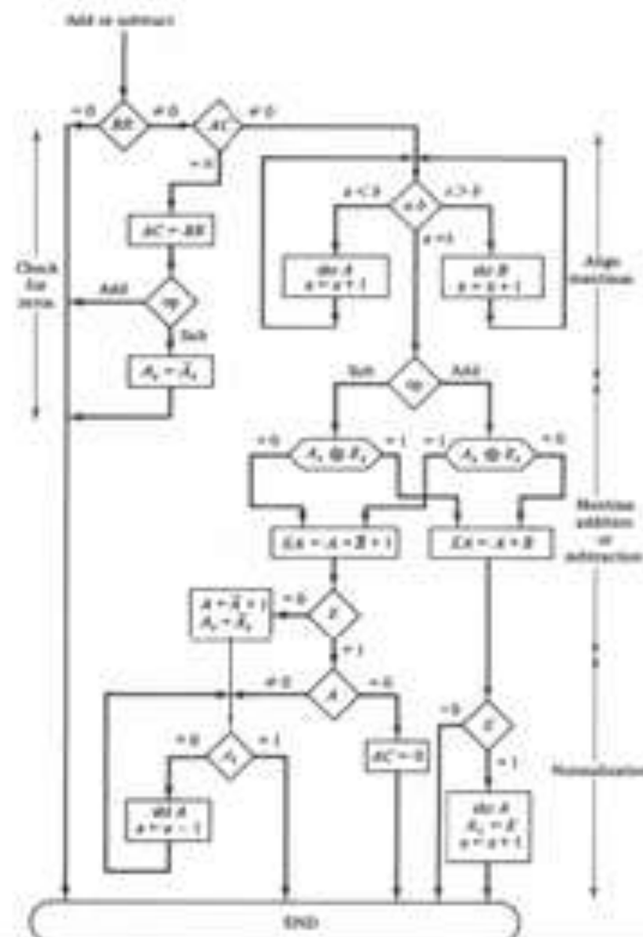
Biased Exponent

- Example
  - Real exponent range is -50 to +49
  - Add bias of 50 for new range of 0 to 99
  - Biased exponent is always a positive number
    - Easier to deal with

Floating-Point Add / Subtract

- Check for zeros
- Align the mantissas
- Add or subtract the mantissas
- Normalize the result

F-P Add / Subtract Flowchart



$$AC \leftarrow AC + BR$$

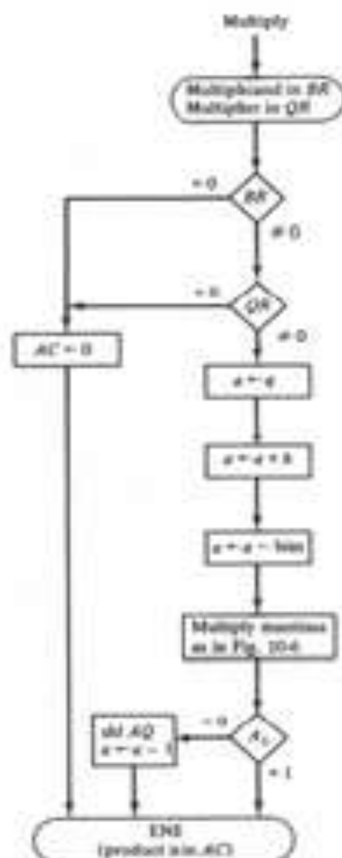
or

$$AC \leftarrow AC - BR$$

Floating-Point Multiply

- Check for zeros
- Add the exponents
- Multiply the mantissas
- Normalize the product

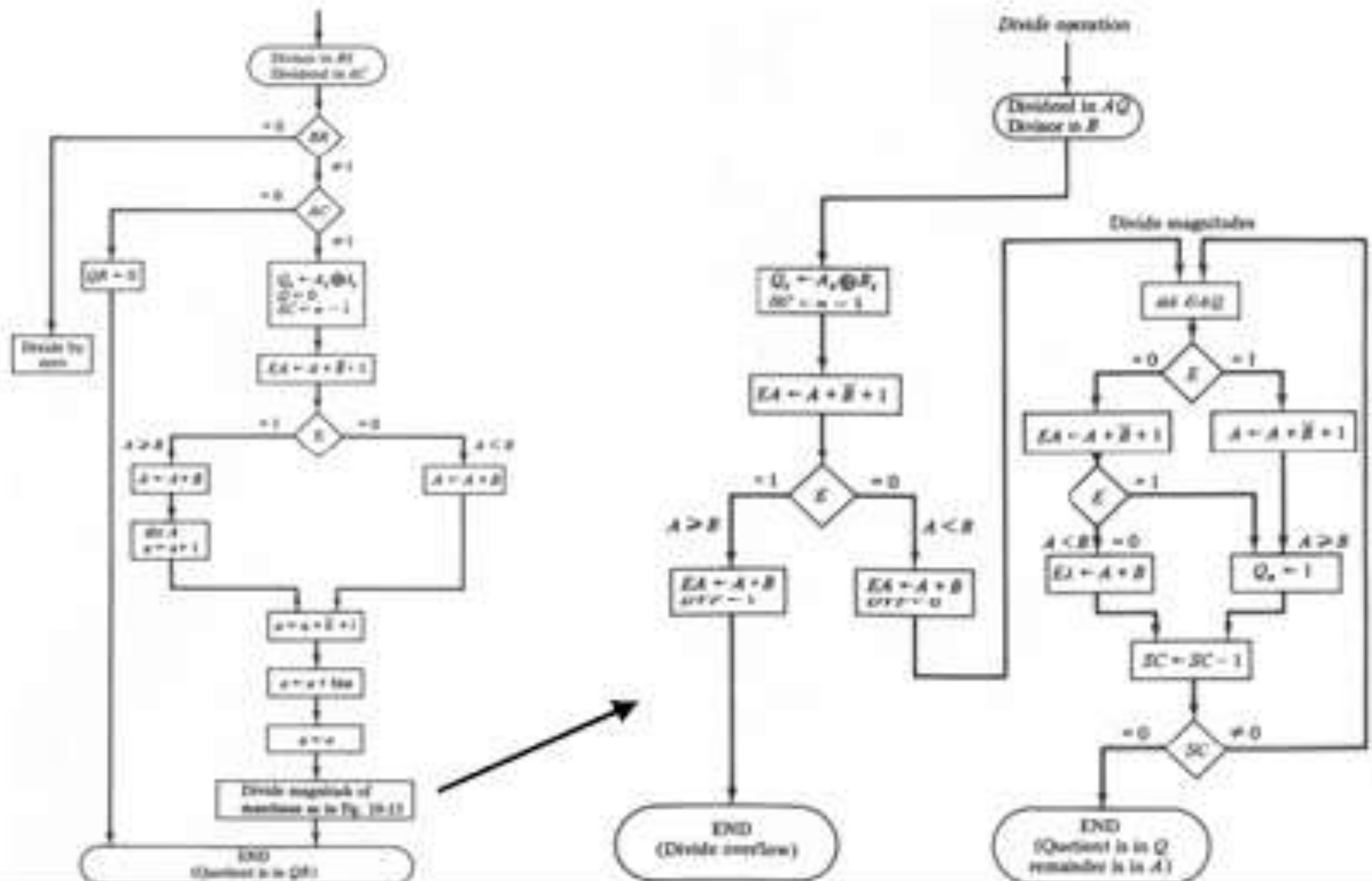
F-P Multiply Flowchart



Floating-Point Division

- Check for zeros
- Initialize registers and evaluate the sign
- Align the dividend
- Subtract the exponents
- Divide the mantissas

F-P Division Flowchart



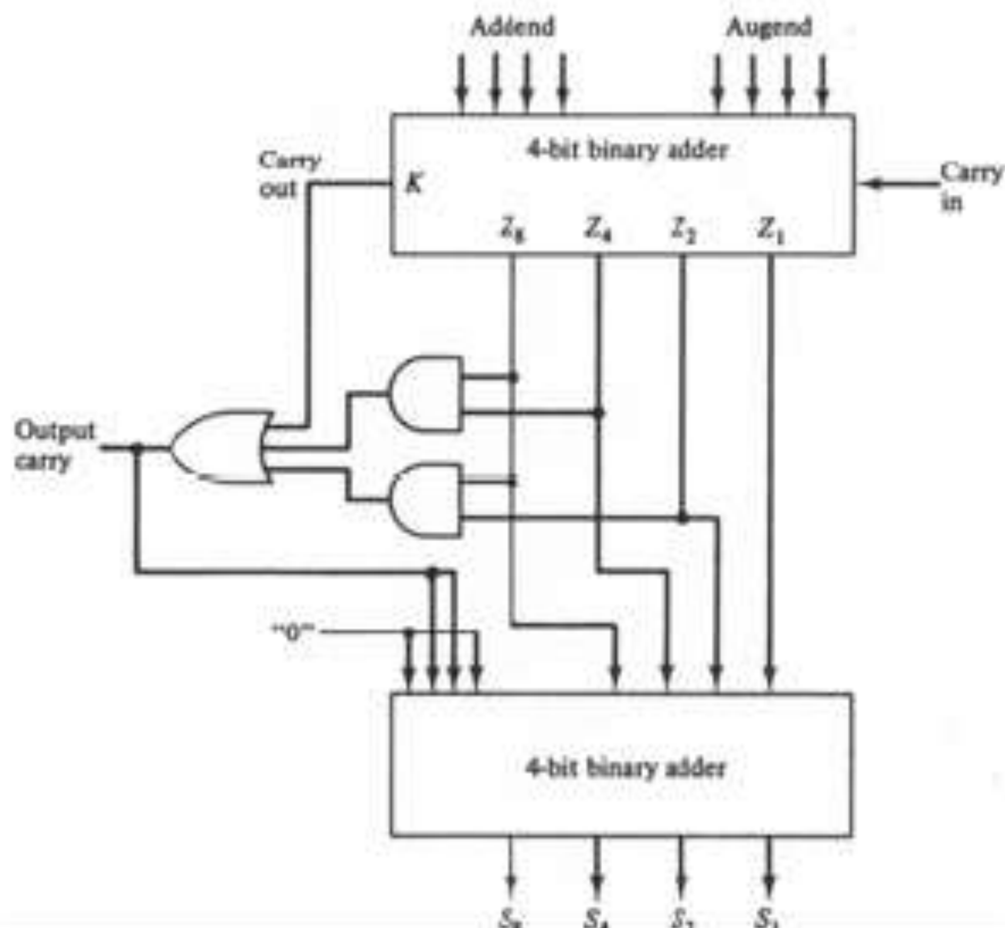
Booth Multiplication Algorithm

- Zeros in multiplier require no addition
  - But shifting still required
- String of 1s in the multiplier from weight 2^k to 2^m can be rewritten as $2^{k+1} - 2^m$
  - Example: 001110 [+14]
    - String of 1s from 2^3 to 2^1: $2^4 - 2^1 = 16 - 2 = 14$
    - Multiplicand M: $M \times 14 = M \times 2^4 - M \times 2^1$
    - Product obtained by M 4 times to the left and subtracting M shifted left once

BCD Adder

- Output can't exceed 9 + 9 + 1 = 19
- If binary sum in BCD digit > 1001, add 0110
- Given
  - Output of binary adder is $Z_8Z_4Z_2Z_1$
  - Output carry K
  - BCD output carry $C = K + Z_8Z_4 + Z_8Z_2$

Block Diagram BCD Adder



Examples

- Example calculations:

Decimal	Binary	Decimal	Binary	Decimal	Binary
9	1001	9	1001	6	0110
7	0111	9	1001	4	0100
16	1 0000	18	1 0010	10	1010
	0110		0110		0110
	0110		1000		1 0000

BCD Subtraction

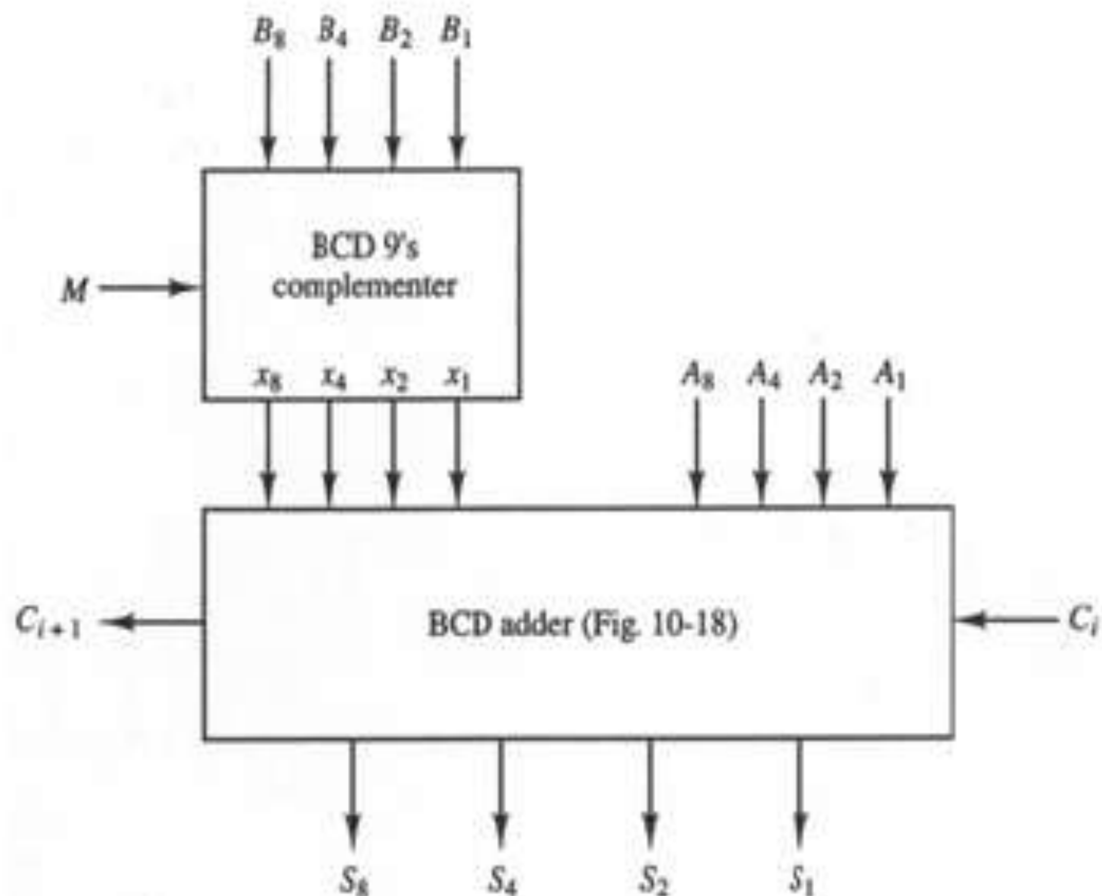
- Subtract by adding 9s complement of subtrahend to minuend
- First 9s complement algorithm
  - Complement bits
  - Add 1010 (decimal 10) and discard carry
- Second 9s complement algorithm
  - Add 0110 (decimal 6)
  - Complement bits

Examples

- 0111 decimal 7
  1000 complement
  +1010 decimal 10
  1 0010 decimal 2

  0111
  + 0110 add decimal 6
  1101
  0010 complement

Stage of Decimal Arithmetic Unit

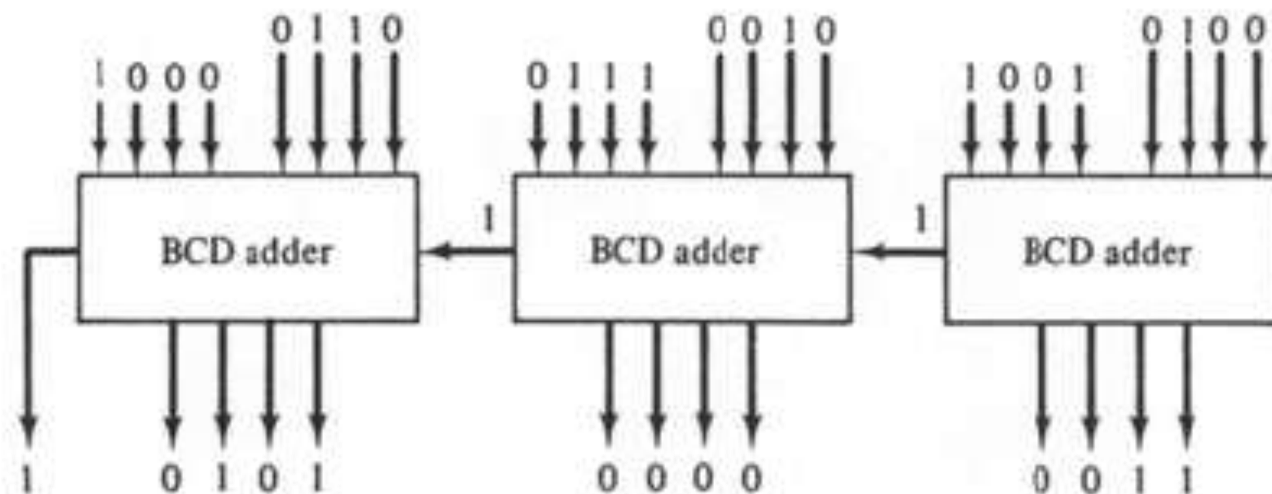


Decimal Arithmetic Microops

TABLE 10-5 Decimal Arithmetic Microoperation Symbols

Symbolic Designation	Description
$A \leftarrow A + B$	Add decimal numbers and transfer sum into A
$\overline{B}$	9's complement of B
$A \leftarrow A + \overline{B} + 1$	Content of A plus 10's complement of B into A
$Q_L \leftarrow Q_L + 1$	Increment BCD number in Q_L
dshr A	Decimal shift-right register A
dshl A	Decimal shift-left register A

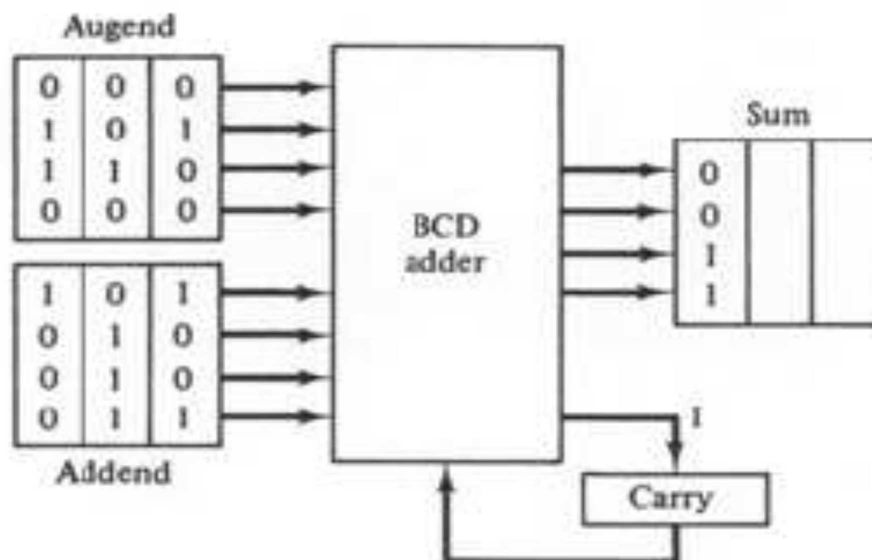
Parallel Decimal Addition



(a) Parallel decimal addition: 624 + 879 = 1503

Fast

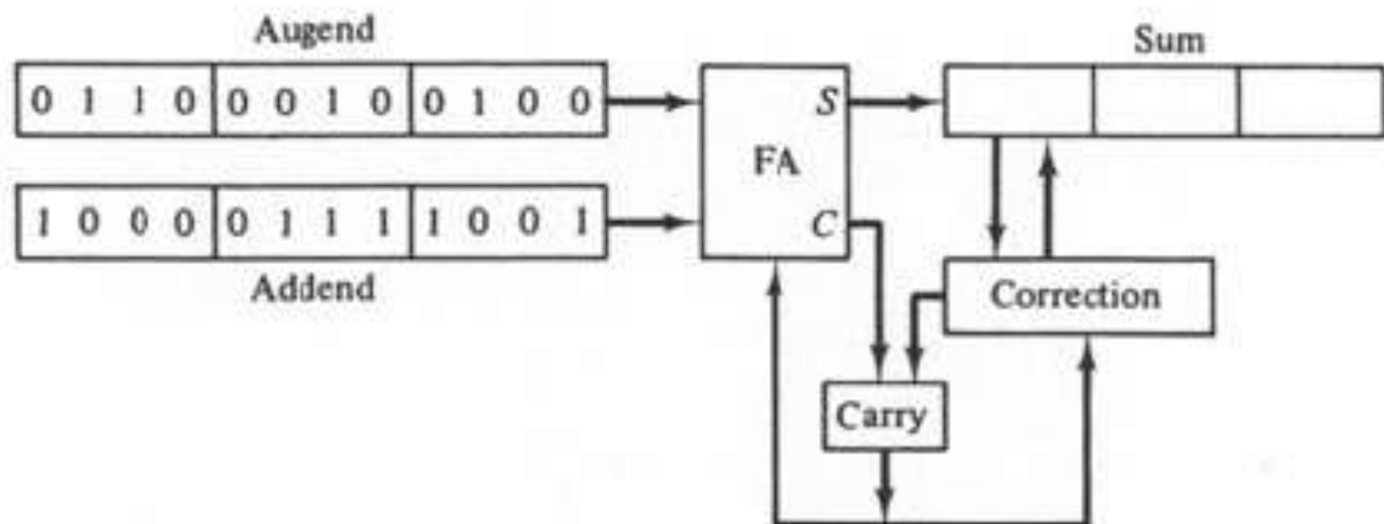
Digit-Serial, Bit-Parallel Dec Add



(b) Digit-serial, bit-parallel decimal addition

Slow

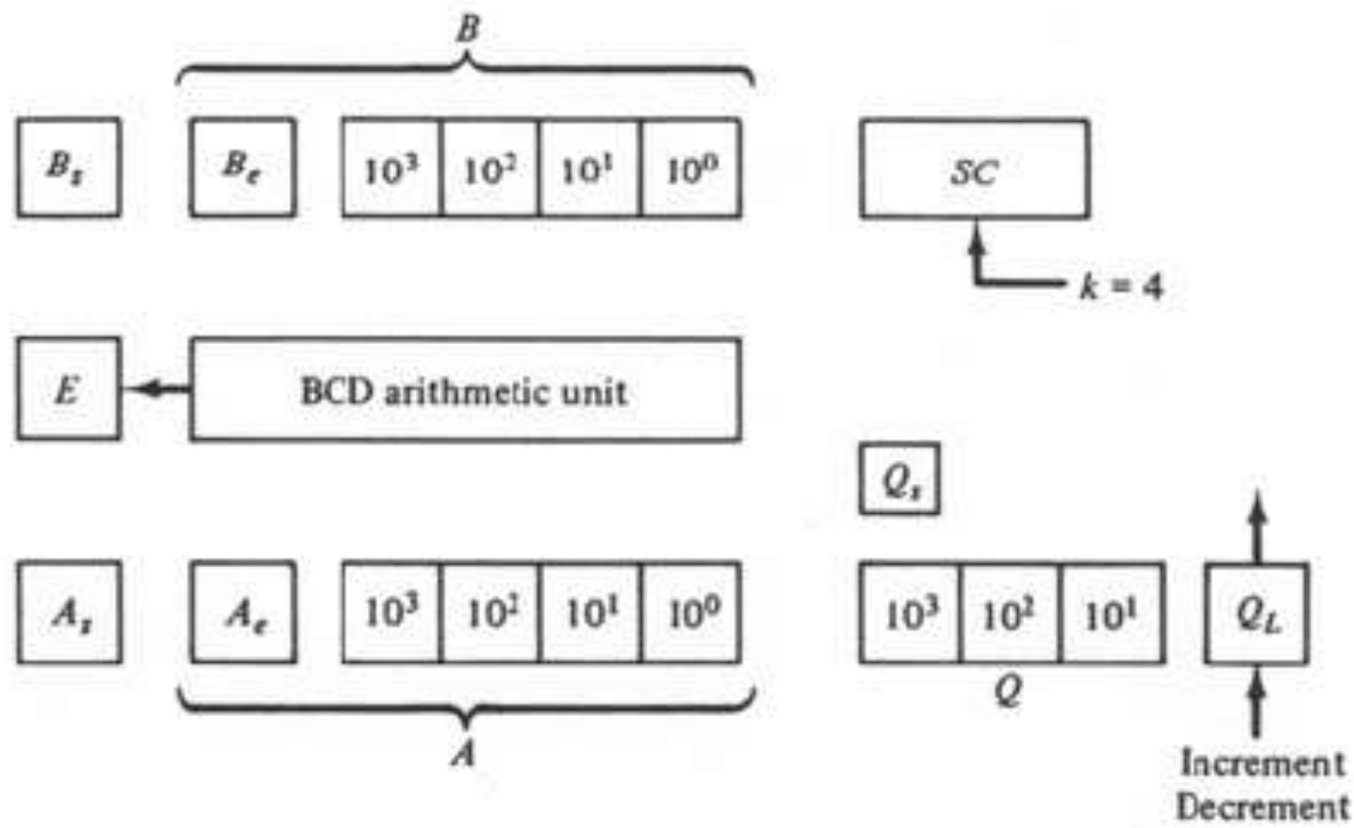
All Serial Decimal Addition



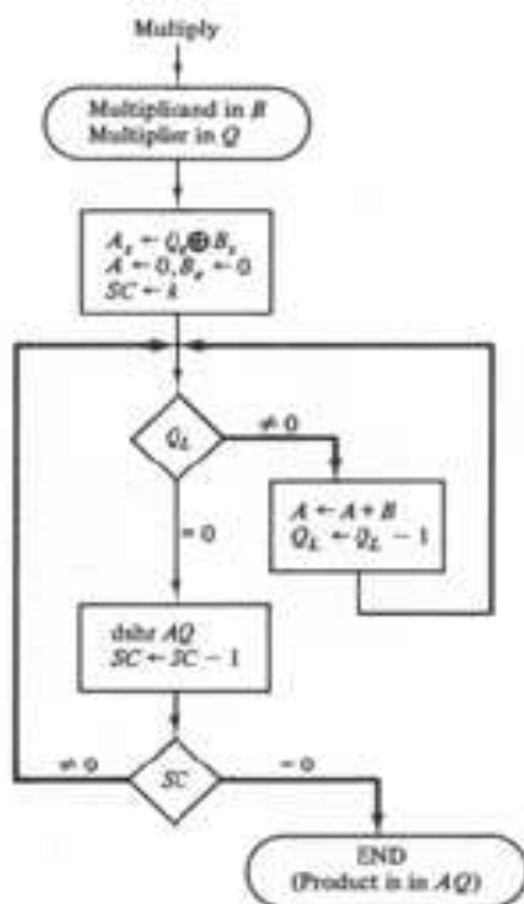
(c) All serial decimal addition

Very slow

Dec Arith Registers for Mult & Div

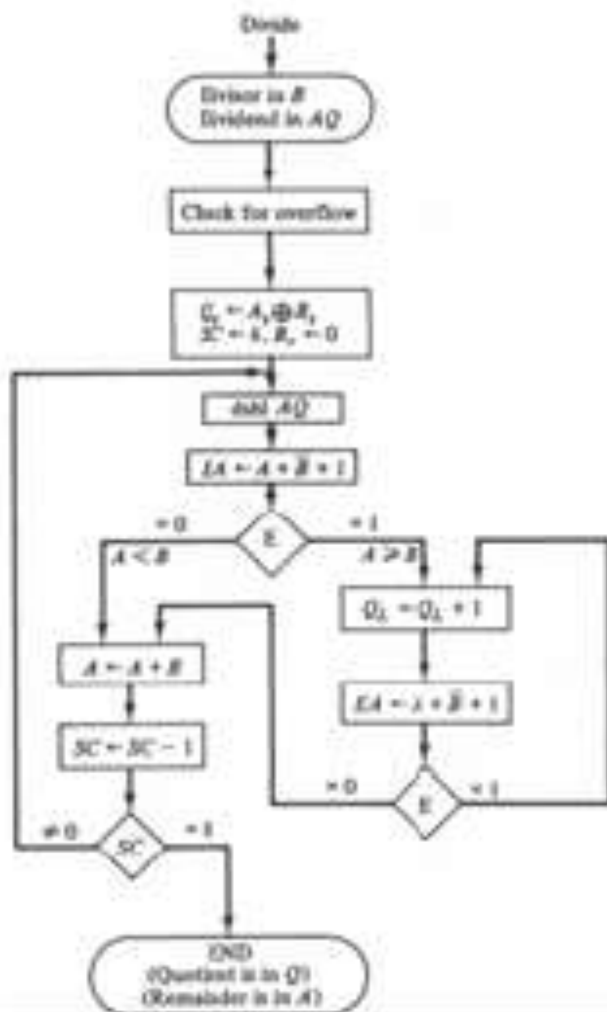


Decimal Multiplication Flowchart



0456	0456	
x 0123	0456	
56088	+ 0456	
	1368	
	0136 8	shift
	0456	
	+ 0456	
	1048	
	0104 88	shift
	+ 0456	
	0560	
	0056 088	shift
	0005 6088	shift

Decimal Division Flowchart



Combinational ALU

The simplest ALUs combine the functions of a twos-complement adder-subtracter with those of a circuit that generates word-based logic functions of the form $f(X,Y)$, for example, AND, XOR, and NOT. They can thus implement most of a CPU's fixed-point data-processing instructions. Figure 4.28 outlines an ALU that has separate subunits for logical and arithmetic operations. The particular class of operation (logical and arithmetic) to be performed is determined by a "mode" control line M attached to a two-way multiplexer that channels the required result to the output bus Z. The specific operation performed by the desired subunit is determined by a "select" control line S as shown. The ALU's logical operations are performed bitwise; that is, the same operation f is applied to every pair of data lines x_i,y_i. The maximum number of distinct logical operations of the form $f(x_i,y_i)$ is 16, which is the number of distinct truth tables of two Boolean variables. Hence the select bus S needs to be of size 4 at most, as in Figure 4.28. S can also be used to select up to 16 different arithmetic operations such as $X + Y$, $X - Y$, $Y - X$, $X + 1$ (increment), $X - 1$ (decrement), and so on, as needed.

Combinational ALU

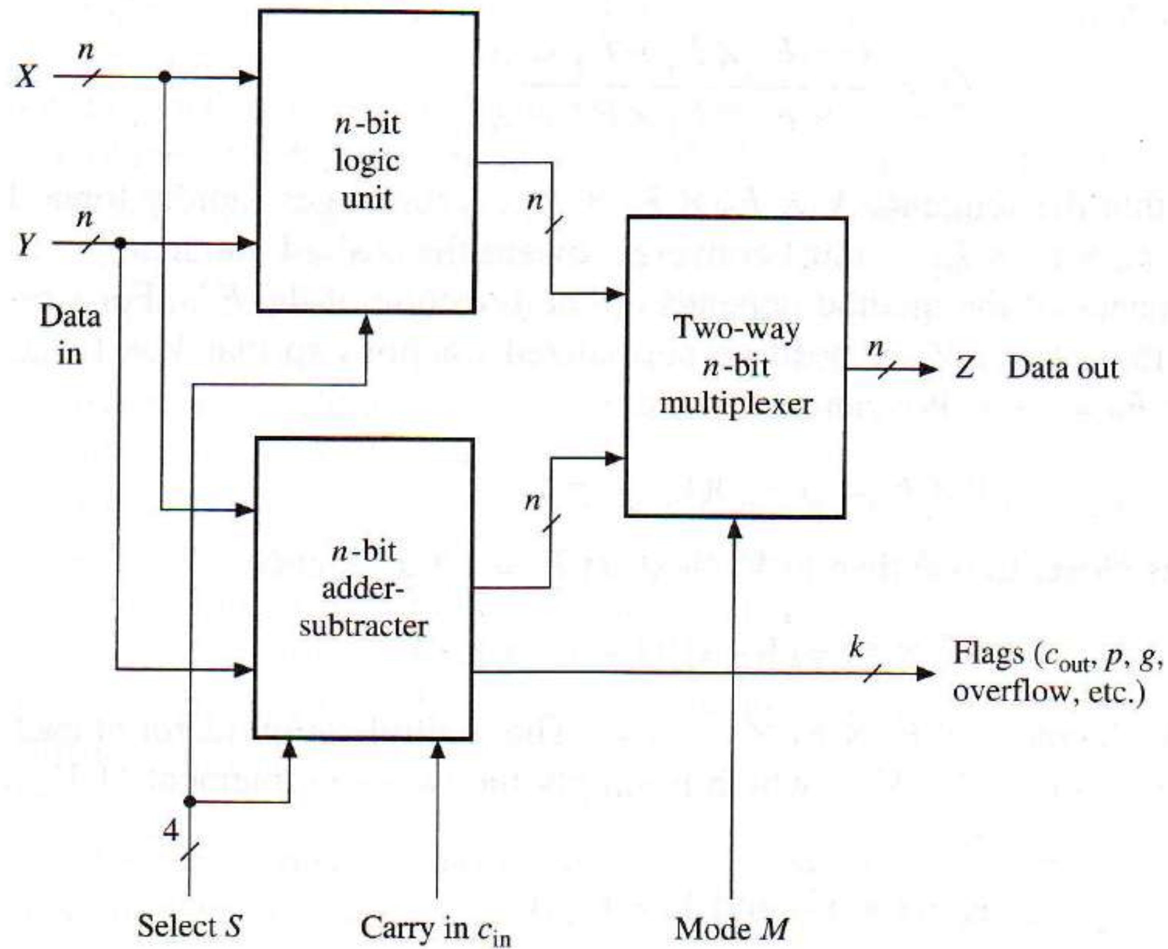


Figure 4.28
A basic n-bit arithmetic-logic unit (ALU).

Combinational ALU

The logical operations in Figure 4.28 can be obtained by generating all four minterms of $f(x_i,y_i)$, namely,

$$m_3 = x_i y_i \quad m_2 = x_i \overline{y}_i \quad m_1 = \overline{x}_i y_i \quad m_0 = \overline{y}_i \, \overline{y}_i$$

for every pair $x_i y_i$ of data bits and by using the control lines $S = S_3S_2S_1S_0$ to select desired subsets of the minterms to be ORed together. In particular, if we construct the sum-of-products expression

$$\begin{aligned} f(x_i,y_i) &= m_3S_3 + m_2S_2 + m_1S_1 + m_0S_0 \\ &= x_i y_i S_3 + x \overline{y}_i S_2 + \overline{x}_i y_i S_1 + \overline{x}_i \, \overline{y}_i S_0 \end{aligned} \tag{4.32}$$

then we see that every combination of $S_3S_2S_1S_0$ produces a different function. For example, $S = 0110$ makes $f(x_i,y_i) = x_i\overline{y}_i + \overline{x}_i y_i$, which is EXCLUSIVE-OR. Because of the bitwise nature of the logic operations, we can replace x_i and y_i in (4.32) with the n-bit words X and Y.

$$f(X,Y) = XYS_3 + X\overline{Y}S_2 + \overline{X}YS_1 + \overline{X}\,\overline{Y}S_0 \tag{4.33}$$

We can now implement the logic unit directly from Equation (4.33), using several n-bit word gates as in Figure 4.29. The adder-subtracter can be designed by any of the techniques presented earlier, with appropriate additional connections to X, Y, and S.

Combinational ALU

Despite its conceptual simplicity, the ALU of Figure 4.28 is more expensive and slower than necessary. For $n = 4$, the logic subunit employs about 25 gates and inverters. If the arithmetic subunit is designed with carry lookahead in the style of Figure 4.6, around 60 gates are needed, depending on the variants of add and subtract that are implemented. The multiplexer in Figure 4.28 also requires additional

gates. The complete 4-bit ALU can therefore be expected to contain more than 100 gates of various kinds and have depth 9 or so. By judicious sharing of functions between the two main subunits, both of these figures can be reduced by a third, as the next example shows.

Sequential ALU

Although, as we have seen, both multiplication and division can be implemented by combinational logic, it is generally impractical to merge these operations with addition and subtraction into a single, combinational ALU. The reason is twofold. Combinational multipliers and dividers are costly in terms of hardware. They are also much slower than addition and subtraction circuits, a consequence of their many logic levels. An *n*-bit combinational multiplier or divider is typically composed of *n* or more levels of add-subtract logic, making multiplication and division at least *n* times slower than addition or subtraction. The number of gates in the multiply-divide logic is also greater by a factor of about *n*. Hence except when *n* is very small, complete ALUs are usually constructed from low-cost sequential circuits where add and subtract each take one clock cycle, while multiplication and division are multicycle operations.

