UNIT 1 INTRODUCTION TO PARALLEL COMPUTING

Structure **Page Nos.**

1.0 INTRODUCTION

Parallel computing has been a subject of interest in the computing community over the last few decades. Ever-growing size of databases and increasing complexity of the new problems are putting great stress on the even the super-fast modern single processor computers. Now the entire computer science community all over the world is looking for some computing environment where current computational capacity can be enhanced by a factor in order of thousands. The most obvious solution is the introduction of multiple processors working in tandem i.e. the introduction of parallel computing.

Parallel computing is the simultaneous execution of the same task, split into subtasks, on multiple processors in order to obtain results faster. The idea is based on the fact that the process of solving a problem can usually be divided into smaller tasks, which may be solved out simultaneously with some coordination mechanisms. Before going into the details of parallel computing, we shall discuss some basic concepts frequently used in parallel computing. Then we shall explain why we require parallel computing and what the levels of parallel processing are. We shall see how flow of data occurs in parallel processing. We shall conclude this unit with a discussion of role the of parallel processing in some fields like science and engineering, database queries and artificial intelligence.

1.1 OBJECTIVES

After going through this unit, you will be able to:

- tell historical facts of parallel computing;
- explain the basic concepts of the discipline, e.g., of program, process, thread, concurrent execution, parallel execution and granularity;
- explain the need of parallel computing;
- describe the levels of parallel processing;
- explain the basic concepts of dataflow computing, and
- describe various applications of parallel computing;

1.2 HISTORY OF PARALLEL COMPUTERS

The experiments with and implementations of the use of parallelism started long back in the 1950s by the IBM. The IBM STRETCH computers also known as IBM 7030 were built in 1959. In the design of these computers, a number of new concepts like overlapping I/O with processing and instruction look ahead were introduced. A serious approach towards designing parallel computers was started with the development of ILLIAC IV in 1964 at the University of Illionis. It had a single control unit but multiple processing elements. On this machine, at one time, a single operation is executed on different data items by different processing elements. The concept of pipelining was introduced in computer CDC 7600 in 1969. It used pipelined arithmatic unit. In the years 1970 to 1985, the research in this area was focused on the development of vector super computer. In 1976, the CRAY1 was developed by Seymour Cray. Cray1 was a pioneering effort in the development of vector registers. It accessed main memory only for load and store operations. Cray1 did not use virtual memory, and optimized pipelined arithmetic unit. Cray1 had clock speed of 12.5 n.sec. The Cray1 processor evloved upto a speed of 12.5 Mflops on 100 × 100 linear equation solutions. The next generation of Cray called Cray XMP was developed in the years 1982-84. It was coupled with 8-vector supercomputers and used a shared memory.

Apart from Cray, the giant company manufacturing parallel computers,Control Data Corporation (CDC) of USA, produced supercomputers, the CDC 7600. Its vector supercomputers called Cyber 205 had memory to memory architecture, that is, input vector operants were streamed from the main memory to the vector arithmetic unit and the results were stored back in the main memory. The advantage of this architecture was that it did not limit the size of vector operands. The disadvantage was that it required a very high speed memory so that there would be no speed mismatch between vector arithmetic units and main memory. Manufacturing such high speed memory is very costly. The clock speed of Cyber 205 was 20 n.sec.

In the 1980s Japan also started manufacturing high performance vector supercomputers. Companies like NEC, Fujitsu and Hitachi were the main manufacturers. Hitachi developed S-810/210 and S-810/10 vector supercomputers in 1982. NEC developed SX-1 and Fujitsu developed VP-200. All these machines used semiconductor technologies to achieve speeds at par with Cray and Cyber. But their operating system and vectorisers were poorer than those of American companies.

1.3 PROBLEM SOLVING IN PARALLEL

This section discusses how a given task can be broken into smaller subtasks and how subtasks can be solved in parallel. However, it is essential to note that there are certain applications which are inherently sequential and if for such applications, a parallel computer is used then the performance does not improve.

1.3.1 Concept of Temporal Parallelism

In order to explain what is meant by parallelism inherent in the solution of a problem, let us discuss an example of submission of electricity bills. Suppose there are 10000 residents in a locality and they are supposed to submit their electricity bills in one office.

Let us assume the steps to submit the bill are as follows:

1) Go to the appropriate counter to take the form to submit the bill.

2) Submit the filled form along with cash.

3) Get the receipt of submitted bill.

Assume that there is only one counter with just single office person performing all the tasks of giving application forms, accepting the forms, counting the cash, returning the cash if the need be, and giving the receipts.

This situation is an example of sequential execution. Let us the approximate time taken by various of events be as follows:

Giving application form = 5 seconds
Accepting filled application form and counting the cash and returning, if required = 5mnts, i.e., 5 ×60= 300 sec.
Giving receipts = 5 seconds.
Total time taken in processing one bill = 5+300+5 = 310 seconds.

Now, if we have 3 persons sitting at three different counters with

i) One person giving the bill submission form

ii) One person accepting the cash and returning,if necessary and

iii) One person giving the receipt.

The time required to process one bill will be 300 seconds because the first and third activity will overlap with the second activity which takes 300 sec. whereas the first and last activity take only 10 secs each. This is an example of a parallel processing method as here 3 persons work in parallel. As three persons work in the same time, it is called temporal parallelism. However, this is a poor example of parallelism in the sense that one of the actions i.e., the second action takes 30 times of the time taken by each of the other two actions. The word 'temporal' means 'pertaining to time'. Here, a task is broken into many subtasks, and those subtasks are executed simultaneously in the time domain. In terms of computing application it can be said that parallel computing is possible, if it is possible, to break the computation or problem in to identical independent computation. Ideally, for parallel processing, the task should be divisible into a number of activities, each of which take roughly same amount of time as other activities.

1.3.2 Concept of Data Parallelism

consider the situation where the same problem of submission of 'electricity bill' is handled as follows:
Again, three are counters. However, now every counter handles all the tasks of a resident in respect of submission of his/her bill. Again, we assuming that time required to submit one bill form is the same as earlier, i.e., 5+300+5=310 sec.

We assume all the counters operate simultaneously and each person at a counter takes 310 seconds to process one bill. Then, time taken to process all the 10,000 bills will be $310 \times (9999/3) + 310 \times 1\text{sec}$.
This time is comparatively much less as compared to time taken in the earlier situations, viz. 3100000 sec. and 3000000 sec respectively.

The situation discussed here is the concept of data parallelism. In data parallelism, the complete set of data is divided into multiple blocks and operations on the blocks are applied parallely. As is clear from this example, data parallelism is faster as compared to earlier situations. Here, no synchronisation is required between counters(or processers). It is more tolerant of faults. The working of one person does not effect the other. There is no communication required between processors. Thus, interprocessor communication is less. Data parallelism has certain disadvantages. These are as follows:

i) The task to be performed by each processor is predecided i.e., asssignment of load is static.

ii) It should be possible to break the input task into mutally exclusive tasks. In the given example, space would be required counters. This requires multiple hardware which may be costly.

The estimation of speedup achieved by using the above type of parallel processing is as follows:

Let the number of jobs = m
Let the time to do a job = p

If each job is divided into k tasks,

Assuming task is ideally divisible into activities, as mentioned above then,

Time to complete one task = p/k
Time to complete n jobs without parallel processing = n.p
Time to complete n jobs with parallel processing = $\frac{n * p}{k}$

$$\text{Speed up} \quad \frac{\text{time to complete the task if parallelism is not used}}{\text{time to complete the task if parallelism is used}}$$

$$= \frac{np}{n\frac{p}{k}}$$

$$= k$$

1.4 PERFORMANCE EVALUATION

In this section, we discuss the primary attributes used to measure the performance of a computer system. Unit 2 of block 3 is completely devoted to performance evaluation of parallel computer systems. The performance attributes are:

i) **Cycle time(T):** It is the unit of time for all the operations of a computer system. It is the inverse of clock rate (l/f). The cycle time is represented in n sec.

ii) **Cycles Per Instruction(CPI):** Different instructions takes different number of cycles for exection. CPI is measurement of number of cycles per instruction

iii) **Instruction count(I_c):** Number of instruction in a program is called instruction count. If we assume that all instructions have same number of cycles, then the total execution time of a program
= number of instruction in the program*number of cycle required by one instruction *time of one cycle.

Hence, execution time T=I_c*CPI*Tsec.

Practically the clock frequency of the system is specified in MHz. Also, the processor speed is measured in terms of million instructions per sec(MIPS).

1.5 SOME ELEMENTARY CONCEPTS

In this section, we shall give a brief introduction to the basic concepts like, program, process, thread, concurrency and granularity.

1.5.1 The Concept of Program

From the programmer's perspective, roughly a program is a
well-defined set of instructions, written in some programming language, with defined sets of inputs and outputs. From the operating systems perspective, a program is an executable file stored in a secondary memory. Software may consist of a single program or a number of programs. However, a program does nothing unless its instructions are executed by the processor. Thus a program is a passive entity.

1.5.2 The Concept of Process

Informally, a process is a program in execution, after the program has been loaded in the main memory. However, a process is more than just a program code. A process has its own address space, value of program counter, return addresses, temporary variables, file handles, security attributes, threads, etc.

Each process has a life cycle, which consists of creation, execution and termination phases. A process may create several new processes, which in turn may also create a new process. In UNIX operating system environment, a new process is created by *fork* system call. Process creation requires the following four actions:

i) **Setting up the process description:** Setting up the process description requires the creation of a *Process Control Block* (PCB). A PCB contains basic data such as process identification number, owner, process status, description of the allocated address space and other implementation dependent process specific information needed for process management.
ii) **Allocating an address space:** There are two ways to allocate address space to processes: sharing the address space among the created processes or allocating separate space to each process.
iii) **Loading the program into the allocated address space:** The executable program file is loaded into the allocated memory space.
iv) **Passing the process description to the process scheduler:** once, the three steps of process creation as mentioned above are completed, the information gatherd through the above-mentioned steps is sent to the process scheduler which allocates processor(s) resources to various competing to-be-executed processses queue.

The process execution phase is controlled by the process scheduler. Process scheduling may be per process or per thread. The process scheduling involves three concepts: process state, state transition and scheduling policy.

A process may be in one of the following states:

- **New:** The process is being created.
- **Running:** The process is being executed on a single processor or multiple processors.
- **Waiting:** The process is waiting for some event to occur.
- **Ready:** The process is ready to be executed if a processor is available.
- **Terminated:** The process has finished execution.

At any time, a process may be in any one of the above mentioned states. As soon as the process is admitted into job queue, it goes into ready state. When the process scheduler dispatches the process, its state becomes running. If the process is completely executed then it is terminated and we say that it is in terminated state. However, the process may return to ready state due to some interrupts or may go to waiting state due to some I/O activity. When I/O activity is over it may go to ready state. The state transition diagram is shown in *Figure 1*:

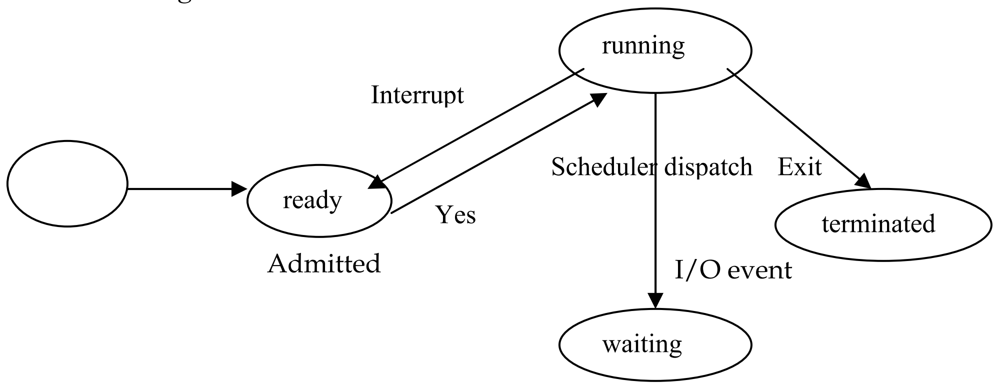


Figure1: Process state transition diagram

The scheduling policy may be either pre-emptive or non pre-emptive. In pre-emptive policy, the process may be interrupted. Operating systems have different scheduling policies. For example, to select a process to be executed, one of the secheduling policy may be: First In First Out(FIFO).

When the process finishes execution it is terminated by system calls like *abort*, releasing all the allocated resources.

1.5.3 The Concept of Thread

Thread is a sequential flow of control within a process. A process can contain one or more threads. Threads have their own program counter and register values, but they are share the memory space and other resources of the process. Each process starts with a single thread. During the execution other threads may be created as and when required. Like processes, each thread has an execution state (running, ready, blocked or terminated). A thread has access to the memory address space and resources of its process. Threads have similar life cycles as the processes do. A single processor system can support concurrency by switching execution between two or more threads. A multi-processor system can support parallel concurrency by executing a separate thread on each processor. There are three basic methods in concurrent programming languages for creating and terminating threads:

- **Unsynchronised creation and termination:** In this method threads are created and terminated using library functions such as CREATE_PROCESS, START_PROCESS, CREATE_THREAD, and START_THREAD. As a result of these function calls a new process or thread is created and starts running independent of its parents.

- **Unsynchronised creation and synchronized termination:** This method uses two instructions: FORK and JOIN. The FORK instruction creates a new process or thread. When the parent needs the child's (process or thread) result, it calls JOIN instruction. At this junction two threads (processes) are synchronised.

- **Synchronised creation and termination:** The most frequently system construct to implement synchronization is

COBEGIN…COEND. The threads between the COBEGIN…COEND construct are executed in parallel. The termination of parent-child is suspended until all child threads are terminated.

We can think of a thread as basically a lightweight process. However, threads offer some advantages over processes. The advantages are:

i) It takes less time to create and terminate a new thread than to create, and terminate a process. The reason being that a newly created thread uses the current process address space.

ii) It takes less time to switch between two threads within the same process, partly because the newly created thread uses the current process address space.

iii) Less communication overheads -- communicating between the threads of one process is simple because the threads share among other entities the address space. So, data produced by one thread is immediately available to all the other threads.

1.5.4 The Concept of Concurrent and Parallel Execution

Real world systems are naturally concurrent, and computer science is about modeling the real world. Examples of real world systems which require concurrency are railway networks and machines in a factory. In the computer world, many new operating systems support concurrency. While working on our personal computers, we may download a file, listen to streaming audio, have a clock running , print something and type in a text editor. A multiprocessor or a distributed computer system can better exploit the inherent concurrency in problem solutions than a uniprocessor system. Concurrency is achieved either by creating simultaneous processes or by creating threads within a process. Whichever of these methods is used, it requires a lot of effort to synchronise the processses/threads to avoid race conditions, deadlocks and starvations.

Study of concurrent and parallel executions is important due to following reasons:

i) Some problems are most naturally solved by using a set of co-operating processes.
ii) To reduce the execution time.

The words "concurrent" and "parallel" are often used interchangeably, however they are distinct.

Concurrent execution is the temporal behaviour of the N-client 1-server model where only one client is served at any given moment. It has dual nature; it is sequential in a small time scale, but simultaneous in a large time scale. In our context, a processor works as server and process or thread works as client. Examples of concurrent languages include Adam, concurrent Pascal, Modula-2 and concurrent PROLOG).
Parallel execution is associated with the N-client N-server model. It allows the servicing of more than one client at the same time as the number of servers is more than one. Examples of parallel languages includes Occam-2, Parallel C and strand-88.

Parallel execution does not need explicit concurrency in the language. Parallelism can be achieved by the underlying hardware. Similarly, we can have concurrency in a language without parallel execution. This is the case when a program is executed on a single processor.

1.5.5 Granularity

Granularity refers to the amount of computation done in parallel relative to the size of the whole program. In parallel computing, granularity is a qualitative measure of the ratio of computation to communication. According to granularity of the system, parallel-processing systems can be divided into two groups: fine-grain systems and coarse-grain systems. In fine-grained systems, parallel parts are relatively small and that means more frequent communication. They have low computation to communication ratio and require high communication overhead. In coarse-grained systems parallel parts are relatively large and that means more computation and less communication. If granularity is too fine it is possible that the overhead required for communications and synchronization between tasks takes longer than the computation. On the other hand, in coarse-grain parallel systems, relatively large amount of computational work is done. They have high computation to communication ratio and imply more opportunity for performance increase.

The extent of granularity in a system is determined by the algorithm applied and the hardware environment in which it runs. On an architecturally neutral system, the granularity does affect the performance of the resulting program. The communication of data required to start a large process may take a considerable amount of time. On the other hand, a large process will often have less communication to do during processing. A process may need only a small amount of data to get going, but may need to receive more data to continue processing, or may need to do a lot of communication with other processes in order to perform its processing. In most cases the overhead associated with communications and synchronization is high relative to execution speed so it is advantageous to have coarse granularity.

1.5.6 Potential of Parallelism

Problems in the real world vary in respect of the degree of inherent parallelism inherent in the respective problem domain. Some problems may be easily parallelized. On the other hand, there are some inherent sequential problems (for example computation of Fibonacci sequence) whose parallelization is nearly impossible. The extent of parallelism may be improved by appropriate design of an algorithm to solve the problem consideration. If processes don't share address space and we could eliminate data dependency among instructions, we can achieve higher level of parallelism. The concept of speed up is used as a measure of the *speed up* that indicates up to what extent to which a sequential program can be parallelised. Speed up may be taken as a sort of degree of inherent parallelism in a program. In this respect, Amdahl, has given a law, known as Amdahl's Law, which states that potential program speedup is defined by the fraction of code (P) that can be parallelised:

$$\text{Speed up} = \frac{1}{1-P}$$

If no part of the code can be parallelized, P = 0 and the speedup = 1 i.e. it is an inherently sequential program. If all of the code is parallelized, P = 1, the speedup is infinite. But practically, the code in no program can made 100% parallel. Hence speed up can never be infinite.

If 50% of the code can be parallelized, maximum speedup = 2, meaning the code will run twice as fast.

If we introduce the number of processors performing the parallel fraction of work, the relationship can be modeled by:

$$\text{Speed up} = \frac{1}{{}^{P}\!/_{N} + S}$$

Where P = parallel fraction, N = number of processors and S = serial fraction.

The *Table 1* shows the value of speed up for different values N and P.

Table 1

N	Speedup P = .50	Speedup P = .90	Speedup P = .99
10	1.82	5.26	9.17
100	1.98	9.17	50.25
1000	1.99	9.91	90.99
10000	1.99	9.91	99.02

The *Table 1* suggests that speed up increases as P increases. However, after a certain limits N does not have much impact on the value of speed up. The reason being that, for N processors to remain active, the code should be, in some way or other, be divisible in roughly N parts, independent part, each part taking almost same amount of time.

Check Your Progress 1

1) Explain the life cycle of a process.

……………………………………………………………………………………………
……………………………………………………………………………………………
……………………………………………………………………………………………
………………………………………………………………………………………

2) What are the advantages of threads over processes?

……………………………………………………………………………………………
……………………………………………………………………………………………
……………………………………………………………………………………………
………………………………………………………………………………………

3) Differentiate concurrent and parallel executions.

……………………………………………………………………………………………
……………………………………………………………………………………………
……………………………………………………………………………………………
………………………………………………………………………………………

4) What do understand by the granularity of a parallel system?

……………………………………………………………………………………………
……………………………………………………………………………………………
……………………………………………………………………………………………
…………………………………………………………………………………………..

1.6 THE NEED OF PARALLEL COMPUTATION

With the progress of computer science, computational speed of the processors has also increased many a time. However, there are certain constraints as we go upwards and face large complex problems. So we have to look for alternatives. The answer lies in parallel computing. There are two primary reasons for using parallel computing: save time and solve larger problems. It is obvious that with the increase in number of processors working in parallel, computation time is bound to reduce. Also, they're some scientific problems that even the fastest processor to takes months or even years to solve. However, with the application of parallel computing these problems may be solved in a few hours. Other reasons to adopt parallel computing are:

i) **Cost savings:** We can use multiple cheap computing resources instead of paying
ii) heavily for a supercomputer.

iii) **Overcoming memory constraints:** Single computers have very finite memory resources. For large problems, using the memories of multiple computers may overcome this obstacle. So if we combine the memory resources of multiple computers then we can easily fulfill the memory requirements of the large-size problems.

iv) **Limits to serial computing:** Both physical and practical factors pose significant constraints to simply building ever faster serial computers. The speed of a serial computer is directly dependent upon how fast data can move through hardware. Absolute limits are the speed of light ($3*10^8$ m/sec) and the transmission limit of copper wire ($9*10^8$ m/sec). Increasing speeds necessitate increasing proximity of processing elements. Secondly, processor technology is allowing an increasing number of transistors to be placed on a chip. However, even with molecular or atomic-level components, a limit will be reached on how small components can be made. It is increasingly expensive to make a single processor faster. Using a larger number of moderately fast commodity processors to achieve the same (or better) performance is less expensive.

1.7 LEVELS OF PARALLEL PROCESSING

Depending upon the problem under consideration, parallelism in the solution of the problem may be achieved at different levels and in different ways. This section discusses various levels of parallelism. Parallelism in a problem and its possible solutions may be exploited either manually by the programmer or through automating compilers. We can have parallel processing at four levels.

1.7.1 Instruction Level

It refers to the situation where different instructions of a program are executed by different processing elements. Most processors have several execution units and can execute several instructions (usually machine level) at the same time. Good compilers can reorder instructions to maximize instruction throughput. Often the processor itself can do this. Modern processors even parallelize execution of micro-steps of instructions within the same pipe. The earliest use of instruction level parallelism in designing PE's to enhance processing speed is pipelining. Pipelining was extensively used in early Reduced Instruction Set Computer (RISC). After RISC, super scalar processors were developed which execute multiple instruction in one clock cycle. The super scalar processor design exploits the parallelism available at instruction level by enhancing the number of arithmetic and functional units in PE's. The concept of instruction level parallelism was further modified and applied in the design of Very Large Instruction Word (VLIW) processor, in which one instruction word encodes more than one operation. The idea of executing a number of instructions of a program in parallel by scheduling them on a single processor has been a major driving force in the design of recent processors.

1.7.2 Loop Level

At this level, consecutive loop iterations are the candidates for parallel execution. However, data dependencies between subsequent iterations may restrict parallel execution of instructions at loop level. There is a lot of scope for parallel execution at loop level.

Example: In the following loop in C language,

```
for (i=0;  i <= n; i++)
A(i) = B(i)+ C(i)
```

Each of the instruction A(i) =B(i)+C(i) can be executed by different processing elements provided there are at least n processing elements. However, the instructions in the loop:

```
for ( J=0, J<= n, J++)
A(J) = A(J-1) + B(J)
```

cannot be executed parallely as A(J) is data dependent on A(J-1). This means that before exploiting the loop level parallelism the data dependencies must be checked:

1.7.3 Procedure Level

Here, parallelism is available in the form of parallel executable procedures. In this case, the design of the algorithm plays a major role. For example each thread in Java can be spawned to run a function or method.

1.7.4 Program Level

This is usually the responsibility of the operating system, which runs processes concurrently. Different programs are obviously independent of each other. So parallelism can be extracted by operating the system at this level.

Check Your Progress 2

1) What are the advantages of parallel processing over sequential computations?
……………………………………………………………………………………………
……………………………………………………………………………………………
……………………………………………………………………………………………
……………………………………………………………………………………………
2) Explains the various levels of parallel processing.
……………………………………………………………………………………………
……………………………………………………………………………………………
……………………………………………………………………………………………
……………………………………………………………………………………

1.8 DATAFLOW COMPUTING

An alternative to the von Neumann model of computation is the dataflow computation model. In a dataflow model, control is tied to the flow of data. The order of instructions in the program plays no role on the execution order. Execution of an instruction can take place when all the data needed by the instruction are available. Data is in continuous flow independent of reusable memory cells and its availability initiates execution. Since, data is available for several instructions at the same time, these instructions can be executed in parallel.

For the purpose of exploiting parallelism in computation Data Flow Graph notation is used to represent computations. In a data flow graph, the nodes represent instructions of the program and the edges represent data dependency between instructions. As an example, the dataflow graph for the instruction z = w × (x+y) is shown in *Figure 2*.

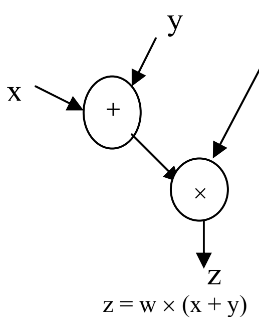


Figure 2: DFG for z = w × (x+y)

Data moves on the edges of the graph in the form of data tokens, which contain data values and status information. The asynchronous parallel computation is determined by the firing rule, which is expressed by means of tokens: a node of DFG can fire if there is a token on each of its input edges. If a node fires, it consumes the input tokens, performs the associated operation and places result tokens on the output edge. Graph nodes can be single instructions or tasks comprising multiple instructions.

The advantage of the dataflow concept is that nodes of DFG can be self-scheduled. However, the hardware support to recognize the availability of necessary data is much more complicated than the von Neumann model. The example of dataflow computer includes Manchester Data Flow Machine, and MIT Tagged Token Data Flow architecture.

