

**Diyala University /College of Engineering/Computer Engineering Department**

# **Parallel Processing**

*Based on M. Morris Mano “Computer System Architecture”--Assist. Lecturer Ahmed Salah Hameed*

# Pipeline and Vector Processing

# 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.
- concurrent data processing to achieve faster execution.

- **The purpose:**

1. **Speed up the computer processing capability**
2. **Increase the throughput**

*Throughput* the amount of processing that can be accomplished during a given interval of time.

- **The side effects**

1. **The amount of hardware increases**
2. **The cost of the system increases**

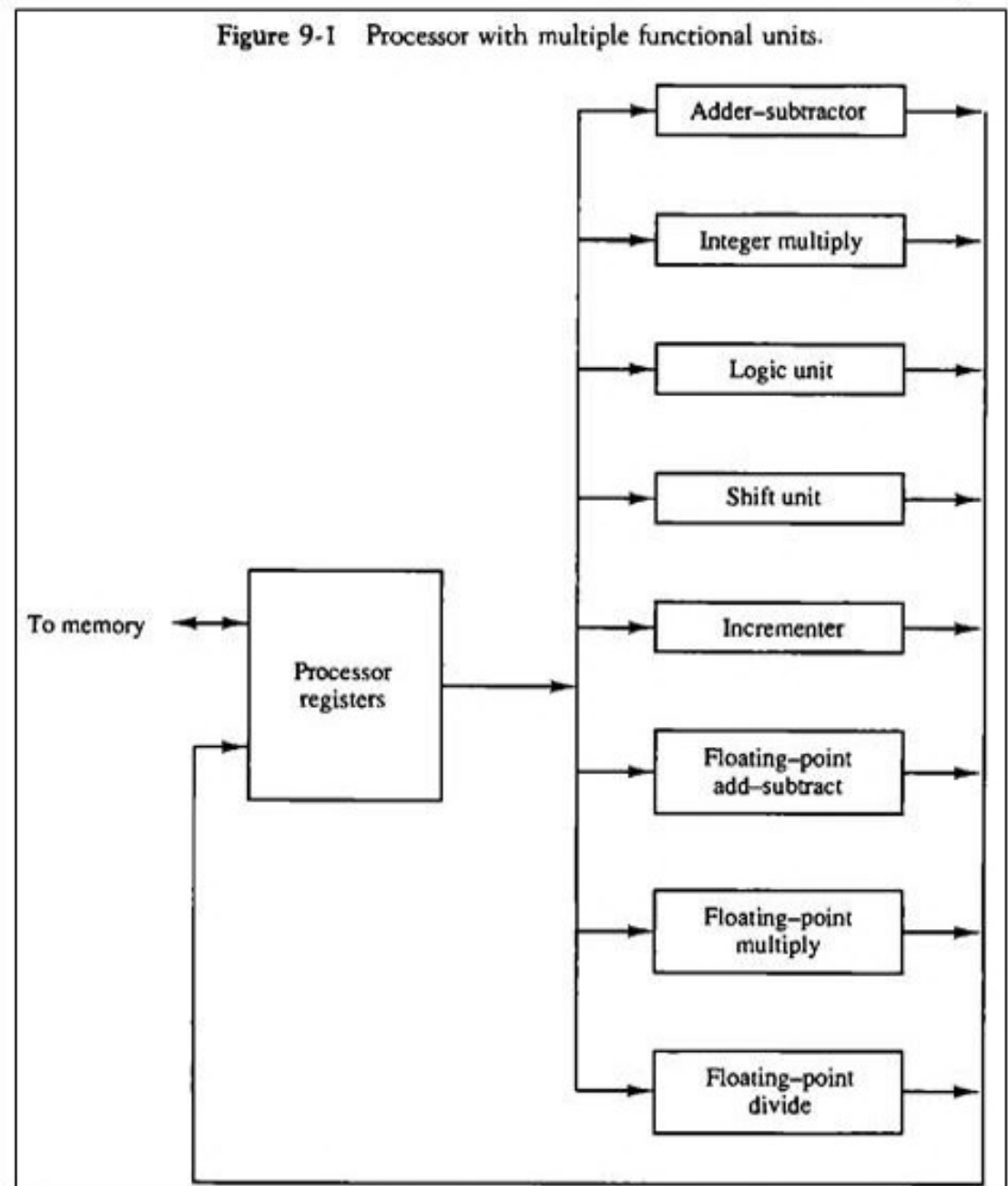
# Parallel processing / levels of complexity

- At the lower level

Serial Shift register VS parallel load registers

- At the higher level

Multiplicity of functional units that perform identical or different operations simultaneously.



# Parallel processing or parallel computers

- It can be considered
  1. from the internal organization of the processors
  2. from the interconnection structure between processors
  3. or from the flow of information through the system.

## Flynn's classification

### » Instruction Stream

- Sequence of Instructions read from memory

### » Data Stream

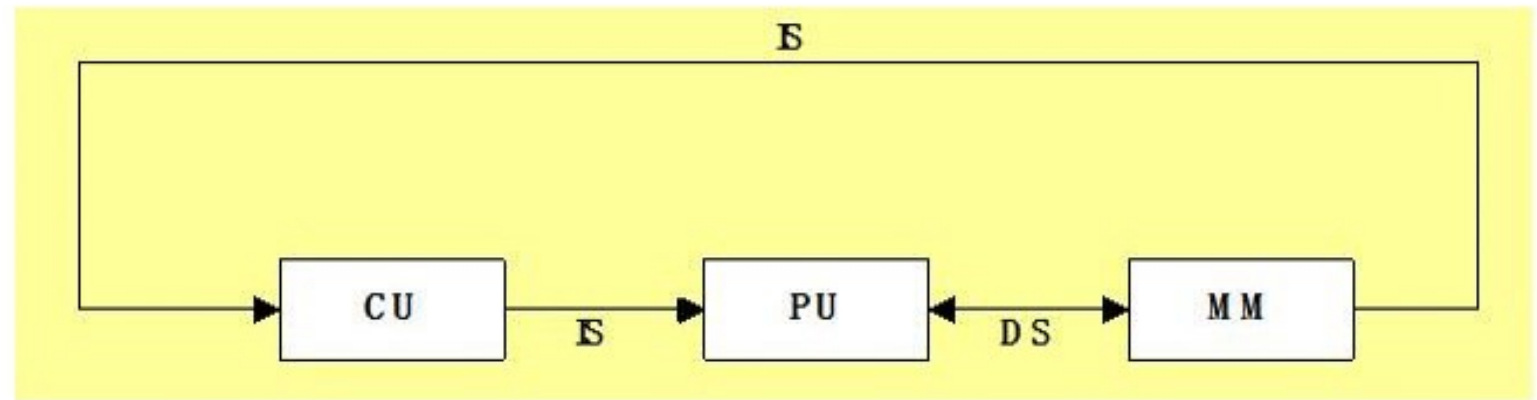
- Operations performed on the data in the processor

		Number of <i>Data Streams</i>	
		Single	Multiple
Number of <i>Instruction Streams</i>	Single	SISD	SIMD
	Multiple	MISD	MIMD

