CS405
Computer System Architecture

MODULE 3

Syllabus

- 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

Multi processor System Interconnects

- Parallel processing demands the use of efficient system interconnects for fast communication among multiple processors and shared memory, I/O, and peripheral devices.
- Hierarchical buses, crossbar switches and multistage networks are often used for this purpose.

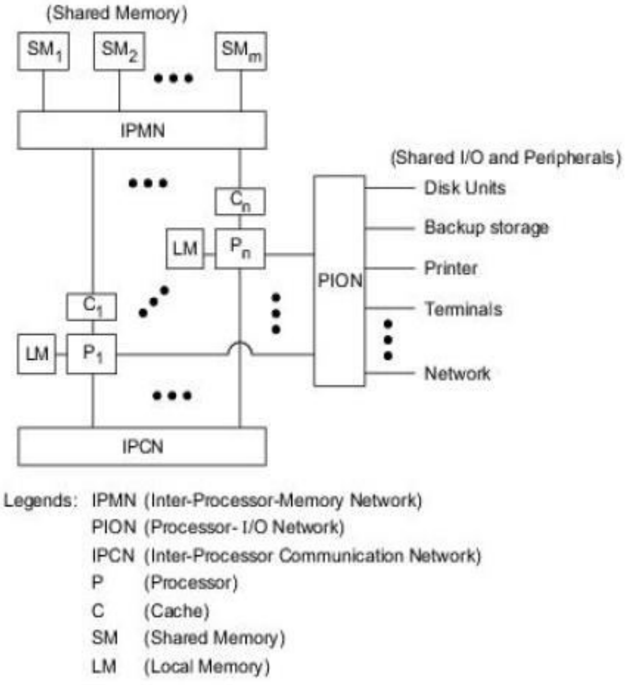
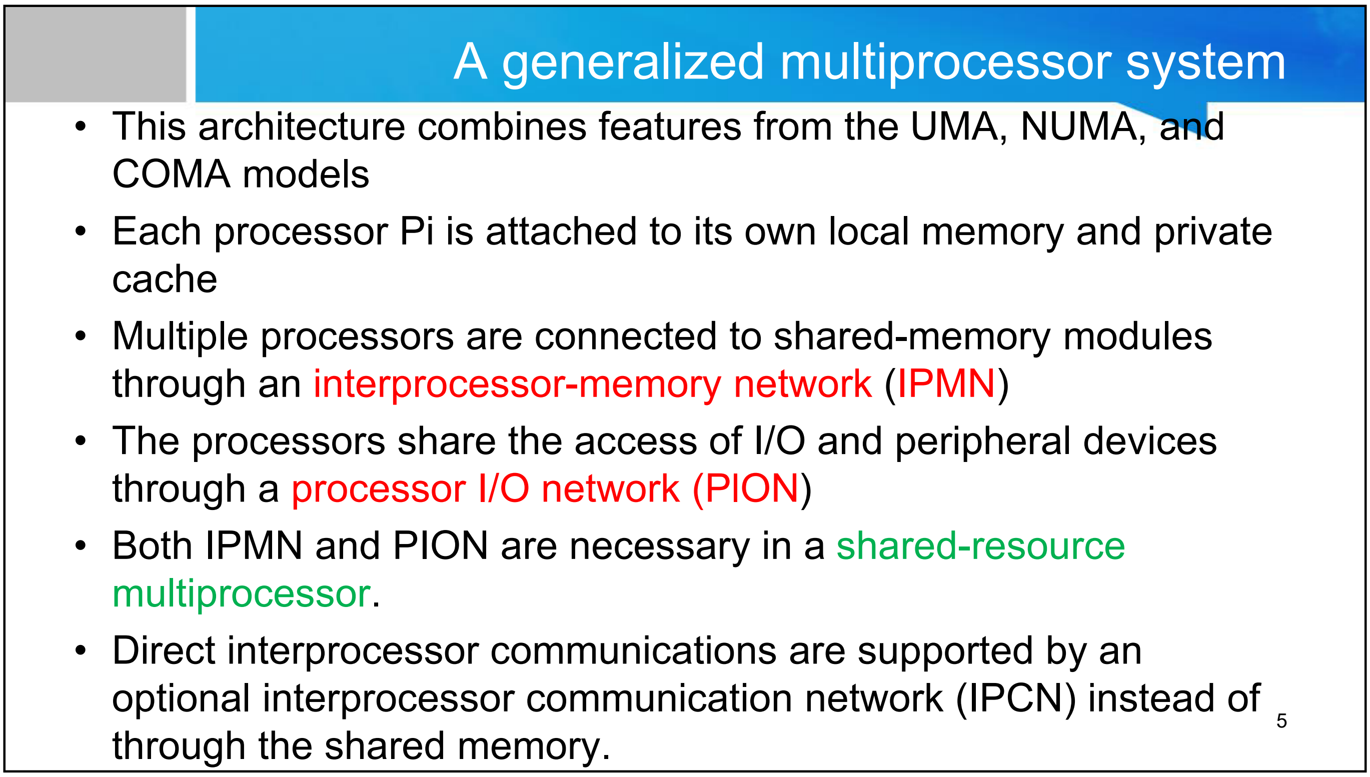


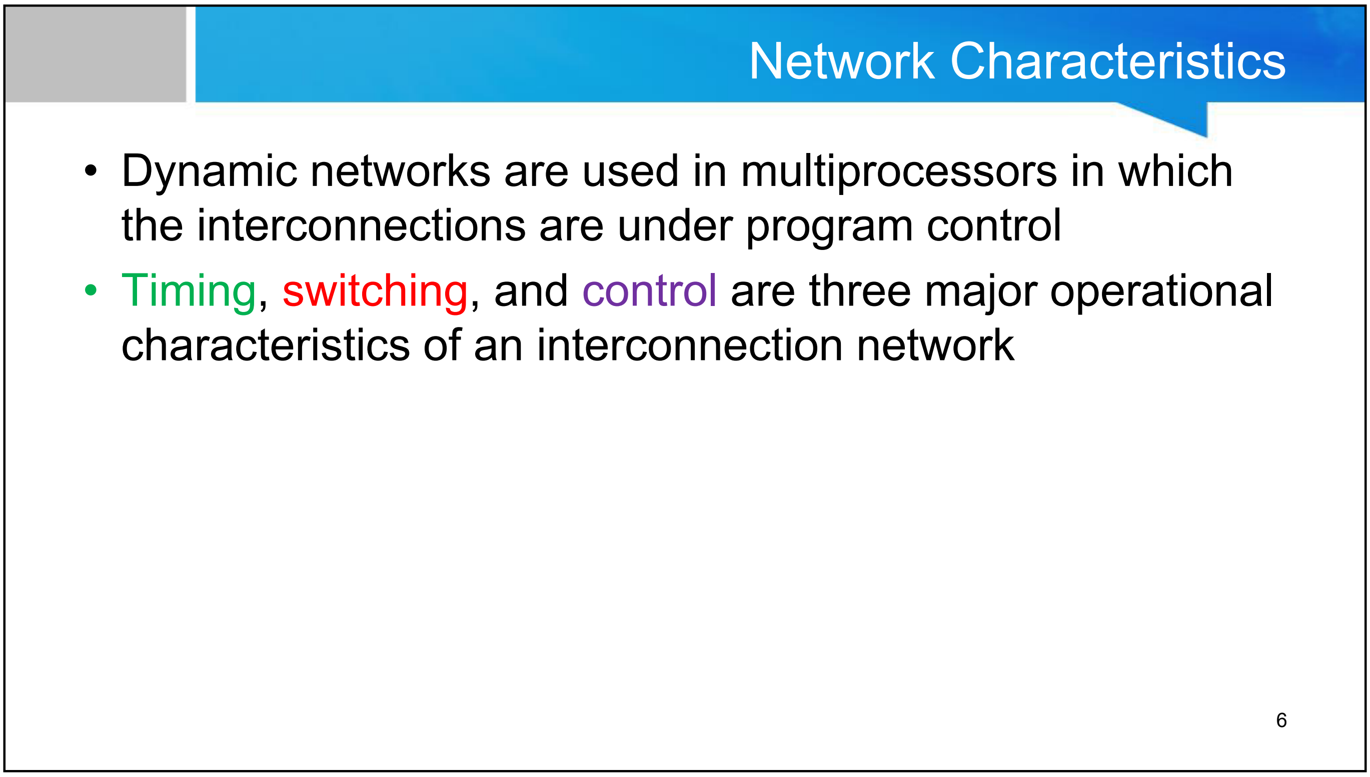
Fig. 7.1 Interconnection structures in a generalized multiprocessor system with local memory, private caches, shared memory, and shared peripherals

A generalized multiprocessor system



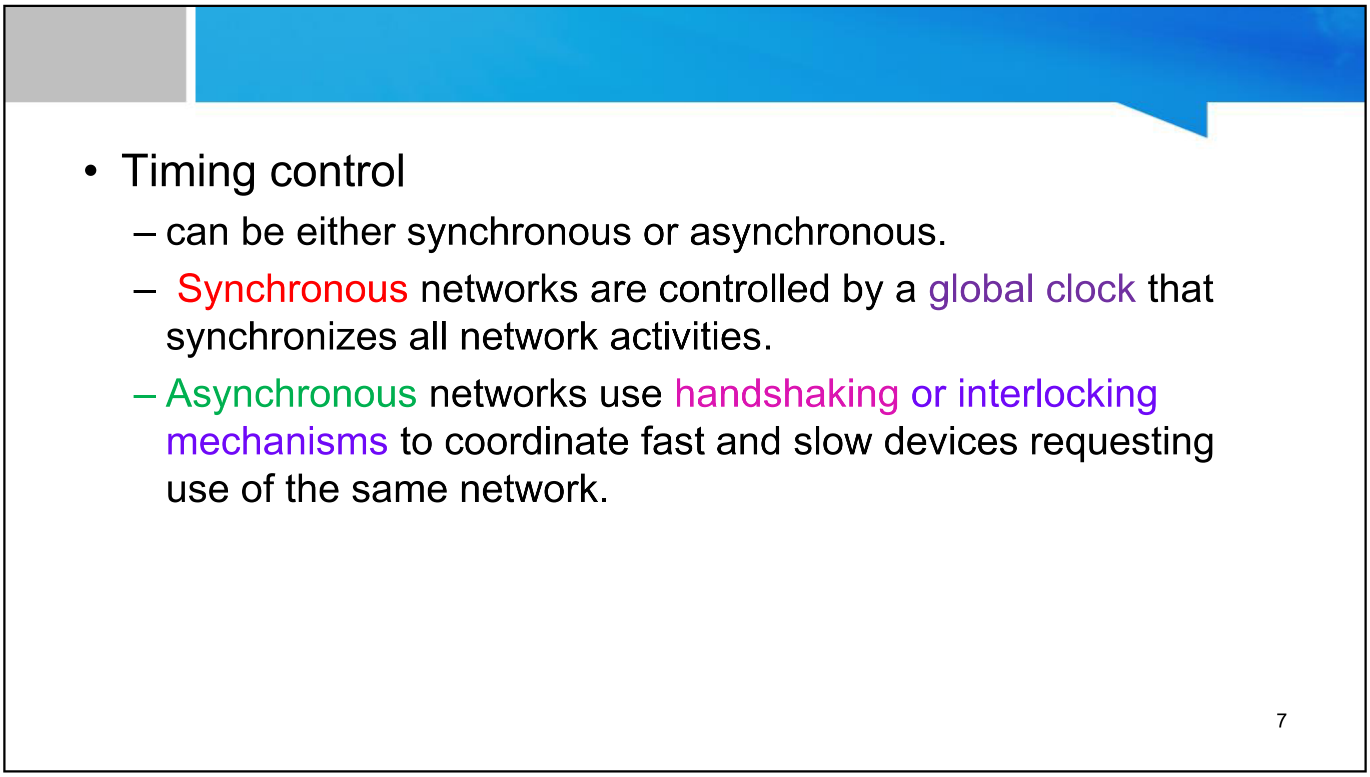
A generalized multiprocessor system

- This architecture combines features from the UMA, NUMA, and COMA models
- Each processor Pi is attached to its own local memory and private cache
- Multiple processors are connected to shared-memory modules through an interprocessor-memory network (IPMN)
- The processors share the access of I/O and peripheral devices through a processor I/O network (PION)
- Both IPMN and PION are necessary in a shared-resource multiprocessor.
- Direct interprocessor communications are supported by an optional interprocessor communication network (IPCN) instead of through the shared memory.

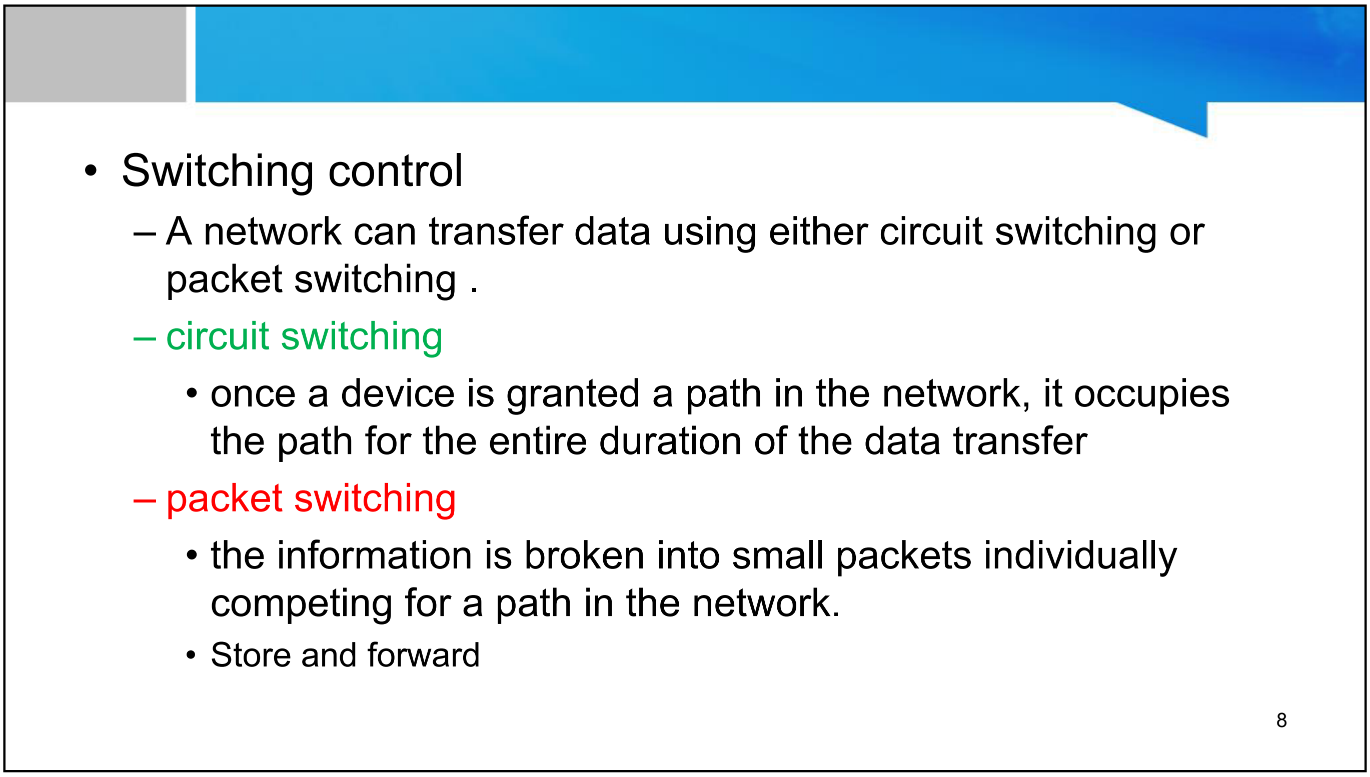


Network Characteristics

- Dynamic networks are used in multiprocessors in which the interconnections are under program control
- Timing, switching, and control are three major operational characteristics of an interconnection network



- Timing control
  - can be either synchronous or asynchronous.
  - Synchronous networks are controlled by a global clock that synchronizes all network activities.
  - Asynchronous networks use handshaking or interlocking mechanisms to coordinate fast and slow devices requesting use of the same network.



- Switching control
  - A network can transfer data using either circuit switching or packet switching .
  - 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.
    - Store and forward

- Network control strategy
  - classified as centralized or decentralized .
  - centralized control
    - a global controller receives requests from all devices attached to the network and grants the network access to one or more requesters.
  - distributed system
    - requests are handled by local devices independently.

Hierarchical Bus Systems

- A bus systems consists of a hierarchy of buses connecting various system and subsystem components in a computer.
- Each bus is formed with a number of signal, control, and power lines.
- Different buses are used to perform different interconnection functions

Hierarchical Bus Systems

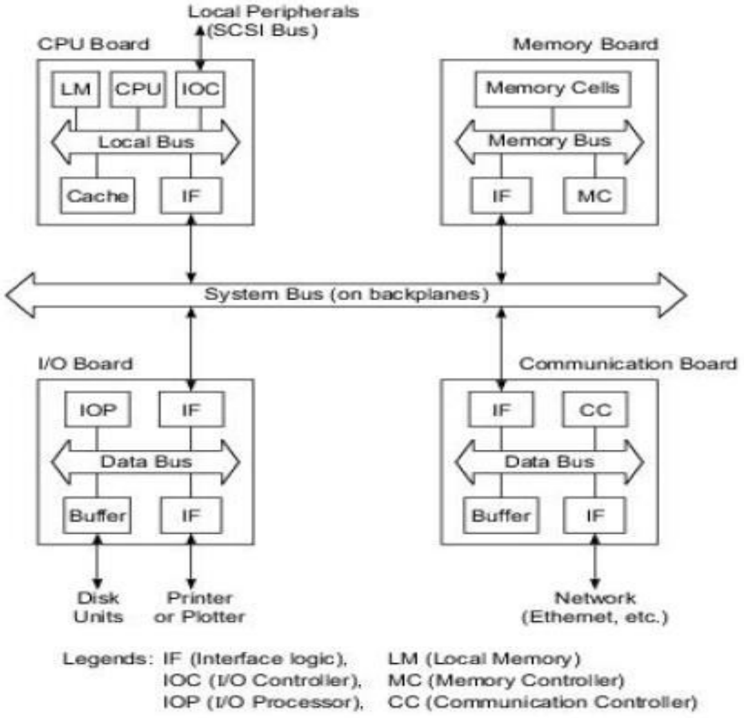


Fig. 7.2 Bus systems at board level, backplane level, and I/O level

- IF (Interface logic)
- LM (Local Memory)
- IOC (I/O Controller)
- MC (Memory Controller)
- IOP (I/O Processor)
- CC (Communication Controller)

- Local Bus
  - Buses implemented Within processor chips or on printed circuit boards are called local buses.
  - On a processor board one may find a local bus which provides a common communication path among major components(chips) mounted on the board.
- Memory bus
  - A memory board uses a memory bus to connect the memory with the interface logic.
- Data bus
  - An I/O or network interface chip or board uses a data bus.
- Each of these local buses consists of signal and utility lines

Backplane Bus

- A Backplane is a printed circuit on which many connectors are used to plug in functional boards.
- A system bus consisting of shared signal paths and utility lines, is built on the backplane.
- This system bus provides a common communication path almong all plug-in boards
- Eg: VME bus, Multibus II, Futurebus+

I/O Bus

- Input/output devices are connected to a computer system through an I/O bus such as the SCSI (Small Computer Systems Interface) bus.
- This bus is made of coaxial cables with taps connecting disks, printer, and other devices to a processor through an I/O controller.

Hierarchical Buses and Cache

- A hierarchical cache/bus architecture for designing a scalable multiprocessor

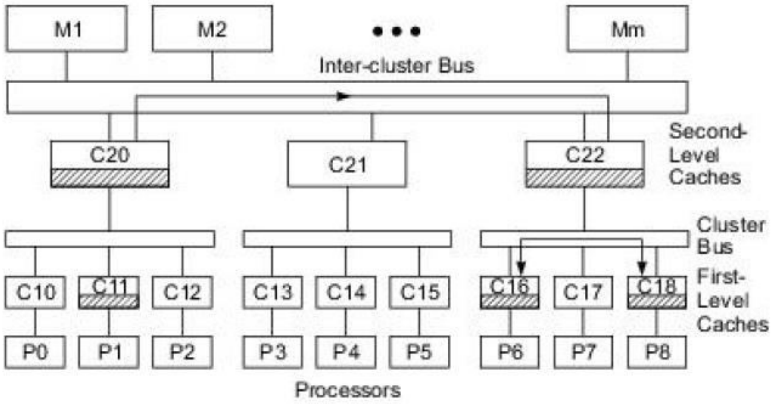


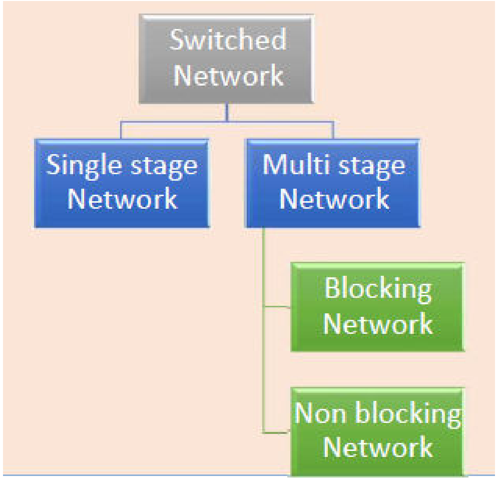
Fig. 7.3 A hierarchical cache/bus architecture for designing a scalable multiprocessor (Courtesy of Wilson; reprinted from *Proc. of Annual Int. Symp. on Computer Architecture, 1987*)

- This is a multilevel tree structure in which the leaf nodes are processors and their private caches (denoted Pj, and C_{1j}).
- These are divided into several clusters, each of which is connected through a cluster bus.
- An intercluster bus is used to provide communications among the clusters.
- Second level caches (denoted as C2,) are used between each cluster bus and the intercluster bus.
- Each second-level cache must have a capacity that is at least an order of magnitude larger than the sum of the capacities of all first-level caches connected beneath it.

- 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.
- The upper-level caches form another level of shared memory between each cluster and the main memory modules connected to the intercluster bus.
- Most memory requests should be satisfied at the lower-level caches.
- Inter cluster cache coherence is controlled among the second level caches and the resulting effects are passed to the lower level.

Crossbar Switch & Multiport Memory

- Switched networks provide dynamic interconnections between the inputs and outputs
- Crossbar networks are mostly used in small or medium-size systems.
- The multistage networks can be extended to larger systems if the increased latency problem can be suitably addressed



Switched Network

Single stage Network

Multi stage Network

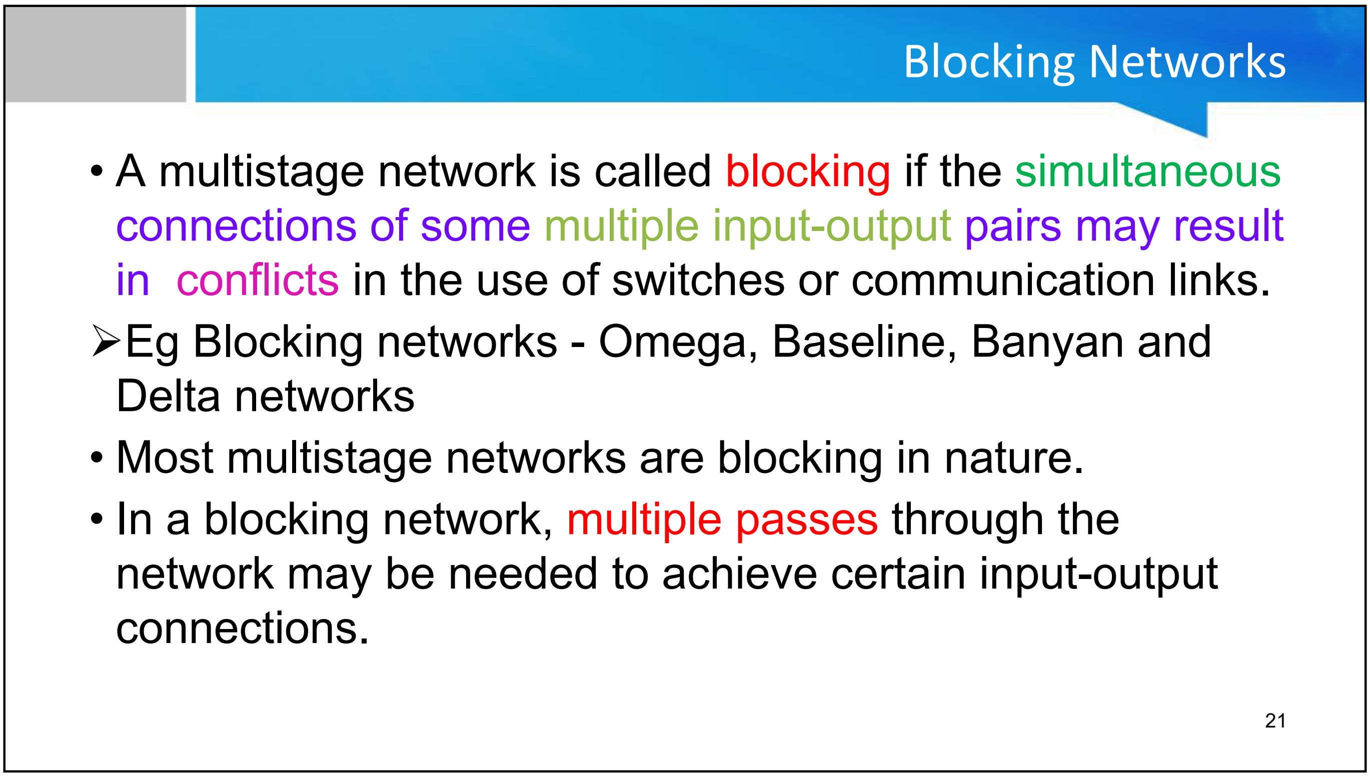
Blocking Network

Non blocking Network

Network Stages

- Depending on the interstage connections used:
- ➢Single-stage network :
  - also called a recirculating network because data items may have to recirculate through the single stage many times before reaching their destination.
  - A single-stage network is cheaper to build, but multiple passes may be needed to establish certain connections.
  - Eg : crossbar switch and multiport memory organization

- ➢Multistage network :
  - –consists of more than one stage of switch boxes.
  - –Such a network should be able to connect from any input to any output.
  - –The choice of interstage connection patterns determines the network connectivity.
  - –These patterns may be the same or different at different stages, depending the class of networks to be designed.
  - –Eg: Omega network, Flip network, and Baseline networks

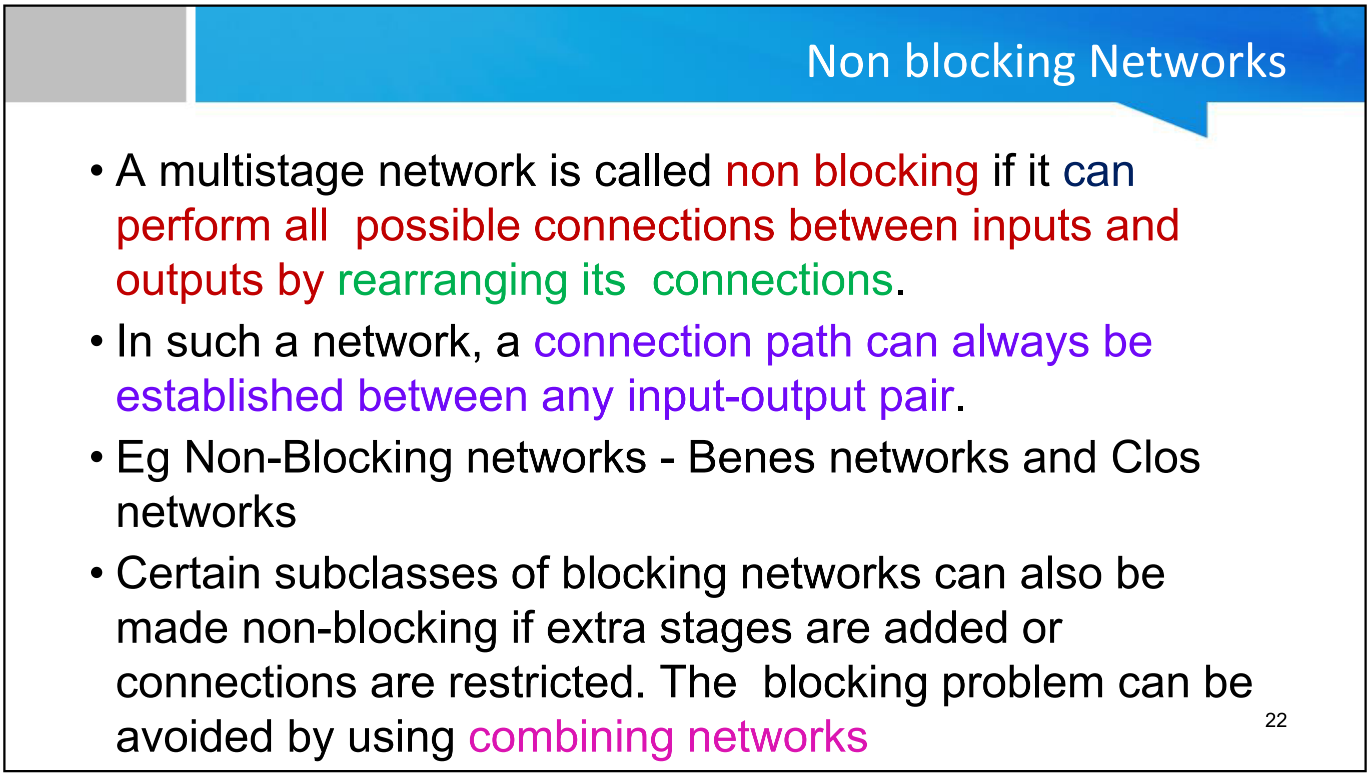


Blocking Networks

- 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.

➢Eg Blocking networks - Omega, Baseline, Banyan and Delta networks

- Most multistage networks are blocking in nature.
- In a blocking network, multiple passes through the network may be needed to achieve certain input-output connections.



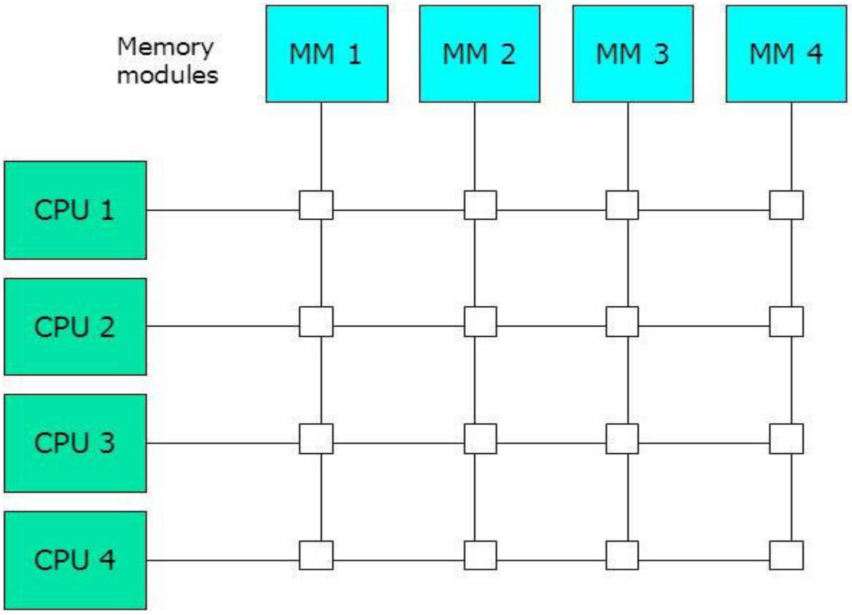
Non blocking Networks

- A multistage network is called non blocking if it can perform all possible connections between inputs and outputs by rearranging its connections.
- In such a network, a connection path can always be established between any input-output pair.
- Eg Non-Blocking networks - Benes networks and Clos networks
- Certain subclasses of blocking networks can also be made non-blocking if extra stages are added or connections are restricted. The blocking problem can be avoided by using combining networks

Crossbar Networks

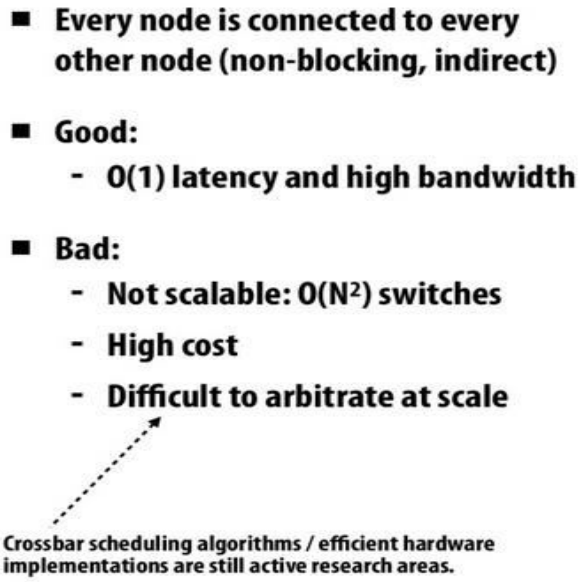
- In a crossbar network, every input port is connected to a free output port through a crosspoint switch without blocking.
- A crossbar network is a single-stage network built with unary switches at the crosspoints.
- Once the data is read from the memory, its value is returned to the requesting processor along the same crosspoint switch.
- In general, such a crossbar network requires the use of nxm crosspoint switches.
- A square crossbar (n=m) can implement any of the n! permutations without blocking.

Crossbar switch



- A crossbar switch network is a singlestage, nonblocking, permutation network.
- Each crosspoint in a crossbar network is a unary switch which can be set open or closed, providing a point-to-point connection path between the source and destination.
- 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

Pros and Cons - Crossbar Networks



- **Every node is connected to every other node (non-blocking, indirect)**
- **Good:**
  - **O(1) latency and high bandwidth**
- **Bad:**
  - **Not scalable: $O(N^2)$ switches**
  - **High cost**
  - **Difficult to arbitrate at scale**

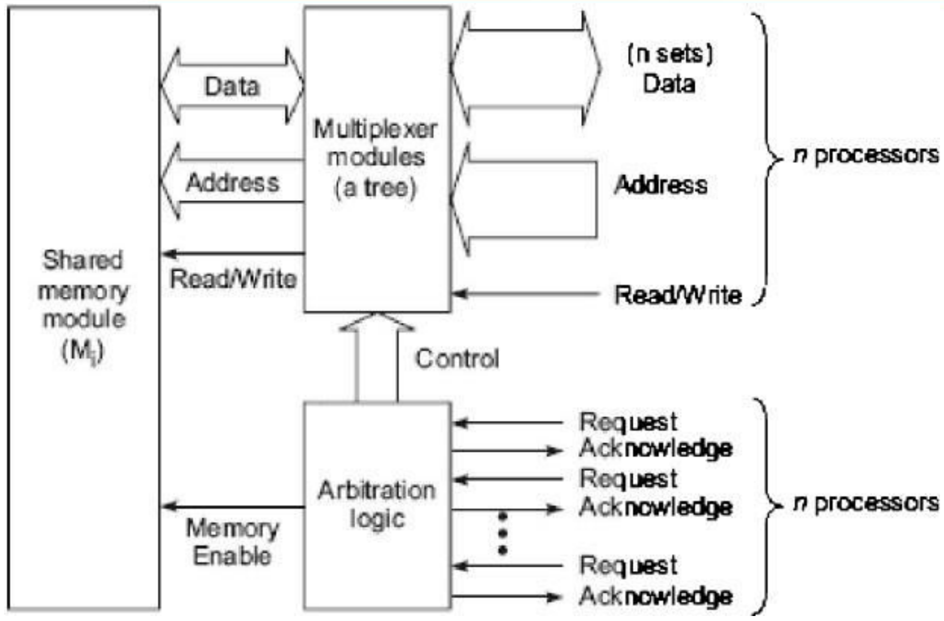
Crossbar scheduling algorithms / efficient hardware implementations are still active research areas.

Crosspoint Switch Design(Operations)

- Out of n crosspoint switches in each column of an n x m crossbar mesh, only one can be connected at a time.
- To resolve the conflict for each memory module, each crosspoint switch must be designed with extra hardware.
- Each crosspoint switch requires the use of a large number of connecting lines accommodating address, data path, and control signals.
  - ➢Means each crosspoint has a complexity matching of a bus of same width

- For an n x n crossbar network, n^2 sets of crosspoint switches and a large number of lines are needed
- 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.
- 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



Cont..

- n sets of data, address, and read/write lines are connected to the input of the multiplexer tree.
- 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.

Cont..

- 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. Note that the data path established is bidirectional, in order to serve both read and write requests for different memory cycles.

Crossbar Limitations

- A single processor can send many requests to multiple memory modules.
- For an n x n crossbar network, at most n memory words can be delivered to at most n processors in each cycle.
- The crossbar network offers the highest bandwidth of n data transfers per cycle.
- Since all necessary switching and conflict resolution logic are built into the crosspoint switch, the processor interface and memory port logic are much simplified and cheaper.
- A crossbar network is cost-effective only for small multiprocessors.
- A single-stage crossbar network is not expandable once it is built.
- Redundancy or parity-check lines can be built into each crosspoint switch to enhance the fault tolerance and reliability of the crossbar network.

Crossbar Summary

- An NXN Crossbar allows all N inputs to be connected simultaneously to all N outputs
- It allows all one-to-one mappings, called permutations.
- No. of permutations = N!
- When two or more inputs request the same output, it is called CONFLICT. Only one of them is connected and others are either dropped or buffered
- When processors access memories through crossbar, this situation is called memory access conflicts

MULTIPORT MEMORY

Hierarchical Buses and Cache

- A hierarchical cache/bus architecture for designing a scalable multiprocessor

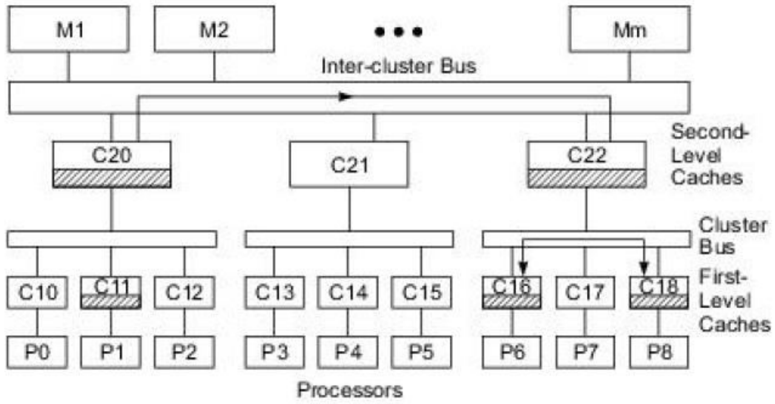
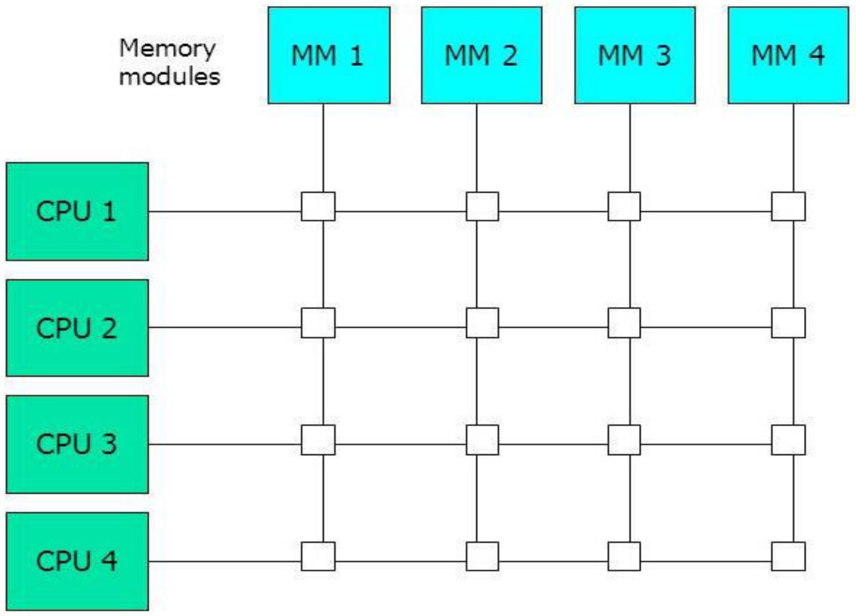


Fig. 7.3 A hierarchical cache/bus architecture for designing a scalable multiprocessor (Courtesy of Wilson; reprinted from *Proc. of Annual Int. Symp. on Computer Architecture, 1987*)

Crossbar switch



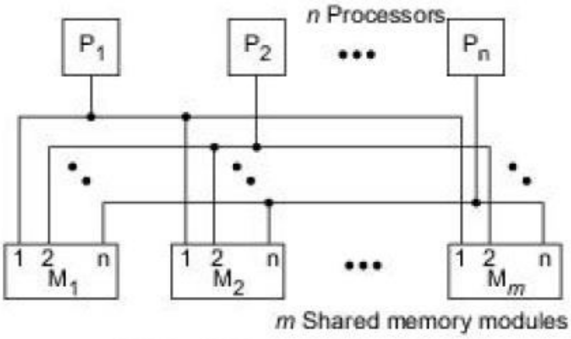
MULTIPORT MEMORY

- Building crossbar network into large system is cost prohibitive some mainframe multiprocessors use a multiport memory organization

The idea is to move all crosspoint arbitration and switching functions associated with each memory module into the memory controller.

- Thus the memory module becomes more expensive due to the added access ports and associated logic.

