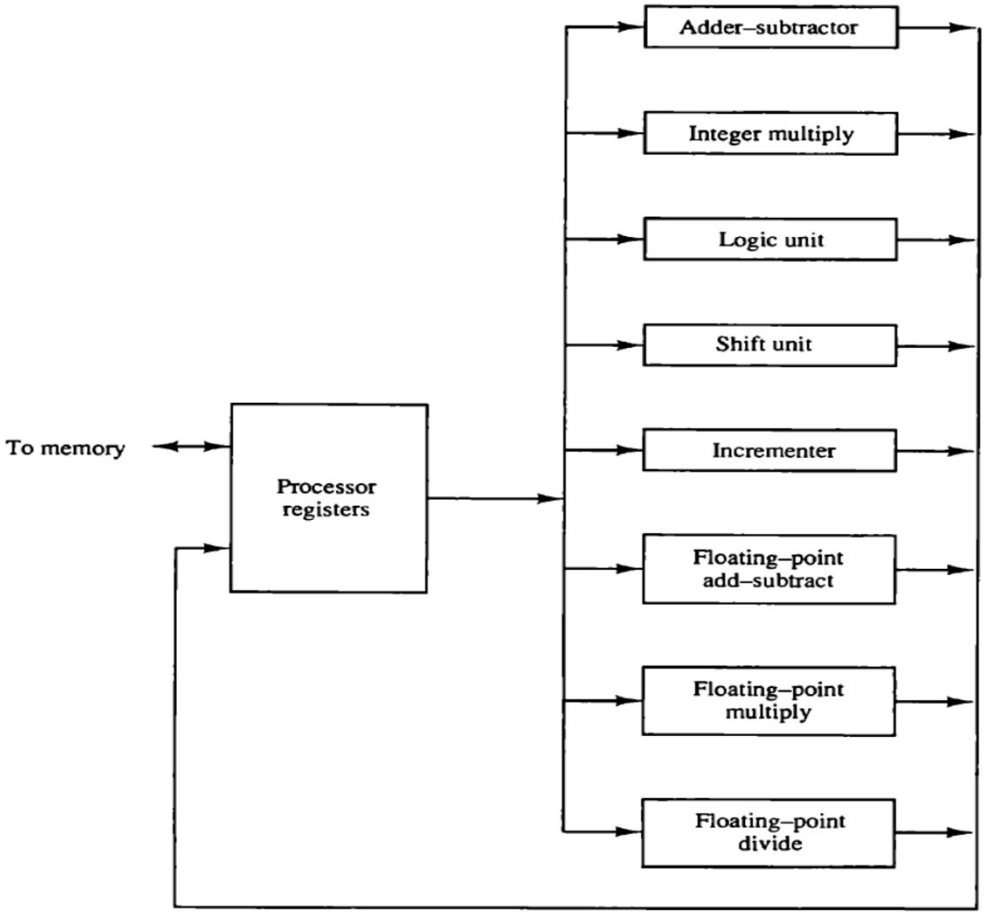
Unit 6: :Pipeline and Vector Processing

Parallel Processing

- Term used to support simultaneous data processing task to increase the computation speed of the processor
- Perform *concurrent* data processing to achieve faster execution time
- Computer can have more than one processing units for parallel processing
- Multiple Functional Unit :
    - ***Separate the execution unit into eight functional units operating in parallel***

Figure 9-1 Processor with multiple functional units.



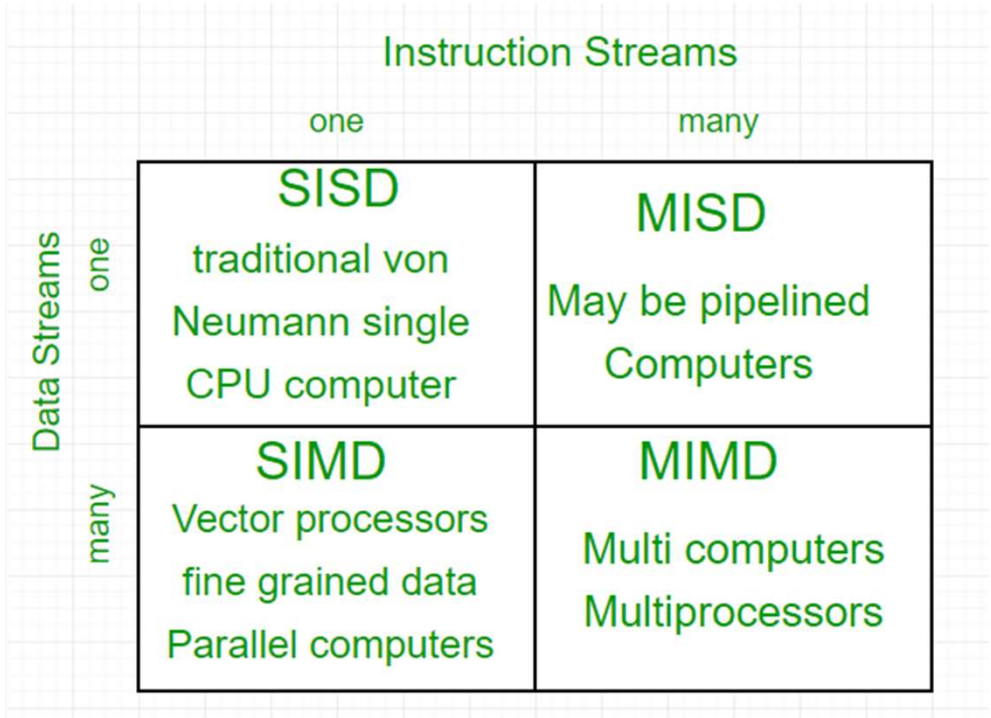
- Divide execution unit in 8 functional units operating in parallel
- Operands, in the registers are applied to the one of the units based on operation specified by that unit
- All the units are separate from each other so one number can be shifted when other is being incremented
- There are variety of ways to classify parallel processing based on internal organization of processors, interconnection structure between processor or flow of information through system

Flynn's Taxonomy

- One of the classification is derived by M.J.Flynn that consider number of instructions and data items that are manipulated simultaneously
- Normal operation of computer: fetch the instruction and execute the instruction
- The sequence of instruction read from the memory is called **instruction stream**
- The operation performed on the data in processor is called **data stream**
- Parallel processing may occur in instruction stream, in data stream or in both

