Advanced Computer Architecture

HWANG, CHAPTER 7

MULTIPROCESSORS AND MULTICOMPUTERS

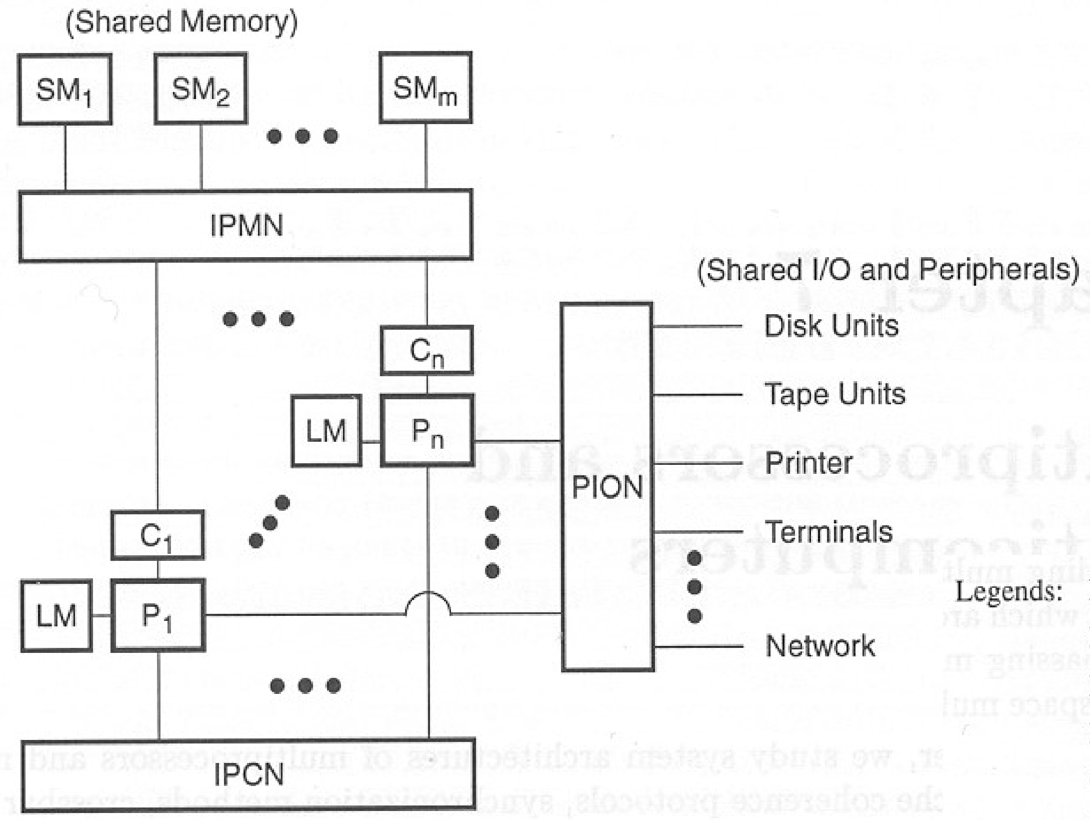
7.1 MULTIPROCESSOR SYSTEM INTERCONNECTS

Multiprocessors Vs Multicomputers

The **main difference** between multiprocessor and multicomputer is that the **multiprocessor is a system with two or more CPUs that is capable of performing multiple tasks at the same time while a multicomputer is a system with multiple processors that are connected via an interconnection network to perform a computation task.**

Parallel processing demands the use of efficient system interconnects for fast communication among multiple processors and shared memory, I/O and peripheral devices.

Generalized Multiprocessor System



Legends:		
	IPMN	(Inter-Processor-Memory Network)
	PION	(Processor-I/O Network)
	IPCN	(Inter-Processor Communication Network)
	P	(Processor)
	C	(Cache)
	SM	(Shared Memory)
	LM	(Local Memory)

Generalized Multiprocessor System

Each processor Pi is attached to its own local memory and private cache.

Multiple processors connected to shared memory through interprocessor memory network (IPMN).

Processors share access to I/O and peripherals through processor-I/O network (PION).

Both IPMN and PION are necessary in a shared-resource multiprocessor.

An optional interprocessor communication network (IPCN) can permit processor communication without using shared memory.

Interconnection Network Characteristics

Timing Protocol

- Synchronous – controlled by a global clock
- Asynchronous – use handshaking or interlock mechanisms

Switching Method

- Circuit switching – a pair of communicating devices control the path for the entire duration of data transfer
- Packet switching – large data transfers broken into smaller pieces, each of which can compete for use of the path

Network Control

- Centralized – global controller receives and acts on requests
- Distributed – requests handled by local devices independently

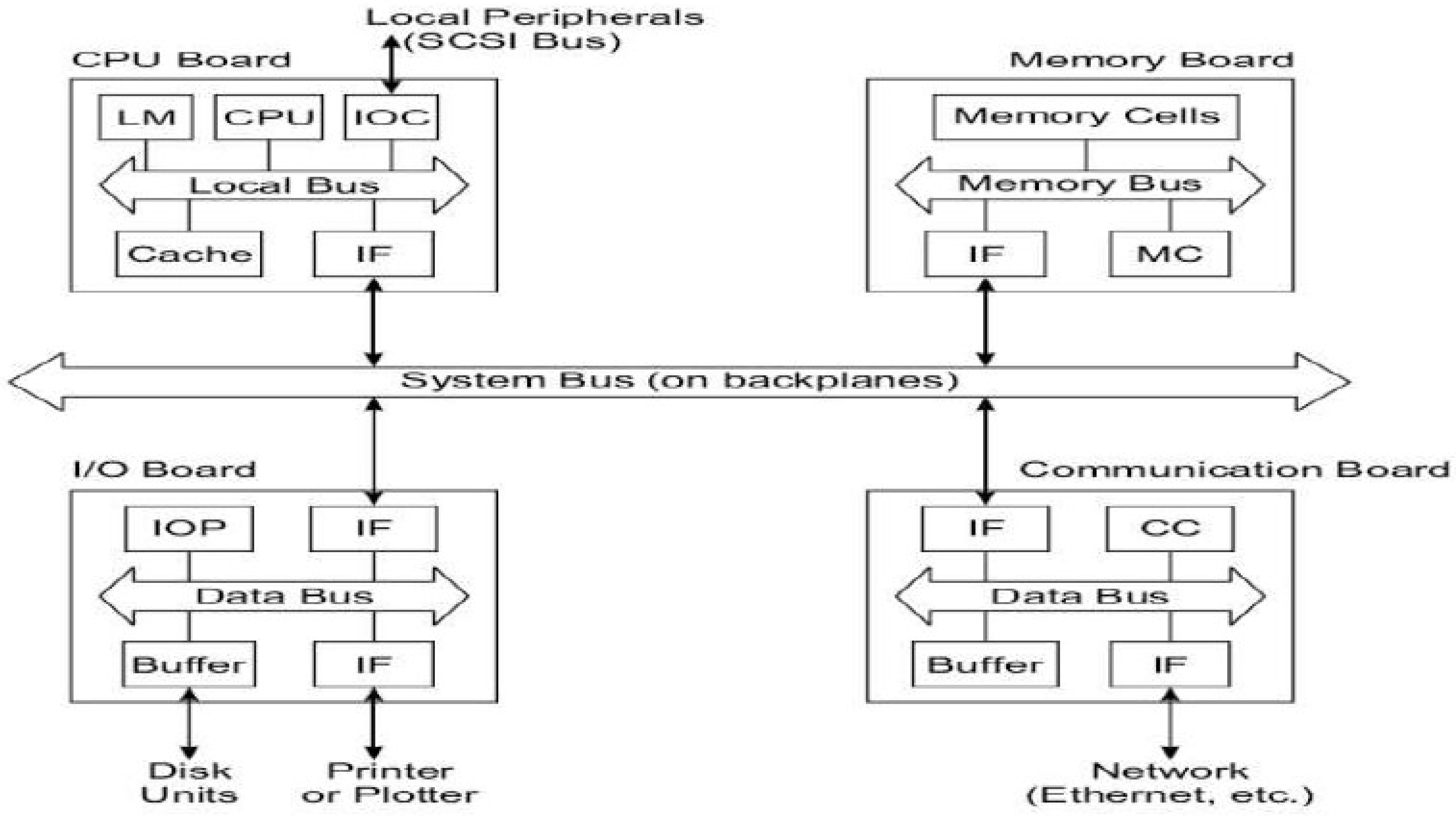
7.1.1 Hierarchical Bus Systems

A bus system is a hierarchy of buses connection various system and subsystem components.

Each bus has a complement of control, signal, and power lines.

There is usually a variety of buses in a system:

- Local bus – (usually integral to a system board) connects various major system components (chips)
- Memory bus – used within a memory board to connect the interface, the controller, and the memory cells
- Data bus – might be used on an I/O board or VLSI chip to connect various components
- Backplane – like a local bus, but with connectors to which other boards can be attached



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

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

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.

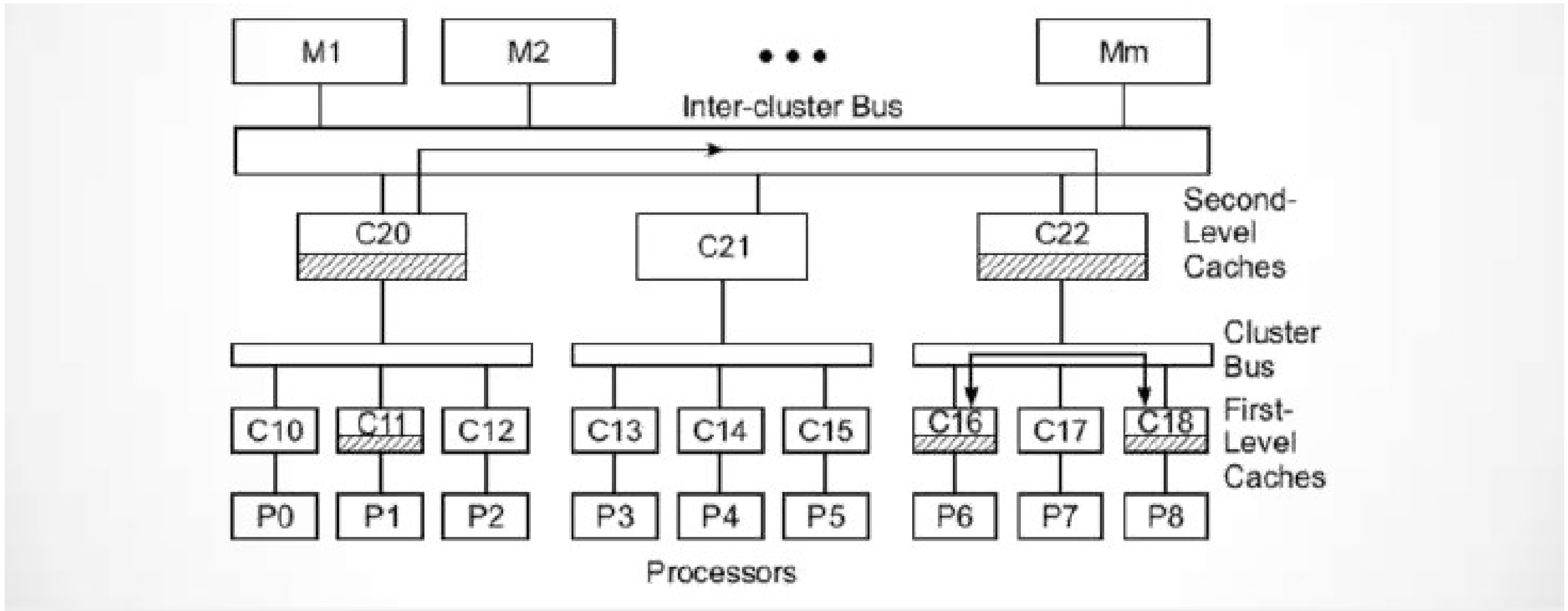


Fig. 7.3 A hierarchical cache/bus architecture for designing a scalable multiprocessor (Courtesy of Wilson; reprinted from *Proc. of Annual Int. Symp. on Computer Architecture, 1987*)

Bridges

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
- Serving as cache and memory agents

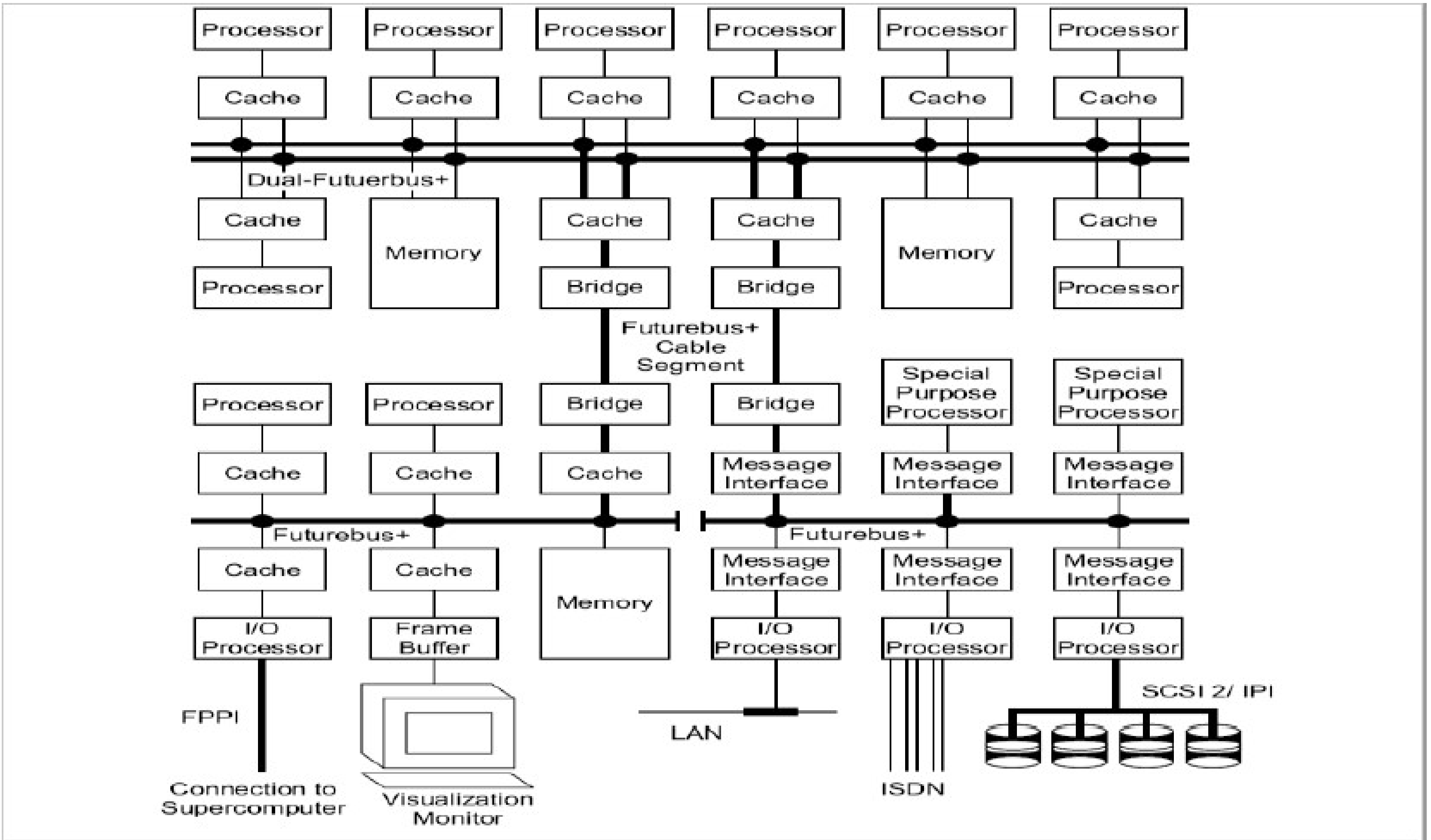
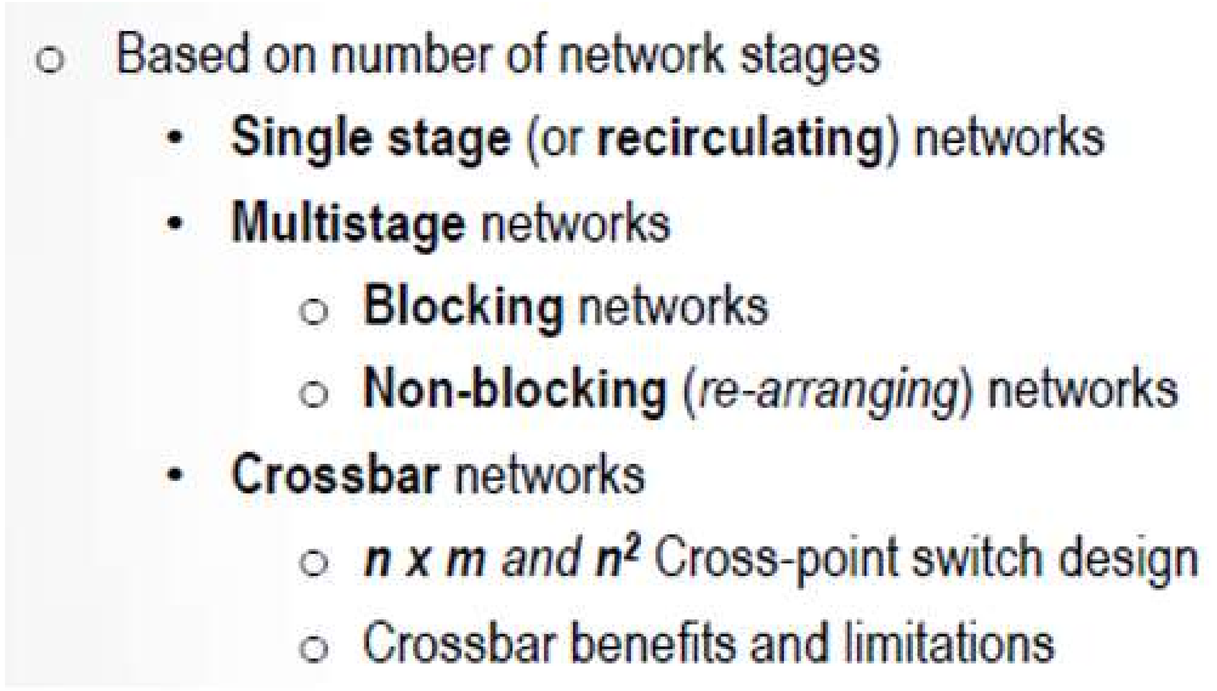


Fig. 7.5 A multiprocessor system using multiple Futurebus+ segments (Reprinted with permission from IEEE Standard 896.1-1991, copyright © 1991 by IEEE, Inc.)

7.1.2 Crossbar Switch and Multiport Memory



- Based on number of network stages
  - **Single stage** (or **recirculating**) networks
  - **Multistage** networks
    - **Blocking** networks
    - **Non-blocking** (*re-arranging*) networks
  - **Crossbar** networks
    - ***n x m** and **n^2*** Cross-point switch design
    - Crossbar benefits and limitations

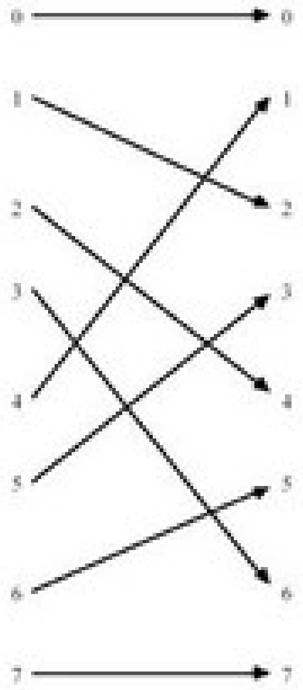
Network stages

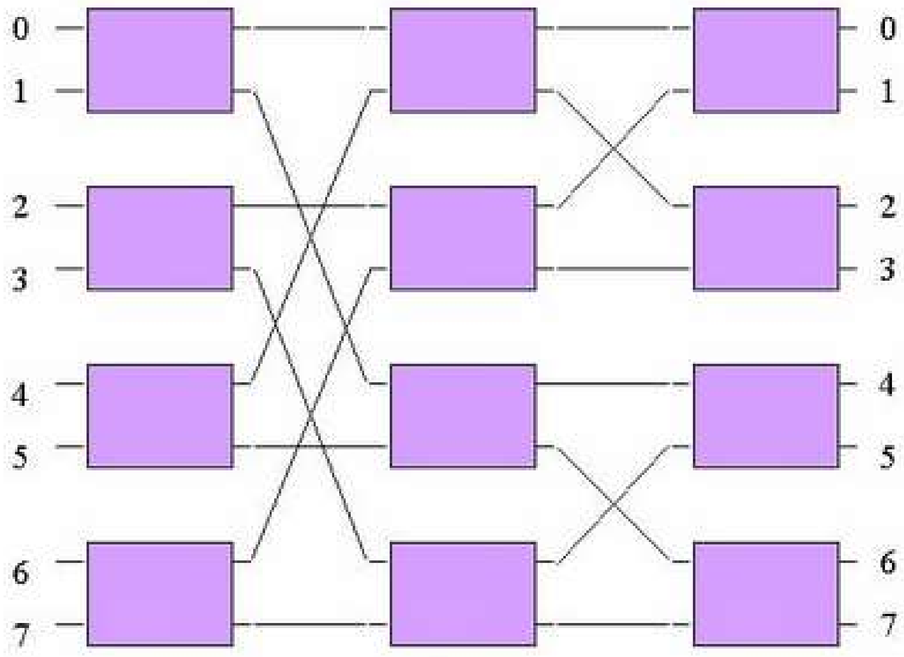
Single stage networks are sometimes called *recirculating networks* because data items may have to pass through the single stage many times.

- The crossbar switch and the multiported memory organization (seen later) are both single-stage networks.

This is because even if two processors attempted to access the same memory module (or I/O device at the same time, only one of the requests is serviced at a time.

Multistage networks consist of multiple stages of switch boxes, and should be able to connect any input to any output.

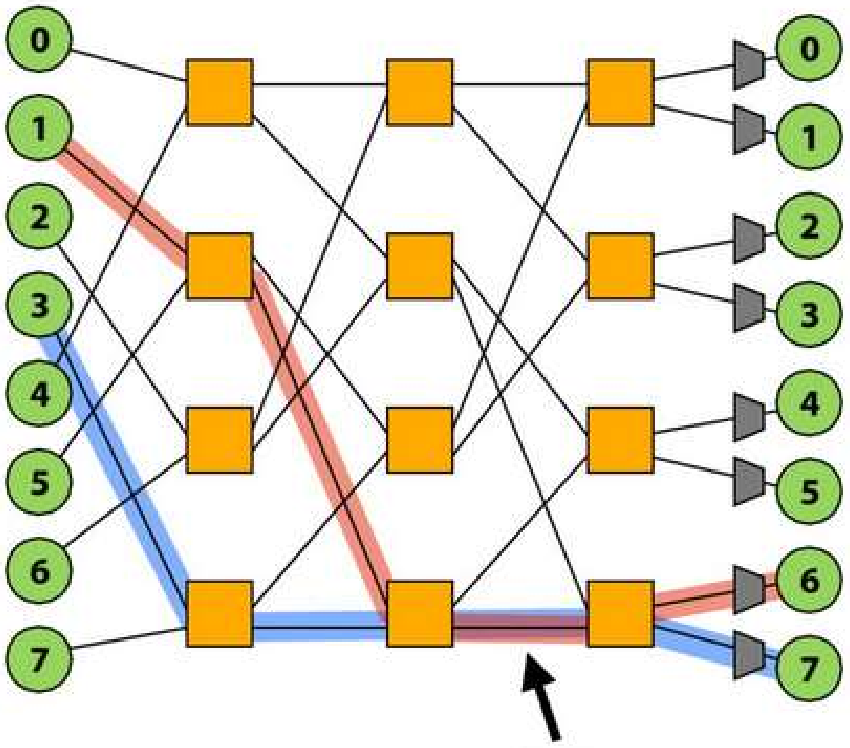

0
1
2
3
4
5
6
7
0
1
2
3
4
5
6
7


0
1
2
3
4
5
6
7
0
1
2
3
4
5
6
7

Blocking vs nonblocking 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.

A nonblocking multistage network can perform all possible connections between inputs and outputs by rearranging its connections.


0
1
2
3
4
5
6
7
0
1
2
3
4
5
6
7

Crossbar Networks

Crossbar networks connect every input to every output through a crosspoint switch.

A crossbar network is a single stage, non-blocking permutation network.

In an n-processor, m-memory system, $n \times m$ 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.

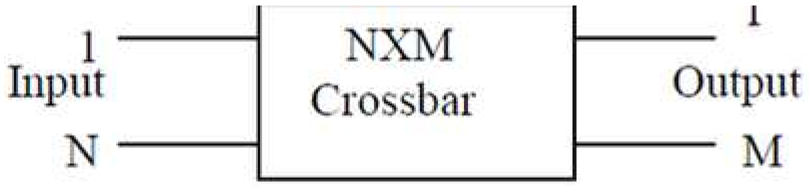
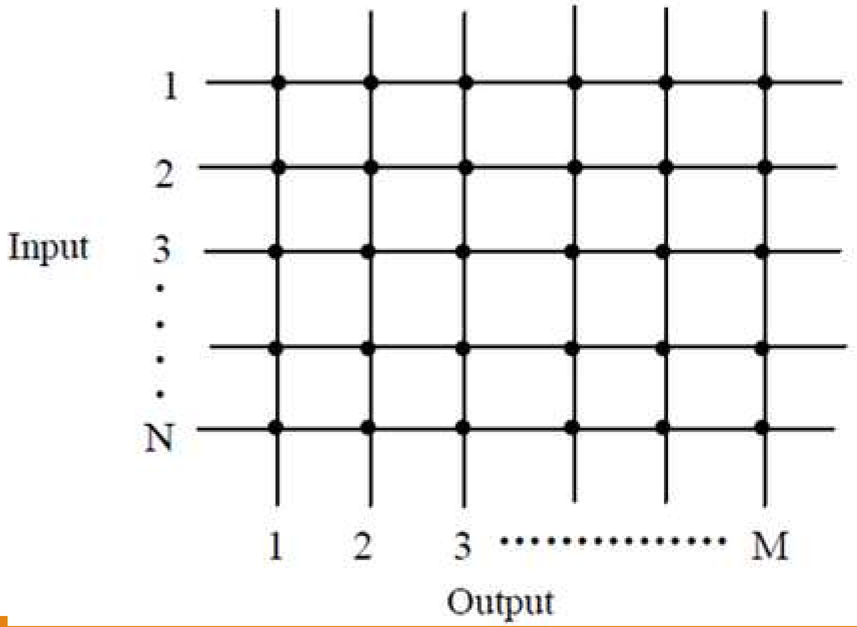
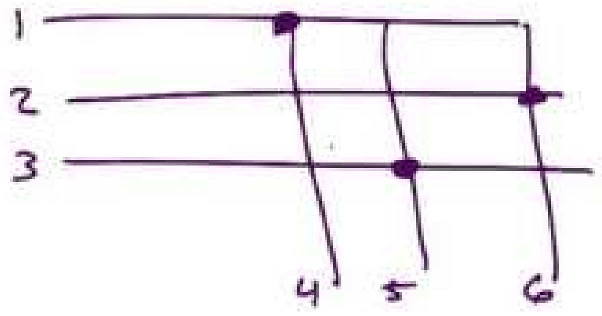


Figure: 3(a)



Switches



Crosspoint Switch Design

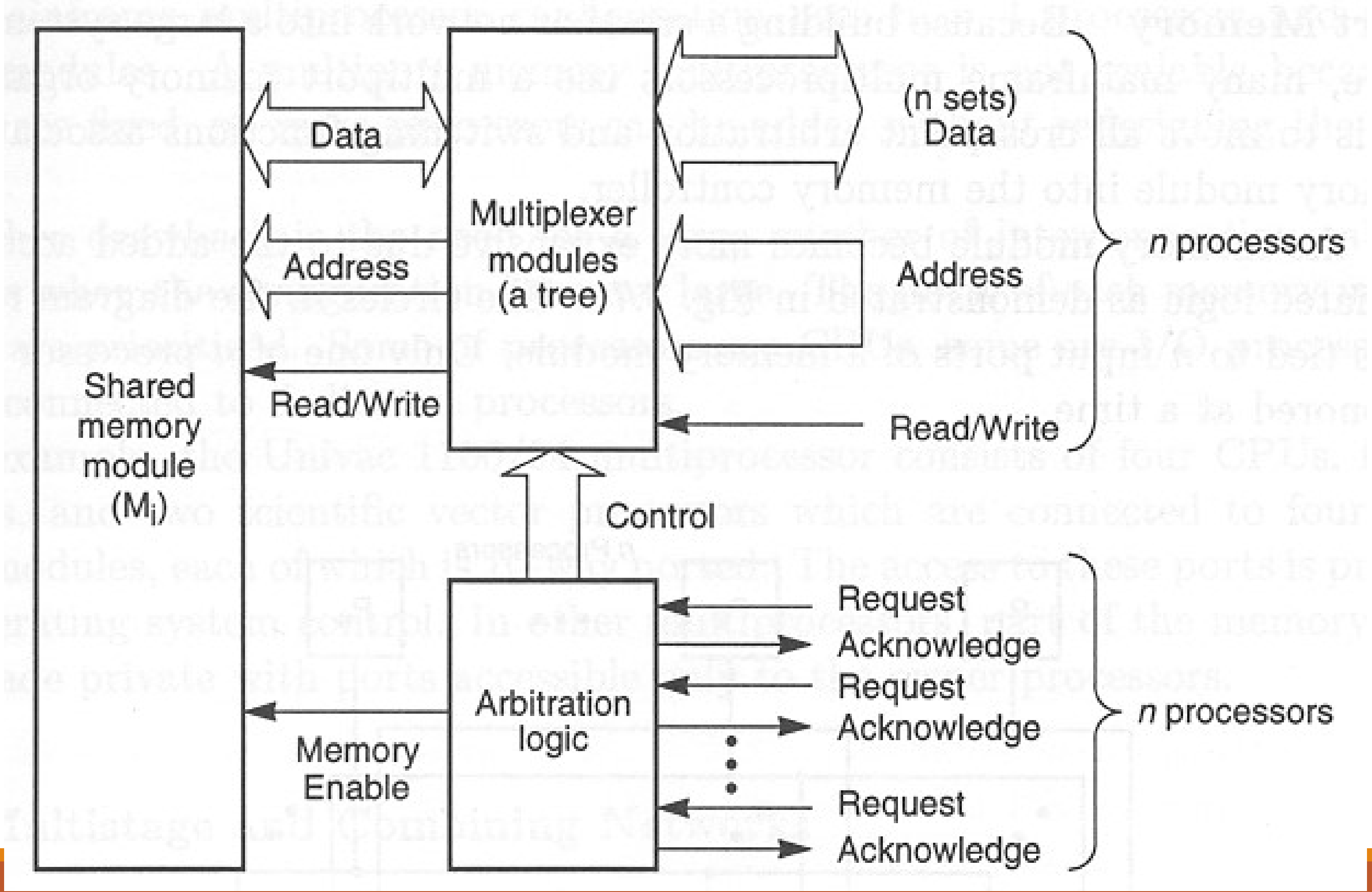
Out of n crosspoint switches in each column of an $n \times m$ crossbar mesh, only one can be connected at a time.

Crosspoint switches must be designed to handle the potential contention for each memory module.

Each processor provides a request line, a read/write line, a set of address lines, and a set of data lines to a crosspoint switch for a single column.

The crosspoint switch eventually responds with an acknowledgement when the access has been completed.

