COMPUTER ORGANIZATION AND ARCHITECTURE
UNIT –V
OVERVIEW OF PIPELINIG TECHNIQUE & PROCESSOR ARCHITECTURE
TOPIC- PARALLEL PROCESSING

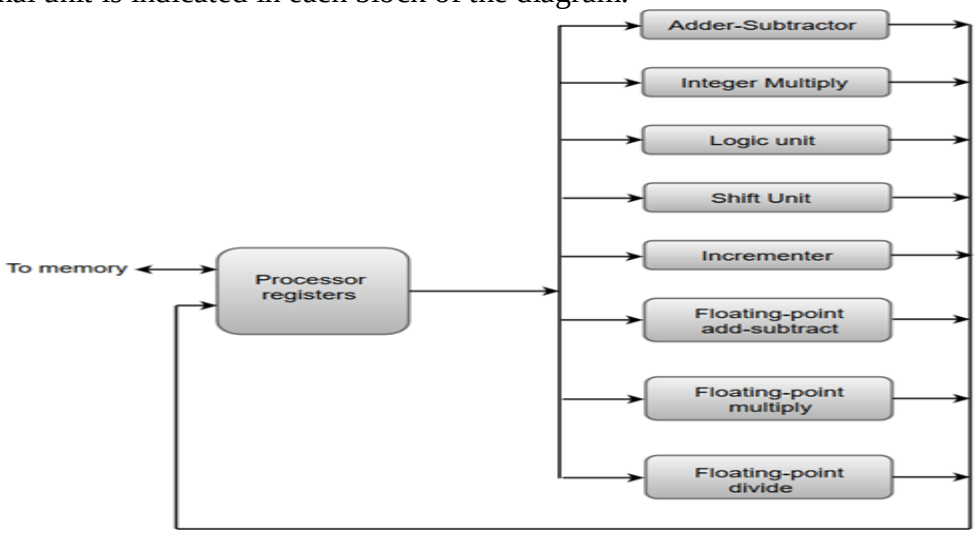
Parallel Processing

Need of Parallel Processing

- The need of parallel processing is to enhance the computer processing capability and increase its throughput, i.e. the amount of processing that can be accomplished during a given interval of time.

Introduction

- Parallel processing can be described as a class of techniques which enables the system to achieve simultaneous data-processing tasks to increase the computational speed of a computer system.
- A parallel processing system can carry out simultaneous data-processing to achieve faster execution time.
- The primary purpose of parallel processing is to enhance the computer processing capability and increase its throughput, i.e. the amount of processing that can be accomplished during a given interval of time.
- A parallel processing system can be achieved by having a multiplicity of functional units that perform identical or different operations simultaneously.
- The data can be distributed among various multiple functional units.
- The following diagram shows one possible way of separating the execution unit into eight functional units operating in parallel and the operation performed in each functional unit is indicated in each block of the diagram:



- The adder and integer multiplier performs the arithmetic operation with integer numbers.
- The floating-point operations are separated into three circuits operating in parallel.

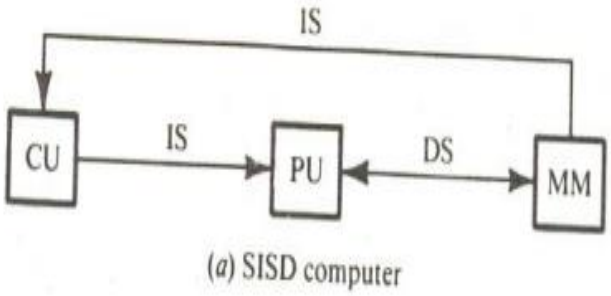
- The logic, shift, and increment operations can be performed concurrently on different data.
- All units are independent of each other.

Classification of Parallel Processing (or) Flynn's classification (or) Flynn's Taxonomy

- There are a variety of ways that parallel processing can be classified. it can be considered from the:
  - Internal organization of the processors
  - Interconnection structure between processors
  - The flow of information through the system
- One classification introduced by M. J. Flynn considers the organization of a computer system by the number of instructions and data items that are manipulated simultaneously.
- The normal operation of a computer is to fetch instructions from memory and execute them in the processor.
- The sequence of instructions read from memory constitutes an instruction stream.
- The operations performed on the data in the processor constitute a data stream
- Parallel processing may occur in the instruction stream, in the data stream, or in both.
- Flynn's classification divides computers into four major groups as follows:
  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)

1.Single Instruction stream, Single Data stream (SISD)

- 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.
- Ex: Von-Neumann computers

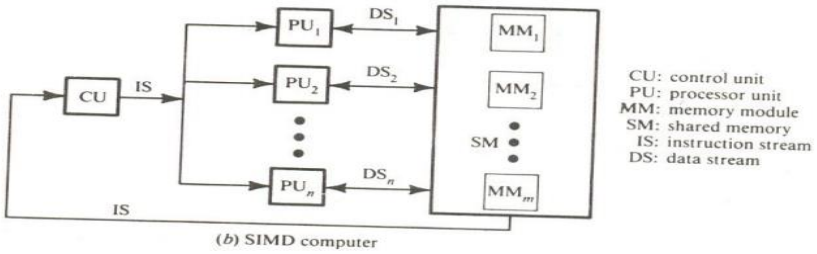


(a) SISD computer

2.Single Instruction stream, Multiple Data stream (SIMD)

- Here there are multiple processing elements which is supervised by the same control unit.
- ALL PE's receive the same instruction broadcast from the control unit but operate on different data sets from data streams.
- The shared memory subsystem may contain multiple modules.

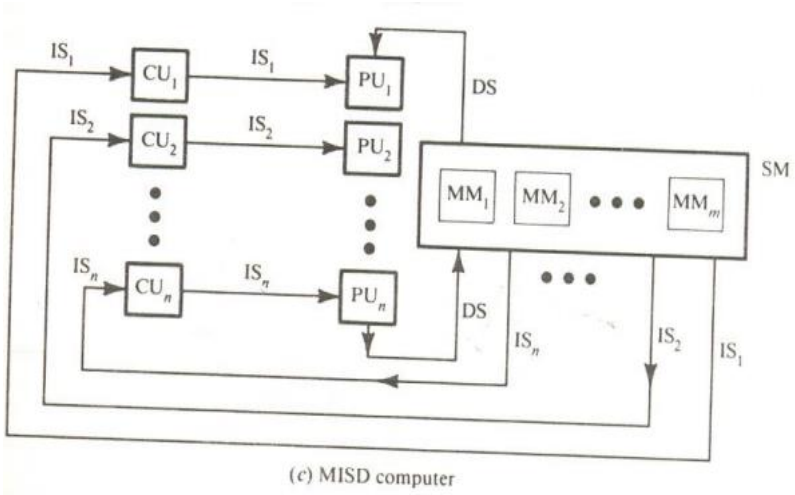
Ex: Pipelined Processors



(b) SIMD computer

3. Multiple Instruction stream, Single Data stream (MISD)

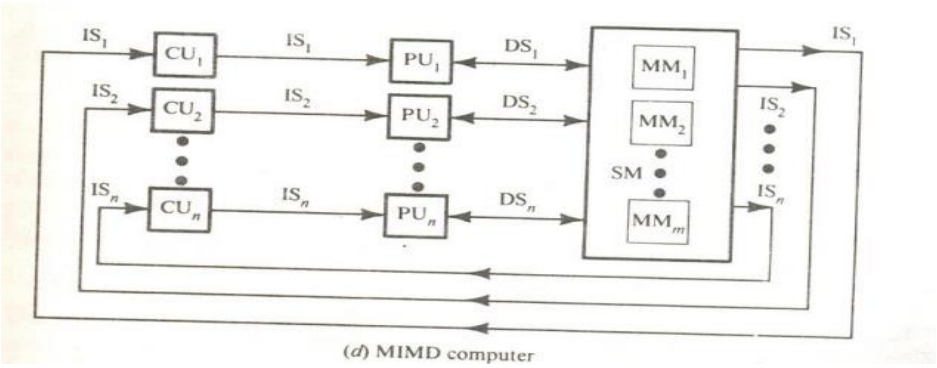
- Here there are n processor units, each receiving distinct instructions operating over the same data stream.
- The results of the one processor become the input of the next processor in the macro pipe.
- This structure receives less attention and has been challenged as impractical in some application.



(c) MISD computer

4.**Multiple Instruction stream, Multiple Data stream (MIMD)**

- Most multiprocessor systems and multiple computer systems come under this category.
- An intrinsic MIMD computer implies interactions among the 'n' processors because all memory stream are derived from the same data space shared by all processors.
- If the 'n' data streams were derived from disjointed subspaces of the shared memories, it is called as multiple SISD (MSISD).
- Ex: superscalar Processors.



(d) MIMD computer

Advantages of Parallel Computing over Serial Computing

1. It saves time and money as many resources working together will reduce the time

and cut potential costs.
2. It can be impractical to solve larger problems on Serial Computing.
3. It can take advantage of non-local resources when the local resources are finite.

Disadvantages of Parallel Computing over Serial Computing

1. Increases the cost of computers since more hardware is required.
2. Multicore architectures consume higher power.
3. Parallel architectures are difficult to achieve.
4. It increases the overhead cost due to synchronization and data transfers.