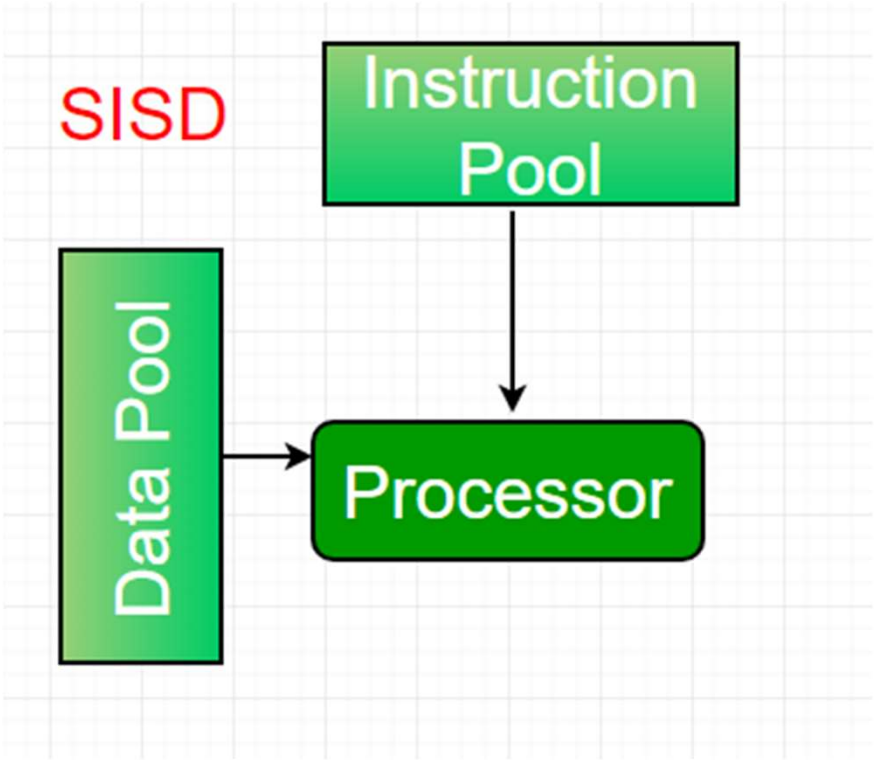
- Flynn's classification divides computers into four major groups that are:

1. **Single instruction stream, single data stream (SISD)**
2. **Single instruction stream, multiple data stream (SIMD)**
3. **Multiple instruction stream, single data stream (MISD)**
4. **Multiple instruction stream, multiple data stream (MIMD)**


Instruction Streams
one
many
Data Streams
one
many
SISD
traditional von Neumann single CPU computer
MISD
May be pipelined Computers
SIMD
Vector processors
fine grained data
Parallel computers
MIMD
Multi computers
Multiprocessors

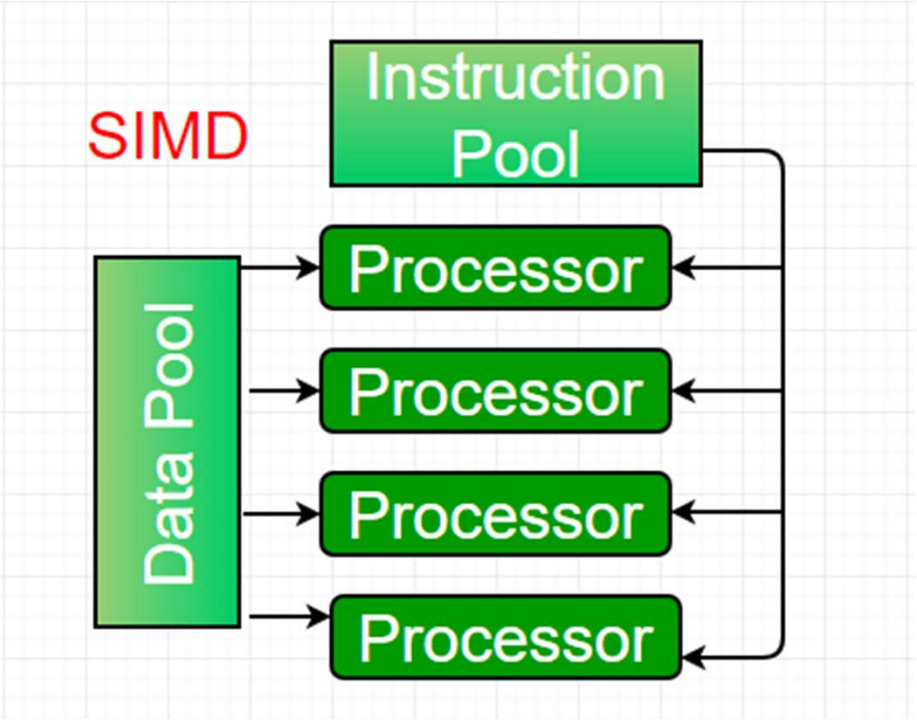
Single-instruction, single-data (SISD) systems –

- An SISD computing system is a uniprocessor machine which is capable of executing a single instruction, operating on a single data stream.
- In SISD, machine instructions are processed in a sequential manner and computers adopting this model are popularly called sequential computers.
- Most conventional computers have SISD architecture.
- All the instructions and data to be processed have to be stored in primary memory.


SISD
Instruction Pool
Data Pool
Processor

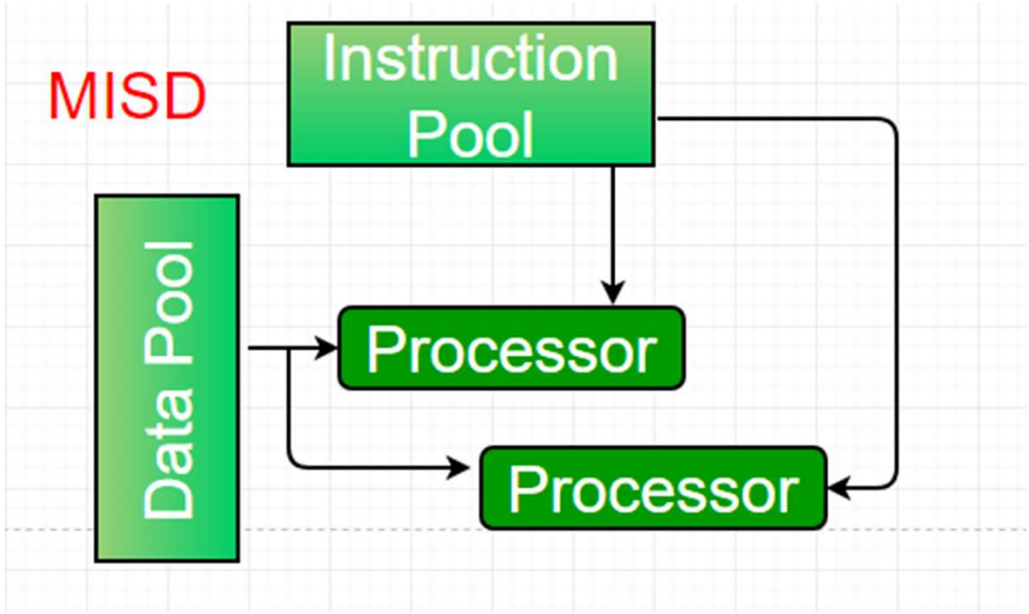
Single-instruction, multiple-data (SIMD) systems –

- An SIMD system is a multiprocessor machine capable of executing the same instruction on all the CPUs but operating on different data streams.
- Machines based on an SIMD model are well suited to scientific computing since they involve lots of vector and matrix operations.
- So that the information can be passed to all the processing elements (PEs) organized data elements of vectors can be divided into multiple sets(N-sets for N PE systems) and each PE can process one data set.
- Dominant representative SIMD systems is Cray's vector processing machine.


