Chapter 9: Introduction to Parallel Processing

Traditionally, the computer has been viewed as a sequential machine. Most computer programming languages require the programmer to specify algorithms as sequences of instructions. Processors execute programs by executing machine instructions in a sequence and one at a time. Each instruction is executed in a sequence of operations (fetch instruction, fetch operands, perform operation, store results). This view of the computer has never been entirely true. At the micro-operation level, multiple control signals are generated at the same time. Instruction pipelining, at least to the extent of overlapping fetch and execute operations, has been around for a long time. Both of these are examples of performing functions in parallel. This approach is taken further with superscalar organization, which exploits instruction-level parallelism. With a superscalar machine, there are multiple execution units within a single processor, and these may execute multiple instructions from the same program in parallel. Some of the prominent approaches to parallel organization are:

Symmetric multiprocessors (SMPs): SMP organization consists of multiple processors share a common memory, interconnected by a bus or some sort of switching arrangement. This organization raises the issue of cache coherence.

Clusters: it is a group of interconnected independent computers working together as a whole working as a unified computing resource, that create the illusion of being one machine. The term whole computer means a system that can run on its own part from cluster.

Non-uniform memory access (NUMA): It is a shared memory multiprocessor in which the access time from a given processor to a word in memory varies with the location on the memory word. The NUMA approach is relatively new.

Parallel processing

Parallel processing is a term used to denote a large class of techniques that provide simultaneous data processing tasks for the sole purpose of the increasing the computational speed of computer system. A parallel processing system is able to perform concurrent data processing to achieve faster execution time. For example when an instruction is being is executed in ALU the next instruction can be read from memory. The system can also have two or more CPUs so that multiple processing can be done during a given time interval. This technique used more hardware and thus cost of the system increases. Parallel processing can be done in many ways for example using parallel set of registers, shifters, these are relatively less complex and those having multiple operations.

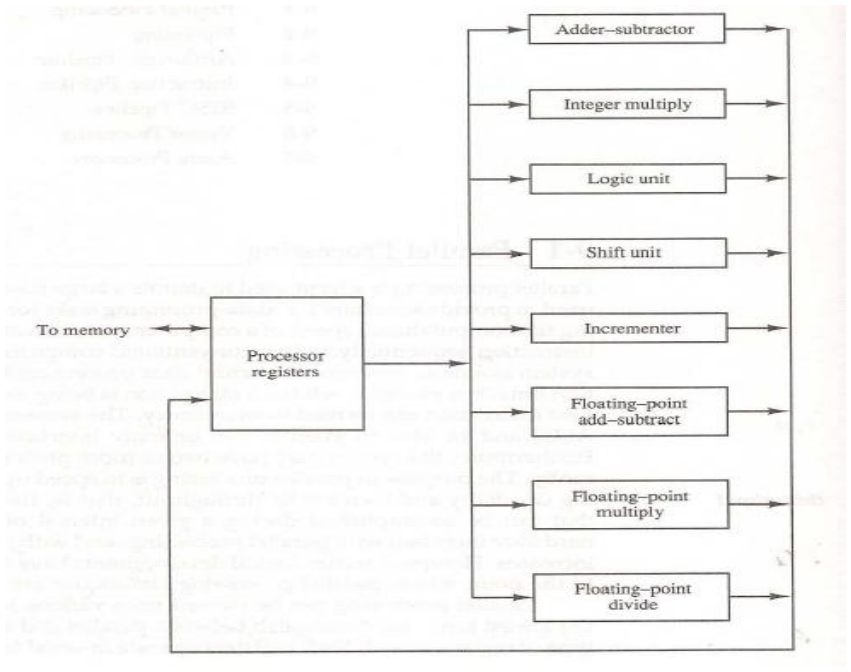


Figure: processor with multiple functional units

- Figure shows one possible way of separating the execution unit into eight functional units.
- All units are independent of each other so that one number can be shifted while other number can be incremented or multiplied.

A multi functional organization is usually associated with a complex control unit to co-ordinate all activities among the various components.

MULTIPLE PROCESSOR ORGANIZATIONS

Types of Parallel Processor Systems

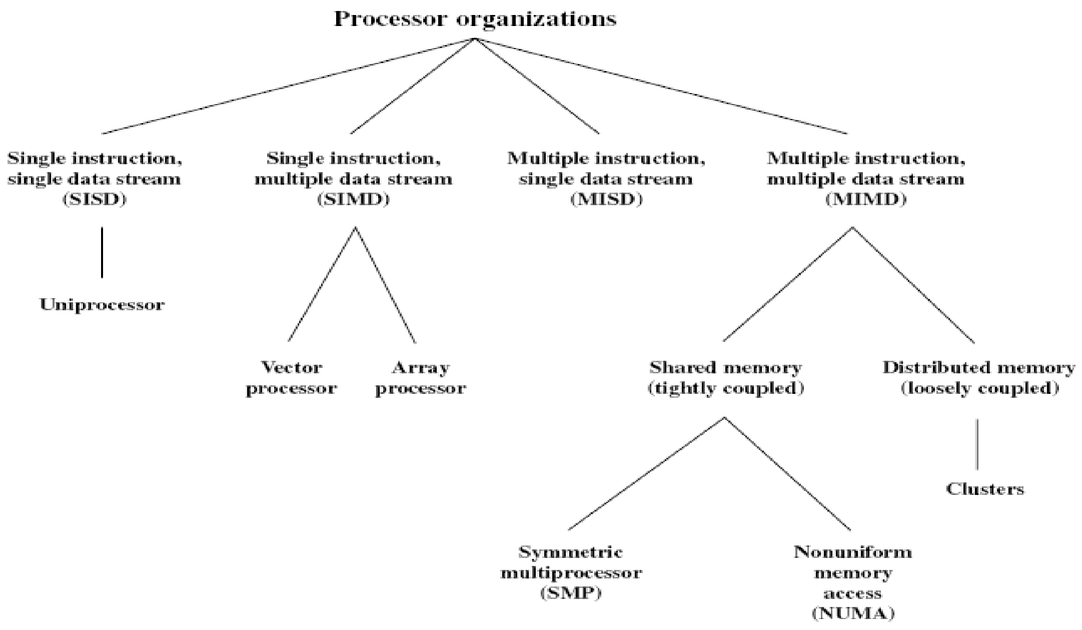
A taxonomy first introduced by Flynn is still the most common way of categorizing systems with parallel processing capability. Flynn proposed the following categories of computer systems:

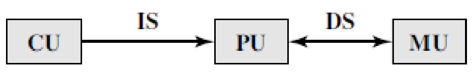
• **Single instruction, single data (SISD) stream:** A single processor executes a single instruction stream to operate on data stored in a single memory. Uni-processors fall into this category.

• **Single instruction, multiple data (SIMD) stream:** A single machine instruction controls the simultaneous execution of a number of processing elements. Each processing element has an associated data memory, so that each instruction is executed on a different set of data by the different processors. Vector and array processors fall into this category.

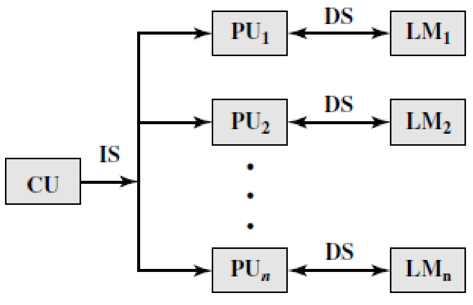
• **Multiple instruction, single data (MISD) stream:** A sequence of data is transmitted to a set of processors, each of which executes a different instruction sequence. This structure is not commercially implemented.

• **Multiple instruction, multiple data (MIMD) stream:** A set of processors simultaneously execute different instruction sequences on different data sets. SMPs, clusters, and NUMA systems fit into this category. With the MIMD organization, the processors are general purpose; each is able to process all of the instructions necessary to perform the appropriate data transformation. MIMDs can be further subdivided by the means in which the processors communicate. If the processors share a common memory, then each processor accesses programs and data stored in the shared memory, and processors communicate with each other via that memory. The most common form of such system is known as a **symmetric multiprocessor (SMP)**. In an SMP, multiple processors share a single memory or pool of memory by means of a shared bus or other interconnection mechanism; a distinguishing feature is that the memory access time to any region of memory is approximately the same for each processor. A more recent development is the **non-uniform memory access (NUMA)** organization.

