IFT 307
Computer Organization and Architecture

Mr. Ibrahim Lawal

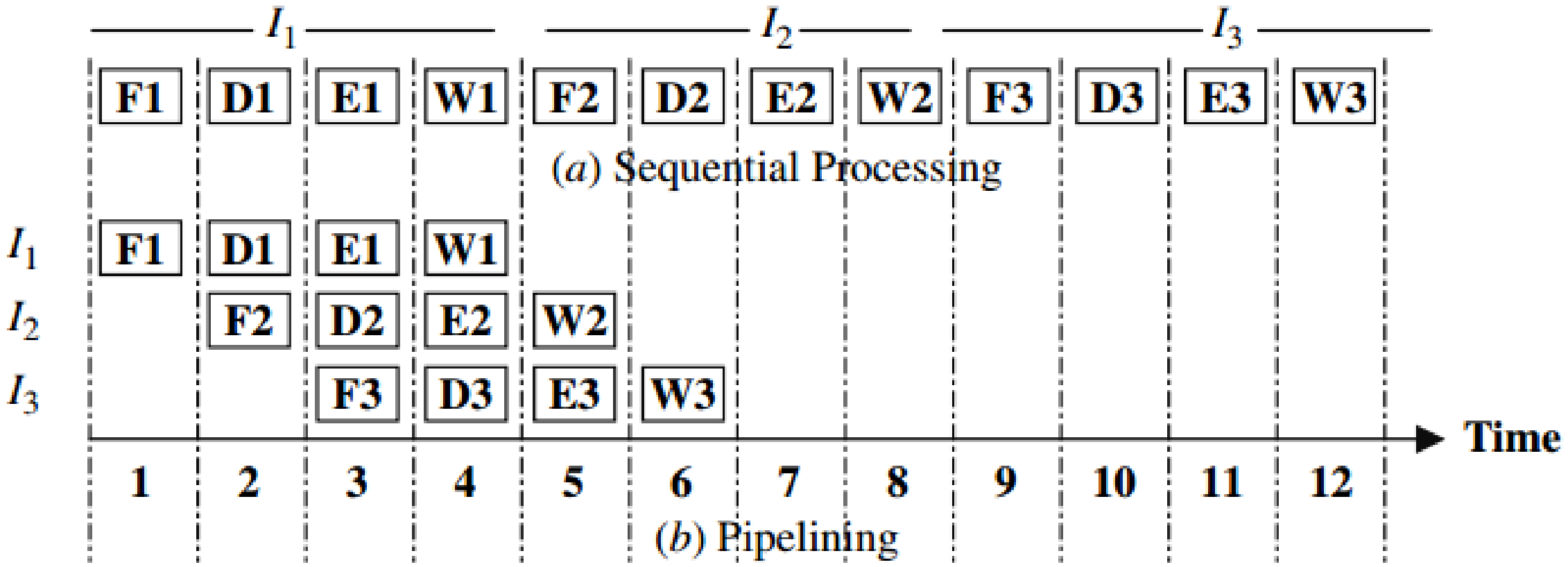
Pipelining Design Techniques

General Concepts

- Pipelining refers to the technique in which a given task is divided into a number of subtasks that need to be performed in sequence.
- Each subtask is performed by a given functional unit.

- The units are connected in a serial fashion and all of them operate simultaneously.
- The use of Pipelining improves the performance as compared to the traditional sequential execution of tasks.

