

## **2. Parallel Architectures**

### **2.1 Objectives**

- To introduce the principles and classification of parallel architectures.
- To discuss various forms of parallel processing.
- To explore the characteristics of parallel architectures.

### **2.2 Introduction**

Parallel processing has emerged as an area with the potential of providing satisfactory ways of meeting real-time computation requirements in various applications and the quest for speed, which is a natural tendency of human beings, as discussed in Chapter 1. As such, various forms of parallel architectures have evolved and research is being carried out worldwide to discover new architectures that can perform better than existing architectures. With speed as the prime goal, the factors generally considered in developing a new architecture include the internal circuitry of processors or PEs, number of PEs, arrangement of PEs and memory modules in an architecture, the communication mechanism among the PEs and between the PEs and memory modules, number of instruction and data streams, nature of memory connections with the PEs, nature and types of interconnection among the PEs, and program overlapping. The internal circuitry of PEs plays a vital role in designing parallel architecture. Some parallel architectures are designed with small number of PEs of complex internal circuitry to enhance the overall performance of the architecture. Other architectures, on the other hand, are designed with a substantial number of PEs of simple internal circuitry to achieve the desired performance. The performance of an architecture can often be increased by adding more PEs. However, this is possible only up to a certain limit, as adding more PEs to the architecture incurs more communication overhead and it may not be cost effective beyond a limit. The arrangement of processors is also crucial to the design of a parallel architecture. Processors can be arranged in different forms using various types of interconnection strategies, such as static and dynamic. In a static form, the format of the architecture remains fixed and is not expandable by

adding more processors, while in a dynamic format more processors can be added under system control to meet a particular requirement. The communication mechanism among PEs in some architectures is straightforward, while in others the communication scheme is complicated and requires extra effort and circuitry. Similar to the arrangement of PEs, the arrangement of memory modules in an architecture is important and contributes to the development of varied forms of parallel architectures. The memory arrangement in some architectures is global, i.e., all PEs use a common memory or memory modules, which helps to establish communication among the processors, while others have memory modules associated with individual PEs and communication is established via messages passing among the PEs. Some forms of parallel architecture have evolved on the basis of the methodology an algorithm is implemented. A pipeline architecture is an example of this. There are also special forms of parallel architectures developed on the basis of merging the characteristics of various forms of existing parallel architectures. Many attempts have also been made to develop application-specific parallel architectures. For example, vector parallel architectures have been developed to execute vector intensive algorithms, DSP devices have been designed for efficient implementation of signal processing, e.g. digital filtering algorithms.

Indeed, research and development in parallel computing is a continuing effort, and it is accordingly expected that various forms of parallel architectures will emerge in the future. Whatever may be the basis of development, each type of parallel processor has its own characteristics, advantages, disadvantages and suitability in certain application areas.

## 2.3 Classifications

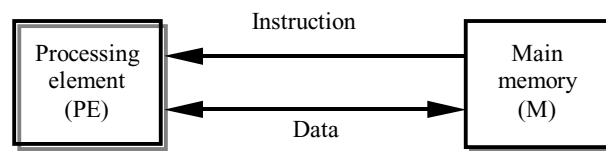
A vast number parallel architecture types have been devised over the years. Accordingly, it is not easy to develop a simple classification system for parallel architectures. Moreover, various types of parallel architecture have overlapping characteristics to different extents. However, the various forms of parallel architecture can be distinguished under the following broad categories:

- Flynn's classification.
- Classification based on memory arrangement and communication among PEs.
- Classification based on interconnections among PEs and memory modules.
- Classification based on characteristic nature of PEs.
- Specific types of parallel architectures.

### 2.3.1 Flynn's Classification

Michael J. Flynn introduced a broad classification of parallel processors based on the simultaneous instruction and data streams during program execution (Flynn, 1966). It is noted here that instruction and data streams are two main steps that occur during program execution, as depicted in Figure 2.1. It shows that during

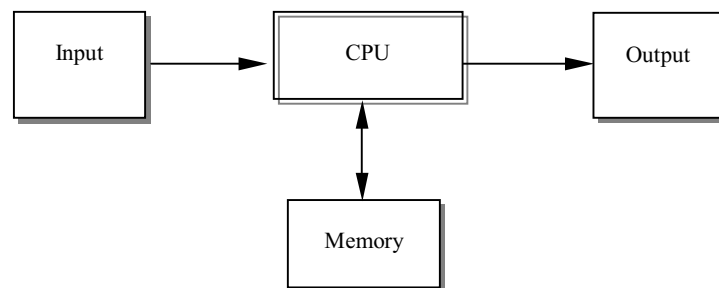
program execution the PE fetches instructions and data from the main memory, processes the data as per the instructions and sends the results to the main memory after processing has been completed. The instructions are referred to as an instruction stream, which flows from the main memory to the PE, and the data is referred to as the data stream, flowing to and from the PE. Based on these streams, Flynn categorised computers into four major classes, which are described below.



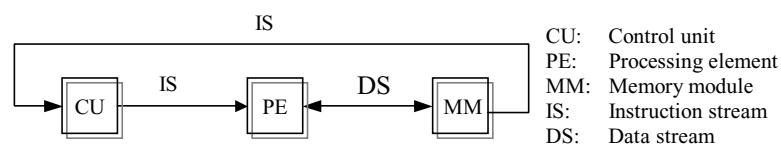
**Figure 2.1.** Instruction stream and data stream

#### *Single-instruction Single-data Stream*

The simple Von Neumann computer shown in Figure 2.2 falls under the category of single-instruction single-data (SISD) stream. An alternative representation of the architecture indicating instruction and data flow is shown in Figure 2.3. The SISD computer architecture possesses a central processing unit (CPU), memory unit and input/output (I/O) devices. The CPU consists of an arithmetic and logic unit (ALU) to perform arithmetic and logical operations, control unit (CU) to perform control operations and registers to store small amounts of data. SISD computers are sequential computers and are incapable of performing parallel operations.



**Figure 2.2.** Von Neumann (SISD) computer architecture

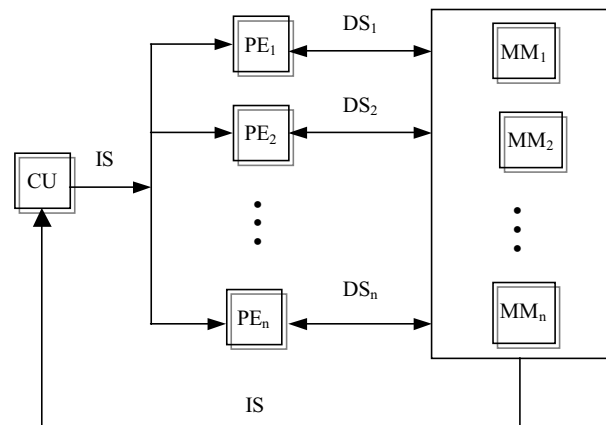


**Figure 2.3.** SISD architecture with instruction and data flow

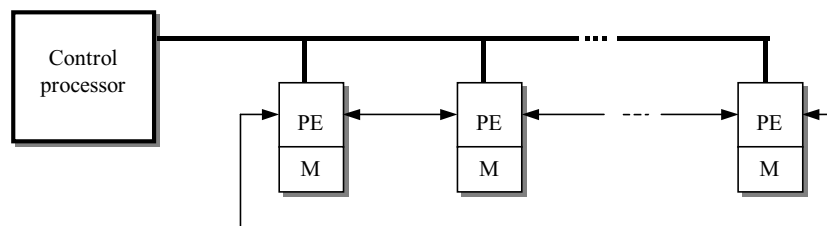
*Single-instruction Multiple-data Stream*

The general structure of a single-instruction multiple-data (SIMD) parallel architecture is shown in Figure 2.4. This architecture possesses a single instruction stream, to process the entire data structure (multiple-data stream). In other words, in this configuration, a single program control unit or control processor controls multiple execution units or execution processors. Each execution processor is equipped with a local memory to store the data it will work on. Each execution processor executes the same instruction issued by the control processor on its local data. The execution processors are capable of communicating with each other when required. SIMD architectures are good for problems where the same operation is executed on a number of different objects, for example image processing. Some other suitable applications of SIMD architectures include matrix operations and sorting. Programming is quite simple and straightforward for this architecture. SIMD architecture could be divided into two subclasses according to the interconnections between the PEs. These are: vector architecture and array architecture.

Organisation of the SIMD vector architecture is shown in Figure 2.5. The PEs are connected to each other via special data links. These links are used to perform simple data exchange operations like shifts and rotations (Hays, 1988). All the PEs are connected to the central control processor to obtain the instructions to be executed.

