Unit-4
Pipelinie and Vector Processing

Contents

- Parallel Processing
- Pipelining
- Arithmetic Pipeline,
- Instruction Pipeline
- RISC Pipeline
- Vector Processing
- Array Processors

Parallel Processing

Parallel Processing

- used to provide *Simultaneous* data processing tasks for the purpose of increasing the computational speed of a computer system.
- Perform *concurrent* data processing to achieve faster execution time.
- For example:

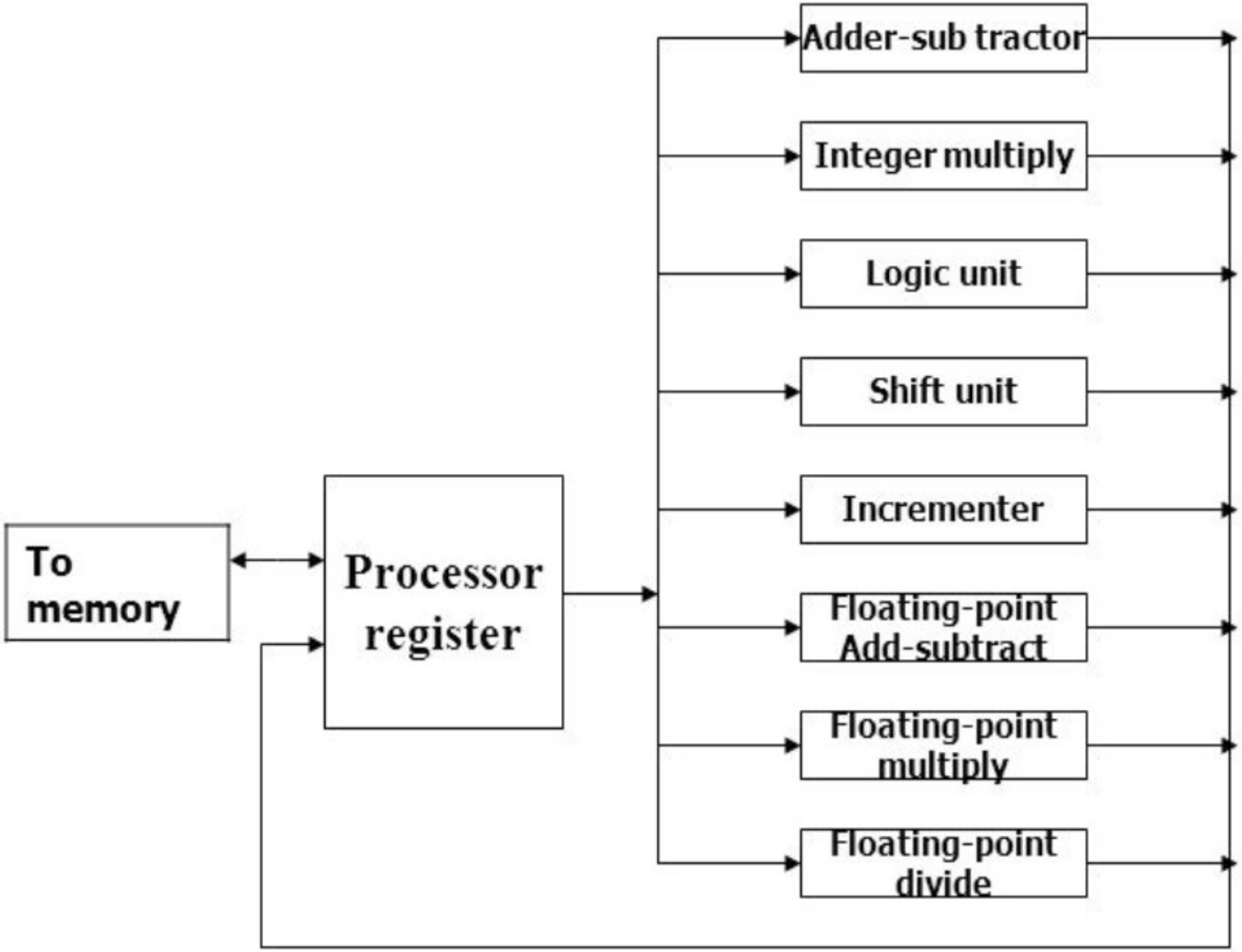
  » while an instruction is being executed in the ALU, the next instruction can be read from memory.

  » The system may have two or more ALUs and be able to execute two or more instructions at the same time .

  » The system may have two or more processors operating concurrently.
- The purpose of parallel processing is to speed up the computer procesing capability and increase its **throughput.**
- **Throughput:** the amount of processing that can be accomplished during a given interval of time.

Parallel processing is established by distributing the data among the multiple functional units.

Processor with Multiple Functional Unit : Fig. 9.1



Processor with Multiple Functional Unit:

- Separates the execution unit into eight functional units operating in parallel.
- The adder and integer multiplier perform the arithmetic operations with integer numbers.
- The floating-point operations are separated into three circuits operating in parallel.
- The logic, shift, and increment operations can be performed concurrently on different data.
- All units are independent of each other, so one number can be shifted while another number is being incremented.
- A multifunctional organization is usually associated with a complex control unit to coordinate all the activities among the various components.

Parallel processing can be classified as:

- The internal organization of the processors
- The interconnection structure between processors
- The flow of information through the system
- The number of instructions and data items that are manipulated simultaneously introduced by M. J. Flynn.

