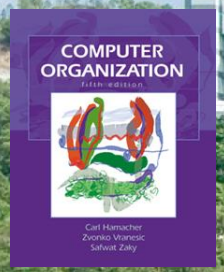
香港中文大學
The Chinese University of Hong Kong

CSCI2510 Computer Organization

Lecture 12: Pipelining

Ming-Chang YANG

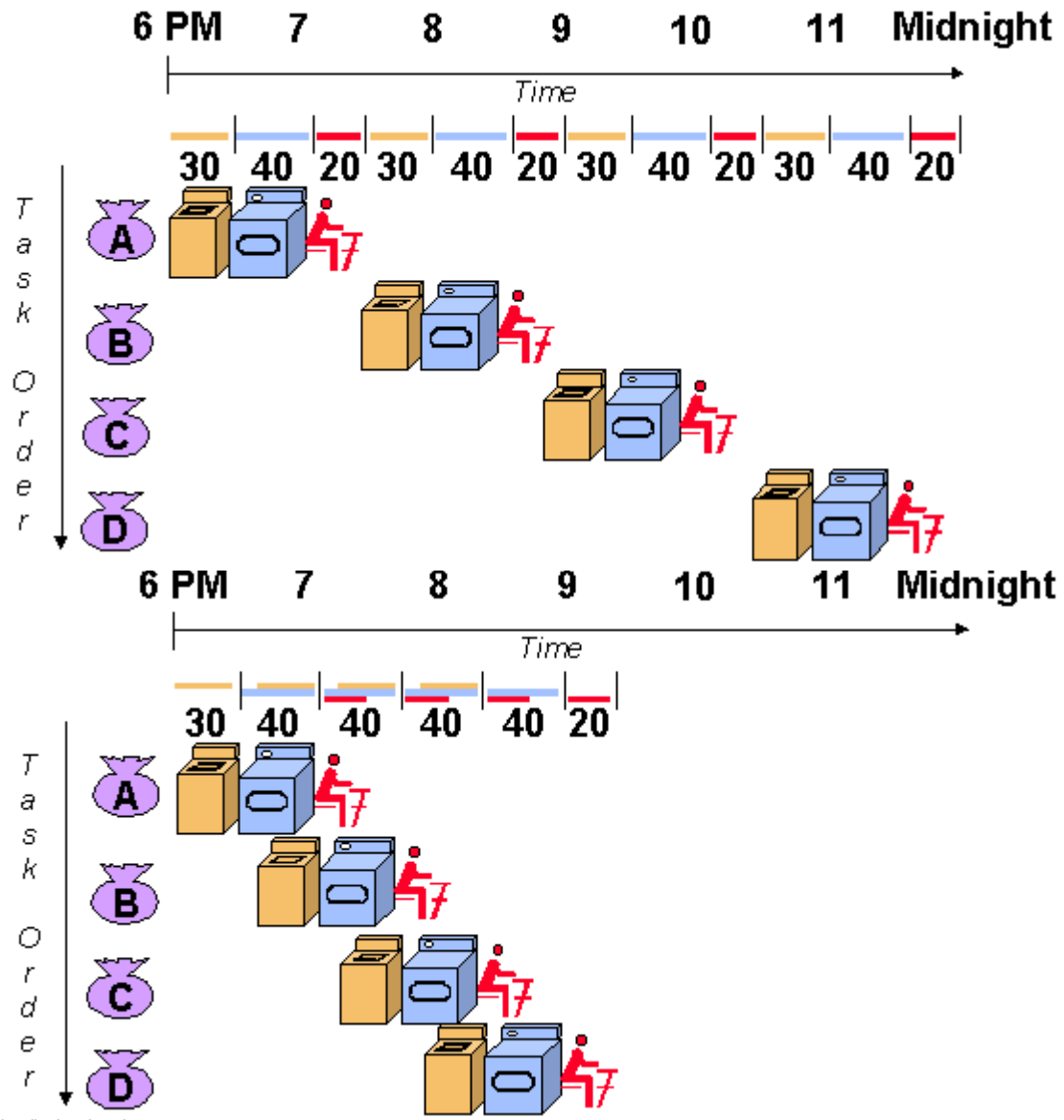
mcyang@cse.cuhk.edu.hk



Reading: Chap. 8 (5th Ed.)

Why We Need Pipelining?

- Real-life example: Four loads of laundry that need to be washed, dried, and folded.
  - Washing: 30 min
  - Drying: 40 min
  - Folding: 20 min
- **Without** pipeline:
  - 360 min in total
- **With** pipeline:
  - 210 min in total!



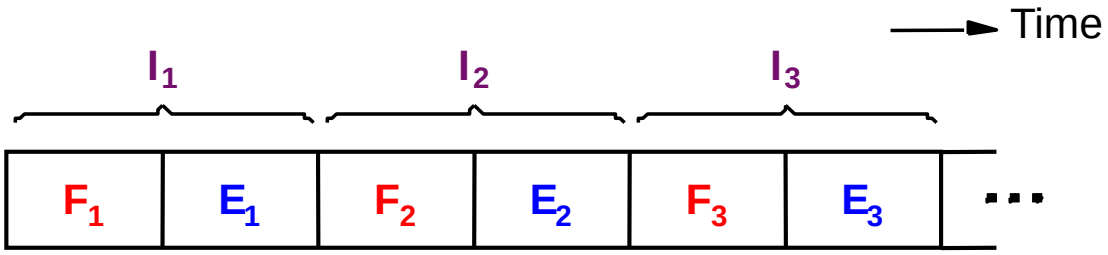
https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/pipelining/index.html

Outline

- Sequential Execution vs Pipelining
- Pipeline Stall: Hazard
  - Data Hazard
  - Instruction Hazard
  - Structural Hazard
- Superscalar Operation

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

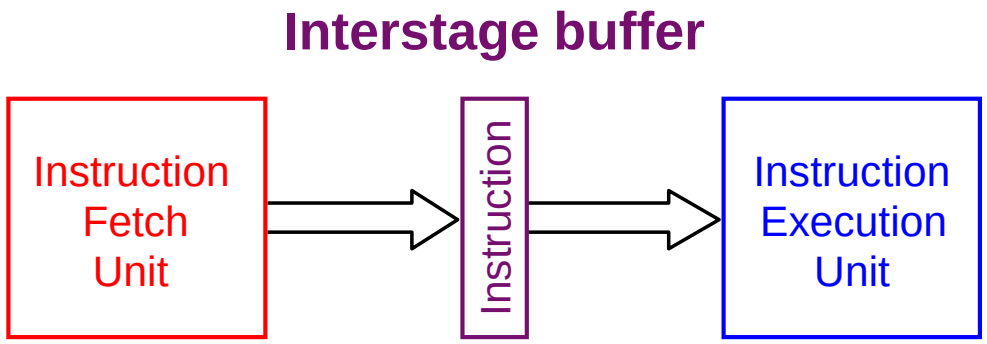


- How to improve the speed of execution?
  - Use faster technologies to build CPU and memory ($$$).
  - Arrange hardware to do multiple operations at a time ($).

Separate HW & Interstage Buffer

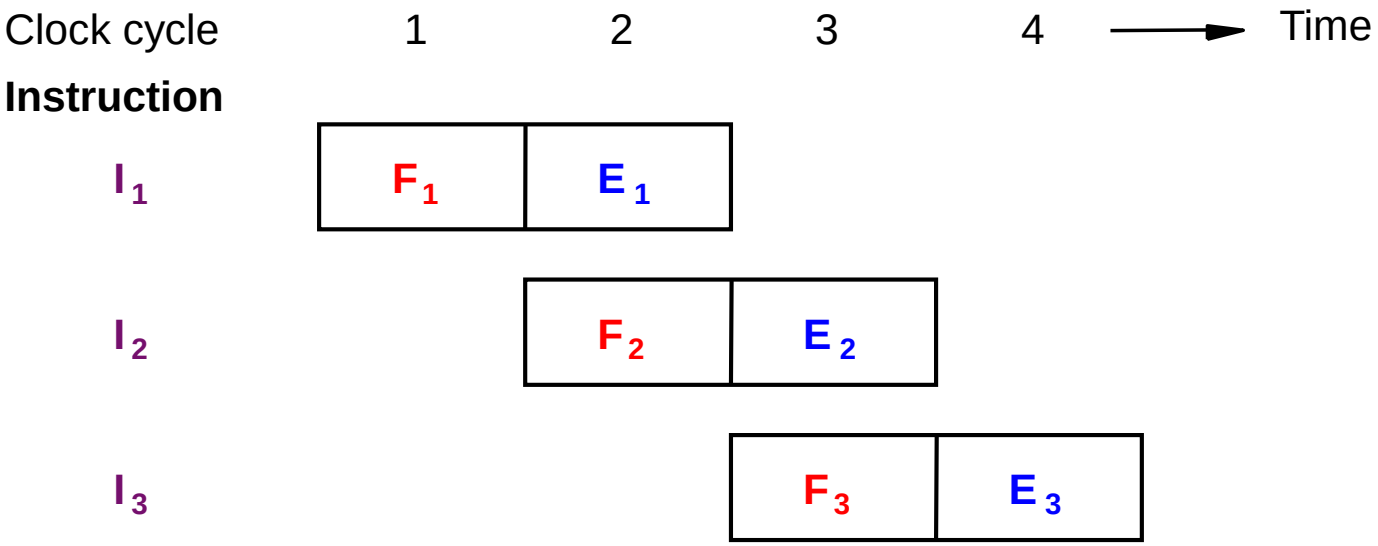
- Consider a computer having two **separate hardware units**:
  - One hardware unit is for fetching instructions.
  - The other hardware unit is for executing instructions.

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



Basic Idea of Instruction Pipelining (1/2)

- Assume the computer is controlled by a clock.
  - The fetch and execute steps of any instruction can be completed 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.

