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*.

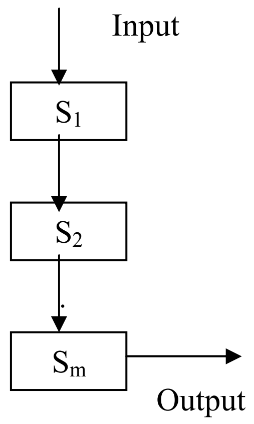

Figure 1: m-Segment Pipeline Processor

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.

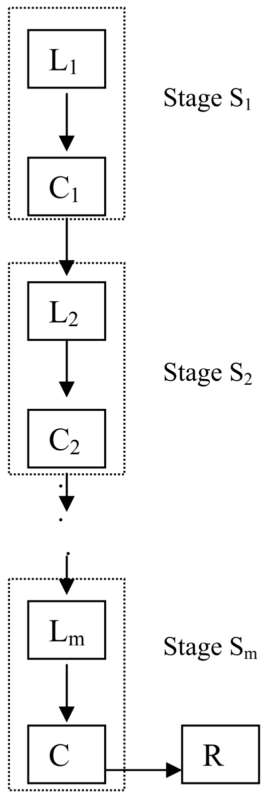

Figure 2: Pipelined Processor

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*).

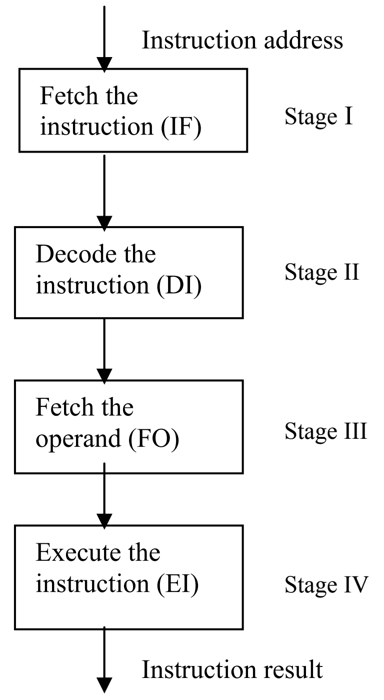

Figure 3: Instruction Pipeline

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.

