BSC CSIT 3rd CHAPTER 6

PIPELINE AND VECTOR PROCESSING (LH-6)

Rolisha Sthapit

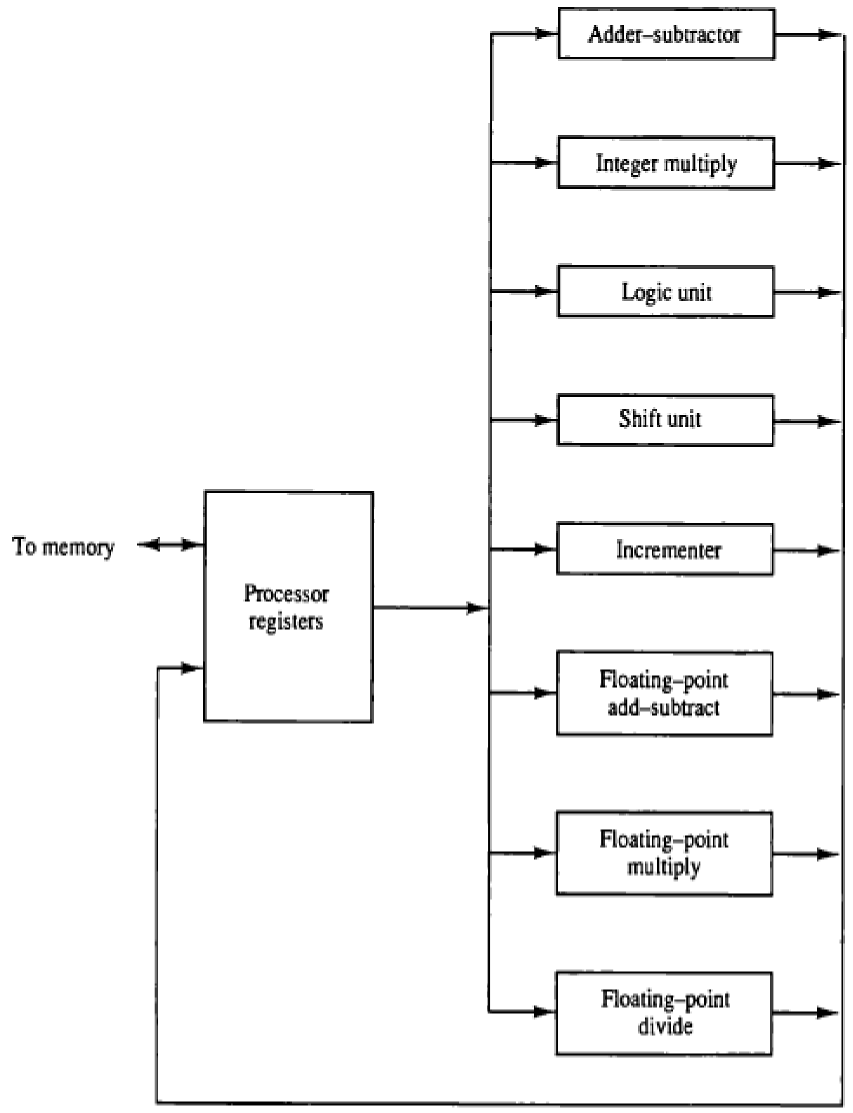
CONTENTS

6.1 Parallel Processing, Flynn's Classification of Computers

Parallel Processing:

- Parallel processing is a term used to denote a large class of techniques that are used to provide simultaneously data processing task for the purpose of increasing the computational speed of computer system.
- It is also used to speed up the computer processing capability and increase its throughput (the amount of task those are completed during given interval of time). It is called parallel computing.
- The system may have two or more ALUs and be able to execute two or more instructions at the same time. It can be viewed from various levels of complexity. At the lowest level, we distinguish between parallel and serial operations by the type of registers used. At higher level, it can be achieved by having multiple functional unit that perform identical or different operations simultaneously.

Figure 9-1 Processor with multiple functional units.



There are variety of ways that parallel processing can be classified:

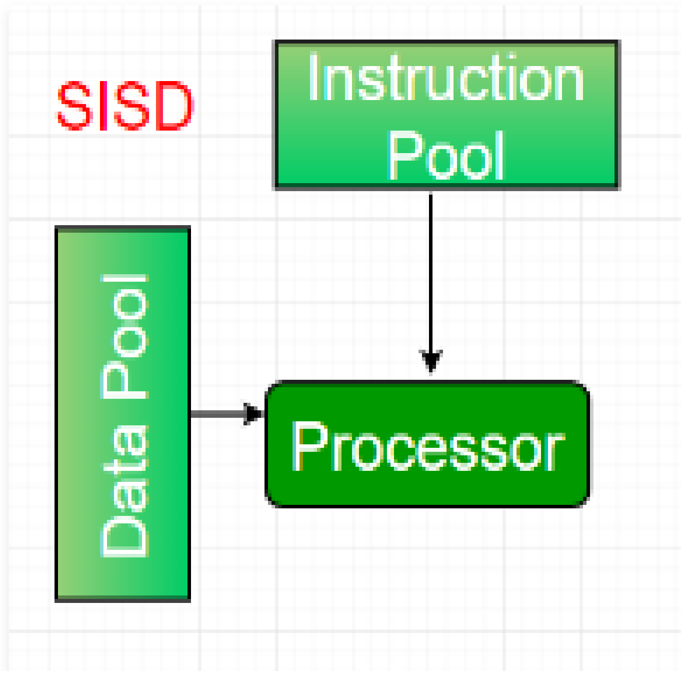
1. Internal organization of the processor
2. Interconnection structure between processors
3. The flow of information through the system.

Flynn’s Classification of Computers:

- There are four groups of computers according to the M.J.Flynn’s classification based on the number of concurrent instruction and data streams manipulated by the computer system.
- The normal operation of a computer is to fetch instructions from memory and execute them in processor.
- The sequence of instructions read from memory constitutes an *instruction stream*.
- The operations performed on the data in the processor constitute a *data stream*.
- Parallel processing may occur in the instruction stream, in the data stream, or in both.
- The Flynn’s classification of computer are:
    - single instruction stream, single data stream (**SISD**)
    - single instruction stream, multiple data stream (**SIMD**)
    - multiple instruction stream, single data stream (**MISD**)
    - multiple instruction stream, multiple data stream (**MIMD**)

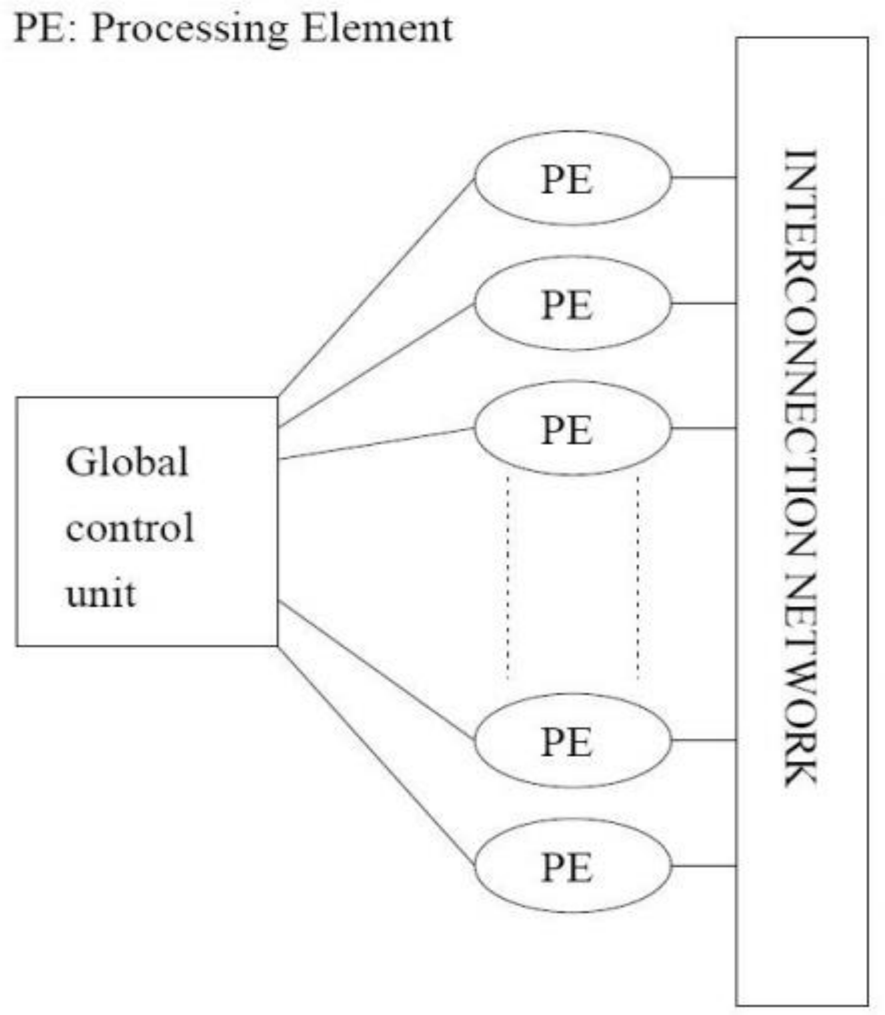
1. Single instruction stream, single data stream (**SISD**):

- Represents the organization of a single computer containing a control unit, processor unit and a memory unit.
- Instructions are executed sequentially and the system may or may not have internal parallel processing capabilities.
- Parallel processing may be achieved by means of multiple functional units or by pipeline processing.



