MODULE III

Multiprocessors system interconnects - Hierarchical bus systems, Cross bar switch and multiport memory, Multistage and combining networks. Cache Coherence and Synchronization Mechanisms, Cache Coherence Problem, Snoopy Bus Protocol, Directory Based Protocol, Hardware Synchronization Problem

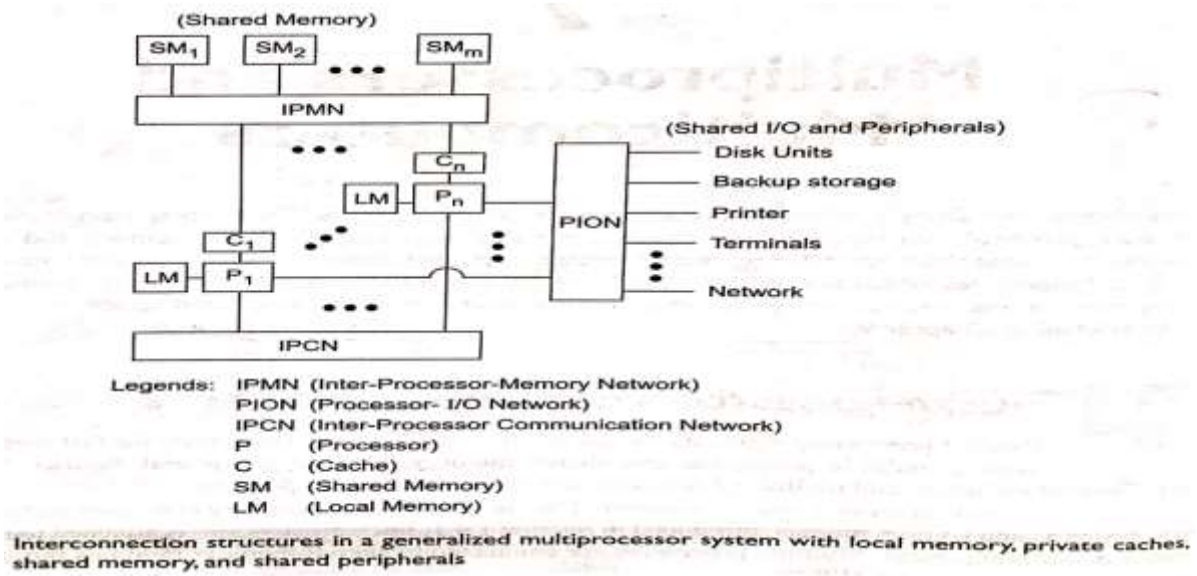
3.1. Multiprocessors System Interconnects

Qn: Explain in detail the various multiprocessor system interconnect architecture?

Parallel processing demands the use of efficient system interconnects for fast communication among multiple processors and shared memory, I/0, and peripheral devices. The following are the commonly used system interconnects this purpose.

- **Hierarchical buses.**
- **crossbar switches. And**
- **multistage networks**

Generalized multiprocessor system is depicted below. Each Pi is attached to its own local memory and private cache. Multiple processors are connected to shared-memory modules through an **inter-processors-memory network(IPMN).** These processors share the access of I/O and peripheral devices through **processor I/O network(PION).** Direct interprocessor communications are supported by an optional **interprocessor communication network (IPCN)** instead of through the shared memory.



Interconnection structures in a generalized multiprocessor system with local memory, private caches, shared memory, and shared peripherals

FIG: 3.1

Interconnection Network Choices/Characteristics

- The networks are designed with many choices,the choices are based on the topology, **timing, switching and control strategy**. In case of dynamic network

the multiprocessors interconnections are under program control.

- **Timing control** can be either synchronous and asynchronous
  - **Synchronous** – controlled by a global clock which synchronizes all network activity.
  - **Asynchronous** – use handshaking or interlock mechanisms for communication and especially suitable for coordinating devices with different speed.
- **Switching Method**
  - **Circuit switching** – once a device is granted a path in the network, it occupies the path for the entire duration of the data transfer.
  - **Packet switching** – the information is broken into small packets individually competing for a path in the network
- **Network Control** strategies are:-
  - **Centralized** – global controller receives and acts on requests
  - **Distributed** – requests handled by local devices independently

Three different types of Multiprocessor System interconnects are

1. **Hierarchical Bus systems**
2. **Crossbar Switch and multiport memory**
3. **Multistage and Combining Netorks**

3.1.1 Hierarchical Bus Systems.

- A bus system consists of a hierarchy of buses connecting various systems and components in a computer. Different buses are used to perform different interconnections
- **Local Bus** – implemented within processor chips or on printed-circuit boards.
- **Memory Bus –** used to connect memory with interface logic
- **Data bus –** I/O or network interface chip or board uses data bus
- **Backplane bus –** is a printed circuit board on which many connectors are used to plug in functional boards.
- **System bus –** built on backplane, provides a common communication path among all plug-in boards.

- **I/O Bus –** used to connect I/O devices to a computer system. Ex: SCSI bus. Made of coaxial cables, with taps connecting disks, printer, and other devices to a processor through an I/O controller .Special interface logic is used to connect various board types to the backplane bus.

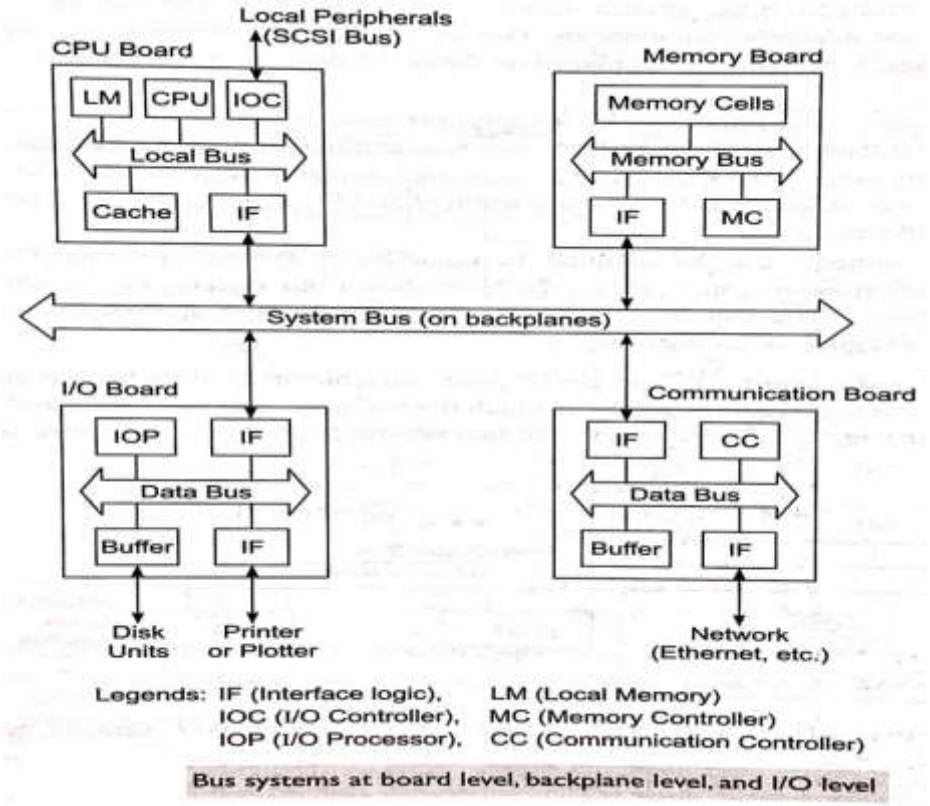


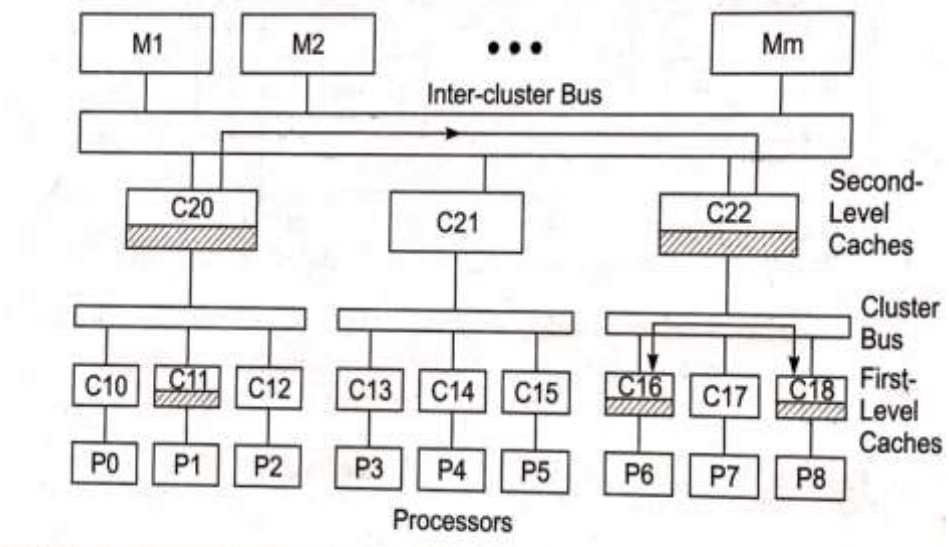
FIG: 3.2

Hierarchical bus systems can be used to build medium sized multiprocessors with less than 100 processors.

Hierarchical Buses and Caches

- **There are numerous ways in which buses, processors, memories, and I/O devices can be organized.**
- **One organization has processors (and their caches) as leaf nodes in a tree, with the buses (and caches) to which these processors connect forming the interior nodes.**
- **This generic organization, with appropriate protocols to ensure cache coherency, can model most hierarchical bus organizations.**

- A multilevel tree structure in which leaf nodes are processors and their private caches.(P0-P8 are processors, C10- C18 are their private caches(first level caches))



A hierarchical cache/bus architecture for designing a scalable multiprocessor

FIG: 3.3

- divided into different clusters each connected through a cluster bus.
- Inter-cluster bus used to provide connection among clusters.
- **Second level caches(C20-C22**) are used between each cluster and inter-cluster bus. Second level cache have higher capacity than sum of capacities of all first level caches connected beneath it.
- Main memory modules are connected to the **Inter-cluster bus.**
- **Most memory request should be satisfied at the first level caches.**

Each single cluster operates as a single-bus system. **Snoopy bus coherence protocols** can be used to establish consistency among first-level caches belonging to the same cluster. **Second-level caches** are used to extend consistency from each local cluster to the upper level.

Qn:Explain the role of bridges in hierarchical networks?

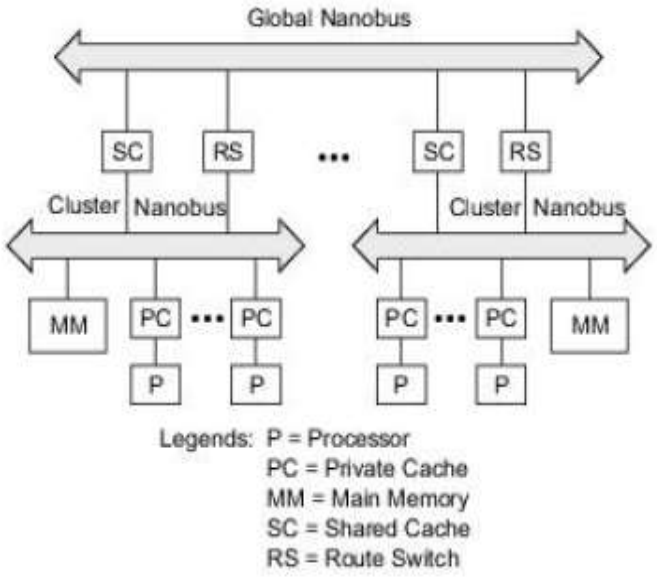
Bridges - The idea of using bridges between multiprocessor clusters is to allow transactions initiated on a local bus to be completed on a remote bus..that is The term bridge is used to denote a device that is used to connect two (or possibly more) buses.

