Unit-5
Pipeline

Marwadi University

Department of Computer Engineering

Computer Organization and Architecture 01CE1402

Prof. Kishan Makadiya

Flynn's classification

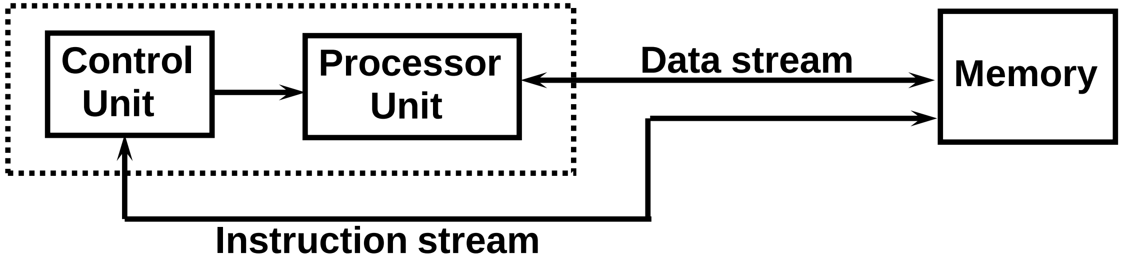
- Flynn's classification
  - Based on the multiplicity of Instruction Streams and Data Streams
- Instruction Stream
  - Sequence of Instructions read from memory
- Data Stream
  - Operations performed on the data in the processor

Flynn’s Taxonomy

		No of Data Stream	
		Single	Multiple
No of Instruction Stream	Single	SISD	SIMD
	Multiple	MISD	MIMD

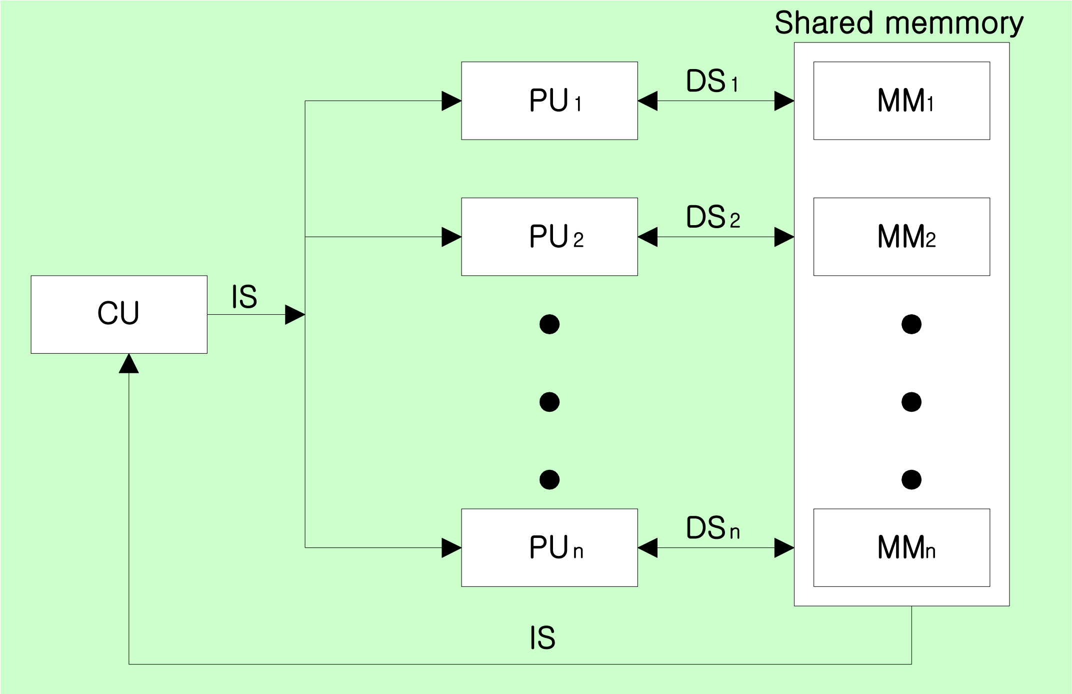
Single Instruction Single Data (SISD)

- SISD represents the organization of a single computer containing a control unit, a processor unit, and a memory unit.
- Instructions are executed sequentially and the system may or may not have internal parallel processing capabilities.



