UNIT-2

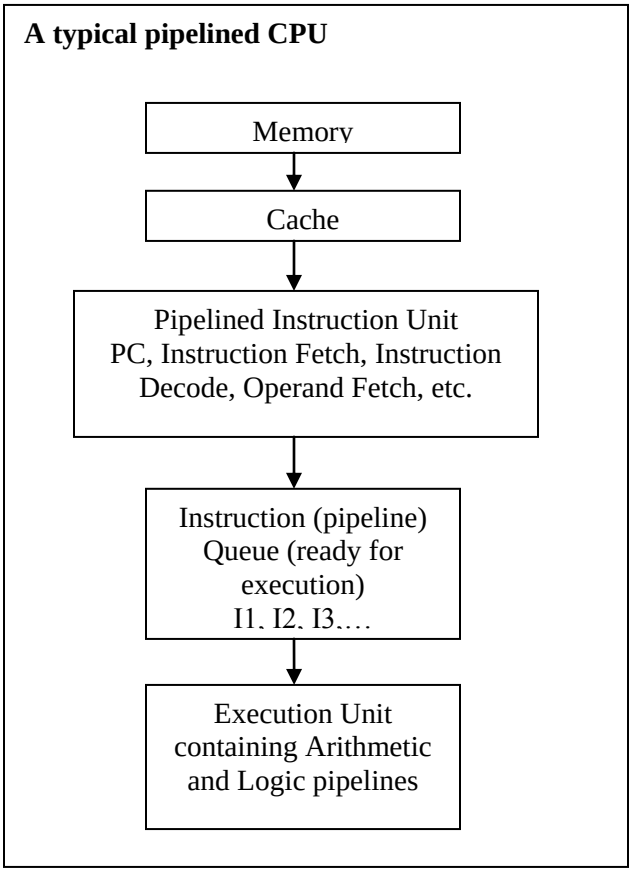
PIPELINE & VECTOR PROCESSING

VECTOR PROCESSING

- A vector instruction contains large array of operands and same set of operations are performed using these operands.
- They need pipelined systems to execute.

PIPELINING CONCEPT

- To achieve pipelining we must subdivide the input process (major task) in to number of subtasks that can be fed to the pipeline one after another without waiting for the results.
- The subtasks are given to dedicated hardware.
- As previous tasks move forward in the system the subsequent tasks enter the system.
- A typical pipelined CPU is as follows:



- So, the Major tasks are getting executed in parallel form, but the Subtasks within them get executed in serial form.
- The major concern in pipelined systems is collision avoidance.

PIPELINED PROCESSING

Pipelined computers perform overlapped computations and are said to implement Temporal Parallelism. Usually there are four major steps in program execution, first is Instruction Fetch (I.F.) from the main memory, second is Instruction Decode (I.D.) to identify the type of operation that needs to be performed, third is Operand Fetch (O.F.) as needed by the instruction and fourth is the Execution (E.X.) of the identified operation.

In non-pipelined computers these steps are performed in a sequential manner for every instruction. In a pipelined system the instructions are collected according to the capacity of the pipelines and this set is sent to the first stage i.e I.F. then the second set of instructions are collected when the previous set reaches the second stage i.e I.D. In the mean time the second set of instructions are sent for I.F., in this manner upper set of instructions finish at once and so on.

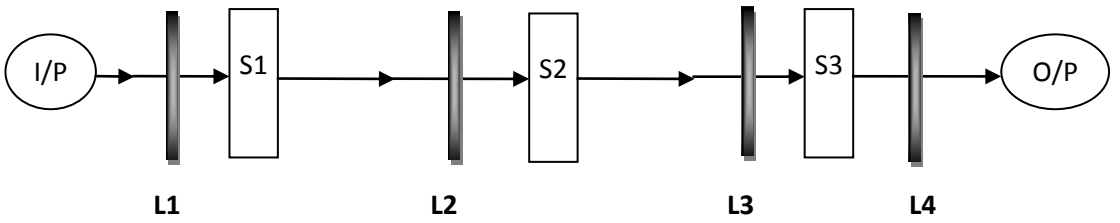
Say for example we have a system than can handle instruction pipeline that has a capacity of 5 instructions, from I1 to I5, diagrammatically the pipelining can be represented as:

Pipelined System

	OUTPUT								
E.X.				I1	I2	I3	I4	I5	
O.F			I1	I2	I3	I4	I5		
I.D.		I1	I2	I3	I4	I5			
I.F.	I1	I2	I3	I4	I5				
	INSTRUCTIONS LOADED IN MEMORY								

- The operations of all stages are synchronized under a common clock control.
- Interface latches are used to hold the intermediate results between adjacent segments.
- These systems perform optimum when same type of operations are preformed through out the pipeline e.g. addition. Whenever there is change in type of instruction i.e from addition to multiplication then the pipeline must be drained and reconfigured.
- They are more appropriate for Vector processing where same type of operation is repeated for an array of operands.
- Example: AP-120B, FPS-164, etc.

Pipelined System (Linear Pipelined System)



L1 to L4 are memory Latches to store and forward intermediate results
S1 to S3 are stages in pipeline which usually contain combinational circuits.

DEDICATED PIPELINES

These pipelines are created for performing a fixed function and the data fed to these systems need to be in specific format. The example of such dedicated pipeline can be a system that performs matrix multiplication. Even though dedicated pipelines perform a single dedicated function but there performance is always better than non-pipelined systems for the same function.

Let's consider the example of matrix multiplication:

Let there be two matrices A and B of size 3x3 and we try to build a pipelined system that can perform the multiplication of A and B and generate a third resultant matrix C.

So, the formula that can be used for this is

$$C_{ij} = \sum_{k=1}^{3} (A_{ik} * B_{kj})$$

where,

i = 1 to 3 , j = 1 to 3 , k = 1 to 3.

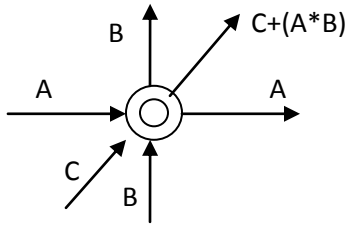
Algorithm:

```
For i=1 to 3 do
{
        For j=1 to 3 do
        {
                For k=1 to 3 do
                {
                        Cij = Cij + ( Aik * Bkj )
                }
        }
}
```

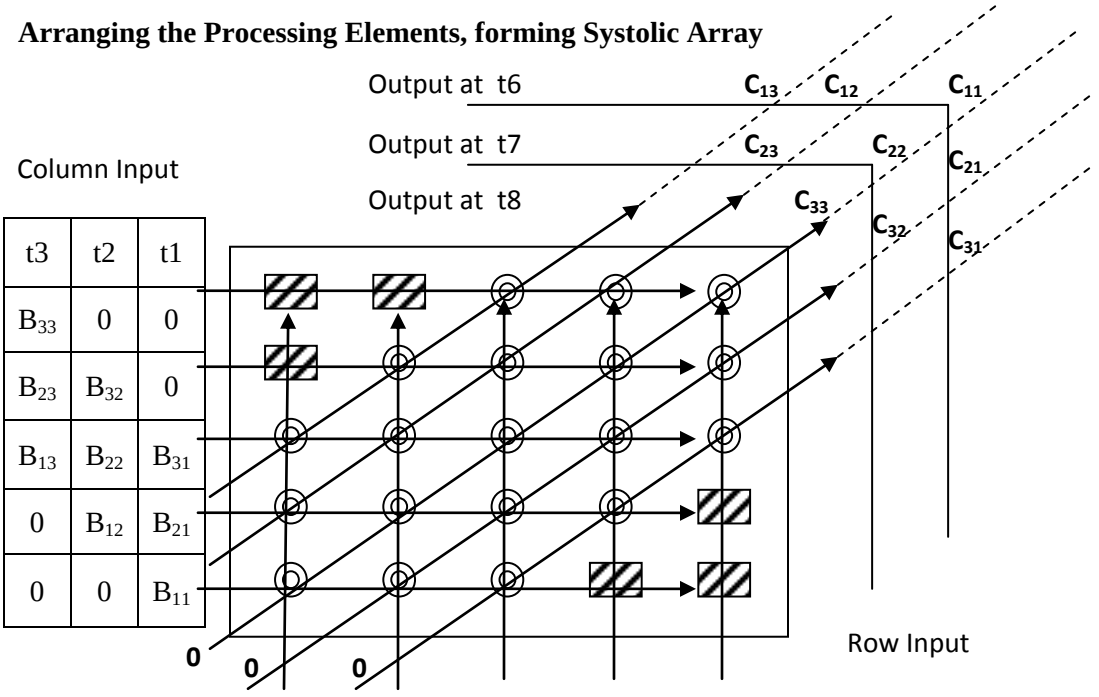
The complexity of this program in non-pipelined system would be 3*3*3 = 27 units of time.

