SECR2033
Computer Organization and Architecture

Lecture slides prepared by “Computer Organization and Architecture”, 9/e, by William Stallings, 2013.

Module 5
Central Processing Unit (CPU)

Objectives:

- To learn the components common to every CPU.
- Be able to explain how each component in CPU contributes to instruction cycle.
- To understand the concept of pipelining in CPU execution.
- Be able to understand micro-operations basis for the design and implementation of the control unit.

Module 5
Central Processing Unit (CPU)

5.1 Processor Organization
5.2 Register Organization
5.3 Instruction Cycles

5.4 Instruction Pipelining

5.5 Control Unit Operation
5.6 Microprogrammed Control

5.7 Summary

Module 5b
Central Processing Unit (CPU)

5.1 Processor Organization

5.2 Register Organization

5.3 Instruction Cycles

5.4 Instruction Pipelining

- Overview
- Pipeline Strategy
- Pipeline Performance
- Pipeline Hazards (Limitations)

5.5 Control Unit Operation

5.6 Microprogrammed Control

5.7 Summary

5 Overview

- Organizational enhancements to the processor can improve performance another approach → instruction pipelining.*
- CPU break *fetch-decode-execute* cycle into tasks that performed in parallel so that more than one tasks can be executed at a time.

- The concept of pipelining is to allow the processing of a new task even though the processing of previous task has not ended.

* William Stallings (2016). *Computer Organization and Architecture: Designing for Performance* (10th Edition). United States: Pearson Education Limited, p.500.

Analogy: Pipeline Laundry

- Anyone who has done a lot of laundry has intuitively used pipelining.

- The non-pipelined approach to laundry would be as follows:

1. Place one dirty load of clothes in the washer.
2. When the washer is finished, place the wet load in the dryer.
3. When the dryer is finished, place the dry load on a table and fold.
4. When folding is finished, ask your roommate to put the clothes away.

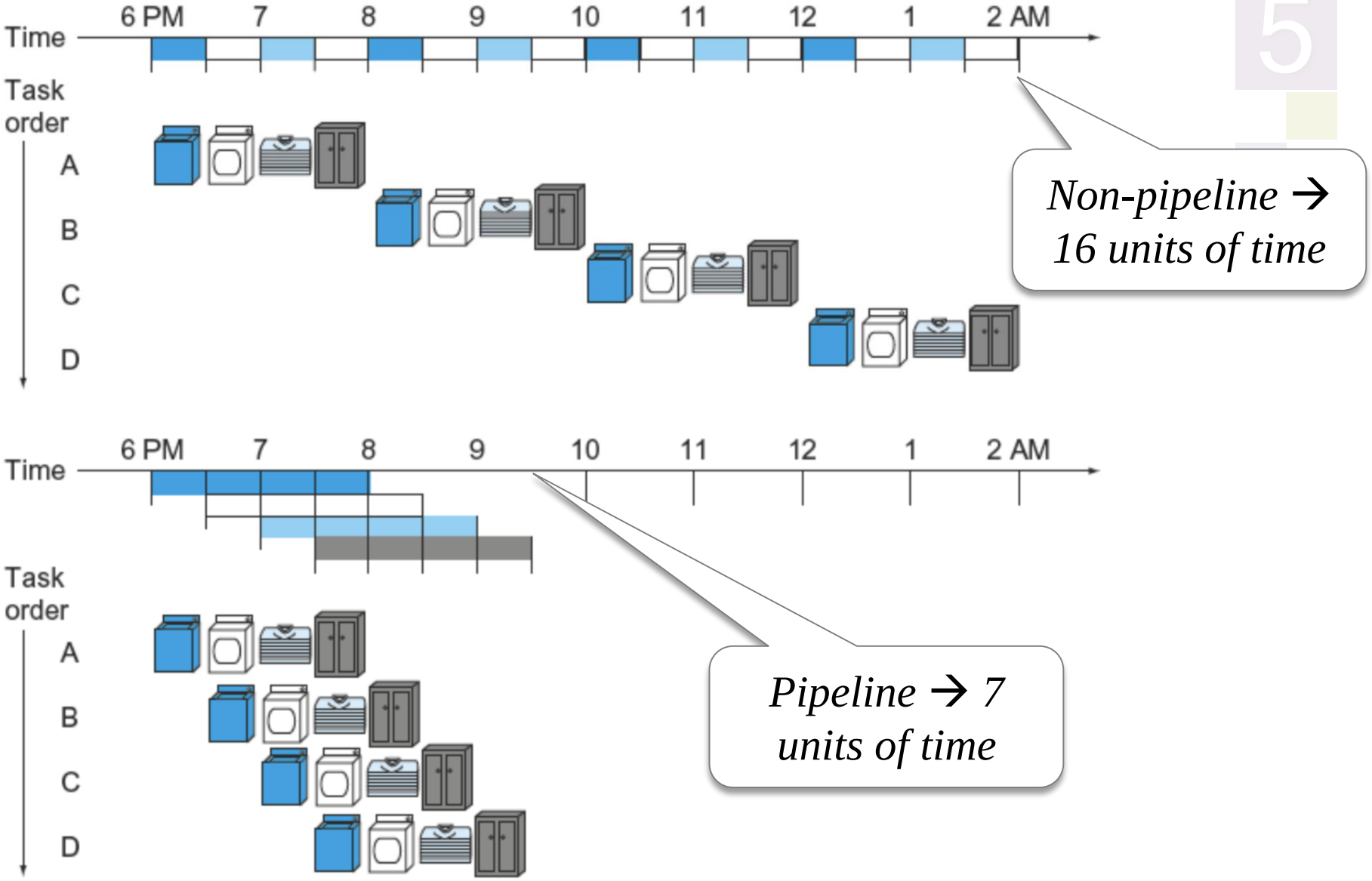


Figure: The laundry analogy (4 loads: A, B, C, D) for pipelining.

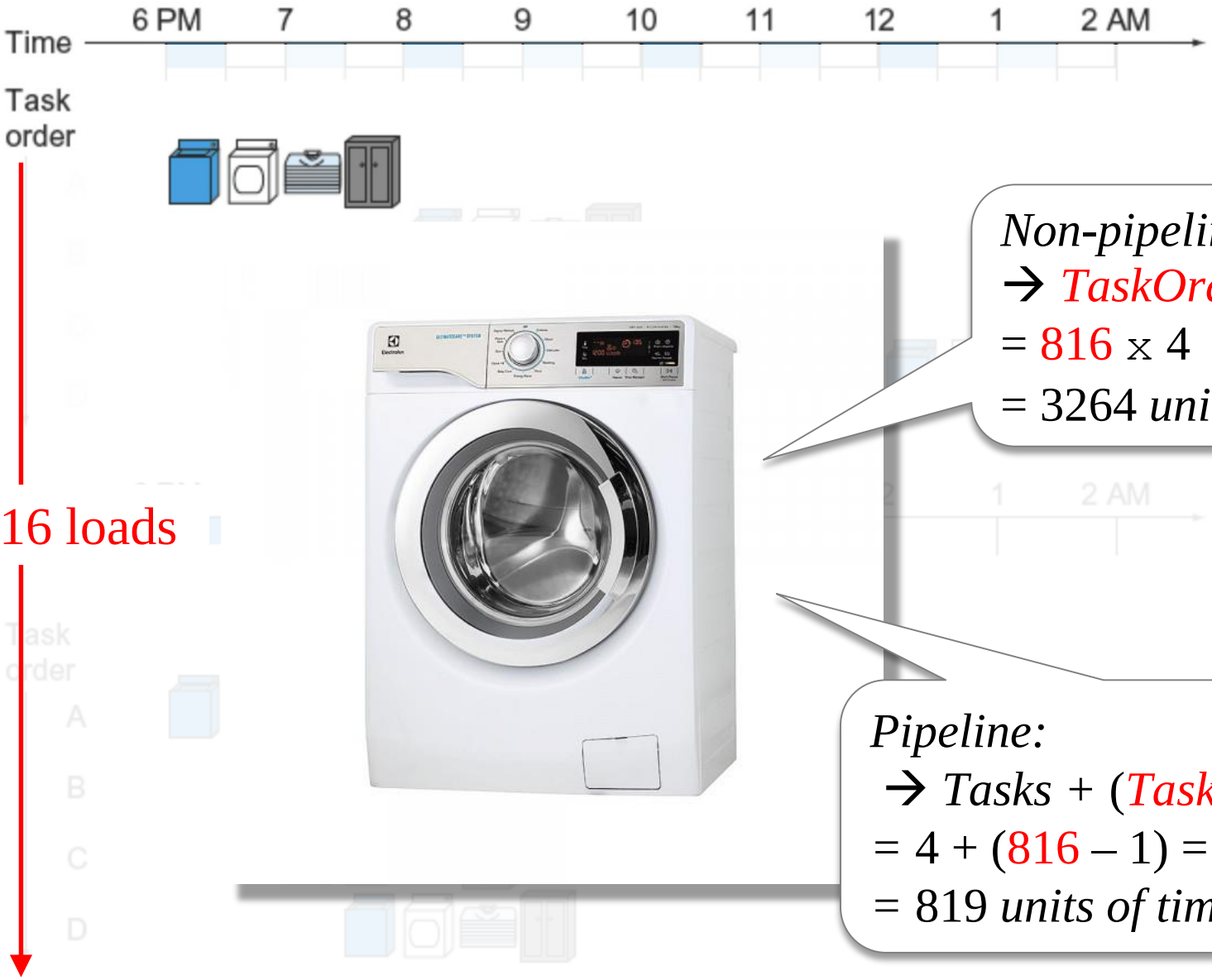
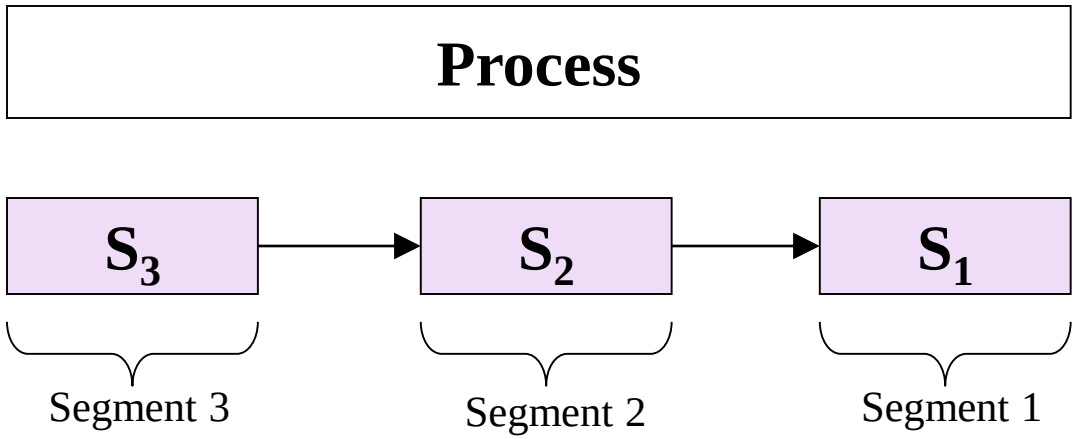


Figure: The laundry analogy (816 loads) for pipelining.

Instruction Pipeline



- A single instruction cycle process is divided into several small independent tasks (or segments).
- Some segments can be performed in parallel.

With the use of pipeline, the average CPU is reduced.

1) Data operands pass through all segments (S) in sequence.
2) Each segment consists of a circuit that performs sub-operation.
3) Segments are separated by registers (R), that hold the intermediate results between stages.