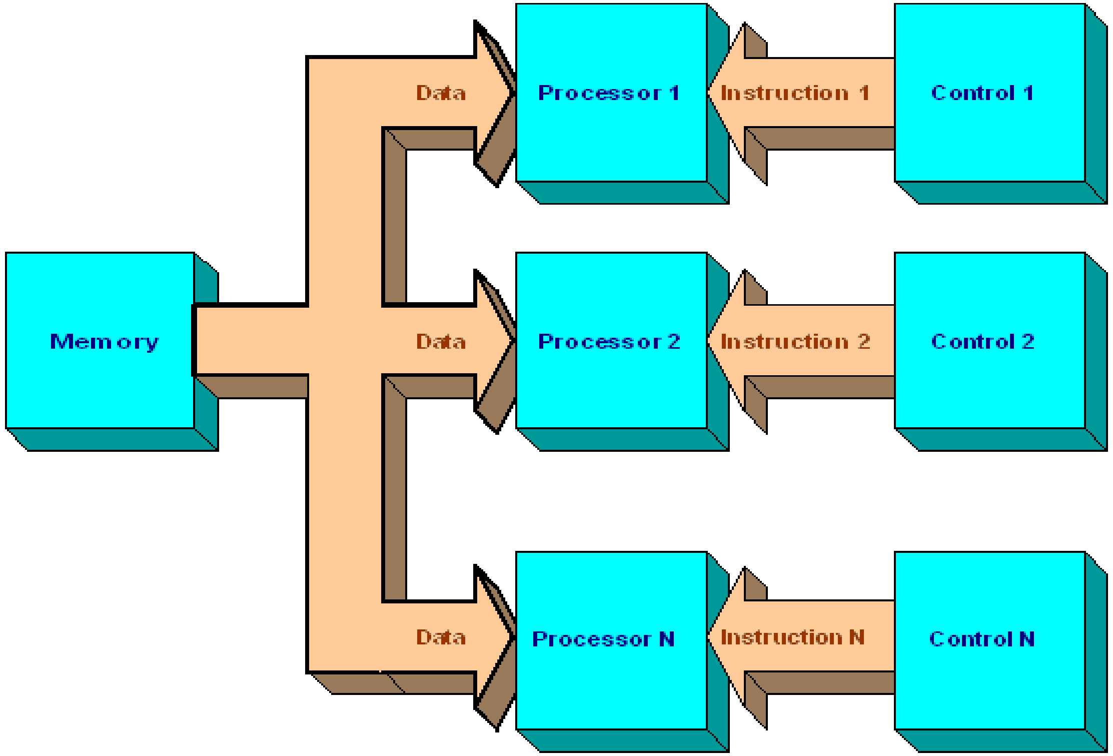
Single Instruction Multiple Data (SIMD)

- SIMD represents an organization that includes many processing units under the supervision of a common control unit.
- All processors receive the same instruction from the control unit but operate on different items of data.



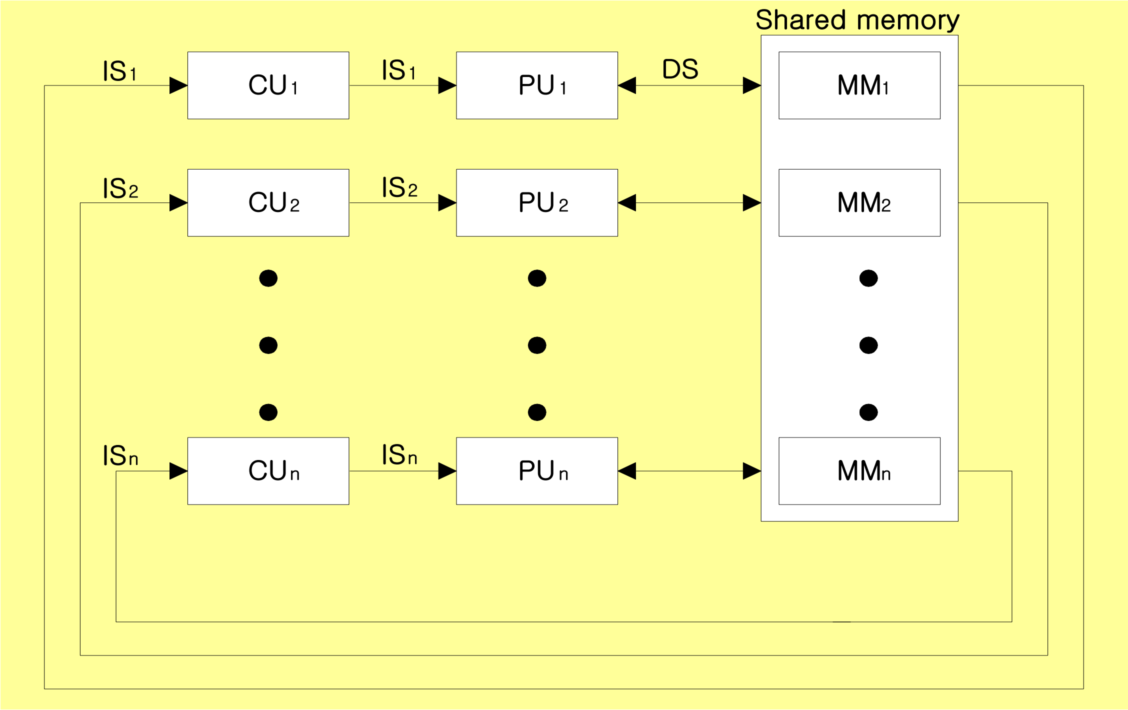
Multiple Instruction Single Data (MISD)

- There is no computer at present that can be classified as MISD.
- MISD structure is only of theoretical interest since no practical system has been constructed using this organization.