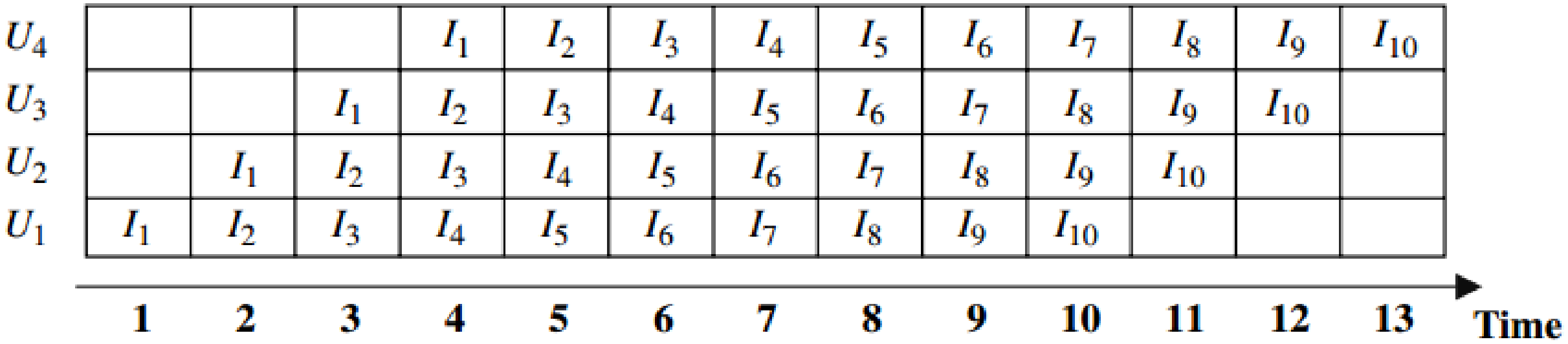
• Below is an illustration of the basic difference between executing four subtasks of a given instruction (in this case **fetching F, decoding D, execution E, and writing the results W**) using pipelining and sequential processing.



Pipelining versus sequential processing

- It is clear that the total time required to process three instructions (I_1, I_2, I_3) is only six time units if four-stage pipelining is used as compared to I_2 time units if sequential processing is used.
- A possible saving of up to 50% in the execution time of these three instructions is obtained.

- In order to formulate some performance measures for the goodness of a pipeline in processing a series of tasks, a space time chart (called the Gantt's Chart) is used.
- The chart shows the succession of the sub-tasks in the pipe with respect to time.



The space –time chart (Gantt chart)

Three performance measures for the goodness of a pipeline are provided:

– Speed-up S(n),

– Throughput U(n), and

– Efficiency E(n).

It is assumed that the unit time T = t units.

• Speedup S(n): – Consider the execution of m tasks (instructions) using n-stages (units) pipeline.

n+m-1 time units are required to complete m tasks.

$$Speed - up\ \mathrm{S(n)} = \frac{\text{Time using sequential processing}}{\text{Time using pipeline processing}}$$

$$Throughput\ U(n) = \#\ \text{of tasks executed per unit time} = \frac{\text{m}}{\text{Time for pipeline}}$$

- Efficiency E(n):

$$Efficiency\ E(n) = \text{Ratio of the actual speed – up to the maximum speed – up} = \frac{Speed - up}{n}$$

