PIPELINE CONTROL AND PARALLEL PROCESSING

Content

Introduction to 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.

Instead of processing each instruction sequentially as in a conventional computer, a parallel processing system is able to perform concurrent data processing to achieve faster execution time.

Continue..

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.

Furthermore, the system may have two or more processors operating concurrently.

Continue...

The purpose of parallel processing is to speed up the computer processing capability and increase its throughput, that is, the amount of processing that can be accomplished during a given interval of time.

The amount of hardware increases with parallel processing. and with it, the cost of the system increases.

However, technological developments have reduced hardware costs to the point where parallel processing techniques are economically feasible.

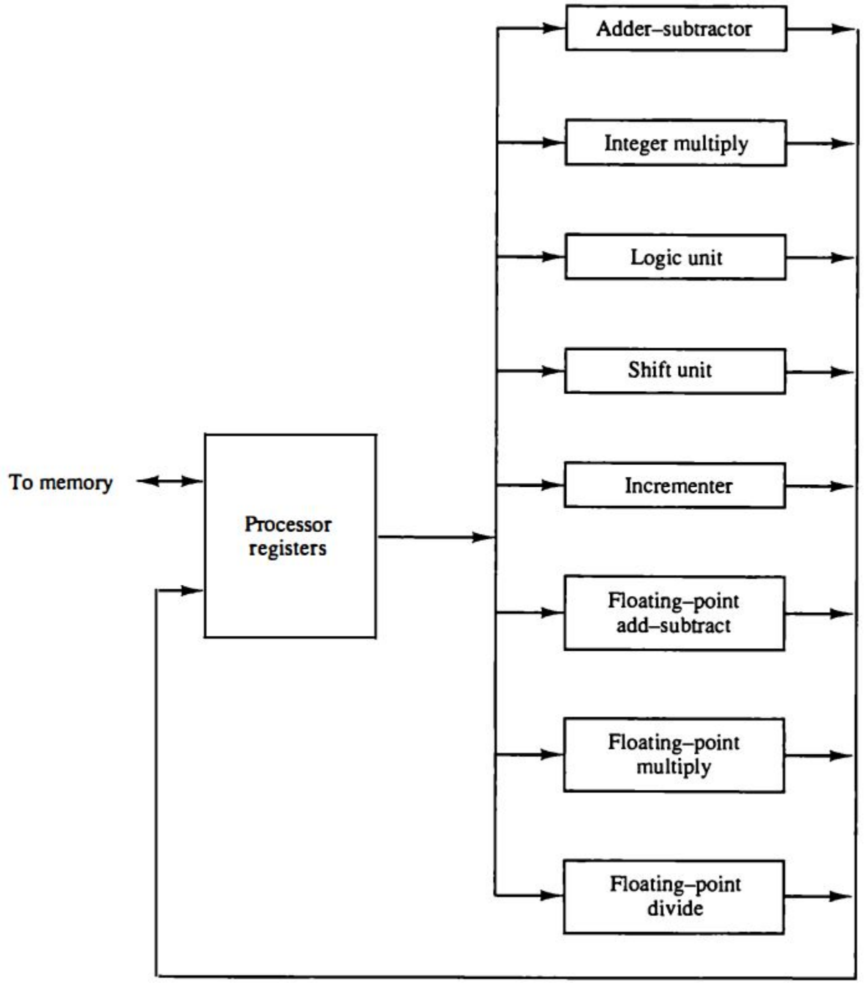
Processor with multiple functional unit

Figure 9-1 shows one possible way of separating the execution unit into eight functional units operating in parallel.

The operands in the registers are applied to one of the units depending on the operation specified by the instruction associated with the operands.

The operation performed in each functional unit is indicated in each block of the diagram.

Figure 9-1 Processor with multiple functional units.



Flynn's classification

There are a variety of ways that parallel processing can be classified. It can be considered from the internal organization of the processors, from the interconnection structure between processors, or from the flow of information through the system. One classification introduced by M. J. Flynn.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)***

Continue..

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. Parallel processing in this case may be achieved by means of multiple functional units or by pipeline processing.

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. The shared memory unit must contain multiple modules so that it can communicate with all the processors simultaneously.

Continue..