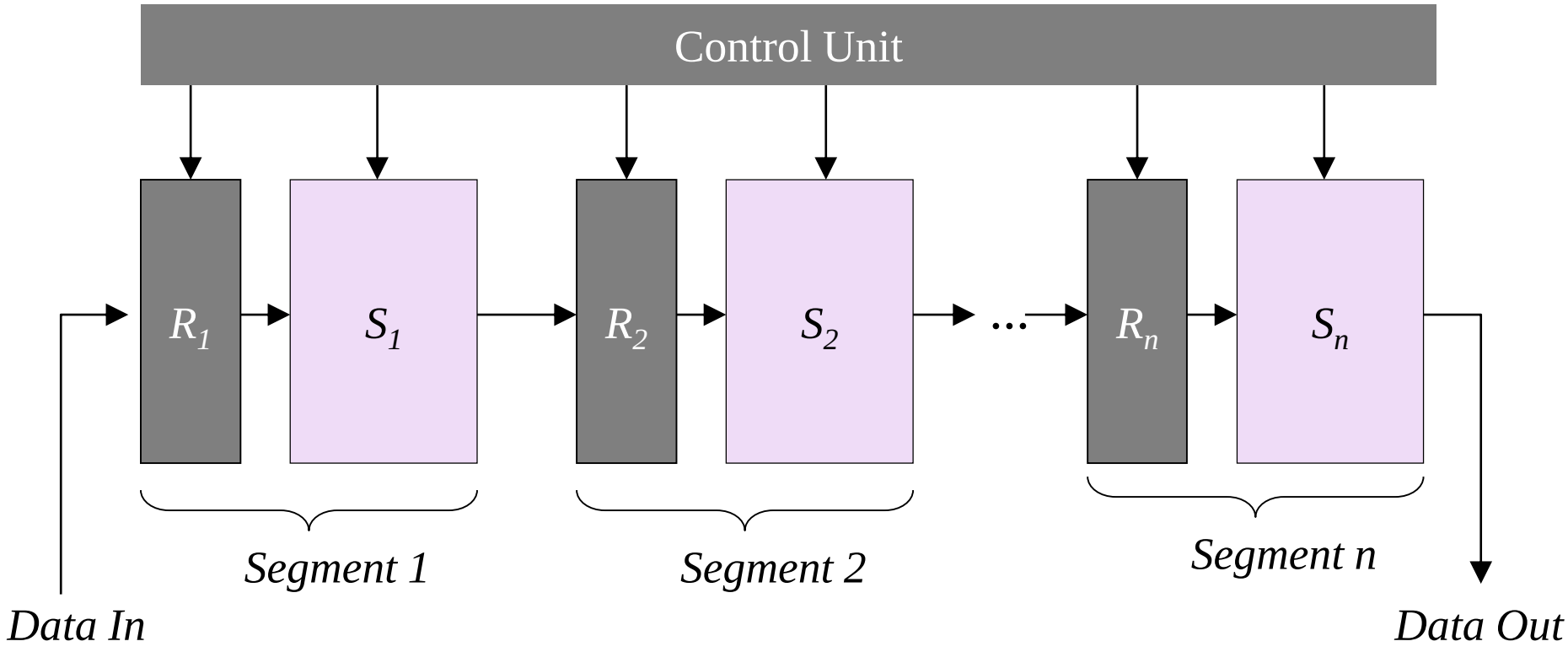
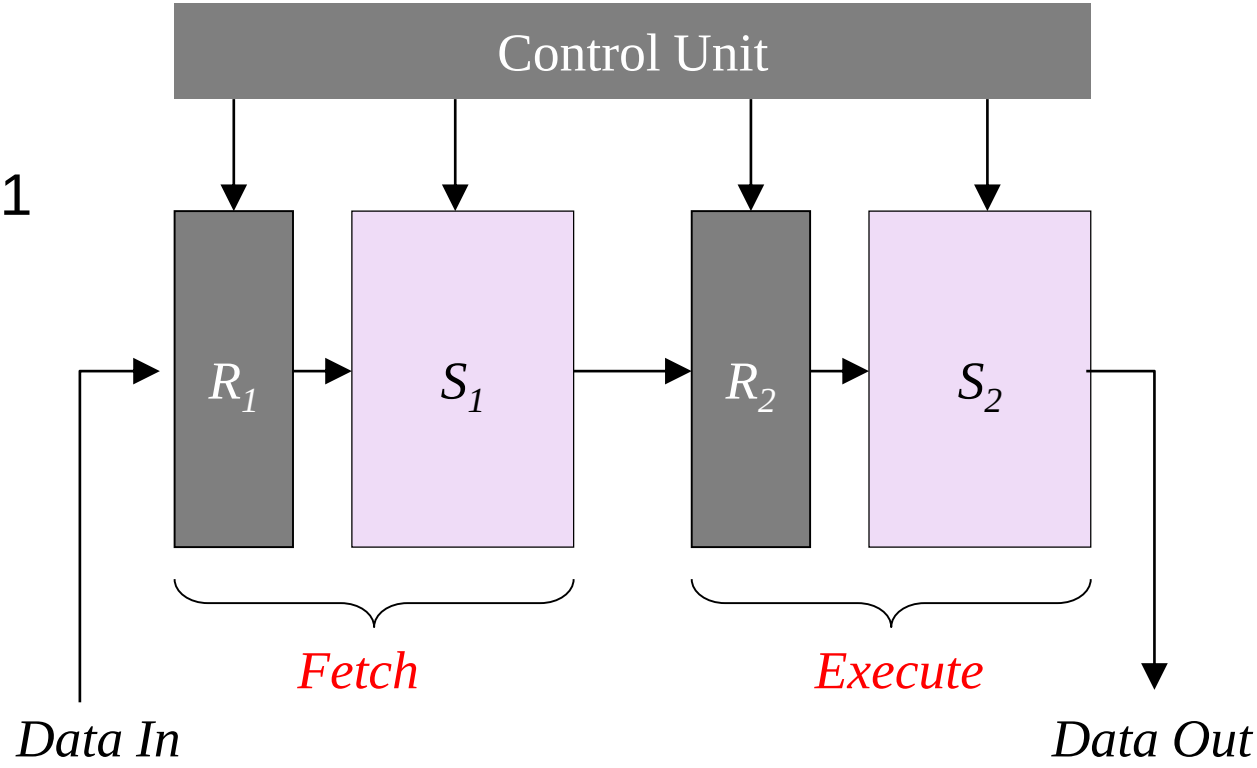
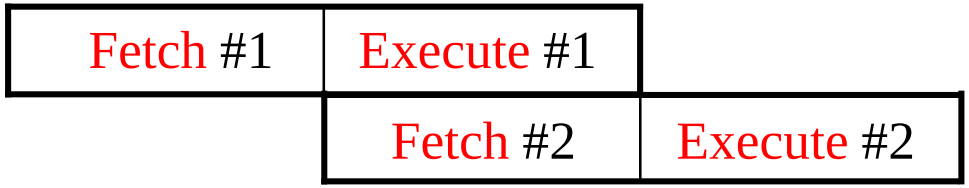


Figure: CPU pipeline structure.

Example 9: Case 1



Pipeline



5 Pipeline Strategy

- The process is referred to as pipelining when new inputs are accepted at one end before previously accepted inputs appear as outputs at the other end.
- As a simple approach, consider subdividing instruction processing into two stages:
  - fetch instruction and
  - execute instruction.

Instruction prefetch or fetch overlap;

This process will speed up instruction execution.

- There are times during the execution of an instruction when main memory is not being accessed.
- This time could be used to fetch the next instruction in parallel with the execution of the current one.

Figure: Two-stage instruction pipeline: *simplified view*

Execution will involve reading and storing operands and the performance of some operation. Thus, the fetch stage may have to wait for some time before it can empty its buffer

- This doubling of execution rate is unlikely for two reasons:

1) The execution time will generally be longer than the fetch time.
2) A conditional branch instruction makes the address of the next instruction to be fetched unknown.

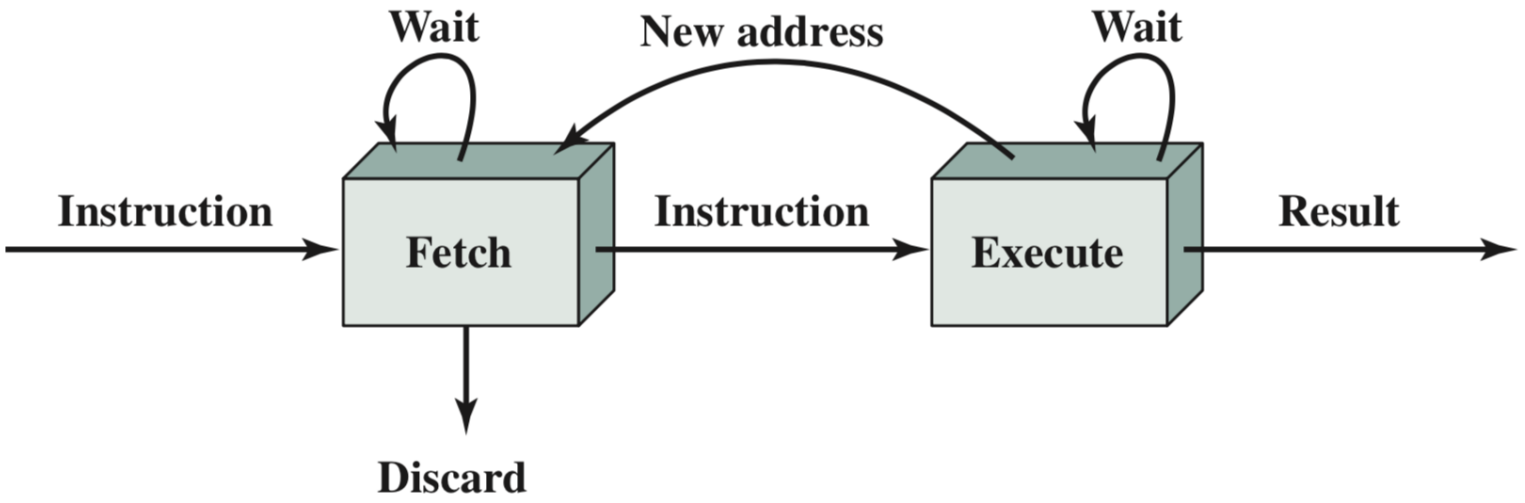


Figure: Two-stage instruction pipeline: *expanded view*

A conditional branch instruction makes the address of the next instruction to be fetched unknown. Thus, the fetch stage must wait until it receives the next instruction address from the execute stage. The execute stage may then have to wait while the next instruction is fetched

- This doubling of execution rate is unlikely for two reasons:

1) The execution time will generally be longer than the fetch time.
2) A conditional branch instruction makes the address of the next instruction to be fetched unknown.

