Module-III

Chapter 5 Bus, Cache and Shared Memory

5.1 Bus Systems

- System bus of a computer operates on contention basis.
- Several active devices such as processors may request use of the bus at the same time.
- Only one of them can be granted access to bus at a time
- The Effective bandwidth available to each processor is inversely proportional to the number of processors contending for the bus.
- For this reason, most bus-based commercial multiprocessors have been small in size.
- The simplicity and low cost of a bus system made it attractive in building small multiprocessors ranging from 4 to 16 processors.

5.1.1 Backplane Bus Specification

- A backplane bus interconnects processors, data storage and peripheral devices in a tightly coupled hardware.
- The system bus must be designed to allow communication between devices on the devices on the bus without disturbing the internal activities of all the devices attached to the bus.
- Timing protocols must be established to arbitrate among multiple requests. Operational rules must be set to ensure orderly data transfers on the bus.
- Signal lines on the backplane are often functionally grouped into several buses as shown in Fig 5.1. Various functional boards are plugged into slots on the backplane. Each slot is provided with one or more connectors for inserting the boards as demonstrated by the vertical arrows.

Data Transfer Bus (DTB)

- Data address and control lines form the data transfer bus (DTB) in VME bus.
- Address lines broadcast data and device address
  - Proportional to log of address space size
- Data lines proportional to memory word length
- Control lines specify read/write, timing, and bus error conditions

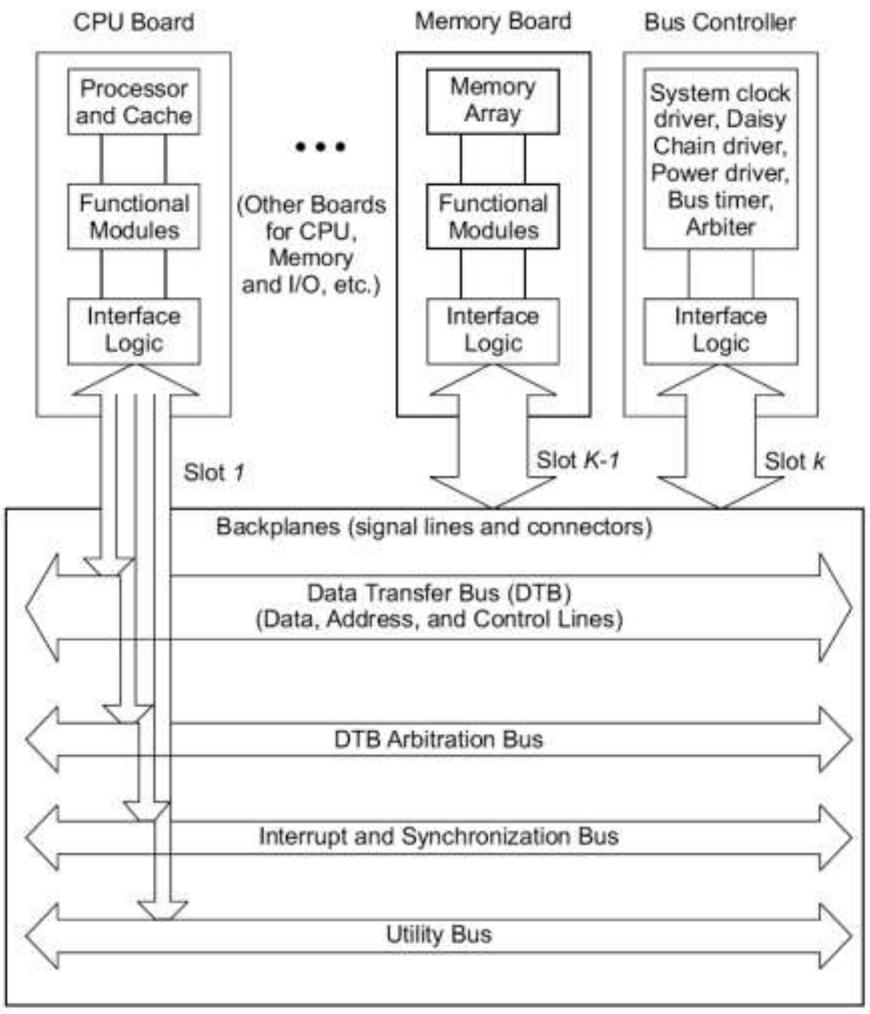

CPU Board
Processor and Cache
Functional Modules
Interface Logic
(Other Boards for CPU, Memory and I/O, etc.)
Memory Board
Memory Array
Functional Modules
Interface Logic
Bus Controller
System clock driver, Daisy Chain driver, Power driver, Bus timer, Arbiter
Interface Logic
Slot 1
Slot K-1
Slot k
Backplanes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

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

Bus Arbitration and Control

- The process of assigning control of the DTB to a requester is called arbitration. Dedicated lines are reserved to coordinate the arbitration process among several requesters.
- The requester is called a master, and the receiving end is called a slave.
- Interrupt lines are used to handle interrupts, which are often prioritized. Dedicated lines may be used to synchronize parallel activities among the processor modules.
- Utility lines include signals that provide periodic timing (clocking) and coordinate the power-up and power-down sequences of the system.
- The backplane is made of signal lines and connectors.
- A special bus controller board is used to house the backplane control logic, such as the system clock driver, arbiter, bus timer, and power driver.

Functional Modules

A functional module is a collection of electronic circuitry that resides on one functional board (Fig. 5.1) and works to achieve special bus control functions.

Special functional modules are introduced below:

- **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 each data transfer takes on the DTB and terminates the DTB cycle if a transfer takes too long.
- **Interrupter** module generates an interrupt request and provides status/ID information when an interrupt handler module requests it.
- **Location monitor** is a functional module that monitors data transfers over the DTB. A power monitor watches the status of the power source and signals when power becomes unstable.
- **System clock driver** is a module that provides a clock timing signal on the utility bus. In addition, board interface logic is needed to match the signal line impedance, the propagation time, and termination values between the backplane and the plug-in boards.

Physical Limitations

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

5.1.2 Addressing and Timing Protocols

- Two types of printed circuit boards connected to a bus: ***active*** and ***passive***
- Active devices like processors can act as bus masters or as slaves at different times.
- Passive devices like memories can act only as slaves.
- The master can initiate a bus cycle
  - Only one can be in control at a time
- The slaves respond to requests by a master
  - Multiple slaves can respond

Bus Addressing

- The backplane bus is driven by a digital clock with a fixed cycle time: *bus cycle*
- Backplane has limited physical size, so will not skew information

- Factors affecting bus delay:
  - Source's line drivers, destination's receivers, slot capacitance, line length, and bus loading effects
- Design should minimize overhead time, so most bus cycles used for useful operations
- Identify each board with a slot number
- When slot number matches contents of high-order address lines, the board is selected as a slave (slot addressing)

Broadcall and Broadcast

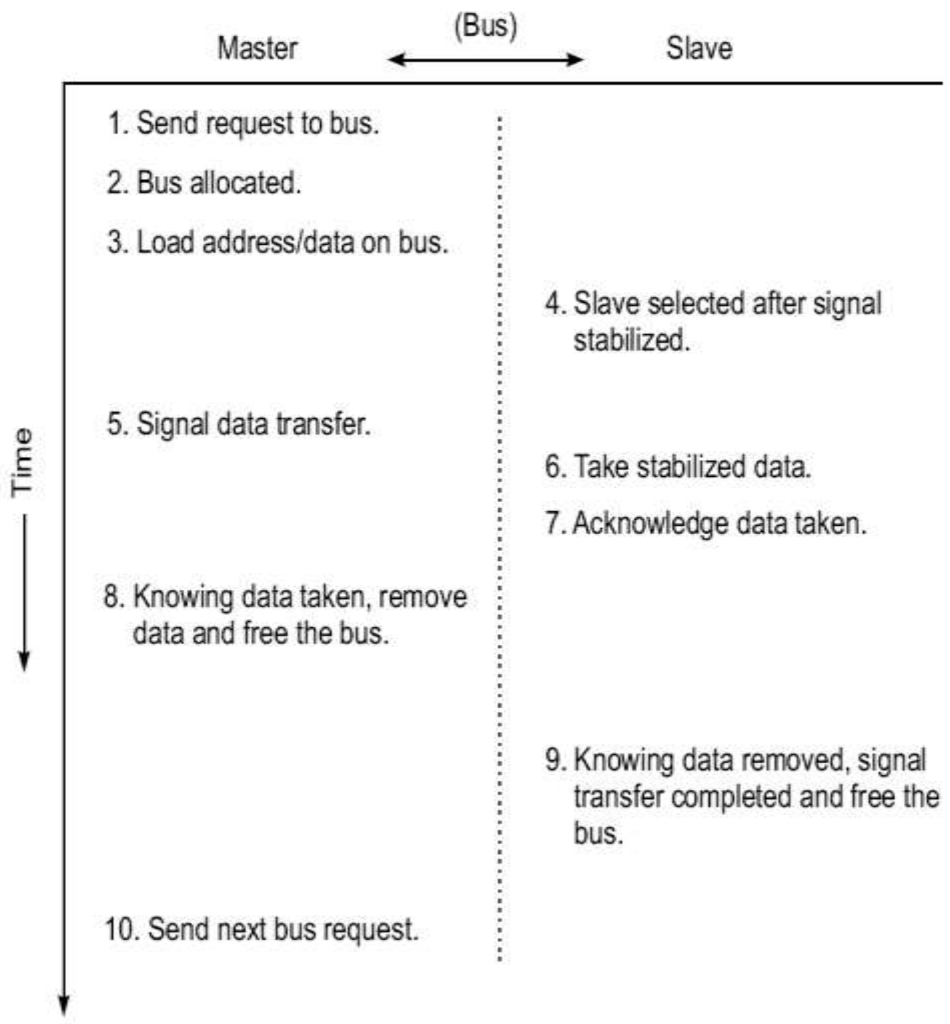

Master
(Bus)
Slave
Time
1. Send request to bus.
2. Bus allocated.
3. Load address/data on bus.
4. Slave selected after signal stabilized.
5. Signal data transfer.
6. Take stabilized data.
7. Acknowledge data taken.
8. Knowing data taken, remove data and free the bus.
9. Knowing data removed, signal transfer completed and free the bus.
10. Send next bus request.

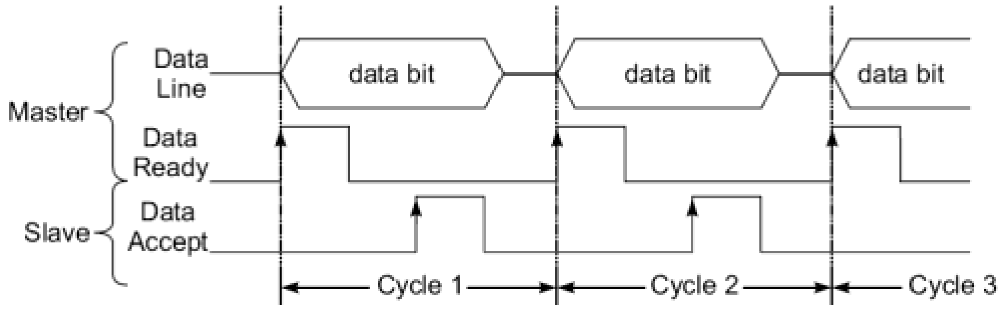
Fig. 5.2 Typical time sequence for information transfer between a master and a slave over a system bus

- Most bus transactions have one slave/master
- **Broadcall**: read operation where multiple slaves place data on bus
  - detects multiple interrupt sources

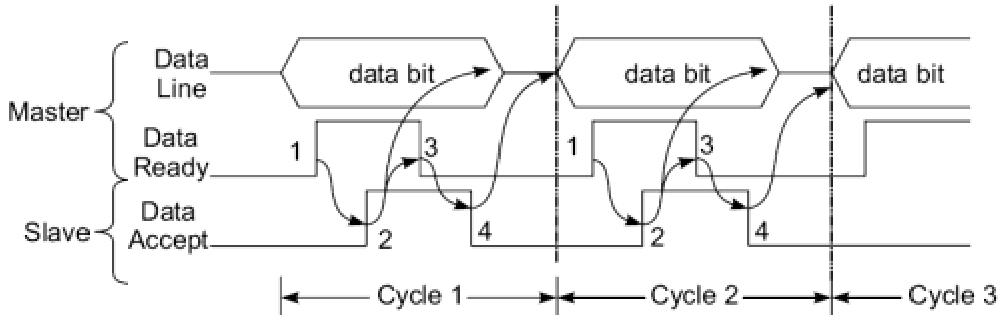
- **Broadcast**: write operation involving multiple slaves
  - Implements multicache coherence on the bus
- Timing protocols are needed to synchronize master and slave operations.
- Figure 5.2 shows a typical timing sequence when information is transferred over a bus from a source to a destination.
- Most bus timing protocols implement such a sequence.

Synchronous Timing

- All bus transaction steps take place at fixed clock edges as shown in Fig. 5.3a.
- The clock signals are broadcast to all potential masters and slaves.
- Clock cycle time determined by slowest device on bus
- Once the data becomes stabilized on the data lines, the master uses Data-ready pulse to initiate the transfer
- The Slave uses Data-accept pulse to signal completion of the information transfer.
- Simple, less circuitry, suitable for devices with relatively the same speed.



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



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

Fig. 5.3 Synchronous versus asynchronous bus timing protocols

Asynchronous Timing

- Based on handshaking or interlocking mechanism as shown in Fig. 5.3b.
- No fixed clock cycle is needed.
- The rising edge (1) of the data-ready signal from the master trioggers the rising (2) of the data-accept signal from the slave.
- The second signal triggers the falling (3) of the data-ready clock and removal of data from the bus.
- The third signal triggers the trailing edge (4) of the data accept clock.
- This four-edge handshaking (interlocking) process is repeated until all the data is transferred.

Advantages: Provides freedom of variable length clock signals for different speed devices

- No response time restrictions
- More flexible

Disadvantage: More complex and costly

5.1.3 Arbitration, Transaction and Interrupt

Arbitration

- Process of selecting next bus master
- Bus tenure is duration of master's control
- It restricts the tenure of the bus to one master at a time.
- Competing requests must be arbitrated on a fairness or priority basis
- Arbitration competition and bus transactions take place concurrently on a parallel bus over separate lines

Central Arbitration

- Uses a central arbiter as shown in Fig 5.4a
- Potential masters are daisy chained in a cascade
- A special signal line propagates *bus-grant* from first master (at slot 1) to the last master (at slot n).
- All requests share the same *bus-request* line
- The *bus-request* signals the rise of the *bus-grant* level, which in turn raises the *bus-busy* level as shown in Fig. 5.4b.

