Basic Computer Architecture and Quantitative Techniques in Computer Design Instruction Pipeline ,Arithmetic Pipeline, Hazards ,Exception and Interrupts

Rupam Sardar

Budge Budge Institute of Technology,Kolakata-700137

Abstract:

We talk about the quantitative approach to computer architecture in this paper. We also go over Amdahl's Law, scalability, parallelism, and the principle of locality in relation to quantitative measurement. We also demystify computer architecture by focusing on smart technical design and cost-performance-power trade-offs. In the context of computer architecture, we think the field has continued to develop and move toward the exacting quantitative foundation of venerable scientific and technical disciplines.

The pipelining principles in processor design are the focus of this study.The fundamentals of the instruction pipeline are covered, and an example-based explanation of how to reduce a pipeline delay is provided.The primary goal is to comprehend how a processor's pipeline functions.The different risks that lead to pipeline deterioration are described, along with ways to reduce them.

Architecture-based development environments are emerging as a useful tool for building reliable distributed systems. By means of the abstract characterization of intricate software
Software reuse and evolution are encouraged in terms of system topologies that involve the interface-level interaction of software parts. Furthermore, as evidenced by research findings in the field of software architecture, it becomes possible to offer formal annotations for the accurate characterization of configuration behavior together with related CASE tools for automated analysis.
Nevertheless, while being essential to attaining software resilience, software fault tolerance—and specifically exception management in that context—has received little attention.

Keywords: Instruction Pipeline ,Arithmetic Pipeline, Hazards

Introduction

The foundation of computer architecture research is now the quantitative method. But because of their extreme complexity, computer systems are costly to create and hard to reason about. As a result, thorough software analysis has become crucial for assessing concepts in the field of computer architecture. The quantitative approach is widely used in the industry for processor and system design since it is the simplest and most affordable method of exploring design possibilities. Furthermore, evaluating novel, radical concepts and describing the characteristics of the design space are much more crucial in research.

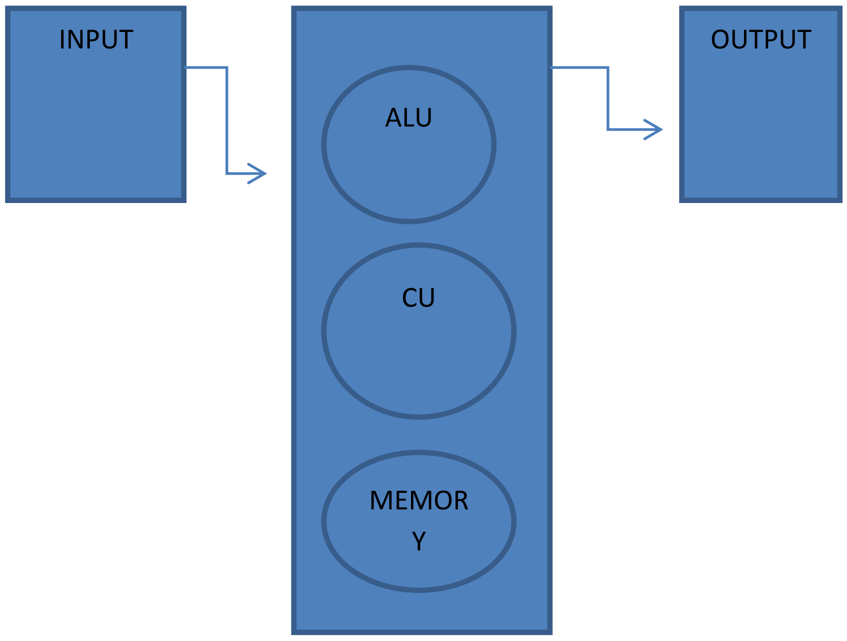


Figure 1 Basic Computer Design

Eight registers, a memory unit, and a control unit make up a basic computer. To move data between registers and between memory and registers, paths must be supplied. If connections are made between each register's output and the inputs of the other registers, there will be an excessive amount of wires. A memory unit, a control unit, and eight registers make up a basic computer. It is necessary to provide pathways for data to go between memory and registers as well as between registers. If there are connections made between each register's output and its input, there will be an excessive amount of wires.

The following hardware components are included in the basic computer:
4096 words, 16 bits per, in an eight register memory unit
Seven flip-flops
Two decryptors
A common bus with 16 bits
Logic gates for control
Logic circuit and adder coupled to the AC input.A collection of machine language instructions that a certain processor can comprehend and carry out are called computer instructions. A computer operates according to the instructions it is given.
A lesson consists of fields, which are groups. Among these fields are:
The field labeled "Opcode" indicates the operation that needs to be carried out.
The address of the operand, or register or memory location, is contained in the address field.
The operand's location is specified in the Mode field.