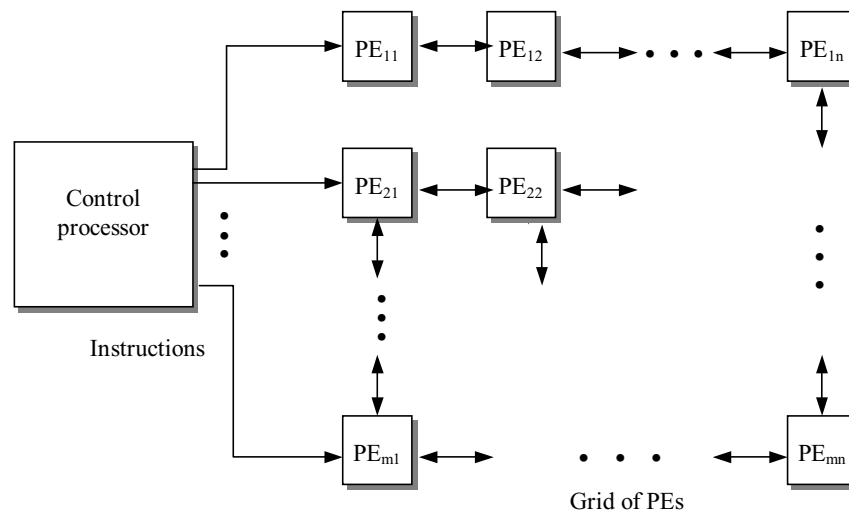


**Figure 2.4.** General structure of SIMD computer architecture



**Figure 2.5.** Vector architecture

The SIMD computer is often used as a synonym for array architecture. Unlike vector architecture, PEs in an array architecture are connected by interconnecting networks. A general form of array structure is shown in Figure 2.6, where a two-dimensional grid of PEs executes instructions provided by the control processor. Each PE is connected to its four neighbours to exchange data. End-around connections also exist on both rows and columns, which are not shown in Figure 2.6. Each PE is capable of exchanging values with each of its neighbours. Each PE possesses a few registers and some local memory to store data. Each PE is also equipped with a special register called a network register to facilitate movement of values to and from its neighbours. Each PE also contains an ALU to execute arithmetic instructions broadcast by the control processor. The central processor is capable of broadcasting an instruction to shift the values in the network registers one step up, down, left or right (Hamacher *et al.*, 2002). Array architecture is very powerful and will suit problems that can be expressed in matrix or vector format.



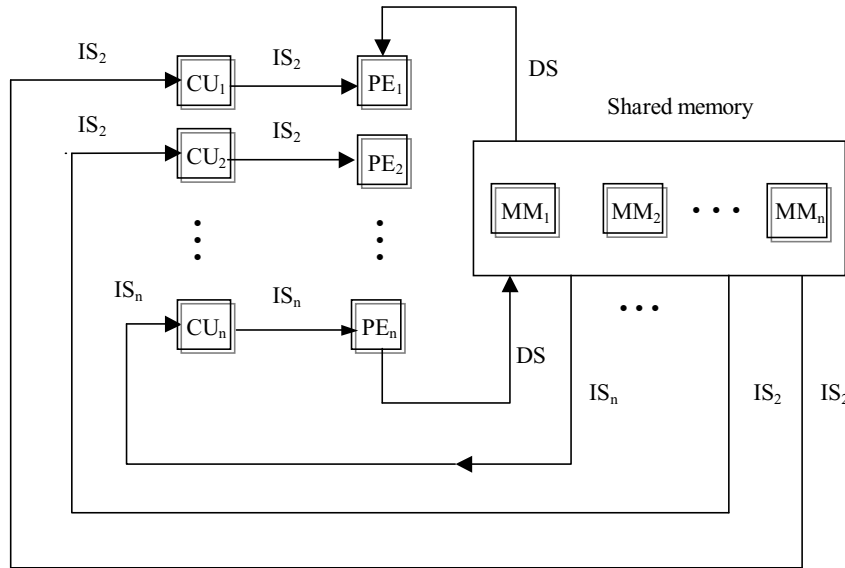
**Figure 2.6.** Array architecture

Both array and vector architectures are specialised machines. They are mainly designed to solve numerical problems comprising substantial numbers of vector and/or matrix elements. The basic difference between vector and array architectures is that high-performance is achieved in vector architecture through exploiting a pipelining mechanism whereas in array architecture a large number of PEs are incorporated to boost the performance. None of the architectures is well suited to enhance the performance of general computation. The most famous example of an array processor is ILLIAC-IV designed at the University of Illinois and built by Burroughs Corporation. It was developed using 64 processors. Other important examples of array processors include the Thinking Machine Corporation's CM-2 processor, which could have up to 65536 processors, Maspar's MP-1216 processors, which could accommodate 16384 processors, and

the Cambridge parallel processing Gamma II plus machine, which could accommodate up to 4096 processors (Hamacher *et al.*, 2002).

#### *Multiple-instruction Single-data Stream*

The general structure of a multi-instruction single-data stream (MISD) architecture is shown in Figure 2.7. This architecture possesses a multiple instruction stream and single data stream. This architecture has not evolved substantially and thus, there are not many examples of the architecture (Hays, 1988). However, a major class of parallel architectures, called pipeline computers can be viewed as MISD architecture. Pipeline computers will be discussed later in this chapter.



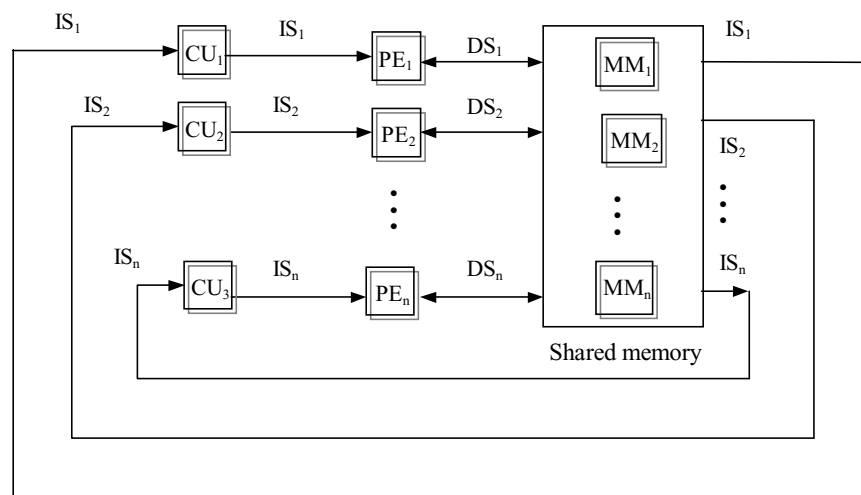
**Figure 2.7.** General structure of MISD computer architecture

For most applications, MISD computers are rather awkward to use, but can be useful in applications of a specialised nature (Akl, 1989; Kourmoulis, 1990). A typical example of one such specialised application is robot vision. For example, a robot inspecting a conveyor belt by sight may have to recognise objects that belong to different classes. An MISD machine can quickly carry out a classification task by assigning each of its processors a different class of objects and after receiving what the robot sees each processor may carry out tests to determine whether the given object belongs to its class or not.

#### *Multiple-instruction Multiple-data Stream*

Figure 2.8 shows the general structure of multiple-instruction multiple-data (MIMD) stream architecture. This architecture is the most common and widely used form of parallel architectures. It comprises several PEs, each of which is capable of executing independent instruction streams on independent data streams.

The PEs in the system typically share resources such as communication facilities, I/O devices, program libraries and databases. All the PEs are controlled by a common operating system. The multiple PEs in the system improves performance and increase reliability. Performance increases due to the fact that computational load is shared by the PEs in the system. Theoretically, if there are  $n$  PEs in the system, the performance will increase by  $n$  times in comparison to a single PE based system. System reliability is increased by the fact that failure of one PE in the system does not cause failure of the whole system.



**Figure 2.8.** General structure of MIMD computer architecture

### 2.3.2 Classification Based on Memory Arrangement and Communication among PEs

Parallel architectures can be classified into two major categories in terms of memory arrangement. These are: shared memory and message passing or distributed memory. In fact, these architectures constitute a subdivision of MIMD parallel architecture. Shared memory and distributed memory architectures are also called tightly coupled and loosely coupled architectures respectively. Each type of architecture has its advantages and disadvantages.

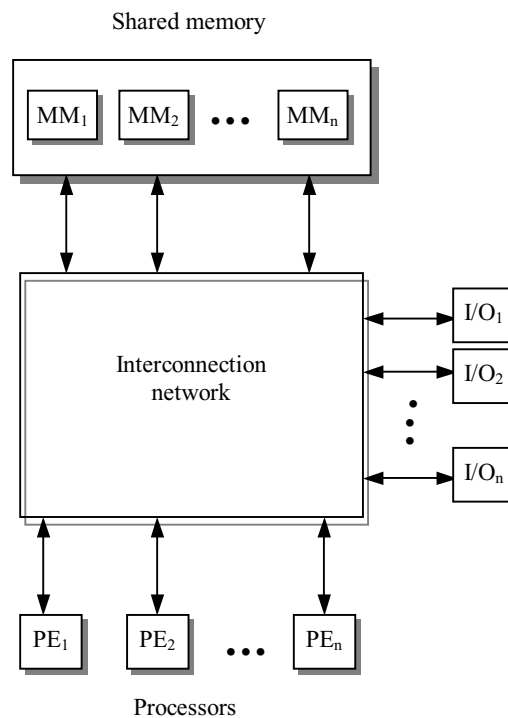
#### *Shared Memory Multiprocessor*

In a shared memory multiprocessor configuration multiple processors share a common memory unit comprising a single or several memory modules. All the processors have equal access to these memory modules and these memory modules are seen as a single address space by all the processors. The memory modules store data as well as serve to establish communication among the processors via some bus arrangement. Communication is established through memory access

instructions. That is, processors exchange messages between one another by one processor writing data into the shared memory and another reading that data from the memory.