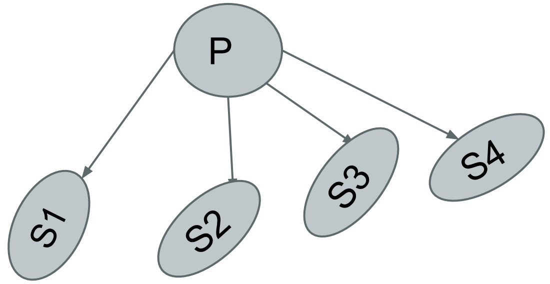
MISD structure is only of theoretical interest since no practical system has been constructed using this organization.

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.

Pipelining

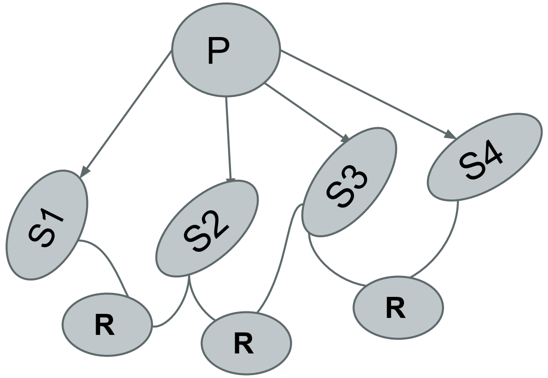
Pipelining

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



Pipelining

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 final result is obtained after the data have passed through all segments.



Example

The pipeline organization will be demonstrated by means of a simple example. Suppose that we want to perform the combined multiply and add operations with a stream of numbers.

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

Continue...

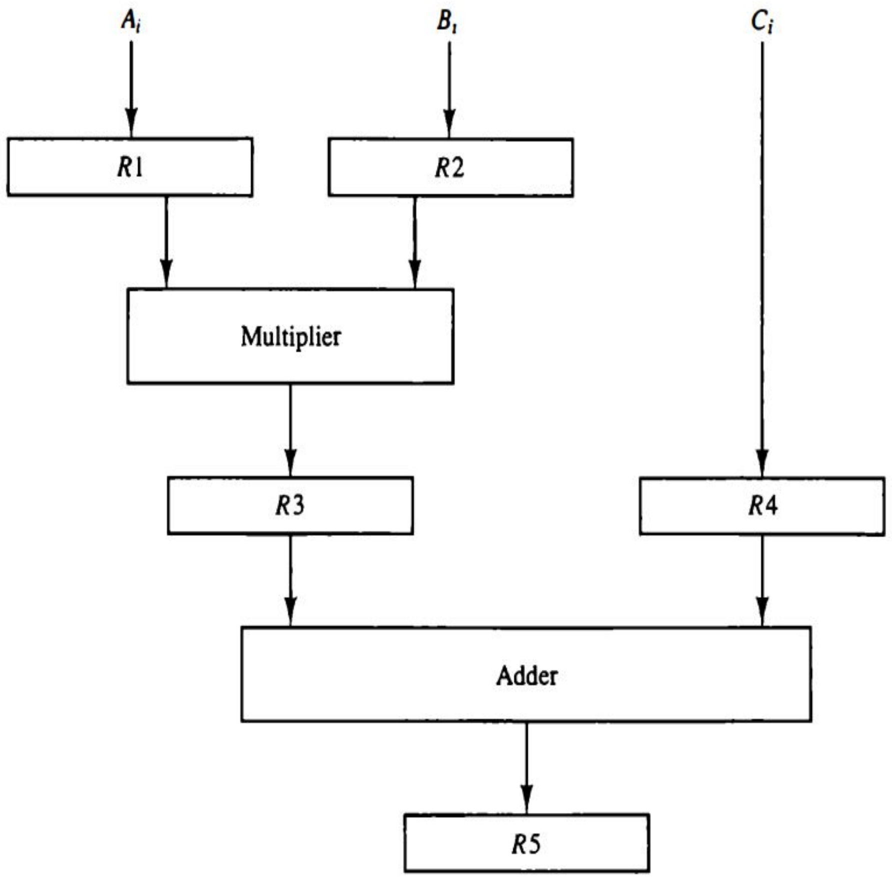
$$A_i * B_i + C_i \qquad \text{for } i = 1, 2, 3, \ldots, 7$$

Each sub operation is to be implemented in a segment within a pipeline. The sub operations performed in each segment of the pipeline are as follows:

$R1 \leftarrow A_i, \quad R2 \leftarrow B_i$	Input A_i and B_i
$R3 \leftarrow R1 * R2, \quad R4 \leftarrow C_i$	Multiply and input C_i
$R5 \leftarrow R3 + R4$	Add C_i to product

Continue...

