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

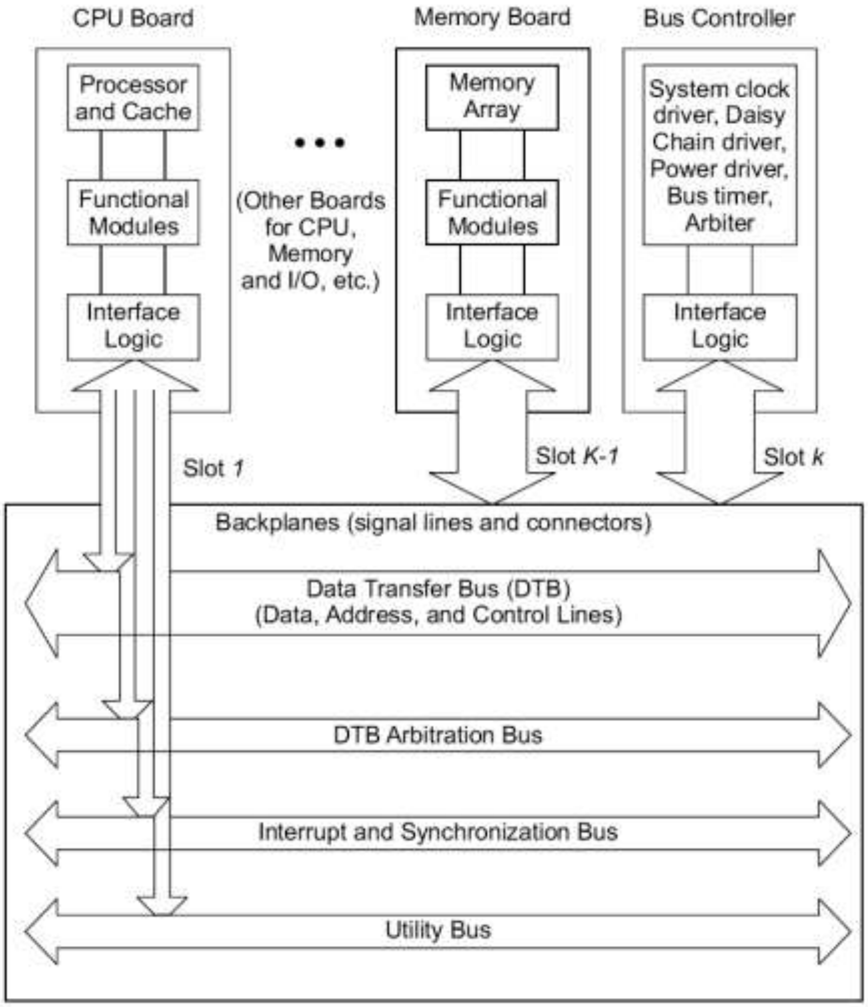


Fig. 5.1 Backplane buses, system interfaces, and slot connections to various functional boards in a multiprocessor system

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

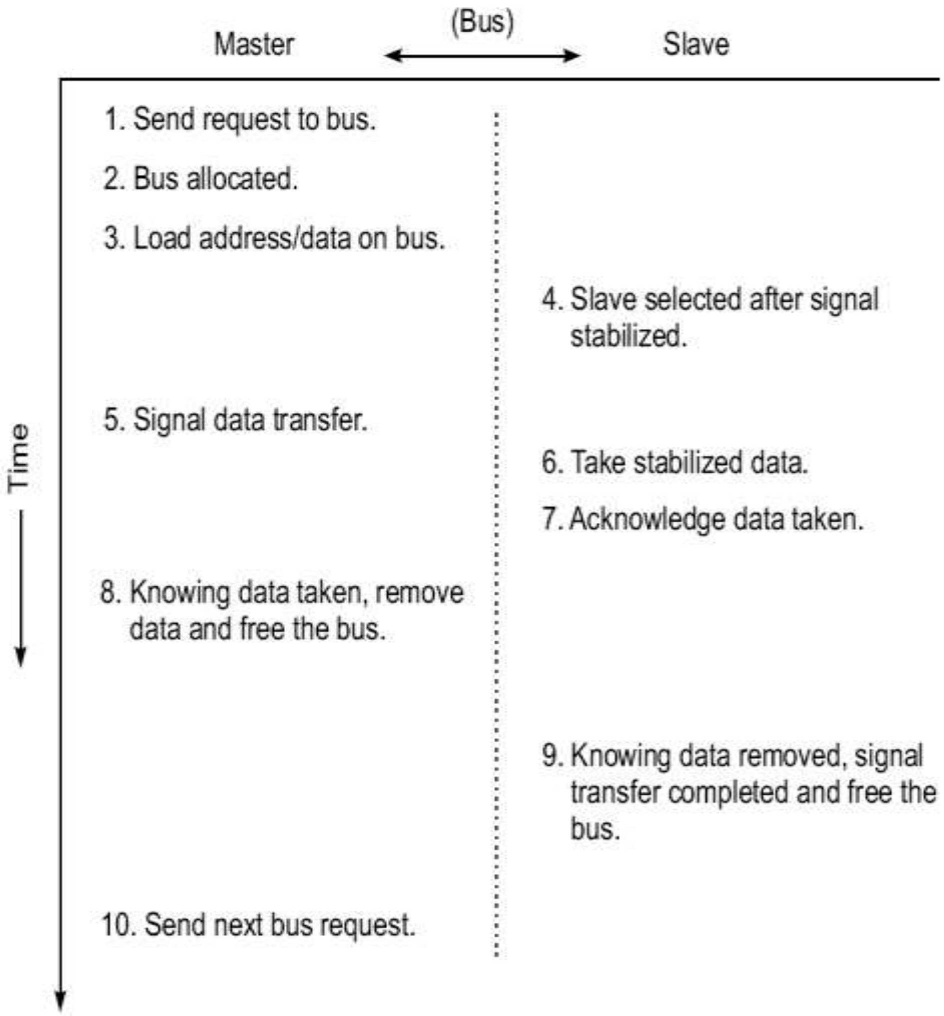

Master
(Bus)
Slave
Time
1. Send request to bus.
2. Bus allocated.
3. Load address/data on bus.
4. Slave selected after signal stabilized.
5. Signal data transfer.
6. Take stabilized data.
7. Acknowledge data taken.
8. Knowing data taken, remove data and free the bus.
9. Knowing data removed, signal transfer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

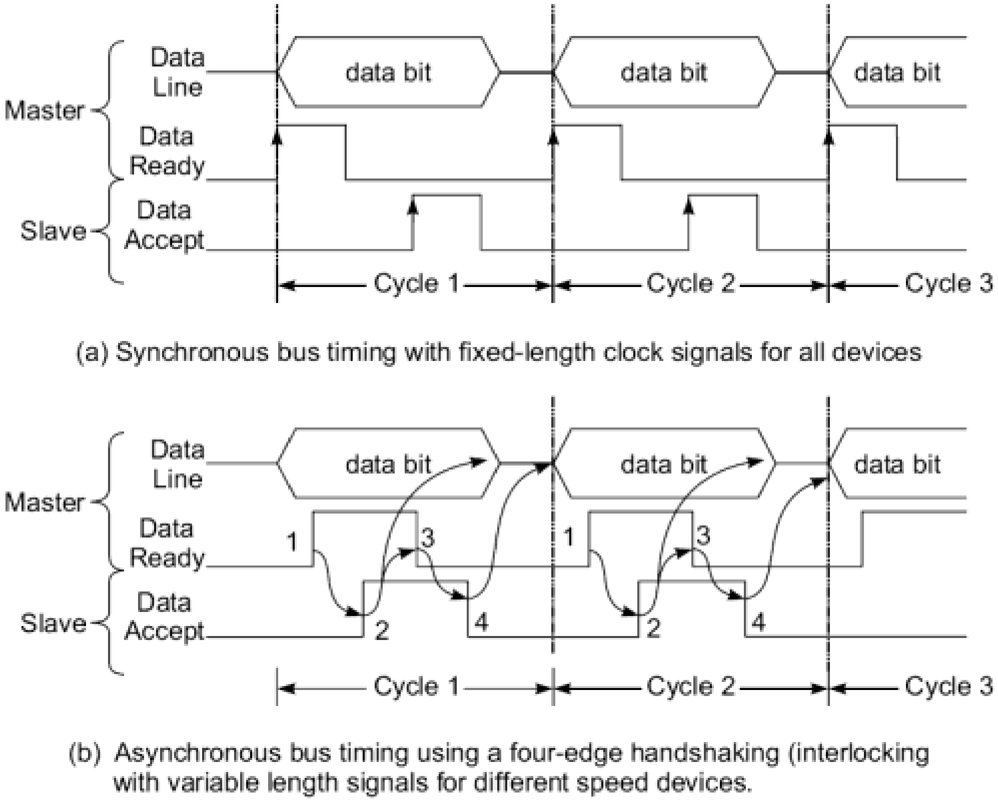
Fig. 5.2 Typical time sequence for information transfer between a master and a slave over a system bus

- 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

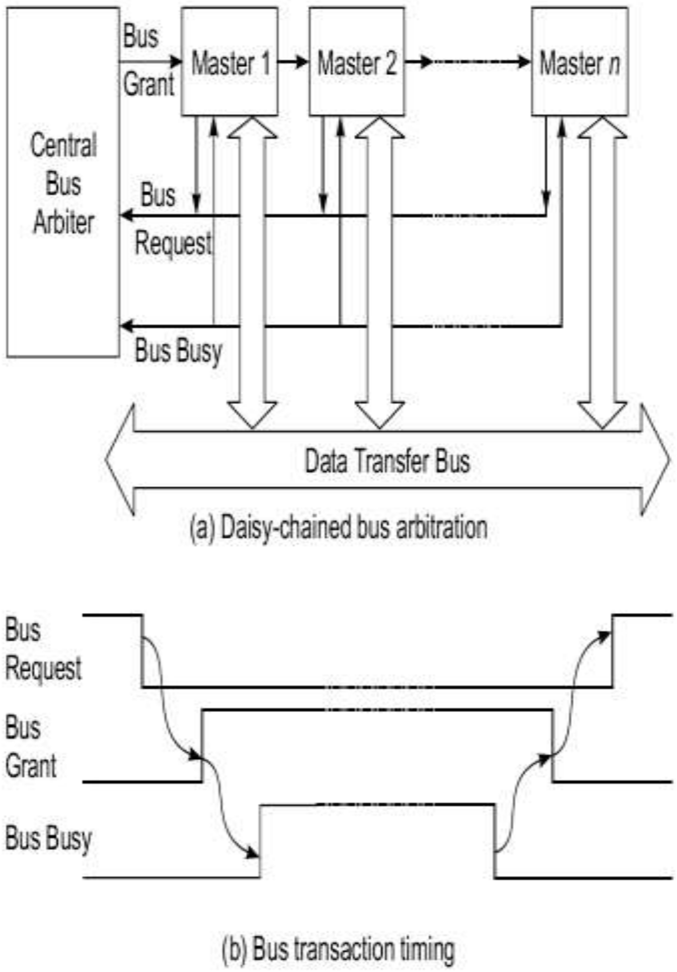
5.1.3 Arbitration, Transaction and Interrupt

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.



(a) Daisy-chained bus arbitration

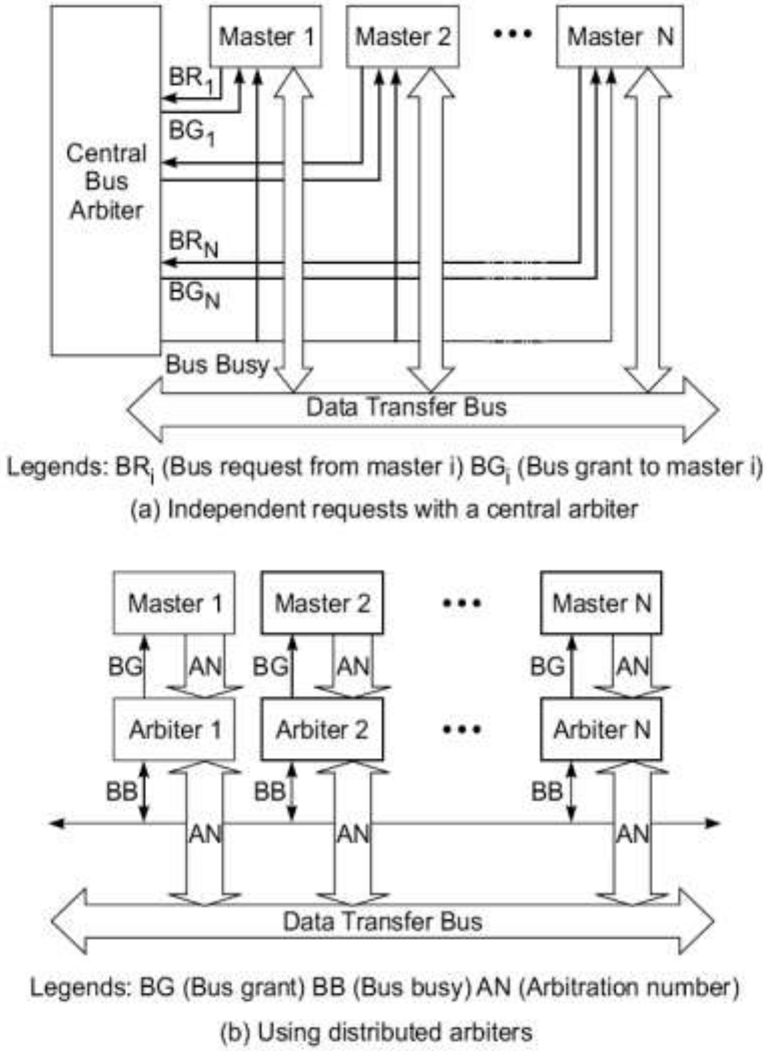
(b) Bus transaction timing

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



Legends: BR_i (Bus request from master i) BG_i (Bus grant to master i)

(a) Independent requests with a central arbiter

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

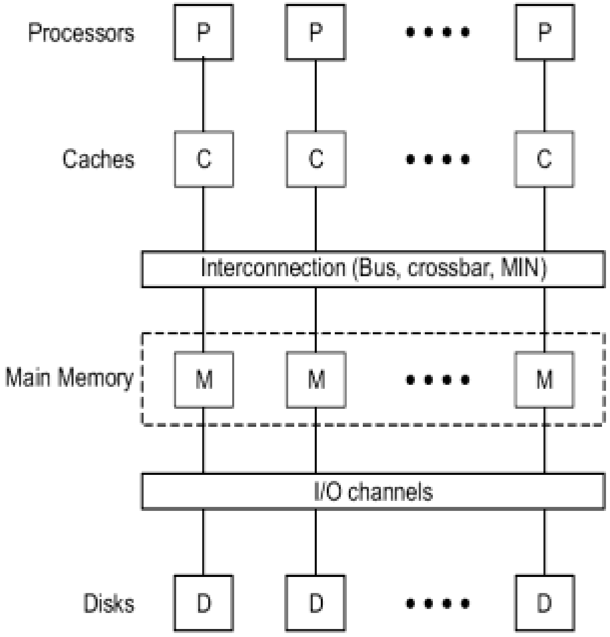


Fig. 5.6 A memory hierarchy for a shared-memory multiprocessor

Physical address cache

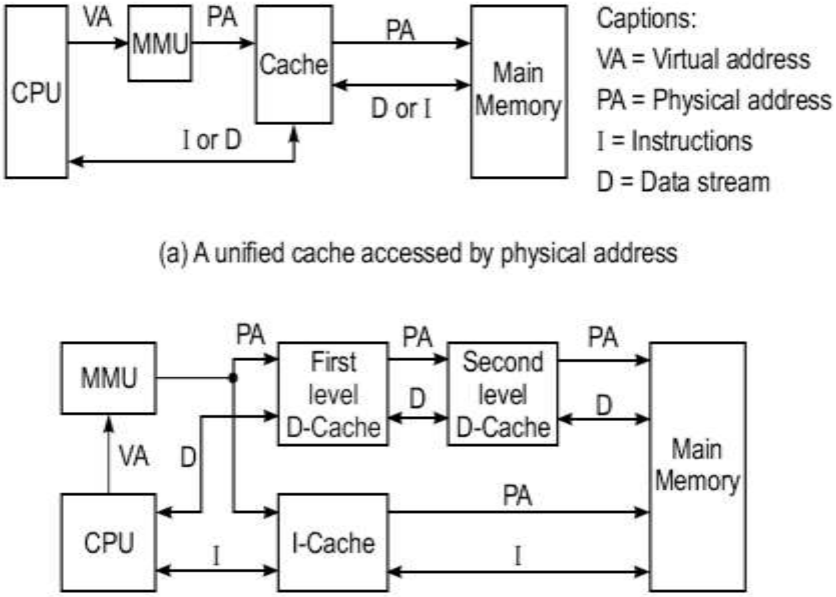
- When cache is addressed by physical address it is called physical address cache. The cache is indexed and tagged with physical address.
- Cache lookup must occur after address translation in TLB or MMU. No aliasing is allowed so that the address is always uniquely translated without confusion.
- After 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

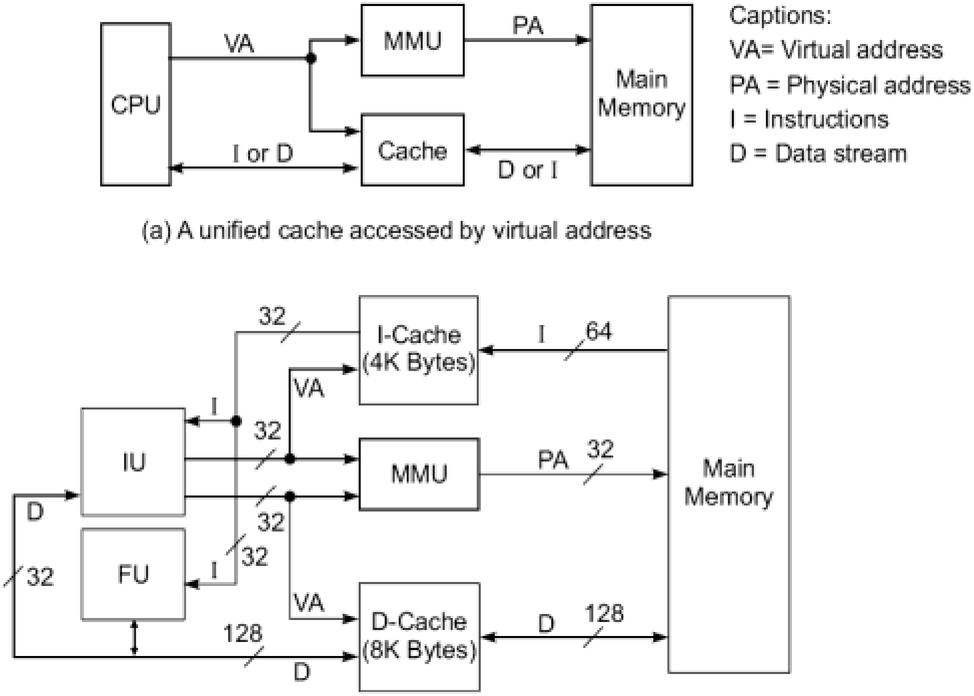


(a) A unified cache accessed by physical address

(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

ïg. 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**
- 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 $\underline{B}_i$

  $B_j \rightarrow \underline{B}_i$ if i=j mod m

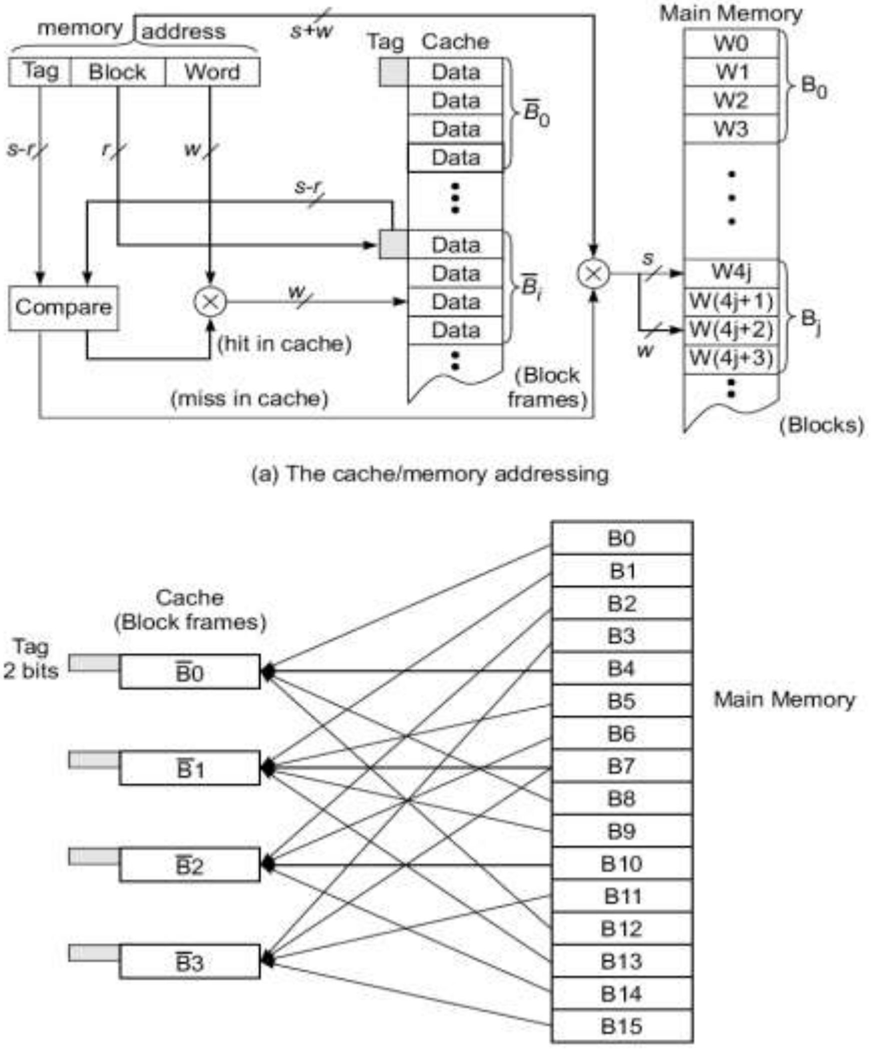
- There is a unique block frame $\underline{B}_i$ that each B_j can load into.
- There is no way to implement a block replacement policy.
- This Direct mapping is very rigid but is the simplest cache organization to implement.

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 $\overline{B}_i$ if $i = j$ (modulo 4)

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

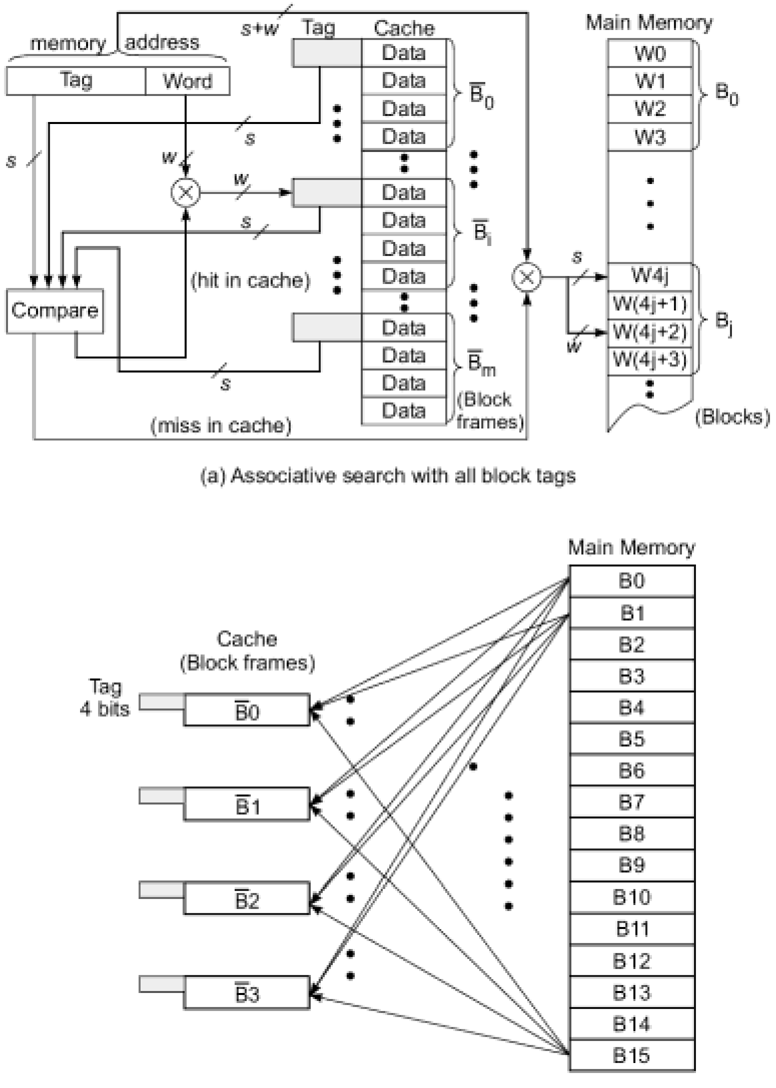
The block field (r bits) is used to implement the (modulo-m) placement, where **$m=2^r$**

Once the block **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

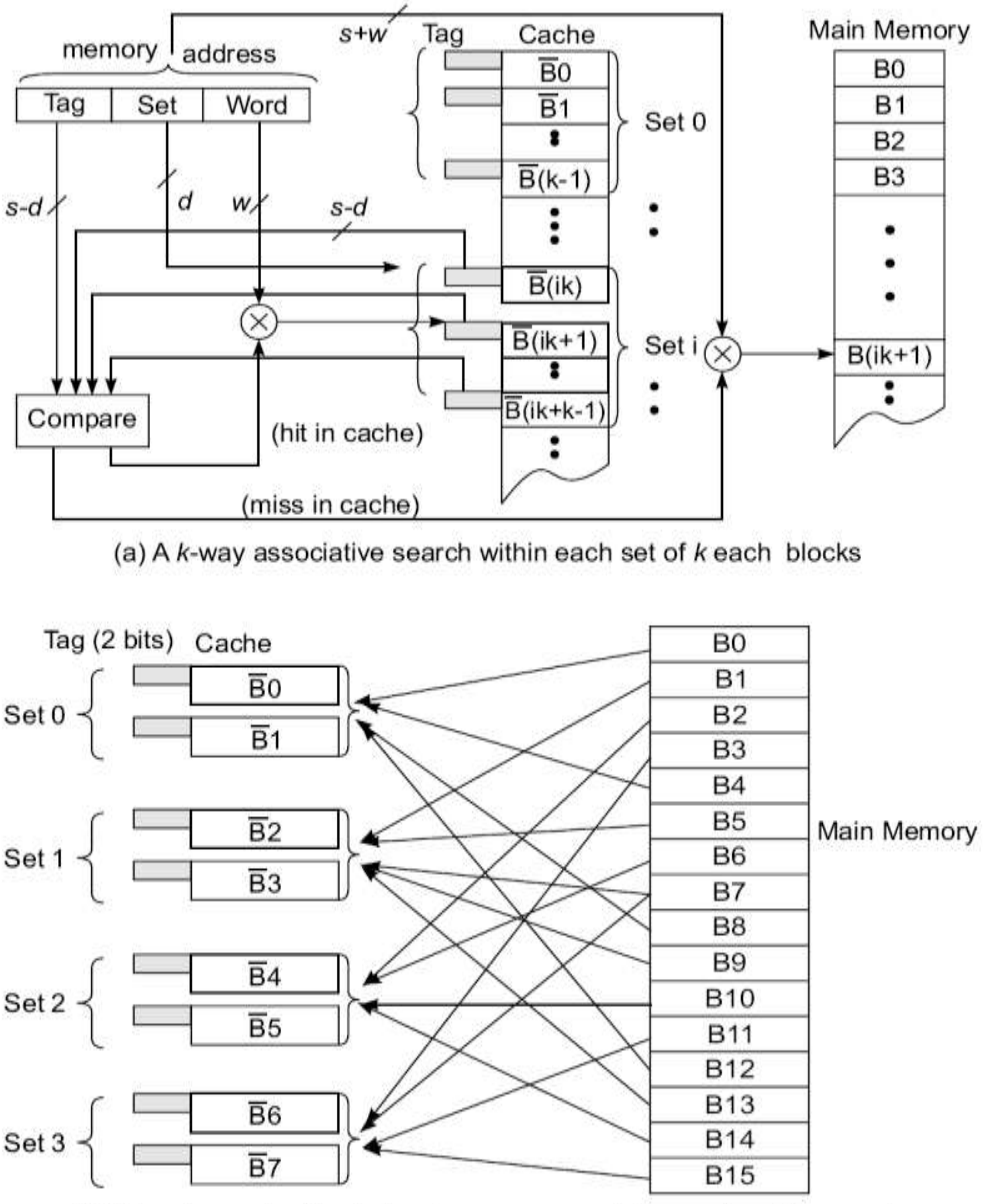
Fig. 5.10 Fully associative cache organization and a mapping example

- **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.

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

- 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.

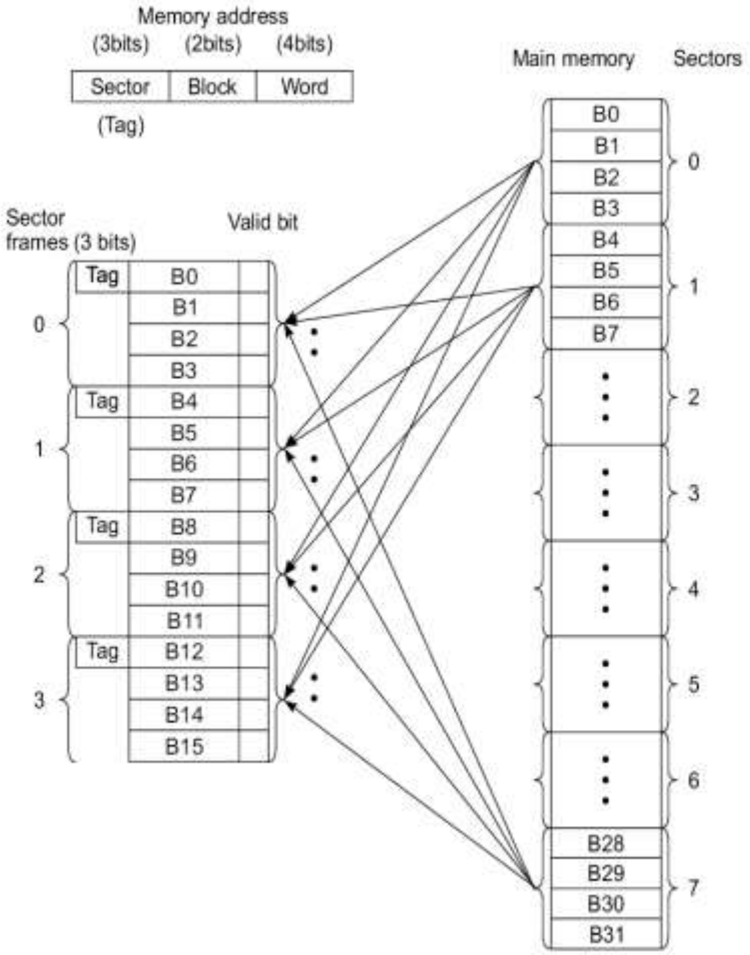


Fig. 5.12 A four-way sector mapping cache organization

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

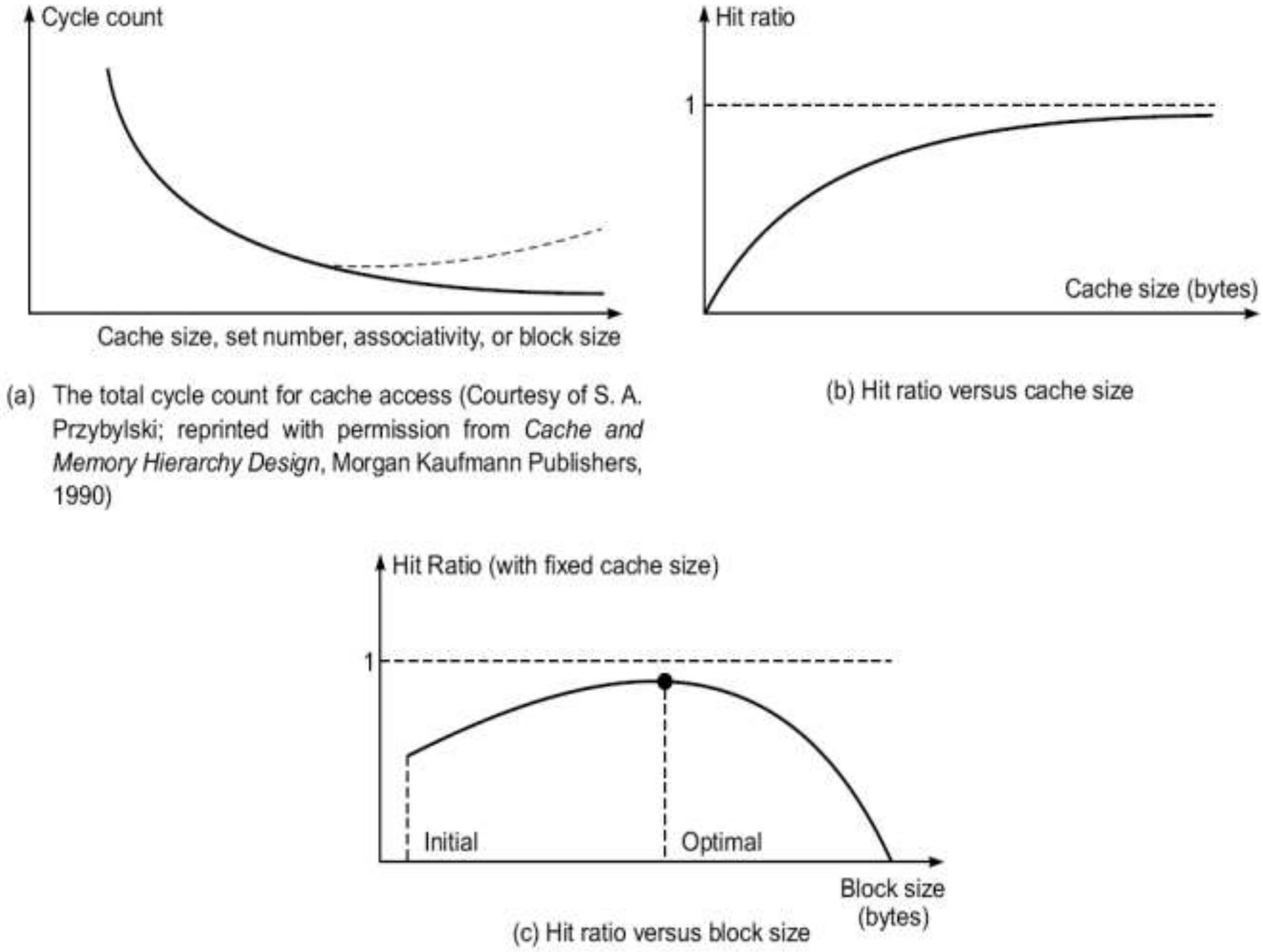
- This refers to the number of basic machine cycles needed for cache access, update and coherence control.
- Cache speed is affected by underlying static or dynamic RAM technology, the cache organization and the cache hit ratios.
- The write through or write back policy also affect the cycle count.
- 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.

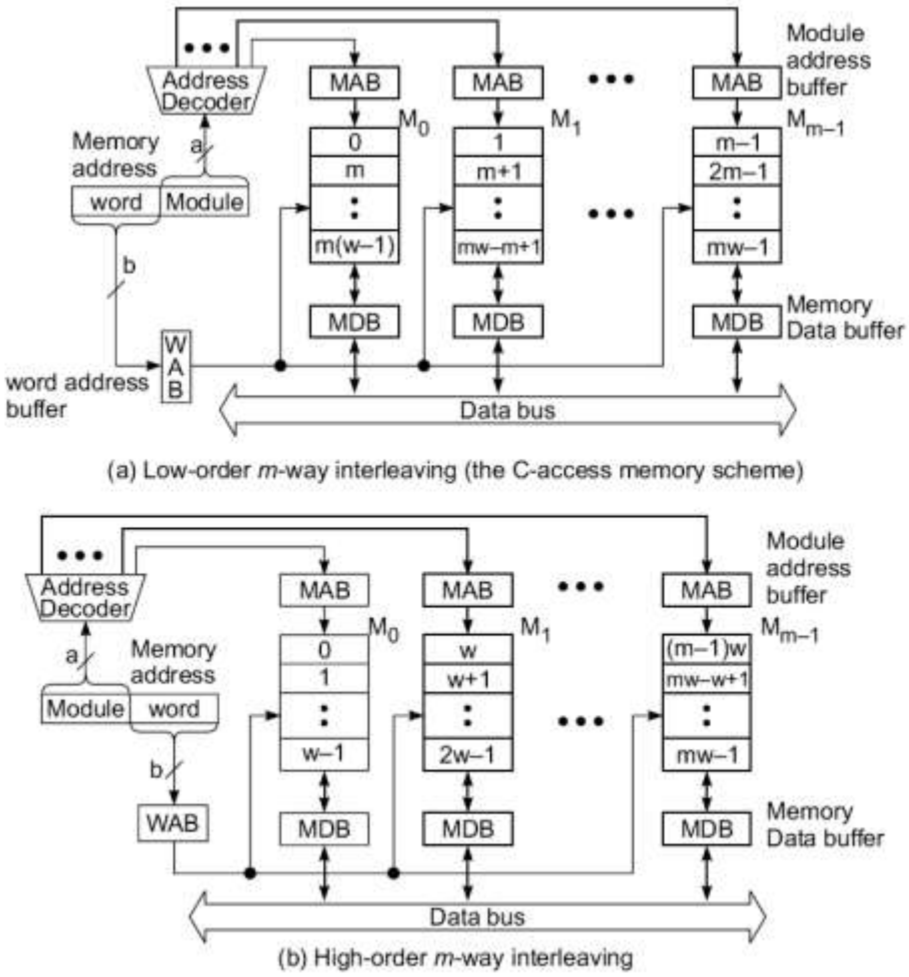
Consider a main memory formed with $\mathbf{m = 2^a}$ memory modules, each containing $\mathbf{w = 2^b}$ words of memory cells. The total memory capacity is $\mathbf{m.w = 2^{a+b}}$ words.

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



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

(b) High-order *m*-way 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)

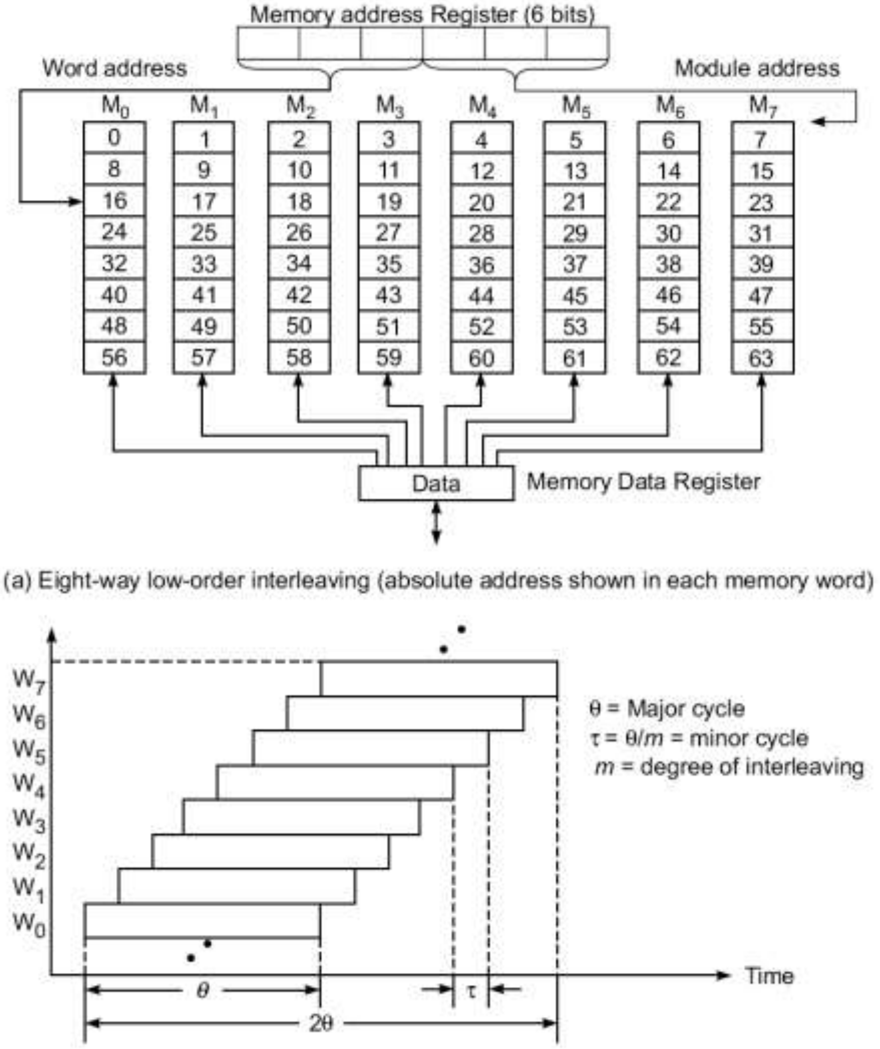
Low-order interleaving

- 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



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

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

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

When $\mathbf{n} \rightarrow \infty$ (very long vector), $\mathbf{t_1} \rightarrow \boldsymbol{\theta}/\mathbf{m} = \boldsymbol{\tau}$.

As $\mathbf{n} \rightarrow \mathbf{1}$ (scalar access), $\mathbf{t_1} \rightarrow \boldsymbol{\theta}$.

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.

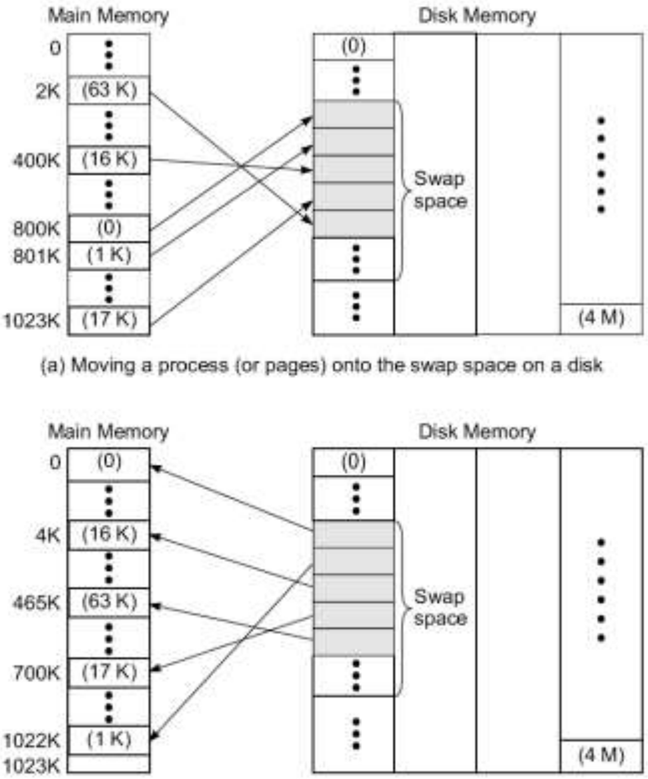
Allocation Policies

- **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



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

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

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.

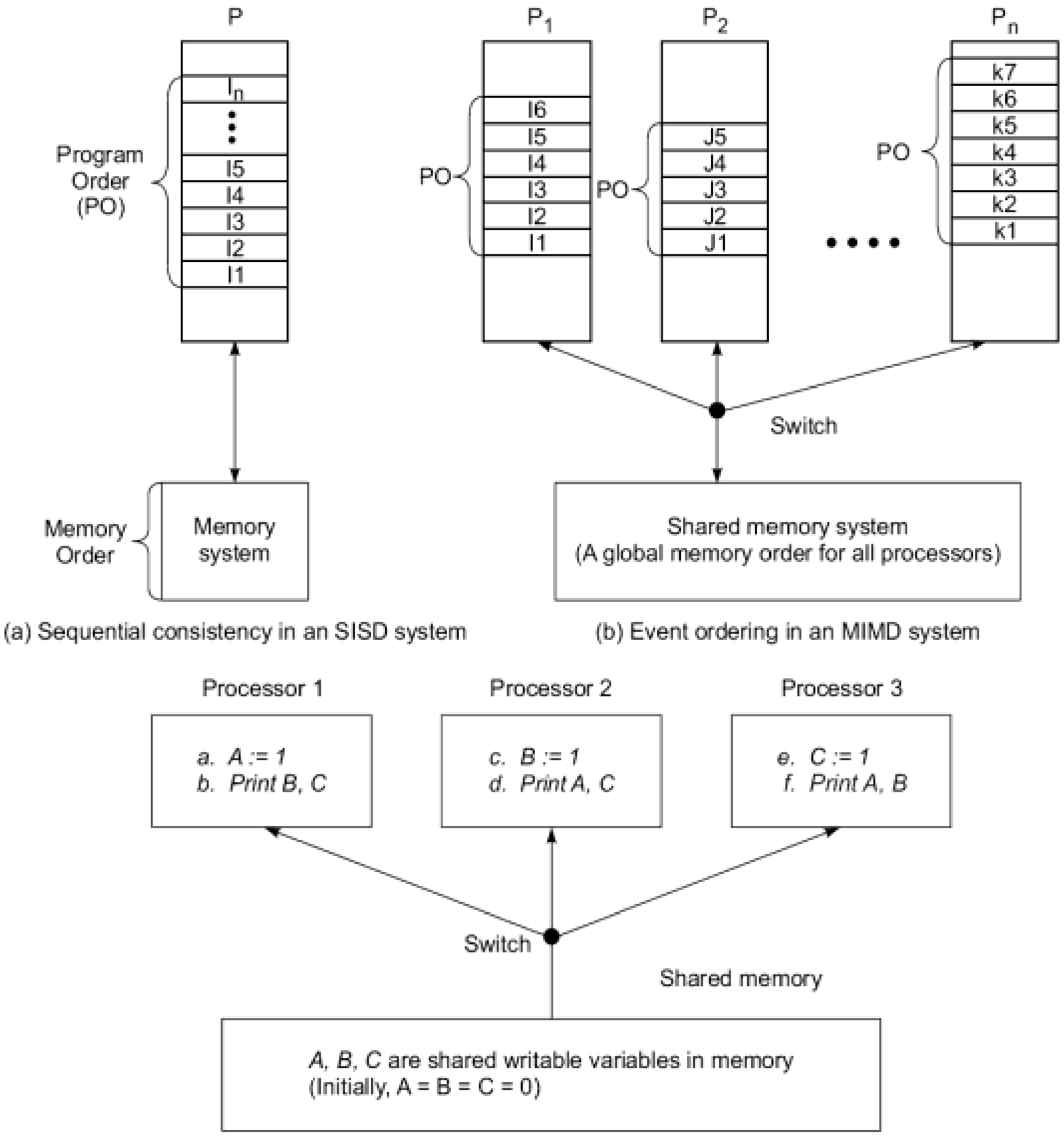
Three primitive memory operations for the purpose of specifying memory consistency models are defined:

(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.



(a) Sequential consistency in an SISD system

(b) Event ordering in an MIMD system

(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

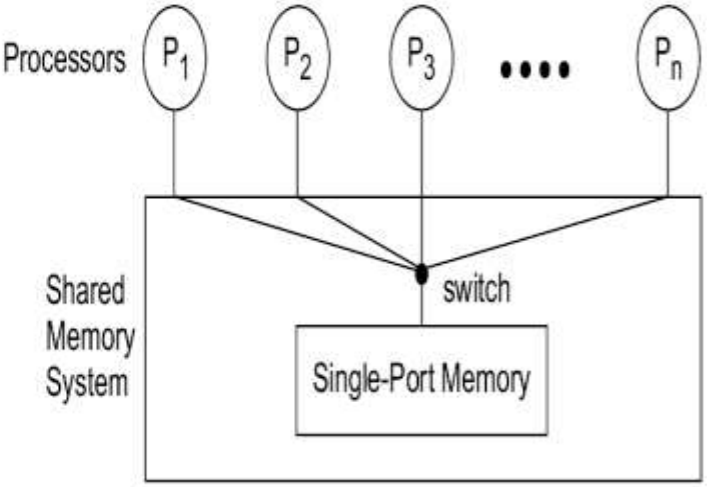


Fig. 5.20 Sequential consistency memory model (Courtesy of Sindhu, Frailong, and Cekleov; reprinted with permission from *Scalable Shared-Memory Multiprocessors*, Kluwer Academic Publishers, 1992)

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

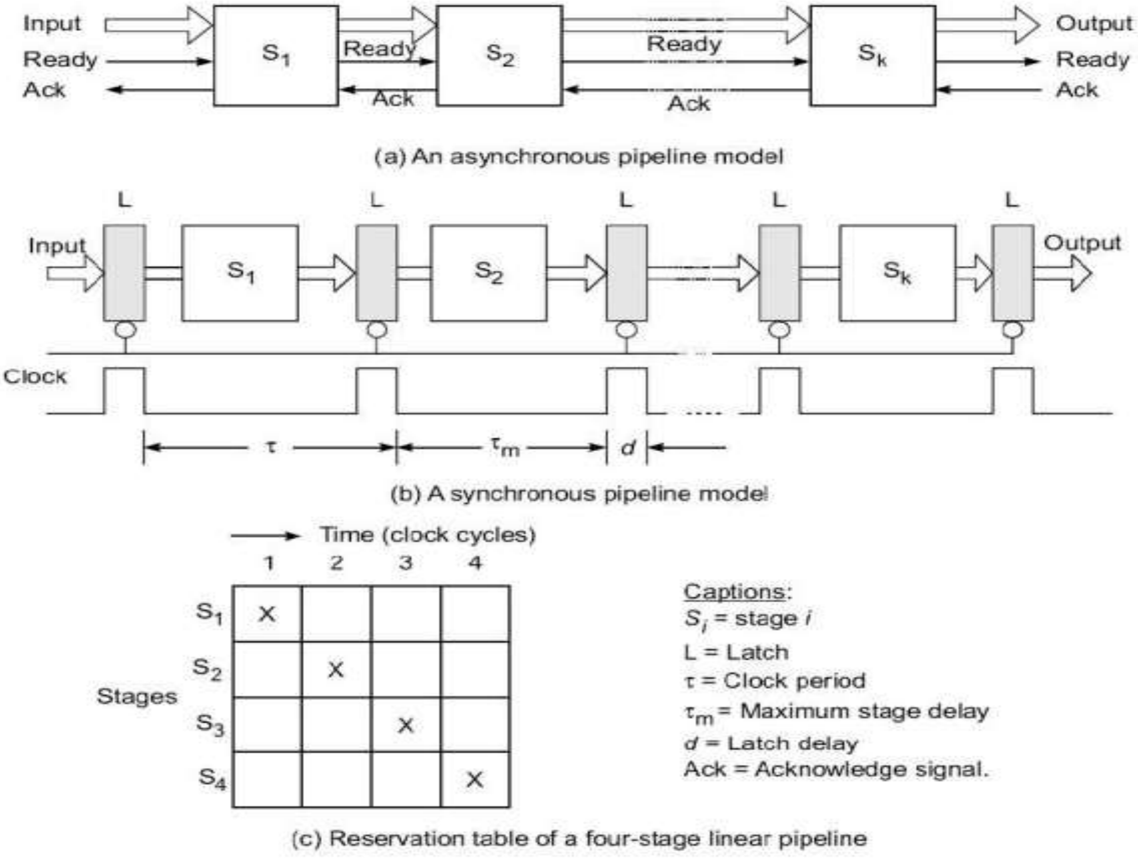
6.1 Linear Pipeline Processors

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 $\mathbf{S_1}$.
- The processed results are passed from stage $\mathbf{S_i}$ to stage $\mathbf{S_{i+1}}$, for all i=1,2,….,k-1. The final result emerges from the pipeline at the last stage $\mathbf{S_n}$.
- Depending on the control of data flow along the pipeline, we model linear pipelines in two categories: **Asynchronous** and **Synchronous**.



(a) An asynchronous pipeline model

(b) A synchronous pipeline model

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

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

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, τ_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

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

- where **τ** is the clock period.
- Consider an equivalent-function nonpipelined processor which has a flow-through delay of ***kτ***. 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** → ∞.

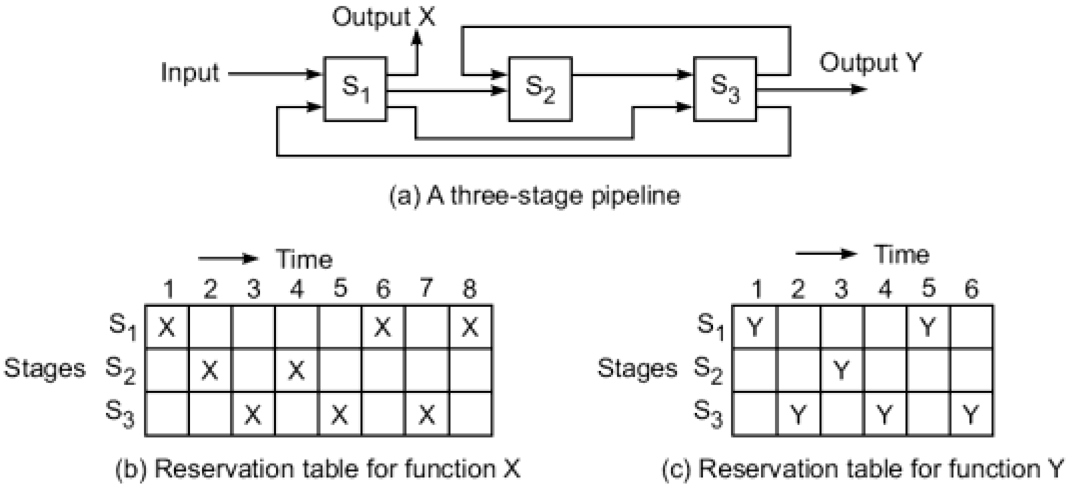
6.2 Non Linear Pipeline Processors

- 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.

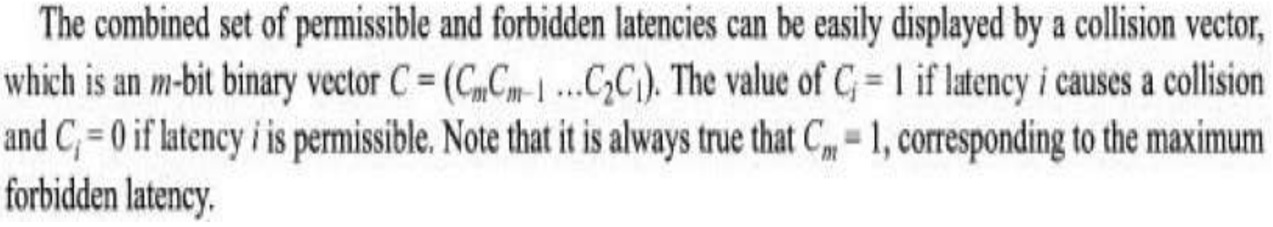
- A dynamic pipeline may be specified by more than one reservation table. Each reservation table displays the time-space flow of data through the pipeline for one function evaluation.
- Different functions may follow different paths on the reservation table.
- A number of pipeline configurations may 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

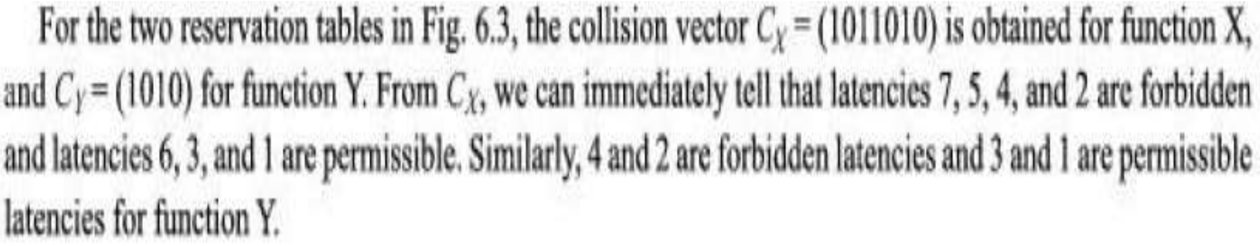
- The number of time units (clock cycles) between two initiations of a pipeline is the latency between them.
- Latency values must be non negative integers. A latency of k means that two initiations are separated by k clock cycles.
- Any attempt by two or more initiations to use the same pipeline stage 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

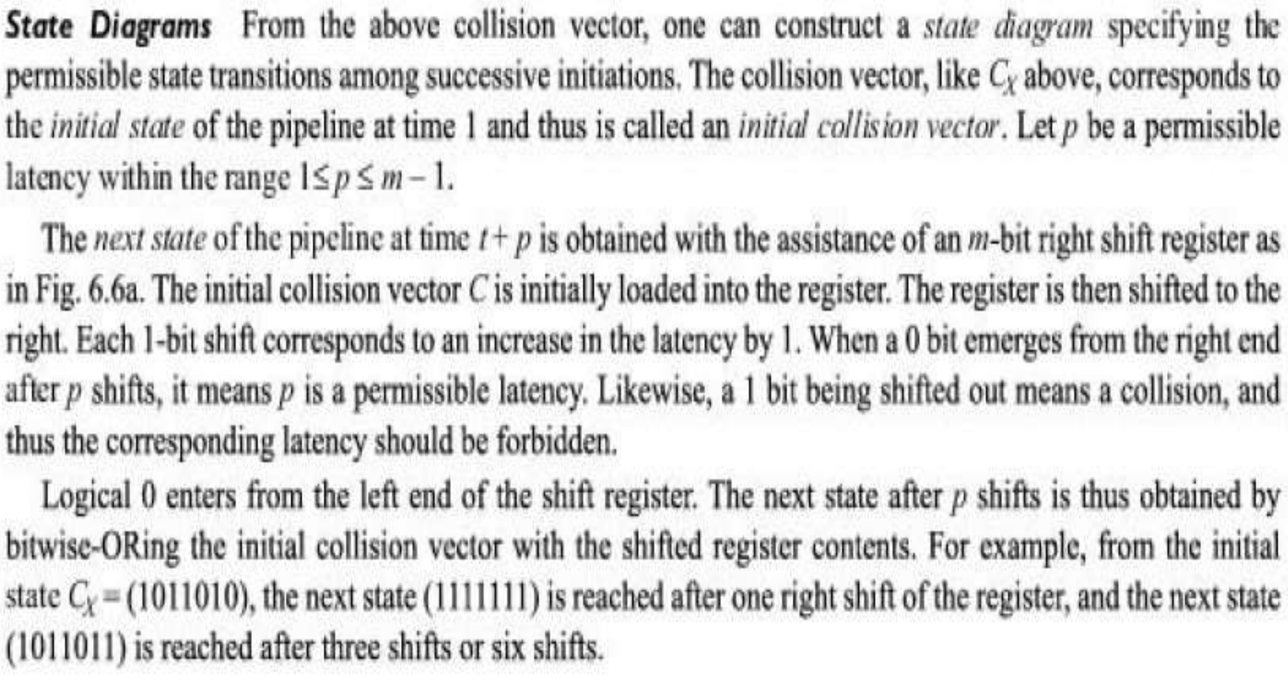
- When scheduling events in a nonlinear pipeline, the main objective is to obtain the shortest average latency between initiations without causing collisions.
- **Collision Vector**: By examining the reservation table, one can distinguish the set of permissible latencies from the set of forbidden latencies.
- 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 $C = (C_m C_{m-1} \ldots C_2 C_1)$. 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.



