13-1 Characteristics of Multiprocessors

A multiprocessor system is an interconnection of two or more CPUs with memory and input–output equipment. The term "processor" in *multiprocessor* can mean either a central processing unit (CPU) or an input–output processor (IOP). However, a system with a single CPU and one or more IOPs is usually not included in the definition of a multiprocessor system unless the IOP has computational facilities comparable to a CPU. As it is most commonly defined, a multiprocessor system implies the existence of multiple CPUs, although usually there will be one or more IOPs as well. As mentioned in Sec. 9-1, multiprocessors are classified as multiple instruction stream, multiple data stream (MIMD) systems.

MIMD

There are some similarities between multiprocessor and multicomputer systems since both support concurrent operations. However, there exists an important distinction between a system with multiple computers and a system with multiple processors. Computers are interconnected with each other by means of communication lines to form a *computer network*. The network consists of several autonomous computers that may or may not communicate with each other. A multiprocessor system is controlled by one operating system that provides interaction between processors and all the components of the system cooperate in the solution of a problem.

microprocessor

VLSI

Although some large-scale computers include two or more CPUs in their overall system, it is the emergence of the microprocessor that has been the major motivation for multiprocessor systems. The fact that microprocessors take very little physical space and are very inexpensive brings about the feasibility of interconnecting a large number of microprocessors into one composite system. Very-large-scale integrated circuit technology has reduced the cost of computer components to such a low level that the concept of applying multiple processors to meet system performance requirements has become an attractive design possibility.

Multiprocessing improves the reliability of the system so that a failure or error in one part has a limited effect on the rest of the system. If a fault causes one processor to fail, a second processor can be assigned to perform the functions of the disabled processor. The system as a whole can continue to function correctly with perhaps some loss in efficiency.

The benefit derived from a multiprocessor organization is an improved system performance. The system derives its high performance from the fact that computations can proceed in parallel in one of two ways.

1. Multiple independent jobs can be made to operate in parallel.
2. A single job can be partitioned into multiple parallel tasks.

An overall function can be partitioned into a number of tasks that each processor can handle individually. System tasks may be allocated to special-purpose processors whose design is optimized to perform certain types of processing efficiently. An example is a computer system where one processor performs the computations for an industrial process control while others monitor and control the various parameters, such as temperature and flow rate. Another example is a computer where one processor performs high-speed floating-point mathematical computations and another takes care of routine data-processing tasks.

Multiprocessing can improve performance by decomposing a program into parallel executable tasks. This can be achieved in one of two ways. The user can explicitly declare that certain tasks of the program be executed in parallel. This must be done prior to loading the program by specifying the parallel executable segments. Most multiprocessor manufacturers provide an operating system with programming language constructs suitable for specifying parallel processing. The other, more efficient way is to provide a compiler with multiprocessor software that can automatically detect parallelism in a user's program. The compiler checks for *data dependency* in the program. If a program depends on data generated in another part, the part yielding the needed data must be executed first. However, two parts of a program that do not use data generated by each can run concurrently. The parallelizing compiler checks the entire program to detect any possible data dependencies. These that have no data dependency are then considered for concurrent scheduling on different processors.

tightly coupled

Multiprocessors are classified by the way their memory is organized. A multiprocessor system with common shared memory is classified as a *shared-memory* or *tightly coupled multiprocessor*. This does not preclude each processor from having its own local memory. In fact, most commercial tightly coupled multiprocessors provide a cache memory with each CPU. In addition, there is a global common memory that all CPUs can access. Information can therefore be shared among the CPUs by placing it in the common global memory.

loosely coupled

An alternative model of microprocessor is the *distributed-memory* or *loosely coupled* system. Each processor element in a loosely coupled system has its own private local memory. The processors are tied together by a switching scheme designed to route information from one processor to another through a message-passing scheme. The processors relay program and data to other processors in packets. A packet consists of an address, the data content, and some error detection code. The packets are addressed to a specific processor or taken by the first available processor, depending on the communication system used. Loosely coupled systems are most efficient when the interaction between tasks is minimal, whereas tightly coupled systems can tolerate a higher degree of interaction between tasks.

13-2 Interconnection Structures

The components that form a multiprocessor system are CPUs, IOPs connected to input–output devices, and a memory unit that may be partitioned into a number of separate modules. The interconnection between the components can have different physical configurations, depending on the number of transfer paths that are available between the processors and memory in a shared memory system or among the processing elements in a loosely coupled system. There are several physical forms available for establishing an interconnection network. Some of these schemes are presented in this section:

1. Time-shared common bus
2. Multiport memory
3. Crossbar switch
4. Multistage switching network
5. Hypercube system

Time-Shared Common Bus

A common-bus multiprocessor system consists of a number of processors connected through a common path to a memory unit. A time-shared common bus for five processors is shown in Fig. 13-1. Only one processor can communicate with the memory or another processor at any given time. Transfer

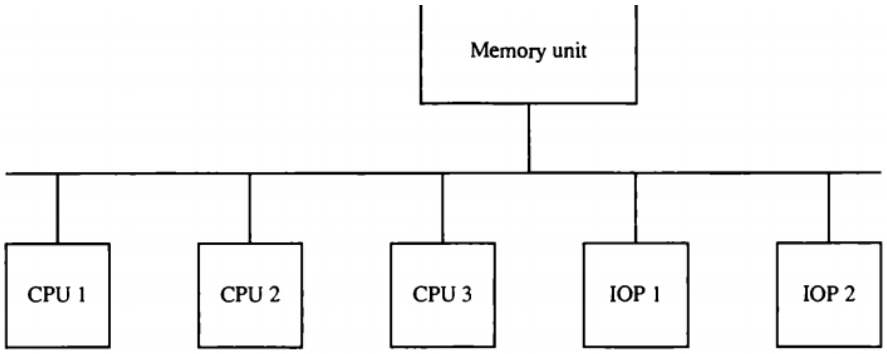


Figure 13-1 Time-shared common bus organization.

operations are conducted by the processor that is in control of the bus at the time. Any other processor wishing to initiate a transfer must first determine the availability status of the bus, and only after the bus becomes available can the processor address the destination unit to initiate the transfer. A command is issued to inform the destination unit what operation is to be performed. The receiving unit recognizes its address in the bus and responds to the control signals from the sender, after which the transfer is initiated. The system may exhibit transfer conflicts since one common bus is shared by all processors. These conflicts must be resolved by incorporating a bus controller that establishes priorities among the requesting units.

A single common-bus system is restricted to one transfer at a time. This means that when one processor is communicating with the memory, all other processors are either busy with internal operations or must be idle waiting for the bus. As a consequence, the total overall transfer rate within the system is limited by the speed of the single path. The processors in the system can be kept busy more often through the implementation of two or more independent buses to permit multiple simultaneous bus transfers. However, this increases the system cost and complexity.