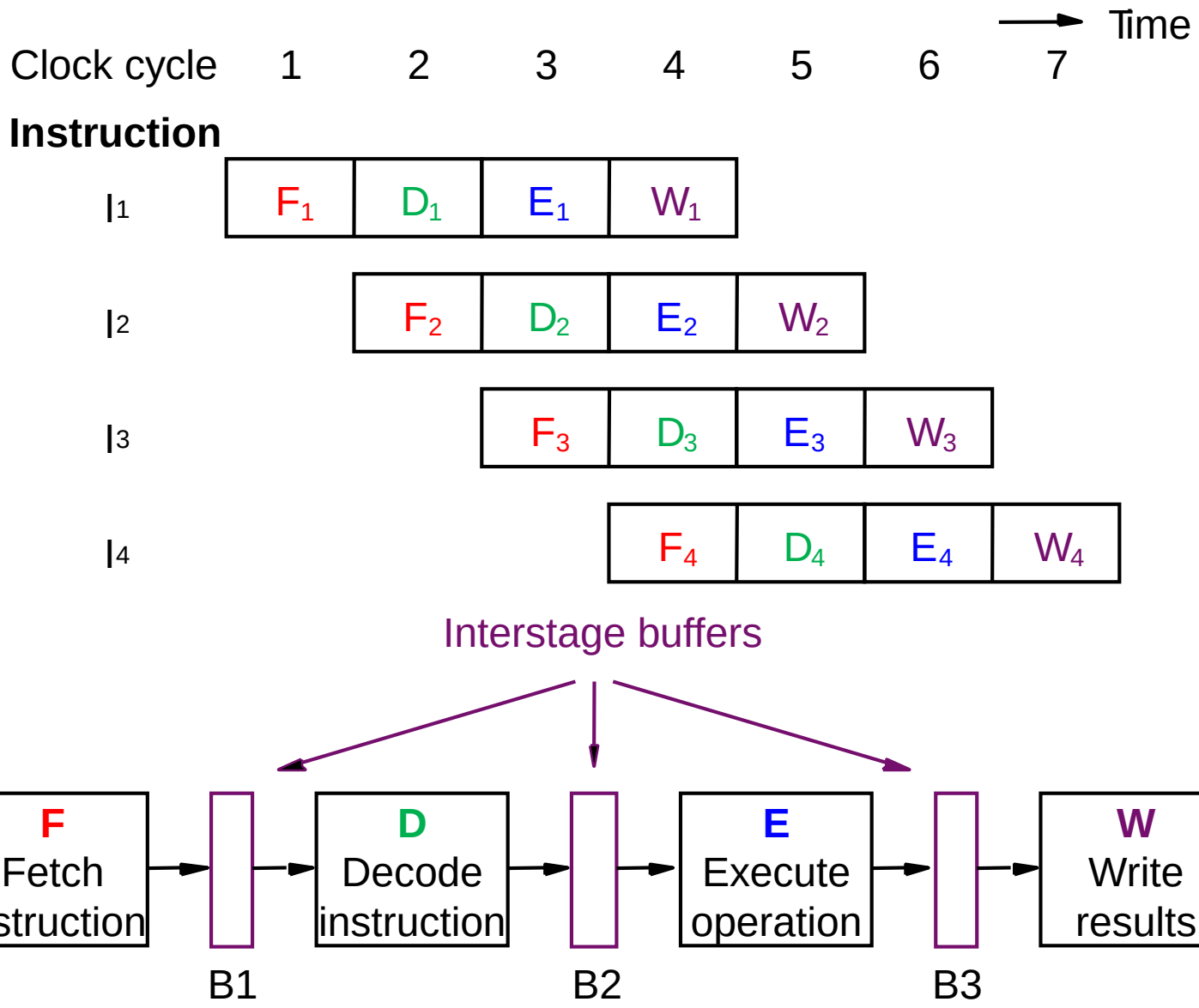


Basic Idea of Instruction Pipelining (2/2)

- Parallelism is increased by overlapping the fetch and execute steps.
  - If executions sustain for a long time, the completion rate of a two-stage pipelining will be twice.

- More is better? How about 4-stage pipeline?
  - **F**: **Fetch** instruction from memory
  - **D**: **Decode** instruction and **fetch** source operands
  - **E**: **Execute** instruction
  - **W**: **Write** the result

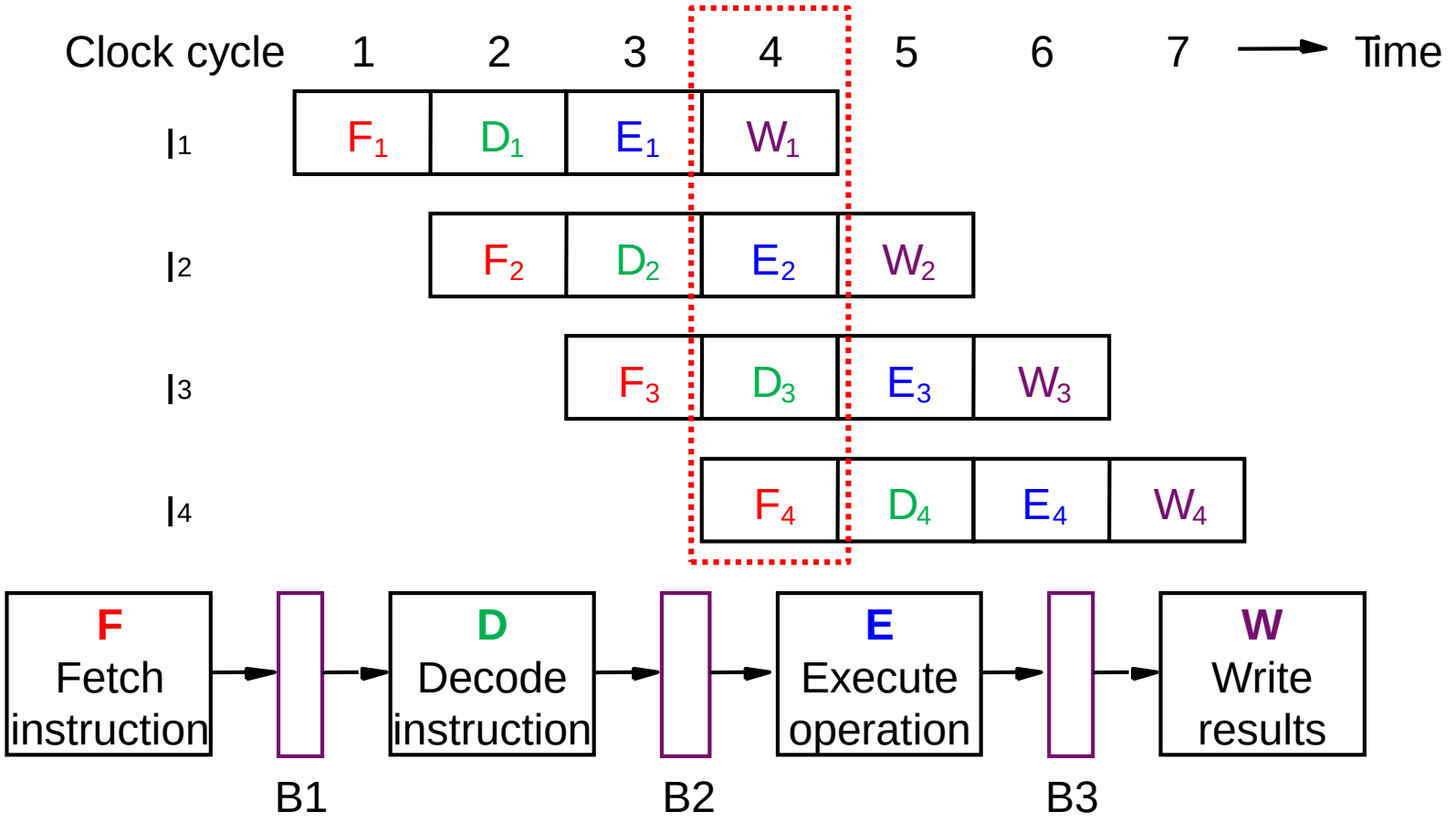
4-Stage Pipeline (1/2)



Class Exercise 12.1

Student ID: _____ Date:
Name: _____ _____

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



4-Stage Pipeline (2/2)

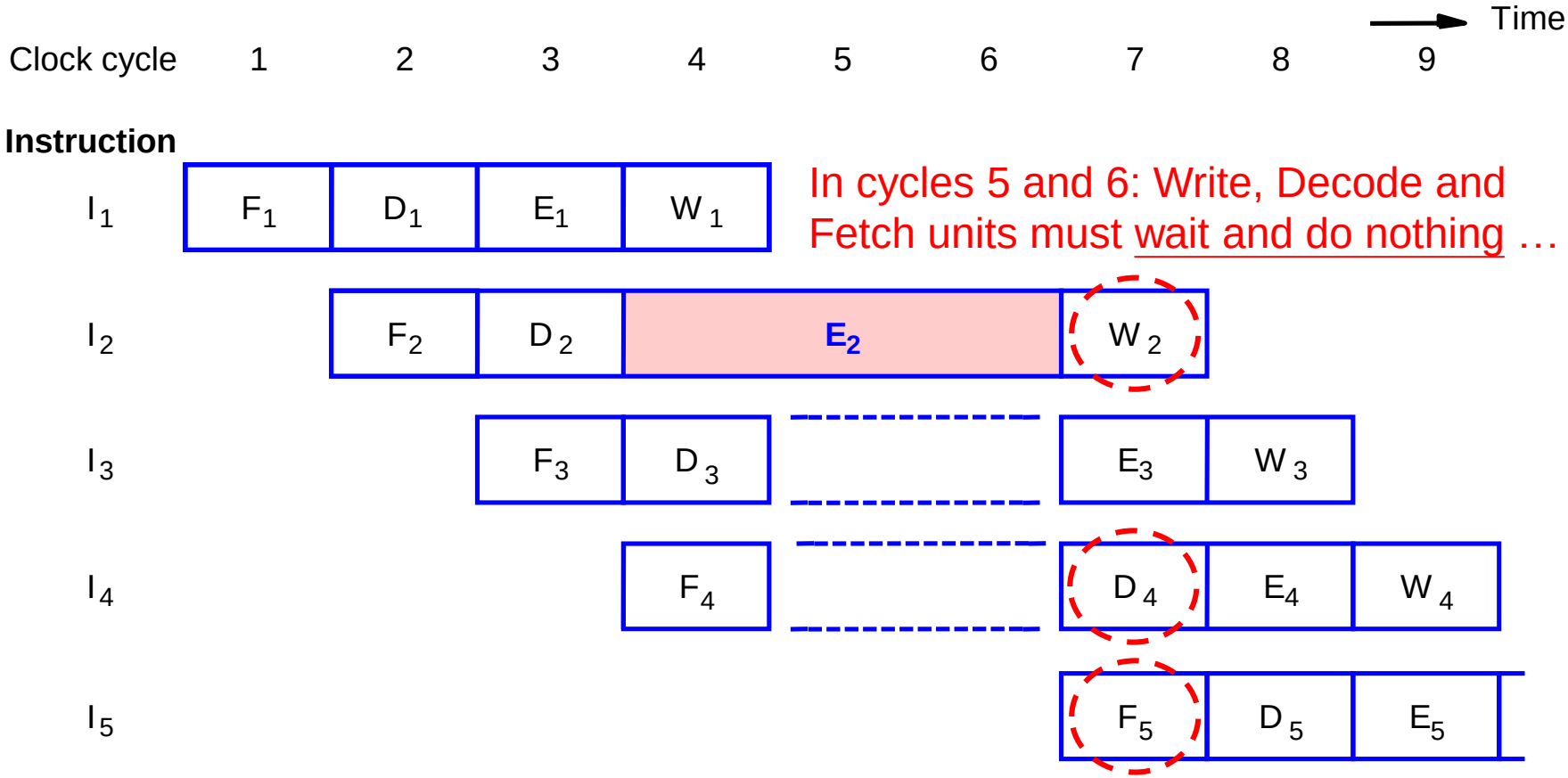
- The four hardware units perform their tasks simultaneously without interfering others.
  - The required information is passed from one unit to the next through a interstage buffer.
- Each stage should be roughly the same maximum clock period.
  - Why? A unit that completes its task early is idle for the remainder of the clock period.
- *Question: What is the ideal speedup of an N-stage pipeline compared to the sequential execution?*

Outline

- Sequential Execution vs Pipelining
- Pipeline Stall: Hazard
  - Data Hazard
  - Instruction Hazard
  - Structural Hazard
- Superscalar Operation

Reality: Pipeline may Stall

- If a pipeline stage requires more than 1 cycle, others have to wait (pipeline stalled)
  - E.g. E_2 requires three cycles to complete