Programming this architecture is quite straightforward and attractive. The executable programming codes are stored in the memory for each processor to execute. The data related to each program is also stored in this memory. Each program can gain access to all data sets present in the memory if necessary. The executable codes and shared data for the processor can be created and managed in different ways by designing parallel programming language or using existing sequential languages such as C/C++. There is no direct processor-to-processor communication involved in the programming process; instead communication is handled mainly via the shared memory modules. Access to these memory modules can easily be controlled through an appropriate programming mechanism such as multitasking. However, this architecture suffers from a bottleneck problem when a number of processors endeavour to access the global memory at the same time. This limits the scalability of the system. As a remedy to this problem most large, practical shared memory systems have some form of hierarchical or distributed memory structure such that processors can access physically nearby memory locations faster than distant memory locations. This is called non-uniform memory access. Figure 2.9 shows a general form of shared memory multiprocessor architecture.

Shared memory multiprocessors can be of two types, namely uniform memory access (UMA) architecture and non-uniform memory access (NUMA) architecture.



**Figure 2.9.** Shared-memory multiprocessor



As the name suggests, the memory access time to the different parts of the memory are almost the same in the case of UMA architectures. UMA architectures are also called symmetric multiprocessors. An UMA architecture comprises two or more processors with identical characteristics. The processors share the same main memory and I/O facilities and are interconnected by some form of bus-based interconnection scheme such that the memory access time is approximately the same for all processors. All processors can perform the same functions under control of an integrated operating system, which provides interaction between processors and their programs at the job, task, file and data element levels (Stallings, 2003). The IBM S/390 is an example of UMA architecture.

In the case of NUMA architectures the memory access time of processors differs depending on which region of the main memory is accessed. A subclass of NUMA system is cache coherent NUMA (CC-NUMA) where cache coherence is maintained among the caches of various processors. The main advantage of a CC-NUMA system is that it can deliver effective performance at higher levels of parallelism than UMA architecture.

#### *Message Passing Multicomputer*

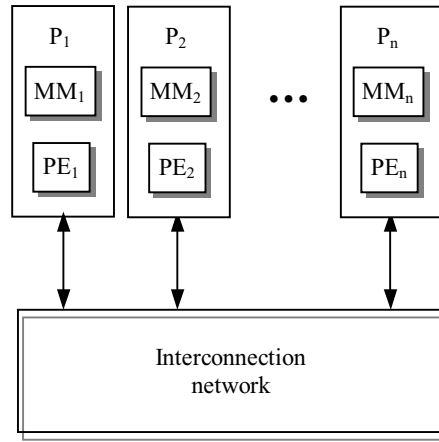
A distributed memory architecture is different from a shared memory architecture in that each unit of this architecture is a complete computer building block including the processor, memory and I/O system. A processor can access the memory, which is directly attached to it. Communication among the processors is established in the form of I/O operations through message signals and bus networks. For example, if a processor needs data from another processor it sends a signal to that processor through an interconnected bus network demanding the required data. The remote processor then responds accordingly. Certainly, access to local memory is faster than access to remote processors. Most importantly, the further the physical distance to the remote processor, the longer it will take to access the remote data. On one hand this architecture suffers from the drawback of requiring direct communication from processor to processor, on the other hand, the bottleneck problem of shared memory architecture does not exist. A general form of shared memory architecture is shown in Figure 2.10.

The speed performance of distributed memory architecture largely depends upon how the processors are connected to each other. It is impractical to connect each processor to the remaining processors in the system through independent cables. It can work for a very low number of processors but becomes nearly impossible as the number of processors in the system increases. However, attempts have been made to overcome this problem and as a result several solutions have emerged. The most common of these is to use specialised bus networks to connect all the processors in the system in order that each processor can communicate with any other processor attached to the system.

### **2.3.3 Classification Based on Interconnections between PEs and Memory Modules**

Parallel architectures are also classified in terms of interconnecting network arrangements for communication among the various PEs included in the

architecture. In fact, this classification is quite specific to MIMD architectures as they, generally, comprises multiple PEs and memory modules. The various interconnecting communication networks used for establishing communication schemes among the PEs of a parallel architecture include: linear, shared single bus, shared multiple bus, crossbar, ring, mesh, star, tree, hypercube and complete graph. Among these interconnecting networks, linear, mesh, ring, star, tree, hypercube and complete graph are static connection structures whereas shared single bus, shared multiple bus and crossbar are dynamic interconnection structures as they are reconfigurable under system control (Hays, 1988).



**Figure 2.10.** Distributed-memory multiprocessor

#### *Linear Network*

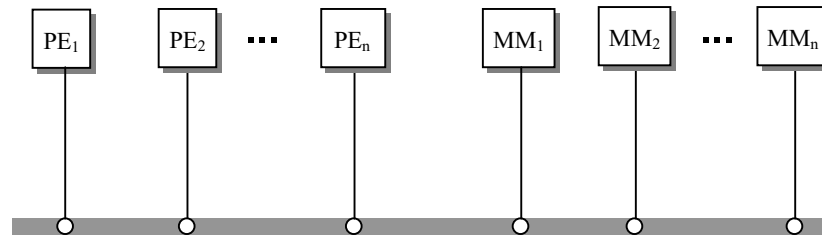
A number of nodes are connected through buses in a linear format to form a network of nodes as shown in Figure 2.11. Every node, except the nodes at the two ends, in this configuration is directly connected to two other nodes. Thus, to connect  $n$  nodes in this configuration  $n-1$  buses are required and the maximum internodes distance is  $n-1$ .



**Figure 2.11.** Linear interconnection structure

#### *Single Shared Bus Network*

The single shared bus interconnection structure, shown in Figure 1.12, is widely used in parallel architectures. A number of PEs and memory units are connected to a single bus in this case, through which communication is established among the PEs and memory units connected to it.



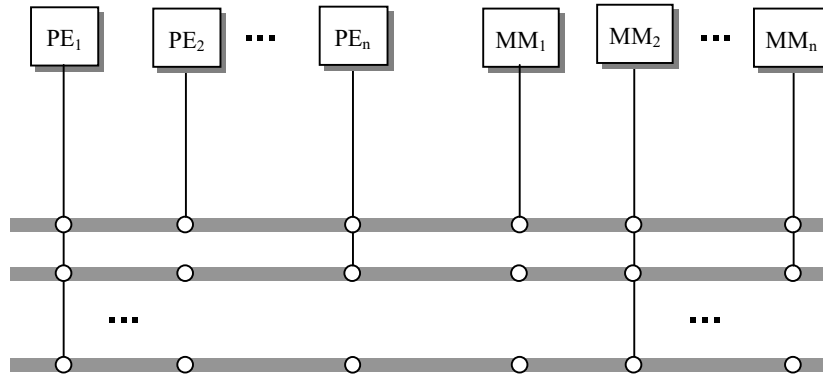
**Figure 2.12.** Single bus interconnection structure

The operation of the single bus interconnection structure, in its simplest form, is as follows: when a processor issues a read request to a memory location it holds the bus until it receives the expected data from the memory module. It will require some time for the memory module to access the data from the appropriate location. The transfer scheme also will need some time and another request from any processor will not be initiated until the transfer is completed, which means the bus will remain idle for a certain amount of time that can be as high as two-thirds of the total time required for the transfer. This problem has been overcome by using a split transfer protocol scheme whereby the bus can handle a number of requests from different processors to different memory modules. In this case after transferring the address of the first request the bus starts the transfer of the next request so that two requests are executed in parallel. If none of the two requests has completed, the bus can be assigned a third request. When the first memory module completes its access cycle, the bus is used to send the data to the destination processor. As another module completes its access cycle the data is transferred using the bus, and the process continues. The split transfer protocol increases the performance of the bus at the cost of complexity. The complexity increases due to maintaining synchronisation and coordination among the requests, processors and memory modules.

One of the limitations of single bus interconnection is that a large number of processors and memory modules cannot be connected to a bus. The number of modules to be connected with the bus could be increased by using a wider bus with increased bandwidth. However, the bandwidth of a single bus is limited by the connection for the use of the bus and by the increased propagation delays caused by the electrical loadings when many modules are connected (Hamacher *et al.*, 2002).

