Prepared by, Sunil Kumar B.L. and Kishor Shivathaya, Canara Engineering College, Benjanapadavu.

ACA_NOTES

(15CS72)

MODULE-3

Chapter-5

Reference: Advanced Computer Architecture by Kai Hwang , Naresh Jotwani.

Chapter 5

BUS, CACHE, AND SHARED MEMORY

5.1 BUS SYSTEMS

Thc system bus of a computer system operates on a contention basis. Several active devices such as processors may request use of the bus at thc same timc. Only one of them can bc granted access at a timc. Thc effective bandwidth available to each processor is inversely proportional to thc numbcr of processors contending for thc bus.

5.1.1 Backplane Bus Specification

A backplanc bus interconnects processors, data storage, and peripheral devices in a tightly coupled hardware configuration.Timing protocols must be established to arbitrate among multiple requests.Signal lines on the backplane are often functionally grouped into several buses as shown in below Fig. 5.l. The Various functional boards are plugged into slots on thc backplane. Each slot is provided with one or more connectors for inserting the boards as demonstrated by thc vertical arrows in below Fig. 5.1.

Data Transfer Bus

Data, address, and control lines form the **data transfer bus (DTB)** in a VME bus. The **addressing lines** are used to broadcast the data and device address. Address modifier lines can be used to define special addressing modes. The data lines are often proportional to the memory word length.The **DTB control lines** are used to indicate read/write, timing control, and bus error-conditions.

Bus Arbitration and Control

The process of assigning control of the DTB to a requester is called **arbitration**. 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. **Utility lines** include signals that provide periodic timing (clocking) and coordinatc the power-up and power-down sequences of the system.

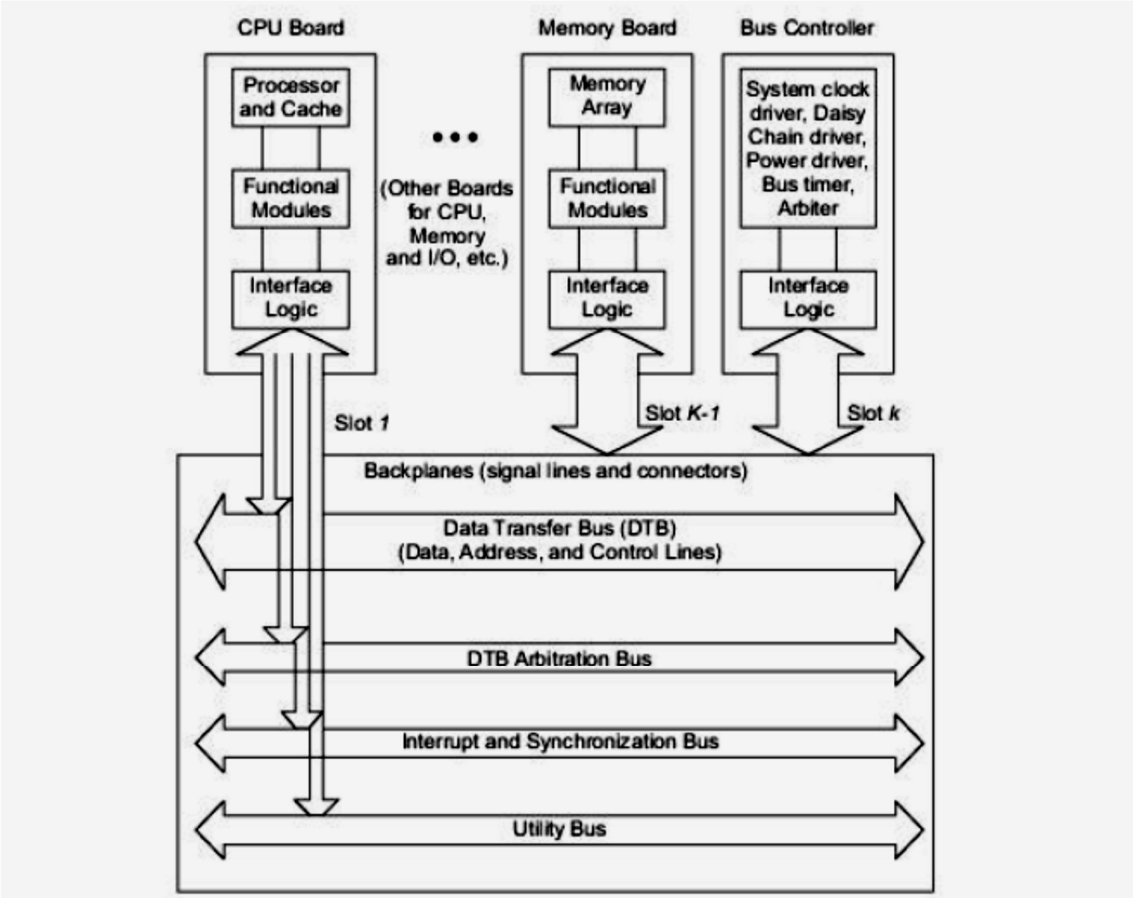


Fig. 5.1 Backplane buses, system interfaces, and slot connections to various functional boards in a multiprocessor system

Functional Module

A **Functinal module** is a collection of electronic circuitry that resides on one functional board as shown in Fig. 5.1 and works to achieve special bus control functions.

Special functional modules are as follows:

An arbiter is a functional module that accepts bus requests from requester module and grants control of the DTB to one requester at a time.

A **bus timer** measures the time each data transfer takes on the DTB and terminates the DTB cycle if a transfer takes too long.

An interrupter module generates an interrupt request and provides status/ID information when an interrupt handler module requests it.

A Location monitor is a functional module that monitors data transfers over the DTB. A power monitor watches the status of thc power source and signals when power becomes unstable.

A system clock driver is a module that provides a clock timing signal on thc utility bus. ln addition, board interface logic is needed to match the signal line impedance, the propagation time, and termination values between the backplanc 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.

5.1.2 Addressing and Timing Protocols

There are two types of IC chips or printed-circuit boards connected to a bus: **active and passive**. Active devices like processors can act as bus masters or as slaves at difierent times. Passive devices like memories can act only as slaves.

Bus adddressing

The backplane bus is driven by a digital clock with a fixed cycle time called the **bus cycle**. Each device can be identified with a **device number**. When the device number matches the contents of high order address lines, the device is selected as a slave. This addressing allows the allocation of a logical device address under software control, which increases the application flexibility.

Broadcall and Broadcast

A broadcall is a read operation involving multiple slaves placing their data on the bus lines. Special AND or OR operations over these data are performed on the bus from the selected slaves. Broadcall oprerations are used to detect multiplc interrupt sources.

A broadcast is a write operation involving multiple slaves. This operation is essential in implementing multicache coherence on the bus. Timing protocols are needed to synchronize master and slave operations. Figure 5.2 below 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.

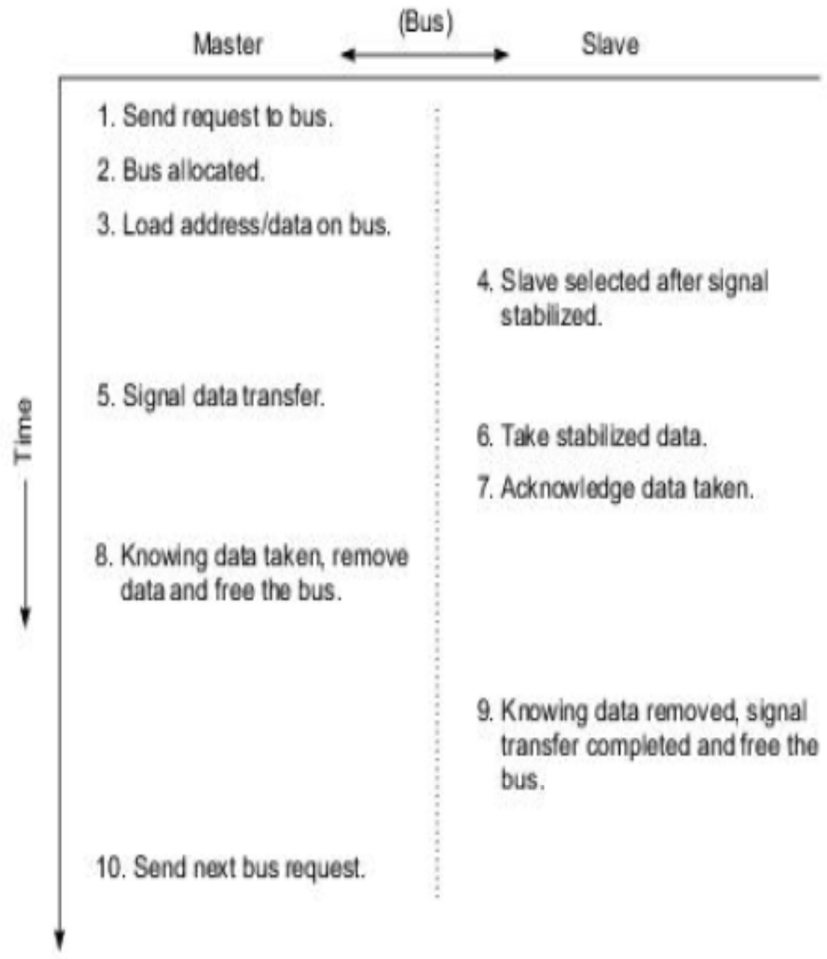
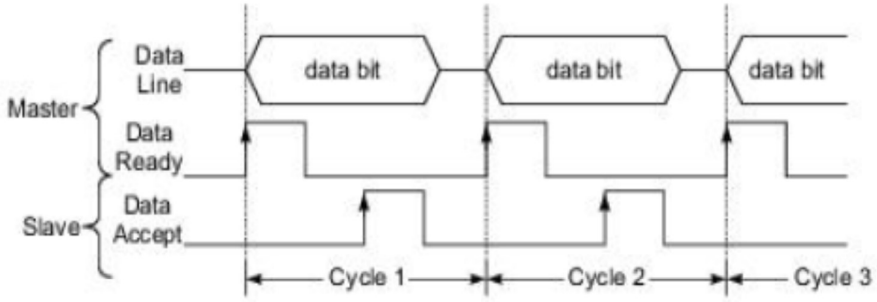


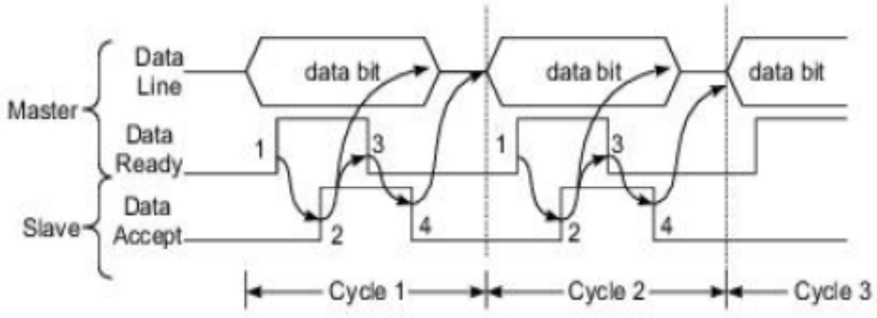
Fig. 5.2 Typical time sequence for information transfer between a master and a slave over a system bus

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. Once the data becomes stabilized on the data lines, the master uses a data-ready pulse to initiate the transfer. The slave uses a data-accept pulse to signal completion of the information transfer. A synchronous bus is simple to control, requires less control circuitry, and thus costs less. lt is suitable for connecting devices having relatively the speed. Otherwise, the slowest device will slow down the entire bus operation.



