Pipeline and vector Processing

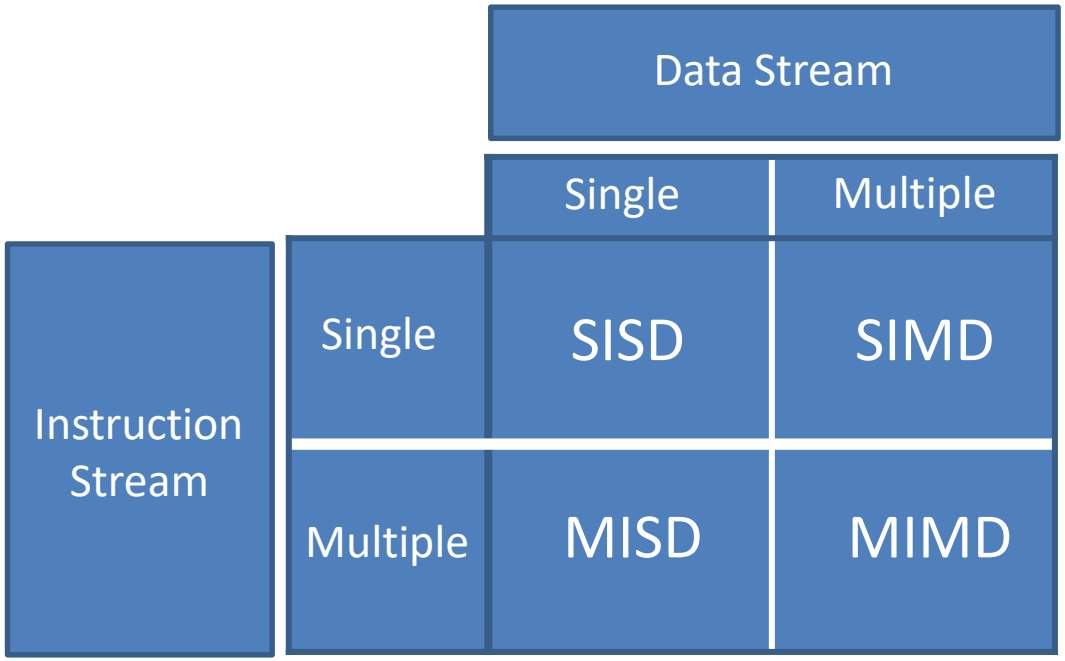
Prof. Usman Sindhi
GUNI-ICT

Topics to be covered

- ☐ Flynn's taxonomy
- ☐ Parallel Processing
- ☐ Pipelining
- ☐ Arithmetic Pipeline
- ☐ Instruction Pipeline

Flynn's Taxonomy

Flynn's Taxonomy



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

- 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.