Stall & Hazard

- **Hazard**: Any condition that causes pipeline to stall.
- Another example: A cache miss occurs in F_2:

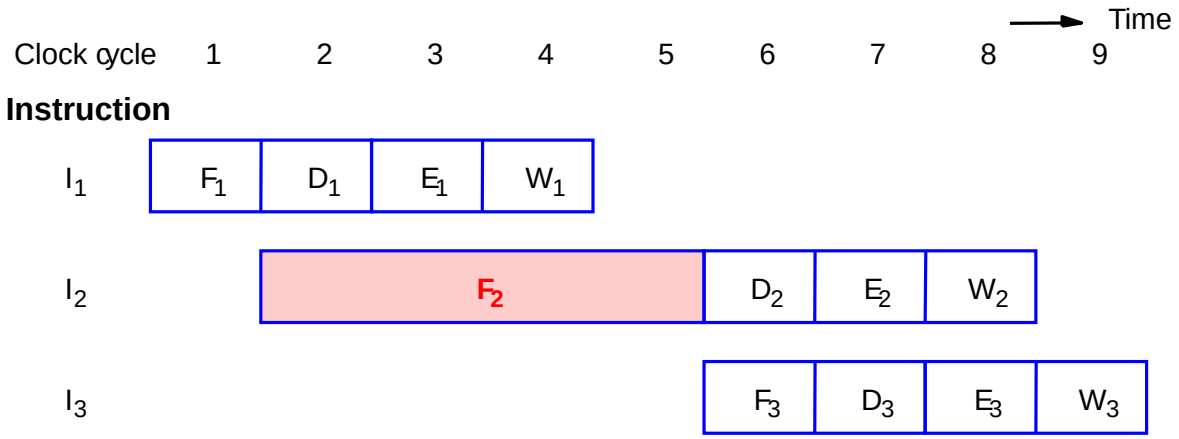


Figure: Instruction execution steps in successive clock cycles.

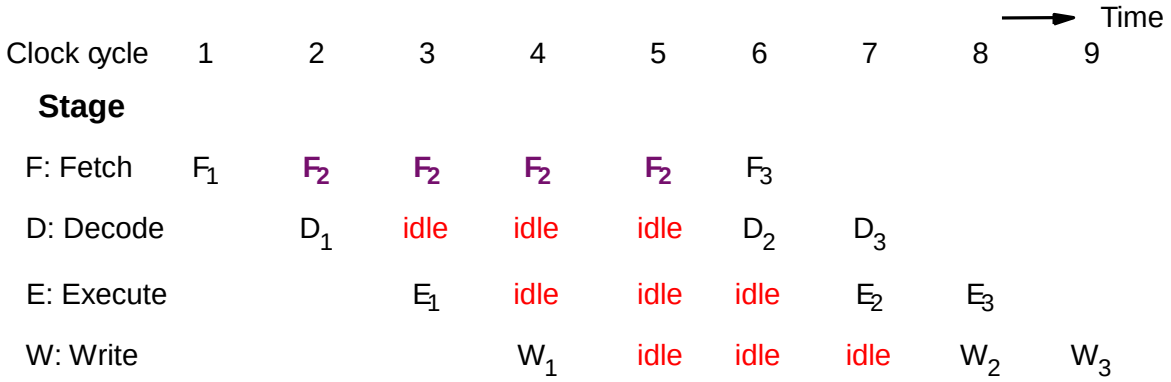


Figure: Statuses of processor stages in successive clock cycles.

Types of Hazards

- **Data Hazard**
  - Either the source or the destination operands of an instruction are not available when required.
- **Instruction Hazard**
  - A delay in the availability of an instruction (this may be a result of a miss in the cache).
- **Structural Hazard**
  - Two instructions require the use of a given hardware resource at the same time.

Outline

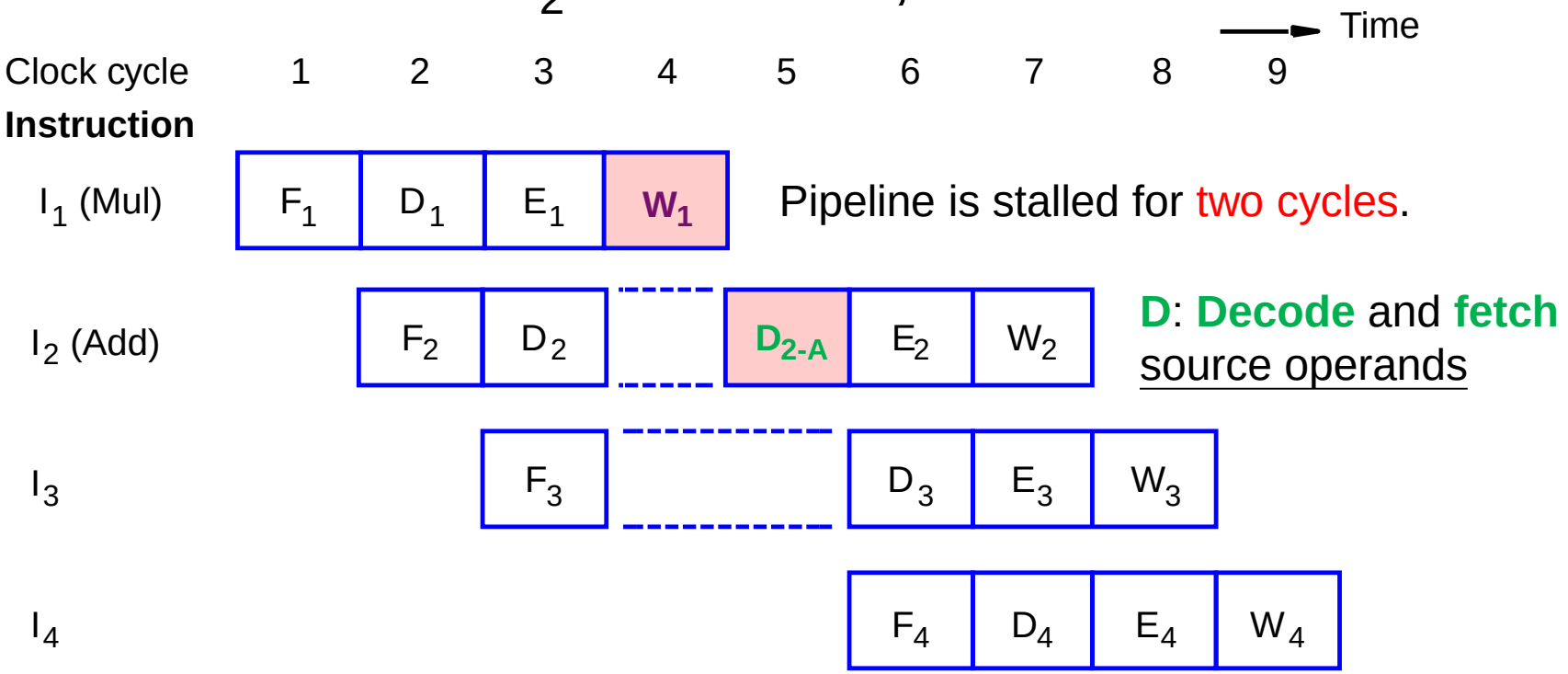
- Sequential Execution vs Pipelining
- Pipeline Stall: Hazard
  - Data Hazard
  - Instruction Hazard
  - Structural Hazard
- Superscalar Operation

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.

Class Exercise 12.2

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

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

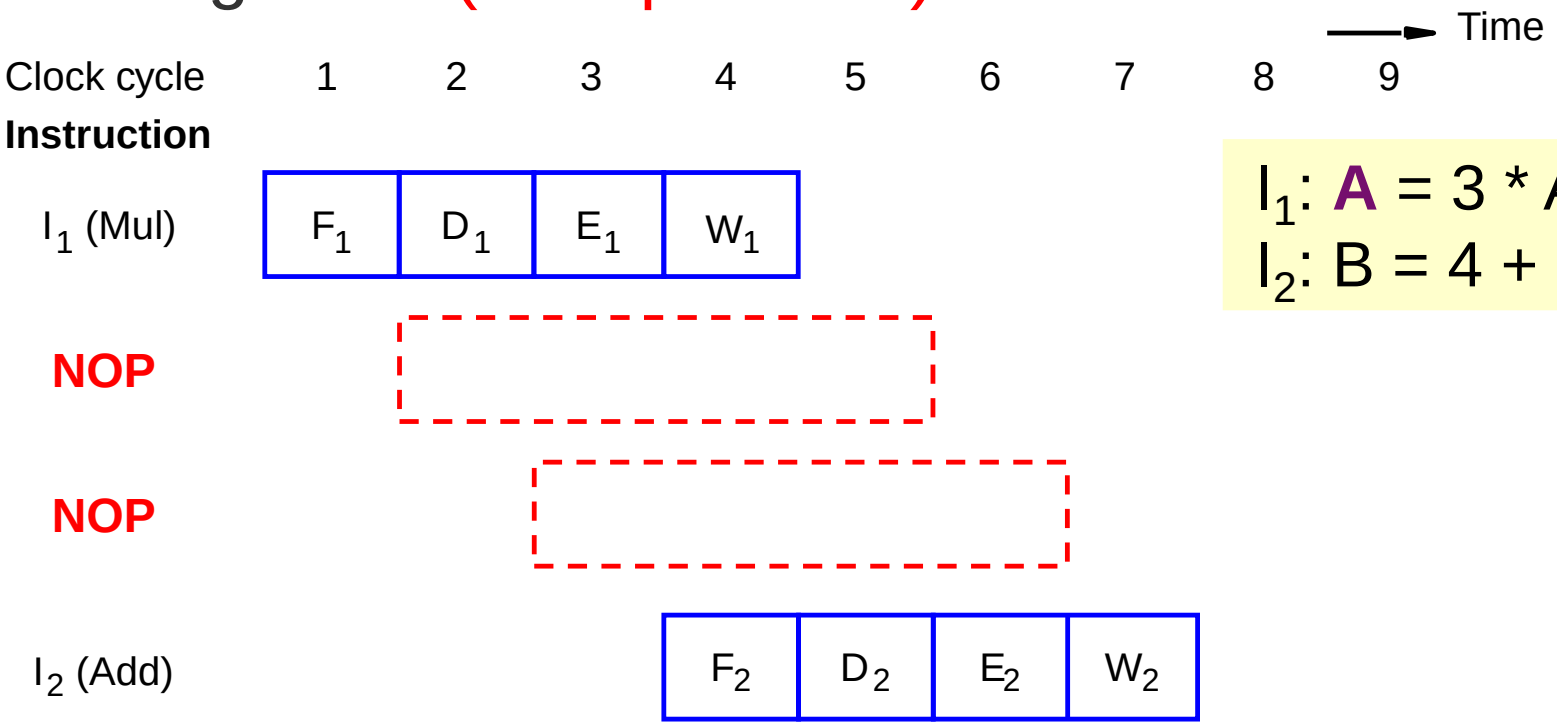
I_2: B = 20 + **C**;

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.