The interconnected buses may use the same standards, or they may be different (e.g. PCI and ISA buses in a modern PC).
Bridge functions include

- Communication protocol conversion
- Interrupt handling in split transactions
- Serving as cache and memory agents

Example of multiprocessor using hierarchical buses:(Optional to study. Atleast do familiar with the name)

Encore Ultramax multiprocessor architecture



The Ultramax multiprocessor architecture using hierarchical buses with multiple clusters (Courtesy of Encore Computer Corporation, 1987)

FIG: 3.4

The Ultramax had a **two-level hierarchical-bus architecture** as depicted above. The Ultramax architecture was very similar to that the architecture shown in fig 3.3, except that the **global Nanobus** was used only for intercluster communications.

3.1.2 CROSSBAR SWITCH AND MULTIPORT MEMORY

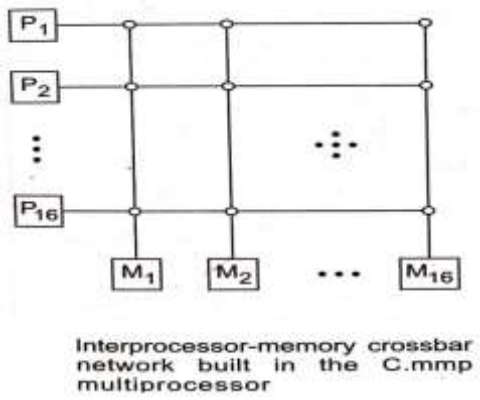
Qn:Exemplify blocking and non blocking multistage networks:
Qn:What is the use of crossbar network?

- Switched networks provide **dynamic interconnections** between the inputs and outputs. Major classes of switched networks are specified below based on **number of stages and blocking or non blocking**
- **Crossbar networks** are mostly used in **small or medium-size systerns.**
- The **multistage networks** can be **extended to larger** systems if the increased latency problem can be suitably addressed.

Network Stages

- **Single stage networks** are sometimes called recirculating networks because data items may have to pass through the single stage many times. The crossbar switch and the multiported memory organization (seen later) are both single-stage networks.

- **Multistage networks** consist of multiple stages of switch boxes, and should be able to connect from any input to any output.
- A multistage network is called **blocking** if the simultaneous connections of some multiple input output pairs may result in conflicts in the use of switches or communication links.In fact most multistage networks are blocking in nature.,eg:Omega,Baseline, Banyan etc..
- A **nonblocking multistage network** can perform all possible connections between inputs and outputs by rearranging its connections. Eg:Benes network, Clos network etc..
- **Crossbar networks** Every input port is connected to a free output port through a crosspoint switch(circles in fig below) without blocking. A crossbar network is a single stage, non-blocking permutation network(one-to-one), built with unary switches at the cross points.



Interprocessor-memory crossbar network built in the C.mmp multiprocessor

Fig:3.5

- Once the data is read from the memory. its value is returned to the requesting processor along the same crossp-point switch.

- In **an n-processor, m-memory system, n * m crosspoint switches** will be required. Each crosspoint is a unary switch which can be open or closed, providing a point-to-point connection path between the processor and a memory module.

- All processors can send memory requests independently and asynchronously. This poses the problem of multiple requests destined for the same memory module at the same time. In such cases, only one of the requests is serviced at a time.

Crosspoint Switch Design

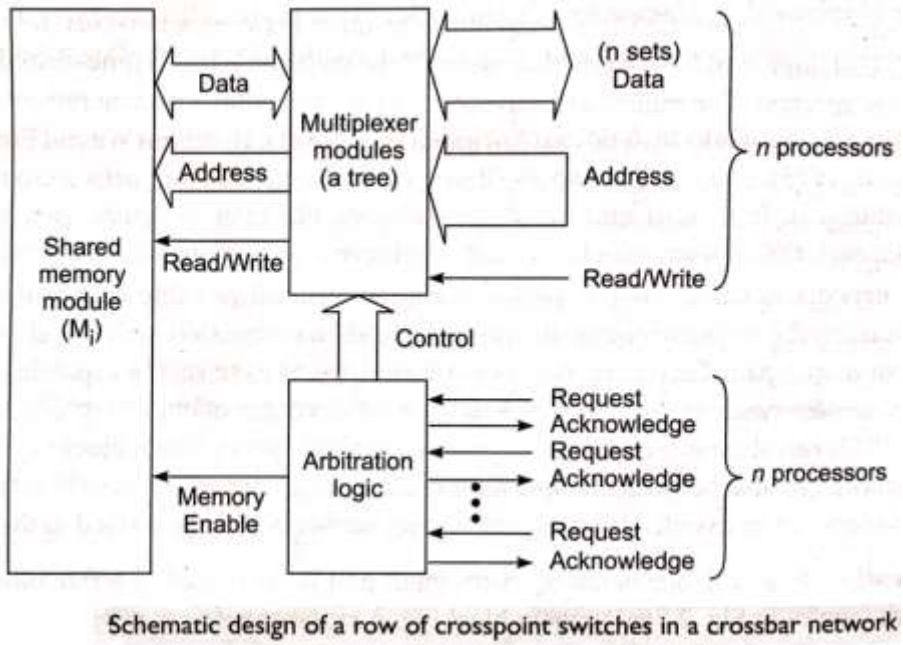
- Note that each memory module can satisfy only one processor request at a time.However, each processor can generate a sequence of addresses to access multiple memory modules simultaneously.
- Thus, Out of n crosspoint switches **in each column** of an *n* * m crossbar mesh, **only one** can be connected at a time. Crosspoint switches must be designedwith **extra hardware** to handle the **potential contention** for each memory module.

- For an *n x n* crossbar network, this implies that n^2 sets of crosspoint switches and a large number of lines(address ,data, control)are needed. So far only relatively small crossbar networks with n <=16 have been built into commercial machines.

- **On each row** of the crossbar mesh, **multiple crosspoint** switches can be connected simultaneously. Simultaneous data transfers can take place in a crossbar between n pairs of processors and memories.

Figure below 3.6 shows the schematic design of a row of crosspoint switches in a single crossbar network.

- **Multiplexer modules** are used to select one of n read or write requests for service.
- Each processor sends in an independent request, and the **arbitration logic** makes the selection based on certain **fairness or priority rules**.



Schematic design of a row of crosspoint switches in a crossbar network

Fig:3.6

- A **4-bit control signal** will be generated for n = I6 processors.
- Based on the control signal received, only one out of n sets of information lines is selected as the output of the **multiplexer tree.**

- The memory address is entered for both read and write access. In the case of **read,** the data fetched from memory are returned to the selected processor in the reverse direction using the data path established In the **case of write,** the data on the data path are stored in memory.
- **Acknowledge signals** are used to indicate the arbitration result to all requesting processors. These signals initiate data transfer and are used to **avoid conflicts.**

Crossbar Limitations

- Crossbar network is cost-effective only for small multiproccssors with a few processors accessing a few memory modules.
- Expansion is limited by the size of switch matrix
- Complexity of switch grows as square of no of modules in the system – thus does not ensure performance in proportion to cost.
- This interconnection system is complex to design and expensive to fabricate.

MULTIPORT MEMORY

Qn:Describe how multiport memories used in multistage networks?

- As crossbar n/w becomes expensive multiprocessors use a multiport memory organization.
- It moves all crosspoint arbitration and switching functions associated with each memory module into the **memory controller.**
- Thus,the memory module becomes more expensive due to added access ports and associated logic.

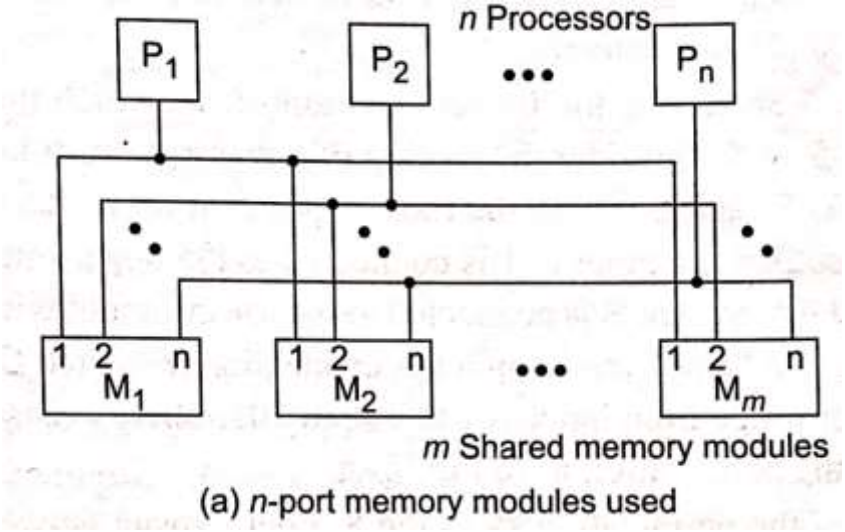
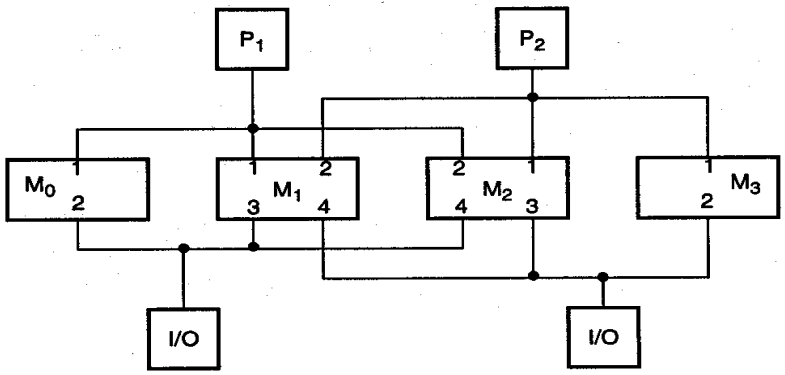


Fig:3.7

- The circles in diagram represent n switches tied to n input ports of memory module. Only one of n processor requests can be honored at a time.
- The multiport memory organisation is a compromise between low cost, low performance bus system and high cost high bandwidth crossbar system.

Disadvantages

- Memory structure becomes e**xpensive** when m and n becomes very large.
- **Large no of interconnection cables** and connectors are required for large configurations.
- A multiport memory multiprocessor is **not scalable** because once the ports are fitted, no more processors can be added without redesigning the memory controller.



(b) Memory ports prioritized or privileged in each module by numbers

Fig:3.8

The ports of each memory rnodule in Fig. Above 3.8 are prioritized. Some of the processors are CPUs, some are i/o processors, and some are connected to dedicated processors.

4 MULTISTAGE AND COMBINING NETWORKS

Prerequisite for studying this topic

DATA ROUTING FUNCTIONS

- *Used for inter-PE data exchange.. The versatility of a routing n/w will reduce the time needed for data exchange and thus improve performance.*

Commonly used Data-routing functions among PE's

- *Shifting*
- *Rotation*
- *Permutation(one to one)*
- *Broadcast(one to all)*
- *Multicast (one to many)*
- *Shuffle*
- *Exchange.*

  Theses routing functions can be implemented on ring, mesh, hypercube or multistage networks

PERMUTATIONS

For n objects, there are n! Permutations by which n objects can be reordered.

For example, Π= (a, b, c) (d, e) stands for a→ b, b→c, c→a, d→e and e→d in a

*circular fashion. Cycle (a, b, c) has a period of 3 and cycle (d, e) has a period of 2. So, the total permutation (π) has a period of 3*2 =6.*

Permutation can be implemented by using crossbar switch, multistage n/w , shifting or broadcast operations.

PERFECT SHUFFLE and EXCHANGE (A special permutation function for parallel processing applications)

Means left shifting of binary bit string by 1 bit and moving the most significant bit at the least significant position.