Sequential ALU Basic Design

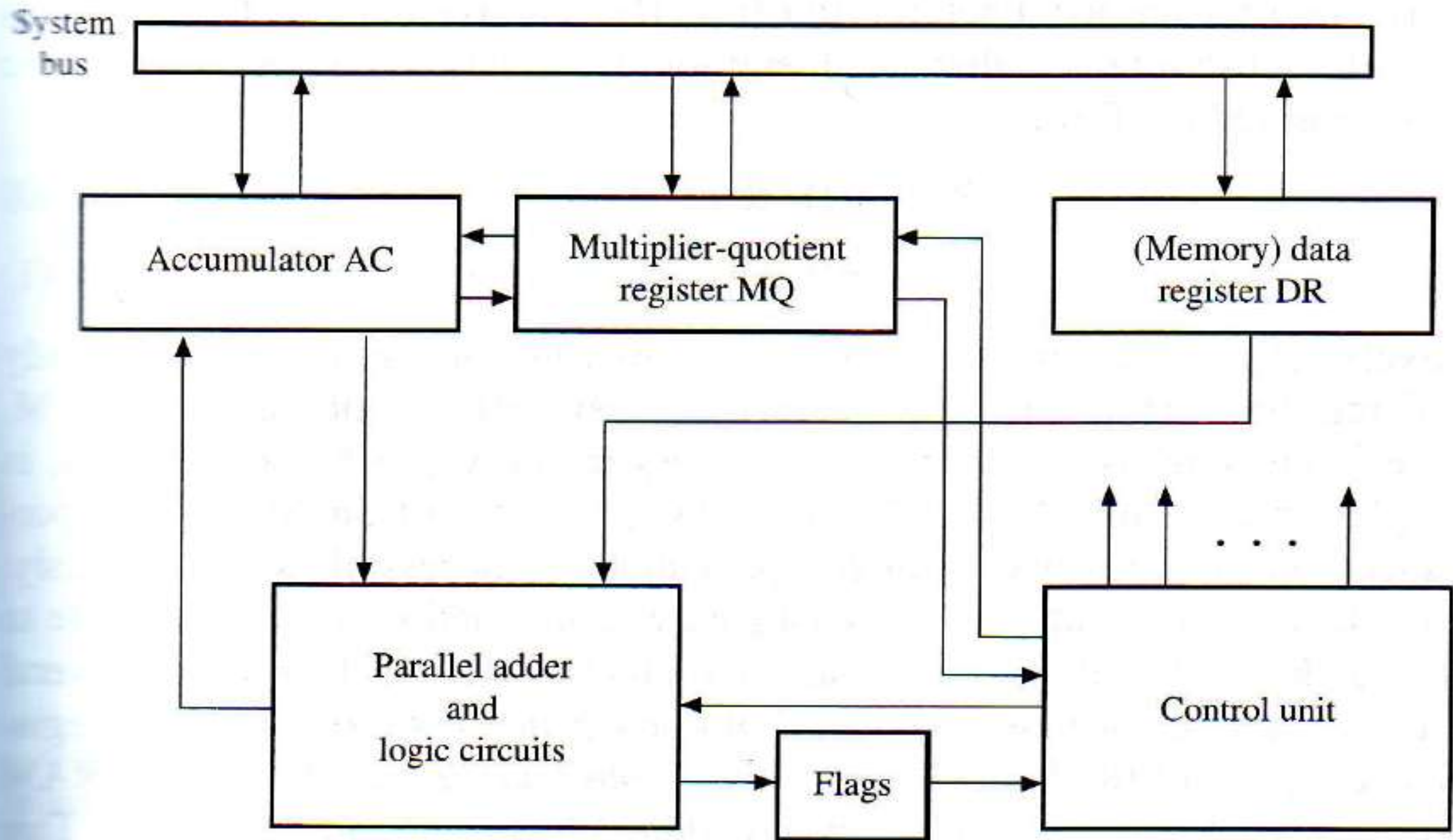


Figure 4.32
Structure of a basic sequential ALU.

Sequential ALU

Basic design. Figure 4.32 shows a widely used sequential ALU design that aims at minimizing hardware costs. This ALU organization is found in the IAS computer (Figure 1.11) and in many computers built after IAS. It is intended to implement multiplication and division using one of the sequential digit-by-digit shift-and-add/subtract algorithms discussed earlier. Three one-word registers are used for operand storage: the accumulator AC, the multiplier-quotient register MQ, and the data register DR. AC and MQ are organized as a single register AC.MQ capable of left- and right-shifting. Additional data processing is provided by a combinational ALU capable of addition, subtraction, and logical operations; we will refer to this unit as the add-subtract unit. This unit derives its inputs from AC and DR and places its results in AC. The MQ register is so-called because it stores the multiplier during multiplication and the quotient during division. DR stores the multiplicand or divisor, while the result (product or quotient and remainder) is stored in the register-pair AC.MQ. The role of these registers is defined concisely as follows:

Sequential ALU

Addition	AC := AC + DR
Subtraction	AC := AC – DR
Multiplication	AC.MQ := DR × MQ
Division	AC.MQ := MQ/DR
AND	AC := AC ***and*** DR
OR	AC := AC ***or*** DR
EXCLUSIVE-OR	AC := AC ***xor*** DR
NOT	AC := ***not***(AC)

DR can serve as a memory data register to store data addressed by an instruction address field ADR. Then DR can be replaced by M(ADR) in the above list of ALU operations, resulting in a one-address memory-referencing format.

Register File

Register files. Modern CPUs retain special registers like the multiplier-quotient register MQ for multiplication and division, but the accumulator AC and the data register DR are usually replaced by a set of general-purpose registers $R_0{:}R_{m-1}$ known as a register file RF. Each register R_i in RF is individually addressable—its address is the subscript i—so that arithmetic-logic instructions can take the generic two- and three-address forms

$$R_2 := f(R_1,R_2) \tag{4.40}$$

$$R_3 := f(R_1,R_2) \tag{4.41}$$

respectively. Hence the processor can retain intermediate results in fast, easily accessed registers, rather than having to pack them off to external memory M. Clearly RF functions as a small random-access memory (RAM) and, in fact, is often implemented using a fast RAM technology. RF differs from M in one important respect: RF requires two or three operands to be accessible simultaneously. For example, to implement (4.40) as a single-cycle instruction, we must be able to read R_1 and R_2, and write to R_2 in the same clock cycle. RF then needs several access ports for simultaneously reading from or writing to several different registers. Hence a register file is often realized as a *multiport RAM*. A standard RAM has just one access port with an associated address bus *ADR* and data bus *D*. This port can be used to read or write the data word in the single word location we denote by M(*ADR*).

Register File

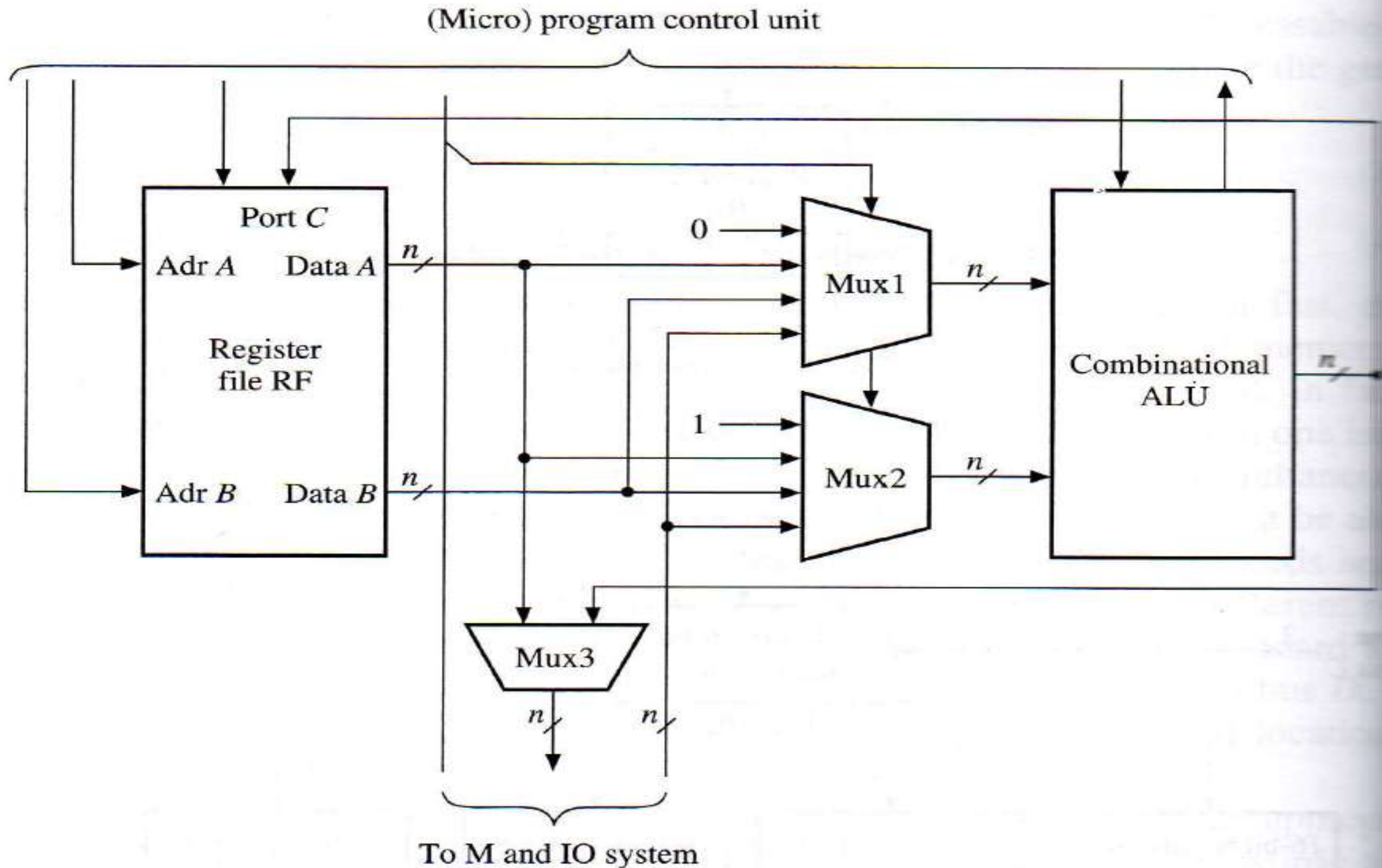


Figure 4.34
A generic datapath unit with an ALU and a register file.

Floating Point Arithmatic

Let (X_M, X_E) be the floating-point representation of a number X, which therefore has the numerical value $X_M \times B^{X_E}$. Recall from section 3.2.3 that the mantissa (significand) X_M and the exponent X_E are fixed-point numbers and that the base B is the same as the base (radix) of X_M. To simplify the discussion, we make the following realistic assumptions:

1. X_M is an n_M-bit binary (twos-complement or sign-magnitude) fraction.
2. X_E is an n_E-bit integer in excess-$2^{n_E - 1}$ code, implying an exponent bias of $2^{n_E - 1}$.
3. $B = 2$.

We also assume that the floating-point numbers are stored in normal form only; hence the final result of each floating-point arithmetic operation should be normalized.

Basic Operation

Basic operations. General formulas for floating-point addition, subtraction, multiplication, and division are given in Figure 4.40. Multiplication and division are relatively simple because the mantissas and exponents can be processed independently. Floating-point multiplication requires a fixed-point multiplication of the mantissas and a fixed-point addition of the exponents. For example, if $X = 1.32400111 \times 10^{17}$ and $Y = 1.04799245 \times 10^{21}$, the product $X \times Y$ is given by $(1.32400111 \times 1.04799245) \times 10^{(17+21)} = 1.38758607 \times 10^{38}$. Floating-point division requires a fixed-point division involving the mantissas and a fixed-point subtraction involving the exponents. Thus multiplication and division are not much harder to implement than the corresponding fixed-point operations.

Basic Operation

Floating-point addition and subtraction are complicated by the fact that the exponents of the two input operands must be made equal before the corresponding mantissas can be added or subtracted. As suggested by Figure 4.40, this exponent equalization can be done by right-shifting the mantissa X_M associated with the smaller exponent X_E a total of $Y_E - X_E$ digit positions to form a new mantissa

Addition	$X + Y = (X_M 2^{X_E - Y_E} + Y_M) \times 2^{Y_E}$	where $X_E \leq Y_E$
Subtraction	$X - Y = (X_M 2^{X_E - Y_E} - Y_M) \times 2^{Y_E}$	
Multiplication	$X \times Y = (X_M \times Y_M) \times 2^{X_E + Y_E}$	
Division	$X/Y = (X_M / Y_M) \times 2^{X_E - Y_E}$	

Figure 4.40
The four basic arithmetic operations for floating-point numbers.

Basic Operation

$X_M 2^{X_E - Y_E}$, which can then be combined directly with Y_M. Thus floating-point addition and subtraction have three main steps:

1. Compute $Y_E - X_E$, a fixed-point subtraction.
2. Shift X_M by $Y_E - X_E$ places to the right to form $X_M 2^{X_E - Y_E}$.
3. Compute $X_M 2^{X_E - Y_E} \pm Y_M$, a fixed-point addition or subtraction.

For example, to add the decimal floating-point numbers $X = 1.32400111 \times 10^{17}$ and $Y = 1.04799245 \times 10^{21}$, we first compute $Y_E - X_E = 21 - 17 = 4$, identifying X_E as the smaller exponent. We then right-shift X_M by four places to obtain $X_M 2^{-4} = 0.00013240$. Finally, we perform the mantissa addition $X_M 2^{-4} + Y_M = 0.00013240 + 1.04799245 = 1.04812485$, so the final result has mantissa 1.04812485 and exponent 21.

Basic Operation

Each floating-point arithmetic operation needs an extra step in order to normalize the result. A number $X = (X_M, X_E)$ is normalized by left-shifting (right-shifting) X_M and decrementing (incrementing) X_E by 1 to compensate for each one-digit shift of X_M. As noted earlier, a twos-complement fraction is normalized when the sign bit x_{n-1} differs from the bit x_{n-2} on its right, a fact used to terminate the normalization process. A sign-magnitude fraction is normalized by left-shifting the magnitude part until there are no leading 0s, that is, until $x_{n-2} = 1$. (The normalization rules are different if the base B is not two.) The left-most bit of the mantissa may be hidden, since normalization fixes its value; see the discussion of

Algorithm for floating point Operation

Pipelined floating point operation

I/O Interface

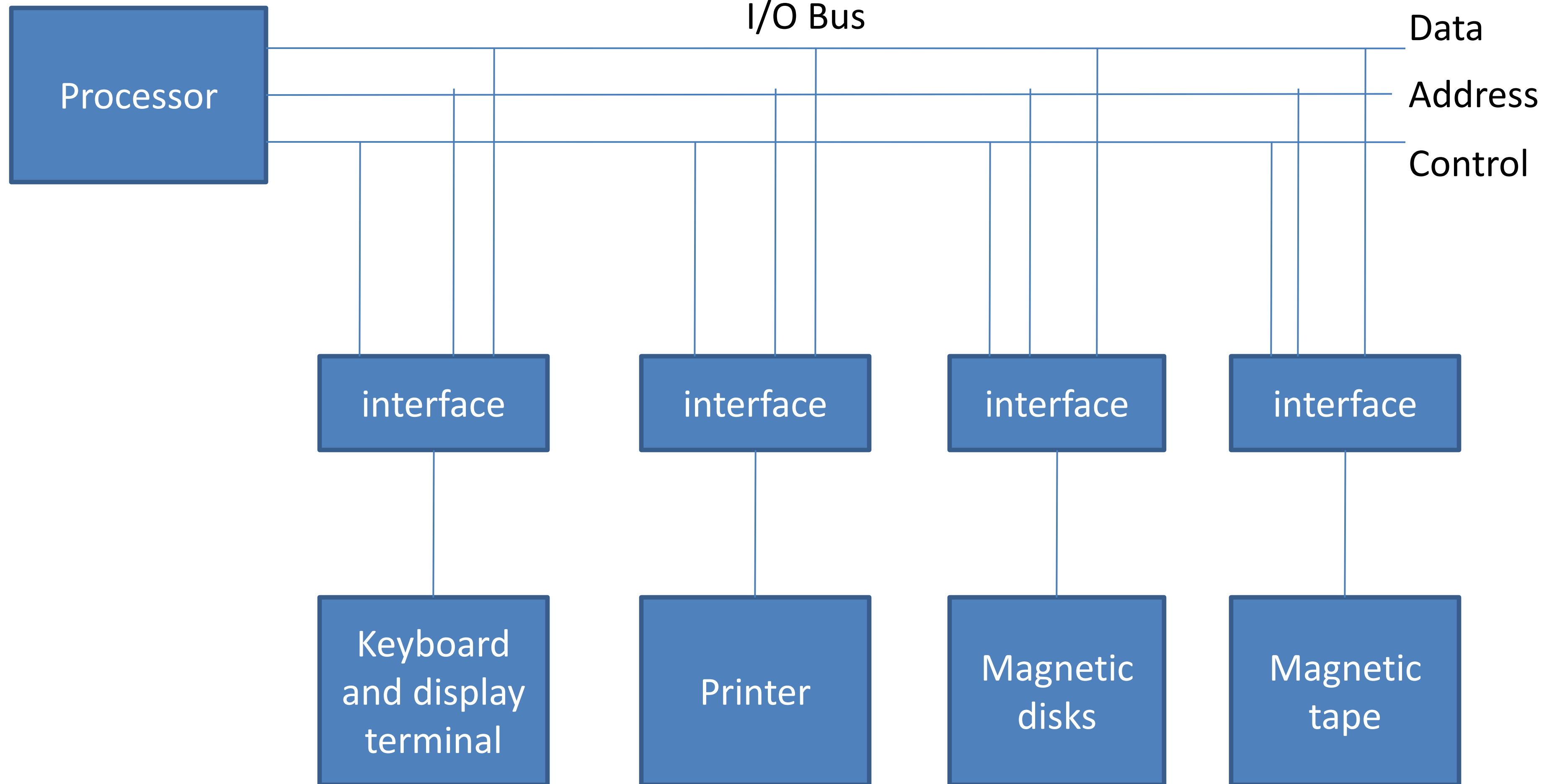
Input-Output interface provides a method for transferring information between internal storage and external I/O devices. The purpose of the communication link is to resolve the differences that exist between the central computer and each peripheral. The major differences are :

1. Peripherals are electromechanical and electromagnetic devices and their manner of operation of the CPU and memory, which are electronic devices. Therefore, a conversion of signal values may be required.

2. The data transfer rate of peripherals is usually slower than the transfer rate of the CPU, and consequently, a synchronization mechanism may be needed.
3. Data codes and formats in peripherals differ from the word format in the CPU and memory.
4. The operating modes of peripheral are different from each other and each must be controlled so as not to disturb the operation of other peripherals connected to the CPU.

To resolve these differences computer systems include special hardware components between the CPU and peripherals to supervise and synchronize all input and output transfers. These components are called “interface units” because they interface between the processor bus and peripheral device. In addition each device may have its own controller that supervises the operation of the particular mechanism in the peripherals.

I/O Bus and Interface Modules



Connection of I/O bus to input-output devices.

I/O Commands

There are four types of commands that an interface may receive. They are classified as control, status, data output, and data input.

1. Control Command:- A control command is issued to activate peripheral and inform it what to do.

   For example:- A magnetic tape unit may be instructed to backspace the tape by one record, to rewind the tape, or to start the tape moving in the forward direction.

2. Status Command:- A status command is used to test various status conditions in the interface and the peripheral.

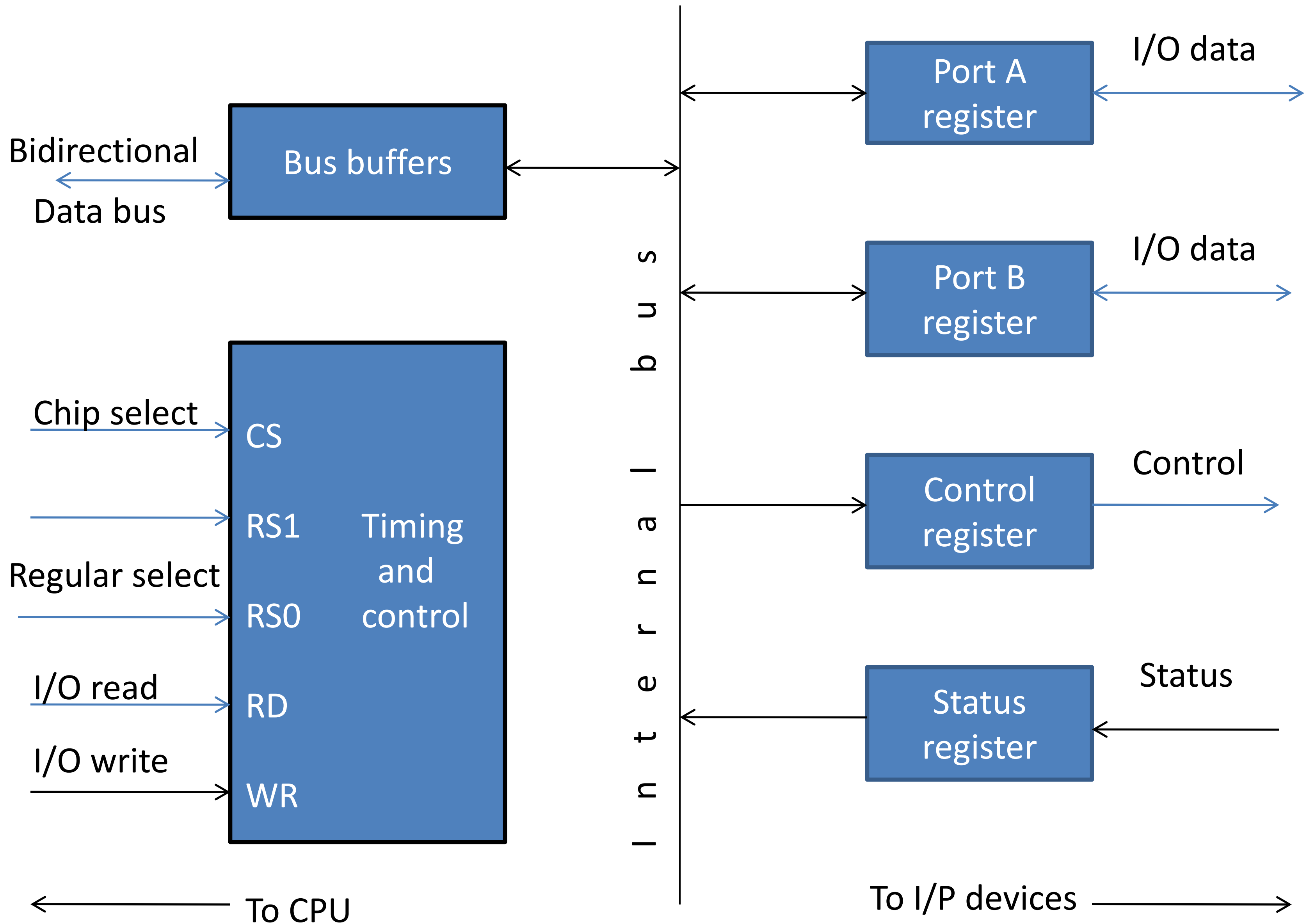
For example:- the computer may wish to check the status of the peripheral before a transfer is initiated. During the transfer, one or more errors may occur which are detected by the interface. These errors are designated by setting bits in a status register that the processor can read at certain intervals.

3. Output data:- It causes the interface to respond by transferring data from the bus into one of its register. Consider an example with a tape unit. The computer starts the tape moving by issuing a control command.

The processor then monitor the status of the tape by means of a status command. When the tape is in the correct position the processor issues a data output command. The interface responds to the address and command and transfers the information from the data lines in the bus to its buffer register. The interface then communicates with the tape controller and sends the data to be stored on tape.

4. Input data:- The data input command is the opposite of the data output. In this case the interface receive an item of data from the peripheral and places it in its buffer register. The processor checks if data are available by means of a status command and then issues a data input command. The interface places the data on the data lines, where they are accepted by the processor.

Example of I/O Interface



Example of I/O Interface unit

CS	RS1	RS0	Register selected
0	*	*	None: data bus in high impedance
1	0	0	Port A register
1	0	1	Port B register
1	1	0	Control register
1	1	1	Status register

Synchronization

- The process that communicate, do so through a synchronization mechanism. A process executes with unpredictable velocity and generates events and actions that must be recognized by other co-operating processes. The set of constraints on the ordering of these events constitutes the set of synchronization required for the operating processes. The synchronization technique is used to delay execution of a process in order to satisfy such constraints.

In a multiprocessor system, processes can execute concurrently until they need to interact. Planned and controlled interaction is known as process communication or process synchronization. Process communication must take place through shared or global variables. Co-operating process must communicate to synchronize or limit their concurrency.

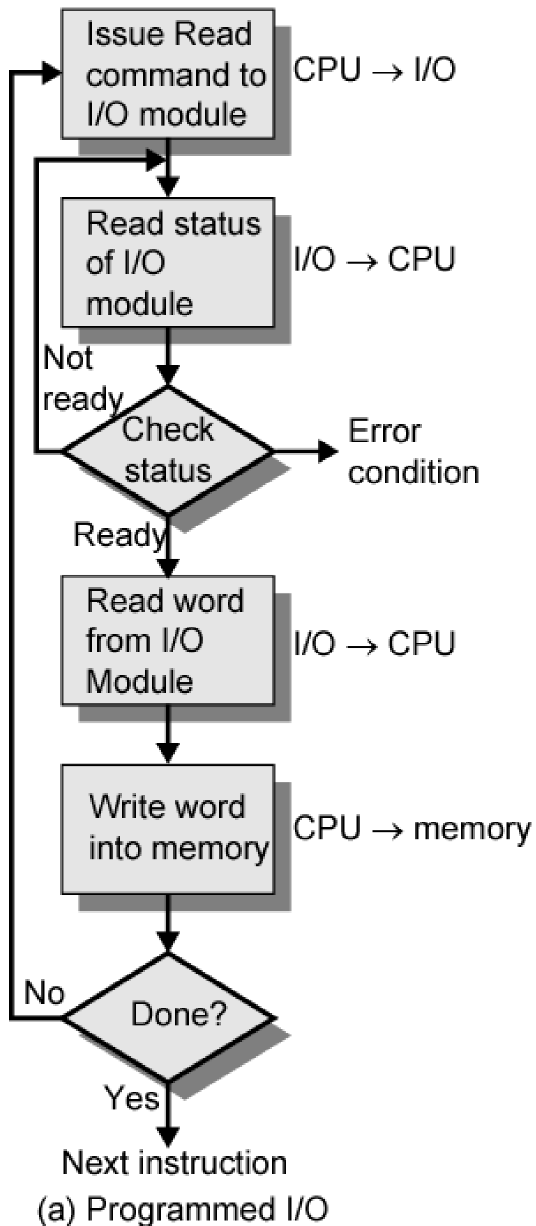
Two types of synchronization are generally needed while using shared variable.

1. Mutual Exclusion :- Mutual exclusion ensures that a physical or virtual resource is held indivisibly.
2. Condition Synchronization :- When a shared data object is in a state that is not appropriate for executing a given operation, any process which attempts such an operation must be delayed. Such operation must be delayed until the state of data objects to the desired value as a result of other process being executed. This type of synchronization is called " Condition synchronization".

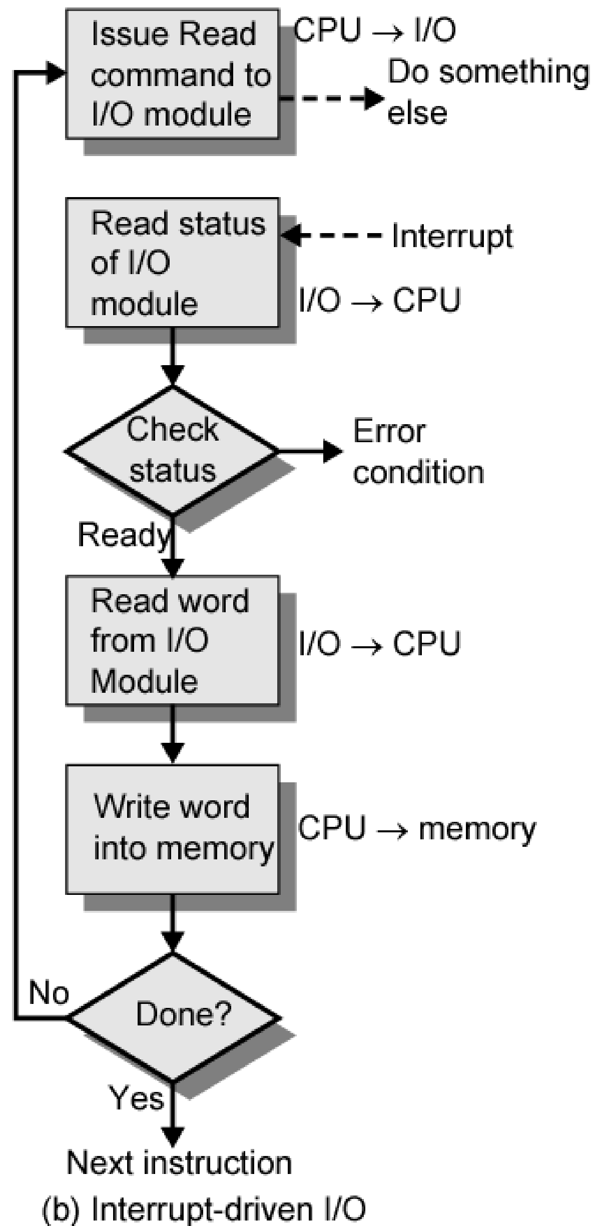
Input Output Techniques

- Programmed
- Interrupt driven
- Direct Memory Access (DMA)

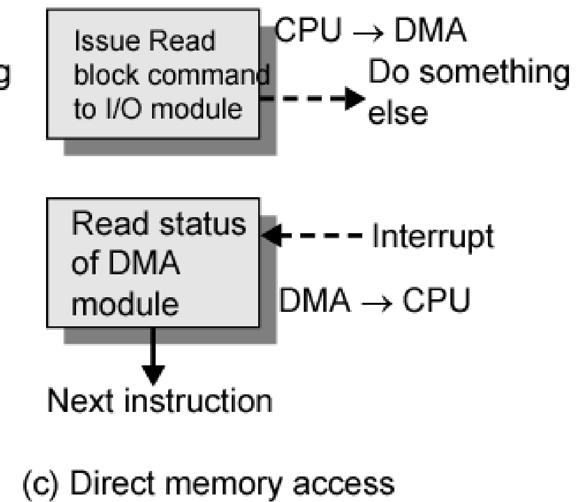
Three Techniques for Input of a Block of Data



(a) Programmed I/O



(b) Interrupt-driven I/O



(c) Direct memory access

Programmed I/O

- CPU has direct control over I/O
  - Sensing status
  - Read/write commands
  - Transferring data
- CPU waits for I/O module to complete operation
- Wastes CPU time

Programmed I/O - detail

- CPU requests I/O operation
- I/O module performs operation
- I/O module sets status bits
- CPU checks status bits periodically
- I/O module does not inform CPU directly
- I/O module does not interrupt CPU
- CPU may wait or come back later

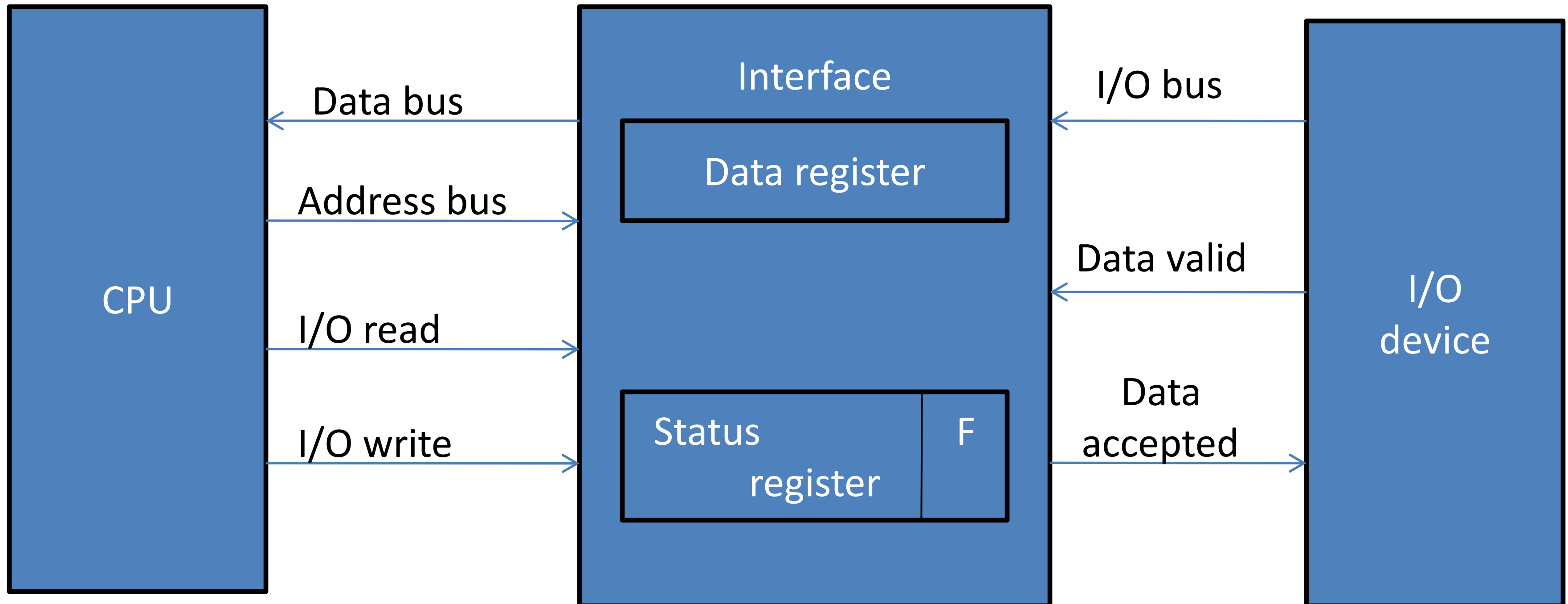
Programmed I/O

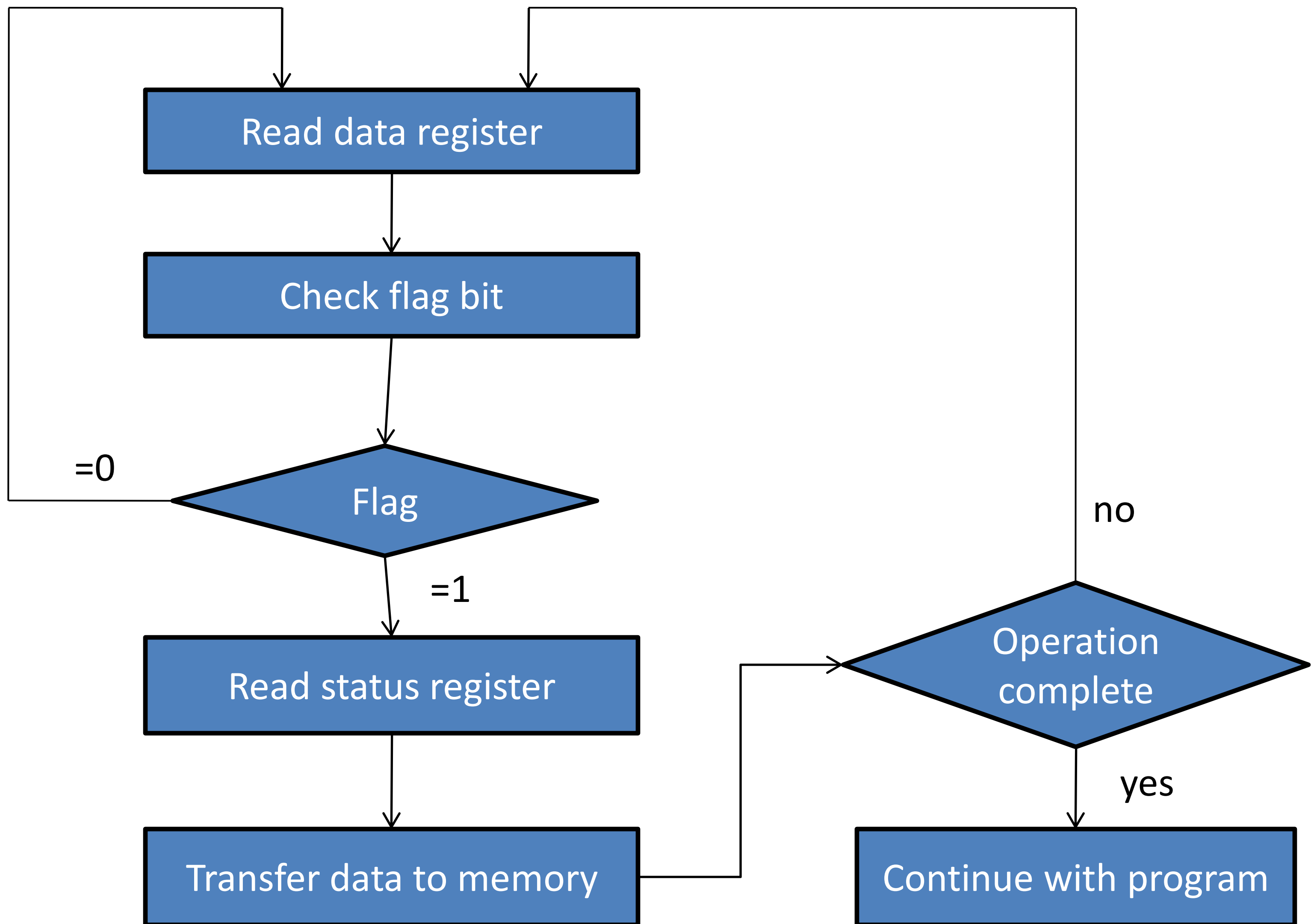
Programmed I/O operations are the result of I/O instructions written in the computer program. Each data item transfer is initiated by an instruction in the program. Usually the transfer is to read and from a CPU register and peripheral. Other instructions are needed to transfer the data to and from CPU and memory. Transferring data under program control requires constant monitoring of the peripheral by the CPU. Once a data transfer is initiated, the CPU is required to monitor the interface to see when a transfer can again be made.

Example of Programmed I/O

In the programmed I/O method, the I/O devices does not have direct access to memory. A transfer from an I/O device to memory requires the execution of several instruction by the CPU, including an input instruction to transfer the data from the device to the CPU and a store instruction to transfer the data from the CPU to memory. Other instruction may be needed to verify that the data are available from the device and to count the numbers of words transferred.