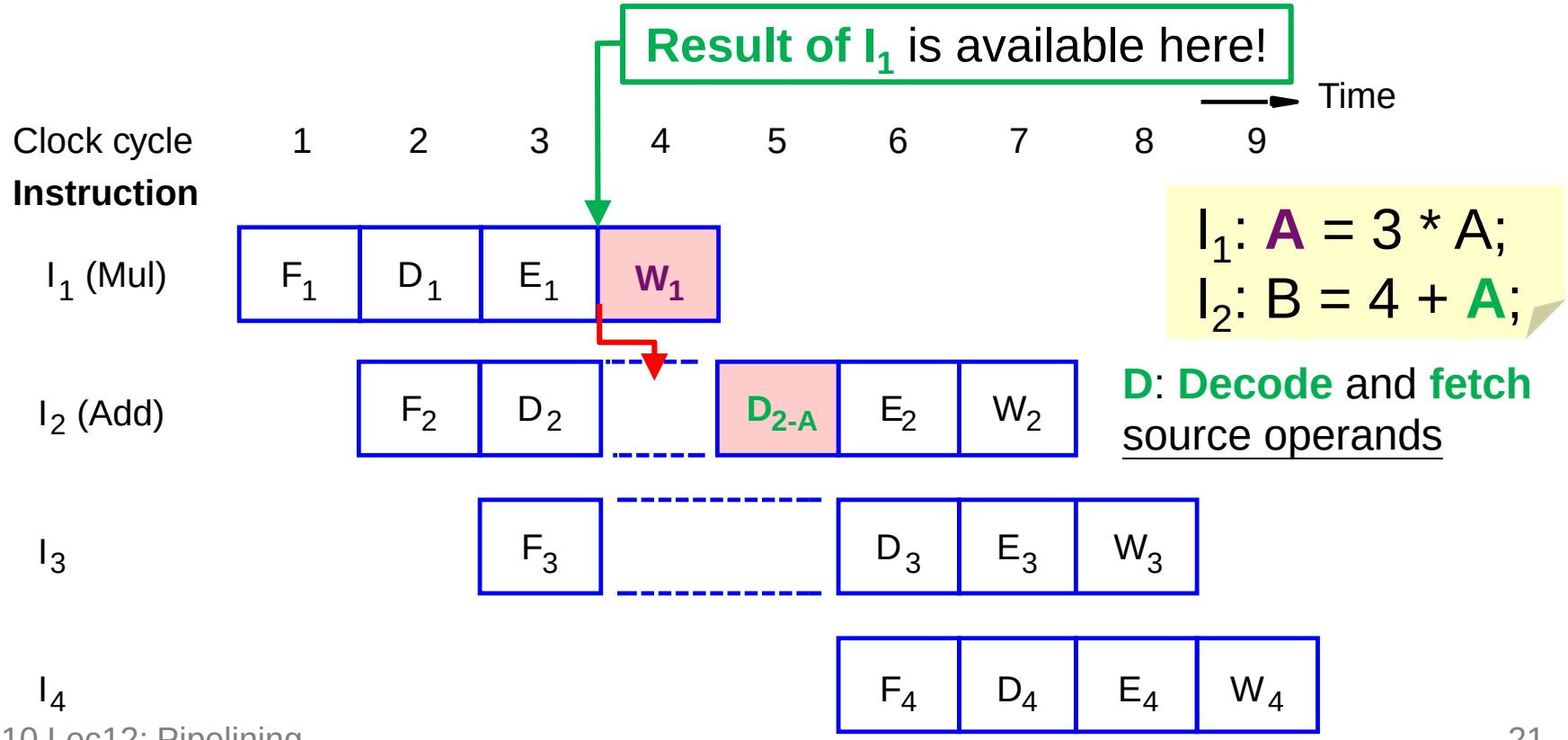
I_1: **A** = 3 * A;
I_2: B = 4 + **A**;

  - Advantage: Simpler hardware, less cost
  - Disadvantage: Larger code size, less flexibility, and reduced performance

Question: Do we really avoid the pipline stalling?

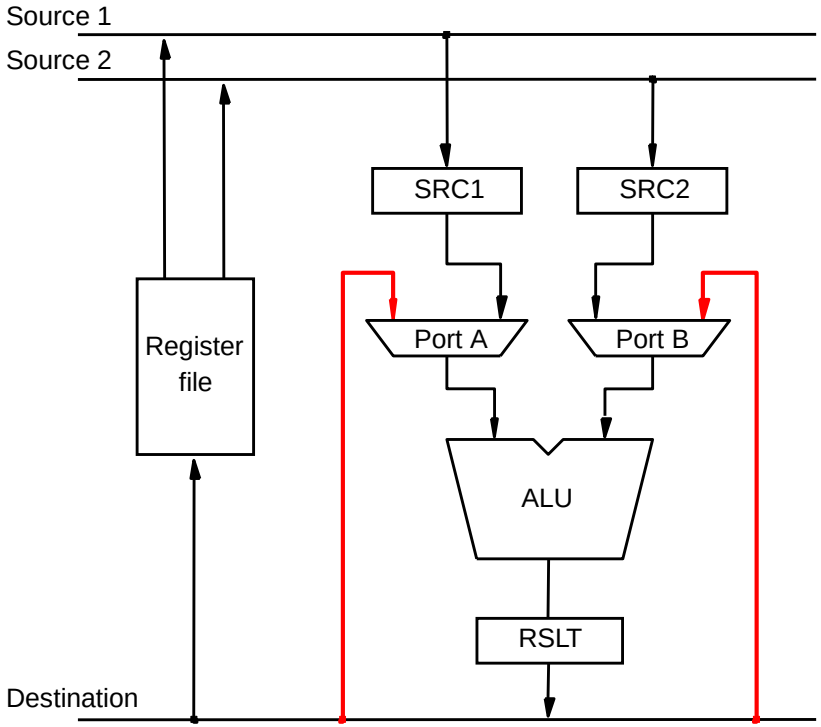
Hardware Solution to Data Hazard (1/2)

- The data hazard arises because I_2 is waiting for data to be written in the register **A**.
- In fact, the result of I_1 is available at the output of ALU.
- Delay is reduced if the result can be forwarded to E_2.

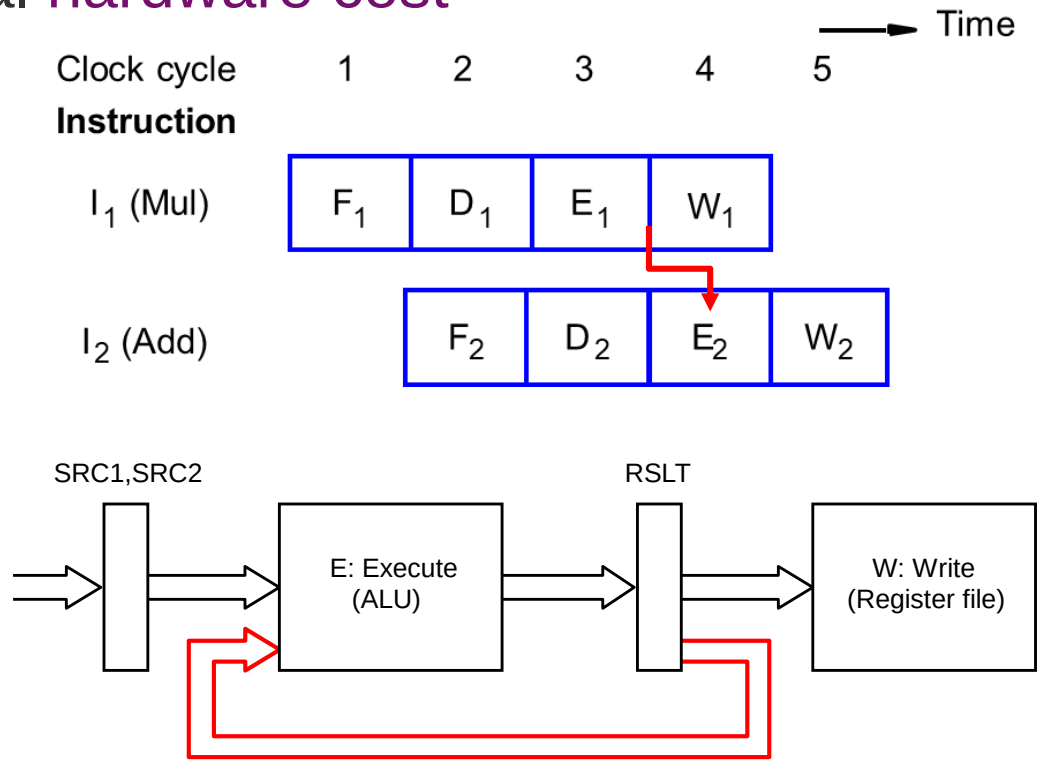


Hardware Solution to Data Hazard (2/2)

- **Operand Forwarding**: By introducing the forwarding path, the execution of I_2 can proceed without stalling.
  - Disadvantage: Additional hardware cost



(a) Datapath (3 buses)



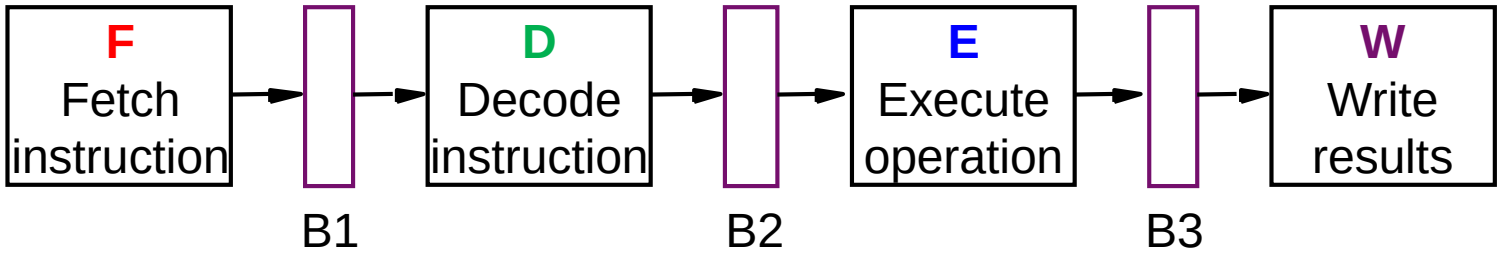
(b) Source and result registers

Outline

- Sequential Execution vs Pipelining
- Pipeline Stall: Hazard
  - Data Hazard
  - Instruction Hazard
  - Structural Hazard
- Superscalar Operation

