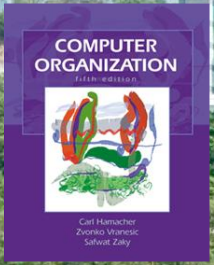
香港中文大學
The Chinese University of Hong Kong

# *CSCI2510 Computer Organization*
# Lecture 11: Pipelining

**Ming-Chang YANG**

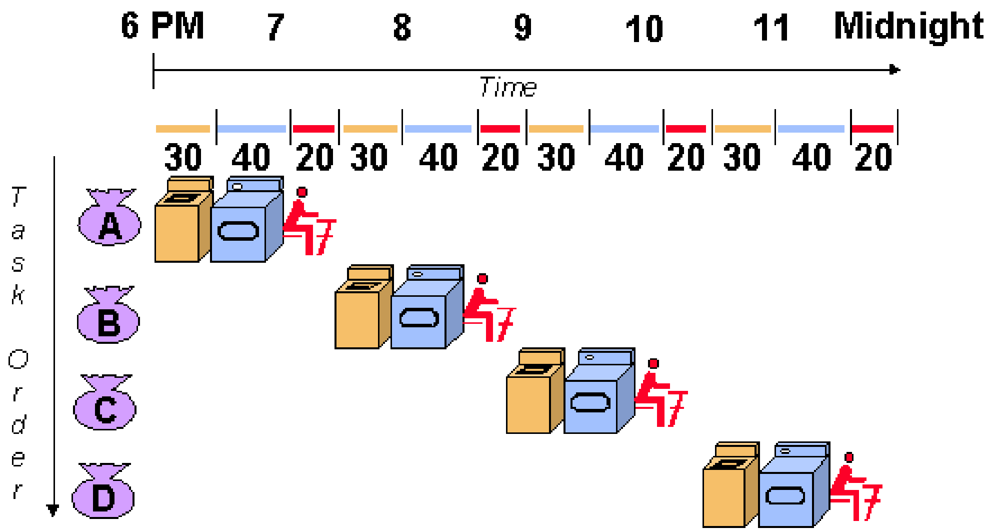
*mcyang@cse.cuhk.edu.hk*

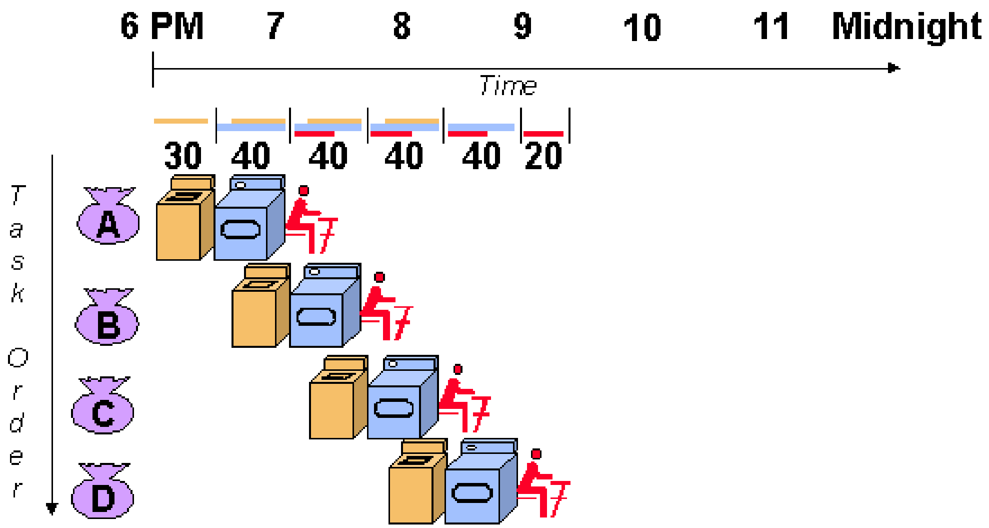


*Reading: Chap. 8 (5th Ed.)*

# Why Do We Need Pipelining?

- Real-life Example: Four loads of laundry that need to be washed, dried, and folded.
  - Washing: 30 minutes
  - Drying: 40 minutes
  - Folding: 20 minutes
- **Without** pipeline:
  - $(30+40+20)*4=$ 360 minutes in total
- **With** pipeline:
  - $30+40*4+20=$ 210 minutes in total





https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/pipelining/index.html

## Outline

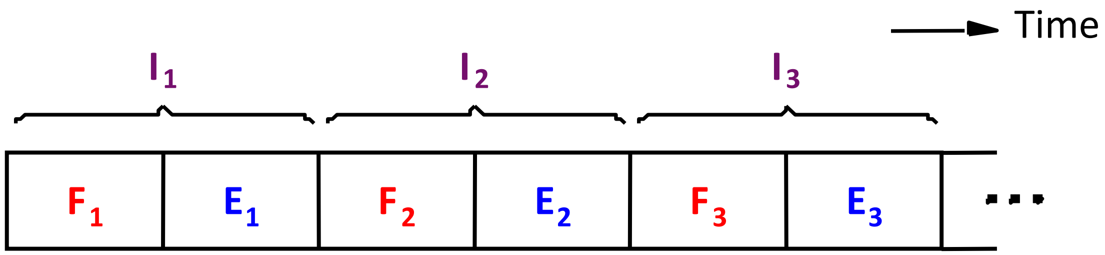
- Sequential Execution vs Pipelining
- Pipeline Stall: Hazard
  - Data Hazard
  - Instruction Hazard
  - Structural Hazard
- Superscalar and Out-of-Order Execution

# Recall: Sequential Execution

- The processor fetches and executes instructions, one after the other.
  - **$F_i$**: Fetch steps for instruction **$I_i$**
  - **$E_i$**: Execute steps for instruction $I_i$

***PC**: contains the memory address of the next instruction to be fetched.*
***IR**: holds the instruction that is currently being executed.*

- Execution of a program consists of a sequential sequence of fetch and execute steps:



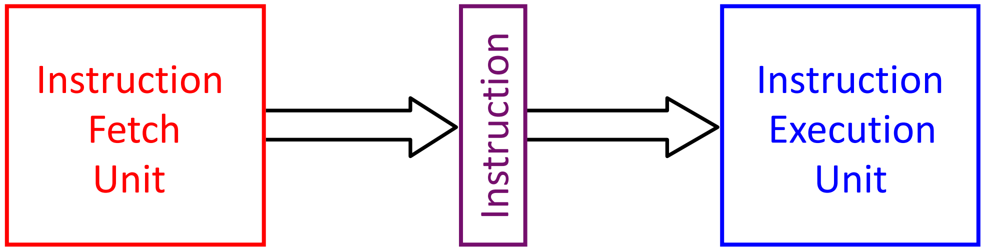
- How to improve the speed of execution?
  - Use faster technologies to build CPU and memory ($$$).
  - **Arrange hardware to perform multiple tasks at a time ($).**

# Separate HW & Interstage Buffer

- Consider a computer having two **separate hardware units**:
  - One hardware unit is for fetching instructions.
  - The other hardware unit is for executing instructions.

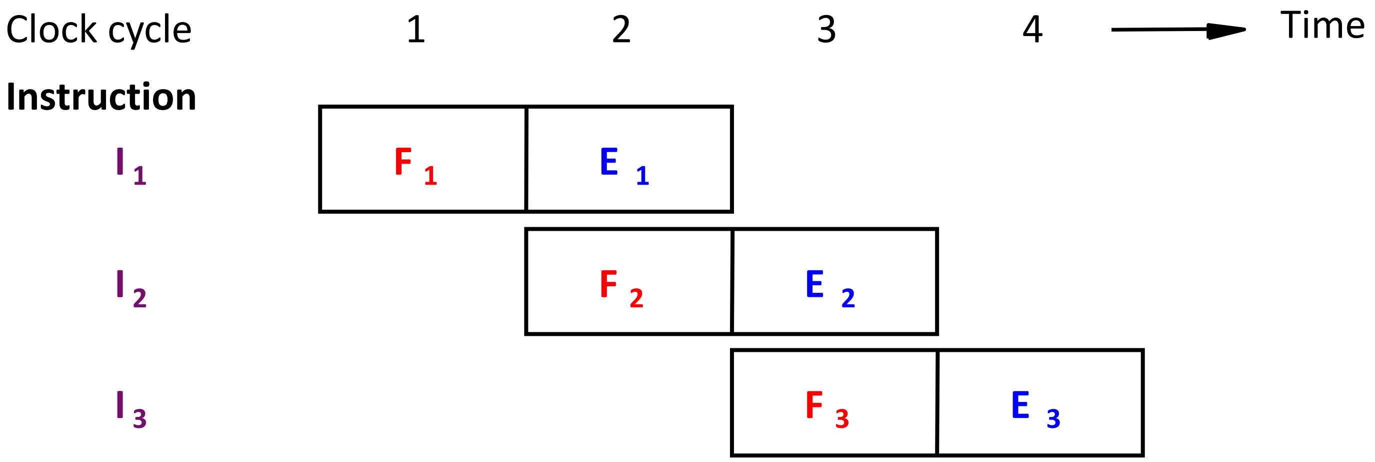
- **Interstage Buffer**: Deposit the fetched instruction.
  - Execution unit executes the deposited instruction.
  - Fetch unit fetches the next instruction at the same time.

**Interstage buffer**



## Basic Idea of Instruction Pipelining

- Assume the computer is controlled by a clock.
  - Both fetch and execute can be done in one clock cycle.
- Fetch and execute units form a two-stage pipeline:
  - Both units are kept busy all the time.
  - An interstage buffer is needed to hold the instruction.



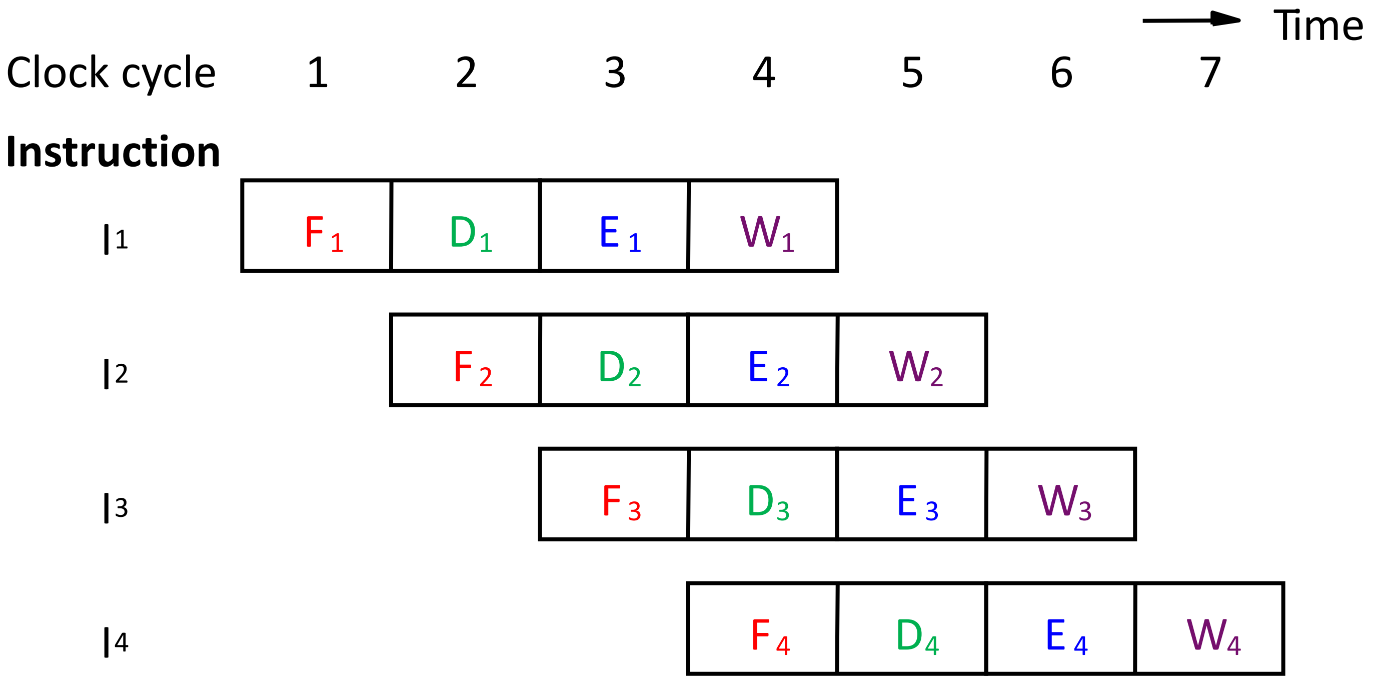
  - Parallelism is increased by overlapping fetch and execute.
    - If executions sustain for a long time, the completion rate of a two-stage pipelining will be twice (more stages always better?).