SIMD
Instruction Pool
Data Pool
Processor
Processor
Processor
Processor

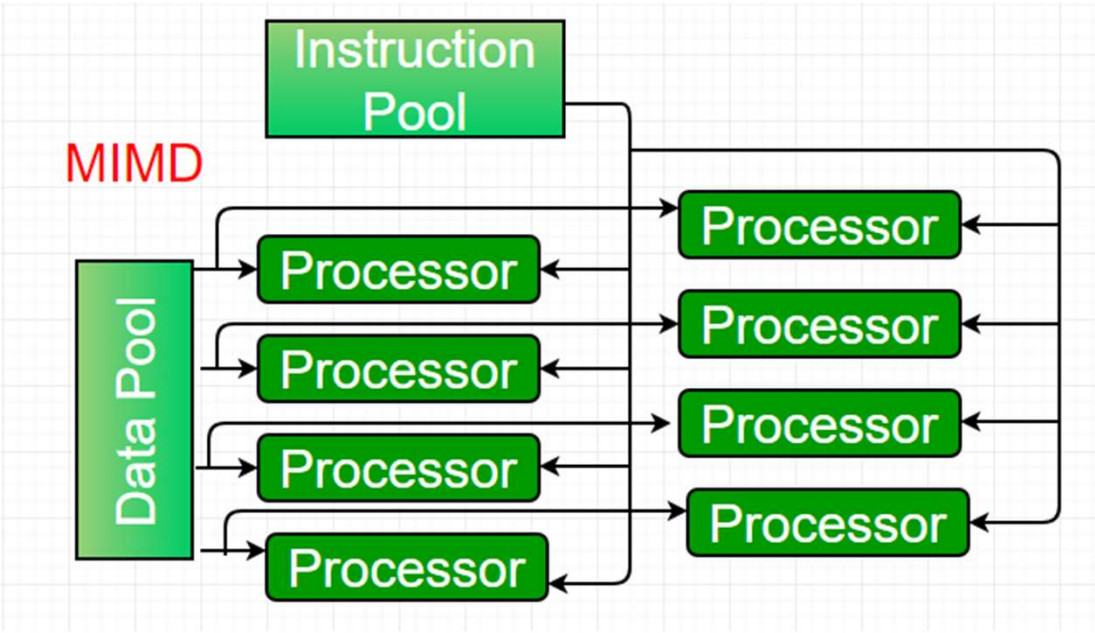
Multiple-instruction, single-data (MISD) systems –

- An MISD computing system is a multiprocessor machine capable of executing different instructions on different PEs but all of them operating on the same dataset .
- Example Z = sin(x)+cos(x)+tan(x)
- The system performs different operations on the same data set.
- Machines built using the MISD model are not useful in most of the application, a few machines are built, but none of them are available commercially.


MISD
Instruction Pool
Data Pool
Processor
Processor

Multiple-instruction, multiple-data (MIMD) systems –

- An MIMD system is a multiprocessor machine which is capable of executing multiple instructions on multiple data sets.
- Each PE in the MIMD model has separate instruction and data streams; therefore machines built using this model are capable to any kind of application.
- Unlike SIMD and MISD machines, PEs in MIMD machines work asynchronously.
- MIMD machines are broadly categorized into **shared-memory MIMD** and **distributed-memory MIMD** based on the way PEs are coupled to the main memory.


Instruction Pool
MIMD
Data Pool
Processor
Processor
Processor
Processor
Processor
Processor
Processor
Processor

Pipelining

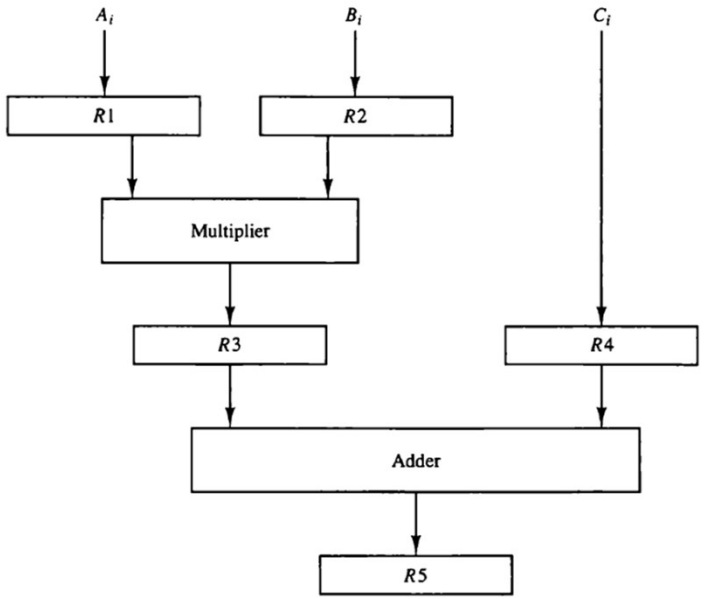
- **Pipelining is a technique** of decomposing a sequential process into suboperations, with each subprocess being executed in a special dedicated segment that operates concurrently with all other segments.
- **A pipeline** can be visualized as a collection of processing segments through which binary information flows. Each segment performs partial processing dictated by the way the task is 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.
- **The name** "pipeline" implies a flow of information analogous to an industrial assembly line.
- **It is characteristic** of pipelines that several computations can be in progress in distinct segments at the same time. The overlapping of computation is made possible by associating a register with each segment in the pipeline.
- **The registers** provide isolation between each segment so that each can operate on distinct data simultaneously.

- **Perhaps the simplest way** of viewing the pipeline structure is to imagine that each segment consists of an input register followed by a combinational circuit.
- The register holds the data and the combinational circuit performs the suboperation in the particular segment.
- **The output** of the combinational circuit in a given segment is applied to the input register of the next segment.
- A clock is applied to all registers after enough time has elapsed to perform all segment activity. In this way the information flows through the pipeline one step at a time.
- **The pipeline organization** will be demonstrated by means of a simple example.
- Suppose that we want to perform the combined multiply and add operations with a stream of numbers. $A_i * B_i + C_i$ for i = 1, 2, 3, . .. , 7

- **Each suboperation** is to be implemented in a segment within a pipeline.
- Each segment has one or two registers and a combinational circuit. R1 through R5 are registers that receive new data with every clock pulse. The multiplier and adder are combinational circuits.
- The suboperations performed in each segment of the pipeline are as follows:
- **Segment 1:- R1** $\leftarrow A_i$, R2 $\leftarrow B_i$ Input A_i and B_i
- **Segment 2:-** R3 $\leftarrow$ R1 * R2, R4 $\leftarrow C_i$ Multiply and input C_i
- **Segment 3:-** R5 $\leftarrow$ R3 + R4 Add C_i to product

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

Figure 9-2 Example of pipeline processing.