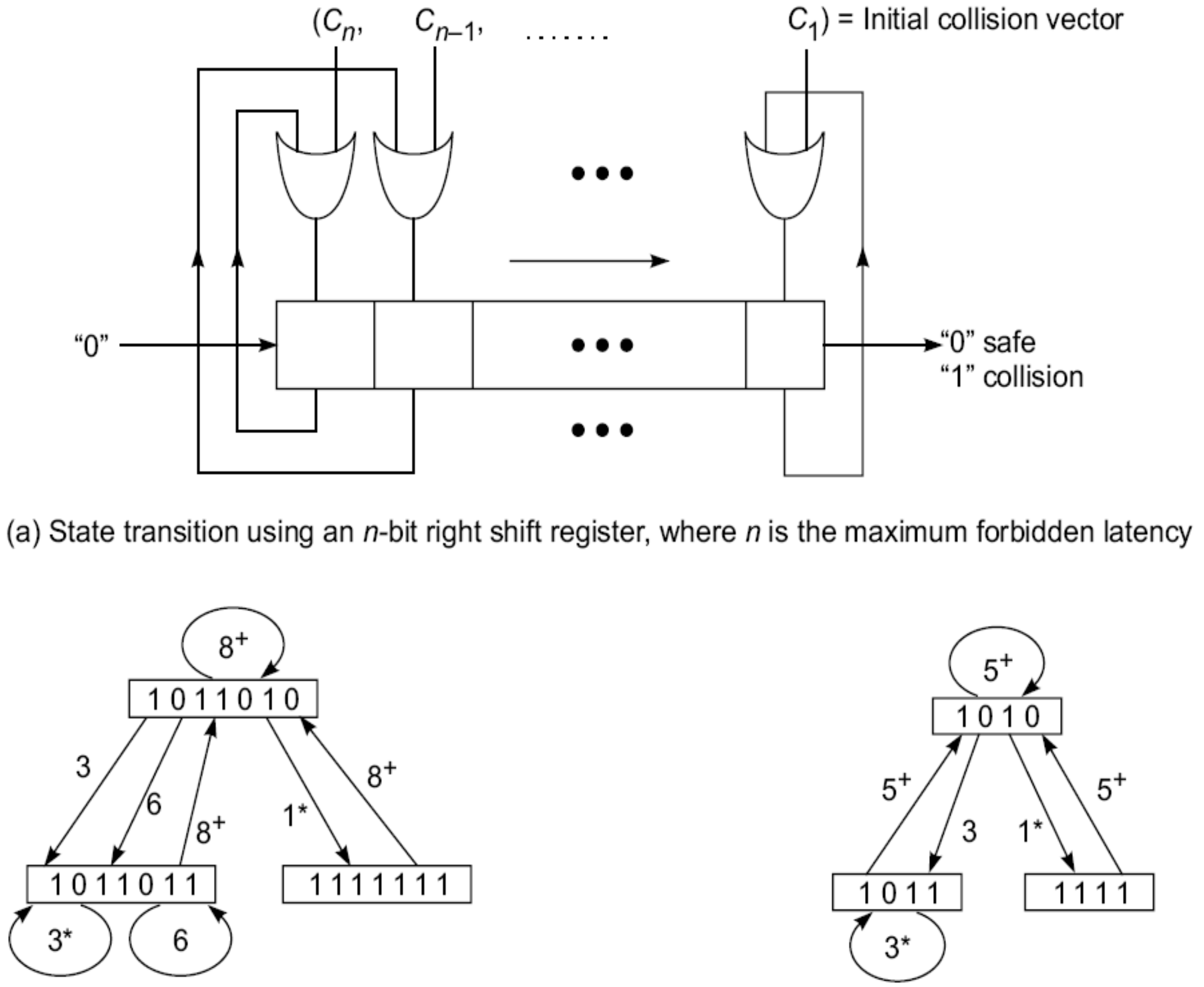
For the two reservation tables in Fig. 6.3, the collision vector $C_X = (1011010)$ is obtained for function X, and $C_Y = (1010)$ for function Y. From C_X, we can immediately tell that latencies 7, 5, 4, and 2 are forbidden and latencies 6, 3, and 1 are permissible. Similarly, 4 and 2 are forbidden latencies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



(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.

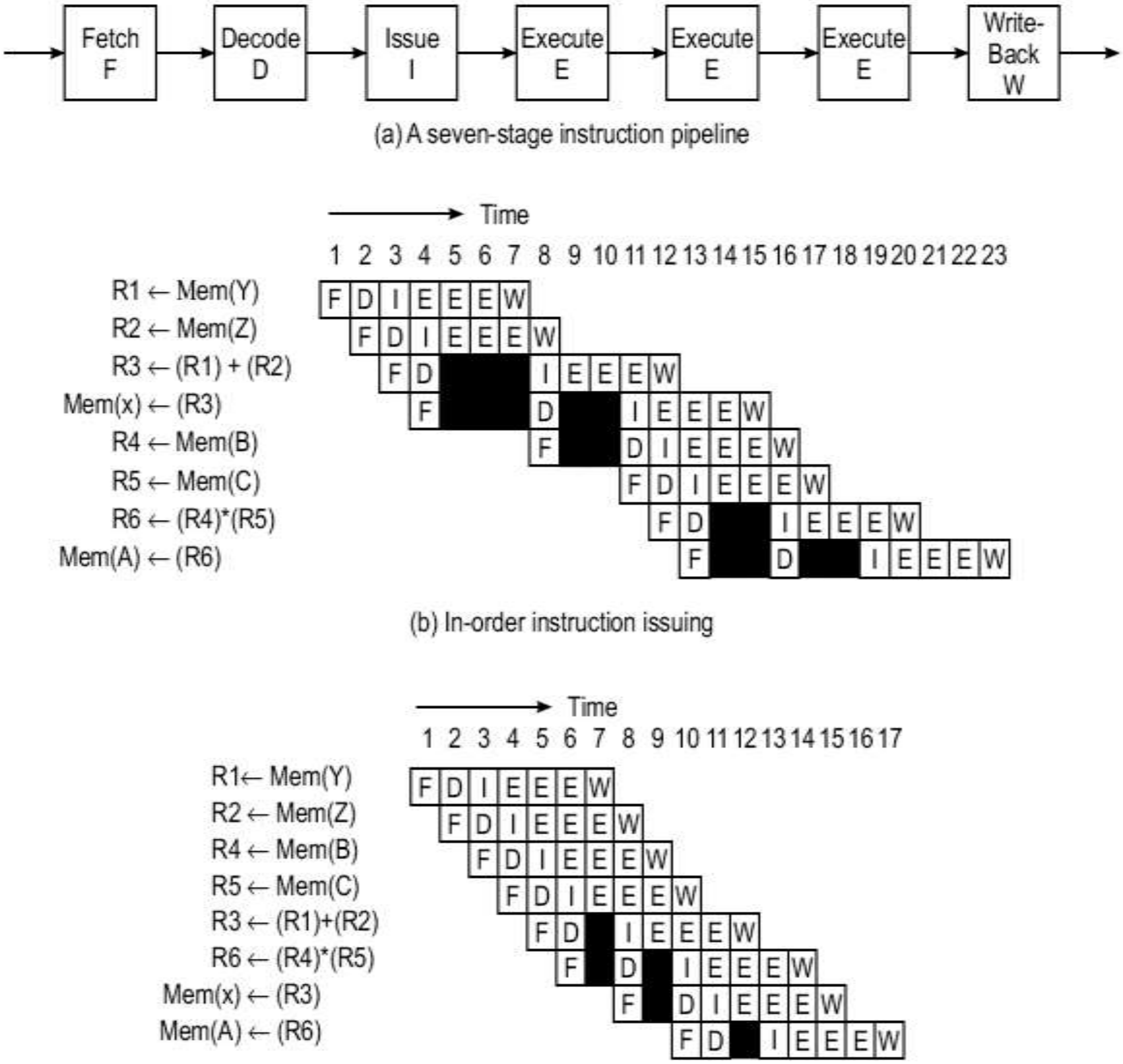
(3) The average latency of any greedy cycle is upper-bounded by the number of 1's in the initial collision vector plus 1. This is also an upper bound on the MAL.

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.



(a) A seven-stage instruction pipeline

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

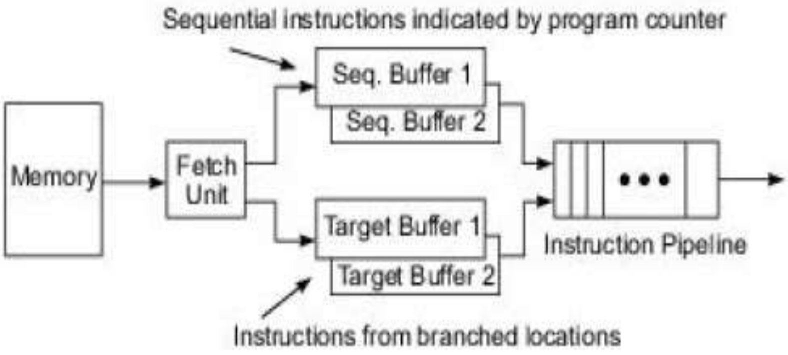
Fig. 6.9 Pipelined execution of X = Y + Z and A = B × C (Courtesy of James Smith; reprinted with permission from *IEEE Computer*, July 1989)

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).

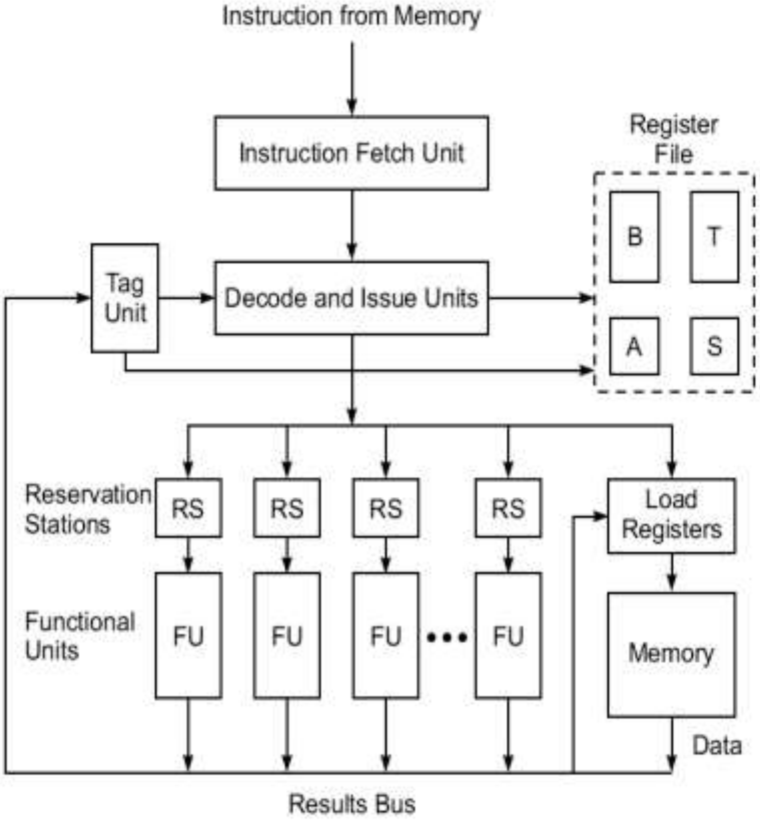


Fig. 6.12 A pipelined processor with multiple functional units and distributed reservation stations supported by tagging (Courtesy of G. Sohi; reprinted with permission from *IEEE Transactions on Computers*, March 1990)

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).**

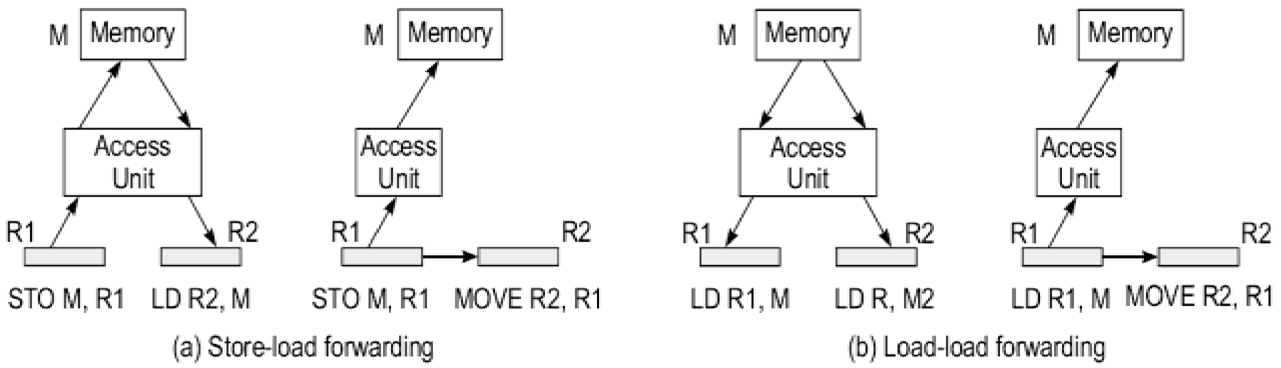


Fig. 6.13 Internal data forwarding by replacing memory-access operations with register transfer operations

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.

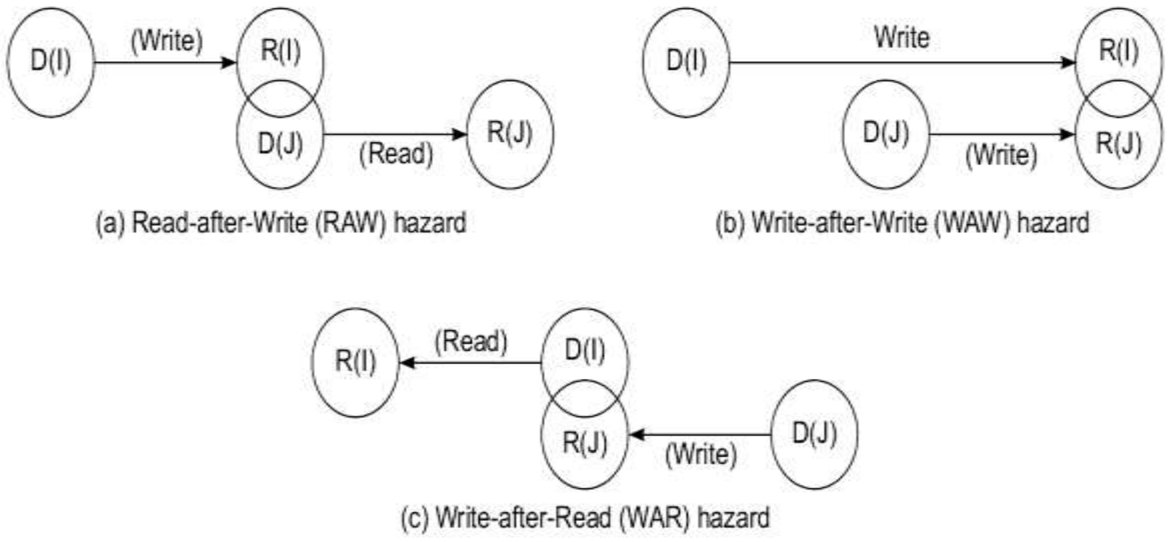

D(I)
(Write)
R(I)
D(J)
(Read)
R(J)
(a) Read-after-Write (RAW) hazard
D(I)
Write
R(I)
D(J)
(Write)
R(J)
(b) Write-after-Write (WAW) hazard
R(I)
(Read)
D(I)
R(J)
(Write)
D(J)
(c) Write-after-Read (WAR) hazard

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

- 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

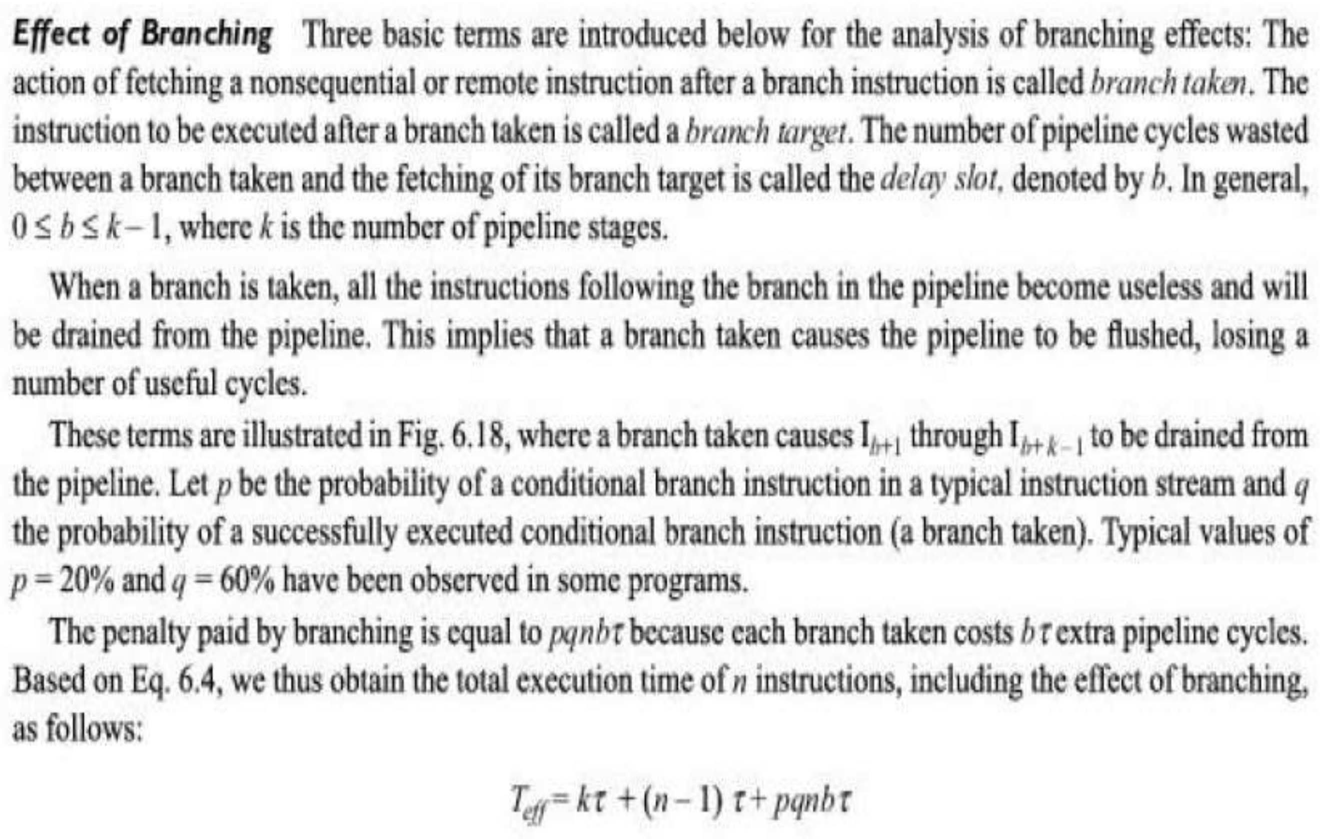
Listed below are conditions under which possible hazards can occur:

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

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

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

6.3.4 Branch Handling Techniques



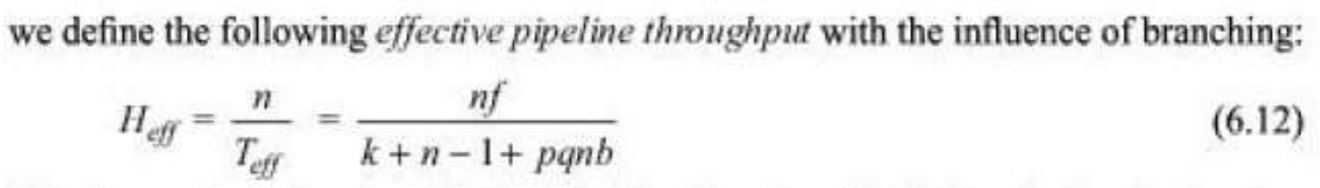
Effect of Branching Three basic terms are introduced below for the analysis of branching effects: The action of fetching a nonsequential or remote instruction after a branch instruction is called *branch taken*. The instruction to be executed after a branch taken is called a *branch target*. The number 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.

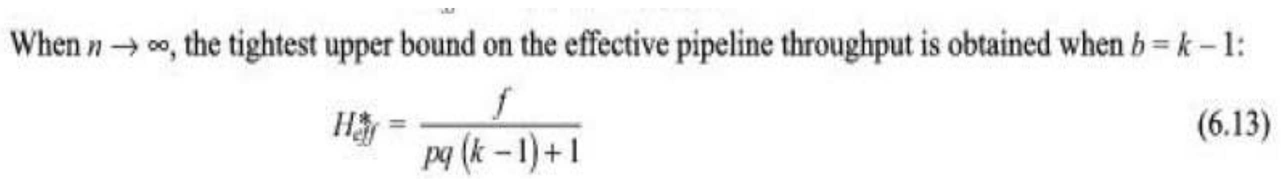
The penalty paid by branching is equal to $pqnb\tau$ because each branch taken costs $b\tau$ extra pipeline cycles. Based on Eq. 6.4, we thus obtain the total execution time of n instructions, including the effect of branching, as follows:

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



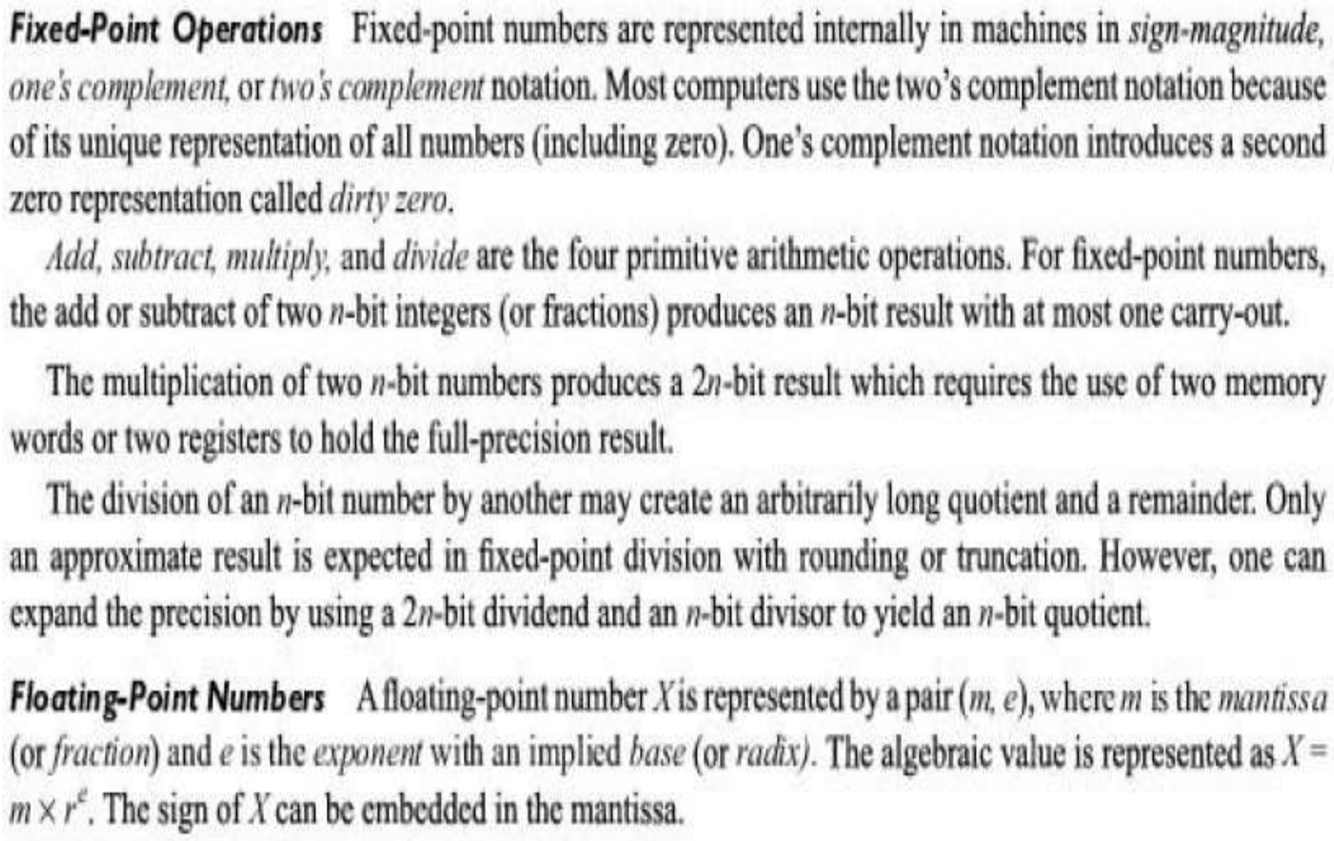
we define the following *effective pipeline throughput* with the influence of branching:

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



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

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



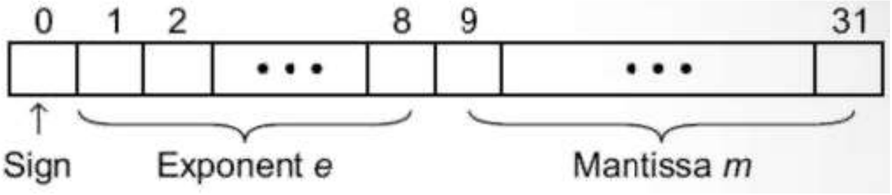
Fixed-Point Operations Fixed-point numbers are represented internally in machines in *sign-magnitude, one's complement*, or *two's complement* notation. Most computers use the two's complement notation because of its unique representation of all numbers (including zero). One's complement notation introduces 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 $e_x \le e_y$ and base $r = 2$.

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

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

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

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

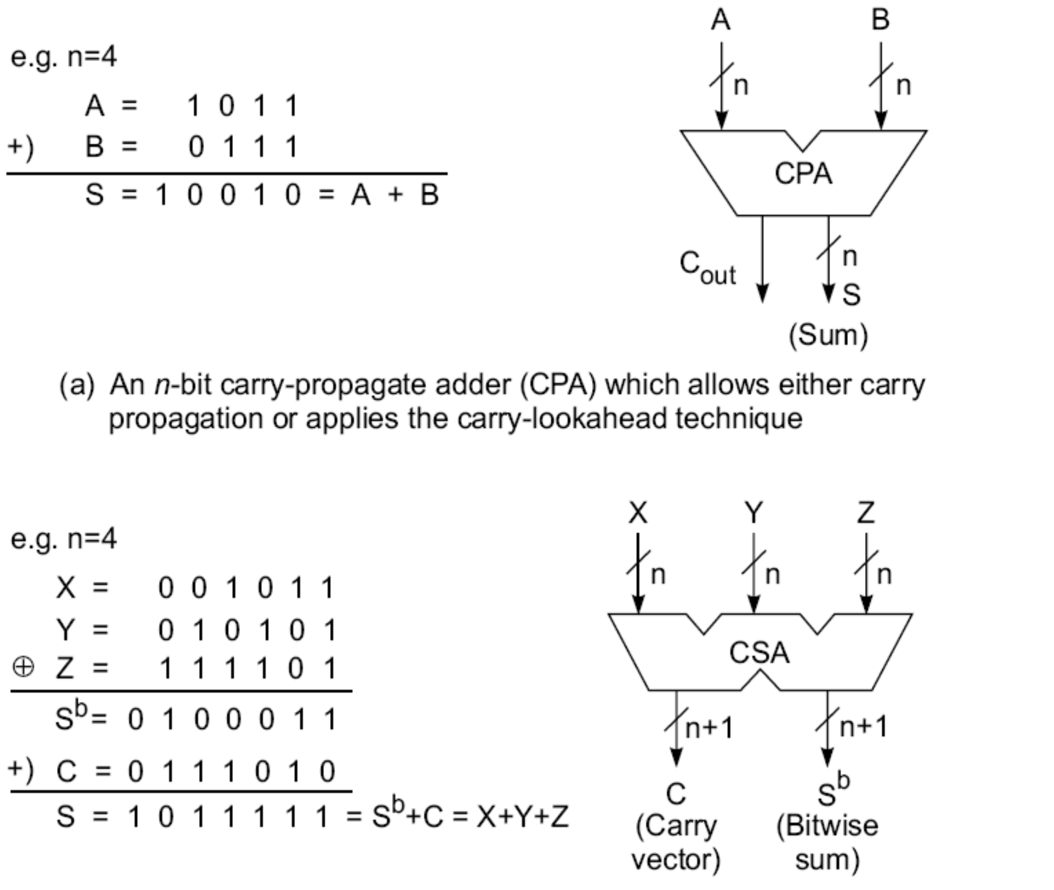
The above equations clearly identify the number of arithmetic operations involved in each floating-point function. These operations can be divided into two halves: One half is for exponent operations such as comparing their relative magnitudes or adding/subtracting them; the other half is for mantissa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

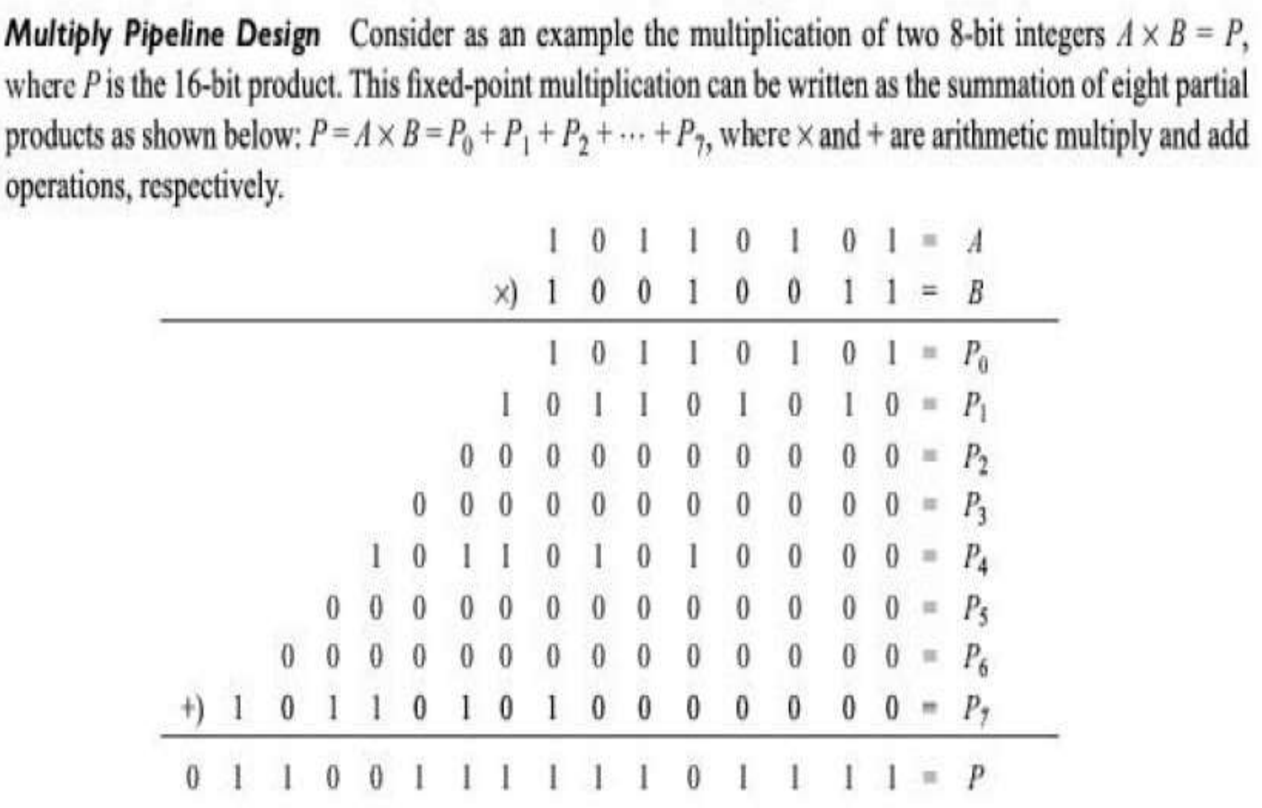
In a CPA, the carries generated in successive digits are allowed to propagate from the low end to the high end, using either ripple carry propagation or some carry looka-head technique.



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

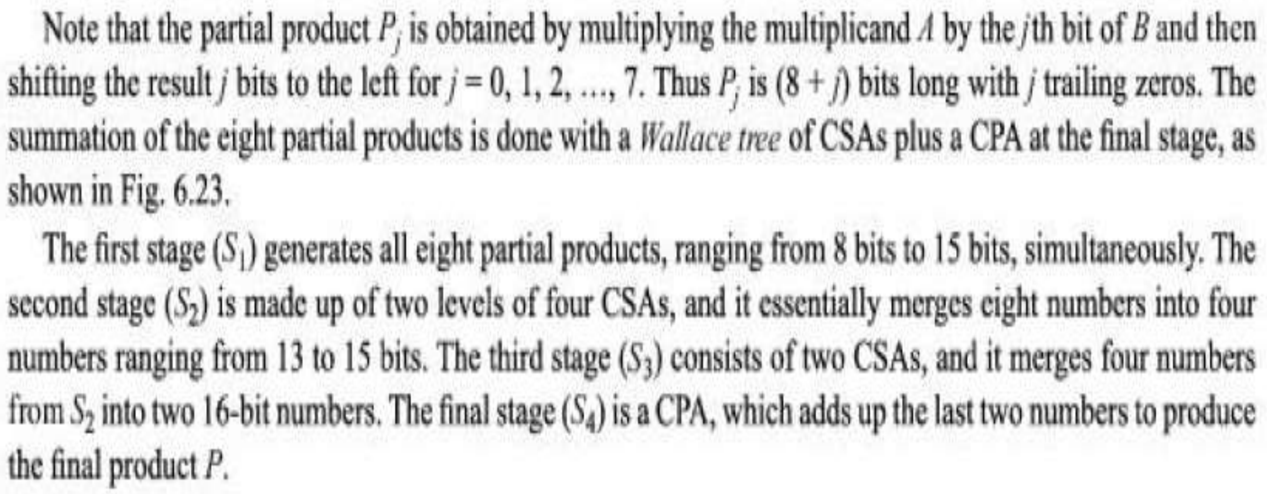
(b) An *n*-bit carry-save adder (CSA), where S^b is the bitwise sum of X, Y, and Z, and C is a carry vector generated without carry propagation between digits

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 × and + are arithmetic multiply and add operations, respectively.

$$
\begin{array}{cccccccccccccccccc}
 & & & & & & & & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & = & A \\
 & & & & & & & \times) & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & = & B \\
\hline
 & & & & & & & & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & = & P_0 \\
 & & & & & & & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & = & P_1 \\
 & & & & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & = & P_2 \\
 & & & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & = & P_3 \\
 & & & & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & = & P_4 \\
 & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & = & P_5 \\
 & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & = & P_6 \\
+) & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & = & P_7 \\
\hline
0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & = & P
\end{array}
$$



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.

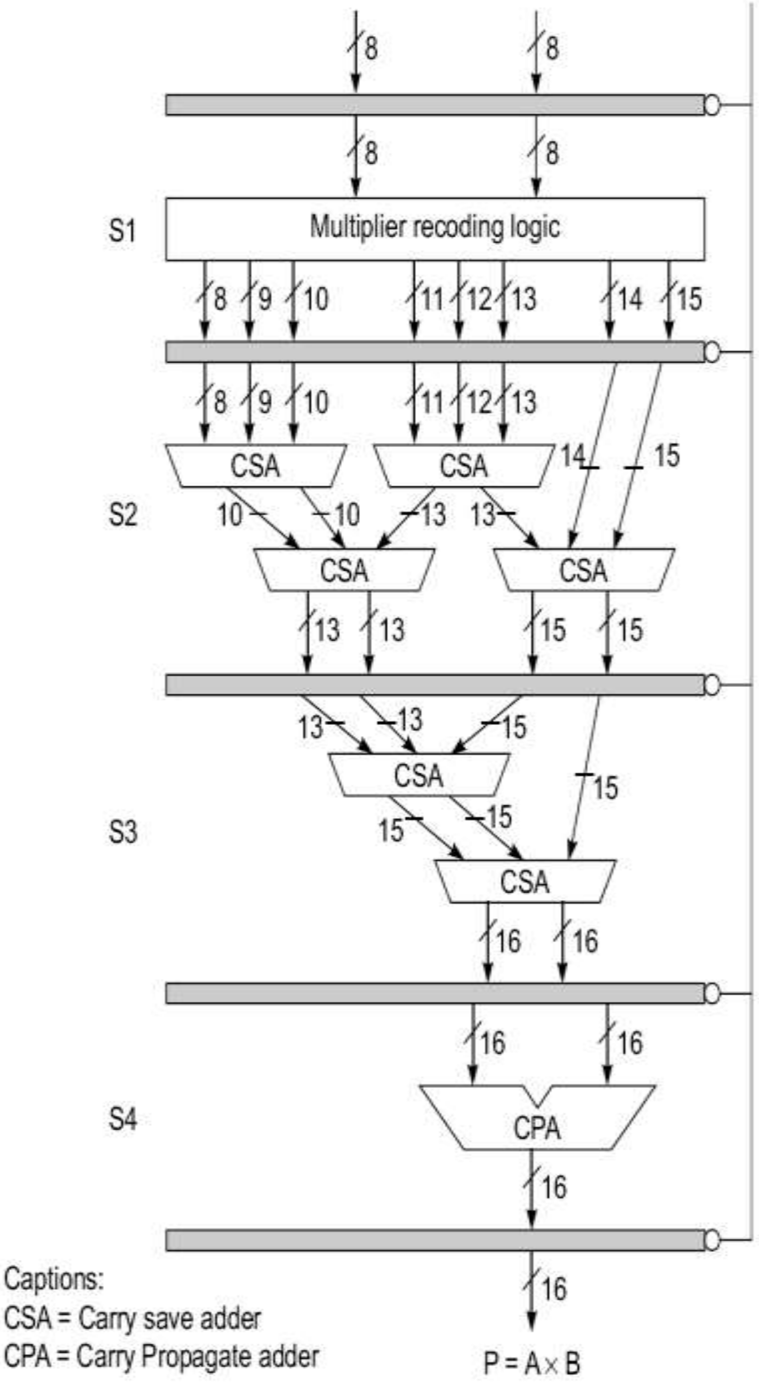


Fig. 6.23 A pipeline unit for fixed-point multiplication of 8-bit integers (The number along each line indicates the line width.)

MODULE-IV

Chapter-7 Multiprocessors and Multicomputers

7.1 Multiprocessor system interconnect

- Parallel processing demands the use of efficient system interconnects for fast communication among multiple processors and shared memory, I/O and peripheral devices.
- Hierarchical buses, crossbar switches and multistage networks are often used for this purpose.
- A generalized multiprocessor system is depicted in Fig. 7.1. This architecture combines features from the UMA, NUMA and COMA models.

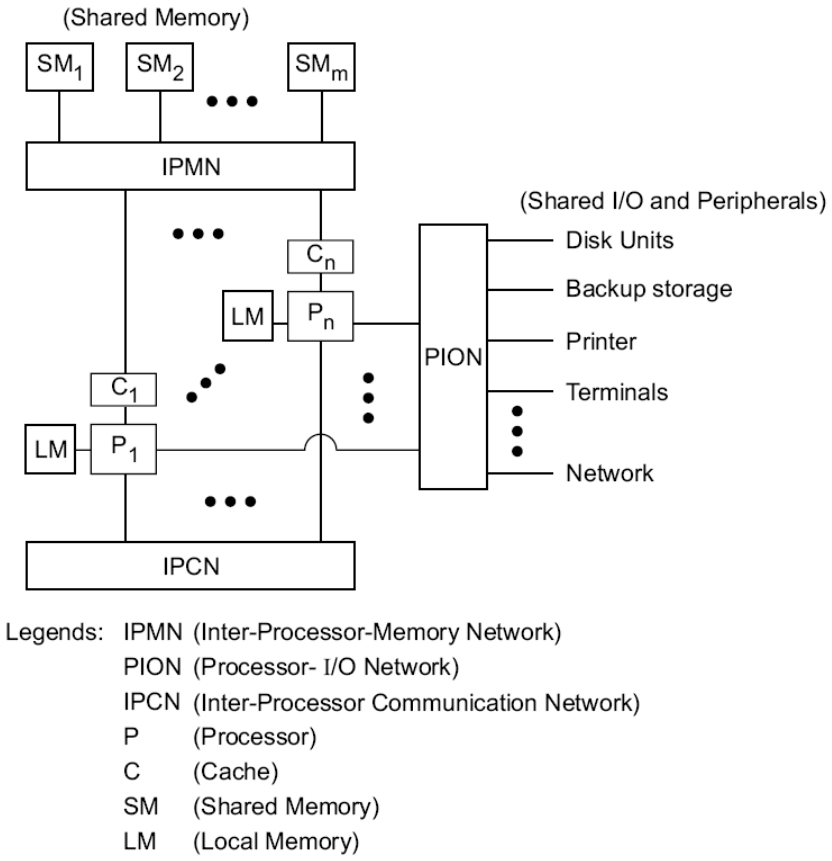


Fig. 7.1 Interconnection structures in a generalized multiprocessor system with local memory, private caches, shared memory, and shared peripherals

- Each processor Pi is attached to its own local memory and private cache.
- These multiple processors connected to share memory through interprocessor memory network (IPMN).

- Processors share the access of I/O and peripheral devices through Processor-I/O Network (PION). Both IPMN and PION are necessary in a shared-resource multiprocessor.
- An optional Interprocessor Communication Network (IPCN) can permit processor communication without using shared memory.

Network Characteristics

The networks are designed with many choices like timing, switching and control strategy like in case of dynamic network the multiprocessors interconnections are under program control.

Timing

- Synchronous – controlled by a global clock which synchronizes all network activity.
- Asynchronous – use handshaking or interlock mechanisms for communication and especially suitable for coordinating devices with different speed.

Switching Method

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

Network Control

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

7.1.1 Hierarchical Bus Systems

- A *bus system* consists of a hierarchy of buses connecting various system and subsystem **components** in a computer.
- Each bus is formed with a number of signal, control, and power lines. Different buses are used to perform different interconnection functions.
- In general, the hierarchy of bus systems are packaged at different levels as depicted in Fig. 7.2, including local buses on boards, backplane buses, and I/O buses.

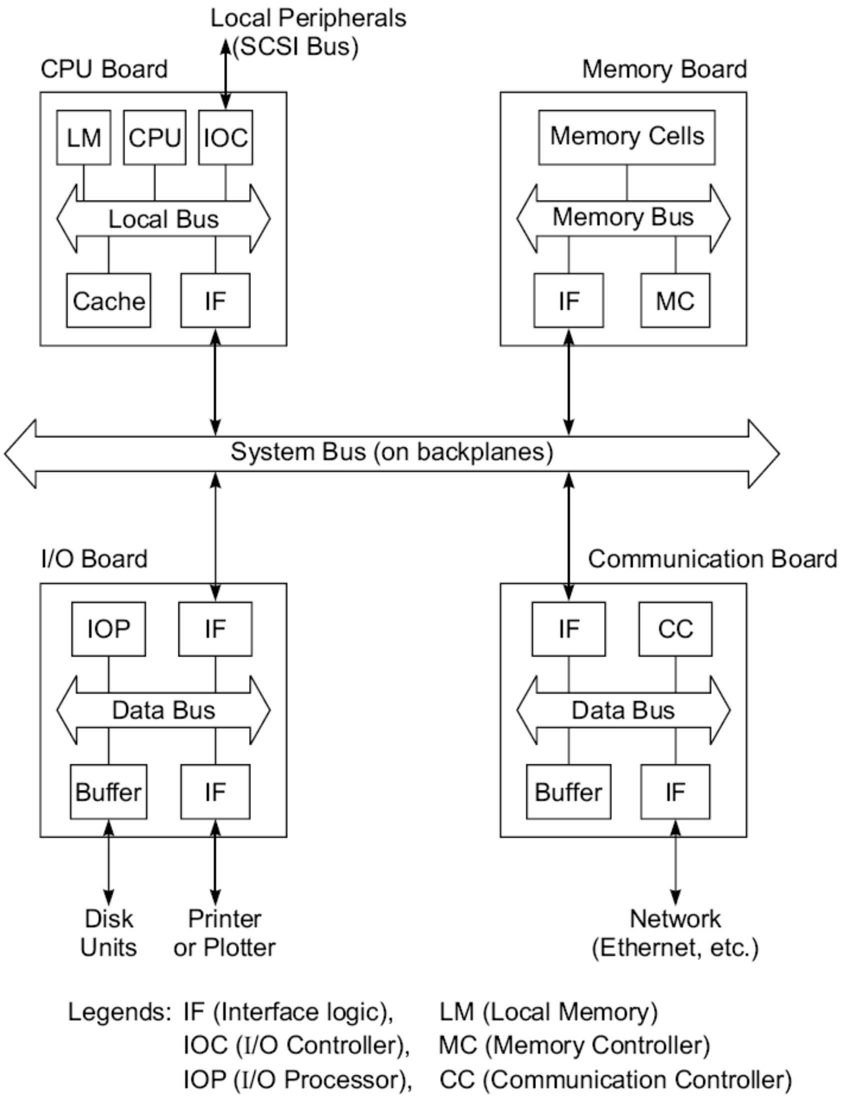


Fig. 7.2 Bus systems at board level, backplane level, and I/O level

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

Backplane Bus

A backplane is a printed circuit on which many connectors are used to plug in functional boards. A system bus, consisting of shared signal pathsand utility lines, is built on the backplane.This system bus provides a common communication path among all plug-in boards.

I/O Bus

Input/Output devices are connected to a comuter system through an I/O bus such as the SCSI(Small Computer Systems Interface) bus.

This bus is made of coaxial cables with taps connecting disks, printer and other devices to a processor through an I/O controller.

Special interface logic is used to connect various board types to the backplane bus.

Hierarchical Buses and Caches

This is a multilevel tree structure in which the leaf nodes are processors and their private caches (denoted P_j and C_{1j} in Fig. 7.3). These are divided into several clusters, each of which is connected through a cluster bus.

An intercluster bus is used to provide communications among the clusters. Second level caches (denoted as C2i) are used between each cluster bus and the intercluster bus. Each second level cache must have a capacity that is at least an order of magnitude larger than the sum of the capacities of all first-level caches connected beneath it.

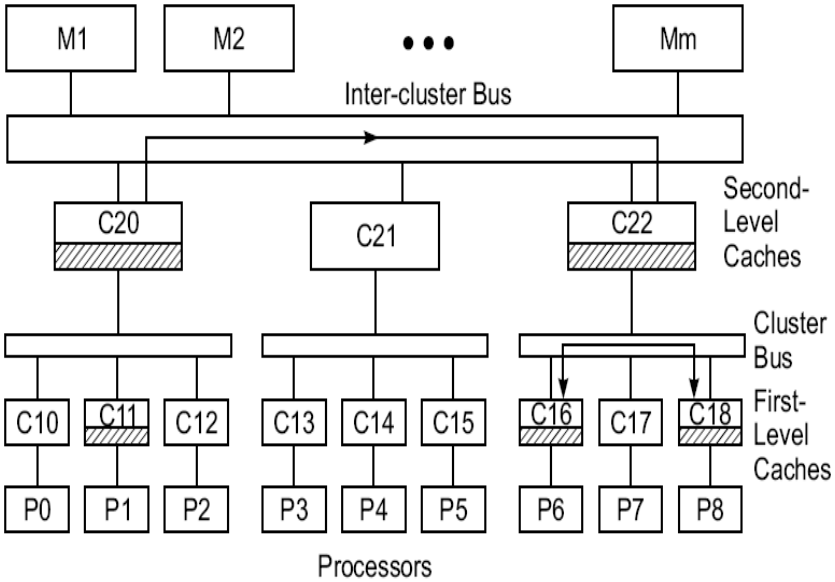


Fig. 7.3 A hierarchical cache/bus architecture for designing a scalable multiprocessor (Courtesy of Wilson; reprinted from *Proc. of Annual Int. Symp. on Computer Architecture, 1987)*

- Each single cluster operates on a single-bus system. Snoopy bus coherence protocols can be used to establish consistency among first level caches belonging to the same cluster.
- Second level caches are used to extend consistency from each local cluster to the upper level.
- The upper level caches form another level of shared memory between each cluster and the main memory modules connected to the intercluster bus.
- Most memory requests should be satisfied at the lower level caches.
- Intercluster cache coherence is controlled among the second-level caches and the resulting effects are passed to the lower level.

7.1.2 Crossbar Switch and Multiport Memory

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

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

Multistage Networks

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

A multistage network is called blocking if the simultaneous connections of some multiple input/output pairs may result in conflicts in the use of switches or communication links.

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

Crossbar Networks

Crossbar networks connect every input to every output through a crosspoint switch. A crossbar network is a single stage, non-blocking permutation network.

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

Crosspoint Switch Design

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

Crosspoint switches must be designed to handle the potential contention for each memory module. A crossbar switch avoids competition for bandwidth by using O(N^2) switches to connect N inputs to N outputs.

Although highly non-scalable, crossbar switches are a popular mechanism for connecting a small number of workstations, typically 20 or fewer.

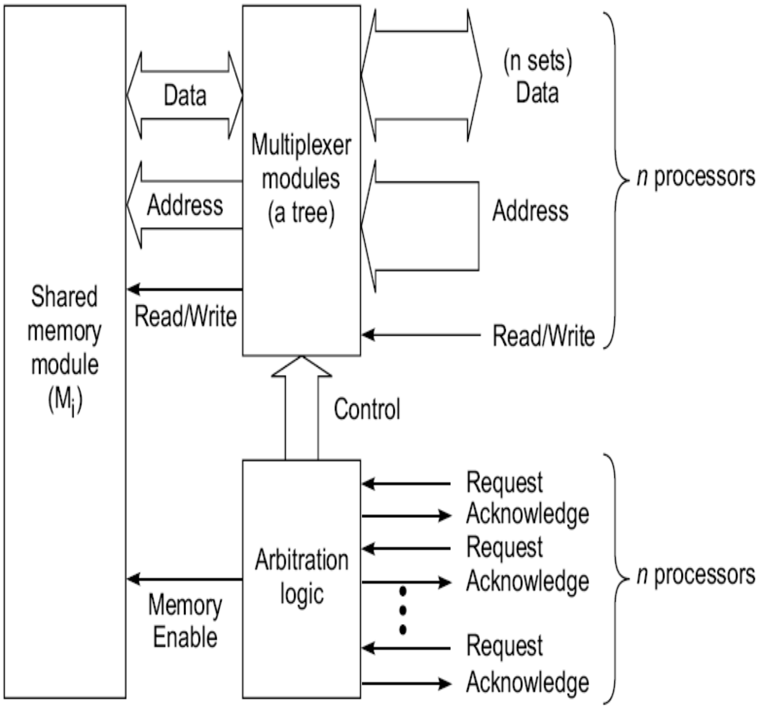


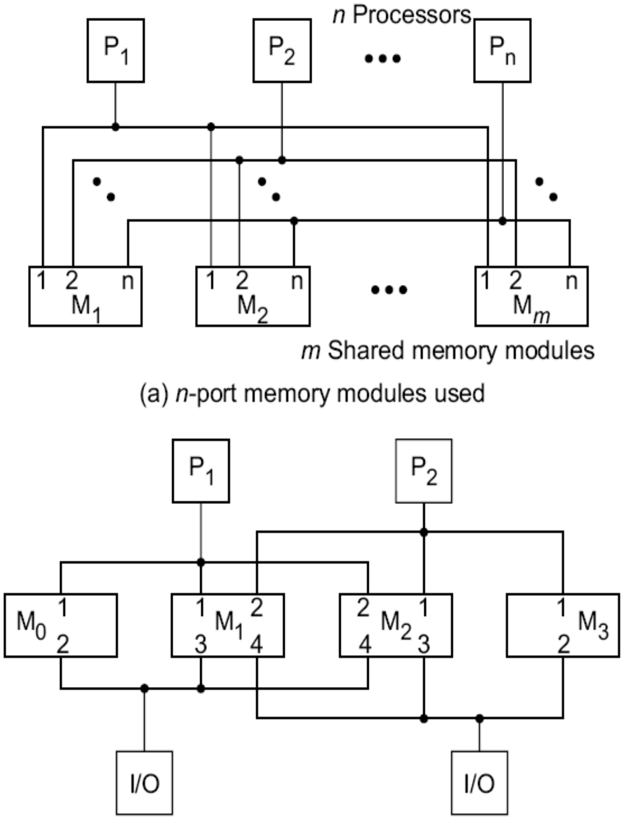
Fig. 7.6 Schematic design of a row of crosspoint switches in a crossbar network

Each processor provides a request line, a read/write line, a set of address lines, and a set of data lines to a crosspoint switch for a single column. The crosspoint switch eventually responds with an acknowledgement when the access has been completed.

Multiport Memory

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

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



(a) *n*-port memory modules used

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

Fig. 7.7 Multiport memory organizations for multiprocessor systems (Courtesy of P. H. Enslow, *ACM Computing Surveys*, March 1977)

A two function switch can assume only two possible state namely state or exchange states. However a four function switch box can be any of four possible states. A multistage network is capable of connecting any input terminal to any output terminal. Multi-stage networks are basically constructed by so called shuffle-exchange switching element, which is basically a 2 x 2 crossbar. Multiple layers of these elements are connected and form the network.

7.1.3 Multistage and Combining Networks

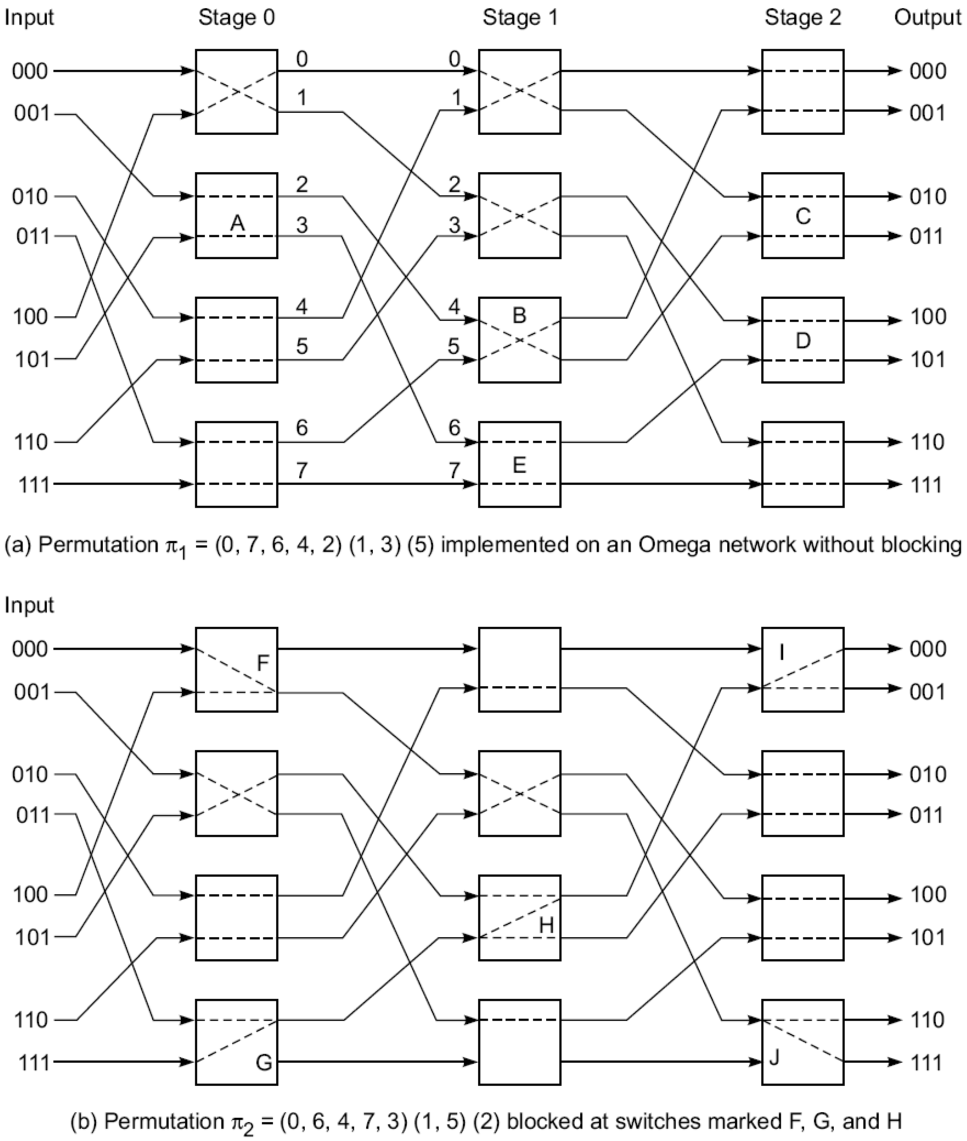
Multistage networks are used to build larger multiprocessor systems. We describe two multistage networks, the Omega network and the Butterfly network, that have been built into commercial machines.

Routing in Omega Networks

An 8-input Omega network is shown in Fig. 7.8.

In general, an n-input Omega network has l**og$_2$n** stages. The stages are labeled from 0 to log$_2$n — 1 from the input end to the output end.

Data routing is controlled by inspecting the destination code in binary. When the ith high-order bit of the destination code is a 0, a 2 x 2 switch at stage *i* connects the input to the upper output. Otherwise, the input is directed to the lower output.



(a) Permutation π_1 = (0, 7, 6, 4, 2) (1, 3) (5) implemented on an Omega network without blocking

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

Fig. 7.8 Two switch settings of an 8 × 8 Omega network built with 2 × 2 switches

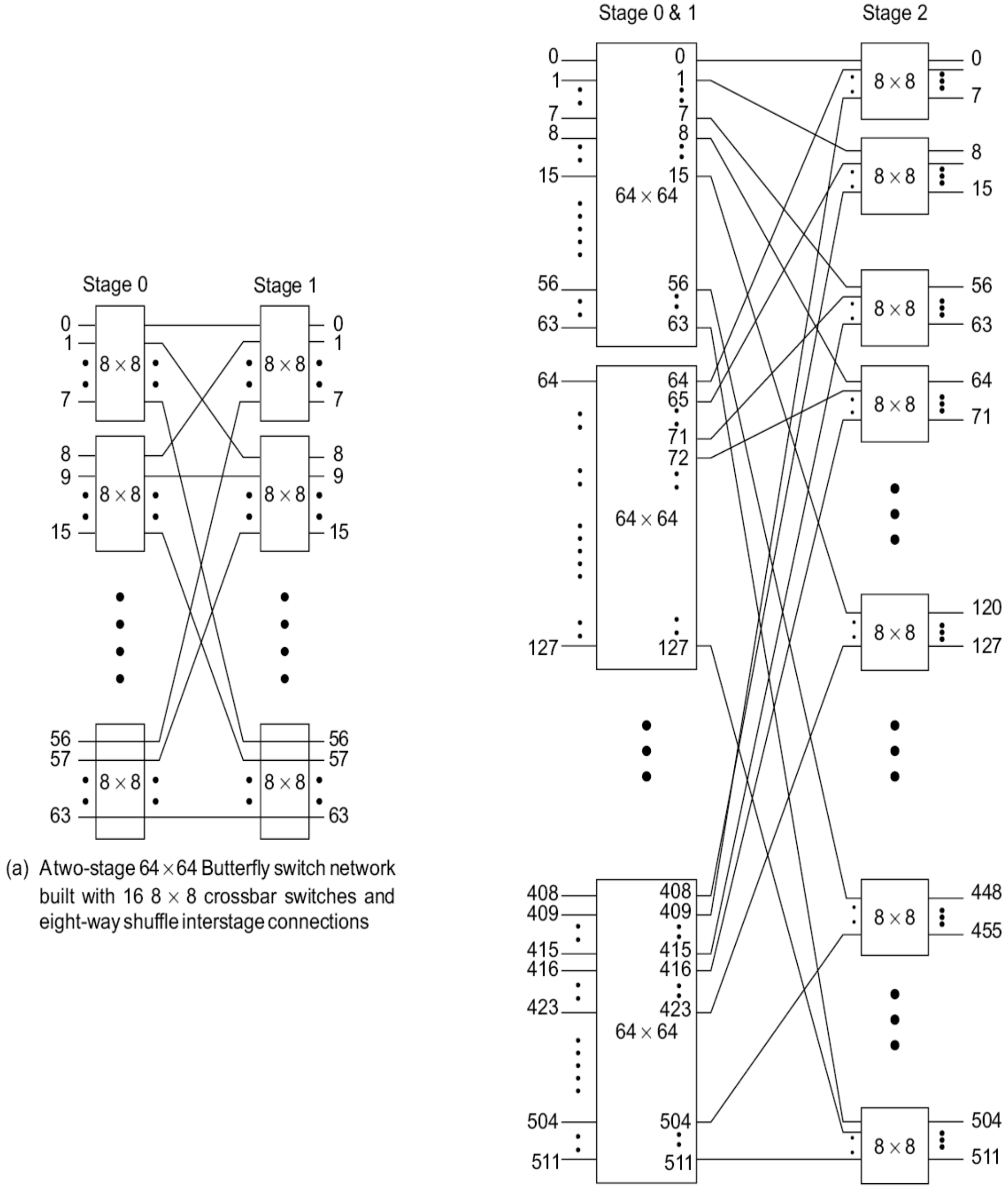
- Two switch settings are shown in Figs. 7.8a and b with respect to permutations Π_1 = (0,7,6,4,2) (1,3)(5) and Π_2= (0,6,4,7,3) (1,5)(2), respectively.
- The switch settings in Fig. 7.8a are for the implementation of Π_1, which maps 0 →7, 7 →6, 6→4, 4→2, 2 →0, 1 →3, 3 →1, 5 →5.
- Consider the routing of a message from input 001 to output 011. This involves the use of switches A, B, and C. Since the most significant bit of the destination 011 is a "zero," switch A must be set straight so that the input 001 is connected to the upper output (labeled 2).

