CHAPTER 11

Pipelining

Learning objectives

- Introduction
- Pipelining
- Concept of Pipeline
- Arithmetic, Branch, and Load-Store Instruction Pipelining
- Basic Performance Issues in Pipelining
- Hazards in Pipelining
- Advanced Compiler Technology
- Parallel Architecture
- Taxonomy of Parallel Architectures
- Symmetric Multiprocessors
- Non-uniform Memory Access (NUMA)
- Interconnection Networks
- Cache Coherence

INTRODUCTION

The use of multiple buses and direct paths in the processor data path enables us to reduce the number of steps required to fetch and execute instructions. Can we further reduce the number of steps by increasing the connectivity? It may be possible; however, it is very difficult, because many of the steps are somewhat sequential in nature. Instead of reducing the number of steps required to fetch and execute each instruction, we can, however, reduce the total number of steps required to fetch and execute a sequence of instructions by overlapping the interpretation of multiple instructions. In order to overlap the execution of multiple instructions, a commonly used technique is pipelining, similar in spirit to the assembly-line processing used in factories.

In this Chapter, we discuss the basic concepts involved in designing instruction pipelines. Performance measures of a pipeline are introduced. The main issues contributing to instruction pipeline hazards are discussed and some possible solutions are introduced. In addition, we introduce the concept of arithmetic pipelining together with the problems involved in designing such a pipeline. Our coverage concludes with a review of a recent pipeline processor and parallel architectures.

11.1. PIPELINING

Pipelining is an implementation technique whereby multiple instructions are overlapped in execution; it takes advantage of parallelism that exists among the actions needed to execute an instruction. Today, pipelining is the key implementation technique used to make fast CPUs.

A pipeline is like an assembly line. In an automobile assembly line, there are many steps, each contributing something to the construction of the car. Each step operates in parallel with the other steps, although on a different car. In a computer pipeline, each step in the pipeline completes a part of an instruction.

Like the assembly line, different steps are completing different parts of different instructions in parallel. Each of these steps is called a pipe stage or a pipe segment. The stages are connected one to the next to form a pipe-instructions enter at one end, progress through the stages, and exit at the other end, just as cars would in an assembly line.

Small Concept

Pipelining is technique to execute multiple instruction simultensly step-by-step.

In an automobile assembly line, throughput is defined as the number of cars per hour and is determined by how often a completed car exits the assembly line. Likewise, the throughput of an instruction pipeline is determined by how often an instruction exits the pipeline. Because the pipe stages are hooked together, all the stages must be ready to proceed at the same time, just as we would require in an assembly line. The time required between moving an instruction one step down the pipeline is a processor cycle. Because all stages proceed at the same time, the length of a processor cycle is determined by the time required for the slowest pipe stage, just as in an auto assembly line, the longest step would determine the time between advancing the line. In a computer, this processor cycle is usually 1 clock cycle (sometimes it is 2, rarely more).

Similar to the assembly line, the success of a pipeline depends upon dividing the execution of an instruction among a number of subunits (stages), each performing part of the required operations. A possible division is to consider instruction fetch (F), instruction decode (D), operand fetch (F), instruction execution (E), and store of results (S) as the subtasks needed for the execution of an instruction. In this case, it is possible to have up to five instructions in the pipeline at the same time, thus reducing instruction execution latency.

11.2. CONCEPT OF PIPELINE

Pipelining refers to the technique in which a given task is divided into a number of subtasks that need to be performed in sequence. Each subtask is performed by a given functional unit. The units are connected in a serial fashion and all of them operate simultaneously. The use of pipelining improves the performance compared to the traditional sequential execution of tasks. Figure 11.1 shows an illustration of the basic difference between executing four subtasks of a given instruction (in this case fetching F, decoding D, execution E, and writing the results W) using pipelining and sequential processing.

As shown in the figure, it is clear that the total time required to process three instructions (I_1, I_2, I_3) is only six time units if four-stage pipelining is used as compared to 12 time units if sequential processing is used. A possible saving of up to 50% in the execution time of these

three instructions is obtained. In order to formulate some performance measures for the goodness of a pipeline in processing a series of tasks, a space time chart (called the Gantt's chart) is used. The chart shows the succession of the subtasks in the pipe with respect to time.

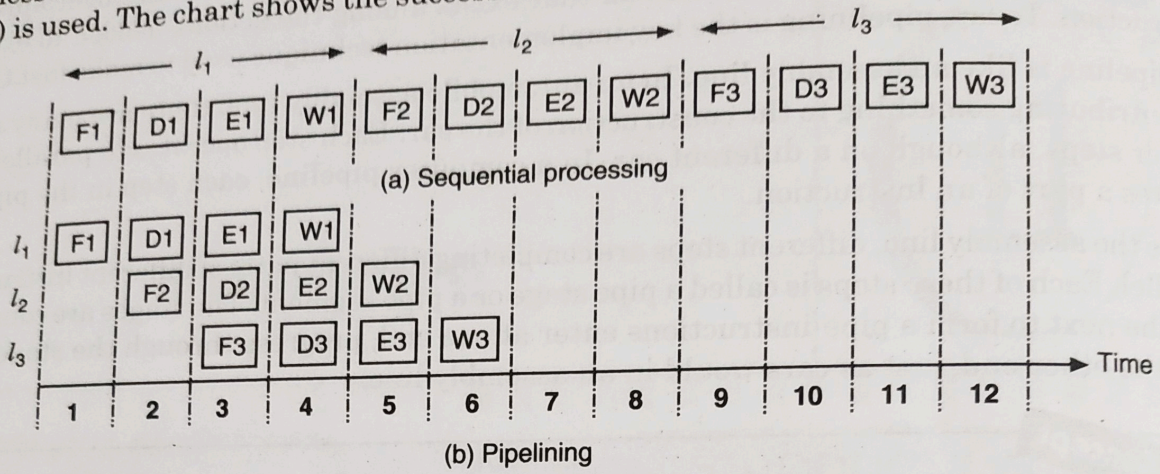


Fig. 11.1. Pipelining versus sequential processing (adapted from book Fundamentals of Computer Organization and Architecture, by M. Abd-El-Barr and H. El-Rewini)

Small Concept

The use of pipelining improves the performance compared to the traditional sequential execution of tasks.

Figure 11.2, shows a Gantt's chart. In this chart, the vertical axis represents the subunits (four in this case) and the horizontal axis represents time (measured in terms of the time unit required for each unit to perform its task). In developing the Gantt's chart, we assume that the time (T) taken by each subunit to perform its task is the same; we call this the unit time.

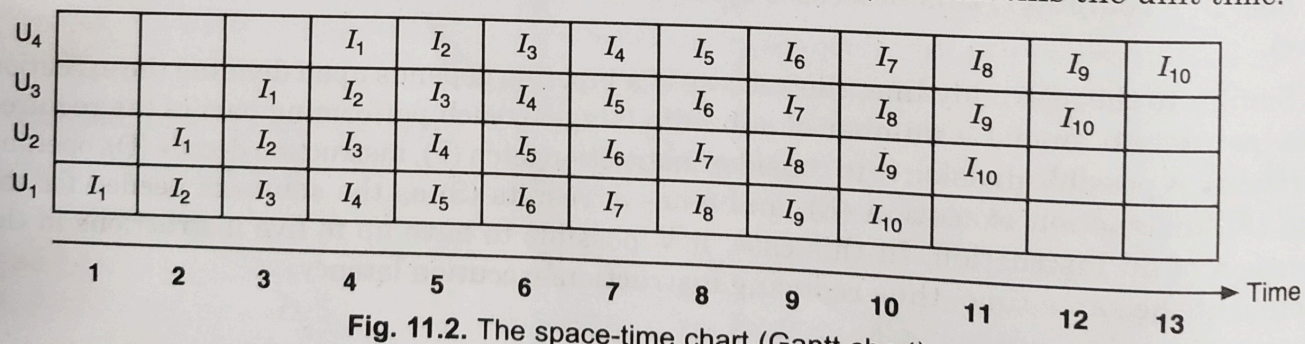


Fig. 11.2. The space-time chart (Gantt chart)

As can be seen from the figure, 13 time units are needed to finish executing 10 instructions (I_1 to I_{10}). This is to be compared to 40 time units if sequential processing is used (ten instructions each requiring four time units).

11.3. ARITHMETIC, BRANCH, AND LOAD-STORE INSTRUCTION PIPELINING

The flow of instructions through a pipeline follows the steps normally taken when an instruction is executed. In the discussion below we consider how three classes of instructions: arithmetic, branch, and load-store, are executed, and then we relate this to how the instructions are pipelined.

Arithmetic Instruction: Consider the "normal" sequence of events when an arithmetic instruction is executed in a load-store machine:

1. Fetch the instruction from memory;
2. Decode the instruction (it is an arithmetic instruction, but the CPU has to find that out through a decode operation);
3. Fetch the operands from the register file;
4. Apply the operands to the ALU;
5. Write the result back to the register file.

Branch Instructions: There are similar patterns for other instruction classes. For branch instructions the sequence is:

1. Fetch the instruction from memory;
2. Decode the instruction (it is a branch instruction);
3. Fetch the components of the address from the instruction or register file;
4. Apply the components of the address to the ALU (address arithmetic);
5. Copy the resulting effective address into the PC, thus accomplishing the branch.

Load and Store Instructions: The sequence for load and store instructions is:

1. Fetch the instruction from memory;
2. Decode the instruction (it is a load or store instruction);
3. Fetch the components of the address from the instruction or register file;
4. Apply the components of the address to the ALU (address arithmetic);
5. Apply the resulting effective address to memory along with a read (load) or write (store) signal. If it is a write signal, the data item to be written must also be retrieved from the register file.

The three sequences above show a high degree of similarity in what is done at each stage:

1. Fetch,
2. Decode,
3. Operand fetch,
4. ALU operation,
5. Result writeback.

These five phases are similar to the four phases discussed in chapter 6 except that we have refined the fourth phase, "execute," into two subphases: ALU operation and writeback, as illustrated in Fig. 11.3. A result writeback is not always needed, and one way to deal with this is to have two separate subphases (ALU and writeback) with a bypass path for situations when a writeback is not needed. For this discussion, we take a simpler approach, and force all instructions to go entirely through each phase, whether or not that is actually needed.

