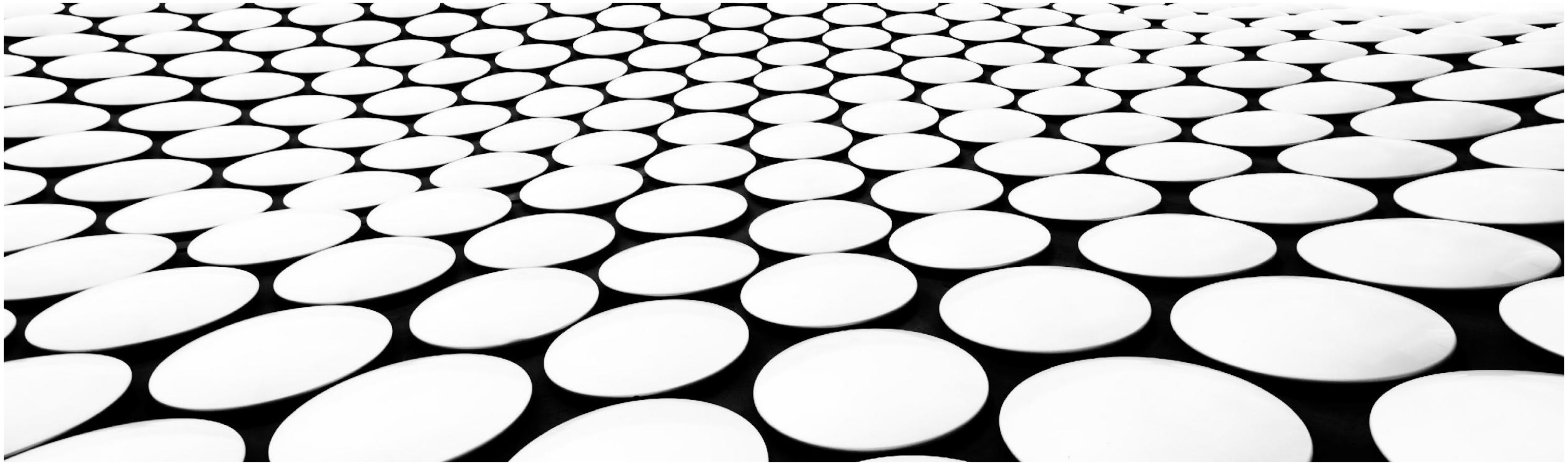
PIPELINING AND HAZARDS

UNIT - IV



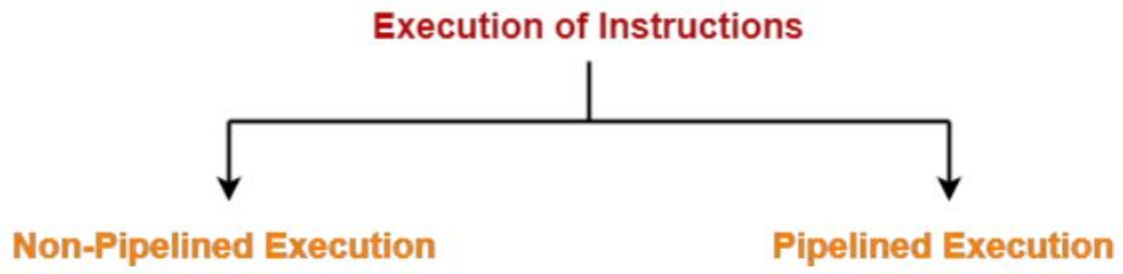
CONTENT

1. Introduction of Pipelining
2. Instruction Pipelining
3. Arithmetic Pipelining
4. Pipelining Hazards
5. Numerical on Pipelining

1. INTRODUCTION OF PIPELINING

INTRODUCTION

- A program consists of several number of instructions. These instructions may be executed in the following two-way:



- The primary goal of computer architecture is to enhance the performance and speed of the computer. This can be achieved by:
  - Improving the hardware
  - Arranging the hardware so that multiple operations can be performed simultaneously.

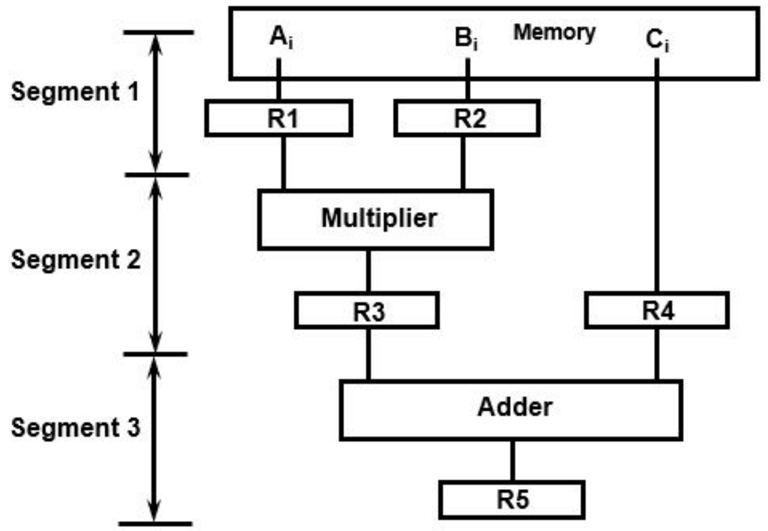
INTRODUCTION

- In **non-pipelined (sequential) architecture**, all the instructions of a program are executed sequentially one after the other
- **Pipelining is referred as**
  - A technique in which a given task is divided into a number of subtasks that need to be performed in sequence.
  - One of the processes of arranging the hardware so that simultaneous execution of multiple instructions takes place, thus, improving the overall performance.
- The main advantage of pipelining is the simultaneous execution of various subtasks, which improves the system's throughput.

PIPELINING

- A technique of decomposing a sequential process into suboperations, with each subprocess being executed in a partially dedicated segment that operates concurrently with all other segments.
- **Example:**

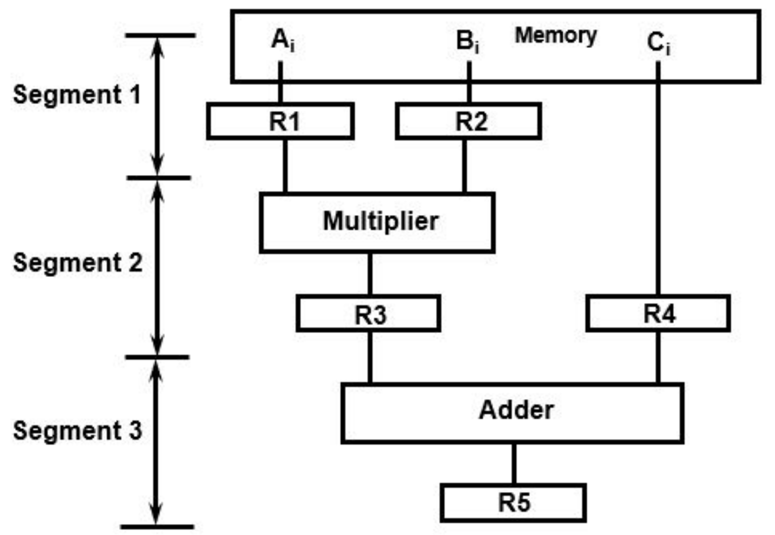
$A_i * B_i + C_i$ for $i = 1, 2, 3, \ldots, 7$



$R1 \leftarrow A_i$, $R2 \leftarrow B_i$	Load A_i and B_i
$R3 \leftarrow R1 * R2$, $R4 \leftarrow C_i$	Multiply and load C_i
$R5 \leftarrow R3 + R4$	Add

OPERATIONS IN EACH PIPELINING STAGE

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



$R1 \leftarrow A_i,\ R2 \leftarrow B_i$	Load A_i and B_i
$R3 \leftarrow R1 * R2,\ R4 \leftarrow C_i$	Multiply and load C_i
$R5 \leftarrow R3 + R4$	Add

Pipelined Execution

Clock Pulse Number	Segment 1		Segment 2		Segment 3
	R1	R2	R3	R4	R5
1	A1	B1			
2	A2	B2	A1 * B1	C1	
3	A3	B3	A2 * B2	C2	A1 * B1 + C1
4	A4	B4	A3 * B3	C3	A2 * B2 + C2
5	A5	B5	A4 * B4	C4	A3 * B3 + C3
6	A6	B6	A5 * B5	C5	A4 * B4 + C4
7	A7	B7	A6 * B6	C6	A5 * B5 + C5
8			A7 * B7	C7	A6 * B6 + C6
9					A7 * B7 + C7

Non-Pipelined Execution: 3*7 = 21 Clock pulses

INSTRUCTION CYCLE

- There are several stages of processing an instruction. A pipeline can be of three, four, five, or six stages.