#### *Multiple Shared Buses Network*

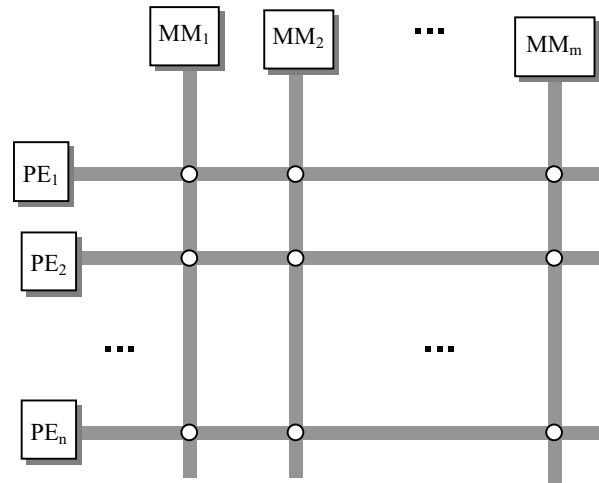
The single shared bus network cannot cope with large numbers of PEs and memory units as mentioned above. Multiple shared bus networks are used in such cases. The structure of a multiple shared bus network is shown in Figure 2.13, where each processor and memory are connected to one or more of the available buses, each of which possesses all the attributes of an independent system bus. Besides reducing the communication load per bus, a degree of fault tolerance is provided, since the system can be designed to continue operation, possibly with reduced performance, if an individual bus fails (Hays, 1988).



**Figure 2.13.** Multiple buses interconnection structure

*Crossbar Interconnection Network*

The structure of crossbar architecture is shown in Figure 2.14. In this architecture, all the PEs and memory modules are interconnected through a multibus crossbar network system where subscript  $m$  denotes the memory and  $n$  denotes the PEs. The crossbar architecture becomes very complex as the number of memory modules and PEs increases.

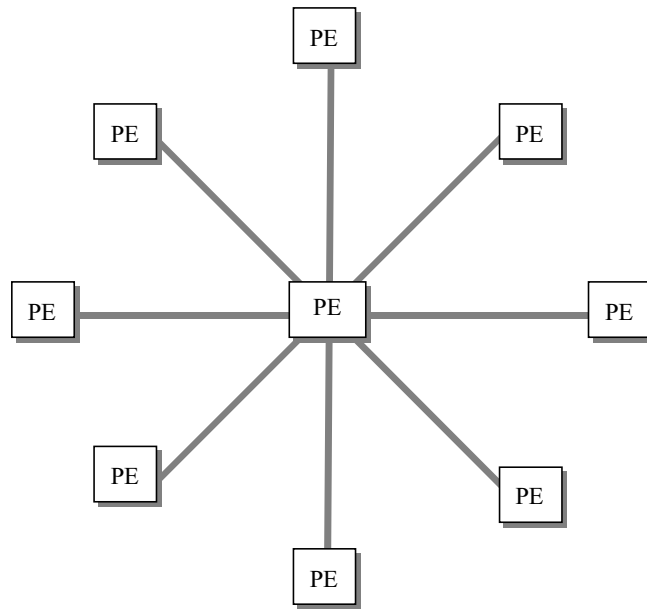


**Figure 2.14.** Crossbar interconnection structure

*Star Interconnection Network*

The star interconnection, as shown in Figure 2.15, is one of the simplest interconnection networks. In this configuration  $n-1$  buses are required to connect

$n$  nodes and the maximum internode distance is 2. A node in this structure can communicate with any other node through the node in the centre.



**Figure 2.15.** Star interconnection structure

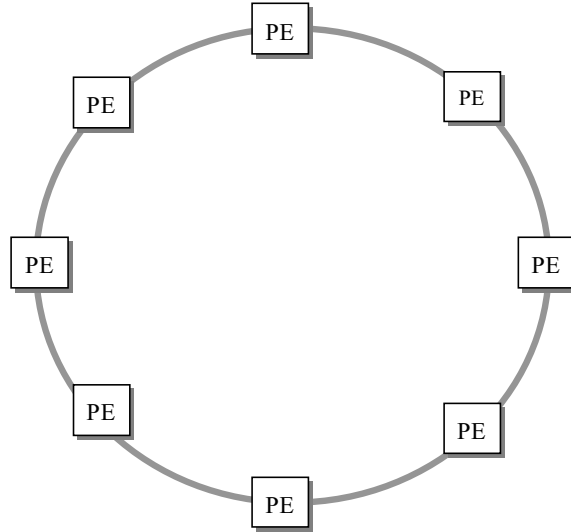
#### *Ring Interconnection Network*

The ring network, shown in Figure 2.16, is also one of the simplest interconnection topologies. This interconnection is very easy to implement. In the case of ring interconnection  $n$  buses are required to connect  $n$  nodes and the maximum internodes distance is  $n/2$ . Rings can be used as building blocks to form other interconnection structures such as mesh, hypercube and tree. A ring-based two-stage tree structure is shown in Figure 2.17. However, the highest-level ring could be a bottleneck for traffic in this case. Commercial machines such as Hewlett-Packard's Exemplar V2600 and Kendal Square Research's KSR-2 have been designed using ring networks.

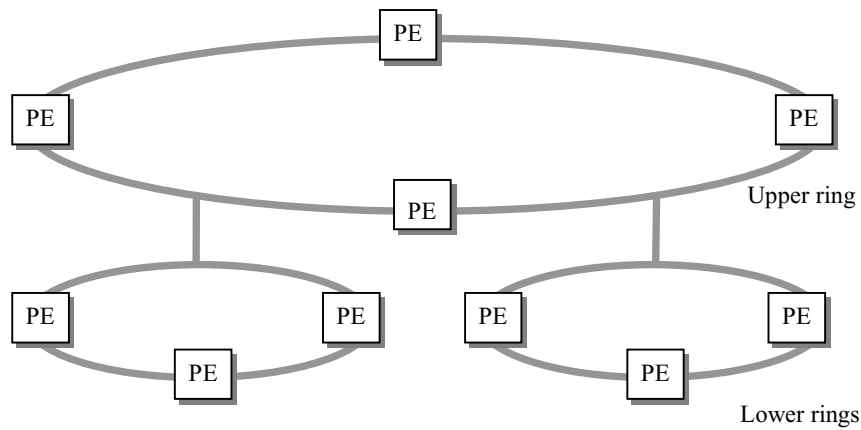
#### *Tree Interconnection Network*

Tree structure is another important and useful interconnection topology. There could be a number of levels in a tree structure. The general form of an  $n$ -level tree structure is shown in Figure 2.18. In this case any intermediate node acts as a medium to establish communication between its parents and children. Through this mechanism communication could also be established between any two nodes in the structure. A tree structure can be highly effective if a small portion of traffic goes through the root node otherwise due to bottleneck problems performance

deteriorates rapidly. The possibility of bottleneck problems is less in a flat tree structure where there is a large number of nodes at the higher levels.



**Figure 2.16.** Ring interconnection structure

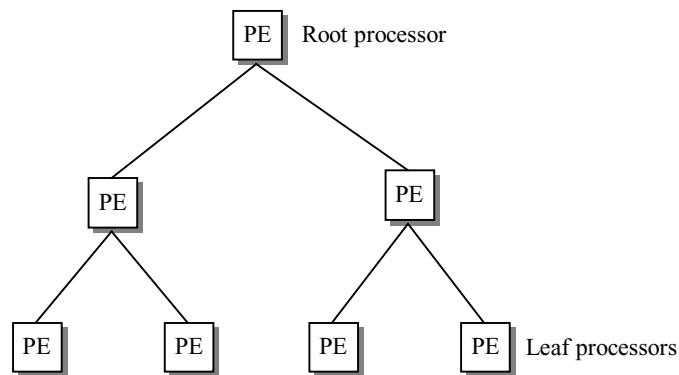


**Figure 2.17.** Two-stage tree networks based on ring networks

#### *Hypercube Interconnection Network*

Hypercube is a popular interconnection network architecture, especially for NUMA multiprocessors. An  $n$ -dimensional hypercube can connect  $2^n$  nodes each of which includes a processor, a memory module and some I/O capability. A three-dimensional hypercube is shown in Figure 2.19. The edges of the cube represent bi-directional communication links between two neighbouring nodes. The nodes

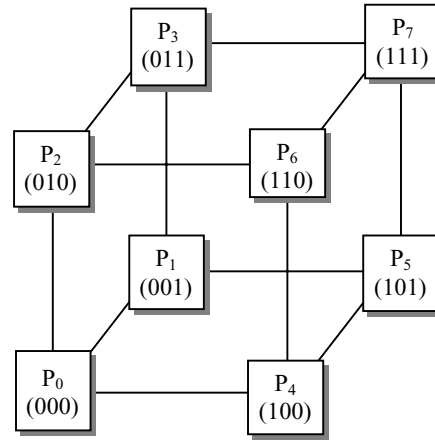
are normally labelled using binary addresses in a way that the addresses of the two neighbouring nodes differ by one bit position. Transferring messages from one node to another in a hypercube structure is accomplished with the help of binary addresses assigned to each of the nodes. In this transferring scheme the binary address of the source node and the destination nodes are compared from least to most significant bits and transfer to the destination is performed through some intermediate nodes in between. For example, the transfer of message from node  $P_i$  to a node  $P_j$  takes place as follows. First the binary addressees of  $P_i$  and  $P_j$  are compared from least to most significant bits. Suppose they differ in bit position  $p$ . Node  $P_i$  then sends a message to the neighbouring node  $P_k$  whose address differs from  $P_i$  in bit position  $p$ . Node  $P_k$  then passes the message to its appropriate neighbours using the same scheme. The message gets closer to the destination node with each of these passes and finally reaches it after several passes. Consider, for example, that node  $P_3$  in Figure 2.19 wants to send a message to node  $P_6$ . It will require two passes through node  $P_2$ .