Multiple Instruction Multiple Data (MIMD)

- MIMD organization refers to a computer system capable of processing several programs at the same time.
- Most multiprocessor and multicomputer systems can be classified in this category.
- Contains multiple processing units.
- Execution of multiple instructions on multiple data.



Parallel Processing

- Parallel processing is a term used to denote a large class of techniques that are used to provide simultaneous data-processing tasks for the purpose of increasing the computational speed of a computer system.
- Purpose of parallel processing is to speed up the computer processing capability and increase its *throughput*.
- *Throughput*:

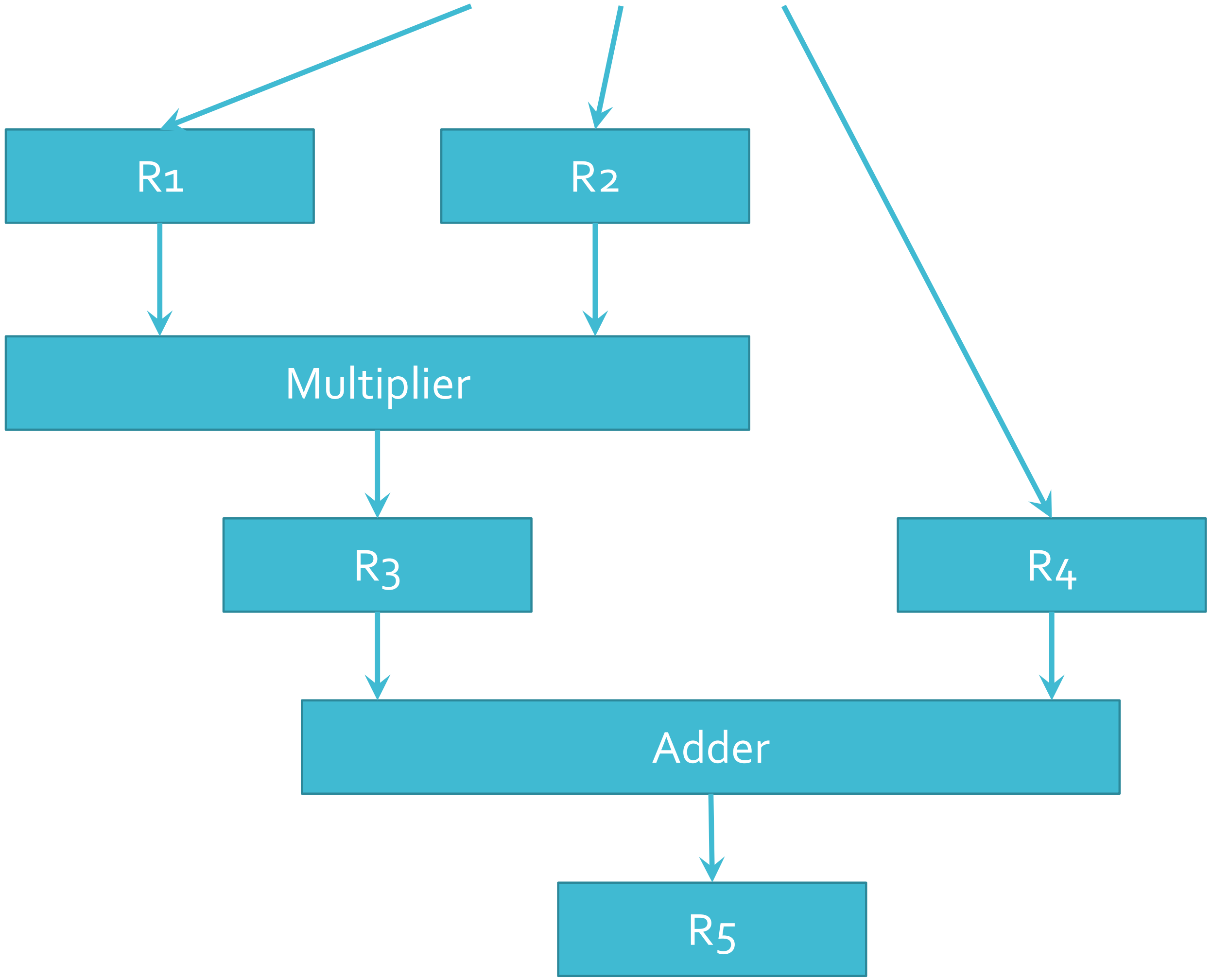
  The amount of processing that can be accomplished during a given interval of time.