Figure 9-2 Example of pipeline processing.



Continue...

The five registers are loaded with new data every clock pulse.

The first clock pulse transfers A1 and B1 into R1 and R2.

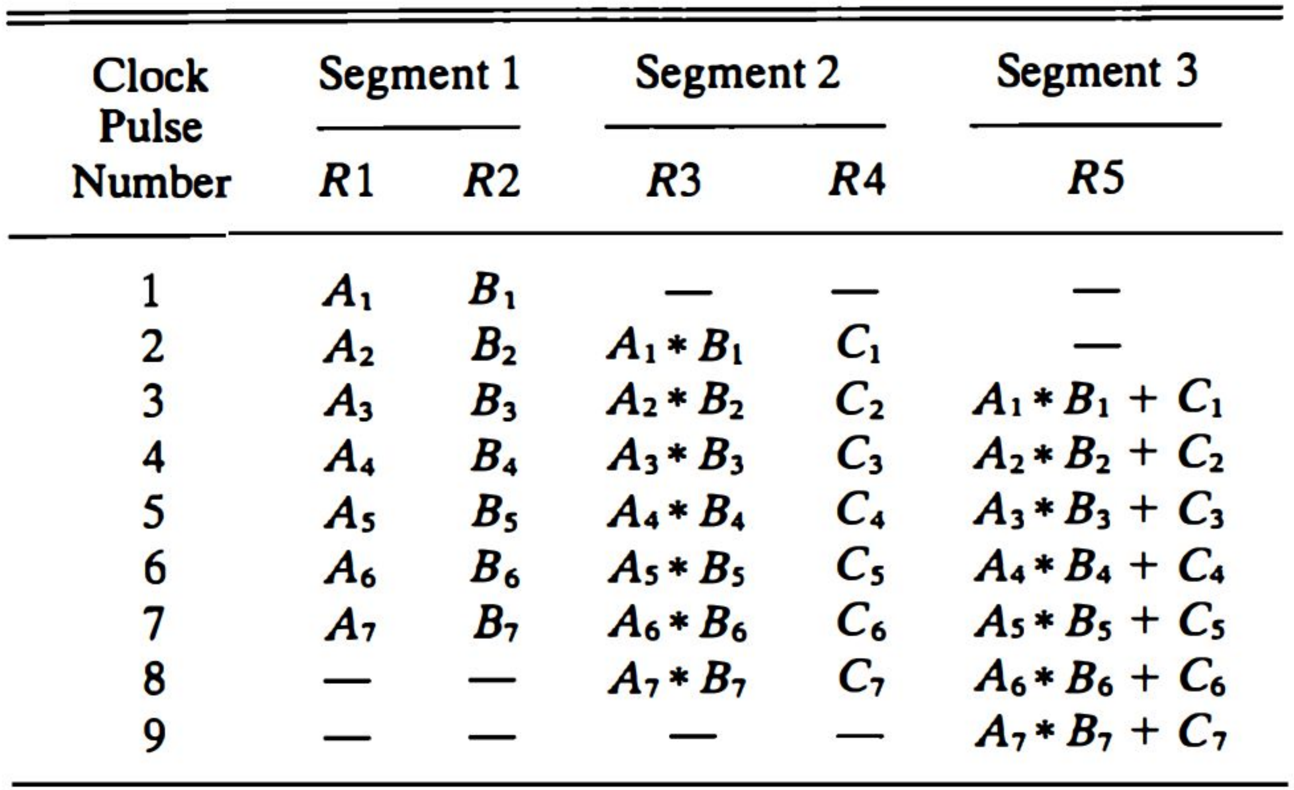
The second clock pulse transfers the product of R1 and R2 into R3 and C1 into R4.

The same clock pulse transfers A2 and B2 into R1 and R2.

The third clock pulse operates on all three segments simultaneously.

Continue..

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	—	—	—	—	$A_7 * B_7 + C_7$

Four-segment pipeline.

