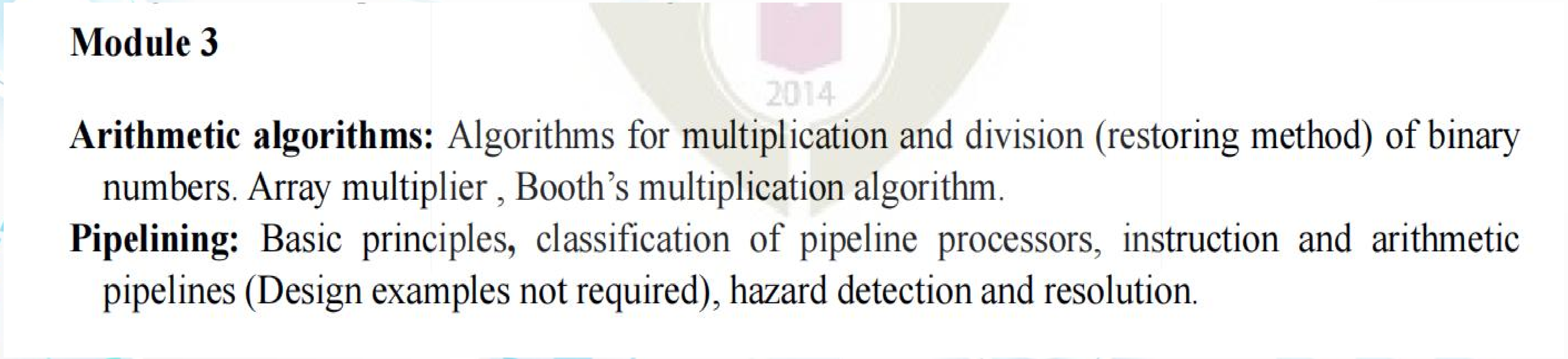
Computer Organization and Architecture

Prepared By,
Jaseela Beevi S
Assistant Professor
Dept. of CSE

Prepared By,
Jaseela Beevi S
Assistant Professor
Dept. of CSE, TKMCE

Module III - Contents



Module 3

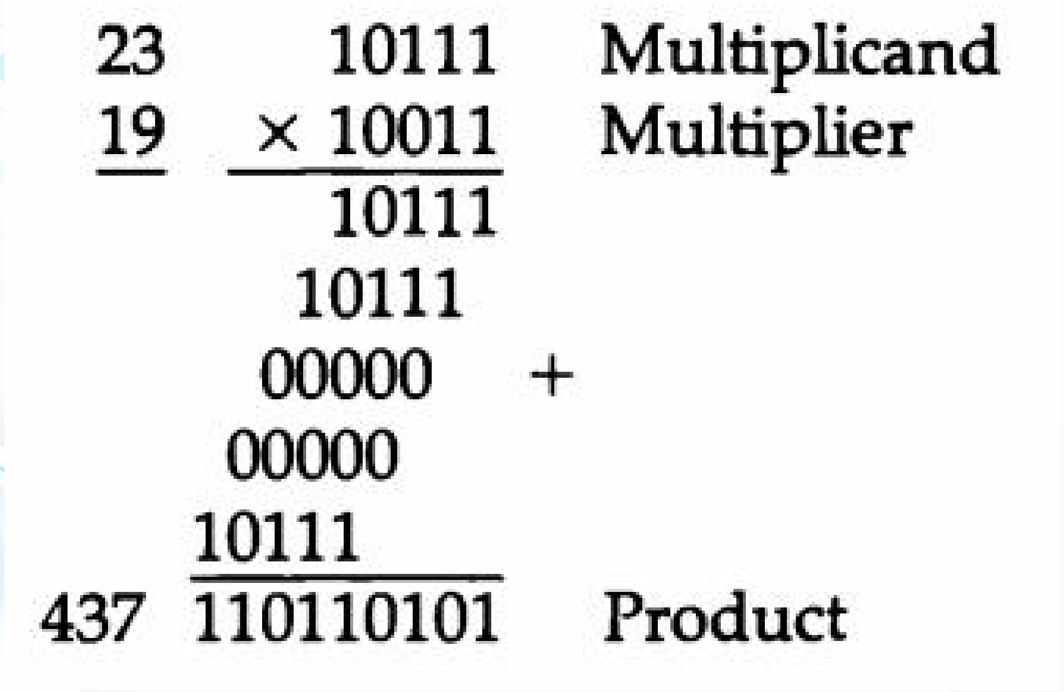
Arithmetic algorithms: Algorithms for multiplication and division (restoring method) of binary numbers. Array multiplier , Booth's multiplication algorithm.

Pipelining: Basic principles, classification of pipeline processors, instruction and arithmetic pipelines (Design examples not required), hazard detection and resolution.

Algorithms for Multiplication

Multiplication of two fixed - point binary numbers in signed magnitude representation requires sussessive shift and add operation.

Multiplicant ---> $(10111)_2$ = $(23)_{10}$
Multiplier ---> $(10011)_2$ = $(9)_{10}$



```
 23        10111    Multiplicand
 19      × 10011    Multiplier
 --      -------
           10111
          10111
         00000    +
        00000
       10111
       ---------
437    110110101    Product
```

Hardware Implementation of Multiply operation

of the processor

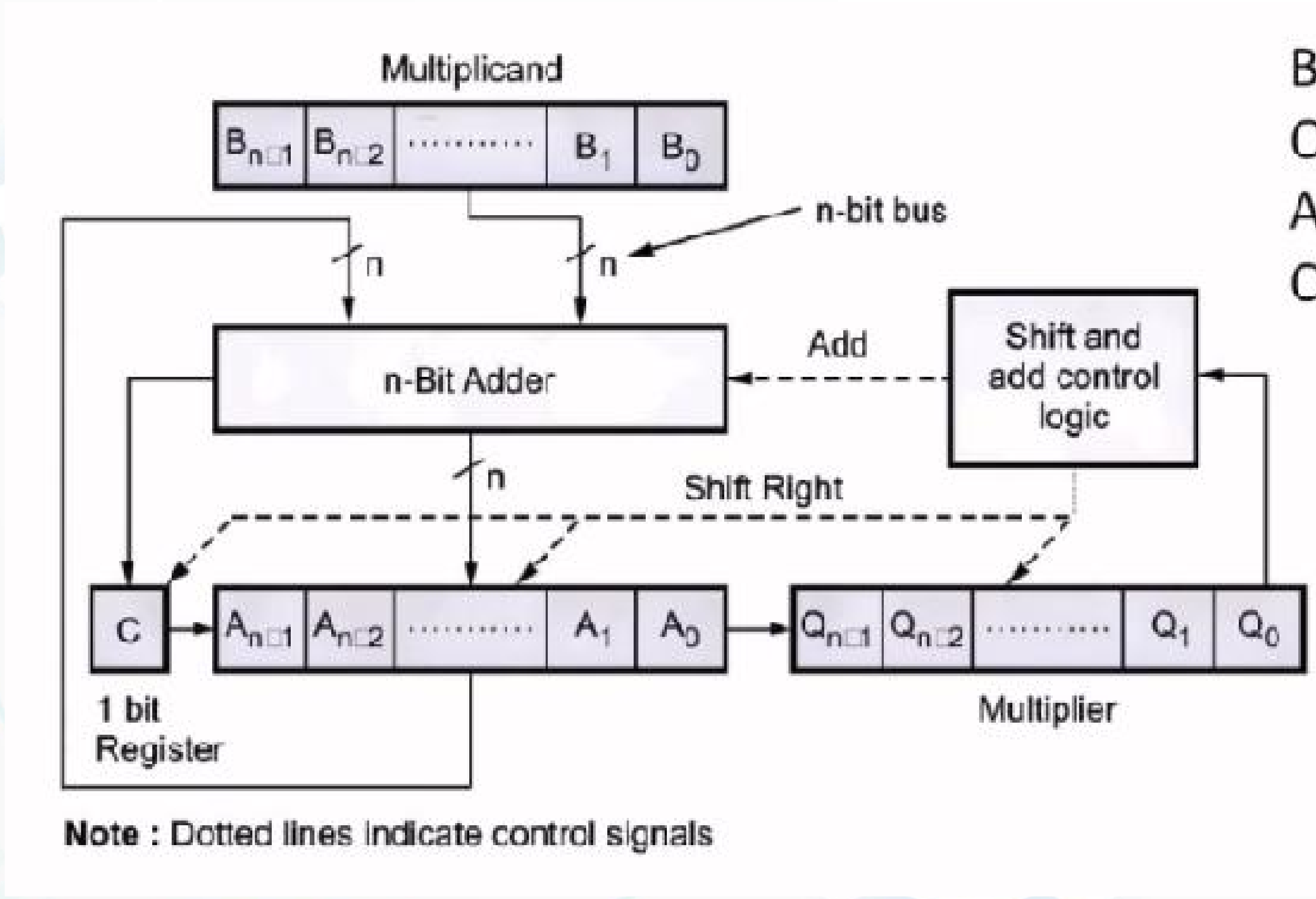
When multiplication is implemented in a digital computer, do the following

1) Provide an adder for the summation of only two binary numbers and successively accumulate the partial product in a register.

2) Instead of shifting the multiplicand to the left, the partial product is shifted to right.

3) When the corresponding bit of the multiplier is ‘0’, there is no need to add all 0’s to the partial product. since it will not alter the value.