Pipelining

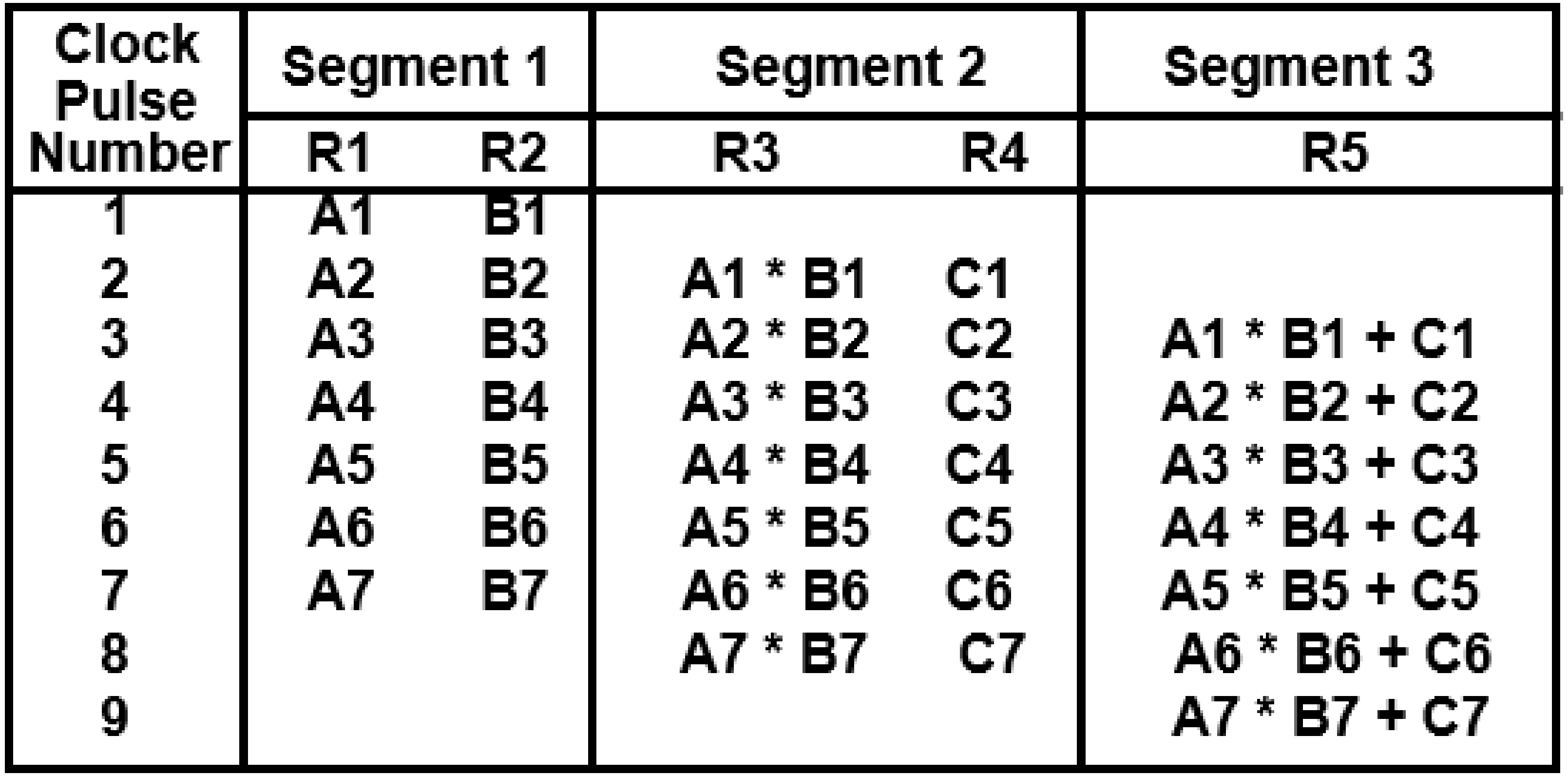
- Pipeline is a technique of decomposing a sequential process into sub operations, with each sub process being executed in a special dedicated segment that operates concurrently with all other segments.
- A pipeline can be visualized as a collection of processing segments through which binary information flows.
- Each segment performs partial processing dictated by the way the task is partitioned.
- The result obtained from the computation in each segment is transferred to the next segment in the pipeline.
- The registers provide isolation between each segment.
- The technique is efficient for those applications that need to repeat the same task many times with different sets of data.