3-stage

Fetch
Decode
Execute

4-stage

Fetching the instruction
Decoding the instruction
Executing the instruction
Write Back

5-stage

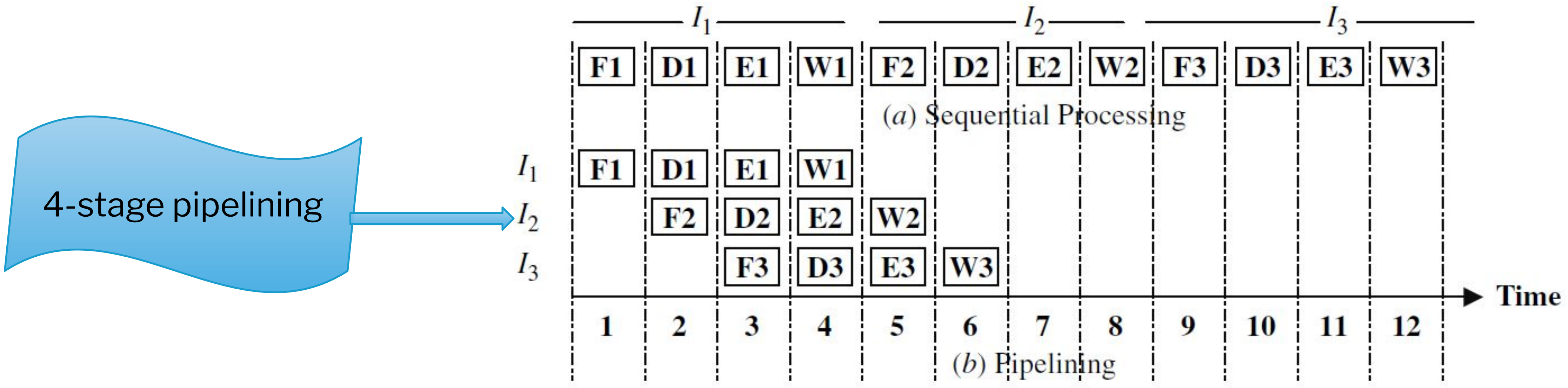
Fetching the instruction
Decoding the instruction
Memory Access for operands
Executing the instruction
Write Back

6-stage

Fetching the instruction
Decoding the instruction
Calculate the effective address of the operand
Memory Access for operands
Executing the instruction
Write Back

SEQUENTIAL VS. PIPELINED EXECUTION OF INSTRUCTIONS

- Consider that there are three instructions- I1, I2, and I3 and there are 4 stages of execution – Fetch (F), Decode (D), Execute (E), and Write back (W).
- It takes **12 machine cycles** to execute these three instructions in **sequential processing** and only **6 machine cycles** in **pipelining.**

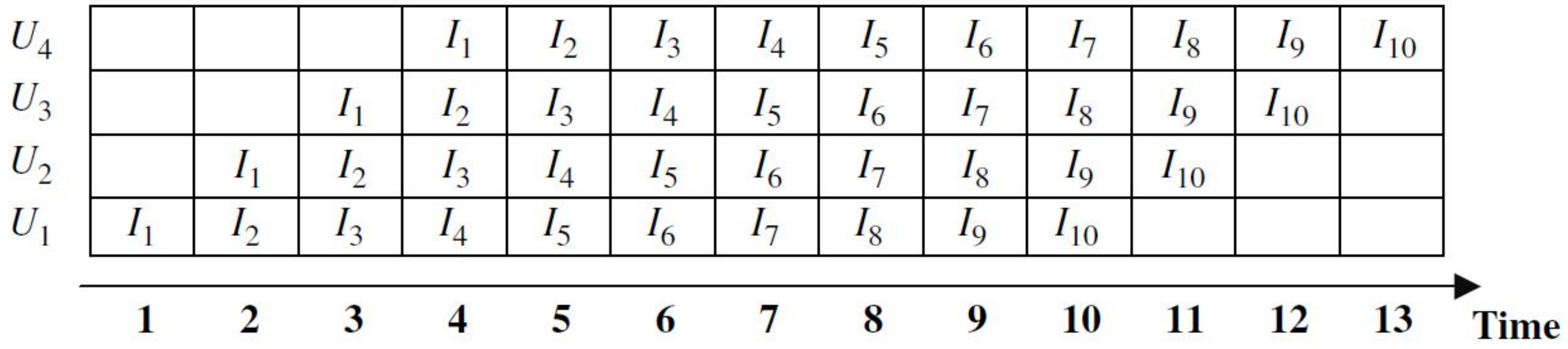


Pipelining versus sequential processing

PERFORMANCE MEASURES FOR THE GOODNESS OF A PIPELINE

- In order to formulate the performance measures for the goodness of a pipeline in processing a series of instructions
  - A space-time chart (called the Gantt's chart) is used.
  - In this chart, the vertical axis represents the segments (four in this case) and the horizontal axis represents time (the time (T) taken by each subunit to perform its task is the same, therefore, known as unit time)

13 time units are required to execute 10 instructions using 4-stage pipelining



	1	2	3	4	5	6	7	8	9	10	11	12	13
U_4				I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	I_9	I_{10}
U_3			I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	I_9	I_{10}	
U_2		I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	I_9	I_{10}		
U_1	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	I_9	I_{10}			

The space–time chart (Gantt chart)

PERFORMANCE MEASURES FOR THE GOODNESS OF A PIPELINE

SPEED-UP S(N)

- Consider the execution of *n* instructions using the *k*-segments pipeline.
- **In pipelined execution**, the first instruction will take T clock cycles to process, but the other instructions will take just 1 more clock cycle.
- **The speedup can be calculated as:**

Speedup (S) = Cycles of non-pipelined processor/ cycle of pipelined processor

- $$S = \frac{n * t_n}{(k + n - 1) * t_p}$$

n = number of tasks

k = k = number of segments pipeline

t_n = clock cycle time (non-pipelined)

t_p = clock cycle time (pipelined)

SPEED-UP S(N)

- As the number of tasks increases,
- Take the limit as n approaches to infinity, (n+k-1) approaches to n, resulting in theoretical speedup (max) of:
- $$\lim \quad n \rightarrow \infty \quad Smax = \frac{t_n}{t_p}$$

