UNIT-5

Pipelining and Parallel Processors

Basic Concepts of Pipelining :

Introduction:

1. Pipelining is a technique of decomposing a sequential process into suboperations, with each subprocess being executed in a special dedicated segment that operates concurrently with all other segments.
2. A pipeline can be visualized as a collection of processing segments through which binary information flows.
3. Each segment performs partial processing dictated by the way the task is partitioned.
4. The result obtained from the computation in each segment is transferred to the next segment in the pipeline. The final result is obtained after the data have passed through all segments.
5. It is characteristic of pipelines that several computations can be in progress in distinct segments at the same time. The overlapping of computation is made possible by associating a register with each segment in the pipeline. The registers provide isolation between each segment so that each can operate on distinct data simultaneously.

Pipeline organization is demonstrated by means of a simple example.

Suppose that we want to perform the combined multiply and add operations with a stream of numbers. Ai* Bi + Ci for i = 1, 2, 3, . . . , 7 Each suboperation is to be implemented in a segment within a pipeline. Each segment has one or two registers and a combinational circuit as shown in Fig below:

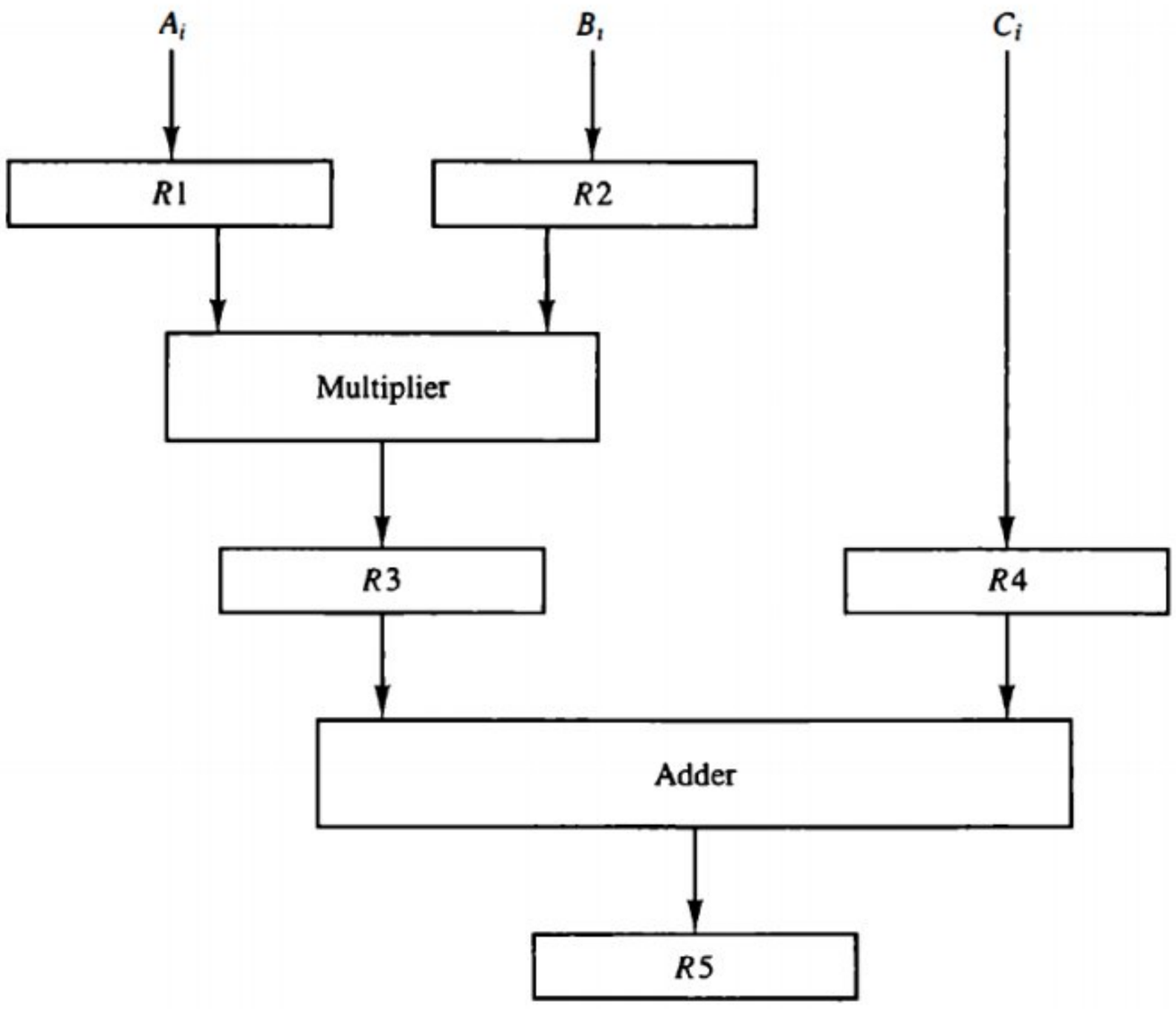
The suboperations performed in each segment of the pipeline are as follows:

R 1 <--Ai, R2 <--Bi

R3 <--R 1 * R2, R4 <--C,

R5 <--R3 + R4

- Input A, and B,
- Multiply and input C,
- Add C; to product



Example of pipeline processing.

Clock Pulse Number	Segment 1		Segment 2		Segment 3
	$R1$	$R2$	$R3$	$R4$	$R5$
1	A_1	B_1	—	—	—
2	A_2	B_2	$A_1 * B_1$	C_1	—
3	A_3	B_3	$A_2 * B_2$	C_2	$A_1 * B_1 + C_1$
4	A_4	B_4	$A_3 * B_3$	C_3	$A_2 * B_2 + C_2$
5	A_5	B_5	$A_4 * B_4$	C_4	$A_3 * B_3 + C_3$
6	A_6	B_6	$A_5 * B_5$	C_5	$A_4 * B_4 + C_4$
7	A_7	B_7	$A_6 * B_6$	C_6	$A_5 * B_5 + C_5$
8	—	—	$A_7 * B_7$	C_7	$A_6 * B_6 + C_6$
9	—	—	—	—	$A_7 * B_7 + C_7$

Table: Contents of Registers in pipeline

The five registers are loaded with new data every clock pulse. The first clock pulse transfers A1 and B1 into R 1 and R2. The second dock pulse transfers the product of R 1 and R2 into

R3 and C1 into R4. The same clock pulse transfers A2 and B2 into R 1 and R2. The third clock pulse operates on all three segments simultaneously. It places A, and B, into R1 and R2, transfers the product of R1 and R2 into R3, transfers C, into R4, and places the sum of R3 and R4 into RS. It takes three clock pulses to fill up the pipe and retrieve the first output from RS. From there on, each dock produces a new output and moves the data one step down the pipeline. This happens as long as new input data flow into the system. When no more input data are available, the clock must continue until the last output emerges out of the pipeline.