COMPUTER ORGANIZATION AND ARCHITECTURE

UNIT –V

OVERVIEW OF PIPELINIG TECHNIQUE & PROCESSOR ARCHITECTURE

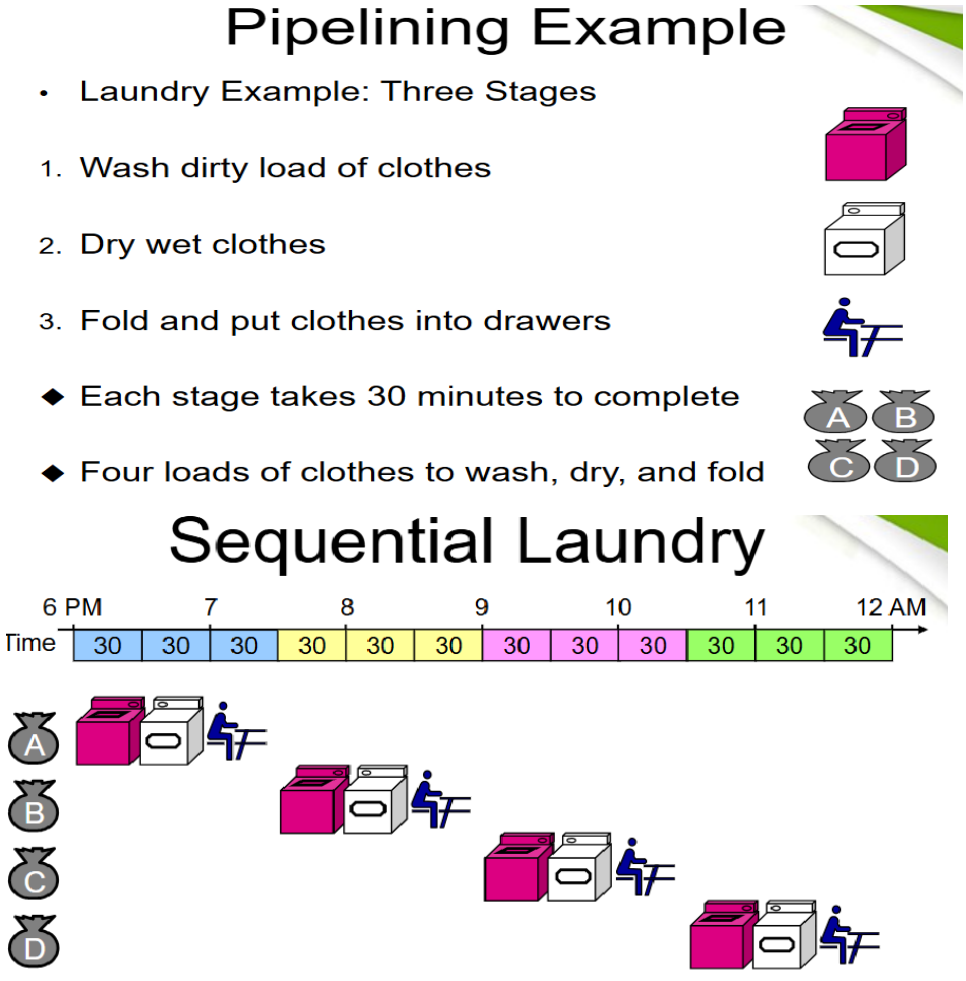
TOPIC- PIPELINING

Pipelining

- Pipelining is a technique of decomposing a sequential process into sub operations, with each sub process being executed in a special dedicated segment that operates concurrently with all other segments.
- 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.

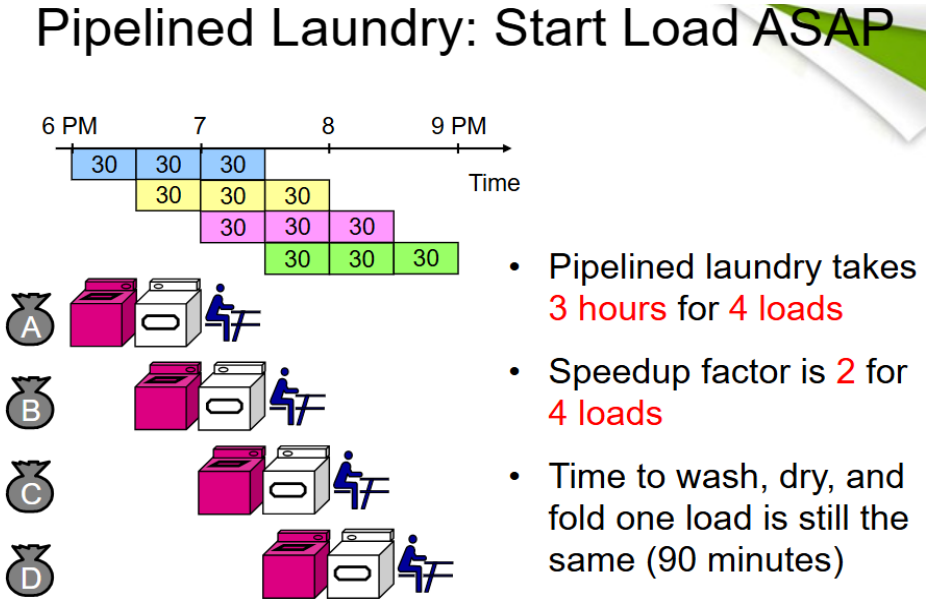
Examples of Pipelining

Example 1:



Sequential laundry takes 6 hours for 4 loads

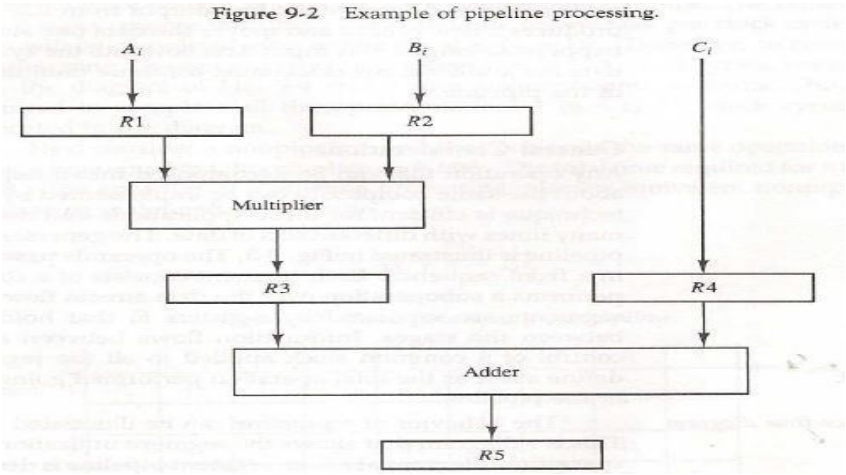
Intuitively, we can use pipelining to speed up laundry



Example 2:

- 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 sub operation is to be implemented in a segment within a pipeline.
  Segment 1: R1←A*i*,R2←B*i* Input A*i* and B*i*
  Segment 2:R3←R1*R2,R4←C*i* Multiply and input C*i*
  Segment 3:R5←R3+R4 Add C*i* to product
- The five registers are loaded with new data every clock pulse.
- The effect of each clock is shown in Table 9-1.
- The first clock pulse transfers A1 and 31 into R1 andR2.
- The second clock pulse transfers the product of R1 and R2 into R3 and C1into 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 A3 and B3 into R1 and R2, transfers the product of R1 and R2 into R3, transfers C2 into R4, and places the sum of R3 and R4 into R5.
- It takes three clock pulses to fill up the pipe and retrieve the first output from R5.
- From there on, each clock produces a new output and moves the data one step down the pipeline.
- Each segment has one or two registers and a combinational circuit as shown in Fig. 9-2.

Figure 9-2 Example of pipeline processing.



- We are considering the implementation of A[7] array with B[7] array

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	—	—	—	—	$A_7 * B_7 + C_7$

- If the above task is executed without the pipelining, then each data operation will take 5 cycles, totally they are 35 cycles of CPU are needed to perform the operation.
- But if are using the concept of pipeline, the task can be executed in 9 cycles.

Principles of Pipelining

- Each task is divided into number of successive tasks.
- A pipeline stage is associated with each task.
- Same amount of time is required for each pipeline stage.
- The output of one stage is given as input to next stage.
- Pipeline is clocked synchronously

General considerations

- 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 general structure of a four-segment pipeline is illustrated in Fig. 9-3.

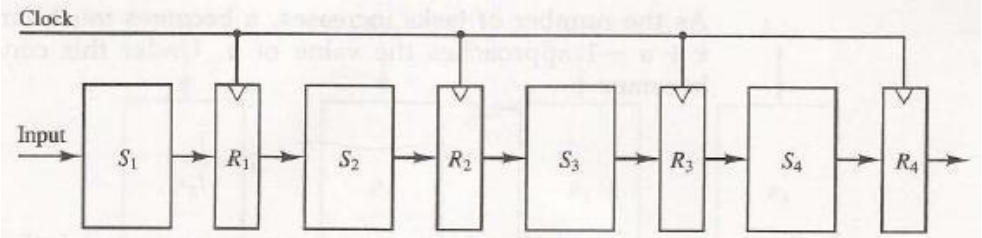


Figure 9-3 Four-segment pipeline.

- The behavior of a pipeline can be illustrated with a space-time diagram.
- The space-time diagram of a four-segment pipeline is demonstrated in Fig. 9-4.

Serial execution vs Pipelining

- In a k-segment pipeline with a clock cycle time t_p is used to execute a task.
- The first task T1 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 $k+(n-1)$ t_p clock cycles.
- Consider a non pipeline 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.

Figure 9-4 Space-time diagram for pipeline.

Segment:	1	2	3	4	5	6	7	8	9	Clock cycles
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	

- The speedup of a pipeline processing over an equivalent non-pipelined 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-pipelined circuits, i.e.,$t_n = kt_p$, the speedup reduces to $S=kt_p/t_p=k$.
- This shows that maximum speedup 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. 9-5, where four identical circuits are connected in parallel.

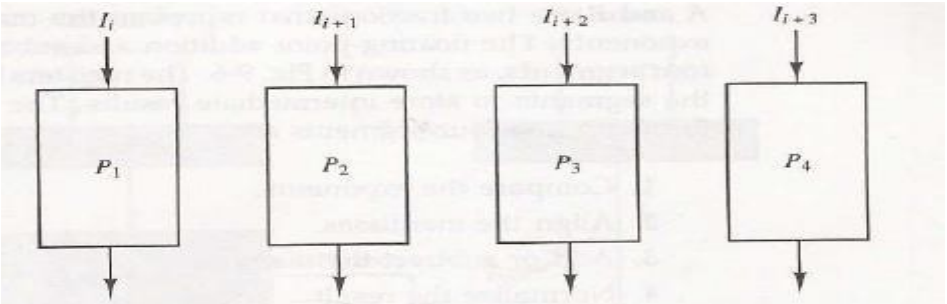


Figure 9-5 Multiple functional units in parallel.