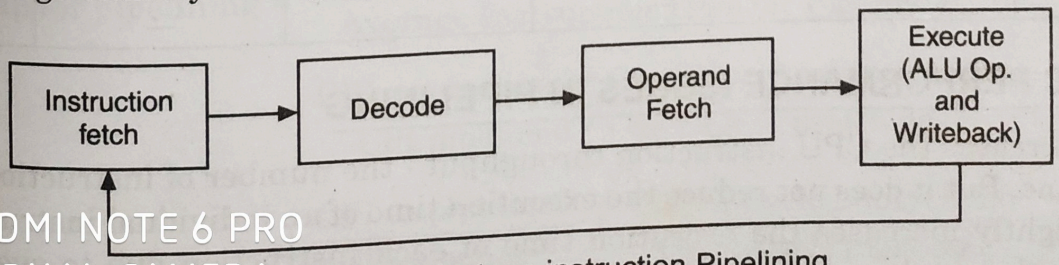


Fig. 11.3. Four stage instruction Pipelining

How Pipelines Works: The pipeline is divided into segments and each segment can execute it operation concurrently with the other segments. Once a segment completes an operation, it passes the result to the next segment in the pipeline and fetches the next operations from the preceding segment.

Advantages:

- More efficient use of processor.
- Quicker time of execution of large number of instructions.

Disadvantages:

- Pipelining involves adding hardware to the chip.
- Inability to continuously run the pipeline at full speed because of pipeline hazard switch disrupts, the smooth execution of the pipeline.

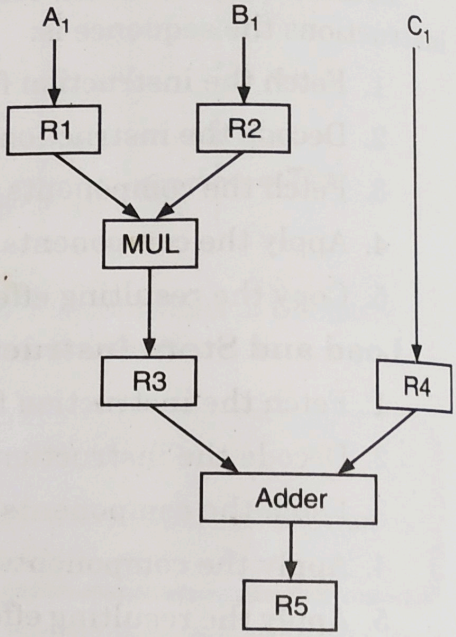


Fig. 11.4.

Example of Instruction Pipelining: Suppose we want to perform the combined multiply and add operations with a stream of numbers:

$$A_i * B_i + C_i \text{ for } i = 1, 2, 3, \ldots, 7$$

The sub-operations performed in each segment of the pipeline are as follows:

$$R_1 \rightarrow A_i,$$
$$R_2 \rightarrow B_i$$
$$R_3 \rightarrow R_1 * R_2$$
$$R_4 \rightarrow C_i$$
$$R_5 \rightarrow R_3 + R_4$$

Clock Pulse Number	Segment 1		Segment 2		Segment 3
	R_1	R_2	R_3	R_4	R_5
1	A_1	B_1	—	—	—
2	A_2	B_2	$A_1 * B_1$	C_1	—
3	A_3	B_3	$A_2 * B_2$	C_2	$A_1 * B_1 + C_1$
4	A_4	B_4	$A_3 * B_3$	C_3	$A_2 * B_2 + C_2$
5	A_5	B_5	$A_4 * B_4$	C_4	$A_3 * B_3 + C_3$
6	A_6	B_6	$A_5 * B_5$	C_5	$A_4 * B_4 + C_4$
7	A_7	B_7	$A_6 * B_6$	C_6	$A_5 * B_5 + C_5$
8	—	—	$A_7 * B_7$	C_7	$A_6 * B_6 + C_6$
9	—	—	—	—	$A_7 * B_7 + C_7$

11.4. BASIC PERFORMANCE ISSUES IN PIPELINING

Pipelining increases the CPU instruction throughput - the number of instructions completed per unit of time. But it does not reduce the execution time of an individual instruction. In fact, it usually slightly increases the execution time of each instruction due to overhead in the pipeline control.

The increase in instruction throughput means that a program runs faster and has lower total execution time.

Limitations on practical depth of a pipeline arise from:

- **Pipeline Latency:** The fact that the execution time of each instruction does not decrease puts limitations on pipeline depth;
- **Imbalance Among Pipeline Stages:** Imbalance among the pipe stages reduces performance since the clock can run no faster than the time needed for the slowest pipeline stage;
- **Pipeline Overhead:** Pipeline overhead arises from the combination of pipeline register delay (setup time plus propagation delay) and clock skew.

Once the clock cycle is as small as the sum of the clock skew and latch overhead, no further pipelining is useful, since there is no time left in the cycle for useful work.

EXAMPLE 11.1.

An unpipelined processor with 5 execution stages of 30 nano seconds (ns), 40 ns, 30 ns, 40 ns, and 50 ns.

$$\text{Instruction latency} = (30 + 40 + 30 + 40 + 50)\ \text{ns}$$
$$= 190\ \text{ns}$$

Assume that we implement pipelining on this processor. Also assume that with pipelining, the clock skew adds 5 ns of overhead to each execution stage. In the pipelined implementation, the length of the pipe stages must all be same which is the speed of the slowest stage plus overhead.

$$\text{Length of a pipelined stage} = \text{MAX(lengths of unpipelined stages)} + \text{overhead}$$
$$= (50 + 5)\ \text{ns}$$
$$= 55\ \text{ns}$$
$$\text{Instruction latency} = 55\ \text{ns}$$
$$\text{Average instruction time for non-pipelined} = 190\ \text{ns}$$
$$\text{Average instruction time for pipelined} = 55\ \text{ns}$$
$$\text{Speedup} = 190 / 55$$
$$= 3.45$$

Speedup Equation for Pipelining:

$$\text{Speedup for pipelining} = \frac{\text{Average Instruction Time for Unpipelined}}{\text{Average Instruction Time for Pipelined}}$$

$$= \frac{\text{CPI unpipelined} * \text{Clock Cycle unpipelined}}{\text{CPI pipelined} * \text{Clock Cycle pipelined}}$$

$$= \frac{\text{CPI unpipelined}}{\text{CPI pipelined}} * \frac{\text{Clocvk Cycle unpipelined}}{\text{Clock Cycle pipelined}}$$

where CPI is clock cycles per instruction.

$$\text{Speedup for pipeling} = \frac{\text{Ideal CPI} * \text{Pipeline Depth}}{\text{CPi Pipelined}} * \frac{\text{Clock Cycle unpipelined}}{\text{Clock Cycle pipelined}}$$

$$\text{CPI Pipelined} = \text{Ideal CPI} + \text{Pipeline stall clock cycles per instructions}$$

$$\text{Speedup for pipeling} = \frac{\text{Ideal CPI} * \text{Pipeline Depth}}{\text{Ideal CPI} * \text{Pipeline stall CPI}} * \frac{\text{Clock Cycle unpipelined}}{\text{Clock Cycle pipelined}}$$

$$\text{Speedup for pipeling} = \frac{\text{Pipeline Depth}}{1 + \text{Pipeline stall CPI}} * \frac{\text{Clock Cycle Unpipelined}}{\text{Clock Cycle pipelined}}$$

EXAMPLE 11.2.

Given a non-pipelined architecture running at 1 GHz, that takes 5 cycle to finish an instruction. You want to make it pipelined with 5 stages. The increase in hardware forces you to run the machine at 800 MHz. The only stall caused by

- Memory - (30% of the total instructions) a stall of 70 cycles happens in 2 % of the memory instructions.
- Branch - (20 % of the total instructions) a stall of 2 cycles happens in 20 % of the memory instructions.

What is the speedup?

Penalty (memory) = 30 % * 2 % * 70 cycles = 0.42 cycles

Penalty (branch) = 20 % * 20 % * 2 cycles = 0.08 cycles

CPI unpipelined = 5

Ideal CPI = CPI unpipelined / Pipeline depth

= 5 / 5 = 1

CPI Pipelined = Ideal CPI + Pipelined stall clock cycles per instructions

= 1 + Penalty (memory) + Penalty (branch)

= 1 + 0.42 + 0.08 = 1.5

Clock cycle unpipelined = 1 ns

Clock cycle pipelined = (1 GHz / 0.8 GHz) * 1 ns

= 1.25 ns

$$\text{Speedup for pipeling} = \frac{\text{CPI Unpipelined}}{\text{CPI Pipelined}} * \frac{\text{Clock Cycle unpipelined}}{\text{Clock Cycle pipelined}}$$

$$= \frac{5}{1.5} * \frac{1}{1.25} = 2.67$$

11.5. HAZARDS IN PIPELINING

Pipelining can efficiently increase the performance of a processor by overlaping execution of instructions. But the efficiency of the pipelining depends upon, how the problem encountered during the implementation of pipelining is handled. These problems are known as as Hazards.

Types of Hazards:

- Structural Hazards (Resource Bound).
- Control Hazards (Pipelining Bubbles / branch difficulties).
- Data Hazards (Data Dependencies).

Structural Hazards

When a machine is pipelined, the overlapped execution of instructions requires pipelining of functional units and duplication of resources to allow all possible combinations of instructions in the pipeline.

If some combination of instructions cannot be accommodated because of a resource conflict, the machine is said to have a structural hazard.

Common instances of structural hazards arise when:

- Some functional unit is not fully pipelined. Then a sequence of instructions using that unpipelined unit cannot proceed at the rate of one per clock cycle.
- Some resource has not been duplicated enough to allow all combinations of instructions in the pipeline to execute.