- The middle bit in 011 is a "one," thus input 4 to switch B is connected to the lower output with a "crossover" connection.
- The least significant bit in 011 is a "one," implying a flat connection in switch C.
- Similarly, the switches A, E, and D are set for routing a message from input 101 to output 101. There exists no conflict in all the switch settings needed to implement the permutation Π_1 in Fig. 7.8a.

- Now consider implementing the permutation Π_2 in the 8-input Omega network (Fig. 7.8b0. Conflicts in switch settings do exist in three switches identified as F, G, and H. The conflicts occurring at F are caused by the desired routings 000→ 110 and 100→ 111.
- Since both destination addresses have a leading bit 1, both inputs to switch F must be connected to the lower output.
- To resolve the conflicts, one request must be blocked.
- Similarly we see conflicts at switch G between 011→ 000 and 111→011, and at switch H between 101→001 and 011→ 000. At switches I and J, broadcast is used from one input to two outputs, which is allowed if the hardware is built to have four legitimate states as shown in fig. 2.24a.
- The above example indicates the fact that not all permutations can be implemented in one pass through the Omega network.

Routing in Butterfly Networks

- This class of networks is constructed with crossbar switches as building blocks. Fig. 7.10 shows two Butterfly networks of different sizes.
- Fig. 10a shows a 64-input Butterfly network built with two stages ($2=\log_8 64$) of 8X8 crossbar switches.
- The eight-way shuffle function is used to establish the interstage connections between stage 0 and stage 1.
- In Fig. 7.10b, a three-stage Butterfly network is constructed for 512 inputs, again with 8X8 crossbar switches.
- Each of the 64X64 boxes in Fig. 7.10b is identical to the two-stage Butterfly network in Fig. 7.10a.



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

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

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

- In total, sixteen 8x8 crossbar switches are used in Fig. 7.10a and 16 x 8+8 x 8 = 192 are used in Fig. 7.10b. Larger Butterfly networks can be modularly constructed using more stages.
- Note that no broadcast connections are allowed in a Butterfly network, making these networks a restricted subclass of Omega networks.

The Hot-Spot Problem

- When the network traffic is nonuniform, a hot spot may appear corresponding to a certain memory module being excessively accessed by many processors at the same time.
- For example, a semaphore variable being used as a synchronization barrier may become a hot spot since it is shared by many processors.
- Hot spots may degrade the network performance significantly. In the NYU Ultracomputer and the IBM RP3 multiprocessor, a combining mechanism has been added to the Omega network.
- The purpose was to combine multiple requests heading for the same destination at switch points where conflicts are taking place.
- An atomic read-modify-write primitive Fetch&Add(x,e), has been developed to perform parallel memory updates using the combining network.

Fectch&Add

- This atomic memory operation is effective in implementing an N-way synchronization with a complexity independent of N.
- In a Fetch&Add(x, e) operation, i is an integer variable in shared memory and e is an integer increment.
- When a single processor executes this operation, the semantics is

```
Fetch&Add(x, e)
  {   temp $\leftarrow$ x;
      x $\leftarrow$ temp + e;                                  (7.1)
      return temp   }
```

- When N processes attempt to Fetch&Add(x, e) the same memory word simultaneously, the memory is updated only once following a serialization principle.
- The sum of the N increments, $e_1 + e_2 + \cdots + e_N$, is produced in any arbitrary serialization of the N requests.
- This sum is added to the memory word x, resulting in a new value $x + e_1 + e_2 + \cdots + e_N$
- The values returned to the N requests are all unique, depending on the serialization order followed.
- The net result is similar to a sequential execution of N Fetch&Adds but is performed in one indivisible operation.
- Two simultaneous requests are combined in a switch as illustrated in Fig. 7.11.

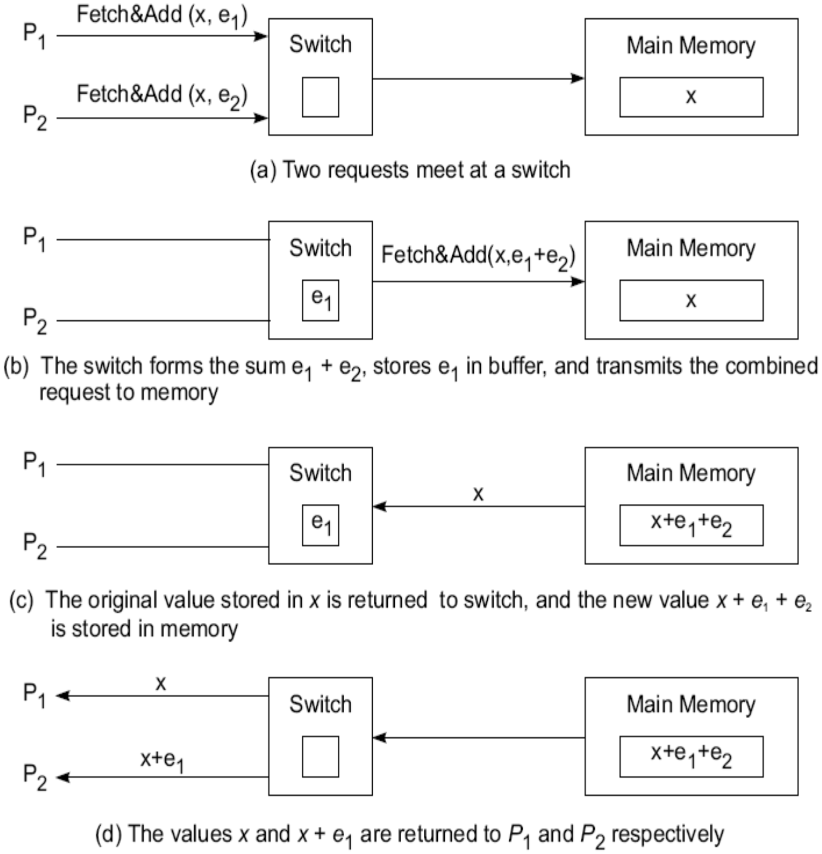

P1
P2
Fetch&Add (x, e1)
Fetch&Add (x, e2)
Switch
Main Memory
x
(a) Two requests meet at a switch
Fetch&Add(x,e1+e2)
e1
(b) The switch forms the sum e1 + e2, stores e1 in buffer, and transmits the combined request to memory
x+e1+e2
(c) The original value stored in x is returned to switch, and the new value x + e1 + e2 is stored in memory
x+e1
(d) The values x and x + e1 are returned to P1 and P2 respectively

Fig. 7.11 Two Fetch&Add operations are combined to access a shared variable simultaneously via a combining network

One of the following operations will be performed if processor P1 executes **Ans_1 ← Fetch&Add(x,e_1)** and P_2 executes **Ans_2 ← Fetch&Add(x,e_2)** simultaneously on the shared variable x.

If the request from P_1 is executed ahead of that from P_2, the following values are returned:

$Ans_1 \leftarrow x$

$Ans_2 \leftarrow x + e_1$ (7.2)

If the execution order is reversed, the following values arc returned:

$Ans_1 \leftarrow x + e_2$

$Ans_2 \leftarrow x$

Regardless of the executing order, the value **$x + e_1 + e_2$** is stored in memory.

It is the responsibility of the switch box to form the sum **$e_1 + e_2$**, transmit the combined request **Fetch&Add(x, $e_1 + e_2$)**, store the value **e1 (or e2)** in a wait buffer of the switch and return the values x and **$x + e_1$** to satisfy the original requests **Fetch&Add(x, e_1)** and **Fetch&Add(x, e_2)** respectively, as shown in fig. 7.11 in four steps.

7.2 Cache Coherence and Synchronization Mechanisms

Cache Coherence Problem:

- In a memory hierarchy for a multiprocessor system, data inconsistency may occur between adjacent levels or within the same level.
- For example, the cache and main memory may contain inconsistent copies of the same data object.
- Multiple caches may possess different copies of the same memory block because multiple processors operate asynchronously and independently.
- Caches in a multiprocessing environment introduce the cache coherence problem. When multiple processors maintain locally cached copies of a unique shared-memory location, any local modification of the location can result in a globally inconsistent view of memory.
- Cache coherence schemes prevent this problem by maintaining a uniform state for each cached block of data.
- Cache inconsistencies caused by data sharing, process migration or I/O are explained below.

Inconsistency in Data sharing:

The cache inconsistency problem occurs only when multiple private caches are used.

In general, three sources of the problem are identified:

- ✓ sharing of writable data,
- ✓ process migration
- ✓ I/O activity.

- Consider a multiprocessor with two processors, each using a private cache and both sharing the main memory.
- Let X be a shared data element which has been referenced by both processors. Before update, the three copies of X are consistent.
- If processor P writes new data X’ into the cache, the same copy will be written immediately into the shared memory under a write through policy.
- In this case. inconsistency occurs between the two copies (X and X') in the two caches.
- On the other hand, inconsistency may also occur when a write back policy is used, as shown on the right.
- The main memory will be eventually updated when the modified data in the cache are replaced or invalidated.

Process Migration and I/O

The figure shows the occurrence of inconsistency after a process containing a shared variable X migrates from processor 1 to processor 2 using the write-back cache on the right. In the middle, a process migrates from processor 2 to processor1 when using write-through caches.

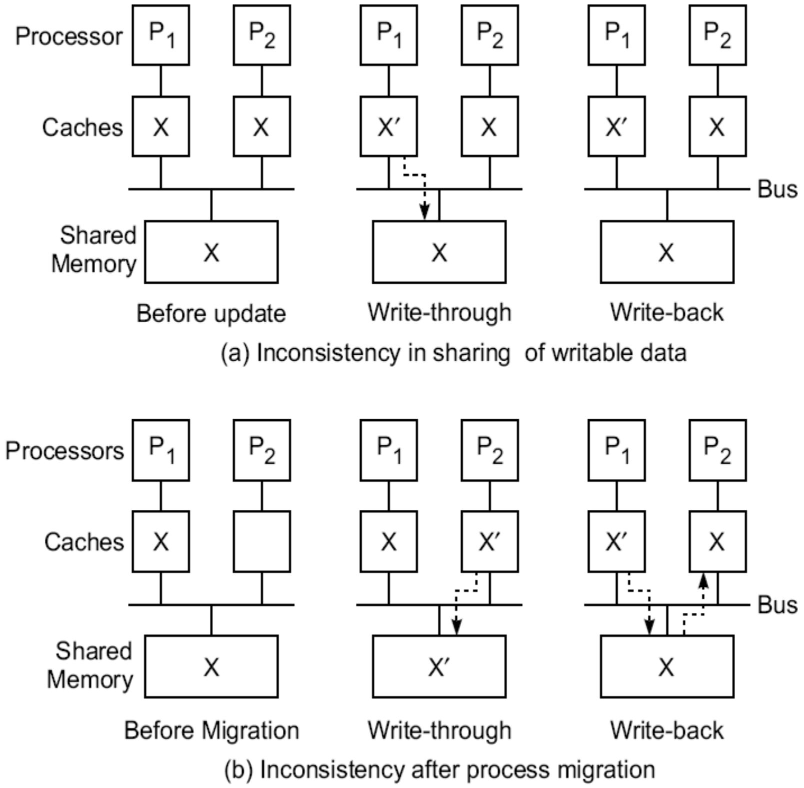


Fig. 7.12 Cache coherence problems in data sharing and in process migration (Adapted from Dubois, Scheurich, and Briggs 1988)

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

Two Protocol Approaches for Cache Coherence

- Many of the early commercially available multiprocessors used bus-based memory systems.
- A bus is a convenient device for ensuring cache coherence because it allows all processors in the system to observe ongoing memory transactions.
- If a bus transaction threatens the consistent state of a locally cached object, the cache controller can take appropriate actions to invalidate the local copy.
- Protocols using this mechanism to ensure coherence are called snoopy protocols because each cache snoops on the transactions of other caches.

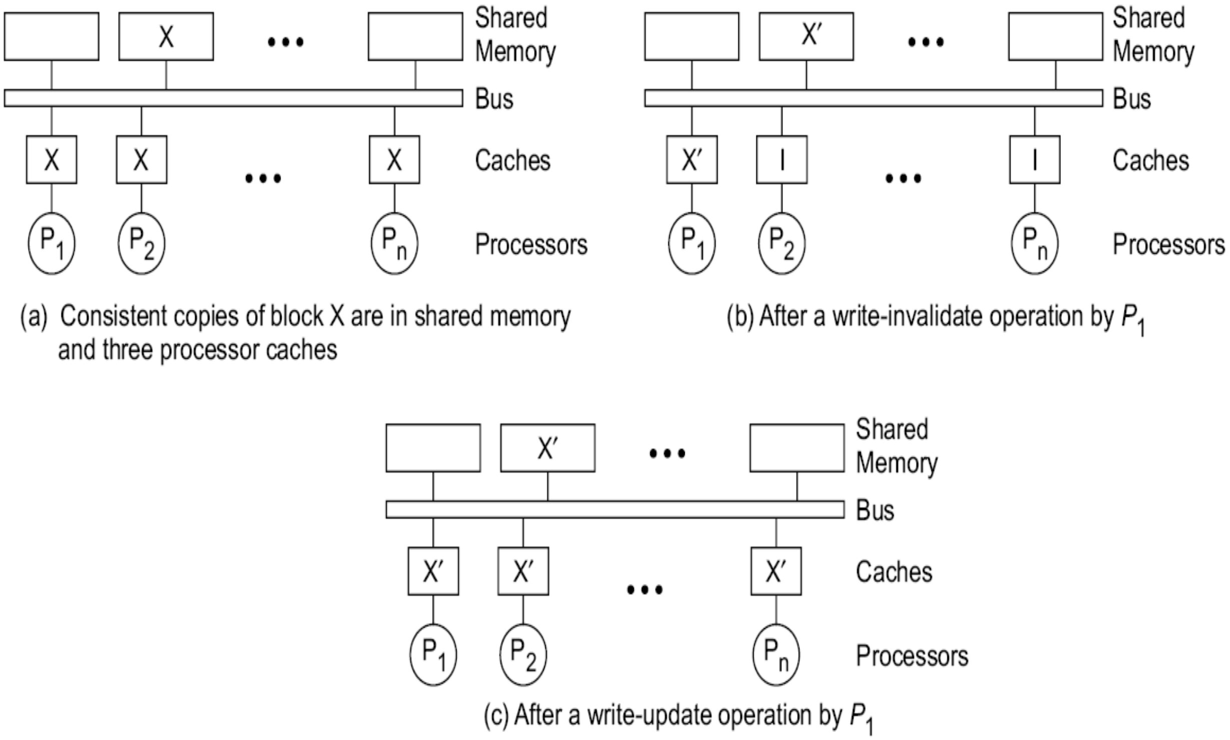
- On the other hand, scalable multiprocessor systems interconnect processors using short point-to-point links in direct or multistage networks.
- Unlike the situation in buses, the bandwidth of these networks increases as more processors are added to the system.
- However, such networks do not have a convenient snooping mechanism and do not provide an efficient broadcast capability. In such systems, the cache coherence problem can be solved using some variant of directory schemes.

Protocol Approaches for Cache Coherence:

1. Snoopy Bus Protocol
2. Directory Based Protocol

1. Snoopy Bus Protocol

- Snoopy protocols achieve data consistency among the caches and shared memory through a bus watching mechanism.
- In the following diagram, two snoopy bus protocols create different results. Consider 3 processors (P_1, P_2, P_n) maintaining consistent copies of block X in their local caches (Fig. 7.14a) and in the shared memory module marked X.



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

(b) After a write-invalidate operation by P_1

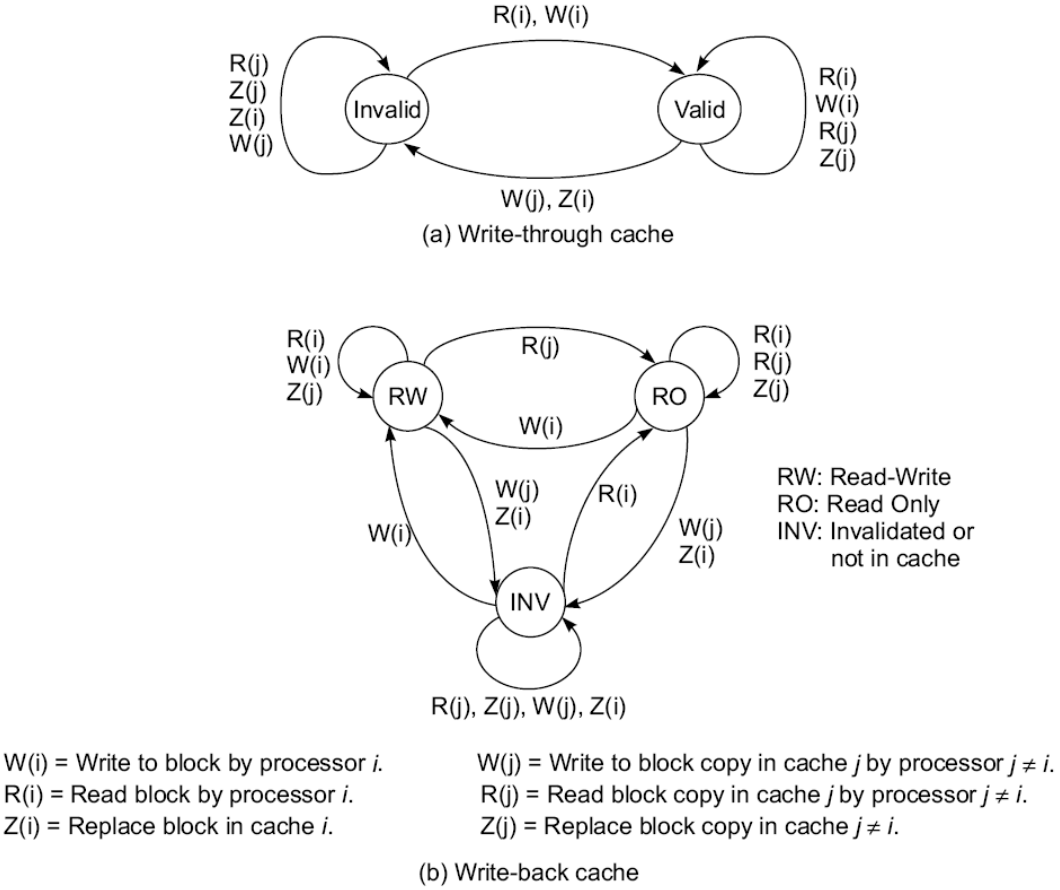
(c) After a write-update operation by P_1

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

- Using a write-invalidate protocol, the processor P_1 modifies (writes) its cache from X to X', and all other copies are invalidated via the bus (denoted I in Fig. 7.14b). Invalidated blocks are called dirty, meaning they should not be used.
- The write-update protocol (Fig. 7.14c) demands the new block content X' be broadcast to all cache copies via the bus.
- The memory copy also updated if write through caches are used. In using write-back caches, the memory copy is updated later at block replacement time.

Write Through Caches:

- The states of a cache block copy change with respect to read, write and replacement operations in the cache shows the state transitions for two basic write-invalidate snoopy protocols developed for write-through and write-back caches, respectively.
- A block copy of a write through cache i attached to processor i can assume one of two possible cache states: **valid or invalid.**


R(i), W(i)
R(j)
Z(j)
Z(i)
W(j)
Invalid
Valid
R(i)
W(i)
R(j)
Z(j)
W(j), Z(i)
R(i)
W(i)
Z(j)
R(j)
RW
RO
W(i)
R(i)
R(j)
Z(j)
W(j)
Z(i)
R(i)
W(i)
W(j)
Z(i)
INV
R(j), Z(j), W(j), Z(i)
RW: Read-Write
RO: Read Only
INV: Invalidated or not in cache
W(i) = Write to block by processor *i*.
R(i) = Read block by processor *i*.
Z(i) = Replace block in cache *i*.
W(j) = Write to block copy in cache *j* by processor *j* ≠ *i*.
R(j) = Read block copy in cache *j* by processor *j* ≠ *i*.
Z(j) = Replace block copy in cache *j* ≠ *i*.

(a) Write-through cache

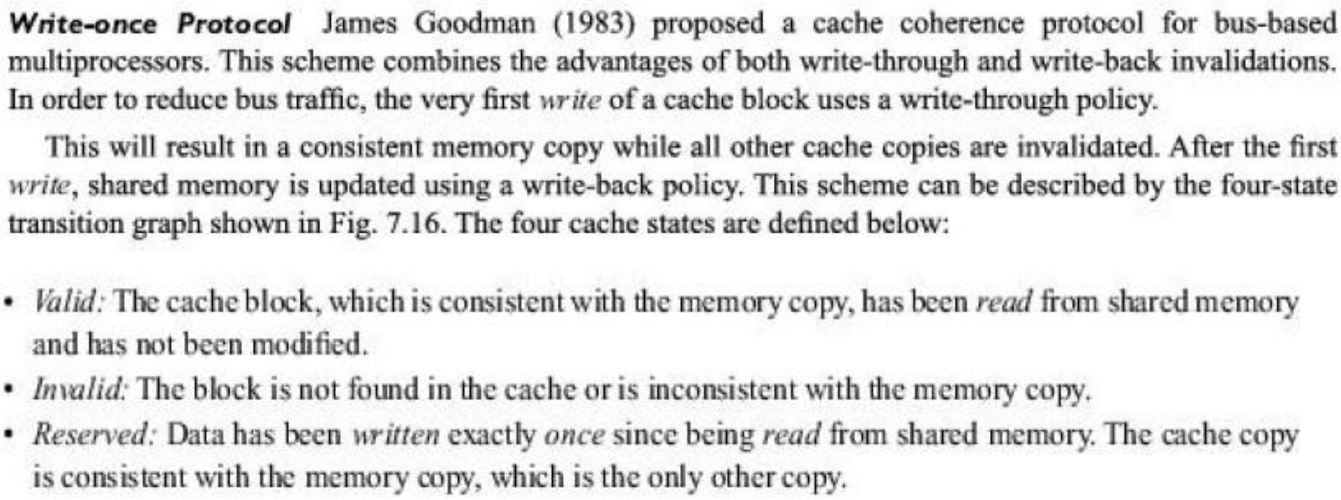
(b) Write-back cache

Fig. 7.15 Two state-transition graphs for a cache block using write-invalidate snoopy protocols (Adapted from Dubois, Scheurich, and Briggs, 1988)

- A remote processor is denoted j, where j # i. For each of the two cache states, six possible events may take place.
- Note that all cache copies of the same block use the same transition graph in making state changes.
- In a valid state (Fig. 7.15a), all processors can read (R(i), R(j)) safely. Local processor i can also write(W(i)) safely in a valid state. The invalid state corresponds to the case of the block either being invalidated or being replaced (Z(i) or Z(j)).

Write Back Caches:

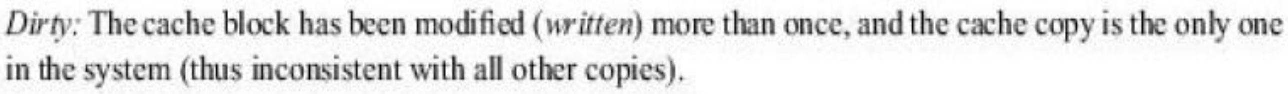
- The valid state of a write-back cache can be further split into two cache states, **Labeled RW(read-write)** and **RO(read-only)** as shown in Fig.7.15b.
- The INV (invalidated or not-in-cache) cache state is equivalent to the invalid state mentioned before. This three-state coherence scheme corresponds to an ownership protocol.
- When the memory owns a block, caches can contain only the RO copies of the block. In other words, multiple copies may exist in the RO state and every processor having a copy (called a keeper of the copy) can read (R(i),R(j)) safely.
- The Inv state is entered whenever a remote processor writes (W(j)) its local copy or the local processor replaces (Z(i)) its own block copy.
- The RW state corresponds to only one cache copy existing in the entire system owned by the local processor i.
- Read (R(i)) and write(W(i)) can be safely performed in the RW state. From either the RO state or the INV state, the cache block becomes uniquely owned when a local write (W(i)) takes place.



Write-once Protocol James Goodman (1983) proposed a cache coherence protocol for bus-based multiprocessors. This scheme combines the advantages of both write-through and write-back invalidations. In order to reduce bus traffic, the very first *write* of a cache block uses a write-through policy.

This will result in a consistent memory copy while all other cache copies are invalidated. After the first *write*, shared memory is updated using a write-back policy. This scheme can be described by the four-state transition graph shown in Fig. 7.16. The four cache states are defined below:

- *Valid:* The cache block, which is consistent with the memory copy, has been *read* from shared memory and has not been modified.
- *Invalid:* The block is not found in the cache or is inconsistent with the memory copy.
- *Reserved:* Data has been *written* exactly *once* since being *read* from shared memory. The cache copy is consistent with the memory copy, which is the only other copy.



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

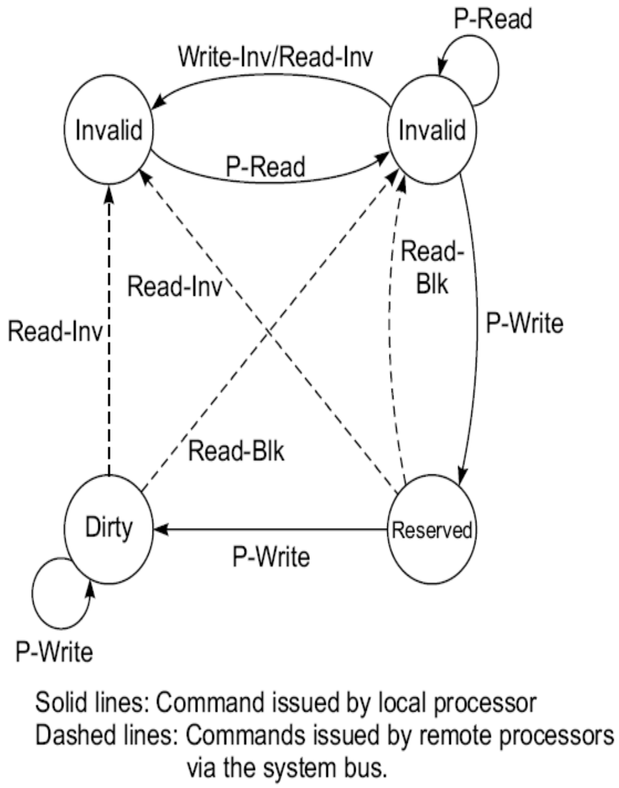

P-Read
Write-Inv/Read-Inv
Invalid
Invalid
P-Read
Read-Inv
Read-Inv
Read-Blk
Read-Blk
P-Write
Dirty
Reserved
P-Write
P-Write
Solid lines: Command issued by local processor
Dashed lines: Commands issued by remote processors via the system bus.

Fig. 7.16 Goodman's write-once cache coherence protocol using the write invalidate policy on write-back caches (Adapted from James Goodman 1983, reprinted from Stenstrom, *IEEE Computer*, June 1990)

2. Directory Based Protocol

A write-invalidate protocol may lead to heavy bus traffic caused by *read-misses*, resulting from the processor updating a variable and other processors trying to read the same variable. On the other hand, the write-update protocol may update data items in remote caches which will never be used by other processors. In fact, these problems pose additional limitations in using buses to build large multiprocessors.

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

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

Tang (1976) proposed the first directory scheme, which used a *central directory* containing duplicates of all cache directories. This central directory, providing all the information needed to enforce consistency, is usually very large and must be associatively searched, like the individual cache directories. Contention and long search times are two drawbacks in using a central directory for a large multiprocessor.

A distributed-directory scheme was proposed by Censier and Feautrier (1978). Each memory module maintains a separate directory which records the state and presence information for each memory block. The state information is local, but the presence information indicates which caches have a copy of the block.

In Fig. 7.17, a *read-miss* (thin lines) in cache 2 results in a request sent to the memory module. The memory controller retransmits the request to the dirty copy in cache 1. This cache *writes back* its copy. The memory module can supply a copy to the requesting cache. In the case of a *write-hit* at cache 1 (bold lines), a command is sent to the memory controller, which sends invalidations to all caches (cache 2) marked in the presence vector residing in the directory D_1.

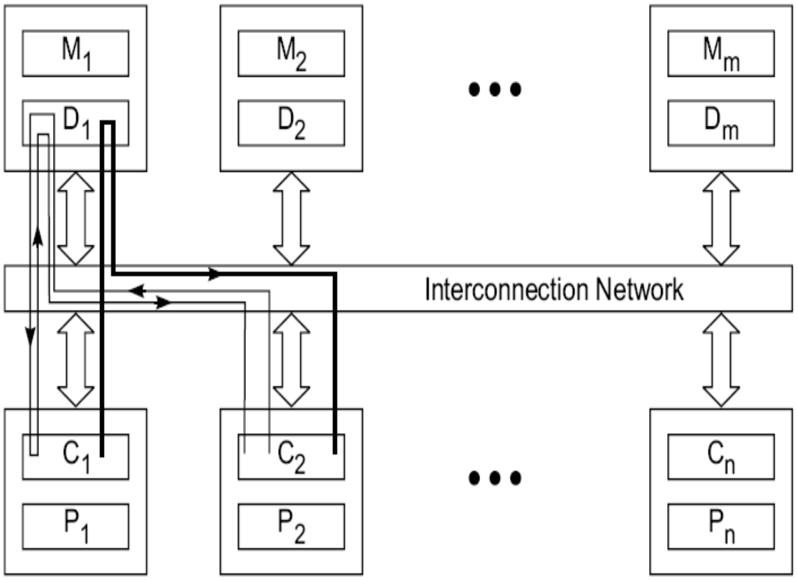


Fig. 7.17 Basic concept of a directory-based cache coherence scheme (Courtesy of Censier and Feautrier, *IEEE Trans. Computers*, Dec. 1978)

A cache-coherence protocol that does not use broadcasts must store the locations of all cached copies of each block of shared data. This list of cached locations, whether centralized or distributed, is called a *cache directory*. A directory entry for each block of data contains a number of *pointers* to specify the locations of copies of the block. Each directory entry also contains a dirty bit to specify whether a particular cache has permission to write the associated block of data.

Different types of directory protocols fall under three primary categories: *full map directories*, *limited directories*, and *chained directories*. Full-map directories store enough data associated with each block in global memory so that every cache in the system can simultaneously store a copy of any block of data. That is, each directory entry contains *N* pointers, where *N* is the number of processors in the system.

Full-Map Directories The full-map protocol implements directory entries with one bit per processor and a dirty bit. Each bit represents the status of the block in the corresponding processor's cache (present or absent). If the dirty bit is set, then one and only one processor's bit is set and that processor can write into the block.

A cache maintains two bits of state per block. One bit indicates whether a block is valid, and the other indicates whether a valid block may be written. The cache coherence protocol must keep the state bits in the memory directory and those in the cache consistent.

Figure 7.18a illustrates three different states of a full-map directory. In the first state, location X is missing in all of the caches in the system. The second state results from three caches (C1, C2, and C3) requesting copies of location X. Three pointers (processor bits) are set in the entry to indicate the caches that have copies of the block of data. In the first two states, the dirty bit on the left side of the directory entry is set to clean (C), indicating that no processor has permission to write to the block of data. The third state results from cache C3 requesting write permission for the block. In the final state, the dirty bit is set to dirty (D), and there is a single pointer to the block of data in cache C3.

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

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

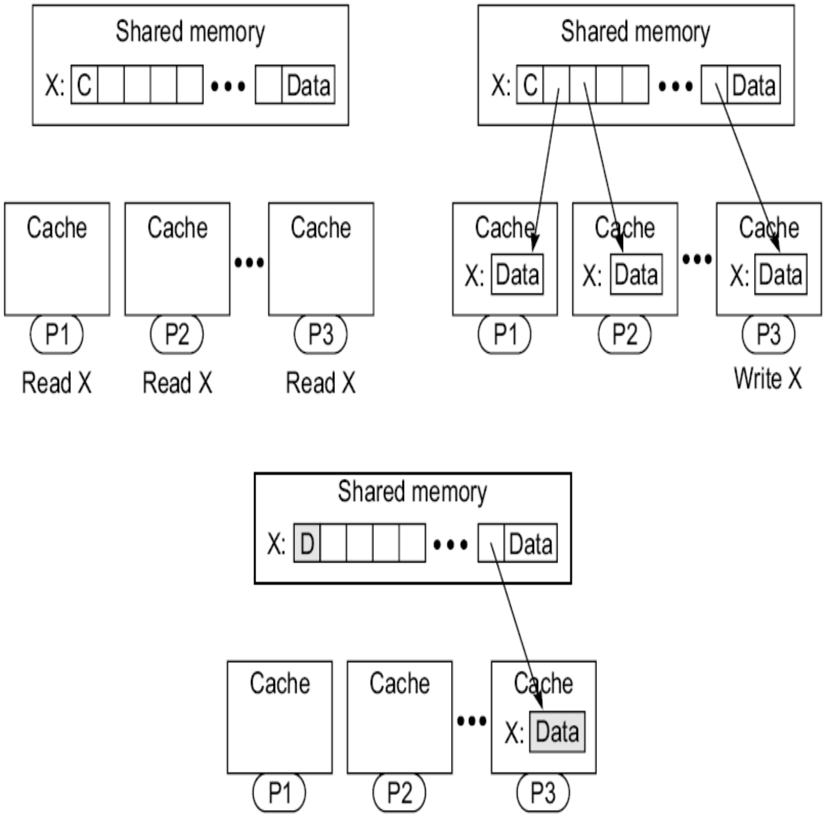
(2) Cache C3 issues a write request to the memory module containing location X and stalls processor P3.

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

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

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

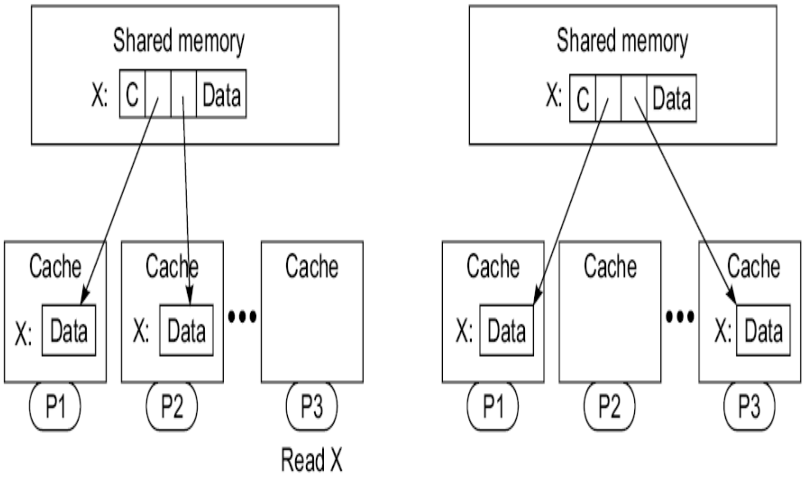
(6) Cache C3 receives the write permission message, updates the state in the cache, and reactivates processor P3.



(a) Three states of a full-map directory

Limited Directories Limited directory protocols are designed to solve the directory size problem. Restricting the number of simultaneously cached copies of any particular block of data limits the growth of the directory to a constant factor.

A directory protocol can be classified as $Dir_i\ X$ using the notation from Agarwal et al (1988). The symbol i stands for the number of pointers, and X is NB for a scheme with no broadcast. A full-map scheme without broadcast is represented as $Dir_N\ NB$. A limited directory protocol that uses $i < N$ pointers is denoted $Dir_i\ NB$. The limited directory protocol is similar to the full-map directory, except in the case when more than i caches request read copies of a particular block of data.



(b) Eviction in a limited directory

Figure 7.18b shows the situation when three caches request read copies in a memory system with a Dir_2 NB protocol. In this case, we can view the two-pointer directory as a two-way set-associative cache of pointers to shared copies. When cache C3 requests a copy of location X, the memory module must invalidate the copy in either cache Cl or cache C2. This process of pointer replacement is called *eviction*. Since the directory acts as a set-associative cache, it must have a pointer replacement policy.

If the multiprocessor exhibits processor locality in the sense that in any given interval of time only a small subset of all the processors access a given memory word, then a limited directory is sufficient to capture this small worker set of processors.

Chained Directories Chained directories realize the scalability of limited directories without restricting the number of shared copies of data blocks. This type of cache coherence scheme is called a *chained* scheme because it keeps track of shared copies of data by maintaining a chain of directory pointers.

The simpler of the two schemes implements a singly linked chain, which is best described by example (Fig. 7.18c). Suppose there are no shared copies of location X. If processor P1 reads location X, the memory sends a copy to cache C1, along with a *chain termination* (CT) pointer. The memory also keeps a pointer to cache C1. Subsequently, when processor P2 reads location X, the memory sends a copy to cache C2, along with the pointer to cache C1. The memory then keeps a pointer to cache C2.

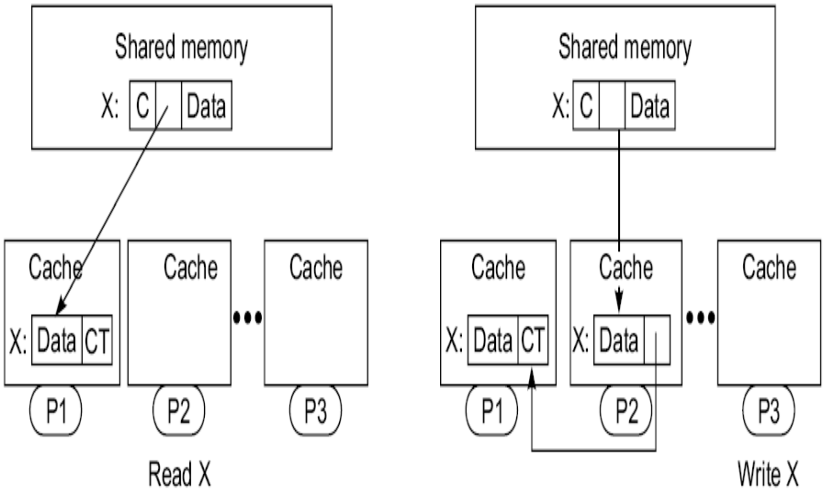
By repeating the above step, all of the caches can cache a copy of the location X. If processor P3 writes to location X, it is necessary to send a data invalidation message down the chain. To ensure sequential consistency, the memory module denies processor P3 write permission until the processor with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

The possibility of cache block replacement complicates chained-directory protocols.

Suppose that caches C1 through CN all have copies of location X and that location X and location Y map to the same (direct-mapped) cache line. If processor P_i reads location Y, it must first evict location X from its cache with the following possibilities:

(1) Send a message down the chain to cache C_{i-1} with a pointer to cache C_{i+1} and splice C_i out of the chain, or

(2) Invalidate location X in cache C_{i+1} through cache C_N.



(c) The chained directory

Fig. 7.18 Three types of cache directory protocols (Courtesy of Chaiken et al., *IEEE Computer*, June 1990)

7.4 Message – Passing Mechanisms

Message passing in a multicomputer network demands special hardware and software support. In this section, we study the store-and-forward and wormhole routing schemes and analyze their communication latencies. We introduce the concept of virtual channels. Deadlock situations in a message-passing network are examined. We show how to avoid deadlocks using virtual channels.

7.4.1 Message-Routing Schemes

Message formats are introduced below. Refined formats led to the improvement from store-and-forward to wormhole routing in two generations of multicomputers. A handshaking protocol is described for asynchronous pipelining of successive routers along a communication path. Finally, latency analysis is conducted to show the time difference between the two routing schemes presented.

Message Formats Information units used in message routing are specified in Fig. 7.26. A *message* is the logical unit for internode communication. It is often assembled from an arbitrary number of fixed-length packets, thus it may have a variable length.

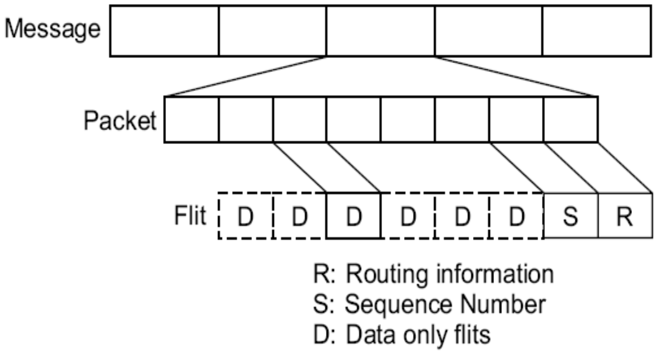


Fig.7.26 The format of message, packets, and flits (control flow digits) used as information units of communication in a message-passing network

A *packet* is the basic unit containing the destination address for routing purposes. Because different packets may arrive at the destination asynchronously, a sequence number is needed in each packet to allow reassembly of the message transmitted.

A packet can be further divided into a number of fixed-length *flits* (flow control digits). Routing information (destination) and sequence number occupy the header flits. The remaining flits are the data elements of a packet.

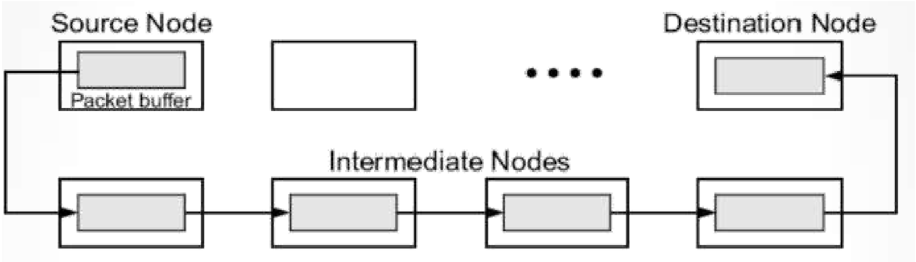
In multicomputers with store-and-forward routing, packets are the smallest unit of information transmission. In wormhole-routed networks, packets are further subdivided into flits. The flit length is often affected by the network size.

The packet length is determined by the routing scheme and network implementation. Typical packet lengths range from 64 to 512 bits. The sequence number may occupy one to two flits depending on the message length. Other factors affecting the choice of packet and flit sizes include channel bandwidth, router design, network traffic intensity, etc.

Two Message Passing mechanisms are:

1. Store and Forward Routing
2. Wormhole Routing

1. Store and Forward Routing

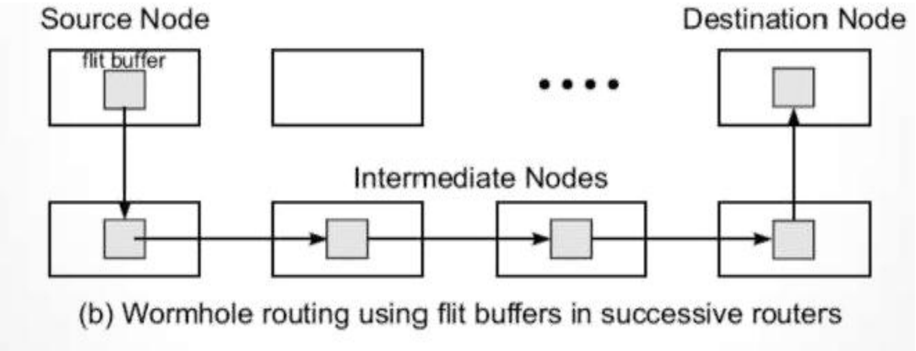


(a) Store-and-forward routing using packet buffers in successive nodes

- Packets are the basic unit of information flow in a store-and-forward network.
- Each node is required to use a packet buffer.

- A packet is transmitted from a source node to a destination node through a sequence of intermediate nodes.
- When a packet reaches an intermediate node, it is first stored in the buffer.
- Then it is forwarded to the next node if the desired output channel and a packet buffer in the receiving node are both available.

2. Wormhole Routing



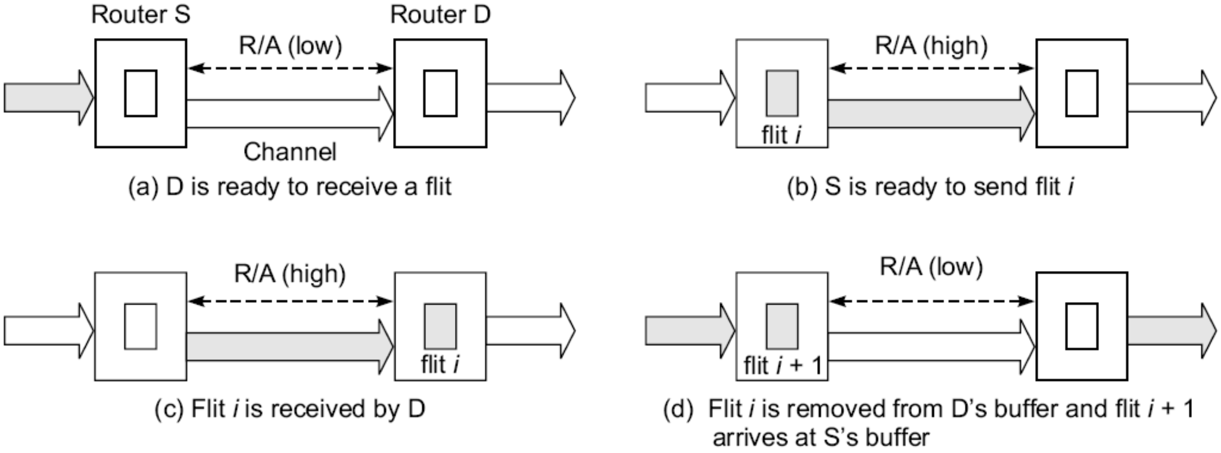
(b) Wormhole routing using flit buffers in successive routers

- Packets are subdivided into smaller flits. Flit buffers are used in the hardware routers attached to nodes.
- The transmission from the source node to the destination node is done through a sequence of routers.
- All the flits in the same packet are transmitted in order as inseparable companions in a pipelined fashion.
- Only the header flit knows where the packet is going.
- All the data flits must follow the header flit.
- Flits from different packets cannot be mixed up. Otherwise they may be towed to the wrong destination.

Asynchronous Pipelining

- The pipelining of successive flits in a packet is done asynchronously using a handshaking protocol as shown in Fig. 7.28. Along the path, a 1-bit ready/request (R/A) line is used between adjacent routers.

- When the receiving router (D) is ready (7.28a) to receive a flit (ie., a flit buffer is available), it pulls the R/A line low. When the sending router (S) is ready (Fig. 2.8b), it raises the line high and transmits flit I through the channel.
- While the flit is being received by D (Fig. 7.28c), the R/A line is kept high. After flit I is removed from D's buffer (ie., transmitted to the next node) (Fig. 7.28d), the cycle repeats itself for the transmission of the next flit i+1 until the entire packet is transmitted.



(a) D is ready to receive a flit

(b) S is ready to send flit *i*

(c) Flit *i* is received by D

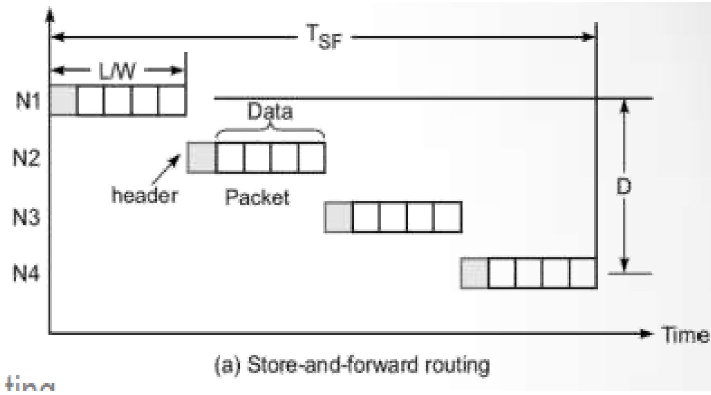
(d) Flit *i* is removed from D's buffer and flit *i* + 1 arrives at S's buffer

Fig. 7.28 Handshaking protocol between two wormhole routers (Courtesy of Lionel Ni, 1991)

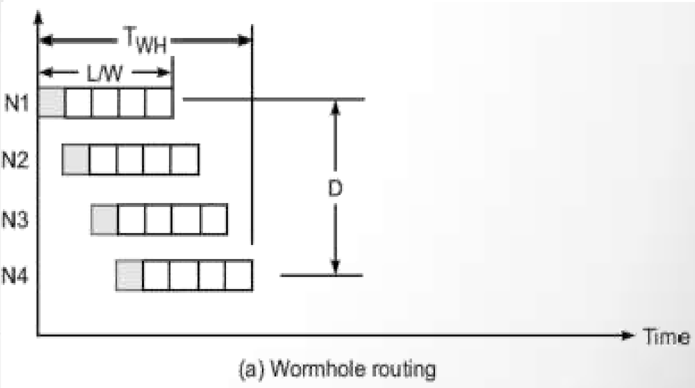
Advantages:

- Very efficient
- Faster clock

__**Latency Analysis:**__



(a) Store-and-forward routing



(a) Wormhole routing

- The communication latency in store-and-forward networks is directly proportional to the distance (the number of hops) between the source and the destination.

  $\mathbf{T_{SF} = L\ (D + 1)\ /\ W}$

- Wormhole Routing has a latency almost independent of the distance between the source and the destination

  $\mathbf{T_{WH} = L\ /\ W + F\ D\ /\ W}$

where, L: Packet length (in bits)
W: Channel Bandwidth (in bits per second)
D: Distance (number of nodes traversed minus 1)
F: Flit length (in bits)

7.4.2 Deadlock and Virtual channels

The communication channels between nodes in a wormhole-routed multicomputer network are actually shared by many possible source and destination pairs. The sharing of a physical channel leads to the concept of virtual channels.

Virtual channels

- A virtual channel is logical link between two nodes. It is formed by a flit buffer in the source node, a physical channel between them and a flit buffer in the receiver node.
- Four flit buffers are used at the source node and receiver node respectively. One source buffer is paired with one receiver buffer to form a virtual channel when the physical channel is allocated for the pair.
- Thus the physical channel is time shared by all the virtual channels. By adding the virtual channel the channel dependence graph can be modified and one can break the deadlock cycle.
- Here the cycle can be converted to spiral thus avoiding a deadlock. Virtual channel can be implemented with either unidirectional channel or bidirectional channels.
- However a special arbitration line is needed between adjacent nodes interconnected by bidirectional channel. This line determines the direction of information flow.
- The virtual channel may reduce the effective channel bandwidth available to each request.

- There exists a tradeoff between network throughput and communication latency in determining the degree of using virtual channels.

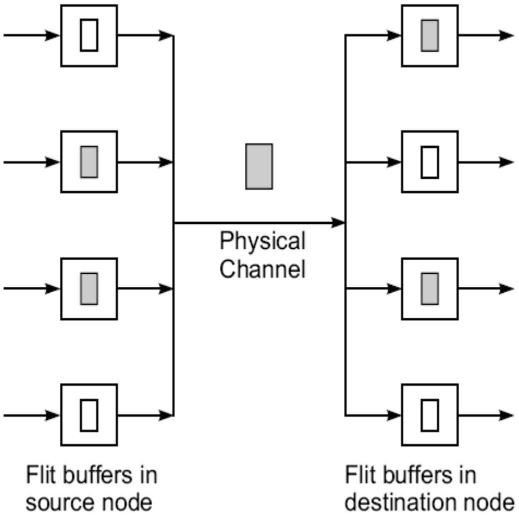


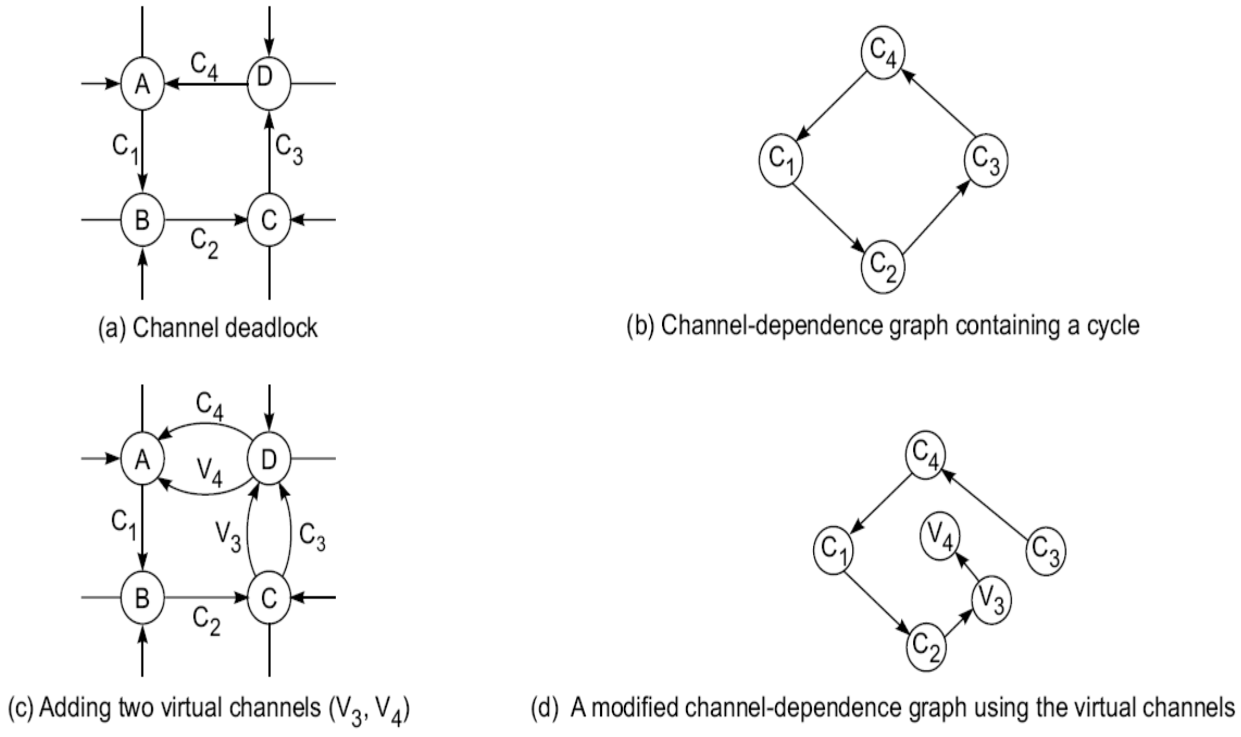
Fig. 7.30 Four virtual channels sharing a physical channel with time multiplexing on a flit-by-flit basis

Deadlock Avoidance

By adding two virtual channels, V_3 and V_4 in Fig. 7.32c, one can break the deadlock cycle. A modified channel-dependence graph is obtained by using the virtual channels V_3 and V_4, after the use of channel C_2, instead of reusing C_3 and C_4.

The cycle in Fig. 7.32b is being converted to a spiral, thus avoiding a deadlock. Channel multiplexing can be done at the flit level or at the packet level if the packet length is sufficiently short.

Virtual channels can be implemented with either unidirectional channels or bidirectional channels.



(a) Channel deadlock

(b) Channel-dependence graph containing a cycle

(c) Adding two virtual channels (V_3, V_4)

(d) A modified channel-dependence graph using the virtual channels

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

Chapter-8 Multivector and SIMD Computers

8.1 Vector Processing Principles

Vector Processing Definitions

Vector: A vector is a set of scalar data items, all of the same type, stored in memory. Usually, the vector elements are ordered to have a fixed addressing increment between successive elements called the stride.

Vector Processor: A vector processor is an ensemble of hardware resources, including vector registers, functional pipelines, processing elements, and register counters, for performing vector operations.

Vector Processing: Vector processing occurs when arithmetic or logical operations are applied to vectors. It is distinguished from scalar processing which operates on one or one pair of data.
Vector processing is faster and more efficient than scalar processing.

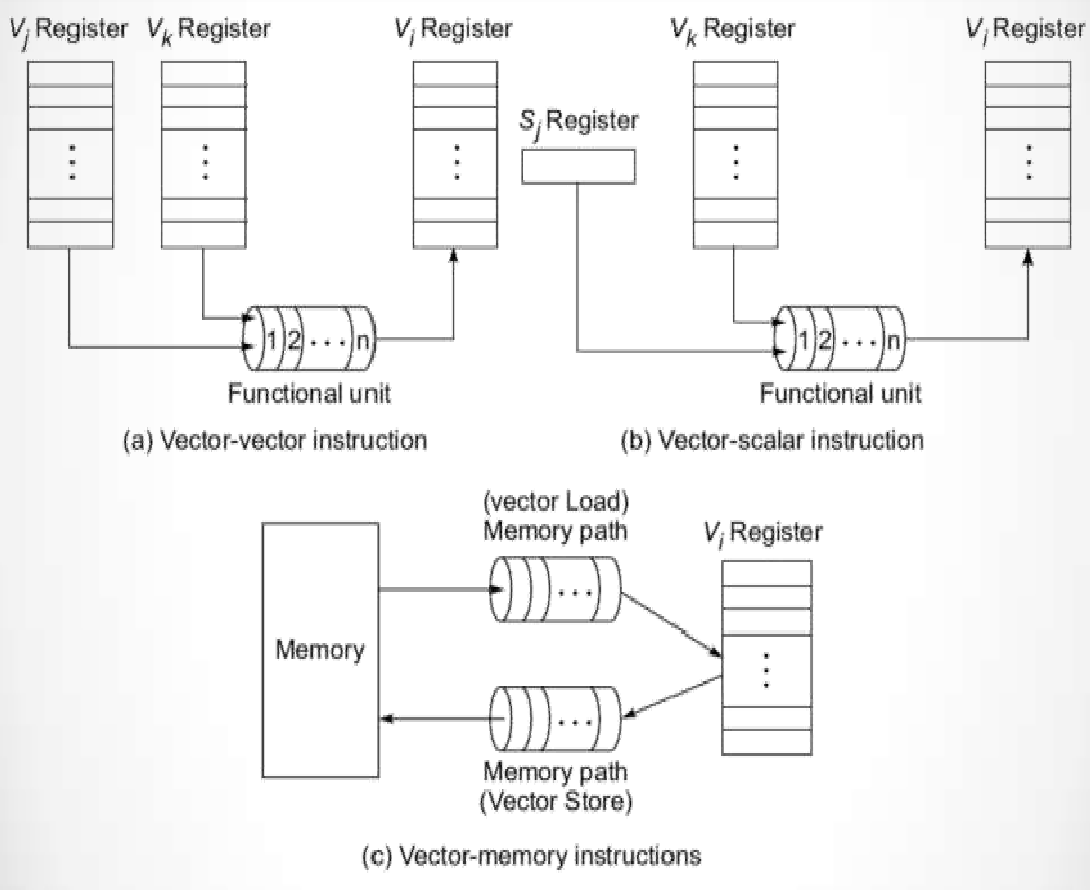
Vectorization: The conversion from scalar code to vector code is called vectorization.

Vectorizing Compiler: A compiler capable of vectorization is called a Vectorizing Compiler (vectorizer).

8.1.1 Vector Instruction Types

There are six types of vector instructions. These are defined by mathematical mappings between their working registers or memory where vector operands are stored.

