02/05/2020

unit - 5

Pipelining

TOPIC 1: Pipelining : Basic concepts

→ The speed of execution of programs is influenced by many factors. 1→ one way to improve performance is to use faster circuit technology to build the processor and the main memory.

2→ Another possibility is to arrange the H.W so that more than one operation can be performed at the same time.

→ "Pipelining" is particularly effective way of organizing concurrent activity in a computer system. The basic idea is very simple.

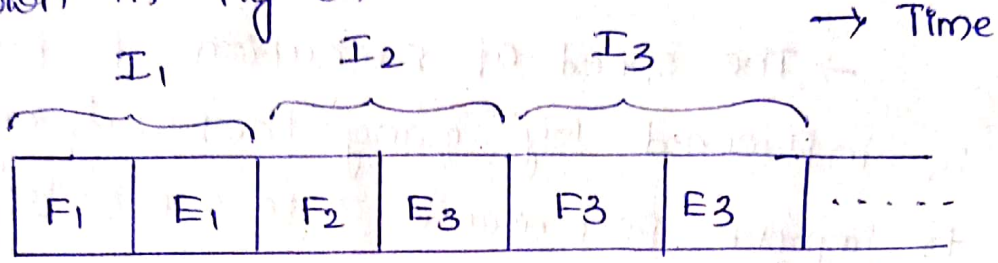
→ In pipelined architecture,

* Multiple instructions are executed parallely.
* This style of executing the instructions is highly efficient.

→ consider how the idea of pipelining can be used in computer. The processor executes a program by fetching and executing instructions, one after the other.

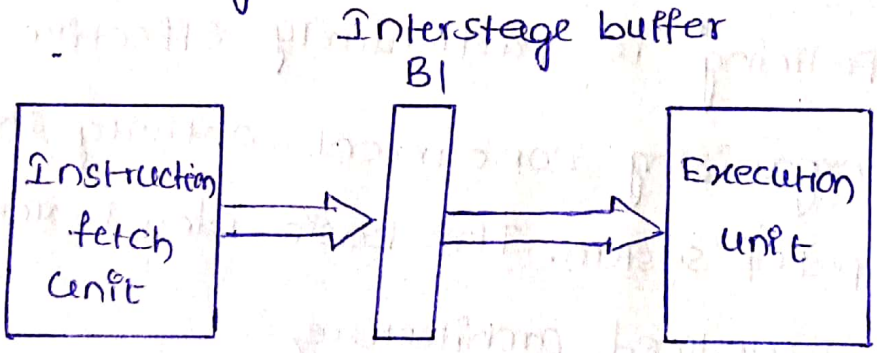
→ Let F_i and E_i refer to the fetch and execute steps for Instruction I_i.

→ Execution of a program consists of a sequence of fetch and Execute steps as shown in fig. (a)



(a) Sequential execution

Now consider a computer that has two separate hardware units, one for fetching instructions and another for executing them, as shown in fig (b).

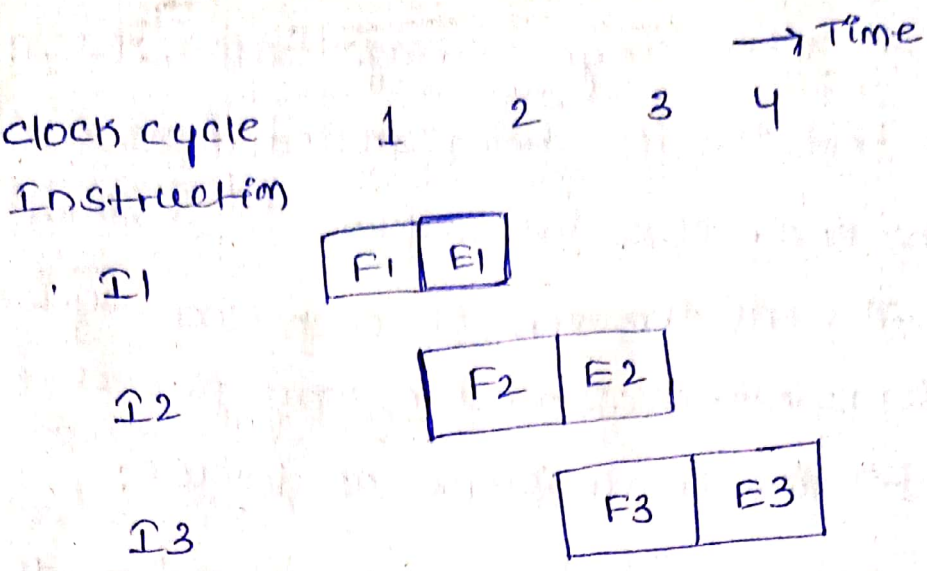


(b) Hardware Organization

→ The instruction fetched by the fetch unit is deposited in an intermediate storage buffer, B1.

→ This buffer is needed to enable the execution unit to execute the instruction while the fetch unit is fetching the next instruction.

→ The results of execution are deposited in the destination location specified by the instruction.



(c) Pipelined execution

fig: 1.1 Basic idea of instruction pipelining.

→ The computer is controlled by a clock, whose period is such that the fetch and execute steps of any instruction can each be completed in one clock cycle.

→ Operation of the computer proceeds as in fig: c.

→ In the first clock cycle, the fetch unit fetches an instruction I_1 (step F1) and stores it in buffer B1 at the end of the clock cycle.

→ In the second clock cycle, the instruction fetch unit proceeds with the fetch operation for instruction I2 (step F2).

→ Meanwhile, the Execution unit performs the operation specified by instruction I1, which is available to it in buffer B1 (step E1).

→ An inter-stage storage buffer, B1, is needed to hold the i/f being passed from one stage to the next. New information is loaded into this buffer at the end of each clock cycle.

→ A pipelined processor may process each instruction in 4 steps as follows:

F Fetch: read the instruction from the memory.

D Decode: decode the instructions & fetch the source operand (s). ↳ code into understandable

E Execute: perform the operation specified by the instruction. ALU

N Write: store the result in the destination location.

→ The sequence of events for this case is shown in fig 8.2(a). Four instructions are in progress at any given time. This means that 4 distinct hardware units are needed, as shown fig 8.2(b)

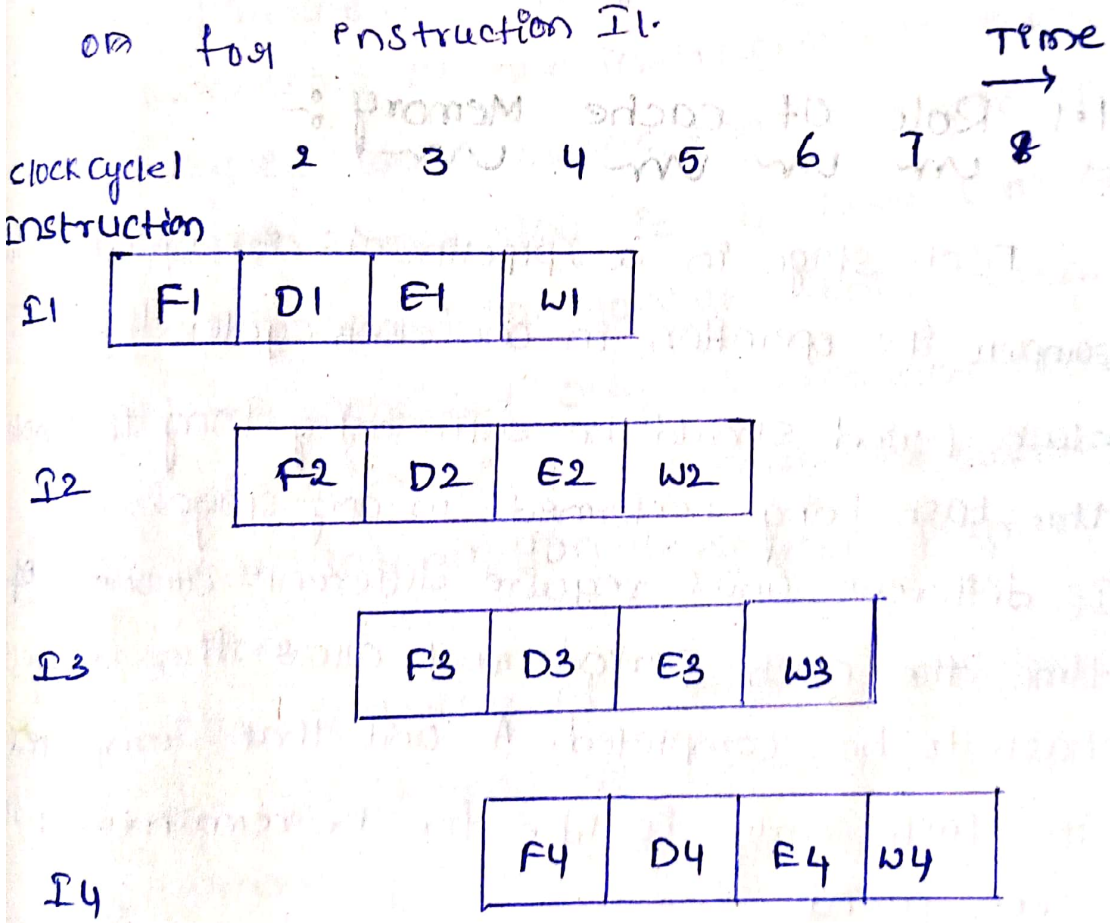
→ As an instruction progress through the pipeline, all the information needed by the stages downstream must be passed along.

For Ex: During clock cycle 4, the i/f in the buffer is as follows.

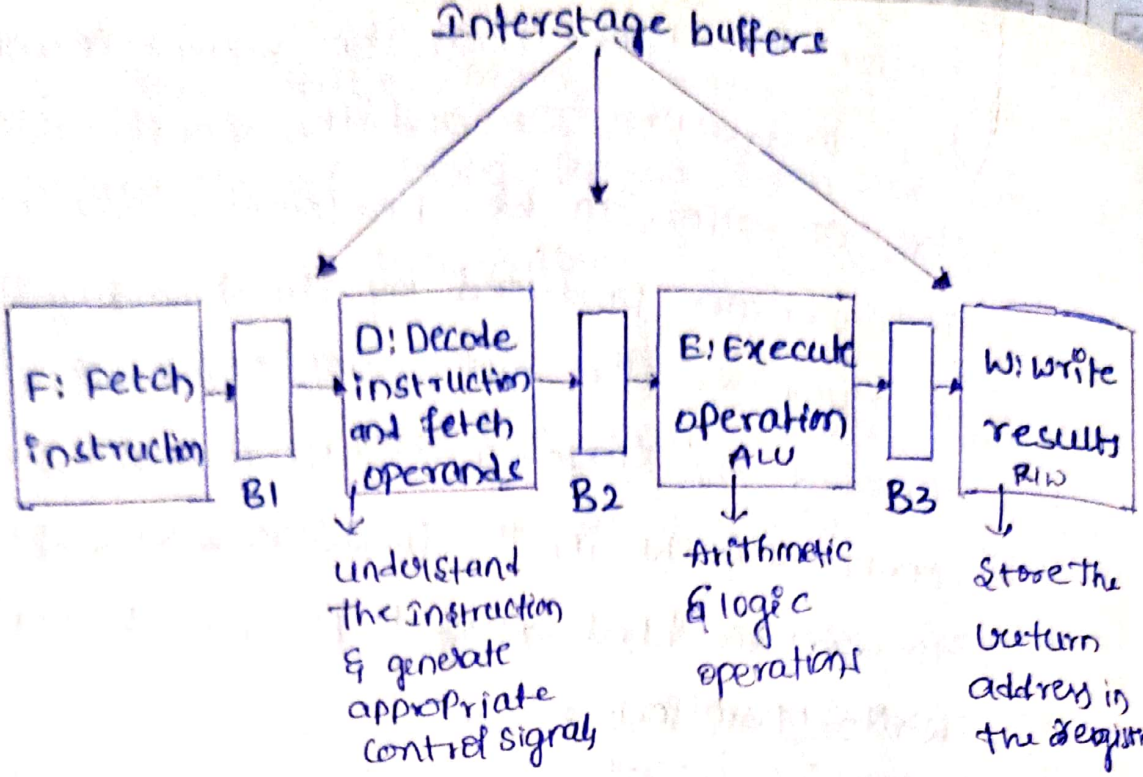
1. Buffer B1 holds instruction I3, which was fetched in cycle 3 and is being decoded by the instruction-decoding unit.