Hardware Implementation of Multiply operation


Multiplicand
B_{n-1} B_{n-2} ………… B_1 B_0
n-bit bus
n
n
n-Bit Adder
Add
Shift and add control logic
n
Shift Right
C
A_{n-1} A_{n-2} ………… A_1 A_0
Q_{n-1} Q_{n-2} ………… Q_1 Q_0
1 bit Register
Multiplier
Note : Dotted lines indicate control signals

B- Multiplicand

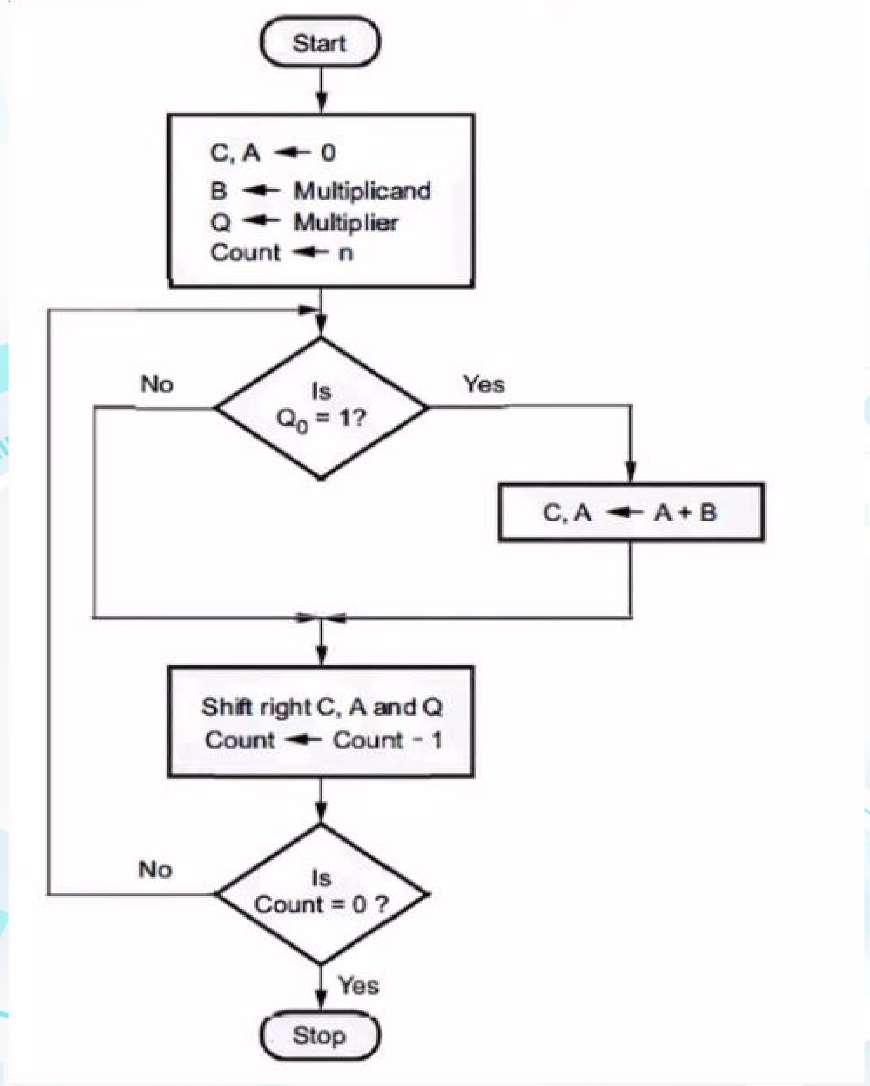
Q- Multiplier

A- Partial Products

C- Carry

This is also known as Partial Sum Approach

Flowchart of Multiply operation



Multiplication - Partial Sum Approach

processor

|M| = n-bits, |Q| = n-bits, |M x Q| = (n+n) = 2n bits

Multiplicant (M) : 1 1 1 1   Multiplier (Q) : 1 1 1 1

Result : (1 1 1 0 0 0 0 1)

Multiplication - Partial Sum Approach

processor

For every bit in Q(from right to left),

(1) If the bit is 0, perform right shift once on the register’s content.

(2) If the bit is 1, first add M with the most significant n-bits of the register’s content then perform right shift once.

Multiplicant(M) : 1010

Multiplier(Q): 0110

Multiplication - Partial Sum Approach

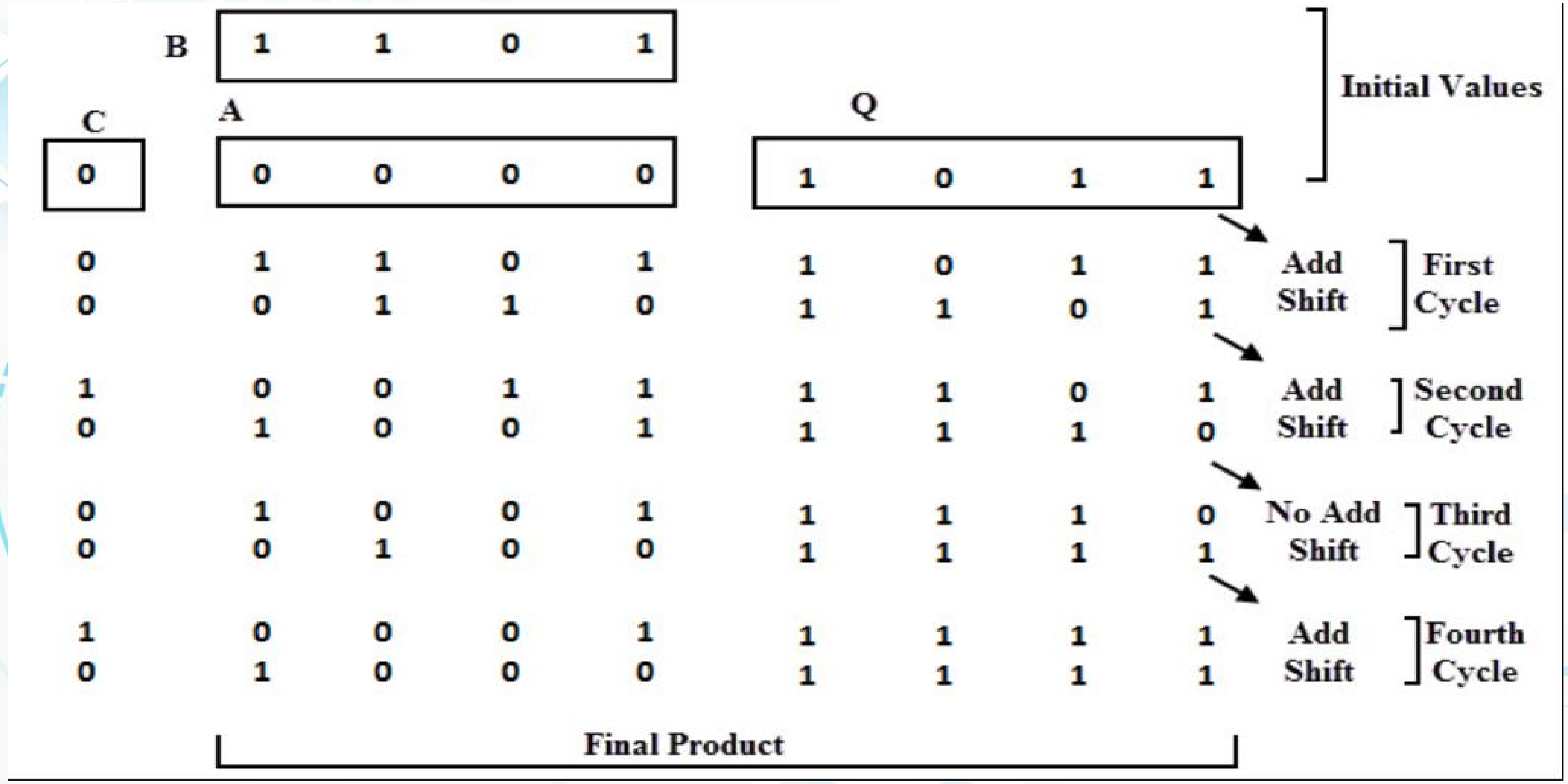
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
1	0	1	0				
1	0	1	0	0	0	0	0
0	1	0	1	0	0	0	0
1	0	1	0				
1	1	1	1	0	0	0	0
0	1	1	1	1	0	0	0
0	0	1	1	1	1	0	0

Multiplicand (M) : 1010
Multiplier (Q) : 0110

No. of right shifts = n
No. of additions = no. of 1's in Q

Multiplication - Partial Sum Approach

the processor



		B									
		1	1	0	1						
C	A				Q						
0	0	0	0	0	1	0	1	1		Initial Values
0	1	1	0	1	1	0	1	1	Add	First Cycle
0	0	1	1	0	1	1	0	1	Shift	
1	0	0	1	1	1	1	0	1	Add	Second Cycle
0	1	0	0	1	1	1	1	0	Shift	
0	1	0	0	1	1	1	1	0	No Add	Third Cycle
0	0	1	0	0	1	1	1	1	Shift	
1	0	0	0	1	1	1	1	1	Add	Fourth Cycle
0	1	0	0	0	1	1	1	1	Shift	
	Final Product									

Binary Division

Divident = 1 0 1 0 1 0 = $(42)_{10}$

Divisor = 1 0 1 1 = $(11)_{10}$

0 1 1 $\rightarrow$ Quotient $(3)_{10}$

Divisor $\leftarrow$ 1 0 1 1 | 1 0 1 0 1 0 $\rightarrow$ Divident

1 0 1 1

1 0 1 0 0

1 0 1 1

1 0 0 1 $\rightarrow$ Reminder $(9)_{10}$

Binary Division

Division algorithms can be grouped into 2 classes.