An example of data transfer from an I/O device through an interface into the CPU is shown in figure:-





Flowchart for CPU program to input data

I/O Commands

- CPU issues address
  - Identifies module (& device if >1 per module)
- CPU issues command
  - Control - telling module what to do
    - e.g. spin up disk
  - Test - check status
    - e.g. power? Error?
  - Read/Write
    - Module transfers data via buffer from/to device

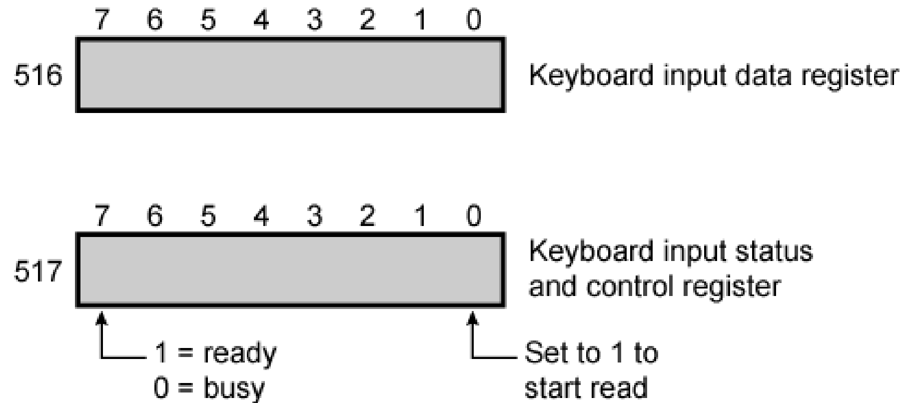
Addressing I/O Devices

- Under programmed I/O data transfer is very like memory access (CPU viewpoint)
- Each device given unique identifier
- CPU commands contain identifier (address)

I/O Mapping

- Memory mapped I/O
  - Devices and memory share an address space
  - I/O looks just like memory read/write
  - No special commands for I/O
    - Large selection of memory access commands available
- Isolated I/O
  - Separate address spaces
  - Need I/O or memory select lines
  - Special commands for I/O
    - Limited set

Memory Mapped and Isolated I/O



ADDRESS	INSTRUCTION	OPERAND	COMMENT
200	Load AC	"1"	Load accumulator
	Store AC	517	Initiate keyboard read
202	Load AC	517	Get status byte
	Branch if Sign = 0	202	Loop until ready
	Load AC	516	Load data byte

(a) Memory-mapped I/O

ADDRESS	INSTRUCTION	OPERAND	COMMENT
200	Load I/O	5	Initiate keyboard read
201	Test I/O	5	Check for completion
	Branch Not Ready	201	Loop until complete
	In	5	Load data byte

(b) Isolated I/O

Interrupt Mechanism

Data transfer between the CPU and I/O device is initiated by the CPU. However, the CPU can not start the transfer unless the device is ready to communicate with the CPU. The readiness of the device can be determined from an interrupt signal. The CPU respond to the Interrupt request by storing the return address from PC into a memory stack and then the program branches to service routine that processes the required transfer.

Priority Interrupt

A priority interrupt is a system that establishes a priority over the various sources to determine which condition is to be services first when two or more request arrive simultaneously. The system may also determine which condition are permitted to interrupt the computer while another interrupt is being serviced. Higher – priority interrupt levels are assigned to request which, if delayed or interrupted, could have serious consequences. Device with high speed transfer such as magnetic disk are given high priority. When two devices interrupt the computer at the same time, the computer services the device, with the higher priority first.

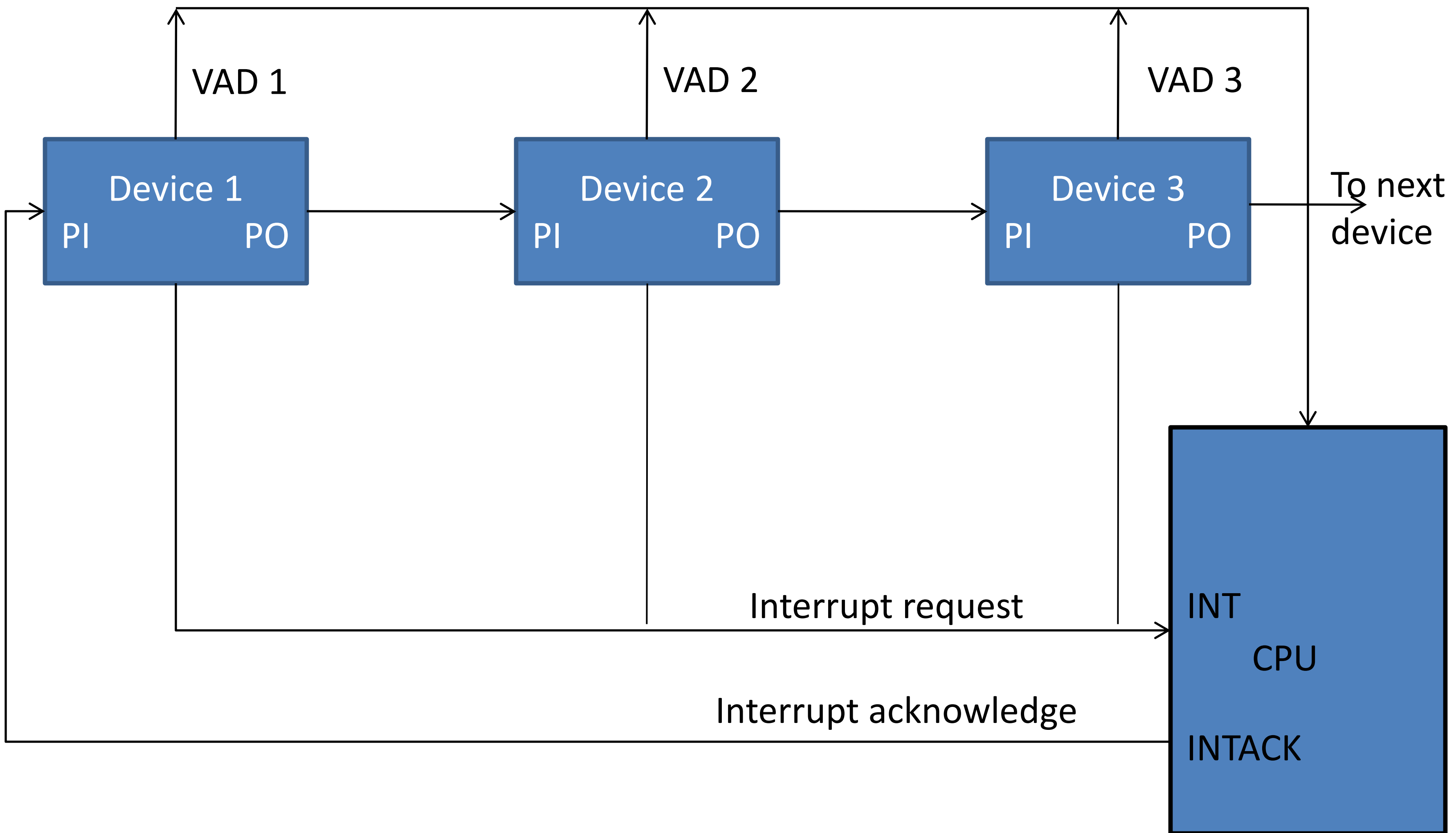
Polling

Establishing the priority of simultaneous interrupts can be done by software or hardware. A polling procedure is used to identify the highest priority source by software means. In this method there is a one common branch address for all interrupts. The program that take care of interrupt begin at the branch address and polls the interrupt source in sequence. The order in which they are tested determines the priority of each interrupt. The highest priority interrupt is tested first, and if its interrupt signal is on, control branches to service routine for this source. Otherwise the next lower priority source is tested, and so on.

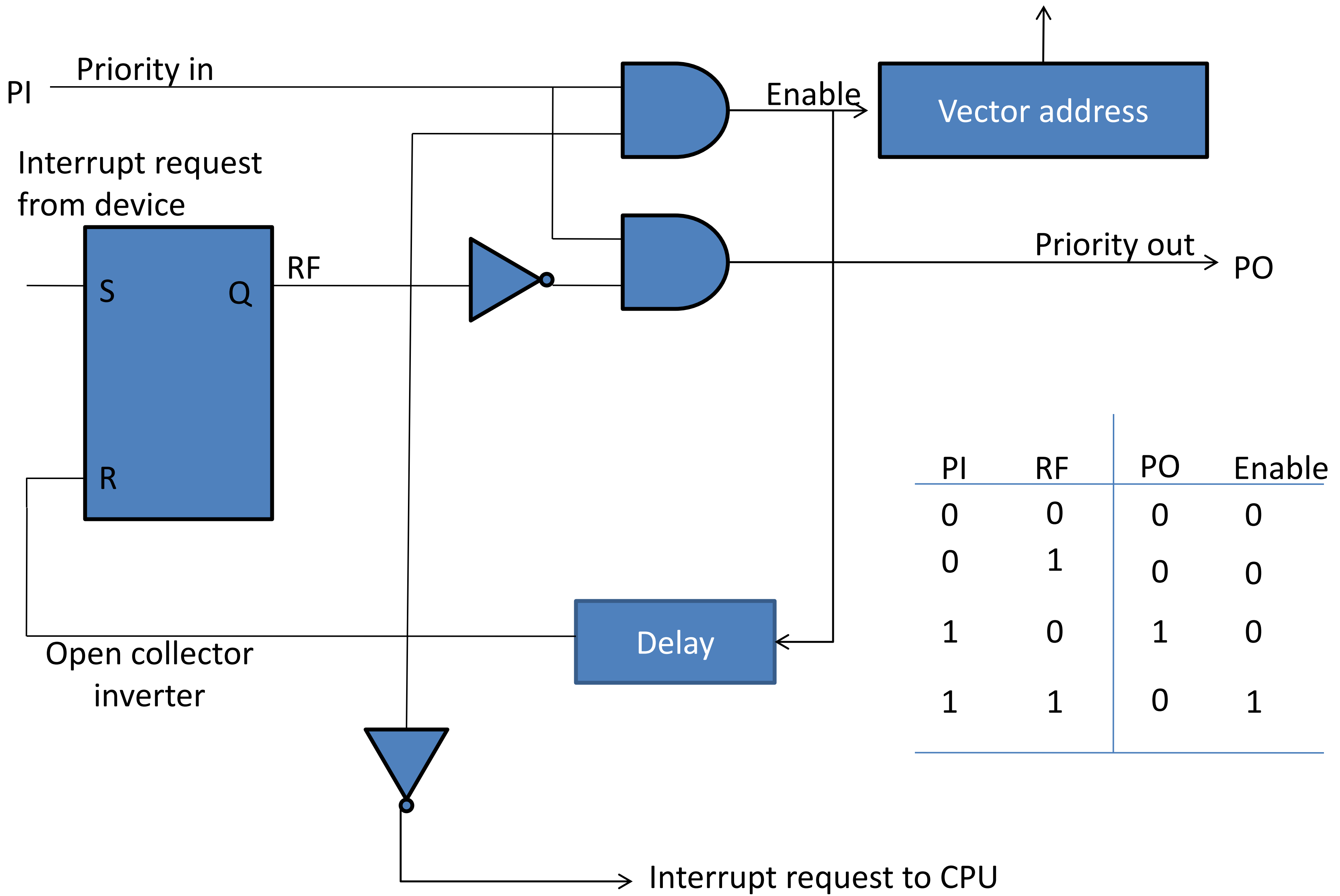
Daisy-Chaining Priority

The hardware priority interrupt can be established by either a serial or parallel connection of interrupt lines. The serial connection is also known as the daisy chaining method.

The daisy chaining method of establishing priority consist of a serial connection of all devices that request an interrupt. The device with the highest priority is places in the first position followed by lower priority devices up to the device with the lowest priority, which is places last in the chain. The method of connection between three devices and the CPU is shown in figure:-

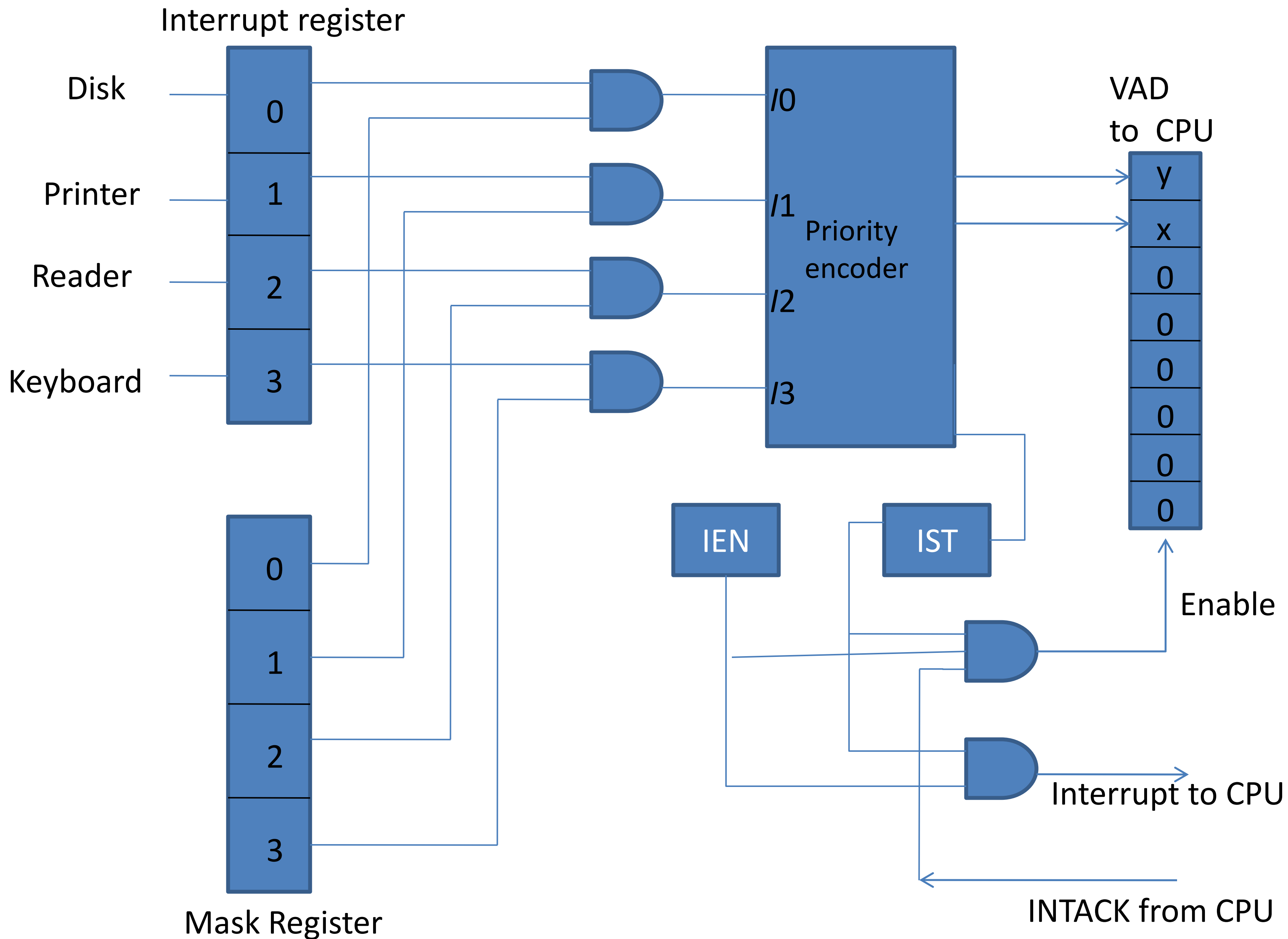


Daisy-Chaining Priority Interrupt



PI	RF	PO	Enable
0	0	0	0
0	1	0	0
1	0	1	0
1	1	0	1

One stage of the daisy chaining priority arrangement



Priority Interrupt Hardware

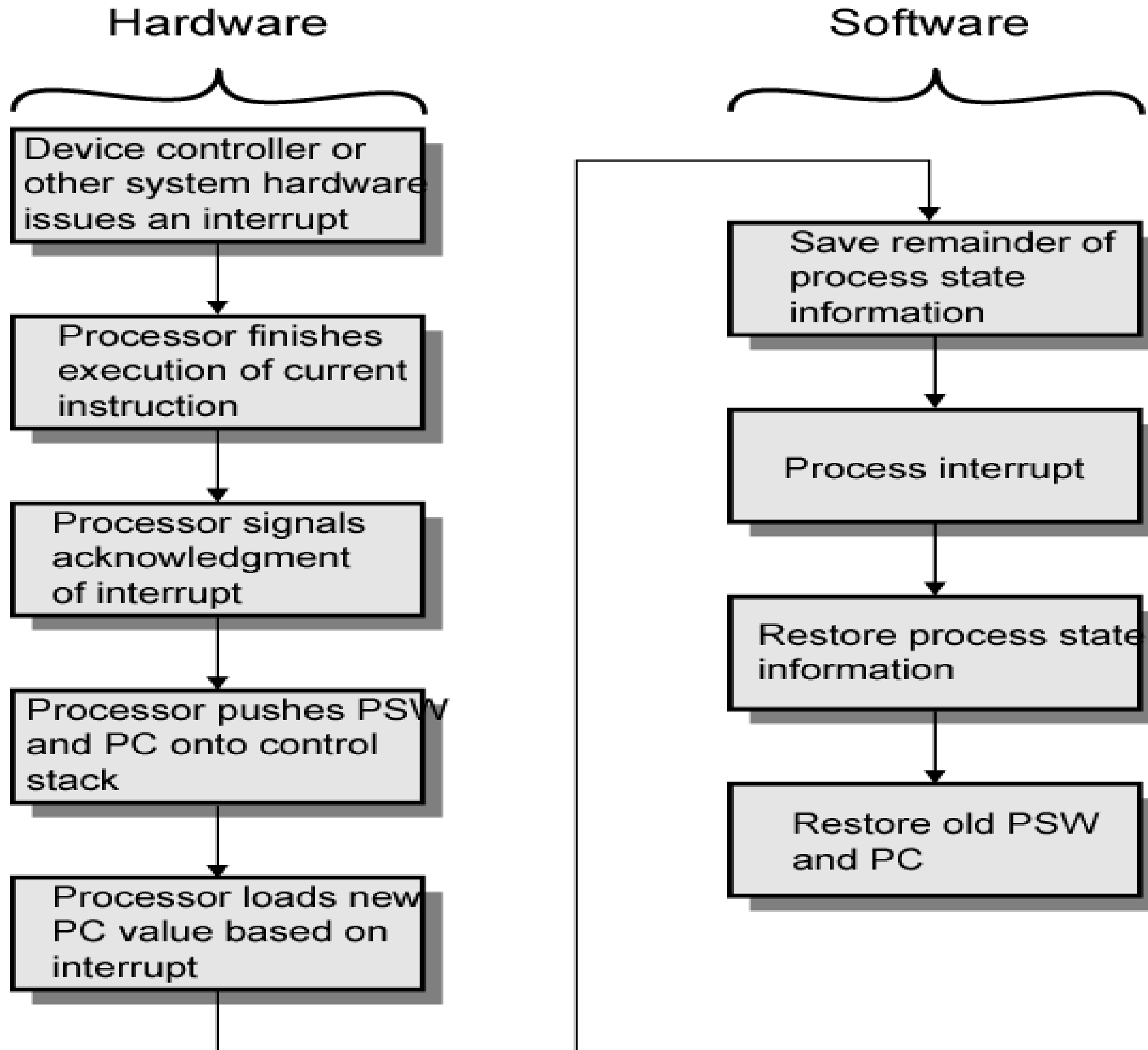
Interrupt Driven I/O

- Overcomes CPU waiting
- No repeated CPU checking of device
- I/O module interrupts when ready

Interrupt Driven I/O
Basic Operation

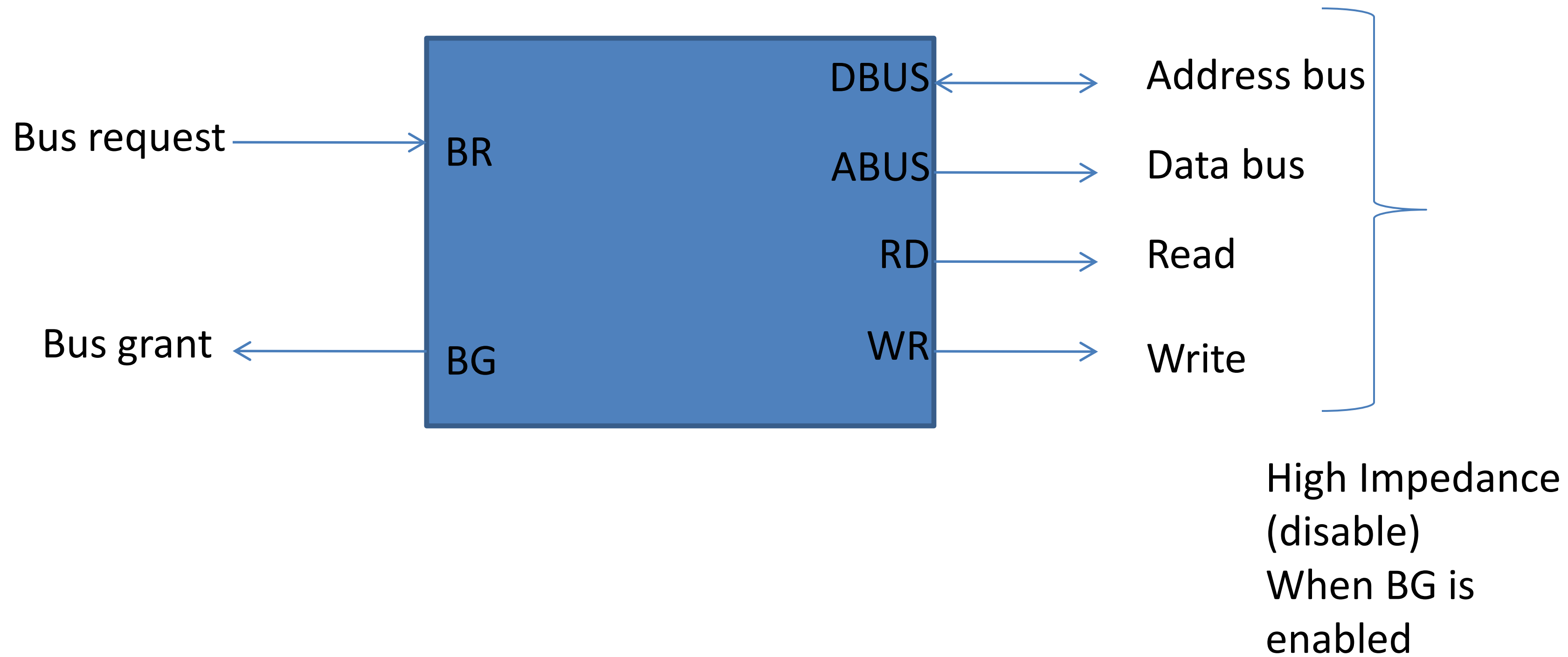
- CPU issues read command
- I/O module gets data from peripheral whilst CPU does other work
- I/O module interrupts CPU
- CPU requests data
- I/O module transfers data

Simple Interrupt Processing

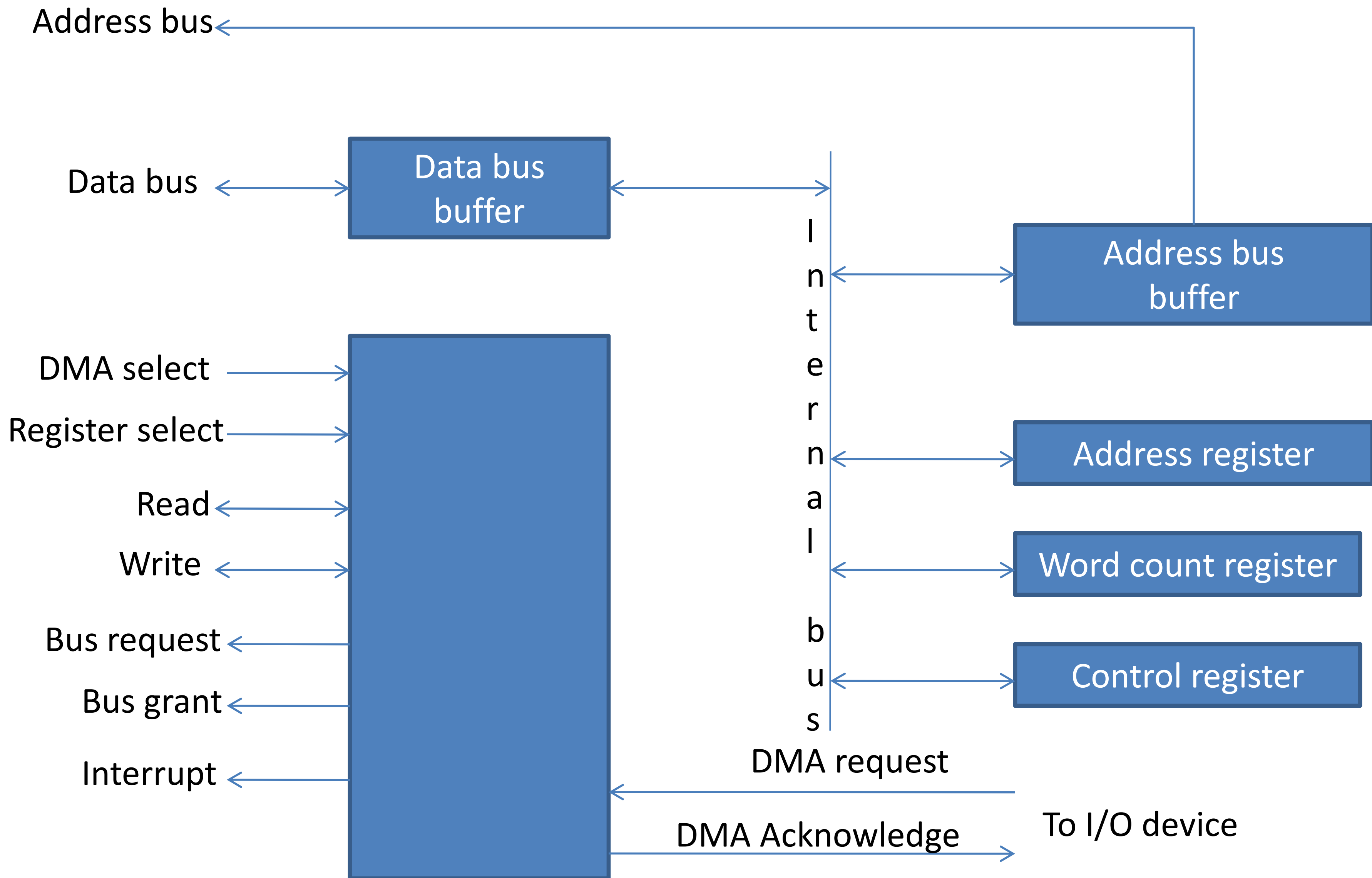


Direct Memory Access (DMA)

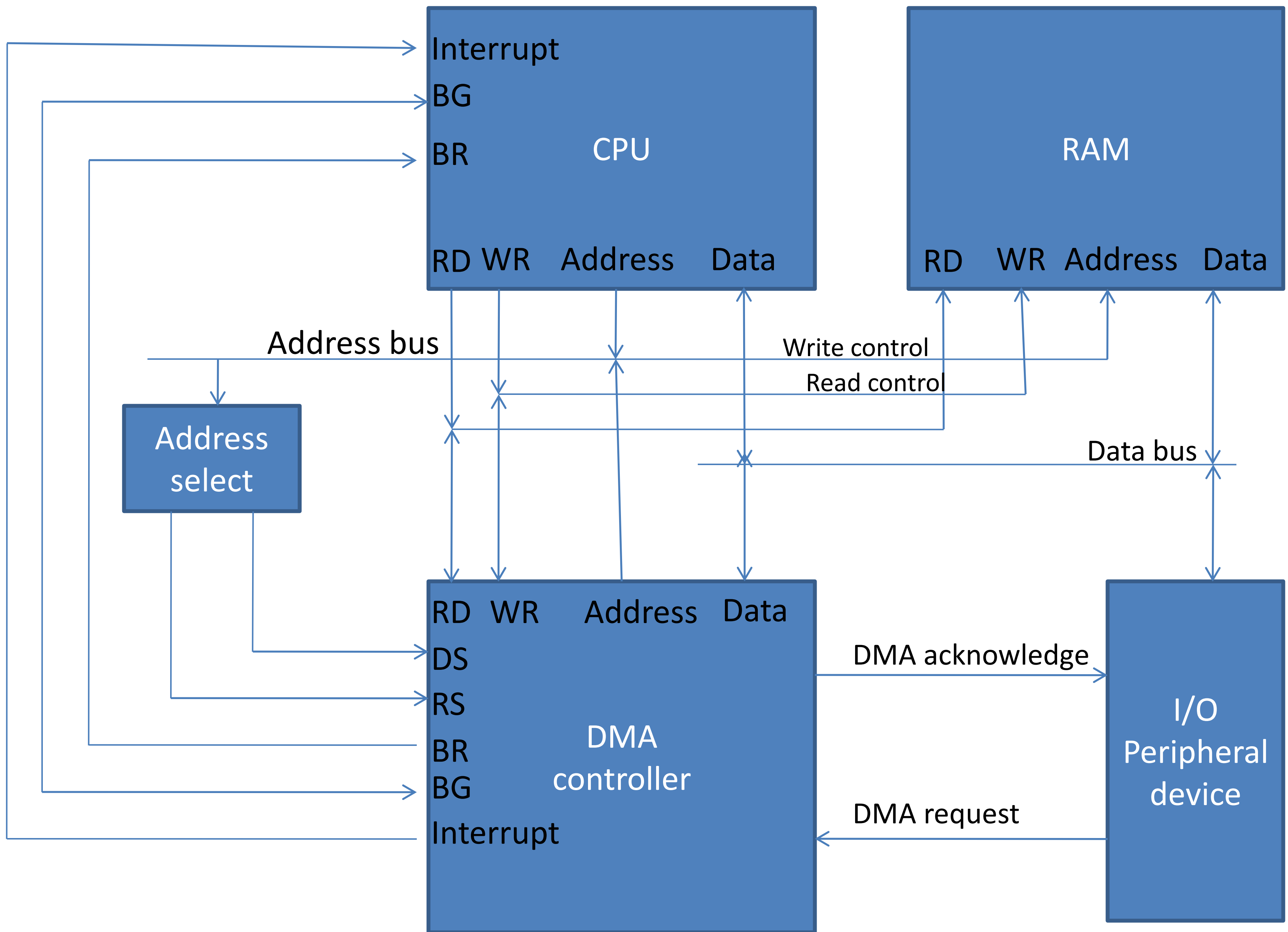
The transfer of data between a fast storage device such as magnetic disk and memory is often limited by the speed of the CPU. Removing the CPU from the path and letting the peripheral device manage the memory buses directly would improve the speed of transfer. This transfer technique is called direct memory access (DMA). During DMA transfer the CPU is idle and has no control of the memory buses. A DMA controller takes over the buses to manage the transfer directly between the I/O devices and memory.



CPU bus signals for DMA transfer



Block diagram of DMA controller



DMA transfer in a computer system

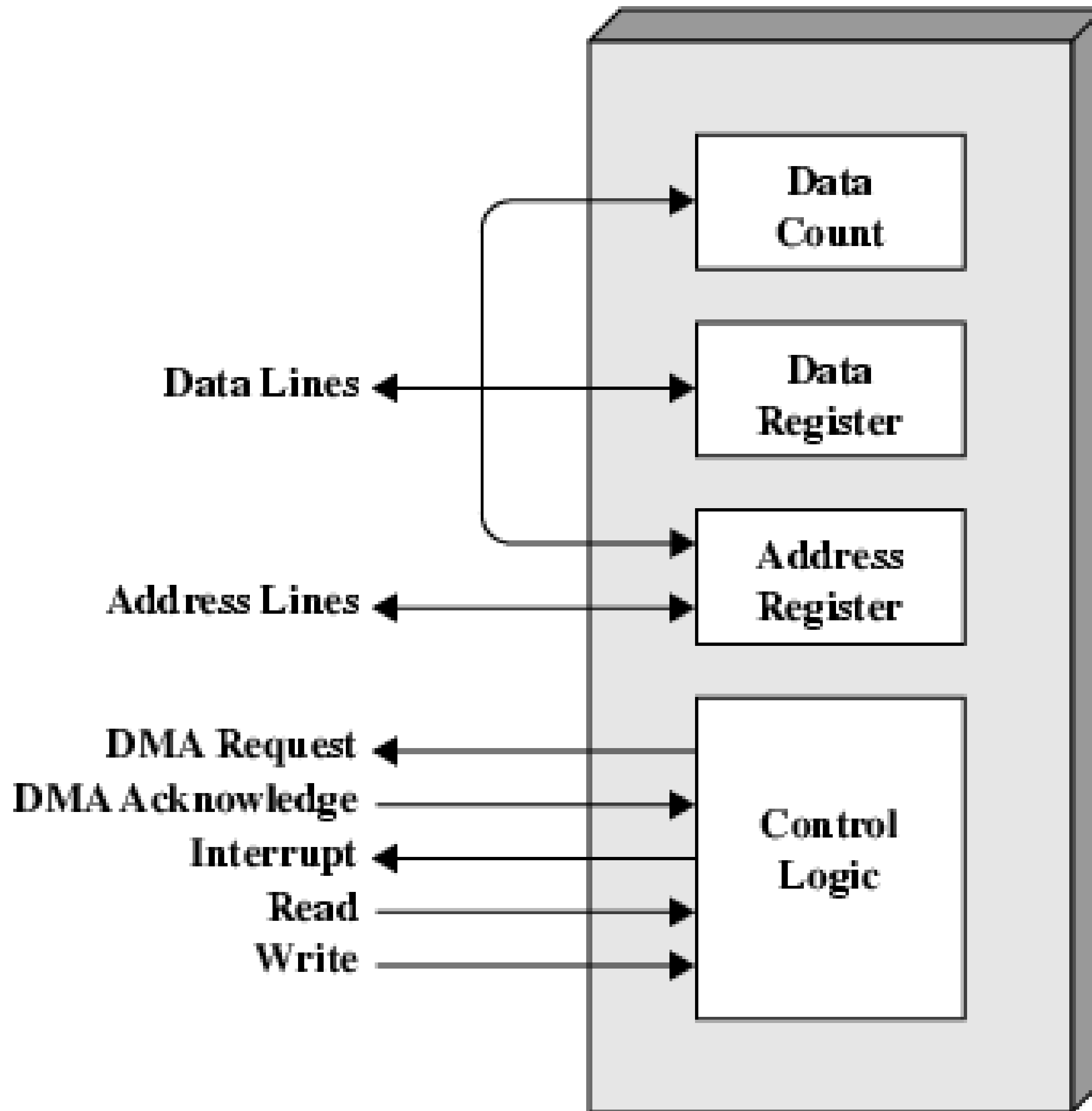
Direct Memory Access

- Interrupt driven and programmed I/O require active CPU intervention
  - Transfer rate is limited
  - CPU is tied up
- DMA is the answer

DMA Function

- Additional Module (hardware) on bus
- DMA controller takes over from CPU for I/O

Typical DMA Module Diagram



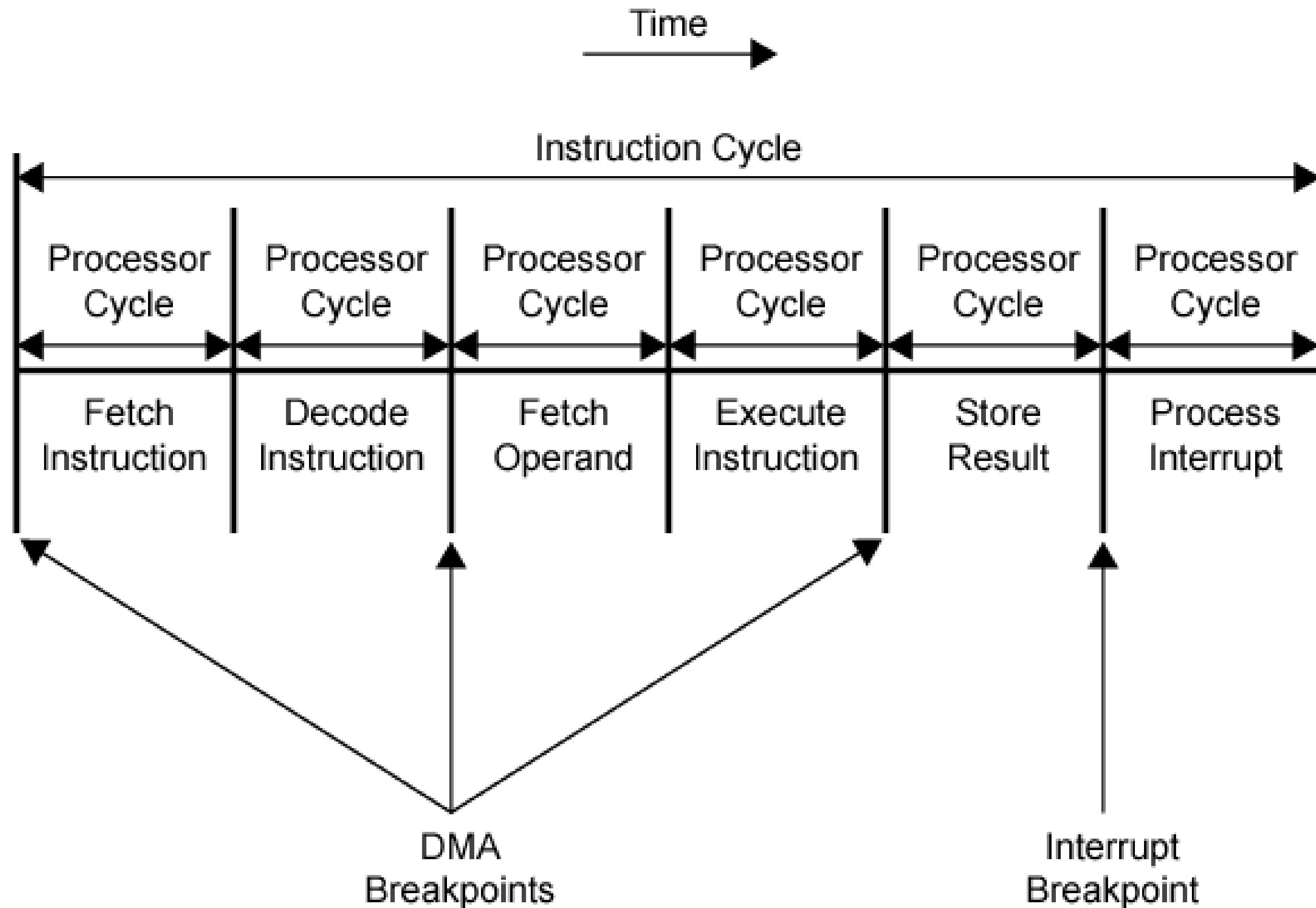
DMA Operation

- CPU tells DMA controller:-
  - Read/Write
  - Device address
  - Starting address of memory block for data
  - Amount of data to be transferred
- CPU carries on with other work
- DMA controller deals with transfer
- DMA controller sends interrupt when finished

