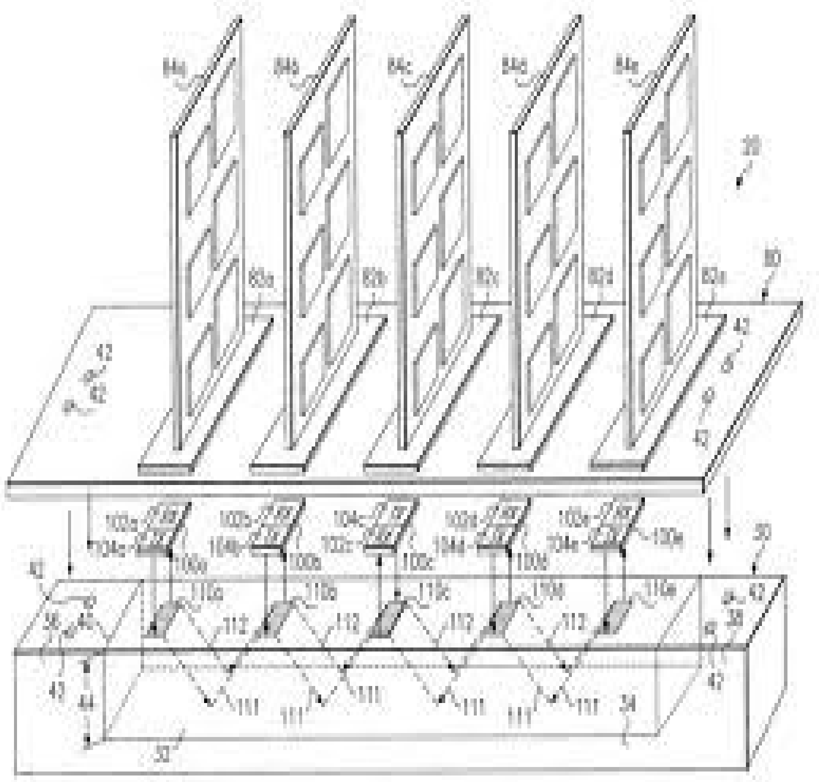
Bus, Cache and shared memory

Bus System

- System bus of a computer system operates on contention basis
- Effective bandwidth available to each processor is inversely proportional to the number of processor contending for the bus
- This is the reason for simplicity (4-6 processors) and low cost

Backplane bus specification

- It interconnects processors, data storage and peripheral devices in a tightly coupled h/w configuration
- Allow the interconnection between the devices



- Timing protocols
  - Operational rules-orderly data transfers on the bus

Backplane Multiprocessor System

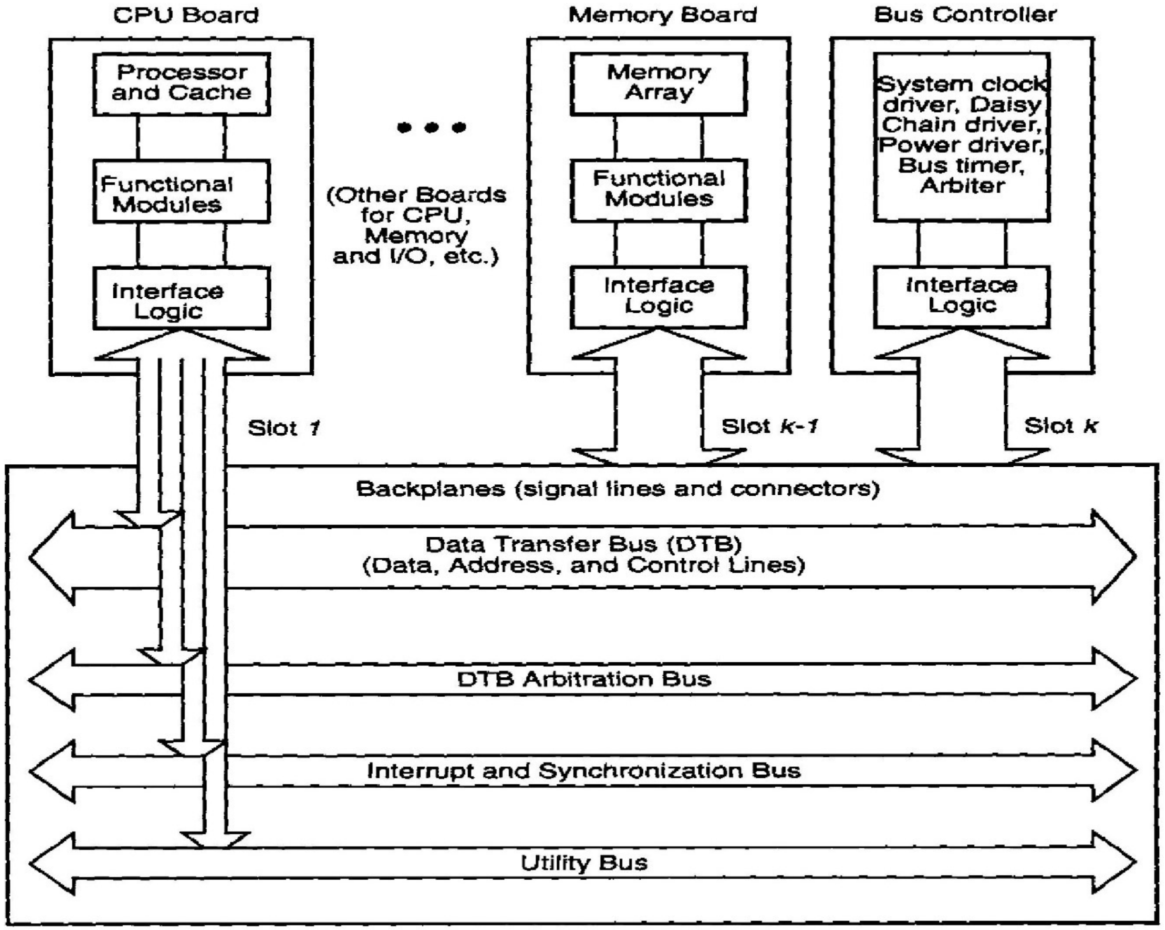

CPU Board
Processor and Cache
Functional Modules
Interface Logic
Slot 1
(Other Boards for CPU, Memory and I/O, etc.)
Memory Board
Memory Array
Functional Modules
Interface Logic
Slot k-1
Bus Controller
System clock driver, Daisy Chain driver, Power driver, Bus timer, Arbiter
Interface Logic
Slot k
Backplanes (signal lines and connectors)
Data Transfer Bus (DTB) (Data, Address, and Control Lines)
DTB Arbitration Bus
Interrupt and Synchronization Bus
Utility Bus

Figure 5.1 Backplane buses, system interfaces, and slot connections to vario functional boards in a multiprocessor system.

Data Transfer Bus (DTB)

- Data, address and control lines form Data transfer bus (DTB).
  - Transfers data, address and control signals
- Addressing lines are used to broadcast the data and address
- The number of addressing lines is proportional to the logarithm of the size of the address space

- Data lines are proportional to the memory word length
  - VME bus system has 32 address lines and 32 data lines
- Control lines are used to indicate read/write, timing control and bus error conditions

Bus Arbitration and Control

- Sharing of buses in an optimal way
- The process of assigning control of the DTB to a requester is called *arbitration*
- Requester is called a *master* and the receiving end is called a *slave*
- *Interrupt* lines are used to handle interrupts
  - This is prioritized

Functional modules

- It is a collection of electronic circuitry that resides on one functional board
- *Arbiter* is a functional module that accepts bus requests from the requester module and grants control of the DTB to one requester at a time
- *Bus timer* measures the time for each data transfer

- An *interrupter* module generates an interrupt request and provides status info when an interrupt handler module requests
- A *location monitor-* monitors the data transfers over the DTB
- A *power monitor* watches the status of the power source and signals when power becomes unstable
- A *system clock driver* provides a clock timing signal
  - Board interface logic is needed to match the signal line impedance, propagation time and termination b/w backplane and plug-in boards

Physical limitations

- Due to electrical, mechanical & packaging limitations, limited number of boards can be plugged into a single backplane
- Multiple backplane buses can be mounted on the same backplane chassis
- Bus system is difficult to scale, limited by contention and packaging constraints

Addressing and Timing protocols

- Two IC chips connected to a bus:
  1. active
  2. passive
- Active chips are like processors act as bus master
- Passive chips are memories can act only as slaves

- Master can initiate a bus cycle
- Slaves respond to requests by a master

- Only one master can control the bus at a time
- One or more slaves can respond the master’s requests at the same time

Bus Addressing

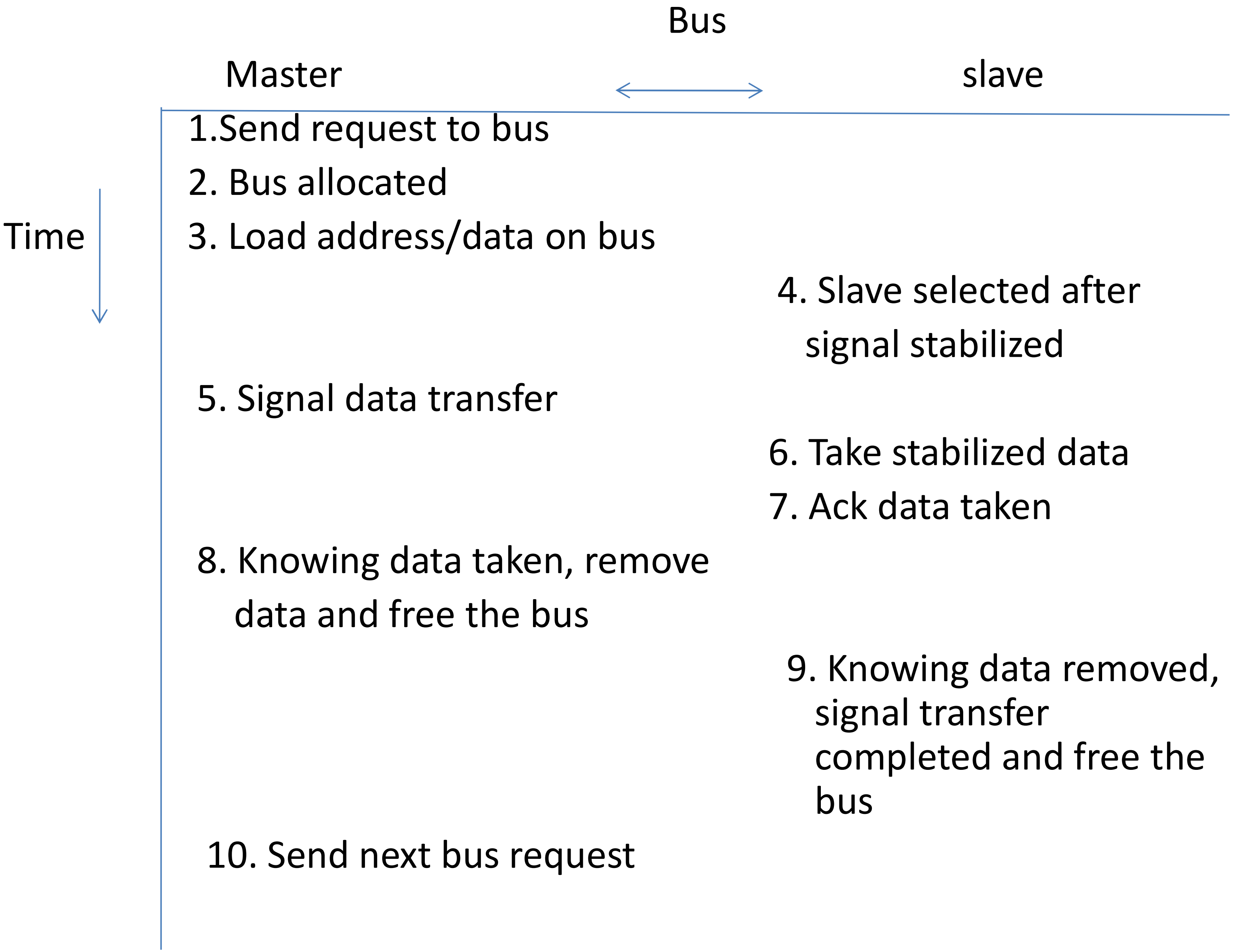
- Backplane is driven by a fixed cycle time called bus cycle
- Bus cycle is determined by the electrical, mechanical and packaging characteristics
- To optimize the performance, the bus should be designed to minimize the time required for
    1. Request handling
    2. Arbitration
    3. Addressing
    4. Interrupts

Bus Addressing

- Identify each board with a slot number
- When slot # matches contents of high-order address lines, the board is selected as a slave (slot addressing)

Broadcall and broadcast

- Broadcall is a read operation
  - Multiple slaves placing their data on the bus lines
  - It is used to detect multiple interrupt sources
- Broadcast is a write operation
  - Multiple slaves writing their data into their storage devices
  - Timing protocol is needed to synchronize the master and slave operations



Bus

Master

slave

Time

1.Send request to bus

2. Bus allocated

3. Load address/data on bus

4. Slave selected after signal stabilized

5. Signal data transfer

6. Take stabilized data

7. Ack data taken

8. Knowing data taken, remove data and free the bus

9. Knowing data removed, signal transfer completed and free the bus

10. Send next bus request

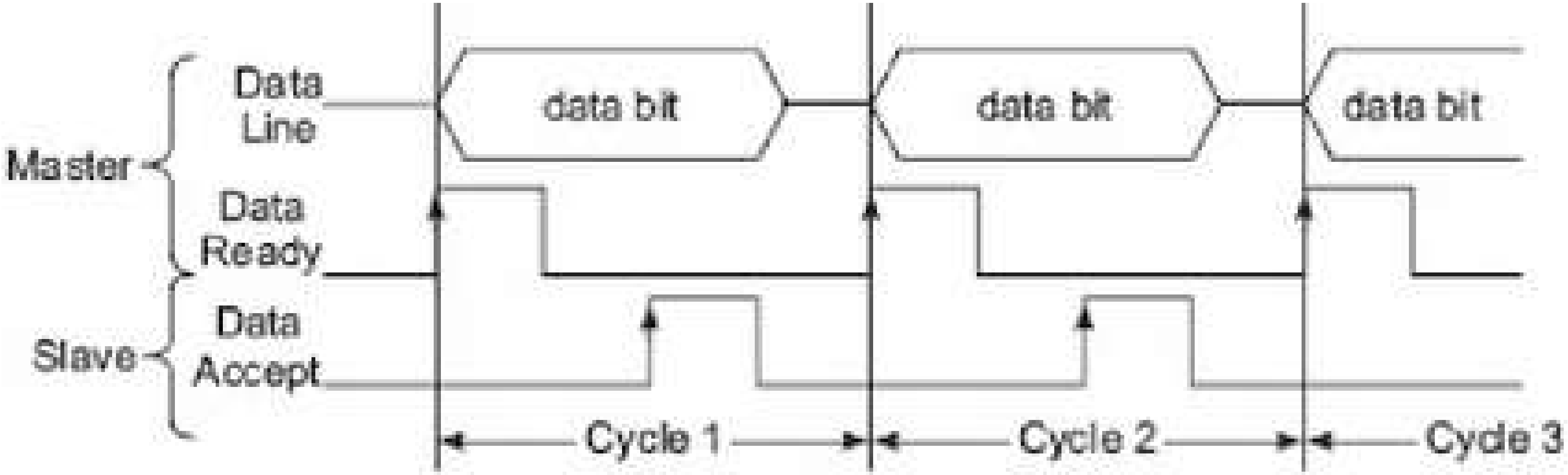
Synchronous Timing

- Fixed clock pulses
- Steps:

  1. First data should be stabilized on the data lines

  2. Master uses a data-ready pulse to initiate the data transfer

  3. Slave uses a data-accept pulse to signal completion of the information transfer



(a) Synchronous bus timing with fixed-length clock signals for all devices

Advantages:

1. simple to control
2. requires less control circuitary
3. cost is less

Disadvantages:

1. suitable for connecting devices having relatively same speed otherwise, slower device will slow down the entire bus operation

Asynchronous Timing

- Based on a handshaking or interlocking mechanism
- No fixed clock cycle is needed
- Data-ready, data-accept

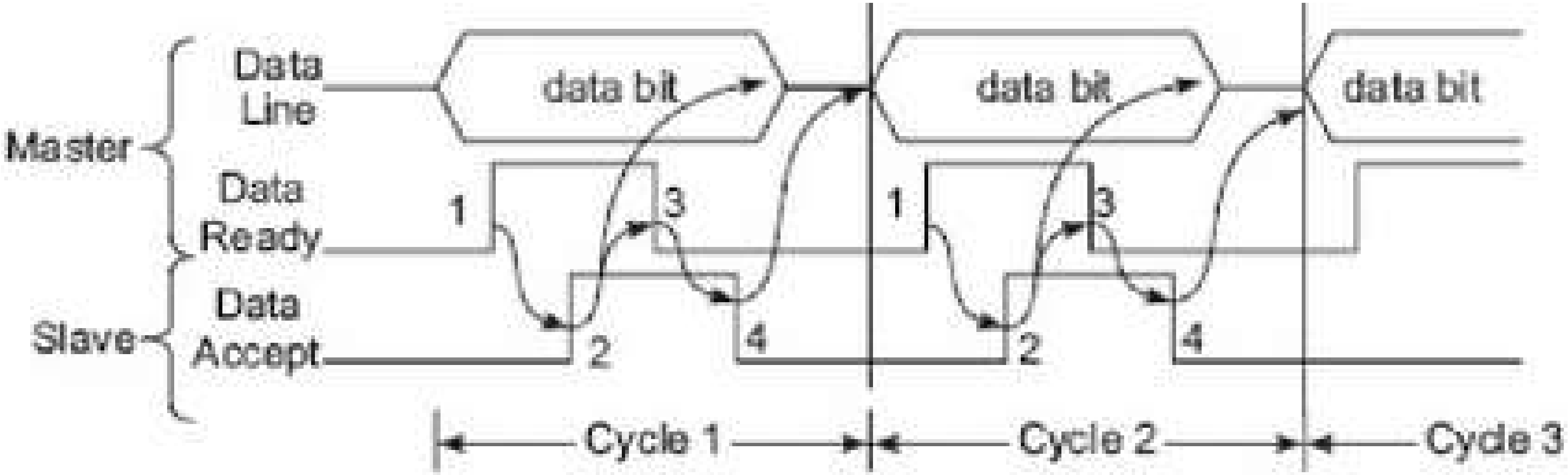
Rising Edge:

1. In master, the data ready signal triggers the data-accept signal of slave indicates data transfer

Trailing Edge:

In master, the data-ready signal triggers the data-accept signal of slave indicates the removal of data from the bus

- Adv: variable length clock signals in different speed
- Fast and slow devices can be connected
- flexibility



(b) Asynchronous bus timing using a four-edge handshaking (interlocking with variable length signals for different speed devices.

Arbitration, Transaction and Interrupt

- Process of selecting the next bus master is called *arbitration*
- Types

  Central arbitration

  Distributed arbitration

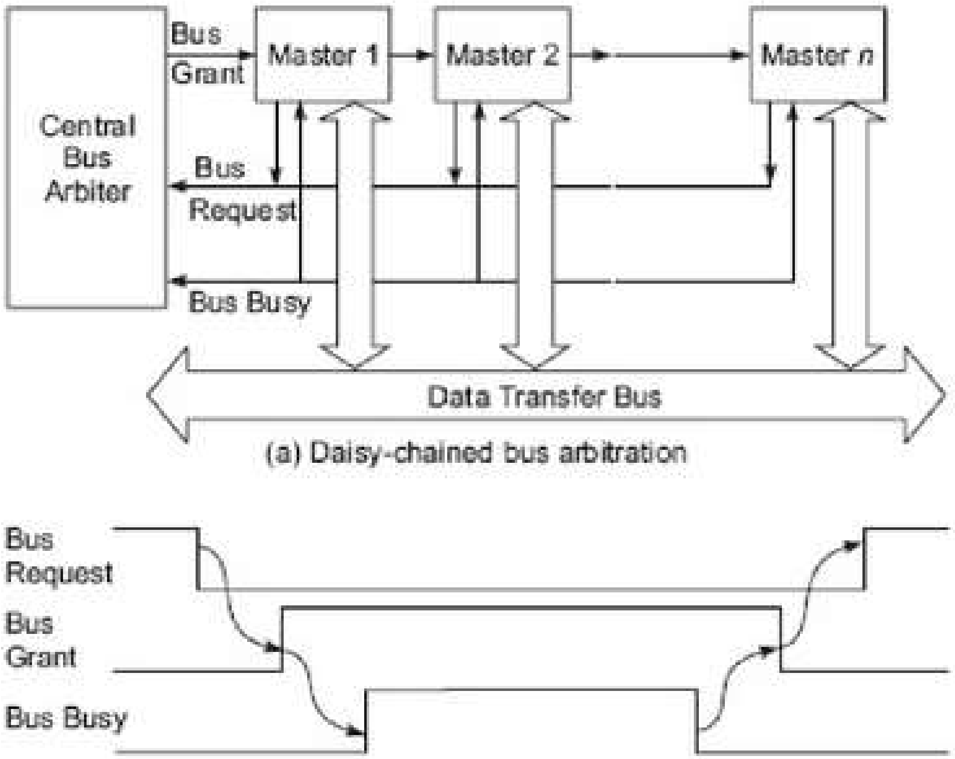
Central Arbitration

- Each master can send request
- All requests share the same bus-request line
- Allocation is based on the priority
- Adv: Simplicity

    Additional devices can be added

Disadv: Fixed priority

Slowness



(a) Daisy-chained bus arbitration

(b) Bus transaction timing

Fig. 5.4 Central bus arbitration using shared requests and daisy-chained bus

II. Independent Requests and Grants

It is also possible to have independent multiple bus-request and bus-grant signal lines for each potential master.

In this scheme, no daisy-chaining is used. Still we have a common central arbiter.

Fig. 6.6. shows an independent requests with a central arbiter.

Advantages of this scheme are as follows :

1. More flexibility is provided by bus lines.
2. Faster arbitration as compared to daisy-chaining.

Disadvantages of this scheme

Large number of arbitration lines are required.

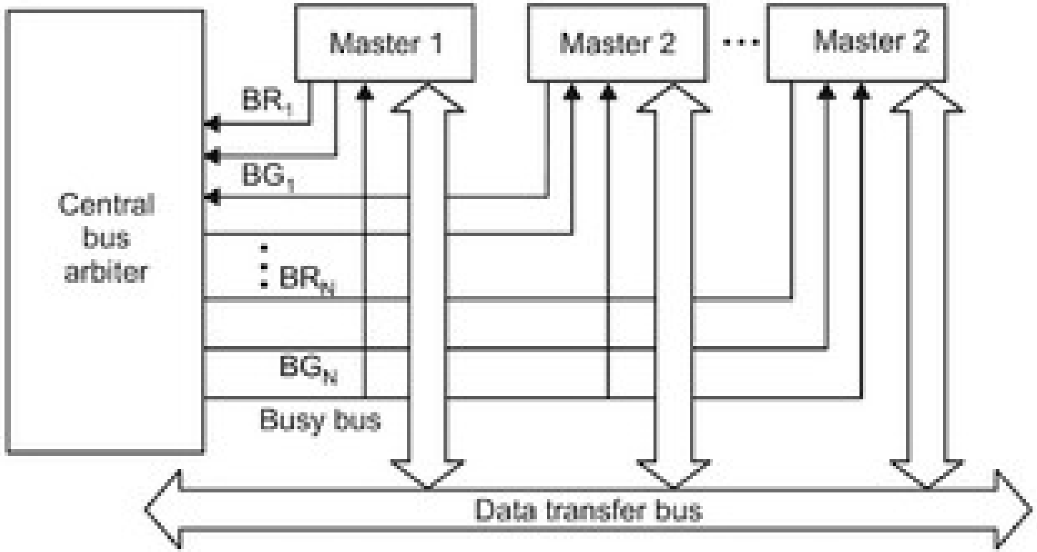
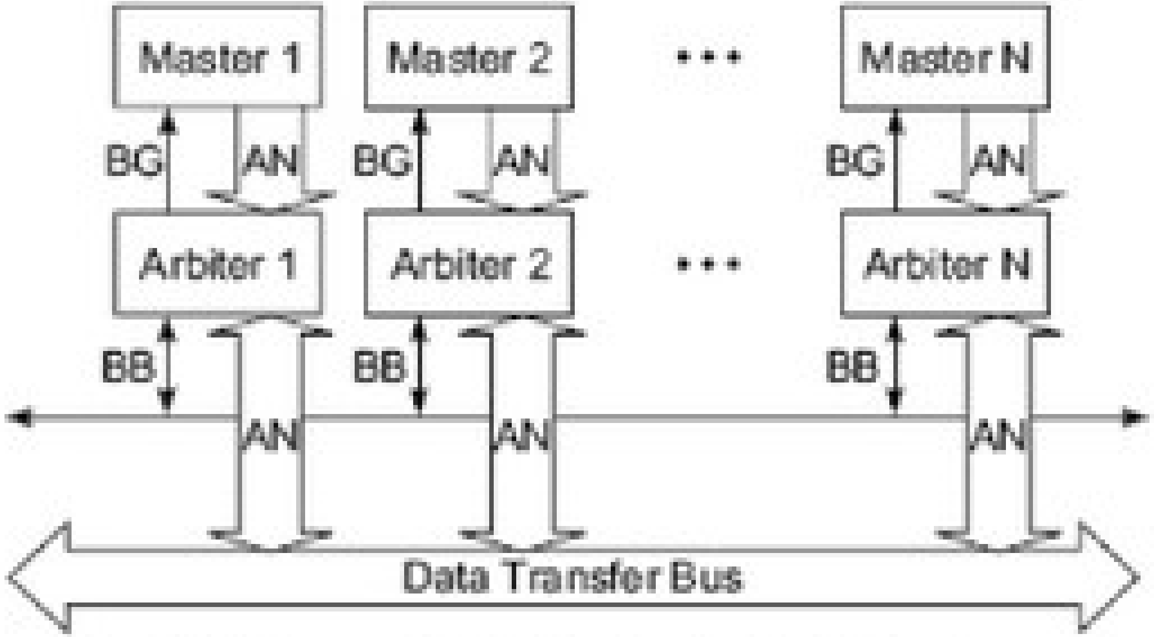


Fig. 6.6. Independent Requests method.

Distributed Arbitration

- Each master is equipped with its own arbiter
- Each arbiter has arbitration number
- Arbitration number is used to resolve the arbitration competition
- When two or more compete for the arbitration, winner is high arbitration number

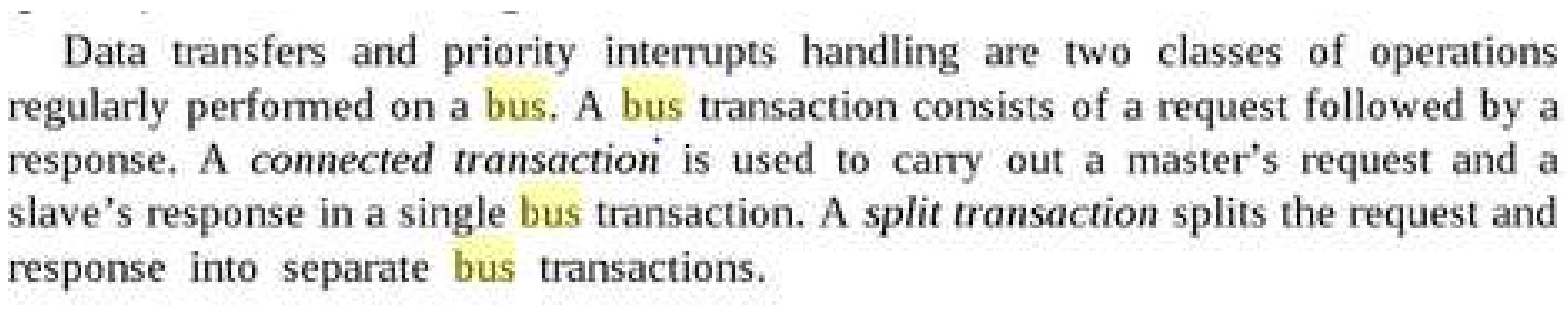


Legends: BG (Bus grant) BB (Bus busy) AN (Arbitration number)

(b) Using distributed arbiters

Transaction Modes

- Address-only transfer
  - Consists of an address transfer followed by no data.
- Compelled data transfer
  - Consists of an address transfer followed by a block of one or more data transfer to one or more contiguous addresses.
- Packet data transfer
  - Consists of an address transfer followed by a fixed length block of data transfers (packet) from a set of contiguous addresses.