Slow Algorithms

- Restoring Division
- Non restoring Division

Fast Algorithms

- Division by Series Expansion
- Newton - Raphson Division

Restoring Division

- Initialize registers with values,

  Q = Dividend,

  A = 0,

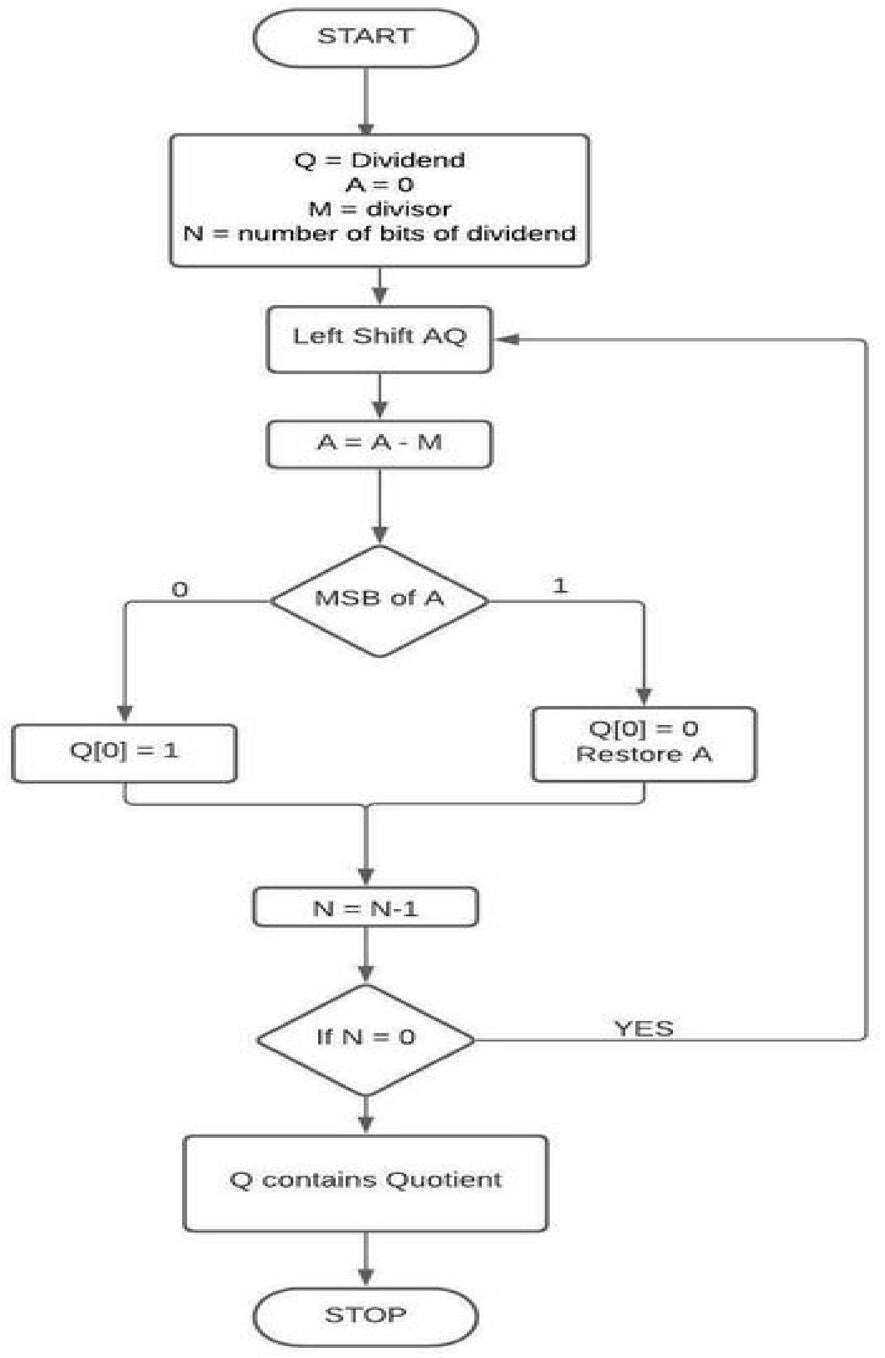
  M = divisor,

  N = number of bits of dividend.
- Left shift AQ means taking register A and Q as a single unit.
- Subtract A with M and store in A.
- Check the most significant bit of A:

  If it is 0, set the least significant bit to 1.

  Else, set the least significant bit to 0.
- Restore the value of A and decrement the value of counter N.
- If N = 0, break the loop; otherwise, go to step 2.
- The quotient is stored in register Q.

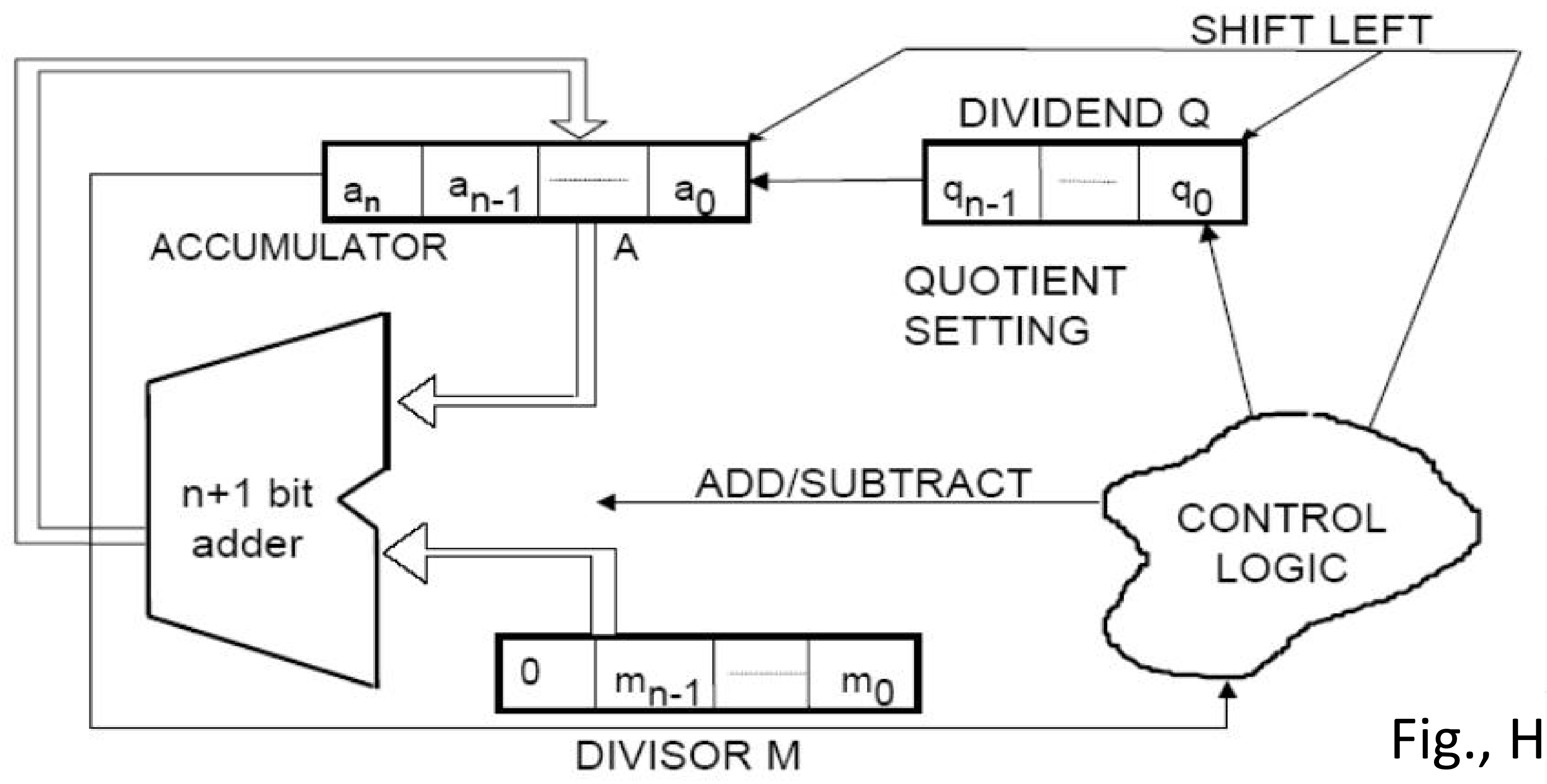
Restoring Division Flowchart



Restoring Division

n	M	A	Q	Action/Operation
4	0 0 1 1	0 0 0 0 0	1 0 1 1	Initialization
		0 0 0 0 1	0 1 1 ?	Shift left AQ
		1 1 1 1 0	0 1 1 ?	A = A - M
3		0 0 0 0 1	0 1 1 0	Q[0] <-- 0, Restore A
		0 0 0 1 0	1 1 0 ?	Shift left AQ
		1 1 1 1 1	1 1 0 ?	A = A - M
2		0 0 0 1 0	1 1 0 0	Q[0] <-- 0, Restore A
		0 0 1 0 1	1 0 0 ?	Shift left AQ
		0 0 0 1 0	1 0 0 ?	A = A - M
1		0 0 0 1 0	1 0 0 1	Q[0] <-- 1,
		0 0 1 0 1	0 0 1 ?	Shift left AQ
		0 0 0 1 0	0 0 1 ?	A = A - M
0		0 0 0 1 0	0 0 1 1	Q[0] <-- 1, Restore A

Restoring Division



Hardware Design of Restoring Division Algorithm.

Fig., Hardware Circuit to divide two n-bit numbers using restoring algorithm

Multiplication - Booth’s Algorithm

anization of the processor

Booth’s algorithm gives a procedure for multiplying binary integer’s in 2’s complement representation.

It operates on the the fact that strings of 0’s in the multiplier requires no addition but just shifting, and a string of 1’s in the multiplier from bit weight 2^k to weight 2^m can be treated as $2^{k+1} - 2^m$

For example, the binary number 001110 (+14) has a string of 1’s from 2^3 to 2^1.