**Figure 2.18.** Tree interconnection structure

The hypercube structure is very reliable. If a faulty link is detected while passing a message from source to destination node through the shortest route; the message can be passed using another route. A hypercube is homogeneous in nature, as the system appears the same when viewed from any of its outside nodes. Thus, programming the hypercube is simple because all nodes can execute the same programs on different data when collaborating on a common task (Hays, 2003).

Many commercial multiprocessors have used hypercube interconnections including the Intel iPSC. A seven-dimensional hypercube has been used in this machine using 128 nodes. The NCUBE's NCUBE/ten used 1024 nodes in a 10-dimensional hypercube. However, the hypercube structure has lost much of its popularity since the advent of the mesh interconnection structure as an effective alternative (Hamacher *et al.*, 2002).



**Figure 2.19.** Hypercube interconnection structure

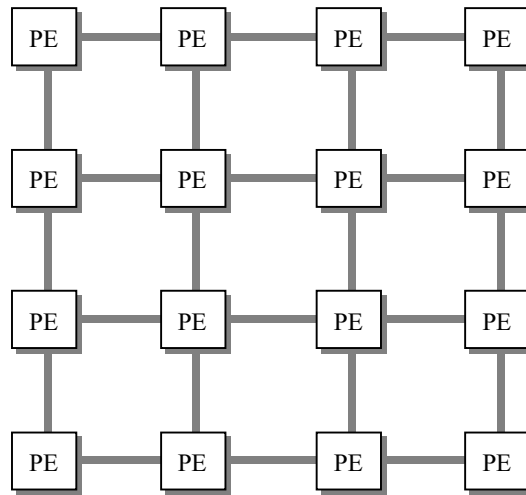
#### *Mesh and Torus Interconnection Network*

Mesh is a popular interconnection network structure used to connect large numbers of nodes. It came into being in the 1990s as an alternative to hypercube in large multiprocessors. A 16-node mesh structure is shown in Figure 2.20. To formulate a mesh structure comprising  $n$  nodes  $2(n - n^{0.5})$  buses are required and the maximum internodes distance is  $2(n^{0.5} - 1)$ . Routing in a mesh is established in various ways. One of the simplest and most popular ways is to choose the path between a source node  $n_i$  and a destination node  $n_j$  then proceed in the horizontal direction from  $n_i$  to  $n_j$ . When the column in which  $n_j$  resides is reached the transfer proceeds in the vertical direction along that column. The Intel's Paragon is a well-known mesh-based multiprocessor. If a wraparound connection is made between the nodes at opposite edges the result is a network that consists of a set of bi-directional rings in the  $X$  direction connected by a similar set of rings in the  $Y$  direction. This network is called a Torus (Hamacher *et al.*, 2002). The average latency in a torus is less than in a mesh at the expense of complexity. Fujitsu's AP3000 is a torus connection based machine.

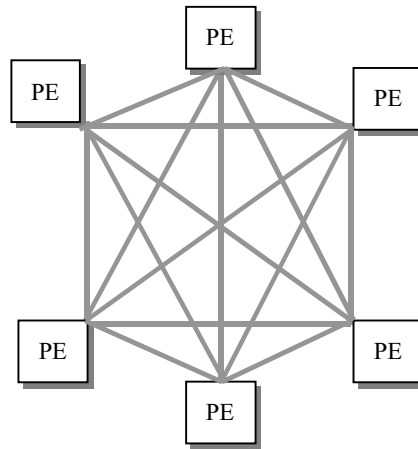
#### *Complete Graph Interconnection Network*

In a complete graph interconnection structure several processors are connected in the complete graph format as depicted in Figure 2.21. Here, each node can directly communicate with any other node without going through or touching any intermediate node. However, it requires many buses. For a complete graph with  $n$  nodes the number of buses required is  $n(n-1)/2$  and the maximum internode distance is 1.





**Figure 2.20.** Mesh interconnection structure



**Figure 2.21.** Complete graph interconnection structure

#### *Switching or Dynamic Interconnection Structures*

Dynamic parallel architectures are reconfigurable under system control and the control is generally achieved through different kinds of switching circuits. One such switch is shown in Figure 2.22, which is an AND gate controlling the connection between two lines namely  $m$  and  $n$ . When line  $n$  is high (say, a binary 1) indicating that a connection is required to be made with the line  $m$ , the control line will go high and the connection will be established.

Another two-state switching element is shown in Figure 2.23. Each switch has a pair of input buses  $x_1$  and  $x_2$ , and a pair of output buses  $y_1$  and  $y_2$ , assisted by

some form of control mechanism. The buses connected to the switch could be used to establish processor-to-processor or processor-to-memory links. The switch  $S$  has two states, determined by a control line, the through or direct state, as depicted in Figure 2.23, where  $y_1 = x_1$  (i.e.,  $y_1$  is connected to  $x_1$ ) and  $y_2 = x_2$  and a cross state where  $y_1 = x_2$  (i.e.,  $y_1$  is connected to  $x_2$ ) and  $y_2 = x_1$ . Using  $S$  as a building block, multistage switching networks of the type can be constructed for use as interconnection networks in parallel computers.

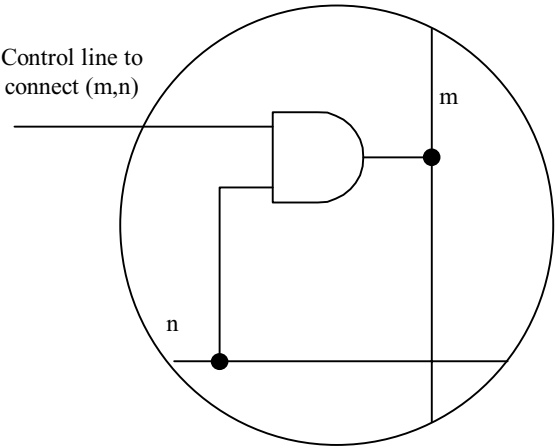


Figure 2.22. A typical crossbar switch

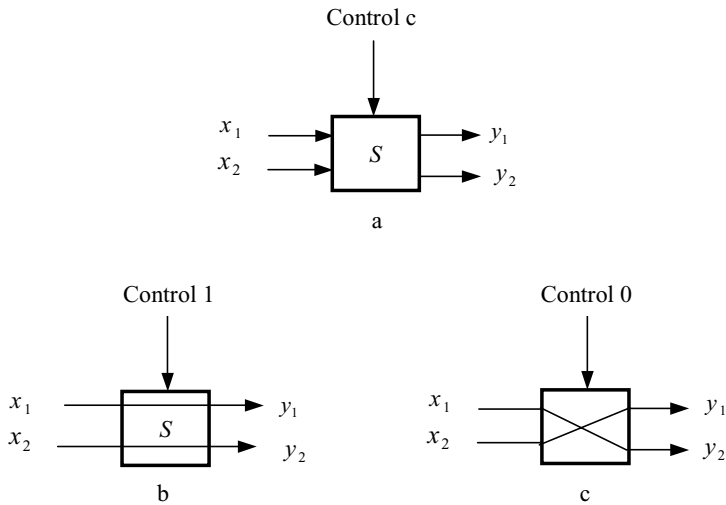


Figure 2.23. Two-states switching mechanisms

A three-stage switching network of this type is shown in Figure 2.24. The network contains 12 switching elements and is intended to provide dynamic connections between the processors. By setting the control signals of the switching elements in various ways, a large number of different interconnection patterns is possible (Hays, 1988). The number of stages, the fixed connections linking the stages, and the dynamic states of the switching elements, in general, determines the possibilities.

A comparison of features of a selected set of interconnection structures is given in Table 2.1.

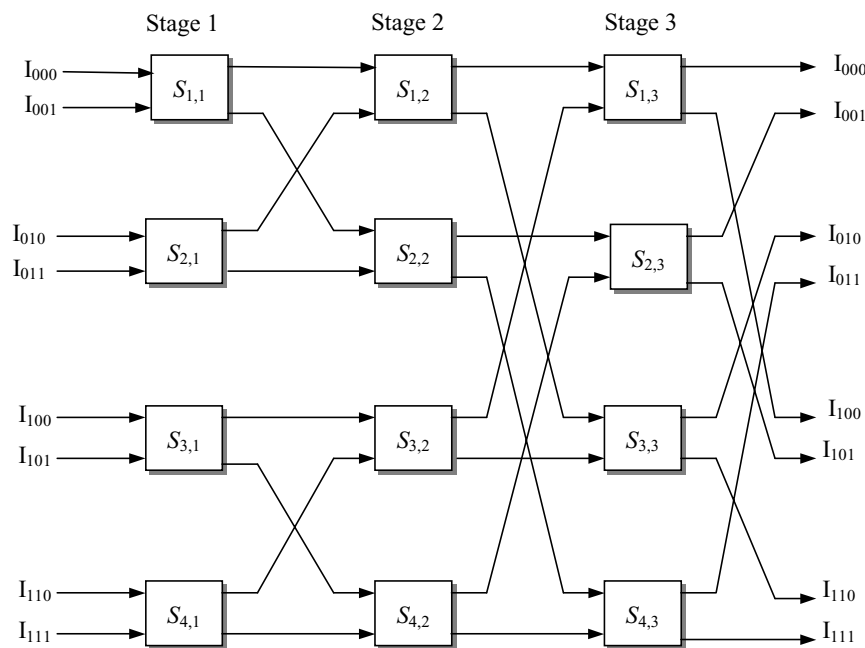


Figure 2.24. Three-stage switching network

### 2.3.4 Classification Based on Characteristic Nature of Processing Elements

Parallel architectures are also classified in terms of the nature of the PEs comprising them. An architecture may consist of either only one type of PE or various types of PEs. The different types of processors that are commonly used to form parallel architectures are described below.