1. Vector - Vector instructions
2. Vector - Scalar instructions
3. Vector - Memory instructions
4. Vector reduction instructions
5. Gather and scatter instructions
6. Masking instructions



(a) Vector-vector instruction

(b) Vector-scalar instruction

(c) Vector-memory instructions

Fig. 8.1 Vector instruction types in Cray-like computers

1. **Vector - Vector instructions:** One or two vector operands are fetched form the respective vector registers, enter through a functional pipeline unit, and produce result in another vector register.

   F1: Vi → Vj

   F2: Vi x Vj →Vk

Examples: V1 = sin(V2), V3 = V1+ V2

2. **Vector - Scalar instructions**

   Elements of vector register are multiplied by a scalar value.

   F3: s x Vi → Vj

   Examples: V2 = 6 + V1

3. **Vector - Memory instructions:** This corresponds to Store-load of vector registers (V) and the Memory (M).

   F4: M → V (Vector Load)

   F5: V → M (Vector Store)

   Examples: X = V1 V2 = Y

4. **Vector reduction instructions:** include maximum, minimum, sum, mean value.

   F6: Vi → s

   F7: Vi x Vj → s

5. **Gather and scatter instructions** Two instruction registers are used to gather or scatter vector elements randomly throughout the memory corresponding to the following mappings

   F8: M → Vi x Vj (Gather)

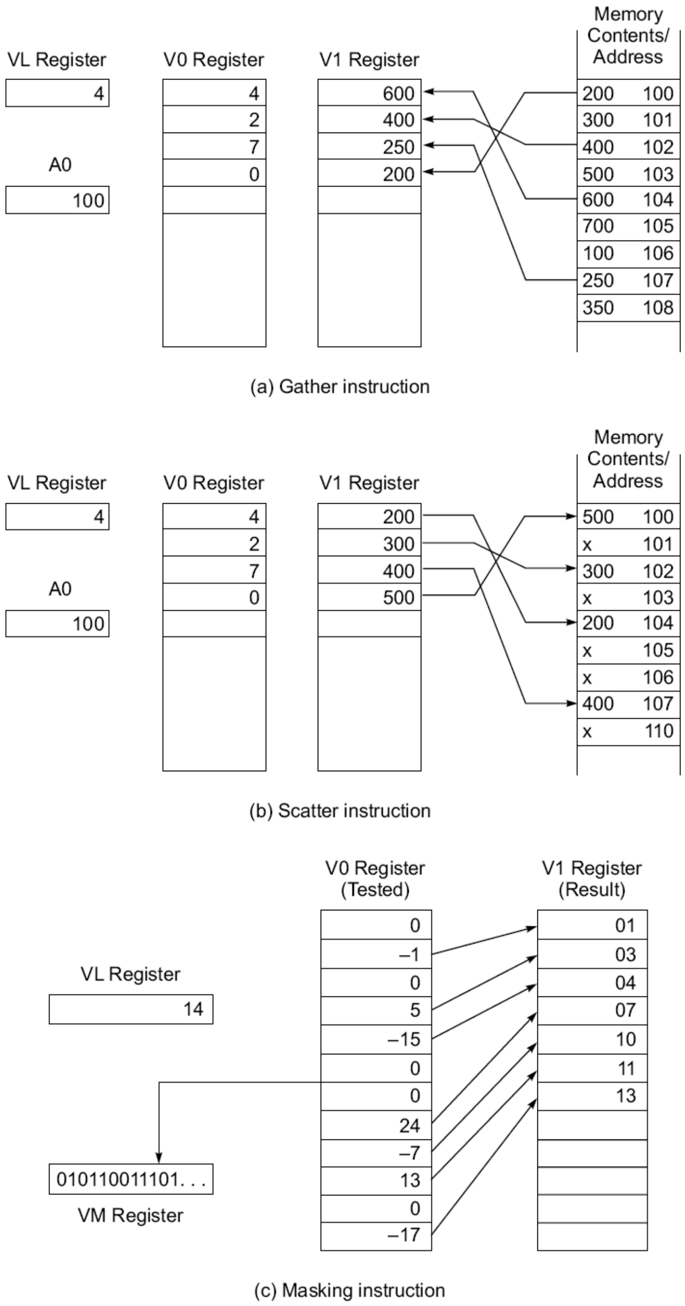
   F9: Vi x Vj → M (Scatter)

   Gather is an operation that fetches from memory the nonzero elements of a sparse vector using indices.

   Scatter does the opposite, storing into memory a vector in a sparse vector whose nonzero entries are indexed.

6. **Masking instructions** The Mask vector is used to compress or to expand a vector to a shorter or longer index vector (bit per index correspondence).

F10: Vi x Vm → Vj (Vm is a ***binary vector***)



(a) Gather instruction

(b) Scatter instruction

(c) Masking instruction

Fig. 8.2 Gather, scatter and masking operations on the Cray Y-MP (Courtesy of Cray Research, 1990)

- The *gather*, *scatter*, and *masking* instructions are very useful in handling sparse vectors or sparse matrices often encountered in practical vector processing applications.
- Sparse matrices are those in which most of the entries arc zeros.
- Advanced vector processors implement these instructions directly in hardware.

8.1.2 Vector-Access Memory Schemes

The flow of vector operands between the main memory and vector registers is usually pipelined with multiple access paths.

Vector Operand Specifications

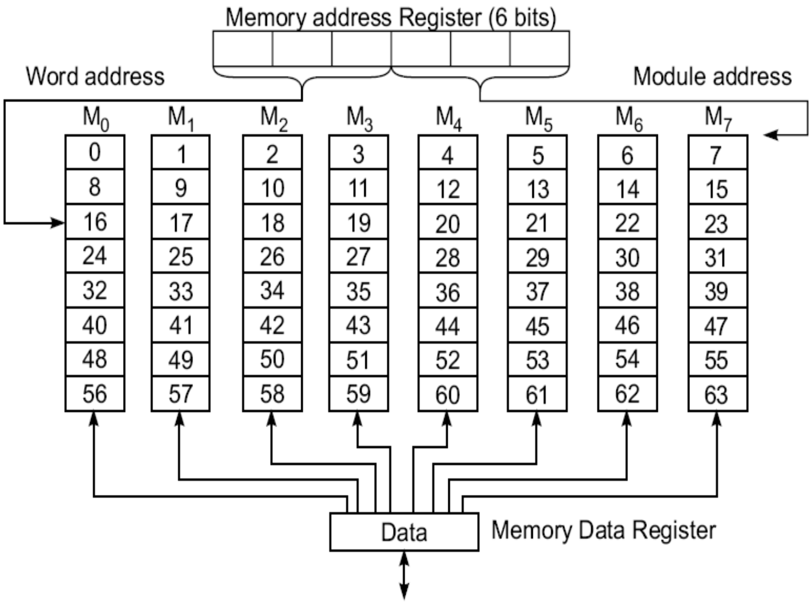
- Vector operands may have arbitrary length.
- Vector elements are not necessarily stored in contiguous memory locations.
- To access a vector a memory, one must specify its base, stride, and length.
- Since each vector register has fixed length, only a segment of the vector can be loaded into a vector register.
- Vector operands should be stored in memory to allow pipelined and parallel access. Access itself should be pipelined.

Three types of Vector-access memory organization schemes

1. C-Access memory organization

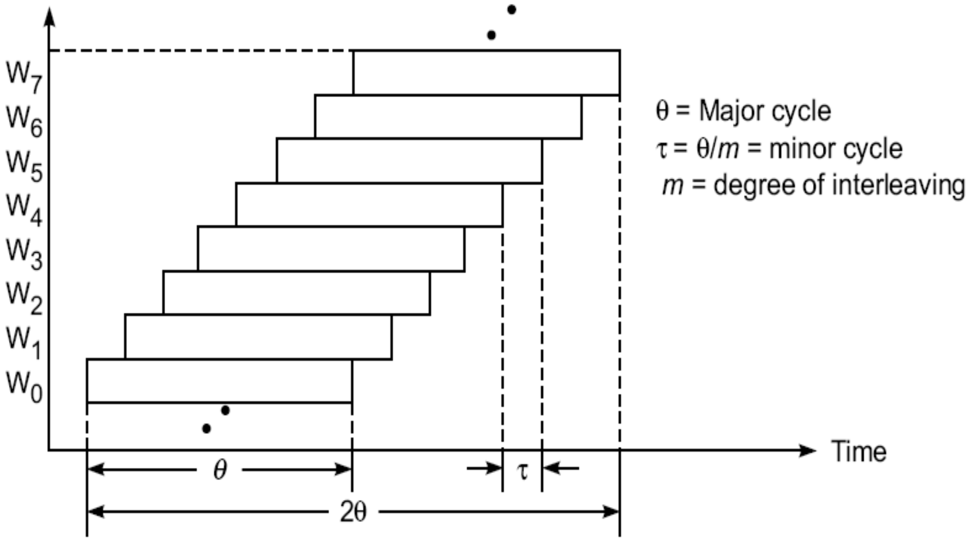
The m-way low-order memory structure, allows **m** words to be accessed concurrently and overlapped.

The access cycles in different memory modules are staggered. The low-order **a** bits select the modules, and the high-order **b** bits select the word within each module, where **m=2^a** and **a+b = n** is the address length.



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)



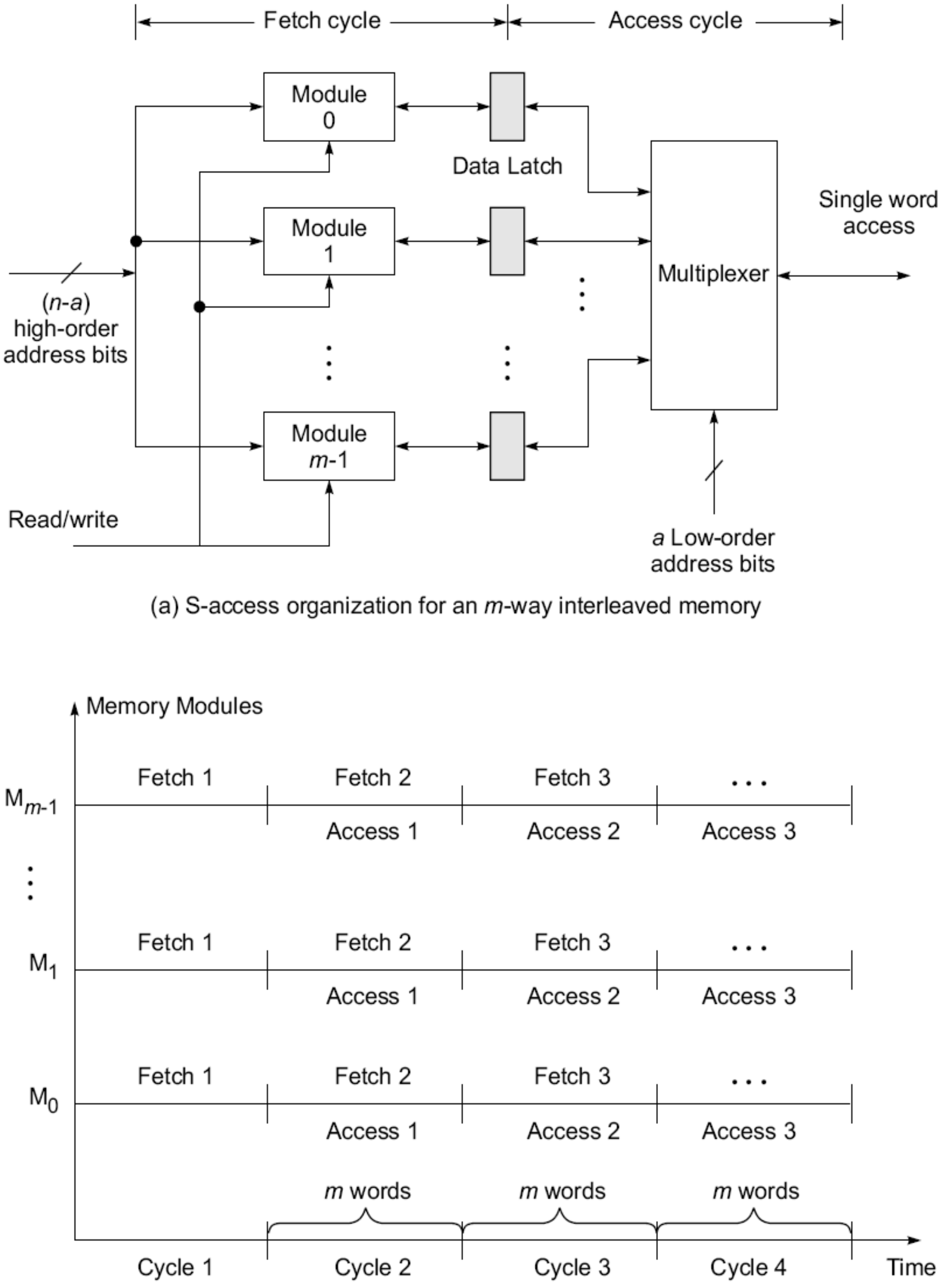
(b) Pipelined access of eight consecutive words in a C-access memory

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

- To access a vector with a stride of 1, successive addresses are latched in the address buffer at the rate of one per cycle.
- Effectively it takes m **minor** cycles to fetch m words, which equals one **(major) memory** cycle as stated in Fig. 5.16b.
- If the stride is 2, the successive accesses must be separated by two minor cycles in order to avoid access conflicts. This reduces the memory throughput by one-half.
- If the stride is 3, there is no module conflict and the maximum throughput (m words) results.
- In general, C-access will yield the maximum throughput of m words per memory cycle if the stride is relatively prime to m, the number of interleaved memory modules.

2. S-Access memory organization

All memory modules are accessed simultaneously in a synchronized manner. The high order **(n-a)** bits select the same offset word from each module.



(a) S-access organization for an *m*-way interleaved memory

(b) Successive vector accesses using overlapped fetch and access cycles

Fig. 8.3 The S-access interleaved memory for vector operands access

At the end of each memory cycle (Fig. 8.3b), $\mathbf{m = 2^a}$ consecutive words are latched. If the stride is greater than 1, then the throughput decreases, roughly proportionally to the stride.

3. C/S-Access memory organization

- Here C-access and S-access are combined.
- **n** access buses are used with **m** interleaved memory modules attached to each bus.

- The **m** modules on each bus are **m-way** interleaved to allow C-access.
- In each memory cycle, at most **m.n** words are fetched if the **n** buses are fully used with pipelined memory accesses

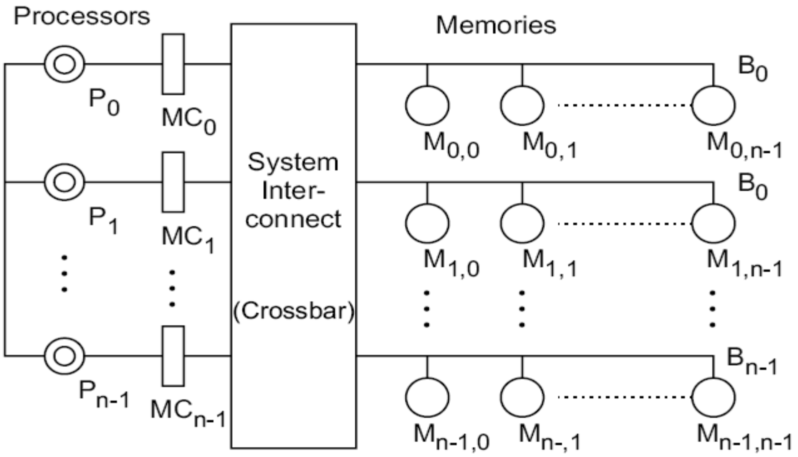


Fig. 8.4 The C/S memory organization with $m = n$. (Courtesy of D.K. Panda, 1990)

- The C/S-access memory is suitable for use in vector multiprocessor configurations.
- It provides parallel pipelined access of a vector data set with high bandwidth.
- A special vector cache design is needed within each processor in order to guarantee smooth data movement between the memory and multiple vector processors.

8.3 Compound Vector Processing

A compound vector function (CVF) is defined as a composite function of vector operations converted from a looping structure of linked scalar operations.

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

where X(I) and Y(I), I=1, 2,…. N, are two source vectors originally residing in the memory. After the computation, the resulting vector is stored back to the memory. S is an immediate constant supplied to the multiply instruction.

After vectorization, the above scalar SAXPY code is converted to a sequence of five vector instructions:

M(x : x + N-1) → V1	Vector Load
M(y : y + N-1) → V2	Vector Load
S X V1 → V1	Vector Multiply
V2 X V1 → V2	Vector Add
V2 → M(y : y + N-1)	Vector Store

X and y are starting memory addresses of the X and Y vectors, respectively; V1 and V2 are two N-element vector registers in the vector processor.

CVF: Y(1:N) = S X(1:N) + Y(1:N) or Y(I) = S X(I) + Y(I)

where Index I implies that all vector operations involve N elements.

- Typical CVF for one-dimensional arrays are load, store, multiply, divide, logical and shifting operations.
- The number of available vector registers and functional pipelines impose some restrictions on how many CVFs can be executed simultaneously.
- **Chaining**:

  Chaining is an extension of technique of internal data forwarding practiced in scalar processors. Chaining is limited by the small number of functional pipelines available in a vector processor.
- **Strip-mining:**

  When a vector has a length greater than that of the vector registers, segmentation of the long vector into fixed-length segments is necessary. One vector segment is processed at a time (in Cray computers segment is 64 elements).
- **Recurrence**:

  The special case of vector loops in which the output of a functional pipeline may feed back into one of its own source vector registers

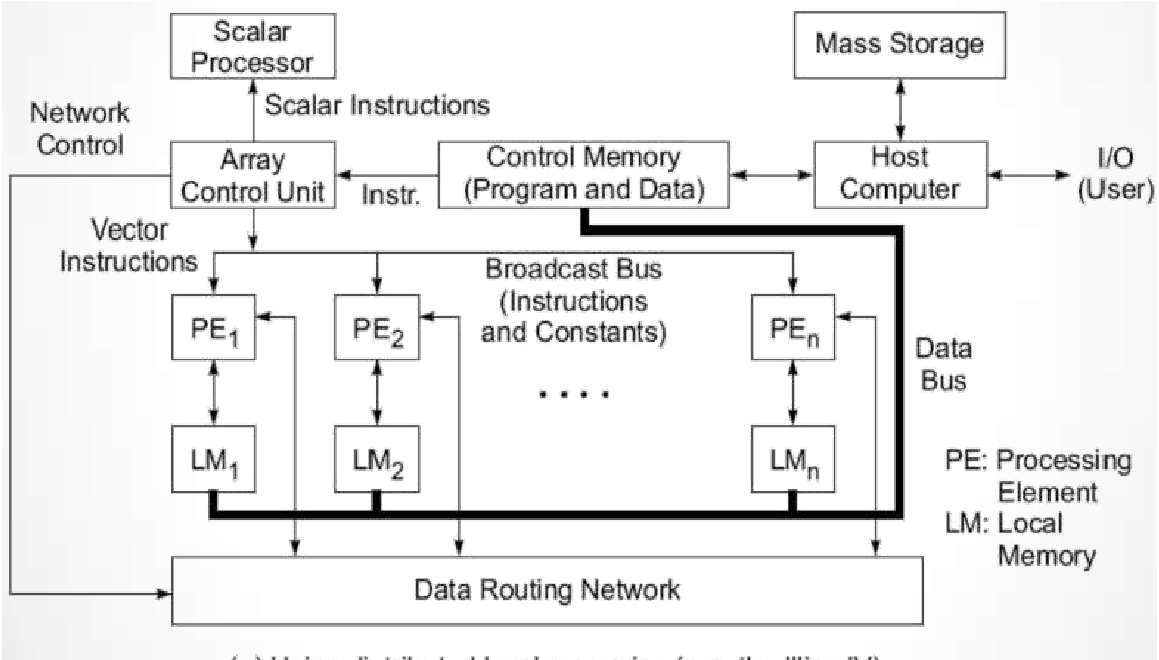
8.4 SIMD Computer Organizations

SIMD Implementation Models OR (Two models for constructing SIMD Super Computers)

SIMD models differentiates on base of memory distribution and addressing scheme used.

Most SIMD computers use a single control unit and distributed memories, except for a few that use associative memories.

1. Distributed memory model



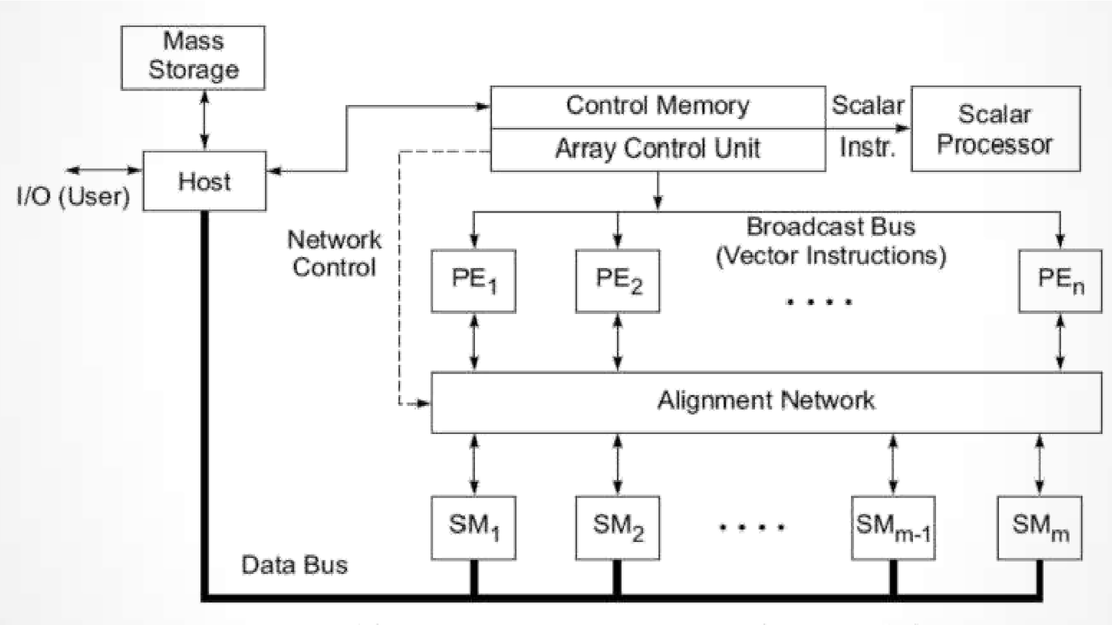
(a) Using distributed local memories (e.g. the Illiac IV)

- Spatial parallelism is exploited among the PEs.
- A distributed memory SIMD consists of an array of PEs (supplied with local memory) which are controlled by the array control unit.
- Program and data are loaded into the control memory through the host computer and distributed from there to PEs local memories.
- An instruction is sent to the control unit for decoding. If it is a scalar or program control operation, it will be directly executed by a scalar processor attached to the control unit.
- If the decoded instruction is a vector operation, it will be broadcast to all the PEs for parallel execution.
- Partitioned data sets are distributed to all the local memories attached to the PEs trough a vector data bus.

- PEs are interconnected by a data routing network which performs inter-PE data communications such as shifting, permutation and other routing operations.

2. Shared Memory Model

- An alignment network is used as the inter-PE memory communication network. This network is controlled by control unit.
- The alignment network must be properly set to avoid access conflicts.
- Figure below shows a variation of the SIMD computer using shared memory among the PEs.
- Most SIMD computers were built with distributed memories.



(b) Using shared-memory modules (e.g. the BSP)

8.4.2 CM-2 Architecture

The Connection Machine CM-2 produced by Thinking Machines Corporation was a fine-grain MPP computer using thousands of bit-slice PEs in parallel to achieve a peak processing speed of above 10 Gflops.

Program Execution Paradigm

All programs started execution on a front-end, which issued microinstructions to the back-end processing array when data-parallel operations were desired. The sequencer broke down these microinstructions and broadcast them to all data processors in the array.

Data sets and results could be exchanged between the front-end and the processing array in one of three ways as shown in the figure:

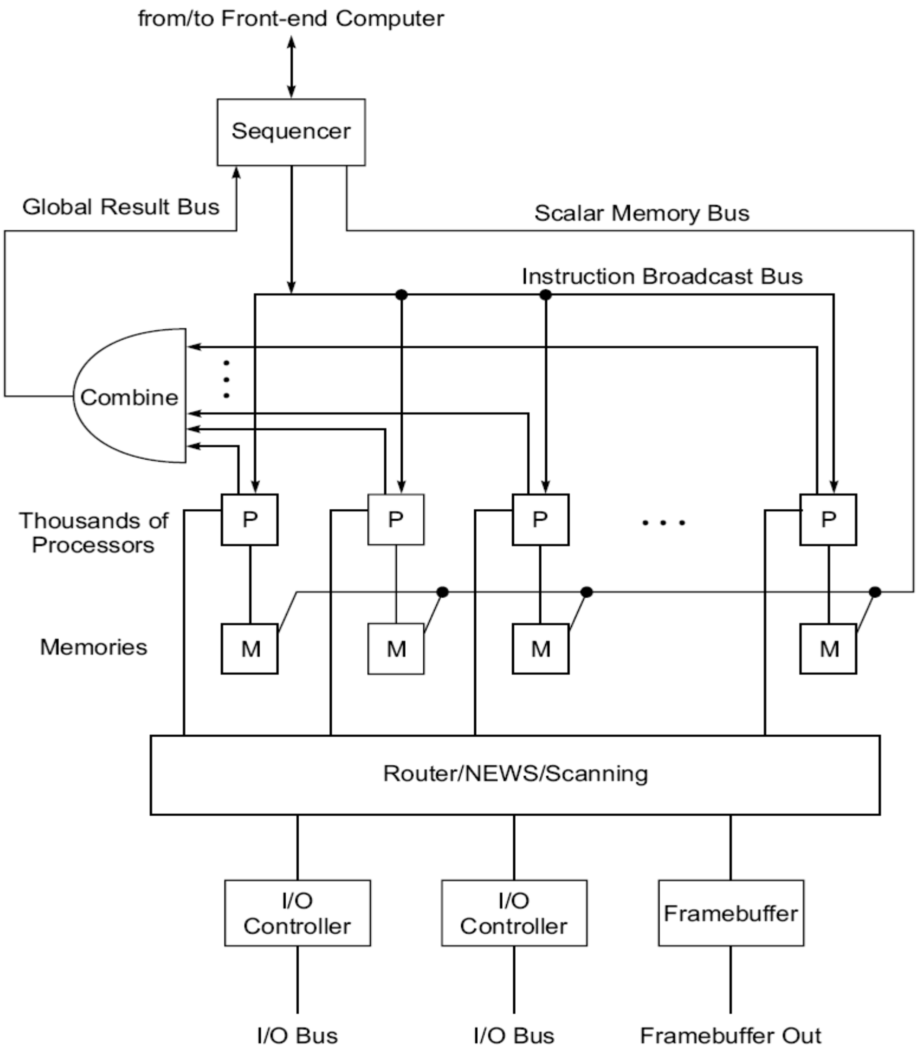


Fig. 8.23 The architecture of the Connection Machine CM-2 (Courtesy of Thinking Machines Corporation, 1990)

- **Broadcasting:** Broadcasting was carried out through the broadcast bus to all data processors at once.
- **Global combining:** Global combining allowed the front-end to obtain the sum, largest value, logical OR etc, of values one from each processor.
- **Scalar memory bus:** Scalar bus allowed the front-end to read or to write one 32-bit value at a time from or to the memories attached to the data processors.

Processing Array

The processing array contained from 4K to 64K bit-slice data processors(PEs), all of which were controlled by a sequencer.

Processing Nodes

Each data processing node contained 32 bit-slice data processors, an optional floating point accelerator and interfaces for inter processor communication.

Hypercube Routers

The router nodes on all processor chips were wired together to frm a Boolean n-cube. A full configuration of CM-2 had 4096 router nodes on processor chips interconnected as a 12-dimensional hypercube.

Major Applications of CM-2

The CM-2 has been applied in almost all the MPP and grand challenge applications.

- Used in document retrieval using relevance feedback,
- in memory based reasoning as in the medical diagnostic system called QUACK for simulating the

  diagnosis of a disease,
- in bulk processing of natural languages.
- the SPICE-like VLSI circuit analysis and layout,
- computational fluid dynamics,
- signal/image/vision processing and integration,
- neural network simulation and connectionist modeling,
- dynamic programming,
- context free parsing,
- ray tracing graphics,
- computational geometry problems.

8.4.3 MasPar MP-1 Architecture

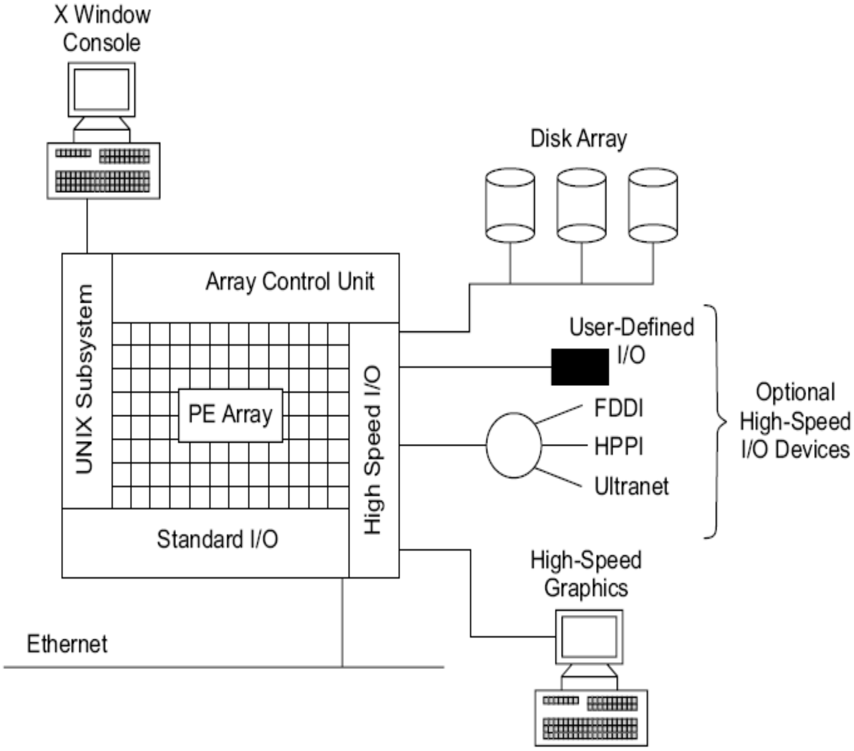
The MP-1 architecture consists of four subsystems:

i) PE array,

ii) Array Control Unit (ACU),

iii) UNIX subsystem with standard I/O,

iv) High-speed I/O subsystem



(a) MP-1 System Block Diagram

- **The UNIX subsystem** handles traditional serial processing.
- **The high-speed I/O**, working together with the **PE array**, handles massively parallel computing.
- The MP-1 family includes configurations with 1024, 4096. and up to 16,384 processors. The peak performance of the 16K-processor configuration is 26,000 MIPS in 32-bit RISC integer operations. The system also has a peak floating-point capability of 1.5 Gfiops in single-precision and 650 Mflops in double-precision operations.
- **Array Control Unit** The ACU is a 14-MIPS scalar RISC processor using a demand paging instruction memory. The ACU fetches and decodes MP-1 instructions, computes addresses and scalar data values, issues control signals to the PE array, and monitors the status of the PE array.

The ACU is microcoded to achieve horizontal control of the PE array. Most scalar ACU instructions execute in one 70-ns clock. The whole ACU is implemented on one PC board.

An implemented functional unit, called a *memory machine*, is used in parallel with the ACU. The memory machine performs PE array load and store operations, while the ACU broadcasts arithmetic, logic, and routing instructions to the PEs for parallel execution.

Chapter 9- Scalable, Multithreaded, and Dataflow Architectures

9.1 Latency Hiding Techniques.

9.1.1 Shared Virtual Memory

Single-address-space multiprocessors/multicomputers must use shared virtual memory.

The Architecture Environment

- The Dash architecture was a large-scale, cache-coherent, NUMA multiprocessor system as depicted in Fig. 9.1.

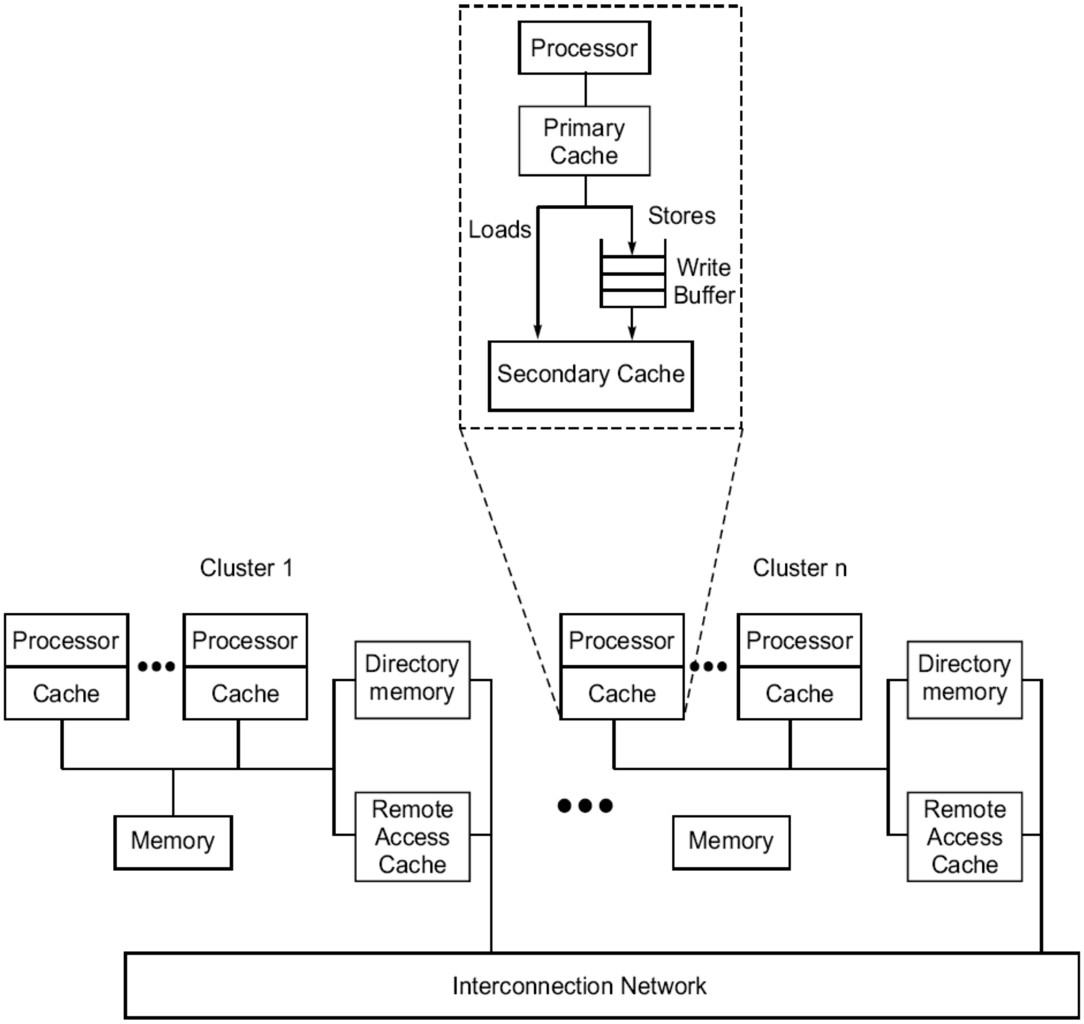


Fig. 9.1 A scalable coherent cache multiprocessor with distributed shared memory modeled after the Stanford Dash (Courtesy of Anoop Gupta et al, *Proc. 1991 Ann. Int. Symp. Computer Arch.*)

- It consisted of multiple multiprocessor clusters connected through a scalable, low latency interconnection network.

- Physical memory was distributed among the processing nodes in various clusters. The distributed memory formed a global address space.
- Cache coherence was maintained using an invalidating, distributed directory-based protocol. For each memory block, the directory kept track of remote nodes caching it.
- When a write occurred, point-to-point messages were sent to invalidate remote copies of the block.
- Acknowledgement messages were used to inform the originating node when an invalidation was completed.

- Two levels of local cache were used per processing node. Loads and writes were separated with the use of write buffers for implementing weaker memory consistency models.
- The main memory was shared by all processing nodes in the same cluster. To facilitate prefetching and the directory-based coherence protocol, directory memory and remote-access caches were used for each cluster.
- The remote-access cache was shared by all processors in the same cluster.

The SVM Concept

- Figure 9.2 shows the structure of a distributed shared memory. A global virtual address space is shared among processors residing at a large number of loosely coupled processing nodes.
- The idea of Shared virtual memory (SVM) is to implement coherent shared memory on a network of processors without physically shared memory.
- The coherent mapping of SVM on a message-passing multicomputer architecture is shown in Fig. 9.2b.

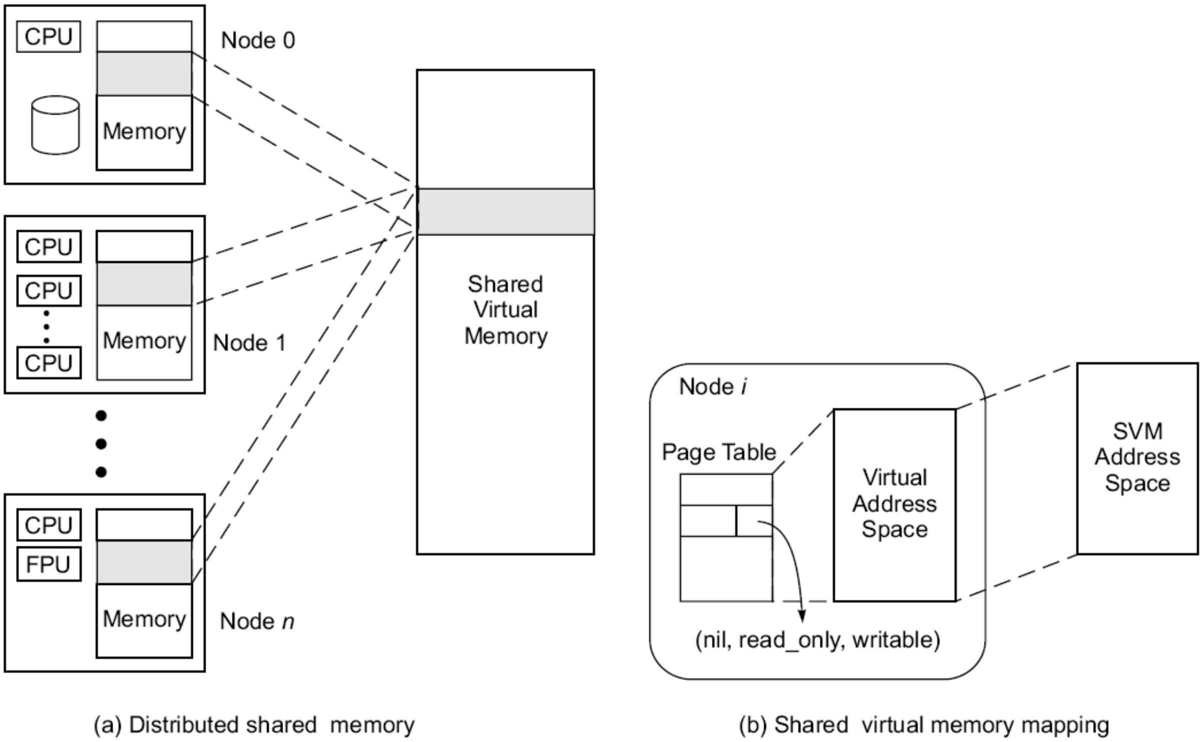
- The system uses virtual addresses instead of physical addresses for memory references.

- Each virtual address space can be as large as a single node can provide and is shared by all nodes in the system.
- The SVM address space is organized in pages which can be accessed by any node in the system. A memory-mapping manager on each node views its local memory as a large cache of pages for its associated processor.

Page Swapping

- A memory reference causes a page fault when the page containing the memory location is not in a processor’s local memory.

- When a page fault occurs, the memory manager retrieves the missing page from the memory of another processor.
- If there is a page frame available on the receiving node, the page is moved in.
- Otherwise, the SVM system uses page replacement policies to find an available page frame, swapping its contents to the sending node.
- A hardware MMU can set the access rights (nil, read-only} writable) so that a memory access violating memory coherence will cause a page fault.
- The memory coherence problem is solved in IVY through distributed fault handlers and their servers. To client programs, this mechanism is completely transparent.
- The large virtual address space allows programs to be larger in code and data space than the physical memory on a single node.
- This SVM approach offers the ease of shared-variable programming in a message-passing environment.
- In addition, it improves software portability and enhances system scalability through modular memory growth.



(a) Distributed shared memory

(b) Shared virtual memory mapping

Fig. 9.2 The concept of distributed shared memory with a global virtual address space shared among all processors on loosely coupled processing nodes in a massively parallel architecture (Courtesy of Kai Li, 1992)

Latency hiding can be accomplished through 4 complementary approaches:

i) **Prefetching techniques** which bring instructions or data close to the processor before they are actually needed

ii) **Coherent caches** supported by hardware to reduce cache misses

iii) **Relaxed memory consistency models** by allowing buffering and pipelining of memory references

iv) **Multiple-contexts support** to allow a processor to switch from one context to another when a long latency operation is encountered.

1. Prefetching Techniques

Prefetching uses knowlwdge about the expected misses in a program to move the corresponding data close to the processor before it is actually needed.

Prefetching can be classified based on whether it is

- Binding
- Non binding

or whether it is controlled by

- hardware
- software

- **Binding prefetching** : the value of a later reference (eg, a register load) is bound at the time when the prefetch completes.
- **Non binding prefetching** : brings data close to the processor, but the data remains visible to the cache coherence protocol and is thus kept consistent until the processor actually reads the value.
- **Hardware Controlled Prefetching**: includes schemes such as long cache lines and instruction lookahead.
- **Software Controlled Prefetching**: explicit prefetch instructions are issued. Allows the prefetching to be done selectively and extends the possible interval between prefetch issue and actual reference.

2. Coherent Caches

- While the cache coherence problem is easily solved for small bus-based multiprocessors through the use of snoopy cache coherence protocols, the problem is much more complicated for large scale multiprocessors that use general interconnection networks.

- As a result, some large scale multiprocessors did not provide caches, others provided caches that must be kept coherent by software, and still others provided full hardware support for coherent caches.
- Caching of shared read-write data provided substantial gains in performance. The largest benefit came from a reduction of cycles wasted due to read misses. The cycles wasted due to write misses were also reduced.
- Hardware cache coherence is an effective technique for substantially increasing the performance with no assistance from the compiler or programmer.

3. Relaxed memory consistency models

Some different consistency models can be defined by relaxing one or more requirements in sequential consistency called relaxed consistency models. These consistency models do not provide memory consistency at the hardware level. In fact, the programmers are responsible for implementing the memory consistency by applying synchronization techniques.

There are 4 comparisons to define the relaxed consistency:

- Relaxation
- Synchronizing vs non-synchronizing
- Issue vs View-Based
- Relative Model Strength

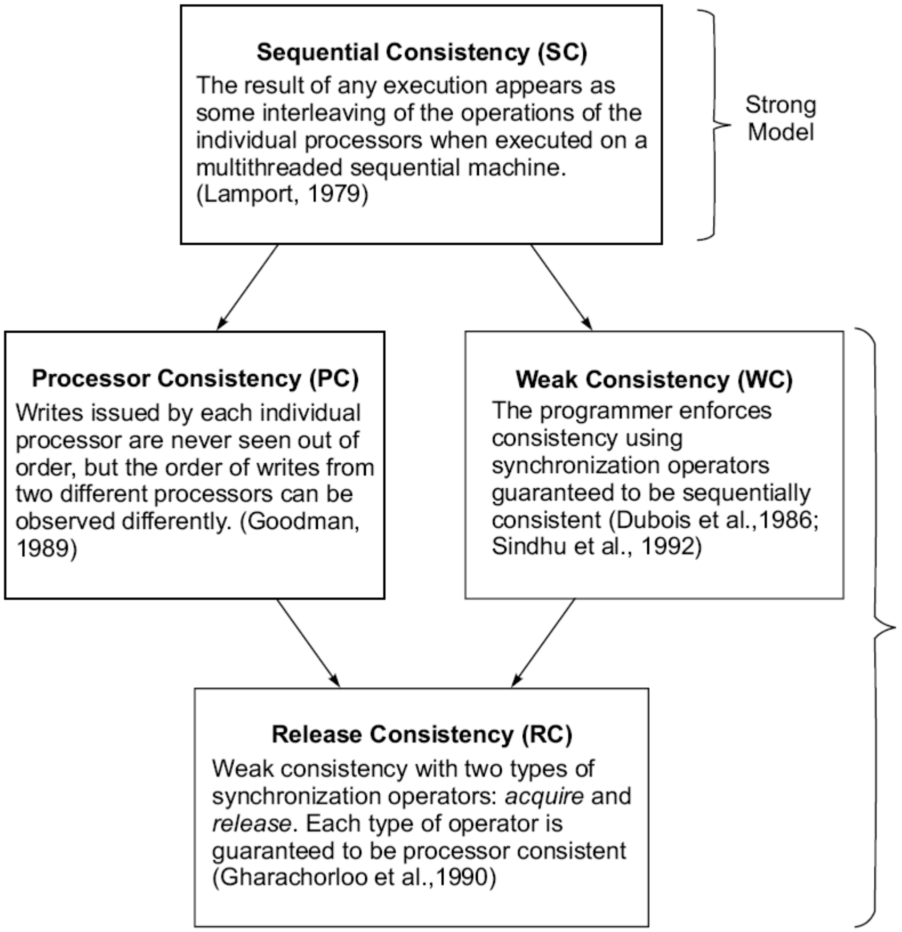

Sequential Consistency (SC)
The result of any execution appears as some interleaving of the operations of the individual processors when executed on a multithreaded sequential machine. (Lamport, 1979)
Strong Model
Processor Consistency (PC)
Writes issued by each individual processor are never seen out of order, but the order of writes from two different processors can be observed differently. (Goodman, 1989)
Weak Consistency (WC)
The programmer enforces consistency using synchronization operators guaranteed to be sequentially consistent (Dubois et al.,1986; Sindhu et al., 1992)
Relaxed Models
Release Consistency (RC)
Weak consistency with two types of synchronization operators: *acquire* and *release*. Each type of operator is guaranteed to be processor consistent (Gharachorloo et al.,1990)

Fig. 9.8 Intuitive definitions of four memory consistency models. The arrows point from strong to relaxed consistencies (Courtesy of Nitzberg and Lo, *IEEE Computer,* August 1991)

9.2 Principles of Multithreading

9.2.1 Multithreading Issues and Solutions

Multithreading demands that the processor be designed to handle multiple contexts simultaneously on a context-switching basis.

Architecture Environment

Multithreading MPP system is modeled by a network of Processor (P) and memory (M) nodes as shown in Fig. 9.11a. The distributed memories form a global address space.

Four machine parameters are defined below to analyze the performance of this network:

1. **The Latency (L):** This is the communication latency on a remote memory access. The value of L includes the network delays, cache-miss penalty and delays caused by contentions in split transactions.

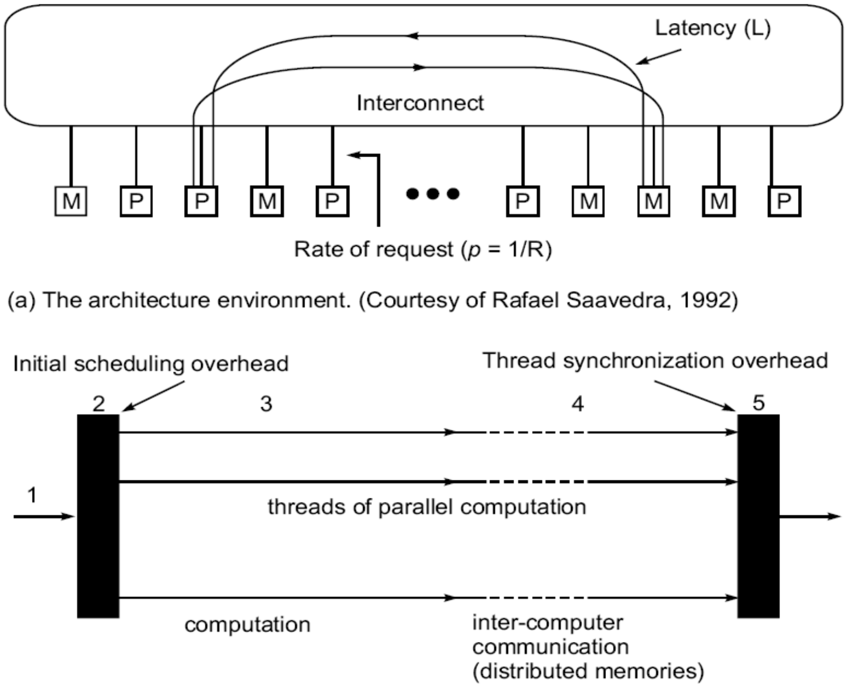
2. **The number of Threads (N):** This is the number of threads that can be interleaved in each processor. A thread is represented by a context consisting of a program counter, a register set and the required context status words.
3. **The context-switching overhead (C):** This refers to the cycles lost in performing context switching in a processor. This time depends on the switch mechanism and the amount of processor states devoted to maintaining active threads.
4. **The interval between switches (R):** This refers to the cycles between switches triggered by remote reference. The inverse p=1/R is called the rate of requests for remote accesses. This reflects a combination of program behavior and memory system design.

In order to increase efficiency, one approach is to reduce the rate of requests by using distributed coherent caches. Another is to eliminate processor waiting through multithreading.

Multithreaded Computations

Fig 9.11b shows the structure of the multithreaded parallel computations model.

The computation starts with a sequential thread (1), followed by supervisory scheduling (2), where the processors begin threads of computation (3), by intercomputer messages that update variables among the nodes when the computer has distributed memory (4), and finally by synchronization prior to beginning the next unit of parallel work (5).


Latency (L)
Interconnect
M
P
P
M
P
P
M
M
M
P
Rate of request (p = 1/R)
Initial scheduling overhead
Thread synchronization overhead
2
3
4
5
1
threads of parallel computation
computation
inter-computer communication (distributed memories)

(a) The architecture environment. (Courtesy of Rafael Saavedra, 1992)

(b) Multithreaded computation model. (Courtesy of Gordon Bell, *Commun. ACM*, August 1992)

Fig. 9.11 Multithreaded architecture and its computation model for a massively parallel processing system

The communication overhead period (4) inherent in distributed memory structures is usually distributed throughout the computation and is possibly completely overlapped.

Message passing overhead in multicomputers can be reduced by specialized hardware operating in parallel with computation.

Communication bandwidth limits granularity, since a certain amount of data has to be transferred with other nodes in order to complete a computational grain. Message passing calls (4) and synchronization (5) are nonproductive.

Fast mechanisms to reduce or to hide these delays are therefore needed. Multithreading is not capable of speedup in the execution of single threads, while weak ordering or relaxed consistency models are capable of doing this.

Problems of Asynchrony

Massively parallel processors operate asynchronously in a network environment. The asynchrony triggers two fundamental latency problems:

1. Remote loads
2. Synchronizing loads

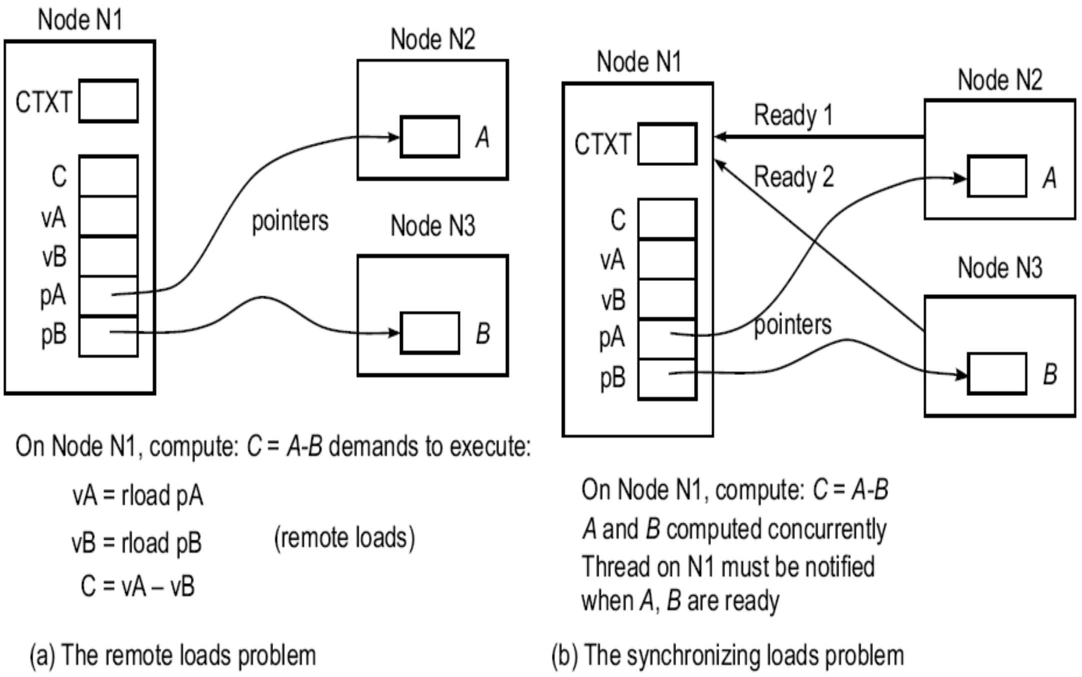
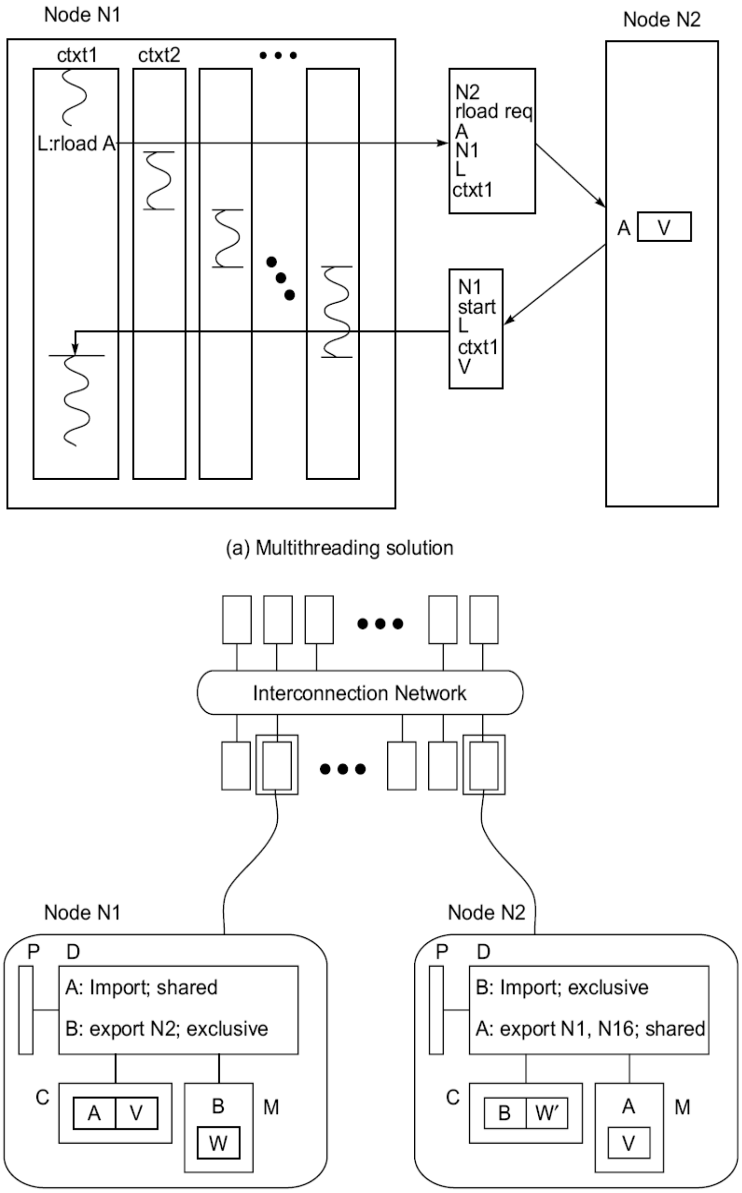

Node N1
CTXT
C
vA
vB
pA
pB
pointers
Node N2
A
Node N3
B
On Node N1, compute: C = A-B demands to execute:
vA = rload pA
vB = rload pB
C = vA – vB
(remote loads)
(a) The remote loads problem
Node N1
CTXT
C
vA
vB
pA
pB
Ready 1
Ready 2
pointers
Node N2
A
Node N3
B
On Node N1, compute: C = A-B
A and B computed concurrently
Thread on N1 must be notified
when A, B are ready
(b) The synchronizing loads problem

Fig. 9.12 Two common problems caused by asynchrony and communication latency in massively parallel processors (Courtesy of R.S. Nikhil, Digital Equipment Corporation, 1992)

Solutions to Asynchrony Problem

1. Multithreading Solutions
2. Distributed Caching



(a) Multithreading solution

P = Processor; D = Directory; C = Cache; M = Memory

(b) Distributed cacheing

Fig. 9.13 Two solutions for overcoming the asynchrony problems (Courtesy of R. S. Nikhil, Digital Equipment Corporation, 1992)

1. **Multithreading Solutions** – Multiplex among many threads

When one thread issues a remote-load request, the processor begins work on another thread, and so on (Fig. 9.13a).

- Clearly the cost of thread switching should be much smaller than that of the latency of the remote load, or else the processor might as well wait for the remote load's response.
- As the internode latency increases, more threads are needed to hide it effectively. Another concern is to make sure that messages carry continuations. Suppose, after issuing a remote load from thread T1 (Fig 9.13a), we switch to thread T2, which also issues a remote load.
- The responses may not return in the same order. This may be caused by requests traveling different distances, through varying degrees of congestion, to destination nodes whose loads differ greatly, etc.
- One way to cope with the problem is to associate each remote load and response with an identifier for the appropriate thread, so that it can be reenabled on the arrival of a response.

2. Distributed Caching

- The concept of Distributed Caching is shown in Fig. 9.13b. every memory location has an owner node. For example, N1 owns B and N2 owns A.
- The directories are used to contain import-export lists and state whether the data is shared (for reads, many caches may hold copies) or exclusive (for writes, one cache holds the current value).
- The directories multiplex among a small number of contexts to cover the cache loading effects.

- The Distributed Caching offers a solution for the remote-loads problem, but not for the synchronizing-loads problem.
- Multithreading offers a solution for remote loads and possibly for synchronizing loads.
- The two approaches can be combined to solve both types of remote access problems.

9.2.2 Multiple-Context Processors

Multithreaded systems are constructed with multiple-context (multithreaded) processors.

Enhanced Processor Model

- A conventional single-thread processor will wait during a remote reference, it is idle for a period of time L.
- A multithreaded processor, as modeled in Fig. 9.14a, will suspend the current context and switch to another, so after some fixed number of cycles it will again be busy doing useful work, even though the remote reference is outstanding.
- Only if all the contexts are suspended (blocked) will the processor be idle.

The objective is to maximize the fraction of time that the processor is busy, we will use the efficiency of the processor as our performance index, given by:

$$\text{Efficiency} = \frac{\text{busy}}{\text{busy} + \text{switching} + \text{idle}}$$

where busy, switching and idle represent the amount of time, measured over some large interval, that the processor is in the corresponding state.

The basic idea behind a multithreaded machine is to interleave the execution of several contexts on order to dramatically reduce the value of idle, but without overly increasing the magnitude of switching.

Context-Switching Policies

Different multithreaded architectures are distinguished by the context-switching policies adopted.

Four switching policies are:

1. **Switch on Cache miss –** This policy corresponds to the case where a context is preempted when it causes a cache miss.

   In this case, R is taken to be the average interval between misses (in Cycles) and L the time required to satisfy the miss.