COMPUTER ORGANIZATION AND ARCHITECTURE
UNIT –V
OVERVIEW OF PIPELINIG TECHNIQUE & PROCESSOR ARCHITECTURE
TOPIC- ARITHMETIC PIPELINE

Arithmetic Pipeline

- An arithmetic pipeline divides an arithmetic problem into various sub problems for execution in various pipeline segments.
- Arithmetic Pipelines are mostly used in high-speed computers.
- They are used to implement floating-point operations, multiplication of fixed-point numbers, and similar computations encountered in scientific problems.
- To understand the concepts of arithmetic pipeline, let us consider an example of a pipeline unit for floating-point addition and subtraction with two normalized floating-point binary numbers defined as:

$$X = A * 2^a = 0.9504 * 10^3$$
$$Y = B * 2^b = 0.8200 * 10^2$$

Where **A** and **B** are two fractions that represent the mantisas, **a** and **b** are the exponents.

- The combined operation of floating-point addition and subtraction is divided into four segments.
- Each segment contains the corresponding sub-operation to be performed in the given pipeline.
- The sub-operations that are shown in the four segments are:
  1. Compare the exponents by subtraction.
  2. Align the mantissas.
  3. Add or subtract the mantissas.
  4. Normalize the result.
- The process or flowchart arithmetic pipeline for floating point addition is shown in the diagram.

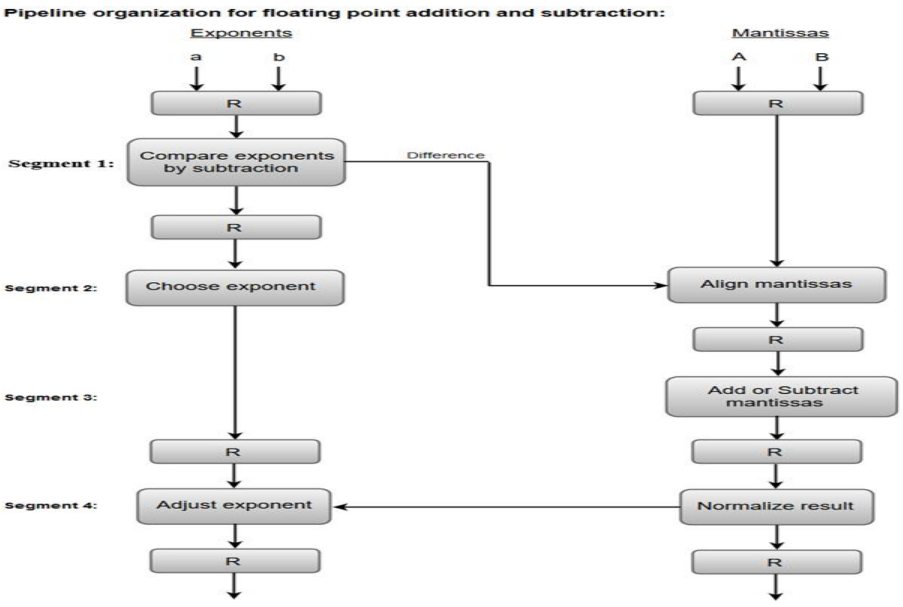


Figure : Flowchart of Arithmetic Pipeline

1. Compare exponents by subtraction:

- The exponents are compared by subtracting them to determine their difference. The larger exponent is chosen as the exponent of the result.
- The difference of the exponents, i.e., **3 - 2 = 1** determines how many times the mantissa associated with the smaller exponent must be shifted to the right.

2. Align the mantissas:

- The mantissa associated with the smaller exponent is shifted according to the difference of exponents determined in segment one.

$X = 0.9504 * 10^3$

$Y = 0.08200 * 10^3$

3. Add mantissas:

- The two mantissas are added in segment three.

$Z = X + Y = 1.0324 * 10^3$

4. Normalize the result:

- After normalization, the result is written as:

$Z = 0.10324 * 10^4$

COMPUTER ORGANIZATION AND ARCHITECTURE
UNIT –V
OVERVIEW OF PIPELINIG TECHNIQUE & PROCESSOR ARCHITECTURE
TOPIC- INSTRUCTION PIPELINE

Instruction Pipeline

- Pipeline processing can occur not only in the data stream but in the instruction stream as well.
- Most of the digital computers with complex instructions require instruction pipeline to carry out operations like fetch, decode and execute instructions.
- In this a stream of instructions can be executed by overlapping fetch, decode and execute phases of an instruction cycle.
- This type of technique is used to increase the throughput of the computer system.
- An instruction pipeline reads instruction from the memory while previous instructions are being executed in other segments of the pipeline.
- Thus we can execute multiple instructions simultaneously.
- The pipeline will be more efficient if the instruction cycle is divided into segments of equal duration.
- In general, the computer needs to process each instruction with the following sequence of steps:
  1. Fetch 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 results.
- The flowchart for instruction pipeline is shown below.

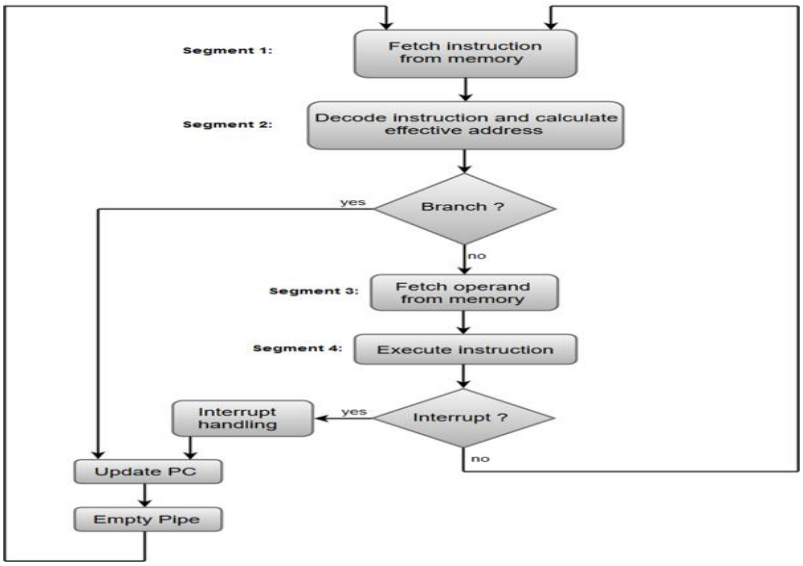


Figure 3.1: Flowchart of Instruction Pipeline

- Figure 3.1 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.
- 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.
- Figure 3.2 shows the operation of a instruction pipeline with the help of a four segment pipeline.

- The instruction cycle is divided in four segments or phases:
  - **Segment 1:**
    FI- Fetches an instruction from memory
  - **Segment 2:**
    DA-The instruction fetched from memory is decoded in the second segment, and the effective address is calculated in a separate arithmetic circuit.
  - **Segment 3:**
    FO- An operand from memory is fetched in the third segment.
  - **Segment 4:**
    EX-The instructions are finally executed in the last segment of the pipeline organization.

Instruction
Branch

Stage	1	2	3	4	5	6	7	8	9	10	11	12	13
1	FI	DA	FO	EX									
2		FI	DA	FO	EX								
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 3.2: Four Segment Instruction Pipeline

- Here the instruction is fetched on first clock cycle in segment 1.
- Now it is decoded in next clock cycle, then operands are fetched and finally the instruction is executed.

- The fetch and decode phase overlap due to pipelining i.e by the time the first instruction is being decoded, next instruction is fetched by the pipeline.
- In case of third instruction, it is a branch instruction.
- Here when it is being decoded, 4th instruction is fetched simultaneously.
- But as it is a branch instruction it may point to some other instruction when it is decoded.
- Thus fourth instruction is kept on hold until the branch instruction is executed.
- When it gets executed then the fourth instruction is copied back and the other phases continue as usual.

COMPUTER ORGANIZATION AND ARCHITECTURE
UNIT –V
OVERVIEW OF PIPELINIG TECHNIQUE & PROCESSOR ARCHITECTURE
TOPIC- RISC PIPELINE

RISC Pipeline

- Among the characteristics attributed to RISC 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 sub operations, 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 there 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, and 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.
    1. These instructions use register indirect addressing. They usually need three or four stages in the pipeline.
    2. 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.
    3. 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.
    1. 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.
    2. RISC can achieve pipeline segments, requiring just one clock cycle.

Example: Three-Segment Instruction Pipeline

- A typical set of instructions for a RISC processor consists of three types of instructions:

1. The data manipulation instructions: operate on data in processor registers
2. The data transfer instructions
3. The program control instructions

Consider the hardware operation for such a computer.

- The *control section* fetches the instruction from program memory into an instruction register.

1. 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 sub operations and implemented in three segments:

1. I:Instruction fetch
   - Fetches the instruction from program memory
2. 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.
3. 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:

1. LOAD:R1←M[address 1]
2. LOAD:R2←M[address 2]
3. ADD: R3 ←R1 +R2
4. STORE:M[address3]←R3

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

• Figure9-9(b) shows the same program with a no-op instruction inserted after the load to R2 instruction.

• The data is loaded intoR2 in clock cycle4.

• 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.

• 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.

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

9 (a) Pipeline timing with data conflict

	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

9 (b) Pipeline timing with delayed load

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.

1. Load from memory to R1
2. Increment R2
3. Add R3 to R4
4. Subtract R5 from R6
5. Branch to address X

- In Fig. 9-10(a) the compiler inserts two no-op instructions after the branch.
- The branch address X is transferred to PC in clock cycle 7.
- The fetching of the instruction at X is delayed by two clock cycles by the no-op instructions.
- The instruction at X starts the fetch phase at clock cycle 8 after the program counter PC has been updated.
- The program inFig.9-10(b) is rearranged by placing the add and subtract instructions after the branch instruction instead of before as in the original program.
- PC is updated to the value of X in clock cycle 5, but add and subtract instructions are fetched from memory and executed in the proper sequence.
- In other words, if the load instruction is at address 101 and X is equal to 350, the branch instruction is fetched from address 103.
- The add instruction is fetched from address 104 and executed in clock cycle 6. The subtract instruction is fetched from address 105and executed in clock cycle 7.
- Since the value of X is transferred to PC with clock cycle 5 in the E segment, the instruction fetched from memory at clock cycle6 is from address350, which is the instruction at the branch address.