Instruction Hazard

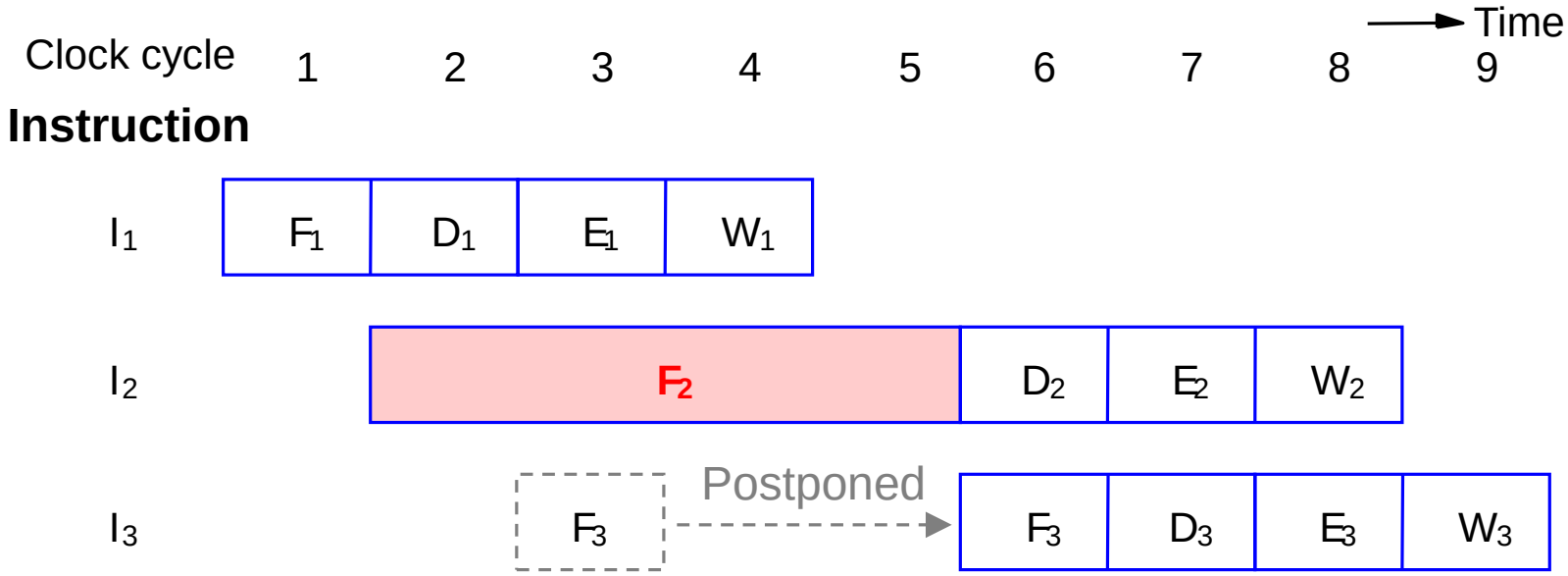
- Recall: The purpose of the instruction fetch unit is to supply the execution units with instructions.



  - **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.
  1) **Cache miss**
  2) **Brach instruction** *(both unconditional and conditional)*

Instruction Hazard: Cache Miss

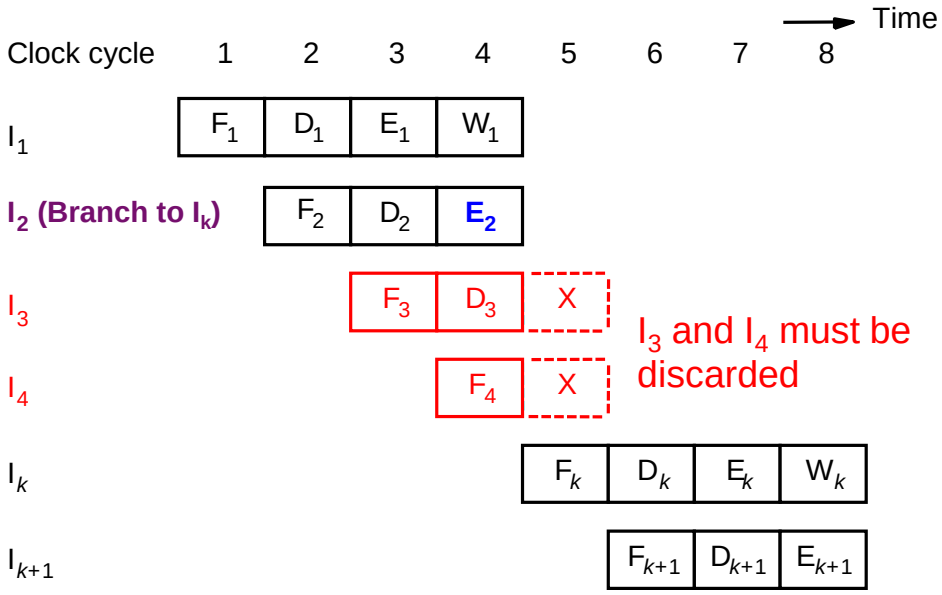
- The effect of a cache miss on the pipelined operation is as follows:



  - 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.

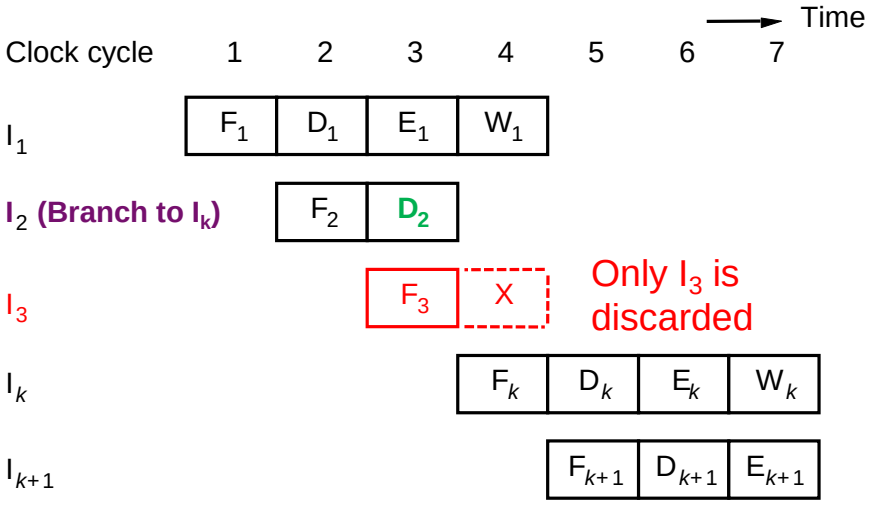
Instruction Hazard: Unconditional Branch

- 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.