Instruction Pipelining:

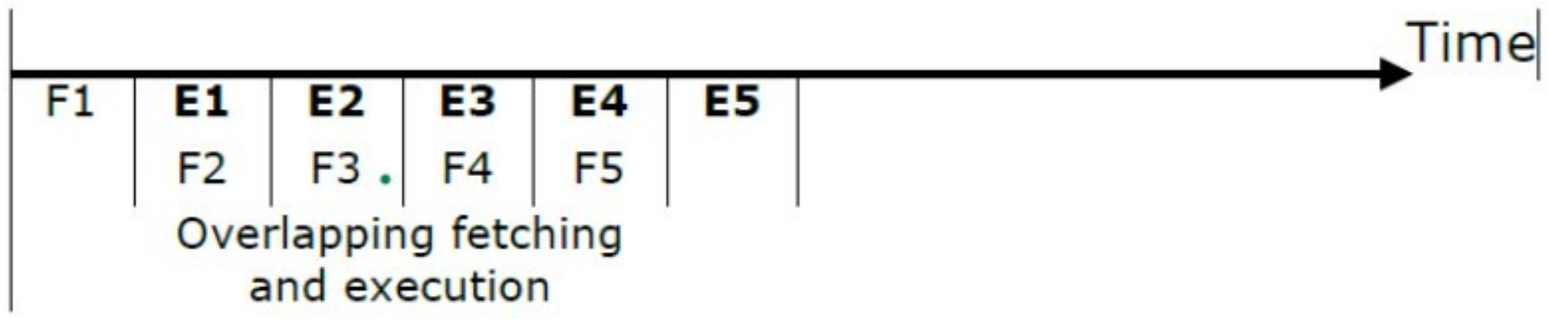
Instruction Pipelining is an implementation technique in which ***multiple instructions are overlapped in execution.*** An instruction requires several steps which mainly involve fetching, decoding and execution.

If these steps are performed one after the other, they will take a long time.

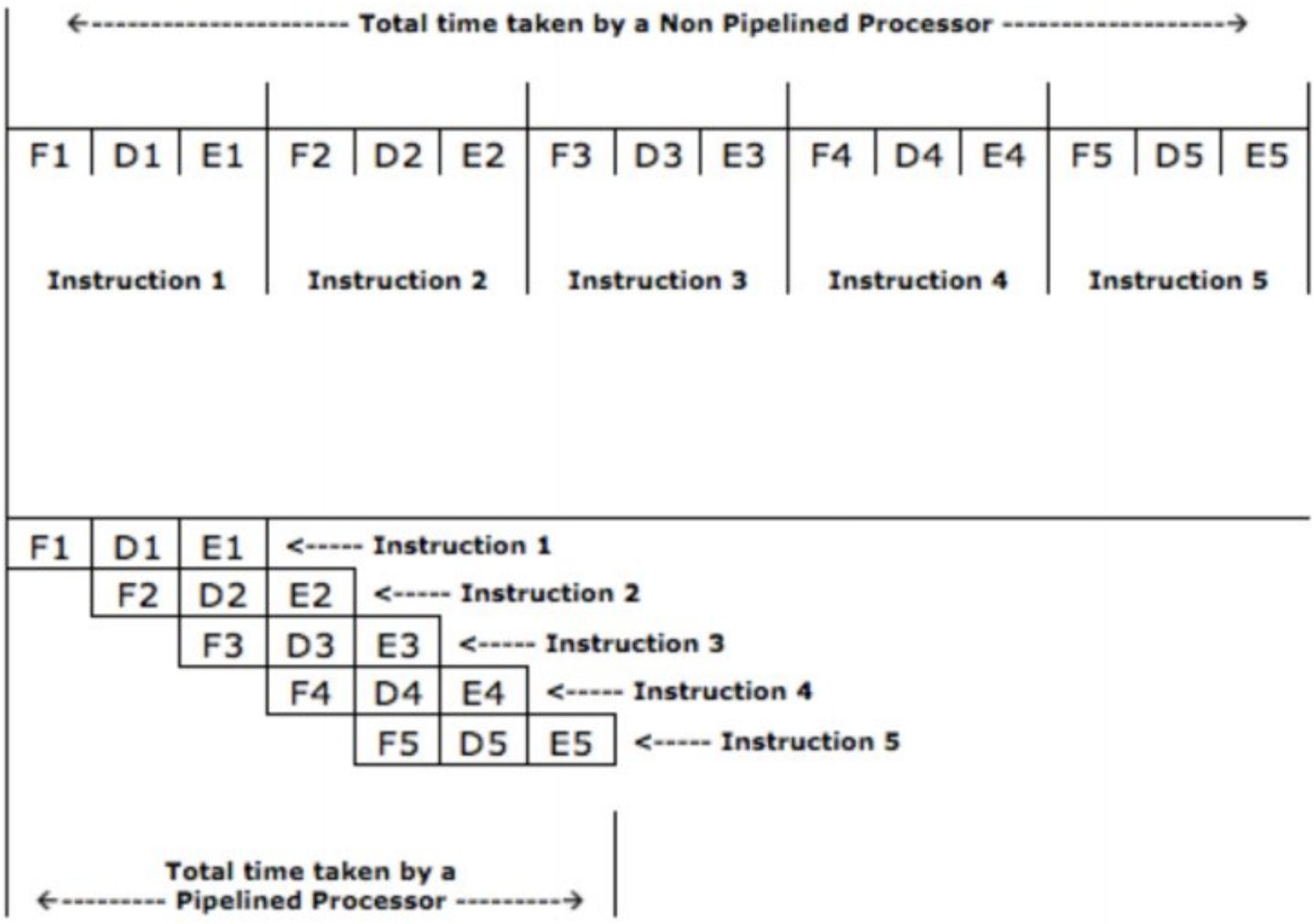
As processors became faster, several of these steps started to get overlapped, resulting in faster processing. This is done by a mechanism called pipelining.

2 STAGE PIPELINING - 8086

Here the instruction process in divided into two stages of fetching and execution. Fetching of the next instruction takes place while the current instruction is being executed. Hence two instructions are being processed at any point of time.



3 STAGE PIPELINING –ARM 7



Consider the case where a k-segment pipeline with a clock cycle time t_p ,is used to execute n tasks. The first task T1 requires a time equal to Kt_p, to complete its operation since there are k segments in the pipe. The remaining n - 1 tasks emerge from the pipe at the rate of one task per clock cycle and they will be completed after a time equal to (n - 1) tp . Therefore, to complete n tasks using a k-segment pipeline requires k + (n - 1) clock cycles.