Data transfers and priority interrupts handling are two classes of operations regularly performed on a bus. A bus transaction consists of a request followed by a response. A *connected transaction* is used to carry out a master's request and a slave's response in a single bus transaction. A *split transaction* splits the request and response into separate bus transactions.

Interrupt Mechanisms

- It is a request from I/O or other devices to a processor for service or attention
- Priority interrupt bus is used to pass the interrupt signals
- The interrupter must provide status and ID information

Futurebus+ Goals

- Open bus standard to support:
  - 64 bit address space
  - Throughput required by multi-RISC or future generations of multiprocessor architectures
- Expandable or scalable
- Independent of particular architectures and processor technologies

Standard Requirements

- Independence for an open standard
- Asynchronous timing protocol
- Optional packet protocol
- Distributed arbitration protocols
- Support of high reliability and fault tolerant applications
- Ability to lock modules w/o deadlock or livelock

Standard Requirements

- Circuit-switched and split transaction protocols
- Support of real-time mission critical computations w/multiple priority levels
- 32 or 64 bit addressing
- Direct support of snoopy cache-based procs.
- Compatible message passing protocols

Futurebus+ Signal Lines

- Information (150 – 306)
- Synchronization (7)
- Bus arbitration (18)
- Handshake (6)

Information Lines

- 64 address lines multiplexed with lower order 64 data lines
- Data path can be up to 256 bits wide
- Tag lines extend address/data modes (opt)
- Command lines carry info from master
- Status lines used by slaves to respond
- Capability lines to declare special bus transactions
- Parity check lines for protection

Synchronization Lines

- Coordinate exchange of address, command, capability status and data
- Address/data handshake lines used by both master and slaves
- Bus tenure line used to coordinate transfer of bus control

Arbitration and Misc. Lines

- Arbitration bus lines carry a number to signify precedence of competitors
- Central arbitration lines for central bus control
- Geographical lines for slot addresses
- Additional lines for utility, clock, and power connections

Cache Memory Organization

Inventor of Cache memory

- **M. V. Wilkes, “Slave Memories and Dynamic Storage Allocation,”**
- ***IEEE Transactions on Electronic Computers*, vol. EC-14, no. 2,**
- **pp. 270-271, April 1965.**

Cache Memory Organization

Capacity
Access Time
Cost

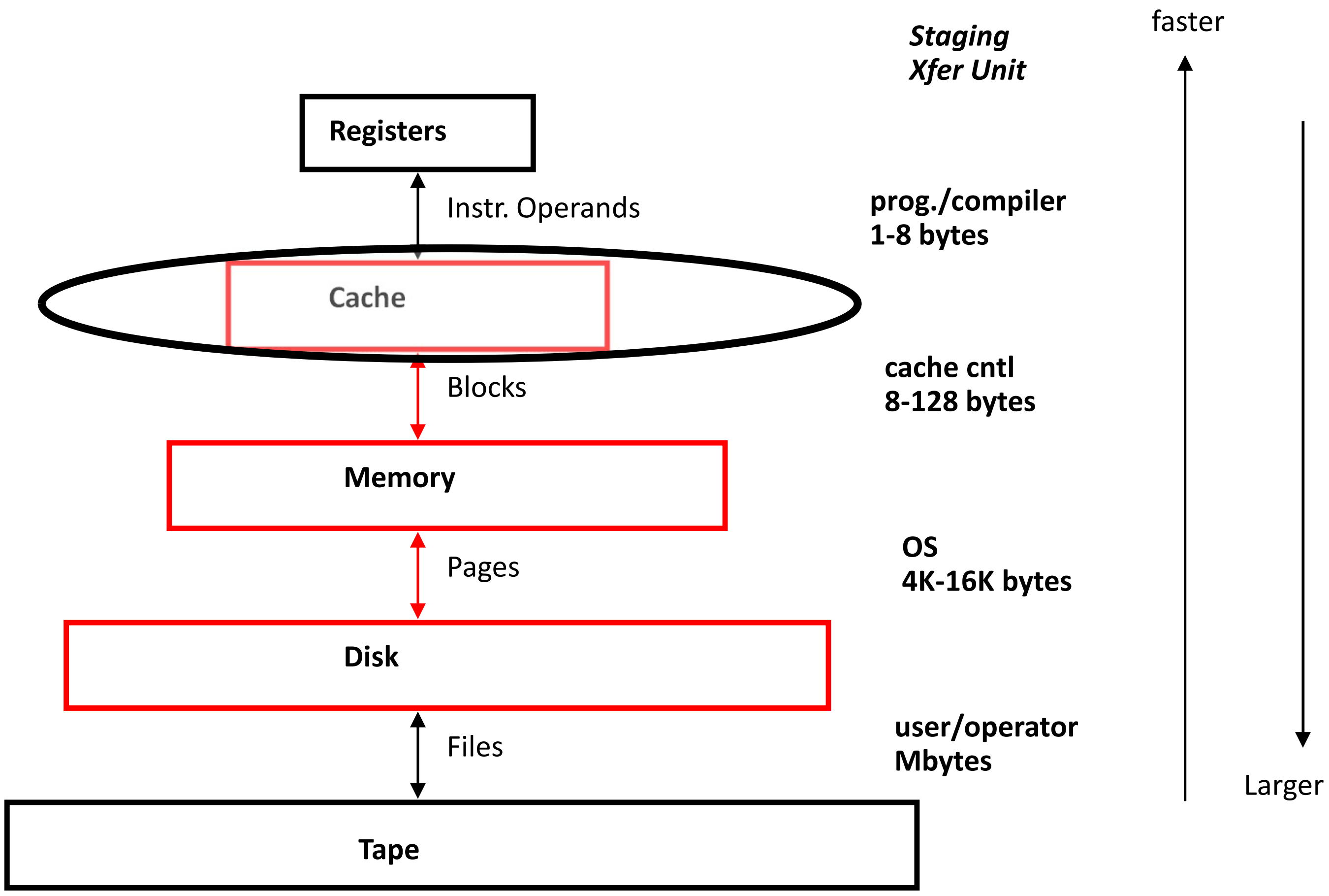
CPU Registers
100s Bytes
<10s ns

Cache
K Bytes
10-100 ns
1-0.1 cents/bit

Main Memory
M Bytes
200ns- 500ns
$.0001-.00001 cents /bit

Disk
G Bytes, 10 ms
(10,000,000 ns)
10^{-5} - 10^{-6} cents/bit

Tape
infinite
sec-min
10^{-8}



Registers

Instr. Operands

Cache

Blocks

Memory

Pages

Disk

Files

Tape

Staging
Xfer Unit

prog./compiler
1-8 bytes

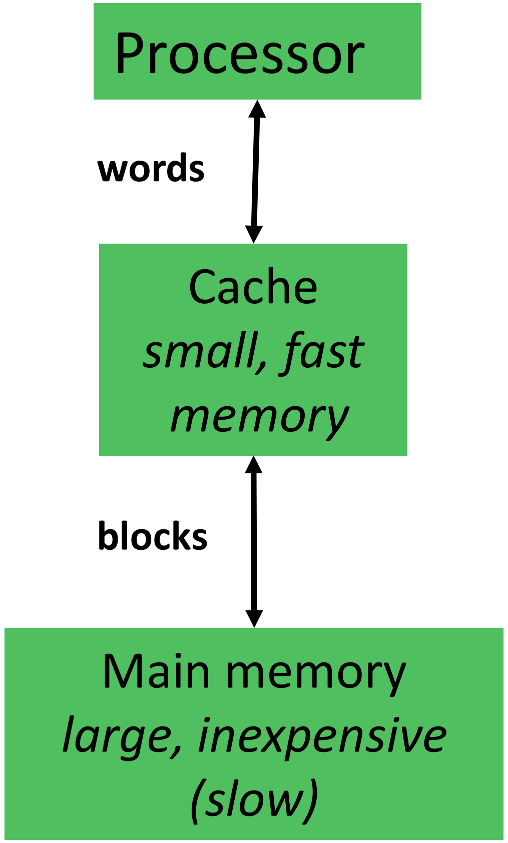
cache cntl
8-128 bytes

OS
4K-16K bytes

user/operator
Mbytes

faster

Larger



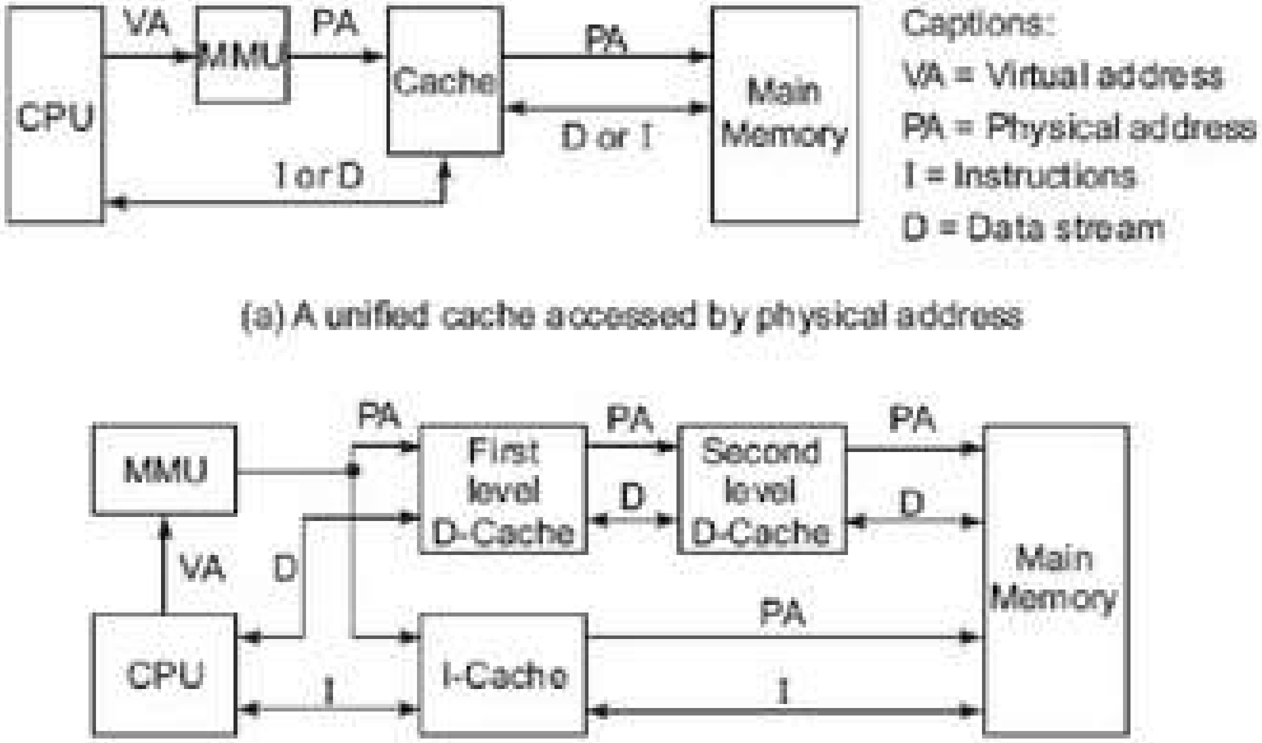
- Processor does all memory operations with cache.
- *Miss* – If requested word is not in cache, a *block* of words containing the requested word is brought to cache, and then the processor request is completed.
- *Hit* – If the requested word is in cache, read or write operation is performed directly in cache, without accessing main memory.
- *Block* – minimum amount of data transferred between cache and main memory.

Cache addressing models

Physical Address Caches

- When a cache is accessed with a physical memory address

  Ex: VAX8600, Intel i486
- Data is written through the memory immediately via *write-through(WT)* cache
- Delayed until block replacement by using a *write-back (WB) cache*



(a) A unified cache accessed by physical address

(b) Split caches accessed by physical address in the Silicon Graphics workstation

Fig. 5.7 Physical address models for unified and split caches

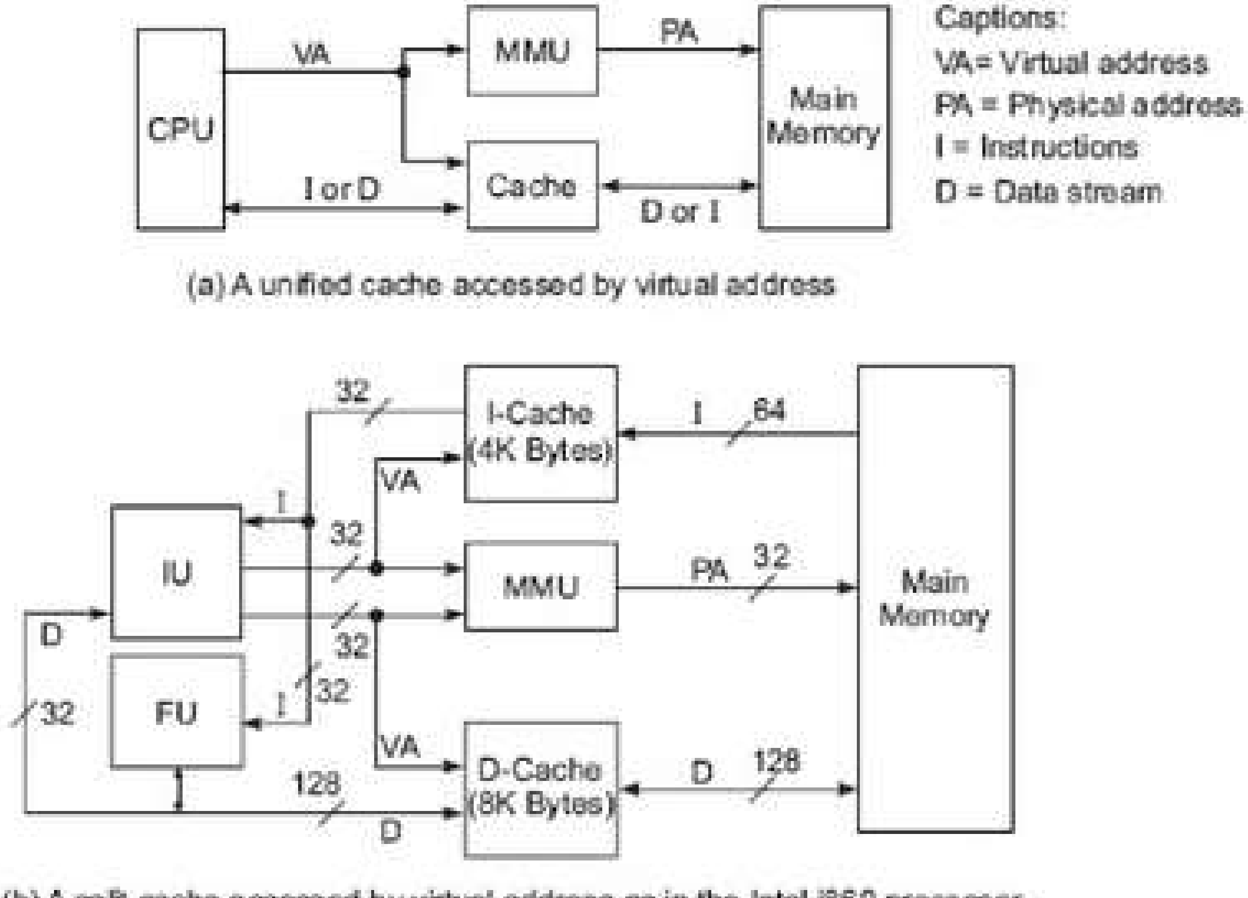
Virtual Address Caches

- When a cache is indexed or tagged with the virtual address

  Ex: Intel i486

  Adv: Accessing is faster

  Efficiency is high



(a) A unified cache accessed by virtual address

(b) A split cache accessed by virtual address as in the Intel i860 processor

Fig. 5.8 Virtual address models for unified and split. caches (Courtesy of Intel Corporation, 1989)

Aliasing Problem

- Different logically addressed data have the same index/tag in the cache
- Confusion if two or more processors access the same physical cache location
- Flush cache when aliasing occurs, but leads to slowdown
- Apply special tagging with a process key or with a physical address

Cache Mapping

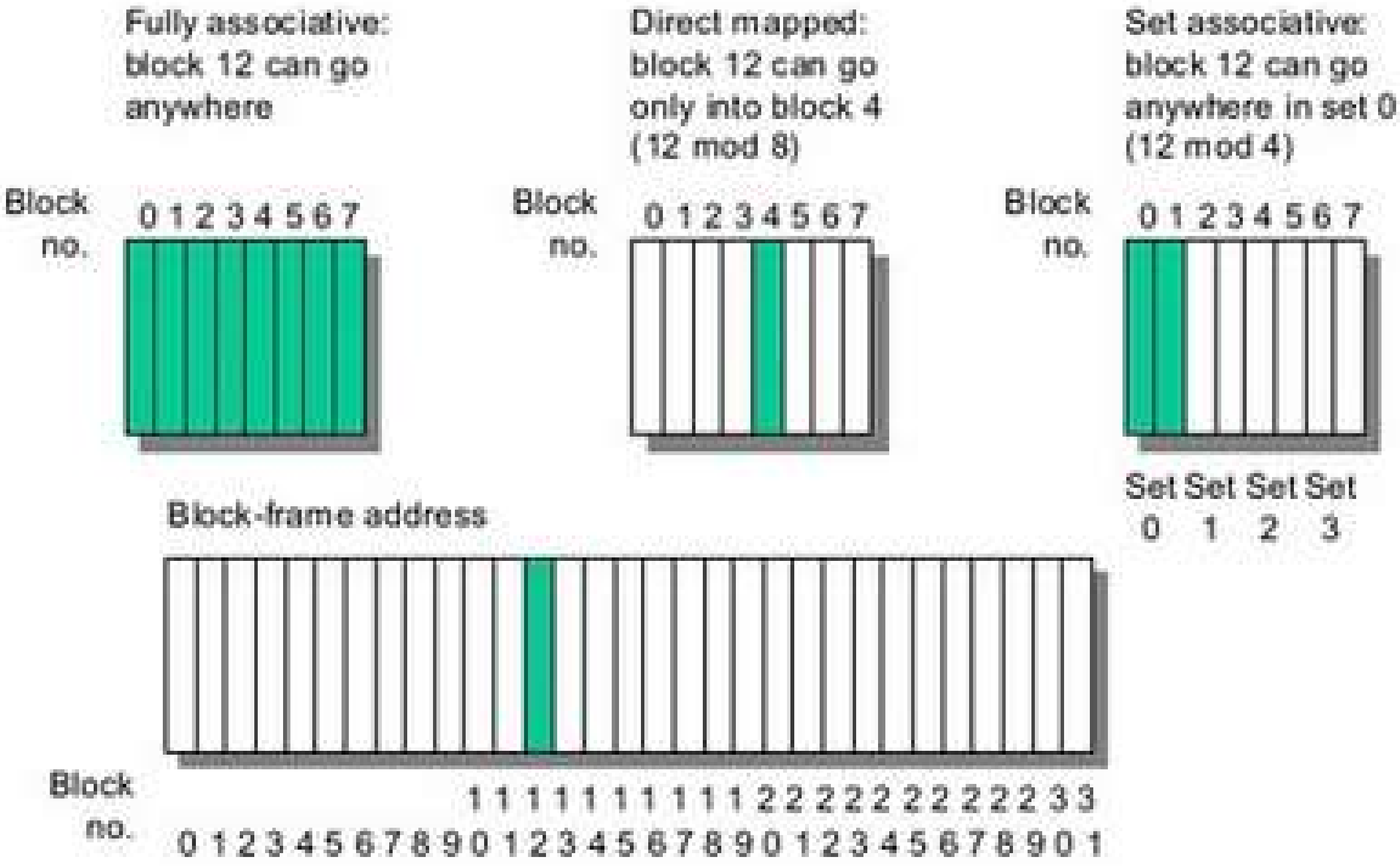
- The transfer of information from main memory to cache memory is conducted in units of cache blocks
- Four block placement schemes:

  a. Direct-mapping cache

  b. Fully-associative cache

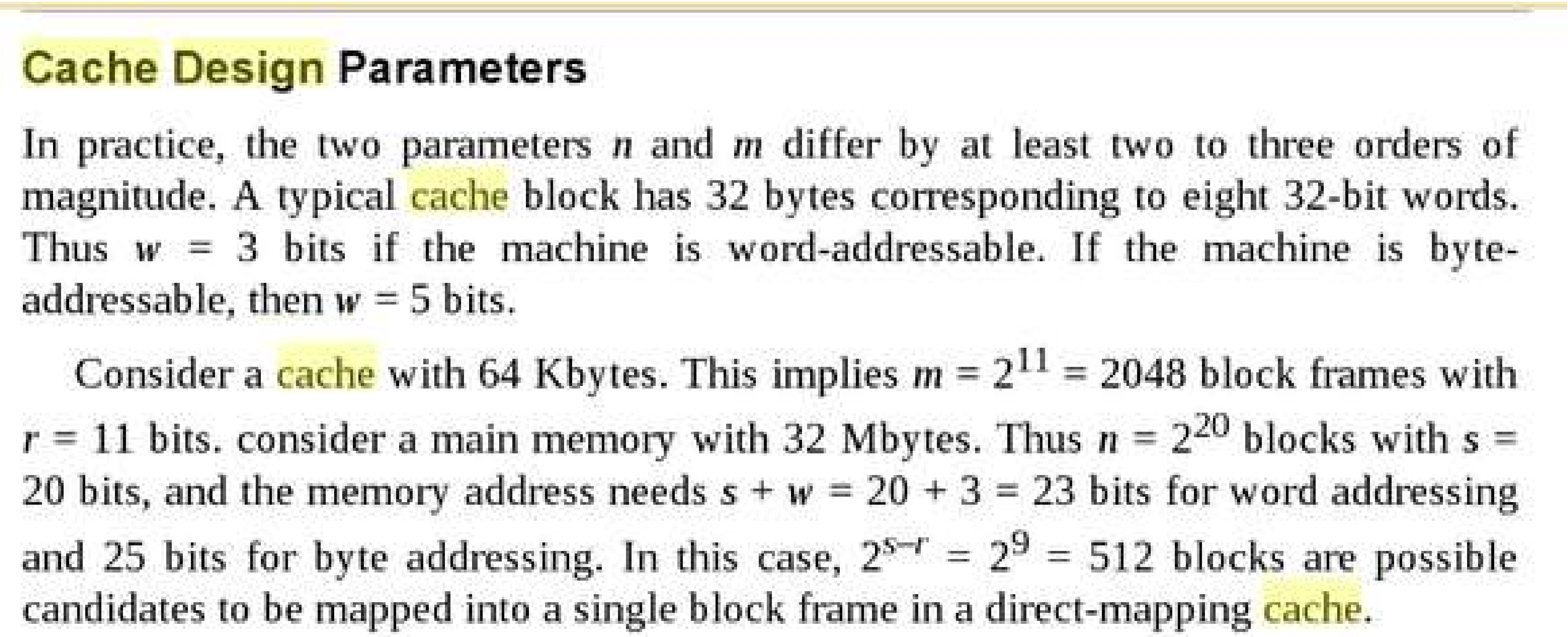
  c. Set Associative cache

  d. Sector cache


Fully associative:
block 12 can go
anywhere
Direct mapped:
block 12 can go
only into block 4
(12 mod 8)
Set associative:
block 12 can go
anywhere in set 0
(12 mod 4)
Block no.
0 1 2 3 4 5 6 7
Set 0
Set 1
Set 2
Set 3
Block-frame address
1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1

Block Placement Schemes

- Performance depends upon cache access patterns, organization, and management policy
- Blocks in caches are block frames($\underline{B}_i$), and blocks in main memory (B_j)
- $\underline{B}_i$ ($i \leq m$), B_j ($i \leq n$), $n<<m$, $n=2^s$, $m=2^r$
- Each block has b words $b=2^w$, for cache total of $mb=2^{r+w}$ words, main memory of $nb= 2^{s+w}$ words



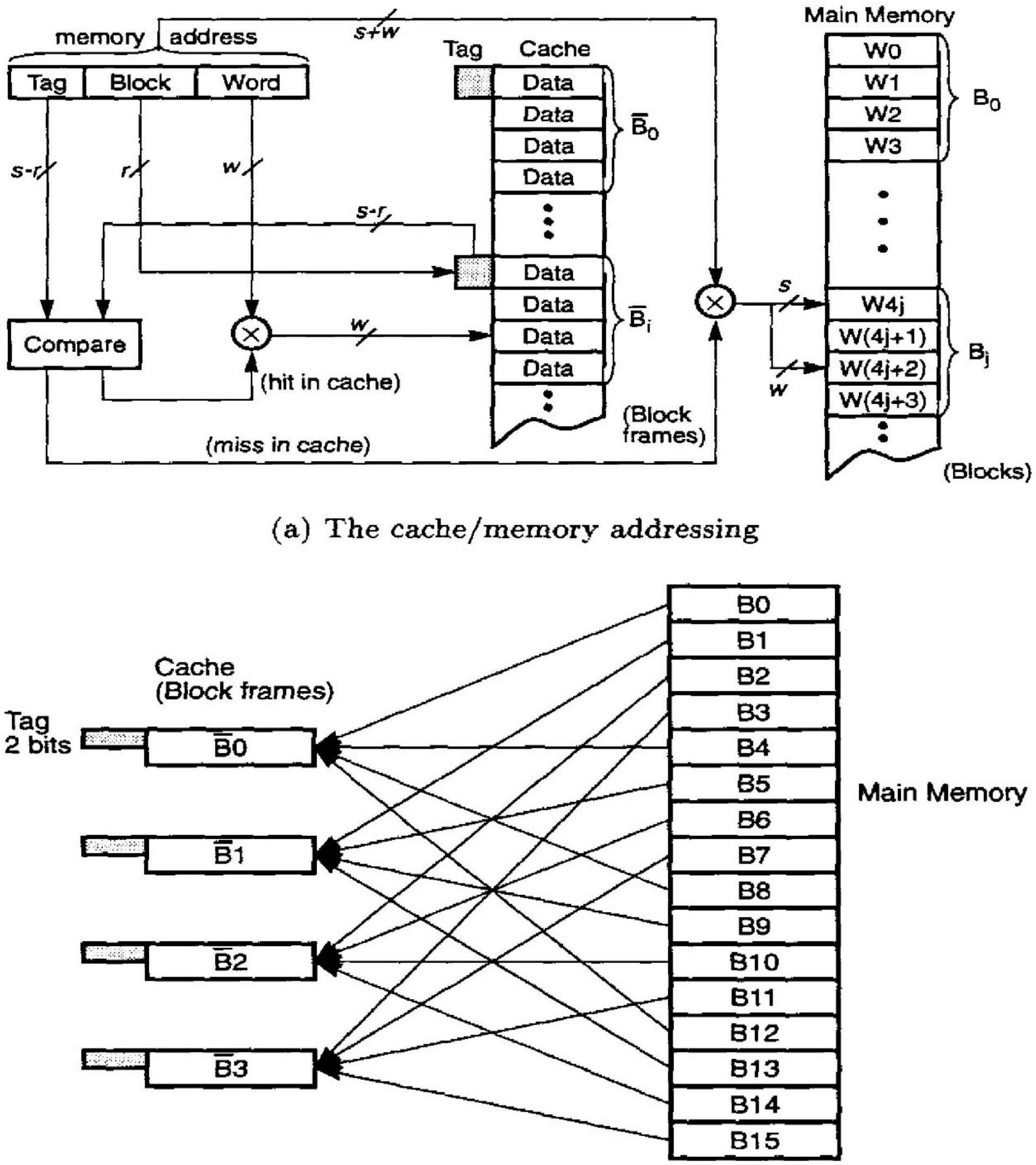
Cache Design Parameters

In practice, the two parameters n and m differ by at least two to three orders of magnitude. A typical cache block has 32 bytes corresponding to eight 32-bit words. Thus $w = 3$ bits if the machine is word-addressable. If the machine is byte-addressable, then $w = 5$ bits.

