ADVANCE COMPUTER ARCHITECTURE

UNIT-3

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.

Let us consider an example of combined multiplication and addition operation to get a better understanding of the pipeline organization.

The combined multiplication and addition operation is done with a stream of numbers such as:

$A_i * B_i + C_i$ for $i = 1, 2, 3, \ldots\ldots, 7$

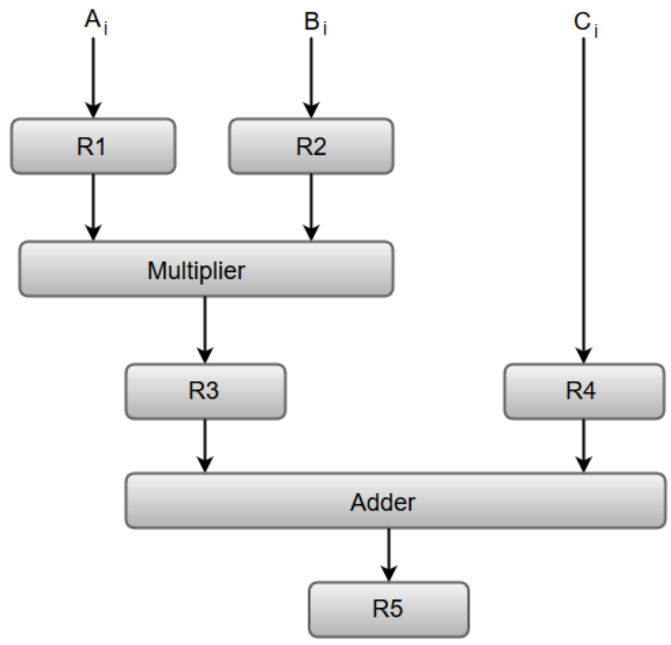
The operation to be performed on the numbers is decomposed into sub-operations with each sub-operation to be implemented in a segment within a pipeline.

The sub-operations performed in each segment of the pipeline are defined as:

$R1 \leftarrow A_i,\ R2 \leftarrow B_i$	Input A_i, and B_i
$R3 \leftarrow R1 * R2,\ R4 \leftarrow C_i$	Multiply, and input C_i
$R5 \leftarrow R3 + R4$	Add C_i to product

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

Pipeline Processing:



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.

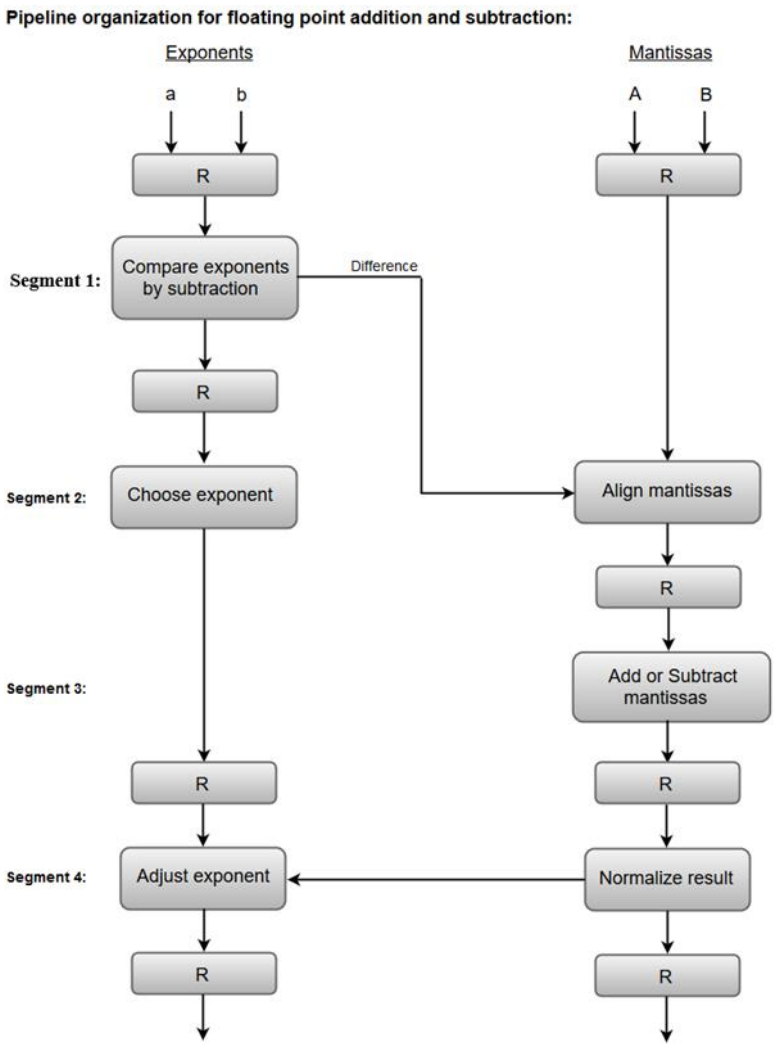
The inputs to the floating-point adder pipeline are two normalized floating-point binary numbers defined as:

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

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.



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.

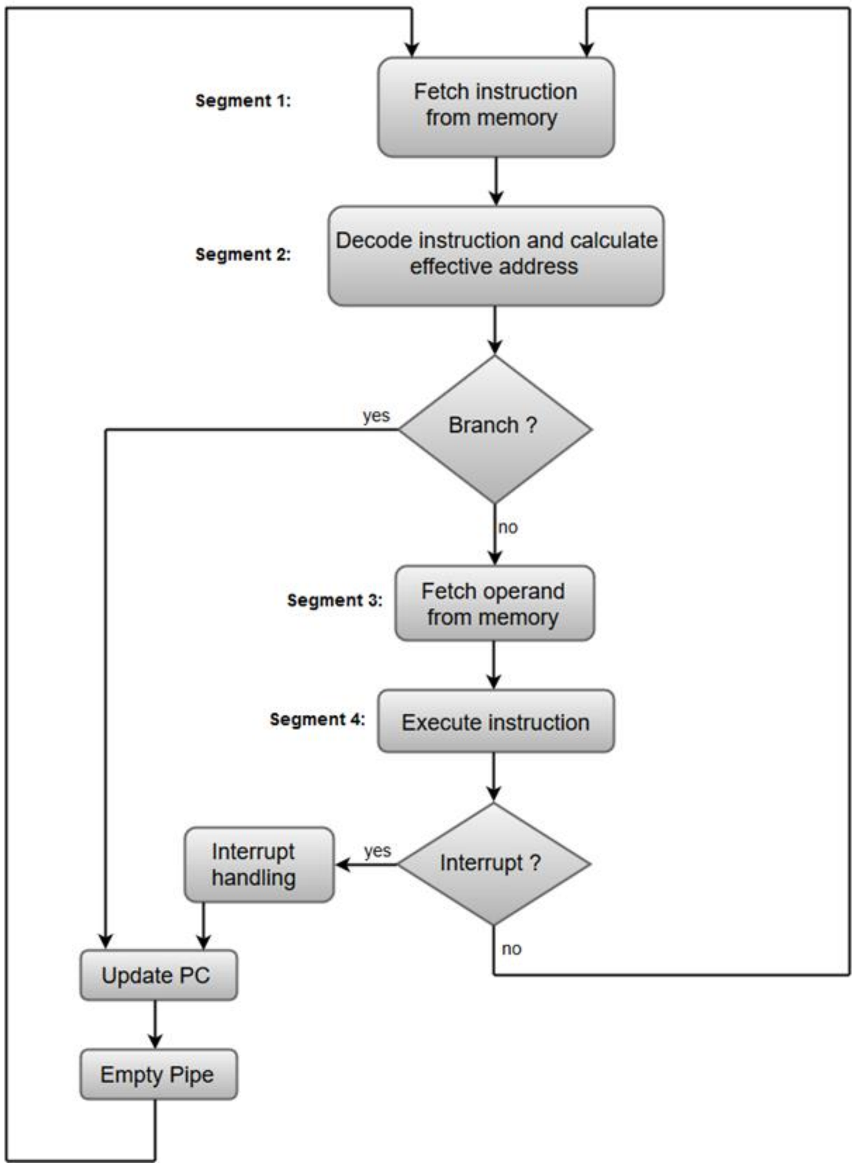
1. Fetch instruction from memory.
2. Decode the instruction.
3. Calculate the effective address.
4. Fetch the operands from memory.
5. Execute the instruction.
6. Store the result in the proper place.