(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

It is based on a hand shaking or interlocking mechanism as illustrated in Fig. 5.3b. No fixed clock cycle is needed. The rising edge(1) of the

data-ready signal from the master triggers the rising (2) of the **data-accept** signal from the slave. The second signal triggers the falling(3) of the data-ready clock and the removal of data from the bus. The third signal triggers the trailing edge(4) of the data-accept clock. This four-edge handshaking process is repeated until all the data are transferred. The advantage of using an asynchronous bus lies in the freedom of using variable length clock signals for different speed devices.

Arbitration,Transaction, and Interrupt

The process of selecting the next bus master is called Arbitration. The duration of a master's control of the bus is called bus tenture.

Central Arbitration

As illustrated in below Fig. 5.4a, a central arbitration scheme uses a central arbiter, Potential masters are daisy-chained in a cascade. A special signal line is used to propagate a bus-grant signal level from the first master to the last master. Each potential master can send a bus request. All requests share the same bus-request line. As shown in Fig. 5.4b, the bus-request signals the rise of the bus-gram level which in turn raises the bus-busy level. A fixed priority is set in a daisy chain from left to right. When the bus transaction is complete, the bus-busy level is lowered, which triggers the falling of the bus grant signal and the subsequent rising of the bus-request signal.

The advantage of this arbitration scheme is its simplicity. Additional devices can be added anywhere in the daisy chain by sharing the same set of arbitration lines. **The disadvantage** is a fixed-priority sequence violating the fairness practice. Another drawback is its slowness in propagating the bus-grant signal along the daisy chain. Whenever a higher-priority device fails, all the lower-priority devices on the right of the daisy chain cannot use the bus.

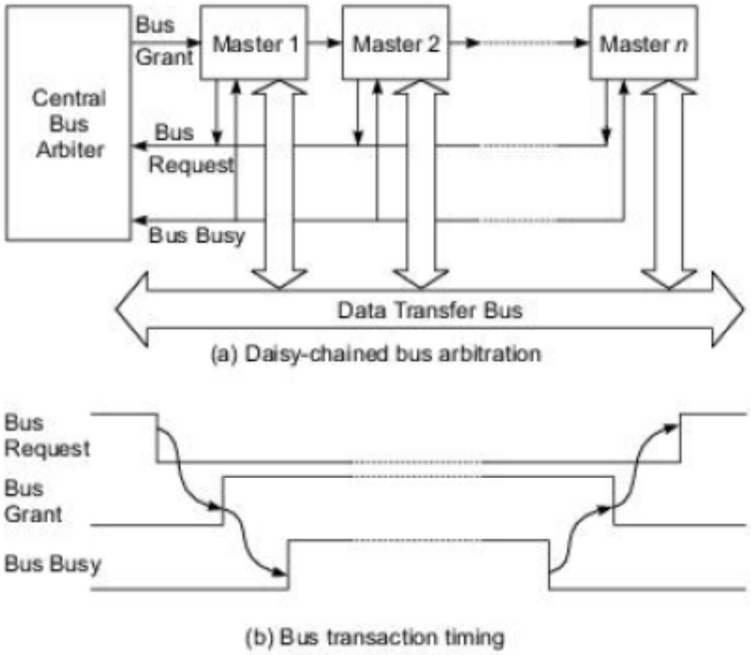


Fig. 5.4 Central bus arbitration using shared requests and daisy-chained bus grants with a fixed priority

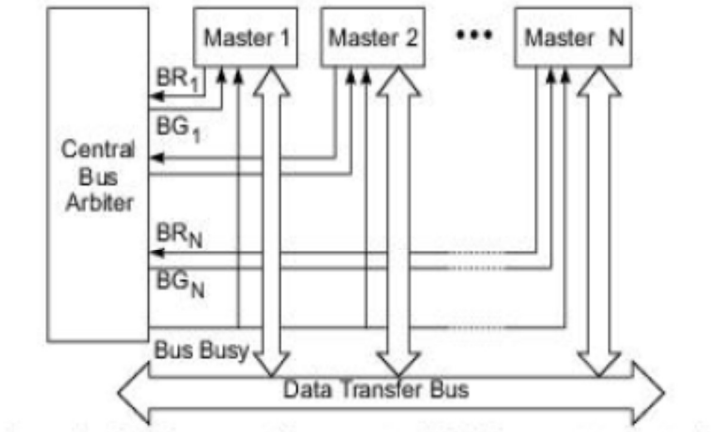
Independent Requests and Grant

The below Fig. 5.4.a, shows multiple bus-request and bus-grant signal lines which are independently provided for each potential master. No daisy-chaining is used in this scheme.The arbitration among potential masters is carried out by a central arbiter and any priority based or faimess-based bus allocation policy can be implemented. The advantage of using independent requests and grants in bus arbitration is their flexibility and faster arbitration time compared with the daisy-chained policy. The drawback is the large number of arbitration lines used.

Distributed arbitration

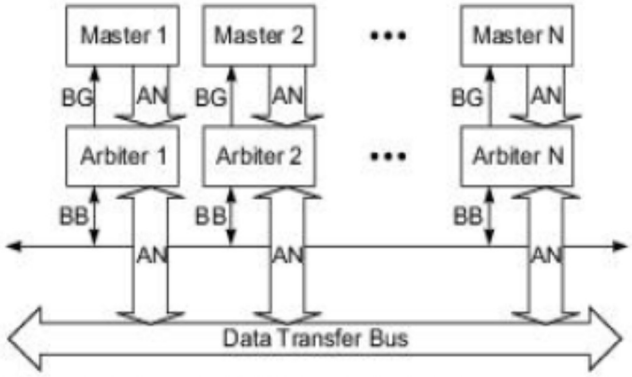
The idea of using distributed arbiters is depicted in below Fig. 5.5b. Each potential master is equipped with its own arbiter and a unique arbitration number. The arbitration number is used to resolve the arbitration competition. When two or more devices compete for the bus,

the winner is the one whose arbitration number is the largest. All potential masters can send their arbitration numbers to the shared-bus request/grant (SBRG) lines on the arbitration bus via their respective arbiters.Each arbiter compares the resulting number on the SHRG lines with its own arbitration 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. The distributed arbitration policy is priority-based. Multibus-ll and Futurebus+ adopted such a distributed arbitration scheme.



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

Transaction Mode

An address only transfer consists of an address transfer followed by no data. **A compelled data transfer** consists of an address transfer followed by a block of one or more data transfers to one or more contiguous addresses. **A packet data transfer** consists of an address transfer followed by a fixed length block of data transfers from a set of contiguous addresses.

A Connected transaction is used to carry out a master's request and a slave's response in a single bus transaction. **A split transaction splits** the request and response into separate bus transactions.Split transactions allow devices with a long data latency or access time to use the bus resources in a more efficient way. **A complete split** transaction may require two or more connected bus transactions.

Interrupt Mechanism

An 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. The interrupter must provide status and identification information. A functional module can be used to serve as an intcrmpt handler.

Priority interrupts are handled at many levels. For example, the VME bus uses seven interrupt-request lines. Up to seven interrupt handlers can be used to handle multiple interrupts. Interrupts can also be handled by message-passing using the data bus lines on a time-sharing basis. The use of time-shared data bus lines to implement interrupts is called Virtual interrupt.

5.1.4 IEEE Futurebus+ and Other Standards

The key features of the IEEE Futurebus-Standard 896.1-1991 are as follows:

1) Architecture, processor, and technology independent open standard available to all designers.

2) A fully asynchronous (compelled) timing protocol for data transfer with hand-shaking flow control.

3) An optional source-synchronized (packet) protocol for high-speed block data transfers.

4) Fully distributed parallel arbitration protocols to support a rich variety of bus transactions including broadcast, broadcall, and thrid party transactions.

5) Support of high reliability and fault-tolerant applications with provisions for live card insertion, removal, parity checks on all litres and feedback checking, and no daisy-chained signals to facilitate dynamic system reconfiguration in the event of module failure.

6) Use of multilevel mechanisms for the locking of modules and avoidance of deadlock or liveloek.

7) Circuit-switched and split transaction protocols plus support for memory commands for implementing remote lock and SIMD-like operations.

8) Support of real-time mission-critical computations with multiple priority levels and consistent priority treatment, plus support of a distributed clock synchronization protocol.

9) Support of 32 or 5 bit addressing with dynamically sized data buses from 32 to 64, 128, and 256 bits to satisfy different bandwidth demands.

10) Direct support of snoopy cache-based multiproccssors with recursive protocols to support large systems interconnected by multiple buses.

Technology Architecture Independence

Any bus standard should aim to achieve **technology independence** through basing the protocols on fundamental principles and optimizing them for maximum communication efficiency rather than a particular generation or type of processor. **Architecture independence** should provide a flexible general-purpose solution to cache consistency within which other cache protocols operate compatibly while at the same time providing an elegant unification with the message-passing protocols used in a multicomputer environment.

5.3 CACHE MEMORY ORGANIZATIONS

Most multiprocessor systems use private caches associated with different processors as shown in below Fig. 5.6. Caches can be addressed using either a physical address or a virtual address.