Consider a cache with 64 Kbytes. This implies $m = 2^{11} = 2048$ block frames with $r = 11$ bits. consider a main memory with 32 Mbytes. Thus $n = 2^{20}$ blocks with $s = 20$ bits, and the memory address needs $s + w = 20 + 3 = 23$ bits for word addressing and 25 bits for byte addressing. In this case, $2^{s-r} = 2^9 = 512$ blocks are possible candidates to be mapped into a single block frame in a direct-mapping cache.

Direct Mapping Cache

- Direct mapping of *n/m* memory blocks to one block frame in the cache
- Placement is by using modulo-*m* function
- $B_j \rightarrow \underline{B}_i$ if $i=j \bmod m$
- Unique block frame $\underline{B}_i$ that each B_j loads into.
- Simplest organization to implement
- No way to implement replacement policy.



(a) The cache/memory addressing

(b) Block B_j can be mapped to block frame $\bar{B}_i$ if $i = j$ (modulo 4)

Figure 5.10 Direct-mapping cache organization and a mapping example.

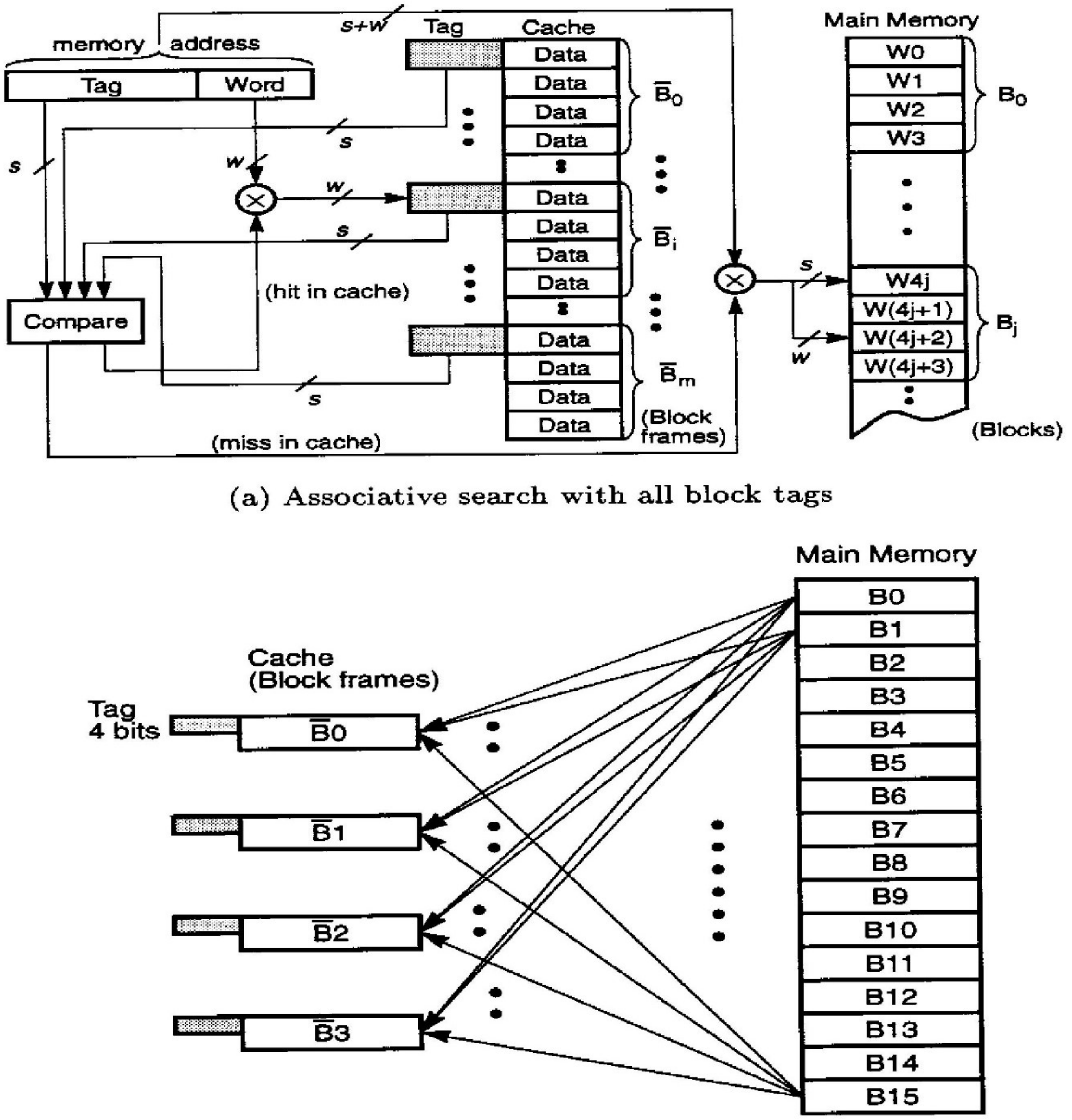
Direct Mapping Cache

- Advantages
  - Simple hardware
  - No associative search
  - No page replacement policy
  - Lower cost
  - Higher speed

- Disadvantages
  - Rigid mapping
  - Poorer hit ratio
  - Prohibits parallel virtual address translation
  - Use larger cache size with more block frames to avoid contention

Fully Associative Cache

- Each block in main memory can be placed in any of the available block frames
- *s*-bit tag needed in each cache block ($s > r$)
- An *m*-way associative search requires the tag to be compared w/ all cache block tags
- Use an associative memory to achieve a parallel comparison w/all tags concurrently



(a) Associative search with all block tags

(b) Every block is mapped to any of the four block frames identified by the tag

Figure 5.11 Fully associative cache organization and a mapping example.

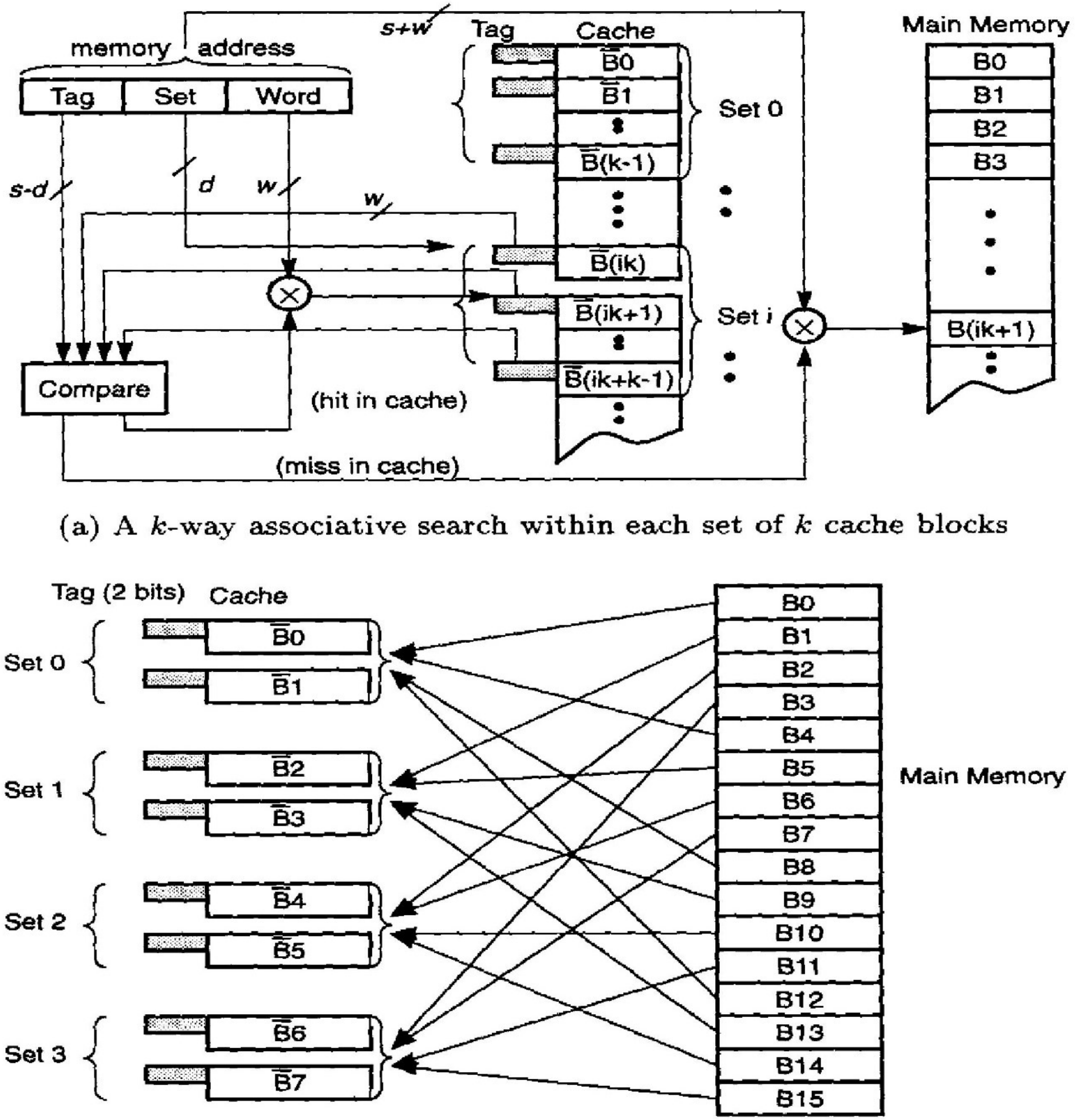
Fully Associative Caches

- Advantages:
  - Offers most flexibility in mapping cache blocks
  - Higher hit ratio
  - Allows better block replacement policy with reduced block contention

- Disadvantages:
  - Higher hardware cost
  - Only moderate size cache
  - Expensive search process

Set Associative Caches

- In a *k*-way associative cahe, the *m* cache block frames are divided into $v=m/k$ sets, with *k* blocks per set
- Each set is identified by a *d*-bit set number
  - $v=2^d$
- Compare the tag w/the *k* tags w/in the identified set
- $B_j \rightarrow \underline{B}_f \in S_i$ if $j (\mathrm{mod}\ v) = i$



(a) A k-way associative search within each set of k cache blocks

(b) Mapping cache blocks in a two-way associative cache with four sets

Figure 5.12 Set-associative cache organization and a two-way associative mapping example.

Sector Mapping Cache

- Partition cache and main memory into fixed size sectors then use fully associative search
- Use sector tags for search and block fields within sector to find block
- Only missing block loaded for a miss
- The *i*th block in a sector placed into the th block frame in a destined sector frame
- Attach a valid/invalid bit to block frames

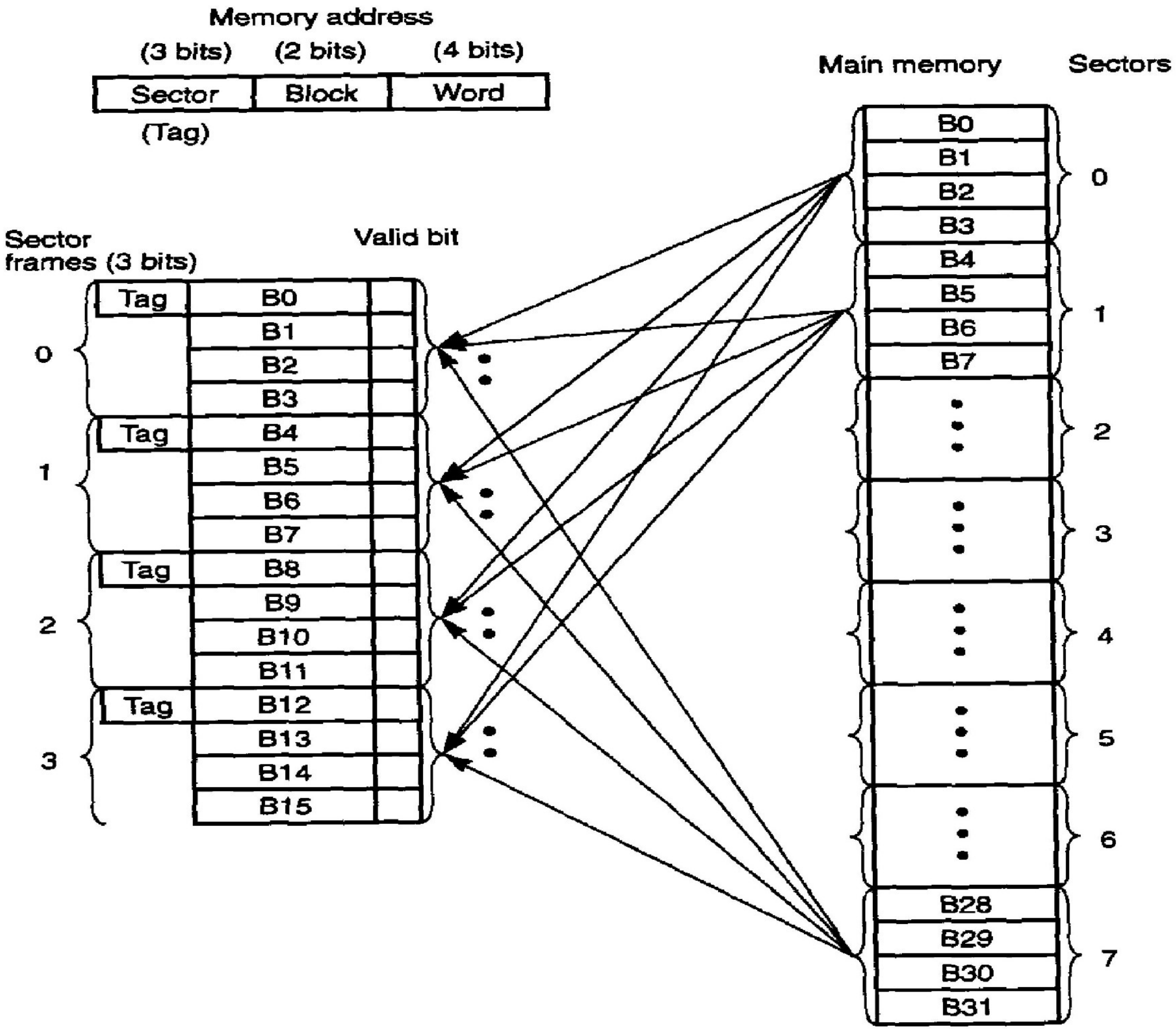


Figure 5.13 A four-way sector mapping cache organization.

Cache Performance Issues

- Cycle count: # of m/c cycles needed for cache access, update, and coherence
- Hit ratio: how effectively the cache can reduce the overall memory access time
- Program trace driven simulation: present snapshots of program behavior and cache responses
- Analytical modeling: provide insight into the underlying processes

Cycle Counts

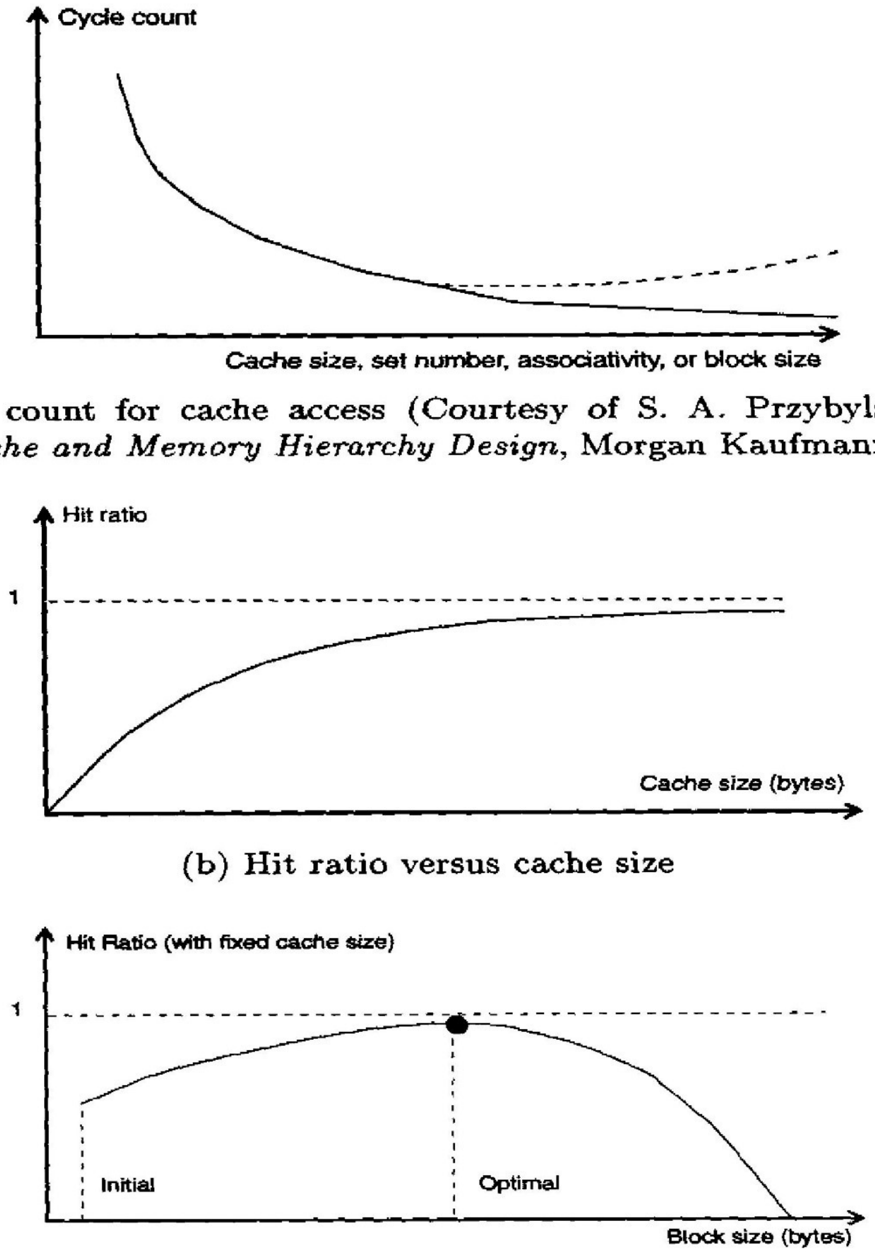
- Cache speed affected by underlying static or dynamic RAM technology, organization, and hit ratios
- Write-thru/write-back policies affect count
- Cache size, block size, set number, and associativity affect count
- Directly related to hit ratio

Hit Ratio

- Affected by cache size and block size
- Increases w.r.t. increasing cache size
- Limited cache size, initial loading, and changes in locality prevent 100% hit ratio

Effect of Block Size

- With fixed cache size, block size has impact
- As block size increases, hit ratio improves due to spatial locality
- Peaks at optimum block size, then decreases
- If too large, many words in cache not used



(a) The total cycle count for cache access (Courtesy of S. A. Przybylski; reprinted with permission from *Cache and Memory Hierarchy Design*, Morgan Kaufmann Publishers, 1990)

(b) Hit ratio versus cache size

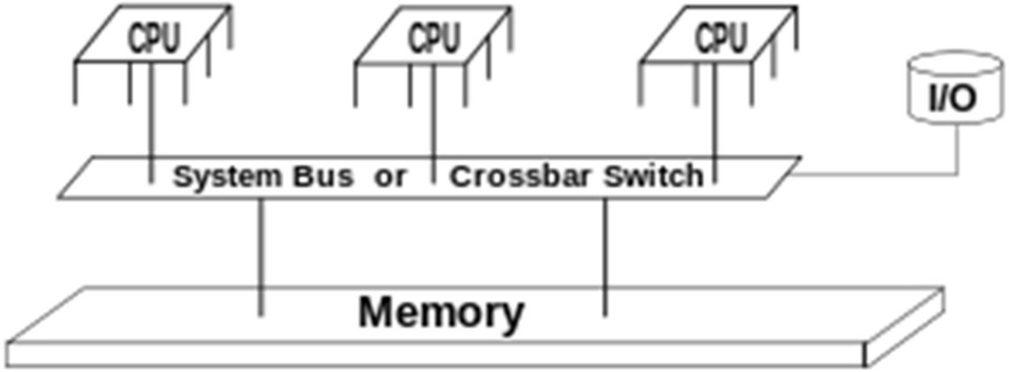
(c) Hit ratio versus block size

Figure 5.14 Cache performance versus design parameters used.

5.3 Shared memory organization

In computer science, **shared memory** is memory that may be simultaneously accessed by multiple programs with an intent to provide communication among them or avoid redundant copies.

Shared memory is an efficient means of passing data between programs. Depending on context, programs may run on a single processor or on multiple separate processors.



Characteristics of shared memory systems

- Any processor can *directly* reference any memory location.
- Communication occurs implicitly as result of loads and stores.
- Location of data in memory is transparent to the programmer.
- Inherently provided on wide range of platforms.
- Memory may be physically distributed among processors.

Interleaved Memory Organization

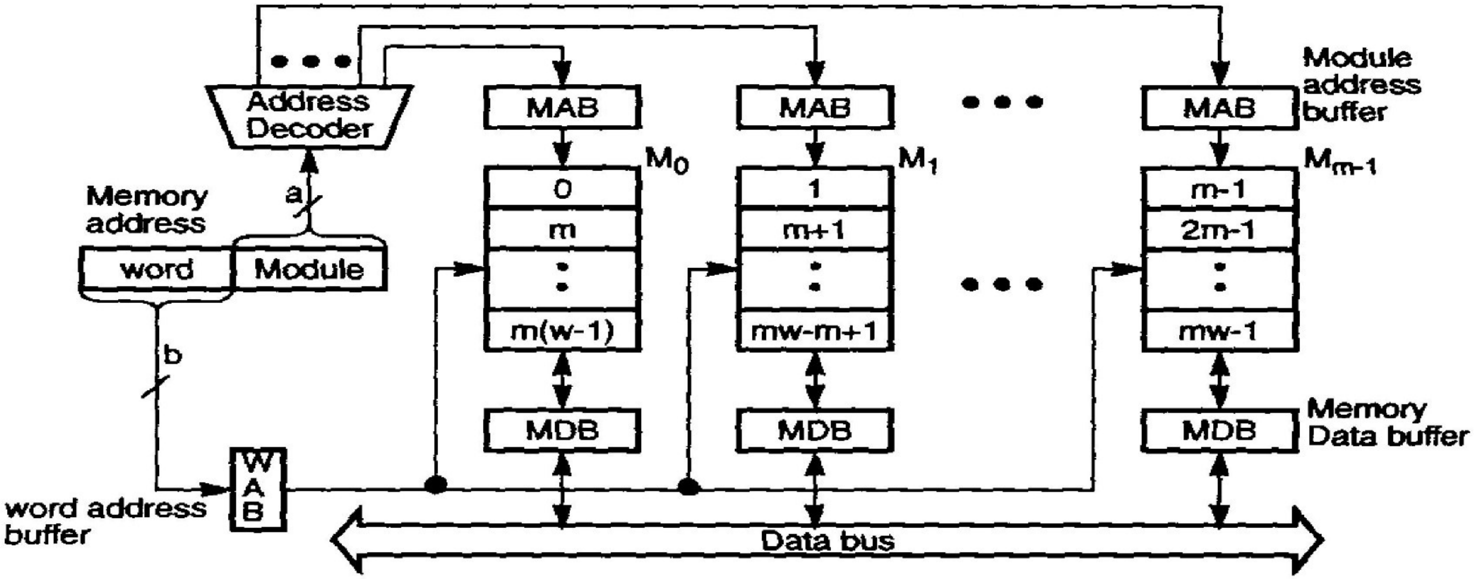
- Goal is to close the speed gap b/t CPU/cache and main memory access
- Provides higher b/w for pipelined access of contiguous memory locations

Memory Interleaving

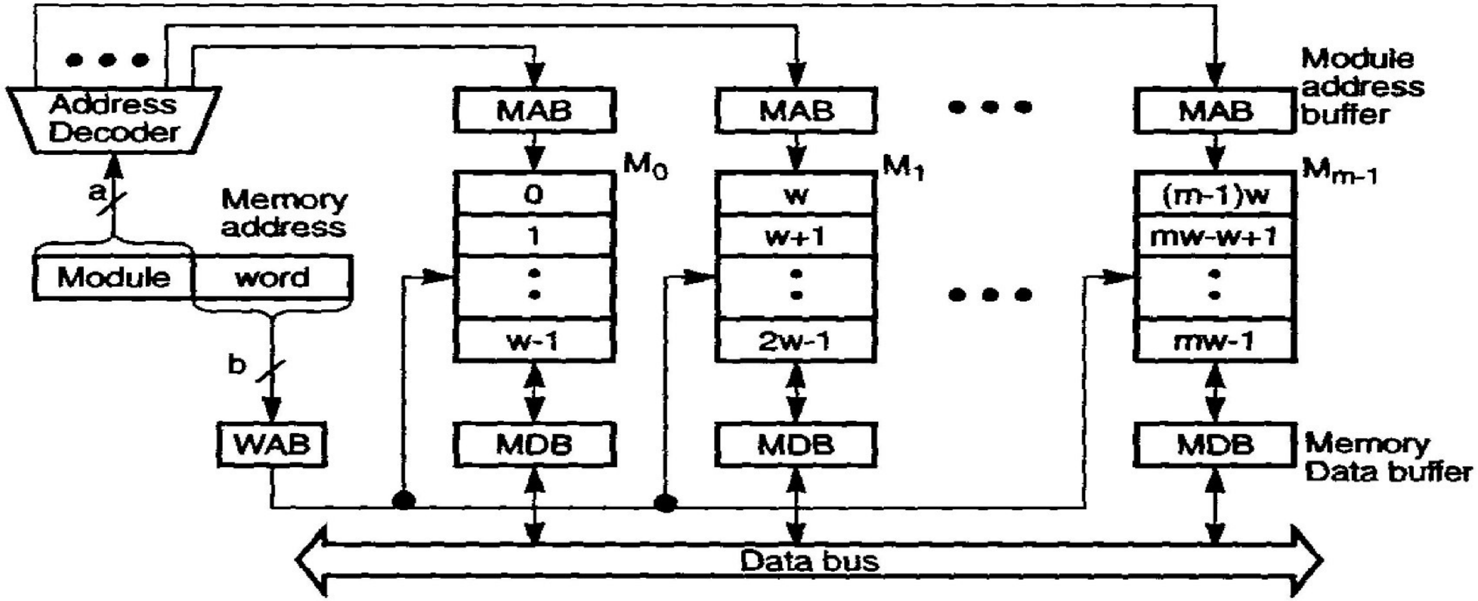
- Main memory has multiple modules connected to system bus or n/w
- Can present different addresses to different modules for parallel/pipelined access
- m=2^a modules, w/ w= 2^b words
- Varying linear address assignments
- Have random and block accesses

Addressing Formats

- *Low-order interleaving*: spread contiguous locations across modules horizontally
  - Lower *a* bits identify module, *b* for word
  - Supports block access in pipeline fashion
- *High-order:* contiguous locations within same module
  - Higher *a* bits identify module, *b* for word
  - Cannot support block access



(a) Low-order m-way interleaving (the C-access memory scheme)



(b) High-order m-way interleaving

Two interleaved memory organizations with $m = 2^a$ modules and $w = 2^b$ words per module

An eight-way interleaved memory (with $m = 8$ and $w = 8$ and thus $a = b = 3$)

ss2

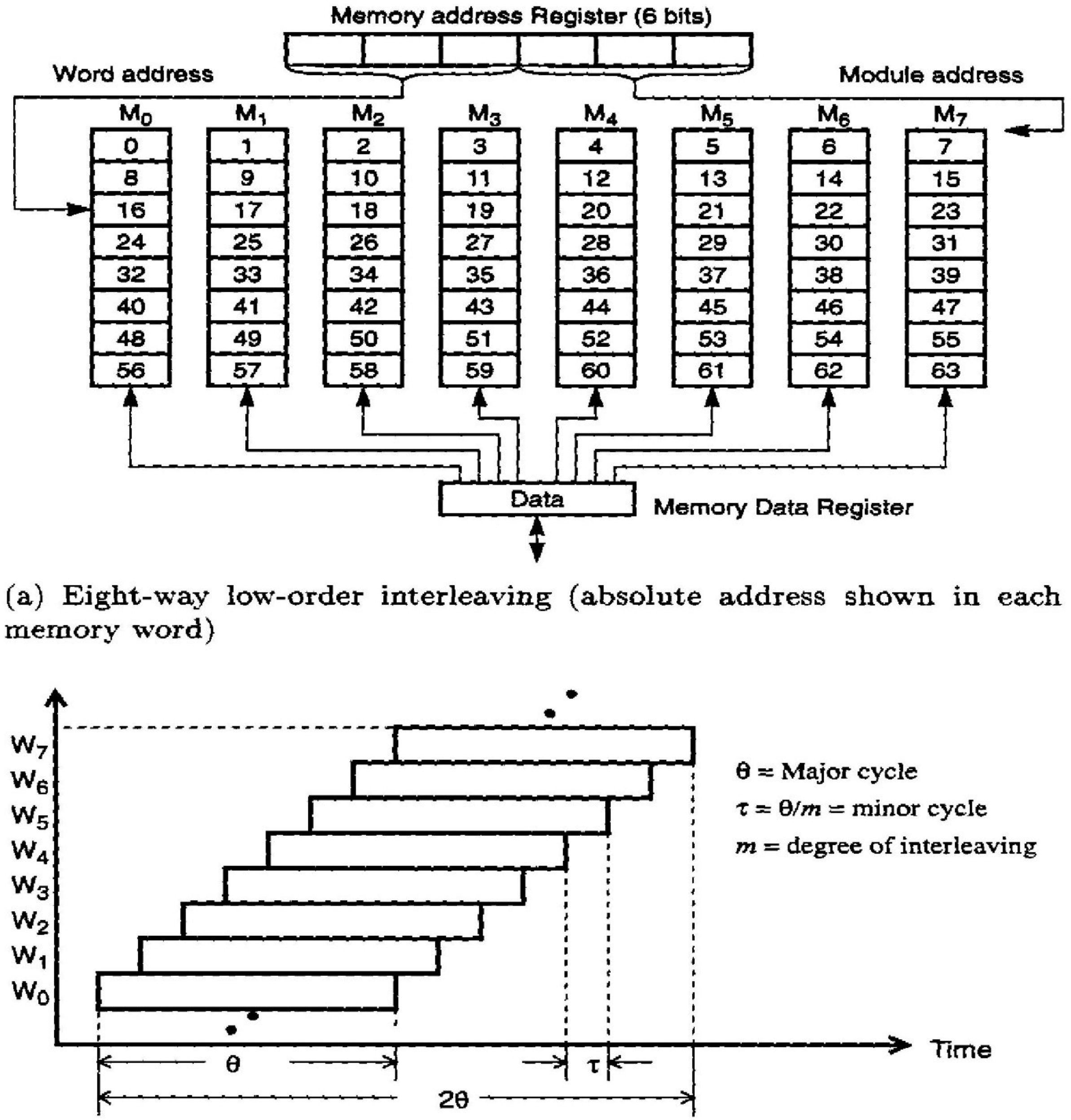
Pipelined Memory Access

- Overlap access of *m* memory modules
- *Major cycle* divided into *m minor cycles*
- $\tau = \theta/m$ *m*=degree of interleaving
- θ=total time to complete access of one word
- τ=actual time to produce one word
- Total block access time is 2θ
- Effective access time of each word is τ

Slide 75

ss2 important

sunil shetty, 11-09-2019



M_0	M_1	M_2	M_3	M_4	M_5	M_6	M_7
0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

(a) Eight-way low-order interleaving (absolute address shown in each memory word)

(b) Pipelined access of eight consecutive words in a C-access memory

Figure 5.16 Multiway interleaved memory organization and the C-access timing chart.