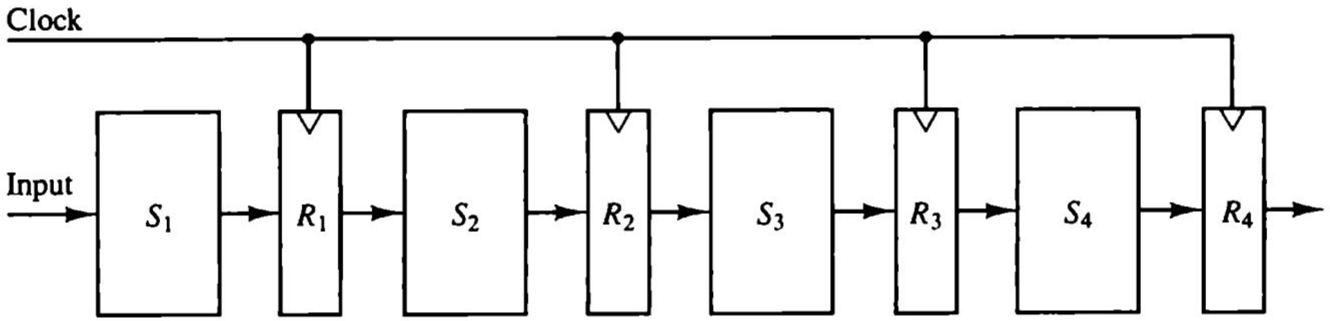


Figure 9-3 Four-segment pipeline.

Instruction Pipeline

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 parallel operations.

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.

Instruction cycle

Computers with complex instructions require other phases in addition to the fetch and execute to process an instruction completely. 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.

Example: Four-Segment 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 register so that the instruction execution and storing of the result can be combined into one segment. This reduces the instruction pipeline into four segments.

Continue...

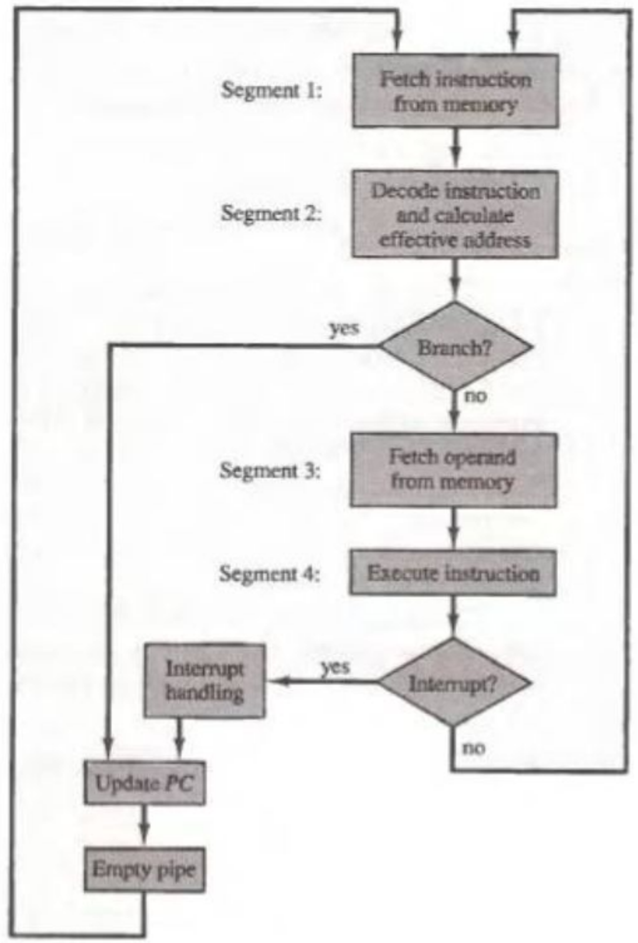


Figure 9-7 Four-segment CPU pipeline.

Continue...

Figure 9-7 shows how the instruction cycle in the CPU can be processed with a four-segment pipeline.

While an instruction is being executed in segment 4, the next instruction in sequence is busy fetching an operand from memory in segment 3.

The effective address may be calculated in a separate arithmetic circuit for the third instruction, and whenever the memory is available, the fourth and all subsequent instructions can be fetched and placed in an instruction FIFO.

Thus up to four sub operations in the instruction cycle can overlap and up to four different instructions can be in progress of being processed at the same time.

Continue...

Figure 9-8 shows the operation of the instruction pipetine. The time in the axis is divided into steps of equal duration. The four segments are represented in the diagram with an abbreviated symbol.