Pipelining example

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



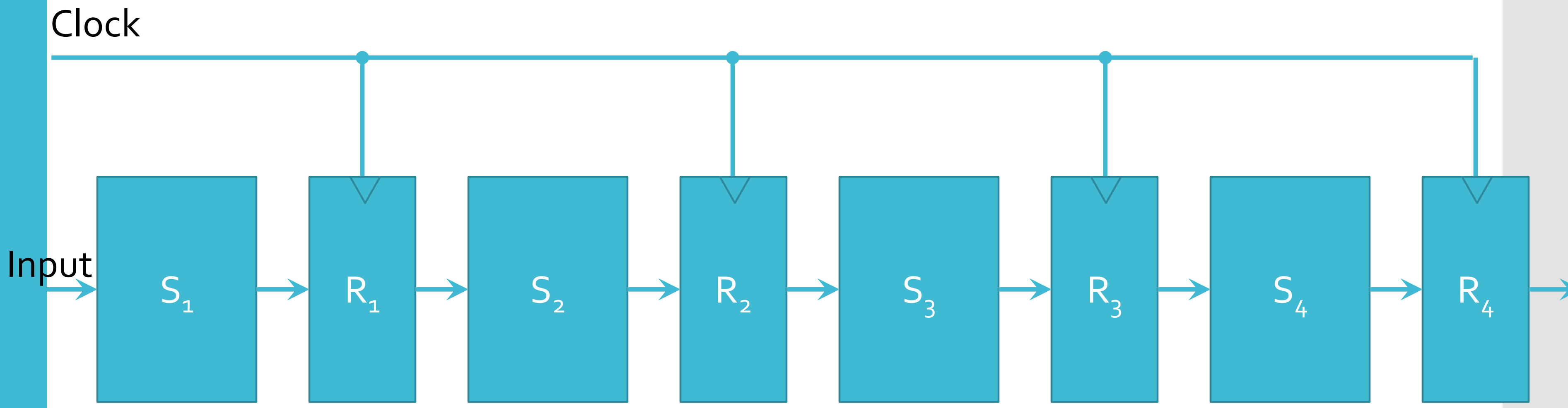
Operations In each Pipeline Stage



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

Pipelining

- General structure of four segment pipeline



Space-time Diagram

Segment	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	
1																	4
2																	4
3																	4
4																	4

Non Pipelined Architecture

= 16

Segment	1	2	3	4	5	6	7	
1								4
2								1
3								1
4								1

→ Clock cycles

= 7

Pipelined Architecture

Speedup

- **n**: Number of tasks to be performed
- Conventional Machine (Non-Pipelined)
  - **tn**: Clock cycle
  - **t1**: Time required to complete the n tasks
  - **t1 = n * tn**
- Pipelined Machine (k stages)
  - **tp**: Clock cycle (time to complete each sub-operation)
  - **tk**: Time required to complete the n tasks
  - **tk = (k + n - 1) * tp**
- Speedup
  - **S**: Speedup

Speedup

- Speedup of a pipeline processing over an equivalent non-pipeline processing is defined by the ratio

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

- If number of tasks in pipeline increases w.r.t. number of segments then n becomes larger than $k-1$, under this condition speedup becomes

$$S = \frac{nt_n}{nt_p} = \frac{t_n}{t_p}$$

- Assuming time to process a task in pipeline and non-pipeline circuit is same then

$$\boxed{S = \frac{kt_p}{t_p} = k}$$

- Theoretically maximum speedup achieved is the number of segments in the pipeline.

Pipeline And Multiple Function Units

- Example
- - 4-stage pipeline
- - sub-opertion in each stage; tp = 20 nS
- - 100 tasks to be executed
- - 1 task in non-pipelined system; 20*4 = 80 nS
- Pipelined System:-
- (k + n - 1)*tp = (4 + 99) * 20 = 2060 nS
- Non-Pipelined System:-
- n*k*tp = 100 * 4* 20 = 8000 nS
- Speedup:-
- Sk = Non-Pipelined System/ Pipelined System

  =8000 / 2060 = 3.88

Arithmetic Pipeline

- Usually found in high speed computers.
- Used to implement floating point operations, multiplication of fixed point numbers and similar operations.

Example of Arithmetic Pipeline

- Consider an example of floating point addition and subtraction.

$$X = A \ \times 10^a$$

$$Y = B \ \times 10^b$$

- A and B are two fractions that represent the mantissas and a and b are the exponents.

Example of Arithmetic Pipeline

- Consider the two normalized floating-point numbers:

  $X = 0.9504 \times 10^3$ $\quad$ $Y = 0.8200 \times 10^2$

- Segment-1: The larger exponent is chosen as the exponent of result.
- Segment-2: Aligning the mantissa numbers