1.9 APPLICATIONS OF PARALLEL PROCESSING

Parallel computing is an evolution of serial computing that attempts to emulate what has always been the state of affairs in the natural world. In the natural world, it is quite common to find many complex, interrelated events happening at the same time. Examples of concurrent processing in natural and man-made environments include:

- Automobile assembly line
- Daily operations within a business
- Building a shopping mall
- Ordering an aloo tikki burger at the drive through.

Hence, parallel computing has been considered to be “the high end of computing” and has been motivated by numerical simulations of complex systems and “Grand Challenge Problems” such as:

- Weather forecasting
- Predicting results of chemical and nuclear reactions

- DNA structures of various species
- Design of mechanical devices
- Design of electronic circuits
- Design of complex manufacturing processes
- Accessing of large databases
- Design of oil exploration systems
- Design of web search engines, web based business services
- Design of computer-aided diagnosis in medicine
- Development of MIS for national and multi-national corporations
- Development of advanced graphics and virtual reality software, particularly for the entertainment industry, including networked video and multi-media technologies
- Collaborative work (virtual) environments

1.9.1 Scientific Applications/Image processing

Most of parallel processing applications from science and other academic disciplines, are mainly have based upon numerical simulations where vast quantities of data must be processed, in order to create or test a model. Examples of such applications include:

- Global atmospheric circulation,
- Blood flow circulation in the heart,
- The evolution of galaxies,
- Atomic particle movement,
- Optimisation of mechanical components.

The production of realistic moving images for television and the film industry has become a big business. In the area of large computer animation, though much of the work can be done on high specification workstations, yet the input will often involve the application of parallel processing. Even at the cheap end of the image production spectrum, affordable systems for small production companies have been formed by connecting cheap PC technology using a small LAN to farm off processing work on each image to be produced.

1.9.2 Engineering Applications

Some of the engineering applications are:

- Simulations of artificial ecosystems,
- Airflow circulation over aircraft components.

Airflow circulation is a particularly important application. A large aircraft design company might perform up to five or six full body simulations per working day.

1.9.3 Database Query/Answering Systems

There are a large number of opportunities for speed-up through parallelizing a Database Management System. However, the actual application of parallelism required depends very much on the application area that the DBMS is used for. For example, in the financial sector the DBMS generally is used for short simple transactions, but with a high number of transactions per second. On the other hand in a Computer Aided Design (CAD) situation (e.g., VLSI design) the transactions would be long and with low traffic rates. In a Text query system, the database would undergo few updates, but would be required to do complex pattern matching queries over a large number of entries. An example of a computer designed to speed up database queries is the Teradata computer, which employs parallelism in processing complex queries.

1.9.4 AI Applications

Search is a vital component of an AI system, and the search operations are performed over large quantities of complex structured data using unstructured inputs. Applications of parallelism include:

- Search through the rules of a production system,
- Using fine-grain parallelism to search the semantic networks created by NETL,
- Implementation of Genetic Algorithms,
- Neural Network processors,
- Preprocessing inputs from complex environments, such as visual stimuli.

1.9.5 Mathematical Simulation and Modeling Applications

The tasks involving mathematical simulation and modeling require a lot of parallel processing. Three basic formalisms in mathematical simulation and modeling are Discrete Time System Simulation (DTSS), Differential Equation System Simulation (DESS) and Discrete Event System Simulation (DEVS). All other formalisms are combinations of these three formalisms. DEVS is the most popular. Consequently a number of software tools have been designed for DEVS. Some of such softwares are:

- Parsec, a C-based simulation language for sequential and parallel execution of discrete-event simulation models.
- Omnet++ a discrete-event simulation software development environment written in C++.
- Desmo-J a Discrete event simulation framework in Java.
- Adevs (A Discrete Event System simulator) is a C++ library for constructing discrete event simulations based on the Parallel DEVS and Dynamic Structure DEVS formalisms.
- Any Logic is a professional simulation tool for complex discrete, continuous and hybrid systems.

1.10 INDIA'S PARALLEL COMPUTERS

In India, the development and design of parallel computers started in the early 80's. The Indian Government established the Centre for Development of Advanced Computing (CDAC) in 1988 with the aim of building high-speed parallel machines. CDAC designed and built a 256 processors computer using INMOS T8000 series processors in 1991. The other groups which developed parallel machines were at Centre for Development of Telematics, Bhabha Atomic Research Centre, Indian Institute for Sciences, Defence Research and Development Organisation. The systems developed by these organisations are said to be the state of the art of parallel computers. It is generally agreed that all the computers built by 2020 will be inherently parallel.

India's Parallel Computer

Next, we enumerate sailent features of various generations of parallel systems developed in India.

Sailent Features of PARAM series:

PARAM 8000 CDAC 1991: 256 Processor parallel computer, INMOS 8000 transputer as processing element. Peak performance of 1 Gigaflop. Application software weak.

PARAM 8600 CDAC 1994: PARAM 8000 enhanced with Intel i860 vector microprocessor. One vector processor for 4 INMOS 8000. Vectorized Fortran. Improved software for numerical applications.

PARAM 9000/SS CDAC 1996 : Used Sunsparc II processors and an interconnection switch made of INMOS transputers.

Salient Features of MARK Series:

Flosolver Mark I NAL 1986: Used 4 Intel 8086 processors with 8087 co-processors. Proof of concept design.

Flosolver Mark II NAL 1988: 16 Intel 80386 processor and 80387 floating-point processor connected to Multibus II backplane bus for interprocessor communication. Used for solving fluid dynamics problems using Fortran.

Flosolver Mark III NAL 1991: 8 Intel i860 vector processors connected using message passing co-processor on a back plane bus. i860 were rated at 80 Mflops peak. Fluid dynamics Codes were optimized for the architecture.

Salient Features of ANUPAM Series:

ANUPAM Model 1 BARC 1993: 8 Intel i860 processors connected to a Multibus II. Eight such clusters connected by using 16-bit SCSI interface. Used Front-end processor to allocate tasks to the parallel computing cluster. One parallel program at a time could be run. Fortran environment.

ANUPAM Model 2 BARC 1997: DEC Alpha processors connected by ATM switch in a cluster. DEC Unix environment. High Performance Fortran compiler to run data parallel programs.

1.11 PARALLEL TERMINOLOGY USED

Some of the more commonly used terms associated with parallel computing are listed below.

Task

A logically discrete section of a computational work. A task is typically a program or program-like set of instructions that is executed by a processor.

Parallel Task

A task, some parts of which can be executed by more than one multiple processor at same point of time (yields correct results)

Serial Execution
Execution of a program sequentially, one statement at a time. In the simplest sense, this is what happens on a one processor machine. However, even most of the parallel tasks also have some sections of a parallel program that must be executed serially.

Parallel Execution

Execution of sections of a program by more than one processor at the same point of time.

Shared Memory

Refers to the memory component of a computer system in which the memory can accessed directly by any of the processors in the system.

Distributed Memory

Refers to network based memory in which there is no common address space for the various memory modules comprising the memory system of the (networked) system. Generally, a processor or a group of processors have some memory module associated with it, independent of the memory modules associated with other processors or group of processors.

Communications

Parallel tasks typically need to exchange data. There are several ways in which this can be accomplished, such as, through a shared memory bus or over a network. The actual event of data exchange is commonly referred to as communication regardless of the method employed.

Synchronization

The process of the coordination of parallel tasks in real time, very often associated with communications is called synchronisation. Often implemented by establishing a synchronization point within an application where a task may not proceed further until another task(s) reaches the same or logically equivalent point.
Synchronization usually involves waiting by at least one task, and can therefore cause a parallel application's execution time to increase.

Granularity

In parallel computing, granularity is a qualitative measure of the ratio of computation to communication.

Coarse Granularity: relatively large amounts of computational work are done between communication events

Fine Granularity: relatively small amounts of computational work are done between communication events

Observed Speedup
Observed speedup of a code which has been parallelized, is defined as:

$$\frac{\texttt{wall-clock time of serial}}{\texttt{wall-clock time of parallel}}$$

Granularity is one of the simplest and most widely used indicators for a parallel program's performance.

Parallel Overhead

The amount of time required to coordinate parallel tasks, as opposed to doing useful work. Parallel overhead can include factors such as:

Task start-up time
Synchronisations
Data communications
Software overhead imposed by parallel compilers, libraries, tools, operating system, etc.

Massively Parallel System

Refers to a parallel computer system having a large number of processors. The number in 'a large number of' keeps increasing, and, currently it means more than 1000.

Scalability

Refers to a parallel system's (hardware and/or software) ability to demonstrate a proportionate increase in (parallel) speedup with the addition of more processors. Factors that contribute to scalability include:

Hardware - particularly memory-cpu bandwidths and network communications:

- *Application algorithm*
- *Parallel overhead related*
- *Characteristics of your specific application and coding*

Check Your Progress 3

1) Explain dataflow computation model.
……………………………………………………………………………………………
……………………………………………………………………………………………
……………………………………………………………………………………………
……………………………………………………………………………………
2) Enumerate applications of parallel processing.
……………………………………………………………………………………………
……………………………………………………………………………………………
……………………………………………………………………………………………
……………………………………………………………………………………………

1.12 SUMMARY

In this unit, a number of introductory issues and concepts in respect of parallel computing are discussed. First of all, section 1.2 briefly discusses history of parallel computing. Next section discusses two types of parallelism, viz, temporal and data parallelisms. The issues relating to performance evaluation of a parallel system are discussed in section 1.4. Section 1.5 defines a number of new concepts. Next section explains why parallel computation is essential for solving computationally difficult problems. Section 1.7 discusses how parallelism can be achieved at different levels within a program. Dataflow computing is a different paradigm of computing as compared to the most frequently used Von-Neumann-architecture based computing. Dataflow computing allows to exploit easily the inherent parallelism in a problem and its solution. Issues related to Dataflow computing are discussed in section 1.8. Applications of parallel computing are discussed in section 1.9. India's effort at developing parallel computers is briefly discussed in section 1.10. A glossary of parallel computing terms is given in section 1.11.

1.13 SOLUTIONS/ANSWERS

Check Your Progress 1

1) Each process has a life cycle, which consists of creation, execution and termination phases of the process. A process may create several new processes, which in turn may also create still new processes, using system calls. In UNIX operating system environment, a new process is created by *fork* system call. Process creation requires the following four actions:

i) **Setting up the process description:** Setting up the process description requires the creation of a *Process Control Block* (PCB). A PCB contains basic data such as process identification number, owner, process status, description of the allocated address space and other implementation dependent process specific information needed for process management.

ii) **Allocating an address space:** There are two ways to allocate address space to processes; sharing the address space among the created processes or allocating separate space to each process.

iii) **Loading the program into the allocated address space:** The executable program file is loaded into the allocated memory space.

iv) **Passing the process description to the process scheduler:** The process created is then passed to the process scheduler which allocates the processor to the competing processes.

The process execution phase is controlled by the process scheduler. Process scheduling may be per process or per thread. The process scheduling involves three concepts: process states, state transition diagram and scheduling policy.
A process may be in one of the following states:

- **New:** The process is being created.
- **Running:** The process is being executed on a single or multiple processors.
- **Waiting:** The process is waiting for some event to occur.
- **Ready:** The process is ready to be executed if a processor is available.

- **Terminated:** The process has finished execution.

At any time a process may be in any one of the above said states. As soon as the process is admitted into the job queue, it goes into ready state. When the process scheduler dispatches the process, its state becomes running. When the process is completely executed then it is terminated and we say that it is in the terminated state. However, the process may return to ready state due to some interruption or may go to waiting state due to some I/O activity. When I/O activity is over it may go to ready state. The state transition diagram is shown below in the Figure 3:

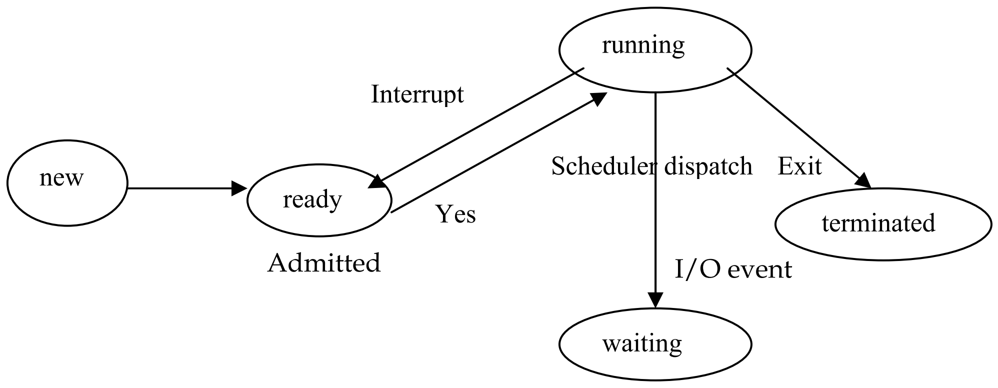


Figure 3: Process state transition diagram

The scheduling policy may be either pre-emptive or non pre-emptive. In pre-emptive policy, the process may be interrupted. Operating systems have different scheduling policies. One of the well-known policies is First In First Out(FIFO) to select the process to be executed.When the process finishes execution it is terminated by system calls like *abort*.

2) Some of the advantages that threads offer over processes include:

i) It takes less time to create and terminate a new thread than it takes for a process, because the newly created thread uses the current process address space.
ii) It takes less time to switch between two threads within the same process, partly because the newly created thread uses the current process address space.
iii) Less communication overheads -- communicating between the threads of one process is simple because the threads share the address space, in particular. So, data produced by one thread is immediately available to all the other threads.

3) The words “concurrent” and “parallel” are often used interchangeably, however they are distinct. Concurrent execution is the temporal behaviour of the N-client 1-server model where only one client is served at any given moment. It has a dual nature: it is sequential in a small time scale, but simultaneous in large time scale. In our context the processor works as a server and a process or a thread works as a client. For facilitating expression of concurrent programs, number of concurrent languages are available including Ada, concurrent pascal, Modula-2 and concurrent PROLOG.

Parallel execution is associated with the N-client N-server model. It allows the servicing of more than one client at the same time as the number of servers is more than one. For facilitating expression of parallel programs, a number of parallel languages are available including: Occam-2, Parallel C and strand-88.

Parallel execution does not need explicit concurrency in the language. Parallelism can be achieved by the underlying hardware. Similarly, we can have concurrency in a language without parallel execution. This is the case when a program is executed on a single processor.

4) Granularity refers to the amount of computation done in parallel relative to the size of the whole program. In parallel computing, granularity is a qualitative measure of the ratio of computation to communication. According to granularity of the system, parallel-processing systems can be divided into two groups: fine-grain systems and coarse-grain systems. In fine-grained systems parallel parts are relatively small and which means more frequent communication. Fine-grain processings have low computation to communication ratio and require high communication overhead. In coarse grained systems parallel parts are relatively large and which means more computation and less communication. If granularity is too fine it is possible that the overhead required for communications and synchronization between tasks takes longer than the computation. On the other hand, in coarse-grain parallel systems, relatively large amounts of computational work is done. Coarse-grain processings have high computation to communication ratio and imply more opportunity for performance increase.

Check Your Progress 2

1) Parallel computing has the following advantages over sequential computing:

i) Saves time
ii) Solves larger problems.
iii) Large pool of memory.

2) Levels of parallel processing:

We can have parallel processing at four levels.

i) **Instruction Level:** Most processors have several execution units and can execute several instructions (usually machine level) at the same time. Good compilers can reorder instructions to maximize instruction throughput. Often the processor itself can do this. Modern processors even parallelize execution of micro-steps of instructions within the same pipe.

ii) **Loop Level:** Here, consecutive loop iterations are candidates for parallel execution. However, data between subsequent iterations may restrict parallel execution of instructions at loop level. There is a lot of scope for parallel execution at loop level.

iii) **Procedure Level:** Here parallelism is available in the form of parallel executable procedures. Here the design of the algorithm plays a major role. For example each thread in Java can be spawned to run a function or method.

iv) **Program Level:** This is usually the responsibility of the operating system, which runs processes concurrently. Different programs are obviously independent of each other. So parallelism can be extracted by the operating system at this level.

Check Your Progress 3

1) An alternative to the von Neumann model of computation is dataflow computation model. In a dataflow model control is tied to the flow of data. The order of instructions in the program plays no role in the execution order. Computations take place when all the data items needed for initiating execution of an instruction are available. Data is in continuous flow independent of reusable memory cells and its availability initiates execution. Since data may be available for several instructions at the same time, these instructions can be executed in parallel.

The potential for parallel computation, is reflected by the dataflow graph, the nodes of which are the instructions of the program and the edges of which represent data dependency between instructions. The dataflow graph for the instruction z = w × (x+y) is shown below.

.

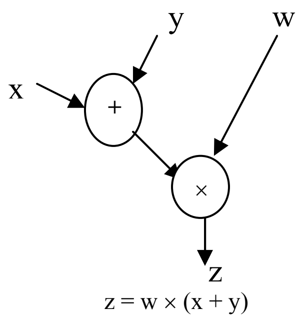


Figure 4: DFG for z = w × (x+y)

Data moves on the edges of the graph in the form of data tokens, which contain data values and status information. The asynchronous parallel computation is determined by the firing rule, which is expressed by means of tokens: a node of DFG can fire if there is a token on each input edge. If a node fires, it consumes the input tokens, performs the associated operation and places result tokens on the output edge. Graph nodes can be single instructions or tasks comprising multiple instructions.

2) Please refer Section 1.9.

UNIT 2 CLASSIFICATION OF PARALLEL COMPUTERS

2.0 INTRODUCTION

Parallel computers are those that emphasize the parallel processing between the operations in some way. In the previous unit, all the basic terms of parallel processing and computation have been defined. Parallel computers can be characterized based on the data and instruction streams forming various types of computer organisations. They can also be classified based on the computer structure, e.g. multiple processors having separate memory or one shared global memory. Parallel processing levels can also be defined based on the size of instructions in a program called grain size. Thus, parallel computers can be classified based on various criteria. This unit discusses all types of classification of parallel computers based on the above mentioned criteria.

2.1 OBJECTIVES

After going through this unit, you should be able to:

- explain the various criteria on which classification of parallel computers are based;
- discuss the Flynn's classification based on instruction and data streams;
- describe the Structural classification based on different computer organisations;
- explain the Handler's classification based on three distinct levels of computer: Processor control unit (PCU), Arithmetic logic unit (ALU), Bit-level circuit (BLC), and
- describe the sub-tasks or instructions of a program that can be executed in parallel based on the grain size.

2.2 TYPES OF CLASSIFICATION

The following classification of parallel computers have been identified:

1) Classification based on the instruction and data streams
2) Classification based on the structure of computers
3) Classification based on how the memory is accessed
4) Classification based on grain size

All these classification schemes are discussed in subsequent sections.

2.3 FLYNN'S CLASSIFICATION

This classification was first studied and proposed by Michael Flynn in 1972. Flynn did not consider the machine architecture for classification of parallel computers; he introduced the concept of *instruction* and *data* streams for categorizing of computers. All the computers classified by Flynn are not parallel computers, but to grasp the concept of parallel computers, it is necessary to understand all types of Flynn's classification. Since, this classification is based on instruction and data streams, first we need to understand how the instruction cycle works.

2.3.1 Instruction Cycle

The instruction cycle consists of a sequence of steps needed for the execution of an instruction in a program. A typical instruction in a program is composed of two parts: Opcode and Operand. The Operand part specifies the data on which the specified operation is to be done. (See *Figure 1*). The Operand part is divided into two parts: addressing mode and the Operand. The addressing mode specifies the method of determining the addresses of the actual data on which the operation is to be performed and the operand part is used as an argument by the method in determining the actual address.

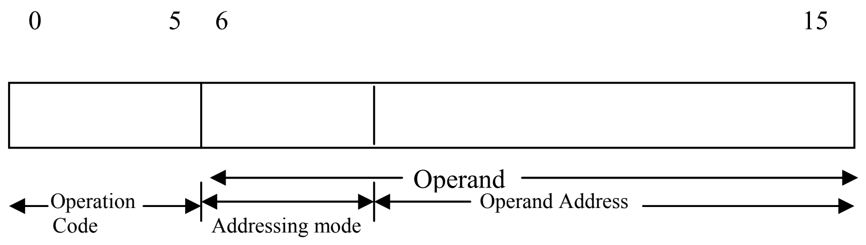


Figure 1: Opcode and Operand

The control unit of the CPU of the computer fetches instructions in the program, one at a time. The fetched Instruction is then decoded by the decoder which is a part of the control unit and the processor executes the decoded instructions. The result of execution is temporarily stored in Memory Buffer Register (MBR) (also called Memory Data Register). The normal execution steps are shown in *Figure 2*.

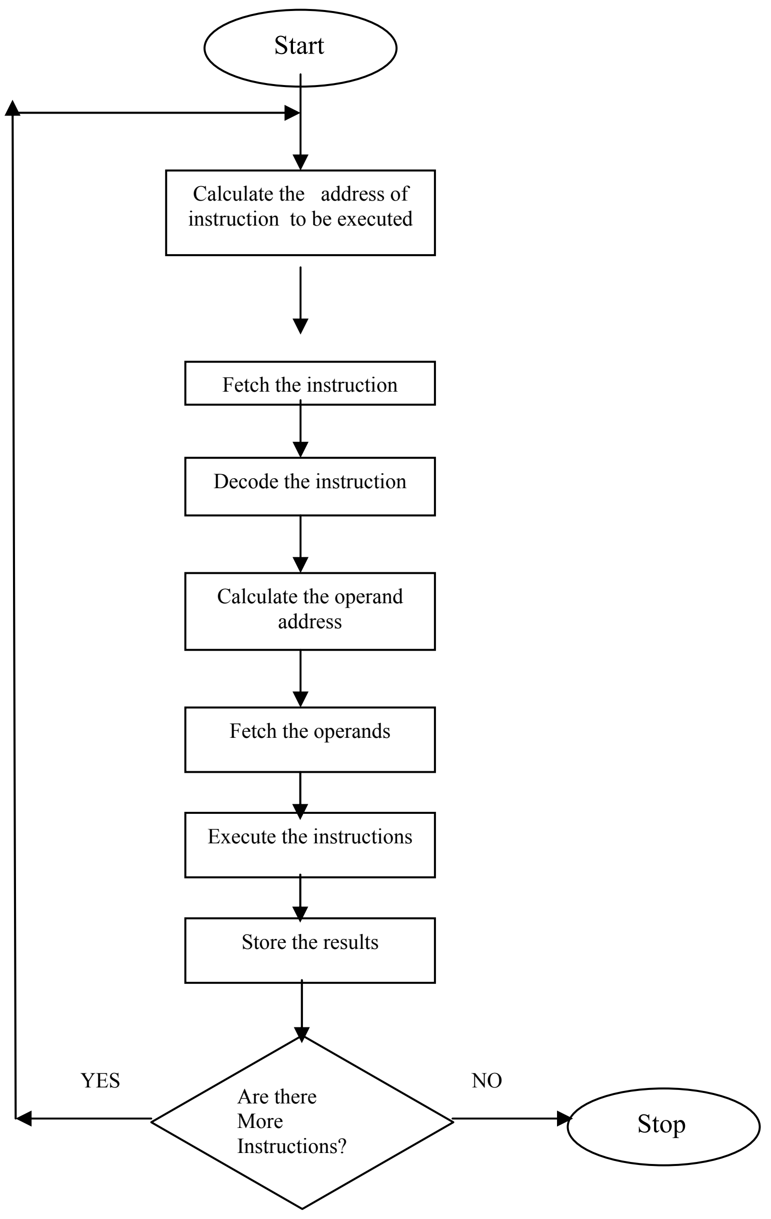


Figure 2: Instruction execution steps

2.3.2 Instruction Stream and Data Stream

The term 'stream' refers to a sequence or flow of either instructions or data operated on by the computer. In the complete cycle of instruction execution, a flow of instructions from main memory to the CPU is established. This flow of instructions is called ***instruction stream.*** Similarly, there is a flow of operands between processor and memory bi-directionally. This flow of operands is called ***data stream***. These two types of streams are shown in *Figure 3*.

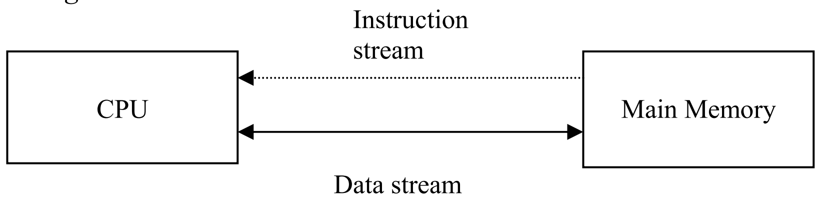


Figure 3: Instruction and data stream

Thus, it can be said that the sequence of instructions executed by CPU forms the Instruction streams and sequence of data (operands) required for execution of instructions form the Data streams.