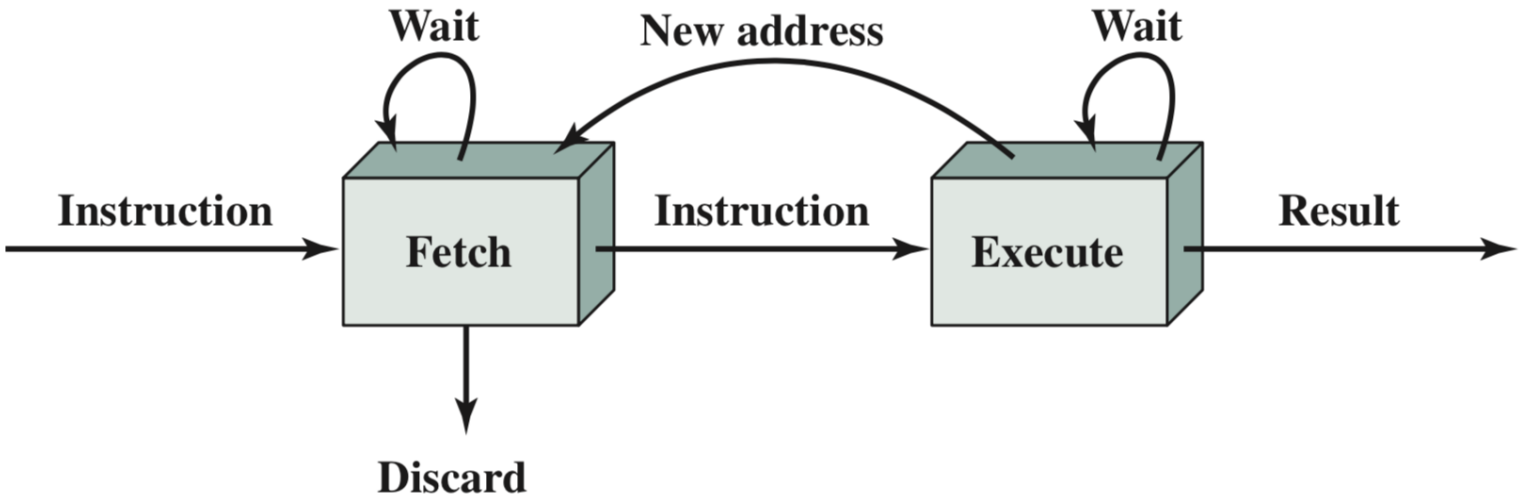


Figure: Two-stage instruction pipeline: *expanded view*

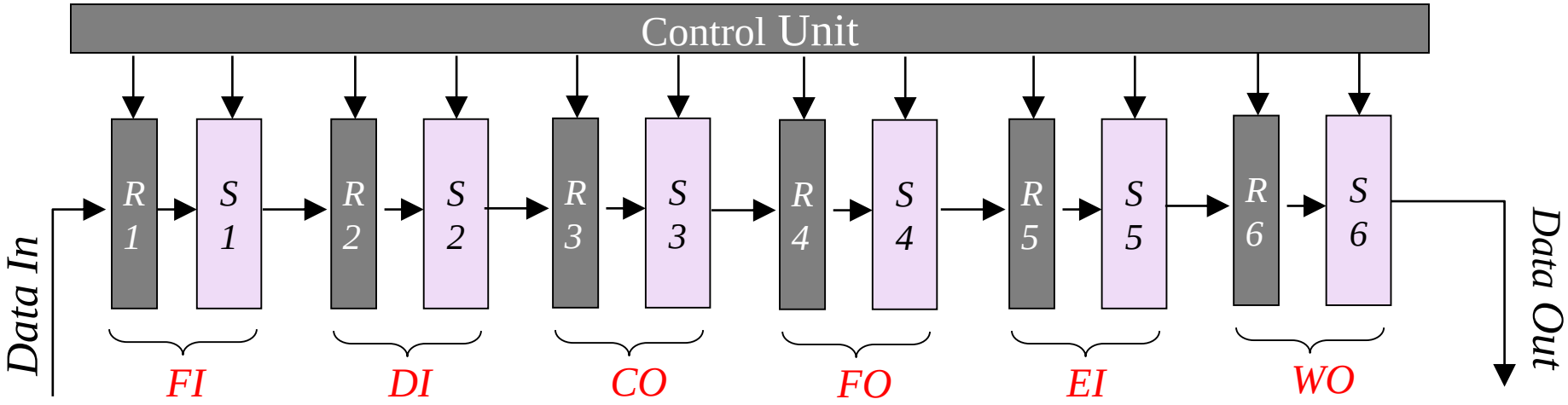
- To gain further speedup, the pipeline must have more stages.
- Let consider the following decomposition of the instruction processing:

  - *Fetch Instruction* (FI): Read next expected instruction into buffer.
  - *Decode Instruction* (DI): Determine the opcode and the operand specifiers.
  - *Calculate Operands* (CO): Calculate the effective address of each source operand.
  - *Fetch Operands* (FO): Fetch each operand from memory.
  - *Execute Instruction* (EI): Perform the indicated operation and store the result.
  - *Write Operand* (WO): Store the result in memory.

Example 10: Case 2

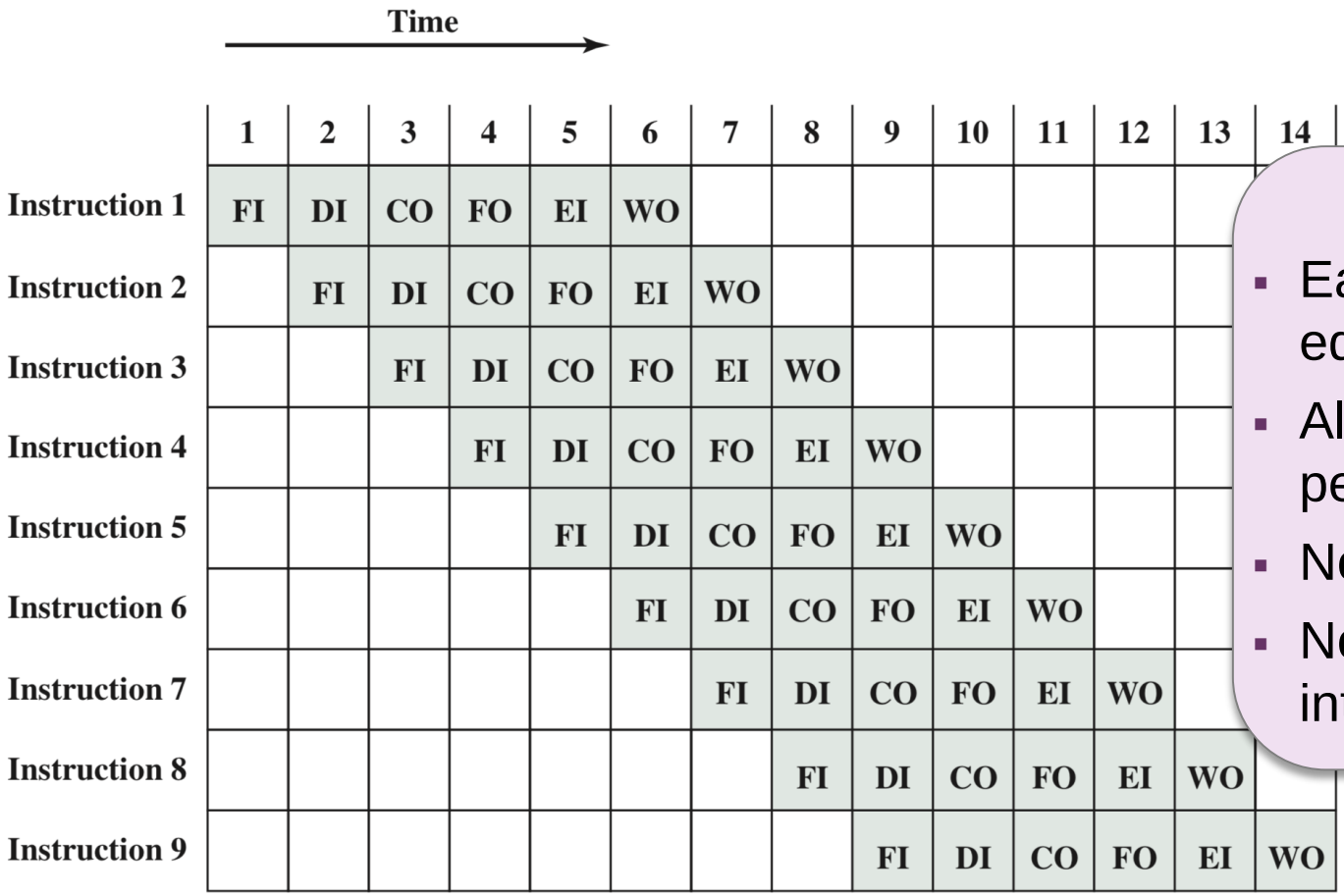
To implement the instruction cycle with multiple stages in simpler form:

- Use multiple execution "functional units" to parallelize the actual execution phase of several instructions.



Example 11:

Figure shows that a 6 stages pipeline can reduce the execution time for 9 instructions from 54 time units to 14 time units.



Assumptions:

- Each stages → equal duration.
- All stages can be performed in parallel.
- No memory conflicts.
- No branching or interrupt happen.

Figure: Timing Diagram for Instruction Pipeline Operation

5 Pipeline Performance

- Here some simple measures of pipeline performance and relative speedup, S.
- Let the execution time, T, required for a non-pipeline and pipeline respectively, and pipeline with k stages to execute n instructions as:

Non-pipeline:

t_n → the cycle time

$$T_n = t_n \times n$$

Pipeline:

t_p → the cycle time

$$T_p = t_p(k + (n-1))$$

Note: $k = 1$

- The speedup, S, for the instruction pipeline compared to execution without the pipeline is defined as:

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

- If n is too large from k, $n >> k$, and when $t_n = k \times t_p$, thus maximum speedup:

$$S_{\max} = k$$

Example 12: Pipeline (Non-ideal case)

Given the 4 segment pipeline whereby each has a delay time as follow:

- Segment 1 : 40*ns*
- Segment 2 : 25*ns*
- Segment 3 : 45*ns*
- Segment 4 : 45*ns*

The delay time for the interface register is 5*ns*. Calculate the:

a) cycle time of the non-pipeline and pipeline.

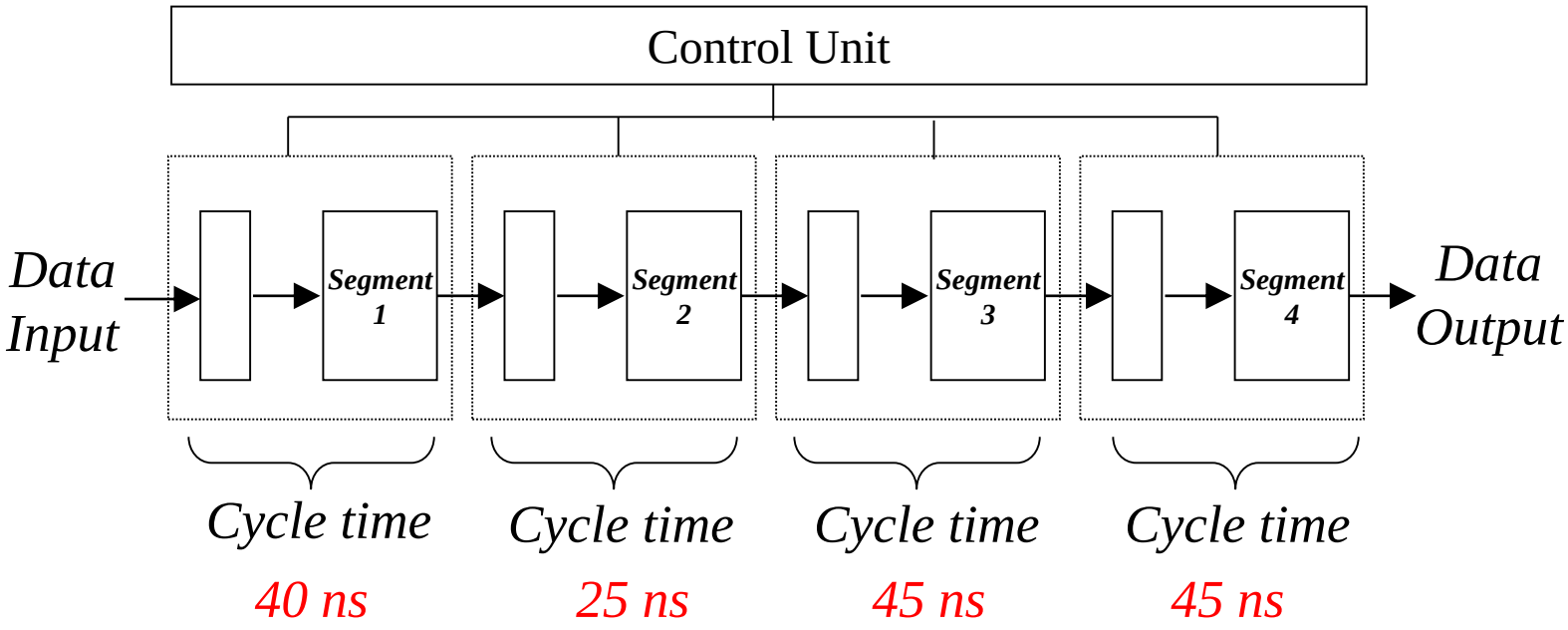
b) execution time for 100 tasks.

c) real speedup.

d) Maximum speedup.