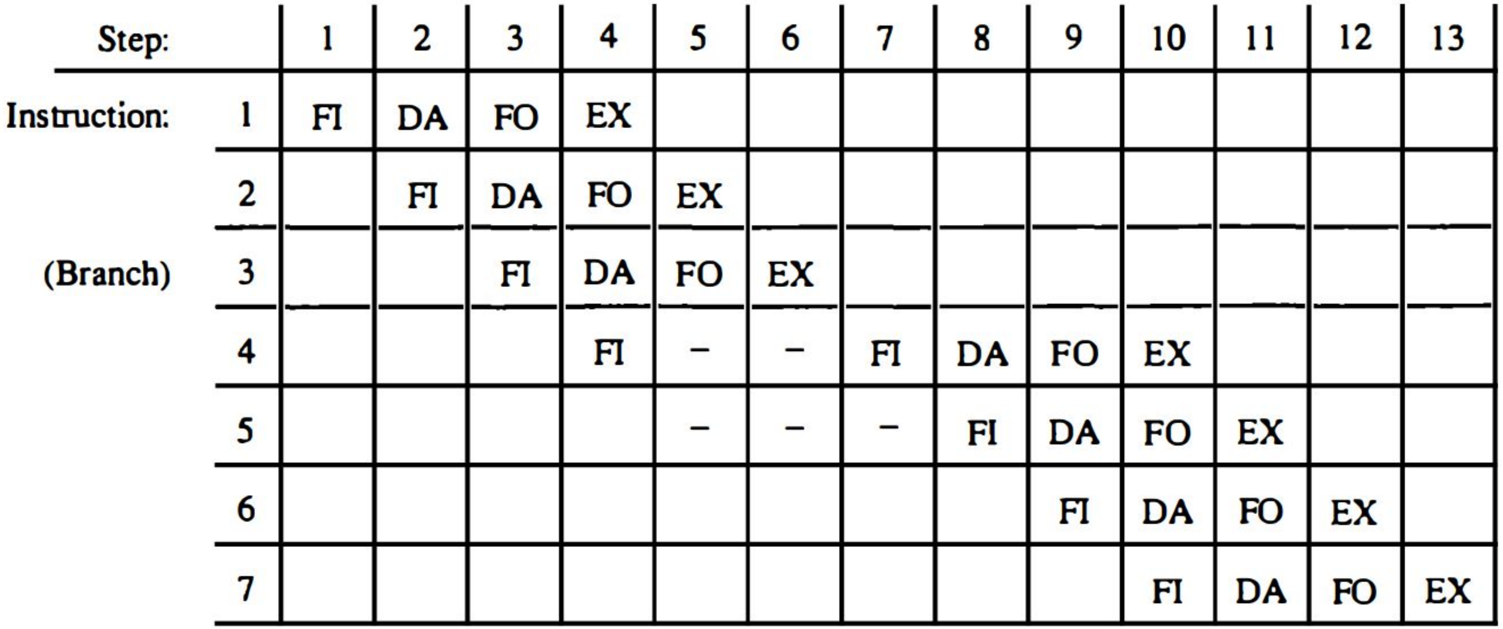
1. FI is the segment that fetches an instruction.

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.

Timing of instruction pipeline.



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.

Continue...

It is assumed that the processor has separate instruction and data memories so that the operation in FI and FO can proceed at the same time. In the absence of a branch instruction, each segment operates on different instructions.

Thus, in step 4, instruction 1 is being executed in segment EX; the operand for instruction 2 is being fetched in segment FO; instruction 3 is being decoded in segment DA; and instruction 4 is being fetched from memory in segment FI.

Continue...

Assume now 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.

If the branch is taken, a new instruction is fetched in step 7. If the branch is not taken, the instruction fetched previously in step 4 can be used.

The pipeline then continues until a new branch instruction is encountered.

Continue...

Another delay may occur in the pipeline if the EX segment needs to store the result of the operation in the data memory while the FO segment needs to fetch an operand. In that case, segment FO must wait until segment EX has finished its operation.

Continue...

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.

Dependencies and Data Hazard in Pipeline

Dependencies and Data Hazard in Pipeline

In the domain of central processing unit (CPU) design, hazards are problems with the instruction pipeline in CPU microarchitectures when the next instruction cannot execute in the following clock cycle (1) and can potentially lead to incorrect computation results.