Schematic of a Crosspoint Switch

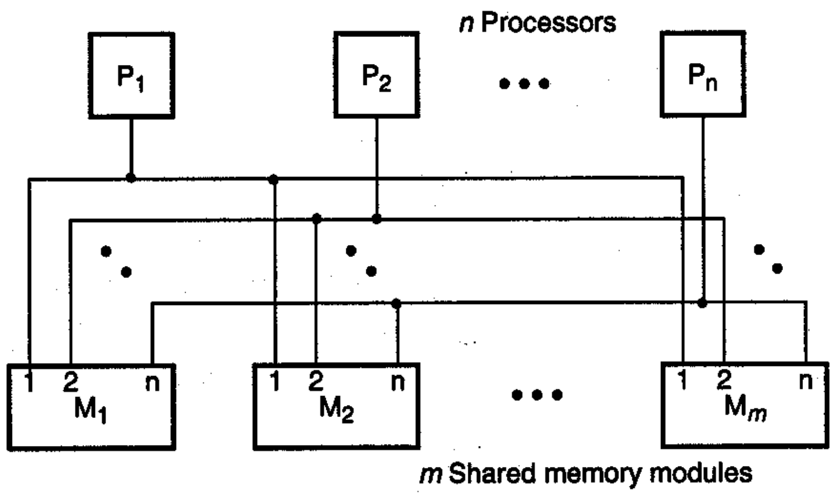


Multiport Memory

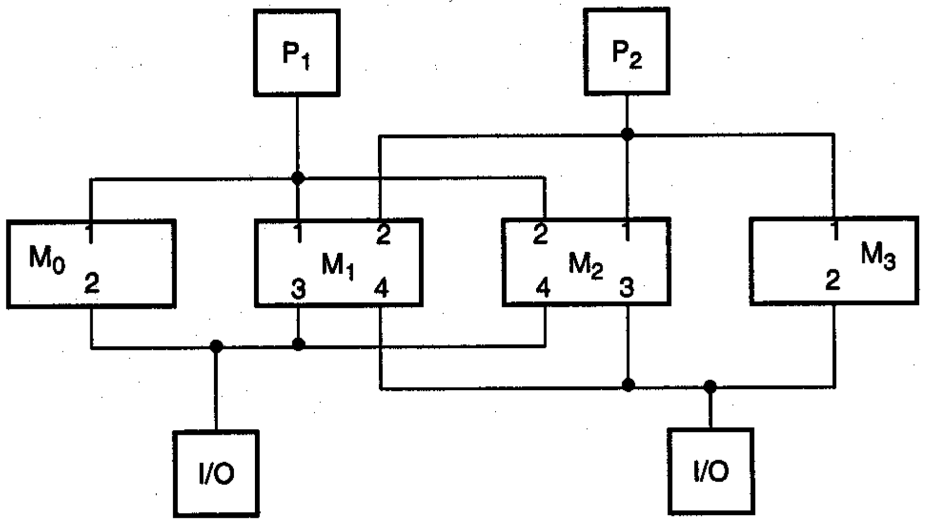
Since crossbar switches are expensive, and not suitable for systems with many processors or memory modules, *multiport memory* modules may be used instead.

A multiport memory module has multiple connections points for processors (or I/O devices), and the memory controller in the module handles the arbitration and switching that might otherwise have been accomplished by a crosspoint switch.

Multiport Memory Examples



(a) n-port memory module



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

7.1.3 Omega Networks

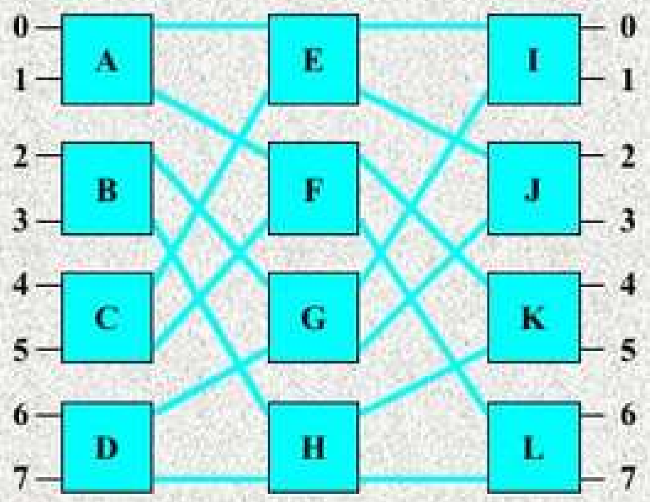
N-input Omega networks, in general, have $\log_2 n$ stages, with the input stage labeled 0.

The interstage connection (ISC) pattern is a perfect shuffle.

Routing is controlled by inspecting the destination address. When the *i*-th highest order bit is 0, the 2×2 switch in stage i connects the input to the upper output. Otherwise it connects the input to the lower output.

8 x 8 Omega Network

- Consists of four 2 x 2 switches per stage.
- The fixed links between every pair of stages are identical.
- A perfect shuffle is formed for the fixed links between every pair of stages.
- Has complexity of O(n lg n).
- For 8 possible inputs, there are a total of 8! = 40,320 1 to 1 mappings of the inputs onto the outputs. But only 12 switches for a total of 2^{12} = 4096 settings. Thus, network is blocking.



How to read the figure:

•Pick a number at the left (e.g., 4 = 100)

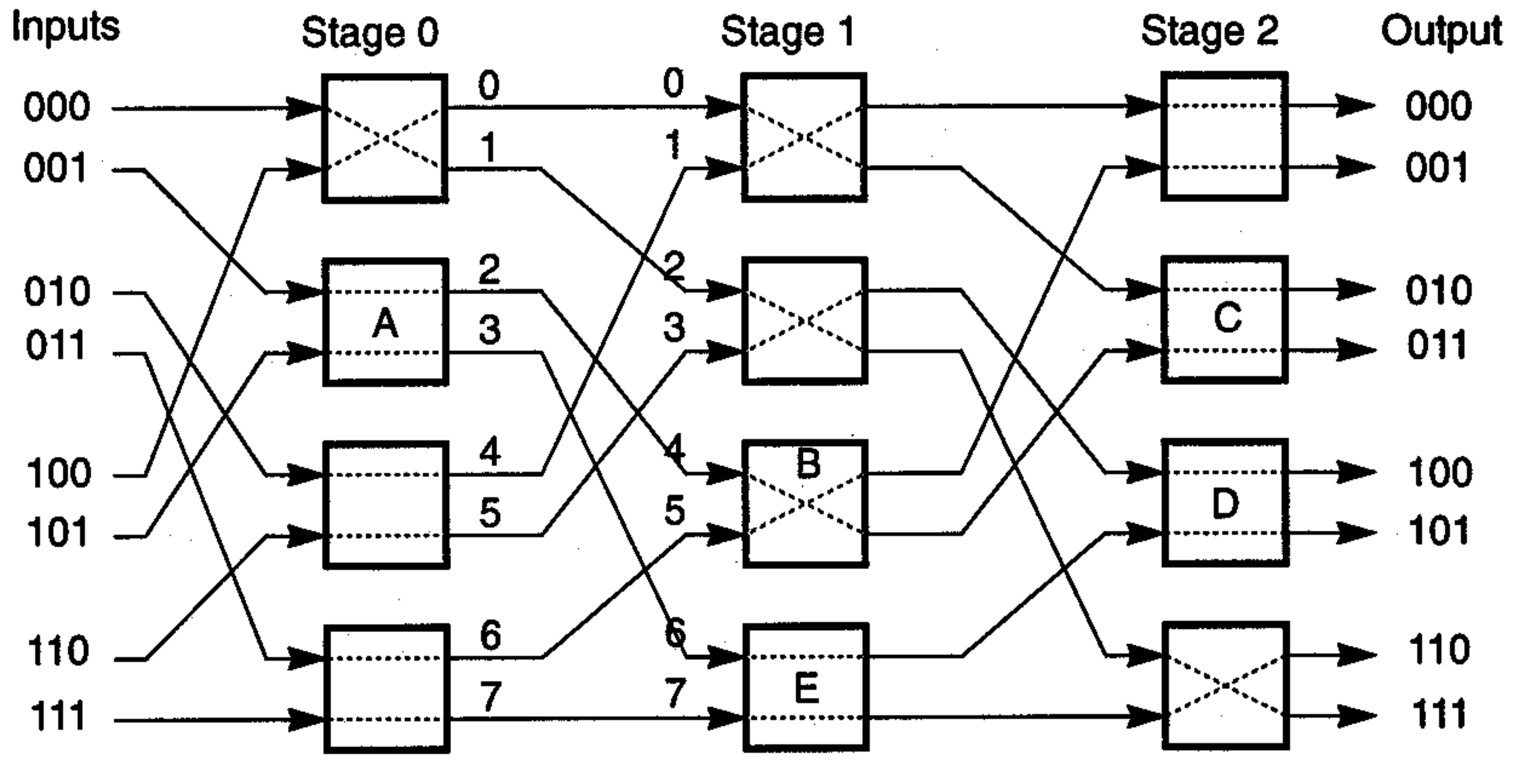
•Rotate left: 100 ---> 001 (= 1)

•Connect 4 to 1

You have to do this in **every stage**

no.stages depends on i/p o/p bits

Omega Network without Blocking



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

Consider the routing message from input 001 to output 011.

Involves switches A,B and C.

compare MSB bit

(001 to 011): straight, cross, straight

(101 to 101): A, E, D straight

Blocking Effects

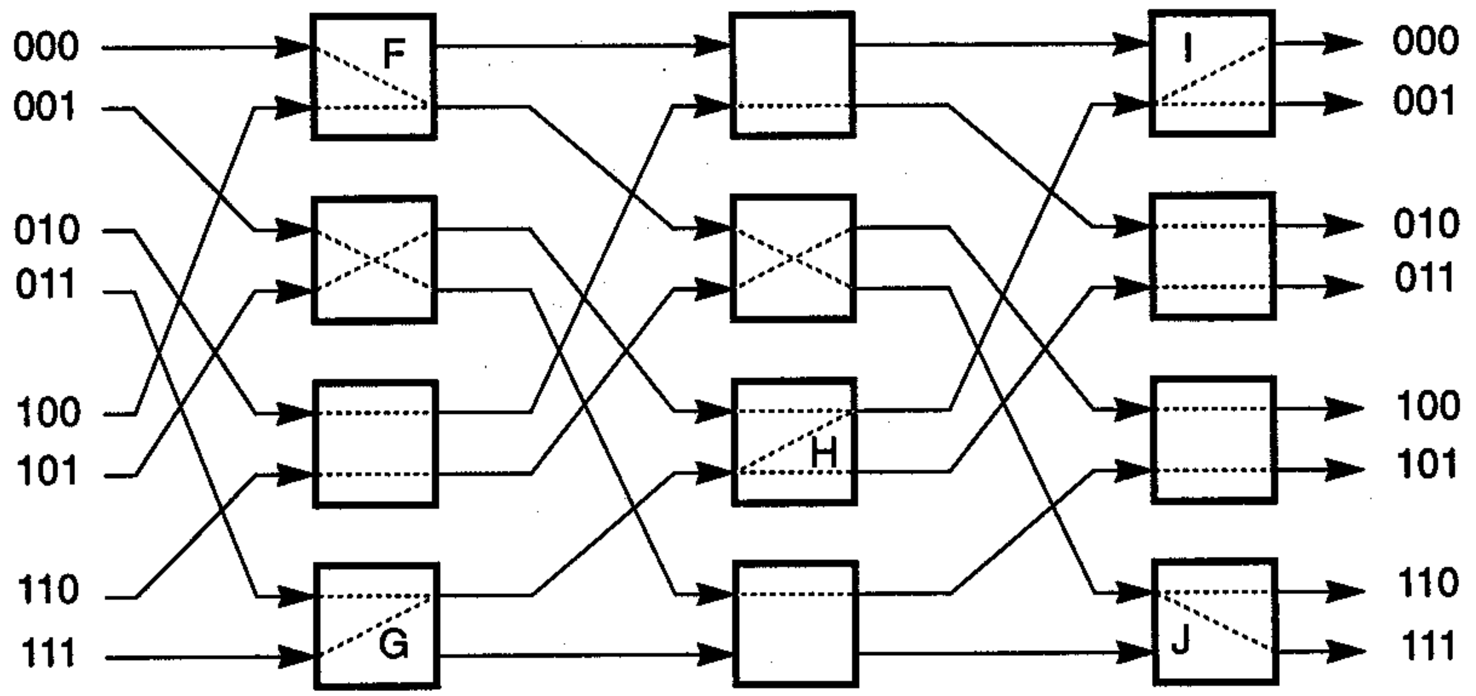
Blocking exists in an Omega network when the requested permutation would require that a single switch be set in two positions simultaneously.

Obviously this is impossible, and requires that one of the permutation requests be blocked and tried in a later pass.

In general, with 2×2 switches, an Omega network can implement $n^{n/2}$ permutations in a single pass. For $n = 8$, this is about 10% of all possible permutations.

In general, a maximum of $\log_2 n$ passes are needed for an n-input Omega network.

Omega Network with Blocking



(b) Permutation $\pi_2 = (0,6,4,7,3)(1,5)(2)$ blocked at switches marked F, G, and H

Conflicts at F by 000->110 and 100->111

Conflicts at G by 011->000 and 111->011

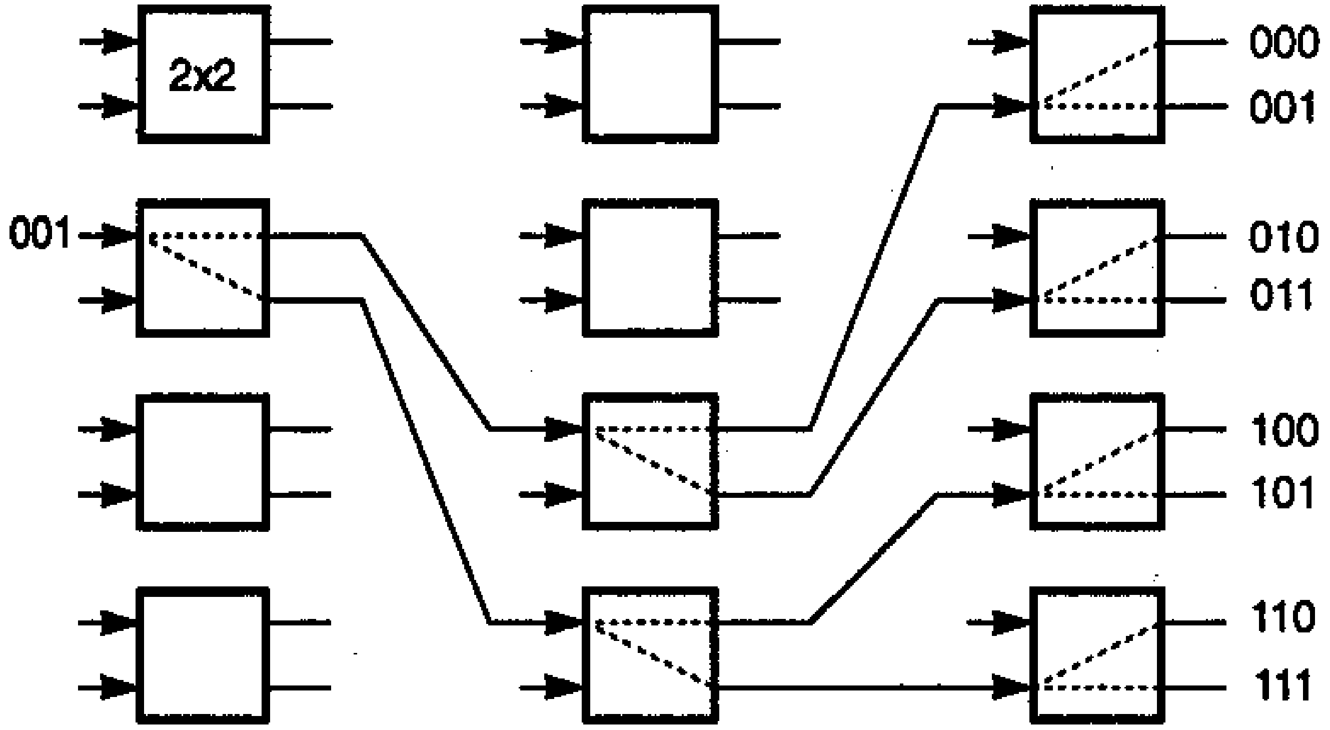
Omega Broadcast

An Omega network can be used to broadcast data to multiple destinations.

The switch to which the input is connected is set to the broadcast position (input connected to both outputs).

Each additional switch (in later stages) to which an output is directed is also set to the broadcast position.

Omega Broadcast



(a) Broadcast connections

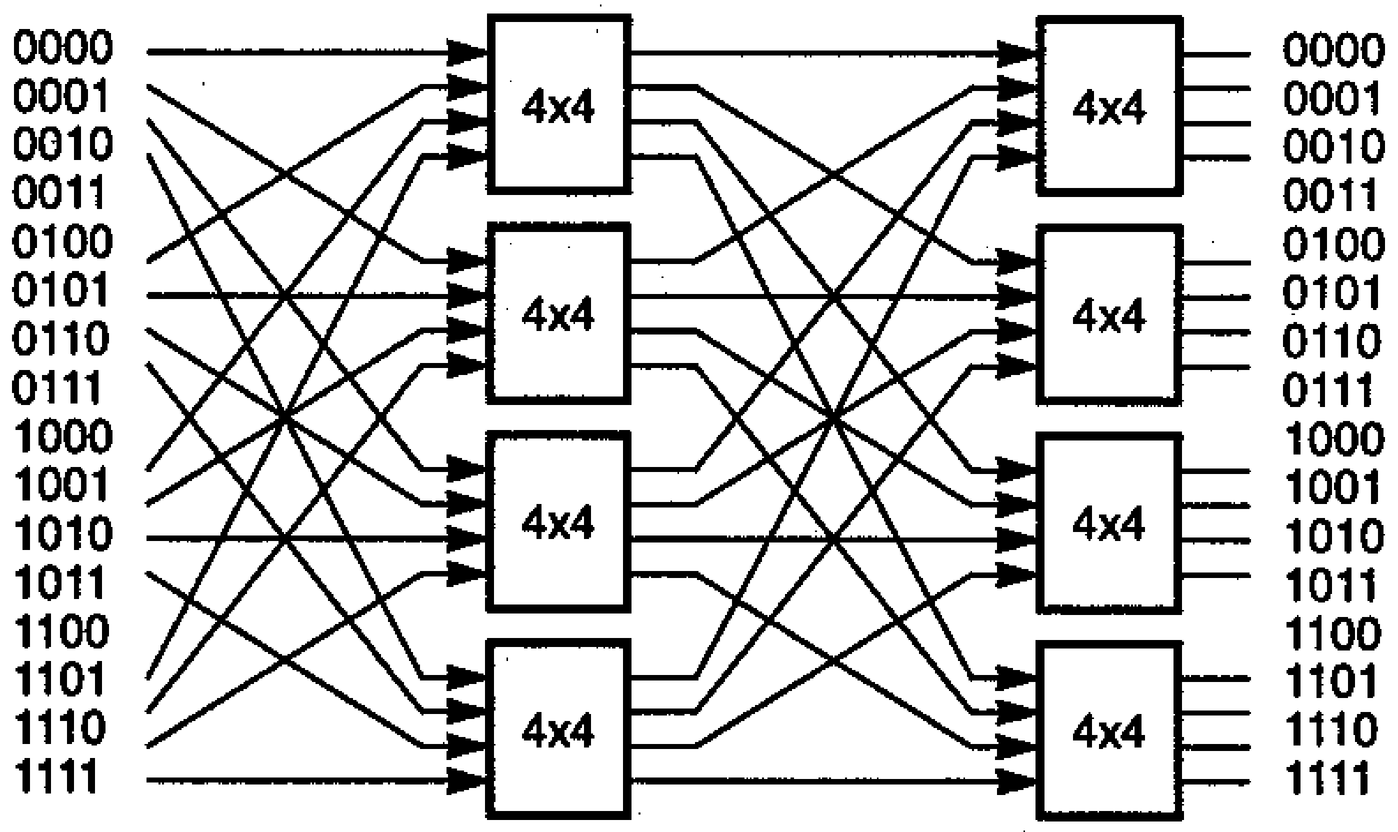
Larger Switches

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 $\log_4 16$ 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.

Omega Network with 4×4 Switches



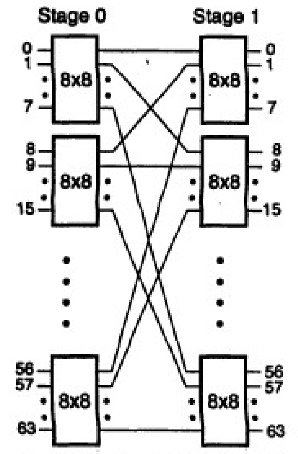
Butterfly Networks

Butterfly networks are built using crossbar switches instead of those found in Omega networks.

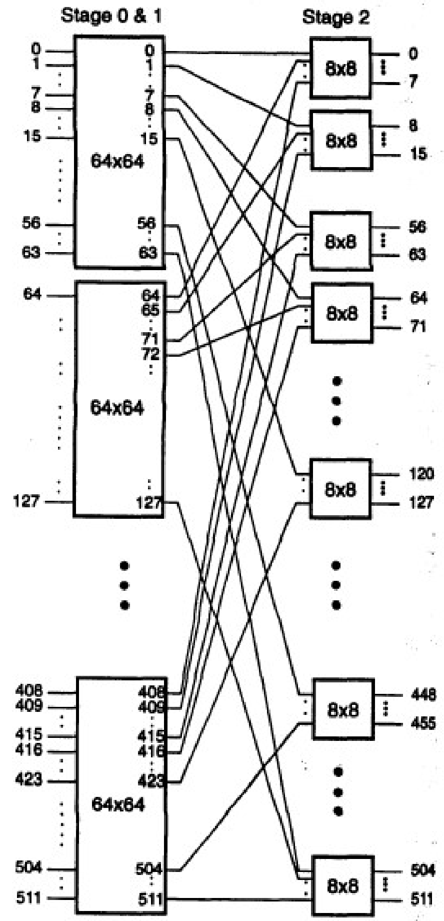
There are no broadcast connections in a butterfly network, making them a restricted subclass of the Omega networks.

Ex BBN Butterfly machine

See also MQ p 70-72



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



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

Figure 7.10 Modular construction of Butterfly switch networks with 8 × 8 crossbar switches. (Courtesy of BBN Advanced Computers, Inc., 1990)

(Hwang)

Hot Spots

When a particular memory module is being heavily accessed by multiple processors at the same time, we say a *hot spot* exists.

For example, if multiple processors are accessing the same memory location with a spin lock implemented with a test and set instruction, then a hot spot may exist.

Obviously, hot spots may significantly degrade the network performance.

Dealing With Hot Spots

To avoid the hot spot problems, we may develop special operations that are actually implemented partially by the network.

Consider the instruction Fetch&Add(x,e), which has the following definition (x is a memory location, and the returned value is stored in a processor register and e integer increment):

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

Implementing Fetch&Add

When *n* processors attempt to execute Fetch&Add on the same location simultaneously, the network performs a serialization on the requests, performing the following steps atomically.

- *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 x.

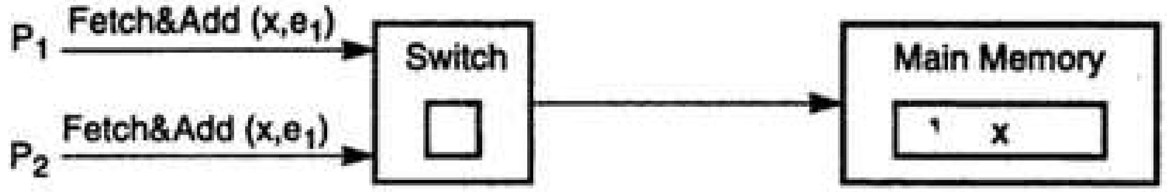
Note that multiple simultaneous test and set instructions could be handled in a similar manner.

The Cost of Fetch&Add

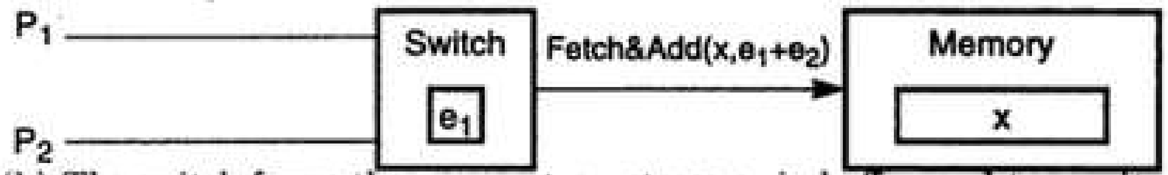
Clearly a feature like Fetch&Add is not available at no cost.

Each switch in the network must be built to detect the Fetch&Add requests (distinct from other requests), queuing them until the operation can be atomically completed.

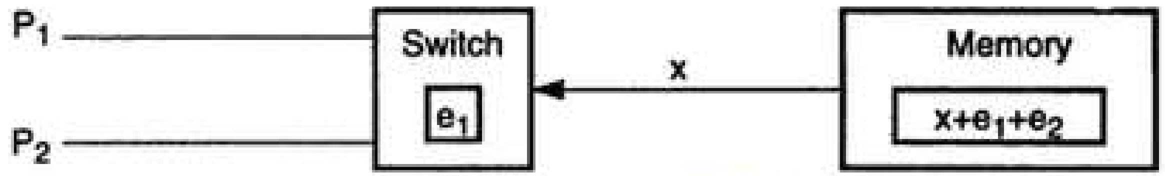
Additional switch cycles may be required, increasing network latency significantly.



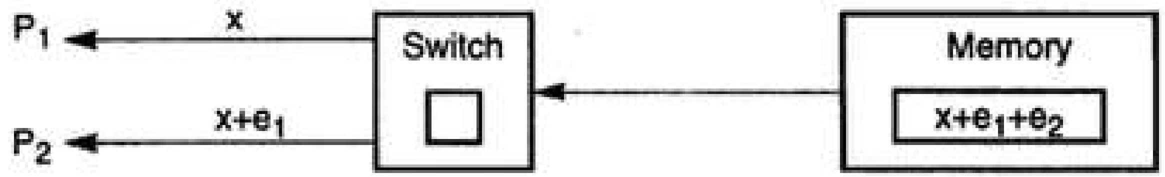
(a) Two requests meet at a switch



(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

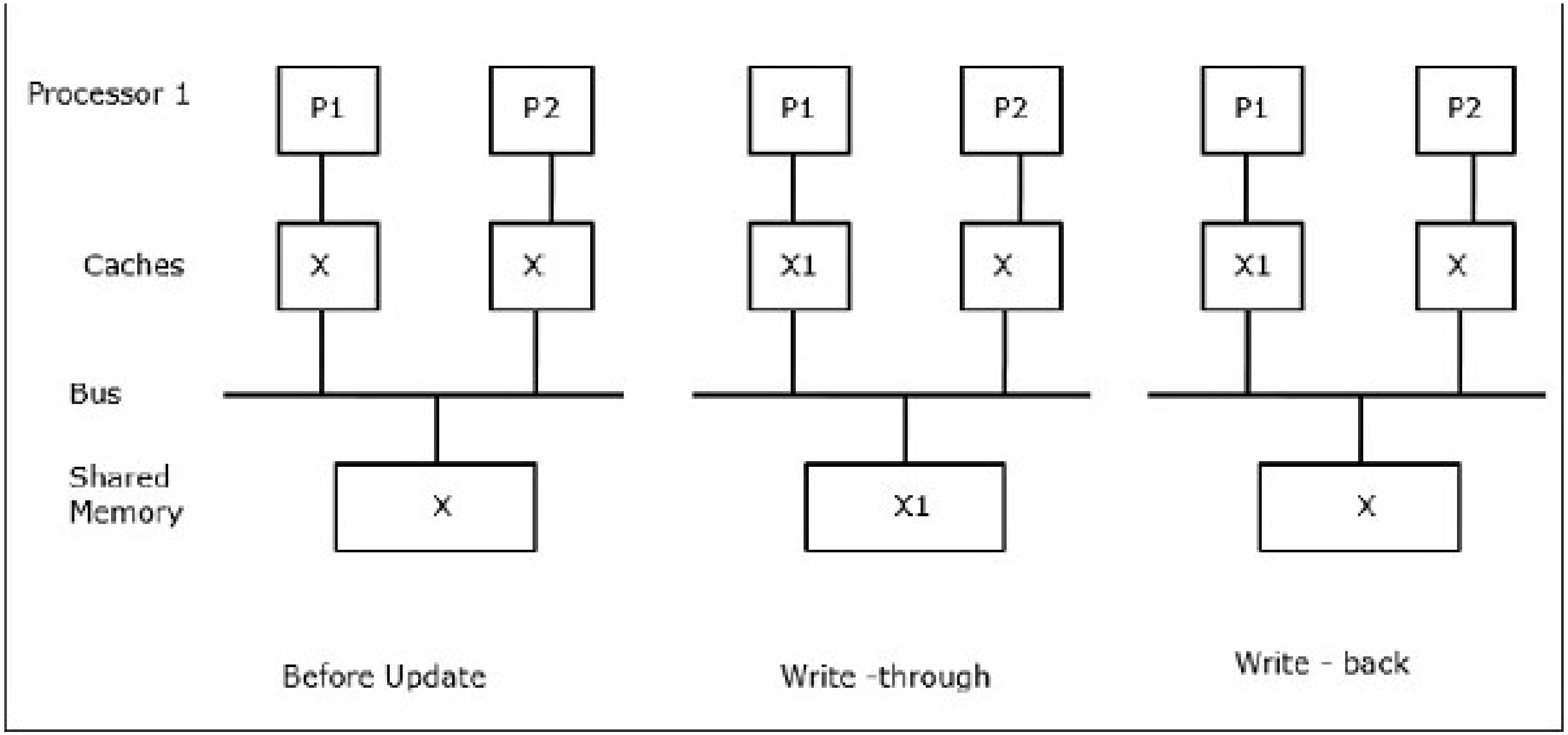
7.2 CACHE COHERENCE & SYNCHRONIZATION

The Cache Coherence Problem

In a multiprocessor system, data inconsistency may occur among adjacent levels or within the same level of the memory hierarchy. For example, the cache and the main memory may have inconsistent copies of the same object.

As multiple processors operate in parallel, and independently multiple caches may possess different copies of the same memory block, this creates **cache coherence problem**.

Cache coherence schemes help to avoid this problem by maintaining a uniform state for each cached block of data.


Processor 1
Caches
Bus
Shared Memory
P1
P2
X
X
X
Before Update
P1
P2
X1
X
X1
Write -through
P1
P2
X1
X
X
Write - back

Let X be an element of shared data which has been referenced by two processors, P1 and P2. In the beginning, three copies of X are consistent.

If the processor P1 writes a new data X1 into the cache, by using **write-through policy**, the same copy will be written immediately into the shared memory. In this case, inconsistency occurs between cache memory and the main memory.

When a **write-back policy** is used, the main memory will be updated when the modified data in the cache is replaced or invalidated.

Causes of Cache Inconsistency

In general, there are three sources of inconsistency problem –

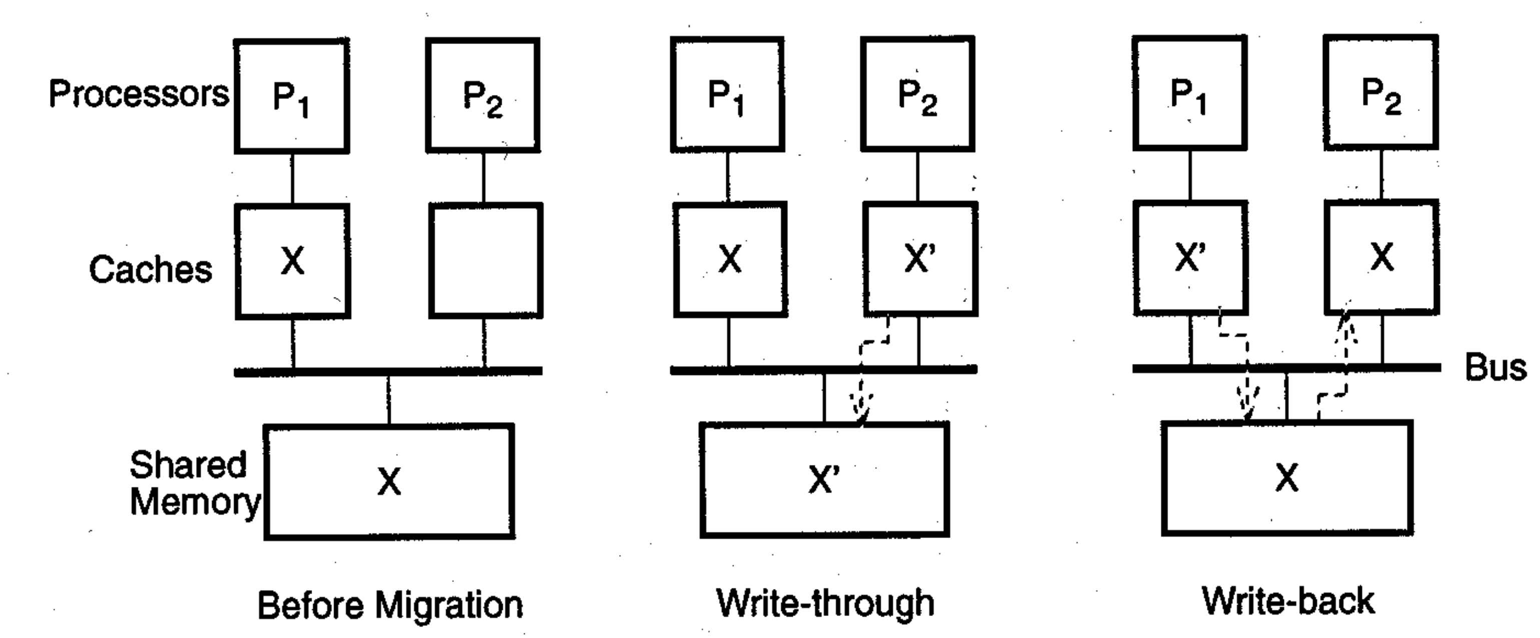
- Sharing of writable data
- Process migration
- I/O activity

Inconsistency After Process Migration

If a process accesses variable X (resulting in it being placed in the processor cache), and is then moved to a different processor and modifies X (to X1), then the caches on the two processors are inconsistent.

This problem exists regardless of whether write-through caches or write-back caches are used.