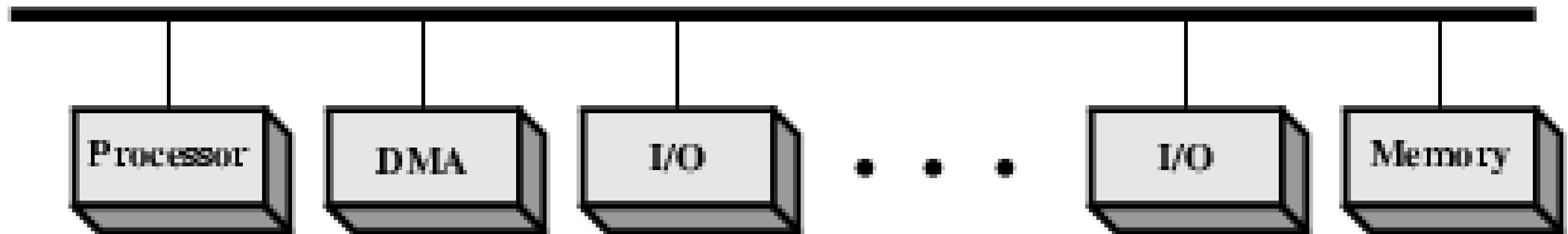
DMA Transfer
Cycle Stealing

- DMA controller takes over bus for a cycle
- Transfer of one word of data
- Not an interrupt
  - CPU does not switch context
- CPU suspended just before it accesses bus
  - i.e. before an operand or data fetch or a data write
- Slows down CPU but not as much as CPU doing transfer

DMA and Interrupt Breakpoints During an Instruction Cycle

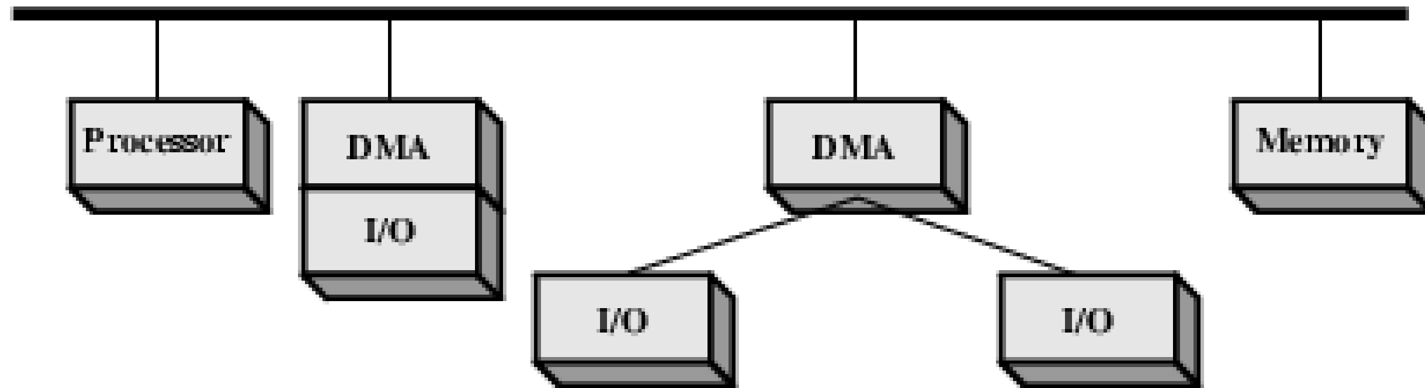


DMA Configurations (1)



- Single Bus, Detached DMA controller
- Each transfer uses bus twice
  - I/O to DMA then DMA to memory
- CPU is suspended twice

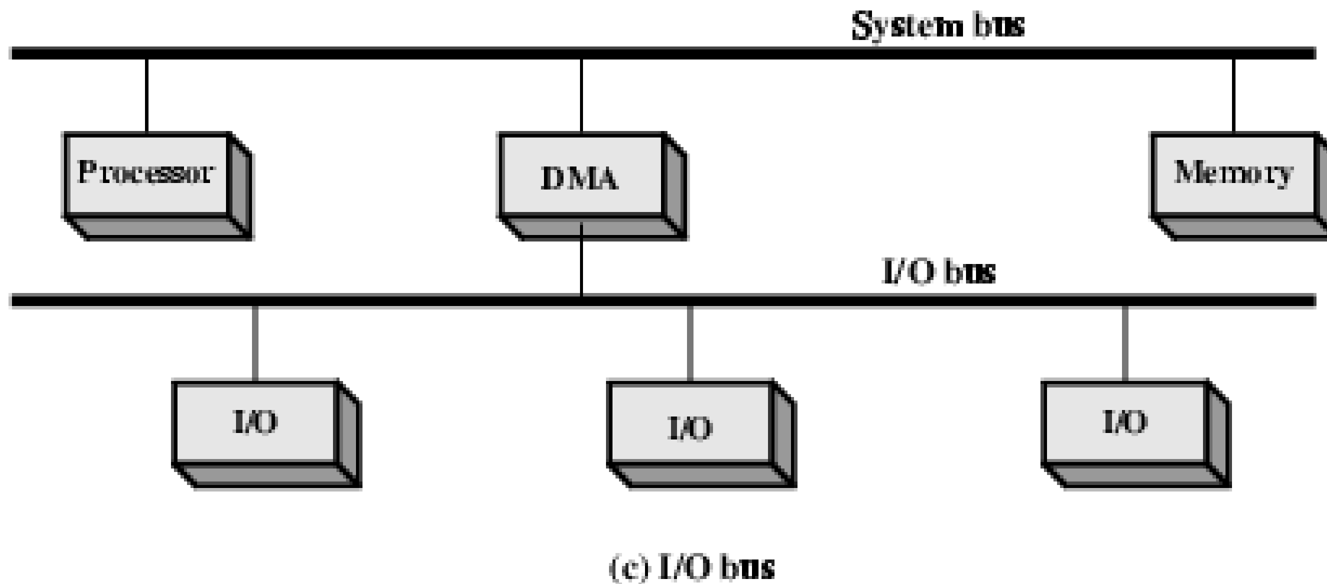
DMA Configurations (2)



(b) Single-bus, Integrated DMA-I/O

- Single Bus, Integrated DMA controller
- Controller may support >1 device
- Each transfer uses bus once
  - DMA to memory
- CPU is suspended once

DMA Configurations (3)

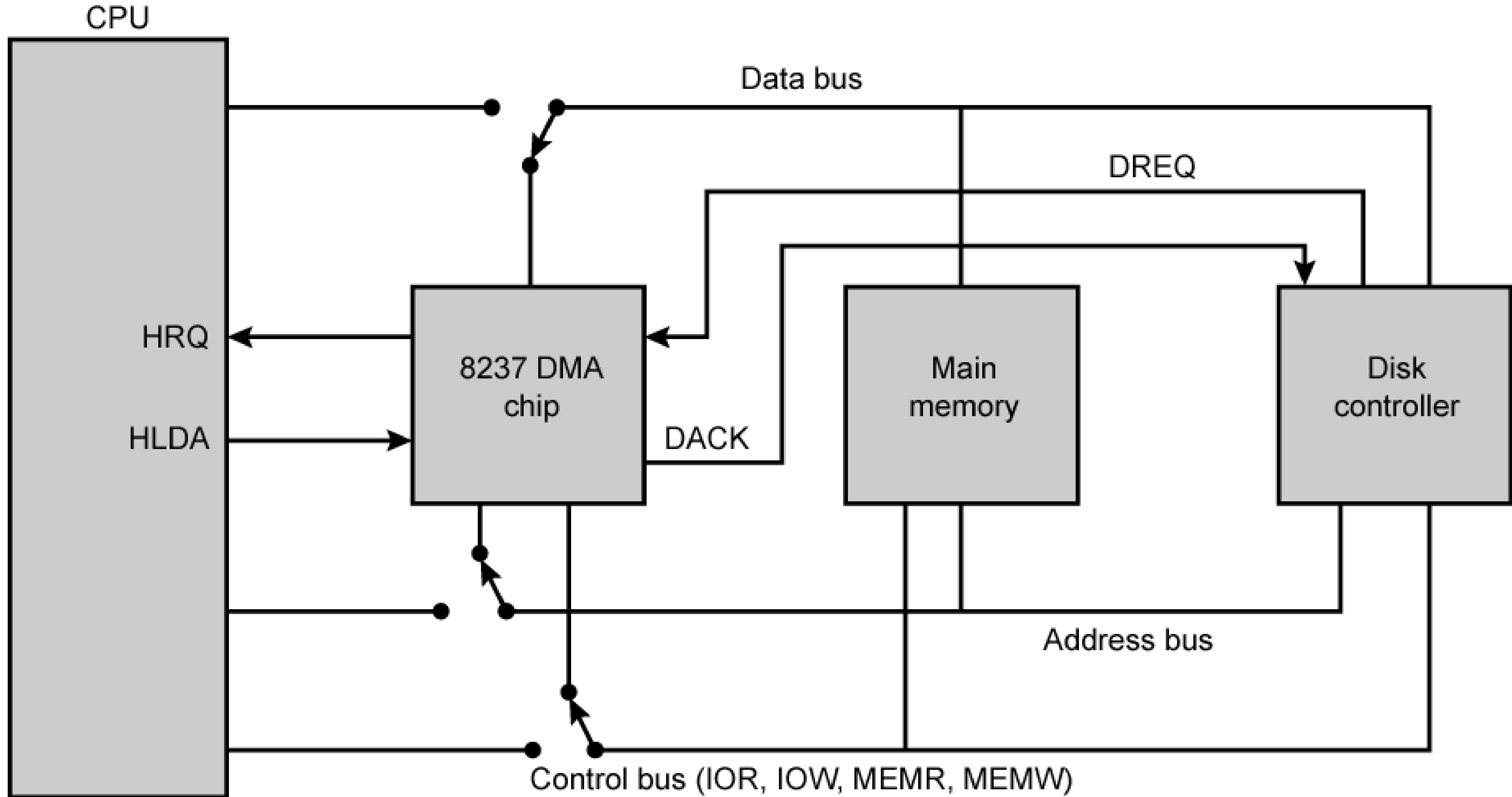


- Separate I/O Bus
- Bus supports all DMA enabled devices
- Each transfer uses bus once
  - DMA to memory
- CPU is suspended once

Intel 8237A DMA Controller

- Interfaces to 80x86 family and DRAM
- When DMA module needs buses it sends HOLD signal to processor
- CPU responds HLDA (hold acknowledge)
  - DMA module can use buses
- E.g. transfer data from memory to disk
  1. Device requests service of DMA by pulling DREQ (DMA request) high
  2. DMA puts high on HRQ (hold request),
  3. CPU finishes present bus cycle (not necessarily present instruction) and puts high on HDLA (hold acknowledge). HOLD remains active for duration of DMA
  4. DMA activates DACK (DMA acknowledge), telling device to start transfer
  5. DMA starts transfer by putting address of first byte on address bus and activating MEMR; it then activates IOW to write to peripheral. DMA decrements counter and increments address pointer. Repeat until count reaches zero
  6. DMA deactivates HRQ, giving bus back to CPU

8237 DMA Usage of Systems Bus



DACK = DMA acknowledge
DREQ = DMA request
HLDA = HOLD acknowledge
HRQ = HOLD request

Instruction pipeline

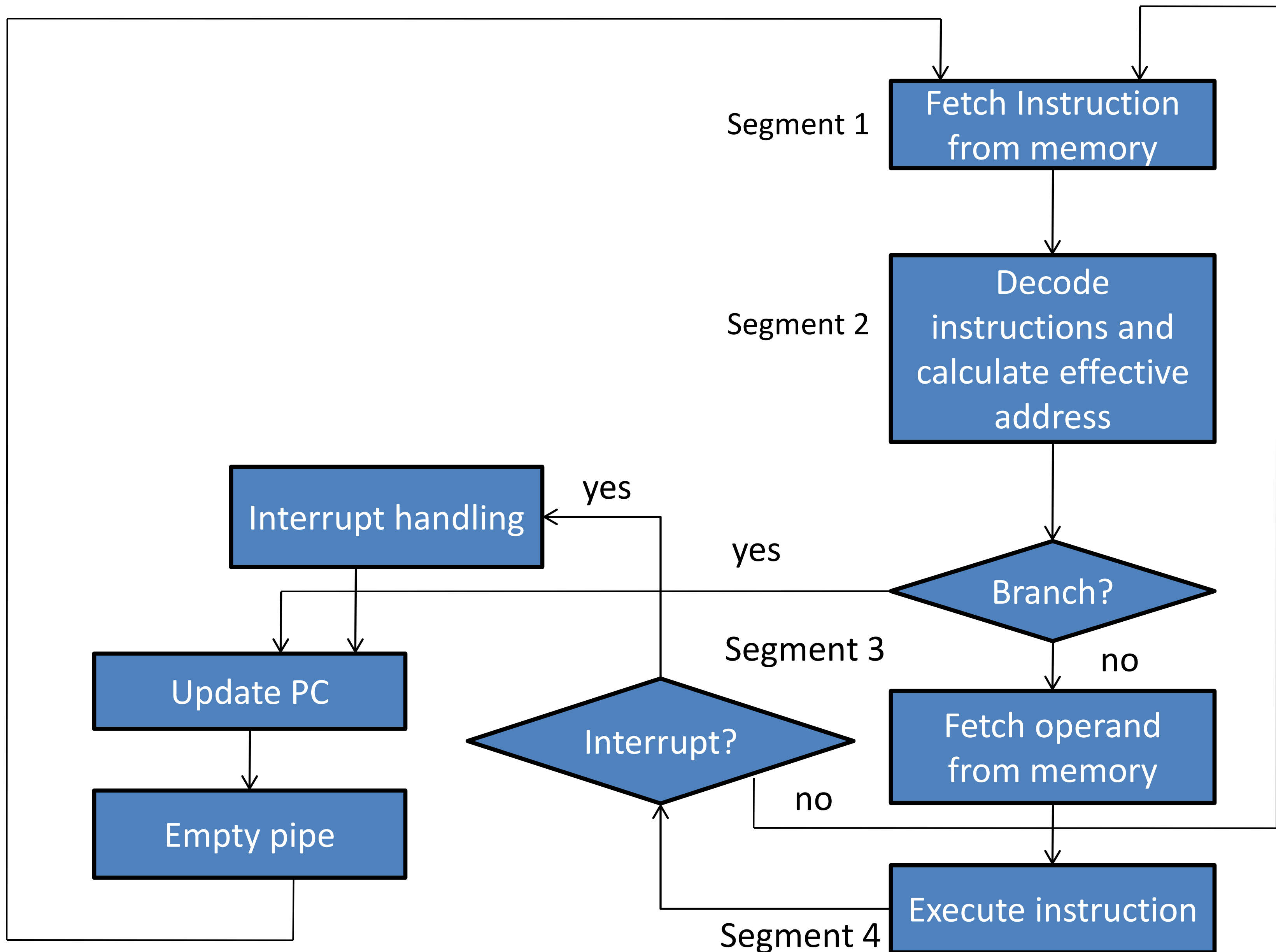
An Instruction pipeline operates on a stream of instructions by overlapping the fetch, decode, and execute phases of instruction cycle.

The instruction pipeline reads consecutive instructions from memory while previous instruction are being executed in other segments. This type of unit that forms a queue rather than stack. The instructions are inserted into FIFO buffer so that they can be executed on a first in first out basis. Thus the instruction stream can be placed in a queue, waiting for decoding and processing by execution segment.

In most general case, the computer needs to process each instruction with the following sequence of steps:

1. Fetch the instruction from memory.
2. Decode the instruction.
3. Calculate the effective address.
4. Fetch the operands from memory.
5. Execute the instruction.
6. Store the result in the proper place.

Example –Four segment CPU pipeline



Pipeline Conflicts

There are three major difficulties that cause the instruction pipeline to deviate from its normal operation

1. Resource conflicts caused by access to memory by two segments at the same time. Most of these conflicts can be resolved by using separate instruction and data memories.
2. Data dependency conflicts arise when an instruction depends on the result of a previous instruction, but this result is not yet available.
3. Branch difficulties arise from branch and other instructions that change the value of PC.

Vector Processing

Vector processing used in vast number of computations that will take a conventional computer days or even weeks to complete. In many science and engineering applications, the problems can be formulated in terms of vector and matrices that land themselves to vector processing.

Use of vector processing

1 Long range weather forecasting.
2 Petroleum exploration.
3 Seismic data analysis.
4 Medical diagnosis.
5 Aerodynamics and space flight simulation.
6 Artificial Intelligence and expert systems.
7 Mapping the human genome.
8 Image processing.

What is Superscalar?

- Common instructions (arithmetic, load/store, conditional branch) can be initiated and executed independently
- Equally applicable to RISC & CISC
- In practice usually RISC

Why Superscalar?

- Most operations are on scalar quantities
- Improve these operations to get an overall improvement

Superpipelined

- Many pipeline stages need less than half a clock cycle
- Double internal clock speed gets two tasks per external clock cycle
- Superscalar allows parallel fetch execute

Superscalar and Superpipelined Processors

- Logical evolution of pipeline designs resulted in 2 high-performance execution techniques
- Superpipeline designs
  - Observation: a large number of operations do not require the full clock cycle to complete
  - High performance can be obtained by subdividing the clock cycle into a number of sub intervals
    - » Higher clock frequency!
  - Subdivide the “macro” pipeline H/W stages into smaller (thus faster) substages and clock data through at the higher clock rate
  - Time to complete individual instructions does not change
    - » Degree of parallelism goes up
    - » Perceived speedup goes up

- Superscalar
  - Implement the CPU such that more than one instruction can be performed (completed) at a time
  - Involves replication of some or all parts of the CPU/ALU
  - Examples:
    - » Fetch multiple instructions at the same time
    - » Decode multiple instructions at the same time
    - » Perform add and multiply at the same time
    - » Perform load/stores while performing ALU operation
  - Degree of parallelism and hence the speedup of the machine goes up as more instructions are executed in parallel

Superscalar design limitations

- Data dependencies: must insure computed results are the same as would be computed on a strictly sequential machine
  - Two instructions can not be executed in parallel if the (data) output of one is the input of the other or if they both write to the same output location
  - Consider:

S1:	A = B + C
S2:	D = A + 1
S3:	B = E + F
S4:	A = E + 3

- Resource dependencies
  - In the above sequence of instructions, the adder unit gets a real workout!
  - Parallelism is limited by the number of adders in the ALU

- Instruction issue policy: in what order are instructions issued to the execution unit and in what order do they finish?
  - In-order issue, in-order completion
    - » Simplest method, but severely limits performance
    - » Strict ordering of instructions: data and procedural dependencies or resource conflicts delay all subsequent instructions
    - » "Slow" execution of some instructions delay all subsequent instructions
  - In-order issue, out-of-order completion
    - » Any number of instructions can be executed at a time
    - » Instruction issue is still limited by resource conflicts or data and procedural dependencies
    - » Output dependencies resulting from out-of-order completion must be resolved
    - » "Instruction" interrupts can be tricky

- Out-of-order issue, out-of-order completion
  - » Decode and execute stages are decoupled via an instruction buffer "window"
  - » Decoded instructions are "stored" in the window awaiting execution
  - » Functional units will take instructions from the window in an attempt to stay busy
    - ◆ This can result in out-of-order execution

      S1: A = B + C
      S2: D = E + 1
      S3: G = E + F
      S4: H = E * 3

  - » "Antidependence" class of data dependencies must be dealt with

- Register renaming
  - Output dependencies and antidependencies are eliminated by the use of a register "pool" as follows
    - » For each instruction that writes to a register X, a "new" register X is instantiated
    - » Multiple "register Xs" can co-exist
  - Consider

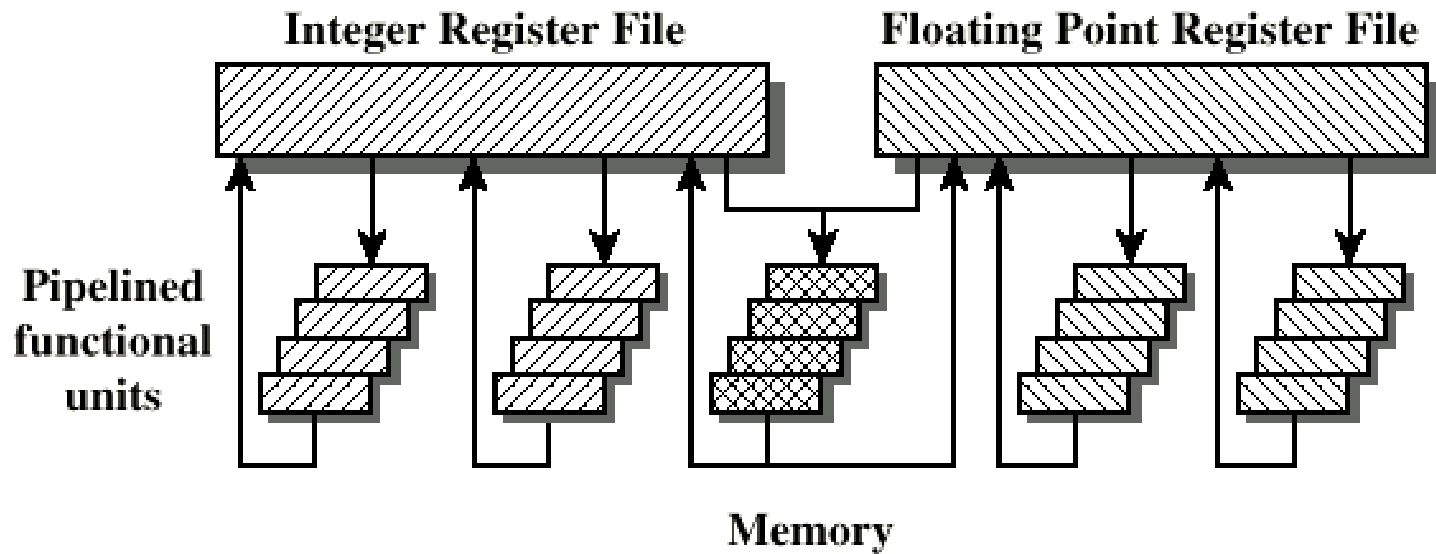
S1: R3 = R3 + R5
S2: R4 = R3 + 1
S3: R3 = R5 + 1
S4: R7 = R3 + R4

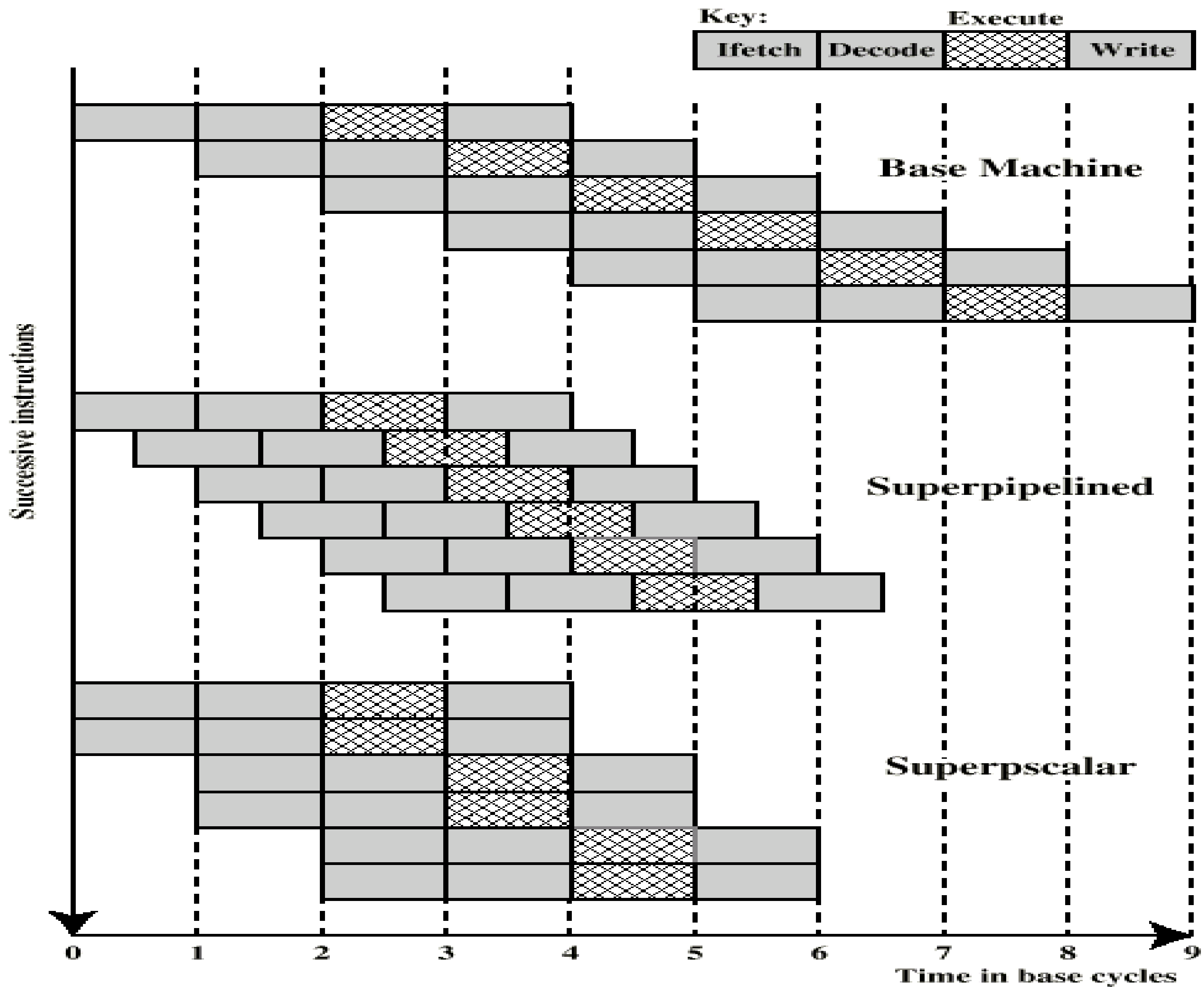
becomes

S1: R3b = R3a + R5a
S2: R4b = R3b + 1
S3: R3c = R5a + 1
S4: R7b = R3c + R4b

- Impact on machine parallelism
  - Adding (ALU) functional units without register renaming support may not be cost-effective
    - Performance is limited by data dependencies
  - Out-of-order issue benefits from large instruction buffer windows
    - Easier for a functional unit to find a pending instruction

General Superscalar Organization




Key:
Ifetch
Decode
Execute
Write
Base Machine
Superpipelined
Superpscalar
Successive instructions
0
1
2
3
4
5
6
7
8
9
Time in base cycles

Limitations

- Instruction level parallelism
- Compiler based optimisation
- Hardware techniques
- Limited by
  – True data dependency
  – Procedural dependency
  – Resource conflicts
  – Output dependency
  – Antidependency

Unit – 05

Memory Systems

&

Multiprocessor

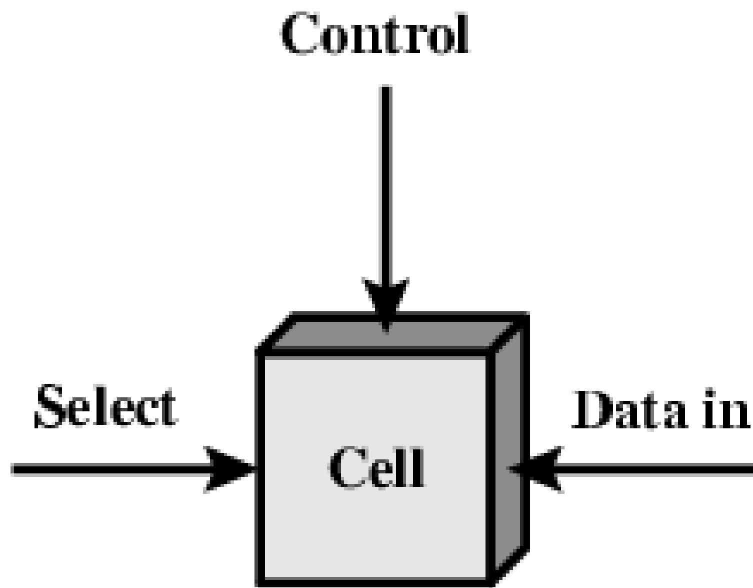
Semiconductor Memory

- RAM
  - Misnamed as all semiconductor memory is random access
  - Read/Write
  - Volatile (contents are lost when power switched off)
  - Temporary storage
  - Static or dynamic
    - Dynamic is based on capacitors – leaks thus needs refresh
    - Static is based on flip-flops – no leaks, does not need refresh

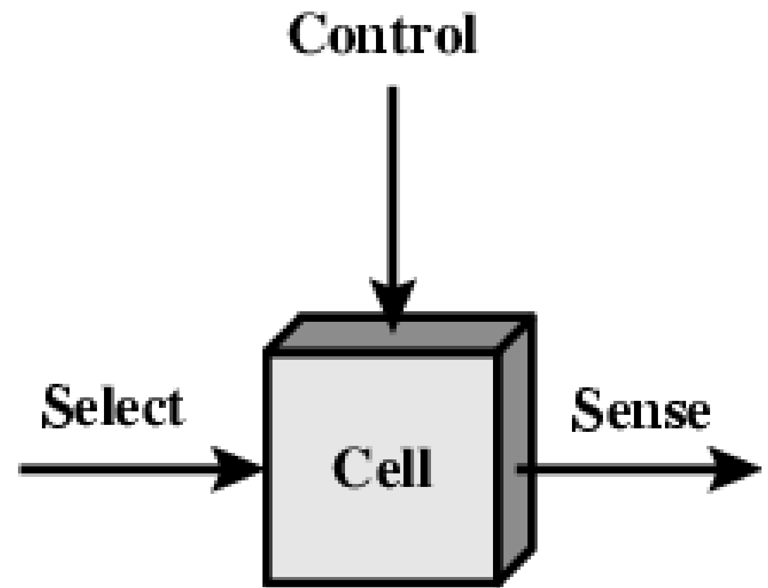
Semiconductor Memory Types

Memory Type	Category	Erasure	Write Mechanism	Volatility
Random-access memory (RAM)	Read-write memory	Electrically, byte-level	Electrically	Volatile
Read-only memory (ROM)	Read-only memory	Not possible	Masks	Nonvolatile
Programmable ROM (PROM)			Electrically	
Erasable PROM (EPROM)	Read-mostly memory	UV light, chip-level		
Electrically Erasable PROM (EEPROM)		Electrically, byte-level		
Flash memory		Electrically, block-level		

Memory Cell Operation



(a) Write



(b) Read

Dynamic RAM

- The bulk of a processor's main memory is comprised of dynamic RAM
- Manufacturers have focused on memory sizes rather than speed
- In contrast to SRAM, DRAM uses a single transistor and capacitor to store a bit
- DRAM requires that the address applied to the device be asserted in a row address (RAS) and a column address (CAS)
- The requirement of RAS and CAS of course kills the access time, but since it reduces package pinout, it allows for higher memory densities

Dynamic RAM

- RAS and CAS use the same pins, with each being asserted during either the RAS or the CAS phase of the address
- There are two metrics used to describe DRAM's performance:
  - *Access time* is defined as the time between assertion of RAS to the availability of data
  - Cycle time is defined as the minimum time before a next access can be granted
- Manufacturers like to quote access times, but cycle times are more relevant because they establish throughput of the system

Unit – 05

Memory Systems

&

Multiprocessor

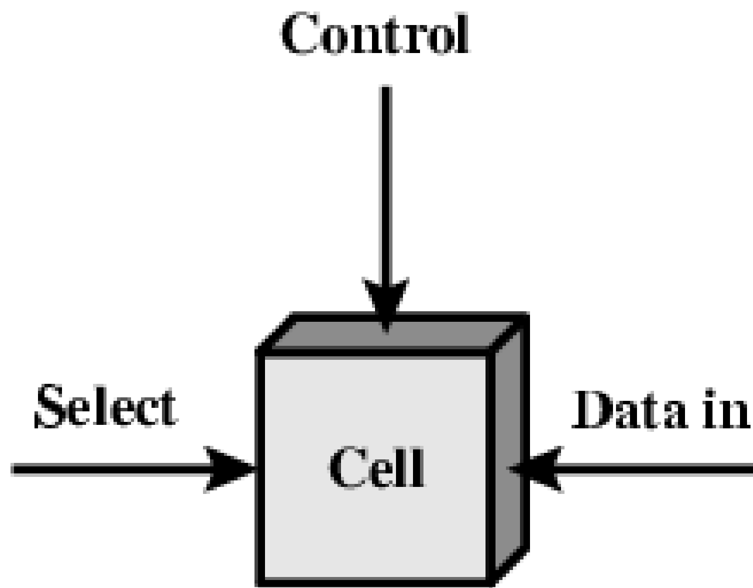
Semiconductor Memory

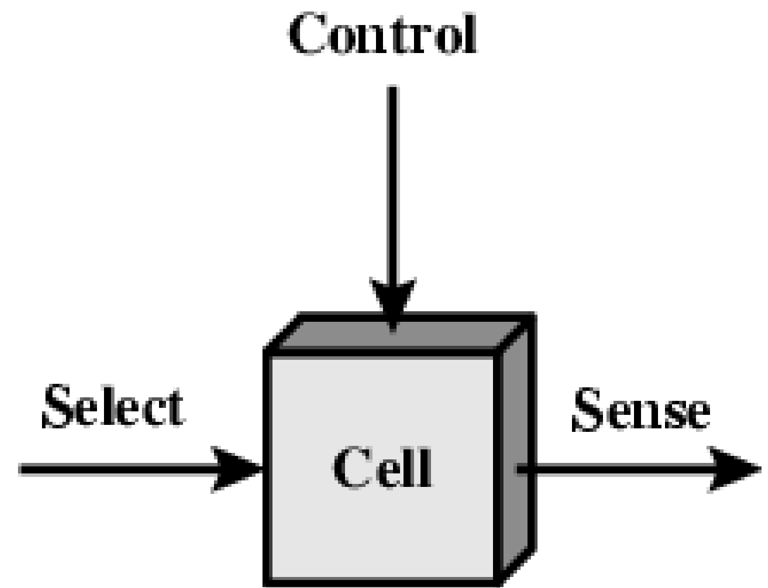
- RAM
  - Misnamed as all semiconductor memory is random access
  - Read/Write
  - Volatile (contents are lost when power switched off)
  - Temporary storage
  - Static or dynamic
    - Dynamic is based on capacitors – leaks thus needs refresh
    - Static is based on flip-flops – no leaks, does not need refresh

Semiconductor Memory Types

Memory Type	Category	Erasure	Write Mechanism	Volatility
Random-access memory (RAM)	Read-write memory	Electrically, byte-level	Electrically	Volatile
Read-only memory (ROM)	Read-only memory	Not possible	Masks	Nonvolatile
Programmable ROM (PROM)			Electrically	
Erasable PROM (EPROM)	Read-mostly memory	UV light, chip-level		
Electrically Erasable PROM (EEPROM)		Electrically, byte-level		
Flash memory		Electrically, block-level		

Memory Cell Operation





Dynamic RAM

- The bulk of a processor's main memory is comprised of dynamic RAM
- Manufacturers have focused on memory sizes rather than speed
- In contrast to SRAM, DRAM uses a single transistor and capacitor to store a bit
- DRAM requires that the address applied to the device be asserted in a row address (RAS) and a column address (CAS)
- The requirement of RAS and CAS of course kills the access time, but since it reduces package pinout, it allows for higher memory densities

Dynamic RAM

- RAS and CAS use the same pins, with each being asserted during either the RAS or the CAS phase of the address
- There are two metrics used to describe DRAM's performance:
  - *Access time* is defined as the time between assertion of RAS to the availability of data
  - Cycle time is defined as the minimum time before a next access can be granted
- Manufacturers like to quote access times, but cycle times are more relevant because they establish throughput of the system