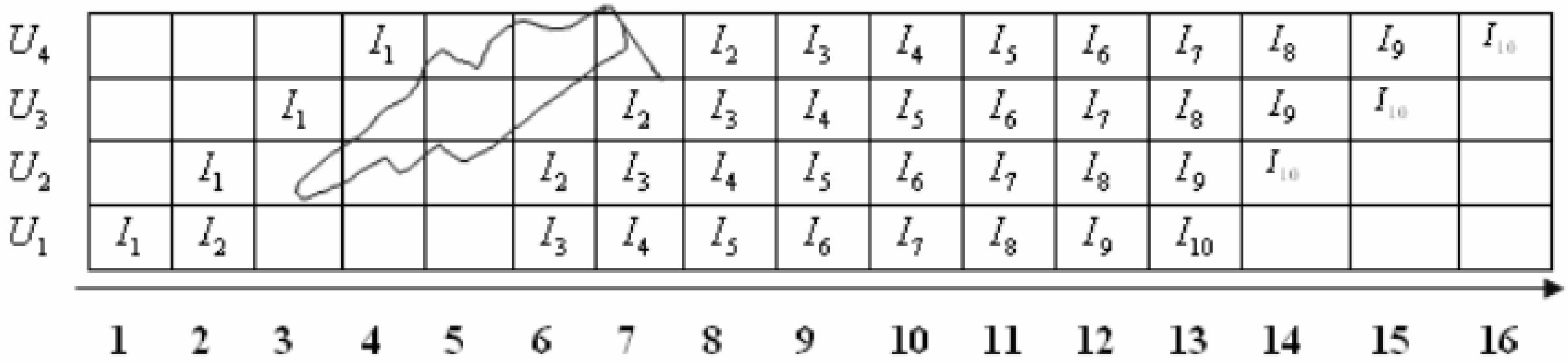
Instruction Pipeline

• A pipeline stall: A Pipeline operation is said to have been stalled if one unit (stage) requires more time to perform its function, thus forcing other stages to become idle.

- Due to the extra time units needed for instruction to be fetched, the pipeline stalls.
- Such situations create what is known as pipeline bubble (or pipeline hazards).

Consider, for example, the case of an instruction fetch that incurs a cache miss. Assume also that a cache miss requires three extra time units.

Figure below illustrates the effect of having instruction I_2 incurring a cache miss (assuming the execution of ten instructions I_1 to I_{10}).



Effect of a cache miss on the pipeline

- The figure above shows that due to the extra time units needed for instruction I_2 to be fetched, the pipeline stalls, that is, fetching of instruction I_2 and subsequent instructions are delayed.

- The creation of a pipeline bubble leads to wasted unit times, thus leading to an overall increase in the number of time units needed to finish executing a given number of instructions

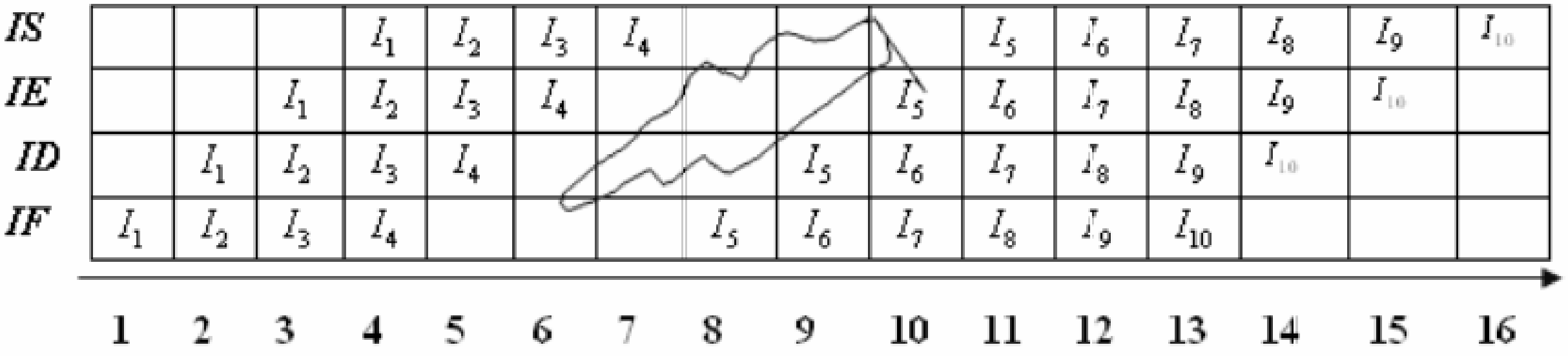
Pipeline “Stall” due to Instruction Dependency:

Correct operation of a pipeline requires that operation performed by a stage MUST NOT depend on the operation(s) performed by other stage(s).

– **Instruction dependency** refers to the case whereby fetching of an instruction depends on the results of executing a previous instruction.
– Instruction dependency manifests itself in the execution of a conditional branch instruction.

– For example, in the case of a "branch if negative“ instruction, the next instruction to fetch will not be known until the result of executing that “branch if negative” instruction is known.