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

10 (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

10 (b) Rearranging the instructions

COMPUTER ORGANIZATION AND ARCHITECTURE
UNIT –V
OVERVIEW OF PIPELINIG TECHNIQUE & PROCESSOR ARCHITECTURE
TOPIC- DESIGN ISSUES: HAZARDS, DATA HAZARDS

Hazards (or) Pipeline Hazards

Definition

- Hazard is problems with the instruction pipeline in central processing unit (CPU) that potentially result in incorrect computation. (or)
- Any situation or a condition that causes a pipeline to stall is known as a hazard.
- Pipeline Hazards are situations that prevent the next instruction in the instruction stream from executing during its designated clock cycle.
- The instruction is said to be stalled.
- When an instruction is stalled, all instructions later in the pipeline after the stalled instruction are also stalled.
- Instructions earlier than the stalled instruction can continue but no new instructions can be fetched.

Types of Hazards

They are typically three types of hazards:

1. Data Hazards.
2. Structural Hazards.
3. Control (or) Branch (or) Instruction Hazards.

Data Hazards

- Data Hazard occurs when instructions that exhibit data dependency in different stages of a pipeline. (or)
- Data hazard is a situation in which the pipeline is stalled because the data to be operated on is delayed for some reason.
- For example, stage E in the four stage pipeline in Figure 6.1 is responsible for arithmetic and logic operations such as divide, may require more time to complete in which I2 requires three cycles to complete from cycle 4 through cycle 6.
- Thus in cycles 5 and 6, the write stage must be told to do nothing, because it has no data to work with.
- Meanwhile, the information in buffer B2 must remain intact until the Execute stage has completed its operation.
- This means that stage 2 and stage 1 are blocked from accepting new instructions because the information in B1 cannot be overwritten.
- Thus, steps D4 and F5 must be postponed.
- Pipelined operation is said to have stalled for two clock cycles which leads to hazard.
- A data hazard is any condition in which either the source or destination operands of an instruction are not available at the time expected in the pipeline.

• As a result some operation has to be delayed, and the pipeline stalls.

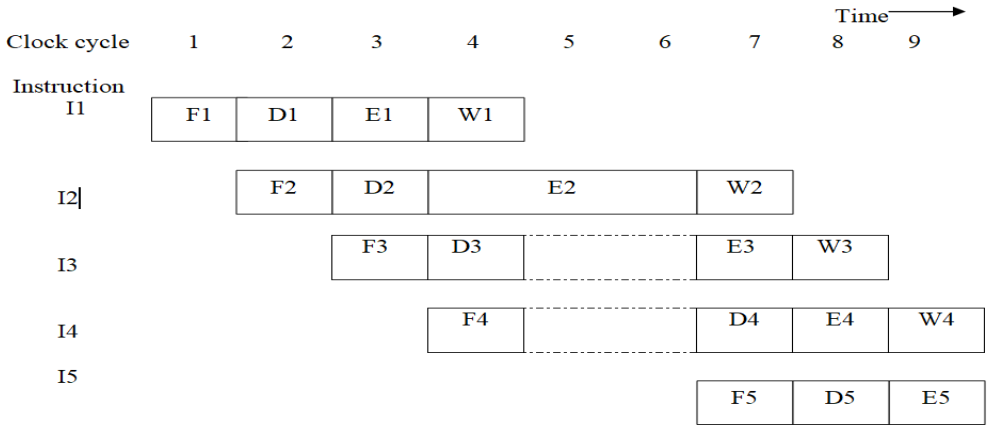


Figure 6.1: Effect of execution operation taking more than one clock cycle.

Examples of Data Hazards

Example 1:

- Consider a program that contains two instructions I_1 followed by I_2.
- When this program is executed in a pipeline, the execution of I_2 can begin before the execution of I_1 is completed.
- The results generated by I_1 may not be available for use by I_2.
- We must ensure that the result of a sequential execution of instructions is identical when same instructions are executed concurrently.
- Consider the two operations: A=5.

  A ← 3 + A

  B ← 4 * A
- When these operations are performed in order given the result is B= 32.
- But if they are performed concurrently, the value of A used in computing B would be original value 5, leading to incorrect results.
- If these two operations are performed by instructions in a program, then the instructions must be executed one after the other, because the data used in second instruction depends on result of first instruction.
- When two operations depend on each other, they must be performed sequentially in the correct order.
- Consider the other two operations:

  A ← 5 * C

  B ← 20+C

These operations can be performed concurrently, because the operations are independent.

Example 2: consider two instructions:

Mul R2, R3, R4
Add R5, R4, R6.

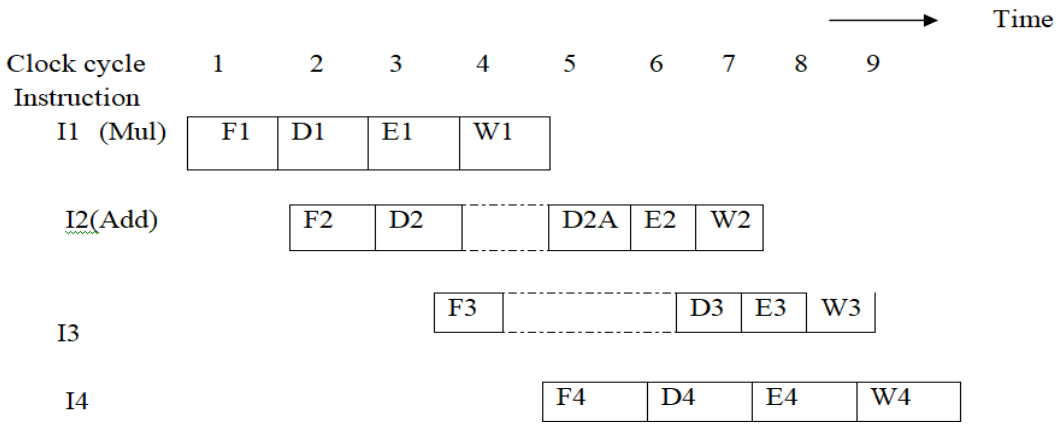


Figure 6.2: Pipeline stalled by data dependency between D2 and W1.

- In the above Figure 6.2, the result of multiply is placed into register R4, which in turn is one of the two source operands of Add instruction.
- Assume that multiply operation takes one clock cycle to complete execution.
- As decode unit decodes the Add instruction in cycle 3, it realizes that R4 is used as a source operand.
- Hence, the D step of that instruction cannot be completed until the W step of multiply instruction has been completed.
- Completion of Step D_2 must be delayed to clock cycle 5.
- Instruction I_3 is fetched in cycle 3, but its decoding must be delayed because the step D_3 cannot precede D_2.
- Hence, pipelined execution is stalled for two cycles.

Types of Data Hazards

There are three situations in which a data hazard can occur:

1. Read after write (RAW), true dependency (or) flow dependency.
2. Write after read (WAR), false dependency (or) anti-dependencies.
3. Write after write (WAW), output dependency.

- Consider two instructions i and j, with i occurring before j in program order.

Read After Write (RAW): (j tries to read a source before i writes to it)

- A read after write (RAW) data hazard refers to a situation where an instruction refers to a result that has not yet been calculated or retrieved.
- This can occur because even though an instruction is executed after a previous instruction, the previous instruction has not been completely processed through the pipeline.

Example: $\mathbf{I_1: R_2 \leftarrow R_1 + R_3}$

$\mathbf{I_2: R_4 \leftarrow R_2 + R_3}$

- The first instruction is calculating a value to be saved in register 2, and the second instruction is going to use this value to compute a result for register 4.

- However, in a pipeline, when we fetch the operands for the 2nd operation, the results from the first will not yet have been saved, and hence we have a data dependency.
- There is a data dependency with instruction 2, as it is dependent on the completion of instruction 1.

Write After Read (WAR):(j tries to write a destination before it is read by i).

- A write after read (WAR) data hazard represents a problem with concurrent execution.

Example: $I_1: R_4 \leftarrow R_1 + R_3$

$I_2: R_3 \leftarrow R_1 + R_2$

- If in a situation that there is a chance that i2 may be completed before i1 (i.e. with concurrent execution)
- we must ensure that we do not store the result of register 3 before i1 had a chance to fetch the operands.

Write After Write (WAW): (j tries to write an operand before it is written by i).

- A write after write (WAW) data hazard may occur in a concurrent execution environment.

For example:

$I_1: R_2 \leftarrow R_1 + R_2$

$I_2: R_2 \leftarrow R_4 + R_7$

- We must delay the WB (Write Back) of i2 until the write back of i1 is completed.

COMPUTER ORGANIZATION AND ARCHITECTURE
UNIT –V
OVERVIEW OF PIPELINIG TECHNIQUE & PROCESSOR ARCHITECTURE
TOPIC- DESIGN ISSUES: RESOLVING DATA HAZARDS

Resolving Data Hazards

- There are several solutions and algorithms used to resolve data hazards:

1.Pipeline Bubbling

- Pipeline Bubble also known as pipeline break or a pipeline stall is a software method for preventing data hazards from occurring.
- When instructions are fetched, control logic determines whether a hazard could occur.
- If it is true, then the control logic inserts a NOP instruction (No operation) into the pipeline.
- Thus, before the next instruction (which could cause hazard) is executed, the previous one will have sufficient time to complete and prevent hazard.
- If the responsibility of detecting dependencies is left entirely to the software, the compiler must insert the NOP instruction to obtain correct results.
- If the number of NOPs is equal to number of stages in the pipeline, the processor has been cleared of all instructions and can proceed free from hazards.
- This is called flushing the pipeline. All forms of stalling introduce a delay before the processor can resume execution.