A more economical implementation of a dual bus structure is depicted in Fig. 13-2. Here we have a number of local buses each connected to its own local memory and to one or more processors. Each local bus may be connected to a CPU, an IOP, or any combination of processors. A system bus controller links each local bus to a common system bus. The I/O devices connected to the local IOP, as well as the local memory, are available to the local processor. The memory connected to the common system bus is shared by all processors. If an IOP is connected directly to the system bus, the I/O devices attached to it may be made available to all processors. Only one processor can communicate with the shared memory and other common resources through the system bus at any given time. The other processors are kept busy communicating with their local memory and I/O devices. Part of the local memory may be designed

shared memory

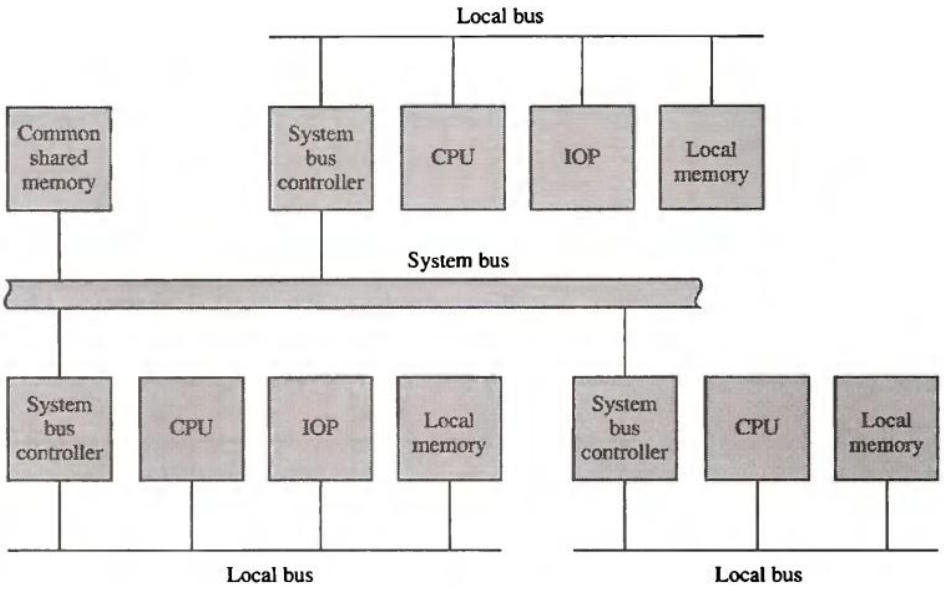


Figure 13-2 System bus structure for multiprocessors.

as a cache memory attached to the CPU (see Sec. 12-6). In this way, the average access time of the local memory can be made to approach the cycle time of the CPU to which it is attached.

Multiport Memory

A multiport memory system employs separate buses between each memory module and each CPU. This is shown in Fig. 13-3 for four CPUs and four memory modules (MMs). Each processor bus is connected to each memory module. A processor bus consists of the address, data, and control lines required to communicate with memory. The memory module is said to have four ports and each port accommodates one of the buses. The module must have internal control logic to determine which port will have access to memory at any given time. Memory access conflicts are resolved by assigning fixed priorities to each memory port. The priority for memory access associated with each processor may be established by the physical port position that its bus occupies in each module. Thus CPU 1 will have priority over CPU 2, CPU 2 will have priority over CPU 3, and CPU 4 will have the lowest priority.

The advantage of the multiport memory organization is the high transfer rate that can be achieved because of the multiple paths between processors and memory. The disadvantage is that it requires expensive memory control logic and a large number of cables and connectors. As a consequence, this intercon-

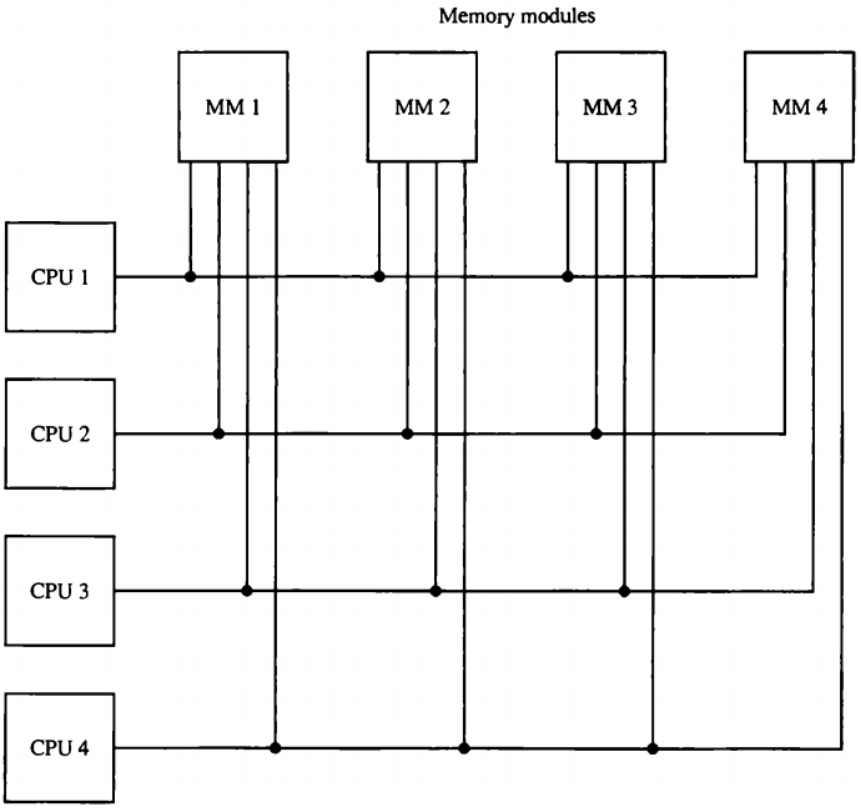


Figure 13-3 Multiport memory organization.

nection structure is usually appropriate for systems with a small number of processors.

Crossbar Switch

The crossbar switch organization consists of a number of crosspoints that are placed at intersections between processor buses and memory module paths. Figure 13-4 shows a crossbar switch interconnection between four CPUs and four memory modules. The small square in each crosspoint is a switch that determines the path from a processor to a memory module. Each switch point has control logic to set up the transfer path between a processor and memory. It examines the address that is placed in the bus to determine whether its particular module is being addressed. It also resolves multiple requests for access to the same memory module on a predetermined priority basis.

Figure 13-5 shows the functional design of a crossbar switch connected to one memory module. The circuit consists of multiplexers that select the data,