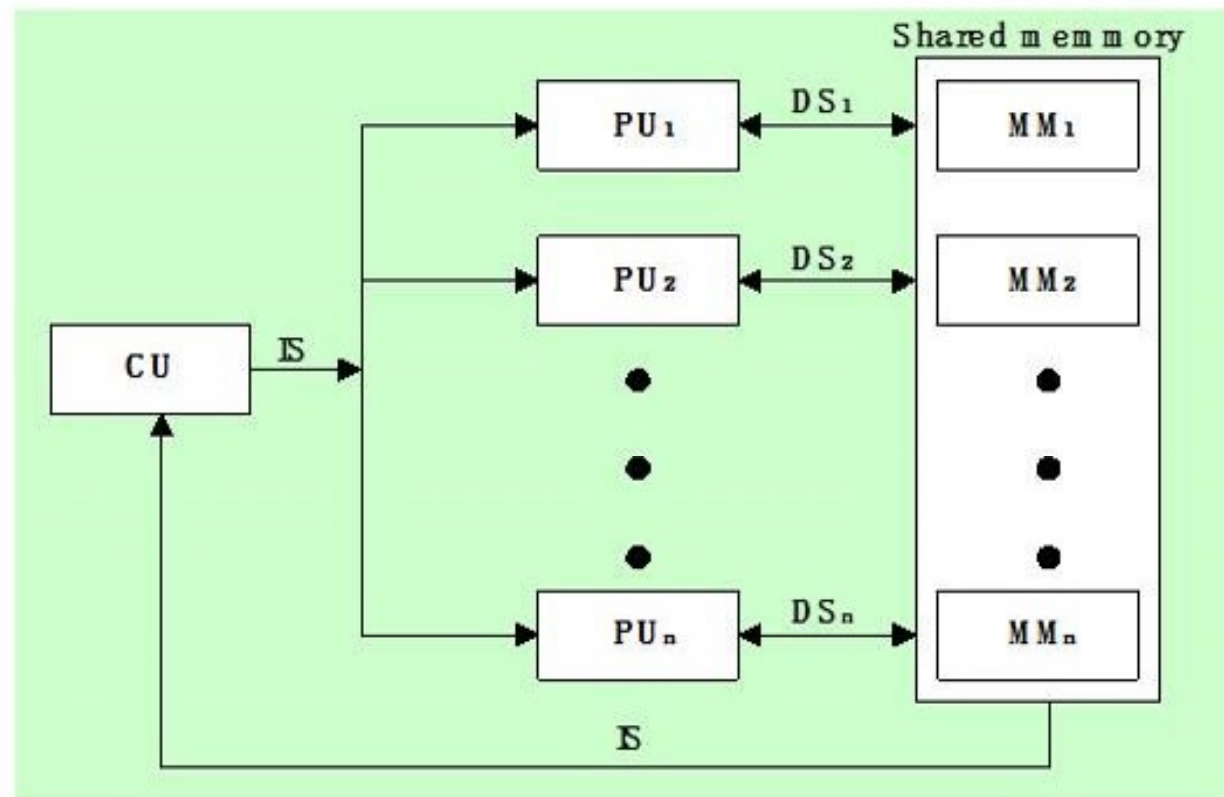


# Flynn's classification

## 1. SISD

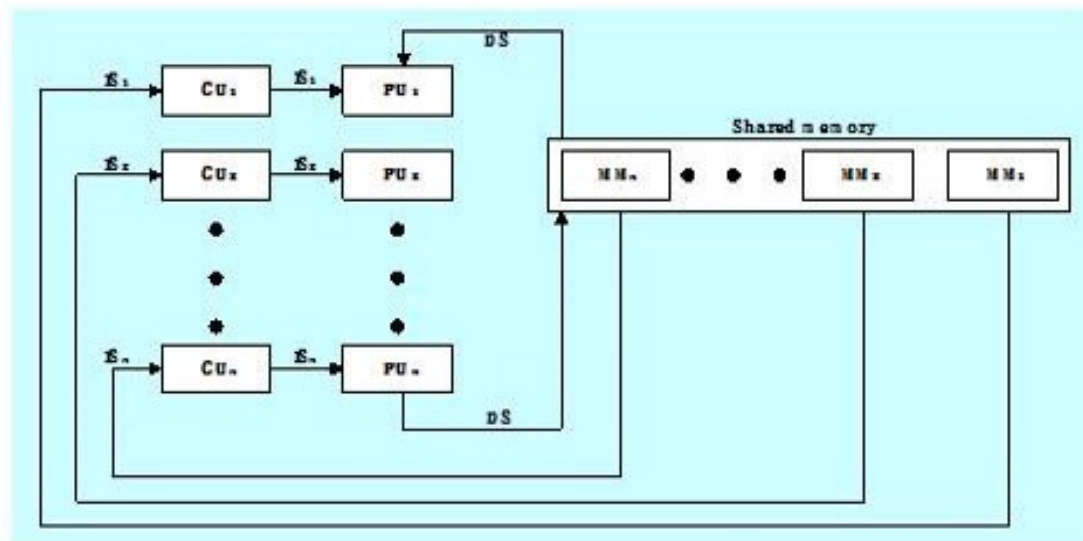


## 2. SIMD

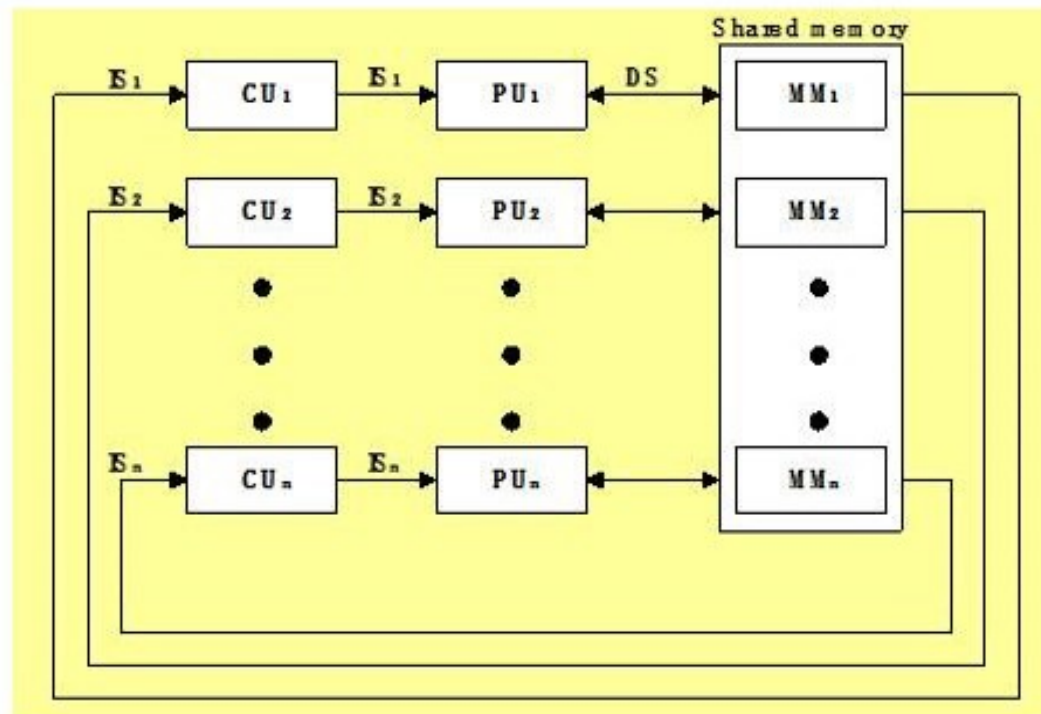


# Flynn's classification

## 3. MISD



## 4. MIMD



# Main topics of the Chapter

- **Pipeline processing : Sec. 9-2**
  - 1) Arithmetic pipeline : Sec. 9-3
  - 2) Instruction pipeline : Sec. 9-4
- **Vector processing : adder/multiplier pipeline , Sec. 9-6**
- **Array processing : array processor , Sec. 9-7**
  - 1) Attached array processor : Fig. 9-14
  - 2) SIMD array processor : Fig. 9-15



# 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 name **“pipeline”** implies a flow of information analogous to an industrial assembly line.