Example: **I_1: Mul R_2, R_3, R_4**
NOP
NOP
I_2: Add R_5, R_4, R_6

Advantages

- Inserting NOP Instructions to the compiler leads to simpler hardware

Disadvantages

- Inserting NOP instructions leads to larger code size

2.Operand/ Data forwarding Technique

- The Operand forwarding technique is the hardware based method used to resolve data hazards, introduced by the sequence of instructions.
- The data hazard in the Figure 6.3 arises when I2 is waiting for data to be written in the register file.
- However, these data are available at the output of ALU once the Execute stage completes step E_1.
- So, the delay can be reduced or possibly eliminated, if we arrange the hardware such that the result of I_1 is forwarded directly for use in step E_2 which is known as Data Forwarding.

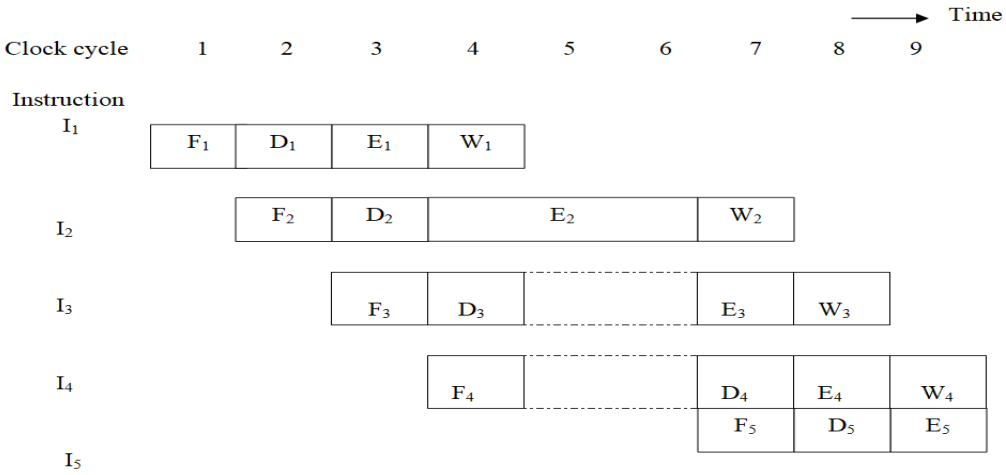


Figure 6.3: Data hazard situation between the instructions.

The data forwarding mechanism works as follows:

- Forwarding is implemented by feeding back the output of previous instruction into the next stages of the pipeline as soon as the output of that instruction is available.

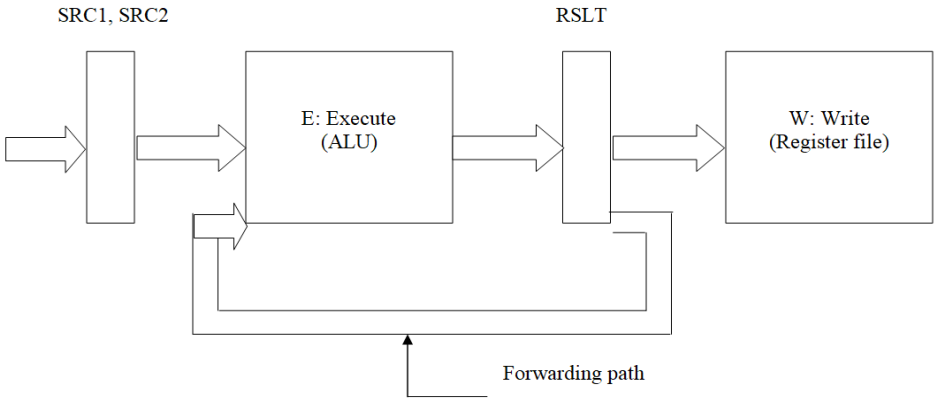


Figure 6.4: Effect of Data forwarding on the processor

COMPUTER ORGANIZATION AND ARCHITECTURE
UNIT –V
OVERVIEW OF PIPELINIG TECHNIQUE & PROCESSOR ARCHITECTURE
TOPIC- DESIGN ISSUES: STRUCTURAL HAZARDS

Structural Hazards

- Structural hazards occur due to resource conflicts.
- This is a situation when two instructions require the use of a given hardware resource at the same time.
- The most common case in which this hazard may arise is in access to memory.
- One instruction may need to access the memory as part of Execute or Write stage while another instruction is being fetched.
- If instruction and data reside in the same cache unit, only one instruction can proceed and other instruction is delayed.
- Many processors use separate instruction and data caches to avoid this delay.

Examples of structural Hazards

Example 1: Load X (R_1), R_2.

- In Figure 6.7, the memory address, X+ [R1], is computed in Step E2 in cycle 4, then memory access takes place in cycle 5.
- The operand read from memory is written into register R2 in cycle 6.
- This means that execution step of this instruction takes two clock cycles (cycles 4 and 5).
- t causes the pipeline to stall for one cycle, because both I_2 and I_3 require access to the register file in cycle 6.

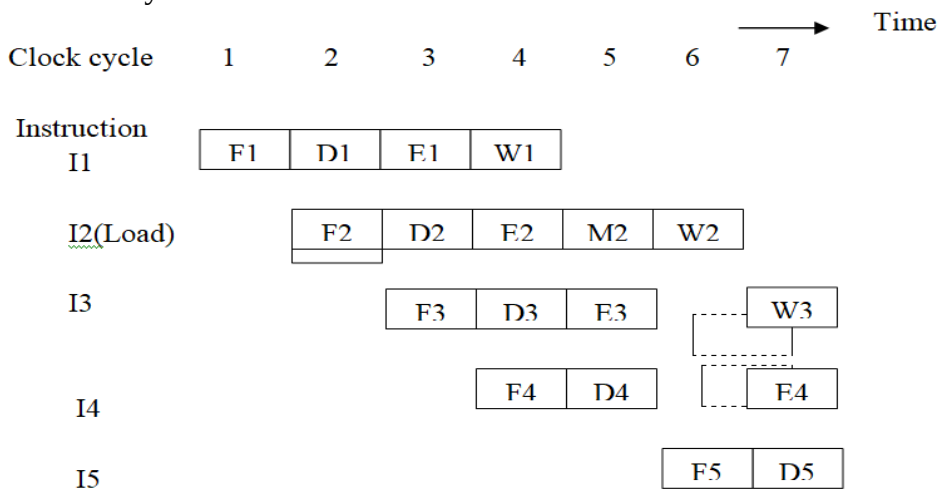


Figure 6.7: Effect of a load Instruction on Pipeline.

- The register file cannot handle two operations at once.
- If the register file had two input ports, that is, if it allowed two simultaneous write operations, the pipeline would not be stalled.

- Structural hazards are avoided by providing sufficient hardware resources on the processor chip.

- Pipelining does not result in individual instructions being executed faster rather it is the throughput that increases where throughput is measured by rate at which instruction execution is completed.

- At any time one of the stages in the pipeline cannot complete its operation in one clock cycle, the pipeline stalls and performance of the pipeline degrades.

Other examples of Structural Hazards:

- **Example 2:**
  A machine has shared a single-memory pipeline for data and instructions. As a result, when an instruction contains a data-memory reference (load), it will conflict with the instruction reference for a later instruction (instruction 3).

Clock cycle number								
Instr	1	2	3	4	5	6	7	8
Load	IF	ID	EX	MEM	WB			
Instr 1		IF	ID	EX	MEM	WB		
Instr 2			IF	ID	EX	MEM	WB	
Instr 3				IF	ID	EX	MEM	WB

Structural hazard situation between Load and Instr 3.

- To resolve this, we stall the pipeline for one clock cycle when a data-memory access occurs.
- The effect of the stall is actually to occupy the resources for that instruction slot.

Clock cycle number									
Instr	1	2	3	4	5	6	7	8	9
Load	IF	ID	EX	**MEM**	WB				
Instr 1		IF	ID	EX	MEM	WB			
Instr 2			IF	ID	EX	MEM	WB		
Stall				Bubble	Bubble	Bubble	Bubble	bubble	
Instr 3					**IF**	ID	EX	MEM	WB

Resolving structural hazard by Bubbling/stalling.

- Instruction 1 assumed not to be data-memory reference (load or store), otherwise Instruction 3 cannot start execution for the same reason as above.
- To simplify it is also commonly shown like this:

Clock cycle number									
Instr	1	2	3	4	5	6	7	8	9
Load	IF	ID	EX	MEM	WB				
Instr 1		IF	ID	EX	MEM	WB			
Instr 2			IF	ID	EX	MEM	WB		
Instr 3				Stall	IF	ID	EX	MEM	WB

Stalling of Instr 3 to resolve structural hazard situations.

- Introducing stalls degrades performance .
- Designer allow structural hazards for two reasons:
- **To reduce cost**. For example, machines that support both an instruction and a cache access every cycle (to prevent the structural hazard of the above example) require at least twice as much total memory.
- **To reduce the latency of the unit.** The shorter latency comes from the lack of pipeline registers that introduce overhead.

COMPUTER ORGANIZATION AND ARCHITECTURE
UNIT –V
OVERVIEW OF PIPELINIG TECHNIQUE & PROCESSOR ARCHITECTURE
TOPIC- DESIGN ISSUES: BRANCH (OR) CONTROL (OR) INSTRUCTION HAZARDS

Branch Hazards: Introduction

- Branch hazards (also known as control or instruction hazards) occur with branches.
- On many instruction pipeline micro architectures, the processor will not know the outcome of the branch when it needs to insert a new instruction into the pipeline (normally the *fetch* stage).
- Branch hazards occur when the processor is told to branch i.e., if a certain condition is true, then jump from one part of the instruction stream to another - not necessarily to the next instruction sequentially.
- In such a case, the processor cannot tell in advance whether it should process the next instruction (when it may instead have to move to a distant instruction).
- This can result in the processor doing unwanted actions.