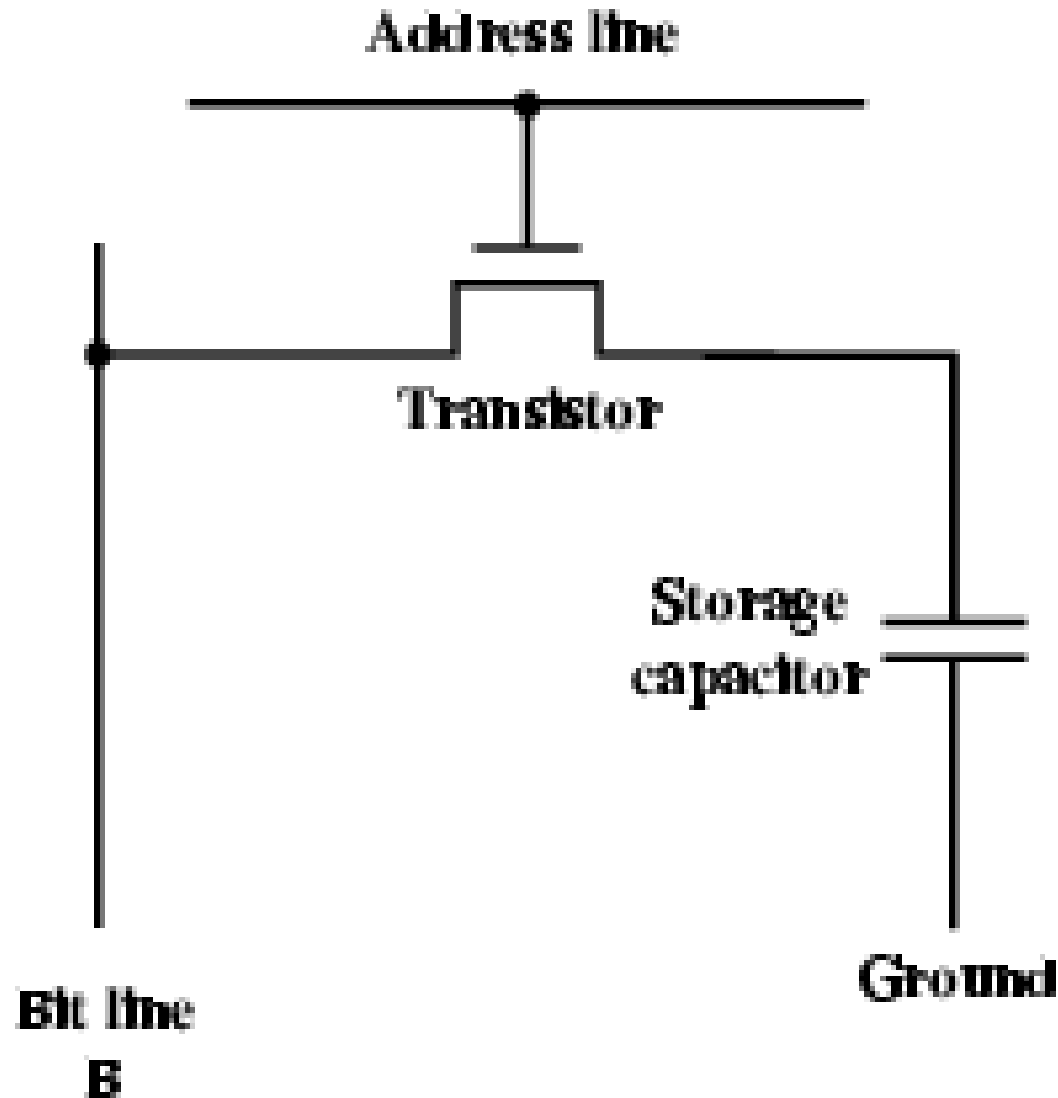
Dynamic RAM

- Of course the charge leaks slowly from the storage capacitor in DRAM and it needs to be refreshed continually
- During the refresh phase, all accesses are *held-off*, which happens once every 1 – 100 ms and slightly impacts the throughput
- DRAM bandwidth can be increased by operating it in *paged mode*, when several CASs are applied for each RAS
- A notation such as 256x16 means 256 thousand columns of cells standing 16 rows deep

Dynamic RAM

- Bits stored as charge in capacitors
- Charges leak
- Need refreshing even when powered
- Simpler construction
- Smaller per bit
- Less expensive
- Need refresh circuits
- Slower
- Used in main memory
- Essentially analogue
  - Level of charge determines value

Dynamic RAM Structure



DRAM Operation

- Address line active when bit read or written
  - Transistor switch closed (current flows)
- Write
  - Voltage to bit line
    - High for 1 low for 0
  - Then signal address line
    - Transfers charge to capacitor
- Read
  - Address line selected
    - transistor turns on
  - Charge from capacitor fed via bit line to sense amplifier
    - Compares with reference value to determine 0 or 1
  - Capacitor charge must be restored

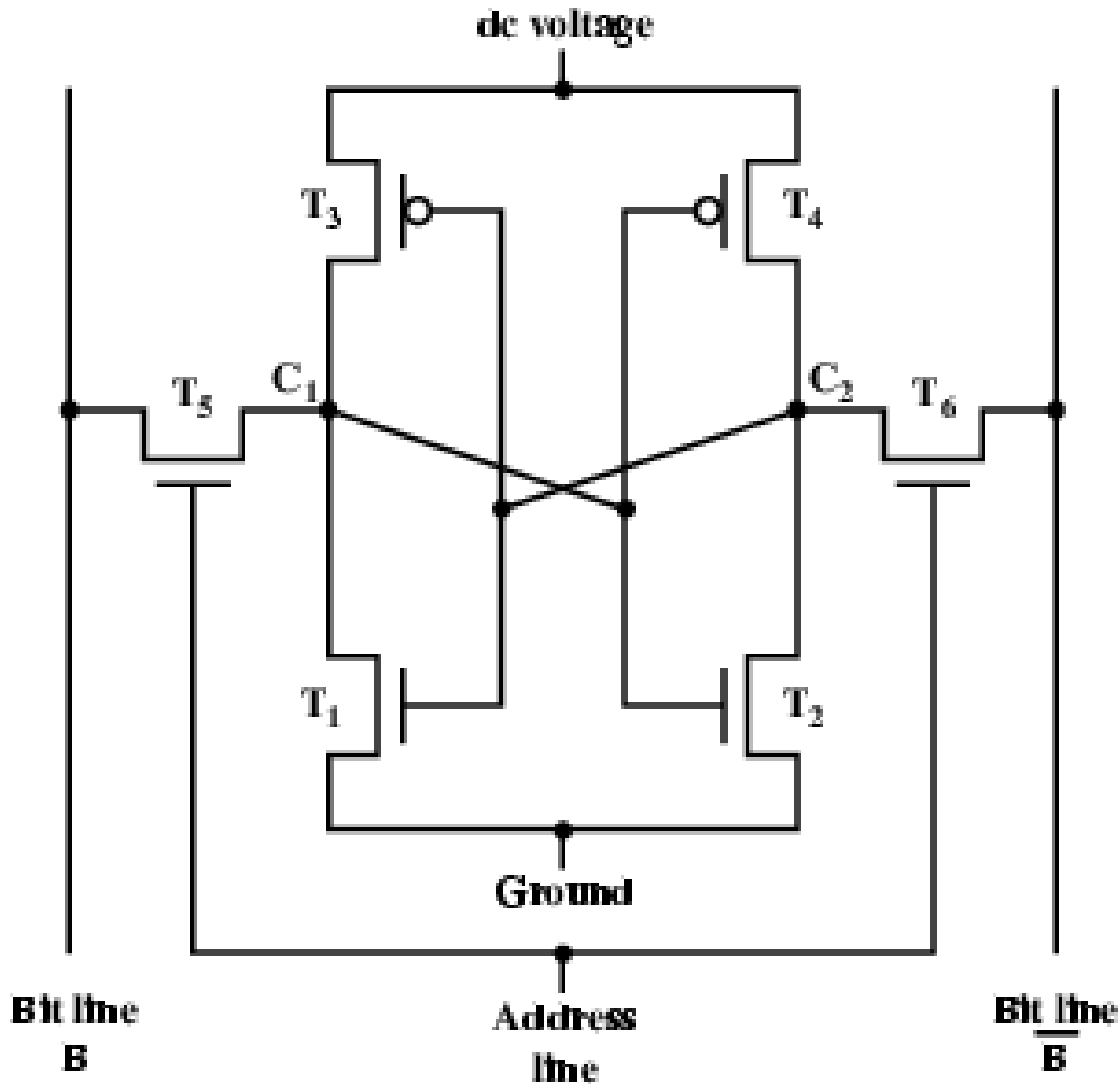
Static RAM

- Uses 4-6 transistors to store a single bit of data
- Provides a fast access time at the expense of lower bit densities
- For this reason registers and cache subsystems are fabricated using SRAM technology
- Static RAM is considerably more expensive than Dynamic RAM
- However, since it doesn't need to be refreshed, its power consumption is much lower than DRAM
- Also, the absence of the refresh circuitry makes it easier to interface to
- The simplicity of the memory circuitry compensates for the more costly technology

Static RAM

- Bits stored as on/off switches
- No charges to leak
- No refreshing needed when powered
- More complex construction
- Larger per bit
- More expensive
- Does not need refresh circuits
- Faster
- Cache
- Digital
  - Uses flip-flops

Static RAM Structure



Static RAM Operation

- Transistor arrangement gives stable logic state
- State 1
  - C_1 high, C_2 low
  - T_1 T_4 off, T_2 T_3 on
- State 0
  - C_2 high, C_1 low
  - T_2 T_3 off, T_1 T_4 on
- Address line transistors T_5 T_6 is switch
- Write – apply value to B & compliment to $\overline{B}$
- Read – value is on line B

SRAM v DRAM

- Both volatile
  - Power needed to preserve data
- Dynamic cell
  - Simpler to build, smaller
  - More dense
  - Less expensive
  - Needs refresh
  - Larger memory units
- Static
  - Faster
  - Doesn’t need refresh
  - Cache
  - Consumes more power

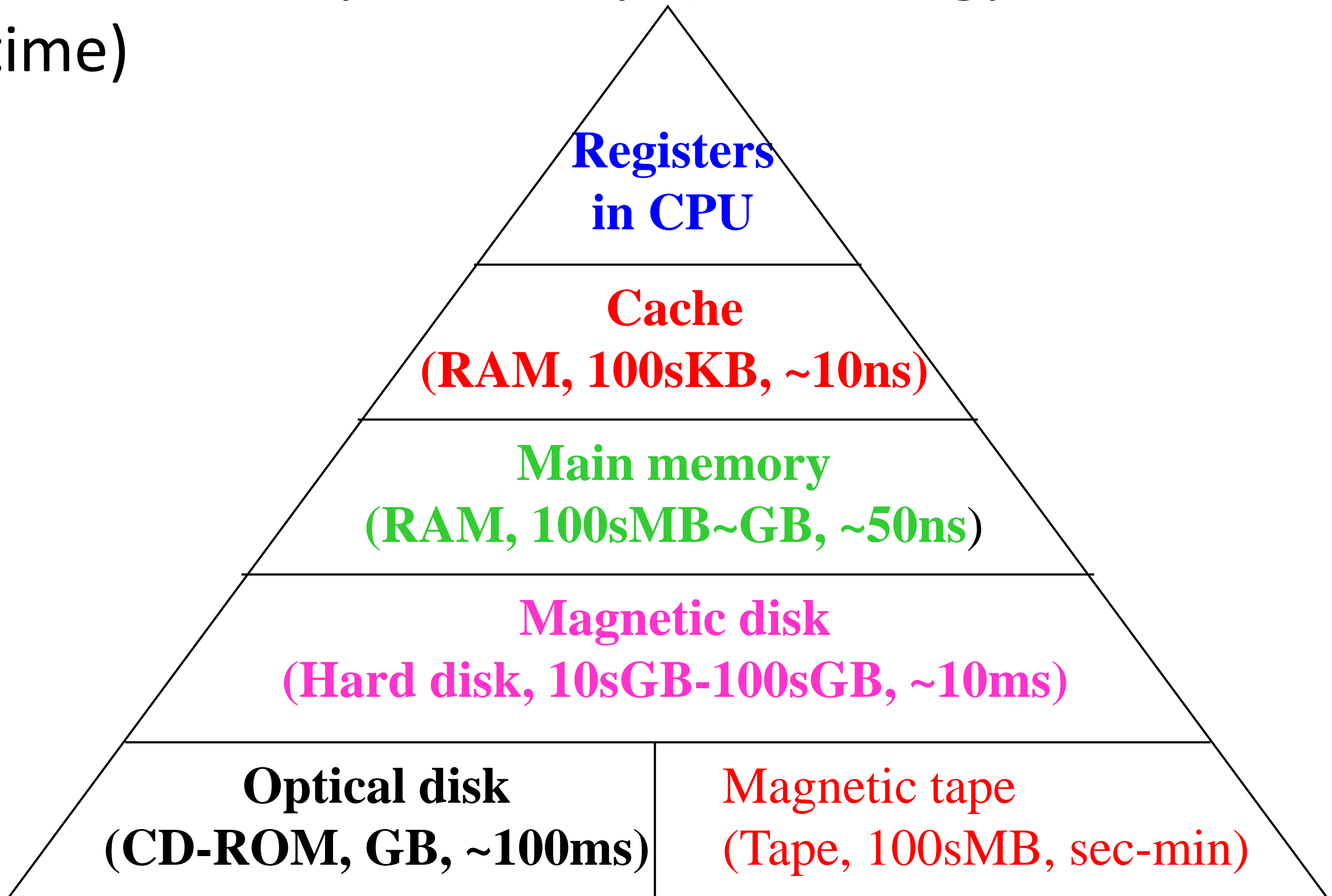
Memory

Memory Hierarchy

- Major design objective - to provide adequate storage capacity
  - at an acceptable level of performance
  - at a reasonable cost

- The use of a hierarchy of storage devices can meet this goal:
- Registers internal to the CPU for temporary data storage (small in number but very fast)
  - External storage for data and programs (relatively large and fast)
  - External permanent storage (much larger and much slower)

Typical memory hierarchy: (Technology, size, access time)



- Each decreasing level in the hierarchy consists of modules of larger capacity, slower access time, and lower cost/bit.

- Goal of the memory hierarchy - try to match the processor speed with the rate of information transfer from the highest element in the hierarchy.

- The memory hierarchy works because of *locality of reference*
  - Memory references made by the processor, for both instructions and data, tend to cluster together
    - » Instruction loops, subroutines
    - » Data arrays, tables
  - Keep these clusters in high speed memory (e.g. cache memory) to reduce the average delay in accessing data;
  - Over time, the clusters being referenced will change - memory management must deal with this (i.e. cache replacement).

Main Memory

- A main memory is a collection of *words*, used for storing programs or data; each word may consist of one or more bytes; conventionally, one word = two bytes.
- Physically, a memory of *N* words can be constructed using an *N*-word SRAM or DRAM, as described in the previous chapter.
- Logically, a memory of *N* words is like an array of *N* elements in Java or any other high-level language; e.g. a 10-element array in Java: `int MEM = new int[10].`
- A byte or word in a memory is often called a **memory cell**. Each cell in the memory can be located individually by its **address**, and thus written to or read from.
- The difference between an address and what is stored at that address cannot be over-emphasized; e.g. address $1000 may contain any bit pattern.

Addressing main memory

- We draw a memory as an array of 16-bit (2-byte) words, and consider the addresses for storing bytes, words and long words in it.

- **Bytes**. Byte is the smallest unit that can be addressed. Bytes are addressed as follows:

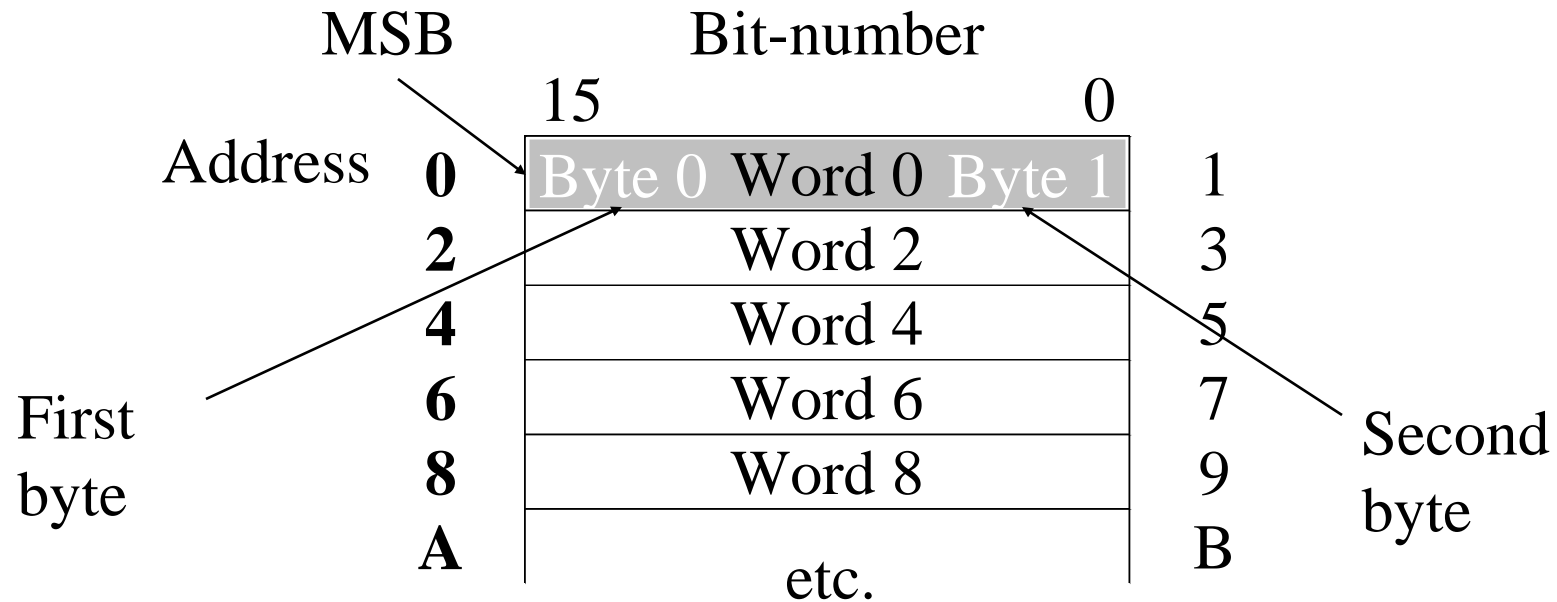
Bit-number

Address	15 … 8	7 … 0	Address
0	Byte 0	Byte 1	1
2	Byte 2	Byte 3	3
4	Byte 4	Byte 5	5
6	Byte 6	Byte 7	7
8	Byte 8	Byte 9	9
A			B

etc.

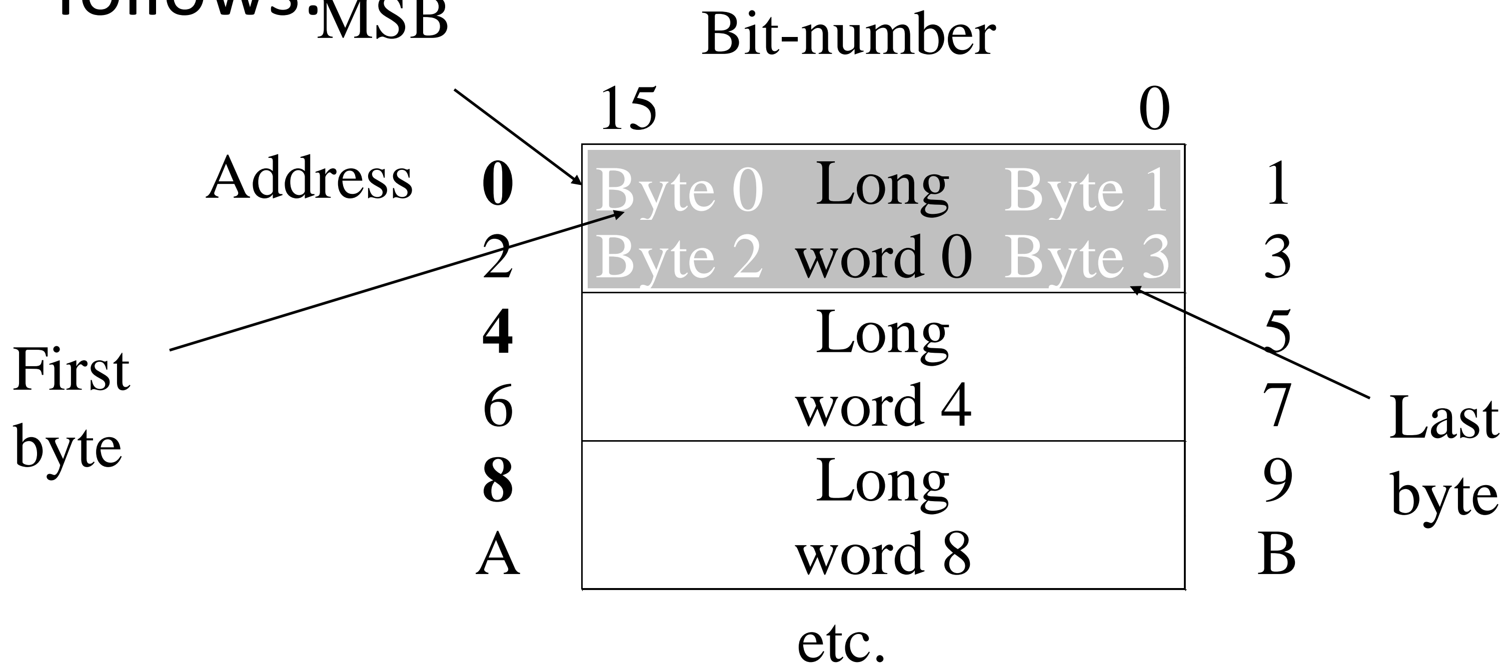
- Successive bytes in memory are stored at consecutive byte addresses, e.g. 0, 1, 2, 3…, as above.

- **Words**. Each consists of 2 bytes, addressed as follows:



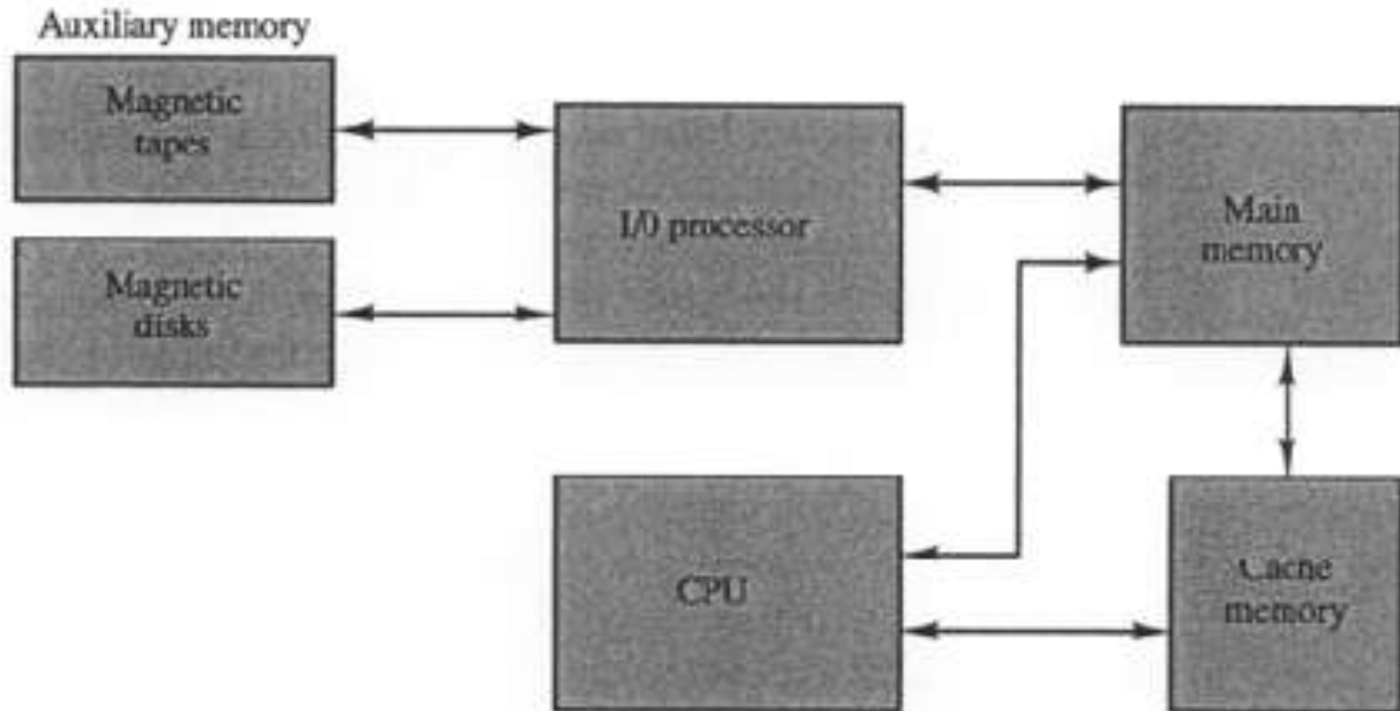
- Words are stored and accessed at **even** addresses (at even byte numbers), e.g. 0, 2, 4, 6…, as above.

- **Long words.** Each consists of 4 bytes, addressed as follows:

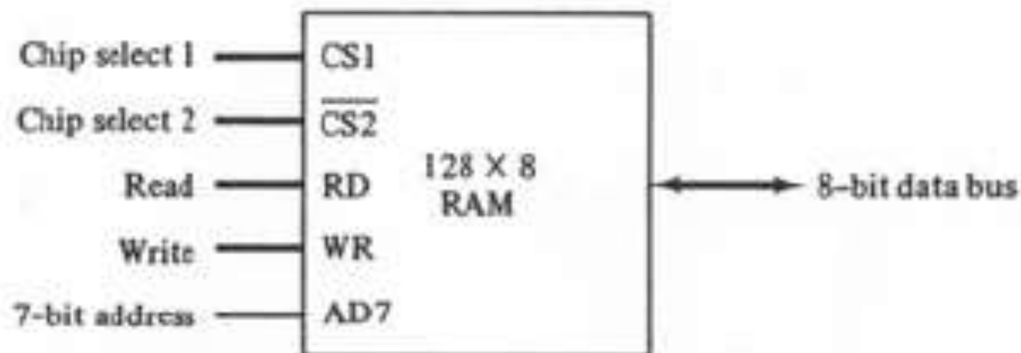


- Long words are stored and accessed at **even** addresses that are multiple of 4, e.g. 0, 4, 8, C…, as above.

Memory Hierarchy



RAM Chip

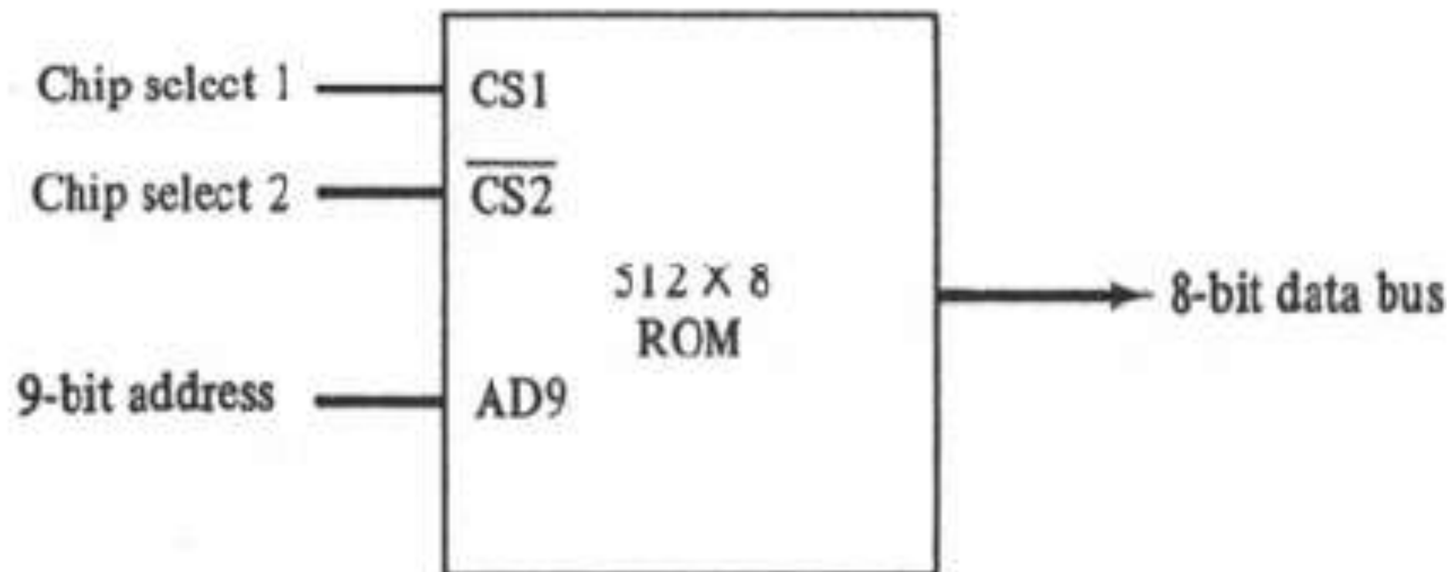


(a) Block diagram

CS1	$\overline{CS2}$	RD	WR	Memory function	State of data bus
0	0	×	×	Inhibit	High-impedance
0	1	×	×	Inhibit	High-impedance
1	0	0	0	Inhibit	High-impedance
1	0	0	1	Write	Input data to RAM
1	0	1	×	Read	Output data from RAM
1	1	×	×	Inhibit	High-impedance

(b) Function table

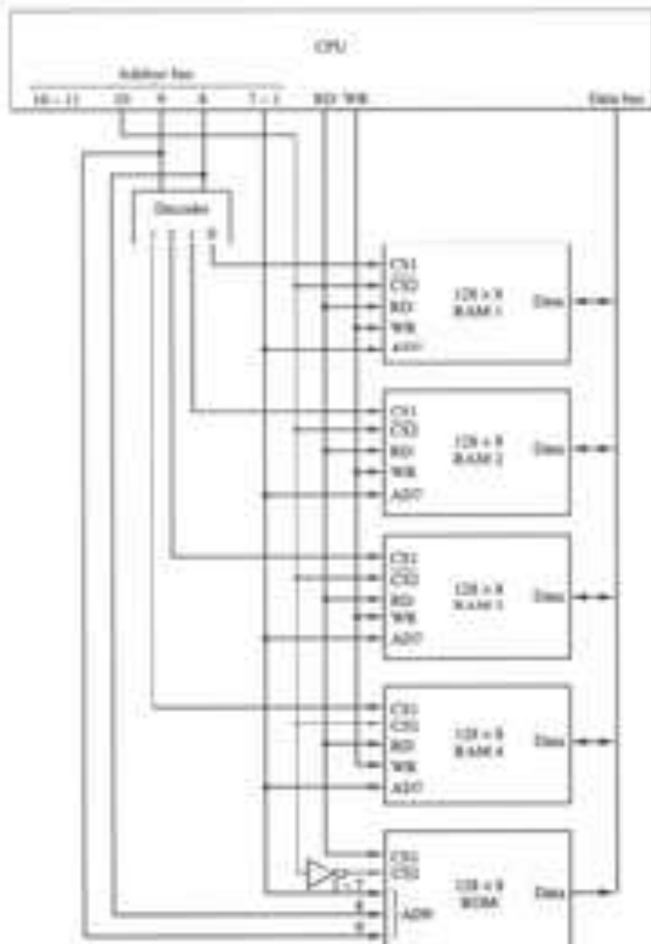
ROM Chip



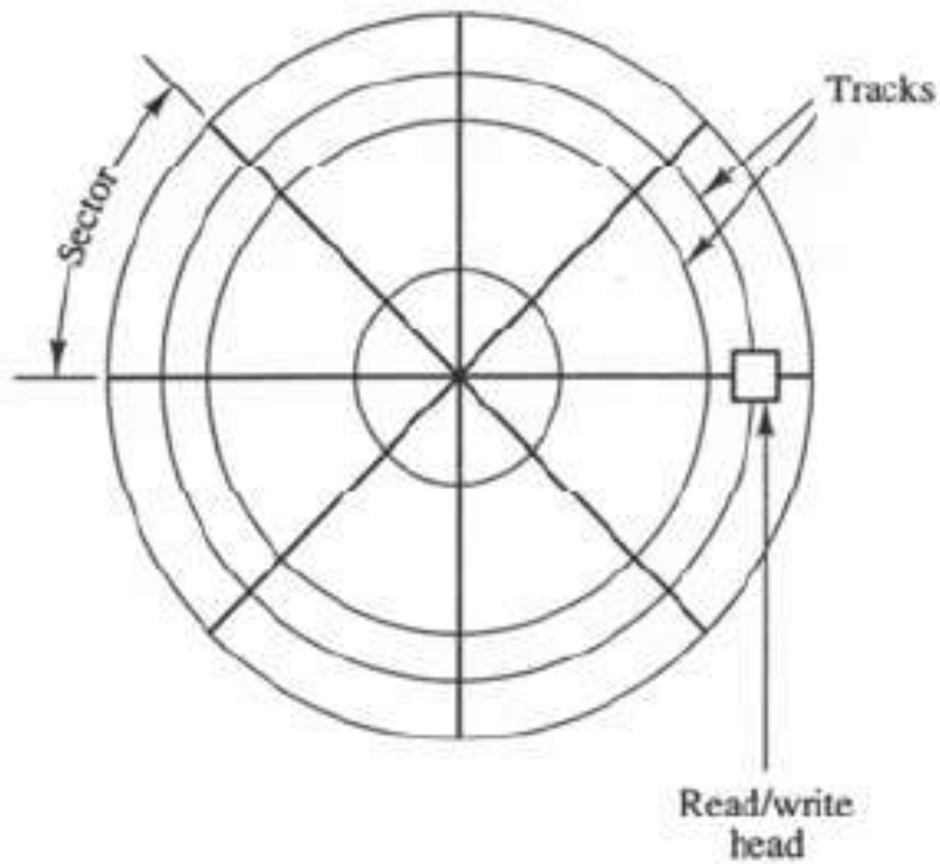
Memory Address Map

Component	Hexadecimal address	Address bus									
		10	9	8	7	6	5	4	3	2	1
RAM 1	0000–007F	0	0	0	x	x	x	x	x	x	x
RAM 2	0080–00FF	0	0	1	x	x	x	x	x	x	x
RAM 3	0100–017F	0	1	0	x	x	x	x	x	x	x
RAM 4	0180–01FF	0	1	1	x	x	x	x	x	x	x
ROM	0200–03FF	1	x	x	x	x	x	x	x	x	x

CPU / Memory Connection



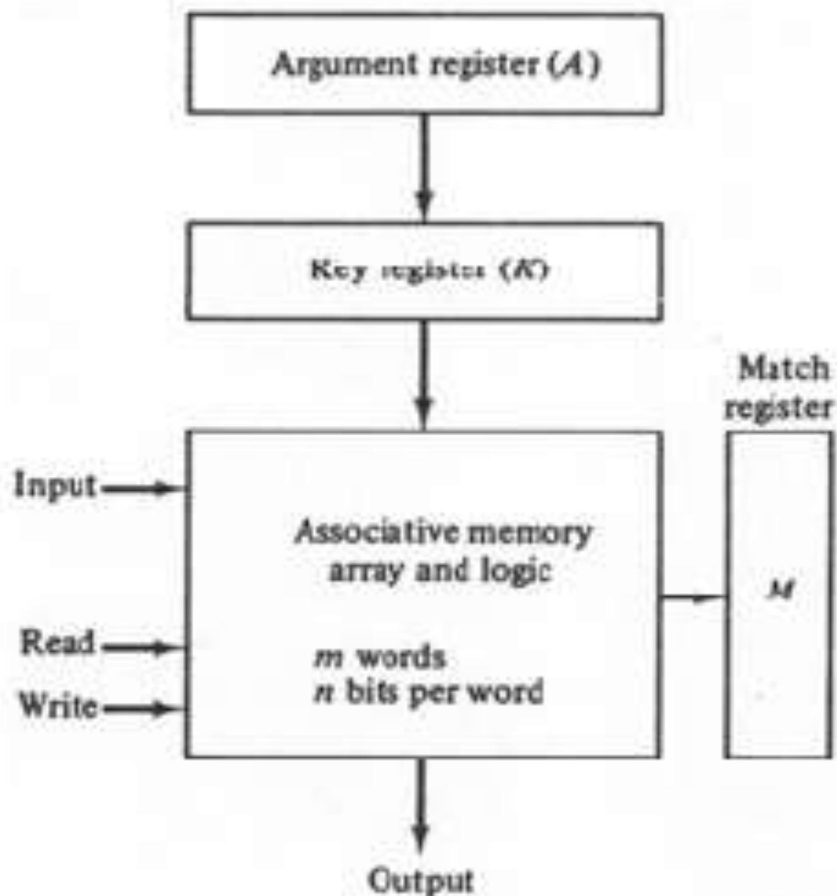
Magnetic Disk



Other Types of Magnetic Storage

- Head-Per-Track disk
- Drum
- Tape

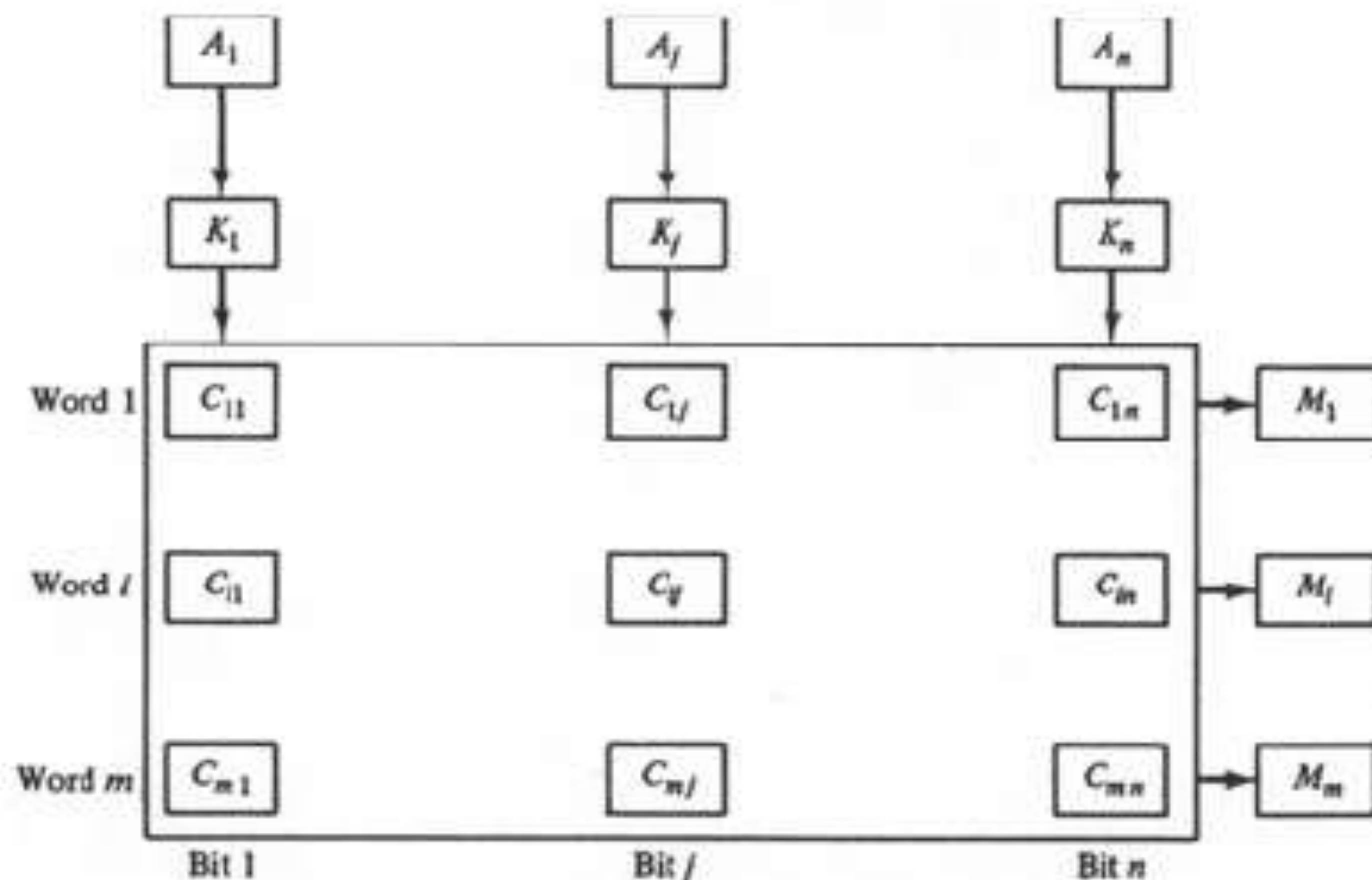
Associative Memory



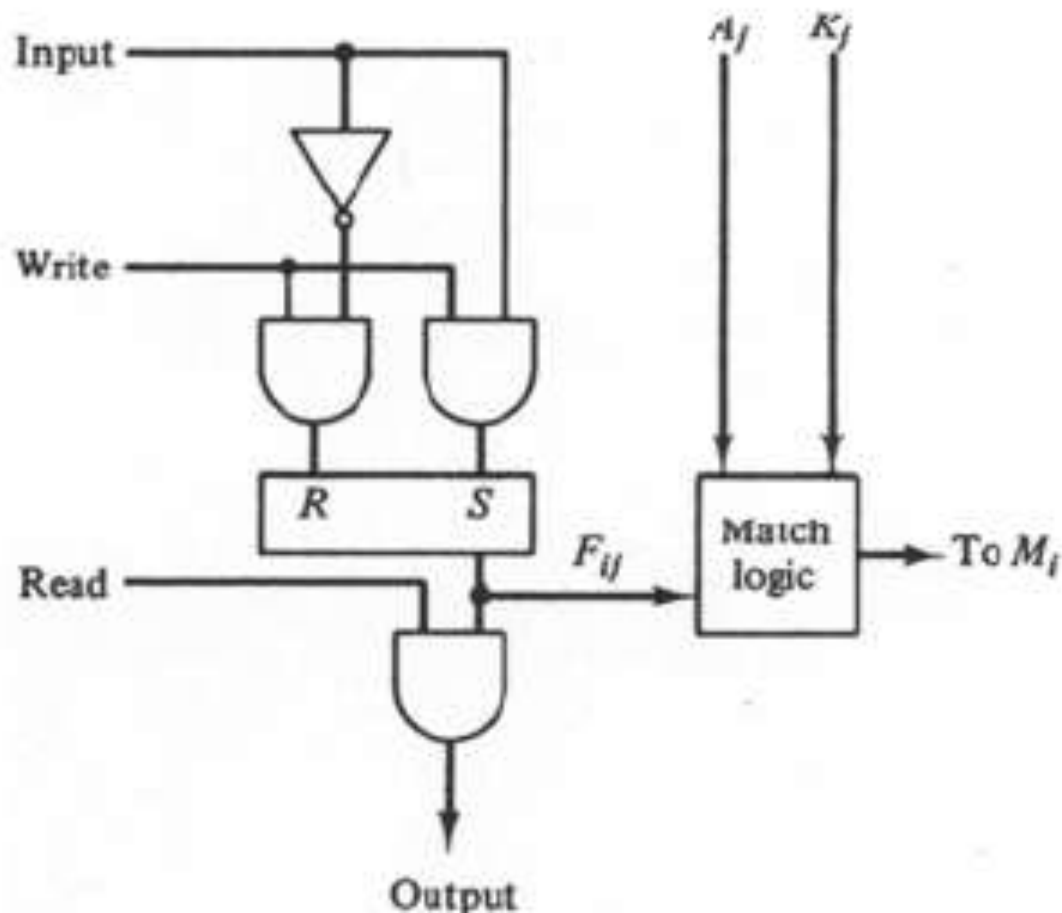
Associative Memory Example

- A 101 111100 Argument
- K 111 000000 Key (mask)
- Word 1 100 111100 no match
- Word 2 101 000001 match