If time it takes to process a task is the same in pipelined and non-pipelined circuits then,

$$Smax = \frac{k * t_p}{t_p} = k$$

Mainly two types of pipelining

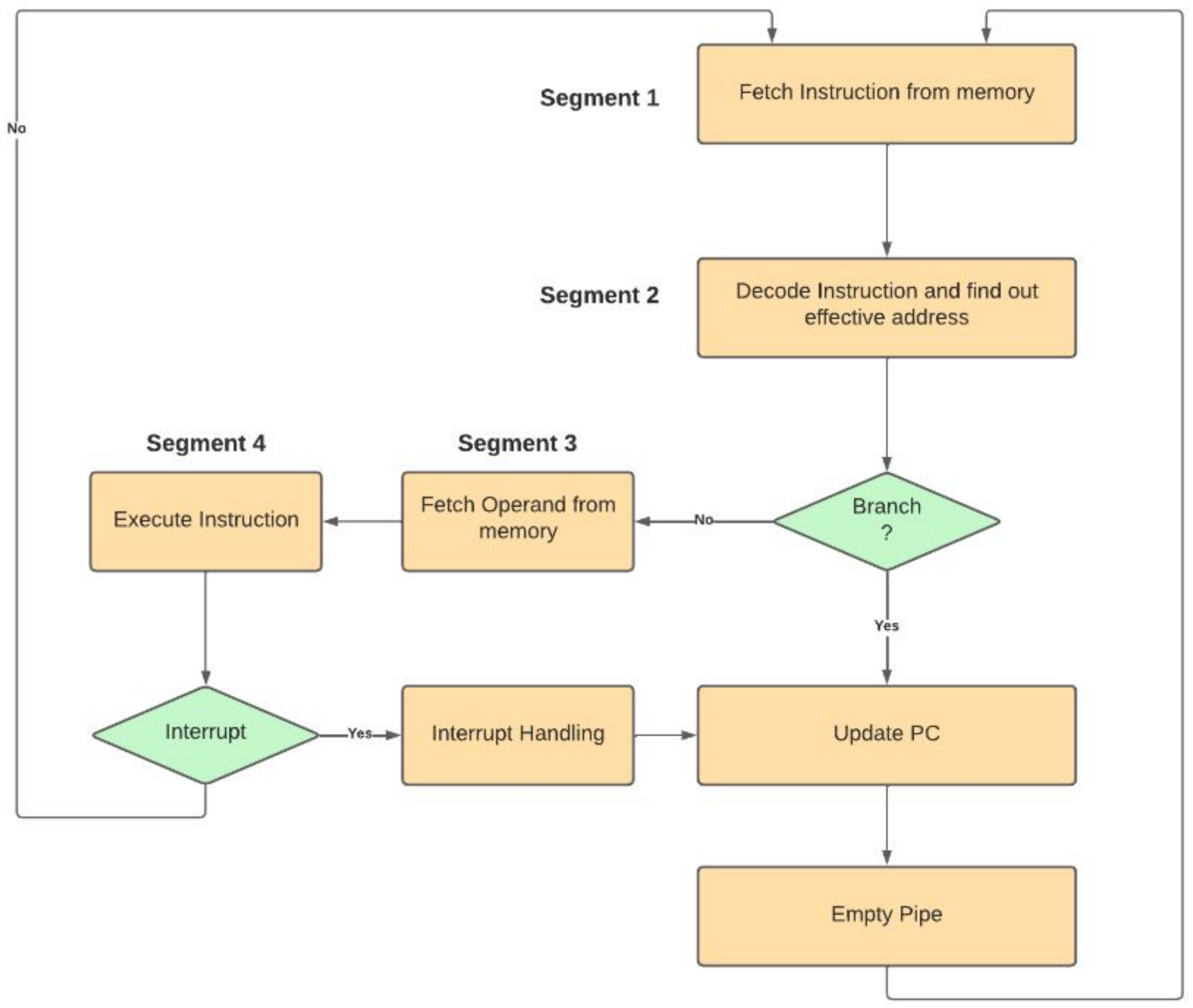
1. Instruction Pipelining
2. Arithmetic Pipelining

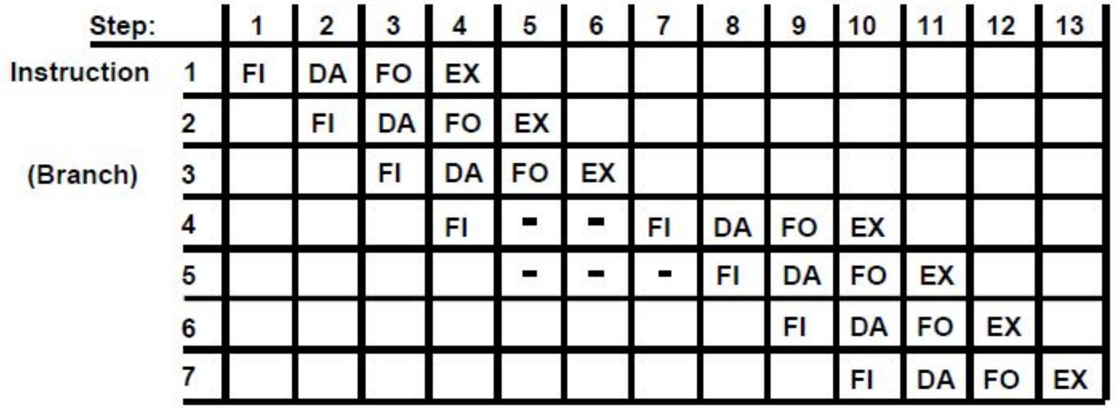
2. INSTRUCTION PIPELINING

INSTRUCTION PIPELINING

- Instruction pipelines are used to divide the task of executing a stream of instructions into subtasks to be executed in different pipeline segments to improve the throughput of the computer system.
- For example, if we have a stream of instructions, then one segment of the pipeline can read the instructions while another segment can decode the previous instruction. In this way, more than one instruction will be handled simultaneously by the computer system which will improve its throughput. The instruction pipeline will be more efficient if the instructions are divided into equal-duration segments.
- A typical example of an instruction pipeline used by computer systems consists of the following segments:
  - Segment 1: This segment will fetch the instruction from the memory
  - Segment 2: This segment will decode the instruction and find out the effective address
  - Segment 3: This segment will fetch the operands from the memory
  - Segment 4: This segment will execute the instruction

INSTRUCTION PIPELINING





	Step:	1	2	3	4	5	6	7	8	9	10	11	12	13
Instruction	1	FI	DA	FO	EX									
	2		FI	DA	FO	EX								
(Branch)	3			FI	DA	FO	EX							
	4				FI	-	-	FI	DA	FO	EX			
	5					-	-	-	FI	DA	FO	EX		
	6									FI	DA	FO	EX	
	7										FI	DA	FO	EX

3. ARITHMETIC PIPELINE

ARITHMETIC PIPELINING

- Arithmetic pipelines are used to divide an arithmetic task into subtasks to be executed in different pipeline segments.
- The main purpose is to speed up the arithmetic operations
- Pipeline arithmetic units **are usually found in very high-speed computers**. They are used to implement floating-point operations, multiplication of fixed-point numbers, and similar computations encountered in scientific problems.
  - Floating-point operations are easily decomposed into suboperations.
  - A pipeline multiplier is essentially an array multiplier, with special adders designed to minimize the carry propagation time through the partial products.

ARITHMETIC PIPELINING: FLOATING POINT ADDER

We know that two floating point numbers are represented in their normalized form using mantissa and exponents. Mantissa represents the precision of the number and the exponent represents the range.

