Unit – 5B
Pipeline and Vector Processing

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.
- 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.
    - The combinational circuit performs the suboperation in the particular segment.
- A clock is applied to all registers after *enough time* has elapsed to perform all segment activity.
- The pipeline organization will be demonstrated by means of a simple example.
    - To perform the combined multiply and add operations with a stream of numbers
      $A_i * B_i + C_i$ for $i = 1, 2, 3, \ldots, 7$
- Each suboperation is to be implemented in a segment within 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
- Each segment has one or two registers and a combinational circuit as shown in Fig. 9-2.
- The five registers are loaded with new data every clock pulse. The effect of each clock is shown in Table 4-1.

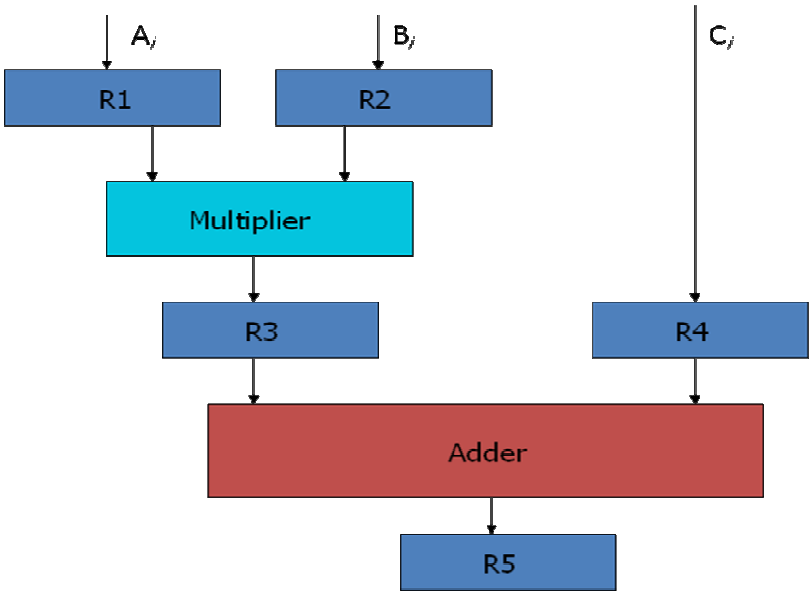

Fig 4-1: Example of pipeline processing

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	--	--	--	--	$A_7*B_7+C_7$

Table 4-1: Content of Registers in Pipeline Example

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 general structure of a four-segment pipeline is illustrated in Fig. 4-2.
- We define a *task* 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.

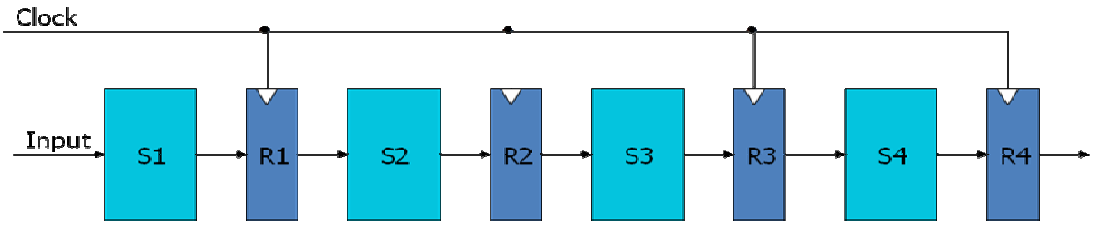

Fig 4-2: Four Segment Pipeline

- The space-time diagram of a four-segment pipeline is demonstrated in Fig. 4-3.
- Where a *k*-segment pipeline with a clock cycle time t_p is used to execute *n* tasks.
  - The first task T_1 requires a time equal to kt_p to complete its operation.
  - The remaining *n*-1 tasks will be completed after a time equal to $(n-1)t_p$
  - Therefore, to complete *n* tasks using a *k*-segment pipeline requires $k+(n-1)$ clock cycles.
- Consider a nonpipeline unit that performs the same operation and takes a time equal to t_n to complete each task.
  - The total time required for *n* tasks is nt_n.