The number can be represented as $2^{k+1} - 2^m = 2^4 - 2^1 = 16 - 2 = 14$

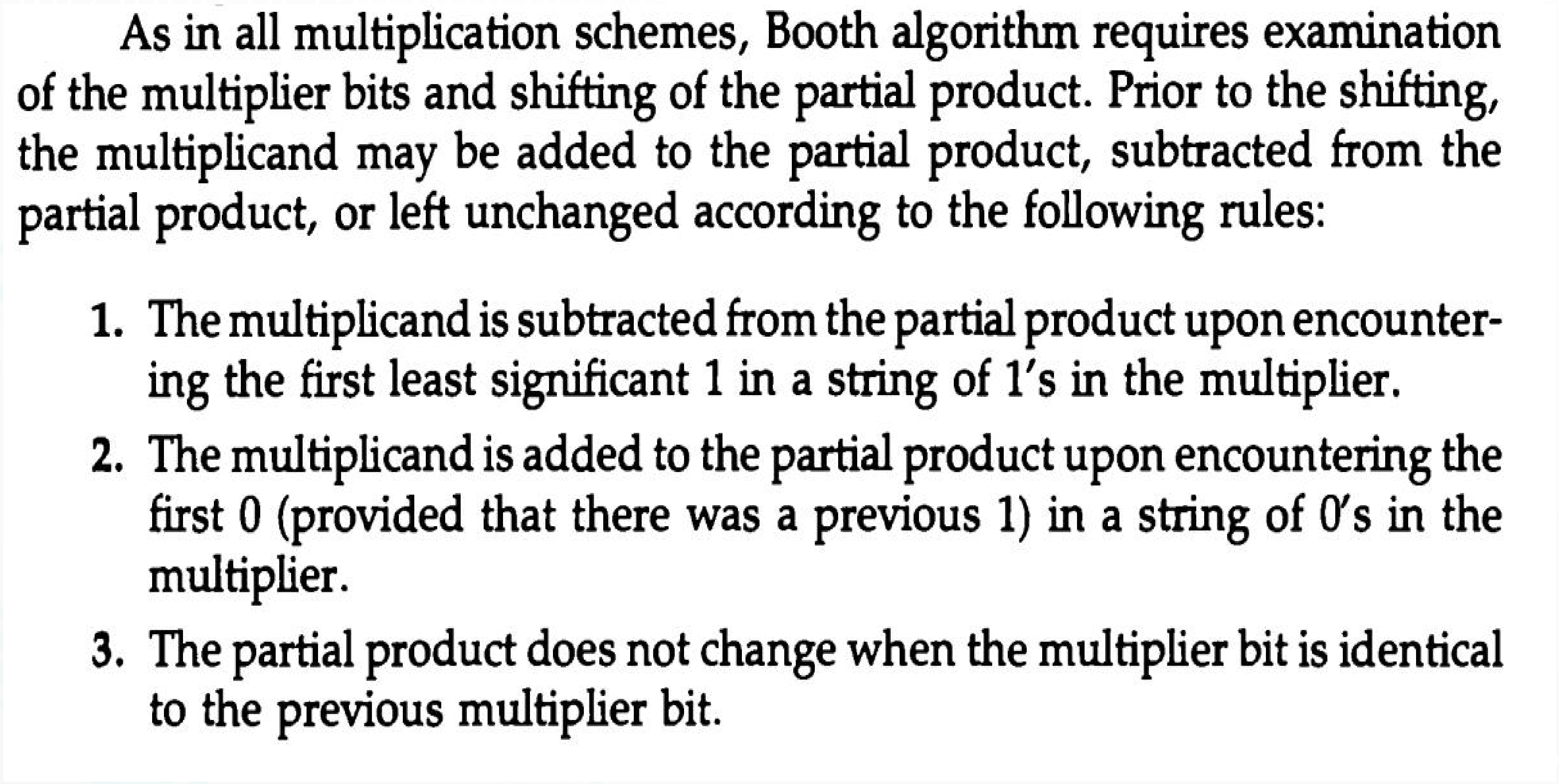
Multiplication - Booth’s Algorithm

anization of the processor

Therefore the multiplication of *M x 14,* where *M* is the multiplicand and 14 is the multiplier, can be done as $M \times 2^4 - M \times 2^1$.

Thus the product can be obtained by shifting the binary multiplicand *M* 4 times to the left and subtracting *M* shifted left once.

Multiplication - Booth’s Algorithm



As in all multiplication schemes, Booth algorithm requires examination of the multiplier bits and shifting of the partial product. Prior to the shifting, the multiplicand may be added to the partial product, subtracted from the partial product, or left unchanged according to the following rules:

1. The multiplicand is subtracted from the partial product upon encountering the first least significant 1 in a string of 1’s in the multiplier.
2. The multiplicand is added to the partial product upon encountering the first 0 (provided that there was a previous 1) in a string of 0’s in the multiplier.
3. The partial product does not change when the multiplier bit is identical to the previous multiplier bit.

Multiplication - Booth’s Algorithm

anization of the processor

classroom lecture...

plz ref. your lecture notes.

Pipeline Hazards

anization of the processor

Hazards - Any condition that causes the pipeline to stall is called a hazard.

Pipeline hazards can be grouped into three categories.

1) Data Hazard
2) Structural Hazard
3) Control Hazard

Pipeline Hazards

anization of the processor

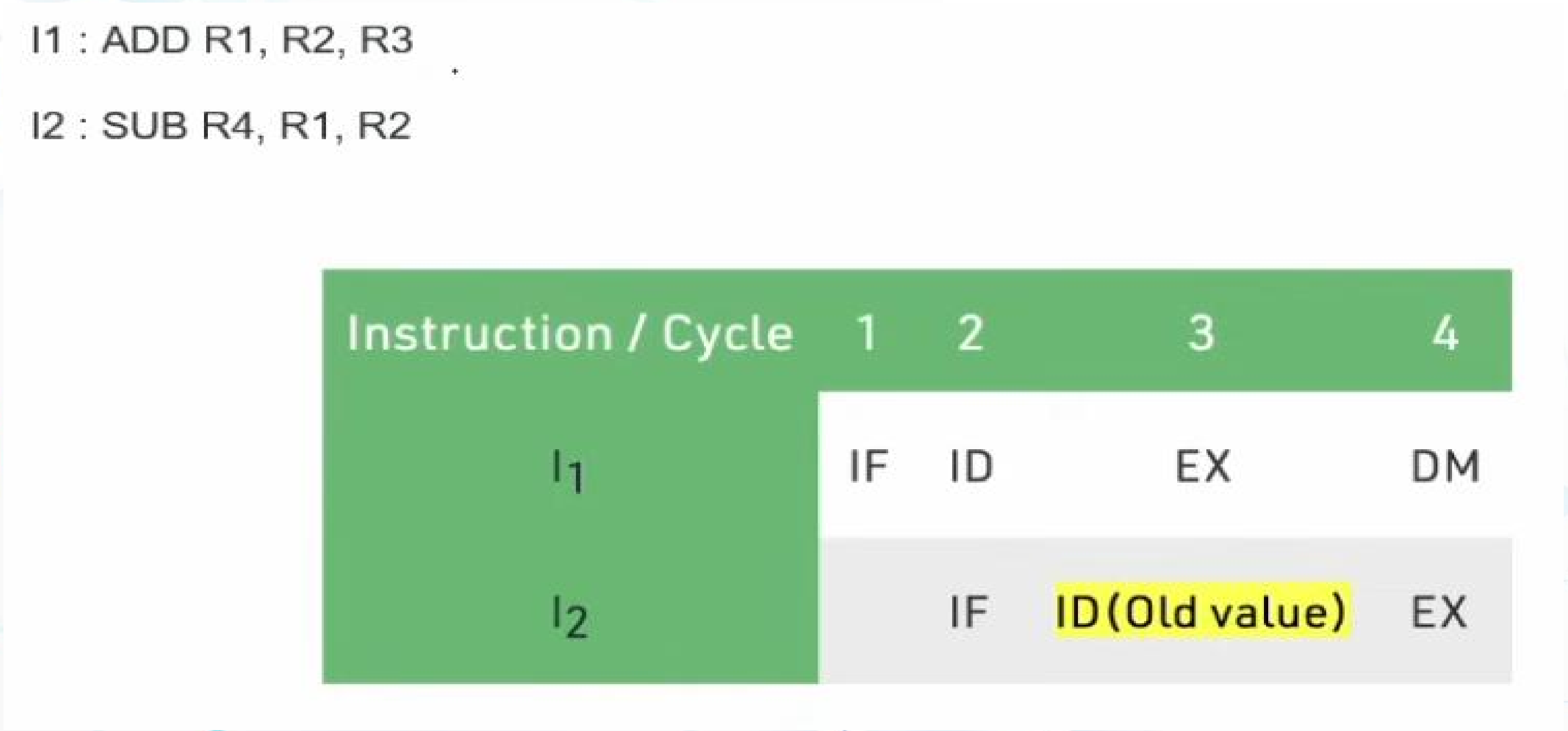
Data Hazards

- Any condition in which either the source or destination operands of an instruction are not available at the time expected in the pipeline.

- 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.

Pipeline Hazards

anization of the processor



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 (Old value)	EX

Pipeline Hazards

Solution for Data Hazard: Operand Forwarding

In operand forwarding, we use the **interface registers** present between the stages to hold the intermediate output so that dependent instruction can access new value from the interface register directly.