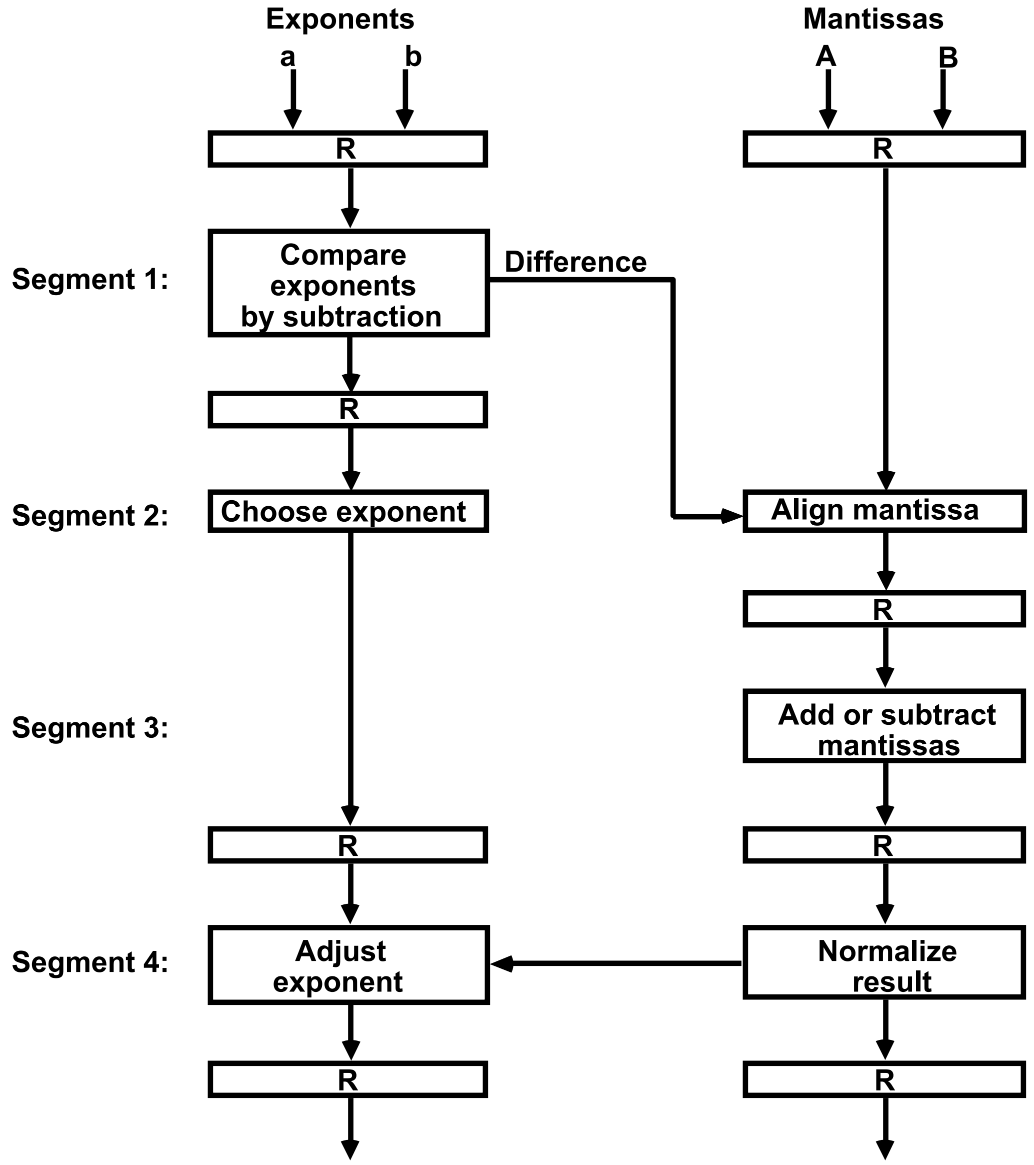
Let us consider two floating point numbers *X* and *Y*.

$$X = A \times 2^a$$
$$Y = B \times 2^b$$

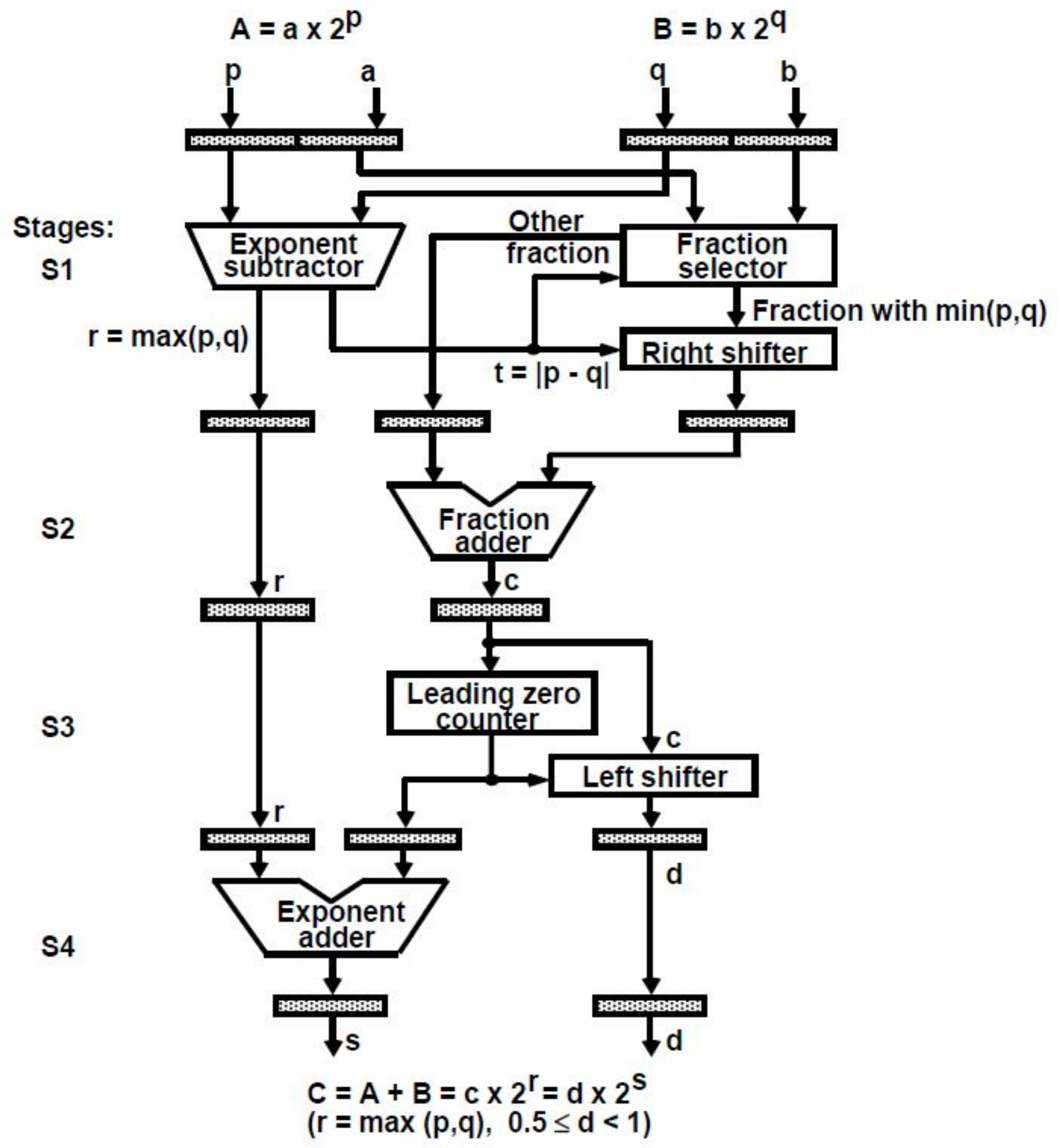
A and *B* are two fractions that represent the mantissa and *a* and *b* are the exponents.

1. Compare the exponents
2. Align the mantissa
3. Add/sub the mantissa
4. Normalize the result

In the arithmetic pipeline, these four steps are performed in four different segments to improve the **speed** and **throughput** of the system



ARITHMETIC PIPELINING: FLOATING POINT ADDER



EXAMPLE: FLOATING POINT ADDER

- The following numerical example may clarify the suboperations performed in each segment. For simplicity, we use decimal numbers, although Figure refers to binary numbers.

