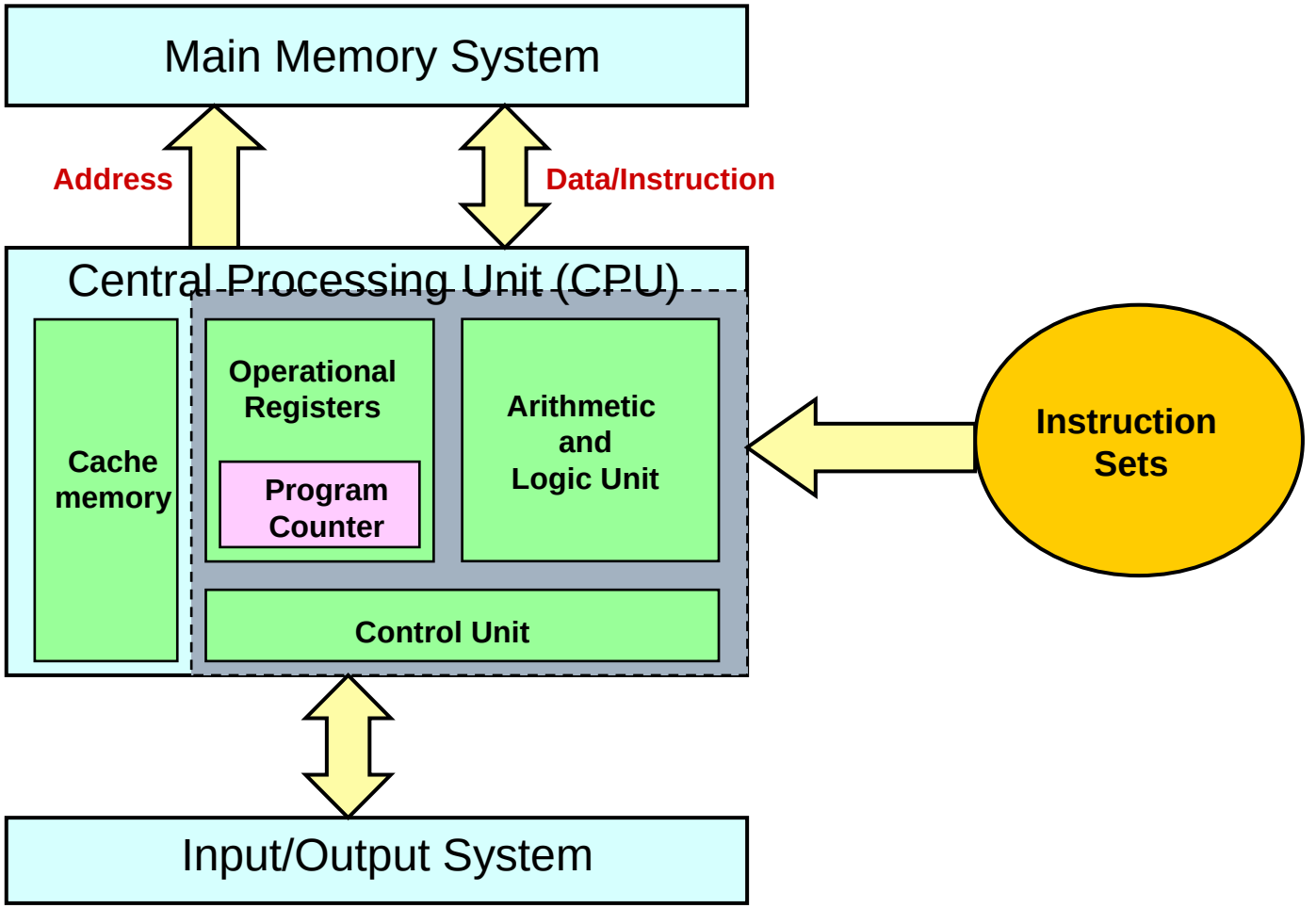
Chapter 9
Pipelining

Jin-Fu Li
Department of Electrical Engineering
National Central University
Jungli, Taiwan