2.3.3 Flynn's Classification

Flynn's classification is based on multiplicity of instruction streams and data streams observed by the CPU during program execution. Let I_s and D_s are minimum number of streams flowing at any point in the execution, then the computer organisation can be categorized as follows:

1) **Single Instruction and Single Data stream (SISD)**

In this organisation, sequential execution of instructions is performed by one CPU containing a single processing element (PE), i.e., ALU under one control unit as shown in *Figure 4*. Therefore, SISD machines are conventional serial computers that process only one stream of instructions and one stream of data. This type of computer organisation is depicted in the diagram:

$I_s = D_s = 1$

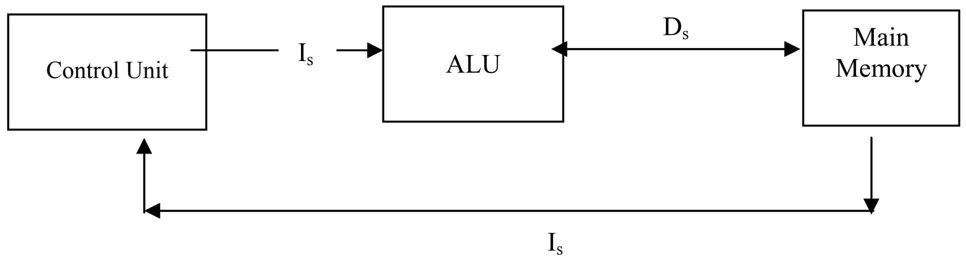


Figure 4: SISD Organisation

Examples of SISD machines include:

- CDC 6600 which is unpipelined but has multiple functional units.
- CDC 7600 which has a pipelined arithmetic unit.
- Amdhal 470/6 which has pipelined instruction processing.
- Cray-1 which supports vector processing.

2) **Single Instruction and Multiple Data stream (SIMD)**

In this organisation, multiple processing elements work under the control of a single control unit. It has one instruction and multiple data stream. All the processing elements of this organization receive the same instruction broadcast from the CU. Main memory can also be divided into modules for generating multiple data streams acting as a *distributed memory* as shown in *Figure 5*. Therefore, all the processing elements simultaneously execute the same instruction and are said to be 'lock-stepped' together. Each processor takes the data from its own memory and hence it has on distinct data streams. (Some systems also provide a shared global memory for communications.) Every processor must be allowed to complete its instruction before the next instruction is taken for execution. Thus, the execution of instructions is synchronous. Examples of SIMD organisation are ILLIAC-IV, PEPE, BSP, STARAN, MPP, DAP and the Connection Machine (CM-1).

This type of computer organisation is denoted as:

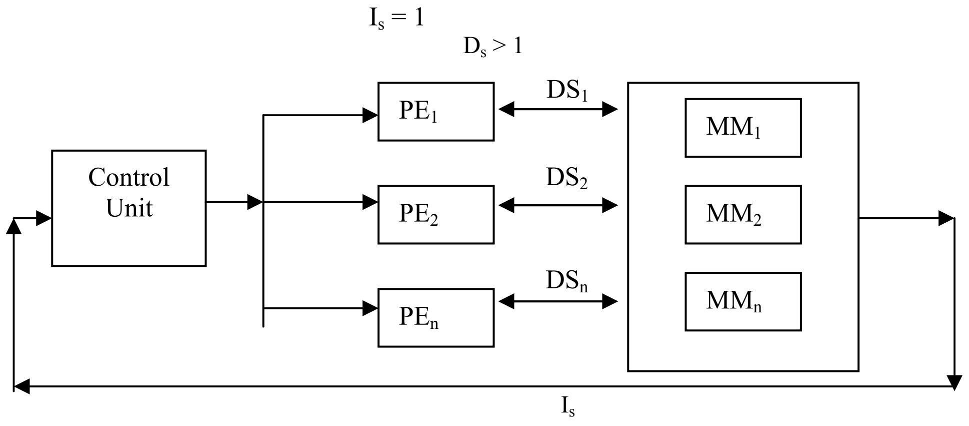


Figure 5: SIMD Organisation

3) **Multiple Instruction and Single Data stream (MISD)**

In this organization, multiple processing elements are organised under the control of multiple control units. Each control unit is handling one instruction stream and processed through its corresponding processing element. But each processing element is processing only a single data stream at a time. Therefore, for handling multiple instruction streams and single data stream, multiple control units and multiple processing elements are organised in this classification. All processing elements are interacting with the common shared memory for the organisation of single data stream as shown in *Figure 6*. The only known example of a computer capable of MISD operation is the C.mmp built by Carnegie-Mellon University.
This type of computer organisation is denoted as:

$I_s > 1$
$D_s = 1$

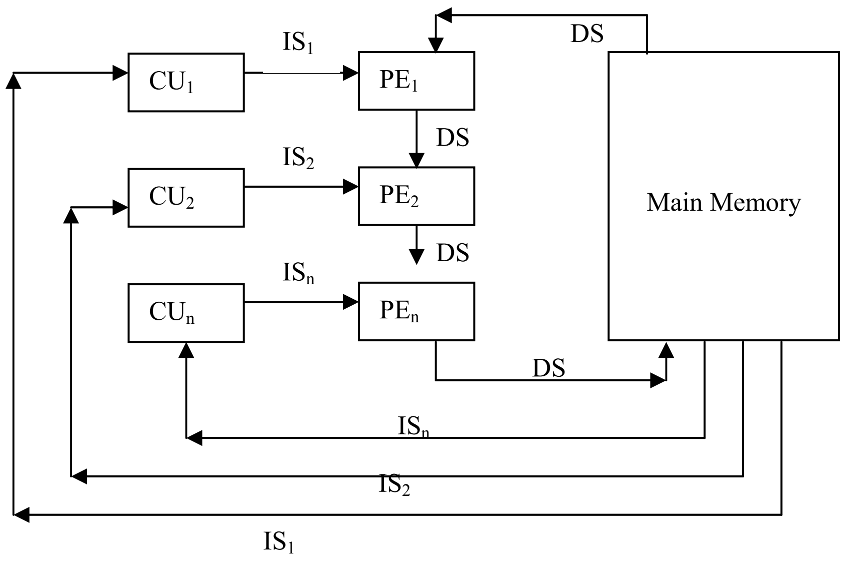


Figure 6: MISD Organisation

This classification is not popular in commercial machines as the concept of single data streams executing on multiple processors is rarely applied. But for the specialized applications, MISD organisation can be very helpful. For example, Real time computers need to be fault tolerant where several processors execute the same data for producing the redundant data. This is also known as N- version programming. All these redundant data are compared as results which should be same; otherwise faulty unit is replaced. Thus MISD machines can be applied to fault tolerant real time computers.

4) **Multiple Instruction and Multiple Data stream (MIMD)**

In this organization, multiple processing elements and multiple control units are organized as in MISD. But the difference is that now in this organization multiple instruction streams operate on multiple data streams . Therefore, for handling multiple instruction streams, multiple control units and multiple processing elements are organized such that multiple processing elements are handling multiple data streams from the Main memory as shown in *Figure 7*. The processors work on their own data with their own instructions. Tasks executed by different processors can start or finish at different times. They are not lock-stepped, as in SIMD computers, but run asynchronously. This classification actually recognizes the parallel computer. That means in the real sense MIMD organisation is said to be a Parallel computer. All multiprocessor systems fall under this classification. Examples include; C.mmp, Burroughs D825, Cray-2, S1, Cray X-MP, HEP, Pluribus, IBM 370/168 MP, Univac 1100/80, Tandem/16, IBM 3081/3084, C.m*, BBN Butterfly, Meiko Computing Surface (CS-1), FPS T/40000, iPSC.

This type of computer organisation is denoted as:

$I_s > 1$
$D_s > 1$

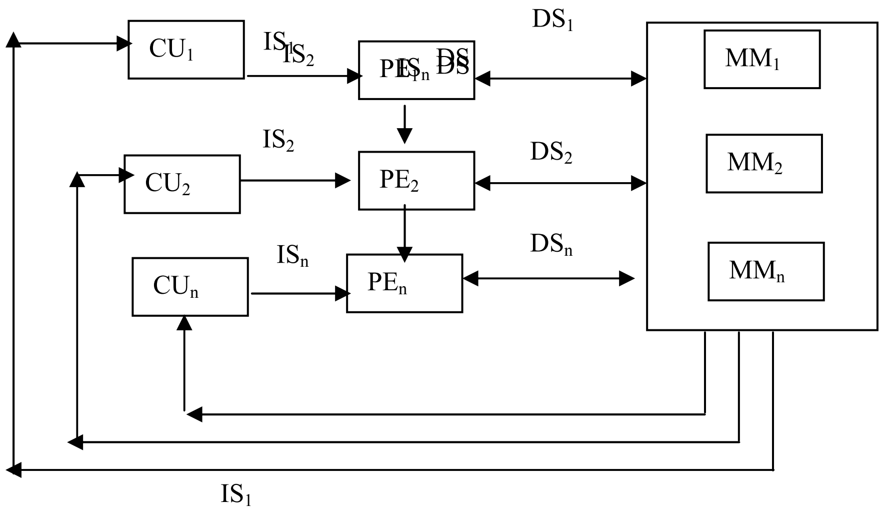


Figure 7: MIMD Organisation

Of the classifications discussed above, MIMD organization is the most popular for a parallel computer. In the real sense, parallel computers execute the instructions in MIMD mode.

Check Your Progress 1

1) What are various criteria for classification of parallel computers?
………………………………………………………………………………………
………………………………………………………………………………………
………………………………………………………………………………………
………………………………………………………………………………………
2) Define instruction and data streams.
………………………………………………………………………………………
………………………………………………………………………………………

……………………………………………………………………………………
……………………………………………………………………………………

3) State whether True or False for the following:
   a) SISD computers can be characterized as $I_s > 1$ and $D_s > 1$
   b) SIMD computers can be characterized as $I_s > 1$ and $D_s = 1$
   c) MISD computers can be characterized as $I_s = 1$ and $D_s = 1$
   d) MIMD computers can be characterized as $I_s > 1$ and $D_s > 1$

2.4 HANDLER'S CLASSIFICATION

In 1977, Wolfgang Handler proposed an elaborate notation for expressing the pipelining and parallelism of computers. Handler's classification addresses the computer at three distinct levels:

- Processor control unit (PCU),
- Arithmetic logic unit (ALU),
- Bit-level circuit (BLC).

The PCU corresponds to a processor or CPU, the ALU corresponds to a functional unit or a processing element and the BLC corresponds to the logic circuit needed to perform one-bit operations in the ALU.

Handler's classification uses the following three pairs of integers to describe a computer:

Computer = (p * p', a * a', b * b')

Where p = number of PCUs
Where p'= number of PCUs that can be pipelined
Where a = number of ALUs controlled by each PCU
Where a'= number of ALUs that can be pipelined
Where b = number of bits in ALU or processing element (PE) word
Where b'= number of pipeline segments on all ALUs or in a single PE

The following rules and operators are used to show the relationship between various elements of the computer:

- The '*' operator is used to indicate that the units are pipelined or macro-pipelined with a stream of data running through all the units.
- The '+' operator is used to indicate that the units are not pipelined but work on independent streams of data.
- The 'v' operator is used to indicate that the computer hardware can work in one of several modes.
- The '~' symbol is used to indicate a range of values for any one of the parameters.
- Peripheral processors are shown before the main processor using another three pairs of integers. If the value of the second element of any pair is 1, it may omitted for brevity.

Handler's classification is best explained by showing how the rules and operators are used to classify several machines.

The CDC 6600 has a single main processor supported by 10 I/O processors. One control unit coordinates one ALU with a 60-bit word length. The ALU has 10 functional units which can be formed into a pipeline. The 10 peripheral I/O processors may work in parallel with each other and with the CPU. Each I/O processor contains one 12-bit ALU. The description for the 10 I/O processors is:

CDC 6600I/O = (10, 1, 12)

The description for the main processor is:

CDC 6600main = (1, 1 * 10, 60)

The main processor and the I/O processors can be regarded as forming a macro-pipeline so the '*' operator is used to combine the two structures:

CDC 6600 = (I/O processors) * (central processor = (10, 1, 12) * (1, 1 * 10, 60)

Texas Instrument's Advanced Scientific Computer (ASC) has one controller coordinating four arithmetic units. Each arithmetic unit is an eight stage pipeline with 64-bit words. Thus we have:

ASC = (1, 4, 64 * 8)

The Cray-1 is a 64-bit single processor computer whose ALU has twelve functional units, eight of which can be chained together to from a pipeline. Different functional units have from 1 to 14 segments, which can also be pipelined. Handler's description of the Cray-1 is:

Cray-1 = (1, 12 * 8, 64 * (1 ~ 14))

Another sample system is Carnegie-Mellon University's C.mmp multiprocessor. This system was designed to facilitate research into parallel computer architectures and consequently can be extensively reconfigured. The system consists of 16 PDP-11 'minicomputers' (which have a 16-bit word length), interconnected by a crossbar switching network. Normally, the C.mmp operates in MIMD mode for which the description is (16, 1, 16). It can also operate in SIMD mode, where all the processors are coordinated by a single master controller. The SIMD mode description is (1, 16, 16). Finally, the system can be rearranged to operate in MISD mode. Here the processors are arranged in a chain with a single stream of data passing through all of them. The MISD modes description is (1 * 16, 1, 16). The 'v' operator is used to combine descriptions of the same piece of hardware operating in differing modes. Thus, Handler's description for the complete C.mmp is:

C.mmp = (16, 1, 16) v (1, 16, 16) v (1 * 16, 1, 16)

The '*' and '+' operators are used to combine several separate pieces of hardware. The 'v' operator is of a different form to the other two in that it is used to combine the different operating modes of a single piece of hardware.

While Flynn's classification is easy to use, Handler's classification is cumbersome. The direct use of numbers in the nomenclature of Handler's classification's makes it much more abstract and hence difficult. Handler's classification is highly geared towards the description of pipelines and chains. While it is well able to describe the parallelism in a single processor, the variety of parallelism in multiprocessor computers is not addressed well.

2.5 STRUCTURAL CLASSIFICATION

Flynn's classification discusses the behavioural concept and does not take into consideration the computer's structure. Parallel computers can be classified based on their structure also, which is discussed below and shown in *Figure 8*.

As we have seen, a parallel computer (MIMD) can be characterised as a set of multiple processors and shared memory or memory modules communicating via an interconnection network. When multiprocessors communicate through the global shared memory modules then this organisation is called ***Shared memory computer*** or ***Tightly***

coupled systems as shown in *Figure 9.* Similarly when every processor in a multiprocessor system, has its own local memory and the processors communicate via messages transmitted between their local memories, then this organisation is called ***Distributed memory computer*** or ***Loosely coupled system*** as shown in *Figure 10*. *Figures 9* and *10* show the simplified diagrams of both organisations.

The processors and memory in both organisations are interconnected via an interconnection network. This interconnection network may be in different forms like crossbar switch, multistage network, etc. which will be discussed in the next unit.

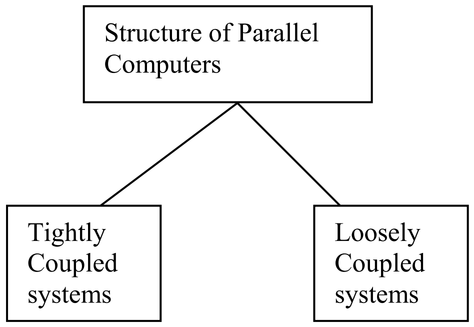


Figure 8: Structural classification

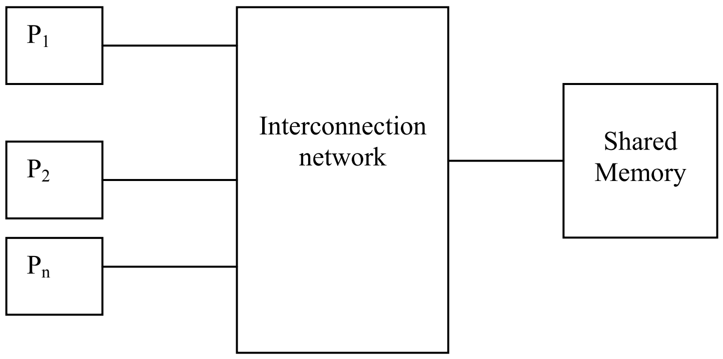


Figure 9: Tightly coupled system

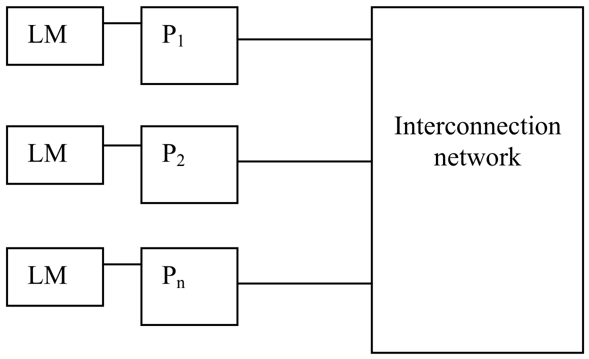


Figure 10 Loosely coupled system

2.5.1 Shared Memory System / Tightly Coupled System

Shared memory multiprocessors have the following characteristics:

- Every processor communicates through a shared global memory.

- For high speed real time processing, these systems are preferable as their throughput is high as compared to loosely coupled systems.

In tightly coupled system organization, multiple processors share a global main memory, which may have many modules as shown in detailed *Figure 11*. The processors have also access to I/O devices. The inter- communication between processors, memory, and other devices are implemented through various interconnection networks, which are discussed below.

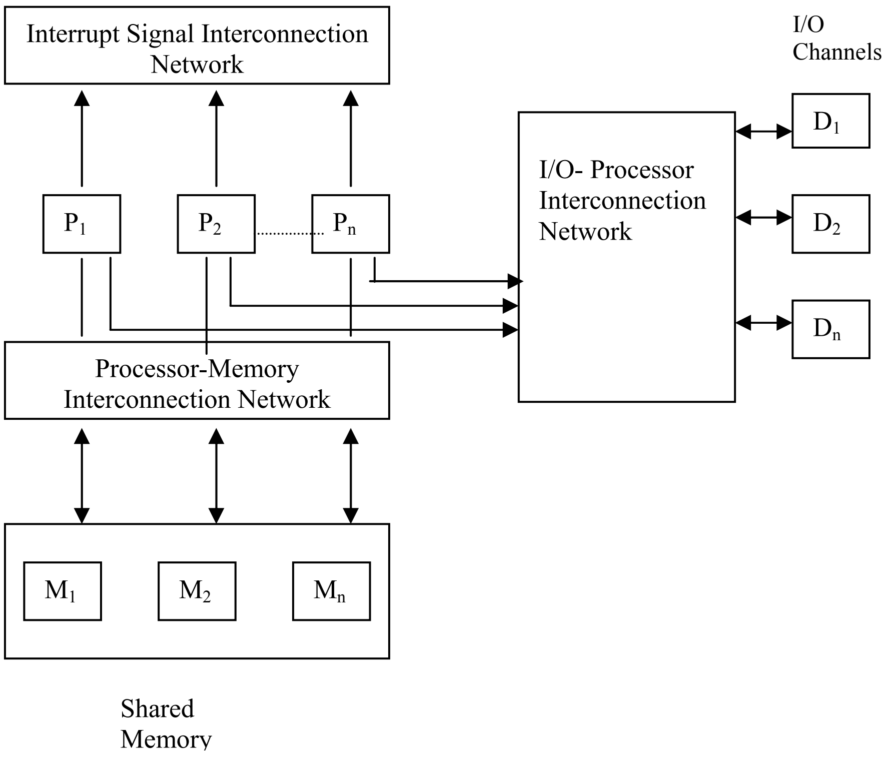

Interrupt Signal Interconnection Network
I/O Channels
I/O- Processor Interconnection Network
P_1
P_2
P_n
D_1
D_2
D_n
Processor-Memory Interconnection Network
M_1
M_2
M_n
Shared Memory

Figure 11: Tightly coupled system organization

i) **Processor-Memory Interconnection Network (PMIN)**

This is a switch that connects various processors to different memory modules. Connecting every processor to every memory module in a single stage while the crossbar switch may become complex. Therefore, multistage network can be adopted. There can be a conflict among processors such that they attempt to access the same memory modules. This conflict is also resolved by PMIN.

ii) **Input-Output-Processor Interconnection Network (IOPIN)**

This interconnection network is used for communication between processors and I/O channels. All processors communicate with an I/O channel to interact with an I/O device with the prior permission of IOPIN.

iii) **Interrupt Signal Interconnection Network (ISIN)**

When a processor wants to send an interruption to another processor, then this interrupt first goes to ISIN, through which it is passed to the destination processor. In this way, synchronisation between processor is implemented by ISIN. Moreover, in case of failure of one processor, ISIN can broadcast the message to other processors about its failure.

Since, every reference to the memory in tightly coupled systems is via interconnection network, there is a delay in executing the instructions. To reduce this delay, every processor may use cache memory for the frequent references made by the processor as shown in *Figure 12*.

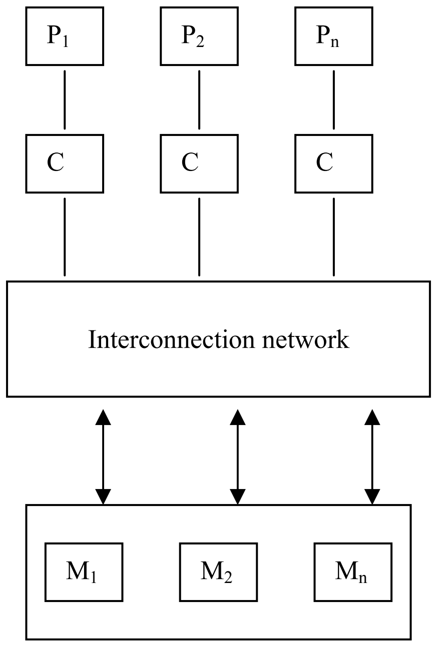


Figure 12: Tightly coupled systems with cache memory

The shared memory multiprocessor systems can further be divided into three modes which are based on the manner in which shared memory is accessed. These modes are shown in *Figure 13* and are discussed below.

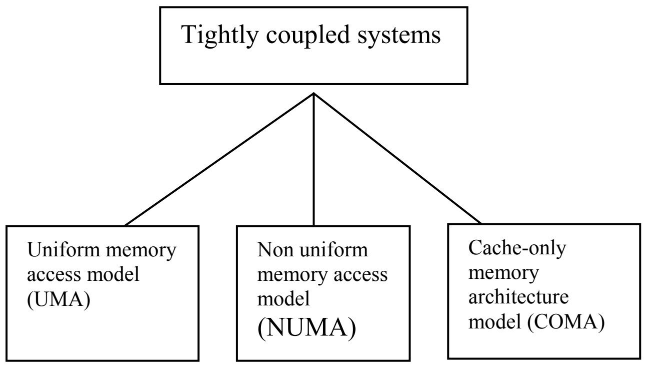

Tightly coupled systems
Uniform memory access model (UMA)
Non uniform memory access model (NUMA)
Cache-only memory architecture model (COMA)

Figure 13: Modes of Tightly coupled systems

2.5.1.1 Uniform Memory Access Model (UMA)

In this model, main memory is uniformly shared by all processors in multiprocessor systems and each processor has equal access time to shared memory. This model is used for time-sharing applications in a multi user environment.

2.5.1.2 Non-Uniform Memory Access Model (NUMA)

In shared memory multiprocessor systems, local memories can be connected with every processor. The collection of all local memories form the global memory being shared. In this way, global memory is distributed to all the processors. In this case, the access to a local memory is uniform for its corresponding processor as it is attached to the local memory. But if one reference is to the local memory of some other remote processor, then the access is not uniform. It depends on the location of the memory. Thus, all memory words are not accessed uniformly.

2.5.1.3 Cache-Only Memory Access Model (COMA)

As we have discussed earlier, shared memory multiprocessor systems may use cache memories with every processor for reducing the execution time of an instruction. Thus in NUMA model, if we use cache memories instead of local memories, then it becomes COMA model. The collection of cache memories form a global memory space. The remote cache access is also non-uniform in this model.

2.5.2 Loosely Coupled Systems

These systems do not share the global memory because shared memory concept gives rise to the problem of memory conflicts, which in turn slows down the execution of instructions. Therefore, to alleviate this problem, each processor in loosely coupled systems is having a large local memory (LM), which is not shared by any other processor. Thus, such systems have multiple processors with their own local memory and a set of I/O devices. This set of processor, memory and I/O devices makes a computer system. Therefore, these systems are also called multi-computer systems. These computer systems are connected together via message passing interconnection network through which processes communicate by passing messages to one another. Since every computer system or node in multicomputer systems has a separate memory, they are called distributed multicomputer systems. These are also called loosely coupled systems, meaning that nodes have little coupling between them as shown in *Figure 14*.

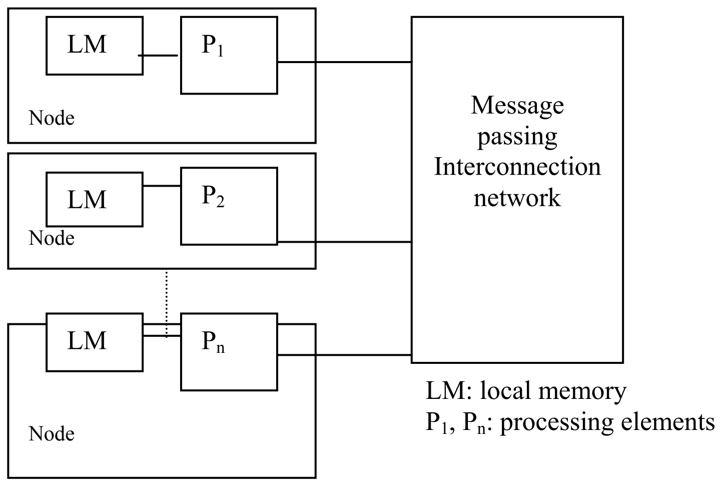

LM
P_1
Node
LM
P_2
Node
LM
P_n
Node
Message passing Interconnection network
LM: local memory
P_1, P_n: processing elements

Figure 14: Loosely coupled system organisation

Since local memories are accessible to the attached processor only, no processor can access remote memory. Therefore, these systems are also known as no-remote memory access (NORMA) systems. Message passing interconnection network provides connection to every node and inter-node communication with message depends on the type of interconnection network. For example, interconnection network for a non-hierarchical system can be shared bus.