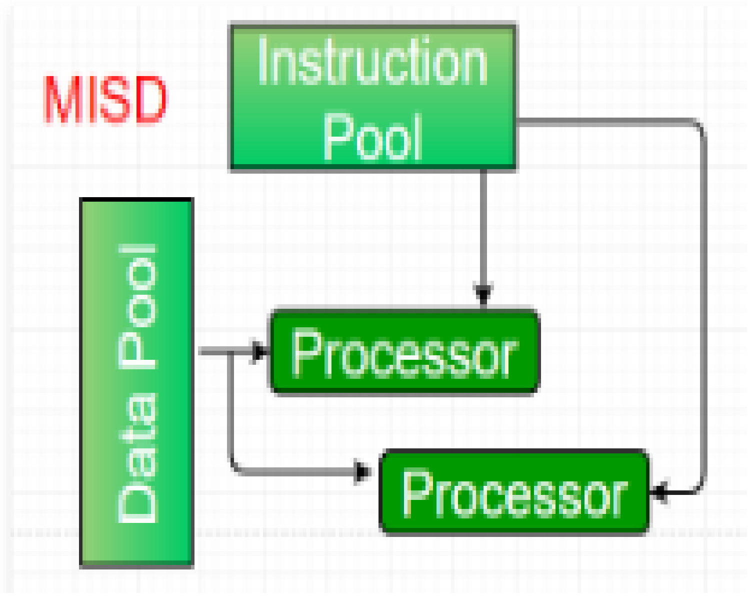
2. Single instruction stream, multiple data stream (**SIMD**):

- Represents the organization that includes many processing units under the supervision of a common control unit.
- All processors receives the same instruction from the control unit but operate on different items of a data.
- The shared memory unit must contain multiple modules so that it can communicate with all the processors simultaneously.
- Application of SIMD is vector and array processing.



3. Multiple instruction stream, single data stream (**MISD**):

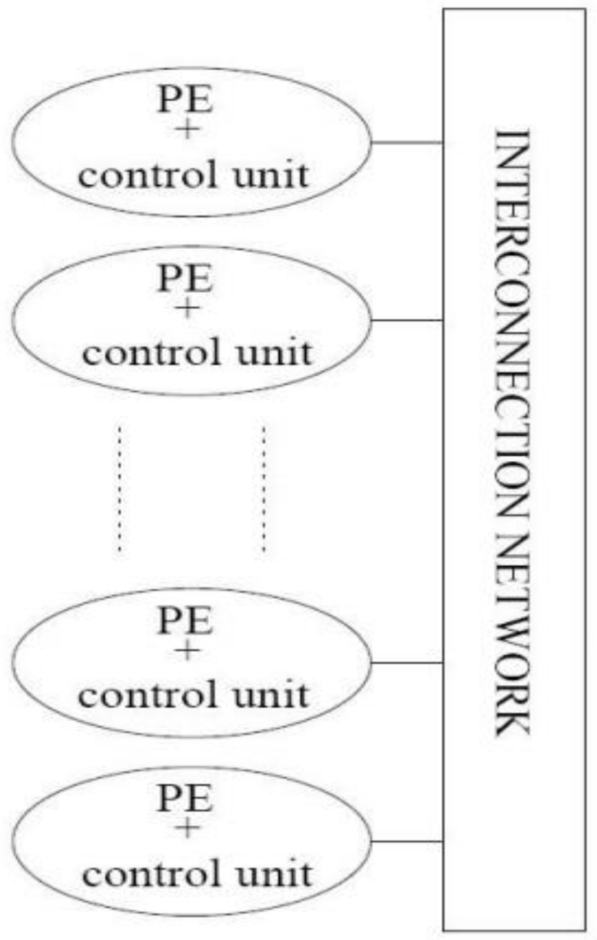
- MISD has many functional units which perform different operations on the same data. It is a theoretical model of computer since no practical system has been constructed using this organization.



4. Multiple instruction stream, multiple data stream (**MIMD**):

MIMD refers to a computer system capable of processing several programs at the same time.

Eg: multiprocessor and multicomputer system.



6.2 Pipelining

- Pipelining is a technique of decomposing a sequential process into sub operations, with each sub process being executed in a special dedicated segment that operates concurrently with all other segments.
- Each segment performs partial processing dictated by the way the task partitioned.
- The result obtained from the computation in each segment is transferred to the next segment in the pipeline. The final result is obtained after the data have passed through all segments.
- It is characteristic of pipelines that several computations can be in progress in distinct segments at the same time.

Example

Suppose we want to perform the combined multiply and add operations with a stream of numbers.

Eg: $A_i*B_i + C_i$ for i=1,2,…..7

Each suboperation is to be implemented in a segment with a pipeline

$R1 \leftarrow A_i$, $R2 \leftarrow B_i$	Input A_i and B_i
$R3 \leftarrow R1*R2$, $R4 \leftarrow C_i$	Multiply and input C_i
$R5 \leftarrow R3+R4$	Add C_i to product

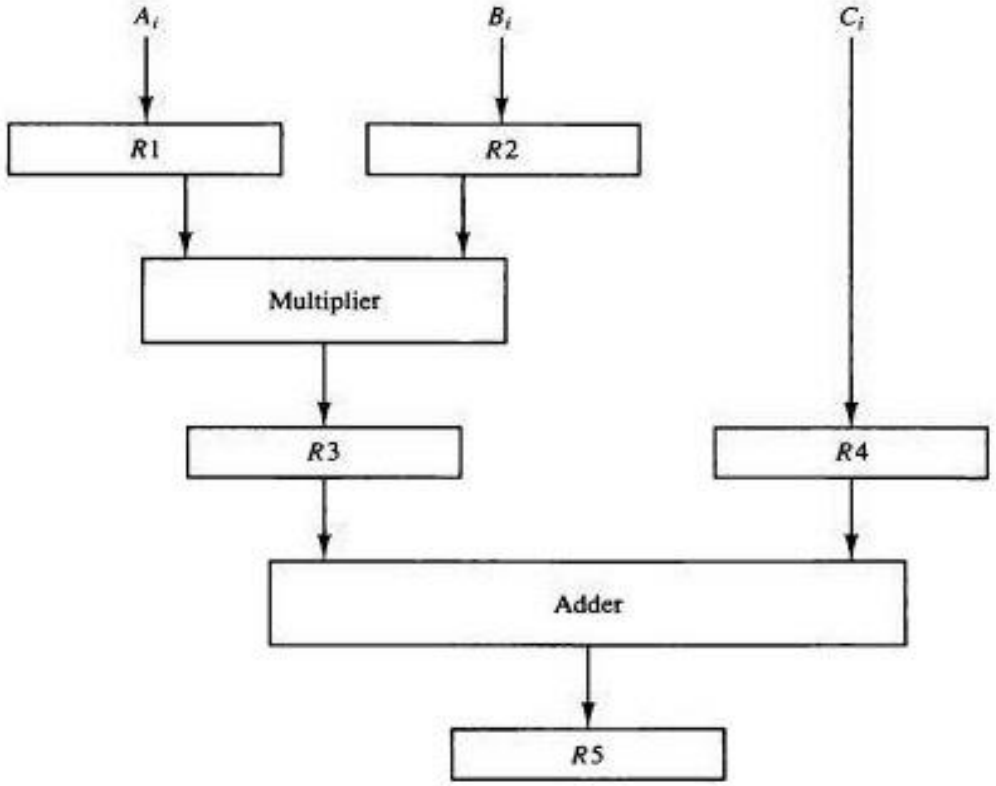


Fig: Example of pipeline processing

TABLE 9-1 Content of Registers in Pipeline Example

Clock Pulse Number	Segment 1		Segment 2		Segment 3
	$R1$	$R2$	$R3$	$R4$	$R5$
1	A_1	B_1	—	—	—
2	A_2	B_2	$A_1 * B_1$	C_1	—
3	A_3	B_3	$A_2 * B_2$	C_2	$A_1 * B_1 + C_1$
4	A_4	B_4	$A_3 * B_3$	C_3	$A_2 * B_2 + C_2$
5	A_5	B_5	$A_4 * B_4$	C_4	$A_3 * B_3 + C_3$
6	A_6	B_6	$A_5 * B_5$	C_5	$A_4 * B_4 + C_4$
7	A_7	B_7	$A_6 * B_6$	C_6	$A_5 * B_5 + C_5$
8	—	—	$A_7 * B_7$	C_7	$A_6 * B_6 + C_6$
9	—	—	—	—	$A_7 * B_7 + C_7$

General Considerations:

Any operation that can be decomposed into a sequence of suboperations of about the same complexity can be implemented by a pipeline processor. The technique is efficient for those applications that need to repeat the same task many times with different sets of data. The general structure of a 4 segment pipeline is given below:

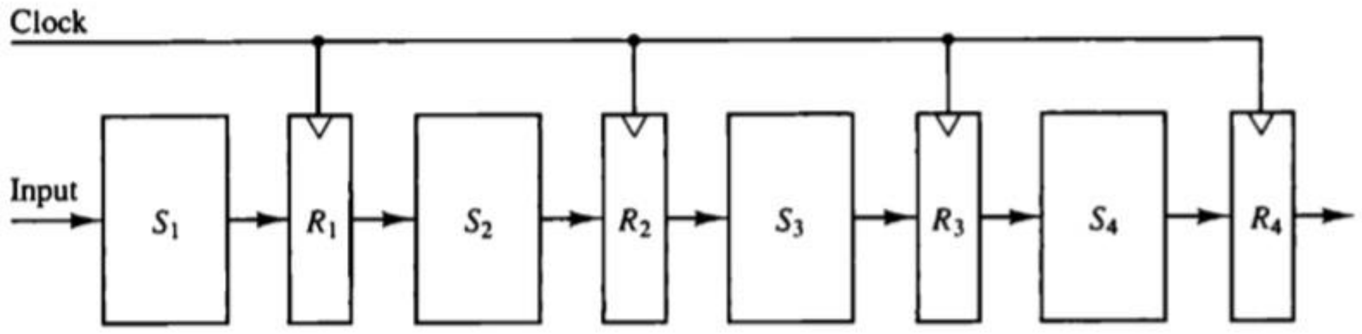


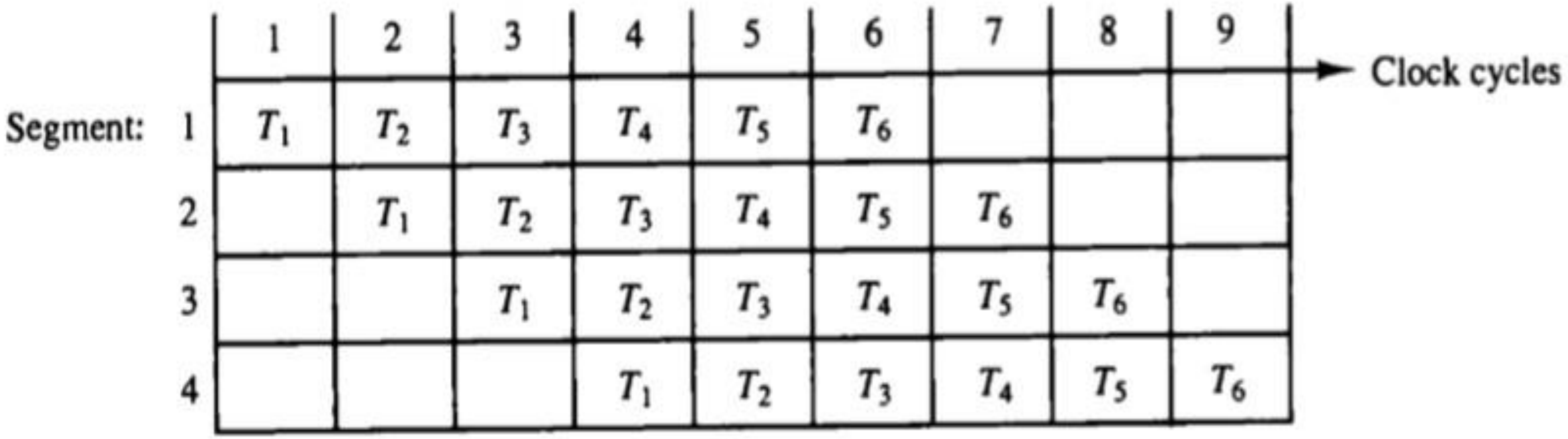
Figure 9-3 Four-segment pipeline.

The operands pass through all four segments in a fixed sequence. Each sequence consists of a combination circuit S_i that performs a suboperation over the data stream flowing through the pipe. The segments are separated by register R_i that hold the intermediate results between the stages.

A **task** is defined as the total operation performed going through all the segments in the pipeline. The behavior of a pipeline can be illustrated with a **space-time** diagram. It shows the segment utilization as a function of time. The space-time diagram of a 4 segment pipeline is given below:

Figure: Space time diagram of 4 segment and 6 tasks

Figure 9-4 Space-time diagram for pipeline.



	1	2	3	4	5	6	7	8	9	→ Clock cycles
Segment: 1	T_1	T_2	T_3	T_4	T_5	T_6				
2		T_1	T_2	T_3	T_4	T_5	T_6			
3			T_1	T_2	T_3	T_4	T_5	T_6		
4				T_1	T_2	T_3	T_4	T_5	T_6	

Assignment:
Draw the space time diagram for a 6 segment pipeline showing the time it takes to process 8 tasks.

Speedup Equation:

- Consider a K-segment pipeline with a clock cycle time t_p to execute n tasks. The first task T_1 require time Kt_p to complete. The remaining (n-1) tasks finish at the rate of one task per clock cycle and will be completed after time (n-1) t_p. Therefore, to complete n task, using a K-segment pipeline requires K+(n-1) clock cycles. The total time to complete the n task is $[Kt_p + (n - 1) t_p] = (K + n - 1) t_p$.

Example:

Segment (K) = 4

Task (n) = 6

Then, total clock cycle = K+(n-1)= 4+(6-1) = 9

- Consider a non–pipeline unit that performs the same operation and takes t_n time to complete each task. The total time required for n tasks would be nt_n.

The speedup of pipeline processing over an equivalent non–pipeline processing is defined by the ratio:

$$\text{Speedup (S)}=\frac{Total\ time\ taken\ by\ non-pipeline\ \ structure\ to\ complete\ n\ taks}{Total\ time\ taken\ by\ pipeline\ \ structure\ to\ complete\ n\ taks}$$

$$S = \frac{nt_n}{(K + n - 1)\ tp}$$

As number of tasks increases, n becomes much larger than K - 1, the Speedup becomes:

$$S = \frac{t_n}{t_p}$$

If we assume that the time it takes to process a task is same in the pipeline and non-pipeline circuits then $t_n = Kt_p$ Then the speed up reduces to

$$S = \frac{Kt_p}{t_p} = K$$

This shows that the theoretical maximum speed that a pipeline can provide is K where K is the number of the segments in the pipeline.

Numerical

1. A non-pipeline system has 50 nanosecond time to process a task. The same task can be processed in a 6 segment pipeline with a clock cycle of 10 nanosecond. Determine the speed ratio of the pipeline for 100 task. What is the maximum speed that can be achieved.

Solution:

tn = 50 ns

Segment (K) = 6

tp = 10 ns

Task (n)= 100

$$S = \frac{ntn}{(K+n-1)tp} = \frac{100*50}{(6+100-1)*10} = 4.76$$

For maximum speed up

$$S = \frac{Kt_p}{t_p} = K = 6$$

2. Calculate pipeline speedup if time taken to complete a task in conventional machine is 25 ns. In the pipeline machine, one task is divided into 5 segments and each sub operation tasks take 4 ns. Number of tasks to be completed is 100. [Ans: 6.01]
3. Calculate the speed up rate of 5-segment pipeline with a clock cycle time 25ns to execute 100 tasks. [Ans=4.8] [tn=Ktp = 5*25 = 125]
4. Consider a 5 segment pipeline where each segment takes three clock cycle and the clock cycle time is 5 ns. If 100 jobs are to be executed then calculate the pipeline speed up.

Solution: K=5 , tp=3*5=15

tn= Ktp= 5*15= 75

n = 100

5. A non–pipeline system takes 100 ns to process a task. The same task can be processed in a six-segment pipeline with time delay of each segment in the pipeline is as follows; 20 ns, 25 ns, 30 ns. Determine the speed of ratio of pipeline for 100 tasks. [Ans:3.17]

Solution,

tn = 100 ns

K = 6

tp = 30 ns

n = 100

$$S = \frac{ntn}{(K+n-1)tp}$$

6. Suppose that time delays of four segments are t1 = 60ns, t2 = 70 ns, t3 = 100 ns, t4 = 80 ns and interface register have a delay of 10 ns. Determine the speed up ratio.

Solution: Here,

tp = 100 + 10 = 110 ns

tn = t1 + t2 + t3 + t4 +tr=60+70+100+80+10 = 320 ns

S= $\frac{tn}{tp}$ = 320/110 = 2.9

6.3 Arithmetic Pipeline

Pipeline arithmetic units are usually found in very high speed computers. They are used to implement floating point operations, multiplication of fixed point numbers, and similar computations encountered in scientific problems.

We will now show an example of a pipeline unit for floating point addition and subtraction. The inputs to the floating point adder pipeline are two normalized floating point binary numbers.

$$X = A \times 2^a$$

$$Y = B \times 2^b$$

The floating point addition and subtraction can be performed in four segments, as shown in figure. The registers labeled R are placed between the segments to store intermediate results.

The suboperations that are performed in the four segments are:

1. Compare the exponents.
2. Align the mantissas.
3. Add or subtract the mantissas.
4. Normalize the result.

- Procedure: The exponents are compared by subtracting them to determine their difference. The larger exponent is chosen as the exponent of the result. The exponent difference determines how many times the mantissa associated with the smaller exponent must be shifted to the right. This produces an alignment of the two mantissas. It should be noted that the shift must be designed as a combinational circuit to reduce the shift time. The two mantissas are added or subtracted in segment 3. The result is normalized in segment 4. When an overflow occurs, the mantissa of the sum or difference is shifted right and the exponent incremented by one. If an underflow occurs, the number of leading zeros in the mantissa determines the number of left shifts in the mantissa and the number that must be subtracted from the exponent.