Ex: shuffle(001) – 010
Shuffle(101) – 011

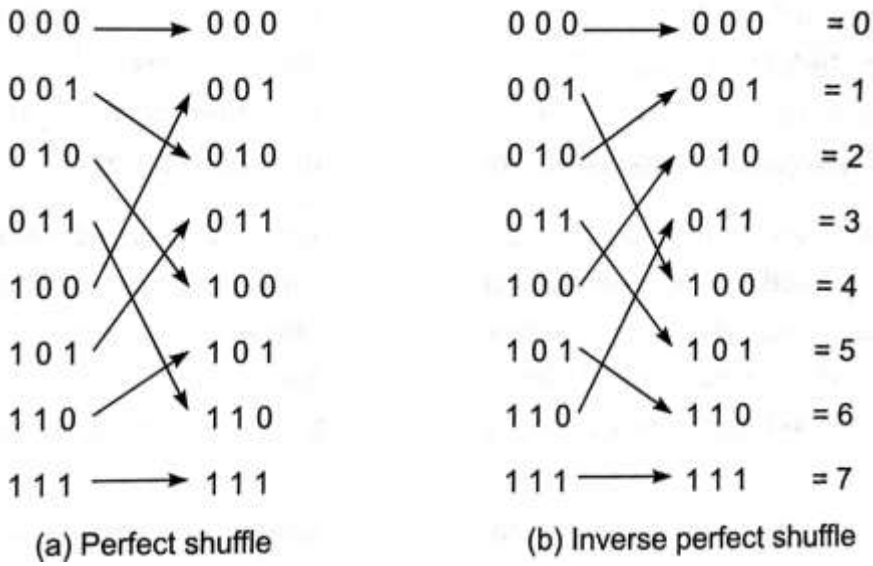


Fig:3.9

DYNAMIC CONNECTION NETWORKS ***(3types)***

- *Used for general-purpose and multi purpose applications*
- *Used to implement all communication patterns based on program demands.*
- *Switches and arbiters used along connection paths to provide dynamic connectivity*
- *In increasing order of cost and performance dynamic conn n/w's include*
    1. *Bus systems* ***(Refer text)***
    2. *Multistage interconnecton networks****(MIN) (briefed below)***
    3. *Crossbar switch networks* ***(Refer text)***

MULTISTAGE INTERCONNECTION NETWORKS (MINs)

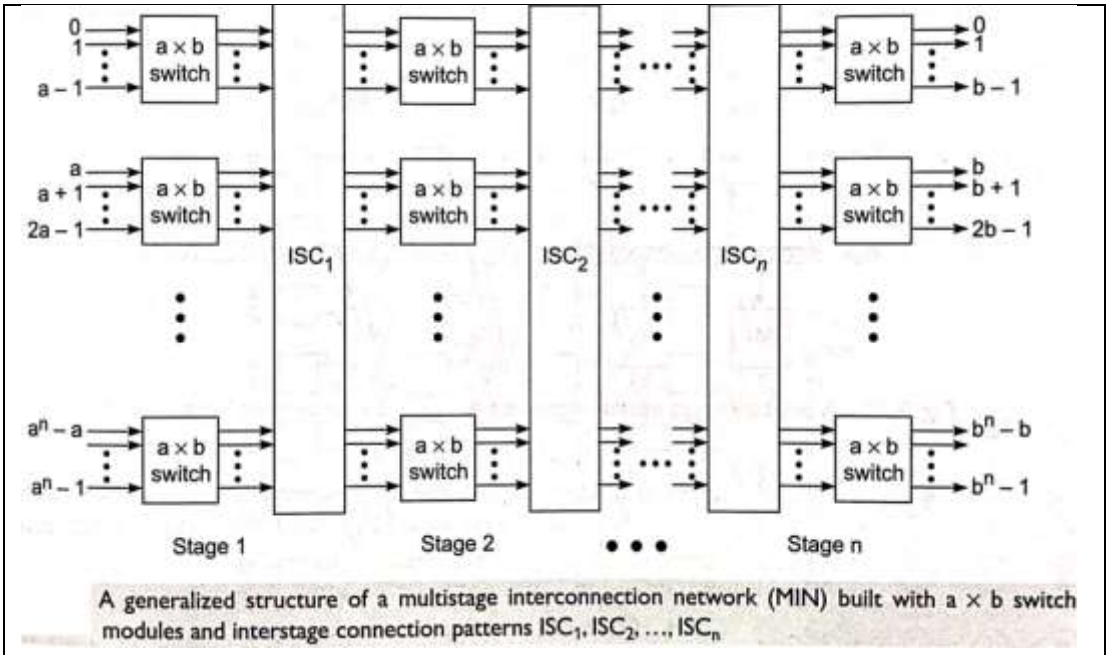


Fig:3.10

- *A no: of a x b switches used in each stage.*
- *Fixed interstage connections are used between switches in adjacent stage. They can be dynamically set to establish the desired connections between inputs and outputs*
- *Diff classes of MIN's differ in switch modules used and in the kind of* ***inter stage connection(ISC)*** *patterns used*
    - *Simplest switch module(2x2)*
    - ***ISC patterns –perfect shuffle,*** *butterfly, multiway shuffle, crossbar, cube connection*

One of the ISC patterns are shown below :

1.1 Omega Network

- *4 possible connections of 2x2 switches used in omega n/w shown below*

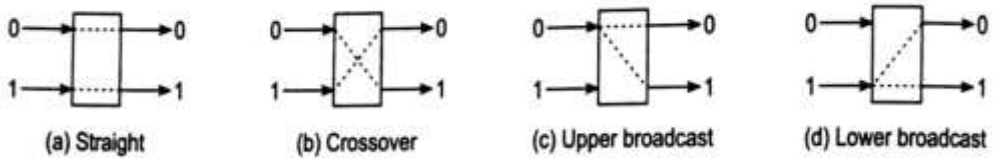
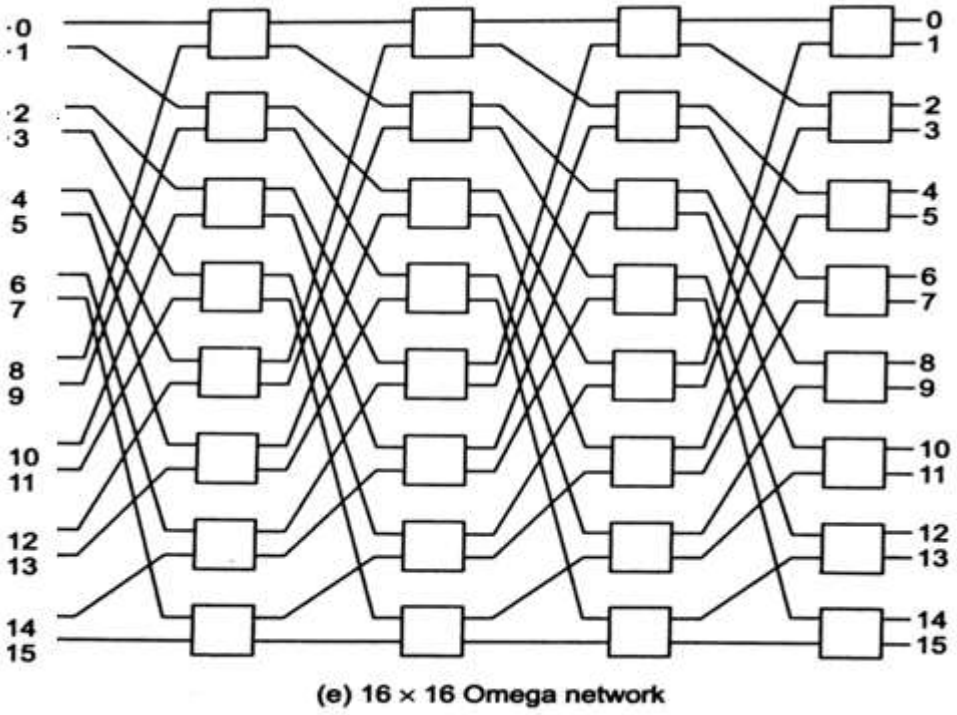


Fig:3.11

- *A 16x16 omega n/w needs four stages of 2x2 switches*
- *ISC pattern is* ***perfect shuffle*** *over 16 objects.*
- *n input Omega n/w requires* $\boldsymbol{\log_2 n}$ ***stages*** *of 2x2 switches.*
- *Each stage require* ***n/2 switch modules****(16/2=8)*
- *Total no of switches in n/w –* $\boldsymbol{(n\log_2 n)/2}$ ***switches –*** $\boldsymbol{(16\log_2 16)/2}$*=32 switches in total*

- *Each switch module is controlled individually.*
-



(e) 16 × 16 Omega network

Fig:3.12

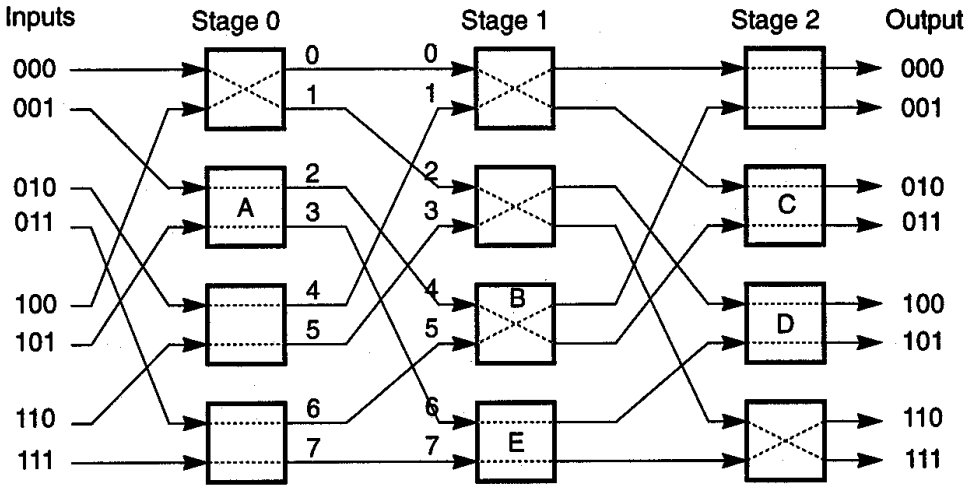
- Multistage networks are used to build larger multiprocessor systems.
- Here we discuss 2 multistage networks:
  - **Omega Network**
  - **Butterfly network**
  - And a special class of multistage network called **combining networks**

Qn:Explain blocking and non blocking network with the help of omega network?