  $X = 0.9504 \times 10^3$ $\quad$ $Y = 0.0820 \times 10^3$

- Segment-3: Addition of the two mantissas produces the sum

  $$Z = 1.0324 \times 10^3$$

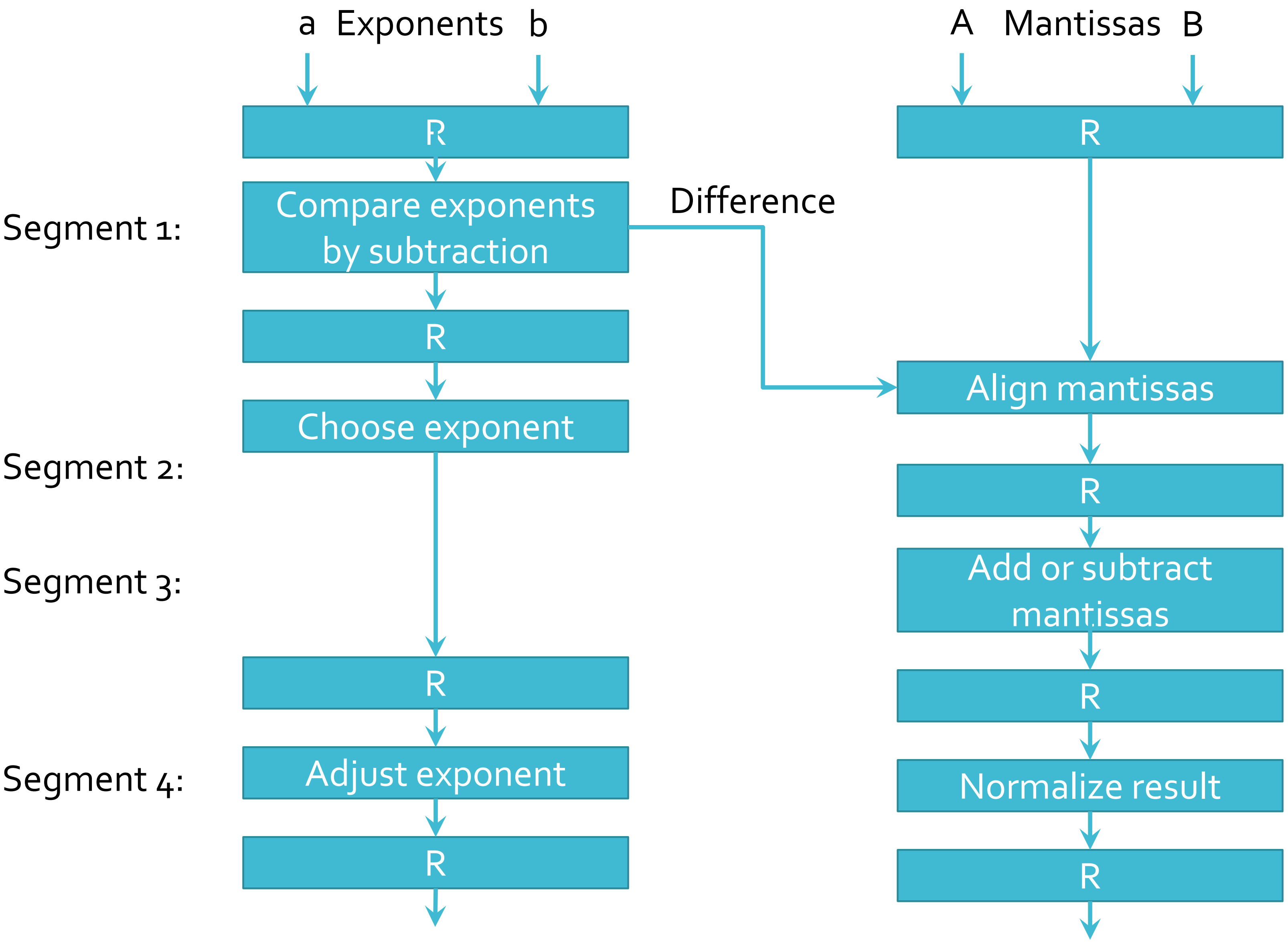
- Segment-4: Normalize the result

  $$Z = 0.10324 \times 10^4$$

Example of Arithmetic Pipeline

- The sub-operations that are performed in the four segments are:
  1. Compare the exponents
  2. Align the mantissas
  3. Add or subtract the mantissas
  4. Normalize the result

Arithmetic pipeline



Instruction Pipeline

- In the most general case, the computer needs to process each instruction with the following sequence of steps
  1. Fetch the 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.
- Different segments may take different times to operate on the incoming information.
- Some segments are skipped for certain operations.
- The design of an instruction pipeline will be most efficient if the instruction cycle is divided into segments of equal duration.