Now, we decide a processing element that can be used in our system:

Processing Element



Arranging the Processing Elements, forming Systolic Array



A_{11}	A_{12}	A_{13}	0	0	input at time t1
0	A_{21}	A_{22}	A_{23}	0	input at time t2
0	0	A_{31}	A_{32}	A_{33}	input at time t3

From the above pipelined arrangement the output that we get is

- At time t6: C_{12}, C_{12}, C_{11}, C_{21} and C_{32}
- At time t7: C_{23}, C_{22} and C_{32}
- At time t8: C_{33}

So, compared to non-pipelined system which takes 27 time units we need only 8 time units to complete the same task of matrix multiplication.
The speedup factor is = 27/8 i.e. 3.35 times the non-pipelined system.

The dedicated pipelines are meant to perform the same task always.
The data flows in only one direction, that too from stage i to stage j where j = i +1 and also j cannot be less than i or greater than i+1.

GENERAL PIPELINES AND RESERVATION TABLES

Usually there are two categories of pipelines, first is linear pipeline and second is non-linear pipeline. In **linear pipelining** there are no feedback connections from output to input, so output input is totally independent of output. In **non-linear pipelining** there are feed-back connections and feed-forward connections from output to input, so there occurs a change in the input depending upon the type of output. The timing of the non-linear pipelined systems becomes very crucial to avoid any mistakes while feed-back or feed-forward operations. The non-linear systems can be multi-functional systems where more that one functions occur to the data passing through the system.

The MUX are multiplexers which along with a control input would decide which of the inputs should be forwarded if more than one input comes to the MUX.

The elements L1 to L4 are latches which help to just store and forward the input that they receive. These latches are also used as delays to synchronize the movement of data from one stage of pipeline to another stage and also they help avoid any intermediate data loss.

Reservation Tables are used to denote which part of the pipeline gets reserved for operations at specific instance in time. If we are using an uni-functional system then we have single reservation table and when we have a multi-functional system then the number of reservation tables are equal to the number of functions.

For example is we have two functions A and B in a multi-functional non-linear pipelined system then following table can represent the reservation table:

Reservation table for function A						
	t1	**t2**	**t3**	**t4**	**t5**	**t6**
S1	**A**			**A**		
S2		**A**				**A**
S3			**A**		**A**	

Here S1,S2 and S3 are the stages in the pipeline, A is the function, t1 to t6 are the instances of time where a particular stage of pipeline (Sn) is engaged in performing the function (A).

Reservation table for function B						
	t1	**t2**	**t3**	**t4**	**t5**	**t6**
S1			**B**		**B**	
S2		**B**				
S3	**B**			**B**		**B**

Here S1,S2 and S3 are the stages in the pipeline, B is the function, t1 to t6 are the instances of time where a particular stage of pipeline (Sn) is engaged in performing the function (B).

If we are using a uni-functional system then we use X mark instead of A or B and there will only be one reservation table to represent the system.

Different reservation tables can be made for different tasks that occupy the resources of stages S1 to S3.

COLLISIONS AND COLLISION AVOIDANCE

At a given instance of time more than one stage can be engaged in performing tasks, but tasks trying to access the same stage at same time will cause a Collision problem. As one stage can perform only one task at any give instance of time. In order to avoid Collision problems we need the reservation table to determine the proper schedule of the pipeline in order to avoid collision.

Latency: It is the step difference between successive initiation of two tasks. For example if a task I is initiated at time instance t_0 and another task J is initiated at time instance t_5 then the latency would be $t_5 - t_0 = 5$ units of time steps.

Latency Sequence: It is sequence formed by arranging the latencies of series of tasks that are performed one after another in succession. For example if a task I is initiated at time instance t_0 then another task J is initiated at time instance t_5 then another task K is initiated at time instance t_8 then the latency sequence of tasks I, J, and K would be 5,3.

Latency Cycle: It is the latencies that repeat but do not lead to collision. For example 5,3,5,3.

Collision Vector (C): It is the collection of all the latency sequence of a particular task. We can create a collision vector for every task present in the system. This helps us to identify the **Forbidden states** where new tasks should not be initiated as it would result in collision.

Forbidden Set (F): It is the set of all those latencies where new tasks should not be initiated as that would lead to collision.
Consider the following example:
Let there be a task 'I' as the first task in the system whose reservation table is given as follows:

Reservation table for task 'I'									
	t0	**t1**	**t2**	**t3**	**t4**	**t5**	**t6**	**t7**	**t8**
S1	**X**								**X**
S2		**X**	**X**					**X**	
S3				**X**					
S4					**X**	**X**			
S5							**X**	**X**	

According to the reservation table of the First task we have to create the collision vector as this is the first state in which the system is set.
Step1: We find the column differences between every pair of crosses in each row.

	Latency (Forbidden Latency)	**Calculated by**	
Row1	8	t8 – t0	<< Maximum Forbidden Latency
Row2	1,5,6	t2 – t1, t7 – t2, t7 – t1	
Row3	0	t3	
Row4	1	t5 – t4	
Row5	1	t7 – t6	

Step2: Now we have the Forbidden Latency Set (F) = {1, 5, 6, 8}

Step3: Now we compute the collision vector C, which is a binary vector. The values in this vector are either 0 or 1. The number of elements in the vector should be equal to the maximum forbidden latency, in our example it is 8. So we will have 8 elements in the collision vector.

C = { Cn, Cn-1, …., C2, C1}

Where, Cn to C1 should be either 0 or 1 depending up on the following condition:

Ci = 1 only if i is a member of set F

else Ci = 0

So, C1 = 1, C5=1, C6=1 and C8=1 and others are 0.

	C8	**C7**	**C6**	**C5**	**C4**	**C3**	**C2**	**C1**
Value =	1	0	1	1	0	0	0	1

C = {1 0 1 1 0 0 0 1}

Step3: We put the elements of the collision vector in a queue (shift register) and pop out the elements (right shift). If a 1 comes out then we cannot initiate a task at that point and if 0 comes out then we can initiate a new task at that point. In the mean the time 0s are inserted in the left most positions where the places are getting vacant.

	Initial vector	**Right Shift (pop)**	**Next vector**	**Right Shift (pop)**	**Next vector**	**And so on…**
Value =	1011000**1**	1 so new task will not be initiated	**0**1011000	0 so new task can be initiated	**00**101100	

Step4: When we get a 0 we initiate a new task and also we need to recalculate the newly formed collision vector by bitwise ORing the original collision vector with recently achieved collision vector, the result of this ORing would give us the new collision vector.

1011000**1** >> **0**1011000 >> **00**101100 at this point a new task will be introduced so we calculate the new collision vector as follows:

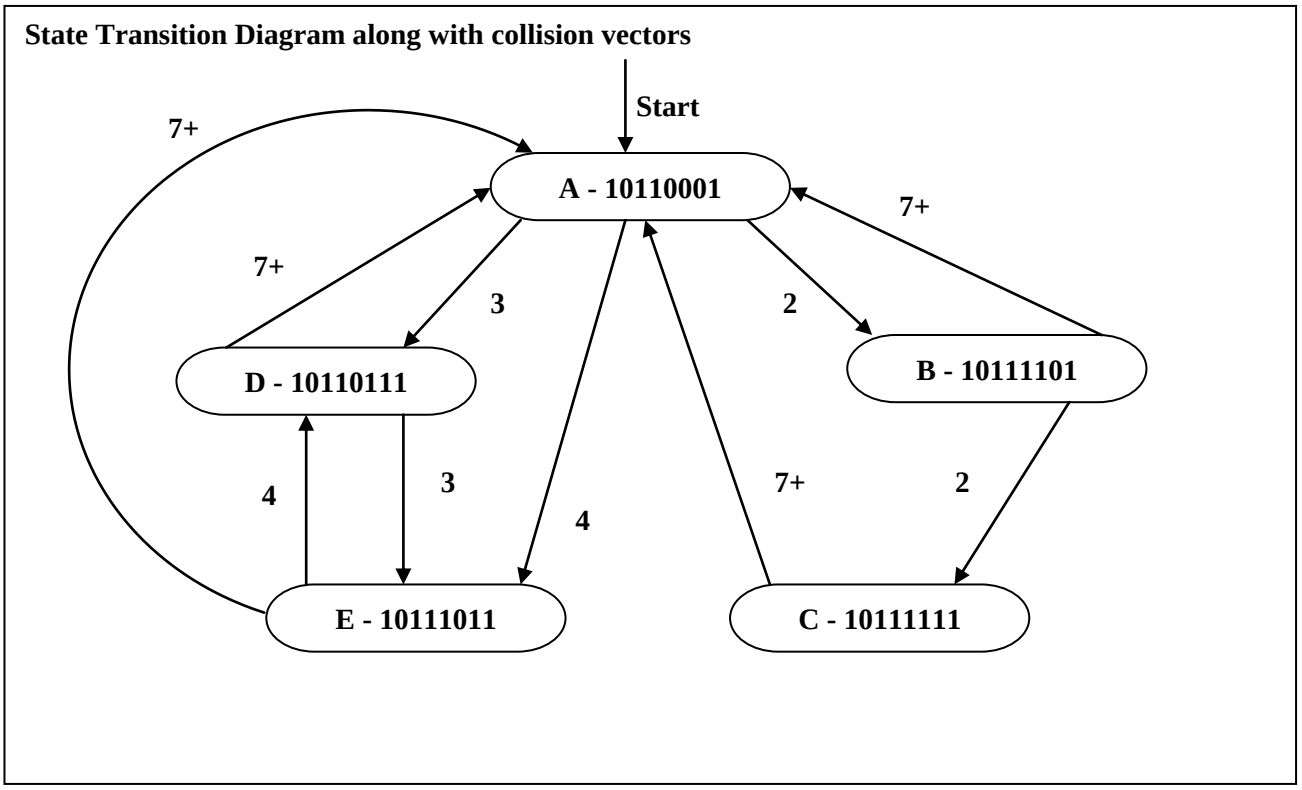
Initial vector	1011000**1**
ORing	**OR**
Vector where new task will be introduced	**00**101100
New resultant vector	**10111101**

So 10111101 becomes the new C (collision vector) now further calculations will be done in the basis of this new C.

Again the above mentioned steps from 1 to 4 can be repeated but now with the newly formed collision vector in place of the old one.

By analyzing the initial collision vector we can identify that at 2,3,4, and 7 we can introduce new tasks in to the system.

Following state diagram represents the state of the pipelined system under consideration.



The cycles in the diagram indicate that there exists a sustainable and stable way in which the tasks can be initiated without collisions.

Optimizing the throughput: To optimize the throughput we have to find the optimum path using which the pipeline utility can be maximized and required time can be minimized and also the cycle can be used to sustain the optimum path. For this we need to find the average latency time for every path and then pick the smallest value path. So here we find the Minimum Average Latency (MAL) from the available set of average latencies.

Average Latency = (Sum of Latencies in the path) / (Number of steps needed to go through the path)
Evaluation of the paths:

Path	Path elements	Average Latency
Path1	A – B – A	(2+7)/2 = 4.5
Path2	A – B – C – A	(2+2+7)/3 = 3.6
Path3	A – D – A	(3+7)/2 = 5
Path4	**A – D – E – D**	**(3+4+3)/3 = 3.3**
Path5	A – D – E – D – A	(3+4+3+7)/4 = 4.2
Path6	A – E – D – A	(4+3+7)/3 = 4.6
Path7	A – E – A	(4+7)/2 = 5.5

The optimum path here is A-D-E-D which is having the least average latency of 3.3 and the system after reaching E can be in a sustainable cycle of E-D. So A-D-E-D has the MAL value of 3.3.

The above mentioned collision vector and its analysis was meant for Uni-functional pipelines, but in case of Multi-functional pipelines we need to consider collisions at following situations and build collision vectors for those situations.
Let there be two functions A and B, also it possible that any function can initiate at any instance of time. Then in this case we need to generate the collision vector for four different conditions:

1. When A initiates and at the same time again A initiates.
2. When B initiates and at the same time again B initiates.
3. When A initiates and at the same time B initiates.
4. When B initiates and at the same time A initiates.

For this a collision matrix is created

CLASSIFICATION OF PIPELINE PROCESSORS

1. Arithmetic Pipelining.
2. Instruction Pipelining.
3. Processor Pipelining.
4. Uni-functional and Multi-functional Pipelining.
5. Static and Dynamic Pipelining.
6. Scalar and Vector Pipelining.

Arithmetic Pipeline : The arithmetic logic units of a computer can be segmentized for pipeline operations in various data formats. Well-known arithmetic pipeline examples are the four-stage pipes used in Star-100, the eight-stage pipes used in the TI-ASC, the up to 14 pipelines stages used in the Cray-1, and up to 26 stages per pipe in the Cyber-205.

Instruction Pipelining : The execution of a stream of instruction can be pipelined by overlapping the execution of the current instruction with the fetch, decode, and operand fetch of subsequent instruction. This technique is also known as instruction lookahead. Almost all high-performance computers are now equipped with instruction-execution pipelines.

Processor Pipelining : This refers to the pipeline processing of the same data stream by a cascade of processors, each of which processes a specific task. The data stream passes the first processor with results stored in a memory block which is also accessible by the second processor. The second processor then passes the refined results to the third, and so on. The pipelining of multiple processors is not yet well accepted as a common practice.

Unifunctional & Multifunction Pipelines : A pipeline unit with a fixed and dedicated function, such as the floating-point adder is called **unifunctional**. The Cray-1 has 12 unifunctional pipeline units for various scalar, vector, fixed-point, and floating-point operations. A **multifunction** pipe may perform different subsets of stages in the pipeline. The TI-ASC. has four multifunction pipeline processors, each of which is reconfigurable for a variety of arithmetic logic operations at different times.

Static & Dynamic Pipelines: A static pipeline may assume only one functional configuration at a time. **Static** pipelines can be either unifunctional or multi-functional. Pipelining is made possible in static pipes only if instructions of the same type are to be executed continuously. The function performed by a static pipeline should not change frequently. Otherwise, its performance may be very low. A **dynamic** pipeline processor permits several functional configurations to exist simultaneously. In this sense, a dynamic pipeline must be multifunctional. On the other hand, a unifunctional pipe must be static. The dynamic configuration needs much more elaborate control and sequencing mechanisms than those for static pipelines. Most existing computers are equipped with static pipes, either unifunctional or multifunctional.

Scalar & Vector Pipelines : Depending on the instruction or data types, pipeline processors can be also classified as scalar pipelines and vector pipelines. A **scalar** pipeline processes a sequence of scalar operands under the control of DO loop. Instructions in a small DO loop are often prefetched into the instruction buffer. The required scalar operands for repeated scalar instructions are moved into a data cache in order to continuously supply the pipeline with operands. The IBM System/360 Model 91 is typical example of a machine equipped with scalar pipelines. However, the Model 91 does not have a cache. **Vector** pipelines are specially designed to handle vector instructions over vector operands. Computers having vector instructions are often called vector processors. The design of a vector pipeline is expended from that of a scalar pipeline. The handling of vector operands in vector pipelines is under firmware and hardware controls (rather than under software controls as in scalar pipelines).

PIPELINING MODELS

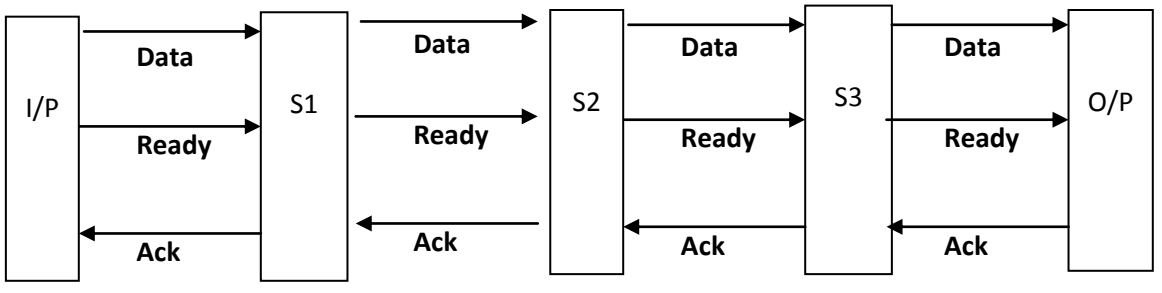
1. Linear Pipelining Models
2. Non-Linear Pipelining Model.
3. Instruction Pipelining Model
4. Arithmetic Pipelining Model.
5. Superscalar Pipelining Model.
6. Super-pipelined Model.