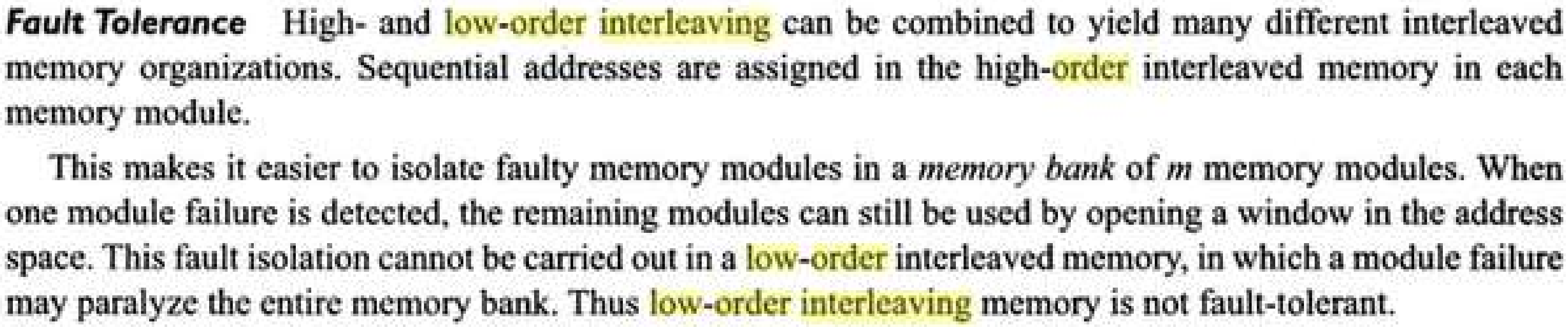
Memory Bandwidth

- Memory b/w *B* of *m*-way interleaved memory is upper bounded by *m* and lower bounded by 1
- Hellerman estimate of *B* is $B = m^{0.56} \simeq \sqrt{m}$
  - 16 modules then B is 4 times faster than single module.
- Based on single processor system, conflicts reduce it further

Avg. time to access one element in a vector

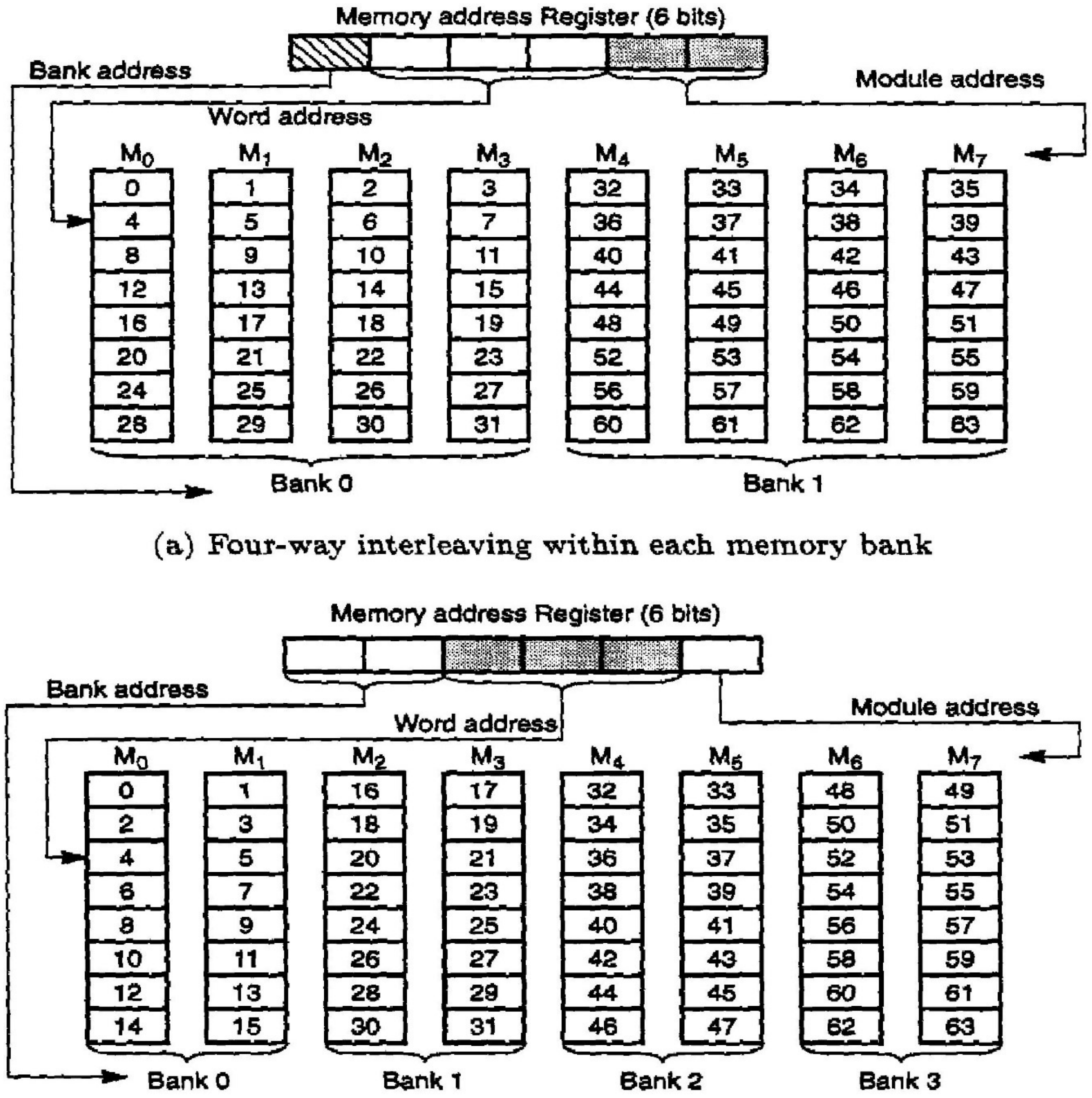
$$t_1 = \frac{\theta}{m}\left(1 + \frac{m-1}{n}\right)$$

Fault Tolerance



Fault Tolerance High- and low-order interleaving can be combined to yield many different interleaved memory organizations. Sequential addresses are assigned in the high-order interleaved memory in each memory module.

This makes it easier to isolate faulty memory modules in a *memory bank* of *m* memory modules. When one module failure is detected, the remaining modules can still be used by opening a window in the address space. This fault isolation cannot be carried out in a low-order interleaved memory, in which a module failure may paralyze the entire memory bank. Thus low-order interleaving memory is not fault-tolerant.



M0	M1	M2	M3	M4	M5	M6	M7
0	1	2	3	32	33	34	35
4	5	6	7	36	37	38	39
8	9	10	11	40	41	42	43
12	13	14	15	44	45	46	47
16	17	18	19	48	49	50	51
20	21	22	23	52	53	54	55
24	25	26	27	56	57	58	59
28	29	30	31	60	61	62	63

(a) Four-way interleaving within each memory bank

M0	M1	M2	M3	M4	M5	M6	M7
0	1	16	17	32	33	48	49
2	3	18	19	34	35	50	51
4	5	20	21	36	37	52	53
6	7	22	23	38	39	54	55
8	9	24	25	40	41	56	57
10	11	26	27	42	43	58	59
12	13	28	29	44	45	60	61
14	15	30	31	46	47	62	63

(b) Two-way interleaving within each memory bank

Figure 5.17 Bandwidth analysis of two alternative interleaved memory organizations over eight memory modules. (Absolute address shown in each memory bank.)

Higher degree of interleaving, the higher the potential memory bandwidth if the system is fault-free.

Memory Allocation Schemes

- Virtual memory allows many s/w processes time-shared use of main memory
- *Memory manager* handles the swapping
- It monitors amount of available main memory and decides which processes should reside and which to remove

Allocation Policies

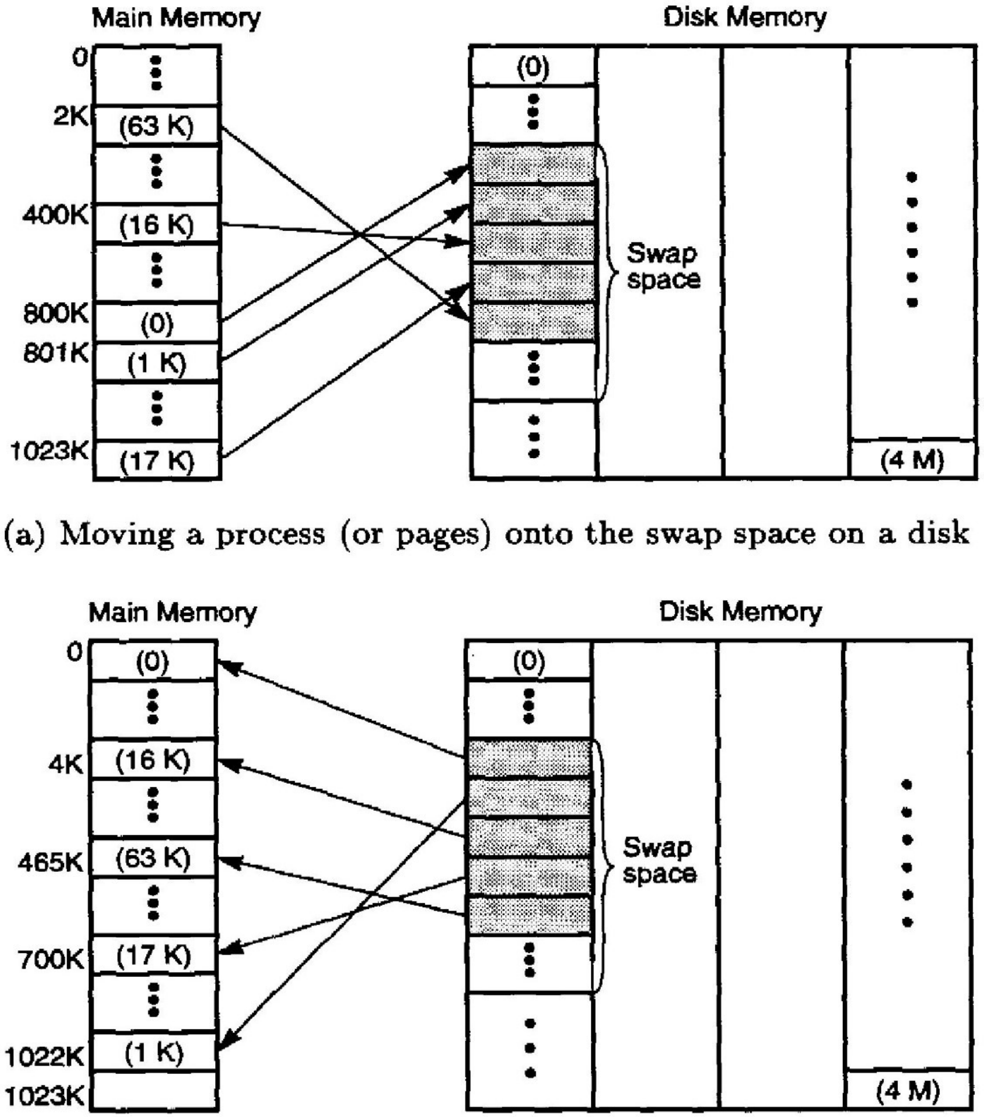
- *Memory swapping:* process of moving blocks of data between memory levels
- *Nonpreemptive allocation:* if full, then swaps out some of the allocated processes
  - Easier to implement, less efficient
- *Preemptive:*has freedom to preempt an executing process
  - More complex, expensive, and flexible

Allocation Policies

- *Local allocation:* considers only the resident working set of the faulty process
  - Used by most computers
- *Global allocation:* considers the history of the working sets of all resident processes in making a swapping decision

Swapping Systems

- Allow swapping only at entire process level
- *Swap device:* configurable section of a disk set aside for temp storage of data swapped
- *Swap space:* portion of disk set aside
- Depending on system, may swap entire processes only, or the necessary pages



(a) Moving a process (or pages) onto the swap space on a disk

(b) Swapping in a process (or pages) to the memory

Figure 5.18 The concept of memory swapping in a virtual memory hierarchy (virtual page addresses are identified by numbers within parentheses, assuming a page size of 1K words).

Swapping in UNIX

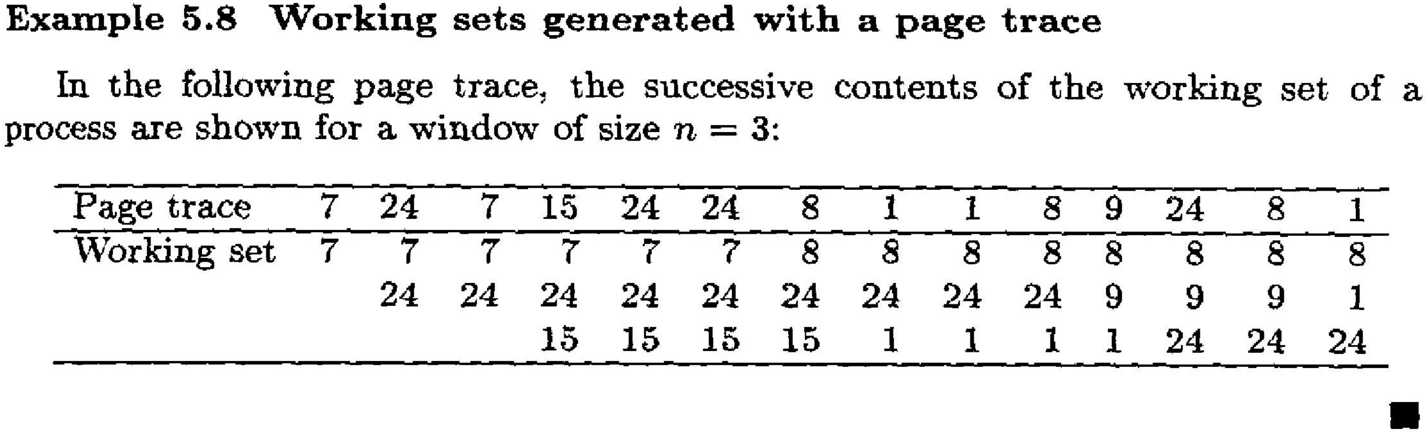
- System calls that result in a swap:
  - Allocation of space for child process being created
  - Increase in size of a process address space
  - Increased space demand by stack for a process
  - Demand for space by a returning process swapped out previously
- Special process 0 is the *swapper*

Demand Paging Systems

- Allows only pages to be transferred b/t main memory and swap device
- Pages are brought in only on demand
- Allows process address space to be larger than physical address space
- Offers flexibility to dynamically accommodate large # of processes in physical memory on time-sharing basis

Working Sets

- Set of pages referenced by the process during last *n* memory refs (*n*=window size)
- Only working sets of active processes are resident in memory



Example 5.8 Working sets generated with a page trace

In the following page trace, the successive contents of the working set of a process are shown for a window of size $n = 3$:

Page trace	7	24	7	15	24	24	8	1	1	8	9	24	8	1
Working set	7	7	7	7	7	7	8	8	8	8	8	8	8	8
		24	24	24	24	24	24	24	24	24	9	9	9	1
				15	15	15	15	1	1	1	1	24	24	24

■

Other Policies

- *Hybrid memory systems* combine advantages of swapping and demand paging
- *Anticipatory paging* prefetches pages based on anticipation
  - Difficult to implement

Memory Consistency/Inconsistency

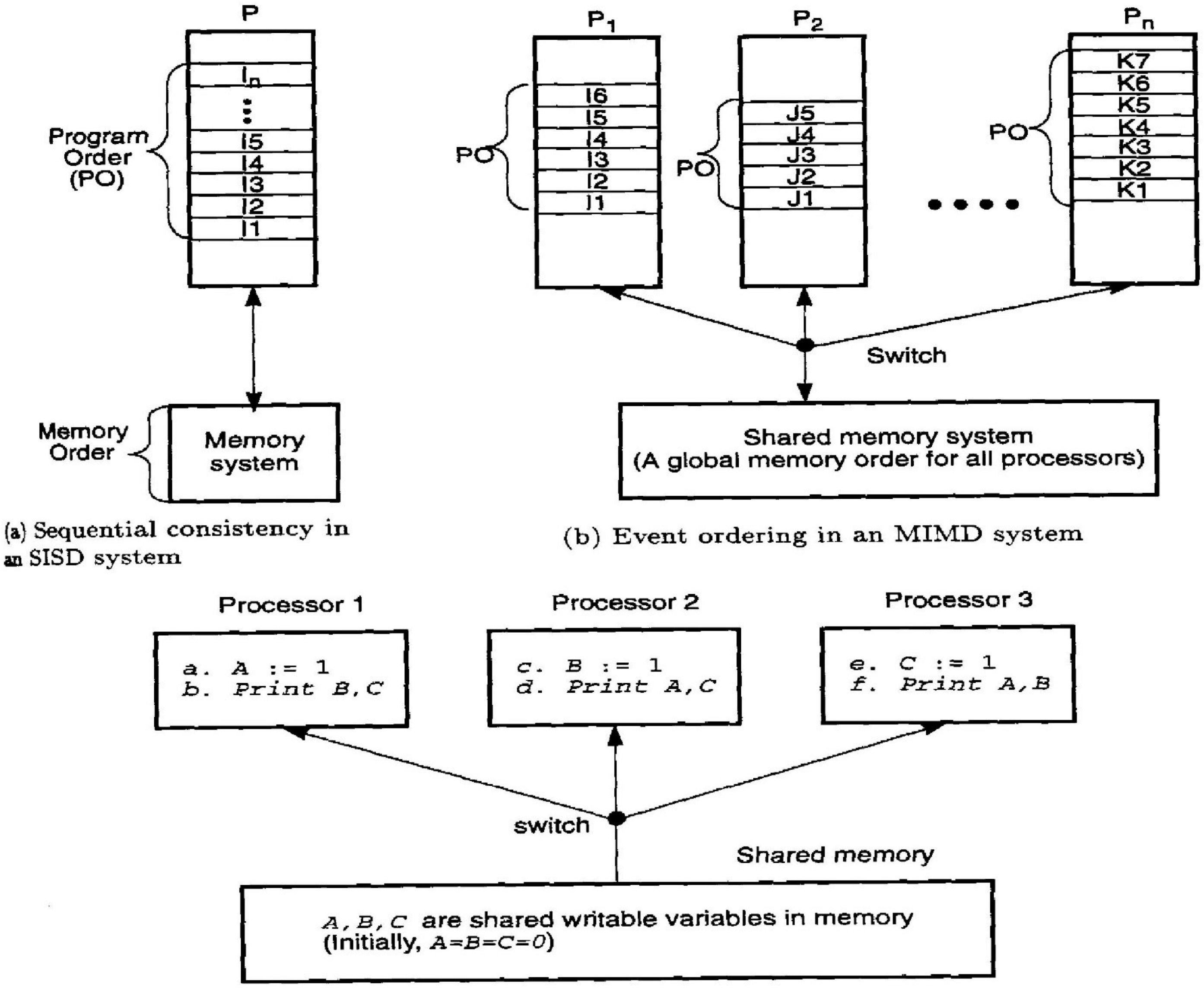
- *Memory inconsistency:* when memory access order differs from program execution order
- *Sequential consistency*: memory accesses (I and D) consistent with program execution order

Memory Consistency Issues

- *Memory model:* behavior of a shared memory system as observed by processors
- Choosing a memory model – compromise b/t a strong model minimally restricting s/w and a weak model offering efficient implementation
- Primitive memory ops: load, store, swap

Event Orderings

- *Processes:* concurrent instruction streams executing on different processors
- Consistency models specify the order by which events from one process should be observed by another
- *Event ordering* helps determine if a memory event is legal for concurrent accesses
- *Program order:* order by which memory access occur for execution of a single process, w/o any reordering



```
a. A := 1
b. Print B,C
```

```
c. B := 1
d. Print A,C
```

```
e. C := 1
f. Print A,B
```

(a) Sequential consistency in an SISD system

(b) Event ordering in an MIMD system

(c) A parallel program for Example 5.9

Figure 5.19 The access ordering of memory events in a uniprocessor and in a multiprocessor, respectively. (Courtesy of Dubois and Briggs, *Tutorial Notes on Shared-Memory Multiprocessors*, Int. Symp. Computer Arch., May 1990)

Primitive Memory Operations

- *Load* by P_i complete wrt P_k when issue of a *store* to same location by P_k does not affect value returned by *load*
- *Store* by P_i complete wrt P_k when an issued *load* to same address by P_k returns the value by this *store*
- *Load* is *globally* performed if it is performed wrt all processors and if the *store* that is the source of the returned value has been performed wrt to all

Difficulty in Maintaining Correctness on an MIMD

- If no synch. among instruction streams, then large # of different instruction interleavings
- Could change execution order, leading to more possibilities
- If accesses are not atomic, then different processors can observe different interleavings

Atomicity

- Categories of memory behavior:
  - Program order preserved and uniform observation sequence by all processors
  - Out of program order allowed and uniform observation sequence by all processors
  - Out of program order allowed and nonuniform sequences observed by different processors

Atomicity

- *Atomic memory accesses:* memory updates are known to all processors at the same time
- *Non-atomic:* having individual program orders that conform is not a sufficient condition for sequential consistency
  - Multiprocessor cannot be strongly ordered

Lamport’s Definition of Sequential Consistency

- A multiprocessor system is *sequentially consistent* if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program

Sequential Consistency

- Sufficient conditions:
  - Before a *load* is allowed to perform wrt any other processor, all previous *loads* must be globally performed and all previous *stores* must be performed wrt all processors
  - Before a *store* is allowed to perform wrt any other processor, all previous *loads* must be globally performed and all previous *stores* must be performed wrt to all processors

Sequential Consistency Axioms

- The memory order conforms to a total binary order in which shared memory is accessed in real time over all *loads*/*stores*
- A *load* always returns the value written by the latest *store* to the same location
- If 2 ops appear in particular program order, same memory order
- *Swap* op is atomic with respect to *stores*. No other *store* can intervene b/t *load* and *store* parts of *swap*
- All *stores* and *swaps* must eventually terminate

Implementation Considerations

- A single port s/w services one op at a time
- Order in which s/w is thrown determines global order of memory access ops
- *Strong ordering* preserves the program order in all processors
- Sequential consistency model leads to poor memory performance due to the imposed strong ordering of memory events

Weak Consistency Models

- Multiprocessor model may range from strong (sequential) consistency to various degrees of weak consistency
- Two models considered
  - DSB model
  - TSO model

DSB Model

- All previous *synch.* accesses performed before *load* or *store* allowed
- All previous *load/stores* performed before a *synch.* allowed
- *Synch.* accesses sequentially consistent with respect to one another

TSO Model

- *Load* returns latest *store* result
- Memory order is a total binary relation over all pairs of *store* ops
- If 2 *stores* appear in part. program order, same memory order
- If a mem op follows a *load* in prog order, must also follow *load* in mem order
- *Swap* op atomic with respect to other *stores* – no other *store* can interleave b/t *load/store* parts of *swap*
- All *stores/swaps* must eventually terminate

CSE539: Advanced Computer Architecture

Chapter 6
Pipelining and Superscalar Techniques

Book: “Advanced Computer Architecture – Parallelism, Scalability, Programmability”, Hwang & Jotwani