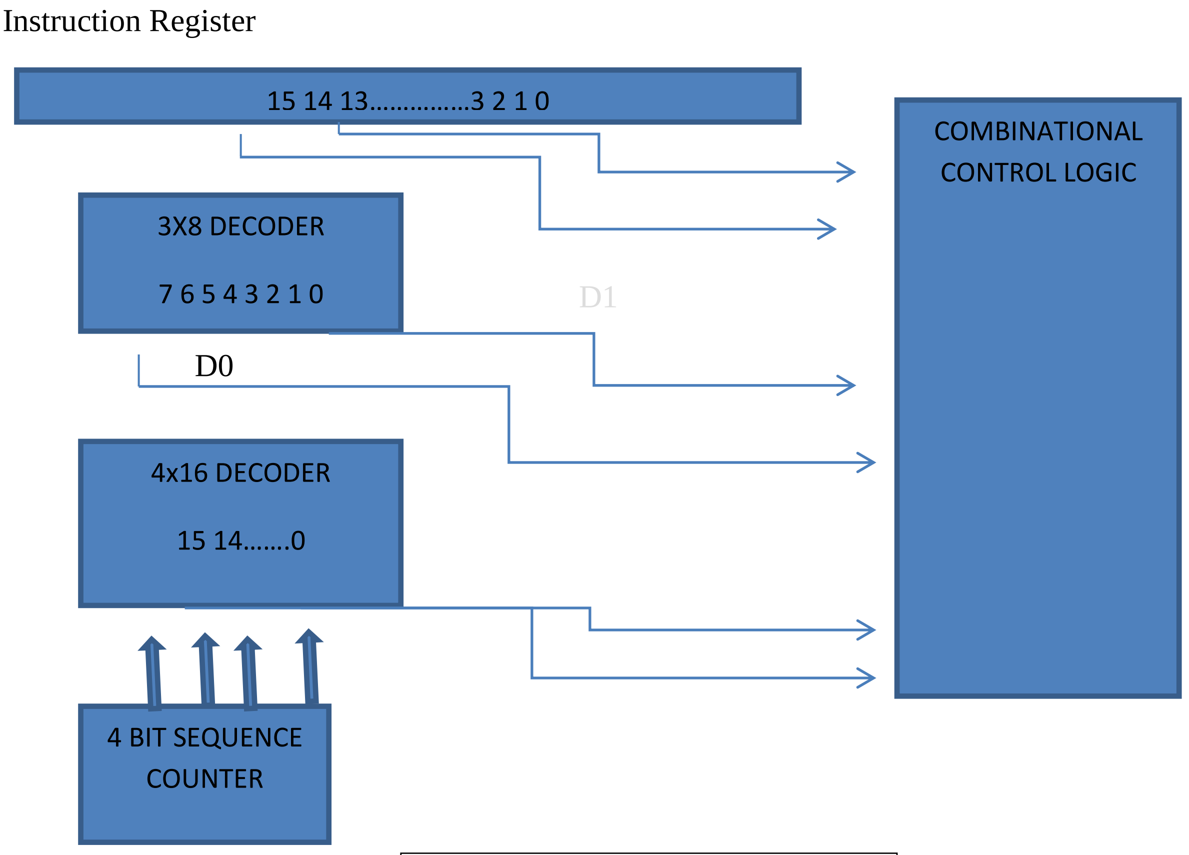


Figure 2 : Basic Computer Design

Instruction Set Completeness

If the computer has a sufficient number of instructions in each of the following categories, then the set of instructions is considered complete:

Instructions in arithmetic, logic, and shift
a collection of guidelines for transferring data between CPU registers and memory.
Instructions that verify status circumstances in addition to instructions that govern the program.
Instructions for Input and Output.Program execution order can be altered by using program control instructions like branch instructions.
The instructions for input and output serve as a conduit between the user and the computer. It is necessary to move data and programs into memory and to return computation results to the user.

Cycle of Instruction

A computer's memory unit stores programs, which are collections of instructions. The processor carries out these instructions by completing a cycle for every instruction.Each instruction cycle in a simple computer consists of the following stages:Retrieve the instruction from the memory.Interpret the instructions.From memory, read the effective address.Carry out the directive.Input-Output Configuration for the Instruction CycleCycle of Instruction A computer's memory unit stores programs, which are collections of instructions. The processor carries out these instructions by completing a cycle for every instruction. Each instruction cycle in a simple computer consists of the following stages: Retrieve the instruction from the memory. Interpret the instructions. From memory, read the effective address. Carry out the directive

Input-Output Configuration

In computer architecture, input-output devices act as an interface between the machine and the user.

Instructions and data stored in the memory must come from some input device. The results are displayed to the user through some output device.

The following block diagram shows the input-output configuration for a basic computer.

Information is sent and received via the input-output terminals.
Eight bits of an alphanumeric code will always be included in the amount of data sent.
An input register called "INPR" holds the data entered using the keyboard.
The output register, or "OUTR," contains the printer's data.
Registers INPR and OUTR exchange data in parallel with the AC and serially with a communication interface.
Information from the keyboard is received via the transmitter interface, which then sends it to INPR.
Information from OUTR is received via the receiver interface, which then serially transmits it to the printer.

Creation of a Simple Computer
The hardware parts of a basic computer are as follows.

A 4096-word memory unit with 16 bits per word
Registers include the following: PC (Program counter), TR (Temporary register), SC (Sequence Counter), IR (Instruction register), DR (Data register), AC (Accumulator), and OUTR (Output register).
Flip-Flops: IEN, FGI, FGO, S, E, R, and I

The equivalent input and output flags, FGI and FGO, are referred to as control flip-flops.
A 3 x 8 operation decoder and a 4 x 16 timing decoder are the two decoders.
A common bus with 16 bits
Logic Gate Control
The AC input is coupled to the logic and adder circuits.

Quantitative Computer Design
Principles that are helpful in computer analysis and design
Quickly create the common case!
When making a design trade-off, give preference to the frequent situation—which is typically simpler—rather than the infrequent option.

For instance, prioritize optimizing the scenario in which overflow does not occur, as overflow in addition is rare.

Goal: Identify the frequently occurring case.
Calculate the amount that performance can be improved by speeding it up.

Principles that are helpful in computer analysis and design
Quickly create the common case!
When making a design trade-off, give preference to the frequent situation—which is typically simpler—rather than the infrequent option.

