Unit-IV

Overview of Pipelining Techniques: Parallel Processing, Pipelining, Arithmetic Pipeline, Instruction Pipeline, RISC Pipeline, Design Issues, Hazards: Structural Hazards, Data Hazards and Control Hazards, Static Branch Prediction, Dynamic Branch Prediction.

Introduction to Processor Architecture: CISC Characteristics, RISC Characteristics, Differences between CISC and RISC Characteristics, its advantages, disadvantages of CISC over RISC

Parallel Processing

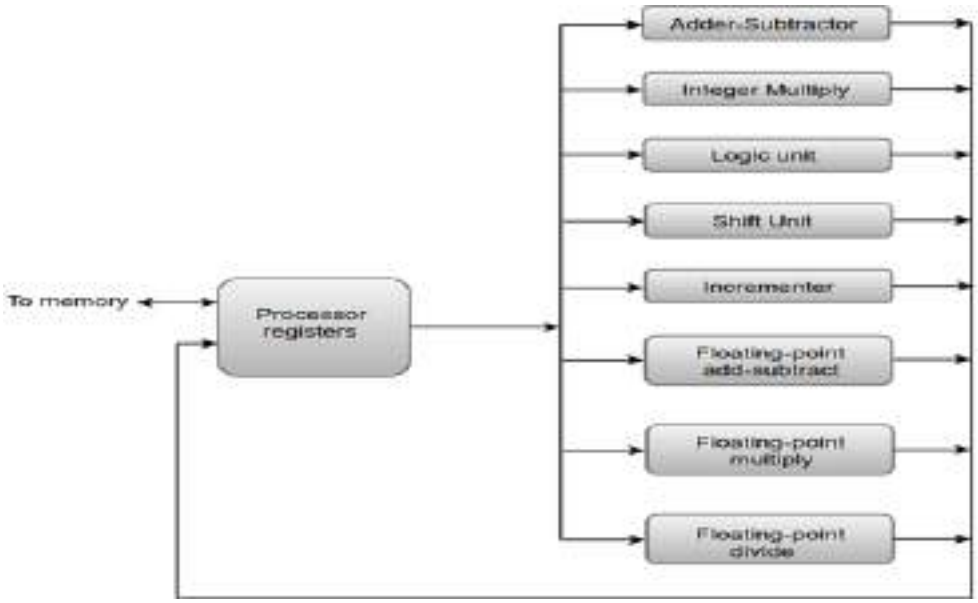
Parallel processing can be described as a class of techniques which enables the system to achieve simultaneous data-processing tasks to increase the computational speed of a computer system.

A parallel processing system can carry out simultaneous data-processing to achieve faster execution time. For instance, while an instruction is being processed in the ALU component of the CPU, the next instruction can be read from memory.

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.

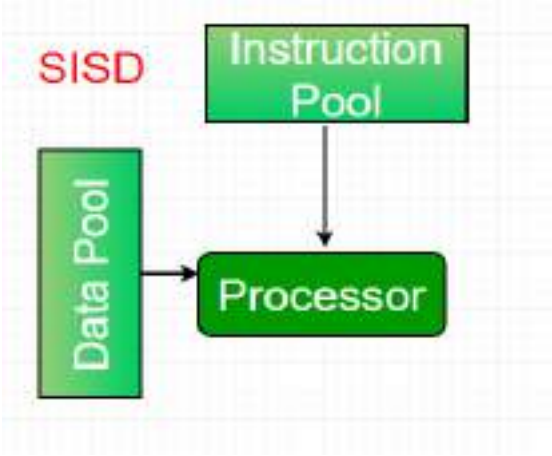
The following diagram shows one possible way of separating the execution unit into eight functional units operating in parallel. The operation performed in each functional unit is indicated in each block of the diagram:



- 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, so one number can be shifted while another number is being incremented.

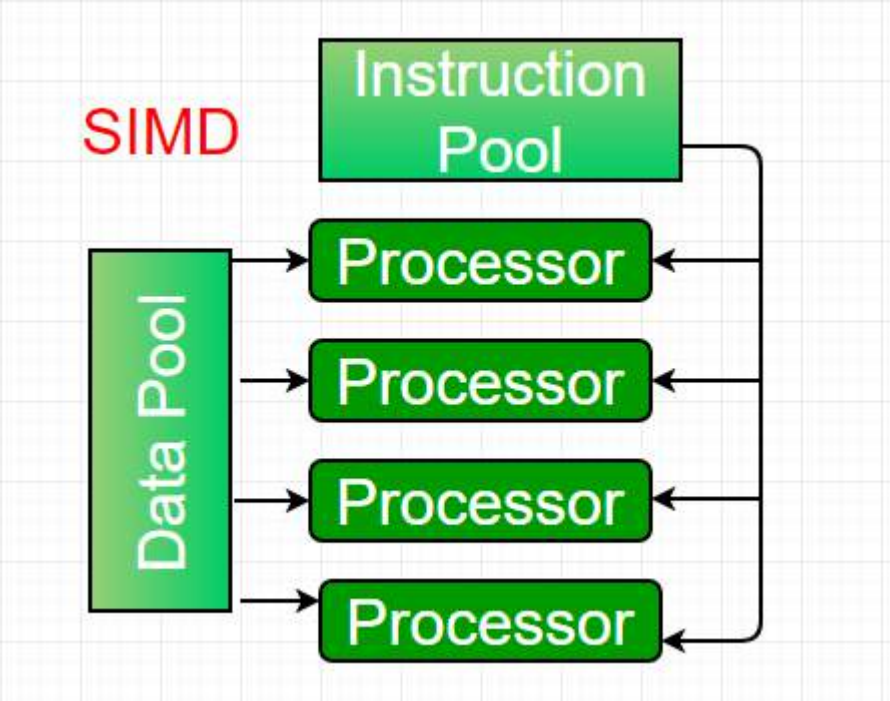
Flynns Classification:

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



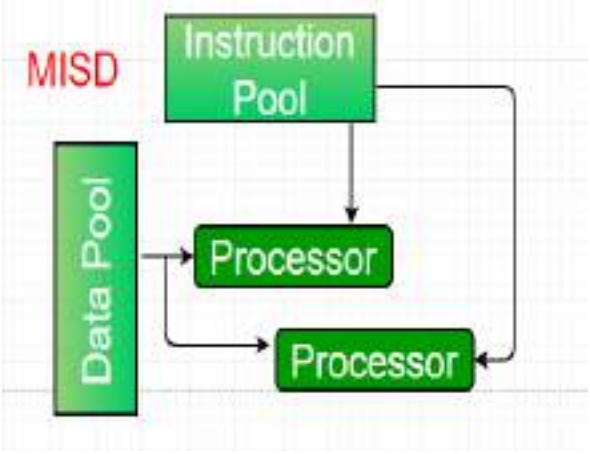
The speed of the processing element in the SISD model is limited(dependent) by the rate at which the computer can transfer information internally. Dominant representative SISD systems are IBM PC, workstations.

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



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

An MISD computing system is a multiprocessor machine capable of executing different instructions on different PEs but all of them operating on the same dataset .



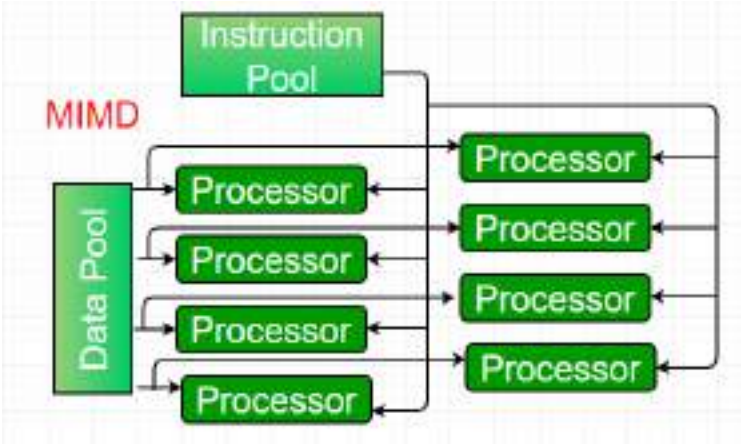
Example Z = sin(x)+cos(x)+tan(x)

The system performs different operations on the same data set. Machines built using the MISD model are not useful in most of the application, a few machines are built, but none of them are available commercially.

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

An MIMD system is a multiprocessor machine which is capable of executing multiple

instructions on multiple data sets. Each PE in the MIMD model has separate instruction and data streams; therefore machines built using this model are capable to any kind of application. Unlike SIMD and MISD machines, PEs in MIMD machines work asynchronously.



MIMD machines are broadly categorized into **shared-memory MIMD** and **distributed-memory MIMD** based on the way PEs are coupled to the main memory.

In the **shared memory MIMD** model (tightly coupled multiprocessor systems), all the PEs are connected to a single global memory and they all have access to it. The communication between PEs in this model takes place through the shared memory, modification of the data stored in the global memory by one PE is visible to all other PEs. Dominant representative shared memory MIMD systems are Silicon Graphics machines and Sun/IBM's SMP (Symmetric Multi-Processing).