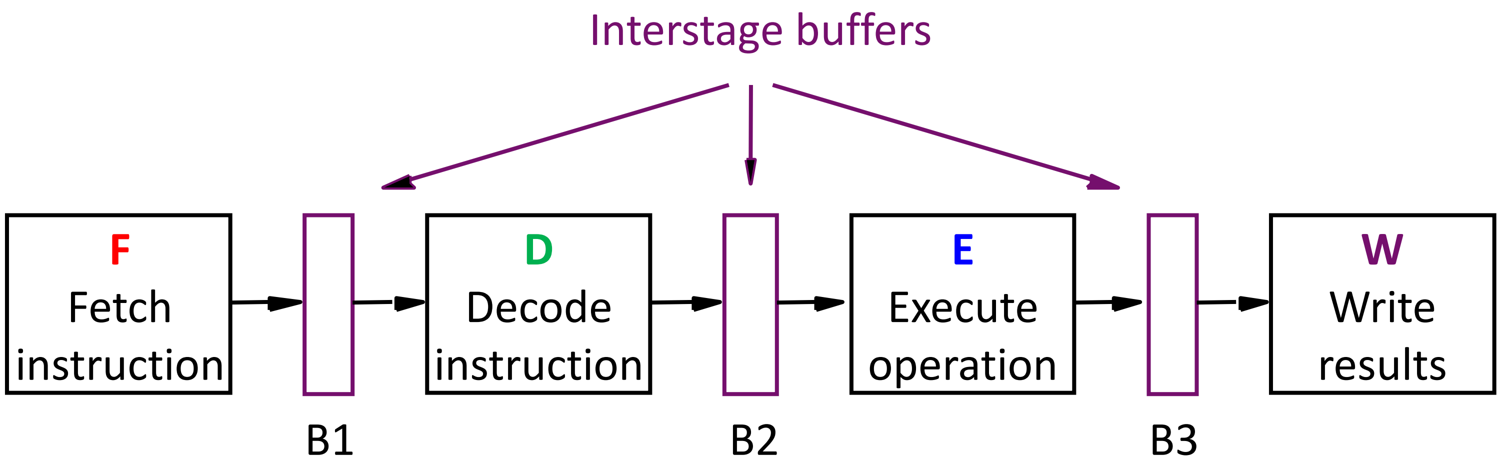
## 4-Stage Pipeline (1/2)

- **Design Principles of Pipeline**
  1) All stages should be able to perform their tasks simultaneously without interfering others.
     - The required information (i.e., instruction) is passed from one unit to the next through an interstage buffer.
  2) Each stage should take roughly the same maximum clock period (i.e., a clock cycle) to complete its task.
     - Why? A stage that completes its task early will be idle.

- **Example: 4-Stage Pipeline**
  - F: **Fetch** instruction from memory
  - D: **Decode** instruction and **fetch** source operands
  - E: **Execute** instruction
  - W: **Write** the result

# 4-Stage Pipeline (2/2)



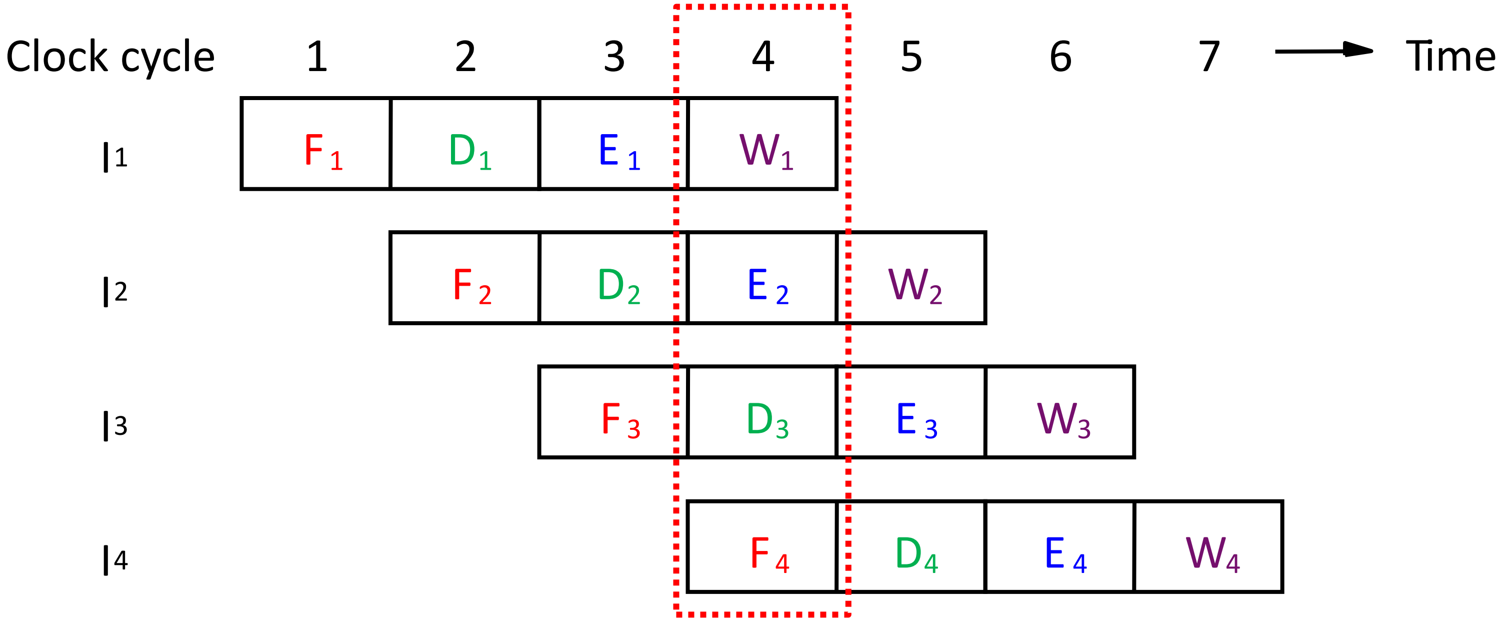


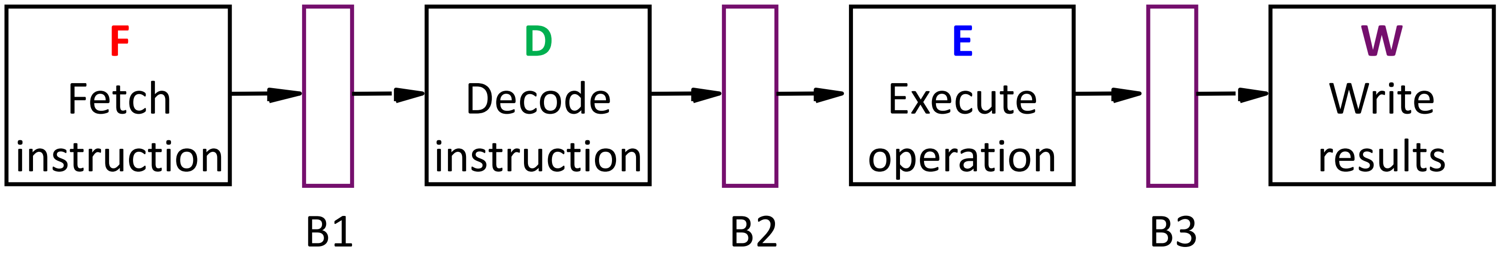
# Class Exercise 11.1

Student ID: ____________ Date:

Name: ________________ ________

- During clock cycle 4, what is the information hold by the three interstage buffers (i.e., B1, B2, and B3) respectively?



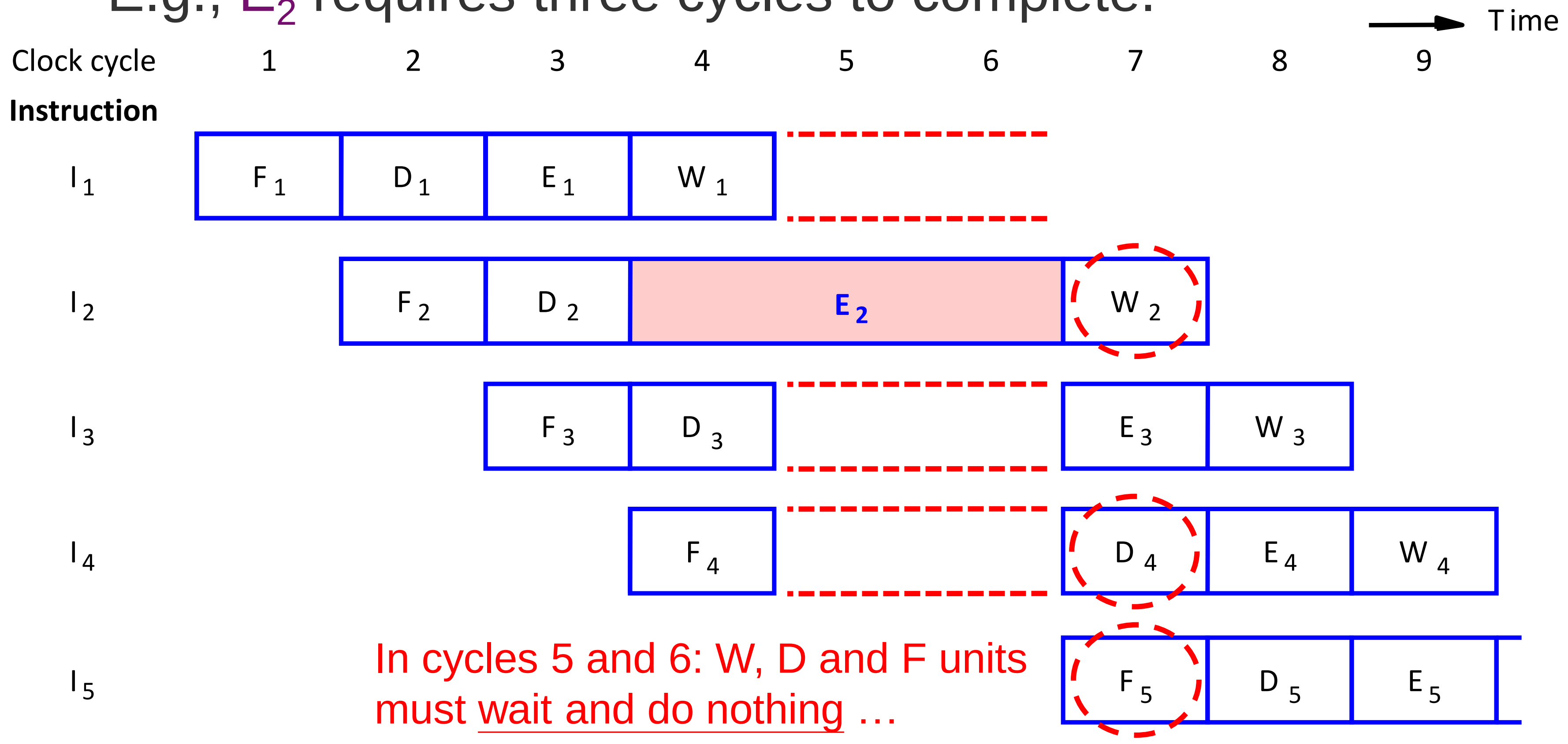


## Outline

- Sequential Execution vs Pipelining
- Pipeline Stall: Hazard
  - Data Hazard
  - Instruction Hazard
  - Structural Hazard
- Superscalar and Out-of-Order Execution

# Reality: Stall & Hazard

- If any pipeline stage requires more than 1 cycle, other stages must wait, causing the pipeline to stall.
  - E.g., $E_2$ requires three cycles to complete.