Inconsistency after Process Migration



Before Migration

Write-through

Write-back

Inconsistency Caused by I/O

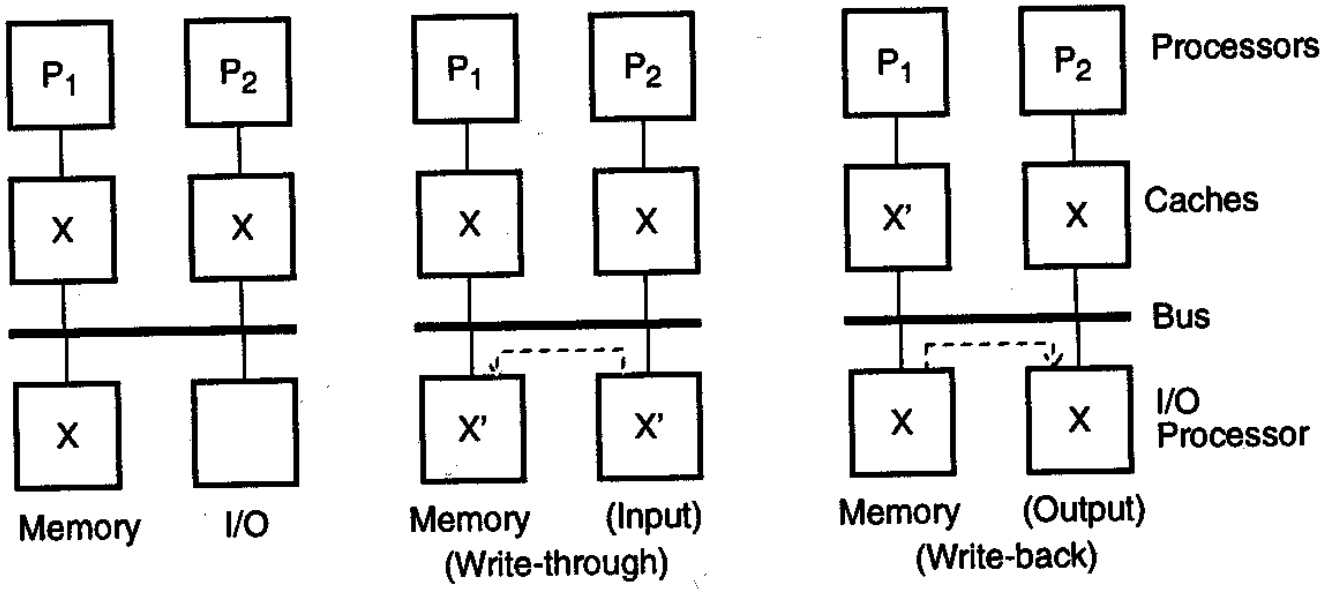
Data movement from an I/O device to a shared primary memory usually does not cause cached copies of data to be updated.

As a result, an input operation that writes X causes it to become inconsistent with a cached value of X.

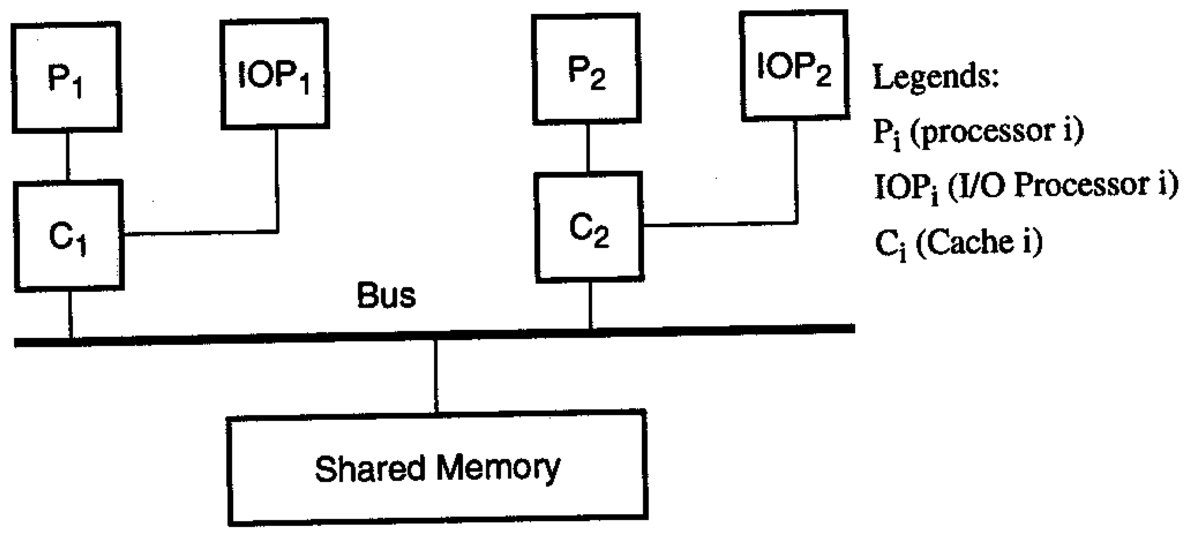
Likewise, writing data to an I/O device usually use the data in the shared primary memory, ignoring any potential cached data with different values.

A potential solution to this problem is to require the I/O processors to maintain consistency with at least one of the processor’s private caches, thus “passing the buck” to the processor cache coherence solution (which will we see).

I/O Operations Bypassing the Cache



A Possible Solution



Cache Coherence Protocols

When a bus is used to connect processors and memories in a multiprocessor system, each cache controller can "snoop" on all bus transactions, whether they involve the current processor or not. If a bus transaction affects the consistency of a locally-cached object, then the local copy can be invalidated.

If a bus is not used (e.g. a crossbar switch or network is used), then there is no convenient way to "snoop" on memory transactions. In these systems, some variant of a directory scheme is used to insure cache coherence.

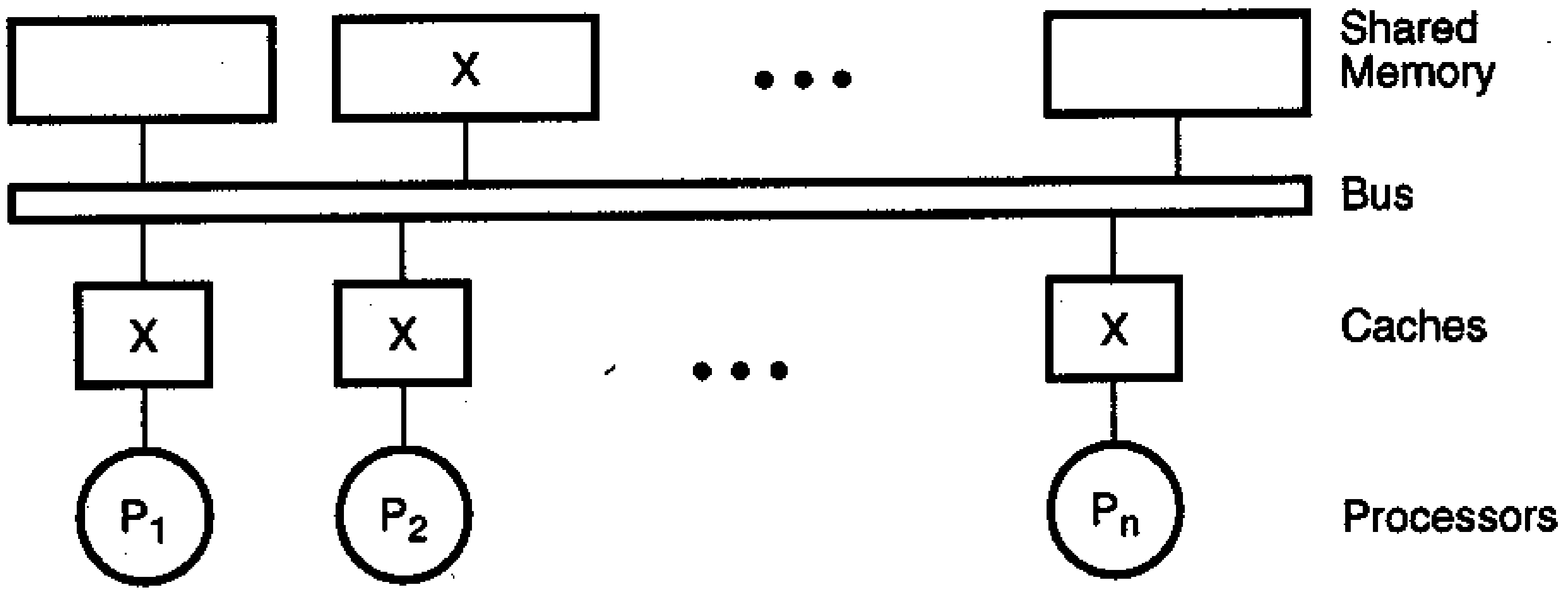
7.2.2 Snoopy Bus Protocols

Two basic approaches

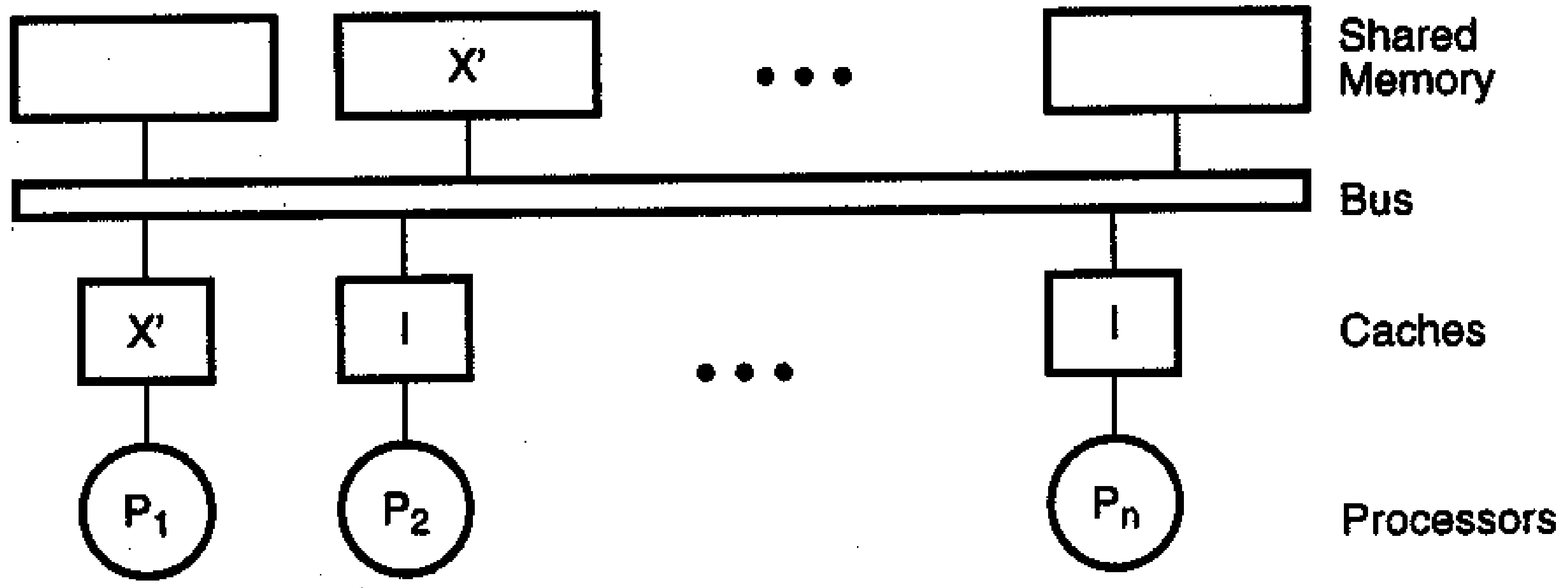
- write-invalidate – invalidate all other cached copies of a data object when the local cached copy is modified (invalidated items are sometimes called "dirty")
- write-update – broadcast a modified value of a data object to all other caches at the time of modification

Snoopy bus protocols achieve consistency among caches and shared primary memory by requiring the bus interfaces of processors to watch the bus for indications that require updating or invalidating locally cached objects.

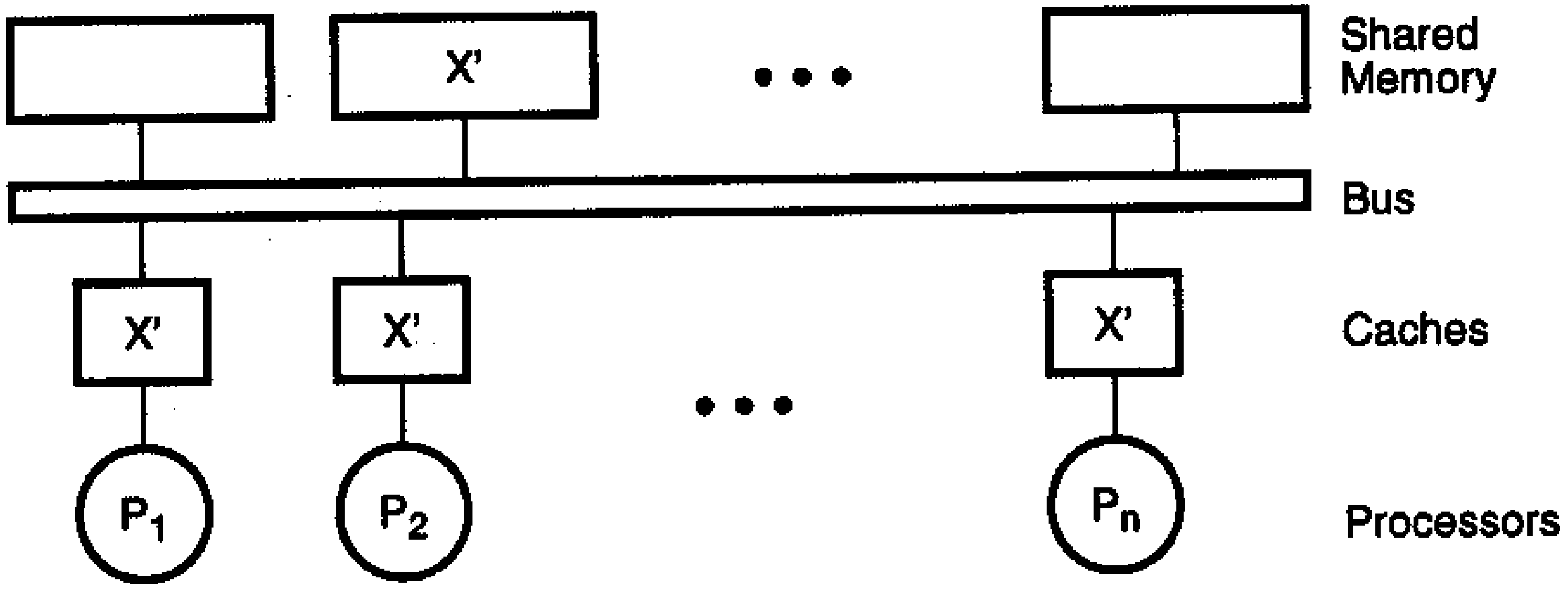
Initial State – Consistent Caches



After Write-Invalidate by P_1



After Write-Update by P_1



Operations on Cached Objects

Read – as long as an object has not been invalidated, read operations are permitted, and obviously do not change the object's state

Write – as long as an object has not been invalidated, write operations on the local object are permitted, but trigger the appropriate protocol action(s).

Replace –the cache block containing an object is replaced (by a different block)

Write-Through Cache

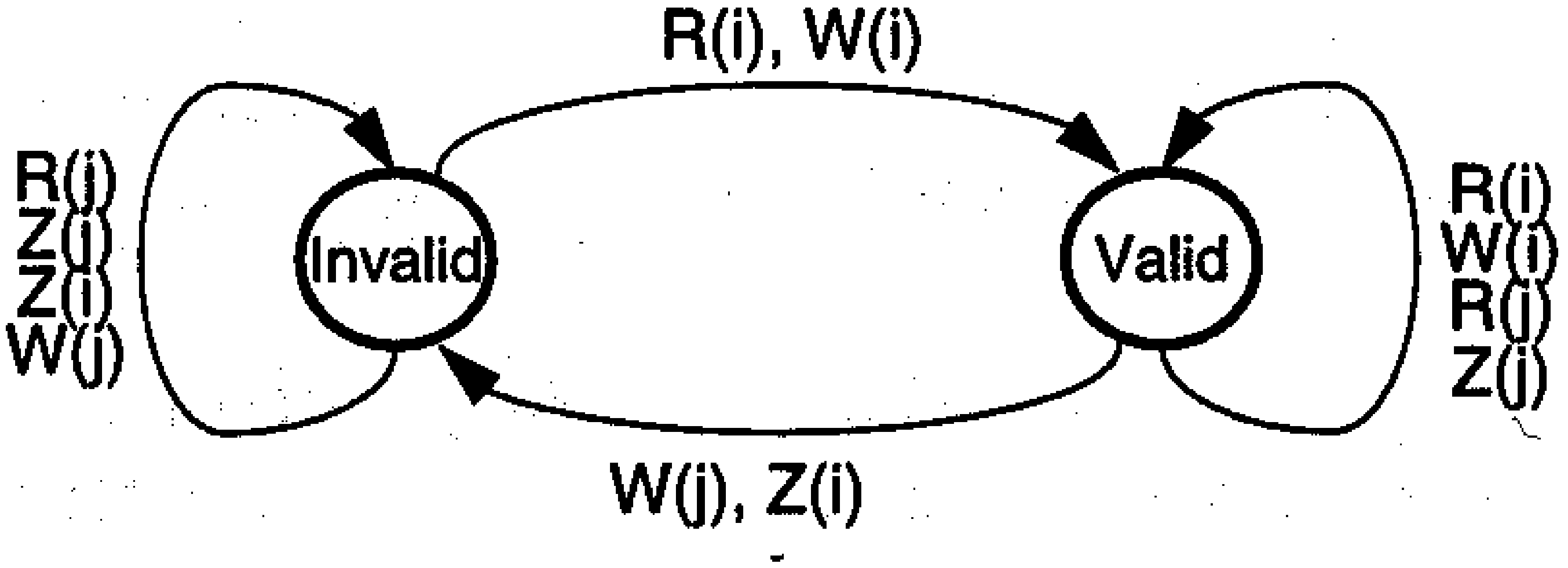
In the transition diagram (next slide), the two possible object states in the "local" cache (valid and invalid) are shown.

The operations that may be performed are read, write, and replace by the local processor or a remote processor.

Transitions from locally valid to locally invalid occur as a result of a remote processor write or a local processor replacing the cache block.

Transitions from locally invalid to locally valid occur as a result of the local processor reading or writing the object (necessitating, of course, the fetch of a consistent copy from shared memory).

Write-Through Cache State Transitions



R = Read, W = Write, Z = Replace
i = local processor, j = other processor

Write-Back Cache

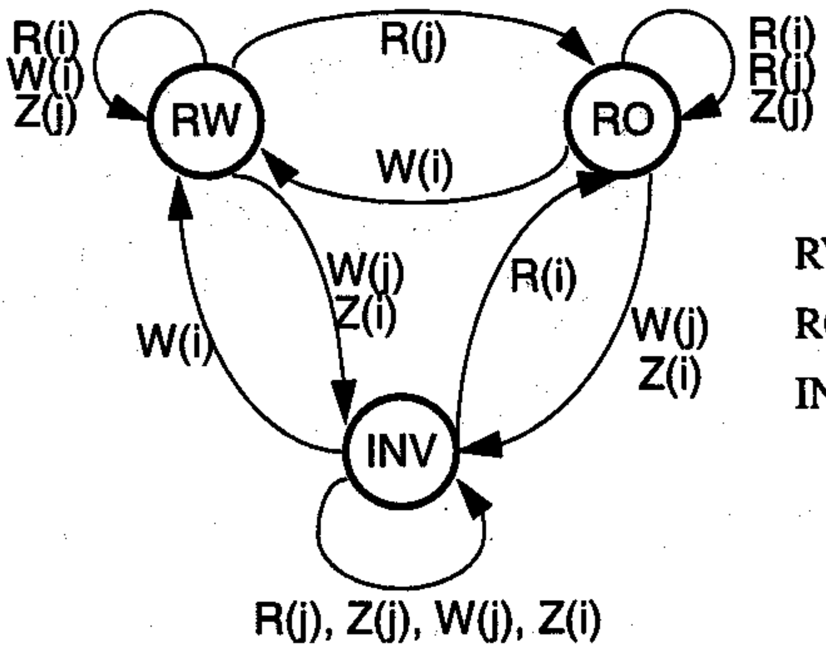
The state diagram for the write-back protocol divides the valid state into RW and RO states.

The protocol essentially gives “ownership” of the cache block containing the object to a processor when it does a write operation.

Before an object can be modified, ownership for exclusive access must first be obtained by a read-only bus transaction which is broadcast to all caches and memory.

If a modified block copy exists in a remote cache, memory must first be updated, the copy invalidated, and ownership transferred to the requesting cache.

Write-Back Cache



RW: Read-Write

RO: Read Only

INV: Invalidated or not in cache

W(i) = Write to block by processor *i*.

R(i) = Read block by processor *i*.

Z(i) = Replace block in cache *i*.

W(j) = Write to block copy in cache *j* by processor $j \neq i$.

R(j) = Read block copy in cache *j* by processor $j \neq i$.

Z(j) = Replace block copy in cache $j \neq i$.

Goodman’s Cache Coherence Protocol

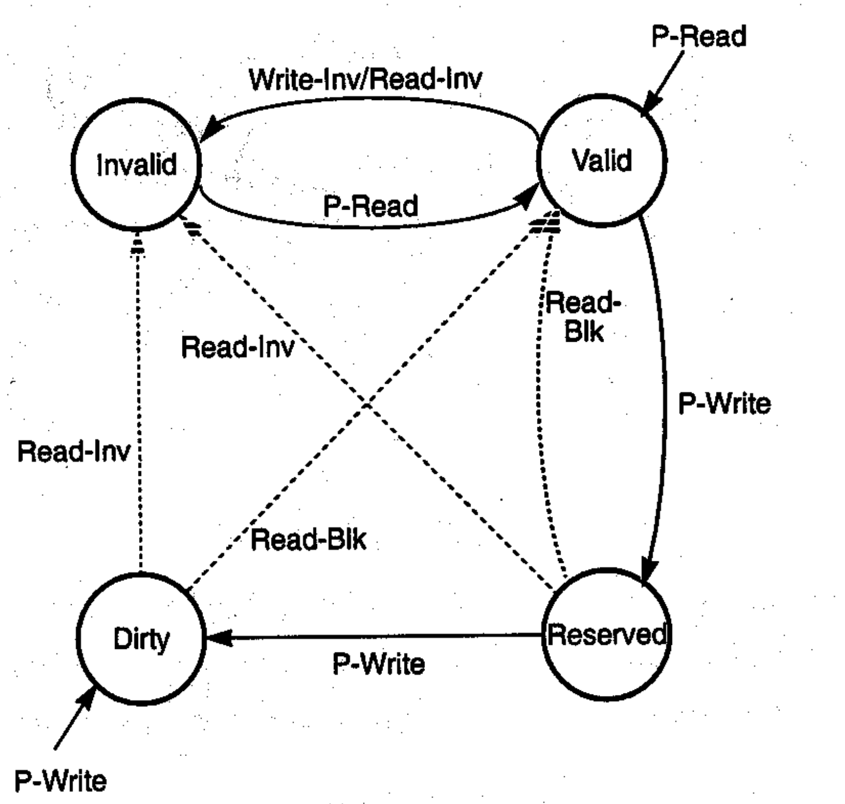
Combines advantages of write-back and write-through protocols.

First write of a cache block uses write-through.

Cache states (see previous slide):

- Valid: block is consistent with memory, has been read, but not modified.
- Invalid: block not in cache, or is inconsistent with memory.
- Reserved: block written once after being read and is consistent with memory copy (which is the only other copy).
- Dirty: block modified more than once, inconsistent with all other copies.

Goodman's Write-Once Protocol State Diagram



Solid lines: Command issued by local processor

Dashed lines: Commands issued by remote processors via the system bus.

Commands and State Transitions

Local processor accesses:

- Read-hit or read-miss (P-Read) – transition to <u>valid</u> state.
- Write-hit (P-Write)
  - First one results in transition to <u>reserved</u> state.
  - Additional writes go to (or stay in) <u>dirty</u> state.
- Write-miss – transition to <u>dirty</u> state.

Remote processor invalidation commands (issued over snoopy bus):

- Read-invalidate – read a block and invalidate all other copies.
- Write-invalidate – invalidate all other copies of a block.
- Bus-read (Read-blk) – normal read; transition to <u>valid</u> state.

Cache events and actions

Read miss

Write miss

Read hit

Write hit

Block replacement

Snoopy Bus Protocol Performance

Depends heavily on the workload.

Goal

- Reduce bus traffic and effective memory access time

In uniprocessors:

- bus traffic and memory-access time heavily influenced by cache misses.
- Miss ratio increases 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).
- Data pollution point increases with larger cache sizes.

Snoopy Bus Protocol Performance

In multiprocessor systems

- Write-invalidate protocol
  - Better handles process migrations and synchronization than other protocols.
  - Cache misses can result from invalidations sent by other processors before a cache access, which significantly increases bus traffic.
  - Bus traffic may increase as block sizes increase.
  - Write-invalidate facilities writing synchronization primitives.
  - Average number of invalidated cache copies is small in a small multiprocessor.
- Write-update procotol
  - Requires bus broadcast facility
  - May update remote cached data that is never accessed again
  - Can avoid the back and forth effect of the write-invalidate protocol for data shared among multiple caches
  - Can’t be used with long write bursts
  - Requires extensive tracing to identify actual behavior

7.2.3 Directory-based Protocols

The snoopy bus-based protocols may be adequate for relatively small multiprocessor systems, but are wholly inadequate for large multiprocessor systems.

Commands (in the form of messages) to control the consistency of remote caches must be sent only to those processors with caches containing a copy of the affected block (since broadcast is very expensive in a multistage network – like Omega).

This gives rise to *directory-based protocols*.

Directory Structures

Cache directories store information on where (in which processors) copies of cache blocks reside.

Central directory approaches (with copies of all cache directories) is very large, and requires an associative search (like the individual cache directories).

Distributed directory approaches maintains separate director which records state and presence information for each memory block.

Types of Directory Protocols

Directory entries are pairs identifying cache blocks and processor caches holding those blocks.

Three different types of directory protocols:

- Full-map directories – each directory entry can identify all processors with cached copies of data; with *N* processors, each directory entry must have *N* processor ID identifiers.
- Limited directories – each entry has a fixed number of processor identifiers, regardless of the system size.
- Chained directories – emulate full-map directories by distributing entries among the caches.

Full-map Protocols

Directory entries have one bit per processor in the system, and another bit to indicate if the data has been modified (“dirty”).

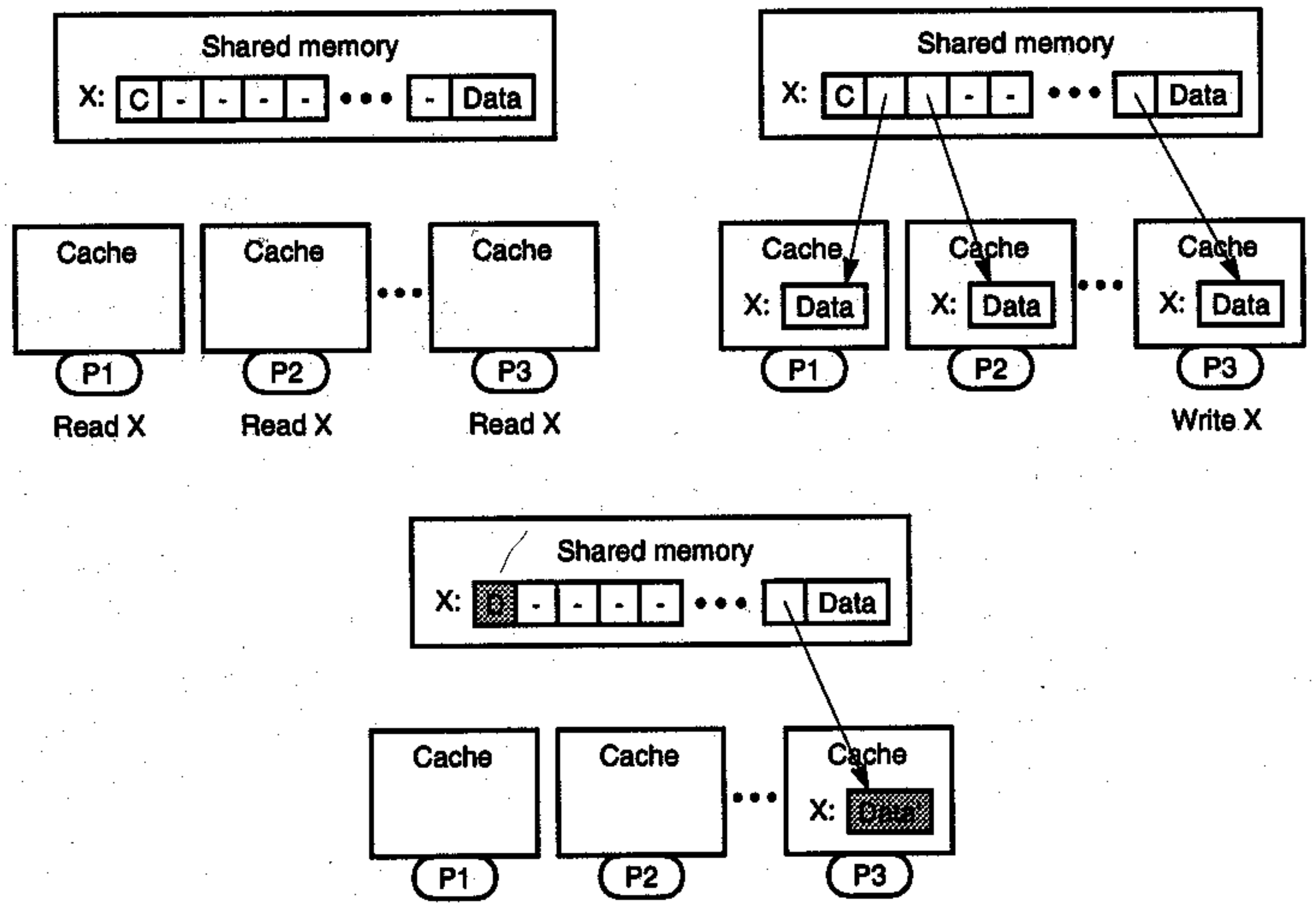
If the dirty bit is set, then only one processor must be identified in the bit map; only that processor is allowed to write the block into memory.

Cache maintains two bits of state information per block:

- Is the cached block valid?
- Can a valid cached block be written to memory?

The purpose of the cache coherence protocol is to keep the cache’s state bits and those in the memory directory consistent.

Three States of a Full-Map Directory



Full Map State Changes

In the first state (upper left in previous slide), X is missing from all caches.

In the second state, three caches are requesting copies of X.   The bits of the three processors are set, and the dirty bit is still ‘C’ (clean), since no processor has requested to write X.

In the third state, the dirty bit is set (‘D’), since a processor requested to write X. Only the corresponding processor has it’s bit set in the map.

Write Actions

Cache C3 detects the block is valid, but the processor doesn't have write permission.

Write request issued to memory, stalling the processor.

Other caches receive invalidate requests and send acknowledgements to memory.

Memory receives acknowledgements, sets dirty bit, clears pointers to other processors, sends write permission to C3.

- By waiting for acknowledgements, the memory ensures sequential consistency.

C3 gets write permission, updates cache state, and reactivates the processor.

Full-Map Protocol Benefits

The full-map protocol provides an upper bound on the performance of centralized directory-based cache coherence.

It is not scalable, however, because of the excessive memory overhead it incurs.

Limited Directories

Designed to solve the directory size problem.

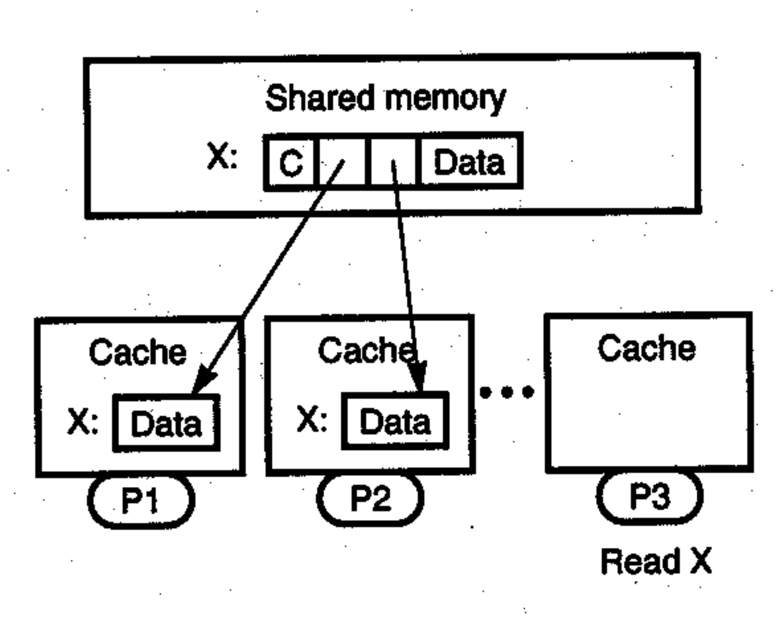
Restricts the number of cached copies of a datum, thus limiting the growth of the directory.