Check Your Progress 2

1) What are the various rules and operators used in Handler's classification for various machine types?

……………………………………………………………………………………………..
……………………………………………………………………………………………
……………………………………………………………………………………………
..……………………………………………………………………………………

2) What is the base for structural classification of parallel computers?
…………………………………………………………………………………………..
…………………………………………………………………………………………..
…………………………………………………………………………………………..
…………………………………………………………………………………………..

3) Define loosely coupled systems and tightly coupled systems.
…………………………………………………………………………………………
…………………………………………………………………………………………
..……………………………………………………………………………………..
…..……………………………………………………………………………………..

4) Differentiate between UMA, NUMA and COMA.
…………………………………………………………………………………………..
…………………………………………………………………………………………..
…………………………………………………………………………………………..
…………………………………………………………………………………………..

2.6 CLASSIFICATION BASED ON GRAIN SIZE

This classification is based on recognizing the parallelism in a program to be executed on a multiprocessor system. The idea is to identify the sub-tasks or instructions in a program that can be executed in parallel. For example, there are 3 statements in a program and statements S1 and S2 can be exchanged. That means, these are not sequential as shown in *Figure 15*. Then S1 and S2 can be executed in parallel.

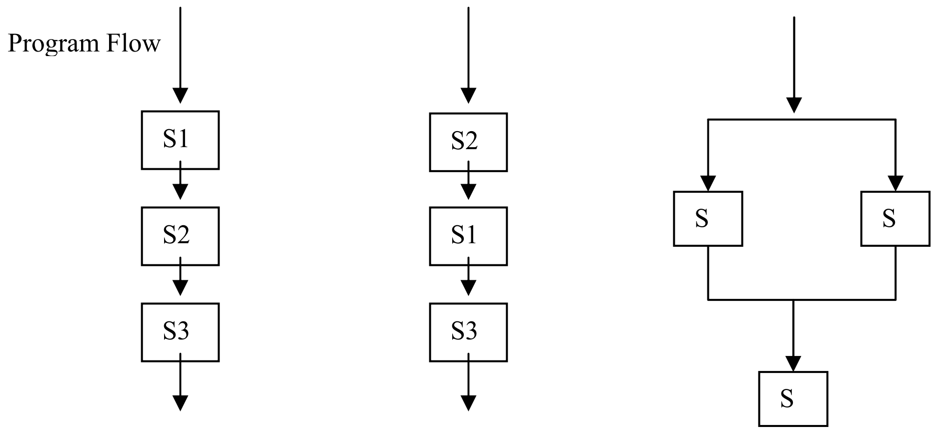


Figure 15: Parallel execution for S1 and S2

But it is not sufficient to check for the parallelism between statements or processes in a program. The decision of parallelism also depends on the following factors:

- Number and types of processors available, i.e., architectural features of host computer
- Memory organisation
- Dependency of data, control and resources

2.6.1 Parallelism Conditions

As discussed above, parallel computing requires that the segments to be executed in parallel must be independent of each other. So, before executing parallelism, all the conditions of parallelism between the segments must be analyzed. In this section, we discuss three types of dependency conditions between the segments
(shown in *Figure 16*).

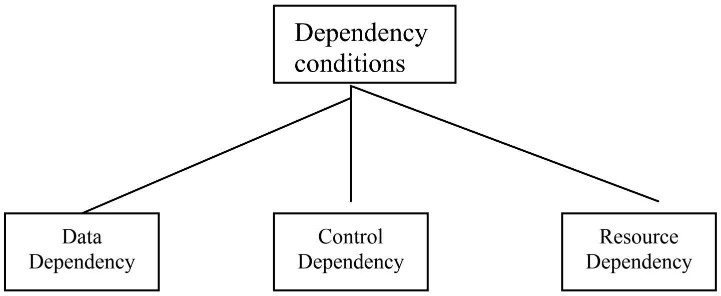


Figure 16: Dependency relations among the segments for parallelism

Data Dependency: It refers to the situation in which two or more instructions share same data. The instructions in a program can be arranged based on the relationship of data dependency; this means how two instructions or segments are data dependent on each other. The following types of data dependencies are recognised:

i) **Flow Dependence :** If instruction I_2 follows I_1 and output of I_1 becomes input of I_2, then I_2 is said to be flow dependent on I_1.

ii) **Antidependence :** When instruction I_2 follows I_1 such that output of I_2 overlaps with the input of I_1 on the same data.

iii) **Output dependence :** When output of the two instructions I_1 and I_2 overlap on the same data, the instructions are said to be output dependent.

iv) **I/O dependence :** When read and write operations by two instructions are invoked on the same file, it is a situation of I/O dependence.

Consider the following program instructions:

I_1: a = b
I_2: c = a + d
I_3: a = c

In this program segment instructions I_1 and I_2 are Flow dependent because variable a is generated by I_1 as output and used by I_2 as input. Instructions I_2 and I_3 are Antidependent because variable a is generated by I_3 but used by I_2 and in sequence I_2 comes first. I_3 is flow dependent on I_2 because of variable c. Instructions I_3 and I_1 are Output dependent because variable a is generated by both instructions.

Control Dependence: Instructions or segments in a program may contain control structures. Therefore, dependency among the statements can be in control structures also. But the order of execution in control structures is not known before the run time. Thus, control structures dependency among the instructions must be analyzed carefully. For example, the successive iterations in the following control structure are dependent on one another.

```
For ( i= 1; I<= n ; i++)
{
        if (x[i - 1] == 0)
                x[i] =0
        else
                x[i] = 1;
}
```

Resource Dependence : The parallelism between the instructions may also be affected due to the shared resources. If two instructions are using the same shared resource then it is a resource dependency condition. For example, floating point units or registers are shared, and this is known as *ALU dependency.* When memory is being shared, then it is called S*torage dependency.*

2.6.2 Bernstein Conditions for Detection of Parallelism

For execution of instructions or block of instructions in parallel, it should be ensured that the instructions are independent of each other. These instructions can be data dependent / control dependent / resource dependent on each other. Here we consider only data dependency among the statements for taking decisions of parallel execution. **A.J. Bernstein** has elaborated the work of data dependency and derived some conditions based on which we can decide the parallelism of instructions or processes.

Bernstein conditions are based on the following two sets of variables:

i) The Read set or input set R_I that consists of memory locations read by the statement of instruction I_1.
ii) The Write set or output set W_I that consists of memory locations written into by instruction I_1.

The sets R_I and W_I are not disjoint as the same locations are used for reading and writing by S_I.

The following are Bernstein Parallelism conditions which are used to determine whether statements are parallel or not:

1) Locations in R_1 from which S_1 reads and the locations W_2 onto which S_2 writes must be mutually exclusive. That means S_1 does not read from any memory location onto which S_2 writes. It can be denoted as:
$R_1 \cap W_2 = \phi$
2) Similarly, locations in R_2 from which S_2 reads and the locations W_1 onto which S_1 writes must be mutually exclusive. That means S_2 does not read from any memory location onto which S_1 writes. It can be denoted as: $R_2 \cap W_1 = \phi$
3) The memory locations W_1 and W_2 onto which S_1 and S_2 write, should not be read by S_1 and S_2. That means R_1 and R_2 should be independent of W_1 and W_2. It can be denoted as : $W_1 \cap W_2 = \phi$

To show the operation of Bernstein's conditions, consider the following instructions of sequential program:

I1 : $x = (a + b) / (a * b)$
I2 : $y = (b + c) * d$
I3 : $z = x^2 + (a * e)$

Now, the read set and write set of I1, I2 and I3 are as follows:

$R_1 = \{a,b\}$	$W_1 = \{x\}$
$R_2 = \{b,c,d\}$	$W_2 = \{y\}$
$R_3 = \{x,a,e\}$	$W_3 = \{z\}$

Now let us find out whether I_1 and I_2 are parallel or not

$R_1 \cap W_2 = \phi$
$R_2 \cap W_1 = \phi$
$W_1 \cap W_2 = \phi$

That means I_1 and I_2 are independent of each other.
Similarly for $I_1 \| I_3$,

$R_1 \cap W_3 = \phi$
$R_3 \cap W_1 \neq \phi$
$W_1 \cap W_3 = \phi$

Hence I_1 and I_3 are not independent of each other.
For $I_2 \| I_3$,

$R_2 \cap W_3 = \phi$
$R_3 \cap W_2 = \phi$

$W_3 \cap W_2 = \phi$

Hence, I_2 and I_3 are independent of each other.
Thus, I_1 and I_2, I_2 and I_3 are parallelizable but I_1 and I_3 are not.

2.6.3 Parallelism based on Grain size

Grain size: Grain size or Granularity is a measure which determines how much computation is involved in a process. Grain size is determined by counting the number of instructions in a program segment. The following types of grain sizes have been identified (shown in *Figure 17*):

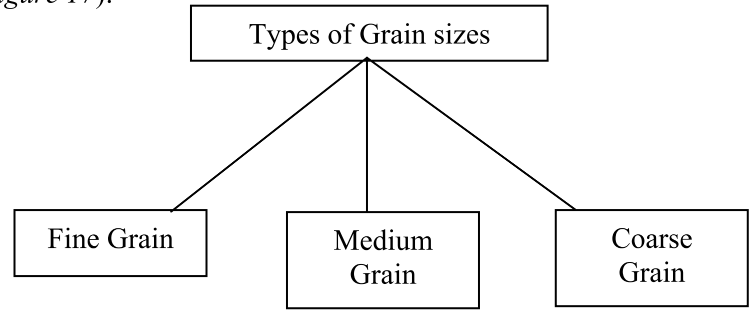


Figure 17: Types of Grain sizes

1) **Fine Grain:** This type contains approximately less than 20 instructions.
2) **Medium Grain:** This type contains approximately less than 500 instructions.
3) **Coarse Grain:** This type contains approximately greater than or equal to one thousand instructions.

Based on these grain sizes, parallelism can be classified at various levels in a program. These parallelism levels form a hierarchy according to which, lower the level, the finer is the granularity of the process. The degree of parallelism decreases with increase in level. Every level according to a grain size demands communication and scheduling overhead. Following are the parallelism levels (shown in *Figure 18*):

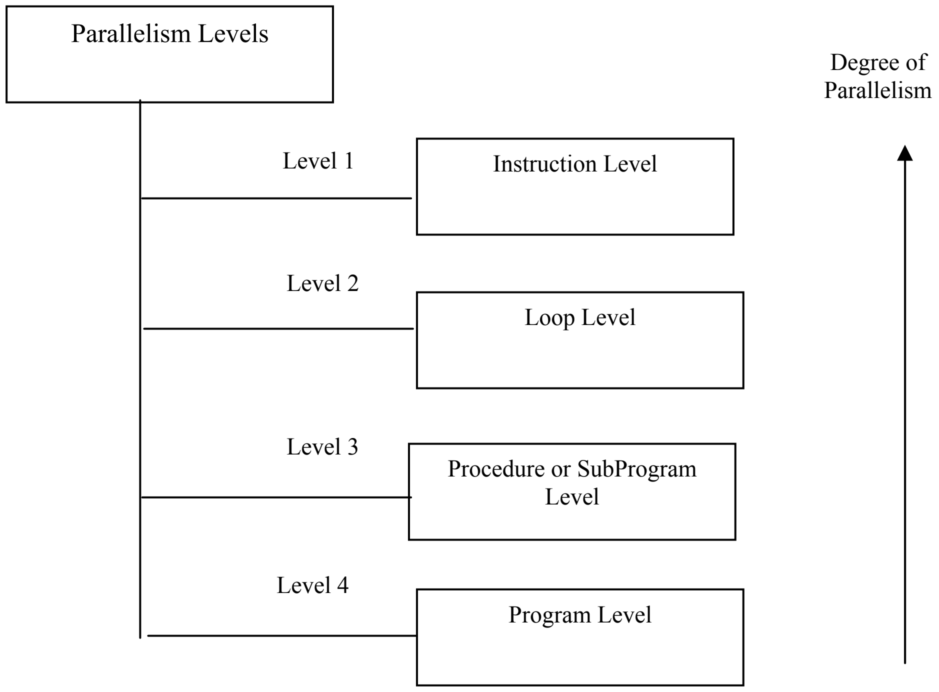


Figure 18: Parallelism Levels

1) **Instruction level:** This is the lowest level and the degree of parallelism is highest at this level. The fine grain size is used at instruction or statement level as only few instructions form the grain size here. The fine grain size may vary according to the type of the program. For example, for scientific applications, the instruction level grain size may be higher. As the higher degree of parallelism can be achieved at this level, the overhead for a programmer will be more.
2) **Loop Level :** This is another level of parallelism where iterative loop instructions can be parallelized. Fine grain size is used at this level also. Simple loops in a program are easy to parallelize whereas the recursive loops are difficult. This type of parallelism can be achieved through the compilers.

3) **Procedure or SubProgram Level:** This level consists of procedures, subroutines or subprograms. Medium grain size is used at this level containing some thousands of instructions in a procedure. Multiprogramming is implemented at this level. Parallelism at this level has been exploited by programmers but not through compilers. Parallelism through compilers has not been achieved at the medium and coarse grain size.

4) **Program Level:** It is the last level consisting of independent programs for parallelism. Coarse grain size is used at this level containing tens of thousands of instructions. Time sharing is achieved at this level of parallelism. Parallelism at this level has been exploited through the operating system.

The relation between grain sizes and parallelism levels has been shown in *Table 1*.

Table 1: Relation between grain sizes and parallelism

Grain Size	Parallelism Level
Fine Grain	Instruction or Loop Level
Medium Grain	Procedure or SubProgram Level
Coarse Grain	Program Level

Coarse grain parallelism is traditionally implemented in tightly coupled or shared memory multiprocessors like the Cray Y-MP. Loosely coupled systems are used to execute medium grain program segments. Fine grain parallelism has been observed in SIMD organization of computers.

Check Your Progress 3

1) Determine the dependency relations among the following instructions:
   I1: a = b+c;
   I2: b = a+d;
   I3: e = a/ f;
   ………………………………………………………………………………………
   ………………………………………………………………………………………
   ………………………………………………………………………………………
   ……………………………………………………………………

2) Use Bernstein's conditions for determining the maximum parallelism between the instructions in the following segment:
   S1: X = Y + Z
   S2: Z = U + V
   S3: R = S + V
   S4: Z = X + R
   S5: Q = M + Z

……………………………………………………………………………………
……………………………………………………………………………………
……………………………………………………………………………………
……………………………………………………………………………

3) Discuss instruction level parallelism.

……………………………………………………………………………………
……………………………………………………………………………………
……………………………………………………………………………………
……………………………………………………………………………

2.7 SUMMARY

In section 2.3, we discussed Flynn's Classification of computers. This classification scheme was suggested by Michael Flynn in 1972 and is based on the concepts of data stream and instruction stream. Next, we discuss Handler's classification scheme in section 2.4. This classification scheme, suggested by Wolfgang Handler in 1977, addresses the computers at the following three distinct levels:

- Processor Control Unit (PCU)
- Arithmetic Logic Unit (ALU)
- Bit-Level Circuit (BLC)

In section 2.5, in context of structural classification of computers, a number of new concepts are introduced and discussed. The concepts discussed include: Tightly Coupled (or shared memory) systems, loosely coupled (or distributed memory) systems. In the case of distributed memory systems, different types of Processor Interconnection Networks (PIN) are discussed. Another classification scheme based on the concept of grain size is discussed in section 2.6.

2.8 SOLUTIONS / ANSWERS

Check Your Progress 1

1) The following criteria have been identified for classifying parallel computers:

   - Classification based on instruction and data streams
   - Classification based on the structure of computers
   - Classification based on how the memory is accessed
   - Classification based on grain size

2) The flow of instructions from the main memory to the CPU is called instruction stream and a flow of operands between processor and memory bi-directionally is known as data stream.

3) a) F
   b) F
   c) F
   d) T

Check Your Progress 2

1) The following rules and operators are used to show the relationship between various elements of the computer:
   - The '*' operator is used to indicate that the units are pipelined or macro-pipelined with a stream of data running through all the units.

- The '+' operator is used to indicate that the units are not pipelined but work on independent streams of data.
- The 'v' operator is used to indicate that the computer hardware can work in one of several modes.
- The '~' symbol is used to indicate a range of values for any one of the parameters.
- Peripheral processors are shown before the main processor using another three pairs of integers. If the value of the second element of any pair is 1, it may be omitted for brevity.

1) The base for structural classification is multiple processors with memory being globally shared between processors or all the processors have their local copy of the memory.
1) When multiprocessors communicate through the global shared memory modules then this organization
is called shared memory computer or tightly coupled systems . When every processor in a multiprocessor system, has its own local memory and the processors communicate via messages transmitted between their local memories, then this organization is called distributed memory computer or loosely coupled system.
1) In UMA, each processor has equal access time to shared memory. In NUMA, local memories are connected with every processor and one reference to a local memory of the remote processor is not uniform. In COMA, all local memories of NUMA are replaced with cache memories.

Check Your Progress 3

1) Instructions I1 and I2 are both flow dependent and antidependent both. Instruction I2 and I3 are output dependent and instructions I1 and I3 are independent.

2) $R_1 = \{Y,Z\}$ $W_1 = \{X\}$
$R_2 = \{U,V\}$ $W_2 = \{Z\}$
$R_3 = \{S,V\}$ $W_3 = \{R\}$
$R_4 = \{X,R\}$ $W_4 = \{Z\}$
$R_5 = \{M,Z\}$ $W_5 = \{Q\}$

Thus, S1, S3 and S5 and S2 & S4 are parallelizable.

3) This is the lowest level and the degree of parallelism is highest at this level. The fine grain size is used at instruction or statement level as only few instructions form the grain size here. The fine grain size may vary according to the type of the program. For example, for scientific applications, the instruction level grain size may be higher. The loops As the higher degree of parallelism can be achieved at this level, the overhead for a programmer will be more.

UNIT 3 INTERCONNECTION NETWORK

Structure **Page Nos.**

3.0 INTRODUCTION

This unit discusses the properties and types of interconnection networks. In multiprocessor systems, there are multiple processing elements, multiple I/O modules, and multiple memory modules. Each processor can access any of the memory modules and any of the I/O units. The connectivity between these is performed by interconnection networks.

Thus, an interconnection network is used for exchanging data between two processors in a multistage network. Memory bottleneck is a basic shortcoming of Von Newman architecture. In case of multiprocessor systems, the performance will be severely affected in case the data exchange between processors is delayed. The multiprocessor system has one global shared memory and each processor has a small local memory. The processors can access data from memory associated with another processor or from shared memory using an interconnection network. Thus, interconnection networks play a central role in determining the overall performance of the multiprocessor systems. The interconnection networks are like customary network systems consisting of nodes and edges. The nodes are switches having few input and few output (say n input and m output) lines. Depending upon the switch connection, the data is forwarded from input lines to output lines. The interconnection network is placed between various devices in the multiprocessor network.

The architecture of a general multiprocessor is shown in *Figure 1*. In the multiprocessor systems, these are multiple processor modules (each processor module consists of a processing element, small sized local memory and cache memory), shared global memory and shared peripheral devices.

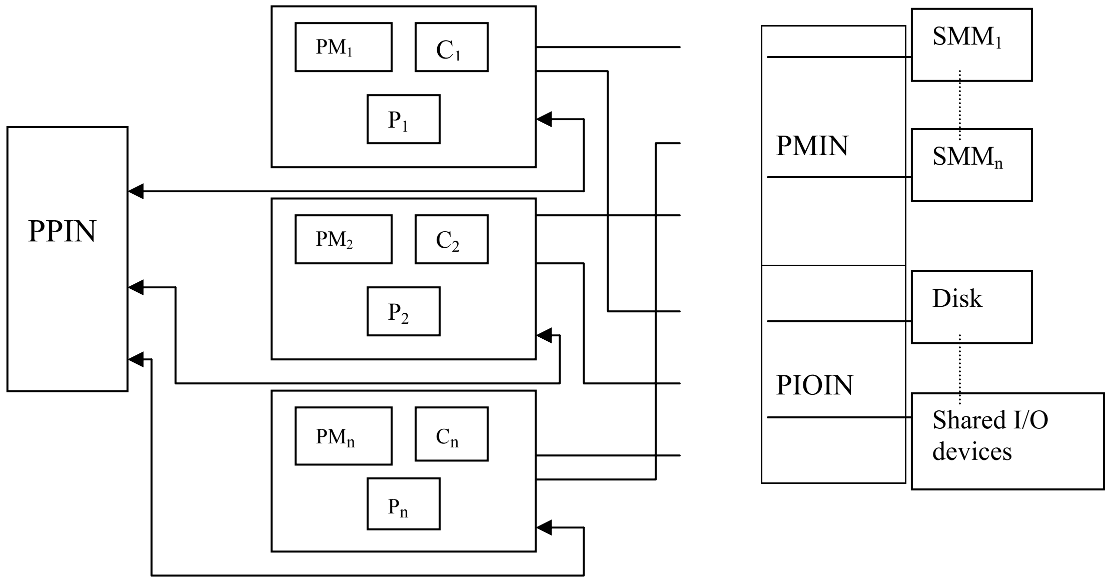


Figure 1: General Multi-Processor

PMIN = Processor to Memory Interconnection Network
PIOIN= Processor to I/O Interconnection Network
PPIN = Processor to Processor Interconnection Network
PM = Processor Module

Module communicates with other modules shared memory and peripheral devices using interconnection networks.

3.1 OBJECTIVES

After studying this unit, students will be able to understand

- discuss the meaning and needs of interconnection network;
- describe the role of interconnection network in a multiprocessor system;
- enumerate the types of interconnection network;
- explain the concept of permutation network;
- discuss the various interconnection networks, and
- describe how matrix multiplication can be carried out on an interconnection network.

3.2 NETWORK PROPERTIES

The following properties are associated with interconnection networks.

1) *Topology:* It indicates how the nodes a network are organised. Various topologies are discussed in Section 3.5.

2) *Network Diameter*: It is the minimum distance between the farthest nodes in a network. The distance is measured in terms of number of distinct hops between any two nodes.

3) *Node degree*: Number of edges connected with a node is called node degree. If the edge carries data from the node, it is called out degree and if this carries data into the node it is called in degree.

4) *Bisection Bandwidth:* Number of edges required to be cut to divide a network into two halves is called bisection bandwidth.

5) *Latency:* It is the delay in transferring the message between two nodes.

6) *Network throughput:* It is an indicative measure of the message carrying capacity of a network. It is defined as the total number of messages the network can transfer per unit time. To estimate the throughput, the capacity of the network and the messages number of actually carried by the network are calculated. Practically the throughput is only a fraction of its capacity.
In interconnection network the traffic flow between nodes may be nonuniform and it may be possible that a certain pair of nodes handles a disproportionately large amount of traffic. These are called "hot spot." The hot spot can behave as a bottleneck and can degrade the performance of the entire network.

7) *Data Routing Functions:* The data routing functions are the functions which when executed establishe the path between the source and the destination. In dynamic interconnection networks there can be various interconnection patterns that can be generated from a single network. This is done by executing various data routing functions. Thus data routing operations are used for routing the data between various processors. The data routing network can be static or dynamic static network

8) *Hardware Cost:* It refers to the cost involved in the implementation of an interconnection network. It includes the cost of switches, arbiter unit, connectors, arbitration unit, and interface logic.

9) *Blocking and Non-Blocking network:* In non-blocking networks the route from any free input node to any free output node can always be provided. Crossbar is an example of non-blocking network. In a blocking network simultaneous route establishment between a pair of nodes may not be possible. There may be situations where blocking can occur. Blocking refers to the situation where one switch is required to establish more than one connection simultaneously and end-to-end path cannot be established even if the input nodes and output nodes are free. The example of this is a blocking multistage network.

10) *Static and Dynamic Interconnection Network:* In a static network the connection between input and output nodes is fixed and cannot be changed. Static interconnection network cannot be reconfigured. The examples of this type of network are linear array, ring, chordal ring, tree, star, fat tree, mesh, tours, systolic arrays, and hypercube. This type of interconnection networks are more suitable for building computers where the communication pattern is more or less fixed, and can be implemented with static connections. In dynamic network the interconnection pattern between inputs and outputs can be changed. The interconnection pattern can be reconfigured according to the program demands. Here, instead of fixed connections, the switches or arbiters are used. Examples of such networks are buses, crossbar switches, and multistage networks. The dynamic networks are normally used in shared memory(SM) multiprocessors.

11) *Dimensionality of Interconnection Network:* Dimensionality indicates the arrangement of nodes or processing elements in an interconnection network. In single dimensional or linear network, nodes are connected in a linear fashion; in two dimensional network the processing elements (PE's) are arranged in a grid and in cube network they are arranged in a three dimensional network.