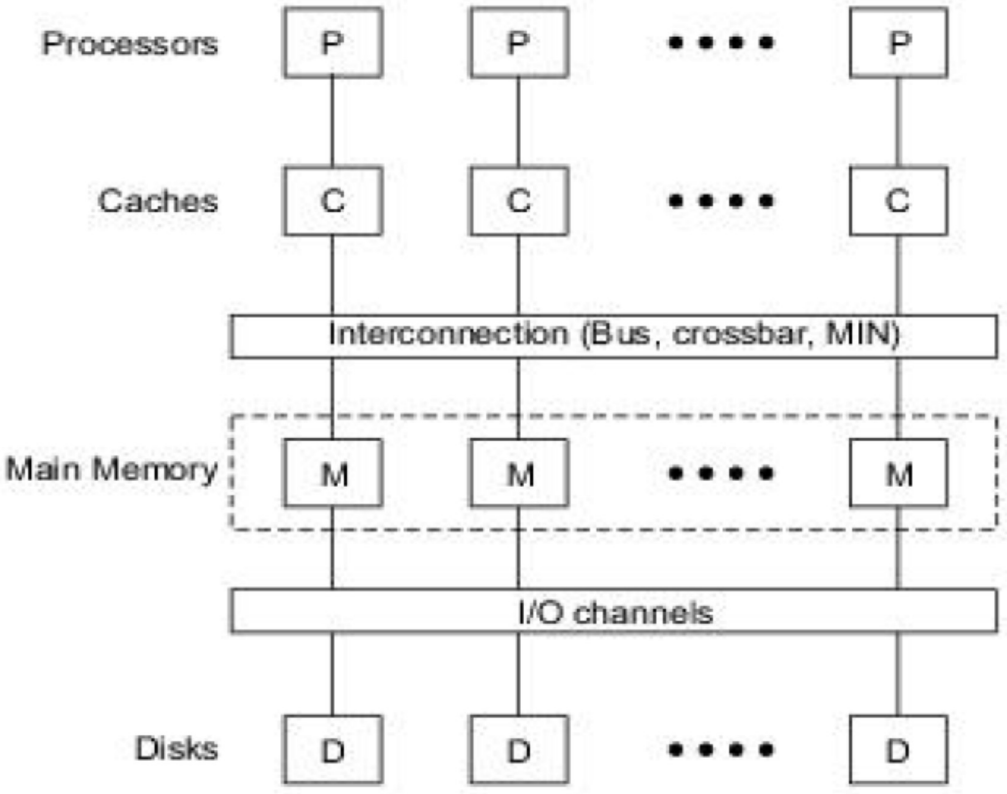


Fig. 5.6 A memory hierarchy for a shared-memory multiprocessor

Following are the two cache design models:

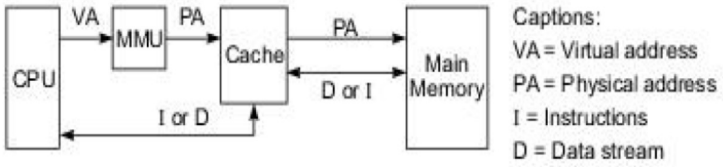
- Physical address cache
- Virtual address cache

Physical address Cache

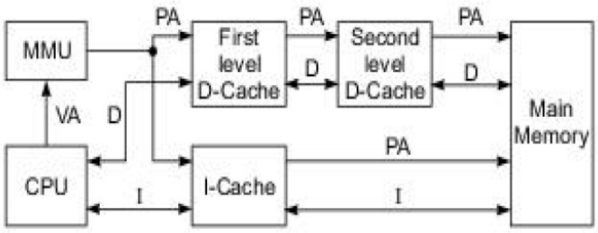
When a cache is accessed with a physical memory address, it is called a **physical address cache**. Physical address cache models are illustrated in below Fig. 5.7. The model is based on the experience of using a unified cache as in the VAX 8600 and the lntel i486. In this case, the cache is indexed and tagged with the physical address. A **cache hit** occurs when the addressed Data/Instruction is found in the cache. Otherwise, a **cache miss** occurs. After a miss, the cache is loaded with the data from the memory. Data is written through the main memory immediately by a **write-through (WT)** cache, or delayed until block replacement by using a **wite back(WB)** cache. The major **advantages of physical address caches** include no need to perform cache flushing, no aliasing problems, and thus fewer cache bugs in the OS kernels. The **shortcoming** is the slowdown in accessing the eache until the MMU ITLB finishes translating the address. This motivates the use of a virtual address cache.Most conventional system designs use a physical address cache because of its simplicity and because it requires little intervention from the OS kernel.

Example 5.1 Cache design in a Silicon Graphics workstation

Figure 5.7b demonstrates the split cache design using the MIPS R3000 CPU in the Silicon Graphics 4-D Series workstation. Both data cache and instruction cache are accessed with a physical address issued from the on-chip MMU. A two-level data cache is implemented in this design.



(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

The first level uses 64 Kbytes of WT D-cache. The second level uses 256 Kbytes of WB D-cache. The single-level I-cache is 64 Kbytes. By the inclusion property, the first-level cache is always a subset of the second-level cache.

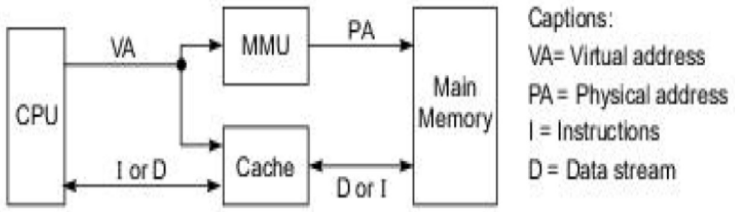
Virtual address cache

When a cache is indexed or tagged with a virtual address as shown in below Fig. 5.8, it is called a Virtual address cache. ln this model, both cache and MMU translation or validation are done in parallel.The virtual address cache is motivated with its enhanced efficiency to access the cache faster. The major problem associated with a virtual address cache is aliasing(the aliasing problem), it is when different logically addressed data have the same index tag in the cache. This aliasing problem may create confusion if two or more processes access the same physical cache location. One way to solve the aliasing problem is to

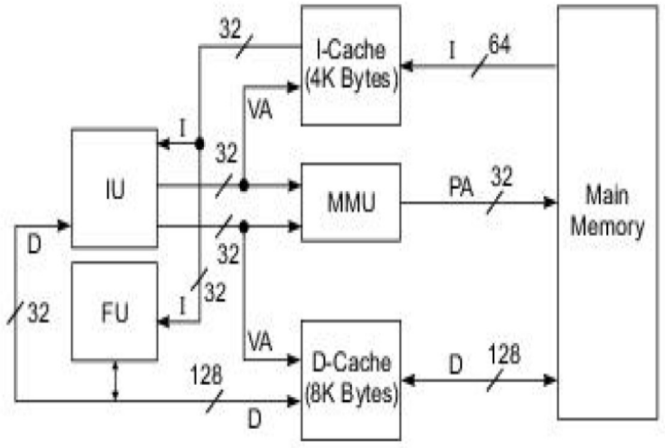
flush the entire eache when ever a context switch occurs. Flushing the cache does not overcome the aliasing problem completely when using a shared memory. Two commonly used methods to get around the virtual address cache problems are to apply special tagging with a process key or with a physical address. For example, the SUN 3/200Series has used a virtual address, write back cache with the capability of being noncacheable. Three-bit keys are used in the cache to distinguish among eight simultaneous contexts.

Example 5.2 The virtual addressed split cache design in Intel i860

Figure 5.8b shows the virtual address design in the Intel i860 using split caches for data and instructions. Instructions are 32 bits wide. Virtual addresses generated by the integer unit (IU) are 32 bits wide, and so are the physical addresses generated by the MMU. The data cache is 8 Kbytes with a block size of 32 bytes. A two-way set-associative cache organization (Sec. 5.2.3) is implemented with 128 sets in the D-cache and 64 sets in the I-cache.



(a) A unified cache accessed by virtual address



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

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

5.2.2 Direct Mapping and Associative Caches

Blocks in caches are called block frames in order to distinguish them from the corresponding blocks in main memory. Block frames are denoted as $\bar{B}_i$ for $i = 0, 1, 2, \ldots, m$. Blocks are denoted as B_j for j=0,1,...,n. Various mappings can be defined from set $\{B_j\}$ to set $\{\bar{B}_i\}$. It is also assumed that n>>m, $n=2^s$ and $m=2^r$. Each block (or block frame) is assumed to have b words, where $b = 2^w$ Thus the cache consists of $m*b = 2^{r+w}$ words. The main memory has $n*b = 2^{s+w}$ words addressed by (s+w) bits. when the block frames are divided into $v= 2^t$ sets, $k = m/v= 2^{r-t}$ blocks are in each set.

Direct mapping cache

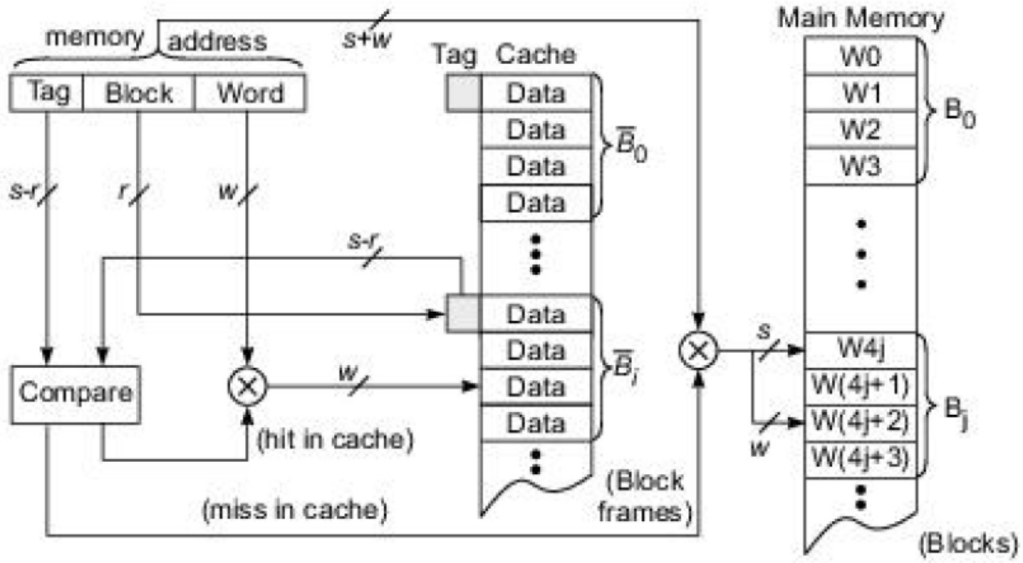
This cache organization is based on a direct mapping of $n/m = 2^{s-r}$ memory blocks, separated by equal distances, to one block frame in the cache. The placement is defined below using a modulo-m function.