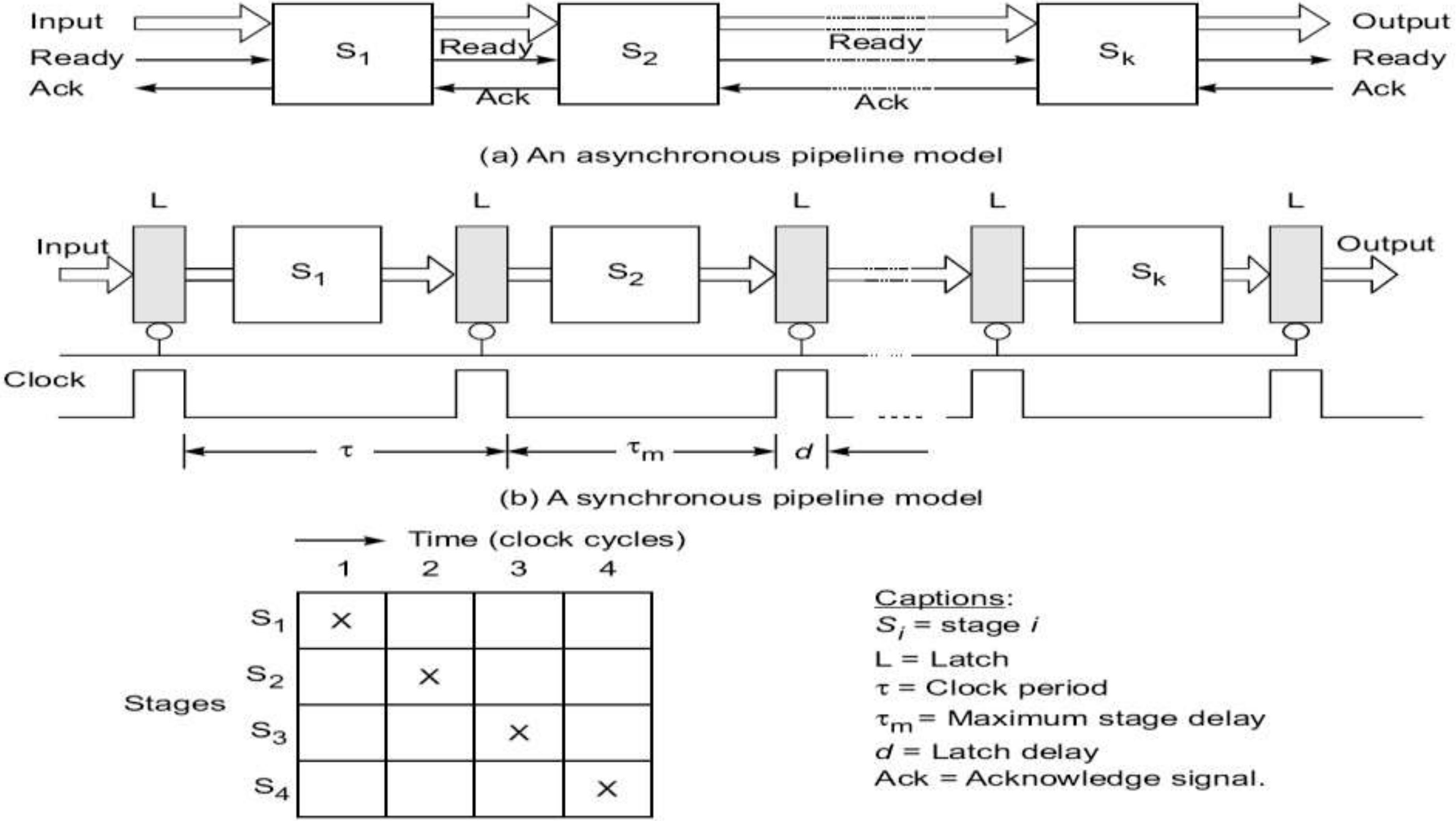
Sumit Mittu
Assistant Professor, CSE/IT
Lovely Professional University
sumit.12735@lpu.co.in

In this chapter…

- Linear Pipeline Processors
- Non-linear Pipeline Processors
- Instruction Pipeline Design
- Arithmetic Pipeline Design
- Superscalar Pipeline Design

LINEAR PIPELINE PROCESSORS

- Linear Pipeline Processor
  - *(Definition)*
- Models of Linear Pipeline
  - Synchronous Model
  - Asynchronous Model
  - *(Corresponding reservation tables)*
- Clocking and Timing Control
  - Clock Cycle
  - Pipeline Frequency
  - Clock skewing
  - Flow-through delay
  - Speedup, Efficiency and Throughput
- Optimal number of Stages and Performance-Cost Ratio (PCR)



(a) An asynchronous pipeline model

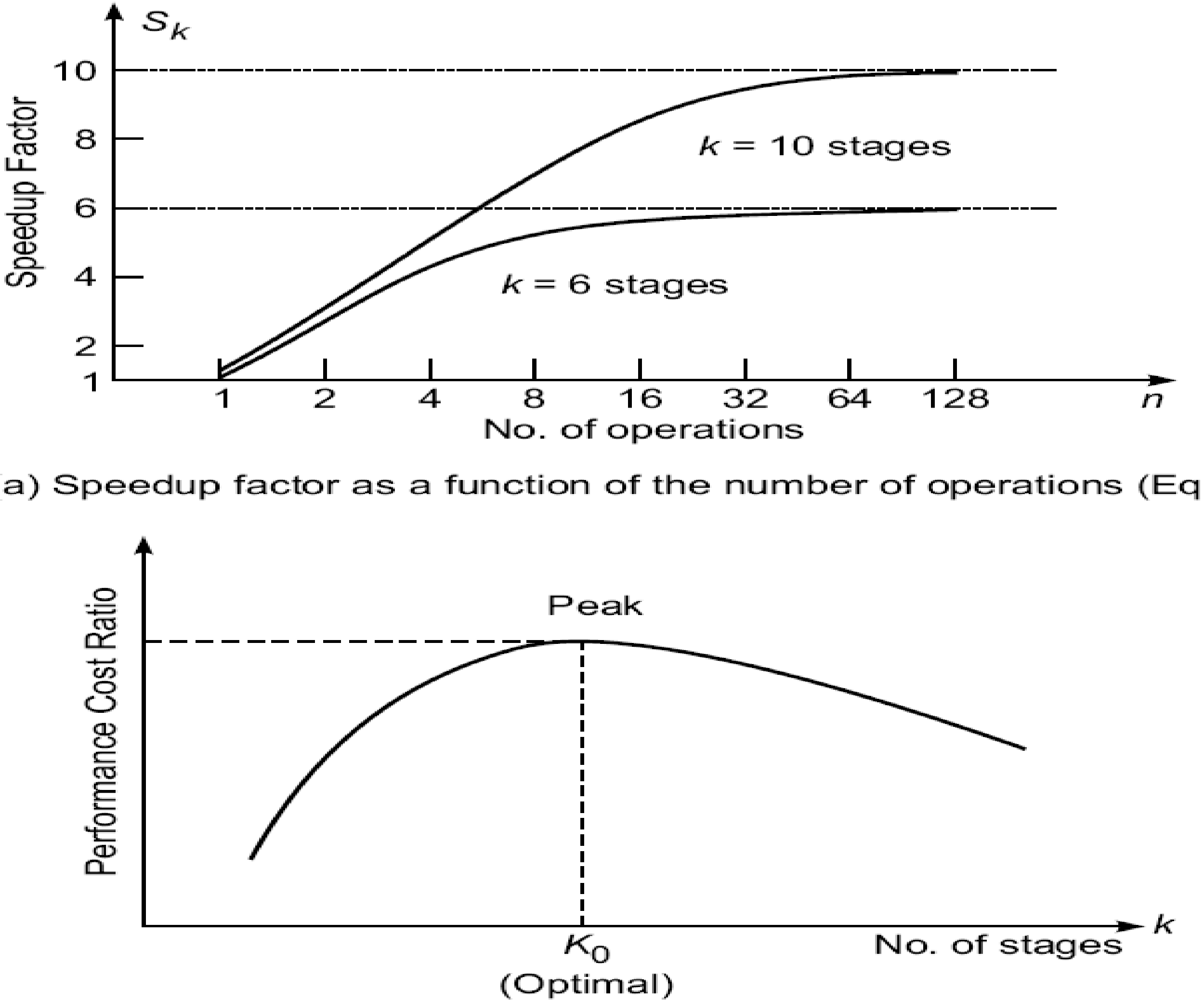
(b) A synchronous pipeline model

	1	2	3	4
S_1	×			
S_2		×		
S_3			×	
S_4				×

Captions:
S_i = stage i
L = Latch
τ = Clock period
τ_m = Maximum stage delay
d = Latch delay
Ack = Acknowledge signal.

(c) Reservation table of a four-stage linear pipeline

Fig. 6.1 Two models of linear pipeline units and the corresponding reservation table



(a) Speedup factor as a function of the number of operations (Eq. 6.5)

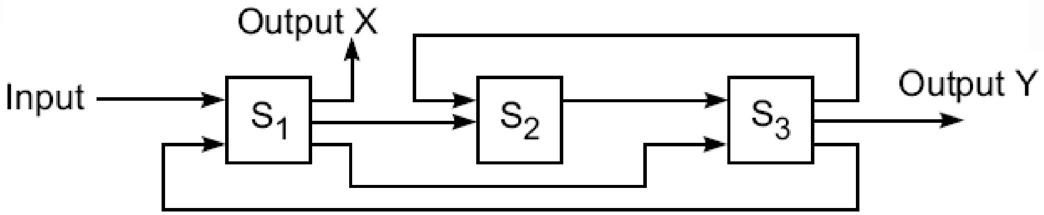
(b) Optimal number of pipeline stages (Eqs. 6.6 and 6.7)

Fig. 6.2 Speedup factors and the optimal number of pipeline stages for a linear pipeline unit

NON-LINEAR PIPELINE PROCESSORS

- Dynamic Pipeline
  - Static v/s Dynamic Pipeline
  - Streamline connection, feed-forward connection and feedback connection
- Reservation and Latency Analysis
  - Reservation tables
  - Evaluation time
- Latency Analysis
  - Latency
  - Collision
  - Forbidden latencies
  - Latency Sequence, Latency Cycle and Average Latency

NON-LINEAR PIPELINE PROCESSORS



(a) A three-stage pipeline

Time →

Stages	1	2	3	4	5	6	7	8
S_1	X					X		X
S_2		X		X				
S_3			X		X		X	

(b) Reservation table for function X

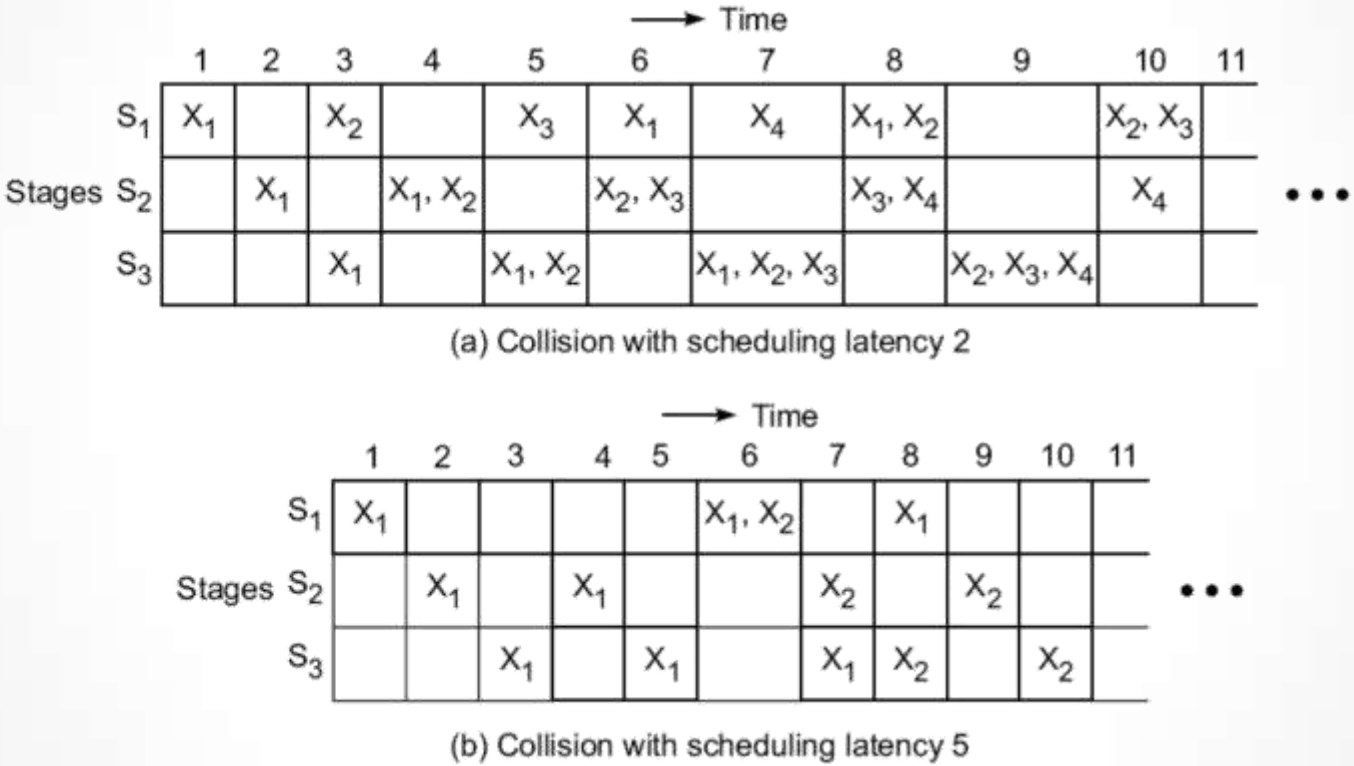
Time →

Stages	1	2	3	4	5	6
S_1	Y				Y	
S_2			Y			
S_3		Y		Y		Y

(c) Reservation table for function Y

Fig. 6.3 A dynamic pipeline with feed forward and feedback connections for two different functions

NON-LINEAR PIPELINE PROCESSORS



→ Time

Stages	1	2	3	4	5	6	7	8	9	10	11
S_1	X_1		X_2		X_3	X_1	X_4	X_1, X_2		X_2, X_3	
S_2		X_1		X_1, X_2		X_2, X_3		X_3, X_4		X_4	
S_3			X_1		X_1, X_2		X_1, X_2, X_3		X_2, X_3, X_4		

• • •

(a) Collision with scheduling latency 2

→ Time

Stages	1	2	3	4	5	6	7	8	9	10	11
S_1	X_1					X_1, X_2		X_1			
S_2		X_1		X_1			X_2		X_2		
S_3			X_1		X_1		X_1	X_2		X_2	

• • •

(b) Collision with scheduling latency 5

Fig. 6.4 Collisions with forbidden latencies 2 and 5 in using the pipeline in Fig. 6.3 to evaluate the function X

Cycle repeats

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
S_1	X_1	X_2				X_1	X_2	X_1	X_2	X_3	X_4				X_3	X_4	X_3	X_4	X_5	X_6	
S_2		X_1	X_2	X_1	X_2						X_3	X_4	X_3	X_4						X_5	•••
S_3			X_1	X_2	X_1		X_1	X_2				X_3	X_4	X_3		X_3					

(a) Latency cycle (1, 8) = 1, 8, 1, 8, 1, 8, ..., with an average latency of 4.5

Cycle repeats

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
S_1	X_1			X_2		X_1	X_3	X_1	X_2	X_4	X_2	X_3	X_5	X_3	X_4	X_6	X_4	X_5	X_7	X_5	
S_2		X_1		X_1	X_2		X_2	X_3		X_3	X_4		X_4	X_5		X_5	X_6		X_6	X_7	•••
S_3			X_1		X_1	X_2	X_1	X_2	X_3	X_2	X_3	X_4	X_3	X_4	X_5	X_4	X_5	X_6	X_5	X_6	

(b) Latency cycle (3) = 3, 3, 3, 3, ..., with an average latency of 3

Cycle repeats

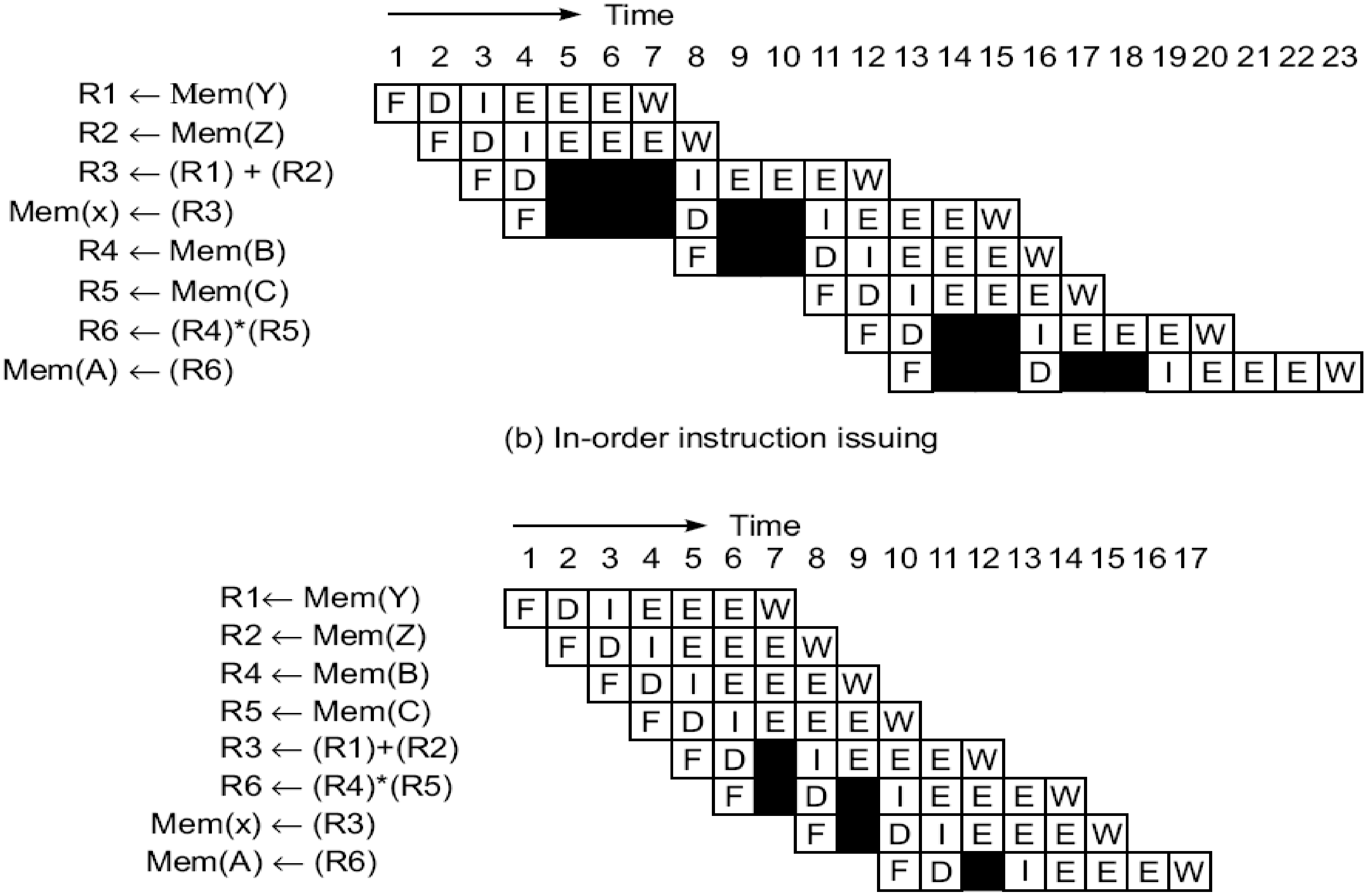
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
S_1	X_1					X_1	X_2	X_1				X_2	X_3	X_2				X_3	X_4	X_3	
S_2		X_1		X_1				X_2		X_2				X_3		X_3				X_4	•••
S_3			X_1		X_1		X_1		X_2		X_2		X_2		X_3	X_3		X_3		X_3	

(c) Latency cycle (6) = 6, 6, 6, 6, ..., with an average latency of 6

Fig. 6.5 Three valid latency cycles for the evaluation of function X

INSTRUCTION PIPELINE DESIGN

- Instruction Execution Phases
  - E.g. Fetch, Decode, Issue, Execute, Write-back
  - In-order Instruction issuing and Reordered Instruction issuing
    - E.g. **X = Y + Z , A = B x C**
- Mechanisms/Design Issues for Instruction Pipelining
  - Pre-fetch Buffers
  - Multiple Functional Units
  - Internal Data Forwarding
  - Hazard Avoidance
- Dynamic Scheduling
- Branch Handling Techniques



Time

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
R1 ← Mem(Y)	F	D	I	E	E	E	W																
R2 ← Mem(Z)		F	D	I	E	E	E	W															
R3 ← (R1) + (R2)			F	D	■	■	■	I	E	E	E	W											
Mem(x) ← (R3)				F	■	■	■	D	■	■	I	E	E	E	W								
R4 ← Mem(B)								F	■	■	D	I	E	E	E	W							
R5 ← Mem(C)											F	D	I	E	E	E	W						
R6 ← (R4)*(R5)												F	D	■	■	I	E	E	E	W			
Mem(A) ← (R6)													F	■	■	D	■	■	I	E	E	E	W

(b) In-order instruction issuing

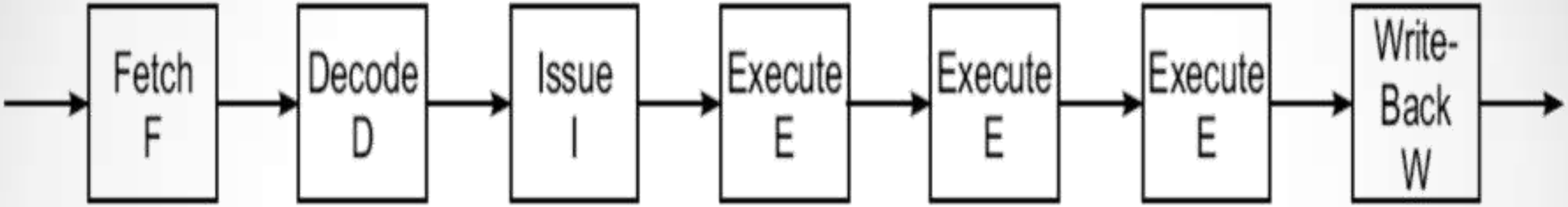
Time

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
R1← Mem(Y)	F	D	I	E	E	E	W										
R2 ← Mem(Z)		F	D	I	E	E	E	W									
R4 ← Mem(B)			F	D	I	E	E	E	W								
R5 ← Mem(C)				F	D	I	E	E	E	W							
R3 ← (R1)+(R2)					F	D	■	I	E	E	E	W					
R6 ← (R4)*(R5)						F	■	D	■	I	E	E	E	W			
Mem(x) ← (R3)								F	■	D	I	E	E	E	W		
Mem(A) ← (R6)										F	D	■	I	E	E	E	W

(c) Reordered instruction issuing

Fig. 6.9 Pipelined execution of X = Y + Z and A = B × C (Courtesy of James Smith; reprinted with permission from *IEEE Computer*, July 1989)

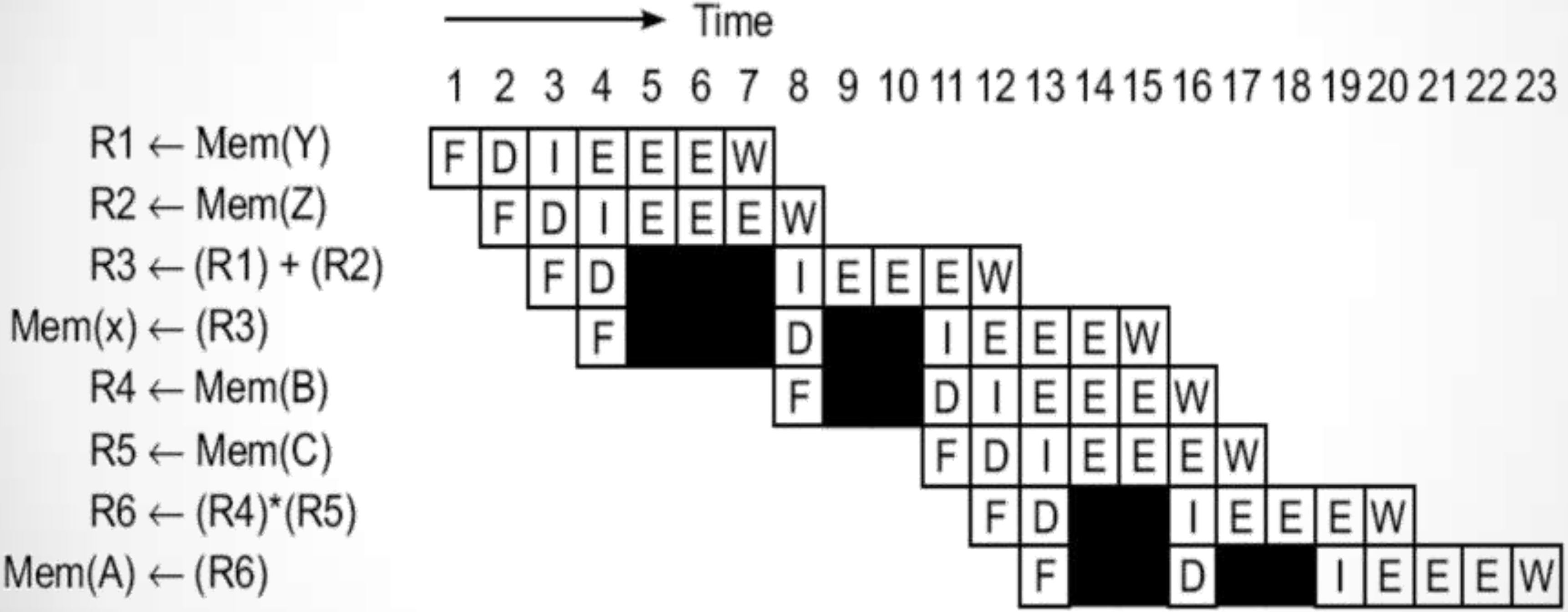
INSTRUCTION PIPELINE DESIGN



(a) A seven-stage instruction pipeline

- **Fetch**: *fetches instructions from memory; ideally one per cycle*
- **Decode**: *reveals instruction operations to be performed and identifies the resources needed*
- **Issue**: *reserves the resources and reads the operands from registers*
- **Execute**: *actual processing of operations as indicated by instruction*
- **Write Back**: *writing results into the registers*

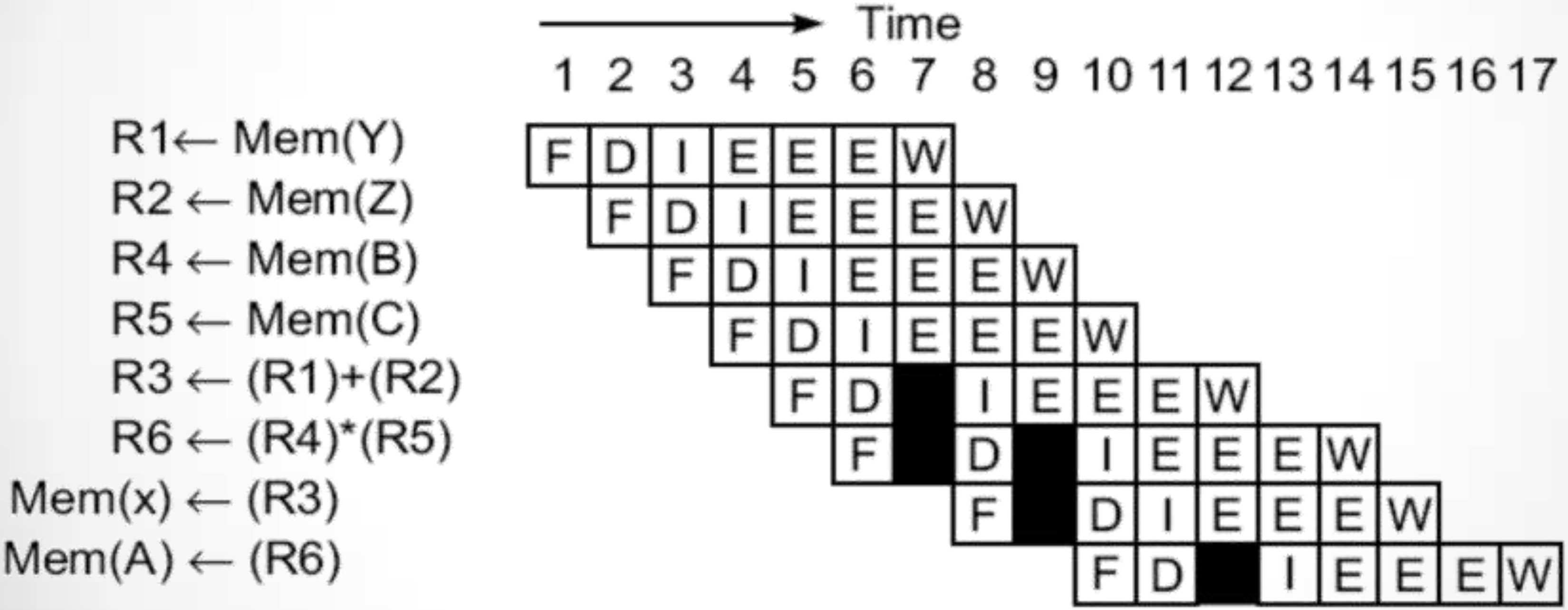
INSTRUCTION PIPELINE DESIGN



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
R1 ← Mem(Y)	F	D	I	E	E	E	W																
R2 ← Mem(Z)		F	D	I	E	E	E	W															
R3 ← (R1) + (R2)			F	D	■	■	■	I	E	E	E	W											
Mem(x) ← (R3)				F	■	■	■	D	■	■	I	E	E	E	W								
R4 ← Mem(B)								F	■	■	D	I	E	E	E	W							
R5 ← Mem(C)											F	D	I	E	E	E	W						
R6 ← (R4)*(R5)												F	D	■	■	I	E	E	E	W			
Mem(A) ← (R6)													F	■	■	D	■	■	I	E	E	E	W