In **Distributed memory MIMD** machines (loosely coupled multiprocessor systems) all PEs have a local memory. The communication between PEs in this model takes place through the interconnection network (the inter process communication channel, or IPC). The network connecting PEs can be configured to tree, mesh or in accordance with the requirement.

The shared-memory MIMD architecture is easier to program but is less tolerant to failures and harder to extend with respect to the distributed memory MIMD model. Failures in a shared-memory MIMD affect the entire system, whereas this is not the case of the distributed model, in which each of the PEs can be easily isolated. Moreover, shared memory MIMD architectures are less likely to scale because the addition of more PEs

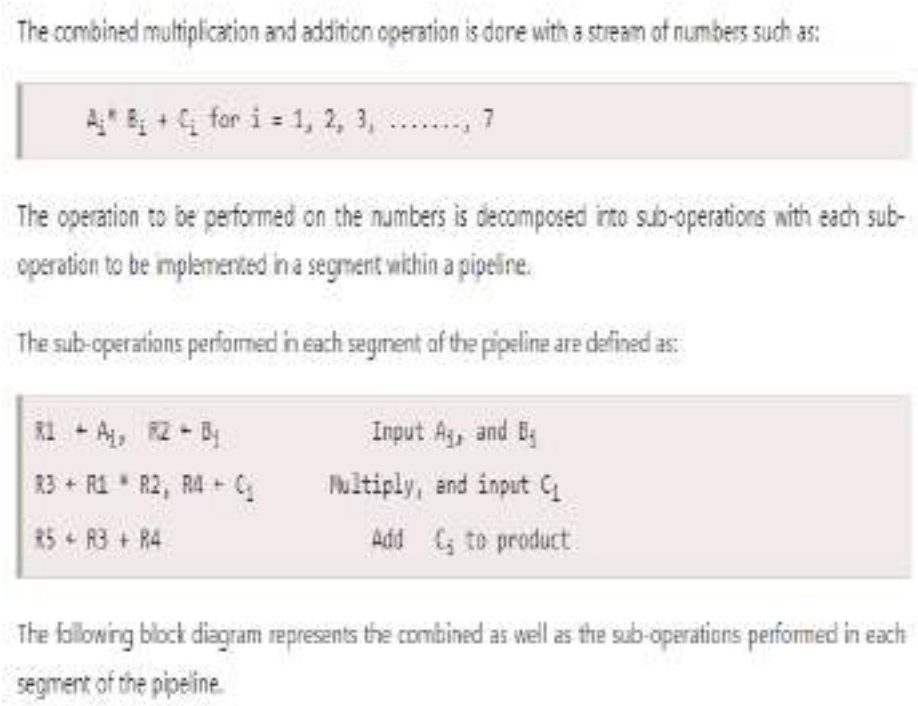
leads to memory contention. This is a situation that does not happen in the case of distributed memory, in which each PE has its own memory. As a result of practical outcomes and user's requirement, distributed memory MIMD architecture is superior to the other existing models.

Pipelining

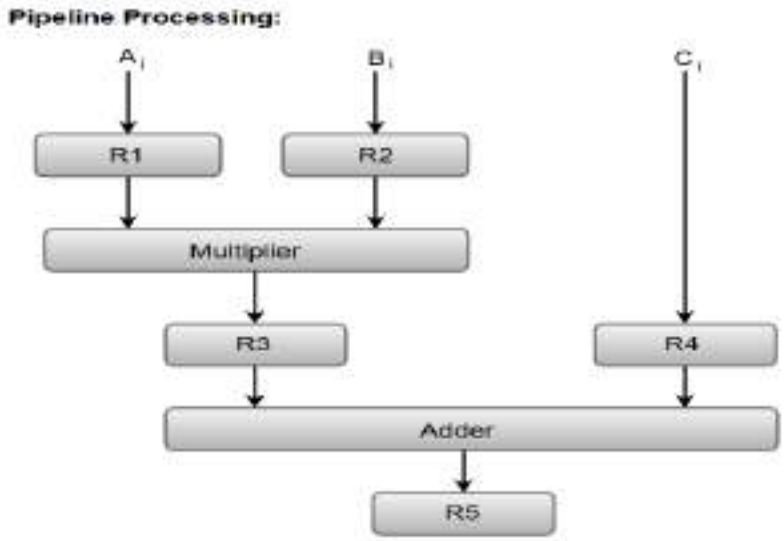
The term Pipelining refers to a technique of decomposing a sequential process into sub-operations, with each sub-operation being executed in a dedicated segment that operates concurrently with all other segments.

The most important characteristic of a pipeline technique is 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.

The structure of a pipeline organization can be represented simply by including an input register for each segment followed by a combinational circuit.



The following block diagram represents the combined as well as the sub-operations performed in each segment of the pipeline.



Registers R1, R2, R3, and R4 hold the data and the combinational circuits operate in a particular segment.

The output generated by the combinational circuit in a given segment is applied as an input register of the next segment. For instance, from the block diagram, we can see that the register R3 is used as one of the input registers for the combinational adder circuit.

In general, the pipeline organization is applicable for two areas of computer design which includes:

1. Arithmetic Pipeline
2. Instruction Pipeline

Arithmetic Pipeline

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 in a more convenient way, let us consider an example of a pipeline unit for floating-point addition and subtraction. The inputs to the floating-point adder pipeline are two normalized floating-point binary numbers defined as:

$$\mathbf{X = A * 2^a = 0.9504 * 10^3}$$

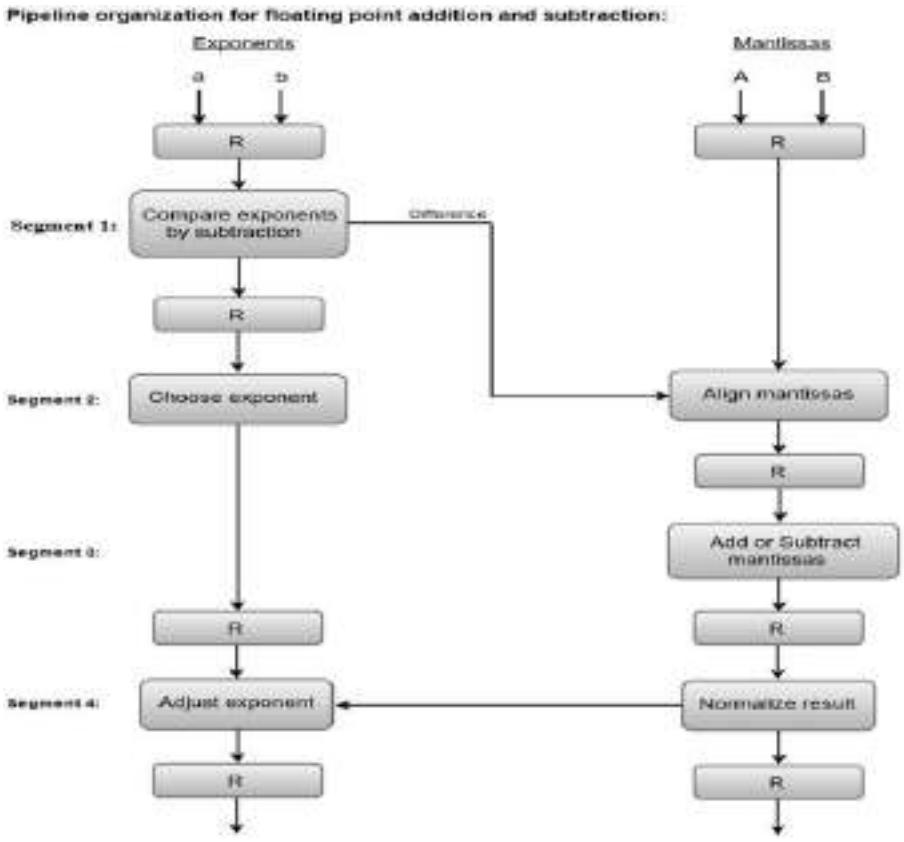
$$Y = B * 2^b = 0.8200 * 10^2$$

Where **A** and **B** are two fractions that represent the mantissa and **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 suboperation to be performed in the given pipeline. The suboperations 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 following block diagram represents the sub operations performed in each segment of the 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:

$$Z = 0.1324 * 10^4$$

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 general, the computer needs to process each instruction with the following sequence of steps.