Parallel processing may occur in the instruction stream, the data stream, or both.

Instruction stream: The sequence of instructions read from memory constitutes an instruction stream.

Data stream: The operations performed on the data in the processor constitutes a data stream.

Flynn's classification divides computers into four major groups as follows:

- Single instruction stream, single data stream – **SISD**
- Single instruction stream, multiple data stream – **SIMD**
- Multiple instruction stream, single data stream – **MISD**
- Multiple instruction stream, multiple data stream – **MIMD**

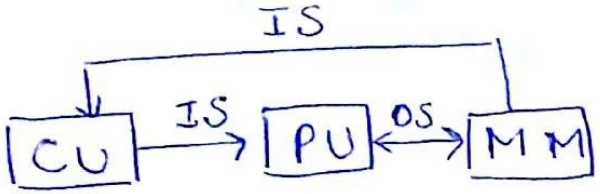
SISD –

- Represents the organization of a single computer containing a control unit, a processor unit, and a memory unit.
- Instructions are executed sequentially.
- Parallel processing may be achieved by means of multiple functional units or by pipeline processing

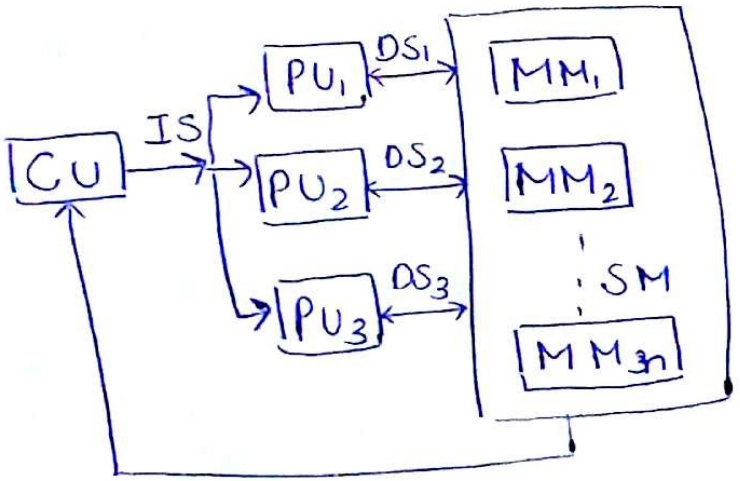
SIMD – Includes multiple processing units with a single control unit. All processors receive the same instruction from the control unit, but operate on different data.

MISD – structure is only of theoretical interest since no practical system has been constructed using this organization.

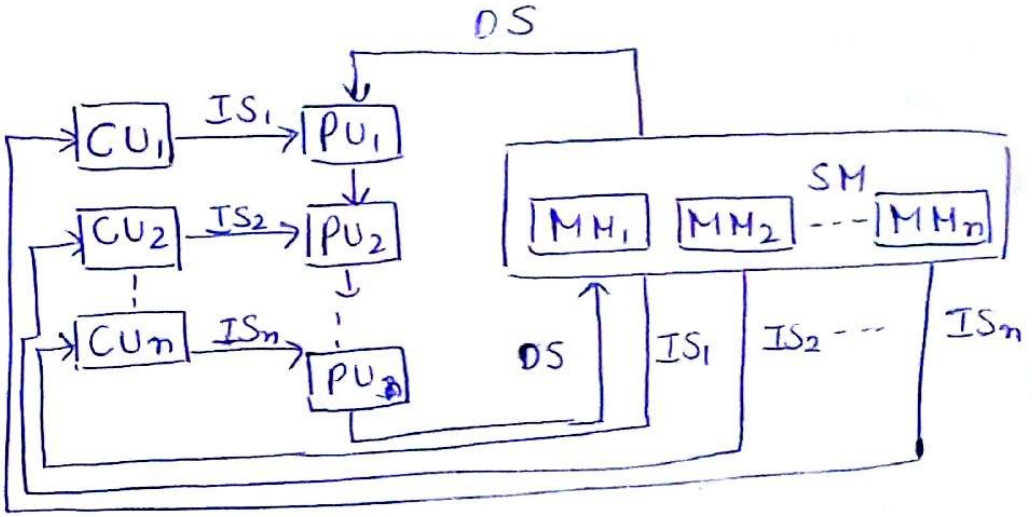
MIMD – A computer system capable of processing several programs at the same time.