Routing in Omega Network

- *ISC pattern is **perfect shuffle***
- An n-input Omega network has **$\log_2$ n stages.** The stages are labeled from 0 to $\log_2$n-1 from the input end to output end. An 8 input n/w will have 3 stages ($\log_2$8=3)
- **Data routing is controlled by inspecting the destination code in binary**. When the ith high-order bit of the destination **code is 0, a 2x2 switch at stage i connects the input to the upper output**, otherwise the input is directed to the lower output
- Let see the switch setting in fig 3.13 below with respect to permutation π

=(0,7,6,4,2) (1,3)(5)



(a) Permutation $\pi_1 = (0,7,6,4,2)(1,3)(5)$ implemented on an Omega network without blocking

Fig:3.13

- The switch settings in above fig are for implementation of π=(0,7,6,4,2)(1,3) which maps 0→7,7→6, 6→4, 4→2, 2→0, 3→1 and 1→3.
- Let's consider routing a message from input 001 to output 011. It involves switches A,B and C in above fig.
- Since the most significant bit(MSB) of the destination 011 is a '0', switch A must be set straight so that the input 001 is connected to the upper output (labeled 2). The middle bit in 011 is a '1' thus input 4 to switch B is connected to the lower output with a crossover connection. The least significant bit in 011 is an '1', implying flat connection in switch C.
- Similarly switches A,E and D are set for routing a message from input 101 to output 101.There **exists no conflict** in all the switch settings needed to implement the permutation *π1* in Fig.3.13

Blocking Omega network

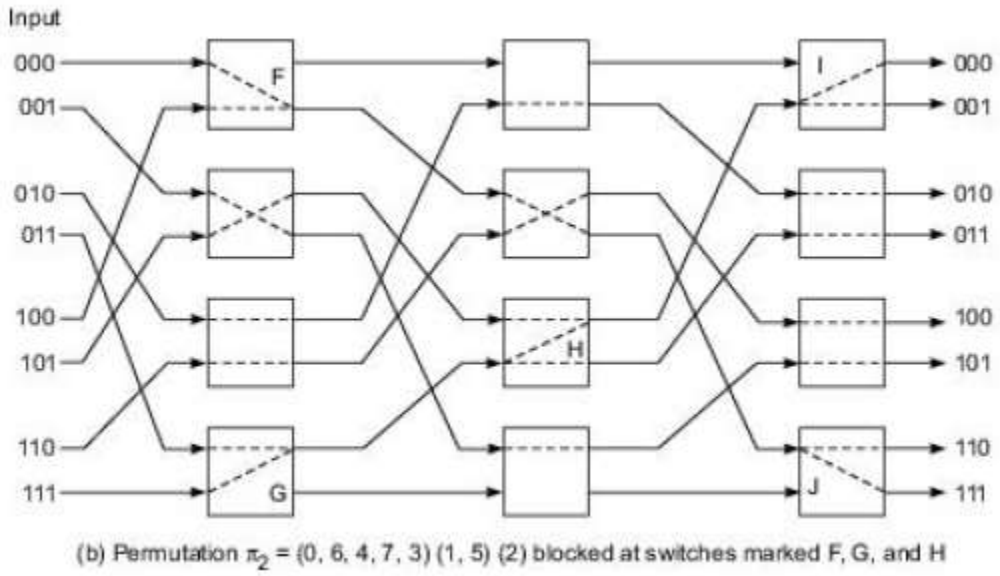


Fig.3.14

- Now consider implementing the permutation π_2 in the 8-input Omega network (Fig.3.14). Conflicts in switch settings do exist in three switches identified as F, G, and H
- . The conflicts occurring
  - **at F are** caused by the desired routings 000 —> 110 and 100 —> lll.
  - **at switch G** between 0ll —> 000 and lll —> 011.
  - and at **switch H** between 101 -—> 001 and 0ll —> 000.

- To resolve the conflicts, **one request must be blocked.**
- At switches I and J, broadcast is used from one input to two outputs.
- The above example indicates the fact that **not all permutations can be implemented in one pass through the Omega network**.

- The **Omega network is a blocking network.** In case of blocking, one can establish the conflicting connections in **several passes**. For the example π_2, we can connect conflicting connections in 2 passes as shown below:-

First pass	Second pass
000-110	100-111
101-001	011-000
111-011	
001-101	
010-010	
110-100	

- In general, if 2 X 2 switch boxes are used, an n-input Omega network can implement $n^{n/2}$ **permutations in a single pass.**
- There are $n!$ **permutations in total.**

Eg:
In an Omega network ,For n=8,
No.of permutations implementable in single pass=8^4
Total permutations = 8!

Therefore **% of permutations implementable in a single pass** through an 8 input omega network = $8^4/8!$
=4096/40320
=0.1016
=10.16%
All others will cause blocking.

Omega Network-Broadcast connections

The Omega network can also be used to broadcast data from one source to many destinations, as exemplified in *Fig. 3.15 below*, using the upper broadcast or lower broadcast switch settings. In *Fig. 3.15*, the message at input 001 is being broadcast to all eight outputs through a binary tree connection.

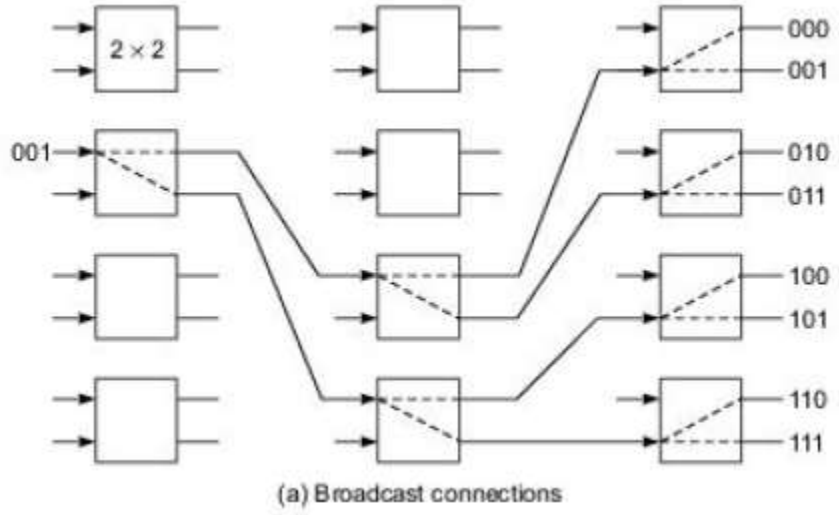


Fig: 3.15

Omega network-Larger Switches

- Larger switches (more inputs and outputs, and more switching patterns) can be used to build an Omega network, resulting in fewer stages.
- For example, with 4×4 switches, only $\log_4 16 = 2$ stages are required for a 16-input switch.
- A k-way perfect shuffle is used as the ISC for an Omega network using $k \times k$ switches.

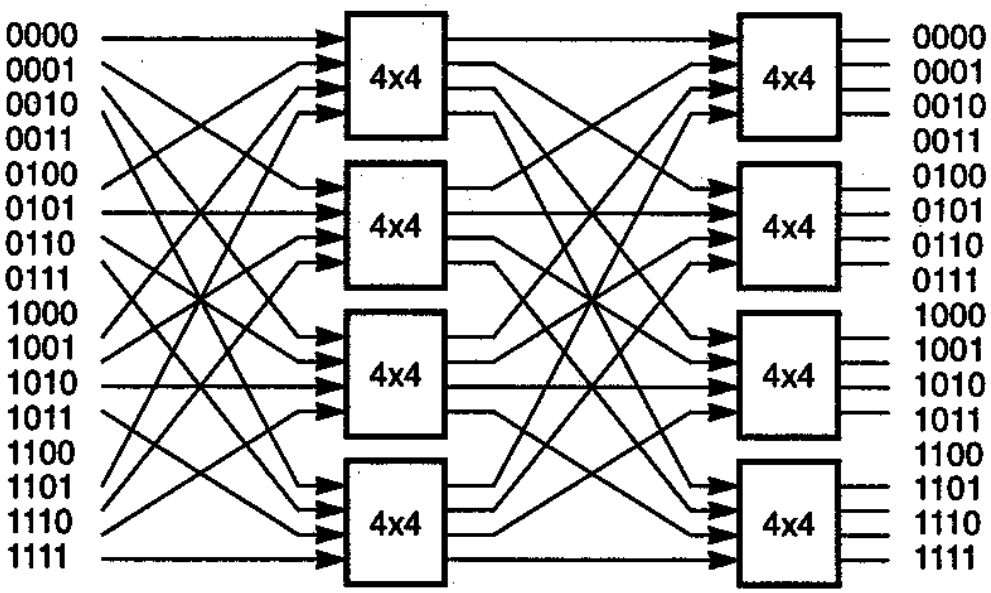


Fig: 3.16 Omega Network with 4×4 Switches(4-way shuffle interstage connections)

Sample Problem

Problem 7.12

(a) Draw a 16-input Omega network using 2×2 switches as building blocks.

(b) Show the switch settings for routing a message from node 1011 to node 0101 and from node 0111 too node 1001 simultaneously. Does blocking exist in this case?

(c) Determine how many permutations can be implemented in one pass through this Omega network. What is the percentage of one-pass permutations among all permutations?

(d) What is the maximum number of passes needed to implement any permutation through the network?

Routing In Butterfly Networks

- This class of network is constructed with crossbar switches as building blocks.
- The fig below shows a 64 input Butterfly n/w built with 2 stages ($\log_8 64=2$) of 8x8 crossbar switches.
- **Eight way shuffle function** is used to establish the interstage connections between stage 0 and stage 1. (there is connection from first box to first port of all other boxes.ie: 0 to 0, 1 to 8, 2 to 16….7 to 56 and from second box to second port of all other boxes, ie: 8 to1, 9 to 9, 10 to 17…15 to 57 and similarly for connections.

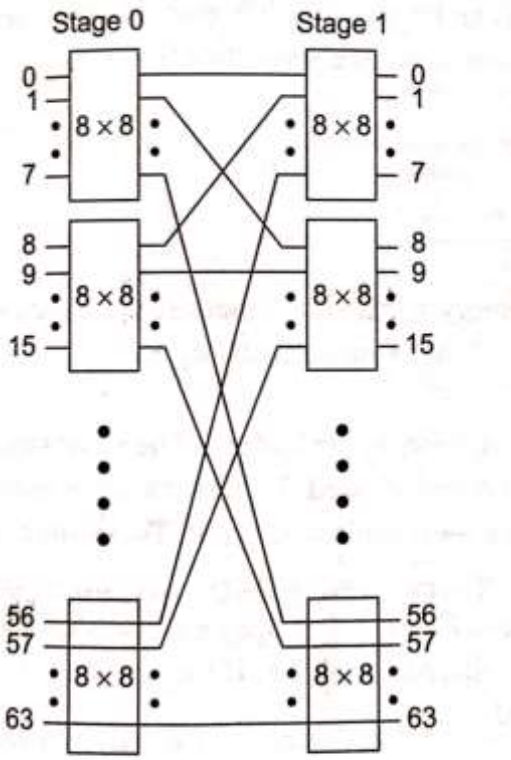


Fig:3.17

- No broadcast connections are allowed in Butterfly networks, making them a restricted subclass of the Omega networks

Qn:Explain Hotspot problem?

Combining Networks-----Hot-Spot Problem

- When network traffic is non uniform a hot spot may appear corresponding to a **certain memory module being excessively accessed by many processors at the same time**. It degrades network performance.
- Ex: semaphore variable used as synchronization barrier may become a hot spot since its shared by many processors.
- Hot spots may **degrade the network performance significantly**. In the NYU Ultracomputer and the IBM

Technique to avoid hot spots- **combining Networks**

- Thus to combine multiple requests heading for the same destination at switch points where conflicts are taking place, a **combining mechanism** has been added to the Omega network.
- An atomic read-modify-write primitive ***Fetch & Add(x, e),*** has been developed to perform parallel memory updates using the combining network.

Fetch & Add

In a Fetch&Add(x, e) operation, 'x' is an integer variable in shared memory and 'e' is an integer increment. When a **single processor executes** this operation, the semantics is:-

```
Fetch&Add (x, e)
        {temp  ←  x;
         x     ←  temp + e;
        return  temp}
```

(NOTE:This function returns the previous value of X)

When N processes attempt Fetch&Add(x, e) at the same memory word simultaneously, the memory is updated only once following a *serialization principle*. The sum of the N increments, e1+e2+e3+….eN is produced in any arbitrary serialization of the N requests.

This sum is added to the memory word x, resulting in a new value x + e1+e2+e3+….eN. **The values returned to the N requests are all unique**, depending on the serialization order followed.

- *x* is returned to one processor, $x+e_1$ to the next, $x+e_1+e_2$, to the next, and so forth.
- The value $x+e_1+e_2+\ldots+e_n$ is stored in memory.