Block B_j is mapped to block frame $\bar{B}_i$:

$$B_j \rightarrow \bar{B}_i, \quad \text{if } i = j \text{ (modulo } m) \tag{5.1}$$

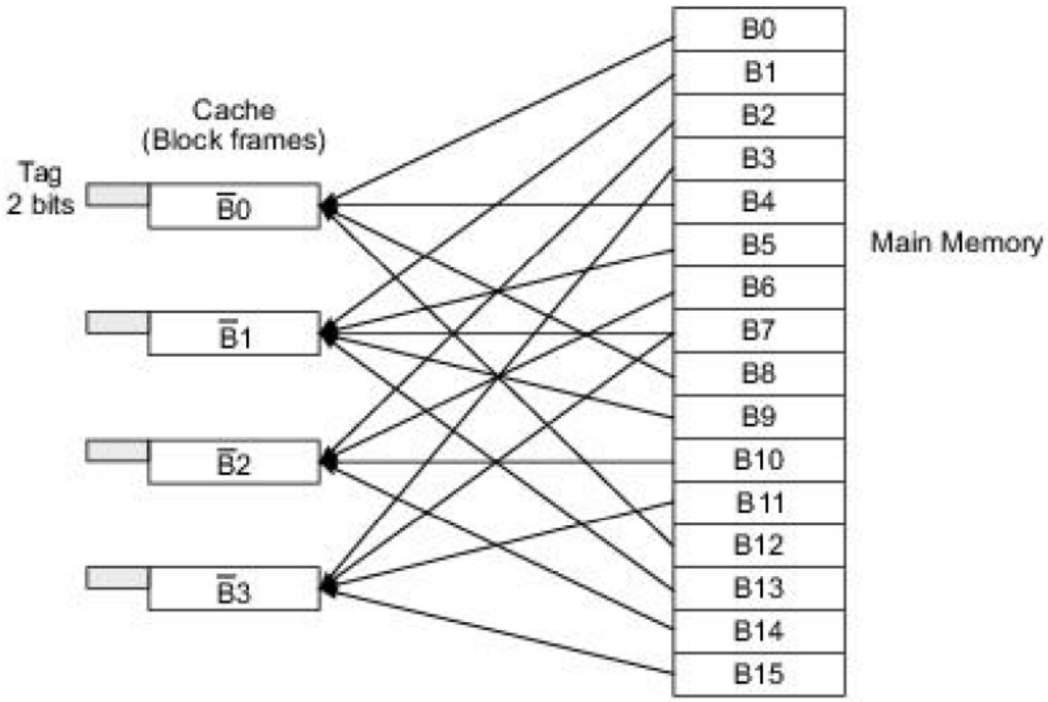
Direct mapping is illustrated in below Fig. 5.9a for the case where each block contains four words (w= 2 bits). The memory address is divided into three fields: The lower w bits specify the word offset within each block. The upper s bits specify the block address in main memory, while the leftmost (s - r) bits specify the tag to be matched. The block field (r bits) is used to implement the (modulo-m) placement, where $m = 2^r$. Once the block is $\bar{B}_i$ uniquely identified by this field, the tag associated with the addressed block is compared with the tag in the memory address. A cache hit occurs when the two tags match. Otherwise a cache miss occurs. In case of a cache hit, the word offset is used to identify the desired data word within the addressed block.

When a miss occurs, the entire memory address (s + w) bits is used to access the main memory. The first s bits locate the addressed block, and the lower w bits locate thc word within the block. **Advantages** of direct-mapping cachc includc simplicity in hardware, no associativc search needed, no page rcplacemcnt algorithm needed, and thus lowcr cost and highcr speed. The **disavantage** is that thc rigid mapping may result in a poorer hit ratio and also prohibits parallel virtual addrcss translation. The hit ratio may drop sharply if many addressed blocks have to map into the samc block frame. **For this reason, direct-mappcd caches tend to use a larger cache size** with morc block frames to avoid thc contention.



(a) The cache/memory addressing

An example mapping is given in below Fig. 5.9b, where n = 16 blocks are mapped to m = 4 block frames, with four possible sources mapping into one destination using modulo-4 mapping.



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

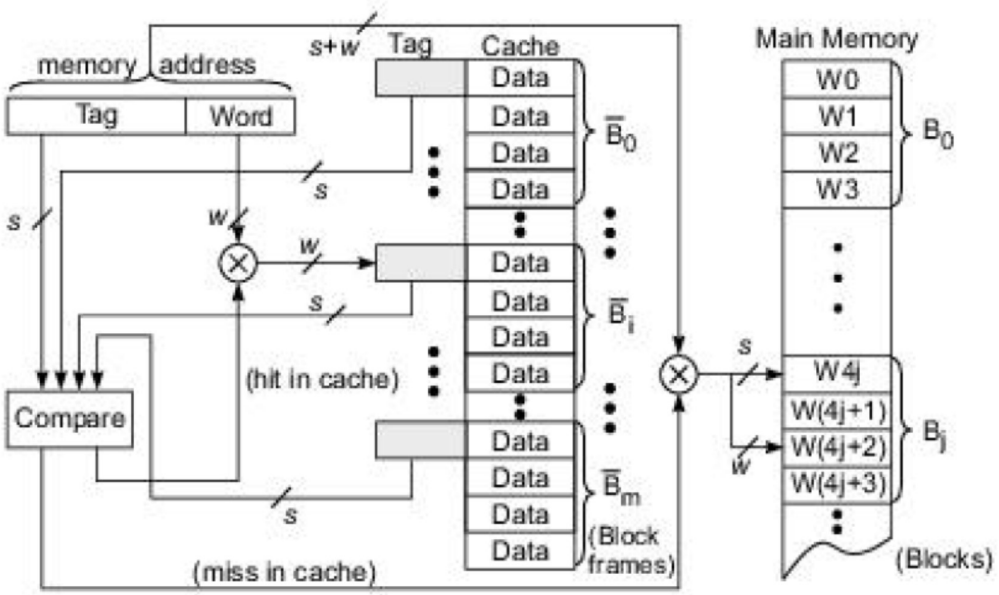
Cache Design Parameter

Consider a cache with 64 Kbytes. This implies m = 2^{11} = 2048 block frames with r = 11 bits. Consider a main memory with 32 Mbytes. Thus n = 2^{20} blocks with s = 20 bits, and the memory address needs s + w = 20+3 = 23 bits for word addressing and 25 bits for byte addressing. In this case, 2^{s-r}= 29 = 512 blocks are possiblc candidatcs to be mapped into a single block frame in a direct-mapping cache.

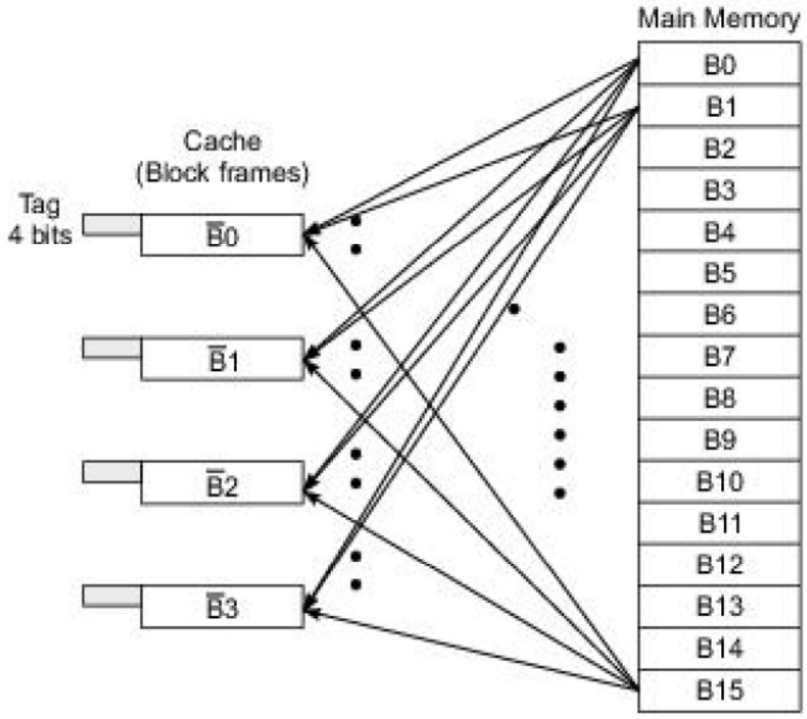
Fully Associative Cache

This cache organization offers the most flexibility in mapping cache blocks. As illustrated in below Fig. 5.10 a, each block in main memory can be placed in any of the available block frames. Because of this flexibility, an s bit tag is needed in cach cachc block. An m-way associativc search requires the tag to be compared with all block tags in the cache. This scheme offers the greatest flexibility in implementing

block rcplacemcnt policics for a highcr hit ratio. The below fig 5.10 b shows a four-way mapping example using a fully associative search. The tag is 4 bits long because 16 possible cachce blocks can bc destined for thc same block frame. Thc major advantage of using full associativity is to allow thc implcmcntation of a better block rcplacemcnt policy with reduced block contention. The major drawback lies in the cxpensive search process requiring a higher hardware cost.



(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

Set-Associative Cache

This design offers a compromise between the two extreme cache designs based on direct mapping and full associativity. Most high-performance systems are based this approach. The below fig 5.11a illustrates the set-associative cache design. ln a k-way associative search, the tag to be compared only with the k tags within the identified set

In general, a block B_j can be mapped into any one of the available frames B_f in a set S_i defined below. The matched tag identifies the current block which resides in the frame.

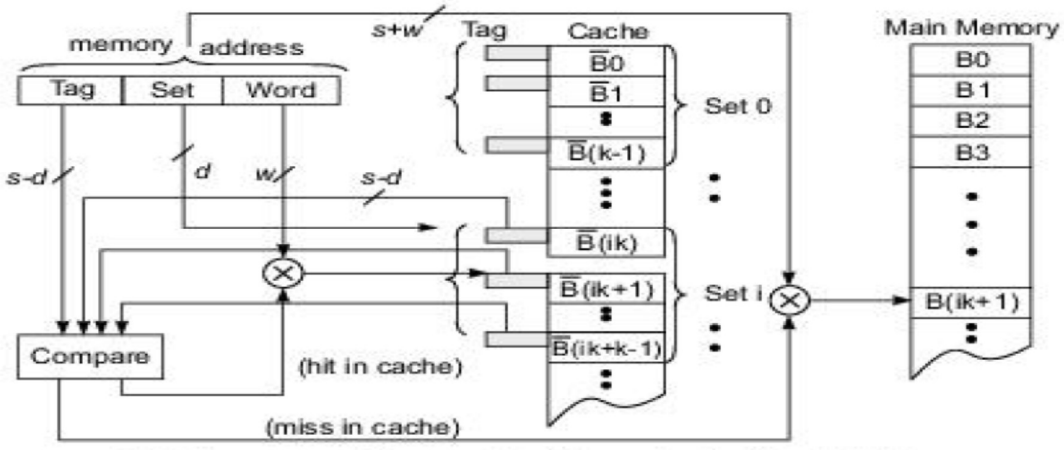
$$B_j \rightarrow \bar{B}_f \in S_i \quad \text{if} \quad j(\text{modulo } v) = i \tag{5.2}$$

Design Tradeoffs The set size (associativity) k and the number of sets v are inversely related by

$$m = v \times k \tag{5.3}$$

The advantages of the set-associative cache include the following:

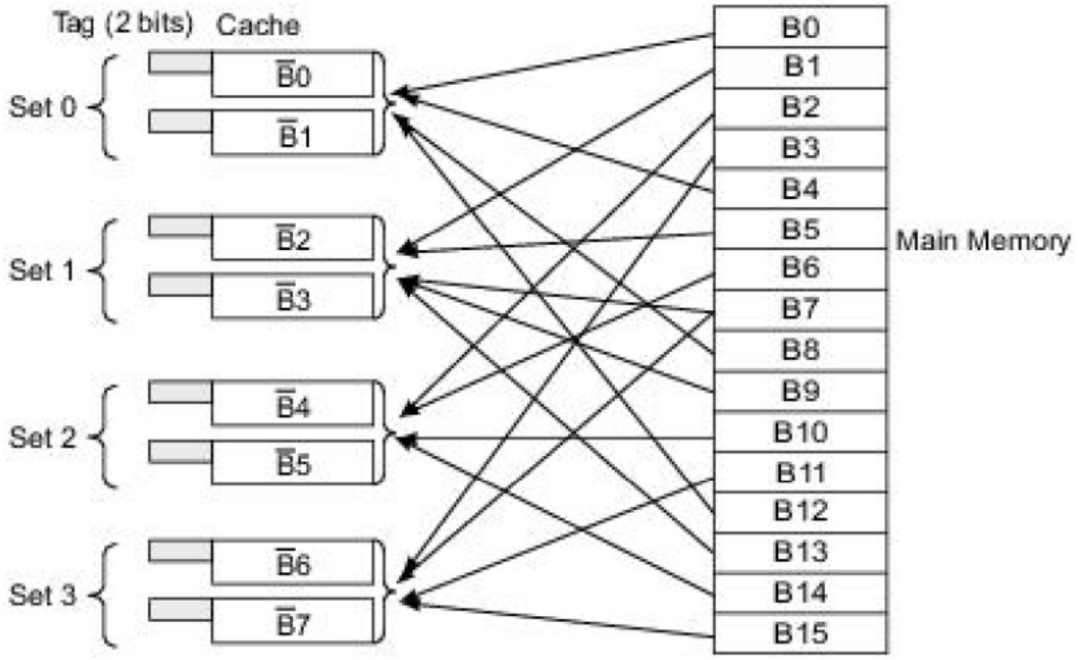
First, the block replacement algorithm needs to consider only a few blocks in the same set. Second, the k-way associative search is easier to implement. Third, many design tradeoffs can be considered (Eq. 5.3) to yield a higher hit ratio in the cache. The cache operation is often used together with TLB.



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

Example 5.4 Set-associative cache design and block mapping

An example is shown in Fig. 5.11b for the mapping of n = 15 blocks from main memory into a two-way associative cache (k = 2) with v = 4 sets over m = B block frames. For the i860, both the D-cache and I-cache are two-way associative (k = 2). There are 128 sets in the D-cache and 64 sets in the I-cache, with 256 and 123 block frames, respectively.



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

Sector Mapping Cache

This scheme partitions both the cache and main memory into fixed-size sectors. Then a fully associative search is applied. That is, each sector can be placed in any of the available sector frames. This scheme compares 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 an available

sector. A valid bit is attached to each block frame to indicate whether the block is valid or invalid. the sector mapp ing cache offers the advantages of being flexible to implcmesnt various block replacement algorithms and being economical to perform a fully associative search across a limited numher of sector tags. Figure 5.12 below shows an example of sector mapping with a sector size of four blocks. Each sector can be mapped to any of the sector frames with full associativity at the sector level.

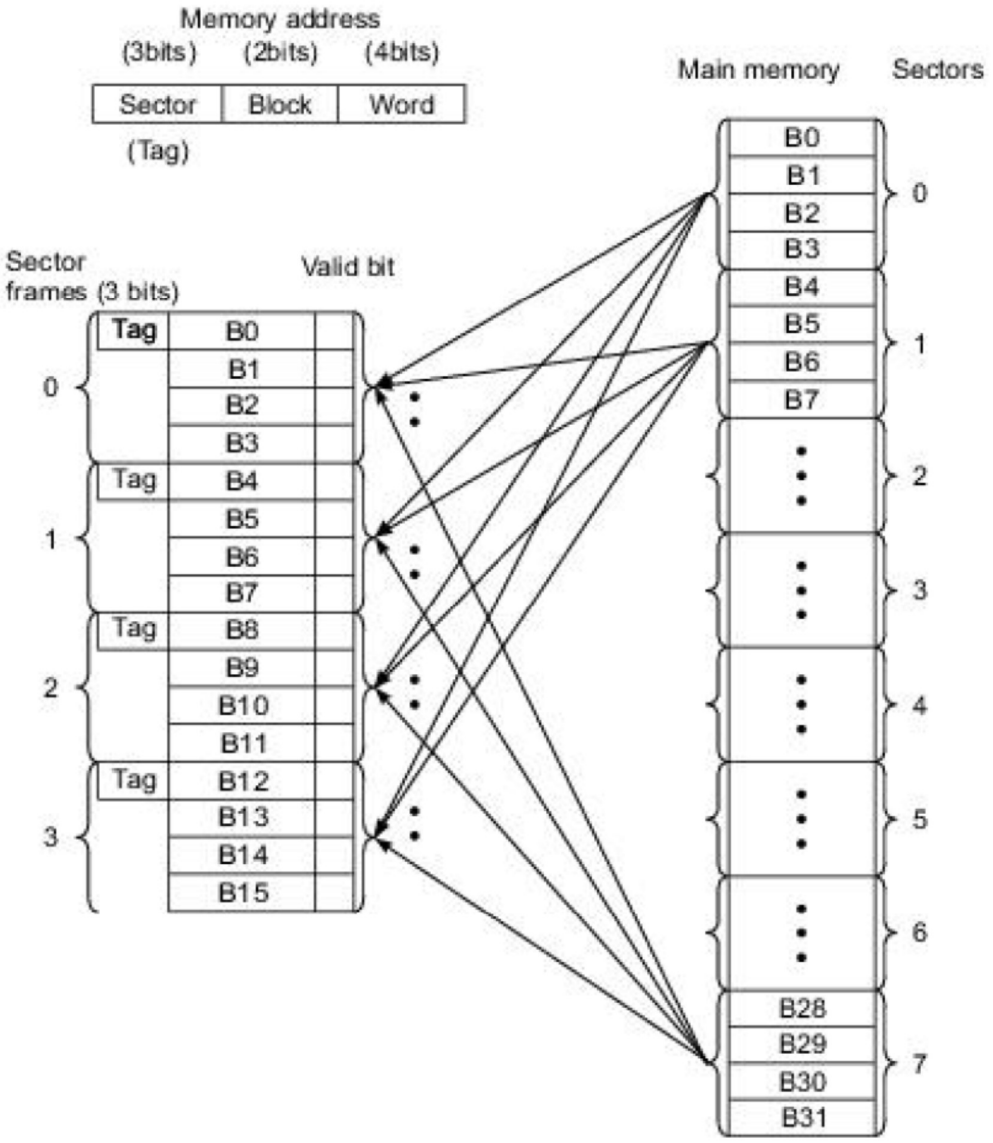


Fig. 5.12 A four-way sector mapping cache organization

5.2.4 Cache Performance Issues

The performance of a cache design concerns two related aspects: **the cycle count and the hit ratio.**

The **cycle count** refers to thc number of basic machine cycles needed for cache access, update, and coherence control.

The **hit ratio** determines how effectively the cache can reduce the overall memory-access time.

Program trace-driven simulation and analytical modeling are two complementary approaches to studying cache performance.

Cycle count

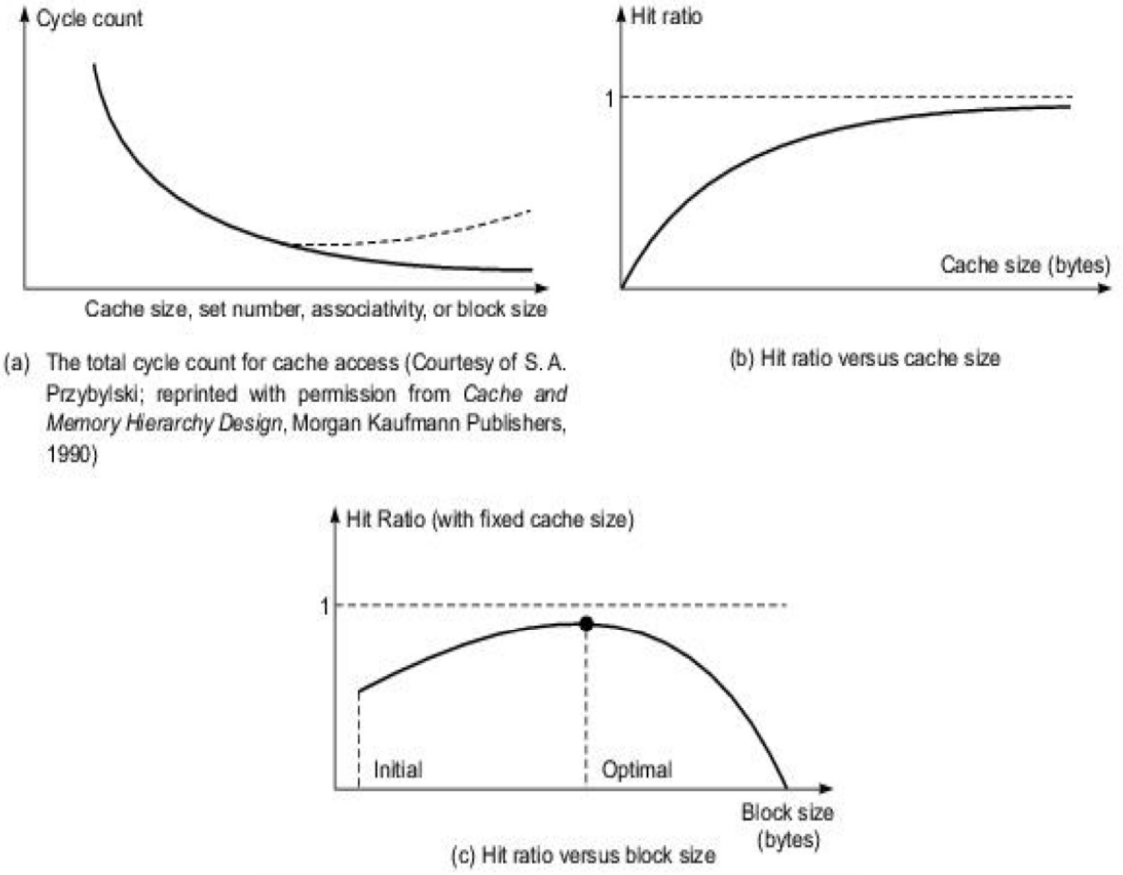
The write-through or write-back policies also affect the cycle count. Cache size, block size, set number, and associativity all affect the cycle count as illustrated in in below Fig. 5.13. The cycle count is directly related to the hit ratio, which decreases almost linearly with increasing values of the above cache parameters. But the decreasing trend becomes flat and after a certain point turns into an increasing trend (the dashed line in Fig. 5.13:a). This is caused primarily by the effect of the block size and the hit ratio.

Hit Ratios

The cache hit ratio is affected by the cache size and by the block size in different ways. These effects are illustrated in below Figs. 5.13b and 5.13c, respectively. Generally, the hit ratio increases with respect to increasing cache size (Fig 5. 13b).