In cycles 5 and 6: W, D and F units must wait and do nothing …

- **Hazard**: Any condition that causes pipeline to stall.

## Types of Hazards

1) **Data Hazard**
   – The operands of an instruction are not available when required.

2) **Instruction Hazard**
   – A delay in the availability of an instruction.

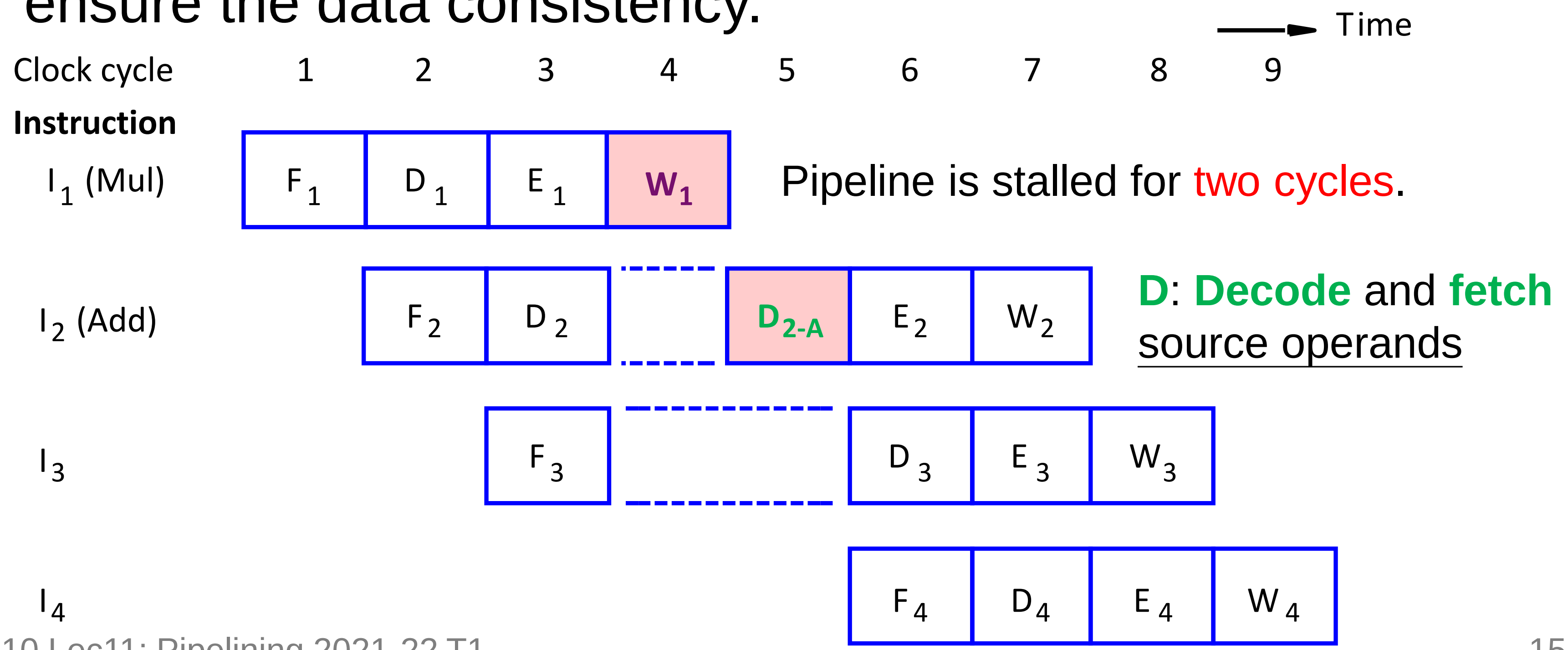
3) **Structural Hazard**
   – Two instructions require the use of a given hardware resource at the same time.

## Outline

- Sequential Execution vs Pipelining
- Pipeline Stall: Hazard
  - Data Hazard
  - Instruction Hazard
  - Structural Hazard
- Superscalar and Out-of-Order Execution

## 1) Data Hazard

- A data hazard is a situation in which the pipeline is stalled because the operands are delayed.
- Example: $I_1$: **A** = 3 * A;
  $I_2$: B = 4 + **A**;
  - Dependent operations must be performed sequentially to ensure the data consistency.



Pipeline is stalled for two cycles.

**D**: **Decode** and **fetch** source operands

## Class Exercise 11.2

- Please specify whether we will encounter data hazards for the following two cases.

**Case A**

$I_1$: A = 5 * **C**;

$I_2$: B = 20 + **C**;

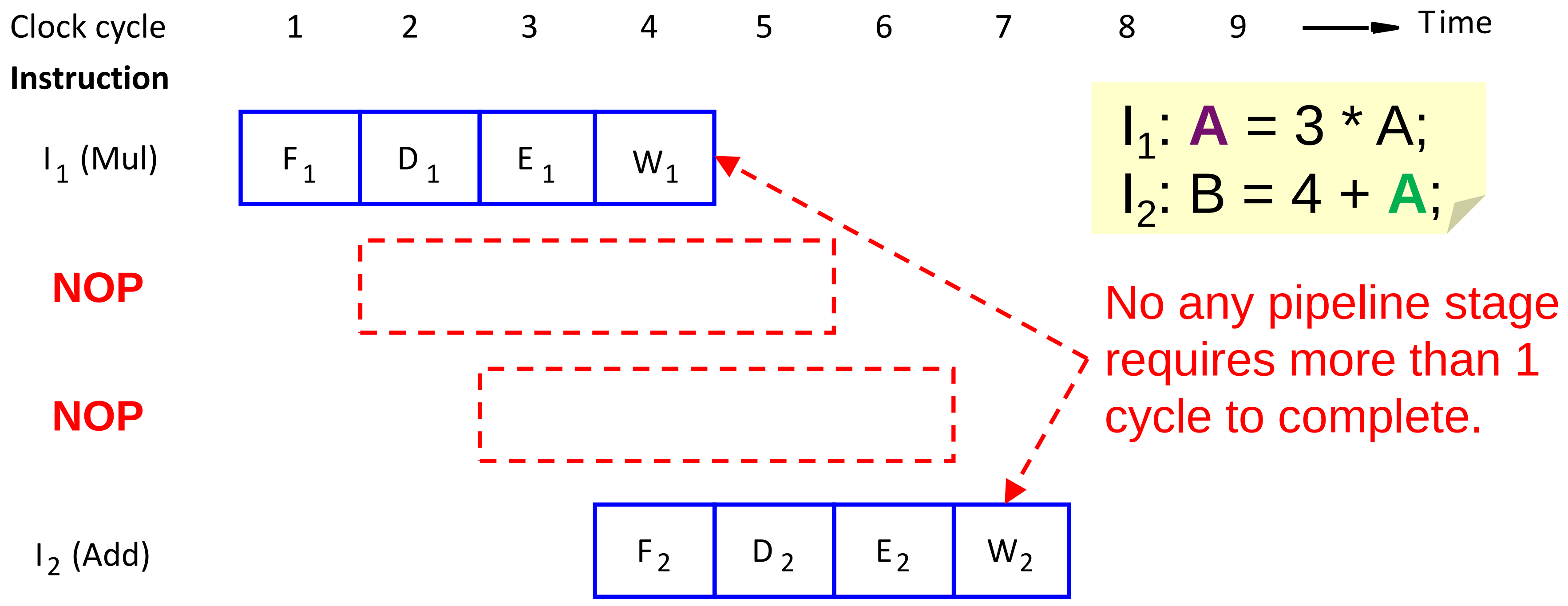
**Case B**

$I_1$: **C** = A * B;

$I_2$: E = **C** + D;

# Software Solution to Data Hazard

- The compiler detects and introduces two-cycle delay by inserting **NOP** (No-operation) instructions.



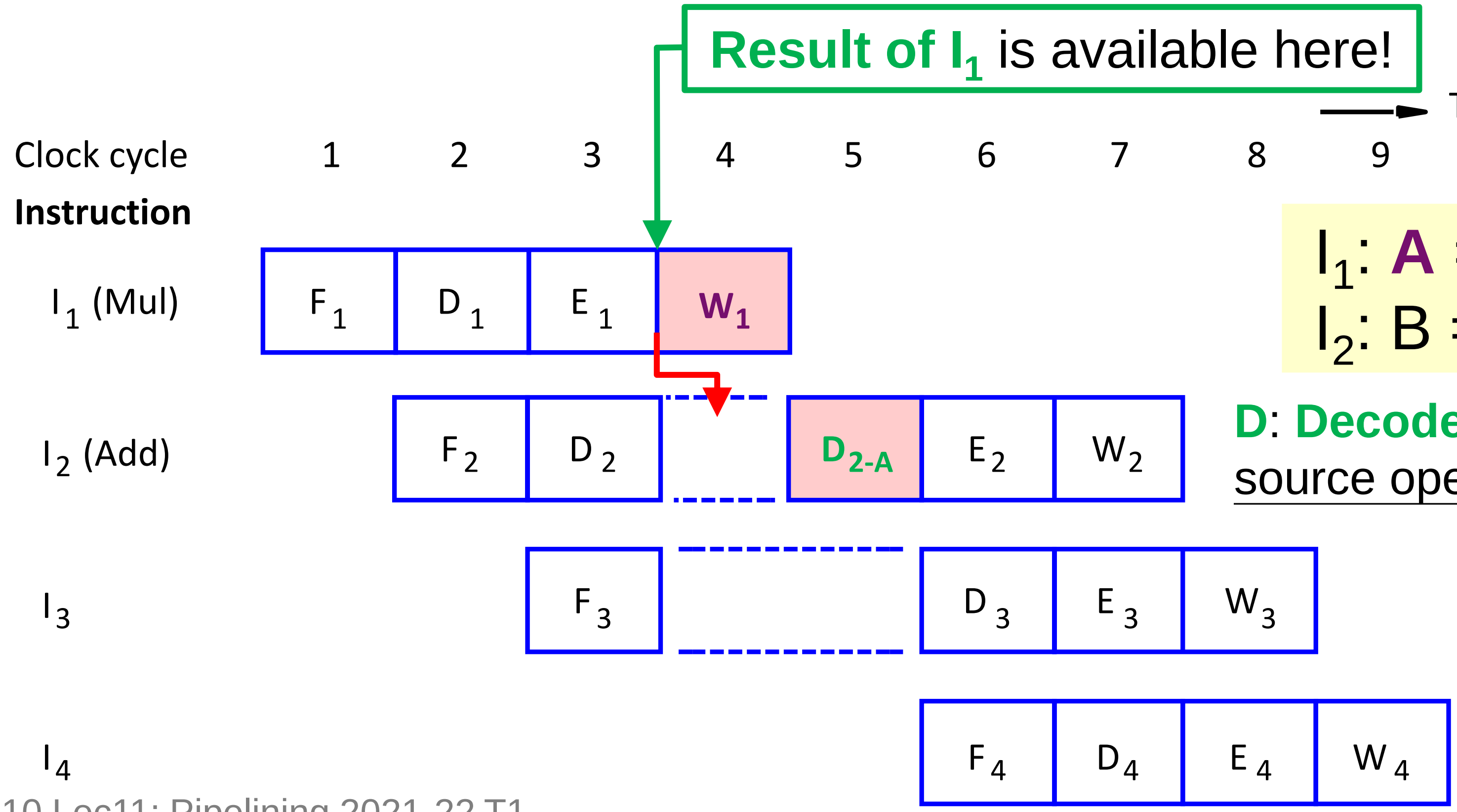
$I_1$: A = 3 * A;
$I_2$: B = 4 + A;

No any pipeline stage requires more than 1 cycle to complete.