Small Concept

Three problems are known as Pipelining Hazards:
(*i*) Structural Hazards (*ii*) Control Hazards (*iii*) Data Hazards

Control Hazards

Control hazards can cause a greater performance loss for DLX pipeline than data hazards. When a branch is executed, it may or may not change the PC (program counter) to something other than its current value plus 4. If a branch changes the PC to its target address, it is a taken branch; if it falls through, it is not taken.

There are many methods to deal with the pipeline stalls caused by branch delay. We discuss four simple compile-time schemes in which predictions are static - they are fixed for each branch during the entire execution, and the predictions are compile-time guesses.

Stall Pipeline: The simplest scheme to handle branches is to freeze or flush the pipeline, holding or deleting any instructions after the branch until the branch destination is known. Advantage: simple both to software and hardware.

Predict Taken: An alternative scheme is to predict the branch as taken. As soon as the branch is decoded and the target address is computed, we assume the branch to be taken and begin fetching and executing at the target address.

Predict Not Taken: A higher performance, and only slightly more complex, scheme is to predict the branch as not taken, simply allowing the hardware to continue as if the branch were not executed. Care must be taken not to change the machine state until the branch outcome is definitely known.

Delayed Branch: The execution cycle with a branch delay of one is:

- Branch instruction
- Sequential successor
- Branch target if take

Data Hazards

A major effect of pipelining is to change the relative timing of instructions by overlapping their execution.

This introduces data and control hazards. Data hazards occur when the pipeline changes the order of read/write accesses to operands so that the order differs from the order seen by sequentially executing instructions on the unpipelined machine.

- **Forwarding:** Reduces the effective pipeline latency so that the certain dependencies do not result in hazards.
- **Data Hazards Classification:** (Consider two instructions i and j, with i occurring before j.) The possible data hazards are:
  - **RAW (Read After Write):** j tries to read a source before i writes it, so j incorrectly gets the old value.
  - **WAW (Write After Write):** j tries to write an operand before it is written by i. The writes end up being performed in the wrong order, leaving the value written by i rather than the value written by j in the destination.
  - **WAR (Write After Read):** j tries to write a destination before it is read by i, so i incorrectly gets the new value.
- **Pipeline Scheduling:** Rearranging instructions in order to avoid hazards.

EXAMPLE 11.3.

Find the hazard in the following code:

$$lw\ \$t_0, 0(\$t_1)$$
$$lw\ \$t_2, 4(\$t_1)$$
$$sw\ \$t_2, 0(\$t_1)$$
$$sw\ \$t_0, 4(\$t_1)$$

Reorder to avoid pipeline stalls.

Solution: The hazard occurs on register $\$t_2$ between the second lw and the first *sw*.

Swapping the two sw instruction removes the hazard:

$$lw\ \$t_0, 0(\$t_1)$$
$$lw\ \$t_2, 4(\$t_1)$$
$$sw\ \$t_0, 4(\$t_1)$$
$$sw\ \$t_2, 0(\$t_1)$$

11.6. ADVANCED COMPILER TECHNOLOGY

The discussion above illustrates the need for more advanced compiler technology to detect potential parallelism and reschedule instructions to take maximum advantage. In this section we look briefly at some of the more advanced techniques that modern compilers use.

- Loop Unrolling
- Score boarding

11.7. PARALLEL ARCHITECTURE

One method of improving the performance of a processor is to decrease the time needed to execute instructions. This will work up to a limit of about 400 MHz (Stone, 1991), at which point an effect known as ringing on busses prohibits further speedup with conventional bus technology. This is not to say that higher clock speeds are not possible, because indeed current microprocessors have clock rates well above 400 MHz, but that "shared bus" approaches become impractical at these speeds. As conventional architectural approaches to improving performance wear thin, we need to consider alternative methods of improving performance.

One alternative approach to increasing bus speed is to increase the number of processors, and decompose and distribute a single program onto the processors. This approach is known as parallel processing, in which a number of processors work collectively, in parallel, on a common problem. We see an example of parallel processing earlier in the chapter with pipelining. For that case, four processors are connected in series (Figure 11-1), each performing a different task, like an assembly line in a factory. The interleaved memory described in Chapter 7 is another example of pipelining.

11.8. TAXONOMY OF PARALLEL ARCHITECTURES

The idea of using multiple processors both to increase performance and to improve availability dates back to the earliest electronic computers. About 40 years ago, Flynn proposed a simple model of categorizing all computers that is still useful today. Flynn's classification scheme is based on identifying two orthogonal streams in a computer. These are the instruction and the data streams. The instruction stream is defined as the sequence of instructions performed by the computer. The data stream is defined as the data traffic exchanged between the memory and the processing unit. According to Flynn's classification, either of the instruction or data streams can be single or multiple. This leads to four distinct categories of computer architectures:

1. Single-instruction single-data streams (SISD).
2. Single-instruction multiple-data streams (SIMD).
3. Multiple-instruction single-data streams (MISD).
4. Multiple-instruction multiple-data streams (MIMD)

Figure 11.5, shows the orthogonal organization of the streams according to Flynn's classification.

		Data Stream	
		Single	Multiple
Instruction Stream	Single	SISD	SIMD
	Multiple	MISD	MIMD

Fig. 11.5. Flynn's classification

1. **Single Instruction Stream, Single Data Stream (SISD):** This category is the uniprocessor. Figure 11.3 given below presents the architecture of SISD. Examples of this architecture are IBM 704, VAX 11/780, CRAY-1.

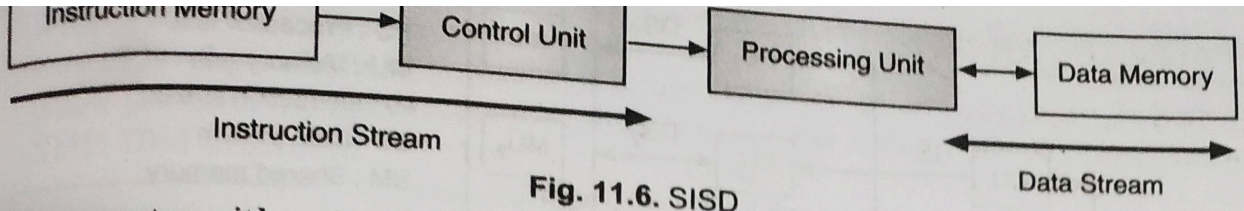


Fig. 11.6. SISD

A computer with a single processor is called a Single Instruction Stream, Single Data Stream (SISD) Computer. It represents the organization of a single computer containing a control unit, a processor unit, and a memory unit. Instructions are executed sequentially and the system may or may not have internal parallel processing. Parallel processing may be achieved by means of a pipeline processing.

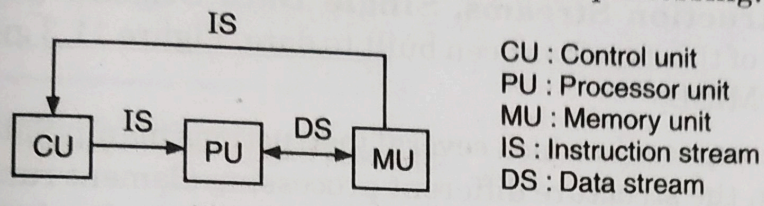


Fig. 11.7. SISD architecture

In such a computer a single stream of instructions and a single stream of data are accessed by the processing elements from the main memory, processed and the results are stored back in the main memory. SISD computer organization is shown in figure below.

2. **Single Instruction Stream, Multiple Data Streams (SIMD):** The same instruction is executed by multiple processors using different data streams. SIMD computers exploit data-level parallelism by applying the same operations to multiple items of data in parallel. Each processor has its own data memory (hence multiple data), but there is a single instruction memory and control processor, which fetches and dispatches instructions. For applications that display significant data-level parallelism, the SIMD approach can be very efficient. Figure 11.4 given below presents the architecture of SIMD. Examples of this architecture are ILLIAC-IV, MPP, CM-2, STARAN.

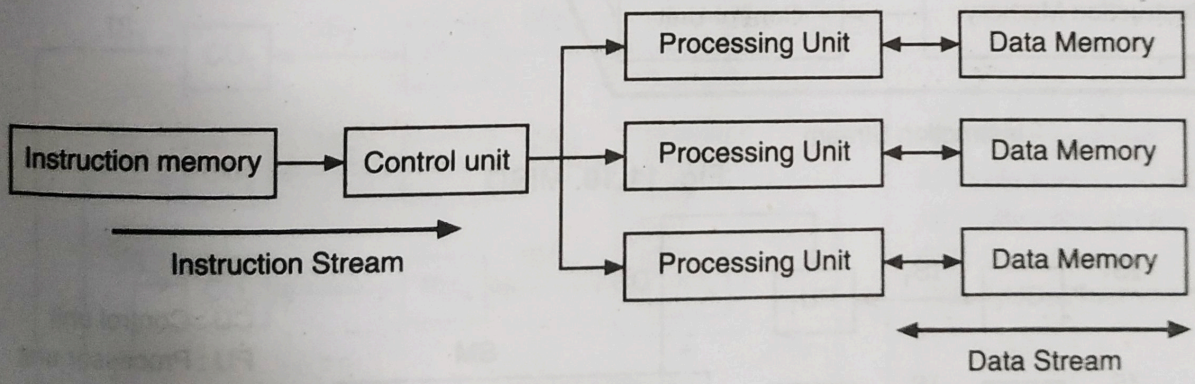


Fig. 11.8. SIMD

It represents an organization of computer which has multiple processors under the supervision of a common control unit. All processors receive the same instruction from the control unit but operate on different items of the data. SIMD computers are used to solve many problems in science which require identical operations to be applied to different data set synchronously. Examples are added a set of matrices simultaneously, such as $\sum_i \sum_k (a_{ik} + b_{ik})$. Such computers are known as array processors. SIMD computer organization is shown in figure below.