#### *CISC Processors*

The acronym CISC stands for Complex Instruction Set Computer. It is a type of processor that uses a complex, powerful instruction set capable of performing

many tasks including memory access, arithmetic calculations and address calculations. Some distinctive features of CISC processors are as follows:

**Table 2.1.** Comparison of features of selected interconnection structures

Network type	Connections/PE	Maximum distance	Bandwidth	Scalability
Bus	1	2	Low	Poor
Crossbar	2	2	High	Good
Ring	2	$n/2$	Low	Good
Complete graph	$n-1$	1	High	Poor
Torus	4	$\sqrt{n}$	Good locally	Good
Hypercube	$\log_2 n$	$\log_2 n$	Good	Good

- CISC instruction sets are large and powerful.
- CISC instructions are executed slowly as each instruction is normally capable of doing many things.
- CISC processors are comparatively difficult to program.
- CISC architectures have pipelines and more registers.
- CISC processors handle only a relatively low number of operations.

CISC processors are generally used in desktop machines. The Motorola 68x0 and the Intel 80x86 families are examples of CISC processors.

#### *RISC Processors*

The abbreviation RISC stands for Reduced Instruction Set Computer. RISC processors have a number of distinguishing characteristics, some of which are as follows:

- RISC processors handle more operations than CISC processors.
- Execution of instructions in a RISC processor is faster than in their CISC counterpart.
- RISC processors support pipelined instruction execution.
- RISC processors contain large number of registers, most of which can be used as general-purpose registers.
- RISC processors are simple to program.

Current RISC processors include the M600-series PowerPC (Motorola/IBM), i960 (Intel), SPARC (Sun), ARM (Advanced RISC Machines), and Am 29000-series (Advanced Micro Devices).

#### *DSP and Vector Processors*

DSP chips are specially designed to execute DSP algorithms and applications such as FFT, correlation, convolution and digital filtering. Such algorithms are used extensively in a variety of DSP applications such as radar, sonar, and weather forecasting. As most DSP operations require additions and multiplications together, DSP processors usually possess adders and multipliers, which can be used in parallel within a single instruction. DSP chips are also capable of handling multiple memory access in a single instruction cycle. One of the major differences between DSP chips and general-purpose processors is that DSP chips are required to deal with real-world problems frequently and they are designed to do so. TMS320C4x, DSP563xx, and DSP96002 are examples of DSP chips.

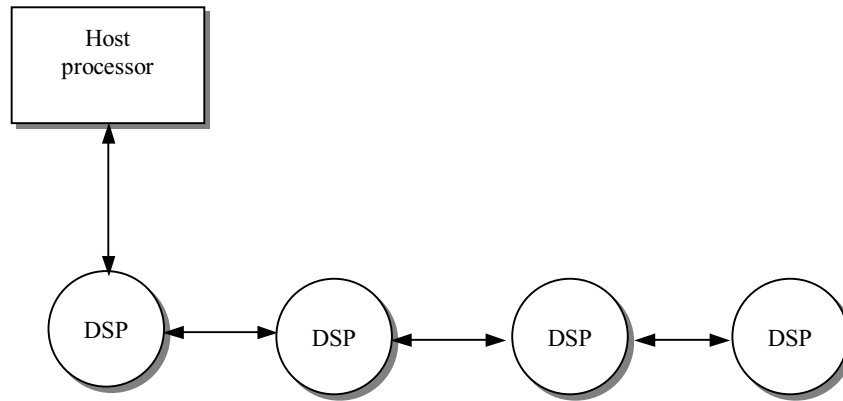
Vector processors are designed to execute vector-intensive algorithms faster than other types of general-purpose and specialised processors. In fact, many algorithms are of regular nature and contain numerous matrix operations. Vector processors are very efficient at executing these types of algorithms. Examples of vector processors are the Intel i860 and i960.

#### *Homogeneous and Heterogeneous Parallel Architectures*

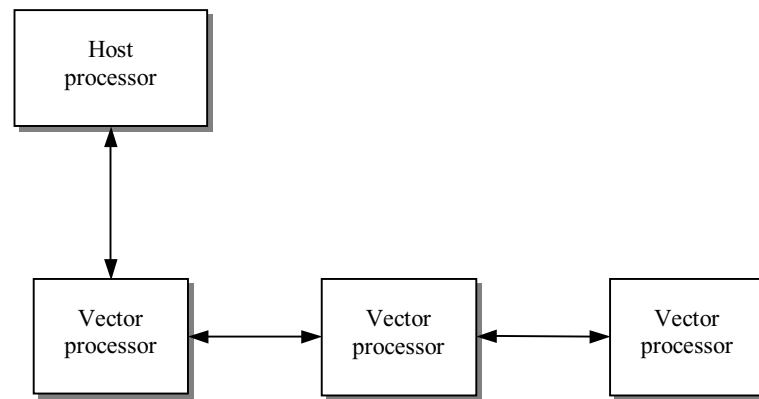
In a conventional parallel system all the PEs are identical. This architecture can be regarded as homogeneous. Figure 2.25 shows the homogeneous architecture of DSP chips and Figure 2.26 shows the homogeneous architecture of vector processors. However, many algorithms are heterogeneous, as they comprise functions and segments of varying computational requirements. Thus, heterogeneous architectures are designed to incorporate diverse hardware and software components in a heterogeneous suite of machines connected by a high-speed network to meet the varied computational requirements of a specific application (Tan and Siegel, 1998). In fact, heterogeneous architectures represent a more general class of parallel processing system. The implementation of an algorithm on a heterogeneous architecture, having PEs of different types and features, can provide a closer match with the varying computing requirements and, thus, lead to performance enhancement. A typical heterogeneous architecture is shown in Figure 2.27, which comprises RISC processors, DSP processors and a vector processor.

### **2.3.5 Specific Types of Parallel Architectures**

Various forms of parallel processors are evolving to cope with complex algorithms and short sample time requirements. Some of the specialised forms of parallel processors are described below.



**Figure 2.25.** Homogeneous architecture of DSP processors

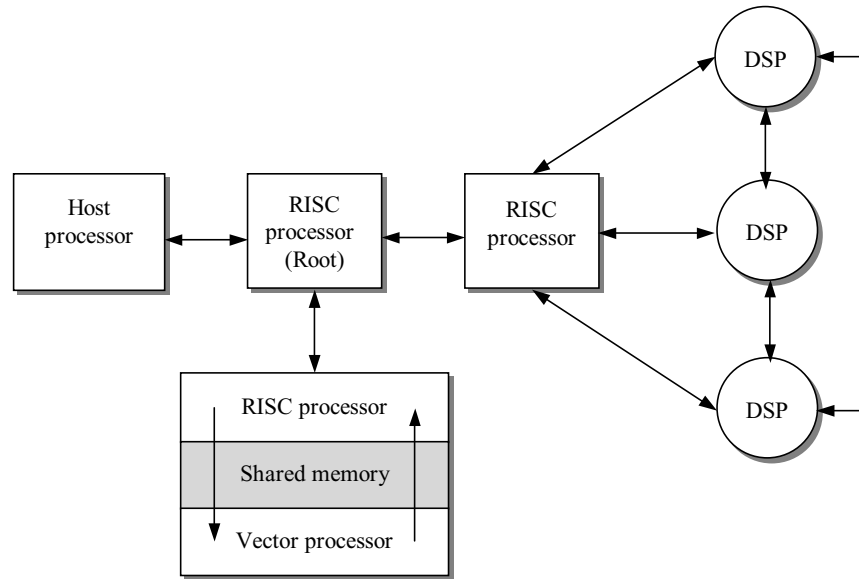


**Figure 2.26.** Homogeneous architecture of vector processors

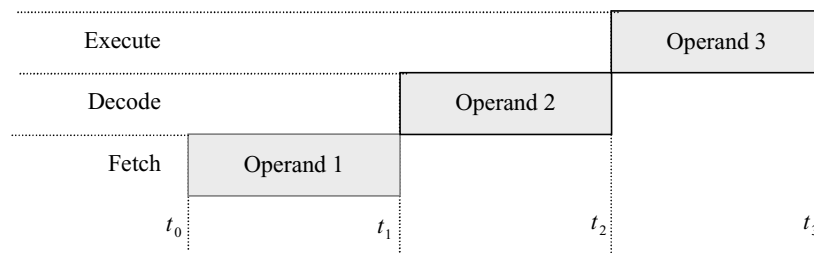
#### *Pipeline Architecture*

Pipeline is a very widely used parallel architecture, designed to execute pipelined instructions. Pipeline is an MISD type processor. However, it could be of MIMD type as well, depending upon the structures and operations.