2. Buffer B2 holds both the source operands for instruction I2 and the specification of the operation to be performed. This is the information produced by the decoding H/W in cycle 3. The buffer also holds the I/f needed by stage E, this I/f must be passed on to stage W in the following clock cycle to enable that stage to perform the required write operation.

3. Buffer B3 holds the results produced by the execution unit and the destination I/f or for instruction I1.



(a) Instruction Execution is divided into 4 steps
→ F, D, E, W Each of this diff units present in the processor. these units can work independently



(b) Hardware Organization

1.2 A 4-Stage pipeline.

1.1 Role Of cache Memory :-

Each stage in a pipeline is excepected to complete its operation in one clock cycle. Hence, the clock period should be sufficientey long to complete the task being performed in any stage.

If different units require different amounts of time, the clock period must allow the longest task to be completed. A unit that completes its task early is idle for the remainder of the clock period.

Hence, pipelining is most effective in improving performance if the tasks being performed in different stages require about the same amount of time.

→ This consideration is particularly important for the instruction fetch step, which is assigned one clock period in fig. 1.2(a).

→ The use of cache memories solves the memory access problem. (↓ segmentation fault) In particular, when a cache is included on the same chip as the processor, access time to the cache is usually the same as the time needed to perform other basic operations inside the processor.

1.2 Pipeline Performance

The pipelined processor in fig 1.2 completes the processing of one instruction in each clock cycle, which means that the rate of instruction processing is 4 times that of sequential operation. However, this increase would be acheived only if pipelined operation as depicted in fig. 1.2(a) could be sustained with out interruption throughout program execution.

For a variety of reasons, one of the pipeline stages may not be able to complete its processing task for a given instruction in the time alloted. For ex: stage E in the four-stage pipeline of fig 1.2b is responsible for arithmetic and logic operations, and one clock cycle is assigned for this task.

Although this may be sufficient for most operations, some operations, such as divide, may require more time to complete.

Fig 1.3 shows an example in which the operation is specified in instruction I2 requires three cycles to complete, from cycle 4 through cycle 6.

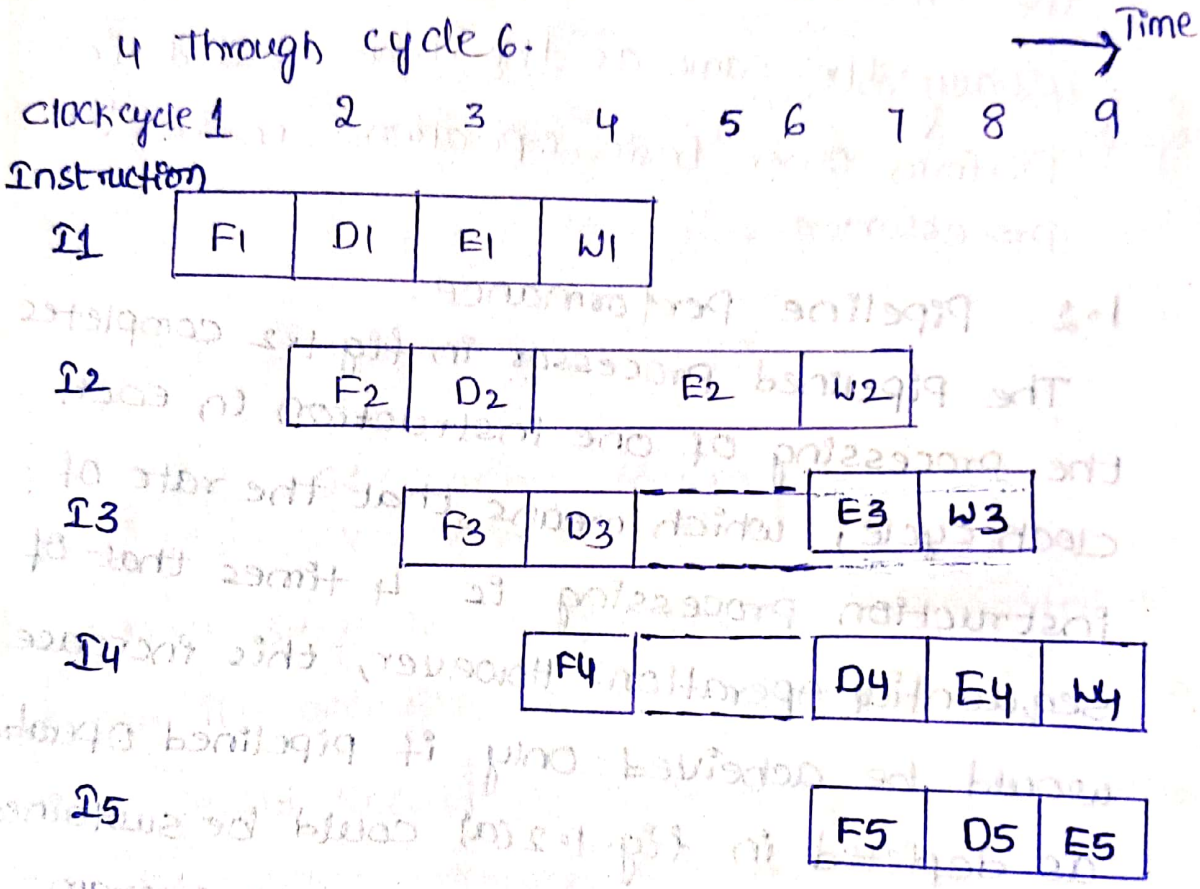


fig: 1.3 Effect of an execution operation taking more than one clock cycle.

Thus, in cycles 5 and 6, the write stage must be told to do nothing, because it has no data to work with. Meanwhile, the information in buffer B2 must remain intact (not damaged) until the Execute stage has completed its operation. This means that stage 2 &, in turn, stag 1 is blocked from accepting new instructions because the i/t in B1 can not be overwritten. Thus, step D4 & F5 must be postponed as shown above.

Pipelined operation in fig 5.3 is said to have been stalled for 2 clock cycles. Any condition that causes the pipeline to stall is called a 'hazard'.

Pipeline Hazards:-

Hazards:- situations that would cause incorrect execution.

→ There are 3 classes of hazards.

1. Structural hazard:-
Resource conflicts when the H.W can not support all possible combination of instructions simultaneously.
2. Data hazard:
An instruction depends on the results of a previous instruction.
3. Branch hazard:
Instructions that change the PC.

5.2 DATA HAZARDS

→ Hazards cause imperfect pipelining.

→ A data hazard is a situation in which the pipeline is stopped (stalled) because the data to be operated on are delayed for some reason.

→ Means a data value is not available when/ where it is needed.

→ Consider a program that contains two instructions, I1 followed by I2. When this program is executed in a pipeline, the execution of I2 can begin before the execution of I1 is completed. This means that the results generated by I1 may not be available for I2.

→ Assume that A=5, and consider the following two operations:

$$A \leftarrow 3+A$$
$$B \leftarrow 4*A$$

When these operations are performed in the order given, the result is B=32. But if they performed concurrently, the value of A used in computing B would be the Original value, 5 leading to an incorrect result.

→ If these two operations are performed by instructions in a program, then the instructions must be executed one after the other, because

the data used in the second instruction depend on the result of the first instruction.

On the other hand, the two operations

$$A \leftarrow 5 \times C$$
$$B \leftarrow 20 + C$$

can be performed concurrently, because these operations are independent.

→ This example illustrates a basic constraint that must be enforced to guarantee correct results.

→ When two operations depend on each other, they must be performed sequentially in the correct order.

→ consider the pipeline in fig: 1.2. The data dependency just described arises when the destination of one instruction is used as a source in the next instruction.

For example, the two instructions give rise to a data dependency.

```
Mul  R2,R3,R4
Add  R5,R4,R6.
```

The result of the multiply instruction is placed into register R4, which in turn is one of the two source operands of the Add instruction.

→ Assuming that the multiply operation takes one clock cycle to complete, execution would proceed as shown in fig 5.6.

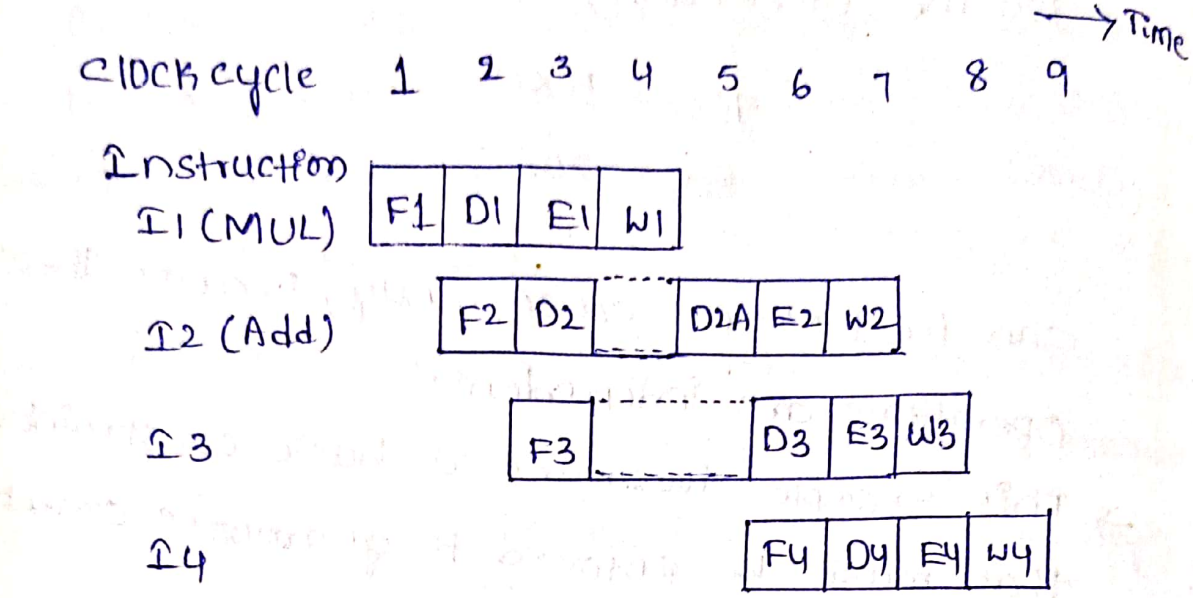


fig:5.6 Pipeline stalled by data dependency between D2 and W1.

