Pipelining

Pipelining is a technique of decomposing a sequential process into sub operations with each sub process being executed in a special dedicated segment that operates concurrently with all other segments.

A pipeline can be visualized as a collection of processing segments through which binary information flows.

The pipeline organization will be demonstrated by means of a simple example.

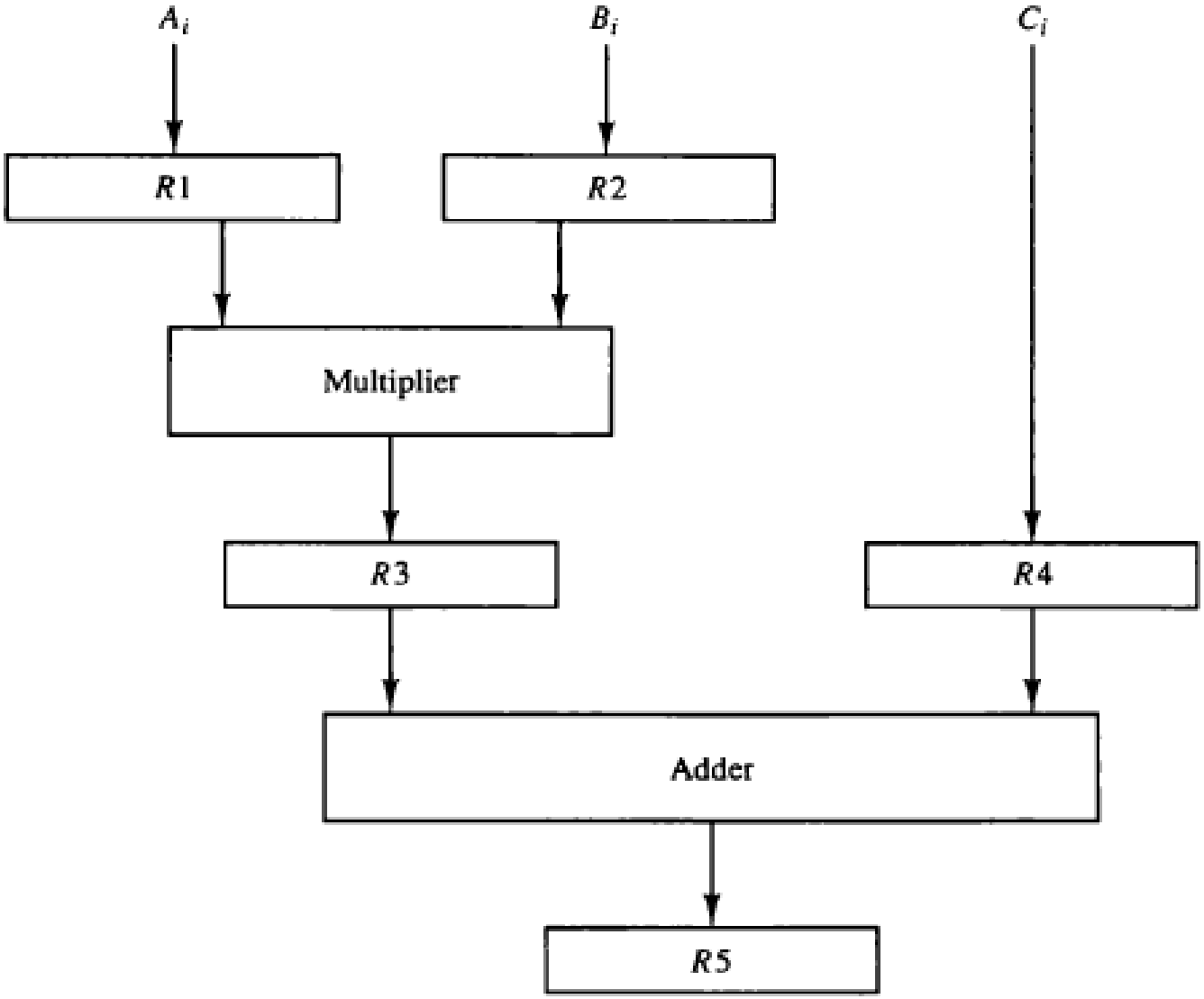
$$A_i * B_i + C_i \qquad \text{for } i = 1, 2, 3, \ldots, 7$$

The sub operations performed in each segment of the pipeline are as follows

$R1 \leftarrow A_i,\ \ R2 \leftarrow B_i$	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

. Each sub operation is to be implemented in a segment within a pipeline. Each segment has one or two registers with a combinational circuit.

Figure 9-2 Example of pipeline processing.