12) *Broadcast and Multicast :*In the broadcast interconnection network, at one time one node transmits the data and all other nodes receive that data. Broadcast is one to all mapping. It is the implementation achieved by SIMD computer systems. Message passing multi-computers also have broadcast networks. In multicast network many nodes are simultaneously allowed to transmit the data and multiple nodes receive the data.

3.3 DESIGN ISSUES OF INTERCONNECTION NETWORK

The following are the issues, which should be considered while designing an interconnection network.

1) *Dimension and size of network:* It should be decided how many PE's are there in the network and what the dimensionality of the network is i.e. with how many neighburs, each processor is connected.
2) *Symmetry of the network:* It is important to consider whether the network is symmetric or not i.e., whether all processors are connected with same number of processing elements, or the processing elements of corners or edges have different number of adjacent elements.

3) *What is data communication strategy?* Whether all processors are communicating with each other in one time unit synchronously or asynchronously on demand basis.

4) *Message Size:* What is message size? How much data a processor can send in one time unit.

5) *Start up time:* What is the time required to initiate the communication process.

6) *Data transfer time:* How long does it take for a message to reach to another processor. Whether this time is a function of link distance between two processors or it depends upon the number of nodes coming in between.

7) *The interconnection network is static or dynamic:* That means whether the configuration of interconnection network is governed by algorithm or the algorithm allows flexibility in choosing the path.

Check Your Progress 1

1) Define the following terms related with interconnection networks.
   i) Node degrees
   ii) Dynamic connection network
   iii) Network diameter

…………………………………………………………………………………………
…………………………………………………………………………………………
…………………………………………………………………………………………

2) What is the significance of a bisection bandwidth?

……………………………………………………………………………………………

……………………………………………………………………………………………

……………………………………………………………………………………………

3.4 VARIOUS INTERCONNECTION NETWORKS

In this section, we will discuss some simple and popularly used interconnection networks

1) Fully connected: This is the most powerful interconnection topology. In this each node is directly connected to all other nodes. The shortcoming of this network is that it requires too many connections.

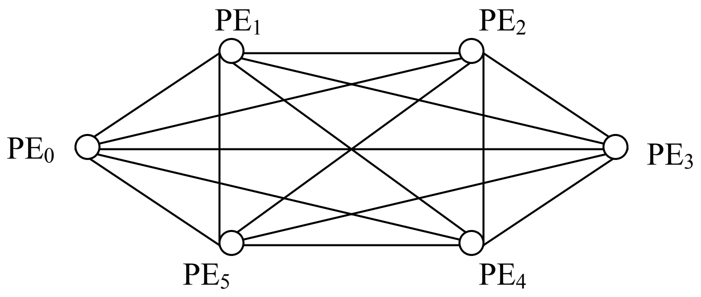


Figure 2: Fully connected interconnection topology

2) Cross Bar: The crossbar network is the simplest interconnection network. It has a two dimensional grid of switches. It is a non-blocking network and provides connectivity between inputs and outputs and it is possible to join any of the inputs to any output.

An N * M crossbar network is shown in the following *Figure 3 (a)* and switch connections are shown in *Figure 3 (b)*.

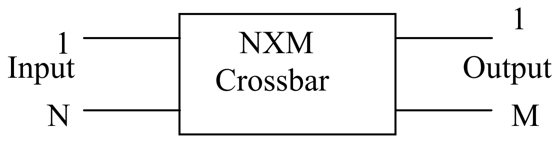


Figure: 3(a)

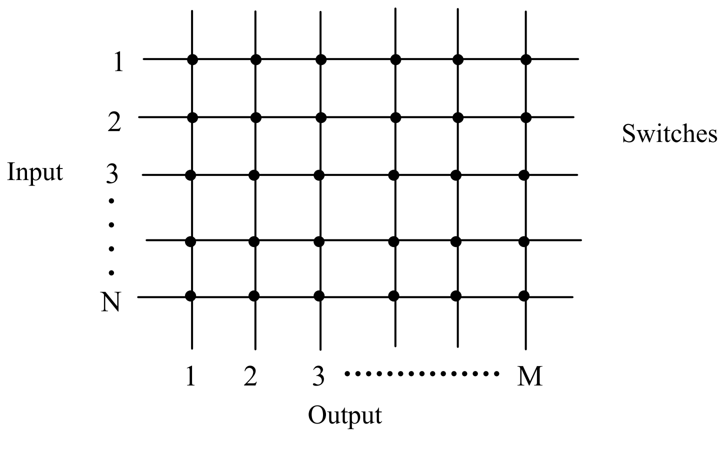


Figure: 3(b)
Figure 3: Crossbar Network

A switch positioned at a cross point of a particular row and particular column. connects that particular row (input) to column (output).

The hardware cost of N*N crossbar switch is proportional to N^2. It creates delay equivalent to one switching operation and the routing control mechanism is easy. The crossbar network requires N^2 switches for N input and N output network.

3) *Linear Array:* This is a most fundamental interconnection pattern. In this processors are connected in a linear one-dimensional array. The first and last processors are connected with one adjacent processor and the middle processing elements are connected with two adjacent processors. It is a one-dimensional interconnection network.

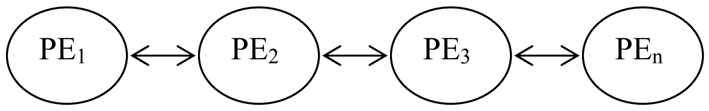


Figure 4: Linear Array

4) *Mesh:* It is a two dimensional network. In this all processing elements are arranged in a two dimensional grid. The processor in rows *i* and column *j* are denoted by PE_i.

The processors on the corner can communicate to two nearest neighbors i.e. PE_{00} can communicate with PE_{01} and PE_{10}. The processor on the boundary can communicate to 3 adjacent processing elements i.e. PE_{01} can communicate with PE_{00}, PE_{02} and PE_{11} and internally placed processors can communicate with 4 adjacent processors i.e. PE_{11} can communicate with PE_{01}, PE_{10}, PE_{12}, and PE_{21}

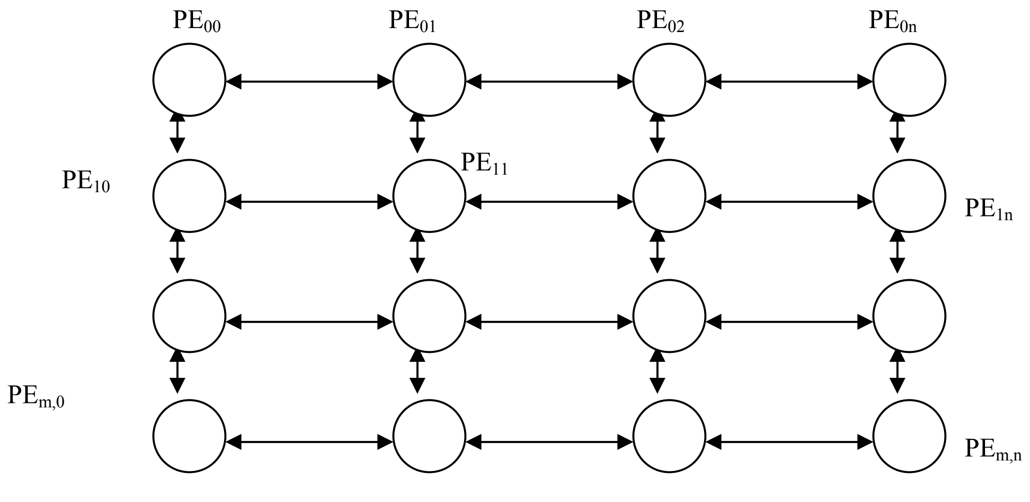


Figure 5: Mesh Network

5) *Ring:* This is a simple linear array where the end nodes are connected. It is equivalent to a mesh with wrap around connections. The data transfer in a ring is normally one direction. Thus, one drawback to this network is that some data transfer may require N/2 links to be traveled (like nodes 2 & 1) where N is the total number of nodes.

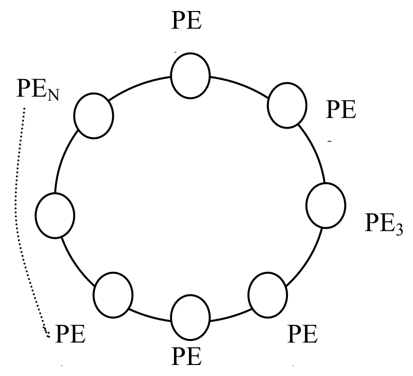


Figure 6: Ring network

6) Torus: The mesh network with wrap around connections is called Tours Network.

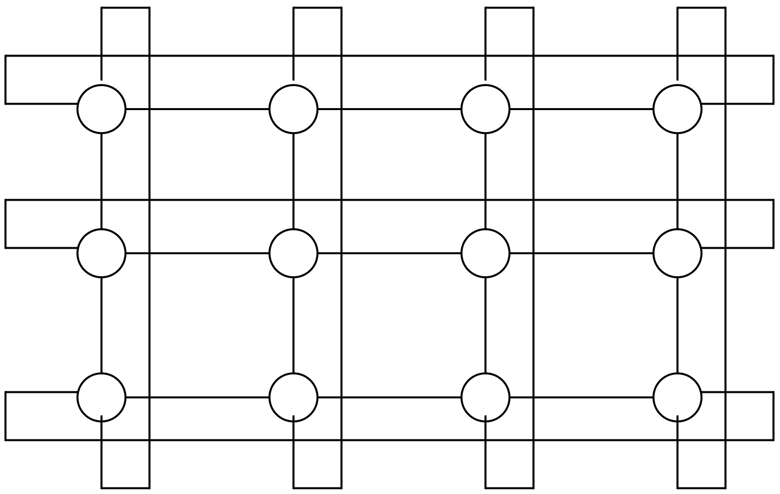


Figure 7: Torus network

7) Tree interconnection network: In the tree interconnection network, processors are arranged in a complete binary tree pattern.

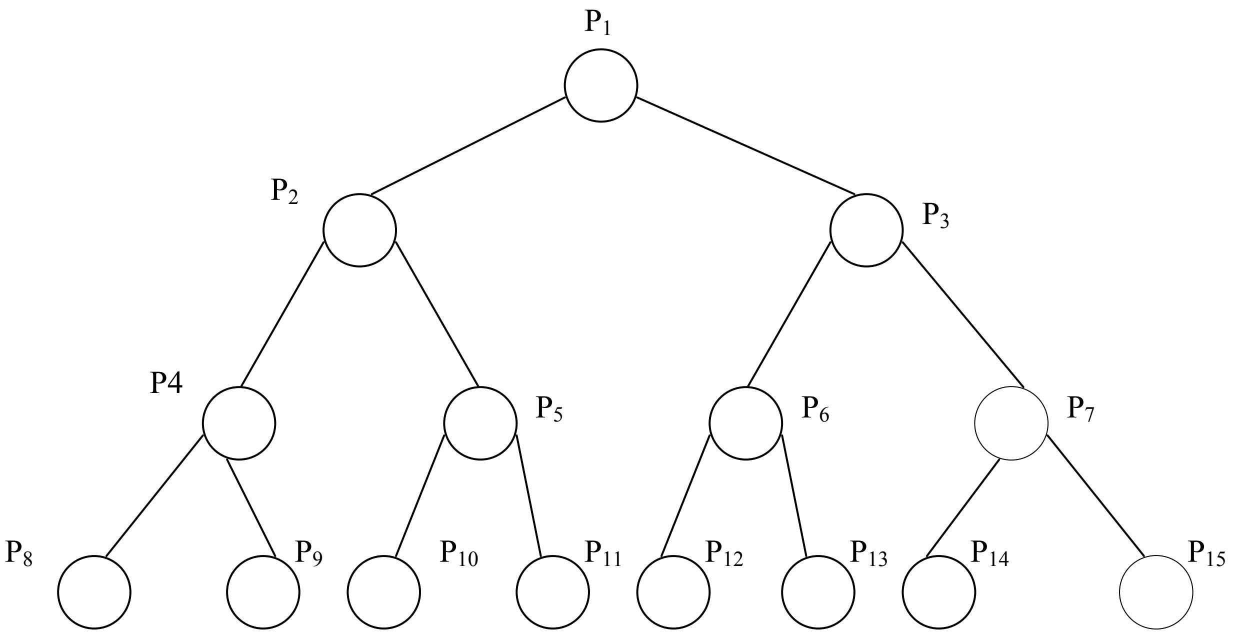


Figure 8:Tree interconnection network

8) *Fat tree:* It is a modified version of the tree network. In this network the bandwidth of edge (or the connecting wire between nodes) increases towards the root. It is a more realistic simulation of the normal tree where branches get thicker towards root. It is the more popular as compared to tree structure, because practically the more traffic occurs towards the root as compared to leaves, thus if bandwidth remains the same the root will be a bottleneck causing more delay. In a tree this problem is avoided because of higher bandwidth.

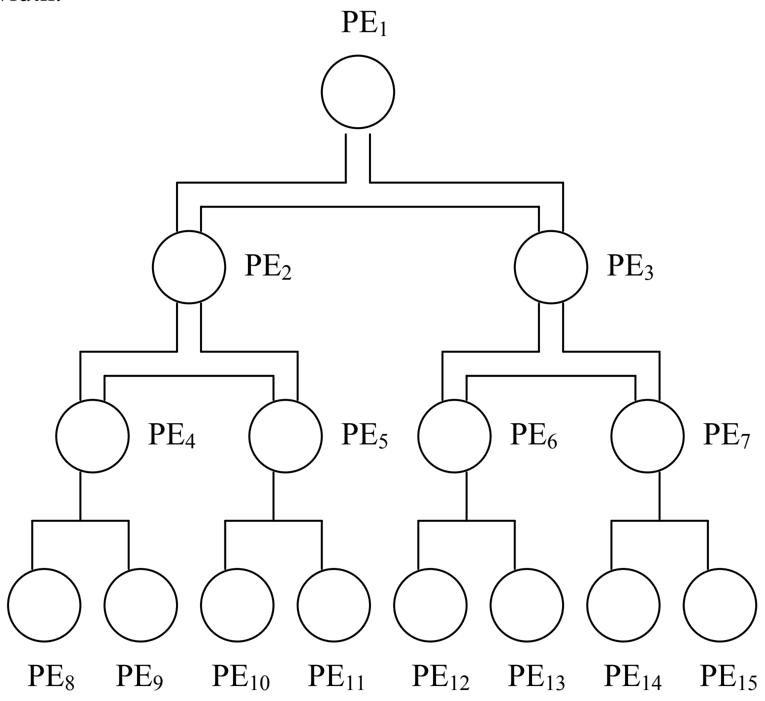


Figure 9: Fat tree

9) *Systolic Array:* This interconnection network is a type of pipelined array architecture and it is designed for multidimensional flow of data. It is used for implementing fixed algorithms. Systolic array designed for performing matrix multiplication is shown below. All interior nodes have degree 6.

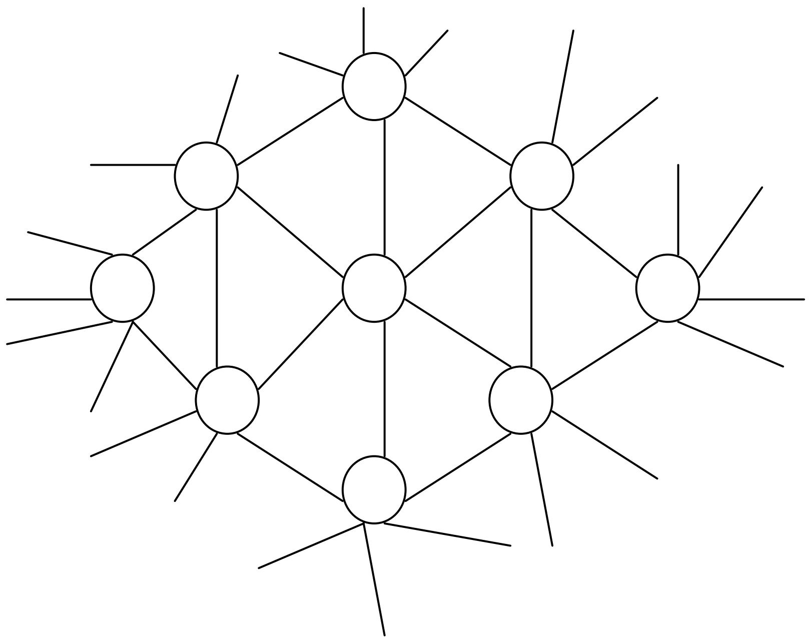


Figure 10: Systolic Array

10) Cube: It is a 3 dimensional interconnection network. In this the PE's are arranged in a cube structure.

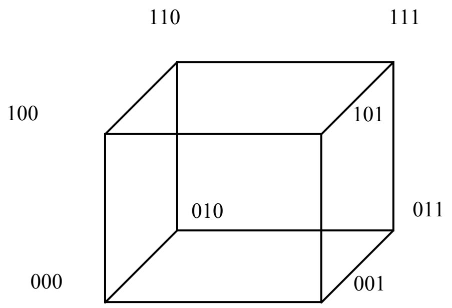


Figure 11: Cube interconnection network

11) Hyper Cube: A Hypercube interconnection network is an extension of cube network. Hypercube interconnection network for n ≥ 3, can be defined recursively as follows:

For n = 3, it cube network in which nodes are assigned number 0, 1, ……,7 in binary. In other words, one of the nodes is assigned a label 000, another one as 001…. and the last node as 111.
Then any node can communicate with any other node if their labels differ in exactly one place, e.g., the node with label 101 may communicate directly with 001, 000 and 111.

For n > 3, a hypercube can be defined recursively as follows:
Take two hypercubes of dimension (n – 1) each having (n –1) bits labels as 00….0, ……11…..1

Next join the two nodes having same labels each (n – 1) -dimension hypercubes and join these nodes. Next prefix '1' the labels of one of the (n – 1) dimensional hypercube and '0' to the labels of the other hypercube. This completes the structure of n-dimensional hypercube. Direct connection is only between that pair of nodes which has a (solid) line connecting the two nodes in the pair.

For n = 4 we draw 4-dimensional hypercube as show in *Figure 12*:

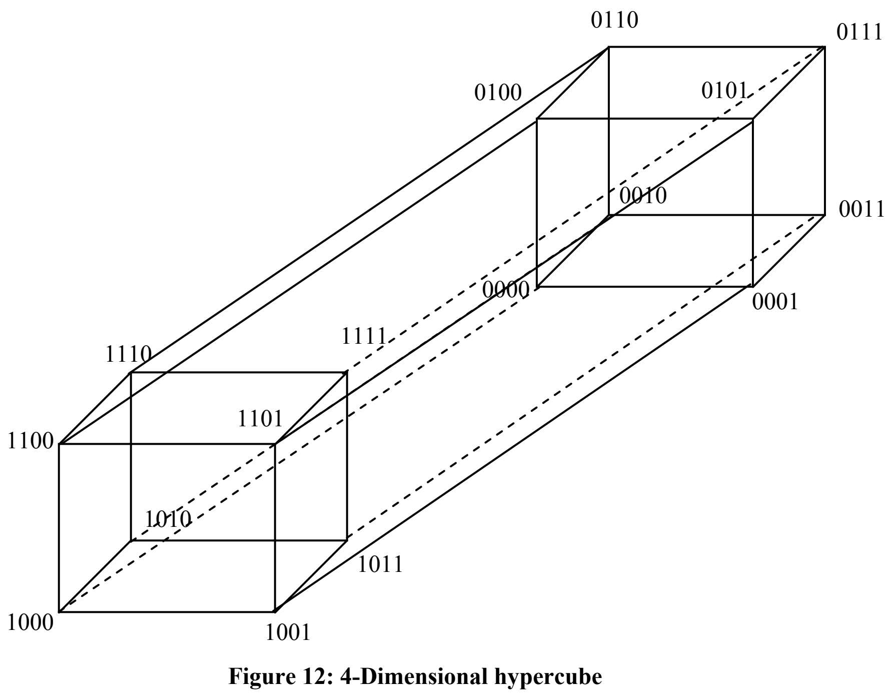


Figure 12: 4-Dimensional hypercube

3.5 CONCEPT OF PERMUTATION NETWORK

In permutation interconnection networks the information exchange requires data transfer from input set of nodes to output set of nodes and possible connections between edges are established by applying various permutations in available links. There are various networks where multiple paths from source to destination are possible. For finding out what the possible routes in such networks are the study of the permutation concept is a must.

Let us look at the basic concepts of permutation with respect to interconnection network. Let us say the network has set of n input nodes and n output nodes.
Permutation P for a network of 5 nodes (i.e., n = 5) is written as follows:

$$P = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ \hline 5 & 4 & 1 & 3 & 2 \end{bmatrix}$$

It means node connections are 1↔5, 2↔4, 3↔1, 4↔3, 5↔2.

The connections are shown in the *Figure 13*.

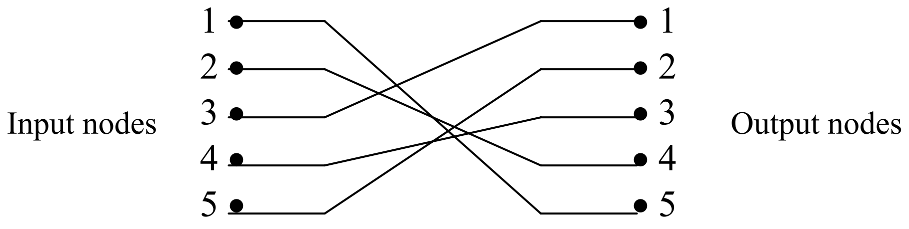


Figure 13: Node-Connections

The other permutation of the same set of nodes may be

$$P = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ \hline 2 & 3 & 5 & 1 & 4 \end{bmatrix}$$

Which means connections are: 1↔2, 2↔3, 3↔5, 4↔1, and 5↔4
Similarly, other permutations are also possible. The Set of all permutations of a 3-node network will be

$$P= \begin{bmatrix} 1 & 2 & 3 \\ 1 & 3 & 2 \end{bmatrix}, \begin{bmatrix} 1 & 2 & 3 \\ 2 & 1 & 3 \end{bmatrix}, \begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 2 & 3 \\ 3 & 1 & 2 \end{bmatrix}, \begin{bmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \end{bmatrix}$$

Connection, $\begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix}$ indicates connection from node 1 to node1, node 2 to node 2, and node 3 to node 3, hence it has no meaning, so it is dropped.

In these examples, only one set of links exist between input and output nodes and means it is a single stage network. It may be possible that there exist multiple links between input and output (i.e. multistage network). Permutation of all these in a multistage network are called permutation group and these are represented by a cycle e.g. permutation

P= (1,2,3) (4,5) means the network has two groups of input and output nodes, one group consists of nodes 1,2,3 and another group consists of nodes 4,5 and connections are 1→2, 2→3, 3→1, and 4→5. Here group (1,2,3) has period 3 and (4,5) has period 2, collectively these groups has periodicity 3×2=6.

Interconnection from all the possible input nodes to all the output nodes forms the permutation group.

The permutations can be combined. This is called composition operation. In composition operation two or more permutations are applied in sequence, e.g. if P_1 and P_2 are two permutations defined as follows:

$$P_1 = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 1 & 3 \end{bmatrix}, \quad P_2 \begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \end{bmatrix}$$

The composition of P_1 and P_2 will be

$$P_1 . P_2 = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 1 & 3 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \end{bmatrix}$$

$$P_1 . P_2 = \begin{bmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \end{bmatrix}$$

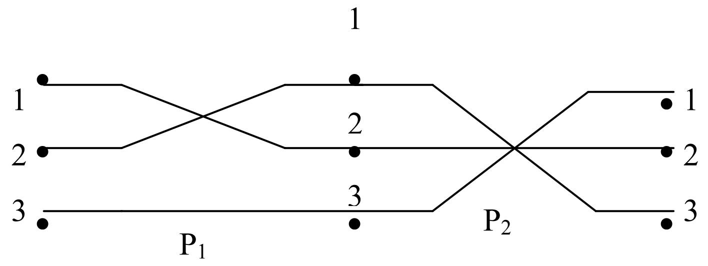


Figure 14 (a)

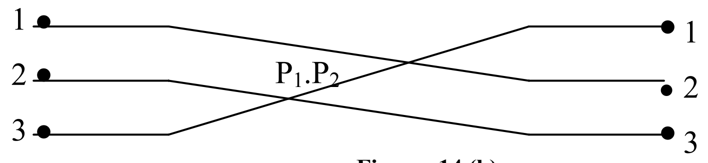


Figure: 14 (b)

Similarly, if $P_3 = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 4 & 2 & 1 & 3 & 5 \end{bmatrix}$ and $P_4 = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 3 & 1 & 5 & 2 & 4 \end{bmatrix}$

then $P_3 . P_4 = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 1 & 3 & 5 & 4 \end{bmatrix}$

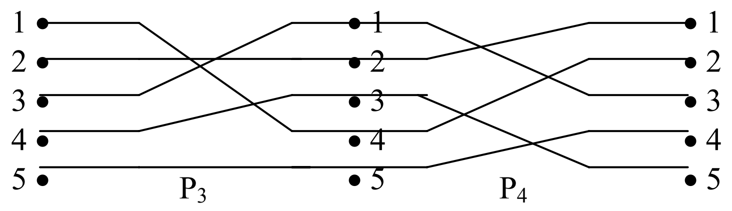
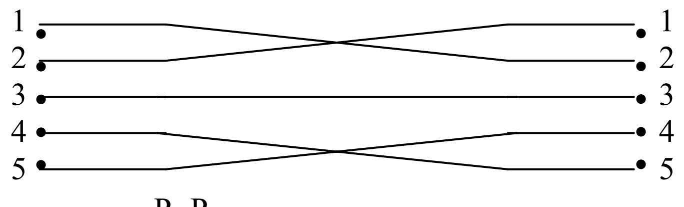


Figure: 15



$P_3.P_4$

Figure: 16

Composition of those permutations P_1 P_2 are represented in Figures 14 (a) and 14 (b)

There are few permutations of special significance in interconnection network. These permutations are provided by hardware. Now, let us discuss these permutations in detail.