Next consider a non-pipeline unit that performs the same operation and
takes a time equal to t_n. to complete each task. The total time required for n tasks is nt_n. The speedup of a pipeline processing over an equivalent non-pipeline processing is defined by the ratio

$$S = \frac{nt_n}{(k + n - 1)t_p}$$

ADVANTAGE OF PIPELINING

The advantage of pipelining is that it increases the performance. As shown by the various examples above, deeper the pipelining, more is the level of parallelism, and hence the processor becomes much faster.

DRAWBACKS/ HAZARDS OF PIPELINING

There are various hazards of pipelining, which **cause a dip** in the performance of the processor. These hazards become even **more prominent** as the **number of pipeline stages increase**. They may occur
due to the following reasons.

1) DATA HAZARD/ DATA DEPENDENCY HAZARD

Data Hazard is caused when **the result (destination) of one instruction becomes the operand (source) of the next instruction**.
Consider two instructions I1 and I2 (I1 being the first).
Assume I1: INC [4000H]
Assume I2: MOV BL , [4000H]
Clearly in I2, BL should get the incremented value of location [4000H].
But this can only happen once I1 has completely finished execution and also written back the result at [4000H].
In a multistage pipeline, I2 may reach execution stage before I1 has finished storing the result at location [4000H], and hence get a wrong value of data.
This is called **data dependency hazard**.
It is solved by inserting NOP (No operation) instructions between such data dependent instructions.

Because of the data hazard, there will be a delay in the pipeline. The data hazards are basically of three types:

1. RAW
2. WAR
3. WAW

To understand these hazards, we will assume we have two instructions I1 and I2, in such a way that I2 follows :

RAW:

RAW hazard can be referred to as 'Read after Write'. It is also known as Flow/True data dependency. If the later instruction tries to read on operand before earlier instruction writes it, in this case, the RAW hazards will occur. The condition to detect the RAW hazard is when O_n (Output of nth instruction) and I_{n+1}(Input of $n+1^{th}$ instruction) both have a minimum one common operand.

I1: add R1, R2, R3

I2: sub R5, R1, R4

WAR

WAR can be referred to as 'Write after Read'. It is also known as Anti-Data dependency. If the later instruction tries to write an operand before the earlier instruction reads it, in this case, the WAR hazards will occur. The condition to detect the WAR hazard is when I_n and O_{n+1} both have a minimum one common operand.

add R1, **R2,** R3
sub **R2,** R5, R4

In a reasonable (in-order) pipeline, the WAR hazard is very uncommon or impossible.

WAW

WAW can be referred to as 'Write after Write'. It is also known as Output Data dependency. If the later instruction tries to write on operand before earlier instruction writes it, in this case, the WAW hazards will occur. The condition to detect the WAW hazard is when O_n and O_{n+1} both have a minimum one common operand.

add **R1,** R2, R3
sub **R1,** R2, R4

2) CONTROL HAZARD/ CODE HAZARD

Pipelining assumes that the program will always flow in a sequential manner.
Hence, it performs various stages of the forthcoming instructions before-hand, while the current instruction is still being executed. While programs are sequential most of the times, it is not true always.
Sometimes, branches do occur in programs.

In such an event, all the forthcoming instructions that have been fetched/ decoded etc have to be flushed/ discarded, and the process has to start all over again, from the branch address. This causes pipeline bubbles, which simply means time of the processor is wasted. Consider the following set of instructions:

Start:
JMP Down
INC BL
MOV CL, DL
ADD AL, BL
…
…
…
Down: DEC CH

JMP Down is a branch instruction.

After this instruction, program should jump to the location "Down" and continue with DEC CH
instruction.

But, in a multistage pipeline processor, the sequentially next instructions after JMP Down have already been fetched and decoded. These instructions will now have to be discarded and fetching will begin all over again from DEC CH. This will keep several units of the architecture idle for some time. This is called a pipeline bubble. The **problem of branching is solved** in higher processors by a method called "**Branch Prediction Algorithm**". It was introduced by **Pentium** processor. It relies on the **previous history** of the instruction as most programs are repetitive in nature. It then **makes a prediction** whether branch will be **taken or not** and hence puts the correct instructions in the pipelines.

3) STRUCTURAL HAZARD

Structural hazards are caused by **physical constraints in the architecture like the buses**. Even in the most basic form of pipelining, we want to execute one instruction and fetch the next one. Now as long as execution only involves registers, pipelining is possible. But **if execution requires to read/ write data from the memory, then it will make use of the buses, which means fetching cannot take place at the same time**. So the fetching unit will have to wait and hence a pipeline bubble is caused. This problem is solved in complex Harvard architecture processors, which use separate memories and separate buses for programs and data. This means fetching and execution can actually happen at the same time without any interference with each other.

E.g.: PIC 18 Microcontroller.

Introduction to Parallel Processors:

- 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 (lOP).
- However, a system with a single CPU and one or more lOPs is usually not included in the definition of a multiprocessor system unless the lOP has computational facilities comparable to a CPU.
- Multiprocessors are classified as **multiple instruction stream, multiple data stream (MIMD) systems.**

- A multiprocessor system is controlled by one operating system that provides interaction between processors and all the components of the system cooperate in the solution of a problem.
- The fact that microprocessors take **very little physical space and are very inexpensive** brings about the feasibility of interconnecting a large number of microprocessors into one composite system.
- **Very-large-scale integrated circuit technology** has reduced the cost of computer components to such a low level that the concept of applying multiple processors to meet system performance requirements has become an attractive design possibility.
- Multiprocessing **improves the reliability** of the system so that a failure or error in one part has a limited effect on the rest of the system. If a fault causes one processor to fail, a second processor can be assigned to perform the functions of the disabled processor. The system as a whole can continue to function correctly with perhaps some loss in efficiency.
- A multiprocessor system derives its high performance from the fact that computations can proceed in parallel in one of two ways.
  1. Multiple independent jobs can be made to operate in parallel.
  2. A single job can be partitioned into multiple parallel tasks.
- An overall function can be partitioned into a number of tasks that each processor can handle individually. System tasks may be allocated to special purpose processors whose design is optimized to perform certain types of processing efficiently.
- Example is a computer where one processor performs highspeed floating-point mathematical computations and another takes care of routine data-processing tasks.
- Multiprocessors are classified by the way their memory is organized.
  1. A multiprocessor system with common shared memory is classified as a shared memory or tightly coupled multiprocessor. This does not preclude each processor from having its own local memory. In fact, most commercial **tightly coupled multiprocessors** provide a cache memory with each CPU. In addition, there is a global common memory that all CPUs can access. Information can therefore be shared among the CPUs by placing it in the common global memory.
  2. An alternative model of microprocessor is the distributed-memory or **loosely coupled system**. Each processor element in a loosely coupled system has its own private local memory. The processors are tied together by a switching scheme designed to route information from one processor to another through a message- passing scheme. The processors relay program and data to other processors in packets. A packet consists of an address, the data content, and some error detection code. The packets are addressed to a specific processor or taken by the first available processor, depending on the communication system used.