5

Solution :


Control Unit
Data Input
Segment 1
Segment 2
Segment 3
Segment 4
Data Output
Cycle time
Cycle time
Cycle time
Cycle time
40 ns
25 ns
45 ns
45 ns

5

a) Cycle time of the non-pipeline and pipeline.

t_n = (Total executions) + interface delay

$$t_n = (40 + 25 + 45 + 45) + 5 = 160ns$$

t_p = (the longest delay for execution) + interface delay

$$t_p = (45 + 5) = 50ns$$

b) Execution time for 100 tasks.

Non-Pipeline:

$$T_n = t_n \times n$$
$$= 160 \times 100$$
$$= 16000ns$$

Pipeline:

$$T_p = t_p(k + (n - 1))$$
$$= 50\,(4 + (100 - 1))$$
$$= 50\,(103) = 5150ns$$

5

c) Real speedup.

$$
\begin{aligned}
S &= \frac{T_n}{T_p} \\
&= \frac{t_n \times n}{t_p(k + (n-1))} \\
&= \frac{16000}{5150} \\
&= 3.1068
\end{aligned}
$$

d) Maximum speedup.

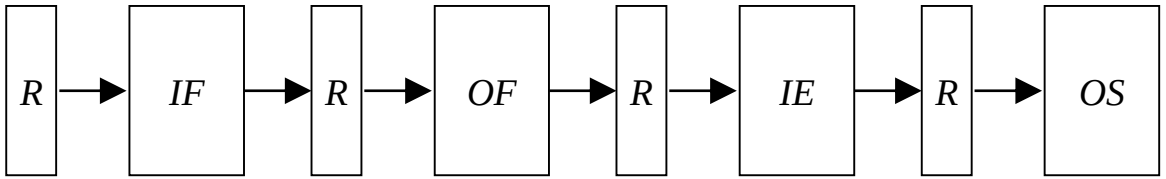
$$S_{\max} = k = 4$$

Activity 2

Example 13:

Segment Name	Segment Execution Time (ns)
Instruction Fetch & Decode (IF)	52
Operand Fetch (OF)	40
Instruction Execute (IE)	30
Operand Store (OS)	40

- There are 4 segments as follows:
- The interface delay is 3 *ns*.
- The simple block diagram of the pipeline:



Draw the space time diagram and calculate the:

a) cycle time of the non-pipeline and pipeline.

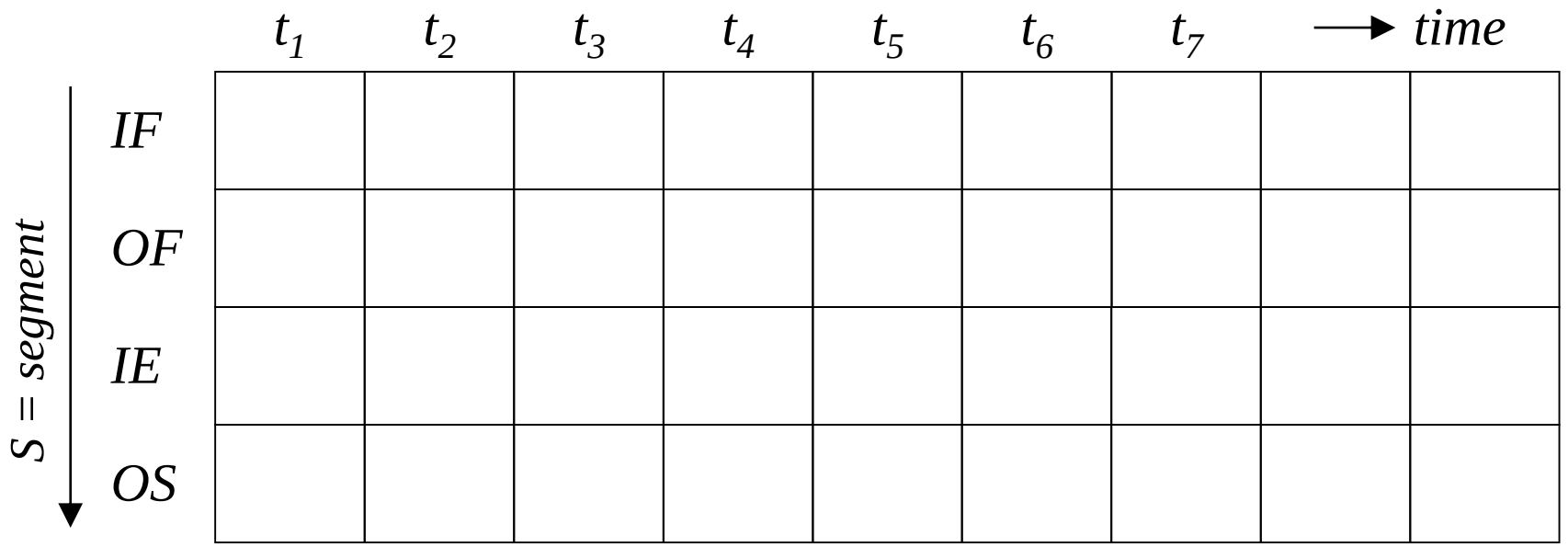
b) execution time for 50 tasks.

c) real speedup.

d) Maximum speedup.

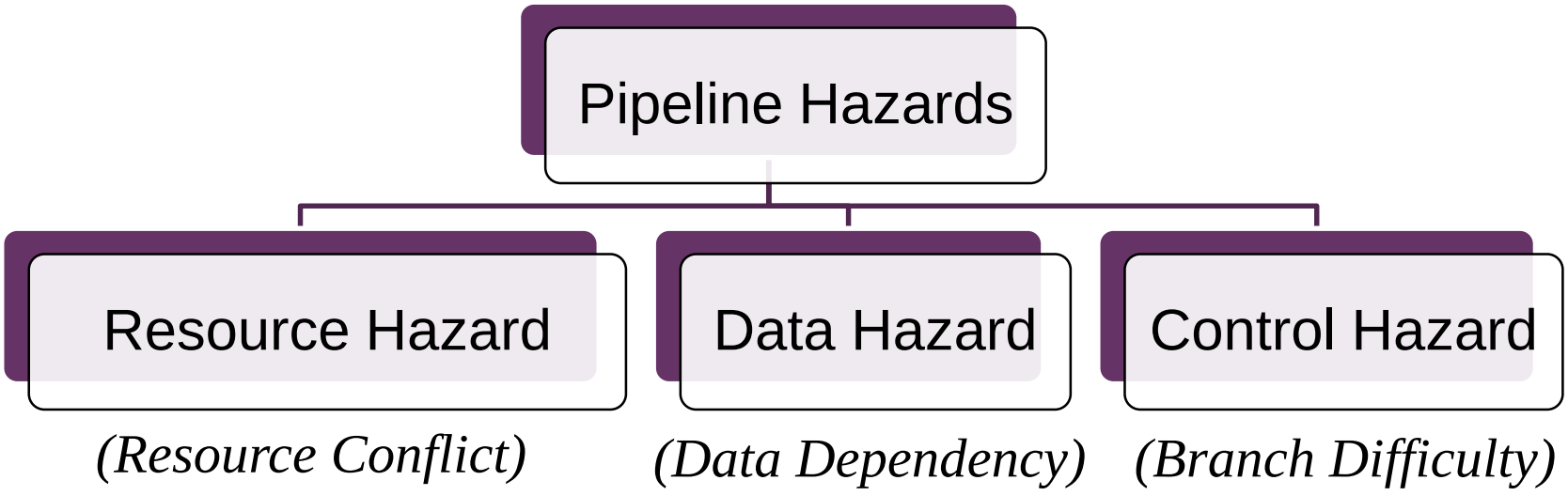
Solution :

- Purpose of diagram – to show the events of a pipeline
- Assume only 4 instructions.



Pipeline Hazards (Limitations)

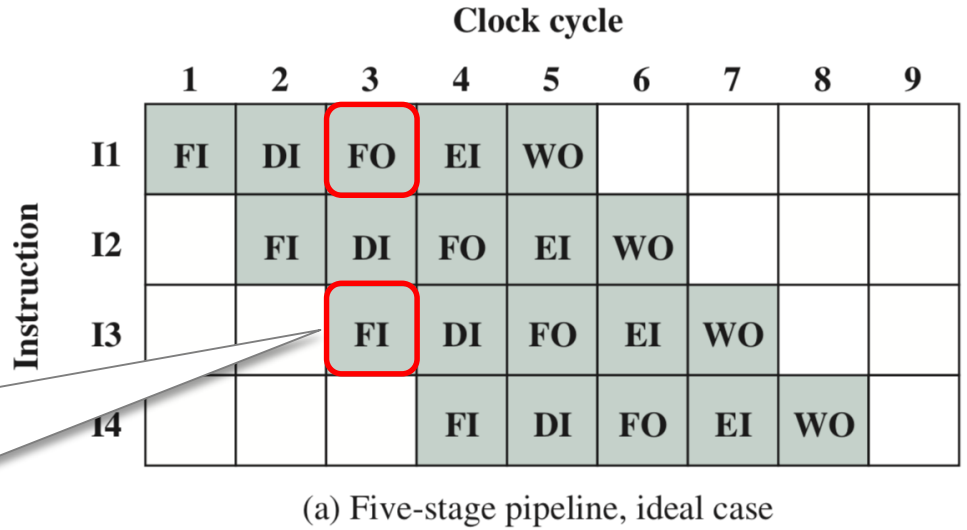
- A pipeline hazard occurs when the pipeline, or some portion of the pipeline, must stall because conditions do not permit continued execution.
- Such a pipeline stall is also referred to as a pipeline bubble.



(a) Resource Hazards

(*Resource Conflict*)

- A resource hazard occurs when two (or more) instructions that are already in the pipeline need the same resource.
- The result is that the instructions must be executed in serial rather than parallel for a portion of the pipeline.