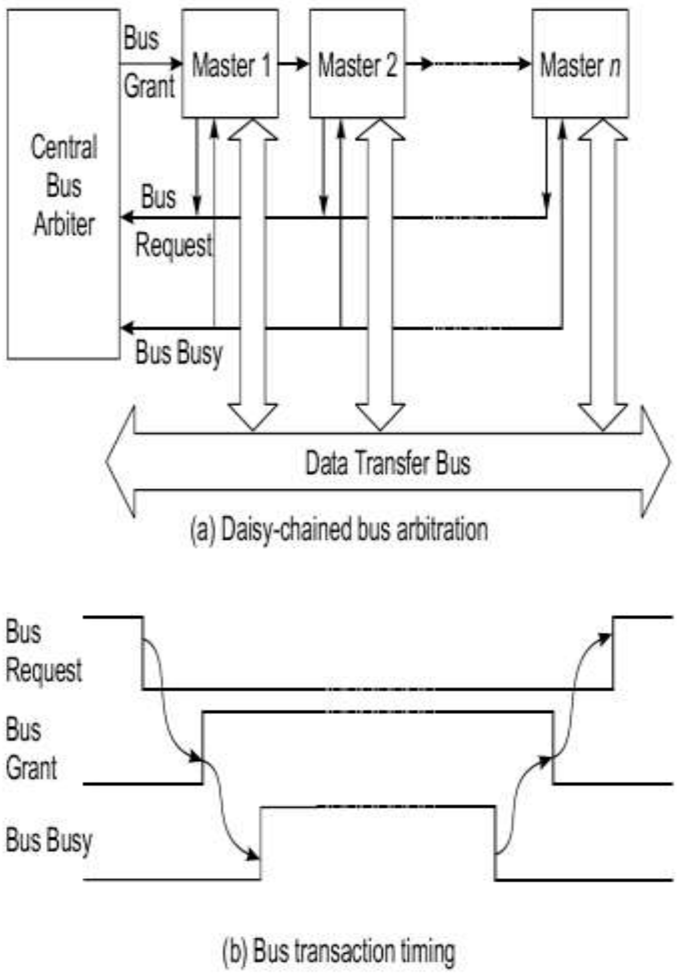
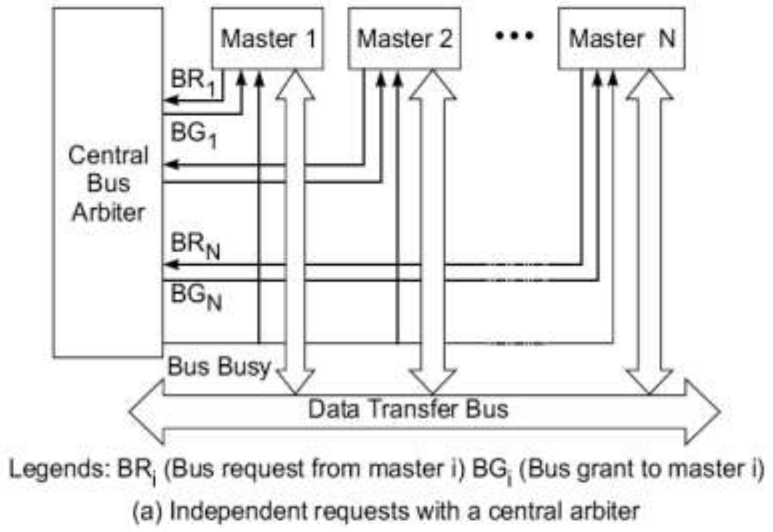


Fig. 5.4 Central bus arbitration using shared requests and daisy-chained bus grants with a fixed priority

- Simple scheme
- Easy to add devices
- Fixed-priority sequence – not fair
- Propagation of bus-grant signal is slow
- Not fault tolerant

Independent Requests and Grants

- Provide independent bus-request and grant signals for each master as shown in Fig5.5a.
- No daisy chaining is used in this scheme.
- Require a central arbiter, but can use a priority or fairness based policy
- More flexible and faster than a daisy-chained policy
- Larger number of lines – costly


Central Bus Arbiter
Master 1
Master 2
Master N
BR1
BG1
BRN
BGN
Bus Busy
Data Transfer Bus
Legends: BRi (Bus request from master i) BGi (Bus grant to master i)
(a) Independent requests with a central arbiter

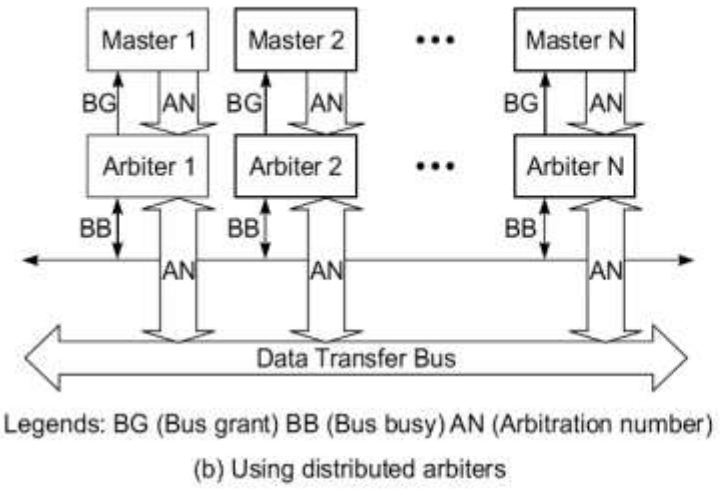

Master 1
Master 2
Master N
BG
AN
Arbiter 1
Arbiter 2
Arbiter N
BB
AN
Data Transfer Bus
Legends: BG (Bus grant) BB (Bus busy) AN (Arbitration number)
(b) Using distributed arbiters

Fig. 5.5 Two bus arbitration schemes using independent requests and distributed arbiters, respectively

Distributed Arbitration

- Each master has its own arbiter and unique arbitration number as shown in Fig. 5.5b.
- Uses arbitration number to resolve arbitration competition
- When two or more devices compete for the bus, the winner is the one whose arbitration number is the largest determined by Parallel Contention Arbitration..
- All potential masters can send their arbitration number to shared-bus request/grant (SBRG) lines and compare its own number with SBRG number.
- If the SBRG number is greater, the requester is dismissed. At the end, the winner's arbitration number remains on the arbitration bus. After the current bus transaction is completed, the winner seizes control of the bus.
- Priority based scheme

Transfer Modes

- ***Address-only* transfer**: no data
- ***Compelled-data* transfer**: Address transfer followed by a block of one or more data transfers to one or more contiguous address.
- ***Packet-data* transfer**: Address transfer followed by a fixed-length block of data transfers from set of continuous address.
- ***Connected***: carry out master's request and a slave's response in a single bus transaction
- ***Split***: splits request and response into separate transactions
  - Allow devices with long latency or access time to use bus resources more efficiently
  - May require two or more connected bus transactions

Interrupt Mechanisms

- ***Interrupt:*** is a request from I/O or other devices to a processor for service or attention
- A priority interrupt bus is used to pass the interrupt signals
- Interrupter must provide status and identification information
- Have an interrupt handler for each request line
- Interrupts can be handled by message passing on data lines on a time-sharing basis.
  - Save lines, but use cycles
  - Use of time-shared data bus lines is a *virtual-interrupt*

5.1.4 IEEE and other Standards

- Open bus standard Futurebus+ 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

The major objectives of the Futurebus+ standards committee were to create a bus standard that would provide a significant step forward in improving the facilities and performance available to the designers of multiprocessor systems.

Below are the design requirements set by the IEEE 896.1-1991 Standards Committee to provide a stable platform on which several generations of computer systems could be based:

- 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 without deadlock or livelock
- 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 multiprocessors.
- Compatible message passing protocols

5.2 Cache Memory Organizations

Cache memory is the fast memory that lies between registers and RAM in memory hierarchy. It holds recently used data and/or instructions.

5.2.1 Cache Addressing Models

- Most multiprocessor systems use private caches for each processor as shown in Fig. 5.6
- Have an interconnection network between caches and main memory
- Caches can be addressed using either a Physical Address or Virtual Address.
- Two different cache design models are:
  - Physical address cache
  - Virtual address cache

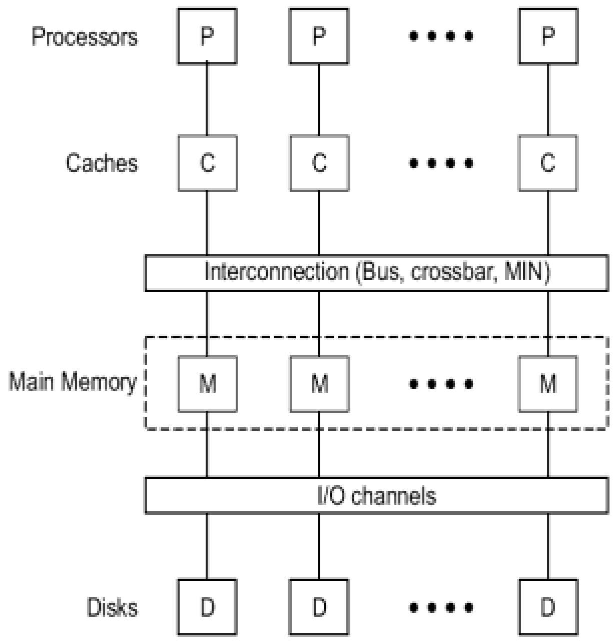


Fig. 5.6 A memory hierarchy for a shared-memory multiprocessor

Physical address cache

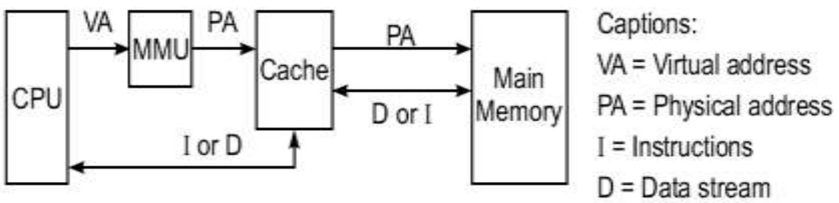
- When cache is addressed by physical address it is called physical address cache. The cache is indexed and tagged with physical address.
- Cache lookup must occur after address translation in TLB or MMU. No aliasing is allowed so that the address is always uniquely translated without confusion.
- After cache miss, load a block from main memory
- Use either write-back or write-through policy

Advantages:

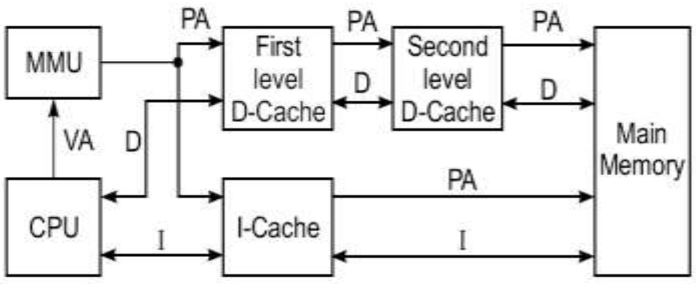
- No cache flushing on a context switch
- No aliasing problem thus fewer cache bugs in OS kernel.
- Simplistic design
- Requires little intervention from OS kernel

Disadvantages:

Slowdown in accessing the cache until the MMU/TLB finishes translating the address



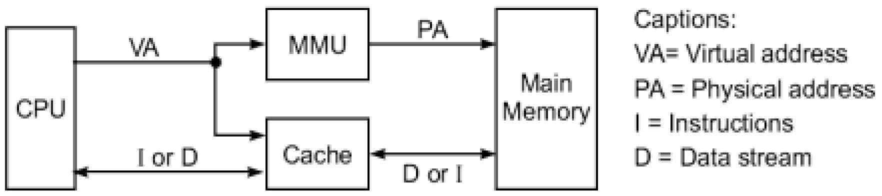
(a) A unified cache accessed by physical address



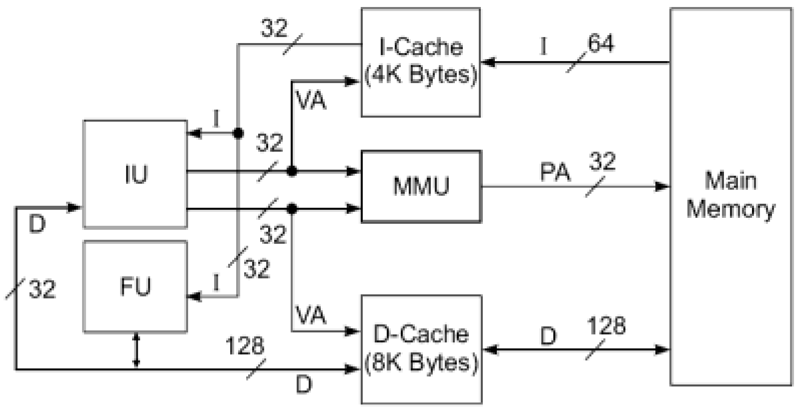
(b) Split caches accessed by physical address in the Silicon Graphics workstation

Fig. 5.7 Physical address models for unified and split caches

Virtual Address caches



(a) A unified cache accessed by virtual address



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

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

- When a cache is indexed or tagged with virtual address it is called virtual address cache.
- In this model both cache and MMU translation or validation are done in parallel.
- The physical address generated by the MMU can be saved in tags for later write back but is not used during the cache lookup operations.

Advantages:

- do address translation only on a cache miss
- faster for hits because no address translation
- More efficient access to cache

Disadvantages:

- Cache flushing on a context switch (example : local data segments will get an erroneous hit for virtual addresses already cached after changing virtual address space, if no cache flushing).
- Aliasing problem (several different virtual addresses cannot span the same physical addresses without being duplicated in cache).

The Aliasing Problem

- The major problem associated with a virtual address cache is aliasing.
- 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

5.2.2 Direct Mapping Cache and Associative Cache

- The transfer of information from main memory to cache memory is conducted in units of cache blocks or cache lines.
- Four block placement schemes are presented below. Each placement scheme has its own merits and demerits.
- The ultimate performance depends upon cache access patterns, organization, and management policy
- Blocks in caches are called **block frames**, and blocks in main memory are called **blocks**
- $\underline{B}_i$ $(i \leq m)$, 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

Direct Mapping Cache

- Direct mapping of **n/m = 2^{s-r}** memory blocks to one block frame in the cache
- Placement is by using modulo-m function. Block B_j is mapped to block frame $\underline{B}_i$

  $$\mathbf{B_j \rightarrow \underline{B}_i \qquad \text{if } i=j \bmod m}$$

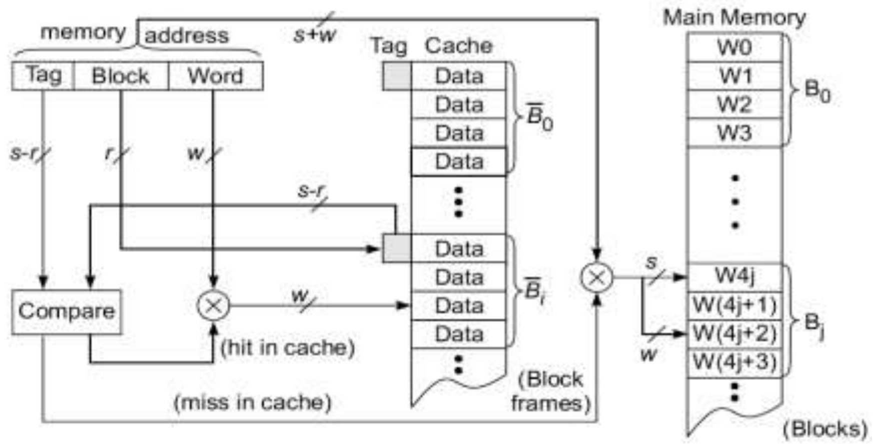
- There is a unique block frame $\underline{B}_i$ that each B_j can load into.
- There is no way to implement a block replacement policy.
- This Direct mapping is very rigid but is the simplest cache organization to implement.

The memory address is divided into 3 fields:

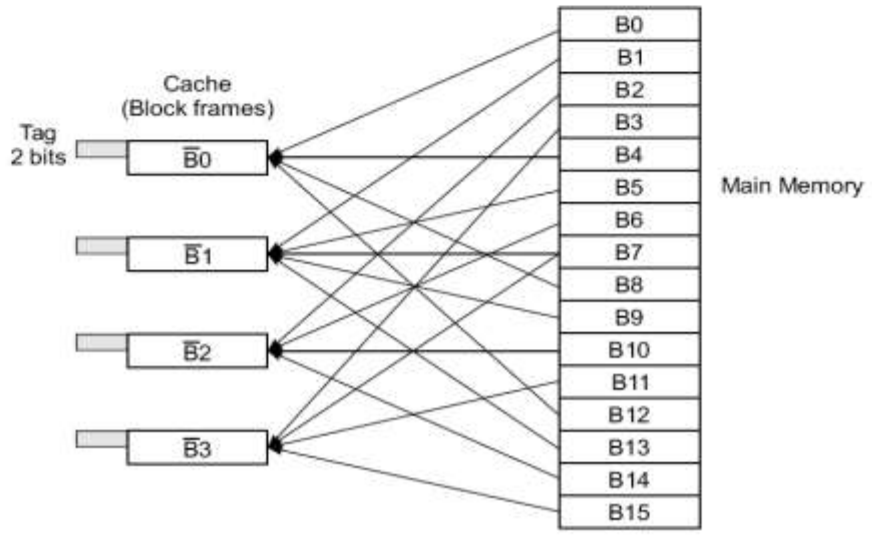
– The lower w bits specify the word offset within each block.