- **Example 1** Consider the execution of ten instructions I_1 –I_{10}on a pipeline consisting of four pipeline stages: IF (instruction fetch), ID (instruction decode), IE (instruction execute), and IS (instruction results store).



Instruction dependency effect on a pipeline

- Assume that the instruction I_4 is a conditional branch instruction and that when it is executed, the branch is not taken, that is, the branch condition(s) is(are) not satisfied.

- Assume also that when the branch instruction is fetched, the pipeline stalls until the result of executing the branch instruction is stored.

- **Pipeline “Stall” due to Data Dependency:** – Data dependency in a pipeline occurs when a source operand of instruction depends on the results of executing a preceding instruction, , $i > j$.

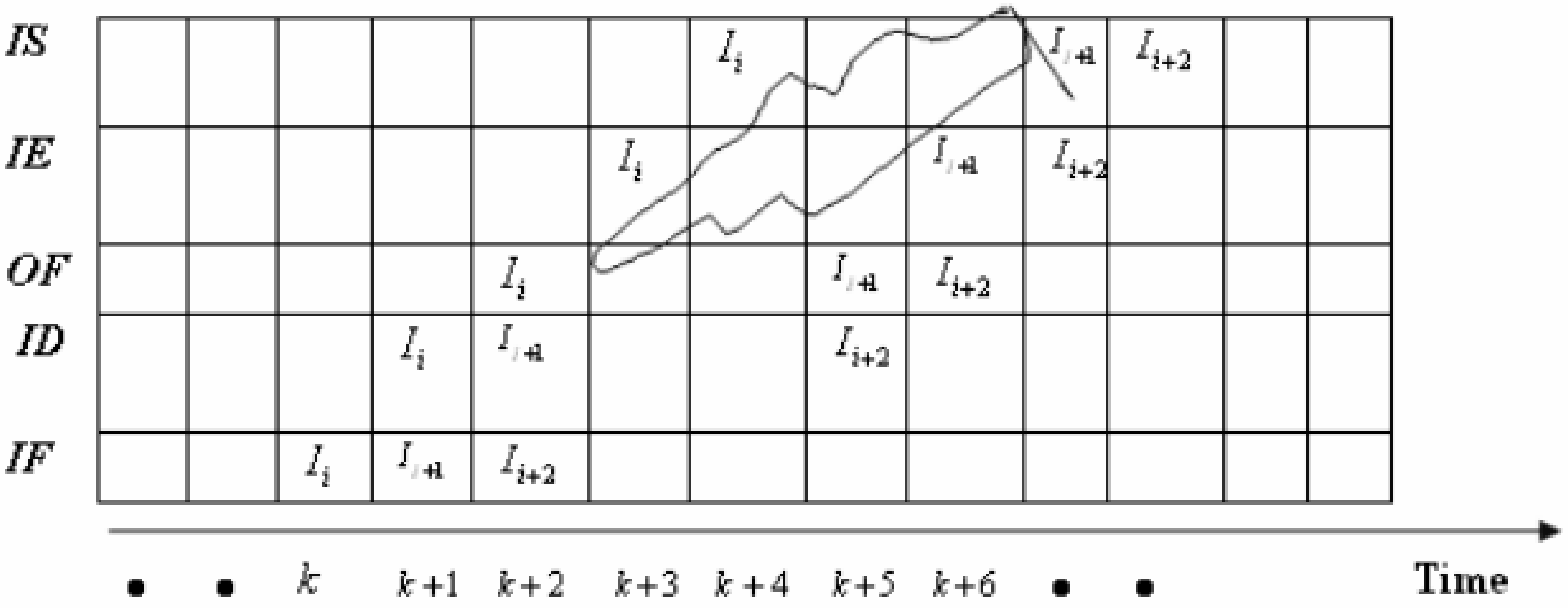
Example 2 Consider the execution of the following piece of code:

ADD	$R_1, R_2, R_3;$	$R_3 \leftarrow R_1 + R_2$
SL	$R_3;$	$R_3 \leftarrow SL(R_3)$
SUB	$R_5, R_6, R_4;$	$R_4 \leftarrow R_5 - R_6$

- In this piece of code, the first instruction, call it I_i, adds the contents of two registers R_1 and R_2 and stores the result in register R_3.
- The second instruction, call it I_{i+1} , shifts the contents of R_3 one bit position to the left and stores the result back into R_3.