For instance, prioritize optimizing the scenario in which overflow does not occur, as overflow in addition is rare.
Designing Computers Quantitatively
Amdahl's Law states that the percentage of time a quicker method of execution can be used determines how much performance can be improved.
A given feature's speedup is defined by Amdahl's law as follows:
Two elements
Fraction enhanced: The fraction of the original machine's compute time that can be converted to benefit from the improvement.
Constantly <= 1. Accelerated speed: Gained improvement through improved manner of execution:

Quantitative Computer Design

- **Amdahl's Law (cont)** :
  - *Execution time using original machine with enhancement:*

$$\text{Exec time}_{new} = \underbrace{\text{Exec time}_{old} \times \left((1 - Fraction_{enhanced})\right.}_{\text{Time spent using UNenhanced mode}} \left. + \frac{Fraction_{enhanced}}{Speedup_{enhanced}} \right)$$

Time spent using enhancement

  - *Speedup* overall *using Amdahl's Law:*

$$Speedup_{overall} = \frac{ExecTime_{old}}{ExecTime_{new}} = \frac{1}{\left((1 - Fraction_{enhanced}) + \dfrac{Fraction_{enhanced}}{Speedup_{enhanced}} \right)}$$

CPU Performance Equation:

$$\text{CPI} = \frac{\text{CPU clock cycles of a program}}{\text{IC}}$$

  - *Therefore, CPU performance is dependent on three characteristics:*

$$\text{CPU time} = \text{IC} \times \text{CPI} \times \text{Clock cycle time} = \frac{\text{IC} \times \text{CPI}}{\text{Clock rate}}$$

Equation for Quantitative Computer Design CPU Performance:
One challenge: It's challenging to alter just one without affecting the others.
Hardware and Organization clock cycle time.
CPI: Architecture of the instruction set and organization.
Instruction count: Compiler technology and instruction set architecture.

$$\text{CPU time} = \left(\sum_{i=1}^{n} CPI_i \times IC_i \right) \times \text{Clock cycle time}$$

Fallacies and Pitfalls

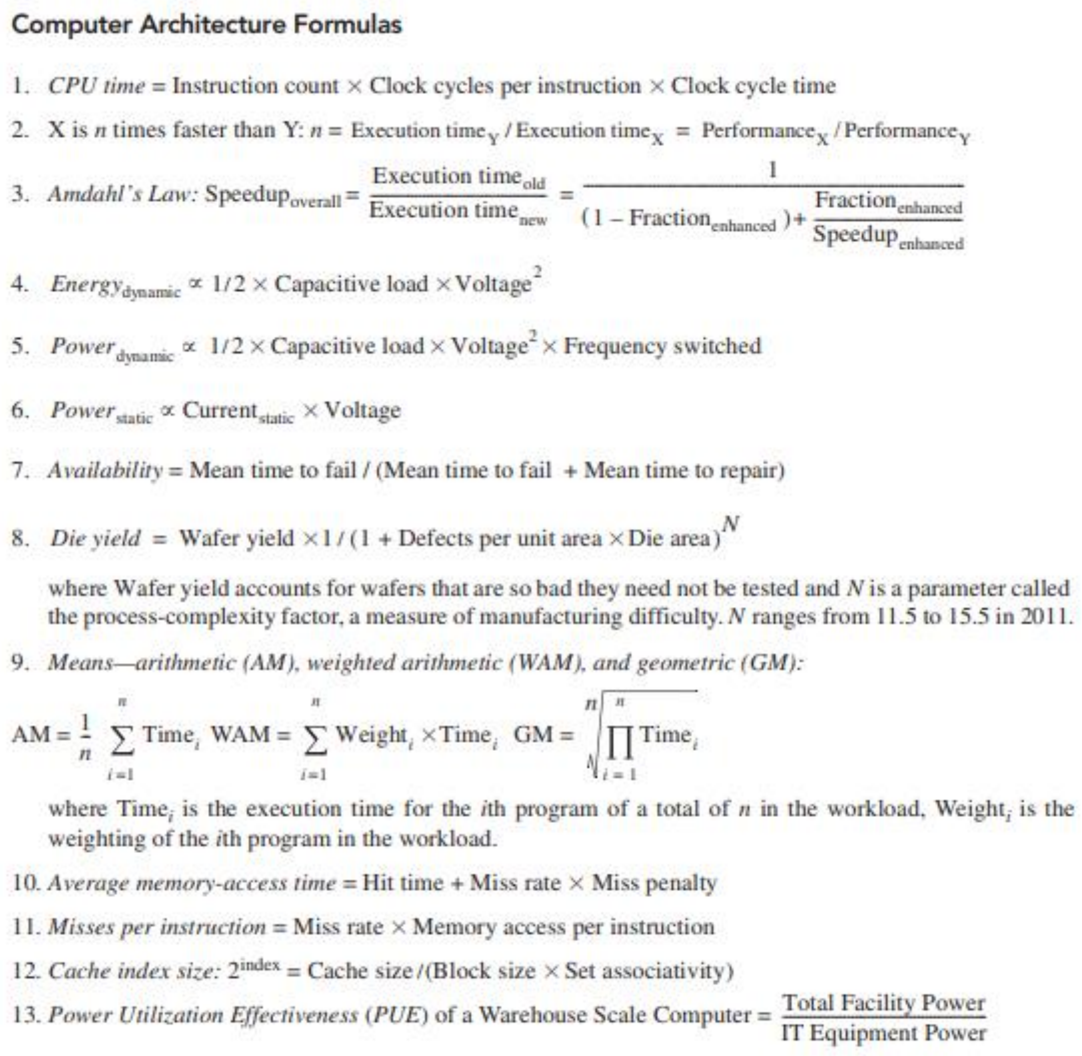
- **MIPS** *(million instruction per second) is* **NOT** *an alternative metric to time.*

$$\text{MIPS} = \frac{\text{Instruction count}}{\text{Exec Time}\times 10^6} = \frac{\text{Clock rate}}{\text{CPI}\times 10^6}$$

- *The implication: the bigger the MIPS, the faster the machine.*

The advantage of parallelism is one of the most crucial strategies for raising output.
As an illustration, consider the application of parallelism at the system level.
· numerous processors and numerous disks can be used to increase the throughput performance on a standard server benchmark, such SPECWeb or TPC-C.
· It is scalable; by distributing the load of processing requests among the disks and processors, throughput may be increased.
Utilizing instruction parallelism at the individual processor level is essential for attaining optimal performance.
Pipelining is among the easiest ways to accomplish this.