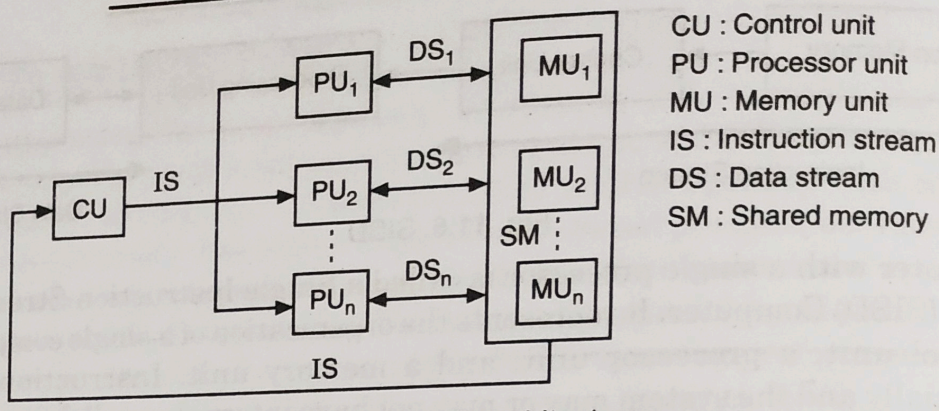


Fig. 11.9. SIMD architecture

3. **Multiple Instruction Streams, Single Data Stream (MISD):** No commercial multiprocessor of this type has been built to date. Figure 11.3 given below presents the architecture of MISD.

   It refers to the computer in which several instructions manipulate the same data stream concurrently. In the structure different processing element run different programs on the same data. This type of processor may be generalized using a 2- dimensional arrangement of processing elements. Such a structure is known as systolic processor. MISD computer organization is shown in figure below.

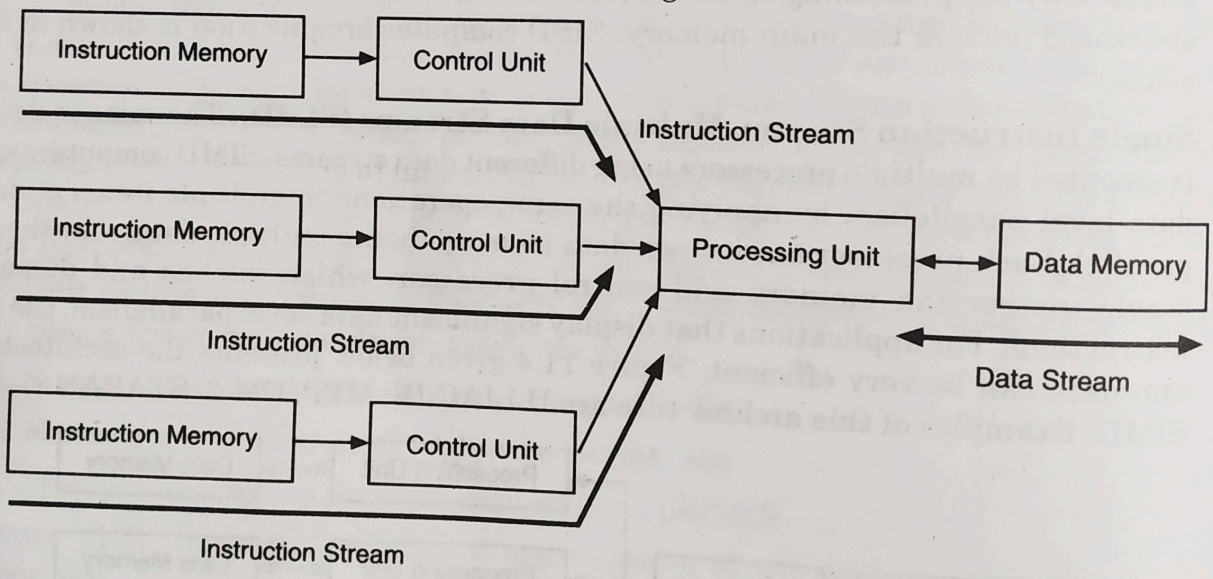


Fig. 11.10. MISD

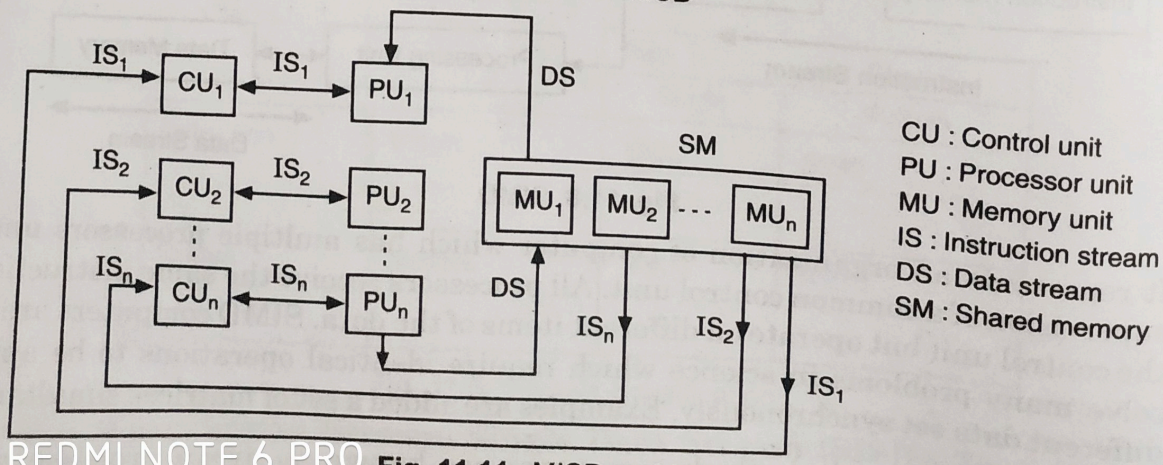


Fig. 11.11. MISD architecture

4. **Multiple Instruction Streams, Multiple Data Streams (MIMD):** Each processor fetches its own instructions and operates on its own data. MIMD computers exploit thread-level parallelism, since multiple threads operate in parallel. Figure 11.12 given below presents the architecture of MIMD. Examples of this architecture are Cm, CRAY XMP, IBM 370/168M.

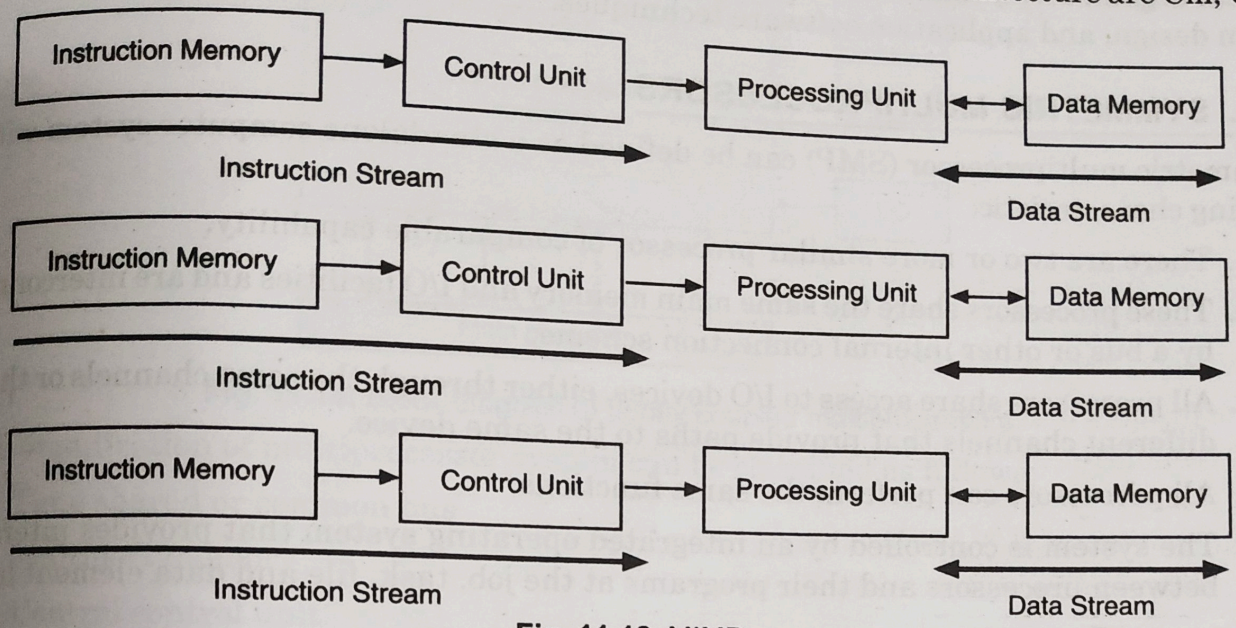


Fig. 11.12. MIMD

In general, thread-level parallelism is more flexible than data-level parallelism and thus more generally applicable. This is a coarse model, as some multiprocessors are hybrids of these categories. Nonetheless, it is useful to put a framework on the design space.

MIMD computers are the general purpose parallel computers. Its organization refers to a computer system capable of processing several programs at a same time. MIMD systems include all multiprocessing systems. MIMD computer organization is shown in figure below.

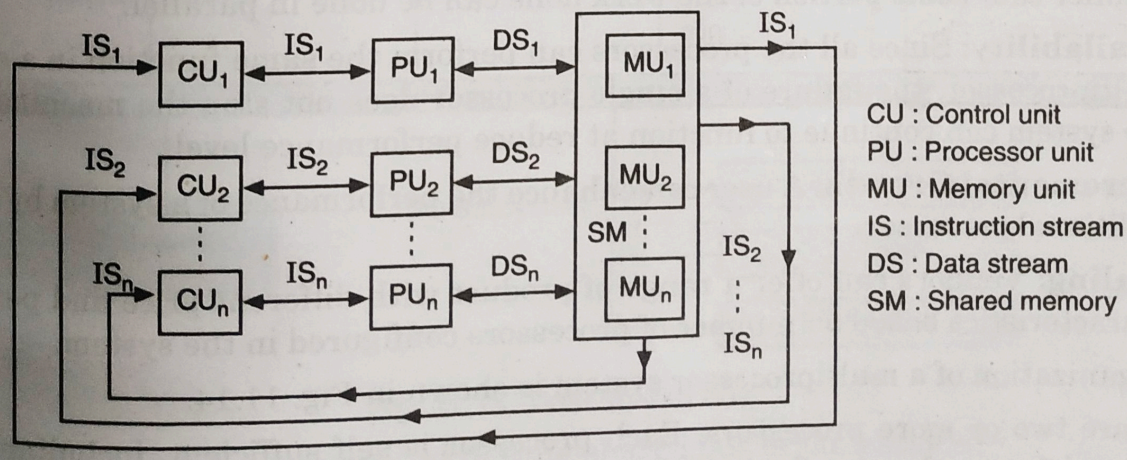


Fig. 11.13. MIMD architecture

Further MIMD can be subdivided into two main categories:

- **Symmetric Multiprocessor (SMP):** In an SMP, multiple processors share a single memory or a pool of memory by means of a shared bus or other interconnection mechanism. A distinguish feature is that the memory access time to any region of memory is approximately the same for each processor.