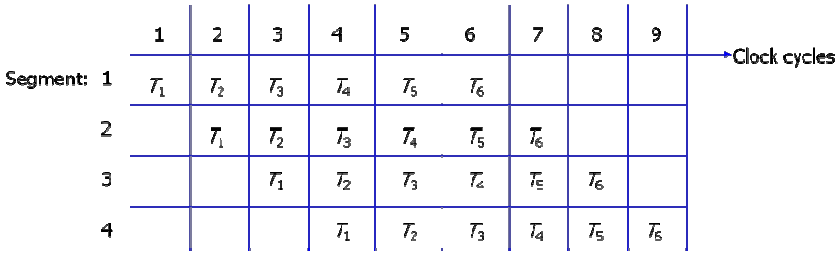

Fig 4-3: Space-time diagram for pipeline

- The *speedup of a pipeline processing* over an equivalent non-pipeline processing is defined by the ratio $S = nt_n/(k+n-1)t_p$.
- If *n* becomes much larger than *k*-1, the speedup becomes $S = t_n/t_p$.
- If we assume that the time it takes to process a task is the same in the pipeline and non-pipeline circuits, i.e., $t_n = kt_p$, the speedup reduces to $S=kt_p/t_p=k$.
- This shows that the theoretical maximum speed up that a pipeline can provide is *k*, where *k* is the number of segments in the pipeline.
- To duplicate the theoretical speed advantage of a pipeline process by means of multiple functional units, it is necessary to construct *k* identical units that will be operating in parallel.
- This is illustrated in Fig. 4-4, where four identical circuits are connected in parallel.
- Instead of operating with the *input data in sequence* as in a pipeline, the parallel circuits accept four input data items *simultaneously* and perform four tasks at the same time.

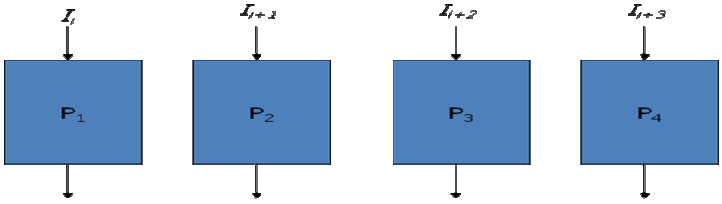

Fig 4-4: Multiple functional units in parallel

- There are various reasons why the pipeline cannot operate at its maximum theoretical rate.
  - Different segments may take different times to complete their sub operation.
  - It is not always correct to assume that a non pipe circuit has the same time delay as that of an equivalent pipeline circuit.
- There are two areas of computer design where the pipeline organization is applicable.
  - Arithmetic pipeline
  - Instruction pipeline

Parallel Processing

- *Parallel processing* is a term used to denote a large class of techniques that are used to provide simultaneous *data-processing tasks* for the purpose of increasing the computational speed of a computer system.
- The purpose of parallel processing is to *speed up the computer processing capability* and *increase its throughput*, that is, the amount of processing that can be accomplished during a given interval of time.
- The amount of hardware increases with parallel processing, and with it, the cost of the system increases.
- Parallel processing 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. e.g. shift registers and registers with parallel load
  - At a higher level, it can be achieved by having a multiplicity of functional units that perform identical or different operations simultaneously.
- Fig. 4-5 shows one possible way of separating the execution unit into eight functional units operating in parallel.
  - A multifunctional organization is usually associated with a complex control unit to coordinate all the activities among the various components.