Shared Memory Multiprocessors:

- A **multiprocessor system consists of a number of processors capable of simultaneously executing independent tasks.** A task may encompass a few

instructions for one pass through a loop, or thousands of instructions executed in a subroutine.

- In a shared-memory multiprocessor**, all processors have access to the same memory**. Tasks running in different processors can access shared variables in the memory using the same addresses. The size of the shared memory is likely to be large.
- Implementing a **large memory in a single module would create a bottleneck when many processors make requests to access the memory simultaneously**. This **problem is alleviated by distributing the memory across multiple modules** so that simultaneous requests from different processors are more likely to access different memory modules, depending on the addresses of those requests.
- **An interconnection network enables any processor to access any module that is a part of the shared memory**. When memory modules are kept physically separate from the processors, all requests to access memory must pass through the network. Below Figure shows such an arrangement.
- A **system which has the same network latency for all accesses from the processors to the memory modules is called a Uniform Memory Access** (UMA) multiprocessor.

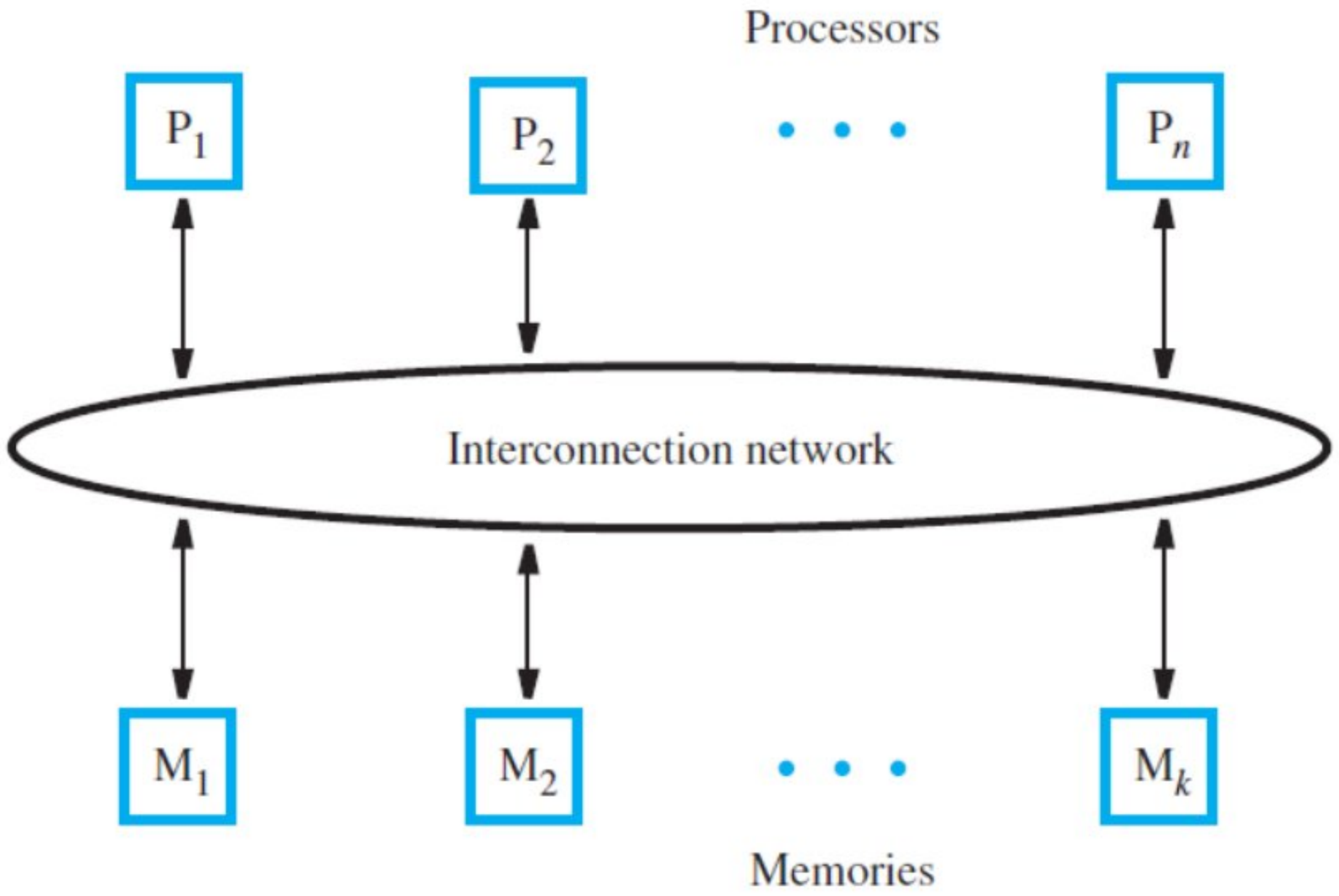


Fig: A UMA multiprocessor.

- For better performance, **it is desirable to place a memory module close to each processor.** The result is a collection of nodes, each consisting of a processor and a memory module. The nodes are then connected to the network, as shown in Figure below:

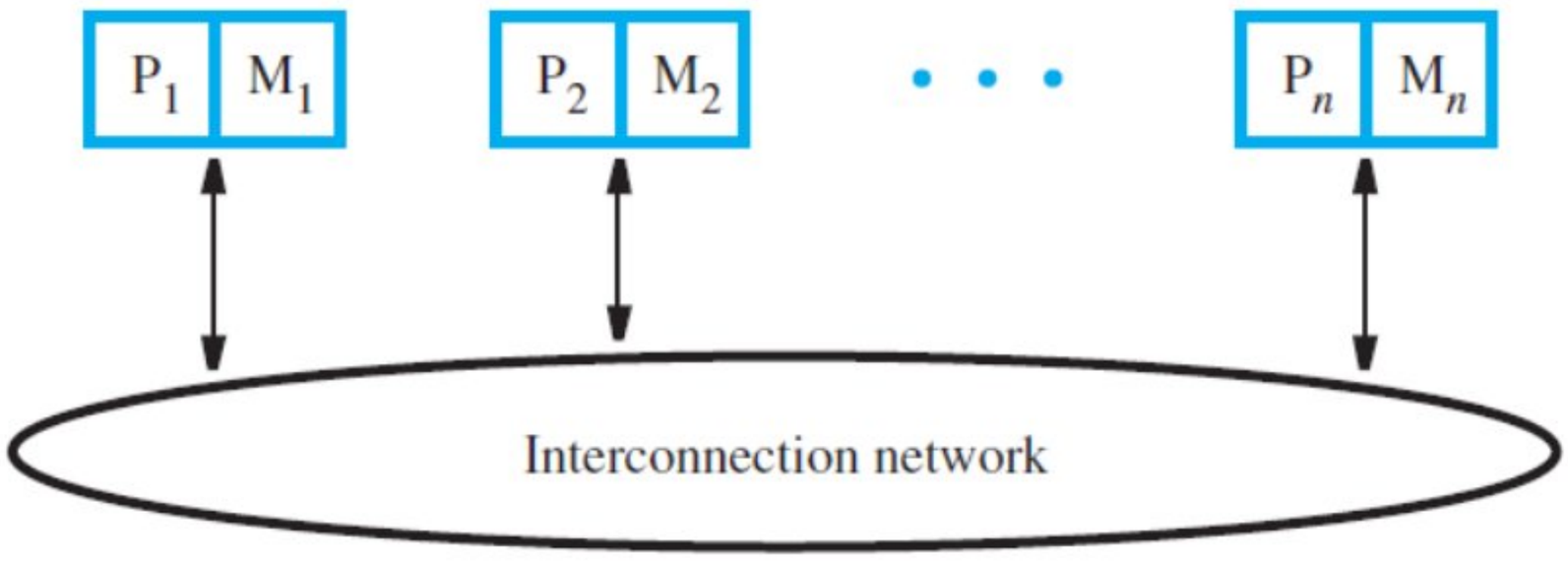


Fig: A NUMA multiprocessor.

- 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 Non-Uniform Memory Access (NUMA) multiprocessors**.

Interconnection Networks:

- The interconnection network must allow information transfer between any pair of nodes in the system. The network may also be used to broadcast information from one node to many other nodes. The traffic in the network consists of requests (such as read and write)
  and data transfers.
- The suitability of a particular network is judged in terms of **cost, bandwidth, effective throughput, and ease of implementation. The term bandwidth refers to the capacity of a transmission link to transfer data and is expressed in bits or bytes per second.** The effective throughput is the actual rate of data transfer. **This rate is less than the available bandwidth because a given link must also carry control information that coordinates the transfer of data.**
- **Information transfer through the network usually takes place in the form of packets of fixed length and specified format.** For example, a read request is likely to be a single packet sent from a processor to a memory module. The packet contains the node identifiers for the source and destination, the address of the location to be read, and a command field that indicates what type of read operation is required. A write request that writes one word in a memory module is also likely to be a single packet that includes the data to be written. On the other hand, a read response may involve an entire cache block requiring several packets for the data transfer.
- Ideally**, a complete packet would be handled in parallel in one clock cycle at any node or switch in the network.** This implies having wide links, comprising many wires. However, **to reduce cost and complexity, the links are often considerably narrower**. In such cases, a packet must be divided into smaller pieces, each of which can be transmitted in one clock cycle.
- The following are few of the interconnection networks that are commonly used in multiprocessors:

Bus:

A bus is a set of lines (wires) that provide a single shared path for information transfer. Buses are most commonly used in UMA multiprocessors to connect a

number of processors to several shared-memory modules. **Arbitration is necessary to ensure that only one of many possible requesters is granted use of the bus at any time.**

The bus is suitable for a relatively small number of processors because of the contention for access to the bus when many processors are connected. A simple bus does not allow a new request to appear on the bus until the response for the current request has been provided. However, if the response latency is high, there may be considerable idle time on the bus. Higher performance can be achieved by using a **split-transaction bus, in which a request and its corresponding response are treated as separate events. Other transfers may take place between them.**

Ring:

A ring network is formed with point-to-point connections between nodes, as shown in Figure below:

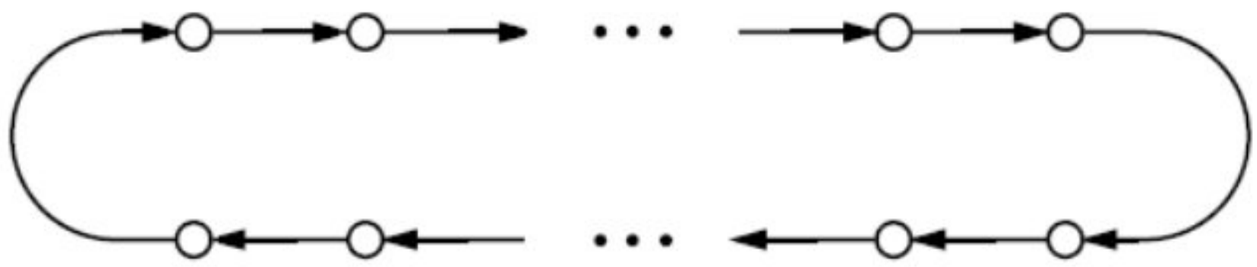
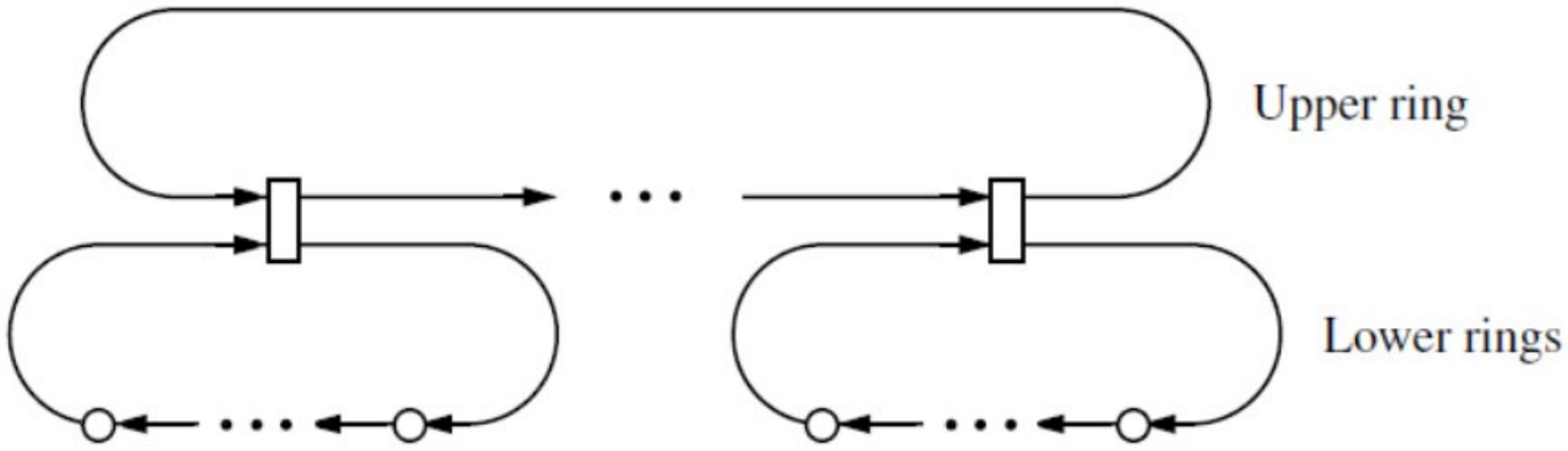


Fig: Simple Ring

A long single ring results in high average latency for communication between any two nodes. This high latency can be mitigated in two different ways. **A second ring can be added to connect the nodes in the opposite direction.** The resulting bidirectional ring halves the average latency and doubles the bandwidth. However, handling of communications is more complex.