  – Advantage: Simpler hardware, less cost
  – Disadvantage: Larger code size, less flexibility, and **"still degraded" performance**

# Hardware Solution to Data Hazard (1/2)

- The data hazard arises because $I_2$ is waiting for data to be written into the destination operand **A**.
- In fact, the result of $I_1$ is available at the output of ALU.
- Delay can be reduced if the result can be "forwarded".

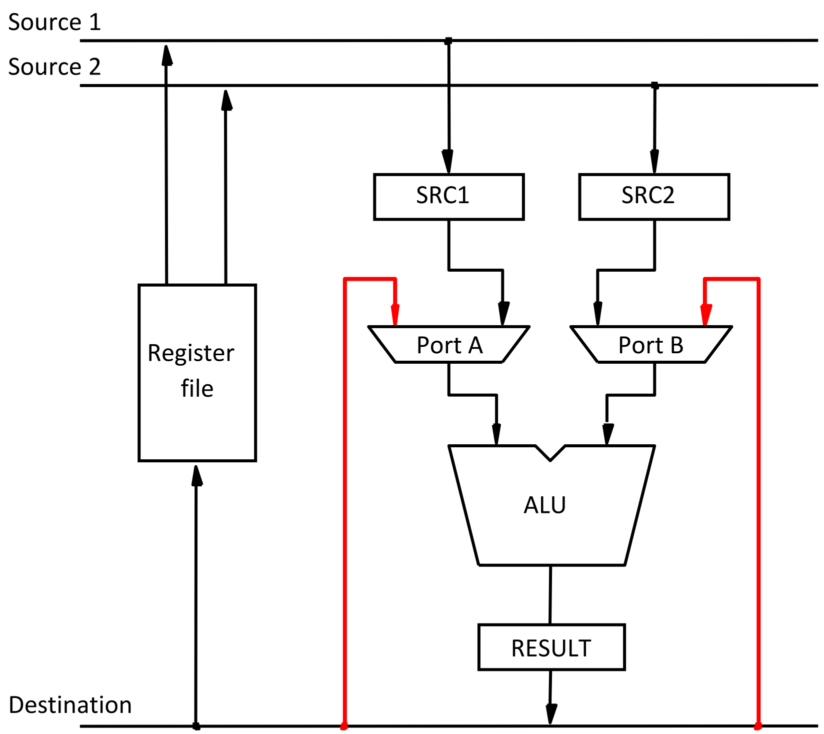


$I_1$: **A** = 3 * A;
$I_2$: B = 4 + **A**;

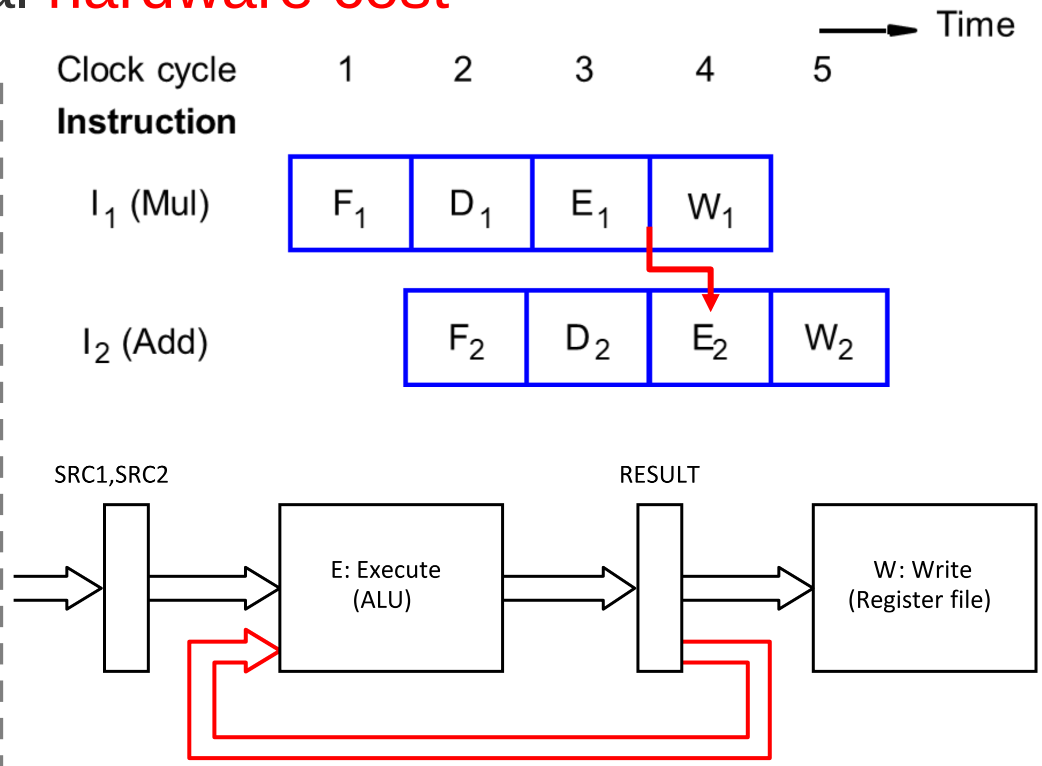
**D**: **Decode** and **fetch** source operands

## Hardware Solution to Data Hazard (2/2)

- **Operand Forwarding**: The execution of $I_2$ can proceed without stalling via the forwarding path.
  - Disadvantage: Additional hardware cost



(a) Datapath (3 buses)



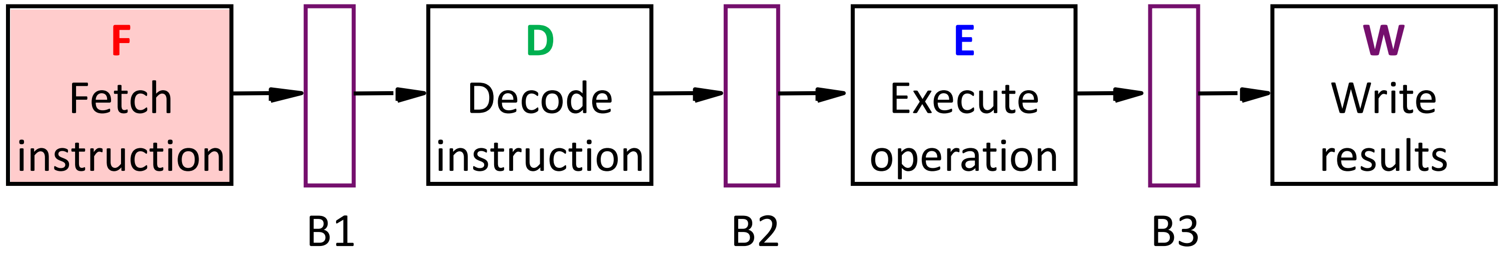
(b) Source and result registers

## Outline

- Sequential Execution vs Pipelining
- Pipeline Stall: Hazard
  - Data Hazard
  - Instruction Hazard
  - Structural Hazard
- Superscalar and Out-of-Order Execution

## 2) Instruction Hazard

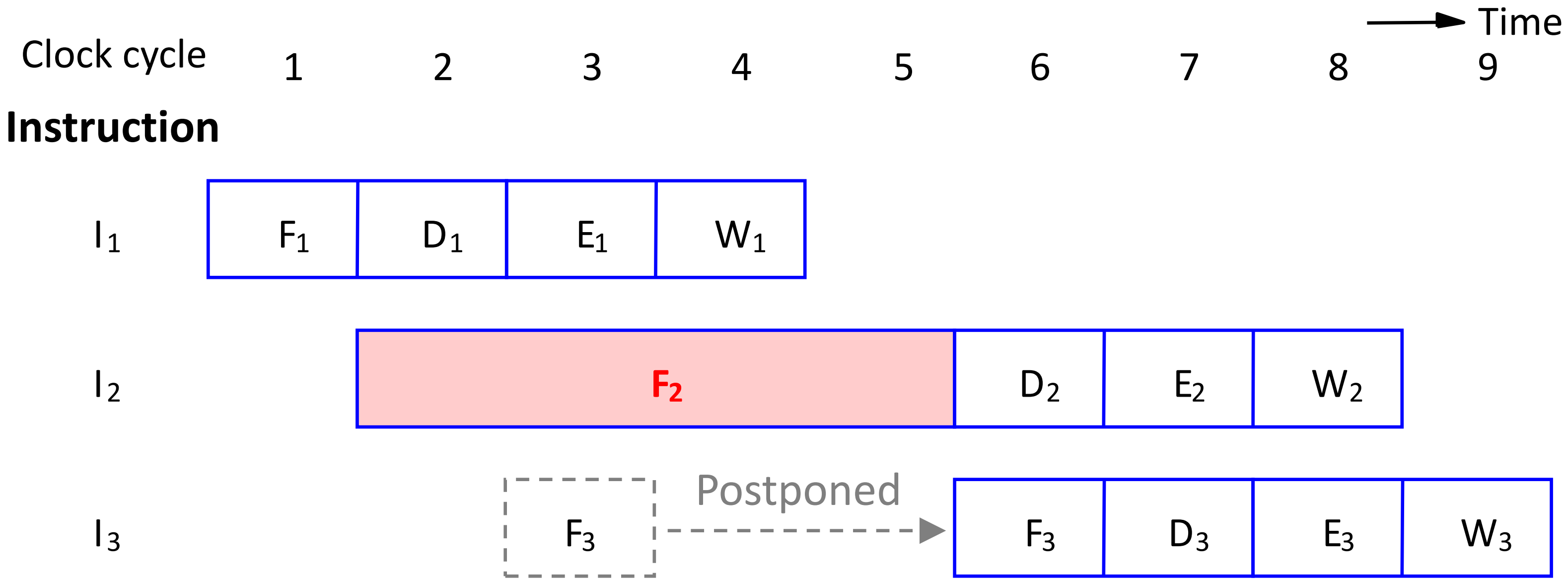
- Recall: The purpose of the instruction fetch unit is to supply the execution units with instructions.