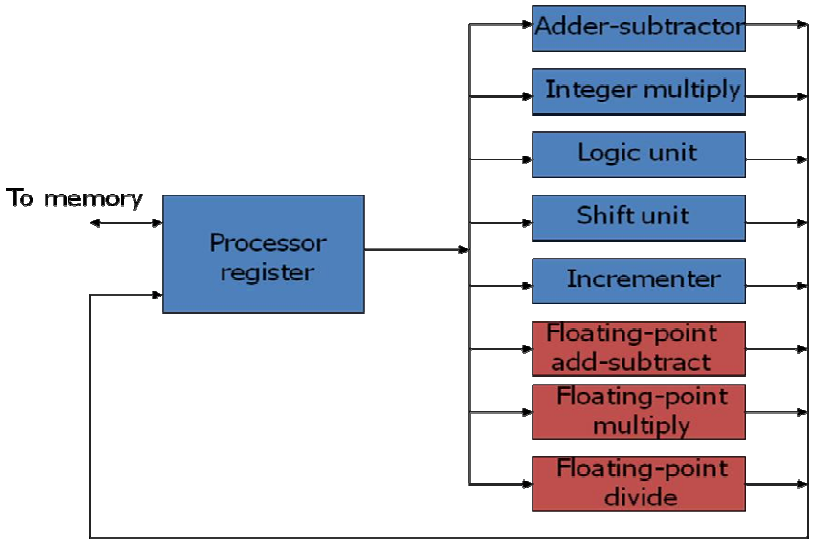

Fig 4-5: Processor with multiple functional units

- There are a variety of ways that parallel processing can be classified.
  - Internal organization of the processors
  - Interconnection structure between processors
  - The flow of information through the system
- M. J. Flynn considers the organization of a computer system by the *number of instructions* and *data items* that are manipulated simultaneously.
  - 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)

SISD

- Represents the organization of a single computer containing a control unit, a 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*.

SIMD

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

MISD & MIMD

- MISD structure is only of theoretical interest since no practical system has been constructed using this organization.
- MIMD organization refers to a computer system capable of processing several programs at the same time. e.g. multiprocessor and multicomputer system
- Flynn's classification depends on the distinction between the performance of the control unit and the data-processing unit.
- It emphasizes the behavioral characteristics of the computer system rather than its operational and structural interconnections.
- One type of parallel processing that does not fit Flynn's classification is pipelining.
- We consider parallel processing under the following main topics:
    - **Pipeline processsing**
        - Is an implementation technique where arithmetic suboperations or the phases of a computer instruction cycle overlap in execution.
    - **Vector processing**
        - Deals with computations involving large vectors and matrices.
    - **Array processing**
        - Perform computations on large arrays of data.

Arithmetic Pipeline

- Pipeline arithmetic units are usually found in very high speed computers
    - Floating–point operations, multiplication of fixed-point numbers, and similar computations in scientific problem
- Floating–point operations are easily decomposed into sub operations.
- An example of a pipeline unit for floating-point addition and subtraction is showed in the following:
    - The inputs to the floating-point adder pipeline are two normalized floating-point binary number

      $X = A \times 2^a$

      $Y = B \times 2^b$
    - A and B are two fractions that represent the mantissas
    - a and b are the exponents
- The floating-point addition and subtraction can be performed in four segments, as shown in Fig. 4-6.
- The suboperations that are performed in the four segments are:
    - *Compare the exponents*
        - The larger exponent is chosen as the exponent of the result.
    - *Align the mantissas*
        - The exponent difference determines how many times the mantissa associated with the smaller exponent must be shifted to the right.
    - *Add or subtract the mantissas*
    - *Normalize the result*

- 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.

- The following numerical example may clarify the suboperations performed in each segment.
- The comparator, shift, adder, subtractor, incrementer, and decrementer in the floating-point pipeline are implemented with combinational circuits.
- Suppose that the time delays of the four segments are t_1=60ns, t_2=70ns, t_3=100ns, t_4=80ns, and the interface registers have a delay of t_r=10ns
  - Pipeline floating-point arithmetic delay: $t_p=t_3+t_r$=110ns
  - Nonpipeline floating-point arithmetic delay: $t_n=t_1+t_2+t_3+t_4+t_r$=320ns
  - Speedup: 320/110=2.9