(a) Five-stage pipeline, ideal case

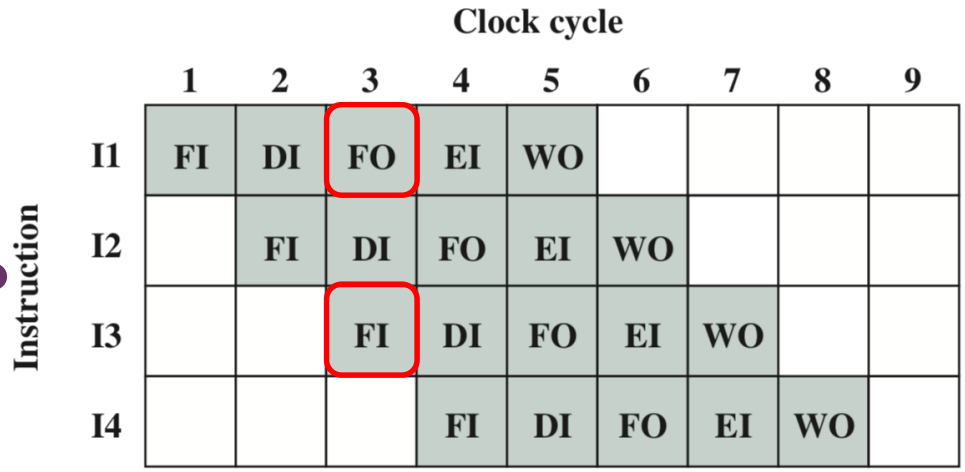
We have a conflict with 2 instructions need the same resource (i.e. memory)
→ *data reads / writes can only happen 1 at a time.*

Figure: Example of resource hazards

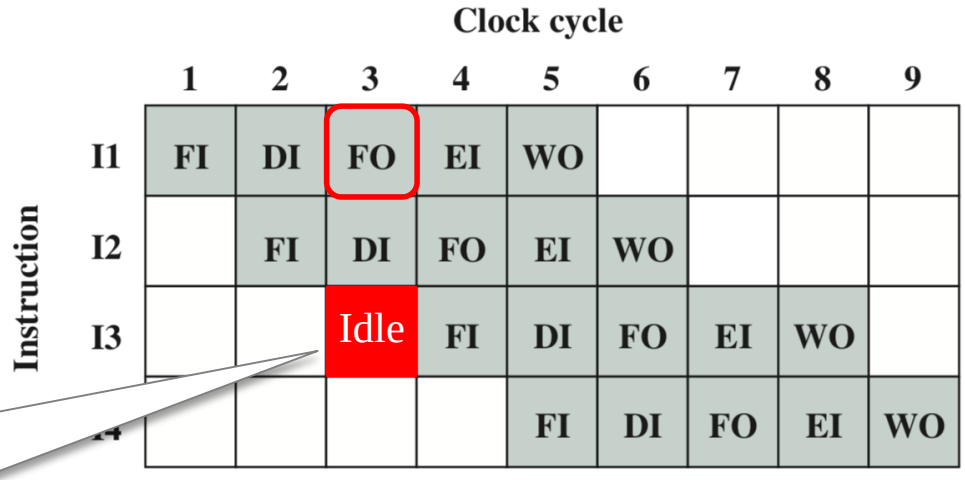
Resource Hazards

Solution:

Fetch Instruction (FI)
Decode Instruction (DI)
Calculate Operands (CO)
Fetch Operands (FO)
Execute Instruction (EI)
Write (store) Operand (WO)



(a) Five-stage pipeline, ideal case



(b) I1 source operand in memory

Source operand for instruction I1 is in memory, rather than a register → idle for one cycle (wasted)

Figure: Example of resource hazards

- Conflict of resources introduce '*pipeline delay*'.
- A delay in any stage can cause pipeline stalls.

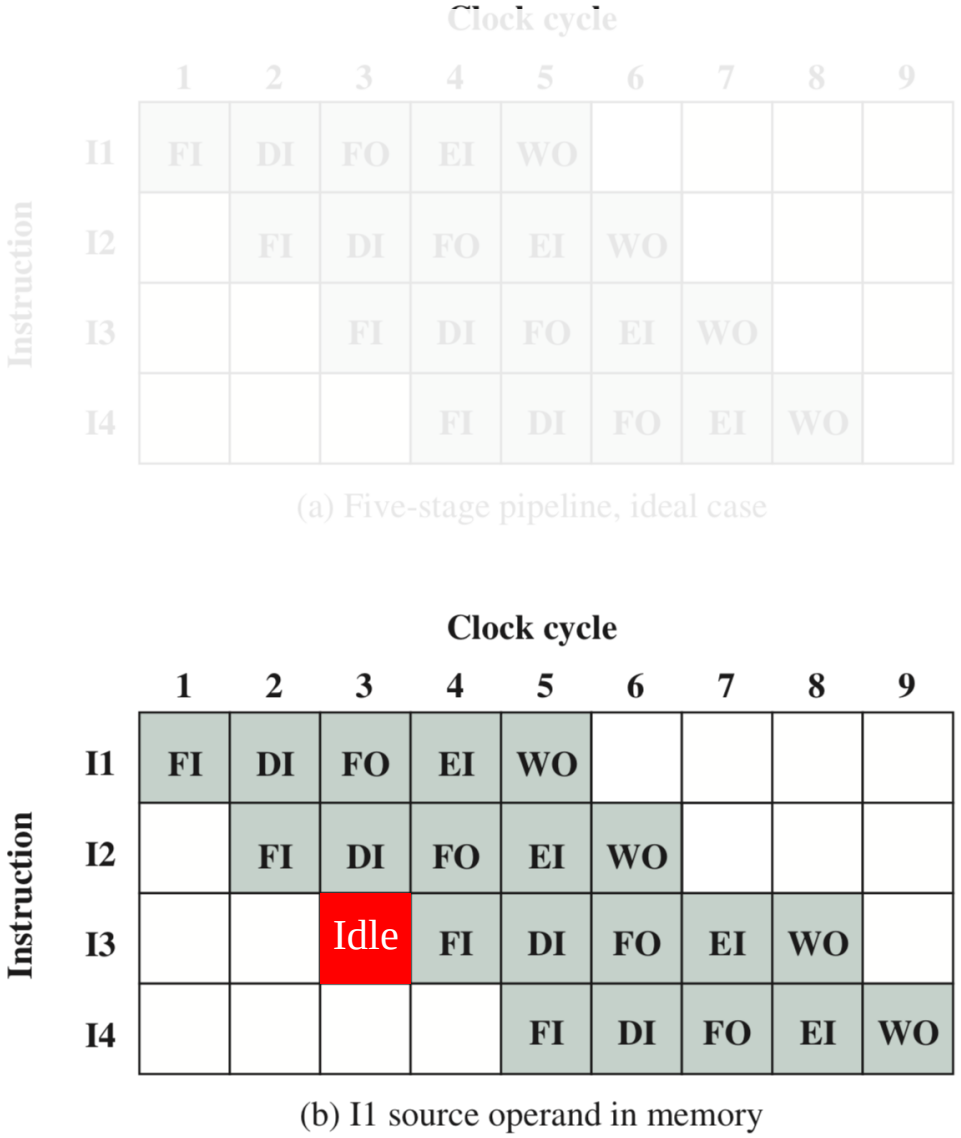


Figure: Example of resource hazards

(b) Data Hazards

(*Data Dependency*)

Example:

```
ADD  EAX, EBX   /* EAX=EAX+EBX
SUB  ECX, EAX   /* ECX=ECX–EAX
```

- A data hazard occurs when there is a conflict in the access of an operand location.
- Two instructions in a program are to be executed in sequence and both access a particular memory or register operand.

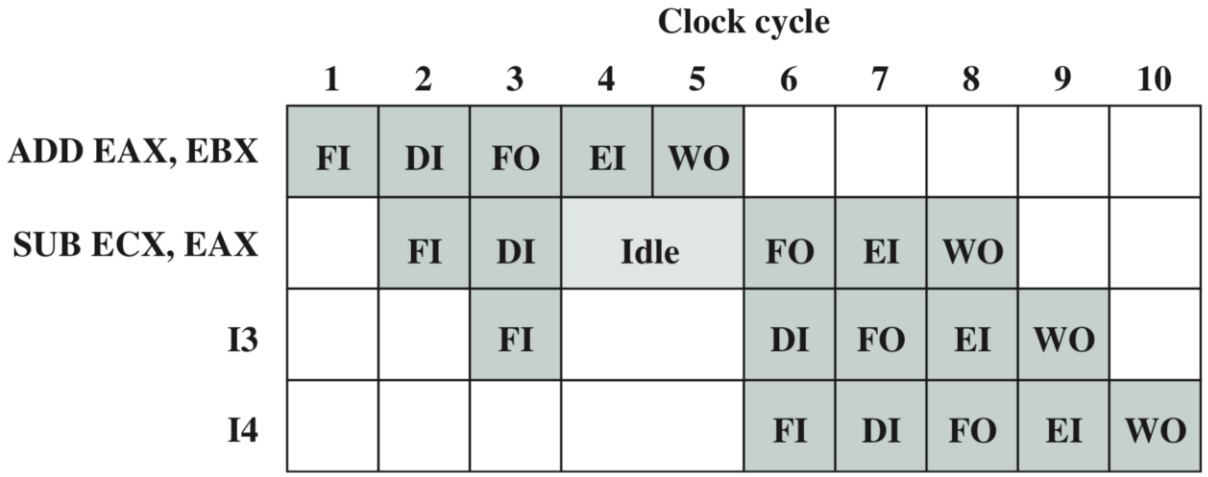


Figure: Example of data hazards

Data Hazards

Example:

```
ADD  EAX, EBX   /* EAX=EAX+EBX
SUB  ECX, EAX   /* ECX=ECX–EAX
```

The ADD *instruction does not update register* EAX *until the end of stage* 5.

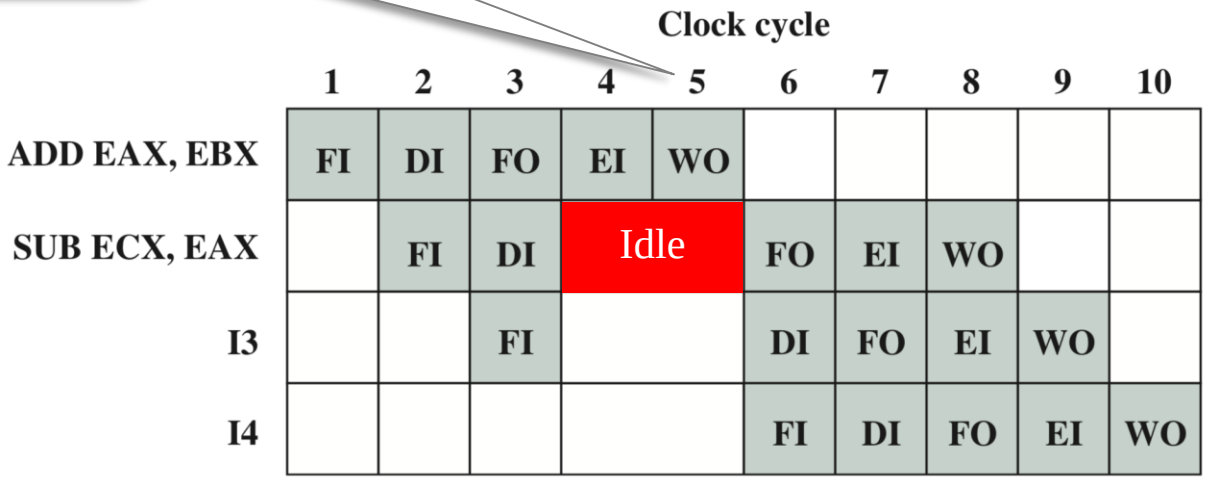


Figure: Example of data hazards

Data Hazards : Solutions

5

Hardware interlocks:

- An interlock is a circuit that detects instructions with data dependency and inserts required delays to resolve conflicts.

Operand forwarding:

- Allowing the result of ALU to be used by other ALU operations in the next instruction cycle.

Delayed load:

- Compiler is used to detect conflict and reorder instructions to delay loading of conflicting data.
- Using NOP (no operation) instruction.