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.

BASICS OF RISC ISA:

RISC-V (pronounced "risk-five") is a ISA standard

– An open source implementation of a reduced instruction set computing (RISC) based instruction set architecture (ISA)

– There was RISC-I, II, III, IV before

• Most ISAs: X86, ARM, Power, MIPS, SPARC

– Commercially protected by patents

– Preventing practical efforts to reproduce the computer systems.

• RISC-V is open – Permitting any person or group to construct compatible computers

– Use associated software

• Originated in 2010 by researchers at UC Berkeley – Krste Asanović, David Patterson and students

• 2017 version 2 of the user space ISA is fixed

– User-Level ISA Specification v2.2

– Draft Compressed ISA Specification v1.79

– Draft Privileged ISA Specification v1.10

Goals in Defining RISC-V:

• A completely open ISA that is freely available to academia and industry

• A real ISA suitable for direct native hardware implementation, not just simulation or binary translation

• An ISA that avoids "over-architecting" for – a particular microarchitecture style (e.g., micro coded, in-order, decoupled, out-of order) or – implementation technology (e.g., full-custom, ASIC, FPGA), but which allows efficient implementation in any of these

• RISC-V ISA includes – A small base integer ISA, usable by itself as a base for customized accelerators or for educational purposes, and – Optional standard extensions, to support general-purpose software development – Optional customer extensions

• Support for the revised 2008 IEEE-754 floating-point standard

RISC-V ISA Principles:

• Generally kept very simple and extendable

• Separated into multiple specifications – User-Level ISA spec (compute instructions) – Compressed ISA spec (16-bit instructions) – Privileged ISA spec (supervisor-mode instructions) – More …

• ISA support is given by RV + word-width + extensions supported – E.g. RV32I means 32-bit RISC-V with support for the (integer) instruction set

User Level ISA:

• Defines the normal instructions needed for computation

– A mandatory Base integer ISA

• I: Integer instructions:

– ALU

– Branches/jumps

– Loads/stores

– Standard Extensions

• M: Integer Multiplication and Division

• A: Atomic Instructions

• F: Single-Precision Floating-Point

• D: Double-Precision Floating-Point

• C: Compressed Instructions (16 bit)

• G = IMAFD: Integer base + four standard extensions – Optional extensions

– Optional extensions

RISC-V ISA:

• Both 32-bit and 64-bit address space variants – RV32 and RV64

• Easy to subset/extend for education/research – RV32IM, RV32IMA, RV32IMAFD, RV32G

RV32/64 Processor State:

• Program counter (pc)

• 32 32/64-bit integer registers (x0-x31) – x0 always contains a 0 – x1 to hold the return address on a call.