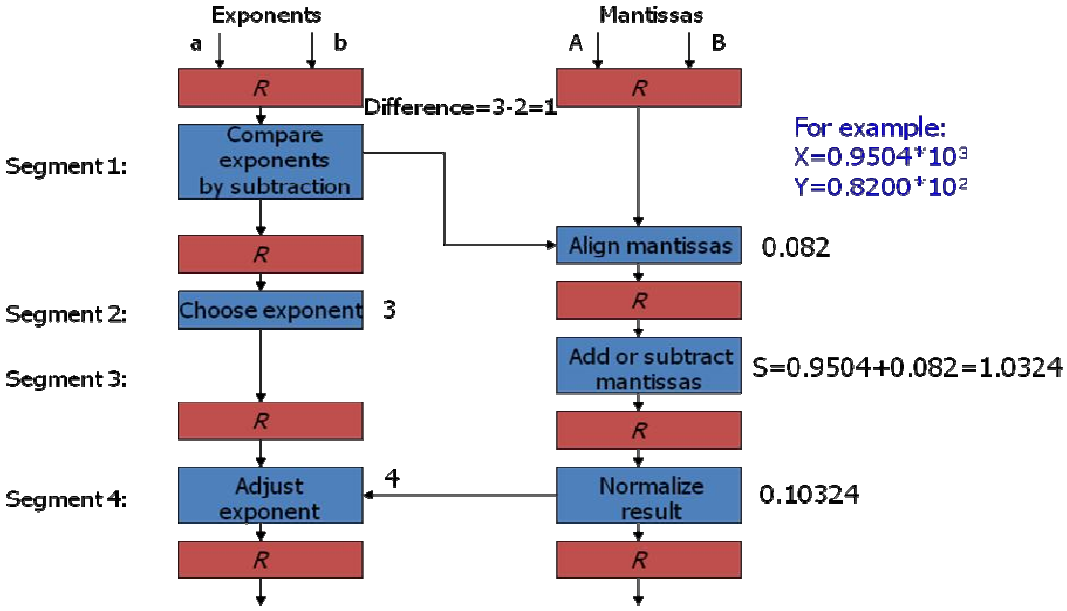

Fig 4-6: Pipeline for floating point addition and subtraction

Instruction Pipeline

- Pipeline processing can occur not only in the *data stream* but in the *instruction* as well.
- Consider a computer with an instruction fetch unit and an instruction execution unit designed to provide a *two-segment* pipeline.
- Computers with complex instructions require other phases in addition to above phases to process an instruction completely.
- In the most general case, the computer needs to process each instruction with the following sequence of steps.
  - Fetch the instruction from memory.
  - Decode the instruction.

- Calculate the effective address.
- Fetch the operands from memory.
- Execute the instruction.
- 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.
  - 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

- Assume that:
  - The decoding of the instruction can be combined with the calculation of the effective address into one segment.
  - The instruction execution and storing of the result can be combined into one segment.
- Fig 4-7 shows how the instruction cycle in the CPU can be processed with a four-segment pipeline.
  - 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.
- An instruction in the sequence may be 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.
  - Similarly, an interrupt request will cause the pipeline to empty and start again from a new address value.

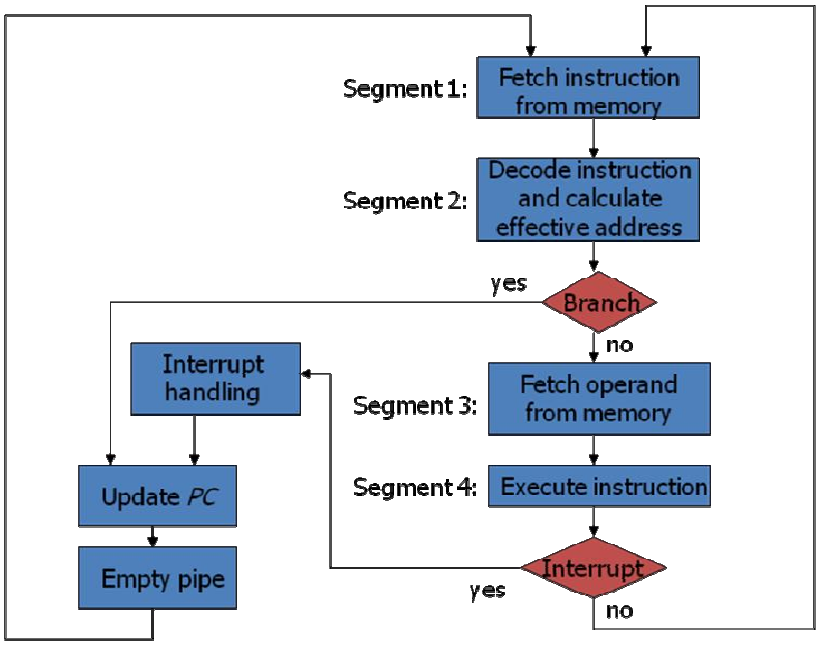

Fig 4-7: Four-segment CPU pipeline

- Fig. 9-8 shows the operation of the instruction pipeline.

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

Fig 4-8: Timing of Instruction Pipeline

  - FI: the segment that fetches an instruction
  - DA: the segment that decodes the instruction and calculate the effective address
  - FO: the segment that fetches the operand
  - EX: the segment that executes the instruction

Pipeline Conflicts

- In general, there are three major difficulties that cause the instruction pipeline to deviate from its normal operation.
  - *Resource conflicts* caused by access to memory by two segments at the same time.