- **The five registers** are loaded with new data every clock pulse. The effect of each clock is shown in Table 1.
- The first clock pulse transfers A1 and B1 into R1 and R2.
- **The second clock pulse** transfers the product of R1 and R2 into R3 and C1 into R4. The same clock pulse transfers A2 and B2 into R1 and R2.
- The third clock pulse operates on all three segments simultaneously.
- **It places A, and B,** into R1 and R2, transfers the product of R1 and R2 into R3, transfers C, into R4, and places the sum of R3 and R4 into RS. It takes three clock pulses to fill up the pipe and retrieve the first output from RS. From there on, each dock produces a new output and moves the data one step down the pipeline.
- **This happens as long** as new input data flow into the system. When no more input data are available, the clock must continue until the last output emerges out of the pipeline

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	—	—	—	—	$A_7 * B_7 + C_7$

Mathematical Notations for Pipelining

- **The general structure** of a four-segment pipeline is illustrated . The operands pass through all four segments in a fixed sequence.
- **Each segment** consists of a combinational circuit S_i that performs a suboperation over the data stream flowing through the pipe.
- **The segments** are separated by registers R_i that hold the intermediate results between the stages.
- **Information flows** between adjacent stages under the control of a common clock applied to all the registers simultaneously.
- **We task define a task** as the total operation performed going through all the segments in the pipeline.
- **The behaviour of a pipeline** can be illustrated with a space-time diagram.
- This is a diagram that shows the segment utilization as a function of time. The space-time diagram of a four-segment pipeline is demonstrated . The horizontal axis displays the time in clock cycles and the vertical axis gives the segment number.

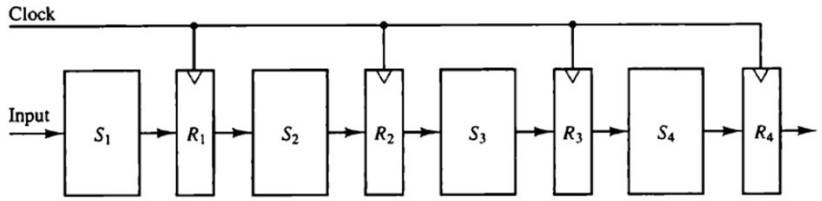


Figure 9-3 Four-segment pipeline.

Figure 9-4 Space-time diagram for pipeline.

Segment:	1	2	3	4	5	6	7	8	9
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

Clock cycles

- We have k segment pipeline , with clock cycle time tp to execute n tasks
- First task T1 requires time equal to **ktp** to complete its operation since there are k segments in pipeline
- Remaining (n-1) tasks emerges from pipeline at rate of one task per clock cycle and they will be completed after time **(n-1)tp**
- so total time required for pipeline ing process is

  ktp+(n-1)tp = (k+(n-1))*tp
- Non pipelining time will be tn time for n tasks so n*tn
- We can find speed up ration by nonpipelined time/pipeline time

- As the number of task increases n>(k-1), so k+n-1 will move forward toward n
- Speed up ratio = ntn/ntp= tn/tp
- Now assume time to process the task is same in pipeline and non-pipeline process than tn = ktp
- Speed up ratio = tn/tp = ktp/tp = k
- So, speed up ration = number of segments

- **Example:-**
- **K =4, tp = 20 ns, n= 100**
- **Calculate speedup ratio**
- Speedup ratio = ntn/(k+(n-1))*tp
- **For non pipeline :- n*tn**
- tn = k*tp = 4*20 = 80 ns
- n*tn = 100*80 = 8000 ns
- **For pipeline := (k+n-1)*tp**
- = (4+100-1)*20 = 2060
- Speed up ratio = 8000/2060 = 3.88

◆ Speedup S : Nonpipeline / Pipeline

- **$S = n \cdot t_n / (k + n - 1) \cdot t_p = 6 \cdot 6\, t_n / (4 + 6 - 1) \cdot t_p = 36\, t_n / 9\, t_n = 4$**
  - » n : *task number* (6)
  - » t_n : *time to complete each task in nonpipeline* (6 cycle times = 6 t_p)
  - » t_p : *clock cycle time* (1 clock cycle)
  - » k : *segment number* (4)

$k + n - 1 \approx n$

- If $n \rightarrow \infty$ 이면, $S = t_n / t_p$
- 한 개의 task를 처리하는 시간이 같을 때 즉, ***nonpipeline*** (t_n) **=** ***pipeline*** ($k \cdot t_p$) 이라고 가정하면,

  $S = t_n / t_p = k \cdot t_p / t_p = k$

  따라서 이론적으로 **k** 배 (***segment*** *개수*) 만큼 처리 속도가 향상된다.

Pipeline에서의 처리 시간 = 9 clock cycles

- **This shows that the theoretical** maximum speedup that a pipeline can provide is k, where k is the number of segments in the pipeline.
- **To clarify the meaning** of the speedup ratio, consider the following numerical example.
- Let the time it takes to process a suboperation in each segment be equal to $t_p = 20$ ns.
- **Assume that the pipeline** has k = 4 segments and executes n = 100 tasks in sequence.
- The pipeline system will take $(k + (n - 1))t_p = (4 + 99) \times 20 = 2060$ ns to complete. Assuming that $t_n = kt_p = 4 \times 20 = 80$ ns, a non pipeline system requires $nkt_p = 100 \times 80 = 8000$ ns to complete the 100 tasks.
- **The speedup** ratio is equal to 8000/2060 = 3.88. As the number of tasks increases, the speedup will approach 4, which is equal to the number of segments in the pipeline. If we assume that $t_n = 60$ ns, the speedup becomes 60/20 = 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.**
- **A pipeline multiplier** is essentially an array multiplier, with special adders designed to minimize the carry propagation time through the partial products.
- **Floating-point** operations are easily decomposed into suboperations.
- **The inputs** to the floating-point adder pipeline are two normalized floating-point binary numbers.
- **$X = A * 2^a$**
- **$Y = B * 2^b$**
- **(A,B – mantissa , a,b:-exponent)**

- **$X = A * 2^a = 0.9504 * 10^3$**
- **$Y = B * 2^b = 0.8200 * 10^2$**
- **We will first compare both the exponents and choose the higher one (in this case 3) so we need to arrange the mantissa so bboth exponents will be equal**
- **$X = A * 2^a = 0.9504 * 10^3$**
- **$Y = B * 2^b = 0.0820 * 10^3$**
- Add the two mantissa so result will be $1.0324 * 10^3$ (overflow condition)

•**A and B are two fractions** that represent the mantissas and a and b are the exponents.

• 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.

•**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.