1. Fetch instruction from memory.

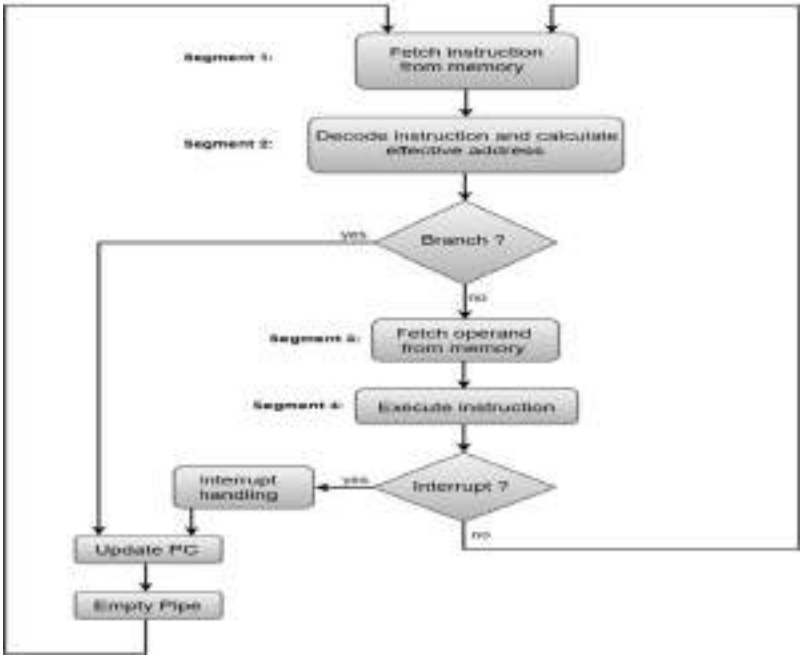
2. Decode the instruction.
3. Calculate the effective address.
4. Fetch the operands from memory.
5. Execute the instruction.
6. Store the result in the proper place.

Each step is executed in a particular segment, and there are times when different segments may take different times to operate on the incoming information. Moreover, there are times when two or more segments may require memory access at the same time, causing one segment to wait until another is finished with the memory.

The organization of an instruction pipeline will be more efficient if the instruction cycle is divided into segments of equal duration. One of the most common examples of this type of organization is a **Four-segment instruction pipeline.**

A **four-segment instruction** pipeline combines two or more different segments and makes it as a single one. For instance, the decoding of the instruction can be combined with the calculation of the effective address into one segment.

The following block diagram shows a typical example of a four-segment instruction pipeline. The instruction cycle is completed in four segments.



Segment 1:

The instruction fetch segment can be implemented using first in, first out (FIFO) buffer.

Segment 2:

The instruction fetched from memory is decoded in the second segment, and eventually, the effective address is calculated in a separate arithmetic circuit.

Segment 3:

An operand from memory is fetched in the third segment.

Segment 4:

The instructions are finally executed in the last segment of the pipeline organization.

RISC Pipeline

RISC stands for Reduced Instruction Set Computers. It was introduced to execute as fast as one instruction per clock cycle. This RISC pipeline helps to simplify the computer architecture's design.

It relates to what is known as the Semantic Gap, that is, the difference between the operations provided in the high-level languages (HLLs) and those provided in computer architectures.

To avoid these consequences, the conventional response of the computer architects is to add layers of complexity to newer architectures. This also increases the number and complexity of instructions together with an increase in the number of addressing modes. The architecture which resulted from the adoption of this "add more complexity" are known as Complex Instruction Set Computers (CISC).

The main benefit of RISC to implement instructions at the cost of one per clock cycle is continually not applicable because each instruction cannot be fetched from memory and implemented in one clock cycle correctly under all circumstances.

The method to obtain the implementation of an instruction per clock cycle is to initiate each instruction with each clock cycle and to pipeline the processor to manage the objective of single-cycle instruction execution.

RISC compiler gives support to translate the high-level language program into a machine language program. There are various issues in managing complexity about data conflicts and branch penalties are taken care of by the RISC processors, which depends on the adaptability of the compiler to identify and reduce the delays encountered with these issues.

Principles of RISCs Pipeline

There are various principles of RISCs pipeline which are as follows −

- Keep the most frequently accessed operands in CPU registers.
- It can minimize the register-to-memory operations.
- It can use a high number of registers to enhance operand referencing and decrease the processor memory traffic.
- It can optimize the design of instruction pipelines such that minimum compiler code generation can be achieved.
- It can use a simplified instruction set and leave out those complex and unnecessary instructions.

Let us consider a three-segment instruction pipeline that shows how a compiler can optimize the machine language program to compensate for pipeline conflicts.

A frequent collection of instructions for a RISC processor is of three types are as follows −

- **Data Manipulation Instructions** − Manage the data in processor registers.
- **Data Transfer Instructions** − These are load and store instructions that use an effective address that is obtained by adding the contents of two registers or a register and a displacement constant provided in the instruction.
- **Program Control Instructions** − These instructions use register values and a constant to evaluate the branch address, which is transferred to a register or the program counter (PC).

Example: Three-Segment Instruction Pipeline

The data manipulation instructions operate on data in processor registers. The data transfer instructions are load and store instructions that use an effective address obtained from the addition of the contents of two registers or a register and a displacement constant provided in the instruction. The program control instructions use register values and a constant to evaluate the branch address, which is transferred to a register or the program counter PC.

The control section fetches the instruction from program memory into an instruction register. The instruction is decoded at the same time that the registers needed for the execution of the instruction are selected. The processor unit consists of a number of registers and an arithmetic logic unit (ALU) that performs the necessary arithmetic, logic, and shift operations. A data memory is used to load or store the data from a selected register in the register file. The instruction cycle can be divided into three suboperations and implemented in three segments:

I: Instruction fetch

A: ALU operation

E: Execute instruction

The I segment fetches the instruction from program memory. The instruction is decoded and an ALU operation is performed in the A segment. The ALU is used for three different functions, depending on the decoded instruction. It performs an operation for a data manipulation instruction, it evaluates the effective address for a load or store instruction, or it calculates the branch address for a program control instruction. The E segment directs the output of the ALU to one of three destinations, depending on the decoded instruction. It transfers the result of the ALU operation into a destination register in the register file, it transfers the effective

address to a data memory for loading or storing, or it transfers the branch address to the program counter.

Delayed Load Consider now the operation of the following four instructions:

1. LOAD: Rl<--M [address 1]
2. LOAD: R2 <--M [address 2]
3. ADD: R3 <--Rl + R2
4. STORE: M[address 3] <--R3

If the three-segment pipeline proceeds without interruptions, there will be a data conflict in instruction 3 because the operand in R2 is not yet available in the A segment.

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 loading.

Clock cycles:	1	2	3	4	5	6
1. Load *R*1	I	A	E			
2. Load *R*2		I	A	E		
3. Add *R*1 + *R*2			I	A	E	
4. Store *R*3				I	A	E

(a) Pipeline timing with data conflict

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

(b) Pipeline timing with delayed load

Figure 9 Three-segment pipeline timing.

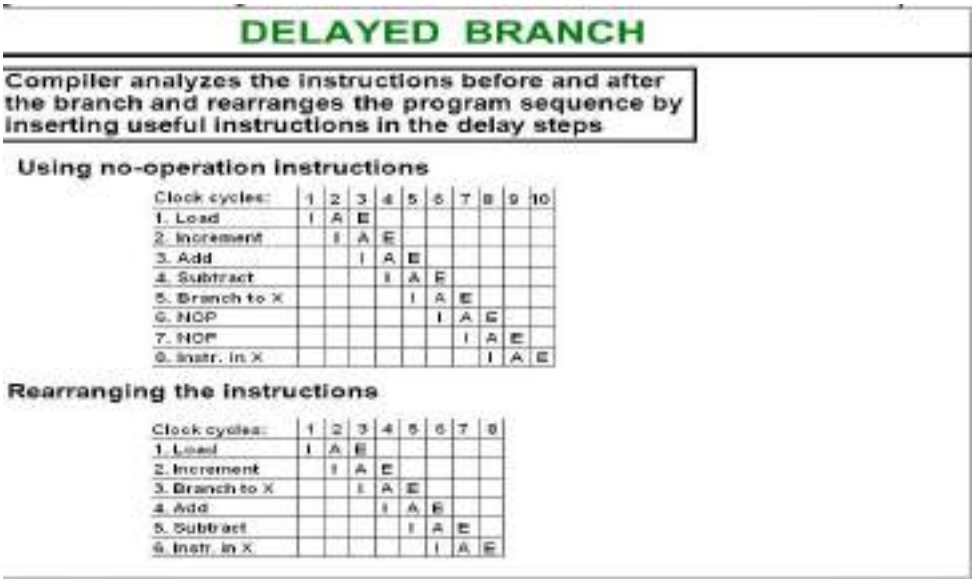
Figure shows the same program with a no-op instruction inserted after the load to R2 instruction. The data is loaded into R2 in clock cycle 4. The add instruction uses the value of R2 in step 5. Thus the no-op instruction is used to advance one clock cycle in order to compensate for the data conflict in the pipeline.

The advantage of the delayed load approach is that the data dependency is taken care of by the compiler rather than the hardware. This results in a simpler hardware segment since the segment does not have to check if the content of the register being accessed is currently valid or not.