        - Can be resolved by using separate instruction and data memories
    - *Data dependency conflicts* arise when an instruction depends on the result of a previous instruction, but this result is not yet available.
    - *Branch difficulties* arise from branch and other instructions that change the value of PC.
- A difficulty that may cause a degradation of performance in an instruction pipeline is due to possible collision of data or address.
    - A data dependency occurs when an instruction needs data that are not yet available.
    - An address dependency may occur when an operand address cannot be calculated because the information needed by the addressing mode is not available.
- Pipelined computers deal with such conflicts between data dependencies in a variety of ways.

Data Dependency Solutions

- *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.
- *Operand forwarding*: uses 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 multiplexers as well as the circuit that detects the conflict.
- *Delayed load*: 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.

Handling of Branch Instructions

- One of the major problems in operating an instruction pipeline is the occurrence of branch instructions.
    - 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.
- Pipelined computers employ various hardware techniques to minimize the performance degradation caused by instruction branching.
- *Prefetch target instruction*: To prefetch the target instruction in addition to the instruction following the branch. Both are saved until the branch is executed.
- *Branch target buffer(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.

- *Loop buffer*: This is a small very high speed register file maintained by the instruction fetch segment of the pipeline.
- *Branch prediction*: A pipeline with branch prediction uses some additional logic to guess the outcome of a conditional branch instruction before it is executed.
- *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.
    - A procedure employed in most RISC processors.
    - e.g. no-operation instruction

RISC Pipeline

- To use an efficient instruction pipeline
    - 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.
    - 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.
    - 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.
    - 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.
    - 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.
    - RISC can achieve pipeline segments, requiring just one clock cycle.
- *Compiler* supported 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.

Example: Three-Segment Instruction Pipeline

- Thee are three types of instructions:
    - The data manipulation instructions: operate on data in processor registers

    - The data transfer instructions:
    - The program control instructions:
- 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).
- 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:
    - I: Instruction fetch
        - Fetches the instruction from program memory
    - 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.
    - 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:
    - LOAD: R1 ← M[address 1]
    - LOAD: R2 ← M[address 2]
    - ADD: R3 ← R1 +R2
    - STORE: M[address 3] ← 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.
  - Load from memory to R1
  - Increment R2
  - Add R3 to R4
  - Subtract R5 from R6
  - Branch to address X

- In Fig. 4-10(a) the compiler inserts *two no-op instructions* after the branch.
  - 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

Vector Processing

- 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. e.g.
  - Long-range weather forecasting
  - Petroleum explorations
  - Seismic data analysis
  - Medical diagnosis
  - Artificial intelligence and expert systems
  - Image processing
  - Mapping the human genome

- 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 Operations

- Many scientific problems require arithmetic operations on large arrays of numbers.
- A vector is an ordered set of a one-dimensional array of data items.
- A vector V of length n is represented as a row vector by V=[v1,v2,…,Vn].
- To examine the difference between a conventional scalar processor and a vector processor, consider the following Fortran DO loop:

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

- This is implemented in machine language by the following sequence of operations.

```
      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
```

- A computer capable of vector processing eliminates the overhead associated with the time it takes to fetch and execute the instructions in the program loop.

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

- A possible instruction format for a vector instruction is shown in Fig. 4-11.
  - This assumes that the vector operands reside in *memory*.
- It is also possible to design the processor with a large number of *registers* and store all operands in registers prior to the addition operation.
  - The base address and length in the vector instruction specify a group of CPU registers.

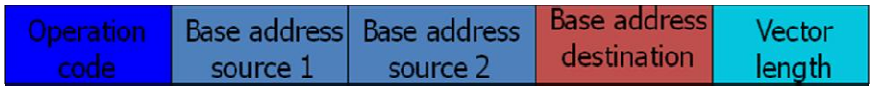

Fig 4-11: Instruction format for vector processor

Matrix Multiplication

- The multiplication of two n x n matrices consists of n^2 inner products or n^3 multiply-add operations.
  - Consider, for example, the multiplication of two 3 x 3 matrices A and B.
  - $c_{11}= a_{11}b_{11}+ a_{12}b_{21}+ a_{13}b_{31}$
  - This requires three multiplication and (after initializing c_{11} to 0) three additions.
- In general, the inner product consists of the sum of *k* product terms of the form $C= A_1B_1+A_2B_2+A_3B_3+\ldots+A_kB_k$.
  - In a typical application k may be equal to 100 or even 1000.