  - F: **Fetch** instruction from memory
  - D: **Decode** instruction and **fetch** source operands
  - E: **Execute** instruction
  - W: **Write** the result
- **Instruction Hazard**: The cases cause the pipeline to stall, because of the delay of instructions.
  - Example 1: Cache miss
  - Example 2: Branch instruction

# Instruction Hazard Ex1: Cache Miss

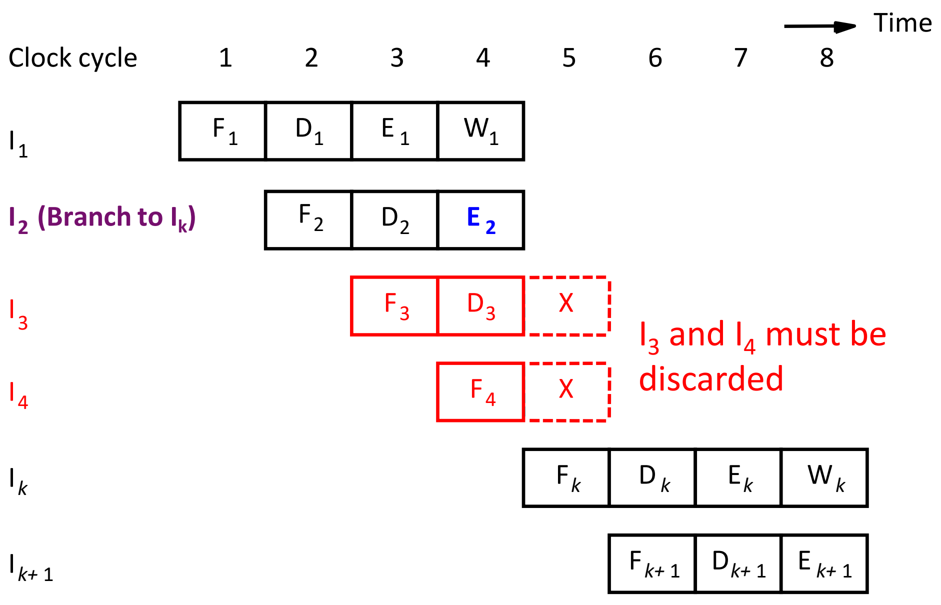
- The effect of a cache miss on the pipelined operation is as follows:


Time
Clock cycle 1 2 3 4 5 6 7 8 9
Instruction
$I_1$ $F_1$ $D_1$ $E_1$ $W_1$
$I_2$ $F_2$ $D_2$ $E_2$ $W_2$
$I_3$ $F_3$ Postponed $F_3$ $D_3$ $E_3$ $W_3$

  – $I_1$ is fetched from the cache in cycle 1.
  – The fetch operation $F_2$ for $I_2$ results in a cache miss.
    - The instruction fetch unit must suspend any further fetch requests until $F_2$ is completed.

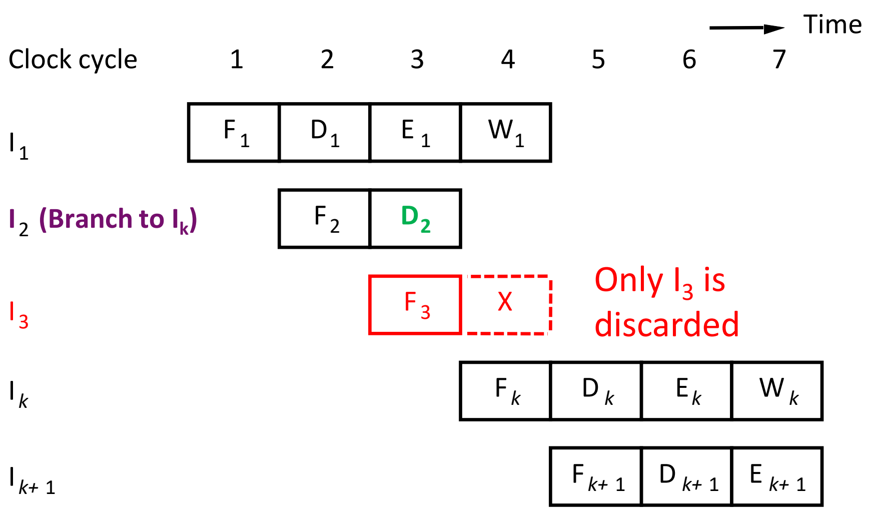
# Instruction Hazard Ex2: Branch

- Branches may also cause the pipeline to stall.
  - **Branch Penalty**: The time lost because of a branch inst.
  - Branch penalty can be reduced by computing the branch address earlier in Decode stage (rather than Execute stage)
    - However, it still results in 1 cycle branch penalty to the pipeline.



Time
Clock cycle 1 2 3 4 5 6 7 8
$I_1$: $F_1$ $D_1$ $E_1$ $W_1$
$I_2$ (Branch to $I_k$): $F_2$ $D_2$ $E_2$
$I_3$: $F_3$ $D_3$ X
$I_4$: $F_4$ X
$I_3$ and $I_4$ must be discarded
$I_k$: $F_k$ $D_k$ $E_k$ $W_k$
$I_{k+1}$: $F_{k+1}$ $D_{k+1}$ $E_{k+1}$

Branch address computed in **Execute stage**
Branch Penalty: **2** clock cycles

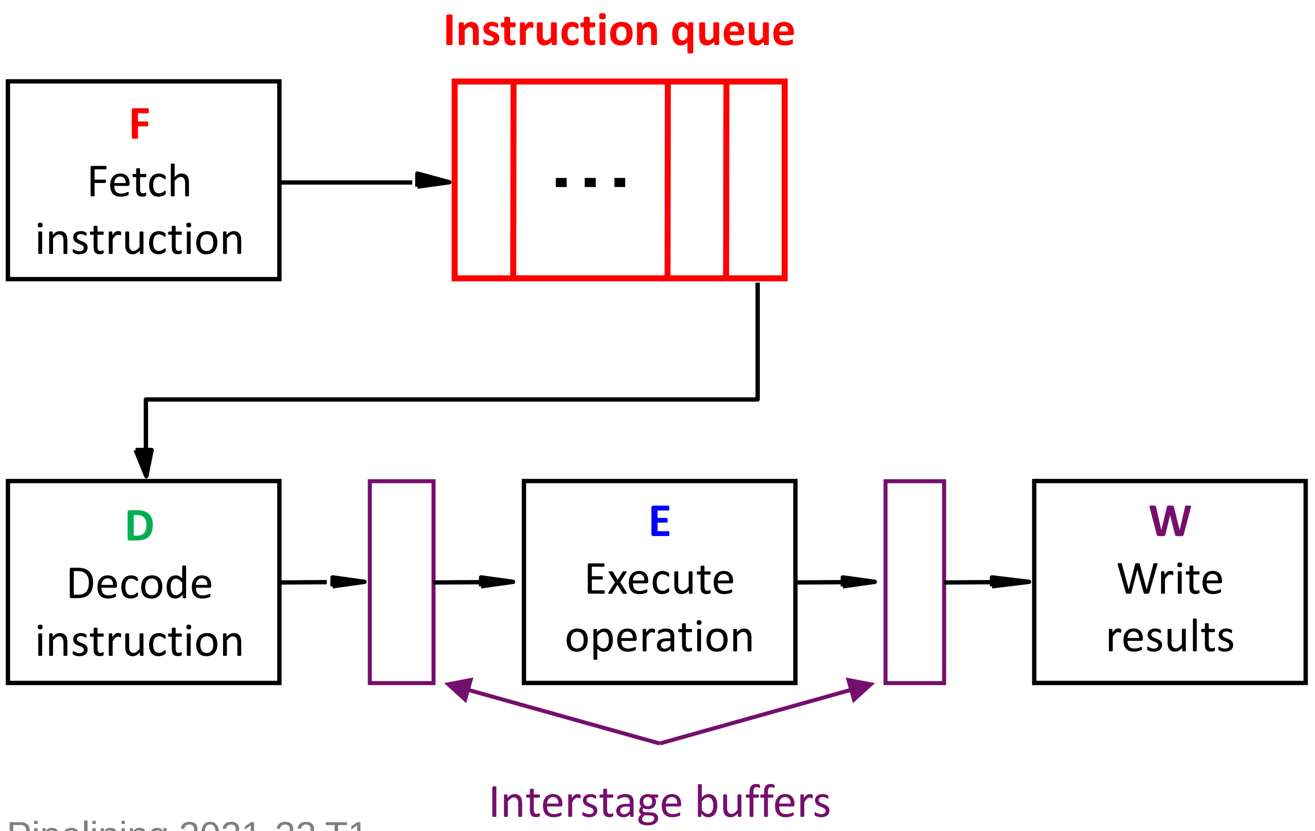


Time
Clock cycle 1 2 3 4 5 6 7
$I_1$: $F_1$ $D_1$ $E_1$ $W_1$
$I_2$ (Branch to $I_k$): $F_2$ $D_2$
$I_3$: $F_3$ X
Only $I_3$ is discarded
$I_k$: $F_k$ $D_k$ $E_k$ $W_k$
$I_{k+1}$: $F_{k+1}$ $D_{k+1}$ $E_{k+1}$

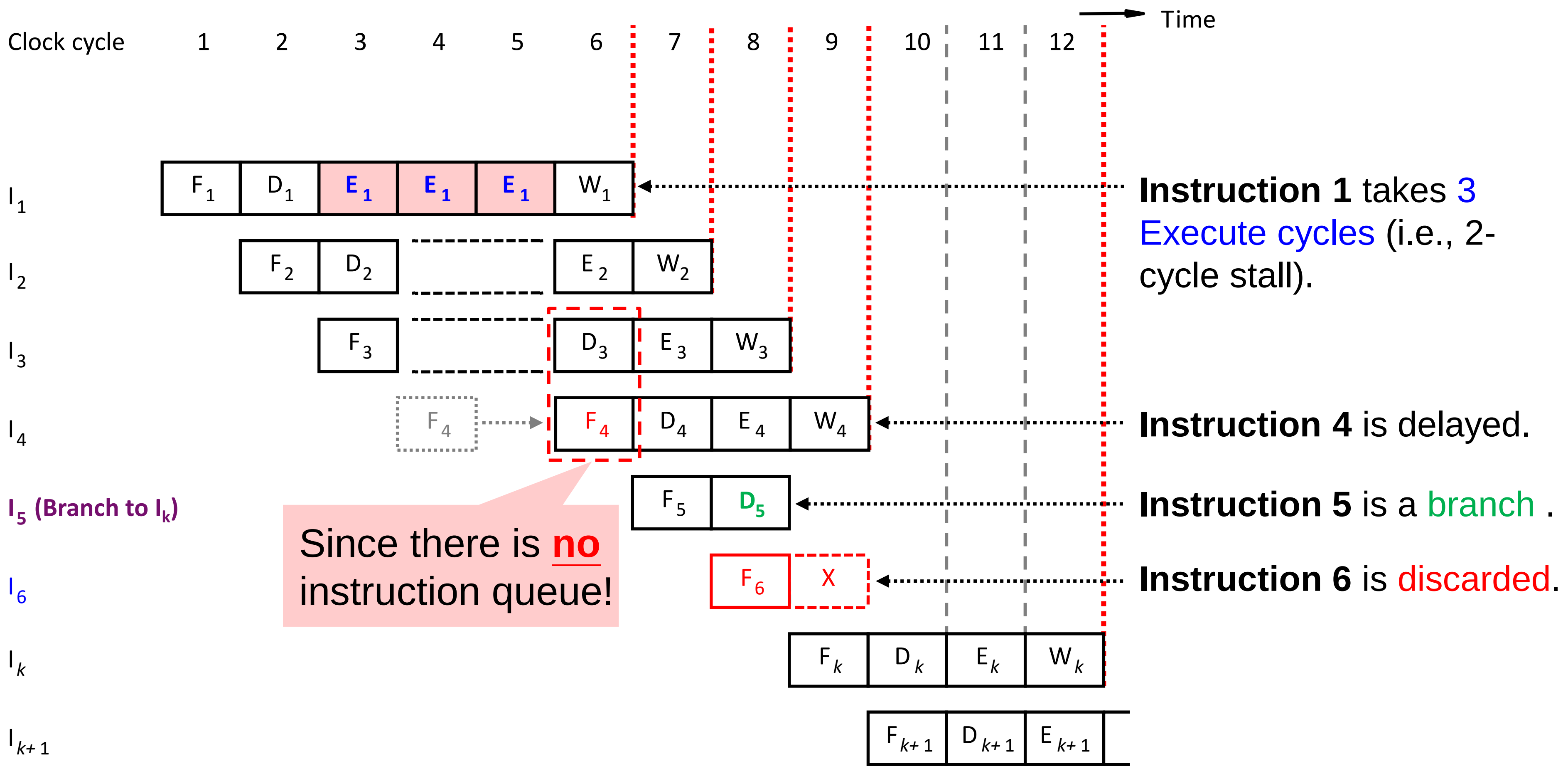
Branch address computed in **Decode stage**
Branch Penalty: **1** clock cycle

## Solution to Instruction Hazard

- **Instruction Queue**: The interstage buffer between Fetch and Decode units can keep multiple instructions.
  - Fetch unit gets and deposits one instruction at a time.
  - Decode unit consumes one instruction at a time.