Linear Pipelining Models

A linear pipeline processor is a cascade of processing stages which are linearly connected to perform a fixed function over a stream of data flowing from one end to another.
They are applied for instruction execution, arithmetic computation and memory access.
Depending up on the control of data flow along the pipeline there are two categories of linear pipelines, first is **Asynchronous and Synchronous pipelines**.

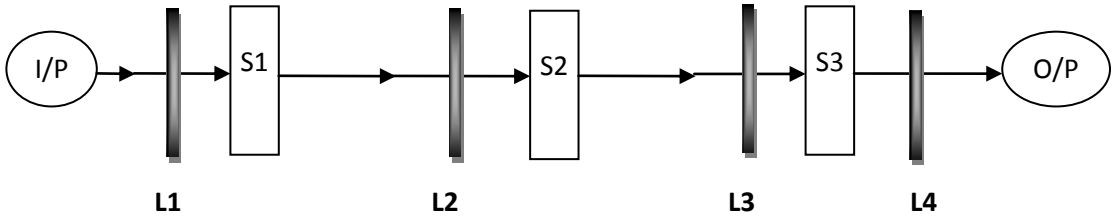
In Asynchronous pipelines the data flow between adjacent stages in asynchronous manner so there is need to handshaking between the stages to avoid collisions. So when stage Si is ready to send then it sends a ready signal to Si+1 and when Si+1 receives the data it then sends an acknowledgement signal to Si about the receipt of the data. So these pipelines have variable throughput.

Asynchronous Linear Pipeline
S1 to S3 are stages in pipeline which usually contain combinational circuits.



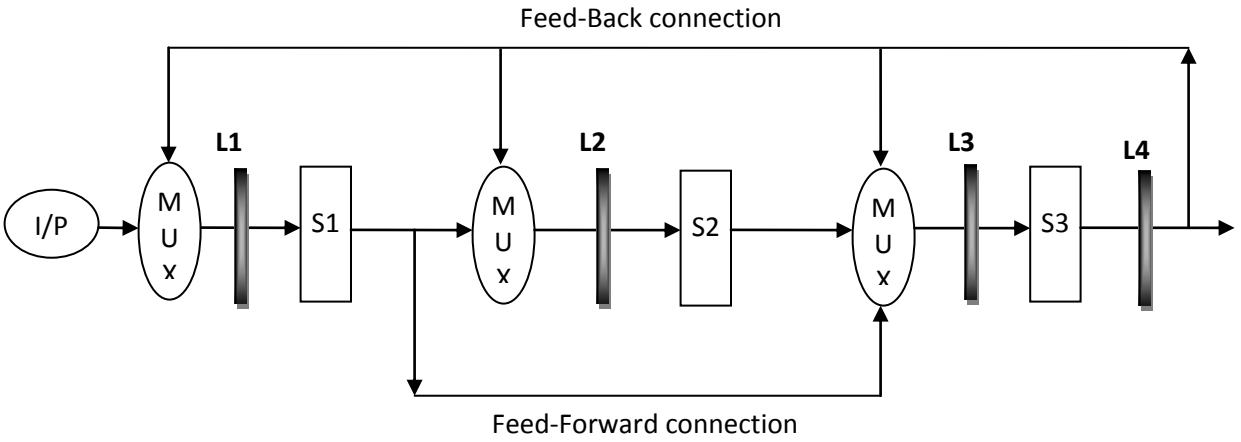
In Synchronous pipelines clocked latches are used to interface between stages. The latches are made using master-slave flip flops. On the arrival of clock pulse, all the latches transfer the data to their next stage.

Synchronous Linear Pipeline
S1 to S3 are stages in pipeline which usually contain combinational circuits.



Nonlinear Pipelining Model:
They are also called Dynamic pipelining models as they can be reconfigured to perform variable functions at different times.
They have feed-back and feed-forward connections in addition to the linear connections.

Non-linear pipelining with feed-back and feed-forward connections



Instruction Pipeline model:
Here a stream of instructions can be executed by the pipeline in an overlapped manner.
A typical instruction sequence of operations which include Instruction Fetch (IF), Instruction Decode (ID), Operand Fetch(OF), Execute (EX) and Memory Write Back (WB).
Each operation can require one or more clock cycles to execute, depending upon the instruction type and memory architecture used.

Pipelined System

W.B.	OUTPUT								
E.X.				I1	I2	I3	I4	I5	
O.F			I1	I2	I3	I4	I5		
I.D.		I1	I2	I3	I4	I5			
I.F.	I1	I2	I3	I4	I5				
	INSTRUCTIONS LOADED IN MEMORY								

OPTIMIZATION OF INSTRUCTION PIPELINES
Various mechanisms are used to optimize the operation of instruction pipeline models.

- Prefetch Buffers:
  - Sequential buffers.
  - Target buffers.
  - Loop buffers.
- Multiple Functional Units:
  - Functional Units.
  - Reservation Stations (RS).
  - Tag Unit.
- Internal Data Forwarding:
  - Store-Load forwarding.
  - Load-Load forwarding.
  - Store-Store forwarding.
- Pipeline Hazard avoidance:
  - Structural hazard.
  - Data hazard.
  - Control hazard.
  - Bubbling the pipeline (pipeline stalling).
  - Scoreboarding.
  - Tomasulo's method.

Prefetch Buffers:
Three types of buffers can be used to match the instruction fetch rate to the pipeline consumption rate.
In one memory-access time, a block of consecutive instructions are fetched into a Prefetch Buffer.
Sequential instructions are loaded into a pair of Sequential Buffers for in-sequence pipelining.
Instructions from a branch target are loaded into a pair of Target Buffers for out-of-sequence pipelining.
Both the buffers work in FIFO fashion.
We can use one buffer to load instructions from memory and use another buffer to feed the instructions into the pipeline.
The two buffers alternate to prevent a collision between instructions flowing into and out of the pipeline.
A Loop Buffer is used to hold sequential instructions contained in a loop. Prefetch instructions are executed repeatedly until all iterations are complete. The Loop buffer works in two ways firstly it contains the instructions sequentially ahead of the current instruction and secondly it recognizes when the target of a branch falls within the loop boundary, so if the target instruction in already there in the loop buffer then extra memory access is avoided.

Multiple Functional Units:
Bottle necks in the pipeline are indicated when we observe lots of crosses in a particular row in the reservation table. To reduce this bottleneck problem multiple functional units are introduced in the system.
The arrangement of hardware is done such that every functional unit has a preceding Reservation Station (RS). These RS help to resolve data or resource dependencies among the successive instructions entering the pipeline. The operands wait in the RS until its dependency problem is solved. Every RS has a unique tag for identification, which is monitored by the Tag Unit.
The Tag unit keeps continuous track of the currently used RS units. This tagging helps the hardware to resolve conflicts between source and destination registers assigned for multiple instructions.
The RS units operate in parallel as soon as the conflicts are resolved.

Internal Data Forwarding:
The throughput of a pipelined system can further improved using Internal Data Forwarding among multiple functional units, for that Memory-access operations can be replaced by Register-transfer operations.
Usually three types of operations are done i.e. Store-Load forwarding, Load-Load forwarding and Store-Store forwarding.

- In Store-Load forwarding we need two set of instructions to transfer data from one register R1 to another register R2. So using one instruction the data from register R1 is Stored in to memory M and using another instruction the data from M is Loaded to R2.
- In Load-Load forwarding we need two set of instructions to transfer data from one memory M to another registers R1 and R2. So using one instruction the data from memory M is Loaded into register R1 and using another instruction the data from M is Loaded into R2.
- In Store-Store forwarding we need two set of instructions to transfer data from two registers R1 and R2 into memory M. So using one instruction the data from register R1 is Stored into memory M and using another instruction the register R2 is Stored into memory M.

So in all the above cases we need data movement from and to memory more than once, which leads to delays.
As memory transfer operations are more time taking and register transfer operations are less time taking, so we apply the following changes.