For simplicity, we use decimal numbers, the two normalized floating-point numbers:

$X = 0.9504 \times 10^3$

$Y = 0.8200 \times 10^2$

The two exponents are subtracted in the first segment to obtain 3-2 = 1. The larger exponent 3 is chosen as the exponent of the result. The next segment shifts the mantissa of Y to the right to obtain

$X = 0.9504 \times 10^3$

$Y = 0.0820 \times 10^3$

This aligns the two mantissas under the same exponent. The addition of the two mantissas in segment 3 produces the sum

$Z = 1.0324 \times 10^3$

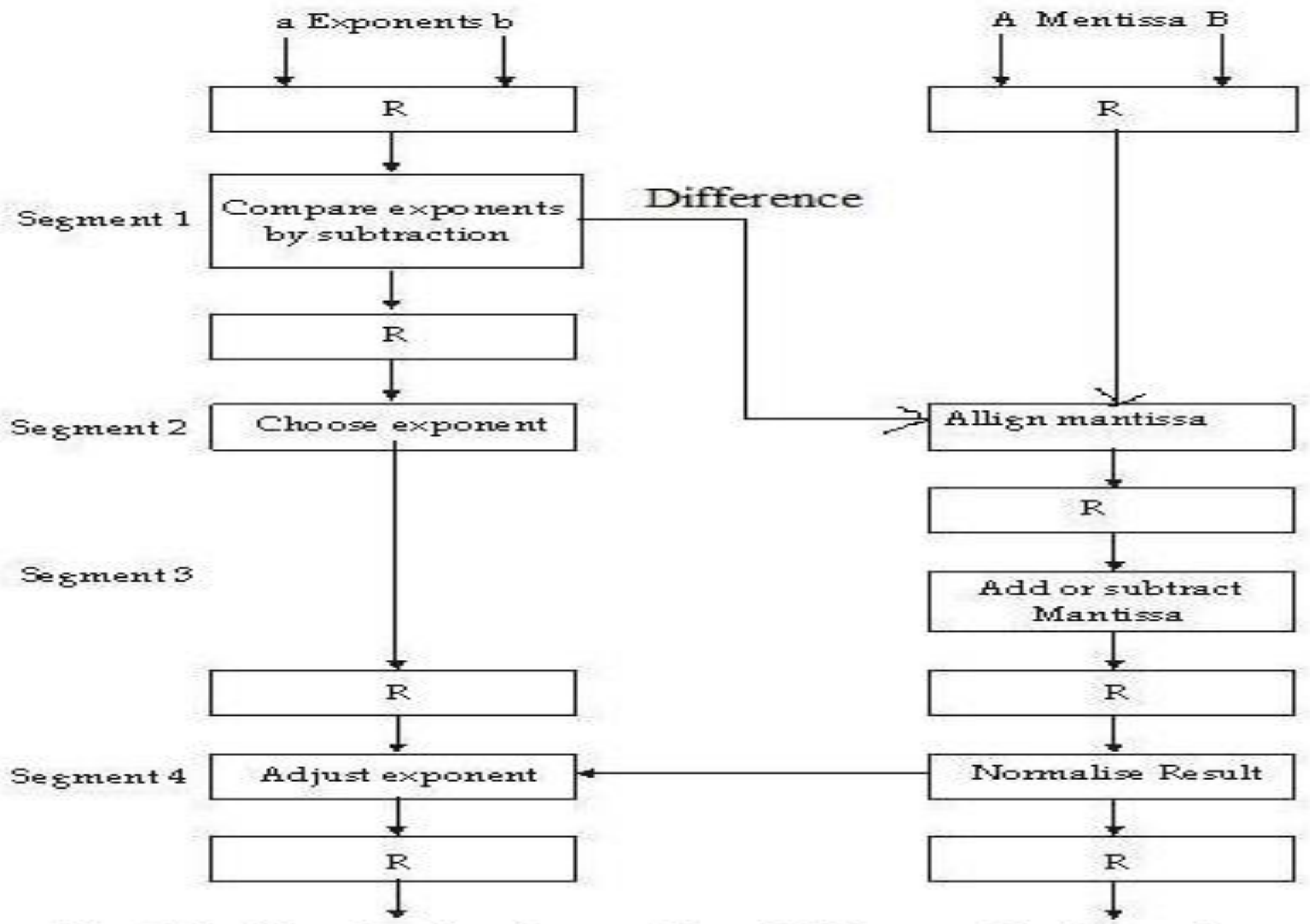


Fig: Pipeline for floating-point addition and subtraction

Instruction Pipeline

- Pipeline processing can not only occur in the *data stream* but in the *instruction stream* as well.
- An instruction pipeline reads consecutive instructions from memory while previous instructions are being executed in other segments. This causes the instruction fetch and execute phases to overlap and perform simultaneous operations. This technique is called instruction pipelining.
- Consider a computer with an instruction fetch unit and instruction execution unit designed to provide 2 segment pipeline. The instruction fetch segment can be implemented by a means of a first-in, first out(FIFO) buffer. This is a type of unit that forms a queue rather than a stack.

- Computers with complex instructions require other phases in addition to the fetch and execute to process an instruction completely. In the most general case, the computer needs to process each instruction with the following sequence of steps:

    i. Fetch the instruction from memory.
    ii. Decode the instruction.
    iii. Calculate the effective address.
    iv. Fetch the operands from memory.
    v. Execute the instruction.
    vi. Store the result in the proper place.

  There are certain difficulties that will prevent the instruction pipeline from operating at its maximum rate. Different segments take different time to operate on the incoming information.

- There are certain difficulties that will prevent the instruction pipeline from operating at its maximum rate. Different segments may take different times to operate on the incoming information. Some segments are skipped for certain operations. For example, a register mode instruction does not need an effective address calculation. Two or more segments may require memory access at the same time, causing one segment to wait until another is finished with the memory.