# Example of the Pipeline Organization

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

$R1 \leftarrow A_i, R2 \leftarrow B_i$

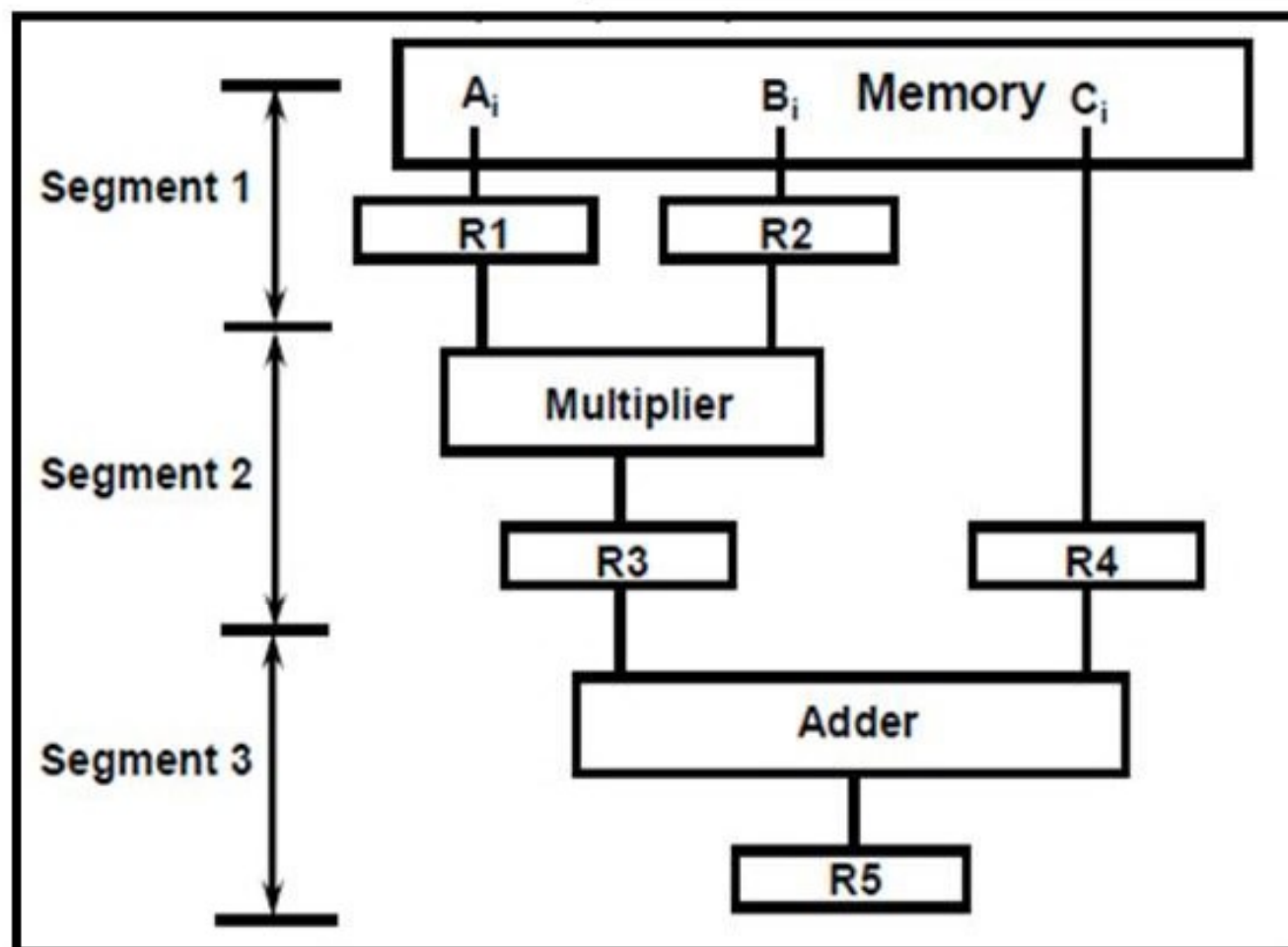
Input  $A_i$  and  $B_i$

$R3 \leftarrow R1 * R2, R4 \leftarrow C_i$

Multiply and input  $C_i$

$R5 \leftarrow R3 + R4$

Add  $C_i$  to product



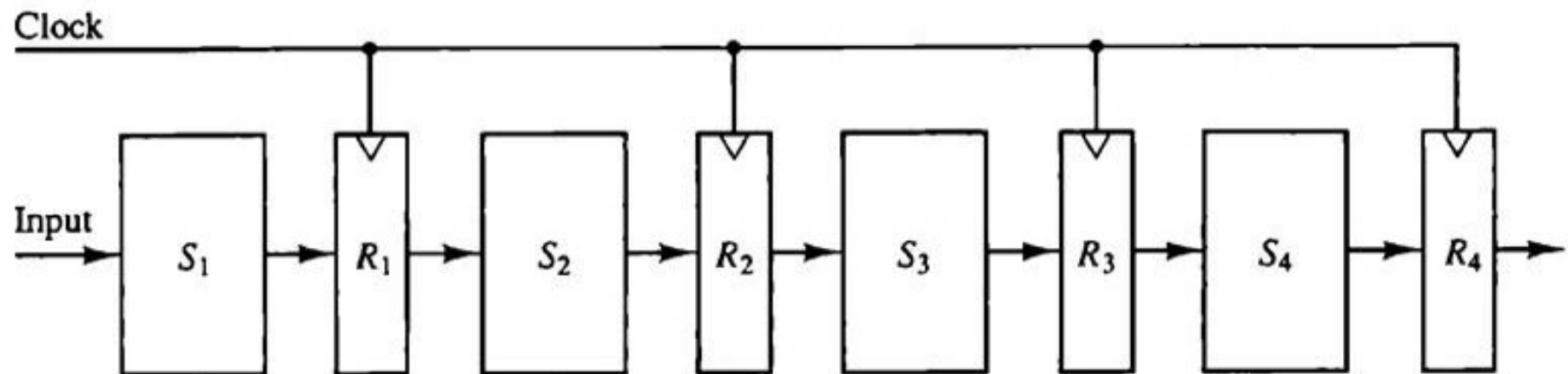
# Example of the Pipeline Organization

TABLE 9-1 Content of Registers in Pipeline Example

Clock Pulse Number	Segment 1		Segment 2		Segment 3
	R1	R2	R3	R4	R5
1	$A_1$	$B_1$	—	—	—
2	$A_2$	$B_2$	$A_1 * B_1$	$C_1$	—
3	$A_3$	$B_3$	$A_2 * B_2$	$C_2$	$A_1 * B_1 + C_1$
4	$A_4$	$B_4$	$A_3 * B_3$	$C_3$	$A_2 * B_2 + C_2$
5	$A_5$	$B_5$	$A_4 * B_4$	$C_4$	$A_3 * B_3 + C_3$
6	$A_6$	$B_6$	$A_5 * B_5$	$C_5$	$A_4 * B_4 + C_4$
7	$A_7$	$B_7$	$A_6 * B_6$	$C_6$	$A_5 * B_5 + C_5$
8	—	—	$A_7 * B_7$	$C_7$	$A_6 * B_6 + C_6$
9	—	—	—	—	$A_7 * B_7 + C_7$

# GENERAL PIPELINE

- Any operation that can be decomposed into a sequence of sub operations of about the same complexity can be implemented by a pipeline processor.
- The technique is efficient for those applications that need to repeat the same task many times with different sets of a task as the total operation performed going through all the segments in the pipeline.