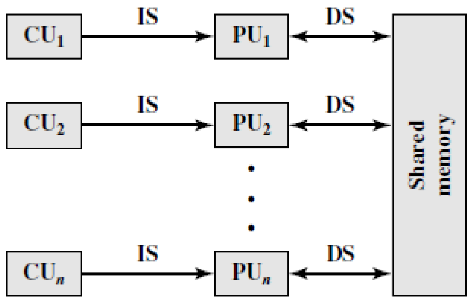

Processor organizations
Single instruction, single data stream (SISD)
Single instruction, multiple data stream (SIMD)
Multiple instruction, single data stream (MISD)
Multiple instruction, multiple data stream (MIMD)
Uniprocessor
Vector processor
Array processor
Shared memory (tightly coupled)
Distributed memory (loosely coupled)
Clusters
Symmetric multiprocessor (SMP)
Nonuniform memory access (NUMA)
Figure 17.1 A Taxonomy of Parallel Processor Architectures



(a) SISD

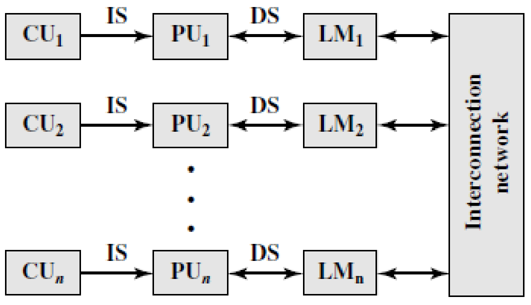


(b) SIMD (with distributed memory)



(c) MIMD (with shared memory)

CU	= Control unit	SISD	= Single instruction, single data stream
IS	= Instruction stream		
PU	= Processing unit	SIMD	= Single instruction, multiple data stream
DS	= Data stream		
MU	= Memory unit	MIMD	= Multiple instruction, multiple data stream
LM	= Local memory		



(d) MIMD (with distributed memory)

SYMMETRIC MULTIPROCESSORS

All single-user personal computers and most workstations contained a single general-purpose microprocessor. As demands for performance increase and as the cost of microprocessors continues to drop, vendors have introduced systems with an SMP organization. The term *SMP* refers to a computer hardware architecture and also to the operating system behavior that reflects that architecture. An SMP can be defined as a standalone computer system with the following characteristics:

1. There are two or more similar processors of comparable capability.

2. These processors share the same main memory and I/O facilities and are interconnected by a bus or other internal connection scheme, such that memory access time is approximately the same for each processor.

3. All processors share access to I/O devices, either through the same channels or through different channels that provide paths to the same device.

4. All processors can perform the same functions (hence the term *symmetric*).

5. The system is controlled by an integrated operating system that provides interaction between processors and their programs at the job, task, file, and data element levels.

The operating system of an SMP schedules processes or threads across all of the processor s. An SMP organization has a number of potential advantages over a uniprocessor organization, including the following:

• **Performance:** If the work to be done by a computer can be organized so that some portions of the work can be done in parallel, then a system with multiple processors will yield greater performance than one with a single processor of the same type.

Availability: In a symmetric multiprocessor, because all processors can perform the same functions, the failure of a single processor does not halt the machine. Instead, the system can continue to function at reduced performance.

• **Incremental growth:** A user can enhance the performance of a system by adding an additional processor.

• **Scaling:** Vendors can offer a range of products with different price and performance characteristics based on the number of processors configured in the system.

Organization

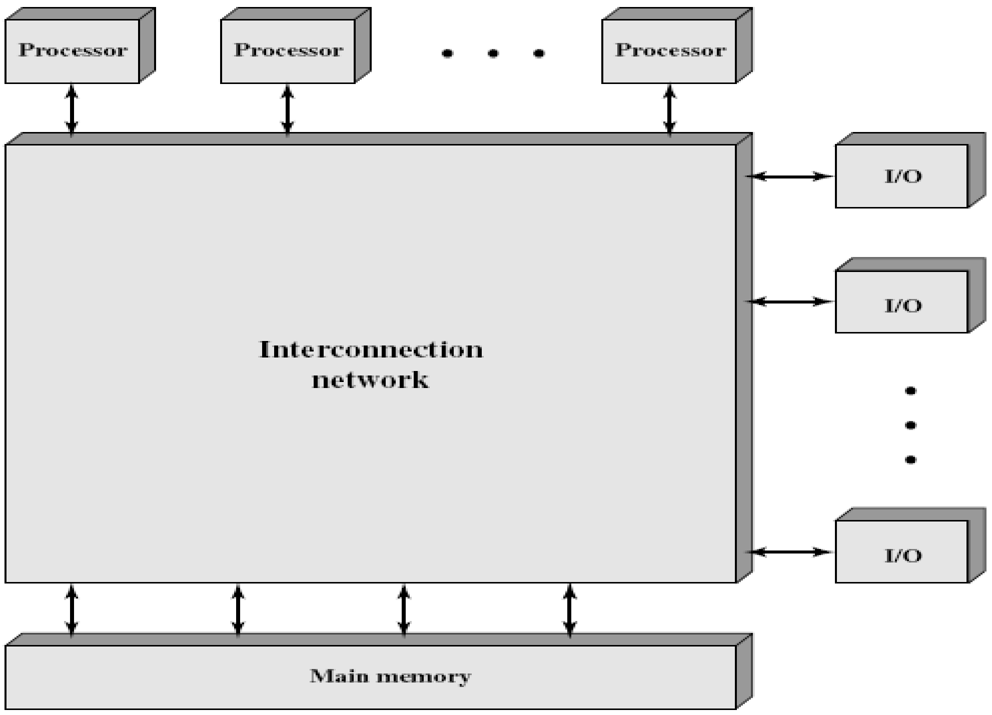


Figure 17.4 Generic Block Diagram of a Tightly Coupled Multiprocessor

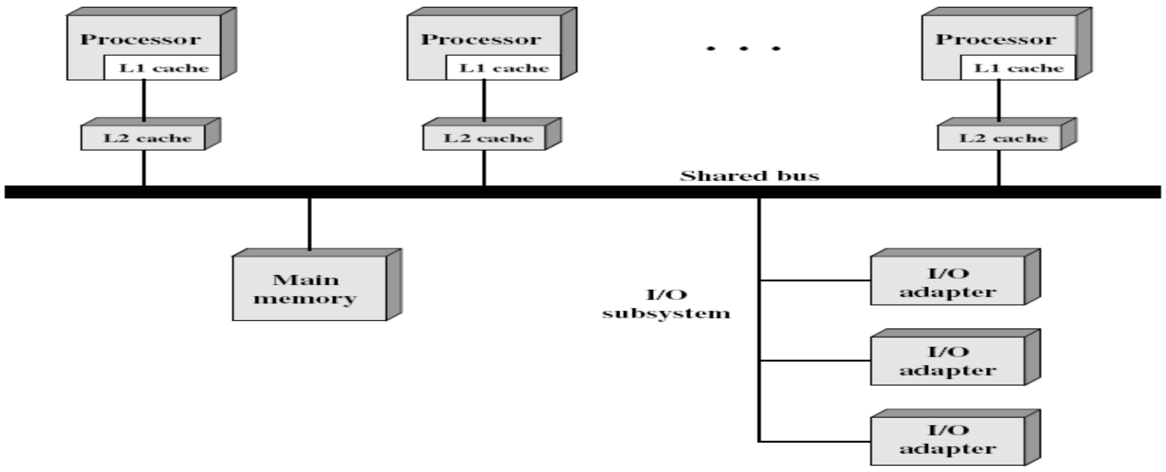


Figure 17.5 Symmetric Multiprocessor Organization

Figure 17.4 depicts in general terms the organization of a multiprocessor system. There are two or more processors. Each processor is self-contained, including a control unit, ALU, registers, and, typically, one or more levels of cache. Each processor has access to a shared main memory and the I/O devices through some form of interconnection mechanism. The processors can communicate with each other through memory (messages and status information left in common data areas). It may also be possible for processors to exchange signals directly. The memory is often organized so that multiple simultaneous accesses to separate blocks of memory are possible. In some configurations, each processor may also have its own private main memory and I/O channels in addition to the shared resources. The most common organization for personal computers, workstations, and servers is the time-shared bus. The time-shared bus is the simplest mechanism for constructing a multiprocessor system (Figure 17.5).The structure and interfaces are basically the same as for a single-processor system that uses a bus interconnection. The bus consists of control, address, and data lines. To facilitate DMA transfers from I/O processors, the following features are provided:

Addressing: It must be possible to distinguish modules on the bus to determine the source and destination of data.

• **Arbitration:** Any I/O module can temporarily function as "master." A mechanism is provided to arbitrate competing requests for bus control, using some sort of priority scheme.

• **Time-sharing:** When one module is controlling the bus, other modules are locked out and must, if necessary, suspend operation until bus access is achieved. These uniprocessor features are directly usable in an SMP organization. In this latter case, there are now multiple processors as well as multiple I/O processors all attempting to gain access to one or more memory modules via the bus.

The bus organization has several attractive features:

• **Simplicity:** This is the simplest approach to multiprocessor organization. The physical interface and the addressing, arbitration, and time-sharing logic of each processor remain the same as in a single-processor system.

• **Flexibility:** It is generally easy to expand the system by attaching more processors to the bus.

• **Reliability:** The bus is essentially a passive medium, and the failure of any attached device should not cause failure of the whole system.

The main drawback to the bus organization is performance. All memory references pass through the common bus. Thus, the bus cycle time limits the speed of the system. To improve performance, it is desirable to equip each processor with a cache memory. This should reduce the number of bus accesses dramatically.

Cache coherence

In multi processor systems it is customary to have one or two levels of cache associated with each processor. This organization is essential to achieve reasonable performance .it does however create a problem called cache coherence problem i.e. if same data is being is used by more than two processor the data will be in their cache. Now if changing the data in one cache is done but not in other or main memory, then same variable will hold different data in different process or multiple copies of same data can exist simultaneously and if processor are allowed to update their own copies freely and inconsistent view of memory can result . We have two different write polices in cache:

1. Write back: write operations are usually made only to the cache. Main memory is only updated when the corresponding cache line is flushed from the cache.
2. Write through: all the operations are made to main memory as well as to the cache, ensuring that main memory is always valid.