In the case of pipeline execution while one instruction is executed, the next instruction in the sequence is decoded, while a further one is fetched. The processor consists of a sequence of stages and the operands are partially executed at each stage and the fully processed result is obtained after the operands have passed through all the stages. A three-stage pipelined processing mechanism is shown in Figure 2.28. As shown, when operand 3 is being executed after having been fetched and decoded, operand 2 is being decoded after having been fetched and operand 1 is being fetched. All stages are busy at all times. In contrast in sequential processing when one stage is busy the other two remain idle.

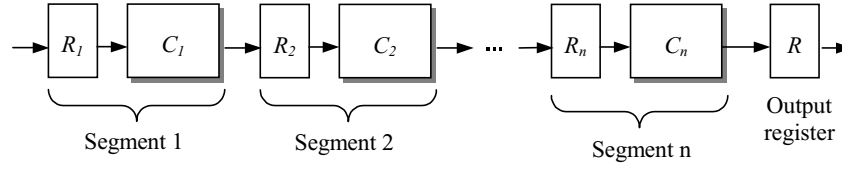


**Figure 2.27.** Heterogeneous architecture



**Figure 2.28.** Three-stage pipeline processing

A pipeline processor consists of a number of stages called segments, each segment comprising an input register and a processing unit. An  $n$ -stage pipeline processor is shown in Figure 2.29. The registers play their role as buffers compensating for any differences in the propagation delays through the processing units (Hays, 1988). Generally, the whole process is controlled by a common clock signal. All the registers change their state synchronously at the start of a clock period of the pipeline. Each register then receives a new set of data from the preceding segment except the first register, which receives data from an external source. In each clock period, all the segments transfer their processed data to the next segment and compute a new set of results.



**Figure 2.29.** A general  $n$ -stage pipeline processing structure

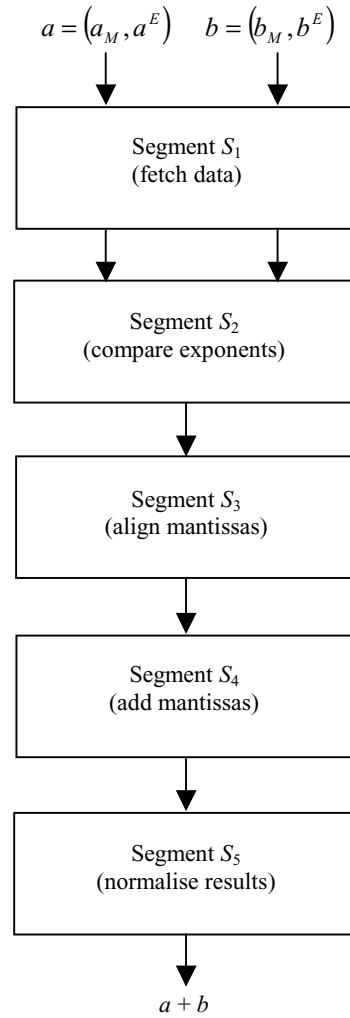
From the operand point of view, pipeline is categorised into two types, namely, the instruction pipeline and arithmetic pipeline. Instruction pipelines are designed to speed up the program control functions of a computer by overlapping the processing of several different instructions namely fetch, decode and execute. Arithmetic pipelines are designed to execute special classes of operands very fast. These arithmetic operations include, multiplication, floating-point operations and vector operations.

#### *Example of an Arithmetic Pipeline*

The concepts of instruction and arithmetic pipelining are similar. However, at the implementation level an arithmetic pipeline is relatively complex. To develop an understanding of pipelining, an example arithmetic pipeline for floating-point addition is illustrated here. Figure 2.30 shows a five-segment floating-point adder pipeline, where  $a$  denotes a sequence of floating-point numbers,  $a_M$  denotes the mantissa of the sequence and  $a^E$  denotes the exponent of the sequence.  $b$  denotes another sequence of floating-point numbers with  $b_M$  and  $b^E$  the mantissa and exponent, respectively. Let two sequences of floating point (normalised) numbers be added using a five-segment floating-point adder, as shown in Figure 2.30. An example of a five-segment floating-point operation is shown in Figure 2.31, where each of the five segments can contain a pair of partially processed scalar operands ( $a_i, b_i$ ). Buffering in the segments ensures that  $S_i$  only receives, as inputs, the results computed by segments  $S_{i-1}$  during the preceding clock period. If the pipeline clock period is  $T$  seconds long, i.e., the execution time of each segment, then it takes a total time of  $XT$  to compute a single sum  $a_i + b_i$ , where  $X (= 5)$  represents the number of segments. This is approximately the time required to do one floating-point addition using a non-pipelined processor, plus the delay due to the buffer registers. Once all five segments of the pipeline are filled with data, a new sum emerges from the fifth segment every  $T$  seconds. Figure 2.32 shows the time space diagram for the process for the first 10 clock cycles. Thus, the time required to perform  $N$  consecutive additions can be calculated as follows:

It follows from the time space diagram in Figure 2.32 that the time required to fill all five segments is  $(5-1)T = 4T$ , therefore for  $X$  segments this will be  $(X-1)T$ , and the total execution time required to compute  $N$  operations will be  $NT + (X-1)T$ , implying that the pipeline's speedup is:





**Figure 2.30.** Five-segment floating-point adder pipeline

$$S(X) = \frac{NXT}{NT + (X-1)T} = \frac{NX}{N + X - 1}$$

For large  $N$  the above approximates to  $S(X) \approx X$ . It is therefore clear that a pipeline with  $X$  segments is  $X$ -times faster than a non-pipelined adder.

Figure 2.33 shows the equivalent block representation of the five-segment floating-point adder pipeline in Figure 2.30, using combinational circuits. Suppose, the time delays of the four segments are  $t_1 = 60$  ns,  $t_2 = 70$  ns,  $t_3 = 100$  ns,

$t_4 = 80$  ns,  $t_5 = 80$  ns and the interface registers have a delay of  $t_r = 10$  ns. The clock period is chosen as  $t_p = t_3 + t_r = 110$  ns. An equivalent nonpipeline floating-point adder will have a delay time  $t_n = t_1 + t_2 + t_3 + t_5 + t_r = 400$  ns. In this case, the pipelined adder has a speedup of  $400/110 = 3.64$  over the non-pipelined adder.

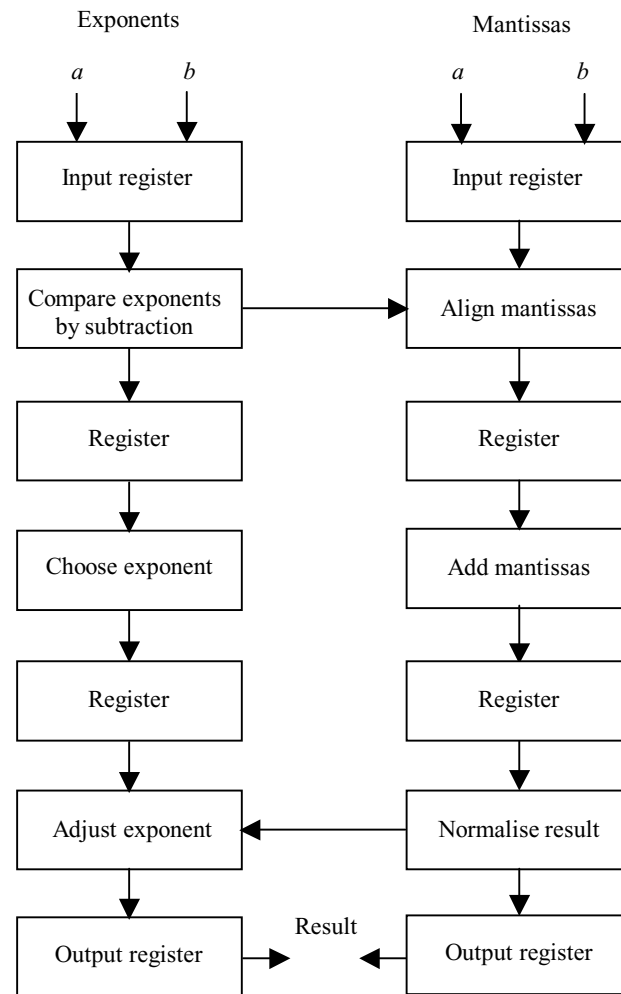
Consider two floating-point numbers,  
 $a_1 = 0.9504 \times 10^3$ , where, mantissa is  $0.9504$  and exponent is  $3$   
and,  $b_1 = 0.8200 \times 10^2$ , where, mantissa is  $0.8200$  and exponent is  $2$ . The activities of the different segments will be as follows:

Segment 1: Fetch the values of  $a$  and  $b$ .  
Segment 2: Compare the exponent of the two numbers. Consider, the larger exponent (which is  $3$ ) as the exponent of the result.  
Segment 3: Align the mantissa of  $b_1$  for exponent  $3$  giving  
 $a_1 = 0.9504 \times 10^3$   
 $b_1 = 0.08200 \times 10^3$ .  
Segment 4: Add mantissas of the two numbers, giving  
 $0.9504 + 0.0820 = 1.0324$ , which is the mantissa of the result.  
Thus, the result before normalisation will be  $c_1 = 1.0324 \times 10^3$ .  
Segment 5: Finally, normalise the result giving  
 $c_1 = 0.10324 \times 10^4$ , available at the output of the pipeline.