**Figure 9-3** Four-segment pipeline.

# SPEED CALCULATION

- **K-segment** pipeline with a **clock cycle time (tp)** is used to execute **n tasks**.
- The first task **T1** requires a time equal to **(k\*tp)** to complete its operation because we have **k** segments in the pipe.
- The remaining **n - 1 tasks** emerge from the pipe at the rate of one task per clock cycle and they will be completed after a time equal to **(n - 1)\*tp**.
- Therefore, to complete **n tasks** using a **k-segment** pipeline:

$$k + (n - 1) \text{ clock cycles}$$

**For example** system with **four** segments and **six** tasks.

The time required to complete all the operations is/

$$4 + (6 - 1) = 9 \text{ clock cycles}$$



# SPEED CALCULATION

Figure 9-4 Space-time diagram for pipeline.

		1	2	3	4	5	6	7	8	9	→ Clock cycles
Segment:	1	$T_1$	$T_2$	$T_3$	$T_4$	$T_5$	$T_6$				
	2		$T_1$	$T_2$	$T_3$	$T_4$	$T_5$	$T_6$			
	3			$T_1$	$T_2$	$T_3$	$T_4$	$T_5$	$T_6$		
	4				$T_1$	$T_2$	$T_3$	$T_4$	$T_5$	$T_6$	

# SPEED CALCULATION

## NONPIPELINE UNIT

- Each task take time equal to  **$t_n$** .
- The total time required for  **$n$**  tasks is

$$n * t_n$$

## PIPELINE UNIT

- To complete  **$n$  tasks** using a  **$k$ -segment** pipeline:

$$k + (n - 1) \text{ clock cycles}$$

- Total time is:  $k + (n - 1) * t_p$

## SPEED UP

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

# SPEED CALCULATION

## NOTES

- As the number of tasks increases,  $n$  becomes much larger than  $k - 1$ , and  $k + n - 1$  approaches the value of  $n$ . Under this condition, the speedup becomes:

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

- If we assume that the time it takes to process a task is the same in the pipeline and nonpipeline circuits, we will have  $t_n = k * t_p$ . So the speedup reduces to:

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

# EXAMPLE ON SPEED CALCULATION

## Example

- 4-stage pipeline
- suboperation in each stage;  $t_p = 20\text{nS}$
- 100 tasks to be executed
- 1 task in non-pipelined system;  $20 \times 4 = 80\text{nS}$

## Pipelined System

$$(k + n - 1) \times t_p = (4 + 99) \times 20 = 2060\text{nS}$$

## Non-Pipelined System

$$t_n = n \times k \times t_p = 100 \times 80 = 8000\text{nS}$$

## Speedup

$$S_k = 8000 / 2060 = 3.88$$

**4-Stage Pipeline is basically identical to the system with 4 identical function units**

# ARITHMETIC PIPELINE

- **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.

**EXAPLE:**

## FLOATING POINT ADDER



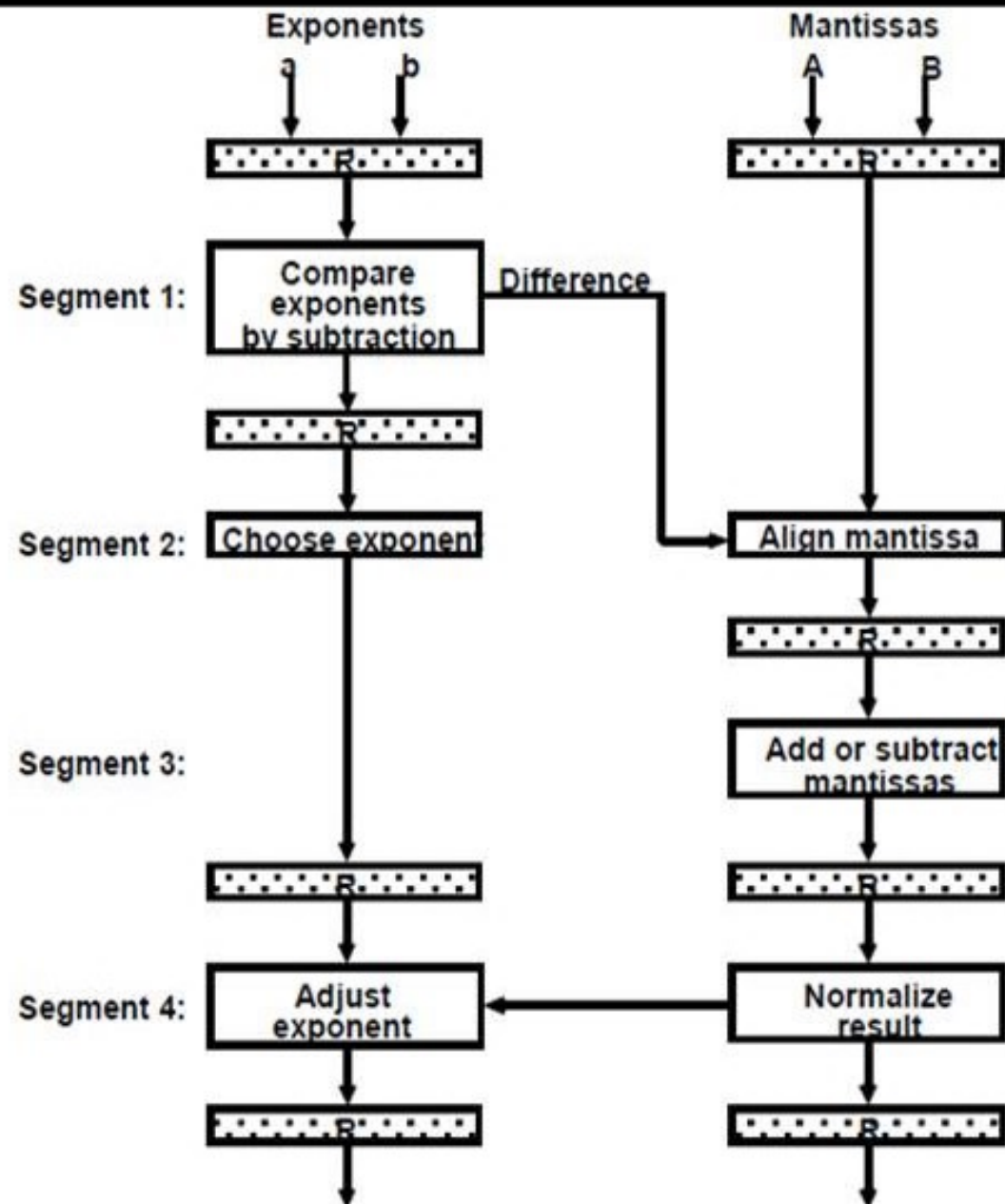
# ARITHMETIC PIPELINE

## Floating-point adder

$$X = A \times 2^a$$

$$Y = B \times 2^b$$

- [1] Compare the exponents
- [2] Align the mantissa
- [3] Add/sub the mantissa
- [4] Normalize the result