I1 has finished WO (t5) then can I2 do EI (t6)

Example 14 :

I1: ADD EAX, EBX
NOP
I2: SUB ECX, EAX
I3:
I4:

Solution of data hazard using Delayed Load → NOP

	1	2	3	4	5	6	7	8	9
									→ *time*
FI	*I1*	*NOP*	*I2*	*I3*	*I4*				
DI		*I1*	*NOP*	*I2*	*I3*	*I4*			
FO			*I1*	*NOP*	*I2*	*I3*	*I4*		
EI				*I1*	*NOP*	*I2*	*I3*	*I4*	
WO					*I1*	*NOP*	*I2*	*I3*	*I4*

Example 15 :

I1: ADD AH, BH
I2: SUB AH, 2
I3: CMP AH, 0

Consider the instructions given.

Where is/are the problem(s) with the pipeline implementation? Briefly justify your answer.

	1	2	3	4	5	6	7		
FI	*I1*	*I2*	*I3*						
DI		*I1*	*I2*	*I3*					
FO			*I1*	*I2*	*I3*				
EI				*I1*	*I2*	*I3*			
WO					*I1*	*I2*	*I3*		

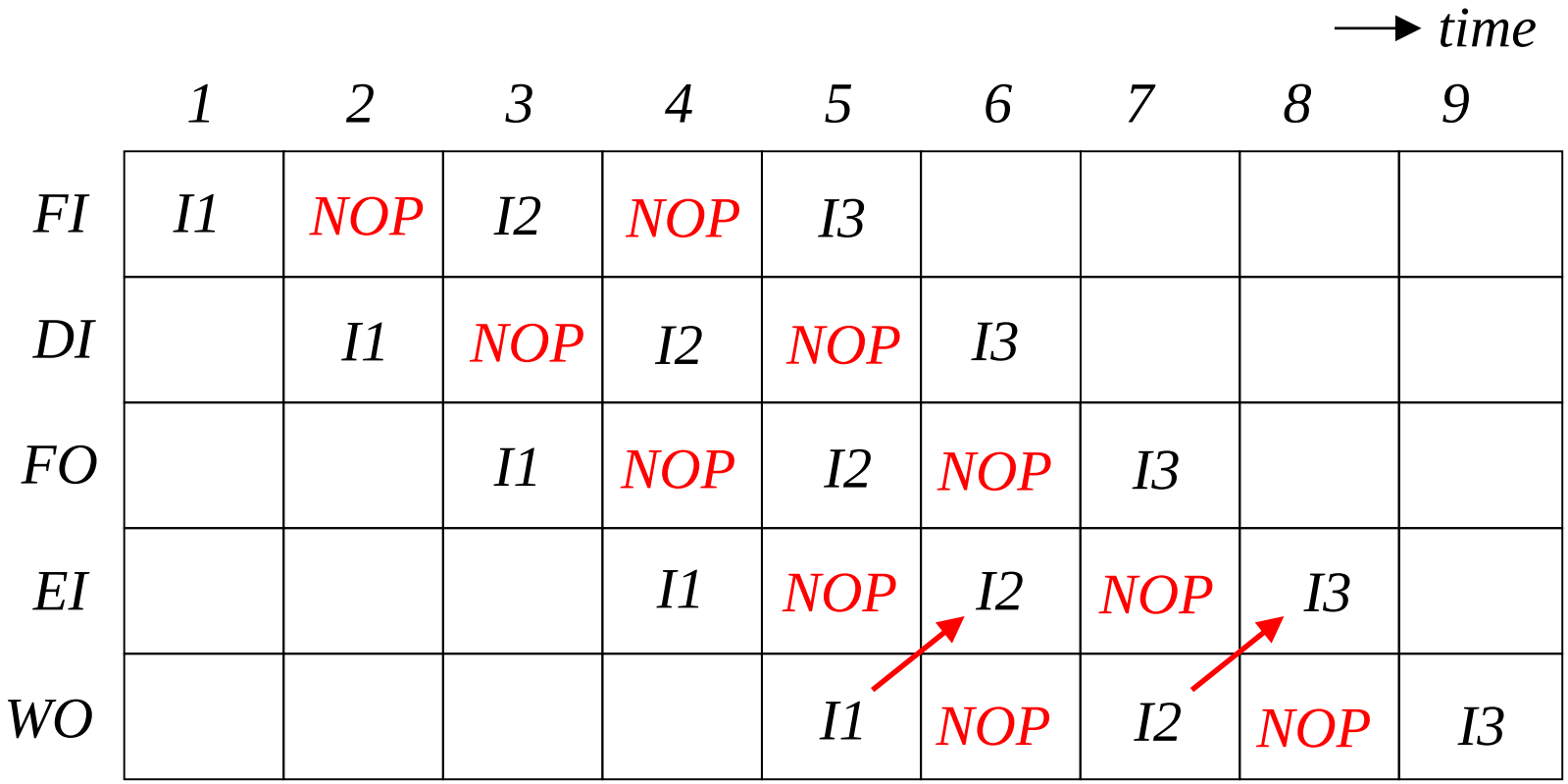
Data hazard → *Both are about to access the same register* AH.

I1 has finished WO (t5) then can I2 do EI (t6) and I2 finished WO (t7) before I3 do EI (t8)

```
I1: ADD  AH, BH
    NOP
I2: SUB  AH, 2
    NOP
I3: CMP  AH, 0
```

Solution :

Solution of data hazard using Delayed Load → NOP



(c) Control Hazards

(*Branch Difficulty*)

- Occurs when the pipeline makes the wrong decision on a branch prediction (change the PC value).
- It brings instructions into the pipeline that must subsequently be discarded.
- One of the major problems in designing an instruction pipeline is assuring a steady flow of instructions to the initial stages of the pipeline.

5

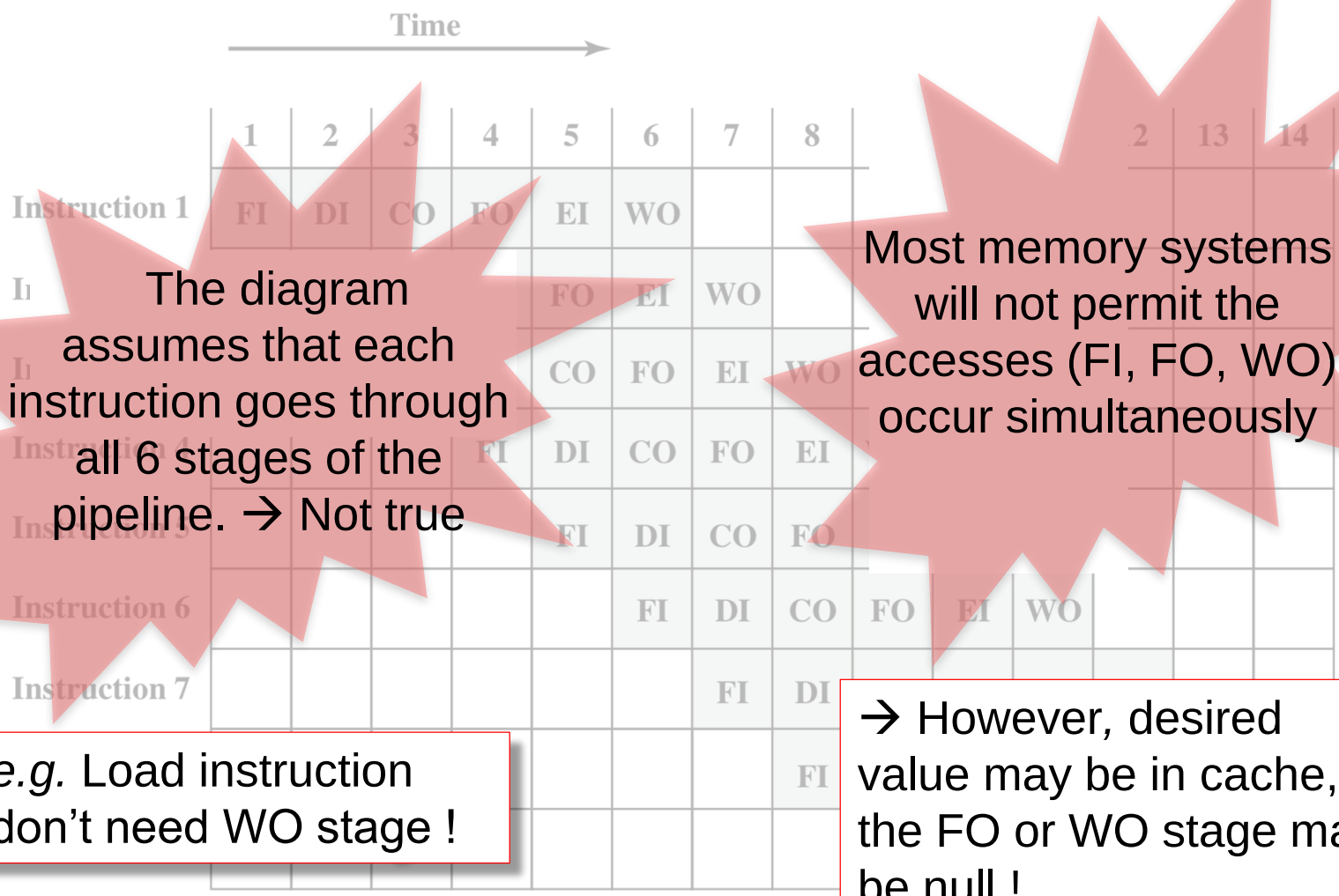
- For the pipeline to have the desired operational speedup, we must “feed it” with long strings of instructions.
- However, 15-20% of instructions in an assembly-level code are (conditional) branches.
- Of these, 60-70% take the branch to a target address.

- Impact of the branch is that pipeline never really operates at its full capacity:
  → limiting the performance improvement that is derived from the pipeline.

Example 16:

Figure shows timing diagram for 9 instruction pipeline operation with 6 stages of pipeline.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Instruction 1	FI	DI	CO	FO	EI	WO								
Instruction 2		FI	DI	CO	FO	EI	WO							
Instruction 3			FI	DI	CO	FO	EI	WO						
Instruction 4				FI	DI	CO	FO	EI	WO					
Instruction 5					FI	DI	CO	FO	EI	WO				
Instruction 6						FI	DI	CO	FO	EI	WO			
Instruction 7							FI	DI	CO	FO	EI	WO		
Instruction 8								FI	DI	CO	FO	EI	WO	
Instruction 9									FI	DI	CO	FO	EI	WO



The diagram assumes that each instruction goes through all 6 stages of the pipeline. → Not true

Most memory systems will not permit the accesses (FI, FO, WO) occur simultaneously

e.g. Load instruction don't need WO stage !

→ However, desired value may be in cache, or the FO or WO stage may be null !