Branch address computed in **Execute stage**

Branch Penalty: **2** clock cycles

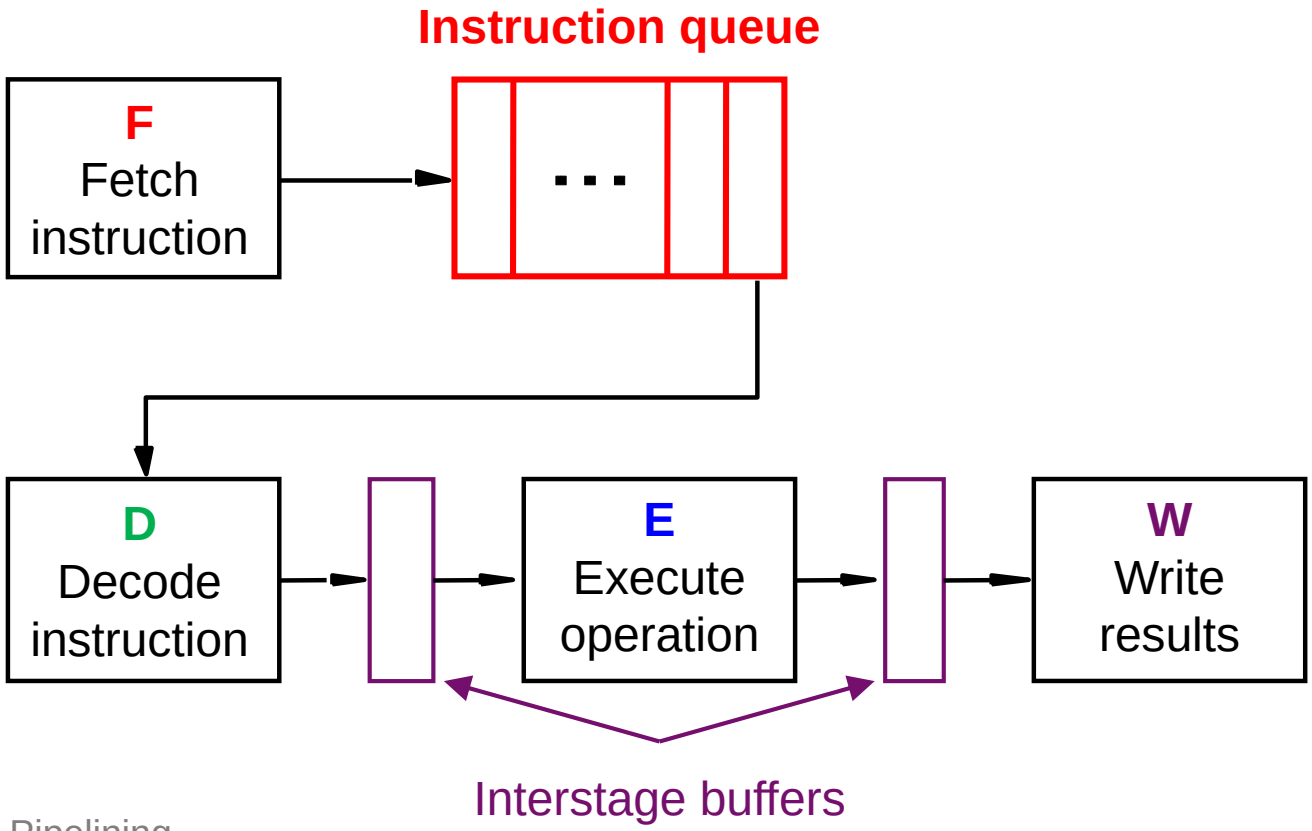


Branch address computed in **Decode stage**

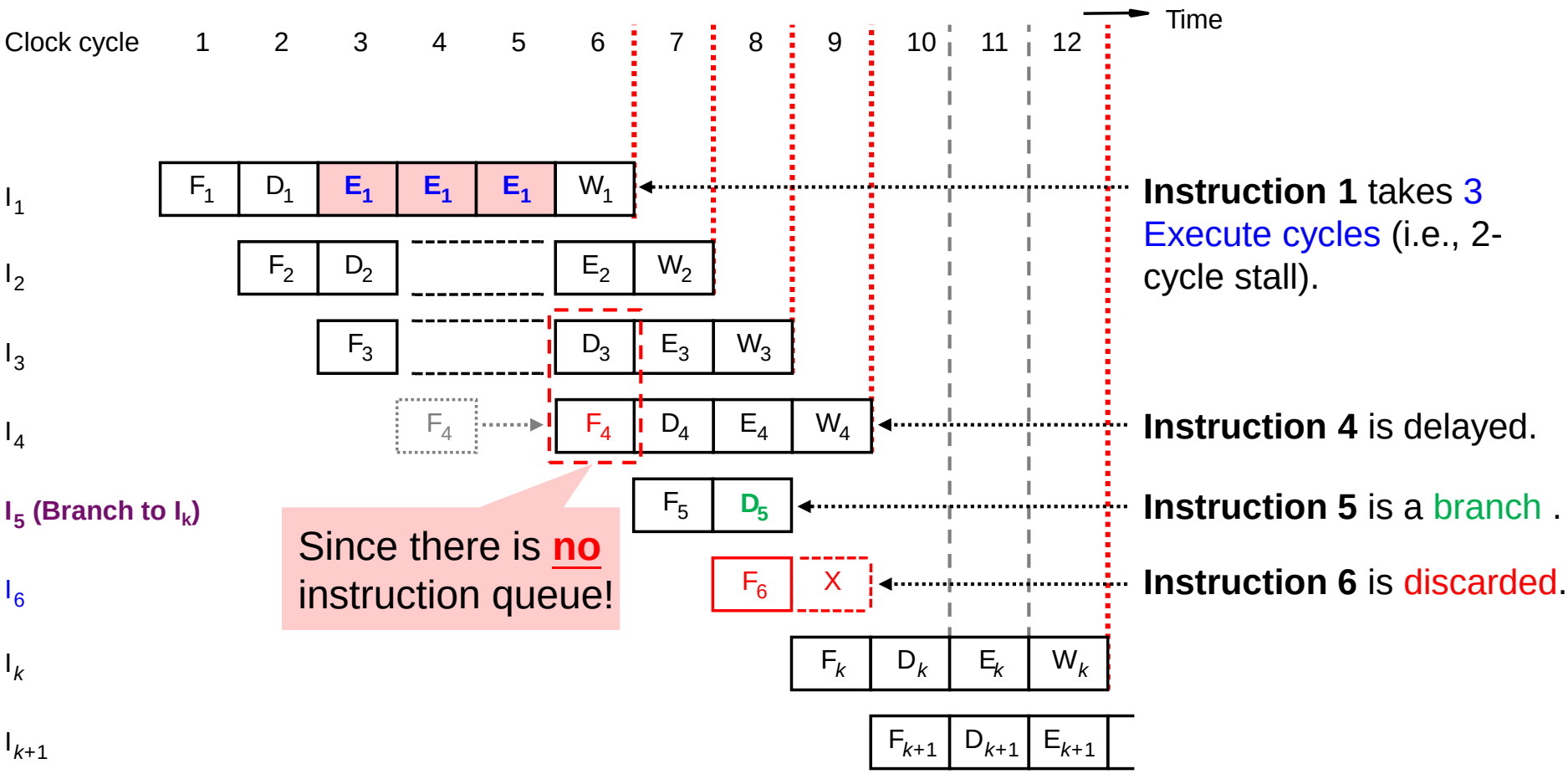
Branch Penalty: **1** clock cycle

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.

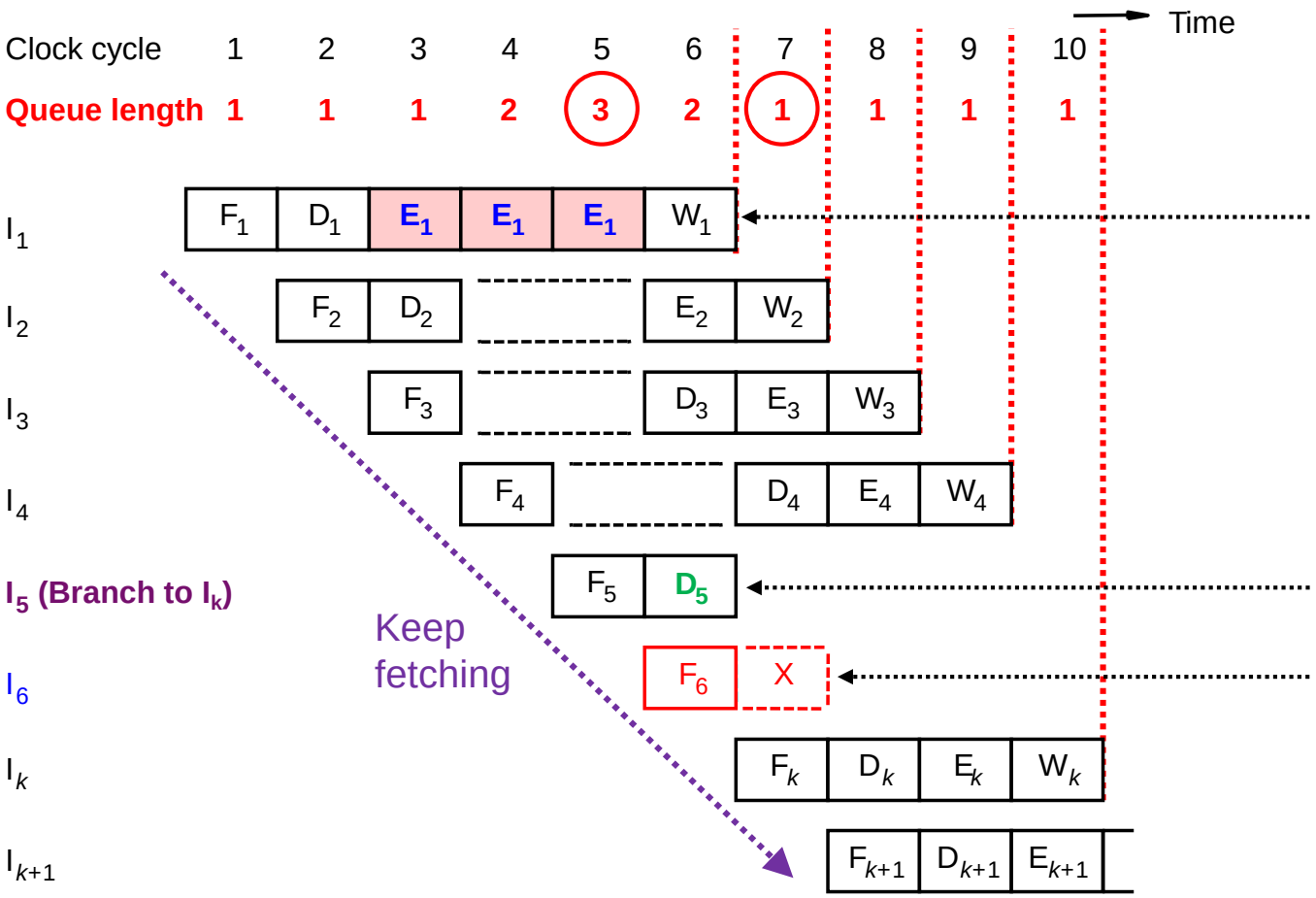


Example: Without Instruction Queue



- F_4, F_5, F_6, F_k, and F_{k+1}, are delayed.
- 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 .

Instruction 6 is discarded, after taking Branch.
The queue length drops to 1 before cycle 8.

- I_6 is still being discarded, but the instruction queue can avoid delaying F_4, F_5, F_6, F_k, and F_{k+1} if the queue is not empty.
- I_1, I_2, I_3, I_4, and I_k can complete in successive cycles.

Without vs With Instruction Queue

- With instruction queue, the branch instruction does not increase the overall execution time (*if the queue is not empty*).
  - Since instructions can complete in successive clock cycles.

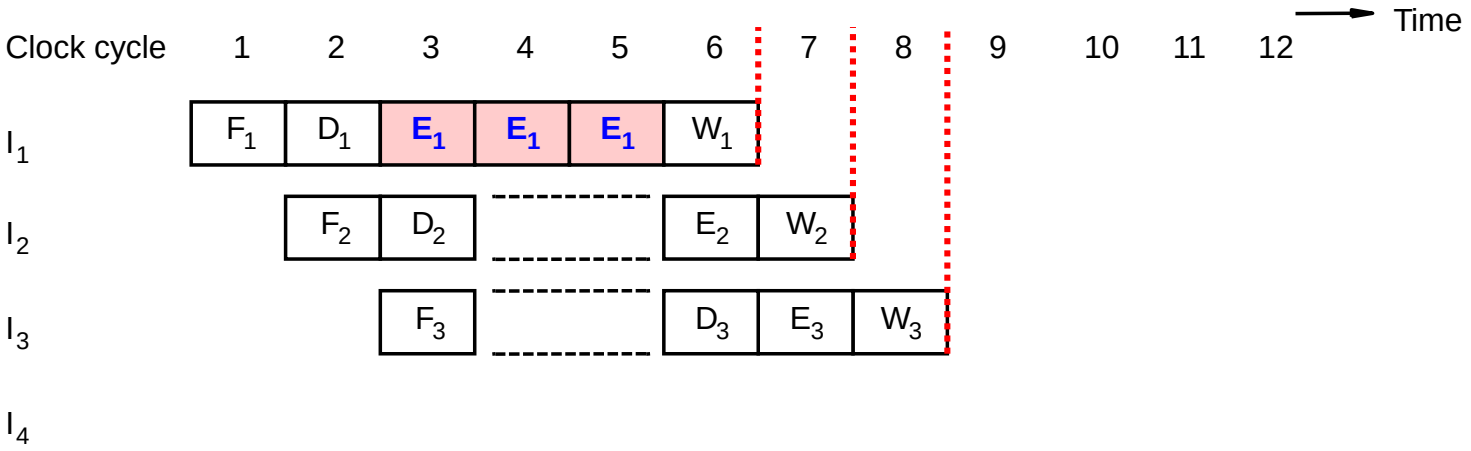
- Branch address is computed in parallel with other instructions, so no cycles lost due to branch.
  - This is called branch folding.

- Instruction queue is also possible to hide the effect of cache miss (*if the queue is not empty*).

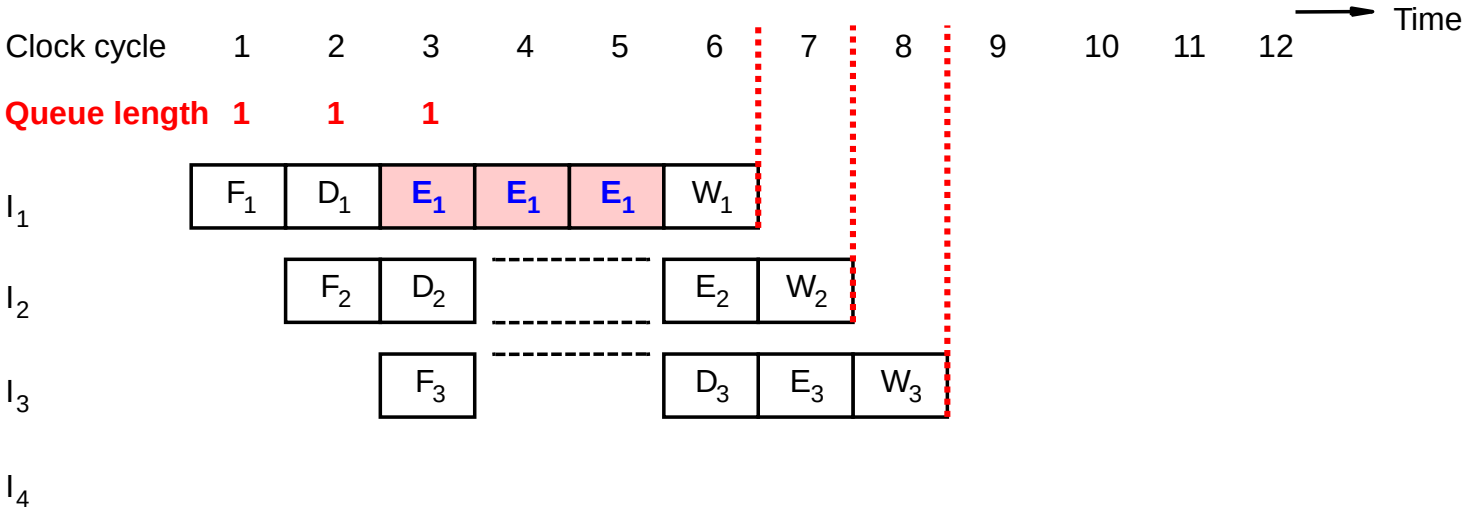
Class Exercise 12.3

- Please show how instruction queue can hide the effect of cache miss (three cycles) caused by F_4.