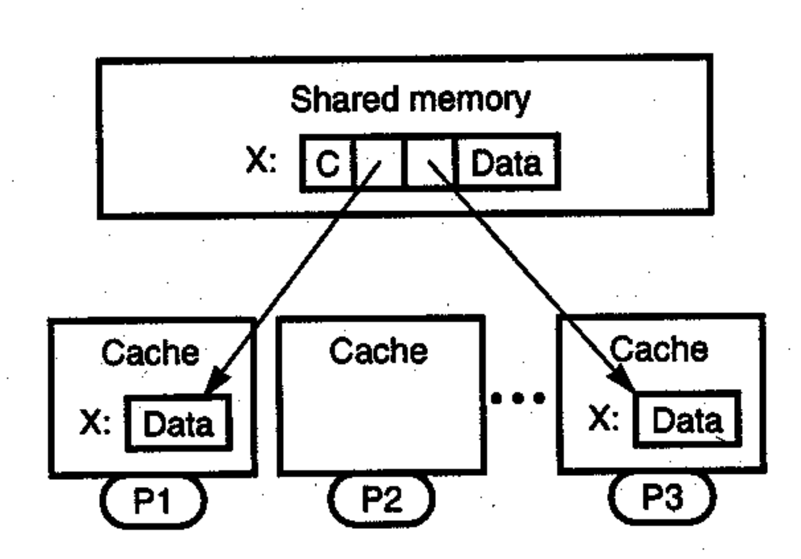
Agrawal notation: $Dir_i X$

- *i* indicates number of pointers in directory
- X is NB for no broadcast, B for broadcast
- E.g. full map with N processors is $Dir_N NB$

In the example (next slide), the left figure shows C1 and C2 holding copies of X. When C3 requests a copy, the C1 or C2 copy must be invalidated using a process called “eviction,” as shown by the right figure.

Eviction in a Limited Directory





Limited Directory Memory Size

In the full-map protocol, it is sufficient to use a single bit to identify if each of the N processors has a copy of the datum.

In a limited directory scheme, processor numbers must be maintained, requiring $log_2 N$ bits each.

If the code being executed on a multiprocessor system exhibits “processor locality,” then a limited directory is sufficient to capture the identity of the processors.

Limited Directory Scalability

Limited directory schemes for cache coherency in non-bus systems are scalable, in that the number of resources required for their implementation grows linearly as the number of processors grows.

Dir_i B protocols exist that allow more than *i* copies of a block to exist in caches, but must use broadcast to invalidate more than *i* copies of a block (because of a write request). Without a broadcast capability in the connection network, ensuring sequential consistency is difficult.

Chained Directories

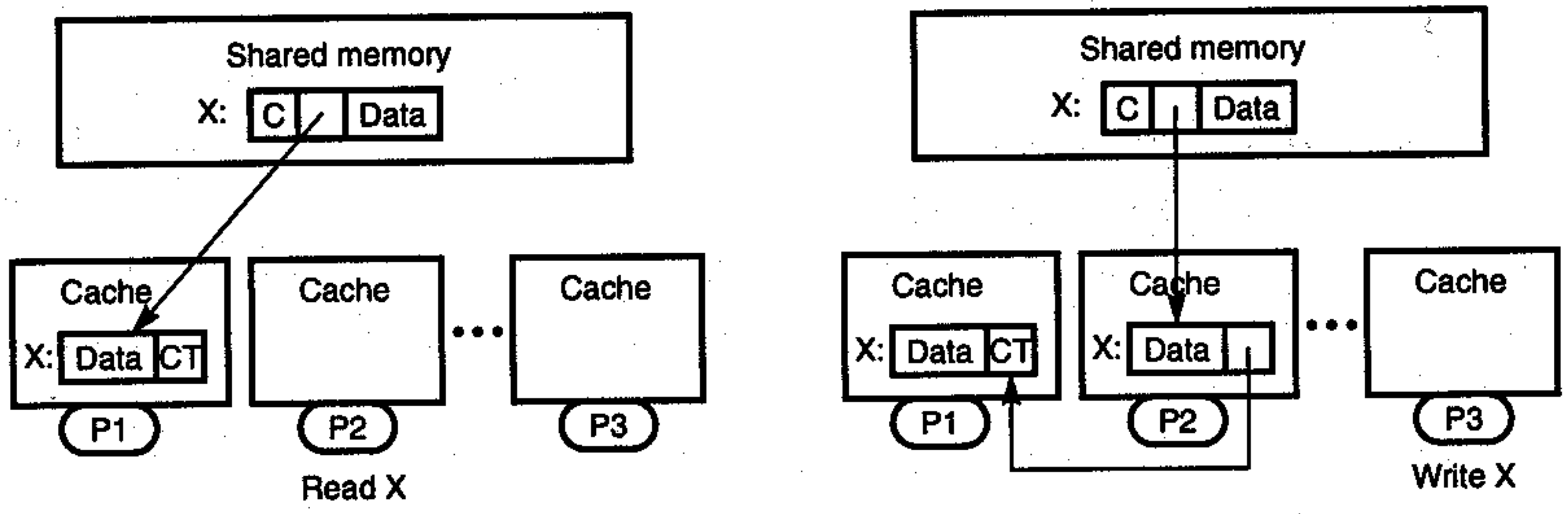
Chained directories are scalable (like limited directories).

They keep track of shared copies of data using a chain of directory pointers.

Each cache must include a pointer (which can be the *chain termination* pointer) to the next cache that contains a datum.

When a processor requests a read, it is sent the datum along with a pointer to the previous head of the list (or a chain termination pointer if it is the only processor requesting the datum).

A Chained Directory Example



Invalidation in Chained Directories

When a processor requests to write a datum, the processor at the head of the list is sent an invalidate request.

Processors pass the invalidate request along until it reaches the processor at the end of the list.

That processor sends an acknowledgement to the memory, which then grants write access to the processor requesting such.

Author suggests this be called the “gossip” protocol.

Complications with Chained Dirs

Suppose processor i requests Y, and the (direct-mapped) cache already contains an entry X which maps to the same location as Y. It must evict X from its cache, thus requiring the list of X's users to be altered.

Two schemes for the list alteration:

- Send a message "down the list" to cache i-1 with a pointer to cache i+1, removing i from the list.
- Invalidate X in caches i+1 through N.

Alternately, a doubly-linked list could be used, with the expected implications for size, speed, and protocol complexity.

Chained directories are scalable, and cache sizes (not number of processors) control the number of pointers.

Alternative Coherency Schemes

Shared caches – allow groups of processors to share caches. Within the group, the coherency problem disappears. Many configurations are possible.

Identify noncacheable data – have the software mark data (using hardware tags) that can be shared (e.g. not instructions or private data), and disallow caching of these.

Flush caches at synchronization – force a rewrite of cached data each time synchronization, I/O, or process migration might affect any of the cached data. Usually this is slow.

Hardware Synchronization Methods

Test and set – TS instruction atomically writes 1 to a memory location and returns its previous value (0 if the controlled resource is free). All processors attempting TS on same location except one will get 1, with one processor getting zero. The “spin lock” is cleared by writing 0 to the location.

Suspend lock – a lock is designed to generate an interrupt when it is released (opened). A process wanting the lock (but finding it closed) will disable disable all interrupts except that associated with the lock and wait.

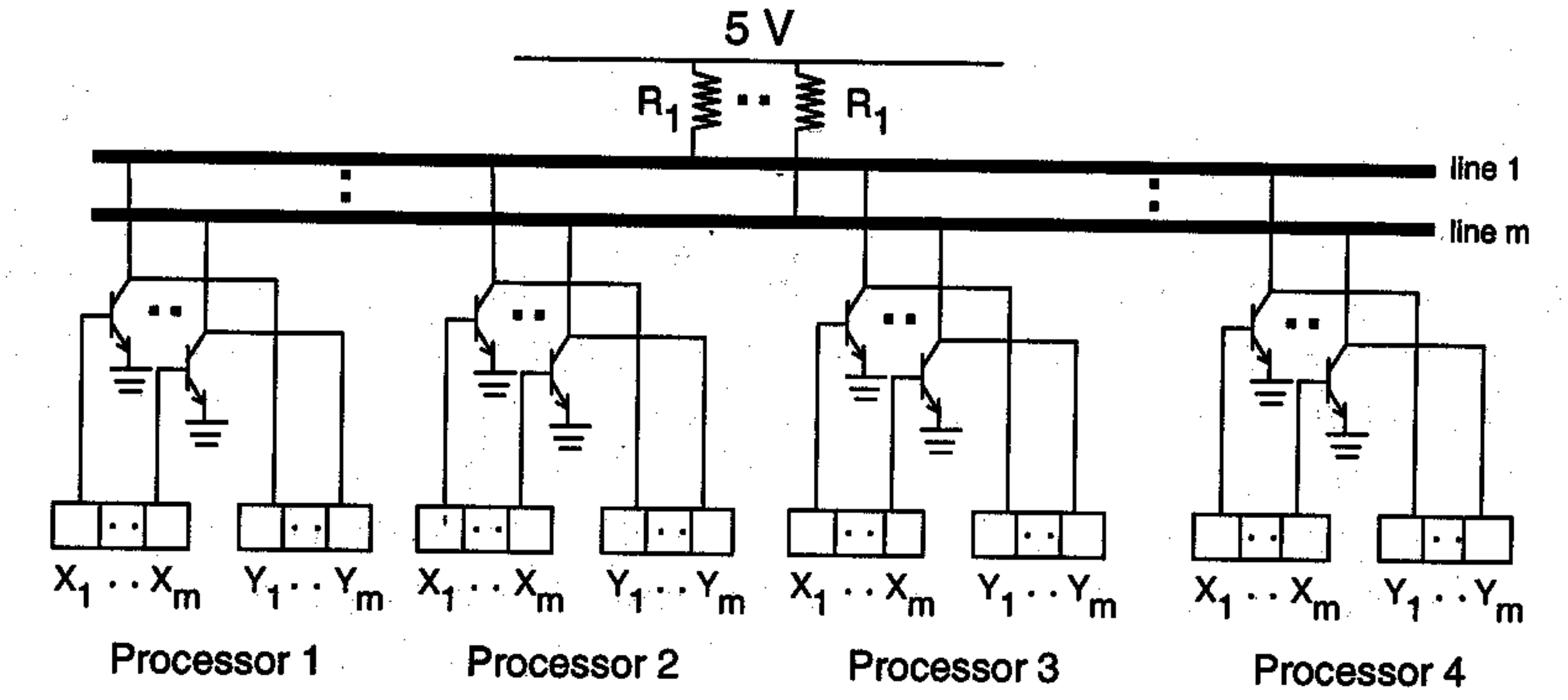
Wired Barrier Synchronization

Barriers are used to block a set of processes until each reaches the same code point.

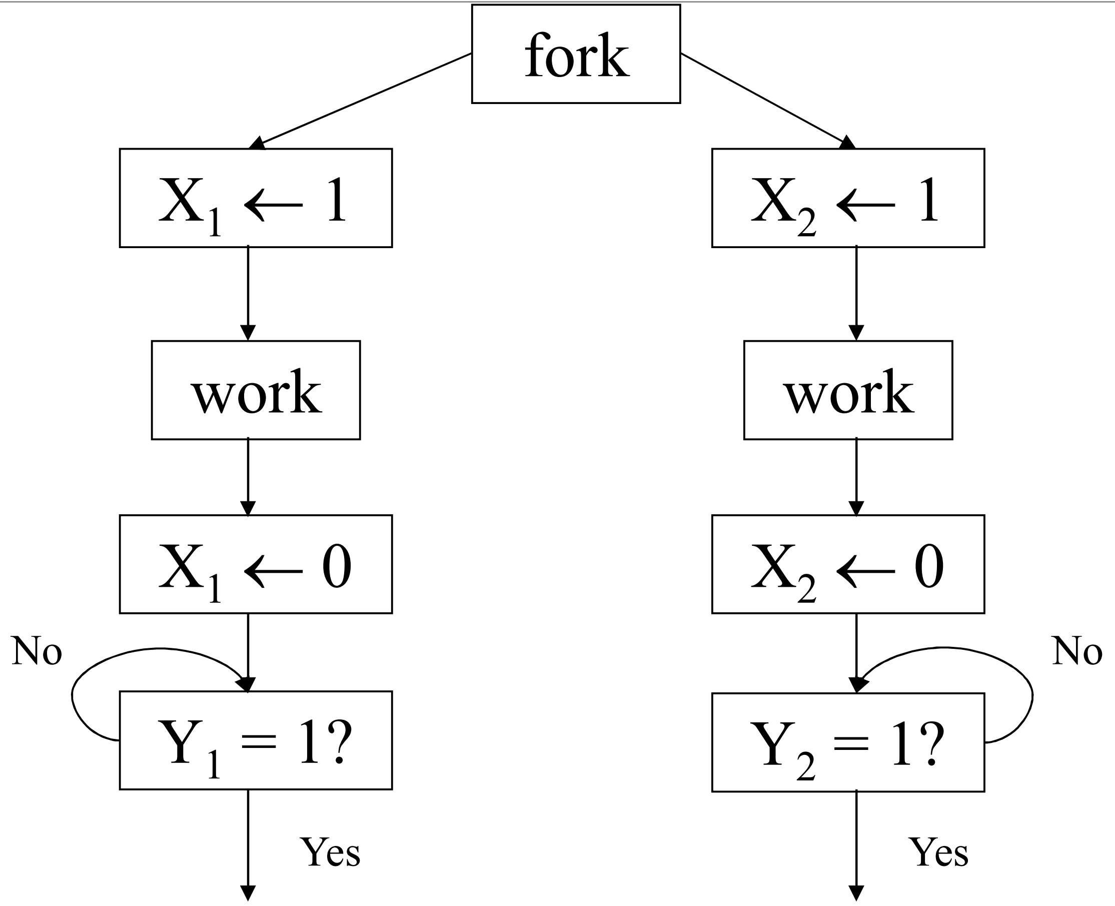
This scheme uses a wire which is “1” unless one of the processors sets its X bit, which forces the wire to “0”. The X bit is set when a process has not yet reached the barrier.

As each process reaches the barrier, it clears its X bit and waits for the Y bit to become “1”; the Y bit reports the state of the wire.

Wired Barrier Implementation



Wired Barrier Example



7.3 THREE GENERATIONS OF MULTICOMPUTERS

Design Choices in the Past

Processors

- Low cost commodity (off-the-shelf) processors

Memory Structure

- Distributed memory organization
- Local memory with each processor

Interconnection Schemes

- Message passing, point-to-point , direct networks with send/receive semantics with/without uniform message communication speed

Control Strategy

- Asynchronous MIMD, MPMD and SPMD operations

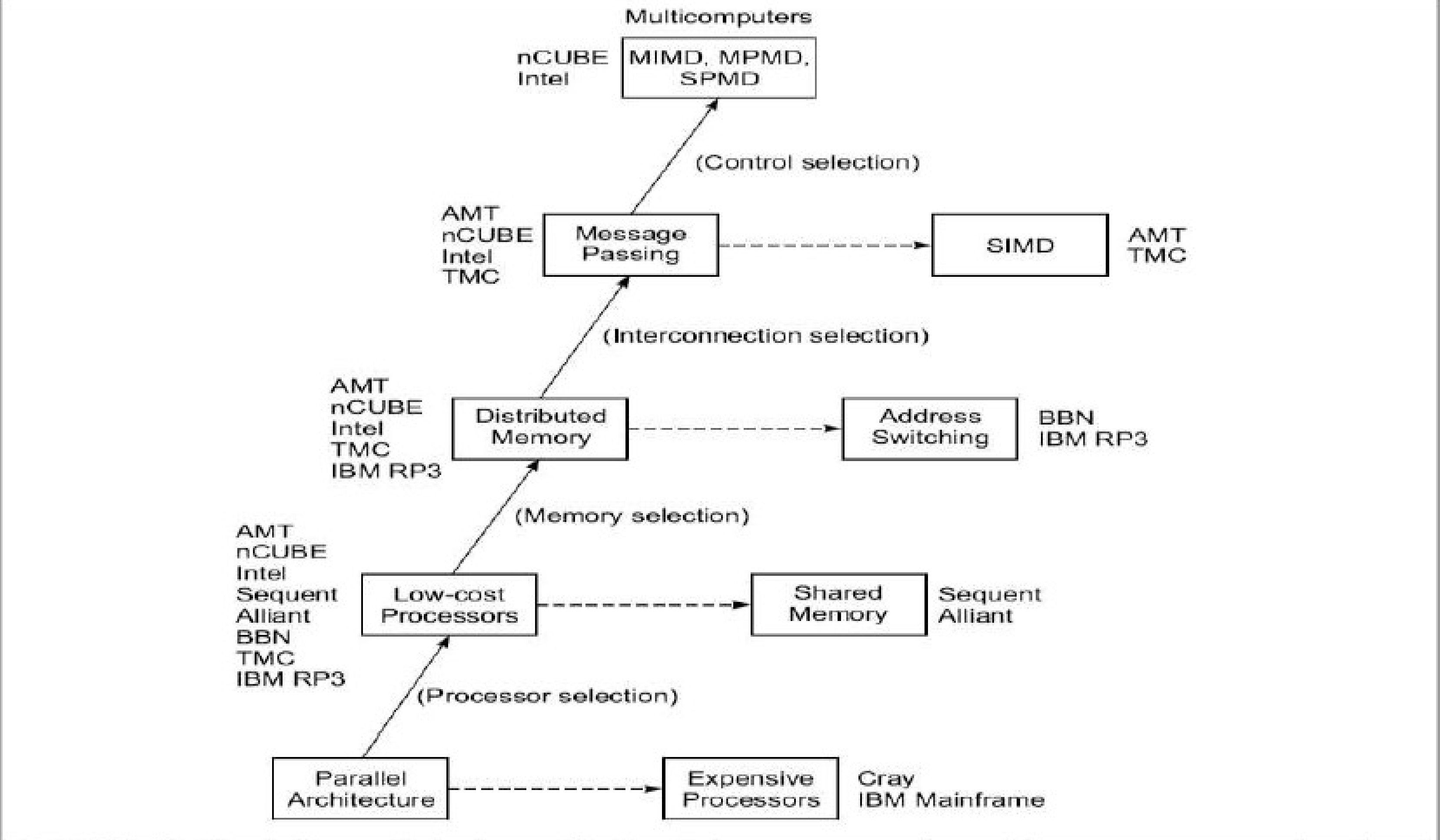


Fig. 7.21 Design choices made in the past for developing message-passing multicomputers compared to those made for other parallel computers (Courtesy of Intel Scientific Computers, 1988)

The Past, Present and Future Development

First Generation

- Example Systems: Caltech’s Cosmic Cube, Intel iPSC/1, Ametek S/14, nCube/10

Second Generation

- Example Systems: iPSC/2, i860, Delta, nCube/2, Supernode 1000, Ametek Series 2010

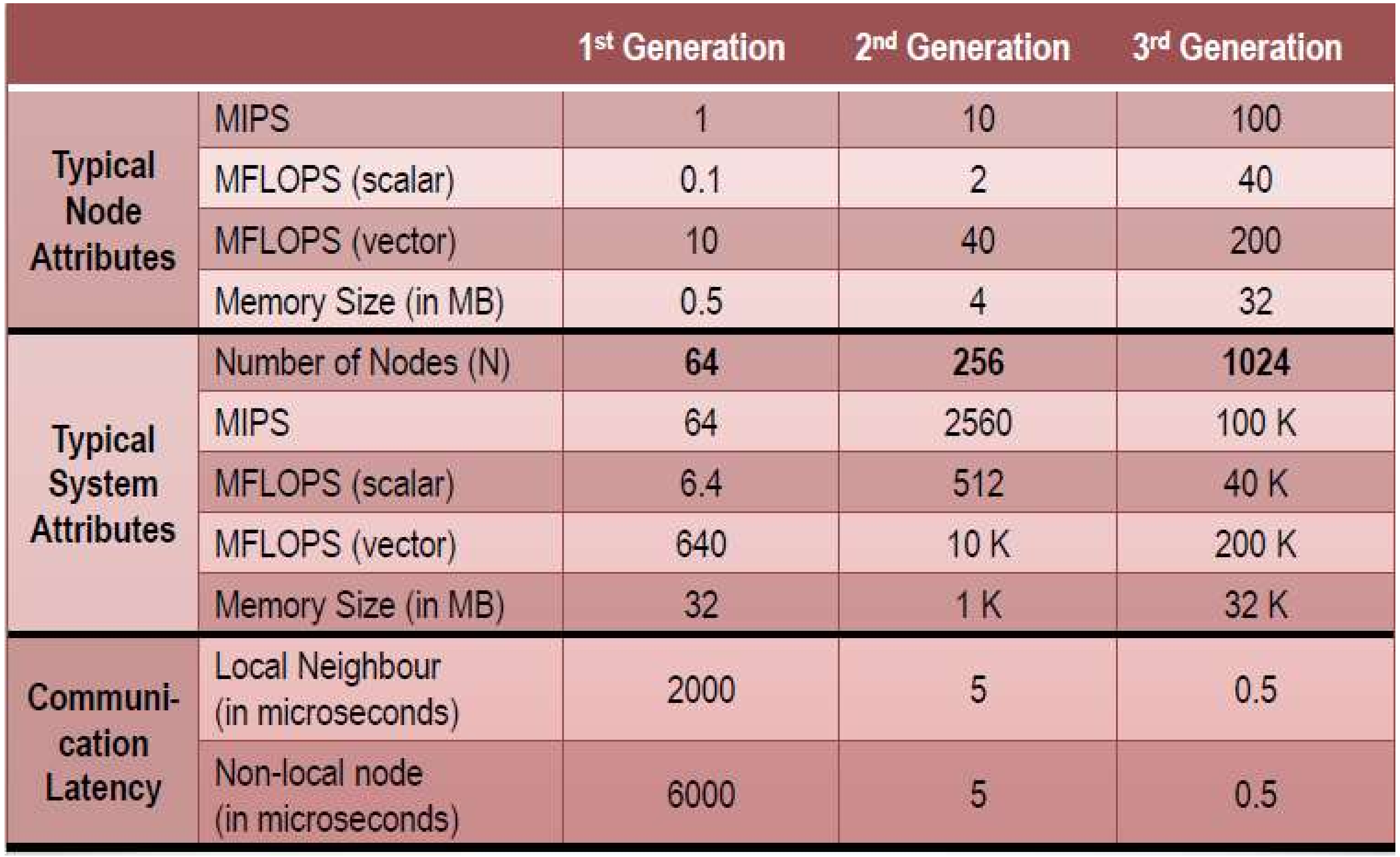
Third Generation

- Example Systems: Caltech’s Mosaic C, J-Machine, Intel Paragon

First and second generation multi-computers are regarded as **medium-grain** systems

Third generation multi-computers were regarded as **fine-grain** systems.

Fine-grain and shared memory approach can, in theory, combine the relative merits of multiprocessors and multi-computers in a **heterogeneous processing** environment



		1st Generation	2nd Generation	3rd Generation
Typical Node Attributes	MIPS	1	10	100
	MFLOPS (scalar)	0.1	2	40
	MFLOPS (vector)	10	40	200
	Memory Size (in MB)	0.5	4	32
Typical System Attributes	Number of Nodes (N)	**64**	**256**	**1024**
	MIPS	64	2560	100 K
	MFLOPS (scalar)	6.4	512	40 K
	MFLOPS (vector)	640	10 K	200 K
	Memory Size (in MB)	32	1 K	32 K
Communi-cation Latency	Local Neighbour (in microseconds)	2000	5	0.5
	Non-local node (in microseconds)	6000	5	0.5

The Intel Paragon System

Previously, homogeneous nodes were used to make hypercube multicomputers, as all the functions were given to the host. So, this limited the I/O bandwidth. Thus to solve large-scale problems efficiently or with high throughput, these computers could not be used.

The Intel Paragon System was designed to overcome this difficulty. It turned the multicomputer into an application server with multiuser access in a network environment.

In late 1992, intel shipped a commercial version of the DELTA, called Paragon. The Paragon uses the same rectangular grid structure as the DELTA, but faster processing nodes.

The Paragon node contains two identical Intel i860XP processors, an improved 50MHz version of the i860 used in previous Intel systems.

This processor has peak rates of 75flops (64-bit) and 42MIPS and can support from 16-128 Mbytes with a 400 Mbytes/sec processor-memory bandwidth and an 800 Mbytes/sec processor-cache bandwidth.

Paragon nodes are organized into three partitions: The Compute partition, the Service Partition and the I/O partition.

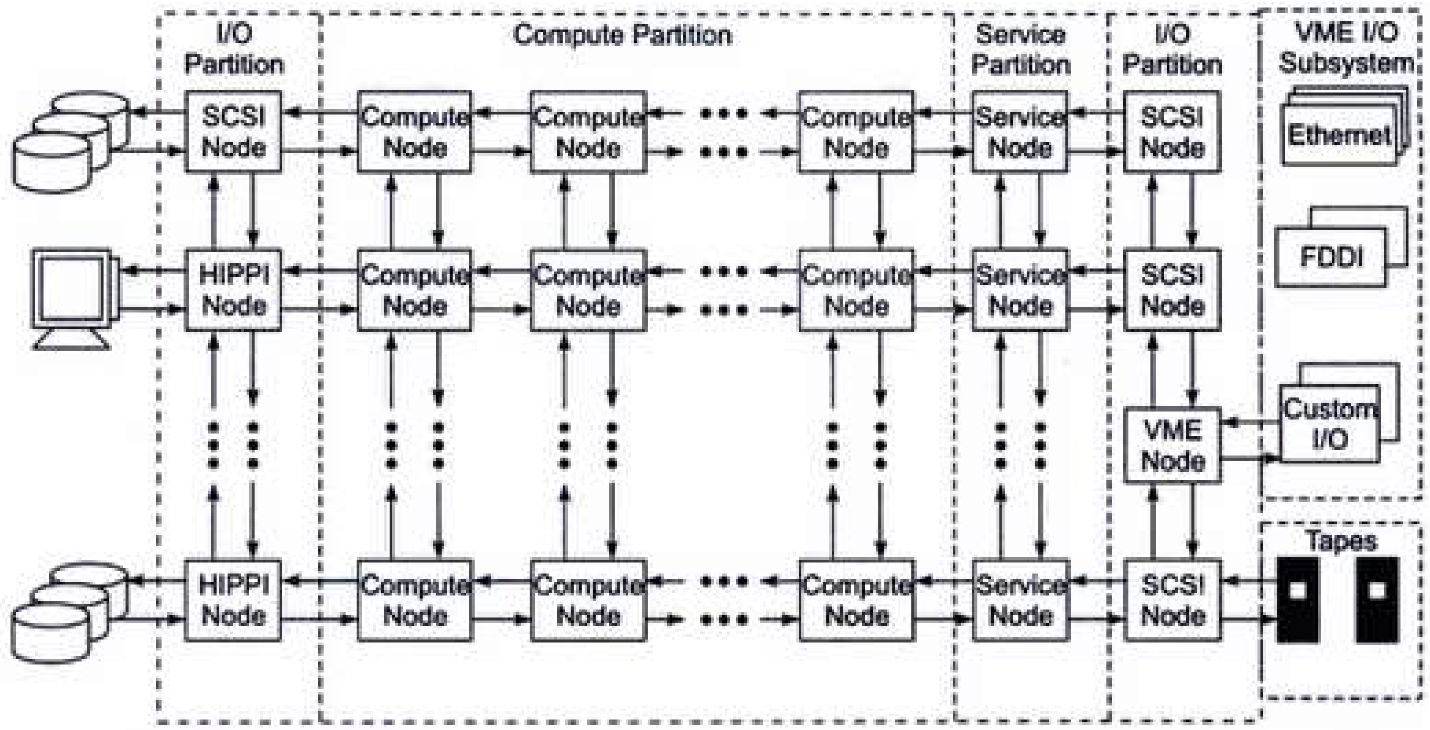
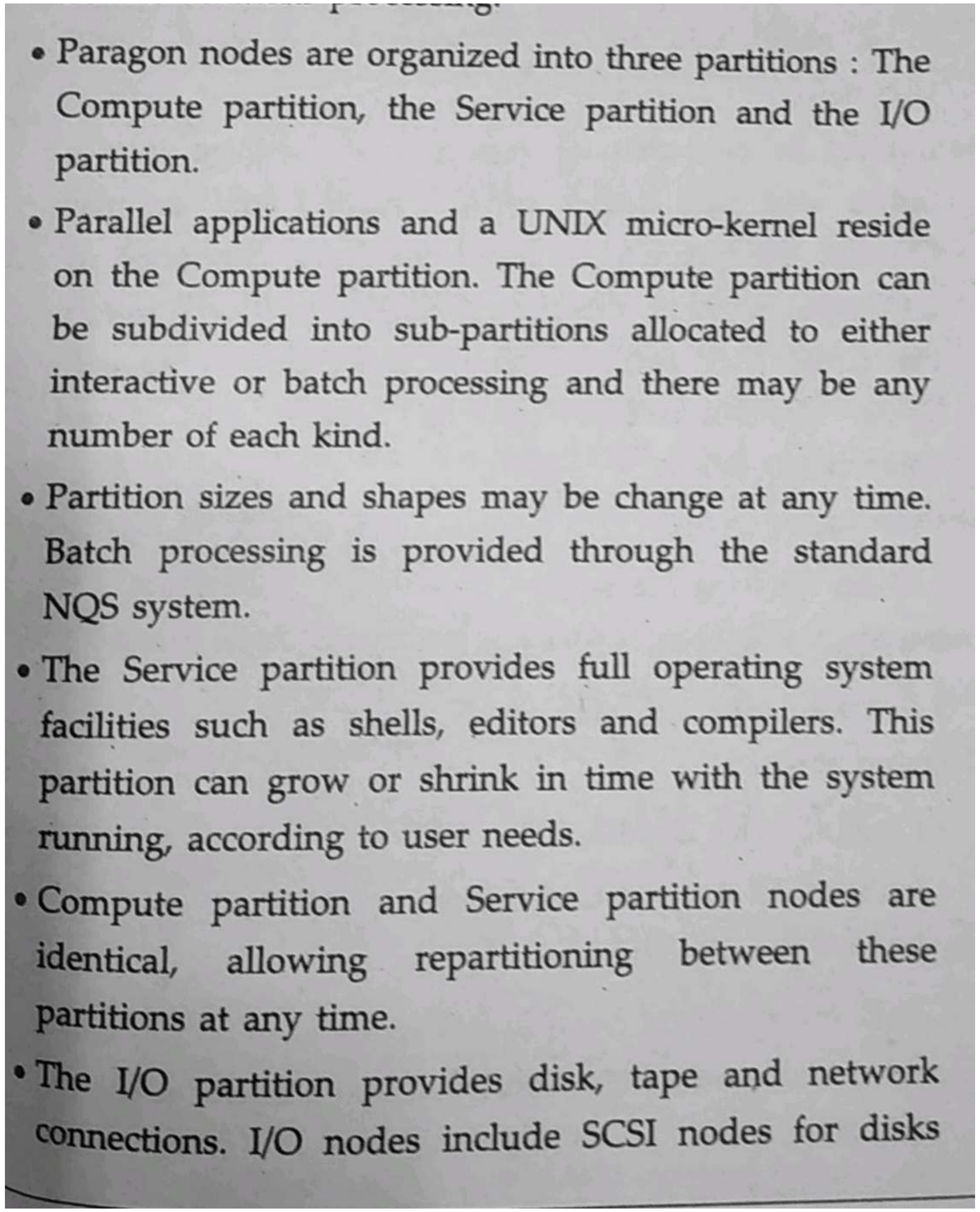
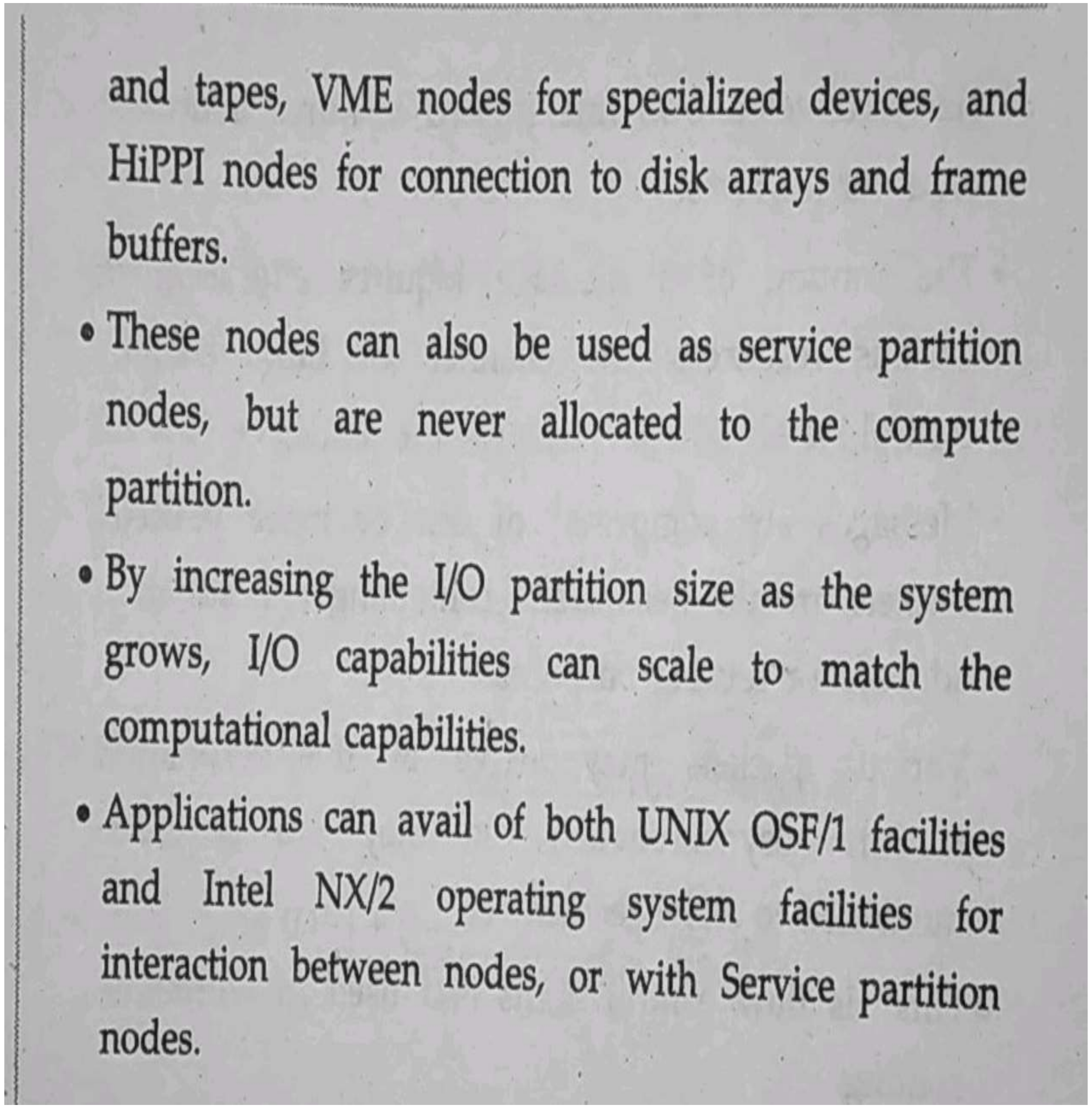


Fig. 7.23 The Intel Paragon system architecture (Courtesy of Intel Supercomputer Systems Division, 1991)



- Paragon nodes are organized into three partitions : The Compute partition, the Service partition and the I/O partition.
- Parallel applications and a UNIX micro-kernel reside on the Compute partition. The Compute partition can be subdivided into sub-partitions allocated to either interactive or batch processing and there may be any number of each kind.
- Partition sizes and shapes may be change at any time. Batch processing is provided through the standard NQS system.
- The Service partition provides full operating system facilities such as shells, editors and compilers. This partition can grow or shrink in time with the system running, according to user needs.
- Compute partition and Service partition nodes are identical, allowing repartitioning between these partitions at any time.
- The I/O partition provides disk, tape and network connections. I/O nodes include SCSI nodes for disks


and tapes, VME nodes for specialized devices, and HiPPI nodes for connection to disk arrays and frame buffers.