Effect of Block Size

With a fixed cache size, cache perfonnance is rather sensitive to block size. Below Figure 5.13c illustrates the rise and an of the hit ratio as the cache block varies from small to large. This block sizc is determined mainly by the temporal locality in typical programs.



(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, 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, 123, and 256, the decrease in the hit ratio is small but when the set number increases to 512 and beyond, the hit ratio decreases faster.

Other Performance Factor

In a performance-directed design, independent blocks, fetch sizes, and fetch strategies also affect the performance in various ways. Multilevel cache hierarchies offer options for expanding the cache effects. Very often, a write-through policy is used in the first-level cache. and a write-back policy in the second-level cache.

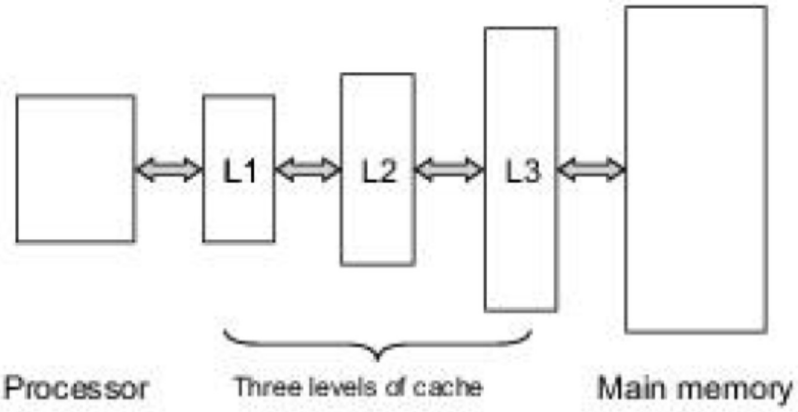


Fig. 5.14 Three levels of cache between processor and main memory

5.3 SHARED-MEMORY ORGANIZATIONS

5.3.1 Interleaved Memory Organization

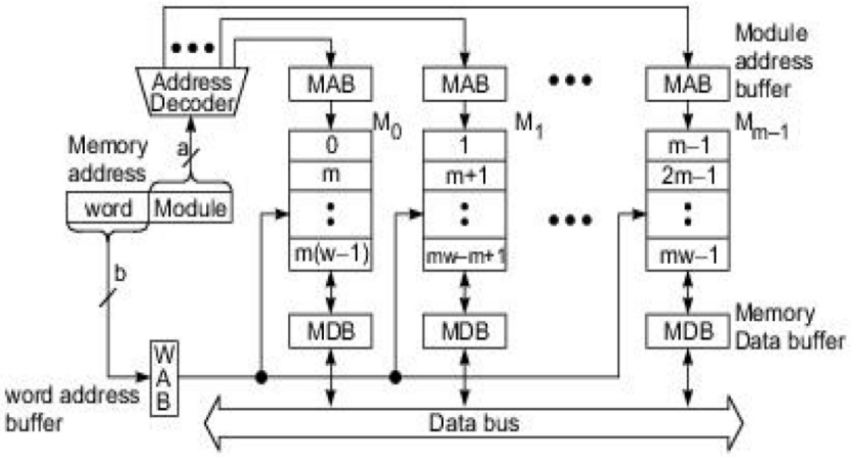
Memory interleaving allows pipelined access of the parallel memory modules and broadens the effective memory bandwidth.

Lower order inrerterleaving

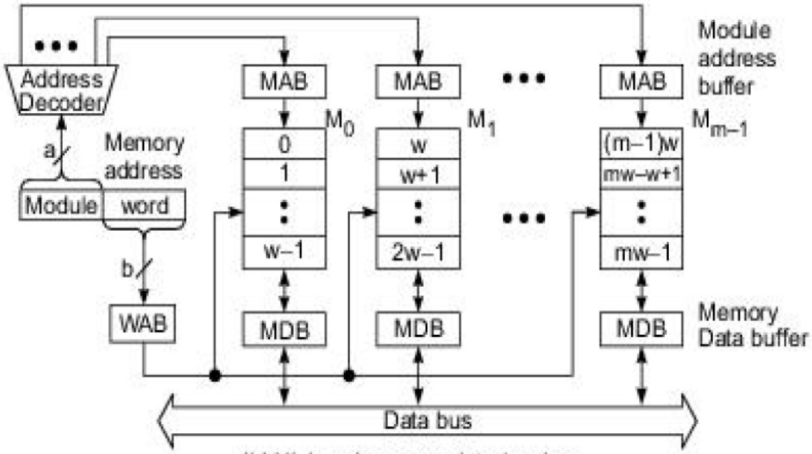
Below Figure 5.15a shows lower order memory interleaving which spreads contiguous memory locations across the m modules horizontally. This implies that the low-order **a** bits of the memory address are used to identify the memory module. The high-order **b** bits are the word addresses (displacement) within each module. A module address decoder is used to distribute module addresses. lower-order interleaving support block access.

High-order interleaving

As shown in below Fig. 5.15b higher 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. ln each memory cycle, only one word is accessed from each module. Thus the high-order interleaving cannot support block access of contiguous locations.



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

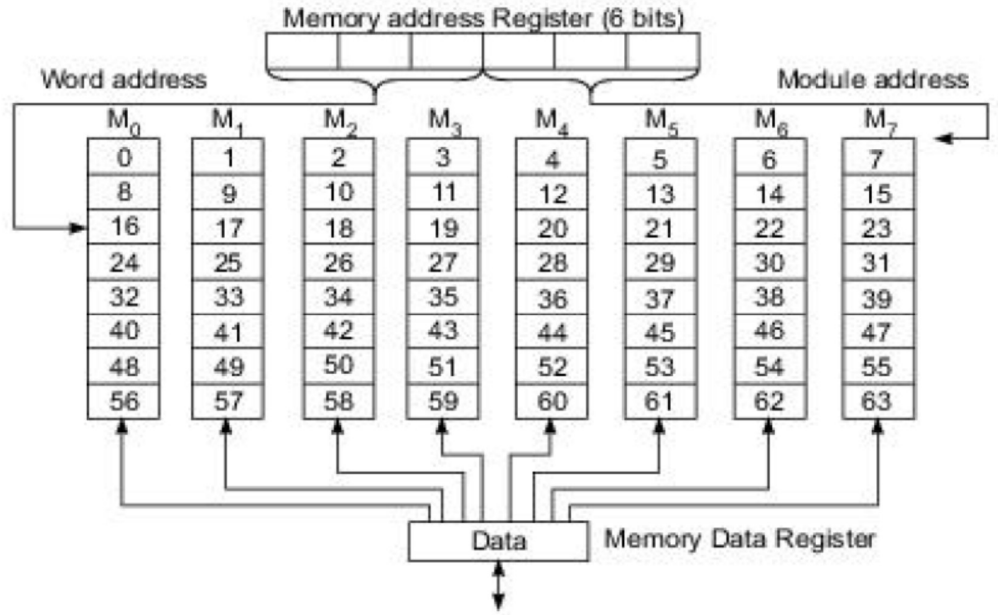


(b) High-order *m*-way interleaving

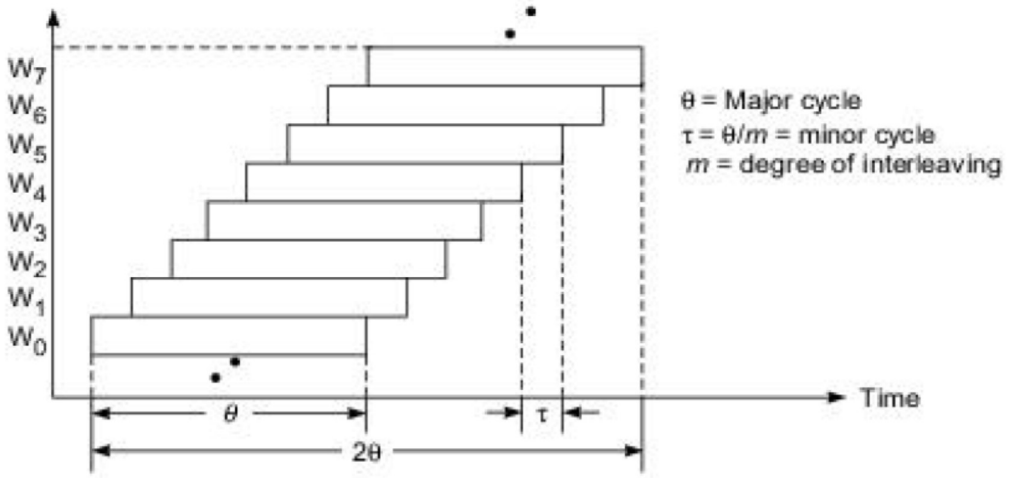
Fig. 5.15 Two interleaved memory organizations with $m = 2^a$ modules and $w = 2^b$ words per module (wo addresses shown in boxes)

Pipelined Memory Access

Access of the m memory modules can be overlapped in a pipelined fashion, For which the memory cycle (called 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 below Fig. 5.16a Let θ be the major cycle and the minor τ cycle. These two cycle times are related as follows: where m is $\tau = \frac{\theta}{m}$ the degree of interleaving.



(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

The timing of the pipelined access of the eight contiguous memory words is shown in above Fig. 5.16b. This type of concurrent access of contiguous words has been called a C-access memory scheme. The major cycle θ is the total time required to complete the access of a single word from a module. The minor cycle τ is the actual time needed to produce one word, assuming overlapped access of successive memory modules separated in every minor cycle.

5.3.2. Bandwidth and Fault Tolerance

Memory Bandwidth

The memory bandwidth B of an m-way interleaved memory is upper bounded by m and lower-botmded by 1. The Hellerman estimate of B is

$$B = m^{0.56} \simeq \sqrt{m}$$

where m is the number of interleaved memory modules.

ln 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 t_1, required to acoess one element in a vector is estimated by

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

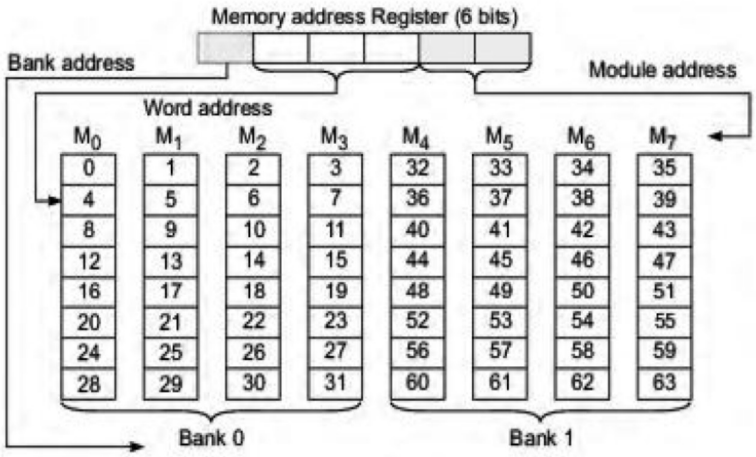
When $n \to \infty$ (very long vector), $t_1 \to \theta/m = \tau$ as derived in Eq. 5.4. As $n \to 1$ (scalar access), $t_1 \to \theta$. Equation 5.6 conveys the message that interleaved memory appeals to pipelined access of long vectors; the longer the better.