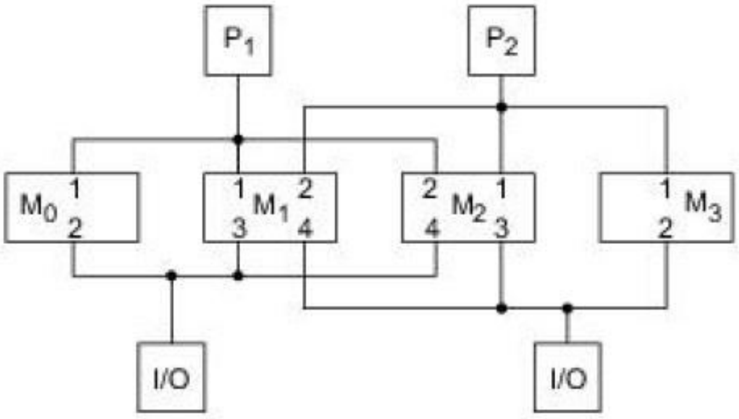
- The circles in the diagram represent n switches tied to n input ports of a memory module.
- Only one of n processor requests can be honored at a time.
- The multiport memory organization is a low-cost, low-performance bus system than a high-cost, high-bandwidth crossbar system.
- The contention bus is timeshared by all processors and device modules attached.
- The multiport memory must resolve conflicts among processors.



(a) n-port memory modules used

- This memory structure becomes expensive when m and n become large.
- A typical mainframe multiprocessor configuration may have n = 4 processors and m = 16 memory modules.
- A multiport memory multiprocessor is not scalable because once the ports are fitted, no more processors can be added without redesigning the memory controller
- Another drawback is the need for a large number of interconnection cables and connectors when the configuration becomes large

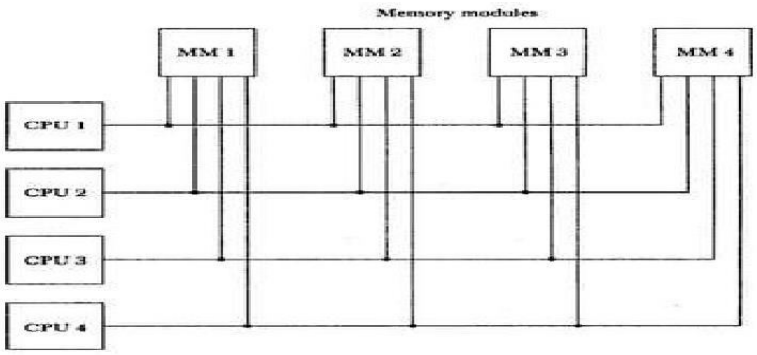
- Memory access conflicts are resolved by assigning fixed priorities to each memory port



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

Summary

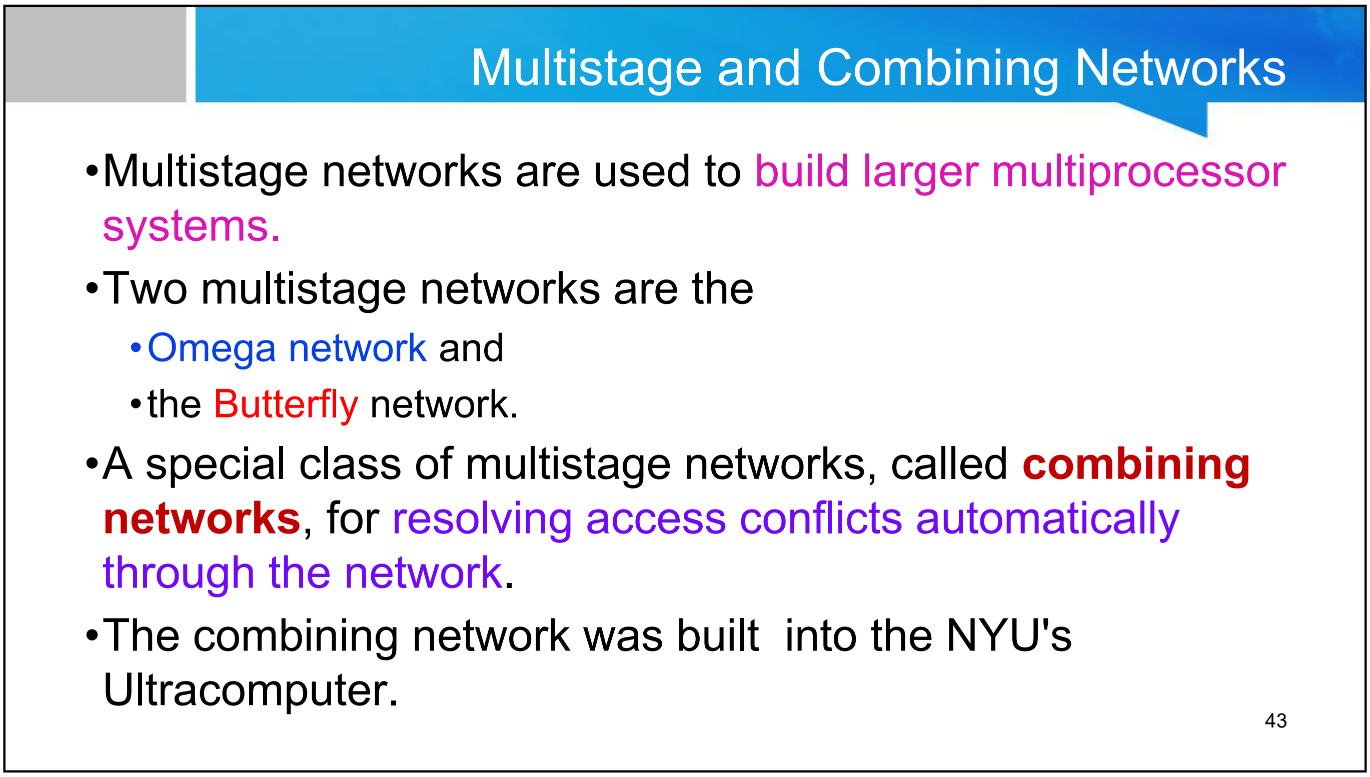
- Requires the most expensive memory units since most of the control and switching circuitry is included in the memory unit
- A large number of cable and connections are required
- Memory access conflicts are resolved by assigning fixed priorities to each memory port
- There is a potential for a very high total transfer rate in the overall system.



COMPUTER SYSTEMS ARCHITECTURE

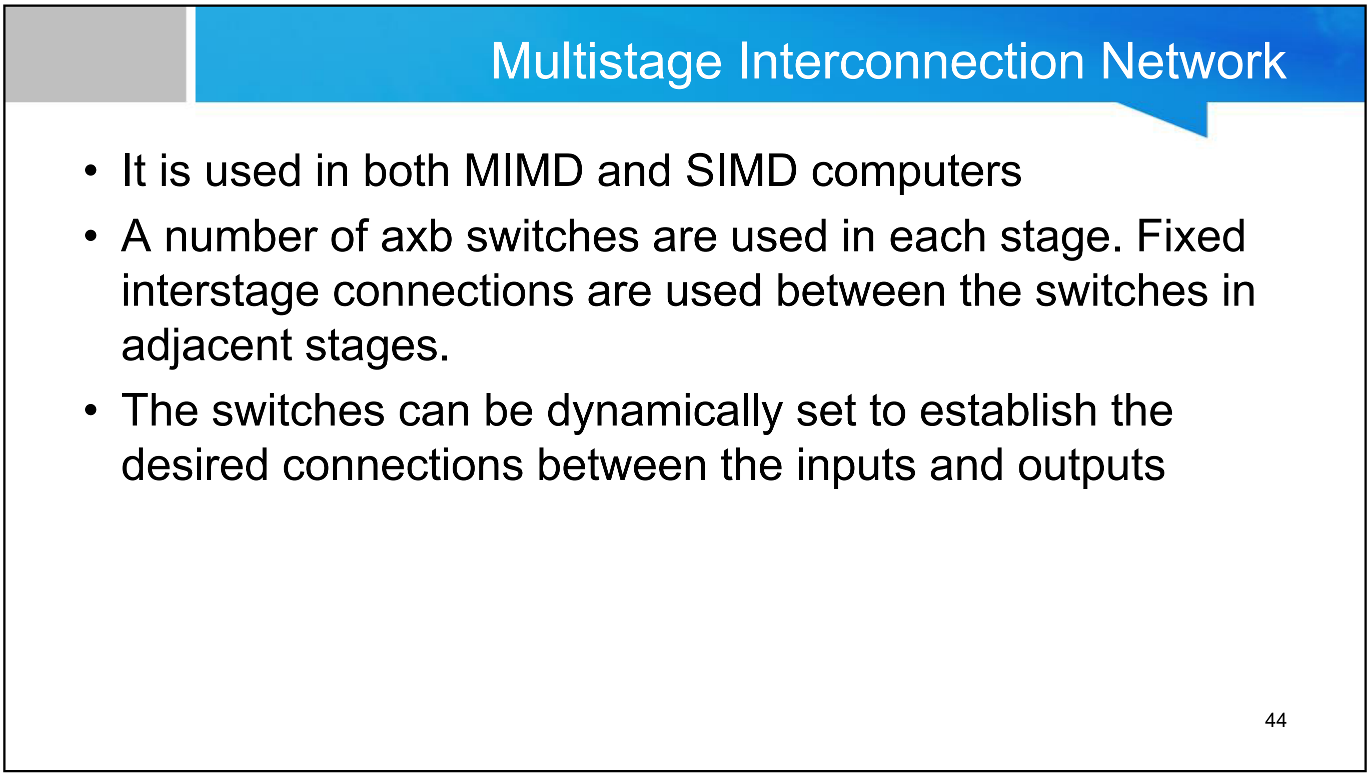
OMEGA NETWORKS (Malayalam)

Introduction to Multistage and Combining Networks



Multistage and Combining Networks

- Multistage networks are used to build larger multiprocessor systems.
- Two multistage networks are the
  - Omega network and
  - the Butterfly network.
- A special class of multistage networks, called **combining networks**, for resolving access conflicts automatically through the network.
- The combining network was built into the NYU's Ultracomputer.



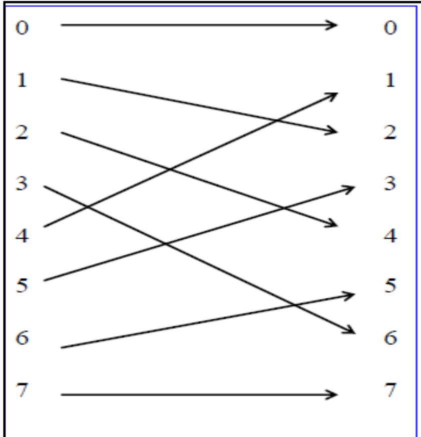
Multistage Interconnection Network

- It is used in both MIMD and SIMD computers
- A number of axb switches are used in each stage. Fixed interstage connections are used between the switches in adjacent stages.
- The switches can be dynamically set to establish the desired connections between the inputs and outputs

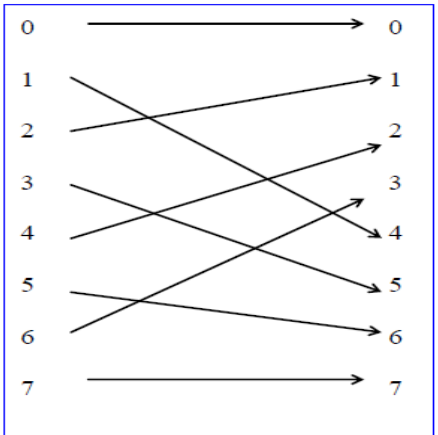
Omega Network

- The omega network is an example of a banyan multistage interconnection network that can be used as a switch fabric.
- The omega differs from the delta network in the pattern of interconnections between the stages
- The omega MIN uses the "perfect shuffle"
  - This means that the connections at each stage represent the movement of a deck of cards divided into 2 equal decks and then shuffled together, with each card from one deck alternating with the corresponding card from the other deck.

What is Shuffle?



(a) Perfect Shuffle



(b) Inverse Perfect Shuffle

shuffle interconnection => rotate left

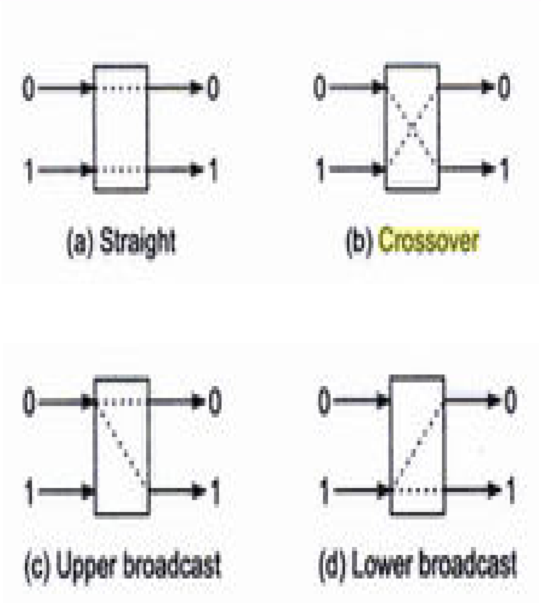
$$S(a_{n-1}\, a_{n-2} \dots a_1\, a_0) = (a_{n-2}\, a_{n-3} \dots a_0\, a_{n-1})$$

Example: 000 ---> 000 ---> 000 ---> 000
001 ---> 010 ---> 100 ---> 001

Omega networks

- They are constructed using 2x2 switches
- An n input omega network has $\log_2 n$ stages of 2x2 switches
- Each stage require n/2 switch modules
- Each switch module is individually controlled
- Possible connections of 2x2 switches
  - Straight
  - Crossover
  - Upper broadcast
  - Lower broadcast

Omega Network

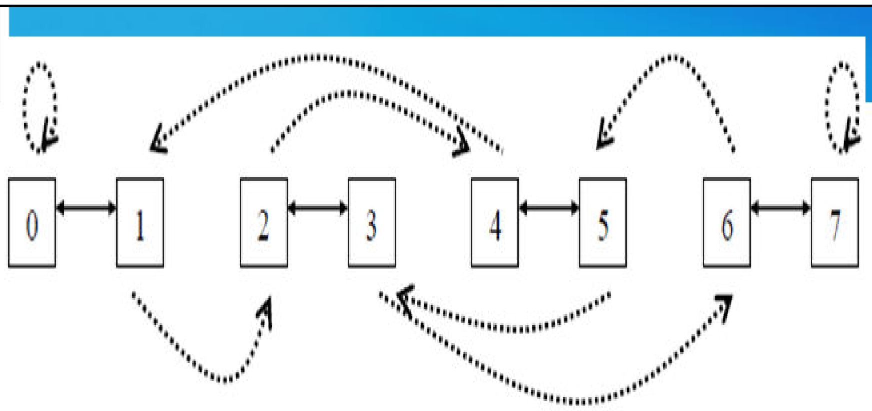


(a) Straight

(b) Crossover

(c) Upper broadcast

(d) Lower broadcast

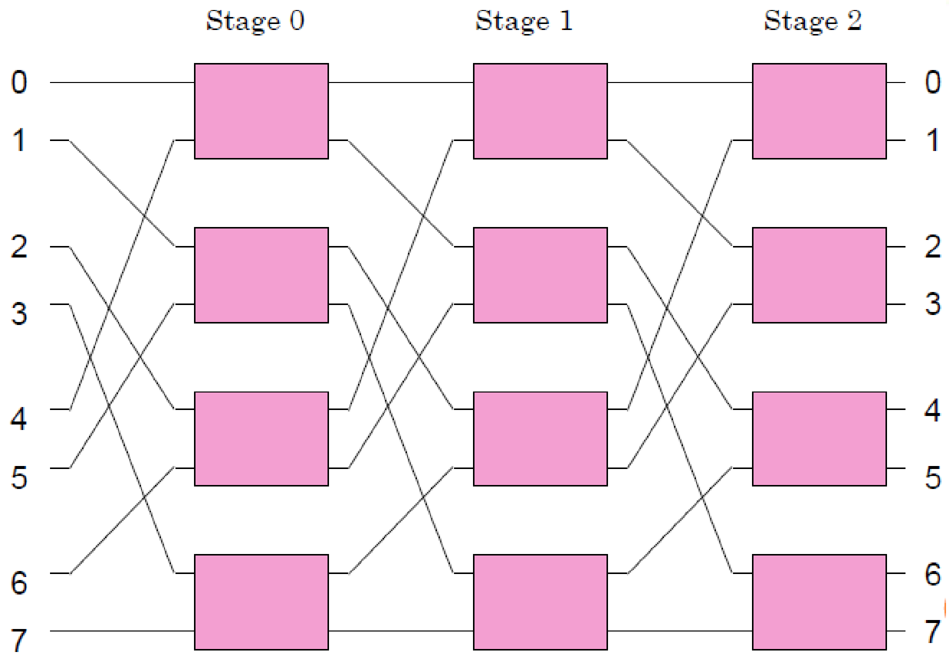


Shuffle Exchange Recirculating Network for N = 8

0 →0, 1→2, 2→4, 3→6, 4→1, 5→3, 6→5, 7→7
shuffle – dashed lines

0 →1, 1→0, 2→3, 3→2, 4→5, 5→4, 6→7, 7→6
Exchange – bold lines

8x8 Omega Network

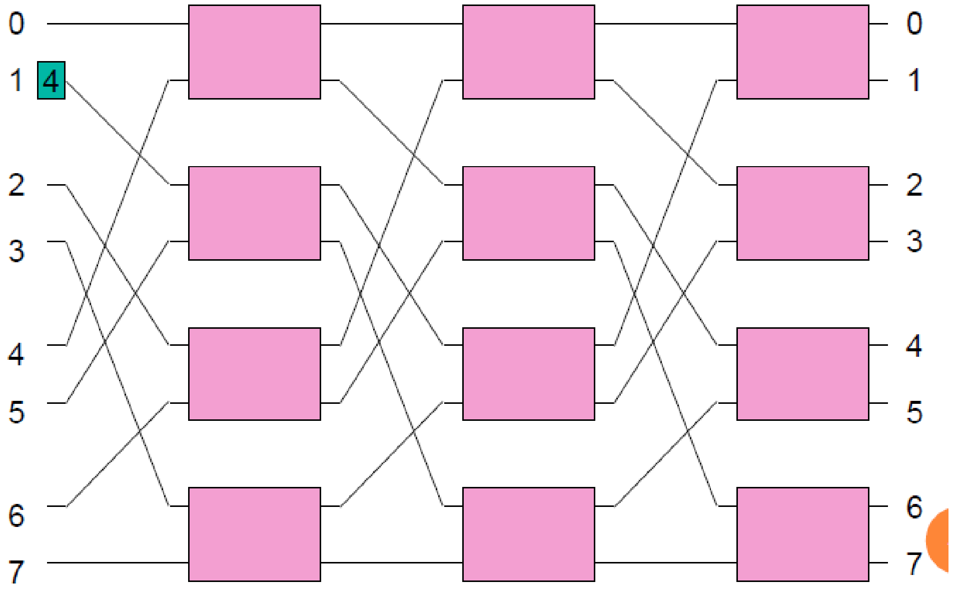


Self Routing in Omega networks

- Complexity: Omega Network Complexity $O(N\log_2 N)$
- Self Routing: Destination Tag Routing
- The source node generates a tag, which is binary equivalent of the destination.
- At each switch, the corresponding tag bit of destination is checked.
- If the bit is 0, the input is connected to the upper output.
- If bit is 1, the input is connected to the lower output.
- If both inputs have either 0 or 1, it is a switch conflict. One of them is connected. The other one is rejected or buffered at the switch (if it has buffer => buffered crossbar)
- Network conflict - Select Round Robin
- Less Bandwidth than crossbar, but more cost effective

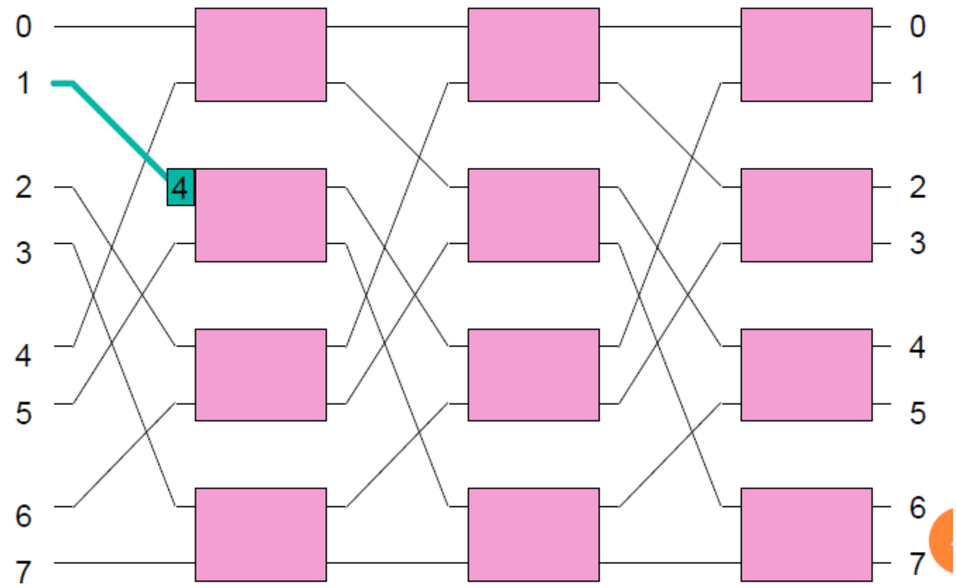
Example of Self Routing Omega Network

Cell destined for output port 4 (= 100_2)



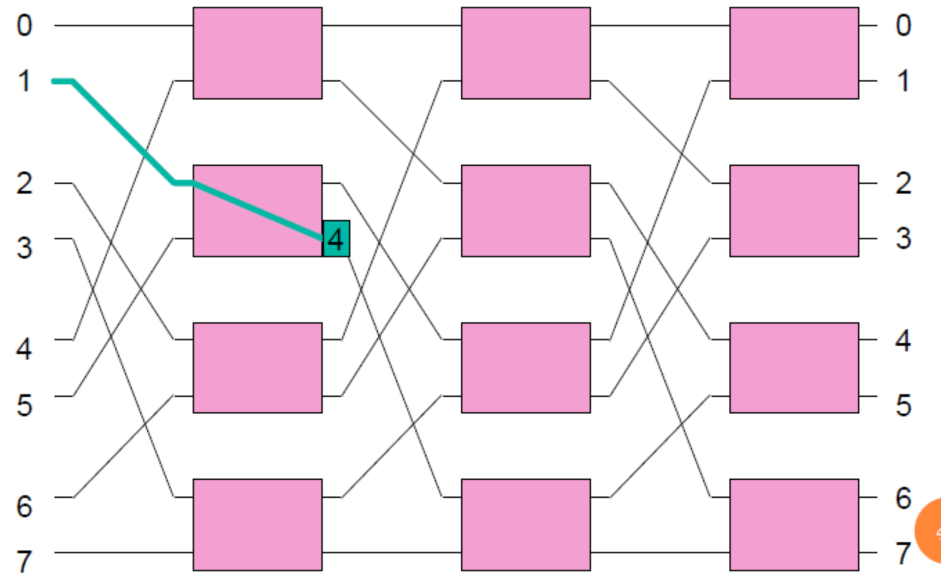
Example of Self Routing Omega Network

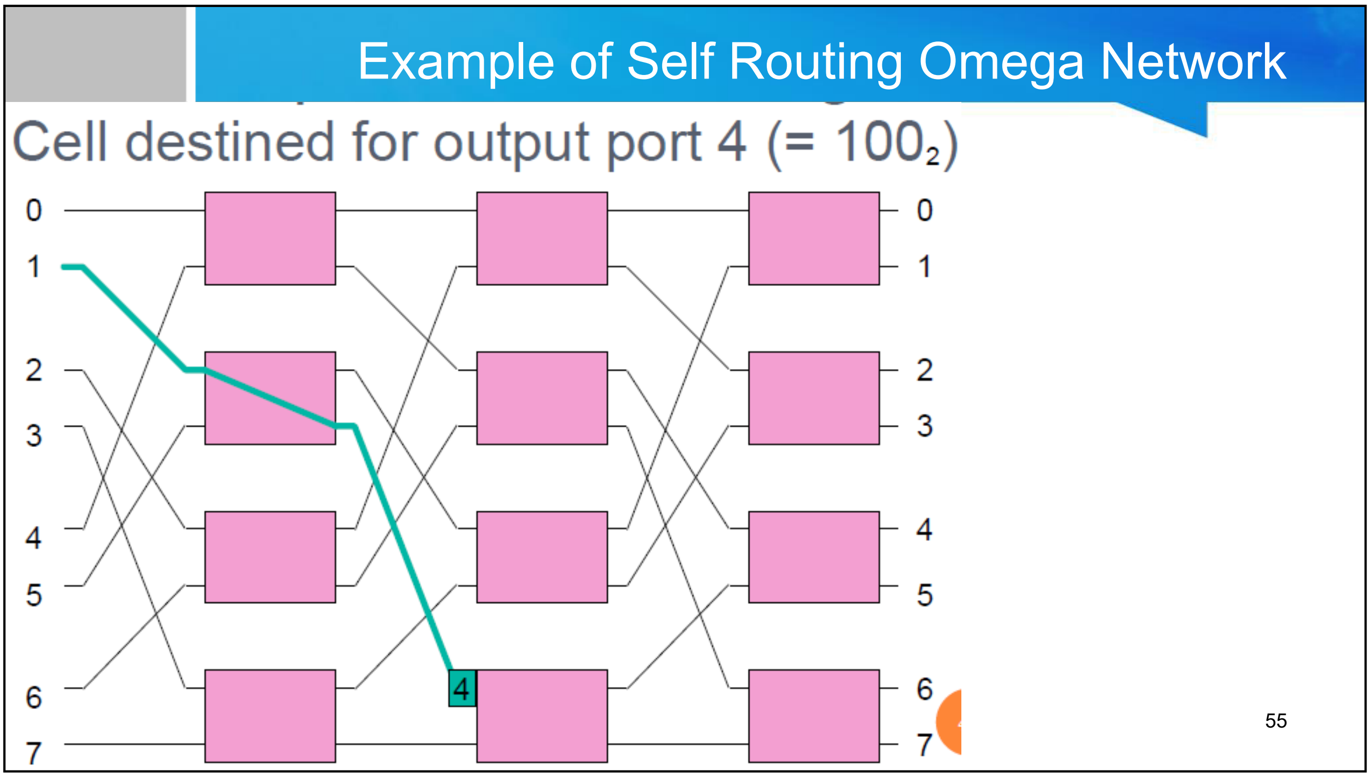
Cell destined for output port 4 (= 100_2)



Example of Self Routing Omega Network

Cell destined for output port 4 (= 100_2)

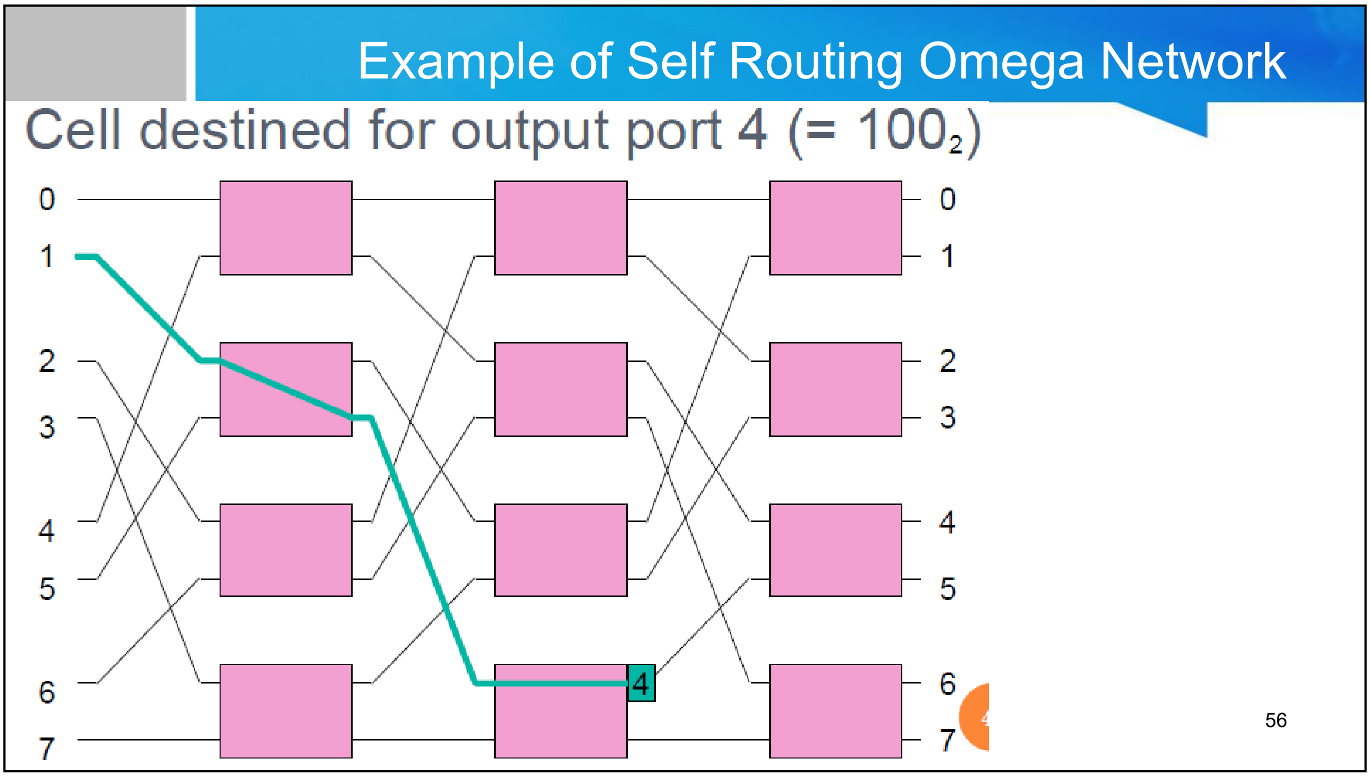




Example of Self Routing Omega Network

Cell destined for output port 4 (= 100_2)

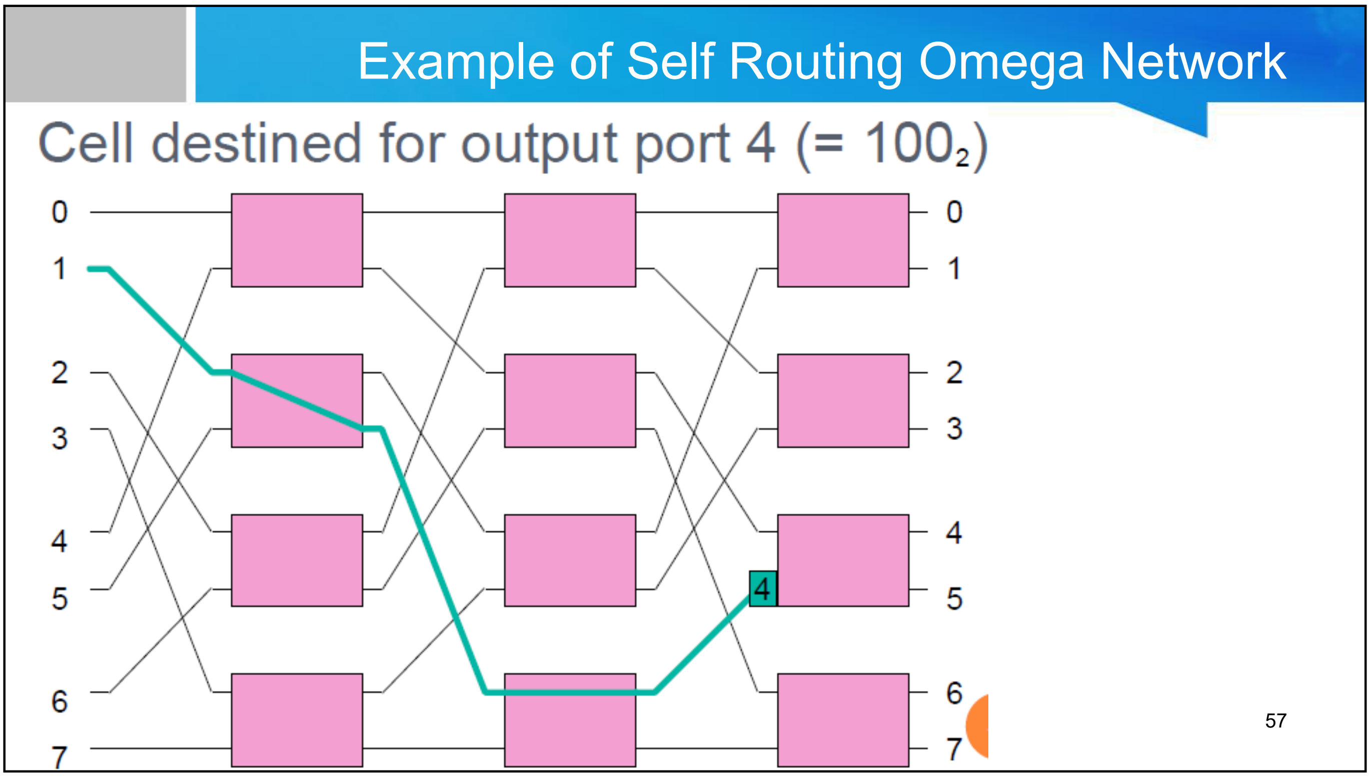
55



Example of Self Routing Omega Network

Cell destined for output port 4 (= 100_2)

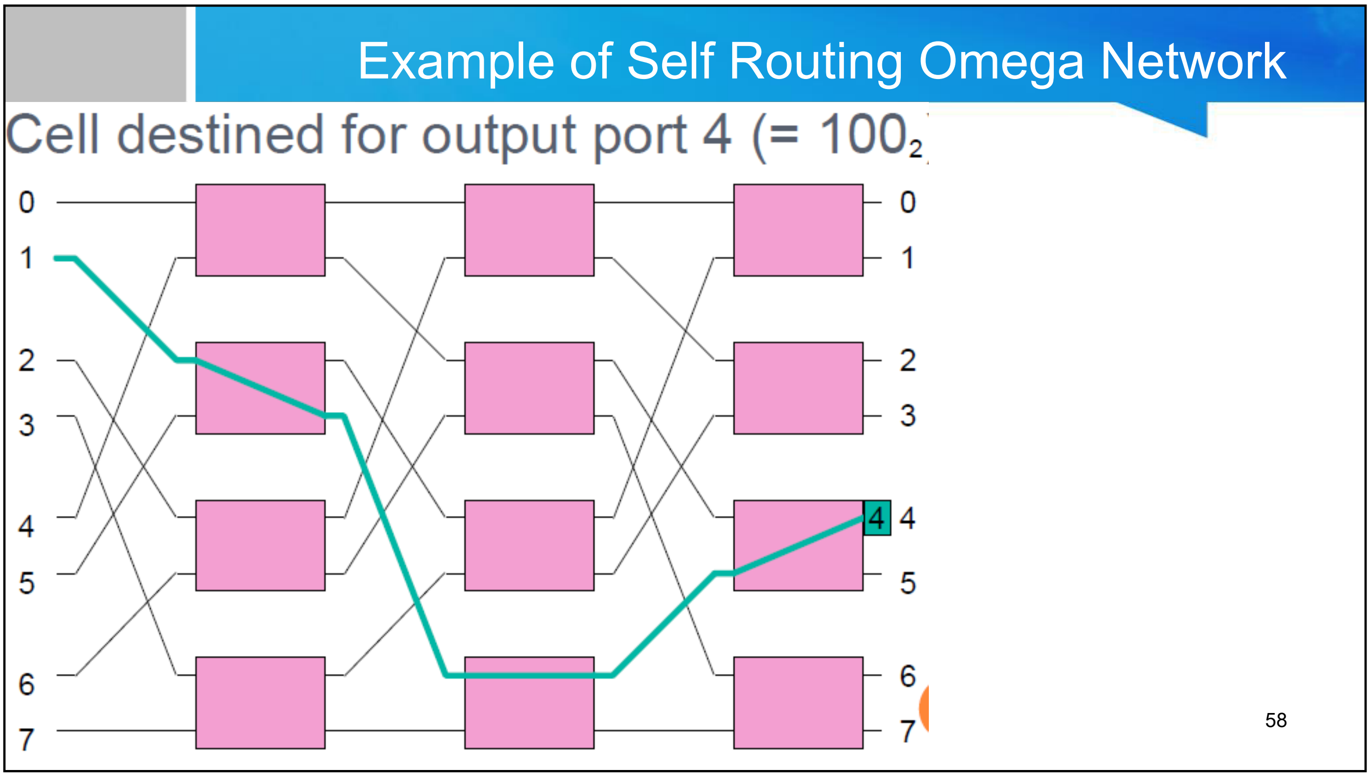
56



Example of Self Routing Omega Network

Cell destined for output port 4 (= 100_2)

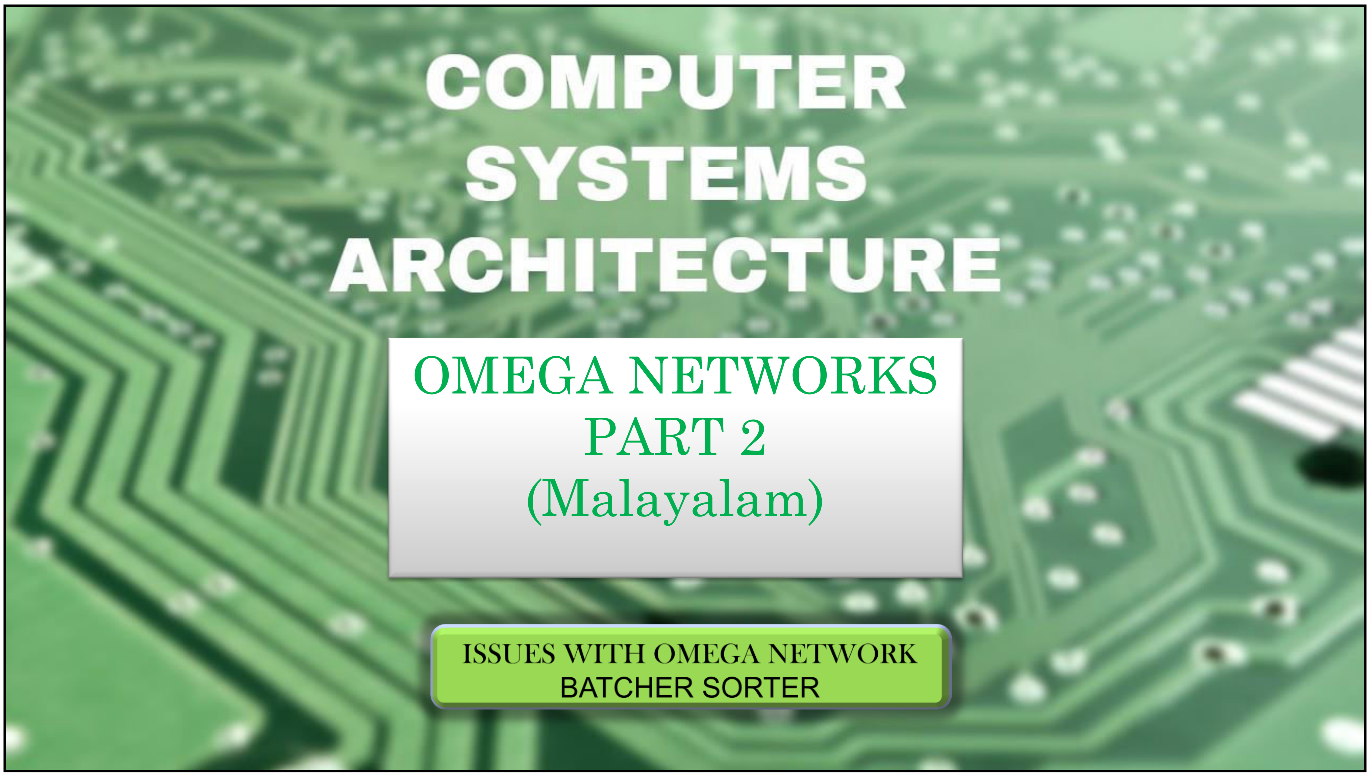
57



Example of Self Routing Omega Network

Cell destined for output port 4 (= 100_2

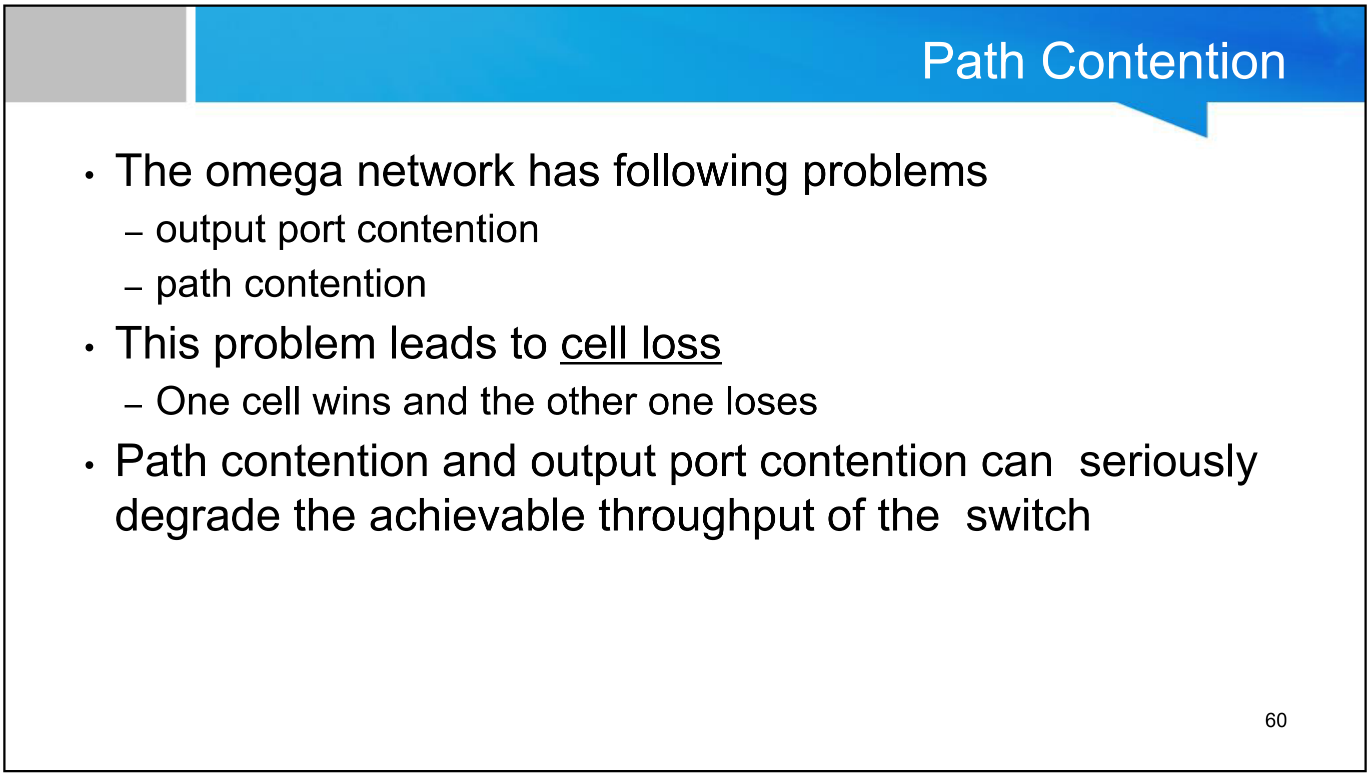
58



COMPUTER SYSTEMS ARCHITECTURE

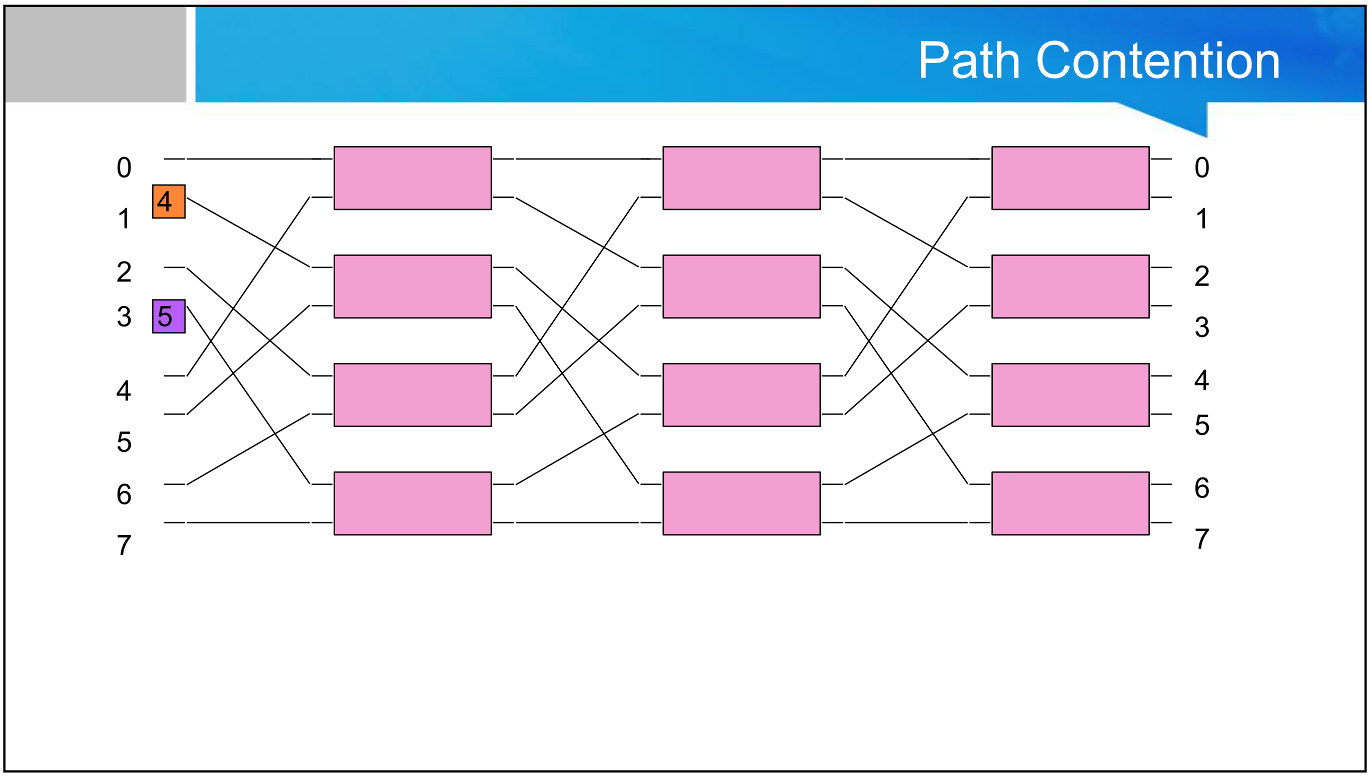
OMEGA NETWORKS PART 2 (Malayalam)