- These nodes can also be used as service partition nodes, but are never allocated to the compute partition.
- By increasing the I/O partition size as the system grows, I/O capabilities can scale to match the computational capabilities.
- Applications can avail of both UNIX OSF/1 facilities and Intel NX/2 operating system facilities for interaction between nodes, or with Service partition nodes.

7.4 MESSAGE PASSING MECHANISMS

Message Passing in Multicomputers

Multicomputers have no shared memory, and each “computer” consists of a single processor, cache, private memory, and I/O devices.

Some “network” must be provided to allow the multiple computers to communicate.

The communication between computers in a multicomputer is called “message passing.”

Message Formats

Messages may be fixed or variable length.

Messages are comprised of one or more packets.

Packets are the basic units containing a destination address (e.g. processor number) for routing purposes.

Different packets may arrive at the destination asynchronously, so they are sequence numbered to allow reassembly.

Flits (flow control digits) are used in wormhole routing; they’re discussed a bit later ☺

Store and Forward Routing

Packets are the basic unit in the store and forward scheme.

An intermediate node must receive a complete packet before it can be forwarded to the next node or the final destination, and only then if the output channel is free and the next node has available buffer space for the packet.

The latency in store and format networks is directly related to the number of intermediate nodes through which the packet must pass.

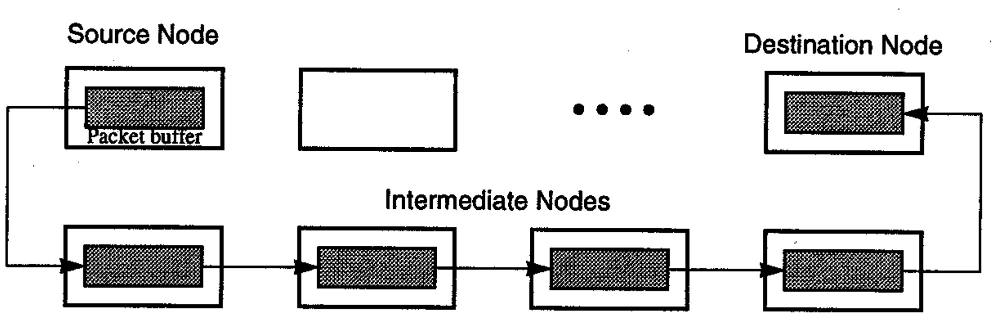
Flits and Wormhole Routing

Wormhole routing divides a packet into smaller fixed-sized pieces called *flits* (flow control digits).

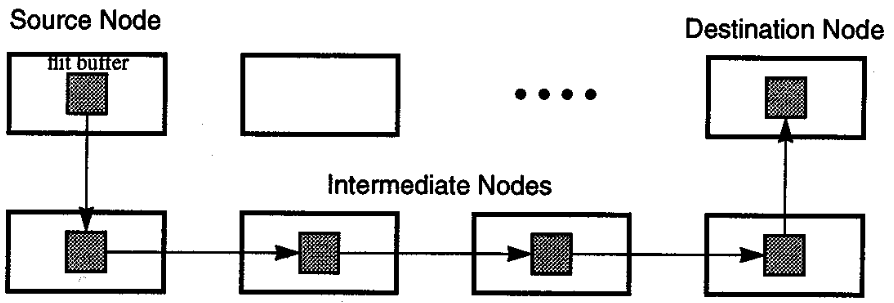
The first flit in the packet must contain (at least) the destination address. Thus the size of a flit must be at least $\log_2 N$ in an N-processor multicomputer.

Each flit is transmitted as a separate entity, but all flits belonging to a single packet must be transmitted in sequence, one immediately after the other, in a pipeline through intermediate routers.

Store and Forward vs. Wormhole



(a) Store-and-forward routing using packet buffers in successive nodes



(b) Wormhole routing using flit buffers in successive routers

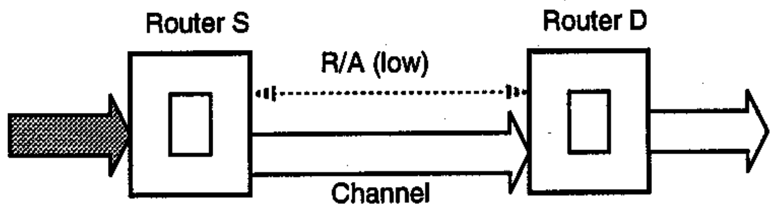
Asynchronous Pipelining

Each intermediate node in a wormhole network, and the source and destination, each have a buffer capable of storing a flit.

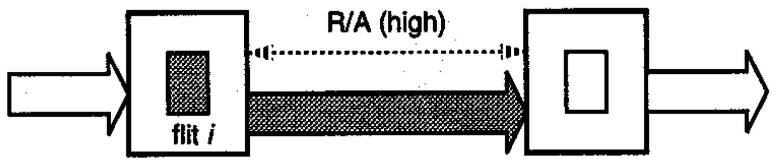
Adjacent nodes communicate requests and acknowledgements using a one-bit *ready/request* (R/A) line.

- When a receiver is ready, it pulls the R/A line low.
- When the sender is ready, it raises the R/A line high and transmits the next flit; the line is left high.
- After the receiver deals with the flit (perhaps sending it on to another node), it lowers the R/A line to indicate it is ready to accept another flit.
- The cycle repeats for transmission of other flits.

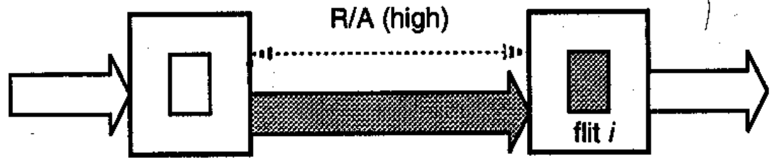
Wormhole Node Handshaking



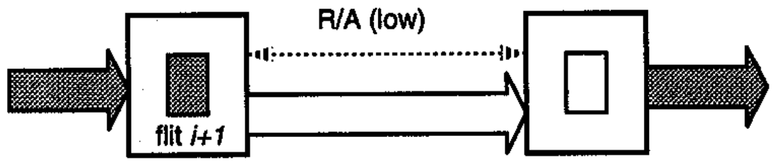
(a) D is ready to receive a flit



(b) S is ready to send flit i



(c) Flit i is received by D



(d) Flit i is removed from D's buffer and flit $i+1$ arrives at S's buffer

Asynchronous Pipeline Speeds

An asynchronous pipeline can be very efficient, and use a clock speed higher than that used in a synchronous pipeline.

The pipeline can be stalled if buffers or successive channels in the path are not available during certain cycles.

A packet could be “buffered, blocked, dragged, detoured” – and just knocked around, in general – if the pipeline stalls.

Latency

Assume

- D = # of intermediate nodes (routers) between the source and destination
- L = packet length (in bits)
- F = flit length (in bits)
- W = the channel bandwidth (in bits/sec)

Ignoring network startup time, propagation and resource delays:

- store and forward latency is $L/W \times (D+1)$, and
- wormhole latency is $L/W + F/W \times D$.

F is usually much smaller than L, and thus D has no significant effect on latency in wormhole systems.

7.4.2 Virtual Channels

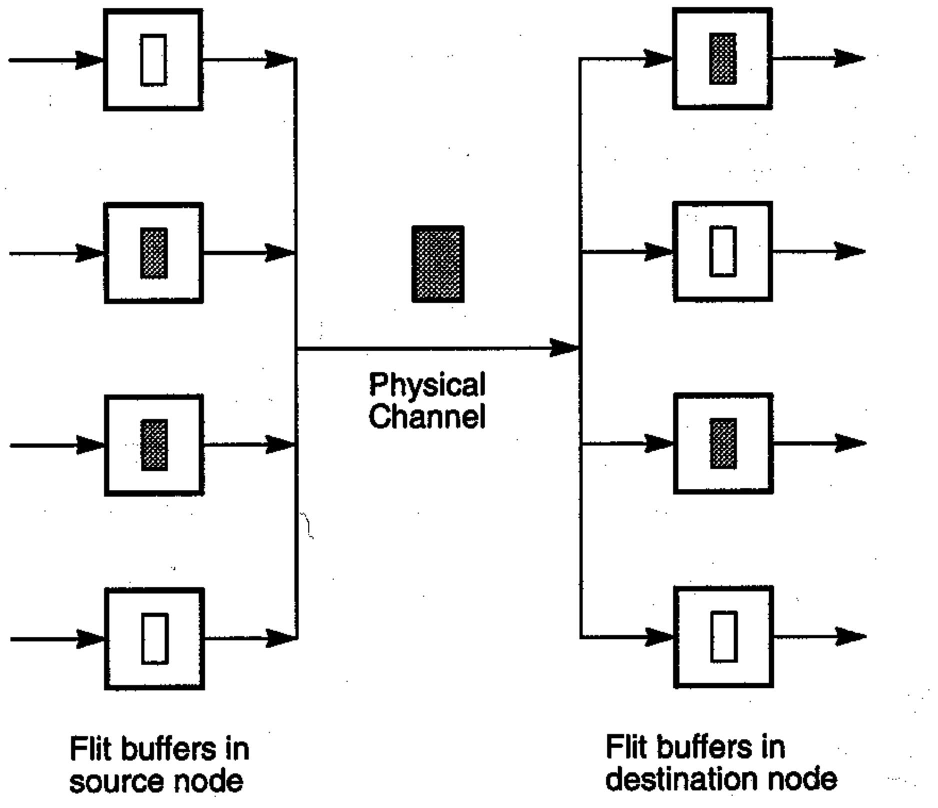
The channels between nodes in a wormhole-routed multicomputer are shared by many possible source and destination pairs.

A “virtual channel” is a pair of flit buffers (in nodes) connected by a shared physical channel.

The physical channel is “time shared” by all the virtual channels.

Other resources (including the R/A line) must be replicated for each of the virtual channels.

Virtual Channel Example

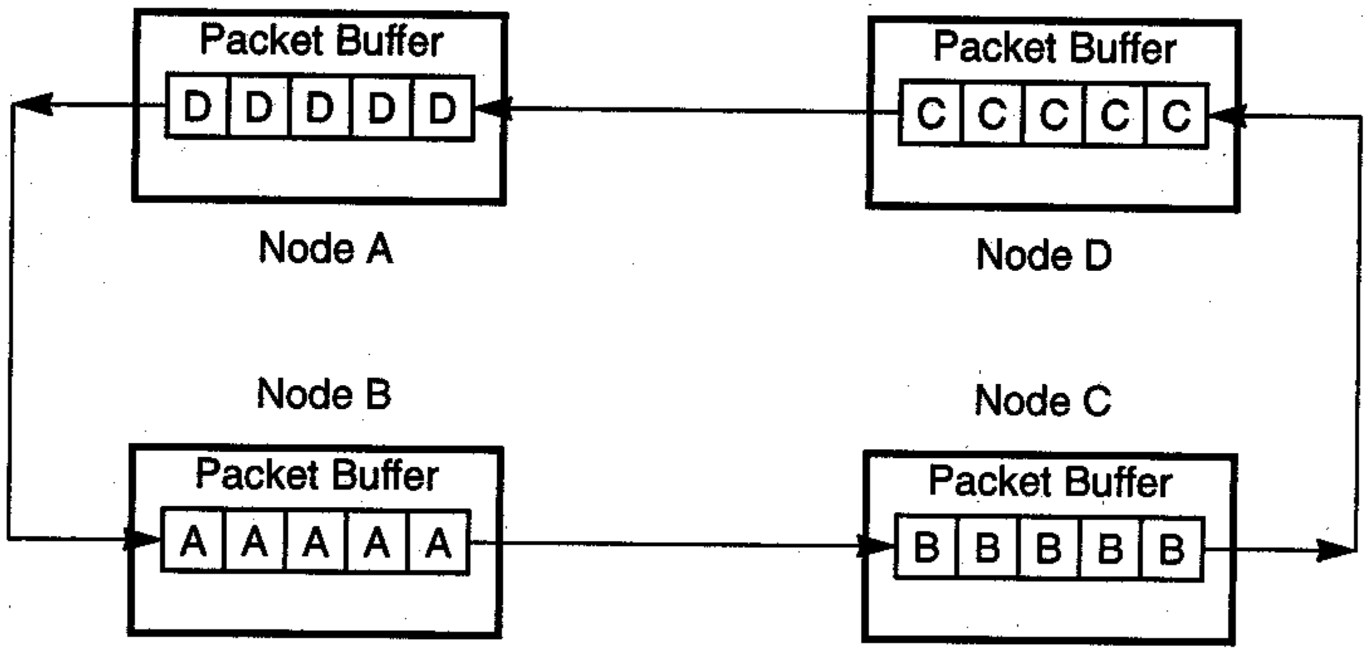


Deadlock

Deadlock can occur if it is impossible for any messages to move (without discarding one).

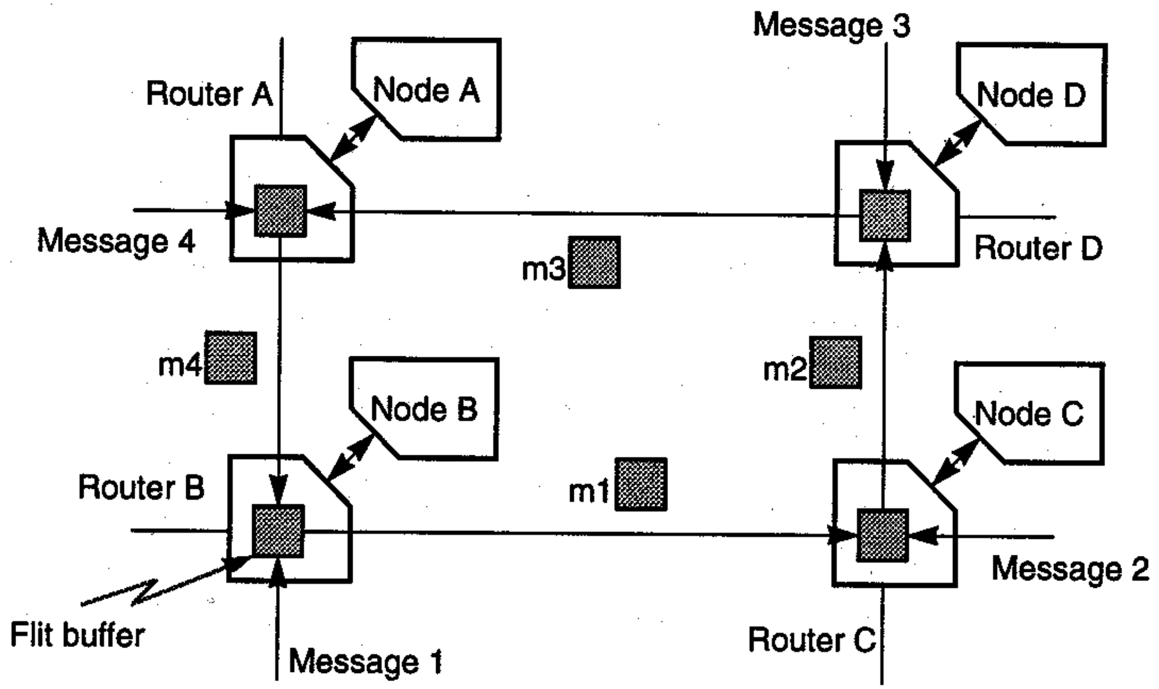
- Buffer deadlock occurs when all buffers are full in a store and forward network. This leads to a circular wait condition, each node waiting for space to receive the next message.
- Channel deadlock is similar, but will result if all channels around a circular path in a wormhole-based network are busy (recall that each “node” has a single buffer used for both input and output).

Buffer Deadlock in a Store and Forward Network



(a) Buffer deadlock among four nodes with store-and-forward routing

Channel Deadlock with Wormhole Routing



(b) Channel deadlock among four nodes with wormhole routing; shaded boxes are flit buffers

Deadlock Avoidance

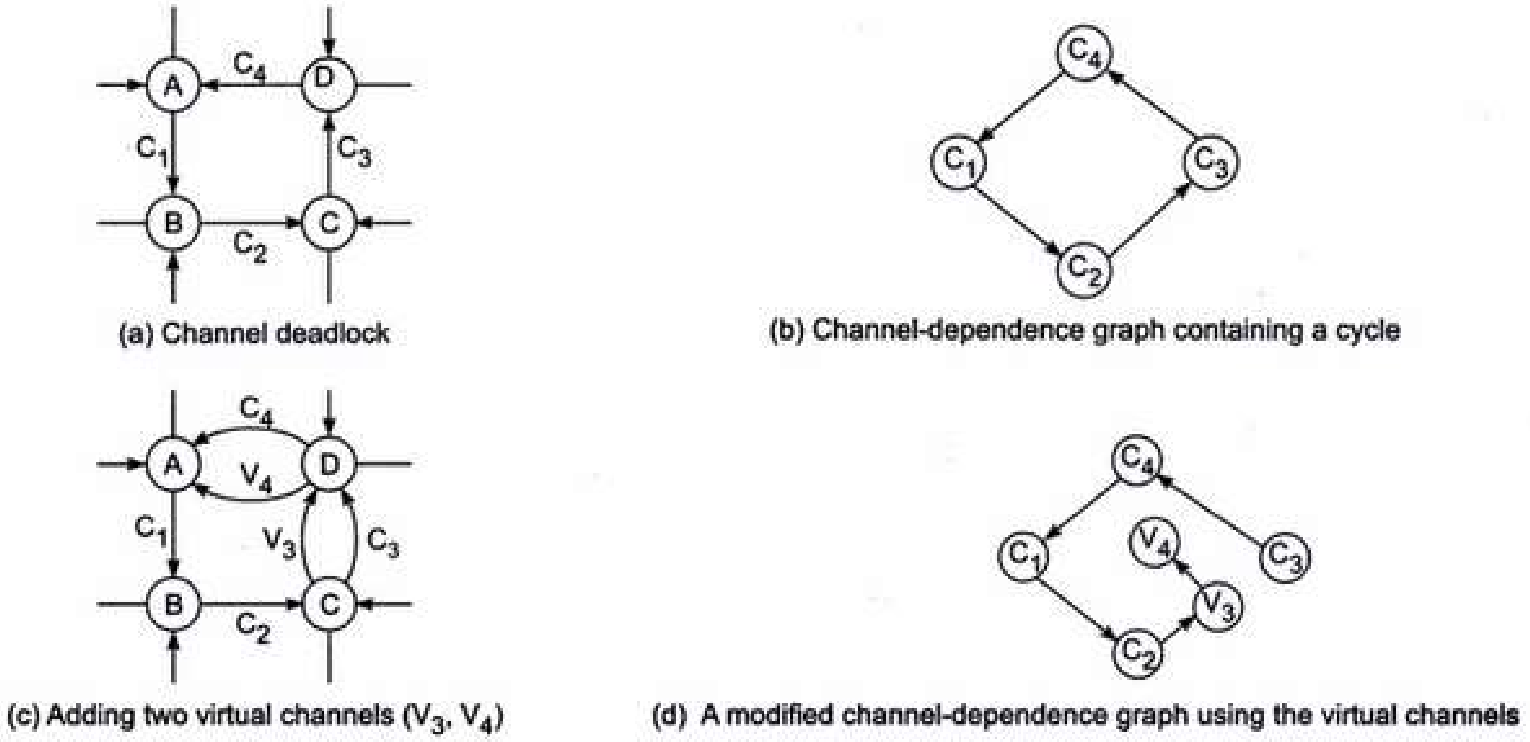

A
B
C
D
C_1
C_2
C_3
C_4
(a) Channel deadlock
(b) Channel-dependence graph containing a cycle
V_3
V_4
(c) Adding two virtual channels (V_3, V_4)
(d) A modified channel-dependence graph using the virtual channels

Fig. 7.32 Deadlock avoidance using virtual channels to convert a cycle to a spiral on a channel-dependence graph

7.4.3 Flow Control

If multiple packets/flits demand the same resources at a given node, then there must be some policy indicating how the conflict is to be resolved.

These policies then determine what mechanisms can be used to deal with congestion and deadlock.

Packet Collision Resolution

Consider the case of two flits both wanting to use the same channel or the same receive buffer at the same time.

How is the "collision" resolved? Who gets the resource? What happens to the other flit?

Virtual Cut-Through Routing

Solution: temporarily store one of the packets in a different buffer.

Positive:

- No messages lost
- Should perform as well as wormhole with no conflicts

Negative:

- Potentially large buffer required (with potentially large delays).
- Not suitable for routers.
- Cycles must be avoided

Blocking

Solution: prevent one of the messages from advancing while the other uses the buffer/channel.

Positive:

- Messages are not lost.

Negative

- Node sending blocked packet is idled.

Discarding

Solution: drop one of the messages in contention for the buffer/channel.

Positive:

- Simple to implement

Negative:

- Loses messages, resulting in a severe waste of resources.

Detour

Solution: send the conflicting message somewhere (anywhere) else.

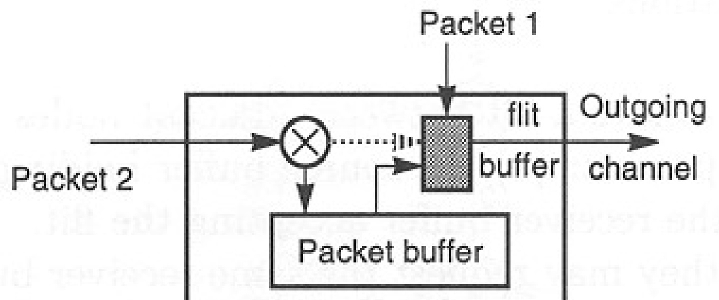
Positive:

- Simple to implement

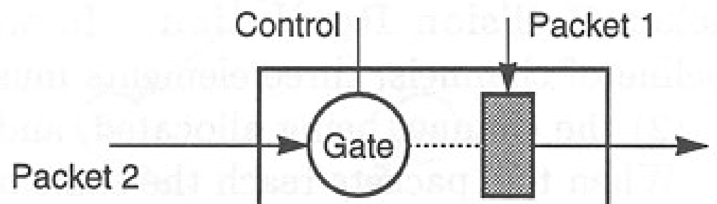
Negative:

- May waste more channel resource than necessary
- May cause other resources to be idled
- May cause livelock (e.g. four dining philosophers, with two seated across from each other conspiring to starve the other two).

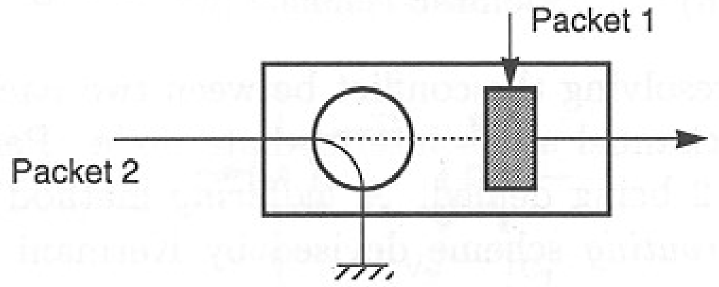
Collision Resolution Techniques



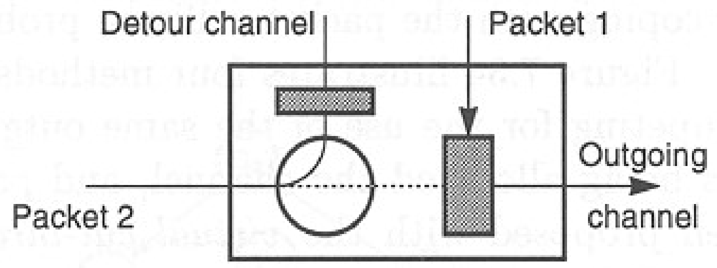
(a) Buffering in virtual cut-through routing



(b) Blocking flow control



(c) Discard and retransmission



(d) Detour after being blocked

Routing

Deterministic routing: the path from source to destination is determined uniquely from the source and destination addresses.

Adaptive routing: the path may depend on network conditions.

Deterministic Routing Using Dimension Ordering

Dimension ordering algorithms are based on the selection of a sequence of channels following a specified order.

For example, routing in a two-dimensional mesh is called X-Y routing, because the X-dimension routing path is decided before choosing the Y-dimension path.

In hypercubes, the example algorithm is called E-cube routing, and again specifies the sequence of channels to be used.

E-cube Routing on a Hypercube

Assume the system has $N = 2^n$ nodes; the dimensions of the hypercube are numbered 1, 2, …, *n*.

Each node has a binary address with *n* bits (numbered *n*-1 to 0). The ith bit in a node address corresponds to the ith dimension.

Source address = *s*, destination address = *d*.

Algorithm:

- Compute direction bit $r_i = s_{i-1}$ xor d_{i-1} for all dimensions. Now set $i = 1$ and $v = s$.
- Route from the current node v to the next node v xor 2^{i-1} if $r_i = 1$; skip this step if $r_i = 0$.
- Move to dimension i + 1 (i.e. $i \leftarrow i + 1$). If $i <= n$, go to the previous step.

E-cube Routing Example

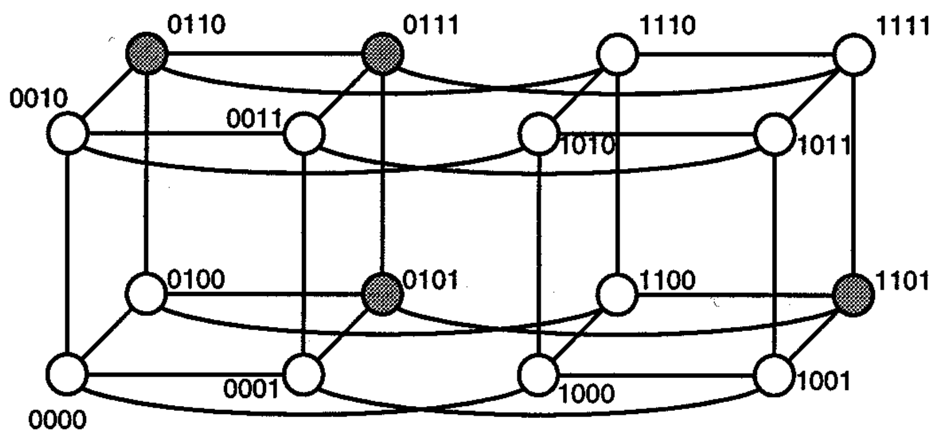
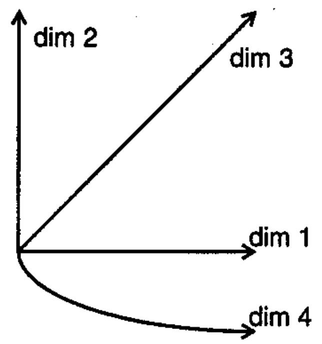


Figure 7.35 E-cube routing on a hypercube computer with 16 nodes.



Source: s=0110

Destination: d=1101

Route:
0110→0111→0101→1101

E-Cube Routing Example (Detail)

Source Address s = 0110, n = 4 (dimension of cube)

Destination Address d = 1101

“Direction Bits” r = 0110 xor 1101 = 1011

Route from 0110 to 011$\underline{1}$ because r = 101$\underline{1}$

Route from 0111 to 01$\underline{0}$1 because r = 10$\underline{1}$1

Skip dimension 3 because r = 1$\underline{0}$11

Route from 0101 to 1101 because r = $\underline{1}$011

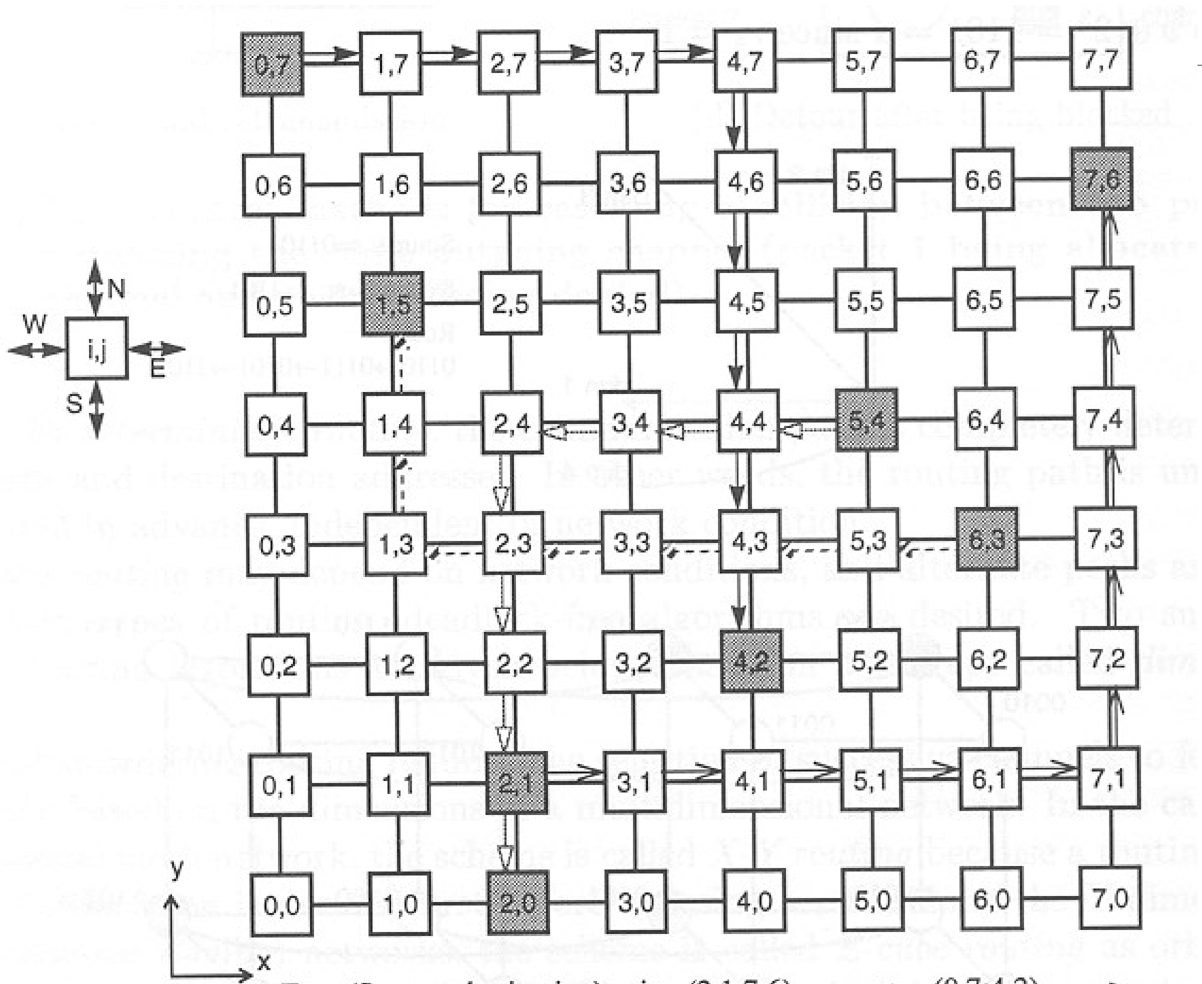
X-Y Routing on a 2-D Mesh

X-Y routing is similar, in concept, to E-cube routing in that the route from the source to the destination is determined completely from their addresses.