Dependencies and Data Hazard in Pipeline

There are mainly three types of dependencies possible in a pipelined processor. These are :

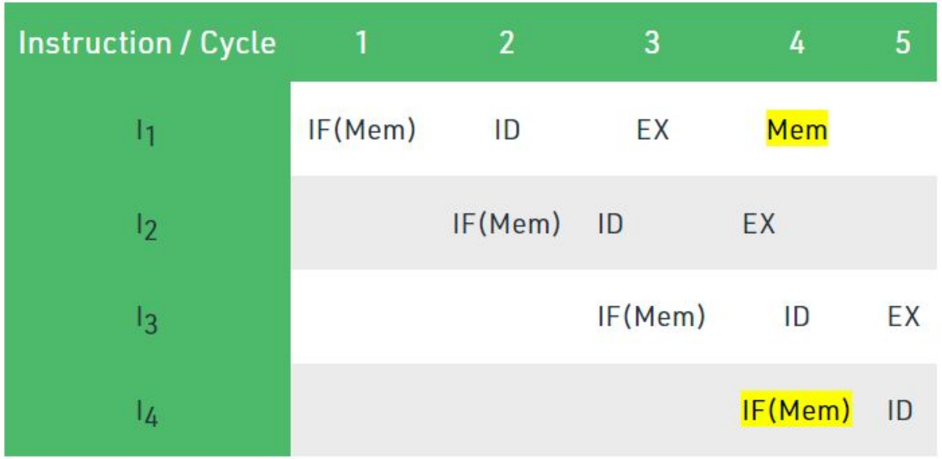
1) Structural Dependency

2) Control Dependency

3) Data Dependency

Structural dependency

This dependency arises due to the resource conflict in the pipeline. A resource conflict is a situation when more than one instruction tries to access the same resource in the same cycle. A resource can be a register, memory, or ALU.

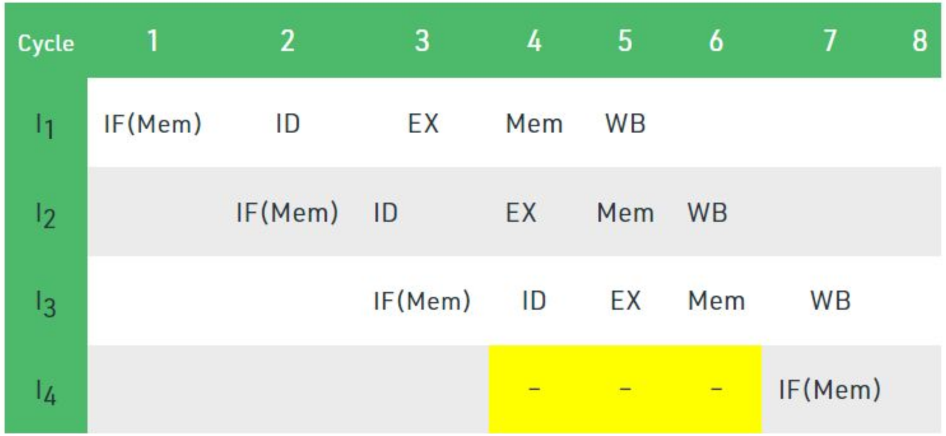


Instruction / Cycle	1	2	3	4	5
I_1	IF(Mem)	ID	EX	Mem	
I_2		IF(Mem)	ID	EX	
I_3			IF(Mem)	ID	EX
I_4				IF(Mem)	ID

Structural dependency

In the above scenario, in cycle 4, instructions I1 and I4 are trying to access same resource (Memory) which introduces a resource conflict.

To avoid this problem, we have to keep the instruction on wait until the required resource (memory in our case) becomes available. This wait will introduce stalls in the pipeline as shown below:

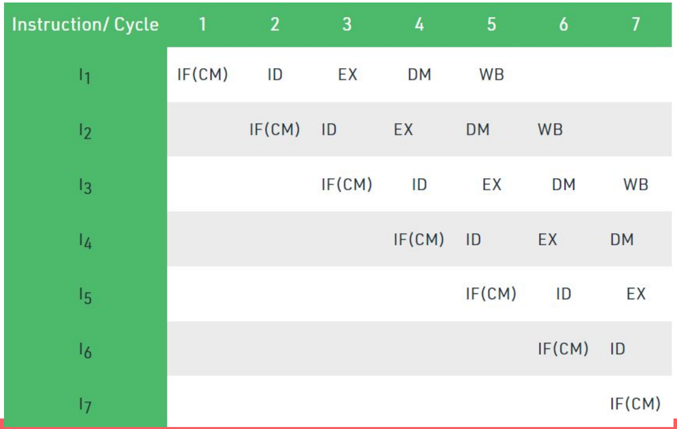


Cycle	1	2	3	4	5	6	7	8
I_1	IF(Mem)	ID	EX	Mem	WB			
I_2		IF(Mem)	ID	EX	Mem	WB		
I_3			IF(Mem)	ID	EX	Mem	WB	
I_4				–	–	–	IF(Mem)	

Solution for structural dependency

To minimize structural dependency stalls in the pipeline, we use a hardware mechanism called Renaming.