– The upper s bits specify the block address in main memory

– The leftmost (s-r) bits specify the tag to be matched



(a) The cache/memory addressing



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

Fig. 5.9 Direct-mapping cache organization and a mapping example

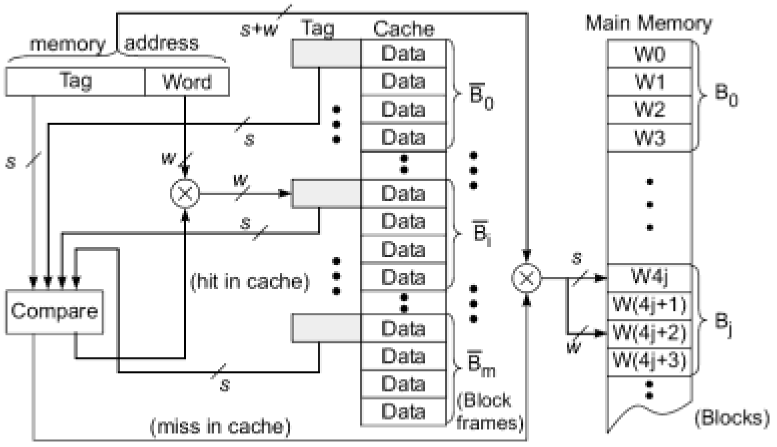
The block field (r bits) is used to implement the (modulo-m) placement, where $\mathbf{m=2^r}$

Once the block $\underline{\mathbf{B_i}}$ is uniquely identified by this field, the tag associated with the addressed block is compared with the tag in the memory address.

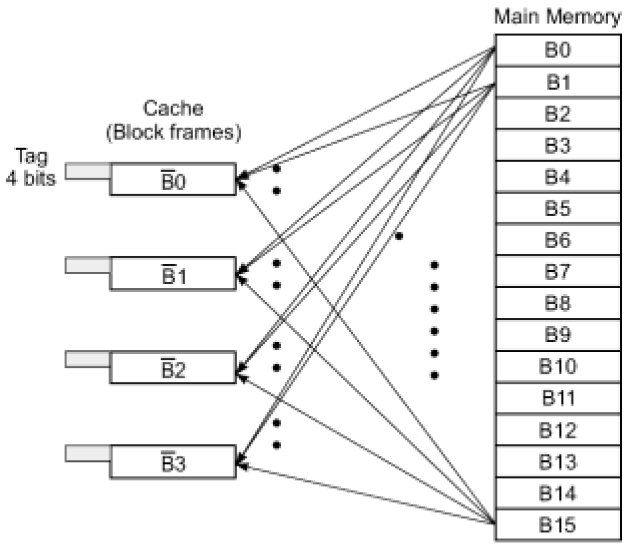
- **Advantages**
  - Simple hardware
  - No associative search
  - No page replacement policy
  - Lower cost
  - Higher speed
- **Disadvantages**
  - Rigid mapping
  - Poorer hit ratio
  - Prohibits parallel virtual address translation
  - Use larger cache size with more block frames to avoid contention

Fully Associative Cache

- Each block in main memory can be placed in any of the available block frames as shown in Fig. 5.10a.
- Because of this flexibility, an s-bit tag needed in each cache block.
- As $s > r$, this represents a significant increase in tag length.
- The name fully associative cache is derived from the fact that an m-way associative search requires tag to be compared with all block tags in the cache. This scheme offers the greatest flexibility in implementing block replacement policies for a higher hit ratio.
- An *m*-way comparison of all tags is very time consuming if the tags are compared sequentially using RAMs. Thus an associative memory is needed to achieve a parallel comparison with all tags simultaneously.
- This demands higher implementation cost for the cache. Therefore, a Fully Associative Cache has been implemented only in moderate size.
- Fig. 5.10b shows a four-way mapping example using a fully associative search. The tag is 4-bits long because 16 possible cache blocks can be destined for the same block frame.



(a) Associative search with all block tags



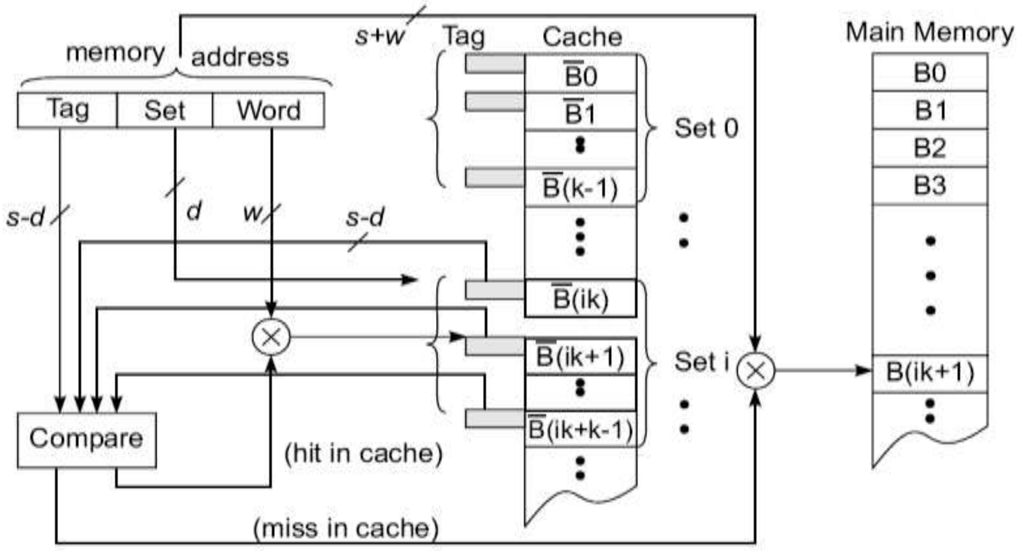
(b) Every block is mapped to any of the four block frames identified by the tag

Fig. 5.10 Fully associative cache organization and a mapping example

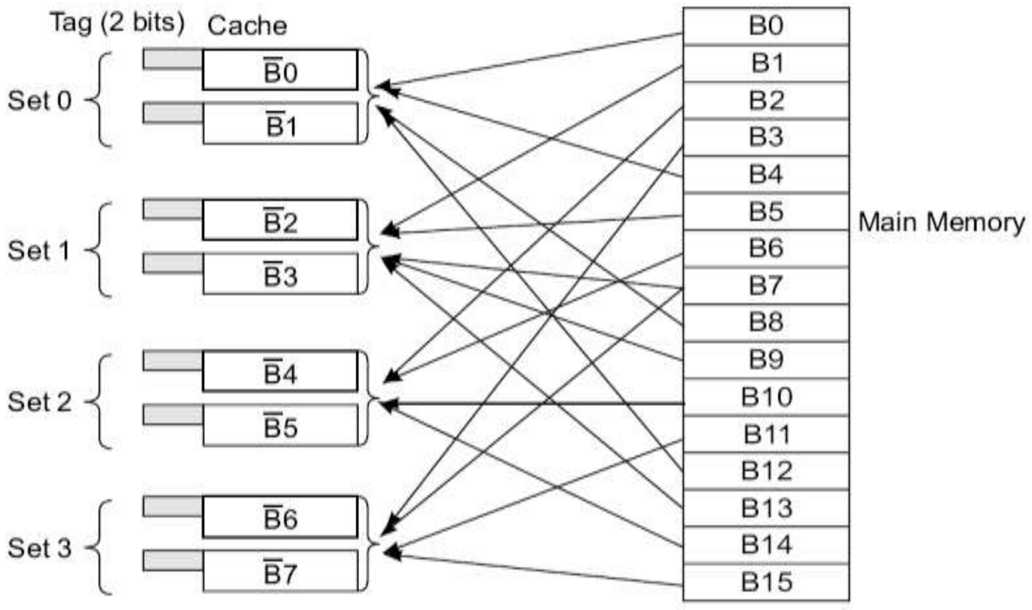
- **Advantages:**
  - Offers most flexibility in mapping cache blocks
  - Higher hit ratio
  - Allows better block replacement policy with reduced block contention
- **Disadvantages:**
  - Higher hardware cost
  - Only moderate size cache
  - Expensive search process

Set Associative Caches

- In a *k*-way associative cache, 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, where $\mathbf{2^d = v}$.
- The cache block tags are now reduced to **s-d** bits.
- In practice, the set size k, or associativity, is chosen as 2, 4, 8, 16 or 64 depending on a tradeoff among block size w, cache size m and other performance/cost factors.



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



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

Fig. 5.11 Set-associative cache organization and a two-way associative mapping example

- Compare the tag with the *k* tags within the identified set as shown in Fig 5.11a.
- Since k is rather small in practice, the k-way associative search is much more economical than the full associativity.
- In general, a block $\mathbf{B_j}$ can be mapped into any one of the available frames $\underline{\mathbf{B_f}}$ in a set $\mathbf{S_i}$ defined below.

$$\mathbf{B_j \rightarrow \underline{B_f} \in S_i \qquad if\ \ j(mod\ v) = i}$$

- The matched tag identifies the current block which resides in the frame.

Sector Mapping Cache

- Partition both the cache and main memory into fixed size sectors. Then use fully associative search ie., each sector can be placed in any of the available sector frames.
- The memory requests are destined for blocks, not for sectors.
- This can be filtered out by comparing the sector tag in the memory address with all sector tags using a fully associative search.
- If a matched sector frame is found (a cache hit), the block field is used to locate the desired block within the sector frame.
- If a cache miss occurs, the missing block is fetched from the main memory and brought into a congruent block frame in available sector.
- That is the *i*th block in a sector must be placed into the ith block frame in a destined sector frame.
- Attach a valid bit to each block frame to indicate whether the block is valid or invalid.

- When the contents of the block frame are replaced from a new sector, the remaining block frames in the same sector are marked invalid. Only the block frames from the most recently referenced sector are marked valid for reference.

Advantages:

- Flexible to implement various bkock replacement algorithms
- Economical to perform a fully associative search a limited number of sector tags.
- Sector partitioning offers more freedom in grouping cache lines at both ends of the mapping.

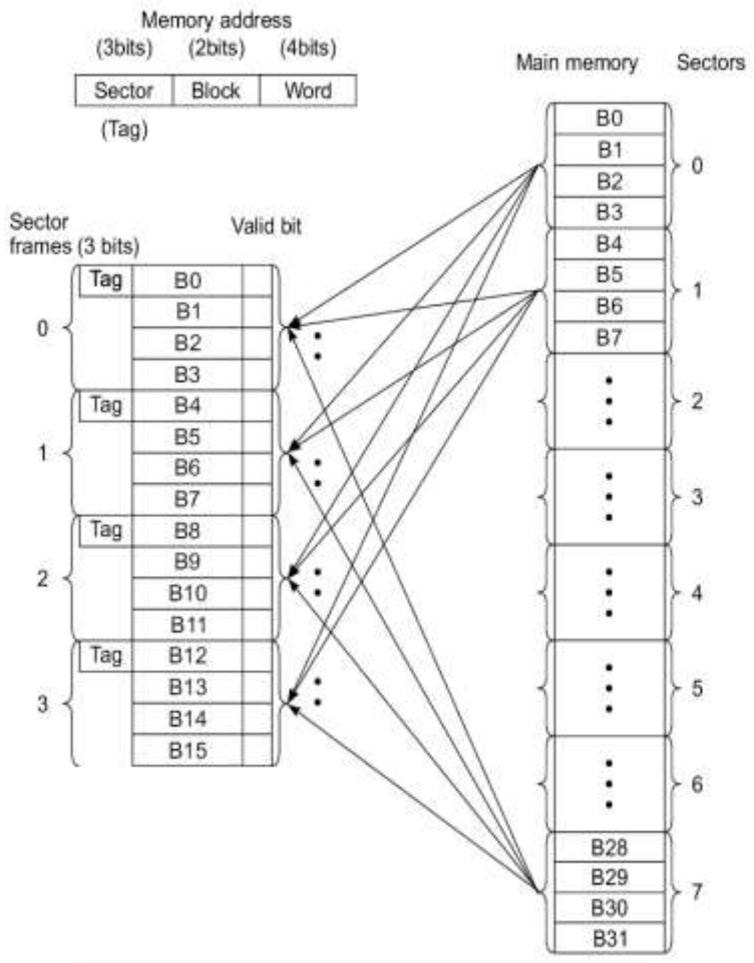


Fig. 5.12 A four-way sector mapping cache organization

4.2.4 Cache Performance Issues

As far as the performance of cache is considered the trade off exist among the cache size, set number, block size and memory speed. Important aspect in cache designing with regard to performance are :

Cycle counts

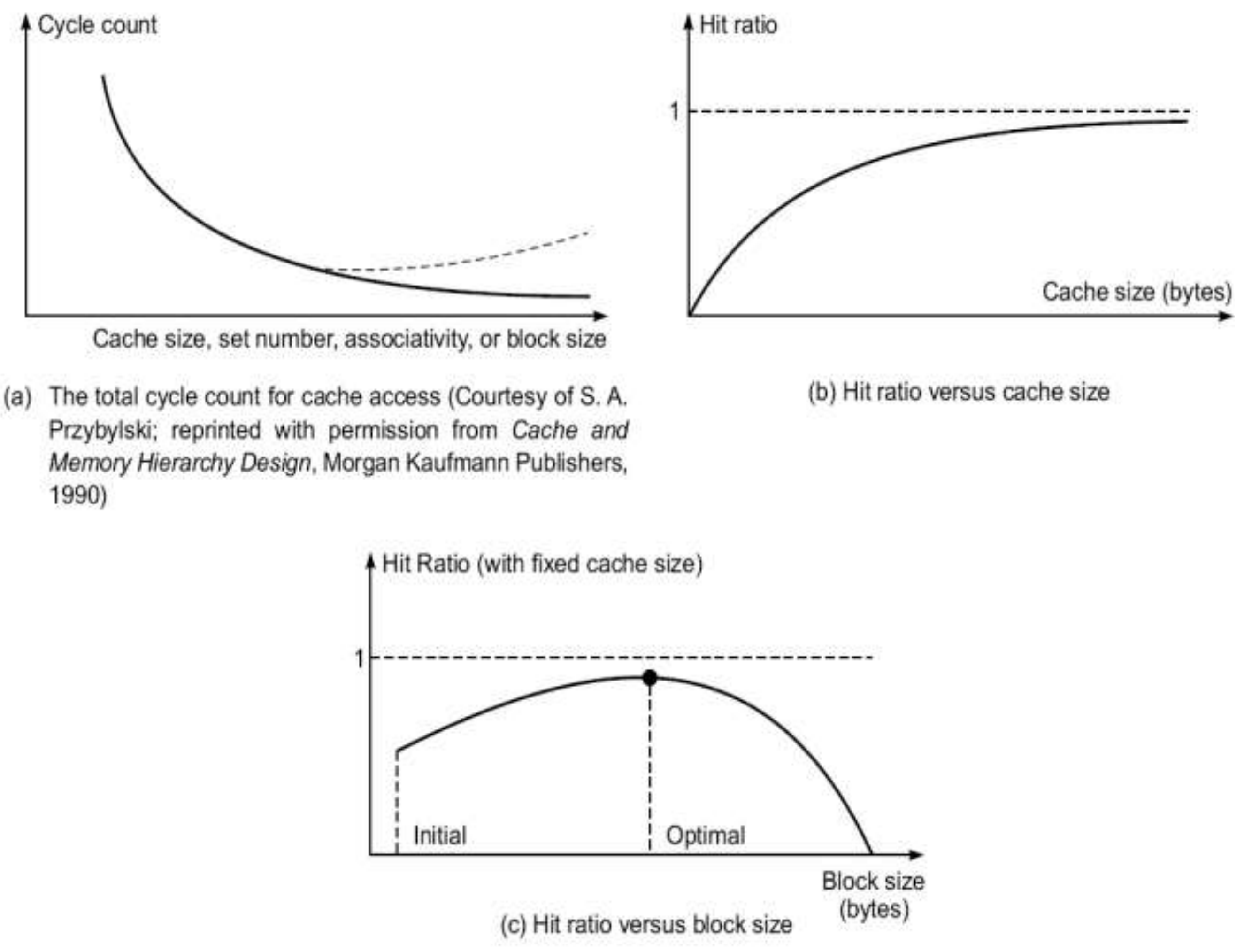
- This refers to the number of basic machine cycles needed for cache access, update and coherence control.
- Cache speed is affected by underlying static or dynamic RAM technology, the cache organization and the cache hit ratios.
- The write through or write back policy also affect the cycle count.
- Cache size, block size, set number, and associativity affect count
- The cycle count is directly related to the hit ratio, which decreases almost linearly with increasing values of above cache parameters.