Outline

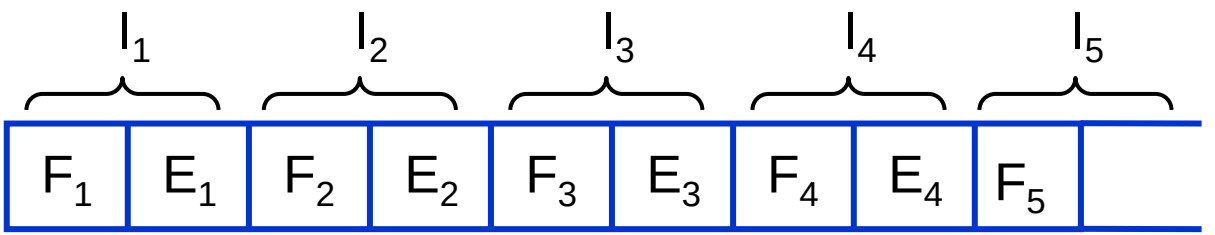
- Basic Concepts
- Data Hazards
- Instruction Hazards

Content Coverage



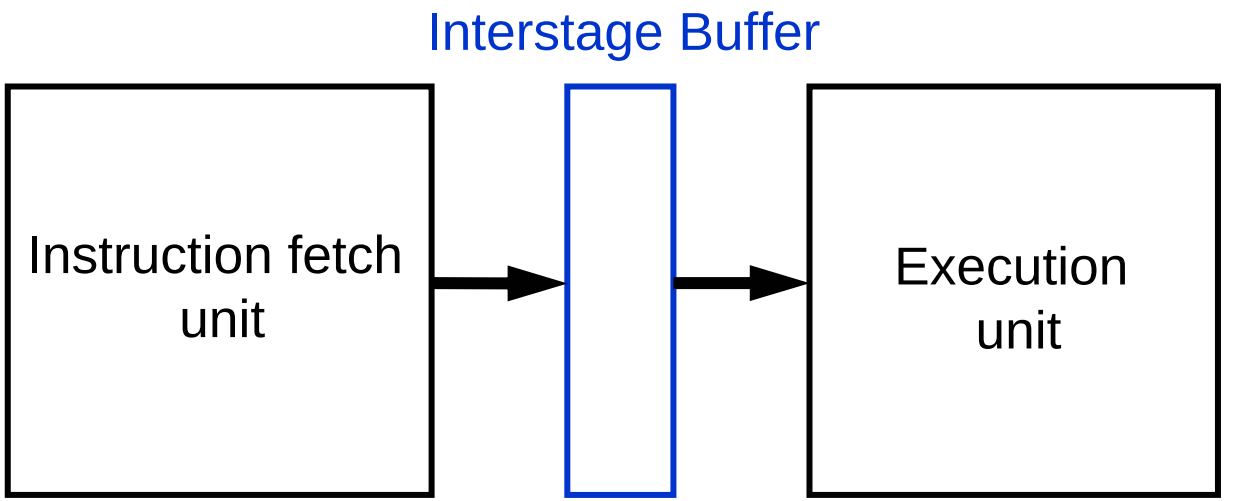
Basic Concepts

- Pipelining is a particularly effective way of organizing concurrent activity in a computer system
- Let F_i and E_i refer to the fetch and execute steps for instruction I_i
- Execution of a program consists of a sequence of fetch and execute steps, as shown below