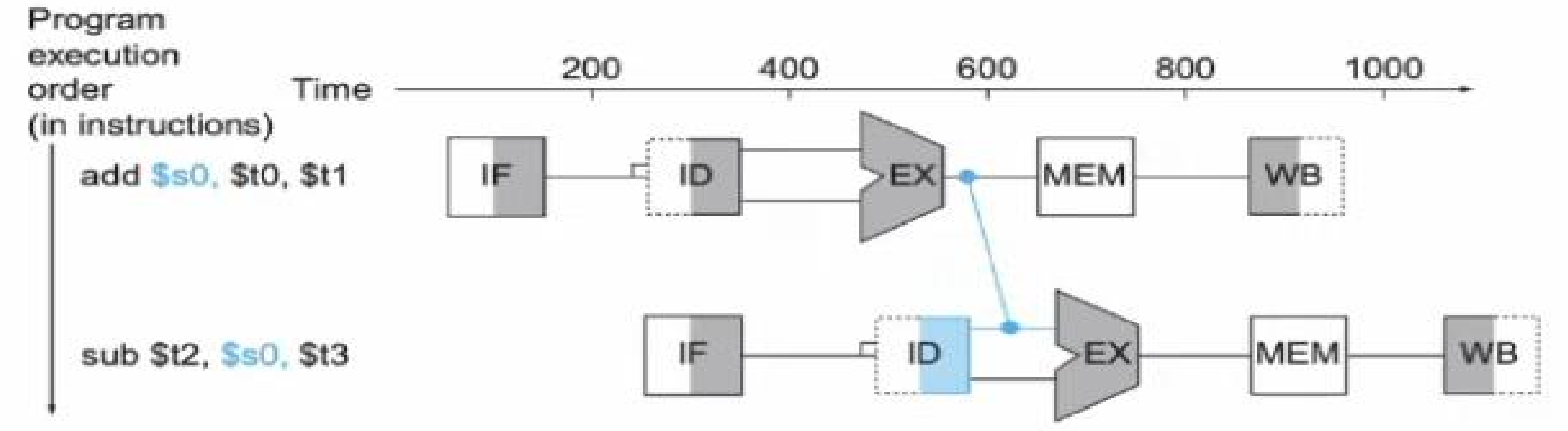


FIGURE 4.29 Graphical representation of forwarding. The connection shows the forwarding path from the output of the EX stage of add to the input of the EX stage for sub, replacing the value from register $s0 read in the second stage of sub.

Pipeline Hazards

anization of the processor

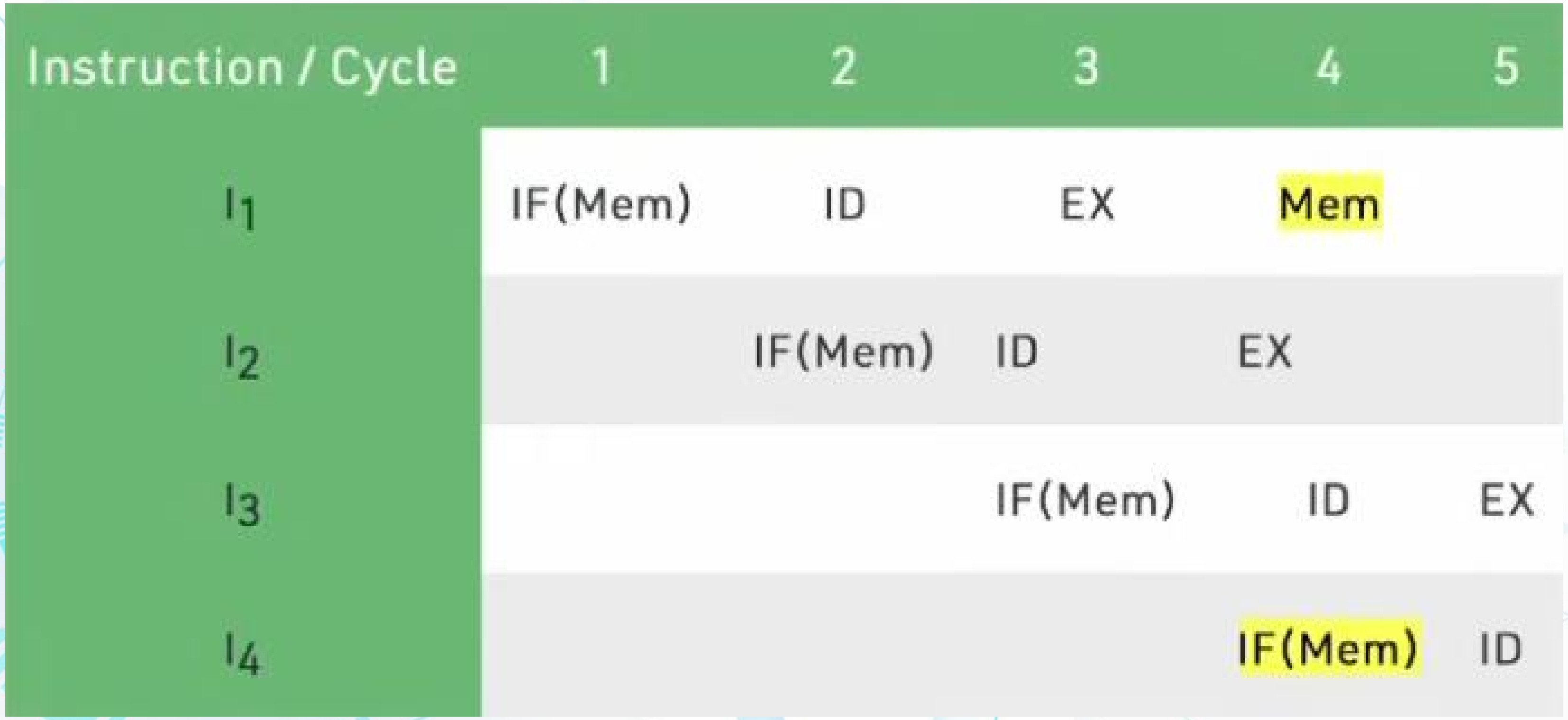
Structural Hazards

✓It means that the hardware cannot support the combination of instructions that we wnat to execute in the following clock cycle.

✓This dependency arises due to the resource **conflict** in the pipeline.

✓A **resource conflict** is a situation,when more than one instruction tries to access the same resource in the same cycle. a resource can be a register, memory or ALU.

Pipeline Hazards

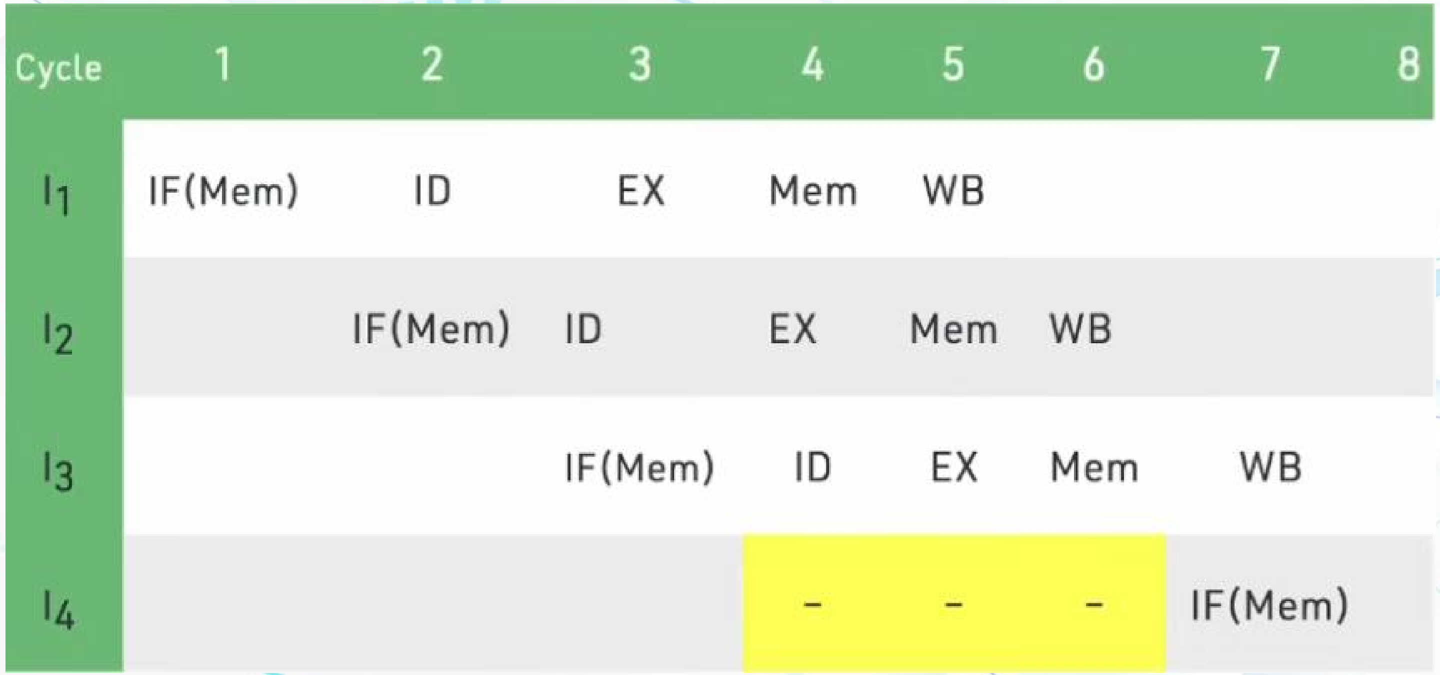


Instruction / Cycle	1	2	3	4	5
I_1	IF(Mem)	ID	EX	Mem	
I_2		IF(Mem)	ID	EX	
I_3			IF(Mem)	ID	EX
I_4				IF(Mem)	ID

Pipeline Hazards

anization of the processor

Solution for Structural Hazards



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)	

Stall:
To avoid this problem, we have to keep the instruction on wait until the required resource (memory in this case) becomes available.