It is clear that a write back policy can result in inconsistency. If two caches contain the same line and the line is updated in one cache, the other cache will unknowingly have an invalid data. Subsequent read to that invalid line produce invalid result. Even with write through policy, inconsistency can occur unless other caches monitor the memory traffic or receive some direct notification of the update. To solve cache coherence problem we discuss two approaches software and hardware.

Software solutions:

Software cache coherence schemes attempt to avoid the need for additional hardware circuitry and logic by relying on the compiler and operating system to deal with the problem. Software approaches are attractive because the overhead of detecting potential problems is transferred from run time to compile time, and the design complexity is transferred from hardware to software. On the other hand, compile-time software approaches generally must make conservative decisions, leading to inefficient cache utilization. Compiler-based coherence mechanisms perform an analysis on the code to determine which data items may become unsafe for caching, and they mark those items accordingly. The operating system or hardware then prevents non-cacheable items from being cached. The simplest approach is to prevent any shared data variables from being cached. This is too conservative, because a shared data structure may be exclusively used during some periods and may be effectively read-only during other periods. It is only during periods when at least one process may update the variable and at least one other process may access the variable that cache coherence is an issue. More efficient approaches analyze the code to determine safe periods for shared variables. The compiler then inserts instructions into the generated code to enforce cache coherence during the critical periods.

Hardware Solutions

Hardware-based solutions are generally referred to as cache coherence protocols. These solutions provide dynamic recognition at run time of potential inconsistency conditions. Because the problem is only dealt with when it actually arises, there is more effective use of caches, leading to improved performance over a software approach. In addition, these approaches are transparent to the programmer and the compiler, reducing the software development burden. Hardware schemes differ in a number of particulars, including where the state information about data lines is held, how that information is organized, where coherence is enforced, and the enforcement mechanisms. In general, hardware schemes can be divided into two categories: directory protocols and snoopy protocols.

DIRECTORY PROTOCOLS Directory protocols collect and maintain information about where copies of lines reside. Typically, there is a centralized controller that is part of the main memory controller, and a directory that is stored in main memory. The directory contains global state information about the contents of the various local caches. When an individual cache controller makes a request, the centralized controller checks and issues necessary commands for data transfer between memory and caches or between caches. It is also responsible for keeping the state information up to date; therefore, every local action that can affect the global state of a line must be reported to the central controller. Typically, the controller maintains information about which processors have a copy of which lines. Before a processor can write to a local copy of a line, it must request exclusive access to the line from the controller. Before granting this exclusive access, the controller sends a message to all processors with a cached copy of this line, forcing each processor to invalidate its copy. After receiving acknowledgments back from each such processor, the controller grants exclusive access to the requesting processor. When another processor tries to read a line that is exclusively granted to another processor, it will send a miss notification to the controller. The controller then issues a command to the processor holding that line that requires the processor to do a write back to main memory. The line may now be shared for reading by the original processor and the requesting processor. Directory schemes suffer from the drawbacks of a central bottleneck and the overhead of communication between the various cache controllers and the central controller. However, they are effective in large-scale systems that involve multiple buses or some other complex interconnection scheme.

SNOOPY PROTOCOLS

Snoopy protocols distribute the responsibility for maintaining cache coherence among all of the cache controllers in a multiprocessor. A cache must recognize when a line that it holds is shared with other caches. When an update action is performed on a shared cache line, it must be announced to all other caches by a broadcast mechanism. Each cache controller is able to “snoop” on the network to observe these broadcasted notifications, and react accordingly. Snoopy protocols are ideally suited to a bus-based multiprocessor, because the shared bus provides a simple means for broadcasting and snooping. However, because one of the objectives of the use of local caches is to avoid bus accesses, care must be taken that the increased bus traffic required for broadcasting and snooping does not cancel out the gains from the use of local caches. Two basic approaches to the snoopy protocol have been explored: write invalidate and write update (or write broadcast).With a write-invalidate protocol, there can be multiple readers but only one writer at a time. Initially, a line may be shared among several caches for reading purposes. When one of the caches wants to perform a write to the line, it first issues a notice that invalidates that line in the other caches, making the line exclusive to the writing cache. Once the line is exclusive, the owning processor can make cheap local writes until some other processor requires the same line. With a write-update protocol, there can be multiple writers as well as multiple readers. When a processor wishes to update a shared line, the word to be updated is distributed to all others, and caches containing that line can update it. Neither of these two approaches is superior to the other under all circumstances. Performance depends on the number of local caches and the pattern of memory reads and writes. Some systems implement adaptive protocols that employ both write-invalidate and write-update mechanisms.

The MESI Protocol

To provide cache consistency on an SMP, the data cache often supports a protocol known as MESI. For MESI, the data cache includes two status bits per tag, so that each line can be in one of four states:

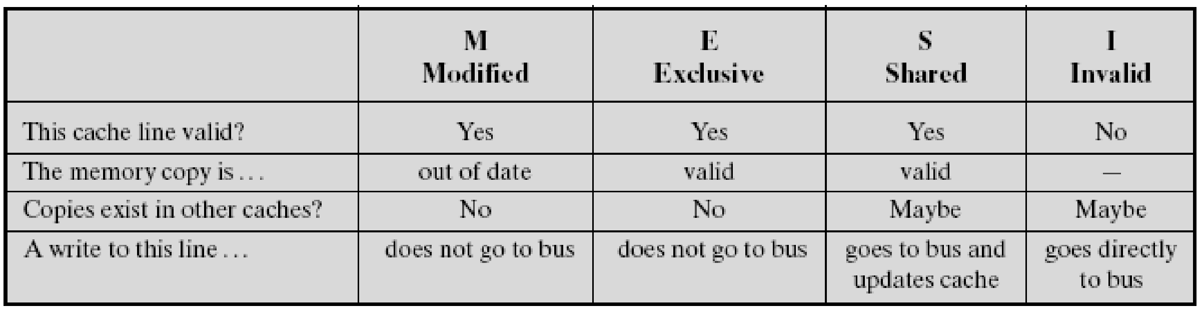
• **Modified:** The line in the cache has been modified (different from main memory) and is available only in this cache.

• **Exclusive:** The line in the cache is the same as that in main memory and is not present in any other cache.

• **Shared:** The line in the cache is the same as that in main memory and may be present in another cache.

• **Invalid:** The line in the cache does not contain valid data.

Table 17.1 MESI Cache Line States



	M Modified	E Exclusive	S Shared	I Invalid
This cache line valid?	Yes	Yes	Yes	No
The memory copy is . . .	out of date	valid	valid	—
Copies exist in other caches?	No	No	Maybe	Maybe
A write to this line . . .	does not go to bus	does not go to bus	goes to bus and updates cache	goes directly to bus

Vector processing

There is a class of computational problems that are beyond the capabilities of a conventional computer, like the programs that require vector or matrix calculation, where large amount of calculation has to be done. Computers with vector processing capabilities are in demand in specialized applications. Some of the areas of application are long range weather forecasting, petroleum exploration, seismic data analysis, medical diagnosis, aerodynamics and space flight simulator, artificial intelligence and expert systems, image processing.

Vector operations

Many engineering and scientific problems require arithmetic operations on large array of numbers; these are usually formulated as vector and matrix of floating point numbers. A vector is an ordered set of one dimensional array of data items i.e. as V= [V1, V2, V3…….Vn].A conventional sequence computer is capable of processing operands one at time. Consequently operations on vector must be broken into single computations with subscribed variables. For calculation using convectional processor, it is performed as:

```
For (i=1; i<=100; i))

{C (i) =A (i) +B (i);

}
```

This constitutes a program loop that read a pair of operands from array A and B and perform a floating point addition. The loop control variable is then updated and steps repeated 100 times.

A computer capable of vector processing eliminates the overhead associated with the time it takes to fetch and execute the instruction in the program loops. It allows operation to be specified with a single vector instruction of the form C (1:100) =A (1:100) +B (1:100)

The vector instruction includes the initial address of the operands, the length of the vectors and the operations to be performed all in one composite instruction. The addition is done with a pipelined floating point adder similar to the arithmetic pipeline. A possible instruction format for a vector instruction is

Op-code	Base address source 1	Base address source 2	Base address destination	Vector length

This is essentially a three address instruction with three fields specifying the base address of the operand and an additional field that gives the length of the data items in the vector.

See matrix multiplication from book example.