As the Decode unit decodes the Add instruction in cycle 3, it realizes that R4 is used as a source operand. Hence, the D step of that instruction can not be completed until the W step of the multiply instruction has been completed.

→ Completion of step D2 must be delayed to clock cycle 5, and is shown as step D2A in the figure.

→ Instruction I3 is fetched in cycle 3, but its decoding must be delayed because step D3 can not precede D2. Hence, Pipelined execution is stalled for

5.2.1 Operand Forwarding

→ The data hazard just described arises because one instruction, instruction I2 in fig: 5.6, is waiting for data to be written in the register file. However, these data are available at the output of the ALU once the Execute stage completes step E1.

→ Hence, the delay can be reduced, or possibly eliminated, if we arrange for the result of instruction I1 to be forwarded directly for use in step E2.

→ Fig 5.7(a) shows a part of the processor data path involving the ALU and the register file. This arrangement is similar to the three-bus structure except that registers SRC1, SRC2, & RSLT have been added. These registers contain the inter-stage buffers needed for pipelined operation, as illustrated in fig: 5.7b with reference to fig. 1.2b registers SRC1 and SRC2 are part of buffer B2 and RSLT is part of B3.

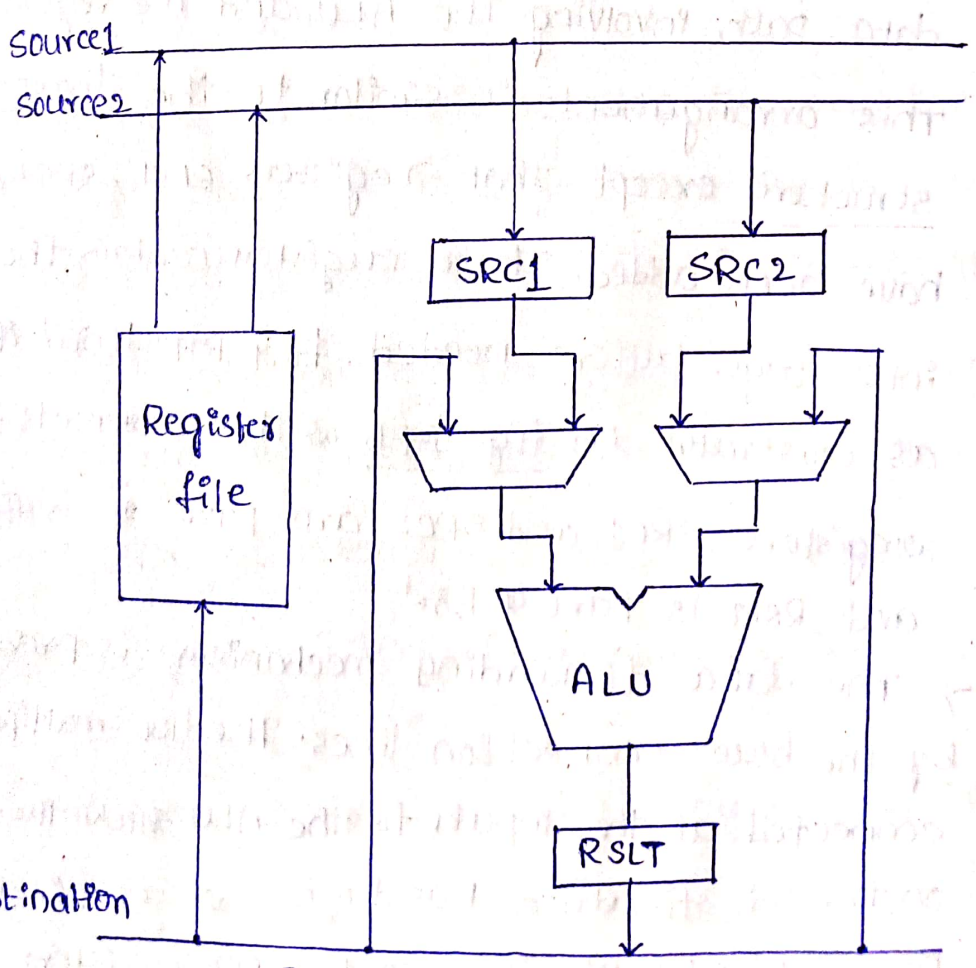
→ The data forwarding mechanism is provided by the blue connection lines. The two multiplexers connected at the inputs to the ALU allow the data on the destination bus to be selected instead of the contents of either SRC1 or SRC2 register.

→ When the instructions in fig 5.6 are executed in the datapath of fig 5.7 the operations performed in each clock cycle are as follows.

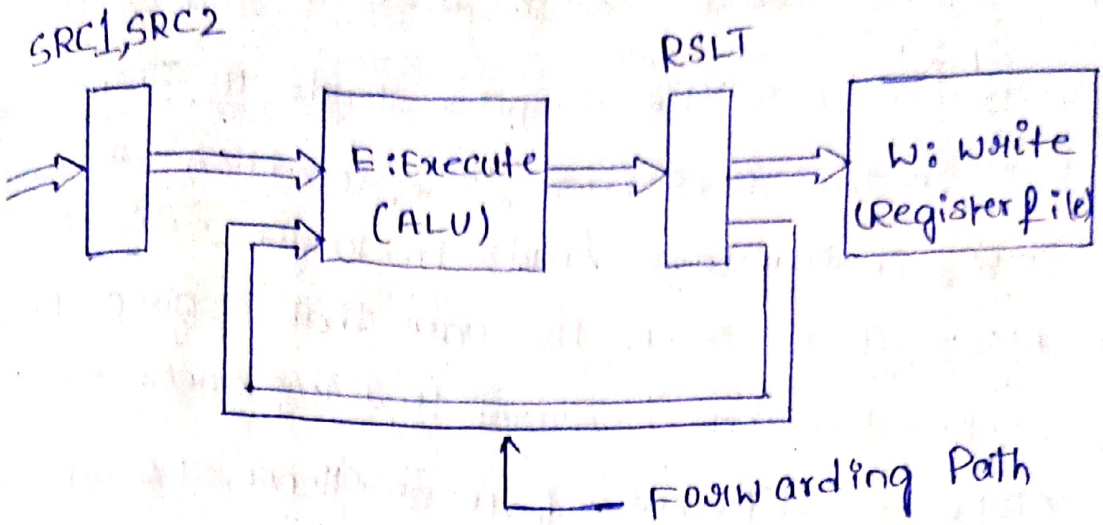
→ After decoding instruction I2 and detecting the data dependency, a decision is made to use data forwarding.

The operand not involved in the dependency, register R2, is read and loaded in register SRC1 in clock cycle 3.

In the next clock cycle, the product produced by instruction I1 is available in register RSLT, and because of the forwarding connection, it can be used in step E2. Hence, execution of I2 proceeds without interruption.



(a) Data Path



(b) Position of the source & result registers in the processor pipeline.

Fig: 5.7 : Operand forwarding in a pipelined processor.

5.2.2. Handling Data Hazards in Software

In fig 5.6, we assumed the data dependency is discovered by the hardware while the instruction is being decoded. The control hardware delays reading register R4 until cycle 5, thus introducing a 2 cycle stall unless operand forwarding is used.

An alternative approach is to leave the task of detecting data dependencies and dealing with them to the software.

In this case, the compiler can introduce the two-cycle delay needed between instructions I1 and I2 by inserting NOP (No-operation) instructions, as follows:

```
I1 : MUL R2, R3, R4
     NOP
     NOP
I2 : Add R5, R4, R6.
```

→ Leaving tasks such as inserting NOP instructions to the compiler leads to simpler hardware. But on the other hand, the insertion of NOP slots, instructions leads to larger code size. Also, it is often the case that a given processor architecture has several H.W implementations, offering different features. Hence, would lead to reduced performance on a different implementation.

5.2.3 Side Effects → An operation, fn, or exp is said to have a side effect if it modifies

Sometimes an instruction changes the contents of a register other than the one named as destination. An instruction that uses an auto increment or auto decrement addressing mode is an example. In addition to storing new data in its destination location, the instruction changes the contents of a source register used to access one of its operands. When a location other than one explicitly named in an instruction as destination operand is affected, The instruction is said to have a "side Effect".

For Ex: Stack instructions, such as Push & Pop, produce similar side Effects because they implicitly use the auto increment and auto decrement addressing modes.

Topic 3: Instruction Hazards (or) control hazard (or) (Branch hazard) :-

→ Any condition that causes a stall in the pipeline operations can be called a "hazard".

→ The instruction fetch unit of the CPU is responsible for providing a stream of instructions to the execution unit.

→ Whenever this stream is interrupted, the pipeline stalls.

→ A branch instruction may also cause the pipeline to stall. (→ that can cause a comp to execute a diff instru.)

→ We will now examine the effect of branch instructions and the techniques that can be used.

3.1 Unconditional Branches :-

→ Instruction that passes control to a different part of the program. [JUMP (or) BRANCH] (→ skip over the portion of pgm)

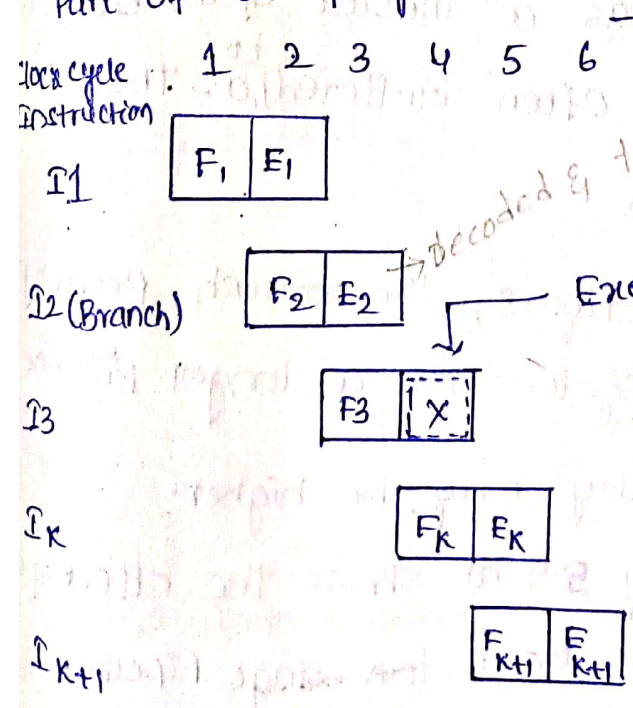
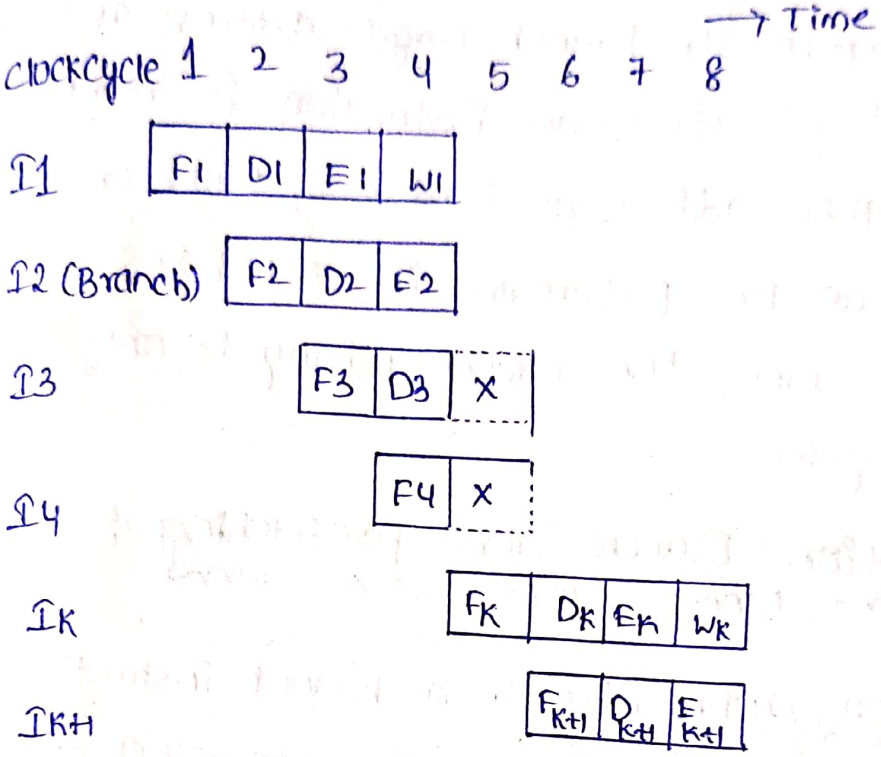