- **Nonuniform Memory Access (NUMA):** The memory access time to different regions of memory may differ for a NUMA processor.

The design issues relating to SMPs and NUMA are complex, involving issues relating to physical organization, interconnection structures, inter processor communication, operating system design, and application software techniques.

11.10. SYMMETRIC MULTIPROCESSORS

A symmetric multiprocessor (SMP) can be defined as a standalone computer system with the following characteristic:

1. There are two or more similar processor 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.
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.
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.

Small Concept

In the SMP, access time of memory is same for each processor.

The operating system of a SMP schedules processors or thread across all of the processors. SMP has a potential advantages over uniprocessor architecture:

- **Performance:** A system with multiple processors will perform in a better way than one with a single processor of the same type if the task can be organized in such a manner that some portion of the work done can be done in parallel.
- **Availability:** Since all the processors can perform the same function in a symmetric multiprocessor, the failure of a single processor does not stop the machine. Instead, the system can continue to function at reduce performance level.
- **Incremental Growth:** A user can enhance the performance of a system by adding an additional processor.
- **Sealing:** Vendors can offer a range of product with different price and performance characteristics based on number of processors configured in the system.

The organization of a multiprocessor system is shown in Fig. 11.14.

There are two or more processors. Each processor is self sufficient, including a control unit, ALU, registers and cache. Each processor has access to a shared main memory and the I/O devices through an interconnection network. The processor 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 signal 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.

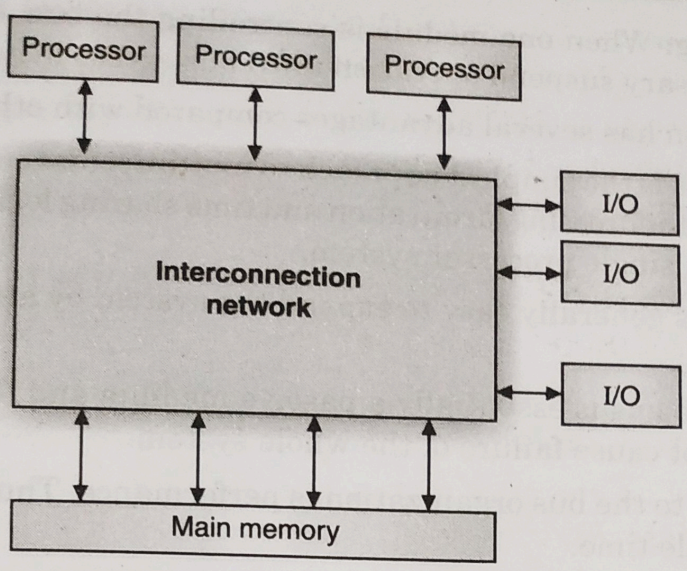


Fig. 11.14. Block diagram of tightly coupled multiprocessors

The organization of multiprocessor system can be classified as follows:

- Time shared or common bus
- Multiport memory
- Central control unit.

Time Shared Bus

Time shared bus is the simplest mechanism for constructing a multiprocessor system. The bus consists of control, address and data lines. The block diagram is shown in Fig. 11.15.

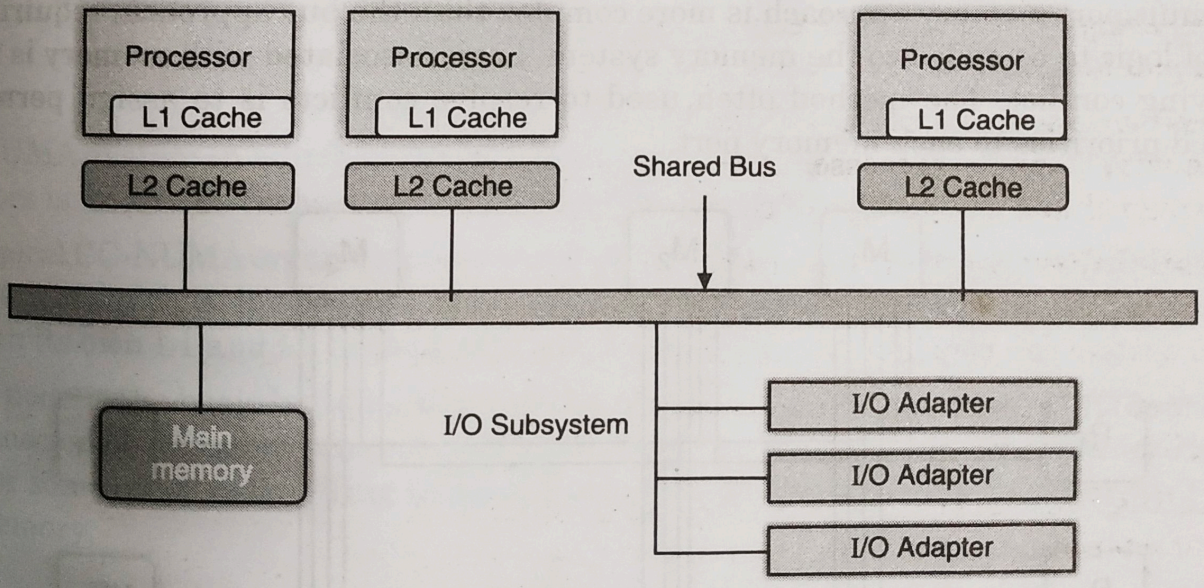


Fig. 11.15. Time shared bus

The following features are provided in time-shared bus organization:

- **Addressing:** It must be possible to distinguish modules on the bus to determine the source and destination of data.
- **Arbitration:** Any input/output module can temporarily function as "master". A mechanism is provided to arbitrate competing request for bus control, using some sort of priority scheme.

- **Time Shearing:** When one module is controlling the bus, other modules are locked out and if necessary suspend operation until bus access in achieved.

The bus organization has several advantages compared with other approaches:

- **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 processor 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. Thus, the speed of the system is limited by the bus cycle time.

To improve performance, each processor can be equipped with local cache memory.

The use of cache leads to a new problem which is known as cache coherence problem. Each local cache contains an image of a portion of main memory. If a word is altered in one cache, it may invalidate a word in another cache. To prevent this, the other processors must perform an update in its local cache.

Multiport Memory

The multiport memory approach allows the direct, independent access of main memory modules by each processor and io module.

The multiport memory system is shown in Fig. 11.16.

The multiport memory approach is more complex than the bus approach, requiring a fair amount of logic to be added to the memory system. Logic associated with memory is required for resolving conflict. The method often used to resolve conflicts is to assign permanently designated priorities to each memory port.

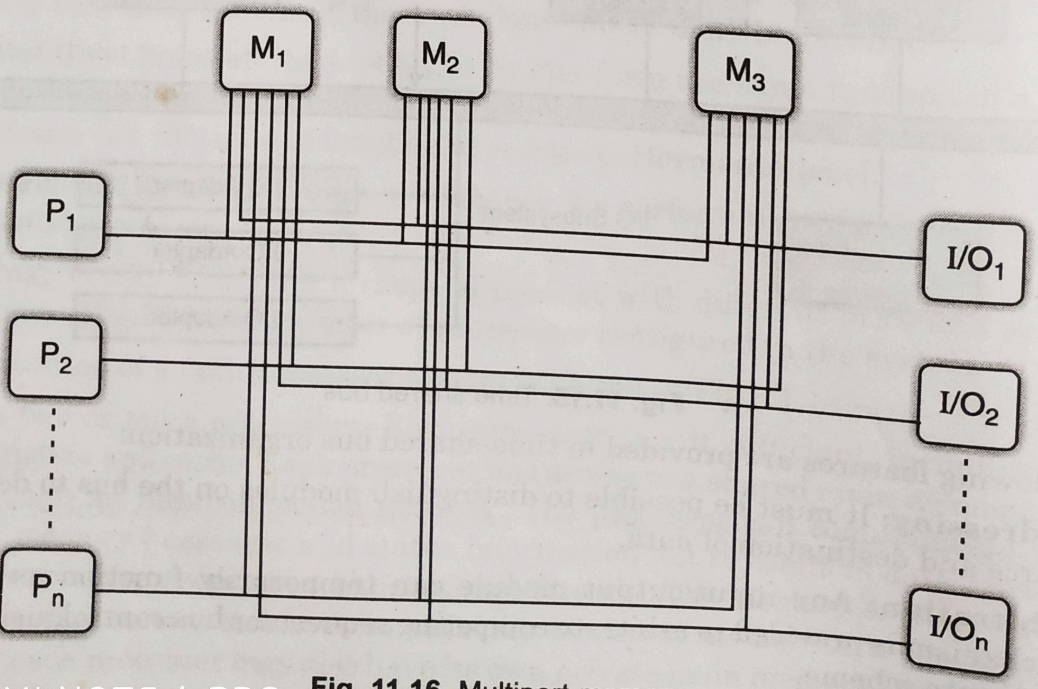


Fig. 11.16. Multiport memory

11.11. NON-UNIFORM MEMORY ACCESS (NUMA)

In NUMA architecture, all processors have access to all parts of main memory using loads and stores. The memory access time of a processor differs depending on which region of main memory is accessed. The last statement is true for all processors; however, for different processors, which memory regions are slower and which are faster differ.