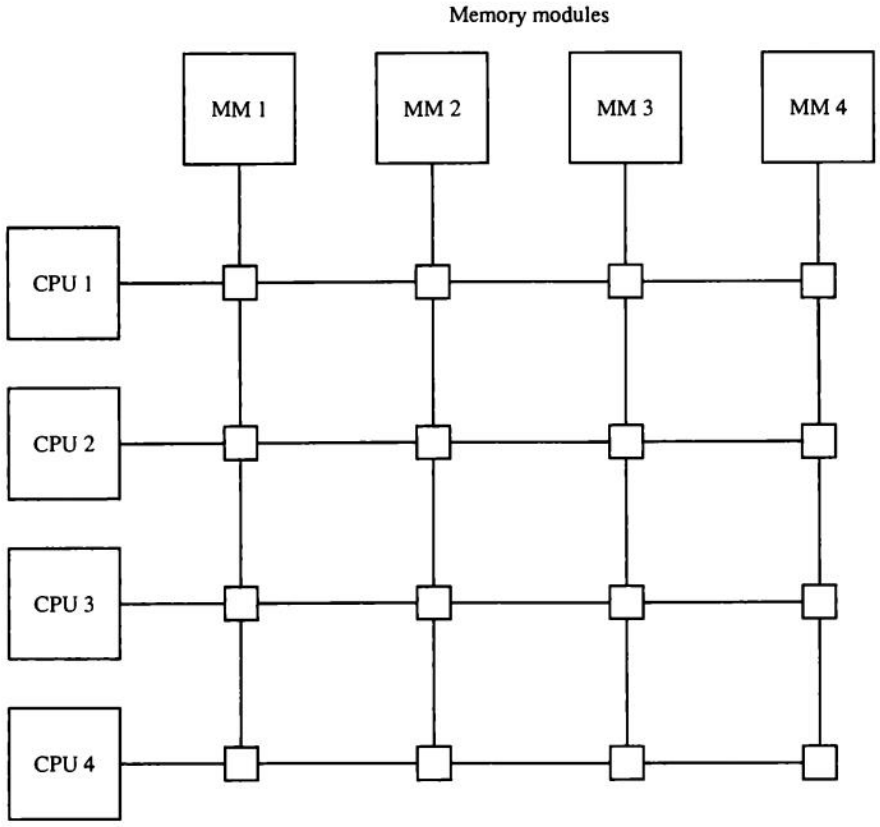
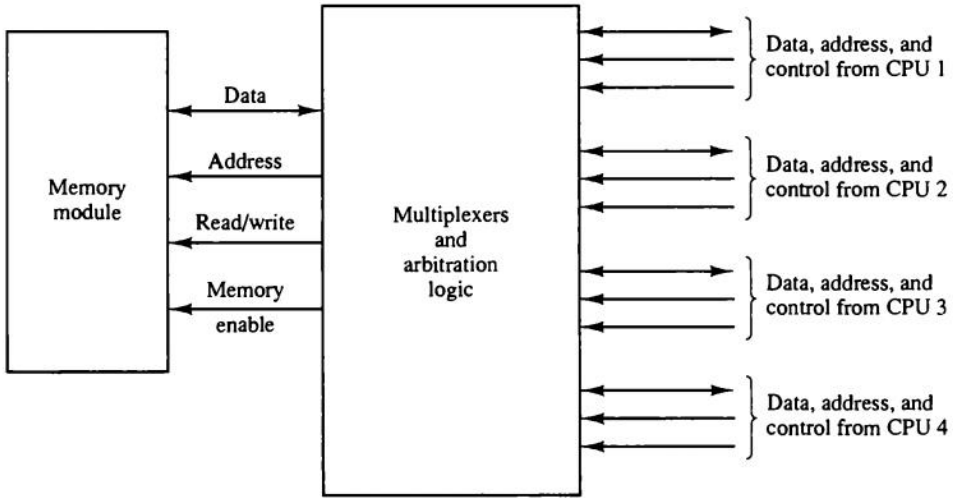


Figure 13-4 Crossbar switch.

Figure 13-5 Block diagram of crossbar switch.



address, and control from one CPU for communication with the memc module. Priority levels are established by the arbitration logic to select one Cl when two or more CPUs attempt to access the same memory. The multiplex are controlled with the binary code that is generated by a priority encoc within the arbitration logic.

A crossbar switch organization supports simultaneous transfers from memory modules because there is a separate path associated with each mc ule. However, the hardware required to implement the switch can becoı quite large and complex.

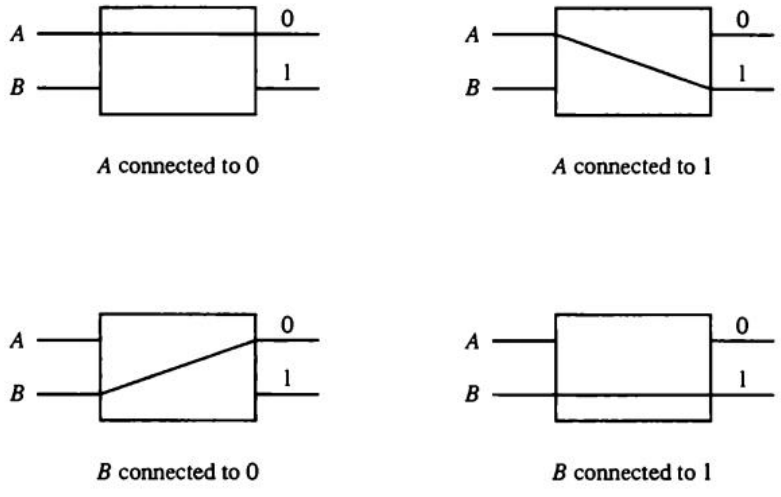
Multistage Switching Network

erchange switch

The basic component of a multistage network is a two-input, two-outp interchange switch. As shown in Fig. 13-6, the 2 × 2 switch has two inpu labeled A and B, and two outputs, labeled 0 and 1. There are control sign (not shown) associated with the switch that establish the interconnecti between the input and output terminals. The switch has the capability connecting input A to either of the outputs. Terminal B of the switch beha in a similar fashion. The switch also has the capability to arbitrate betwe conflicting requests. If inputs A and B both request the same output termin only one of them will be connected; the other will be blocked.

Using the 2 × 2 switch as a building block, it is possible to build multistage network to control the communication between a number of sour and destinations. To see how this is done, consider the binary tree shown Fig. 13-7. The two processors P_1 and P_2 are connected through switches to eig memory modules marked in binary from 000 through 111. The path fron source to a destination is determined from the binary bits of the destinati

Figure 13-6 Operation of a 2 × 2 interchange switch.



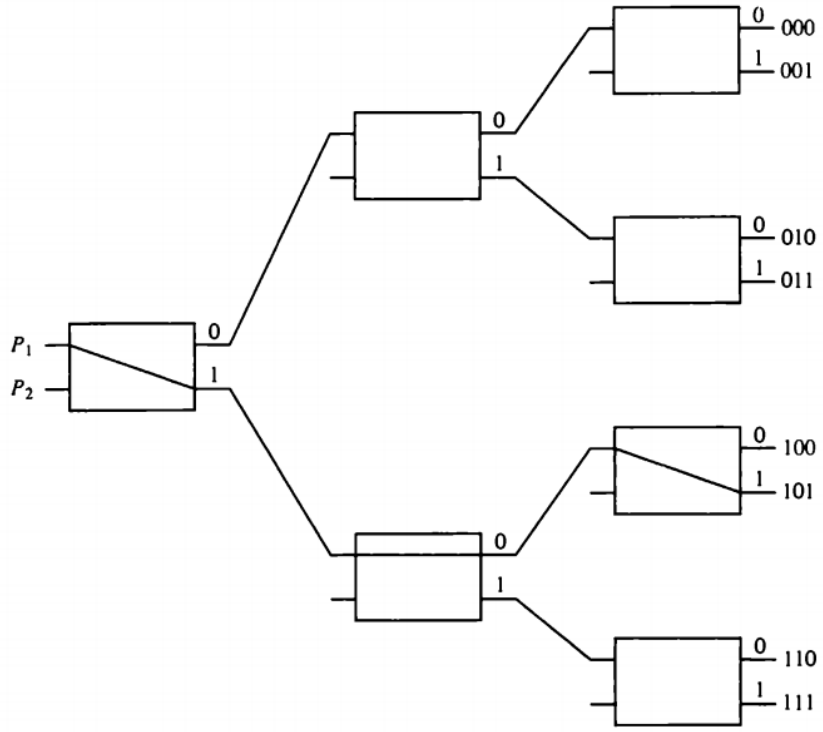


Figure 13-7 Binary tree with 2 × 2 switches.

number. The first bit of the destination number determines the switch output in the first level. The second bit specifies the output of the switch in the second level, and the third bit specifies the output of the switch in the third level. For example, to connect P_1 to memory 101, it is necessary to form a path from P_1 to output 1 in the first-level switch, output 0 in the second-level switch, and output 1 in the third-level switch. It is clear that either P_1 or P_2 can be connected to any one of the eight memories. Certain request patterns, however, cannot be satisfied simultaneously. For example, if P_1 is connected to one of the destinations 000 through 011, P_2 can be connected to only one of the destinations 100 through 111.

omega network

Many different topologies have been proposed for multistage switching networks to control processor–memory communication in a tightly coupled multiprocessor system or to control the communication between the processing elements in a loosely coupled system. One such topology is the omega switching network shown in Fig. 13-8. In this configuration, there is exactly one path from each source to any particular destination. Some request patterns, however, cannot be connected simultaneously. For example, any two sources cannot be connected simultaneously to destinations 000 and 001.