ISSUES WITH OMEGA NETWORK
BATCHER SORTER

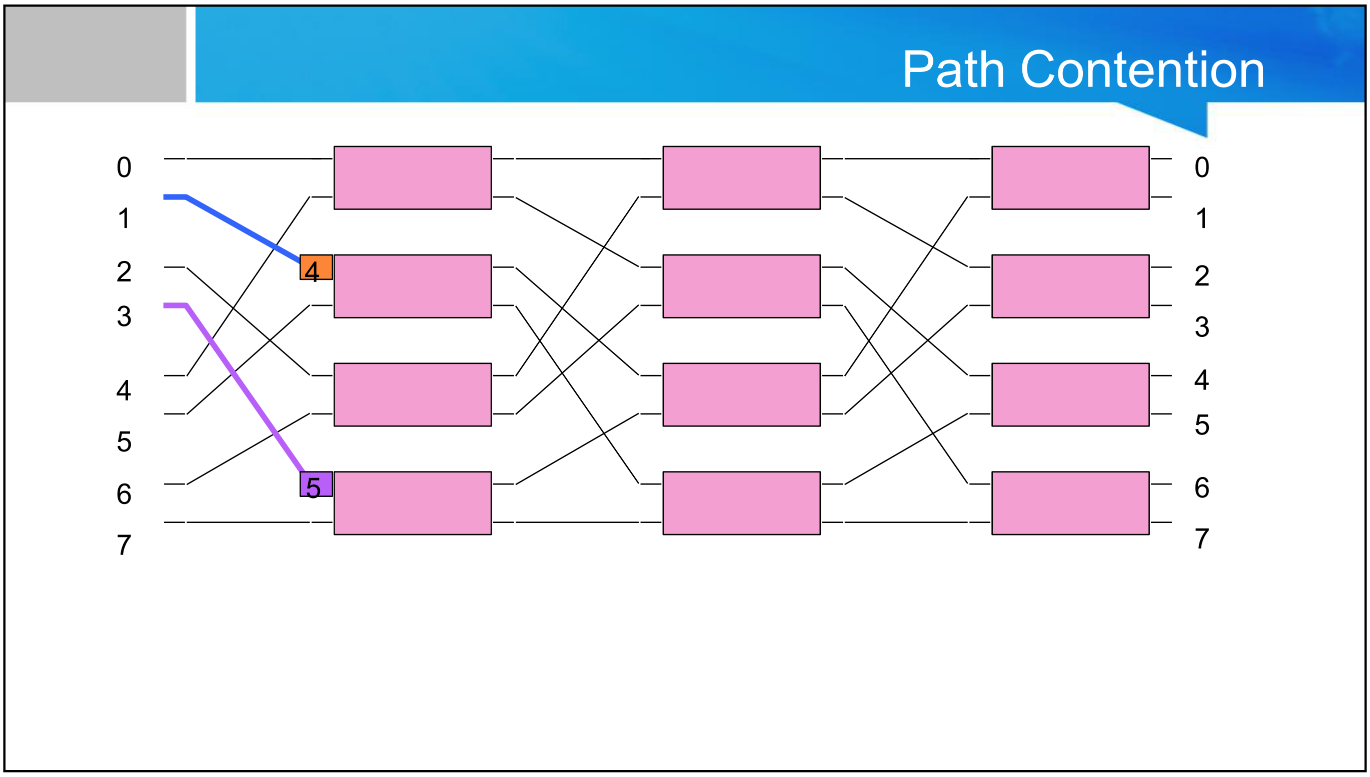


Path Contention

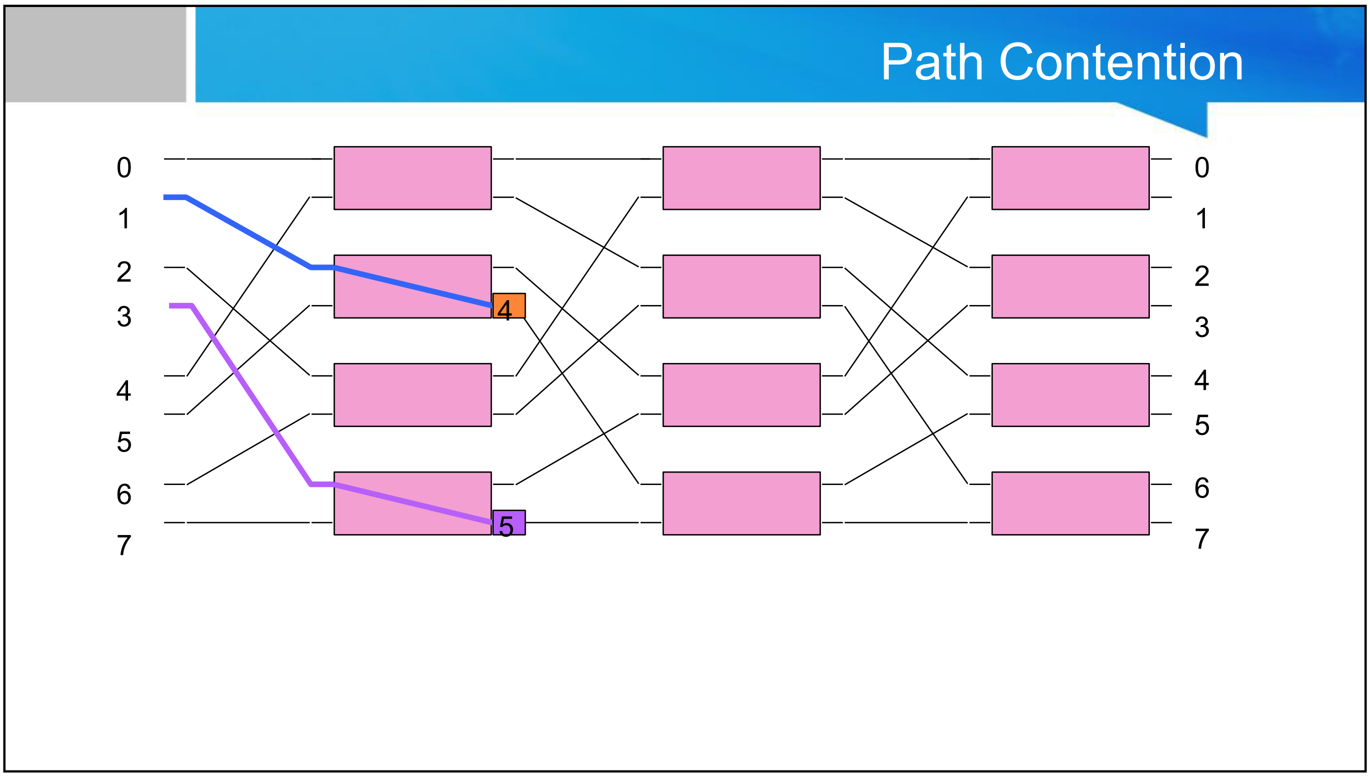
- The omega network has following problems
  - output port contention
  - path contention
- This problem leads to <u>cell loss</u>
  - One cell wins and the other one loses
- Path contention and output port contention can seriously degrade the achievable throughput of the switch