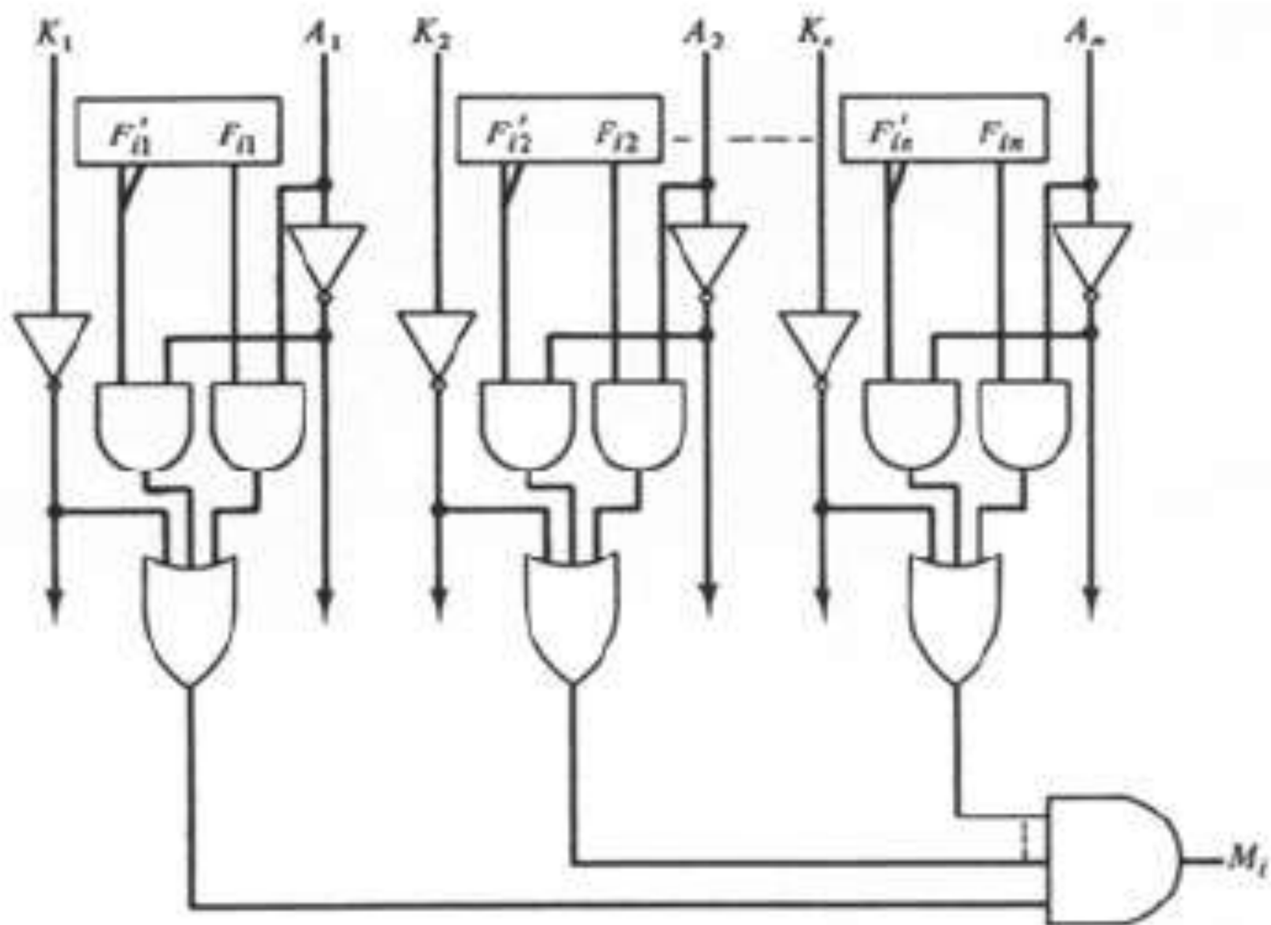
m Words, *n* Cells Per Word



One Cell Of Associative Memory



One Word Match Logic

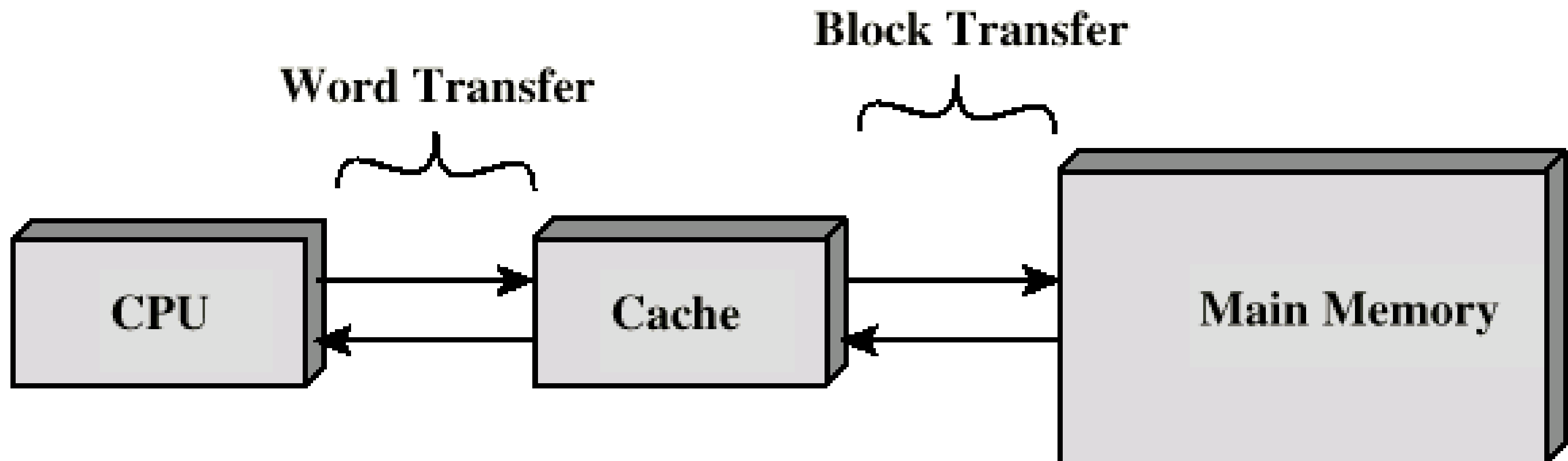


Tag Register

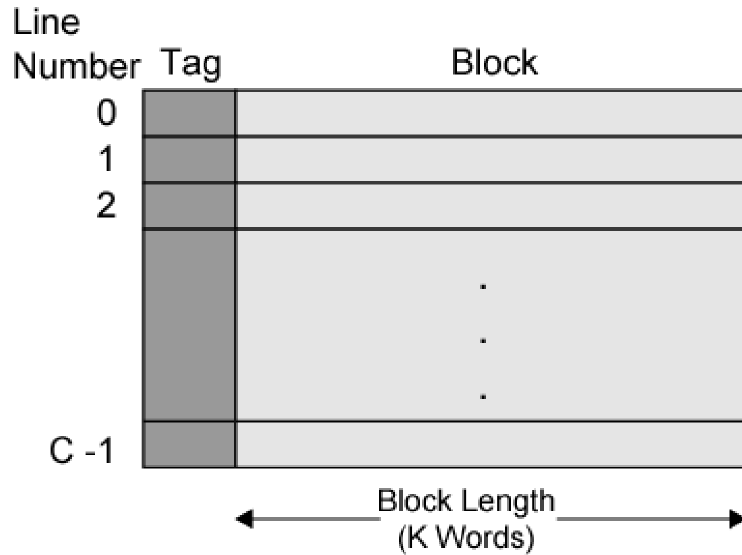
- One bit for each associative memory word
- Set to 1 for active word
- Reset to 0 for deleted word
- Masked along with argument word so only active words are compared

Cache

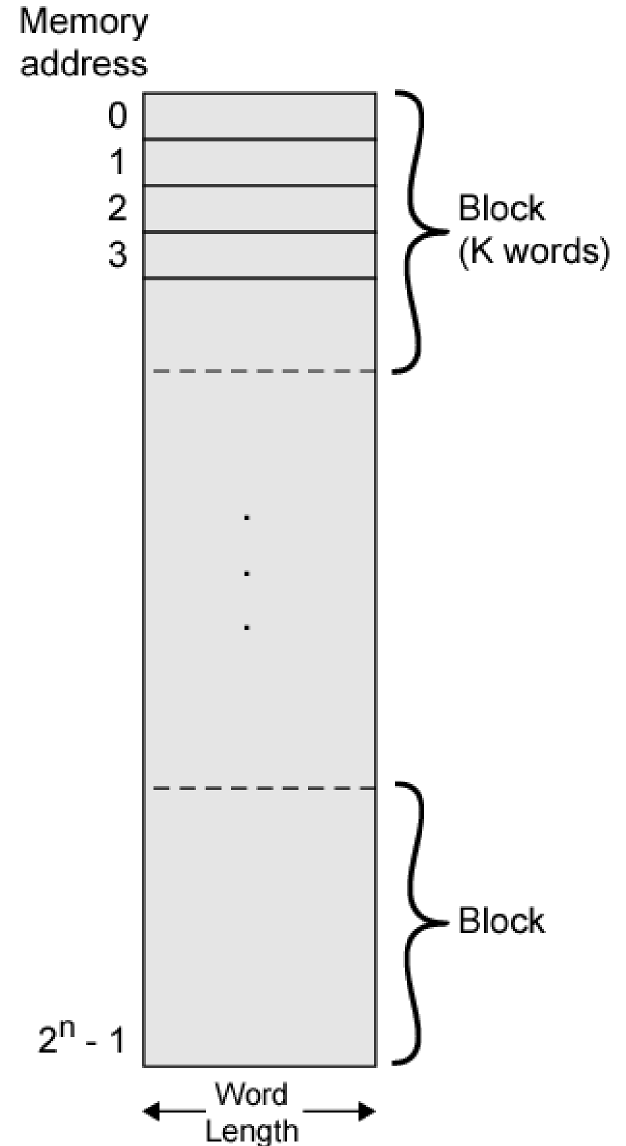
- Small amount of fast memory
- Sits between normal main memory and CPU
- May be located on CPU chip or module



Cache/Main Memory Structure



(a) Cache

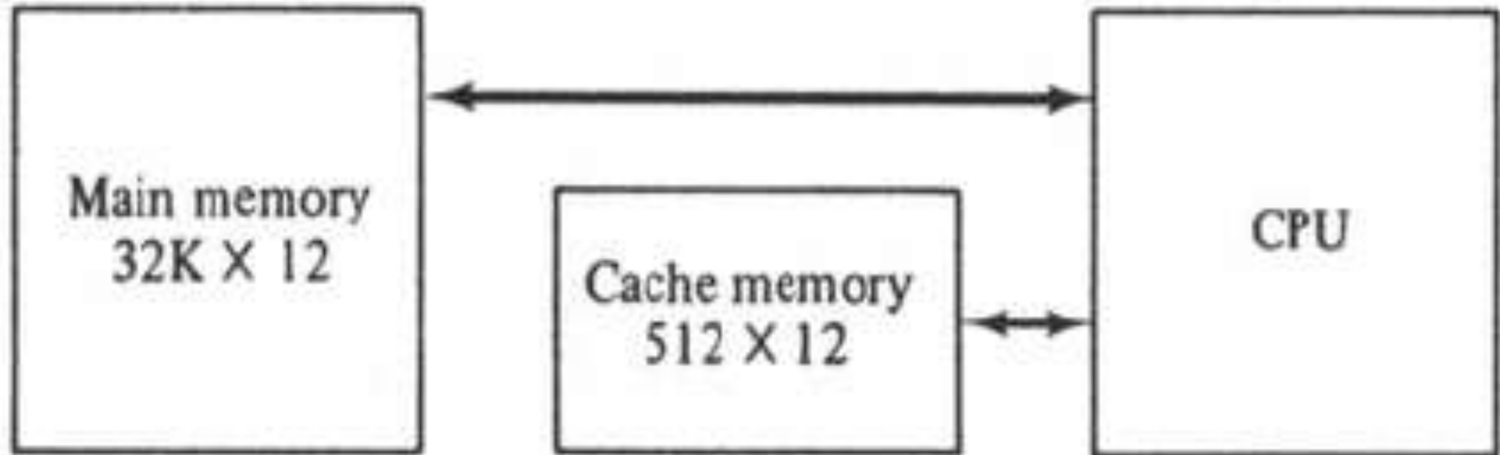


(b) Main memory

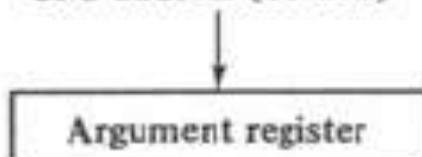
Cache Memory Terms

- Locality of reference
  - Memory references at any given interval of time tend to be confined within a few localized areas in memory
- Cache memory
  - Fast, small memory with fastest access speed compared to other memory types
- Hit ratio
  - Cache hits / (cache hits + cache misses)

Cache Memory



Associative Mapping Cache



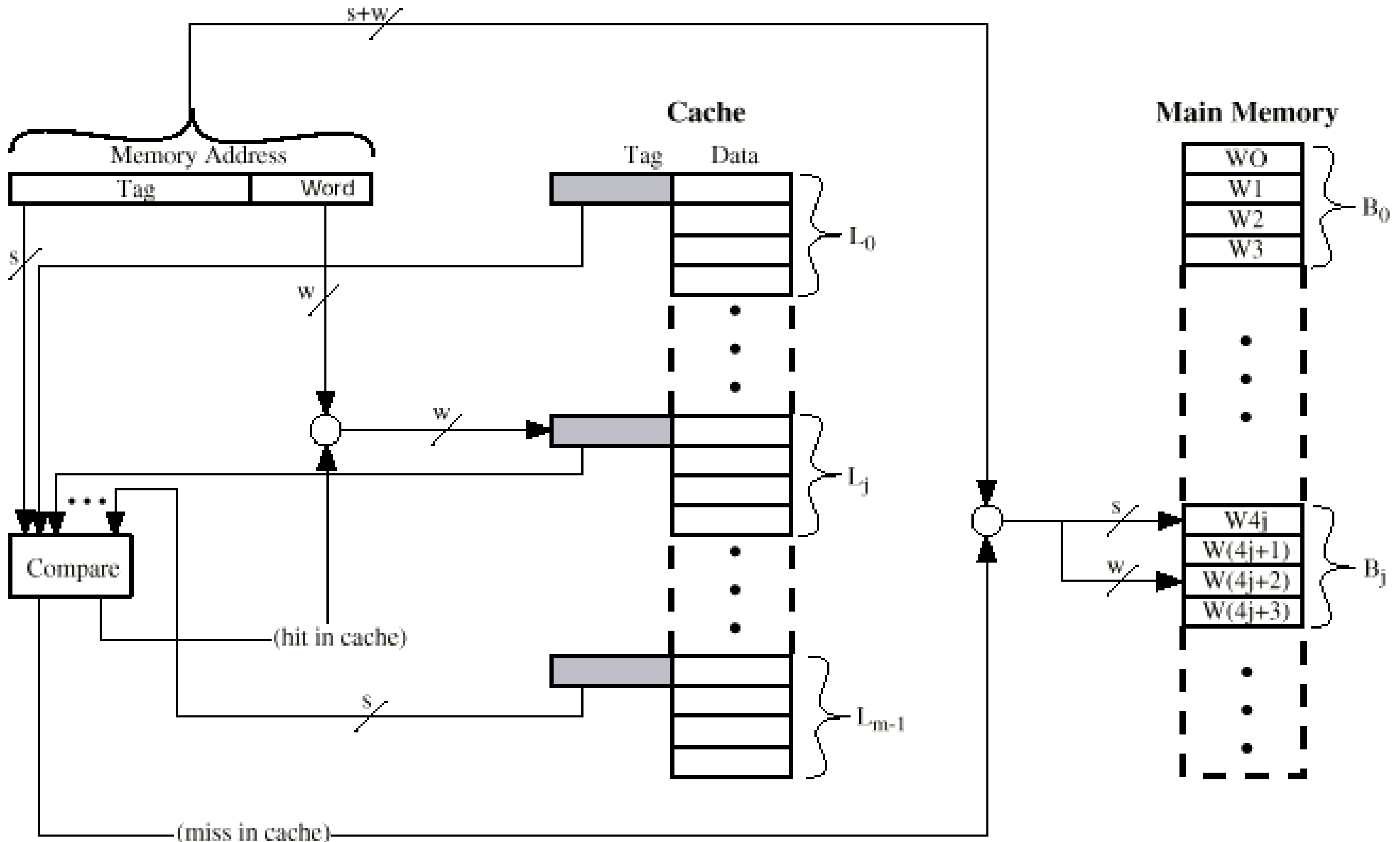
Address	Data
0 1 0 0 0	3 4 5 0
0 2 7 7 7	6 7 1 0
2 2 3 4 5	1 2 3 4

Numbers in octal

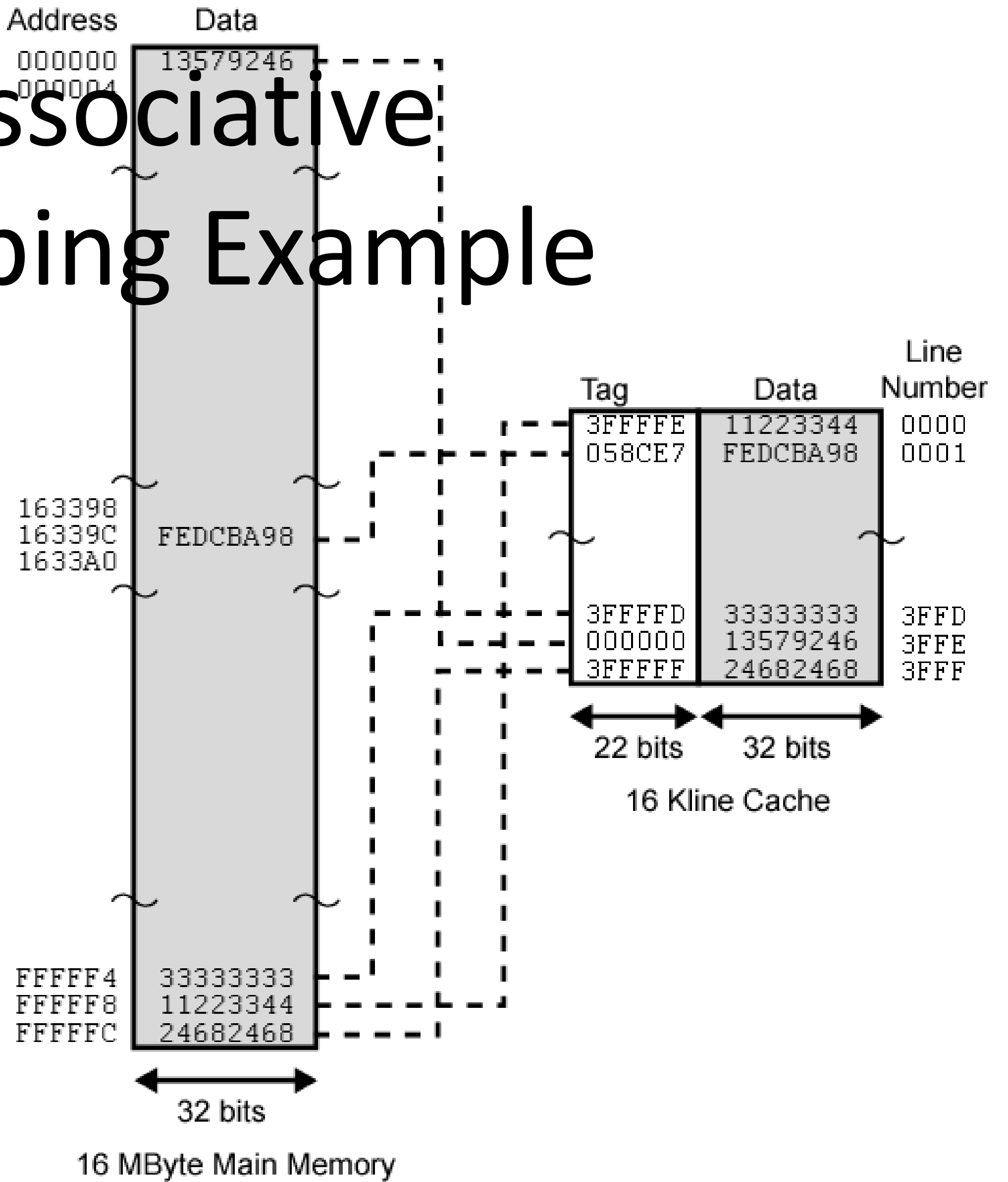
Associative Mapping

- A main memory block can load into any line of cache
- Memory address is interpreted as tag and word
- Tag uniquely identifies block of memory
- Every line’s tag is examined for a match
- Cache searching gets expensive

Fully Associative Cache Organization



Associative Mapping Example



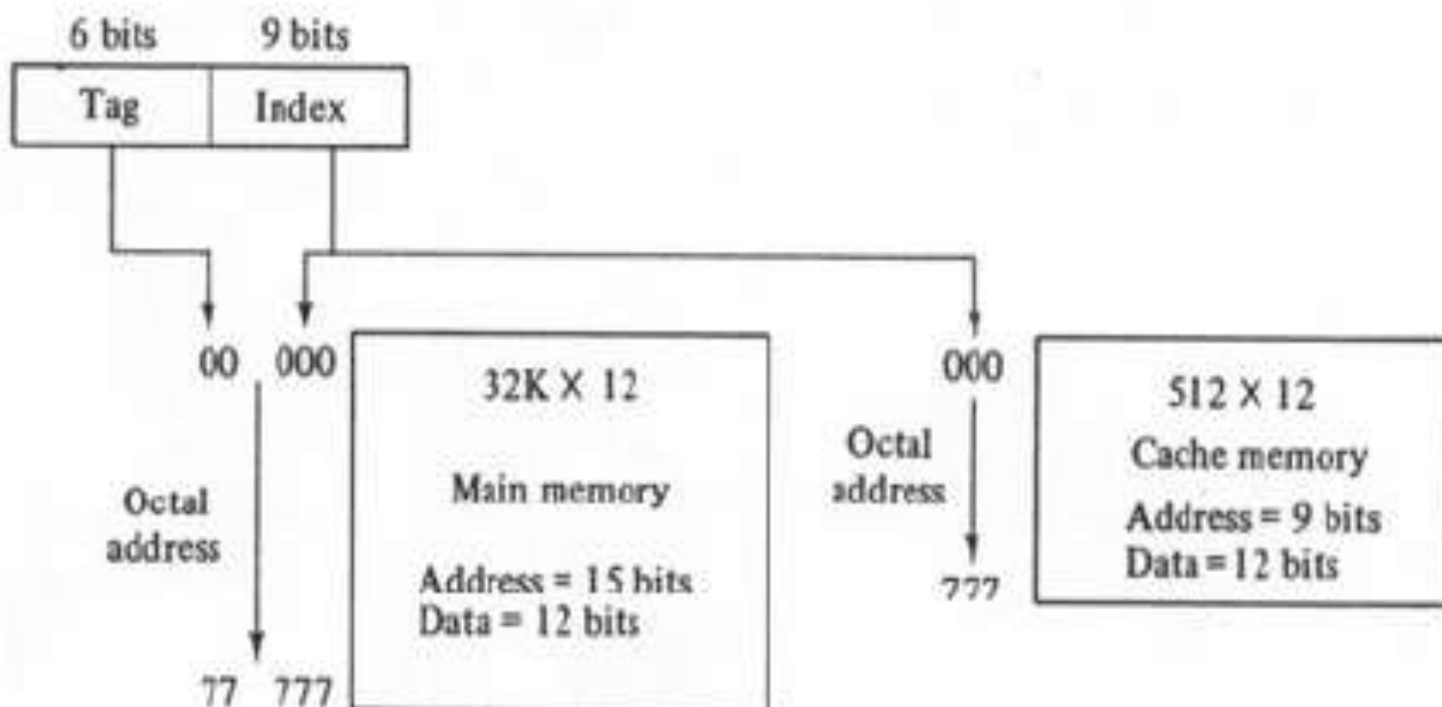
Associative Mapping Address Structure

- 22 bit tag stored with each 32 bit block of data
- Compare tag field with tag entry in cache to check for hit
- Least significant 2 bits of address identify which 16 bit word is required from 32 bit data block
- e.g.
  - Address Tag Data Cache line
  - FFFFFC FFFFFC 24682468 3FFF

Associative Mapping Summary

- Address length = (s + w) bits
- Number of addressable units = 2s+w words or bytes
- Block size = line size = 2w words or bytes
- Number of blocks in main memory = 2s+ w/2w = 2s
- Number of lines in cache = undetermined
- Size of tag = s bits

Main & Cache Relationships



Direct Mapping Cache

Memory address	Memory data
00000	1 2 2 0
00777	2 3 4 0
01000	3 4 5 0
01777	4 5 6 0
02000	5 6 7 0
02777	6 7 1 0

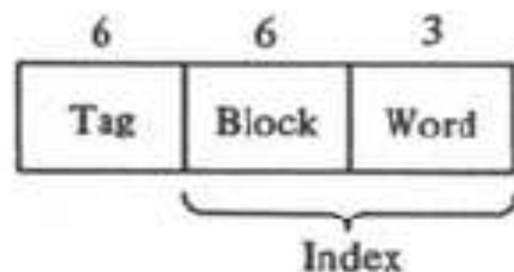
(a) Main memory

Index address	Tag	Data
000	0 0	1 2 2 0
777	0 2	6 7 1 0

(b) Cache memory

Direct With 8 Words / Block

	Index	Tag	Data
Block 0	000	0 1	3 4 5 0
	007	0 1	6 5 7 8
Block 1	010		
	017		
Block 63	770	0 2	
	777	0 2	6 7 1 0



Direct Mapping

- Each block of main memory maps to only one cache line
  - i.e. if a block is in cache, it must be in one specific place
- Address is in two parts
- Least Significant w bits identify unique word
- Most Significant s bits specify one memory block
- The MSBs are split into a cache line field r and a tag of s-r (most significant)

Direct Mapping Address Structure

Tag s-r	Line or Slot r	Word w
8	14	2

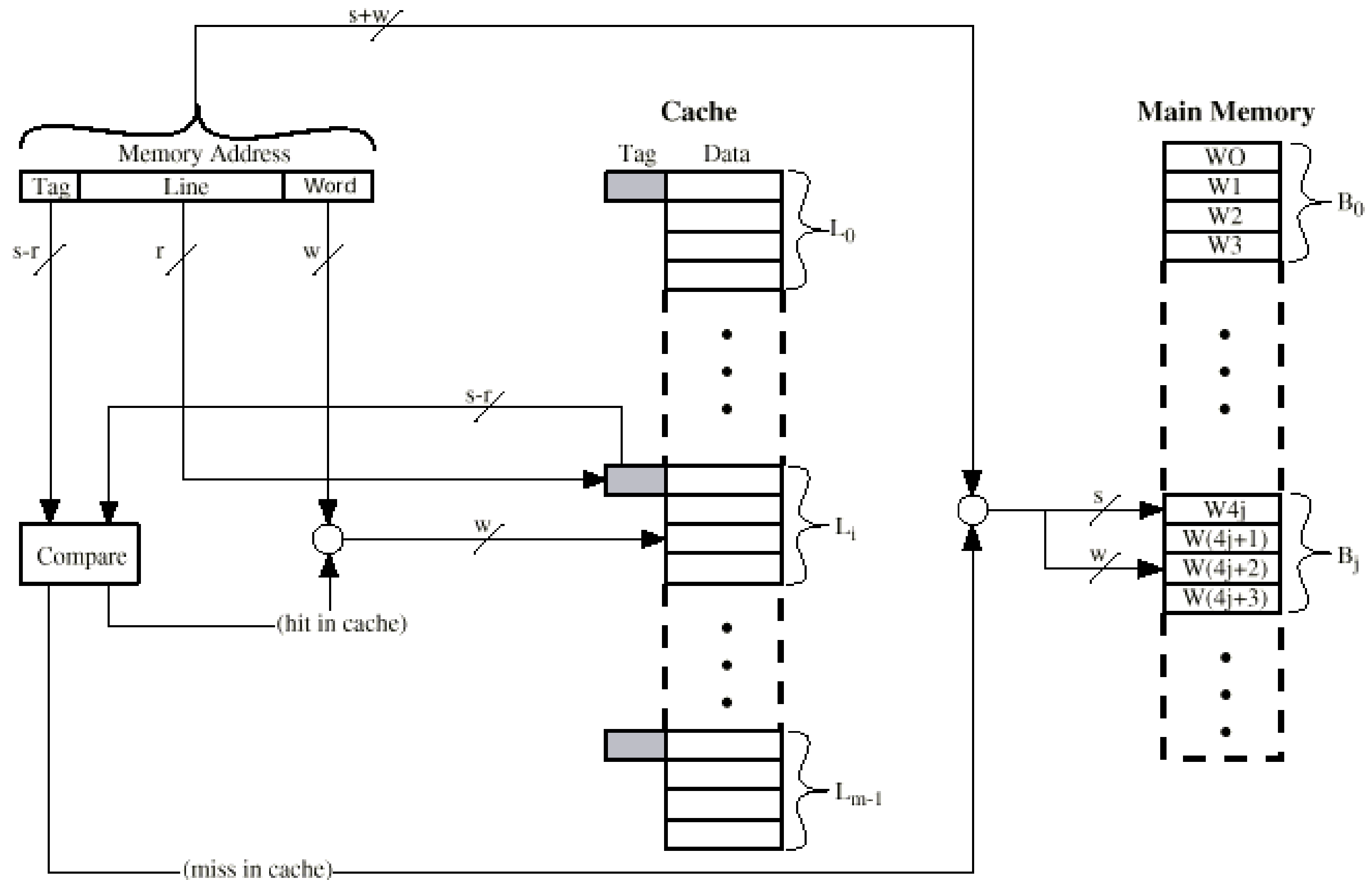
- 24 bit address
- 2 bit word identifier (4 byte block)
- 22 bit block identifier
  - 8 bit tag (=22-14)
  - 14 bit slot or line
- No two blocks in the same line have the same Tag field
- Check contents of cache by finding line and checking Tag

Direct Mapping
Cache Line Table

- Cache line Main Memory blocks held
- 0 0, m, 2m, 3m…2s-m
- 1 1,m+1, 2m+1…2s-m+1

- m-1 m-1, 2m-1,3m-1…2s-1

Direct Mapping Cache Organization



Dire

Tag

Line + Word

Data

00

0000
0004
FFF8
FFFC

13579246

16

0000
0004
339C
FFFC

77777777
11235813
FEDCBA98
12345678

FF

0000
0004
FFF8
FFFC

11223344
24682468

32 bits

16-MByte main memory

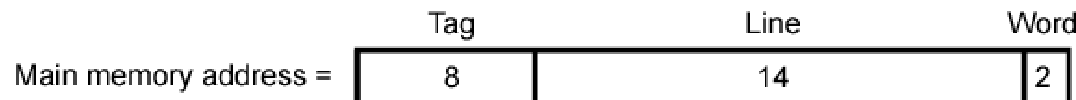
Tag | Data | Line Number

00 13579246 0000
16 11235813 0001
16 FEDCBA98 0CE7
FF 11223344 3FFE
16 12345678 3FFF

8 bits

32 bits

16-Kline cache



Direct Mapping Summary

- Address length = (s + w) bits
- Number of addressable units = 2s+w words or bytes
- Block size = line size = 2w words or bytes
- Number of blocks in main memory = 2s+ w/2w = 2s
- Number of lines in cache = m = 2r
- Size of tag = (s – r) bits

Direct Mapping pros & cons

- Simple
- Inexpensive
- Fixed location for given block
  - If a program accesses 2 blocks that map to the same line repeatedly, cache misses are very high