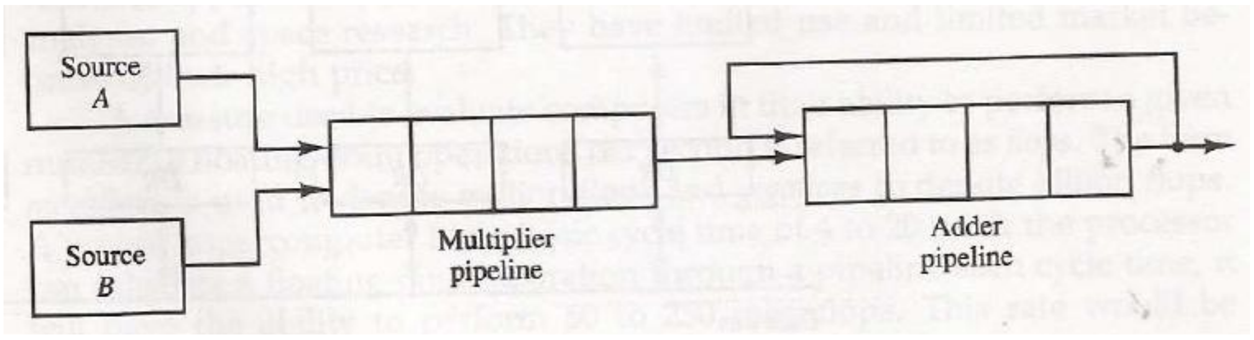


Figure: pipeline for calculating an inner product

Memory interleaving

Pipeline and vector processing often require simultaneous access to memory from two or more sources or fetching of instruction and an operand at the same time, so instead of using two memory buses the memory can be partitioned into a number of modules in a memory array together with its own address and data registers. Figure shows the four modules, the address registers receive information from a common address bus and data register communicated with bi-directional data bus. The two least significant bits of the address can be used to distinguish between four modules. The modular system permits one module to initiate a memory access while other modules are in process of reading or writing a word and each module can accept memory request independent of the state of other modules. This technique is called memory interleaving.

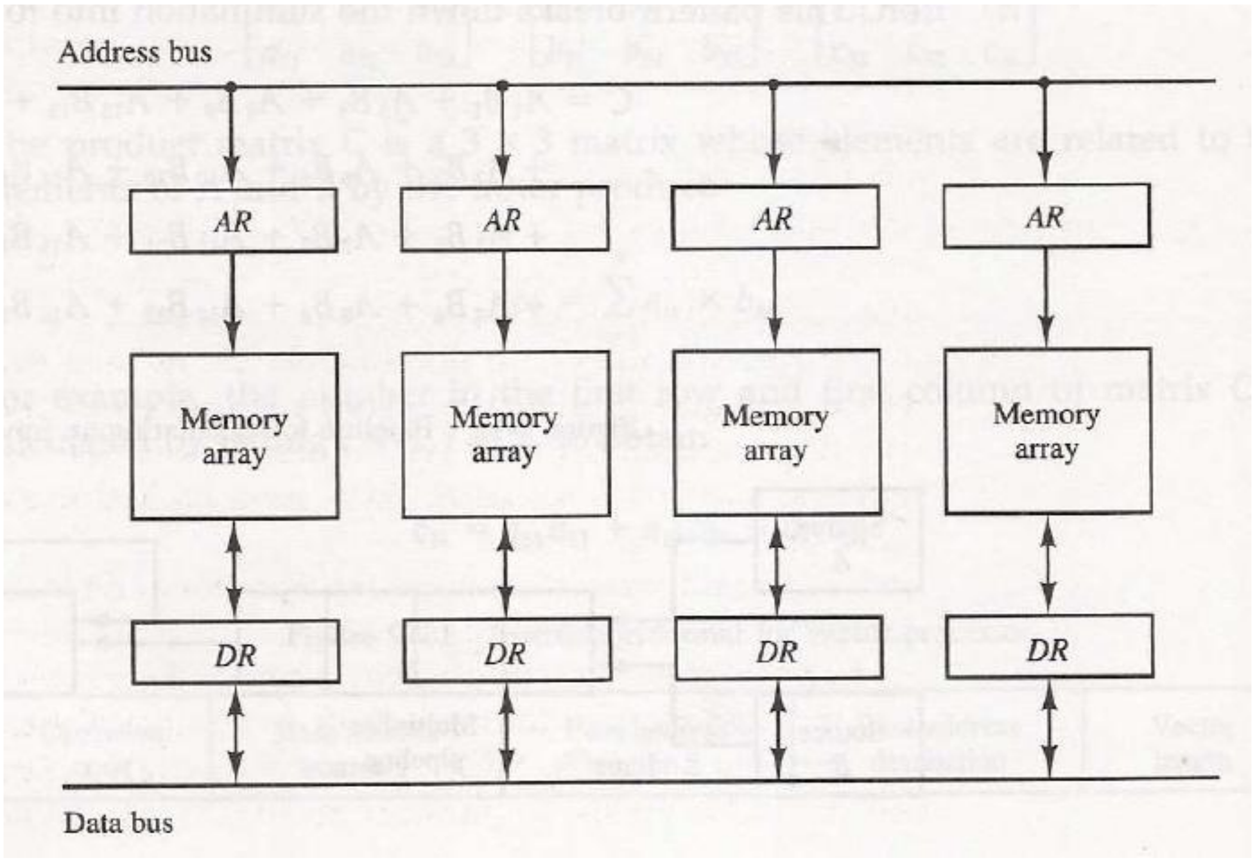


Figure: multiple module memory organization

Array processor

An array processor is a processor that performs computations on large array of data. An attached array processor is an auxiliary processor attached to a general purpose computer, intended to improve the performance of the host computer in specific numerical computation task.

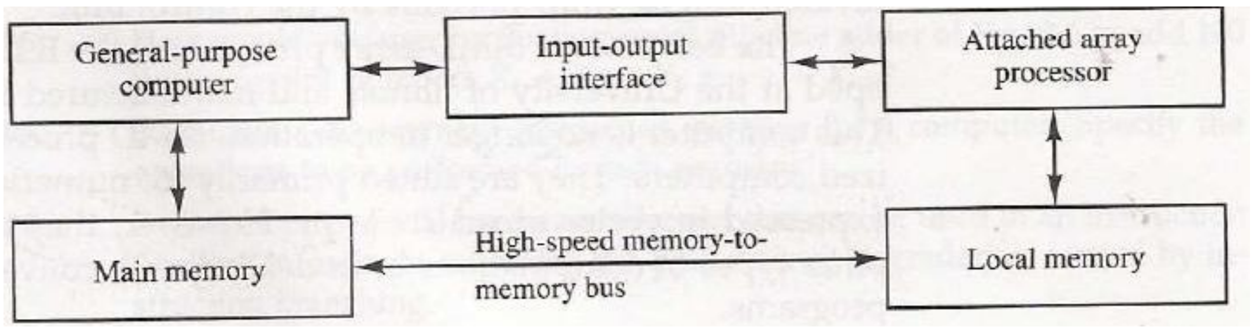


Figure: attached array processor with host computer

Attached array processor

Its purpose is to enhance the performance of the computer by providing vector processing for complex scientific application to a conventional host computer. It achieves high performance by means of parallel processing with multiple functional units, which include arithmetic unit containing one or more pipelined floating point adders and multipliers. The attached array processor can be connected to the back end of the general purpose commercial computers that treats like an external interface.

SIMD array processor

An SIMD array processor is a computer with multiple processing units operating in parallel. The processing units are synchronized to perform the same operation under the control of common control unit, thus providing a single instruction stream with multiple data (SIMD) organization. A block diagram is as shown in figure. It contains a set of identical processing elements (PEs) having its own memory M, each processor elements including ALU, floating point arithmetic unit and working registers. Master control unit controls the operations in processor elements and main memory for storage of programs. Vector instructions are broadcast to all PEs simultaneously each PEs uses operands stored in its local memory. Vector operands are distributed to the local memories prior to the parallel execution of the instruction.

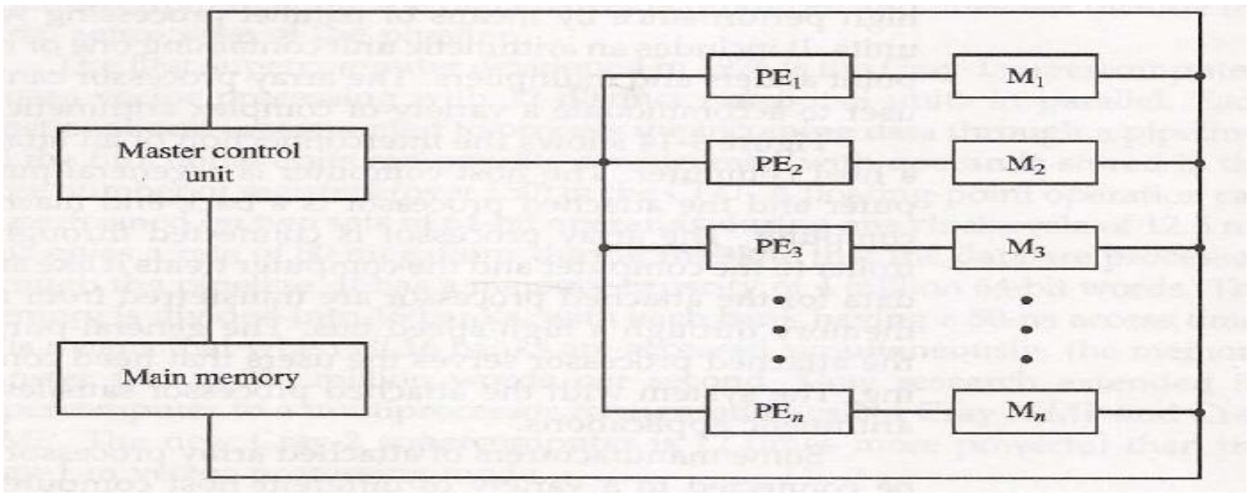


Figure: SIMD Array processor organization

Multithreading

The most important measure of performance for a processor is the rate at which it executes instructions. This can be expressed as

$$MIPS\, rate = f * IPC$$

Where f is the processor clock frequency, in MHz, and IPC (instructions per cycle) is the average number of instructions executed per cycle. Accordingly, designers have pursued the goal of increased performance on two fronts: increasing clock frequency and increasing the number of instructions executed or, more properly, the number of instructions that complete during a processor cycle. As designers have increased IPC by using an instruction pipeline and then by using multiple parallel instruction pipelines in a superscalar architecture. With pipelined and multiple-pipeline designs, the principal problem is to maximize the utilization of each pipeline stage. To improve throughput, designers have created ever more complex mechanisms, such as executing some instructions in a different order from the way they occur in the instruction stream and beginning execution of instructions that may never be needed. But this approach may be reaching a limit due to complexity and power consumption concerns.