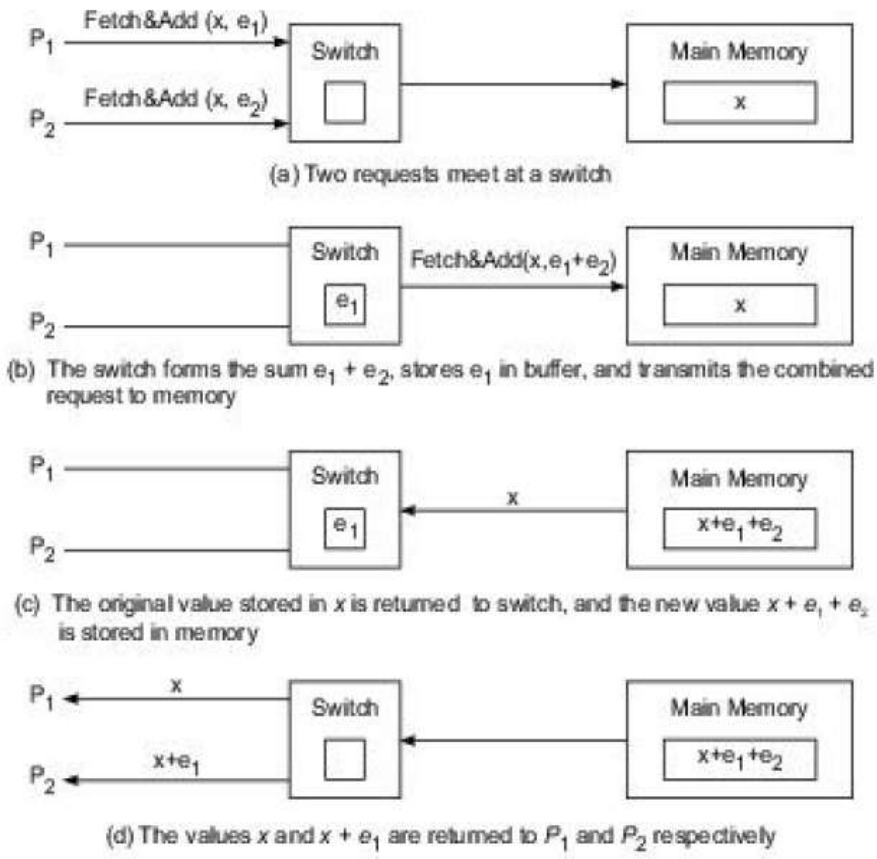
The net result is similar to a sequential execution of N Fetch & adds but is performed in **one indivisible operation.** Two simultaneous requests are combined in a switch as illustrated in **Fig.3.18.**

One of the following operations will be performed if processor P1 executes Ans1 <— fetch & add(x,e1) and P2 executes Ans2 <— fetch & add(x,e2)simultaneously on the shared variable x. If the request from P1 is executed before P2 , the following values are returned:

$Ans_1 \leftarrow x$

$Ans_2 \leftarrow x + e_1$

Regardless of the executing order, the value x +e1+e2 is stored in memory. It is the responsibility of the switch box to form the sum e1 + e2 , transmit the combined request Fetch &,Add(x,e1+e2), store the value e1 (e2) in a wait butter of the switch, and return the values x and x+e1, to satisfy the original requests Fetch &Add(x,e1) and Fetch &Add(x, e2), respectively.

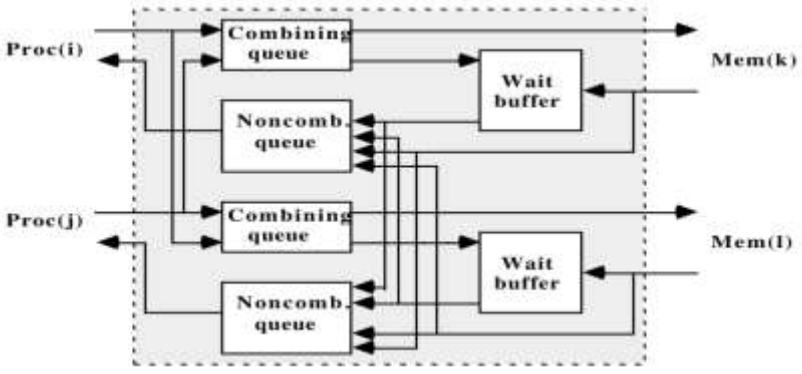


Two Fetch&Add operations are combined to access a shared variable simultaneously via a combining network

Fig:3.18

Just for reference

Structure of a combining switch.



Applications of combining networks

- The Fetch & Add primitive is very effective in accessing sequentially allocated queue structures in parallel, or in forking out parallel processes with identical code that operate on different data sets.

3.2.CACHE COHERENCE AND SYNCHRONIZATION MECHANISMS

Cache coherence protocols for coping with the multicache inconsistency problem are considered below.

- **Snoopy protocols** are designed for bus-connected systems.
- **Directory—based protocols** apply to network-connected systems.

3.2.1 Cache Coherence problem

- Cache and main memory may contain inconsistent copies of the same object. Multiple caches may possess different copies of the same memory block because multiple processors operate independently and asynchronously.
- **Cache coherence schemes** prevent this problem by maintaining a uniform state for each cached block of data.

Qn:What are the causes of cache inconsistency? Explain the mechanisms used for handling cache inconsistencies?

Qn:Explain any three sources of cache data inconsistency and discuss the possible protocols to overcome the cache data inconsistency problem?

- **Cache inconsistency can be caused due to**
    1. Inconsistency in data sharing
    2. Process migration
    3. Inconsistency due to I/O operations

➢ **Inconsistency in data sharing**

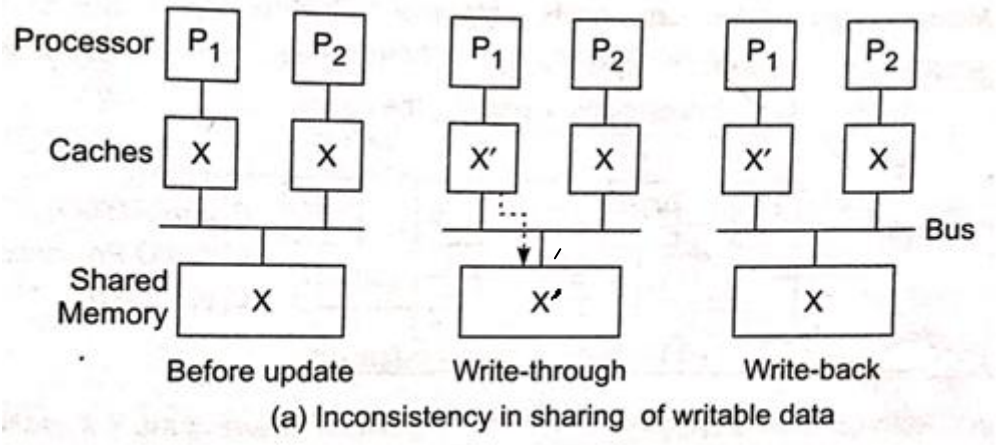


Fig:3.19

- Consider a multiprocessor with two processors, each using a private cache and both sharing the main memory. Let X be a shared data element which has been

referenced by both processors.

- Before update value of X is same in memory as well as both the caches
- In **write through policy** , if P1 updates X to X' the same copy will be written immediately to main memory. But inconsistency occurs between the cached copies

➢ In **write-back policy** the main memory will be updated only when the modified data in the cache is replaced or invalidated.so inconsistency occurs between the cached copy and shared memory.

Qn: Describe the inconsistencies caused by process migration?

➢ **Process Migration and I/O**

- ➢ Process migration means transferring of sufficient amount of the state of the process from, one processor to another processor.
- ➢ The process migration is done in the case of load sharing.(heavily loaded to lightly loaded processor)

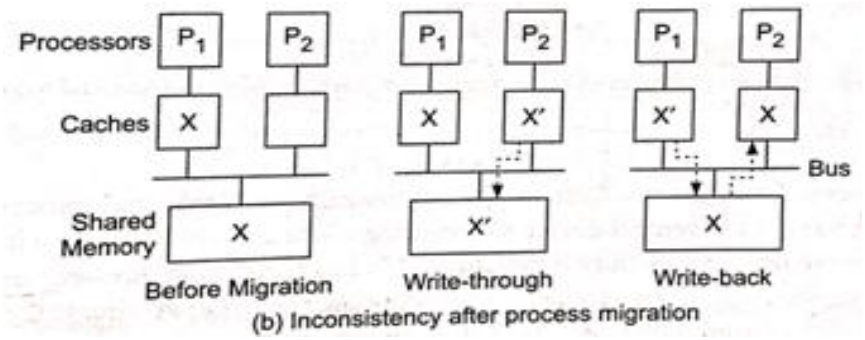


Fig:3.20

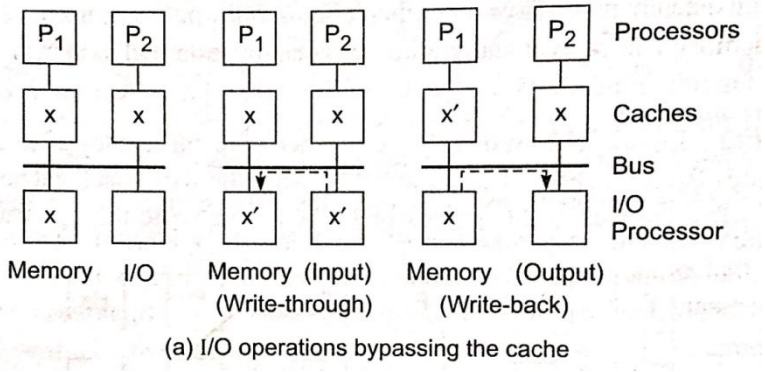
- Initially before migration, P1 have the value x in its cache and also x in shared memory.
- **Now x has been migrated to p2 and ,If p2 reads x and updates it to x'(Centre fig), in write through policy,** the value will be updated simultaneously in shared memory also. But in p1 its still x. so inconsistencies exists.

- Now in case of write back, If p1 reads the value x , modifies it to x' ,but it will not be updated in shared memory which is being read in the p2, while executing the process. So here the updated value is not used by the p2 to perform the particular task.. **(Right most figure)(P1 to p2).**

➢ **Inconsistency due to I/O operations.**

Qn: How can the I/O operations bypassing cache result in cache coherence problem? Also, suggest a solution for preventing it?

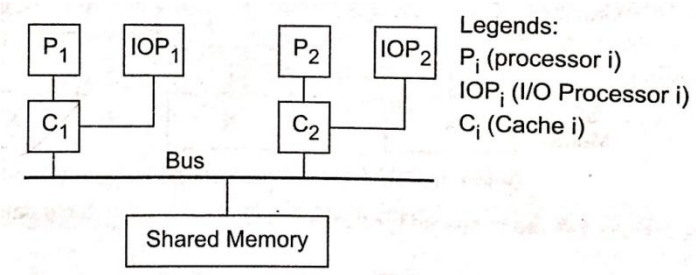
- Data movement from an I/o device to a shared memory usually does not cause the cached copies of data to be updated.
- As a result, an input operation that writes x will write data to the shared primary memory,causes it to become inconsistent with a cached value of x.
- Likewise, writing data to an I/O device usually use the data in the shared primary memory, ignoring any potential cached data with different values.



(a) I/O operations bypassing the cache

Fig:3.21

- When I/O processors loads a new data X' into the main memory (middle fig above) inconsistency occurs between caches and shared memory.
- When outputting a data directly from the shared memory(Bypassing the cache) also create inconsistency.
- Possible solution is to attach I/O processors to private caches(C1 and C2) as shown in fig below. This way I/O processors share caches with the CPU. The I/O consistency can be maintained if cache-to-cache consistency is maintained via the bus.



(b) A possible solution

Fig:3.22

TWO PROTOCOL APPROACHES to maintain Cache consistency

Qn:Explain any two cache coherence protocol ?