Set-Associative Mapping Cache

Index	Tag	Data	Tag	Data
000	0 1	3 4 5 0	0 2	5 6 7 0
777	0 2	6 7 1 0	0 0	2 3 4 0

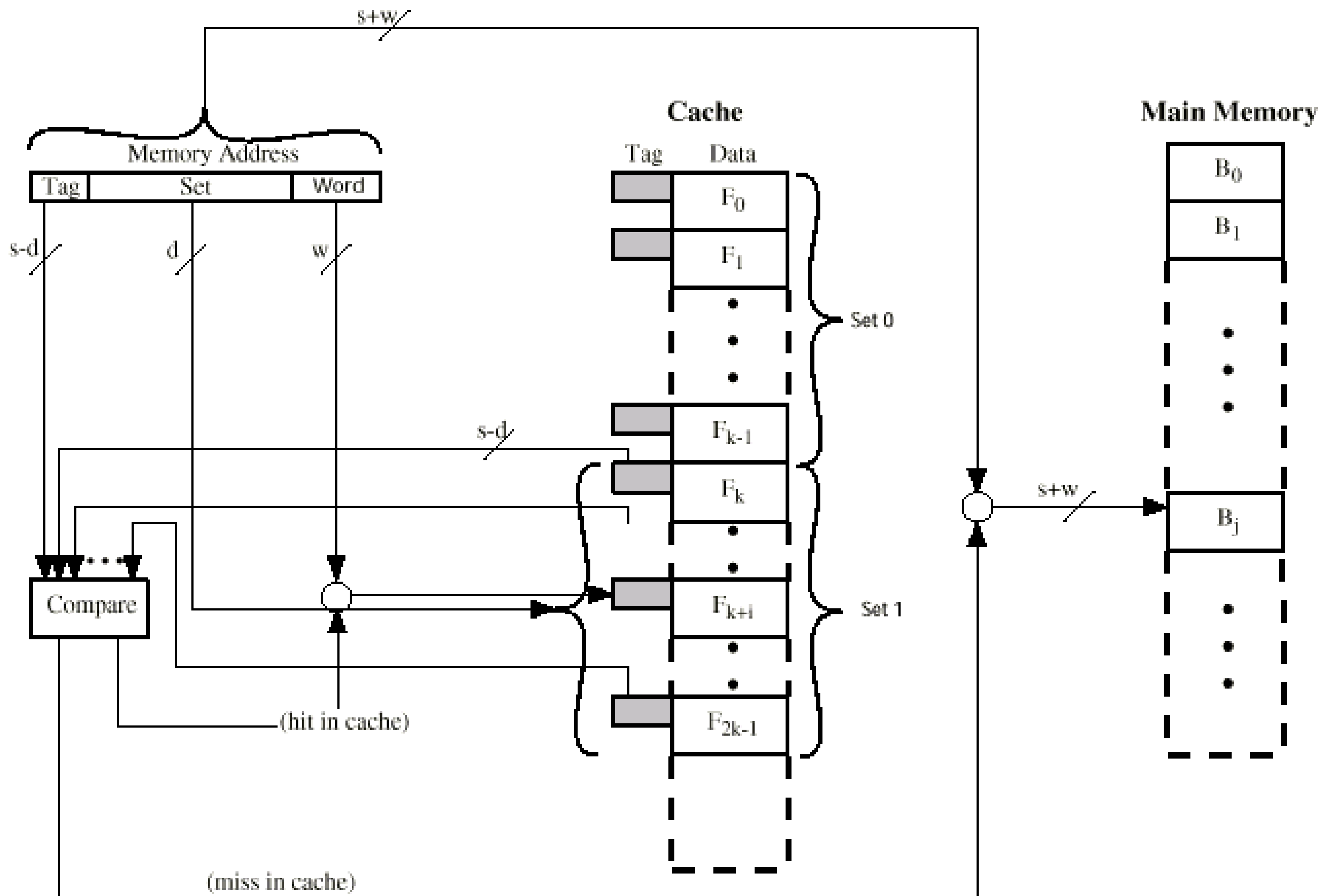
Set Associative Mapping

- Cache is divided into a number of sets
- Each set contains a number of lines
- A given block maps to any line in a given set
  - e.g. Block B can be in any line of set i
- e.g. 2 lines per set
  - 2 way associative mapping
  - A given block can be in one of 2 lines in only one set

Set Associative Mapping Example

- 13 bit set number
- Block number in main memory is modulo 2^{13}
- 000000, 00A000, 00B000, 00C000 … map to same set

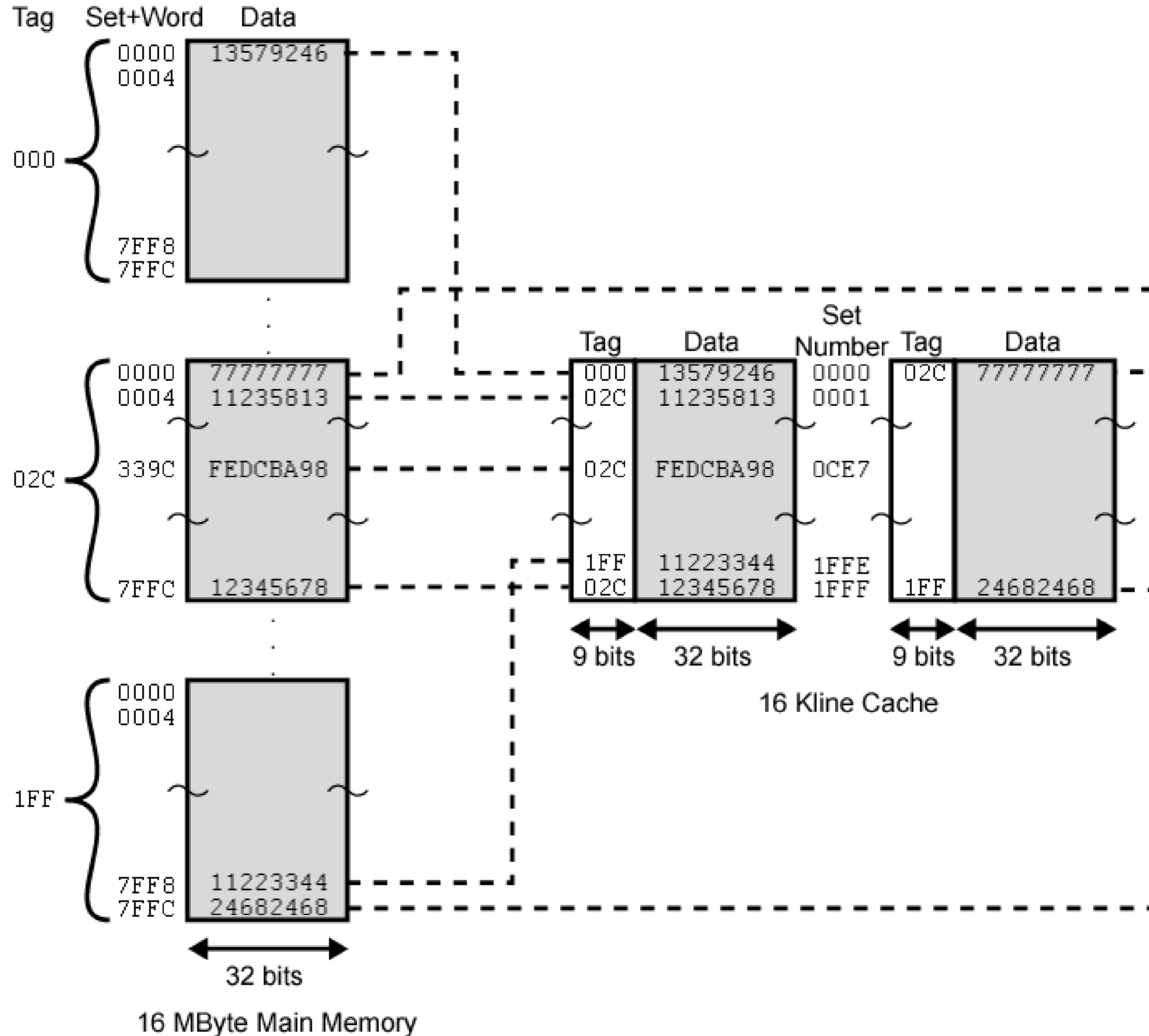
Two Way Set Associative Cache Organization



Set Associative Mapping Address Structure

Tag 9 bit	Set 13 bit	Word 2 bit

- Use set field to determine cache set to look in
- Compare tag field to see if we have a hit
- e.g
  - Address number   Tag   Data   Set
  - 1FF 7FFC   1FF   12345678   1FFF
  - 001 7FFC   001   11223344   1FFF



Main Memory Address =

Tag	Set	Word
9	13	2

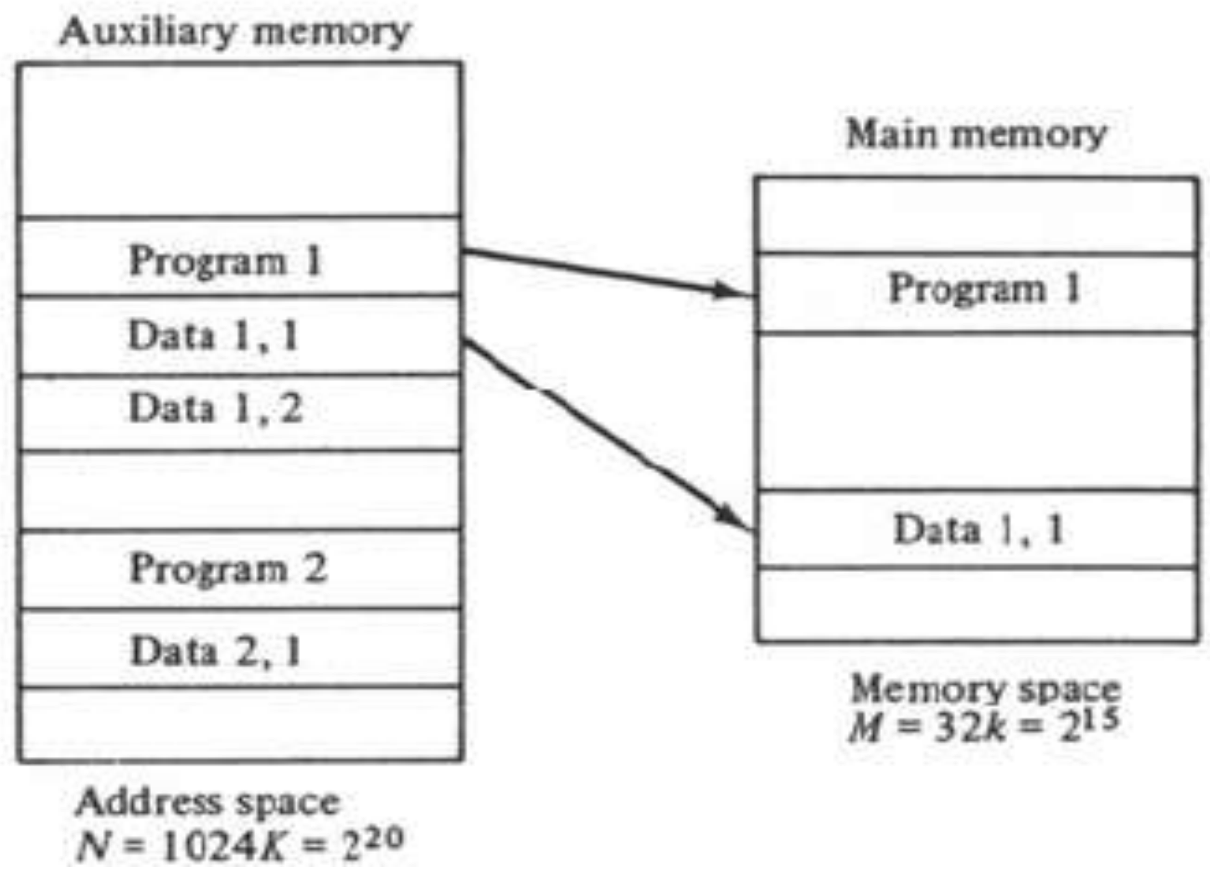
Set Associative Mapping Summary

- Address length = (s + w) bits
- Number of addressable units = 2s+w words or bytes
- Block size = line size = 2w words or bytes
- Number of blocks in main memory = 2d
- Number of lines in set = k
- Number of sets = v = 2d
- Number of lines in cache = kv = k * 2d
- Size of tag = (s – d) bits

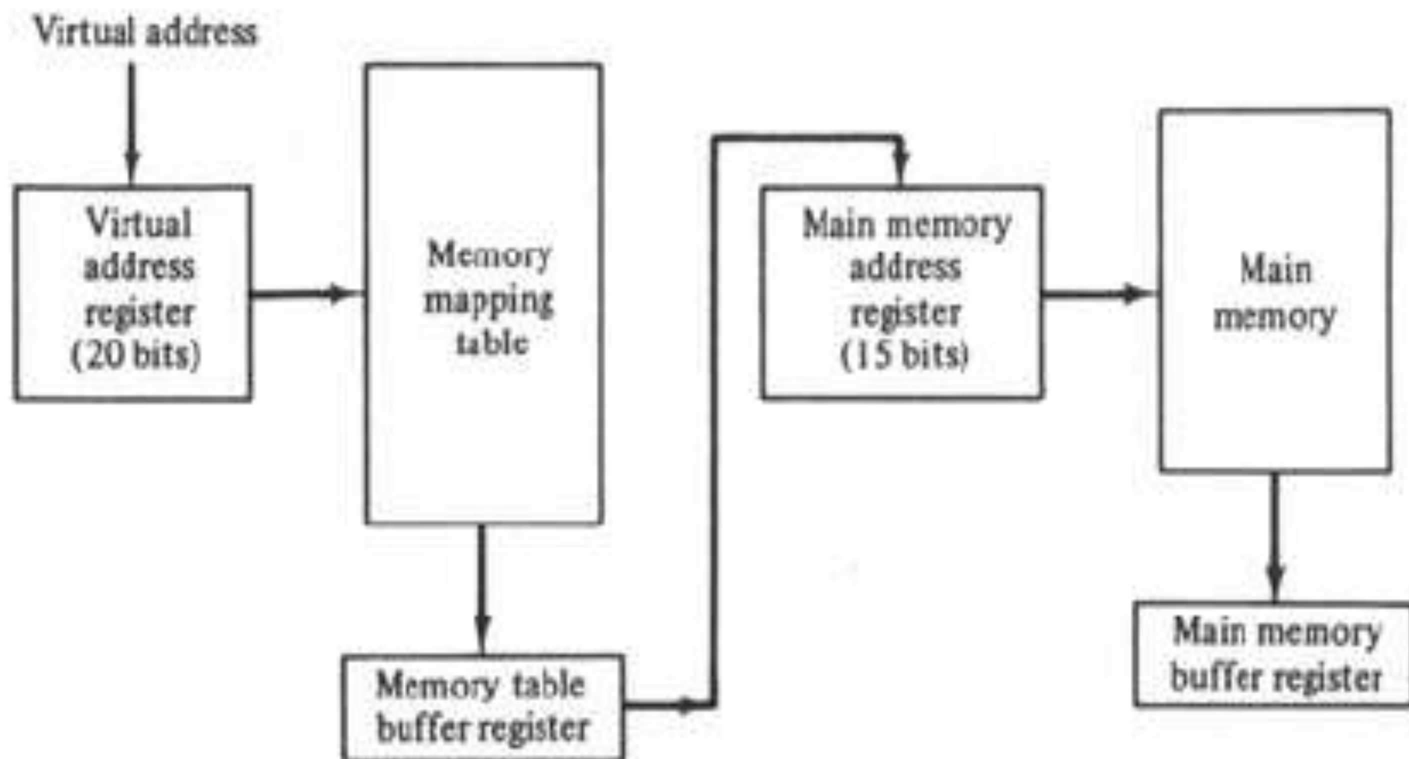
Writing Into Cache

- Write-through
  - Update cache and main memory in parallel
  - Needed when using DMA for I/O device communication
- Write-back
  - Update cache only
  - Update main memory only when updated word is flushed from cache

Address / Memory Space Reln



Mapping A Virtual Address



Address & Memory Spaces

Page 0
Page 1
Page 2
Page 3
Page 4
Page 5
Page 6
Page 7

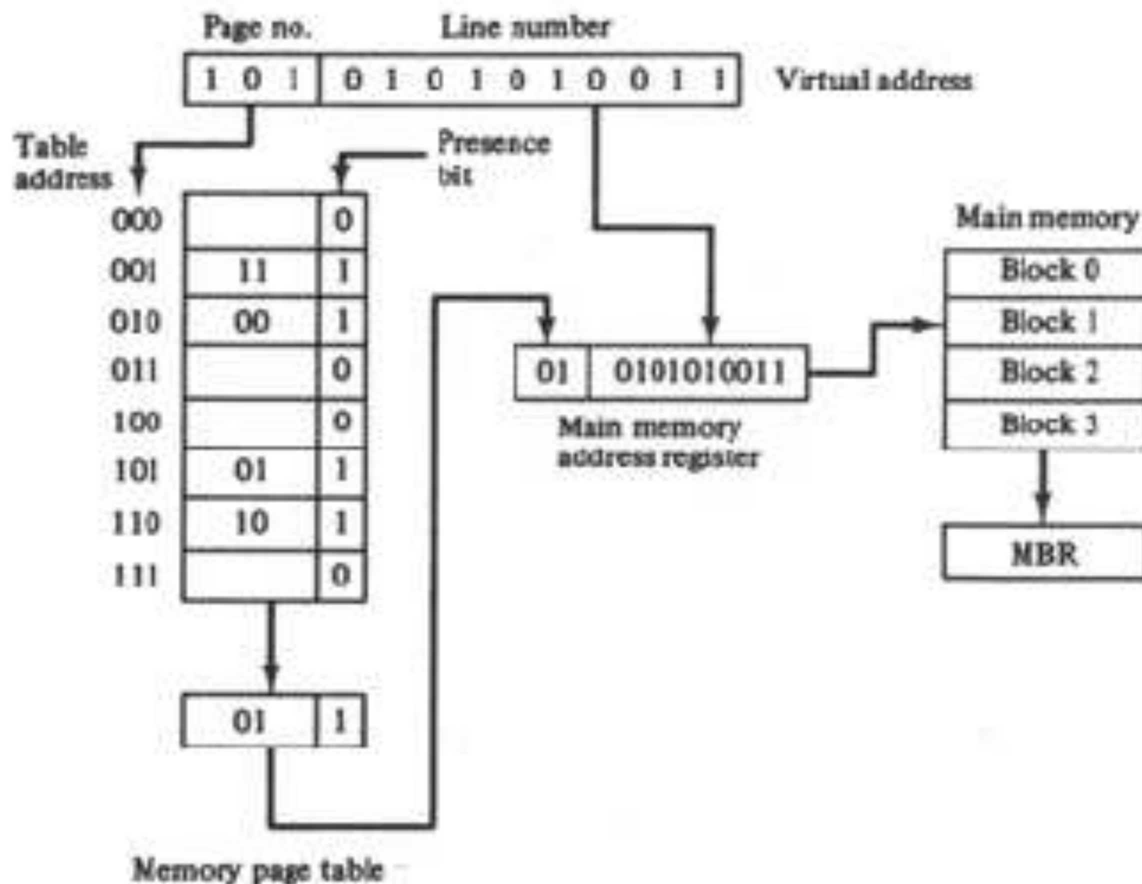
Address space
$N = 8K = 2^{13}$

Block 0
Block 1
Block 2
Block 3

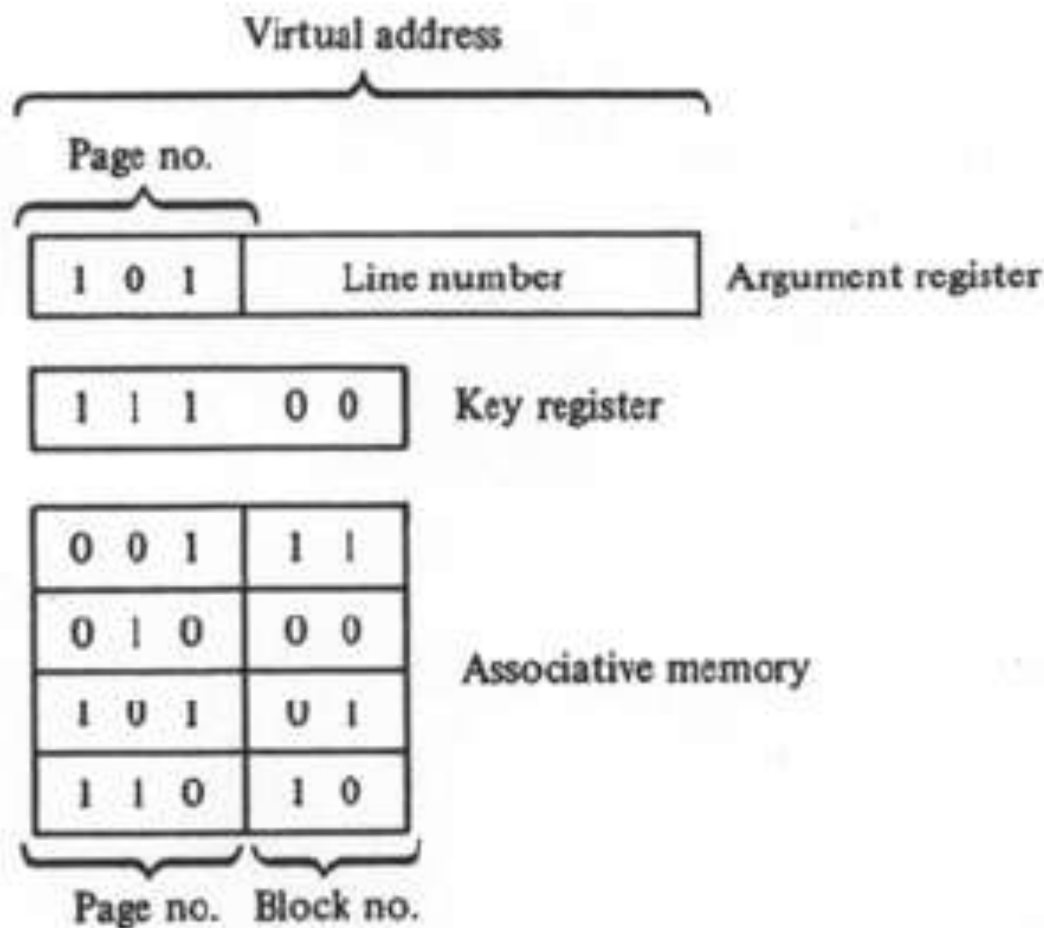
Memory space
$M = 4K = 2^{12}$

Block can also be called a "page frame"

Paging System Memory Table



Associative Memory Page Table



Page Frame Usage

- When a program's execution is initiated by the operating system, one or more pages are loaded into main memory
- Every time a page is referenced and it is not currently in main memory, a page fault is generated
  - Page is loaded from disk into an unused page frame
  - If all pages frames are in use, need method for freeing one up

Page Replacement Algorithms

- First-In, First-Out (FIFO)
  - Replace page that has been in memory the longest
- Least Recently Used (LRU)
  - Every time page is referenced, its aging counter is reset to zero
  - All aging counters are incremented by 1 every fixed time interval
  - Page with highest aging counter value is replaced

One Additional Detail

- Replacing (also called "stealing") a page that has not been updated is a simple process: the page frame is simply overwritten with the new page being read in from disk
- Stealing an updated page requires that it be written out to disk first before the page frame can be overwritten

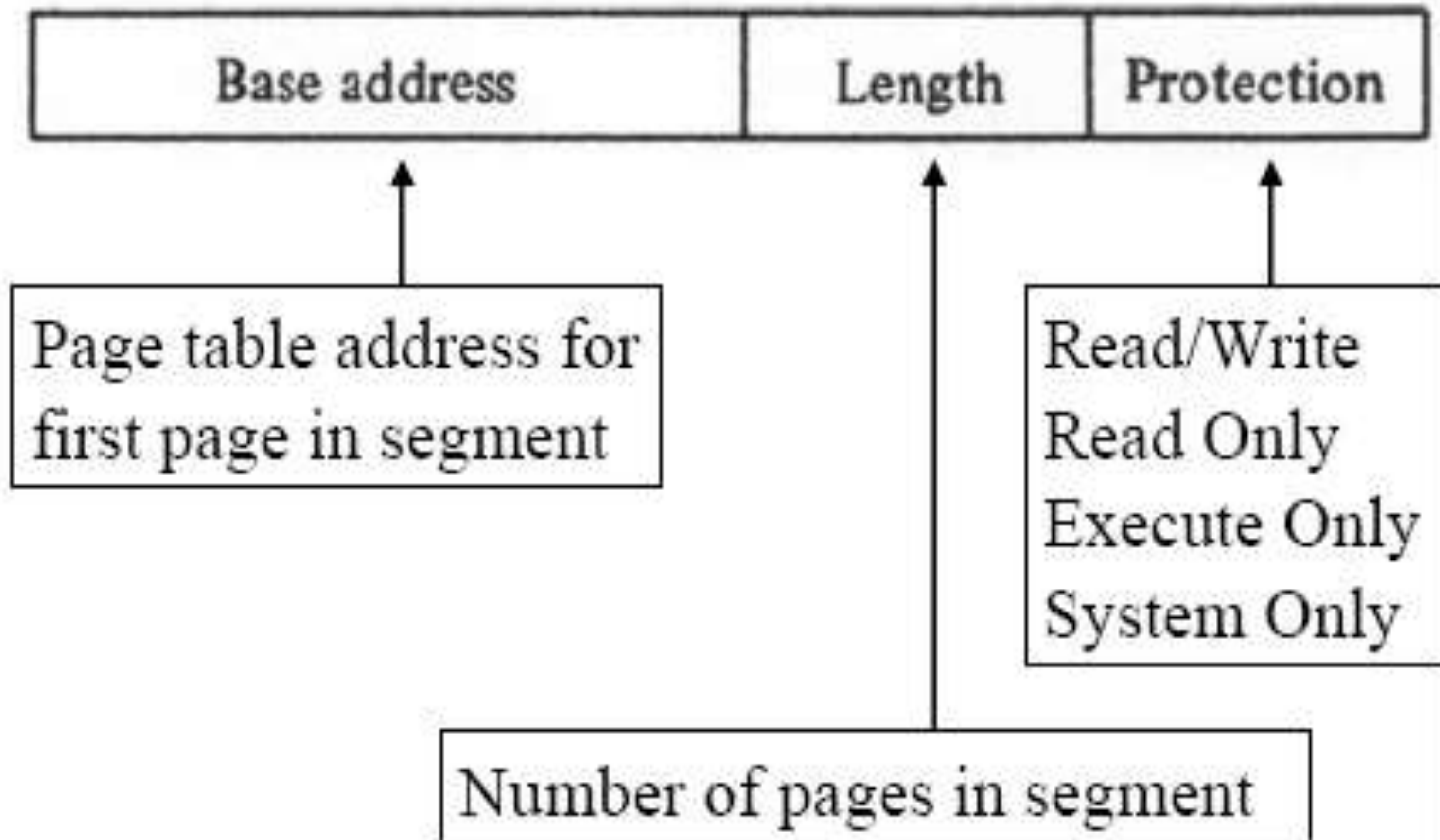
Memory Management Unit

- Supports dynamic storage relocation that maps logical-to-physical memory references
- Allows sharing of common programs loaded into memory
- Prevents unauthorized access or modification

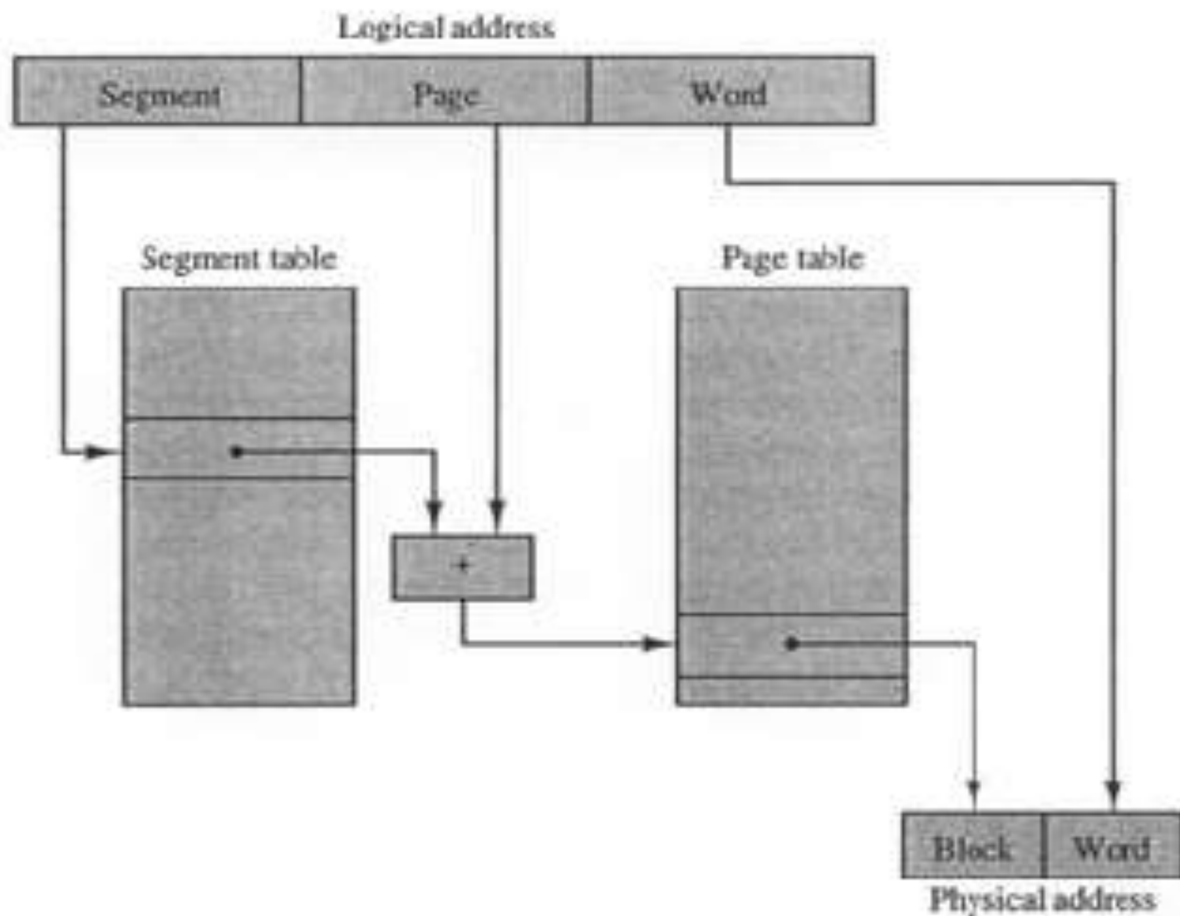
Segment

- Collection of one or more pages
- Defined by a programmer or the operating system
- A program is a collection of one or more segments
- Each segment has an associated segment descriptor

Segment Descriptor

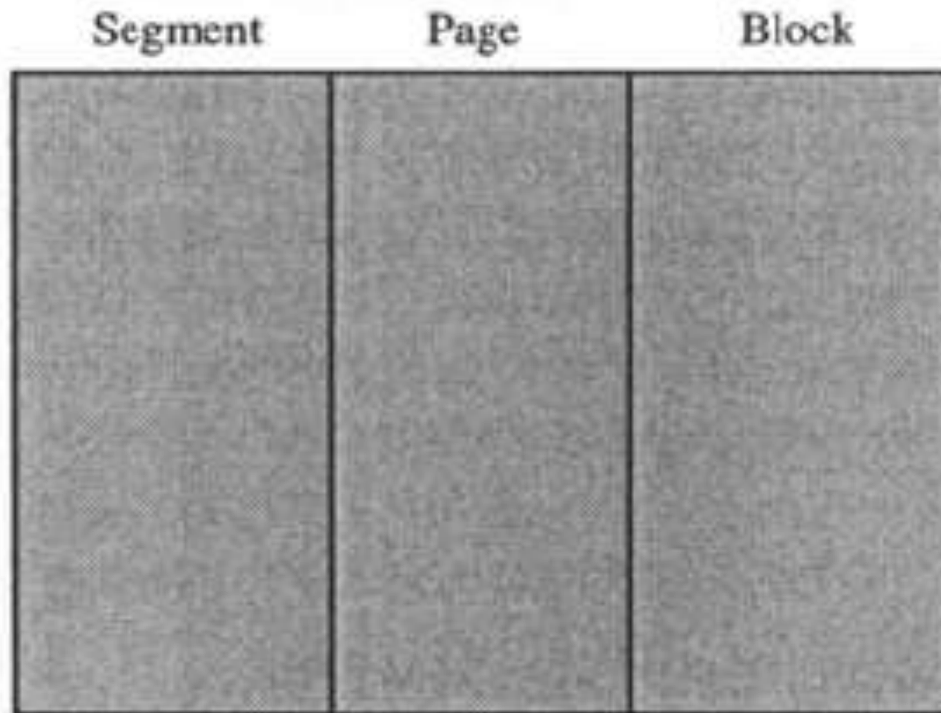


Logical-to-Physical Address



Translation Look-Aside Buffer

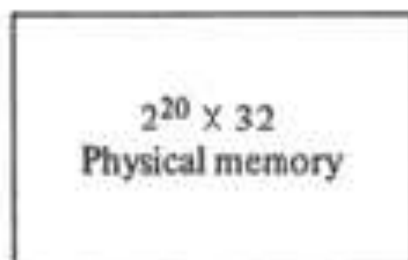
Argument register



associative memory

Logical & Physical Address Ex.

(a) Logical address format: 16 segments of 256 pages each, each page has 256 words



(b) Physical address format: 4096 blocks of 256 words each, each word has 32 bits

Memory Address Assignment Ex.

Hexadecimal address	Page number
60000	Page 0
60100	Page 1
60200	Page 2
60300	Page 3
60400 604FF	Page 4

(a) Logical address assignment

Segment	Page	Block
6	00	012
6	01	000
6	02	019
6	03	053
6	04	A61

(b) Segment-page versus memory block assignment

Segment & Page Table Mapping

Logical address (in haxadecimal)

6	02	7E

Segment table

0	
6	35
F	A3

Page table

00	
35	012
36	[illegible]
37	019
38	053
39	A61
A3	012

Physical memory

00000	Block 0
000FF	
01200	Block 12
012FF	
01900	
0197E	32-bit word
019FF	

Associative Memory TLB

Segment	Page	Block
6	02	019
6	04	A61

Multiprocessors

Overview

- Multiple instruction, multiple data (MIMD)
- One operating system controls the CPUs
- Multiple jobs can execute in parallel
- Single job can be partitioned into multiple parallel tasks
- Computer can continue to run when one CPU fails

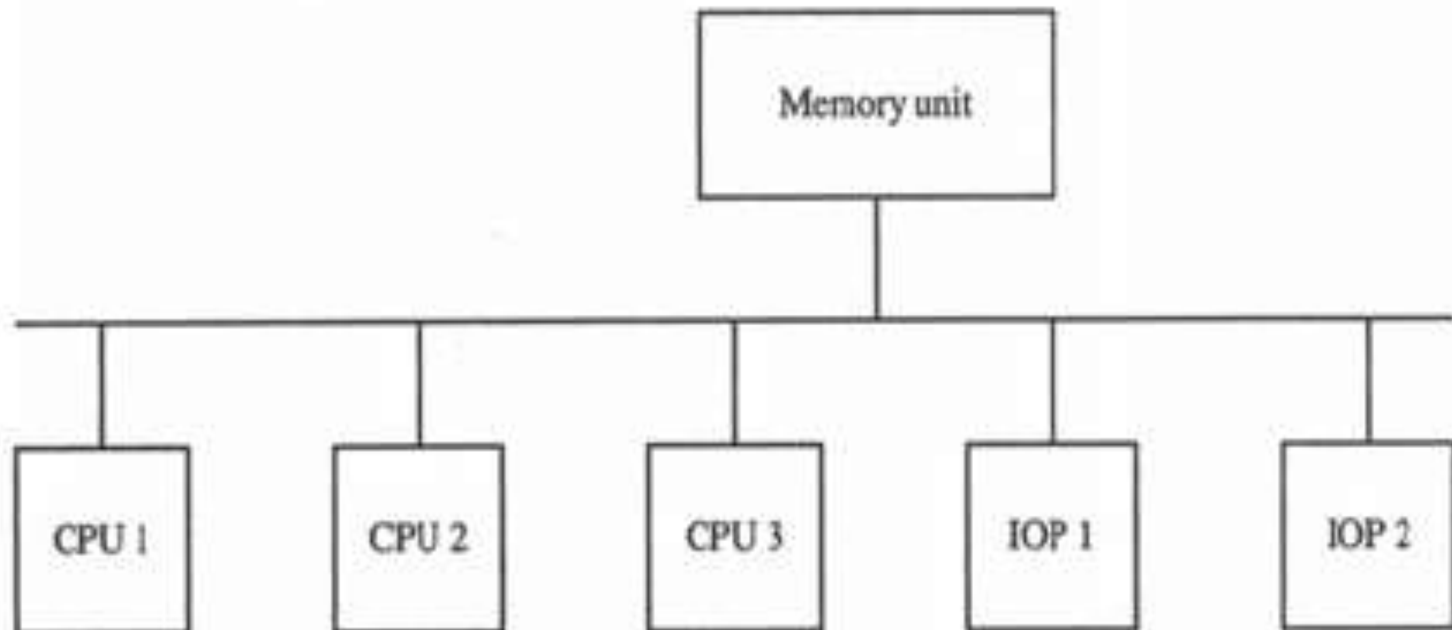
Classification

- Tightly coupled
  - Common, shared main memory
  - Each CPU can still have its own cache memory
  - CPUs share data by writing it into main memory
- Loosely coupled
  - Each CPU has its own main memory
  - CPUs share data by passing messages