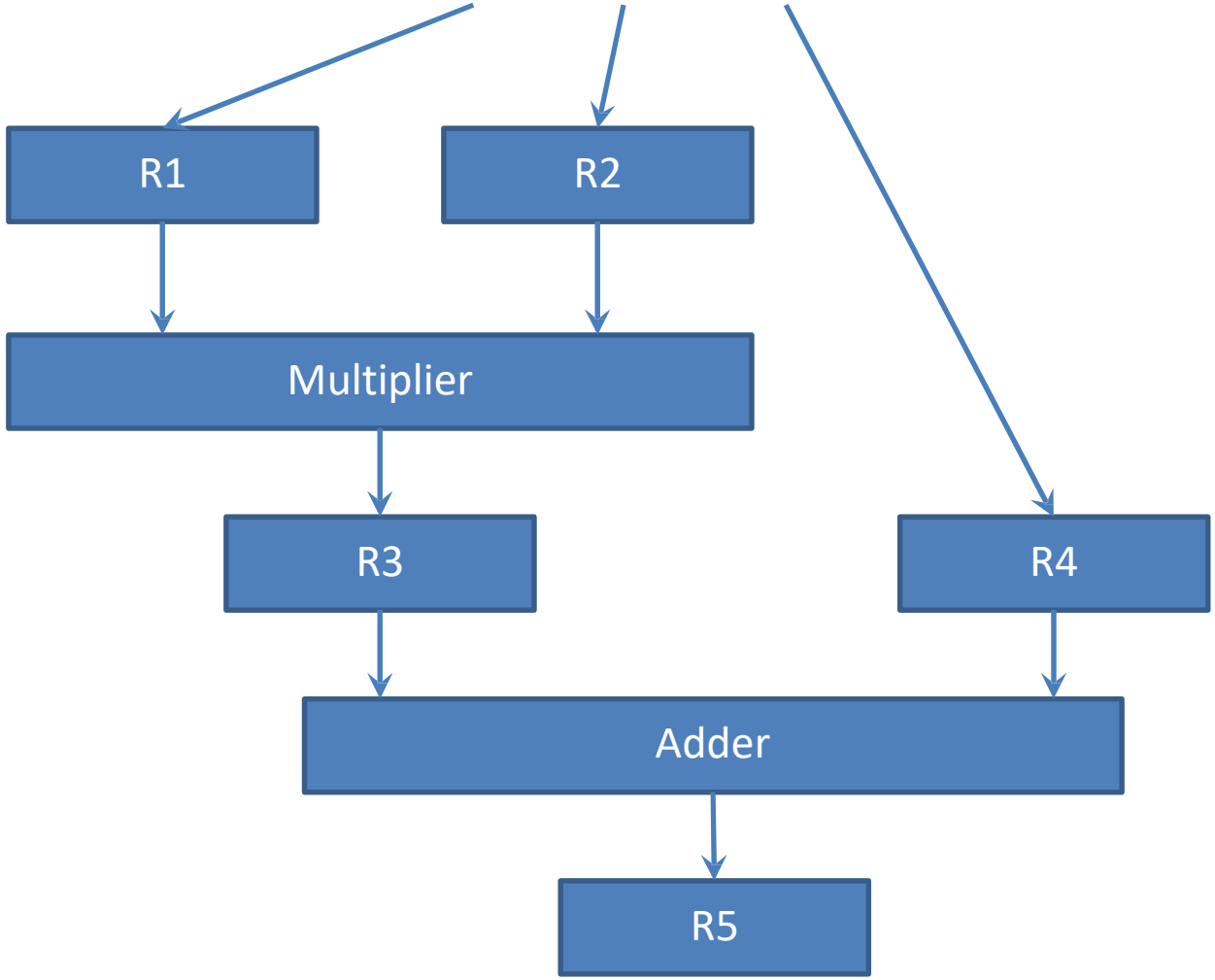
Pipelining

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 $i = 1,2,3, \ldots, 7$

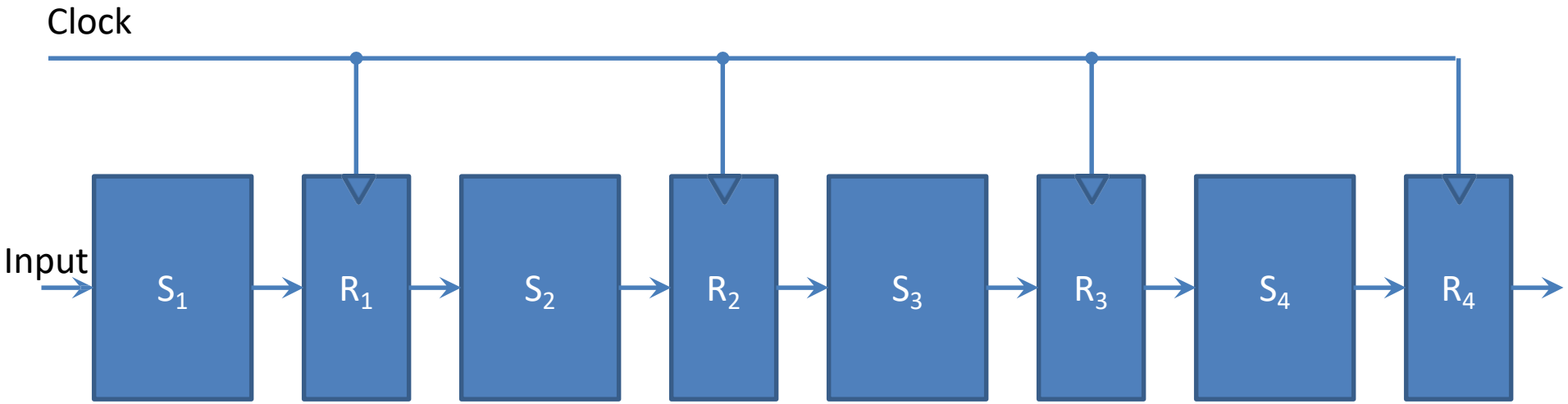


Content of register in pipeline example

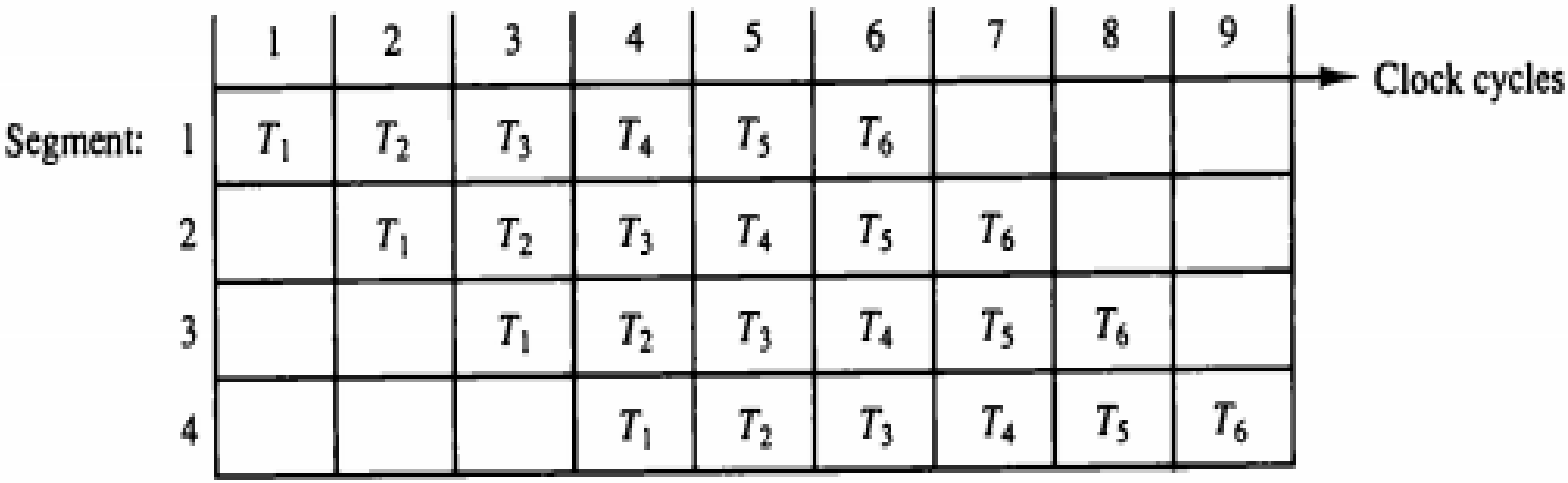
Clock Pulse Number	Segment 1		Segment 2		Segment 3
	$R1$	$R2$	$R3$	$R4$	$R5$
1	A_1	B_1	—	—	—
2	A_2	B_2	$A_1 * B_1$	C_1	—
3	A_3	B_3	$A_2 * B_2$	C_2	$A_1 * B_1 + C_1$
4	A_4	B_4	$A_3 * B_3$	C_3	$A_2 * B_2 + C_2$
5	A_5	B_5	$A_4 * B_4$	C_4	$A_3 * B_3 + C_3$
6	A_6	B_6	$A_5 * B_5$	C_5	$A_4 * B_4 + C_4$
7	A_7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

Pipelining

☐ General structure of four segment pipeline



Space-time Diagram



Speedup

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

$$S = \frac{nt_{\mathrm{n}}}{(k+n-1)t_{\mathrm{p}}}$$

- 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

$$S = \frac{nt_{\mathrm{n}}}{nt_{\mathrm{p}}} = \frac{t_{\mathrm{n}}}{t_{\mathrm{p}}}$$

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

$$S = \frac{kt_{\mathrm{p}}}{t_{\mathrm{p}}} = k$$

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

Arithmetic Pipeline

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:

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

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

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

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

  $Z = 1.0324 \times 10^3$

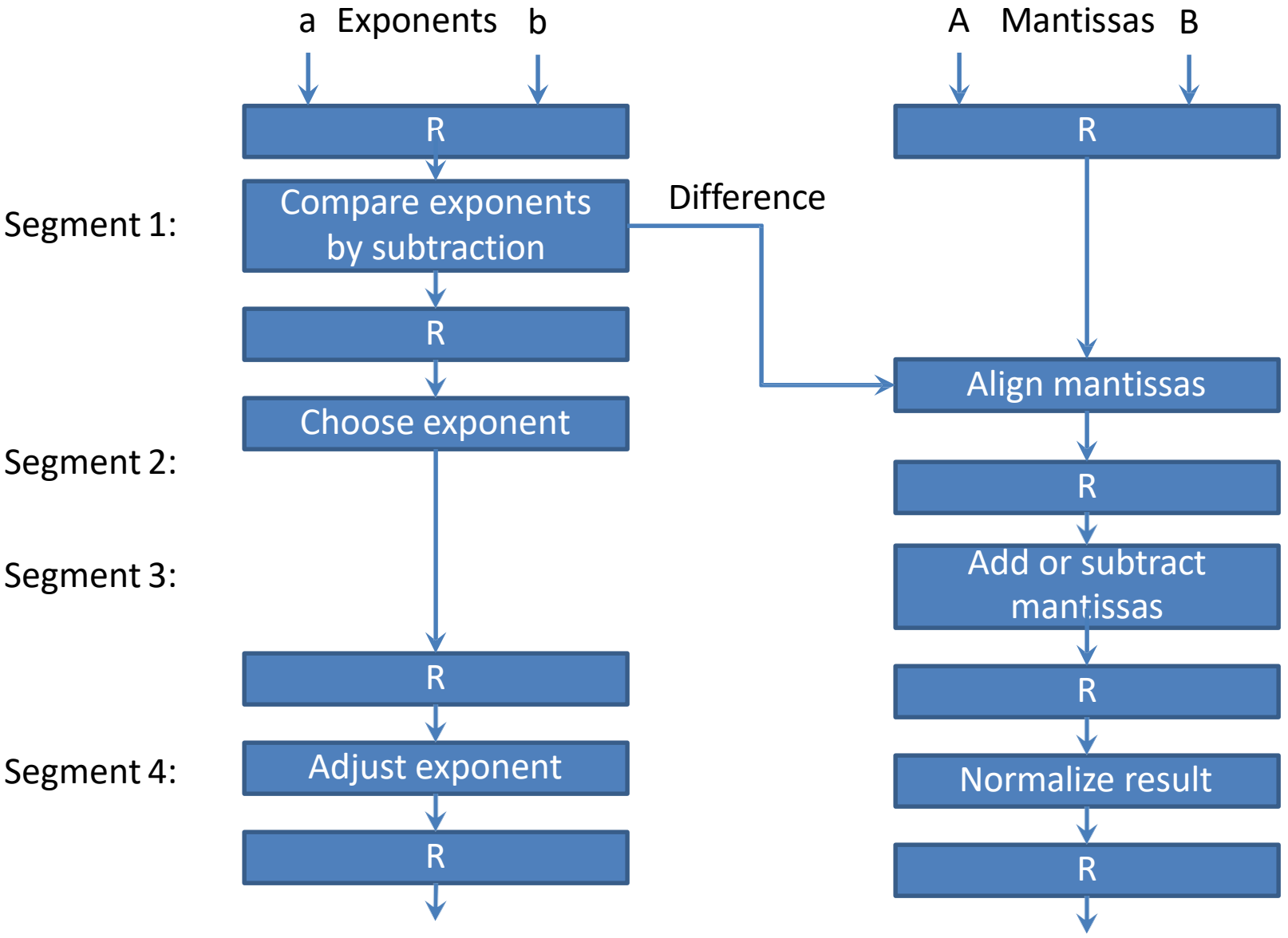
- Segment-4: Normalize the result

  $Z = 0.10324 \times 10^4$

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

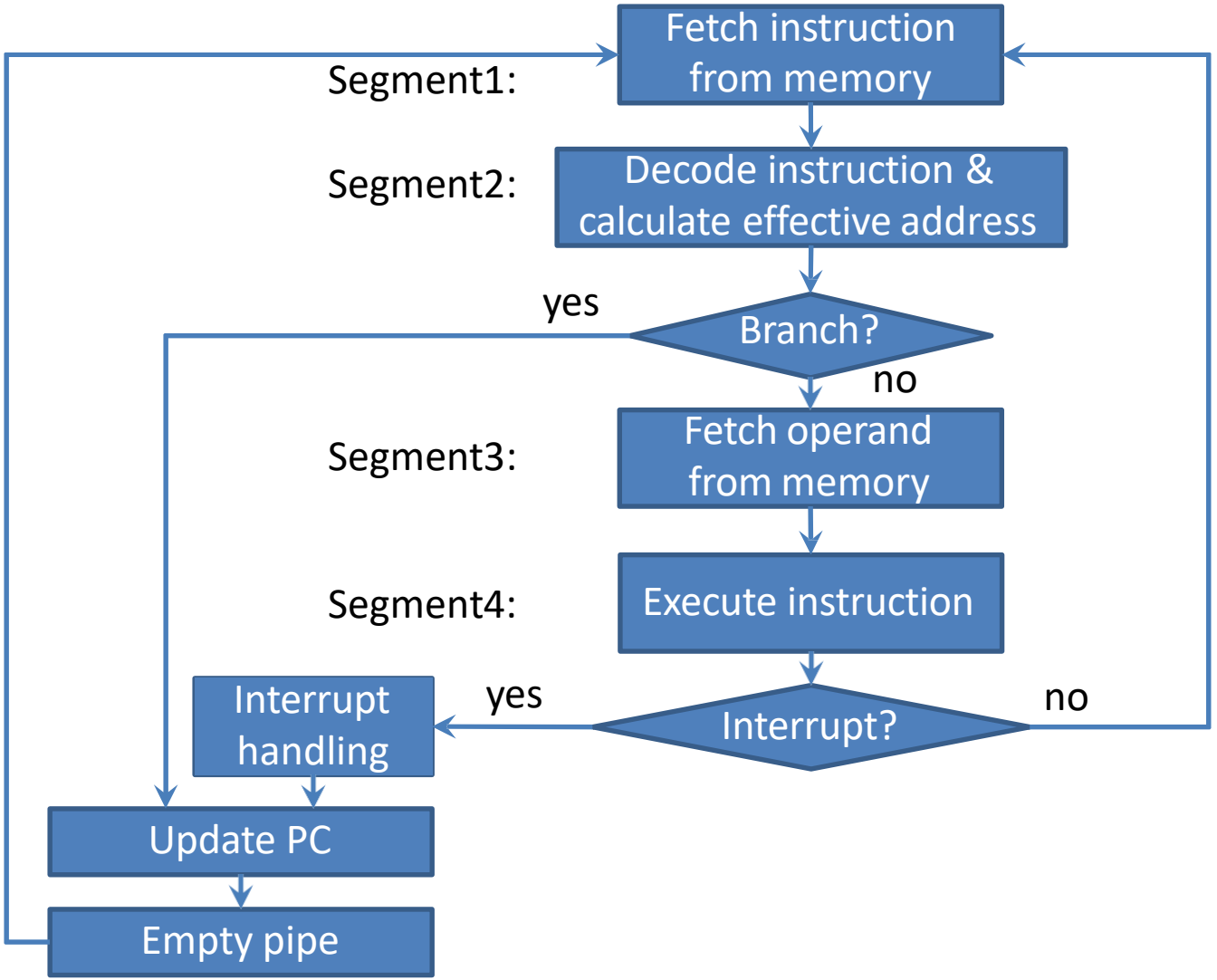
Instruction Pipeline

- 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

Four segment CPU pipeline



Space-time Diagram

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

Data Dependency

- The most straightforward method is to insert *hardware interlocks.*
- An interlock is a circuit that detects instructions whose source operands are destinations of instructions farther up in the pipeline.
- Another technique called *operand forwarding* uses special hardware to detect a conflict and then avoid it by routing the data through special paths between pipeline segments.
- The compiler for such computers is designed to detect a data conflict and reorder the instructions as necessary to delay the loading of the conflicting data by inserting no-operation instructions. This method is referred to as *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 target
  - Branch target buffer
  - Loop buffer
  - Delayed branch

Handling Branch Instructions

- **Prefetch Target Instruction**

- Fetch instructions in streams, branch not taken and branch taken.
- Both are saved until branch is executed. Then, select the right instruction stream and discard the wrong stream

- **Branch Target Buffer(BTB; Associative Memory)**

- Entry: Address of previously executed branches;

  Target instruction and the next few instructions

- When fetching an instruction, search BTB.
- If found, fetch the instruction stream in BTB;
- If not, new stream is fetched and update BTB

Handling Branch Instructions

- **Loop Buffer(High Speed Register file)**

- Storage of entire loop that allows executing a loop without accessing memory.

- **Delayed Branch**

- A procedure employed in most RISC processors is the *delayed branch.*

- Compiler detects the branch and rearranges the instruction sequence by inserting useful instructions that keep the pipeline busy in the presence of a branch instruction.