Qn:Describe the snoopy protocol used to ensure cachecoherence in multiprocessor systems

1. **Snoopy Bus Protocol** (used in commercial multiprocessors using bus based memory systems)

   A bus is a convenient device for ensuring cache coherence because it allows all processors in the system to observe ongoing memory transactions. If a bus transaction threatens the consistent state of a locally cached object, the **cache controller** can take appropriate actions to invalidate the local copy. Protocols using this mechanism to -ensure coherence are called snoopy protocol because each cache snoops on the transactions of other caches.
2. **Directory based Protocols** (used in scalable multiprocessor system interconnect processors)

3.2.2 SNOOPY BUS PROTOCOLS

- Processors with private cache tied to a common bus ,practices 2 approaches to maintain cache consistency

1. **Write Invalidate policy** – invalidates all remote copies when a local cache block is updated. In fig below processor P1 modifies(write) its cache from X to X' and all other copies are invalidated via the bus. Invalidated blocks are called dirty, means, they should not be used.

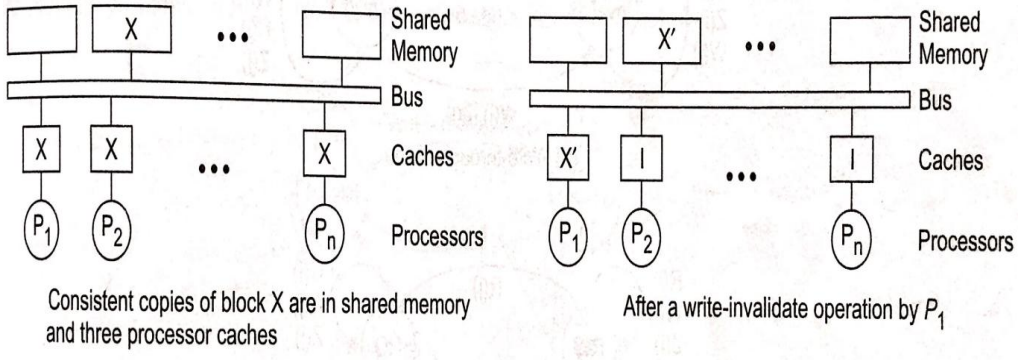
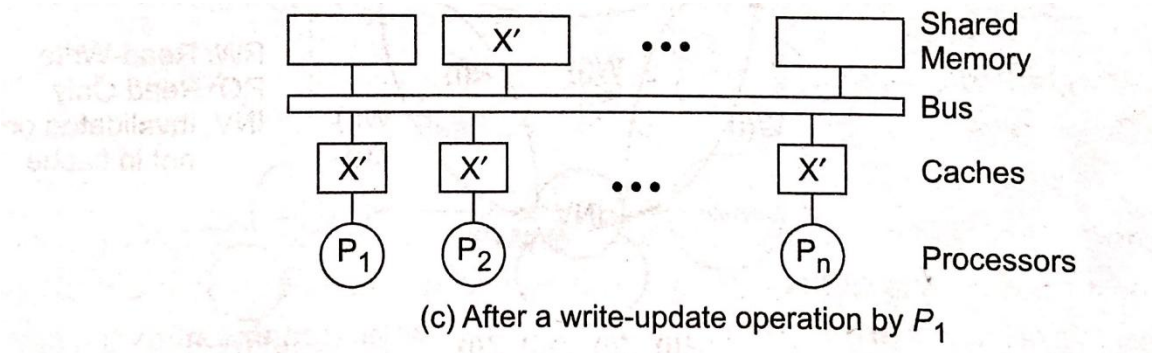


Fig:3.23

2. **Write Update policy –** broadcast the new data block to all caches containing a copy of the block. In fig below the new updated block content X' is broadcast to all cache copies via the bus. The memory copy is also updated if **write-through caches** are used, if **write-back caches** are used memory copy is updated later when the block is replaced



Write-invalidate and write-update coherence protocols for write through caches

Fig:3.24

Write- Through Caches:

- The states of a cache block copy change with respect to3 operations in the cache:-
  - **Read**
  - **Write**
  - **Replacement**
- A block copy of a **writethrough cache i** attached to **processor i** can assume one of two possible cache states:
  - **valid or**
  - **invalid**

• Write-Invalidate and Write-Through

Event	Actions
Read Hit	**Use the local copy from the cache.**
Read Miss	**Fetch a copy from global memory. Set the state of this copy to Valid.**
Write Hit	**Perform the write locally. Broadcast an Invalid command to all caches. Update the global memory.**
Write Miss	**Get a copy from global memory. Broadcast an invalid command to all caches. Update the global memory. Update the local copy and set its state to Valid.**
Replace	**Since memory is always consistent, no write back is needed when a block is replaced.**

In a valid state (Fig. above),

- all processors can read(R(i),R(j)) safely.
- Local processor i can also write (W(i)) safely.
-

The invalid state corresponds to :

- the case of the block either being invalidated or being replaced (Z(i) or Z(j))
- a write (w(j)) by a remote processor into its cache copy, makes all other copies become invalidated.

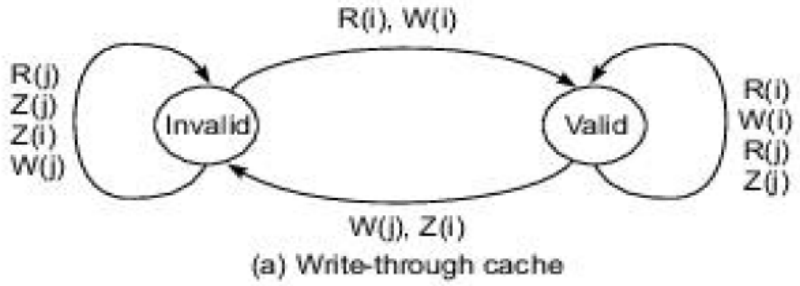


Fig 3.25:State transition graph for a cache block using write through ,write-invalidate snoopy protocol

Write-Back Caches:

The **valid state** of a write-back cache can be further split into two cache states,labelled:-

- **RW (read –write/ Exclusive) and**
- **RO (read only/ Shared)** as shown in Fig. below.

The **INV (invalidated or not-in-cache)** cache state is equivalent to the invalid state mentioned before.

This three-state coherence scheme corresponds to an **ownership protocol.**

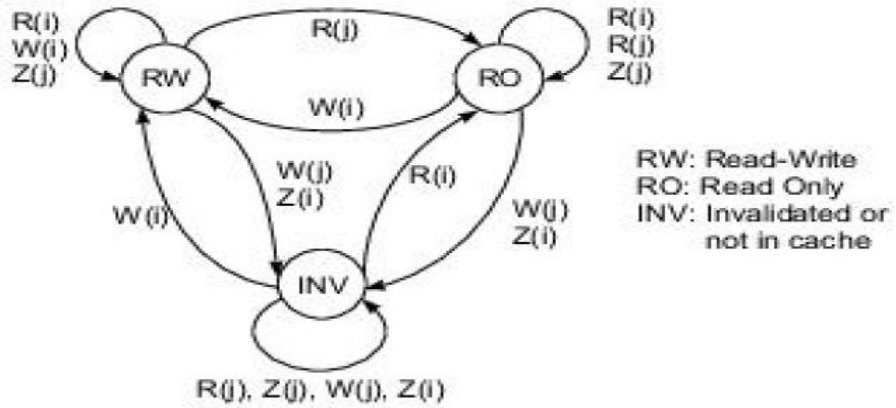


Fig:3.26 State transition graph for a cache block using write-back ,write-invalidate snoopy protocol

j denotes a **remote processor** ,where j!=i	
W[i] = Write to block by processor i**.** **(Write hit)**	**W[j]** = Write to block copy In cache j by processor ,j!=I (Write miss)
R[i] = Read block by processor i **(Read Hit)**	**R[j]** =read block copy In cache j by processor ,j!=i **(Read Miss-replacement action to be performed)**
Z[i] =Replace block in cache i **(Invalidation by local processor)**	**Z[j]** = Replace block copy in cache j!=i **(Write miss-action to be performed is replacement**

	by remote processor j)

- When the **memory owns a block,** caches can contain only the **RO(Shared)** copies of the block. In other words, multiple copies may exist in the RO state and every processor having a copy (called a keeper of the copy) can read(R(i)), R(j)) the copy safely.

- The **INV state** is entered whenever a remote processor writes (w(j)) its local copy or the local processor replaces (Z(i)) its own block copy.

- The **RW state(Exclusive)** corresponds to **only one cache copy existing in the entire system owned by the local processor i**. Read (R(i)) and write (w(i)) can be safely performed in the RW state

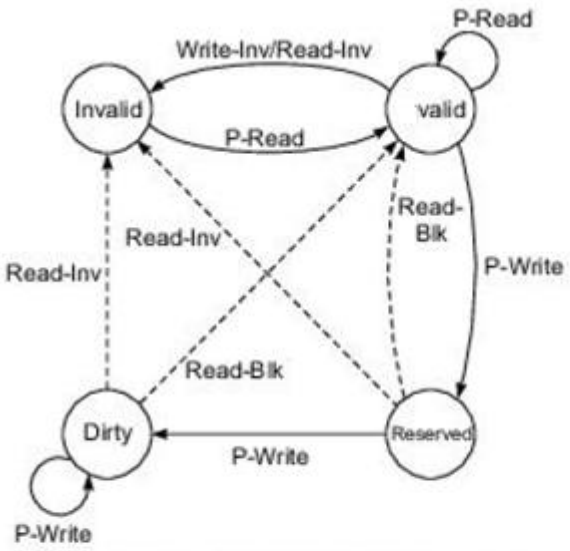
- . From either the RO state or the INV state, the cache block becomes uniquely owned when a local write **(w(i)** takes place.

- Write-Invalidate and Write-Back

Event	Action
Read Hit	**Use the local copy from the cache.**
Read Miss:	**If no Exclusive (Read-Write) copy exists, then supply a copy from global memory. Set the state of this copy to Shared (Read-Only). If an Exclusive (Read-Write) copy exists, make a copy from the cache that set the state to Exclusive (Read-Write), update global memory and local cache with the copy. Set the state to Shared (Read-Only) in both caches.**
Write Hit	**If the copy is Exclusive (Read-Write), perform the write locally. If the state is Shared (Read-Only), then broadcast an Invalid to all caches. Set the state to Exclusive (Read-Write).**
Write Miss	**Get a copy from either a cache with an Exclusive (Read-Write) copy, or from global memory itself. Broadcast an Invalid command to all caches. Update the local copy and set its state to Exclusive (Read-Write).**
Block Replacement	**If a copy is in an Exclusive (Read-Write) state, it has to be written back to main memory if the block is being replaced. If the copy is in Invalid or Shared (Read-Only) states, no write back is needed when a block is replaced.**

Write-once Protocol :-a cache coherence protocol for bus-based multiprocessors.

- This scheme combines the advantages of both write-through and write-hack invalidations.
- In order to reduce bus traffic, the very first write of a cache block uses a **write-through policy**.This will result in a consistent memory copy while all other cache copies are invalidated.
- After the first write, shared memory is updated using a **write-back policy**. This scheme can he described by the four-state transition graph shown in Fig. below The four cache states are defined below:



Goodman's write-once cache coherence protocol using the write invalidate policy on write-back caches (Adapted from James Goodman 1983, reprinted from Stenstrom, *IEEE Computer*, June 1990)

Fig: 3.27

Valid: The cache block, which is consistent with the memory copy, has been read from shared memory and has not been modified.

Invalid: The block is not found in the cache or is inconsistent with the memory copy.

Reserved: Data has been written exactly once since being read from shared memory. The cache copy is consistent with the memory copy, which is the only other copy.

Dirty: The cache block has been modified (written) more than once, and the cache copy is the only one in the system (thus inconsistent with all other copies).

To maintain consistency, the protocol requires two different sets of commands. The solid lines in Fig. above correspond to access commands issued by a local processor labeled ***read-miss, write-miss and write –hit***. Whenever a ***read-miss occurs, the valid state*** is entered.

- The first ***write-hit*** leads to the ***reserved state***.
- The second ***write-hit*** leads to the ***dirty state,***
- and all future ***write-hits*** stay in the ***dirty state***.
- Whenever a ***write-miss*** occurs, the eache block enters the *dirty state*.