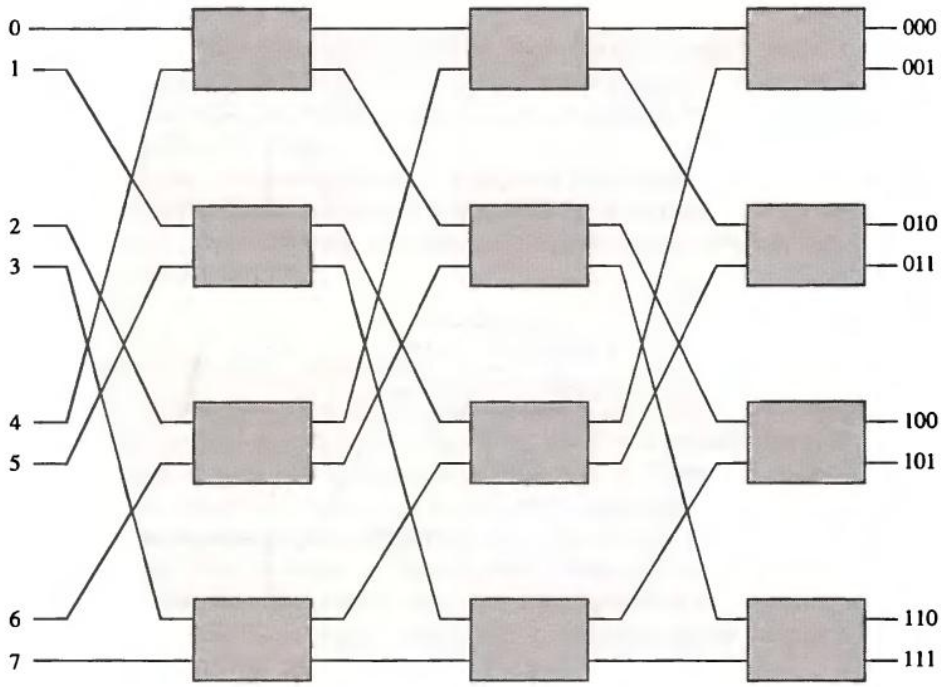


Figure 13-8 8 × 8 omega switching network.

A particular request is initiated in the switching network by the source, which sends a 3-bit pattern representing the destination number. As the binary pattern moves through the network, each level examines a different bit to determine the 2 × 2 switch setting. Level 1 inspects the most significant bit, level 2 inspects the middle bit, and level 3 inspects the least significant bit. When the request arrives on either input of the 2 × 2 switch, it is routed to the upper output if the specified bit is 0 or to the lower output if the bit is 1.

In a tightly coupled multiprocessor system, the source is a processor and the destination is a memory module. The first pass through the network sets up the path. Succeeding passes are used to transfer the address into memory and then transfer the data in either direction, depending on whether the request is a read or a write. In a loosely coupled multiprocessor system, both the source and destination are processing elements. After the path is established, the source processor transfers a message to the destination processor.

Hypercube Interconnection

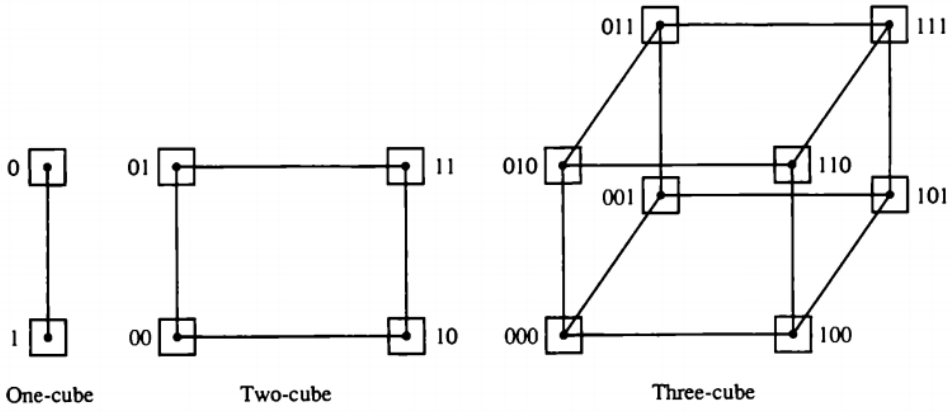
The hypercube or binary n-cube multiprocessor structure is a loosely coupled system composed of $N = 2^n$ processors interconnected in an n-dimensional binary cube. Each processor forms a *node* of the cube. Although it is customary

to refer to each node as having a processor, in effect it contains not only a CPU but also local memory and I/O interface. Each processor has direct communication paths to n other neighbor processors. These paths correspond to the *edges* of the cube. There are 2^n distinct n-bit binary addresses that can be assigned to the processors. Each processor address differs from that of each of its n neighbors by exactly one bit position.

Figure 13-9 shows the hypercube structure for $n = 1, 2$, and 3. A one-cube structure has $n = 1$ and $2^n = 2$. It contains two processors interconnected by a single path. A two-cube structure has $n = 2$ and $2^n = 4$. It contains four nodes interconnected as a square. A three-cube structure has eight nodes interconnected as a cube. An n-cube structure has 2^n nodes with a processor residing in each node. Each node is assigned a binary address in such a way that the addresses of two neighbors differ in exactly one bit position. For example, the three neighbors of the node with address 100 in a three-cube structure are 000, 110, and 101. Each of these binary numbers differs from address 100 by one bit value.

Routing messages through an n-cube structure may take from one to n links from a source node to a destination node. For example, in a three-cube structure, node 000 can communicate directly with node 001. It must cross at least two links to communicate with 011 (from 000 to 001 to 011 or from 000 to 010 to 011). It is necessary to go through at least three links to communicate from node 000 to node 111. A routing procedure can be developed by computing the exclusive-OR of the source node address with the destination node address. The resulting binary value will have 1 bits corresponding to the axes on which the two nodes differ. The message is then sent along any one of the axes. For example, in a three-cube structure, a message at 010 going to 001 produces an exclusive-OR of the two addresses equal to 011. The message can be sent along the second axis to 000 and then through the third axis to 001.

Figure 13-9 Hypercube structures for $n = 1,2,3$.



A representative of the hypercube architecture is the Intel iPSC computer complex. It consists of 128 ($n = 7$) microcomputers connected through communication channels. Each node consists of a CPU, a floating-point processor, local memory, and serial communication interface units. The individual nodes operate independently on data stored in local memory according to resident programs. The data and programs to each node come through a message-passing system from other nodes or from a cube manager. Application programs are developed and compiled on the cube manager and then downloaded to the individual nodes. Computations are distributed through the system and executed concurrently.