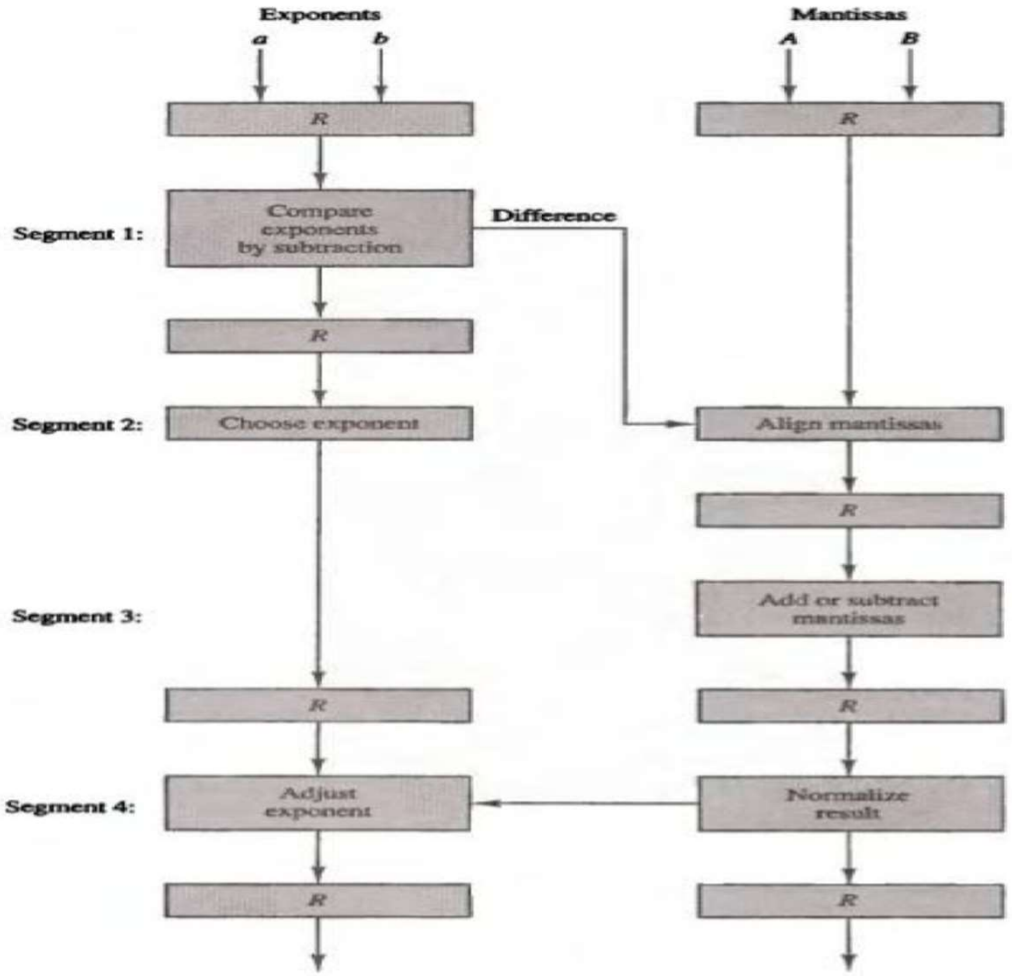


Figure 9-6 Pipeline for floating-point addition and subtraction.

•**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.

•As in our example result is $1.0324*10^3$ here most significant bit is 1(before .) which can not be stored so we need to adjust the exponent by incrementing exponent by shifting right and make it $0.1032 *10^4$

•**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

• If after subtraction most significant bit is 0 (after .) like $0.0112*10^3$ than we have to decrement exponent by shifting left so $0.1120*10^2$

Instruction Pipeline

- **Pipeline processing** can occur not only 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.
- **One possible** digression associated with such a scheme is that an instruction may cause a branch out of sequence.
- In that case the pipeline must be emptied and all the instructions that have been read from memory after the branch instruction must be discarded.

- **Consider a computer** with an instruction fetch unit and an instruction execution unit designed to provide a two-segment pipeline.
- The instruction fetch segment can be implemented by means of a first-in, first-out (FIFO) buffer. This is a type of unit that forms a queue rather than a stack.
- **Whenever the execution unit** is not using memory, the control increments the program counter and uses its address value to read consecutive instructions from memory.
- The instructions are inserted into the FIFO buffer so that they can be executed on a first-in, first-out basis.
- **Thus an instruction stream** can be placed in a queue, waiting for decoding and processing by the execution segment.

- The instruction stream queuing mechanism provides an efficient way for reducing the average access time to memory for reading instructions.
- **Whenever there is space in the FIFO buffer**, the control unit initiates the next instruction fetch phase.
- **The buffer acts** as a queue from which control then extracts the instructions for the execution unit.
- **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.
- 1. Fetch the instruction from memory.
- 2. Decode the instruction.
- 3. Calculate the effective address.
- 4. Fetch the operands from memory.
- 5. Execute the instruction.
- 6. 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 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.
- **Memory access conflicts** are sometimes resolved by using two memory buses for accessing instructions and data in separate modules. ln this way, an instruction word and a data word can be read simultaneously from two different modules.
- **The design of an instruction** pipeline will be most efficient if the instruction cycle is divided into segments of equal duration. The time that each step takes to full fill its function depends on the instruction and the way itis executed.

Four-Segment Instruction Pipeline

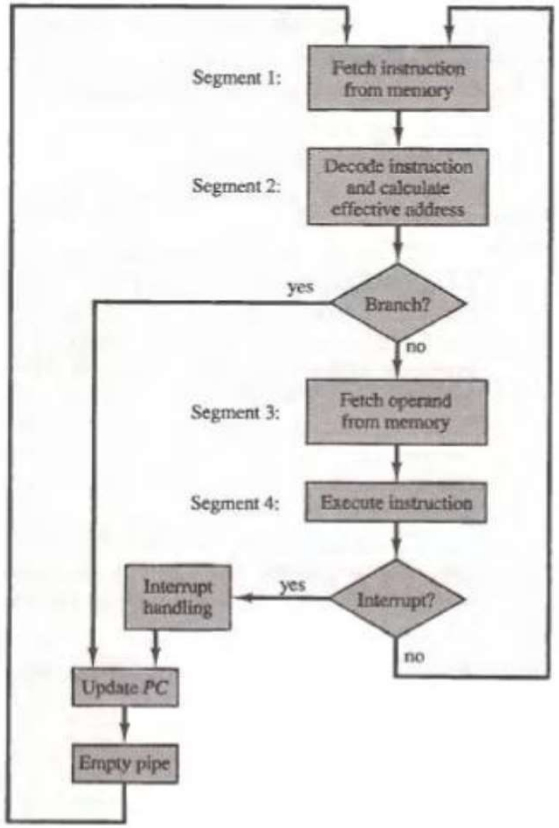


Figure 9-7 Four-segment CPU pipeline.