(b) In-order instruction issuing

INSTRUCTION PIPELINE DESIGN



Time →

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
R1← Mem(Y)	F	D	I	E	E	E	W										
R2 ← Mem(Z)		F	D	I	E	E	E	W									
R4 ← Mem(B)			F	D	I	E	E	E	W								
R5 ← Mem(C)				F	D	I	E	E	E	W							
R3 ← (R1)+(R2)					F	D	■	I	E	E	E	W					
R6 ← (R4)*(R5)						F	■	D	■	I	E	E	E	W			
Mem(x) ← (R3)								F	■	D	I	E	E	E	W		
Mem(A) ← (R6)										F	D	■	I	E	E	E	W

(c) Reordered instruction issuing

INSTRUCTION PIPELINE DESIGN
Mechanisms/Design Issues of Instruction Pipeline

- Pre-fetch Buffers
  - Sequential Buffers
  - Target Buffers
  - Loop Buffers

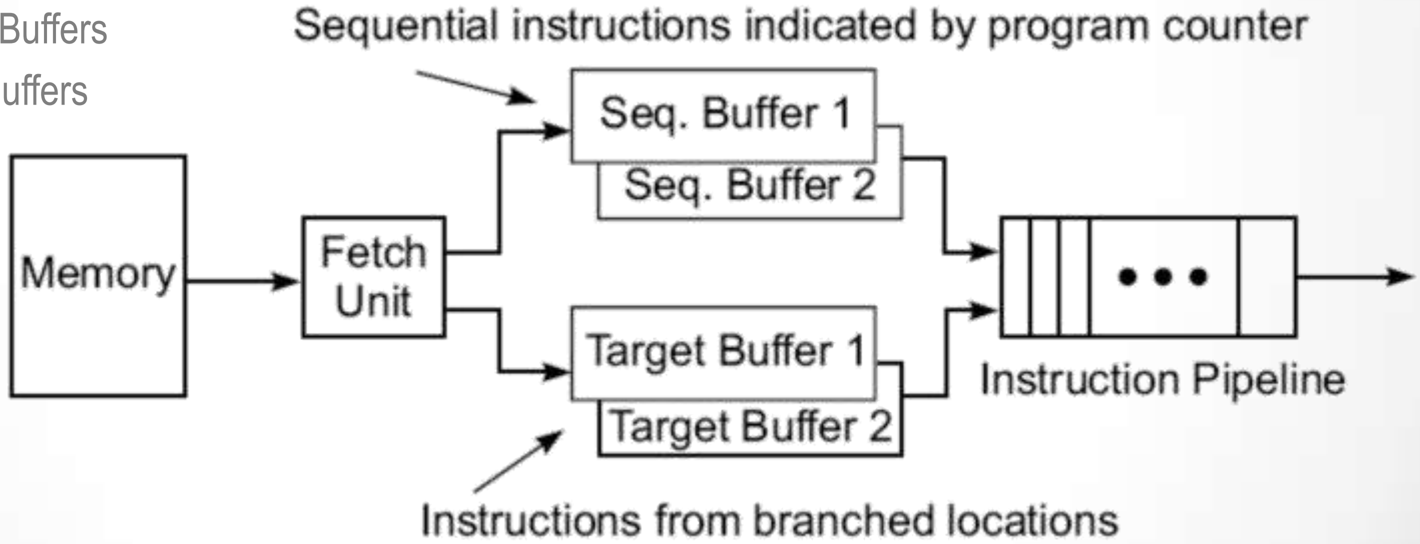


Fig. 6.11 The use of sequential and target buffers

INSTRUCTION PIPELINE DESIGN

Mechanisms/Design Issues of Instruction Pipeline

- Multiple Functional Units
  - Reservation Station and Tags
  - Slow-station as Bottleneck stage
    - Subdivision of Pipeline Bottleneck stage
    - Replication of Pipeline Bottleneck stage
    - *(Example to be discussed)*

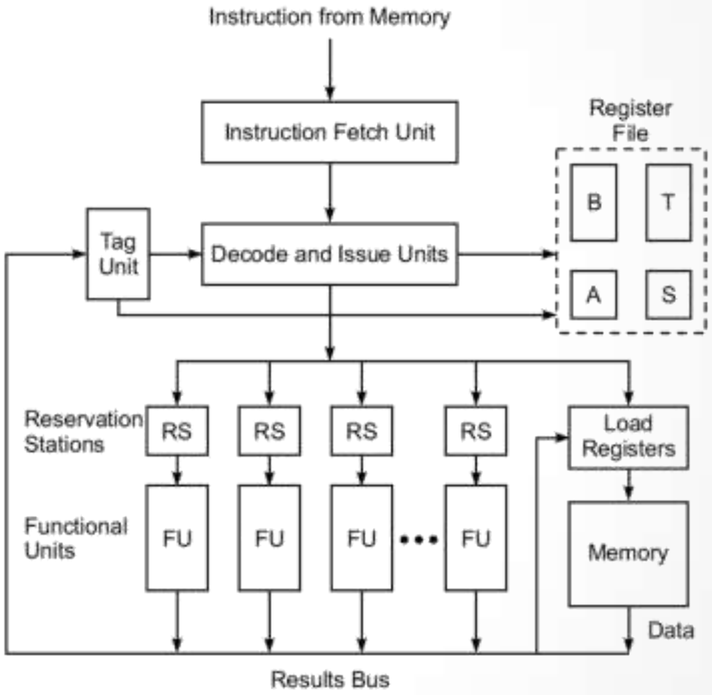


Fig. 6.12 A pipelined processor with multiple functional units and distributed reservation stations supported by tagging (Courtesy of G. Sohi; reprinted with permission from *IEEE Transactions on Computers*, March 1990)

INSTRUCTION PIPELINE DESIGN

Mechanisms/Design Issues of Instruction Pipeline

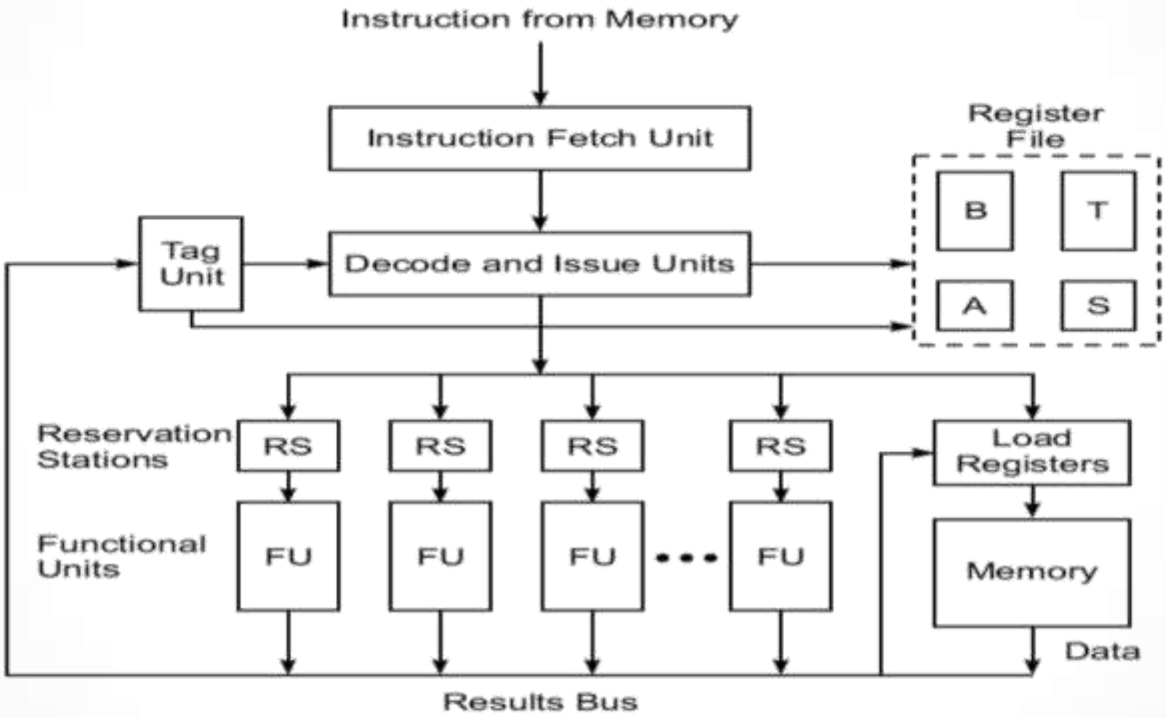


Fig. 6.12 A pipelined processor with multiple functional units and distributed reservation stations supported by tagging (Courtesy of G. Sohi; reprinted with permission from *IEEE Transactions on Computers*, March 1990)

INSTRUCTION PIPELINE DESIGN
Mechanisms/Design Issues of Instruction Pipeline

- Internal Forwarding and Register Tagging
  - Internal Forwarding:
    - *A "short-circuit" technique to replace unnecessary memory accesses by register-register transfers in a sequence of fetch-arithmetic-store operations*
  - Register Tagging:
    - *Use of tagged registers , buffers and reservation stations, for exploiting concurrent activities among multiple arithmetic units*
  - **Store-Fetch Forwarding**
    - *(M ← R1, R2 ← M) replaced by (M ← R1, R2 ← R1)*
  - **Fetch-Fetch Forwarding**
    - *(R1 ← M, R2 ← M) replaced by (R1 ← M, R2 ← R1)*
  - **Store-Store Overwriting**
    - *(M ← R1, M ← R2) replaced by (M ← R2)*

INSTRUCTION PIPELINE DESIGN
Mechanisms/Design Issues of Instruction Pipeline

- Internal Forwarding and Register Tagging

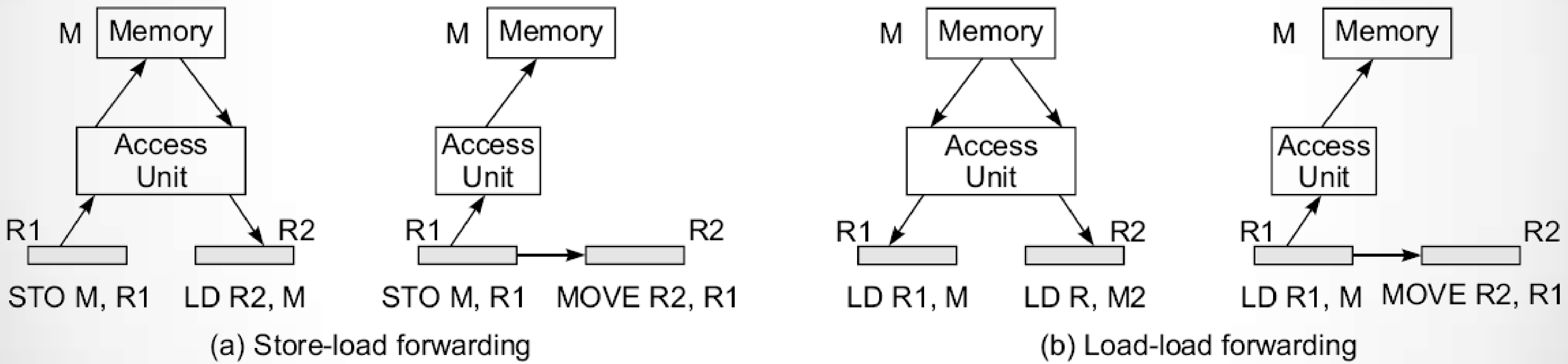
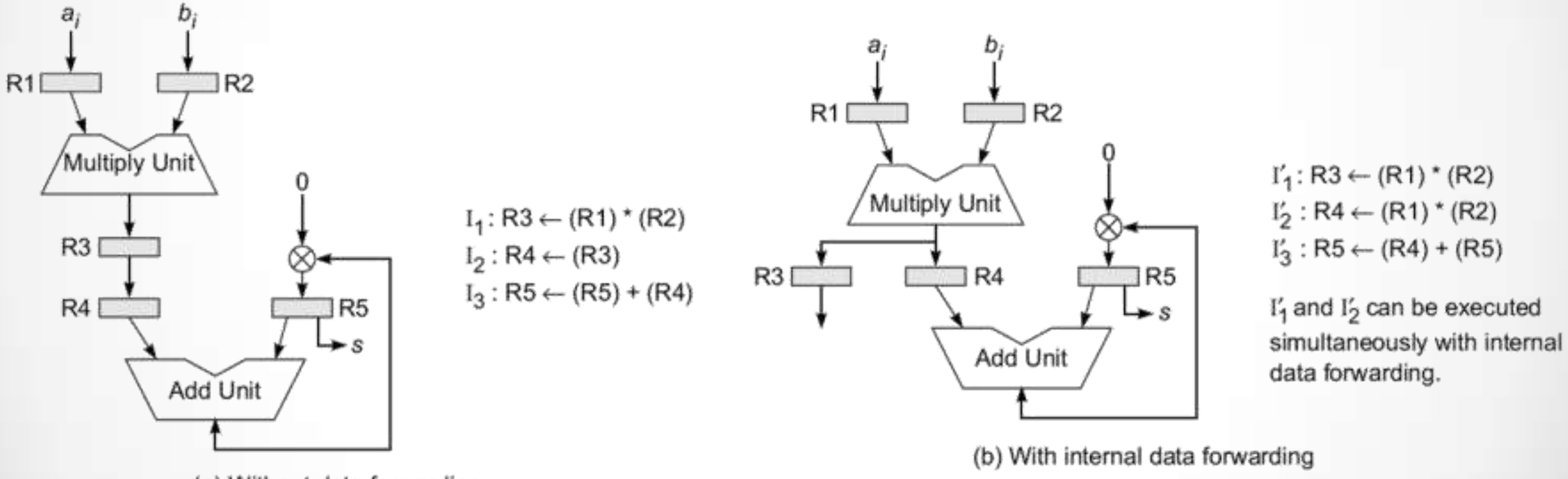


Fig. 6.13 Internal data forwarding by replacing memory-access operations with register transfer operations

INSTRUCTION PIPELINE DESIGN

Mechanisms/Design Issues of Instruction Pipeline

- Internal Forwarding and Register Tagging



$I_1 : R3 \leftarrow (R1) * (R2)$

$I_2 : R4 \leftarrow (R3)$

$I_3 : R5 \leftarrow (R5) + (R4)$

(a) Without data forwarding

$I'_1 : R3 \leftarrow (R1) * (R2)$

$I'_2 : R4 \leftarrow (R1) * (R2)$

$I'_3 : R5 \leftarrow (R4) + (R5)$

I'_1 and I'_2 can be executed simultaneously with internal data forwarding.

(b) With internal data forwarding

Fig. 6.14 Internal data forwarding for implementing the dot-product operation

INSTRUCTION PIPELINE DESIGN
Mechanisms/Design Issues of Instruction Pipeline

- Hazard Detection and Avoidance
  - Domain or Input Set of an instruction
  - Range or Output Set of an instruction
  - Data Hazards: **RAW**, **WAR** and **WAW**
  - Resolution using Register Renaming approach

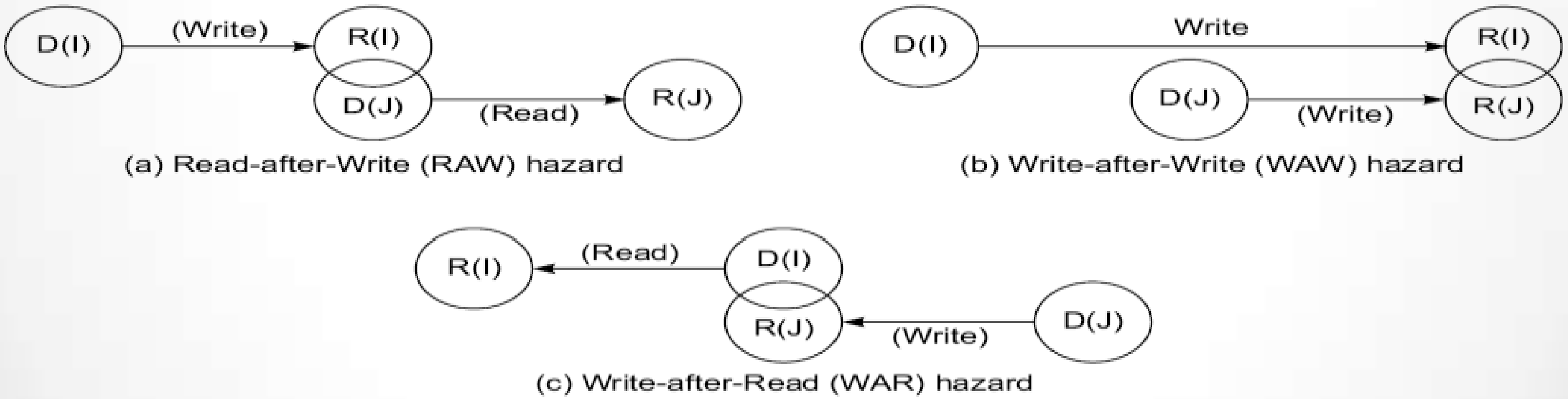
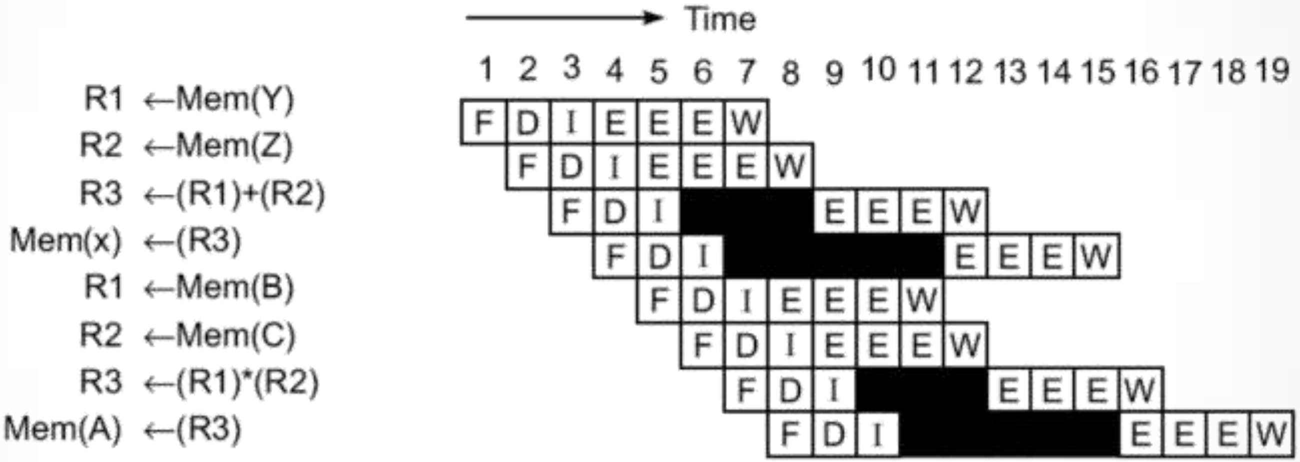

D(I)
(Write)
R(I)
D(J)
(Read)
R(J)
(a) Read-after-Write (RAW) hazard
D(I)
Write
R(I)
D(J)
(Write)
R(J)
(b) Write-after-Write (WAW) hazard
R(I)
(Read)
D(I)
R(J)
(Write)
D(J)
(c) Write-after-Read (WAR) hazard

Fig. 6.15 Possible hazards between read and write operations in an instruction pipeline (instruction I is ahead of instruction J in program order)

INSTRUCTION PIPELINE DESIGN
Dynamic Instruction Scheduling

- Idea of Static Scheduling
  - Compiler based scheduling strategy to resolve Interlocking among instructions
- Dynamic Scheduling
  - **Tomasulo's Algorithm** (Register-Tagging Scheme)
    - Hardware based dependence-resolution
  - **Scoreboarding** Technique
    - Scoreboard: the centralized control unit
    - A kind of data-driven mechanism

INSTRUCTION PIPELINE DESIGN
Dynamic Instruction Scheduling



R1 ←Mem(Y)
R2 ←Mem(Z)
R3 ←(R1)+(R2)
Mem(x) ←(R3)
R1 ←Mem(B)
R2 ←Mem(C)
R3 ←(R1)*(R2)
Mem(A) ←(R3)

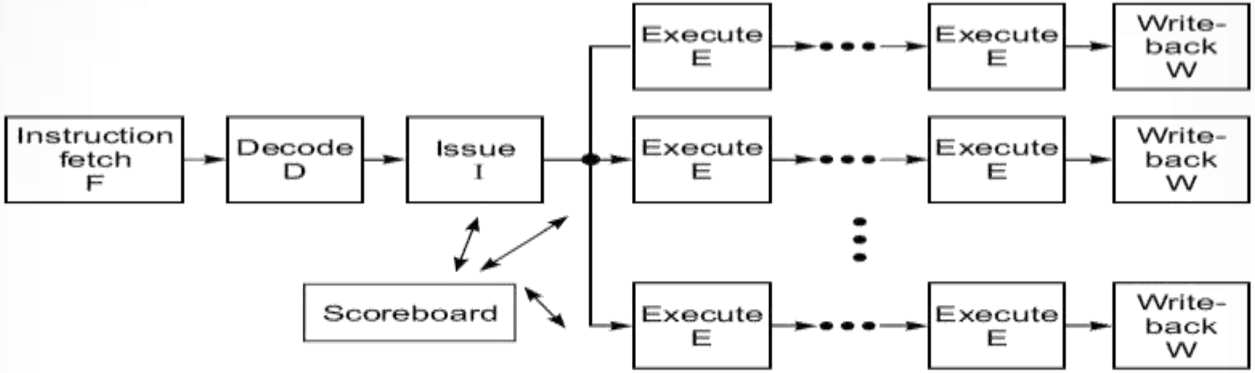
(a) Minimum-register machine code

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
F	D	I	E	E	E	W												
	F	D	I	E	E	E	W											
		F	D	I	■	■	■	E	E	E	W							
			F	D	I	■	■	■	■	■	E	E	E	W				
				F	D	I	E	E	E	W								
					F	D	I	E	E	E	W							
						F	D	I	■	■	■	E	E	E	W			
							F	D	I	■	■	■	■	■	E	E	E	W

(b) The pipeline schedule

Fig. 6.16 Dynamic instruction scheduling using Tomasulo's algorithm on the processor in Fig. 6.12 (Courtesy of James Smith; reprinted with permission from *IEEE Computer*, July 1989)

INSTRUCTION PIPELINE DESIGN
Dynamic Instruction Scheduling



(a) A CDC 6600-like processor

Time →

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
R1 ←Mem(Y)	F	D	I	E	E	E	W												
R2 ←Mem(Z)		F	D	I	E	E	E	W											
R3 ←(R1)+(R2)			F	D	I	■	■	■	E	E	E	W							
Mem(x) ←(R3)				F	D	I	■	■	■	■	■	E	E	E	W				
R4 ←Mem(B)					F	D	I	E	E	E	W								
R5 ←Mem(C)						F	D	I	E	E	E	W							
R6 ←(R4)*(R5)							F	D	I	■	■	■	E	E	E	W			
Mem(A) ←(R6)								F	D	I	■	■	■	■	■	E	E	E	W

(b) The improved schedule from Fig. 6.9b

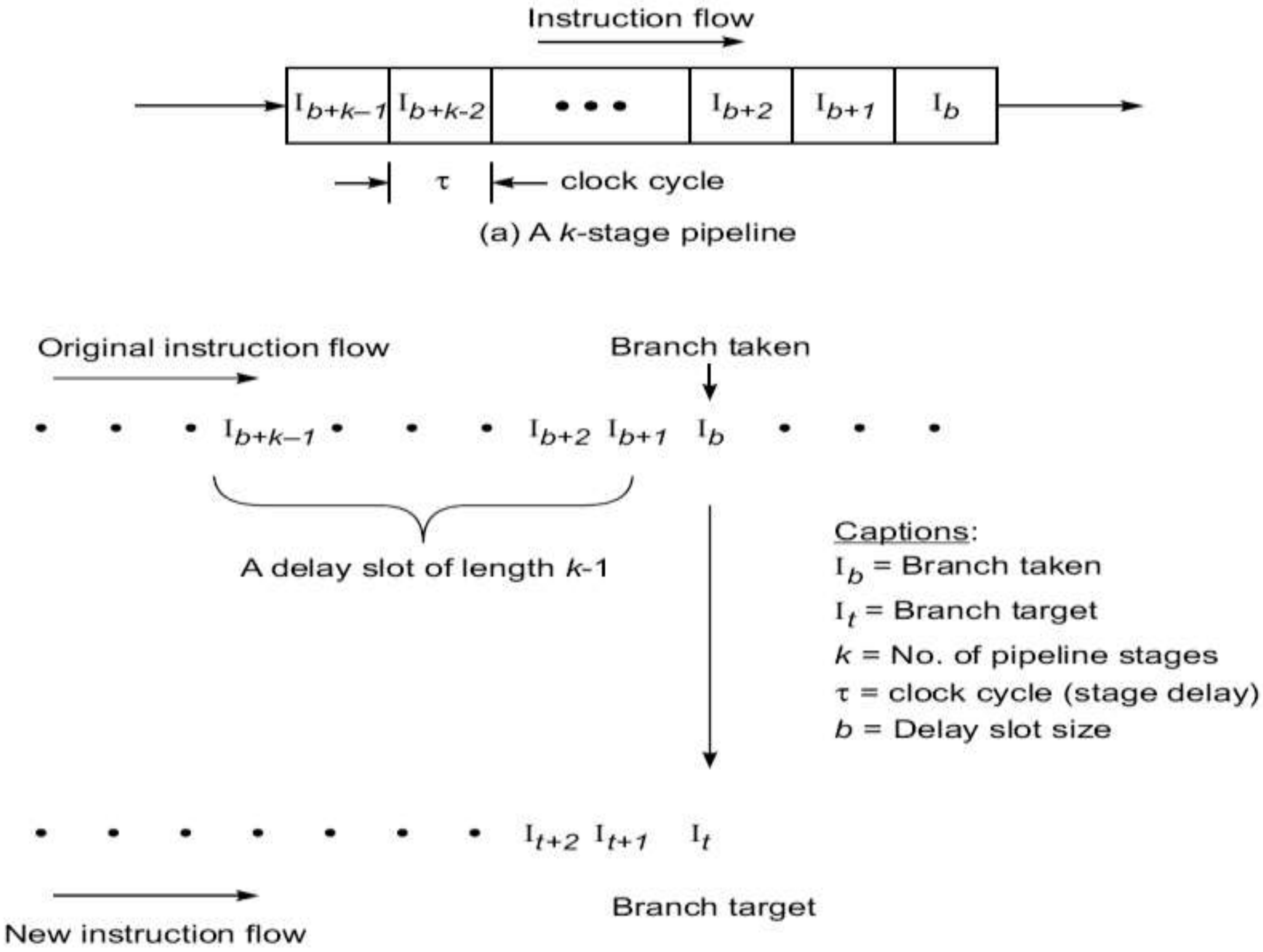
Fig. 6.17 Hardware scoreboarding for dynamic instruction scheduling (Courtesy of James Smith; reprinted with permission from *IEEE Computer*, July 1989)