Delayed Branch was shown that a branch instruction delays the pipeline operation until the instruction at the branch address is fetched. Several techniques for reducing branch penalties 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 for a processor that uses delayed branches is designed to analyze the instructions before and after the branch and rearrange the program sequence by inserting useful instructions in the delay steps. For example, the compiler can determine that the program dependencies allow one or more instructions preceding the branch to be moved into the delay steps after the branch. These instructions are then fetched from memory and executed through the pipeline while the branch instruction is being executed in other segments. The effect is the same as if the instructions were executed in their original order, except that the branch delay is removed. 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 is shown in Fig. 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. 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 in Fig. is rearranged by placing the add and subtract instructions after the branch instruction instead of before as in the original program. Inspection of the pipeline timing shows that PC is updated to the value of X in clock cycle 5, but the 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 105 and 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 cycle 6 is from address 350, which is the instruction at the branch address.



PIPELINE HAZARDS

DEFINITION OF A HAZARD

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.

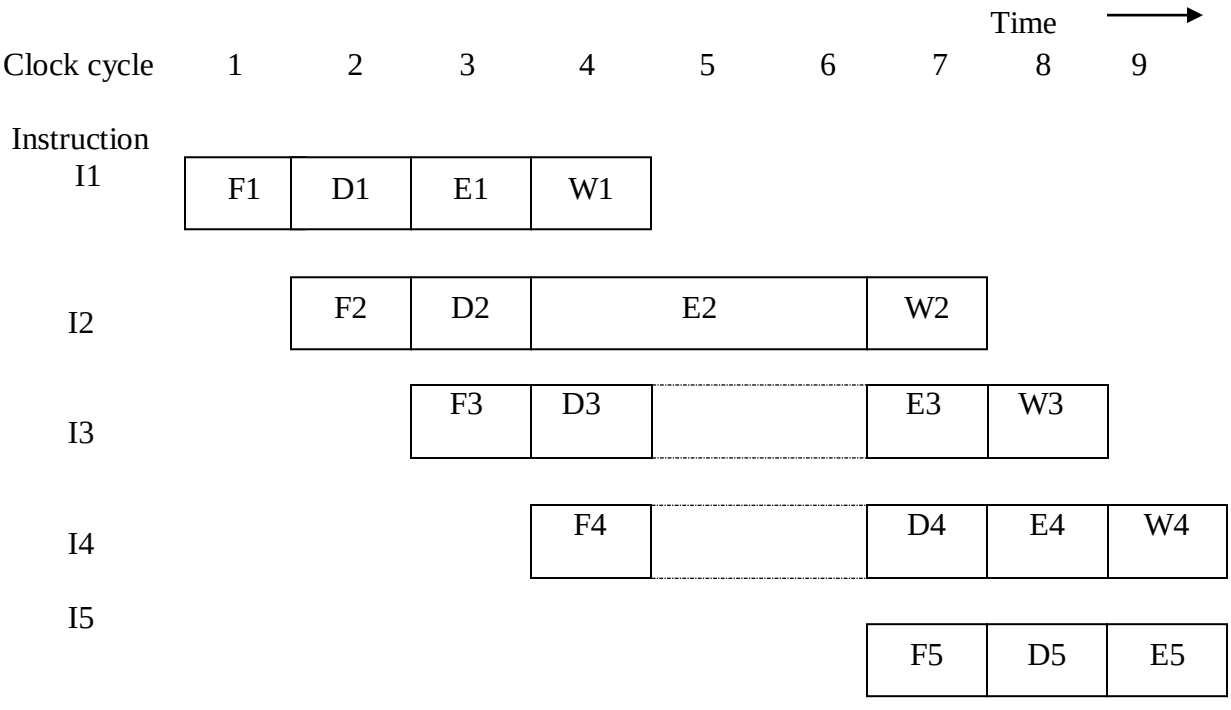


Figure 6.1: Effect of execution operation taking more than one clock cycle.

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.

$$A \leftarrow 3 + A$$

$$B \leftarrow 4 * A$$

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:

$$A \leftarrow 5 * C$$

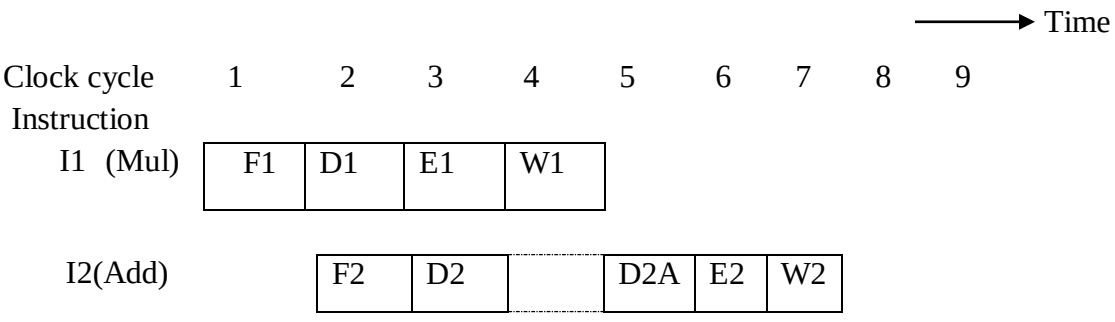
$$B \leftarrow 20+C$$

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

Example 2: consider two instructions:

Mul R2, R3, R4

Add R5, R4, R6.



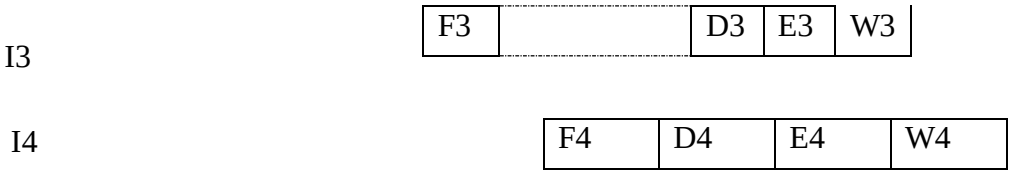


Figure 6.2: Pipeline stalled by data dependency between D2 and W1.

In the above example in 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.

Example 3:

ADD R_1, R_2, R_3

SUB R_4, R_5, R_1

AND R_6, R_1, R_7

OR R_8, R_1, R_9

XOR R_{10}, R_1, R_{11}

All the instructions after the ADD use the result of the ADD instruction (R1). The ADD instruction writes the value of R1 in WB stage and the SUB instruction reads the value during ID stage. This Problem is called as a data hazard. Unless precautions are taken, the SUB instruction will read the wrong value and try to use it.

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.

For example:

I_1:$R_2 \leftarrow R_1 + R_3$

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.

For 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.

Resolving Data Hazards

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

- Insert a pipeline bubble whenever a read after write (RAW) dependency is encountered which is guaranteed to increase latency.
- Use out-of-order execution to potentially prevent bubbles.
- Use Register forwarding/Operand forwarding to use data from later stages in a pipeline.

In case of **out-of-order execution**, two algorithms can be used:

- **Score boarding**, in which case a pipeline bubble will only be needed when there is no functional unit available.
- **Tomasulo algorithm**, which utilizes register renaming allowing the continuous issuing of instructions.

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

Operand 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.

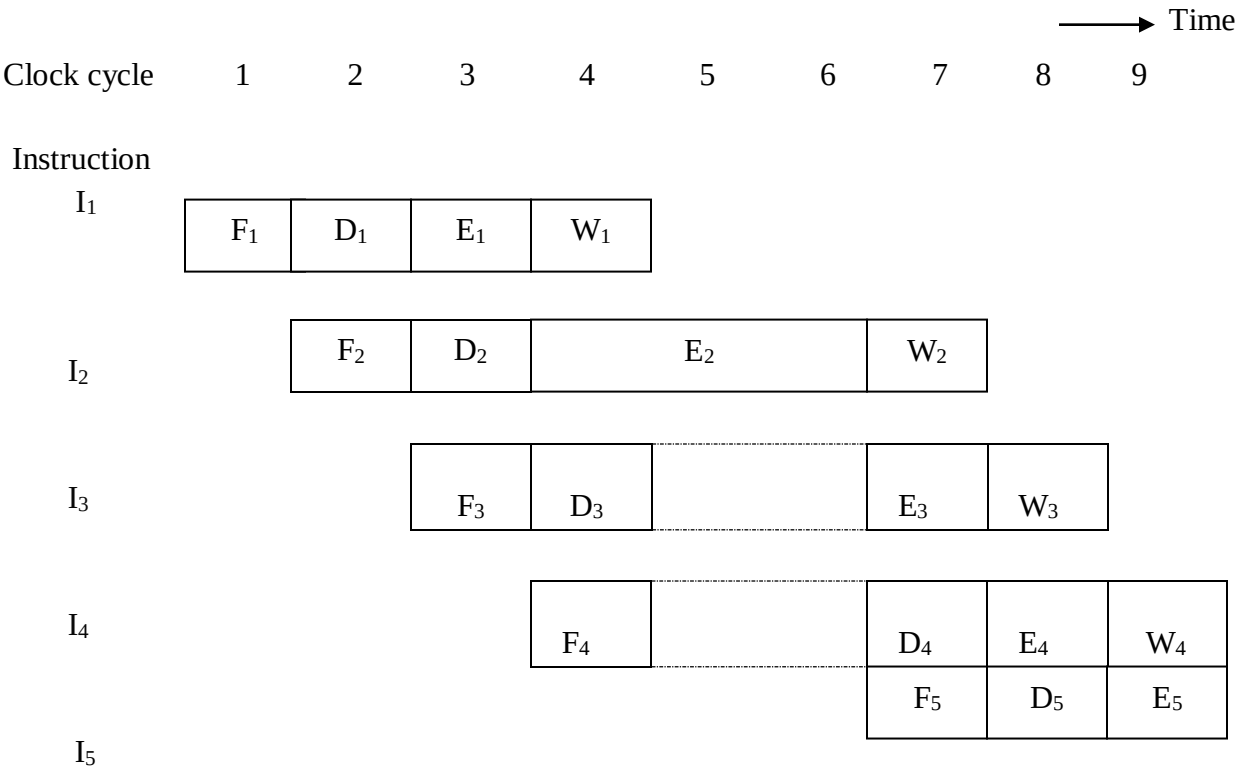


Figure 6.3: Data hazard situation between the instructions.