Another approach is to use a hierarchy of rings. A two-level hierarchy is shown in Figure below: The upper-level ring connects the lower-level rings. The average latency for communication between any two nodes on lower-level rings is reduced with this arrangement. Transfers between nodes on the same lower-level ring need not traverse the

upper-level ring. Transfers between nodes on different lower-level rings include a traversal on part of the upper-level ring.



(b) Hierarchy of rings

Crossbar:

A crossbar is a network that provides a direct link between any pair of units connected to the network. It is typically used in UMA multiprocessors to connect

processors to memory modules. It enables many simultaneous transfers if the same destination is not the target of multiple requests. Below figure shows a crossbar that comprises a collection of switches. For n processors and k memories, n × k switches are needed.

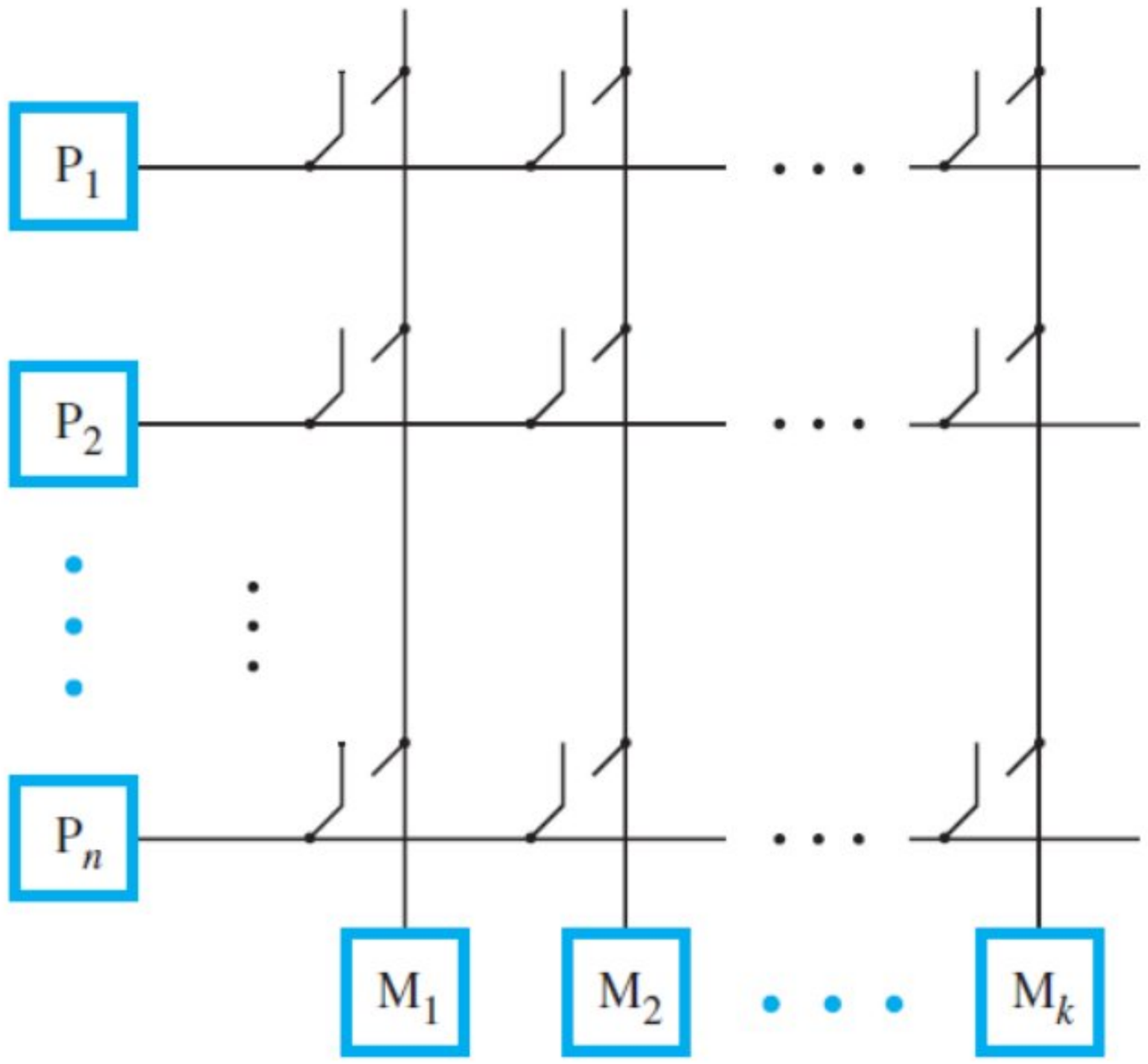


Fig: Crossbar Interconnection Network

Mesh:

A natural way of connecting a large number of nodes is with a two-dimensional mesh, as shown in Figure below:

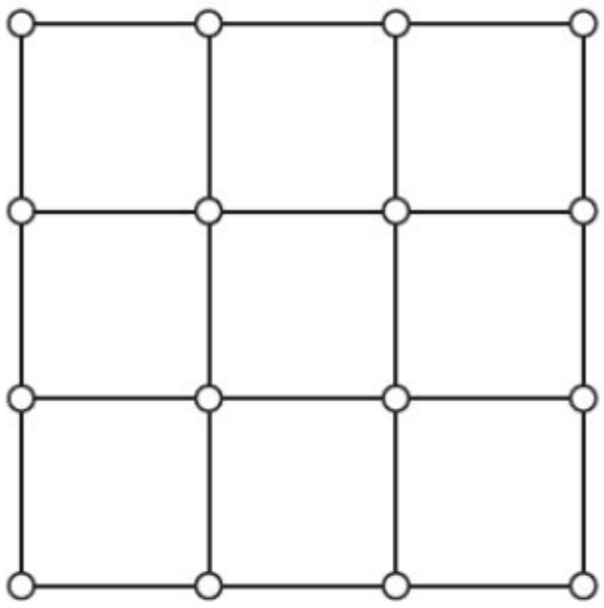


Fig : A two-dimensional mesh network.

Each internal node of the mesh has four connections, one to each of its horizontal and vertical neighbours. Nodes on the boundaries and corners of the mesh have fewer neighbours and hence fewer connections. To reduce latency for communication between nodes that would otherwise be far apart in the mesh, wrap around connections may be introduced between nodes at opposite boundaries of the mesh. **A network with**

such connections is called a torus. All nodes in a torus have four connections. Average latency is reduced, but the implementation complexity for routing requests and responses through a torus is somewhat higher than in the case of a simple mesh.