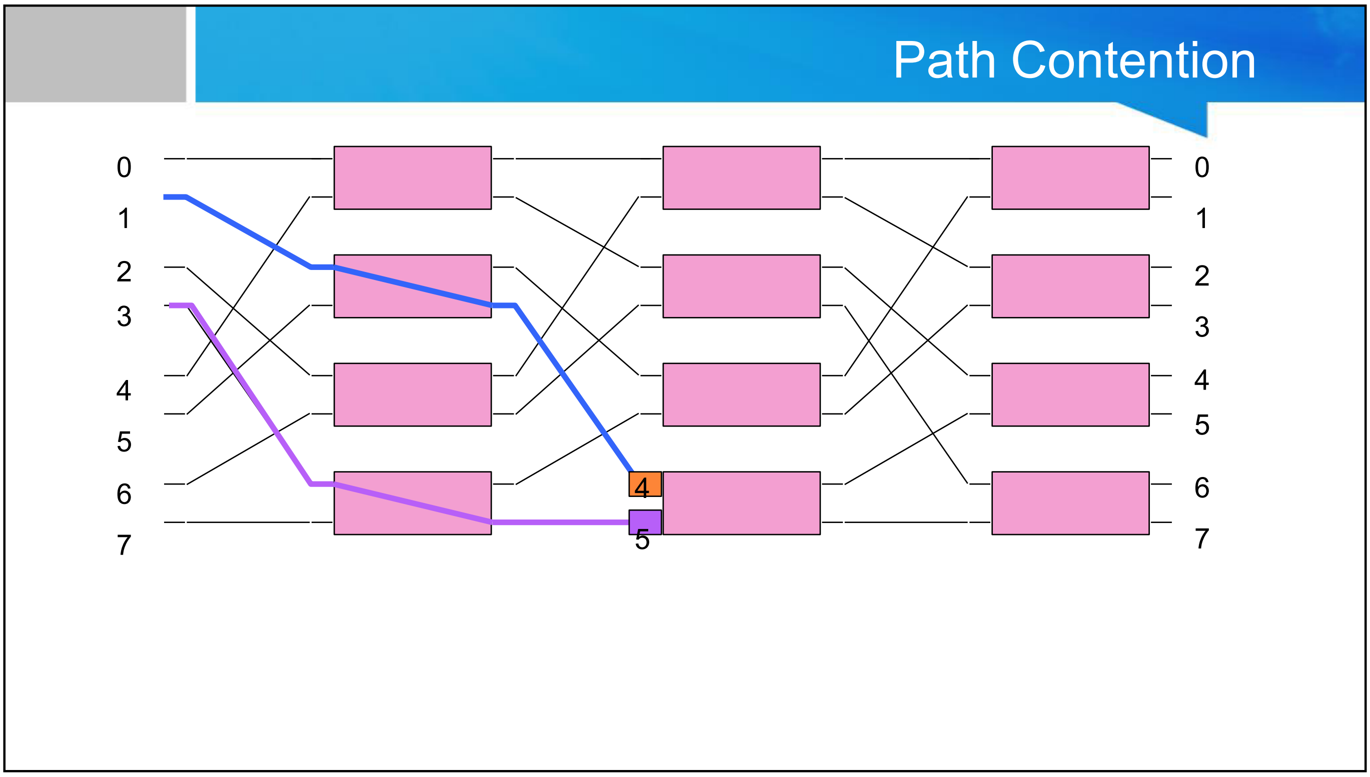
Path Contention



Path Contention

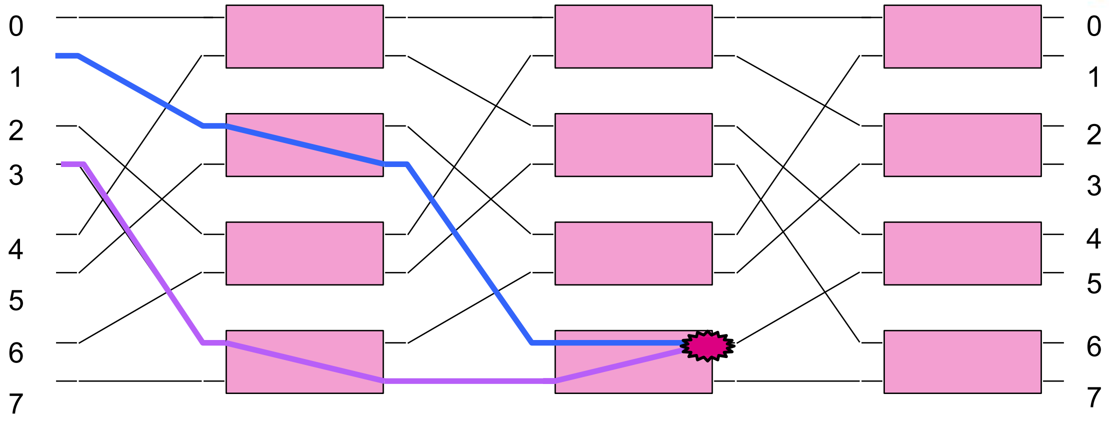


Path Contention

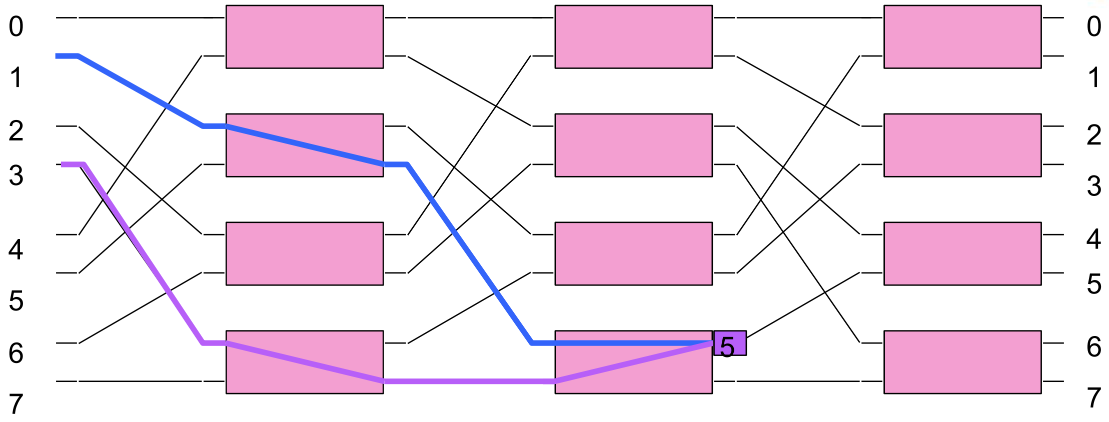


Path Contention

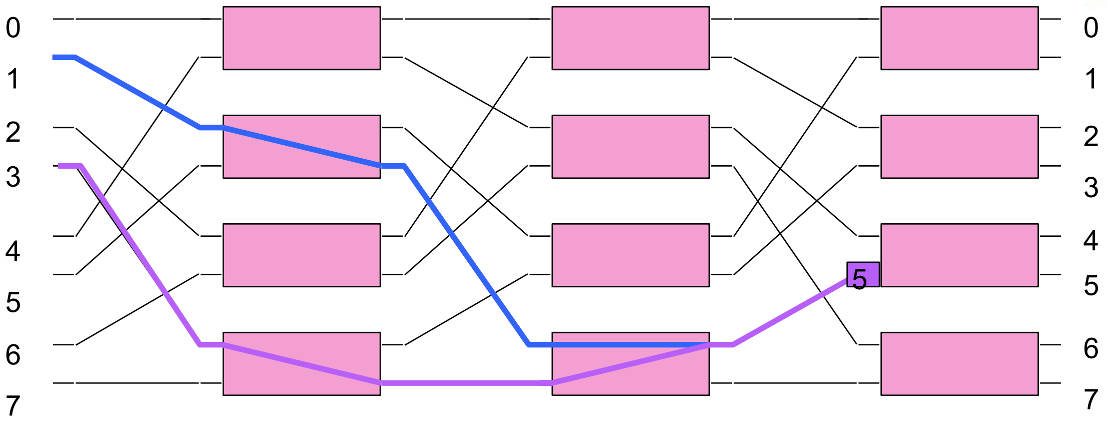
Path Contention



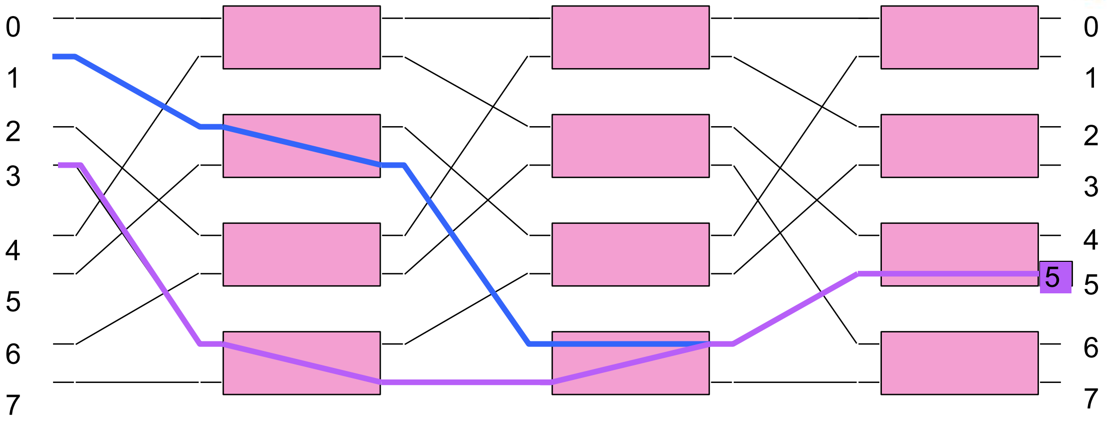
Path Contention



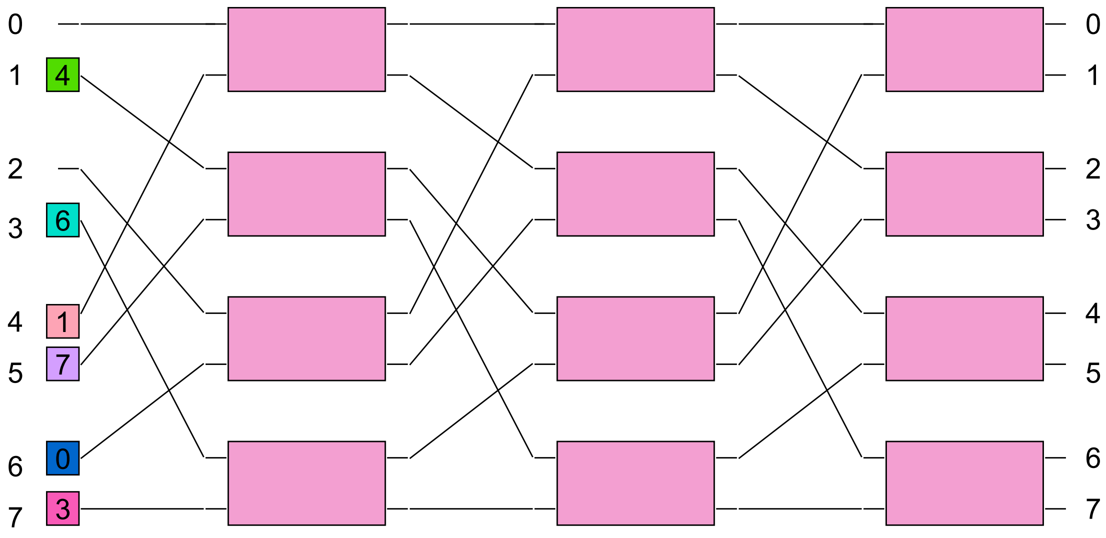
Path Contention



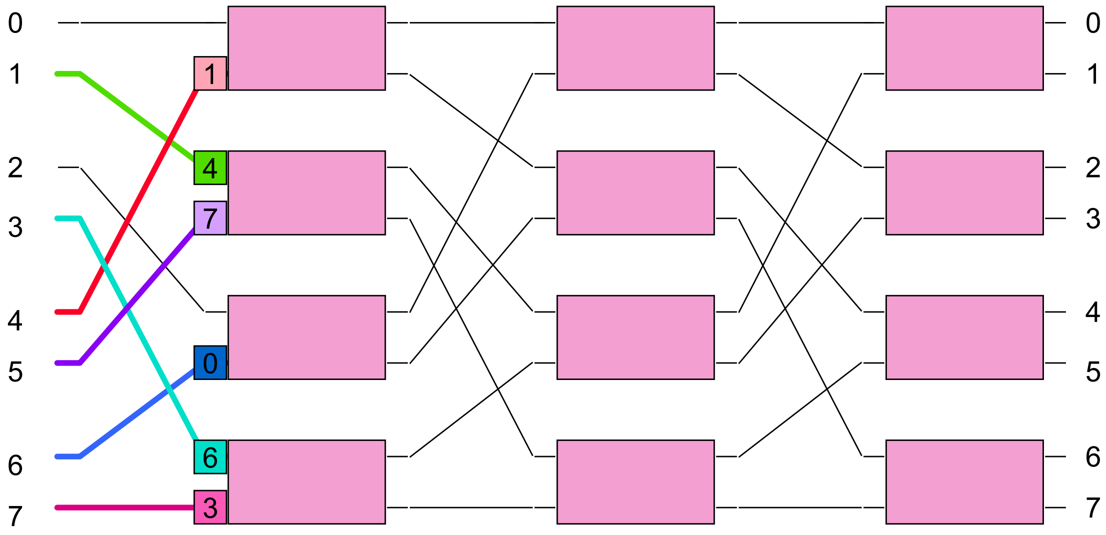
Path Contention



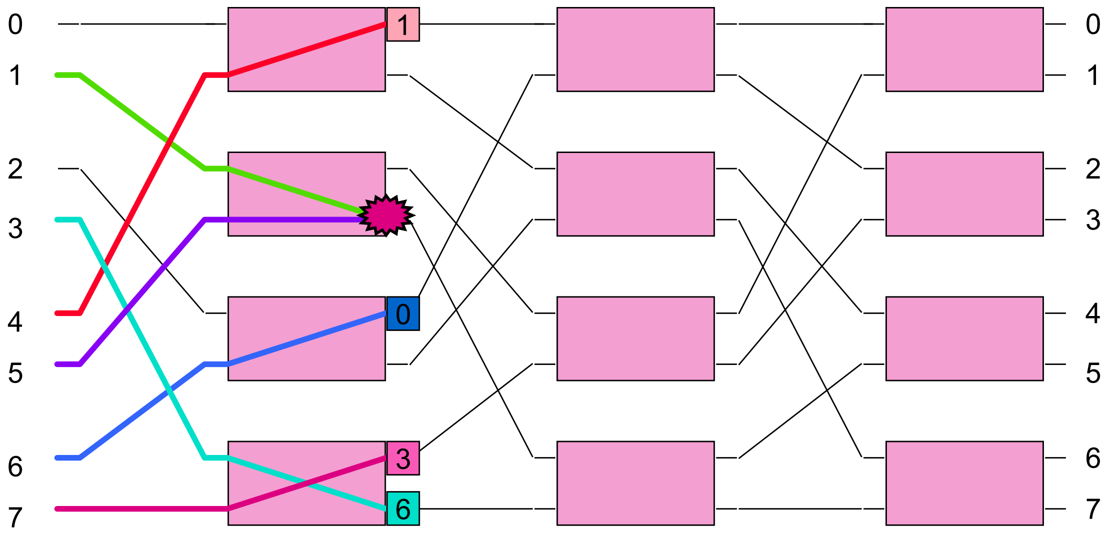
Performance Degradation



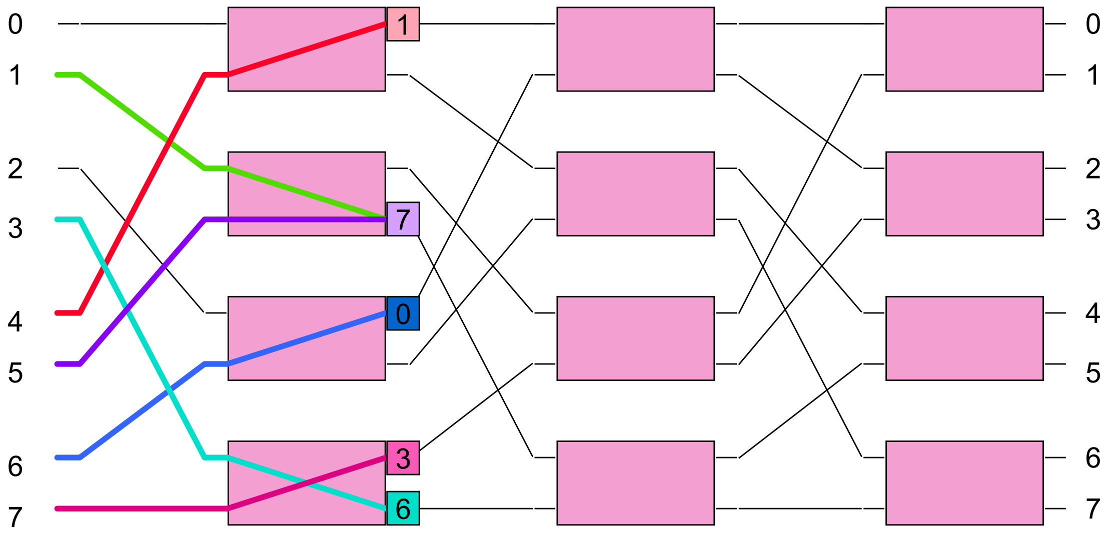
Performance Degradation

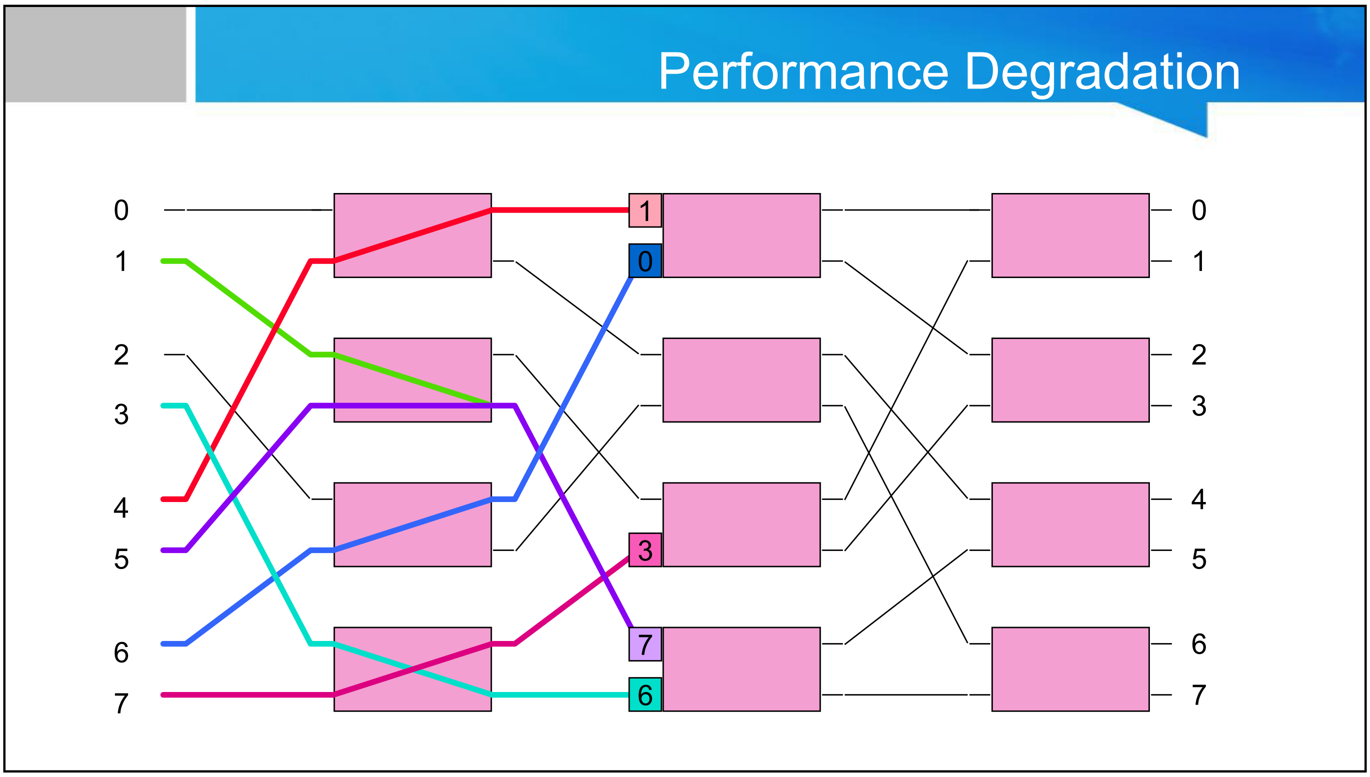


Performance Degradation

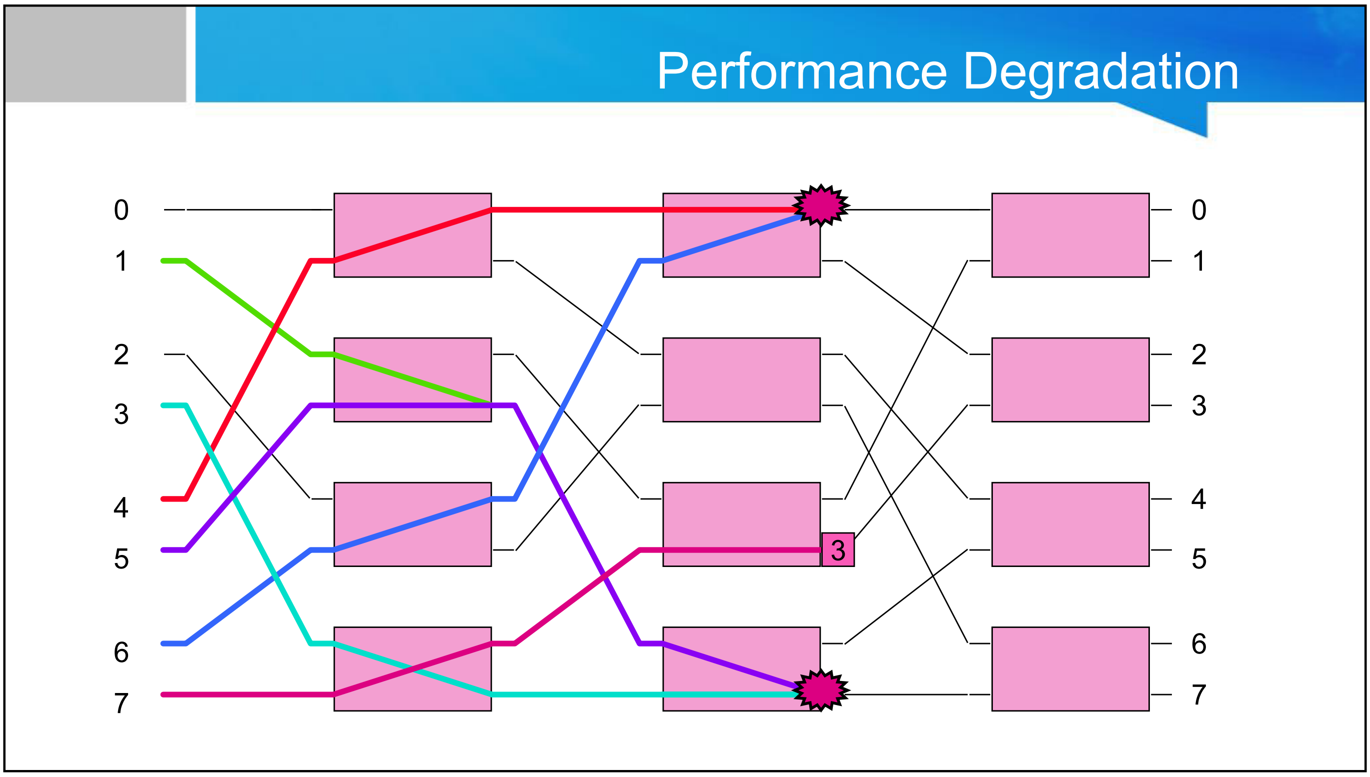


Performance Degradation



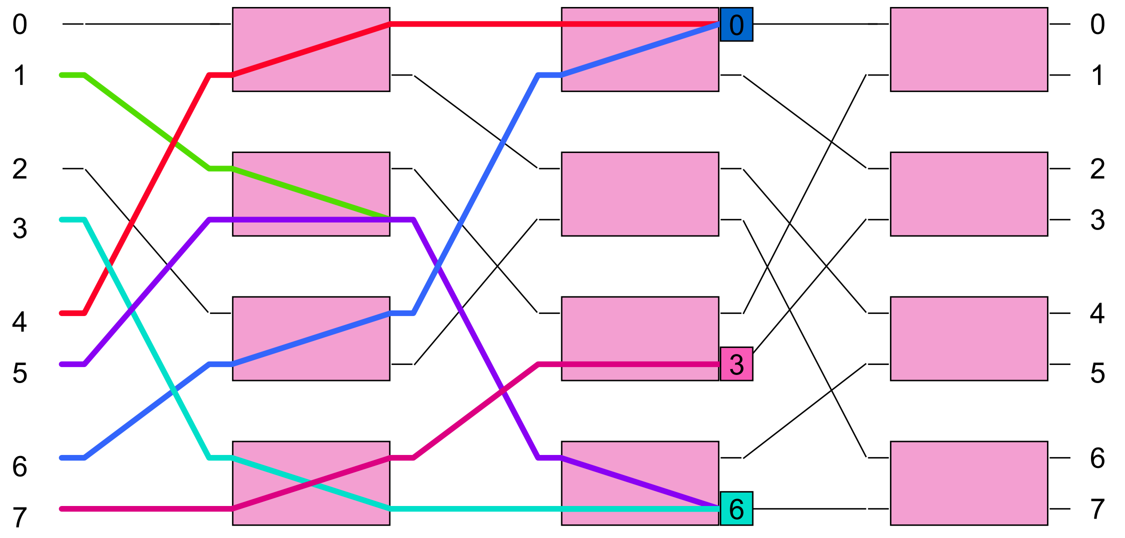


Performance Degradation

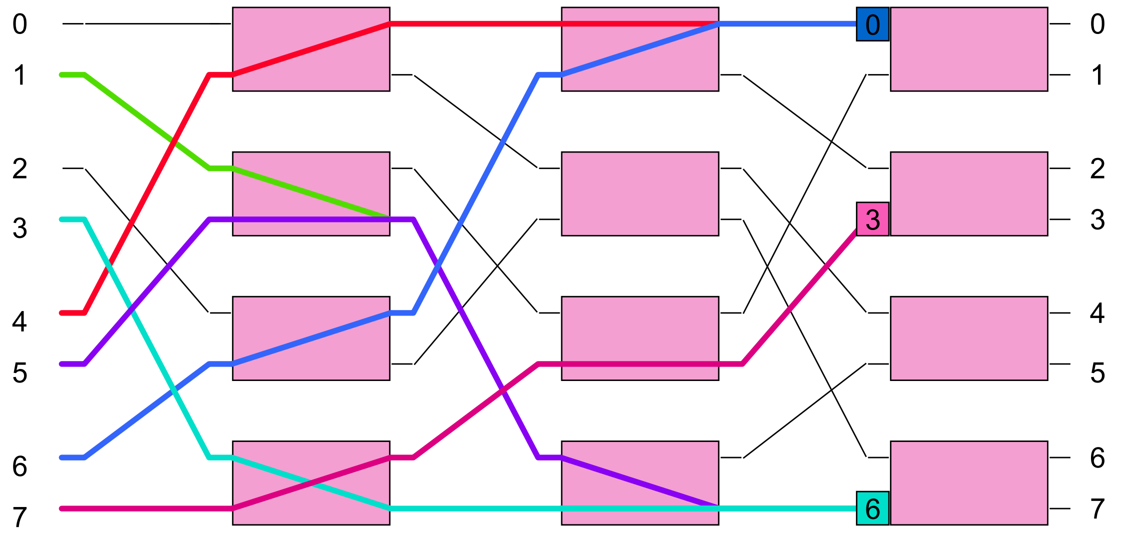


Performance Degradation

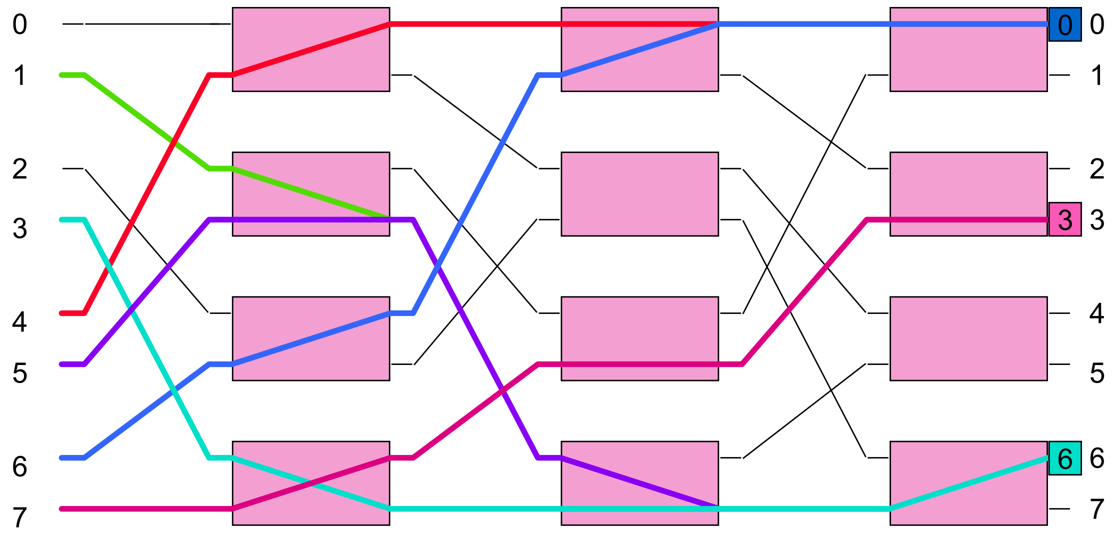
Performance Degradation



Performance Degradation

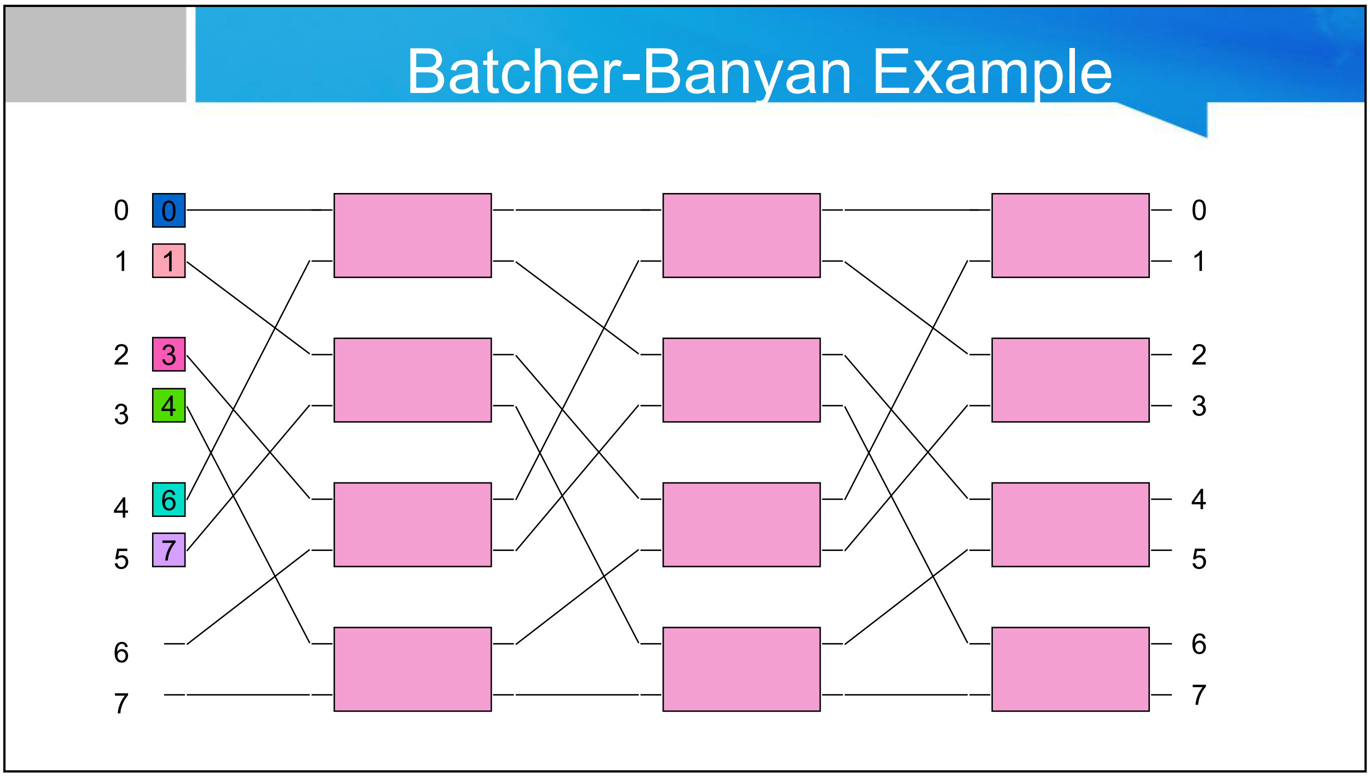


Performance Degradation

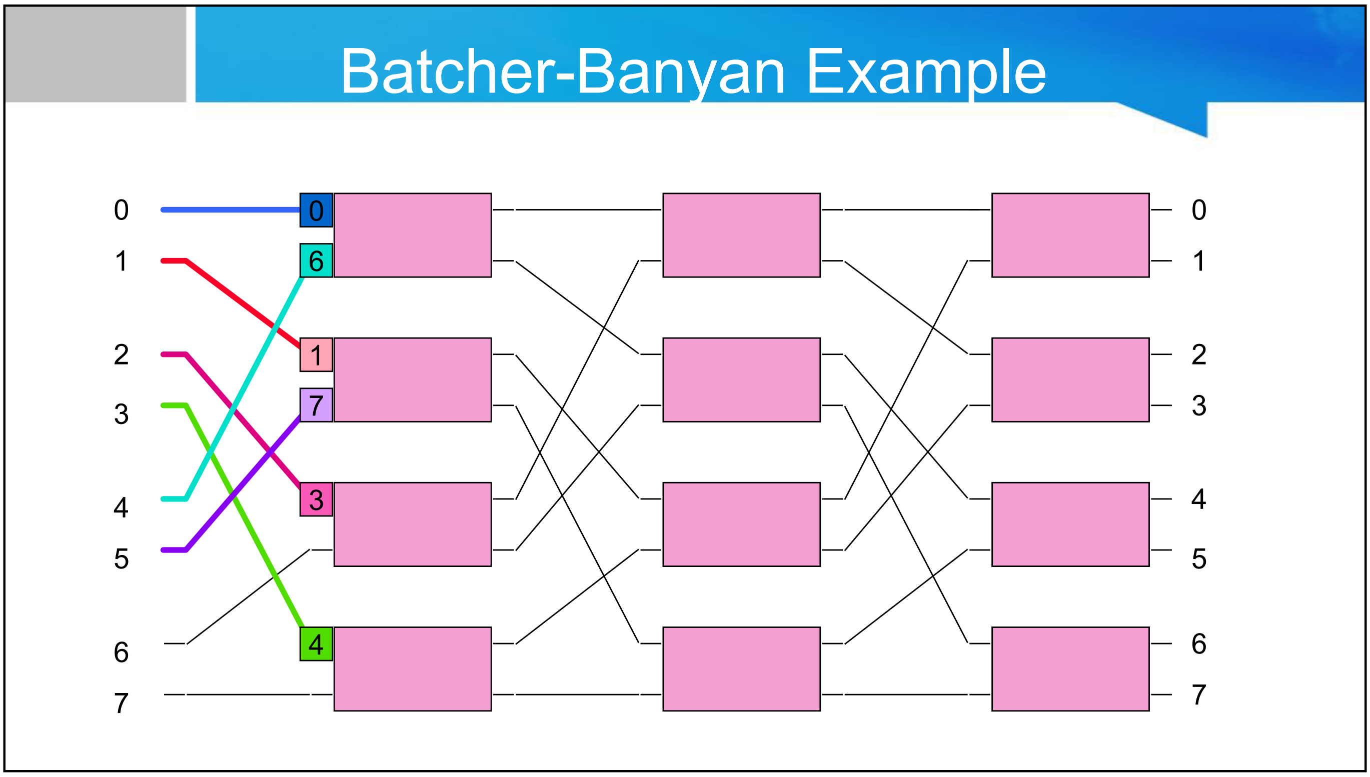


A Solution: Batcher Sorter

- One solution to the contention problem is to sort the cells into monotonically increasing order based on desired destination port
- Done using a bitonic sorter called a Batcher
- Places the M cells into gap-free increasing sequence on the first M input ports
- Eliminates duplicate destinations

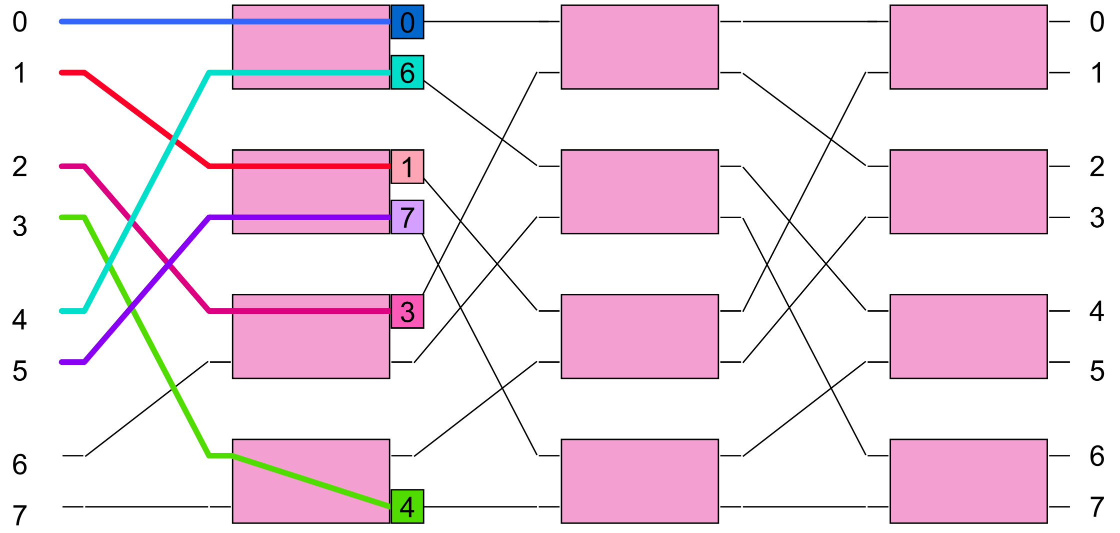


Batcher-Banyan Example

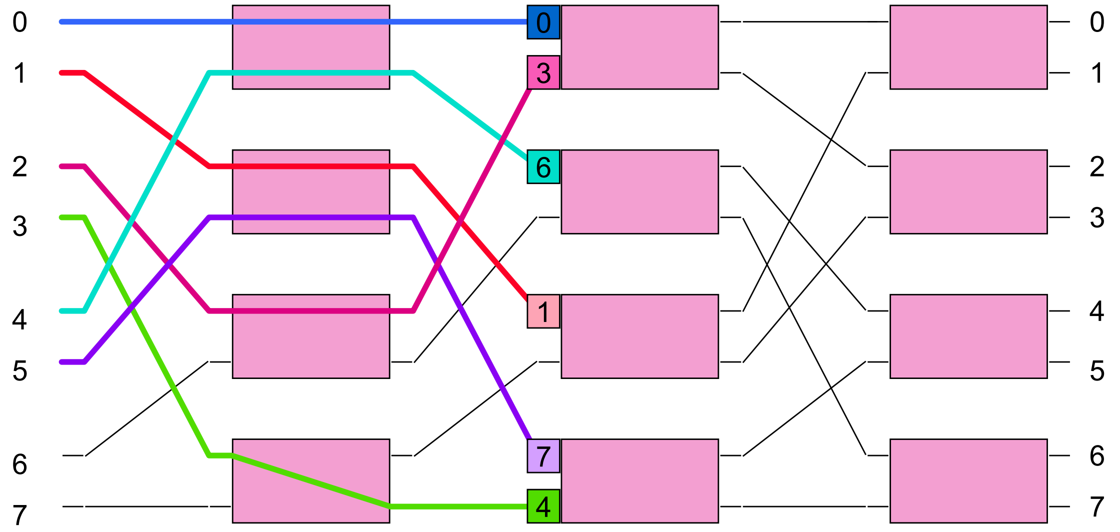


Batcher-Banyan Example

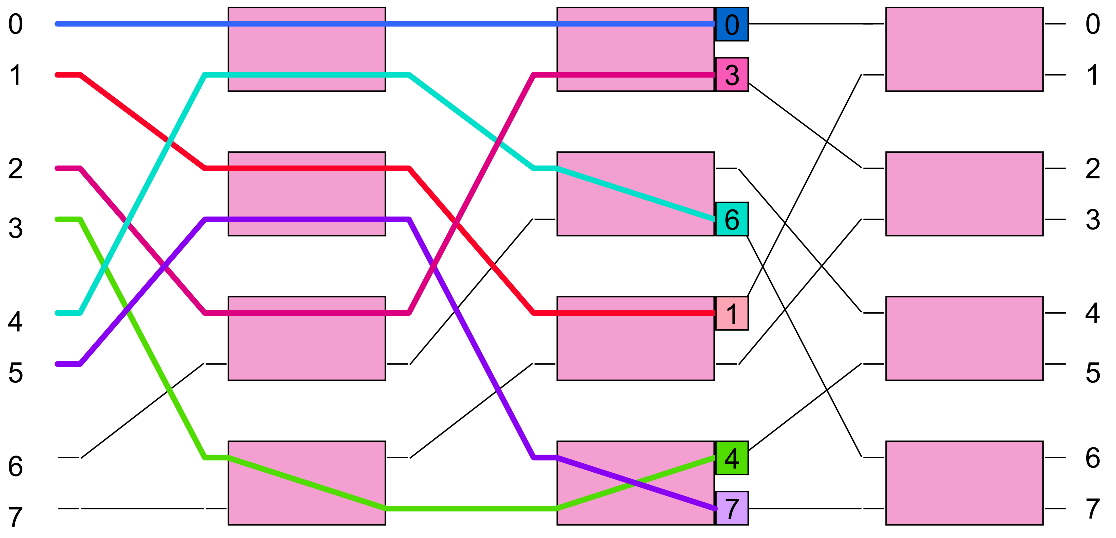
Batcher-Banyan Example



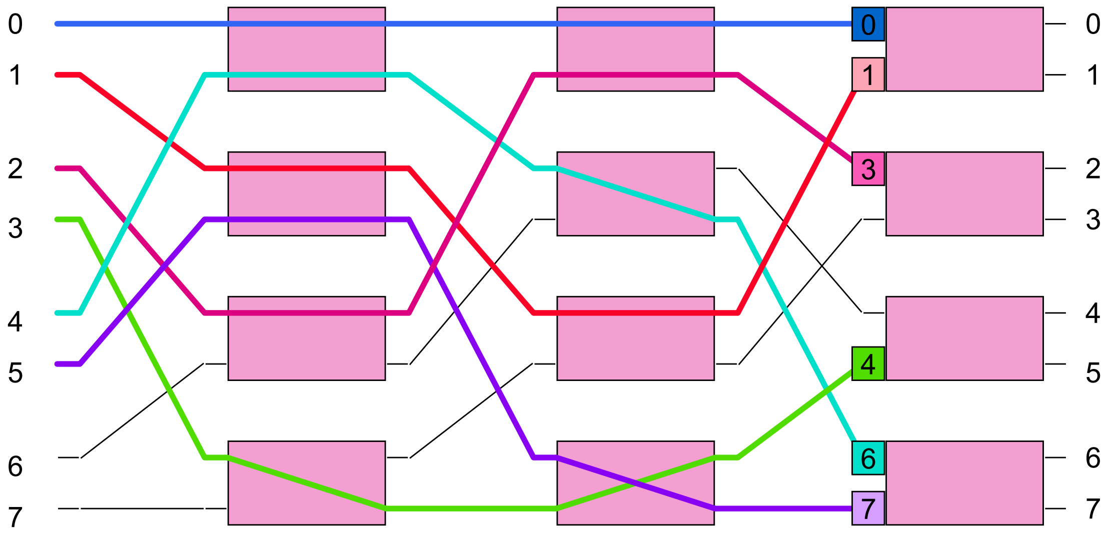
Batcher-Banyan Example



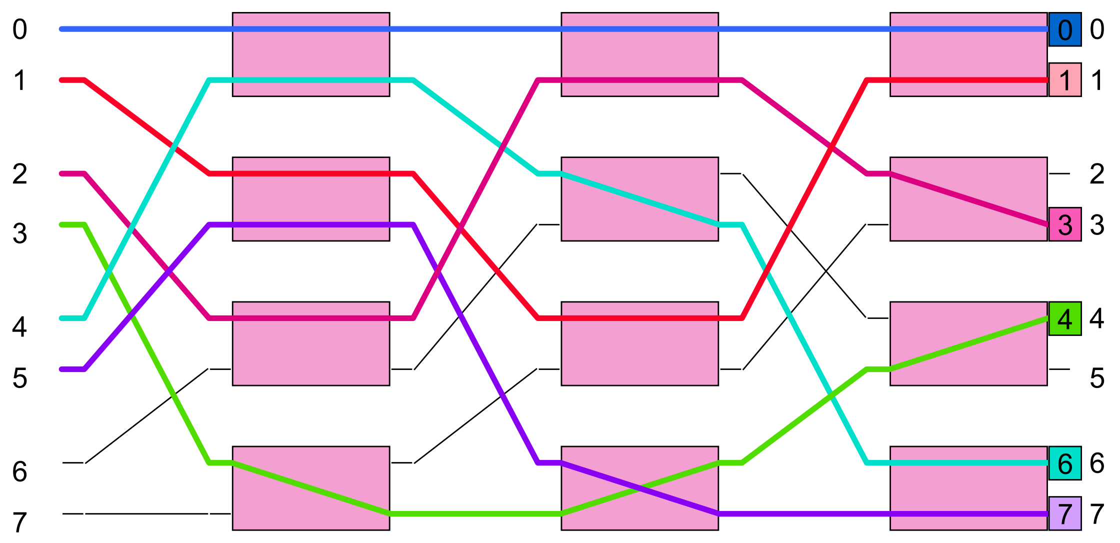
Batcher-Banyan Example



Batcher-Banyan Example

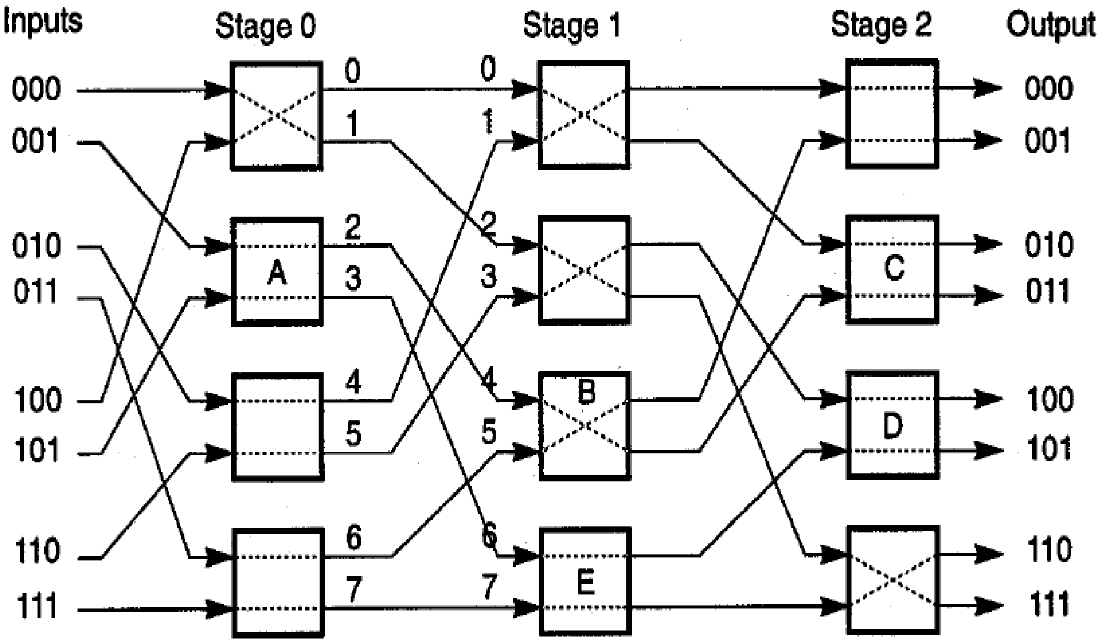


Batcher-Banyan Example



1) Design an 8 input 2 X 2 switches as building blocks. Show the switch settings for the permutation π=(0,7,6,4,2) (1,3)(5). Show the conflicts in switch settings if any. Explain blocking and non-blocking networks

(6 marks)

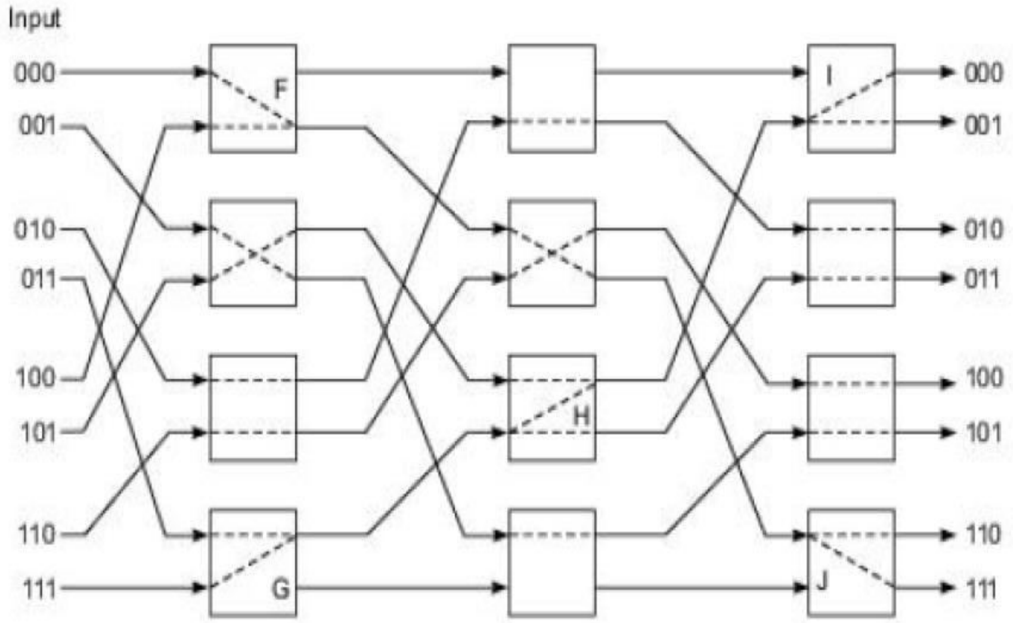


(a) Permutation $\pi_1 = (0,7,6,4,2)(1,3)(5)$ implemented on an Omega network without blocking

- 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***

2) Design an 8 input 2 X 2 switches as building blocks. Show the switch settings for the permutation π2=(0,6,4,7,3) (1,5)(2). Show the conflicts in switch settings if any. Explain blocking and non-blocking networks

(6 marks)



(b) Permutation π_2 = (0, 6, 4, 7, 3) (1, 5) (2) blocked at switches marked F, G, and H

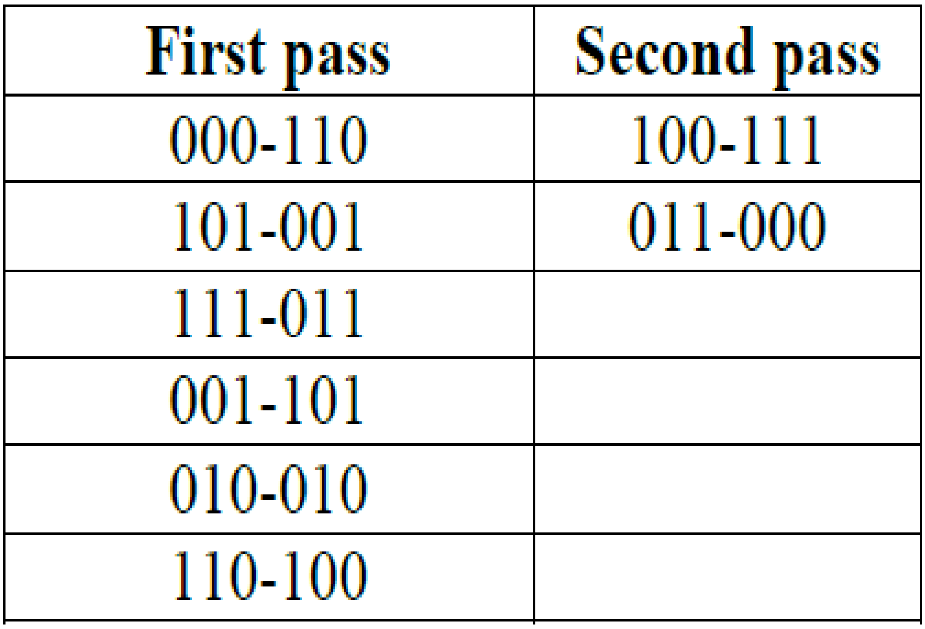
- Now consider implementing the permutation π2 in the 8-input Omega network.
- Conflicts in switch settings do exist in three switches identified as F, G, and H
- The conflicts occurring

✓at F are caused by the desired routings 000 —> 110 and 100 —> lll.

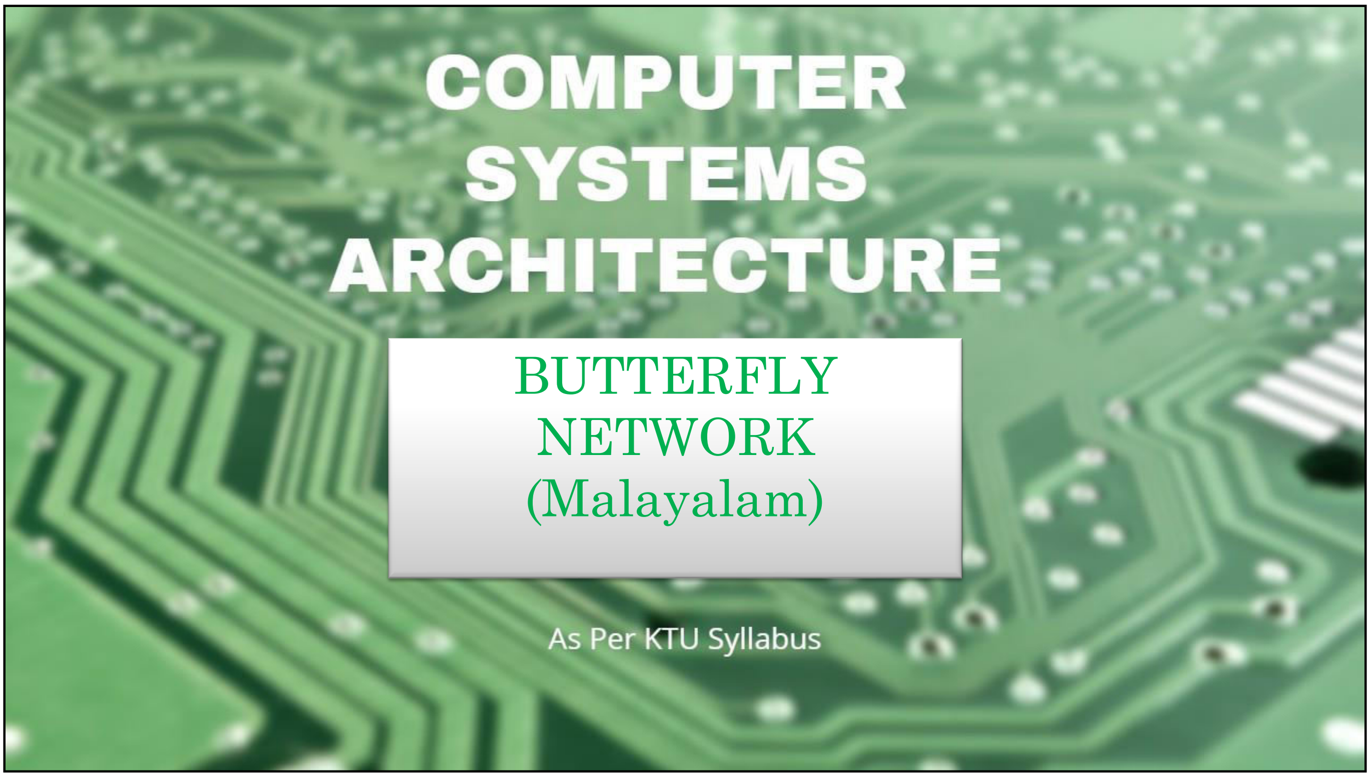
✓at switch G between 0ll —> 000 and lll —> 011.

✓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 π2, 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	



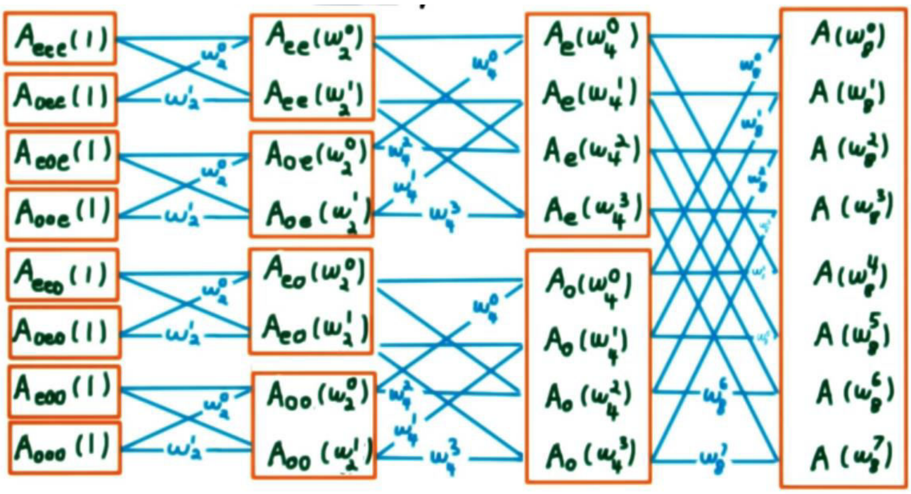
COMPUTER SYSTEMS ARCHITECTURE

BUTTERFLY NETWORK (Malayalam)

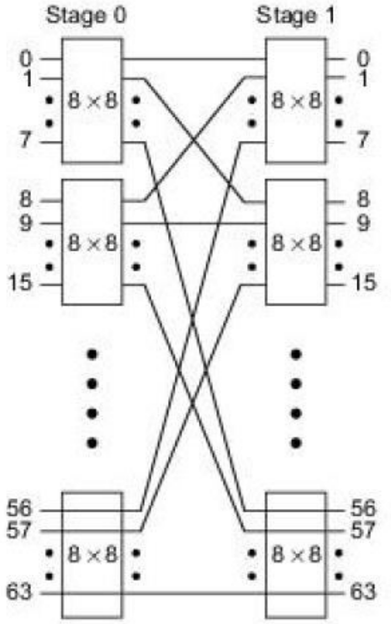
As Per KTU Syllabus

Routing in Butterfly Networks

- 64-input Butterfly network built with two stages ($2 = \log_8 64$) of 8 x 8 crossbar switches. The eight-way shuffle function is used to establish the inter stage connections between stage 0 and stage 1.



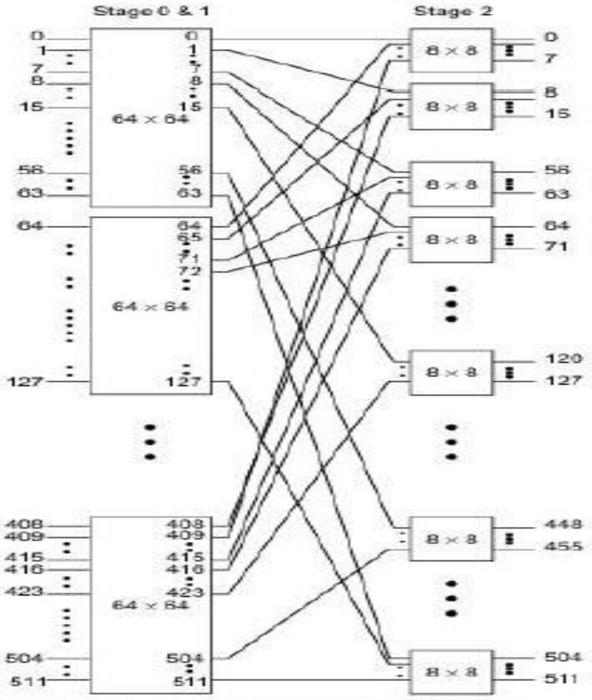
Two-Stage



(a) A two-stage 64 × 64 Butterfly switch network built with 16 8 × 8 crossbar switches and eight-way shuffle interstage connections

- 64-input Butterfly network built with two stages ($2 = \log_8 64$) of 8X8 crossbar switches.
- The eight-way shuffle function is used to establish the interstage connections between stage 0 and stage 1.
- sixteen 8X8 crossbar switches

Three Stage



(b) A three-stage 512 × 512 Butterfly switch network built with 192 8 × 8 crossbar switches

- a three-stage Butterfly network is constructed for 512 inputs, again with 8X8 crossbar switches.
- Each of the 64X64 boxes in this Figure is identical to the two-stage Butterfly network
- 16X8+ 8X8 = 192 crossbar switches are used

- Larger Butterfly networks can be modularly constructed using more stages.
- No broadcast connections are allowed in a Butterfly network making these networks a restricted subclass of Omega networks.

The Hot-Spot Problem

- When the 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.
- ---eg) a semaphore variable
- Hot spots may degrade the network performance significantly.
- 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 as:

```
Fetch&Add $(x, e)$
    $\{temp \leftarrow x;$
        $x \leftarrow temp - e;$
    return $temp\}$
```

- 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 + . . . + 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 + . . . + eN.

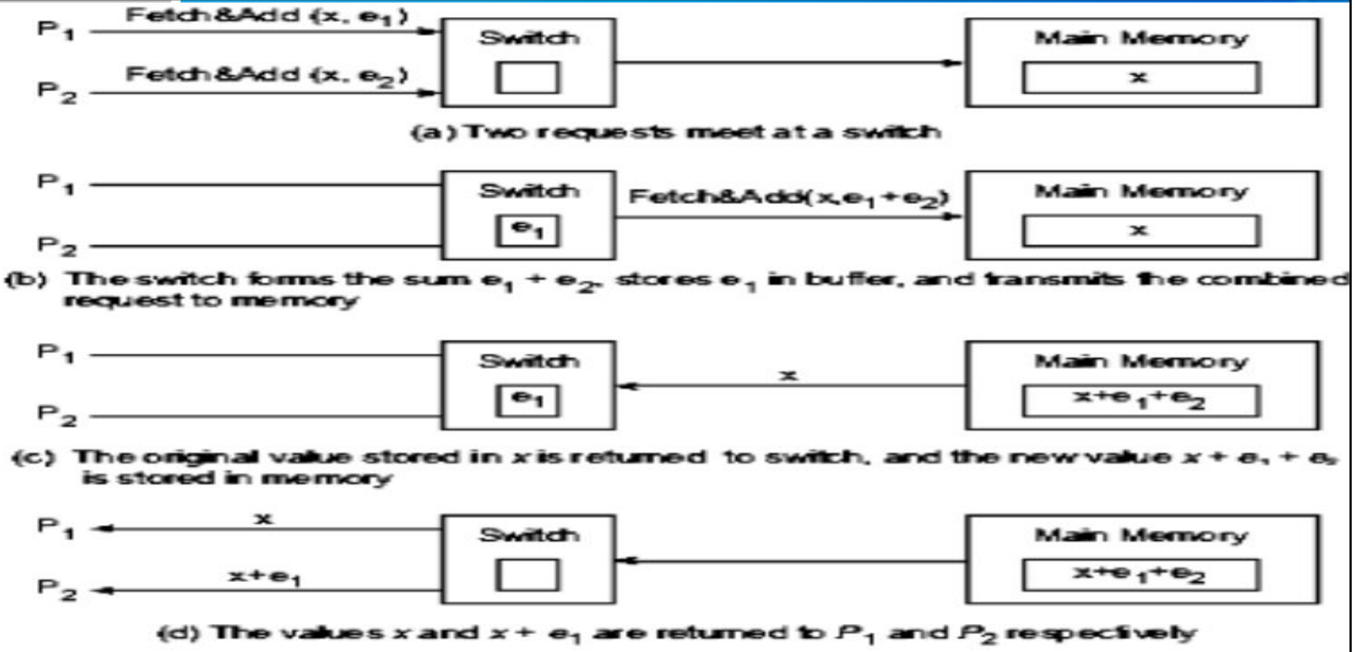
- The values returned to the N requests are all unique, depending on the serialization order followed.
- The net result is similar to a sequential execution of N Fetch&Adds but is performed in one indivisible operation.
- One of the following operations will be performed if processor
  - P1 executes 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 ahead of that from P2, the following values are returned:

$$\begin{aligned} \mathrm{Ans}_1 &\leftarrow x \\ \mathrm{Ans}_2 &\leftarrow x + e_1 \end{aligned}$$

- If the execution order is reversed, the following values are returned:

$$\begin{aligned} \mathrm{Ans}_1 &\leftarrow x + e_2 \\ \mathrm{Ans}_2 &\leftarrow x \end{aligned}$$

- 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 (or 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),



(a) Two requests meet at a switch

(b) The switch forms the sum $e_1 + e_2$, stores e_1 in buffer, and transmits the combined request to memory

(c) The original value stored in x is returned to switch, and the new value $x + e_1 + e_2$ is stored in memory

(d) The values x and $x + e_1$ are returned to P_1 and P_2 respectively

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

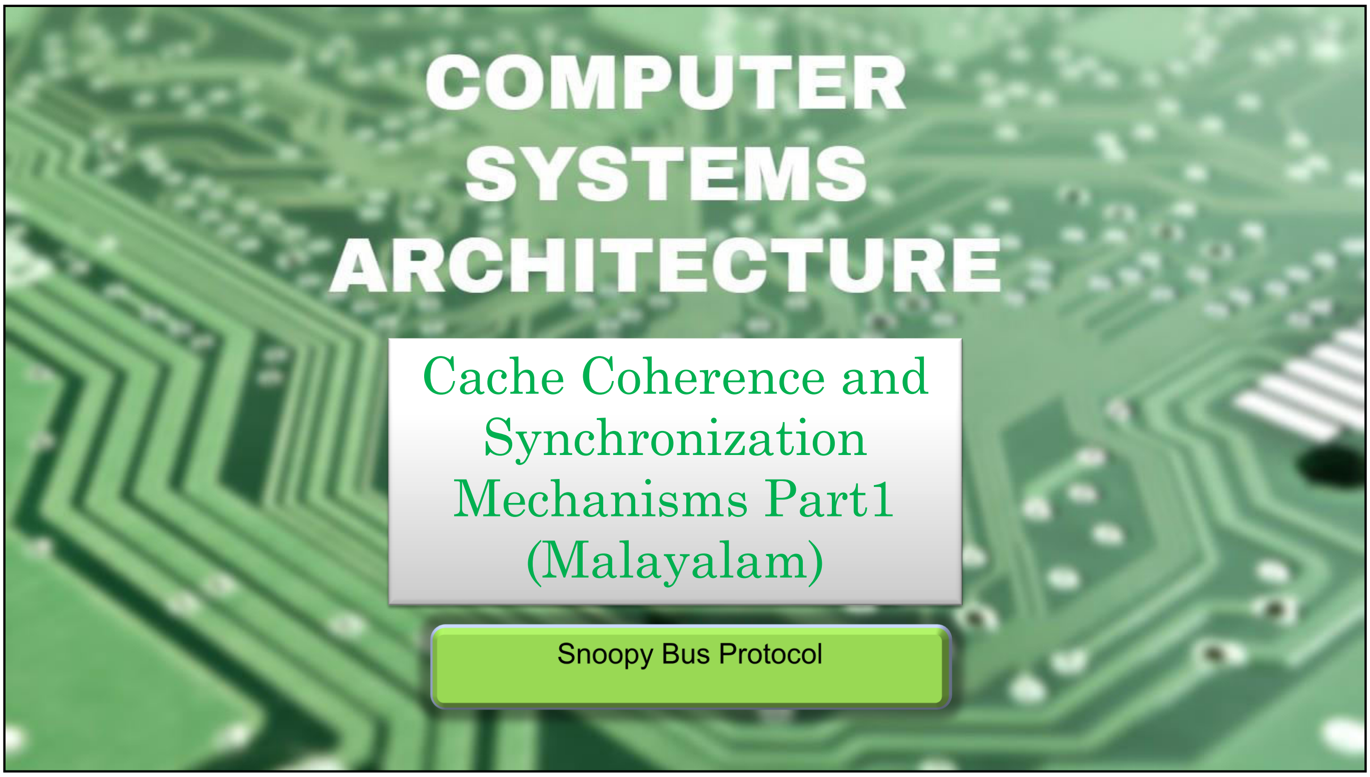
Applications and Drawbacks

- 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.
- The advantage of using a combining network to implement the Fetch&Add operation is achieved at a significant increase in network cost.
- Additional switch cycles are also needed to make the entire operation an atomic memory operation. This may increase the network latency significantly.
- Multistage combining networks have the potential of supporting large-scale multiprocessors with thousands of processors.
- The problem of increased cost and latency may be alleviated with the use of faster and cheaper switching technology in the future.