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.

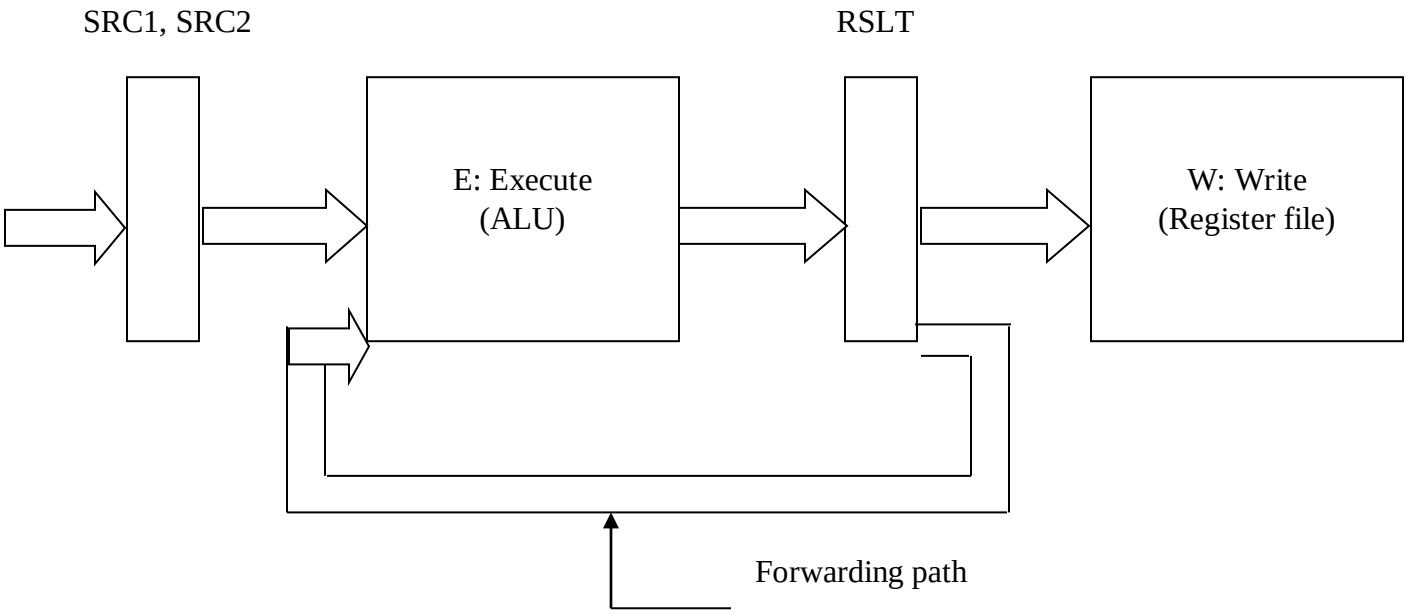


Figure 6.4: Effect of Data forwarding on the processor

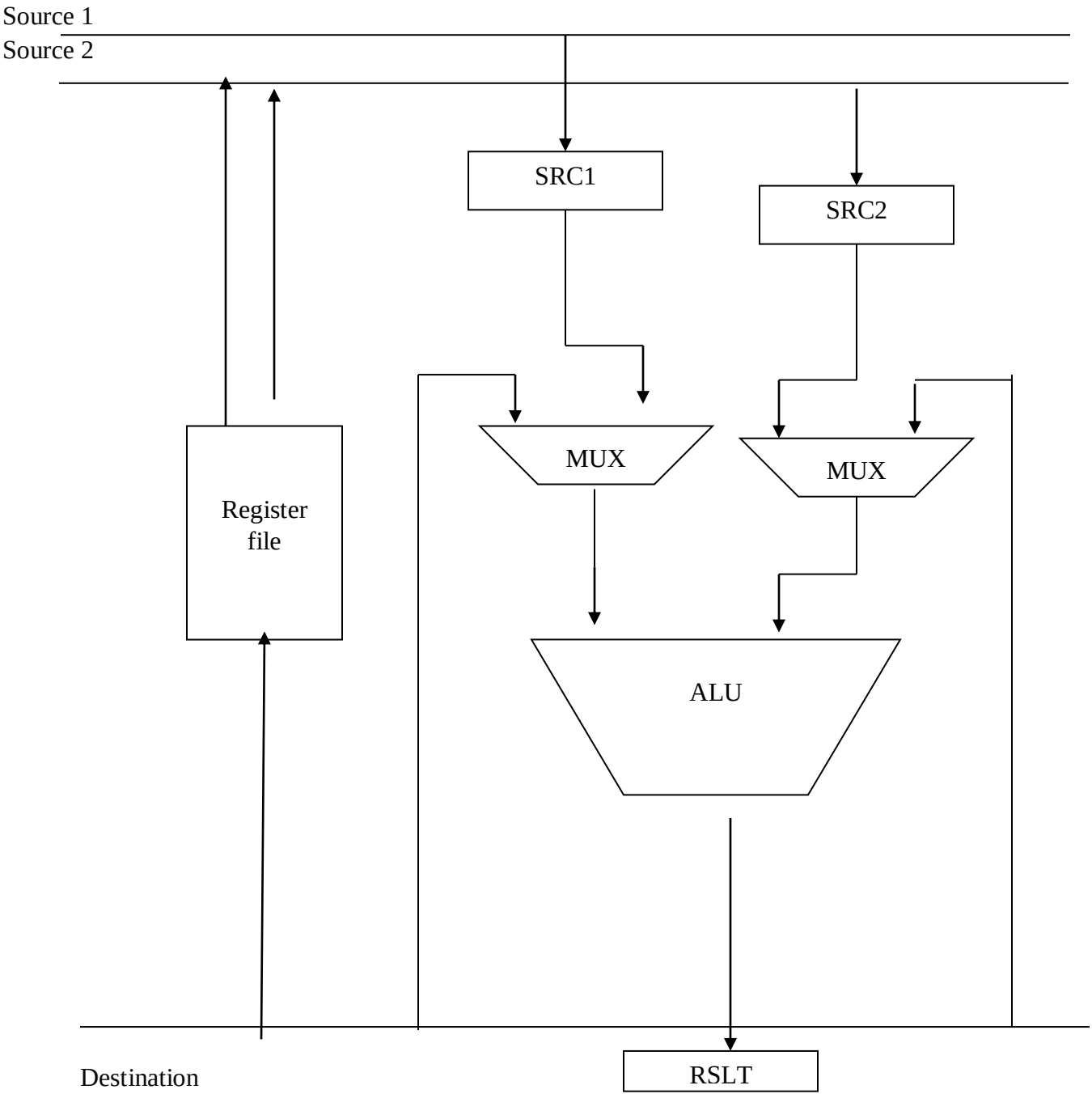


Figure 6.5: Datapath

After decoding the instruction I2 and detecting a data dependency, a decision is made to use data forwarding. The Operand not involved in the dependency, register R2, is read and

loaded in register SRC1 in clock cycle 3, the product produced by instruction I1 is available in the register RSLT, and because of the forwarding connection, it can be used in step E_2. Hence, the execution of I2 proceeds without interruption.

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). It causes the pipeline to stall for one cycle, because both I_2 and I_3 require access to the register file in cycle 6.