13-3 Interprocessor Arbitration

Computer systems contain a number of buses at various levels to facilitate the transfer of information between components. The CPU contains a number of internal buses for transferring information between processor registers and ALU. A memory bus consists of lines for transferring data, address, and read/write information. An I/O bus is used to transfer information to and from input and output devices. A bus that connects major components in a multiprocessor system, such as CPUs, IOPs, and memory, is called a *system bus*. The physical circuits of a system bus are contained in a number of identical printed circuit boards. Each board in the system belongs to a particular module. The board consists of circuits connected in parallel through connectors. Each pin of each circuit connector is connected by a wire to the corresponding pin of all other connectors in other boards. Thus any board can be plugged into a slot in the backplane that forms the system bus.

system bus

The processors in a shared memory multiprocessor system request access to common memory or other common resources through the system bus. If no other processor is currently utilizing the bus, the requesting processor may be granted access immediately. However, the requesting processor must wait if another processor is currently utilizing the system bus. Furthermore, other processors may request the system bus at the same time. Arbitration must then be performed to resolve this multiple contention for the shared resources. The arbitration logic would be part of the system bus controller placed between the local bus and the system bus as shown in Fig. 13-2.

System Bus

A typical system bus consists of approximately 100 signal lines. These lines are divided into three functional groups: data, address, and control. In addition, there are power distribution lines that supply power to the components. For example, the IEEE standard 796 multibus system has 16 data lines, 24 address lines, 26 control lines, and 20 power lines, for a total of 86 lines.

The data lines provide a path for the transfer of data between processors and common memory. The number of data lines is usually a multiple of 8, with 16 and 32 being most common. The address lines are used to identify a memory address or any other source or destination, such as input or output ports. The number of address lines determines the maximum possible memory capacity in the system. For example, an address of 24 lines can access up to 2^{24} (16 mega) words of memory. The data and address lines are terminated with three-state buffers (see Fig. 4-5). The address buffers are unidirectional from processor to memory. The data lines are bidirectional (see Fig. 12-3), allowing the transfer of data in either direction.

synchronous bus

Data transfers over the system bus may be synchronous or asynchronous. In a synchronous bus, each data item is transferred during a time slice known in advance to both source and destination units. Synchronization is achieved by driving both units from a common clock source. An alternative procedure is to have separate clocks of approximately the same frequency in each unit. Synchronization signals are transmitted periodically in order to keep all clocks in the system in step with each other.

asynchronous bus

In an asynchronous bus, each data item being transferred is accompanied by handshaking control signals (see Fig. 11-9) to indicate when the data are transferred from the source and received by the destination.

The control lines provide signals for controlling the information transfer between units. Timing signals indicate the validity of data and address information. Command signals specify operations to be performed. Typical control lines include transfer signals such as memory read and write, acknowledge of a transfer, interrupt requests, bus control signals such as bus request and bus grant, and signals for arbitration procedures.

Table 13-1 lists the 86 lines that are available in the IEEE standard 796 multibus. It includes 16 data lines and 24 address lines. All signals in the multibus are active or enabled in the low-level state. The data transfer control signals include memory read and write as well as I/O read and write. Consequently, the address lines can be used to address separate memory and I/O spaces. The memory or I/O responds with a transfer acknowledge signal when the transfer is completed. Each processor attached to the multibus has up to eight interrupt request outputs and one interrupt acknowledge input line. They are usually applied to a priority interrupt controller similar to the one described in Fig. 11-21. The miscellaneous control signals provide timing and initialization capabilities. In particular, the bus lock signal is essential for multiprocessor applications. This processor-activated signal serves to prevent other processors from getting hold of the bus while executing a test and set instruction. This instruction is needed for proper processor synchronization (see Sec. 13-4).

The six bus arbitration signals are used for interprocessor arbitration. These signals will be explained later after a discussion of the serial and parallel arbitration procedures.

TABLE 13-1 IEEE Standard 796 Multibus Signals

	Signal name
Data and address	
Data lines (16 lines)	DATA0–DATA15
Address lines (24 lines)	ADRS0–ADRS23
Data transfer	
Memory read	MRDC
Memory write	MWTC
IO read	IORC
IO write	IOWC
Transfer acknowledge	TACK
Interrupt control	
Interrupt request (8 lines)	INT0–INT7
Interrupt acknowledge	INTA
Miscellaneous control	
Master clock	CCLK
System initialization	INIT
Byte high enable	BHEN
Memory inhibit (2 lines)	INH1–INH2
Bus lock	LOCK
Bus arbitration	
Bus request	BREQ
Common bus request	CBRQ
Bus busy	BUSY
Bus clock	BCLK
Bus priority in	BPRN
Bus priority out	BPRO
Power and ground (20 lines)	

Reprinted with permission of the IEEE.

Serial Arbitration Procedure

Arbitration procedures service all processor requests on the basis of established priorities. A hardware bus priority resolving technique can be established by means of a serial or parallel connection of the units requesting control of the system bus. The serial priority resolving technique is obtained from a daisy-chain connection of bus arbitration circuits similar to the priority interrupt logic presented in Sec. 11-5. The processors connected to the system bus are assigned priority according to their position along the priority control line. The device closest to the priority line is assigned the highest priority. When multiple devices concurrently request the use of the bus, the device with the highest priority is granted access to it.

Figure 13-10 shows the daisy-chain connection of four arbiters. It is assumed that each processor has its own bus arbiter logic with priority-in and priority-out lines. The priority out (*PO*) of each arbiter is connected to the

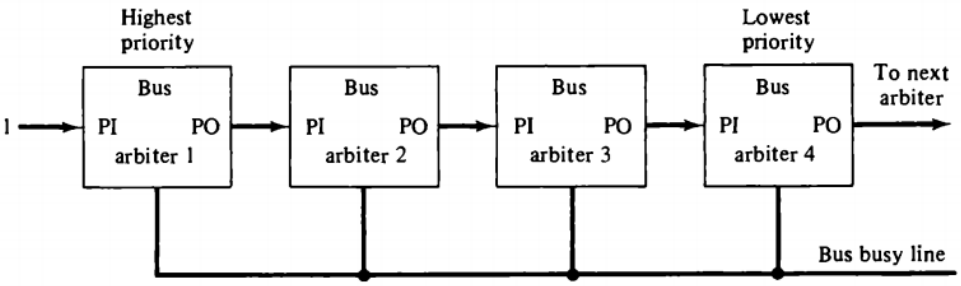


Figure 13-10 Serial (daisy-chain) arbitration.

priority in (*PI*) of the next-lower-priority arbiter. The *PI* of the highest-priority unit is maintained at a logic 1 value. The highest-priority unit in the system will always receive access to the system bus when it requests it. The *PO* output for a particular arbiter is equal to 1 if its *PI* input is equal to 1 and the processor associated with the arbiter logic is not requesting control of the bus. This is the way that priority is passed to the next unit in the chain. If the processor requests control of the bus and the corresponding arbiter finds its *PI* input equal to 1, it sets its *PO* output to 0. Lower-priority arbiters receive a 0 in *PI* and generate a 0 in *PO*. Thus the processor whose arbiter has a $PI = 1$ and $PO = 0$ is the one that is given control of the system bus.