1. Consider the two normalized floating-point numbers:

$$X = 0.9504 * 10^3$$
$$Y = 0.8200 * 10^2$$

2. The two exponents are subtracted in the first segment to obtain 3 - 2 = 1. The larger exponent 3 is chosen as the exponent of the result.
3. The next segment shifts the mantissa of Y to the right to obtain

$$X = 0.9504 * 10^3$$
$$Y = 0.0820 * 10^3$$

EXAMPLE: FLOATING POINT ADDER

4. This aligns the two mantissa under the same exponent. The addition of the two mantissa in segment 3 produces the sum $Z = 1.0324 * 10^3$.
5. The sum is adjusted by normalizing the result so that it has a fraction with a nonzero first digit. This is done by shifting the mantissa once to the right and incrementing the exponent by one to obtain the normalized sum.

$$\mathbf{Z = 0.10324 * 10^4.}$$

6. The comparator, shifter, adder-subtractor, incrementer, and decrementer in the floating-point pipeline are implemented with combinational circuits. Suppose that the time delays of the four segments are $t_1 = 60$ ns, $t_2 = 70$ ns, $t_3 = 100$ ns, $t_4 = 80$ ns, and the interface registers have a delay of $t_r = 10$ ns.
7. The clock cycle is chosen to be $t_p = t_3 + t_r = 110$ ns.
8. An equivalent non-pipeline floating point adder-subtractor will have a delay time $t_n = t_1 + t_2 + t_3 + t_4 + t_r = 320$ ns.
9. In this case the pipelined adder has a speedup of 320/110 = 2.9 over the nonpipelined adder.

4. PIPELINING HAZARDS

PIPELINING HAZARDS

- Pipeline hazards are situations that prevent the next instruction in the instruction stream from executing during its designated clock cycles.
- Any condition that causes a stall in the pipeline operations can be called a hazard.
- There are primarily three types of hazards:

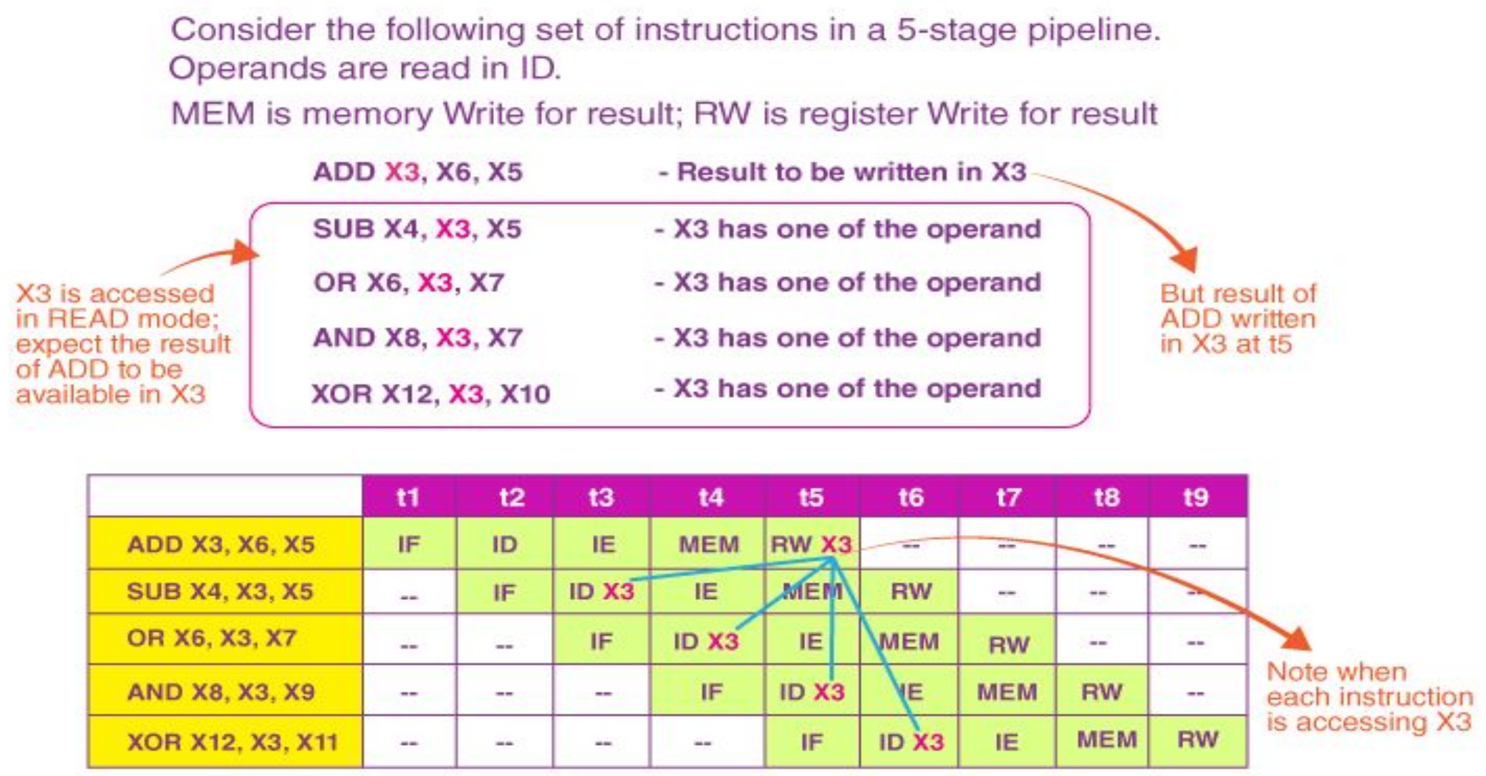
1. Data Hazards
2. Control Hazards or instruction Hazards
3. Structural Hazards.

1. DATA HAZARDS

- A data hazard is any condition in which either the source or the 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.
- When the execution of an instruction is dependent on the results of a prior instruction that's still being processed in a pipeline, data hazards occur. If the execution is done in a pipelined processor, it is highly likely that the interleaving of these two instructions can lead to incorrect results due to data dependency between the instructions. Thus the pipeline needs to be stalled as and when necessary to avoid errors.

1. DATA HAZARDS

- Consider the following scenario.



Consider the following set of instructions in a 5-stage pipeline.
Operands are read in ID.
MEM is memory Write for result; RW is register Write for result

ADD X3, X6, X5	- Result to be written in X3
SUB X4, X3, X5	- X3 has one of the operand
OR X6, X3, X7	- X3 has one of the operand
AND X8, X3, X7	- X3 has one of the operand
XOR X12, X3, X10	- X3 has one of the operand

X3 is accessed in READ mode; expect the result of ADD to be available in X3

But result of ADD written in X3 at t5

	t1	t2	t3	t4	t5	t6	t7	t8	t9
ADD X3, X6, X5	IF	ID	IE	MEM	RW X3	--	--	--	--
SUB X4, X3, X5	--	IF	ID X3	IE	MEM	RW	--	--	--
OR X6, X3, X7	--	--	IF	ID X3	IE	MEM	RW	--	--
AND X8, X3, X9	--	--	--	IF	ID X3	IE	MEM	RW	--
XOR X12, X3, X11	--	--	--	--	IF	ID X3	IE	MEM	RW

Note when each instruction is accessing X3

DATA HAZARDS CLASSIFICATION

- Data hazards are divided into three types according to the order in which READ or WRITE operations are performed on the register:

1. **Flow/True Data Dependency [RAW (or Read after Write)]:**

This is when one instruction makes use of data from a previous instruction.

Example,

```
ADD X0, X1, X2

SUB X4, X3, X0
```

DATA HAZARDS CLASSIFICATION

2. **Anti-Data Dependency [WAR (or Write after Read)]**

When the second instruction is written to a register before the first instruction is read, this is known as a race condition. In the case of a simple structure of a pipeline, this is uncommon. WAR, on the other hand, can occur in some machines having complex and specific instructions.