The Principal of Place
Programs frequently reuse previously used data and instructions.
Only 10% of a program's code is used for 90% of its execution time.
Based on a program's previous accesses, we can anticipate the instructions and data it will need in the near future.
Although not as strongly as it does for code accesses, this idea also applies to data accesses.
Categories: Items that have been accessed recently are probably going to be accessed again soon, according to temporal locality.
According to the theory of spatial proximity, references to entities with nearby addresses typically occur in close succession throughout time.



Computer Architecture Formulas

1. *CPU time* = Instruction count $\times$ Clock cycles per instruction $\times$ Clock cycle time
2. X is *n* times faster than Y: $n = \text{Execution time}_Y / \text{Execution time}_X = \text{Performance}_X / \text{Performance}_Y$
3. *Amdahl's Law:* $\text{Speedup}_{\text{overall}} = \dfrac{\text{Execution time}_{\text{old}}}{\text{Execution time}_{\text{new}}} = \dfrac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \dfrac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}$
4. $\textit{Energy}_{\text{dynamic}} \propto 1/2 \times \text{Capacitive load} \times \text{Voltage}^2$
5. $\textit{Power}_{\text{dynamic}} \propto 1/2 \times \text{Capacitive load} \times \text{Voltage}^2 \times \text{Frequency switched}$
6. $\textit{Power}_{\text{static}} \propto \text{Current}_{\text{static}} \times \text{Voltage}$
7. *Availability* = Mean time to fail / (Mean time to fail + Mean time to repair)
8. *Die yield* = Wafer yield $\times 1 / (1 + \text{Defects per unit area} \times \text{Die area})^N$

   where Wafer yield accounts for wafers that are so bad they need not be tested and *N* is a parameter called the process-complexity factor, a measure of manufacturing difficulty. *N* ranges from 11.5 to 15.5 in 2011.
9. *Means—arithmetic (AM), weighted arithmetic (WAM), and geometric (GM):*

$$\text{AM} = \frac{1}{n}\sum_{i=1}^{n} \text{Time}_i \quad \text{WAM} = \sum_{i=1}^{n} \text{Weight}_i \times \text{Time}_i \quad \text{GM} = \sqrt[n]{\prod_{i=1}^{n} \text{Time}_i}$$

   where Time_i is the execution time for the *i*th program of a total of *n* in the workload, Weight_i is the weighting of the *i*th program in the workload.

10. *Average memory-access time* = Hit time + Miss rate $\times$ Miss penalty
11. *Misses per instruction* = Miss rate $\times$ Memory access per instruction
12. *Cache index size:* 2^{index} = Cache size /(Block size $\times$ Set associativity)
13. *Power Utilization Effectiveness (PUE)* of a Warehouse Scale Computer = $\dfrac{\text{Total Facility Power}}{\text{IT Equipment Power}}$

The steps that an instruction goes through when it is transferred between the several processor segments—fetching, buffering, decoding, and execution—are represented by the instruction pipeline. While earlier instructions are being carried out in other segments, one section reads instructions from the memory.Pipeline for instructions

The steps that an instruction goes through when it is transferred between the several processor segments—fetching, buffering, decoding, and execution—are represented by the instruction pipeline. While earlier instructions are being carried out in other segments, one section reads instructions from the memory. The throughput of the entire system increases as a result of the overlapping nature of these activities. The instruction cycle can be divided into equal-duration parts to further boost the efficiency of the pipeline. The pipeline for arithmetic The components of an arithmetic operation that can be divided up and overlapped while being carried out are represented by the arithmetic pipeline. It can be applied to various mathematical operations, including multiplication of fixed-point integers and floating-point calculations. Any intermediate findings that are subsequently forwarded to the following step for additional processing are kept in registers.The benefits of pipelining Pipelining's primary benefit is that it shortens the processor's cycle time. This is due to its ability to process multiple instructions at once and shorten the time between finished instructions. The processor's total throughput is increased by pipelining, even though the time it takes to execute an instruction is still dependent on its size, priority, and complexity.

Moreover, the clock frequency of pipelined CPUs is typically higher than that of RAM. This enhances the system's dependability and facilitates its worldwide deployment.The benefits of pipelining Pipelining's primary benefit is that it shortens the processor's cycle time. This is due to its ability to process multiple instructions at once and shorten the time between finished instructions. The processor's total throughput is increased by pipelining, even though the time it takes to execute an instruction is still dependent on its size, priority, and complexity.