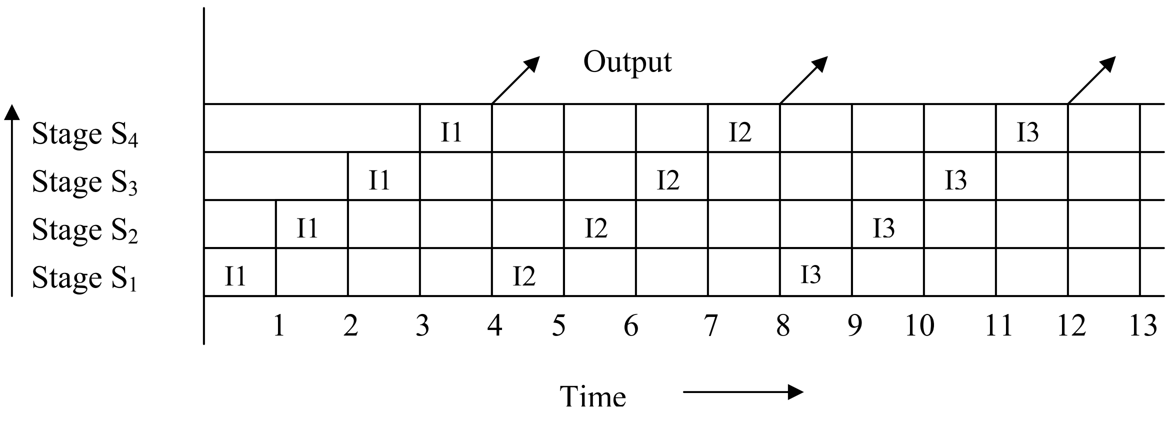

Figure 4(a) Space-time diagram for Non-pipelined Processor

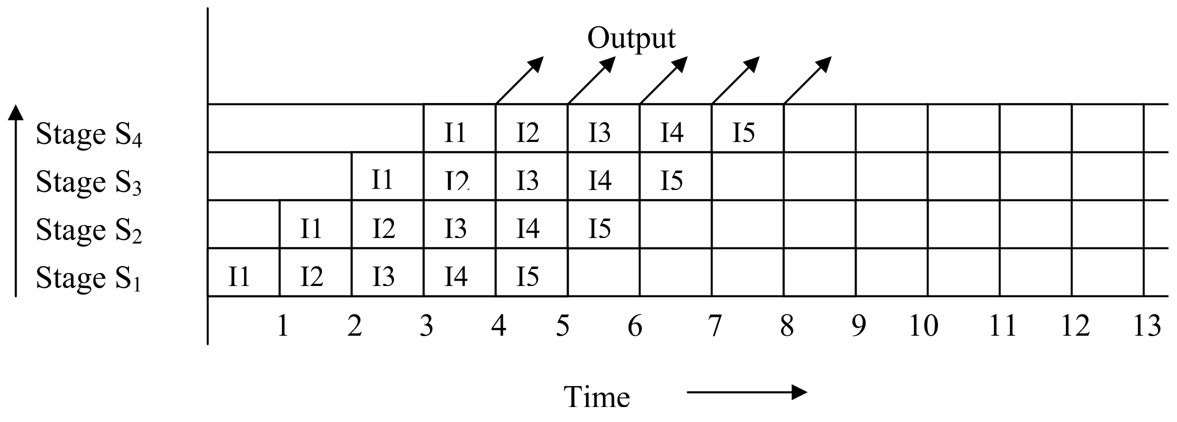

Figure 4(b) Space-time diagram for Overlapped Instruction pipelined Processor

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*.