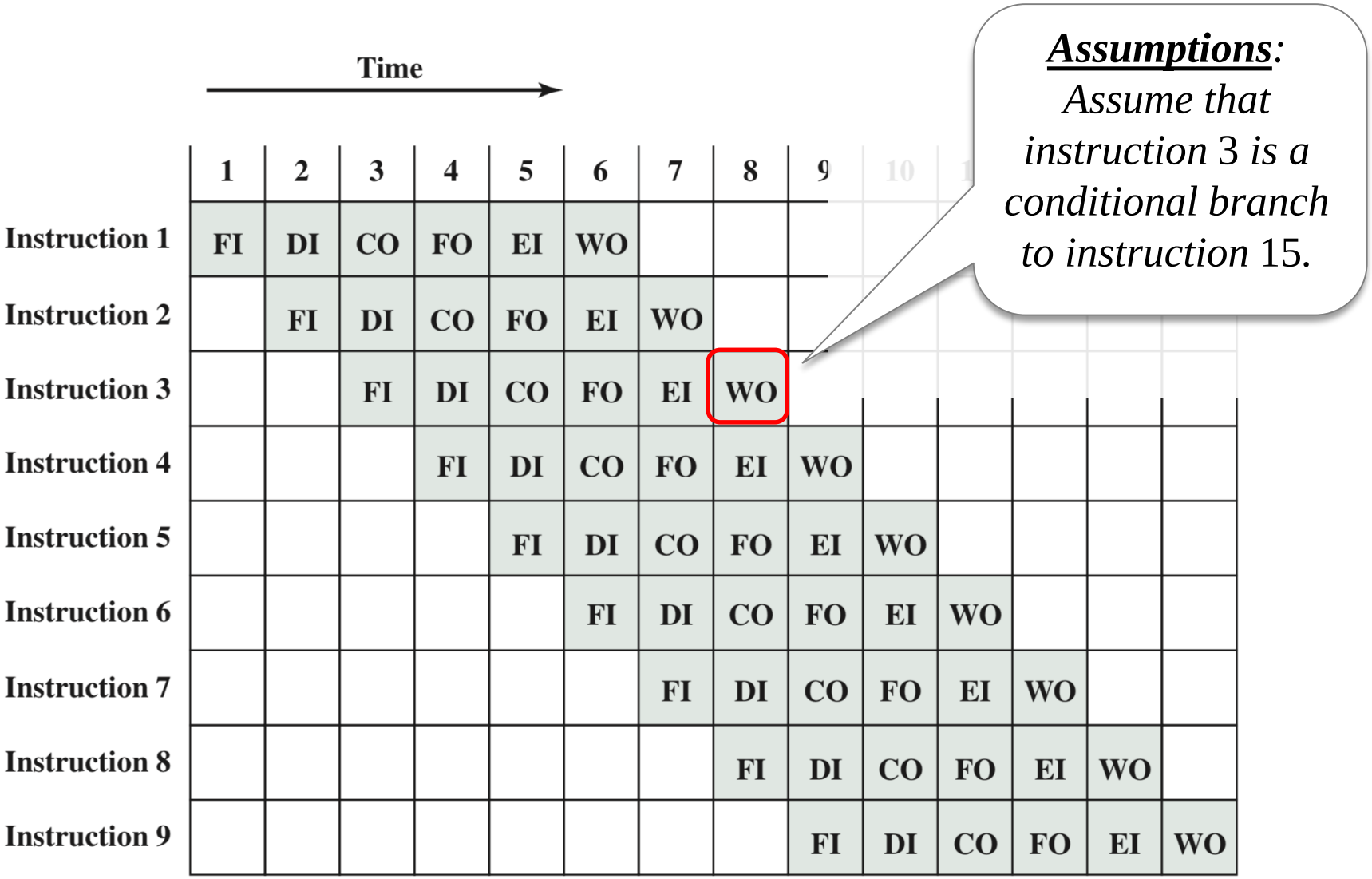
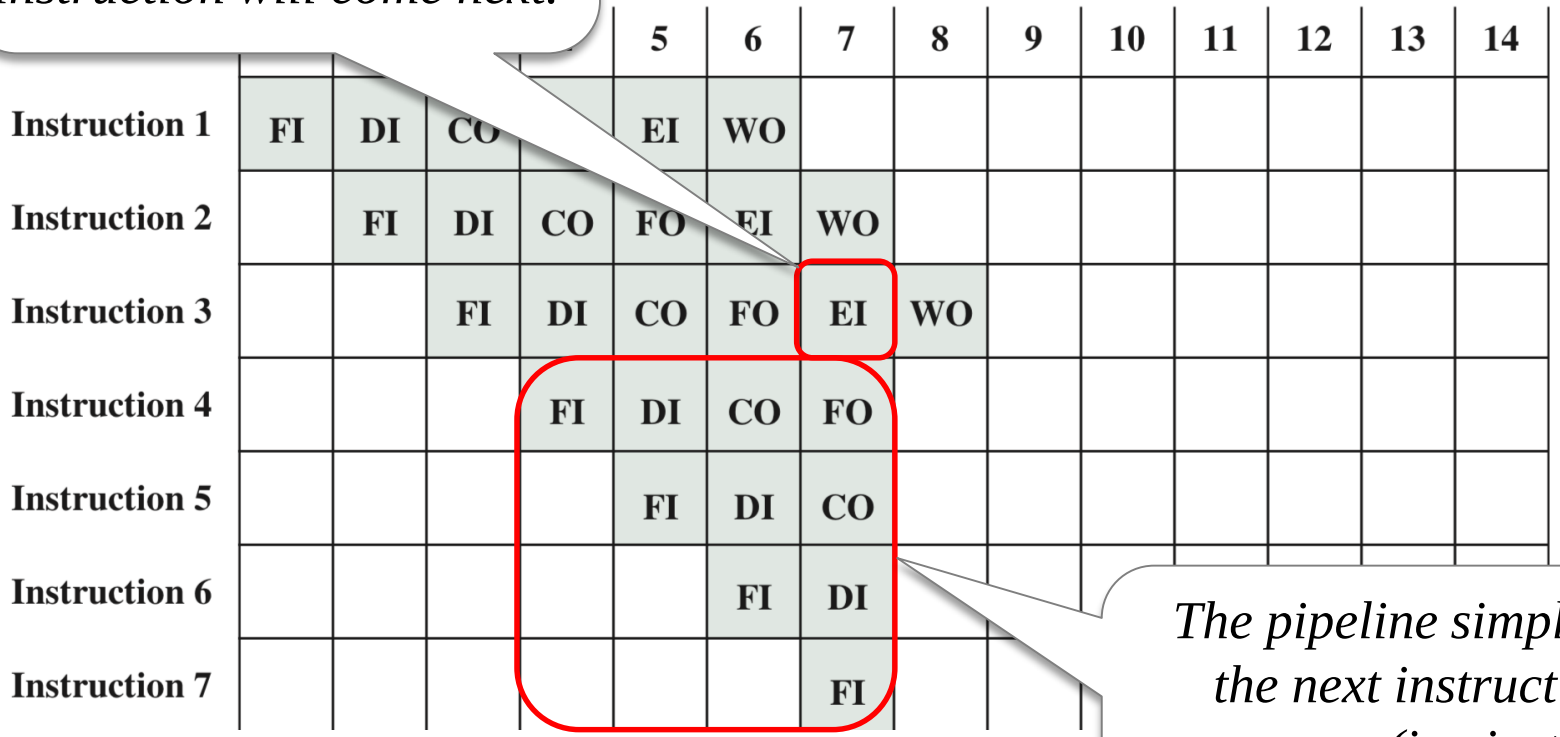


Figure: The Effect of a Conditional Branch on Instruction Pipeline Operation

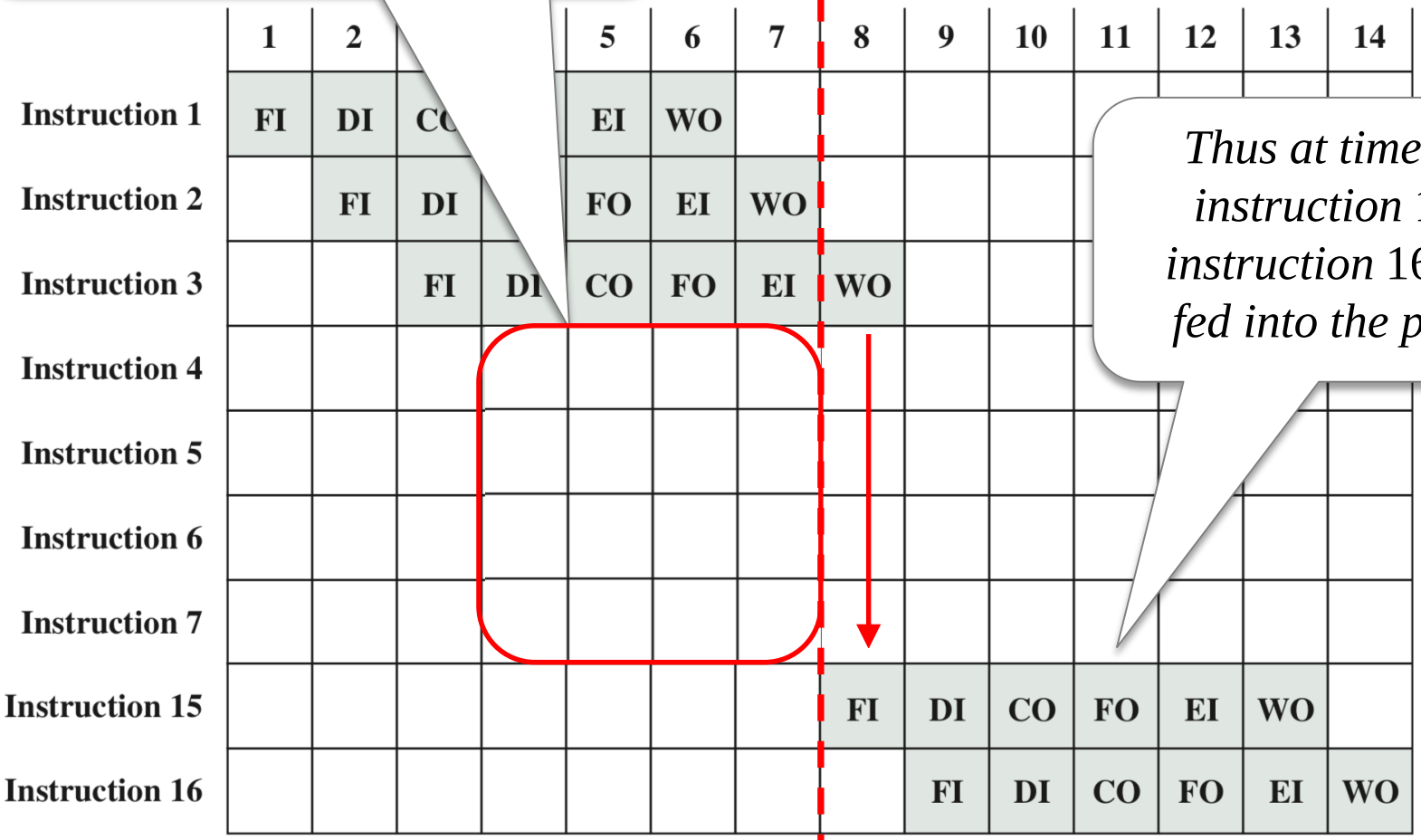
Until the instruction 3 is executed, there is no way of knowing which instruction will come next.



The pipeline simply loads the next instruction in sequence (i.e. instruction 4, 5, 6 & 7) and proceeds.

Figure: The Effect of a Conditional Branch on Instruction Pipeline Operation

Until the end of time unit 7, the instructions in the pipeline that are not useful must be cleared.



Thus at time unit 8, instruction 15 and instruction 16 will be fed into the pipeline.