Instruction Pipeline

- Assume that the decoding of the instruction can be combined with the calculation of the effective address into one segment.
- Assume further that most of the instructions place the result into a processor registers so that the instruction execution and storing of the result can be combined into one segment.
- This reduces the instruction pipeline into four segments.
  1. FI: Fetch an instruction from memory
  2. DA: Decode the instruction and calculate the effective address of the operand
  3. FO: Fetch the operand
  4. EX: Execute the operation

Instruction Pipeline

- Execution of Three Instructions in a 4-Stage Pipeline

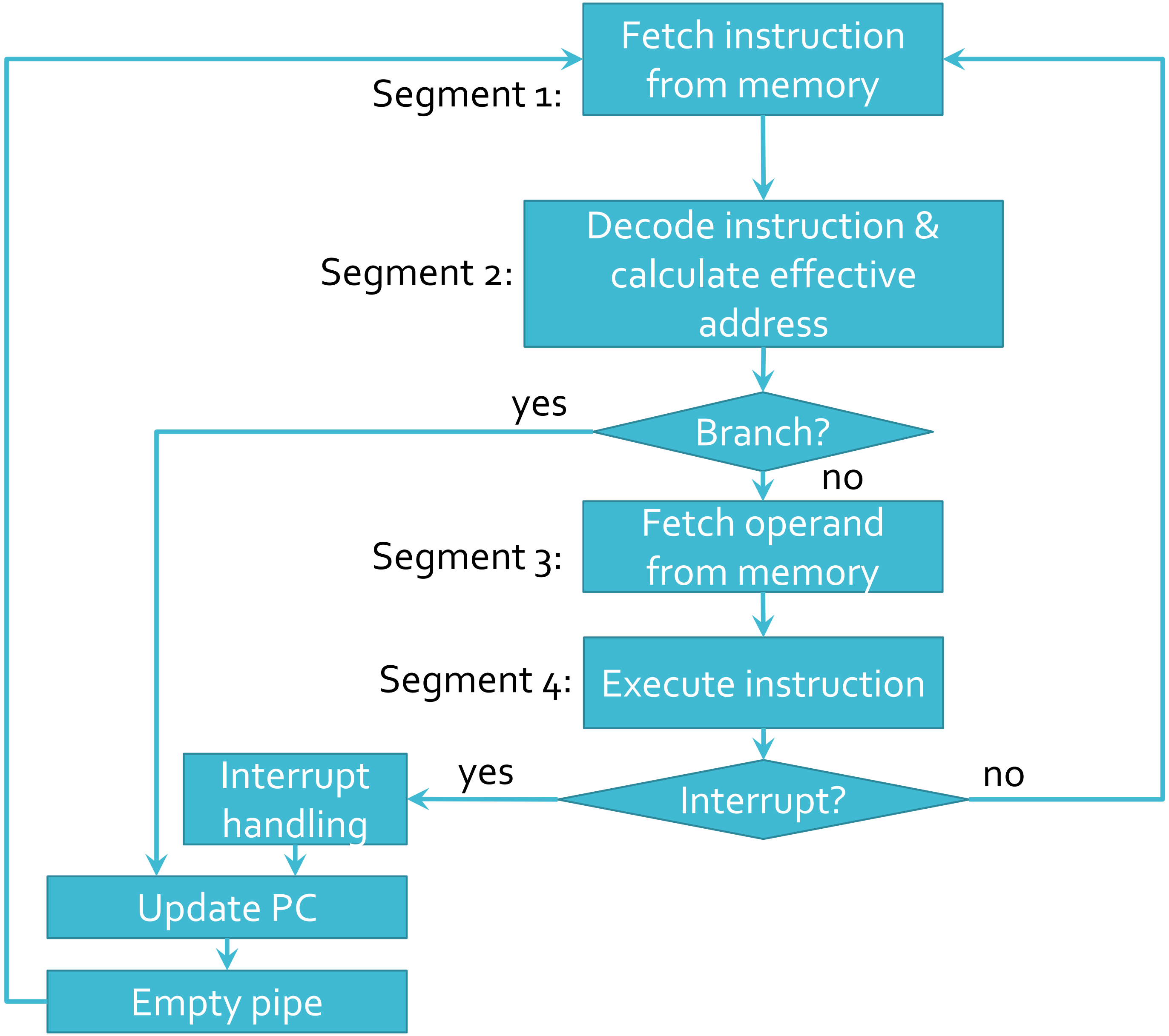
	1	2	3	4	5	6	7	8	9	10	11	12
1	FI	DA	FO	EX								
2					FI	DA	FO	EX				
3									FI	DA	FO	EX

Conventional

	1	2	3	4	5	6
1	FI	DA	FO	EX		
2		FI	DA	FO	EX	
3			FI	DA	FO	EX

Pipelined

Four segment CPU pipeline



Space-time Diagram

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

Pipeline Conflict

- There are three major difficulties that cause the instruction pipeline conflicts.
  1. Resource conflicts caused by access to memory by two segments at the same time.
  2. Data dependency conflicts arise when an instruction depends on the result of a previous instruction, but this result is not yet available.
  3. Branch difficulties arise from branch and other instructions that change the value of PC.

