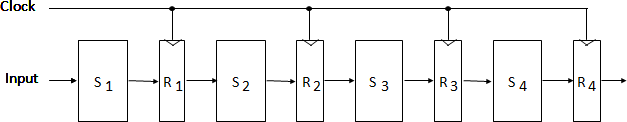
# Explain pipelining technique. Draw the general structure of four segment pipeline.

* + Pipeline 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.
  + 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.
  + 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.
  + Any operation that can be decomposed into a sequence of sub operations 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 four-segment pipeline is illustrated in Figure 6.5.



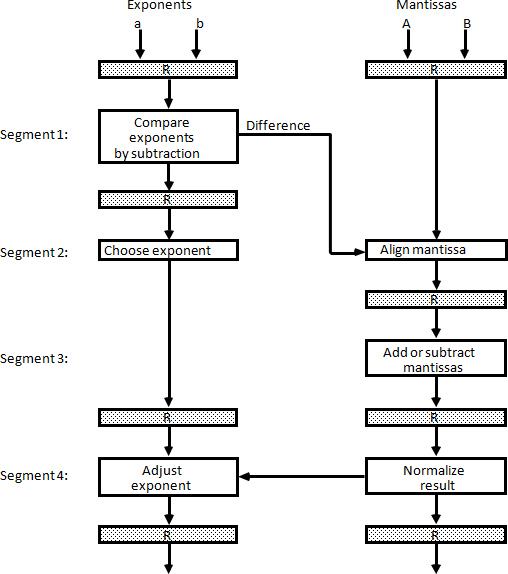
**Figure 6.5: General Structure of Four-Segment Pipeline**

* + The operands pass through all four segments in a fixed sequence.
  + Each segment consists of a combinational circuit S, which performs a sub operation over the data stream flowing through the pipe.
  + The segments are separated by registers R, which 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 define a task as the total operation performed going through all the segments in the pipeline.

# Draw and explain Arithmetic Pipeline.

* + The inputs to the floating-point adder pipeline are two normalized floating-point binary numbers.

X = A x 2a Y = B x 2b



**Figure 6.6: Pipeline for floating-point addition and subtraction**

* + 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 6.6.
  + The registers labeled R are placed between the segments to store intermediate results.
  + The sub-operations 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 following numerical example may clarify the sub-operations performed in each segment.
   * For simplicity, we use decimal numbers, although Figure 6.6 refers to binary numbers.
   * Consider the two normalized floating-point numbers: X = 0.9504 x 103

Y = 0.8200 x 102

* + 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 x 103

Y = 0.0820 x 103

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

Z = 1.0324 x 103

# Explain the Instruction Pipelining with example.

**OR Discuss four-segment instruction pipeline with diagram(s).**

* + Pipeline processing can occur in data stream as well as in instruction stream.
  + An instruction pipeline reads consecutive instructions from memory while previous instructions are being executed in other segments.
  + This causes the instruction fetch and executes 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.
  + The buffer acts as a queue from which control then extracts the instructions for the execution unit.

***Instruction cycle:***

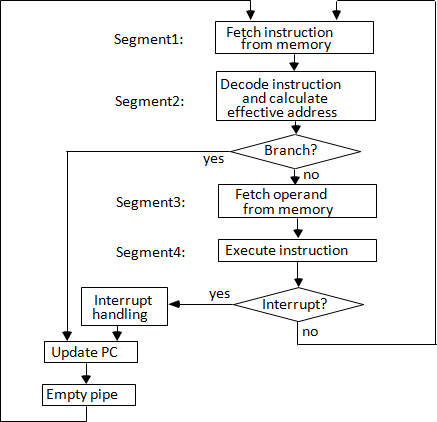
* + The fetch and execute to process an instruction completely.
  + In the most general case, the computer needs to process each instruction with the following sequence of steps

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.
   * 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 fulfill its function depends on the instruction and the way it is executed.