INSTRUCTION PIPELINE DESIGN
Branch Handling Techniques

- Branch Taken, Branch Target, Delay Slot
- Effect of Branching
  - Parameters:
    - **k** : No. of stages in the pipeline
    - **n** : Total no. of instructions or tasks
    - **p** : Percentage of Brach instructions over **n**
    - **q** : Percentage of successful branch instructions (branch taken) over **p**.
    - **b** : Delay Slot
    - τ : Pipeline Cycle Time
  - Branch Penalty = **q of (p of n) * bτ = pqnbτ**
  - Effective Execution Time:
    - $T_{eff} = [k + (n-1)]\,\tau + pqnb\tau = [k + (n-1) + pqnb]\tau$

INSTRUCTION PIPELINE DESIGN
Branch Handling Techniques

- Effect of Branching
  - Effective Throughput:
    - $H_{eff} = n/T_{eff}$
    - $H_{eff} = n / \{[k + (n-1) + pqnb]\tau\} = nf / [k + (n-1) + pqnb]$
    - As **n**→***Infinity*** and **b = k-1**
      - $H^{*}_{eff} = f / [pq(k-1)+1]$
    - If **p=0** and **q=0** (no branching occurs)
      - $H^{**}_{eff} = f = 1/\tau$
  - Performance Degradation Factor
    - $D = 1 – H^{*}_{eff} / f = pq(k-1) / [pq(k-1)+1]$


Instruction flow
I_{b+k-1}
I_{b+k-2}
• • •
I_{b+2}
I_{b+1}
I_b
τ
clock cycle
Original instruction flow
Branch taken
• • • I_{b+k-1} • • • I_{b+2} I_{b+1} I_b • • •
A delay slot of length k-1
Captions:
I_b = Branch taken
I_t = Branch target
k = No. of pipeline stages
τ = clock cycle (stage delay)
b = Delay slot size
• • • • • • • I_{t+2} I_{t+1} I_t
Branch target
New instruction flow

(a) A k-stage pipeline

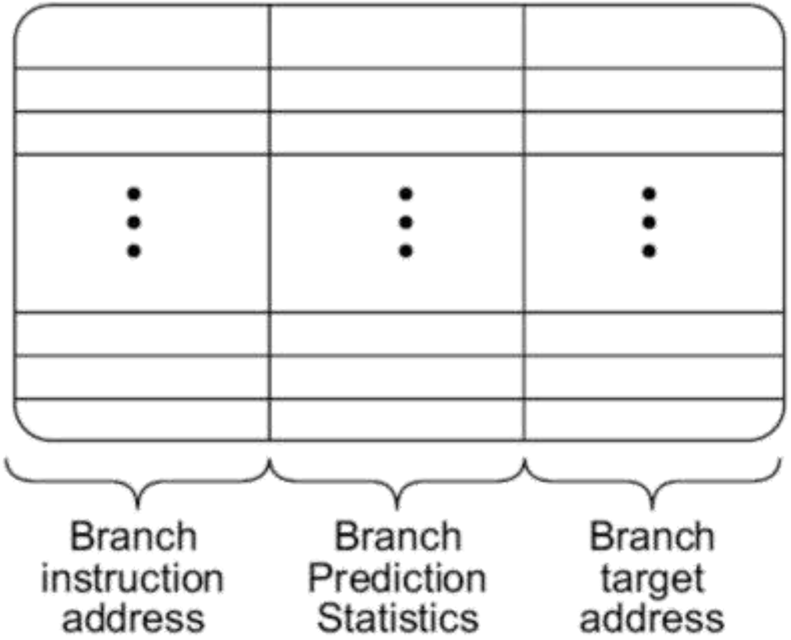
(b) An instruction stream containing a branch taken

Fig. 6.18 The decision of a branch taken at the last stage of an instruction pipeline causes $b \leq k - 1$ previously loaded instructions to be drained from the pipeline

INSTRUCTION PIPELINE DESIGN
Branch Handling Techniques

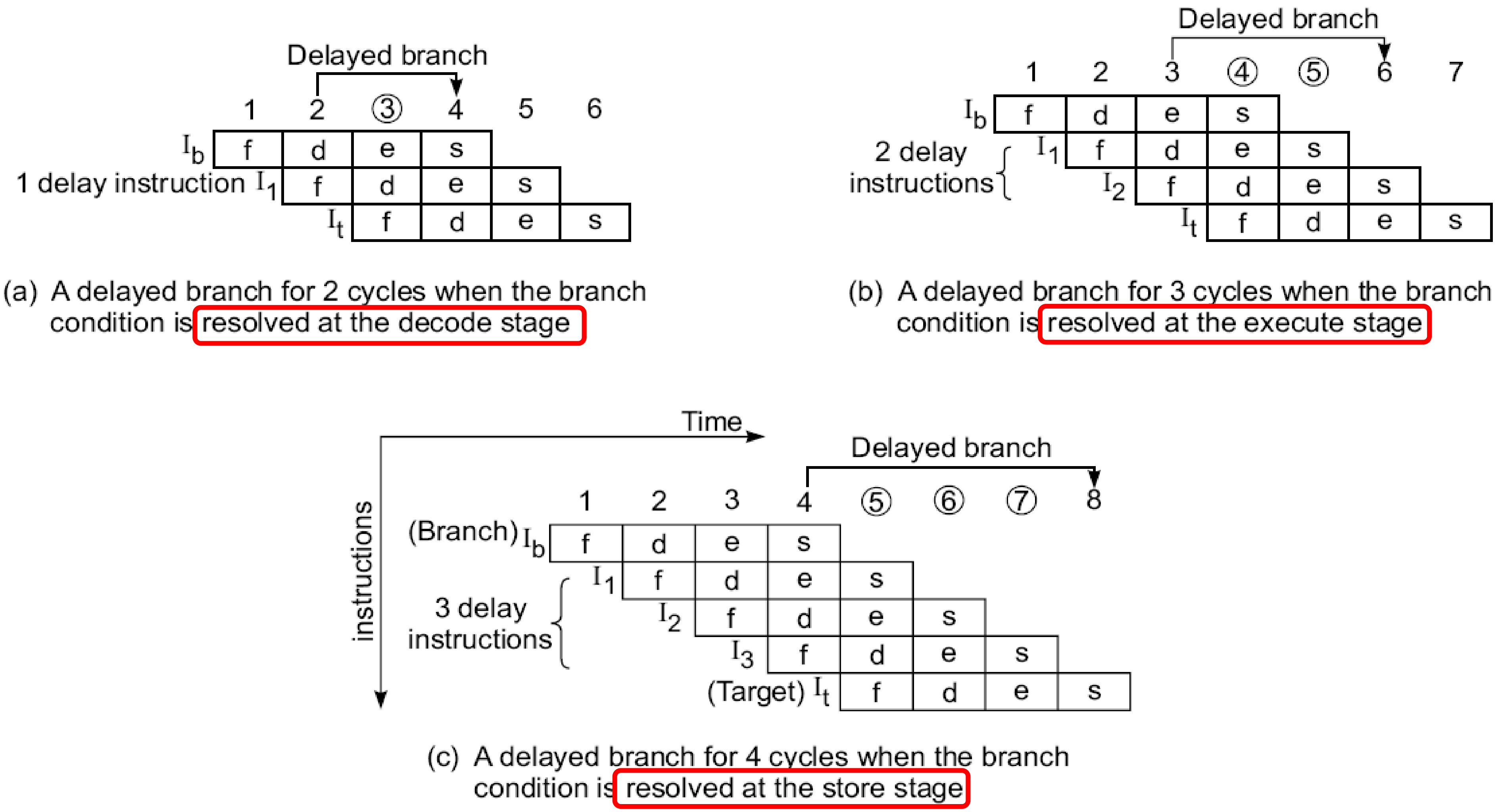
- Branch Prediction
  - **Static Branch Prediction:** based on branch code types
  - **Dynamic Branch prediction:** based on recent branch history
    - **Strategy 1:** Predict the branch direction based on information found at decode stage.
    - **Strategy 2:** Use a cache to store target addresses at effective address calculation stage.
    - **Strategy 3:** Use a cache to store target instructions at fetch stage
  - **Brach Target Buffer Organization**
- Delayed Branches
  - A *delayed branch* of ***d*** cycles allows at most ***d-1*** useful instructions to be executed following the *branch taken*.
  - Execution of these instructions should be independent of branch instruction to achieve a **zero branch penalty**

INSTRUCTION PIPELINE DESIGN
Branch Handling Techniques



(a) Branch target buffer organization

Fig. 6.19 Branch history buffer and a state transition diagram used in dynamic branch prediction (Courtesy of Lee and Smith, *IEEE Computer*, 1984)



(a) A delayed branch for 2 cycles when the branch condition is resolved at the decode stage

(b) A delayed branch for 3 cycles when the branch condition is resolved at the execute stage

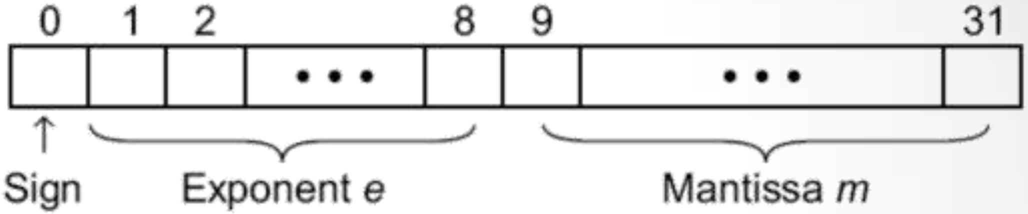
(c) A delayed branch for 4 cycles when the branch condition is resolved at the store stage

Fig. 6.20 The concept of delayed branch by moving independent instructions or NOP fillers into the delay slot of a four-stage pipeline

ARITHMETIC PIPELINE DESIGN
Computer Arithmetic Operations

- Finite-precision arithmetic
- Overflow and Underflow
- Fixed-Point operations
  - Notations:
    - *Signed-magnitude, one’s complement* and *two-complement* notation
  - Operations:
    - *Addition*: (n bit, n bit) ➔ (n bit) Sum, 1 bit output carry
    - *Subtraction*: (n bit, n bit) ➔ (n bit) difference
    - *Multiplication*: (n bit, n bit) ➔ (2n bit) product
    - *Division*: (2n bit, n bit) ➔ (n bit) quotient, (n bit) remainder

ARITHMETIC PIPELINE DESIGN
Computer Arithmetic Operations



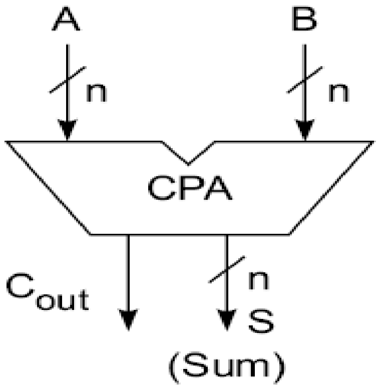
- Floating-Point Numbers
  - *X* = *(m, e)* representation
    - ***m***: mantissa or fraction
    - **e**: exponent with an implied base or radix ***r***.
    - Actual Value $X = m * r^{e}$
  - Operations on numbers **X = (m_x, e_x)** and **Y = (m_y, e_y)**
    - *Addition*: $(m_x * r^{ex-ey} + m_y, e_y)$
    - *Subtraction*: $(m_x * r^{ex-ey} - m_y, e_y)$
    - *Multiplication*: $(m_x * m_y, e_x+e_y)$
    - *Division*: $(m_x / m_y, e_x - e_y)$
- Elementary Functions
  - Transcendental functions like: Trigonometric, Exponential, Logarithmic, etc.

ARITHMETIC PIPELINE DESIGN
Static Arithmetic Pipelines

- Separate units for fixed point operations and floating point operations
- Scalar and Vector Arithmetic Pipelines
- Uni-functional or Static Pipelines
- Arithmetic Pipeline Stages
  - Majorly involve hardware to perform: **Add** and **Shift** micro-operations
  - **Addition** using: *Carry Propagation Adder (CPA)* and *Carry Save Adder (CSA)*
  - **Shift** using: *Shift Registers*
- Multiplication Pipeline Design
  - E.g. To multiply two 8-bit numbers that yield a 16-bit product using CSA and CPA *Wallace Tree.*

e.g. n=4

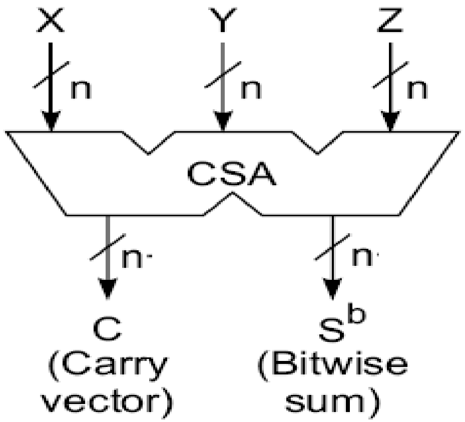
A = 1 0 1 1
+) B = 0 1 1 1
S = 1 0 0 1 0 = A + B



(a) An *n*-bit carry-propagate adder (CPA) which allows either carry propagation or applies the carry-lookahead technique

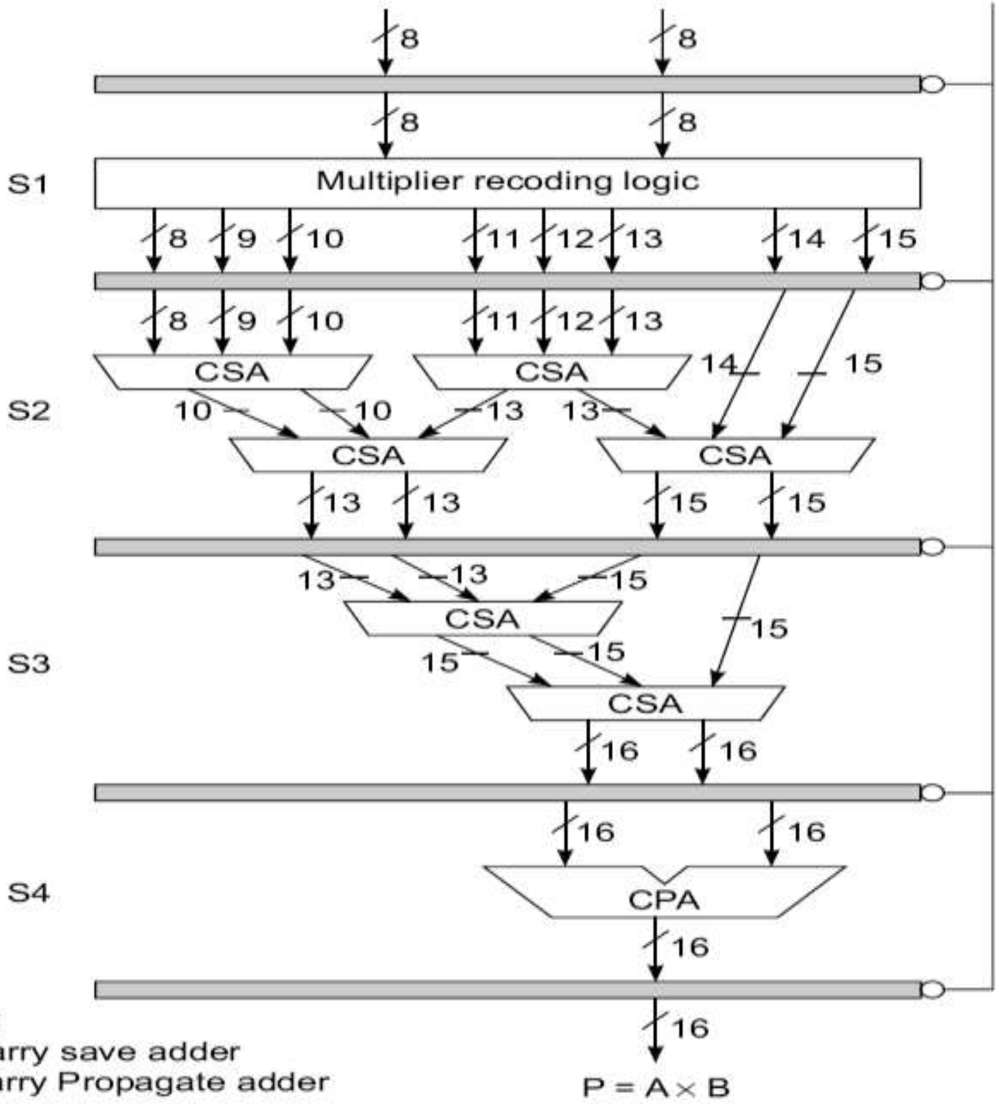
e.g. n=4

X = 0 0 1 0 1 1
Y = 0 1 0 1 0 1
⊕ Z = 1 1 1 1 0 1
S^b= 0 1 0 0 0 1 1
+) C = 0 1 1 1 0 1 0
S = 1 0 1 1 1 0 1 = S^b+C = X+Y+Z



(b) An *n*-bit carry-save adder (CSA), where S^b is the bitwise sum of X, Y, and Z, and C is a carry vector generated without carry propagation between digits

Fig. 6.22 Distinction between a carry-propagate adder (CPA) and a carry-save adder (CSA)



Captions:
CSA = Carry save adder
CPA = Carry Propagate adder

Fig. 6.23 A pipeline unit for fixed-point multiplication of 8-bit integers (The number along each line indicates the line width.)

ARITHMETIC PIPELINE DESIGN
Multifunctional Arithmetic Pipelines

- Multifunctional Pipeline:
  - **Static** multifunctional pipeline
  - **Dynamic** multifunctional pipeline
- Case Study: T1/ASC static multifunctional pipeline architecture

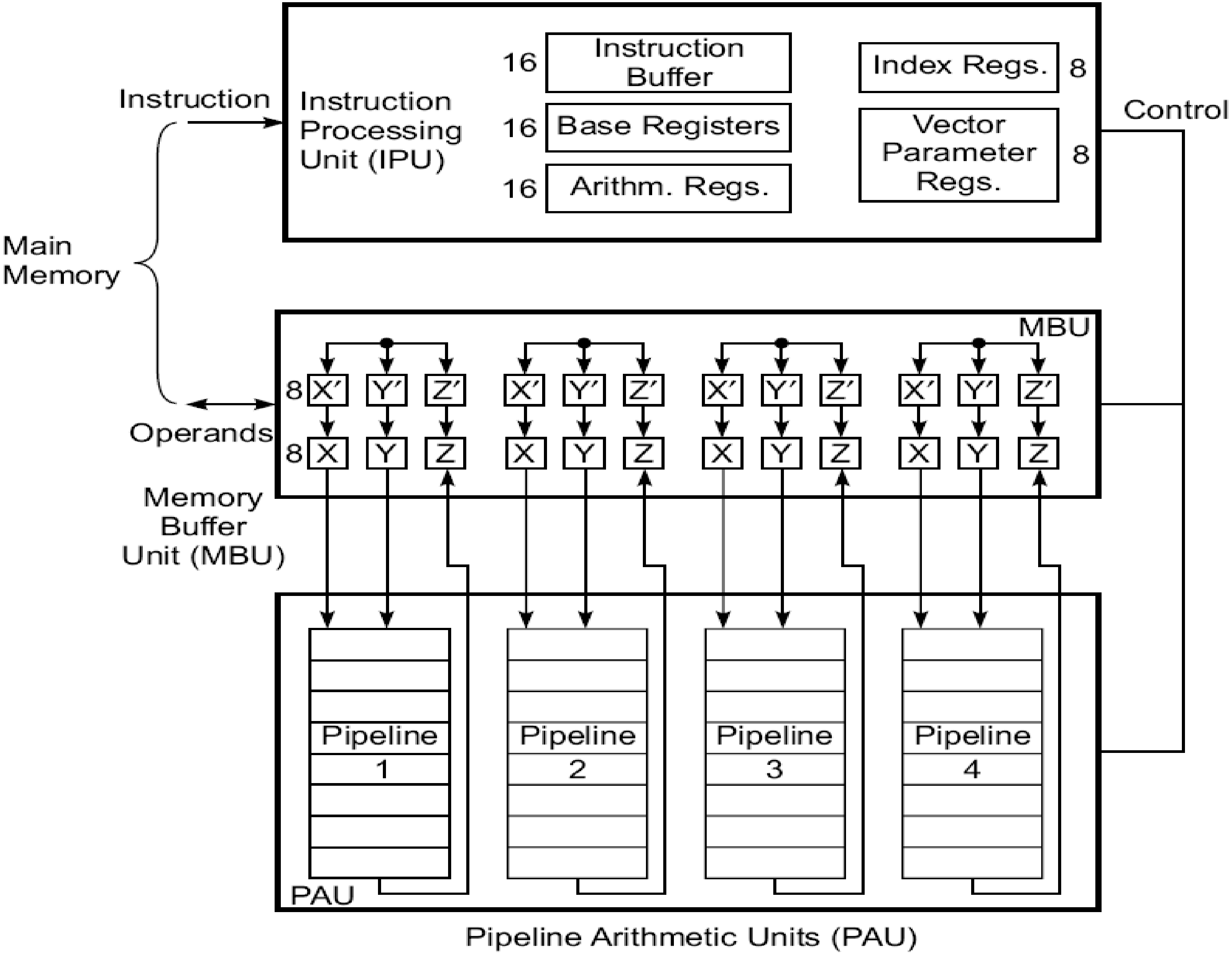
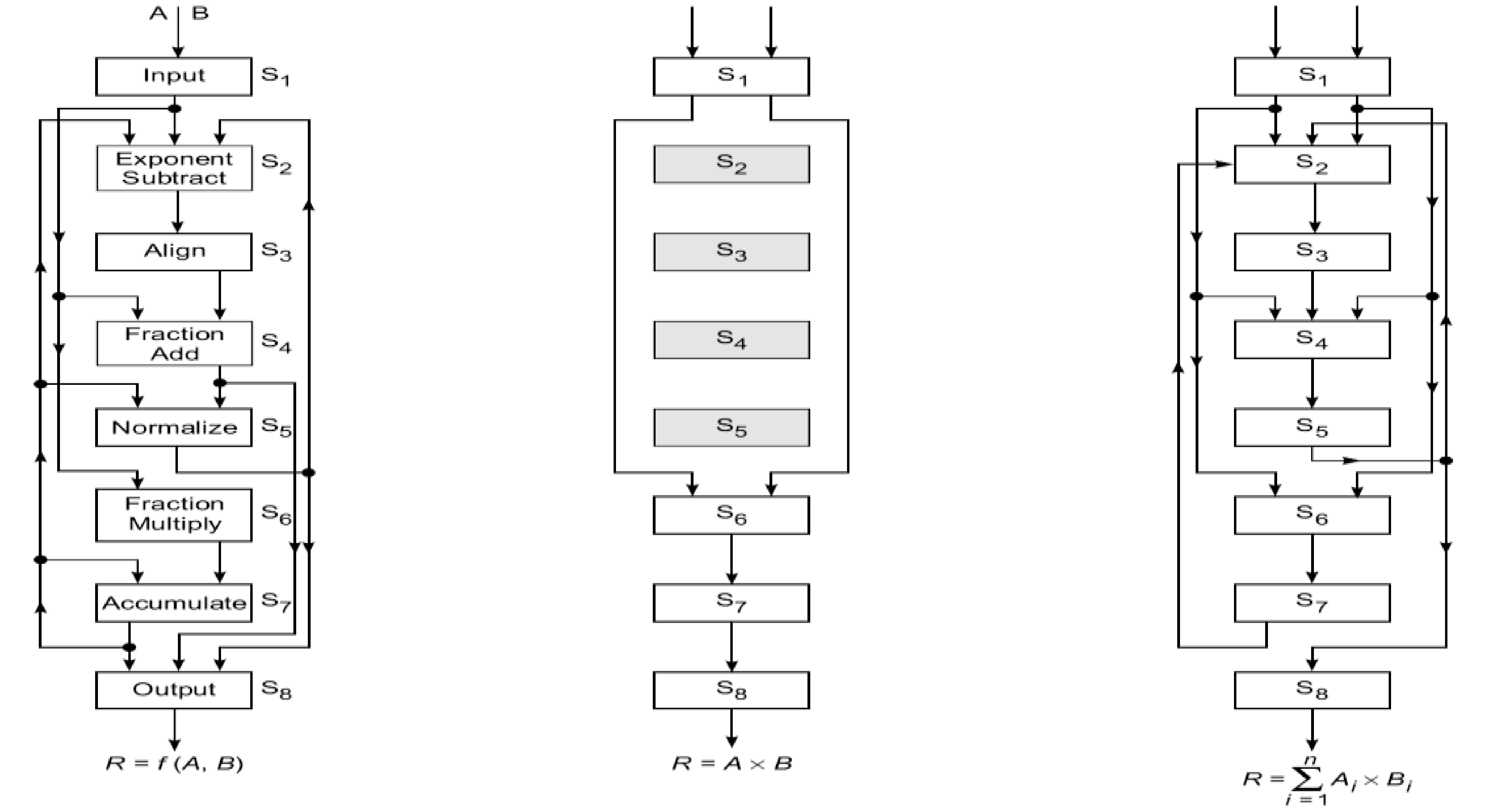


Fig. 6.26 The architecture of the TI Advanced Scientific Computer (ASC) (Courtesy of Texas Instruments, Inc.)