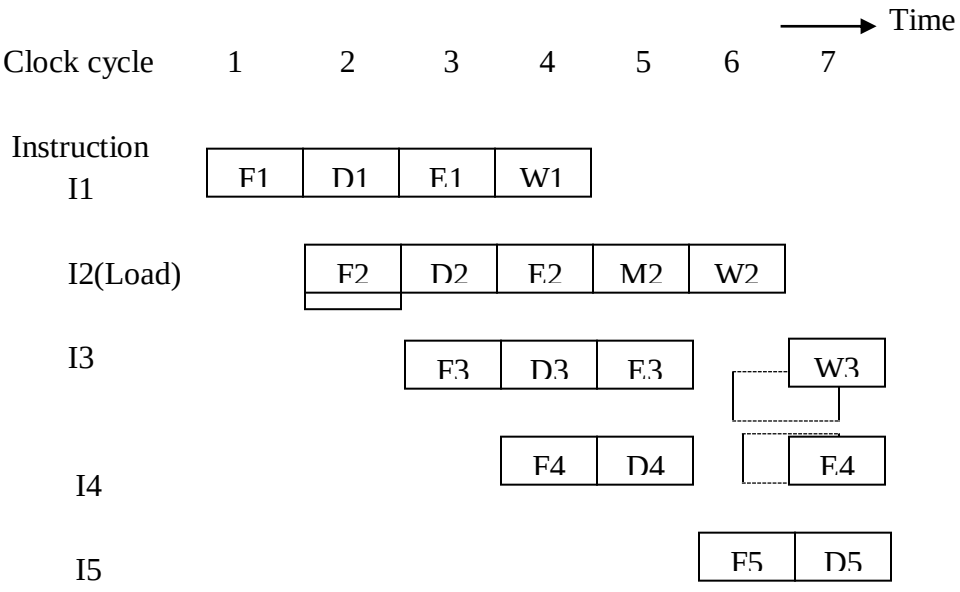


Figure : Effect of a load Instruction on Pipeline.

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 may have only one register-file write port, but in some cases the pipeline might want to perform two writes in a clock cycle.

Example 3:

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

Figure : 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. The following table shows how the stalls are actually implemented.

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

Figure : 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

Figure : 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.

BRANCH HAZARDS

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).

Branching 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

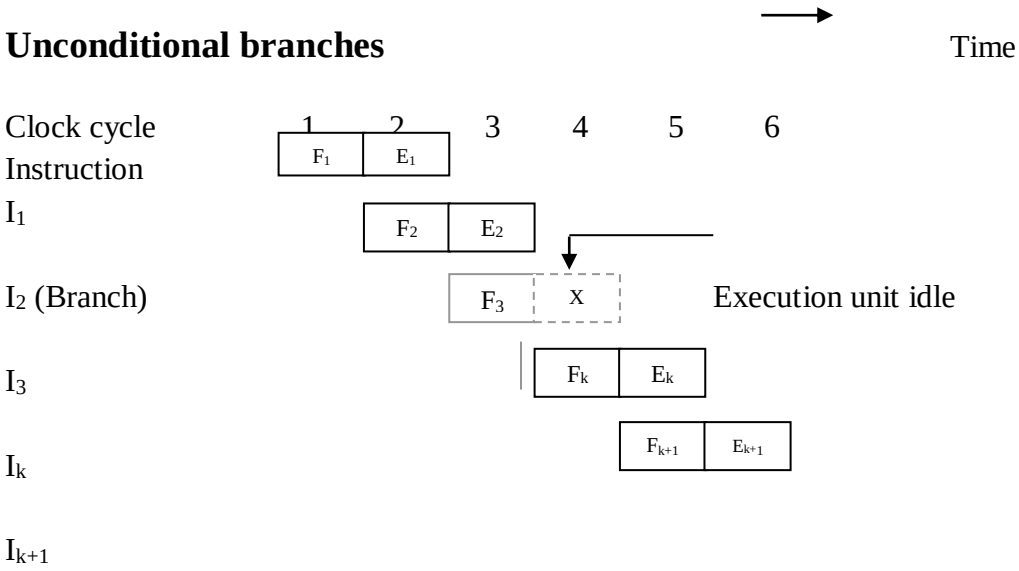


Figure : An idle cycle caused by a branch instruction.

In Figure 6.11, 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.

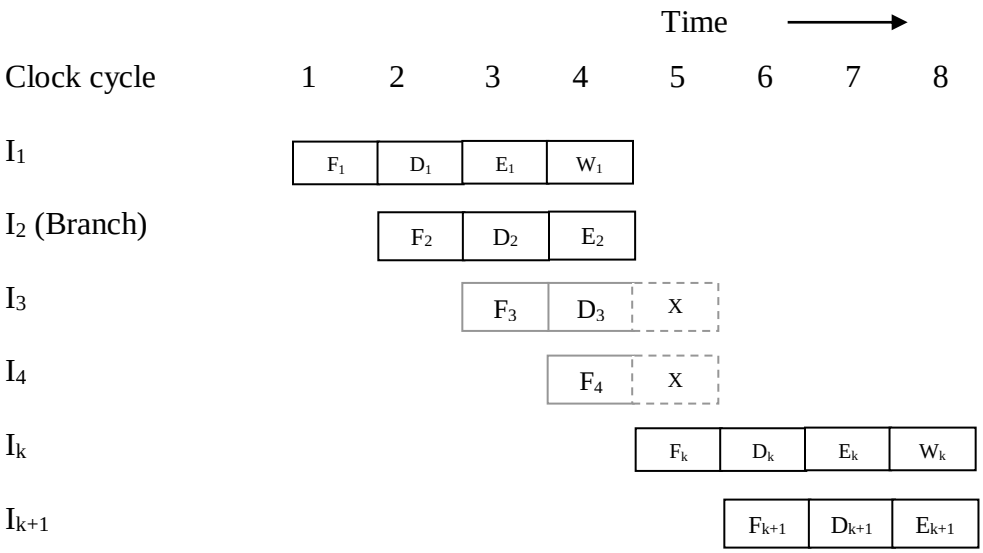


Figure : Branch address computed in Execute stage.

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.

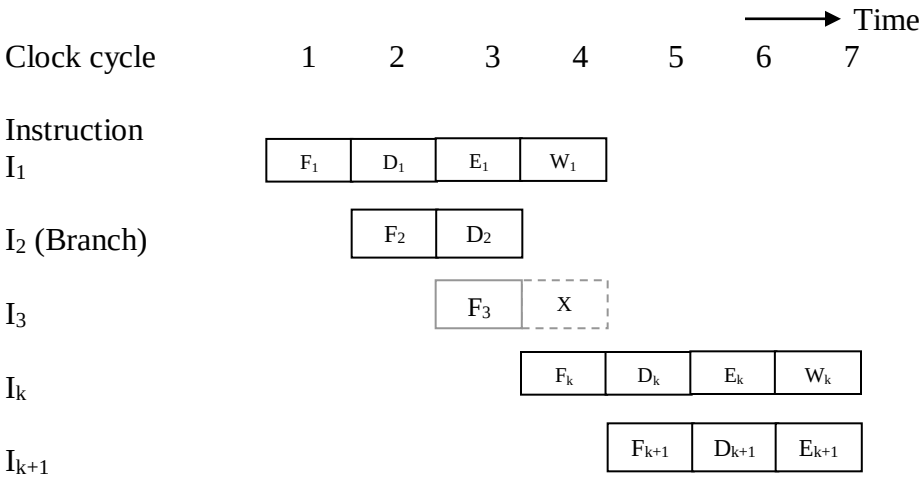


Figure : Branch address computed in Decode stage

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.

Instruction Queue and Prefetching

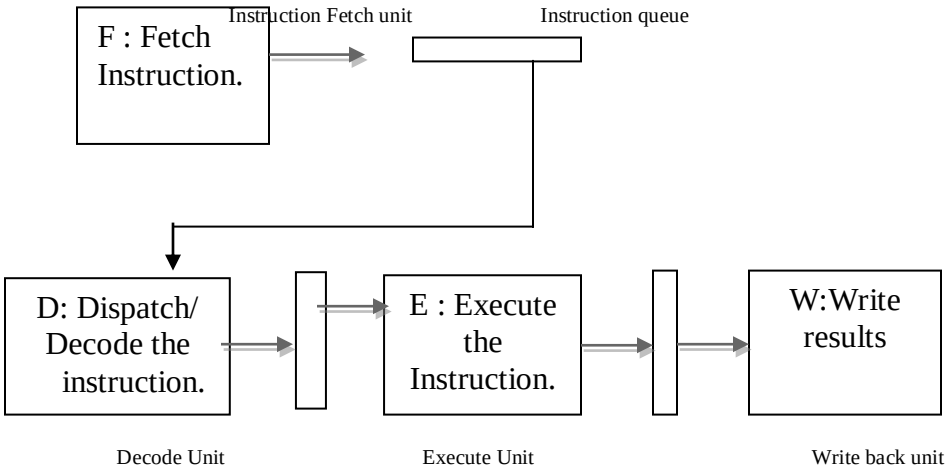


Figure : Use of instruction queue in the hardware organization.

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. Both of these tasks can be performed in step D_2.

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. 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, 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.