In X-Y routing, the message travels “horizontally” (in the X-dimension) from the source node to the “column” containing the destination, where the message travels vertically.

There are four possible direction pairs, east-north, east-south, west-north, and west-south.

X-Y Routing Example



Four (Source; destination) pairs: (2,1;7,6) ⟶ (0,7;4,2) ⟶

(5,4; 2,0) ⟶ (6,3; 1,5) - - - ⟶

Dimension Ordering Characteristics

In general, X-Y routing can be expanded to an n-dimensional mesh.

Both X-Y routing and E-cube routing can be shown to be deadlock free. (Hint: compare with Havender’s “Standard Allocation Pattern” for resource use in an OS.)

Both techniques can be used with store-and-forward or wormhole routing networks to produce minimal routes.

Dimension ordering does not work on a torus.

Adaptive Routing

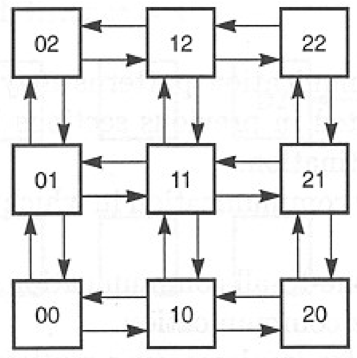
The main purpose of adaptive routing is to avoid deadlock.

Adaptive routing makes use of virtual channels between nodes to make routing more economical and feasible to implement.

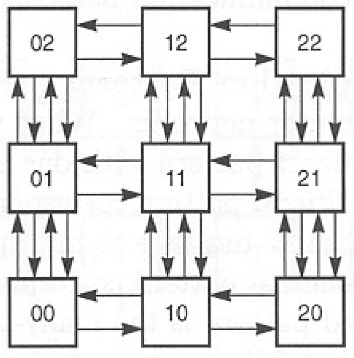
Virtual channels allow the network to exhibit different characteristics at different times (that is, it “adapts”).

For example, (c) and (d) on the next slide are adaptive configurations of (a), but they prevent deadlock from occurring, since they allow only west-{north/south} routing (in c), or east-{north/south} routing (in d).

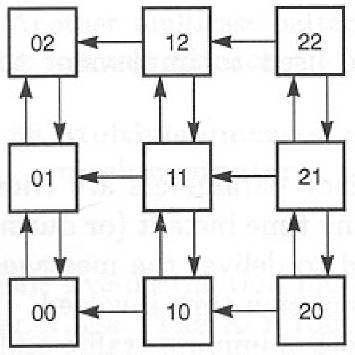
Adaptive Use of Virtual Channels to Avoid Deadlock



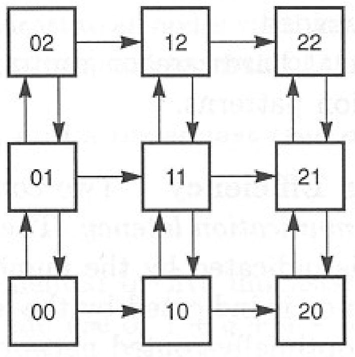
(a) Original mesh without virtual channel



(b) Two pairs of virtual channels in Y-dimension



(c) For a westbound message



(d) For an eastbound message

Communication Patterns

Four possible patterns

- Unicast – traditional one to one communication
- Multicast – one to many communication, with one message sent to multiple destinations
- Broadcast – one to all communication, with one message sent to every possible destination
- Conference – many to many communication

Note that each of these can be implemented using simple sequential transmission of messages (unicast).

Efficiency Parameters

Two common efficiency parameters are:

- channel traffic – the number of channels used at any time instant to deliver messages
- communication latency – the longest time required for any packet to reach its destination

An optimal network would minimize both of these parameters for the communication patterns it uses.

However, these efficiency parameters are interrelated, and achieving minimums in each may not be possible.

Latency is more important than traffic in a store-and-forward network.

Traffic demand is more important than latency in a wormhole-routed network.

Example 5-Destination Multicast

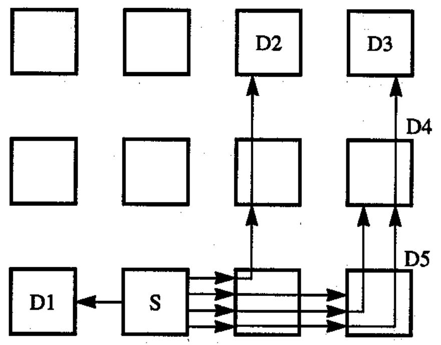
(a) Five unicasts, with traffic demand = 13 and latency = 4 (assuming one “hop” per unit time).

(b) Tree multicast with branching at multiple levels, with traffic demand = 7 and latency = 4.

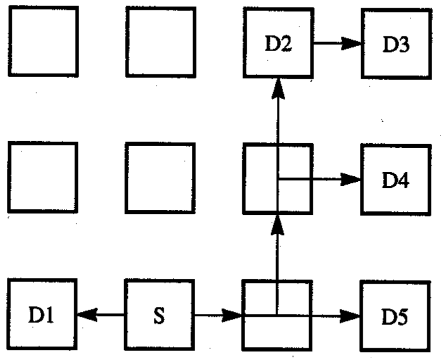
(c) Tree multicast with only one branching node, with traffic demand = 6 and latency = 5.

(d) Broadcast to all nodes with spanning tree.

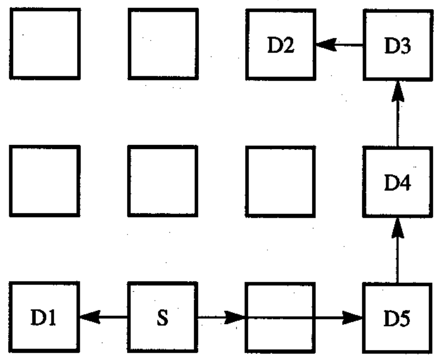
Multicast & Broadcast Patterns



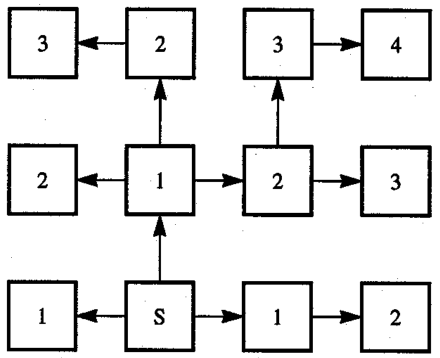
(a) Five unicasts with traffic = 13 and distance = 4



(b) A multicast pattern with traffic = 7 and distance = 4



(c) Another multicast pattern with traffic = 6 and distance = 5



(d) Broadcast to all nodes via a tree (numbers in nodes correspond to levels of the tree)

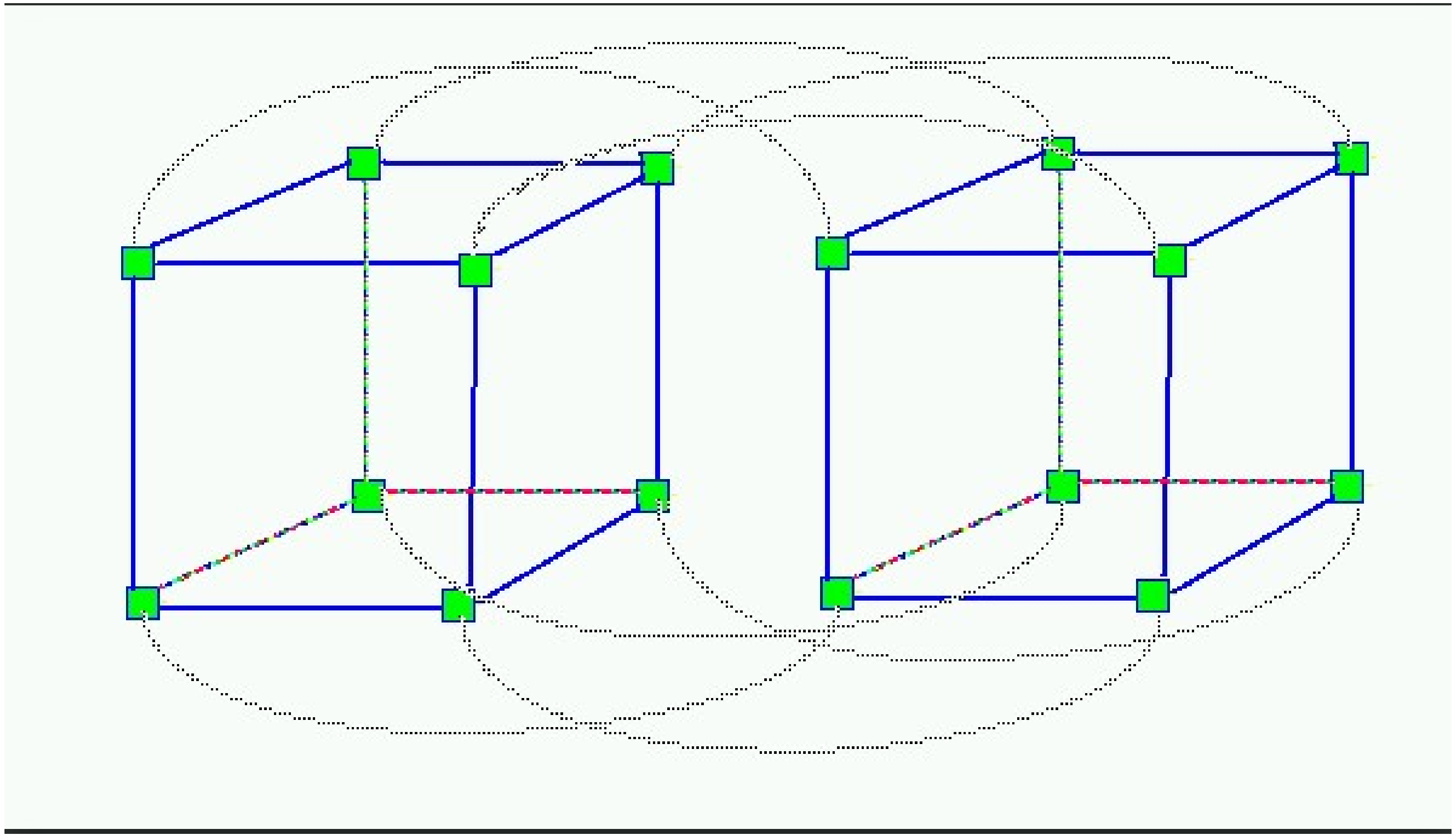
Hypercube Multicast/Broadcast

Broadcast on a hypercube of dimension n will have a latency not exceeding n.

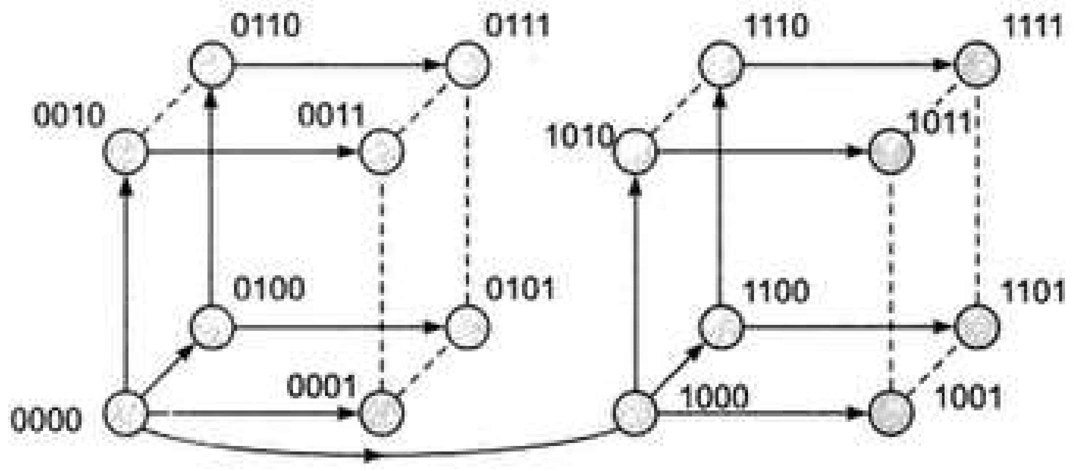
A greedy algorithm for building a tree selects, at each node, the nodes in dimensions that will reach the largest number of remaining destinations (e.g. find the minimm cover set).

In the event of a tie, any of the tied dimensions can be selected (which means the resulting tree is not necessarily unique).

Note that all communication channels at each level of the multicast/broadcast tree must be ready at the same time, or else additional buffering might be required.

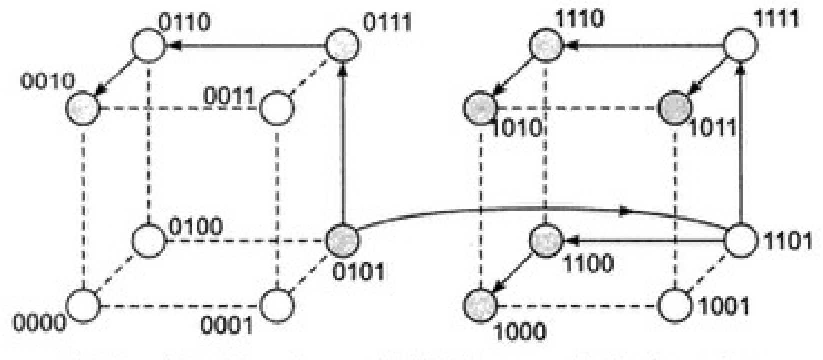


Broadcast on Hypercube



(a) Broadcast tree for a 4-cube rooted at node 0000

Multicast on Hypercube – greedy algorithm



(b) A multicast tree from node 0101 to seven destination nodes 1100, 0111, 1010, 1110, 1011, 1000, and 0010

Sending packet through the dimensions

1st level channel:
0101->0111 & 0101->1101
2nd level channel:
1101->1111, 1101->1100 & 0111->0110
3rd level channel:
1111->1110, 1111->1011, 1100->1000 & 0110->0010
4th level channel:
1110->1010

Virtual Networks

With multiple virtual channels between nodes, it is possible to dynamically reconfigure a network into one of perhaps many different "virtual networks."

The advantages of having many such virtual networks are

- routing needs can be used to tailor networks that yield results with simple and efficient routing algorithms
- deadlock can be completely eliminated (e.g. by not allowing cycles to exist in the virtual network)

Of course, adding channels to the network will increase the cost

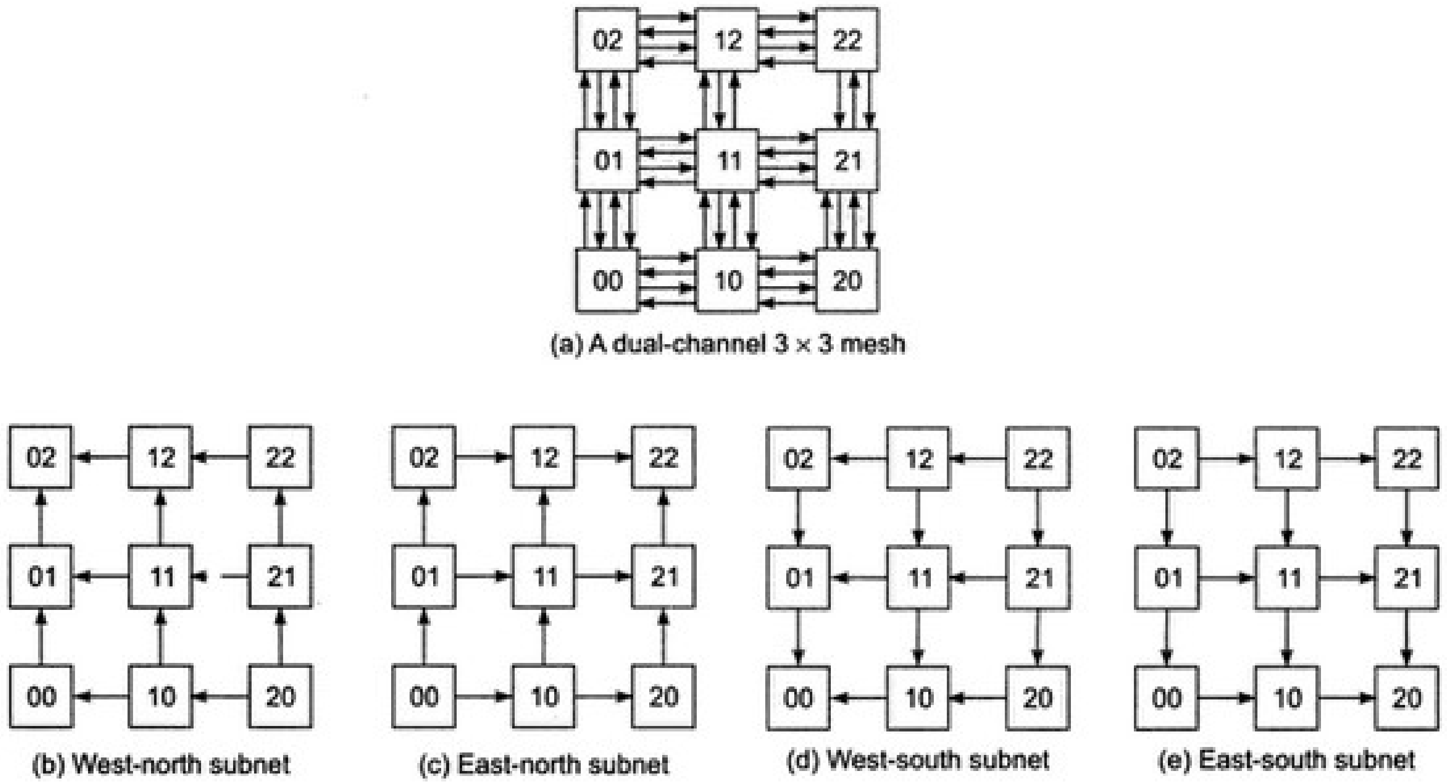


Fig. 7.39 Four virtual networks implementable from a dual-channel mesh

Network Partitioning

Another benefit of having virtual channels between nodes is the ability to dynamically partition a network into multiple subnetworks for multicast communication.

Each subnet can carry a different multicast message at the same time.

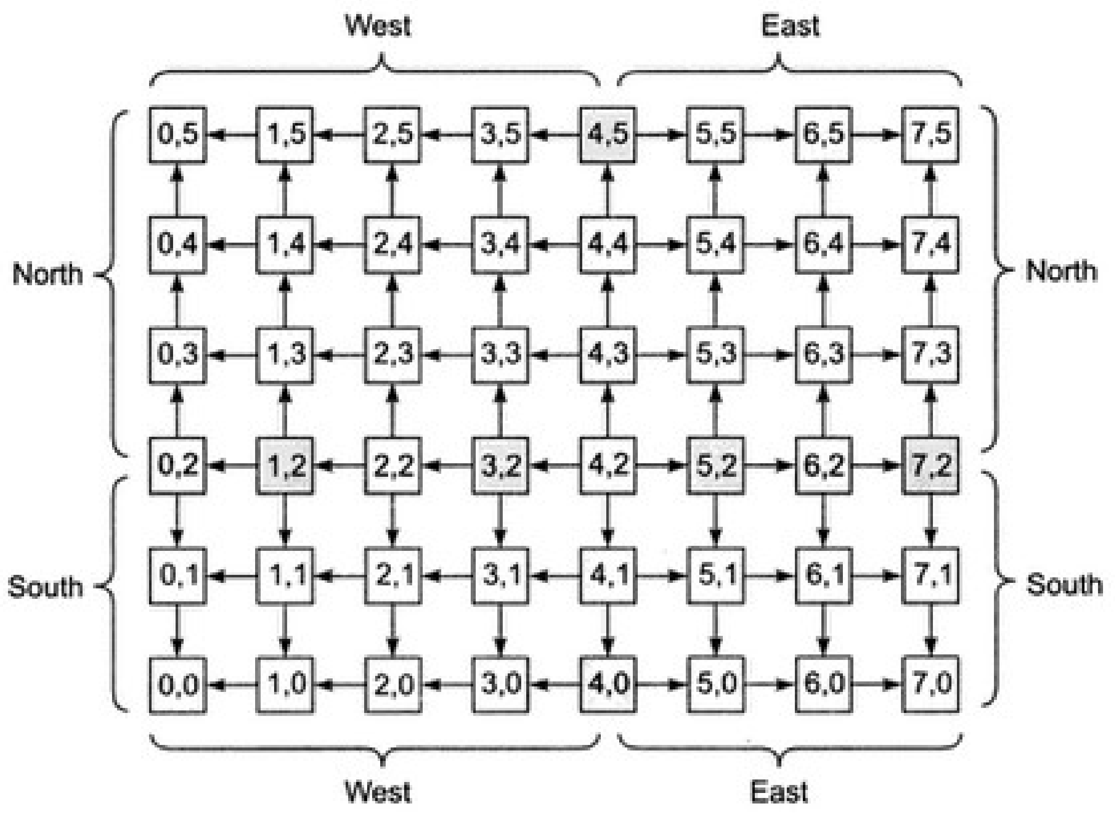


Fig. 7.40 Partitioning of a 6 × 8 mesh into four subnets for a multicast from source node (4,2). Shaded nodes are along the boundary of adjacent subnets (Courtesy of Lin, McKinly, and Ni, 1991)

Parallel Processors

Session 10

Multivector and SIMD Computers

Vector Processing Principles

- Vector:
  - A set of scalar data items
  - All of the same type
  - Stored in memory
- Stride:
  - Address increments between successive elements of a vector
- Vector Processor:
  - Hardware resources to perform vector operations:
    - Vector registers
    - Functional pipelines
    - Processing elements
    - Register counters
- Vector Processing:
  - Arithmetic or logic operations on vectors
- Vectorization:
  - Conversion from scalar code to vector code

Performance

- Vector processing:
  - Faster
  - More efficient
  - Reduced software overhead
    - Loop control
    - Memory access
  - Matches with pipelining mechanism
- Speedup:
  - Vectorization ratio
  - Speed ration between vector and scalar operations
- Costs:
  - Hardware costs
  - Compiler (vectorizing compiler or vectorizer)
  - Programming skills

Vector Instruction Types

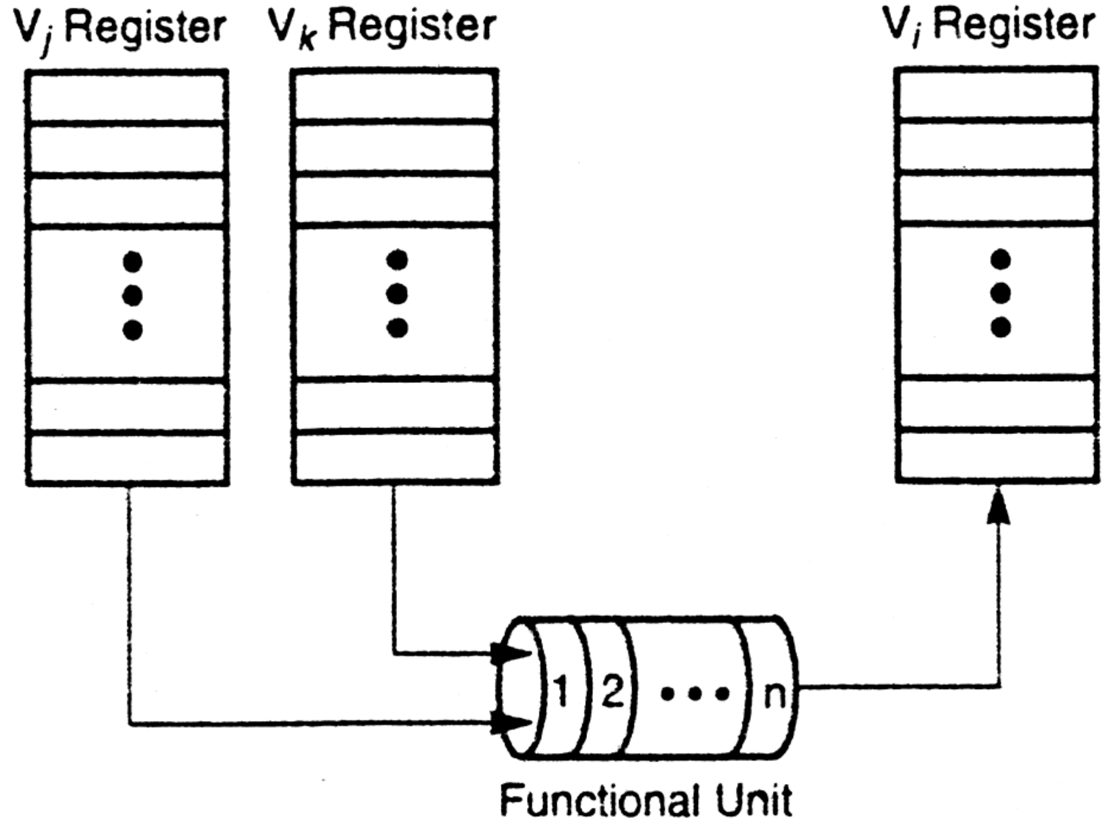
- Vector-Vector Instructions
- Vector-Scalar Instructions
- Vector-Memory Instructions
- Vector Reduction Instructions
- Gather and Scatter Instructions
- Masking Instructions

Vector-Vector Instructions

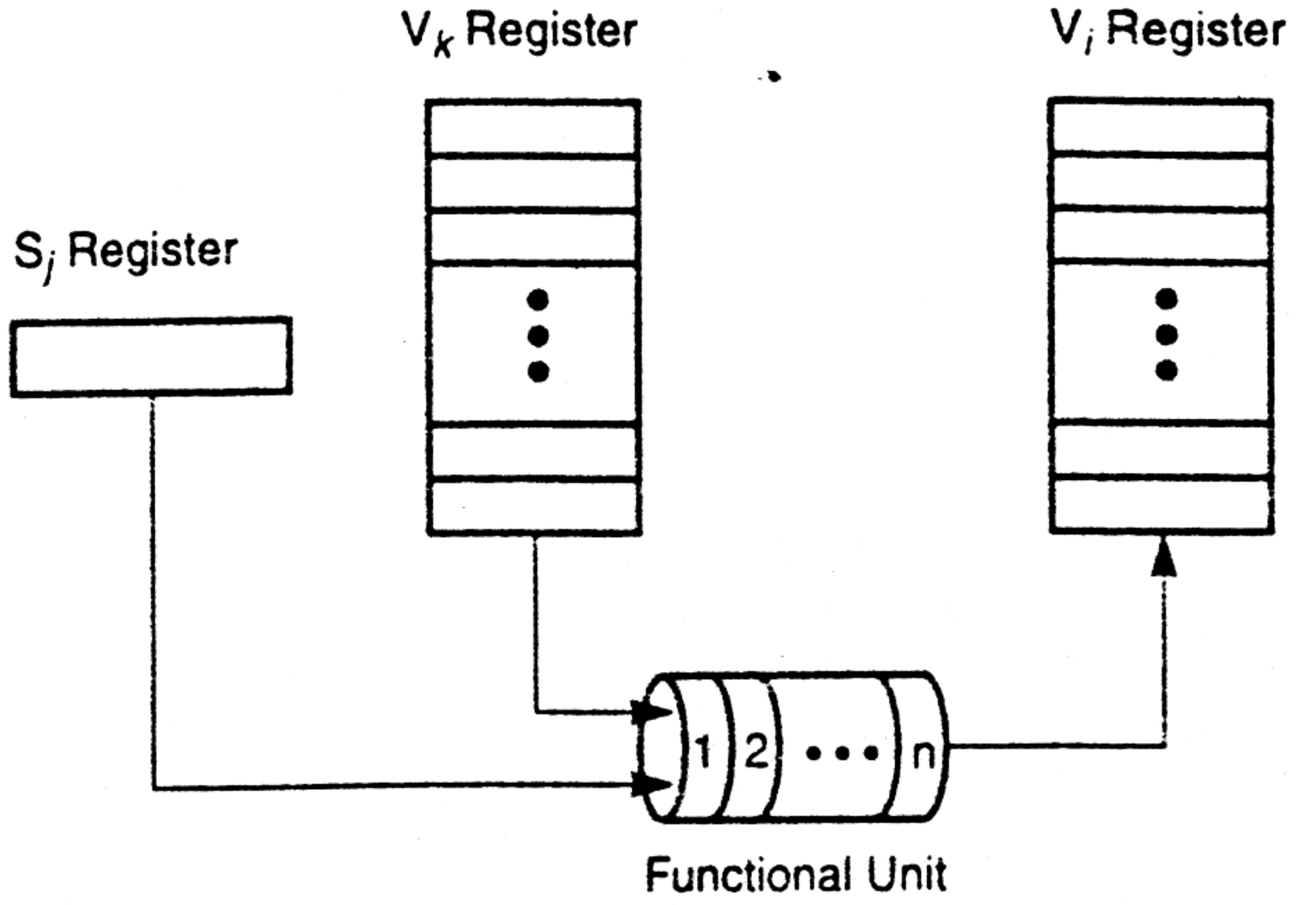
- $V_i \rightarrow V_j$
- $V_j \times V_k \rightarrow V_i$
- Examples:

  $V_1 = \sin(V_2)$

  $V_3 = V_1 + V_2$

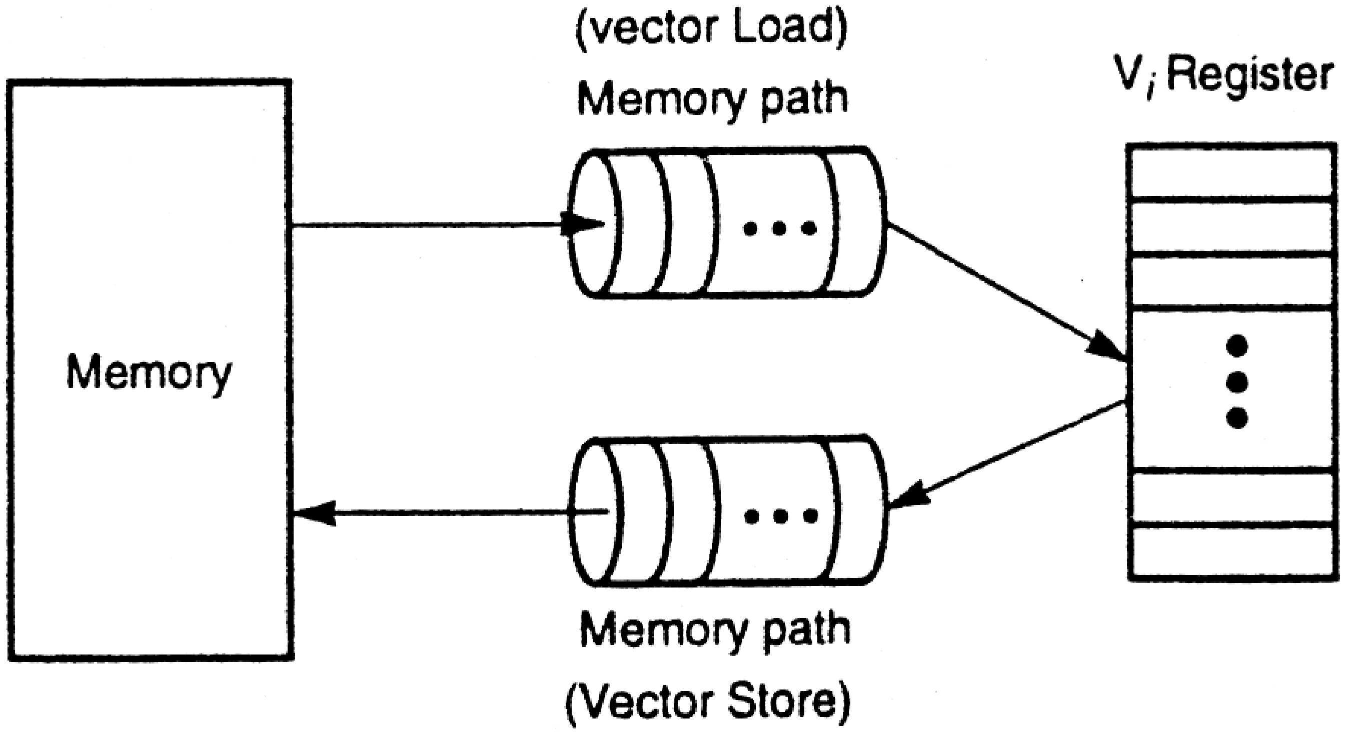


Vector-Scalar Instructions



s x $V_i \rightarrow V_j$

Vector-Memory Instructions



- Vector load:
  - M $\rightarrow$ V_i
- Vector store:
  - V_i $\rightarrow$ M

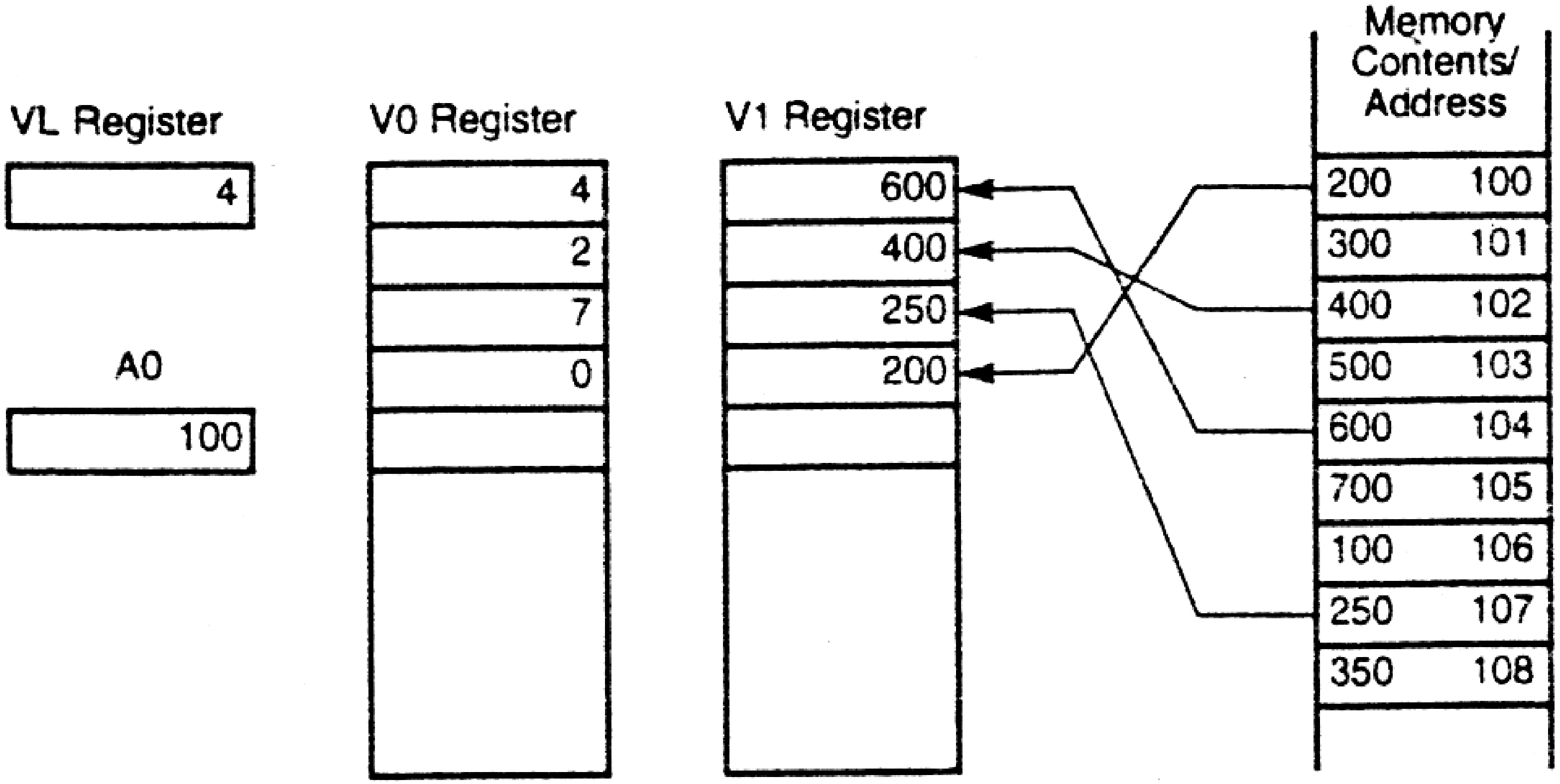
Vector Reduction Instructions

- Mappings:
  - $V_i \rightarrow S_j$
  - $V_i \times V_j \rightarrow S_k$
- Examples:
  - maximum of all elements
  - minimum of all elements
  - sum of all elements
  - mean value of all elements
  - dot product:
    - $s = \Sigma\, a_i \times b_i$ for $A = (a_i)$ and $B = (b_i)$

Gather and Scatter Instructions

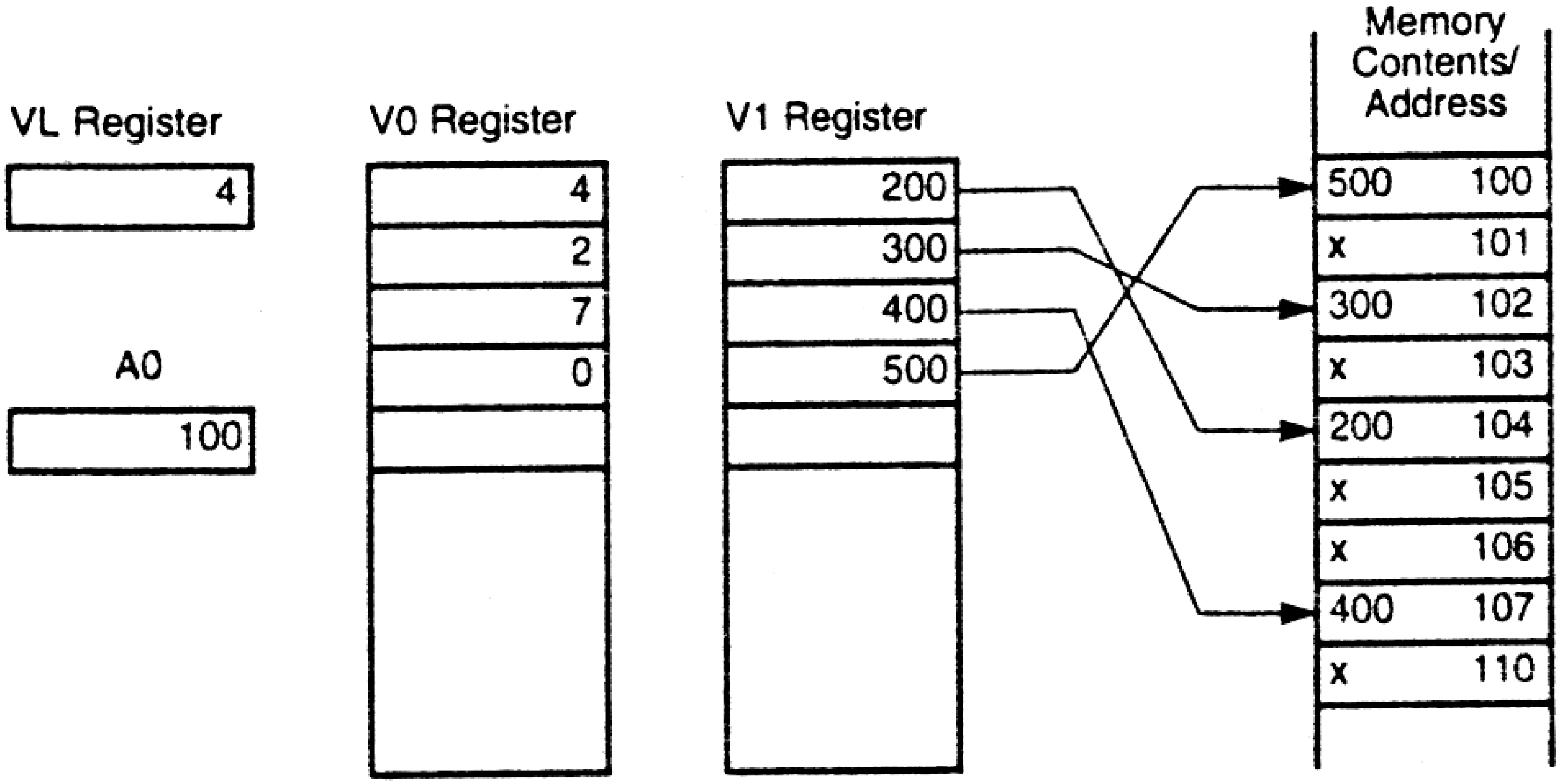
- Gather:
  - $M \rightarrow V_1 \times V_0$
  - V_1 contains the data and V_0 is used as an index
  - Fetches from memory the nonzero elements of a sparse vector using indices that themselves are indexed
- Scatter:
  - $V_1 \times V_0 \rightarrow M$
  - V_1 contains the data and V_0 is used as an index
  - Stores a vector into memory in a sparse vector whose nonzero entries are indexed

Gather



- Gather:
  - M → $V_1 \times V_0$
  - V_1 contains the data and V_0 is used as an index
  - Fetches from memory the nonzero elements of a sparse vector using indices that themselves are indexed

Scatter

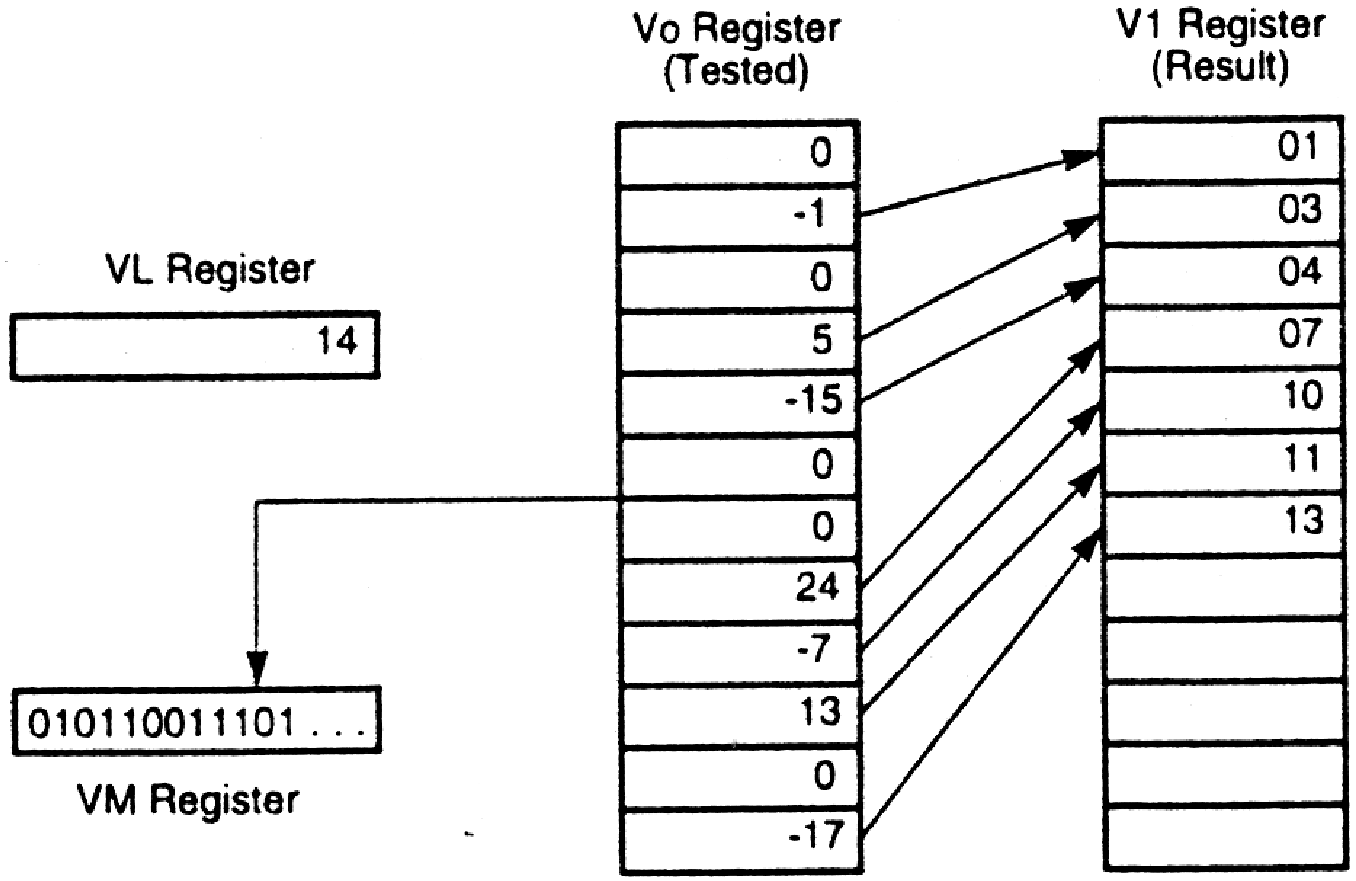


- Scatter:
  - $V_1 \times V_0 \rightarrow M$
  - V_1 contains the data and V_0 is used as an index
  - Stores a vector into memory in a sparse vector whose nonzero entries are indexed

Masking Instructions

- A mask vector is used to:
  - Compress a vector to a shorter index vector
  - Expand a vector to a longer index vector
- Mapping:
  - $V_0 \times V_m \rightarrow V_1$

Masking Instructions

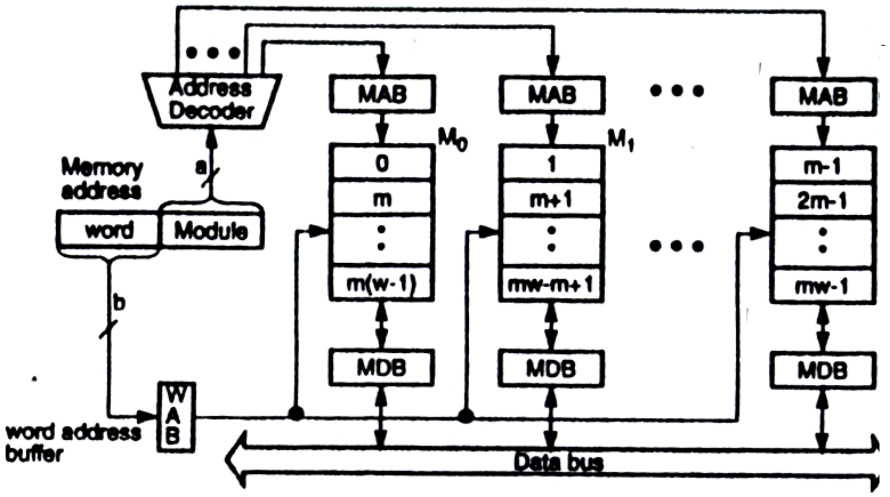


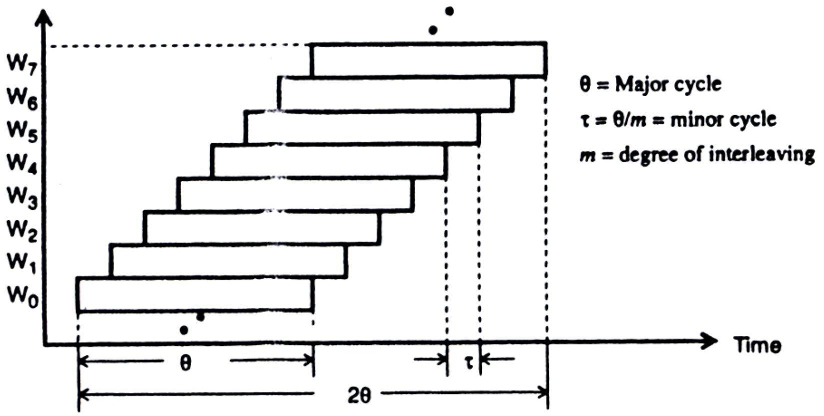
- A mask vector is used to:
  - Compress a vector to a shorter index vector
  - Expand a vector to a longer index vector
- Mapping:
  - $V_0 \times V_m \rightarrow V_1$

Vector Operands and Memory Access

- Arbitrary length
- Arbitrary distribution in memory
  - A matrix is either row major or column major
  - Each row or column can be used as a vector
  - Vector elements are not necessarily in contiguous memory locations
    - Row elements are in contiguous locations with stride n (n is the matrix order)
    - Column elements are in locations with stride n
    - Diagonal elements are in locations with stride n+1
- To access a vector in memory:
  - Base address
  - Stride
  - Length
- Fast vector access necessary to match the pipeline rate
- The access path itself is pipelined: **access pipe**

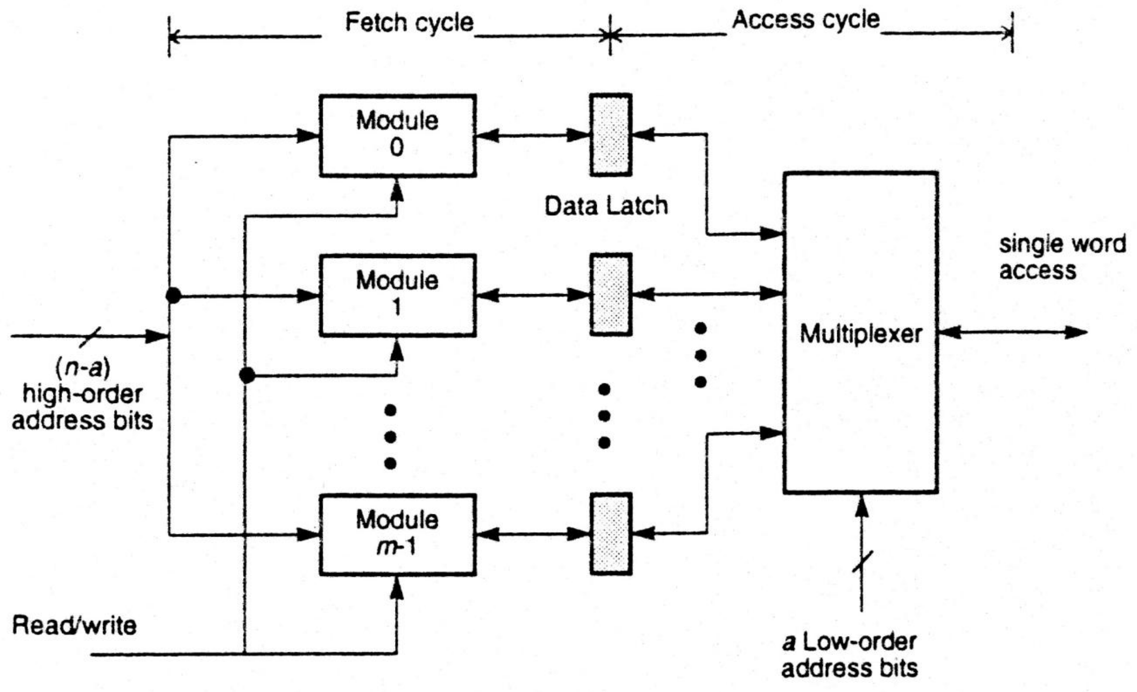
C-Access Memory Organization





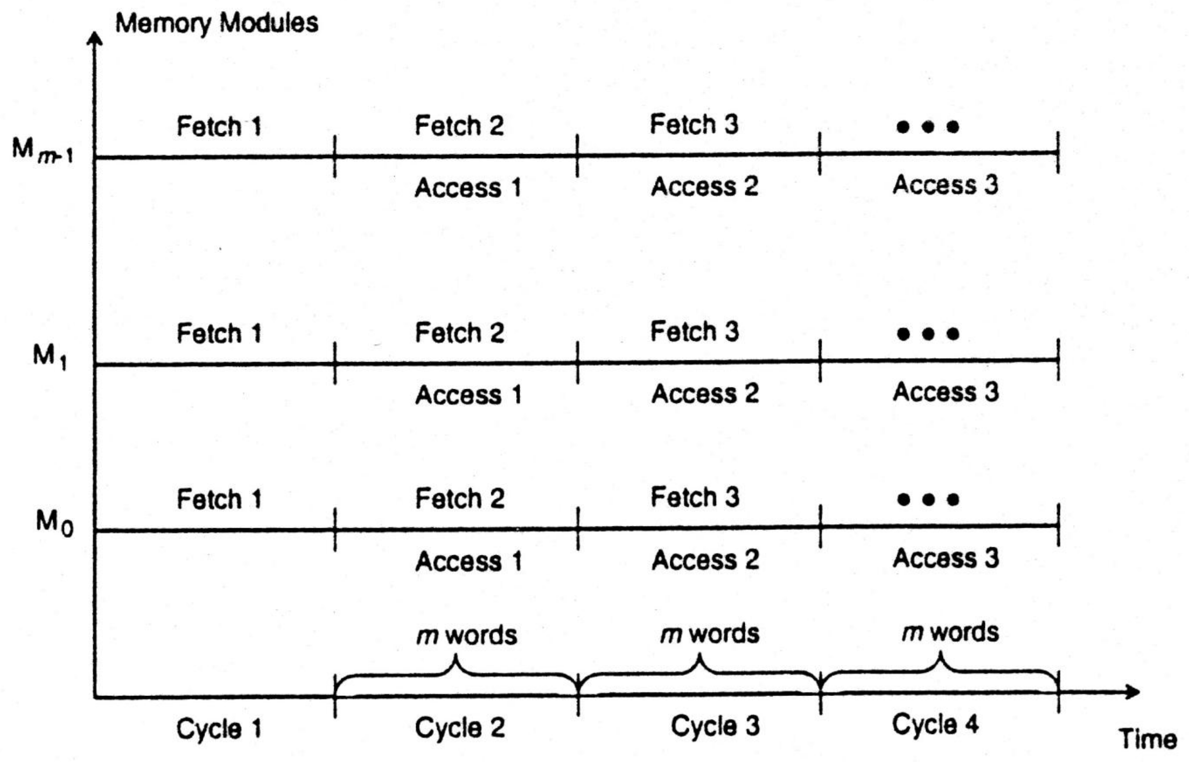
- Vector access scheme from interleaved memory modules
- m-way low-order interleaved memory structure
- Allows m memory words to be accessed **concurrently**
- This is called C-access

S-Access Memory Organization



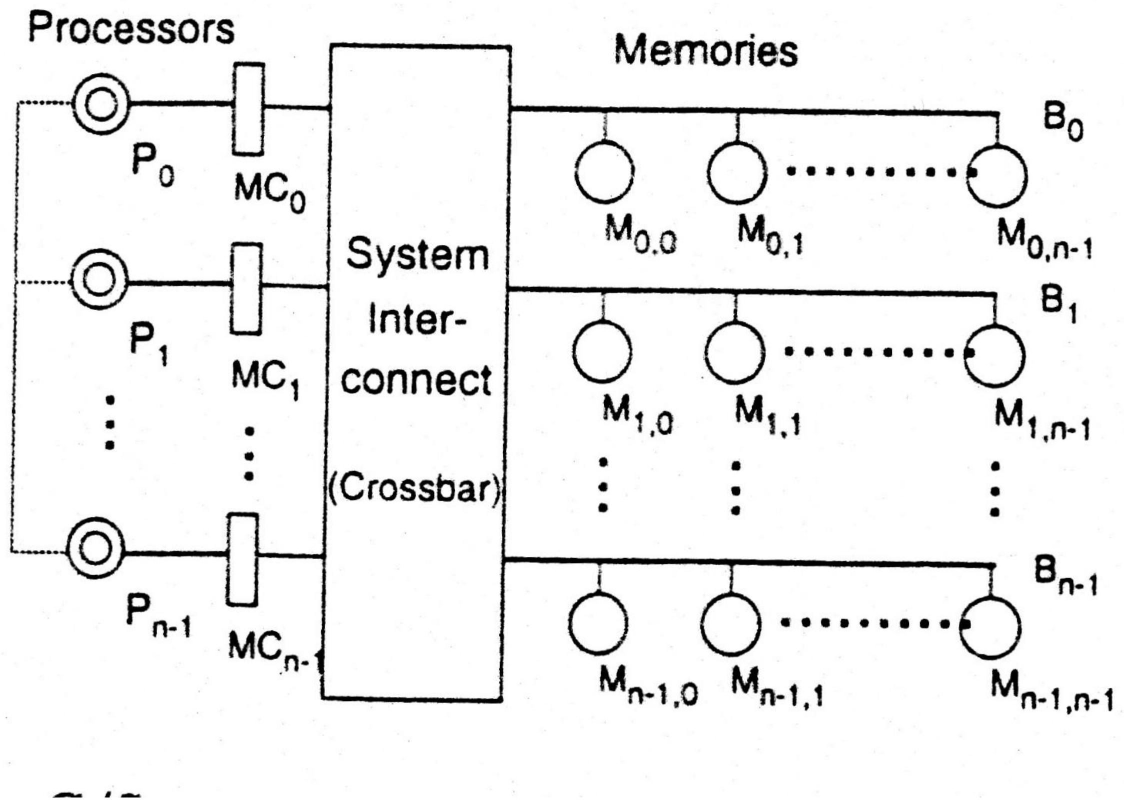
- Similar to low-order interleaved memory
  - High order bits select modules
  - Words from modules are latched at the same time
  - Low order bits select words from data latches
  - This is done through the multiplexed with higher speeds (minor cycles)
- Allows **simultaneous** access
- This is called S-access

Interleaved Fetch and Access



- If the minor cycle is selected 1/m
  - m words (one row) is accessed in 2 memory (major) cycles
- If fetch and access to the latches are interleaved
  - m words is accessed in 1 memory cycle

C/S Access



- C-access and S-access are combined
- n access busses with m interleaved memory modules
- The m modules on each bus are m-way interleaved to allow C-access
- The n busses operate in parallel to allow S-access

Balanced Vector/Scalar Ratio

- In a supercomputer separate hardware resources are dedicated to concurrent vector and scalar operations
- Vector processing is needed for regularly structured parallelism in scientific and engineering computations
- For a better performance these two types of operations must be balanced
- Vector balance point:
  - Percentage of vector code to achieve equal utilization of vector and scalar hardware
  - In best case none of the vector and scalar hardware is idle at any time

Vector Balance Point

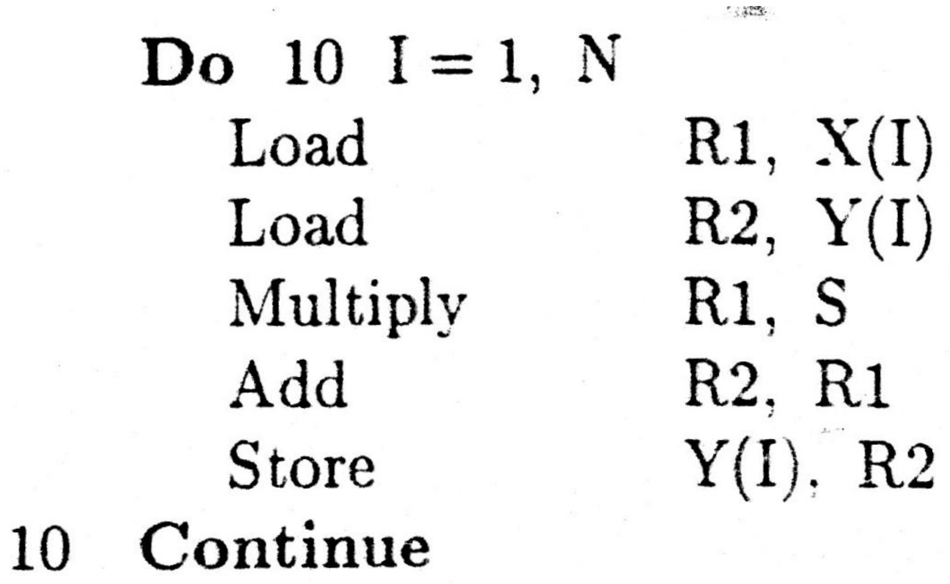
- Percentage of vector code to achieve equal utilization of vector and scalar hardware
- Example:
  - System capability:
    - 9 Mflops in vector mode
    - 1 Mflops in scalar mode
  - Equal time will be spent in each mode if the code is:
    - 90% vector code
    - 10% scalar code
  - The vector balance point is 0.9

Compound Vector Processing

CVF

- Compound Vector Function:
  - A composite function of vector operations converted from a looping structure of linked scalar operations

Example



```
    Do 10 I = 1, N
        Load            R1, X(I)
        Load            R2, Y(I)
        Multiply        R1, S
        Add             R2, R1
        Store           Y(I), R2
10  Continue
```

- X(I) and Y(I) are two source vectors with length N in memory

Vectorized Code

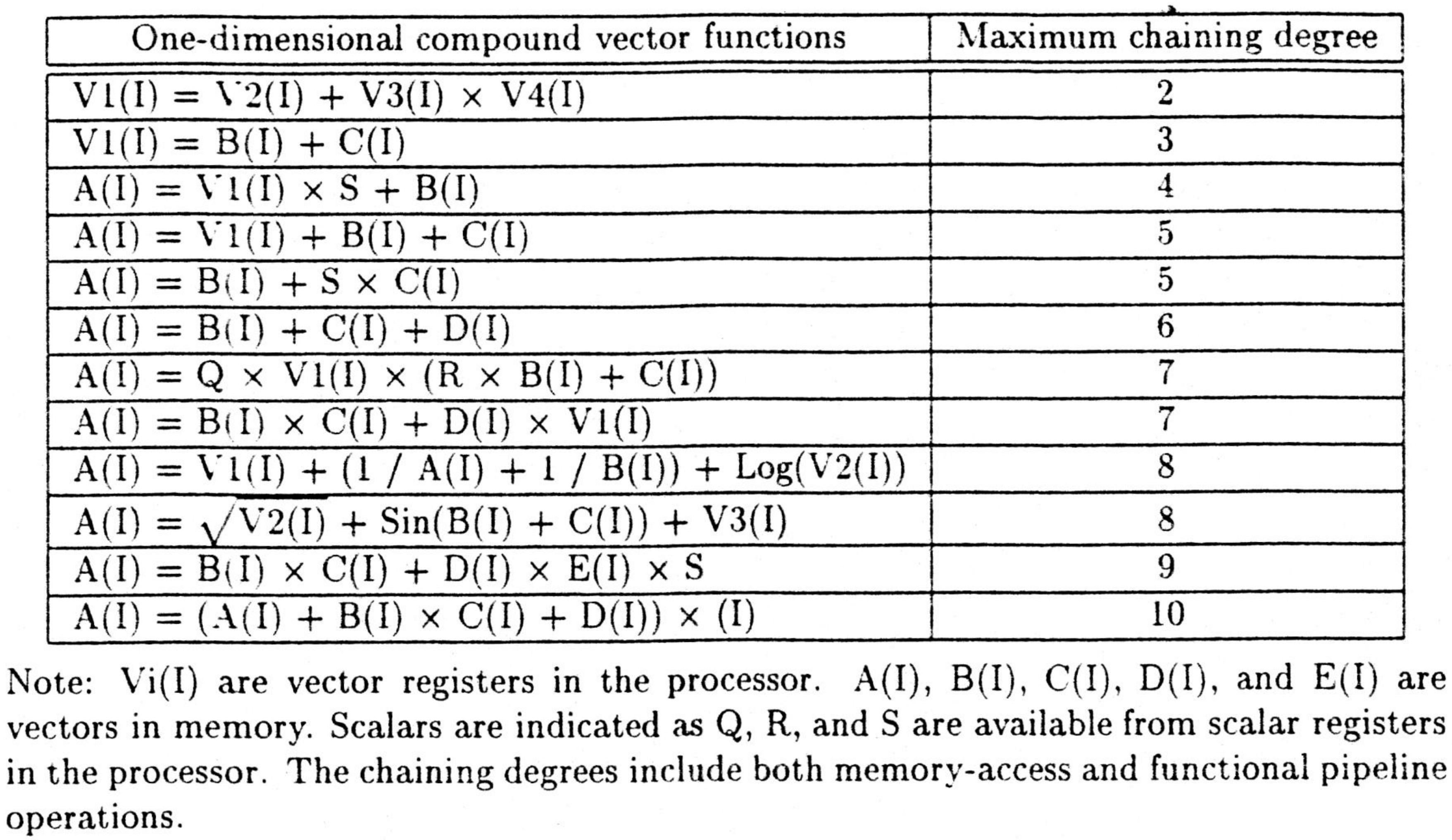
$$\begin{array}{rcll}
M(x : x + N - 1) & \rightarrow & V1 & \textit{Vector load} \\
M(y : y + N - 1) & \rightarrow & V2 & \textit{Vector load} \\
S \times V1 & \rightarrow & V1 & \textit{Vector multiply} \\
V2 + V1 & \rightarrow & V2 & \textit{Vector add} \\
V2 & \rightarrow & M(y : y + N - 1) & \textit{Vector store}
\end{array}$$

- Expressed as a **CVF**:

$$Y(1 : N) = S \times X(1 : N) + Y(1 : N)$$

$$Y(I) = S \times X(I) + Y(I)$$

Compound Vector Functions



One-dimensional compound vector functions	Maximum chaining degree
$V1(I) = V2(I) + V3(I) \times V4(I)$	2
$V1(I) = B(I) + C(I)$	3
$A(I) = V1(I) \times S + B(I)$	4
$A(I) = V1(I) + B(I) + C(I)$	5
$A(I) = B(I) + S \times C(I)$	5
$A(I) = B(I) + C(I) + D(I)$	6
$A(I) = Q \times V1(I) \times (R \times B(I) + C(I))$	7
$A(I) = B(I) \times C(I) + D(I) \times V1(I)$	7
$A(I) = V1(I) + (1 / A(I) + 1 / B(I)) + \mathrm{Log}(V2(I))$	8
$A(I) = \sqrt{V2(I)} + \mathrm{Sin}(B(I) + C(I)) + V3(I)$	8
$A(I) = B(I) \times C(I) + D(I) \times E(I) \times S$	9
$A(I) = (A(I) + B(I) \times C(I) + D(I)) \times (I)$	10

Note: Vi(I) are vector registers in the processor. A(I), B(I), C(I), D(I), and E(I) are vectors in memory. Scalars are indicated as Q, R, and S are available from scalar registers in the processor. The chaining degrees include both memory-access and functional pipeline operations.