Unconditional branches

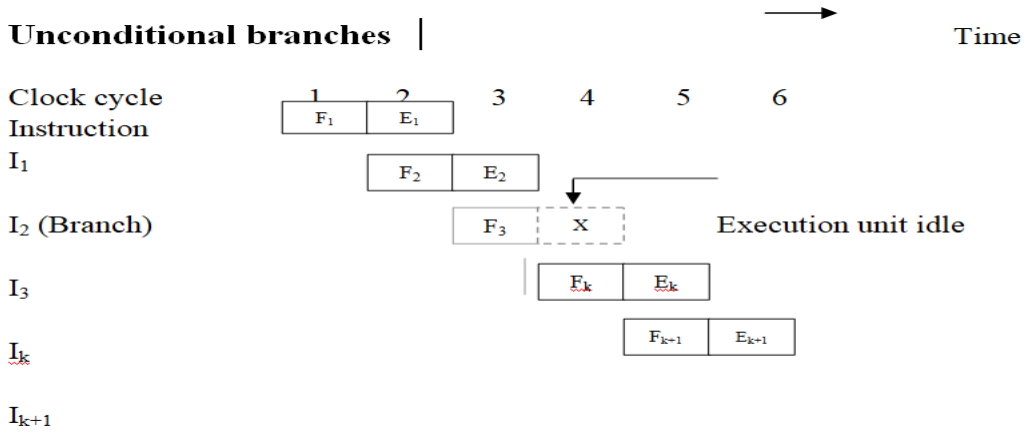


Figure : An idle cycle caused by a branch instruction.

- In the above Figure, Instructions I_1 to I_3 are stored at successive memory addresses and I_2 is a branch instruction.
- Let the branch target be I_k. In clock cycle 3, the fetch operation for instruction I_3 is in progress at the same time that the branch instruction is being decoded and target address computed.
- In clock cycle 4, processor must discard I_3, which has been incorrectly fetched, and fetch instruction I_k.
- In meantime, the hardware unit is responsible for the Execute step (E) must be told to do nothing during that clock period. Thus, the pipeline is stalled for one clock cycle.

• The time lost as a result of a branch instruction is often referred to as branch penalty.
• Here, the branch penalty is one clock cycle. For a longer pipeline, the branch penalty may be higher.

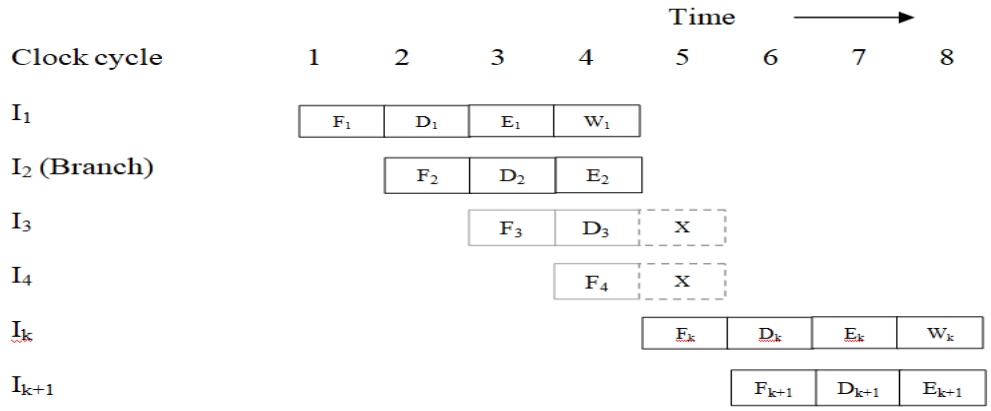


Figure : Branch address computed in Execute stage.

• Branch address is computed in step E_2.
• Instructions I_3 and I_4 must be discarded and the target instruction I_k is fetched in clock cycle 5.
• The branch penalty is two clock cycles.
• Reducing the branch penalty requires the branch instructions to be computed earlier in the pipeline.
• The branch penalty can be minimized by finding the branch instructions either in fetch stage or the decode stage of the pipeline rather than in the execute stage.

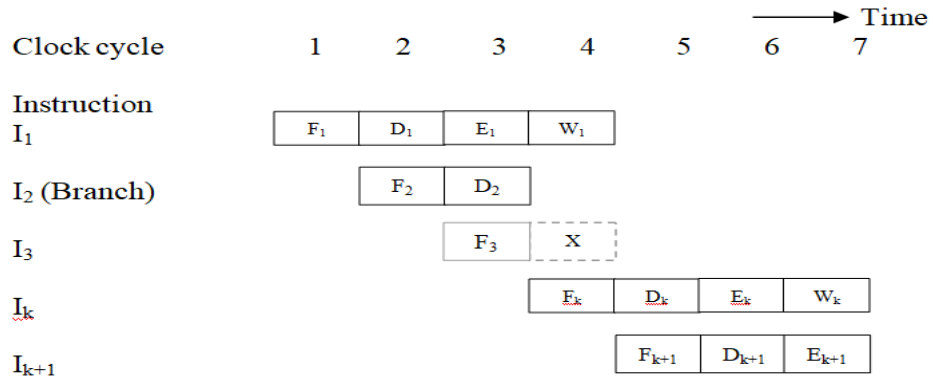


Figure : Branch address computed in Decode stage

Instruction Queue and Prefetching

• Either a cache misses or branch instruction stalls the pipeline for one or more clock cycles.
• To reduce this effect many processors use fetch units that can fetch the instructions before they are needed and put them in a queue.

• The fetch unit has a dedicated hardware to identify the branch instructions & compute the branch target address as quickly as possible after an instruction is fetched.

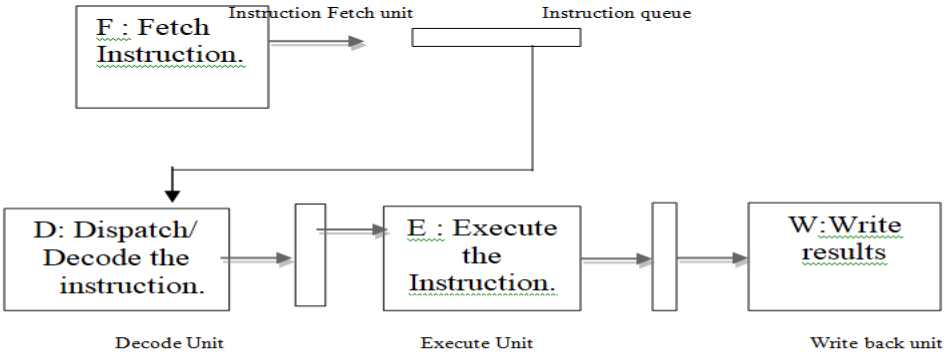


Figure : Use of instruction queue in the hardware organization.

• A separate dispatch unit takes the instruction from the front of the queue and sends them to the execution units. It also performs decoding function.

• To be effective, the fetch unit must have a sufficient decoding and processing capabilities to recognize.

• When the dispatch unit is not able to issue the instructions for execution because of data hazards the fetch unit continues to fetch the instructions and add them to the queue.

• If there is a delay in fetching because of branch or cache miss, dispatch unit continues to dispatch the instructions from the instruction queue.

• Having instruction queue is beneficial with cache misses. When a cache miss occurs, dispatch unit continues to send the instructions for execution as long as the queue is not empty.

• Meanwhile the desired cache block is read from the main memory

• We assume that initially the queue contains one instruction. Every fetch operation adds one instruction to the queue and every dispatch operation reduces the queue length by one.

Conditional Branches

• Conditional branch introduces the added hazard caused by dependency of the branch condition on the result of preceding instruction.

• The decision to branch cannot be made until the execution of that instruction has been completed.

Delayed Branch

• In Figure 6.15, the processor fetches the instruction I_3 before it determines whether the current instruction I_2 is a branch instruction.

• When the execution of I_2 is completed and a branch is made, the processor must discard I_3 and fetch the instruction at the branch target.

• The location following a branch instruction is called a branch delay slot.

• There may be more than one delay slot depending on the time it takes to execute a branch instruction.

• The instruction in the delay slots are always fetched and at least partially executed before the branch decision is made and the branch target address is computed.

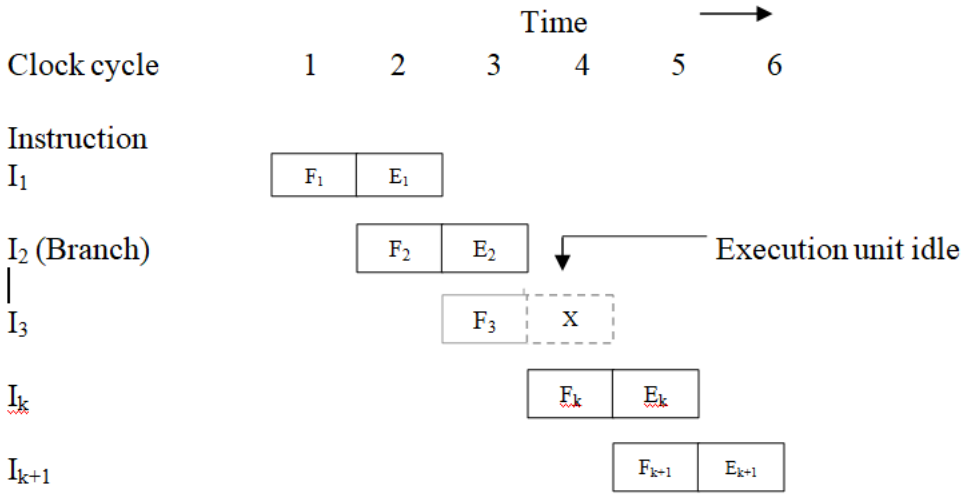


Figure 6.15: An idle cycle caused by a branch instruction.

- A technique called delay branching can minimize the penalty incurred as a result of conditional branch instructions.
- The instructions in the delay slots are always fetched.
- We would like to arrange them to be fully executed whether or not the branch is taken.
- The objective is to place useful instructions in these slots.
- If no useful instructions can be placed in the delay slots, these slots must be filled with NOP instruction.

Controlling Branch Hazards

Software solutions include the following:

1. Branch spreading.
2. Scheduling the branch delay slot.
3. Software (static) Branch prediction: Use branch prediction and essentially guesstimate which instructions to insert, in which case a pipeline bubble will only be needed in the case of an incorrect prediction.
4. Trace scheduling.
5. Loop unrolling.
6. Pipeline Bubbling: insert a pipeline bubble, guaranteed to increase latency.
7. Scheduling across branches.