Cache Coherence:

1. A shared-memory multiprocessor is easy to program. *Each variable in a program has a unique address location in the memory, which can be accessed by any processor.* However, *each processor has its own cache*. Therefore, it is *necessary to deal with the possibility that copies of shared data may reside in several caches*.
2. When any processor writes to a shared variable in its own cache, all other caches that contain a copy of that variable will then have the old, incorrect value. They must be informed of the change so that they can either update their copy to the new value or invalidate it. This is the issue of maintaining *cache coherence, which requires having a consistent view of shared data in multiple caches*.
3. The write-through approach changes the data in both the cache and the main memory. The write-back approach changes the data only in the cache; the main memory copy is updated when a modified data block in the cache has to be replaced. Similar approaches can be used to address cache coherence in a multiprocessor system.

Write Through Protocol:

A write-through protocol can be implemented in one of two ways:

1) First version is based on updating the values in other caches. When a processor writes a new value to a block of data in its cache, the new value is also written into the memory module containing the block being modified. Since copies of this block may exist in other caches, these copies must be updated to reflect the change caused by the Write operation. The simplest way of doing this is to broadcast the written data to the caches of all processors in the system. As each processor receives the broadcast data, it updates the contents of the affected cache block if this block is present in its cache.

2) The second version of the write-through protocol is based on invalidation of copies. When a processor writes a new value into its cache, this value is also sent to the appropriate location in memory, and all copies in other caches are invalidated. Again, broadcasting can be used to send the invalidation requests throughout the system.

Write-Back protocol:

- ✓ Maintaining coherence with the write-back protocol **is based on the concept of ownership of a block of data in the memory.** Initially, the memory is the owner of all blocks, and the memory retains ownership of any block that is read by a processor to place a copy in its cache.
- ✓ If some processor wants to write to a block in its cache, it must first become the exclusive owner of this block. To do so, **all copies in other caches must first be invalidated with a broadcast request**. The new owner of the block may then modify the contents at will without having to take any other action.
- ✓ **Read: When another processor wishes to read a block that has been modified, the request for the block must be forwarded to the current owner.** The data

are then sent to the requesting processor by the current owner**. The data are also sent to the appropriate memory module, which reacquires ownership and updates the contents of the block in the memory.** The cache of the processor that was the previous owner retains a copy of the block. **Subsequent requests from other processors to read the same block are serviced by the memory module containing the block.**

✓ When another processor wishes to write to a block that has been modified, the current owner sends the data to the requesting processor. **It also transfers ownership of the block to the requesting processor and invalidates its cached copy**. Since the block is being modified by the new owner, the contents of the block in the memory are not updated. The next request for the same block is serviced by the new owner.

✓ The ***write-back protocol has the advantage of creating less traffic than the write-through protocol***. This is because a processor is likely to perform several writes to a cache block

before this block is needed by another processor. With the write-back protocol, these writes are performed only in the cache, once ownership is acquired with an invalidation request.

Snoopy Caches:

✓ In multiprocessors that connect a modest number of processors to the memory modules using a single bus, ***cache***

coherence can be realized using a scheme known as **snooping.**

✓ In a single-bus system, **all transactions between processors and memory modules occur via requests and responses on the bus.** Suppose that each processor cache has a controller circuit that observes, or snoops, all transactions on the bus.

✓ Below are some scenarios for the write-back protocol and how cache coherence is enforced**:**

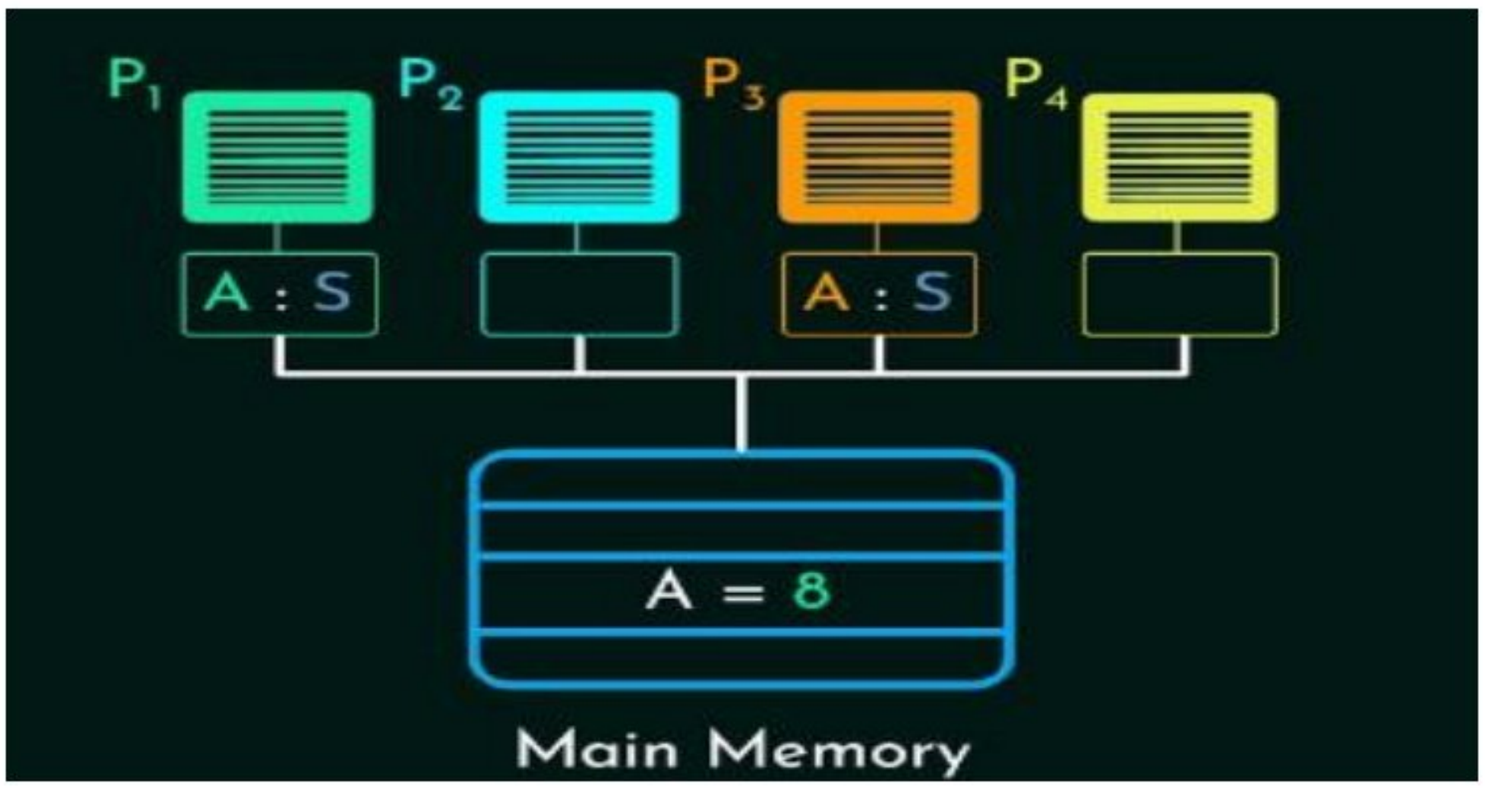
➢ Consider a processor that has previously read a copy of a block from the memory into its cache. Before writing to this block for the first time, **the processor must broadcast an invalidation request to all other caches, whose controllers accept the request and invalidate any copies of the same block.** This action causes the requesting processor to become the new owner of the block. The processor may then write to the block and mark it as being modified. No further broadcasts are needed from the same processor to write to the modified block in its cache.