Example: Four-segment Instruction Pipeline

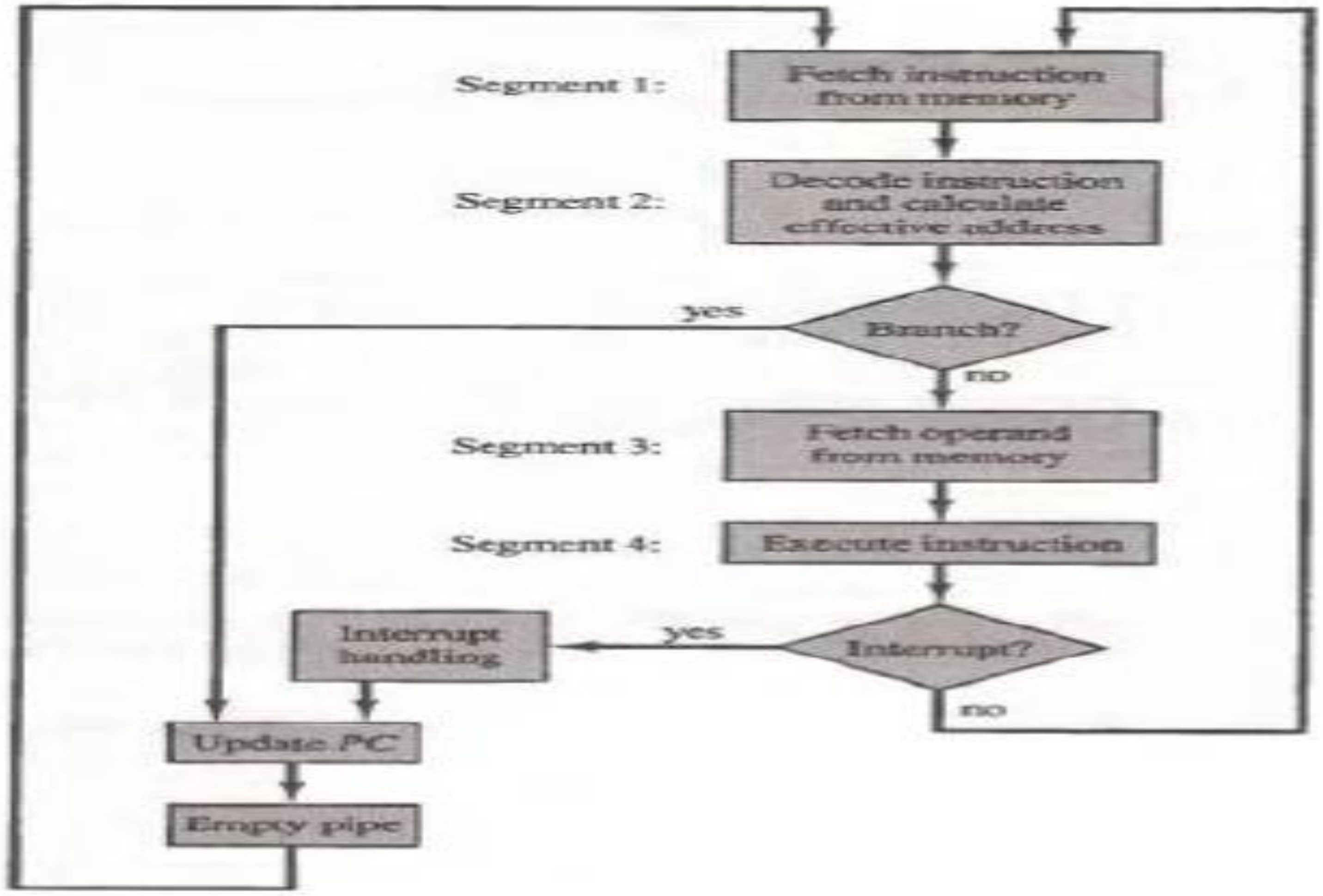
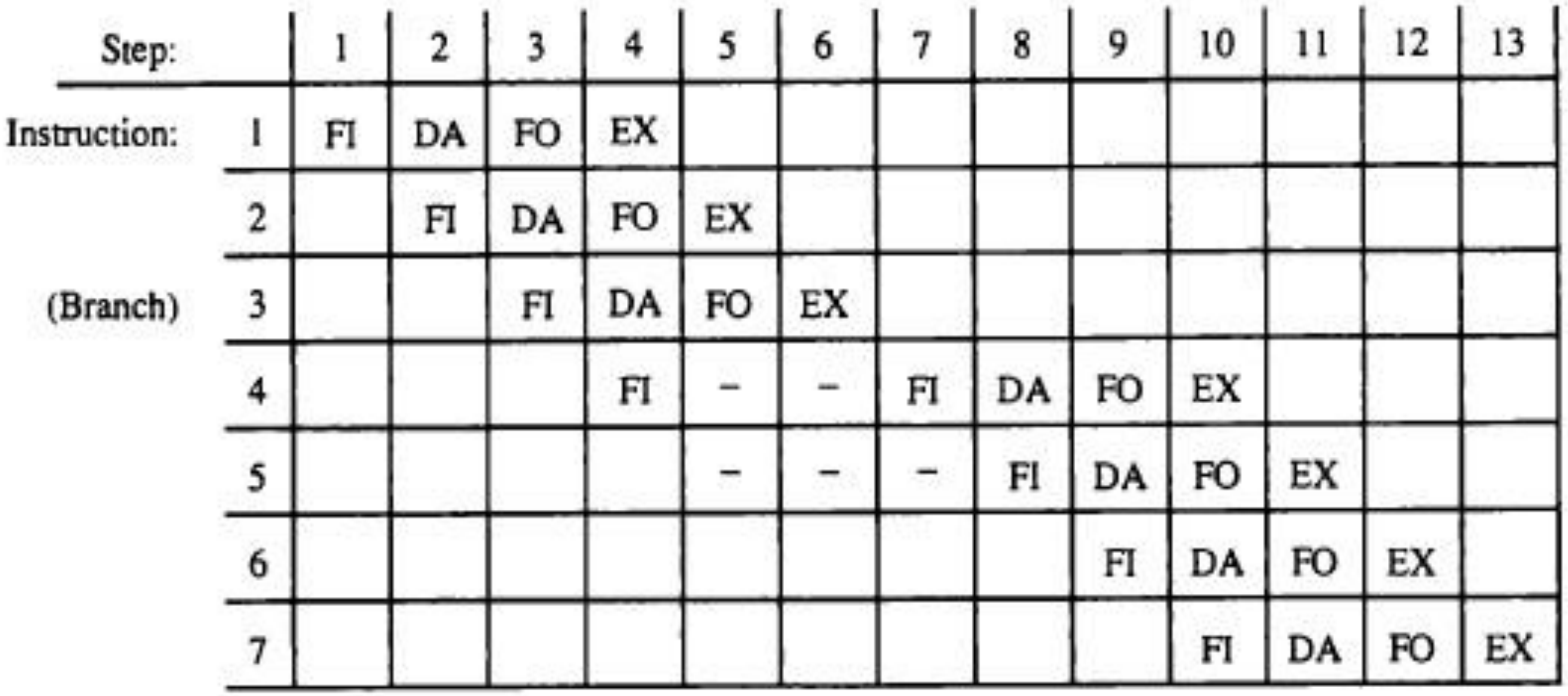


Figure 9-7 Four-segment CPU pipeline.

- Assume that the decoding of the instruction can be contained with the calculation of the effective address into 1 segment and the instruction execution and storing of the result can be combined into 1 segment. Figure shows how the instruction cycle on the CPU can be processed with a four segment pipeline. Therefore, upto four sub operations in the instruction cycle can overlap and upto 4 different instructions can be in progress of being processed at the same time.
- An instruction in the sequence may cause a branch out of normal sequence. In that case, the pending operations in the last two segments are completed and all information stored in the instruction buffer is deleted. The pipeline then restarts from the new address stored in the program counter. Similarly, an interrupt request will cause the pipeline to empty and start again from new address value.



Step:		1	2	3	4	5	6	7	8	9	10	11	12	13
Instruction:	1	FI	DA	FO	EX									
	2		FI	DA	FO	EX								
(Branch)	3			FI	DA	FO	EX							
	4				FI	–	–	FI	DA	FO	EX			
	5					–	–	–	FI	DA	FO	EX		
	6									FI	DA	FO	EX	
	7										FI	DA	FO	EX

Figure 9-8 Timing of instruction pipeline.

- FI is the segment that fetches an instruction.
- DA is the segment that decodes the instruction and calculates the effective address.
- FO is the segment that fetches and operand.
- EX is the segment that executes the segment and stores the value.

Pipeline Hazards and its solution (Important)

In general, there are two major difficulties that cause the instruction pipeline to deviate from its normal operation.

1. **Resource Conflict (Structural Hazard):** It is caused by access to memory by two segments at the same time.
2. **Data Dependency Conflict (Data Hazard):** It arises when an instruction depends on the previous instruction but the result is not yet available.
3. **Branch Difficulties (Control Hazard):** It arises from branch and other instructions that changes the value of program counter.

Solutions:

1. **Resource Conflict Solution :** It can be resolved by using separate instruction and data memory.
2. **Data Dependency Solution:** A collision occurred when an instruction cannot proceed because the previous instructions did not complete certain operations. A data dependency occurs when an instructions needs data and are not yet available. Pipelined computers deal with such conflicts between data dependencies in a variety of ways.

a. **Hardware Interlocks:** An interlock is a circuit that detects instructions whose source operands are destinations of instructions farther up in the pipeline. This approach maintains the program sequence by using hardware to insert the required delays.

b. **Operand Forwarding:** It uses a special hardware to detect a conflict and then avoid it by routing the data through special paths between pipeline segments. This method requires additional hardware paths through multiplexer as well as the circuit that detects the conflict.

c. **Delayed Load:** The compiler for such computers is designed o detect a data conflict and reorder the instructions necessary to delay the loading of the conflicting data by inserting no-operation instructions.

3. Handling of branch instructions: One of the major problem in operating an instruction pipeline is the occurrence of branch instruction. Pipelined computers employ various hardware techniques to minimize the performance degradation caused by instructions branching.

a. **Prefetch Target Instructions:** Both the branch target instruction & the instruction following the branch are pre fetched and are saved until the branch instruction is executed. If branching occurs then branch target instruction is continuous.

b. **Branch target buffer (BTB):** A branch target buffer is an associative memory included in fetch segment of the pipeline. Each entry in the BTB consists of the address of previously executed branch instructions and the target instruction of that branch. It also stores the next few instructions after the branch target instruction. This way, the branch instructions that have occurred previously are readily available in the pipeline without interruption.

c. **Loop buffer:** This is a small very high speed register file maintained by the instruction fetch segment of the pipeline. When a program loop is detected, it is stored in the loop buffer including all the branches. The program loop can be executed directly without having access to memory.

d. **Branch prediction:** A pipeline with branch prediction uses some additional logic to guess the outcome of a conditional branch instruction before it is executed. The pipeline then begins prefetching the instruction stream from the predicted path.

e. **Delayed Branch:** In this procedure, Compiler detects the branch instructions, and re-arranges the machine language code by inserting useful instructions that keep the pipeline operating without interruption. It is used in RISC processor. Example: No operation instruction.

RISC Pipeline

- To use an efficient instruction pipeline

  o To implement an instruction pipeline using a small number of suboperations, with each being executed in one clock cycle.

  o Because of the fixed-length instruction format, the decoding of the operation can occur at the same time as the register selection.

  o Therefore, the instruction pipeline can be implemented with two or three segments.

  - One segment fetches the instruction from program memory
  - The other segment executes the instruction in the ALU
  - Third segment may be used to store the result of the ALU operation in a destination register

- The data transfer instructions in RISC are limited to load and store instructions.

  o These instructions use register indirect addressing. They usually need three or four stages in the pipeline.

  o To prevent conflicts between a memory access to fetch an instruction and to load or store an operand, most RISC machines use two separate buses with two memories.

  o Cache memory: operate at the same speed as the CPU clock

- One of the major advantages of RISC is its ability to execute instructions at the rate of one per clock cycle.

  o In effect, it is to start each instruction with each clock cycle and to pipeline the processor to achieve the goal of single-cycle instruction execution.

  o RISC can achieve pipeline segments, requiring just one clock cycle.

- Compiler supported that translates the high-level language program into machine language program.
    - o Instead of designing hardware to handle the difficulties associated with data conflicts and branch penalties.
    - o RISC processors rely on the efficiency of the compiler to detect and minimize the delays encountered with these problems.