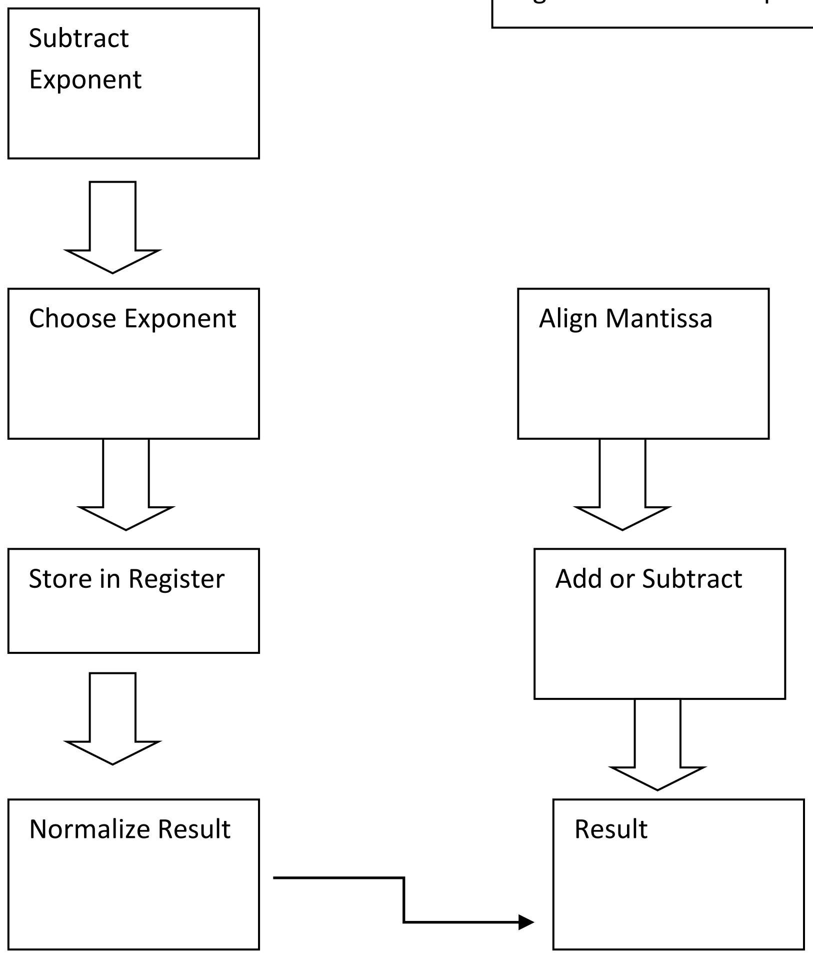
Moreover, the clock frequency of pipelined CPUs is typically higher than that of RAM. This enhances the system's dependability and facilitates its worldwide deployment. The benefits of pipelining Pipelining's primary benefit is that it shortens the processor's cycle time. This is due to its ability to process multiple instructions at once and shorten the time between finished instructions. The processor's total throughput is increased by pipelining, even though the time it takes to execute an instruction is still dependent on its size, priority, and complexity.

Moreover, the clock frequency of pipelined CPUs is typically higher than that of RAM. This enhances the system's dependability and facilitates its worldwide deployment.

Arithmetic Pipeline

An arithmetic pipeline divides an arithmetic problem into various sub problems for execution in various pipeline segments. It is used for floating point operations, multiplication and various other computations

Figure 1Arithmetic Pipeline



Instruction Pipeline:

Sequential instructions are read from memory into an instruction pipeline, with previous instructions being executed in different areas. Pipeline processing is seen in the streams of instructions and data.

Contents Table of

Section 1: Instruction Pipeline in Computer Architecture; Segments 2 through 4

Computer Architecture Instruction Pipeline
Both the data stream and the instruction stream can undergo pipeline processing. Most digital computers with complex instructions would need an instruction pipeline to carry out operations like fetching, decoding, and executing instructions.

Generally speaking, the computer needs to process each instruction in the following order:

1. Obtaining the manual from memory

2. Interpreting the acquired guidance

3. Determining the effective address

Retrieve the operands from the specified memory.