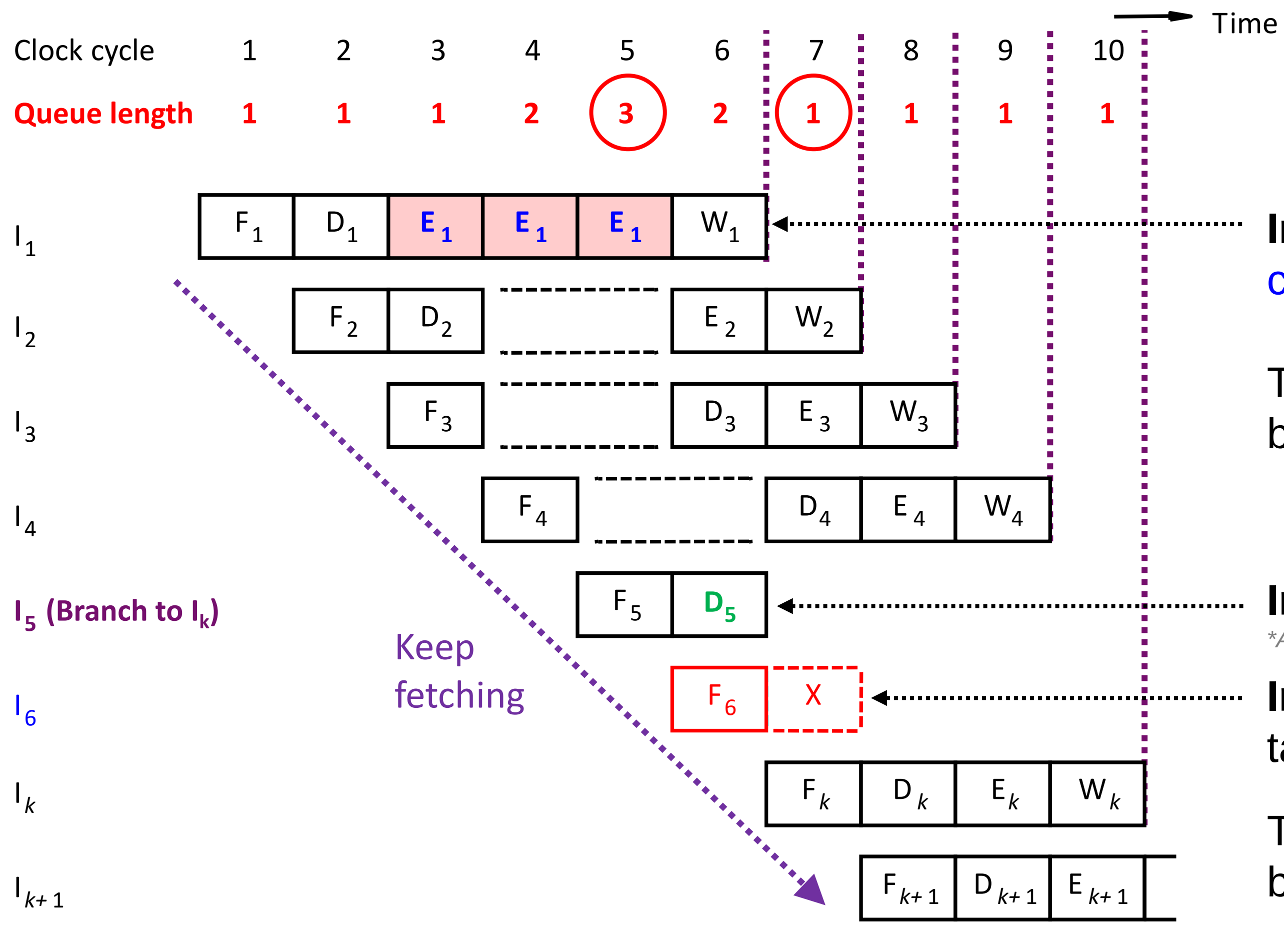
# Example: Without Instruction Queue



- Without the instruction queue:

  $I_1$, $I_2$, $I_3$, $I_4$, and $I_k$ cannot complete in successive cycles.

# Example: With Instruction Queue



**Instruction 1** takes 3 Execute cycles (i.e., 2-cycle stall),

The queue length rises to 3 before cycle 6.

**Instruction 5** is a branch .

**Assume D3 and D5 can be performed at the same time.*

**Instruction 6** is discarded, after taking Branch.

The queue length drops to 1 before cycle 8.

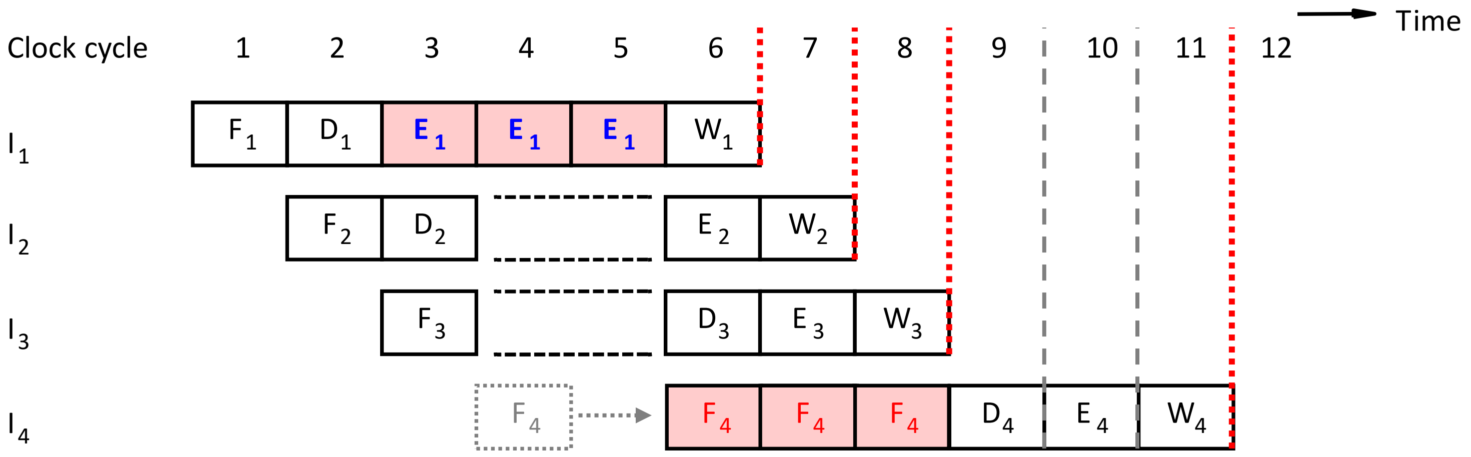
- With the instruction queue:

  $I_6$ is still discarded but $I_1$, $I_2$, $I_3$, $I_4$, and $I_k$ can be "possibly" completed in successive cycles.

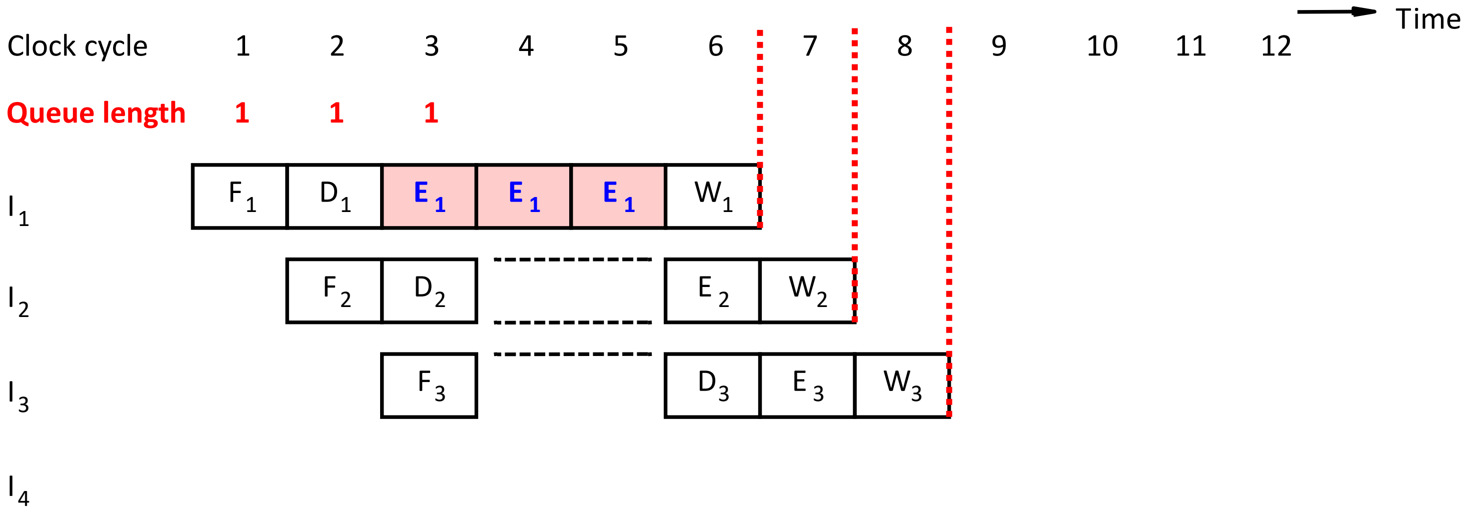
# Class Exercise 11.3

- Please show how the instruction queue can help hide the delay of cache miss (3 cycles) caused by $F_4$.