# ARITHMETIC PIPELINE

$$X = 0.9504 \times 10^3$$

$$Y = 0.8200 \times 10^2$$

1. Compare the exponents.
2. Align the mantissas.
3. Add or subtract the mantissas.
4. Normalize the result.

$$X = 0.9504 \times 10^3$$

$$Y = 0.0820 \times 10^3$$

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

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

# ARITHMETIC PIPELINE EXAMPLE

- Suppose that the time delays of the four segments are as following:

1.  $t_1 = 60 \text{ ns}$
2.  $t_2 = 70 \text{ ns}$
3.  $t_3 = 100 \text{ ns}$
4.  $t_4 = 80 \text{ ns}$
5. the interface registers have a delay of  $t_r = 10 \text{ ns}$ .

## Pipeline

- The clock cycle is chosen to be:  
 $t_p = t_3 + t_r = 110 \text{ ns}$ .

## Non Pipeline

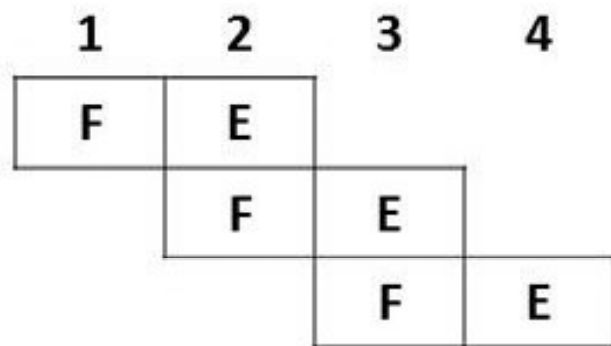
- Delay time  $t_n = t_1 + t_2 + t_3 + t_4 + t_r = 320 \text{ ns}$ .

## Speed up

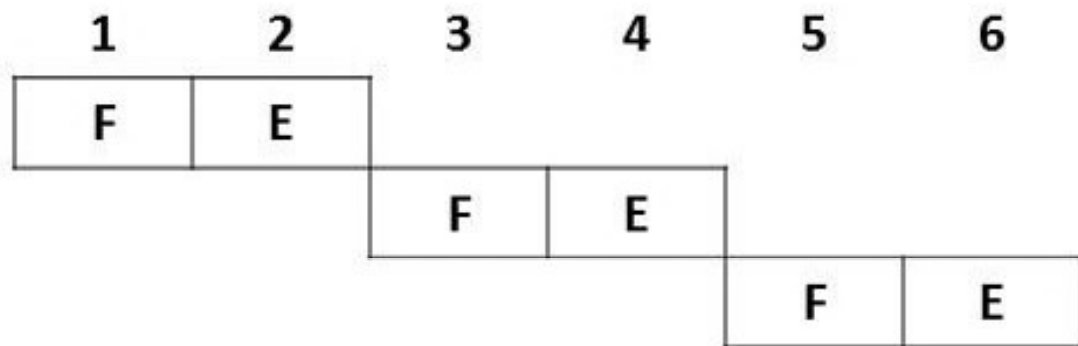
- the pipelined adder has a speedup of  $320/110 = 2.9$  over the non pipelined adder.

# INSTRUCTION PIPELINE

- **An instruction pipeline** reads consecutive instructions from memory while previous instructions are being executed in other segments. This causes the instruction fetch and execute phases to overlap and perform simultaneous operations.
- **Simple Example:**
- Consider a computer with an instruction fetch unit and an instruction execution unit designed to provide a two-segment pipeline.



2-segment pipelined



Non pipelined

# INSTRUCTION PIPELINE

- **NOTE 1:** Computers with complex instructions require other phases in addition to the fetch and execute to process an instruction completely.
  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.
- **NOTE 2:** Difficulties that will prevent the instruction pipeline from operating at its maximum rate.
  1. Different segments may take different times to operate on the incoming information.
  2. Some segments are skipped for certain operations. For example, a register mode instruction does not need an effective address calculation.
  3. Two or more segments may require memory access at the same time, causing one segment to wait until another is finished with the memory.



# INSTRUCTION PIPELINE

- **NOTE 3:** Memory access conflicts are sometimes resolved by using two memory buses for accessing instructions and data in separate modules. In this way, an instruction word and a data word can be read simultaneously from two different modules.
- **NOTE 4:** The design of an instruction pipeline will be most efficient if the instruction cycle is divided into segments of equal duration. The time that each step takes to fulfill its function depends on the instruction and the way it is executed.

# EXAMPLE: FOUR-SEGMENT INSTRUCTION PIPELINE

Segment1: Fetch instruction from memory

Segment2: Decode instruction and calculate effective address

Branch?

Segment3: Fetch operand from memory

Segment4: Execute instruction

Interrupt?

Interrupt handling

Update PC

Empty pipe

1. **FI** is the segment that fetches an instruction.
2. **DA** is the segment that decodes the instruction and calculates the effective address.
3. **FO** is the segment that fetches the operand.
4. **EX** is the segment that executes the instruction.

Step:		1	2	3	4	5	6	7	8	9	10	11	12	13
Instruction: (Branch)	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

Figure 9-8 Timing of instruction pipeline.

# CONFLICTS OF INSTRUCTION PIPELINE

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

1. *Resource conflicts* 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.
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

Occurs when the execution of an instruction depends on the results of a previous instruction

```
ADD    R1, R2, R3
SUB    R4, R1, R5
```

Data hazard can be dealt with either hardware techniques or software technique

## Hardware Technique

### *Interlock*

- hardware detects the data dependencies and delays the scheduling of the dependent instruction by stalling enough clock cycles

### *Forwarding* (bypassing, short-circuiting)

- Accomplished by a data path that routes a value from a source (usually an ALU) to a user, bypassing a designated register. This allows the value to be produced to be used at an earlier stage in the pipeline than would otherwise be possible

## Software Technique

Instruction Scheduling(compiler) for *delayed load*

# OPERAND FORWARDING

Example:

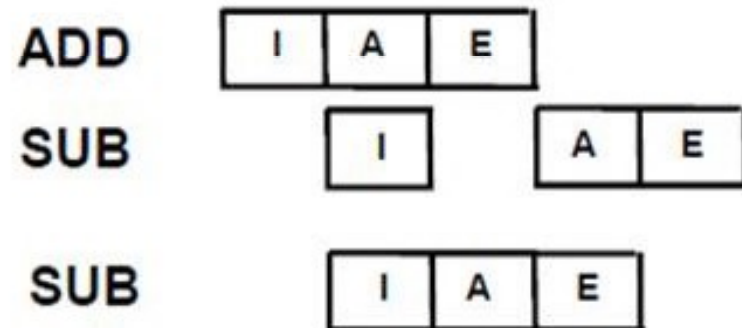
ADD R1, R2, R3  
SUB R4, R1, R5

3-stage Pipeline

I: Instruction Fetch

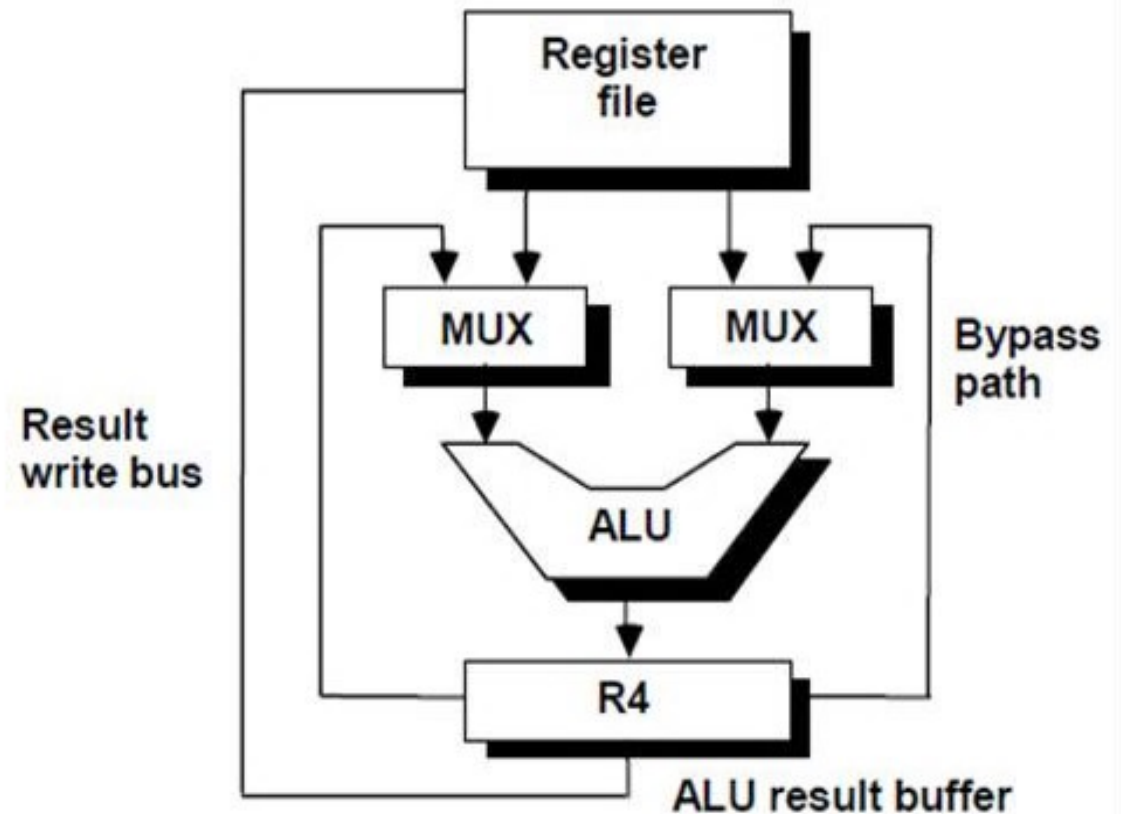
A: Decode, Read Registers,  
ALU Operations

E: Write the result to the  
destination register



Without Bypassing

With Bypassing





# DELAYED LOAD

$a = b + c;$   
 $d = e - f;$

## Unscheduled code:

```
      LW      Rb, b
      LW      Rc, c
→     ADD     Ra, Rb, Rc
→     SW      a, Ra
      LW      Re, e
      LW      Rf, f
→     SUB     Rd, Re, Rf
→     SW      d, Rd
```

## Scheduled Code:

```
      LW      Rb, b
      LW      Rc, c
      LW      Re, e
      ADD     Ra, Rb, Rc
      LW      Rf, f
      SW      a, Ra
      SUB     Rd, Re, Rf
→     SW      d, Rd
```

## Delayed Load

A load requiring that the following instruction not use its result

# DELAYED LOAD

1. LOAD:  $R1 \leftarrow M[\text{address } 1]$
2. LOAD:  $R2 \leftarrow M[\text{address } 2]$
3. ADD:  $R3 \leftarrow R1 + R2$
4. STORE:  $M[\text{address } 3] \leftarrow R3$

Clock cycles:	1	2	3	4	5	6
1. Load $R1$	I	A	E			
2. Load $R2$		I	A	E		
3. Add $R1 + R2$			I	A	E	
4. Store $R3$				I	A	E

(a) Pipeline timing with data conflict

Clock cycle:	1	2	3	4	5	6	7
1. Load $R1$	I	A	E				
2. Load $R2$		I	A	E			
3. No-operation			I	A	E		
4. Add $R1 + R2$				I	A	E	
5. Store $R3$					I	A	E

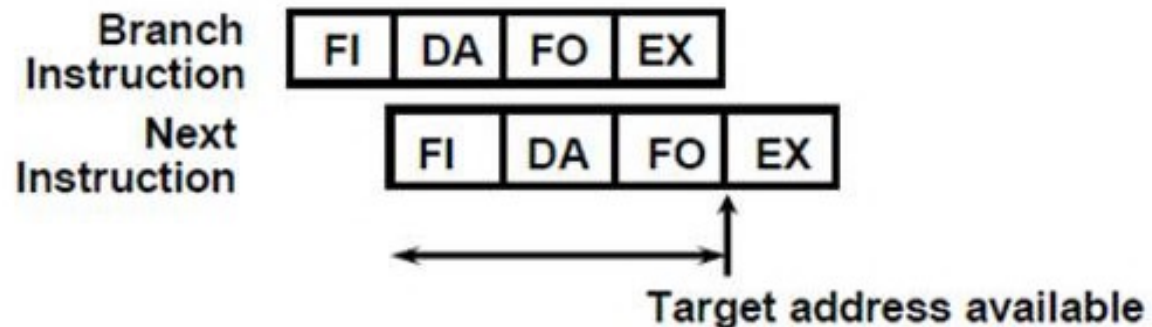
(b) Pipeline timing with delayed load

Figure 9-9 Three-segment pipeline timing.

# HANDLING OF BRANCH INSTRUCTIONS

## Branch Instructions

- Branch target address is not known until the branch instruction is completed



- Stall -> waste of cycle times

## Dealing with Control Hazards

- \* Prefetch Target Instruction
- \* Branch Target Buffer
- \* Loop Buffer
- \* Branch Prediction
- \* Delayed Branch

# HANDLING OF BRANCH INSTRUCTIONS

## *Prefetch the target instruction*

- Instruction in addition to the instruction following the branch. Both are saved until the branch is executed.
- The control chooses the instruction stream of the correct program flow and discard the other one.

## *Branch Target Buffer or BTB*

- The BTB is an associative memory included in the fetch segment of the pipeline.
- Entry: Store the address of a previously executed branch instruction, the target instruction for that branch, and the next few instructions after the branch target instruction.
- When fetching and decoding instruction, it searches the BTB for the address of the instruction:
  1. If it is in the BTB, fetch it from the BTB.
  2. If the instruction is not in the BTB, fetch it from memory and update the BTB.
- The advantage of this scheme is that branch instructions that have occurred previously are readily available in the pipeline without interruption.