- The third instruction, call it I_{i+2} , stores the result of subtracting the content of R_6 from the content of R_5 in register R_4.
- In order to show the effect of such data dependency, we will assume that the pipeline consists of five stages, IF, ID, OF, IE, and IS.

- In this case, the OF stage represents the operand fetch stage.
- The functions of the remaining four stages remain the same as explained before.
- Figure below shows the Gantt's chart for this piece of code. As shown in the figure,

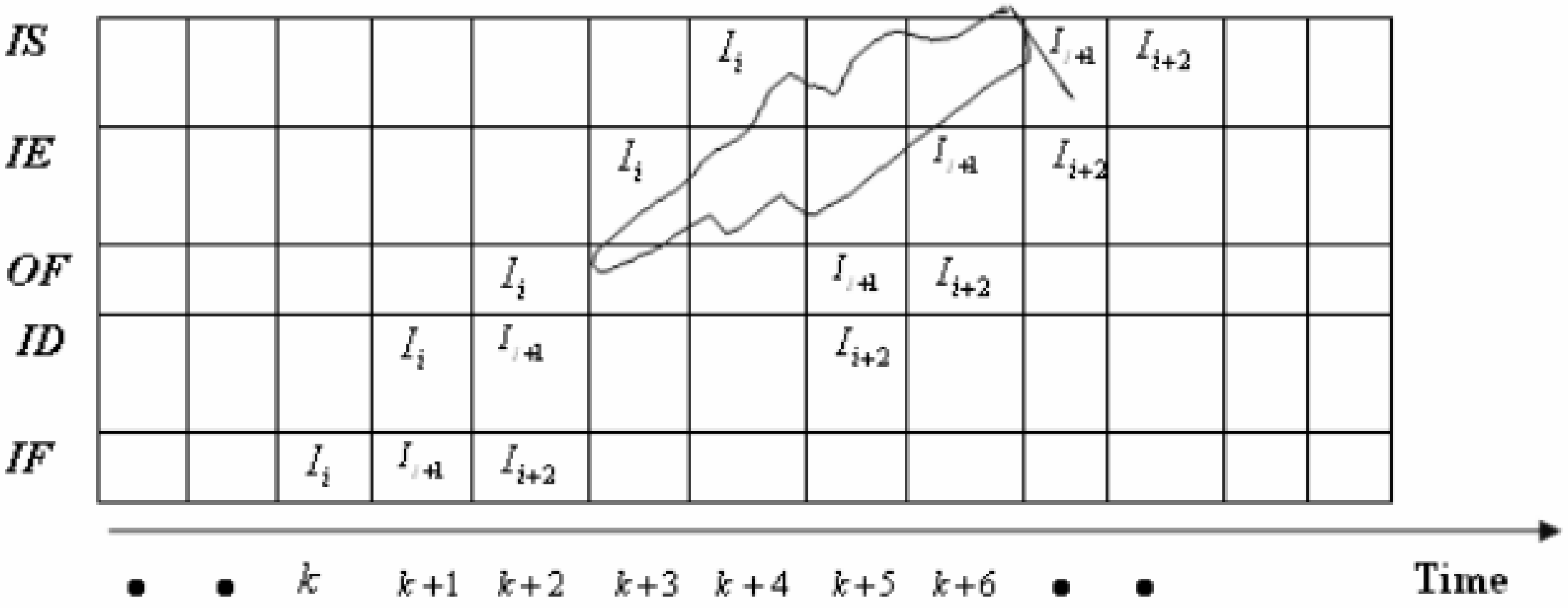


The write-after-write data dependency

- As shown in the figure, although instruction I_{i+1} has been successfully decoded during time unit k + 2, this instruction cannot proceed to the OF unit during time unit k +3. This is because the operand to be fetched by I_{i+1} during time unit k+3 should be the content of register R_3, which has been modified by execution of instruction I_i.