- The inner product calculation on a pipeline vector processor is shown in Fig. 4-12.

$$C = A_1B_1 + A_5B_5 + A_9B_9 + A_{13}B_{13} + \cdots$$
$$+ A_2B_2 + A_6B_6 + A_{10}B_{10} + A_{14}B_{14} + \cdots$$
$$+ A_3B_3 + A_7B_7 + A_{11}B_{11} + A_{15}B_{15} + \cdots$$
$$+ A_4B_4 + A_8B_8 + A_{12}B_{12} + A_{16}B_{16} + \cdots$$

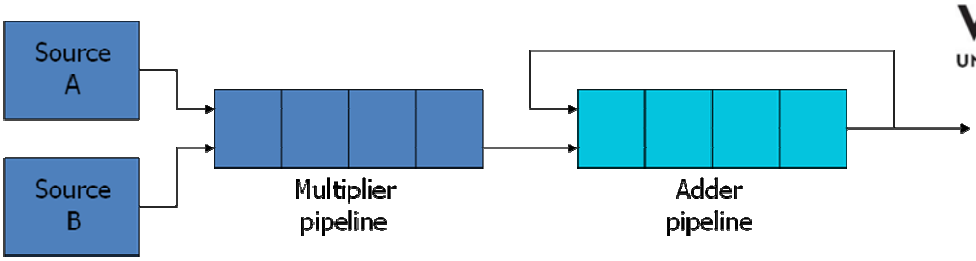

Fig 4-12: Pipeline for calculating an inner product

Memory Interleaving

- *Pipeline* and *vector processors* often require simultaneous access to memory from two or more sources.
  - An instruction pipeline may require the fetching of an instruction and an 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 address and data registers.
- Fig. 4-13 shows a memory unit with four modules.

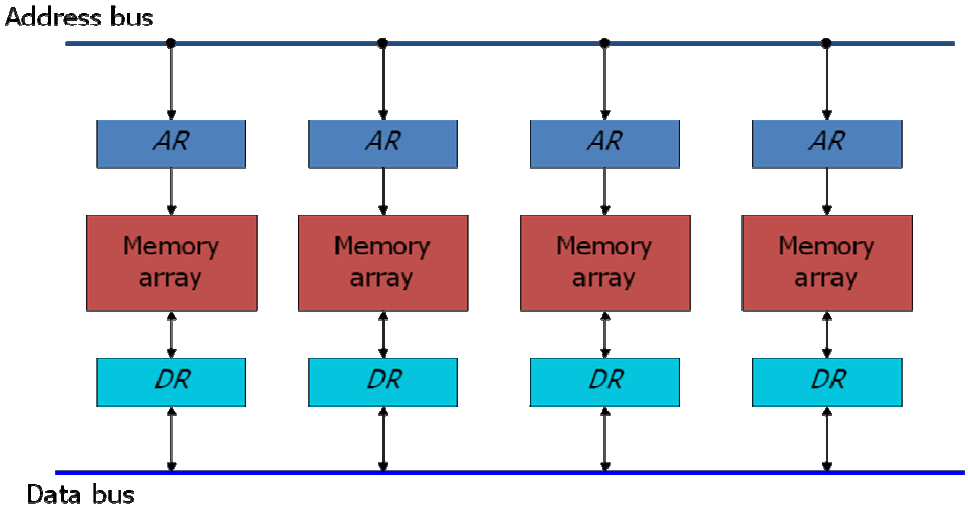

Fig 4-13: Multiple module memory organization

- 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.
- By staggering the memory access, the effective memory cycle time can be *reduced by a factor close to the number of modules*.