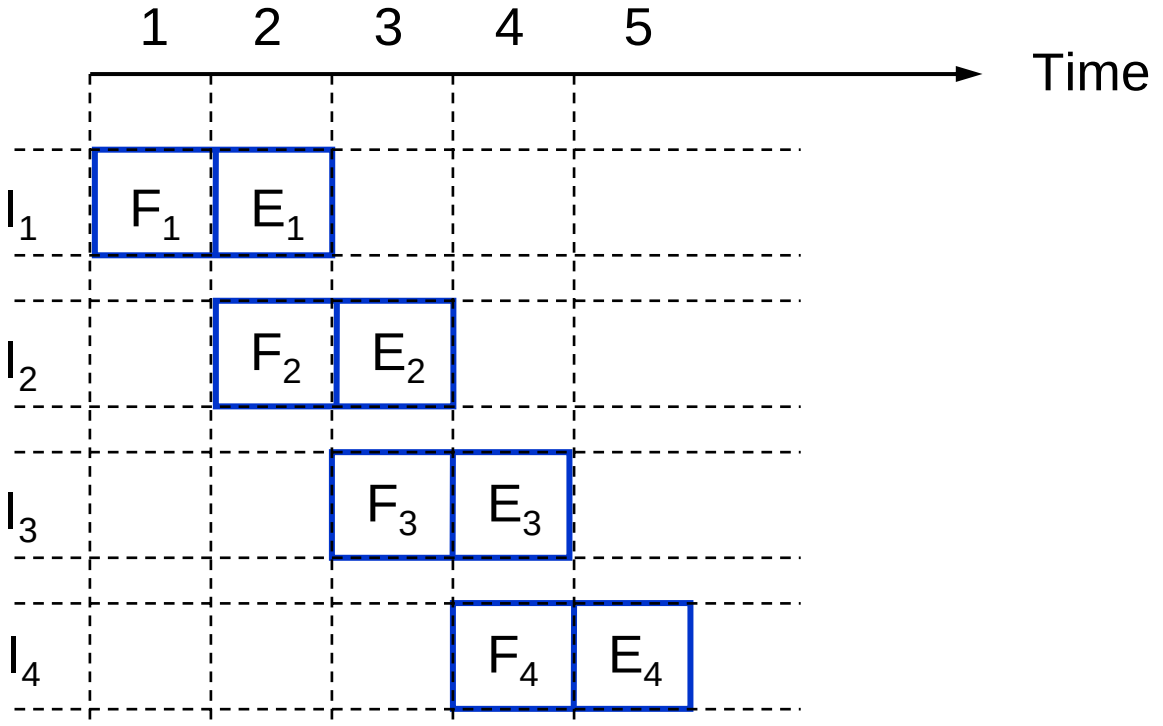


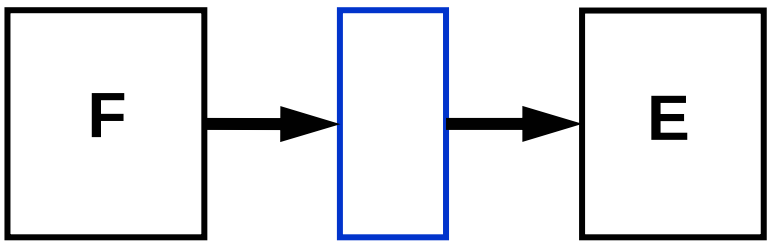
Hardware Organization

- Consider a computer that has two separate hardware units, one for fetching instructions and another for executing them, as shown below