(a) Pipeline stages and interconnections

(b) Fixed-point multiplication

(c) Floating-point dot product

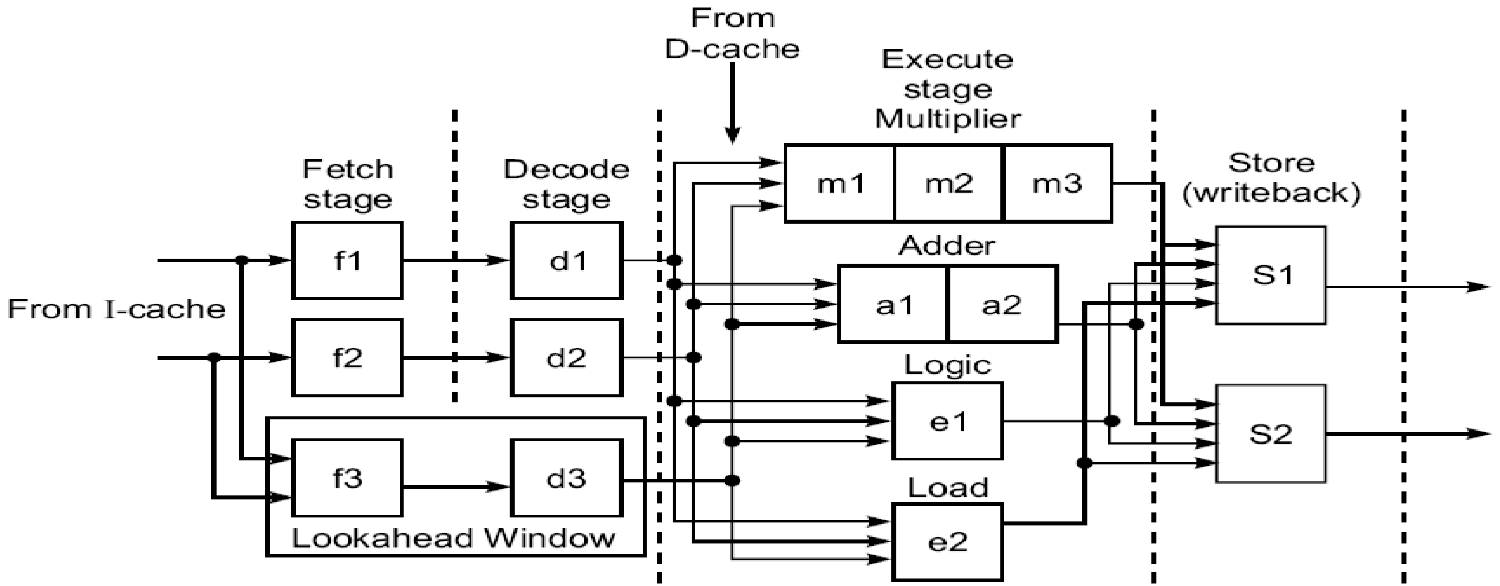
Fig. 6.27 The multiplication arithmetic pipeline of the TI Advanced Scientific Computer and the interstage connections of two representative functions (Shaded stages are unutilized)

SUPERSCALAR PIPELINE DESIGN

- Pipeline Design Parameters
  - Pipeline cycle, Base cycle, Instruction issue rate, Instruction issue Latency, Simple Operation Latency
  - ILP to fully utilize the pipeline
- Superscalar Pipeline Structure
- Data and Resource Dependencies
- Pipeline Stalling
- Superscalar Pipeline Scheduling
  - In-order Issue and in-order completion
  - In-order Issue and out-of-order completion
  - Out-of-order Issue and out-of-order completion
- Superscalar Performance

SUPERSCALAR PIPELINE DESIGN

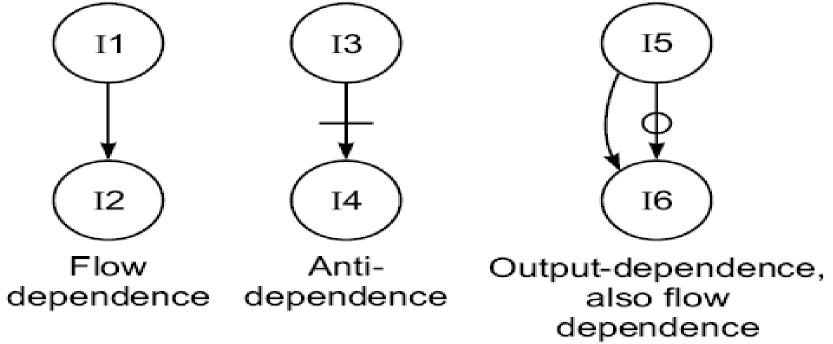
Parameter	Base Scalar Processor	Super Scalar Processor (degree = K)
Pipeline Cycle	1 (base cycle)	K
Instruction Issue Rate	1	K
Instruction Issue Latency	1	1
Simple Operation Latency	1	1
ILP to fully utilize pipeline	1	K



(a) A dual-pipleline, superscalar processor with four functional units in the execution stage and a lookahead window producing out-of-order issues

Program order

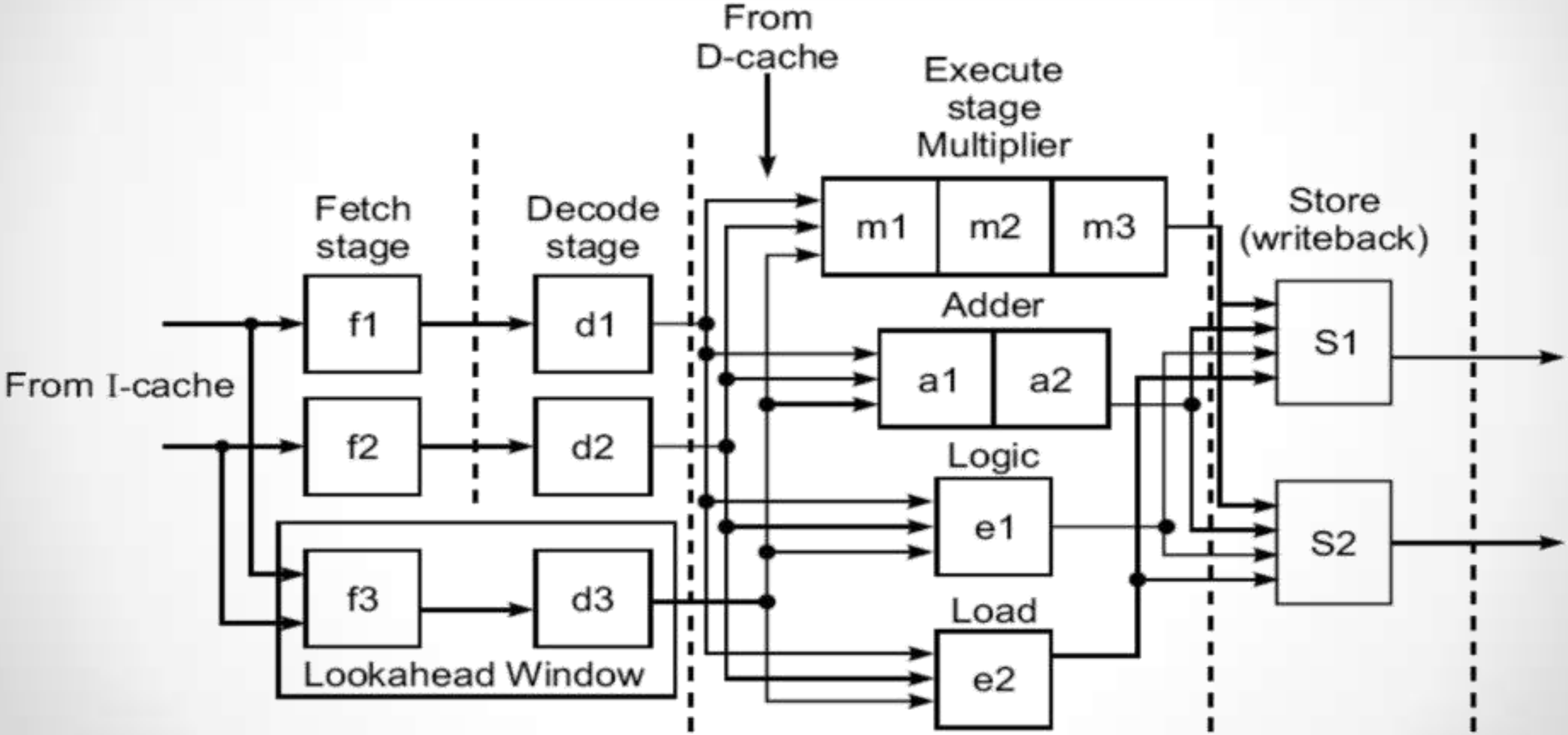
```
I1.  Load   R1,  A      / R1 ← Memory (A) /
I2.  Add    R2,  R1     / R2 ← (R2) + (R1) /
I3.  Add    R3,  R4     / R3 ← (R3) + (R4) /
I4.  Mul    R4,  R5     / R4 ← (R4) * (R5) /
I5.  Comp   R6          / R6 ← (R̅6̅) /
I6.  Mul    R6,  R7     / R6 ← (R6) * (R7) /
```

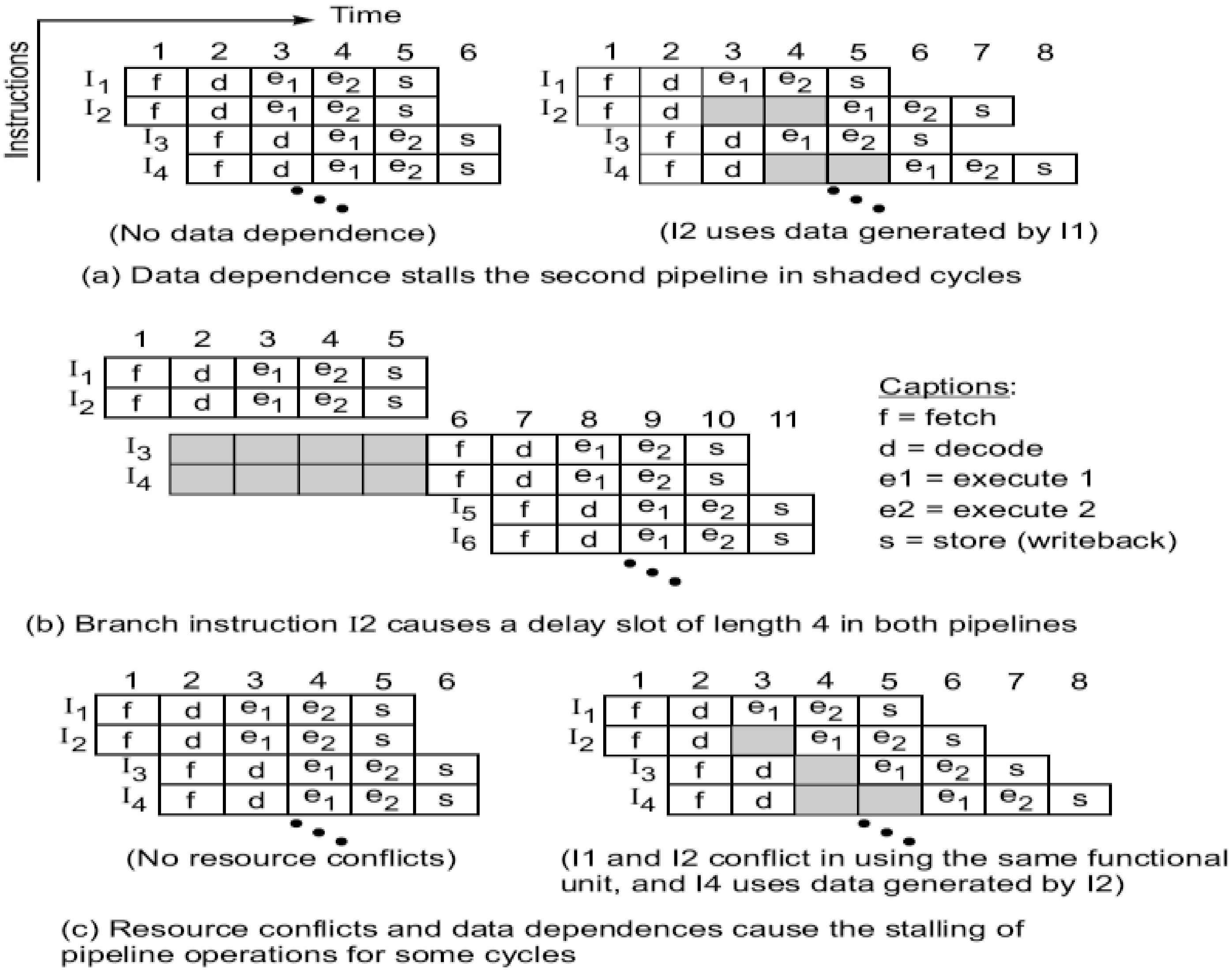


(b) A sample program and its dependence graph, where I2 and I3 share the adder and I4 and I6 share the multiplier

Fig. 6.28 A two-issue superscalar processor and a sample program for parallel execution

SUPERSCALAR PIPELINE DESIGN





	1	2	3	4	5	6
I_1	f	d	e_1	e_2	s	
I_2	f	d	e_1	e_2	s	
I_3		f	d	e_1	e_2	s
I_4		f	d	e_1	e_2	s

(No data dependence)

	1	2	3	4	5	6	7	8
I_1	f	d	e_1	e_2	s			
I_2	f	d	▨	▨	e_1	e_2	s	
I_3		f	d	e_1	e_2	s		
I_4		f	d	▨	▨	e_1	e_2	s

(I2 uses data generated by I1)

(a) Data dependence stalls the second pipeline in shaded cycles

	1	2	3	4	5	6	7	8	9	10	11
I_1	f	d	e_1	e_2	s						
I_2	f	d	e_1	e_2	s						
I_3		▨	▨	▨	▨	f	d	e_1	e_2	s	
I_4		▨	▨	▨	▨	f	d	e_1	e_2	s	
I_5							f	d	e_1	e_2	s
I_6							f	d	e_1	e_2	s

(b) Branch instruction I2 causes a delay slot of length 4 in both pipelines

	1	2	3	4	5	6
I_1	f	d	e_1	e_2	s	
I_2	f	d	e_1	e_2	s	
I_3		f	d	e_1	e_2	s
I_4		f	d	e_1	e_2	s

(No resource conflicts)

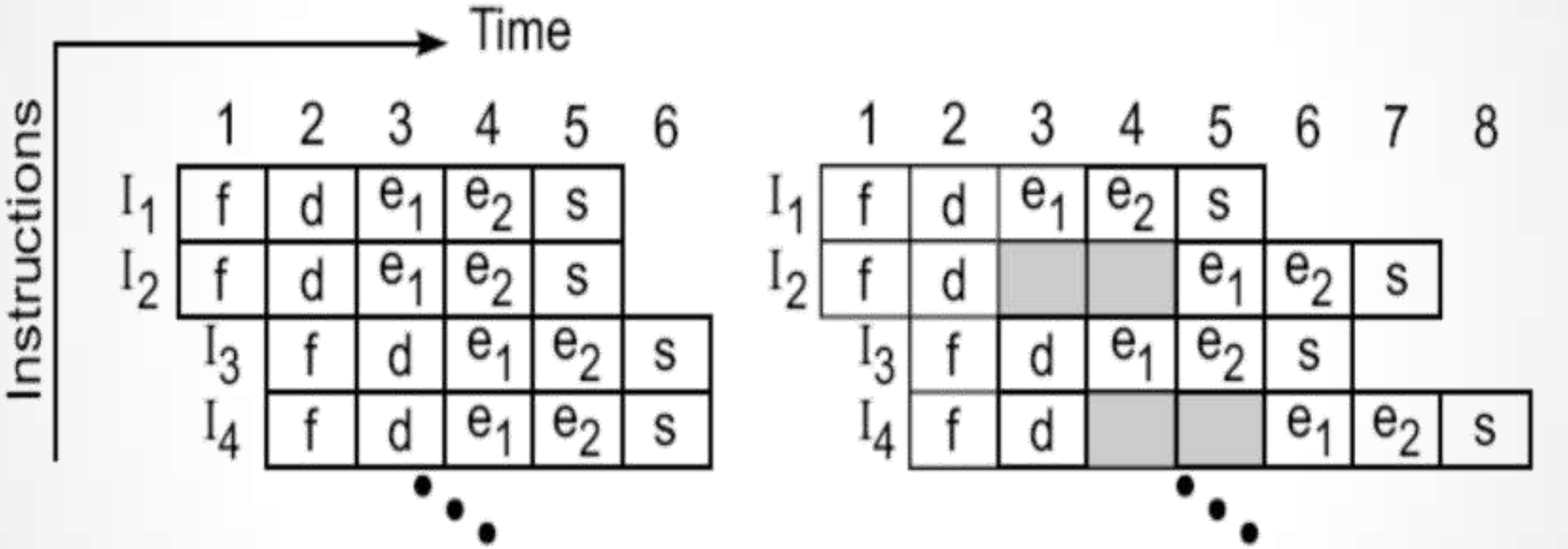
	1	2	3	4	5	6	7	8
I_1	f	d	e_1	e_2	s			
I_2	f	d	▨	e_1	e_2	s		
I_3		f	d	▨	e_1	e_2	s	
I_4		f	d	▨	▨	e_1	e_2	s

(I1 and I2 conflict in using the same functional unit, and I4 uses data generated by I2)

(c) Resource conflicts and data dependences cause the stalling of pipeline operations for some cycles

Fig. 6.29 Dependences and resource conflicts may stall one or two pipelines in a two-issue superscalar processor

SUPERSCALAR PIPELINE DESIGN



	1	2	3	4	5	6
I_1	f	d	e_1	e_2	s	
I_2	f	d	e_1	e_2	s	
I_3		f	d	e_1	e_2	s
I_4		f	d	e_1	e_2	s

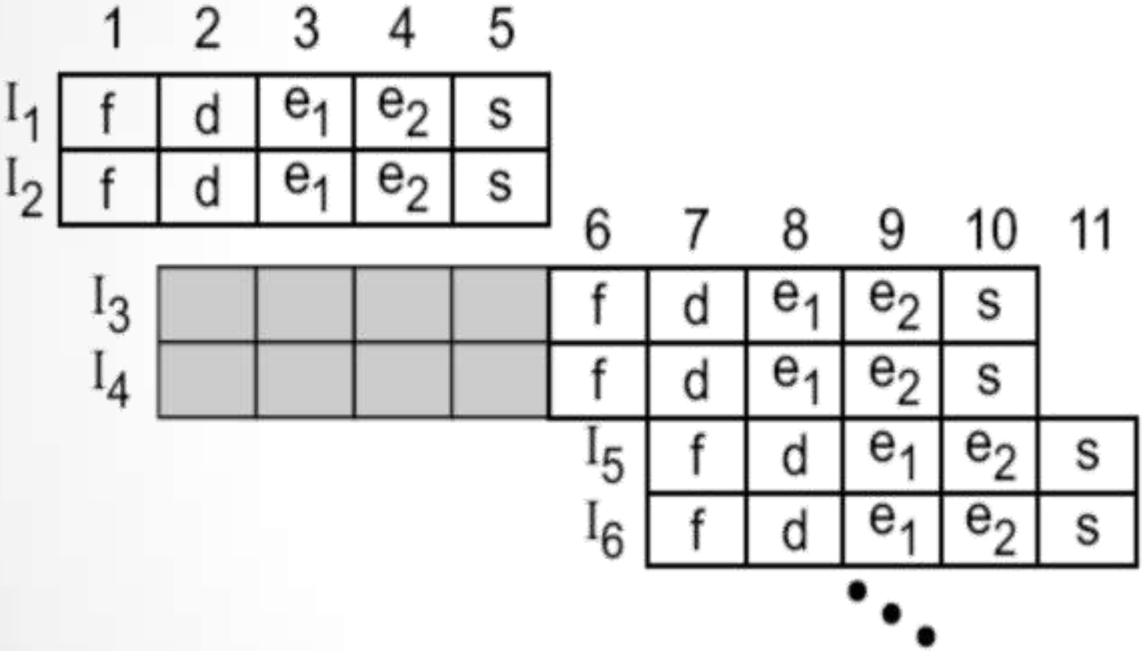
(No data dependence)

	1	2	3	4	5	6	7	8
I_1	f	d	e_1	e_2	s			
I_2	f	d	▨	▨	e_1	e_2	s	
I_3		f	d	e_1	e_2	s		
I_4		f	d	▨	▨	e_1	e_2	s

(I2 uses data generated by I1)

(a) Data dependence stalls the second pipeline in shaded cycles

SUPERSCALAR PIPELINE DESIGN



	1	2	3	4	5	6	7	8	9	10	11
I_1	f	d	e_1	e_2	s						
I_2	f	d	e_1	e_2	s						
I_3		▨	▨	▨	▨	f	d	e_1	e_2	s	
I_4		▨	▨	▨	▨	f	d	e_1	e_2	s	
I_5							f	d	e_1	e_2	s
I_6							f	d	e_1	e_2	s

...

Captions:

f = fetch

d = decode

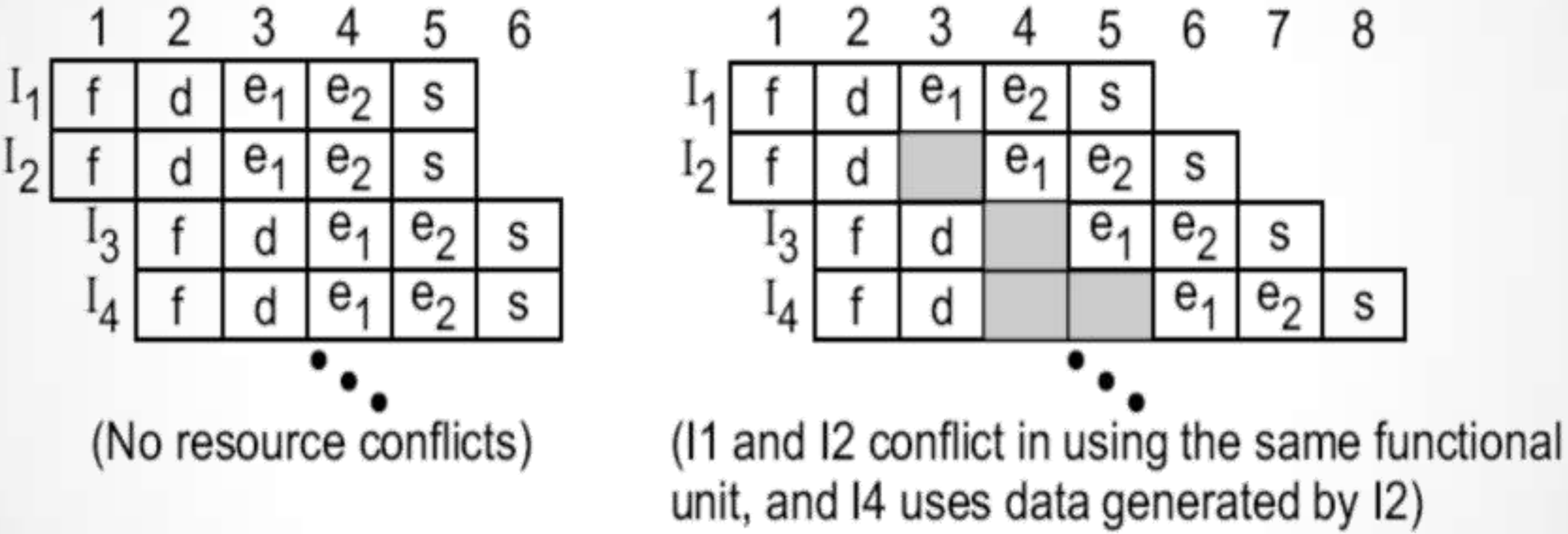
e1 = execute 1

e2 = execute 2

s = store (writeback)

(b) Branch instruction I2 causes a delay slot of length 4 in both pipelines

SUPERSCALAR PIPELINE DESIGN



	1	2	3	4	5	6
I_1	f	d	e_1	e_2	s	
I_2	f	d	e_1	e_2	s	
I_3		f	d	e_1	e_2	s
I_4		f	d	e_1	e_2	s

(No resource conflicts)

	1	2	3	4	5	6	7	8
I_1	f	d	e_1	e_2	s			
I_2	f	d	▨	e_1	e_2	s		
I_3		f	d	▨	e_1	e_2	s	
I_4		f	d	▨	▨	e_1	e_2	s

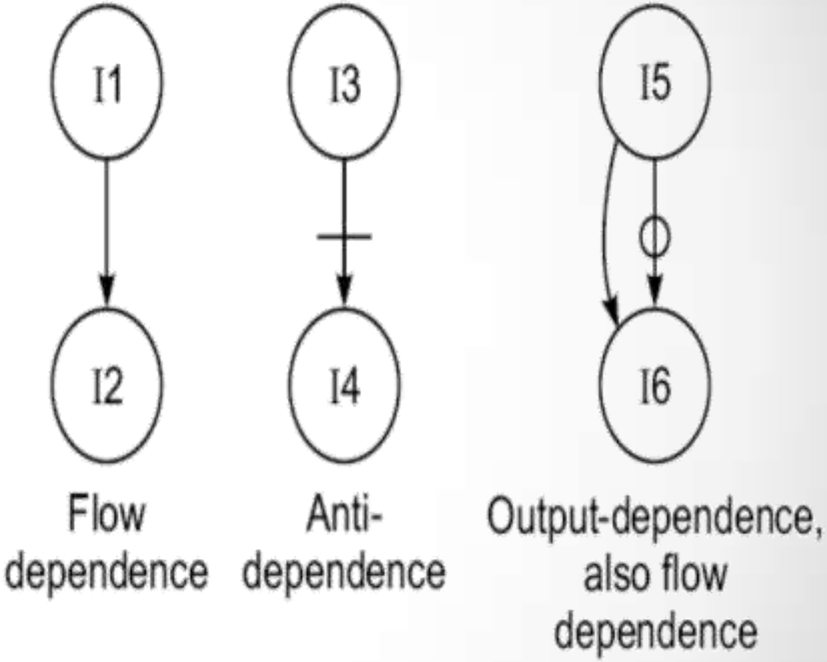
(I1 and I2 conflict in using the same functional unit, and I4 uses data generated by I2)

(c) Resource conflicts and data dependences cause the stalling of pipeline operations for some cycles

SUPERSCALAR PIPELINE DESIGN

Program order

I1.	Load	R1,	A	$/ R1 \leftarrow \text{Memory}(A) /$
I2.	Add	R2,	R1	$/ R2 \leftarrow (R2) + (R1) /$
I3.	Add	R3,	R4	$/ R3 \leftarrow (R3) + (R4) /$
I4.	Mul	R4,	R5	$/ R4 \leftarrow (R4) * (R5) /$
I5.	Comp	R6		$/ R6 \leftarrow (\overline{R6}) /$
I6.	Mul	R6,	R7	$/ R6 \leftarrow (R6) * (R7) /$



(b) A sample program and its dependence graph, where I2 and I3 share the adder and I4 and I6 share the multiplier

Time (clock cycles)

Instructions

	1	2	3	4	5	6	7	8	9
Pipe 1, I_1	f_1	d_1	e_2	s_1					
Pipe 2, I_2	f_2	d_2	▨	a_1	a_2	s_2			
I_3		f_1	d_1	▨	a_1	a_2	s_1		
I_4		f_2	d_2	m_1	m_2	m_3	s_2		
I_5			f_1	d_1	e_1	▨	▨	s_1	
I_6			f_2	d_2	▨	m_1	m_2	m_3	s_2

(a) In-order issue with in-order completion in nine cycles

	1	2	3	4	5	6	7	8	9
Pipe 1, I_1	f_1	d_1	e_2	s_1					
Pipe 2, I_2	f_2	d_2	▨	a_1	a_2	s_2			
I_3		f_1	d_1	▨	a_1	a_2	s_1		
I_4		f_2	d_2	m_1	m_2	m_3	s_2		
I_5			f_1	d_1	e_1	s_1			
I_6			f_2	d_2	▨	m_1	m_2	m_2	s_2

Completion order

	4	5	6	7	8	9
Pipe 1	I_1	▨	I_5	I_3	▨	▨
Pipe 2	▨	▨	I_2	I_4	▨	I_6

(b) In-order issue and out-of-order completion in nine cycles

	1	2	3	4	5	6	7
Pipe 1, I_3	f_1	d_1	a_1	a_2	s_1		
Pipe 2, I_4	f_2	d_2	m_1	m_2	m_3	s_2	
Lookahead Window I_5	f_3	d_3	e_1	s_1	(Pipe 1)		
Pipe 1, I_6		f_1	d_1	m_1	m_2	m_3	s_2
Pipe 2, I_1		f_2	d_2	e_2	s_2		
Pipe 2, I_2			f_1	d_1	a_1	a_2	s_1

Issue order

	1	2	3
Pipe 1	I_3	I_6	▨
Pipe 2	I_4	I_1	I_2
Lookahead	I_5		

Completion order

	4	5	6	7
Pipe 1	I_5	I_3	▨	I_2
Pipe 2	▨	I_1	I_4	I_6

(c) Out-of-order issue and out-of-order completion in seven cycles using an instruction lookahead window in the recoding process

Fig. 6.30 Instruction issue and completion policies for a superscalar processor with and without instruction lookahead support (Timing charts correspond to parallel execution of the program in Fig. 6.28)

SUPERSCALAR PIPELINE DESIGN

- Time required by base scalar machine:
  - $T(1,1) = K + N – 1$
- The ideal execution time required by m-issue superscalar machine:
  - $T(m,1) = K + (N – m)/m$
  - Where,
    - K is the time required to execute first m instructions through m pipelines of k-stages simultaneously
    - Second term corresponds to time required to execute remaining N-m instructions , m per cycle through m pipelines
- The ideal speedup of superscalar machine
  - $S(m,1) = T(1,1)/T(m,1) = m(N + k – 1)/[N+ m(k – 1)]$
- As n → *infinity*, S(m,1) →m