Hardware solutions include the following:

1. The control section can stop the flow of instructions (i.e., halt the pipeline) before fetching the next instruction, until the preceding operation is finished and the results are known. No matter whether the branch is taken or not, this always causes a delay.

2. The second way, called branch prediction, makes a guess as to which way the branch is going to go before it is taken, follows this path, and continues preparing instructions; if the guess later proves to be wrong, the pipeline must be flushed clean and started again with the correct instruction. has been implemented at either the fetch stage or the decode stage).

3.In a third alternative, called prefetching of multiple paths, the control section can fetch the instructions immediately following the conditional branch instruction as well as instructions from the target address of the branch simultaneously, and when the ALU finally figures out the branch conditions, then decide which one of the two prefetched groups of prepared instructions to use.

COMPUTER ORGANIZATION AND ARCHITECTURE
UNIT –V
OVERVIEW OF PIPELINIG TECHNIQUE & PROCESSOR ARCHITECTURE
TOPIC- BRANCH PREDICTION

Branch Prediction

- Branch Prediction is a technique used for reducing branch penalty associated with conditional branches

Introduction to Branch prediction

- Prediction techniques can be used to check or predict whether a branch will be taken or not taken.
- Forecasting the outcome of a branch ahead of time is essential in order to improve the performance of a processor.
- There are two ways in which a branch can be predicted:

a. Static Branch Prediction.
b. Dynamic Branch Prediction.

Static Branch Prediction

- Static prediction is the simplest branch prediction technique.
- It does not rely on information about the dynamic history of the code executing. Instead it predicts the outcome of a branch based solely on the branch instruction.
- The direction of each branch is predicted before a program runs.
- It predicts the branch in the ID stage of the pipeline. Predictions are either based on compile time heuristics or profiling technique.
- Static branch prediction using compile time heuristics involves making a prediction at the compile time. Branch Prediction using compile time heuristics is based on two approaches:

1.Predict Not Taken.
2. Predict Taken.

Predict Not Taken Approach

- The simplest form of static branch prediction using heuristic approach (Predict Not Taken) is to ignore the presence of a branch and assume that the branch will not take place and continue to fetch the instructions in a sequential address order.
- Until the branch is evaluated, execution of instructions is done in a speculative basis.
- Speculative execution means that instructions are executed before the processor is certain that there are in a correct execution sequence.
- No processor registers or memory locations are updated until it is confirmed that these instructions are indeed be executed.
- If the branch decision is made, instruction and their associated data in the execution unit must be purged and correct instructions are fetched and executed.

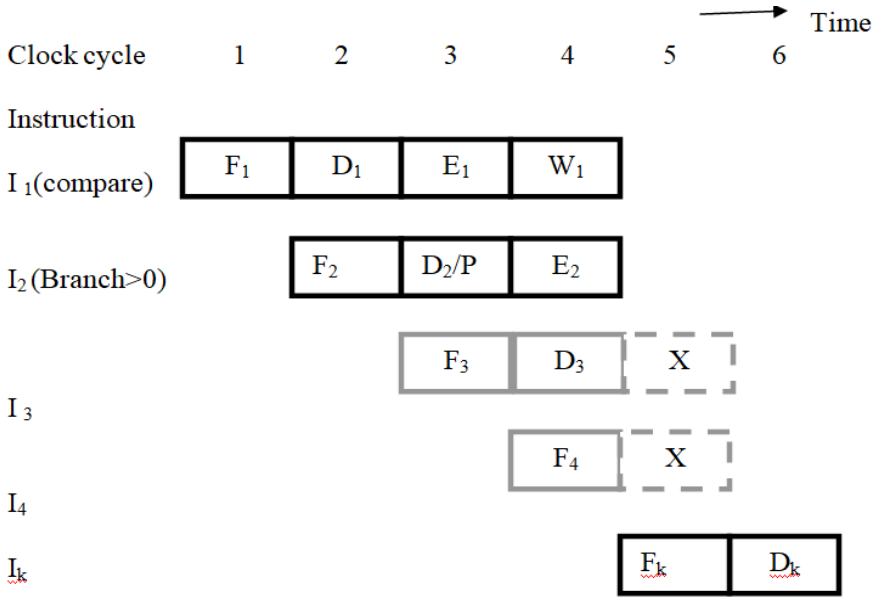


Figure : Timing diagram when a branch decision has been incorrectly predicted as not taken.

- In the above figure, the prediction takes place in cycle 3, while instruction I_3 is being fetched. The fetch unit predicts that the branch will not be taken, and continues to fetch the instructions I_3 and I_4 in a sequential order.
- The results of compare operation are available at end of cycle 3, assuming that these results are forwarded directly to the instruction fetch unit branch condition is evaluated in cycle 4.
- Here instruction fetch realizes that prediction was incorrect and the two instructions in the execution pipe are purged and new instruction I is fetched from branch target address in clock cycle 5.

Predict Taken Approach

- The static branch prediction using heuristic approach (Predict Taken) is a correctly predicted branch will always incur a penalty of at least one clock cycle while the branch target address is being computed.
- Predict Not Taken approach is slightly better than predict taken approach.
- Predict taken scheme is more complex to implement than Predict Not Taken scheme.
- Static branch prediction using profiling technique is based on the profile information that has been obtained from the previous runs of a program.

Dynamic Branch Prediction

- Dynamic Brach Prediction reduces the branch penalty under hardware control.
- The direction of each branch is predicted by recording the information in the hardware of the past branch history during the program execution and is therefore done at runtime.
- Here, the prediction is done in the IF stage of the pipeline.
- The simplest dynamic prediction scheme is a branch prediction buffer or branch history table.

- Branch prediction buffer is a small fast memory which contains history information of the previous outcomes of the branch as a prediction field.
- Branch target can then be accessed as soon as the branch target address is computed but before the branch condition is available.

2-State algorithm

- Suppose the algorithm is started in LNT state.
- When the branch is executed and if the branch is taken, it moves from LNT to LT. Otherwise, it is in LNT state.
- The next time when the same instruction is encountered and when the branch is predicted as taken, the corresponding machine is in LT state. Otherwise, it moves to LNT state.
- Suppose the algorithm is started in LT state.
- When the branch is executed and if the branch is not taken, it moves to LNT state.
- The next time when the same instruction is encountered and when the branch is predicted as taken, the corresponding machine is in LT state.
- This scheme requires one bit of the history information for each branch instruction.
- Better performance can be achieved by keeping more information about the execution history.

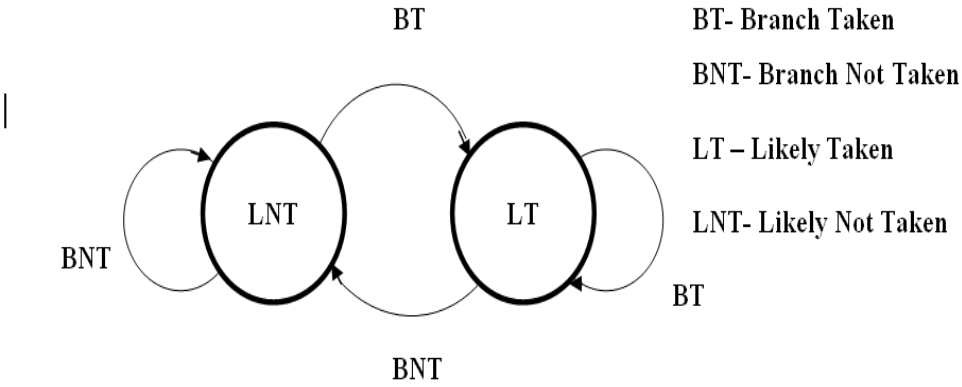
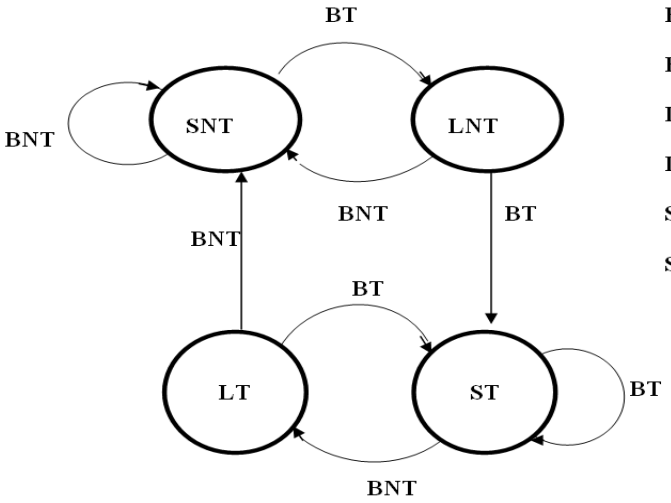


Figure : 2 -State algorithm.

4-State algorithm

- Suppose the algorithm is started in LNT state and the branch is executed and if the branch is taken, it moves from LNT to ST else it moves to SNT.
- Suppose the algorithm is started in SNT state and when it is executed and if the branch is taken it moves from SNT to LNT else it will be SNT state.
- When the branch is in LNT state and when it is executed again and if the branch is taken (i.e.; the prediction is incorrect twice) it moves from LNT to ST.
- Suppose the algorithm is started in LT state and the branch is executed and if the branch is taken, it moves from LT to ST else it moves to SNT.
- Suppose the algorithm is started in ST state and when it is executed and if the branch is taken (the prediction is incorrect) it will be ST state else it moves to LT state.



BT – Branch Taken

BNT- Branch Not Taken

LT- Likely Taken

LNT- Likely Not Taken

SNT- Strongly Not Taken

ST- Strongly Taken

Figure : 4-State algorithm.