• 32 floating-point (FP) registers (f0-f31) – Each can contain a single- or double-precision FP value (32-bit or 64-bit IEEE FP)

• FP status register (fsr), used for FP rounding mode & exception reporting

RISC-V I-Format Instructions:

• Immediate arithmetic and load instructions – rs1: source or base address register number – immediate: constant operand, or offset added to base address

• 2s-complement, sign extended

• Design Principle: Good design demands good compromises – Different formats complicate decoding, but allow 32-bit instructions uniformly – Keep formats as similar as possible

• Different immediate format for store instructions – rs1: base address register number – rs2: source operand register number – immediate: offset added to base address

• Split so that rs1 and rs2 fields always in the same place

Classic five stage pipeline for a RISC processor:

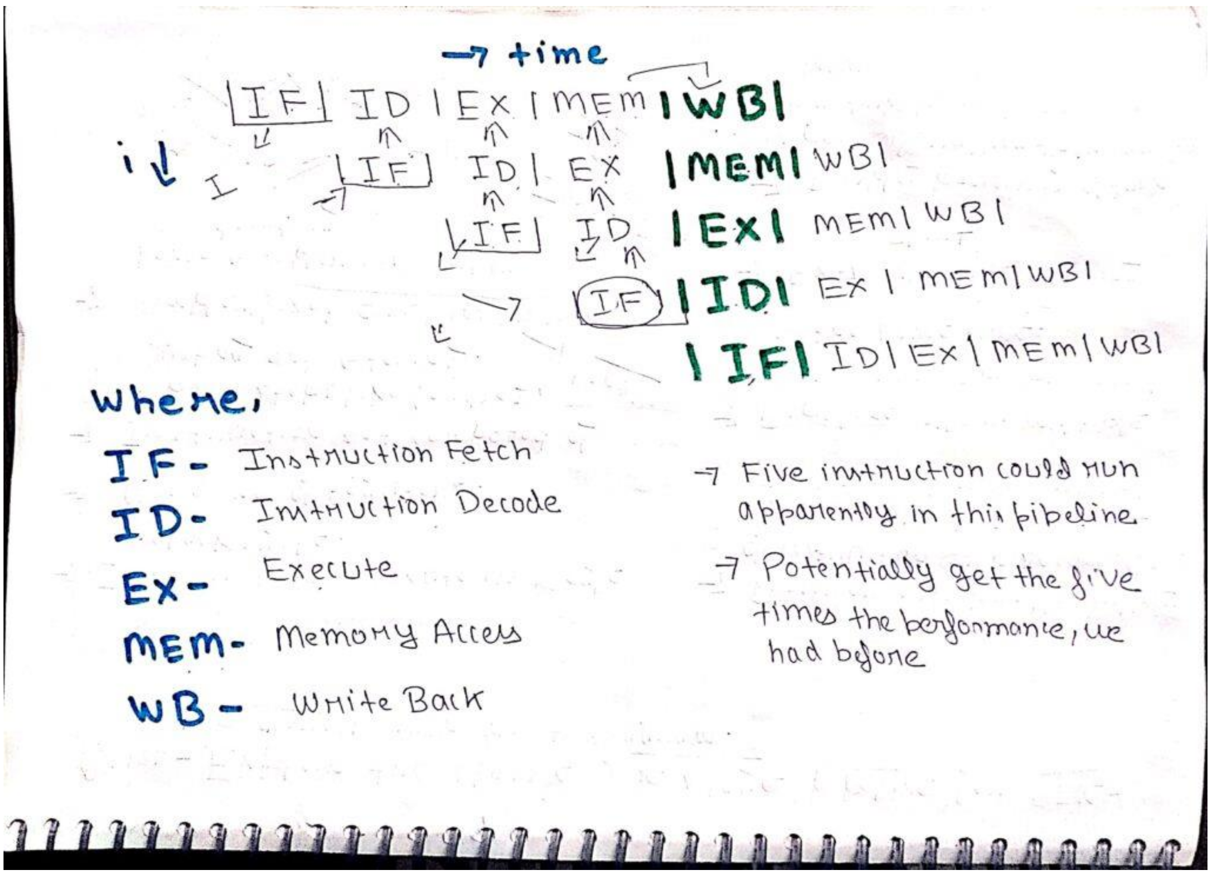

→ time
IF | ID | EX | MEM | WB |
i ↓ I
IF | ID | EX | MEM | WB |
IF | ID | EX | MEM | WB |
IF | ID | EX | MEM | WB |
IF | ID | EX | MEM | WB |
where,
IF - Instruction Fetch
ID - Instruction Decode
Ex - Execute
MEM - Memory Access
WB - Write Back
→ Five instruction could run apparently in this pipeline
→ Potentially get the five times the performance, we had before