Strip-Mining

- Segmentation of a long vector in memory
- Fixed length segments
- Loading and processing the segments one segment at a time
- Segment length matches the vector register size
- More flexible if vector register size can be configured
- The vector register used for the vector is not released until all the segments are processed

Vector Loop

- The program construct for processing long vectors is called a vector loop
- Strip-mining is a part of the vector loop
- All is done in hardware

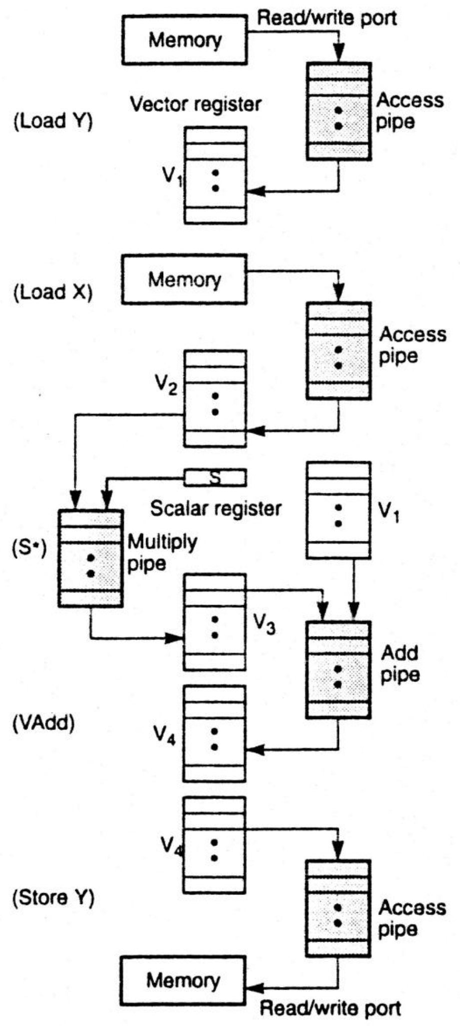
Chaining

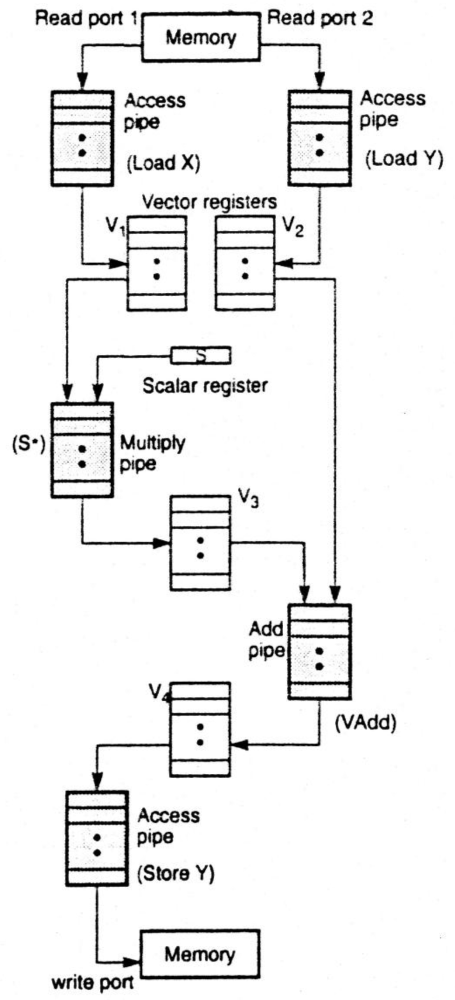
- Chaining of multiple pipelines is used for concurrent processing of several vector operations
- A CVF is a candidate for chaining
- Actual implementation depends on the hardware

Functional Units in the Chain

- Linked vector operations must follow a linear data flow pattern
- Functional pipeline units must be independent of each other
- Same unit cannot be assigned to execute more than one instruction in the same chain
- Vector registers must be used as interface between functional pipelines

Examples of Pipeline Chaining





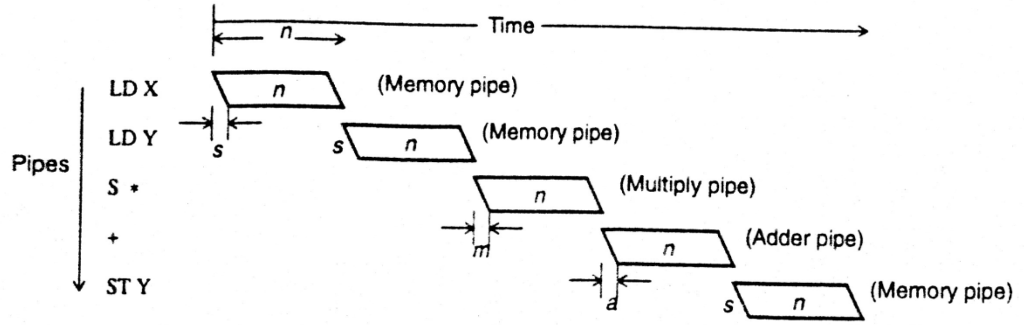
Chaining with only one memory-access pipe compared to chaining with three memory-access pipes

The Vector Registers

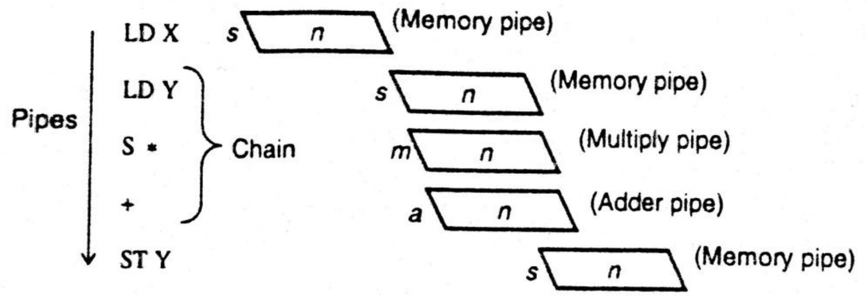
- The successive output results of a pipeline are fed into the vector register one element per cycle
- The vector register is then used as an input register for the next pipeline unit in the chain
- The interface registers must be able to pass one vector element per cycle between adjacent pipelines

Timing in Various Chaining Scenarios in the Example

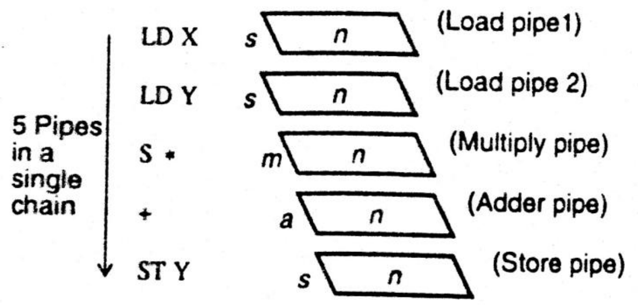
- Sequential execution without chaining



- Chaining with only one memory access pipe



- Chaining with three memory access pipes



Captions:

s = Memory access latency

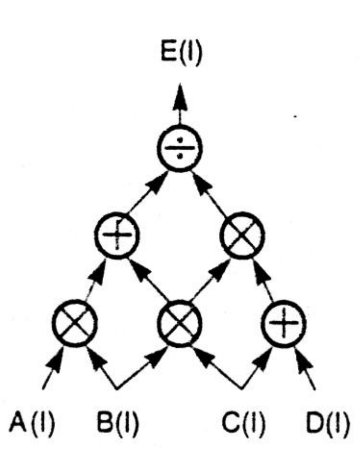
m, *a* = Multiply/add latencies.

n = Time to process *n* elements, one per cycle.

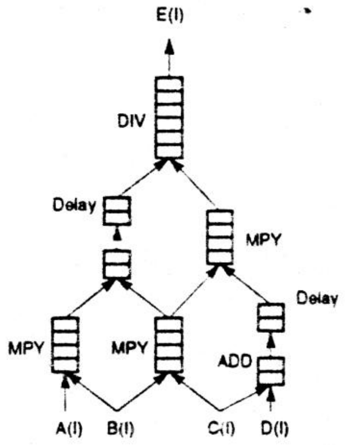
Multipipeline Networking

- Generalization of the idea of linking vector operations (chaining)
- Instead of a linear chain, a pipeline net (pipenet) is built
- Multiple functional pipelines are linked to achieve systolic computation of CVFs
- A systolic array is formed with a network of functional units which are locally connected and operate synchronously
- Unlike a systolic architecture which is fixed, a pipenet can be configured dynamically

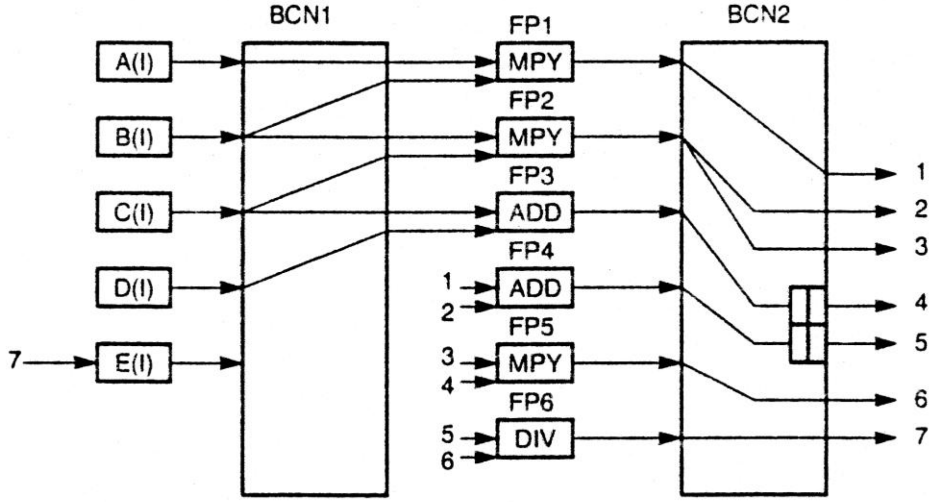
Implementation of a Pipenet



(a) A program graph



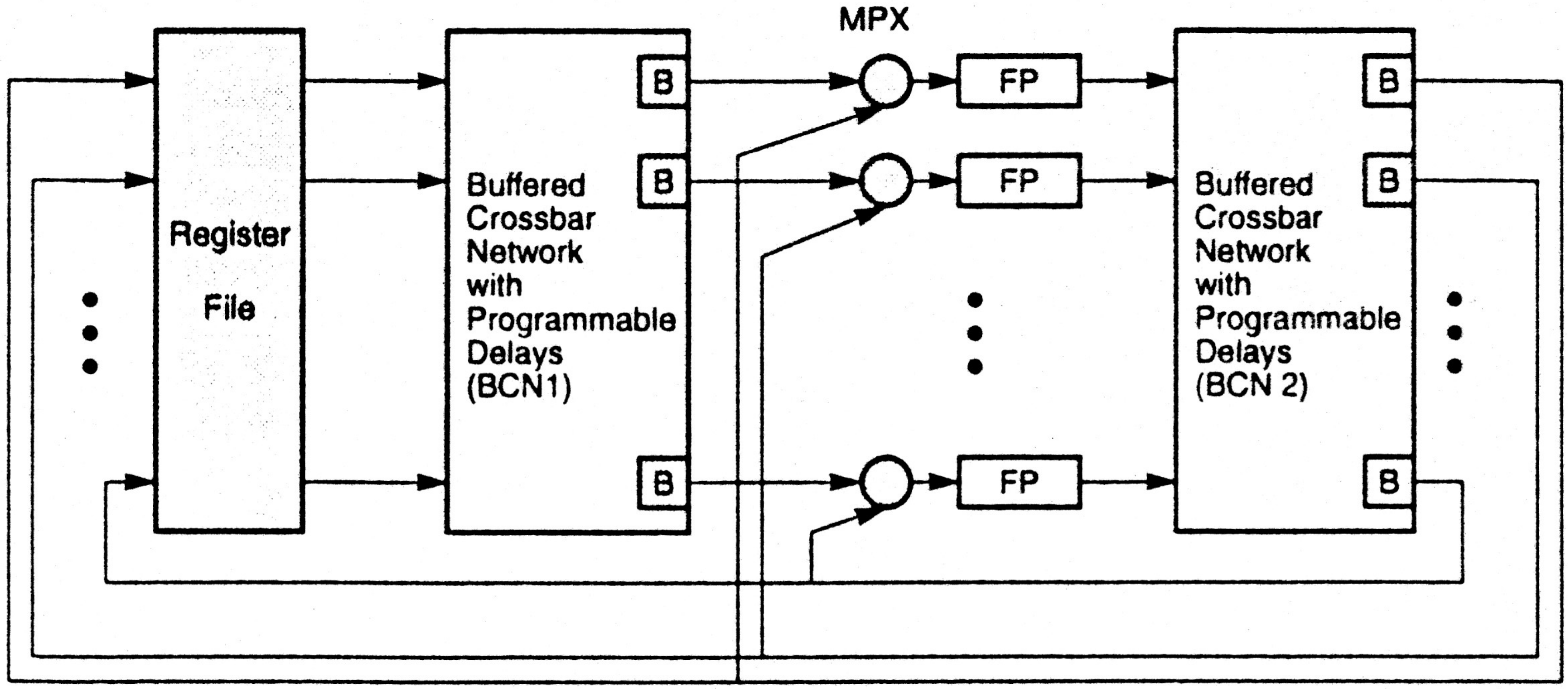
(b) The pipenet



(c) A crossbar implementation

$$E(I) = [A(I) \times B(I) + B(I) \times C(I)] \ / \ [B(I) \times C(I) \times [C(I) + D(I)]]$$

Generalized Pipenet Model



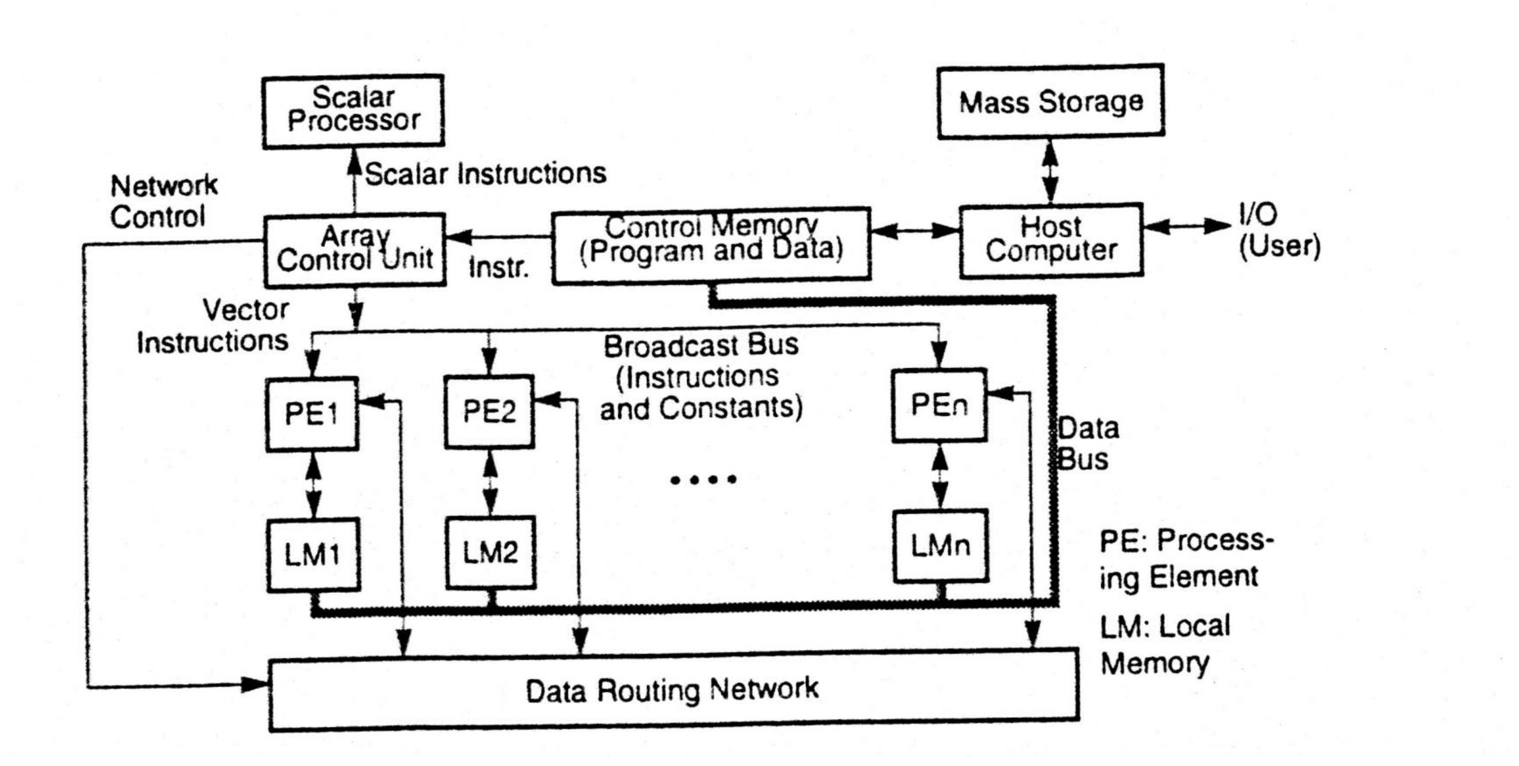
MPX
Register File
Buffered Crossbar Network with Programmable Delays (BCN1)
B
B
B
FP
FP
FP
Buffered Crossbar Network with Programmable Delays (BCN 2)
B
B
B

SIMD Computers

SIMD Computers

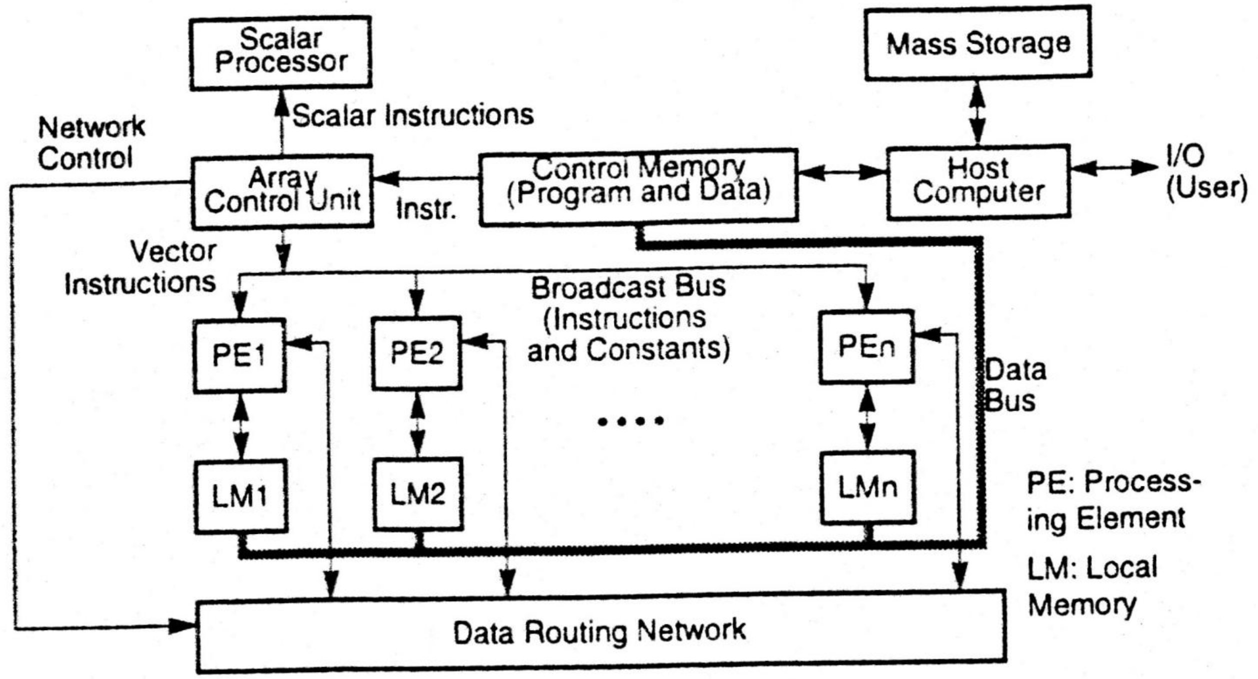
- Vector processing can be carried out by SIMD computers
- Vector instruction’s operands must have a fixed length of n equivalent to the number of PEs
- Two models:
  - Distributed memory model
  - Shared memory model

Distributed-Memory Model



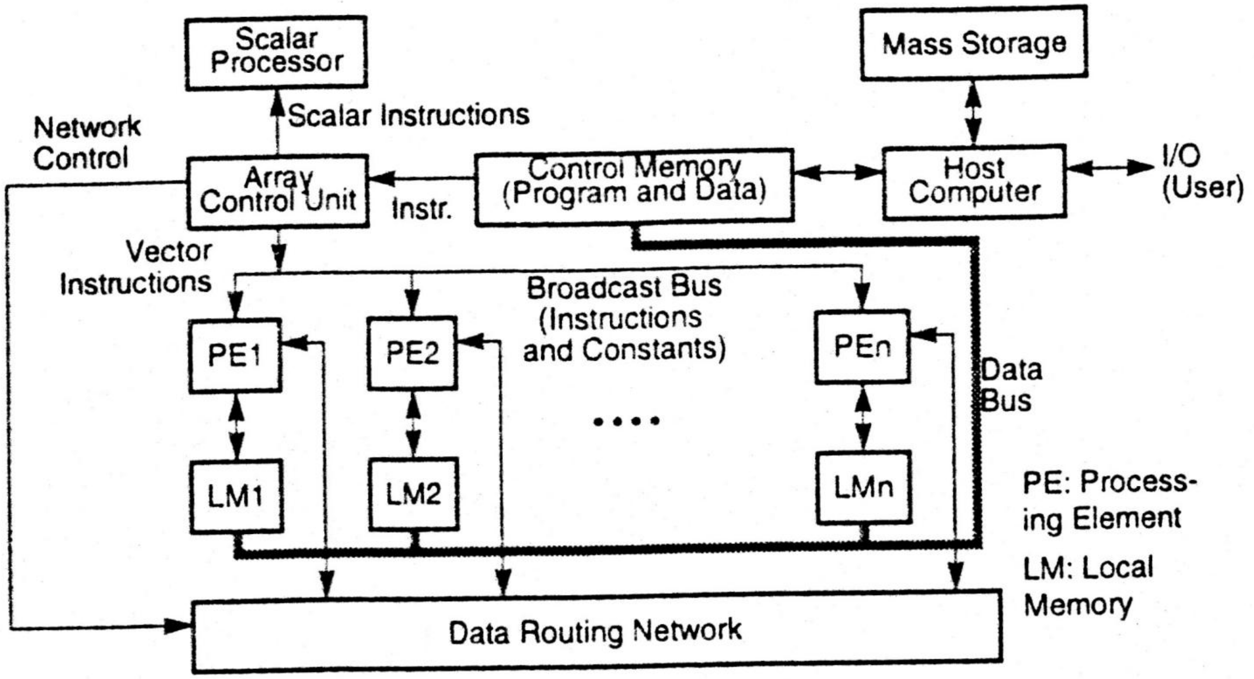
- Spatial parallelism is explored
  - An array of PEs
  - An array control unit
- Program and data are loaded into the control memory through a host unit

Distributed-Memory Model



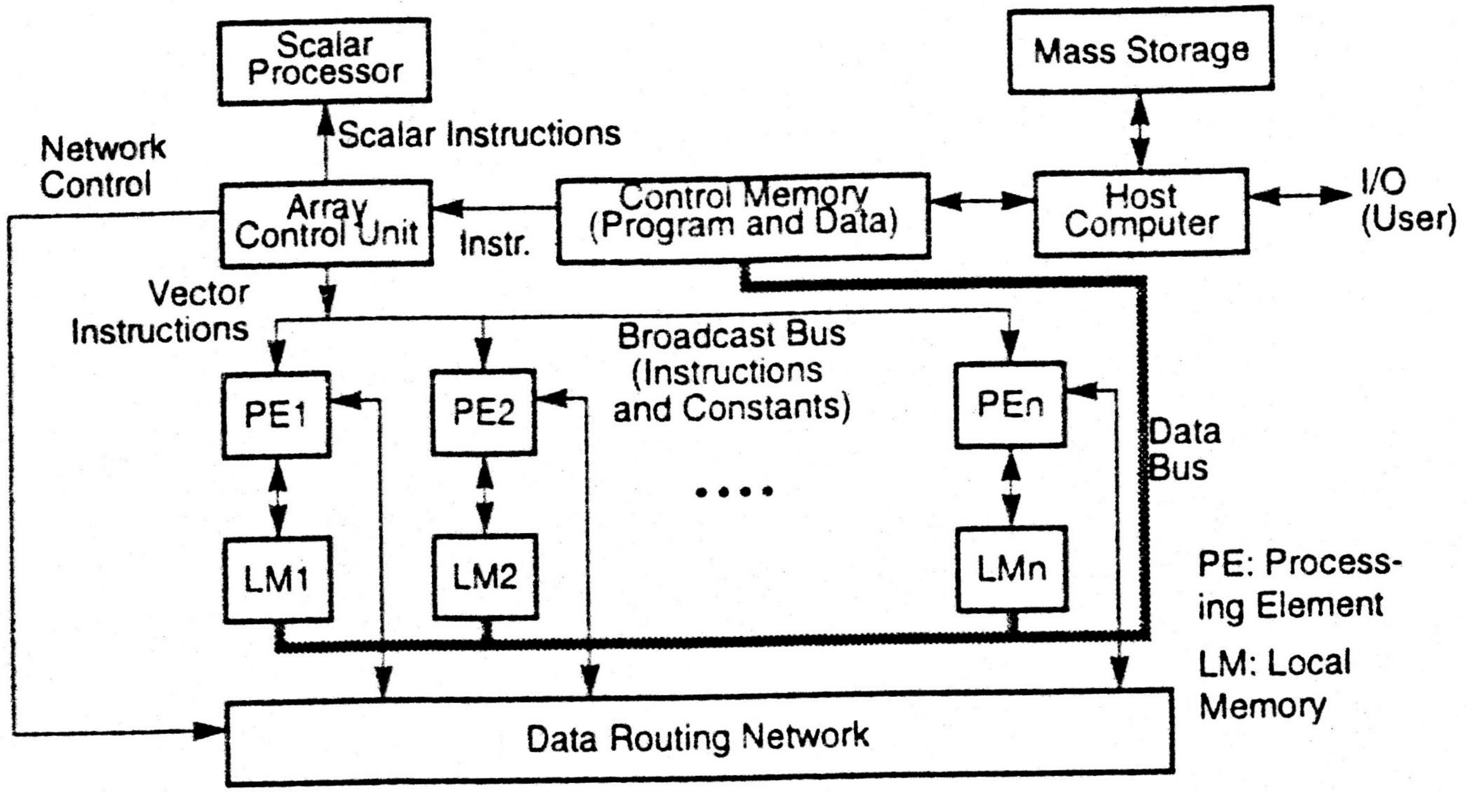
- Instructions are sent to the control unit for decoding
- A scalar or program control operation is directly executed by a scalar processor attached to the control unit
- A vector instruction will be broadcast to all PEs for execution
- Partitioned data sets are distributed to all the local memories attached to the PEs through a vector data bus

Distributed-Memory Model

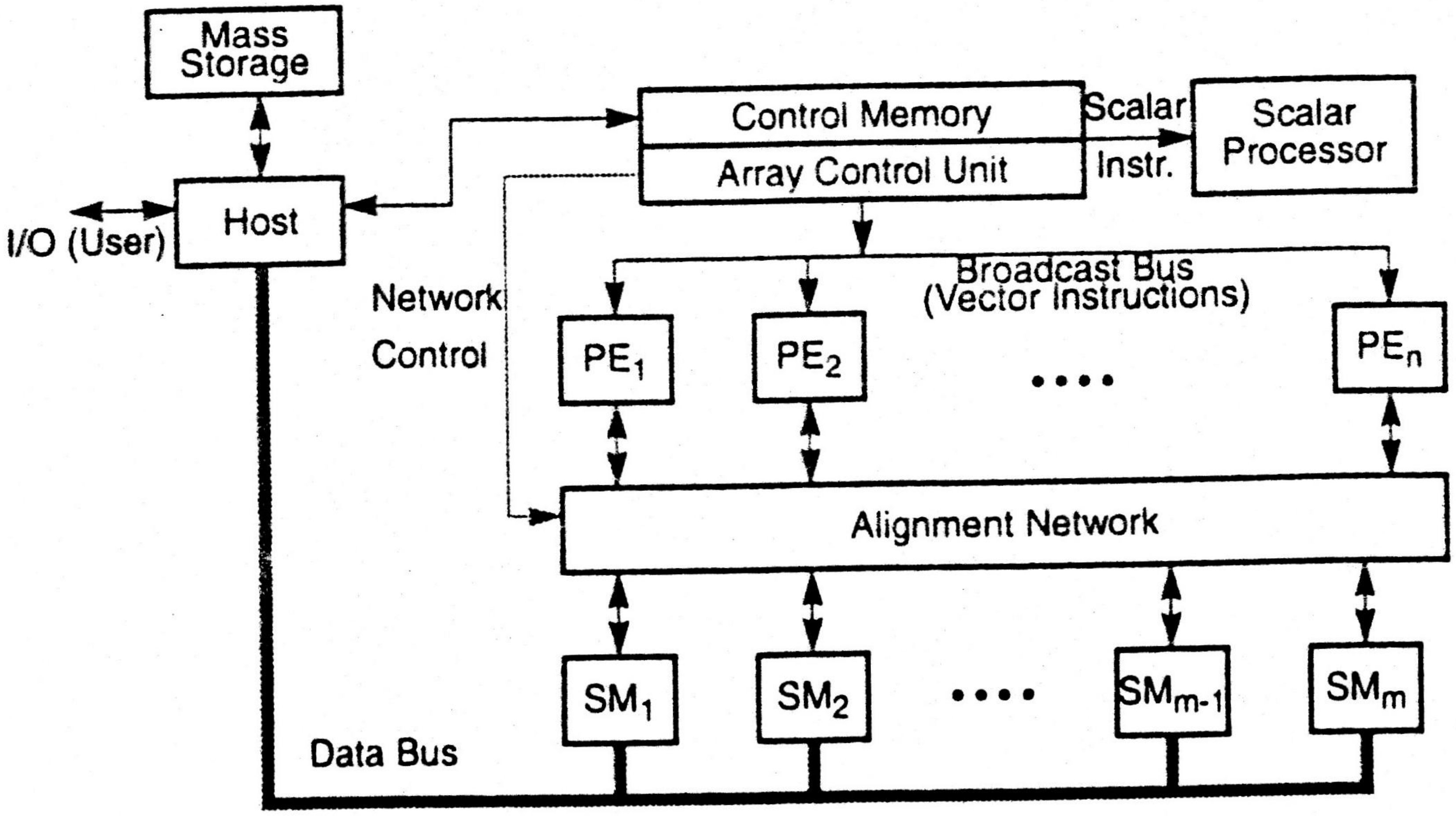


- The PE s are synchronized in hardware by the control unit
- The same instruction is executed by all the PEs in the same cycle
- Masking logic is provided to disable any PE from participating in a given instruction cycle
- The PE s are interconnected by a data-routing network which performs inter-PE data communications
- The data-routing network is under program control through the control unit

Distributed-Memory SIMD Model



Shared-Memory SIMD Model



Mass Storage
Control Memory
Scalar Instr.
Scalar Processor
Array Control Unit
I/O (User)
Host
Network Control
Broadcast Bus (Vector Instructions)
PE$_1$
PE$_2$
....
PE$_n$
Alignment Network
SM$_1$
SM$_2$
....
SM$_{m-1}$
SM$_m$
Data Bus