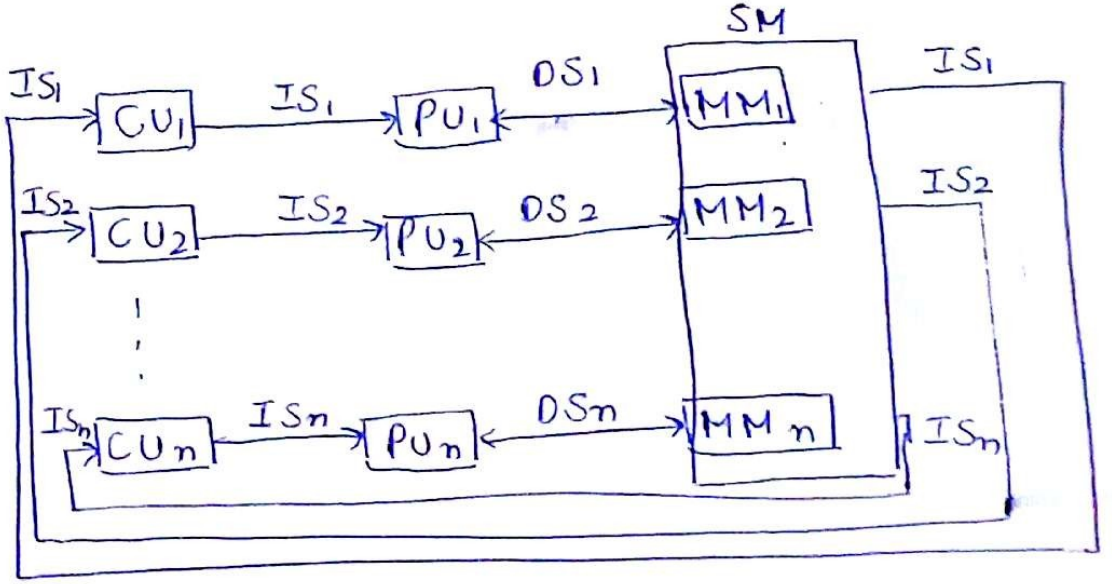
SISD Computer



SIMD Computer



MISD comp.



MIMD comp.

IS → Instruction Stream

DS → Data

MM → MEMORY MODULE

SM → SHARED MEMORY

CU → CONTROL UNIT

PU → PROCESSOR "

Applications of Parallel Processing

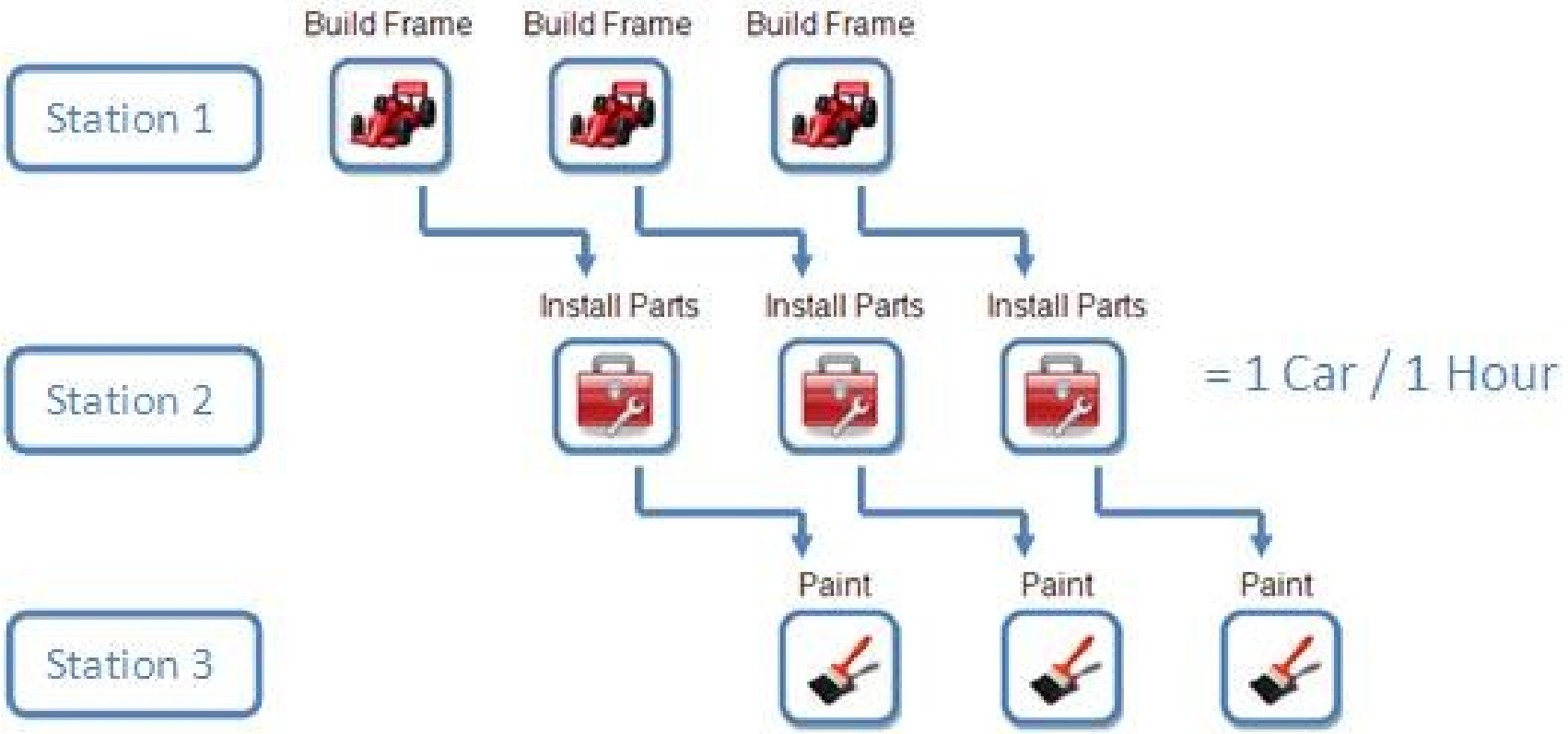
- Numeric weather prediction
- Finite element analysis
- Artificial Intelligence and Automation
- Genetic Engineering
- Weapon Research and Defense
- Medical Applications

One type of parallel processing that does not fit Flynn's classification is pipelining.