**Without Instruction Queue**



**With Instruction Queue**

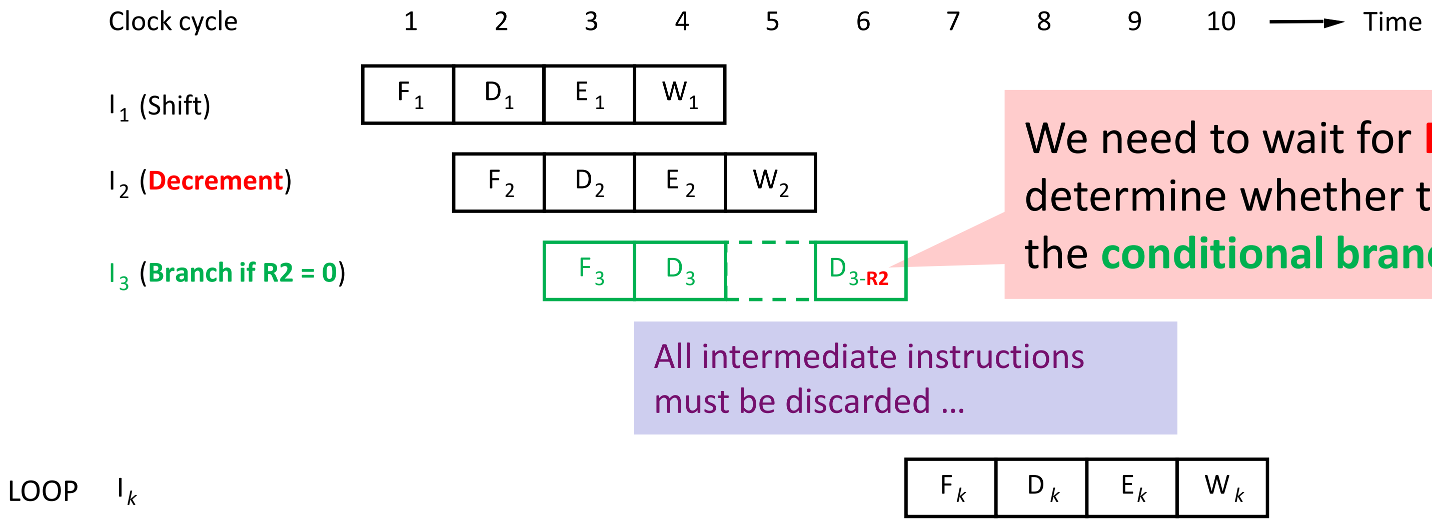


# Instruction Hazard: Conditional Branch

- Conditional branches may worsen the hazard.
  - Since the condition is based on the preceding instruction.
- Example:

| | | |
|---|---|---|
| LOOP | Shift_left | R1 |
| | Decrement | R2 |
| | **Branch=0** | LOOP |
| NEXT | Add | R1,R3 |

**R2** is used as the **branch condition**.



We need to wait for **R2** to determine whether to perform the **conditional branching**.

All intermediate instructions must be discarded …

# Solution 1) Delayed Branch (1/2)

- The location(s) following a branch instruction is called branch delay slot(s).
  - There may be more than one branch delay slot, depending on how long it takes to execute a branch.
- **Delayed branching** can minimize the penalty by
  - Placing useful instructions in branch delay slot(s), and
  - Internally re-ordering the instructions.

| | | |
|---|---|---|
| LOOP | Shift_left | R1 |
| | Decrement | R2 |
| | **Branch=0** | LOOP |
| | Branch Delay Slot | |
| NEXT | Add | R1,R3 |

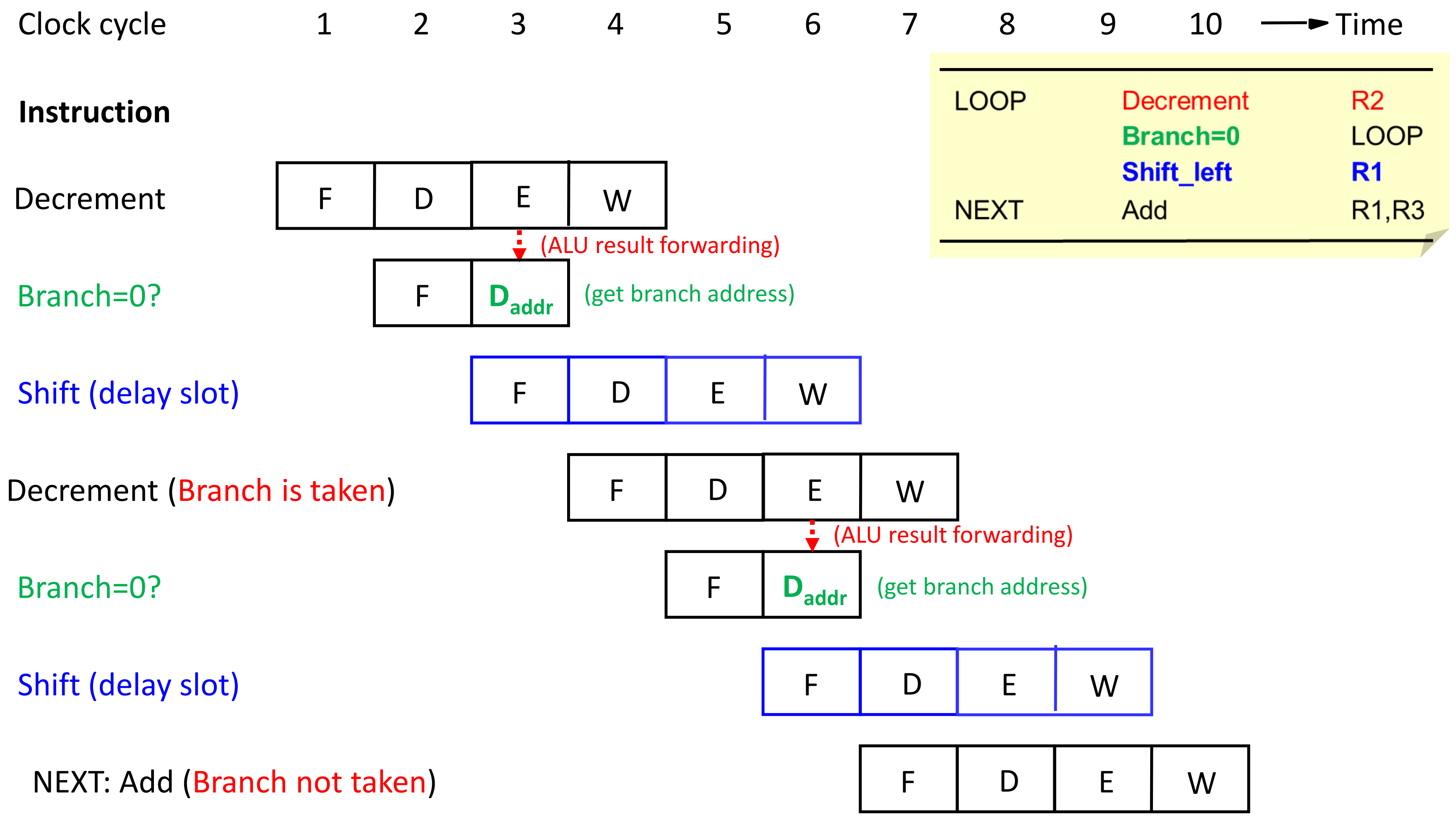
(a) Original program loop

| | | |
|---|---|---|
| LOOP | Decrement | R2 |
| | **Branch=0** | LOOP |
| | **Shift_left** | **R1** |
| NEXT | Add | R1,R3 |

(b) **Internally** Re-ordered instructions
(actual program logic NOT affected)

# Solution 1) Delayed Branch (2/2)

- Delayed branching can minimize the branch penalty.



Clock cycle 1 2 3 4 5 6 7 8 9 10 Time
Instruction
Decrement F D E W
(ALU result forwarding)
Branch=0? F $D_{addr}$
(get branch address)
Shift (delay slot) F D E W
Decrement (Branch is taken) F D E W
(ALU result forwarding)
Branch=0? F $D_{addr}$
(get branch address)
Shift (delay slot) F D E W
NEXT: Add (Branch not taken) F D E W

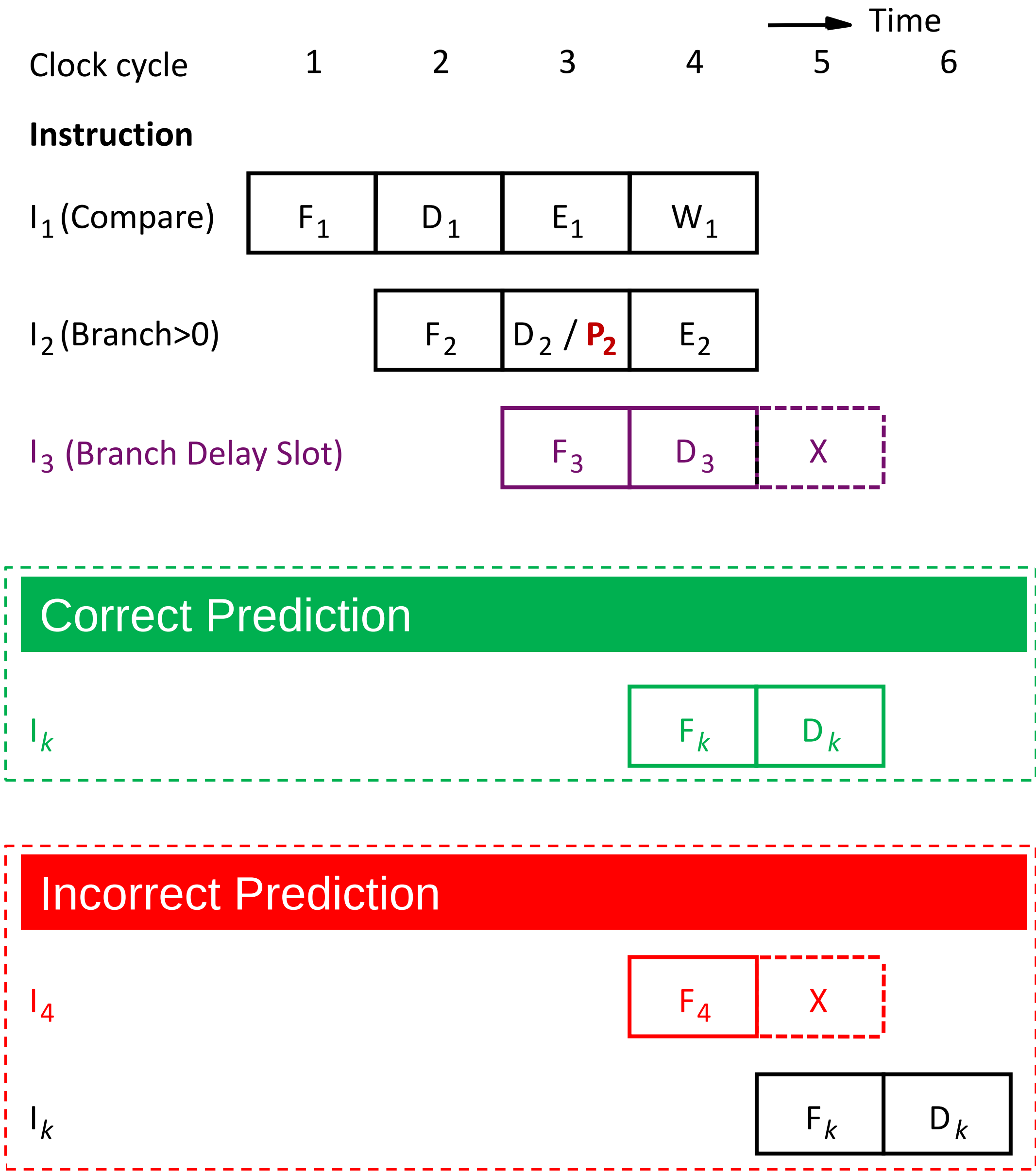
| | | |
|---|---|---|
| LOOP | Decrement | R2 |
| | Branch=0 | LOOP |
| | Shift_left | R1 |
| NEXT | Add | R1,R3 |

## Class Exercise 11.4

- Suppose a pipelined processor has two branch delay slots but does not utilize the delayed branch technique. If 20 percent of the instructions executed are branch instructions, what is the required number of cycles to complete 100 instructions?