Hit ratio

- The hit ratio is number of hits divided by total number of CPU references to memory (hits plus misses).
- Hit ratio is 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 a fixed cache size, cache performance is sensitive to the block size.
- 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

Fig. 5.13 Cache performance versus design parameters used

Effect of set number

- In a set associative cache, the effects of set number are obvious.
- For a fixed cache capacity, the hit ratio may decrease as the number of sets increases.
- As the set number increases from 32 to 64, 128 and 256, the decrease in the hit ratio is rather small.
- When the set number increases to 512 and beyond, the hit ratio decreases faster.

5.3 Shared Memory Organizations

Memory interleaving provides a higher bandwidth for pipelined access of continuous memory locations.

Methods for allocating and deallocating main memory to multiple user programs are considered for optimizing memory utilization.

5.3.1 Interleaved Memory Organization

- In order to close up the speed gap between the CPU/cache and main memory built with RAM modules, an *interleaving* technique is presented below which allows pipelined access of the parallel memory modules.
- The memory design goal is to broaden the *effective memory bandwidth* so that more memory words can be accessed per unit time.
- The ultimate purpose is to match the memory bandwidth with the bus bandwidth and with the processor bandwidth.

Memory Interleaving

- The main memory is built with multiple modules.
- These memory modules are connected to a system bus or a switching network to which other resources such as processors or I/O devices are also connected.
- Once presented with a memory address, each memory module returns with one word per cycle.
- It is possible to present different addresses to different memory modules so that parallel access of multiple words can be done simultaneously or in a pipelined fashion.

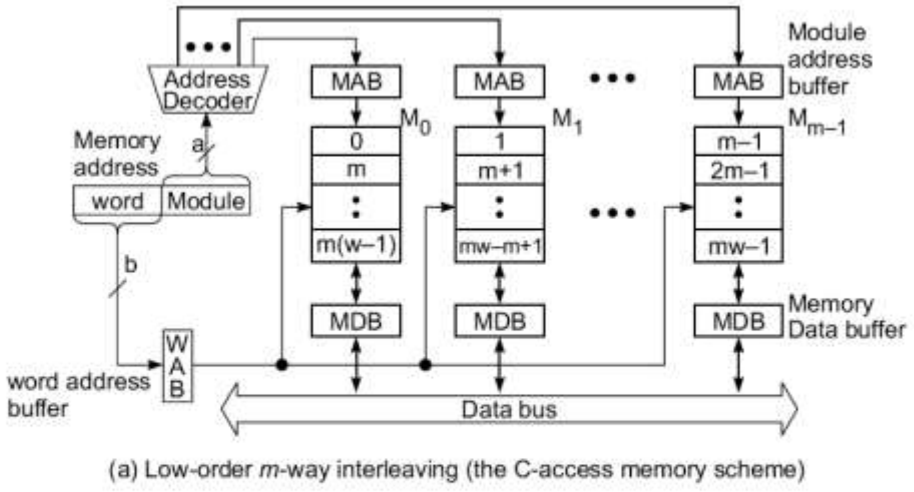
Consider a main memory formed with $\mathbf{m = 2^a}$ memory modules, each containing $\mathbf{w = 2^b}$ words of memory cells. The total memory capacity is $\mathbf{m.w = 2^{a+b}}$ words.

These memory words are assigned linear addresses. Different ways of assigning linear addresses result in different memory organizations.

Besides random access, the main memory is often block-accessed at consecutive addresses.

Figure 5.15 shows two address formats for memory interleaving.

- Low-order interleaving
- High-order interleaving



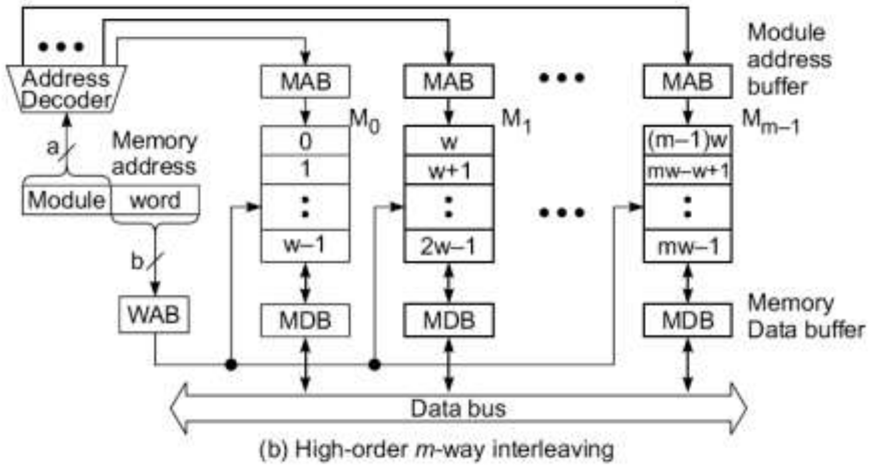


Fig. 5.15 Two interleaved memory organizations with $m = 2^a$ modules and $w = 2^b$ words per module (word addresses shown in boxes)

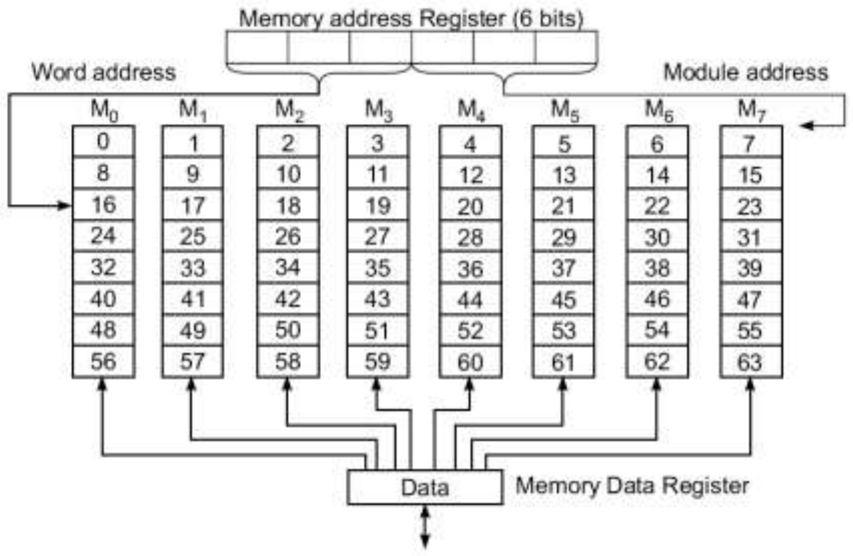
Low-order interleaving

- Low-order interleaving spreads contiguous memory locations across the **m** modules horizontally (Fig. 5.15a).
- This implies that the low-order **a** bits of the memory address are used to identify the memory module.
- The high-order **b** bits are the word addresses (displacement) within each module.
- Note that the same word address is applied to all memory modules simultaneously. A module address decoder is used to distribute module addresses.

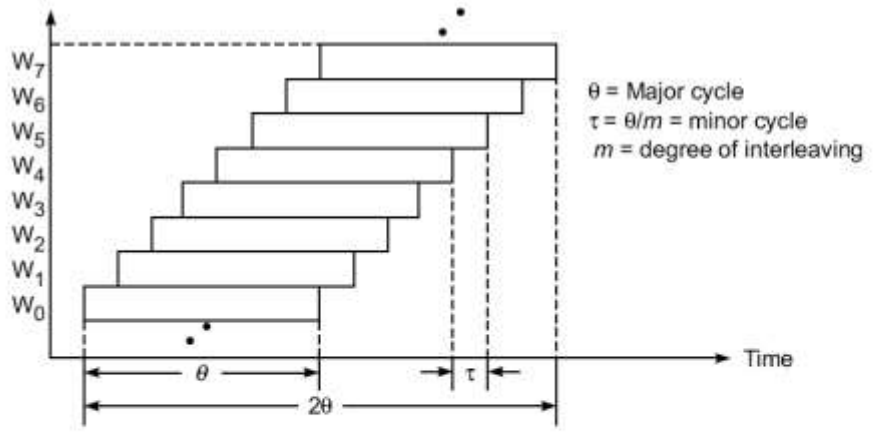
High-order interleaving

- High-order interleaving uses the high-order **a** bits as the module address and the low-order **b** bits as the word address within each module (Fig. 5.15b).
- Contiguous memory locations are thus assigned to the same memory module. In each memory cycle, only one word is accessed from each module.
- Thus the high-order interleaving cannot support block access of contiguous locations.

Pipelined Memory Access



(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

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

- Access of the **m** memory modules can be overlapped in a pipelined fashion.
- For this purpose, the memory cycle (called the *major cycle)* is subdivided into **m** minor cycles.

- An eight-way interleaved memory (with m=8 and w=8 and thus a=b=3) is shown in Fig. 5.16a.
- Let **θ** be the major cycle and **τ** the minor cycle. These two cycle times are related as follows:

  $$\tau = \theta / m$$

  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**
- The timing of the pipelined access of the 8 contiguous memory words is shown in Fig. 5.16b.
- This type of concurrent access of contiguous words has been called a C-access memory scheme.

5.3.2 Bandwidth and Fault Tolerance

Hellerman (1967) has derived an equation to estimate the effective increase in memory bandwidth through multiway interleaving. A single memory module is assumed to deliver one word per memory cycle and thus has a bandwidth of 1.

Memory Bandwidth

The memory bandwidth B of an m-way interleaved memory is upper-bounded by m and lower-bounded by *I*. The Hellerman estimate of *B* is

$$B = m^{0.56} \sim \sqrt{m} \qquad (5.5)$$

where m is the number of interleaved memory modules.

- This equation implies that if 16 memory modules are used, then the effective memory bandwidth is approximately four times that of a single module.
- This pessimistic estimate is due to the fact that block access of various lengths and access of single words are randomly mixed in user programs.
- Hellerman's estimate was based on a single-processor system. If memory-access conflicts from multiple processors (such as the hot spot problem) are considered, the effective memory bandwidth will be further reduced.
- In a vector processing computer, the access time of a long vector with n elements and stride distance 1 has been estimated by Cragon (1992) as follows:
- It is assumed that the *n* elements are stored in contiguous memory locations in an m-way interleaved memory system.

The average time $\mathbf{t_1}$ required to access one element in a vector is estimated by

$$t_1 = \frac{\theta}{m}\left(1 + \frac{m-1}{n}\right) \tag{5.6}$$

When **n** → ∞ (very long vector), $\mathbf{t_1}$ → **θ/m** = **τ**.

As **n** → **1** (scalar access), $\mathbf{t_1}$ → **θ**.

Equation 5.6 conveys the message that interleaved memory appeals to pipelined access of long vectors; the longer the better.

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 bo 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.

5.3.3 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 allocation:**has freedom to preempt an executing process
  - More complex, expensive, and flexible
- **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

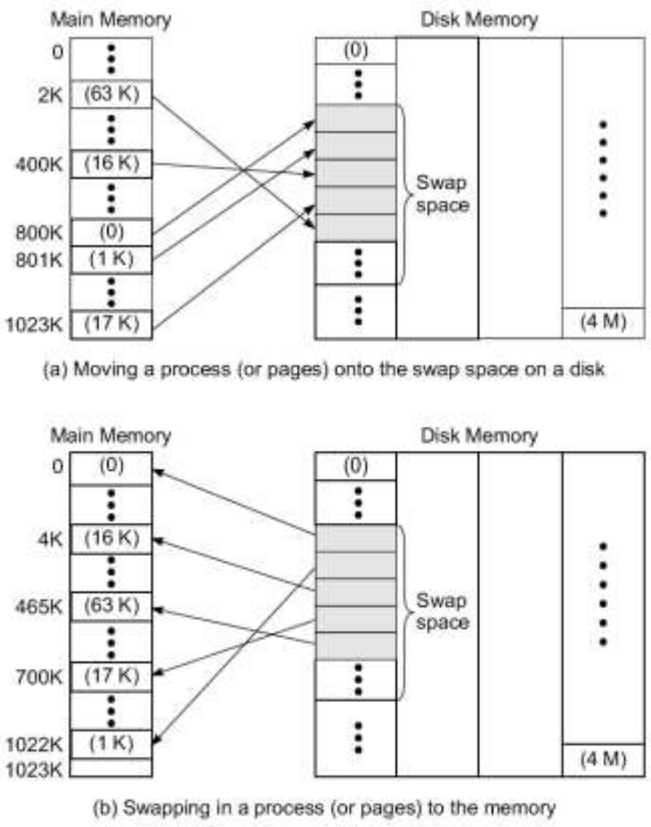


Fig. 5.18 The concept of memory swapping in a virtual memory hierarchy (virtual page addresses are identified by numbers within parentheses, assuming a page size of 1 K words)

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

Other Policies

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

5.4 Sequential and Weak Consistency Models

- **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 between a strong model minimally restricting s/w and a weak model offering efficient implementation
- **Primitive memory operations**: 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

The ***event ordering*** can he used to declare whether a memory event is legal or illegal, when several processes are accessing a common set of memory locations.

A ***program order*** is the order by which memory accesses occur for the execution of a single process, provided that no program reordering has taken place.

Three primitive memory operations for the purpose of specifying memory consistency models are defined:

(1) A ***load*** by processor $\mathbf{P_i}$ is considered ***performed*** with respect to processor $\mathbf{P_k}$ at a point of time when the issuing of a ***store*** to the same location by $\mathbf{P_k}$ cannot affect the value returned by the *load.*

(2) A ***store*** by P, is considered ***performed*** with respect to $\mathbf{P_k}$ at one time when an issued ***load*** to the same address by $\mathbf{P_k}$ returns the value by this ***store***.

(3) A ***load*** is ***globally performed*** if it is performed with respect to all processors and if the ***store*** that is the source of the returned value has been performed with respect to all processors.

- As illustrated in Fig. 5.19a, a processor can execute instructions out of program order using a compiler to resequence instructions in order to boost performance.
- A uniprocessor system allows these out-of-sequence executions provided that hardware interlock mechanisms exist to check data and control dependences between instructions.
- When a processor in a multiprocessor system executes a concurrent program as illustrated in Fig. 5.19b, local dependence checking is necessary but may not be sufficient to preserve the intended outcome of a concurrent execution.

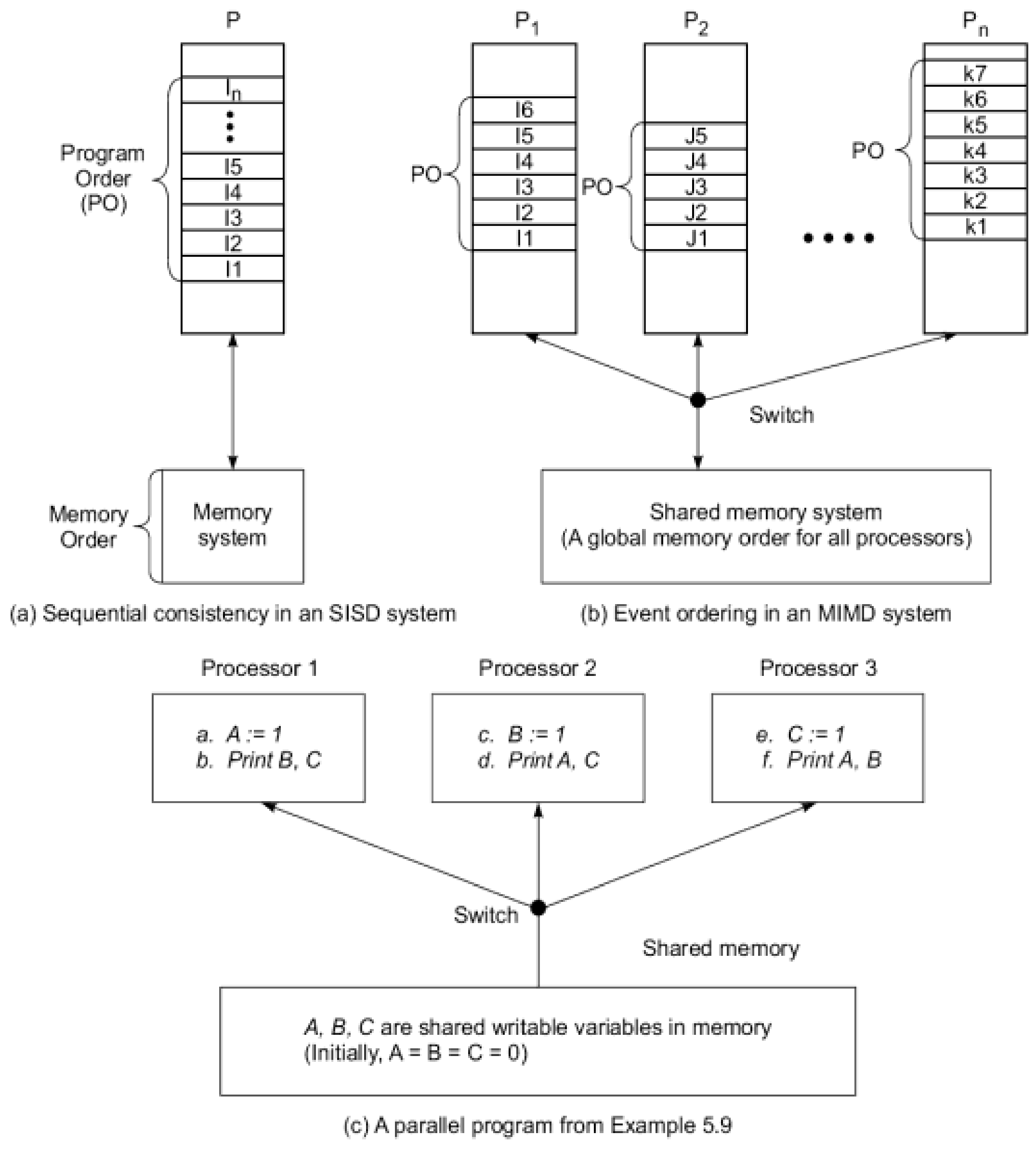

P
In
I5
I4
I3
I2
I1
Program Order (PO)
Memory Order
Memory system
(a) Sequential consistency in an SISD system
P1
I6
I5
I4
I3
I2
I1
PO
P2
J5
J4
J3
J2
J1
PO
Pn
k7
k6
k5
k4
k3
k2
k1
PO
Switch
Shared memory system (A global memory order for all processors)
(b) Event ordering in an MIMD system
Processor 1
a. A := 1
b. Print B, C
Processor 2
c. B := 1
d. Print A, C
Processor 3
e. C := 1
f. Print A, B
Switch
Shared memory
A, B, C are shared writable variables in memory (Initially, A = B = C = 0)
(c) A parallel program from Example 5.9

Fig. 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)

Difficulty in Maintaining Correctness on an MIMD

(a) The order in which instructions belonging to different streams are executed is not fixed in a parallel program. If no synchronization among the instruction streams exists, then a large number of different instruction interleavings is possible.