5. Carrying out the directive

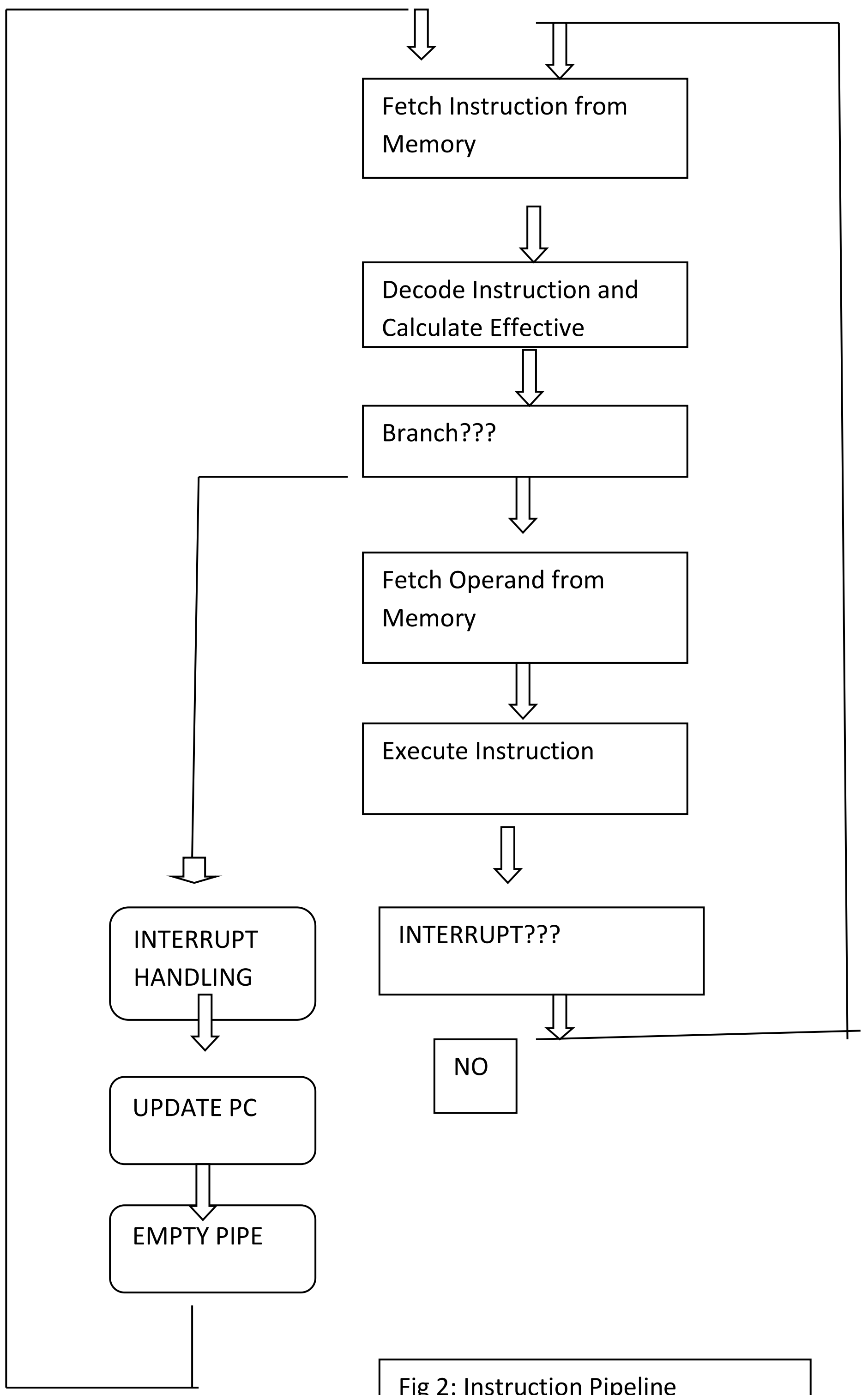
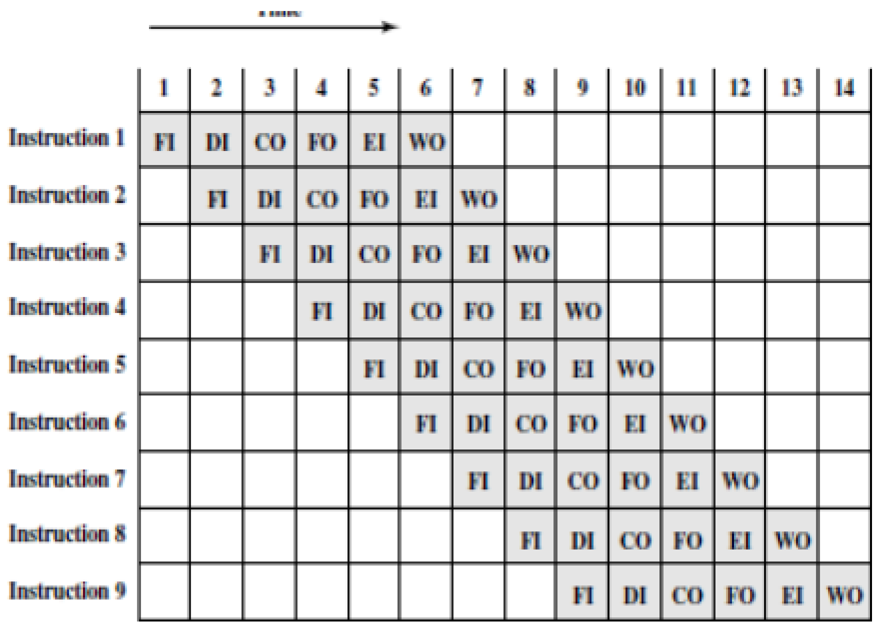
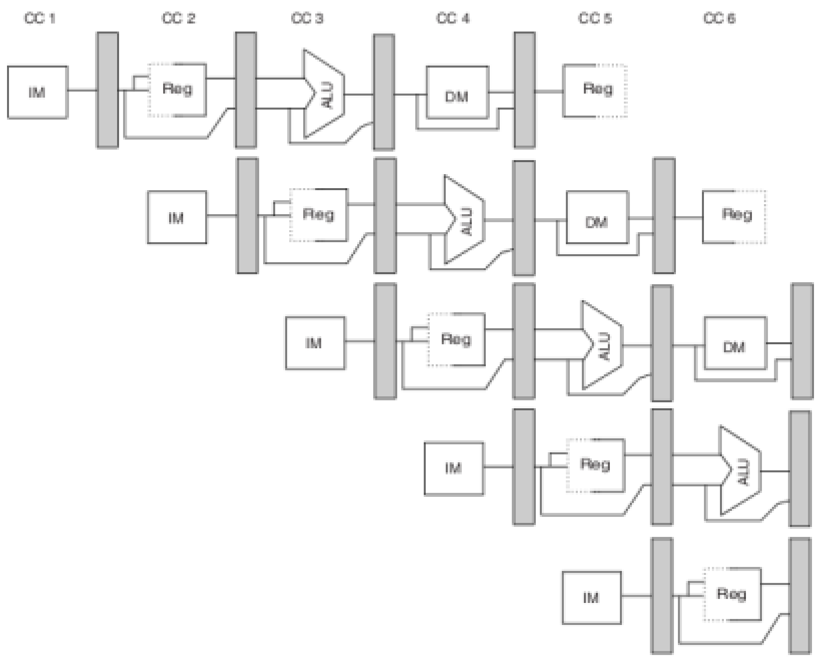

Fetch Instruction from Memory
Decode Instruction and Calculate Effective
Branch???
Fetch Operand from Memory
Execute Instruction
INTERRUPT???
NO
INTERRUPT HANDLING
UPDATE PC
EMPTY PIPE

Fig 2: Instruction Pipeline

Read the next anticipated instruction into a buffer using the fetch instruction (FI). 2. Decode instruction (DI): Find the operand specifiers and the opcode. Compute operands (CO): Determine each source operand's effective address. This could include address calculations using displacement, register indirect, indirect, or other methods. 4. Operands to be fetched (FO): Get every operand out of memory. 5. Execute instruction (EI): Carry out the given operation and, if applicable, store the outcome in the designated operand destination. 6. Write operand (WO): Retain the outcome for later use. A six-stage pipeline can cut the time it takes to execute nine instructions from 54 to 14 time units, as shown in



	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Instruction 1	FI	DI	CO	FO	EI	WO								
Instruction 2		FI	DI	CO	FO	EI	WO							
Instruction 3			FI	DI	CO	FO	EI	WO						
Instruction 4				FI	DI	CO	FO	EI	WO					
Instruction 5					FI	DI	CO	FO	EI	WO				
Instruction 6						FI	DI	CO	FO	EI	WO			
Instruction 7							FI	DI	CO	FO	EI	WO		
Instruction 8								FI	DI	CO	FO	EI	WO	
Instruction 9									FI	DI	CO	FO	EI	WO



Pipelining System

Performance of Pipelines with Stalls

A stall causes the pipeline performance to degrade from the ideal performance.