An alternative approach, which allows for a high degree of instruction-level parallelism without increasing circuit complexity or power consumption, is called ***multithreading.*** In essence, the instruction stream is divided into several smaller streams, known as threads, such that the threads can be executed in parallel. The variety of specific multithreading designs, realized in both commercial systems and experimental systems.

Some of the terms used in multithreading are:

• **Process:** An instance of a program running on a computer. A process embodies two key characteristics:

—**Resource ownership:** A process includes a virtual address space to hold the process image; the process image is the collection of program, data, stack, and attributes that define the process. From time to time, a process may be allocated control or ownership of resources, such as main memory, I/O channels, I/O devices, and files.

—**Scheduling/execution:** The execution of a process follows an execution path (trace) through one or more programs. This execution may be interleaved with that of other processes. Thus, a process has an execution state (Running, Ready, etc.) and a dispatching priority and is the entity that is scheduled and dispatched by the operating system.

• **Process switch:** An operation that switches the processor from one process to another, by saving all the process control data, registers, and other information for the first and replacing them with the process information for the second

• **Thread:** A dispatchable unit of work within a process. It includes a processor context (which includes the program counter and stack pointer) and its own data area for a stack (to enable subroutine branching). A thread executes sequentially and is interruptible so that the processor can turn to another thread.

- **Thread switch:** The act of switching processor control from one thread to another within the same process. Typically, this type of switch is much less costly than a process switch.

Thus, a thread is concerned with scheduling and execution, whereas a process is concerned with both scheduling/execution and resource ownership. The multiple threads within a process share the same resources. This is why a thread switch is much less time consuming than a process switch. All of the commercial processors and most of the experimental processors so far have used explicit multithreading. These systems concurrently execute instructions from different explicit threads, either by interleaving instructions from different threads on shared pipelines or by parallel execution on parallel pipelines. Implicit multithreading refers to the concurrent execution of multiple threads extracted from a single sequential program. These implicit threads may be defined either statically by the compiler or dynamically by the hardware.

Approaches to Explicit Multithreading

At minimum, a multithreaded processor must provide a separate program counter for each thread of execution to be executed concurrently. The designs differ in the amount and type of additional hardware used to support concurrent thread execution. In general, instruction fetching takes place on a thread basis. The processor treats each thread separately and may use a number of techniques for optimizing single-thread execution, including branch prediction, register renaming, and superscalar techniques.

Broadly speaking, there are four principal approaches to multithreading:

- **Interleaved multithreading:** This is also known as **fine-grained multithreading**. The processor deals with two or more thread contexts at a time, switching from one thread to another at each clock cycle. If a thread is blocked because of data dependencies or memory latencies, that thread is skipped and a ready thread is executed.

- **Blocked multithreading:** This is also known as **coarse-grained multithreading**. The instructions of a thread are executed successively until an event occurs that may cause delay, such as a cache miss. This event induces a switch to another thread. This approach is effective on an in-order processor that would stall the pipeline for a delay event such as a cache miss.

- **Simultaneous multithreading (SMT):** Instructions are simultaneously issued from multiple threads to the execution units of a superscalar processor. This combines the wide superscalar instruction issue capability with the use of multiple thread contexts.

- **Chip multiprocessing:** In this case, the entire processor is replicated on a single chip and each processor handles separate threads. The advantage of this approach is that the available logic area on a chip is used effectively without depending on ever-increasing complexity in pipeline design. This is referred to as multi-core.

For the first two approaches, instructions from different threads are not executed simultaneously. Instead, the processor is able to rapidly switch from one thread to another, using a different set of registers and other context information. This results in a better utilization of the processor's execution resources and avoids a large penalty due to cache misses and other latency events. The SMT approach involves true simultaneous execution of instructions from different threads, using replicated execution resources. Chip multiprocessing also enables simultaneous execution of instructions from different threads. Figure illustrates some of the possible pipeline architectures that involve multithreading and contrasts these with approaches that do not use multithreading. Each horizontal row represents the potential issue slot or slots for a single execution cycle; that is, the width of each row corresponds to the maximum number of instructions that can be issued in a single clock cycle.3 The vertical dimension represents the time sequence of clock cycles. An empty (shaded) slot represents an unused execution slot in one pipeline. A no-op is indicated by N.

The first three illustrations in Figure 17.8 show different approaches with a scalar (i.e., single-issue) processor:

- **Single-threaded scalar:** This is the simple pipeline found in traditional RISC and CISC machines, with no multithreading.
- **Interleaved multithreaded scalar:** This is the easiest multithreading approach to implement. By switching from one thread to another at each clock cycle, the pipeline stages can be kept fully occupied, or close to fully occupied. The hardware must be capable of switching from one thread context to another between cycles.
- **Blocked multithreaded scalar:** In this case, a single thread is executed until a latency event occurs that would stop the pipeline, at which time the processor switches to another thread. Figure 17.8c shows a situation in which the time to perform a thread switch is one cycle, whereas Figure 17.8b shows that thread switching occurs in zero cycles.
- **Superscalar:** This is the basic superscalar approach with no multithreading. Until relatively recently, this was the most powerful approach to providing parallelism within a processor. Note that during some cycles, not all of the available issue slots are used. During these cycles, less than the maximum number of instructions is issued; this is referred to as *horizontal loss*. During other instruction cycles, no issue slots are used; these are cycles when no instructions can be issued; this is referred to as *vertical loss*.
- **Interleaved multithreading superscalar:** During each cycle, as many instructions as possible are issued from a single thread. With this technique, potential delays due to thread switches are eliminated, as previously discussed. However, the number of instructions issued in any given cycle is still limited by dependencies that exist within any given thread.
- **Blocked multithreaded superscalar:** Again, instructions from only one thread may be issued during any cycle, and blocked multithreading is used.
- **Very long instruction word (VLIW):** A VLIW architecture, such as IA-64, places multiple instructions in a single word. Typically, a VLIW is constructed by the compiler, which places operations that may be executed in parallel in the same word. In a simple

VLIW machine (Figure 17.8g), if it is not possible to completely fill the word with instructions to be issued in parallel, no-ops are used.

- **Interleaved multithreading VLIW:** This approach should provide similar efficiencies to those provided by interleaved multithreading on a superscalar architecture.
- **Blocked multithreaded VLIW:** This approach should provide similar efficiencies to those provided by blocked multithreading on a superscalar architecture. The final two approaches illustrated in Figure 17.8 enable the parallel, simultaneous execution of multiple threads:
- **Simultaneous multithreading:** Figure 17.8i shows a system capable of issuing 8 instructions at a time. If one thread has a high degree of instruction-level other cycles, instructions from two or more threads may be issued. If sufficient threads are active, it should usually be possible to issue the maximum number of instructions on each cycle, providing a high level of efficiency.
- **Chip multiprocessor (multicore):** Figure 17.8k shows a chip containing four processors, each of which has a two-issue superscalar processor. Each processor is assigned a thread, from which it can issue up to two instructions per cycle.