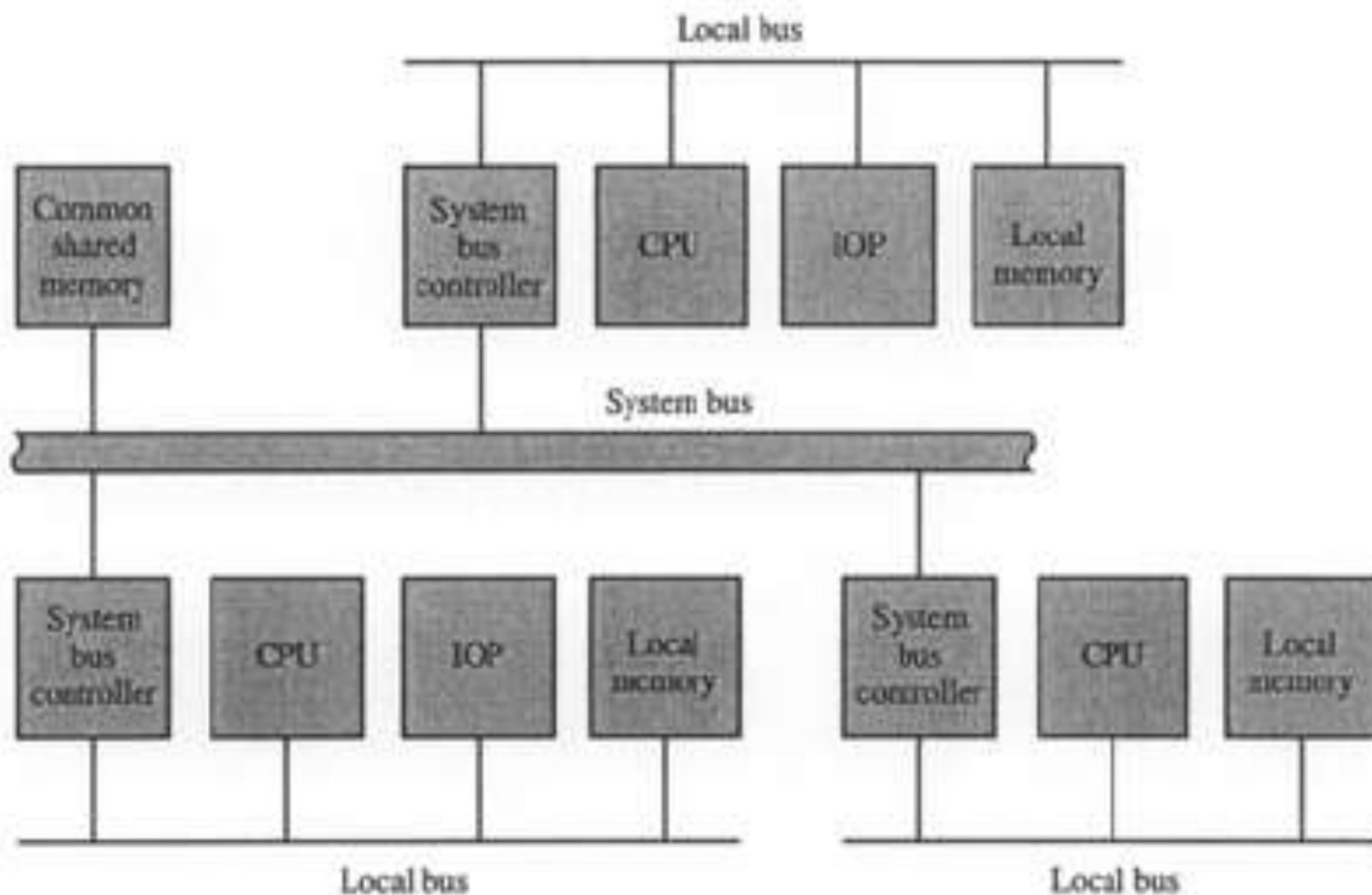
Interconnection Structures

- Time-shared common bus
- Multiport memory
- Crossbar switch
- Multistage switching network
- Hypercube system

Time-Shared Common Bus



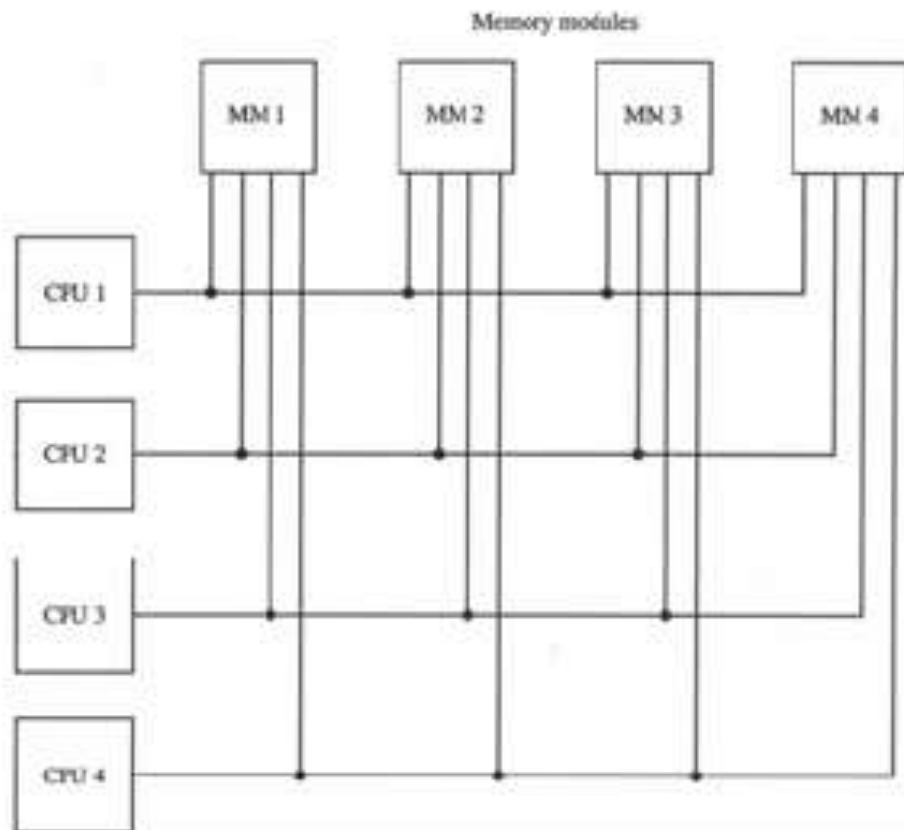
Common Bus With Local Busses



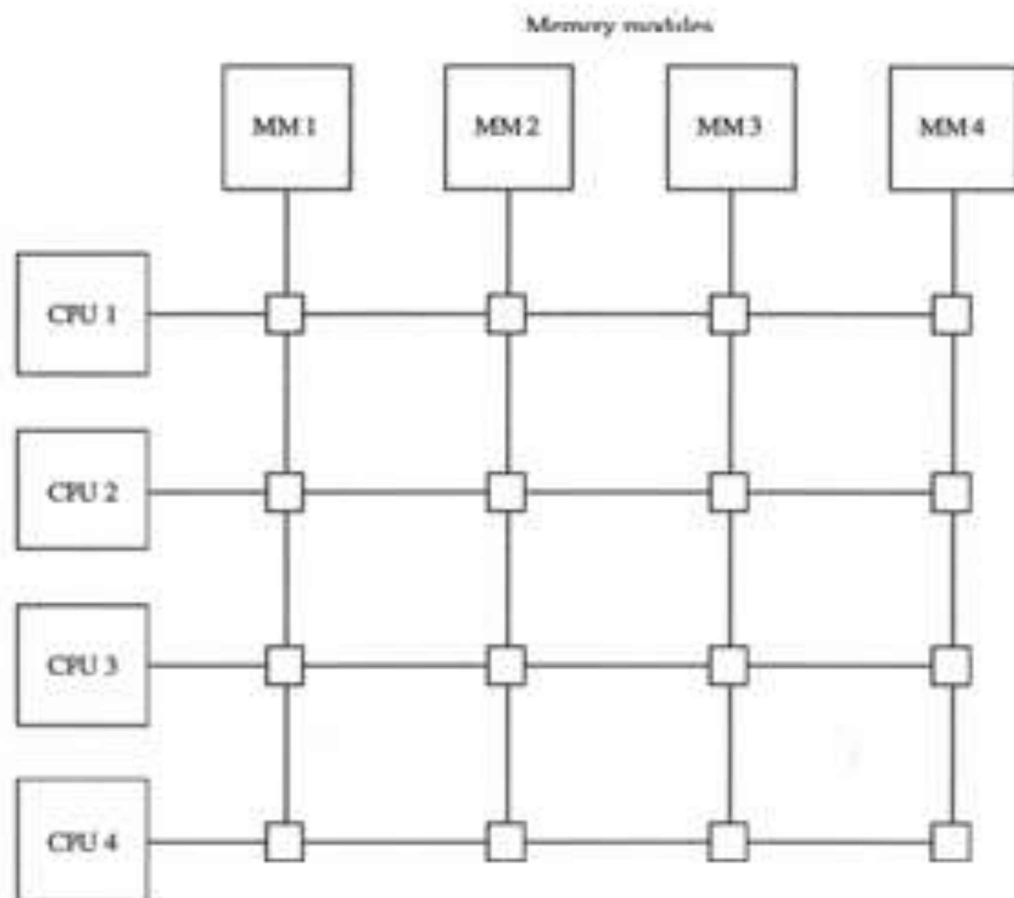
Multiport Memory

- Each processor bus is connected to each memory module
- Memory module has as many ports as there are processor busses connected to it
- Memory module access conflicts resolved by assigning a fixed priority to each of its ports

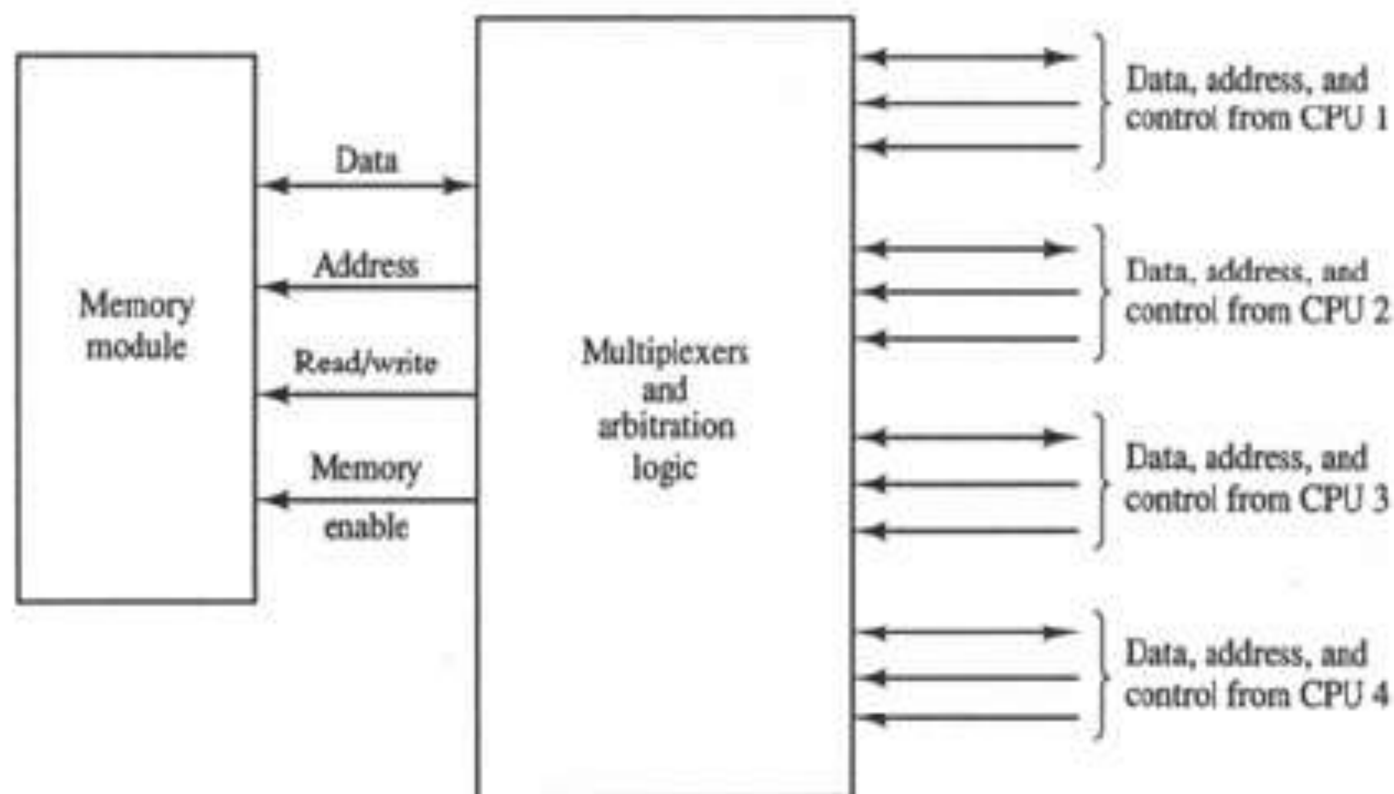
Multiport Memory Diagram



Crossbar Switch

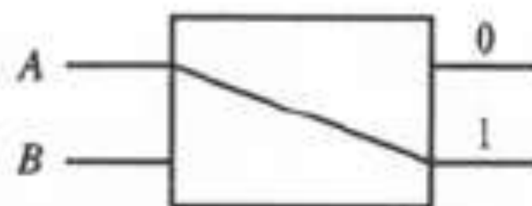


Crossbar Switch Block Diagram



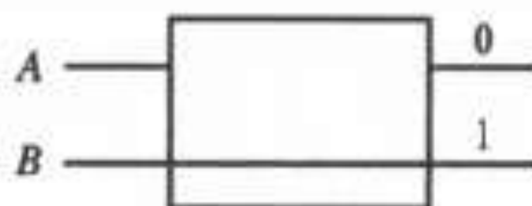
2x2 Interchange Switch Operation

A connected to 0



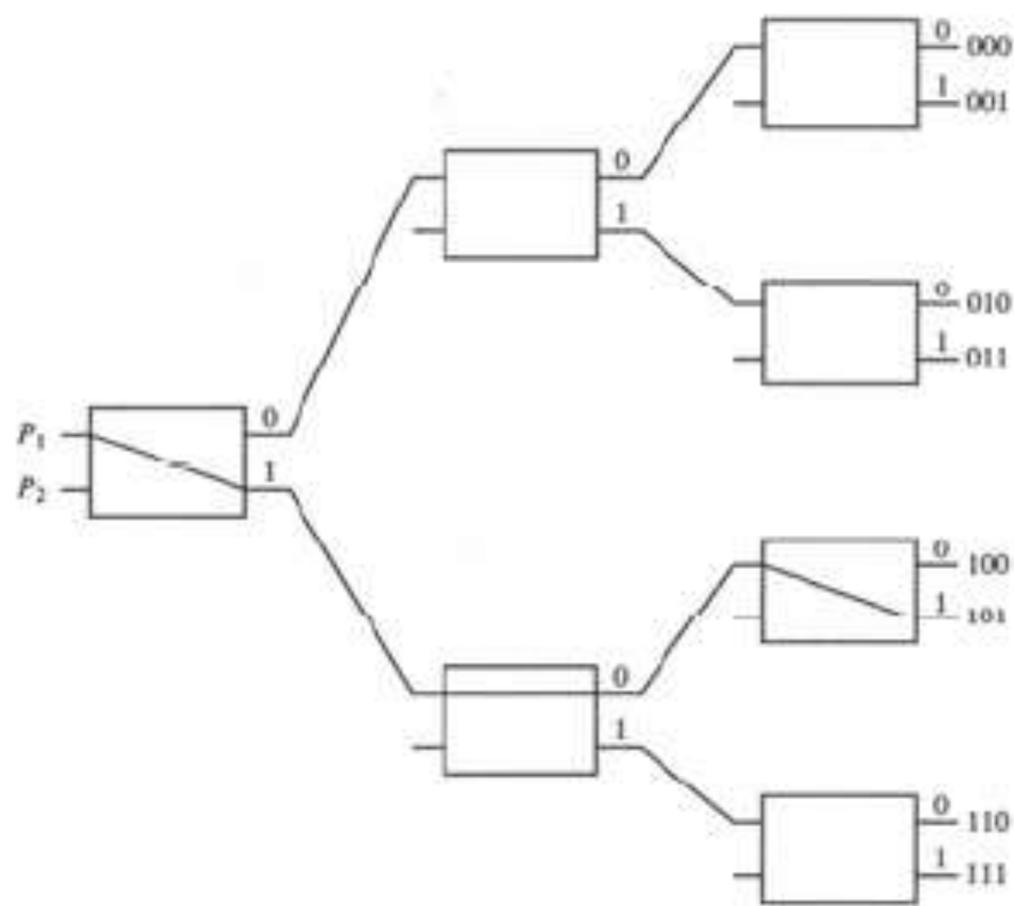
A connected to 1

B connected to 0

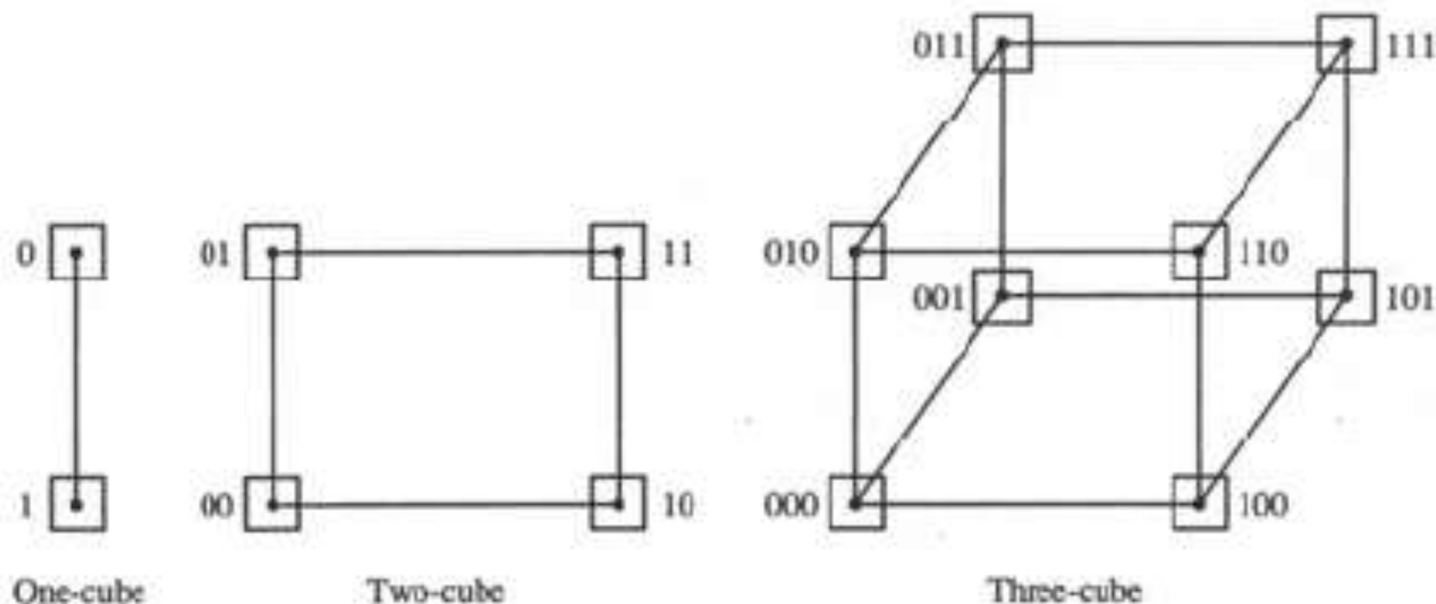


B connected to 1

Binary Tree With 2x2 Switches



Hypercube Structures

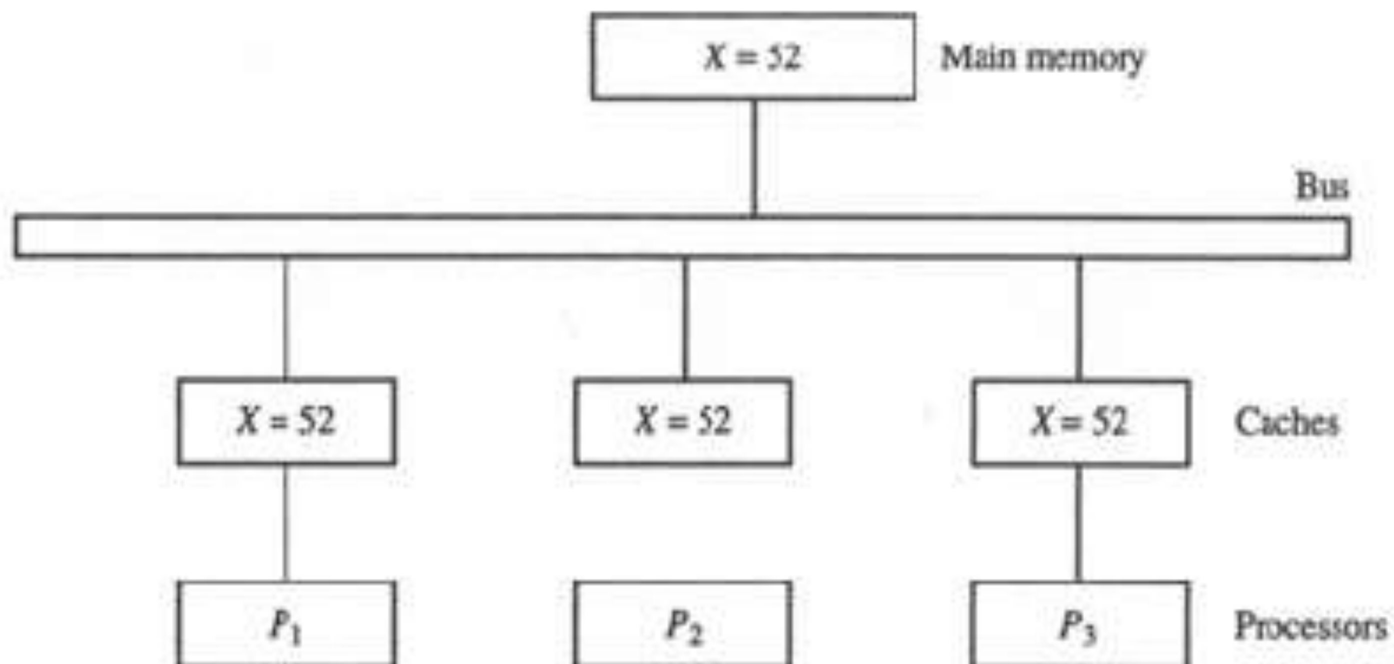


Each node has a CPU, local memory, and I/O interface
Nodes communicate by passing messages

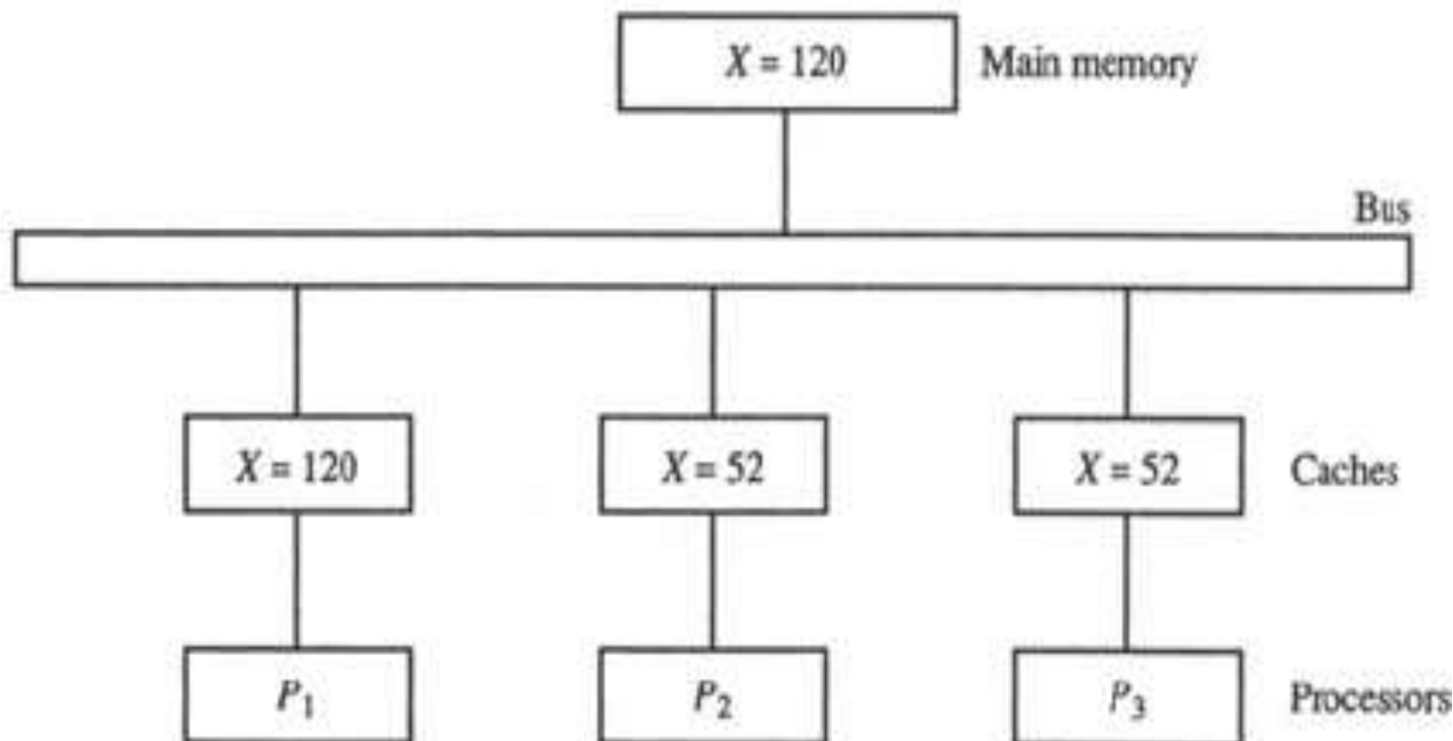
Cache Coherence

- Value returned on a load instruction is always the value given by the latest store instruction with the same address

Load X

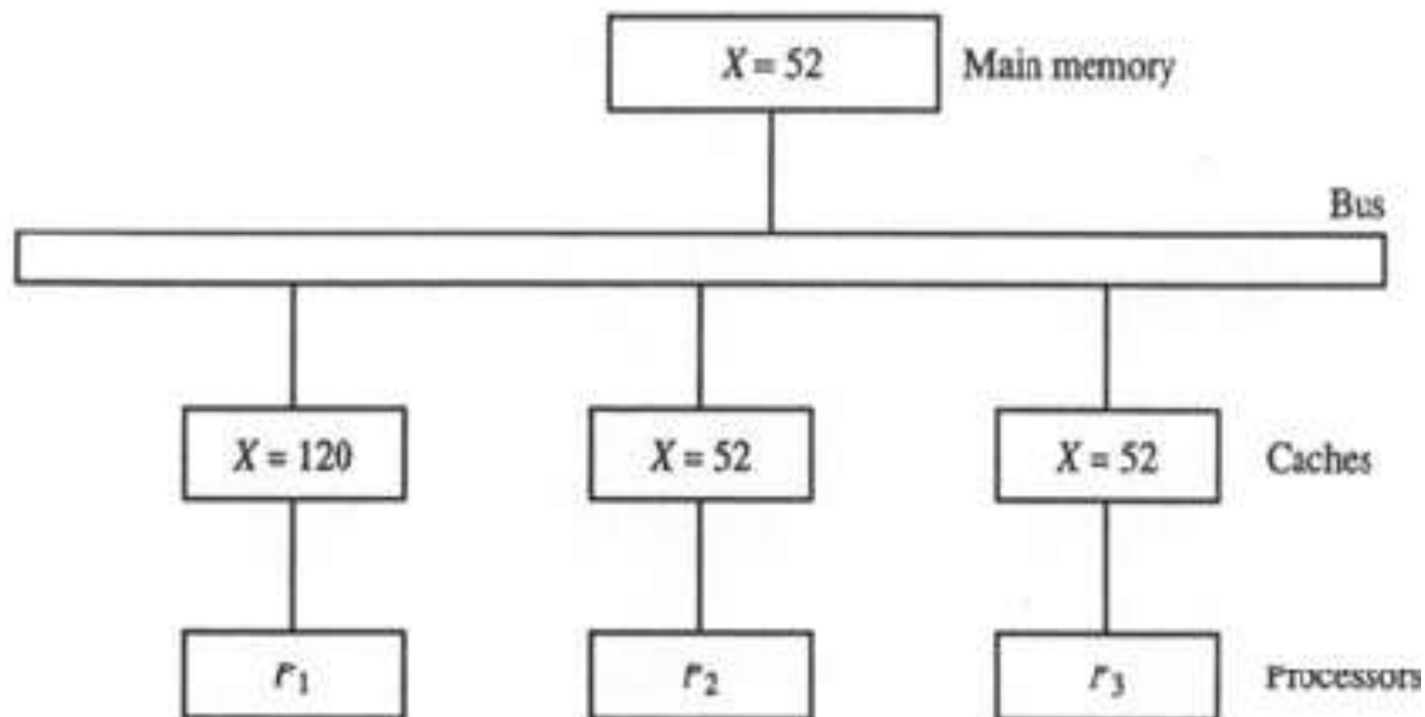


Write-Through Cache Policy



(a) With write-through cache policy

Write-Back Cache Policy



(b) With write-back cache policy

Cache Coherence Solutions

- Disallow private caches (!)
- Restrict cache to non-shared and read-only data
- Memory block status stored in global table
  - Each block tagged as RO or RW
  - All caches can have RO data
  - Only one cache can have RW data
- Each cache has a snoopy controller
  - Watches bus for write operations
  - Invalidates cache entry if address appears

Interprocessor Arbitration

Computer system contains a number of buses at various levels to facilitate the transfer of information between components. The CPU Contains a number of Internal buses for transferring information between processor register and ALU. A memory bus consists of lines for transferring data, address, and read/write information. An I/O bus is used to transfer information to and from I/O devices.

System Bus:-

A bus that connects major components in a multi-processor system such as CPUs, IOPs, memory, is called s system bus.

System Bus

A typical system bus consists of approximately 100 signal lines. These lines are divided into three functional groups: data, address, and control.

In addition, there are power distribution lines that supply power to the components.

For example, the IEEE standard 796 multibus system has 16 data lines, 24 address lines, 26 control lines, and 20 power lines, for total of 86 lines.

Data lines provide a path for the transfer of data between processors and common memory.

Address lines are used to identify a memory address or any other source or destination , such as input or output ports.

Synchronous Bus & Asynchronous Bus

- **Synchronous Bus**

  In a synchronous bus each data item is transferred during a time slice known in advance to both source and destination units.

  Synchronization is achieved by driving both units from a common clock source.

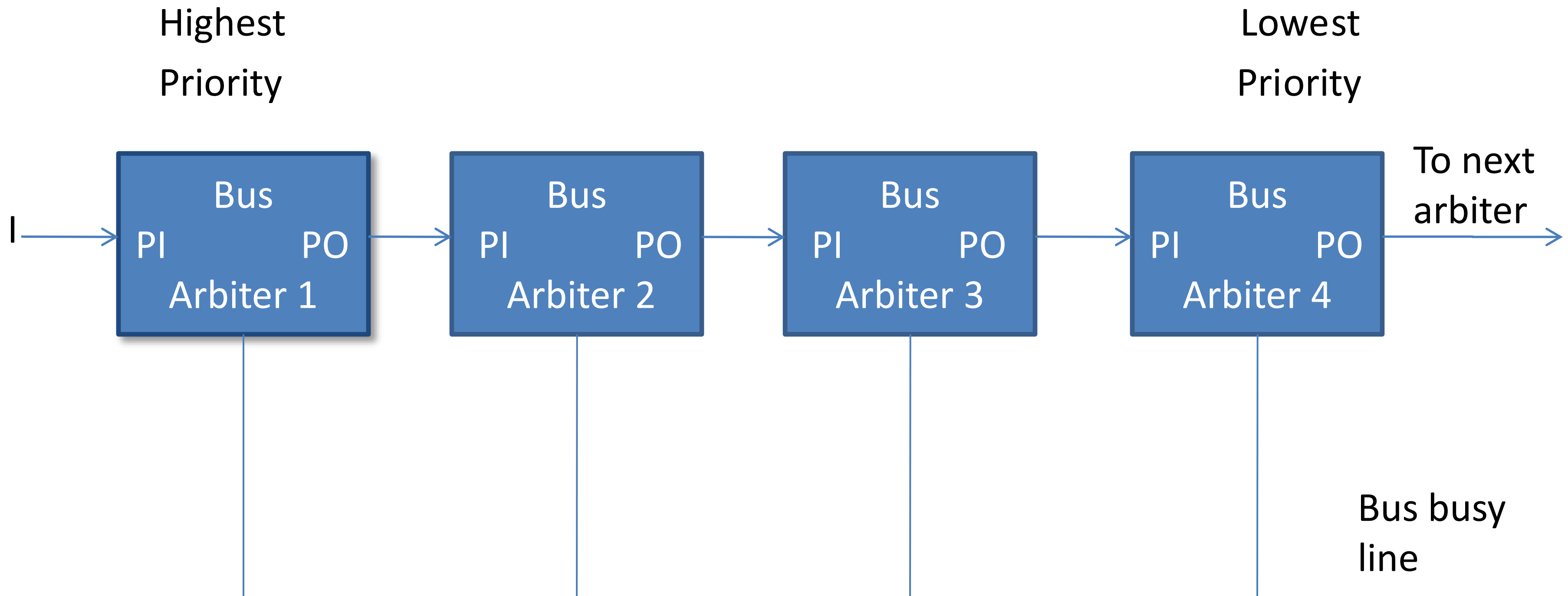
- **Asynchronous Bus**

  In an Asynchronous bus each data item being transferred is accompanied by handshaking control signal to indicate when the data are transferred from the source and achieved by the destination.

- Control lines provide signals for controlling the information transfer between units . The signals indicate the validity of data and address

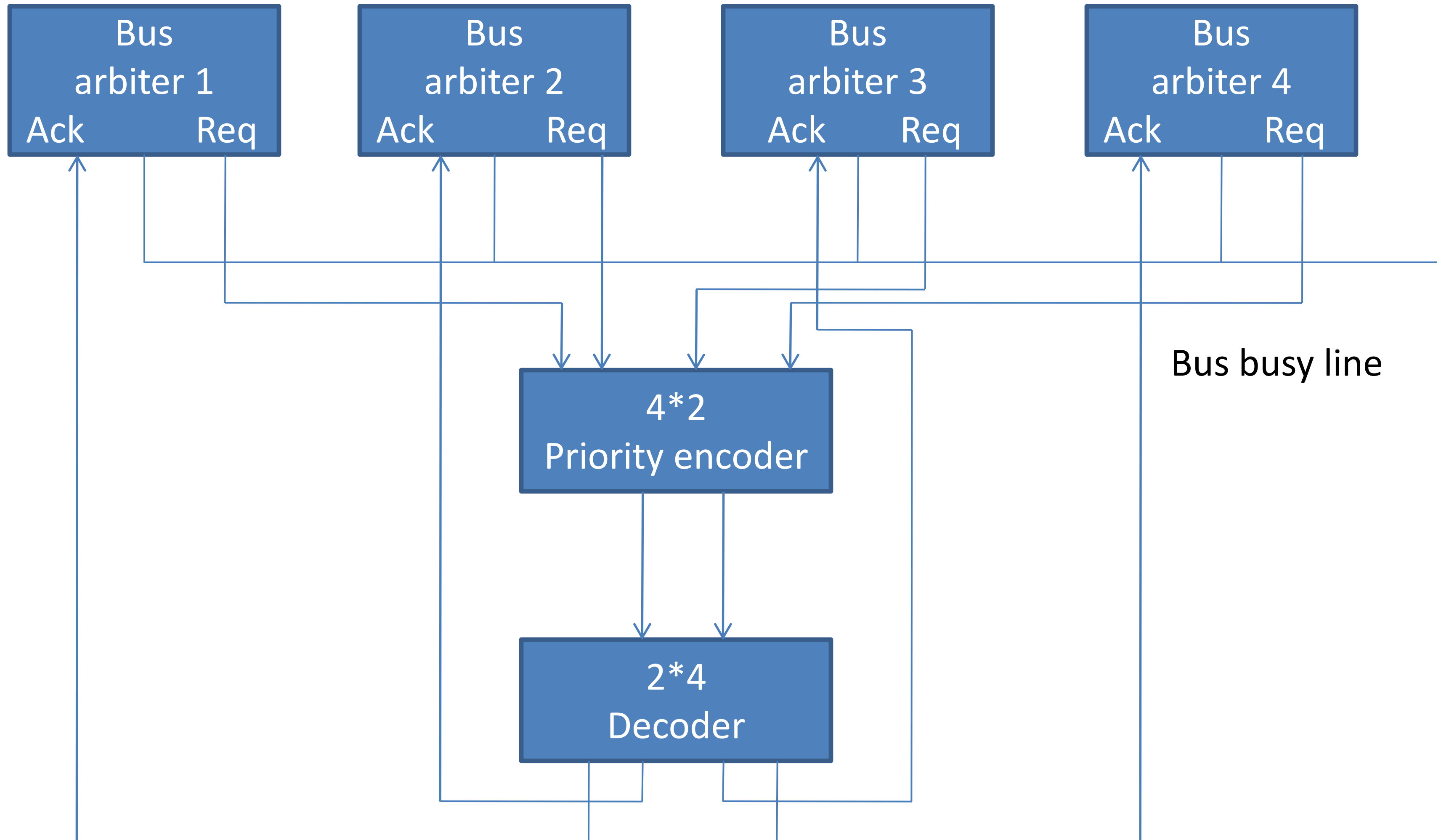
  Information. Command signals specify operation to be performed. Typical control lines include transfer signals such as memory read and write , acknowledge of a transfer , Interrupt requests, bus control signal such as bus request and bus grant , and signals for arbitration procedures.

Serial Arbitration Procedure



Serial (Daisy Chain) arbitration

Parallel Arbitration



Interprocessor Communication

- The various processors in a multiprocessor system must be provided with a facility for communicating with each other. A communication path can be established through common input output channels. In a shared memory multiprocessor system, the most common procedure is to set aside a portion of a memory that is accessible to all processors.

  In addition to shared memory, a multiprocessor system may have other shared resources. For example, a magnetic disk storage unit connected to an IOP may be available to all CPUs. This provides a facility for sharing of system programs stored in the disk.

- There are three organizations that have been used in the design of operating system for multiprocessors:

  - Master-slave configuration
  - Separate operating system
  - Distribute operating system

Master slave mode

- In a master slave mode, one processor, designated the master, always executes the operating system functions. If a slave processor needs an operating system service, it must request it by interrupting the master and waiting until the current program can be interrupted.

Separate operating system organization

- In the separate operating system organization, each processor can execute the operating system routines it needs. This organization is more suitable for loosely coupled systems where every processor may have its own copy of the entire operating system.

Distributed operating system organization

- In the distributed operating system organization, the operating system routines are distributed among the available processors. However particular operating system function is assigned to only one processor at a time. This type of operating system is also referred to as a floating operating system since the routines float from one processor to another and the execution of the routines may be assigned to defferent processors at different times.

Interprocessor synchronization

- The instruction set of a multiprocessor contains basic instruction that are used to implement communication and synchronization between cooperating processes. Communication refers to the exchange of data between different processes.
- Synchronization refers to the special case where the data used to communicate between processors is control information. Synchronization is needed to enforce the correct sequence of processes and to ensure mutually exclusive access to shared writable data.

Unit - 02

Principles of Computer design