Multistage Networks in Real Systems

❖Examples:

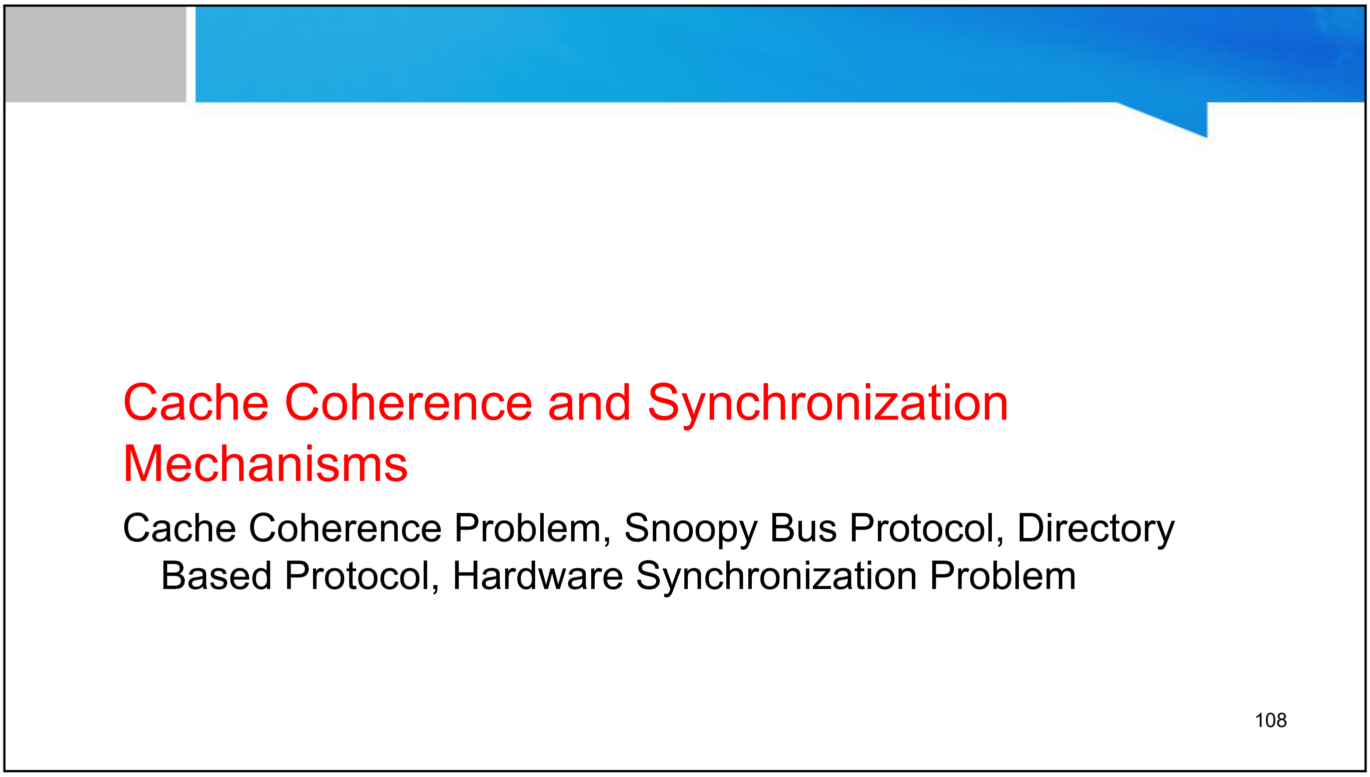
- IBM RP3 using a high-speed Omega network
- Cedar multiprocessor, Ultracomputer using Multistage Omega networks
- BBN Butterfly processor (TC2000) using Butterfly network
- Cray Y-MP multiprocessor, The Alliant FX/2800 uses Crossbar networks



COMPUTER SYSTEMS ARCHITECTURE

Cache Coherence and Synchronization Mechanisms Part1 (Malayalam)

Snoopy Bus Protocol



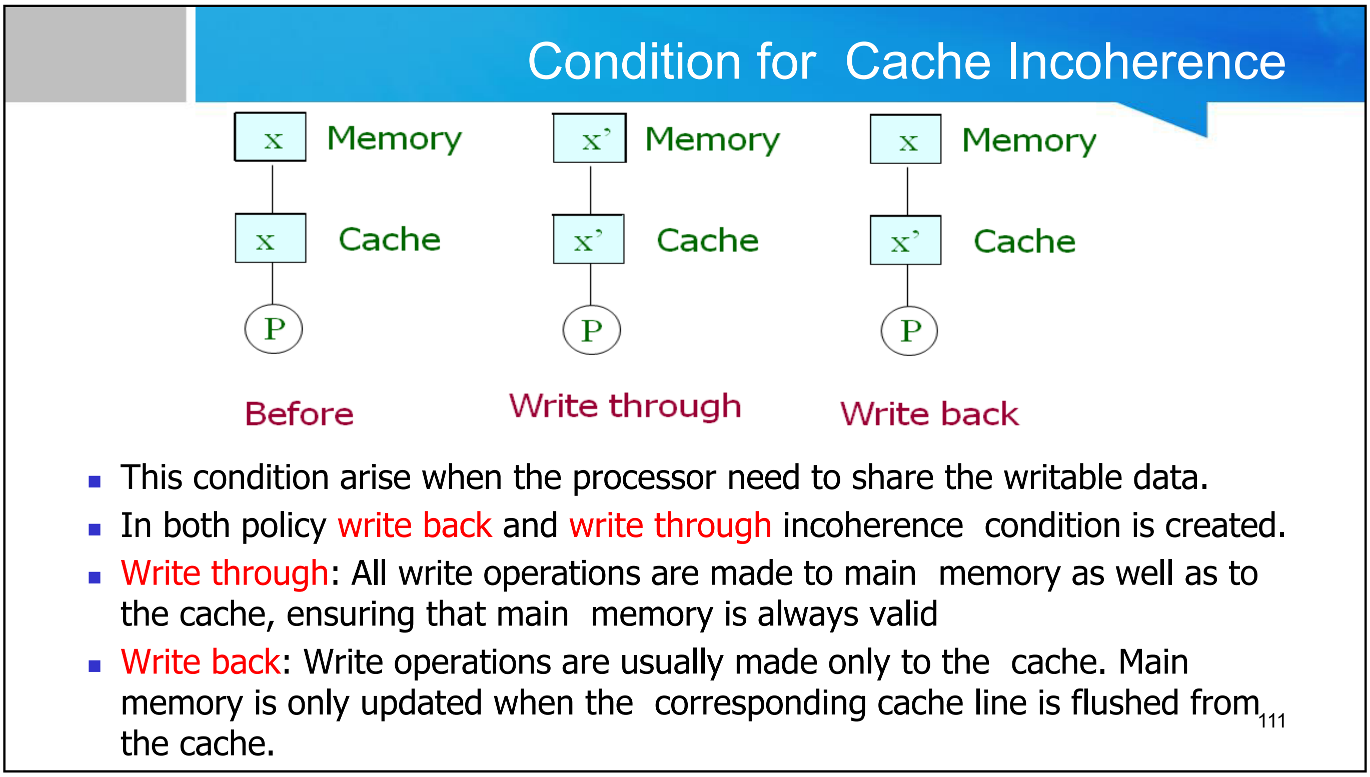
Cache Coherence and Synchronization Mechanisms

Cache Coherence Problem, Snoopy Bus Protocol, Directory Based Protocol, Hardware Synchronization Problem

Cache Coherence In Shared Memory Multiprocessors

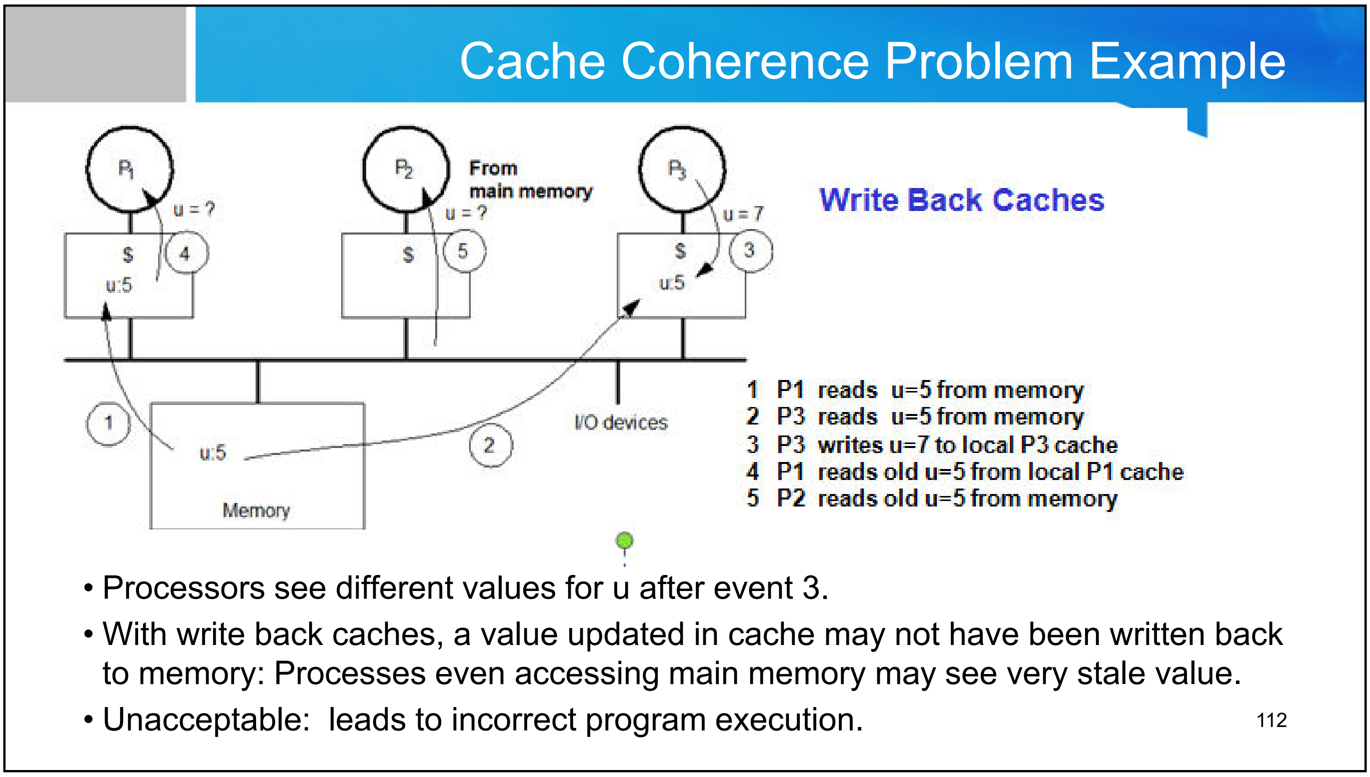
- Caches in a multiprocessing environment introduce the cache coherence problem.
- It is a cache inconsistency problem
- When multiple processors maintain locally cached copies of a unique shared-memory location, any local modification of the location can result in a globally inconsistent view of memory.
- Copies of a variable can be present in multiple caches.
- A write by one processor may not become visible to others:
  - Processors accessing stale (old) value in their private caches

- Caused by:
  - ➢ Inconsistencies due to Sharing writable data
  - ➢ Inconsistencies due to Process migration.
  - ➢ Inconsistencies due to I/O activity.
- Software and/or hardware actions needed to ensure write visibility to all processors thus maintaining cache coherence.
  - i.e. Processors must see the most updated value



Condition for Cache Incoherence

- This condition arise when the processor need to share the writable data.
- In both policy write back and write through incoherence condition is created.
- Write through: All write operations are made to main memory as well as to the cache, ensuring that main memory is always valid
- Write back: Write operations are usually made only to the cache. Main memory is only updated when the corresponding cache line is flushed from the cache.



Cache Coherence Problem Example

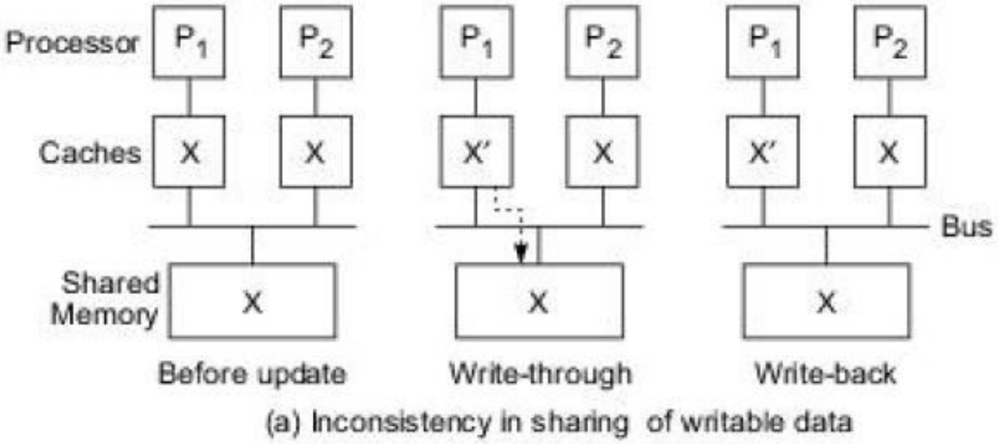
Write Back Caches

1 P1 reads u=5 from memory
2 P3 reads u=5 from memory
3 P3 writes u=7 to local P3 cache
4 P1 reads old u=5 from local P1 cache
5 P2 reads old u=5 from memory

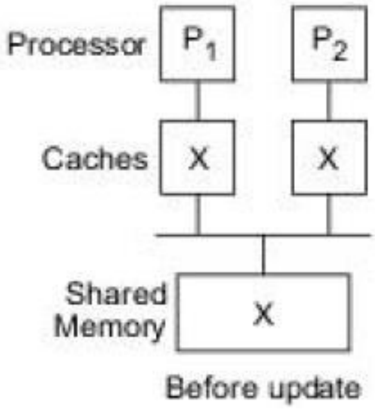
- Processors see different values for u after event 3.
- With write back caches, a value updated in cache may not have been written back to memory: Processes even accessing main memory may see very stale value.
- Unacceptable: leads to incorrect program execution.

1. Inconsistency in Data Sharing

- The cache inconsistency problem occurs only when multiple private caches are used.
- three sources of the problem are
  - sharing of writeable data
  - Process migration, and
  - I/O activity

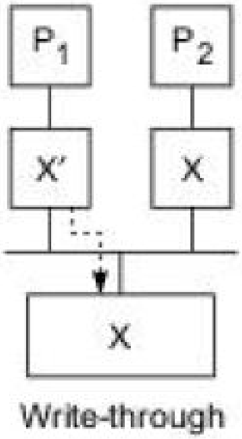


(a) Inconsistency in sharing of writable data



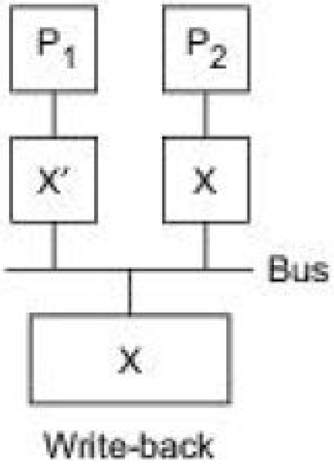
- multiprocessor with two processors, each using a private cache and both sharing the main memory.
- Let X be a shared data element which has been referenced by both processors.
- Before update, the three copies of X are consistent.

Write through



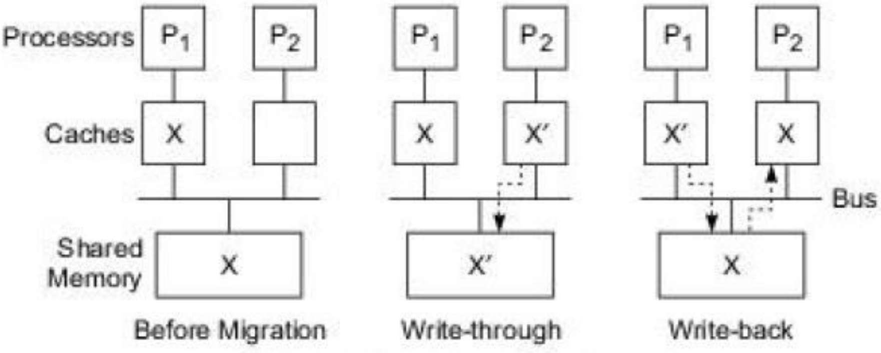
- If processor P1 writes new data X' into the cache, the same copy will be written immediately into the shared memory under a write-through policy.
- In this case. inconsistency occurs between the two copies (X' and X) in the two caches

Write Back



- inconsistency may also occur when a write-back policy is used
- The main memory will be eventually updated when the modified data in the cache are replaced or invalidated

2. Process Migration Inconsistency

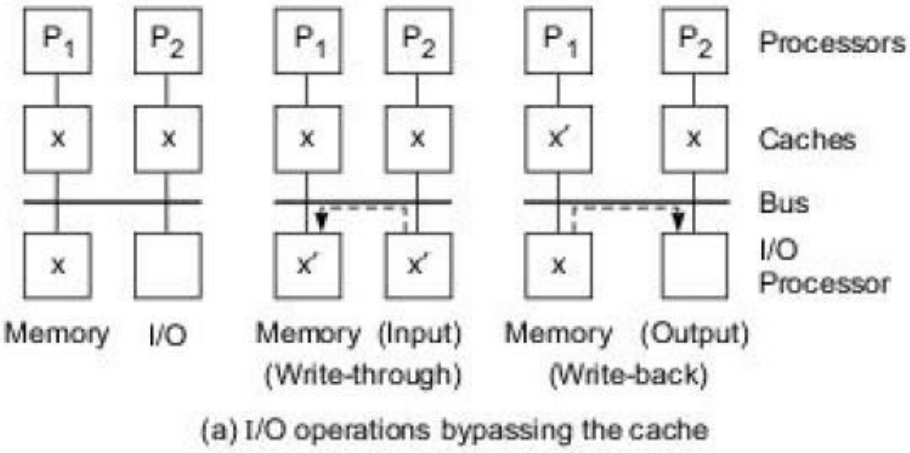


(b) Inconsistency after process migration

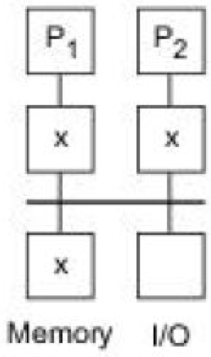
- the occurrence of inconsistency after a process containing a shared variable X migrates from processor 1 to processor 2 using the write-back cache on the right.
- In the middle, a process migrates from processor 2 to processor 1 when using write-through caches.

- In both cases, inconsistency appears between the two cache copies, labelled X and X’.
- Special precautions must be exercised to avoid such inconsistencies.
- A coherence protocol must be established before processes can safely migrate from one processor to another

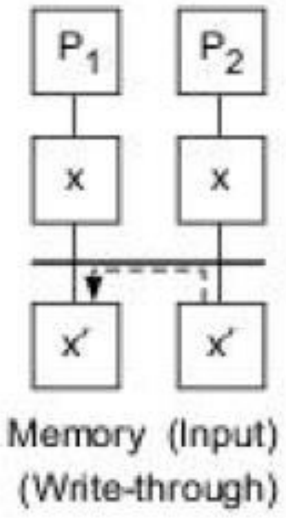
3. I/O Operation Inconsitencies



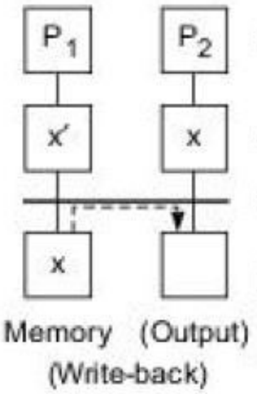
(a) I/O operations bypassing the cache



- Inconsistency problems may occur during I/O operations that bypass the caches.

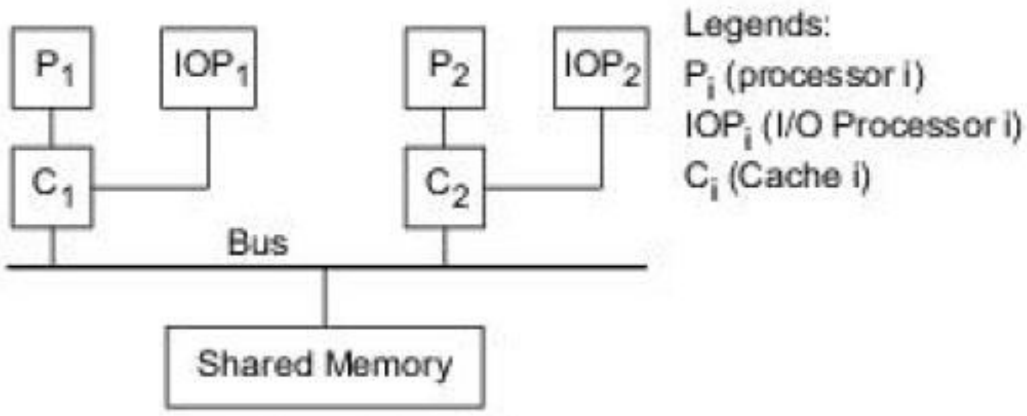


- when the I/O processor loads new data X’ into the main memory, bypassing the write through caches, inconsistency occurs between cache 1 and the shared memory.



- When outputting a data directly from the shared memory (bypassing the caches), the write-back caches also create inconsistency.

A Possible Solution to I/O Problem



(b) A possible solution

- attach the I/O processors (IOP1, and IOP2) to the private caches (C1 and C2) respectively
- 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.
- shortcoming
  - likely increase in cache perturbations and
  - the poor locality of I/O data, which may result in higher miss ratios.

Cache Coherence Approach

1. Snoopy Bus Protocol
   - ➢ Used in bus-based systems where all processors observe memory transactions and take proper action to invalidate or update local cache content if needed.
2. Directory Based Approach
   - ➢ Used in scalable cache-coherent multiprocessor systems where cache directories are used to store where copies of cache blocks reside.
3. Shared Caches:
   - ➢ No private caches.
   - ➢ This limits system scalability.

4. Non-cacheable Data:
   - Not to cache shared writable data:
   - Locks, process queues.
   - Data structures protected by critical sections.
   - Only instructions or private data is cacheable.
   - Data is tagged by the compiler.
5. Cache Flushing
   - Flush cache whenever a synchronization primitive is executed.
   - Slow unless special hardware is used.

Two Protocol Approach

- Protocols to ensure coherence are called snoopy protocols because each cache snoops on the transactions of other caches.
- Scalable multiprocessor systems interconnect processors using short point-to-point links in direct or multistage networks.
  - In such systems, the cache coherence problem can be solved using some variant of directory schemes.

COMPUTER SYSTEMS ARCHITECTURE

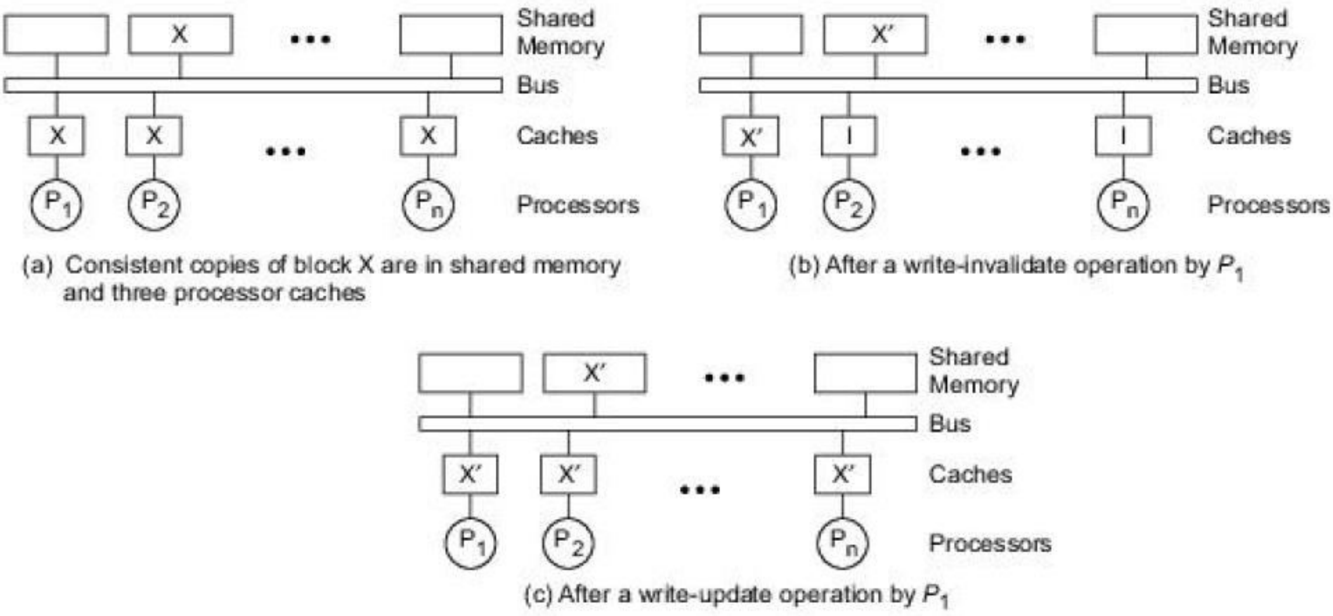
SNOOPY BUS PROTOCOL (Malayalam)

Cache Coherence and Synchronization Mechanisms Part2

- Used in bus-based systems where all processors observe memory transactions and take proper action to invalidate or update local cache content if needed.
- each cache snoops on the transactions of other caches.
- Applicable to private caches associated with processors tied to a common bus,

- two approaches have been practiced for maintaining cache consistecncy
  - Write invalidate
    - write-invalidate policy will invalidate all remote copies when a local cache block is updated
  - Write update policies
    - Write update policy will broadcast the new data block to all caches containing a copy of the block

- The memory copy is also updated if write-through caches are used.
- In using write-back caches, the memory copy is updated later at block replacement time.
- Write-once Protocol 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.



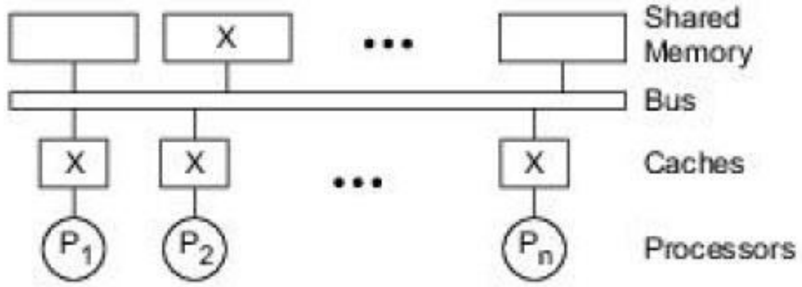
(a) Consistent copies of block X are in shared memory and three processor caches

(b) After a write-invalidate operation by P_1

(c) After a write-update operation by P_1

Fig. 7.14 Write-invalidate and write-update coherence protocols for write through caches (1: invalidate)

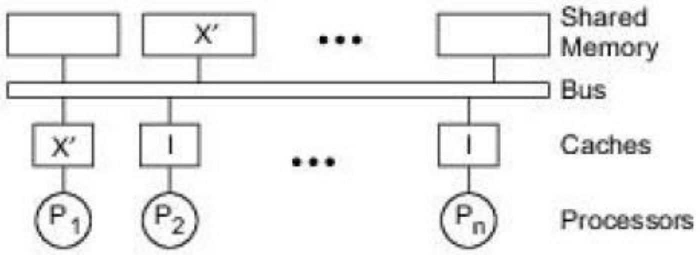
- Snoopy protocols achieve data consistency among the caches and shared memory through a bus watching mechanism.
- two snoopy bus protocols create different results.



(a) Consistent copies of block X are in shared memory and three processor caches

- Consider three processors (P1, P2, and P3)
- maintaining Consistent copies of block X in their local caches and in the shared-memory module marked X.

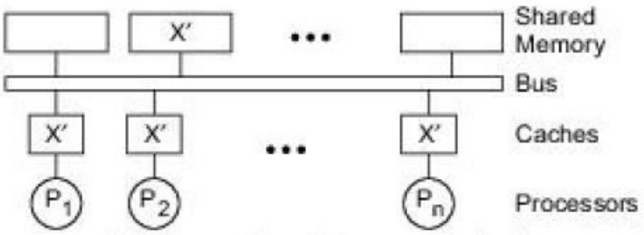
Write Invalidate protocol



(b) After a write-invalidate operation by P_1

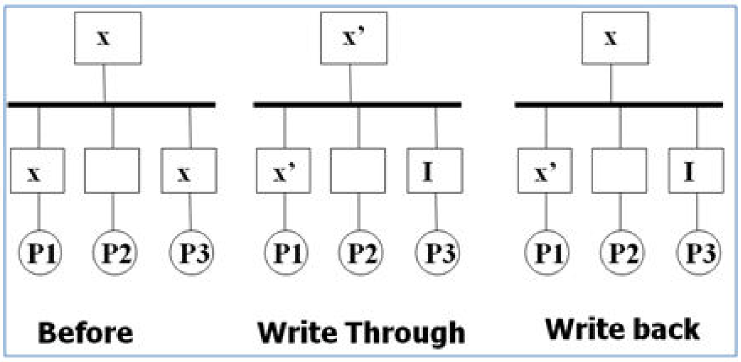
- Using a write-in validate protocol, the processor P1 modifies (writes) its cache from X to X’, and all other copies are invalidated via the bus
- invalidated blocks are sometimes called dirty, meaning they should not be used

Write Update Protocol



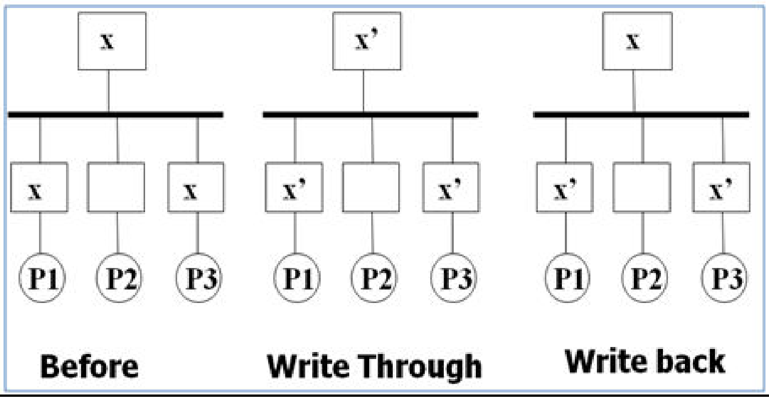
(c) After a write-update operation by P_1

- The write update protocol demands the new block content X' be broadcast to all cache copies via the bus. The memory copy is also updated if write-through caches are used.
- In using write-back caches, the memory copy is updated later at block replacement time.



Write-invalidate

Write-Update



Write-Through Caches

- The states of a cache block copy change with respect to read, write, and replacement operations in the cache.
- A block copy of a write-through cache *i* attached to processor *i* can assume one of two possible cache states: Valid or Invalid.
- In a valid state
  - all processors can read (R(i), R(j)) safely.
  - Local processor *i* can also write (W (i)) safely in a valid state.
- The invalid state
  - corresponds to the case of the block either being invalidated or being replaced (Z(i) or Z(j)).

- A remote processor writes (W(j)) into its cache copy, all other cache copies become invalidated.
- The cache block in cache *i* becomes valid whenever a successful read (R(i)) or write (W(i)) is carried out by a local processor *i*.

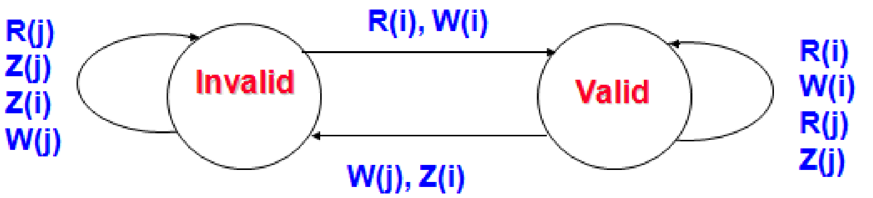
Write-invalidate Snoopy Bus Protocol: For Write-Through Caches

- The states of a cache block copy change with respect to read, write, and replacement operations in the cache.
- The state of a cache block copy of processor *i* can take one of two states (*j* represents a remote processor):
  - Valid State:
    - •All processors can read *(R(i),R(j))* safely.
    - •Local processor *i* can also write *(W(i))*
    - •In this state after a successful read *((R(i))* or write *(W(i))*
  - Invalid State: not in cache or,
    - •Block being invalidated.
    - •Block being replaced *(Z(i)* or *(Z(j))*
    - •When a remote processor writes *((W(j))* to its cache copy, all other cache copies become invalidated.
  - Bus write cycles are higher than bus read cycles due to request invalidations.

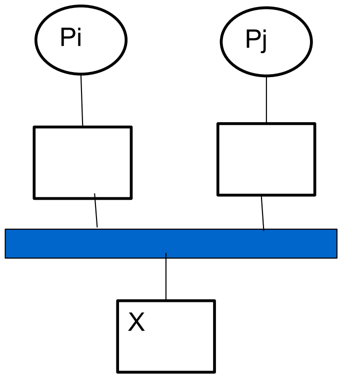
Write-invalidate Snoopy Bus Protocol

For Write-Through Caches

State Transition Diagram

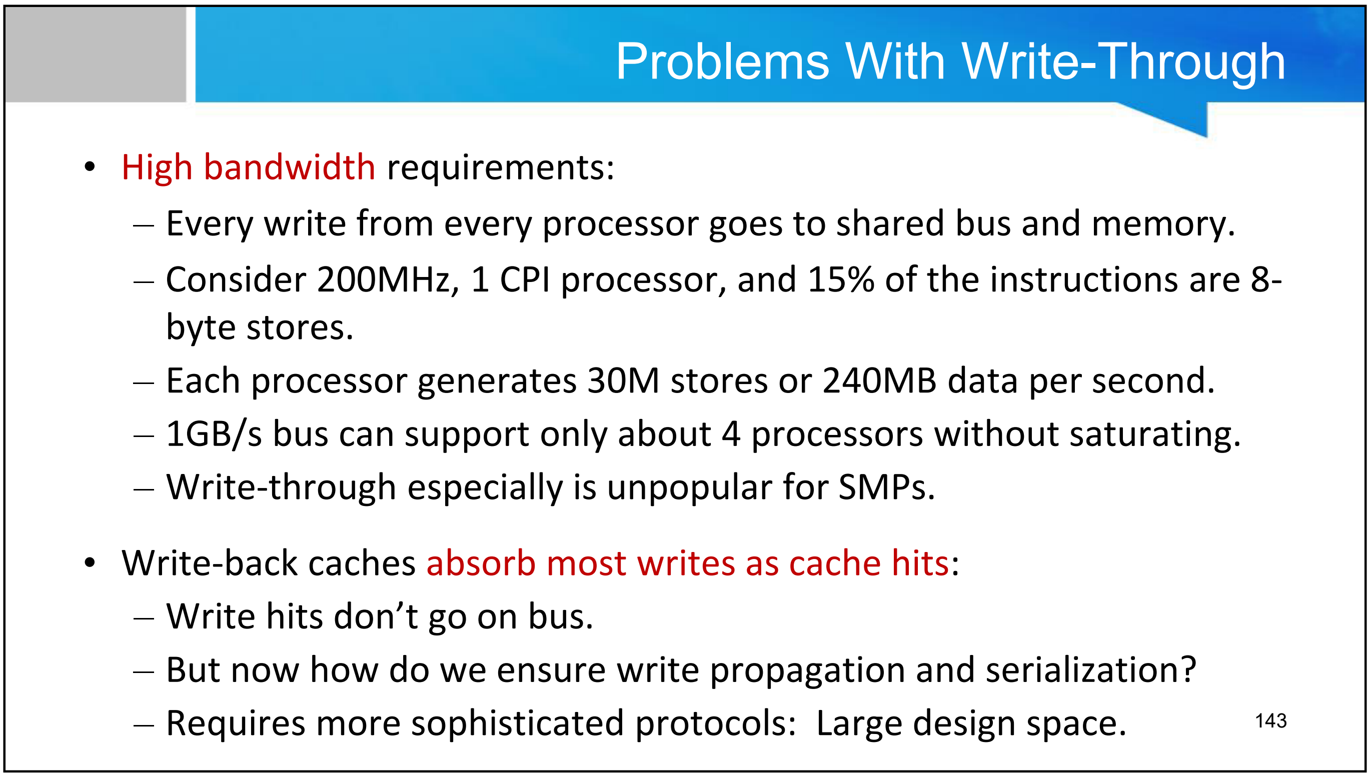


W(i) = Write to block by processor i
W(j) = Write to block copy in cache j by processor j ≠ i
R(i) = Read block by processor i.
R(j) = Read block copy in cache j by processor j ≠ i
Z(i) = Replace block in cache .
Z(j) = Replace block copy in cache j ≠ i



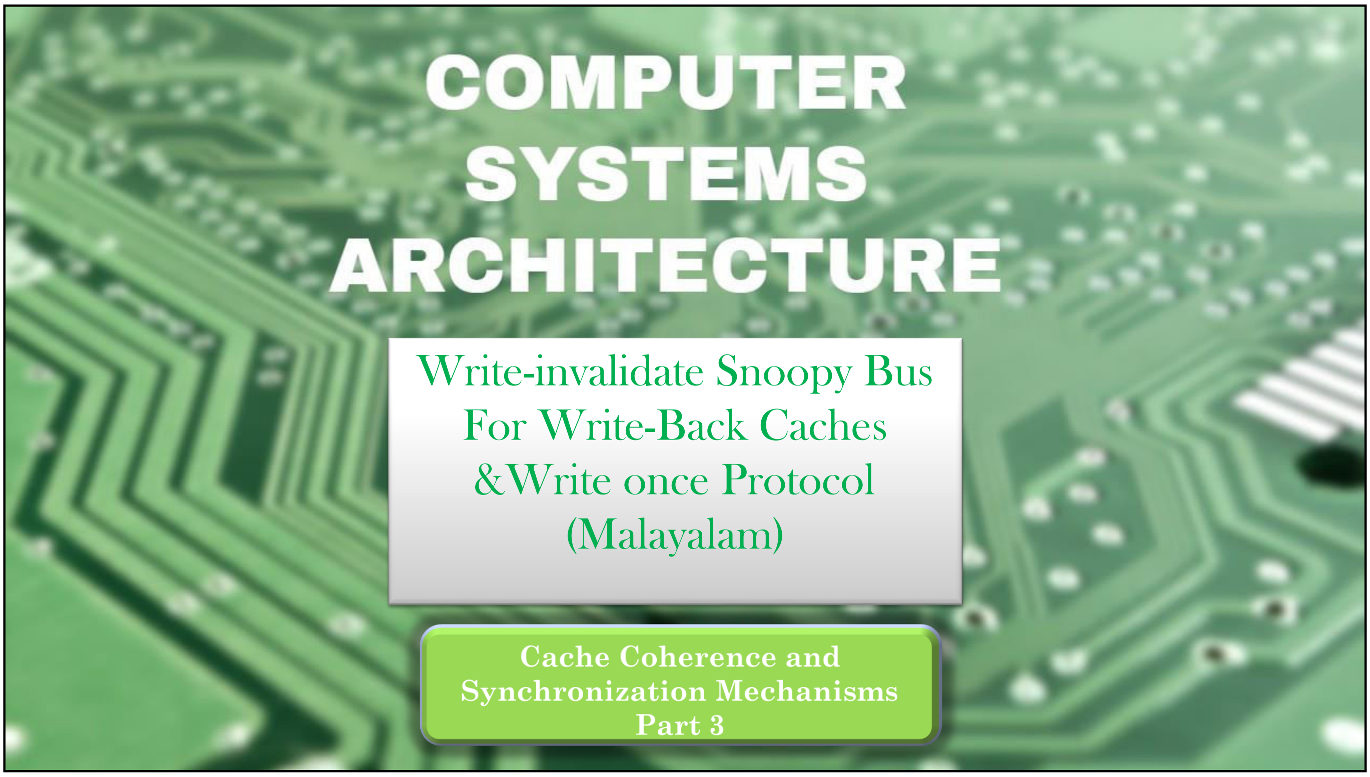
Shared memory module

i - local processor j - other processor



Problems With Write-Through

- High bandwidth requirements:
  - Every write from every processor goes to shared bus and memory.
  - Consider 200MHz, 1 CPI processor, and 15% of the instructions are 8-byte stores.
  - Each processor generates 30M stores or 240MB data per second.
  - 1GB/s bus can support only about 4 processors without saturating.
  - Write-through especially is unpopular for SMPs.
- Write-back caches absorb most writes as cache hits:
  - Write hits don't go on bus.
  - But now how do we ensure write propagation and serialization?
  - Requires more sophisticated protocols: Large design space.



COMPUTER SYSTEMS ARCHITECTURE

Write-invalidate Snoopy Bus For Write-Back Caches &Write once Protocol (Malayalam)

Cache Coherence and Synchronization Mechanisms Part 3

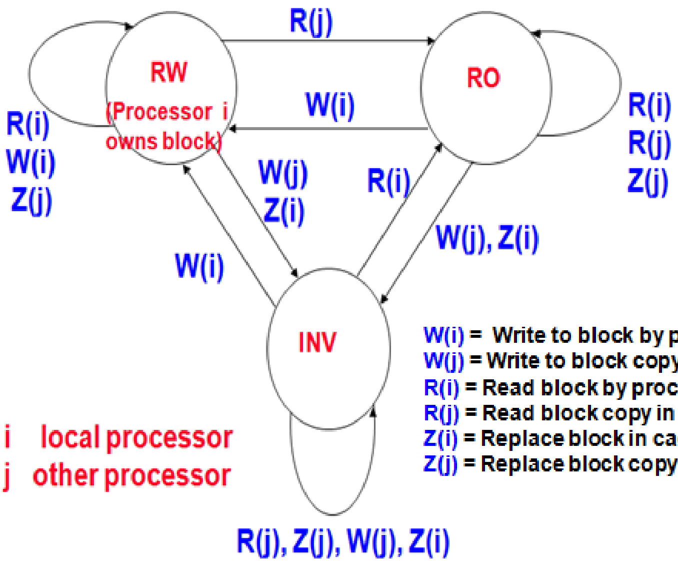
Write-invalidate Snoopy Bus Protocol: For Write-Back Caches

- Corresponds to ownership protocol.
- The valid state in write-through protocol is divided into two states :
  - RW (read-write): (this processor i owns block)
    - The only cache copy existing in the system; owned by the local processor.
    - Read *(R(i))* and *(W(i))* can be safely performed in this state.
  - RO (read-only):
    - Multiple cache block copies exist in the system; owned by memory.
    - Reads *((R(i)), ((R(j))* can safely be performed in this state.
  - INV (invalid):
    - Entered when : Not in cache or,
    - A remote processor writes *((W(j))* to its cache copy.
    - A local processor replaces *((Z(i))* its own copy.

- A cache block is uniquely owned after a local write *((W(i))*
- Before a block is modified, ownership for exclusive access is obtained by a read-only bus transaction broadcast to all caches and memory.
- If a modified remote block copy exists, memory is updated, local copy is invalidated and ownership transferred to requesting cache.