## Example: Four-Segment Instruction Pipeline

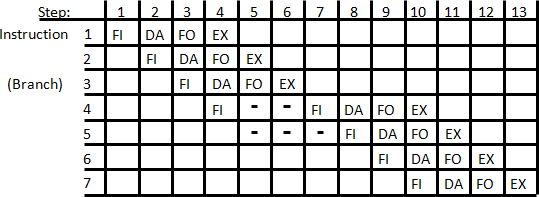
* + Assume that the decoding of the instruction can be combined with the calculation of the effective address into one segment.
  + Assume further that most of the instructions place the result into a processor registers so that the instruction execution and storing of the result can be combined into one segment.
  + This reduces the instruction pipeline into four segments.

1. FI: Fetch an instruction from memory
2. DA: Decode the instruction and calculate the effective address of the operand
3. FO: Fetch the operand
4. EX: Execute the operation
   * Figure 6.7 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 sub operations in the instruction cycle can overlap and up to four different instructions can be in progress of being processed at the same time.



**Figure 6.7: Four-segment CPU pipeline**

* + Figure 6.8 shows the operation of the instruction pipeline. The time in the horizontal axis is divided into steps of equal duration. The four segments are represented in the diagram with an abbreviated symbol.



**Figure 6.8: Timing of Instruction Pipeline**

* + It is assumed that the processor has separate instruction and data memories so that the operation in FI and FO can proceed at the same time.
  + Thus, in step 4, instruction 1 is being executed in segment EX; the operand for instruction 2 is being fetched in segment FO; instruction 3 is being decoded in segment DA; and instruction 4 is being fetched from memory in segment FI.
  + Assume now that instruction 3 is a branch instruction.
  + As soon as this instruction is decoded in segment DA in step 4, the transfer from FI to DA of the other instructions is halted until the branch instruction is executed in step 6.
  + If the branch is taken, a new instruction is fetched in step 7. If the branch is not taken, the instruction fetched previously in step 4 can be used.
  + The pipeline then continues until a new branch instruction is encountered.
  + Another delay may occur in the pipeline if the EX segment needs to store the result of the operation in the data memory while the FO segment needs to fetch an operand.
  + In that case, segment FO must wait until segment EX has finished its operation.

# What is pipeline conflict? Explain data dependency and handling of branch instruction in detail.

## Pipeline conflict:

There are three major difficulties that cause the instruction pipeline conflicts.

* 1. Resource conflicts caused by access to memory by two segments at the same time.
  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 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.
  + Similarly, 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 as follows.

### 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.

### Operand forwarding:

* + It 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:

* + Sometimes compiler has the responsibility for solving data conflicts problems.
  + 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 instruction, this method is called 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.
  + The branch instruction breaks the normal sequence of the instruction stream, causing difficulties in the operation of the instruction pipeline.
  + Various hardware techniques are available to minimize the performance degradation caused by instruction branching.

### Pre-fetch target:

* + One way of handling a conditional branch is to prefetch the target instruction in addition to the instruction following the branch.
  + 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.

### Branch target buffer:

* + 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.
  + The advantage of this scheme is that branch instructions that have occurred previously are readily available in the pipeline without interruption.

### Loop buffer:

* + 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.

### 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 pre-fetching the instruction stream from the predicted path.
  + A correct prediction eliminates the wasted time caused by branch penalties.

### Delayed branch:

* + A procedure employed in most RISC 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.

# Explain (i) Vector Processing (ii) Vector Operations. Explain how matrix multiplication is carried out on a computer supporting Vector Computations.

# 

# *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.

### Applications of Vector processing

* 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

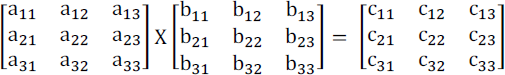
***Vector Operations***

* + Many scientific problems require arithmetic operations on large arrays of numbers.
  + These numbers are usually formulated as vectors and matrices of floating-point 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 V3 … Vn] that may be

represented as a column vector if the data items are listed in a column.