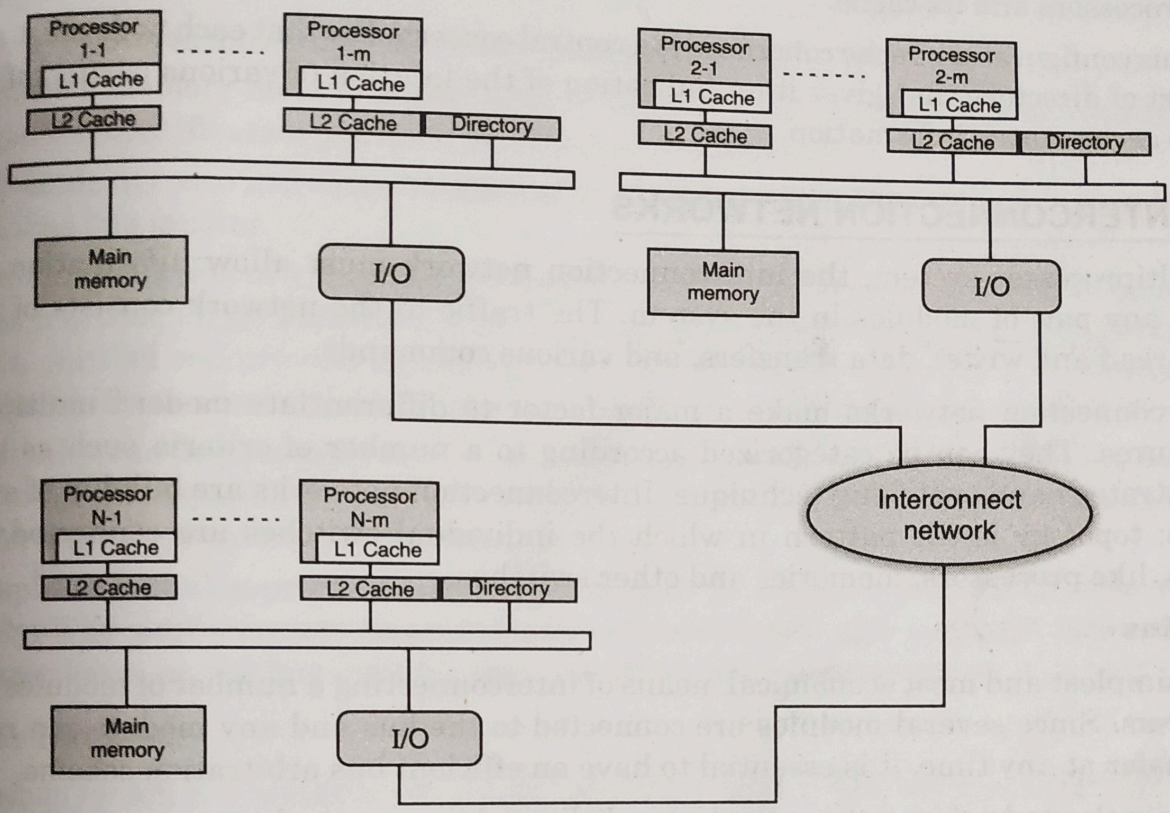


Fig. 11.17. CC-NUMA Organization

A NUMA system in which cache coherence is maintained among the cache of the various processors is known as cache-cohence NUMA (CC-NUMA)

A typical CC-NUMA organization is shown in the Fig. 11.27. There are multiple independent nodes, each of which is, in effect, an SMP organization. Each node contains multiple processors, each with its own L1 and L2 caches, plus main memory.

The node is the basic building block of the overall CC NUMA organization. The nodes are interconnected by means of some communication facility, which could be a switching mechanism, a ring, or some other networking facility. Each node in the CC-NUMA system includes some main memory.

Small Concept

The memory access time to different regions of memory may differ for NUMA processor.

From the point of view of the processors, there is only a single addressable memory, with each location having a unique system-wide address. When a processor initiates a memory access, if the requested memory location is not in the processors cache, then the L_2 cache

initiates a fetch operation. If the desired line is in the local portion of the main memory, the line is fetch across the local bus.

If the desired line is in a remote portion of the main memory, then an automatic request is send out to fetch that line across the interconnection network, deliver it to the local bus, and then deliver it to the requesting cache on that bus. All of this activity is atomic and transparent to the processors and its cache.

In this configuration, cache coherence is a central concern. For that each node must maintain some sort of directory that gives it an indication of the location of various portion of memory and also cache status information.

11.12. INTERCONNECTION NETWORKS

In a multiprocessor system, the interconnection network must allow information transfer between any pair of modules in the system. The traffic in the network consists of requests (such as read and write), data transfers, and various commands.

Interconnection networks make a major factor to differentiate modern multiprocessor architectures. They can be categorized according to a number of criteria such as topology, routing strategy and switching technique. Interconnection networks are bulidup of switching elements; topology is the pattern in which the individual switches are connected to other elements, like processors, memories and other switches.

Single Bus

The simplest and most economical means of interconnecting a number of modules is to use a single bus. Since several modules are connected to the bus and any module can request a data transfer at any time, it is essential to have an efficient bus arbitration scheme.

In a simple mode of operation, the bus is dedicated to a particular source-destination pair for the full duration of the requested transfer. For example, when a processor uses a read request on the bus, it holds the bus until it receives the desired data from the memory module.

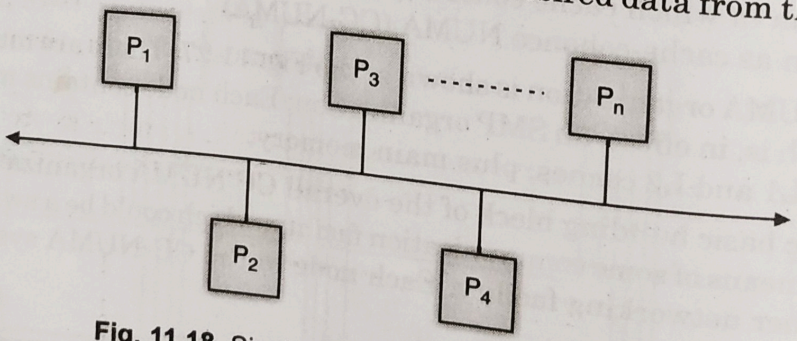


Fig. 11.18. Single bus interconnection network

Since the memory module needs a certain amount of time to access the data bus, the bus will be idle until the memory is ready to respond with the data. Then the data is transferred to the processors. When this transfer is completed, the bus can be assigned to handle another request.

A scheme known as the split- transaction protocol makes it possible to use the bus during the idle period to serve another request. Consider the following method of handling a series of read requests possibly from different processor.

After transferring the address involved in the first request, the bus may be reassigned to transfer the address of the second request; assuming that this request is to a different memory module.

At this point, we have two modules proceeding with read access cycle in parallel.

In split- transaction protocol, performance is improved at the cost of increased bus complexity.

There are two reasons why complexity increases:

- Since a memory module needs to know which source initiated a given read request, a source identification tag must be attached to the request.
- Complexity also increases because all modules, not just the processor, must be able to act as bus muster.

The main limitation of a single bus is that the number of modules that can be connected to the bus is not that large. Networks that allow multiple independent transfer operations to proceed in parallel can provide significantly increased data transfer rate. If neither module has finished with its access, the bus may be reassigned to a third request and so on.

Eventually, the first memory module completes its access cycle and uses the bus to transfer the data to the corresponding source. As other modules complete their cycles, the bus is needed to transfer their data to the corresponding sources.

The split transaction protocol allows the bus and the available bandwidth to be used more efficiently. The performance improvement achieved with this protocol depends on the relationship between the bus transfer time and the memory access time.

Crossbar Network

Crossbar switch is a versatile switching network. It is basically a network of switches. Any module can be connected to any other module by closing the appropriate switch. Such networks, where there is a direct link between all pairs of nodes are called fully connected networks.

In a fully connected network, many simultaneous transfers are possible. If n sources need to send data to n distinct destinations then all of these transfers can take place concurrently. Since no transfer is prevented by the lack of a communication path, the crossbar is called a nonblocking switch.

In the Fig. 11.19, of crossbar interconnection network, a single switch is shown at each cross point. In actual multiprocessor system, the paths through the crossbar network are much wider.

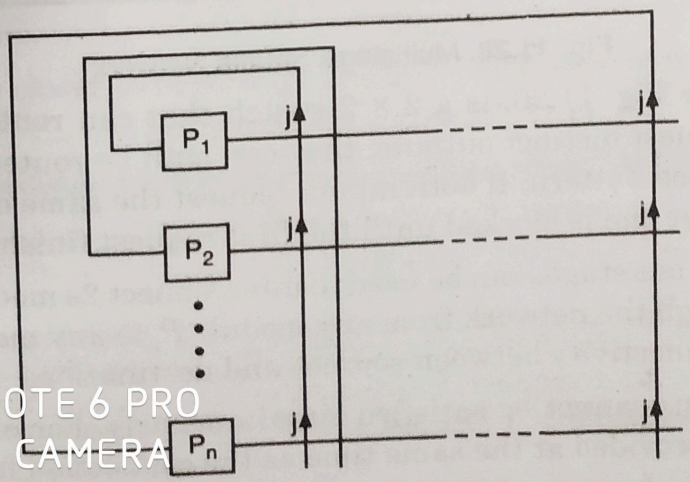


Fig. 11.19. Crossbar Interconnection Network

If there are modules in a network, than the number of cross point is in a network to interconnect modules. The total number of switches becomes large as increases. In a crossbar switch, conflicts occur when two or more concurrent requests are made to the same destination device. These conflicting requests are usually handled on a predetermined priority basis.

The crossbar switch has the potential for the highest bandwidth and system efficiency. However, because of its complexity and cost, it may be cost effective for a large multiprocessor system.