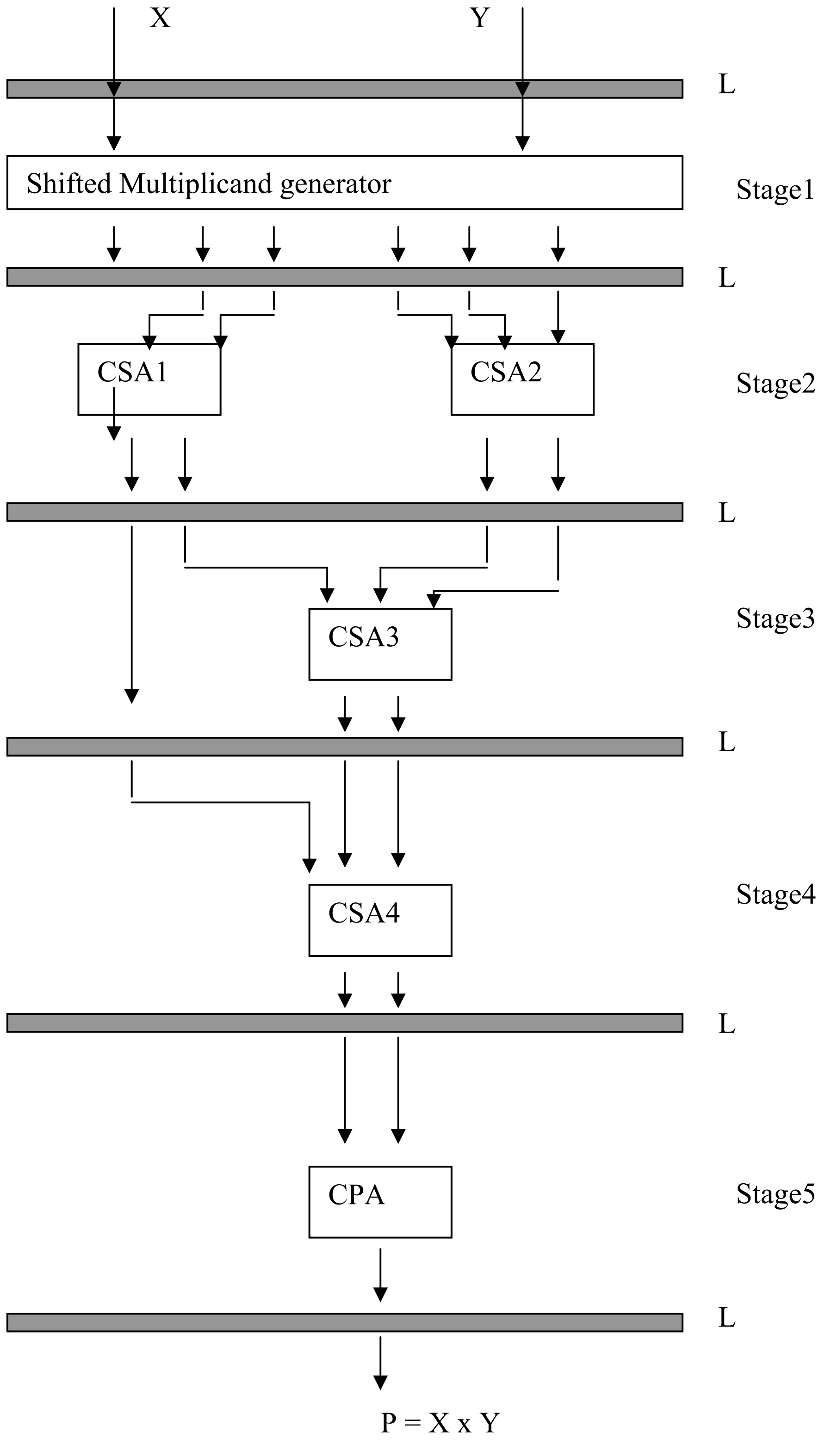

Figure 5: Arithmetic pipeline for Multiplication of two 6-digit fixed numbers

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*.

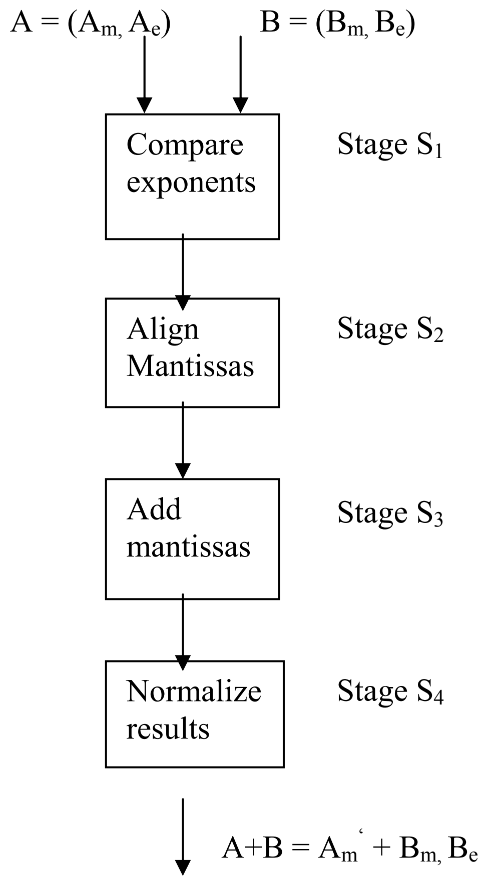

Figure 6: Arithmetic Pipeline for Floating point addition of two numbers

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 $n>=\infty$, 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 -> V

  F2 : V × V -> 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 × V -> 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 -> S

  F5 : V × V -> 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-> V

  F7 : V -> 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:

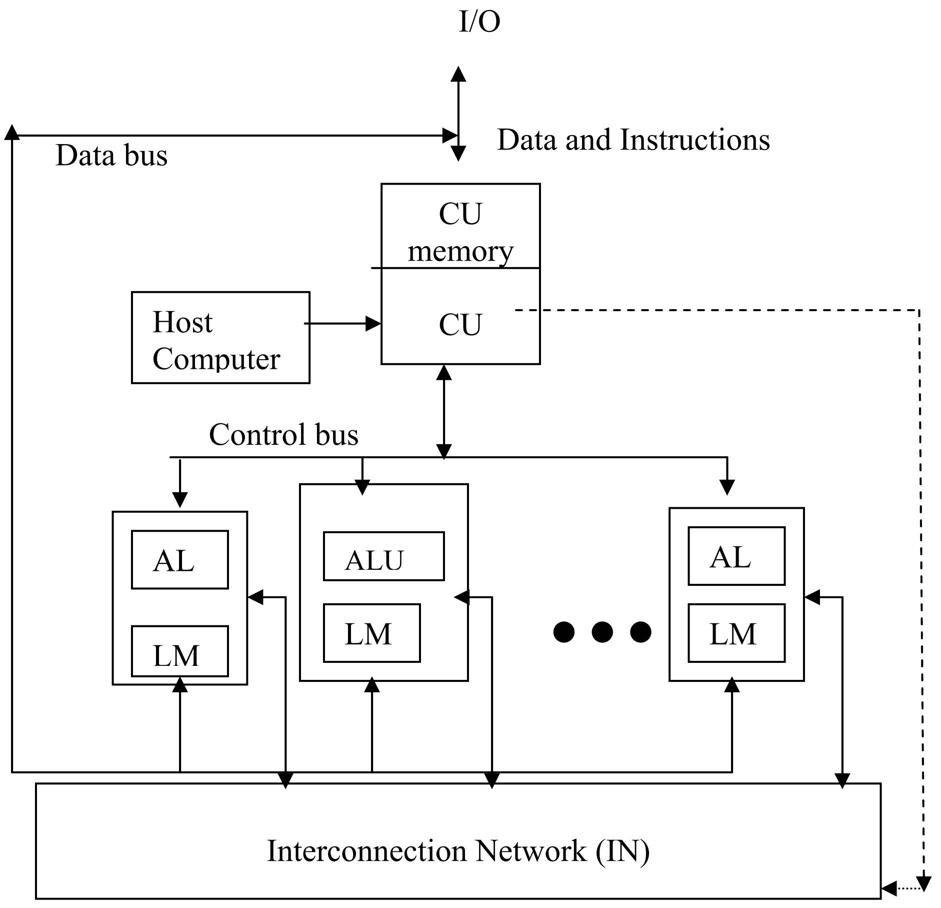

Figure 7: Organisation of SIMD Array Processor

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*.

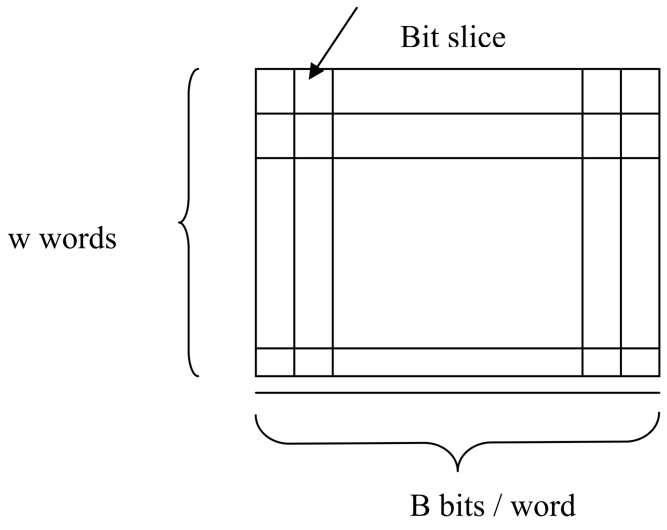

Figure 8: Associative memory

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.