A processor may be in the middle of a bus operation when a higher-priority processor requests the bus. The lower-priority processor must complete its bus operation before it relinquishes control of the bus. The bus busy line shown in Fig. 13-10 provides a mechanism for an orderly transfer of control. The busy line comes from open-collector circuits in each unit and provides a wired-OR logic connection. When an arbiter receives control of the bus (because its $PI = 1$ and $PO = 0$) it examines the busy line. If the line is inactive, it means that no other processor is using the bus. The arbiter activates the busy line and its processor takes control of the bus. However, if the arbiter finds the busy line active, it means that another processor is currently using the bus. The arbiter keeps examining the busy line while the lower-priority processor that lost control of the bus completes its operation. When the bus busy line returns to its inactive state, the higher-priority arbiter enables the busy line, and its corresponding processor can then conduct the required bus transfers.

Parallel Arbitration Logic

The parallel bus arbitration technique uses an external priority encoder and a decoder as shown in Fig. 13-11. Each bus arbiter in the parallel scheme has a bus request output line and a bus acknowledge input line. Each arbiter enables the request line when its processor is requesting access to the system

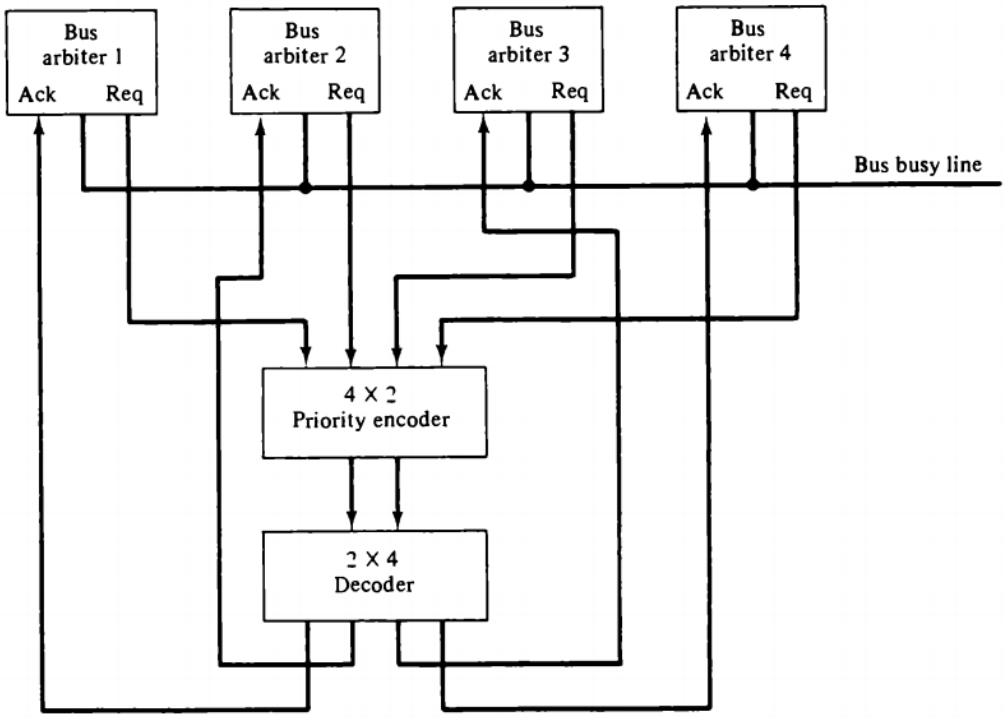


Figure 13-11 Parallel arbitration.

bus. The processor takes control of the bus if its acknowledge input line is enabled. The bus busy line provides an orderly transfer of control, as in the daisy-chaining case.

Figure 13-11 shows the request lines from four arbiters going into a 4 × 2 priority encoder. The output of the encoder generates a 2-bit code which represents the highest-priority unit among those requesting the bus. The truth table of the priority encoder can be found in Table 11-2 (Sec. 11-5). The 2-bit code from the encoder output drives a 2 × 4 decoder which enables the proper acknowledge line to grant bus access to the highest-priority unit.

We can now explain the function of the bus arbitration signals listed in Table 13-1. The bus priority-in BPRN and bus priority-out BPRO are used for a daisy-chain connection of bus arbitration circuits. The bus busy signal BUSY is an open-collector output used to instruct all arbiters when the bus is busy conducting a transfer. The common bus request CBRQ is also an open-collector output that serves to instruct the arbiter if there are any other arbiters of lower-priority requesting use of the system bus. The signals used to construct a parallel arbitration procedure are bus request BREQ and priority-in BPRN,

corresponding to the request and acknowledge signals in Fig. 13-11. The bus clock BCLK is used to synchronize all bus transactions.

Dynamic Arbitration Algorithms

The two bus arbitration procedures just described use a static priority algorithm since the priority of each device is fixed by the way it is connected to the bus. In contrast, a dynamic priority algorithm gives the system the capability for changing the priority of the devices while the system is in operation. We now discuss a few arbitration procedures that use dynamic priority algorithms.

time slice

The *time slice* algorithm allocates a fixed-length time slice of bus time that is offered sequentially to each processor, in round-robin fashion. The service given to each system component with this scheme is independent of its location along the bus. No preference is given to any particular device since each is allotted the same amount of time to communicate with the bus.

polling

In a bus system that uses *polling*, the bus grant signal is replaced by a set of lines called poll lines which are connected to all units. These lines are used by the bus controller to define an address for each device connected to the bus. The bus controller sequences through the addresses in a prescribed manner. When a processor that requires access recognizes its address, it activates the bus busy line and then accesses the bus. After a number of bus cycles, the polling process continues by choosing a different processor. The polling sequence is normally programmable, and as a result, the selection priority can be altered under program control.

LRU

The *least recently used* (LRU) algorithm gives the highest priority to the requesting device that has not used the bus for the longest interval. The priorities are adjusted after a number of bus cycles according to the LRU algorithm. With this procedure, no processor is favored over any other since the priorities are dynamically changed to give every device an opportunity to access the bus.

FIFO

In the *first-come, first-serve* scheme, requests are served in the order received. To implement this algorithm, the bus controller establishes a queue arranged according to the time that the bus requests arrive. Each processor must wait for its turn to use the bus on a first-in, first-out (FIFO) basis.

rotating daisy-chain

The *rotating daisy-chain* procedure is a dynamic extension of the daisy-chain algorithm. In this scheme there is no central bus controller, and the priority line is connected from the priority-out of the last device back to the priority-in of the first device in a closed loop. This is similar to the connections shown in Fig. 13-10 except that the *PO* output of arbiter 4 is connected to the *PI* input of arbiter 1. Whichever device has access to the bus serves as a bus controller for the following arbitration. Each arbiter priority for a given bus cycle is determined by its position along the bus priority line from the arbiter

whose processor is currently controlling the bus. Once an arbiter releases the bus, it has the lowest priority.

13-4 Interprocessor Communication and Synchronization

The various processors in a multiprocessor system must be provided with a facility for communicating with each other. A communication path can be established through common input–output channels. In a shared memory multiprocessor system, the most common procedure is to set aside a portion of memory that is accessible to all processors. The primary use of the common memory is to act as a message center similar to a mailbox, where each processor can leave messages for other processors and pick up messages intended for it.

The sending processor structures a request, a message, or a procedure, and places it in the memory mailbox. Status bits residing in common memory are generally used to indicate the condition of the mailbox, whether it has meaningful information, and for which processor it is intended. The receiving processor can check the mailbox periodically to determine if there are valid messages for it. The response time of this procedure can be time consuming since a processor will recognize a request only when polling messages. A more efficient procedure is for the sending processor to alert the receiving processor directly by means of an interrupt signal. This can be accomplished through a software-initiated interprocessor interrupt by means of an instruction in the program of one processor which when executed produces an external interrupt condition in a second processor. This alerts the interrupted processor of the fact that a new message was inserted by the interrupting processor.