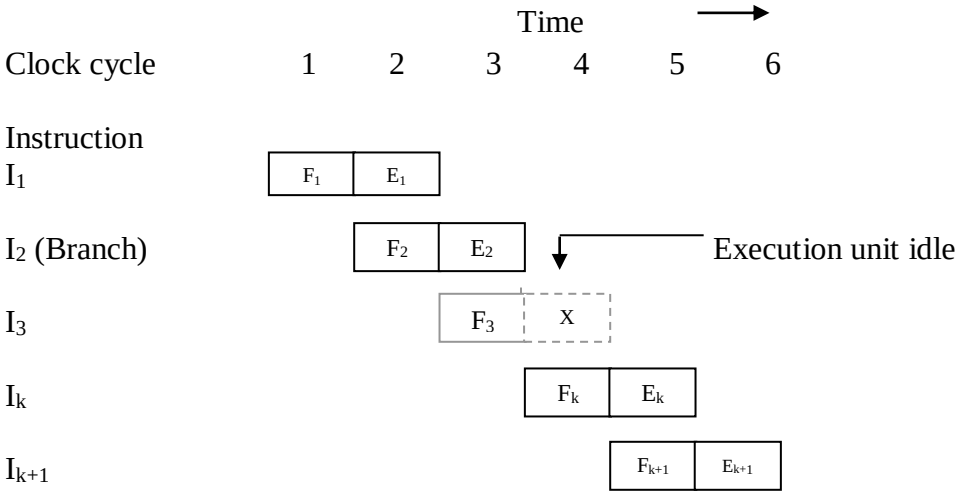


Figure 6.15: An idle cycle caused by a branch instruction.

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

Finally, problems arise when a conditional branch instruction is encountered and a previous instruction that sets the condition codes has not been completed yet and, therefore, the control section - not knowing yet whether the branch conditions are satisfied – does not know whether to prefetch the instruction immediately following this conditional branch instruction or the instruction at the target address of the branch.

Software solutions include the following:

1. Branch spreading.
2. Scheduling the branch delay slot.
3. Branch folding.
4. 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.
5. Trace scheduling.
6. Loop unrolling.
7. Pipeline Bubbling: insert a pipeline bubble, guaranteed to increase latency.
8. Register renaming.
9. 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 (upon which the branch decision will be based) 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. The Intel 486follows this approach and assumes that the branch is taken and – in parallel with the operation of the pipeline stages – the CPU runs a “speculative fetch cycle” to the target address of the branch. If the CPU evaluates the

branch condition as true, the fetch to the target address refills the pipeline; otherwise, the processor loses three clock cycles. (Such hardware prediction 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.

If a branch causes a pipeline bubble after incorrect instructions have entered the pipeline, care must be taken to prevent any of the wrongly-loaded instructions from having any effect on the processor state excluding energy wasted processing them before they were discovered to be loaded incorrectly.

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.

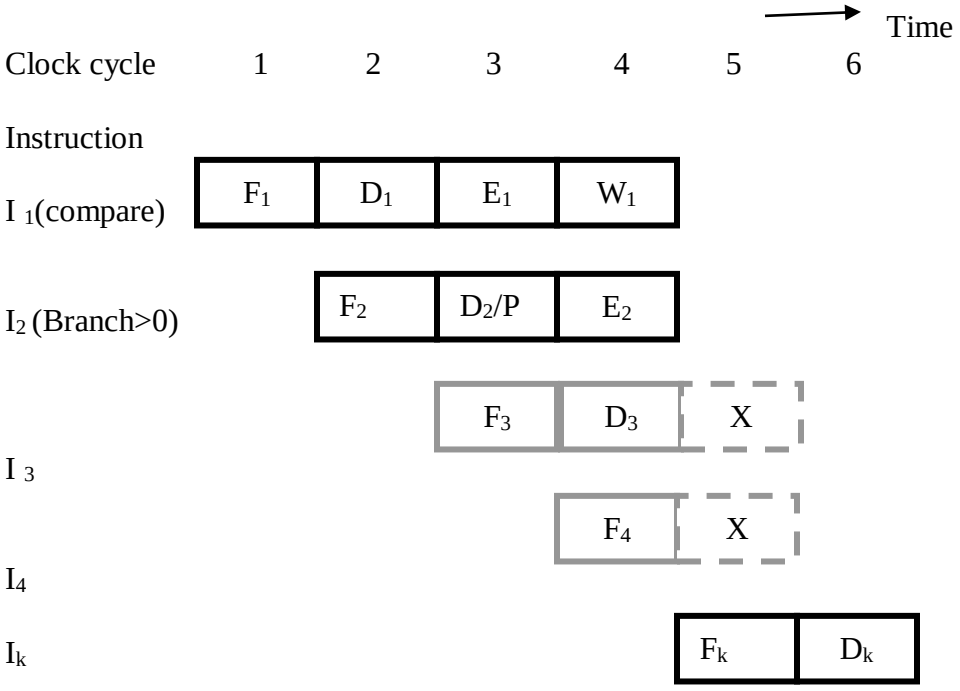


Figure : Timing diagram when a branch decision has been incorrectly predicted as not taken.

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.

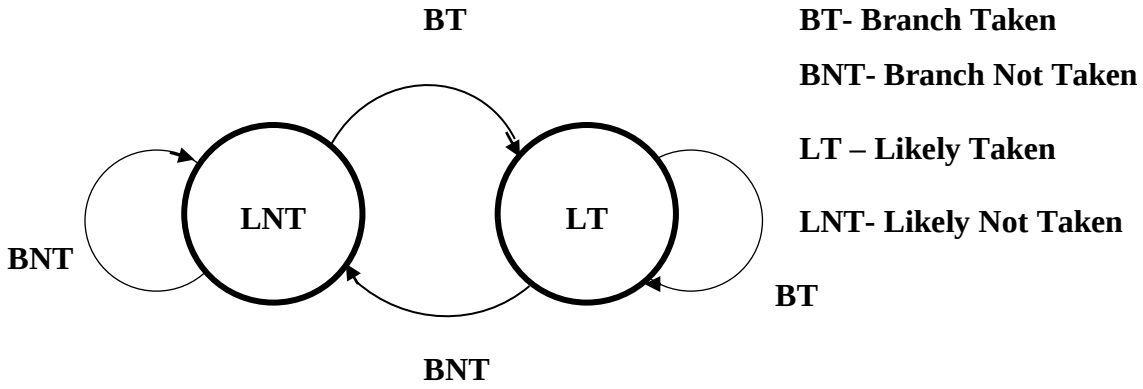


Figure : 2 -State algorithm.

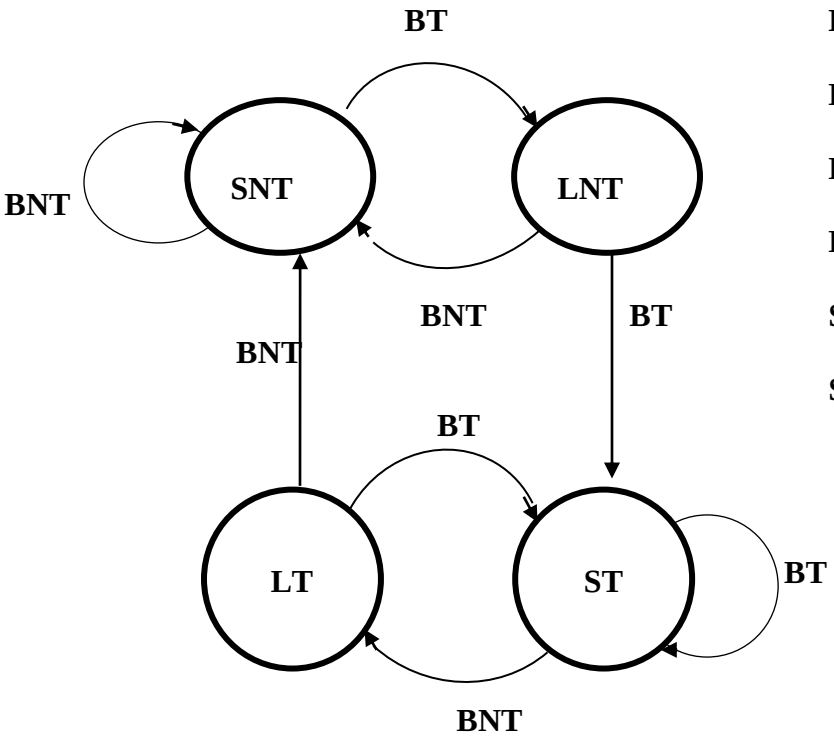
4-State algorithm

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.

Introduction to Processor Architecture

COMPLEX INSTRUCTION SET COMPUTER (CISC)

CISC is an acronym for Complex Instruction Set Computer. A **complex instruction set computer** (**CISC**) (pronounced sisk), is a computer where single instructions can execute several low-level operations (such as a load from memory, an arithmetic operation, and a memory store) and/or capable of multi-step operations or addressing modes within single instruction.

As the price of hardware was decreasing and the price of software was increasing. Researchers decided to take the burden off of the software and use the hardware to do all the work. The idea was to make the job easier for the compiler by having more instructions that were like high-level language statements which leads to CISC.

CISC takes its name from the very large number of instructions (typically hundreds) and addressing modes in its ISA. CISC instructions are of varying lengths (often ranging from 8 - 120 bits) which is as opposed to RISC architecture which has a fixed instruction set.