Arithmetic Pipeline

- Three numbers are associated with a floating point number

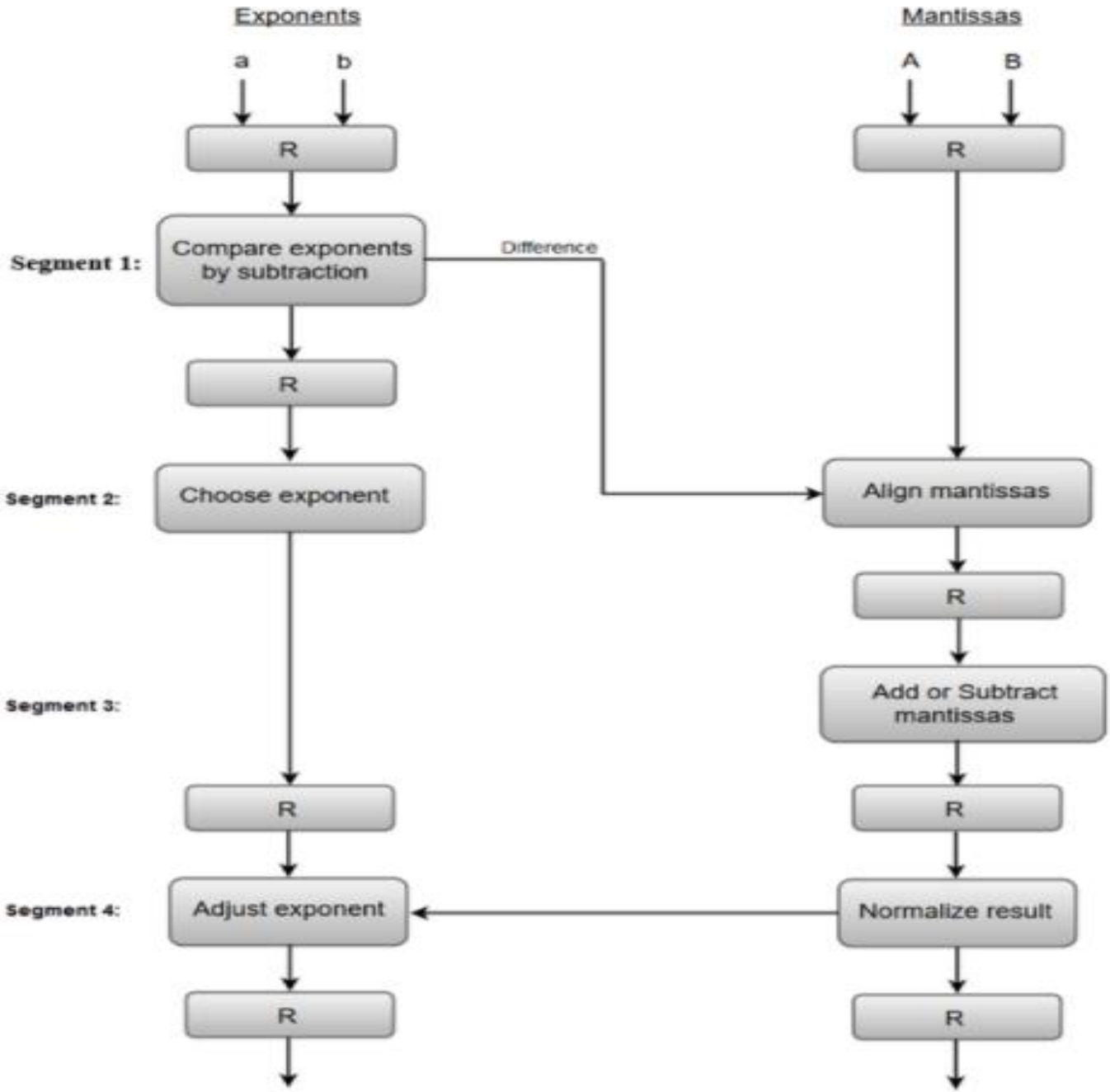
a) A Mantissa (M)

b) An exponent(E)

c) Base(radix) B

- Example : $M*B^E$

1. Compare the exponents of both added numbers
2. Align the Mantissa
3. Perform the addition and subtraction
4. Perform normalization by shifting the resulting mantissa and adjusting the resulting exponent.


Exponents
Mantissas
a
b
A
B
R
R
Segment 1:
Compare exponents by subtraction
Difference
R
Segment 2:
Choose exponent
Align mantissas
R
Segment 3:
Add or Subtract mantissas
R
R
Segment 4:
Adjust exponent
Normalize result
R
R

Example of Floating point Addition

- A= $1.1100 * 2^4$
- B= $1.1000 * 2^2$

- Alignment of B = $0.0110 * 2^4$
- Addition of A and B = $10.0010 * 2^4$
- Normalization: $0.1000 * 2^6$

Instruction Pipeline

In this a stream of instructions can be executed by overlapping fetch, decode and execute phases of an instruction cycle. This type of technique is used to increase the throughput of the computer system.

An instruction pipeline reads instruction from the memory while previous instructions are being executed in other segments of the pipeline. Thus we can execute multiple instructions simultaneously.

The pipeline will be more efficient if the instruction cycle is divided into segments of equal duration.