- In Store-Load forwarding we optimize by eliminating one of the two memory access operations. So using one instruction the data from register R1 is Stored in to memory M and using another instruction the data from R1 is Moved to R2.
- In Load-Load forwarding we optimize by eliminating one of the two memory access operations. So using one instruction the data from memory M is Loaded into the register R1 and using another instruction the data from R1 is Moved to R2.

- In Store-Store forwarding we optimize by eliminating one of the two memory access operations. So using one instruction the data from register R2 is directly Stored in memory M.

This elimination of memory operations drastically improves the throughput.

Pipeline Hazards and Hazard avoidance:
There are three classes of hazards:

Structural Hazards: They arise from resource conflicts when the hardware cannot support all possible combinations of instructions in simultaneous overlapped fashion. A structural hazard occurs when a part of the processor's hardware is needed by two or more instructions at the same time. A structural hazard might occur, for instance, if a program were to execute a branch instruction followed by a computation instruction. Because they are executed in parallel, and because branching is typically slow (requiring a comparison, program counter-related computation, and writing to registers), it is quite possible (depending on architecture) that the computation instruction and the branch instruction will both require the ALU (arithmetic logic unit) at the same time.

Data Hazards: They arise when an instruction depends on the result of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline.

1. **Read after Write (RAW)** or True dependency: An operand is modified and read simultaneously. Because the first instruction may not have finished writing to the operand, the second instruction may use incorrect data. A RAW Data Hazard refers to a situation where we refer to a result that has not yet been calculated, for example:

   i1. R2 <=R1 + R3
   i2. R4 <= R2 + R3

   The 1st instruction is calculating a value to be saved in register 2, and the second is going to use this value to compute a result for register 4. However, in a pipeline, when we fetch the operands for the 2nd operation, the results from the 1st will not yet have been saved, and hence we have a data dependency.
   We say that there is a data dependency with instruction 2, as it is dependent on the completion of instruction 1
2. **Write after Read (WAR)** or Anti dependency: Read an operand and write soon after to that same operand. Because the write may have finished before the read, the read instruction may incorrectly get the new written value. A WAR Data Hazard represents a problem with concurrent execution, for example:

   i1. R4 <= R1 + R3
   i2. R3 <= R1 + R2

   If we are in a situation that there is a chance that i2 may be completed before i1 (i.e. with concurrent execution) we must ensure that we do not store the result of register 3 before i1 has had a chance to fetch the operands.
3. **Write after Write (WAW)** or Output dependency: Two instructions that write to the same operand are performed. The first one issued may finish second, and therefore leave the operand with an incorrect data value. A WAW Data Hazard is another situation which may occur in a Concurrent execution environment, for example:

   i1. R2 <= R1 + R2
   i2. R2 <= R4 x R7

   We must delay the WB (Write Back) of i2 until the execution of i1

Control Hazards: They arise from the pipelining of branches and other instructions that change the PC.
Branching hazards (also known as control hazards) occur when the processor is told to branch - i.e., if a certain condition is true, then jump from one part of the instruction stream to another - not necessarily to the next instruction sequentially. In such a case, the processor cannot tell in advance whether it should process the next instruction (when it may instead have to move to a distant instruction). This can result in the processor doing unwanted actions.

Eliminating Hazards
Eliminating a hazard often requires that some instructions in the pipeline to be allowed to proceed while others are delayed. When the instruction is stalled, all the instructions issued later than the stalled instruction are also stalled. Instructions issued earlier than the stalled instruction must continue, since otherwise the hazard will never clear.
We can delegate the task of removing data dependencies to the compiler, which can fill in an appropriate number of NOP instructions between dependent instructions to ensure correct operation, or re-order instructions where possible.

Bubbling the Pipeline: Bubbling the pipeline (a technique also known as a pipeline break or pipeline stall) is a method for preventing data, structural, and branch hazards from occurring. As instructions are fetched, control logic determines whether a hazard could/will occur. If this is true, then the control logic inserts NOPs into the pipeline. Thus, before the next instruction (which would cause the hazard) is executed, the previous one will have had sufficient time to complete and prevent the hazard. If the number of NOPs is equal to the number of stages in the pipeline, the processor has been cleared of all instructions and can proceed free from hazards. This is called flushing the pipeline. All forms of stalling introduce a delay before the processor can resume execution.

Other methods include on-chip solutions such as:

- **Scoreboarding method:** Scoreboarding is a centralized method, used in the CDC 6600 computer, for dynamically scheduling a pipeline so that the instructions can execute out of order when there are no conflicts and the hardware is available. In a scoreboard, the data dependencies of every instruction are logged. Instructions are released only when the scoreboard determines that there are no conflicts with previously issued and incomplete instructions. If an instruction is stalled because it is unsafe to continue, the scoreboard monitors the flow of executing instructions until all dependencies have been resolved before the stalled instruction is issued.
  Instructions are decoded in order and go through the following four stages:
  1. Issue: The system checks which registers will be read and written by this instruction. This information is remembered as it will be needed in the following stages. In order to avoid output dependencies (WAW - Write after Write) the instruction is stalled until instructions intending to write to the same register are completed. The instruction is also stalled when required functional units are currently busy.
  2. Read operands: After an instruction has been issued and correctly allocated to the required hardware module, the instruction waits until all operands become available. This procedure resolves read dependencies (RAW - Read after Write) because registers which are intended to be written by another instruction are not considered available until they are actually written.
  3. Execution: When all operands have been fetched, the functional unit starts its execution. After the result is ready, the scoreboard is notified.
  4. Write Result: In this stage the result is about to be written to its destination register. However, this operation is delayed until earlier instructions—which intend to read registers this instruction wants to write to—have completed their read operands stage. This way, so called data dependencies (WAR - Write after Read) can be addressed.

- **Tomasulo's method:** The Tomasulo algorithm is a hardware algorithm developed in 1967 by Robert Tomasulo from IBM. It allows sequential instructions that would normally be stalled due to certain dependencies to execute non-sequentially (out-of-order execution). It was first implemented for the IBM System/360 Model 91's floating point unit.
  The Tomasulo algorithm also uses a common data bus (CDB) on which computed values are broadcast to all the reservation stations that may need it. This allows for improved parallel execution of instructions which may otherwise stall under the use of scoreboarding.
  Tomasulo's algorithm implements **register renaming** through the use of what are called **reservation stations**. Reservation stations are buffers which fetch and store instruction operands as soon as they're available. Source operands point to either the register file or to other reservation stations. Each reservation station corresponds to one instruction. Once all source operands are available, the instruction is sent for execution, provided a functional unit is also available. Once execution is complete, the result is buffered at the reservation station. Thus, unlike in scoreboarding where the functional unit would stall during a WAR hazard, the functional unit is free to execute another instruction. The reservation station then sends the result to the register file and any other reservation station which is waiting on that result. WAW hazards are handled since only the last instruction (in program order) actually writes to the registers. The other results are buffered in other reservation stations and are eventually sent to any instructions waiting for those results. WAR hazards are handled since reservation stations can get source operands from either the register file or other reservation stations (in other words, from another instruction). In Tomasulo's algorithm, the control logic is distributed among the reservation stations, whereas in scoreboarding, the scoreboard keeps track of everything.

ARITHMETIC PIPELINING MODEL:

The principles used in instruction pipelining can be used in order to improve the performance of computers in performing arithmetic operations such as add, subtract, and multiply. In this case, these principles will be used to realize the arithmetic circuits inside the ALU.

Some functions of the ALU can be pipelined and Complex functions has to be decomposed

The major task here is to find a multistage sequential algorithm to compute the arithmetic function and also the algorithm's steps should be balanced. After this the stages in the pipeline can be designed in which buffers (synchronizing latches) are to be placed between the stages.

Usually arithmetic pipelines are static pipelines that perform only one function. They perform fixed-point and floating-point operations using separate pipelined ALUs. When integrated in a system they are known as Integer Units and Floating-point Units. Although, Multi-functional pipelines are also built using feed-back and feed-forward interconnections.

The majority of mathematical operations are implemented by using a combination of Add and Shifting operations.