Fault Tolerance

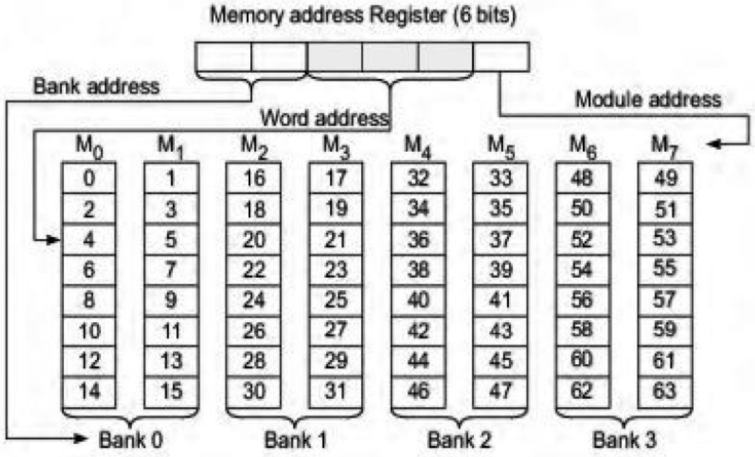
Sequential addresses are assigned in the high-order interleaved memory in each memory module. This makes it easier to isolate faulty memory modules in a memory bank of m memory modules. When one module failure is detected, the remaining modules can still be used by opening a window in the address space. Thus low-order interleaving memory is not fault-tolerant.

Example 5.6 Memory banks, fault tolerance, and bandwidth tradeoffs

In Fig. 5.17, two alternative memory addressing schemes are shown which combine the high- and low-order interleaving concepts. These alternatives offer a better bandwidth in case of module failure. A four-way low-order interleaving is organized in each of two memory banks in Fig. 5.17a.



(a) Four-way interleaving within each memory bank



(b) Two-way interleaving within each memory bank

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

5.3.3 Memory Allocation Schemes

The portion ofthe OS kemel which handles the allocation and deallocation of main memory to csecuting processes is called the **memory manager.**

Allocation policies

Memory swaping is the process of moving blocks of information between the levels of a memory hierarchy.

The swapping policy can be made either **nonpreemptive or preemptive**.

ln nonpreemptive allocation, the incoming block can be placed only in a free region of the main memory. A preemptive allcation scheme allows the placement of an incoming block in a region presently occupied by another process.

When the main memory space is fully allocated, the nonpreemptive scheme swaps out some of the allocated processes (or pages) to vacate space for the incoming block. A preemptive scheme has the freedom to preempt an executing process.

The nonpreemptive scheme is easy to implement, but it may not yield the best memory utilization.The preemptive scheme offers more flexibility, but it requires mechanisms be established to determine which pages or processes are to be swapped out and to avoid trashing caused by an excessive amount of swapping between memory levels.

Swapping System

This refers to memory systems which allow swapping to occur only at the entire process level. A swap device is a configurable section of a disk which is set aside for temporary storage of information being swapped out of the main memory. The portion of the disk memory space set aside for a swap device is called the swap space, as show in below Fig. 5.13.

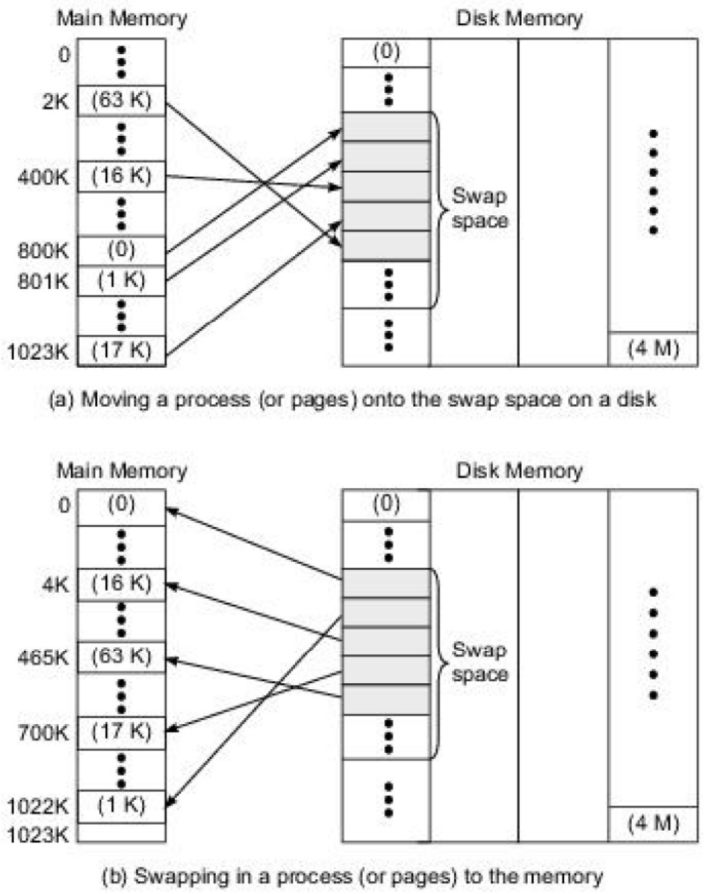


Fig. 5.18 The concept of memory swapping in a virtual memory hierarchy (virtual page addresses are identified by numbers within parentheses, assuming a page size of 1 K words)

A simple example is shown in above Fig. 5.18 to illustrate the concepts of swapping out and Swapping in a process consisting of five resident pages identified by virtual page addresses 0, IK. 16K. 17K. and 63k with an assumed page size of 1K words (or 4 Kbytes for a 32-bit word length). Figure 5.18a shows the allocation of physical memory before the swapping. The main memory is assumed to have 1024 page Frames, and the disk can accommodate 4M pages. The five resident pages, scattered around the main memory, are swapped out to the swap device in contiguous pages as shown by the shaded boxes. Later on, the entire process may be required to swap beck into the main memory, as depicted in Fig. 5. 18b.

Swapping in UNIX

In UNIX/OS, the kernel swaps out a process to create free memory space under the following system calls:

- The allocation of space fora child process being created.
- The increase in the size ofa process address space.
- The increased space demand by the stack ibr a process.
- The demand for space by a returning process swapped out previously.
- A special process 0 is reserved as a swapper.

Demand Paging System

Demanad paging memory allocation policy allows only pages (instead of processes) to be transferred between the main memory and the swap device. In Fig. 5.18, individual pages of a process are allowed to be independently swapped out and in, which makes a demand paging system. The major advantage of demand paging is that it offers the flexibility to dynamically accomodate a large number of processes in the physieal memory on a time-sharing or multiprogrammcd basis with significantly enlarged address spaces. It matches nicely with the working-set concept. If the lremel keeps only the working sets with a suflieicrntly large window in the main memory, many more active processes can concurrently reside in the memory than the swapping system can provide. Thus undemanded pages are not involved in the swapping process.

Example 5.7 Working sets generated with a page trace

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

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

Hybrid Memory System

The VAX/VMS and UNIX System V are hybrid memory systems which combine the advantages of both swapping and demand paging. When several processes simultaneously are in the ready-to-run-but-swapped state, the swappcr may choose to swap out several processes entirely to vacate the needed space. This scheme may lower the page fault rate and reduce thrashing. Other virtual memory systems may use anticipatory paging, which prefetches pages based on anticipation. This scheme is rather difficult to implement.

5.4 SEQUENTIAL AND WEAK CONSISTENCY MODELS

5.4.1 Atomicity and Event Ordering

The problem of memory inconsistency arises when the memory-access order differs from the program execution order. As illustrated in below Fig. 5.19a, A uniprocessor system maps an SISD sequence into a similar execution sequence. Thus memory accesses (for instructions and data) are consistent with the program execution order. This properly has been called sequential consistency.

In a shared-memory multiprocessor. there are multiple instruction sequences in different processors as shown in below Fig. 5.l9b. Different ways of interleaving the MIMD instruction sequences into a global memoryaccess sequence lead to different shared memory behaviors.

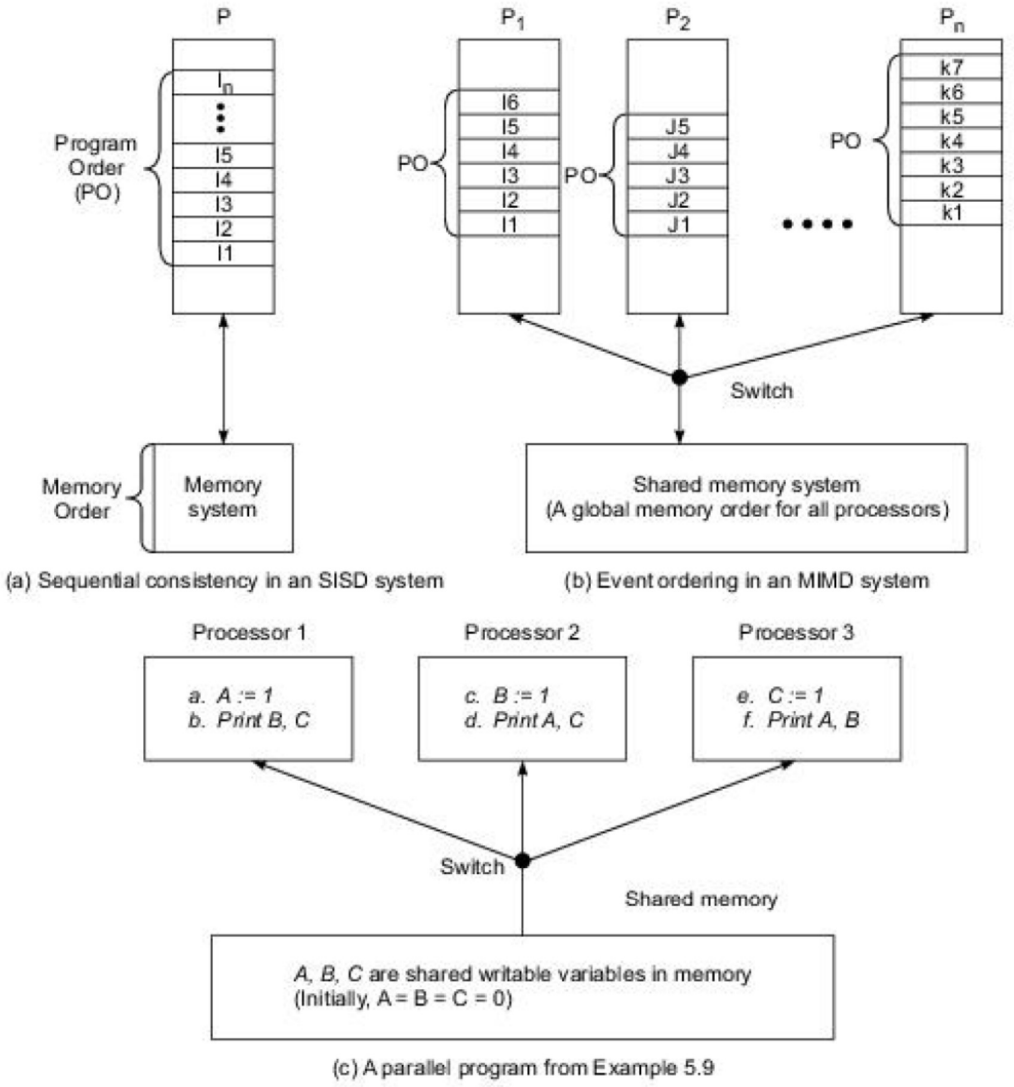


Fig. 5.19 The access ordering of memory events in a uniprocessor and in a multiprocessor, respectively (Courtesy of Dubois and Briggs, *Tutorial Notes on Shared-Memory Multiprocessors*, Int. Symp. Computer Arch., May 1990)

Memory Consistency Issue

The behavior of shared- memory system as observed by processors is called a memory model. Specification of the memory model answers three fundamental questions:

(1) What behavior should a programmer/compiler expect from a shared-memory multiprocessor?