Pipeline Hazards

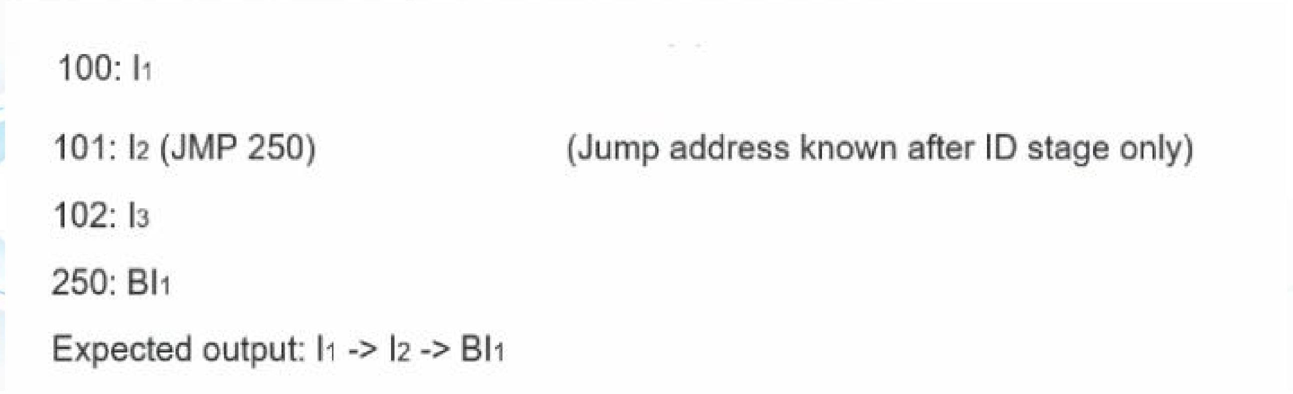
Control Hazards:

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.

Pipeline Hazards



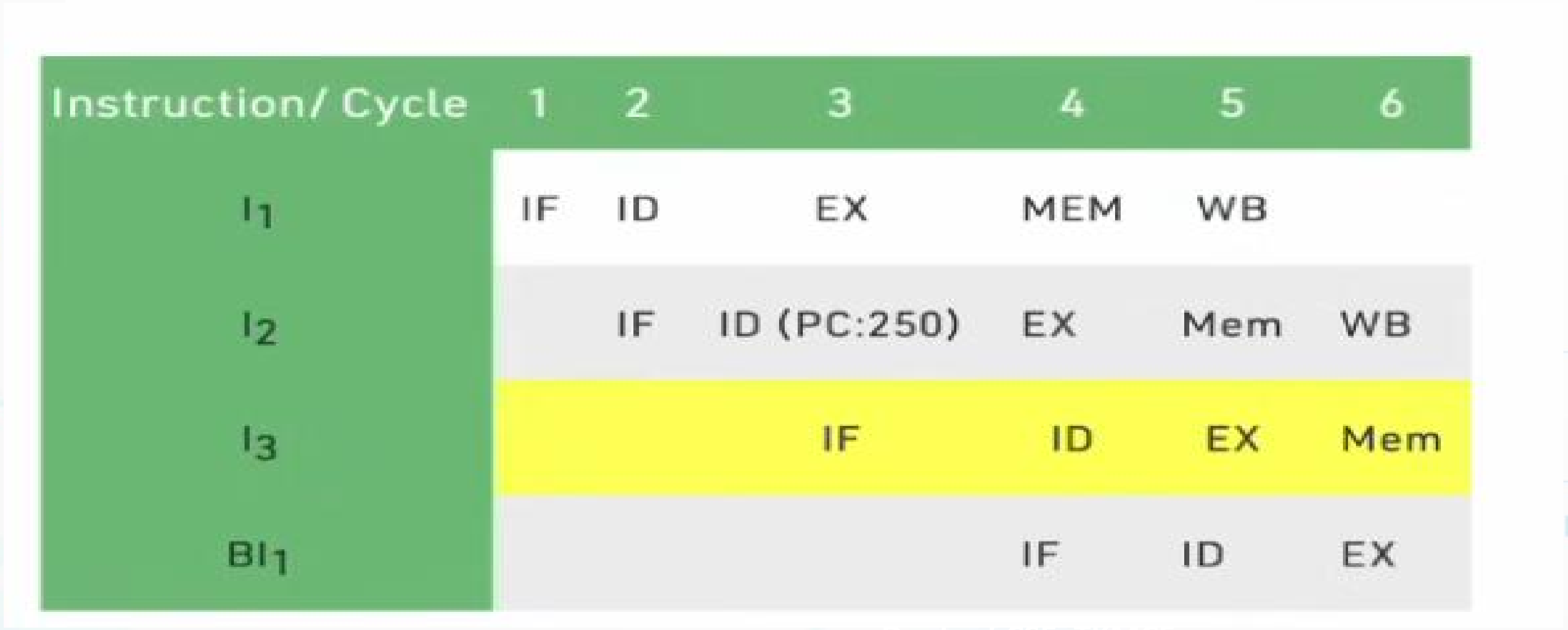
100: I_1

101: I_2 (JMP 250) (Jump address known after ID stage only)

102: I_3

250: BI_1

Expected output: I_1 -> I_2 -> BI_1

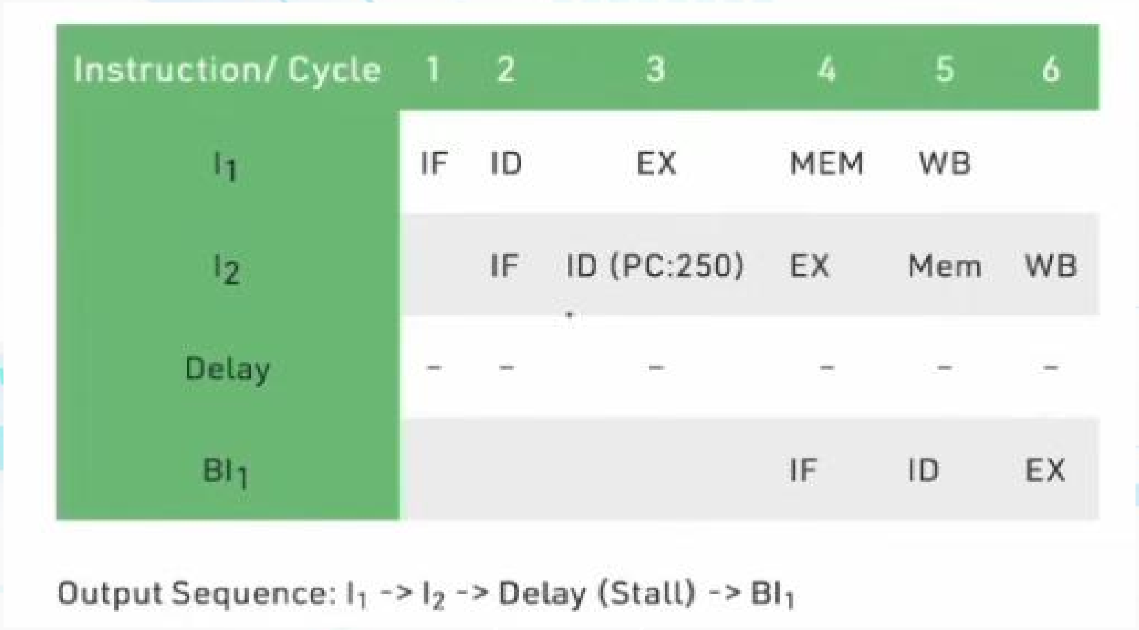


Instruction/ Cycle	1	2	3	4	5	6
I_1	IF	ID	EX	MEM	WB	
I_2		IF	ID (PC:250)	EX	Mem	WB
I_3			IF	ID	EX	Mem
BI_1				IF	ID	EX

Pipeline Hazards

Solution for Control Hazard:

1) Stall : To correct the above problem, we need to stop the instruction fetch until we get the target address of branch instruction.



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: I_1 -> I_2 -> Delay (Stall) -> BI_1

Pipeline Hazards

2) Branch Prediction

a) predict always that branches will be untaken. When you are right, the pipeline proceeds at full speed. Only when branches are taken does the pipeline stall.