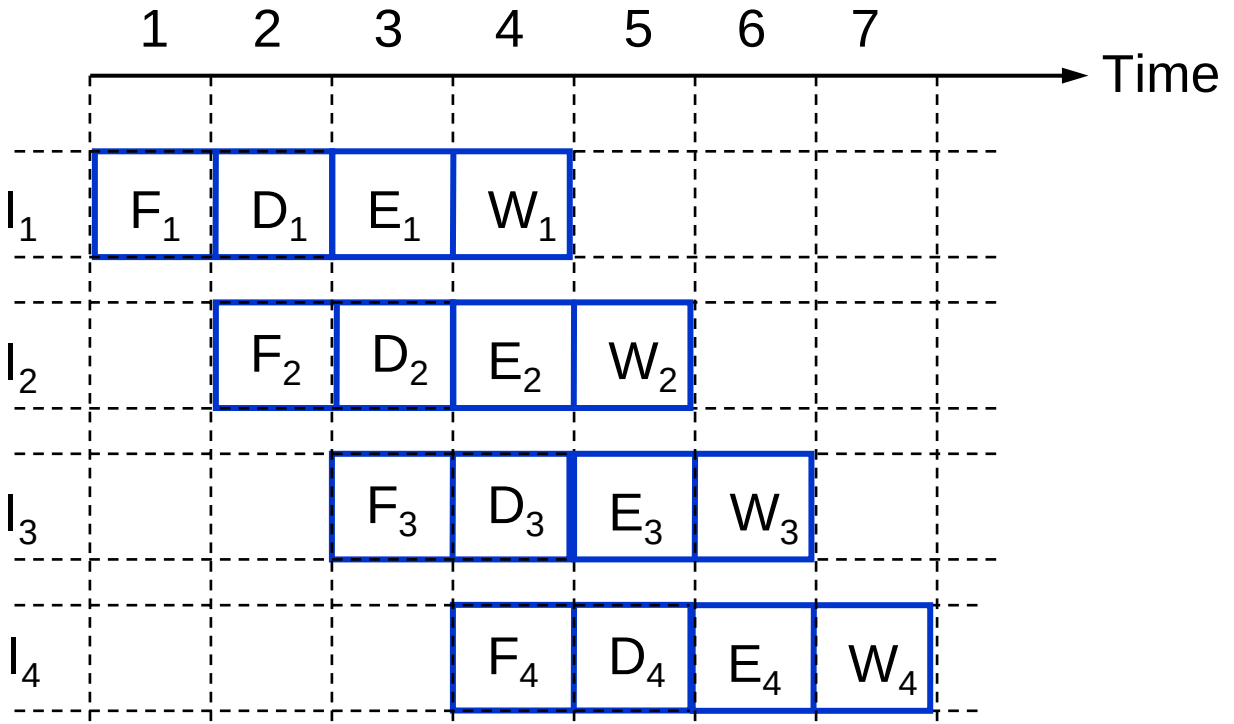


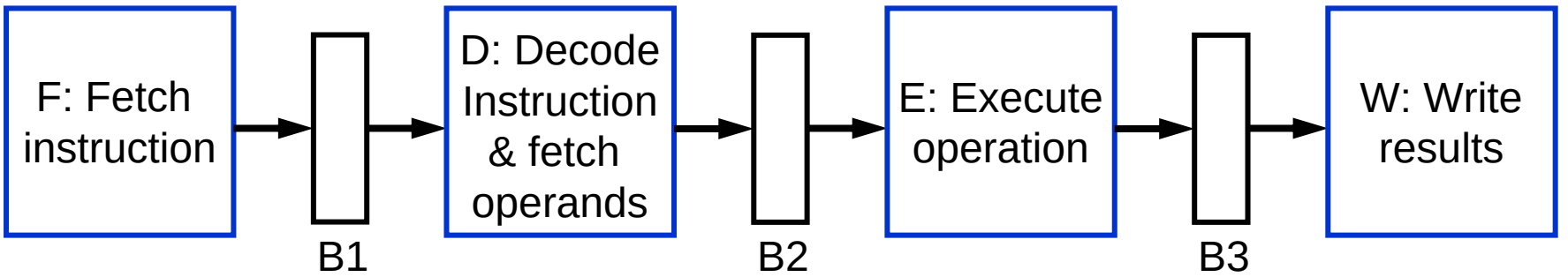
Basic Idea of Instruction Pipelining





A 4-Stage Pipeline

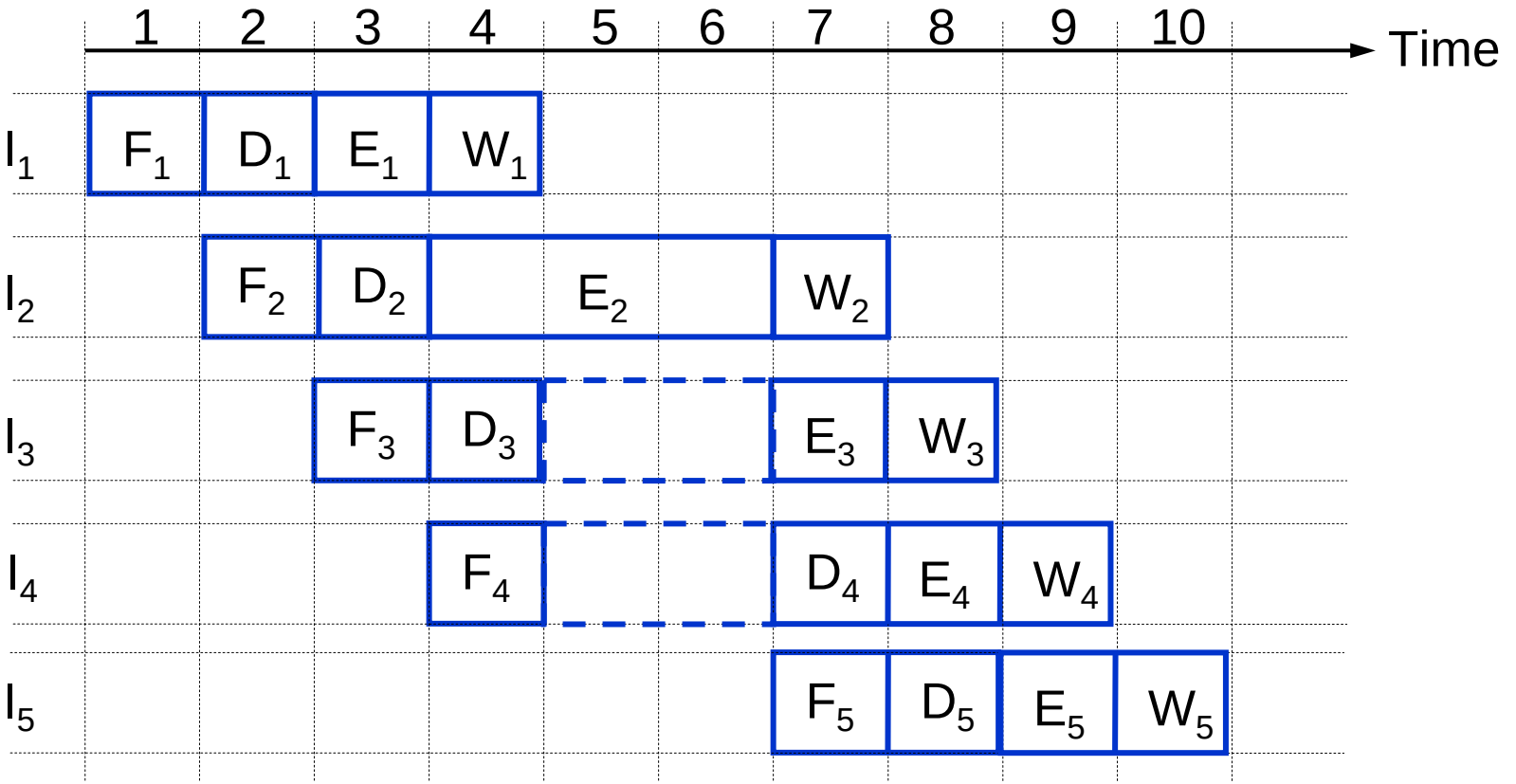




Pipeline Performance

- The pipeline processor show in last slide completes the processing of one instruction in each clock cycle, which means that the rate of instruction processing is four times that of sequential operation.
- The potential increase in performance resulting from pipelining is proportional to the number of pipeline stages.
- However, this increase would be achieved only if pipelined operation could be sustained without interruption throughout program execution