Write-invalidate Snoopy Bus Protocol For Write-Back Caches

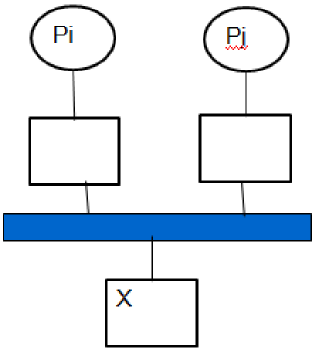
State Transition Diagram



i local processor
j other processor

RW: Read-Write
RO: Read Only
INV: Invalidated or not in cache

W(i) = Write to block by processor i
W(j) = Write to block copy in cache j by processor j ≠ i
R(i) = Read block by processor i.
R(j) = Read block copy in cache j by processor j ≠ i
Z(i) = Replace block in cache.
Z(j) = Replace block copy in cache j ≠ i

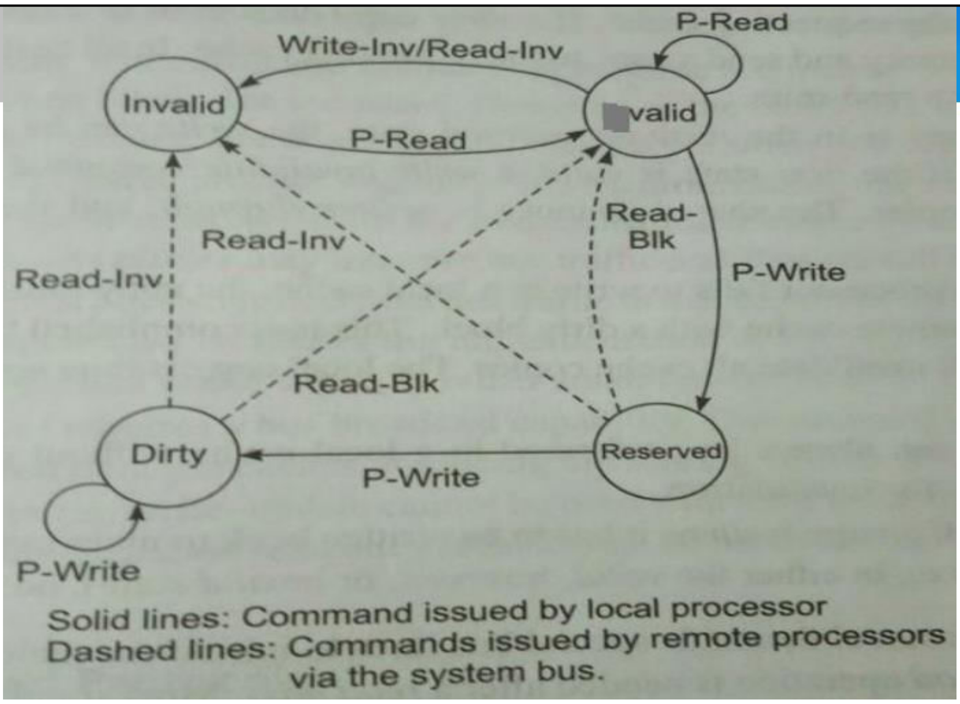


Shared memory module

- The valid state of a write-back cache can be further split into two cache states,
  - labeled RW (read-write) and RO (read-only).
  - The INV cache state is equivalent to the invalid state.
- When the memory owns a block, caches can contain only the RO copies of the block.
- 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 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.

3. Write-once Protocol

- Cache coherence protocol for bus-based multiprocessors.
- combines the advantages of both write-through and write-back 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.



Solid lines: Command issued by local processor
Dashed lines: Commands issued by remote processors via the system bus.

❑ The four cache states are: **Valid, Invalid, Reserved and Dirty.**

- 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).

- 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.
- The bus-read command corresponds to a normal memory read by a remote processor via the bus

Cache Events and Actions

❑ The memory-access and invalidation commands trigger following events and actions:

➢**Read-miss :** no copy…like **INVALID**

- When a processor wants to read a block and it is not in the cache, a read-miss occurs.
- This initiates a bus-read operation.
- If no dirty copy exists, then the main memory that has a consistent copy, supplies a copy to the requesting cache memory.
- If a dirty copy exists in a remote cache memory, that cache will restrain the main memory and send a copy to the requesting cache memory.
- In both cases, the cache copy will enter valid state after a read miss.

Write Once Protocol

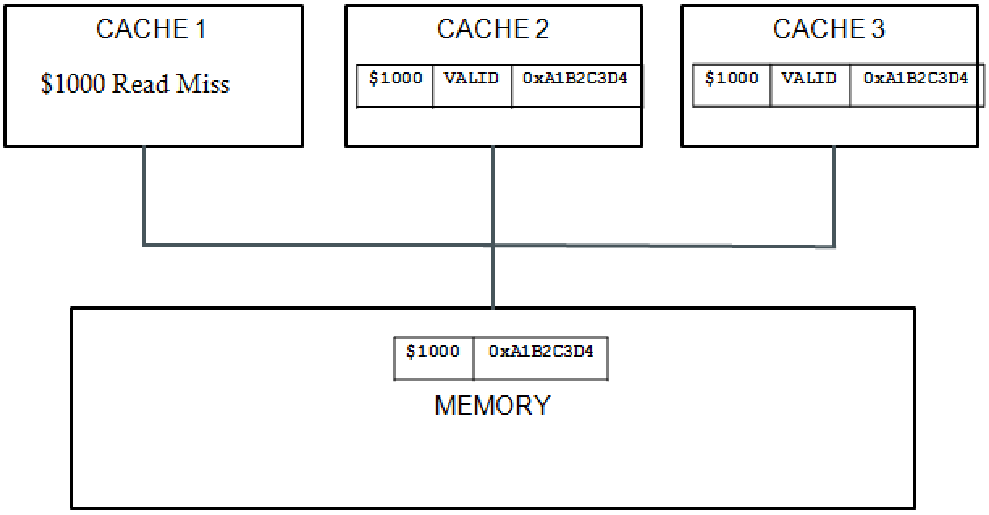
Read Miss

Scenario 1

Write Once Protocol

Read Miss

Scenario 1



CACHE 1

$1000 Read Miss

CACHE 2

$1000 VALID 0xA1B2C3D4

CACHE 3

$1000 VALID 0xA1B2C3D4

$1000 0xA1B2C3D4

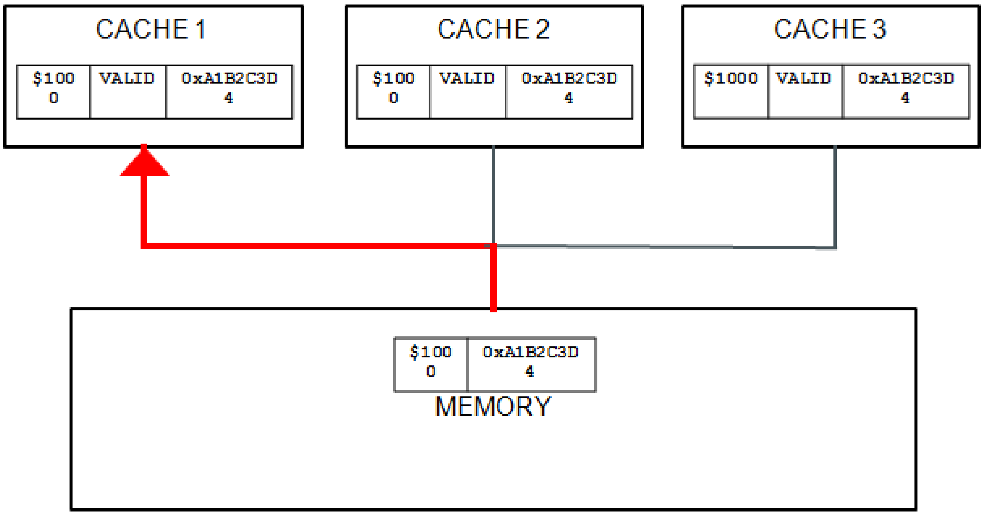
MEMORY

Write Once Protocol

Bus-Read Command

Read Miss..

Scenario 1



CACHE 1

$1000 VALID 0xA1B2C3D4

CACHE 2

$1000 VALID 0xA1B2C3D4

CACHE 3

$1000 VALID 0xA1B2C3D4

$1000 0xA1B2C3D4

MEMORY

Write Once Protocol

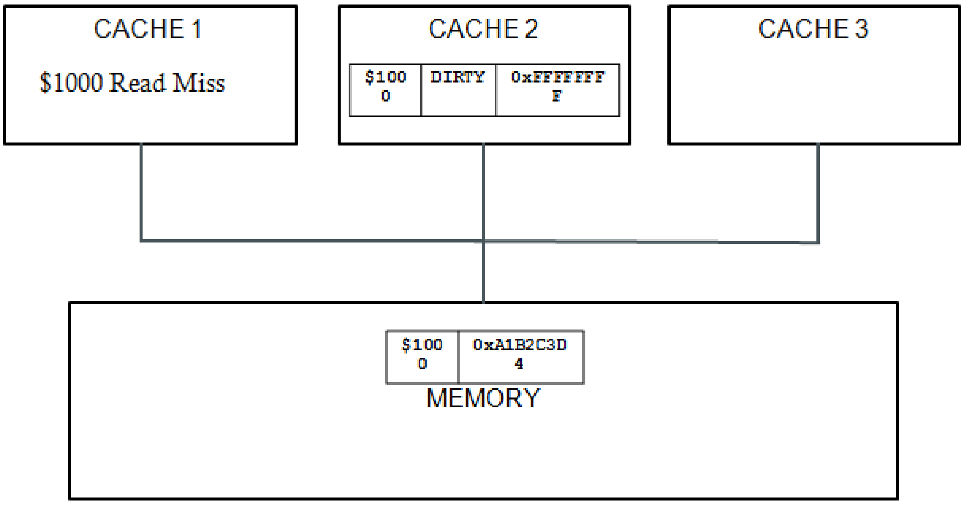
Read Miss

Scenario 2

Write Once Protocol

Read Miss

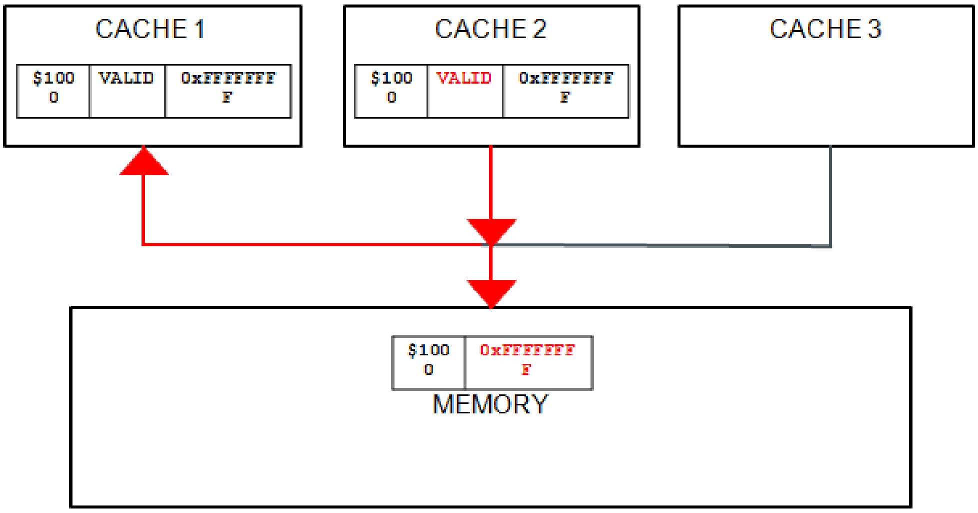
Scenario 2



Write Once Protocol

Bus-Read Command

Read Miss.. Scenario 2



➢Write-hit

- If the block state is **DIRTY**, write proceeds locally in cache and still in **DIRTY** state
- If the block state is **RESERVED**, write proceeds locally in cache, but state is set to **DIRTY**
- If the block state is **VALID**, word is written through to memory, and state is changed to **RESERVED**. A **Write-Invalidate command** is broadcasted to all the caches, invalidating their copies .Other caches with the same block observe this bus write, and change their states to INVALID

Write Once Protocol

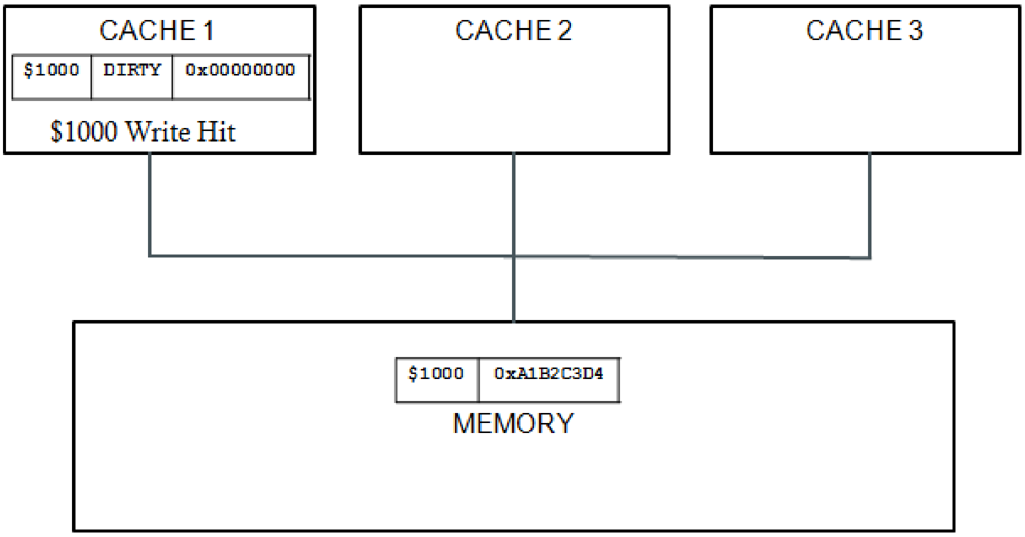
Write Hit

Scenario 1

Write Once Protocol

Write Hit

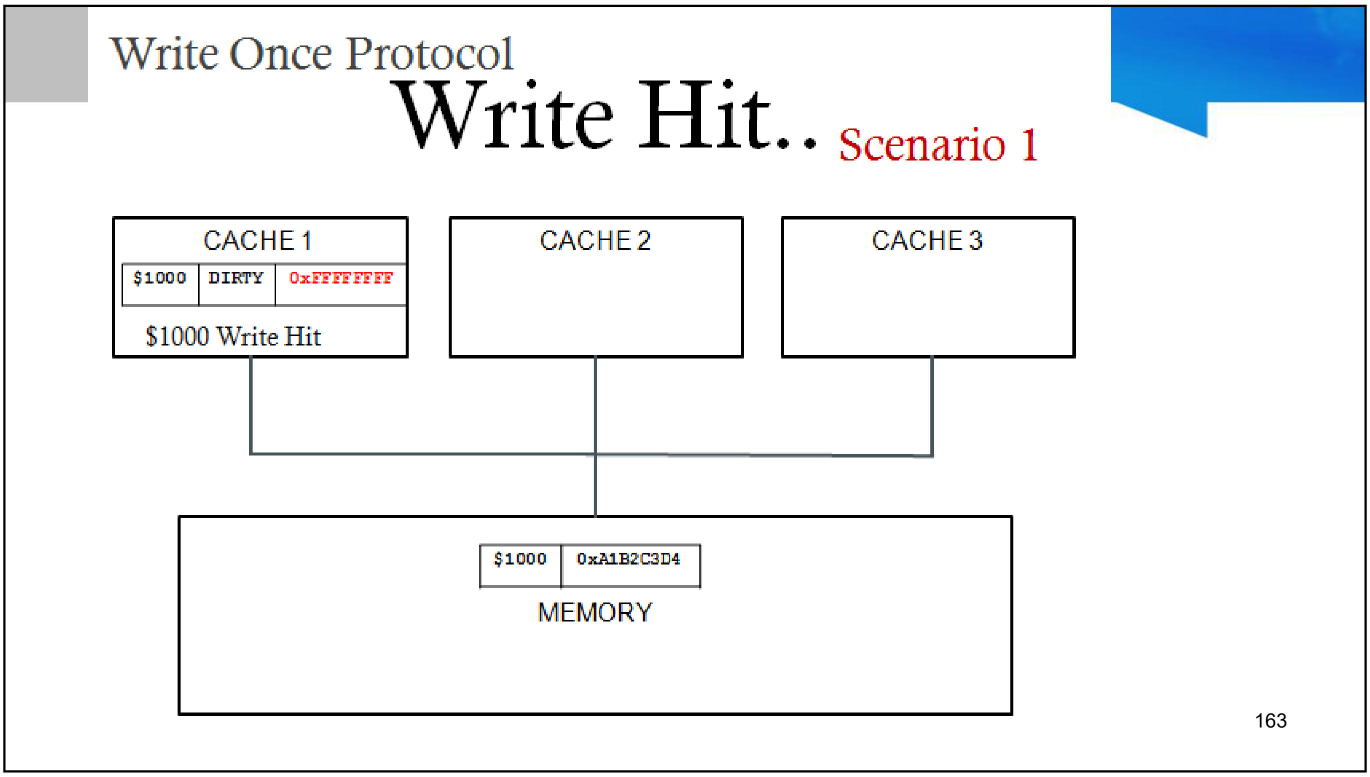
Scenario 1



$1000	DIRTY	0x00000000

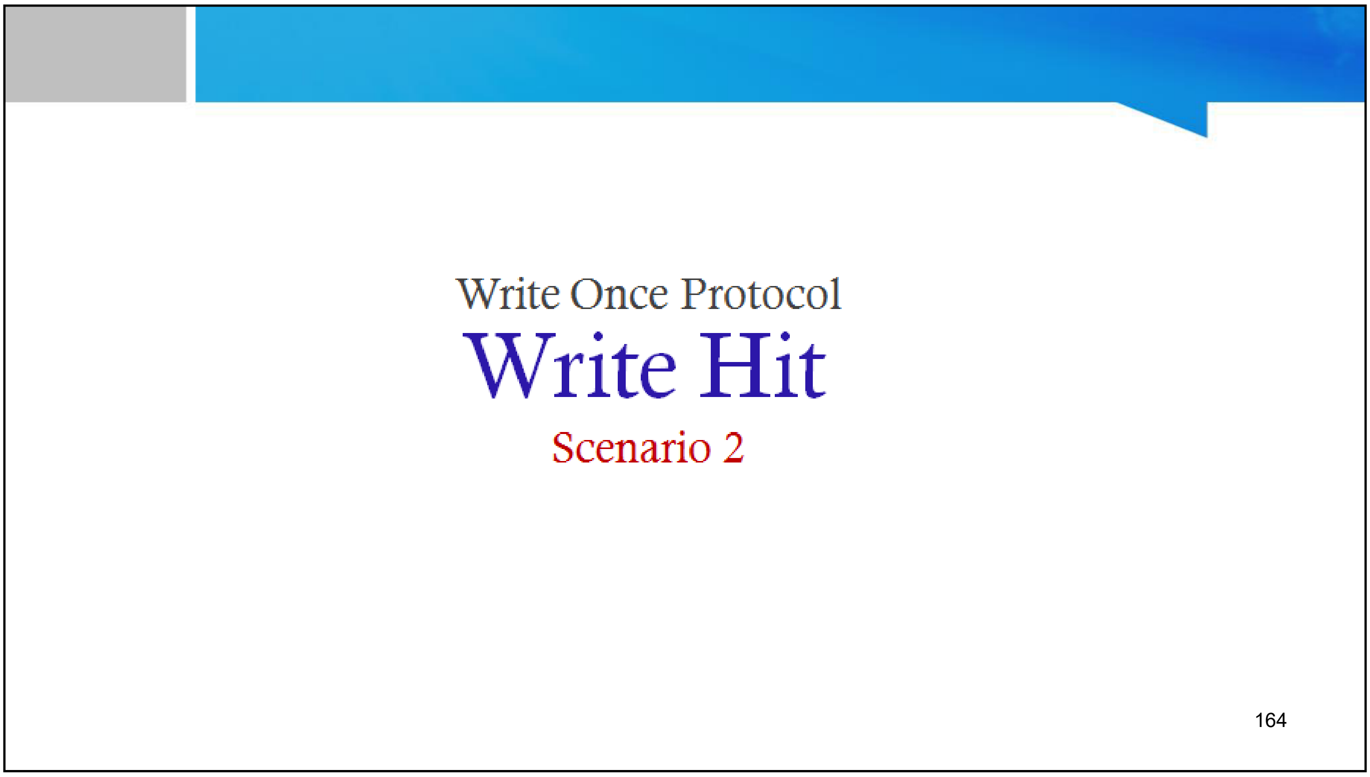
$1000 Write Hit

$1000	0xA1B2C3D4



Write Once Protocol

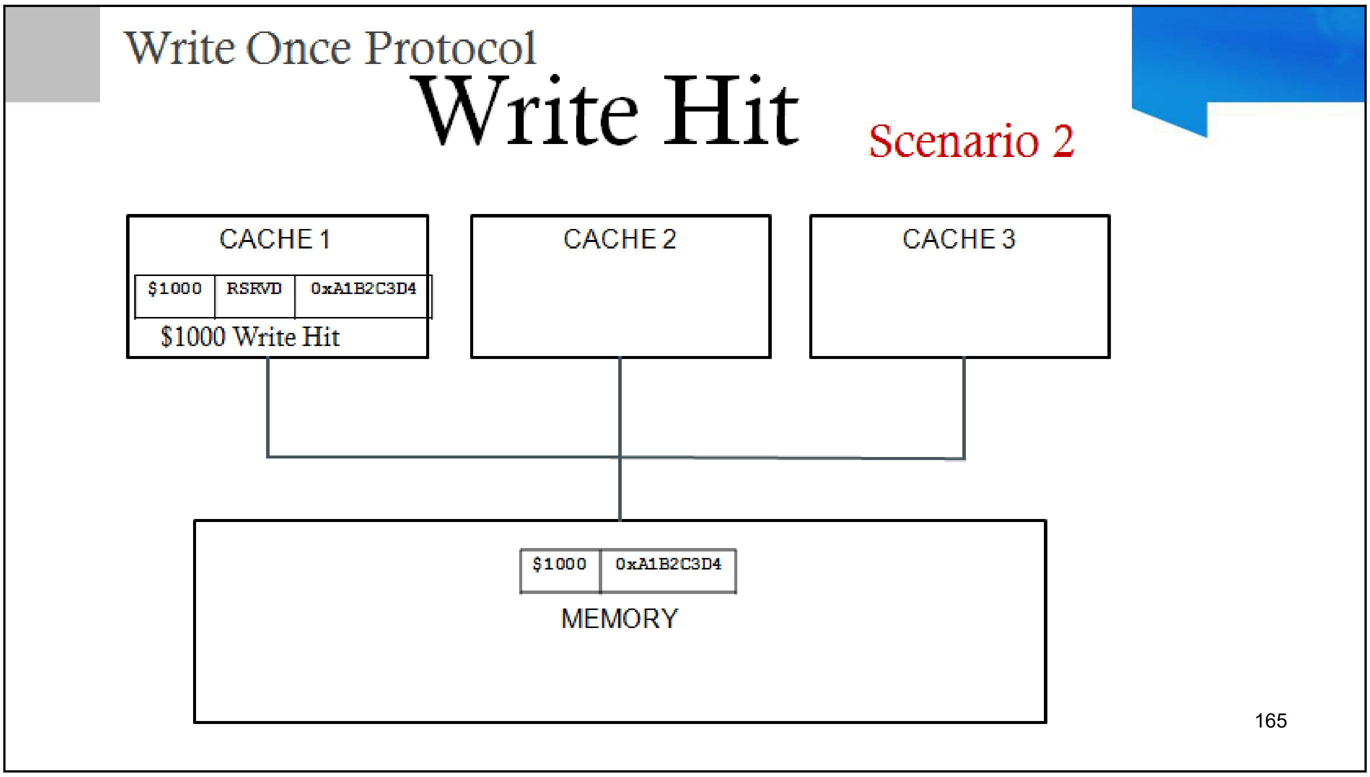
Write Hit.. Scenario 1



Write Once Protocol

Write Hit

Scenario 2



Write Once Protocol

Write Hit

Scenario 2

CACHE 1

$1000	RSRVD	0xA1B2C3D4

$1000 Write Hit

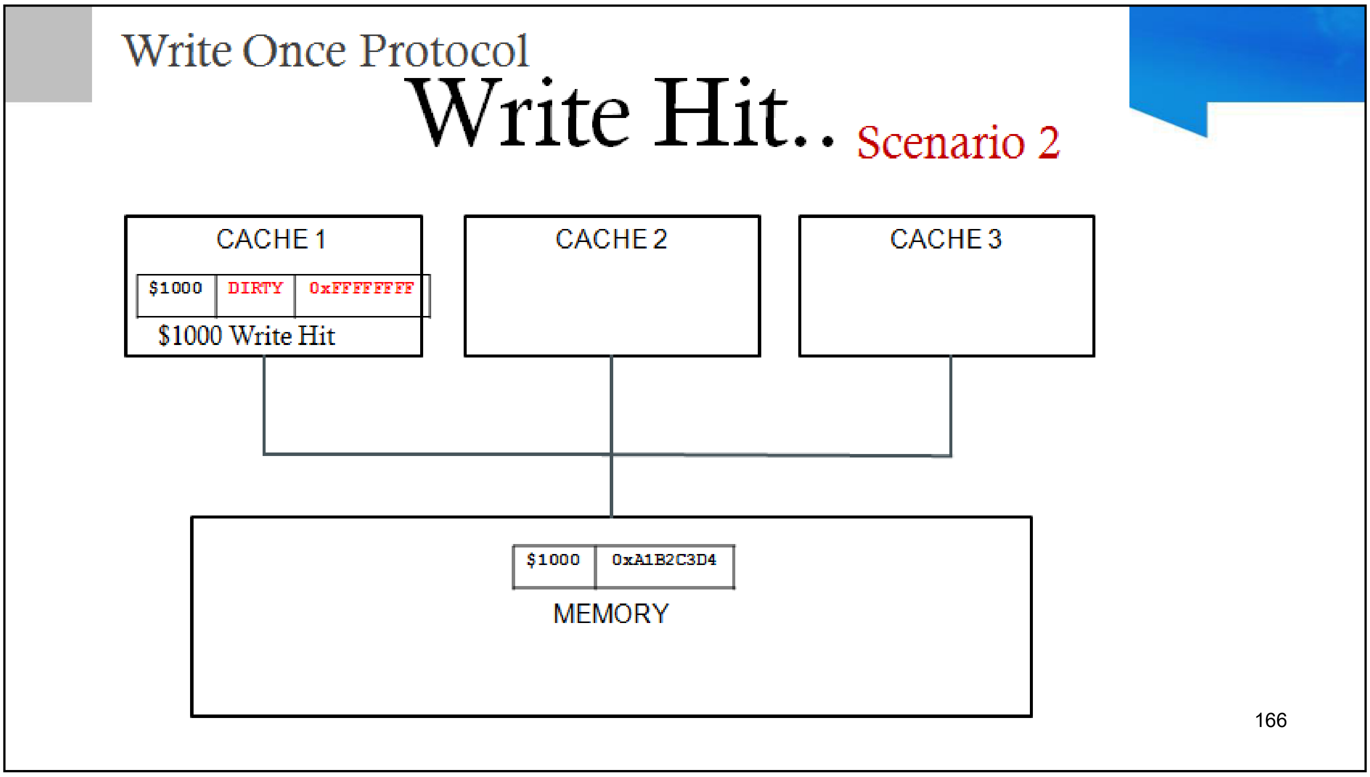
CACHE 2

CACHE 3

$1000	0xA1B2C3D4

MEMORY

165



Write Once Protocol

Write Hit.. Scenario 2

CACHE 1

$1000	DIRTY	0xFFFFFFFF

$1000 Write Hit

CACHE 2

CACHE 3

$1000	0xA1B2C3D4

MEMORY

166

Write Once Protocol

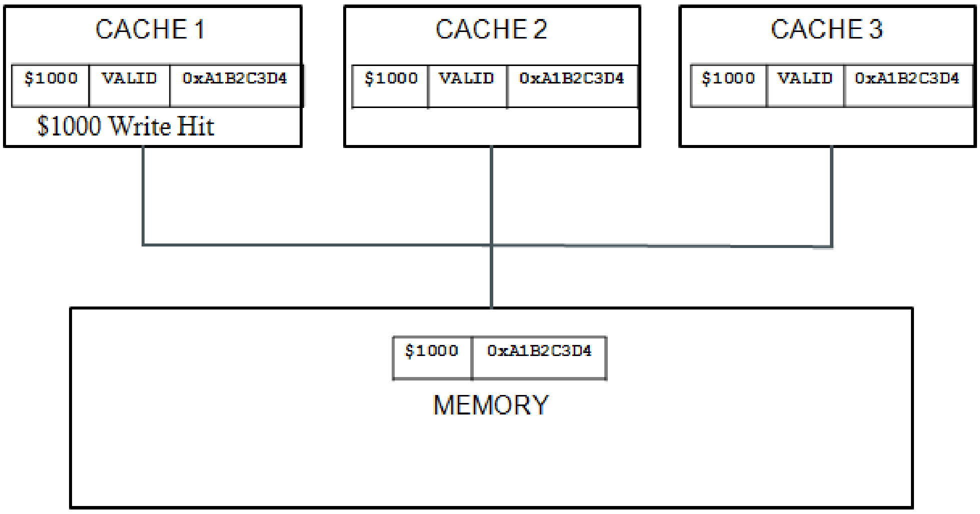
Write Hit

Scenario 3

Write Once Protocol

Write Hit

Scenario 3



CACHE 1

$1000	VALID	0xA1B2C3D4

$1000 Write Hit

CACHE 2

$1000	VALID	0xA1B2C3D4

CACHE 3

$1000	VALID	0xA1B2C3D4

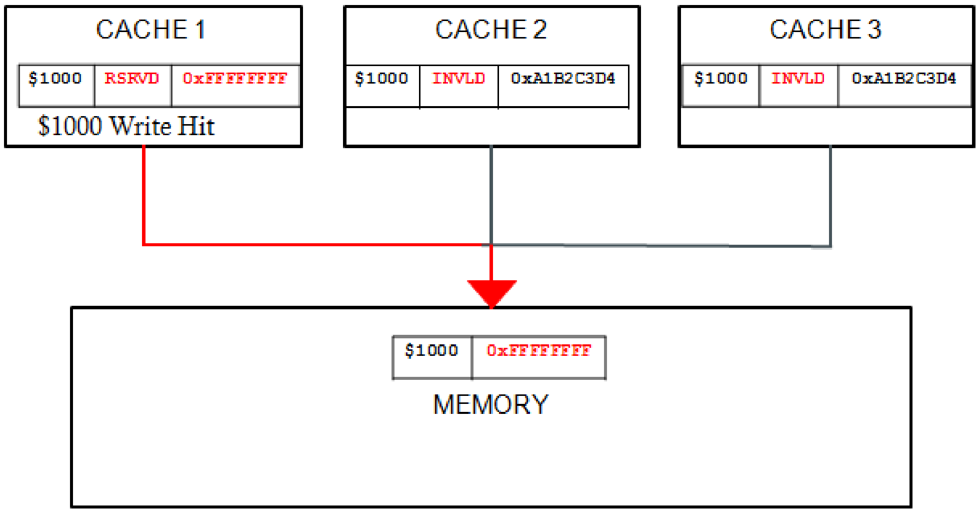
$1000	0xA1B2C3D4

MEMORY

Write Once Protocol

Write - invalidate command

Write Hit.. Scenario 3



➢Write-miss

- If no other cache has a DIRTY copy, then read from memory, state set to **RESERVED**
- If any cache has a DIRTY copy, the block is loaded in the from the cache with DIRTY copy, and that cache is invalidated
- **Read-Invalidate command**, which will invalidate all cache copies.
- Then, local copy proceed with write, and the state is set to **DIRTY**