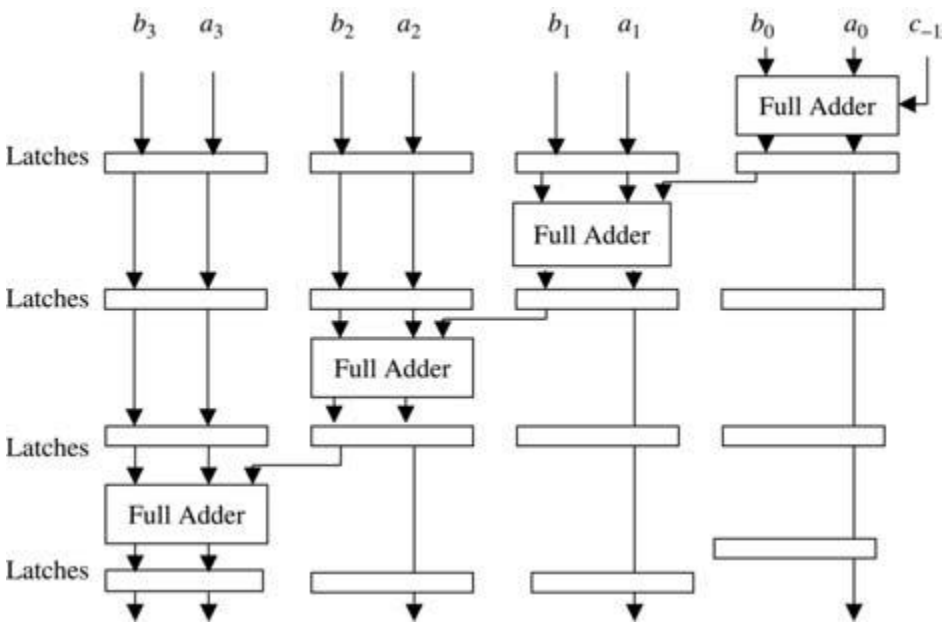
Fixed-Point Arithmetic Pipelines:

The basic fixed point arithmetic operation performed inside the ALU is the addition of two n-bit operands A and B.

Addition of these two operands can be performed using a number of techniques where the techniques may differ in basically two attributes: degree of complexity and achieved speed.

A simple realization may lead to a slower circuit while a complex realization may lead to a faster circuit.

Fixed-Point Arithmetic Pipeline Model with carry propagating to next stage



Floating-Point (FP) Arithmetic Pipelines:

The main operations needed in FP addition are exponent comparison (EC), exponent alignment (EA), addition (AD), and normalization (NZ).

The pipeline organization is to have a four-stage pipeline each performing an operation from EC, EA, AD, and NZ.

For example, if we want to add two floating point numbers,

$A = a * 2^p$ and $B = b * 2^q$ to generate the result $C = A+B = d * 2^s$

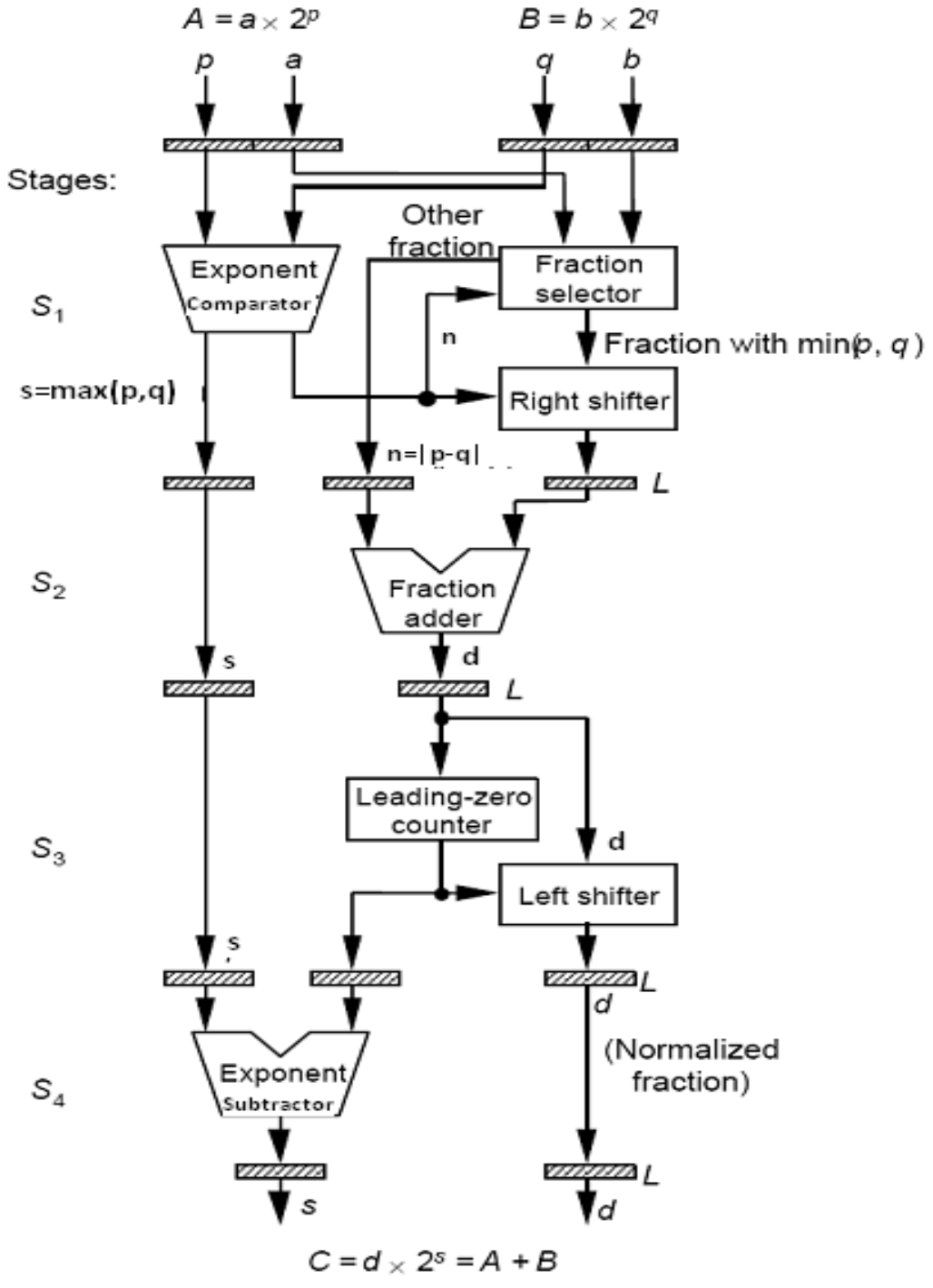
Where, **$d = a + b$ and $s = max(p,q)$**

So, we first equalize the exponents p and q by selecting the greatest and accordingly adjust the fractions a or b whose exponent is changed. Then we add the fractions **a** and **b** to generate the result and if the result has leading zeros after decimal point then we re-adjust that fraction and its respective exponent.

The steps in details would be;

1. Find the difference between the exponents, let the difference be **n**, which should be an absolute value (only positive). **$n = | p-q |$**
2. Compare the exponents and find the smallest. Then add the difference value **n** to the smallest exponent.
3. Due to step 2 we have adjust the corresponding fraction part so we **Right Shift** the fraction part by **n** digits.
4. Now we add the fraction parts to generate the fraction result. **$C = (a+b) * 2^s$**
5. If we encounter leading zeros (0.**000**3) in the result then we have readjust this by eliminating the leading zeros. We count the number of leading zeros, say **m** and we **Left Shift** the fraction part by **m** digits and also reduce the exponent by a value of **m**.

Floating-Point (FP) Arithmetic Pipeline for adding two FP numbers



This is useful when we have two vectors, each containing a series of FP numbers and we want to use the above system to add the elements of the vectors in a pipelined fashion.