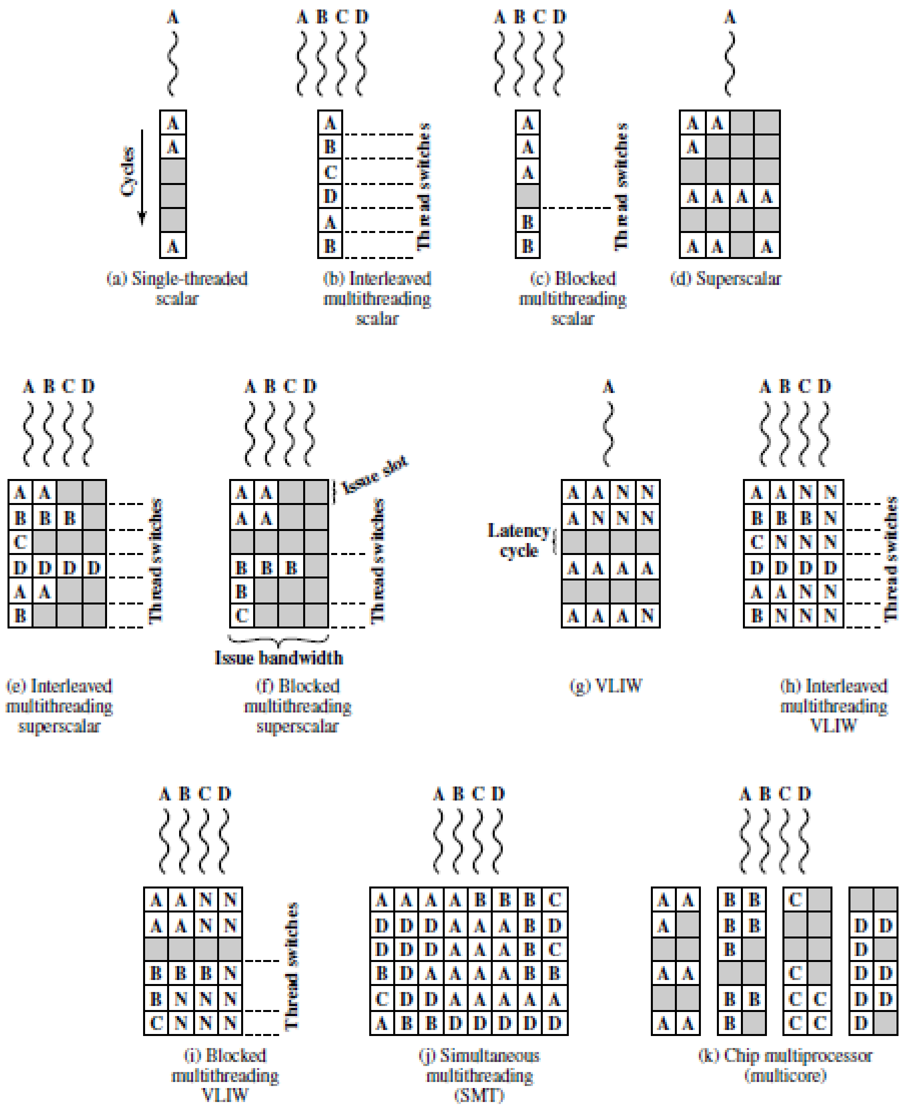

Cycles
Thread switches
Issue slot
Issue bandwidth
Latency cycle
(a) Single-threaded scalar
(b) Interleaved multithreading scalar
(c) Blocked multithreading scalar
(d) Superscalar
(e) Interleaved multithreading superscalar
(f) Blocked multithreading superscalar
(g) VLIW
(h) Interleaved multithreading VLIW
(i) Blocked multithreading VLIW
(j) Simultaneous multithreading (SMT)
(k) Chip multiprocessor (multicore)

Figure 17.8 Approaches to Executing Multiple Threads

Interconnection structure in multiprocessor system

The components that form a multiprocessor system are CPUs, IOPs connected to input-output devices and a memory unit that may be portioned into a number of separate modules. The interconnection between the components can have different physical configuration depending on the number of transfer paths that are available between the processor and memory in a shared memory system or among the processing elements in a loosely coupled system. Some of these schemes are represented as below:

1. Time shared common bus.
2. Multi-port memory.
3. Crossbar switch.
4. Multi-stage switching network.
5. Hypercube system.

1. Time shared common bus

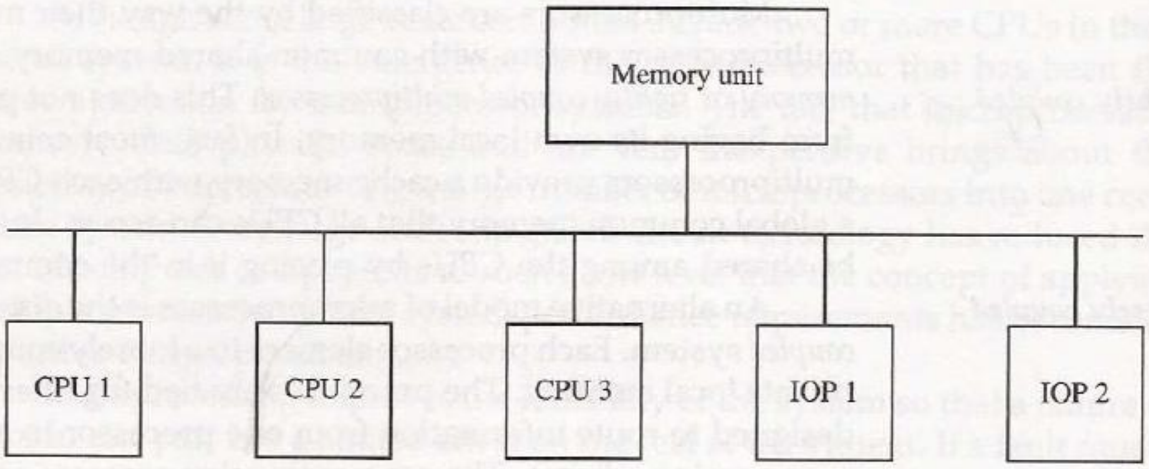


Figure 13-1 Time-shared common bus organization.

- Contains number of multiprocessor connected through a common path to a memory unit. A time shared common bus for five processor is shown as in figure.
- Only one processor can communicate with the memory or another processor at any given time. Transfer 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 process address of the destination unit to initiate the transfer.
- A command is issued to inform the destination unit what operation is to be performed, then receiving unit recognizes its address in the bus and responds to the control signal from sender, after which the transfer is initiated. The conflicts if any must be resolved by incorporating a bus controller that established 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 memory all other processors are either busy with internal operations or must remain idle for waiting for bus. As consequences the total overflow of the transfer will be limited by the speed of single path.

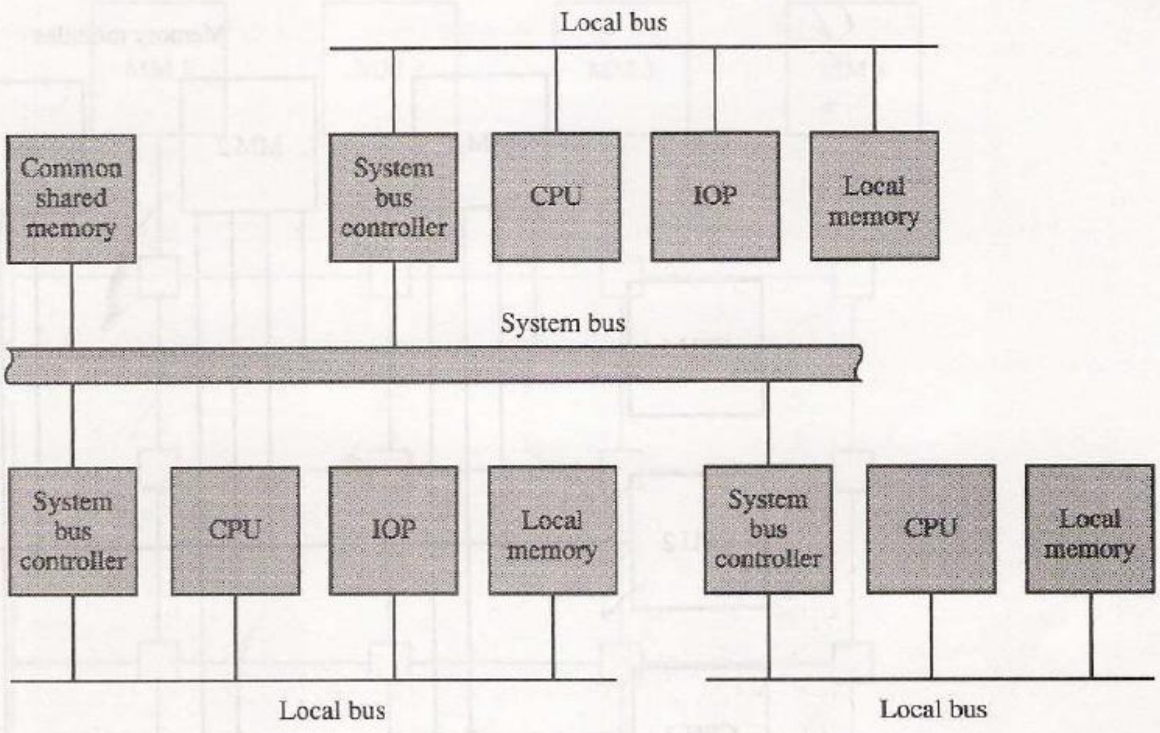


Figure 13-2 System bus structure for multiprocessors.

2. Multiport memory

- A multiport memory system employs separate buses between each memory module and each CPU. As in figure a processor bus consists of address, data and control lines required to communicate with memory. The memory 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.
- The priority for memory access associated with each processor may be established by the physical port position that is bus occupies in each module. i.e. CPU1 will have priority over CPU2, the CPU3….
- The advantage of multiport memory organization is high transfer rate that can be achieved because of the multiple paths between processor and memory.
- The disadvantage is that it required expensive memory control logic and large number of cables and connectors.