Multistage Network

The bus and crossbar systems use a single stage of switching to provide a path from a source to a destination. In multistage network, multiple stages of switches are used to setup a path between source and destination. Such networks are less costly than the crossbar structure, yet they provide a reasonably large number of parallel paths between source and destinations. In the Fig. 11.20, it shows a three-stage network that called a shuffle network that interconnects eight modules. The term "shuffle" describes the pattern of connections from the outputs of one stage to the inputs of the next stage.

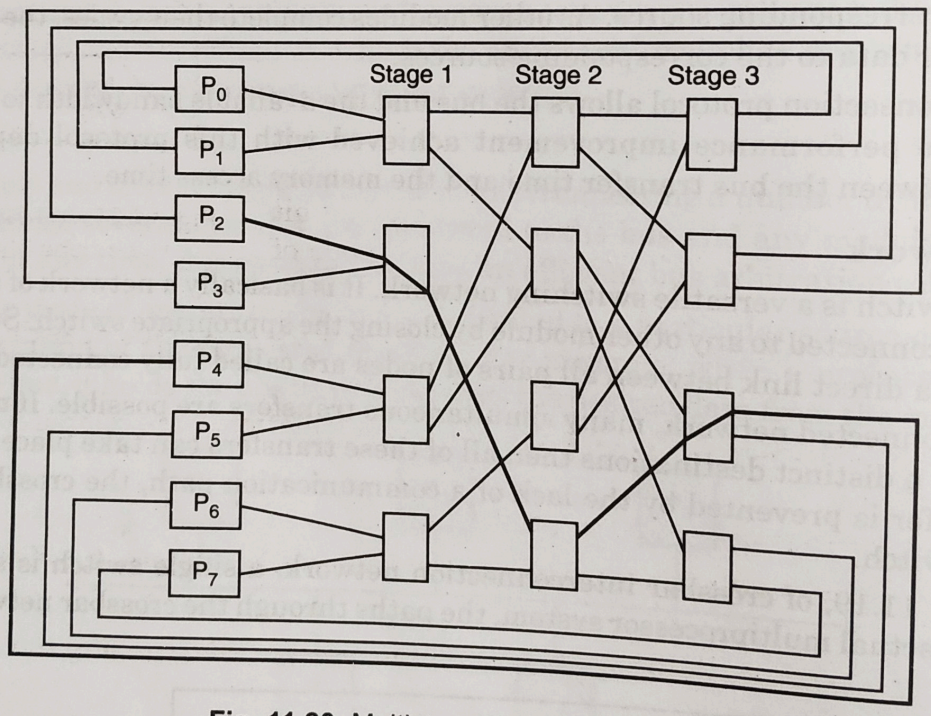


Fig. 11.20. Multistage Shuffle Network

The switchbox in the Fig. 11.20, is a 2 × 2 switch that can route either input to either output. If the inputs request distinct outputs, they can both be routed simultaneously in the straight through or crossed pattern. If both inputs request the same output, only one request can be satisfied. The other one is blocked until the first request finishes using the switch.

A network consisting of s stages can be used to interconnect 2s modules. In this case, there is exactly one path through the network from any module P_i to any module P_j. Therefore, this network provides full connectivity between sources and destinations.

Many request patterns cannot be satisfied simultaneously. For example, the connection from P_2 to P_7 can not be provided at the same time as the connection from P_3 to P_6.

A multistage network is less expansive to implement than a crossbar network. If n nodes are to be interconnected using this scheme, then we must use $s = \log_2 n$ stages with $n/2$ switches per stage. Since each switches contains four switches, the total number of switches is

$$4 \times \frac{n}{2} \times \log_2 n = 2n \times \log_2 n$$

which, for a large network, is considerably less than the n^2 switches needed in a crossbar network.

Multistage networks are less capable of providing concurrent connection than crossbar switches. The connection path between P_2 and P_4 is indicated by RED lines in the Fig. 11.20.

Hypercube Networks

A hypercube is an n-dimensional cube that interconnects $2n$ nodes. In addition to the communication circuit, each node usually includes a processor and a memory module as well as some input/output capability.

The Fig. 11.21, shows a three dimensional hypercube. The small circles represent the communication circuits in the nodes. The edge of the cube represent bi-directional communication links between neighbouring nodes.

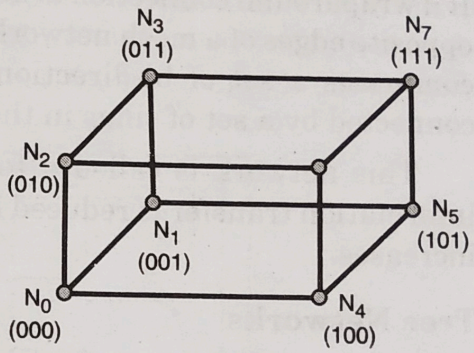


Fig. 11.21. A 3-dimensional Hypercube Network

In an an n-dimensional hypercube each node is directly connected to n neighbours. A useful way to label the nodes is to assign binary addresses to them in such a way that the addresses of any two neighbours differs in exactly one bit position. The functional units are attached to each node of the hypercube.

Routing messages through the hypercube is easy. If the processor at node N_i wishes to send a message to node N_j, it proceeds as follows:

- The binary addresses of the source, i, and the destination, j, are compared from least to most significant bits.
- Suppose that they differ first in position P.
- Node N_i then sends the message to its neighbour whose address, k, differs from i in bit position P.
- Node N_k forwards the message to the appropriate neighbour using the same address comparison scheme.
- The message gets closer to destination node N_j with each of these hops from one node to another.
- For example, a message from node N_0 to N_5 transverse the following way:

$$N_0 : 0\ 0\ 0$$
$$N_5 : 1\ 0\ 1$$

Message traverses from node N_0 to N_1, they differ in 1st bit position. Then message traverses from N_1 to N_5 hey differ in 3 rd bit position.

Therefore, it takes two hops. The maximum distance that any message needs to travel in an n-dimensional hypercube is n-hops.

Mesh Networks

Mesh network is another way to interconnect a large number of nodes in a multiprocessor system. An example of a mesh with 16 nodes is given in the Fig. 11.22.

The link between the nodes are bi-directional. The functional unit are attached to the each node of the mesh network.

Routing in a mesh network can be done in several ways. One of the simplest and most effective possibilities is to choose the path between a source node N_i and a destination node N_j such that the transfer first takes place in the horizontal directional from N_i towards N_j.

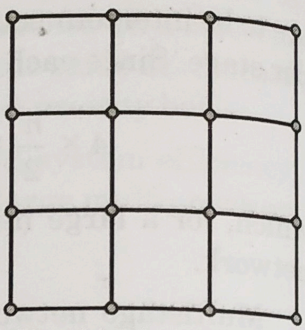


Fig. 11.22. (a) A 2-D Mesh Network

When the column in which N_j resides is reached, the transfer proceeds in the vertical direction along this column. If a wraparound connection is made between the nodes at the opposite edges of a mesh network, the result is a network that comprises a set of bi-directional rings in the X-direction connected by a set of rings in the Y-direction.

This network is called a torus. The average latency of information transfer is reduced in a torus, but the complexity increases.

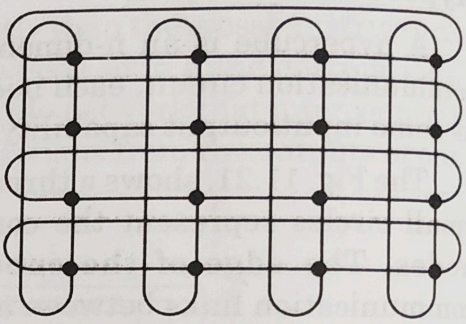


Fig. 11.22. (b) Torus network

Tree Networks

A hierarchically structured network implemented in the form of a tree is another interconnection topology. A four way tree that interconnects 16 modules is shown in the Fig. 11.23.

In this tree, each parent node allows communication between two of its children at a time. An intermediate-level node, for example node A in the figure, can provide a connection from one of its child node to its parent. This enables two leaf nodes that are any distance apart to communicate. Only one path at any time can be established through a given node in the tree.

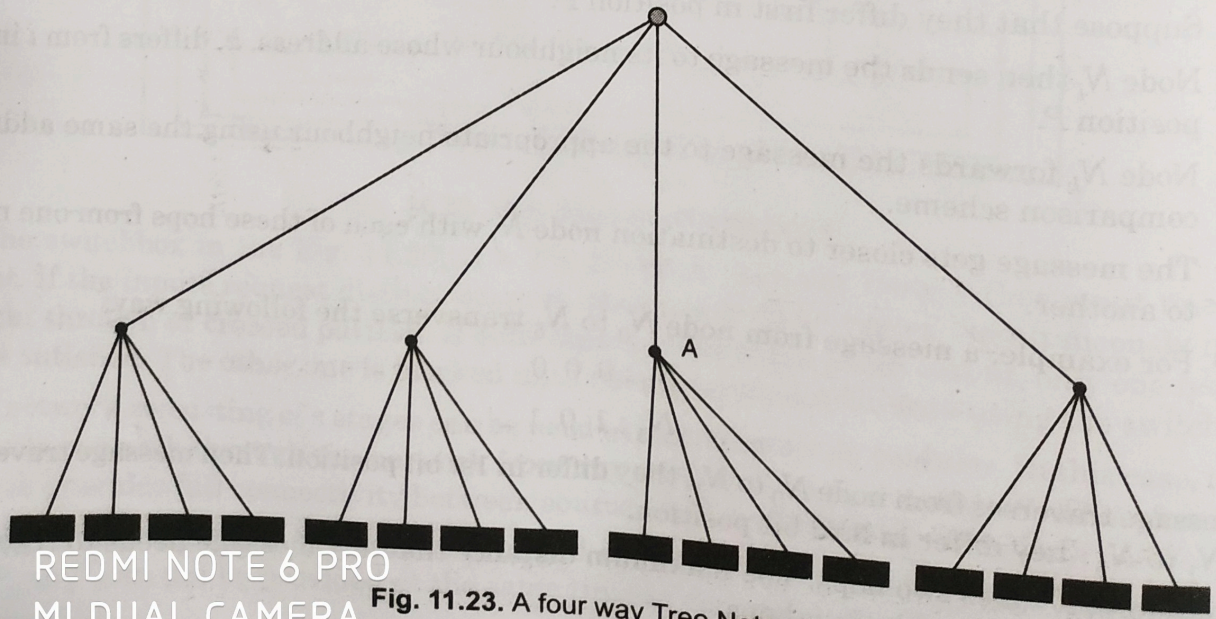


Fig. 11.23. A four way Tree Network

To reduce the possibility of a bottleneck, the number of links in the upper levels of a tree hierarchy can be increased. This is done in a fat tree network, in which each node in the tree (except at the top level) has more than one parent.