fig: 5.8 An idle cycle caused by a branch instruction

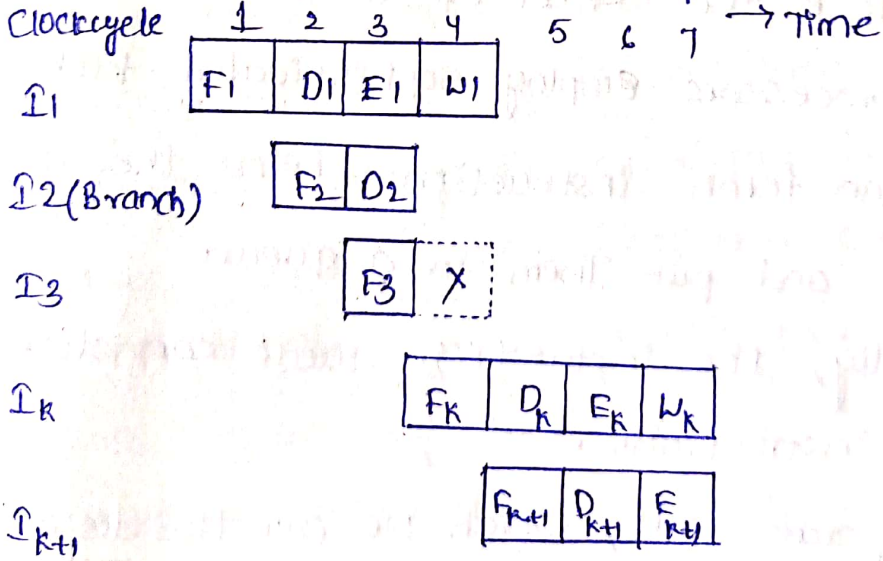
→ Above fig shows a sequence of instructions being executed in a two-stage pipeline.
→ Instructions I1 to I3 are stored at successive memory addresses, and I2 is a branch instruction.
→ Let the branch target be instruction Ik. In clock cycle 3, the fetch operation for instruction I3 is in progress at the same time that the branch instruction is being decoded and the target address is computed.
→ In clock cycle 4, the processor must discard I3, which has been incorrectly fetched, and fetch instruction Ik.
→ In the meantime, the hardware unit responsible for the Execute(E) step must be told to do nothing during that clock period.
→ Thus, the pipeline is stalled for one clock cycle.
→ The time lost as a result of a branch instruction is often referred to as the Branch penalty.
→ In the above figure, the branch penalty is one clock cycle. For a longer pipeline, the branch penalty may be higher.
→ For example, fig 5.9(a) shows the effect of a branch instruction on a four-stage pipeline.

→ We have assumed that the branch address is computed in step E2. Instructions I3 and I4 must be discarded, and the target instruction, Ik, is fetched in clock cycle 5.

→ Thus, the branch penalty is two clock cycles.



(a) Branch address computed in Execute stage



(b) Branch address computed in Decode stage

fig: 5.9 Branch Timing.

→ Reducing the branch penalty requires the branch address to be computed earlier in the pipeline.

→ Typically, the instruction fetch unit has dedicated hardware to identify a branch instruction and compute the branch target address as quickly as possible after an instruction is fetched.

→ With this additional hardware, both of these tasks can be performed in step D2.

→ In this case, the branch penalty is only one clock cycle.

Instruction Queue and prefetching :-

→ Either a cache miss or a branch instruction stalls the pipeline for one or more clock cycles.

→ To reduce the effect of these interruptions, many processors employ sophisticated fetch units that can fetch instructions before they are needed and put them in a queue.

→ Typically, the instruction queue can store several instructions.

→ A separate unit, which we call the dispatch unit, takes instructions from the front of the queue and sends them to the execution unit.

→ This leads to the organization shown in fig 5.10 This dispatch unit also performs the decoding function

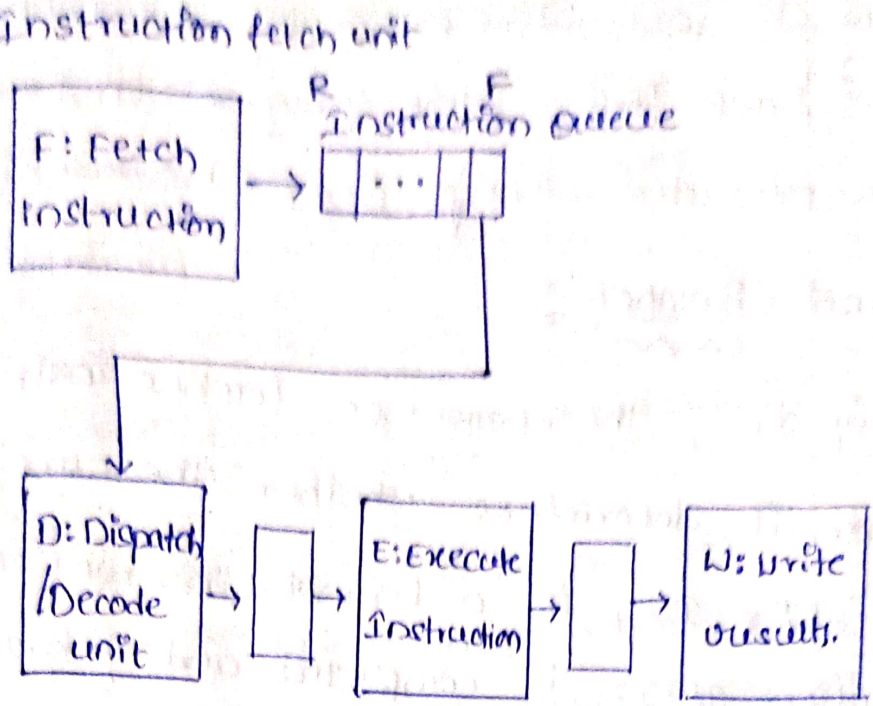


fig: 5.10 Use of an Instruction queue in the hardware organization.

3.2 Conditional Branches and Branch Prediction

→ A conditional branch instruction branches to a new address only if a certain condition is true. Usually the condition is about the values in two registers. (if then else, while)

→ conditional branch, which may or may not cause branching, depending on some condition. ↳ creating copies of pgms.

→ Branch instructions are used to implement control flow in program loops and conditionals. (Executing a particular sequence of instructions only if certain condition is True).

→ Branch instructions occur frequently. In fact that some program instructions are executed many times because of loops.

Delayed Branch :-

In fig 5.8, the processor fetches instruction I3 before it determines whether the current instruction, I2, is a branch instruction. When execution of I2 is completed and a branch is to be made, the processor must discard I3 and fetch the instruction at the branch target.

→ The location following a branch instruction is called a "branch delay slot"

→ There may be more than one branch delay slot depending on the time it takes to execute a branch instruction.

→ For example, there are two branch delay slots in fig 5.9(a) and one delay slot in fig 5.9(b)

→ A technique called "delayed branching" can minimize the penalty incurred as a result of conditional branch instructions.

→ The idea is simple, the instructions in the delay slots are always fetched. Therefore, we would like to arrange for them to be fully executed whether or not the branch is taken.

→ consider the instruction sequence given in fig. 5.12(a). Register R2 is used as a counter to determine the number of times the contents of register R1 are shifted left.

→ For a Processor with one delay slot, the instructions can be reordered as shown in fig 5.12(b).

→ The shift instruction is fetched while the branch instruction is being executed. After evaluating the branch condition, the processor fetches the instruction at LOOP or at NEXT, depending on whether the branch condition is true or false.

LOOP	Shift-left	R1	counter
	Decrement	R2	
	BRANCH=0	LOOP	
NEXT	Add	R1,R3	

(a) Original program loop

LOOP	Decrement	R2
	Branch=0	LOOP
	Shift-left	R1
NEXT	Add	R1,R3

(b) Reordered instructions

5.12 Reording of instructions for a delayed branch

→ The Effectiveness of the delayed branch approach depends on how often it is possible to reorder instructions as shown in above fig 5.12.

→ Thus, if increasing the no.of pipeline stages involves an increase in the no.of branch delay slots, the potential gain in performance may not be fully realized.

Branch Prediction:-

→ Another technique for reducing the branch penalty associated with conditional branches is to attempt to predict whether or not a particular branch will be taken.

Various techniques used to predict whether a branch will be taken. They are

- → Predict Never Taken
- → Predict Always Taken → Static
- → Predict by Opcode
- → Taken / Not Taken Switch
- → Branch History Table → Dynamic

(Static - 123)

→ They do not depend on the execution history.

1. Predict Never Taken:-

Always assume that the branch will not be taken and continue to fetch instruction in sequence

2. Predict Always Taken:-

Always assume that the branch will be taken and always fetch from target.

3. predict by opcode:-

Decision based on the opcode of the branch instruction. The processor assumes that the branch will be taken for certain branch opcodes and not for others.

Dynamic (4,5) - They depend on the execution history.

→ They attempt to improve the accuracy of prediction by recording the history of conditional branch instructions in a program.

4. Influence on Instruction sets.

→ Some instructions are much better suited to Pipeline execution than others.

→ Addressing modes

→ condition code flags

Addressing Modes :-

→ Addressing modes include simple ones and complex ones.

→ In choosing the addressing modes to be implemented in pipelined processor, we must consider the effect of each addressing mode on instruction flow in the pipeline!

*Side effects

→ The extent to which complex addressing modes cause the pipeline to stall.

→ Whether a given mode is likely to be used by compilers. (whether the particular mode is using in compiler or not)

→ To compare various approaches, we assume a simple model for accessing Operands in the memory.