Let's look at a simple equation for finding the actual speedup from pipelining,

$$\begin{aligned}\text{Speedup from pipelining} &= \frac{\text{Average instruction time unpipelined}}{\text{Average instruction time pipelined}}\\ &= \frac{\text{CPI unpipelined} \times \text{Clock cycle unpipelined}}{\text{CPI pipelined} \times \text{Clock cycle pipelined}}\\ &= \frac{\text{CPI unpipelined}}{\text{CPI pipelined}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}\end{aligned}$$

Pipelining can be thought of as decreasing the CPI or the clock cycle time. Since it is traditional to use the CPI to compare pipelines, let's start with that assumption. The ideal CPI on a pipelined processor is almost always 1. Hence, we can compute the pipelined CPI:

$$\begin{aligned}\text{CPI pipelined} &= \text{Ideal CPI} + \text{Pipeline stall clock cycles per instruction}\\ &= 1 + \text{Pipeline stall clock cycles per instruction}\end{aligned}$$

If we ignore the cycle time overhead of pipelining and assume the stages are perfectly balanced, then the cycle time of the two processors can be equal, leading to

$$\text{Speedup} = \frac{\text{CPI unpipelined}}{1 + \text{Pipeline stall cycles per instruction}}$$

One important simple case is where all instructions take the same number of cycles, which must also equal the number of pipeline stages (also called the depth of the pipeline). In this case, the unpipelined CPI is equal to the depth of the pipeline, leading to

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall cycles per instruction}}$$

If there are no pipeline stalls, this leads to the intuitive result that pipelining can improve performance by the depth of the pipeline.

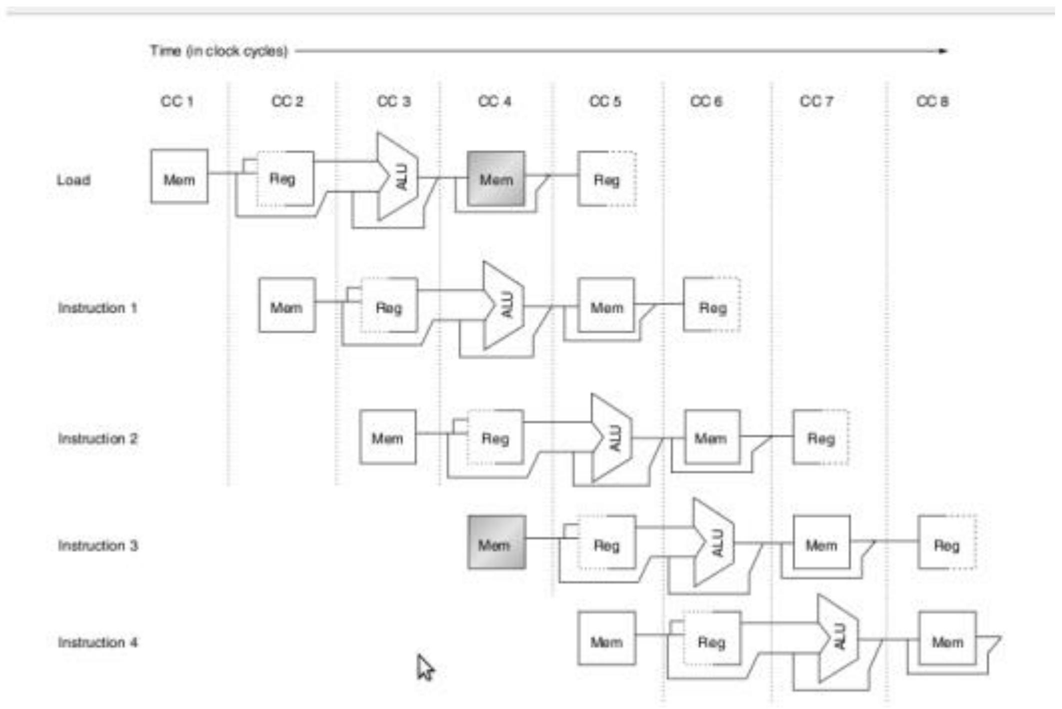
In cases where the pipe stages are perfectly balanced and there is no overhead, the clock cycle on the pipelined processor is smaller than the clock cycle of the unpipelined processor by a factor equal to the pipelined depth:

$$\text{Clock cycle pipelined} = \frac{\text{Clock cycle unpipelined}}{\text{Pipeline depth}}$$

$$\text{Pipeline depth} = \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}$$

Structural Hazards

Pipelining of functional units and duplication of resources are necessary for the overlapping execution of instructions in a pipelined processor in order to accommodate all possible combinations of instructions. A processor is said to have a structural hazard if resource conflicts prevent some combination of instructions from being executed.



Data Hazards and its Handling Methods

Read after Write (RAW), Write after Read (WAR), Write after Write (WAW), and Read after Read (RAR) are the four categories of data dependencies. Below is an explanation of each of these.

Read Following Writing (RAW):
It is sometimes referred to as flow dependency or true reliance. It happens when a later instruction needs the value that an earlier instruction provided. For instance, ADD R1, --, --; SUB --, R1, --; Stalls must manage these risks.

Write after Read (WAR): Anti-dependency is another name for it. These risks arise when an instruction uses its output register immediately after it has been read by a prior instruction. As an illustration,

Write after Write (WAW): ADD --, R1, --; SUB R1, --, --;
Another name for it is output dependence. These risks arise when an instruction's output register is written after it has been written by another instruction. For instance, Read after Read (RAR): ADD R1, --, --; SUB R1, --, --;
When two instructions read from the same register, it happens. As an illustration, ADD --, R1, --; SUB --, R1, --;
These Read after Read (RAR) risks don't affect the processor because reading a register value doesn't alter the register value.

Managing Data Hazards: We employ a number of techniques to manage data hazards, including stall insertion, code reordering, and forwarding.

Below is an explanation of each of these.