(b) If for performance reasons the order of execution of instructions belonging to the same stream is different from the program order, then an even larger number of instruction interleavings is passible.

(c) If accesses are not atomic with multiple copies of the same data coexisting as in a cache-based system, then different processors can individually observe different interleavings during the same execution. In this case, the total number of possible execution instantiations of a program becomes even larger.

Atomicity

Three categories of multiprocessor 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

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.

5.4.2 Sequential Consistency Model

- **Sufficient conditions:**
  1. 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
  2. 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

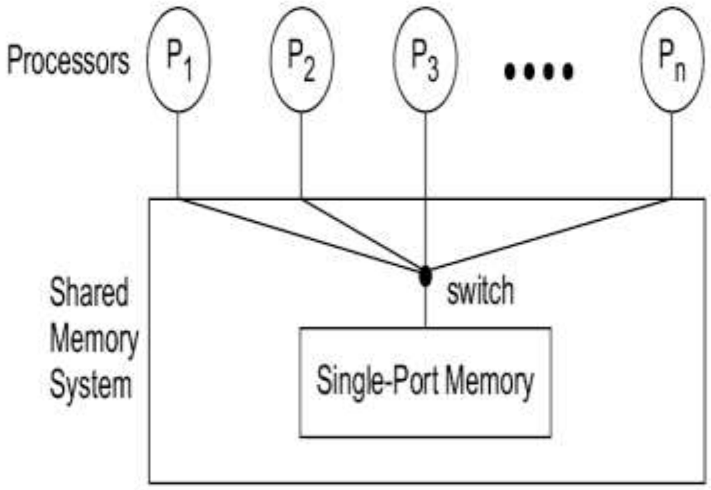


Fig. 5.20 Sequential consistency memory model (Courtesy of Sindhu, Frailong, and Cekleov; reprinted with permission from *Scalable Shared-Memory Multiprocessors*, Kluwer Academic Publishers, 1992)

Sequential Consistency Axioms

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

Implementation Considerations

- A single port software services one op at a time
- Order in which software 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

5.4.3 Weak Consistency Models

- Multiprocessor model may range from strong (sequential) consistency to various degrees of weak consistency
- Two models considered
  - DSB (Dubois, Scheurich and Briggs) model
  - TSO (Total Store Order) model

DSB Model

Dubois, Scheurich and Briggs have derived a weak consistency model by relating memory request ordering to synchronization points in the program. We call this the DSB model specified by the following 3 conditions:

1. All previous *synchronization* accesses must be performed, before a *load* or a *store* access is allowed to perform wrt any other processor.
2. All previous *load and store* accesses must be performed, before a *synchronization* access is allowed to perform wrt any other processor.
3. *Synchronization* accesses sequentially consistent with respect to one another

TSO Model

Sindhu, Frailong and Cekleov have specified the TSO weak consistency model with 6 behavioral axioms.

1. *Load* returns latest *store* result
2. Memory order is a total binary relation over all pairs of *store* operations
3. If two *stores* appear in a particular program order, then they must also appear in the same memory order
4. If a memory operation follows a *load* in program order, then it must also follow *load* in memory order
5. *A swap* operation is atomic with respect to other *stores* – no other *store* can interleave between *load/store* parts of *swap*
6. All *stores and swaps* must eventually terminate.

Chapter-6 Pipelining and Superscalar Techniques

6.1 Linear Pipeline Processors

A linear pipeline processor is a cascade of processing stages which are linearly connected to perform a fixed function over a stream of data flowing from one end to the other.

In modern computers, linear pipelines are applied for instruction execution, arithmetic computation, and memory-access operations.

6.1.l Asynchronous & Synchronous models

- A linear pipeline processor is constructed with k processing stages. External inputs(operands) are fed into the pipeline at the first stage **S_1**.
- The processed results are passed from stage **S_i** to stage **S_{i+1}**, for all i=1,2,….,k-1. The final result emerges from the pipeline at the last stage **S_n**.
- Depending on the control of data flow along the pipeline, we model linear pipelines in two categories: **Asynchronous** and **Synchronous**.

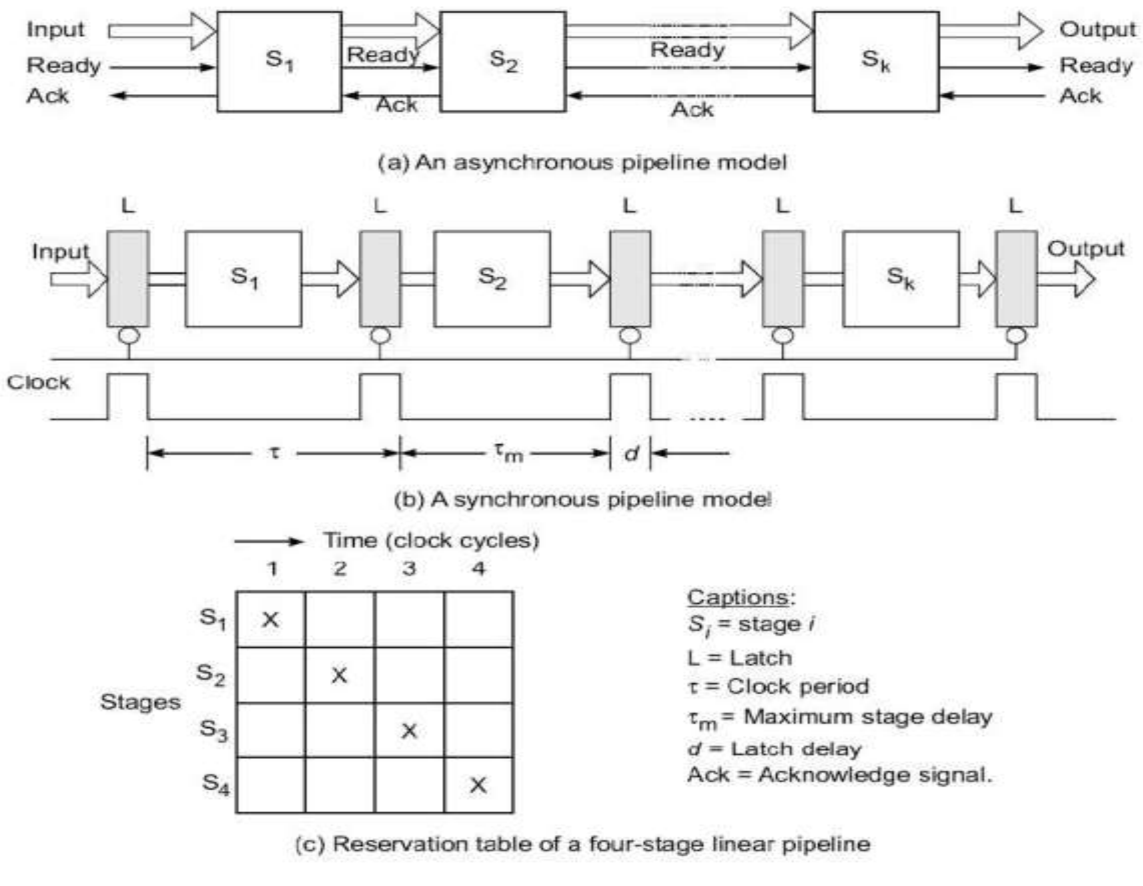


Fig. 6.1 Two models of linear pipeline units and the corresponding reservation table

Asynchronous Model

- As shown in the figure data flow between adjacent stages in an asynchronous pipeline is controlled by a handshaking protocol.
- When stage $\mathbf{S_i}$ is ready to transmit, it sends a ready signal to stage $\mathbf{S_{i+1}}$. After stage receives the incoming data, it returns an acknowledge signal to $\mathbf{S_i}$.
- Asynchronous pipelines are useful in designing communication channels in message- passing multicomputers where pipelined wormhole routing is practiced Asynchronous pipelines may have a variable throughput rate.
- Different amounts of delay may be experienced in different stages.

Synchronous Model:

- Synchronous pipelines are illustrated in Fig. Clocked latches are used to interface between stages.
- The latches are made with master-slave flip-flops, which can isolate inputs from outputs.
- Upon the arrival of a clock pulse All latches transfer data to the next stage simultaneously.
- The pipeline stages are combinational logic circuits. It is desired to have approximately equal delays in all stages.
- These delays determine the clock period and thus the speed of the pipeline. Unless otherwise specified, only synchronous pipelines are studied.
- The utilization pattern of successive stages in a synchronous pipeline is specified by a reservation table.

- For a linear pipeline, the utilization follows the diagonal streamline pattern shown in Fig. 6.1c.
- This table is essentially a space-time diagram depicting the precedence relationship in using the pipeline stages.
- Successive tasks or operations are initiated one per cycle to enter the pipeline. Once the pipeline is filled up, one result emerges from the pipeline for each additional cycle.
- This throughput is sustained only if the successive tasks are independent of each other.

6.1.2 Clocking and Timing Control

The clock cycle $\boldsymbol{\tau}$ of a pipeline is determined below. Let $\boldsymbol{\tau_i}$ be the time delay of the circuitry in stage $\mathbf{S_i}$ and **d** the time delay of a latch, as shown in Fig 6.1b.

Clock Cycle and Throughput :

Denote the maximum stage delay as τ_m ,and we can write τ as

$$\tau = \max_i \{ \tau_i \}_1^k + d = \tau_m + d$$

- At the rising edge of the clock pulse, the data is latched to the master flip-flops of each latch register. The clock pulse has a width equal to **d**.
- In general, $\tau_m >> d$ by one to two orders of magnitude.
- This implies that the maximum stage delay τ_m dominates the clock period. The pipeline frequency is defined as the inverse of the clock period.

  $$f = 1 / \tau$$
- If one result is expected to come out of the pipeline per cycle, f represents the maximum throughput of the pipeline.
- Depending on the initiation rate of successive tasks entering the pipeline, the actual throughput of the pipeline may be lower than f.
- This is because more than one clock cycle has elapsed between successive task initiations.

Clock Skewing:

- Ideally, we expect the clock pulses to arrive at all stages (latches) at the same time.
- However, due to a problem known as clock skewing the same clock pulse may arrive at different stages with a time offset of **s**.
- Let t_{max} be the time delay of the longest logic path within a stage
- t_{min} is the shortest logic path within a stage.
- To avoid a race in two successive stages, we must choose

  $\tau_m >= t_{max} + s$ and $d <= t_{min} - s$
- These constraints translate into the following bounds on the clock period when clock skew takes effect:

  $$d + t_{max} + s <= \tau <= \tau_m + t_{min} - s$$
- In the ideal case $s = 0$, $t_{max} = \tau_m$, and $t_{min} = d$. Thus, we have $\tau = \tau_m + d$

6.1.3 Speedup, Efficiency and Throughput of Pipeline

Ideally, a linear pipeline of *k* stages can process n tasks in $k + (n — 1)$ clock cycles, where *k* cycles are needed to complete the execution of the very first task and the remaining n-1 tasks require $n - 1$ cycles.

- Thus the total time required is

$$T_k = [k + (n - 1)]\tau$$

- where $\boldsymbol{\tau}$ is the clock period.
- Consider an equivalent-function nonpipelined processor which has a flow-through delay of $k\tau$. The amount of time it takes to execute **n** tasks on this nonpipelined processor is,

$$\mathbf{T_1 = nk\tau}$$

Speedup Factor

The speedup factor of a k-stage pipeline over an equivalent nonpipelined processor is defined as

$$S_k = \frac{T_1}{T_k} = \frac{nk\tau}{k\tau + (n-1)\tau} = \frac{nk}{k + (n-1)}$$

Efficiency and Throughput

The efficiency $\mathbf{E_k}$ of a linear k-stage pipeline is defined as

$$E_k = \frac{S_k}{k} = \frac{n}{k + (n-1)}$$

The efficiency approaches **1** when **n** → ∞ , and a lower bound on $\mathbf{E_k}$ is **1/k** when n = 1.

The pipeline throughput H_k is defined as the number of tasks (operations) performed per unit time:

$$H_k = \frac{n}{[k + (n-1)]\tau} = \frac{nf}{k + (n-1)}$$

The maximum throughput f occurs when $\mathbf{E_k}$ → 1 as **n** → ∞.

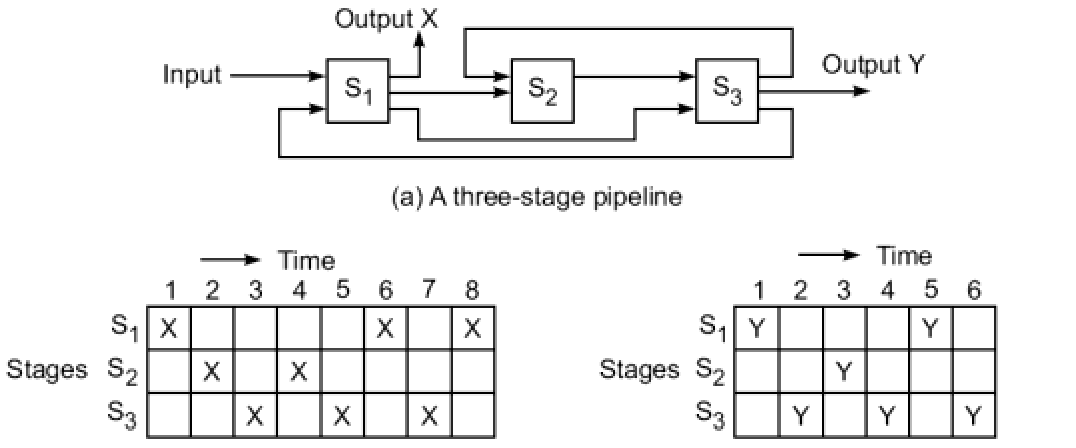
6.2 Non Linear Pipeline Processors

- A dynamic pipeline can be reconfigured to perform variable functions at different times.
- The traditional linear pipelines are static pipelines because they are used to perform fixed functions.
- A dynamic pipeline allows feed forward and feedback connections in addition to the streamline connections.

6.2.1 Reservation and Latency analysis:

- In a static pipeline, it is easy to partition a given function into a sequence of linearly ordered subfunctions.