# HANDLING OF BRANCH INSTRUCTIONS

## *The loop buffer*

- This is a small very high speed register file maintained by the instruction fetch segment of the pipeline.
- The program loop can be executed directly without having to access memory until the loop mode is removed by the final branching out.

## *Branch prediction*

- A pipeline with branch prediction uses some additional logic to guess the outcome of a conditional branch instruction before it is executed.
- The pipeline then begins prefetching the instruction stream from the predicted path.
- A correct prediction eliminates the wasted time caused by branch penalties.

## *Delayed branch*

- The compiler detects the branch instructions and rearranges the machine language code sequence by inserting useful instructions that keep the pipeline operating without interruptions.
- An example of delayed branch is the insertion of a no-operation instruction after a branch instruction.



# EXAMPLE OF DELAYED BRANCH

Load from memory to R1

Increment R2

Add R3 to R4

Subtract R5 from R6

Branch to address X



# EXAMPLE OF DELAYED BRANCH

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. No-operation						I	A	E		
7. No-operation							I	A	E	
8. Instruction in X								I	A	E

(a) Using no-operation instructions

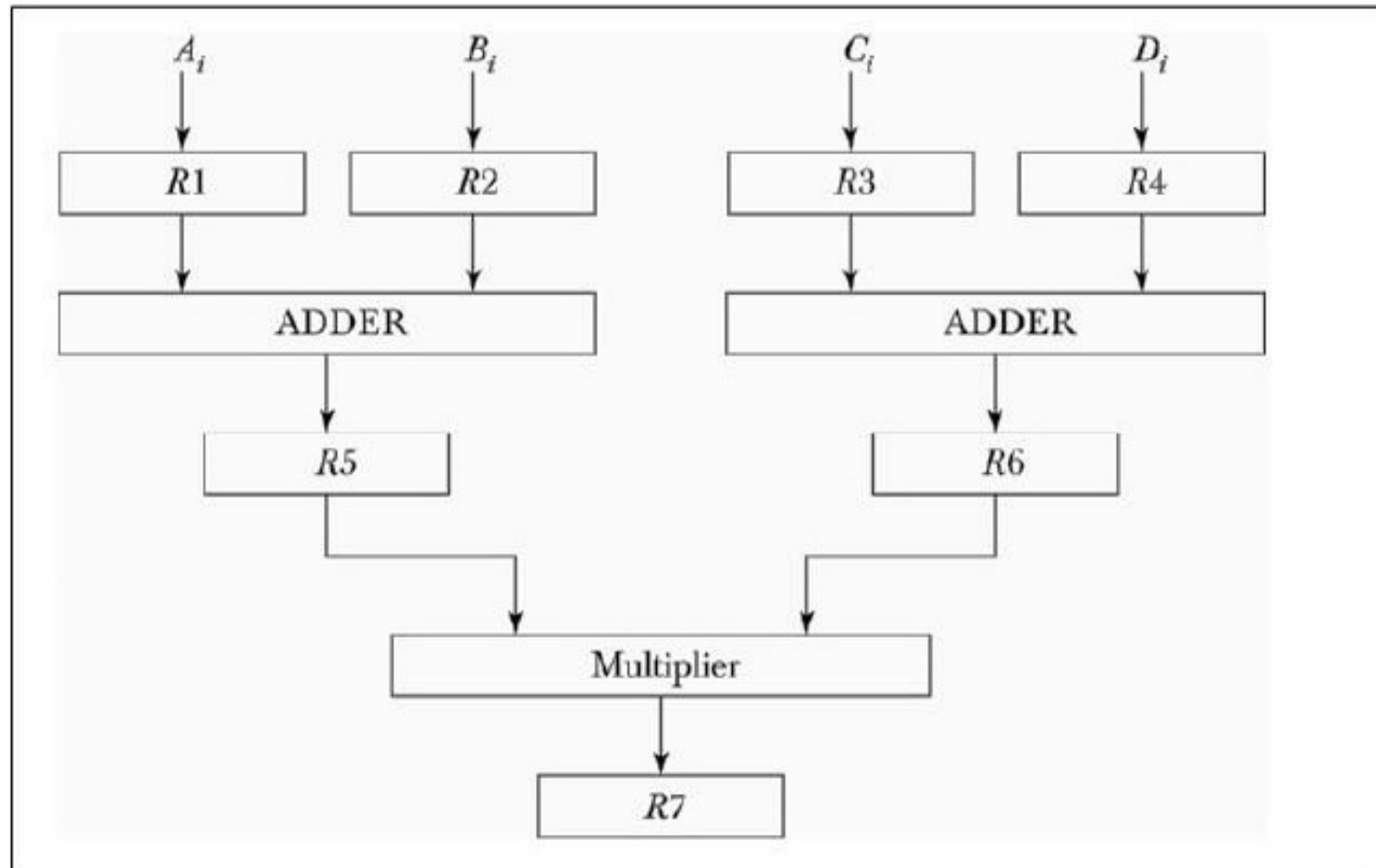
# EXAMPLE OF DELAYED BRANCH

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. Instruction in X						I	A	E

(b) Rearranging the instructions

# PROBLEMS


- 9-1.** 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. List the contents of all registers in the pipeline for  $i = 1$  through 6.



# PROBLEMS

- 9-2. Draw a space-time diagram for a six-segment pipeline showing the time it takes to process eight tasks.

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						$T_1$	$T_2$	$T_3$	$T_4$	$T_5$	$T_6$	$T_7$	$T_8$

$(k + n - 1)t_p = 6 + 8 - 1 = 13$  cycles
 

# PROBLEMS

- 9-3.** Determine the number of clock cycles that it takes to process 200 tasks in a six-segment pipeline.

$k = 6$  segments

$n = 200$  tasks  $(k + n - 1) = 6 + 200 - 1 = 205$  cycles

# PROBLEMS

- 9-4.** A nonpipeline 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?

$$t_n = 50 \text{ ns}$$

$$k = 6$$

$$t_p = 10 \text{ ns}$$

$$n = 100$$

$$S = \frac{nt_n}{(k+n-1)t_p} = \frac{100 \times 50}{(6+99) \times 10} = 4.76$$

$$S_{\max} = \frac{t_n}{t_p} = \frac{50}{10} = 5$$



# PROBLEMS

- 9-5.** The pipeline of Fig. 9-2 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 transfer into R3, and 15 ns to add the two numbers into R5.
- What is the minimum clock cycle time that can be used?
  - A nonpipeline 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?
  - Calculate the speedup of the pipeline for 10 tasks and again for 100 tasks.
  - What is the maximum speedup that can be achieved?

$$(a) t_p = 45 + 5 = 50 \text{ ns} \quad k = 3$$

$$(b) t_n = 40 + 45 + 15 = 100 \text{ ns}$$

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

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

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

# PROBLEMS

- 9-6.** It is necessary to design a pipeline for a fixed-point multiplier that multiplies two 8-bit binary integers. Each segment consists of a number of AND gates and a binary adder similar to an array multiplier as shown in Fig. 10-10.
- How many AND gates are there in each segment, and what size of adder is needed?
  - How many segments are there in the pipeline?
  - If the propagation delay in each segment is 30 ns, what is the average time that it takes to multiply two fixed-point numbers in the pipeline?