Forwarding: It enriches the pipeline with additional circuitry. The reason this method works is that the required values go via a wire faster than the pipeline segment's computation of the outcome.
Reordering codes: To reorder codes, we require a specific kind of software. This kind of software is known as a hardware-dependent compiler.
Stall Insertion: this technique reduces pipeline efficiency and throughput by inserting one or more stall (no-op) instructions into the pipeline. This delays the execution of the present instruction until the necessary operand is written to the register file.

Handling Control Hazards

This module's goals are to examine delayed branching, distinguish between static and dynamic branch prediction, and talk about how to deal with control hazards.

An issue arises when a series of instructions branches off. To maintain the pipeline, an instruction needs to be fetched every clock cycle. But until the branch is fixed, we won't know where to get the next instruction, which is problematic. Unlike the data hazards we looked at in the previous courses, this delay in figuring out which instruction to retrieve is referred to as a control hazard or branch hazard. Control dependences lead to control dangers.

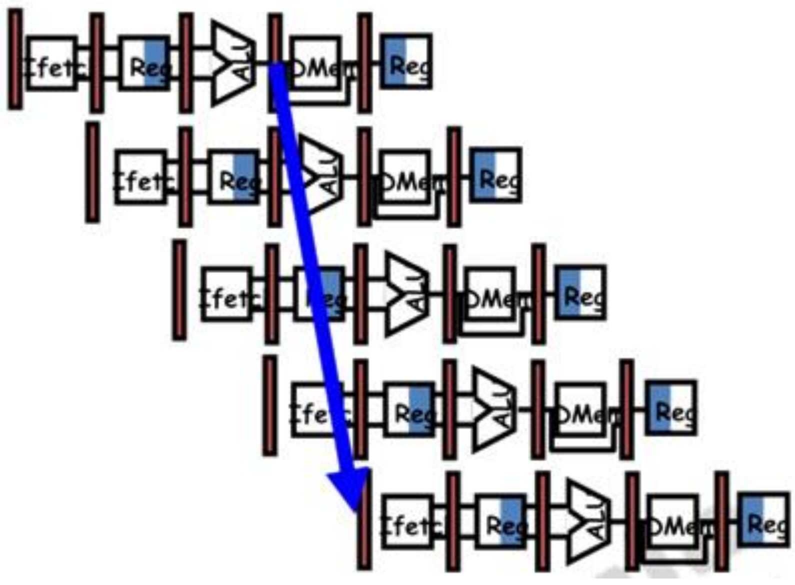


Figure Control Hazard

There are basically two ways of handling control hazards:

1. Stall until the branch outcome is known or perform the fetch again

2. Predict the behavior of branches

a. Static prediction by the compiler

b. Dynamic prediction by the hardware

Exception and Interrupt:

Unexpected events that interfere with the regular flow of instructions being executed by the processor are known as exceptions and interruptions. An unforeseen processor-related event is an exception. An unanticipated outside event interrupts the operation. The hardware begins running the code that responds to an exception or interrupt by taking action whenever one happens. This could include terminating a process, displaying an error message, establishing contact with an external device, or even bringing down the whole computer system by bringing up a “Blue Screen of Death” and stopping the CPU.

Following the handling of the exception or interrupt, the kernel proceeds as follows: Choose a process to restore and resume; restore the context of the chosen process; and resume the selected process's execution.

References :

[1] Patterson, D. A. and Hennessy L. J. 2007 , "Computer Architecture – A Quantitative Approach", Mc Graw Hill.

[2] McMahon, F. M. [1986]. "The Livermore FORTRAN kernels: A computer test of numerical performance range," Tech. Rep. UCRL-55745, Lawrence Livermore National Laboratory, Univ. of California, Livermore (December).

[3] Nairy, C., and D. Soltis [2003]. "Itanium 2 processor microarchitecture," IEEE Micro 23:2 (March–April), 44–55.

[4] Mead, C., and L. Conway [1980]. Introduction to VLSI Systems, Addison-Wesley, Reading, Mass.

[5] Mellor-Crummey, J. M., and M. L. Scott [1991]. "Algorithms for scalable synchronization on shared-memory multiprocessors," ACM Trans. on Computer Systems 9:1 (February), 21–65.

[6] Menabrea, L. F. [1842]. "Sketch of the analytical engine invented by Charles Babbage," Bibiothèque Universelle de Genève (October).

[7] Lam, M. S., E. E. Rothberg, and M. E. Wolf [1991]. "The cache performance and optimizations of blocked algorithms," Fourth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems, Santa Clara, Calif., April 8–11. SIGPLAN Notices 26:4 (April), 63–74.

[8] Jordan, H. F. [1983]. "Performance measurements on HEP—a pipelined MIMD computer," Proc. 10th Int'l Symposium on Computer Architecture (June), Stockholm, 207–212.

[9] Jordan, K. E. [1987]. "Performance comparison of large-scale scientific processors: Scalar mainframes, mainframes with vector facilities, and supercomputers," Computer 20:3 (March), 10–23. [10] Jouppi, N. P. [1990]. "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," Proc. 17th Annual Int'l Symposium on Computer Architecture, 364–73.

[11] Jouppi, N. P. [1998]. "Retrospective: Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," 25 Years of the Int'l Symposia on Computer Architecture (Selected Papers), ACM, 71–73.

[12] Jouppi, N. P., and D. W. Wall [1989]. "Available instruction-level parallelism for superscalar and superpipelined processors," Proc. Third Conf. on Architectural Support for Programming Languages and Operating Systems, IEEE/ACM (April), Boston

[13] Book : Computer Organization by Hamacher.

[14] Parallelism and pipelining by David G. Messerschmitt,University Of California.

[15] Pipelining Design Techniques by Mostafa Abd-ElBarr & Hesham El-Rewini

[16] Project Management Graphics by Edward Tufte: B.S. and M.S. in statistics, Stanford University, 1964. Ph.D. in political science, Yale University, 1968