The dashed lines correspond to invalidation commands issued by remote processors via the snoopy bus.
The ***read-invalidate command*** reads a block and invalidates all other copies. The ***write-invalidate command*** invalidates all other copies of a block.

Cache Events and Action: The memory-access and invalidation commands trigger the following events and actions:

Read-Miss: When a processor wants to read a block that is not in the cache, a ***read-miss*** occurs. A bus read operation will be initiated. If ***no dirty copy*** exists, then main memory has a consistent copy and supplies a copy to the requesting cache. If a ***dirty copy*** does exist in a remote cache, that cache will inhibit the main memory and send a copy to the requesting cache. In all cases, the cache copy will enter the ***valid state*** after a rcad-miss.

Write-Hit: If the copy is in the ***dirty or reserved state***, the write can be carried out locally and the new state is ***dirty.*** If the new state is ***valid***, a ***write-invalidate*** command is broadcast to all caches, invalidating their copies. The shared memory is written through, and the resulting state is reserved after this first write.

Write-miss: When a processor fails to write in a local cache, the copy must come either from the **main memory or from a remote cache with a dirty block**. This is accomplished by sending a read-invalidate command which will invalidate all cache copies. The local copy is thus updated and ends up in a dirty state.

Read-hit: Read-hits can always be performed in a local cache without causing a state transition or using the snoopy bus for invalidation.

Block Replacement: If a copy is dirty, it has to be written back to main memory by block replacement. If the copy is clean (i.e., in either the valid, reserved, or invalid state), no replacement will take place.

Protocol Performance issues

- Snoopy Cache Protocol **Performance determinants**:
  - Workload Patterns
  - Implementation Efficiency
- **Goals/Motivation** behind using snooping mechanism
  - Reduce bus traffic

  - Reduce effective memory access time
- **Data Pollution** Point
  - Miss ratio decreases as block size increases, up to a **data pollution point** (that is, as blocks become larger, the probability of finding a desired data item in the cache increases).
  - The miss ratio starts to increasing as the block size increases to data pollution point.
- **Ping-Pong** effect on data shared between multiple caches
  - If two processes update a data item alternately, data will continually migrate between two cacheswith high miss-rate

3.2.3 DIRECTORY BASED PROTOCOLS

Qn:Advantage of Directory based over Snoopy bus protocol ?

Qn: Differentiate between central and distributed directory ?

- **Write invalidate** scheme for Snoopy Bus causes heavy bus traffic since it invalidates cache copies on each update.
- **Write update scheme** for Snoopy bus updates all cache copies, some of which may never be used.
- Hence in large multiprocessors with hundreds of processors broadcasting becomes expensive. This leads to **Directory based protocols.**

Directory Structure

- In multistage network, Cache coherence is supported by using **cache directories** to store information on where copies of cache block reside. Directory can be **Central or Distributed**
- Directory scheme using a **Central Directory** stores duplicates of all cache directories. Contention and long search time are the drawbacks in using central directory..
- In **Distributed directory** scheme **Each memory module maintains a separate directory** which records t**he state and presence information** for each memory block. The state information is local, but the presence information indicates which caches have a copy of the block.
-

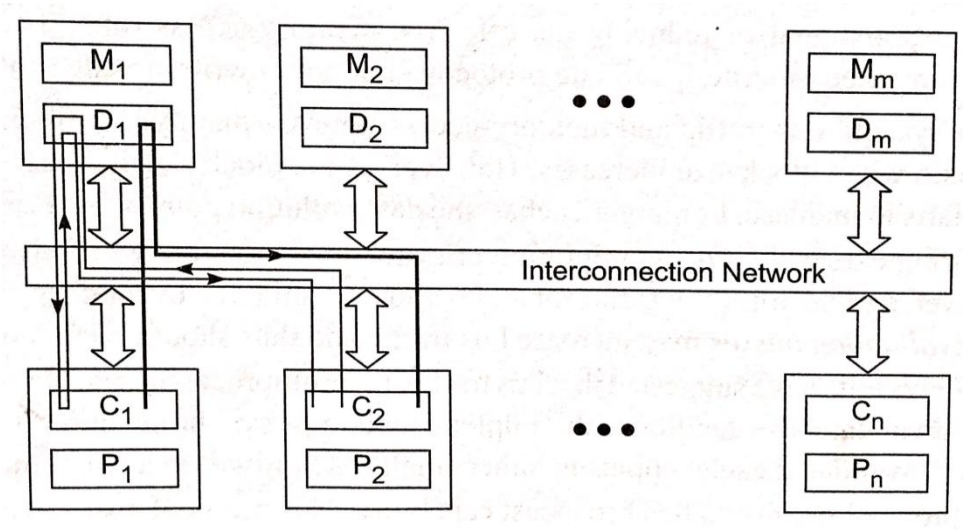


Fig: 3.28: Basic concept of a directory based cache coherence scheme

- In fig above a ***read-miss (thin lines)*** in cache 2 results in a request sent to memory module. The memory controller retransmits request to dirty copy in cache 1. This cache ***writes back*** its copy. The memory module can supply a copy to the requesting cache.
- In the case of a ***write-hit*** at cache 1 (bold lines) a command is send to the memory controller which sends invalidation to all caches (cache 2) marked in t**he presence vector residing in the directory D1.**
- A directory entry for each block of data contains **a number of pointers to specify the locations of copies of the block.** Each directory entry also contains **a dirty bit to specify whether a particular cache has permission to Write the associated block of data.**

Qn:Explain Full map directory based protocol?

3 different types of directory protocols

1 Full Map directories:- Full-map directories store enough data associated with each block in global memory so that every cache in the system can simultaneously store a copy of any block of data. That is. each directory entry contains N pointers, where N is the number of processors in the system.

2. Limited directories;- they have a fixed number of pointers per entry, regardless of the system size.

3. Chained Directories:-They emulate the full-map schemes by distributing the directory among the caches

1. **Full Map Directories**
    - **Directory entry** contains one bit per processor and a dirty bit. Each bit represents the status of the block in corresponding processor's

cache(present or absent)

- If dirty bit is set, then only one processor's bit is set and that processor can write into the block.
- A cache maintains 2 bits per block. One bit indicates whether block is valid, and other indicates whether a valid block may be written.

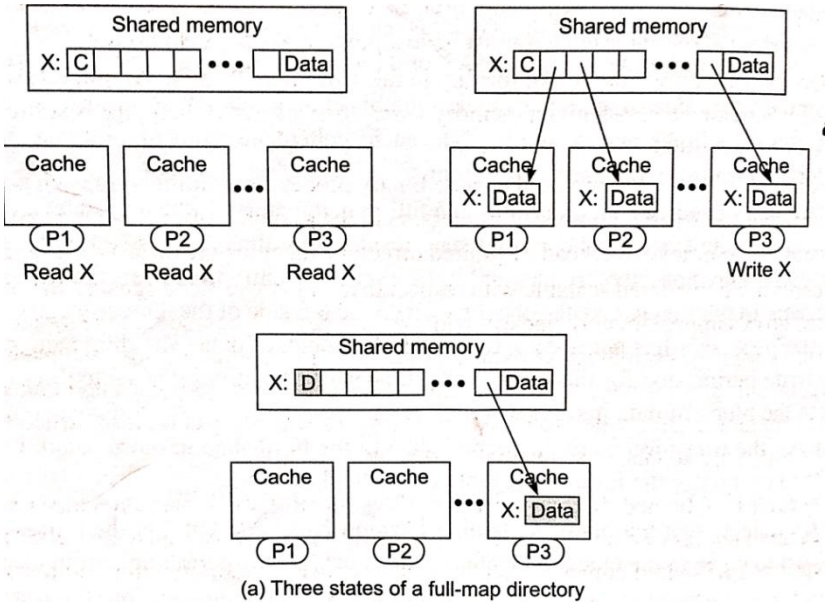


Fig: 3.29

- In above fig, **in first state**, location X is missing in all caches in the system.
- The **second state** results from three cache (C1, C2 and C3) requesting copies of location X. Three pointers (processor bits) are set in the entry to indicate the caches that have copies of the block of data.
- In first two states the dirty bit on left side of the directory entry is set to clean (C), indicating that no processor has permission to write to the block of data.
- The **third state results** from cache C3 requesting write permission for the block. In the final state the dirty bit is set to dirty (D) and there is a single pointer to the block and data in cache.
- ***Memory consumed by Directory is proportional to size of memory O(N) multiplied by the size of the directory O(N). Thus the total memory overhead is square of number of processors O(N^2)***

Let us examine the transition from the second state to the third state in more detail. Once processor P3 issues the write to cache C3, the following events will take place:

(1) Cache C3 detects that the block containing location X is valid but that the processor does not have permission to write to the block, indicated by the block's write-permission bit in the cache.

(2) Cache C3 issues a write request to the memory module containing location X and stalls processor P3.

(3) The memory module issues invalidate requests to caches Cl and C2.

(4) Caches Cl and C2 receive the invalidate requests, set the appropriate bit to indicate that the block containing location X is invalid and send acknowledgments back to the memory module.

(5) The memory module receives the acknowledgments, sets the dirty hit, clears the pointers to caches Cl and C2, and sends write permission to cache C3.

(6) Cache C3 receives the write permission message, updates the state in the cache, and reactivates processor P3.

The memory module waits to receive the acknowledgments before allowing processor P3 to complete its write transaction. By waiting for acknowledgments, the protocol guarantees that the memory system ensures sequential consistency.

Qn:Explain the term Eviction?

2. Limited Directories

- Designed to **Solves the directory size problem** by restricting the number of simultaneously cached copies of any particular block of data limits the growth of the directory to a constant factor.
- A directory protocol can be classified as $Dir_i\ X$ using the notation from Agarwal et al (1988). **The symbol i stands for the number of pointers**, and X is **NB for a scheme with no broadcast** A full-map scheme without broadcast is represented as $Dir_N\ NB$. A limited directory protocol that uses i< N pointers is denoted $Dir_i\ NB$.

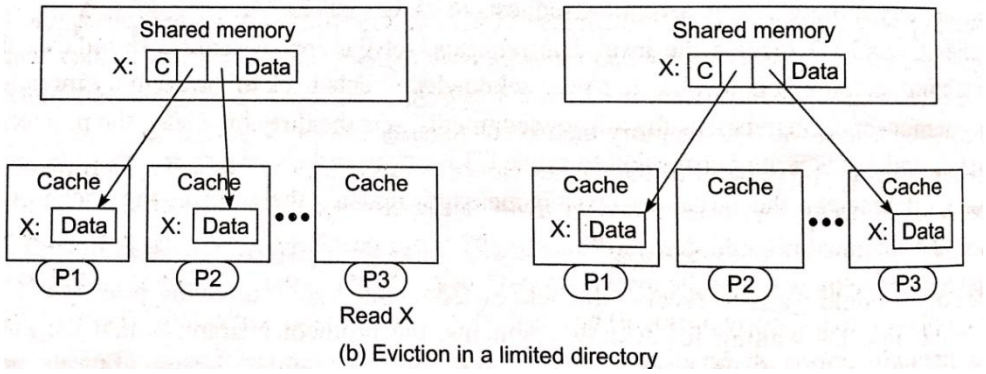


Fig: 3.30

- the above fig shows situation when three cache request read copies in a memory system with Dir_2 NB protocol(only 2 copies are allowed). The two pointer directory can be viewed as a two-way set

associative cache.

- When cache C3 requests copy of location X, the memory module must invalidate the copy in either C1 or C2 cache. This process of pointer replacement is called **eviction.**

Qn: Describe gossip protocol?