   Here, the processor switches contexts only when it is certain that the current one will be delayed for a significant number of cycles.
2. **Switch on every load -** This policy allows switching on every load, independent of whether it will cause a miss or not.

   In this case, R represents the average interval between loads. A general multithreading model assumes that a context is blocked for L cycles after every switch; but in the case of a switch-on-load processor, this happens only if the load causes a cache miss.
3. **Switch on every instruction** – This policy allows switching on every instruction, independent of whether it is a load or not. Successive instructions become independent , which will benefit pipelined execution.
4. **Switch on block of instruction** – Blocks of instructions from different threads are interleaved. This will improve the cache-hit ratio due to locality. It will also benefit single-context performance.

MODULE-5

Chapter-10 Parallel Programming Models, Languages and Compilers

10.1 Parallel Programming Models

Programming model -> simplified and transparent view of computer hardware/software system.

- Parallel Programming Model are specifically designed for multiprocessors, multicomputer or vector/SIMD computers.

We have 5 programming models-:

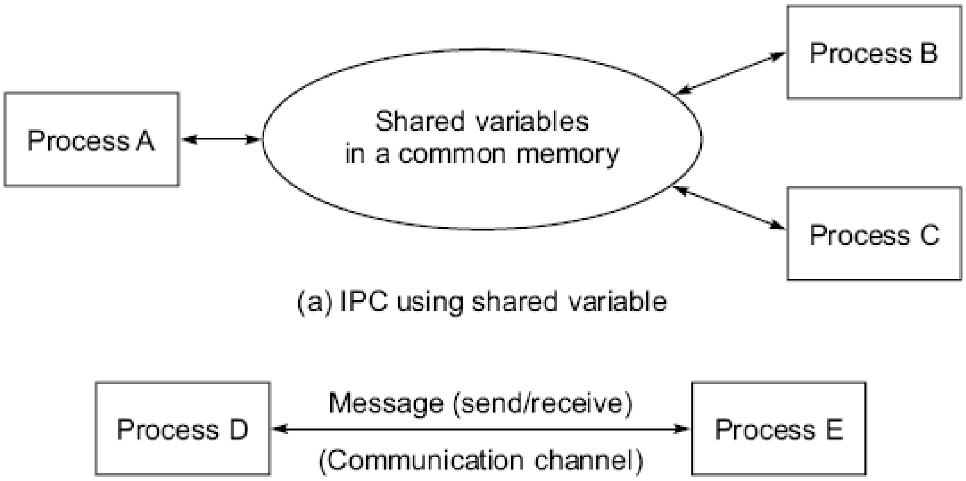
1. Shared-Variable Model
2. Message-Passing Model
3. Data-Parallel Model
4. Object Oriented Model
5. Functional and Logic Model

1. Shared-Variable Model

- In all programming system, processors are **active resources** and memory & IO devices are **passive resources**. Program is a collection of processes. Parallelism depends on how IPC(Interprocess Communication) is implemented. Process address space is shared.
- To ensure orderly IPC, a mutual exclusion property requires that shared object must be shared by only 1 process at a time.

Shared Variable communication

- Used in multiprocessor programming
- Shared variable IPC demands use of shared memory and mutual exclusion among multiple processes accessing the same set of variables.



(a) IPC using shared variable

(b) IPC using message passing

Fig. 10.1 Two basic mechanisms for interprocess communication (IPC).

Critical Section

- Critical Section(CS) is a code segment accessing shared variable, which must be executed by only one process at a time and which once started must be completed without interruption.
- It should satisfy following requirements-:

✓ **Mutual Exclusion:** At most one process executing CS at a time.

✓ **No deadlock in waiting**: No circular wait by 2 or more process.

✓ **No preemption:** No interrupt until completion.

✓ **Eventual Entry**: Once entered CS,must be out after completion.

Protected Access

- Granularity of CS affects the performance.
- If CS is too large,it may limit parallism due to excessive waiting by process.
- When CS is too small,it may add unnecessary code complexity/Software overhead.

4 operational Modes

- Multiprogramming
- Multiprocessing
- Multitasking

Multithreading

2. Message Passing Model

Two processes D and E residing at different processor nodes may communicate wit each other by passing messages through a direct network. The messages may be instructions, data,synchronization or interrupt signals etc. Multicomputers are considered loosely coupled multiprocessors.

Synchronous Message Passing

- No shared Memory
- No mutual Exclusion
- Synchronization of sender and reciever process just like telephone call.
- No buffer used.
- If one process is ready to cummunicate and other is not,the one that is ready must be blocked.

Asynchronous Message Passing

- Does not require that message sending and receiving be synchronised in time and space.
- Arbitrary communication delay may be experienced because sender may not know if and when the message has been received until acknowledgement is received from receiver.
- This scheme is like a postal service using mailbox with no synchronization between senders and receivers.

3. Data Parallel Model

- Used in SIMD computers. Parallelism handled by hardware synchronization and flow control.
- Fortran 90 ->data parallel lang.
- Require predistrubuted data sets.

Data Parallelism

- This technique used in array processors(SIMD)
- Issue->match problem size with machine size.

Array Language Extensions

- Various data parallel language used
- Represented by high level data types
- CFD for Illiac 4,DAP fortran for Distributed array processor,C* for Connection machine
- Target to make the number of PE’s of problem size.

4. Object Oriented Model

- Objects dynamically created and manipulated.
- Processing is performed by sending and receiving messages among objects.

Concurrent OOP

- Need of OOP because of abstraction and reusability concept.

- Objects are program entities which encapsulate data and operations in single unit.
- Concurrent manipulation of objects in OOP.

Actor Model

- This is a framework for Concurrent OOP.
- Actors -> independent component
- Communicate via asynchronous message passing.
- 3 primitives -> create, send-to and become.

Parallelism in COOP

3 common patterns for parallelism-:

1) Pipeline concurrency
2) Divide and conquer
3) Cooperative Problem Solving

5. Functional and logic Model

- Functional Programming Language-> Lisp,Sisal and Strand 88.

  Logic Programming Language-> Concurrent Prolog and Parlog

Functional Programming Model

- Should not produce any side effects.
- No concept of storage,assignment and branching.
- Single assignment and data flow language functional in nature.

Logic Programming Models

- Used for knowledge processing from large database.
- Supports implicitly search strategy.
- And parallel execution and Or Parallel Reduction technique used.
- Used in artificial intelligence

10.2 Parallel Languages and Compilers

- ✓ Programming environment is collection of s/w tools and system support.
- ✓ Parallel Software Programming environment needed.
- ✓ Users still forced to focus on hardware details rather than parallelism using high level abstraction.

10.2.1 Language Features for Parallelism

Language features for parallel programming for parallel programming into 6 categories:

1. Optimization Features
2. Availability Features
3. Synchronization/communication Features
4. Control Of Parallelism
5. Data Parallelism Features
6. Process Management Features

1. Optimization Features

- Conversion of sequential Program to Parallel Program.
- The purpose is to match s/w parallelism with hardware parallelism.
- Software in Practice-:

1) Automated Parallelizer

Express C automated parallelizer and Allaint FX Fortran compiler.

2) Semiautomated Parallizer

Needs compiler directives or programmers interaction.

2. Availability Features

Enhance user friendliness, make language portable for large no of parallel computers and expand the applicability of software libraries.

1) Scalability

Language should be scalable to number of processors and independent of hardware topology.

2) Compatibility

Compatible with sequential language.

3) Portability

Language should be portable to shared memory multiprocessor, message passing or both.

3. Synchronization/Communication Features

- Shared Variable (locks) for IPC
- Remote Procedure Calls
- Data Flow languages

- Mailbox,Semaphores,Monitors

4. Control of Parallelism

- Coarse,Medium and fine grain
- Explicit vs implicit parallelism
- Global Parallelism
- Loop Parallelism
- Task Parallelism
- Divide and Conquer Parallelism

5. Data Parallelism Features

How data is accessed and distributed in either SIMD and MIMD computers.

- Runtime automatic decomposition

Data automatically distributed with no user interaction.

- Mapping Specification

User specifies patterns and input data mapped to hardware.

- Virtual Processor Support

Compilers made statically and maps to physical processor.

- Direct Access to shared data

Shared data is directly accessed by operating system.

6. Process Management Features

Support efficient creation of parallel process,multithreading/multitasking,program partitioning and replication and dynamic load balancing at run time.

1) Dynamic Process Creation at Run Time.

2) Creation of lightweight processes.

3) Replication technique.

4) Partitioned Networks.

5) Automatic Load Balancing

10.2.2 Parallel Language Constructs

Special language constructs and data array expressions ar presented below for exploiting parallelism in programs.

Fortran 90 Array Notation

A multidimensional data array is represented by an array name indexed by a sequence of subscript triplets, one for each dimension. Triplets for different dimensions separated by commas.

Examples are:

$e_1 : e_2 : e_3$

$e_1 : e_2$

$e_1 : * : e_3$

$e_1 : *$

e_1

$*$

where each e1 is an arithmetic expression that must produce a scalar integer value. The first expression e1 is a lower bound, the second e2 an upper bound and the third e3 an increment (stride).

For example, **B(1:4:3, 6:8:2, 3)** represents four elements **B(1, 6, 3), B(4, 6, 3), B(1, 8, 3),** and **B(4, 8, 3),** of a three-dimensional array.

When the third expression in a triplet is missing, a unit stride is assumed. The * notation in the second expression indicates all elements in that dimension starting from e1 or the entire dimension if e1 is also omitted.

Parallel Flow Control

- The conventional Fortran Do loop declares that all scalar instructions within the **(Do, Enddo)** pair are executed sequentially, and so are the successive iterations.
- To declare parallel activities, we use the **(Doall, Endall)** pair.
- All iterations in the **Doall** loop are totally independent of each other. They can be executed in parallel if there are sufficient resources.
- When the successive loops depend on each other, we use the **(Doacross, EndAcross)** pair to declare parallelism with loop-carried dependences.

- The **(ForAll, EndAll)** and **(ParDo, ParEnd)** commands can be interpreted either as a **Doall** loop or as a **Doacross** loop.

```
        Doacross I=2, N
            Do J=2, N
S1:             A9I, J) = (A(I, J-1)) + A(I, J+1)) / 2
            Enddo
        Endacross
```

Another program construct is **(Cobegin, Coend)** pair. All computations specified within the block could be executed in parallel.

```
Cobegin
P1
P2
…..
Pn
Coend
```

Causes processes **P1, P2,… Pn** to start simultaneously and to proceed concurrently until they have all ended. The command **(Parbegin, Parend)** has equivalent meaning.

During the execution of a process, we can use a **Fork Q** command to spawn a new process Q:

```
Process P               Process Q
…..                     ………
Fork Q                  ………
……                      End
Join Q
```

The **Join Q** command recombines the two processes into one process.

10.2.3 Optimizing Compilers for Parallelism

- Role of compiler to remove burden of optimization and generation.

  3 Phases are-:

  1) Flow analysis
  2) Optimization
  3) Code Generation

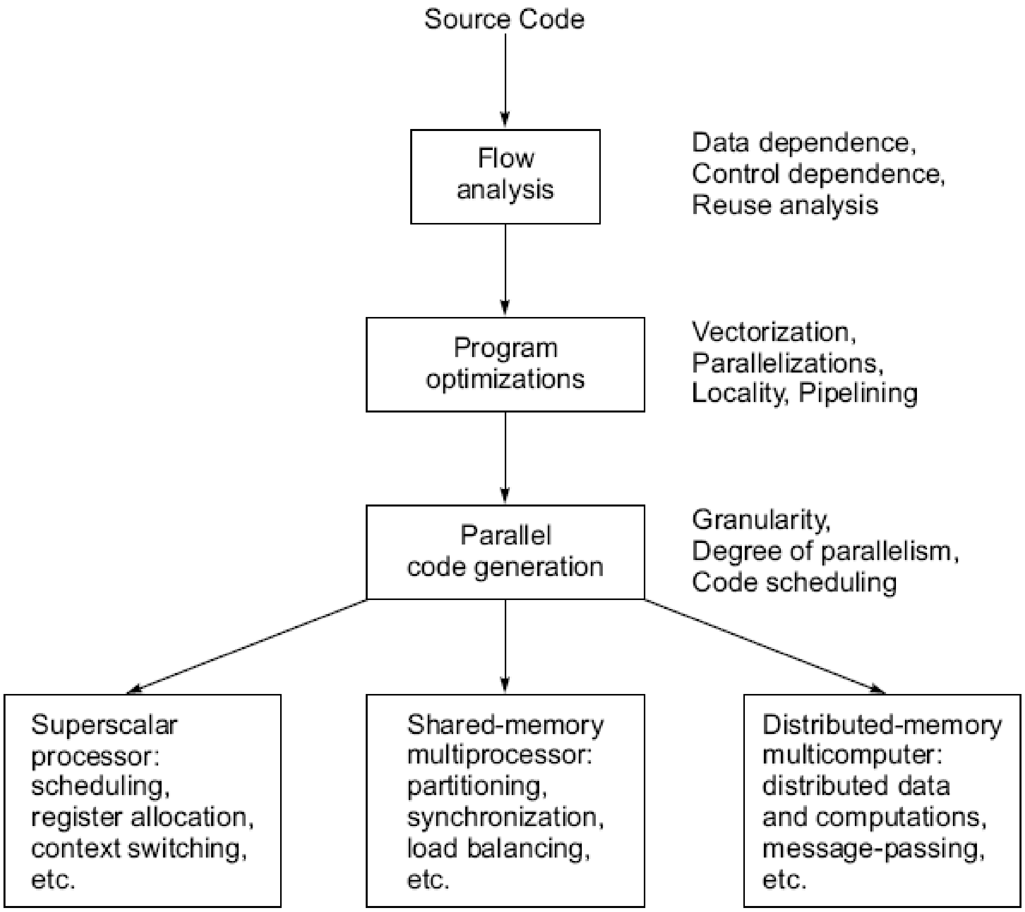

Source Code
Flow analysis
Data dependence, Control dependence, Reuse analysis
Program optimizations
Vectorization, Parallelizations, Locality, Pipelining
Parallel code generation
Granularity, Degree of parallelism, Code scheduling
Superscalar processor: scheduling, register allocation, context switching, etc.
Shared-memory multiprocessor: partitioning, synchronization, load balancing, etc.
Distributed-memory multicomputer: distributed data and computations, message-passing, etc.

Fig. 10.4 Compilation phases in parallel code generation

1) Flow analysis

- Reveals design flow patters to determine data and control dependencies.
- Flow analysis carried at various execution levels.

1)Instruction level->VLSI or superscaler processors.

2)Loop level->Simd and systolic computer

3)Task level->Multiprocessor/Multicomputer

2) Optimization

- Transformation of user program to explore hardware capability.
- Explores better performance.
- Goal to maximise speed of code execution.
- To minimize code length.
- Local and global optimizations.

Machine dependent Transformation

3) Parallel Code Generation

Compiler directive can be used to generate parallel code.

- 2 optimizing compilers-:

  1) Parafase and Parafase 2

  2) PFC and Parascope

Parafase and Parafase2

- Transforms sequential programs of fortran 77 into parallel programs.
- Parafase consists of 100 program that are encoded and passed.
- Pass list indentifies dependencies and converts it to concurrent program.
- Parafase2 for c and pascal in extension to fortran.

PFC AND Parascope

- Translates Fortran 77 to Fortran 90 code.
- PFC package extended to PFC + for parallel code generation on shared memory multiprocessor.
- PFC performs analysis as following steps below-:
- PFC performs analysis as following steps below-:

  1) Inter-procedure Flow analysis

  2) Transformation

  3) Dependence analysis

  4) Vector Code Generation

10.3 Dependence Analysis of Data Arrays

(Refer Text book)

Chapter-11 Parallel Program Development and Environments

11.2 Synchronization and Multiprocessing Modes

11.2.1 Principles of Synchronization

- The performance and correctness of a parallel program execution rely heavily on efficient synchronization among concurrent computations in multiple processors.
- The source of synchronization problem is the sharing of writable objects (data structures) among processes. Once a writable object permanently becomes read-only, the synchronization problem vanishes at that point.
- Synchronization consists of implementing the order of operations in an algorithm by observing the dependences for writable data.
- Lowe-level synchronization primitives are often implemented directly in hardware. Resources such as the CPU, bus or network and memory units may also be involved in synchronization of parallel computations.

  The following methods are used for implementing efficient synchronization schemes.

  - Atomic Operations
  - Wait Protocols
  - Fairness policies'
  - Access order
  - Sole access protocols

11.2.2 Multiprocessor Execution Modes

Multiprocessor supercomputers are built for vector processing as ewll as for parallel processing across multiple processors.

Multiprocessing modes include parallel execution from the fine-grain process level to the medium-grain task level and to the coarse-grain program level.

Multiprocessing Requirements

- Fast context switching among multiple processes resident in processors
- Multiple register sets to facilitate context switching
- Fast memory access with conflict-free memory allocation

- Effective synchronization mechanism among multiple processors
- Software tools for achieving parallel processing and performance monitoring
- System and application software for interactive users.

Multitasking Environments

Multitasking exploits parallelism at several levels:

- Functional units are pipelined or chained together
- Multiiple functional units are closed concurrently
- I/O and CPU activities are overlapped
- Multiple CPUs cooperate on a single program to achieve minimal execution time

11.2.3 Multitasking on Cray Multiprocessors

Three levels of multitasking are:

1. Macrotasking
2. Microtasking
3. Autotasking

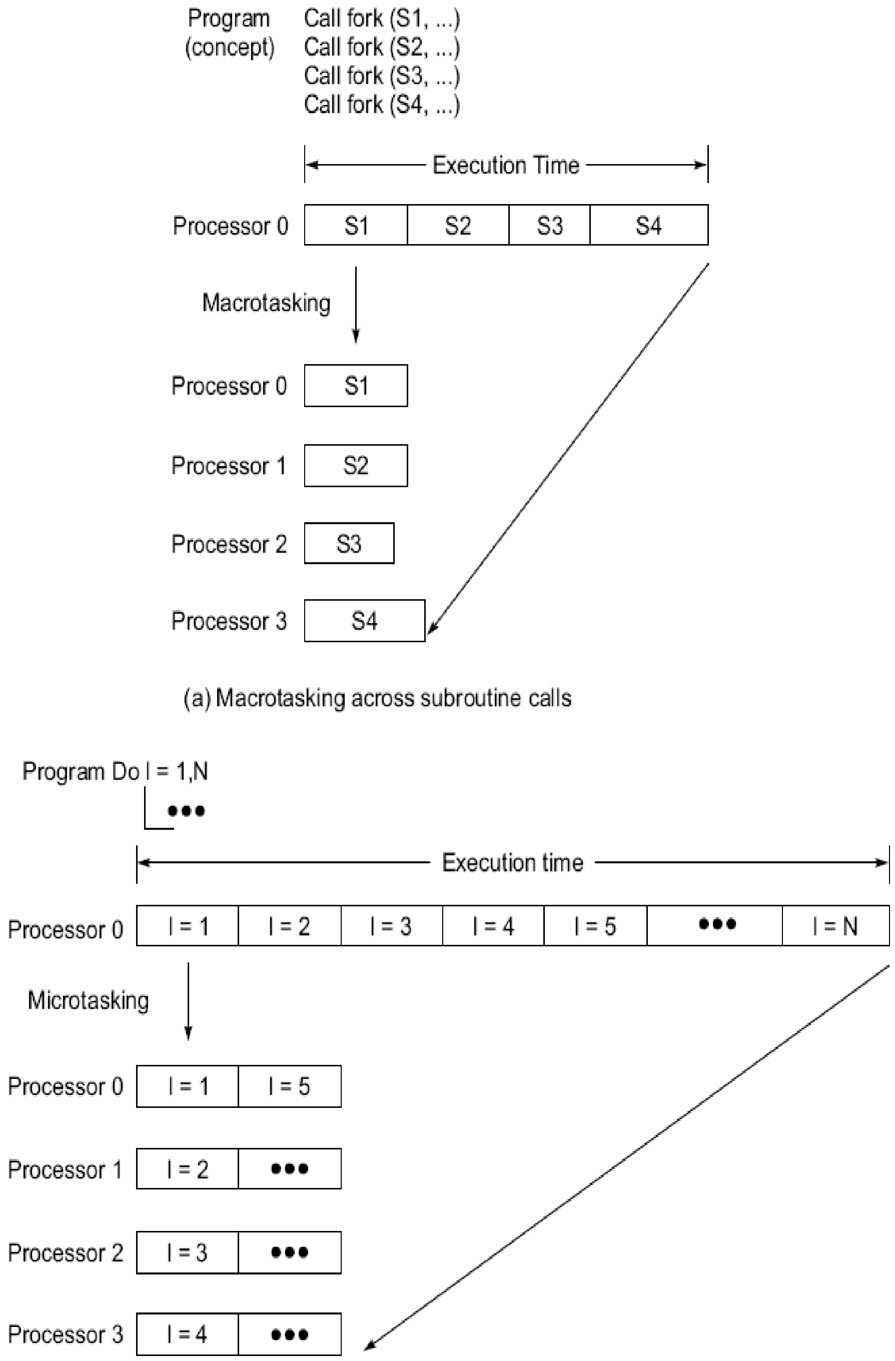
Macrotasking: When multitasking is conducted at the level of subroutine calls, it is called macrotasking with medium to coarse grains. The concept of macrotasking is shown in Fig 11.4a.

A main program forks a subroutine S1 and then forks out three additional subroutines S2, S3 and S4.

Microtasking: This corresponds to multitasking at the loop level with finer granularity. Compiler directives are often used to declare parallel execution of independent or dependent iterations of a looping program construct.

Fig 11.4b illustrates the spread of every four instructions of a Do loop to four processors simultaneously through microtasking.

Autotasking: It automatically divides a program into discrete tasks for parallel execution on a multiprocessor.



(a) Macrotasking across subroutine calls

(b) Microtasking across loop iterations

Fig. 11.4 Multitasking at two different processing levels

Chapter-12 Instruction Level Parallelism

12.1 Computer Architecture

(a) Computer Architecture is defined as the arrangement by which the various system building blocks—processors, functional units, main memory, cache, data paths, and so on—are interconnected and inter-operated to achieve desired *system performance.*

(b) Processors make up the most important part of a computer system. Therefore, in addition to (a), **processor design** also constitutes a central and very important element of computer architecture. Various functional elements of a processor must be designed, interconnected and inter-operated to achieve desired ***processor performance***.

- *System performance* is the key benchmark in the study of computer architecture. A computer system must solve the real world problem, or support the real world application, for which the user is installing it.
- A basic rule of system design is that *there should be no performance bottlenecks in the system*.
- Typically, a performance bottleneck arises when one part of the system.
- In a computer system, the key subsystems are processors, memories, I/O interfaces, and the data paths connecting them. Within the processors, we have subsystems such as functional units, registers, cache memories, and internal data buses.
- Within the computer system as a whole—or within a single processor—designers do not wish to create bottlenecks to system performance.

Example 12.1 Performance bottleneck in a system

In Fig. 12.1 we see the schematic diagram of a simple computer system consisting of four processors, a large shared main memory, and a processor-memory bus.

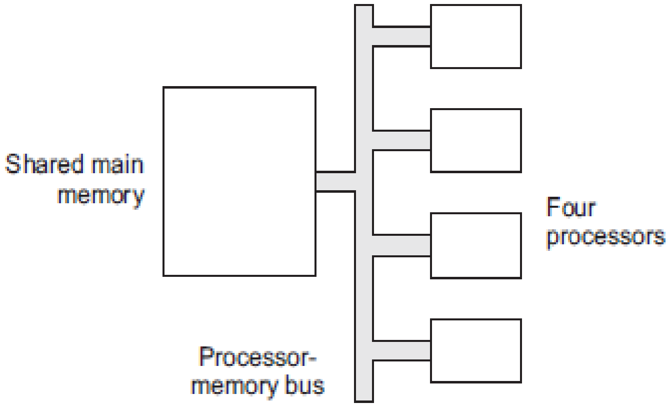


Fig. 12.1 A simple shared memory multiprocessor system

For the three subsystems, we assume the following performance figures:

(i) Each of the four processors can perform double precision floating point operations at the rate of 500 million per second, i.e. 500 MFLOPs.

(ii) The shared main memory can read/write data at the aggregate rate of 1000 million 32-bit words per second.

(iii) The processor-memory bus has the capability of transferring 500 million 32-bit words per second to/from main memory.

12.4 Model of a Typical Processor

- A processor with *load-store* instruction set architecture and a set of programmable registers as seen by the assembly language programmer or the code generator of a compiler.
- Whether these registers are bifurcated into separate sets of integer and floating point registers is not important for us at present, nor is the exact number of these registers.
- To support parallel access to instructions and data at the level of the fastest cache, we assume that L1 cache is divided into instruction cache and data cache, and that this split L1 cache supports single cycle access for instructions as well as data.
- Some processors may have an *instruction buffer* in place of L1 instruction cache; for the purposes of this section, however, the difference between them is not important.
- The first three pipeline stages on our prototype processor are ***fetch***, ***decode*** and ***issue***.
- Following these are the various functional units of the processor, which include integer unit(s), floating point unit(s), load/store unit(s), and other units as may be needed for a specific design.

- Let us assume that our superscalar processor is designed for *k* instruction issues in every processor clock cycle.
- Clearly then the *fetch*, *decode* and *issue* pipeline stages, as well as the other elements of the processor, must all be designed to process *k* instructions in every clock cycle.

- On multiple issue pipelines, *issue* stage is usually separated from *decode* stage. One reason for thus increasing a pipeline stage is that it allows the processor to be driven by a faster clock.
- *Decode* stage must be seen as preparation for instruction *issue* which—by definition—can occur only if the relevant functional unit in the processor is in a state in which it can accept one more operation for execution.
- As a result of the *issue*, the operation is handed over to the functional unit for execution.

- The process of issuing instructions to functional units also involves instruction scheduling. For example, if instruction I_j cannot be issued because the required functional unit is not free, then it may still be possible to issue the next instruction I_{j+1}—provided that no dependence between the two prohibits issuing instruction I_{j+1}.
- When instruction scheduling is specified by the compiler in the machine code it generates, we refer to it as *static scheduling*.
- In theory, static scheduling should free up the processor hardware from the complexities of instruction scheduling; in practice, though, things do not quite turn out that way.
- If the processor control logic schedules instruction *on the fly*—taking into account inter-instruction dependences as well as the state of the functional units—we refer to it as *dynamic scheduling*.
- Much of the rest of this chapter is devoted to various aspects and techniques of dynamic scheduling.
- The basic aim in both types of scheduling—static as well as dynamic—is to maximize the instruction level parallelism which is exploited in the executing sequence of instructions.
- The process of issuing instructions to functional units also involves instruction scheduling.
- A branch prediction unit has also been shown in Fig. 12.4 and Fig. 12.5 to implement some form of a branch prediction algorithm

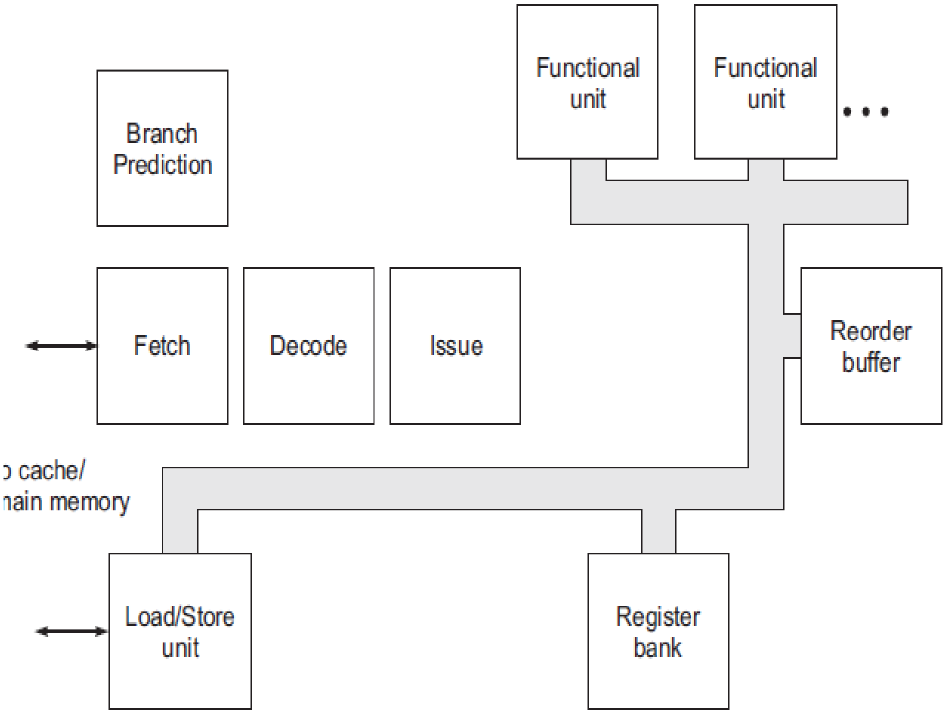


Fig. 12.4 Processor design with reorder buffer

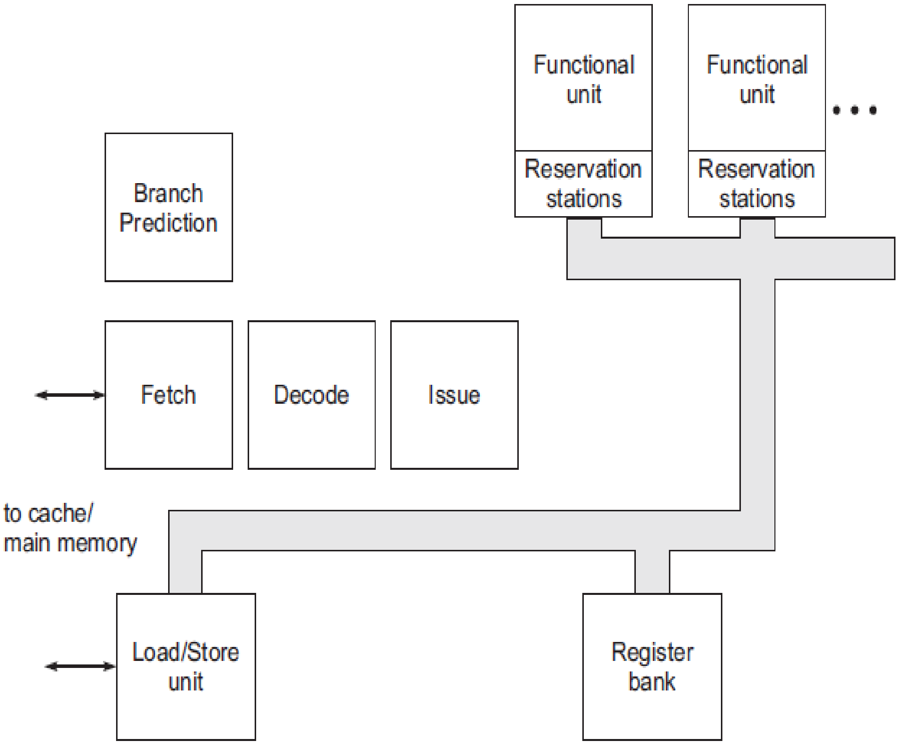


Fig. 12.5 Processor design with reservation stations on functional units

Figure 12.5 shows a processor design in which functional units are provided with reservation stations. Such designs usually also make use of operand forwarding over a common data bus (CDB), with tags to identify the source of data on the bus. Such a design also implies register renaming, which resolves RAW and WAW dependences.

12.5 Compiler-detected Instruction Level Parallelism

One relatively simple technique which the compiler can employ is known as loop unrolling, by which independent instructions from multiple successive iterations of a loop can be made to execute in parallel. Unrolling means that the body of the loop is repeated n times for n successive values of the control variable—so that one iteration of the transformed loop performs the work of n iterations of the original loop.

Loop Unrolling

Independent instructions from multiple successive iterations of a loop can be made to execute in parallel. *Unrolling* means that the body of the loop is repeated *n* times for *n* successive values of the control variable—so that one iteration of the transformed loop performs the work of *n* iterations of the original loop.

Consider the following body of a loop in a user program, where all the variables except the loop control variable i are assumed to be floating point:

```
for i = 0 to 58 do
        c[i] = a[i]*b[i] – p*d[i];
```

Now suppose that machine code is generated by the compiler as though the original program had been written as:

```
for j = 0 to 52 step 4 do
{
        c[j] = a[j]*b[j] – p*d[j];
        c[j + 1] = a[j + 1]*b[j + 1] – p*d[j + 1];
        c[j + 2] = a[j + 2]*b[j + 2] – p*d[j + 2];
        c[j + 3] = a[j + 3]*b[j + 3] – p*d[j + 3];
}
c[56] = a[56]*b[56] – p*d[56];
c[57] = a[57]*b[57] – p*d[57];
```

c[58] = a[58]*b[58] – p*d[58];

Note carefully the values of loop variable j in the transformed loop. the two program fragments are equivalent, in the sense that they perform the same computation.

Of course the compiler does not transform one source program into another—it simply produces machine code corresponding to the second version, with the *unrolled* loop.

12.6 Operand Forwarding

It helps in reducing the impact of true data dependences in the instruction stream. Consider the following simple sequence of two instructions in a running program:

ADD R1, R2, R3

SHIFTR #4, R3, R4

The result of the ADD instruction is stored in destination register R3, and then shifted right by four bits in the second instruction, with the shifted value being placed in R4.

Thus, there is a simple RAW *dependence* between the two instructions—the output of the first is required as input operand of the second represented in the form of graph as below:

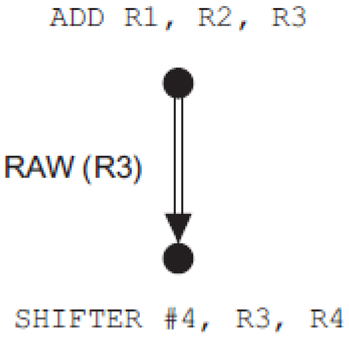


Fig. 12.6 RAW dependence between two instructions

- In a pipelined processor, ideally the second instruction should be executed one stage—and therefore one clock cycle—behind the first.
- However, the difficulty here is that it takes one clock cycle to transfer ALU output to destination register R3, and then another clock cycle to transfer the contents of register R3 to ALU input for the right shift. Thus a total of two clock cycles are needed to bring the result of the first instruction where it is needed for the second instruction.

- Therefore, as things stand, the second instruction above cannot be executed just one clock cycle behind the first.
- This sequence of data transfers has been illustrated in Fig. 12.7 (a). In clock cycle Tk, ALU output is transferred to R3 over an internal data path. In the next clock cycle Tk + 1, the content of R3 is transferred to ALU input for the right shift.
- When carried out in this order, clearly the two data transfer operations take two clock cycles.

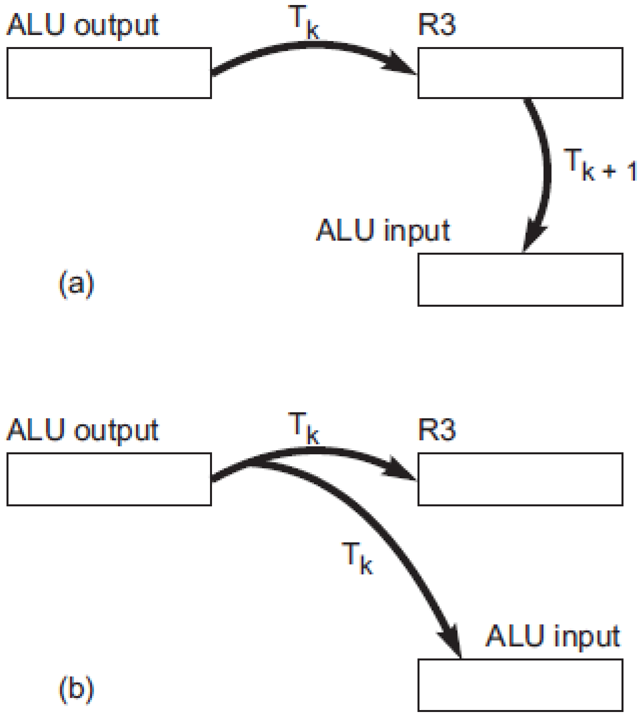


Fig. 12.7 Two data transfers (a) in sequence and (b) in parallel.

- But note that the required two transfers of data can be achieved in only one clock cycle if ALU output is sent to both R3 and ALU input in the same clock cycle—as illustrated in Fig. 12.7 (b).
- In general, if X is to be copied to Y, and in the next clock cycle Y is to be copied to Z, then we can just as well copy X to both Y and Z in one clock cycle.

12.7 Reorder Buffer

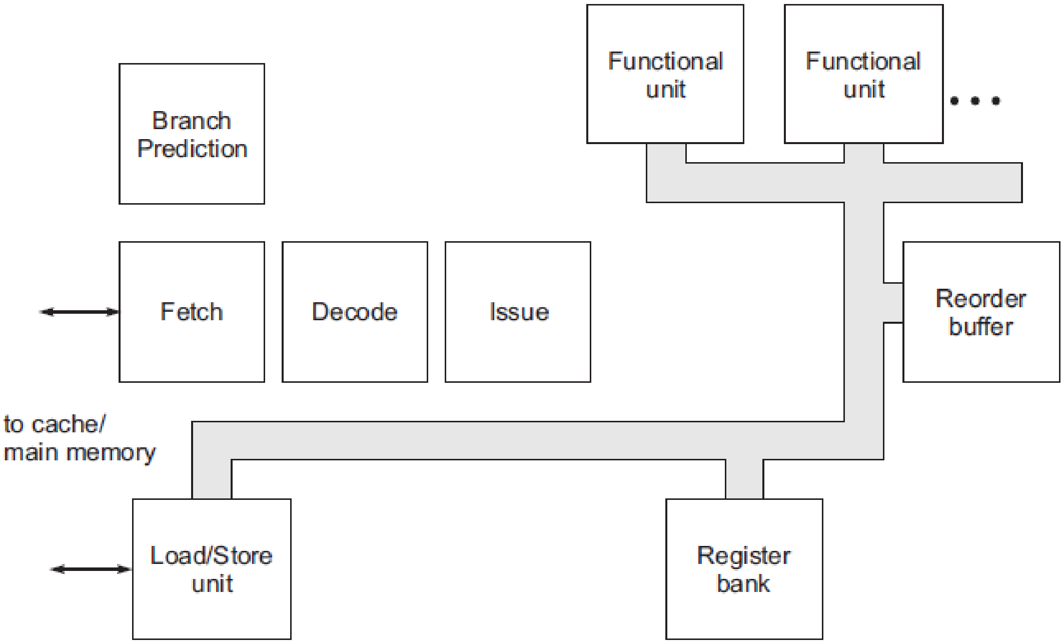
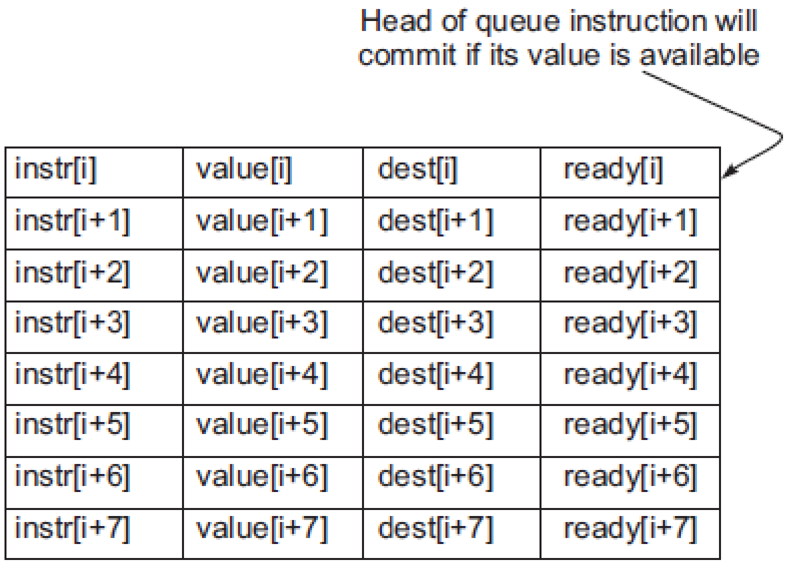


Fig. 12.4 Processor design with reorder buffer

- Since instructions execute in parallel on multiple functional units, the reorder buffer serves the function of bringing completed instructions back into an order which is consistent with program order.
- Note that instructions may *complete* in an order which is not related to program order, but must be *committed* in program order.
- At any time, *program state* and *processor state* are defined in terms of instructions which have been committed—i.e. their results are reflected in appropriate registers and/or memory locations.
- Entries in the reorder buffer are completed instructions, which are queued in program order.
- However, since instructions do not necessarily complete in program order, we also need a flag with each reorder buffer entry to indicate whether the instruction in that position has completed.



Head of queue instruction will commit if its value is available

instr[i]	value[i]	dest[i]	ready[i]
instr[i+1]	value[i+1]	dest[i+1]	ready[i+1]
instr[i+2]	value[i+2]	dest[i+2]	ready[i+2]
instr[i+3]	value[i+3]	dest[i+3]	ready[i+3]
instr[i+4]	value[i+4]	dest[i+4]	ready[i+4]
instr[i+5]	value[i+5]	dest[i+5]	ready[i+5]
instr[i+6]	value[i+6]	dest[i+6]	ready[i+6]
instr[i+7]	value[i+7]	dest[i+7]	ready[i+7]

Fig. 12.9 Entries in a reorder buffer of size eight

Figure 12.9 shows a reorder buffer of size eight. Four fields are shown with each entry in the reorder buffer—instruction identifier, value computed, program-specified destination of the value computed, and a flag indicating whether the instruction has completed (i.e. the computed value is available).

We now take a brief look at how the use of reorder buffer addresses the various types of dependences in the program.

1. **Data Dependences:** A RAW dependence—i.e. true data dependence—will hold up the execution of the dependent instruction if the result value required as its input operand is not available. As suggested above, operand forwarding can be added to this scheme to speed up the supply of the needed input operand as soon as its value has been computed.

WAR and WAW dependences—i.e. anti-dependence and output dependence, respectively—also hold up the execution of the dependent instruction and create a possible pipeline stall. We shall see below that the technique of register renaming is needed to avoid the adverse impact of these two types of dependences.

2**. Control Dependences:**

Suppose the instruction(s) in the reorder buffer belong to a branch in the program which should not have been taken—i.e. there has been a mis-predicted branch. Clearly then the reorder buffer should be flushed along with other elements of the pipeline.

Therefore the performance impact of control dependences in the running program is determined by the accuracy of branch prediction technique employed.

The reorder buffer plays no direct role in the handling of control dependences.

3. **Resource Dependences:** If an instruction needs a functional unit to execute, but the unit is not free, then the instruction must wait for the unit to become free—clearly no technique in the world can change that.

In such cases, the processor designer can aim to achieve at least this: if a subsequent instruction needs to use another functional unit which is free, then the subsequent instruction can be executed out of order.

12.8 Register Renaming

- Traditional compilers and assembly language programmers work with a fairly small number of programmable registers.
- Therefore the only way to make a larger number of registers available to instructions under execution within the processor is to make the additional registers *invisible* to machine language instructions.
- Instructions *under execution* would use these additional registers, even if instructions making up the machine language program stored in memory cannot refer to them.

For example, let us say that the instruction:

FADD R1, R2, R5

is followed by the instruction:

FSUB R3, R4, R5

Both these instructions are writing to register R5, creating thereby a WAW dependence—i.e. output dependence—on register R5.

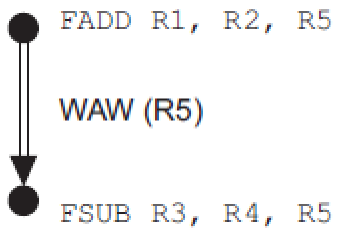


Fig. 12.10 WAW dependence

- Let FSUB write its output value to a register other than R5, and let us call that other register X. Then the instructions which use the value generated by FSUB will refer to X, while the instructions which use the value generated by FADD will continue to refer to R5.
- Now, since FADD and FSUB are writing to two different registers, the output dependence or WAW between them has been removed.

- When FSUB *commits*, then the value in R5 should be updated by the value in X—i.e. the value computed by FSUB.
- Then the physical register X, which is not a program visible register, can be freed up for use in another such situation.
- Note that here we have *mapped*—or *renamed*—R5 to X, for the purpose of storing the result of FSUB, and thereby removed the WAW dependence from the instruction stream. A pipeline stall will now not be created due to the WAW dependence.

12.9 Tomasulo's Algorithm

- *Register renaming* was also an implicit part of the original algorithm.
- For register renaming, we need a set of program invisible registers to which programmable registers are re-mapped.
- Tomasulo's algorithm requires these program invisible registers to be provided with reservation stations of functional units.
- Let us assume that the functional units are internally pipelined, and can complete one operation in every clock cycle.
- Therefore each functional unit can initiate one operation in every clock cycle—provided of course that a reservation station of the unit is ready with the required input operand value or values.
- Note that the exact depth of this functional unit pipeline does not concern us for the present.
- Figure 12.12 shows such a functional unit connected to the common data bus, with three reservation stations provided on it.

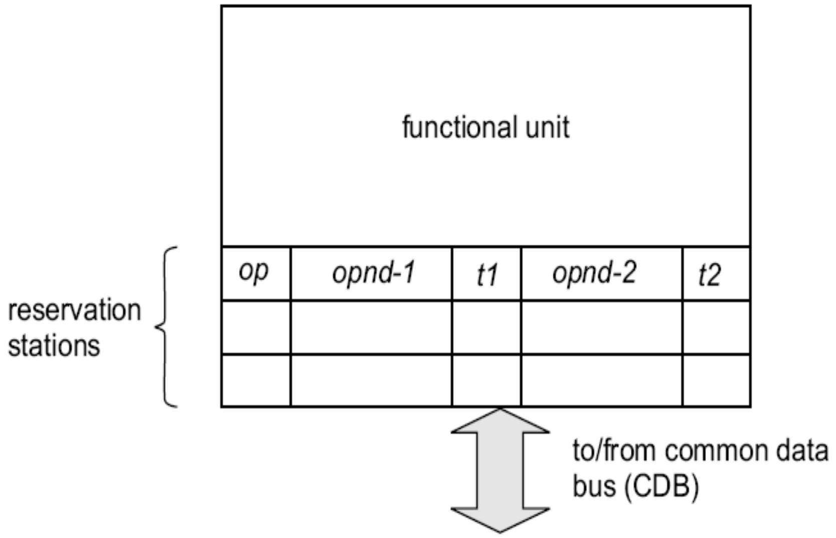


Fig. 12.12 Reservation stations provided with a functional unit

The various fields making up a typical reservation station are as follows:

op	operation to be carried out by the functional unit
opnd-1 &	
opnd-2	two operand values needed for the operation
t1 & t2	two source tags associated with the operands

- When the needed operand value or values are available in a reservation station, the functional unit can initiate the required operation in the next clock cycle.
- At the time of instruction issue, the reservation station is filled out with the operation code (*op*).
- If an operand value is available, for example in a programmable register, it is transferred to the corresponding source operand field in the reservation station.
- However, if the operand value is not available at the time of issue, the corresponding source tag (*t1* and/or *t2*) is copied into the reservation station.
- The source tag identifies the source of the required operand. As soon as the required operand value is available at its source—which would be typically the output of a functional unit—the data value is forwarded over the common data bus, along with the source tag.
- This value is copied into all the reservation station operand slots which have the matching tag.
- Thus operand forwarding is achieved here with the use of tags. All the destinations which require a data value receive it in the same clock cycle over the common data bus, by matching their stored operand tags with the source tag sent out over the bus.

Tomasulo's algorithm and RAW dependence

- Assume that instruction I1 is to write its result into R4, and that two subsequent instructions I2 and I3 are to read—i.e. make use of—that result value.
- Thus instructions I2 and I3 are truly data dependent (RAW dependent) on instruction I1. See Fig. 12.13.
- Assume that the value in R4 is not available when I2 and I3 are issued; the reason could be, for example, that one of the operands needed for I1 is itself not available.
- Thus we assume that I1 has not even started executing when I2 and I3 are issued.
- When I2 and I3 are issued, they are parked in the reservation stations of the appropriate functional units.
- Since the required result value from I1 is not available, these reservation station entries of I2 and I3 get source tag corresponding to the output of I1—i.e. output of the functional unit which is performing the operation of I1.
- When the result of I1 becomes available at its functional unit, it is sent over the common data bus along with the tag value of its source—i.e. output of functional unit.
- At this point, programmable register R4 as well as the reservation stations assigned to I2 and I3 have the matching source tag—since they are waiting for the same result value, which is being computed by I1.

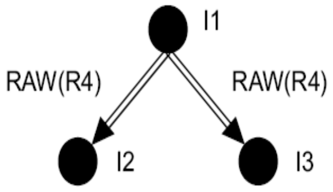


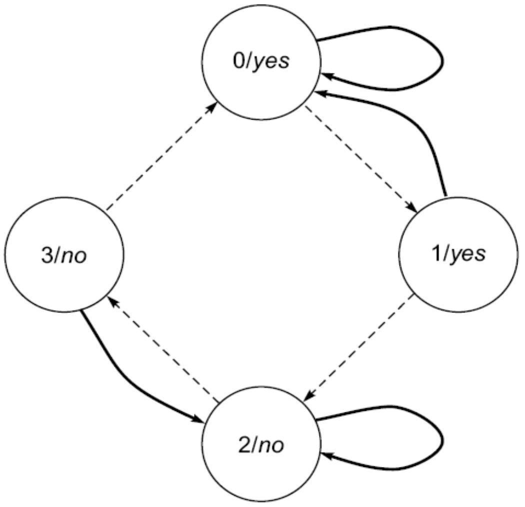
Fig. 12.13 Example of RAW dependences

- When the tag sent over the common data bus matches the tag in any destination, the data value on the bus is copied from the bus into the destination.
- The copy occurs at the same time into all the destinations which require that data value. Thus R4 as well as the two reservation stations holding I2 and I3 receive the required data value, which has been computed by I1, at the same time over the common data bus.

- Thus, through the use of source tags and the common data bus, in one clock cycle, three destination registers receive the value produced by I1—programmable register R4, and the operand registers in the reservation stations assigned to I2 and I3.
- Let us assume that, at this point, the second operands of I2 and I3 are already available within their corresponding reservation stations.
- Then the operations corresponding to I2 and I3 can begin in parallel as soon as the result of I1 becomes available—since we have assumed here that I2 and I3 execute on two separate functional units.

12.10 Branch Prediction

- About 15% to 20% of instructions in a typical program are branch and jump instructions, including procedure returns.
- Therefore—if hardware resources are to be fully utilized in a superscalar processor—the processor must start working on instructions beyond a branch, even before the branch instruction itself has completed. This is only possible through some form of branch prediction.
- What can be the logical basis for branch prediction? To understand this, we consider first the reasoning which is involved if one wishes to predict the result of a tossed coin.



In states 0 & 1: Branch taken
In states 2 & 3 Branch no taken

Solid line: Correct predication.
Broken line: incorrect prediction

Fig. 12.15 State transition diagram of 2-bit branch predictor[8]

- A basic branch prediction technique uses a so-called *two-bit predictor*. A two-bit counter is maintained for every conditional branch instruction in the program. The two-bit counter has four possible states; these four states and the possible transitions between these states are shown in Fig. 12.15.

- When the counter state is 0 or 1, the respective branch is predicted as *taken*; when the counter state is 2 or 3, the branch is predicted as *not taken*.
- When the conditional branch instruction is executed and the actual branch outcome is known, the state of the respective two-bit counter is changed as shown in the figure using solid and broken line arrows.
- When two successive predictions come out wrong, the prediction is changed from *branch taken* to *branch not taken*, and *vice versa*.

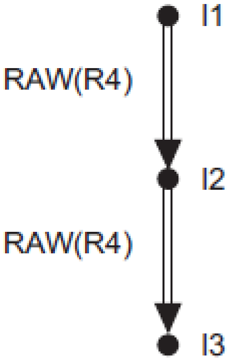


Fig. 12.14 Example of RAW & WAR dependences

- In Fig. 12.14, state transitions made on mis-predictions are shown using broken line arrows, while solid line arrows show state transitions made on predictions which come out right.
- This scheme uses a two-bit counter for every conditional branch, and there are many conditional branches in the program.
- Overall, therefore, this branch prediction logic needs a few kilobytes or more of fast memory.
- One possible organization for this branch prediction memory is in the form of an array which is indexed by low order bits of the instruction address.
- If twelve low order bits are used to define the array index, for example, then the number of entries in the array is 4096.
- To be effective, branch prediction should be carried out as early as possible in the instruction pipeline.
- As soon as a conditional branch instruction is decoded, branch prediction logic should predict whether the branch is taken.

- Accordingly, the next instruction address should be taken either as the branch target address (i.e. branch is taken), or the sequentially next address in the program (i.e. branch is not taken).

12.12 Thread Level Parallelism

- One way to reduce the burden of dependences is to combine—with hardware support within the processor—instructions from multiple independent threads of execution.
- Such hardware support for multi-threading would provide the processor with a pool of instructions, in various stages of execution, which have a relatively smaller number of dependences amongst them, since the threads are independent of one another.
- Let us consider once again the processor with instruction pipeline of depth eight, and with targeted superscalar performance of four instructions completed in every clock cycle.
- Now suppose that these instructions come from four independent threads of execution. Then, on average, the number of instructions in the processor at any one time from one thread would be 4 X 8/4 = 8.
- With the threads being independent of one another, there is a smaller total number of data dependences amongst the instructions in the processor.
- Further, with control dependences also being separated into four threads, less aggressive branch prediction is needed.
- Another major benefit of such hardware-supported multi-threading is that pipeline stalls are very effectively utilized.
- If one thread runs into a pipeline stall—for access to main memory, say—then another thread makes use of the corresponding processor clock cycles, which would otherwise be wasted.
- Thus hardware support for multi-threading becomes an important latency hiding technique.
- To provide support for multi-threading, the processor must be designed to switch between threads—either on the occurrence of a pipeline stall, or in a round robin manner.
- As in the case of the operating system switching between running processes, in this case the hardware context of a thread within the processor must be preserved.

Depending on the specific strategy adopted for switching between threads, hardware support for **multi-threading** may be classified as one of the following:

1. Coarse-grain multi-threading: It refers to switching between threads only on the occurrence of a major pipeline stall—which may be caused by, say, access to main memory, with latencies of the order of a hundred processor clock cycles.

2. Fine-grain multi-threading: It refers to switching between threads on the occurrence of any pipeline stall, which may be caused by, say, L1 cache miss. But this term would also apply to designs in which processor clock cycles are regularly being shared amongst executing threads, even in the absence of a pipeline stall.

3. Simultaneous multi-threading: It refers to machine instructions from two (or more) threads being issued in parallel in each processor clock cycle. This would correspond to a multiple-issue processor where the multiple instructions issued in a clock cycle come from an equal number of independent execution threads.

MODULE-1

Chapter-1 Parallel Computer Models

1.1 The State of Computing

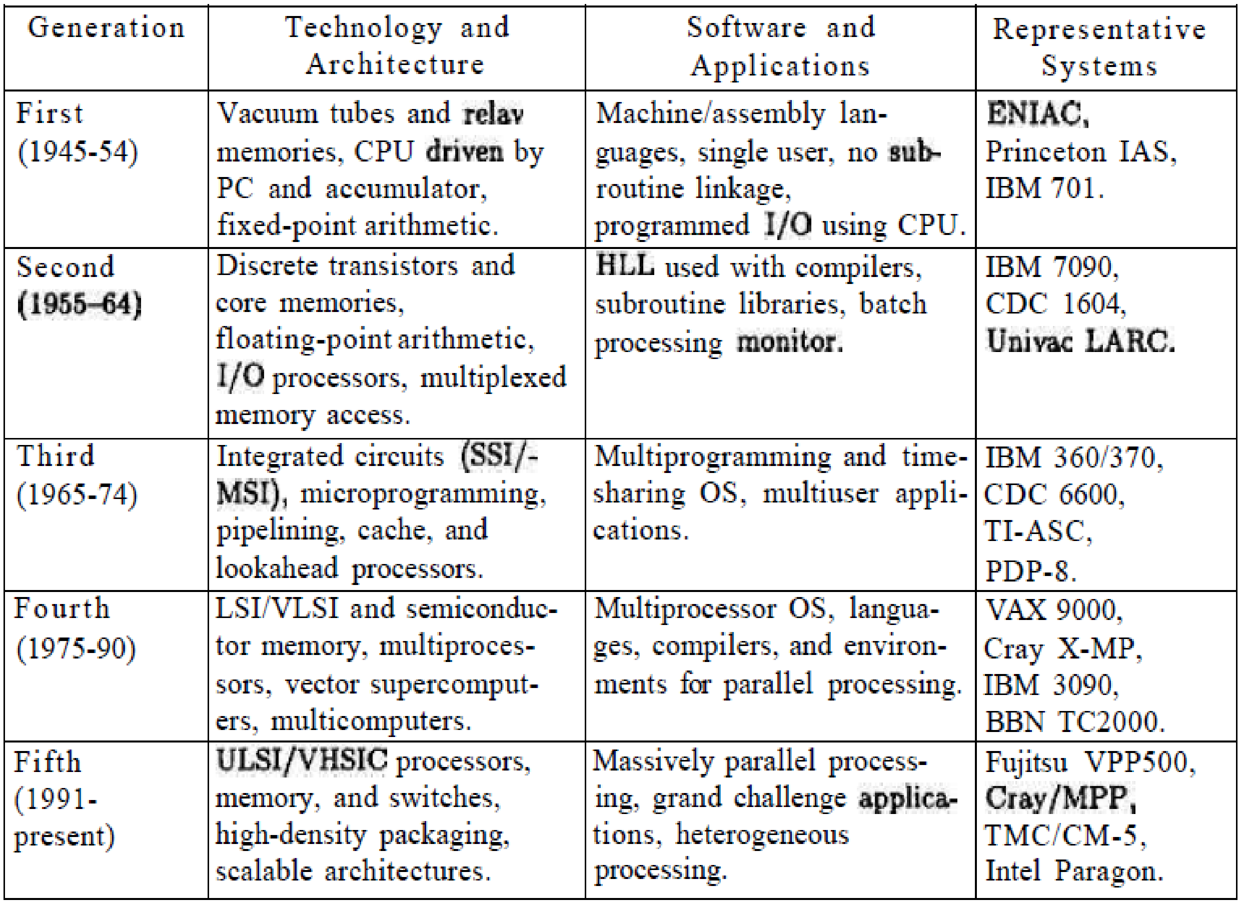
1.1.1 Computer Development Milestones

- Computers have gone through two major stages of development: mechanical and electronic. Prior to 1945, computers were made with mechanical or electromechanical parts.
- The earliest mechanical computer can be traced back to 500 BC in the form of the abacus used in China.
- The abacus is manually operated to perform decimal arithmetic with carry propagation digit by digit.
- Blaise Pascal built a mechanical adder/subtractor in Prance in 1642. Charles Babbage designed a difference engine in England for polynomial evaluation in 1827.
- Konrad Zuse built the first binary mechanical computer in Germany in 1941. Howard Aiken proposed the very first electromechanical decimal computer, which was built as the Harvard Mark I by IBM in 1944.
- Both Zuse's and Aiken's machines were designed for general-purpose computations.

Computer Generations

- Over the past five decades, electronic computers have gone through five generations of development. Each of the first three generations lasted about 10 years.
- The fourth generation covered a time span of 15 years.
- We have just entered the fifth generation with the use of processors and memory devices with more than 1 million transistors on a single silicon chip.