In addition to shared memory, a multiprocessor system may have other shared resources. For example, a magnetic disk storage unit connected to an IOP may be available to all CPUs. This provides a facility for sharing of system programs stored in the disk. A communication path between two CPUs can be established through a link between two IOPs associated with two different CPUs. This type of link allows each CPU to treat the other as an I/O device so that messages can be transferred through the I/O path.

To prevent conflicting use of shared resources by several processors there must be a provision for assigning resources to processors. This task is given to the operating system. There are three organizations that have been used in the design of operating system for multiprocessors: master-slave configuration, separate operating system, and distributed operating system.

In a master-slave mode, one processor, designated the master, always executes the operating system functions. The remaining processors, denoted as slaves, do not perform operating system functions. If a slave processor needs

an operating system service, it must request it by interrupting the master and waiting until the current program can be interrupted.

In the separate operating system organization, each processor can execute the operating system routines it needs. This organization is more suitable for loosely coupled systems where every processor may have its own copy of the entire operating system.

In the distributed operating system organization, the operating system routines are distributed among the available processors. However, each particular operating system function is assigned to only one processor at a time. This type of organization is also referred to as a floating operating system since the routines float from one processor to another and the execution of the routines may be assigned to different processors at different times.

In a loosely coupled multiprocessor system the memory is distributed among the processors and there is no shared memory for passing information. The communication between processors is by means of message passing through I/O channels. The communication is initiated by one processor calling a procedure that resides in the memory of the processor with which it wishes to communicate. When the sending processor and receiving processor name each other as a source and destination, a channel of communication is established. A message is then sent with a header and various data objects used to communicate between nodes. There may be a number of possible paths available to send the message between any two nodes. The operating system in each node contains routing information indicating the alternative paths that can be used to send a message to other nodes. The communication efficiency of the interprocessor network depends on the communication routing protocol, processor speed, data link speed, and the topology of the network.

Interprocessor Synchronization

The instruction set of a multiprocessor contains basic instructions that are used to implement communication and synchronization between cooperating processes. Communication refers to the exchange of data between different processes. For example, parameters passed to a procedure in a different processor constitute interprocessor communication. Synchronization refers to the special case where the data used to communicate between processors is control information. Synchronization is needed to enforce the correct sequence of processes and to ensure mutually exclusive access to shared writable data.

Multiprocessor systems usually include various mechanisms to deal with the synchronization of resources. Low-level primitives are implemented directly by the hardware. These primitives are the basic mechanisms that enforce mutual exclusion for more complex mechanisms implemented in software. A number of hardware mechanisms for mutual exclusion have been developed. One of the most popular methods is through the use of a binary semaphore.

Mutual Exclusion with a Semaphore

A properly functioning multiprocessor system must provide a mechanism that will guarantee orderly access to shared memory and other shared resources. This is necessary to protect data from being changed simultaneously by two or more processors. This mechanism has been termed *mutual exclusion*. Mutual exclusion must be provided in a multiprocessor system to enable one processor to exclude or lock out access to a shared resource by other processors when it is in a *critical section*. A critical section is a program sequence that, once begun, must complete execution before another processor accesses the same shared resource.

critical section

A binary variable called a *semaphore* is often used to indicate whether or not a processor is executing a critical section. A semaphore is a software-controlled flag that is stored in a memory location that all processors can access. When the semaphore is equal to 1, it means that a processor is executing a critical program, so that the shared memory is not available to other processors. When the semaphore is equal to 0, the shared memory is available to any requesting processor. Processors that share the same memory segment agree by convention not to use the memory segment unless the semaphore is equal to 0, indicating that memory is available. They also agree to set the semaphore to 1 when they are executing a critical section and to clear it to 0 when they are finished.

Testing and setting the semaphore is itself a critical operation and must be performed as a single indivisible operation. If it is not, two or more processors may test the semaphore simultaneously and then each set it, allowing them to enter a critical section at the same time. This action would allow simultaneous execution of critical section, which can result in erroneous initialization of control parameters and a loss of essential information.

A semaphore can be initialized by means of a test and set instruction in conjunction with a hardware *lock* mechanism. A hardware lock is a processor-generated signal that serves to prevent other processors from using the system bus as long as the signal is active. The test-and-set instruction tests and sets a semaphore and activates the lock mechanism during the time that the instruction is being executed. This prevents other processors from changing the semaphore between the time that the processor is testing it and the time that it is setting it. Assume that the semaphore is a bit in the least significant position of a memory word whose address is symbolized by SEM. Let the mnemonic TSL designate the "test and set while locked" operation. The instruction

hardware lock

```
TSL   SEM
```

will be executed in two memory cycles (the first to read and the second to write) without interference as follows:

$R \leftarrow M[SEM]$ Test semaphore
$M[SEM] \leftarrow 1$ Set semaphore

The semaphore is tested by transferring its value to a processor register R and then it is set to 1. The value in R determines what to do next. If the processor finds that $R = 1$, it knows that the semaphore was originally set. (The fact that it is set again does not change the semaphore value.) That means that another processor is executing a critical section, so the processor that checked the semaphore does not access the shared memory. If $R = 0$, it means that the common memory (or the shared resource that the semaphore represents) is available. The semaphore is set to 1 to prevent other processors from accessing memory. The processor can now execute the critical section. The last instruction in the program must clear location SEM to zero to release the shared resource to other processors.

Note that the lock signal must be active during the execution of the test-and-set instruction. It does not have to be active once the semaphore is set. Thus the lock mechanism prevents other processors from accessing memory while the semaphore is being set. The semaphore itself, when set, prevents other processors from accessing shared memory while one processor is executing a critical section.

13-5 Cache Coherence

The operation of cache memory is explained in Sec. 12-6. The primary advantage of cache is its ability to reduce the average access time in uniprocessors. When the processor finds a word in cache during a read operation, the main memory is not involved in the transfer. If the operation is to write, there are two commonly used procedures to update memory. In the *write-through* policy, both cache and main memory are updated with every write operation. In the *write-back* policy, only the cache is updated and the location is marked so that it can be copied later into main memory.

In a shared memory multiprocessor system, all the processors share a common memory. In addition, each processor may have a local memory, part or all of which may be a cache. The compelling reason for having separate caches for each processor is to reduce the average access time in each processor. The same information may reside in a number of copies in some caches and main memory. To ensure the ability of the system to execute memory operations correctly, the multiple copies must be kept identical. This requirement imposes a *cache coherence* problem. A memory scheme is *coherent* if the value returned on a load instruction is always the value given by the latest store instruction with the same address. Without a proper solution to the cache coherence problem, caching cannot be used in bus-oriented multiprocessors with two or more processors.

Conditions for Incoherence

Cache coherence problems exist in multiprocessors with private caches because of the need to share writable data. Read-only data can safely be replicated

without cache coherence enforcement mechanisms. To illustrate the problem, consider the three-processor configuration with private caches shown in Fig. 13-12. Sometime during the operation an element X from main memory is loaded into the three processors, P_1, P_2, and P_3. As a consequence, it is also copied into the private caches of the three processors. For simplicity, we assume that X contains the value of 52. The load on X to the three processors results in consistent copies in the caches and main memory.