Example: Three-Segment Instruction Pipeline

- There are three types of instructions:
  - o The data manipulation instructions: operate on data in processor registers
  - o The data transfer instructions
  - o The program control instructions
- The control section fetches the instruction from program memory into an instruction register.

  o The instruction is decoded at the same time that the registers needed for the execution of the instruction are selected.
- The processor unit consists of a number of registers and an arithmetic logic unit (ALU).
- A data memory is used to load or store the data from a selected register in the register file.

- The instruction cycle can be divided into three suboperations and implemented in three segments:
  - o I: Instruction fetch
    - - Fetches the instruction from program memory
  - o A: ALU operation
    - - The instruction is decoded and an ALU operation is performed.
    - - It performs an operation for a data manipulation instruction.
    - - It evaluates the effective address for a load or store instruction.
    - - It calculates the branch address for a program control instruction.
  - o E: Execute instruction
    - - Directs the output of the ALU to one of three destinations, depending on the decoded instruction.
    - - It transfers the result of the ALU operation into a destination register in the register file.
    - -It transfers the effective address to a data memory for loading or storing.
    - - It transfers the branch address to the program counter.

Delayed Load:

Consider the operation of the following four instructions:

o LOAD: $R1 \longleftarrow M[address\ 1]$

o LOAD: $R2 \longleftarrow M[address\ 2]$

o ADD: $R3 \longleftarrow R1 + R2$

o STORE: $M[address\ 3] \longleftarrow R3$

There will be a data conflict in instruction 3 because the operand in R2 is not yet available in the A segment. This can be seen from the timing of the pipeline shown in Fig. 4-9(a).

The E segment in clock cycle 4 is in a process of placing the memory data into R2. The A segment in clock cycle 4 is using the data from R2. It is up to the compiler to make sure that the instruction following the load instruction uses the data fetched from memory. This concept of delaying the use of the data loaded from memory is referred to as delayed load

Clock cycles:	1	2	3	4	5	6
1. Load R1	I	A	E			
2. Load R2		I	A	E		
3. Add R1+R2			I	A	E	
4. Store R3				I	A	E

Fig 4-9(a): Three segment pipeline timing - Pipeline timing with data conflict

- Fig. 4-9(b) shows the same program with a no-op instruction inserted after the load to R2 instruction.

Clock cycle:	1	2	3	4	5	6	7
1. Load R1	I	A	E				
2. Load R2		I	A	E			
3. No-operation			I	A	E		
4. Add R1+R2				I	A	E	
5. Store R3					I	A	E

Fig 4-9(b): Three segment pipeline timing - Pipeline timing with delayed load

Thus the no-op instruction is used to advance one clock cycle in order to compensate for the data conflict in the pipeline. The advantage of the delayed load approach is that the data dependency is taken care of by the compiler rather than the hardware.

- Delayed Branch

  - The method used in most RISC processors is to rely on the compiler to redefine the branches so that they take effect at the proper time in the pipeline. This method is referred to as delayed branch.

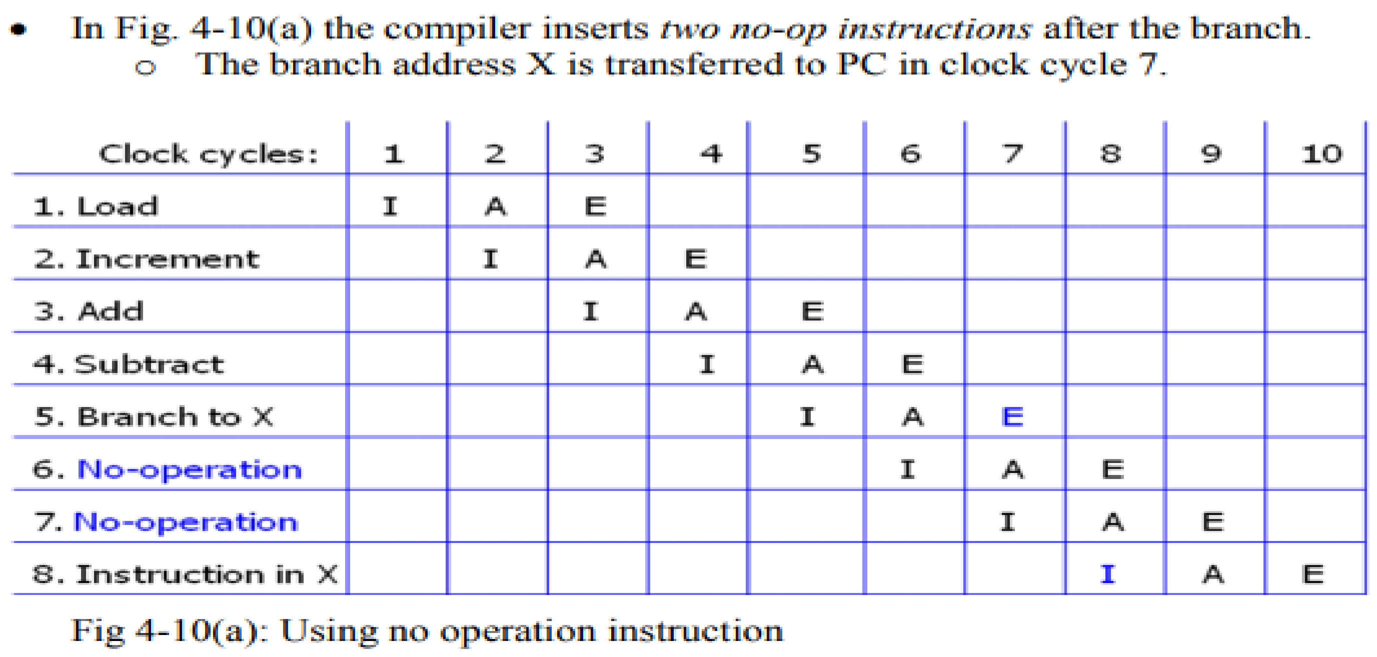
  - The compiler is designed to analyze the instructions before and after the branch and rearrange the program sequence by inserting useful instructions in the delay steps.

  - It is up to the compiler to find useful instructions to put after the branch instruction. Failing that, the compiler can insert no-op instructions.

An Example of Delayed Branch

The program for this example consists of five instructions.

- o Load from memory to R1
- o Increment R2
- o Add R3 to R4
- o Subtract R5 from R6
- o Branch to address X



- In Fig. 4-10(a) the compiler inserts *two no-op instructions* after the branch.
  - o The branch address X is transferred to PC in clock cycle 7.

Clock cycles:	1	2	3	4	5	6	7	8	9	10
1. Load	I	A	E							
2. Increment		I	A	E						
3. Add			I	A	E					
4. Subtract				I	A	E				
5. Branch to X					I	A	E			
6. No-operation						I	A	E		
7. No-operation							I	A	E	
8. Instruction in X								I	A	E

Fig 4-10(a): Using no operation instruction

- The program in Fig. 4-10(b) is rearranged by placing the add and subtract instructions *after the branch instruction*.
  - PC is updated to the value of X in clock cycle 5.

Clock cycles:	1	2	3	4	5	6	7	8
1. Load	I	A	E					
2. Increment		I	A	E				
3. Branch to X			I	A	E			
4. Add				I	A	E		
5. Subtract					I	A	E	
6. Instruction in X						I	A	E

Fig 4-10(b): Rearranging the instructions

6.4 Vector Processing:

There is a class of computational problems that are beyond the capabilities of a conventional computer. These problems characterized by the fact that they require a vast number of computations that will take a conventional computer days or even weeks to complete. In many science and engineering applications, the problems can be formulated in terms of vectors and matrices that lend themselves to vector processing.

Computers with vector processing capabilities are in demand in specialized applications. The following are representative application areas where vector processing is of the utmost importance.

1. Long-range weather forecasting
2. Petroleum explorations
3. Seismic data analysis
4. Medical diagnosis
5. Aerodynamics and space flight simulations
6. Artificial intelligence and expert systems
7. Mapping the human genome
8. Image processing

To achieve the required level of high performance, it is necessary to utilize the fastest and most reliable hardware and apply innovative procedures from vector and parallel processing techniques.

Vector Operation:

A vector is an order set of one dimensional array of data items. A vector V of length ‘n’ is represented as a row vector by V = [V_1, V_2, V_3,……………………, V_n]

Operation on vector must be broken down into single computation with subscripted variables. The element V_1 of the vector V is written as V (I) and the index ‘I’ refers to a memory address or register where the numbers are stored.