Table 1-1 Five Generations of Electronic Computers



Generation	Technology and Architecture	Software and Applications	Representative Systems
First (1945-54)	Vacuum tubes and relav memories, CPU driven by PC and accumulator, fixed-point arithmetic.	Machine/assembly languages, single user, no subroutine linkage, programmed I/O using CPU.	ENIAC, Princeton IAS, IBM 701.
Second (1955–64)	Discrete transistors and core memories, floating-point arithmetic, I/O processors, multiplexed memory access.	HLL used with compilers, subroutine libraries, batch processing monitor.	IBM 7090, CDC 1604, Univac LARC.
Third (1965-74)	Integrated circuits (SSI/MSI), microprogramming, pipelining, cache, and lookahead processors.	Multiprogramming and time-sharing OS, multiuser applications.	IBM 360/370, CDC 6600, TI-ASC, PDP-8.
Fourth (1975-90)	LSI/VLSI and semiconductor memory, multiprocessors, vector supercomputers, multicomputers.	Multiprocessor OS, languages, compilers, and environments for parallel processing.	VAX 9000, Cray X-MP, IBM 3090, BBN TC2000.
Fifth (1991-present)	ULSI/VHSIC processors, memory, and switches, high-density packaging, scalable architectures.	Massively parallel processing, grand challenge applications, heterogeneous processing.	Fujitsu VPP500, Cray/MPP, TMC/CM-5, Intel Paragon.

1.1.2 Elements of Modern Computers

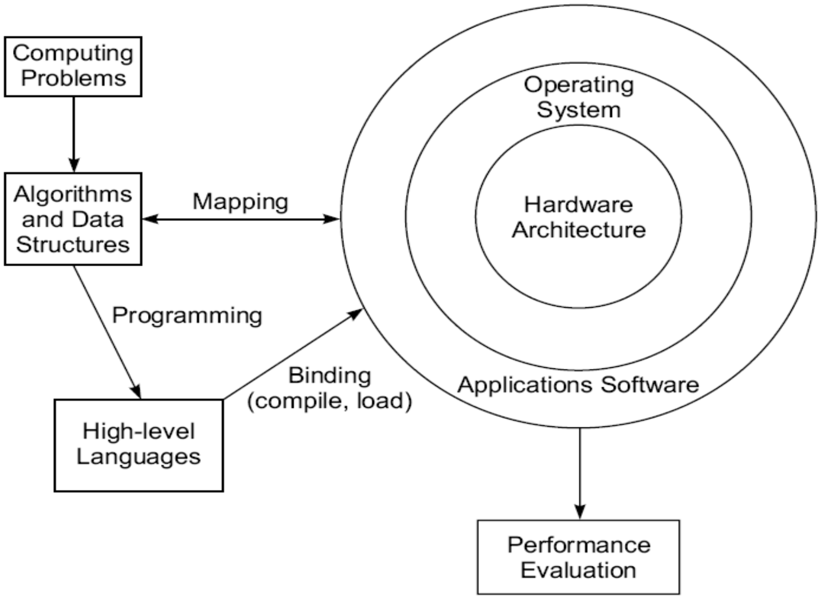


Fig. 1.1 Elements of a modern computer system

- Computing Problems

- The use of a computer is driven by real-life problems demanding fast and accurate solutions. Depending on the nature of the problems, the solutions may require different computing resources.
- For numerical problems in science and technology, the solutions demand complex mathematical formulations and tedious integer or floating-point computations.
- For alpha numerical problems in business and government, the solutions demand accurate transactions, large database management, and information retrieval operations.
- For artificial intelligence (AI) problems, the solutions demand logic inferences and symbolic manipulations.
- These computing problems have been labeled *numerical computing*, *transaction processing*, and *logical reasoning*.
- Some complex problems may demand a combination of these processing modes.

- Hardware Resources

- A modern computer system demonstrates its power through coordinated efforts by hardware resources, an operating system, and **application** software.
- Processors, memory, and peripheral devices form the hardware core of a computer system.
- Special hardware interfaces are often built into I/O devices, such as terminals, workstations, optical page scanners, magnetic ink character recognizers, modems, file servers, voice data entry, printers, and plotters.
- These peripherals are connected to mainframe computers directly or through local or wide-area networks.

- Operating System

- An effective operating system manages the allocation and deallocation of resources during the execution of user programs.
- Beyond the OS, application software must be developed to benefit the users.
- Standard benchmark programs are needed for performance evaluation.
- Mapping is a bidirectional process matching algorithmic structure with hardware architecture, and vice versa.
- Efficient mapping will benefit the programmer and produce better source codes.

- The mapping of algorithmic and data structures onto the machine architecture includes processor scheduling, memory maps, interprocessor communications, etc.
- These activities are usually architecture-dependent.

- System Software Support

- Software support is needed for the development of efficient programs in high-level languages. The source code written in a HLL must be first translated into object code by an optimizing compiler.
- The *compiler* assigns variables to registers or to memory words and reserves functional units for operators.
- An *assembler* is used to translate the compiled object code into machine code which can be recognized by the machine hardware. A *loader* is used to initiate the program execution through the OS kernel.

- Compiler Support

There are three compiler upgrade approaches:

- **Preprocessor:** A preprocessor uses a sequential compiler and a low-level library of the target computer to implement high-level parallel constructs.
- **Precompiler:** The precompiler approach requires some program flow analysis, dependence checking, and limited optimizations toward parallelism detection.
- **Parallelizing Compiler:** This approach demands a fully developed parallelizing or vectorizing compiler which can automatically detect parallelism in source code and transform sequential codes into parallel constructs.

1.1.3 Evolution of Computer Architecture

- The study of computer architecture involves both hardware organization and programming/software requirements.
- As seen by an assembly language programmer, computer architecture is abstracted by its instruction set, which includes opcode (operation codes), addressing modes, registers, virtual memory, etc.
- From the hardware implementation point of view, the abstract machine is organized with CPUs, caches, buses, microcode, pipelines, physical memory, etc.

- Therefore, the study of architecture covers both instruction-set architectures and machine implementation organizations.

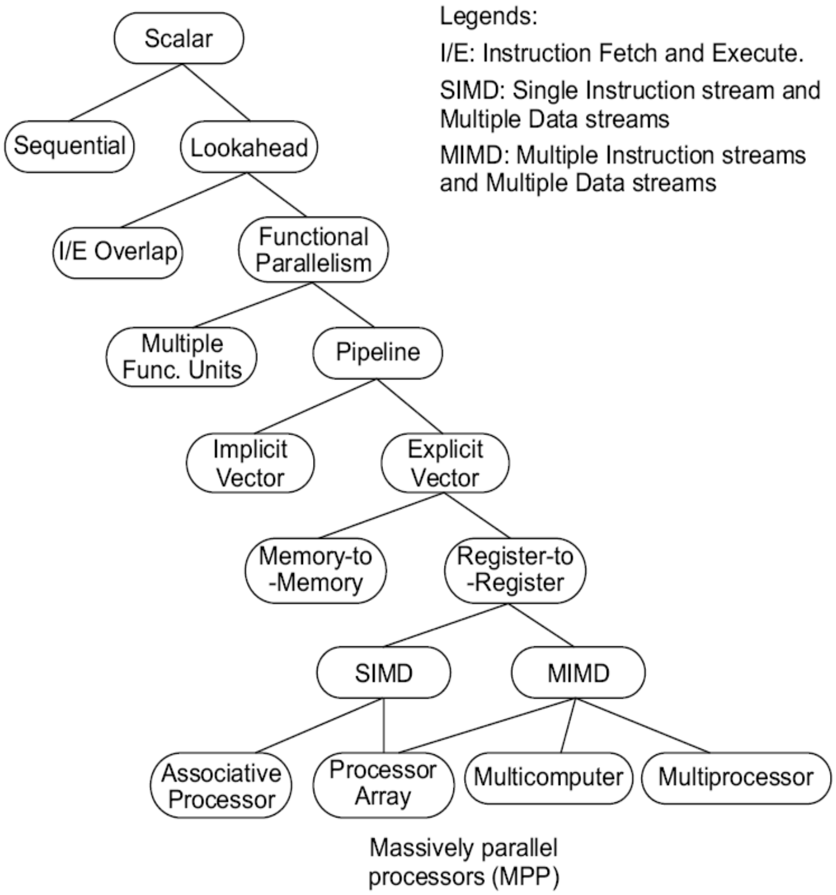


Fig. 1.2 Tree showing architectural evolution from sequential scalar computers to vector processors and parallel computers

Lookahead, Parallelism, and Pipelining

Lookahead techniques were introduced to prefetch instructions in order to overlap I/E (instruction fetch/decode and execution) operations and to enable functional parallelism. Functional parallelism was supported by two approaches:

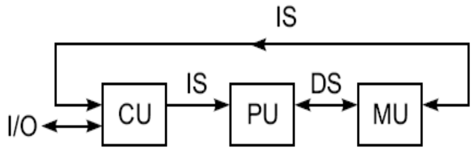
1. using multiple functional units simultaneously,
2. to practice pipelining at various processing levels.

Flynn's Classification

Michael Flynn (1972) introduced a classification of various computer architectures based on notions of instruction and data streams.

1. **SISD** (Single Instruction stream over a Single Data stream) computers
2. **SIMD** (Single Instruction stream over Multiple Data streams) machines
3. **MIMD** (Multiple Instruction streams over Multiple Data streams) machines.
4. **MISD** (Multiple Instruction streams and a Single Data stream) machines

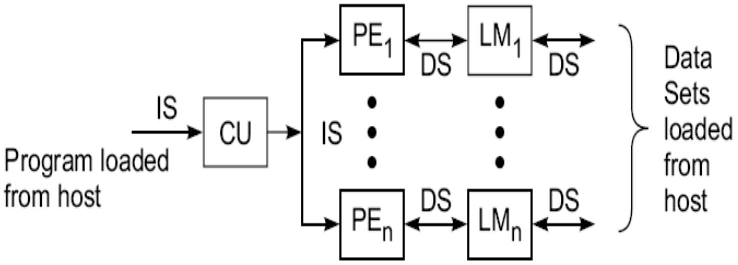
1. SISD (Single Instruction stream over a Single Data stream) computers



(a) SISD uniprocessor architecture

- Conventional sequential machiunes are called SISD computers.
- They are also called scalar processor i.e., one instruction at a time and each instruction have only one set of operands.
- Single instruction: only one instruction stream is being acted on by the CPU during any one clock cycle
- Single data: only one data stream is being used as input during any one clock cycle
- Deterministic execution
- Instructions are executed sequentially.
- This is the oldest and until recently, the most prevalent form of computer
- Examples: most PCs, single CPU workstations and mainframes

2. SIMD (Single Instruction stream over Multiple Data streams) machines

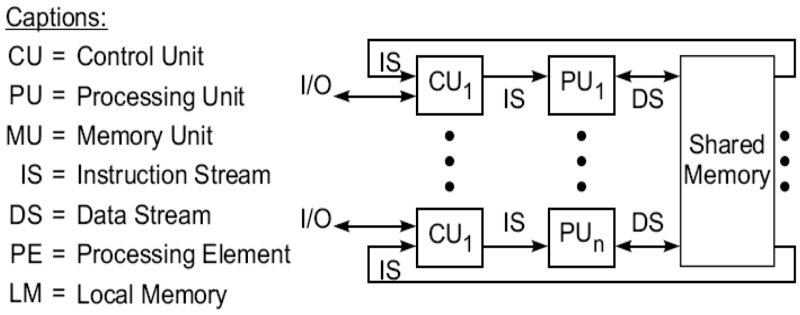


(b) SIMD architecture (with distributed memory)

A type of parallel computer

- **Single instruction**: All processing units execute the same instruction issued by the control unit at any given clock cycle.
- **Multiple data:** Each processing unit can operate on a different data element. The processors are connected to shared memory or interconnection network providing multiple data to processing unit.
- This type of machine typically has an instruction dispatcher, a very high-bandwidth internal network, and a very large array of very small-capacity instruction units.
- Thus single instruction is executed by different processing unit on different set of data.
- Best suited for specialized problems characterized by a high degree of regularity, such as image processing and vector computation.
- Synchronous (lockstep) and deterministic execution.
- Two varieties: Processor Arrays e.g., Connection Machine CM-2, Maspar MP-1, MP-2 and Vector Pipelines processor e.g., IBM 9000, Cray C90, Fujitsu VP, NEC SX-2, Hitachi S820

3. MIMD (Multiple Instruction streams over Multiple Data streams) machines.



(c) MIMD architecture (with shared memory)

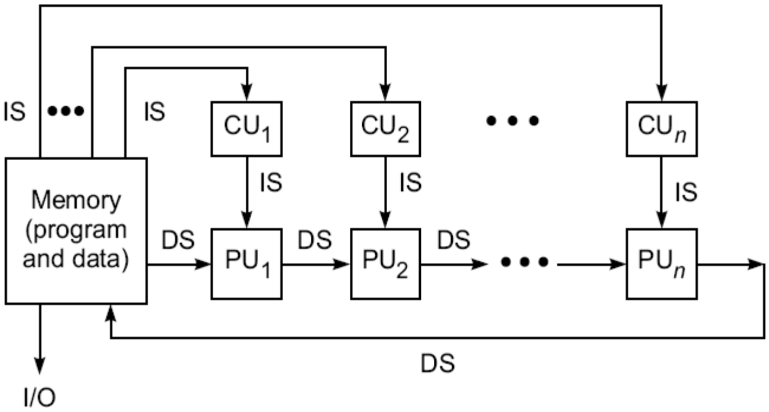
- A single data stream is fed into multiple processing units.
- Each processing unit operates on the data independently via independent instruction streams.
- A single data stream is forwarded to different processing unit which are connected to different control unit and execute instruction given to it by control unit to which it is attached.
- Thus in these computers same data flow through a linear array of processors executing different instruction streams.
- This architecture is also known as **Systolic Arrays** for pipelined execution of specific instructions.

  Some conceivable uses might be:

  1. multiple frequency filters operating on a single signal stream

  2. multiple cryptography algorithms attempting to crack a single coded message.

4. MISD (Multiple Instruction streams and a Single Data stream) machines



(d) MISD architecture (the systolic array)

Fig. 1.3 Flynn's classification of computer architectures (Derived from Michael Flynn,

- **Multiple Instructions:** Every processor may be executing a different instruction stream
- **Multiple Data:** Every processor may be working with a different data stream, multiple data stream is provided by shared memory.
- Can be categorized as loosely coupled or tightly coupled depending on sharing of data and control.
- Execution can be synchronous or asynchronous, deterministic or non-deterministic
- There are multiple processors each processing different tasks.
- Examples: most current supercomputers, networked parallel computer "grids" and multi-processor SMP computers - including some types of PCs.

Development Layers

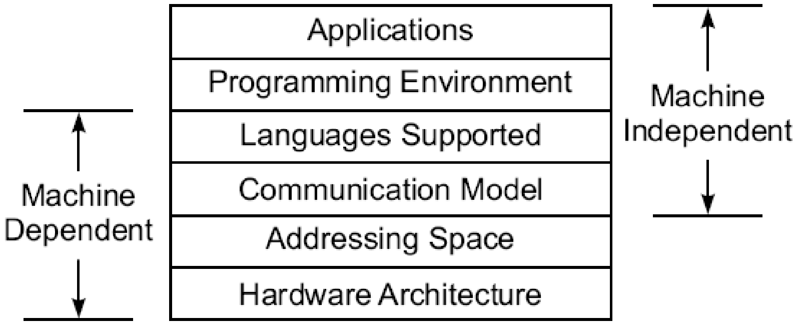


Fig. 1.4 Six layers for computer system development (Courtesy of Lionel Ni, 1990)

Development Layers A layered development of parallel computers is illustrated in Fig. 1.4.

- **Hardware configurations** differ from machine to machine, even those of the same model.
- **Address Space** of a processor in a computer system varies among different architectures. It depends on the memory organization, which is machine-dependent. These features are up to the designer and should match the target application domains.
- We want to develop **Application Programs** and **Programming Environments** which are machine-independent. Independent of machine architecture, the user programs can be ported to many computers with minimum conversion costs.
- **High-level languages** and **Communication Models** depend on the architectural choices made in a computer system. From a programmer's viewpoint, these two layers should be architecture-transparent.
- At present, Fortran, C, Pascal, Ada, and Lisp are supported by most computers.
- However, the **Communication Models**, shared variables versus message passing, are mostly machine-dependent. The Linda approach using *tuple spaces* offers architecture transparent Communication model for parallel computers.
- Application programmers prefer more architectural transparency. However, kernel programmers have to explore the opportunities supported by hardware.
- As a good computer architect, one has to approach the problem from both ends.
- The compilers and OS support should be designed to remove as many architectural constraints as possible from the programmer.

1.1.4 System Attributes affecting Performance

Clock Rate and CPI

- The CPU (or simply the *processor)* of today's digital computer is driven by a clock with a constant *cycle time* (τ in nanoseconds).
- The inverse of the cycle time is the *clock rate* (/ = 1/ τ in megahertz). The size of a program is determined by its *instruction count* (Ic), in terms of the number of machine instructions to be executed in the program.
- Different machine instructions may require different numbers of clock cycles to execute. Therefore, the *cycles* per *instruction* (CPI) becomes an important parameter for measuring the time needed to execute each instruction.
- For a given instruction set, we can calculate an *average* CPI over all instruction types, provided we know their frequencies of appearance in the program.

- An accurate estimate of the average CPI requires a large amount of program code to be traced over a long period of time.
- Unless specifically focusing on a single instruction type, we simply use the term CPI to mean the average value with respect to a given instruction set and a given program mix.

Performance Factors

- Let Ic be the number of instructions in a given program, or the instruction count.
- The CPU time *(T* in seconds/program) needed to execute the program is estimated by finding the product of three contributing factors:

$$\mathbf{T = Ic \times CPI \times \tau} \qquad (1.1)$$

- The execution of an instruction requires going through a cycle of events involving the instruction fetch, decode, operand(s) fetch, execution, and store results.
- In this cycle, only the instruction decode and execution phases are carried out in the CPU.
- The remaining three operations may be required to access the memory. We define a *memory cycle* as the time needed to complete one memory reference.
- Usually, a memory cycle is *k* times the processor cycle τ.
- The value of *k* depends on the speed of the memory technology and processor-memory interconnection scheme used.
- The CPI of an instruction type can be divided into two component terms corresponding to the total processor cycles and memory cycles needed to complete the execution of the instruction.
- Depending on the instruction type, the complete instruction cycle may involve one to four memory references (one for instruction fetch, two for operand fetch, and one for store results). Therefore we can rewrite Eq. 1.1 as follows;

$$\mathbf{T = I_c \times (p + m \times k) \times \tau} \qquad (1.2)$$

where p is the number of processor cycles needed for the instruction decode and execution,

m is the number of memory references needed,

k is the ratio between memory cycle and processor cycle,

I_c is the instruction count,

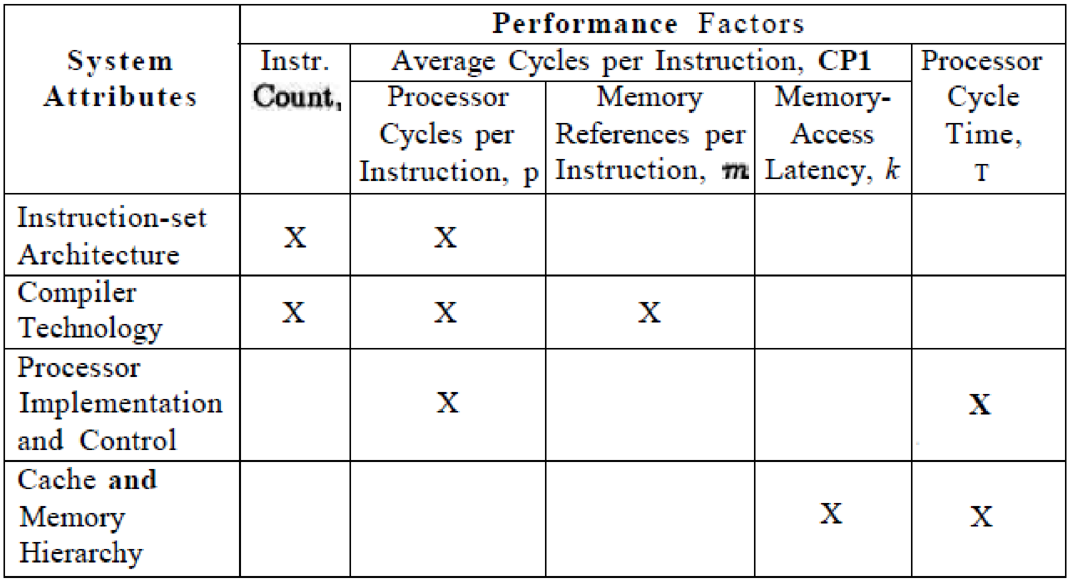
r is the processor cycle time.

Equation 1.2 can be further refined once the CPi components *(p,m,k)* are weighted over the entire instruction set.

System Attributes

- The above five performance factors (I_c, p, *m*, *k*, τ) are influenced by four system attributes: instruction-set architecture, compiler technology, CPU implementation and control, and cache and memory hierarchy, as specified in Table 1.2.
- The instruction-set architecture affects the program length (Ic) and processor cycle needed (p). The compiler technology affects the values of I_c, p and the memory reference count (m).
- The CPU implementation and control determine the total processor time *(p. τ)* needed.
- Finally, the memory technology and hierarchy design affect the memory access latency *(k.* τ). The above CPU time can be used as a basis in estimating the execution rate of a processor.

Table 1.2 Performance Factors Versus System Attributes



System Attributes	Performance Factors				
	Instr. **Count,**	Average Cycles per Instruction, CP1			Processor Cycle Time, T
		Processor Cycles per Instruction, p	Memory References per Instruction, *m*	Memory-Access Latency, *k*	
Instruction-set Architecture	X	X			
Compiler Technology	X	X	X		
Processor Implementation and Control		X			X
Cache **and** Memory Hierarchy				X	X

MIPS Rate

- Let *C* be the total number of clock cycles needed to execute a given program.
- Then the CPU time in Eq. 1.2 can be estimated as $T = C \times \tau = C/f$.
- Furthermore, CPI = C/I_c and $T = I_c \times \text{CPI} \times \tau = I_c \times \text{CPI}/f$. The processor speed is often measured in terms of *million instructions per second* (MIPS).
- We simply call it the MIPS rate of a given processor. It should be emphasized that the MIPS rate varies with respect to a number of factors, including the clock rate (f), the instruction count (I_c), and the CPI of a given machine, as defined below:

$$\text{MIPS rate} = \frac{I_c}{T \times 10^6} = \frac{f}{\text{CPI} \times 10^6} = \frac{f \times I_c}{C \times 10^6} \qquad (1.3)$$

- Based on Eq. 1.3, the CPU time in Eq. 1.2 can also be written as **$T = I_c$ X 10^{-6} / MIPS**.
- Based on the system attributes identified in Table 1.2 and the above derived expressions, we conclude by indicating the fact that the MIPS rate of a given computer is directly proportional to the clock rate and inversely proportional to the CPI.
- All four system attributes, instruction set, compiler, processor, and memory technologies, affect the MIPS rate, which varies also from program to program.

Throughput Rate

- Number of programs a system can execute per unit time, called the *system throughput Ws* (in programs/second).
- In a multiprogrammed system, the system throughput is often lower than the *CPU throughput Wp* defined by:

$$W_p = \frac{f}{I_c \times \text{CPI}} \qquad (1.4)$$

- Note that W_p = (MIPS) X $10^6/I_c$ from Eq. 1.3- The unit for W_p is programs/second.
- The CPU throughput is a measure of how many programs can be executed per second, based on the MIPS rate and average program length (I_c).
- The reason why *Ws* < *Wp* is due to the additional system overheads caused by the I/O, compiler, and OS when multiple programs are interleaved for CPU execution by multiprogramming or timesharing operations.
- If the CPU is kept busy in a perfect program-interleaving fashion, then *Ws* = *Wp*. This will probably never happen, since the system overhead often causes an extra delay and the CPU may be left idle for some cycles.

Programming Environments

- Programmability depends on the programming environment provided to the users.
- Conventional computers are used in a sequential programming environment with tools developed for a uniprocessor computer.
- Parallel computers need parallel tools that allow specification or easy detection of parallelism and operating systems that can perform parallel scheduling of concurrent events, shared memory allocation, and shared peripheral and communication links.

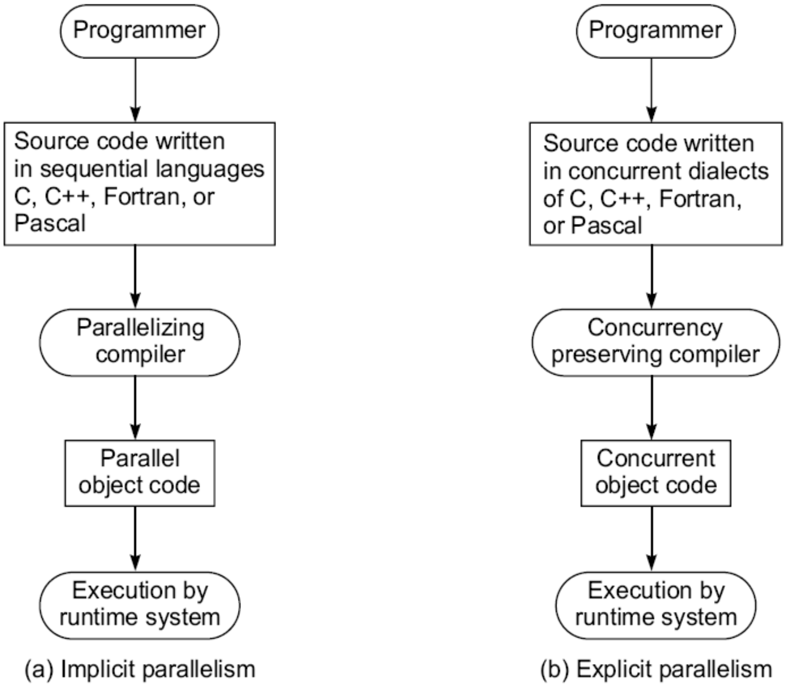

Programmer
Source code written in sequential languages C, C++, Fortran, or Pascal
Parallelizing compiler
Parallel object code
Execution by runtime system
(a) Implicit parallelism
Programmer
Source code written in concurrent dialects of C, C++, Fortran, or Pascal
Concurrency preserving compiler
Concurrent object code
Execution by runtime system
(b) Explicit parallelism

Fig. 1.5 Two approaches to parallel programming (Courtesy of Charles Seitz; adapted with permission from

Implicit Parallelism

- An implicit approach uses a conventional language, such as C, Fortran, Lisp, or Pascal, to write the source program.
- The sequentially coded source program is translated into parallel object code by a parallelizing compiler.
- The compiler must be able to detect parallelism and assign target machine resources. This compiler approach has been applied in programming shared-memory multiprocessors.
- With parallelism being implicit, success relies heavily on the "intelligence" of a parallelizing compiler.
- This approach requires less effort on the part of the programmer.

Explicit Parallelism

- The second approach (Fig. 1.5b) requires more effort by the programmer to develop a source program **using parallel** dialects of C, Fortran, Lisp, or Pascal.
- Parallelism is explicitly specified in the user programs.
- This will significantly reduce the burden on the compiler to detect parallelism.
- Instead, the compiler needs to preserve parallelism and, where possible, assigns target machine resources.

1.2 Multiprocessors and Multicomputers

Two categories of parallel computers are architecturally modeled below. These physical models are distinguished by having a shared common memory or unshared distributed memories.

1. Shared-Memory Multiprocessors

There are 3 shared-memory multiprocessor models:

i. Uniform Memory-access (UMA) model,

ii. Non uniform-Memory-access (NUMA) model

iii. Cache-Only Memory Architecture (COMA) model.

These models differ in how the memory and peripheral resources are **shared** or distributed.

i. Uniform Memory-Access (UMA) model

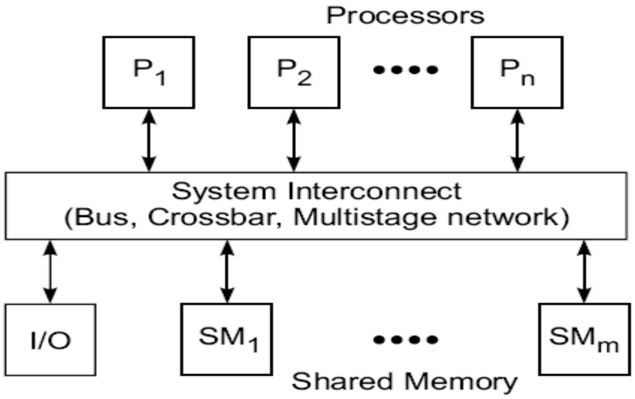


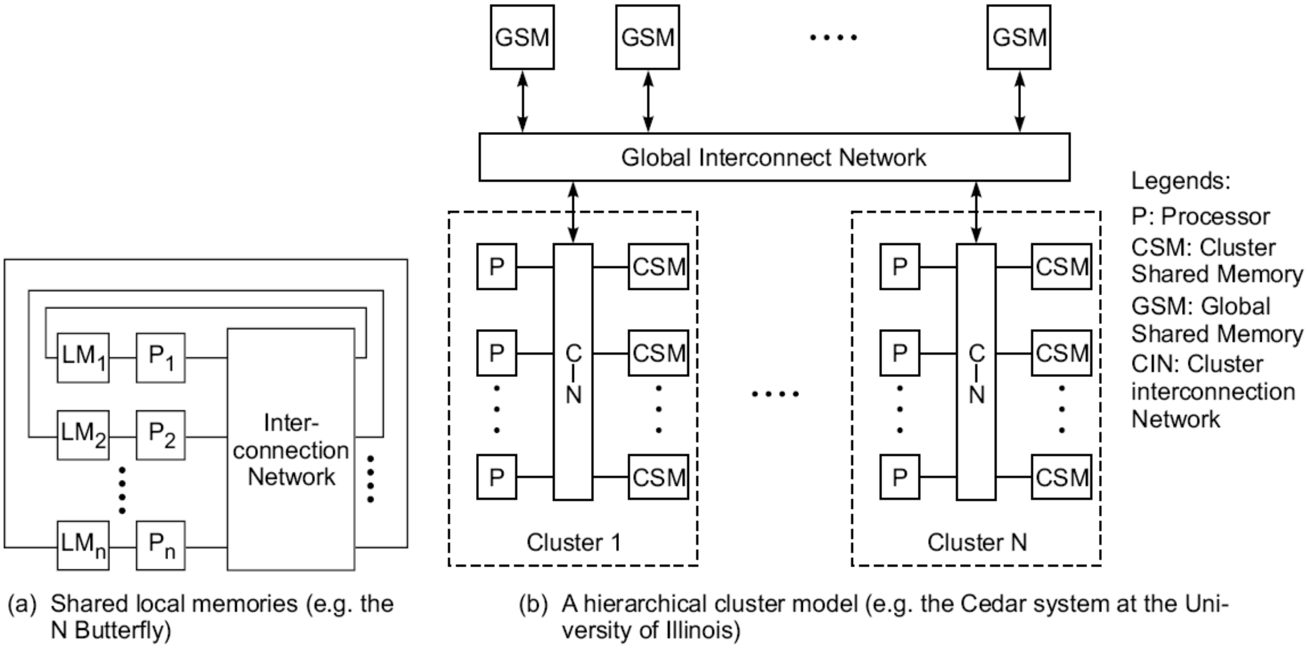
Fig. 1.6 The UMA multiprocessor model

- In a UMA multiprocessor model (Fig. 1.6), the physical memory is uniformly shared by all the processors.
- All processors have equal access time to all memory words, which is why it is called uniform memory access.
- Each processor may use a private cache. Peripherals are also shared in some fashion.
- Multiprocessors are called *tightly coupled systems* dun to the high degree of resource sharing. The system interconnect takes the form of a common bus, a crossbar switch, or a multistage network.
- Most computer manufacturers have *multiprocessor* (MP) extensions of their *uniprocessor*
- (UP) product line.
- The UMA model is suitable for general-purpose and timesharing applications by **multiple** users. It can be used to speed up the execution of a single large program in time-critical applications. To

coordinate parallel events, synchronization and communication among processors are done through using shared variables in the common memory.

- When all processors have equal access to all peripheral devices, the system is called a **symmetric multiprocessor.** In this case, all the processors are equally capable of running the executive programs, such as the OS kernel and I/O service routines.

ii. Non uniform-Memory-Access (NUMA) model


GSM
GSM
GSM
Global Interconnect Network
P
CSM
CIN
Cluster 1
Cluster N
LM1
P1
LM2
P2
LMn
Pn
Inter-connection Network
Legends:
P: Processor
CSM: Cluster Shared Memory
GSM: Global Shared Memory
CIN: Cluster interconnection Network

(a) Shared local memories (e.g. the N Butterfly)

(b) A hierarchical cluster model (e.g. the Cedar system at the University of Illinois)

Fig. 1.7 Two NUMA models for multiprocessor systems

- A NUMA multiprocessor is a shared-memory system in which the access time varies with the location of the memory word.
- Two NUMA machine models are depicted in Fig. 1.7.
- The shared memory is physically distributed to all processors, called *local memories.*
- The collection of all local memories forms a global address space accessible by all processors.
- It is faster to access a local memory with a local processor. The access of remote memory attached to other processors takes longer due to the added delay through the interconnection network.
- The BBN TC-2000 Butterfly multiprocessor assumes the configuration shown in Fig. 1.7a.

iii. Cache-Only Memory Architecture (COMA) model

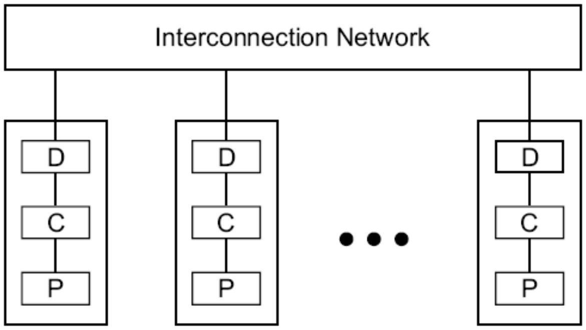


Fig. 1.8 The COMA model of a multiprocessor (P: Processor, C: Cache, D: Directory; e.g. the KSR-1)

- A multiprocessor using cache-only memory assumes the COMA model.
- Examples of COMA machines include the Swedish Institute of Computer Science's Data Diffusion Machine and Kendall Square Research's KSR-1 machine.
- The COMA model is a special case of a NUMA machine, in which the distributed main memories are converted to caches.
- There is no memory hierarchy at each processor node. All the caches form a global address space.
- Remote cache access is assisted by the distributed cache directories (D in Fig. 1.8).
- Depending on the interconnection network used, sometimes hierarchical directories may be used to help locate copies of cache blocks.
- Initial data placement is not critical because data will eventually migrate to where it will be used.

2. Distributed-Memory Multicomputers

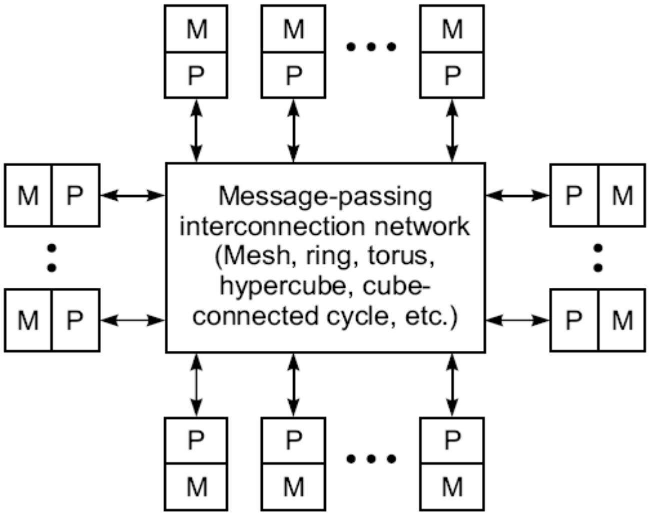

M
P
M
P
M
P
M P
M P
Message-passing interconnection network (Mesh, ring, torus, hypercube, cube-connected cycle, etc.)
P M
P M
P
M
P
M
P
M

Fig. 1.9 Generic model of a message-passing multicomputer

- A distributed-memory multicomputer system is modeled in the above figure consists of multiple computers, often called *nodes,* interconnected by a message-passing network.
- Each node is an autonomous computer consisting of a processor, local memory, and sometimes attached disks or I/O peripherals.
- The message-passing network provides point-to-point static connections among the nodes.
- All local memories are private and are accessible only by local processors.
- For this reason, traditional multicomputers have been called *no-remote-memory-access* (NORMA) machines.
- However, this restriction will gradually be removed in future multi computers with distributed shared memories. Internode communication is carried out by passing messages through the static connection network.

1.3 Multivector and SIMD Computers

We can classify super computers as:

i. **Pipelined vector machines** using a few powerful processors equipped with vector hardware

ii. **SIMD computers** emphasizing massive data parallelism

1.3.1 Vector Supercomputers

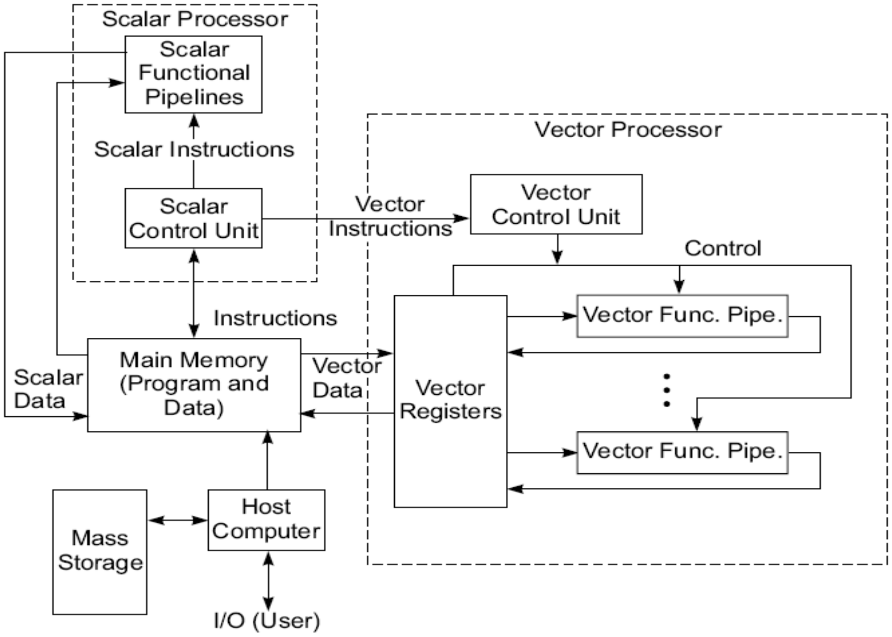


Fig. 1.11 The architecture of a vector supercomputer

- A vector computer is often built on top of a scalar processor.
- As shown in Fig. 1.11, the vector processor is attached to the scalar processor as an optional feature.
- Program and data are first loaded into the main memory through a host computer.
- All instructions are first decoded by the scalar control unit.
- If the decoded instruction is a scalar operation or a program control operation, it will be directly executed by the scalar processor using the scalar functional pipelines.
- If the instruction is decoded as a vector operation, it will be sent to the vector control unit.
- This control unit will supervise the flow of vector data between the main memory and vector functional pipelines.
- The vector data flow is coordinated by the control unit. A number of vector functional pipelines may be built into a vector processor.

Vector Processor Models

- Figure l.ll shows a **register-to-register** architecture.
- Vector registers are used to hold the vector operands, intermediate and final vector results.
- The vector functional pipelines retrieve operands from and put results into the vector registers.
- All vector registers are programmable in user instructions.
- Each vector register is equipped with a component counter which keeps track of the component registers used in successive pipeline cycles.
- The length of each vector register is usually fixed, say, sixty-four 64-bit component registers in a vector register in a Cray Series supercomputer.
- Other machines, like the Fujitsu VP2000 Series, use reconfigurable vector registers to dynamically match the register length with that of the vector operands.

1.3.2 SIMD Supercomputers

SIMD computers have a single instruction stream over multiple data streams.

An operational model of an SIMD computer is specified by a 5-tuple:

M = (N,C, I,M, R)

where

1. ***N*** is the number of *processing elements* (PEs) in the machine. For example, the Illiac IV had 64 PEs and the Connection Machine CM-2 had 65,536 PEs.

2. ***C*** is the set of instructions directly executed by the *control unit* (CU), including scalar and program flow control instructions.
3. **I** s the set of instructions broadcast by the CU to all PEs for parallel execution. These include arithmetic, logic, data routing, masking, and other local operations executed by each active PE over data within that PE.
4. **M** is the set of masking schemes, where each mask partitions the set of PEs into enabled and disabled subsets.

R is the set of data-routing functions, specifying various patterns to be set up in the interconnection network for inter-PE communications.

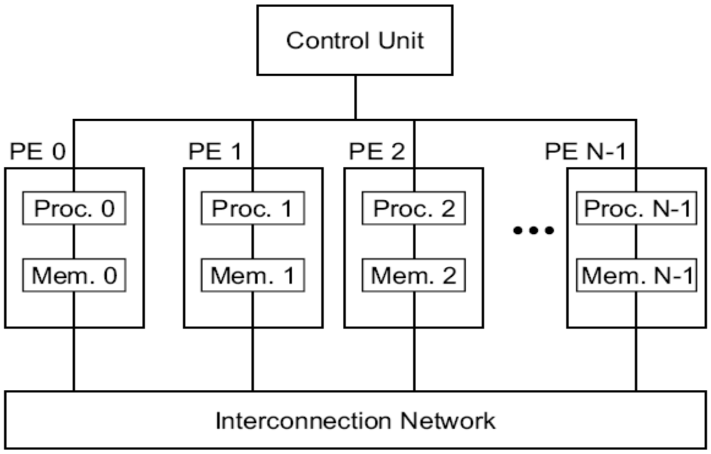


Fig. 1.12 Operational model of SIMD computers

1.4 PRAM AND VLSI MODELS

1.4.1 PRAM model (Parallel Random Access Machine)

- PRAM is a theoretical model of parallel computation in which an arbitrary but finite number of processors can access any value in an arbitrarily large shared memory in a single time step.
- Processors may execute different instruction streams, but work synchronously. This model assumes a shared memory, multiprocessor machine as shown:
- The machine size n can be arbitrarily large
- The machine is synchronous at the instruction level. That is, each processor is executing it's own series of instructions, and the entire machine operates at a basic time step (cycle). Within each cycle, each processor executes exactly one operation or does nothing, i.e. it is idle.

- An instruction can be any random access machine instruction, such as: fetch some operands from memory, perform an ALU operation on the data, and store the result back in memory.
- All processors implicitly synchronize on each cycle and the synchronization overhead is assumed to be zero.
- Communication is done through reading and writing of shared variables.
- Memory access can be specified to be UMA, NUMA, EREW, CREW, or CRCW with a defined conflict policy.
- The PRAM model can apply to SIMD class machines if all processors execute identical instructions on the same cycle or to MIMD class machines if the processors are executing different instructions.
- Load imbalance is the only form of overhead in the PRAM model.

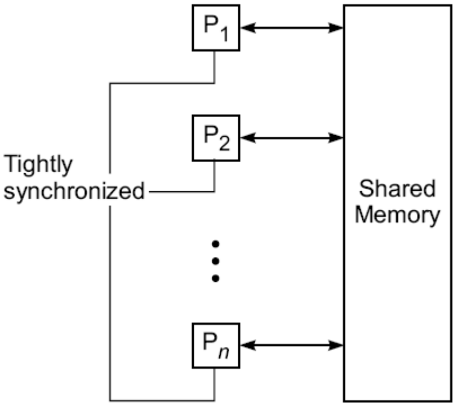


Fig. 1.14 PRAM model of a multiprocessor system with shared memory, on which all *n* processors operate in lockstep in memory access and program execution operations. Each processor can access any memory location in unit time

An n-processor PRAM (Fig. 1.14) has a globally addressable memory.

The shared memory can be distributed among the processors or centralized in one place. The n processors operate on a synchronized read-memory, compute, and write-memory cycle. With shared memory, the model must specify how concurrent read and concurrent write of memory are handled.

Four memory-update options are possible:

- **Exclusive Read (ER)** — This allows at mast one processor to read from any memory location in each cycle, a rather restrictive policy.
- **Exclusive Write (EW)** — This allows at most one processor to write into a memory location at a time.
- **Concurrent Read (CR)** — This allows multiple processors to read the same information from the same memory cell in the same cycle.

- **Concurrent Write (CW)** — This allows simultaneous writes to the same memory location. In order to avoid confusion, some policy must be set up to resolve the write conflicts. Various combinations of the above options lead to several variants of the PRAM model as specified below.

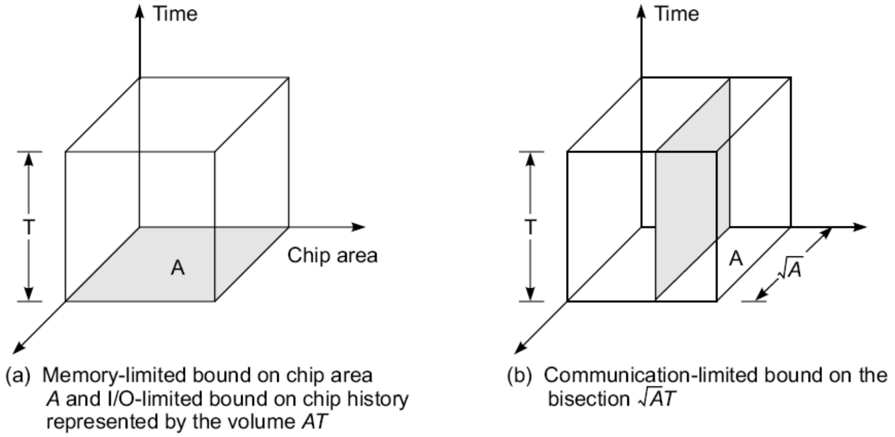
PRAM Variants

There are 4 variants of the PRAM model, depending on how the memory reads and writes are handled.

- **EREW – PRAM Model (Exclusive Read, Exclusive Write):** This model forbids more than one processor from reading or writing the same memory cell simultaneously. This is the most restrictive PRAM model proposed.
- **CREW – PRAM Model (Concurrent Read, Exclusive Write);** The write conflicts are avoided by mutual exclusion. Concurrent reads to the same memory location arc allowed.
- **ERCW – PRAM Model** – This allows exclusive read or concurrent writes to the same memory location.
- **CRCW – PRAM Model (Concurrent Read, Concurrent Write);** This model allows either concurrent reads or concurrent writes to the same memory location.

1.4.2 VLSI Model

Parallel computers rely on the use of VLSI chips to fabricate the major components such as processor arrays memory arrays and large scale switching networks. The rapid advent of very large scale intergrated (VSLI) technology now computer architects are trying to implement parallel algorithms directly in hardware. An AT2 model is an example for two dimension VLSI chips



(a) Memory-limited bound on chip area *A* and I/O-limited bound on chip history represented by the volume *AT*

(b) Communication-limited bound on the bisection $\sqrt{AT}$

Fig. 1.15 The AT^2 complexity model of two-dimensional VLSI chips

Chapter 2: Program & Network properties

2.1 Condition of parallelism

2.2.1 Data and Resource Dependence

- The ability to execute several program segments in parallel requires each segment to be independent of the other segments. We use a dependence graph to describe the relations.
- The nodes of a dependence graph correspond to the program statement (instructions), and directed edges with different labels are used to represent the ordered relations among the statements.
- The analysis of dependence graphs shows where opportunity exists for parallelization and vectorization.

Data dependence:

The ordering relationship between statements is indicated by the data dependence. Five type of data dependence are defined below:

1. **Flow dependence:** A statement S2 is flow dependent on S1 if an execution path exists from s1 to S2 and if at least one output (variables assigned) of S1feeds in as input (operands to be used) to S2 also called RAW $\mathbf{S}_1 \rightarrow \mathbf{S}_2$ hazard and denoted as

2. **Antidependence:** Statement S2 is antidependent on the statement S1 if S2 follows S1 in the program order and if the output of S2 overlaps the input to S1 also called RAW hazard and denoted as $\mathbf{S}_1 \mapsto \mathbf{S}_2$

3. **Output dependence:** Two statements are output dependent if they produce (write) the same output variable. Also called WAW hazard and denoted as $\mathbf{S}_1 \circ\!\!\rightarrow \mathbf{S}_2$

4. **I/O dependence:** Read and write are I/O statements. I/O dependence occurs not because the same variable is involved but because the same file referenced by both I/O statement.

5. **Unknown dependence:** The dependence relation between two statements cannot be determined in the following situations:

   - The subscript of a variable is itself subscribed (indirect addressing)
   - The subscript does not contain the loop index variable.
   - A variable appears more than once with subscripts having different coefficients of the loop variable.
   - The subscript is non linear in the loop index variable.

Parallel execution of program segments which do not have total data independence can produce non-deterministic results.

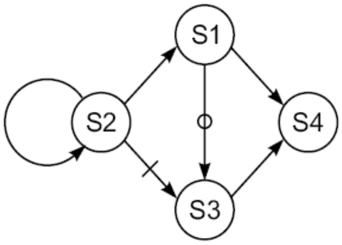
Example: Consider the following fragment of a program:

S1: Load R1, A /R1 ← Memory(A) /

S2: Add R2, R1 /R2 ← (R1) + (R2)/

S3: Move R1, R3 /R1 ← (R3)/

S4: Store B, R1 /Memory(B) ← (R1)/



(a) Dependence graph

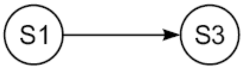
- Here the Flow dependency from **S1 to S2**, **S3 to S4**, **S2 to S2**
- Anti-dependency from **S2 to S3**
- Output dependency **S1 toS3**

S1: Read (4), A(I) /Read array A from file 4/

S2: Rewind (4) /Process data/

S3: Write (4), B(I) /Write array B into file 4/

S4: Rewind (4) /Close file 4/



(b) I/O dependence caused by accessing the same file by the read and write statements

The read/write statements S1 and S2 are I/O dependent on each other because they both access the same file.

Control Dependence:

- This refers to the situation where the order of the execution of statements cannot be determined before run time.
- For example all condition statement, where the flow of statement depends on the output.
- Different paths taken after a conditional branch may depend on the data hence we need to eliminate this data dependence among the instructions.
- This dependence also exists between operations performed in successive iterations of looping procedure. Control dependence often prohibits parallelism from being exploited.

Control-independent example:

```
for (i=0; i<n; i++)
{
        a[i] = c[i];
        if (a[i] < 0) a[i] = 1;
}
```

Control-dependent example:

```
for (i=1; i<n; i++)
{
        if (a[i-1] < 0) a[i] = 1;
}
```

Control dependence also avoids parallelism to being exploited. Compilers are used to eliminate this control dependence and exploit the parallelism.

Resource dependence:

- Data and control dependencies are based on the independence of the work to be done.
- Resource independence is concerned with conflicts in using shared resources, such as registers, integer and floating point ALUs, etc. ALU conflicts are called ALU dependence.
- Memory (storage) conflicts are called storage dependence.

Bernstein's Conditions

Bernstein's conditions are a set of conditions which must exist if two processes can execute in parallel.

Notation

- I_i is the set of all input variables for a process P_i. I_i is also called the read set or domain of *Pi*. *Oi* is the set of all output variables for a process *Pi* . Oi is also called write set.
- If P*1* and P*2* can execute in parallel (which is written as P*1* || P*2*), then:

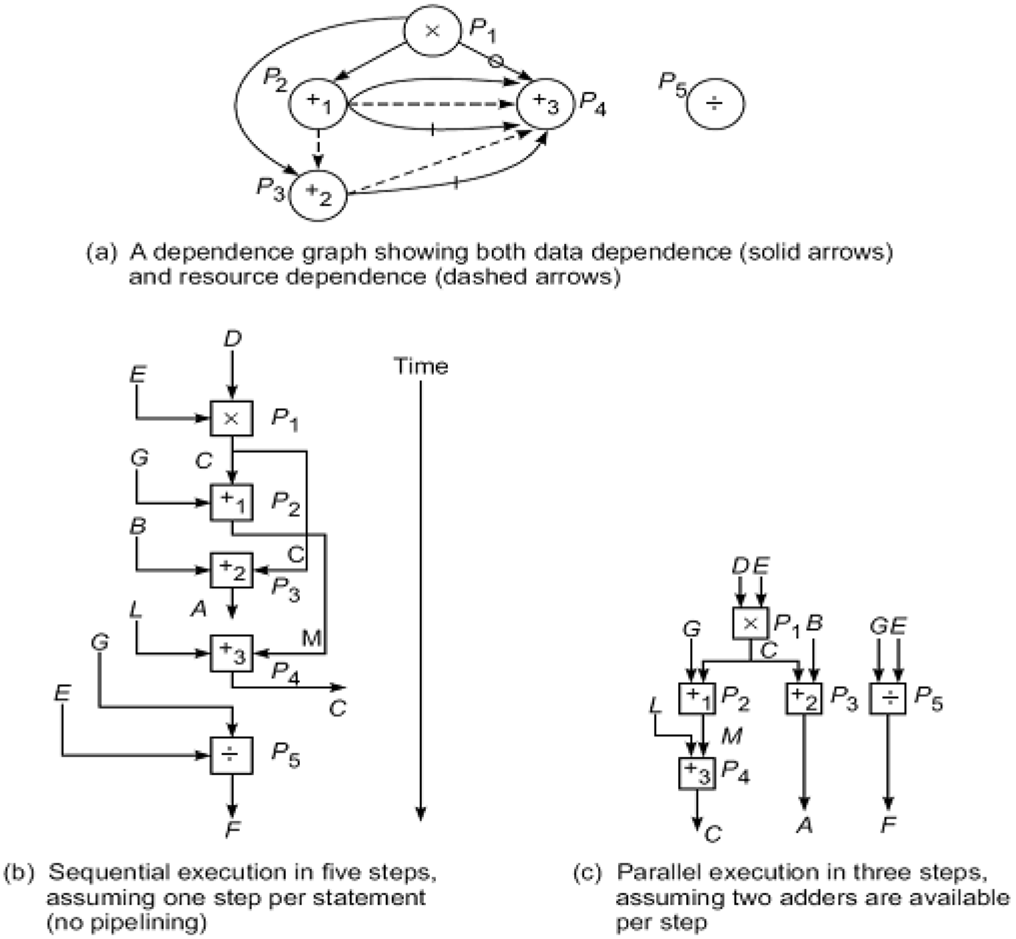
$$
\begin{aligned}
\mathbf{I}_1 \cap \mathbf{O}_2 &= \varnothing \\
\mathbf{I}_2 \cap \mathbf{O}_1 &= \varnothing \\
\mathbf{O}_1 \cap \mathbf{O}_2 &= \varnothing
\end{aligned}
$$

- In terms of data dependencies, Bernstein's conditions imply that two processes can execute in parallel if they are flow-independent, anti-independent, and output-independent.
- The parallelism relation || is commutative (P*i* || P*j* implies P*j* || P*i*), but not transitive (P*i* || P*j* and P*j* || P*k* does not imply P*i* || P*k*) .
- Therefore, || is not an equivalence relation. Intersection of the input sets is allowed.

Example: Detection of parallelism in a program using Bernstein's conditions

Consider the simple case in which each process is a single HLL statement. We want to detect the parallelism embedded in the following 5 statements labeled P1, P2, P3, P4, P5 in program order.

P_1:	C = D x E
P_2:	M = G + C
P_3:	A = B + C
P_4:	C = L + M
P_5:	F = G / E



(a) A dependence graph showing both data dependence (solid arrows) and resource dependence (dashed arrows)

(b) Sequential execution in five steps, assuming one step per statement (no pipelining)

(c) Parallel execution in three steps, assuming two adders are available per step

Fig. 2.2 Detection of parallelism in the program of Example 2.2

- Assume that each statement requires one step to execute. No pipelining is considered here. The dependence graph shown in 2.2a demonstrates flow dependence as well as resource dependence. In sequential execution, five steps are needed (Fig. 2.2b).
- If two adders are available simultaneously, the parallel execution requires only 3 steps as shown in Fig 2.2c.
- Pairwise, there are 10 pairs of statements to check against Bernstein’s conditions. Only 5 pairs, P1||P5, P2||P3, P2||P5, P5||P3 and P4||P5 can execute in parallel as revealed in Fig 2.2a if there are no resource conflicts.
- Collectively, only P2||P3||P5 is possible(Fig. 2.2c) because P2||P3, P3||P5 and P5||P2 are all possible.

2.1.2 Hardware and software parallelism

Hardware parallelism

- Hardware parallelism is defined by machine architecture and hardware multiplicity i.e., functional parallelism times the processor parallelism.
- It can be characterized by the number of instructions that can be issued per machine cycle.
- If a processor issues *k* instructions per machine cycle, it is called a *k-issue* processor.
- Conventional processors are *one-issue* machines.
- This provide the user the information about **peak attainable performance**.

Examples: Intel i960CA is a three-issue processor (arithmetic, memory access, branch).

IBM RS -6000 is a four-issue processor (arithmetic, floating-point, memory access, branch).

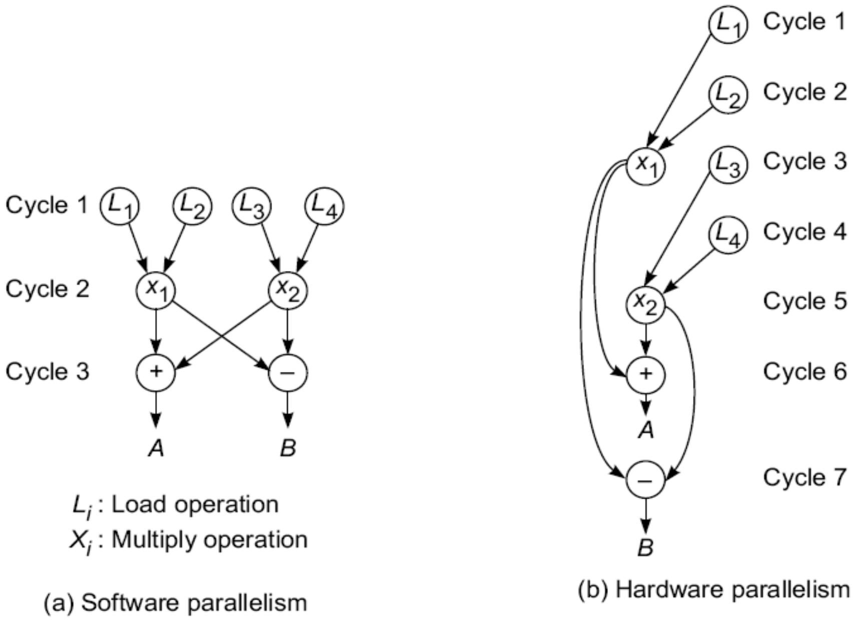
A machine with *n* *k*-issue processors should be able to handle a maximum of *nk* threads simultaneously.

Software Parallelism

Software parallelism is defined by the control and data dependence of programs, and is revealed in the program's flow graph i.e., it is defined by dependencies with in the code and is a function of algorithm, programming style, and compiler optimization.

Example: Mismatch between Software parallelism and Hardware parallelism

- Consider the example program graph in Fig. 2.3a. There are eight instructions (four *loads* and four *arithmetic* operations) to be executed in three consecutive machine cycles.
- Four *load* operations are performed in the first cycle, followed by two *multiply* operations in the second cycle and two *add/subtract* operations in the third cycle.
- Therefore, the parallelism varies from 4 to 2 in three cycles. The average software parallelism is equal to 8/3 = 2.67 instructions per cycle in this example program.
- Now consider execution of the same program by a two-issue processor which can execute one memory access (*load* or write) and one arithmetic *(add, subtract, multiply,* etc.) operation simultaneously.
- With this hardware restriction, the program must execute in seven machine cycles as shown in Fig. 2.3b. Therefore, the *hardware parallelism* displays an average value of 8/7 = 1.14 instructions executed per cycle.
- This demonstrates a mismatch between the software parallelism and the hardware parallelism.



(a) Software parallelism

(b) Hardware parallelism

Fig. 2.3 Executing an example program by a two-issue superscalar processor

- Let us try to match the software parallelism shown in Fig. 2.3a in a hardware platform of a dual-processor system, where single-issue processors are used.
- The achievable hardware parallelism is shown in Fig 2.4. Six processor cycles are needed to execute 12 instructions by two processors.
- S1 and S2 are two inserted store operations, l5 and l6 are two inserted load operations for interprocessor communication through the shared memory.