1) **Perfect Shuffle Permutation:** This was suggested by Harold Stone (1971). Consider N objects each represented by n bit number say X_{n-1}, X_{n-2}, X_0 (N is chosen such that N = 2n.) The perfect shuffle of these N objects is expressed as

$$X_{n-1}, X_{n-2}, X_0 \quad = X_{n-2}, X_0 \quad X_{n-1.}$$

That, means perfect shuffle is obtained by rotating the address by 1 bit left. e.g. shuffle of 8 objects is shown as

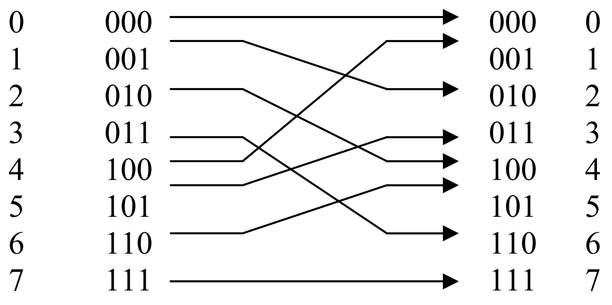


Figure 17: Shuffle of 8 objects

2) **Butterfly permutation:** This permutation is obtained by interchanging the most significant bit in address with least significant bit.

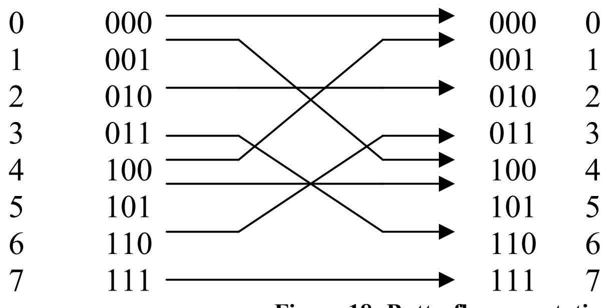


Figure 18: Butterfly permutation

e.g. $X_{n-1,}$ $X_{n-2,}$ and $X_1.X_0 = X_0$ X_{n-2} ……. X_1 X_{n-1}

001↔100, 010↔010
011↔110,

An interconnection network based on this permutation is the butterfly network. A butterfly network is a blocking network and it does not allow an arbitrary connection of

N inputs to N outputs without conflict. The butterfly network is modified in Benz network. The Benz network is a non-blocking network and it is generated by joining two butterfly networks back to back, in such a manner that data flows forward through one and in reverse through the other.

3) **Clos network:** This network was developed by Clos (1953). It is a non-blocking network and provides full connectivity like crossbar network but it requires significantly less number of switches. The organization of Clos network is shown in *Figure 19*:

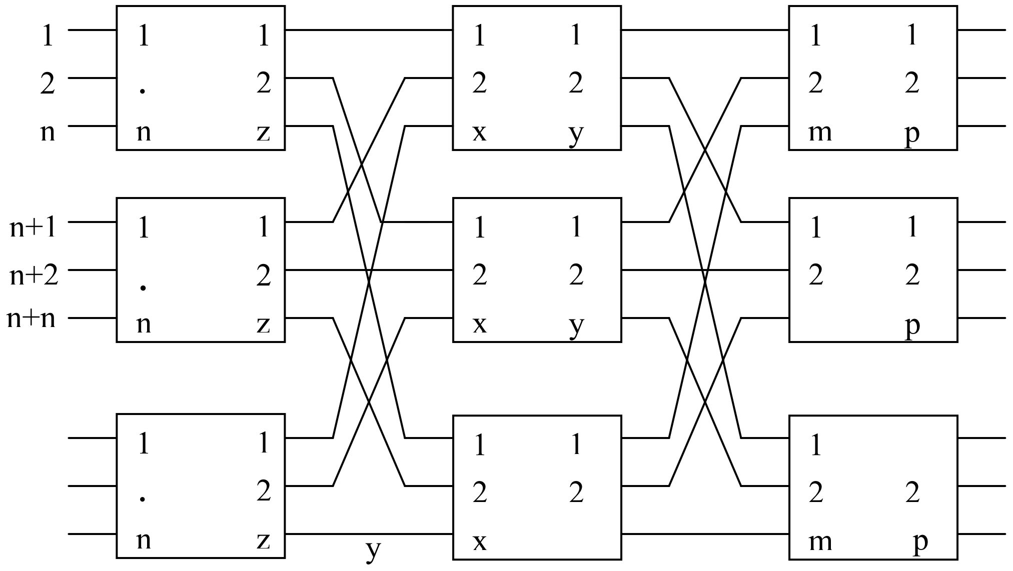


Figure 19: Organisation of Clos network

Consider an I input and O output network
Number N is chosen such that (I= n.x) and (O=p.y).

In Clos network input stage will consist of X switches each having n input lines and z output lines. The last stage will consist of Y switches each having m input lines and p output lines and the middle stage will consist of z crossbar switches, each of size X × Y. To utilize all inputs the value of Z is kept greater than or equal to n and p.

The connection between various stages is made as follows: all outputs of 1st crossbar switch of first stage are joined with 1st input of all switches of middle stage. (i.e., 1st output of first page with 1st middle stage, 2nd output of first stage with 1st input of second switch of middle stage and so on…)

The outputs of second switch of first stage. Stage are joined with 2nd input of various switches of second stage (i.e., 1st output of second switch of 1st stage is joined with 2 input of 1st switch of middle stage and 2nd output of 2nd switch of 1st stage is joined with 2nd input of 2nd switch of middle stage and so on…

Similar connections are made between middle stage and output stage (i.e. outputs of 1st switch of middle stage are connected with 1st input of various switches of third stage.

Permutation matrix of P in the above example the matrix entries will be n

Bens Network: It is a non-blocking network. It is a special type of Clos network where first and last stage consists of 2×2 switches (for n input and m output network it will have n/2 switches of 2 ×2 order and the last stage will have m/2 switch of 2×2 order the middle stage will have two n/2 X m/2 switches. Numbers n and m are assumed to be the power of 2.

Thus, for 16×16 3-stage Bens network first stage and third stage will consist of 8 (2×2) switches and middle stage will consist of 2 switches of size (8×8). The connection of crossbar will be as follows:

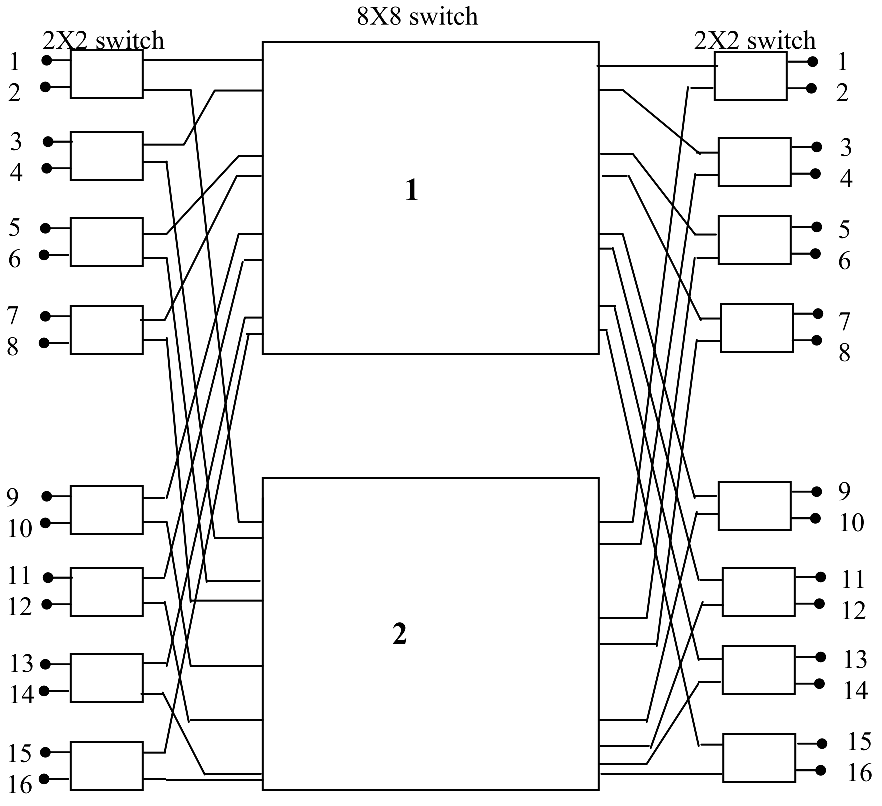


Figure 20: 16X16 3-stage benz network

The switches of various stages provide complete connectivity. Thus by properly configuring the switch any input can be passed to any output.

Consider a Clos network with 3 stages and 3×3 switches in each stage.

Permutation $\begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 8 & 7 & 6 & 2 & 5 & 3 & 9 & 4 & 1 \end{bmatrix}$

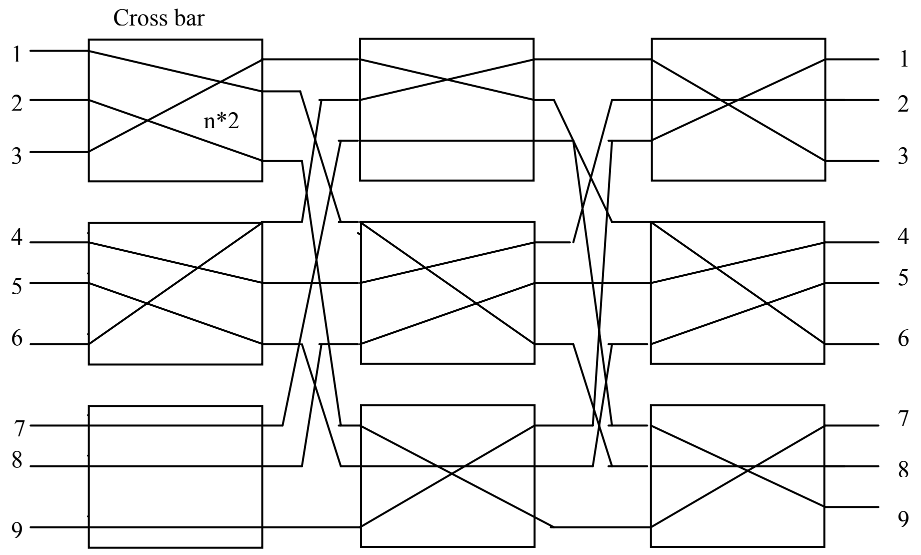


Figure 21: Clos Network

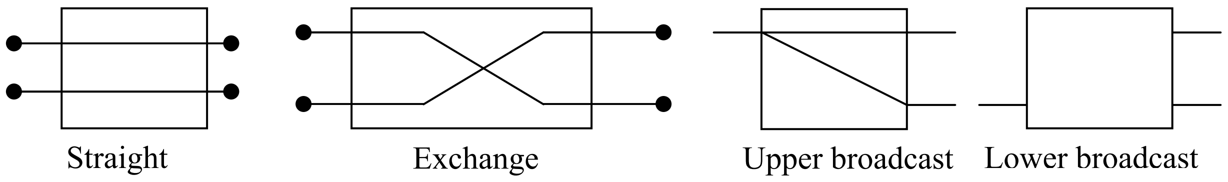
The implementation of this above permutation is shown in *Figure 20*. This permutation can be represented by the following matrix also,

Input \ Output	1	2	3	4	5	6	7	8	9
1								X	
2							X		
3						X			
4		X							
5					X				
6			X						X
7									
8				X					
9	X								

Figure 22: Permutation representation through Matrix

The upper input of all first stage switches will be connected with respective inputs of 1st middle stage switch, and lower input of all first stage switches will be connected with respective inputs of 2nd switch. Similarly, all outputs of 1st switch of middle stage will be connected as upper input of switches at third stage.
In Benz network to reduce the complexity the middle stage switches can recursively be broken into N/4 × N/4 (then N/8 × M/8), till switch size becomes 2×2.

The connection in a 2×2 switch will either be straight, exchange, lower broadcast or upper broadcast as shown in the Figure.



Switch 2x2

The 8×8 Benz network with all switches replaced by a 2×2 is shown in *Figure 23(a)*:

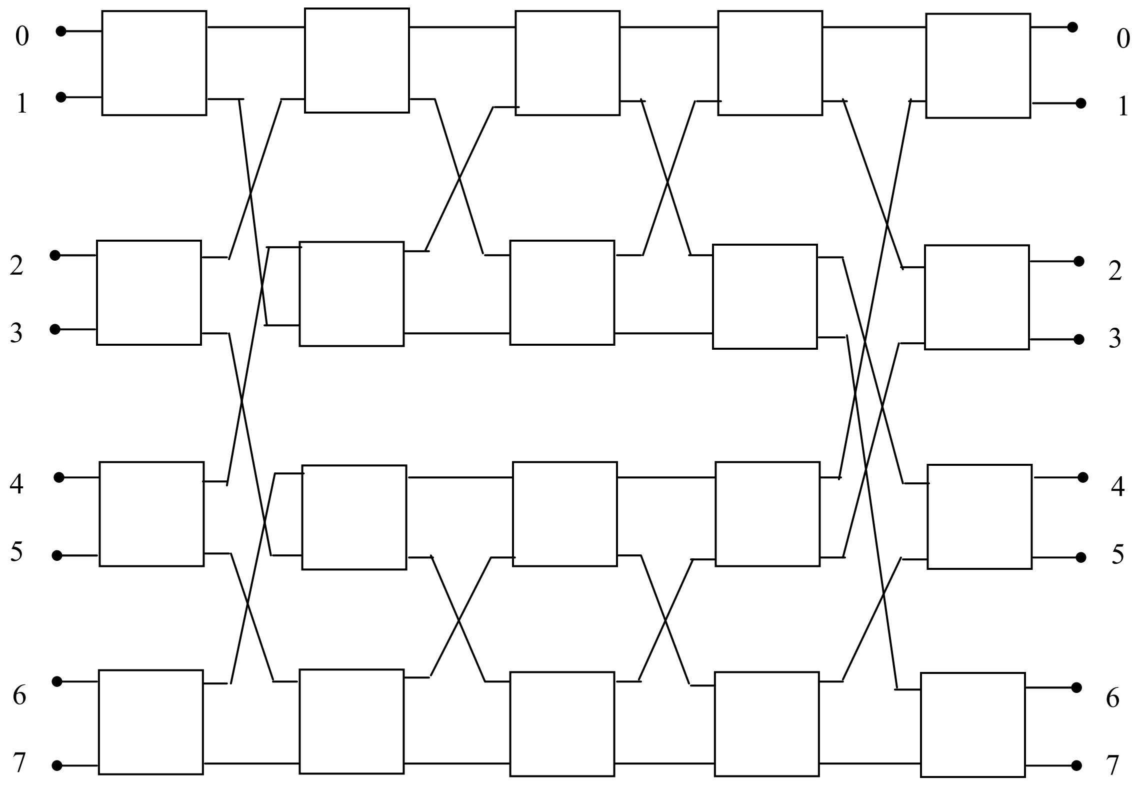


Figure 23 (a): Benz Network

The Bens network connection for permutation

$$P \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 1 & 7 & 3 & 4 & 5 & 6 & 2 & 0 \end{bmatrix}$$

will be as follows:-

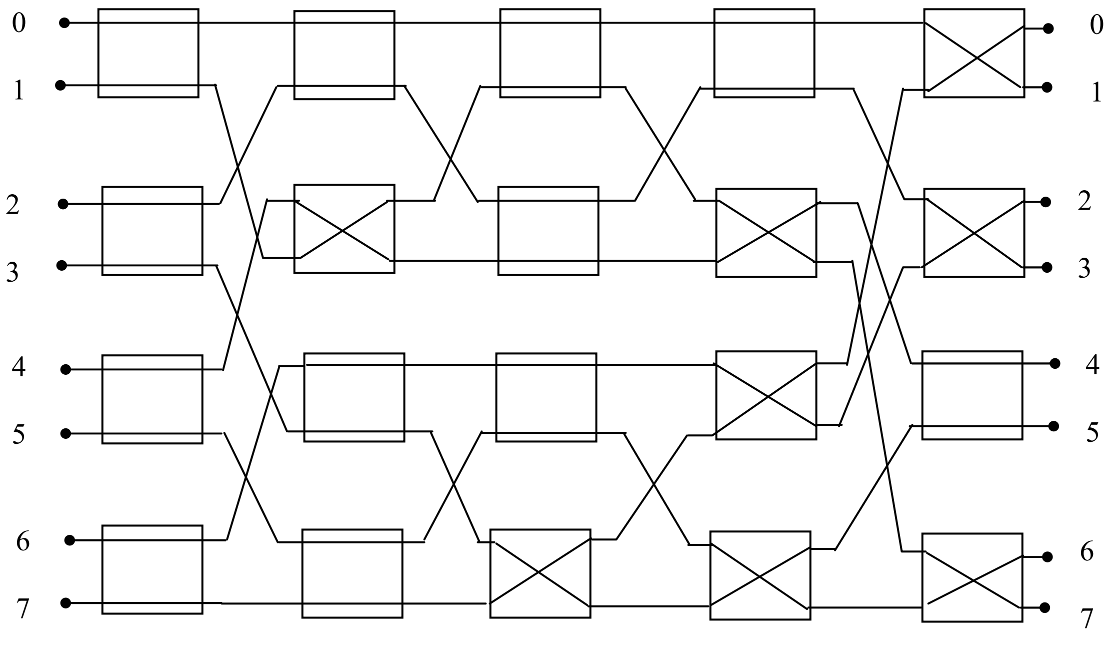


Figure 23 (b): 8X8 BENZ NETWORK OF 4 STAGE

The permutation for P = $\begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ \hline 0 & 7 & 6 & 2 & 5 & 3 & 9 & 1 \end{bmatrix}$ will be as follows

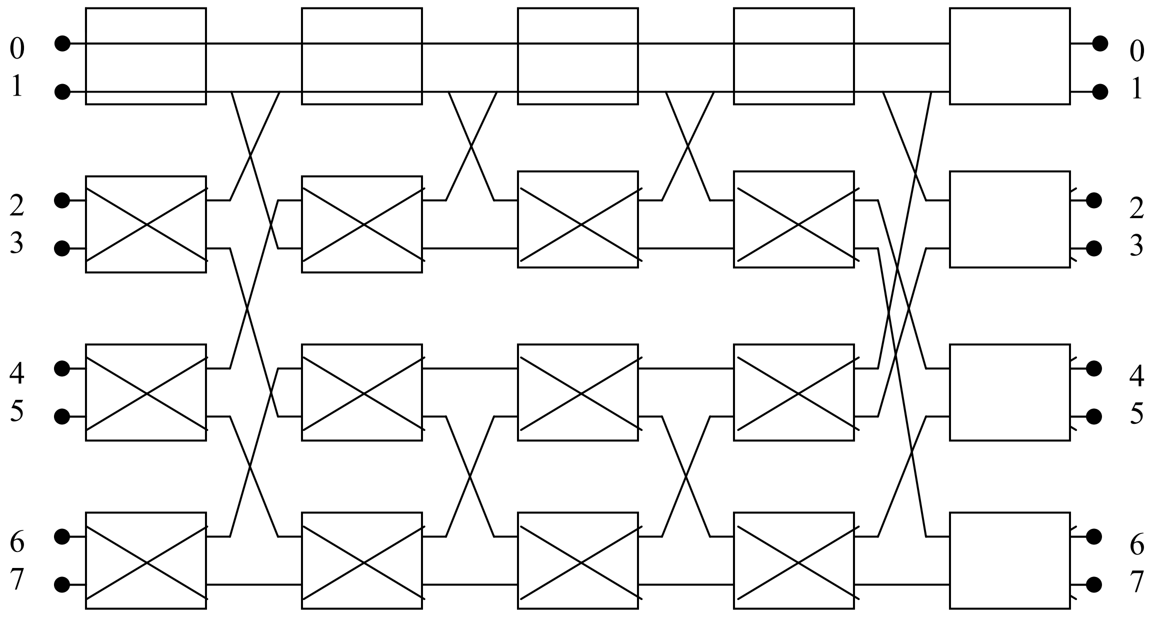


Figure 24: Line number for n

Hardware complexity of Benz Network: - Benz network uses lesser switches and it provides good connectivity. To find hardware complexity of Benz network let us assume that

$$N = 2^n \Rightarrow n = \log_2 N$$

Number of stages in N input Benz network $= 2n-1 = 2\log_2 N - 1$

Number of 2X2 switches in each stage = N/2.

Total number of cells in network = $(N/2)(2\log_2 N-1) = N\log_2 N - N/2$

Number of switches in different networks for various inputs is shown in the following table:

Input	No. of cross bars	Switches Clos Net	Benz Net
2	4	9	4
8	64	69	80
64	4096	1536	1408
256	65536	12888	7680

Thus we can analyze from this table that for larger inputs the Benz Network is the best as it has the least number of switches.

Shuffle Exchange Network:- These networks are based on the shuffle and exchange operations discussed earlier.

3.6 PERFORMANCE METRICS

The performance of interconnection networks is measured on the following parameters.

1) **Bandwidth:** It is a measure of maximum transfer rate between two nodes. It is measured in Megabytes per second or Gigabytes per second.

2) **Functionality:** It indicates how interconnection networks supports data routing, interrupt handling, synchronization, request/message combining and coherence.

3) **Latency:** In interconnection networks various nodes may be at different distances depending upon the topology. The network latency refers to the worst-case time delay for a unit message when transferred through the network between farthest nodes.

4) **Scalability:** It refers to the ability of interconnection networks for modular expansion with a scalable performance with increasing machine resources.

5) **Hardware Complexity:** It refers to the cost of hardware logic like wires, connectors, switches, arbiter etc. that are required for implementation of interconnection network.

3.7 SUMMARY

This unit deals with various concepts about interconnection network. The design issues of interconnection network, types of interconnection network, permutation network and performance metrics of the interconnection networks are discussed.

3.8 SOLUTIONS/ANSWERS

Check Your Progress 1

1) i) **Node degrees:** Number of edges connected with a node is called node degree. If the edge carries data from the node, it is called out degree and if this carries data into the node it is called in degree.

   ii) **Dynamic connection network:** In dynamic network the interconnection pattern between inputs and outputs can be changed. The interconnection pattern can be reconfigured according to the program demands. Here, instead of fixed connections, the switches or arbiters are used. Examples of such networks are buses, crossbar switches, and multistage networks. The dynamic networks are normally used in shared memory(SM) multiprocessors.

   iii) **Network diameter:** It is the minimum distance between the farthest nodes in a network. The distance is measured in terms of number of distinct hops between any two nodes.

2) Bisection bandwidth of a network is an indicator of robustness of a network in the sense that if the bisection bandwidth is large then there may be more alternative routes between a pair of nodes, any one of the other alternative routes may be chosen. However, the degree of difficulty of dividing a network into smaller networks, is inversely proportional to the bisection bandwidth.

UNIT 4 PARALLEL COMPUTER ARCHITECTURE

Structure **Page Nos.**

4.0 INTRODUCTION

We have discussed the classification of parallel computers and their interconnection networks respectively in units 2 and 3 of this block. In this unit, various parallel architectures are discussed, which are based on the classification of parallel computers considered earlier. The two major parametric considerations in designing a parallel computer architecture are: (i) executing multiple number of instructions in parallel, and (ii) increasing the efficiency of processors. There are various methods by which instructions can be executed in parallel and parallel architectures are based on these methods of executing instructions in parallel. Pipelining is one of the classical and effective methods to increase parallelism where different stages perform repeated functions on different operands. Vector processing is the arithmetic or logical computation applied on vectors whereas in scalar processing only one data item or a pair of data items is processed. Parallel architectures have also been developed based on associative memory organizations. Another idea of improving the processor's speed by having multiple instructions per cycle is known as Superscalar processing. Multithreading for increasing processor utilization has also been used in parallel computer architecture. All the architectures based on these parallel-processing types have been discussed in detail in this unit.

4.1 OBJECTIVES

After going through this unit, you will be able to:

- explain the meaning of Pipeline processing and describe pipeline processing architectures;
- identify the differences between scalar, superscalar and vector processing and their architectures;

- describe architectures based on associative memory organisations, and
- explain the concept of multithreading and its use in parallel computer architecture.

4.2 PIPELINE PROCESSING

Pipelining is a method to realize, overlapped parallelism in the proposed solution of a problem, on a digital computer in an economical way. To understand the concept of pipelining, we need to understand first the concept of assembly lines in an automated production plant where items are assembled from separate parts (stages) and output of one stage becomes the input to another stage. Taking the analogy of assembly lines, pipelining is the method to introduce temporal parallelism in computer operations. Assembly line is the pipeline and the separate parts of the assembly line are different stages through which operands of an operation are passed.