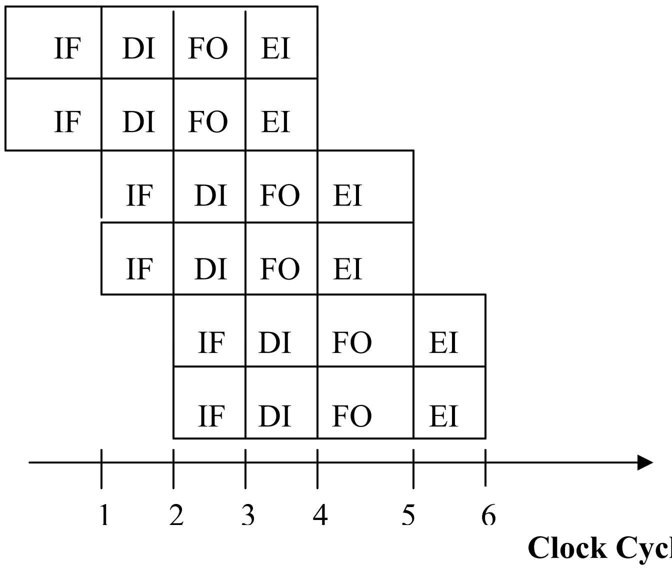

Figure 9: Superscalar Processing of instruction cycle in 4-stage instruction pipeline

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.

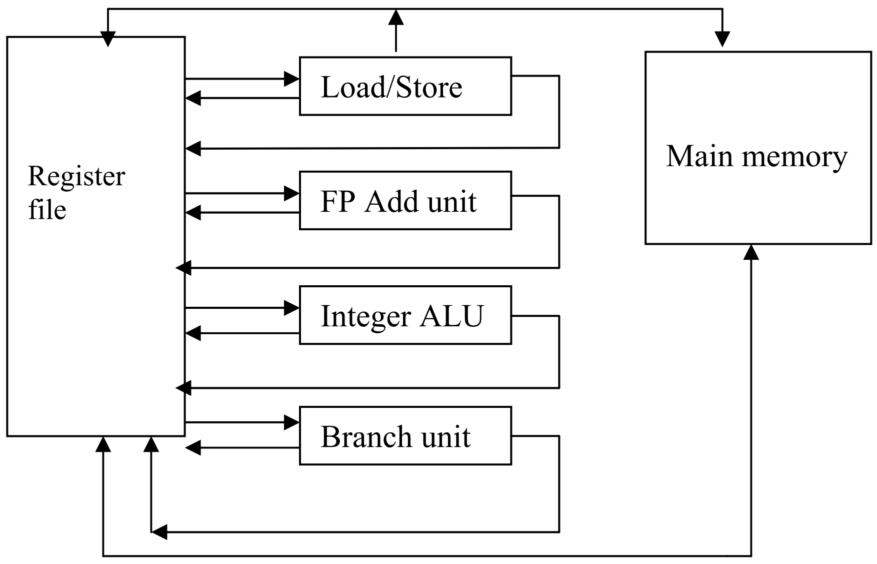

Figure 11: VLIW Processor

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:

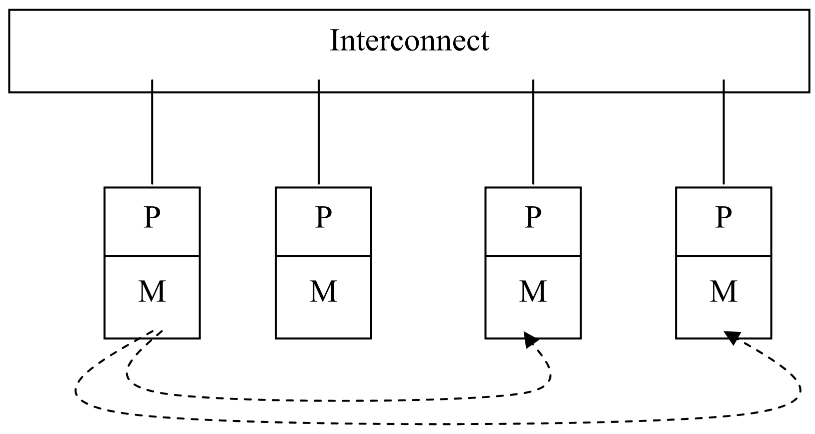

Figure 12: Latency problems in MPP

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.