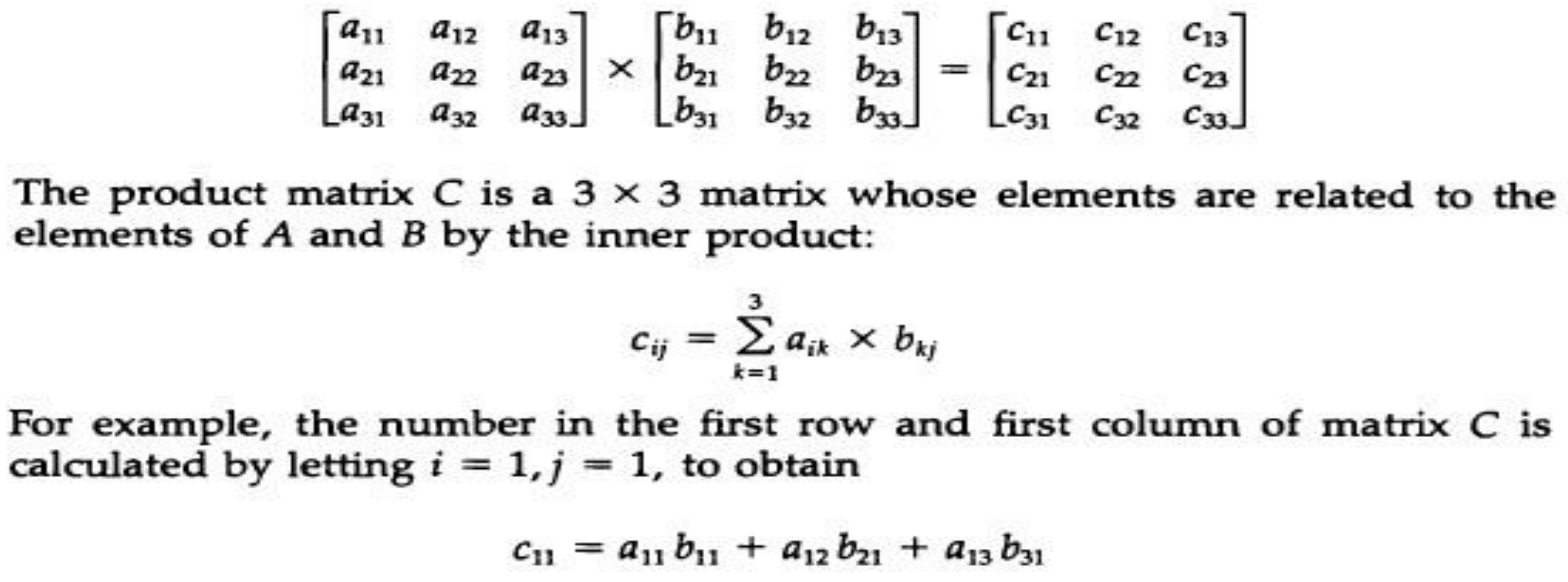
For example program for adding two vectors A and B of length 100 to produce a vector C is represented as:

C(1:100)=A(1:100)+B(1:100)

This vector instruction includes the initial address of the operand, the length of the vector and the operation to be performed in one composite instruction.

Matrix Multiplication:

The multiplication of two n*n matrices consists of n^2 inner products or n^3 multiply-add operations. The multiplication of two 3*3 matrices is represented as:



$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix}$$

The product matrix C is a 3×3 matrix whose elements are related to the elements of A and B by the inner product:

$$c_{ij} = \sum_{k=1}^{3} a_{ik} \times b_{kj}$$

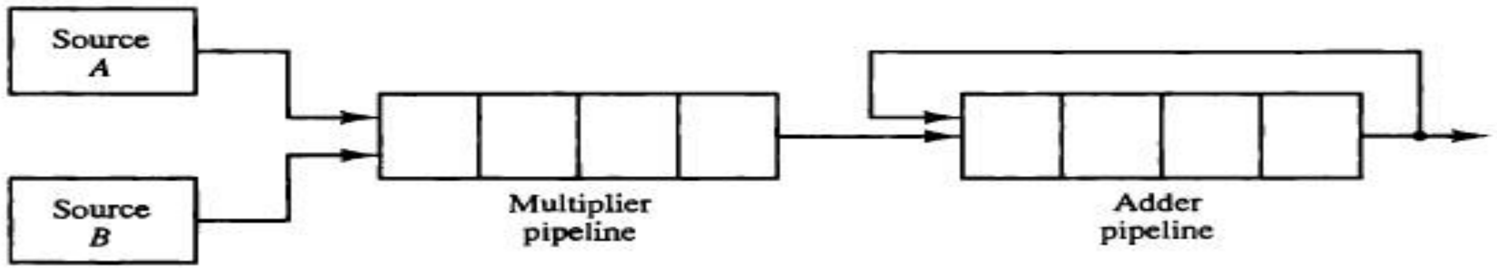
For example, the number in the first row and first column of matrix C is calculated by letting $i = 1, j = 1$, to obtain

$$c_{11} = a_{11} b_{11} + a_{12} b_{21} + a_{13} b_{31}$$

The total no. of multiplication or condition required to complete the matrix product is 27 (9*3).
The inner product consists of the sum of k products in terms of

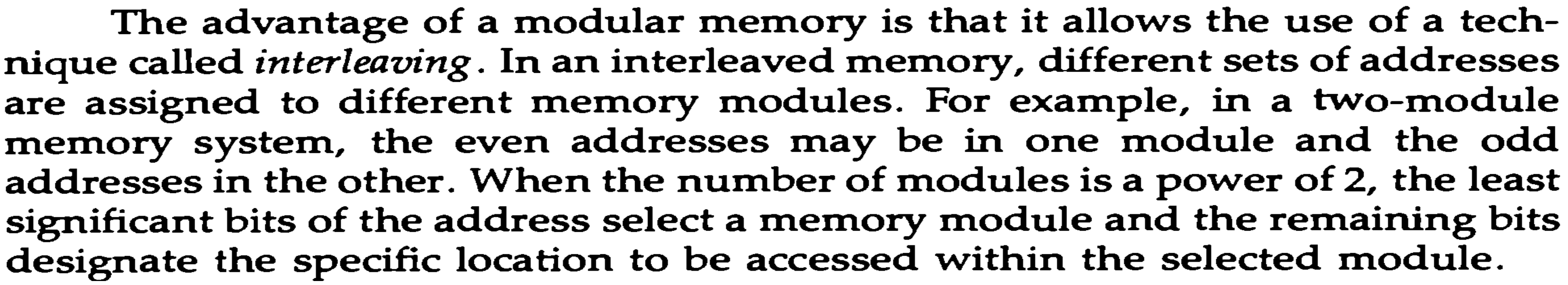
$$\begin{aligned} C = {} & A_1 B_1 + A_5 B_5 + A_9 B_9 + A_{13} B_{13} + \cdots \\ & + A_2 B_2 + A_6 B_6 + A_{10} B_{10} + A_{14} B_{14} + \cdots \\ & + A_3 B_3 + A_7 B_7 + A_{11} B_{11} + A_{15} B_{15} + \cdots \\ & + A_4 B_4 + A_8 B_8 + A_{12} B_{12} + A_{16} B_{16} + \cdots \end{aligned}$$

Figure 9-12 Pipeline for calculating an inner product.



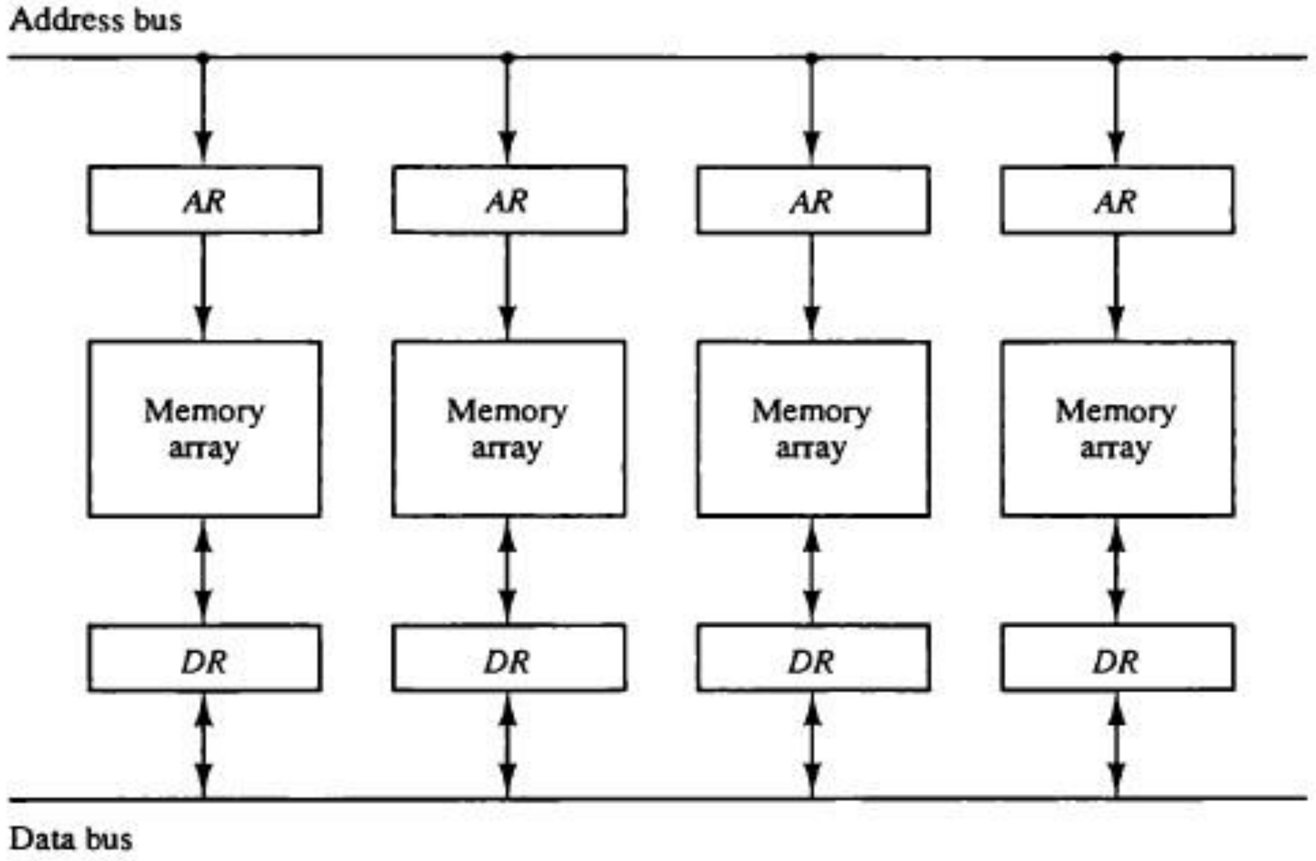
Memory Interleaving

Pipeline and vector processor often require simultaneous access to memory from two or more sources. An instruction pipeline may require the fetching of an instruction and operand at the same time from two different segments. An arithmetic pipeline usually requires two or more operands to enter the pipeline at the same time. Instead of using two memory buses for simultaneous access, the memory can be partitioned into a number of modules connected to a common memory address and data buses. A memory module is a memory array together with its own array and data registers.