SUPERSCALAR AND SUPER PIPELINING MODELS

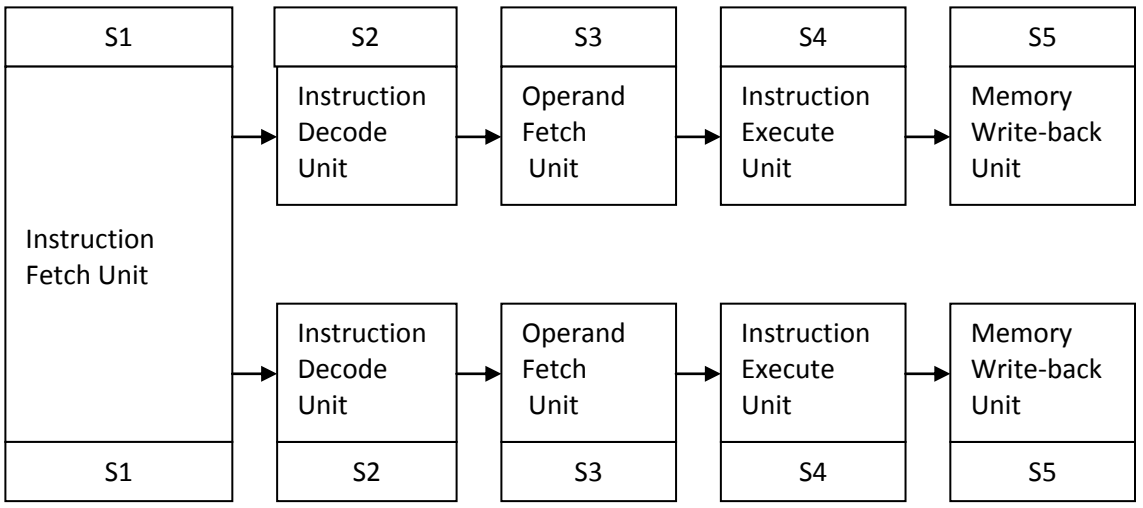
Scalar pipelined machines:

In basic scalar machine one instruction is issued per cycle.
The CPU is essentially a scalar processor consists of multiple functional units.
The floating-point unit can be built on a coprocessor attached to the CPU.
Generic RISC processors are called scalar RISC because they are designed to issue one instruction per cycle, similar to the base scalar processor.

Superscalar pipelined machines:

Usually they have a single **Fetch Unit**, which fetches the series of instructions and pushes them to the pipeline.
There are more than one pipelines which are connected to the Fetch Unit and the pipelines can work concurrently.
Go beyond single instruction pipeline, so dispatch multiple instructions per cycle
It is possible to have multiple ALUs in a particular stage.
Functional units in the various stages may take longer than one clock cycle to execute.

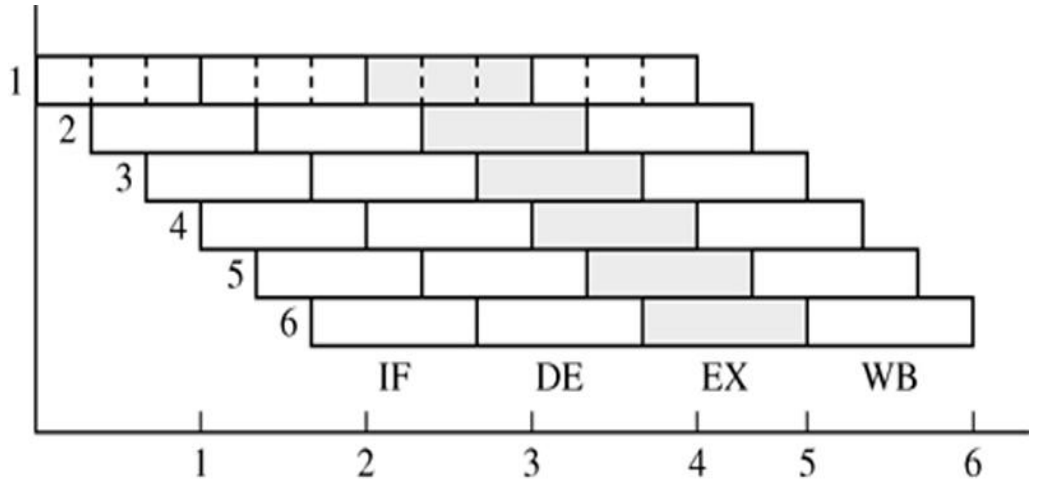
Superscalar pipelined machines:



In a superscalar processor, multiple instruction pipelines are required. This implies that multiple instructions are issued per cycle and multiple results are generated per cycle.
Superscalar processors are designed to exploit more instruction-level parallelism in user programs. Only independent instructions an be executed in parallel without causing a wait state.
The instruction-issue degree in a superscalar processor is limited to 2-5 in practice.
The **degree** of superscalar machine refers to the number of instructions issued per cycle. So if **m** instructions are issued per cycle then the degree of the superscalar machine would be **m**.
Instruction Level Parallelism (ILP) is the maximum number of instructions that can be simultaneously executed in the pipeline. So ILP should be equal to **m** in order to fully utilize the pipelines.

Super-pipelining Models:
Super-pipelining simply allows a processor to improve its performance by running the pipeline at a higher clock rate.
Double internal clock speed gets two tasks per external clock cycle.
If a super-pipelined system is of degree **m (**number of instructions issued per cycle) then the pipeline cycle time would be **1/n** of the base cycle (**n**). So, all the operations will take **m** short cycles each of **1/n** time span. Hence we need a very high-speed clocking mechanism.

Super-pipelined machine of Degree m=3

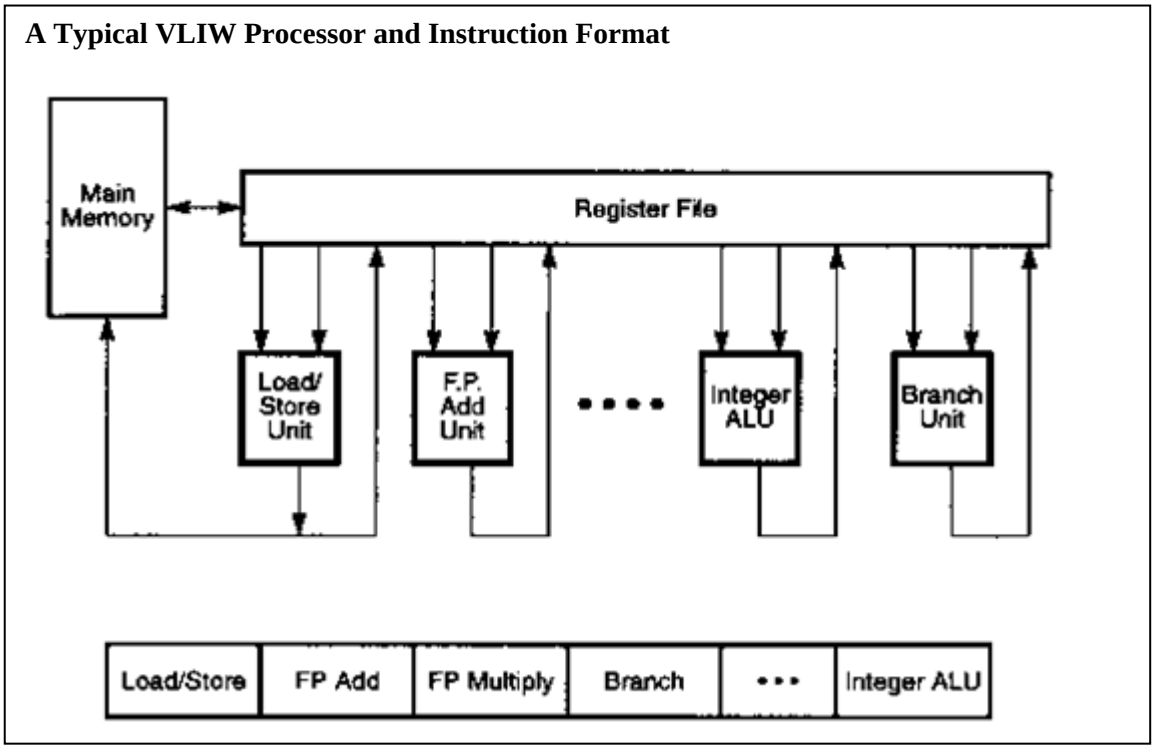


Simultaneously multiple instructions advance through the pipeline stages.
Multiple functional units leads to higher instruction execution throughput.
Able to execute instructions in an order different from that specified by the original program. Out of program order execution allows more parallel processing of instructions.