➢ Now, if another processor broadcasts a read request on the bus for the same block, the memory must not respond because it is not the current owner of the block. The processor owning the requested block snoops the read request on the bus. **Because it holds a modified copy of the requested block in its cache, it asserts a special signal on the bus to prevent the memory from responding**. The owner then broadcasts a copy of the block on the bus, and marks its copy as clean (unmodified). The data response on the bus is accepted by the cache of the processor that issued the read request. **The data response is also accepted by the memory to update its copy of the block. In this case, the memory reacquires ownership of the block, and the block is said to be in a shared state because copies of it are in the caches of two processors**. Coherence is maintained because the two cached copies and the copy of the block in the memory contain the same data. **Subsequent requests from any processor are serviced by the memory**.

➢ Consider now the situation in which **two processors have copies of the same block in their respective caches**, and **both processors attempt to write to the same cache block at the same time**. Since the block is in the shared state, the memory is the owner of the block. Hence, **both processors request the use of the bus to broadcast an invalidation messag**e. One of the processors is granted the use of the bus first. **That processor broadcasts**

its invalidation request and becomes the new owner of the block. Through snooping, the copy of the block in the cache of the other processor is invalidated. When the other processor is later granted the use of the bus, it broadcasts a **read-exclusive request. This request combines a read request and an invalidation** request for the same block. The controller for the first processor snoops the read-exclusive request, provides a data response on the bus, and invalidates the copy in its cache. **Ownership of the block is therefore transferred to the second processor making the request**. The memory is not updated because the block is being modified again. Since the requests from the two processors are handled sequentially, cache coherence is maintained at all times.

- **The scheme just described is based on the ability of cache controllers to observe the activity on the bus and take appropriate actions**. Such schemes are called **snoopy-cache techniques**.



Snooping based protocol

Directory-Based Cache Coherence:

The concept of snoopy caches is easy to implement in single-bus systems. **Large shared memory multiprocessors use interconnection networks such as rings and meshes**. In such systems, **broadcasting every single request to the caches of all processors is inefficient**. A scalable, but more complex, solution to this problem **uses directories** in each memory module **to indicate which nodes may have copies of a given block in the shared state. If a block is modified, the directory identifies the node that is the current owner.** Each request from a processor must be sent first to the memory module containing the relevant block. **The directory information for that block is used to determine the action that is taken. A read request is forwarded to the current owner, if the block is modified. In the case of a write request for a block that is shared, individual invalidations are sent only to nodes that may have copies of the block in question**. The cost and complexity of the directory-based approach for enforcing cache coherence limits its use to large systems. Small multiprocessors, including current multicore chips, typically use snooping.

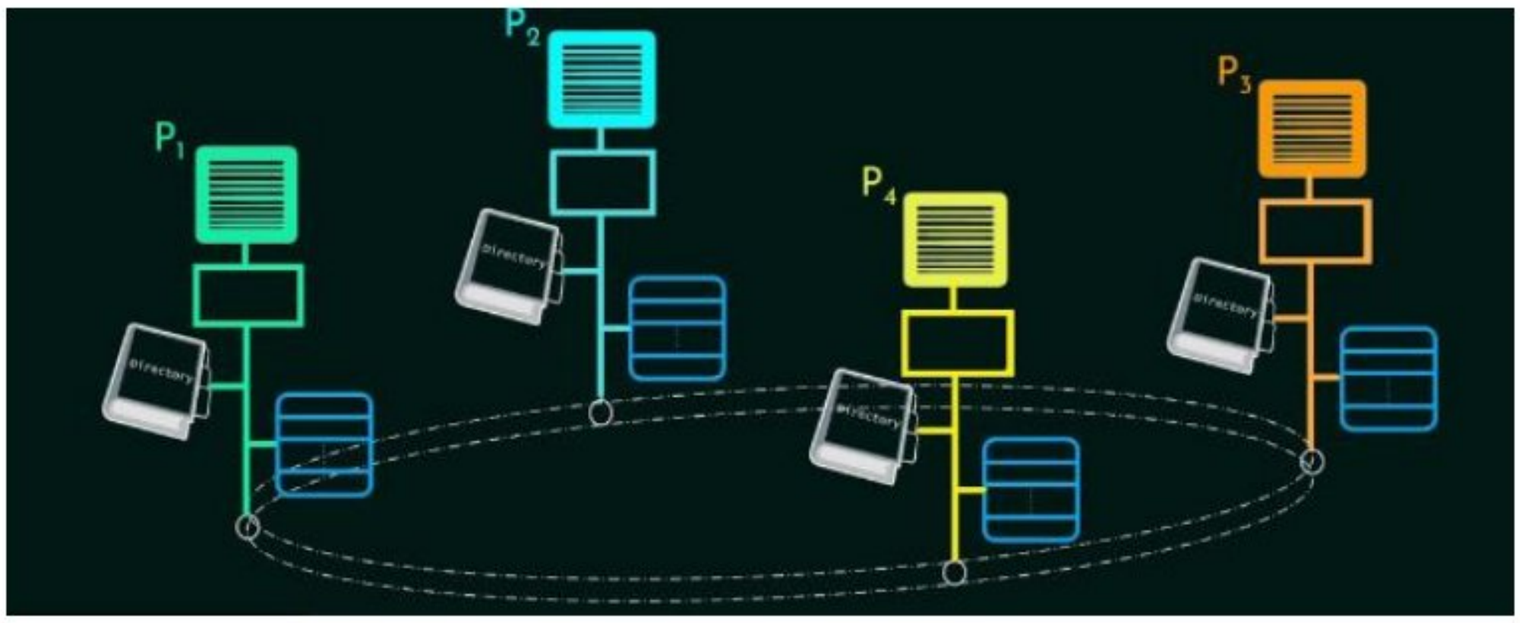


Fig: Directory based protocol