The advantage of a modular memory is that it allows the use of a technique called *interleaving*. In an interleaved memory, different sets of addresses are assigned to different memory modules. For example, in a two-module memory system, the even addresses may be in one module and the odd addresses in the other. When the number of modules is a power of 2, the least significant bits of the address select a memory module and the remaining bits designate the specific location to be accessed within the selected module.

Figure 9-13 Multiple module memory organization.



Super Computers

A commercial computers with vector instructions and pipeline floating point arithmetic operations is referred to as a super computer. It is a computer system best known for its high computational speed, fast and large memory system and the extensive use of parallel processing. They are limited in their use to a number of scientific application such as weather forecasting and space research. Eg: Cray-1: it uses vector processing with 12 distinct functional units in parallel; a large number of registers (over 150); multiprocessor configuration (Cray X-MP and Cray Y-MP)

Array Processing:

An array processor is a processor that performs computation on large array of data. An attached array processor is an auxiliary processor attached to a general purpose computer. SIMD array processor is a processor that has a single instruction multiple data organization. It manipulates vector instruction by means of multiple functional unit responding through a common instructions.

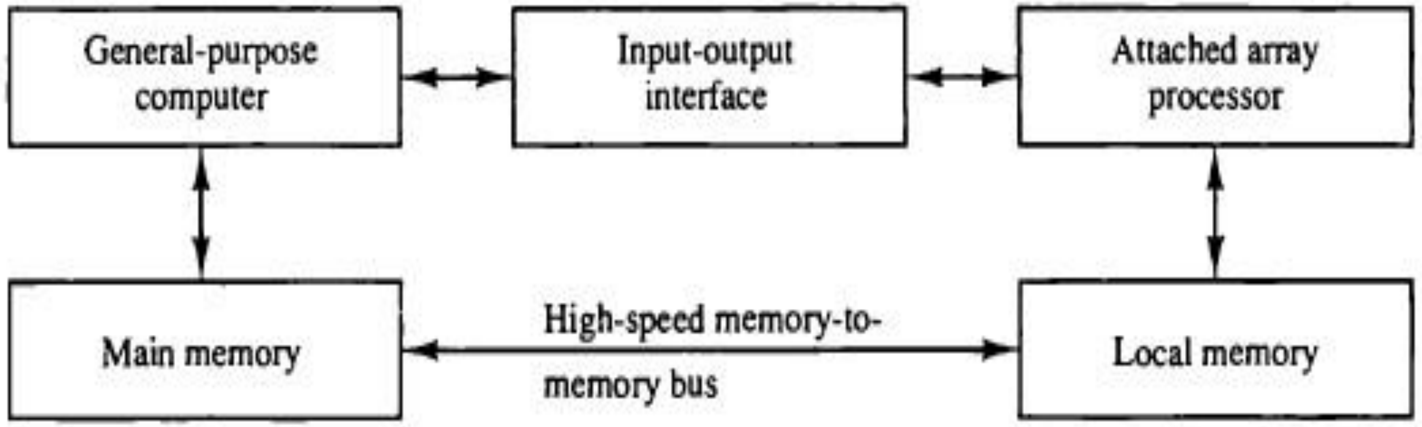
1. **Attached Array Processor:**

It is designed as a peripheral of conventional host computer and its purpose is to enhance the performance of the computer by providing vector processing for complex scientific application. It achieves high performance by means of parallel processing with multiple functional units.

Figure shows the interconnection of an attached array processor to a host computer. The host computer is a general purpose computer and the attached processor is a backend machine driven by the host computer. The array processor is connected through an input-output controller to the computer and the computer treats like an external interface.

The data for the attached processor are transferred from a main memory to a local memory through a high speed bus. The general purpose computer without the attached processor serves the user that needs conventional data processing and the system with attached processor satisfies the need for complex arithmetic application. The objective of the attached array processor is to provide vector manipulation capabilities to a conventional computer.

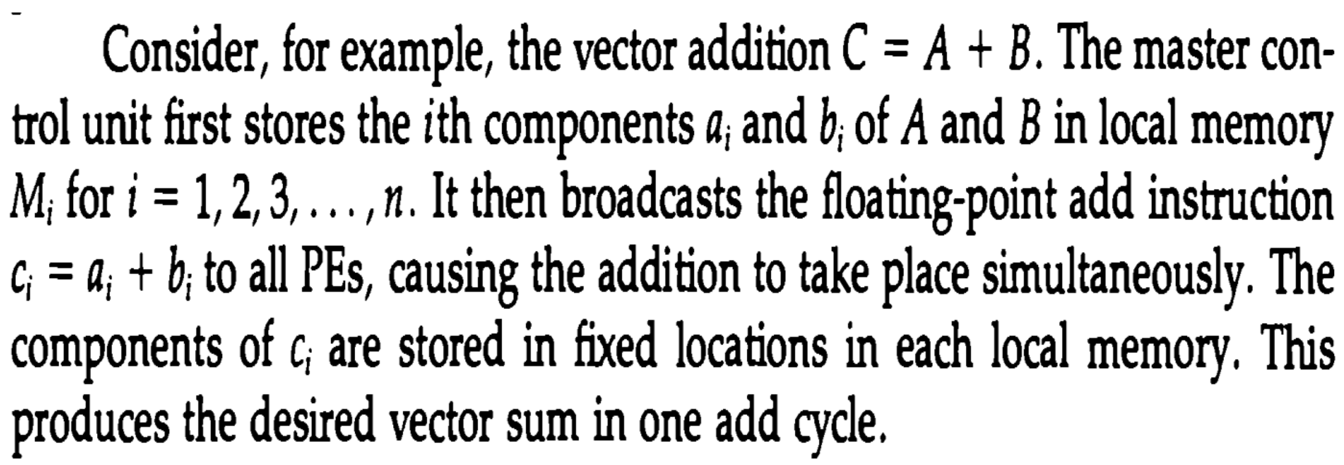
Figure 9-14 Attached array processor with host computer.



2. **SIMD Array Processor:**

It is a computer with multiple processing unit operation in parallel. The processing unit are synchronized to perform the same operations under the control of a common bus unit, providing SIMD organisation.

It contains a set of identical processing elements (PE). Each having a local memory M. Each processing elements includes an ALU, a floating point arithmetic unit and working registers. The master control unit controls the operation in the processor elements. The main memory is used for the storage of the program. Vector instructions are broadcasted to all PE's simultaneously.



Consider, for example, the vector addition $C = A + B$. The master control unit first stores the ith components a_i and b_i of A and B in local memory M_i for $i = 1, 2, 3, \ldots, n$. It then broadcasts the floating-point add instruction $c_i = a_i + b_i$ to all PEs, causing the addition to take place simultaneously. The components of c_i are stored in fixed locations in each local memory. This produces the desired vector sum in one add cycle.

Each PE has a flag that us set when the PE is active and reset when the PE is inactive. This ensures that only those PE that need to participate are active during the execution of the instruction. Eg: ILLIAC IV

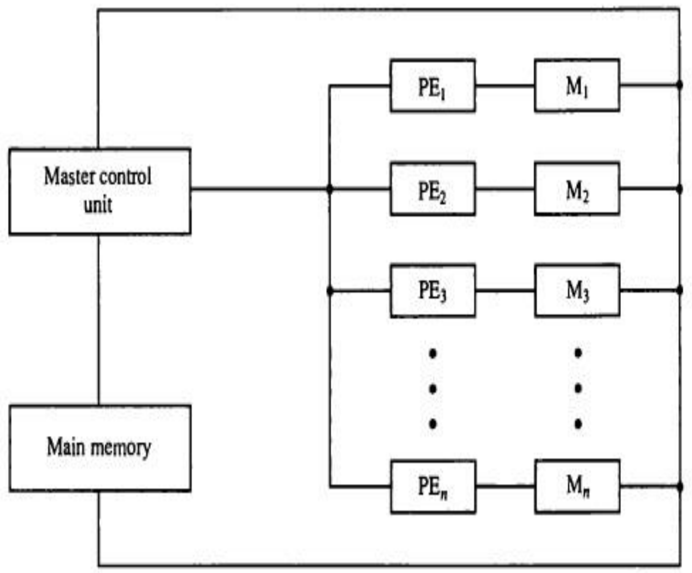


Figure 9-15 SIMD array processor organization.