Example,

```
ADD X2, X1, X0
SUB X0, X3, X4
```

3. Output data dependency [WAW (or Write after Write)]

This is a situation where two simultaneous instructions must write the same register in the same sequence they were issued.

Example,

```
ADD X0, X1, X2

SUB X0, X4, X5
```

DATA HAZARDS CLASSIFICATION

- **Important Note:**

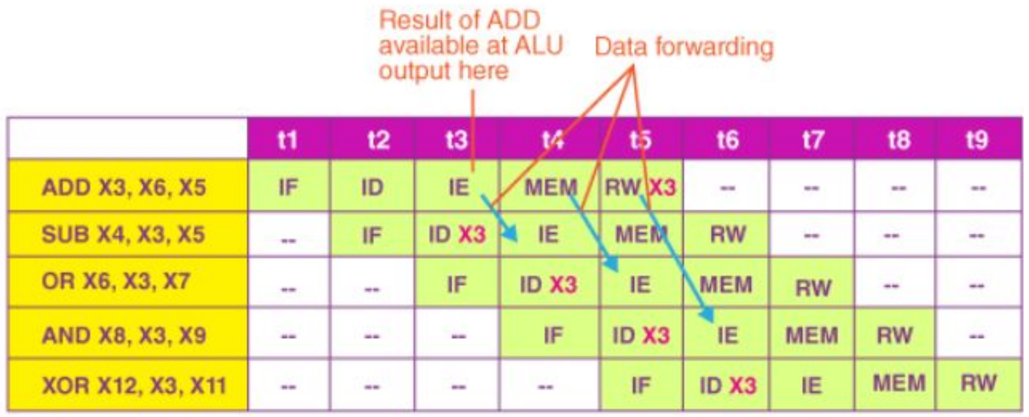
  WAW and WAR hazards can only occur when instructions are executed in parallel or out of order. These occur because the same register numbers have been allotted by the compiler although avoidable.

  This situation is fixed by renaming one of the registers by the compiler or by delaying the updating of a register until the appropriate value has been produced.

  Modern CPUs not only have incorporated Parallel execution with multiple ALUs but also out of order issues and execution of instructions along with many stages of pipelines.

1. DATA HAZARDS

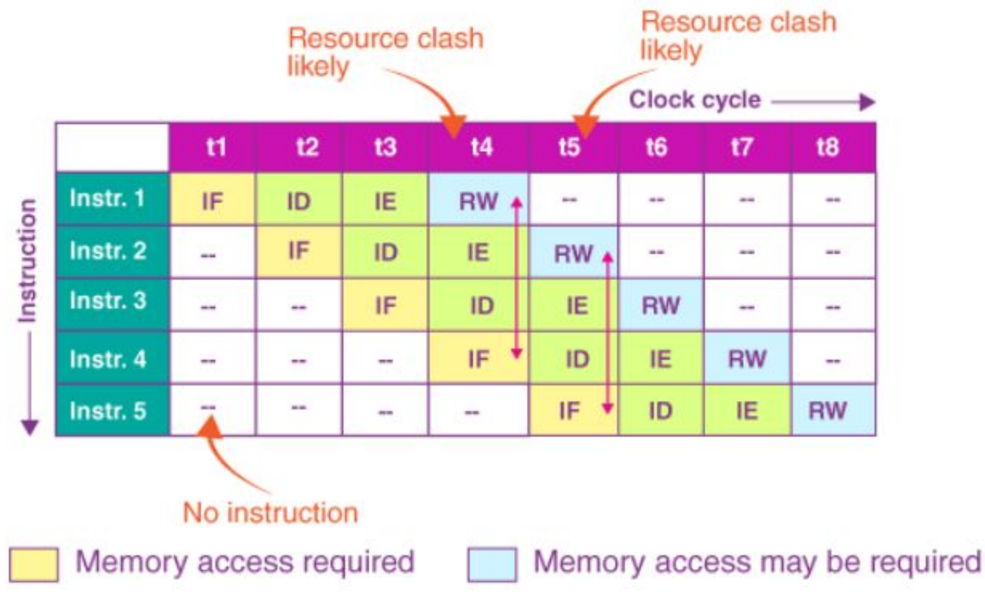
- **Solution 1:** At the IF stage of the SUB instruction, add three bubbles. This will make it easier for SUB – ID (Instruction Decoder) to work at t6. As a result, all subsequent instructions in the pipe are similarly delayed.
- **Solution 2:** Forwarding of Data – Data forwarding is the process of sending a result straight to that functional unit that needs it: a result is transferred from one unit's output to another's input. The goal is to have the solution ready for the next instruction as soon as possible.



	t1	t2	t3	t4	t5	t6	t7	t8	t9
ADD X3, X6, X5	IF	ID	IE	MEM	RW X3	--	--	--	--
SUB X4, X3, X5	--	IF	ID X3	IE	MEM	RW	--	--	--
OR X6, X3, X7	--	--	IF	ID X3	IE	MEM	RW	--	--
AND X8, X3, X9	--	--	--	IF	ID X3	IE	MEM	RW	--
XOR X12, X3, X11	--	--	--	--	IF	ID X3	IE	MEM	RW

2. STRUCTURAL HAZARDS

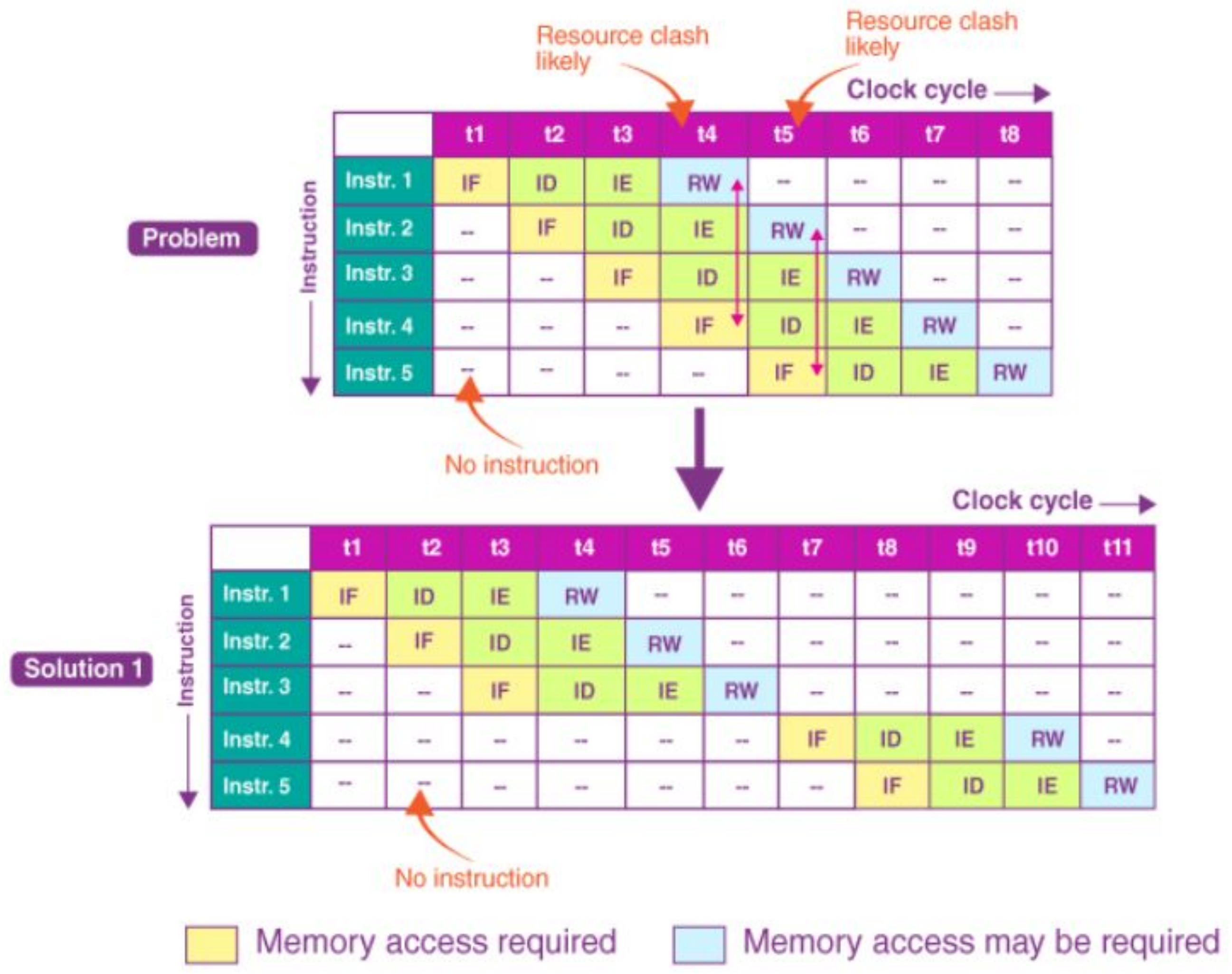
- Hardware resource conflicts among the instructions in the pipeline cause structural hazards. Memory, a GPR Register, or an ALU might all be used as resources here.
- When more than one instruction in the pipe requires access to the very same resource in the same clock cycle, a resource conflict is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