The Fig. 11.24, shows a fat tree in which each node has two parents.

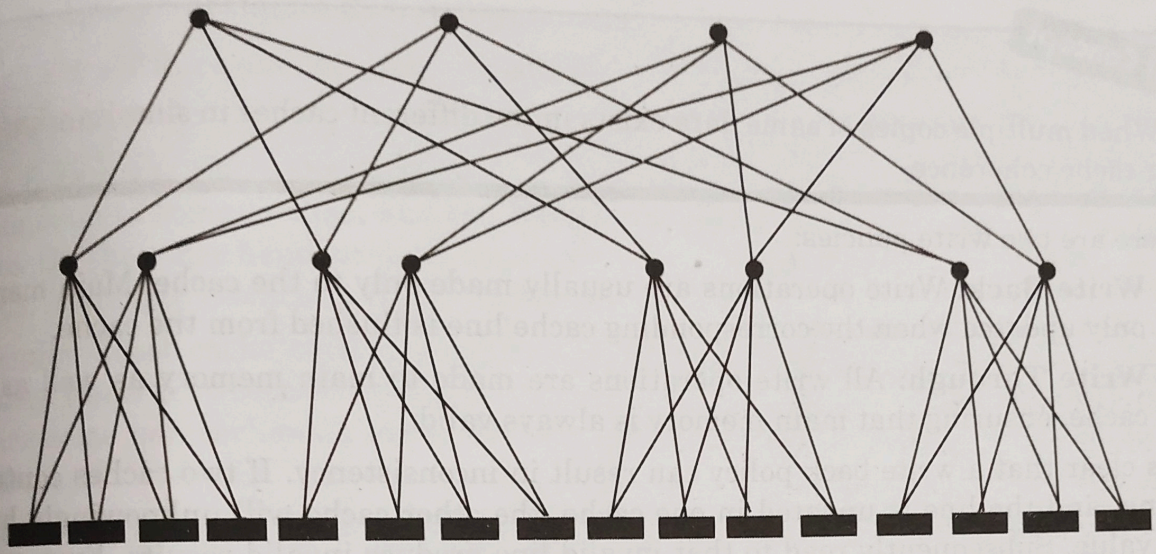
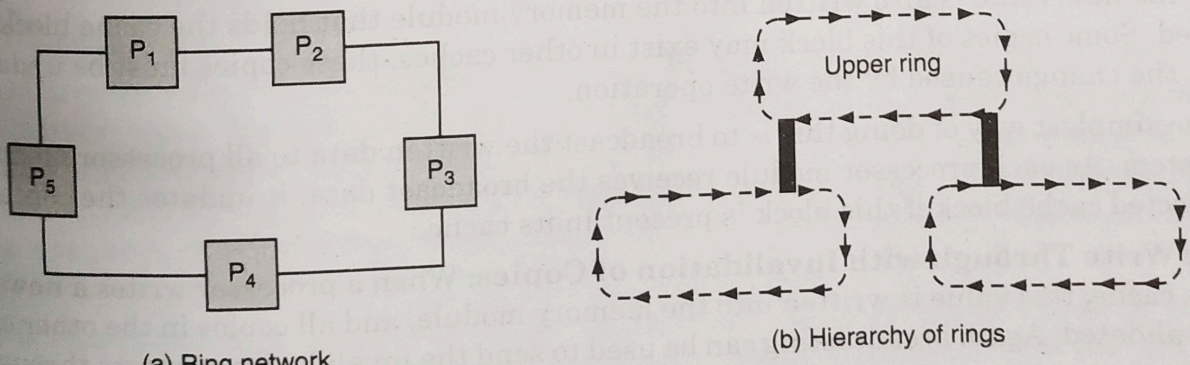


Fig. 11.24. A Fat Tree

Ring Networks

One of the simplest network topologies uses a ring to interconnect the nodes in the system. A single ring is shown in the Fig. 11.25(*a*). The main advantage of the arrangement is that the ring is easy to implement. Links in the ring can be wide, because each node is connected to only two neighbours. It is not useful to construct a very long ring to connect many nodes because the latency of information transfer would be unacceptably large

The simple possibility of using ring in a tree structure; this results in a hierarchy of rings. A hierarchy of rings is shown in the Fig. 11.25(*b*). Having short rings reduces substantially the latency of transfers that involve nodes on the same ring. The latency of transfers between two nodes on different rings is shorter than if a single ring were used.



(a) Ring network

(b) Hierarchy of rings

Fig. 11.25. Hierarchy of Rings

11.13. CACHE COHERENCE

In contemporary multiprocessor system, it is customary to have one or two levels of cache associated with each processor. This organization is essential to achieve high performance.

Cache creates a problem, which is known as the cache coherence problem. The cache coherence problem is: Multiple copies of the same data can exist in different caches simultaneously, and if processors are allowed to update their own copies freely, an inconsistent view of memory can result.

Small Concept

When multiple copies of same data exists in the different caches in simultaneously, its a cache coherence.

There are two write policies:

- **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.
- **Write Through:** All write 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 value. Subsequently read to that invalid line produce invalid results. Even with the write through policy, inconsistency can occur unless other cache monitor the memory traffic or receive some direct notification of the update.

For any cache coherence protocol, the objective is to let recently used local variables get into the appropriate cache and stay there through numerous reads and write, while using the protocol to maintain consistency of shared variables that might be in multiple caches at the same time.

Write Through Protocol: A write through protocol can be implemented in two fundamental versions.

(*a*) Write through with update protocol.

(*b*) Write through with invalidation of copies.

(*a*) Write Through with Update Protocol: When a processor writes a new value into its cache, the new value is also written into the memory module that holds the cache block being changed. Some copies of this block may exist in other caches, these copies must be updated to reflect the change caused by the write operation.

The simplest way of doing this is to broadcast the written data to all processor modules in the system. As each processor module receives the broadcast data, it updates the contents of the affected cache block if this block is present in its cache.

(*b*) Write Through with Invalidation of Copies: When a processor writes a new value into its cache, this value is written into the memory module, and all copies in the other caches are invalidated. Again broadcasting can be used to send the invalidation requests through the system.

Write Back Protocol: In the write-back protocol, multiple copies of a cache block may exist if different processors have loaded (read) the block into their caches. If some processor wants to change this block, it must first become an exclusive owner of this block. When the ownership is granted to this processor by the memory module that is the home location of the cl.. all other copies, including the one in the memory module, are invalidated.

Now the owner of the block may change the contents of the memory. When another processor wishes to read this block, the data are sent to this processor by the current owner. The data are also sent to the home memory module, which requires ownership and updates the block to contain the latest value.

11.13.1. Solutions of Cache Coherence

There are software and hardware solutions for cache coherence problem.

Software Solution

In software approach, the detecting of potential cache coherence problem 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 make conservative decisions, leading to inefficient cache utilization.

Compiler-based cache coherence mechanism perform an analysis on the code to determine which data items may become unsafe for caching, and they mark those items accordingly. So, there are some non cacheable items, and the operating system or hardware does not cache those items.

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 then 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 Solution

Hardware solution provides 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 performances over a software approach.

Hardware schemes can be divided into two categories:

(*a*) Directory protocol and

(*b*) Snoopy protocols.

(*a*) 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 themselves.

It is also responsible for keeping the state information up to date, therefore, every local action that can effect the global state of a line must be reported to the central controller. 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 time, forcing each processors to invalidate its copy.

After receiving acknowledgement 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 processors, it will send a miss notification to the controller. The controller then issues a command to the processor holding that line that requires the processors to do a write back to main memory.

Directory schemes suffer from the drawbacks of a **central bottleneck** and the **overhead of communication** between the various cache controllers and the central controller.

(*b*) Snoopy Protocols: Snoopy protocols distribute the responsibility for maintaining cache coherence among all of the cache controllers in a multiprocessor system. 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 observed these broadcasted notification 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.

Two basic approaches to the snoopy protocol have been explored: Write invalidates or write-update (write-broadcast).

(*i*) **Write Invalidates:** With a write-invalidate protocol, there can be multiple readers but only one write 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 which invalidates the line in the other caches, making the line exclusive to the writing cache. Once the line is exclusive, the owning processor can make local writes until some other processor requires the same line.

(*ii*) **Write Update (Write Broadcast):** With a write update protocol, there can be multiple writers as well as multiple readers. When a processors wishes to update a shared line, the word to be updated is distributed to all others, and caches containing that line can update it.

POINTS TO REMEMBER

- The use of multiple buses and direct paths in the processor data path enables us to reduce the number of steps required to fetch and execute instructions. This chapter discussed how to reduce the time of instruction cycle using parallel processing and pipelining.
- Pipelining is an implementation technique whereby multiple instructions are overlapped in execution; it takes advantage of parallelism that exists among the actions needed to execute an instruction.
- Pipeline divides the execution of an instruction among a number of subunits (stages), each perform part of the required operations. A possible division is to consider instruction fetch (F), instruction decode (D), operand fetch (F), instruction execution (E), and store of results (S) as the subtasks needed for the execution of an instruction.
- This leads to four distinct categories of parallel computer architectures:
  (*i*) Single-instruction single-data streams (SISD)
  (*ii*) Single-instruction multiple-data streams (SIMD)
  (*iii*) Multiple-instruction single-data streams (MISD)
  (*iv*) Multiple-instruction multiple-data streams (MIMD)