To introduce pipelining in a processor P, the following steps must be followed:

- Sub-divide the input process into a sequence of subtasks. These subtasks will make stages of pipeline, which are also known as segments.
- Each stage S_i of the pipeline according to the subtask will perform some operation on a distinct set of operands.
- When stage S_i has completed its operation, results are passed to the next stage S_{i+1} for the next operation.
- The stage S_i receives a new set of input from previous stage S_{i-1} .

In this way, parallelism in a pipelined processor can be achieved such that m independent operations can be performed simultaneously in m segments as shown in *Figure 1*.

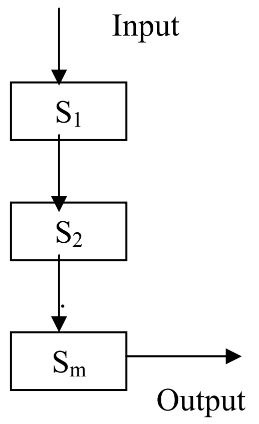


Figure 1: m-Segment Pipeline Processor

The stages or segments are implemented as pure combinational circuits performing arithmetic or logic operations over the data streams flowing through the pipe. Latches are used to separate the stages, which are fast registers to hold intermediate results between the stages as shown in *Figure 2*. Each stage S_i consists of a latch L_i and a processing circuit C_i . The final output is stored in output register R. The flow of data from one stage to another is controlled by a common clock. Thus, in each clock period, one stage transfers its results to another stage.

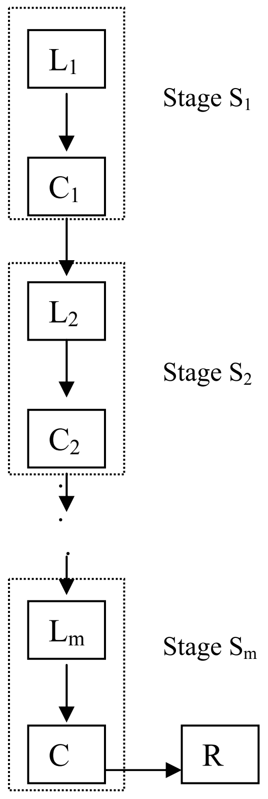


Figure 2: Pipelined Processor

Pipelined Processor: Having discussed pipelining, now we can define a pipeline processor. A pipeline processor can be defined as a processor that consists of a sequence of processing circuits called segments and a stream of operands (data) is passed through the pipeline. In each segment partial processing of the data stream is performed and the final output is received when the stream has passed through the whole pipeline. An operation that can be decomposed into a sequence of well-defined sub tasks is realized through the pipelining concept.

4.2.1 Classification of Pipeline Processors

In this section, we describe various types of pipelining that can be applied in computer operations. These types depend on the following factors:

- Level of Processing
- Pipeline configuration
- Type of Instruction and data

Classification according to level of processing

According to this classification, computer operations are classified as instruction execution and arithmetic operations. Next, we discuss these classes of this classification:

- **Instruction Pipeline:** We know that an instruction cycle may consist of many operations like, fetch opcode, decode opcode, compute operand addresses, fetch operands, and execute instructions. These operations of the instruction execution cycle can be realized through the pipelining concept. Each of these operations forms one stage of a pipeline. The overlapping of execution of the operations through the pipeline provides a speedup over the normal execution. Thus, the pipelines used for instruction cycle operations are known as *instruction pipelines.*

- **Arithmetic Pipeline:** The complex arithmetic operations like multiplication, and floating point operations consume much of the time of the ALU. These operations can also be pipelined by segmenting the operations of the ALU and as a consequence, high speed performance may be achieved. Thus, the pipelines used for arithmetic operations are known as *arithmetic pipelines*.

Classification according to pipeline configuration:

According to the configuration of a pipeline, the following types are identified under this classification:

- **Unifunction Pipelines**: When a fixed and dedicated function is performed through a pipeline, it is called a Unifunction pipeline.

- **Multifunction Pipelines**: When different functions at different times are performed through the pipeline, this is known as Multifunction pipeline. Multifunction pipelines are reconfigurable at different times according to the operation being performed.

Classification according to type of instruction and data:

According to the types of instruction and data, following types are identified under this classification:

- **Scalar Pipelines**: This type of pipeline processes scalar operands of repeated scalar instructions.

- **Vector Pipelines:** This type of pipeline processes vector instructions over vector operands.

4.2.1.1 Instruction Pipelines

As discussed earlier, the stream of instructions in the instruction execution cycle, can be realized through a pipeline where overlapped execution of different operations are performed. The process of executing the instruction involves the following major steps:

- Fetch the instruction from the main memory
- Decode the instruction
- Fetch the operand
- Execute the decoded instruction

These four steps become the candidates for stages for the pipeline, which we call as instruction pipeline (It is shown in *Figure 3*).

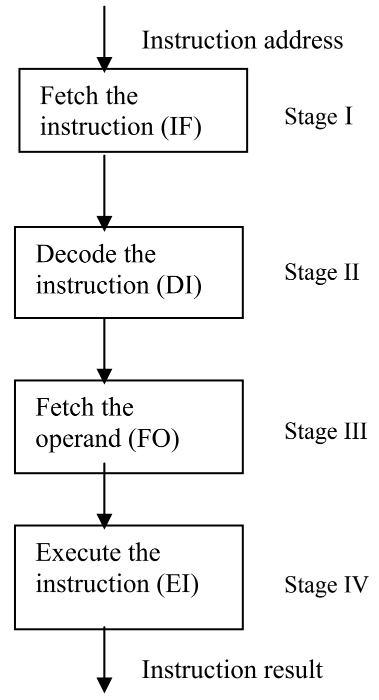


Figure 3: Instruction Pipeline

Since, in the pipelined execution, there is overlapped execution of operations, the four stages of the instruction pipeline will work in the overlapped manner. First, the instruction address is fetched from the memory to the first stage of the pipeline. The first stage fetches the instruction and gives its output to the second stage. While the second stage of the pipeline is decoding the instruction, the first stage gets another input and fetches the next instruction. When the first instruction has been decoded in the second stage, then its output is fed to the third stage. When the third stage is fetching the operand for the first instruction, then the second stage gets the second instruction and the first stage gets input for another instruction and so on. In this way, the pipeline is executing the instruction in an overlapped manner increasing the throughput and speed of execution.

The scenario of these overlapped operations in the instruction pipeline can be illustrated through the space-time diagram. In *Figure 4*, first we show the space-time diagram for non-overlapped execution in a sequential environment and then for the overlapped pipelined environment. It is clear from the two diagrams that in non-overlapped execution, results are achieved only after 4 cycles while in overlapped pipelined execution, after 4 cycles, we are getting output after each cycle. Soon in the instruction pipeline, the instruction cycle has been reduced to ¼ of the sequential execution.

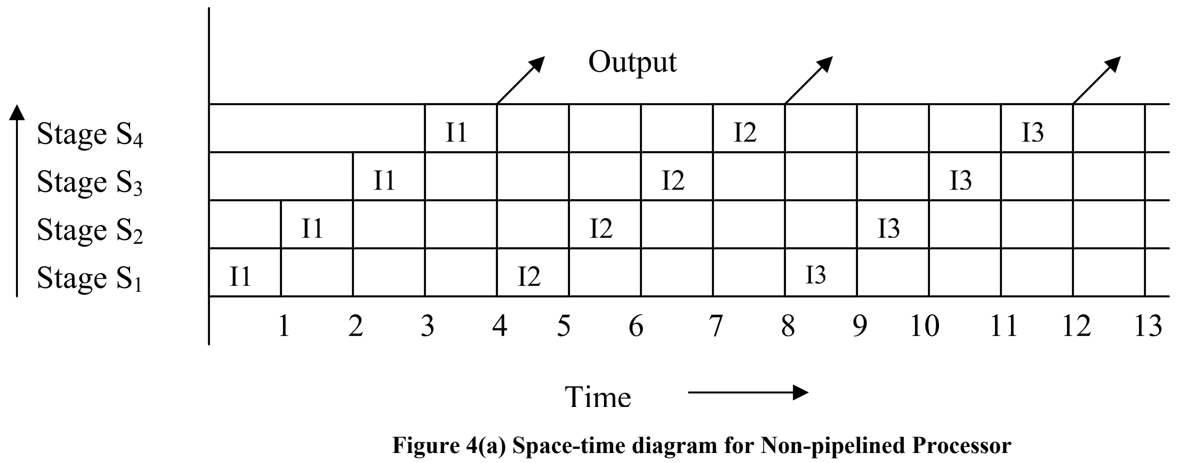


Figure 4(a) Space-time diagram for Non-pipelined Processor

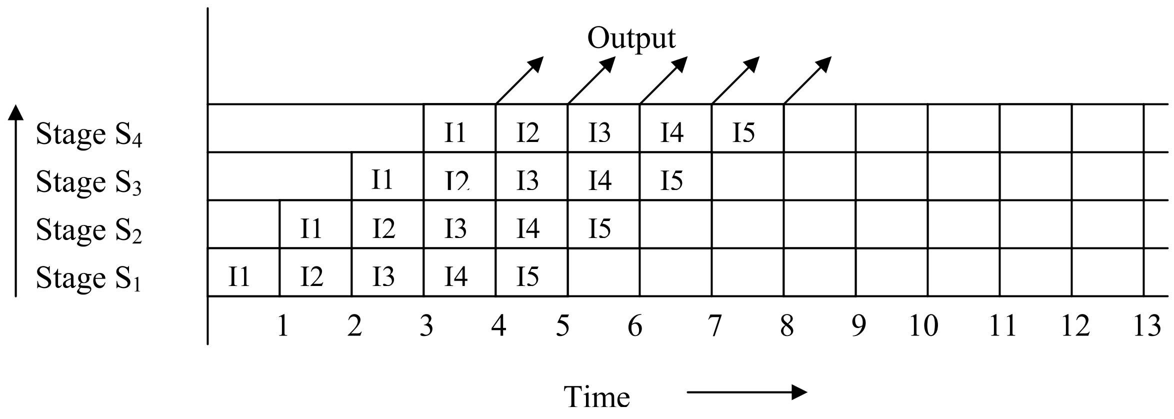


Figure 4(b) Space-time diagram for Overlapped Instruction pipelined Processor

Instruction buffers: For taking the full advantage of pipelining, pipelines should be filled continuously. Therefore, instruction fetch rate should be matched with the pipeline consumption rate. To do this, instruction buffers are used. Instruction buffers in CPU have high speed memory for storing the instructions. The instructions are pre-fetched in the buffer from the main memory. Another alternative for the instruction buffer is the cache memory between the CPU and the main memory. The advantage of cache memory is that it can be used for both instruction and data. But cache requires more complex control logic than the instruction buffer. Some pipelined computers have adopted both.

4.2.1.2 Arithmetic Pipelines

The technique of pipelining can be applied to various complex and slow arithmetic operations to speed up the processing time. The pipelines used for arithmetic computations are called *Arithmetic pipelines*. In this section, we discuss arithmetic pipelines based on arithmetic operations. Arithmetic pipelines are constructed for simple fixed-point and complex floating-point arithmetic operations. These arithmetic operations are well suited to pipelining as these operations can be efficiently partitioned into subtasks for the pipeline stages. For implementing the arithmetic pipelines we generally use following two types of adder:

i) **Carry propagation adder (CPA)**: It adds two numbers such that carries generated in successive digits are propagated.

ii) **Carry save adder (CSA)**: It adds two numbers such that carries generated are not propagated rather these are saved in a carry vector.

Fixed Arithmetic pipelines: We take the example of multiplication of fixed numbers. Two fixed-point numbers are added by the ALU using add and shift operations. This sequential execution makes the multiplication a slow process. If we look at the multiplication process carefully, then we observe that this is the process of adding the multiple copies of shifted multiplicands as show below:

$$
\begin{array}{cccccccccccl}
 & & & & & X_5 & X_4 & X_3 & X_2 & X_1 & X_0 & = X \\
 & & & & & Y_5 & Y_4 & Y_3 & Y_2 & Y_1 & Y_0 & = Y \\
\hline
 & & & & & X_5Y_0 & X_4Y_0 & X_3Y_0 & X_2Y_0 & X_1Y_0 & X_0Y_0 & = P_1 \\
 & & & & X_5Y_1 & X_4Y_1 & X_3Y_1 & X_2Y_1 & X_1Y_1 & X_0Y_1 & & = P_2 \\
 & & & X_5Y_2 & X_4Y_2 & X_3Y_2 & X_2Y_2 & X_1Y_2 & X_0Y_2 & & & = P_3 \\
 & & X_5Y_3 & X_4Y_3 & X_3Y_3 & X_2Y_3 & X_1Y_3 & X_0Y_3 & & & & = P_4 \\
 & X_5Y_4 & X_4Y_4 & X_3Y_4 & X_2Y_4 & X_1Y_4 & X_0Y_4 & & & & & = P_5 \\
X_5Y_5 & X_4Y_5 & X_3Y_5 & X_2Y_5 & X_1Y_5 & X_0Y_5 & & & & & & = P_6 \\
\hline
\\
\hline
\end{array}
$$

Now, we can identify the following stages for the pipeline:

- The first stage generates the partial product of the numbers, which form the six rows of shifted multiplicands.
- In the second stage, the six numbers are given to the two CSAs merging into four numbers.
- In the third stage, there is a single CSA merging the numbers into 3 numbers.
- In the fourth stage, there is a single number merging three numbers into 2 numbers.
- In the fifth stage, the last two numbers are added through a CPA to get the final product.

These stages have been implemented using CSA tree as shown in *Figure 5*.

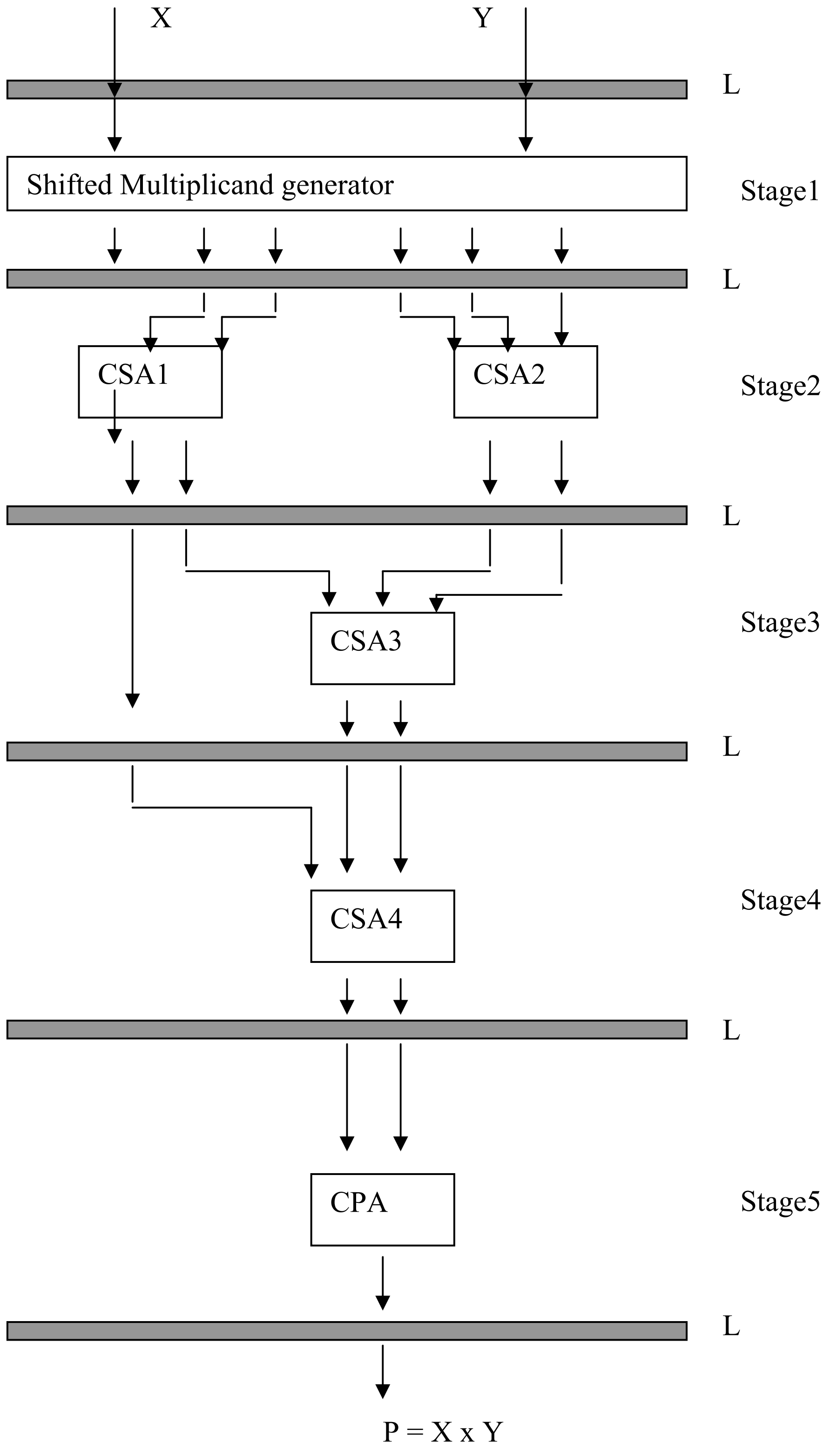


Figure 5: Arithmetic pipeline for Multiplication of two 6-digit fixed numbers

Floating point Arithmetic pipelines: Floating point computations are the best candidates for pipelining. Take the example of addition of two floating point numbers. Following stages are identified for the addition of two floating point numbers:

- First stage will compare the exponents of the two numbers.
- Second stage will look for alignment of mantissas.
- In the third stage, mantissas are added.
- In the last stage, the result is normalized.

These stages are shown in *Figure 6*.

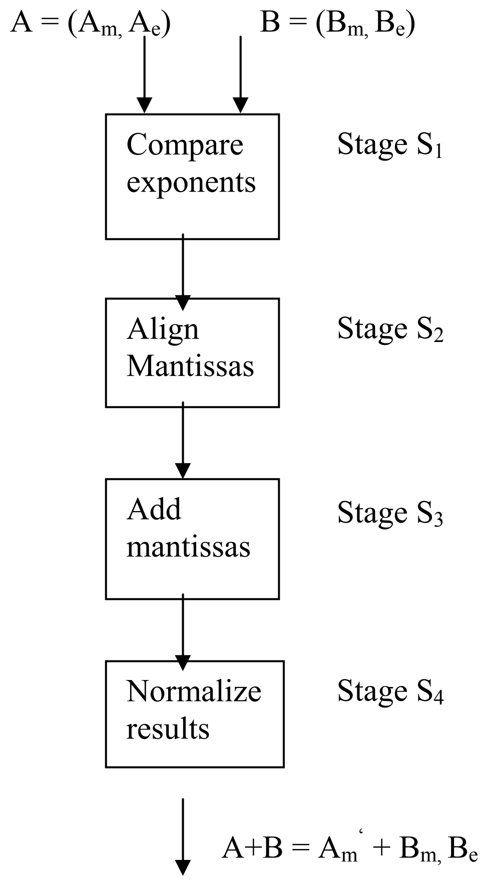


Figure 6: Arithmetic Pipeline for Floating point addition of two numbers

4.2.2 Performance and Issues in Pipelining

Speedup : First we take the speedup factor that is we see how much speed up performance we get through pipelining.

First we take the *ideal case* for measuring the speedup.

Let n be the total number of tasks executed through m stages of pipelines.

Then m stages can process n tasks in clock cycles = m + (n-1)
Time taken to execute without pipelining = m.n

Speedup due to pipelining = m.n/[m +(n-1)].

As n>=∞ , There is speedup of n times over the non-pipelined execution.

Efficiency: The efficiency of a pipeline can be measured as the ratio of busy time span to the total time span including the idle time. Let c be the clock period of the pipeline, the efficiency E can be denoted as:

$$E = (n.\ m.\ c) / m.[m.c + (n-1).c] = n / (m + (n-1)$$

As $n \to \infty$, E becomes 1.

Throughput: Throughput of a pipeline can be defined as the number of results that have been achieved per unit time. It can be denoted as:

$$T = n / [m + (n-1)].\ c = E / c$$

Throughput denotes the computing power of the pipeline.

Maximum speedup, efficiency and throughput are the ideal cases but these are not achieved in the practical cases, as the speedup is limited due to the following factors:

- **Data dependency between successive tasks:** There may be dependencies between the instructions of two tasks used in the pipeline. For example, one instruction cannot be started until the previous instruction returns the results, as both are interdependent. Another instance of data dependency will be when that both instructions try to modify the same data object. These are called *data hazards*.
- **Resource Constraints:** When resources are not available at the time of execution then delays are caused in pipelining. For example, if one common memory is used for both data and instructions and there is need to read/write and fetch the instruction at the same time then only one can be carried out and the other has to wait. Another example is of limited resource like execution unit, which may be busy at the required time.
- **Branch Instructions and Interrupts in the program:** A program is not a straight flow of sequential instructions. There may be branch instructions that alter the normal flow of program, which delays the pipelining execution and affects the performance. Similarly, there are interrupts that postpones the execution of next instruction until the interrupt has been serviced. Branches and the interrupts have damaging effects on the pipelining.

Check Your Progress 1

1) What is the purpose of using latches in a pipelined processor?
…………………………………………………………………………………………
…………………………………………………………………………………………
…………………………………………………………………………………………
…………………………………………………………………………………………
2) Differentiate between instruction pipeline and arithmetic pipeline?
…………………………………………………………………………………………
…………………………………………………………………………………………
…………………………………………………………………………………………
…………………………………………………………………………………………
3) Identify the factors due to which speed of the pipelining is limited?
…………………………………………………………………………………………
…………………………………………………………………………………………
…………………………………………………………………………………………
…………………………………………………………………………………………

4.3 VECTOR PROCESSING

A *vector* is an ordered set of the same type of scalar data items. The scalar item can be a floating pint number, an integer or a logical value. *Vector processing* is the arithmetic or logical computation applied on vectors whereas in scalar processing only one or pair of data is processed. Therefore, vector processing is faster compared to scalar processing. When the scalar code is converted to vector form then it is called v*ectorization*. A *vector processor* is a special coprocessor, which is designed to handle the vector computations.

Vector instructions can be classified as below:

- **Vector-Vector Instructions**: In this type, vector operands are fetched from the vector register and stored in another vector register. These instructions are denoted with the following function mappings:

  $$F1 : V \rightarrow V$$
  $$F2 : V \times V \rightarrow V$$

  For example, vector square root is of F1 type and addition of two vectors is of F2.
- **Vector-Scalar Instructions**: In this type, when the combination of scalar and vector are fetched and stored in vector register. These instructions are denoted with the following function mappings:

  $F3 : S \times V \rightarrow V$ where S is the scalar item

  For example, vector-scalar addition or divisions are of F3 type.
- **Vector reduction Instructions**: When operations on vector are being reduced to scalar items as the result, then these are vector reduction instructions. These instructions are denoted with the following function mappings:

  $$F4 : V \rightarrow S$$
  $$F5 : V \times V \rightarrow S$$

  For example, finding the maximum, minimum and summation of all the elements of vector are of the type F4. The dot product of two vectors is generated by F5.

- **Vector-Memory Instructions**: When vector operations with memory M are performed then these are vector-memory instructions. These instructions are denoted with the following function mappings:

  $$F6 : M \rightarrow V$$
  $$F7 : V \rightarrow V$$

  For example, vector load is of type F6 and vector store operation is of F7.

Vector Processing with Pipelining: Since in vector processing, vector instructions perform the same computation on different data operands repeatedly, vector processing is most suitable for pipelining. Vector processors with pipelines are designed to handle vectors of varying length n where n is the length of vector. A vector processor performs better if length of vector is larger. But large values of n causes the problem in storage of vectors and there is difficulty in moving the vectors to and from the pipelines.

Pipeline Vector processors adopt the following two architectural configurations for this problem as discussed below:

- ***Memory-to-Memory Architecture***: The pipelines can access vector operands, intermediate and final results directly in the main memory. This requires the higher memory bandwidth. Moreover, the information of the base address, the offset and vector length should be specified for transferring the data streams between the main memory and pipelines. STAR-100 and TI-ASC computers have adopted this architecture for vector instructions.

- ***Register*****-to-Register Architecture**: In this organization, operands and results are accessed indirectly from the main memory through the scalar or vector registers. The vectors which are required currently can be stored in the CPU registers. Cray-1 computer adopts this architecture for the vector instructions and its CPY contains 8 vector registers, each register capable of storing a 64 element vector where one element is of 8 bytes.

Efficiency of Vector Processing over Scalar Processing:

As we know, a sequential computer processes scalar operands one at a time. Therefore if we have to process a vector of length n through the sequential computer then the vector must be broken into n scalar steps and executed one by one.