- However, function partitioning in a dynamic pipeline becomes quite involved because the pipeline stages are interconnected with loops in addition to streamline connections.
- A multifunction dynamic pipeline is shown in Fig 6.3a. This pipeline has three stages.
- Besides the streamline connections from S1 to S2 and from S2 to S3, there is a feed forward connection from S1 to S3 and two feedback connections from S3 to S2 and from S3 to S1.
- These feed forward and feedback connections make the scheduling of successive events into the pipeline a nontrivial task.
- With these connections, the output of the pipeline is not necessarily from the last stage.
- In fact, following different dataflow patterns, one can use the same pipeline to evaluate different functions



(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

Reservation Tables:

- The reservation table for a static linear pipeline is trivial in the sense that data flow follows a linear streamline.
- The reservation table for a dynamic pipeline becomes more interesting because a nonlinear pattern is followed.
- Given a pipeline configuration, multiple reservation tables can be generated for the evaluation of different functions.
- Two reservation tables are given in Fig6.3b and 6.3c, corresponding to a function X and a function Y, respectively.
- Each function evaluation is specified by one reservation table. A static pipeline is specified by a single reservation table.

- A dynamic pipeline may be specified by more than one reservation table. Each reservation table displays the time-space flow of data through the pipeline for one function evaluation.
- Different functions may follow different paths on the reservation table.
- A number of pipeline configurations may be represented by the same reservation table.
- There is a many-to-many mapping between various pipeline configurations and different reservation tables.
- The number of columns in a reservation table is called the evaluation time of a given function.

Latency Analysis

- The number of time units (clock cycles) between two initiations of a pipeline is the latency between them.
- Latency values must be non negative integers. A latency of k means that two initiations are separated by k clock cycles.
- Any attempt by two or more initiations to use the same pipeline stage at the same time will cause a collision.
- A collision implies resource conflicts between two initiations in the pipeline. Therefore, all collisions must be avoided in scheduling a sequence of pipeline initiations.
- Some latencies will cause collisions, and some will not.
- Latencies that cause collisions are called **forbidden latencies.**

6.2.2 Collision Free Scheduling

- When scheduling events in a nonlinear pipeline, the main objective is to obtain the shortest average latency between initiations without causing collisions.
- **Collision Vector**: By examining the reservation table, one can distinguish the set of permissible latencies from the set of forbidden latencies.
- For a reservation table with n columns, the maximum forbidden latency in m<=n-1. The permissible latency p should be as small as possible.
- The choice is made in the range 1 <= p <= m-1.
- A permissible latency of p = 1 corresponds to the ideal case. In theory, a latency of 1 can always be achieved in a static pipeline which follows a linear (diagonal or streamlined) reservation table.

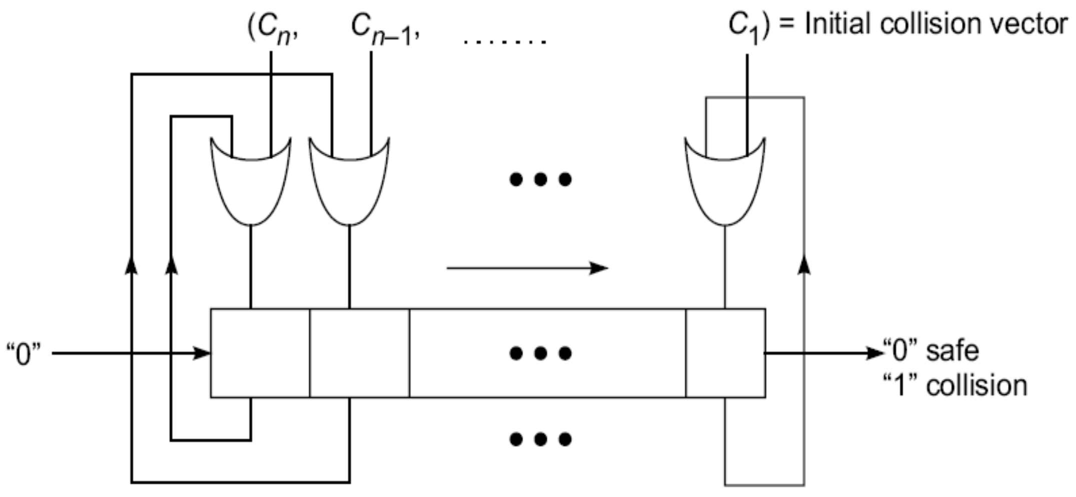
The combined set of permissible and forbidden latencies can be easily displayed by a collision vector, which is an m-bit binary vector $C = (C_m C_{m-1} \dots C_2 C_1)$. The value of $C_i = 1$ if latency i causes a collision and $C_i = 0$ if latency i is permissible. Note that it is always true that $C_m = 1$, corresponding to the maximum forbidden latency.

For the two reservation tables in Fig. 6.3, the collision vector $C_X = (1011010)$ is obtained for function X, and $C_Y = (1010)$ for function Y. From C_X, we can immediately tell that latencies 7, 5, 4, and 2 are forbidden and latencies 6, 3, and 1 are permissible. Similarly, 4 and 2 are forbidden latencies and 3 and 1 are permissible latencies for function Y.

State Diagrams From the above collision vector, one can construct a *state diagram* specifying the permissible state transitions among successive initiations. The collision vector, like C_X above, corresponds to the *initial state* of the pipeline at time 1 and thus is called an *initial collision vector*. Let p be a permissible latency within the range $1 \le p \le m - 1$.

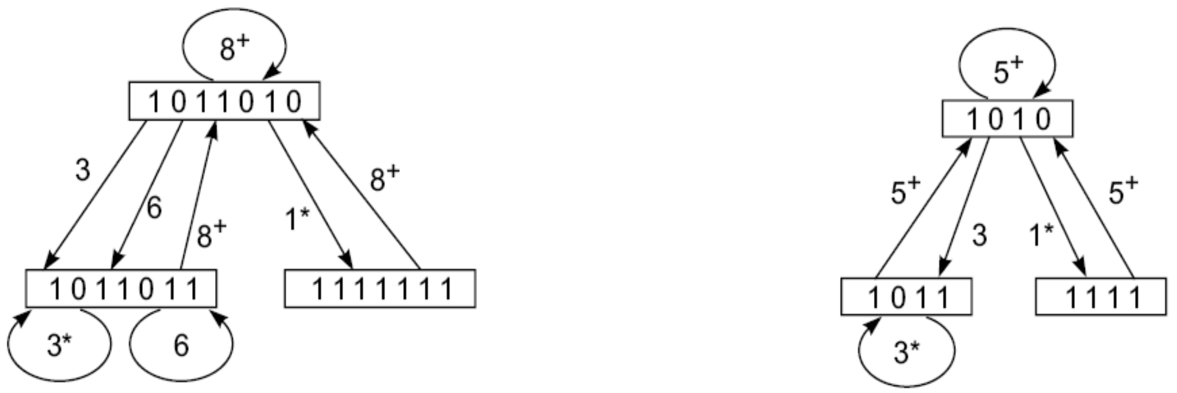
The *next state* of the pipeline at time $t + p$ is obtained with the assistance of an m-bit right shift register as in Fig. 6.6a. The initial collision vector C is initially loaded into the register. The register is then shifted to the right. Each 1-bit shift corresponds to an increase in the latency by 1. When a 0 bit emerges from the right end after p shifts, it means p is a permissible latency. Likewise, a 1 bit being shifted out means a collision, and thus the corresponding latency should be forbidden.

Logical 0 enters from the left end of the shift register. The next state after p shifts is thus obtained by bitwise-ORing the initial collision vector with the shifted register contents. For example, from the initial state $C_X = (1011010)$, the next state (1111111) is reached after one right shift of the register, and the next state (1011011) is reached after three shifts or six shifts.



(C_n, C_{n-1}, ... C_1) = Initial collision vector
"0"
"0" safe
"1" collision

(a) State transition using an n-bit right shift register, where n is the maximum forbidden latency



(b) State diagram for function X

(c) State diagram for function Y

Fig. 6.6 Two state diagrams obtained from the two reservation tables in Fig. 6.3, respectively

Bounds on the MAL In 1972, Shar determined the following bounds on the *minimal average latency* (MAL) achievable by any control strategy on a statically reconfigured pipeline executing a given reservation table:

(1) The MAL is lower-bounded by the maximum number of checkmarks in any row of the reservation table.

(2) The MAL is lower than or equal to the average latency of any greedy cycle in the state diagram.

(3) The average latency of any greedy cycle is upper-bounded by the number of 1's in the initial collision vector plus 1. This is also an upper bound on the MAL.

6.3 Instruction Pipeline Design

6.3.1 Instruction Execution Phases

A typical instruction execution consists of a sequence of operations, including instruction fetch, decode, operand fetch, execute, and write-back phases. These phases are ideal for overlapped execution on a linear pipeline.

Pipelined Instruction Processing A typical instruction pipeline is depicted in Fig. 6.9. The *fetch stage* (F) fetches instructions from a cache memory, ideally one per cycle. The *decode stage* (D) reveals the instruction function to be performed and identifies the resources needed. Resources include general-purpose registers, buses, and functional units. The *issue stage* (I) reserves resources. The operands are also read from registers during the issue stage.

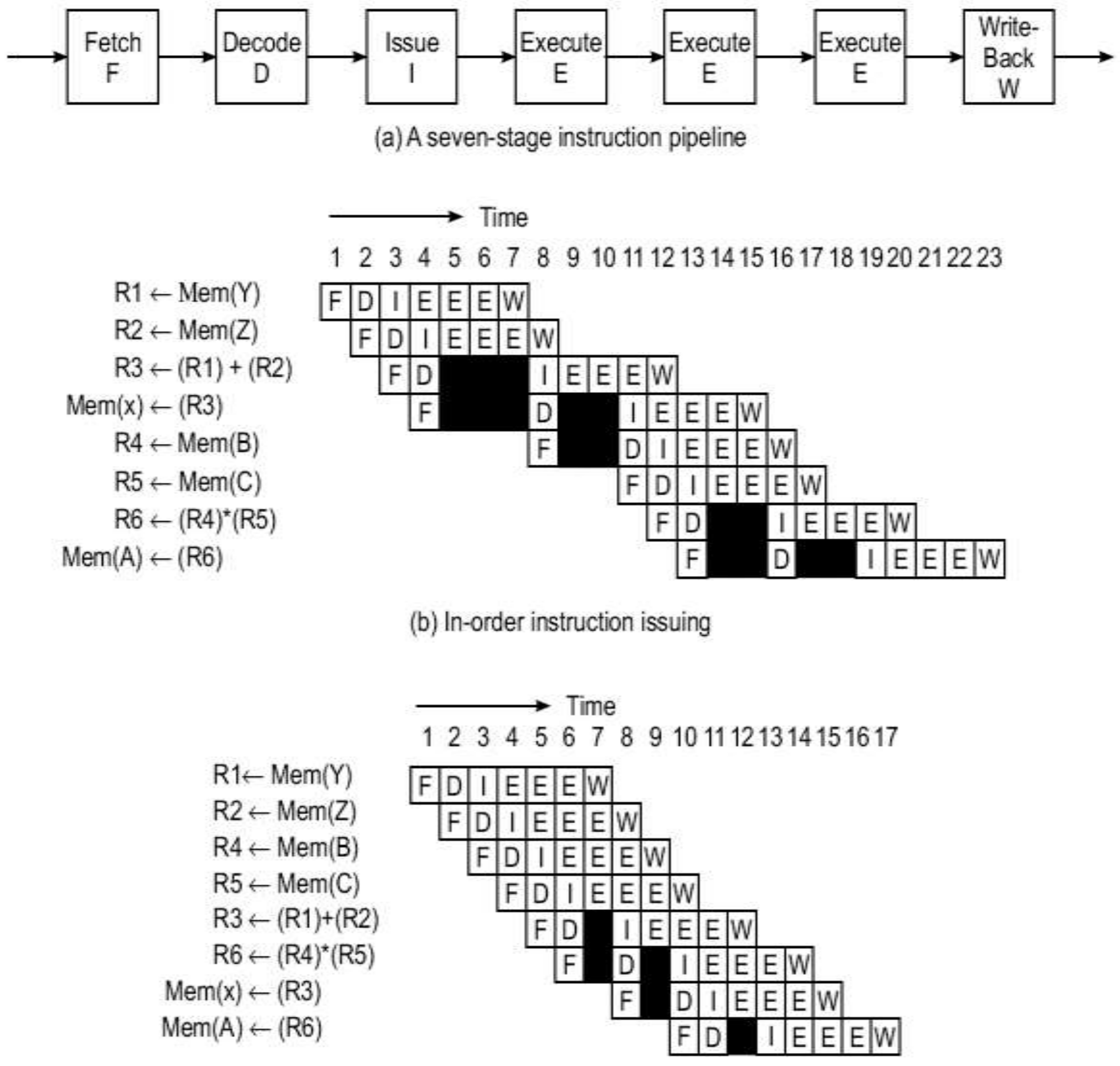


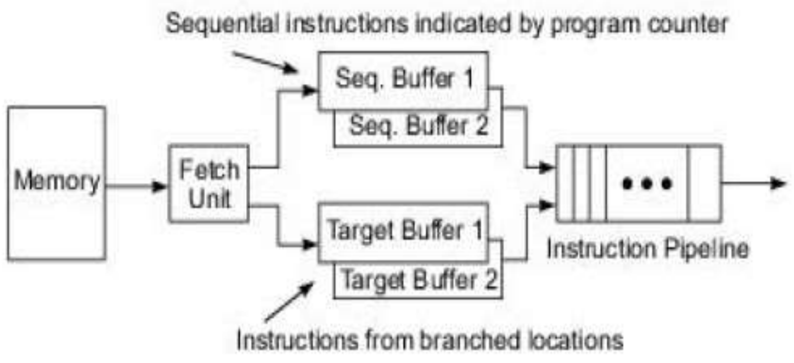
Fig. 6.9 Pipelined execution of X = Y + Z and A = B × C (Courtesy of James Smith; reprinted with permission from *IEEE Computer*, July 1989)

The instructions are executed in one or several *execute stages* (E). Three execute stages are shown in Fig. 6.9a. The last *writeback stage* (W) is used to write results into the registers. Memory load or store operations are treated as part of execution. Figure 6.9 shows the flow of machine instructions through a typical pipeline. These eight instructions are for pipelined execution of the high-level language statements X = Y + Z and A = B × C. Here we have assumed that *load* and *store* instructions take four execution clock cycles, while floating-point *add* and *multiply* operations take three cycles.

6.3.2 Mechanisms for Instruction Pipelining

We introduce instruction buffers and describe the use of cacheing, collision avoidance, multiple functional units, register tagging, and internal forwarding to smooth pipeline flow and to remove bottlenecks and unnecessary memory access operations.

Prefetch Buffers Three types of buffers can be used to match the instruction fetch rate to the pipeline consumption rate. In one memory-access time, a block of consecutive instructions are fetched into a prefetch buffer as illustrated in Fig. 6.11. The block access can be achieved using interleaved memory modules or using a cache to shorten the effective memory-access time as demonstrated in the MIPS R4000.



Sequential instructions are loaded into a pair of *sequential buffers* for in-sequence pipelining. Instructions from a branch target are loaded into a pair of *target buffers* for out-of-sequence pipelining. Both buffers operate in a first-in-first-out fashion. These buffers become part of the pipeline as additional stages.

A conditional branch instruction causes both sequential buffers and target buffers to fill with instructions. After the branch condition is checked, appropriate instructions are taken from one of the two buffers, and instructions in the other buffer are discarded. Within each pair, one can use one buffer to load instructions from memory and use another buffer to feed instructions into the pipeline. The two buffers in each pair alternate to prevent a collision between instructions flowing into and out of the pipeline.

A third type of prefetch buffer is known as a *loop buffer*. This buffer holds sequential instructions contained in a small loop. The loop buffers are maintained by the fetch stage of the pipeline. Prefetched instructions in the loop body will be executed repeatedly until all iterations complete execution. The loop buffer operates in two steps. First, it contains instructions sequentially ahead of the current instruction. This saves the instruction fetch time from memory. Second, it recognizes when the target of a branch falls within the loop boundary. In

Multiple Functional Units Sometimes a certain pipeline stage becomes the bottleneck. This stage corresponds to the row with the maximum number of checkmarks in the reservation table. This bottleneck problem can be alleviated by using multiple copies of the same stage simultaneously. This leads to the use of multiple execution units in a pipelined processor design (Fig. 6.12).