**Figure 2.31.** An example for operations of a five-segments floating-point adder

Clock cycle	1	2	3	4	5	6	7	8	9	10
Segment 1	T <sub>1</sub>	T <sub>2</sub>	T <sub>3</sub>	T <sub>4</sub>	T <sub>5</sub>	T <sub>6</sub>				
Segment 2		T <sub>1</sub>	T <sub>2</sub>	T <sub>3</sub>	T <sub>4</sub>	T <sub>5</sub>	T <sub>6</sub>			
Segment 3			T <sub>1</sub>	T <sub>2</sub>	T <sub>3</sub>	T <sub>4</sub>	T <sub>5</sub>	T <sub>6</sub>		
Segment 4				T <sub>1</sub>	T <sub>2</sub>	T <sub>3</sub>	T <sub>4</sub>	T <sub>5</sub>	T <sub>6</sub>	
Segment 5					T <sub>1</sub>	T <sub>2</sub>	T <sub>3</sub>	T <sub>4</sub>	T <sub>5</sub>	T <sub>6</sub>

**Figure 2.32.** Time space diagram



**Figure 2.33.** Pipeline for floating-point addition

**Problem 2.1:** Consider a four-segment pipeline processor executing 200 tasks. Let the time it takes to process a sub-operation in each segment be 20 ns. Determine the speedup of the pipeline system.

**Solution:** The execution time for a non-pipeline system will be

$$4NT = 4 \times 200 \times 20 = 16000 \text{ ns}$$

where number of tasks  $N = 200$  and sub-operation process time for each segment  $T = 20$  ns .

Execution time for a pipeline system will be,

$$(N + 3)T = (200 + 3) \times 20 = 4060 \text{ ns}$$

Therefore, the speedup will be  $= \frac{16000}{4060} = 3.941$  .

#### *Multiple Pipeline*

Multiple pipeline architecture can be defined as a type of parallel architecture, formed using more than single independent pipeline in parallel. Thus, this architecture is a combination of pipeline and MIMD architectures.

#### *Multiple SIMD*

Multiple SIMD is a specific type of MIMD-SIMD architecture. More precisely, it can be defined as an MIMD type connection of a number of independent SIMD architectures. There are a number of control units for these architectures, each of which controls a subset of the PEs.

#### *Dataflow Architecture*

Another novel parallel architecture is the dataflow model. In this case, the program is represented by a graph of data dependencies as shown in Figure 2.34. The graph is mapped over a number of processors each of which is assigned an operation to be performed and the address of each node that needs the result. A processor performs an operation whenever its data operands are available. The operation of dataflow architectures is quite simple and resembles circular pipelining. A processor receives a message comprising data and the address of its destination node. The address is compared against those in a matching store. If the address is present, the matching address is extracted and the instruction is issued for execution. If not, the address is placed in the store for its partner to arrive. When the result is computed, a new message or token containing the result is sent to each of the destinations mentioned in the instructions (Culler *et al.*, 1999).

#### *Systolic and Wavefront Arrays*

Systolic arrays comprise SIMD, MIMD and pipeline architectures. They are driven by a single clock and hence behave like SIMD architectures. However, they differ from SIMD in that each PE has the option to do different operations. The individual array elements, on the other hand, are MIMD processors and pipeline computations take place along all array dimensions. The systolic array also differs from conventional pipelined function units in that the array structure could be non-linear, the pathways between PEs may be multidirectional and each PE may have a small amount of local instruction and data memory (Culler *et al.*, 1999). Replacing the central clock of the systolic arrays with the concept of data flow forms wavefront arrays and hence wavefront arrays can be regarded as an extension of systolic arrays.

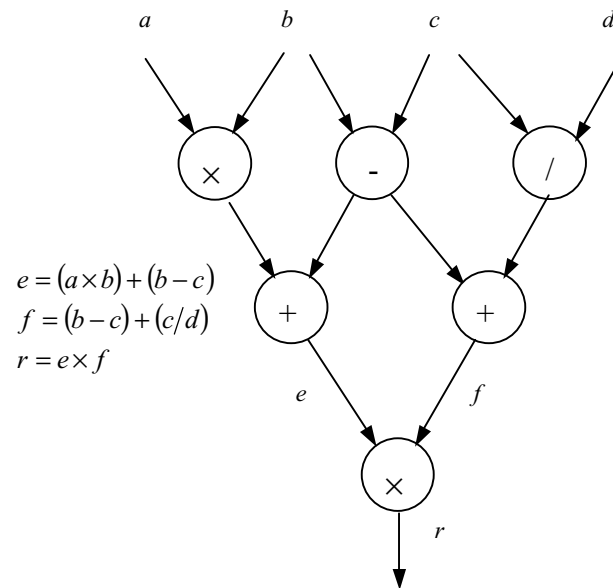


Figure 2.34. Data-flow graph

*Single-program Multiple-data Architecture*

The single-program multiple-data (SPMD) architecture combines the ease of SIMD programming with MIMD flexibility. This system is controlled by a single program and hence the name SPMD.

## 2.4 Summary

A large number of diverse types of parallel architectures are used worldwide. There is also no doubt that there are many other types in the research and/or development stage. However, not all of these suit particular applications. Thus, it is necessary to figure out which parallel architecture would be appropriate for what types of applications. This has essentially been the motivation for the classification of parallel architectures. Flynn first classified parallel architectures based on the instruction and data streams. His classification gives a broad picture of parallel architectures and all parallel architectures could be classified in terms of this broad classification principle. However, this is not enough to fully distinguish one parallel architecture from another and as a result further classifications in terms of more distinctive features have evolved. Such features include memory arrangements, interconnection mechanisms, communication between PEs, memory access time, nature of the processors incorporated in an architecture and so on. For example, when all the processors in an architecture share a single memory, it is called shared memory architecture whereas when each processor uses its own local

memory it is called a distributed memory architecture. A number of architectures have evolved based on the interconnection mechanisms and arrangement of the PEs in the architecture. Homogeneous and heterogeneous types of parallel architecture evolved based on such issues, where a heterogeneous architecture comprises processors with different characteristics and a homogeneous architecture comprises processors with similar characteristics.

## 2.5 Exercises

1. Indicate the major classes of parallel architectures? Describe Flynn's classification of computers.
2. Distinguish between shared memory and distributed memory architectures.
3. What do you understand by UMA and NUMA as used in parallel architectures?
4. Indicate various characteristics of message passing architectures. How does message passing occur in a message passing architecture?
5. Classify parallel architectures on the basis of interconnection networks. Distinguish between static and dynamic interconnection networks.
6. Draw a comparative outline of various interconnection networks.
7. Describe the function of switches used in dynamic interconnection architectures. Briefly explain the working mechanism of a switch.
8. Distinguish between CISC and RISC architectures.
9. Explain the distinctive features of homogeneous and heterogeneous parallel architectures.
10. Indicate the characteristics of vector processor, array processor and DSP devices.
11. What do you understand by pipeline mechanism? Describe the working mechanism of a pipeline architecture.
12. Consider a four-segment pipeline processor executing 4000 tasks. Assume that the time it takes to process a sub-operation in each segment is equal to 30 ns. Determine the speedup for the pipeline system.
13. Consider the time delay of the five segments in the pipeline of Figure 2.33 as:  $t_1 = 45 \text{ ns}$ ,  $t_2 = 30 \text{ ns}$ ,  $t_3 = 95 \text{ ns}$ ,  $t_4 = 50 \text{ ns}$  and  $t_5 = 70 \text{ ns}$ . The delay

time of interface registers is  $t_r = 15 \text{ ns}$ . (a) How long should it take to add 100 pairs of numbers in the pipeline? (b) How can you reduce the total time to about half of the time obtained in part (a)?

14. Illustrate that the speedup of a four-segment floating-point adder for a large number of tasks is nearly 4.
15. How is a systolic array formed? Describe the features of a systolic array and warfront computers.
16. Describe the basic working principles of data-flow architecture.

Parallel Computing for Real-time Signal Processing and  
Control

Tokhi, M.O.; Hossain, M.A.; Shaheed, M.H.

2003, XIV, 254 p. 58 illus., Softcover

ISBN: 978-1-85233-599-1