Pipelining

- Pipelining is a technique of decomposing a sequential process into suboperations, with each subprocess being executed in a special dedicated segment that operates concurrently with all other segments
- 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 final result is obtained after the data have passed through all segments
- Can imagine that each segment consists of an input register followed by an combinational circuit
- A clock is applied to all registers after enough time has elapsed to perform all segment activity
- The information flows through the pipeline one step at a time.

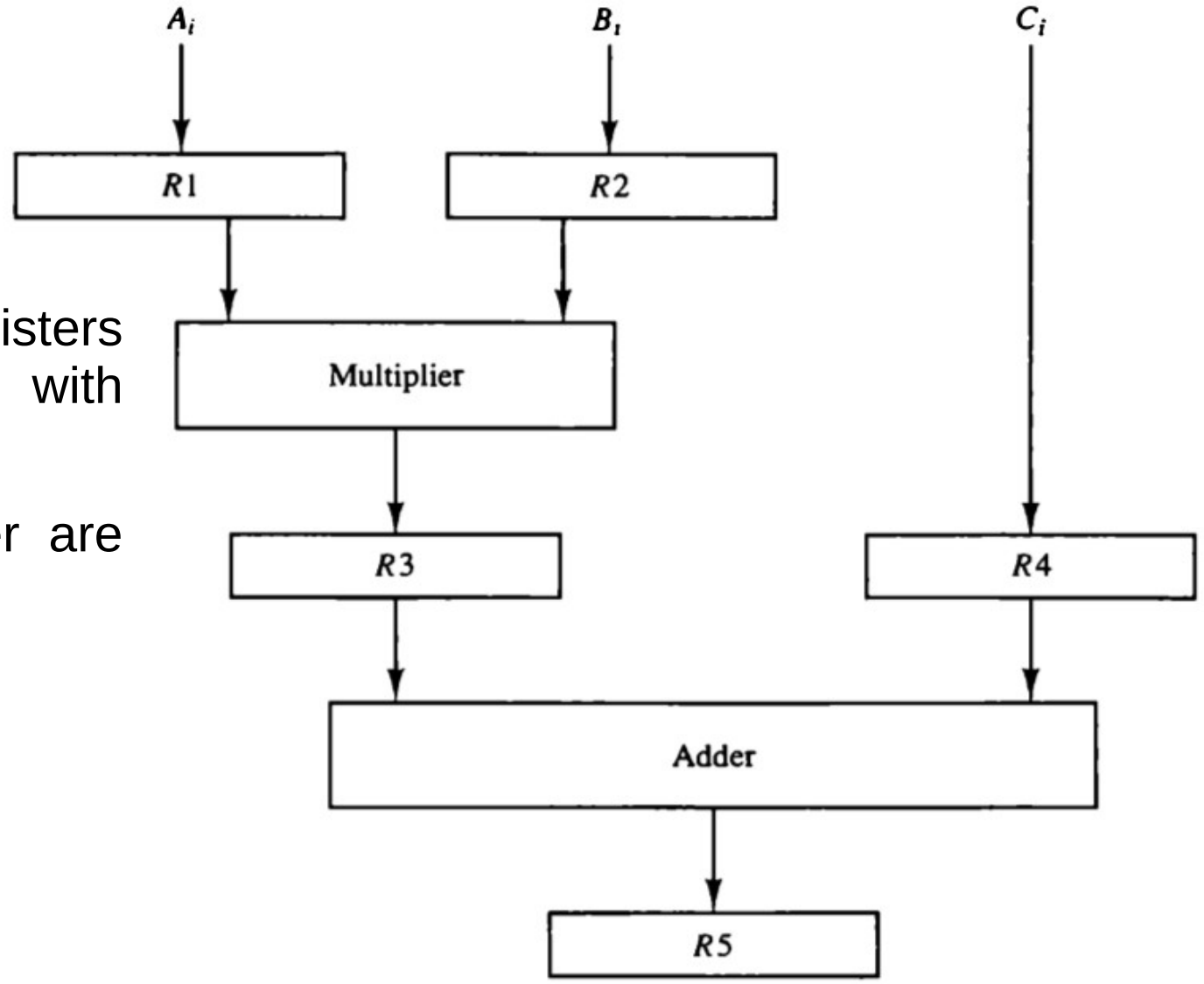
Although each car still takes three hours to finish using pipelining, we can now produce one car each hour rather than one every three hours



Example: Ai * Bi + Ci for i = 1, 2, 3, ..., 7

Each suboperation is to be implemented in a segment within a pipeline. Each segment has one or two registers and a combinational circuit as shown in Fig.9-2.

Figure 9-2 Example of pipeline processing.



R1 through R5 are registers that receive new data with every clock pulse.

The multiplier and adder are combinational circuits.

The suboperations performed in each segment within a pipeline are:

R1 ← Ai , R2 ← Bi (Input Ai & Bi)

R3 ← R1 * R2, R4 ← Ci (Multiply & input Ci)

R5 ← R3 + R4 (Add Ci to product)

The five registers are loaded with new data every clock pulse. The effect of each clock is shown in Table 9-1 .

It takes three clock pulses to fil up the pipe and retrieve the first output from R5.

From there on, each clock produces a new output and moves the data one step down the pipeline.