b) Some branches predicted as taken and some as untaken. Eg. loop

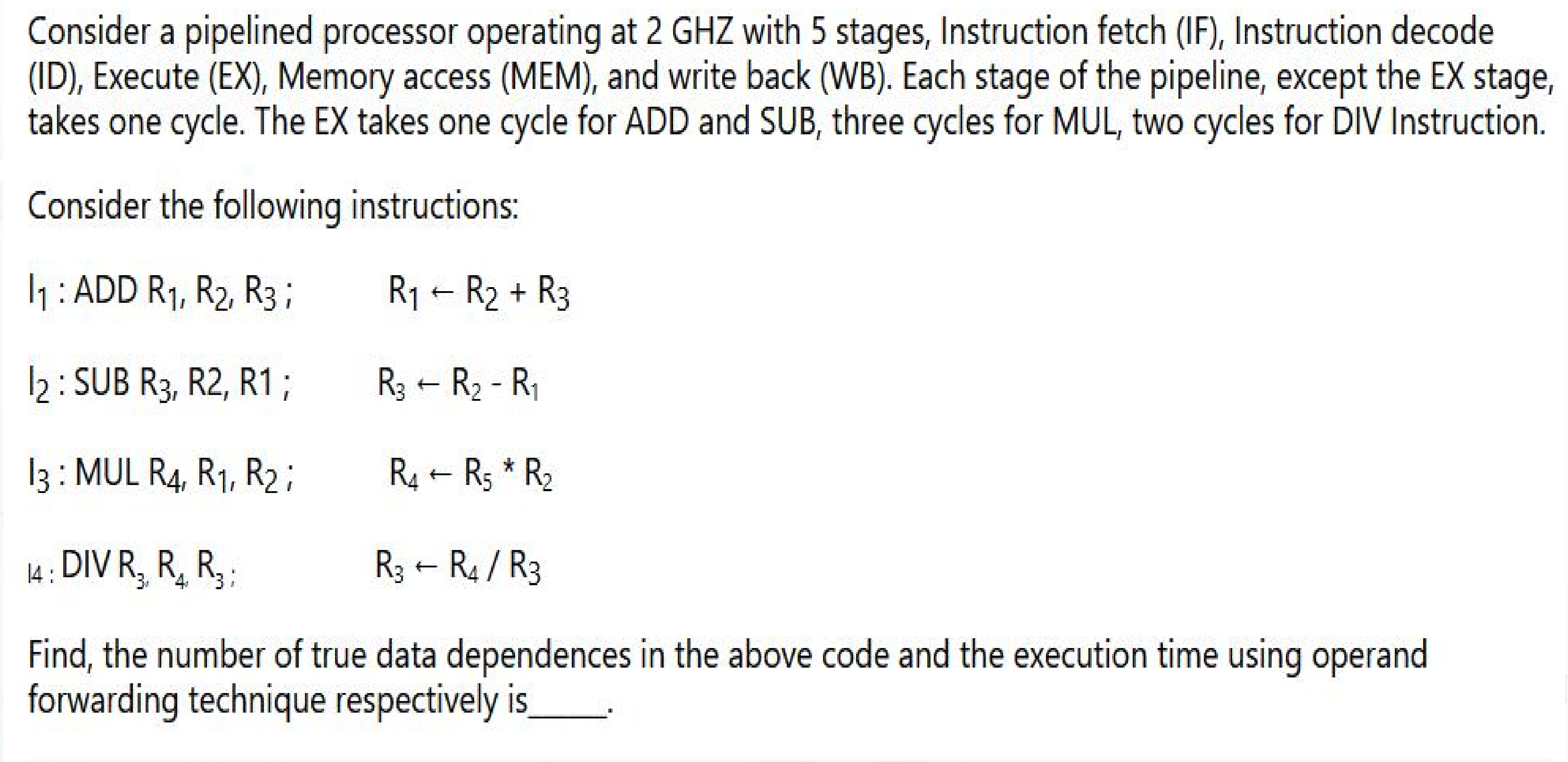
c) Dynamic hardware predictors, make their guesses depending on the behavior of each branch and may chage predictions for a branch over the life of a program.

Pipeline Hazards

With the help of Branch Table Buffer, a historical record is kept (BTB). The BTB is a type of cache that contains a series of entries containing the branch instruction’s PC address and the effective branch address. This is done for each branch instruction that is encountered.

When a conditional branch instruction is encountered, the BTB is queried for the matching branch instruction address. If the target branch address is hit, the next instruction is fetched from the associated target branch address. **Dynamic branch prediction** is the term for this.

Pipeline Hazards



Consider a pipelined processor operating at 2 GHZ with 5 stages, Instruction fetch (IF), Instruction decode (ID), Execute (EX), Memory access (MEM), and write back (WB). Each stage of the pipeline, except the EX stage, takes one cycle. The EX takes one cycle for ADD and SUB, three cycles for MUL, two cycles for DIV Instruction.

Consider the following instructions:

I_1 : ADD R_1, R_2, R_3 ; $R_1 \leftarrow R_2 + R_3$

I_2 : SUB R_3, R2, R1 ; $R_3 \leftarrow R_2 - R_1$

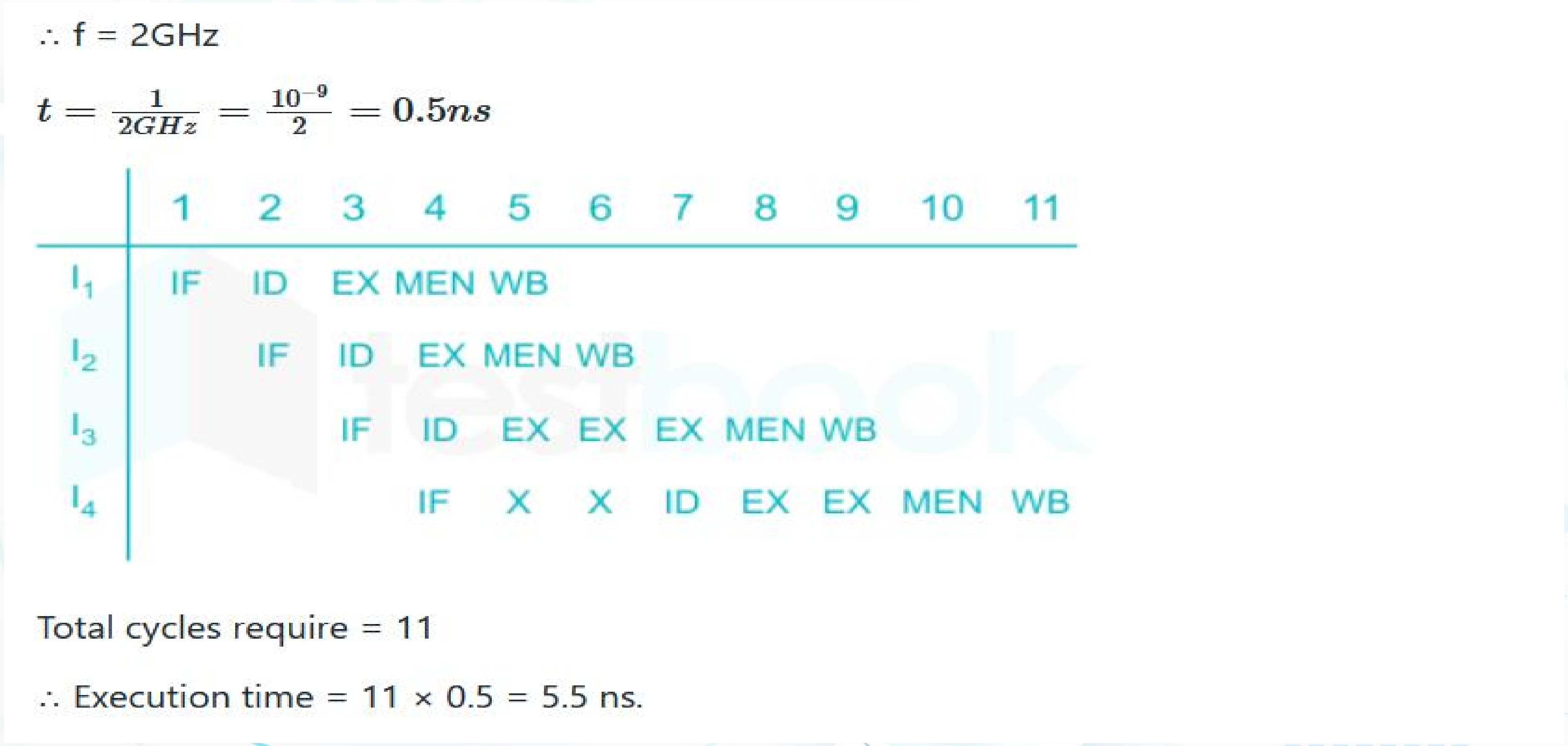
I_3 : MUL R_4, R_1, R_2 ; $R_4 \leftarrow R_5 * R_2$

I4 : DIV R_3, R_4, R_3 ; $R_3 \leftarrow R_4 / R_3$

Find, the number of true data dependences in the above code and the execution time using operand forwarding technique respectively is_____.

Pipeline Hazards

anization of the processor



$\therefore$ f = 2GHz

$$t = \frac{1}{2GHz} = \frac{10^{-9}}{2} = 0.5ns$$

	1	2	3	4	5	6	7	8	9	10	11
I_1	IF	ID	EX	MEN	WB						
I_2		IF	ID	EX	MEN	WB					
I_3			IF	ID	EX	EX	EX	MEN	WB		
I_4				IF	X	X	ID	EX	EX	MEN	WB

Total cycles require = 11

$\therefore$ Execution time = 11 × 0.5 = 5.5 ns.

Classification of Pipeline Processors

anization of the processor

Pipeline processors can be classified as

1) Instruction Pipeline

2) Arithmetic pipeline

Instruction Pipeline Architecture

anization of the processor

Instruction Pipeline:

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.