# Solution 2) Branch Prediction (1/2)

- Attempt to predict whether conditional branch will take place.
  - Delayed branch can be applied together.
- **Branch Prediction**:
  - If we get it right: no lost cycles.
    - Registers and memory cannot be updated until we know we got it right.
  - If we get it wrong, just cancel the instructions.
  - Branch prediction can be dynamic or static.



## Solution 2) Branch Prediction (2/2)

- **Static** Branch Prediction
  - The same choice is used every time the conditional branch is encountered.
  - *For example, a branch instruction at the end of a loop causes a branch to the start of the loop for every pass through the loop except the last one.*
    - *It is helpful to assume this branch will be taken under this case.*
  - A flexible approach is to have the compiler decide.

- **Dynamic** Branch Prediction
  - The choice is influenced by the past behavior.
  - For example, a simple prediction is to use the result of the most recent execution of the branch instruction.

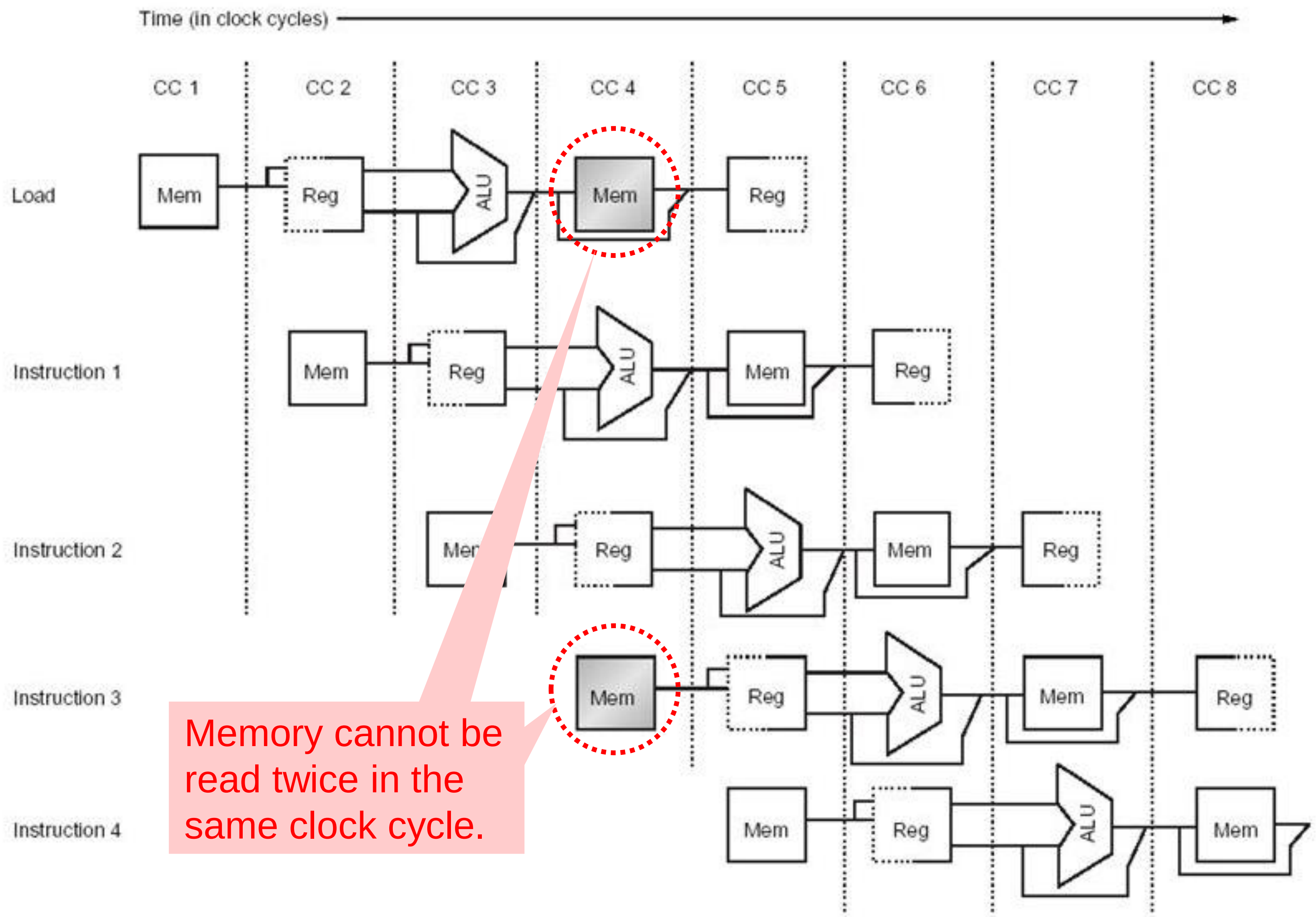
## Outline

- Sequential Execution vs Pipelining
- Pipeline Stall: Hazard
  - Data Hazard
  - Instruction Hazard
  - Structural Hazard
- Superscalar and Out-of-Order Execution

## 3) Structural Hazard

- A structural hazard is the situation when two instructions require the use of a hardware resource at the same time.
- The most common case is in accessing to memory.
  - Case 1: One instruction is accessing memory during the Execute or Write stage; while another is being fetched.
  - Solution 1: Many processors use separate instruction and data caches to avoid this delay.
  - Case 2: Another example is when two instructions require access to the register file at the same time.
  - Solution 2: Let the register file have more input/output ports.
- In general, the structural hazard can be avoided by providing sufficient hardware resources ($$$).

# An Example of Structural Hazard



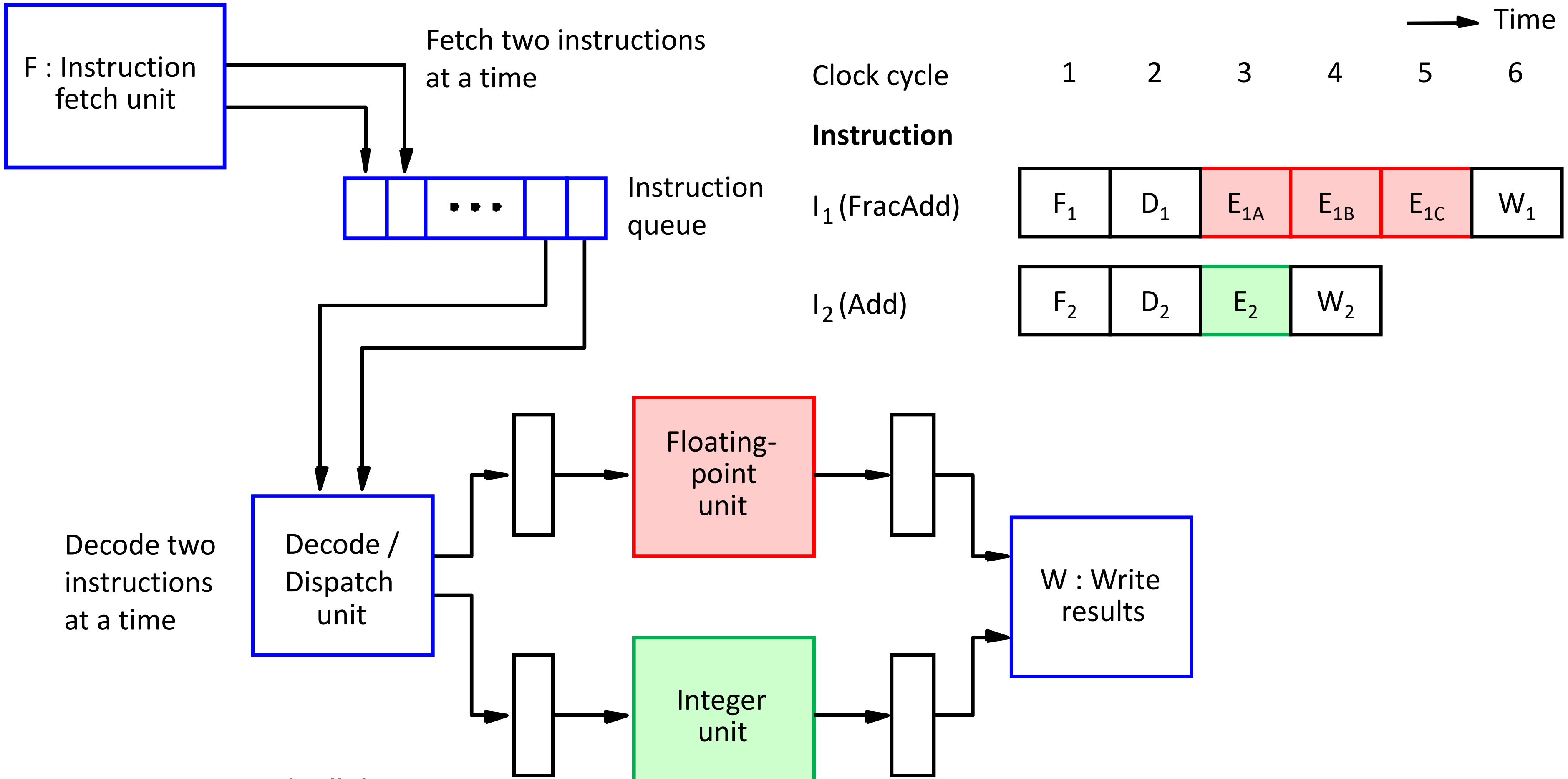
Memory cannot be read twice in the same clock cycle.

## Outline

- Sequential Execution vs Pipelining
- Pipeline Stall: Hazard
  - Data Hazard
  - Instruction Hazard
  - Structural Hazard
- Superscalar and Out-of-Order Execution

# Superscalar Operation

- **Superscalar**: Execute multiple instructions at any time via multiple processing units (i.e., we can execute more than one instruction per cycle)


F : Instruction fetch unit
Fetch two instructions at a time
Instruction queue
Decode two instructions at a time
Decode / Dispatch unit
Floating-point unit
Integer unit
W : Write results
Time

| Clock cycle | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| **Instruction** | | | | | | |
| $I_1$ (FracAdd) | $F_1$ | $D_1$ | $E_{1A}$ | $E_{1B}$ | $E_{1C}$ | $W_1$ |
| $I_2$ (Add) | $F_2$ | $D_2$ | $E_2$ | $W_2$ | | |

## Out-of-Order Execution (1/2)

- Superscalar operation may result in out-of-order execution, and cause data consistency issue.
  - In our previous example, $I_1$ and $I_2$ are dispatched in the same order as they appear.
  - However, their execution is completed out of order.
  - To guarantee a consistent state when out-of-order execution occur, the results of the execution of instructions must be written in program order strictly .
- The **out-of-order execution** can make good use of cycles if instructions can be "properly re-ordered".
  - E.g., the delayed branching technique reorders the instructions to minimize the branch penalty.

# Out-of-Order Execution (2/2)

- Instruction 1 results in a cache miss, and a cache miss can stall entire processor for 20-30 cycles.
- Instruction 2 cannot be executed since it needs R1.

```
R1 ← mem[r0]   /* Instruction 1 */
R2 ← R1 + R2   /* Instruction 2 */
R5 ← R5 + 1    /* Instruction 3 */
R6 ← R6 – R3   /* Instruction 4 */
```

- In instruction queue, look ahead and find instructions 3 and 4 to execute first (reordering).

```
R1 ← mem[r0]   /* Instruction 1 */
R5 ← R5 + 1    /* Instruction 3 */
R6 ← R6 – R3   /* Instruction 4 */
R2 ← R1 + R2   /* Instruction 2 */
```

## Summary

- Sequential Execution vs Pipelining
- Pipeline Stall: Hazard
  - Data Hazard
  - Instruction Hazard
  - Structural Hazard
- Superscalar and Out-of-Order Execution