- **Assume that the decoding** of the instruction can be combined with the calculation of the effective address into one segment.
- **Assume further** that most of the instructions place the result into a processor register so that the instruction execution and storing of the result can be combined into one segment.
- This reduces the instruction pipeline into four segments.
- **Figure shows** how the instruction cycle in the CPU can be processed with a four-segment pipeline.
- While an instruction is being executed in segment 4, the next instruction in sequence is busy fetching an operand from memory in segment 3.

- **The effective address** may be calculated in a separate arithmetic circuit for the third instruction, and whenever the memory is available, the fourth and all subsequent instructions can be fetched and placed in an instruction FIFO.
- **Thus up to four suboperations** in the instruction cycle can overlap and up to four different instructions can be in progress of being processed at the same time.
- **Once in a while**, an instruction in the sequence may be a program control type that causes 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, when acknowledged, will cause the pipeline to empty and start again from a new address value.
- 1. F1 is the segment that fetches an instruction.
- 2. DA is the segment that decodes the instruction and calculates the effective address.
- 3. FO is the segment that fetches the operand.
- 4. EX is the segment that executes the instruction

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.

•**In general**, there are three major difficulties that cause the instruction pipeline to deviate from its normal operation.
•1. **Resource conflicts** caused by access to memory by two segments at the same time. Most of these conflicts can be resolved by using separate instruction and data memories.
• 2. **Data dependency** conflicts arise when an instruction depends on the result of a previous instruction, but this result is not yet available.
• 3. **Branch difficulties** arise from branch and other instructions that change the value of PC.

Data Dependency

- **A difficulty** that may caused a degradation of performance in an instruction pipeline is due to possible collision of data or address.
- **A collision** occurs when an instruction cannot proceed because previous instructions did not complete certain operations.
- **A data dependency** occurs when an instruction needs data that are not yet available.
- **For example**, an instruction in the FO segment may need to fetch an operand that is being generated at the same time by the previous instruction in segment EX.

- **Therefore**, the second instruction must wait for data to become available by the first instruction. Similarly, an address dependency may occur when an operand address cannot be calculated because the information needed by the addressing mode is not available.
- **For example**, an instruction with register indirect mode cannot proceed to fetch the operand if the previous instruction is loading the address into the register.
- **Therefore**, the operand access to memory must be delayed until the required address is available. Pipelined computers deal with such conflicts between data dependencies in a variety of ways.
- **The most straightforward** method is to insert hardware interlocks. An interlock is a circuit that detects instructions whose source operands are destinations of instructions farther up in the pipeline.
- **Detection of this situation** causes the instruction whose source is not available to be delayed by enough clock cycles to resolve the conflict. This approach maintains the program sequence by using hardware to insert the required delays.

- **Another technique** called operand forwarding uses special hardware to detect a conflict and then avoid it by routing the data through special paths between pipeline segments.
- **For example**, instead of transferring an ALU result into a destination register, the hardware checks the destination operand, and if it is needed as a source in the next instruction, it passes the result directly into the ALU input, bypassing the register.
- **This method** requires additional hardware paths through multiplexers as well as the circuit that detects the conflict.
- **A procedure** employed in some computers is to give the responsibility for solving data conflicts problems to the compiler that translates the high-level programming language into a machine language program.
- **The compiler** for such computers is designed to detect a data conflict and reorder the instructions as necessary to delay the loading of the conflicting data by inserting no-operation instructions. This method is referred to as delayed load.

Handling of Branch Instructions

- **One of the major problems** in operating an instruction pipeline is the occurrence of branch instructions.
- **A branch instruction** can be conditional or unconditional. An unconditional branch always alters the sequential program flow by loading the program counter with the target address.
- **In a conditional branch**, the control selects the target instruction if the condition is satisfied or the next sequential instruction if the condition is not satisfied. As mentioned previously, the branch instruction breaks the normal sequence of the instruction stream, causing difficulties in the operation of the instruction pipeline.
- **Pipelined computers** employ various hardware techniques to minimize the performance degradation caused by instruction branching.
- One way of handling a conditional branch is **to prefetch the target instruction** in addition to the instruction following the branch.
- Both are saved until the branch is executed. If the branch condition is successful, the pipeline continues from the branch target instruction.
- An extension of this procedure is to continue fetching instructions from both places until the branch decision is made. At that time control chooses the instruction stream of the correct program flow.

- **Another possibility** is the use of a **branch target buffer or BTB.**
- **The BTB is an associative memory** included in the fetch segment of the pipeline. Each entry in the BTB consists of the address of a previously executed branch instruction and the target instruction for that branch.
- **It also stores** the next few instructions after the branch target instruction. When the pipeline decodes a branch instruction, it searches the associative memory BTB for the address of the instruction.
- **If it is in the BTB**, the instruction is available directly and prefetch continues from the new path.

- **If the instruction** is not in the BTB, the pipeline shifts to a new instruction stream and stores the target instruction in the BTB. The advantage of this scheme is that branch instructions that have occurred previously are readily available in the pipeline without interruption. A variation of the BTB is the 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 in the program, it is stored in the loop buffer in its entirety, including all branches.
- **The program loop** can be executed directly without having to access memory until the loop mode is removed by the final branching out. Another procedure that some computers use is 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. A correct prediction eliminates the wasted time caused by branch penalties. A procedure employed in most ruse processors is the delayed branch .

- **In this procedure**, the compiler detects the branch instructions and rearranges the machine language code sequence by inserting useful instructions that keep the pipeline operating without interruptions.
- **An example** of delayed branch is the insertion of a no-operation instruction after a branch instruction. This causes the computer to fetch the target instruction during the execution of the no operation instruction, allowing a continuous flow of the pipeline.

RISC Pipeline

- **Among the characteristics** attributed to ruse is its ability to use an efficient instruction pipeline.
- **The simplicity** of the instruction set can be utilized to implement an instruction pipeline using a small number of suboperations, with each being executed in one clock cycle.
- **Because of the fixed-length** instruction format, the decoding of the operation can occur at the same time as the register selection. All data manipulation instructions have register-to register operations.
- **Since all operands** are in registers, there is no need for calculating an effective address or fetching of operands from memory.
- 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.
- **A 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. These instructions use register indirect addressing.

- **They usually need three or four stages in the pipeline.**
- 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: one for storing the instructions and the other for storing the data.
- The two memories can sometime operate at the same speed as the CPU clock and are referred to as cache memories.
- **One of the major advantages of RISC is its ability** to execute instructions at the rate of one per clock cycle. It is not possible to expect that every instruction be fetched from memory and executed in one clock cycle.
- **What is done**, in effect, is to start each instruction with each clock cycle and to pipeline the processor to achieve the goal of single-cycle instruction execution.

- **The advantage of RISC over CISC** (complex instruction set computer) is that RISC can achieve pipeline segments, requiring just one clock cycle, while CISC uses many segments in its pipeline, with the longest segment requiring two or more clock cycles.
- Another characteristic of RISC is the support given by the compiler that translates the high-level language program into machine language program.
- **Instead of designing** hardware to handle the difficulties associated with data conflicts and branch penalties, RISC processors rely on the efficiency of the compiler to detect and minimize the delays encountered with these problems.