Figure: The Effect of a Conditional Branch on Instruction Pipeline Operation

- *No instructions are completed during time units 9 through 12;*
- *This is the performance penalty incurred because we could not anticipate the branch.*

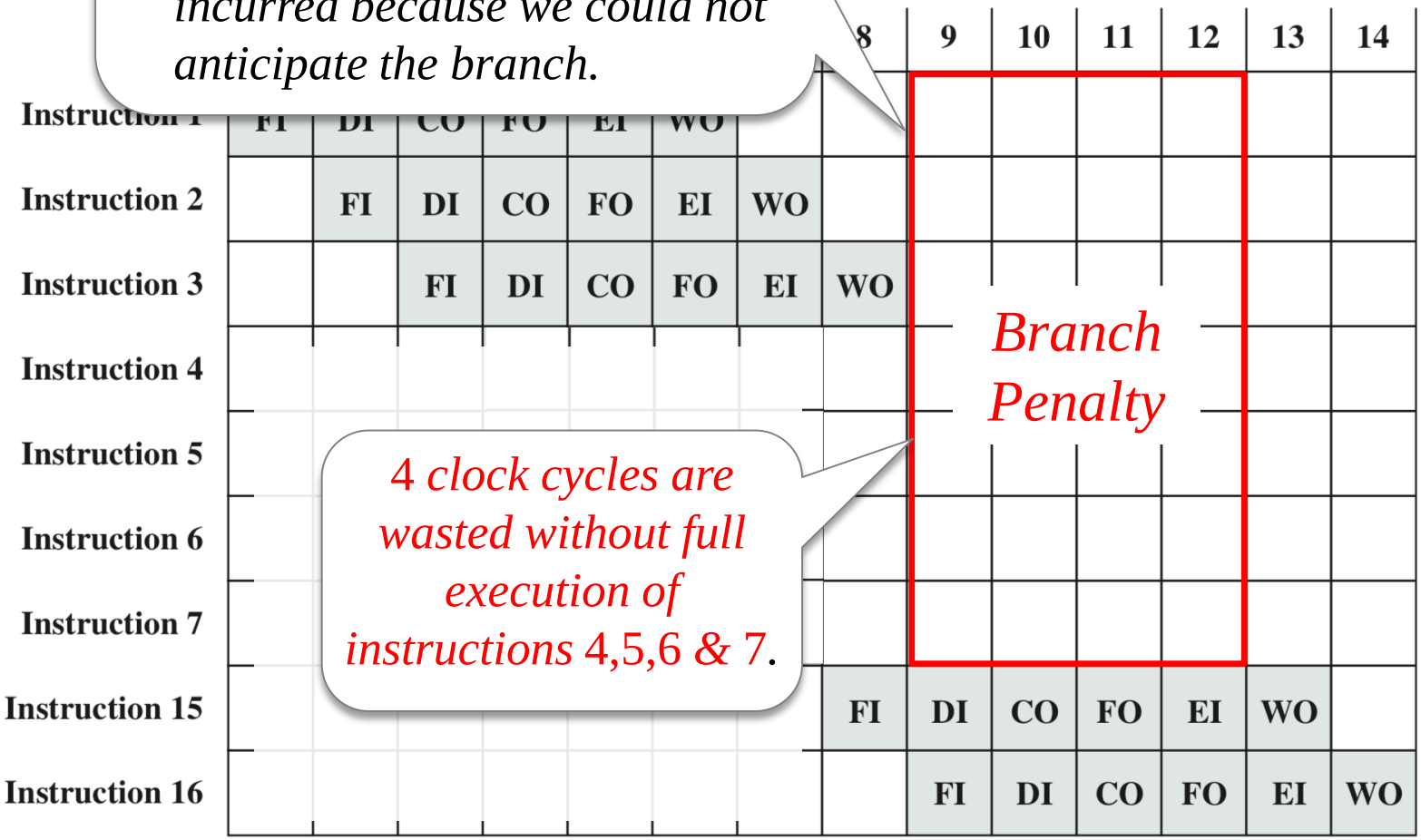


Figure: The Effect of a Conditional Branch on Instruction Pipeline Operation

	FI	DI	CO	FO	EI	WO
1	I1					
2	I2	I1				
3	I3	I2	I1			
4	I4	I3	I2	I1		
5	I5	I4	I3	I2	I1	
6	I6	I5	I4	I3	I2	I1
7	I7	I6	I5	I4	I3	I2
8	I8	I7	I6	I5	I4	I3
9	I9	I8	I7	I6	I5	I4
10		I9	I8	I7	I6	I5
11			I9	I8	I7	I6
12				I9	I8	I7
13					I9	I8
14						I9

Time

(a) No branches

	FI	DI	CO	FO	EI	WO
1	I1					
2	I2	I1				
3	I3	I2	I1			
4	I4	I3	I2	I1		
5	I5	I4	I3	I2	I1	
6	I6	I5	I4	I3	I2	I1
7	I7	I6	I5	I4	I3	I2
8	I15					I3
9	I16	I15				
10		I16	I15			
11			I16	I15		
12				I16	I15	
13					I16	I15
14						I16

(b) With conditional branch

Figure: An Alternative Pipeline Depiction

Control Hazards : Solutions

- the logic needed for pipelining to account for branches and interrupts.

  - Delayed branch (use of NOP instruction).

  - Rearranging the instructions.

Example 17 : Solution for branching using delay branch (with NOP)

When the compiler detect a branch, it will automatically insert several NOP so that there is no interruptions in the pipeline. (Assume 4 segments in a pipeline)

```
        MOV
        INC
        ADD
        RET I1
I1:
```

Solution :

When the compiler detect a branch, it will automatically insert several NOP so that there is no interruptions in the pipeline. (Assume 4 segments in a pipeline)

How many NOP to insert?

$$= No\ of\ segment - 1$$
$$= k - 1$$
$$= 4 - 1 = 3$$

```
        MOV
        INC
        ADD
        RET I1
        NOP
        NOP
        NOP
I1:
```

(Add 3 NOP)

Illustrate with space-time diagram.

Wasted 3 *pipeline clock cycles*

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*
FI	MOV	INC	ADD	RET	NOP	NOP	NOP	I1	I2	I3			
DI		MOV	INC	ADD	RET	NOP	NOP	NOP	I1	I2	I3		
EI			MOV	INC	ADD	RET	NOP	NOP	NOP	I1	I2	I3	
WO				MOV	INC	ADD	RET	NOP	NOP	NOP	I1	I2	I3

Example 18 : Solution for branching using rearranging instructions.

When the compiler detect a branch, it will rearrange the instructions so that there is no wasted pipeline.

```
        MOV
        INC
        ADD
        RET I1
I1:
```


```
        RET I1
        MOV
        INC
        ADD
I1:
```

$$= No\ of\ segment - 1$$

$$= k - 1$$

$$= 4 - 1 = 3 \quad \text{(move up 3)}$$

5

Illustrate with space-time diagram.

No more wasted pipeline !

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*
FI	RET	MOV	INC	ADD	I1	I2	I3						
DI		RET	MOV	INC	ADD	I1	I2	I3					
EI			RET	MOV	INC	ADD	I1	I2	I3				
WO				RET	MOV	INC	ADD	I1	I2	I3			

Activity 3

Example 19 :

Consider 4 segments involved in a pipeline: *Fetch Instruction* (FI), *Decode Instruction* (DI), *Execute Instruction* (EI) and *Write Operand* (WO).

Based on the instructions given, solve the branching problem using:

(a) Delay branch (with NOP).

(b) Rearrange instructions.

```
     I1:          MOV   AX, BX
     I2:          ADD   BX, CX
     I3:          INC   CX
     I4:          JMP   LOSS
     I5:          MOV   DX, 1
     I6:          INC   BX
     . . .
     I12: LOSS    DEC   AX
```

Answer a) and b) by
- Showing the NOP in the segment code
- Showing the illustration of space-time diagram

Solution (a):

Solution for branching using delay branch (with NOP)

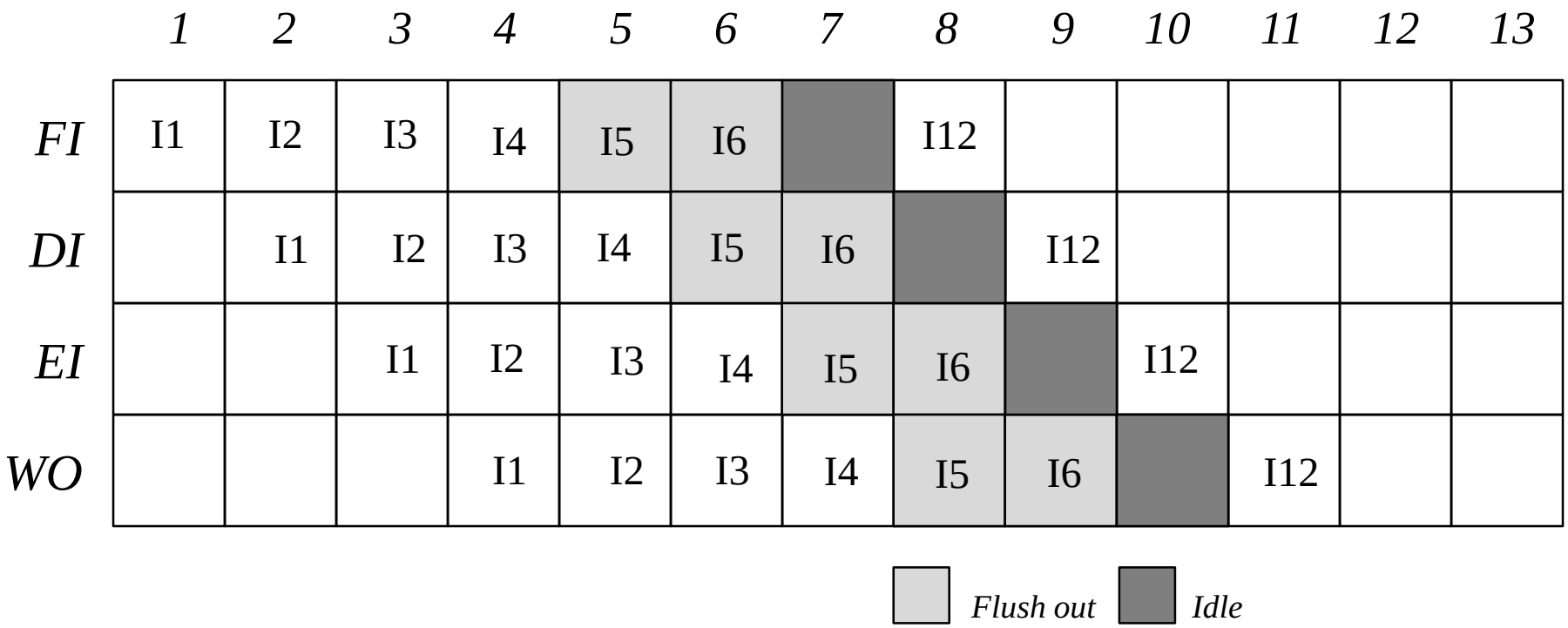
```
I1:          MOV   AX, BX
I2:          ADD   BX, CX
I3:          INC   CX
I4:          JMP   LOSS
             NOP
             NOP
             NOP
I5:          MOV   DX, 1
I6:          INC   BX
. . .
I12: LOSS    DEC   AX
```

(Add 3 NOP)

Total number of NOP to be inserted:

$$= No\ of\ segment - 1$$
$$= k - 1$$
$$= 4 - 1 = 3$$

Illustrate with space-time diagram for (a).



Illustrate with space-time diagram for (a).

Create 3 *NOP instruction cycle before targeted label.*

	1	2	3	4	5	6	7	8	9	10	11	12	13
FI	I1	I2	I3	I4	NOP	NOP	NOP	I12					
DI		I1	I2	I3	I4	NOP	NOP	NOP	I12				
EI			I1	I2	I3	I4	NOP	NOP	NOP	I12			
WO				I1	I2	I3	I4	NOP	NOP	NOP	I12		

Wasted 3 *pipeline clock cycles*

Summary 2

- Instruction pipelining is a powerful technique for enhancing performance but requires careful design to achieve optimum results with reasonable complexity.
- Discussed the principle behind instruction pipelining and how it works in practice.
- Compared and contrasted the various forms of pipeline hazards:
  - *resource conflict*,
  - *data dependency*,
  - *control* or *branch difficulty*.