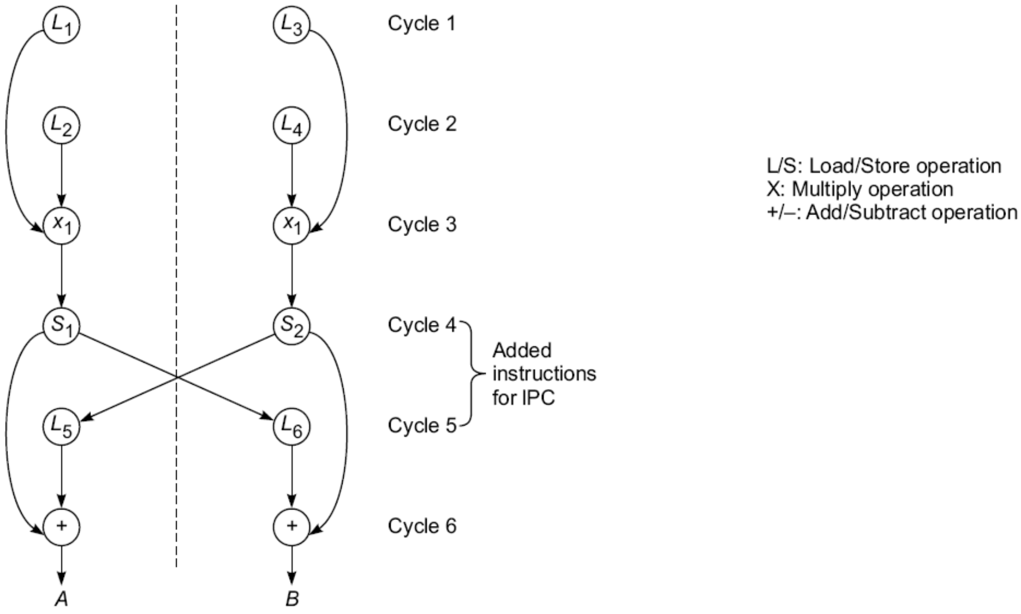


Fig. 2.4 Dual-processor execution of the program in Fig. 2.3a

2.1.3 The Role of Compilers

- Compilers used to exploit hardware features to improve performance. Interaction between compiler and architecture design is a necessity in modern computer development.
- It is not necessarily the case that more software parallelism will improve performance in conventional scalar processors.
- The hardware and compiler should be designed at the same time.

2.2 Program Partitioning & Scheduling

2.2.1 Grain size and latency

- The size of the parts or pieces of a program that can be considered for parallel execution can vary.
- The sizes are roughly classified using the term “granule size,” or simply “granularity.”
- The simplest measure, for example, is the number of instructions in a program part.
- Grain sizes are usually described as fine, medium or coarse, depending on the level of parallelism involved.

Latency

Latency is the time required for communication between different subsystems in a computer. Memory latency, for example, is the time required by a processor to access memory. Synchronization latency is the time required for two processes to synchronize their execution. Computational granularity and communication latency are closely related.

Latency and grain size are interrelated and some general observation are

- As grain size decreases, potential parallelism increases, and overhead also increases.
- Overhead is the cost of parallelizing a task. The principle overhead is communication latency.
- As grain size is reduced, there are fewer operations between communication, and hence the impact of latency increases.
- Surface to volume: inter to intra-node comm.

Levels of Parallelism

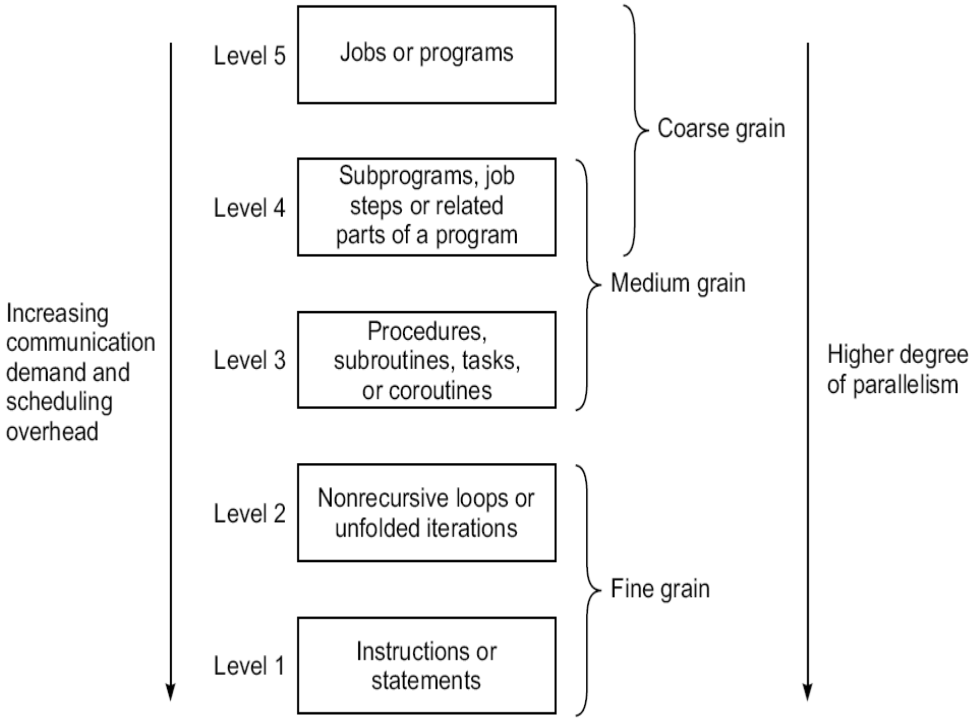

Level 5
Jobs or programs
Level 4
Subprograms, job steps or related parts of a program
Level 3
Procedures, subroutines, tasks, or coroutines
Level 2
Nonrecursive loops or unfolded iterations
Level 1
Instructions or statements
Coarse grain
Medium grain
Fine grain
Increasing communication demand and scheduling overhead
Higher degree of parallelism

Fig. 2.5 Levels of parallelism in program execution on modern computers (Reprinted from Hwang, *Proc. IEEE*, October 1987)

Instruction Level Parallelism

- This fine-grained, or smallest granularity level typically involves less than 20 instructions per grain.
- The number of candidates for parallel execution varies from 2 to thousands, with about five instructions or statements (on the average) being the average level of parallelism.

Advantages:

There are usually many candidates for parallel execution. Compilers can usually do a reasonable job of finding this parallelism

Loop-level Parallelism

- Typical loop has less than 500 instructions. If a loop operation is independent between iterations, it can be handled by a pipeline, or by a SIMD machine.
- Most optimized program construct to execute on a parallel or vector machine.
- Some loops (e.g. recursive) are difficult to handle. Loop-level parallelism is still considered fine grain computation.

Procedure-level Parallelism

- Medium-sized grain; usually less than 2000 instructions.
- Detection of parallelism is more difficult than with smaller grains; interprocedural dependence analysis is difficult and history-sensitive.
- Communication requirement less than instruction level SPMD (single procedure multiple data) is a special case Multitasking belongs to this level.

Subprogram-level Parallelism

- Job step level; grain typically has thousands of instructions; medium- or coarse-grain level.
- Job steps can overlap across different jobs. Multiprograming conducted at this level No compilers available to exploit medium- or coarse-grain parallelism at present.

Job or Program-Level Parallelism

- Corresponds to execution of essentially independent jobs or programs on a parallel computer.
- This is practical for a machine with a small number of powerful processors, but impractical for a machine with a large number of simple processors (since each processor would take too long to process a single job).

Communication Latency

Balancing granularity and latency can yield better performance. Various latencies attributed to machine architecture, technology, and communication patterns used.

Latency imposes a limiting factor on machine scalability.

Ex: Memory latency increases as memory capacity increases, limiting the amount of memory that can be used with a given tolerance for communication latency.

Interprocessor Communication Latency

- Needs to be minimized by system designer
- Affected by signal delays and communication patterns **Ex:** n communicating tasks may require n (n - 1)/2 communication links, and the complexity grows quadratically, effectively limiting the number of processors in the system.

Communication Patterns

- Determined by algorithms used and architectural support provided
- Patterns include permutations broadcast multicast conference
- Tradeoffs often exist between granularity of parallelism and communication demand.

2.2.2 Grain Packing and Scheduling

Two questions:

- How can I partition a program into parallel "pieces" to yield the shortest execution time?
- What is the optimal size of parallel grains?

There is an obvious tradeoff between the time spent scheduling and synchronizing parallel grains and the speedup obtained by parallel execution.

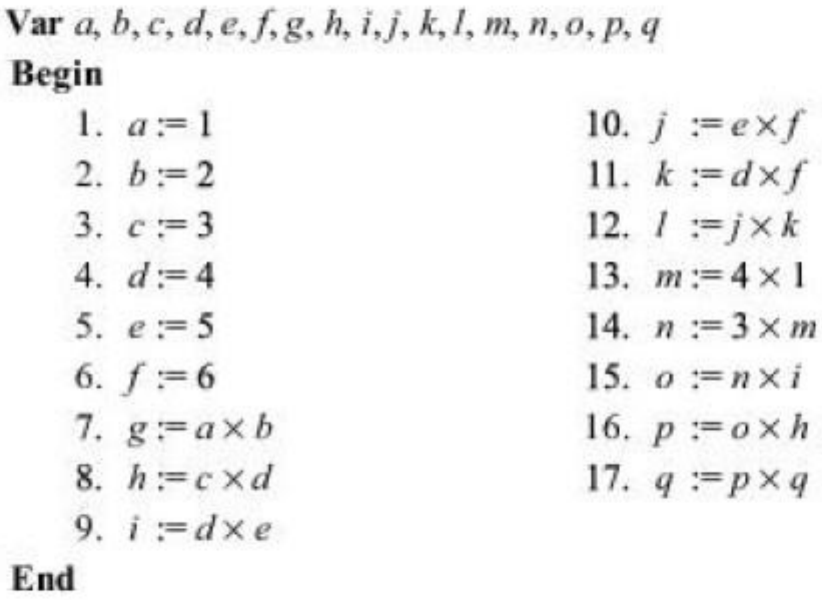
One approach to the problem is called "grain packing."

Program Graphs and Packing (Basic concept of Program Partitioning)

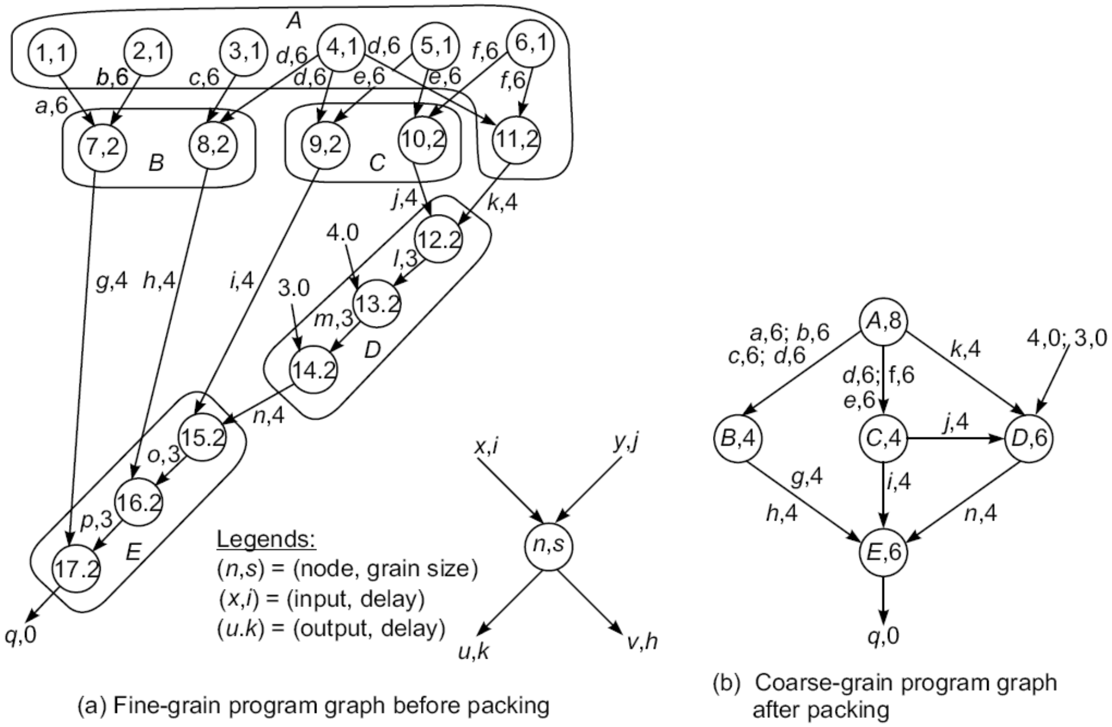
- A program graph shows the structure of the program, similar to dependence graph. Each node in the program graph corresponds to a computational unit in the program.
- Grain size is measured by the number of basic machine cycles needed to execute all the operations within the node.
- Each node is denoted by, **Nodes = { (n,s) }**, where n = node name (id), **s = grain size** (larger s = larger grain size), Fine-grain nodes have a smaller grain size, and coarse-grain nodes have a larger grain size.
- **Edges = { (v,d) }**, where v = variable being "communicated," and d = communication delay.
- Packing two (or more) nodes produces a node with a larger grain size and possibly more edges to other nodes.
- Packing is done to eliminate unnecessary communication delays or reduce overall scheduling overhead.

  Example: Basic concept of Program Partitioning

  Fig. 2.6, shows an example program graph in two different grain sizes.



```
Var a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q
Begin
    1. $a := 1$
    2. $b := 2$
    3. $c := 3$
    4. $d := 4$
    5. $e := 5$
    6. $f := 6$
    7. $g := a \times b$
    8. $h := c \times d$
    9. $i := d \times e$
    10. $j := e \times f$
    11. $k := d \times f$
    12. $l := j \times k$
    13. $m := 4 \times l$
    14. $n := 3 \times m$
    15. $o := n \times i$
    16. $p := o \times h$
    17. $q := p \times q$
End
```



(a) Fine-grain program graph before packing

(b) Coarse-grain program graph after packing

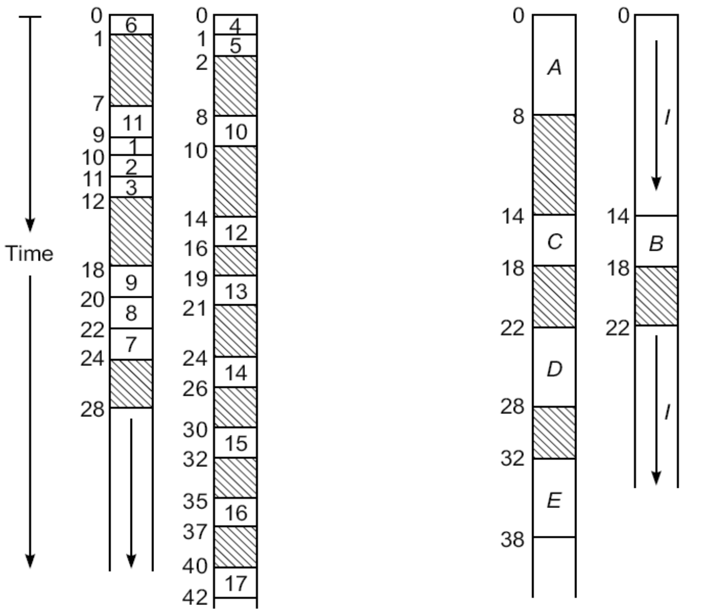
Fig. 2.6 A program graph before and after grain packing in Example 2.4 (Modified from Kruatrachue and Lewis, *IEEE Software*, Jan. 1988)

Scheduling

A schedule is a mapping of nodes to processors and start times such that communication delay requirements are observed, and no two nodes are executing on the same processor at the same time.

Some general scheduling goals are:

- Schedule all fine-grain activities in a node to the same processor to minimize communication delays.
- Select grain sizes for packing to achieve better schedules for a particular parallel machine.



(a) Fine grain (Fig. 2.6a)

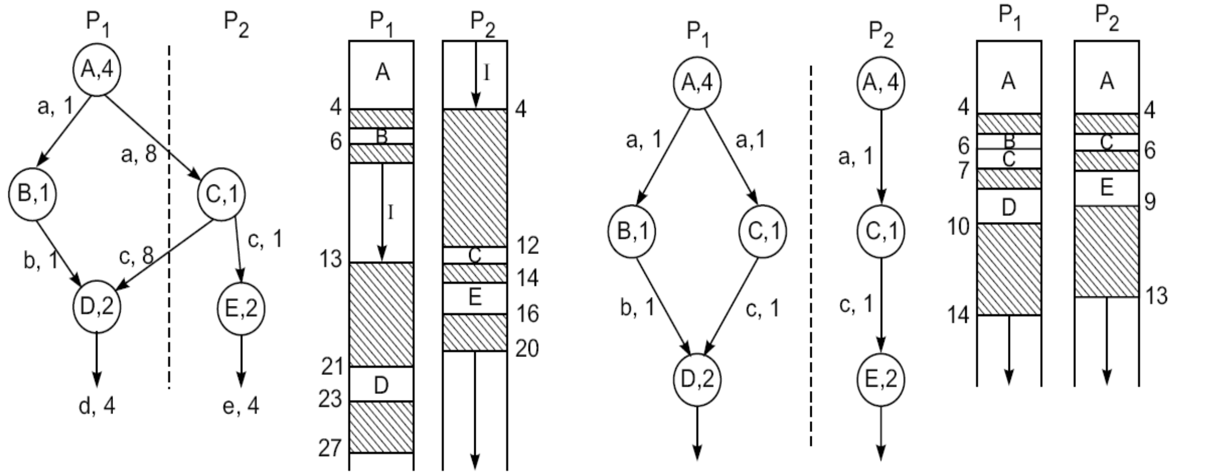
(b) Coarse grain (Fig. 2.6b)

Fig. 2.7 Scheduling of the fine-grain and coarse-grain programs (arrows: idle time; shaded areas: communication delays)

- With respect to the fine-grain versus coarse-grain program graphs in Fig. 2.6, two multiprocessor schedules are shown in Fig. 2.7. The fine-grain schedule is longer (42 time units) because more communication delays were included as shown by the shaded area.
- The coarse-grain schedule is shorter (38 time units) because communication delays among nodes 12, 13 and 14 within the same node D (and also the delays among 15, 16 and 17 within the node E) are eliminated after grain packing.

Node Duplication

- Grain packing may potentially eliminate interprocessor communication, but it may not always produce a shorter schedule.
- By duplicating nodes (that is, executing some instructions on multiple processors), we may eliminate some interprocessor communication, and thus produce a shorter schedule.



(a) Schedule without node duplication

(b) Schedule with node duplication (A → A and A′; C → C and C′)

Fig. 2.8 Node-duplication scheduling to eliminate communication delays between processors (I: idle time; shaded areas: communication delays)

- Figure 2.8a shows a schedule without duplicating any of the 5 nodes. This schedule contains idle time as well as long interprocessor delays (8 units) between P1 and P2.
- In Fig 2.8b, node A is duplicated into A’ and assigned to P2 besides retaining the original copy A in P1.
- Similarly, a duplictaed node C’ is copied into P1 besides the original node C in P2.
- The new schedule is shown in Fig. 2.8b is almost 50% shorter than that in Fig. 2.8a. The reduction in schedule time is caused by elimination of the (a, 8) and (c, 8) delays between the two processors.

Grain packing and node duplication are often used jointly to determine the best grain size and corresponding schedule.

Four major steps are involved in the grain determination and the process of scheduling optimization:\

Step 1: Construct a fine-grain program graph

Step 2: Schedule the fine-grain computation

Step 3: Perform grain packing to produce the coarse grains.

Step 4: Generate a parallel schedule based on the packed graph.

2.3 Program Flow Mechanisms

Control Flow vs. Data Flow

- In Control flow computers the next instruction is executed when the last instruction as stored in the program has been executed where as in Data flow computers an instruction executed when the data (operands) required for executing that instruction is available.

- Control flow machines used shared memory for instructions and data.
- Since variables are updated by many instructions, there may be side effects on other instructions. These side effects frequently prevent parallel processing.
- Single processor systems are inherently sequential.
- Instructions in dataflow machines are unordered and can be executed as soon as their operands are available; data is held in the instructions themselves. *Data tokens* are passed from an instruction to its dependents to trigger execution.

Program Flow Mechanisms

- **Control flow mechanism**: Conventional machines used control flow mechanism in which order of program execution explicitly stated in user programs.
- **Dataflow machines** which instructions can be executed by determining operand availability.
- **Reduction machines** trigger an instruction’s execution based on the demand for its results.

Control flow machines used shared memory for instructions and data. Since variables are updated by many instructions, there may be side effects on other instructions. These side effects frequently prevent parallel processing. Single processor systems are inherently sequential.

Instructions in dataflow machines are unordered and can be executed as soon as their operands are available; data is held in the instructions themselves. *Data tokens* are passed from an instruction to its dependents to trigger execution.

Data Flow Features

No need for

- shared memory
- program counter
- control sequencer

Special mechanisms are required to

- detect data availability
- match data tokens with instructions needing them
- enable chain reaction of asynchronous instruction execution

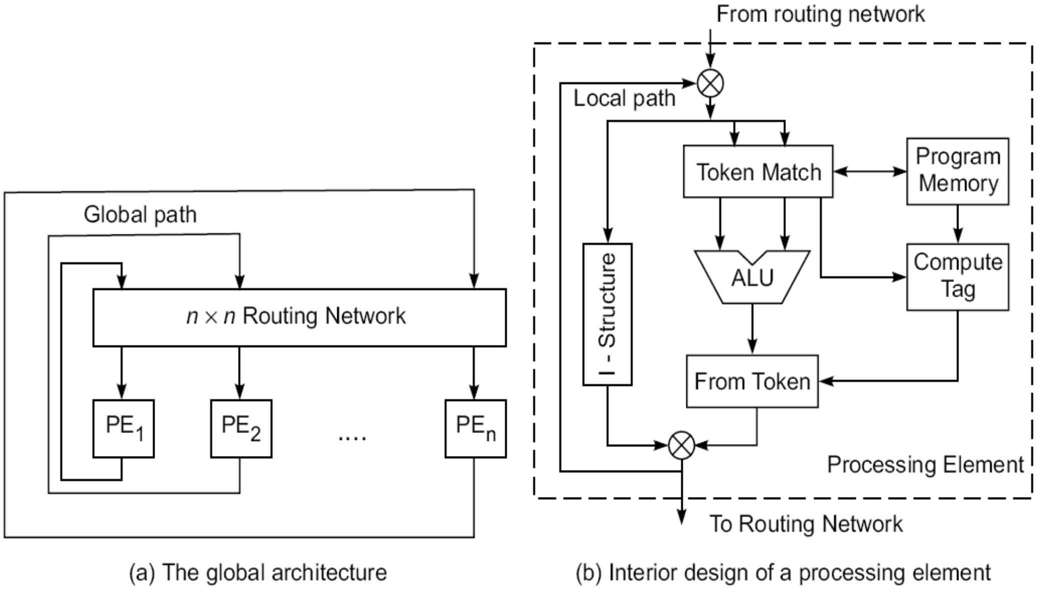


Fig. 2.12 The MIT tagged-token dataflow computer (adapted from Arvind and Iannucci, 1986 with permission)

A Dataflow Architecture

- The Arvind machine (MIT) has N PEs and an N-by-N interconnection network.
- Each PE has a token-matching mechanism that dispatches only instructions with data tokens available.
- Each datum is tagged with
  - address of instruction to which it belongs
  - context in which the instruction is being executed
- Tagged tokens enter PE through local path (pipelined), and can also be communicated to other PEs through the routing network.
- Instruction address(es) effectively replace the program counter in a control flow machine.
- Context identifier effectively replaces the frame base register in a control flow machine.
- Since the dataflow machine matches the data tags from one instruction with successors, synchronized instruction execution is implicit.
- An *I-structure* in each PE is provided to eliminate excessive copying of data structures.
- Each word of the I-structure has a **two-bit tag** indicating whether the value **is empty, full** or **has pending read requests**.
- This is a retreat from the pure dataflow approach.
- Special compiler technology needed for dataflow machines.

Demand-Driven Mechanisms

- Demand-driven machines take a top-down approach, attempting to execute the instruction (a *demander*) that yields the final result.
- This triggers the execution of instructions that yield its operands, and so forth.
- The demand-driven approach matches naturally with functional programming languages (e.g. LISP and SCHEME).

Reduction Machine Models

- **String-reduction model:**
  - each demander gets a separate copy of the expression string to evaluate
  - each reduction step has an operator and embedded reference to demand the corresponding operands
  - each operator is suspended while arguments are evaluated

- **Graph-reduction model:**
  - expression graph reduced by evaluation of branches or subgraphs, possibly in parallel, with demanders given pointers to results of reductions.
  - based on sharing of pointers to arguments; traversal and reversal of pointers continues until constant arguments are encountered.

2.4 System interconnect architecture

Various types of interconnection networks have been suggested for SIMD computers. These are basically classified have been classified on network topologies into two categories namely

1. **Static Networks**
2. **Dynamic Networks**

- Direct networks for static connections
- Indirect networks for dynamic connections
- Networks are used for
  - internal connections in a centralized system among
    - processors
    - memory modules
    - I/O disk arrays
  - distributed networking of multicomputer nodes
- The goals of an interconnection network are to provide
  - low-latency

  - high data transfer rate
  - wide communication bandwidth
- Analysis includes
  - latency
  - bisection bandwidth
  - data-routing functions
  - scalability of parallel architecture

- The topology of an interconnection network can be either static or dynamic. Static networks are formed of point-to-point direct connections which will not change during program execution.
- Dynamic networks are implemented with switched channels, which are dynamically configured to match the communication demand in user programs.
- Packet switching and routing is playing an important role in modern multiprocessor architecture.

Node Degree and Network Diameter:

- The number of edges (links or channels) incident on a node is called the node degree d.
- In the case of unidirectional channels, the number of channels into a node is the in degree, and that out of a node is the out degree.
- Then the node degree is the sum of the two. The node degree reflects the number of IO ports required per node, and thus the cost of a node.
- Therefore, the node degree should be kept a (small) constant, in order to reduce cost.
- The Diameter D of a network is the maximum shortest path between any two nodes.
- The path length is measured by the number of links traversed.
- The network diameter indicates the maximum number of distinct hops between any two nodes, thus providing a figure of communication merit for the network.
- Therefore, the network diameter should be as small as possible from a communication point of view.

Bisection Width:

When a given network is cut into two equal halves, the minimum number of edges (channels) along the cut is called the bisection width b. In the case of a communication network, each edge may correspond to a channel with w bit wires.

To summarize the above discussions, the performance of an interconnection network is affected by the following factors:

Functionality: refers to how the network supports data routing, interrupt handling, synchronization, request-"message combining, and coherence.

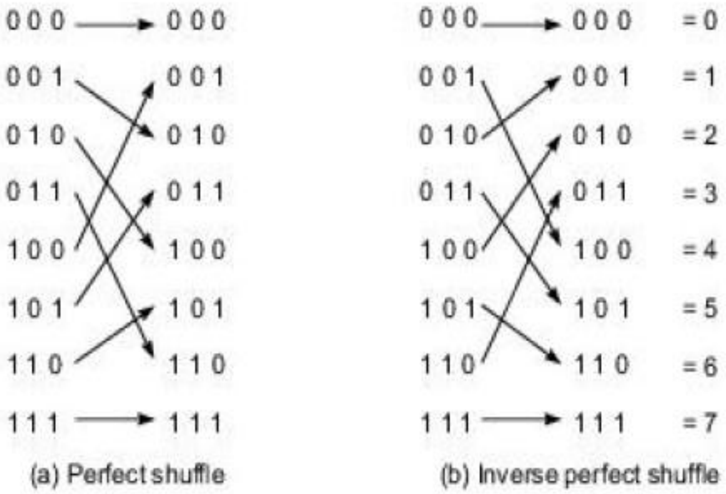
Network Latency:- This refers to the worst-ease time delay for a unit message to be transferred through the network.

Bandwidth: This refers to the maximum data transfer rate, in terms of Mbps or Gbps transmitted through the network.

Hardware Complexity'—This refers to implementation costs such as those for wires, switches, connectors, arbitration, and interface logic.

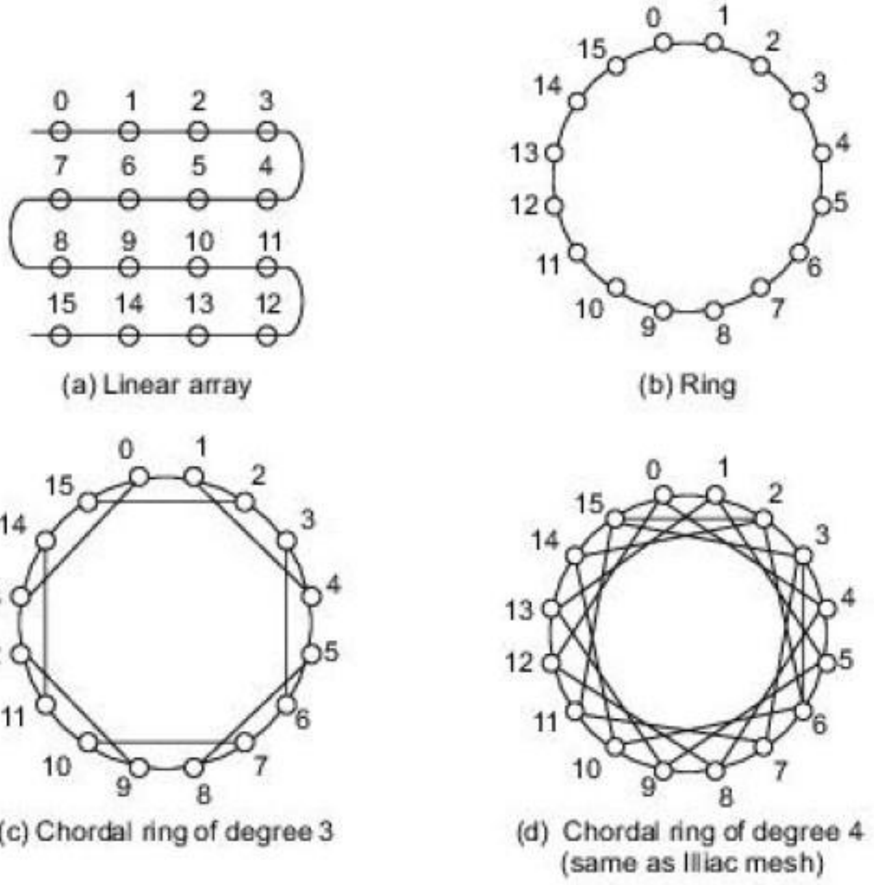
Scalability—This refers to the ability ofa network to be modularly expandable with a scalable performance with increasing machine resources.

Perfect Shuffle and Exchange Perfect shuffle is a special permutation function suggested by Harold Stone (1971) for parallel processing applications. The mapping corresponding to a perfect shuffle is shown in Fig. 2.14a. Its inverse is shown on the right-hand side (Fig. 2.14b).



2.4.2 Static Connection Networks:

Static networks use direct links which are fixed once built. This type of network is more suitable for building computers where the communication patterns are predictable or implementable with static connections. We describe their topologies below in terms of network parameters and comment on their relative merits in relation to communication and scalability.



(a) Linear array

(b) Ring

(c) Chordal ring of degree 3

(d) Chordal ring of degree 4 (same as Illiac mesh)

Dynamic Connection Networks

- Dynamic connection networks can implement all communication patterns based on program demands.
- In increasing order of cost and performance, these include
  - bus systems
  - multistage interconnection networks
  - crossbar switch networks
- Price can be attributed to the cost of wires, switches, arbiters, and connectors.
- Performance is indicated by network bandwidth, data transfer rate, network latency, and communication patterns supported.

Digital Buses

- A bus system (contention bus, time-sharing bus) has
  - a collection of wires and connectors
  - multiple modules (processors, memories, peripherals, etc.) which connect to the wires
  - data transactions between pairs of modules

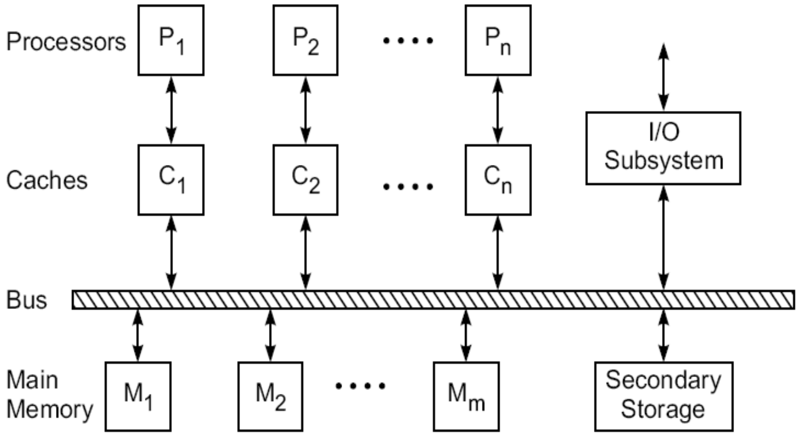
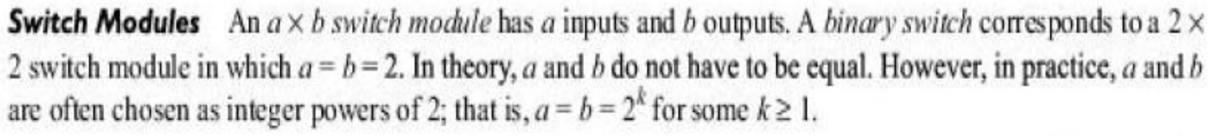
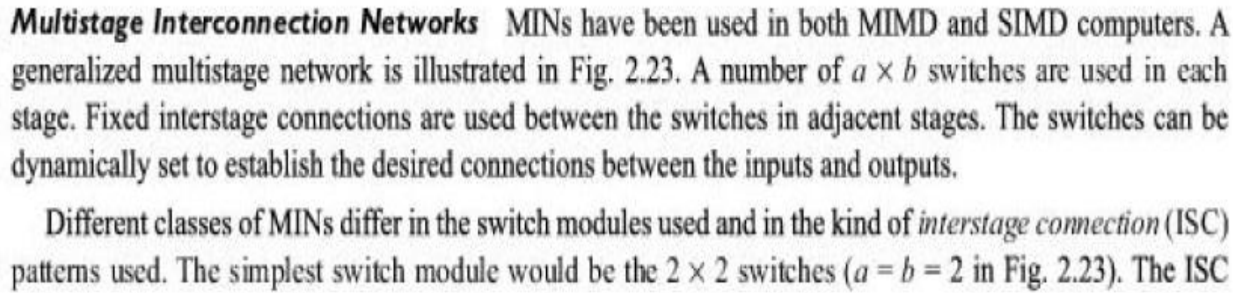


Fig. 2.22 A bus-connected multiprocessor system, such as the Sequent Symmetry S1

- Bus supports only one transaction at a time.
- Bus arbitration logic must deal with conflicting requests.
- Lowest cost and bandwidth of all dynamic schemes.
- Many bus standards are available.

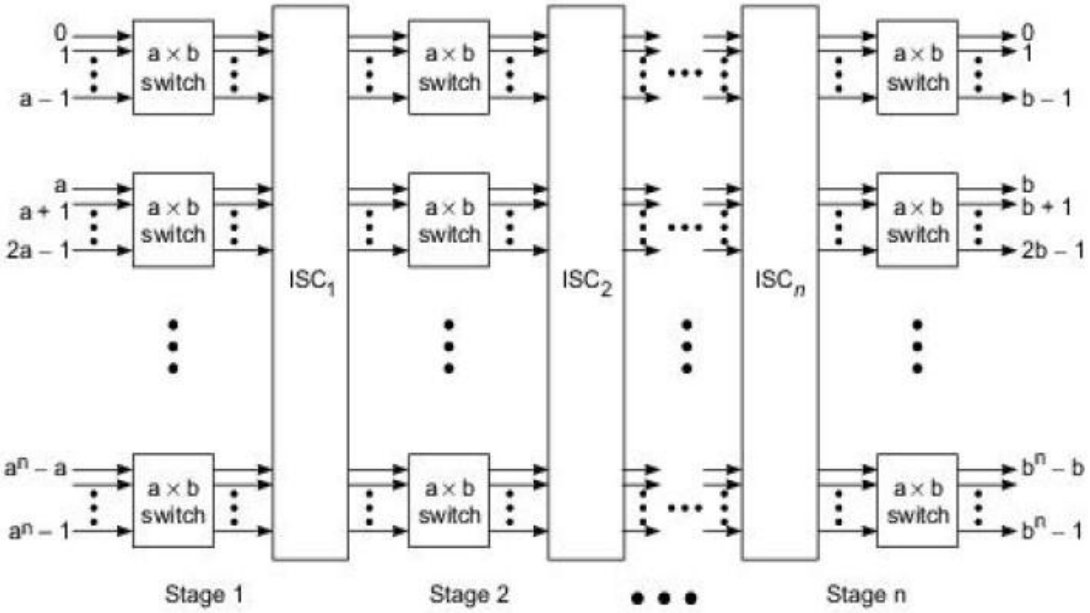


Switch Modules An $a \times b$ *switch module* has a inputs and b outputs. A *binary switch* corresponds to a 2 × 2 switch module in which $a = b = 2$. In theory, a and b do not have to be equal. However, in practice, a and b are often chosen as integer powers of 2; that is, $a = b = 2^k$ for some $k \geq 1$.



Multistage Interconnection Networks MINs have been used in both MIMD and SIMD computers. A generalized multistage network is illustrated in Fig. 2.23. A number of $a \times b$ switches are used in each stage. Fixed interstage connections are used between the switches in adjacent stages. The switches can be dynamically set to establish the desired connections between the inputs and outputs.

Different classes of MINs differ in the switch modules used and in the kind of *interstage connection* (ISC) patterns used. The simplest switch module would be the 2 × 2 switches ($a = b = 2$ in Fig. 2.23). The ISC


0
1
a – 1
a
a + 1
2a – 1
a^n – a
a^n – 1
a × b switch
ISC1
ISC2
ISCn
0
1
b – 1
b
b + 1
2b – 1
b^n – b
b^n – 1
Stage 1
Stage 2
Stage n

MODULE-2

Chapter 4 Processors and Memory Hierarchy

4.1 Advanced Processor Technology

4.1.1 Design Space of Processors

- Processors can be "mapped" to a space that has clock rate and cycles per instruction (CPI) as coordinates. Each processor type occupies a region of this space.
- Newer technologies are enabling higher clock rates.
- Manufacturers are also trying to lower the number of cycles per instruction.
- Thus the "future processor space" is moving toward the lower right of the processor design space.

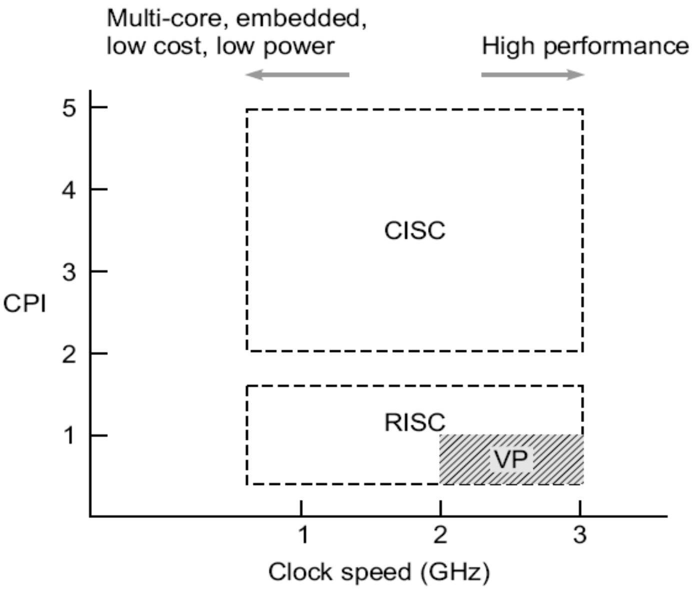


Fig. 4.1 CPI versus processor clock speed of major categories of processors

CISC and RISC Processors

- Complex Instruction Set Computing (CISC) processors like the Intel 80486, the Motorola 68040, the VAX/8600, and the IBM S/390 typically use microprogrammed control units, have lower clock rates, and higher CPI figures than…
- Reduced Instruction Set Computing (RISC) processors like the Intel i860, SPARC, MIPS R3000, and IBM RS/6000, which have hard-wired control units, higher clock rates, and lower CPI figures.

Superscalar Processors

- This subclass of the RISC processors allow multiple instructions to be issued simultaneously during each cycle.

- The effective CPI of a superscalar processor should be less than that of a generic scalar RISC processor.
- Clock rates of scalar RISC and superscalar RISC machines are similar.

VLIW Machines

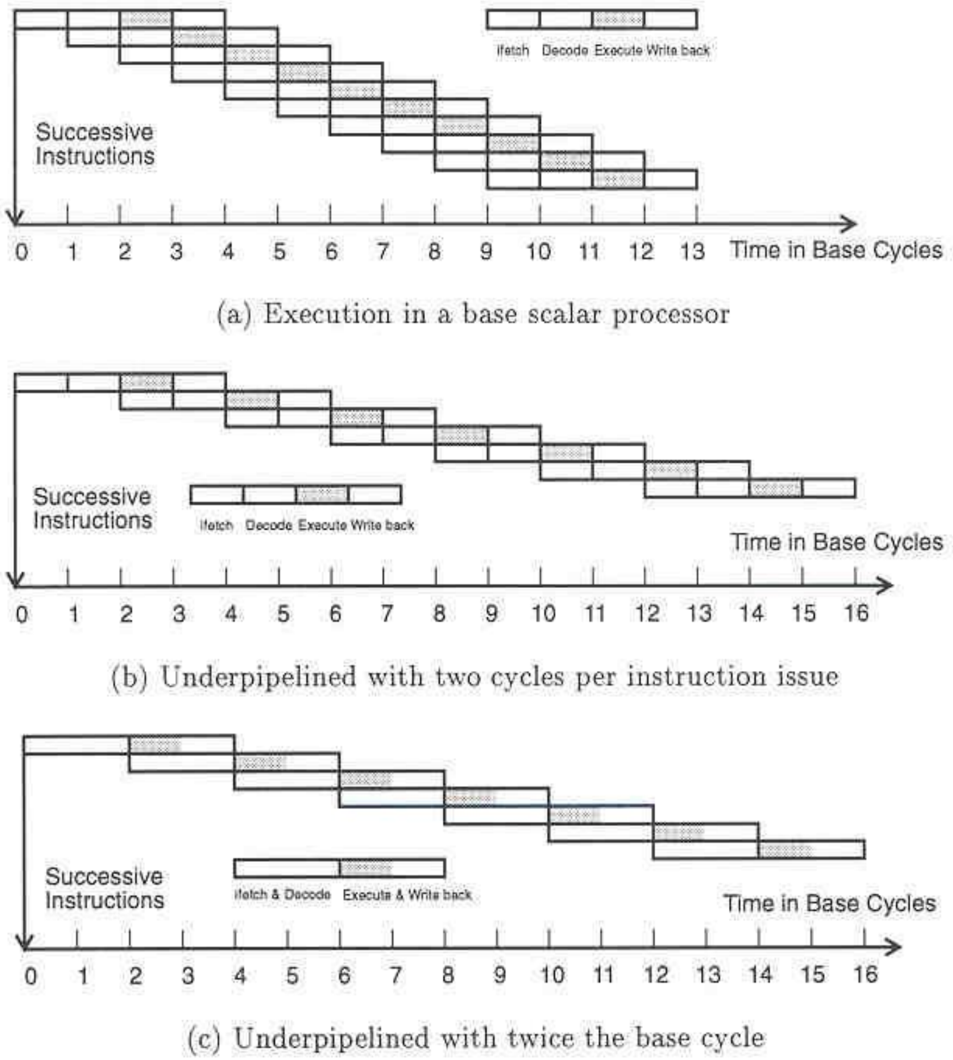
- Very Long Instruction Word machines typically have many more functional units than superscalars (and thus the need for longer – 256 to 1024 bits – instructions to provide control for them).
- These machines mostly use microprogrammed control units with relatively slow clock rates because of the need to use ROM to hold the microcode.

Superpipelined Processors

- These processors typically use a multiphase clock (actually several clocks that are out of phase with each other, each phase perhaps controlling the issue of another instruction) running at a relatively high rate.
- The CPI in these machines tends to be relatively high (unless multiple instruction issue is used).
- Processors in vector supercomputers are mostly superpipelined and use multiple functional units for concurrent scalar and vector operations.

Instruction Pipelines

- Typical instruction includes four phases:
  - fetch
  - decode
  - execute
  - write-back
- These four phases are frequently performed in a pipeline, or “assembly line” manner, as illustrated on the figure 4.2.
- The pipeline, like an industrial assembly line, receives successive instructions from its input end and executes them in a streamlined, overlapped fashion as they flow through.
- A pipeline cycle is intuitively defined as the time required for each phase to complete its operation, assuming equal delay in all phases (pipeline stages).



(a) Execution in a base scalar processor

(b) Underpipelined with two cycles per instruction issue

(c) Underpipelined with twice the base cycle

Figure 4.2 Pipelined execution of successive instructions in a base scalar processor and in two underpipelined cases. Courtesy of Jouppi and Wall; reprinted from *Proc. ASPLOS*, ACM Press, 1989)

Basic definitions associated with Pipeline operations:

- **Instruction pipeline cycle** – the time required for each phase to complete its operation (assuming equal delay in all phases)

- **Instruction issue latency** – the time (in cycles) required between the issuing of two adjacent instructions

- **Instruction issue rate** – the number of instructions issued per cycle (the *degree* of a superscalar)

- **Simple operation latency** – the delay (after the previous instruction) associated with the completion of a simple operation (e.g. integer add) as compared with that of a complex operation (e.g. divide).

- **Resource conflicts** – when two or more instructions demand use of the same functional unit(s) at the same time.

Pipelined Processors

- **A *base scalar processor***:
  - issues one instruction per cycle
  - has a one-cycle latency for a simple operation
  - has a one-cycle latency between instruction issues
  - can be fully utilized if instructions can enter the pipeline at a rate on one per cycle
- For a variety of reasons, instructions might not be able to be pipelines as aggressively as in a base scalar processor. In these cases, we say the pipeline is *underpipelined*.
- CPI rating is 1 for an ideal pipeline. Underpipelined systems will have higher CPI ratings, lower clock rates, or both.

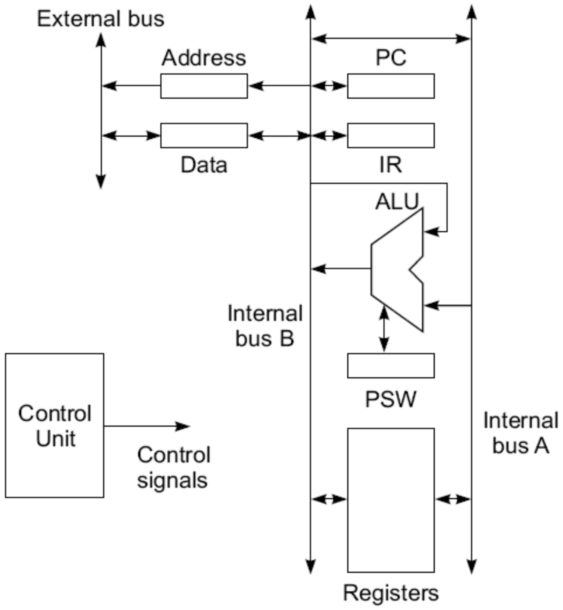


Fig. 4.3 Data path architecture and control unit of a scalar processor

- Figure 4.3 shows the data path architecture and control unit of a typical, simple scalar processor which does not employ an instruction pipeline. Main memory, I/O controllers, etc. are connected to the external bus.
- The control unit generates control signals required for the *fetch*, *decode*, *ALU operation*, *memory access*, and *write result* phases of instruction execution.

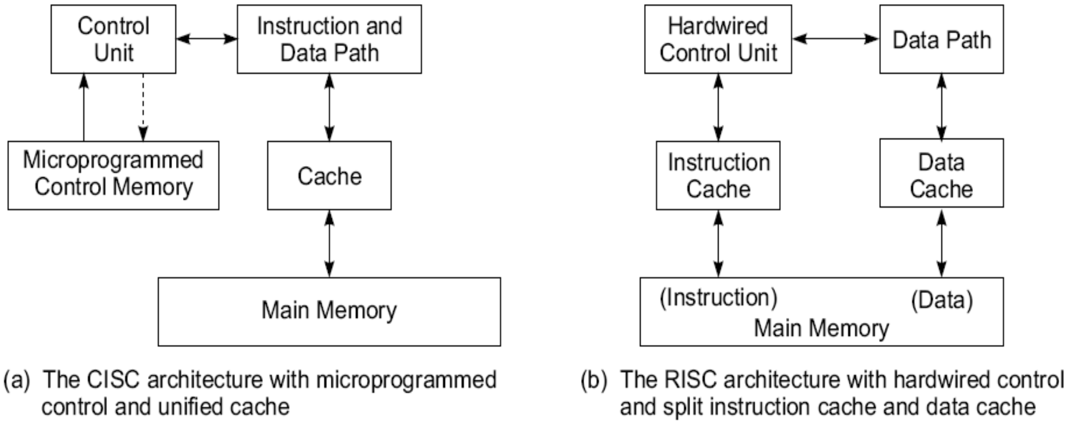
- The control unit itself may employ hardwired logic, or—as was more common in older CISC style processors—microcoded logic.
- Modern RISC processors employ hardwired logic, and even modern CISC processors make use of many of the techniques originally developed for high-performance RISC processors.

4.1.2 Instruction Set Architectures

- **CISC**
  - Many different instructions
  - Many different operand data types
  - Many different operand addressing formats
  - Relatively small number of general purpose registers
  - Many instructions directly match high-level language constructions
- **RISC**
  - Many fewer instructions than CISC (freeing chip space for more functional units!)
  - Fixed instruction format (e.g. 32 bits) and simple operand addressing
  - Relatively large number of registers
  - Small CPI (close to 1) and high clock rates

Architectural Distinctions

- **CISC**
  - Unified cache for instructions and data (in most cases)
  - Microprogrammed control units and ROM in earlier processors (hard-wired controls units now in some CISC systems)
- **RISC**
  - Separate instruction and data caches
  - Hard-wired control units

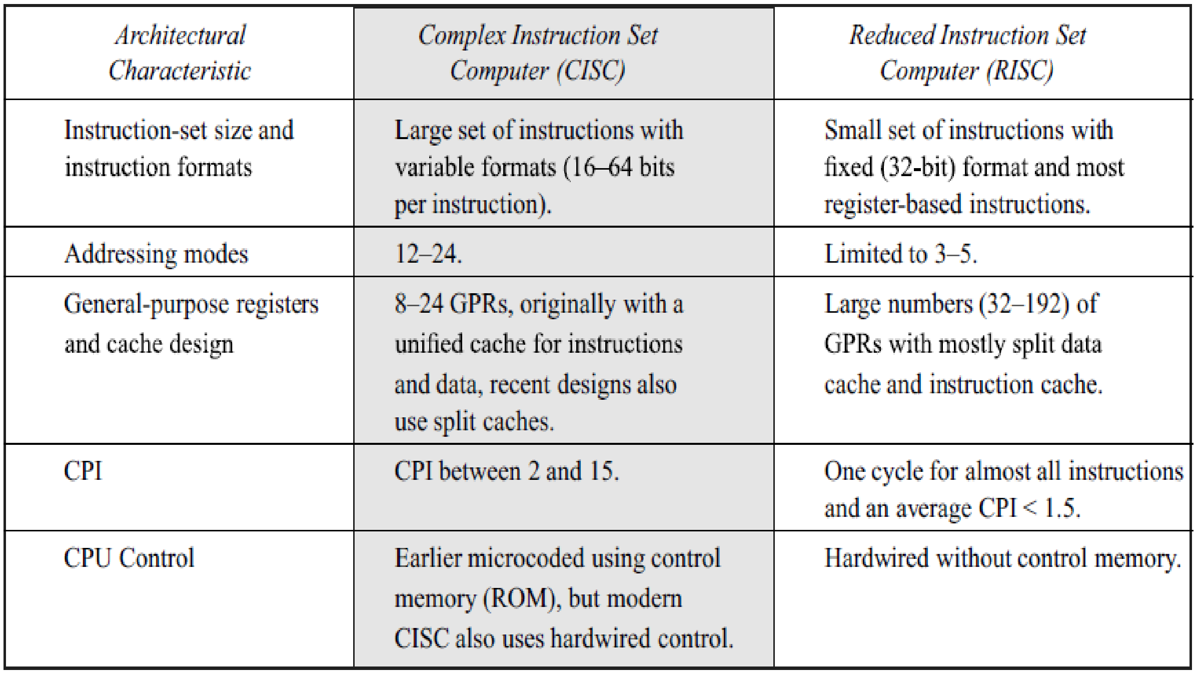


(a) The CISC architecture with microprogrammed control and unified cache

(b) The RISC architecture with hardwired control and split instruction cache and data cache

Fig. 4.4 Distinctions between typical RISC and typical CISC processor architectures (Courtesy of Gordon Bell, 1989)

Table 4.1 *Characteristics of Typical CISC and RISC Architectures*



Architectural Characteristic	*Complex Instruction Set Computer (CISC)*	*Reduced Instruction Set Computer (RISC)*
Instruction-set size and instruction formats	Large set of instructions with variable formats (16–64 bits per instruction).	Small set of instructions with fixed (32-bit) format and most register-based instructions.
Addressing modes	12–24.	Limited to 3–5.
General-purpose registers and cache design	8–24 GPRs, originally with a unified cache for instructions and data, recent designs also use split caches.	Large numbers (32–192) of GPRs with mostly split data cache and instruction cache.
CPI	CPI between 2 and 15.	One cycle for almost all instructions and an average CPI < 1.5.
CPU Control	Earlier microcoded using control memory (ROM), but modern CISC also uses hardwired control.	Hardwired without control memory.

- **CISC Advantages**
  - Smaller program size (fewer instructions)
  - Simpler control unit design
  - Simpler compiler design
- **RISC Advantages**
  - Has potential to be faster
  - Many more registers
- **RISC Problems**
  - More complicated register decoding system
  - Hardwired control is less flexible than microcode

4.1.3 CISC Scalar Processors

- Early systems had only integer fixed point facilities.
- Modern machines have both fixed and floating point facilities, sometimes as parallel functional units.
- Many CISC scalar machines are underpipelined.

Representative CISC Processors:

- VAX 8600
- Motorola MC68040
- Intel Pentium

VAX 8600 processor

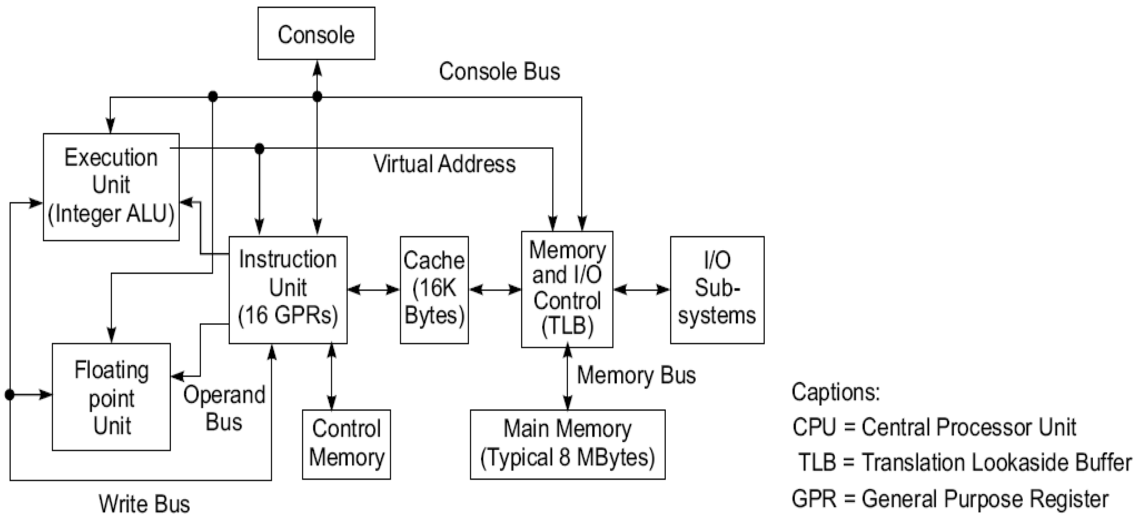

Console
Console Bus
Execution Unit (Integer ALU)
Virtual Address
Instruction Unit (16 GPRs)
Cache (16K Bytes)
Memory and I/O Control (TLB)
I/O Sub-systems
Floating point Unit
Operand Bus
Control Memory
Memory Bus
Main Memory (Typical 8 MBytes)
Write Bus
Captions:
CPU = Central Processor Unit
TLB = Translation Lookaside Buffer
GPR = General Purpose Register

Fig. 4.5 The VAX 8600 CPU, a typical CISC processor architecture (Courtesy of Digital Equipment Corporation, 1985)

- The VAX 8600 was introduced by Digital Equipment Corporation in 1985.
- This machine implemented a typical CISC architecture with microprogrammed control.
- The instruction set contained about 300 instructions with 20 different addressing modes.
- The CPU in the VAX 8600 consisted of two functional units for concurrent execution of integer and floating point instructions.
- The unified cache was used for holding both instructions and data.
- There were 16 GPRs in the instruction unit. Instruction pipelining was built with six stages in the VAX 8600, as in most elsc machines.
- The instruction unit prefetched and decoded instructions, handled branching operations, and supplied operands to the two functional units in a pipelined fashion.
- A Translation Lookaside Buffer (TLB) was used in the memory control unit for fast generation of a physical address from a virtual address.
- Both integer and floating point units were pipelined.
- The performance of the processor pipelines relied heavily on the cache hit ratio and on minimal branching damage to the pipeline flow.

4.1.4 RISC Scalar Processors

- Designed to issue one instruction per cycle
- RISC and CISC scalar processors should have same performance if clock rate and program lengths are equal.

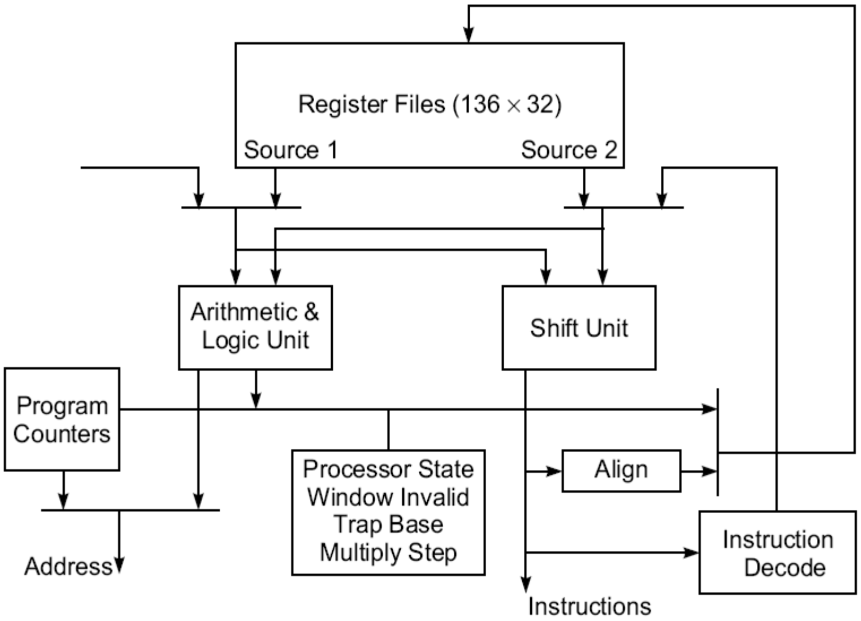
- RISC moves less frequent operations into software, thus dedicating hardware resources to the most frequently used operations.