Without Instruction Queue



With Instruction Queue

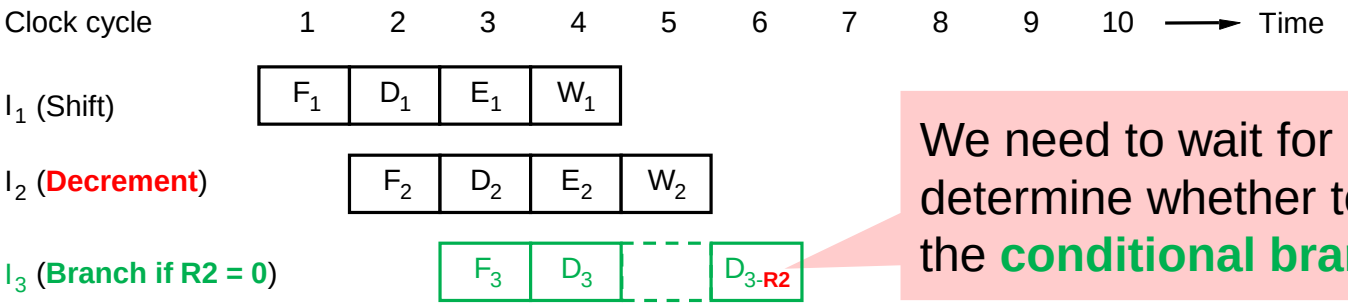


Instruction Hazard: Conditional Branch

- Branch folding is not working for conditional branches.
- Conditional branches may result in added hazard.
  - Since the condition is based on the preceding instruction.
- Example:

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

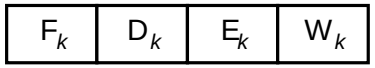
R2 is used as the **branch condition**.



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

All intermediate instructions must be discarded …

LOOP I_k



Solution 1) Delayed Branch (1/2)

- The location following a branch instruction is called a branch delay slot.

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

(a) Original program loop

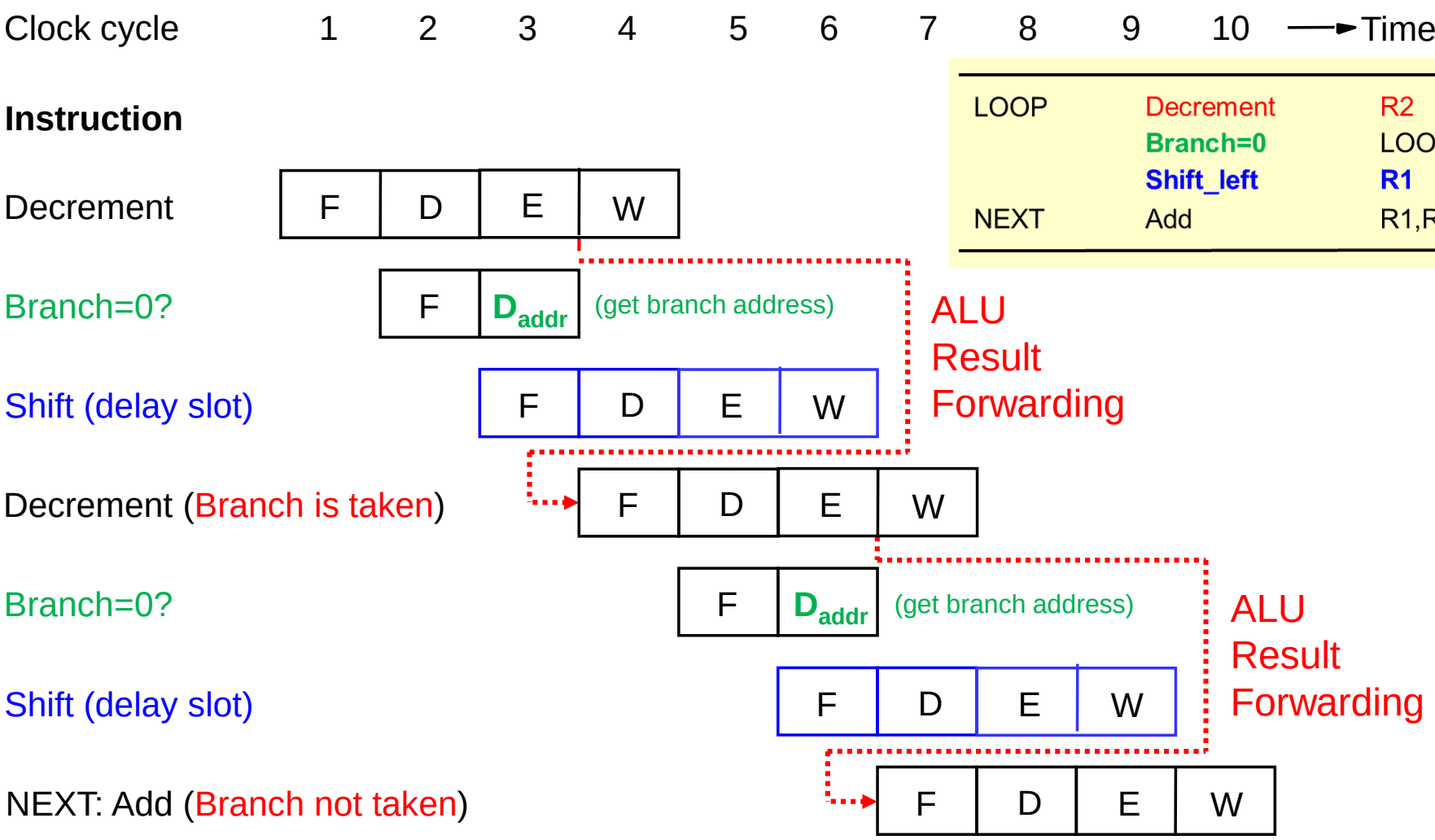
- **Delayed branching** can minimize the penalty by
  - Placing useful instructions in branch delay slots, and
  - Internally re-ordering the instructions.

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

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

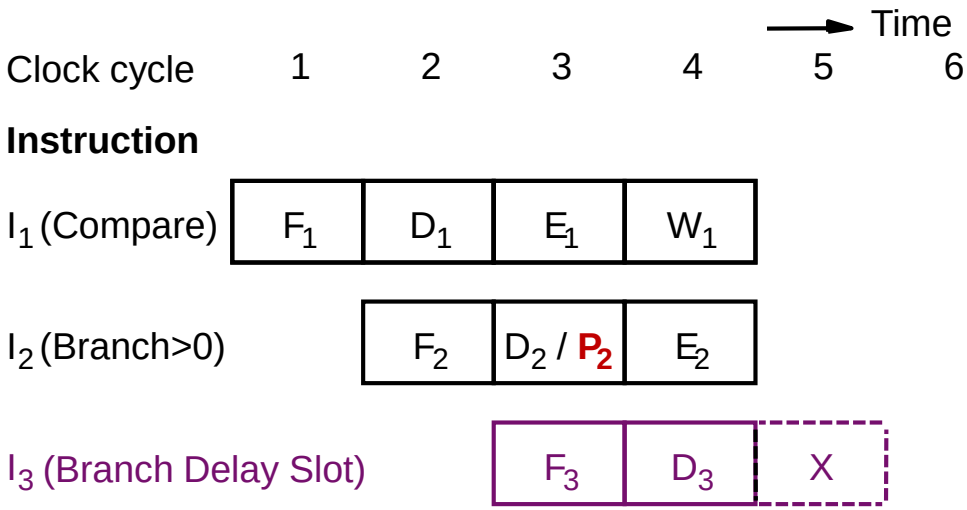
Solution 1) Delayed Branch (2/2)

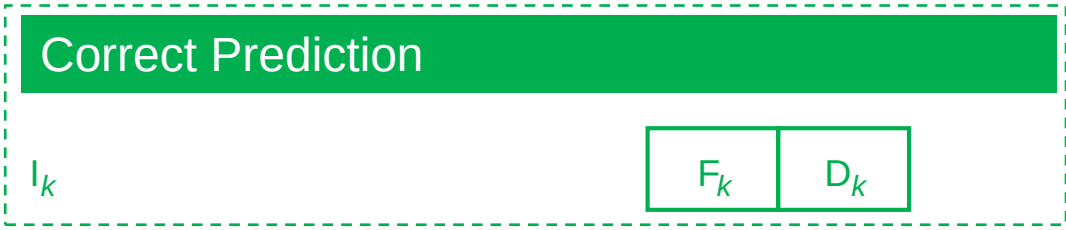
- Delayed branching can minimize the branch penalty.

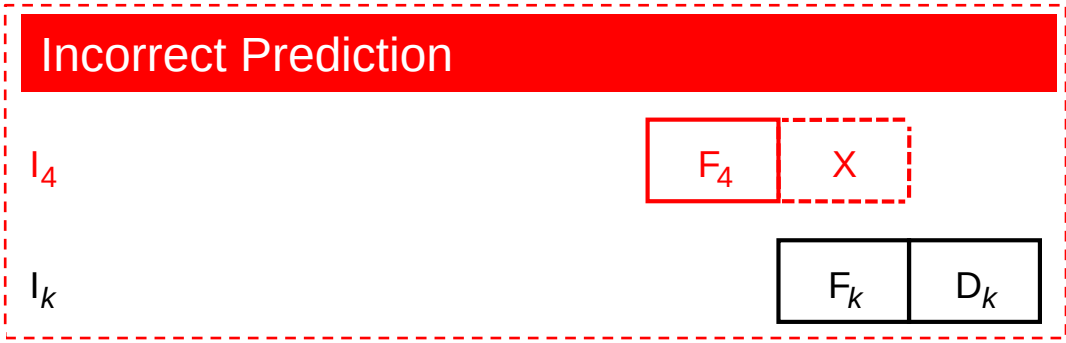


Solution 2) Branch Prediction (1/2)