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	—	—	—	—	$A_7 * B_7 + C_7$

- Any operation that can be decomposed into a sequence of suboperations 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 time with different sets of data

The general structure of a four-segment pipeline is illustrated in Fig. 9-3.

- The operands pass through all four segments in a fixed sequence. Each segment consists of a combinational circuit Si that performs a suboperation over the data stream flowing through the pipe.
- The segments are separated by registers Ri that hold the intermediate results between the stages.
- Information flows between adjacent stages under the control of a common clock applied to all the registers simultaneously.
- A task is the total operation performed going through all segments of a pipeline

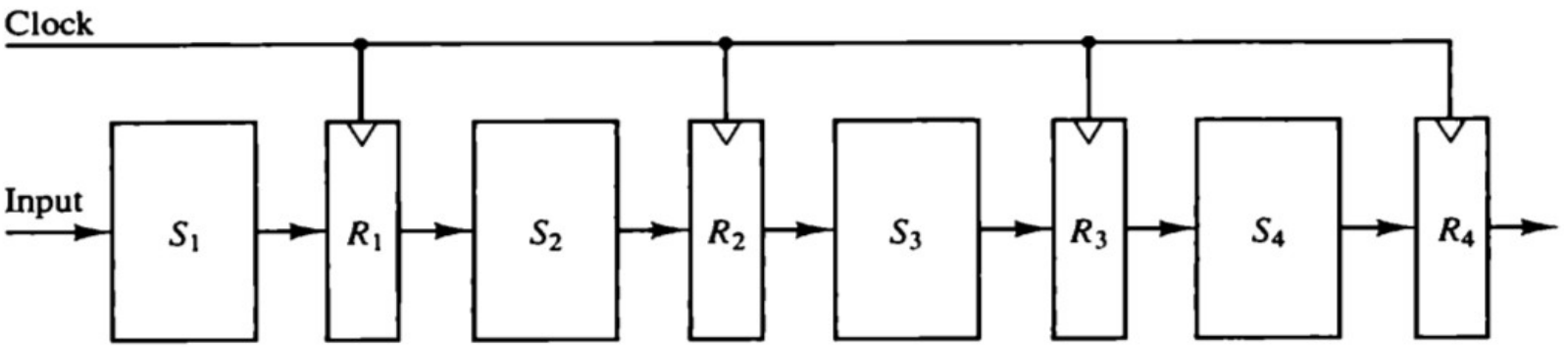
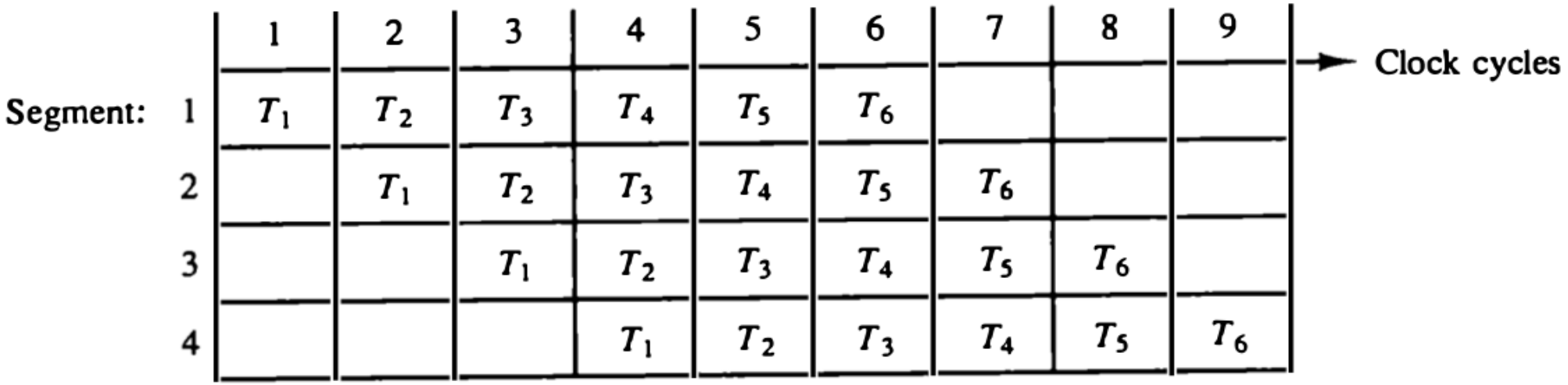


Figure 9-3 Four-segment pipeline.

The behavior of a pipeline can be illustrated with a space-time diagram:

- This shows the segment utilization as a function of time
- The horizontal axis displays the time in clock cycles and the vertical axis gives the segment number.
- The diagram shows six tasks T1 through T6 executed in four segments.
- Initially, task T1 is handled by segment 1. After the first clock, segment 2 is busy with T1, while segment 1 is busy with task T2. Continuing in this manner, the first task T1 is completed after the fourth clock cycle.
- From then on, the pipe completes a task every clock cycle.
- No matter how many segments there are in the system, Once the pipeline is full, it takes only one clock period to obtain an output

Figure 9-4 Space-time diagram for pipeline.