If one of the processors performs a store to X, the copies of X in the caches become inconsistent. A load by the other processors will not return the latest value. Depending on the memory update policy used in the cache, the main memory may also be inconsistent with respect to the cache. This is shown in Fig. 13-13. A store to X (of the value of 120) into the cache of processor P_1 updates memory to the new value in a write-through policy. A write-through policy maintains consistency between memory and the originating cache, but the other two caches are inconsistent since they still hold the old value. In a write-back policy, main memory is not updated at the time of the store. The copies in the other two caches and main memory are inconsistent. Memory is updated eventually when the modified data in the cache are copied back into memory.

Another configuration that may cause consistency problems is a direct memory access (DMA) activity in conjunction with an IOP connected to the system bus. In the case of input, the DMA may modify locations in main memory that also reside in cache without updating the cache. During a DMA output, memory locations may be read before they are updated from the cache when using a write-back policy. I/O-based memory incoherence can be overcome by making the IOP a participant in the cache coherent solution that is adopted in the system.

Solutions to the Cache Coherence Problem

Various schemes have been proposed to solve the cache coherence problem in shared memory multiprocessors. We discuss some of these schemes briefly here. See references 3 and 10 for more detailed discussions.

A simple scheme is to disallow private caches for each processor and have a shared cache memory associated with main memory. Every data access is made to the shared cache. This method violates the principle of closeness of CPU to cache and increases the average memory access time. In effect, this scheme solves the problem by avoiding it.

For performance considerations it is desirable to attach a private cache to each processor. One scheme that has been used allows only nonshared and read-only data to be stored in caches. Such items are called *cachable. Shared writable data are noncachable.* The compiler must tag data as either cachable or noncachable, and the system hardware makes sure that only cachable data are stored in caches. The noncachable data remain in main memory. This method

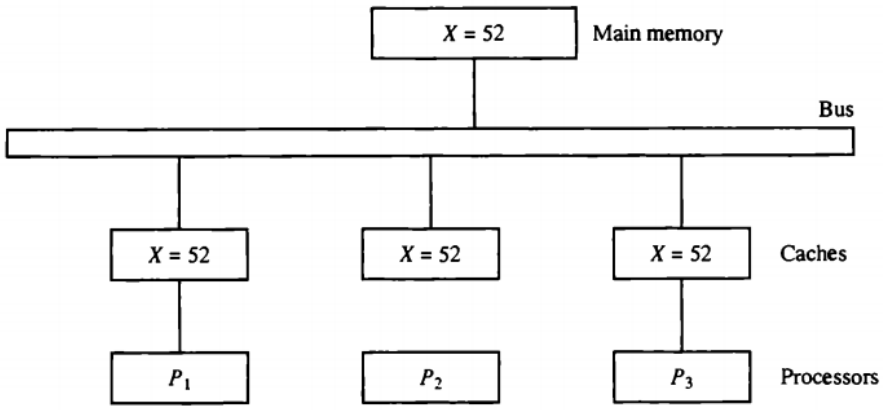
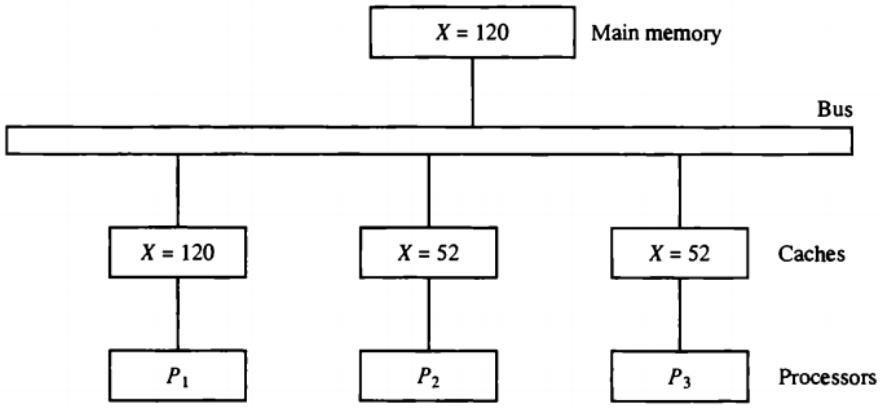
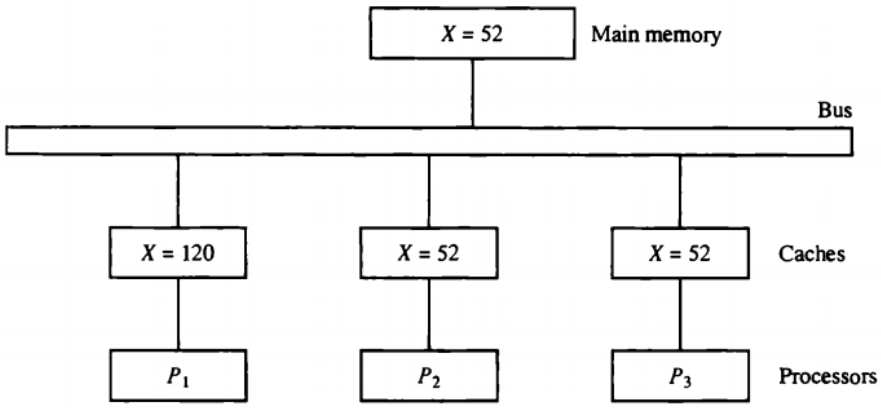


Figure 13-12 Cache configuration after a load on X.

Figure 13-13 Cache configuration after a store to X by processor P_1.



(a) With write-through cache policy



(b) With write-back cache policy

restricts the type of data stored in caches and introduces an extra software overhead that may degradate performance.

A scheme that allows writable data to exist in at least one cache is a method that employs a *centralized global table* in its compiler. The status of memory blocks is stored in the central global table. Each block is identified as *read-only* (RO) or *read and write* (RW). All caches can have copies of blocks identified as RO. Only one cache can have a copy of an RW block. Thus if the data are updated in the cache with an RW block, the other caches are not affected because they do not have a copy of this block.

The cache coherence problem can be solved by means of a combination of software and hardware or by means of hardware-only schemes. The two methods mentioned previously use software-based procedures that require the ability to tag information in order to disable caching of shared writable data. Hardware-only solutions are handled by the hardware automatically and have the advantage of higher speed and program transparency. In the hardware solution, the cache controller is specially designed to allow it to monitor all bus requests from CPUs and IOPs. All caches attached to the bus constantly monitor the network for possible write operations. Depending on the method used, they must then either update or invalidate their own cache copies when a match is detected. The bus controller that monitors this action is referred to as a *snoopy cache controller*. This is basically a hardware unit designed to maintain a bus-watching mechanism over all the caches attached to the bus.

snoopy cache controller

Various schemes have been proposed to solve the cache coherence problem by means of snoopy cache protocol. The simplest method is to adopt a write-through policy and use the following procedure. All the snoopy controllers watch the bus for memory store operations. When a word in a cache is updated by writing into it, the corresponding location in main memory is also updated. The local snoopy controllers in all other caches check their memory to determine if they have a copy of the word that has been overwritten. If a copy exists in a remote cache, that location is marked invalid. Because all caches snoop on all bus writes, whenever a word is written, the net effect is to update it in the original cache and main memory and remove it from all other caches. If at some future time a processor accesses the invalid item from its cache, the response is equivalent to a cache miss, and the updated item is transferred from main memory. In this way, inconsistent versions are prevented.