* + A conventional sequential computer is capable of processing operands one at a time.
  + Consequently, operations on vectors must be broken down into single computations with subscripted variables.
  + The element of vector V is written as V(I) and the index I refers to a memory address or register where the number is stored.

### Matrix Multiplication

* + Matrix multiplication is one of the most computational intensive operations performed in computers with vector processors.
  + An n x m matrix of numbers has n rows and m columns and may be considered as constituting a set of n row vectors or a set of m column vectors.
  + Consider, for example, the multiplication of two 3x3 matrices A and B.
  + The product matrix C is a 3 x 3 matrix whose elements are related to the elements of A and B by the inner product:



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

C11 = a11 b11 + a12 b21 + a13 b31

* + This requires three multiplications and three additions.
  + The total number of multiplications or additions required to compute the matrix product is 9 x 3 = 27.
  + If we consider the linked multiply-add operation c + a x b as a cumulative operation, the product of two n x n matrices requires n3 multiply-add operations.
  + The computation consists of n2 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.
  + In general, the inner product consists of the sum of k product terms of the form

C = A1B1 + A2B2 + A3B3 + A4B4 + + AkBk

* + 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 following figure.

**Source A**

**Multiplier pipeline**

**Source B**

**Figure 6.9: Pipeline for Inner Product**

**Adder pipeline**

* + The values of A and B are either in memory or in processor registers.
  + The floating point multiplier pipeline and the floating point adder pipeline are assumed to have four segments each.
  + All segment registers in the multiplier and adder are initialized to 0.
  + Therefore, the output of the adder is 0 for the first eight cycles until both pipes are full.

C = A1B1 + A5B5 + A9B3A9 + A13B13 + ⋯

+ A2B2 + A6B6 + A10B10 + A14B14 + ⋯

+ A3B3 + A7B7 + A11B11 + A15B15 + ⋯

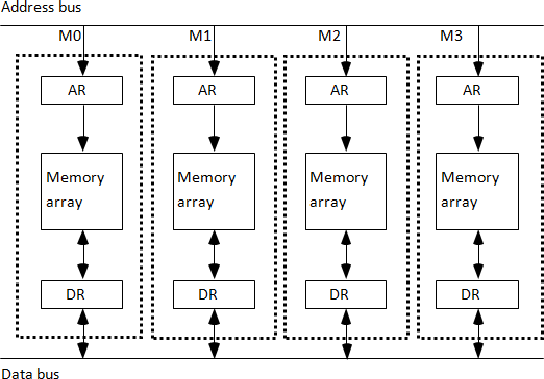
+ A4B4 + A8B8 + A12B12 + A16B16 + ⋯

* + When there are no more product terms to be added, the system inserts four zeros into

the multiplier pipeline.

* + The adder pipeline will then have one partial product in each of its four segments, corresponding to the four sums listed in the four rows in the above equation.
  + The four partial sums are then added to form the final sum.

# Explain Memory Interleaving.

* + 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.
  + Figure 6.10 shows a memory unit with four modules.
  + Each memory array has its own address register AR and data register DR.
  + The address registers receive information from a common address bus and the data registers communicate with a bidirectional data bus.