- **Consider a k-segment pipeline** with a clock cycle time tp to execute n tasks
- The first task T1 requires time **k tp** to complete
- The remaining n-1 tasks finish at the rate of one task per clock cycle and will be completed after time **(n-1)tp**
- The total time to complete the n tasks is **[k+n-1]tp**
- The example of Figure 9-4 requires [4+6-1] clock cycles to finish

- **Consider a nonpipeline unit** that performs the same operation and takes **tn** time to complete each task
- The total time to complete n tasks would be **n tn**

- The speedup of a pipeline processing over an equivalent nonpipeline processing is defined by the ratio

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

- As the number of tasks increase, the speedup becomes

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

- If we assume that the time to process a task is the same in both circuits, $t_n = k\, t_p$

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

- Therefore, the theoretical maximum speedup that a pipeline can provide is k (no. Of segments)

Example:

- Cycle time = tp = 20 ns
- # of segments = k = 4
- # of tasks = n = 100

The pipeline system will take (k + n – 1)tp = (4 + 100 –1)20ns = 2060 ns

Assuming that tn = k tp = 4 * 20 = 80 ns,

A nonpipeline system requires n k tp = 100 * 80 = 8000 ns

The speedup ratio = 8000/2060 = 3.88

As the number of tasks increases, the speedup will approach 4, which is equal to the number of segments in the pipeline.

If we assume that t, = 60 ns, the speedup becomes 60/20 = 3 .

The pipeline cannot operate at its maximum theoretical rate

- One reason is that, different segments may take different times to complete their suboperation, the clock cycle must be chosen to equal the time delay of the segment with the maximum propagation time. This causes all other segments to waste time while waiting for the next clock.
- Moreover, a nonpipe circuit will not always have the same time delay as that of an equivalent pipeline circuit, as many of the intermediate registers will not be needed in a single-unit circuit
- Nevertheless, the pipeline technique provides a faster operation over a purely serial sequence even though the maximum theoretical speed is never fully achieved.

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.

Example for floating-point addition and subtraction

- Inputs are two normalized floating-point binary numbers

  X = A x 2^a

  Y = B x 2^b
- A and B are two fractions that represent the mantissas
- a and b are the exponents

Four segments are used to perform the floating-point addition and subtraction:

- **Compare the exponents**
- **Align the mantissas**
- **Add or subtract the mantissas**
- **Normalize the result**

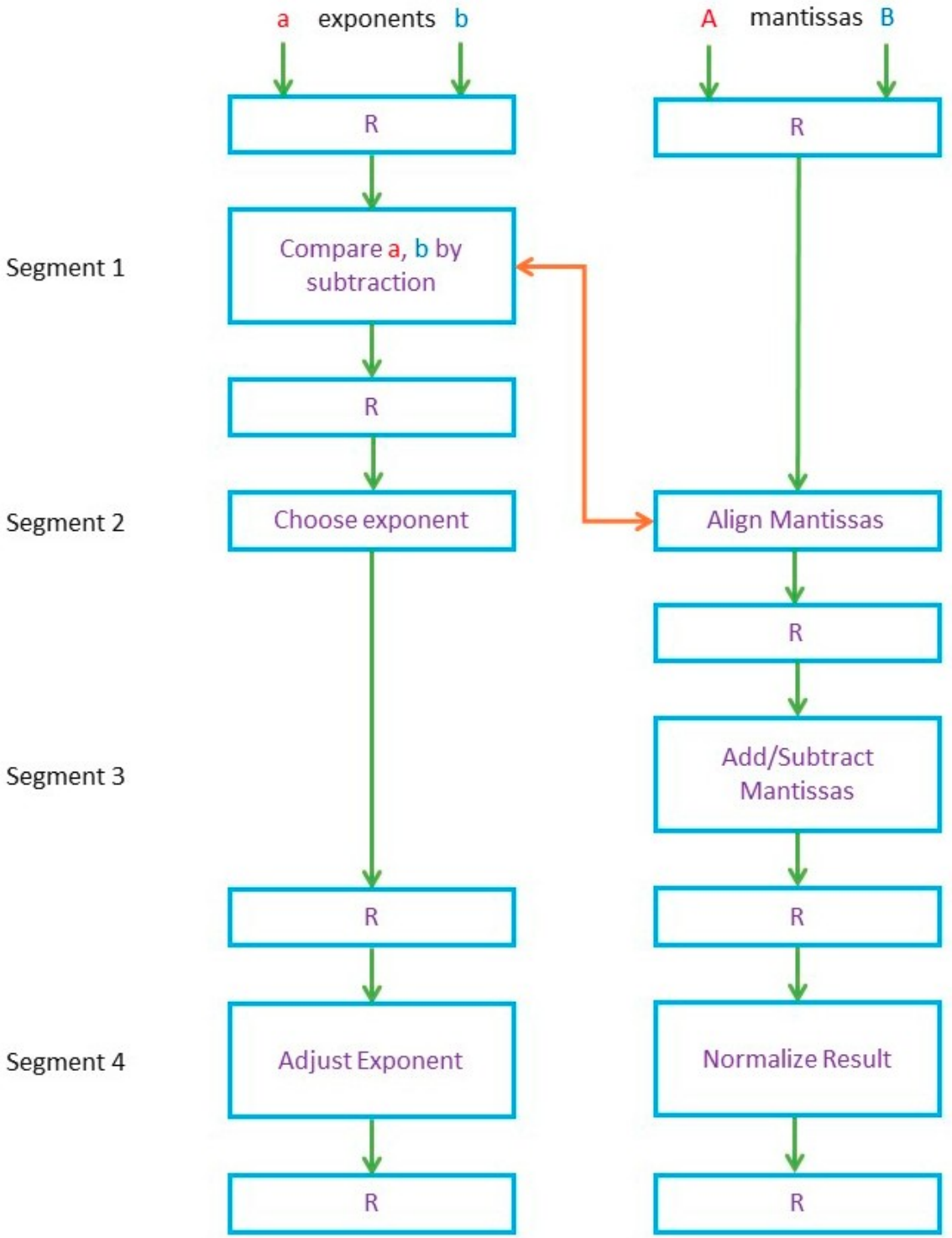


Figure | Arithmetic pipeline for floating point add/subtract operation

- Consider the two normalized floating-point numbers:

  X = 0.9504 x 10^3

  Y = 0.8200 x 10^2

- 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.
- Segment 2 shifts the mantissa of Y to the right to obtain

  X = 0.9504 x 10^3

  Y = 0.0820 x 10^3

- The mantissas are now aligned.
- Segment 3 produces the sum Z = 1.0324 x 10^3
- Segment 4 normalizes the result so that it has a fraction with a nonzero first digit, by shifting the mantissa once to the right and incrementing the exponent by one to obtain

  Z = 0.10324 x 10^4

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.
- If a branch out of sequence occurs, the pipeline must be emptied and all the instructions that have been read from memory after the branch instruction must be discarded.
- Consider a computer with an instruction fetch unit and an instruction execution unit forming a two segment pipeline.
- A FIFO buffer can be used for the fetch segment.
- Thus, an instruction stream can be placed in a queue, waiting for decoding and processing by the execution segment.
- This reduces the average access time to memory for reading instructions.
- Whenever there is space in the buffer, the control unit initiates the next instruction fetch phase.

Instruction Pipeline

The following steps are needed to process each instruction:

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

Example: Four-segment instruction pipeline

1. Fl is the segment that fetches an instruction.

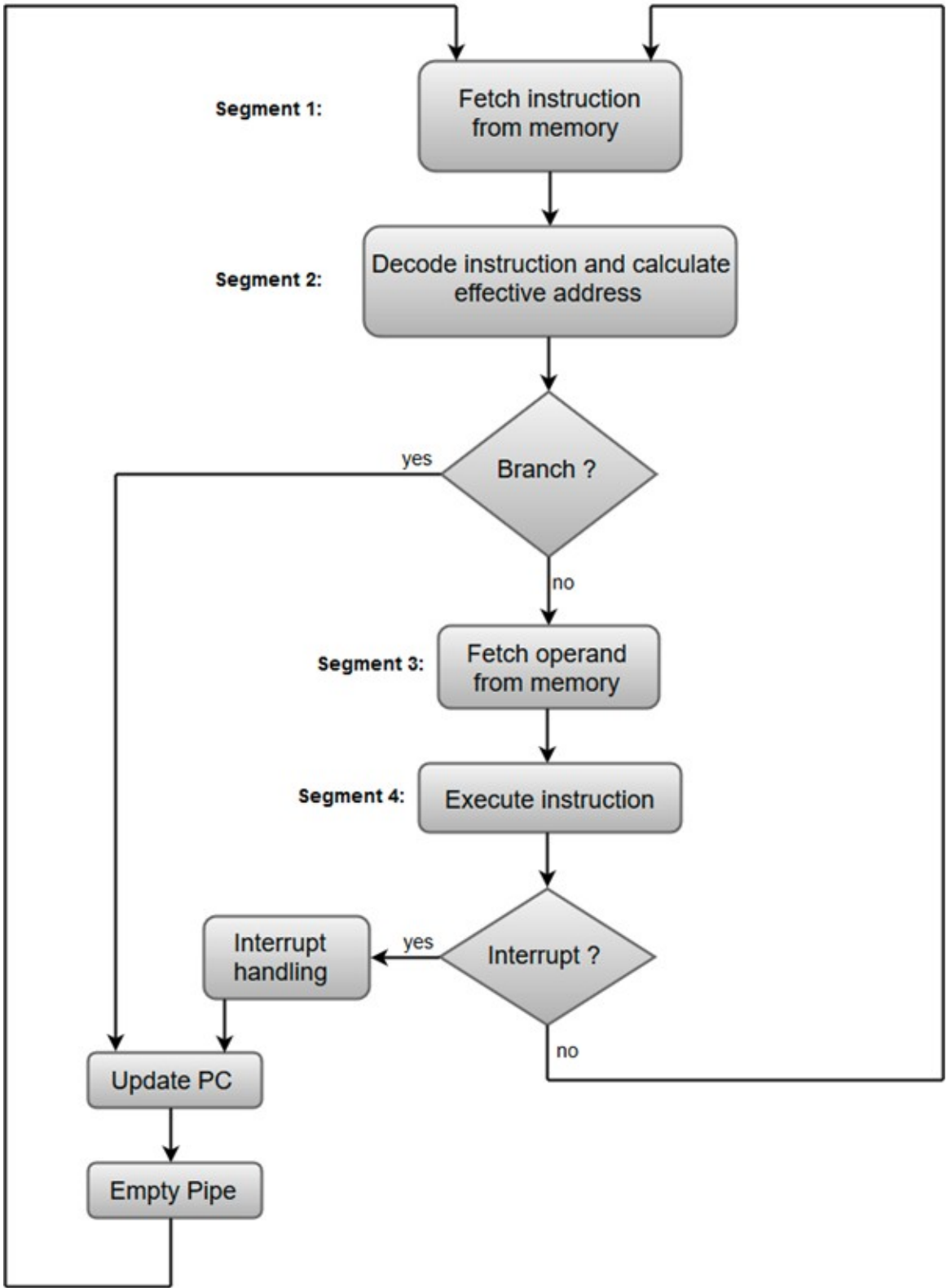
2. DA is the segment that decodes the instruction and calculates the effective address.