3. Chained Directories

- Limited directories have a restriction on the number of shared copies of data blocks.(due to the limitation of pointers).
- Chained directories will not have a restriction on the number of shared copies of data blocks
- They Keeps track of shared copies of **data by maintaining chain of directory pointers.(Singly linked chain)**

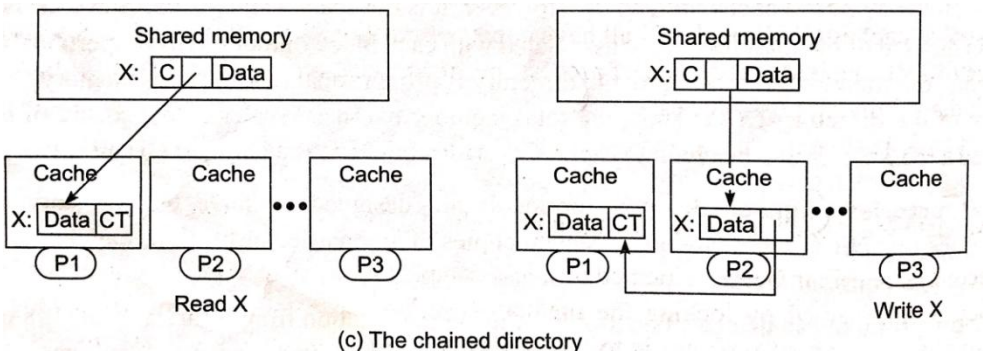


Fig: 3.31

- In the fig above **singly linked chain,** initially assume there are no shared copies of location X.
- Processor P1 reads location X, the memory sends a copy to cache C1 along with **chain termination (CT) pointer**. The memory also keeps a pointer to cache C1.
- Subsequently, When processor C2 reads location X, the memory sends a copy to cache C2, along with the pointer to cache C1. The memory then keeps a pointer to cache C2..

- By repeating the above step, all of the caches can cache a copy of the location X.
- If processor P3 writes to location X, it is necessary to send a **data invalidation message down the chain.**
- To ensure **sequential consistency**, the memory module denies processor P3 write permission until the **processor with the chain termination pointer acknowledges the invalidation of the chain**.
- Perhaps this scheme should be called a ***gossip protocol*** (as opposed to a snoopy protocol) because information is passed from individual to individual rather than being spread by covert observation.

If processor P3 write to location X, data invalidation message has to be send down the chain. Also called as gossip protocol

Cache Design Alternatives:

Qn:Distinguish between private caches and shared caches?

Beyond the **use of private caches**, three design alteratives are suggested below. Each of the design alternatives has its own advantages and shortcomings. There exists insufiicient evidence to determine whether any of the alternatives is always better or worse than the use of private caches.

Shared Cache :

An alternative approach to maintaining cache coherence is to completely eliminate the problem by using shared caches attached to shared-memory modules. No private caches are allowed in this case. This approach will reduce the main memory access time but contributes very little to reducing the overall memory-access time and to resolving access conflicts.

Non-cacheable Data :

Another approach is **not to cache shared writable data.** Shared data are non cacheable, and **only instructions or private data are cacheable** in local caches.
Shared data include locks, process queues. and any other data structures protected by critical sections.
The compiler must tag data as either cacheable or non cacheable. Special hardware tagging must be used to distinguish them.

Cache Flushing :

A third approach is to use cache flushing every time a synchronization primitive is executed. This may work well with transaction processing multiprocessor systems. Cache flushes are slow unless special hardware is used.
Flushing can be made very selective by the compiler in order to increase efficiency. Cache flushing at synchronization, I/O and process migration may be carried out unconditionally or selectively.

3.2.4 Hardware Synchronization Mechanisms

Synchronization is a special form of communication in which control information is exchanged, instead of data, between communicating process residing in the same or

different processors. Synchronization enforces correct sequencing of processors and ensures mutually exclusive access to shared writable data.

Multiprocessor systems use **hardware mechanisms** to implement low-level or primitive synchronization operations**(Discused Below).** or use **software (operating system) level** synchronization mechanisms such as Semaphores or monitors.

Hardware mechanisms to implement synchronisation are:

- **Atomic Operations**
- **Wired Barrier Synchronization**

Atomic Operations

- Atomi**c operations such as *read, write, or read-modify-write* can be used to implement** some synchronization primitives.
- some **interprocessor interrupts** can be used for synchronization purposes.
- For example , **Test & Set and Reset (Lock)** primitives are defined below:

```
Test&Set  (lock)
          temp ← lock;   lock ← 1;
          return temp
```

```
Reset  (lock)
          lock ← 0
```

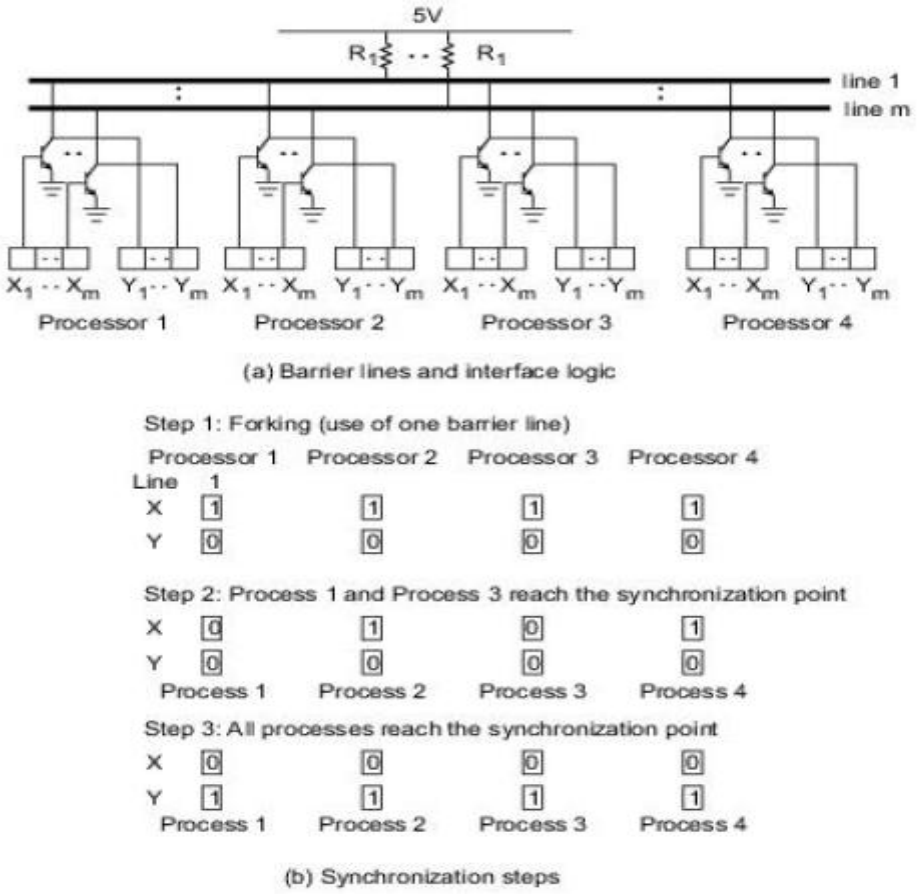
  - To synchronize concurrent processes, the software may repeat Test&.Set until the returned value (temp) becomes 0.
  - This **synchronization primitive may tie up some bus cycles while a processor enters busy-waiting on the spin lock.**
  - To avoid spinning, **interprocessor interrupts** can be used.
  - Lock tied to an interrupt is called a suspended lock.

Wired Barrier Synchronization

- Concurrent processes residing in different processors can be synchronized rising **barriers**. A barrier can be implemented by a shared-memory word which keeps counting the number of processes reaching the barrier.
- **After all processes have updated the barrier count**, the synchronization point has been reached.
- No processor can execute beyond the barrier until the synchronization process is complete.
    - Each processor uses a **dedicated control vector** X = (X1, X2, ..., Xm) and
    - accesses a common **monitor vector** Y = (y1, Y2, ...ym) in shared memory, where m corresponds to the barrier lines used.
    - Each control bit X, is connected to the **base (input)** of a probing transistor.
    - The monitor bit yi checks the **collector voltage (output)** of the transistor.
    - **The wired-NOR connection implies that line i will be high (1) only if control bits Xi, from all processors are low (0).**
    - This demonstrates the ability to use the control bit Xi to signal the completion of a process on processor i.
    - **The bit Xi is set to 1 when a process is initiated and reset to 0 when the process finishes its execution.**

- When all processes finish their jobs, the xi bits from the participating processors are all set to 0; and the barrier line is then raised to high (1), signalling the synchronization barrier has been crossed.
- This timing is watched by all processors through snooping on the Yi bits. **Thus only one barrier line is needed to monitor the initiation and completion of a single synchronization involving many concurrent processes.**

Figure below shows the synchronization of four processes residing on four processors **using one barrier line.**

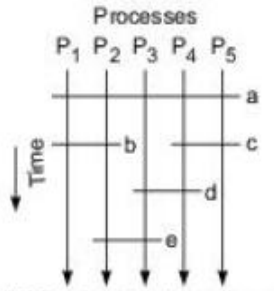


The synchronization of four independent processes on four processors using one wired-NOR barrier line (Adapted from Hwang and Shang, *Proc. Int. Conf. Parallel Processing*, 1991)

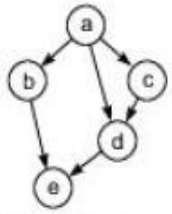
Fig: 3.32

Another example using multiple barrier lines to monitor several synchronisation points is shown below: **(Optional)**

If the synchronization pattern is predicted after compile time, then one can follow the precedence graph of a partially ordered set of processes to perform multiple synchronization as demonstrated in Fig. 7.20.



(a) Synchronization patterns



(b) Precedence graph

Step 0: Initializing the control vectors (use 5 barrier lines)

	Processor 1	Processor 2	Processor 3	Processor 4	Processor 5
X	1 1 0 0 0	1 1 0 0 1	1 0 0 1 1	1 0 1 1 0	1 0 1 0 0
Y	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0
Step 1: Synchronization at barrier a					
X	0 1 0 0 0	0 1 0 0 1	0 0 0 1 1	0 0 1 1 0	0 0 1 0 0
Y	1 0 0 0 0	1 0 0 0 0	1 0 0 0 0	1 0 0 0 0	1 0 0 0 0
Step 2a: Synchronization at barrier b					
X	0 0 0 0 0	0 0 0 0 1	0 0 0 1 1	0 0 1 1 0	0 0 1 0 0
Y	1 1 0 0 0	1 1 0 0 0	1 1 0 0 0	1 1 0 0 0	1 1 0 0 0
Step 2b: Synchronization at barrier c					
X	0 0 0 0 0	0 0 0 0 1	0 0 0 1 1	0 0 0 1 0	0 0 0 0 0
Y	1 1 1 0 0	1 1 1 0 0	1 1 1 0 0	1 1 1 0 0	1 1 1 0 0
Step 3: Synchronization at barrier d					
X	0 0 0 0 0	0 0 0 0 1	0 0 0 0 1	0 0 0 0 0	0 0 0 0 0
Y	1 1 1 1 0	1 1 1 1 0	1 1 1 1 0	1 1 1 1 0	1 1 1 1 0
Step 4: Synchronization at barrier e					
X	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0
Y	1 1 1 1 1	1 1 1 1 1	1 1 1 1 1	1 1 1 1 1	1 1 1 1 1

(c) Synchronization steps

Fig. 7.20 The synchronization of five partially ordered processes using wired-NOR barrier lines (Adapted from Hwang and Shang, *Proc. Int. Conf. Parallel Processing*, 1991)

Fig: 3.33

Here five processes (P_1, P_2, ..., P_5) are synchronized by snooping on five barrier lines corresponding to five synchronization points labeled a, b, c, d, e. At step 0 the control vectors need to be initialized. All five processes are synchronized at point a. The crossing of barrier a is signaled by monitor bit Y_1, which is observable by all processors.

Barriers b and c can be monitored simultaneously using two lines as shown in steps $2a$ and $2b$. Only four steps are needed to complete the entire process. Note that only one copy of the monitor vector Y is maintained in the shared memory. The bus interface logic of each processor module has a copy of Y for local monitoring purposes as shown in Fig. 7.20c.