However, the modified value of R_3 will not be available until the end of time unit k+4. This will require instruction I_{i+1} to wait (at the output of the ID unit) until k+5. Notice that instruction I_{i+2} will have also to wait (at the output of the IF unit) until such time that instruction I_{i+1} proceeds to the ID.

- The net result is that pipeline stall takes place due to the data dependency that exists between instruction I_i and instruction I_{i+1}.
- The data dependency presented in the above example resulted because register R_3 is the destination for both instructions I_i and I_{i+1}.



The write-after-write data dependency

- This is called a **write-after-write** data dependency. Taking into consideration that any register can be written into (or read from), then a total of four different possibilities exist, including the write-after-write case.

The other three cases are **read-after-write**, **write-after-read**, and **read-after-read**. Among the four cases, the read-after-read case should not lead to pipeline stall. This is because a register read operation does not change the content of the register.

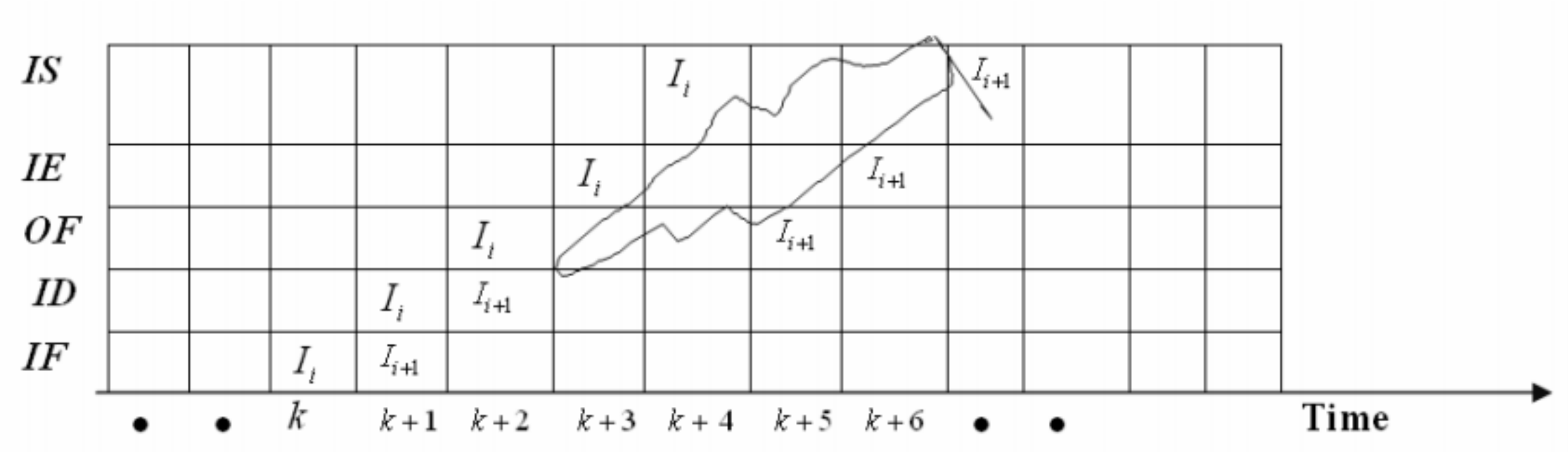
- Among the remaining three cases, the write-after-write (see the above example) and the read-after-write lead to pipeline stall.
- The following piece of code illustrates the read-after-write case:

ADD	$R_1, R_2, R_3;$	$R_3 \leftarrow R_1 + R_2$
SUB	$R_3, 1, R_4;$	$R_4 \leftarrow R_3 - 1$

In this case, the first instruction modifies the content of register R_3 (through a write operation) while the second instruction uses the modified contents of R_3 (through a read operation) to load a value into register R_4.