**Figure 6.10: Multiple module memory organization**

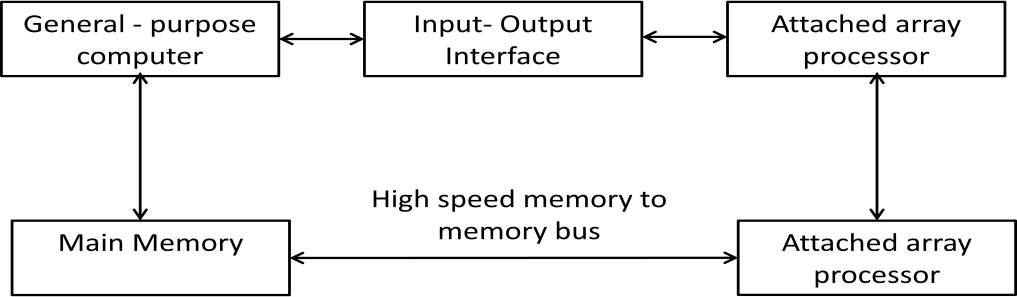
* + The modular system permits one module to initiate a memory access while other modules are in the process of reading or writing a word and each module can honor a memory request independent of the state of the other modules.
  + The advantage of a modular memory is that it allows the use of a technique called interleaving.
  + In an interleaved memory, different sets of addresses are assigned to different memory modules.

# What is an array processor? Explain the different types of array processor.

* + 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 and SIMD array processor.
  + An attached array processor is an auxiliary processor attached to a general-purpose computer.
  + It is intended to improve the performance of the host computer in specific numerical computation tasks.
  + An SIMD array processor is a processor that 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

* + An attached array processor is designed as a peripheral for a conventional host computer, and its purpose is to enhance the performance of the computer by providing vector processing for complex scientific applications.
  + It achieves high performance by means of parallel processing with multiple functional units.
  + It includes an arithmetic unit containing one or more pipelined floating-point adders and multipliers.
  + The array processor can be programmed by the user to accommodate a variety of complex arithmetic problems.
  + Figure 6.11 shows the interconnection of an attached array processor to host computer.

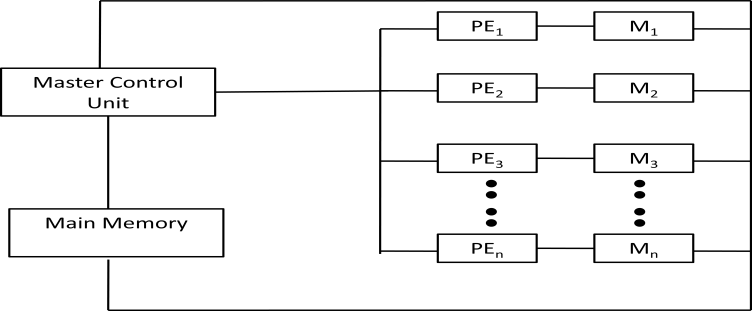


**Figure 6.11: Attached array processor with host computer.**

* + The host computer is a general-purpose commercial computer and the attached processor is a back-end machine driven by the host computer.
  + The array processor is connected through an input-output controller to the computer and the computer treats it like an external interface.
  + The data for the attached processor are transferred from main memory to a local memory through a high-speed bus.
  + The general-purpose computer without the attached processor serves the users that need conventional data processing.
  + The system with the attached processor satisfies the needs for complex arithmetic applications.

## SIMD Array Processor

* + An SIMD array processor is a computer with multiple processing units operating in parallel.
  + The processing units are synchronized to perform the same operation under the control of a common control unit, thus providing a single instruction stream, multiple data stream (SIMD) organization.
  + A general block diagram of an array processor is shown in Figure



**Figure 6.12: SIMD array processor organization**

* + It contains a set of identical processing elements (PEs), each having a local memory M.
  + Each processor element includes an ALU, a floating-point arithmetic unit and working registers.
  + The master control unit controls the operations in the processor elements.
  + The main memory is used for storage of the program.
  + The function of the master control unit is to decode the instructions and determine how the instruction is to be executed.
  + Scalar and program control instructions are directly executed within the master control unit.
  + Vector instructions are broadcast to all PEs simultaneously.
  + Vector operands are distributed to the local memories prior to the parallel execution of the instruction.
  + 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.
  + This ensures that only that PE’S that needs to participate is active during the execution of

the instruction.

* + SIMD processors are highly specialized computers.
  + They are suited primarily for numerical problems that can be expressed in vector matrix form.
  + However, they are not very efficient in other types of computations or in dealing with conventional data-processing programs.