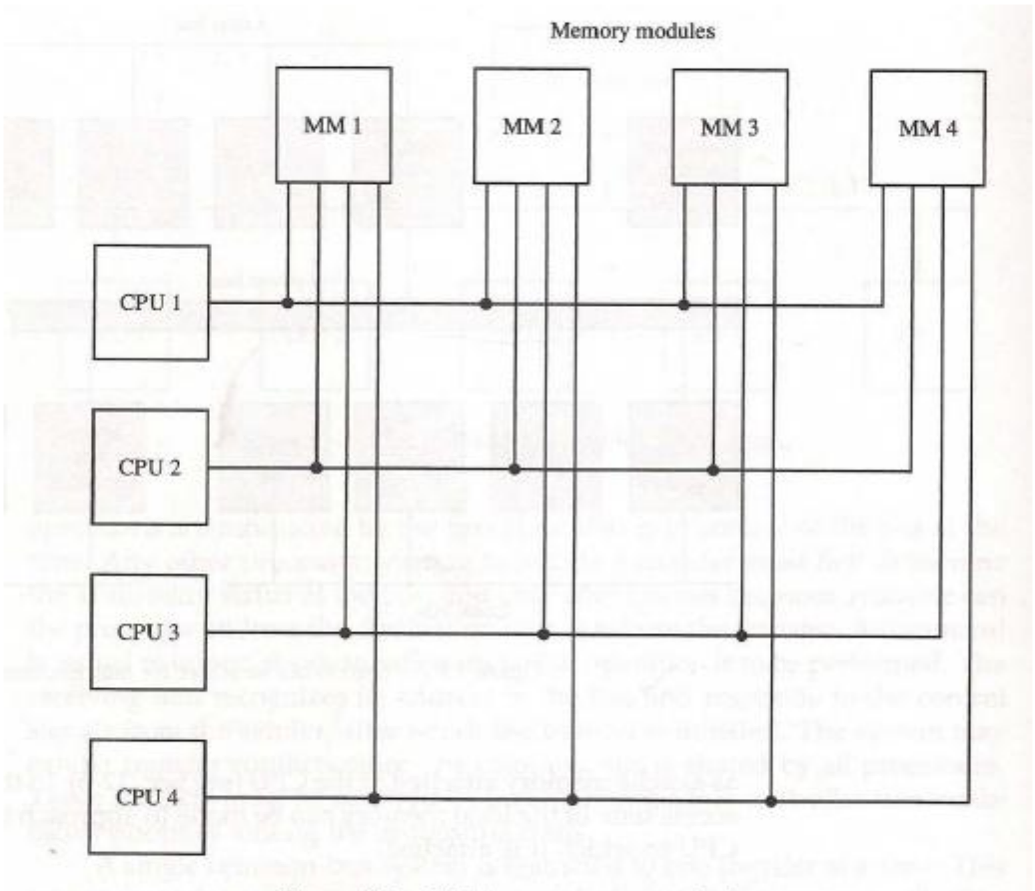


Figure 13-3 Multiport memory organization.

3. Crossbar switch

- The crossbar switch organization consists of a number of cross points that are placed at intersection between processor buses and memory module paths.
- The small square is each cross point is a switch that determines the path from processor to 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 receives multiple requests for access to same memory module on pre-defined priority basis.

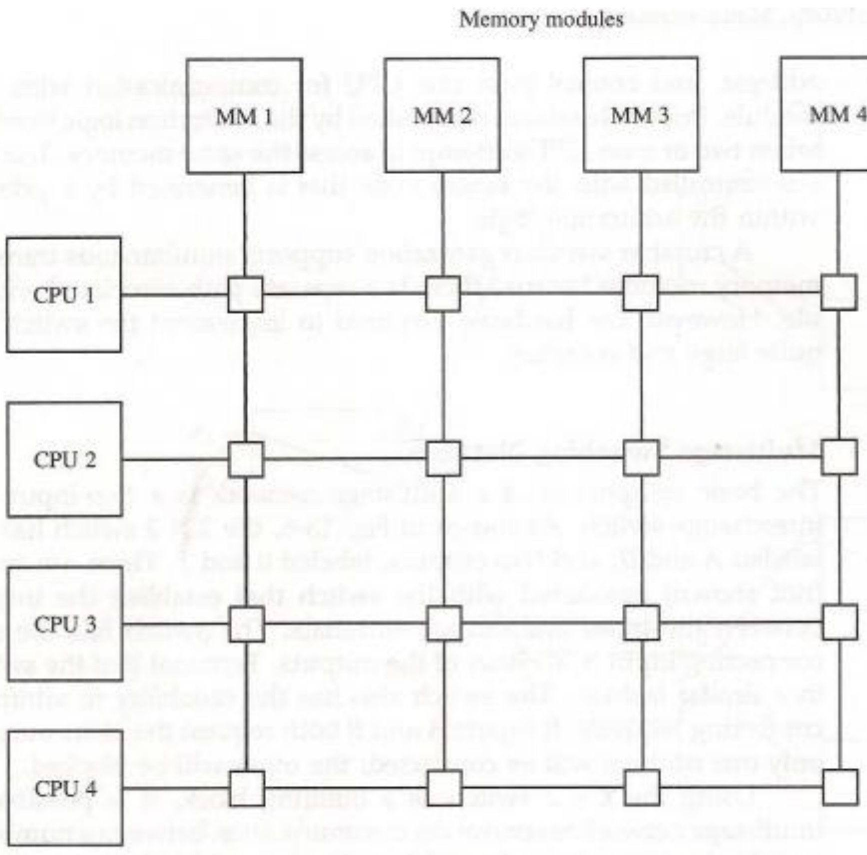
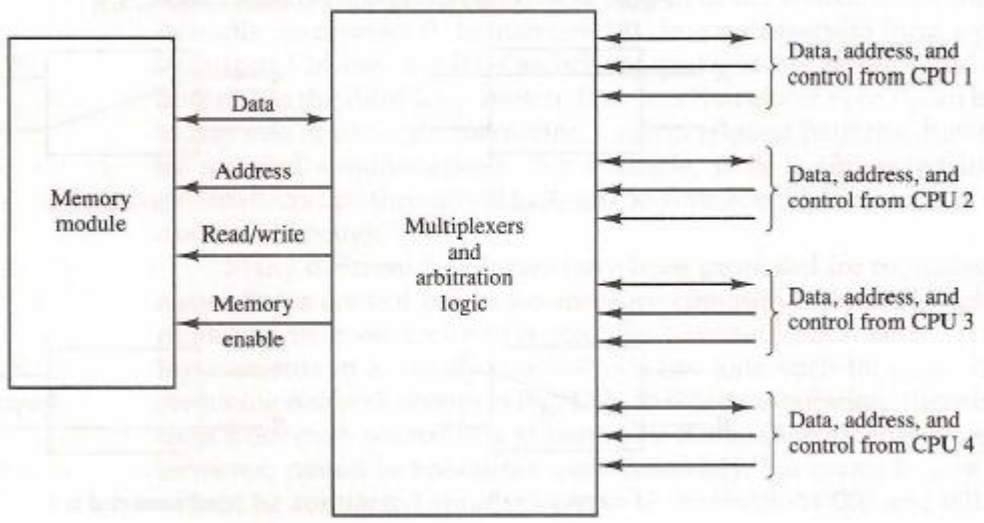


Figure 13-4 Crossbar switch.

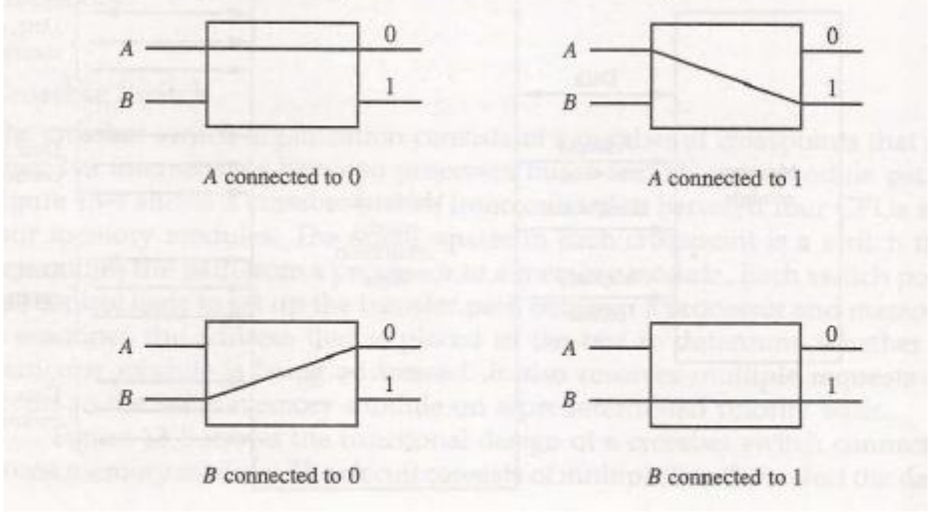
Figure 13-5 Block diagram of crossbar switch.


Memory module
Data
Address
Read/write
Memory enable
Multiplexers and arbitration logic
Data, address, and control from CPU 1
Data, address, and control from CPU 2
Data, address, and control from CPU 3
Data, address, and control from CPU 4

- A crossbar switch organization supports simultaneous transfer from memory module because there is a separate path associated with each module. However the hardware required to implement the switch can become quite large and complex.