- The figure shows the Gantt's chart for this case assuming that the first instruction is called I_i and the second instruction is called I_{i+1}.
- It is clear that the operand of the second instruction cannot be fetched during time unit k+3 and that it has to be delayed until time unit k+5.

- This is because the modified value of the content of register R_3 will not be available until time slot k + 5

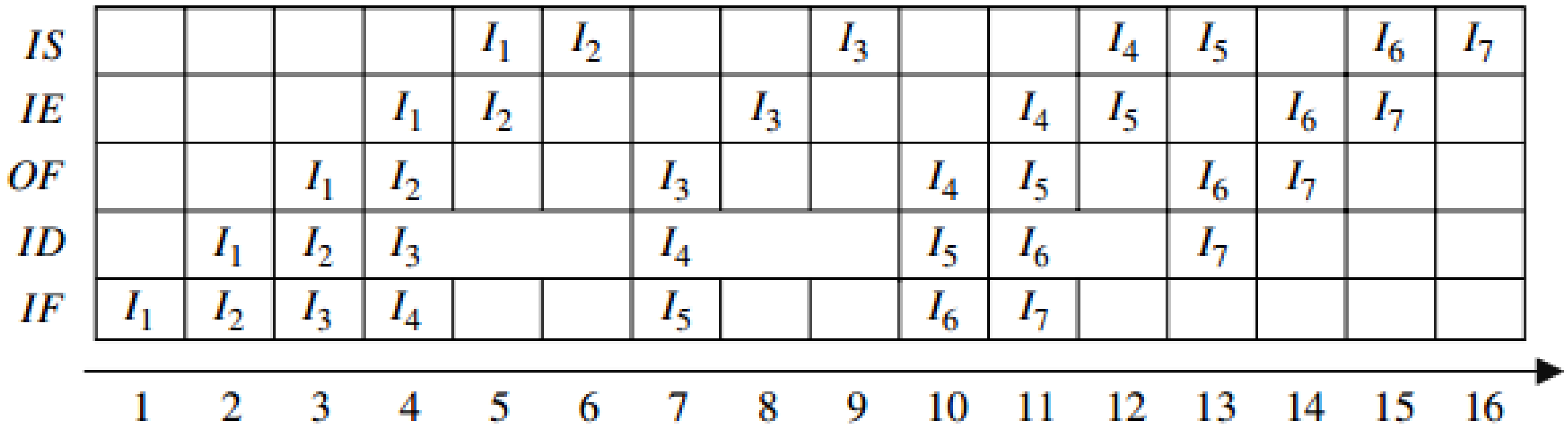


The read-after-write data dependency

Fetching the operand of the second instruction during time slot k+3 will lead to incorrect results

Example 3 Consider the execution of the following sequence of instructions on a five-stage pipeline consisting of IF, ID, OF, IE, and IS. It is required to show the succession of these instructions in the pipeline.

$I_1 \rightarrow$	Load	$-1, R1;$	$R1 \leftarrow -1;$
$I_2 \rightarrow$	Load	$5, R2;$	$R2 \leftarrow 5;$
$I_3 \rightarrow$	Sub	$R2, 1, R2$	$R2 \leftarrow R2 - 1;$
$I_4 \rightarrow$	Add	$R1, R2, R3;$	$R3 \leftarrow R1 + R2;$
$I_5 \rightarrow$	Add	$R4, R5, R6;$	$R6 \leftarrow R4 + R5;$
$I_6 \rightarrow$	SL	$R3$	$R3 \leftarrow SL\ (R3)$
$I_7 \rightarrow$	Add	$R6, R4, R7;$	$R7 \leftarrow R4 + R6;$


IS
IE
OF
ID
IF
I1 I2 I3 I4 I5 I6 I7
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Gantt's chart for Example 3

Based on the results obtained above, we can compute the speed-up and the throughput for executing the piece of code given in Example 3 as:

$$\textit{Speed-up } S(5) = \frac{\text{Time using sequential processing}}{\text{Time using pipeline processing}} = \frac{7 \times 5}{16} = 2.19$$

$$\textit{Throughput } U(5) = \text{No. of tasks executed per unit time} = \frac{7}{16} = 0.44$$

The discussion on pipeline stall due to instruction and data dependencies should reveal three main points about the problems associated with having such dependencies.