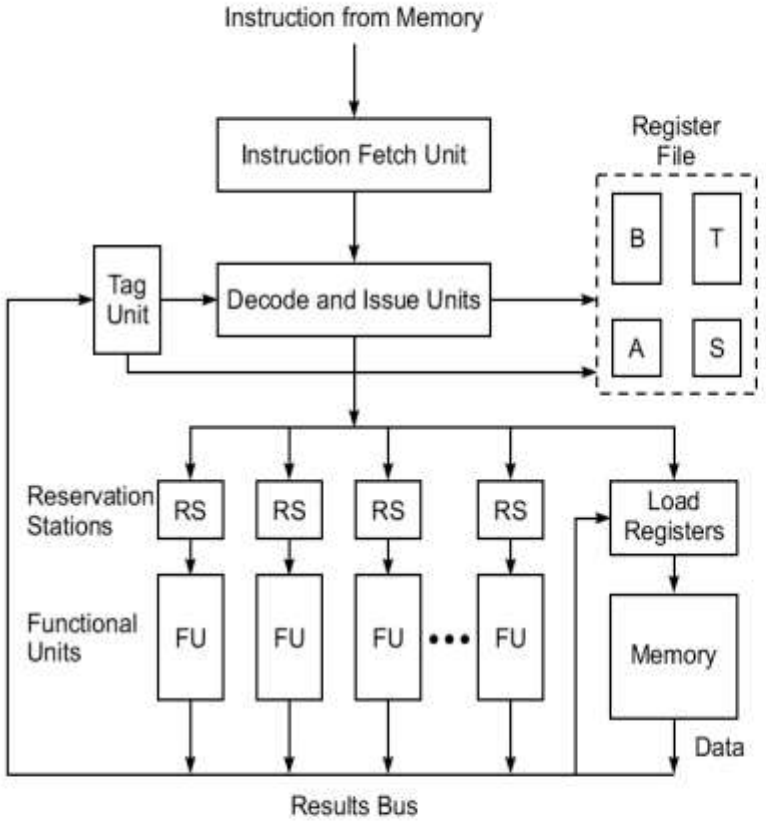


Fig. 6.12 A pipelined processor with multiple functional units and distributed reservation stations supported by tagging (Courtesy of G. Sohi; reprinted with permission from *IEEE Transactions on Computers*, March 1990)

Internal Data Forwarding The throughput of a pipelined processor can be further improved with internal data forwarding among multiple functional units. In some cases, some memory-access operations can be replaced by register transfer operations. The idea is described in Fig. 6.13.

A *store-load forwarding* is shown in Fig. 6.13a in which the *load operation* (LD R2, M) from memory to register R2 can be replaced by the *move* operation (MOVE R2, R1) from register R1 to register R2. Since register transfer is faster than memory access, this data forwarding will reduce memory traffic and thus results in a shorter execution time. Similarly, *load-load forwarding* (Fig. 6.13b) eliminates the second

Load operation (LD R2, M) and replaces it with the move operation (MOVE R2, R1).

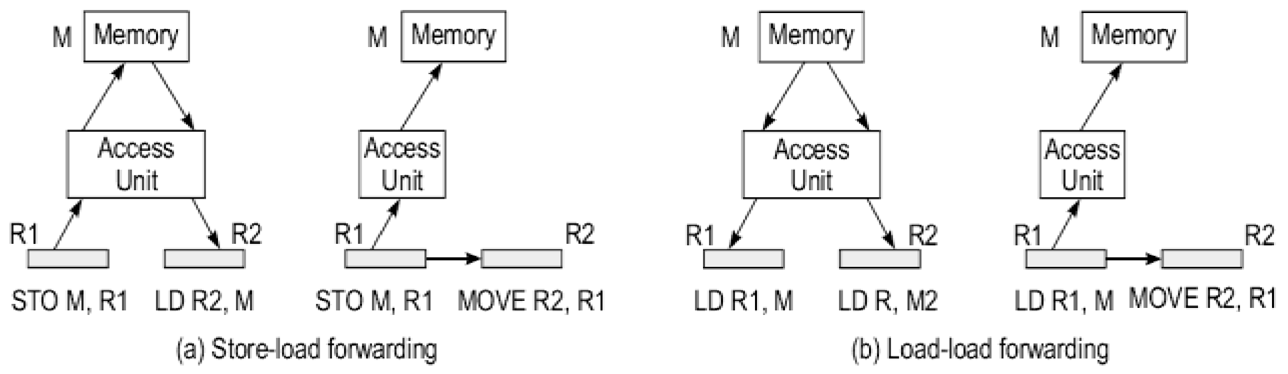


Fig. 6.13 Internal data forwarding by replacing memory-access operations with register transfer operations

Hazard Avoidance

- The read and write of shared variables by different instructions in a pipeline may lead to different results if these instructions are executed out of order.
- As shown in Fig. 6.15, three types of logic hazards are possible:
- Consider two instructions I and J. Instruction J is assumed to logically follow instruction I according to program order.

- If the actual execution order of these two instructions violate the program order, incorrect results may be read or written, thereby producing hazards.
- Hazards should be prevented before these instructions enter the pipeline, such as by holding instruction J until the dependence on instruction I is resolved.

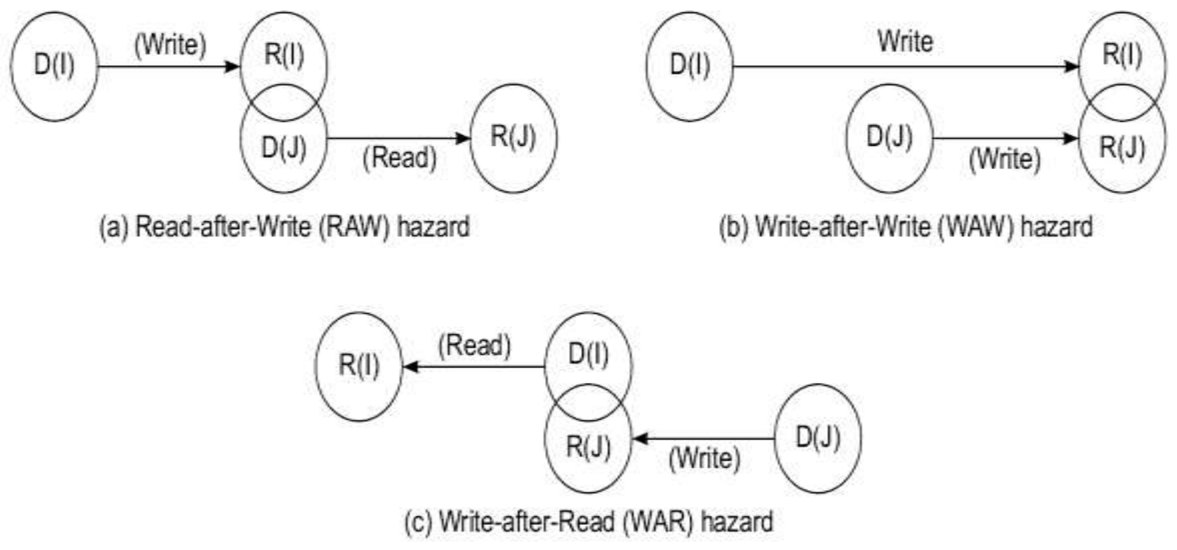

D(I)
(Write)
R(I)
D(J)
(Read)
R(J)
(a) Read-after-Write (RAW) hazard
Write
(Write)
(b) Write-after-Write (WAW) hazard
(c) Write-after-Read (WAR) hazard

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

- We use the notation D(I) and R(I) for the domain and range of an instruction I.
  - **Domain** contains the Input Set to be used by instruction I
  - **Range** contains the Output Set of instruction I

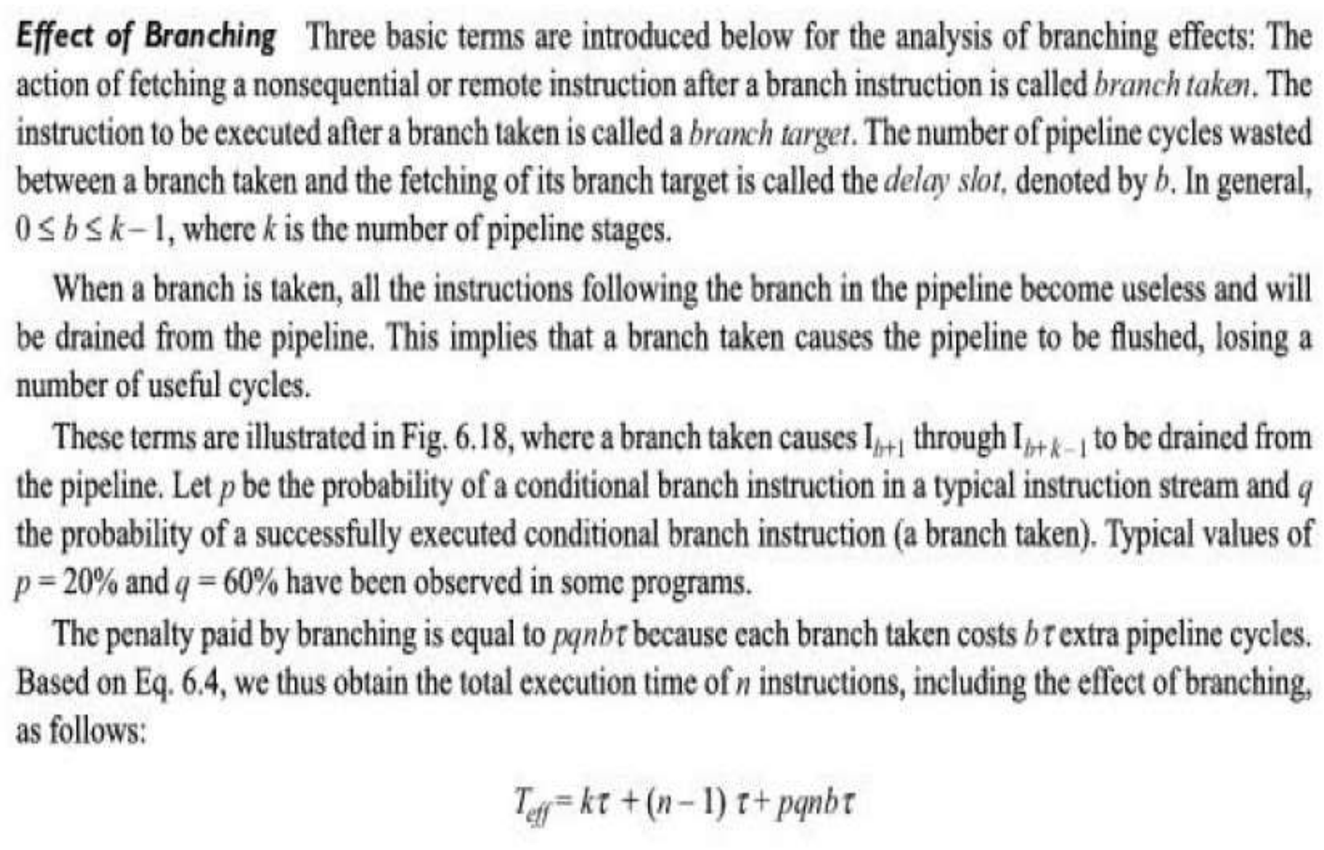
Listed below are conditions under which possible hazards can occur:

R(I) ∩ D(J) ≠ ϕ for RAW hazard (Flow Dependence)

R(I) ∩ R(J) ≠ ϕ for WAW hazard (Anti Dependence)

D(I) ∩ R(J) ≠ ϕ for WAR hazard (Output Dependence)

6.3.4 Branch Handling Techniques



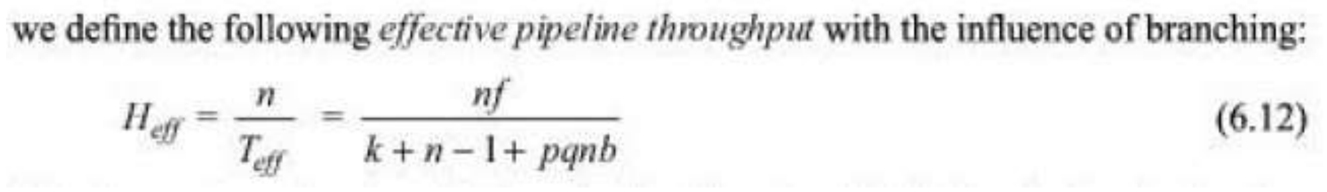
Effect of Branching Three basic terms are introduced below for the analysis of branching effects: The action of fetching a nonsequential or remote instruction after a branch instruction is called *branch taken*. The instruction to be executed after a branch taken is called a *branch target*. The number of pipeline cycles wasted between a branch taken and the fetching of its branch target is called the *delay slot*, denoted by b. In general, $0 \le b \le k-1$, where k is the number of pipeline stages.

When a branch is taken, all the instructions following the branch in the pipeline become useless and will be drained from the pipeline. This implies that a branch taken causes the pipeline to be flushed, losing a number of useful cycles.

These terms are illustrated in Fig. 6.18, where a branch taken causes I_{b+1} through I_{b+k-1} to be drained from the pipeline. Let p be the probability of a conditional branch instruction in a typical instruction stream and q the probability of a successfully executed conditional branch instruction (a branch taken). Typical values of $p = 20\%$ and $q = 60\%$ have been observed in some programs.

The penalty paid by branching is equal to $pqnb\tau$ because each branch taken costs $b\tau$ extra pipeline cycles. Based on Eq. 6.4, we thus obtain the total execution time of n instructions, including the effect of branching, as follows:

$$T_{eff} = k\tau + (n-1)\tau + pqnb\tau$$



we define the following *effective pipeline throughput* with the influence of branching:

$$H_{eff} = \frac{n}{T_{eff}} = \frac{nf}{k+n-1+pqnb} \qquad (6.12)$$

When $n \to \infty$, the tightest upper bound on the effective pipeline throughput is obtained when $b = k - 1$:

$$H^*_{eff} = \frac{f}{pq(k-1)+1} \tag{6.13}$$

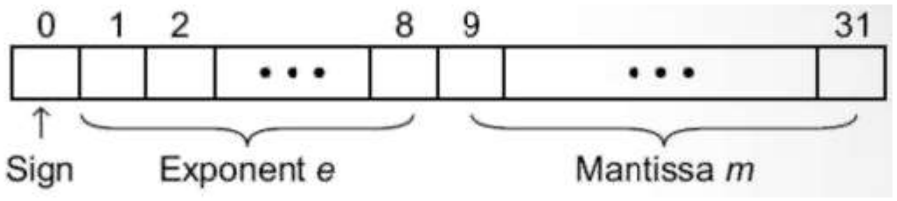
Fixed-Point Operations Fixed-point numbers are represented internally in machines in *sign-magnitude, one's complement,* or *two's complement* notation. Most computers use the two's complement notation because of its unique representation of all numbers (including zero). One's complement notation introduces a second zero representation called *dirty zero*.

Add, subtract, multiply, and *divide* are the four primitive arithmetic operations. For fixed-point numbers, the add or subtract of two n-bit integers (or fractions) produces an n-bit result with at most one carry-out.

The multiplication of two n-bit numbers produces a $2n$-bit result which requires the use of two memory words or two registers to hold the full-precision result.

The division of an n-bit number by another may create an arbitrarily long quotient and a remainder. Only an approximate result is expected in fixed-point division with rounding or truncation. However, one can expand the precision by using a $2n$-bit dividend and an n-bit divisor to yield an n-bit quotient.

Floating-Point Numbers A floating-point number X is represented by a pair (m, e), where m is the *mantissa* (or *fraction*) and e is the *exponent* with an implied *base* (or *radix*). The algebraic value is represented as $X = m \times r^e$. The sign of X can be embedded in the mantissa.



A binary base is assumed with $r = 2$. The 8-bit exponent e field uses an *excess-127* code. The dynamic range of e is $(-127, 128)$, internally represented as $(0, 255)$. The sign s and the 23-bit mantissa field m form a 25-bit sign-magnitude fraction, including an implicit or "hidden" 1 bit to the left of the binary point. Thus the complete mantissa actually represents the value $1.m$ in binary.