(2) How can a definition of the expected behavior guarantee coverage of all contingencies?

(3) How must processors and the memory system behave to ensure consistent adherence to the expected behavior of the multiprocessor?

Event Ordering

Memory events correspond to shared-memory accesses. Consistency models specify the order by which the events from one process should be observed by other processes in the machine.

The event ordering can be 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.

The three primitive memory operations for the purpose of specifing memory consistency models are:

1)A load by processor P_i, is considered , performed with respect to processor P_k, at a point of time when the issuing of a store to the same location by P_k cannot affect the value returned by the load.

2) A store by P_i, is considered performed with respect to P_k at one time when an issued load to the same address by 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 thc store that is the source of the retumed 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 resequencc instructions in order to boost performance.

when a processor in a multiprocessor system executes a concurrent program as illustrated in Fig. 5.l9b, local dependence checking is necessary but may not be sufficient to preserve the intended outcome of a concurrent execution.

Maintaining the correctness and predictability of the execution results is rather complex on an MIMD system for the following reasons:

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 possible.

c) lf 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

Multiprocessor memory behavior can be described in three categories:

1) Program order preserved and uniform observation sequence by all processors.

2) Out-of-program-order allowed and uniform observation sequence by all processors.

3) Out-of-program-order allowed and nonuniform sequences observed by different processors.

This behavioral categorization leads to two classes of shared-memory systems for multtiprocessors: The first allows atomic memory access,

and the second allows nonatomic memory access. A shared-memory access is atomic if the memory updates are known to all processors at the same time. Thus a store is atomic if the value stored becomes readable to all processors at the sarne time.

Example 5.8 Event ordering in a three-processor system (Dubois, Scheurich, and Briggs, 1988)

To illustrate the possible ways of interleaving concurrent program executions among multiple processors updating the same memory, we examine the simultaneous and asynchronous executions of three program segments on the three processors in Fig. 5.19c.

The shared variables are initially set as zeros, and we assume a *Print* statement reads both variables indivisibly during the same cycle to avoid confusion. If the outputs of all three processors are concatenated in the order P_1, P_2, and P_3, then the output forms a 6-tuple of binary vectors.

There are $2^6 = 64$ possible output combinations. If all processors execute instructions in their own program orders, then the execution interleaving a, b, c, d, e, f is possible, yielding the output 001011. Another interleaving, a, c, e, b, d, f, also preserves the program orders and yields the output 111111.

If processors are allowed to execute instructions out of program order, assuming that no data dependences exist among reordered instructions, then the interleaving b, d, f, e, a, c is possible, yielding the output 000000.

Out of 6! = 720 possible execution interleavings, 90 preserve the individual program order. From these 90 interleavings not all 6-tuple combinations can result. For example, the outcome 000000 is not possible if processors execute instructions in program order only. As another example, the outcome 011001 is possible if different processors can observe events in different orders, as can be the case with replicated memories.

5.4.2 Sequential Consistency Model

The sequential consistency(SC) memory model the loads,stores and swaps of all processors appear to execute serially in a single global memory order that conforms to the individual program orders of the processors, as illustrated in below Fig. 5.20.

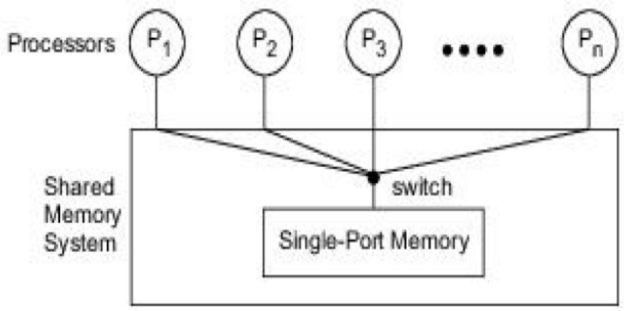


Fig. 5.20 Sequential consistency memory model (Courtesy of Sindhu, Frailong, and Cekleov; reprinted with permission from *Scalable Shared-Memory Multiprocessors*, Kluwer Academic Publishers, 1992)

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 exocuted in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

Dubois, Seheurich, and Briggs (1986) have provided the following two sufficient conditions to achieve sequential consistency in shared-memory access:

(a) Before a load is allowed to perform with respect to any other prooessor, all previous food accesses must be globally performed and all previous store accesses must be performed with respect to all processors.

(b) Before a store is allowed to perform with respect to any other processor, all previous load accesses must be globally performed and all previous store accesses must be performed with respect to all processors.

Sindhu, Frailong, and Cekleov (I992) have specified the sequential consistency memory model with the following five axioms:

1) A load by a processor always returns the value written by the latest store to the same location by other processors.

2) The memory order conforms to a total binary order in which shared memory is accessed in rcal time overall loads and stores with respect to all processor pairs and location pairs.

3) lf two operations appear in a particular program order, then they appear in the same memory order.

4) The swap operation is atomic with respect to other stores. No other store can intervene between the load and store parts of a swap.

5) All stores and swaps must eventually terminate.

Implemantation Considerations

The above Figure 5.20 shows that the shared memory consists of a single port that is able to service exactly one operation at a time, and a switch that connects this memory to one of the processors for the duration of each memory operation. The order in which the switch is thrown from one processor to another determines the global order of memory-access operations. The sequential consistency model implies total ordering of stores/loads at the instruction level. Sequential consistency must hold for any processor in the system. Strong ordering of all shared-memory accesses in the sequential consistency model preserves the program order in all processors. A sequentially consistent multiprocessor cannot determine whether the system is a multitasking uniproccssor or a multiprocessor. interprocessor communication can be implemented with simple loads/stores, such as Dekkcr's algorithm for synchronized entry into a critical scction by multiple processors. All memory accesses must be globally performed in program order.

5.4.3 Weak Consistency Models

The multiprocessor memory model may range anywhere from strong (or sequential) consistency to various degrees of weak consistency.

DSB Model Dubois, Scheurich, and Briggs [198-6) have derived a weak consistency memory model by relating memory request ordering to synchronization points in the program.

The DSB model specified by the following three conditions:

1)All previous synchronization accesses must be performed, before a load or a store access is allowed to perform with respect to any other processor.

2) All previous load and store aceesses must be performed, before a synchronization access is allowed to perform with respect to any other processor

3)Synchronization accesses are sequentially consistent with respect to one another.

A TSO Formal Specification(weak consistency model axioms)

1) A load access is always returned with the latest store to the same memory location issued by any processor in the system.

2) The 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) lf a memory operation follows a food in program order, then it must also follow the load in memory order.

5) A swap operation is atomic with respect to other stores. No other store can interleave between the lined and store parts of a nirrp.

6) All stores and swaps must eventually terminate.

Example 5.9 The TSO weak consistency model used in SPARC architecture (Sun Hicriosystems, lnc,1990 and Sindhu et al.,1992)

Figure 5.21 below shows the weak consistency TSO model developed by Sun Microsystems SPARC architecture group. Sindhu ct al. described that the stores and swaps issued by a processor are placed in a dedicated store buffer for the processor, which is operated as first-in-first-out. Thus the order in which memory executes these operations for a given processor is the same as the order in which the processor issued them (in program order). A load by a processor first checks its store buffer to see if it contains a store to the same location. lf it does, then the load rerums the value of the most recent such store. Otherwise, thc lood goes directly to memory.

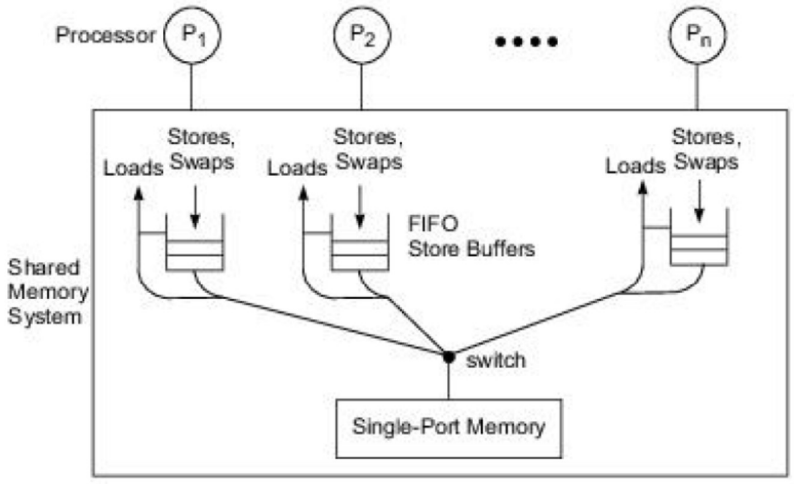


Fig. 5.21 The TSO Weak consistency memory model (Courtesy of Sindhu, Frailong, and Cekleov; reprinted with permission from *Scalable Shared-Memory Multiprocessors*, Kluwer Academic Publishers, 1992)

Comparison of Memory Model

The weak consistency model may offer better performance than the sequential consistency model at the expense of more complex hardware /software support and more programmer awareness of the imposed restrictions.

The DSB and the TSO are two different weak-consistency memory models. The DSB model is weakened by enforcing sequential consistency at synchronization points. The TSO model is weakned by treating reads,stores, and swaps differently using FIFO store buffers.

Sindhu et al. (1992) have identified four system-level issues which also affect the choice of memory model. First, they suggest that one should consider extending the memory model from the processor level to the process level. To do this, one must maintain a process switch sequence. The second issue is the incorporation of I/O locations into the memory model. I/O operations may introduce even more side effects in addition to the normal semantics loads and stores The third issue is code modification.