VLIW – COMPUTERS

A typical VLIW (very long instruction word) machine has instruction words hundreds of bits in length.
Multiple functional units are used concurrently in a VLIW processor.
All functional units share the use of a common large register file.
The VLIW Compiler prepares fixed packets of multiple operations that give the full "plan of execution"
Dependencies are determined by compiler and used to schedule according to function unit latencies
Function units are assigned by compiler and correspond to the position within the instruction packet ("slotting")
Compiler produces fully-scheduled, hazard-free code so here the hardware doesn't have to "rediscover" dependencies.
Compatibility across implementations is a major problem as the VLIW code won't run properly with different number of function units or different latencies and unscheduled events (e.g., cache miss) stall the entire processor.
One VLIW instruction encodes multiple operations; specifically, one instruction encodes at least one operation for each execution unit of the device. For example, if a VLIW device has five execution units, then a VLIW instruction for that device would have five operation fields, each field specifying what operation should be done on that corresponding execution unit. To accommodate these operation fields, VLIW instructions are usually at least 64 bits wide, and on some architectures are much wider.
The following is an instruction for the SHARC (Super Harvard Architecture Single-Chip Computer). In one cycle, it does a floating-point multiply, a floating-point add, and two autoincrement loads. All of this fits into a single 48-bit instruction: f12=f0*f4, f8=f8+f12, f0=dm(i0,m3), f4=pm(i8,m9);

A Typical VLIW Processor and Instruction Format



PROGRAM FLOW MECHANISMS

- Control Flow
- Data Flow
- Demand-Driven Mechanisms (Reduction machines)

Control Flow Mechanism

- Control-flow computers use shared memory to hold program instructions and data objects.
- In Control-flow computers since variables are updated by many instructions, there may be side effects on other instructions. These side effects frequently prevent parallel processing.

Data Flow Mechanism

- In a dataflow computer, the execution of an instruction is driven by data availability instead of being guided by a program counter.
- Computational results (data tokens) are passed directly between instructions.
- Instructions in dataflow machines are unordered and can be executed as soon as their operands are available; data is held in the instructions themselves. Data tokens are passed from an instruction to its dependents to trigger execution. So they are more appropriate for parallel processing.
- No need for
    - shared memory
    - program counter
    - control sequencer
- Special mechanisms are required to
    - detect data availability
    - match data tokens with instructions needing them
    - enable chain reaction of asynchronous instruction execution

Demand-Driven Mechanisms (Reduction machines)

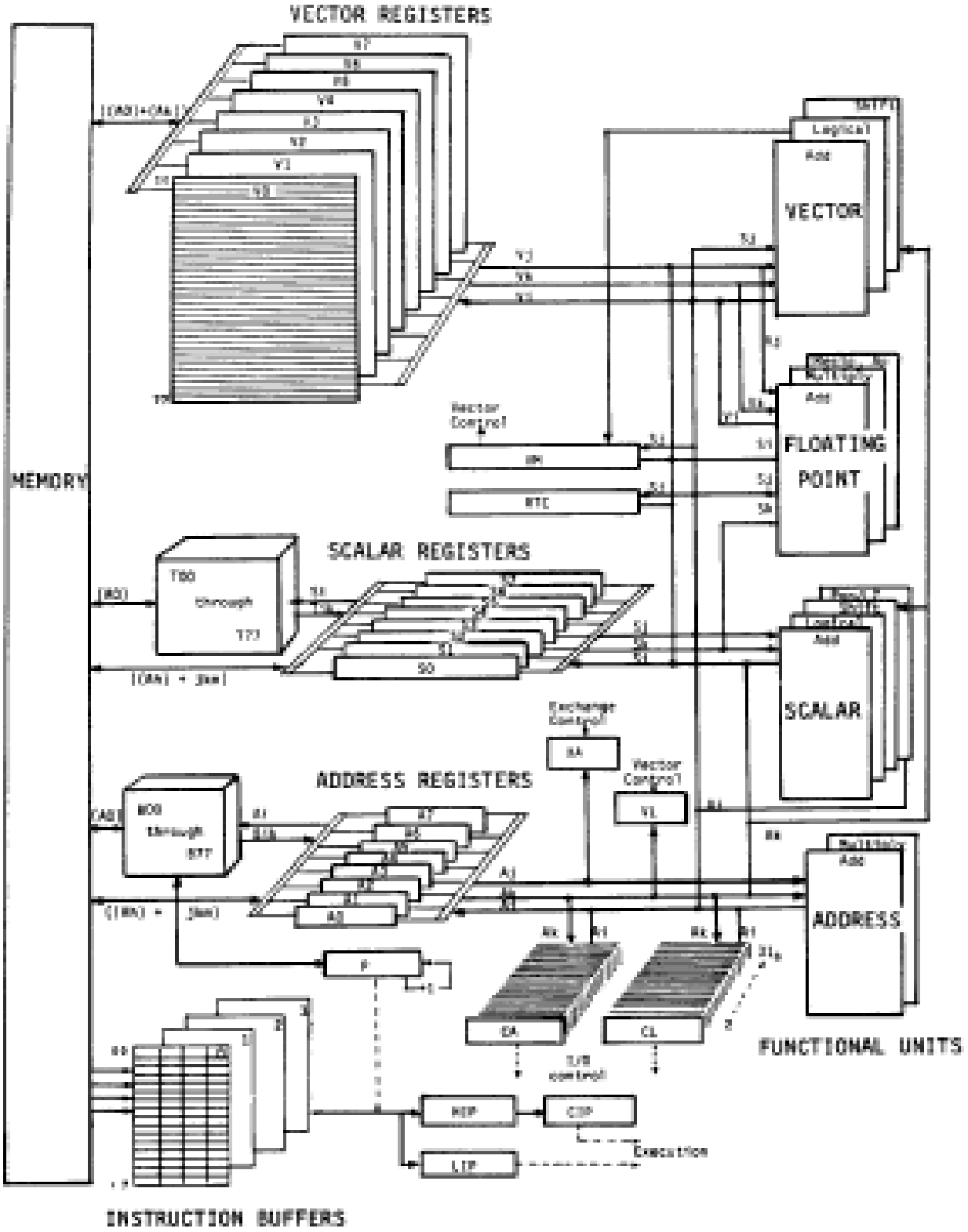
- In a reduction machine, the computation is triggered by the demand for an operation's result.
- A demand-driven computation corresponds to lazy evaluation, because operations are executed only when their results are required by another instruction.

ARCHITECTURE OF CRAY-1

The CRAY-1 is the only computer to have been built to date that satisfies ERDA's Class VI requirement (a computer capable of processing from 20 to 60 million floating point operations per second).
The CRAY-I's Fortran compiler (CVT) is designed to give the scientific user immediate access to the benefits of the CRAY-I's vector processing architecture.
The arithmetic calculations are performed in discrete steps, with each step producing interim results used in subsequent steps. Through a technique called **"chaining,"** the CRAY-1 **vector functional units**, in combination with **scalar and vector registers**, generate interim results and use them again immediately without additional memory references.
Other than its computational capabilities, features that are worth mentioning are :

- its small size, which reduces the distances through which the electrical signals must travel within the computer's framework and allows a 12.5 nanosecond clock period.
- A one million word semiconductor memory equipped with error detection and correction logic (SECD~D); its 64-bit word size; and
- Its optimizing Fortran compiler.

Block-diagram of registers in CRAY-1



Functional Units
There are 12 functional units, organized in four groups: address, scalar, vector, and floating point.
Each functional unit is pipelined into single clock segments.
All of the functional units can operate concurrently so that in addition to the benefits of pipelining (each functional unit can be driven at a result rate of 1 per clock period) we also have parallelism across the units too.

Registers
The basic set of programmable registers are as follows:

- 8 number of 24-bit address (A) registers
- 64 number of 24-bit address-save (B) registers
- 8 number of 64-bit scalar (S) registers
- 64 number of 64-bit scalar-save (T) registers
- 8 number of 64-word (4096-bit) vector (V) registers

Instruction Formats
Instructions are expressed in either one or two 16-bit parcels.

Vector Instructions
On the CRAY-1, vector instructions may issue at a rate of one instruction parcel per clock period.
All vector instructions are one parcel instructions (parcel size = 16 bits).
Vector instructions place a reservation on whichever functional unit they use, including memory, and on the input operand registers.

System Software
CRAY Operating System (COS) and CRAY Fortran Compiler (cFr) are the main softwares that work in CRAY-1 COS:

- It is a batch operating system capable of supporting up to 63 jobs in a multiprogramming environment,
- It is designed to be the recipient of job requests and data files from front-end computers.
- Output from jobs is normally staged back to the front-ends upon job completion.

Other CRAY-1 software includes Cray Assembler Language (CAL) which is a powerful macro assembler, a full range of utilities including a text editor, and some debug aids.

Functional Block Diagram of CRAY-1