(b)	There are 7 segments in the pipeline
(c)	Average time = $\frac{k + n - 1}{n} t_p = \frac{(n + 6) 30}{n}$
	For $n = 10$ $t_{AV} = 48$ ns
	For $n = 100$ $t_{AV} = 31.8$ ns
	For $n \rightarrow \infty$ $t_{AV} = 30$ ns

# PROBLEMS

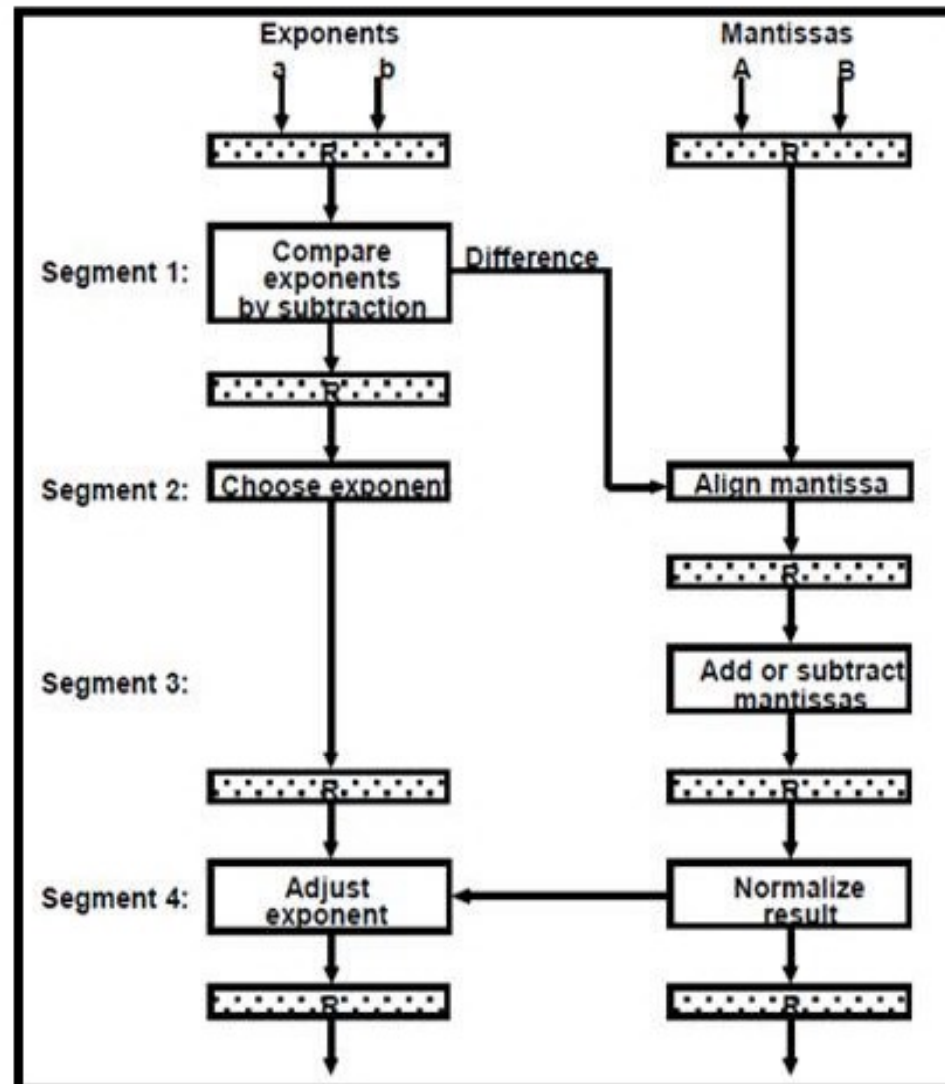
- 9-7.** The time delay of the four segments in the pipeline of Fig. 9-6 are as follows:  $t_1 = 50$  ns,  $t_2 = 30$  ns,  $t_3 = 95$  ns, and  $t_4 = 45$  ns. The interface registers delay time  $t_r = 5$  ns.
- How long would it take to add 100 pairs of numbers in the pipeline?
  - How can we reduce the total time to about one-half of the time calculated in part (a)?

- (a) Clock cycle =  $95 + 5 = 100$  ns (time for segment 3)  
For  $n = 100$ ,  $k = 4$ ,  $t_p = 100$  ns.  
Time to add 100 numbers =  $(k + n - 1) t_p = (4 + 99) 100$   
 $= 10,300$  ns =  $10.3 \mu\text{s}$
- (b) Divide segment 3 into two segments of  $50 + 5 = 55$   
and  $45 + 5 = 50$  ns. This makes  $t_p = 55$  ns;  $k = 5$   
 $(k + n - 1) t_p = (5 + 99) 55 = 5,720$  ns =  $5.72 \mu\text{s}$



# PROBLEMS

- 9-8.** How would you use the floating-point pipeline adder of Fig. 9-6 to add 100 floating-point numbers  $X_1 + X_2 + X_3 + \dots + X_{100}$ ?



# PROBLEMS

**9-9.** Formulate a six-segment instruction pipeline for a computer. Specify the operations to be performed in each segment.

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.

**9-10.** Explain four possible hardware schemes that can be used in an instruction pipeline in order to minimize the performance degradation caused by instruction branching.



# PROBLEMS

**9-11.** Consider the four instructions in the following program. Suppose that the first instruction starts from step 1 in the pipeline used in Fig. 9-8. Specify what operations are performed in the four segments during step 4.

```

Load      R1 ← M[312]
ADD       R2 ← R2 + M[313]
INC       R3 ← R3 + 1
STORE     M[314] ← R3
    
```

	1	2	3	4 <sup>th</sup> step
1. Load R1 ← M [312]	FI	DA	FO	EX
2. Add R2 ← R2 + M [313]	FI	FI	DA	FO
3. Increment R3			FI	DA
4. Store M[314] ← R3				FI

Segment EX: transfer memory word to R1.

Segment FO: Read M[313].

Segment DA: Decode (increment) instruction.

Segment FI: Fetch (the store) instruction from memory.

# PROBLEMS

**9-12.** Give an example of a program that will cause data conflict in the three-segment pipeline of Sec. 9-5.

Load:  $R1 \leftarrow \text{Memory}$   
Increment:  $R1 \leftarrow R1 + 1$

1	2	3	4
I	A	E	
	I	A	E

R1 is loaded in E  
It's too early to increment it in A

**9-13.** Give an example that uses delayed load with the three-segment pipeline of Sec. 9-5.

Insert a No-op instruction between the two instructions in the example of Problem 9-12 (above).

# PROBLEMS

**9-14.** Give an example of a program that will cause a branch penalty in the three-segment pipeline of Sec. 9-5.

		1	2	3	4	5	6	7
101	Add R2 to R3	I	A	E				
102	Branch to 104	I	A	E				
103	Increment R1		-	-				
104	Store R1					I	A	E

**9-15.** Give an example that uses delayed branch with the three-segment pipeline of Sec. 9-5.

Use example of Problem 9-14.		1	2	3	4	5	6
101	Branch to 105	I	A	E	↓		
102	Add R2 to R3		I	A	E		
103	No-operation			I	A	E	
104	Increment R1				↓		
105	Store R1				I	A	E