This hidden bit is not stored with the number. If $0 < e < 255$, then a nonzero normalized number represents the following algebraic value:

$$X = (-1)^s \times 2^{e-127} \times (1.m) \tag{6.15}$$

When $e = 255$ and $m \neq 0$, a *not-a-number* (NaN) is represented. NaNs can be caused by dividing a zero by a zero or taking the square root of a negative number, among many other nondeterminate cases. When $e = 255$ and $m = 0$, an infinite number $X = (-1)^s \infty$ is represented. Note that $+\infty$ and $-\infty$ are represented differently.

When $e = 0$ and $m \neq 0$, the number represented is $X = (-1)^s 2^{-126}(0.m)$. When $e = 0$ and $m = 0$, a zero is represented as $X = (-1)^s 0$. Again, $+0$ and -0 are possible.

The 64-bit (double-precision) floating point can be defined similarly using an excess-l023 code in the exponent field and a 52-bit mantissa field. A number which is nonzero, finite, non-NaN, and normalized, has the following value:

$$X = (-1)^s \times 2^{e-1023} \times (1.m) \tag{6.16}$$

Floating-Point Operations The four primitive arithmetic operations are defined below for a pair of floating-point numbers represented by $X = (m_x, e_x)$ and $Y = (m_y, e_y)$. For clarity, we assume $e_x \leq e_y$ and base $r = 2$.

$$X + Y = (m_x \times 2^{e_x - e_y} + m_y) \times x^{e_y} \tag{6.17}$$

$$X - Y = (m_x \times 2^{e_x - e_y} - m_y) \times x^{e_y} \tag{6.18}$$

$$X \times Y = (m_x \times m_y) \times 2^{e_x + e_y} \tag{6.19}$$

$$X \div Y = (m_x \div m_y) \times 2^{e_x - e_y} \tag{6.20}$$

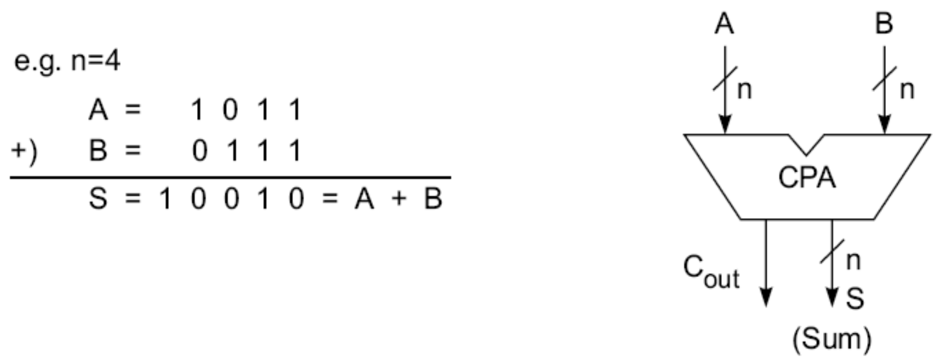
The above equations clearly identify the number of arithmetic operations involved in each floating-point function. These operations can be divided into two halves: One half is for exponent operations such as comparing their relative magnitudes or adding/subtracting them; the other half is for mantissa operations, including four types of fixed-point operations.

Arithmetic Pipeline Stages Depending on the function to be implemented, different pipeline stages in an arithmetic unit require different hardware logic. Since all arithmetic operations (such as *add, subtract, multiply, divide, squaring, square rooting, logarithm,* etc.) can be implemented with the basic add and shifting operations, the core arithmetic stages require some form of hardware to add and to shift.

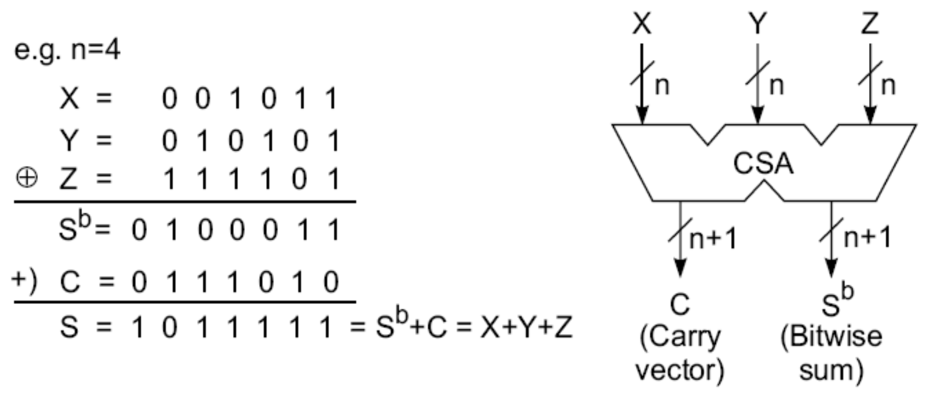
For example, a typical three-stage floating-point adder includes a first stage for exponent comparison and equalization which is implemented with an integer adder and some shifting logic; a second stage for fraction addition using a high-speed carry lookahead adder; and a third stage for fraction normalization and exponent readjustment using a shifter and another addition logic.

Arithmetic or logical shifts can be easily implemented with *shift registers.* High-speed addition requires either the use of a *carry-propagation adder* (CPA) which adds two numbers and produces an arithmetic sum as shown in Fig. 6.22a, or the use of a *carry-save adder* (CSA) to "add" three input numbers and produce one sum output and a carry output as exemplified in Fig. 6.22b.

In a CPA, the carries generated in successive digits are allowed to propagate from the low end to the high end, using either ripple carry propagation or some carry looka-head technique.

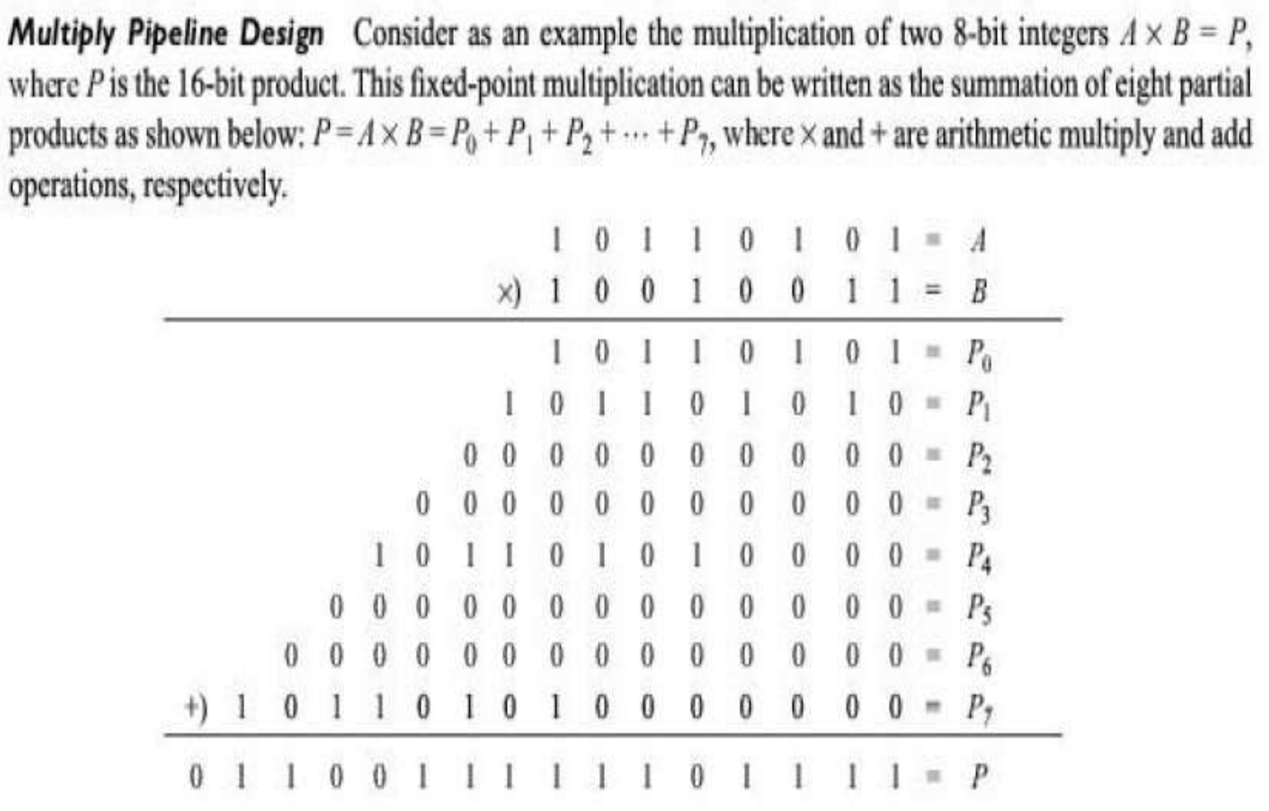


(a) An *n*-bit carry-propagate adder (CPA) which allows either carry propagation or applies the carry-lookahead technique



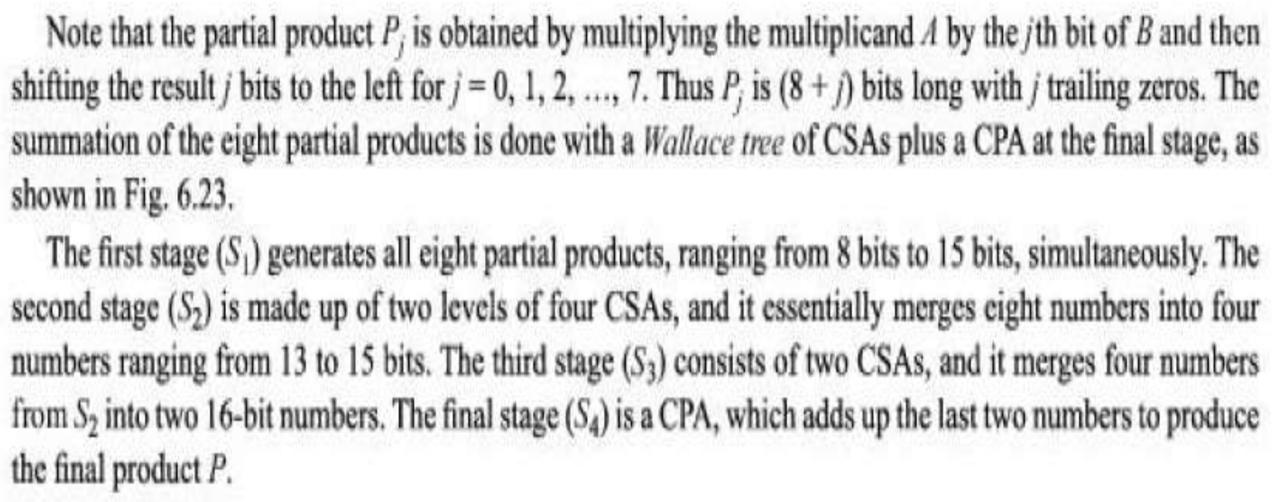
(b) An *n*-bit carry-save adder (CSA), where S^b is the bitwise sum of X, Y, and Z, and C is a carry vector generated without carry propagation between digits

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



Multiply Pipeline Design Consider as an example the multiplication of two 8-bit integers $A \times B = P$, where P is the 16-bit product. This fixed-point multiplication can be written as the summation of eight partial products as shown below: $P = A \times B = P_0 + P_1 + P_2 + \cdots + P_7$, where $\times$ and $+$ are arithmetic multiply and add operations, respectively.

```
                              1 0 1 1 0 1 0 1 = A
                          ×)  1 0 0 1 0 0 1 1 = B
-------------------------------------------------
                              1 0 1 1 0 1 0 1 = P0
                            1 0 1 1 0 1 0 1 0 = P1
                          0 0 0 0 0 0 0 0 0 0 = P2
                        0 0 0 0 0 0 0 0 0 0 0 = P3
                      1 0 1 1 0 1 0 1 0 0 0 0 = P4
                    0 0 0 0 0 0 0 0 0 0 0 0 0 = P5
                  0 0 0 0 0 0 0 0 0 0 0 0 0 0 = P6
            +)  1 0 1 1 0 1 0 1 0 0 0 0 0 0 0 = P7
-------------------------------------------------
              0 1 1 0 0 1 1 1 1 1 1 0 1 1 1 1 = P
```



Note that the partial product P_j is obtained by multiplying the multiplicand A by the jth bit of B and then shifting the result j bits to the left for $j = 0, 1, 2, \ldots, 7$. Thus P_j is $(8 + j)$ bits long with j trailing zeros. The summation of the eight partial products is done with a *Wallace tree* of CSAs plus a CPA at the final stage, as shown in Fig. 6.23.

The first stage (S_1) generates all eight partial products, ranging from 8 bits to 15 bits, simultaneously. The second stage (S_2) is made up of two levels of four CSAs, and it essentially merges eight numbers into four numbers ranging from 13 to 15 bits. The third stage (S_3) consists of two CSAs, and it merges four numbers from S_2 into two 16-bit numbers. The final stage (S_4) is a CPA, which adds up the last two numbers to produce the final product P.

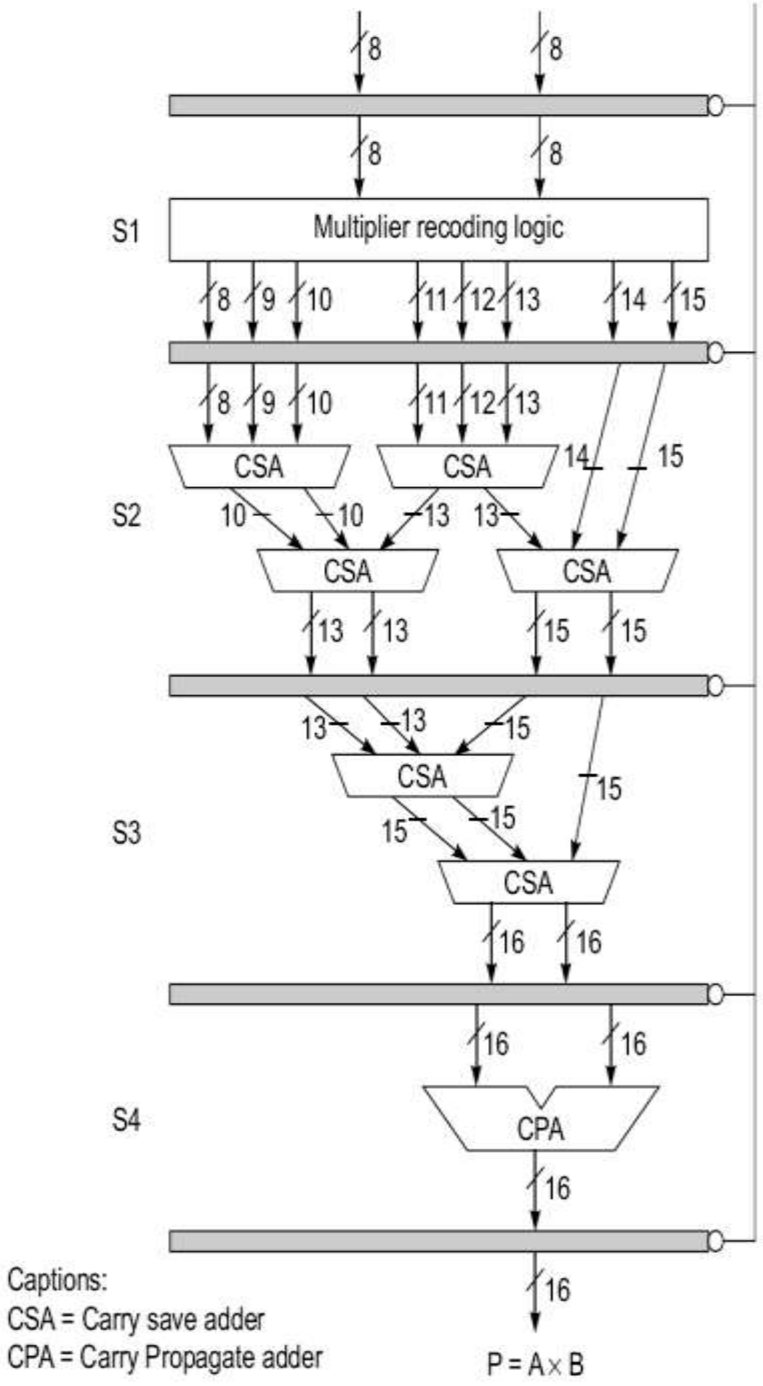


Fig. 6.23 A pipeline unit for fixed-point multiplication of 8-bit integers (The number along each line indicates the line width.)