→ The load instruction Load X(R1), R2 takes 5 cycles to complete execution, as indicated in fig: 5.5, However, The instruction

For ex: Load (R1), R2

Load (X(R1)), R2

Load X(R1), R2 → constant + R1

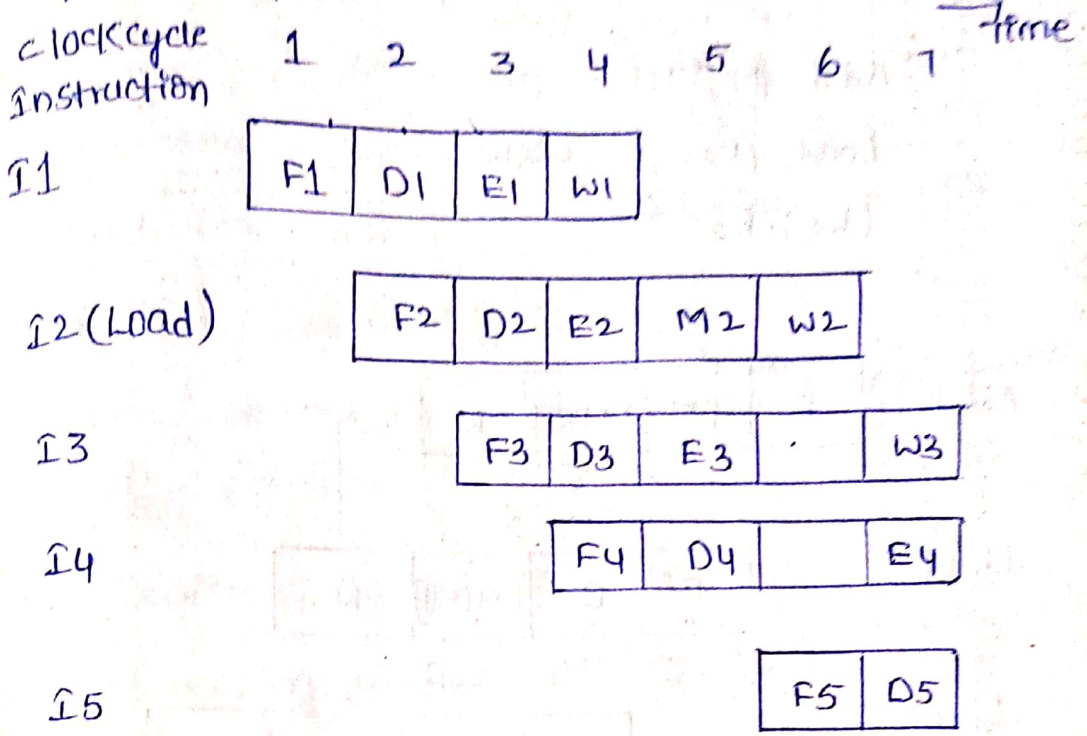
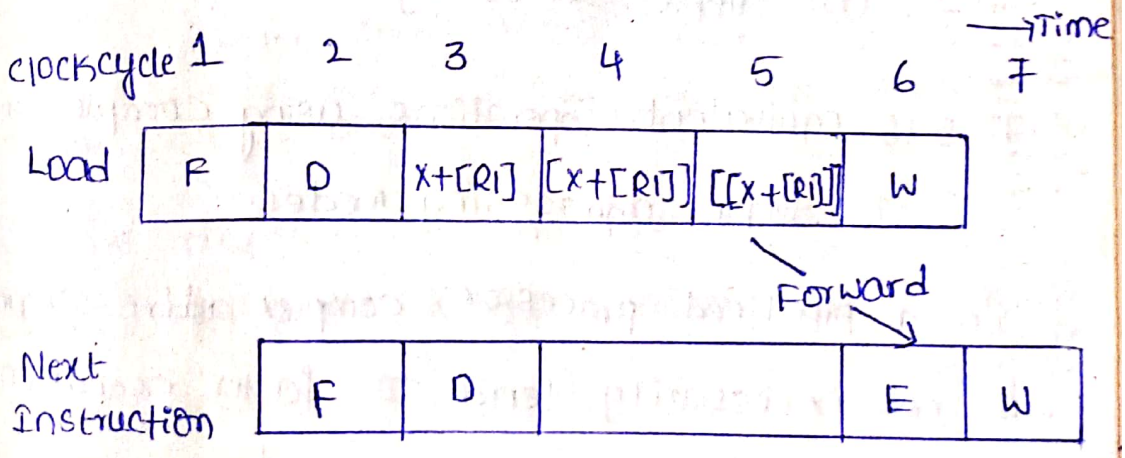


fig: 5.5 Effect of a Load instruction on Pipeline timing.

Complex Addressing Mode:-

Load (X(R1)), R2 → TWO braces so double indexing
↳ X+R1 → R2



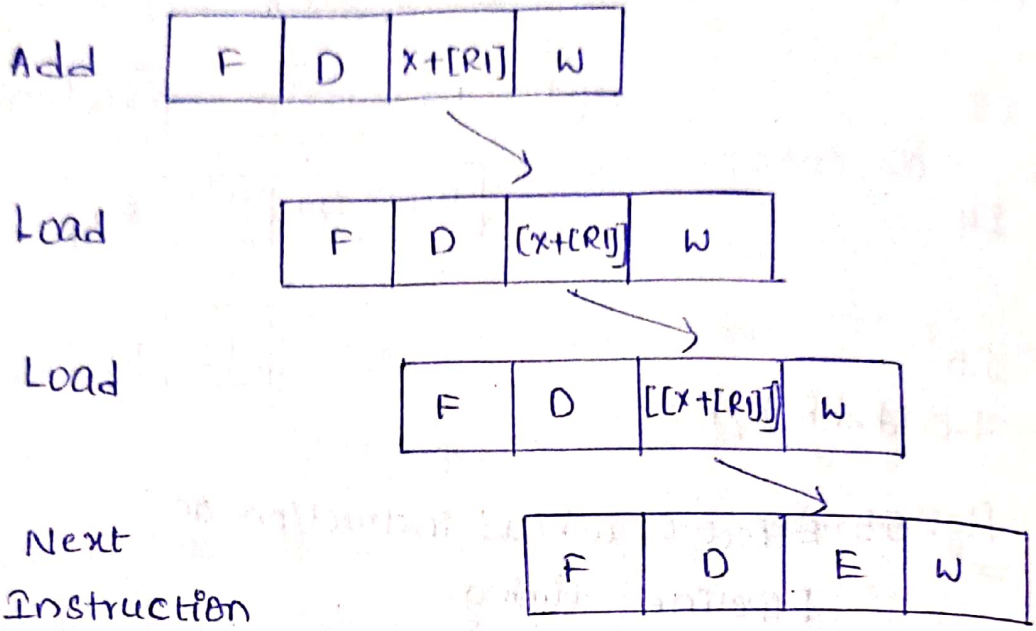
→ With the help of forwarding we can be able to reduce the delays.
→ Multiple Memory access at a time.
(a) complex Addressing mode

Simple Addressing Mode :-

Add #X, R1, R2

Load (R2), R2 Load

(R2), R2



→ For each instruction there is a separate memory access.

(b) Simple addressing mode

fig: 5.16 Equivalent operations using complex and simple addressing modes.

→ In a pipelined processor, complex addressing modes do not necessarily lead to faster execution.

Adv: Reducing the no. of instructions / program space

Disad: cause pipeline to stall / more H.W to decode / and it is not convenient for compiler to work with.

Conclusion: complex addressing modes are not suitable for Pipelined execution.

If Good addressing modes should have :-

1. Access to an operand does not require more than one access to the memory.
2. Only load and store instructions access memory operands.
3. The addressing modes used do not have side Effects.
4. Register, register in indirect, index addressing modes are good addressing modes.

conditional codes :-

→ If an optimizing compiler (↳ we use O.C in this C.C) attempts to reorder instruction to avoid stalling the pipeline when branches or data dependencies between successive instructions occur, it must ensure that reordering does not cause a change in the outcome of a computation.

→ The dependency introduced by the condition-code flags reduces the flexibility available for the compiler to reorder instructions.

→ Means using dependency instructions in condition code flags, it reduces the flexibility while reorder instructions.

Add	R1, R2
compare	R3, R4
Branch=0	----

→ After execution of two stms then it will perform compare the two stmts so it takes more cycles.

(a) A program fragment

compare	R3, R4
Add	R1, R2
Branch=0	----

→ Reduce stalling by reordering stmts but the out-come is same

(b) Instruction reordered

fig: 5.17 Instruction reordering

→ TWO conclusion:-

→ To provide flexibility in reordering instructions, the condition-code flags should be affected by as few instructions as possible. (with few instructions the condition code flags will be affected).

→ The compiler should be able to specify in which instructions of a program the condition codes are affected and in which they are not.

↳ Means the compiler should specify that which kind of instructions are affected by condition code flags and which kind of instructions the condition code flags are not affecting.

Topic 5: Forms of Parallel Processing

→ P.P system is able to performing concurrent data processing to acheive faster execution time

→ Parallel processing is a term used to denote a large class of techniques that are used to provide simultaneous data-processing tasks for the purpose of increasing the computational speed of computer system.

→ The purpose of parallel processing is to speed up the computer processing capability and increase its, throughput, that is, the amount of processing that can be accomplished during a given interval of time.

→ The amount of hardware increases with Parallel processing, and with it, the cost of the system increases.

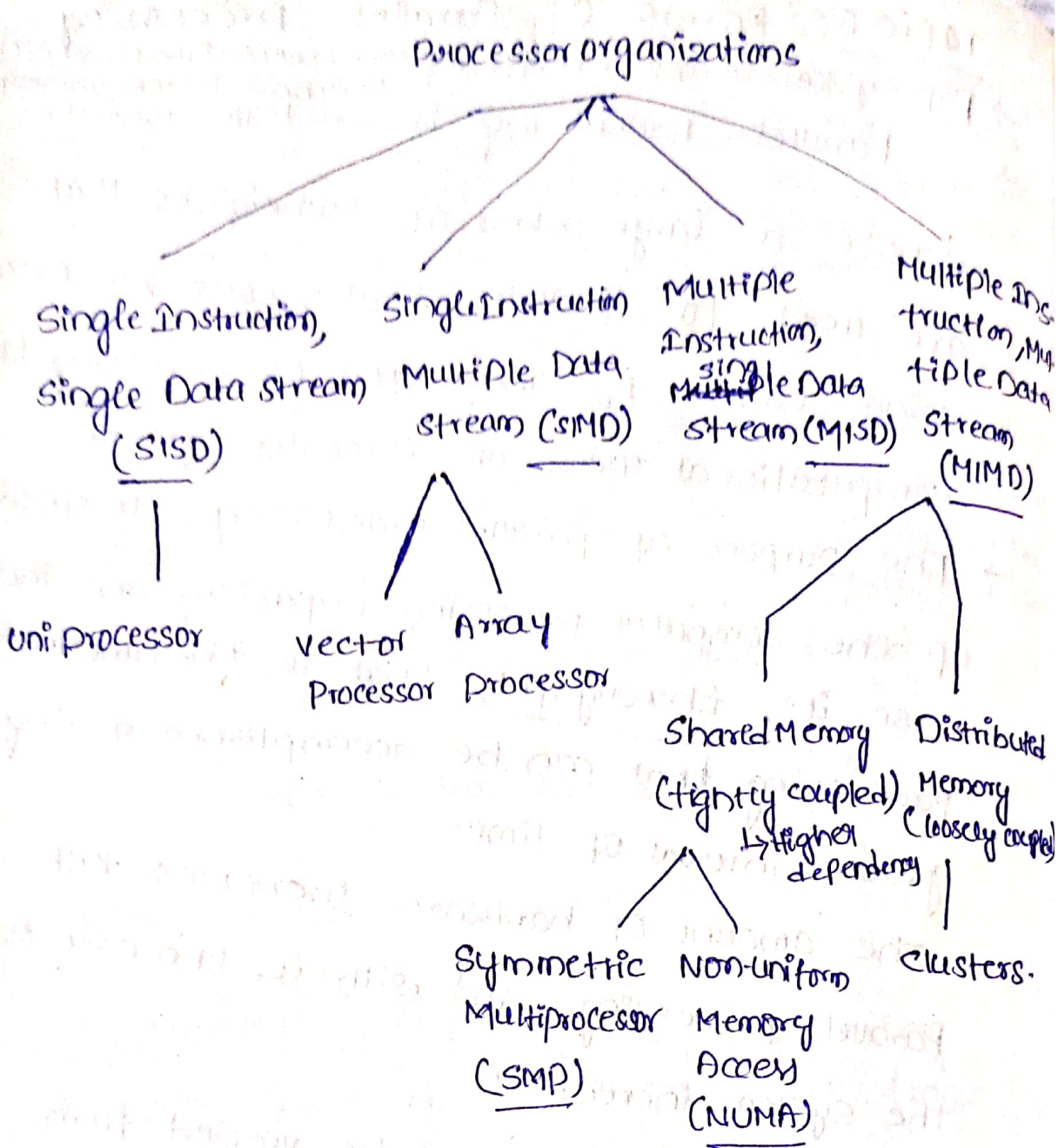
→ parallel processing can be viewed from various levels of complexity

→ There are a variety of ways that parallel processing can be classified.