Three-Segment Instruction Pipeline

- **The data manipulation** instructions operate on data in processor registers.
- The data transfer instructions are load and store instructions that use an effective address obtained from the addition of the contents of two registers or a register and a displacement constant provided in the instruction.
- **The program control** instructions use register values and a constant to evaluate the branch address, which is transferred to a register or the program counter PC.
- **Now consider the hardware operation** for such a computer. The control section fetches the instruction from program memory into an instruction register. 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) that performs the necessary arithmetic, logic, and shift operations.
- 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:

- **The I segment** fetches the instruction from program memory. The instruction is decoded and an ALU operation is performed in the A segment.
- **The ALU** is used for three different functions, depending on the decoded instruction. It performs an operation for a data manipulation instruction, it evaluates the effective address for a load or store instruction, or it calculates the branch address for a program control instruction.
- **The E segment** 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, or it transfers the branch address to the program counter.

I: Instruction fetch

A: ALU operation

E: Execute instruction

Delayed Load

Consider now the operation of the following four instructions:

1. LOAD: $R1 \leftarrow M$[address 1]
2. LOAD: $R2 \leftarrow M$[address 2]
3. ADD: $R3 \leftarrow R1 + R2$
4. STORE: M[address 3] $\leftarrow R3$

Delayed Load

•**Consider** now the operation of the following four instructions:
•1. LOAD: R1 ← M [address 1]
•2. LOAD: R2 ← M [address 2]
•3. ADD: R3 ← R1 + R2
• 4. STORE: M[address 3] ← R3
•**If the three-segment pipeline** proceeds without interruptions, 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. 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, but the value in R2 will not be the correct value since it has not yet been transferred from memory.
•**It is up to the compiler** to make sure that the instruction following the load instruction uses the data fetched from memory. If the compiler cannot find a useful instruction to put after the load, it inserts a no-op (no-operation) instruction.

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

(a) Pipeline timing with data conflict

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

(b) Pipeline timing with delayed load

Figure 9-9 Three-segment pipeline timing.

- **This is a type** of instruction that is fetched from **memory but has no operation**, thus wasting a clock cycle. This concept of delaying the use of the data loaded from memory is referred to as delayed load.
- **Figure** shows the same program with a no-op instruction inserted after the load to R2 instruction. The data is loaded into R2 in clock cycle 4.
- **The add instruction** uses the value of R2 in step 5.
- **Thus the no-op instruction** is used to advance one clock cycle in order to compensate for the data conflict in the pipeline. (Note that no operation is performed in segment A during clock cycle 4 or segment E during clock cycle 5.)
- **The advantage** of the delayed load approach is that the data dependency is taken care of by the compiler rather than the hardware.
- **This results** in a simpler hardware segment since the segment does not have to check if the content of the register being accessed is currently valid or not.

Delayed Branch

Load from memory to *R*1
Increment *R*2
Add *R*3 to *R*4
Subtract *R*5 from *R*6
Branch to address *X*

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

(a) Using no-operation instructions

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

(b) Rearranging the instructions

Vector Processing

There is a class of computational problems that are beyond the capabilities of a conventional computer. These problems are 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.

Long-range weather forecasting
Petroleum explorations
Seismic data analysis
Medical diagnosis
Aerodynamics and space flight simulations
Artificial intelligence and expert systems
Mapping the human genome
Image processing

Vector Operations

- Arithmetic operations on large arrays of numbers
- Conventional scalar processor
- V=[v1,v2,v3…vn]
- V[I] where I means index
- Machine language

```
    Initialize I = 0
20  Read A(I)
    Read B(I)
    Store C(I) = A(I) + B(I)
    Increment I = I + 1
    If  I ≤ 100 go to 20
    Continue
```

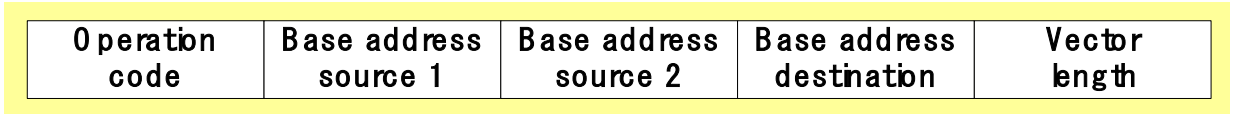
Fortran language

```
      DO  20  I = 1, 100
20    C(I) = A(I) + B(I)
```

- **Vector processor**
- Single vector instruction

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

- **Vector Instruction Format :**



- *ADD* *A* *B* *C* *100*

Matrix Multiplication

3 x 3 matrices multiplication : $\mathbf{n^2 = 9}$ inner product

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

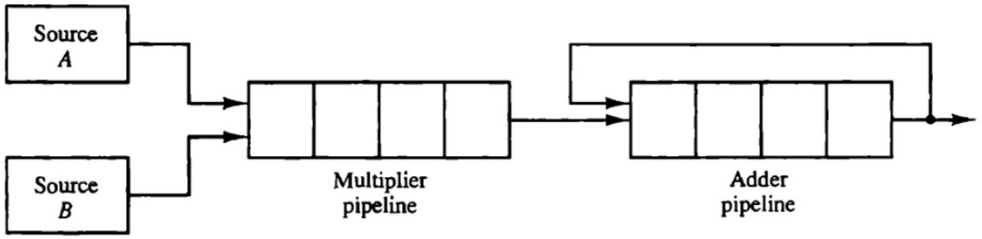
$$c_{ij} = \sum_{k=1}^{3} a_{ik} \times b_{kj}$$