Fig-1 – Diagram of five stage pipeline for RISC Processor

As shown in diagram Fig-1 above We have five Instructions that are :-

1. IF – Instruction Fetch – In this stage the CPU reads instructions from the address in the memory whose value is present in the program counter.
2. ID – Instruction Decode – In this stage, instruction is decoded and the register file is accessed to get the values from the registers used in the instruction.
3. EX- Execute – In this stage, ALU operations are performed.
4. MEM- Memory Access – In this stage, memory operands are read and written from/to the memory that is present in the instruction.
5. WB- Write Back – In this stage, computed/fetched value is written back to the register present in the instructions.

- As shown in Fig-1 **five instructions could run simultaneously in this Pipeline.**
- Which Potentially give the five times the performance that it have before.
- As shown in Fig-1, Time is in Horizontal axes and instructions are in verical axes.

1-First bit of time

- As shown in Fig-1, In first bit of time or first block of time only **IF** work , as it is used to fetch the instructions.
- In first bit of time , no other operation are performing as shown in Fig-1.

2- Second bit of time

- In Second bit of time,**ID** decode the instruction which are fetched and apparently new instruction are fetched by **IF**.
- As shown in Fig-1 , in second bit of time two operations are performing simultaneously.

3-Third bit of time

- In third bit of time as shown in fig-1, Execution take place by **EX** of the decoded operands which are decoded in second bit of time.
- At the same time , the instruction which is fetched in second block of time by **IF** is now going to decode by **ID**.
- A new instructions are fetched by **IF**.
- Three Operations are running now simultaneously in Third bit of time.

4-Fourth bit of time

- Four Operations **IF,ID,EX,MEM** are performed simultaneously in fourth bit of time.
- **IF** fetches new instruction, whereas **ID,EX** and **MEM** operation are performed on the result or output generated in third bit of time.

5-Fifth bit of time

- All the operations are performed simultaneously in fifth bit of time as shown in Fig-1.
- The result are written on register by **WB** in fifth bit of time.

1. At a time when instruction is ***WB***, means write back in which processor are writing result on register than at same time different processor are accessing memory by ***MEM***
2. At same time different processor are executing by ***EX*** than at a same time different instruction are decoding by ***ID***.

As shown in diagram five instructions are executing parallelly at same time in Processor, simultaneous execution of more than one instruction takes place in a pipelined processor.

Performance issues in Pipelining:

Performance in an unpipelined processor is characterized by the cycle time and the execution time of the instructions. In the case of pipelined execution, instruction process

sing is interleaved in the pipeline rather than performed sequentially as in non-pipelined processors. Therefore the concept of the execution time of instruction has no meaning, and the in-depth performance specification of a pipelined processor requires three different measures: the cycle time of the processor and the latency and repetition rate values of the instructions.

The **cycle time** defines the time accessible for each stage to accomplish the important operations. The cycle time of the processor is specified by the worst-case processing time of the highest stage.

Latency defines the amount of time that the result of a specific instruction takes to become accessible in the pipeline for subsequent dependent instruction. Latency is given as multiples of the cycle time.

If the latency of a particular instruction is one cycle, its result is available for a subsequent RAW-dependent instruction in the next cycle. In this case, a RAW-dependent instruction can be processed without any delay. If the latency is more than one cycle, say n-cycles an immediately following RAW-dependent instruction has to be interrupted in the pipeline for n-1 cycles.

There are two different kinds of RAW dependency such as define-use dependency and load-use dependency and there are two corresponding kinds of latencies known as define-use latency and load-use latency.