1. Internal organization of the processors.
2. Interconnection structure between processors

Instruction Stream :- The process of fetching the instruction from Memory.

Data stream :- In order to execute the instruction, that instruction must be in the processor.



→ The flow of information through the system

→ M.J. Flynn considers the organization of a computer system by the no. of instructions and data items that are manipulated simultaneously.

* single instruction stream, single data stream (SISD)
* Single instruction stream, multiple data stream (SIMD)
* Multiple instruction stream, single Data stream (MISD)
* Multiple Instruction stream, Multiple Data stream (MIMD)

SISD :-

Represents the organization of a single computer containing a control unit, a processor unit, and a memory unit. Instructions are executed sequentially and the system may or may not have internal parallel processing capabilities. parallel processing may be achieved by means of multiple functional units or by pipeline processing.

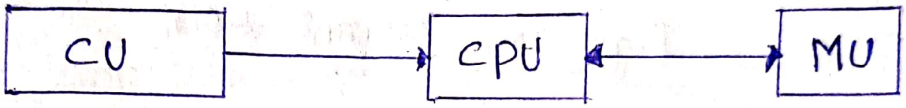


fig: SISD Organization

SIMD :-

Represents an organization that includes many processing units under the supervision of common control unit. All processors receive the same instruction from the control unit but operate on different items of data. The shared memory unit must contain multiple modules so that it can be communicate with all the processors simultaneously.

PE- Processing Element
LM- Local Module
IS - Instruction Stream
DS - Data stream

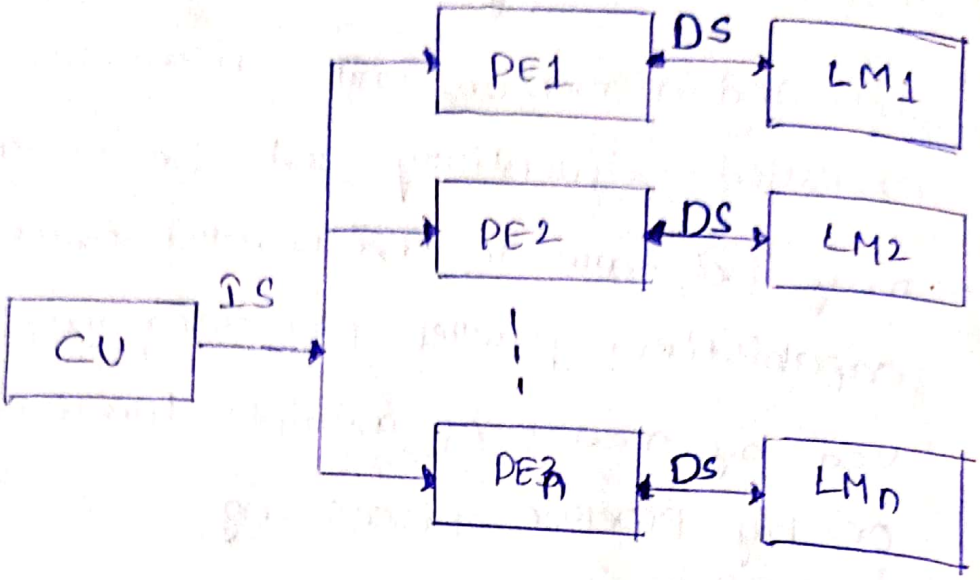


fig: SIMD Organization

MISD & MIMD :-

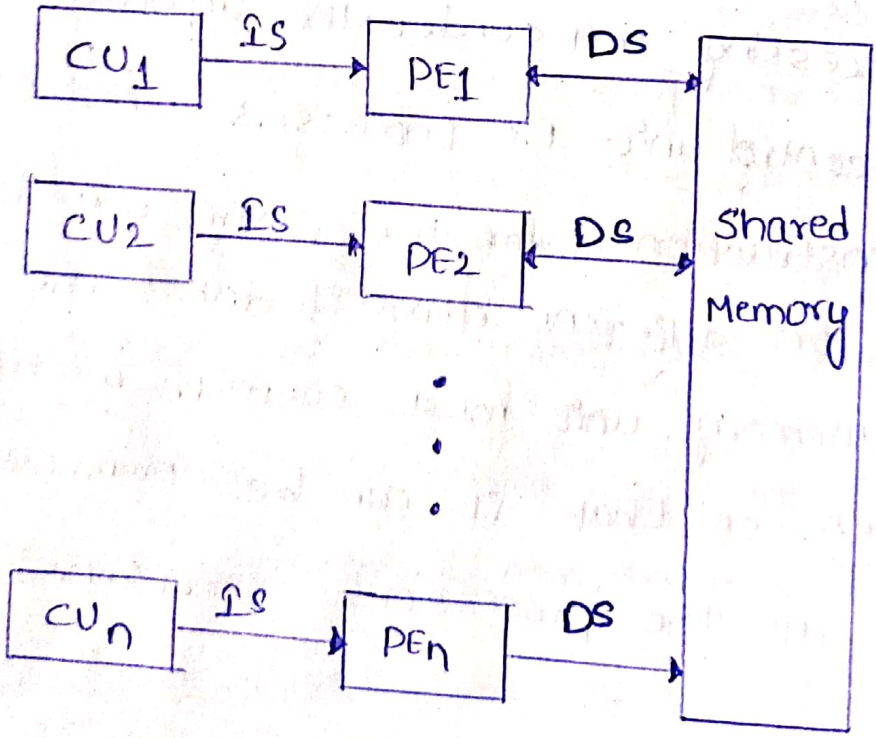


fig: MIMD organization

→ MISD structure is only of theoretical interest since no practical system has been constructed using this organization.

→ MIMD Organization refers to computer system capable of processing several programs at the same time. Eg: Multiprocessor and Multicomputer sys.
↳ many comp ↳ each comp
↳ single sys with multiple CPUs can have multiple processors

→ Flynn's classification depends on the distinction between the performance of the control unit and the data-processing unit.

→ It emphasizes the behavioral characteristics of the computer system rather than its operational and structural interconnections.

→ one type of parallel processing that does not fit Flynn's classification is pipelining.

TOPIC 6: Array Processing

An array processor is a processor that performs computations on large arrays of data. The term is used to refer to two different types of processors.

1. Attached array processor :-
→ operation performed by offline equipment
Is an auxiliary processor. It is intended to improve the performance of the host computer in specific numerical computation tasks.

2. SIMD array processor:-

SIMD has a single-instruction multiple-data organization. It manipulates vector instructions by means of multiple functional units responding to a common instruction.

→ Attached Array processor its purpose is to enhance the performance of the computer by providing vector processing for complex scientific applications.

Parallel processing with multiple functional units:

Fig: illustrates structure of array processor.

A 2-Dimensional array processor with a grid of processing elements executes an instruction stream that is going to be "broadcast" (elements executed simultaneously) from central processor.

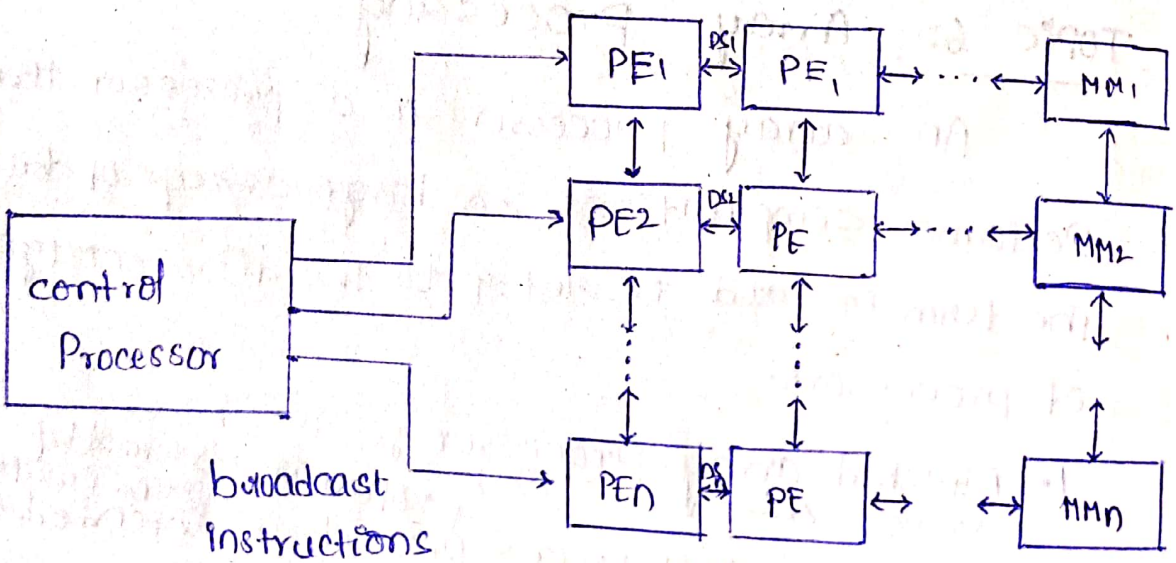


fig: An array of processor.

→ Processing elements are connected to its nearest neighbors for the purpose of exchanging data.

→ To perform calculations quite simple.

→ Each element must be able to exchange values with each of its neighbor on the paths.

→ control processor broadcast an instruction to shift the values in network registers 1 set, up, down, right or left.

→ control processor able to determine when each of processing elements has developed its component of temparature to required accuracy. Therefore each element set on status bit to 1.

→ TO implement loop, sequence of instructions can be broadcast repeatedly.

→ Grid interconnections have a facility that allow controller to detect when all status bits are send to end of iteration.

Eg: ILLIAC-IV, CM-2, MP-1216, GAMMA II plus.