Principles of CISC

- Large instruction set: CISC chips have a large amount of different and complex instructions. The philosophy is that hardware is always faster than software, therefore one should make a powerful instruction set, which provides programmers with assembly instruction to do a lot with shorter programs

- Complex operations

- Complex addressing modes

- Complex hardware, long execution time

- Minimum number of instructions needed for a given task

- Easy to program, simpler compiler

Characteristics of CISC

The common characteristics of CISC architecture are:

- A 2-operand format, where instructions have a source and a destination. Register to register, register to memory, and memory to register commands. Multiple addressing modes for memory, including specialized modes for indexing through arrays
- Variable length instructions where the length often varies according to the addressing mode
- Instructions which require multiple clock cycles to execute
- Complex instruction-decoding logic, driven by the need for a single instruction to support multiple addressing modes
- A small number of general purpose registers
- Several special purpose registers are used. Many CISC designs set aside special registers for the stack pointer, interrupt handling, and so on. This can simplify the hardware design by making the instruction set more complex
- A 'Condition code" register which is set as a side-effect of most instructions. This register reflects whether the result of the last operation is less than, equal to, or greater than zero and records if certain error conditions occur

At the time of their initial development, CISC machines used available technologies to optimize computer performance.

- Microprogramming is as easy as assembly language to implement, and much less expensive than hardwiring a control unit
- The ease of microcoding new instructions allowed designers to make CISC machines upwardly compatible: a new computer could run the same programs as earlier computers because the new computer would contain a superset of the instructions of the earlier computers
- As each instruction became more capable, fewer instructions could be used to implement a given task. This made more efficient use of the relatively slow main memory
- Because microprogram instruction sets can be written to match the constructs of high-level languages, the compiler does not have to be as complicated

Advantages of CISC

- CISC has varying lengths to reduce wasted space in memory
- Has developed a process to manage power which adjusts clock speed and voltage
- Uses less instructions to perform similar instructions than RISC
- Provides programmers with assembly instr. to do a lot with smaller programs
- CISC instructions are complex but specialized
- It has a smaller code size
- Several addressing modes can increase the flexibility
- Can perform complex functions in limited CPU clock cycles

Disadvantages of CISC

- CISC chips are relatively slow (compared to RISC chips) per instruction
- Since as many instructions as possible could be stored in memory with the least possible wasted space, individual instructions could be of almost any length - this means that different instructions will take different amounts of clock time to execute, slowing down the overall performance of the machine
- Variable length instructions are more difficult to process, so CISC chips require many more transistors than comparable RISC designs
- This increase in transistor count has obvious implications for the issue of power dissipation, which is central to mobile computing
- The increase in transistor count also makes it more expensive to produce CISC chips
- The complexity of CISC also makes it more difficult to pipeline than RISC, which again increases the required processor logic and, hence, transistor count

Examples of CISC

- System/360
- VAX
- PDP-11
- Motorola 68000 family
- Intel x86/Pentium CPU’s

REDUCED INSTRUCTION SET COMPUTER (RISC)

In the early 80's the idea of RISC was introduced. The SPARC project was started at Berkeley and the MIPS project at Stanford. RISC stands for 'Reduced Instruction Set Computer'.

RISC chips evolved around the mid-1980. The philosophy is that almost no one uses complex assembly language instructions as used by CISC, and users mostly use compilers which never use complex instructions. RISC architectures are also called *LOAD/STORE* architectures.

Therefore fewer, simpler and faster instructions would be better than the large, complex and slower CISC instructions. However, more instructions are needed to accomplish a task.

To make all instructions of the same length, the number of bits that are used for the opcode is reduced. Thus less number of instructions is provided i.e. the no of instructions is reduced. So, this architecture is called RISC. The main goal of RISC is to make the instructions simple so that they can be easily pipelined and achieve a single clock throughput at higher frequencies.

History of RISC

The first RISC projects came from IBM, Stanford, and UC-Berkeley in the late 70s and early 80s. The IBM 801, Stanford MIPS, and Berkeley RISC 1 and 2 were all designed with a similar philosophy which has become known as RISC. Design features of most RISC processors are:

- **One cycle execution time:** RISC processors have a CPI (clock per instruction) of one cycle. This is due to the optimization of each instruction on the CPU and a technique called PIPELINING
- **Pipelining**: A technique that allows for simultaneous execution of parts, or stages, of instructions to more efficiently process instructions
- **Large number of registers**: the RISC design incorporates a larger number of registers to prevent in large amounts of interactions with memory

Principles of RISC

- Small instruction set
- Simple instructions to allow for fast execution (fewer steps)
- Both operands should be available in registers to allow for short fetch time

- Large number of registers
- Only read/write (load/store) instruction should access the main memory, 1 MM access per instruction
- Simple addressing modes to allow for fast address computation
- Fixed-length instruction with few formats and aligned fields to allow for fast instruction decoding
- Complex tasks are left to the compiler to construct from simple operations, with increased compiler complexity and compiling time
- Simpler and faster hardware implementation suitable for pipelined architecture

Characteristics of RISC

- Reduced instruction set
- Less complex, simple instructions
- Uniform Instruction format
- Simple addressing modes
- Fewer data types in hardware
- Hardwired control unit and machine instructions: (the instructions are hard wired) In a RISC processor, the Execution Unit is no longer controlled by the Control unit with the assistance of extensive microcodes. Instead, the whole operation is achieved in the form of hardwired logic. This greatly accelerates the execution of an instruction
- Few addressing schemes for memory operands with two basic instructions LOAD and STORE
- Many symmetric register which are organized into a register file

Advantages of RISC

- RISC can execute its instructions very fast because the instructions are so simple.
- It uses most commonly used instructions
- It requires less clock cycles to execute
- It requires less memory and more registers
- It uses register windows for function calls
- It requires fewer transistors, which makes them cheaper to design and produce

- It is easier to write powerful optimized compilers, since fewer instructions exist.
- It uses simpler hardware.
- It is lower in cost since more parts can be placed on a single chip

Disadvantages of RISC

- Code size: Usually more instructions are needed and there is a waste in short instructions. (POP, PUSH)
- Complex functions require several instructions
- Little flexibility with memory addressing modes

Examples of RISC

- IBM 360.
- DEC VAX.
- Intel 80x 86 families.
- Motorola 68xxx.

First RISC

The first system known as RISC was the CDC 6600 supercomputer, designed in 1964. The CDC 6600 had load-store architecture with only two addressing modes (register register, and register immediate) and 74 opcodes. The 6600 had eleven pipelined functional units for arithmetic and logic, plus five load units and two store units; the memory had multiple banks so all load-store units could operate at the same time. The clock cycle/instruction issue rate was 10 times faster than the memory access time.

EXAMPLE OF RISC AND CISC

The advantages and disadvantages of RISC and CISC can be better understood with an example in Figure

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.

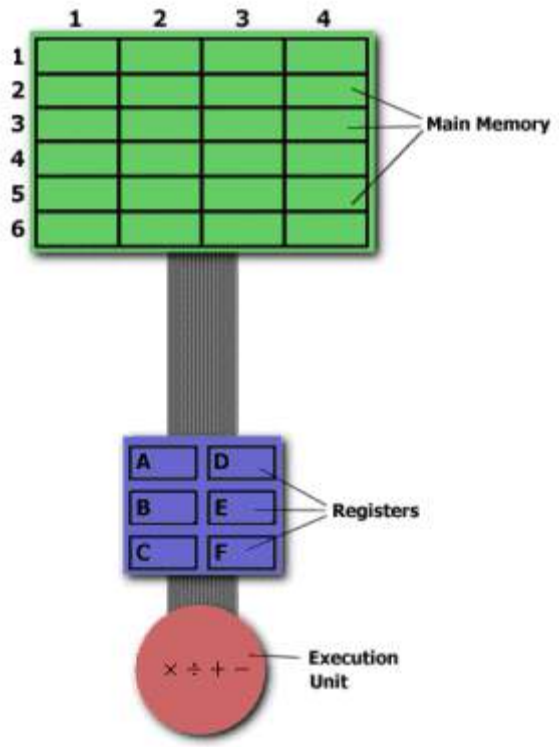


Figure: Multiplying two numbers in the memory

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.

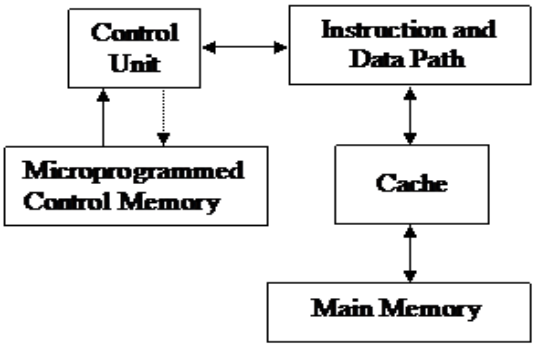
```
LOAD A, 2:3
LOAD B, 5:2
PROD A, B
STORE 2:3, A
```

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 registers.
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. A CISC system has complex instructions such as direct addition between data in two memory locations**. Eg.8085**	19. Pipelining can be implemented easily. Eg. ATMEL AVR
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**.** • Motorola 68000 family**.** • Intel x86/Pentium CPU's	21. **Examples of RISC:** • IBM 360. • DEC VAX. • Intel 80x 86 families. • Motorola 68xxx.
22. Modern CISC 	22. Traditional RISC