Data Dependency

- Data dependency occurs when an instruction needs data that are not yet available.
- Pipelined computers deal with such conflicts between data dependencies in a variety of ways as follows:
  1. Hardware Interlocks
  2. Operand forwarding
  3. Delayed load

Handling Branch Instructions

- The branch instruction breaks the normal sequence of the instruction stream, causing difficulties in the operation of the instruction pipeline.
- Hardware techniques available to minimize the performance degradation caused by instruction branching are as follows:
  - Pre-fetch instruction
  - Branch target buffer
  - Loop buffer
  - Branch prediction
  - Delayed branch

RISC Pipeline

- RISC
  - Machine with a very fast clock cycle that executes at the rate of one instruction per cycle
  - Simple Instruction Set
  - Fixed Length Instruction Format
  - Register-to-Register Operations
- Instruction Cycles of Three-Stage Instruction Pipeline
- Data Manipulation Instructions
  - I: Instruction Fetch
  - A: Decode, Read Registers, ALU Operations
  - E: Write a Register
- Load and Store Instructions
  - I: Instruction Fetch
  - A: Decode, Evaluate Effective Address
  - E: Register-to-Memory or Memory-to-Register
- Program Control Instructions
  - I: Instruction Fetch
  - A: Decode, Evaluate Branch Address
  - E: Write Register(PC)

Delayed Load

LOAD: R1 ← M[address 1]
LOAD: R2 ← M[address 2]
ADD: R3 ← R1 + R2
STORE: M[address 3] ← R3

The concept of delaying the use of the data loaded from memory is referred to as **delayed load**.

- Three-segment pipeline timing

Pipeline timing with data conflict

clock cycle	1	2	3	4	5	6
Load R1	I	A	E			
Load R2		I	A	E		
Add R1+R2			I	A	E	
Store R3				I	A	E

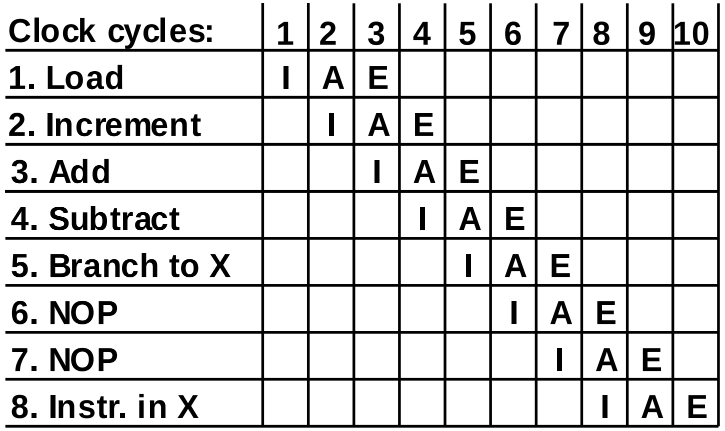
Pipeline timing with delayed load

clock cycle	1	2	3	4	5	6	7
Load R1	I	A	E				
Load R2		I	A	E			
NOP			I	A	E		
Add R1+R2				I	A	E	
Store R3					I	A	E

The data dependency is taken care by the compiler rather than the hardware

Delayed Branch

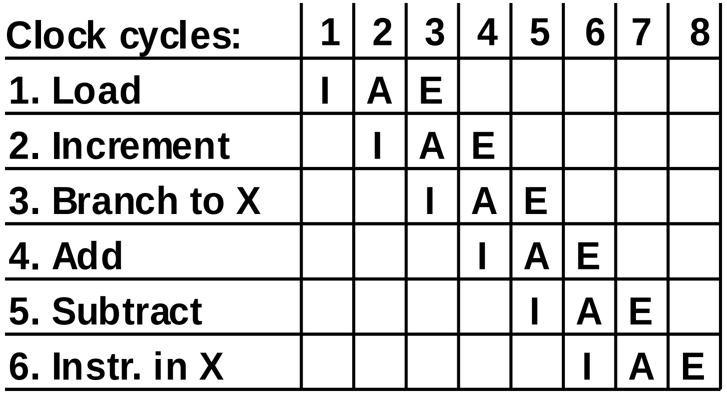
- Compiler analyzes the instructions before and after the branch and rearranges the program sequence by inserting useful instructions in the delay steps.
- Using no-operation instructions



Clock cycles:	1	2	3	4	5	6	7	8	9	10
1. Load	I	A	E							
2. Increment		I	A	E						
3. Add			I	A	E					
4. Subtract				I	A	E				
5. Branch to X					I	A	E			
6. NOP						I	A	E		
7. NOP							I	A	E	
8. Instr. in X								I	A	E

LOAD from memory to R1
Increment R2
Add R3 to R4
Subtract R5 from R6
Branch to address X
Instruction in X

- Rearranging the instructions



Clock cycles:	1	2	3	4	5	6	7	8
1. Load	I	A	E					
2. Increment		I	A	E				
3. Branch to X			I	A	E			
4. Add				I	A	E		
5. Subtract					I	A	E	
6. Instr. in X						I	A	E

ThankYou !!!