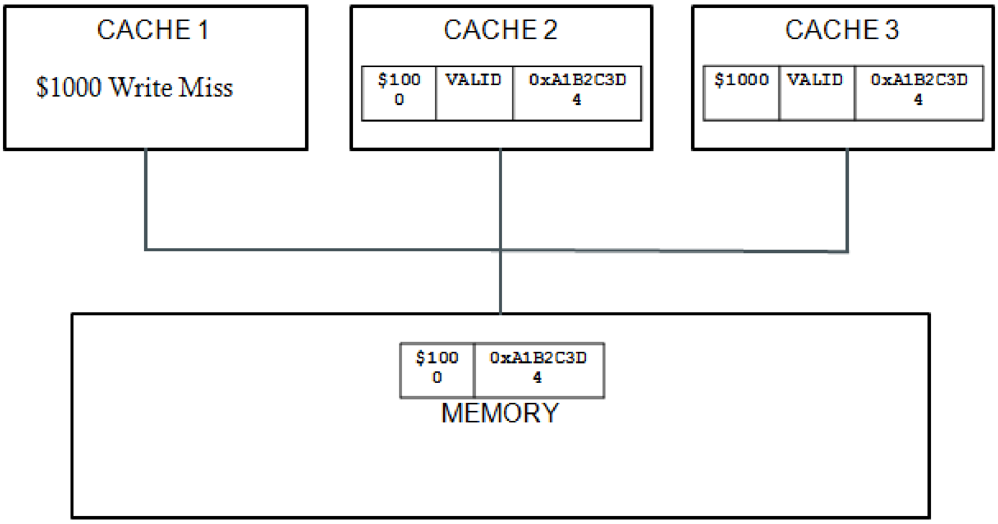
Write Once Protocol

Write Miss

Scenario 1

Write Once Protocol

Write Miss Scenario 1



CACHE 1

$1000 Write Miss

CACHE 2

$100 0	VALID	0xA1B2C3D 4

CACHE 3

$1000	VALID	0xA1B2C3D 4

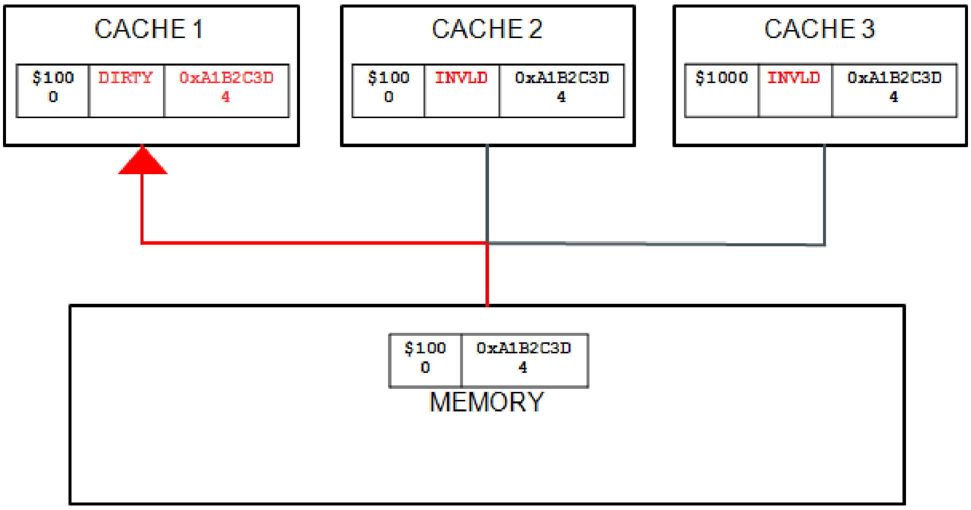
$100 0	0xA1B2C3D 4

MEMORY

Write Once Protocol

Read - invalidate command

Write Miss Scenario 1



CACHE 1

$1000	DIRTY	0xA1B2C3D4

CACHE 2

$1000	INVLD	0xA1B2C3D4

CACHE 3

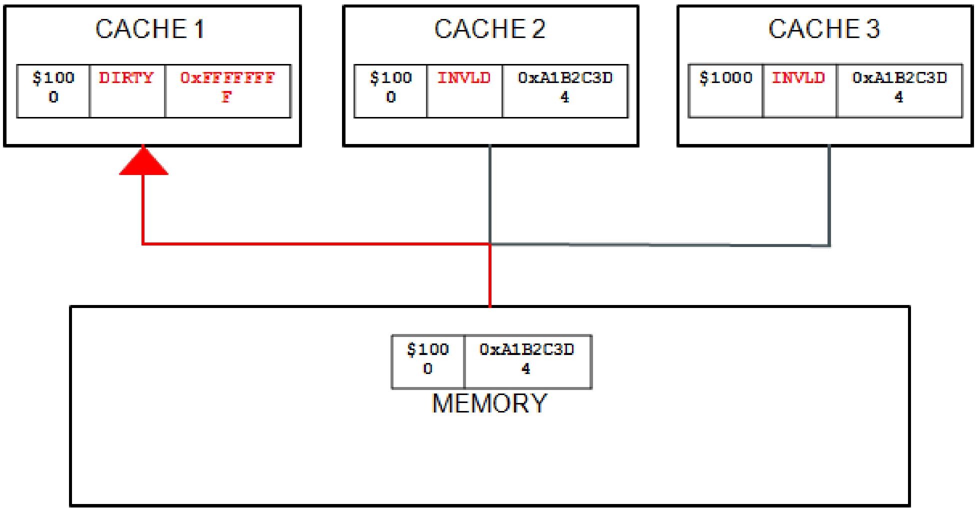
$1000	INVLD	0xA1B2C3D4

$1000	0xA1B2C3D4

MEMORY

Write Once Protocol

Write Miss..Scenario 1



CACHE 1

$1000	DIRTY	0xFFFFFFFF

CACHE 2

$1000	INVLD	0xA1B2C3D4

CACHE 3

$1000	INVLD	0xA1B2C3D4

$1000	0xA1B2C3D4

MEMORY

Write Once Protocol

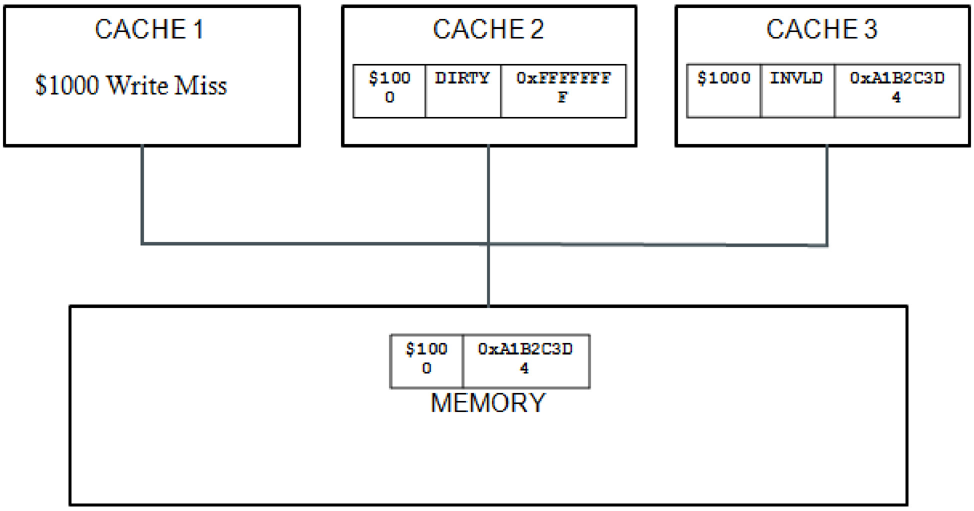
Write Miss

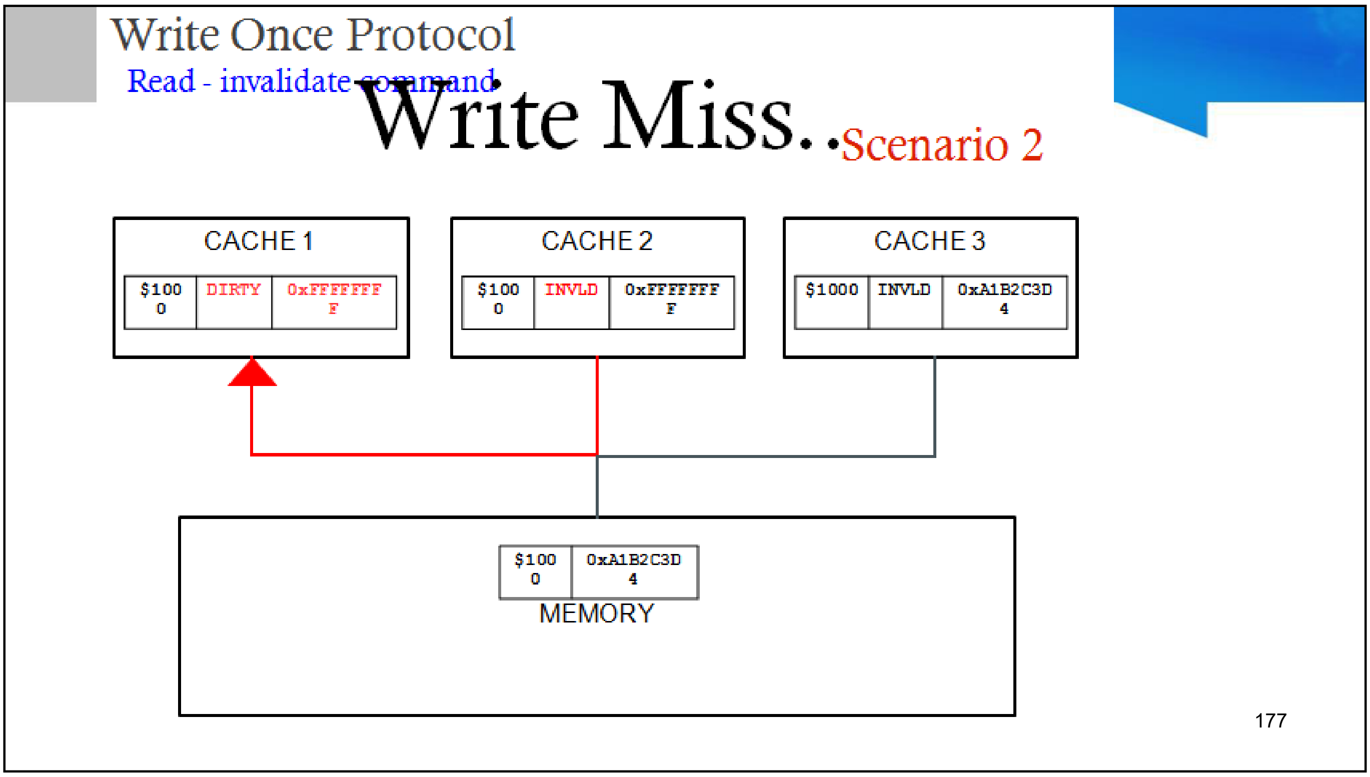
Scenario 2

Write Once Protocol

Write Miss

Scenario 2





Write Once Protocol

Read - invalidate command

Write Miss. •Scenario 2

CACHE 1

$1000	DIRTY	0xFFFFFFFF

CACHE 2

$1000	INVLD	0xFFFFFFFF

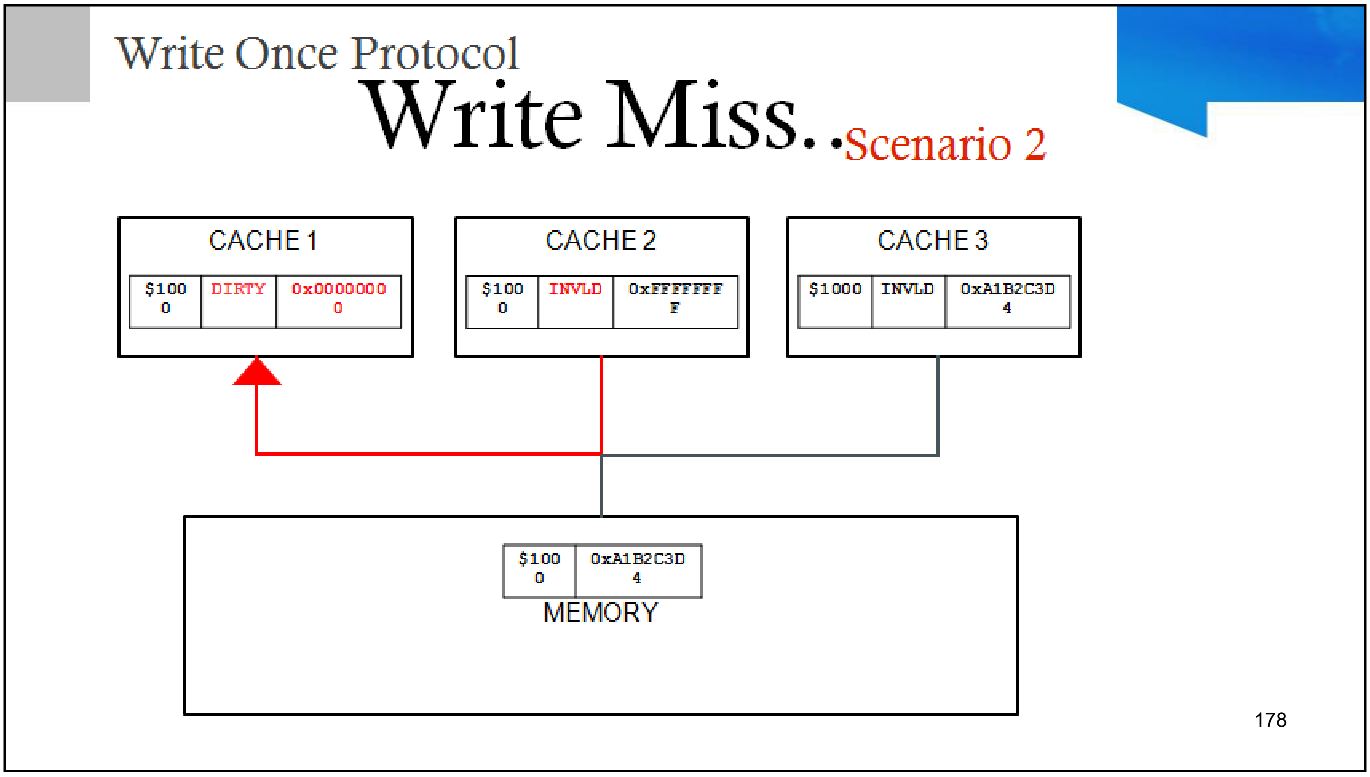
CACHE 3

$1000	INVLD	0xA1B2C3D4

$1000	0xA1B2C3D4

MEMORY

177



Write Once Protocol

Write Miss. •Scenario 2

CACHE 1

$1000	DIRTY	0x00000000

CACHE 2

$1000	INVLD	0xFFFFFFFF

CACHE 3

$1000	INVLD	0xA1B2C3D4

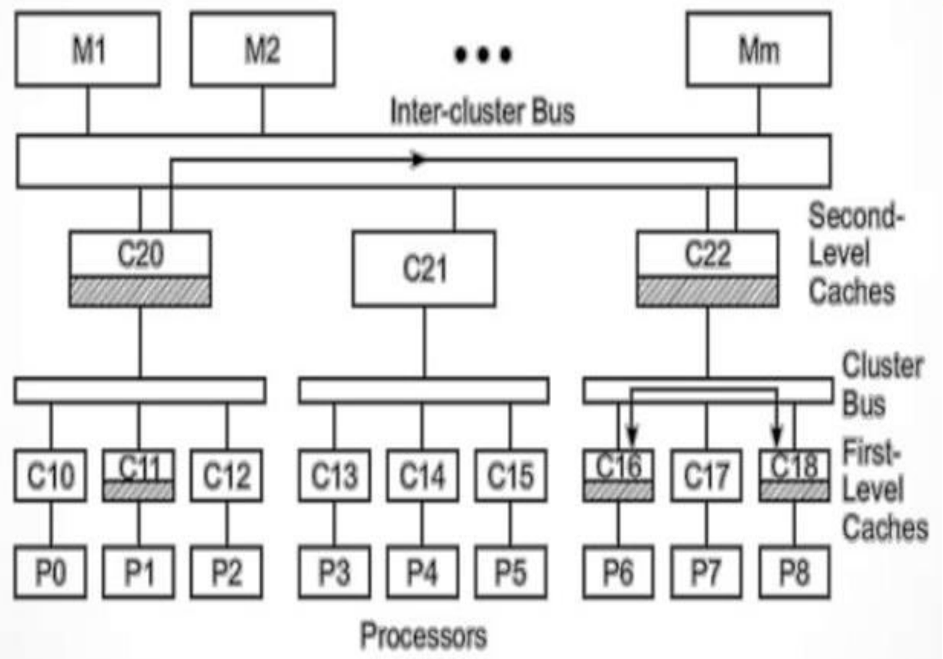
$1000	0xA1B2C3D4

MEMORY

178

- **Read-hit**: Data is in cache **(VALID),**
  - Read-hit is always performed in local cache memory without causing a transition of state or using the snoopy bus for invalidation.
- **Block replacement**
  - When a copy is dirty, it is to be written back to the main memory by block replacement method.
  - However, when the copy is clean (either in valid or reserved or invalid state) , no replacement will take place.

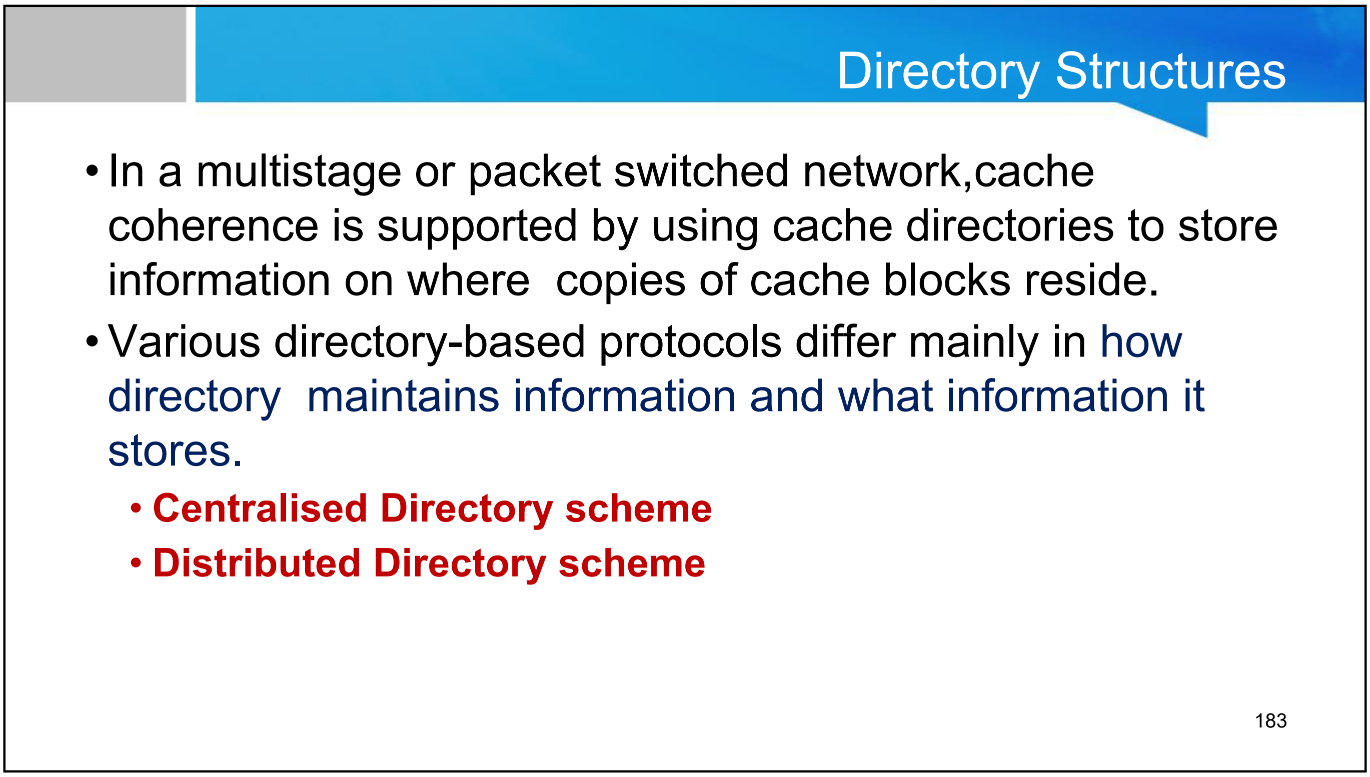
Multilevel Cache Coherence



- An invalidation must propagate vertically up and down in order to invalidate all copies in the shared caches at level 2.
- Suppose processor P1 issues a write request.
- The write request propagates up to highest level and invalidates copies in C20, C22, C16, and C18, to all shaded copies.
- High-level caches like C20 keep track of dirty blocks beneath them.
- A subsequent read request issued by P7 will propagate up the hierarchy because no copies exist.
- When it reaches the top level, cache C20 issues a flush request down to cache C11 and dirty copy is supplied to C11 (private cache of P7).
- An invalidation command or a read request will not propagate down to clusters that do not contain a copy of corresponding block. (C21)

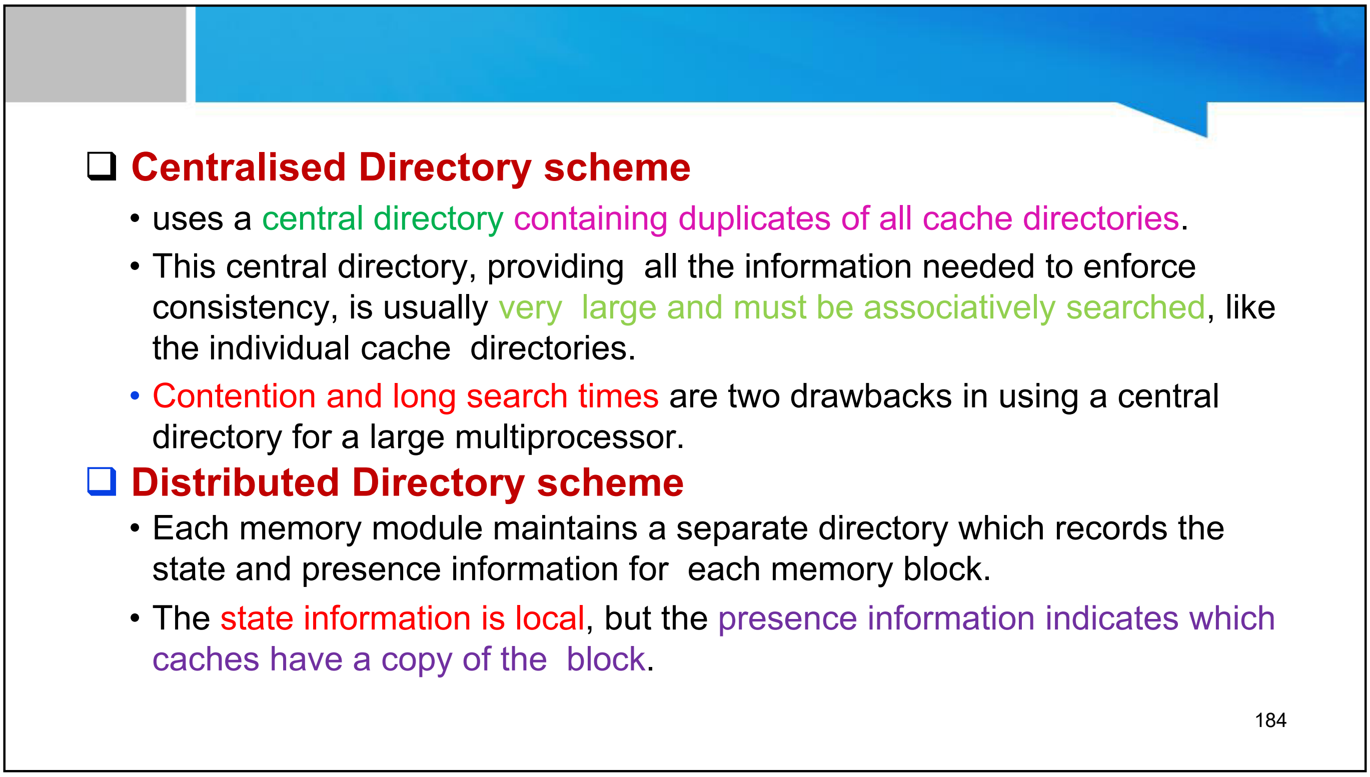
Directory-Based Protocols

- When a multistage or packet switched network is used to build a large multiprocessor with hundreds of processors, the snoopy cache protocols must be modified to suit the network capabilities.
- Since broadcasting is expensive to perform in such a network, consistency commands will be sent only to those caches that keep a copy of the block.
- This leads to directory-based protocols for network-connected multiprocessors.



Directory Structures

- In a multistage or packet switched network,cache coherence is supported by using cache directories to store information on where copies of cache blocks reside.
- Various directory-based protocols differ mainly in how directory maintains information and what information it stores.
  - **Centralised Directory scheme**
  - **Distributed Directory scheme**



❑ **Centralised Directory scheme**

- uses a central directory containing duplicates of all cache directories.
- This central directory, providing all the information needed to enforce consistency, is usually very large and must be associatively searched, like the individual cache directories.
- Contention and long search times are two drawbacks in using a central directory for a large multiprocessor.

❑ **Distributed Directory scheme**

- Each memory module maintains a separate directory which records the 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.

Directory-Based Cache coherence scheme

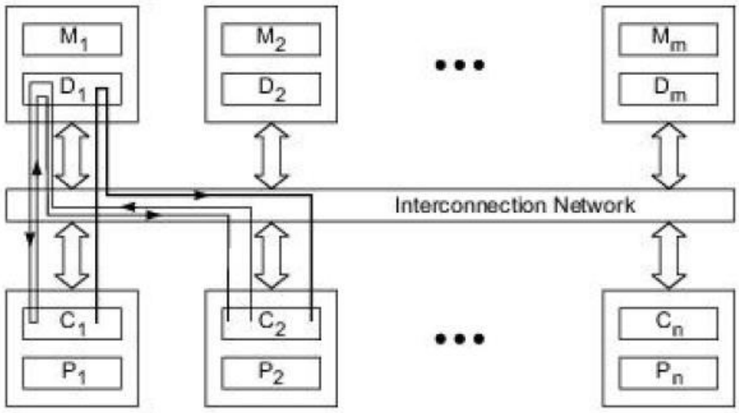


Fig. 7.17 Basic concept of a directory-based cache coherence scheme (Courtesy of Censier and Feautrier, *IEEE Trans. Computers*, Dec. 1978)

- A read-miss (thin lines) in cache2 results in a request sent to the memory module.
  - –The memory controller retransmits the request to the dirty copy in cache1.
  - –This cache1 writes back its copy.
  - –The memory module can supply a copy to the requesting cache.

•In the case of a write-hit at cache1 (bold lines),

- –a command is sent to the memory controller, which sends invalidations to all caches (cache2) marked in the presence vector residing in the directory D1 .

- A cache-coherence protocol that does not use broadcasts must store the locations of all cached copies of each block of shared data.
- This list of cached locations, whether centralized or distributed, is called a cache directory.
- 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.

Cache directory Protocols

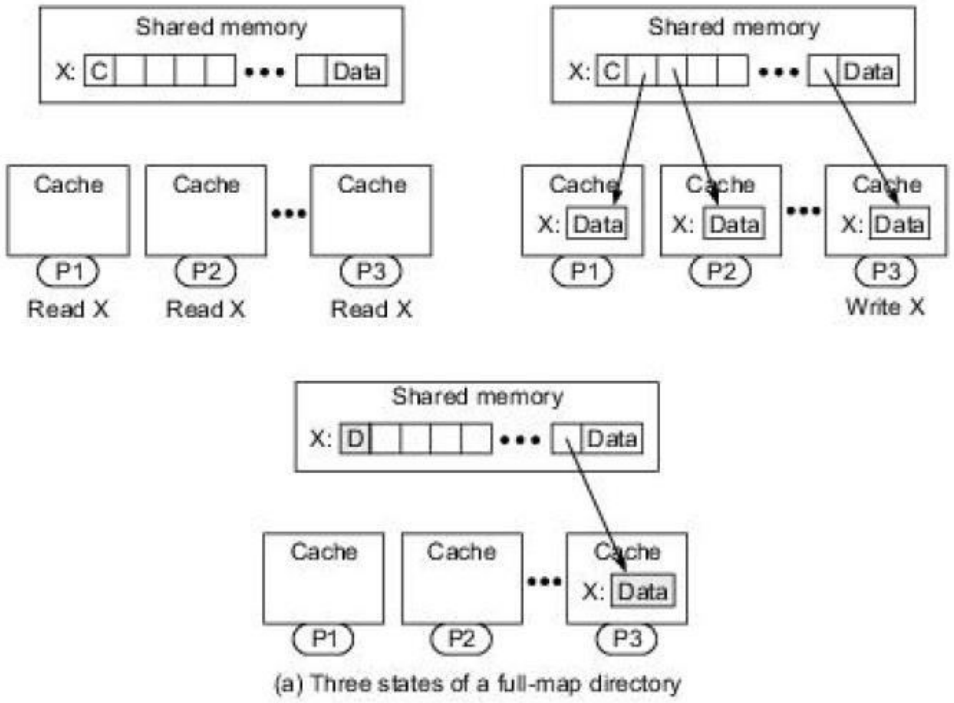
1. Full-Map Directories
   - store enough data associated with each block in global memory so that every cache in the system can simultaneously store a copy of any block of data
   - each directory entry contains N pointers, when: N is the number of processors in the system
2. Limited Directories
   - in that they have a fixed number of pointers per entry, regardless of thc system size
3. Chained Directories
   - emulate the full-map schemes by distributing the directory among the caches.

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.
- The full-map protocol implements directory entries with one bit per

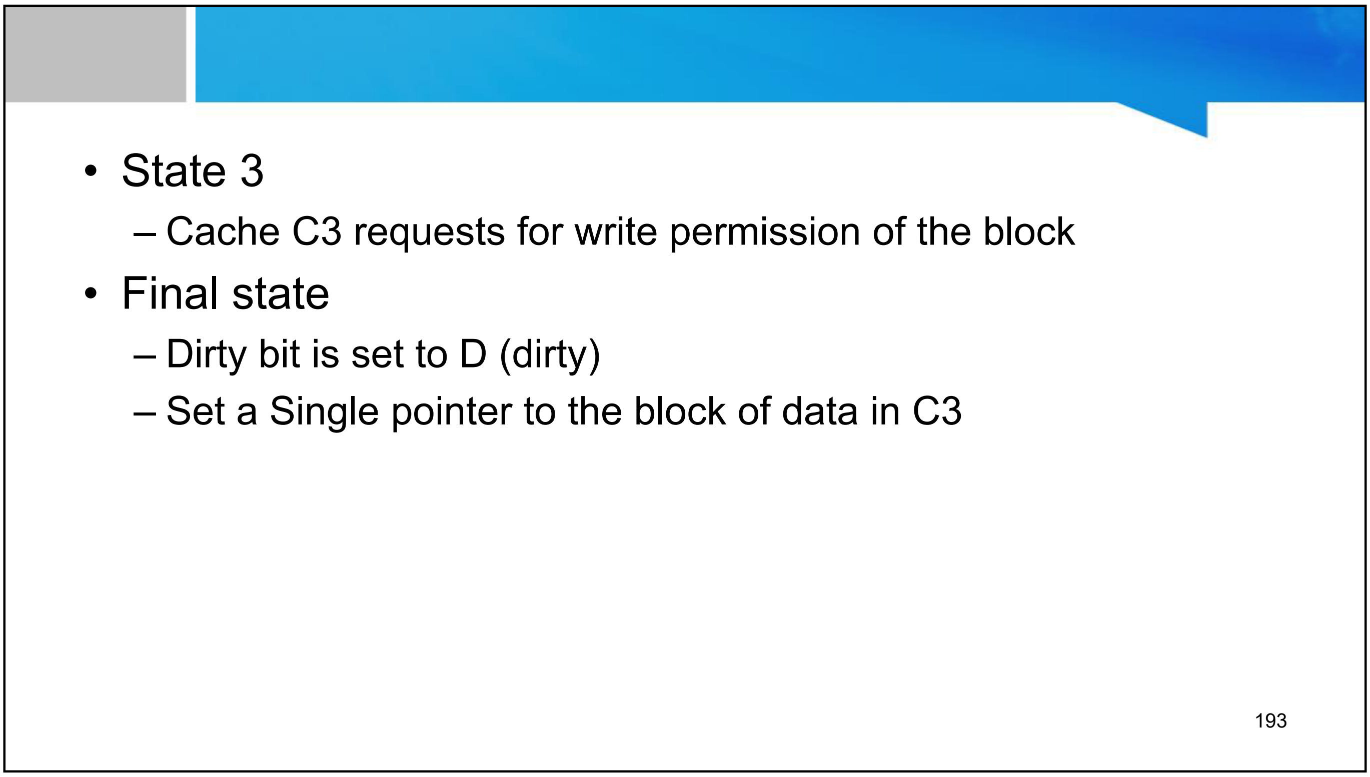
  processor and a dirty bit.

- Bit per processor specify whether the block is present or absent in a processor's cache
  - **Bit is set to 1→**block is present in processor's cache
  - **Bit is reset 0→**block is absent in processor's cache
- If dirty bit is set□only one processor's bit is set & it can perform write into the block
- Cache maintain two types of bits per block
  - One bit indicate whether the block is valid
  - Other bit indicate whether a valid block can be written
- The cache coherence protocol must keep the state bits in memory directory and those in the cache consistent.

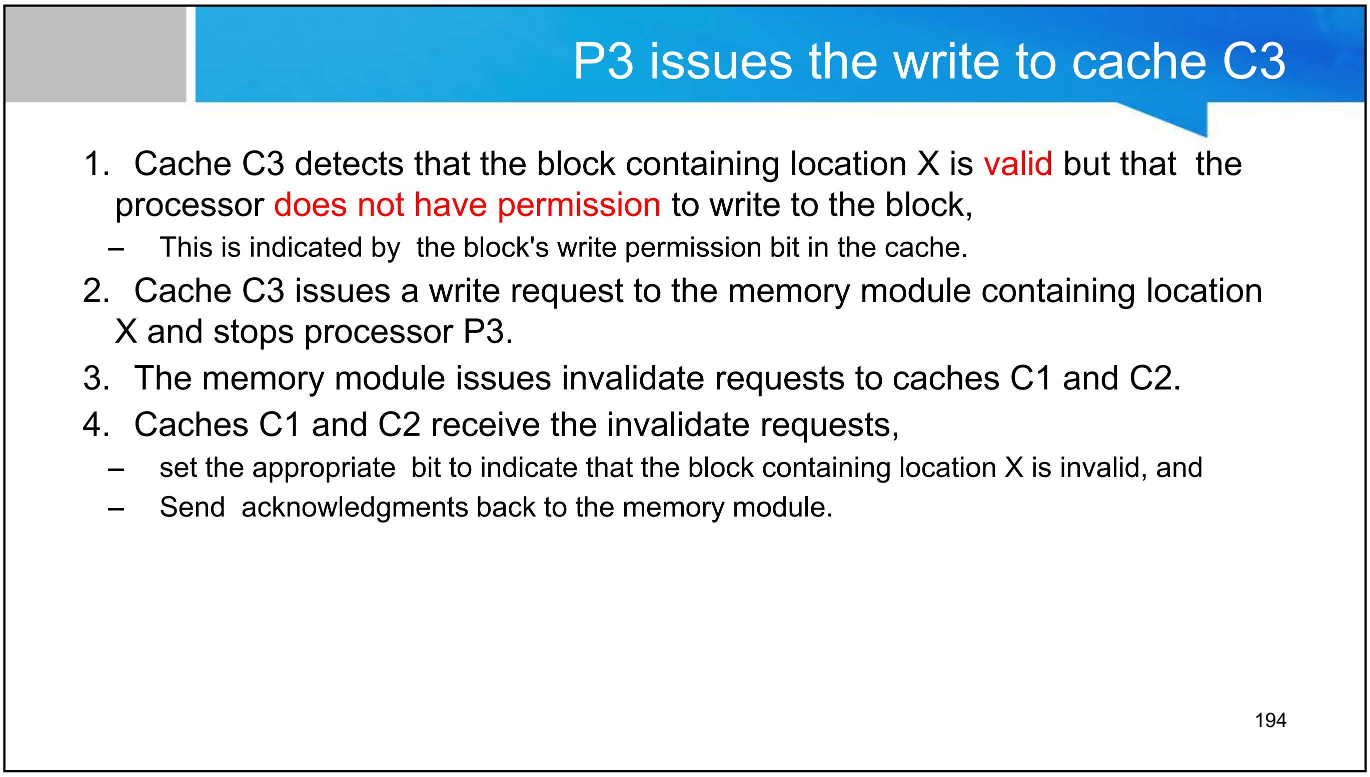


(a) Three states of a full-map directory

- State 1
  - Block X is missing in all the caches in the system
  - Dirty bit on the left of directory entry is set as clean
  - This indicates that no processor has permission to write to the block
- State 2
  - Caches C1, C2, and C3 requests for block X
  - In this situation, 3 pointers(processor bits) are set in the entry
  - This is to indicate the caches that have the copies of block of data
  - Dirty bit on the left of directory entry is set as clean

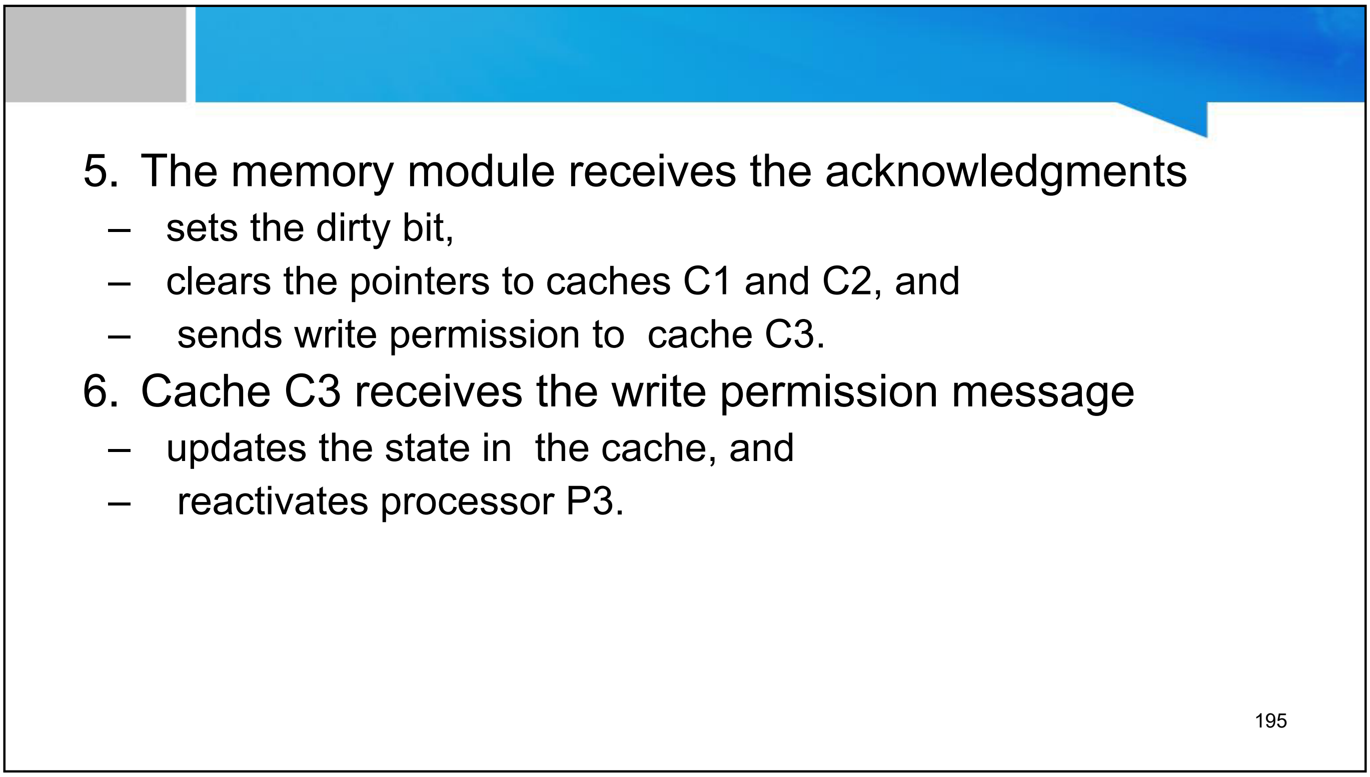


- State 3
  - Cache C3 requests for write permission of the block
- Final state
  - Dirty bit is set to D (dirty)
  - Set a Single pointer to the block of data in C3

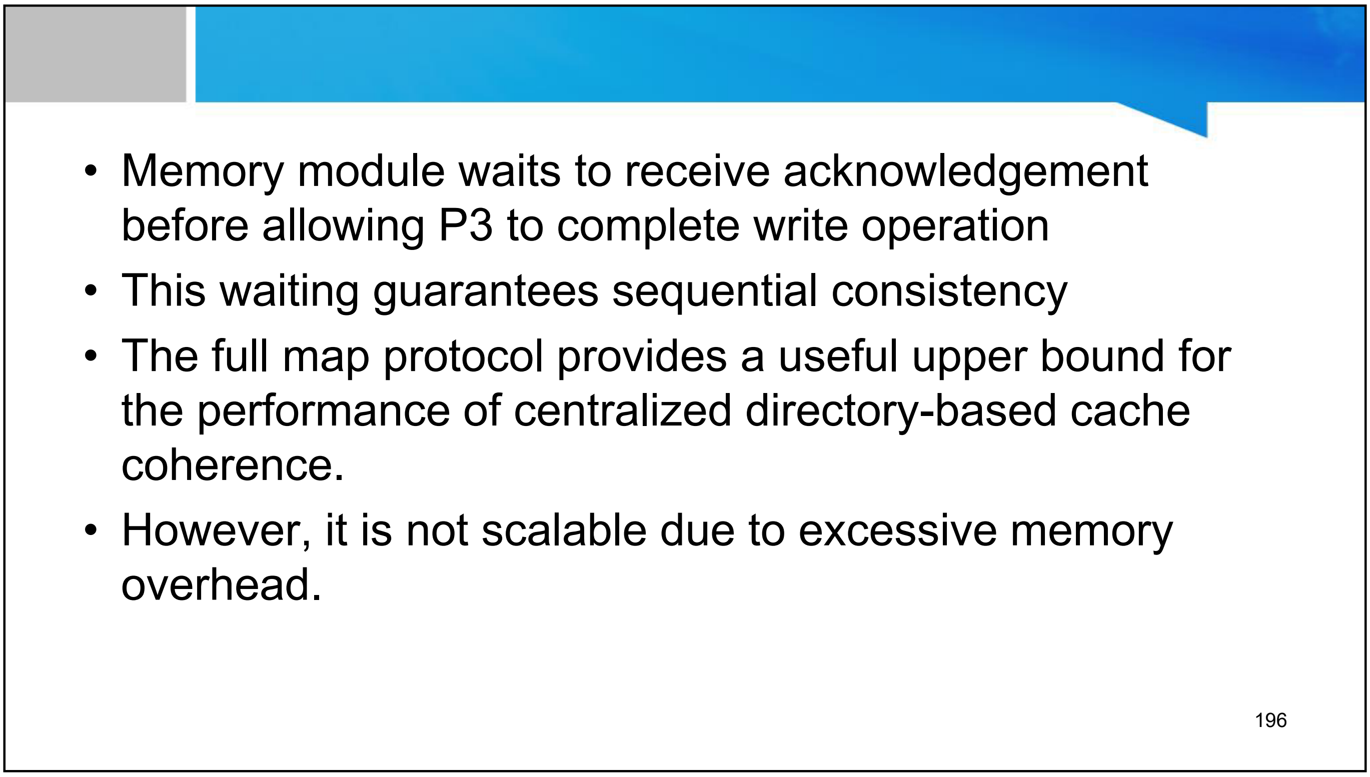


P3 issues the write to cache C3

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,
   - This is 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 stops processor P3.
3. The memory module issues invalidate requests to caches C1 and C2.
4. Caches C1 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 bit,
   - clears the pointers to caches C1 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.



- Memory module waits to receive acknowledgement before allowing P3 to complete write operation
- This waiting guarantees sequential consistency
- The full map protocol provides a useful upper bound for the performance of centralized directory-based cache coherence.
- However, it is not scalable due to excessive memory overhead.

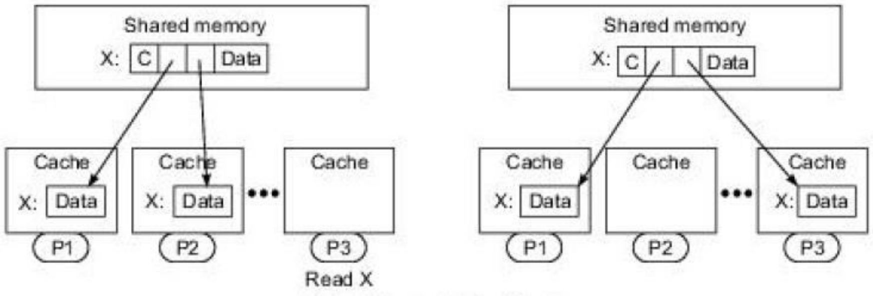
2. Limited Directories

- This protocol is designed to solve directory size problem
- There are only a fixed number of pointers for the directory entries, regardless of no: of processors
- Restriction to the no: of simultaneously cached copies limit the size of the directory

- It is similar to full directory protocol except the case when more than i caches request copies of a block of data
- Limited protocol that uses i< N is denotes as Dir$_i$NB
- i→no of Pointers
- NB→no broadcast

Example

- Suppose that protocol defines is Dir_2NB
  - i.e. only 2 pointers for the directory entries
- C1 and C2 has cached copies of block X
  - Pointers are set for C1 and C2
- Suppose C3 also requests for block X
  - Then memory module must invalidate the copy in either cache C1 or cache C2
- This process of pointer replacement is called eviction
- This protocol is scalable with respect to memory overhead
  - Less no: of pointers are needed



(b) Eviction in a limited directory

Drawback

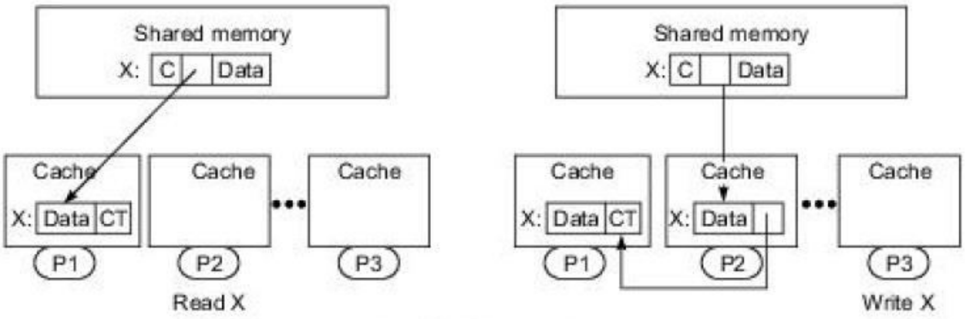
- Restricts the no: of shared copies of data blocks

2. Chained Directories

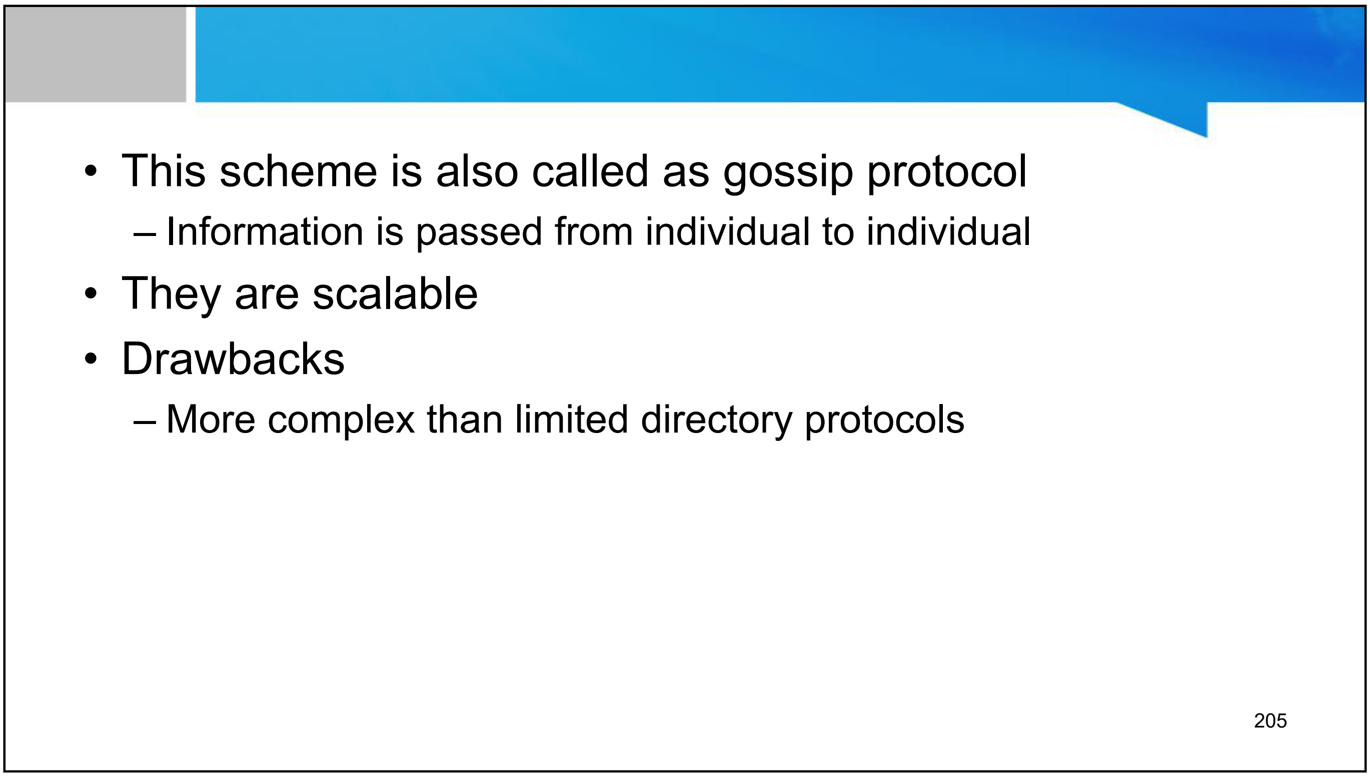
- This method provides scalability without restricting the no: of shared copies of data blocks
- It is called as chained scheme since it keeps track of shared copies of data by maintaining a chain of directory pointers