→ Array processors are highly specialized machines. They are well-suited to numerical Problems expressed in matrix/vector formats but do not have commercial market.

Characteristics of multiprocessors :-

→ A multiprocessor system is an interconnection of two or more CPUs with memory and input-output equipment.

→ The term "processor" in multiprocessor can mean either a central processing unit (CPU) or an input-output processor (IOP).

→ Multiprocessors are classified as multiple instruction stream, multiple data stream (MIMD) systems.

→ The similarity between multiprocessor and multicomputer are both support concurrent operations.

→ Multiprocessing improves the reliability of the system. Multiple independent jobs can be made to operate in parallel.

→ A single job can be partitioned into multiple Paraael tasks.

→ Multiprocessing can improve performance by decomposing a program into parallel executable tasks.

→ Multi-processor's are classified by the way their memory is organized.

→ A multiprocessor system with common shared memory is classified as shared memory or tightly coupled multiprocessor.

→ Each processor element with its own private local memory is classified as a distributed-memory or loosely coupled system.

TOPIC:7:- The Structure Of General-purpose Multi-Processors

1. UMA (Uniform Memory Access) Multiprocessor

→ An interconnection-network permits n processors to access k memories Thus, any of the processors can access any of the memories.

→ The interconnection-network may introduce network-delay between Processor & Memory.

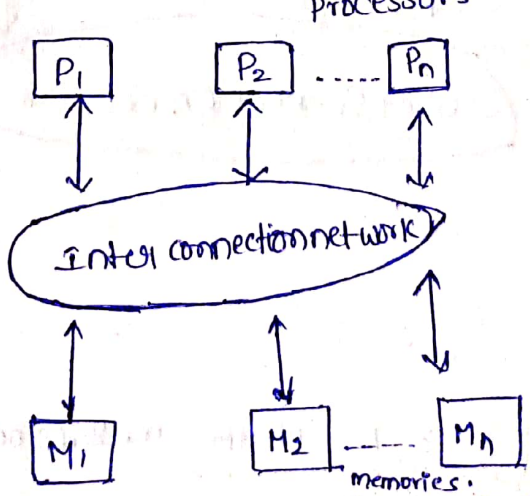
→ A system which has the same network-latency (Time it takes for data) for all accesses from the processors to the memory-modules is called a "UMA Multiprocessor."

→ Although the latency is uniform, it may be large for a network that connects many processors & many memory modules.

→ For a better performance, it is desirable to place a memory-module close to each processor.

Disadvantage:

→ Interconnection-networks with very short delays are costly and complex to implement.



19.9 fig: A UMA Multiprocessor

2. NUMA (Non-Uniform Memory Access) Multiprocessors

→ Memory-modules are attached directly to the processors. (fig. 12.3)

→ The network-latency is avoided when a processor makes a request to access its local memory.

→ However, a request to access a remote-memory-module must pass through the network.

→ Because of the difference in latencies for accessing local and remote portions of the shared memory, systems of this type are called NUMA multiprocessors.

Advantage:

A high computation rate is achieved in all processors.

Disadvantage:

The remote accesses take considerably longer than accesses to the local memory.

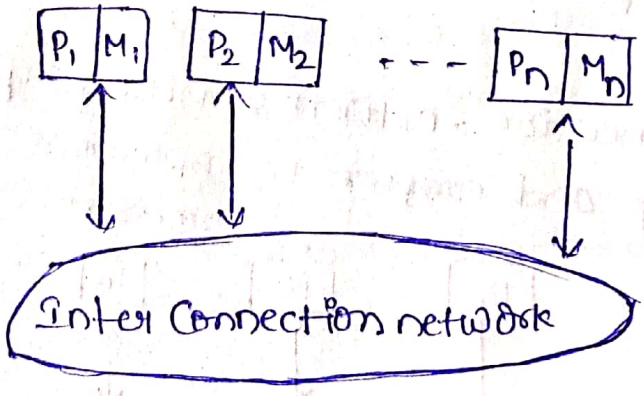


fig: 12.3 A NUMA multiprocessor.

3. Distributed Memory Systems

All memory modules serve as private memories for processors that are directly connected to them. A processor can not access a remote-memory with out the operation of the remote processor. This cooperation takes place in the form of messages exchanged by the processors. Such systems are called DMS.

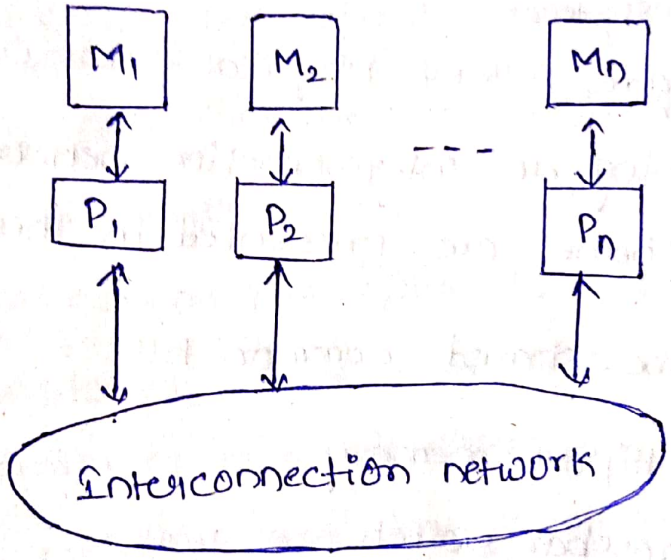


fig: 12.4 A distributed memory system

Topic 8: Interconnection Networks

→ The components that form a multiprocessor system are CPUs, IOPs connected to input-output devices, and a memory unit that may be partitioned into a number of separate modules.

→ The interconnection between the components can have different physical configurations, depending on the number of transfer paths that are available between the processors and memory in a shared memory system.

→ There are several physical forms available for establishing an interconnection network. Some of these schemes are presented in this section:

1. Time-shared common bus
2. Multiport memory
3. Crossbar switch
4. Multistage switching
5. Hypercube system.

1. Time-shared common bus

→ A common-bus multiprocessor system consists of a no. of processors connected through a common path to a memory unit.

→ A time-shared common bus for five processors is shown in fig. 13.1

→ Only one processor can communicate with the memory or another processor at any given time.

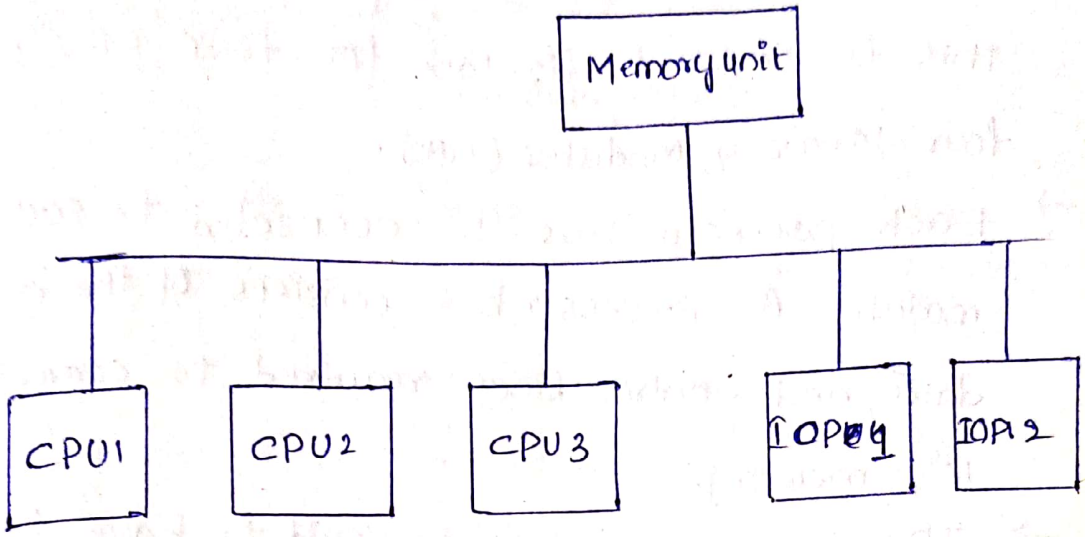


fig: 13.1 Time shared common bus organization.

→ Transfer operations are conducted by the processor that is in control of the bus at the time.

→ A single common-bus system is restricted to one transfer at a time. This means that when one processor is communicating with the memory, all other processors are either busy with internal operations or must be idle waiting for the bus.

→ As a consequence, the total overall transfer rate within the system can be kept busy more often through the implementation of two or more independent buses to permit multiple simultaneous bus transfers.

→ However, this increases the system cost and complexity.

2. Multiport Memory

→ A multiport memory system employs separate buses between each memory module and each CPU. This is shown in fig. 13.3 for four CPUs and four Memory Modules (MMs).

→ Each processor bus is connected to each memory module. A processor bus consists of the address, data, and control lines required to communicate with memory.

→ The memory module is said to have four ports and each port accomodates one of the buses.

→ The module must have internal control logic to determine which port will have access to memory at any given time.

→ Memory access conflicts are resolved by assigning fixed priorities to each memory port.

→ The priority for memory access associated with each processor may be established by the physical port position that its bus occupies in each module.

→ Thus CPU1 will have priority over CPU 2, CPU 2 will have priority over CPU3, and CPU 4 will have the lowest priority.

→ The advantage of the multiport memory organization is the high transfer rate that can be acheived because of the multiple paths between

processors and memory.

→ The disadvantage is that it requires expensive memory contrd logic and a large no. of cables and connectors.

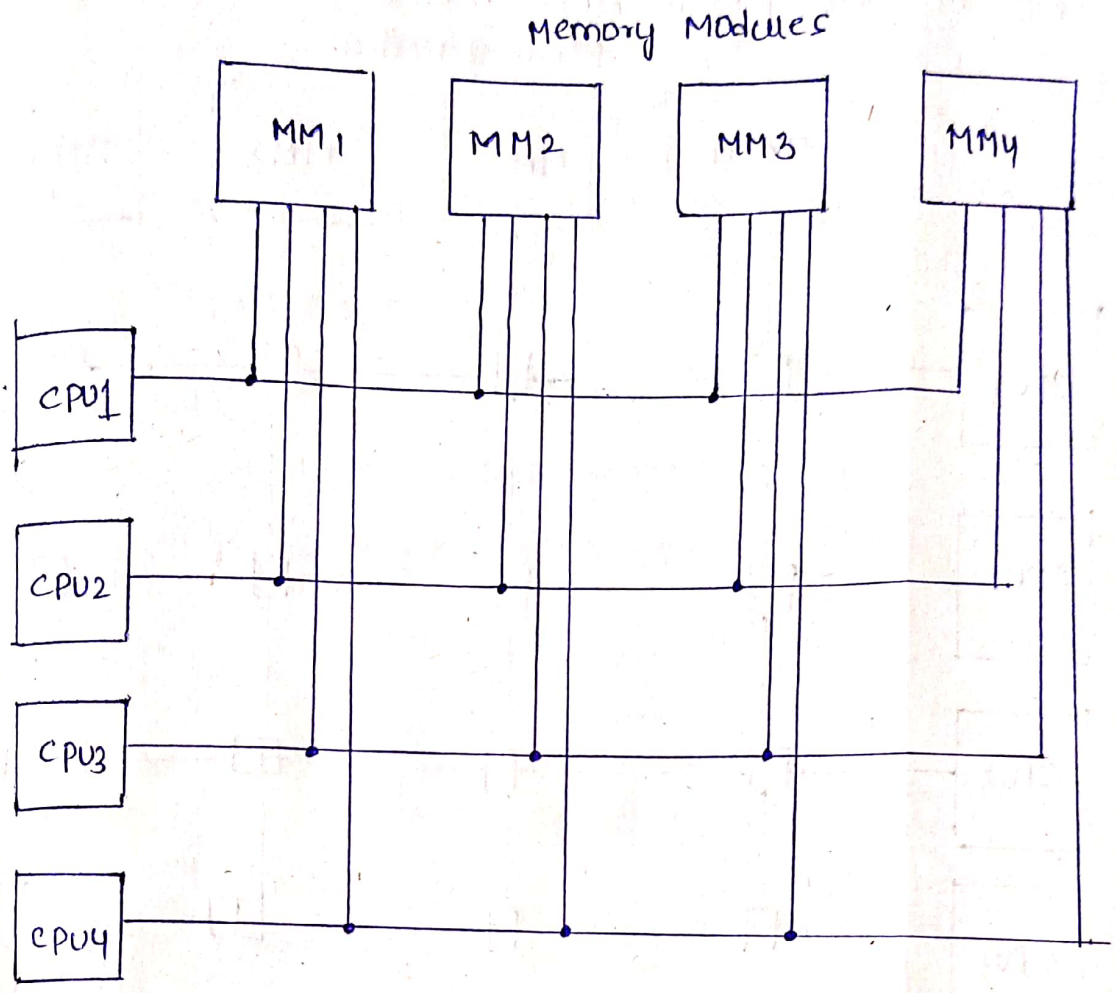


fig: 13.3 Multiport Memory Organization.