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

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

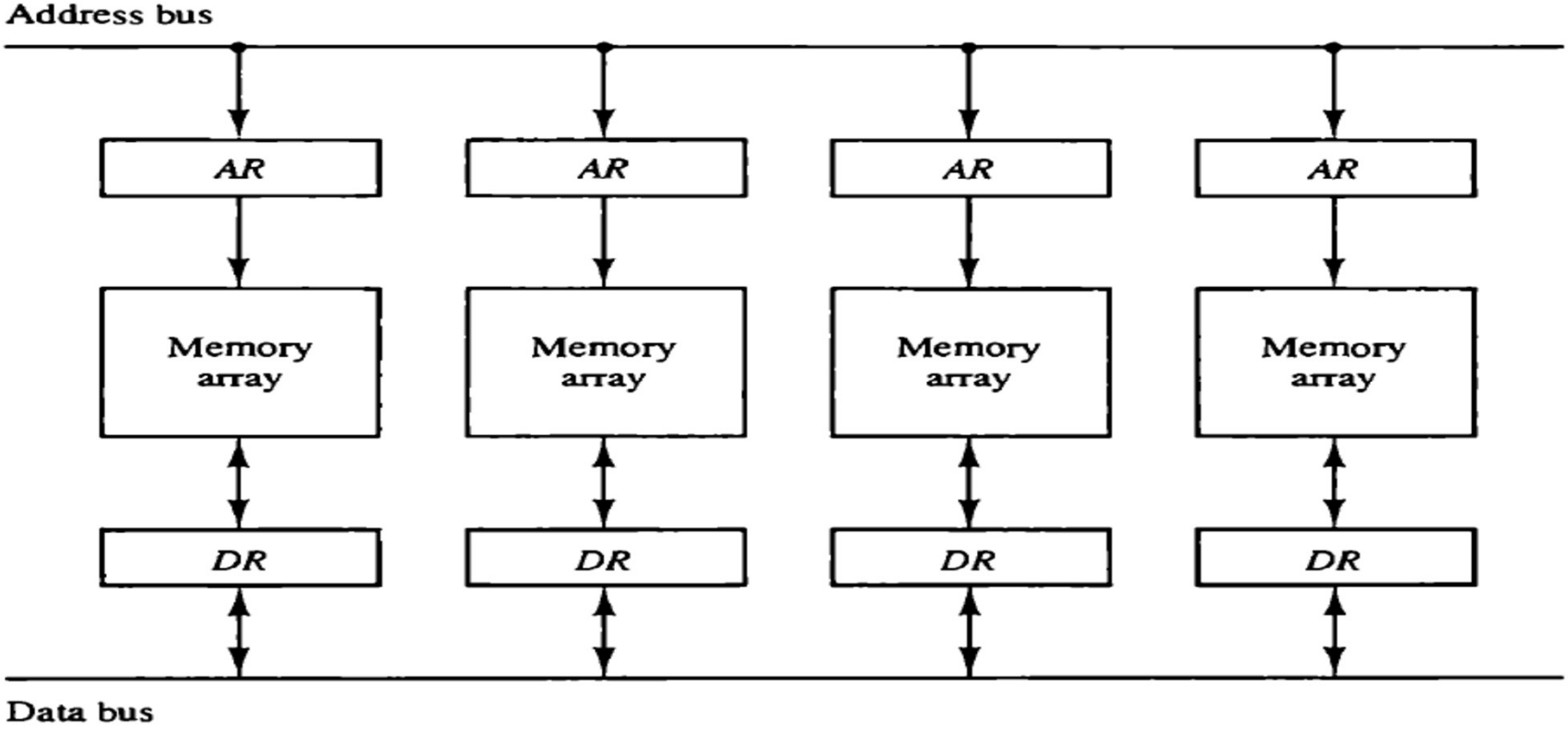
This requires three multiplications and (after initializing c_{11} to 0) three additions. The total number of multiplications or additions required to compute the matrix product is $9 \times 3 = 27$. If we consider the linked multiply–add operation $c + a \times b$ as a cumulative operation, the product of two $n \times n$ matrices requires n^3 multiply–add operations. The computation consists of n^2 inner products, with each inner product requiring n multiply–add operations, assuming that c is initialized to zero before computing each element in the product matrix.

$$C = A_1 B_1 + A_2 B_2 + A_3 B_3 + A_4 B_4 + \cdots + A_k B_k$$



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

Memory Interleaving



Array Processors

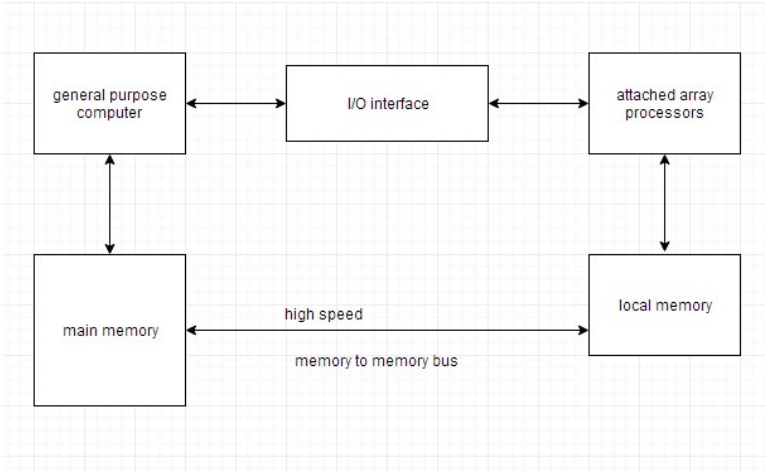
- Array processors are also known as multiprocessors or vector processors. They perform computations on large arrays of data.
- Thus, they are used to improve the performance of the computer
- Types of Array Processors
- There are basically two types of array processors:

1.Attached Array Processors

2.SIMD Array Processors

Attached Array Processors

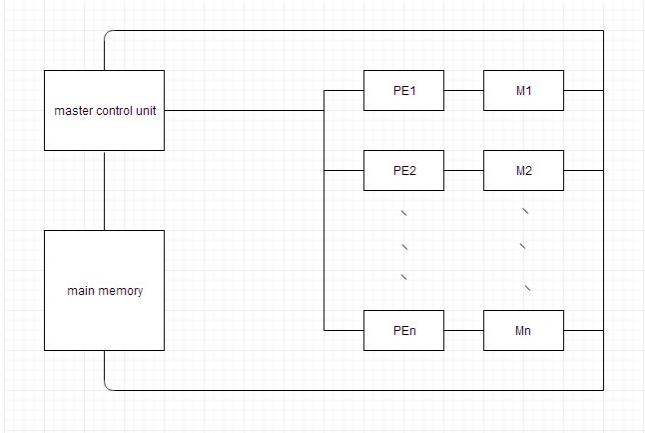
- An attached array processor is a processor which is attached to a general purpose computer and its purpose is to enhance and improve the performance of that computer in numerical computational tasks.
- It achieves high performance by means of parallel processing with multiple functional units.



SIMD Array Processors

- SIMD is the organization of a single computer containing multiple processors operating in parallel.
- The processing units are made to operate under the control of a common control unit, thus providing a single instruction stream and multiple data streams.
- It contains a set of identical processing elements (PE's), each of which is having a local memory M. Each processor element includes an **ALU** and **registers**.
- The master control unit controls all the operations of the processor elements. It also decodes the instructions and determines how the instruction is to be executed.
- The main memory is used for storing the program.
- The control unit is responsible for fetching the instructions. Vector instructions are send to all PE's simultaneously and results are returned to the memory.

- The best known SIMD array processor is the **ILLIAC IV** computer developed by the **Burroughs corps**.
- SIMD processors are highly specialized computers.
- They are only suitable for numerical problems that can be expressed in vector or matrix form and they are not suitable for other types of computations.



Why use the Array Processor

- Array processors increases the overall instruction processing speed.
- As most of the Array processors operates asynchronously from the host CPU, hence it improves the overall capacity of the system.
- Array Processors has its own local memory, hence providing extra memory for systems with low memory.