4. Multistage switching network

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



- The basic component of a multistage network is a two-input, two output interchange switch. As in figure switch has two input labeled A and B two outputs 0 & 1. The control signal (not shown) associated with the switch that establishes the interconnection between input and output terminals. The switch has the capability of connecting input A or B to either of the outputs.

- The switch has the capability to arbitrate between conflicting requests i.e. only one will be connected and other will be blocked.

- Figure shows how 2x2 switch can be used as building block. As in figure two processor p1 and p2 are connected through switch to eight memory modules. Marked in binary form 000 through 111. The path from source to destination is determined fro binary bits of the destination number.

- The first bit of the destination number determines the switch output in first level. The second bit specifies the output of the switch in the second level and third bit specifies the output of the switch in third level.

- Example: if P1 is connected to one of the destinations 000 through 011, p2 can be connected to only one of the destinations 100 through 111.

- Many different topologies have been proposed for multistage switching networks to control process memory communication in a tightly coupled multiprocessor system or to control the communication between the processing elements in loosely couples system. One such topology is the omega switching network in figure.

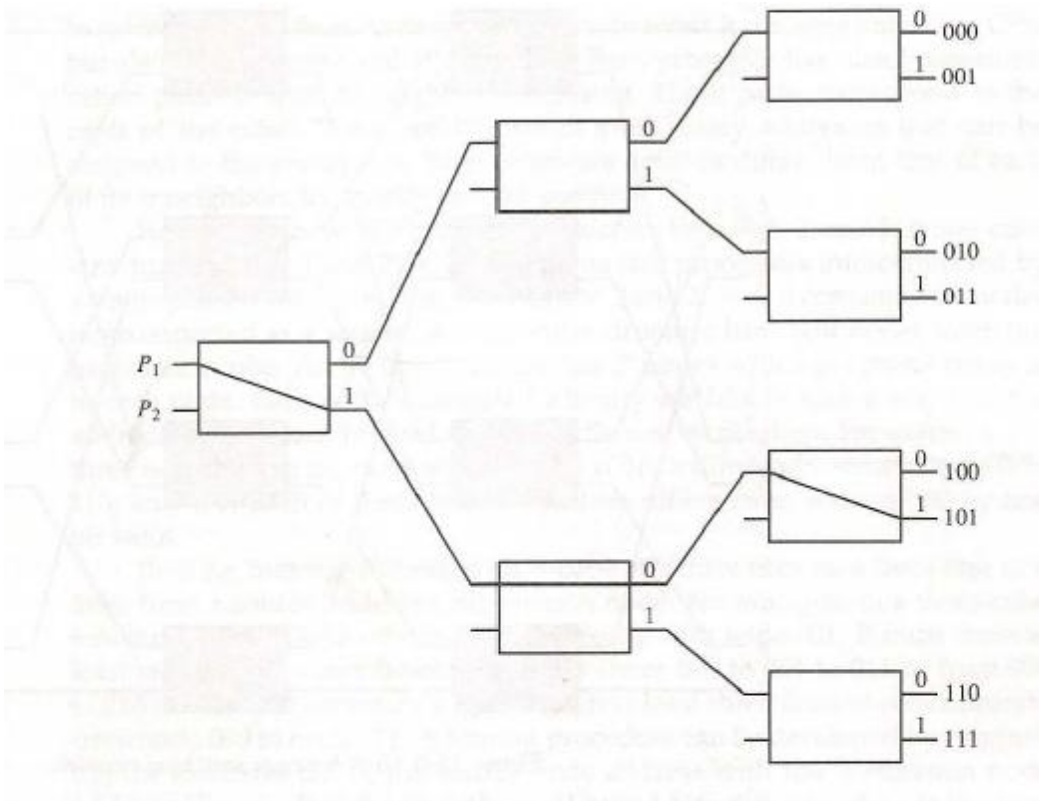


Figure 13-7 Binary tree with 2 × 2 switches.

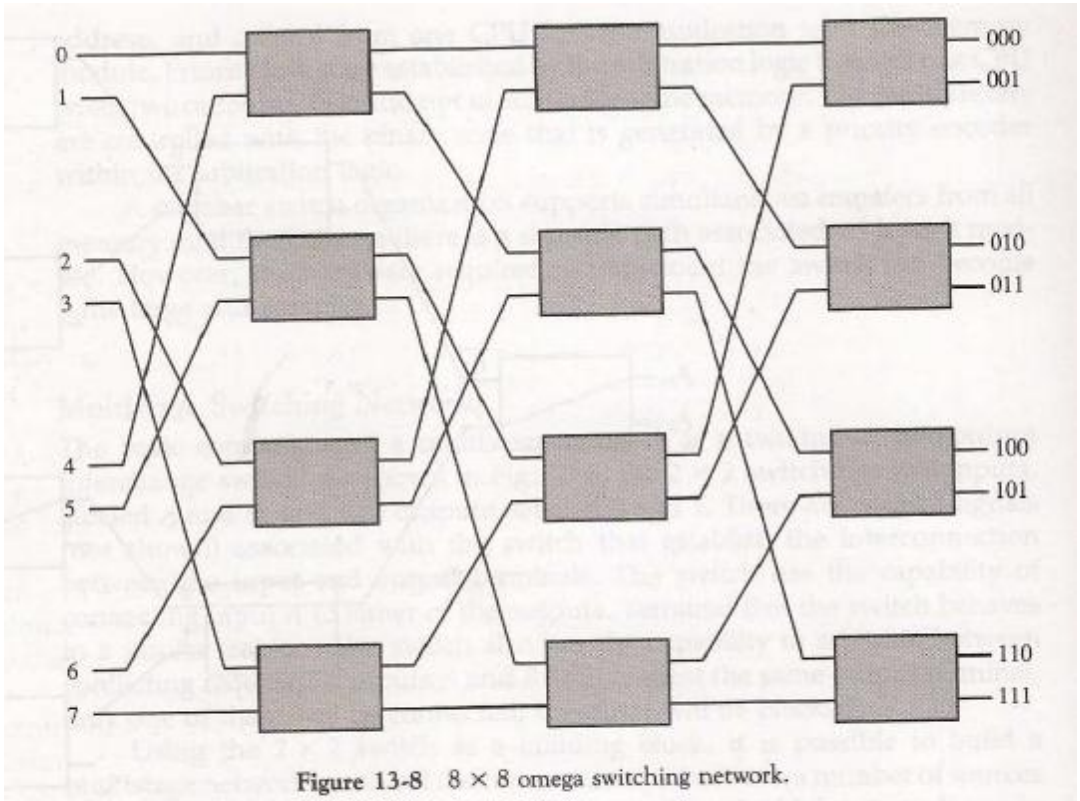


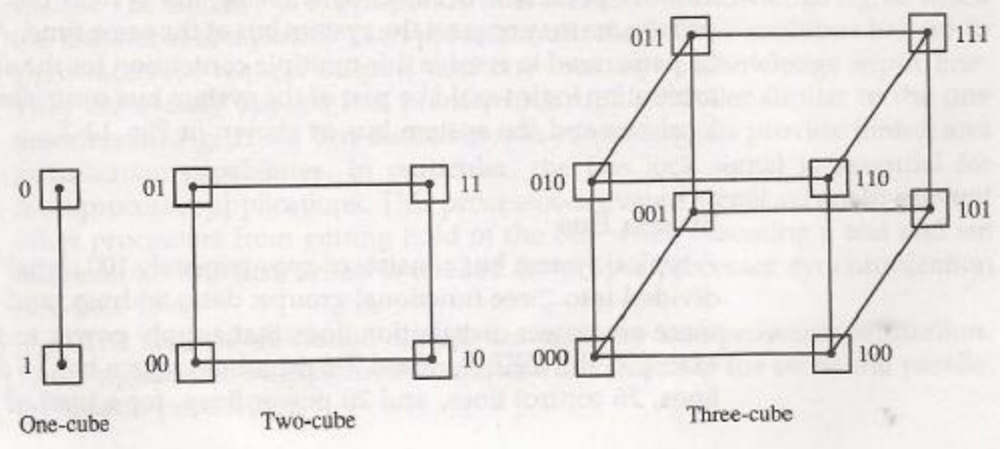
Figure 13-8 8 × 8 omega switching network.

5. Hypercube interconnection

The hyper cube or binary n-cube multi-processor structure is a loosely coupled composed of N=2^n processor interconnected in an n-dimensional binary cube. Each processor from a node of the cube, although it is customary to refer to ach 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 processor. Figure shows the hypercube structure for n=1 and $2^n=2$, for n=2, $2^2=4$ nodes connected in a square.

For example the three neighbor of the ode with address 100 in a three cube structure are 000,110 and 101. Each address differs from address 100 by one bit value. Routing message through an n-cube structure may take form one to n links from source node to 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→001→011 or from 000→010→011). It is necessary to go through at least three links to communicate from node 000 to 111. A routing procedure can be developed by computing XOR 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 the axes. E.g. in a three cube structure a message at 010 going to 001 produce an XOR of the two address equal to 011. The message can be sent along the second axis to 000 and the through the third axis to 001.

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


0
1
One-cube
01
11
00
10
Two-cube
011
111
010
110
001
101
000
100
Three-cube