Supercomputers

- A commercial computer with vector instructions and pipelined floating-point arithmetic operations is referred to as a *supercomputer*.
    - To speed up the operation, the components are *packed tightly* together to minimize the distance that the electronic signals have to travel.
- This is augmented by instructions that process vectors and combinations of scalars and vectors.
- A supercomputer is a computer system best known for its high computational speed, fast and large memory systems, and the extensive use of parallel processing.
    - It is equipped with *multiple functional units* and each unit has its own *pipeline* configuration.
- It is specifically optimized for the type of numerical calculations involving vectors and matrices of floating-point numbers.
- They are limited in their use to a number of scientific applications, such *as numerical weather forecasting, seismic wave analysis, and space research*.
- A measure used to evaluate computers in their ability to perform a given number of floating-point operations per second is referred to as *flops*.
- A typical supercomputer has a basic cycle time of 4 to 20 ns.
- The examples of supercomputer:
- 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)
    - Fujitsu VP-200: 83 vector instructions and 195 scalar instructions; 300 megaflops

Array Processing

- An array processor is a processor that performs computations on large arrays of data.
- The term is used to refer to two different types of processors.
    - Attached array processor:
        - Is an auxiliary processor.
        - It is intended to improve the performance of the host computer in specific numerical computation tasks.
    - SIMD array processor:
        - Has a single-instruction multiple-data organization.
        - It manipulates vector instructions by means of multiple functional units responding to a common instruction.

Attached Array Processor

- Its purpose is to enhance the performance of the computer by providing vector processing for complex scientific applications.
  - Parallel processing with multiple functional units
- Fig. 4-14 shows the interconnection of an attached array processor to a host computer.
- For example, when attached to a VAX 11 computer, the FSP-164/MAX from Floating-Point Systems increases the computing power of the VAX to 100megaflops.
- The objective of the attached array processor is to provide *vector manipulation capabilities* to a conventional computer at a fraction of the cost of supercomputer.

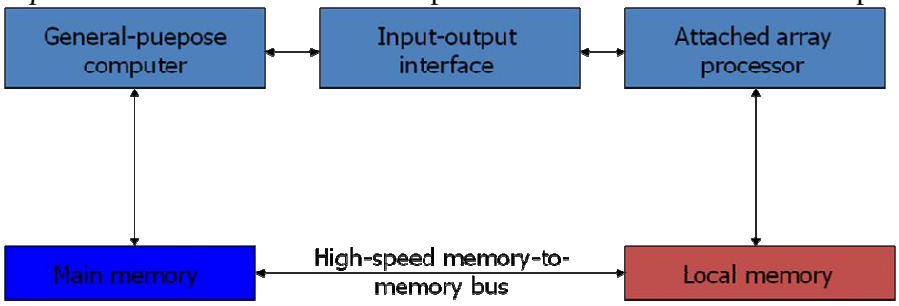

Fig 9-14: Attached array processor with host computer

SIMD Array Processor

- An SIMD array processor is a computer with multiple processing units operating in parallel.
- A general block diagram of an array processor is shown in Fig. 9-15.
  - It contains a set of identical processing elements (PEs), each having a local memory *M*.
  - Each PE includes an ALU, a floating-point arithmetic unit, and working registers.
  - Vector instructions are broadcast to all PEs simultaneously.
- Masking schemes are used to control the status of each PE during the execution of vector instructions.
  - Each PE has a flag that is set when the PE is active and reset when the PE is inactive.
- For example, the ILLIAC IV computer developed at the University of Illinois and manufactured by the Burroughs Corp.
  - Are highly specialized computers.
  - They are suited primarily for numerical problems that can be expressed in vector or matrix form.

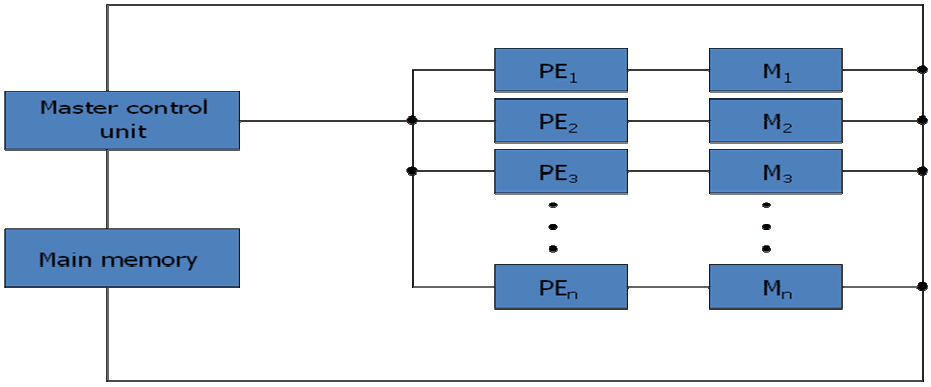

Fig 4-15: SIMD array processor organization