Renaming : According to renaming, we divide the memory into two independent modules used to store the instruction and data separately called Code memory(CM) and Data memory(DM) respectively. CM will contain all the instructions and DM will contain all the operands that are required for the instructions.



Instruction/ Cycle	1	2	3	4	5	6	7
I_1	IF(CM)	ID	EX	DM	WB		
I_2		IF(CM)	ID	EX	DM	WB	
I_3			IF(CM)	ID	EX	DM	WB
I_4				IF(CM)	ID	EX	DM
I_5					IF(CM)	ID	EX
I_6						IF(CM)	ID
I_7							IF(CM)

Data Dependency (Data Hazard)

Data hazards occur when instructions that exhibit data dependence, modify data in different stages of a pipeline. Hazard cause delays in the pipeline. There are mainly three types of data hazards:

1) RAW (Read after Write) [Flow/True data dependency]

2) WAR (Write after Read) [Anti-Data dependency]

3) WAW (Write after Write) [Output data dependency]

Continues...

Let there be two instructions I and J, such that J follow I. Then, RAW hazard occurs when instruction J tries to read data before instruction I writes it.

Eg:

I: R2 <- R1 + R3

J: R4 <- R2 + R3

WAR hazard occurs when instruction J tries to write data before instruction I reads it.

Eg:

I: R2 <- R1 + R3

J: R3 <- R4 + R5

Continues..

WAW hazard occurs when instruction J tries to write output before instruction I writes it.

Eg:

I: R2 <- R1 + R3

J: R2 <- R4 + R5

WAR and WAW hazards occur during the out-of-order execution of the instructions

Example: Three-Segment Instruction Pipeline

The control section fetches the instruction from program memory into an instruction register. The instruction is decoded at the same time that the registers needed for the execution of the instruction are selected. The processor unit consists of a number of registers and an arithmetic logic unit (ALU) that performs the necessary arithmetic, logic, and shift operations. A data memory is used to load or store the data from a selected register in the register file.

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

I: Instruction fetch

A: ALU operation

E: Execute instruction

Continues...

Consider now the operation of the following four instructions:

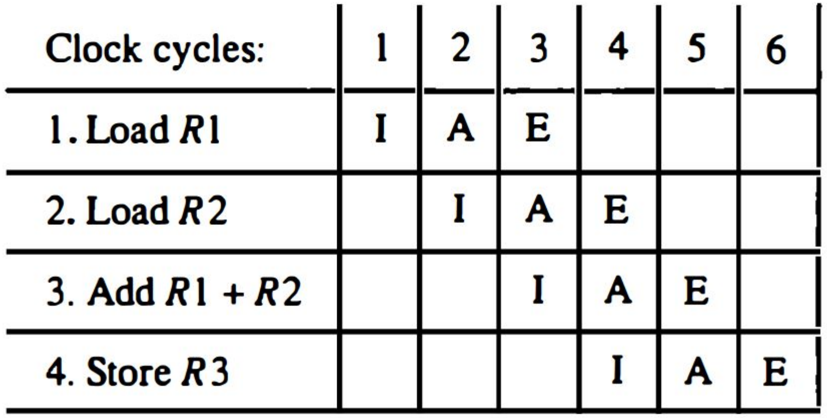
1. LOAD: RI <--M [address 1]

2. LOAD: R2 <--M [address 2]

3. ADD: R3 <--RI + R2

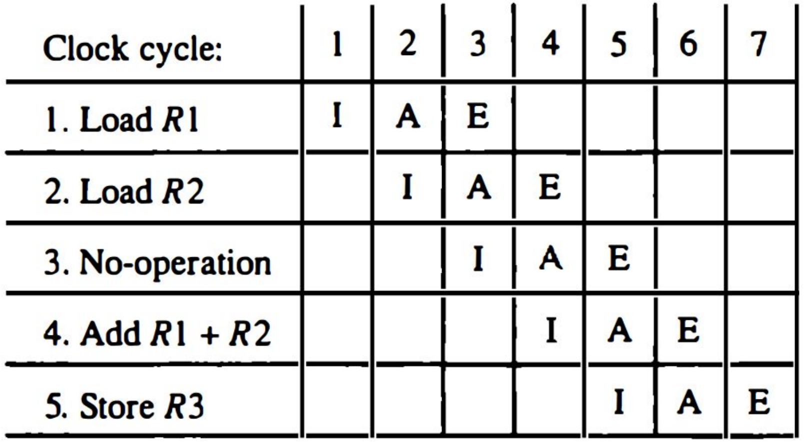
4. STORE: M[address 3] <--R3

Three-segment pipeline timing.