The **define-use latency** of instruction is the time delay occurring after decoding and issue until the result of an operating instruction becomes available in the pipeline for subsequent RAW-dependent instructions. If the value of the define-use latency is one cycle, and immediately following RAW-dependent instruction can be processed without any delay in the pipeline.

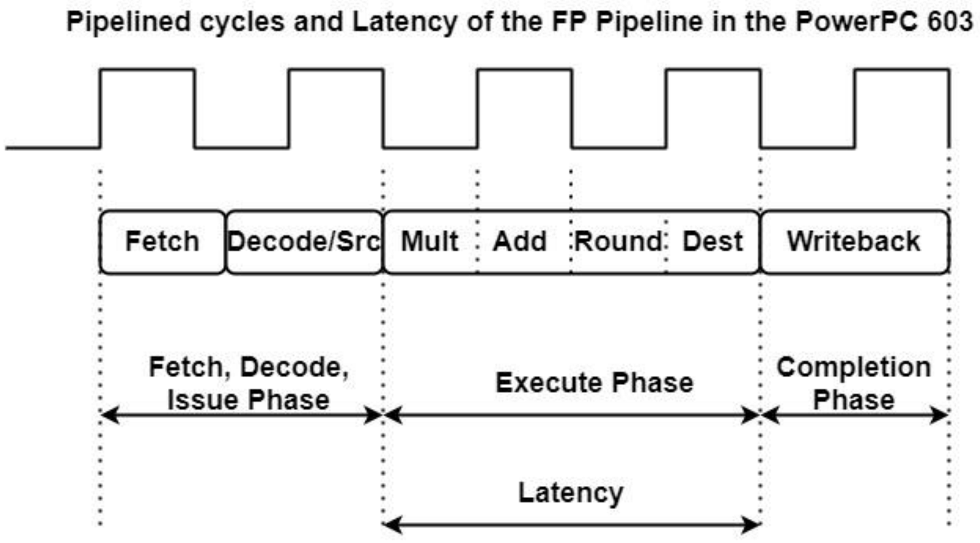
The **define-use delay** of instruction is the time a subsequent RAW-dependent instruction has to be interrupted in the pipeline. The define-use delay is one cycle less than the define-use latency.

The term load-use latency**load-use latency** is interpreted in connection with load instructions, such as in the sequence

```
load r1,  x;
ad  r5,  r1,  r2;
```

In this example, the result of the load instruction is needed as a source operand in the subsequent ad. The notion of **load-use latency** and **load-use delay** is interpreted in the same way as define-use latency and define-use delay.

The latency of an instruction being executed in parallel is determined by the execute phase of the pipeline. It can illustrate this with the FP pipeline of the PowerPC 603 which is shown in the figure.


Pipelined cycles and Latency of the FP Pipeline in the PowerPC 603
Fetch
Decode/Src
Mult
Add
Round
Dest
Writeback
Fetch, Decode, Issue Phase
Execute Phase
Completion Phase
Latency

The Power PC 603 processes FP additions/subtraction or multiplication in three phases. Two cycles are needed for the instruction fetch, decode and issue phase. The subsequent execution phase takes three cycles. At the end of this phase, the result of the operation is forwarded (bypassed) to any requesting unit in the processor. Finally, in the completion phase, the result is written back into the architectural register file.

Pipeline Hazards:

As we all know, the CPU's speed is limited by memory. There's one more case to consider, i.e. a few instructions are at some stage of execution in a pipelined design. There is a chance that these sets of instructions will become dependent on one another, reducing the pipeline's pace. Dependencies arise for a variety of reasons, which we will examine shortly. The dependencies in the pipeline are referred to as hazards since they put the execution at risk.

We can swap the terms, dependencies and hazards since they are used interchangeably in computer architecture. A hazard, in essence, prevents an instruction present in the pipe from being performed during the specified clock cycle. Since each of the instructions may be in a separate machine cycle, we use the term clock cycle.