	t1	t2	t3	t4	t5	t6	t7	t8
Instr. 1	IF	ID	IE	RW	--	--	--	--
Instr. 2	--	IF	ID	IE	RW	--	--	--
Instr. 3	--	--	IF	ID	IE	RW	--	--
Instr. 4	--	--	--	IF	ID	IE	RW	--
Instr. 5	--	--	--	--	IF	ID	IE	RW

2. STRUCTURAL HAZARDS

- **Solution: F**or a portion of the pipeline, instructions must be performed in series rather than parallel.



	t1	t2	t3	t4	t5	t6	t7	t8
Instr. 1	IF	ID	IE	RW	--	--	--	--
Instr. 2	--	IF	ID	IE	RW	--	--	--
Instr. 3	--	--	IF	ID	IE	RW	--	--
Instr. 4	--	--	--	IF	ID	IE	RW	--
Instr. 5	--	--	--	--	IF	ID	IE	RW

	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10	t11
Instr. 1	IF	ID	IE	RW	--	--	--	--	--	--	--
Instr. 2	--	IF	ID	IE	RW	--	--	--	--	--	--
Instr. 3	--	--	IF	ID	IE	RW	--	--	--	--	--
Instr. 4	--	--	--	--	--	--	IF	ID	IE	RW	--
Instr. 5	--	--	--	--	--	--	--	IF	ID	IE	RW

🟨 Memory access required 🟦 Memory access may be required

3. CONTROL HAZARDS

- Control hazards are called Branch hazards and are caused by Branch Instructions. Branch instructions control the flow of program/ instructions execution. Recall that we use conditional statements in the higher-level language either for iterative loops or with conditions checking (correlate with for, while, if, and case statements). These are transformed into one of the variants of BRANCH instructions. It is necessary to know the value of the condition being checked to get the program flow.
- Thus **a Conditional hazard** occurs when the decision to execute an instruction is based on the result of another instruction like a conditional branch, which checks the condition's resultant value.
- The branch and jump instructions decide the program flow by loading the appropriate location in the Program Counter(PC). The PC has the value of the next instruction to be fetched and executed by CPU. Consider the following sequence of instructions.

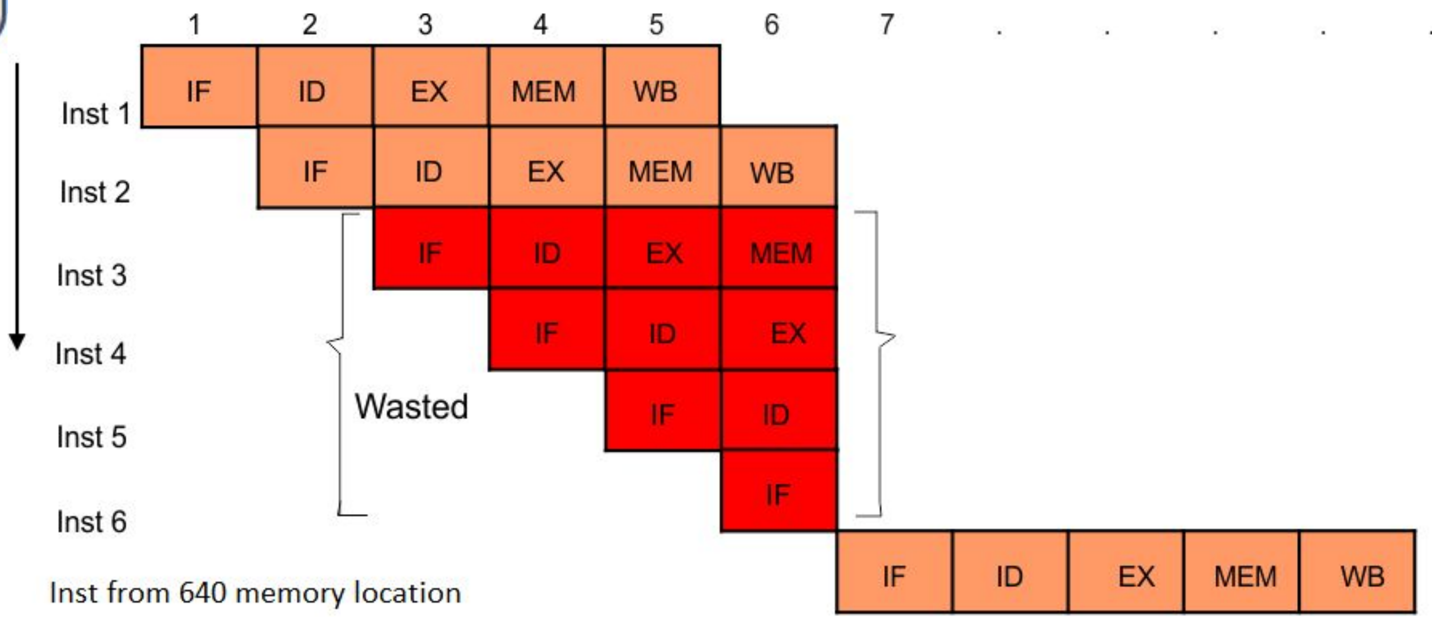
3. CONTROL HAZARDS

PC = 404

400: I_1	ADD	R3, R4, R6
404: I_2	JMP	640
408: I_3	AND	R6, R7, R5

This AND instruction is not going to be executed at all

Because of the unconditional JMP instruction, the next PC will be 640., which gets loaded onto the PC at the end of execution of JMP instruction



SOLUTION FOR CONTROL HAZARDS

1. **Stall:**

   Stall the given pipeline as soon as any branch instructions are decoded. Just don't allow IF anymore. Stalling reduces throughput as it always does. According to statistics, at least 30% of the instructions in a program are BRANCH. With Stalling, the pipeline is effectively operating at 50% capacity.

2. **Prediction:**

   Consider a for or a while loop that is repeated 100 times. We know the program would run 100 times without the given branch condition being met. The program only exits the loop for the 101st time. As a result, it's better to let the pipeline run its course and then flush/undo when the branch condition is met. This has less of an impact on the pipeline's throttle and stalling.

SOLUTION FOR CONTROL HAZARDS

3. **Dynamic Branch Prediction :**

A history record is maintained with the help of Branch Table Buffer (BTB). The BTB is a kind of cache, which has a set of entries, with the PC address of the Branch Instruction and the corresponding effective branch address. This is maintained for every branch instruction that occurs.