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

Control Dependency (Branch Hazards)

This type of dependency occurs during the transfer of control instructions such as BRANCH, CALL, JMP, etc. On many instruction architectures, the processor will not know the target address of these instructions when it needs to insert the new instruction into the pipeline. Due to this, unwanted instructions are fed to the pipeline.

Example:

The program for this example consists of five instructions:

Load from memory to R1

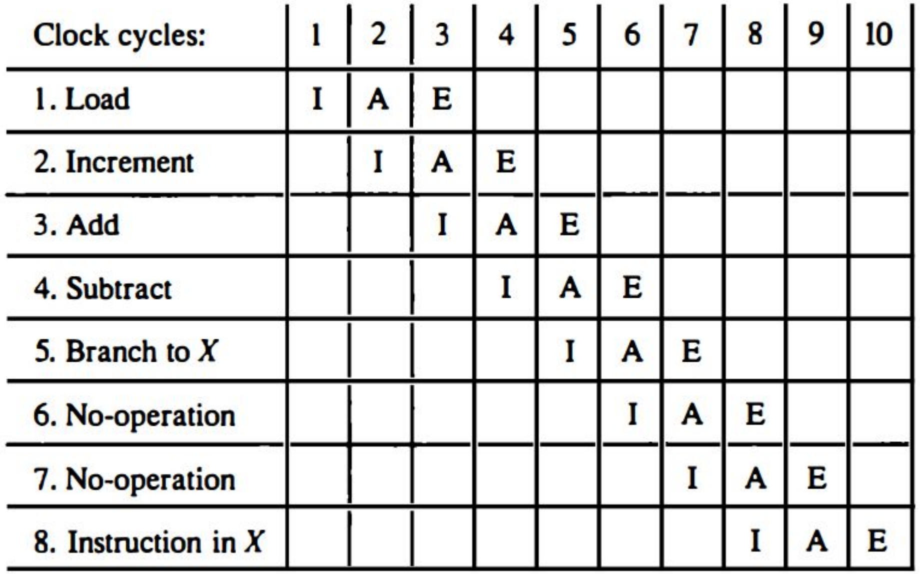
Increment R2

Add R3 to R4

Subtract R5 from R6

Branch to address X

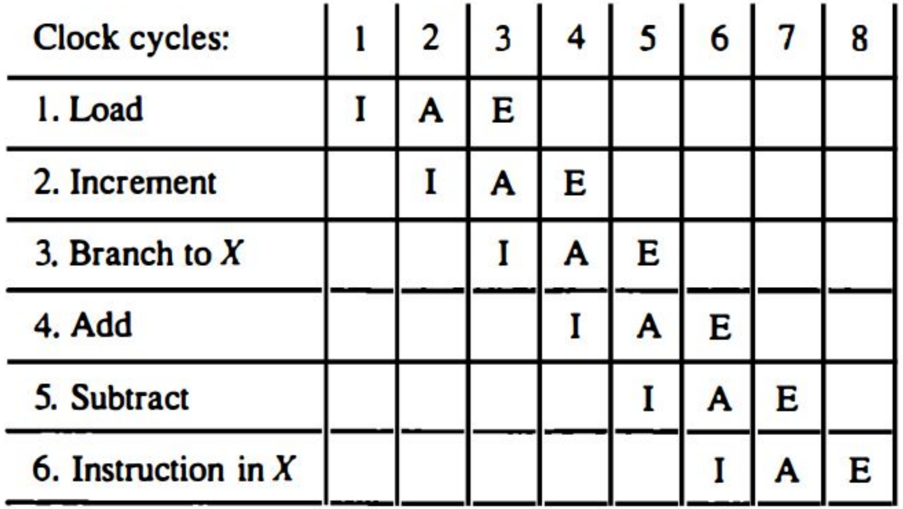
Continues...



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

Continues...



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