The three different types of hazards in computer architecture are:

1. Structural

2. Data

3. Control

Dependencies can be addressed in a variety of ways. The easiest is to introduce a bubble into the pipeline, which stalls it and limits throughput. The bubble forces the next instruction to wait until the previous one is completed.

Structural Hazard

Hardware resource conflicts among the instructions in the pipeline cause structural hazards. Memory, a GPR Register, or an ALU might all be used as resources here. When more than one instruction in the pipe requires access to the very same resource in the same clock cycle, a resource conflict is said to arise. In an overlapping pipelined execution, this is a circumstance where the hardware cannot handle all potential combinations.

Data Hazards

Data hazards in pipelining emerge when the execution of one instruction is dependent on the results of another instruction that is still being processed in the pipeline. The order of the READ or WRITE operations on the register is used to classify data threats into three groups.

Control Hazards

Branch hazards are caused by branch instructions and are known as control hazards in computer architecture. The flow of program/instruction execution is controlled by branch instructions. Remember that conditional statements are used in higher-level languages for iterative loops and condition testing (correlate with while, for, and if case statements). These are converted into one of the BRANCH instruction variations. As a result, when the decision to execute one instruction is reliant on the result of another instruction, such as a conditional branch, which examines the condition's consequent value, a conditional hazard develops.

Example:(structure hazards)

Instruction / Cycle	1	2	3	4	5
I_1	IF(Mem)	ID	EX	Mem	
I_2		IF(Mem)	ID	EX	
I_3			IF(Mem)	ID	EX
I_4				IF(Mem)	ID

In the above scenario, in cycle 4, instructions I1 and I4 are trying to access same resource (Memory) which introduces a resource conflict.

To avoid this problem, we have to keep the instruction on wait until the required resource (memory in our case) becomes available. This wait will introduce stalls in the pipeline as shown below:

Cycle	1	2	3	4	5	6	7	8
I_1	IF(Mem)	ID	EX	Mem	WB			
I_2		IF(Mem)	ID	EX	Mem	WB		
I_3			IF(Mem)	ID	EX	Mem	WB	
I_4				–	–	–	IF(Mem)	

Solution for structural dependency

To minimize structural dependency stalls in the pipeline, we use a hardware mechanism called Renaming.

Renaming:
According to renaming, we divide the memory into two independent modules used to store the instruction and data separately called Code memory(CM) and Data memory(DM) respectively. CM will contain all the instructions and DM will contain all the operands that are required for the instructions.

Instruction / Cycle	1	2	3	4	5	6	7
I_1	IF(CM)	ID	EX	DM	WB		
I_2		IF(CM)	ID	EX	DM	WB	
I_3			IF(CM)	ID	EX	DM	WB
I_4				IF(CM)	ID	EX	DM
I_5					IF(CM)	ID	EX
I_6						IF(CM)	ID

Instruction / Cycle	1	2	3	4	5	6	7
I_7							IF(CM)

Control Dependency (Branch Hazards):

This type of dependency occurs during the transfer of control instructions such as BRANCH, CALL, JMP, etc. On many instruction architectures, the processor will not know the target address of these instructions when it needs to insert the new instruction into the pipeline. Due to this, unwanted instructions are fed to the pipeline.

Consider the following sequence of instructions in the program:
100: I1
101: I2 (JMP 250)
102: I3
.
.
250: BI1
Expected output: I1 -> I2 -> BI1
NOTE: Generally, the target address of the JMP instruction is known after ID stage only.

Instruction/ Cycle	1	2	3	4	5	6
I_1	IF	ID	EX	MEM	WB	
I_2		IF	ID (PC:250)	EX	Mem	WB
I_3			IF	ID	EX	Mem
BI_1				IF	ID	EX

Output Sequence: I1 -> I2 -> I3 -> BI1
So, the output sequence is not equal to the expected output, that means the pipeline is not implemented correctly.