These are: 1. Both instruction and data dependencies lead to added delay in the pipeline.

2. Instruction dependency can lead to the fetching of the wrong instruction.

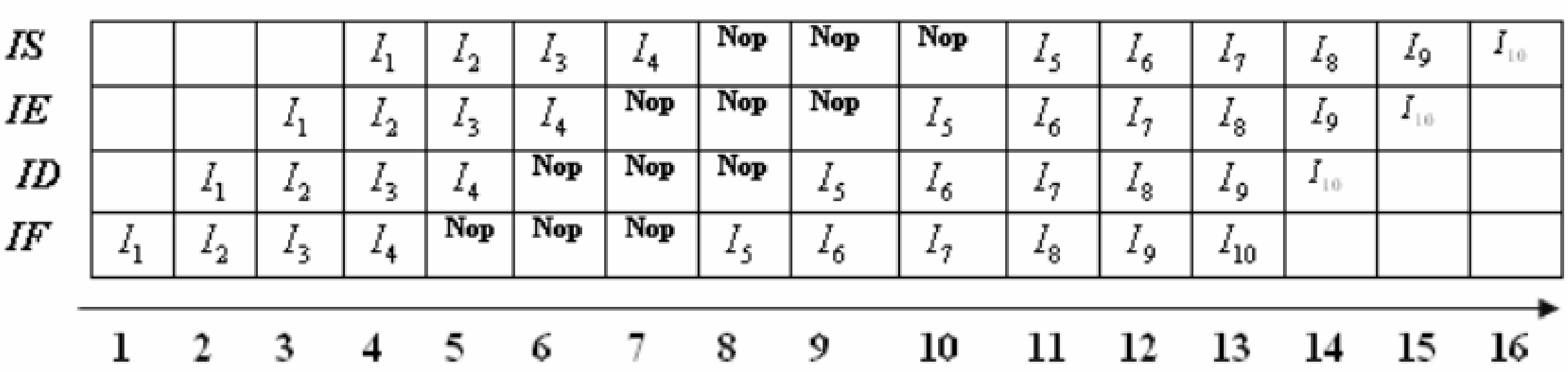
3. Data dependency can lead to the fetching of the wrong operand.

Method used to prevent fetching the wrong instruction or operand: use of NOP (No Operation)

– In real-life situations, a mechanism is needed to guarantee fetching the appropriate instruction at the appropriate time.

– Insertion of “NOP” instructions will help carrying out this task.

– A "NOP" is an instruction that has no effect on the status of the processor. Method used to prevent fetching the wrong instruction or operand: use of NOP (No Operation)


IS
IE
ID
IF
I1 I2 I3 I4 Nop Nop Nop I5 I6 I7 I8 I9 I10
I1 I2 I3 I4 Nop Nop Nop I5 I6 I7 I8 I9 I10
I1 I2 I3 I4 Nop Nop Nop I5 I6 I7 I8 I9 I10
I1 I2 I3 I4 Nop Nop Nop I5 I6 I7 I8 I9 I10
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Methods used to reduce pipeline stall due to instruction dependency:

– Unconditional Branch Instructions

- Reordering of Instructions.
- Use of Dedicated Hardware in the Fetch Unit.
- Pre-computing of Branches and Reordering of Instructions.
- Instruction Pre-fetching.

– Conditional Branch Instructions

- Delayed Branch.
- Prediction of the Next Instruction to Fetch.

- **Methods used to reduce pipeline stall due to data dependency**
  – Hardware Operand Forwarding
  – Software Operand Forwarding
- Store-Fetch.
- Fetch-Fetch.
- Store-Store.

THANK YOU