Branch Instruction Address	Target Branch Address taken

4. **Reordering Instructions:**

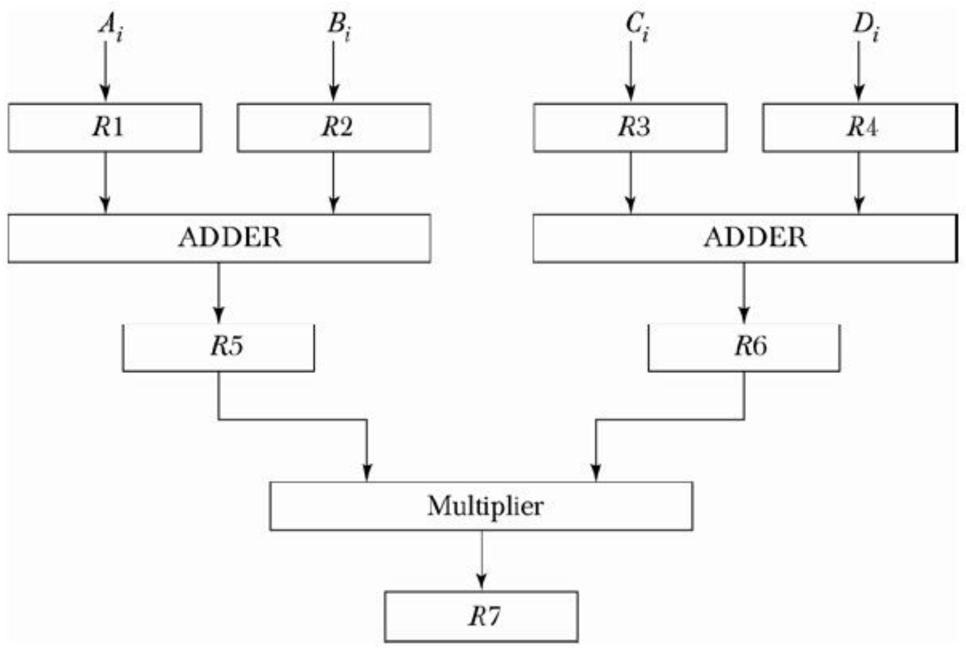
Delayed branching entails reordering the instructions to move the branch instruction later in the sequence, allowing safe and beneficial instructions that are unaffected by the result of a branch to be brought in earlier in the sequence, delaying the fetch of the branch instruction. If such instructions are not available, NOP is used. The Compiler is used to implement this delayed branch.

5. NUMERICAL ON PIPELINING

QUESTION 1

In certain scientific computations it is necessary to perform the arithmetic operation $(A_i + B_i)(C_i + D_i)$ with a stream of numbers. Specify a pipeline configuration to carry out this task. Use the contents of all registers in the pipeline for i = 1 through 6.

Solution:



QUESTION 2

Determine the number of clock cycles that it takes to process 200 tasks in a six-segment pipeline

Solution: Pipelined execution is = n + m – 1

n = 6 segments

m = 200 tasks

(n + m – 1) = 6 + 200 – 1 = **205 cycles**

QUESTION 3

Draw a space-time diagram for a six-segment pipeline showing the time it takes to process eight tasks.

Solution:

Segment	1	2	3	4	5	6	7	8	9	10	11	12	13
1	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8					
2		T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8				
3			T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8			
4				T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8		
5					T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	
6						T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8

$(n + m - 1) = 6 + 8 - 1 =$ **13 cycles**

QUESTION 4

A non-pipeline system takes 50 ns to process a task. The same task can be processed in a six-segment pipeline with a clock cycle of 10 ns. Determine the speedup ratio of the pipeline for 100 tasks. What is the maximum speedup that can be achieved?

Solution: t_n = 50ns

t_p = 10ns

n = 100

K = 6

- The speedup can be calculated as:

$$S = \frac{n * t_n}{(k + n - 1) * t_p} = \frac{100 * 50}{(6 + 100 - 1) * 10} 4.76$$

- The maximum speedup is $Smax = \frac{t_n}{t_p} = \frac{50}{10} = 5$

QUESTION 5

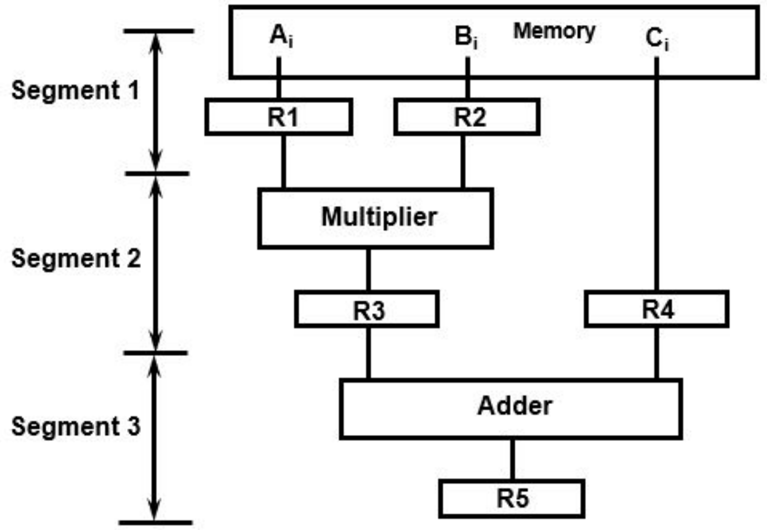
The pipeline of Fig has the following propagation times: 40 ns for the operands to be read from memory into registers R1 and R2, 45 ns for the signal to propagate through the multiplier, 5 ns for the register transfer time into R3, and 15 ns to add the two numbers into R5.

a) What is the minimum clock cycle time that can be used?

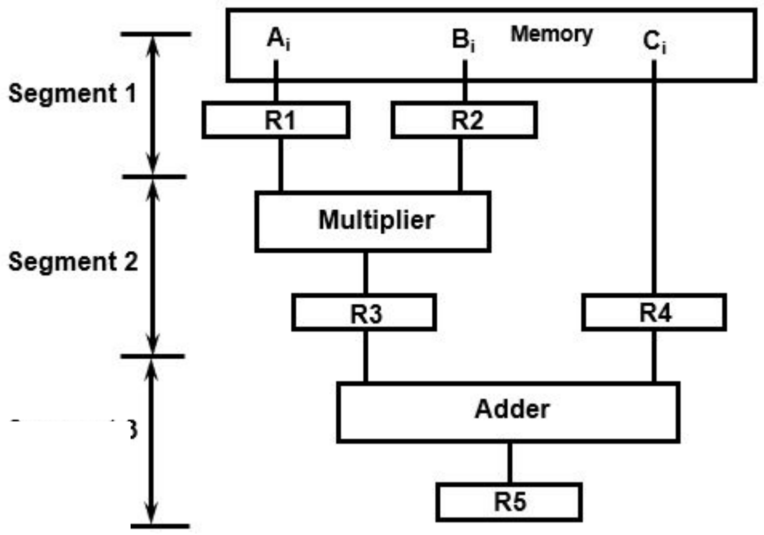
b) A non-pipeline system can perform the same operation by removing R3 and R4. How long will it take to multiply and add the operands without using the pipeline?

c) Calculate the speedup of the pipeline for 10 tasks and again for 100 tasks.

d) What Is the maximum speedup that can be achieved?



QUESTION 5



Solution:

(a) $t_p = 45 + 5 = 50$ ns k = 3
(b) $t_n = 40 + 45 + 15 = 100$ ns

(c) $$S = \frac{nt_n}{(k + n\text{-}1)\, t_p} = \frac{10 \times 100}{(3+9)50} = 1.67$$ for n = 10

$$= \frac{100 \times 100}{(3+99)50} = 1.96$$ for n = 100

(d) $$S_{max} = \frac{t_n}{t_p} = \frac{100}{50} = 2$$