To correct the above problem we need to stop the Instruction fetch until we get target address of branch instruction. This can be implemented by introducing delay slot until we get the target address.

Instruction/ Cycle	1	2	3	4	5	6
I_1	IF	ID	EX	MEM	WB	
I_2		IF	ID (PC:250)	EX	Mem	WB
Delay	–	–	–	–	–	–
BI_1				IF	ID	EX

Output Sequence: I1 -> I2 -> Delay (Stall) -> BI1
As the delay slot performs no operation, this output sequence is equal to the expected output sequence. But this slot introduces stall in the pipeline.

Solution for Control dependency:
Branch Prediction is the method through which stalls due to control dependency can be eliminated. In this at 1st stage prediction is done about which branch will be taken.For branch prediction Branch penalty is zero.

Branch penalty:
The number of stalls introduced during the branch operations in the pipelined processor is known as branch penalty.

NOTE:
As we see that the target address is available after the ID stage, so the number of stalls introduced in the pipeline is 1. Suppose the branch target address would have been present after the ALU stage, there would have been 2 stalls. Generally, if the target address is present after the kth stage, then there will be (k – 1) stalls in the pipeline.
Total number of stalls introduced in the pipeline due to branch instructions = **Branch frequency * Branch Penalty**

Data Dependency (Data Hazard)

Let us consider an ADD instruction S, such that
S : ADD R1, R2, R3
Addresses read by S = I(S) = {R2, R3}
Addresses written by S = O(S) = {R1}
Now, we say that instruction S2 depends in instruction S1, when

$$[I(S1) \cap O(S2)] \cup [O(S1) \cap I(S2)] \cup [O(S1) \cap O(S2)] \neq \phi$$

This condition is called Bernstein condition.
Three cases exist:

- Flow (data) dependence: O(S1) ∩ I (S2), S1 → S2 and S1 writes after something read by S2
- Anti-dependence: I(S1) ∩ O(S2), S1 → S2 and S1 reads something before S2 overwrites it
- Output dependence: O(S1) ∩ O(S2), S1 → S2 and both write the same memory location.

Example: Let there be two instructions I1 and I2 such that:
I1: ADD R1, R2, R3
I2 : SUB R4, R1, R2

When the above instructions are executed in a pipelined processor, then data dependency condition will occur, which means that I2 tries to read the data before I1 writes it, therefore, I2 incorrectly gets the old value from I1.

Instruction / Cycle	1	2	3	4
I_1	IF	ID	EX	DM
I_2		IF	ID(Old value)	EX

To minimize data dependency stalls in the pipeline, operand forwarding is used.
Operand Forwarding:
In operand forwarding, we use the interface registers present between the stages to hold intermediate output so that dependent instruction can access new value from the interface register directly.
Considering the same example:
I1 : ADD R1, R2, R3
I2 : SUB R4, R1, R2

Instruction / Cycle	1	2	3	4
I_1	IF	ID	EX	DM
I_2		IF	ID	EX

Data Hazards
Data hazards occur when instructions that exhibit data dependence, modify data in different stages of a pipeline. Hazard cause delays in the pipeline. There are mainly three types of data hazards:

1) RAW (Read after Write) [Flow/True data dependency]
2) WAR (Write after Read) [Anti-Data dependency]
3) WAW (Write after Write) [Output data dependency]

Let there be two instructions I and J, such that J follow I. Then,

- RAW hazard occurs when instruction J tries to read data before instruction I writes it.
  Eg:
  I: R2 <- R1 + R3
  J: R4 <- R2 + R3
- WAR hazard occurs when instruction J tries to write data before instruction I reads it.
  Eg:
  I: R2 <- R1 + R3
  J: R3 <- R4 + R5
- WAW hazard occurs when instruction J tries to write output before instruction I writes it.
  Eg:
  I: R2 <- R1 + R3
  J: R2 <- R4 + R5

WAR and WAW hazards occur during the out-of-order execution of the instructions.