- 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

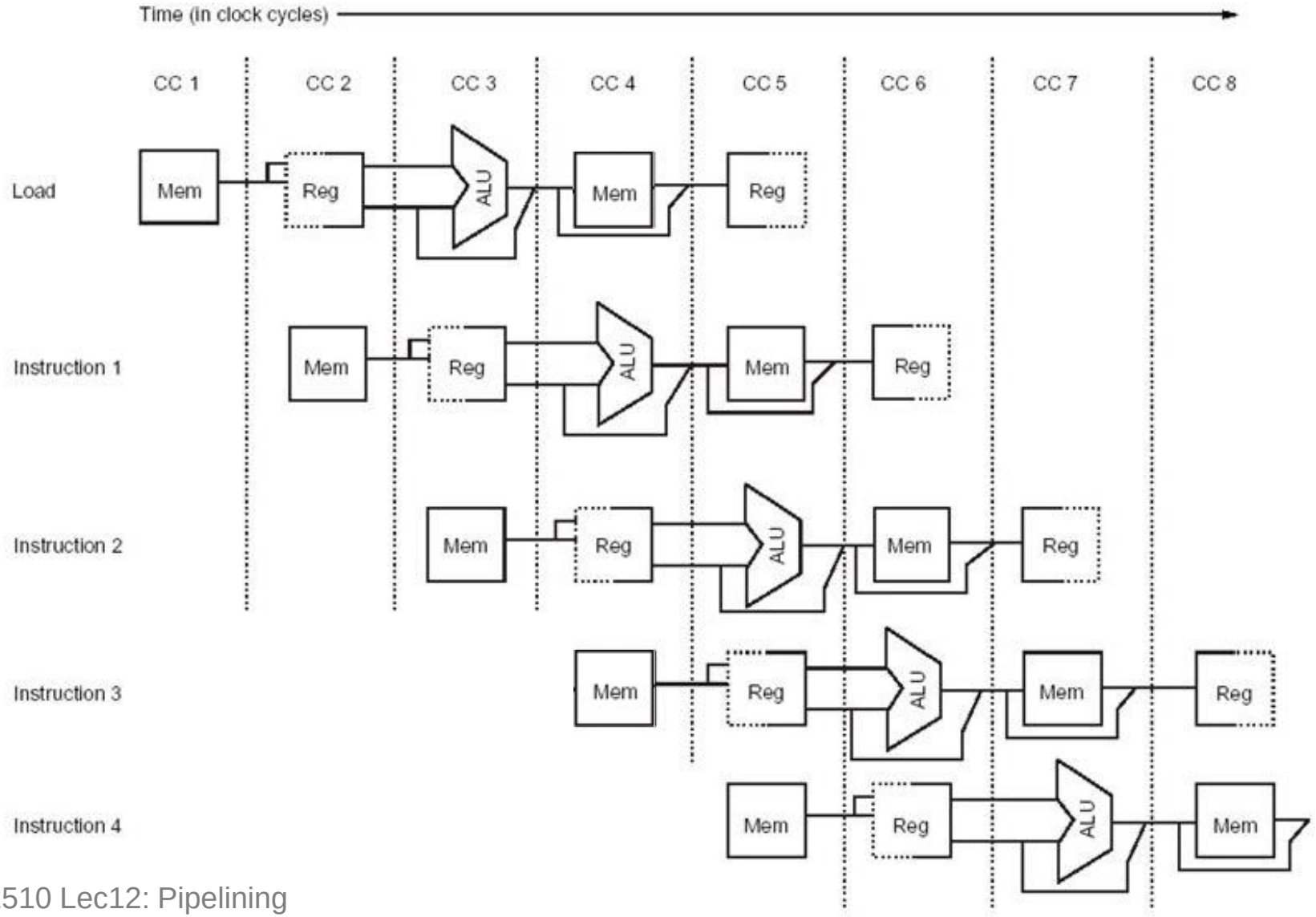
- Sequential Execution vs Pipelining
- Pipeline Stall: Hazard
  - Data Hazard
  - Instruction Hazard
  - Structural Hazard
- Superscalar Operation

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 ($$$).

Class Exercise 12.4

- What is the cause of the following structure hazard?

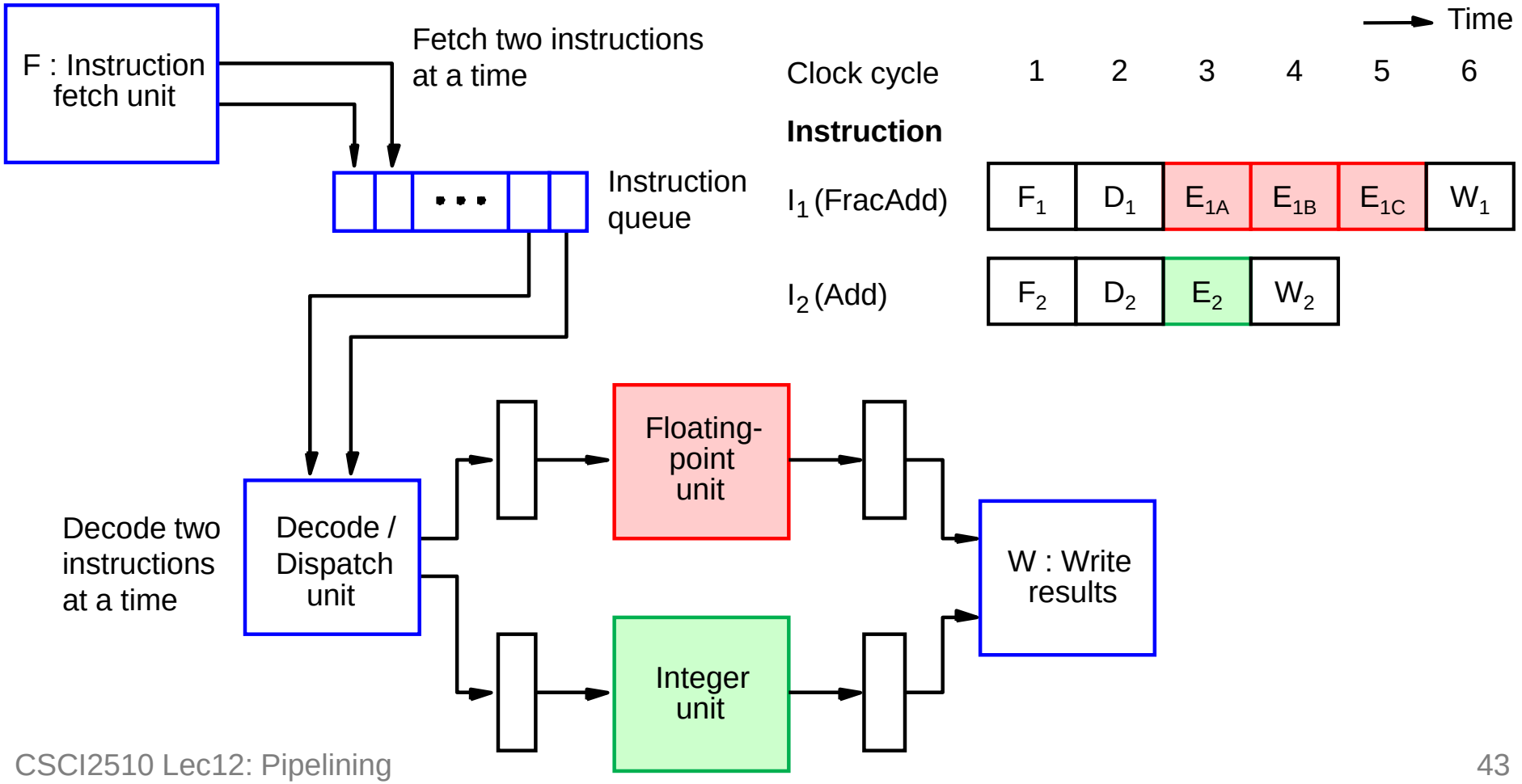


Outline

- Sequential Execution vs Pipelining
- Pipeline Stall: Hazard
  - Data Hazard
  - Instruction Hazard
  - Structural Hazard
- Superscalar Operation

Superscalar Operation (1/2)

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



Superscalar Operation (2/2)

- 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 is also a common technique to make use of instruction cycles by re-ordering instructions.
  - E.g., Delayed branching reorders the instructions to minimize the branch penalty.

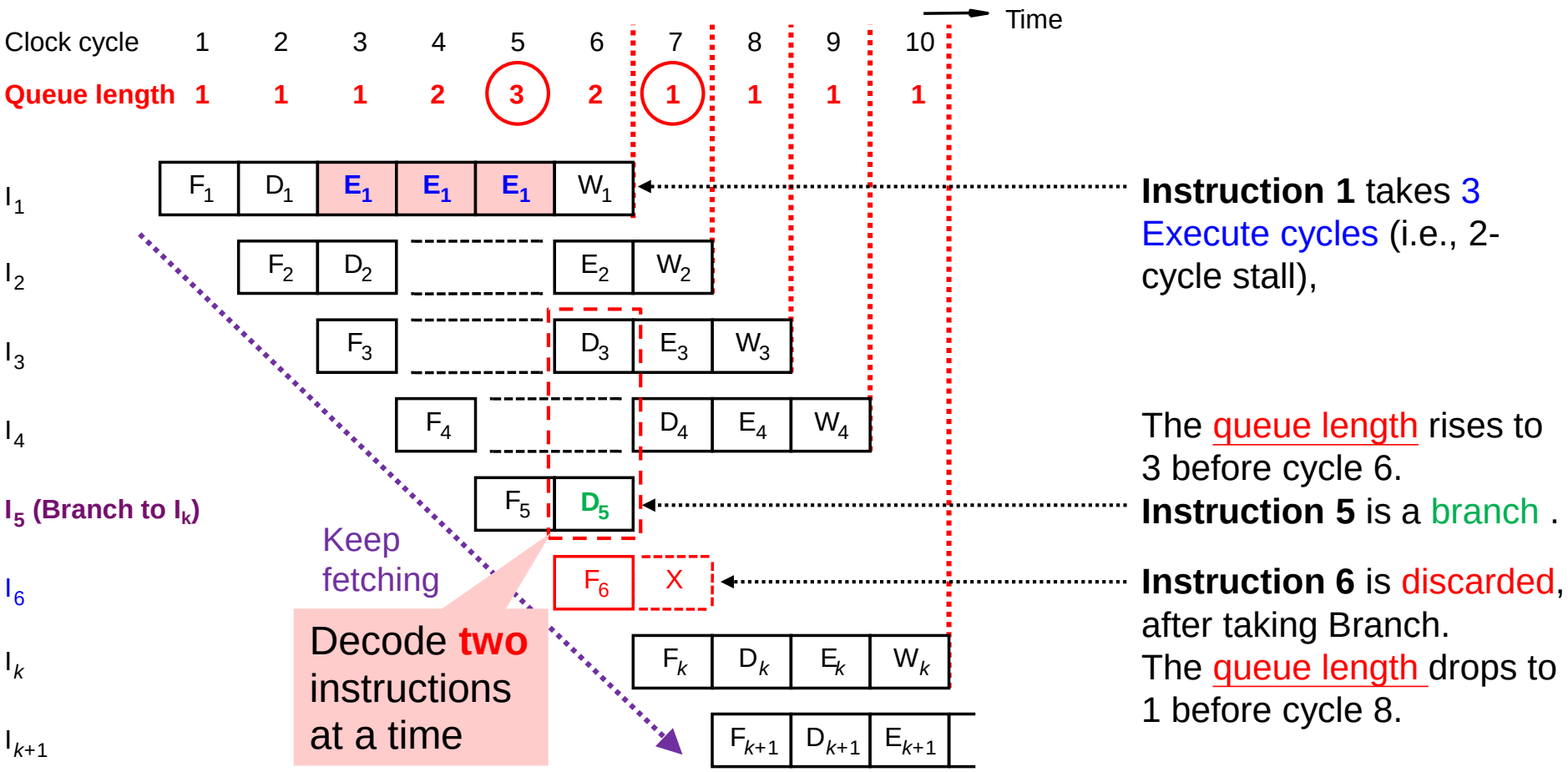
Out-of-Order Execution

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

- 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.
- 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 */
```

Recall: With Instruction Queue



- I_6 is still being discarded, but the instruction queue can avoid delaying F_4, F_5, F_6, F_k, and F_{k+1} if the queue is not empty.
- I_1, I_2, I_3, I_4, and I_k can complete in successive cycles.

Summary

- Sequential Execution vs Pipelining
- Pipeline Stall: Hazard
  - Data Hazard
  - Instruction Hazard
  - Structural Hazard
- Superscalar Operation