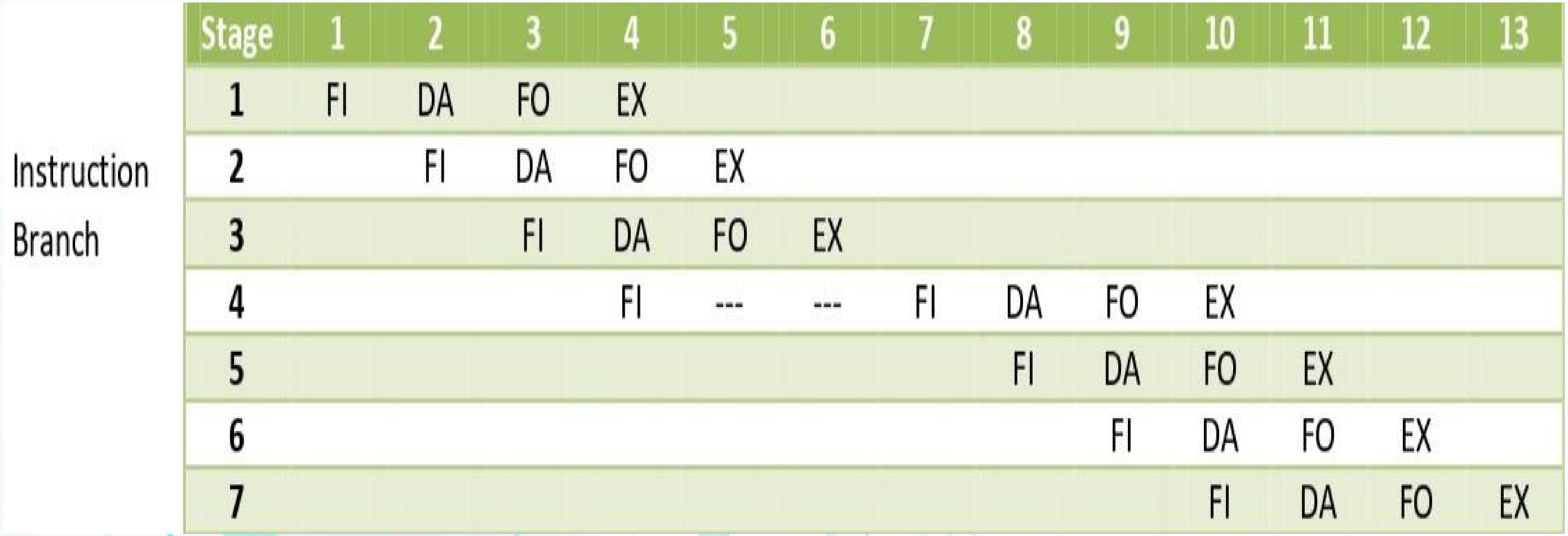
Instruction Pipeline Architecture

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.

Instruction Pipeline Architecture

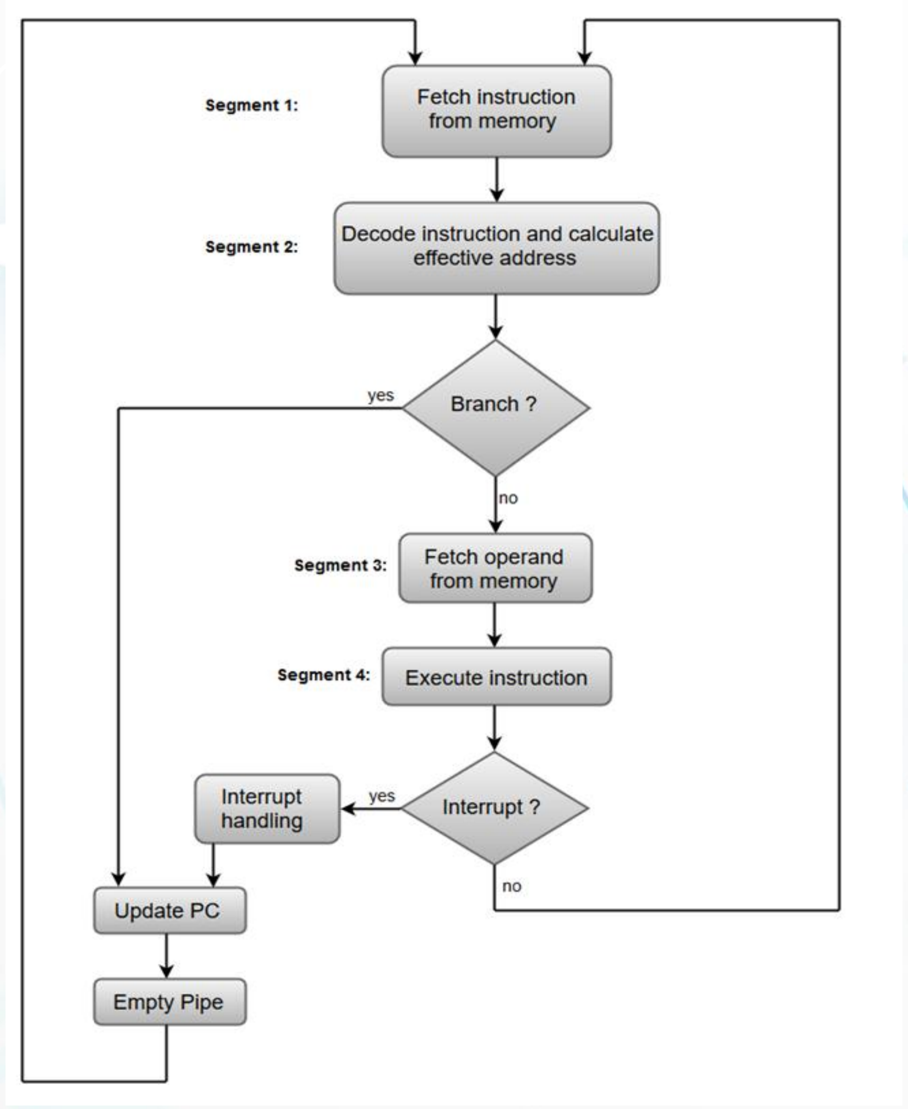
anization of the processor



Instruction Branch

Stage	1	2	3	4	5	6	7	8	9	10	11	12	13
1	FI	DA	FO	EX									
2		FI	DA	FO	EX								
3			FI	DA	FO	EX							
4				FI	---	---	FI	DA	FO	EX			
5								FI	DA	FO	EX		
6									FI	DA	FO	EX	
7										FI	DA	FO	EX

Instruction Pipeline Architecture



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.

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.

$X = A \times 10^{a}$ $\quad$ $Y = B \times 10^{b}$

$= 0.9505 \times 10^{3}$ $\quad$ $= 0.0810 \times 10^{3}$

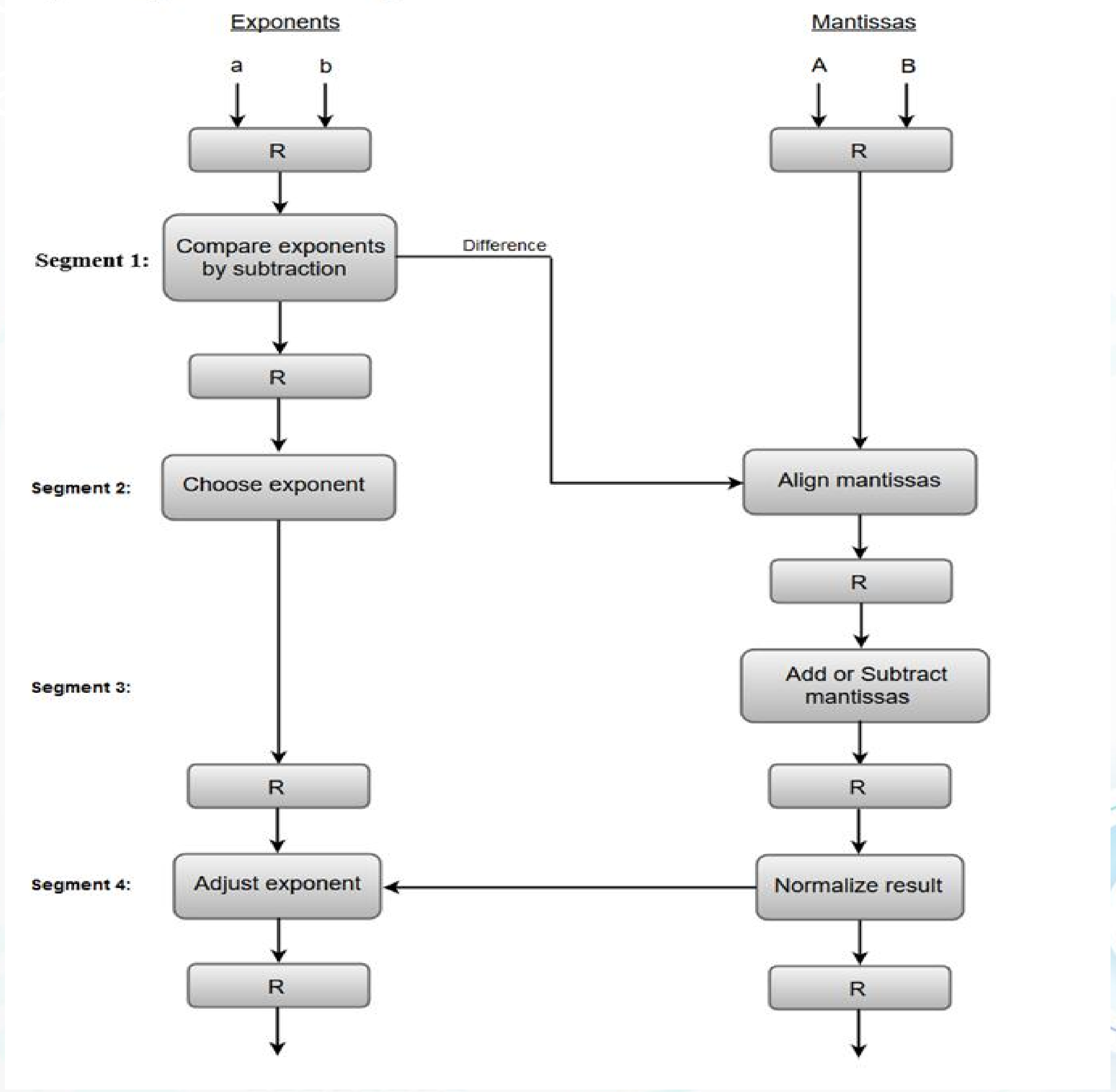
Instruction Pipeline Architecture

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.

Arithmetic Pipeline Architecture

Pipeline organization for floating point addition and subtraction:



Z = X + Y

0.9505×10^3 +

0.0810×10^3

1.0315×10^3 ---normalization---------------> 0.10315×10^4