3. Crossbar switch :-

→ The crossbar switch organization consists of a no. of crosspoints that are placed at intersections between processor buses and memory module paths. Frg: 13.4 shows a crossbar switch interconnection b/w four cpu's and four memory modules. The small square in each cross point is a switch that determines the path from a processor to a memory module.

→ Each switch point determines the path from a Processor to a memory module. Each switch point has control logic to set up the transfer path blw a processor and memory.

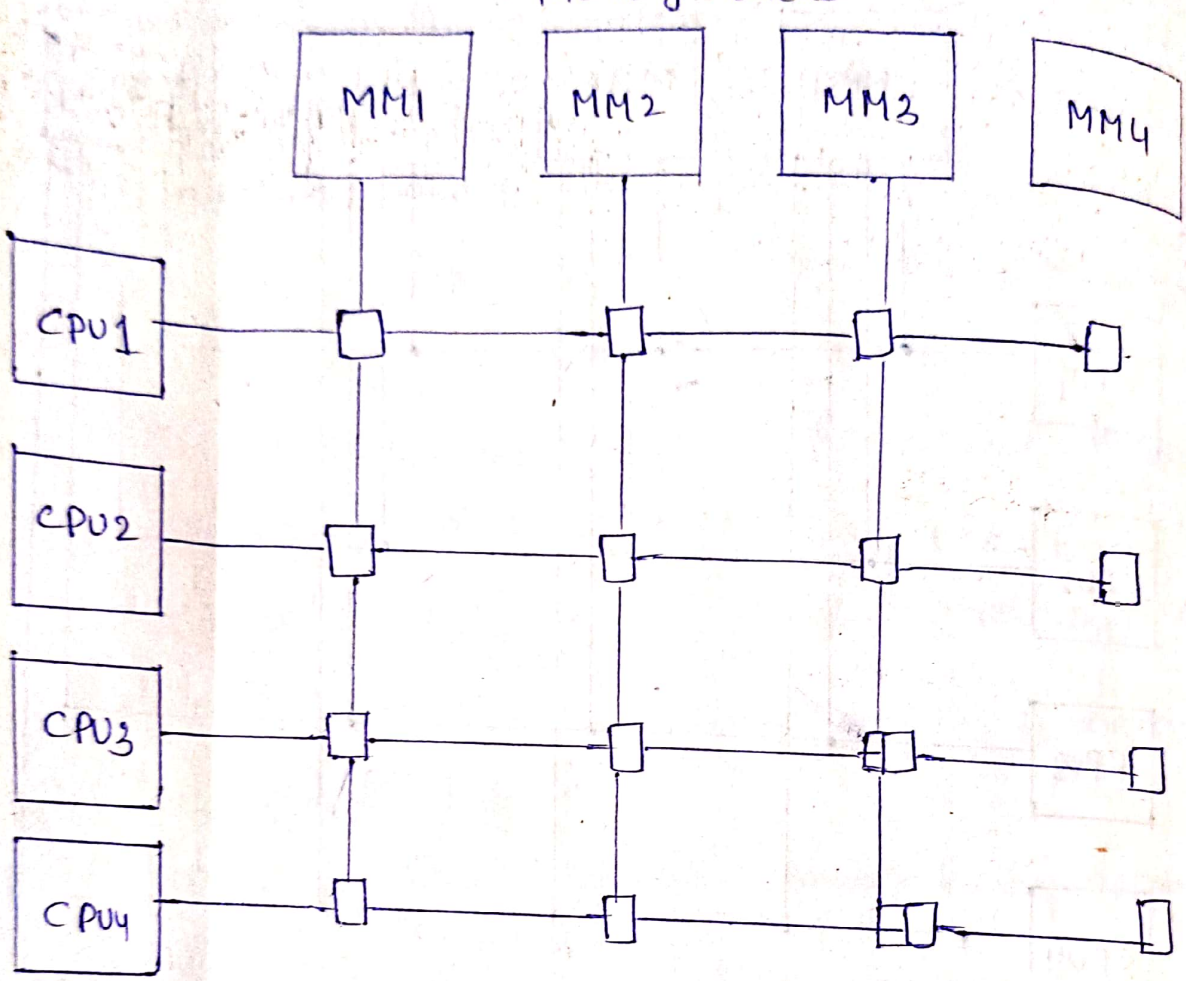


fig: 13.4 cross bar switch

→

→ A crossbar switch organization supports simultaneous transfers from memory modules because there is a separate path associated with each module. However, the hardware required to implement the switch can be quite large and complex.

4. Multistage Switching Network

→ The basic component of a multistage network is a two-input, two-output interchange switch. As shown in fig: 13.6.

→ The 2x2 switch has two input labeled A and B, and two outputs, labeled 0 and 1.

→ There are control signal (not shown) associated with the switch that establish the interconnection between the input and output terminals.

→ The switch has the capability connecting input A to either of the outputs.

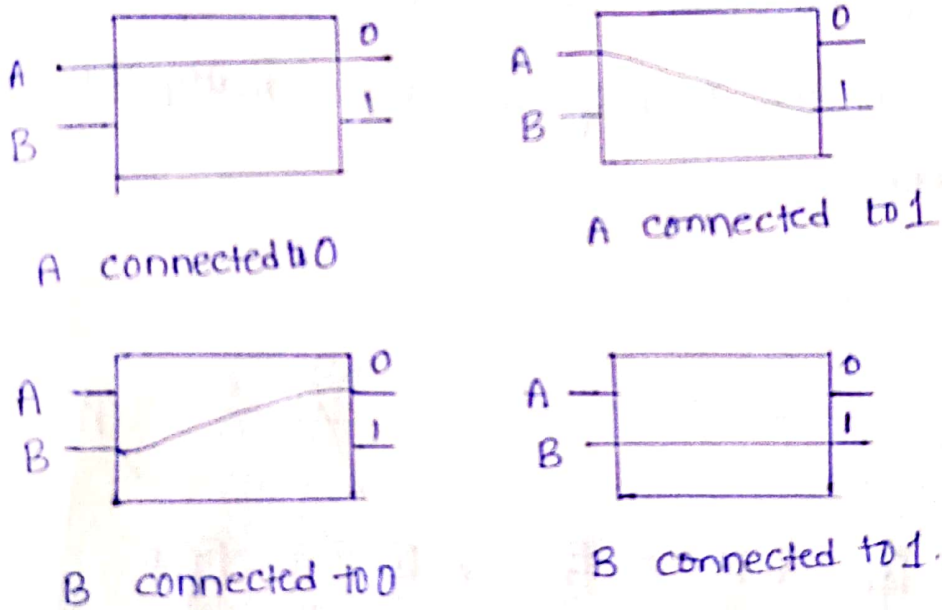


fig: 13.6 Operation of a 2x2 interchange switch

5. Hypercube Interconnection

The hypercube or binary n-cube multiprocessor structure is loosely coupled system composed of $N = 2^n$ processors interconnected in an n-dim-

ensional binary cube.

→ Each processor forms a node of the cube. Although it is customary to refer to each node as having a processor, in effect it contains not only a CPU but also local memory and I/O interface.

→ Each processor has direct communication paths to n other neighbor processors. These paths correspond to the edges of the cube.

→ There are 2^n distinct n-bit binary addresses that can be assigned to the processors.

→ Each processor address differs from that of each of its n neighbors by exactly one bit position.

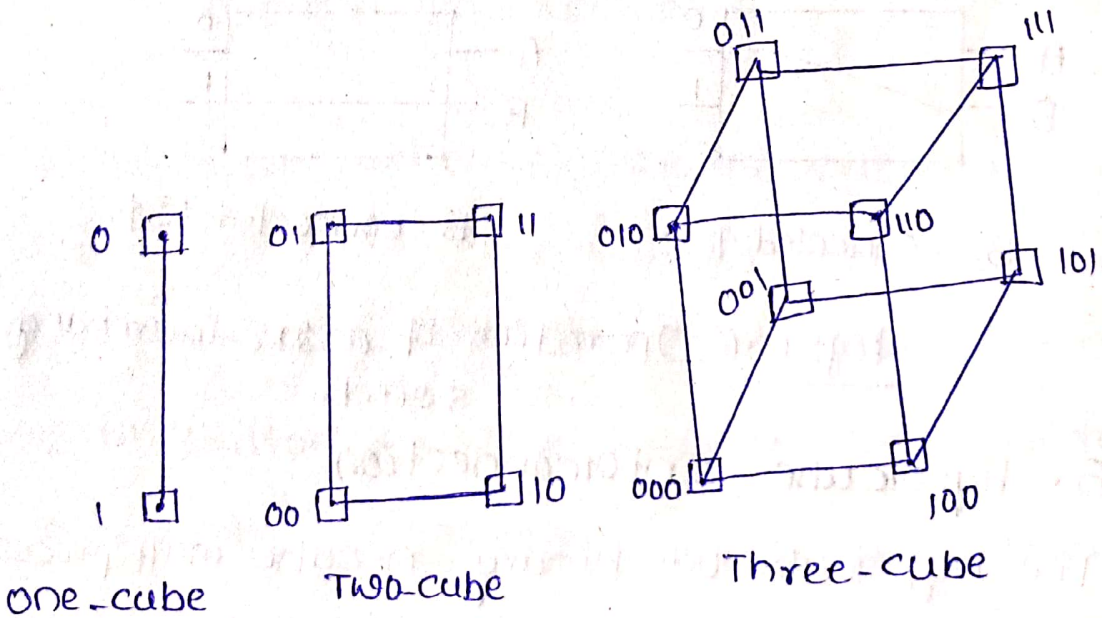


fig: 13.9 Hypercube structure for n = 1, 2, 3 dimensions.