COMPUTER ORGANIZATION AND ARCHITECTURE
UNIT –V
OVERVIEW OF PIPELINIG TECHNIQUE & PROCESSOR ARCHITECTURE
TOPIC- REDUCED INSTRUCTION SET COMPUTER (RISC) & COMPLEX INSTRUCTION SET COMPUTRER(CISC)

Reduced Instruction Set Architecture (RISC)

- It stands for Reduced Instruction Set Computer.
- It is a type of microprocessor architecture that uses a small set of instructions of uniform length.
- These are simple instructions that are generally executed in one clock cycle.
- RISC chips are relatively simple to design and inexpensive.
- The main idea behind this is to simplify hardware by using an instruction set composed of a few basic steps for loading, evaluating, and storing operations just like a load command will load data, a store command will store the data.
- RISC architectures are generally well-suited for high-throughput applications like scientific computing and multimedia processing.
- Their streamlined instructions and efficient use of registers contribute to faster execution, which is crucial for data-intensive tasks.
- **Examples:** SPARC, POWER PC, etc.

Characteristics of RISC

- Simpler instruction, hence simple instruction decoding.
- Instruction comes undersize of one word.
- Instruction takes a single clock cycle to get executed.
- More general-purpose registers.
- Simple Addressing Modes.
- Fewer Data types.
- A pipeline can be achieved.

Advantages of RISC

- **Simpler instructions:** RISC processors use a smaller set of simple instructions, which makes them easier to decode and execute quickly. This results in faster processing times.
- **Faster execution:** Because RISC processors have a simpler instruction set, they can execute instructions faster than CISC processors.
- **Lower power consumption:** RISC processors consume less power than CISC processors, making them ideal for portable devices.

Disadvantages of RISC

- **More instructions required:** RISC processors require more instructions to perform complex tasks than CISC processors.

- **Increased memory usage:** RISC processors require more memory to store the additional instructions needed to perform complex tasks.
- **Higher cost:** Developing and manufacturing RISC processors can be more expensive than CISC processors.

Complex Instruction Set Architecture (CISC)

- It stands for Complex Instruction Set Computer.
- These processors offer hundreds of instructions of variable sizes.
- CISC architecture includes a complete set of special-purpose circuits that carry out these instructions at a very high speed.
- These instructions interact with memory by using complex addressing modes.
- CISC processors reduce the program size and hence lesser number of memory cycles are required to execute the programs. This increases the overall speed of execution.
- The main idea is that a single instruction will do all loading, evaluating, and storing operations just like a multiplication command will do stuff like loading data, evaluating, and storing it, hence it's complex.
- **Examples:** Intel architecture, AMD

Characteristics of CISC

- Complex instruction, hence complex instruction decoding.
- Instructions are larger than one-word size.
- Instruction may take more than a single clock cycle to get executed.
- Less number of general-purpose registers as operations get performed in memory itself.
- Complex Addressing Modes.
- More Data types.

Advantages of CISC

- **Reduced code size:** CISC processors use complex instructions that can perform multiple operations, reducing the amount of code needed to perform a task.
- **More memory efficient:** Because CISC instructions are more complex, they require fewer instructions to perform complex tasks, which can result in more memory-efficient code.
- **Widely used:** CISC processors have been in use for a longer time than RISC processors, so they have a larger user base and more available software.

Disadvantages of CISC

- **Slower execution:** CISC processors take longer to execute instructions because they have more complex instructions and need more time to decode them.
- **More complex design:** CISC processors have more complex instruction sets, which makes them more difficult to design and manufacture.
- **Higher power consumption:** CISC processors consume more power than RISC processors because of their more complex instruction sets.

Key Difference between RISC and CISC Processor

• Instructions are easier to decode in RISC than in CISC, which has a more complex decoding process.

• Calculations in CISC require external memory, but they are not necessary for RISC.

• While CISC only has a single register set, RISC has numerous register sets.

• The execution time of a RISC computer is very low compared to a CISC computer, which is very high.

• RISC architectures emphasize simpler instructions, shorter execution times, and efficient use of registers.

• CISC architectures offer complex instructions, potentially leading to fewer instructions overall but with longer execution times.

• RISC architectures often follow a load-store model, while CISC architectures allow direct memory access instructions.

COMPUTER ORGANIZATION AND ARCHITECTURE
UNIT –V
OVERVIEW OF PIPELINIG TECHNIQUE & PROCESSOR ARCHITECTURE
TOPIC- EXAMPLE OF RISC & CISC, RISC VS CISC

Example of RISC and CISC

Multiplying Two Numbers in the Memory

- The main memory is divided into locations numbered from (row) 1: (column) 1 to (row) 6: (column) 4.
- The execution unit is responsible for carrying out all the computations.

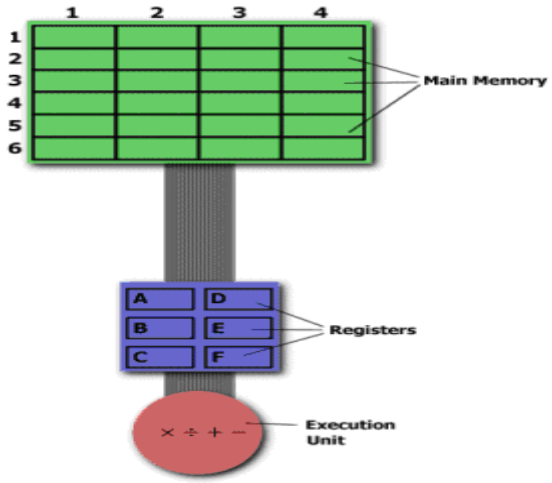


Figure: Multiplying two numbers in the memory

- However, the execution unit can only operate on data that has been loaded into one of the six registers (A, B, C, D, E, or F).
- To find the product of two numbers - one stored in location 2:3 and another stored in location 5:2 - and then store the product back in the location 2:3.

The CISC Approach

- The primary goal of CISC architecture is to complete a task in as few lines of assembly as possible.
- This is achieved by building processor hardware that is capable of understanding and executing a series of operations.
- For this particular task, a CISC processor has a specific instruction (called "MULT"). When executed, this instruction loads the two values into separate registers, multiplies the operands in the execution unit, and then stores the product in the appropriate register.

- The task of multiplying two numbers can be completed with one instruction:

  MULT 2:3, 5:2

- MULT is known as a "complex instruction" ,Which operates directly on the computer's memory banks and does not require the programmer to explicitly call any loading or storing functions.
- One of the primary advantages is that the compiler has to do very little work to translate a high-level language statement into assembly. Because the length of the code is relatively short, very little RAM is required to store instructions. The emphasis is directly into the hardware.

The RISC Approach

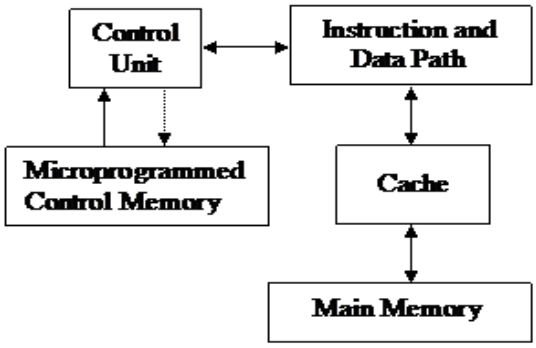
- RISC processors only use simple instructions that can be executed within one clock cycle.
- Thus, the "MULT" command could be divided into three separate commands: "LOAD," which moves data from the memory bank to a register, "PROD," which finds the product of two operands located within the registers, and "STORE," which moves data from a register to the memory banks.

  LOAD A, 2:3
  LOAD B, 5:2
  PROD A, B
  STORE 2:3, A

- Here, there are more lines of code, more RAM is needed to store the assembly level instructions.
- The compiler must also perform more work to convert a high-level language statement into code of this form.
- The advantages of RISC is that because each instruction requires only one clock cycle to execute, the entire program will execute in approximately the same amount of time as the multi-cycle "MULT" command.
- These RISC "reduced instructions" require less transistors of hardware space than the complex instructions, leaving more room for general purpose registers.
- Because all of the instructions execute in a uniform amount of time (i.e. one clock), pipelining is possible.
- Separating the "LOAD" and "STORE" instructions actually reduces the amount of work that the computer must perform.

CISC VERSUS RISC

CISC	RISC
1. Large instruction set.	1. Small instruction set.
2. Emphasis on the hardware.	2. Emphasis on the software.
3. Complex operations.	3. Simple instructions to allow for fast execution (fewer steps).
4. CISC chips have a large amount of different and complex instructions which require multiple cycles.	4. Reduced instructions that take one clock cycle.
5. Instructions are executed one at a time.	5.Uses pipelining to execute instructions
6. Many instructions can reference memory.	6. Only Load and Store instructions can reference memory.
7. Few general registers.	7. Many general register
8. Easy to program, simpler compiler.	8. Complex tasks are left to the compiler to construct from simple operations, with increased compiler complexity and compiling time.
9. Complex addressing modes are infrequently used, and they can always be realized using several simple instructions.	9. Simple addressing modes to allow for fast address computation.
10. CISC requires more transistors. So, harder to design and costly.	10. RISC chips require fewer transistors.
11. It is slower than RISC chips.	11. It is faster than CISC chips.
12. At least 75% of the processor use CISC architecture.	12. RISC architecture is not widely used
13. CISC processor executes microcode instructions.	13. RISC processor has a number of hardwired instructions.
14. CISC processors cannot have a large number of registers.	14. Large number of registers, most of which can be used as general purpose registers.
15. Intel and AMD CPU's are based on CISC architectures.	15. Apple uses RISC chips.

16. Mainly used in normal PC's, Workstations and servers.	16.Mainly used for real time applications.
17. In CISC, software developers do not need to write more lines for the same tasks.	17. RISC puts a greater burden on the software. Software developers need to write more lines for the same tasks.
18. Direct addition between data in two memory locations. Ex.8085.	18. Direct addition is not possible.
19. Pipelining implementation is not easy	19. Pipelining can be implemented easily.
20. CISC approach minimizes the number of instructions per program and increases the number of cycles per instruction.	20. RISC approach maximizes the number of instructions per program and reduces the number of cycles per instruction.
21. **Examples of CISC :** • System/360 • VAX . • PDP-11. • Motorola68000 family. • Intelx86/Pentium CPU's	21. **Examples of RISC:** • IBM 360. • DEC VAX. • Intel80x86 families. • Motorola 68xxx
22. Modern CISC  	22. Traditional RISC