Example

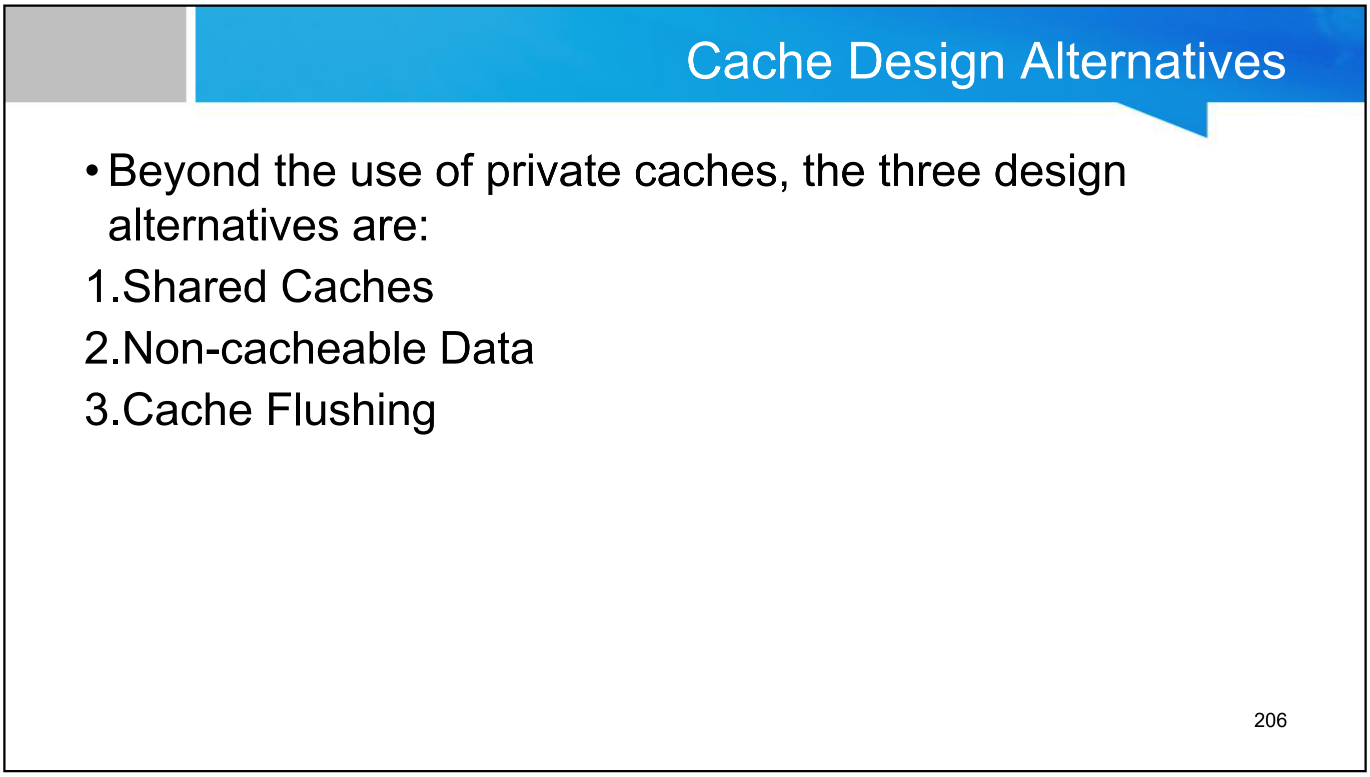
- Suppose there are no shared copies of X
- If P1 reads X, then memory sends the copy of X to C1 along with a chain termination pointer (CT)
- Memory also keeps a pointer to C1
- When P2 reads X, memory sends a copy to C2, along with a pointer to C1
- Memory keeps a pointer to C2
- If P3 writes to X, an invalidation message is sent down the chain
- To ensure sequential consistency,
  - memory denies write permission to P3 until it receives invalidation ack from the processor with CT pointer



(c) The chained directory



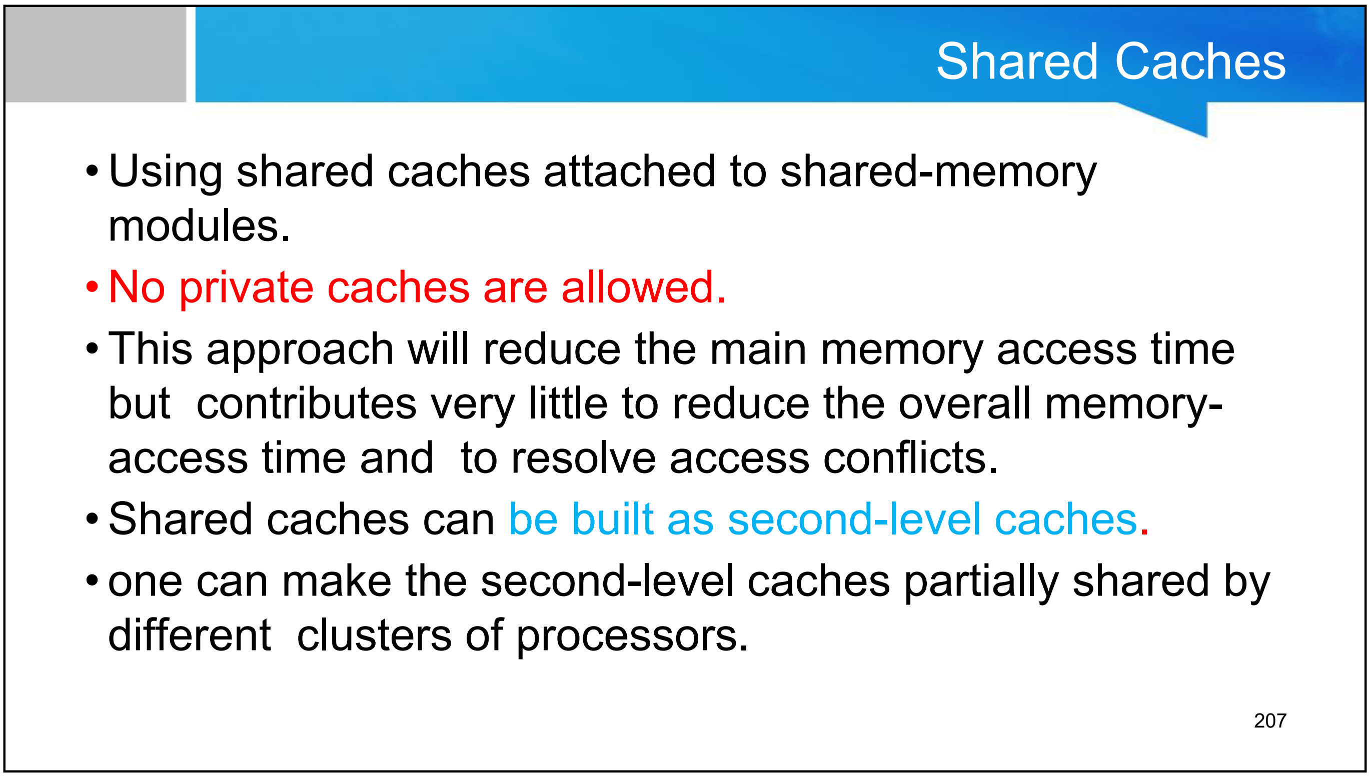
- This scheme is also called as gossip protocol
  - Information is passed from individual to individual
- They are scalable
- Drawbacks
  - More complex than limited directory protocols



Cache Design Alternatives

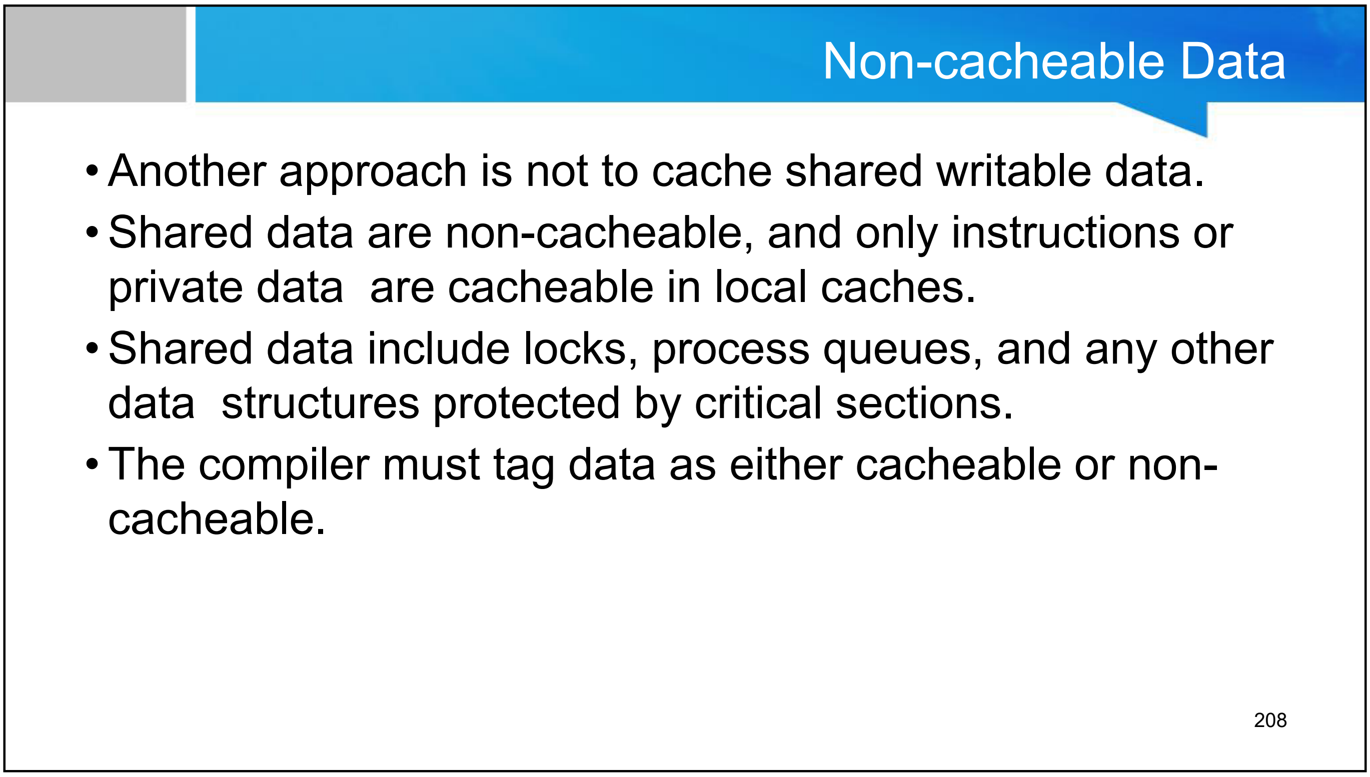
- Beyond the use of private caches, the three design alternatives are:

1. Shared Caches
2. Non-cacheable Data
3. Cache Flushing



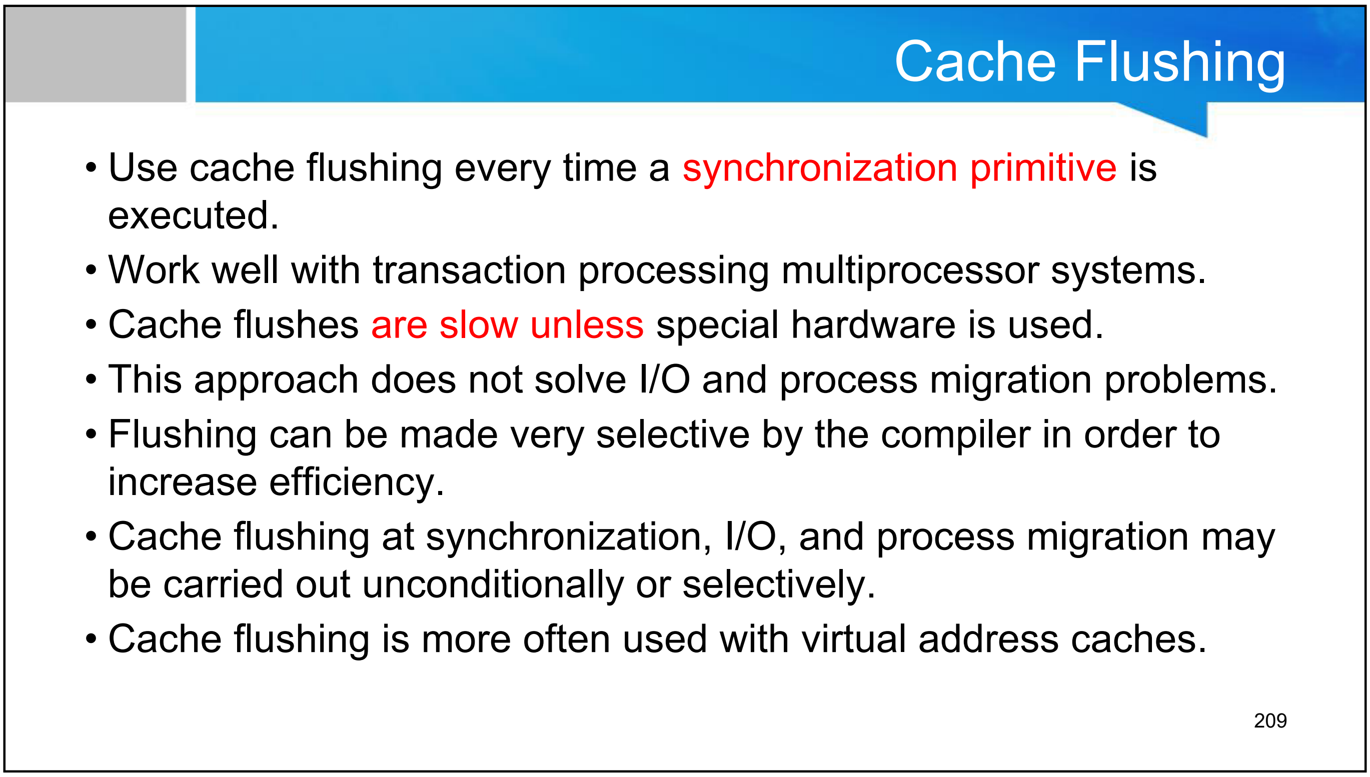
Shared Caches

- Using shared caches attached to shared-memory modules.
- No private caches are allowed.
- This approach will reduce the main memory access time but contributes very little to reduce the overall memory-access time and to resolve access conflicts.
- Shared caches can be built as second-level caches.
- one can make the second-level caches partially shared by different clusters of processors.



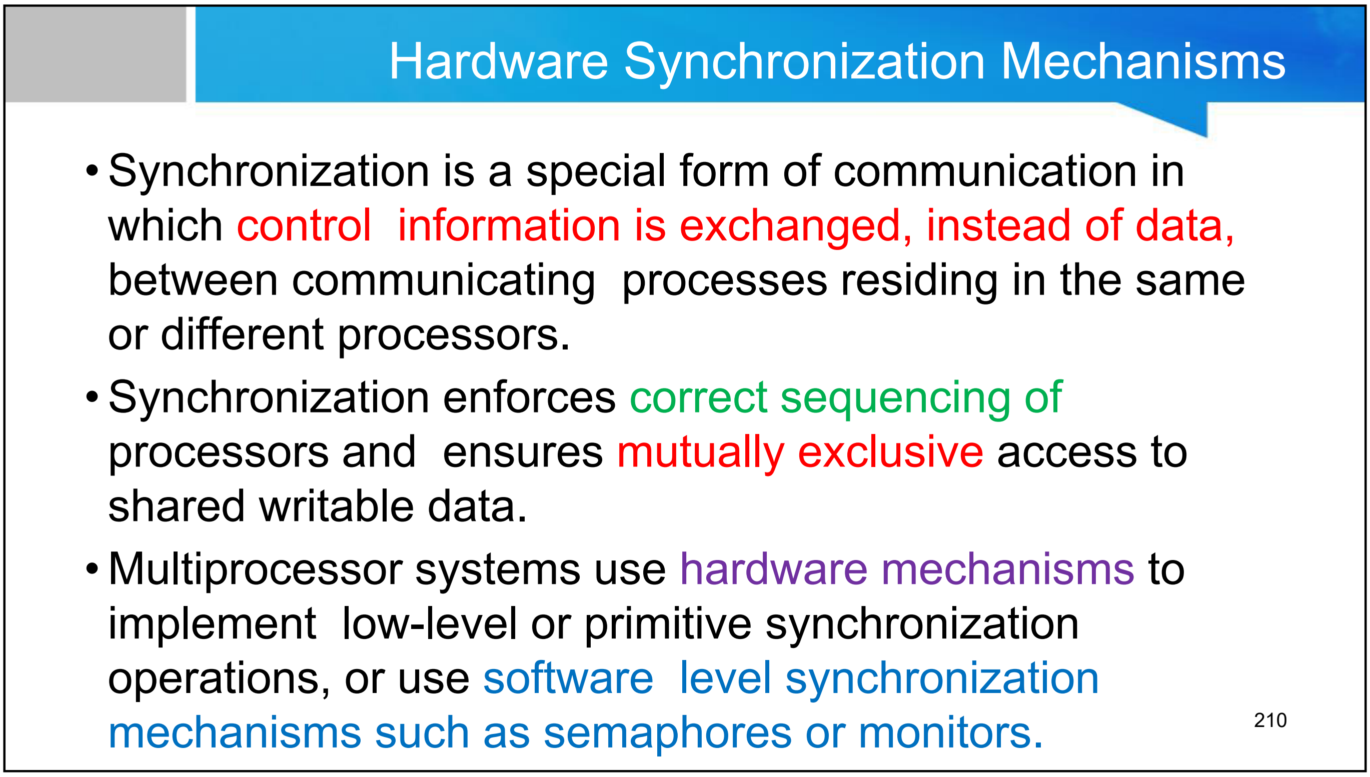
Non-cacheable Data

- 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.



Cache Flushing

- Use cache flushing every time a synchronization primitive is executed.
- Work well with transaction processing multiprocessor systems.
- Cache flushes are slow unless special hardware is used.
- This approach does not solve I/O and process migration problems.
- 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.
- Cache flushing is more often used with virtual address caches.

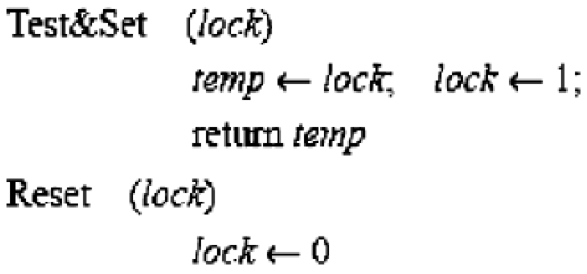


Hardware Synchronization Mechanisms

- Synchronization is a special form of communication in which control information is exchanged, instead of data, between communicating processes 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, or use software level synchronization mechanisms such as semaphores or monitors.

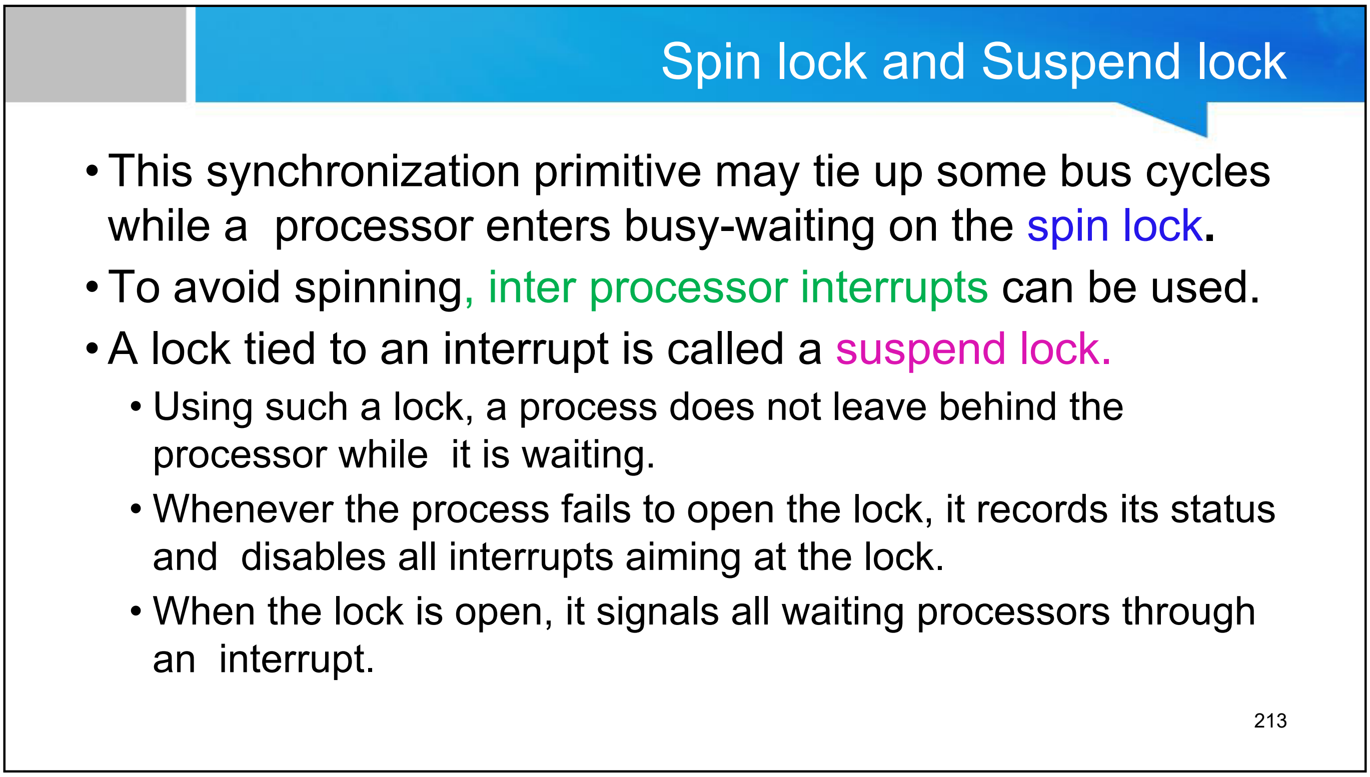
Atomic Operations

- Most multiprocessors are equipped with hardware mechanisms for enforcing atomic operations such as memory read, write, or read-modify-write operations.
- Besides atomic memory operations, some inter processor interrupts can be used for synchronization purposes.
- Eg: synchronization primitives, Test&Set (lock) and Reset (lock).



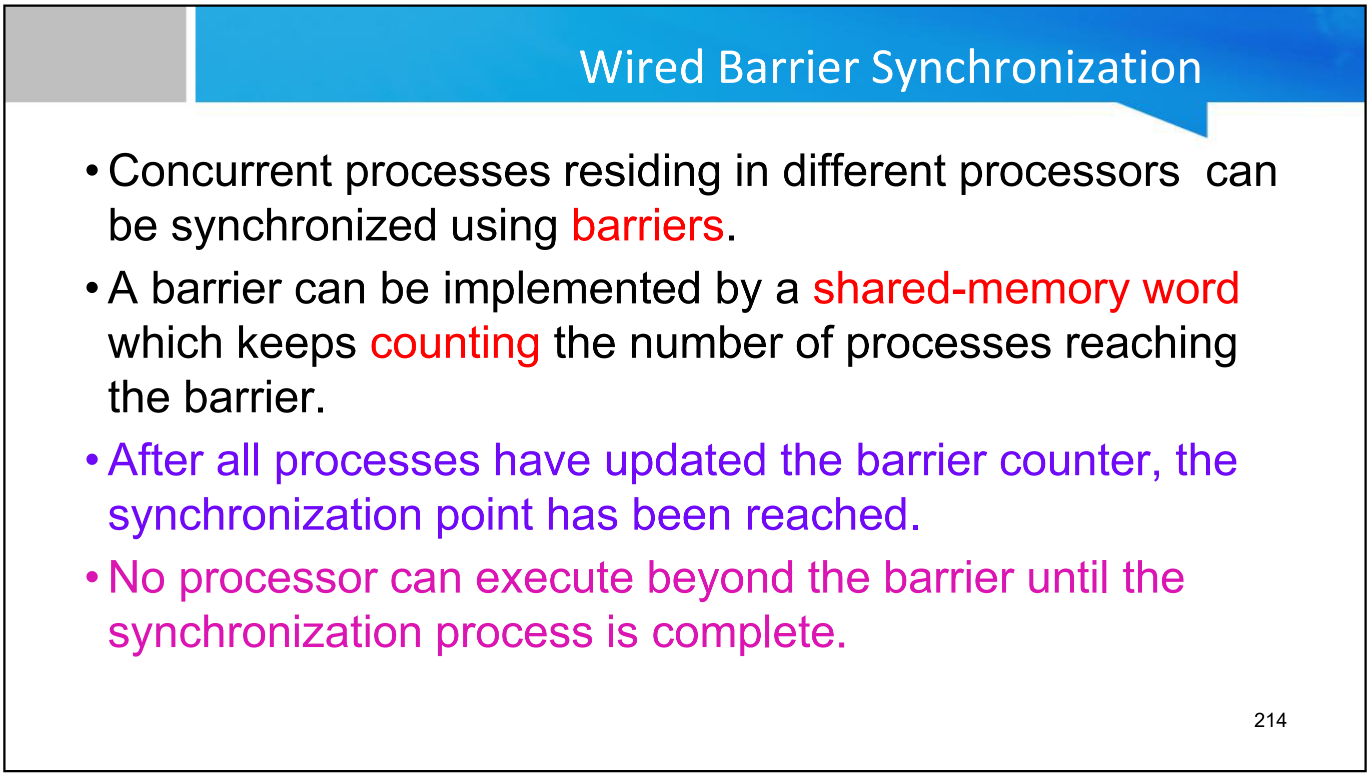
```
Test&Set   (lock)
           temp ← lock;   lock ← 1;
           return temp
Reset   (lock)
           lock ← 0
```

- Test&Set
  - implemented with atomic read-modify-write memory operations.
  - To synchronize concurrent processes, the software may repeat Test&Set until the returned value (temp) becomes 0.



Spin lock and Suspend lock

- This synchronization primitive may tie up some bus cycles while a processor enters busy-waiting on the spin lock.
- To avoid spinning, inter processor interrupts can be used.
- A lock tied to an interrupt is called a suspend lock.
  - Using such a lock, a process does not leave behind the processor while it is waiting.
  - Whenever the process fails to open the lock, it records its status and disables all interrupts aiming at the lock.
  - When the lock is open, it signals all waiting processors through an interrupt.

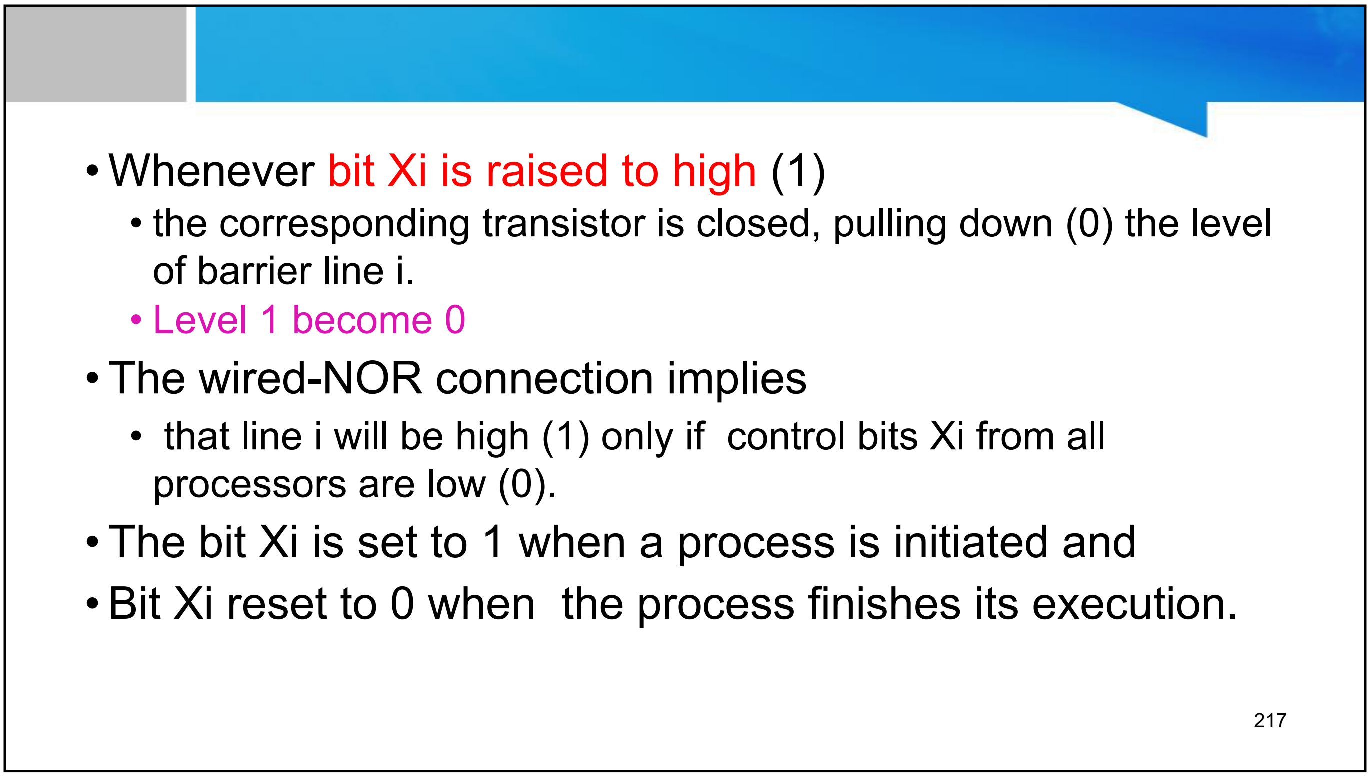


Wired Barrier Synchronization

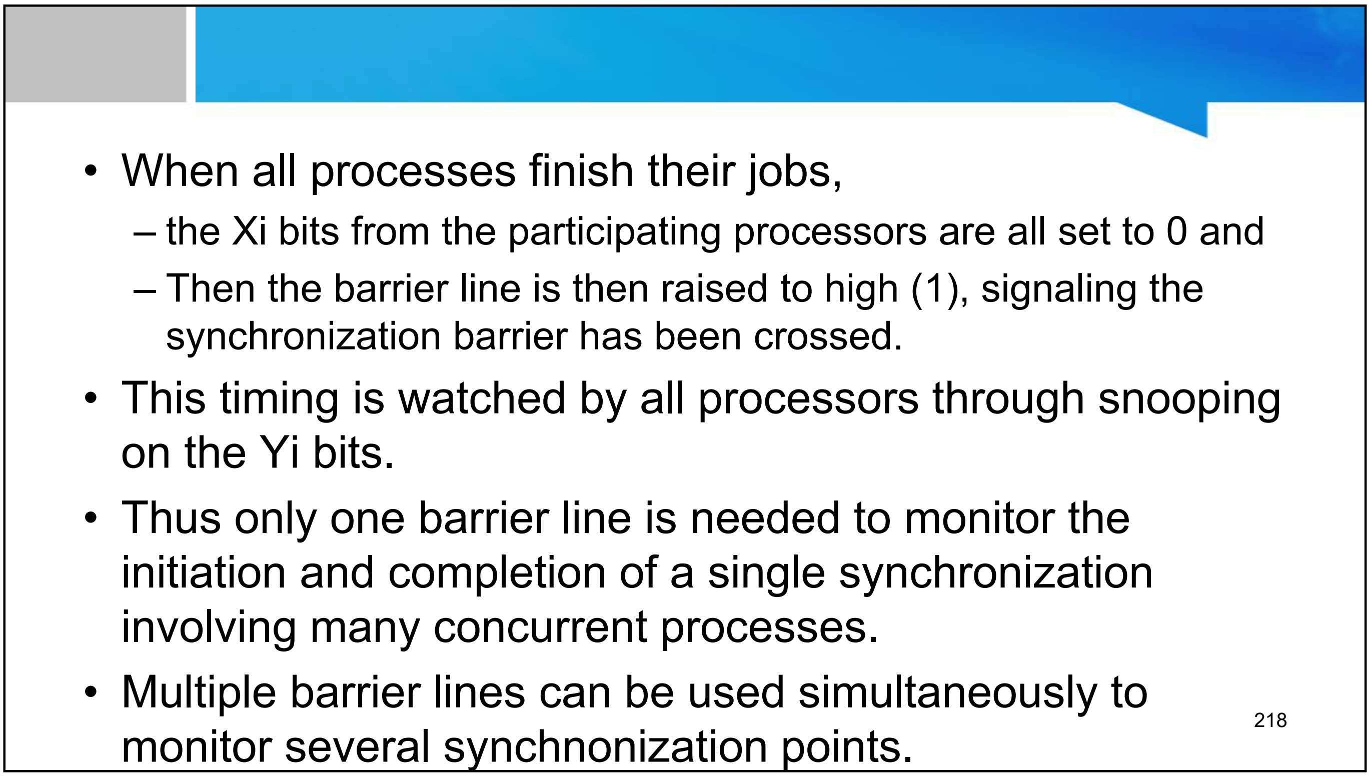
- Concurrent processes residing in different processors can be synchronized using barriers.
- A barrier can be implemented by a shared-memory word which keeps counting the number of processes reaching the barrier.
- After all processes have updated the barrier counter, the synchronization point has been reached.
- No processor can execute beyond the barrier until the synchronization process is complete.

- A wired-NOR logic implements a barrier mechanism for fast synchronization.
- Each processor uses
  - a dedicated control vector X = (X 1, X 2, . . . , Xm) and
  - accesses a common monitor vector Y = (Y1, Y2, ... , Ym) in shared memory,
  - where m corresponds to the barrier lines used.
- The number of barrier lines needed for synchronization depends on the multiprogramming degree and the size of the multiprocessor system.

- Each control bit Xi is connected to the base (input) of a probing transistor.
- The monitor bit Yi checks the collector voltage (output) of transistor.
- Each barrier line is wired-NOR to n transistors from n processors.

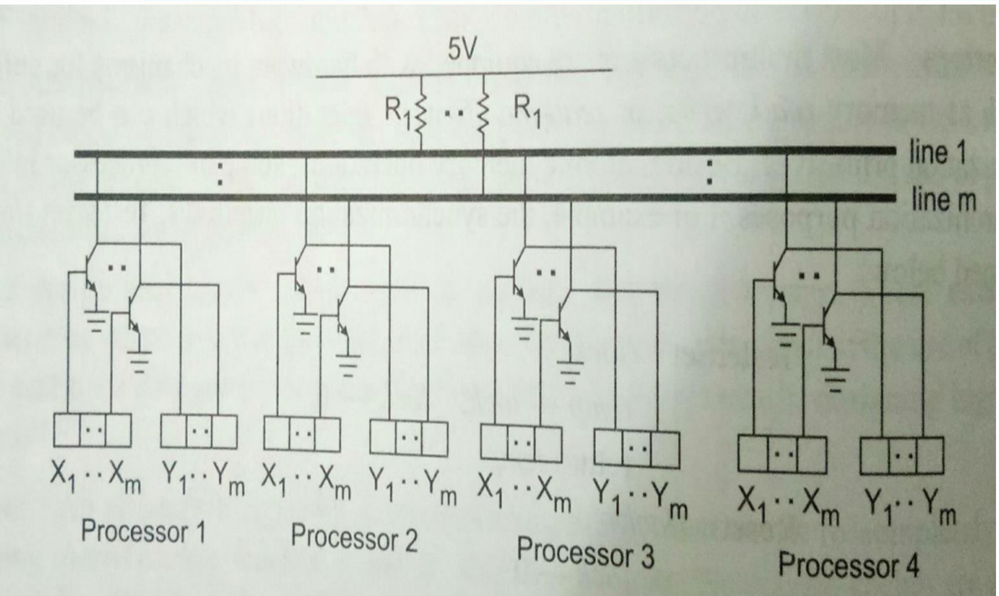


- Whenever bit Xi is raised to high (1)
  - the corresponding transistor is closed, pulling down (0) the level of barrier line i.
  - Level 1 become 0
- The wired-NOR connection implies
  - that line i will be high (1) only if control bits Xi from all processors are low (0).
- The bit Xi is set to 1 when a process is initiated and
- Bit Xi reset to 0 when the process finishes its execution.



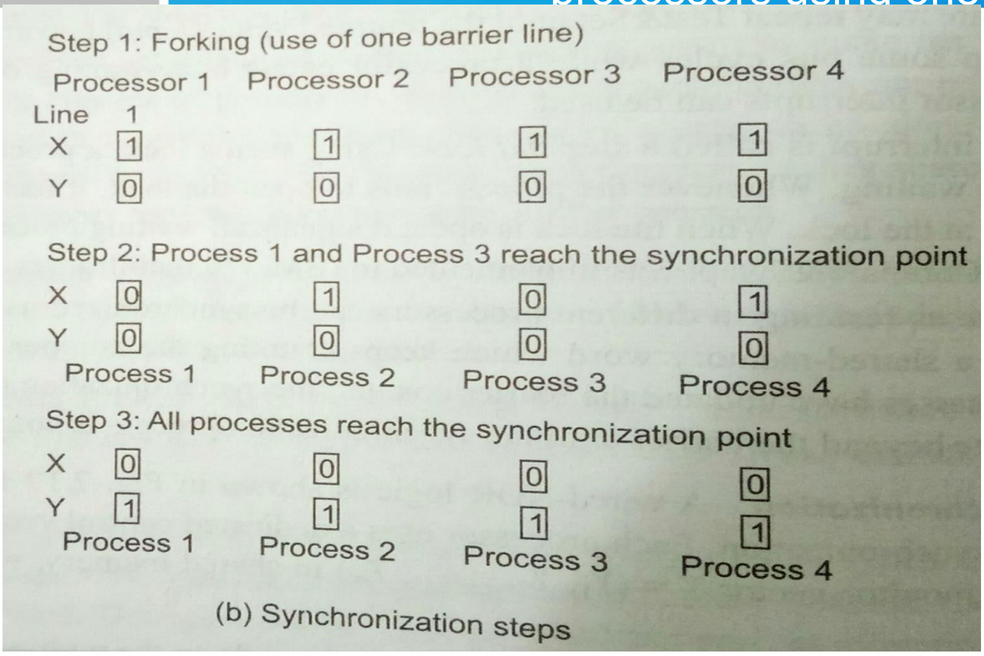
- When all processes finish their jobs,
  - the Xi bits from the participating processors are all set to 0 and
  - Then the barrier line is then raised to high (1), signaling the synchronization barrier has been crossed.
- This timing is watched by all processors through snooping on the Yi bits.
- Thus only one barrier line is needed to monitor the initiation and completion of a single synchronization involving many concurrent processes.
- Multiple barrier lines can be used simultaneously to monitor several synchnonization points.

A wired-NOR logic for barrier mechanism



(a) Barrier lines and interface logic

Synchronization of four processes residing on four processors using one barrier line.



Step 1: Forking (use of one barrier line)

	Processor 1	Processor 2	Processor 3	Processor 4
Line 1				
X	1	1	1	1
Y	0	0	0	0

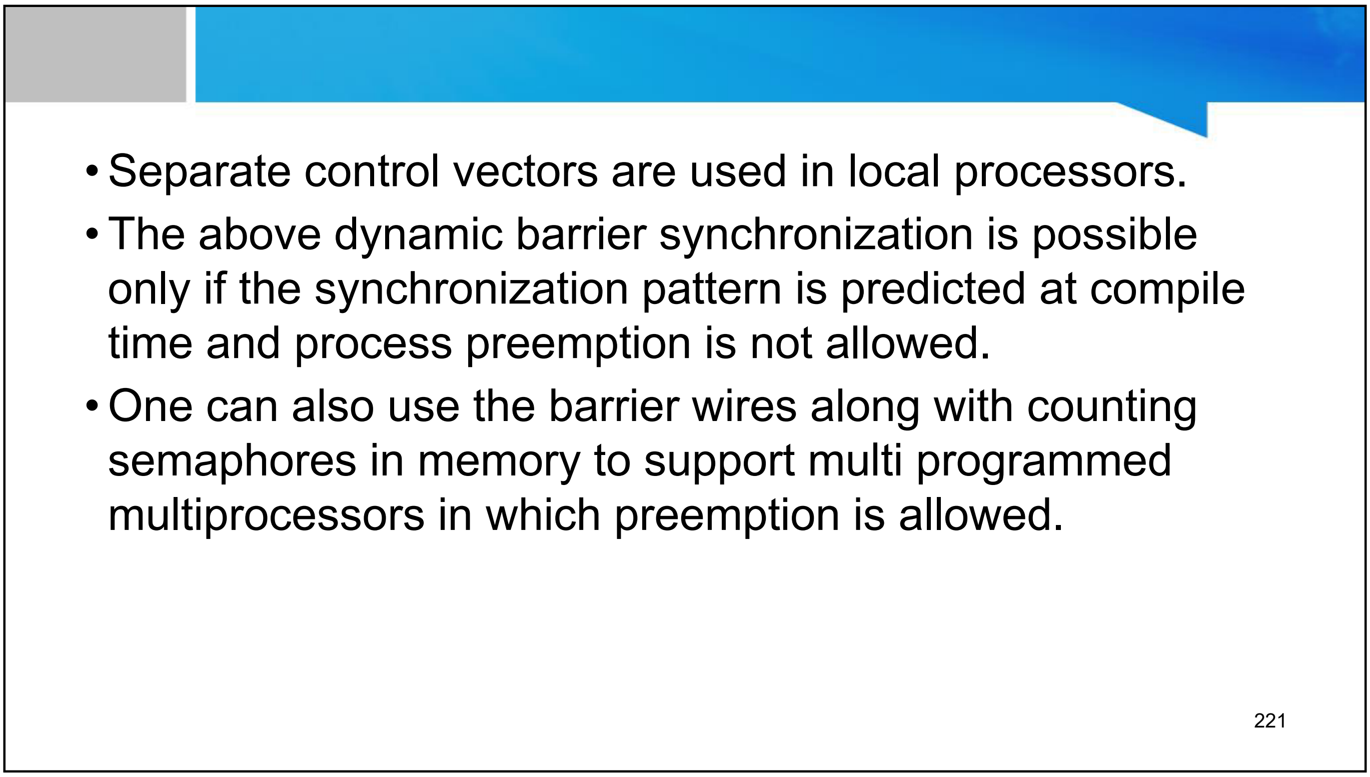
Step 2: Process 1 and Process 3 reach the synchronization point

X	0	1	0	1
Y	0	0	0	0
	Process 1	Process 2	Process 3	Process 4

Step 3: All processes reach the synchronization point

X	0	0	0	0
Y	1	1	1	1
	Process 1	Process 2	Process 3	Process 4

(b) Synchronization steps



- Separate control vectors are used in local processors.
- The above dynamic barrier synchronization is possible only if the synchronization pattern is predicted at compile time and process preemption is not allowed.
- One can also use the barrier wires along with counting semaphores in memory to support multi programmed multiprocessors in which preemption is allowed.