In the most general case computer needs to process each instruction in following sequence of steps:

- Fetch the instruction from memory (FI)
- Decode the instruction (DA)
- Calculate the effective address
- Fetch the operands from memory (FO)
- Execute the instruction (EX)
- Store the result in the proper place

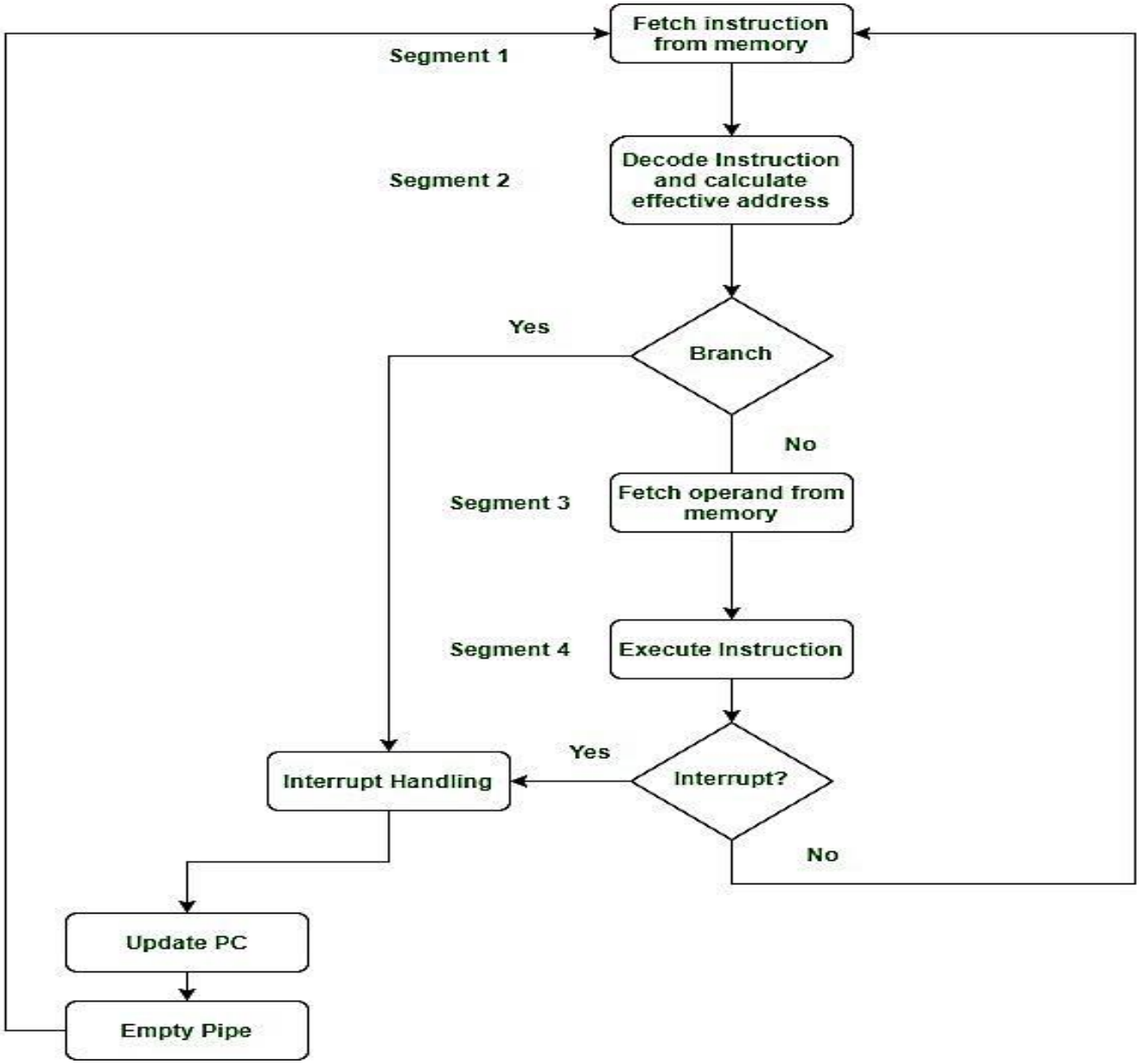
Four Segment Instruction Pipeline

Assume that the decoding of an instruction can be combined with the calculation of the effective address into one segment.

Assume that most of the instructions place the result into a process register so that the instruction execution and storing the result can be combined into one segment.

This reduce the instruction pipeline into four segments.

The flowchart for instruction pipeline is shown below.



Segment 1: (FI)

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

Segment 2: (DA)

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: (FO)

An operand from memory is fetched in the third segment.

Segment 4: (EX)

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

	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

Figure 9-8 Timing of instruction pipeline.

Pipeline Conflicts

In general, there are three major difficulties that cause the instruction pipeline to deviate from its normal operation.

Resource conflicts (Structural hazard) caused by access to memory by two segments at the same time. Most of these conflicts can be resolved by using separate instruction and data memories.

Data dependency (Data Hazard) conflicts arise when an instruction depends on the result of a previous instruction, but this result is not yet available.

Branch difficulties (Control Hazard) arise from branch and other instructions that change the value of PC.