3. FO is the segment that fetches the operand.

4. EX is the segment that executes the instruction

(Assume that most of the instructions store the result in a register so that the **execution and storing of the result can be combined in one segment.)**

Instruction cycle in the CPU processed with a four-segment pipeline.



- Figure shows the operation of the instruction pipetine. The time in the horizontal axis is divided into steps of equal duration.
- Up to four suboperations in the instruction cycle can overlap and up to four different instructions can be in progress of being processed at the same time
- **It is assumed that the processor has separate instruction and data memories so that the operation in Fl and FO can proceed at the same time.**
- Assume that instruction 3 is a branch instruction. As soon as this instruction is decoded in segment DA in step 4, the transfer from FI to DA of the other instructions is halted until the branch instruction is executed in step 6.

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

Instruction Pipeline

There are three major difficulties that cause the instruction pipeline to deviate from its normal operation :

- **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.
- **Data dependency conflicts** arise when an instruction depends on the result of a previous instruction, but this result is not yet available.
- **Branch difficulties** arise from branch and other instructions that change the value of PC .

RICS Pipeline

The simplicity of the RICS instruction set can be utilized to implement an instruction pipeline using a small number of suboperations, with each being executed in one clock cycle.

The instruction cycle can be divided into three suboperations and implemented in three segments:

I: Instruction fetch :The I segment fetches the instruction from program memory.

A: ALU operation : The instruction is decoded and an ALU operation is performed in the A segment. The ALU is used for three different functions, depending on the decoded instruction. It performs an operation for a data manipulation instruction, it evaluates the effective address for a load or store instruction, or it calculates the branch address for a program control instruction.

E: Execute instruction : The E segment directs the output of the ALU to one of three destinations, depending on the decoded instruction. It transfers the result of the ALU operation into a destination register in the register file, it transfers the effective address to a data memory for loading or storing, or it transfers the branch address to the program counter.

Consider now the operation of the following four instructions:

1. LOAD: Rl <-- M [address 1]
2. LOAD: R2 <-- M [address 2]
3. ADD: R3 <-- R l + R2
4. STORE: M[address 3] <-- R3

The E segment in clock cycle 4 is in a process of placing the memory data into R2. The A segment in clock cycle 4 is using the data from R2, but the value in R2 will not be the correct value since it has not yet been transferred from memory.

It is up to the compiler to make sure that the instruction following the load instruction uses the data fetched from memory. If the compiler cannot find a useful instruction to put after the load, it inserts a no-op (no-operation) instruction. This is a type of instruction that is fetched from memory but has no operation, thus wasting a clock cycle. This concept of delaying the use of the data loaded from memory is referred to as ***delayed load.***

Clock cycles:	1	2	3	4	5	6
1. Load *R*1	I	A	E			
2. Load *R*2		I	A	E		
3. Add *R*1 + *R*2			I	A	E	
4. Store *R*3				I	A	E

(a) Pipeline timing with data conflict

Clock cycle:	1	2	3	4	5	6	7
1. Load *R*1	I	A	E				
2. Load *R*2		I	A	E			
3. No-operation			I	A	E		
4. Add *R*1 + *R*2				I	A	E	
5. Store *R*3					I	A	E

(b) Pipeline timing with delayed load

Practice Questions

Q1: Draw the space-time diagram for a 4-segment pipeline executing six tasks.

Q2: Specify the pipeline configuration to carry out the computation :

$(A_i+Bi)(Ci+Di)$. Also list the contents of registers in the pipeline for i=1 through 6.

Q3: Perform the addition of the following floating point numbers using arithmetic pipeline.

$X=0.9504*10^3$

$Y=0.8200*10^2$

Also draw the diagram representing pipeline for the floating point addition and subtraction.

Practice Questions

Q4: Determine the number of clock cycles it takes to process 200 tasks in a 6 segment pipeline.

Solution : k = 6 segments, n = 200 tasks

(k + n – 1) = 6 + 200 – 1 = 205 cycles

Q5: A nonpipellne system takes 50 ns to process a task. The same task can 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 achieved?

Solution: tn = 50 ns, k = 6, tp = 10 ns, n = 100

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

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

Practice Questions

Q5: The lime delay of the four segments in the pipeline are as follows: t1=50ns, t2=30ns, t3=95ns, and t4= 45 ns. The interface registers delay time1= 5 ns.

a. How long would it take to add 100 pairs of numbers in the pipeline?

b. How can we reduce the total time to about one-half of the time calculated in part (a)?

Solution:

(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 µs

(b) Divide segment 3 into two segments of 50 + 5 = 55 and 45 + 5 = 50 ns. This makes tp = 55 ns; k = 5
(k + n – 1) tp = (5 + 99) 55 = 5,720 ns = 5.72 µs

Practice Questions

Q6: Consider the four instructions in the following program. Suppose that the first instruction starts from step 1 in the pipeline used in. Specify what operations are performed in the four segments during step 4.

Solution:

	1	2	3	4^{th} 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.