For example, consider the following vector addition:

$$\mathbf{A + B \longrightarrow C}$$

The vectors are of length 500. This operation through the sequential computer can be

specified by 500 add instructions as given below:

C[1] = A[1] + B[1]
C[2] = A[2] + B[2]
- -
C[500] = A[500] + B[500]

If we perform the same operation through a pipelined-vector computer then it does not break the vectors in 500 add statements. Because a vector processor has the set of vector instructions that allow the operations to be specified in one vector instruction as:

$$\mathbf{A\ (1{:}500) + B\ (1{:}500) \longrightarrow C\ (1{:}500)}$$

Each vector operation may be broken internally in scalar operations but they are executed in parallel which results in mush faster execution as compared to sequential computer.

Thus, the advantage of adopting vector processing over scalar processing is that it eliminates the overhead caused by the loop control required in a sequential computer.

4.4 ARRAY PROCESSING

We have seen that for performing vector operations, the pipelining concept has been used. There is another method for vector operations. If we have an array of n processing elements (PEs) i.e., multiple ALUs for storing multiple operands of the vector, then an n instruction, for example, vector addition, is broadcast to all PEs such that they add all operands of the vector at the same time. That means all PEs will perform computation in parallel. All PEs are synchronised under one control unit. This organisation of synchronous array of PEs for vector operations is called *Array Processor.* The organisation is same as in SIMD which we studied in unit 2. An array processor can handle one instruction and multiple data streams as we have seen in case of SIMD organisation. Therefore, array processors are also called *SIMD array computers*.

The organisation of an array processor is shown in *Figure 7*. The following components are organised in an array processor:

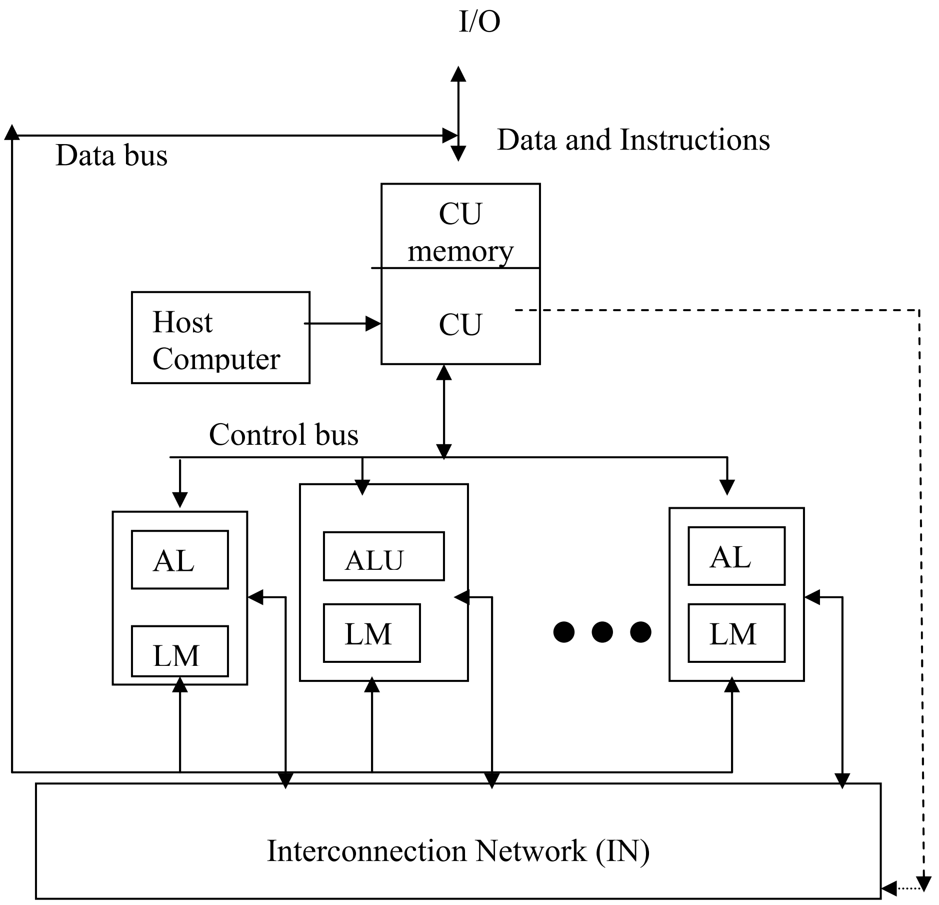


Figure 7: Organisation of SIMD Array Processor

Control Unit (CU) : All PEs are under the control of one control unit. CU controls the inter communication between the PEs. There is a local memory of CU also called CY memory. The user programs are loaded into the CU memory. The vector instructions in the program are decoded by CU and broadcast to the array of PEs. Instruction fetch and decoding is done by the CU only.

Processing elements (PEs) : Each processing element consists of ALU, its registers and a local memory for storage of distributed data. These PEs have been interconnected via an interconnection network. All PEs receive the instructions from the control unit and the different component operands are fetched from their local memory. Thus, all PEs perform the same function synchronously in a lock-step fashion under the control of the CU.

It may be possible that all PEs need not participate in the execution of a vector instruction. Therefore, it is required to adopt a masking scheme to control the status of each PE. A *masking vector* is used to control the status of all PEs such that only enabled PEs are allowed to participate in the execution and others are disabled.

Interconnection Network (IN): IN performs data exchange among the PEs, data routing and manipulation functions. This IN is under the control of CU.

Host Computer: An array processor may be attached to a host computer through the control unit. The purpose of the host computer is to broadcast a sequence of vector instructions through CU to the PEs. Thus, the host computer is a general-purpose machine that acts as a manager of the entire system.

Array processors are special purpose computers which have been adopted for the following:

- various scientific applications,
- matrix algebra,
- matrix eigen value calculations,
- real-time scene analysis

SIMD array processor on the large scale has been developed by NASA for earth resources satellite image processing. This computer has been named *Massively parallel processor* (MPP) because it contains 16,384 processors that work in parallel. MPP provides real-time time varying scene analysis.

However, array processors are not commercially popular and are not commonly used. The reasons are that array processors are difficult to program compared to pipelining and there is problem in vectorization.

4.4.1 Associative Array Processing

Consider that a table or a list of record is stored in the memory and you want to find some information in that list. For example, the list consists of three fields as shown below:

Name	ID Number	Age
Sumit	234	23
Ramesh	136	26
Ravi	97	35

Suppose now that we want to find the ID number and age of Ravi. If we use conventional RAM then it is necessary to give the exact physical address of entry related to Ravi in the instruction access the entry such as:

READ ROW 3

Another alternative idea is that we search the whole list using the Name field as an address in the instruction such as:

READ NAME = RAVI

Again with serial access memory this option can be implemented easily but it is a very slow process. An ***associative memory*** helps at this point and simultaneously examines all the entries in the list and returns the desired list very quickly.
SIMD array computers have been developed with *associative memory*. An associative memory is content addressable memory, by which it is meant that multiple memory words

are accessible in parallel. The parallel accessing feature also support parallel search and parallel compare. This capability can be used in many applications such as:

- Storage and retrieval of databases which are changing rapidly
- Radar signal tracking
- Image processing
- Artificial Intelligence

The inherent parallelism feature of this memory has great advantages and impact in parallel computer architecture. The associative memory is costly compared to RAM. The array processor built with associative memory is called *Associative array processor.* In this section, we describe some categories of associative array processor. Types of associative processors are based on the organisation of associative memory. Therefore, first we discuss about the associative memory organisation.

Associative Memory Organisations

The associative memory is organised in *w* words with *b* bits per word. In w x b array, each bit is called a *cell*. Each cell is made up of a flip-flop that contains some comparison logic gates for pattern match and read-write operations. Therefore, it is possible to read or write in parallel due to this logic structure. A group of bit cells of all the words at the same position in a vertical column is called *bit slice* as shown in *Figure 8*.

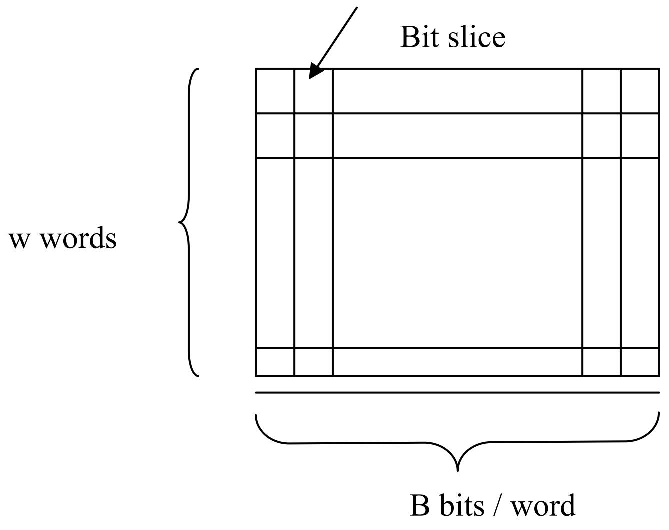


Figure 8: Associative memory

In the organisation of an associative memory, following registers are used:

- *Comparand Register* (C): This register is used to hold the operands, which are being searched for, or being compared with.

- *Masking Register* (M): It may be possible that all bit slices are not involved in parallel operations. Masking register is used to enable or disable the bit slices.

- *Indicator* (I) and *Temporary* (T) *Registers*: Indicator register is used to hold the current match patterns and temporary registers are used to hold the previous match patterns.

There are following two methods for organising the associative memory based on bit slices:

- **Bit parallel organisation:** In this organisation all bit slices which are not masked off, participate in the comparison process, i.e., all words are used in parallel.
- **Bit Serial Organisation:** In this organisation, only one bit slice participate in the operation across all the words. The bit slice is selected through an extra logic and control unit. This organisation is slower in speed but requires lesser hardware as compared to bit parallel which is faster.

Types of Associative Processor

Based on the associative memory organisations, we can classify the associative processors into the following categories:

1) **Fully Parallel Associative Processor**: This processor adopts the bit parallel memory organisation. There are two type of this associative processor:

   - **Word Organized associative processor**: In this processor one comparison logic is used with each bit cell of every word and the logical decision is achieved at the output of every word.

   - **Distributed associative processor**: In this processor comparison logic is provided with each character cell of a fixed number of bits or with a group of character cells. This is less complex and therefore less expensive compared to word organized associative processor.

2) **Bit Serial Associative Processor:** When the associative processor adopts bit serial memory organization then it is called bit serial associative processor. Since only one bit slice is involved in the parallel operations, logic is very much reduced and therefore this processor is much less expensive than the fully parallel associative processor.

PEPE is an example of distributed associative processor which was designed as a special purpose computer for performing real time radar tracking in a missile environment. STARAN is an example of a bit serial associative processor which was designed for digital image processing. There is a high cost performance ratio of associative processors. Due to this reason these have not been commercialised and are limited to military applications.

Check Your Progress 2

1) What is the difference between scalar and vector processing?
   …………………………………………………………………………………………
   …………………………………………………………………………………………
   …………………………………………………………………………………………
   …………………………………………………………………………………………
2) Identify the types of the following vector processing instructions?
   a) C(I) = A(I) AND B(I)
   b) C(I) = MAX(A(I), B(I))
   c) B(I) = A(I) / S, where S is a scalar item
   d) B(I) <- SIN (A(I))

…………………………………………………………………………………………
…………………………………………………………………………………………
…………………………………………………………………………………………
…………………………………………………………………………………………

3) What is the purpose of using the comparand and masking register in the associative memory organisation?

…………………………………………………………………………………………
…………………………………………………………………………………………
…………………………………………………………………………………………
…………………………………………………………………………………………

4.5 SUPERSCALAR PROCESSORS

In scalar processors, only one instruction is executed per cycle. That means only one instruction is issued per cycle and only one instruction is completed. But the speed of the processor can be improved in scalar pipeline processor if multiple instructions instead of one are issued per cycle. This idea of improving the processor's speed by having multiple instructions per cycle is known as *Superscalar processing*. In superscalar processing multiple instructions are issued per cycle and multiple results are generated per cycle. Thus, the basic idea of superscalar processor is to have more instruction level parallelism.

Instruction Issue degree: The main concept in superscalar processing is how many istructions we can issue per cycle. If we can issue k number of instructions per cycle in a superscalar processor, then that processor is called a k-degree superscalar processor. If we want to exploit the full parallelism from a superscalar processor then k instructions must be executable in parallel.

For example, we consider a 2-degree superscalar processor with 4 pipeline stages for instruction cycle, i.e. instruction fetch (IF), decode instruction (DI), fetch the operands (FO), execute the instruction (EI) as shown in *Figure 3*. In this superscalar processor, 2 instructions are issued per cycle as shown in *Figure 9*. Here, 6 instructions in 4 stage pipeline have been executed in 6 clock cycles. Under ideal conditions, after steady state, two instructions are being executed per cycle.

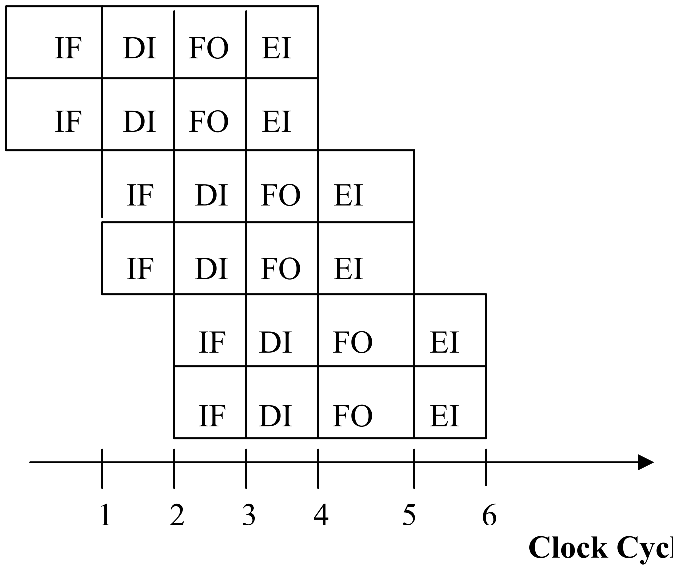


Figure 9: Superscalar Processing of instruction cycle in 4-stage instruction pipeline

For implementing superscalar processing, some special hardware must be provided which is discussed below:

- The requirement of data path is increased with the degree of superscalar processing. Suppose, one instruction size is 32 bit and we are using 2-degree superscalar processor, then 64 data path from the instruction memory is required and 2 instruction registers are also needed.
- Multiple execution units are also required for executing multiple instructions and to avoid resource conflicts.

Data dependency will be increased in superscalar processing if sufficient hardware is not provided. The extra hardware provided is called *hardware machine parallelism*. Hardware parallelism ensures that resource is available in hardware to exploit parallelism. Another alternative is to exploit the *instruction level parallelism* inherent in the code. This is achieved by transforming the source code by an optimizing compiler such that it reduces the dependency and resource conflicts in the resulting code.

Many popular commercial processors have been implemented with superscalar architecture like IBM RS/6000, DEC 21064, MIPS R4000, Power PC, Pentium, etc.

4.6 VLIW ARCHITECTURE

Superscalar architecture was designed to improve the speed of the scalar processor. But it has been realized that it is not easy to implement as we discussed earlier. Following are some problems faced in the superscalar architecture:

- It is required that extra hardware must be provided for hardware parallelism such as instruction registers, decoder and arithmetic units, etc.
- Scheduling of instructions dynamically to reduce the pipeline delays and to keep all processing units busy, is very difficult.

Another alternative to improve the speed of the processor is to exploit a sequence of instructions having no dependency and may require different resources, thus avoiding resource conflicts. The idea is to combine these independent instructions in a compact long word incorporating many operations to be executed simultaneously. That is why; this architecture is called ***very long instruction word (VLIW) architecture***. In fact, long instruction words carry the opcodes of different instructions, which are dispatched to different functional units of the processor. In this way, all the operations to be executed simultaneously by the functional units are synchronized in a VLIW instruction. The size of the VLIW instruction word can be in hundreds of bits. VLIW instructions must be formed by compacting small instruction words of conventional program. The job of compaction in VLIW is done by a compiler. The processor must have the sufficient resources to execute all the operations in VLIW word simultaneously.

For example, one VLIW instruction word is compacted to have load/store operation, floating point addition, floating point multiply, one branch, and one integer arithmetic as shown in *Figure 10*.

Load/Store	FP Add	FP Multiply	Branch	Integer arithmetic

Figure 10: VLIW instruction word

A VLIW processor to support the above instruction word must have the functional components as shown in *Figure 11*. All the functions units have been incorporated according to the VLIW instruction word. All the units in the processor share one common large register file.

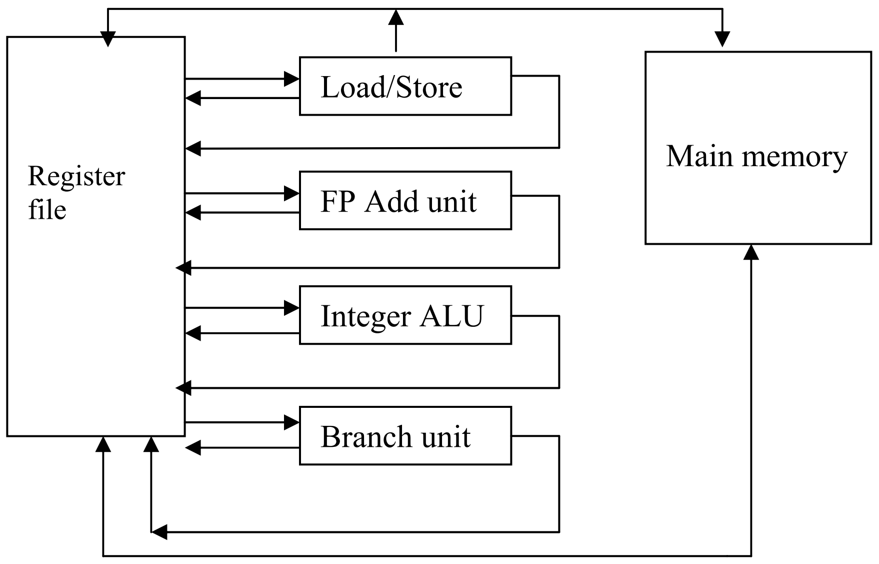


Figure 11: VLIW Processor

Parallelism in instructions and data movement should be completely specified at compile time. But scheduling of branch instructions at compile time is very difficult. To handle branch instructions, *trace scheduling* is adopted. Trace scheduling is based on the prediction of branch decisions with some reliability at compile time. The prediction is based on some heuristics, hints given by the programmer or using profiles of some previous program executions.

4.7 MULTI-THREADED PROCESSORS

In unit 2, we have seen the use of distributed shared memory in parallel computer architecture. But the use of distributed shared memory has the problem of accessing the remote memory, which results in latency problems. This problem increases in case of large-scale multiprocessors like massively parallel processors (MPP).

For example, one processor in a multiprocessor system needs two memory loads of two variables from two remote processors as shown in *Figure 12*. The issuing processor will use these variables simultaneously in one operation. In case of large-scale MPP systems, the following two problems arise:

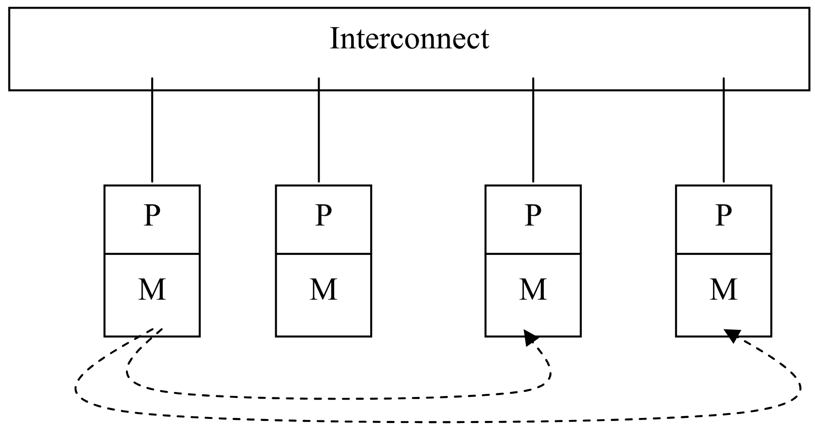


Figure 12: Latency problems in MPP

Remote-load Latency Problem: When one processor needs some remote loading of data from other nodes, then the processor has to wait for these two remote load operations. The longer the time taken in remote loading, the greater will be the latency and idle period of the issuing processor.

Synchronization Latency Problem: If two concurrent processes are performing remote loading, then it is not known by what time two processes will load, as the issuing processor needs two remote memory loads by two processes together for some operation. That means two concurrent processes return the results asynchronously and this causes the synchronization latency for the processor.

Concept of Multithreading: These problems increase in the design of large-scale multiprocessors such as MPP as discussed above. Therefore, a solution for optimizing these latency should be acquired at. The concept of *Multithreading* offers the solution to these problems. When the processor activities are multiplexed among many threads of execution, then problems are not occurring. In single threaded systems, only one thread of execution per process is present. But if we multiplex the activities of process among several threads, then the multithreading concept removes the latency problems.

In the above example, if multithreading is implemented, then one thread can be for issuing a remote load request from one variable and another thread can be for remote load for second variable and third thread can be for another operation for the processor and so on.

Multithreaded Architecture: It is clear now that if we provide many contexts to multiple threads, then processors with multiple contexts are called multithreaded systems. These systems are implemented in a manner similar to multitasking systems. A multithreaded processor will suspend the current context and switch to another. In this way, the processor will be busy most of the time and latency problems will also be optimized. Multithreaded architecture depends on the context switching time between the threads. The switching time should be very less as compared to latency time.

The processor utilization or its efficiency can be measured as:

$$U = P / (P + I + S)$$

where

P = useful processing time for which processor is busy
I = Idle time when processor is waiting
S = Context switch time used for changing the active thread on the procesor

The objective of any parallel system is to keep U as high as possible. U will be high if I and S are very low or negligible. The idea of multithreading systems is to reduce I such that S is not increasing. If context-switching time is more when compared to idle time, then the purpose of multithreaded systems is lost.

Design issues: To achieve the maximum processor utilization in a multithreaded architecture, the following design issues must be addressed:

- *Context Switching time*: $S < I$, that means very fast context switching mechanism is needed.
- *Number of Threads*: A large number of threads should be available such that processor switches to an active thread from the idle state.

Check Your Progress 3

1) What is the difference between scalar processing and superscalar processing?
……………………………………………………………………………………
……………………………………………………………………………………
……………………………………………………………………………………
……………………………………………………………………………………
2) If a superscalar processor of degree 3 is used in 4-stage pipeline instructions, then how many instructions will be executed in 7 clock cycles?
……………………………………………………………………………………
……………………………………………………………………………………
……………………………………………………………………………………
……………………………………………………………………………………
3) What is the condition for compacting the instruction in a VLIW instruction word?
……………………………………………………………………………………
……………………………………………………………………………………
……………………………………………………………………………………
……………………………………………………………………………………

4.8 SUMMARY

In this unit, we discuss some of the well-known architecture that exploits inherent parallelism in the data or the problem domain. In section 4.2, we discuss pipeline architecture, which is suitable for executing solutions of problems in the cases when either execution of each of the instructions or operations can be divided into non-overlapping stages. In section 4.3, vector processing, another architecture concurrent execution, is discussed. The vector processing architecture is useful when the same operation at a time, is to be applied to a number of operands of the same type. The vector processing may be achieved through pipelined architecture, if the operation can be divided into a number of non-overlapping stages. Alternatively, vector processing can also be achieved through array processing in which by a large number of processing elements are used. All these PEs perform an identical operation on different components of the vector operand(s). The goal of all these architectures discussed so far is the same — expediting the execution speed by exploiting inherent concurrency in the problem domain in the data. This goal of expediting execution at even higher, speed is attempted to be achieved through three other architectures discussed in next three sections. In section 4.5, we discuss the architecture known as superscalar processing architecture, under which more than one instruction per cycles may be executed. Next, in section 4.6, we discuss VLIW architecture, which is useful when program codes, have a number of chunks of instructions which have no dependency and also, hence or otherwise require different resources. Finally, in section

4.7, another approach viz. Multi-threaded processors approach, of expediting execution is discussed. Through multi-threaded approach, the problem of some type of latencies encountered in some of the architectures discussed earlier may be overcome.

4.9 SOLUTIONS / ANSWERS

Check Your Progress 1

1) Latches are used to separate the stages, which are fast registers to hold intermediate results between the stages.

2) Instruction pipelines are used to execute the stream of instructions in the instruction execution cycle whereas the pipelines used for arithmetic computations, both fixed and floating point operations, are called arithmetic pipelines.

3) A) Data dependency between successive tasks
B) Unavailability of resources
C) Branch instructions and interrupts in the program

Check Your Progress 2

1) Vector processing is the arithmetic or logical computation applied on vectors whereas in scalar processing only a pair of data is processed at a time.

2) a) Vector-vector instruction b) vector-vector instruction c) vector-scalar instruction d) vector-vector instruction.

3) The purpose of comparand register in associative memory organization is to hold the operands which are being searched for or being compared with. Masking register is used to enable or disable the bit slices.

Check Your Progress 3

1) In scalar processing, only one instruction is executed per cycle but when multiple instructions are issued per cycle and multiple results are generated per cycle then it is known as superscalar processing.
2) 12
3) The condition for compacting multiple instructions in a VLIW word is that the processor must have the sufficient resources to execute all the operations in VLIW word simultaneously.