Representative RISC Processors:

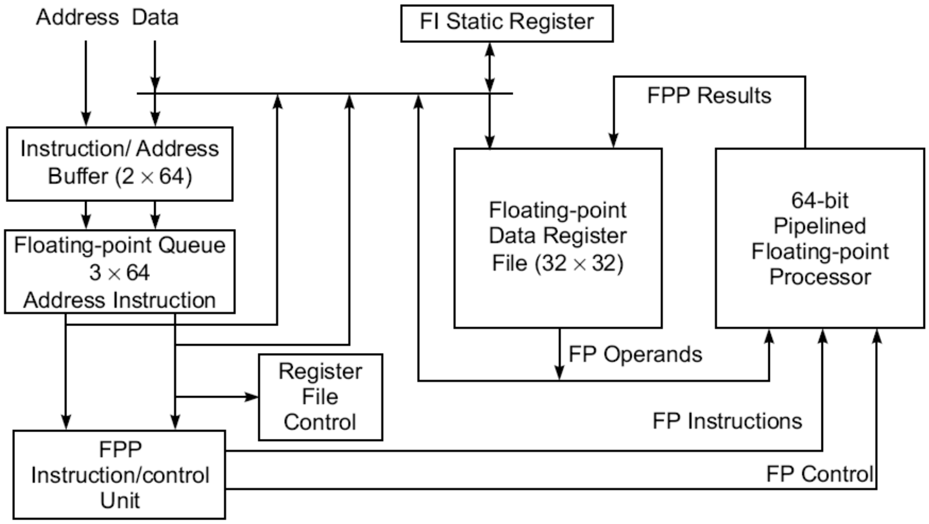
- Sun SPARC
- Intel i860
- Motorola M88100
- AMD 29000

SPARCs (Scalable Processor Architecture) and Register Windows

- SPARC family chips produced by Cypress Semiconductors, Inc. Figure 4.7 shows the architecture of the Cypress CY7C601 SPARC processor and of the CY7C602 FPU.
- The Sun SPARC instruction set contains 69 basic instructions
- The SPARC runs each procedure with a set of thirty-two 32-bit IU registers.
- Eight of these registers are ***global registers*** shared by all procedures, and the remaining 24 are ***window registers*** associated with only each procedure.
- The concept of using overlapped register windows is the most important feature introduced by the Berkeley RISC architecture.



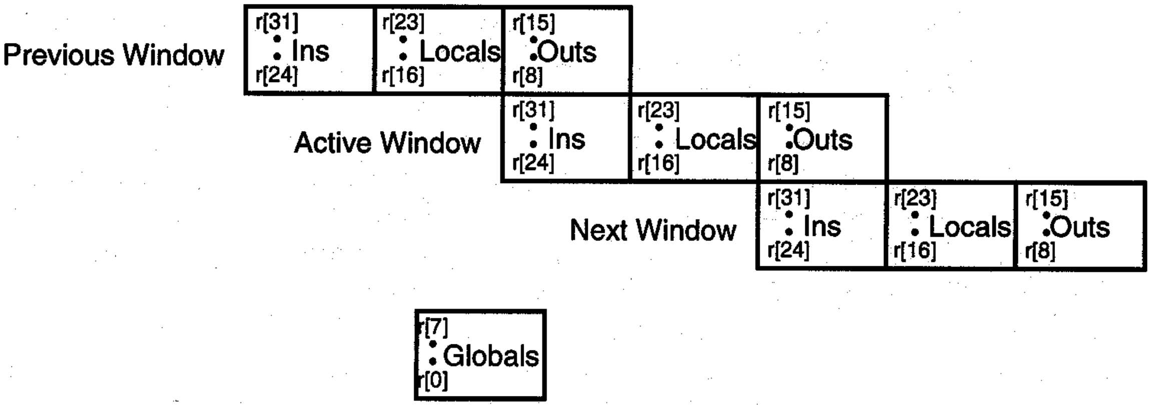
(a) The Cypress CY7C601 SPARC processor


Address
Data
FI Static Register
FPP Results
Instruction/ Address Buffer (2 × 64)
Floating-point Data Register File (32 × 32)
64-bit Pipelined Floating-point Processor
Floating-point Queue 3 × 64 Address Instruction
FP Operands
Register File Control
FP Instructions
FPP Instruction/control Unit
FP Control

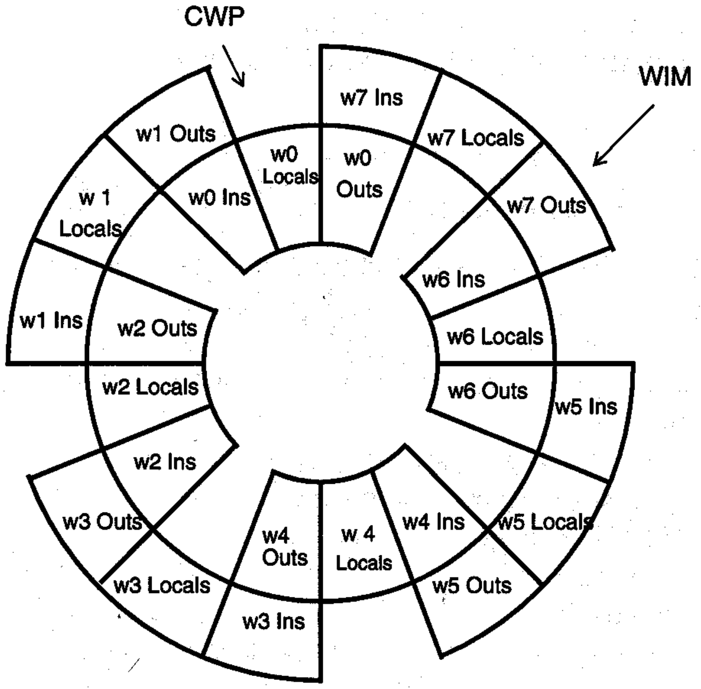
(b) The Cypress CY7C602 floating-point unit

Fig. 4.7 The SPARC architecture with the processor and the floating-point unit on two separate chips (Courtesy of Cypress Semiconductor Co., 1991)

- Fig. 4.8 shows eight overlapping windows (formed with 64 **local registers** and 64 overlapped registers) and eight **globals** with a total of 136 registers, as implemented in the Cypress 601.
- Each register window is divided into three eight-register sections, labeled ***Ins***, ***Locals***, and ***Outs***.
- The local registers are only locally addressable by each procedure. The Ins and Outs are shared among procedures.
- The calling procedure passes parameters to the called procedure via its Outs (r8 to r15) registers, which are the Ins registers of the called procedure.
- The window of the currently running procedure is called the active window pointed to by a current window pointer.
- A window invalid mask is used to indicate which window is invalid. The trap base register serves as a pointer to a trap handler.



(a) Three overlapping register windows and the global registers



(b) Eight register windows forming a circular stack

Figure 4.8 The concept of overlapping register windows in the SPARC architecture. (Courtesy of Sun Microsystems, Inc., 1987)

- A special register is used to create a 64-bit product in multiple step instructions. Procedures can also be called without changing the window.
- The overlapping windows can significantly save the time required for interprocedure communications, resulting in much faster context switching among cooperative procedures.

4.2 Superscalar, Vector Processors

- A CISC or a RISC scalar processor can be improved with a superscalar or vector architecture.
- Scalar processors are those executing one instruction per cycle.
- Only one instruction is issued per cycle, and only one completion of instruction is expected from the pipeline per cycle.
- In a superscalar processor, multiple instructions are issued per cycle and multiple results are generated per cycle.
- A vector processor executes vector instructions on arrays of data; each vector instruction involves a string of repeated operations, which are ideal for pipelining with one result per cycle.

4.2.1 Superscalar Processors

- Superscalar processors are designed to exploit more instruction-level parallelism in user programs.
- Only independent instructions can be executed in parallel without causing a wait state. The amount of instruction level parallelism varies widely depending on the type of code being executed.
- It has been observed that the average value is around 2 for code without loop unrolling. Therefore, for these codes there is not much benefit gained from building a machine that can issue more than three instructions per cycle.
- The instruction-issue degree in a superscalar processor has thus been limited to 2 to 5 in practice.

Pipelining in Superscalar Processors

- The fundamental structure of a three-issue superscalar pipeline is illustrated in Fig. 4.11.
- Superscalar processors were originally developed as an alternative to vector processors, with a view to exploit higher degree of instruction level parallelism.

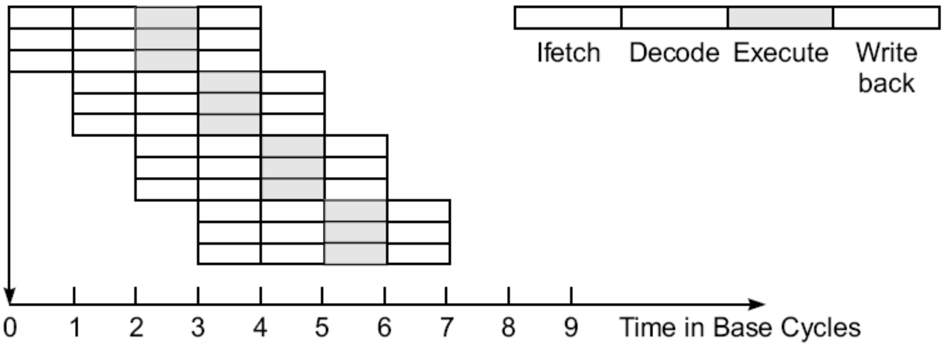


Fig. 4.11 A superscalar processor of degree m = 3

- A superscalar processor of degree m can issue m instructions per cycle.
- The base scalar processor, implemented either in RISC or CISC, has m = 1.
- In order to fully utilize a superscalar processor of degree m, m instructions must be executable in parallel. This situation may not be true in all clock cycles.
- In that case, some of the pipelines may be stalling in a wait state.
- In a superscalar processor, the simple operation latency should require only one cycle, as in the base scalar processor.
- Due to the desire for a higher degree of instruction-level parallelism in programs, the superscalar processor depends more on an optimizing compiler to exploit parallelism.

Representative Superscalar Processors

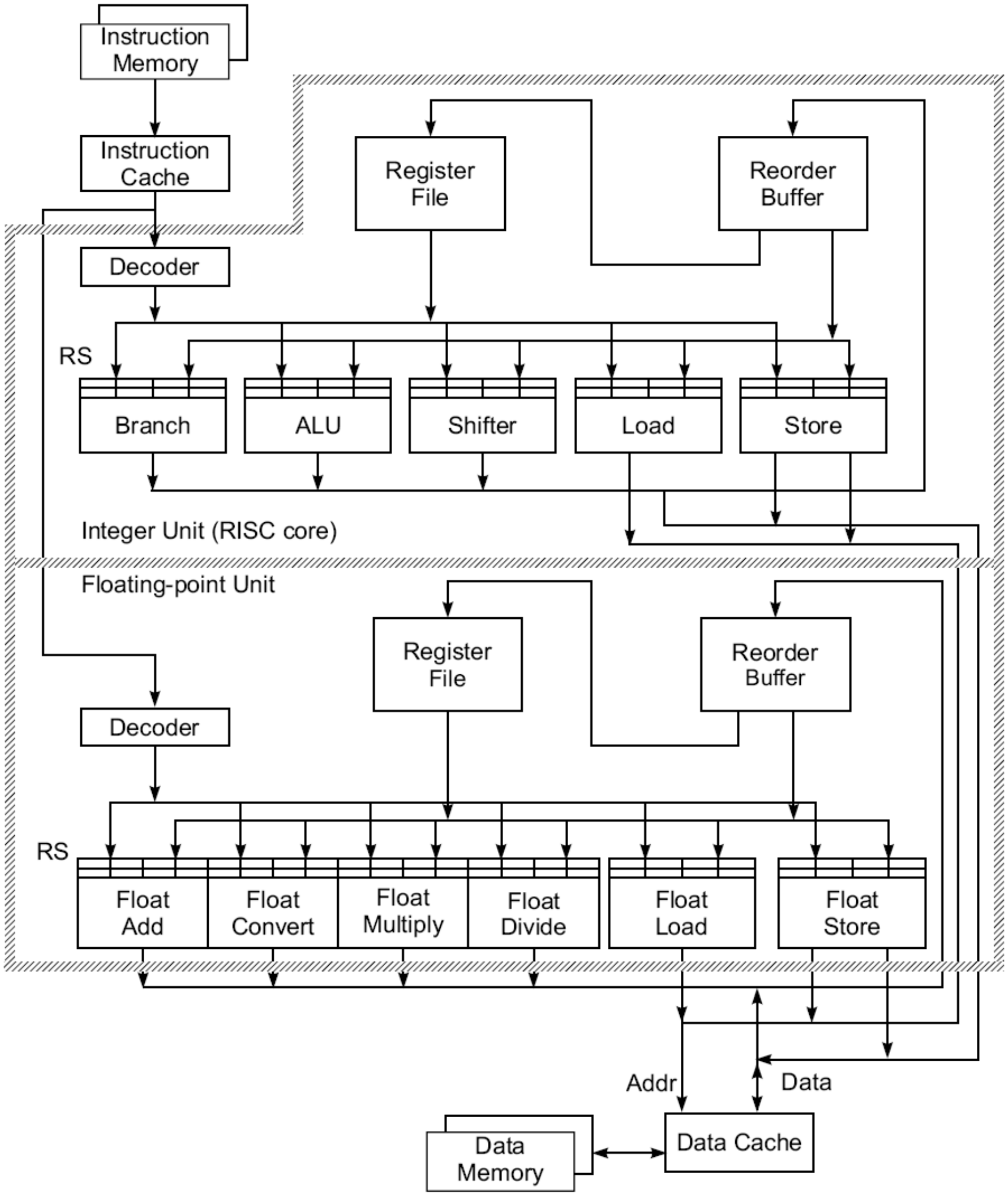


Fig. 4.12 A typical superscalar RISC processor architecture consisting of an integer unit and a floating-point unit (Courtesy of M. Johnson, 1991; reprinted with permission from Prentice-Hall, Inc.)

Typical Superscalar Architecture

- A typical superscalar will have
  - multiple instruction pipelines
  - an instruction cache that can provide multiple instructions per fetch
  - multiple buses among the function units
- In theory, all functional units can be simultaneously active.

4.2.2 VLIW Architecture

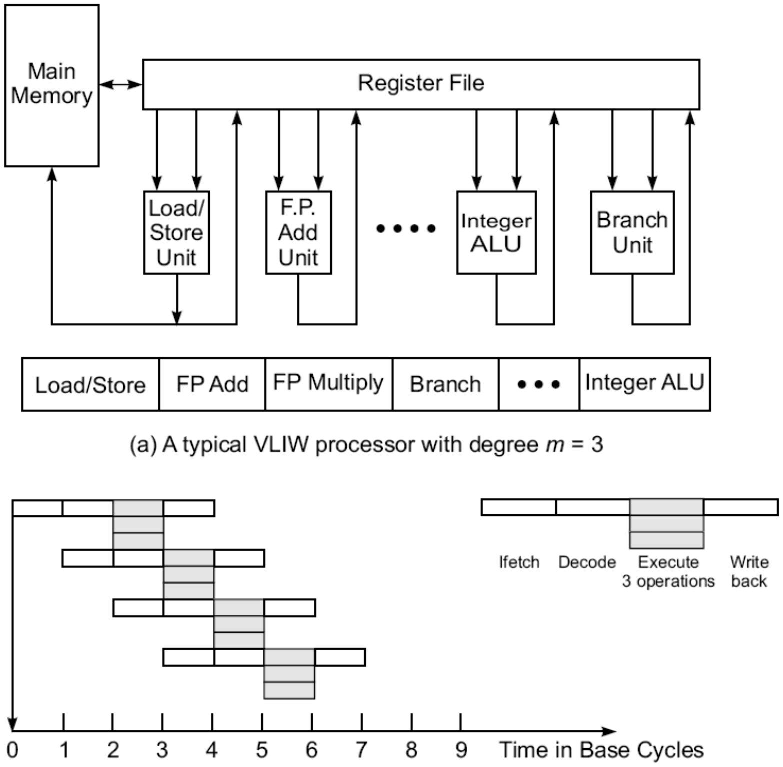
- VLIW = Very Long Instruction Word
- Instructions usually hundreds of bits long.
- Each instruction word essentially carries multiple “short instructions.”
- Each of the “short instructions” are effectively issued at the same time.
- (This is related to the long words frequently used in microcode.)
- Compilers for VLIW architectures should optimally try to predict branch outcomes to properly group instructions.

Pipelining in VLIW Processors

- Decoding of instructions is easier in VLIW than in superscalars, because each “region” of an instruction word is usually limited as to the type of instruction it can contain.

- Code density in VLIW is less than in superscalars, because if a “region” of a VLIW word isn’t needed in a particular instruction, it must still exist (to be filled with a “no op”).

- Superscalars can be compatible with scalar processors; this is difficult with VLIW parallel and non-parallel architectures.



(a) A typical VLIW processor with degree $m = 3$

(b) VLIW execution with degree $m = 3$

Fig. 4.14 The architecture of a very long instruction word (VLIW) processor and its pipeline operations (Courtesy of Multiflow Computer, Inc., 1987)

VLIW Opportunities

- "Random" parallelism among scalar operations is exploited in VLIW, instead of regular parallelism in a vector or SIMD machine.
- The efficiency of the machine is entirely dictated by the success, or "goodness," of the compiler in planning the operations to be placed in the same instruction words.
- Different implementations of the same VLIW architecture may not be binary-compatible with each other, resulting in different latencies.

VLIW Summary

- VLIW reduces the effort required to detect parallelism using hardware or software techniques.
- The main advantage of VLIW architecture is its simplicity in hardware structure and instruction set.
- Unfortunately, VLIW does require careful analysis of code in order to "compact" the most appropriate "short" instructions into a VLIW word.

4.2.3 Vector Processors

- A vector processor is a coprocessor designed to perform vector computations.
- A vector is a one-dimensional array of data items (each of the same data type).
- Vector processors are often used in multipipelined supercomputers.

Architectural types include:

1. **Register-to-Register** (with shorter instructions and register files)
2. **Memory-to-Memory** (longer instructions with memory addresses)

1. Register-to-Register Vector Instructions

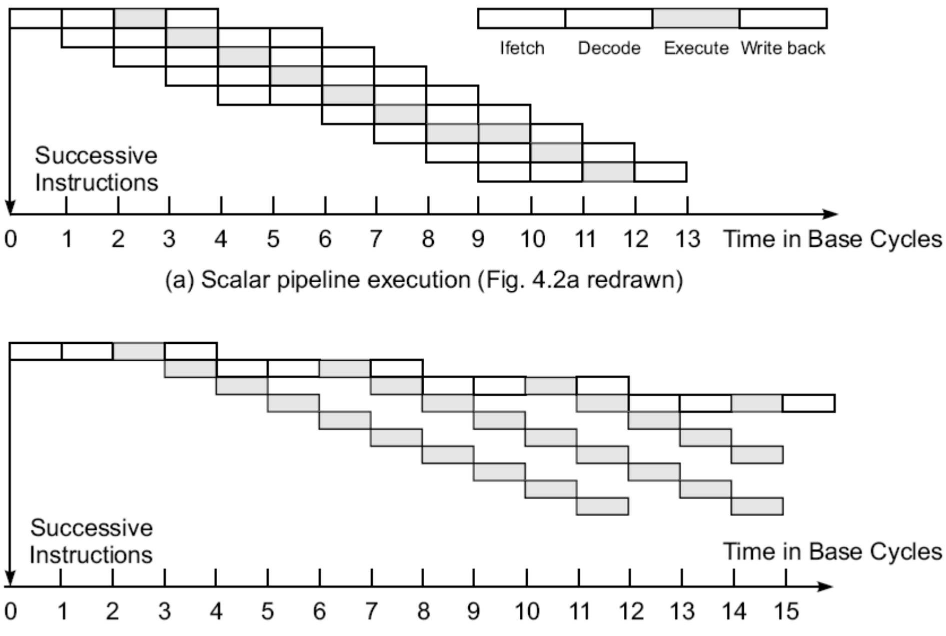
- Assume V_i is a vector register of length n, s_i is a scalar register, M(1:n) is a memory array of length n, and "o" is a vector operation.
- Typical instructions include the following:
  - $\mathbf{V_1 \text{ o } V_2 \rightarrow V_3}$ (element by element operation)
  - $\mathbf{s_1 \text{ o } V_1 \rightarrow V_2}$ (scaling of each element)
  - $\mathbf{V_1 \text{ o } V_2 \rightarrow s_1}$ (binary reduction - i.e. sum of products)
  - $\mathbf{M(1:n) \rightarrow V_1}$ (load a vector register from memory)
  - $\mathbf{V_1 \rightarrow M(1:n)}$ (store a vector register into memory)

  - **o $V_1 \rightarrow V_2$** (unary vector -- i.e. negation)
  - **o $V_1 \rightarrow s_1$** (unary reduction -- i.e. sum of vector)

2. Memory-to-Memory Vector Instructions

- Typical memory-to-memory vector instructions (using the same notation as given in the previous slide) include these:
  - **$M_1(1{:}n)$ o $M_2(1{:}n) \rightarrow M_3(1{:}\ n)$** (binary vector)
  - **s_1 o $M_1(1{:}n) \rightarrow M_2(1{:}n)$** (scaling)
  - **o $M_1(1{:}n) \rightarrow M_2(1{:}n)$** (unary vector)
  - **$M_1(1{:}n)$ o $M_2(1{:}n) \rightarrow M(k)$** (binary reduction)

Pipelines in Vector Processors



(a) Scalar pipeline execution (Fig. 4.2a redrawn)

(b) Vector pipeline execution

Fig. 4.15 Pipelined execution in a base scalar processor and in a vector processor, respectively (Courtesy of Jouppi and Wall; reprinted from Proc. ASPLOS, ACM Press, 1989)

- Vector processors can usually effectively use large pipelines in parallel, the number of such parallel pipelines effectively limited by the number of functional units.
- As usual, the effectiveness of a pipelined system depends on the availability and use of an effective compiler to generate code that makes good use of the pipeline facilities.

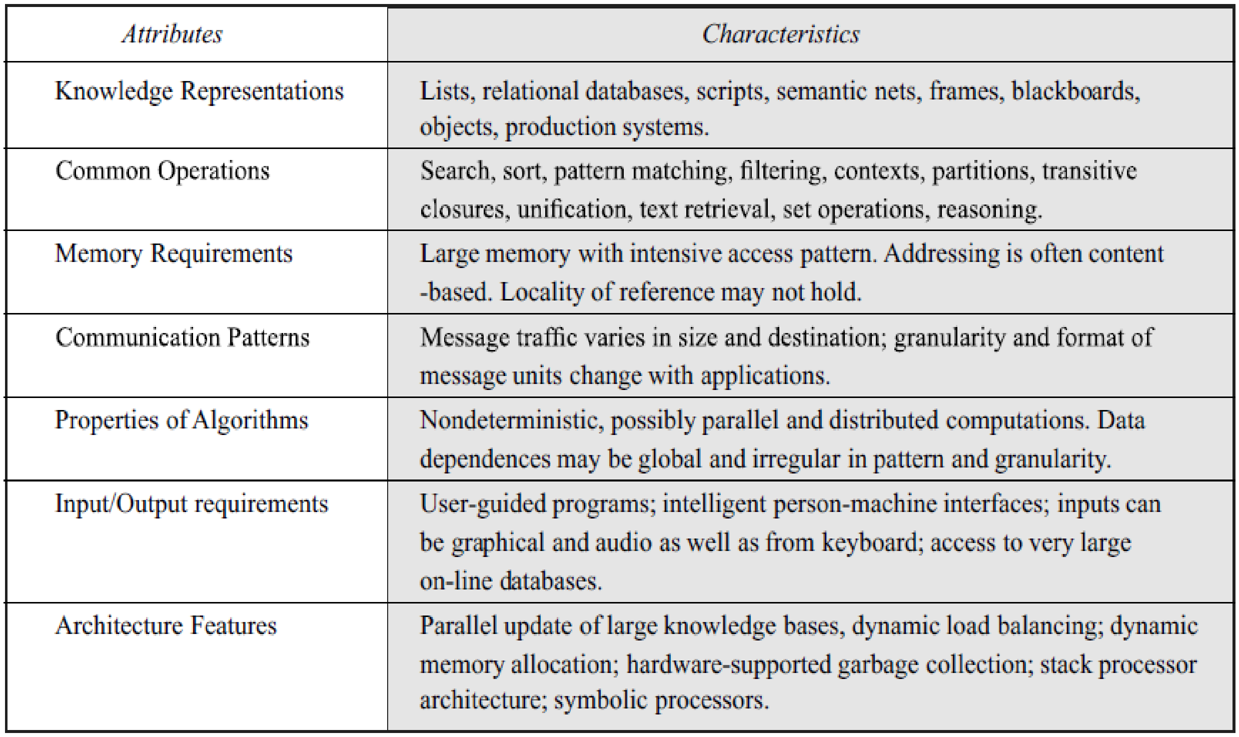
Symbolic Processors

- Symbolic processors are somewhat unique in that their architectures are tailored toward the execution of programs in languages similar to LISP, Scheme, and Prolog.

- In effect, the hardware provides a facility for the manipulation of the relevant data objects with “tailored” instructions.

- These processors (and programs of these types) may invalidate assumptions made about more traditional scientific and business computations.

Table 4.6 *Characteristics of Symbolic Processing*



Attributes	*Characteristics*
Knowledge Representations	Lists, relational databases, scripts, semantic nets, frames, blackboards, objects, production systems.
Common Operations	Search, sort, pattern matching, filtering, contexts, partitions, transitive closures, unification, text retrieval, set operations, reasoning.
Memory Requirements	Large memory with intensive access pattern. Addressing is often content -based. Locality of reference may not hold.
Communication Patterns	Message traffic varies in size and destination; granularity and format of message units change with applications.
Properties of Algorithms	Nondeterministic, possibly parallel and distributed computations. Data dependences may be global and irregular in pattern and granularity.
Input/Output requirements	User-guided programs; intelligent person-machine interfaces; inputs can be graphical and audio as well as from keyboard; access to very large on-line databases.
Architecture Features	Parallel update of large knowledge bases, dynamic load balancing; dynamic memory allocation; hardware-supported garbage collection; stack processor architecture; symbolic processors.

4.3 Memory Hierarchical Technology

- Storage devices such as registers, caches, main memory, disk devices, and backup storage are often organized as a hierarchy as depicted in Fig. 4.17.
- The memory technology and storage organization at each level is characterized by five parameters:

1. **access time t_i** (round-trip time from CPU to ith level)
2. **memory size s_i** (number of bytes or words in level i)
3. **cost per byte c_i**
4. **transfer bandwidth b_i** (rate of transfer between levels)
5. **unit of transfer x_i** (grain size for transfers between levels i and i+1)

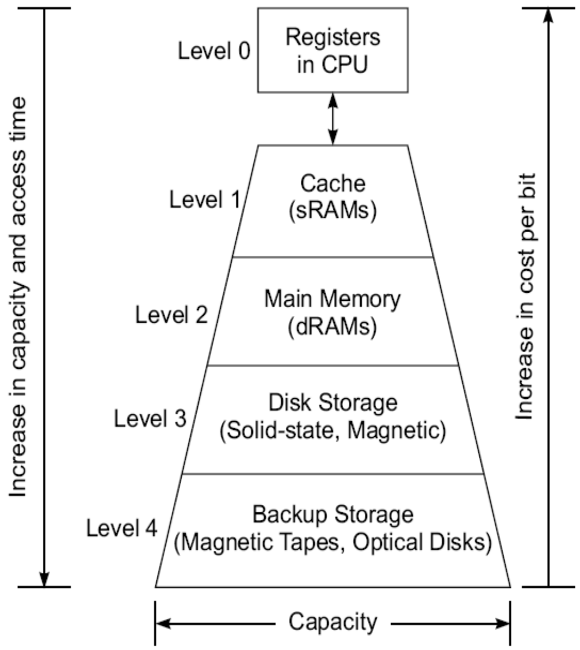

Level 0
Registers in CPU
Level 1
Cache (sRAMs)
Level 2
Main Memory (dRAMs)
Level 3
Disk Storage (Solid-state, Magnetic)
Level 4
Backup Storage (Magnetic Tapes, Optical Disks)
Capacity
Increase in capacity and access time
Increase in cost per bit

Fig. 4.17 A four-level memory hierarchy with increasing capacity and decreasing speed and cost from low to high levels

Memory devices at a lower level are:

- faster to access,
- are smaller in capacity,
- are more expensive per byte,
- have a higher bandwidth, and
- have a smaller unit of transfer.

In general, $\mathbf{t_{i-1} < t_i}$, $\mathbf{s_{i-1} < s_i}$, $\mathbf{c_{i-1} > c_i}$, $\mathbf{b_{i-1} > b_i}$ **and** $\mathbf{x_{i-1} < x_i}$ for i = 1, 2, 3, and 4 in the hierarchy where i = 0 corresponds to the CPU register level.

The cache is at level 1, main memory at level 2, the disks at level 3 and backup storage at level 4.

Registers and Caches

Registers

- The registers are parts of the processor;
- Register assignment is made by the compiler.
- Register transfer operations are directly controlled by the processor after instructions are decoded.
- Register transfer is conducted at processor speed, in one clock cycle.

Caches

- The cache is controlled by the MMU and is programmer-transparent.
- The cache can also be implemented at one or multiple levels, depending on the speed and application requirements.
- Multi-level caches are built either on the processor chip or on the processor board.
- Multi-level cache systems have become essential to deal with memory access latency.

Main Memory (Primary Memory)

- It is usually much larger than the cache and often implemented by the most cost-effective RAM chips, such as DDR SDRAMs, i.e. dual data rate synchronous dynamic RAMs.
- The main memory is managed by a MMU in cooperation with the operating system.

Disk Drives and Backup Storage

- The disk storage is considered the highest level of on-line memory.
- It holds the system programs such as the OS and compilers, and user programs and their data sets.
- Optical disks and magnetic tape units are off-line memory for use as archival and backup storage.
- They hold copies of present and past user programs and processed results and files.
- Disk drives are also available in the form of RAID arrays.

Peripheral Technology

- Peripheral devices include printers, plotters, terminals, monitors, graphics displays, optical scanners, image digitizers, output microfilm devices etc.
- Some I/O devices are tied to special-purpose or multimedia applications.

4.3.2 Inclusion, Coherence, and Locality

Information stored in a memory hierarchy (M1, M2,…, Mn) satisfies 3 important properties:

1. **Inclusion**
2. **Coherence**
3. **Locality**

- We consider cache memory the innermost level M1, which directly communicates with the CPU registers.
- The outermost level M_n contains all the information words stored. In fact, the collection of all addressable words in M_n forms the virtual address space of a computer.

- Program and data locality is characterized below as the foundation for using a memory hierarchy effectively.

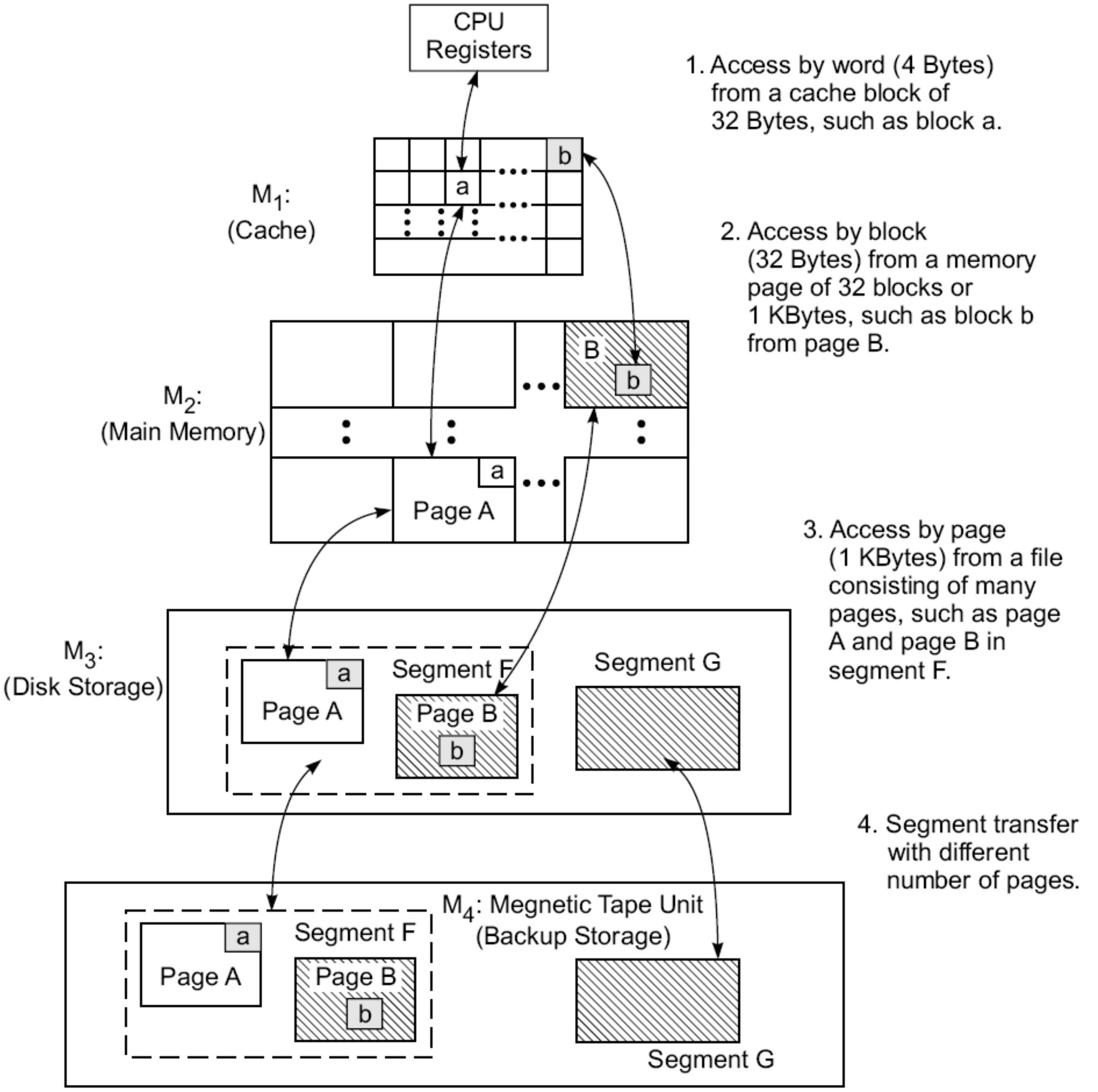

CPU Registers
1. Access by word (4 Bytes) from a cache block of 32 Bytes, such as block a.
M1: (Cache)
a
b
2. Access by block (32 Bytes) from a memory page of 32 blocks or 1 KBytes, such as block b from page B.
M2: (Main Memory)
B
b
a
Page A
3. Access by page (1 KBytes) from a file consisting of many pages, such as page A and page B in segment F.
M3: (Disk Storage)
Page A
a
Segment F
Page B
b
Segment G
4. Segment transfer with different number of pages.
M4: Megnetic Tape Unit (Backup Storage)
Page A
a
Segment F
Page B
b
Segment G

Fig. 4.18 The inclusion property and data transfers between adjacent levels of a memory hierarchy

1. The Inclusion Property

- The inclusion property is stated as:

  $\mathbf{M_1 \subset M_2 \subset ... \subset M_n}$

- The implication of the inclusion property is that all items of information in the “innermost” memory level (cache) also appear in the outer memory levels.

- The inverse, however, is not necessarily true. That is, the presence of a data item in level M_{i+1} does not imply its presence in level M_i. We call a reference to a missing item a "miss."

2. The Coherence Property

The requirement that copies of data items at successive memory levels be **consistent** is called the "coherence property."

Coherence Strategies

- **Write-through**
  - As soon as a data item in M_i is modified, immediate update of the corresponding data item(s) in M_{i+1}, M_{i+2}, … M_n is required.
  - This is the most aggressive (and expensive) strategy.
- **Write-back**
  - The update of the data item in M_{i+1} corresponding to a modified item in M_i is not updated unit it (or the block/page/etc. in M_i that contains it) is replaced or removed.
  - This is the most efficient approach, but cannot be used (without modification) when multiple processors share M_{i+1}, …, M_n.

3. Locality of References

- Memory references are generated by the CPU for either instruction or data access.
- These accesses tend to be clustered in certain regions in time, space, and ordering.

  There are three dimensions of the locality property:

  - ***Temporal locality*** – if location M is referenced at time t, then it (location M) will be referenced again at some time **t+Δt**.
  - ***Spatial locality*** – if location M is referenced at time t, then another location M±Δm will be referenced at time **t+Δt**.
  - ***Sequential locality*** – if location **M** is referenced at time **t**, then locations **M+1, M+2, …** will be referenced at time **t+Δt**, **t+Δt'**, etc.

- In each of these patterns, both Δm and Δt are "small."
- H&P suggest that 90 percent of the execution time in most programs is spent executing only 10 percent of the code.

Working Sets

- The set of addresses (bytes, pages, etc.) referenced by a program during the interval from t to t+ Δt, where Δt is called the *working set parameter*, changes slowly.
- This set of addresses, called the *working set*, should be present in the higher levels of M if a program is to execute efficiently (that is, without requiring numerous movements of data items from lower levels of M). This is called the *working set principle*.

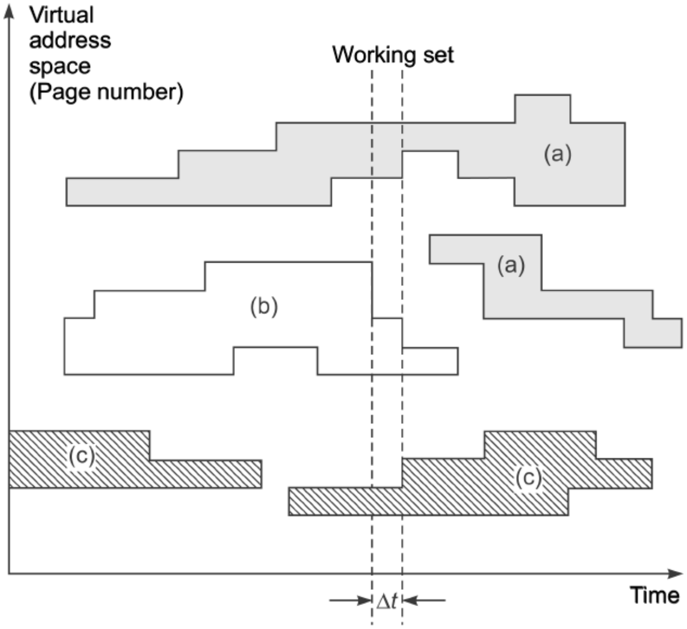


Fig. 4.19 Memory reference patterns in typical program trace experiments, where regions (a), (b), and (c) are generated with the execution of three software processes

4.3.3 Memory Capacity Planning

The performance of a memory hierarchy is determined by the Effective Access Time $\mathbf{T_{eff}}$ to any level in the hierarchy. It depends on the hit ratios and access frequencies.

Hit Ratios

- When a needed item (instruction or data) is found in the level of the memory hierarchy being examined, it is called a *hit*. Otherwise (when it is not found), it is called a *miss* (and the item must be obtained from a lower level in the hierarchy).
- The *hit ratio*, *h*, for M_i is the probability (between 0 and 1) that a needed data item is found when sought in level memory M_i.
- The *miss ratio* is obviously just $1-h_i$.

- We assume $h_0 = 0$ and $h_n = 1$.

Access Frequencies

- The access frequency f_i to level M_i is

  $$\mathbf{f_i = (1-h_1) \times (1-h_2) \times \ldots \times h_i}$$

- Note that $f_1 = h_1$, and $\sum_{i=1}^{n} \mathbf{fi} = 1$

Effective Access Times

- There are different penalties associated with misses at different levels in the memory hierarcy.
  - A cache miss is typically 2 to 4 times as expensive as a cache hit (assuming success at the next level).
  - A page fault (miss) is 3 to 4 magnitudes as costly as a page hit.
- The effective access time of a memory hierarchy can be expressed as

$$\begin{aligned} T_{eff} &= \sum_{i=1}^{n} f_i \cdot t_i \\ &= h_1 t_1 + (1-h_1) h_2 t_2 + \cdots + (1-h_1)(1-h_2)\cdots(1-h_{n-1}) h_n t_n \end{aligned}$$

- The effective access time is still dependent on program behavior and memory design choices.

Hierarchy Optimization

The total cost of a memory hierarchy is estimated as follows:

$$C_{\text{total}} = \sum_{i=1}^{n} c_i \cdot s_i$$

This implies that the cost is distributed over n levels. Since cl > c2 > c3 > … cn, we have to choose s1 < s2 < s3 < … sn.

The optimal design of a memory hierarchy should result in a T_{eff} close to the t_1 of M_1 and a total cost close to the cost of M_n.

The optimization process can be formulated as a linear programming problem, given a ceiling C_0 on the total cost— that is, a problem to minimize

$$T_{eff} = \sum_{i=1}^{n} f_i \cdot t_i$$

subject to the following constraints:

$$s_i > 0,\ t_i > 0 \quad \text{for } i = 1, 2, \ldots, n$$

$$C_{\text{total}} = \sum_{i=1}^{n} c_i \cdot s_i < C_0$$

4.4 Virtual Memory Technology

- To facilitate the use of memory hierarchies, the memory addresses normally generated by modern processors executing application programs are not *physical addresses*, but are rather *virtual addresses* of data items and instructions.
- Physical addresses, of course, are used to reference the available locations in the real physical memory of a system.
- Virtual addresses must be mapped to physical addresses before they can be used.

Virtual to Physical Mapping

- The mapping from virtual to physical addresses can be formally defined as follows:

$$f_t v = \begin{cases} m, & \text{if } m \in M \text{ has been allocated to store} \\ & \text{the data identified by virtual address } v \\ \varnothing & \text{if data } v \text{ is missing in } M \end{cases}$$

- The mapping returns a physical address if a *memory hit* occurs. If there is a *memory miss*, the referenced item has not yet been brought into primary memory.

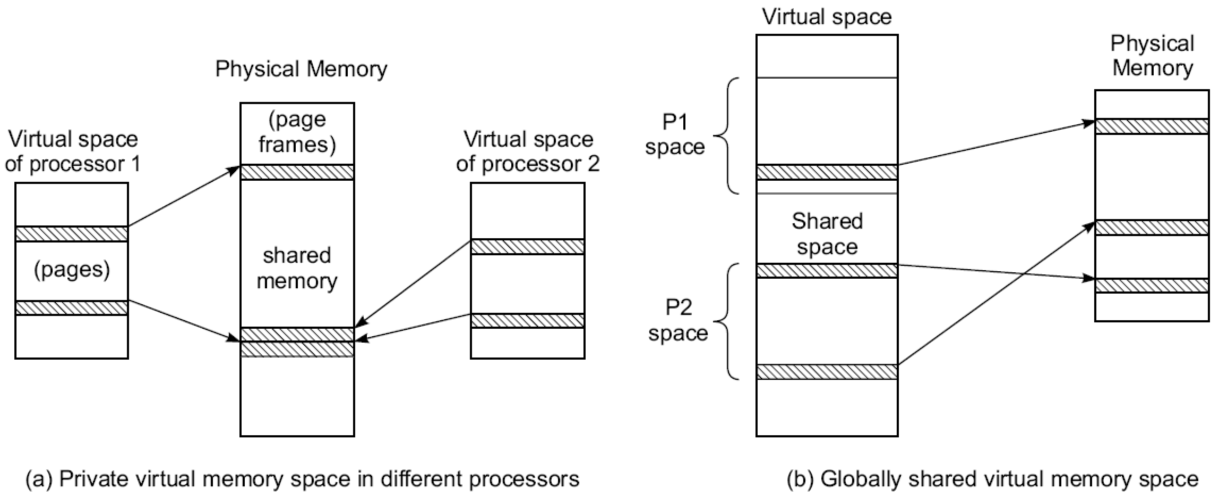
Mapping Efficiency

- The efficiency with which the virtual to physical mapping can be accomplished significantly affects the performance of the system.
- Efficient implementations are more difficult in multiprocessor systems where additional problems such as coherence, protection, and consistency must be addressed.

Virtual Memory Models

1. Private Virtual Memory

2. Shared Virtual Memory



(a) Private virtual memory space in different processors

(b) Globally shared virtual memory space

Fig. 4.20 Two virtual memory models for multiprocessor systems (Courtesy of Dubois and Briggs, tutorial, *Annual Symposium on Computer Architecture*, 1990)

1. Private Virtual Memory

- In this scheme, each processor has a separate virtual address space, but all processors share the same physical address space.
- **Advantages**:
  - Small processor address space
  - Protection on a per-page or per-process basis
  - Private memory maps, which require no locking
- **Disadvantages**
  - The synonym problem – different virtual addresses in different/same virtual spaces point to the same physical page
  - The same virtual address in different virtual spaces may point to different pages in physical memory

2. Shared Virtual Memory

- All processors share a single shared virtual address space, with each processor being given a portion of it.
- Some of the virtual addresses can be shared by multiple processors.

Advantages:

- All addresses are unique

- Synonyms are not allowed

Disadvantages

- Processors must be capable of generating large virtual addresses (usually > 32 bits)
- Since the page table is shared, mutual exclusion must be used to guarantee atomic updates
- Segmentation must be used to confine each process to its own address space
- The address translation process is slower than with private (per processor) virtual memory

Memory Allocation

Both the virtual address space and the physical address space are divided into fixed-length pieces.

– In the virtual address space these pieces are called ***pages***.

– In the physical address space they are called ***page frames***.

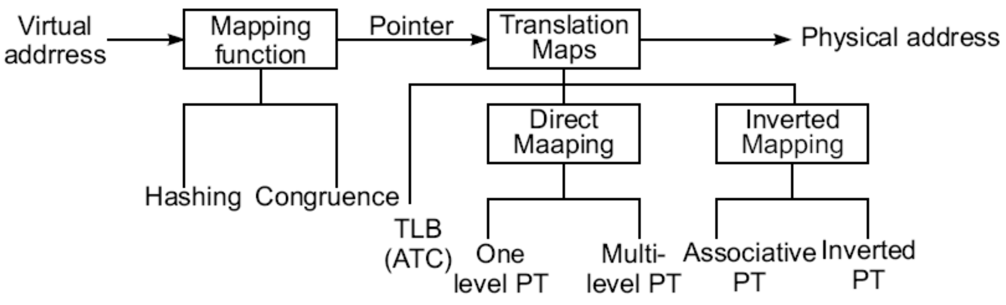
- The purpose of memory allocation is to allocate pages of virtual memory using the page frames of physical memory.

4.4.2 TLB, Paging, and Segmentation

Both the virtual memory and physical memory are partitioned into fixed-length pages. The purpose of memory allocation is to allocate pages of virtual memory to the page frames of the physical memory.

Address Translation Mechanisms

- The process demands the translation of virtual addresses into physical addresses. Various schemes for virtual address translation are summarized in Fig. 4.21a.



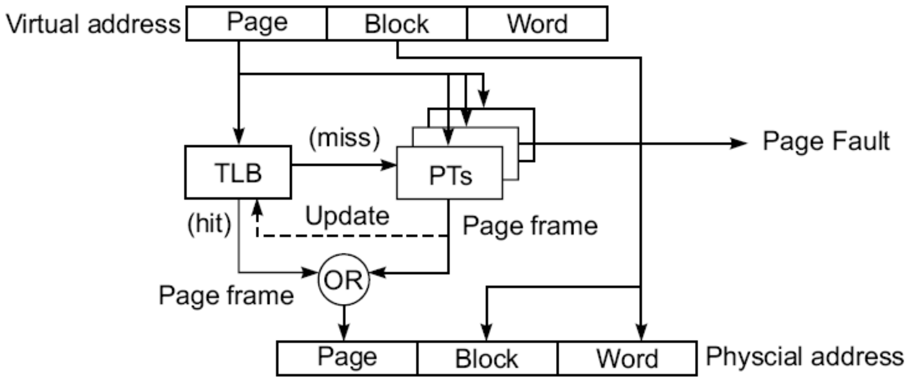
(a) Virtual addrress translation schemes (PT = page table)

- The translation demands the use of *translation maps* which can be implemented in various ways.
- Translation maps are stored in the cache, in associative memory, or in the main memory.

- To access these maps, a mapping function is applied to the virtual address. This function generates a pointer to the desired translation map.
- This mapping can be implemented with a *hashing* or *congruence* function.
- Hashing is a simple computer technique for converting a long page number into a short one with fewer bits.
- The hashing function should randomize the virtual page number and produce a unique hashed number to be used as the pointer.

Translation Lookaside Buffer

- The TLB is a high-speed lookup table which stores the most recently or likely referenced page entries.
- A *page entry* consists of essentially a (virtual page number, page frame number) pair. It is hoped that pages belonging to the same working set will be directly translated using the TLB entries.
- The use of a TLB and PTs for address translation is shown in Fig 4.21b. Each virtual address is divided into 3 fields:
  - The leftmost field holds the virtual page number,
  - the middle field identifies the cache block number,
  - the rightmost field is the word address within the block.



(b) Use of a TLB and PTs for address translation

- Our purpose is to produce the physical address consisting of the page frame number, the block number, and the word address.
- The first step of the translation is to use the virtual page number as a key to search through the TLB for a match.
- The TLB can be implemented with a special associative memory (content addressable memory) or use part of the cache memory.

- In case of a match (a hit) in the TLB, the page frame number is retrieved from the matched page entry. The cache block and word address are copied directly.
- In case the match cannot be found (a miss) in the TLB, a hashed pointer is used to identify one of the page tables where the desired page frame number can be retrieved.

Implementing Virtual Memory

There are 3 approaches to implement virtual memory:

1. Paging
2. Segmentation
3. A combination of the two called **Paged Segmentation**

1. Paging memory

- Memory is divided into fixed-size blocks called pages.
- Main memory contains some number of pages which is smaller than the number of pages in the virtual memory.
- **For example,** if the page size is 2K and the physical memory is 16M (8K pages) and the virtual memory is 4G (2 M pages) then there is a factor of 254 to 1 mapping.
- A page map table is used for implementing a mapping, with one entry per virtual page.

2. Segmented memory

- In a segmented memory management system the blocks to be replaced in main memory are potentially of unequal length and here the segments correspond to logical blocks of code or data.

For example, a subroutine or procedure.

- Segments, then, are ``atomic'' in the sense that either the whole segment should be in main memory, or none of the segment should be there.
- The segments may be placed anywhere in main memory, but the instructions or data in one segment should be contiguous,

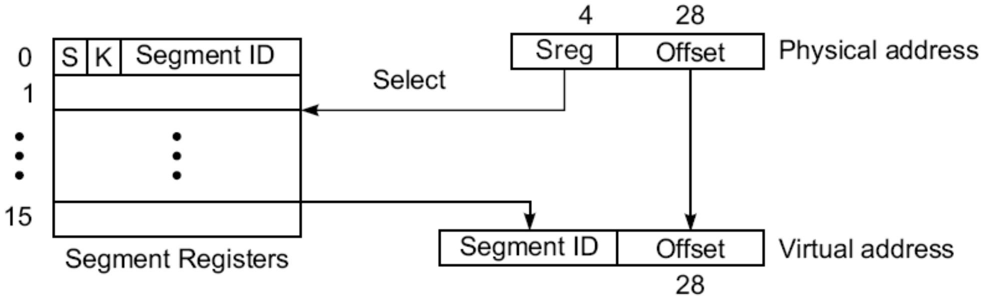
3. Paged Segmentation

- It is a combination of paging and segmentation concepts

- Within each segment, the addresses are divided into fixed size pages
- Each virtual address is divided into 3 fields
  - Segment Number
  - Page Number
  - Offset

Inverted paging

- Besides direct mapping, address translation maps can also be implemented with inverted mapping (Fig. 4.21c).
- An *inverted page table* is created for each page frame that has been allocated to users. Any virtual page number can be paired with a given physical page number.
- Inverted page tables are accessed either by an associative search or by the use of a hashing function.
- In using an inverted PT, only virtual pages that are currently resident in physical memory are included. This provides a significant reduction in the size of the page tables.
- The generation of a long virtual address from a short physical address is done with the help of segment registers, as demonstrated in Fig. 4.21c.



(c) Inverted address mapping

- The leading 4 bits (denoted *sreg*) of a 32-bit address name a segment register.
- The register provides a *segment id* that replaces the 4-bit sreg to form a long virtual address.
- This effectively creates a single long virtual address space with segment boundaries at multiples of 256 Mbytes (228 bytes).
- The IBM RT/PC had a 12-bit segment id (4096 segments) and a 40-bit virtual address space.
- Either associative page tables or inverted page tables can be used to implement inverted mapping.

- The inverted page table can also be assisted with the use of a TLB. An inverted PT avoids the use of a large page table or a sequence of page tables.
- Given a virtual address to be translated, the hardware searches the inverted PT for that address and, if it is found, uses the table index of the matching entry as the address of the desired page frame.
- A hashing table is used to search through the inverted PT.
- The size of an inverted PT is governed by the size of the physical space, while that of traditional PTs is determined by the size of the virtual space.
- Because of limited physical space, no multiple levels are needed for the inverted page table.

4.4.3 Page Replacement Policies

- Memory management policies include the allocation and deallocation of memory pages to active processes and the replacement of memory pages.
- Demand paging memory systems. refers to the process in which a resident page in main memory is replaced by a new page transferred from the disk.
- Since the number of available page frames is much smaller than the number of pages, the frames will eventually be fully occupied.
- In order to accommodate a new page, one of the resident pages must be replaced.
- The goal of a page replacement policy is to minimize the number of possible page faults so that the effective memory-access time can be reduced.
- The effectiveness of a replacement algorithm depends on the program behavior and memory traffic patterns encountered.
- A good policy should match the program locality property. The policy is also affected by page size and by the number of available frames.

Page Traces: A *page trace* is a sequence of *page frame numbers* (PFNs) generated during the execution of a given program.

The following page replacement policies are specified in a demand paging memory system for a page fault at time *t*.

(1) **Least recently used (LRU)**—This policy replaces the page in R(t) which has the longest backward distance:

$$q(t) = y, \quad \text{iff} \quad b_t(y) = \max_{x \in R(t)} \{b_t(x)\}$$

(2) **Optimal (OPT) algorithm**—This policy replaces the page in R(t) with the longest forward distance:

$$q(t) = y, \quad \text{iff} \quad f_t(y) = \max_{x \in R(t)} \{f_t(x)\}$$

(3) **First-in-first-out (FIFO)**—This policy replaces the page in R(t) which has been in memory for the longest time.

(4) **Least frequently used (LFU)**—This policy replaces the page in R(t) which has been least referenced in the past.

(5) **Circular FIFO**—This policy joins all the page frame entries into a circular FIFO queue using a pointer to indicate the front of the queue.

- An allocation bit is associated with each page frame. This bit is set upon initial allocation of a page to the frame.
- When a page fault occurs, the queue is circularly scanned from the pointer position.
- The pointer skips the allocated page frames and replaces the very first unallocated page frame.
- When all frames are allocated, the front of the queue is replaced, as in the FIFO policy.

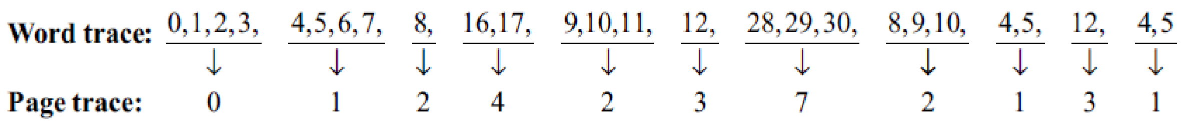
(6) **Random replacement**—This is a trivial algorithm which chooses any page for replacement randomly.

Example:

Consider a paged virtual memory system with a two-level hierarchy: main memory M_1 and disk memory M_2.

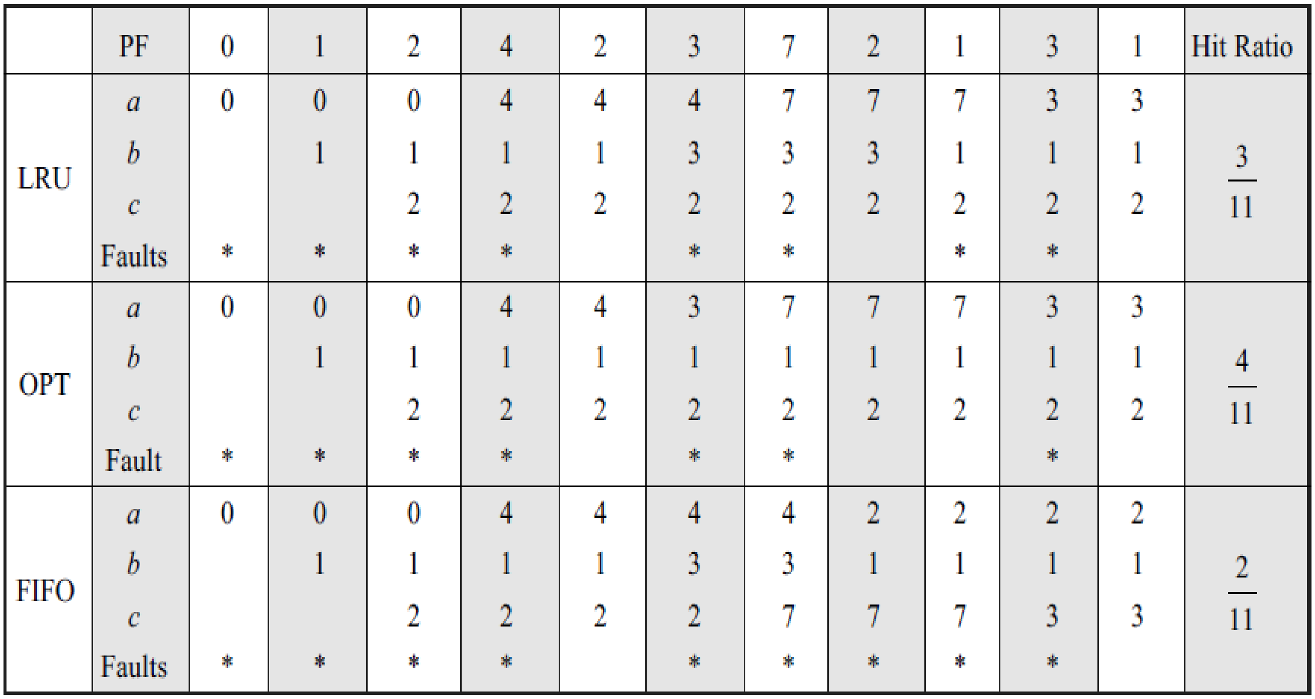
Assume a page size of four words. The number of page frames in M_1 is 3, labeled a, b and c; and the number of pages in M_2 is 10, identified by 0, 1, 2,….9. The ith page in M_2consists of word addresses 4i to 4i + 3 for all i = 0, 1, 2, …, 9.

A certain program generates the following sequence of word addresses which are grouped (underlined) together if they belong to the same page. The sequence of page numbers so formed is the *page trace*:



Word trace:	0,1,2,3,	4,5,6,7,	8,	16,17,	9,10,11,	12,	28,29,30,	8,9,10,	4,5,	12,	4,5
	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
Page trace:	0	1	2	4	2	3	7	2	1	3	1

Page tracing experiments are described below for three page replacement policies: LRU, OPT, and FIFO, respectively. The successive pages loaded in the page frames (PFs) form the trace entries. Initially, all PFs are empty.



	PF	0	1	2	4	2	3	7	2	1	3	1	Hit Ratio
LRU	*a*	0	0	0	4	4	4	7	7	7	3	3	$\frac{3}{11}$
	b		1	1	1	1	3	3	3	1	1	1	
	c			2	2	2	2	2	2	2	2	2	
	Faults	*	*	*	*		*	*		*	*		
OPT	*a*	0	0	0	4	4	3	7	7	7	3	3	$\frac{4}{11}$
	b		1	1	1	1	1	1	1	1	1	1	
	c			2	2	2	2	2	2	2	2	2	
	Fault	*	*	*	*		*	*			*		
FIFO	*a*	0	0	0	4	4	4	4	2	2	2	2	$\frac{2}{11}$
	b		1	1	1	1	3	3	1	1	1	1	
	c			2	2	2	2	7	7	7	3	3	
	Faults	*	*	*	*		*	*	*	*	*		