Hazard

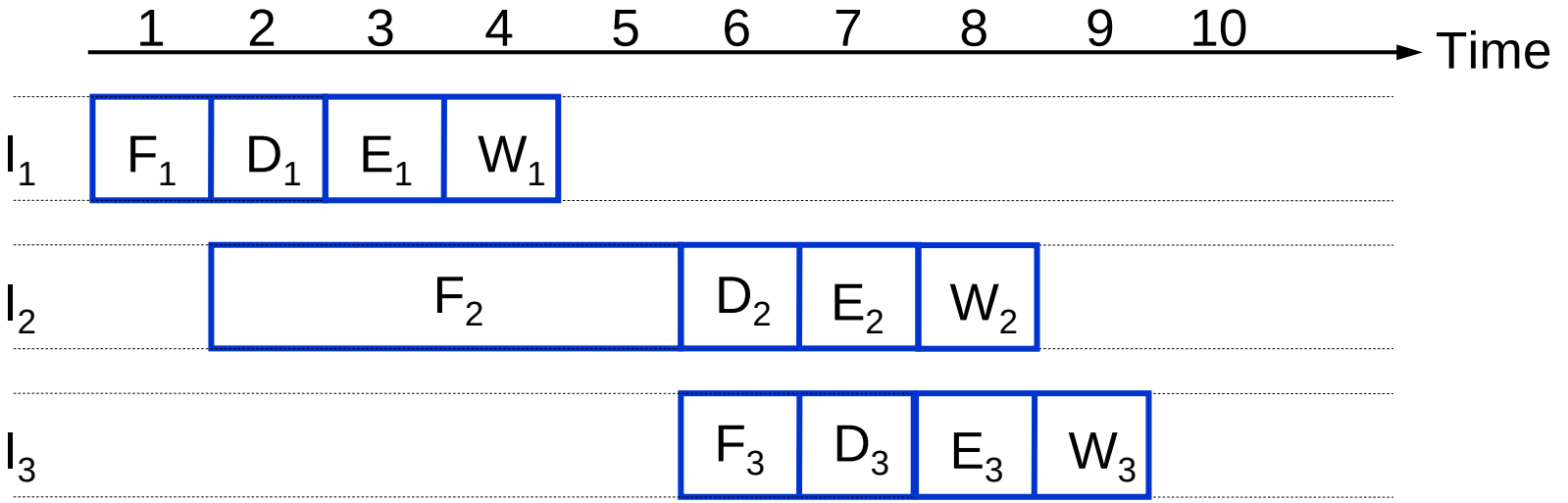


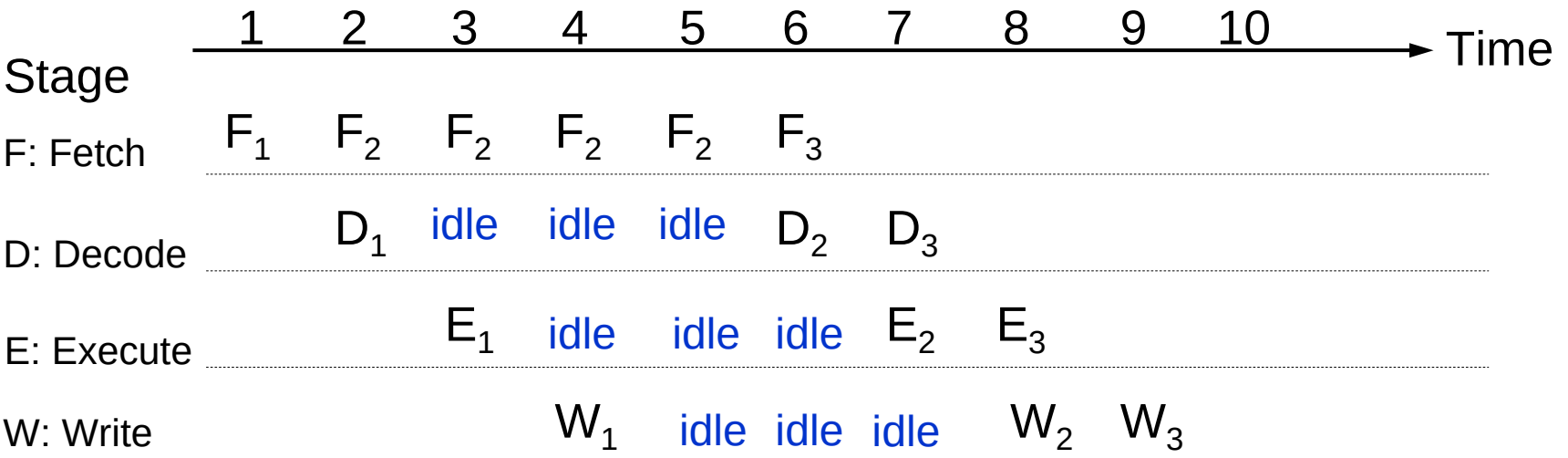
- Pipelined operation in above figure is said to have been stalled for two clock cycles. Any condition that causes the pipeline to stall is called a *harzard*

Data Hazard and Instruction Hazard

- A data hazard is any condition in which either the source or the destination operands of an instruction are not available at the time expected in the pipeline. As a result some operation has to be delayed, and the pipeline stalls
- The pipeline may also be stalled because of a delay in the availability of an instruction. For example, this may be a result of a miss in the cache, requiring the instruction to be fetched from the main memory. Such hazards are often called *control hazards* or *instruction hazards*

An Example of Instruction Hazard



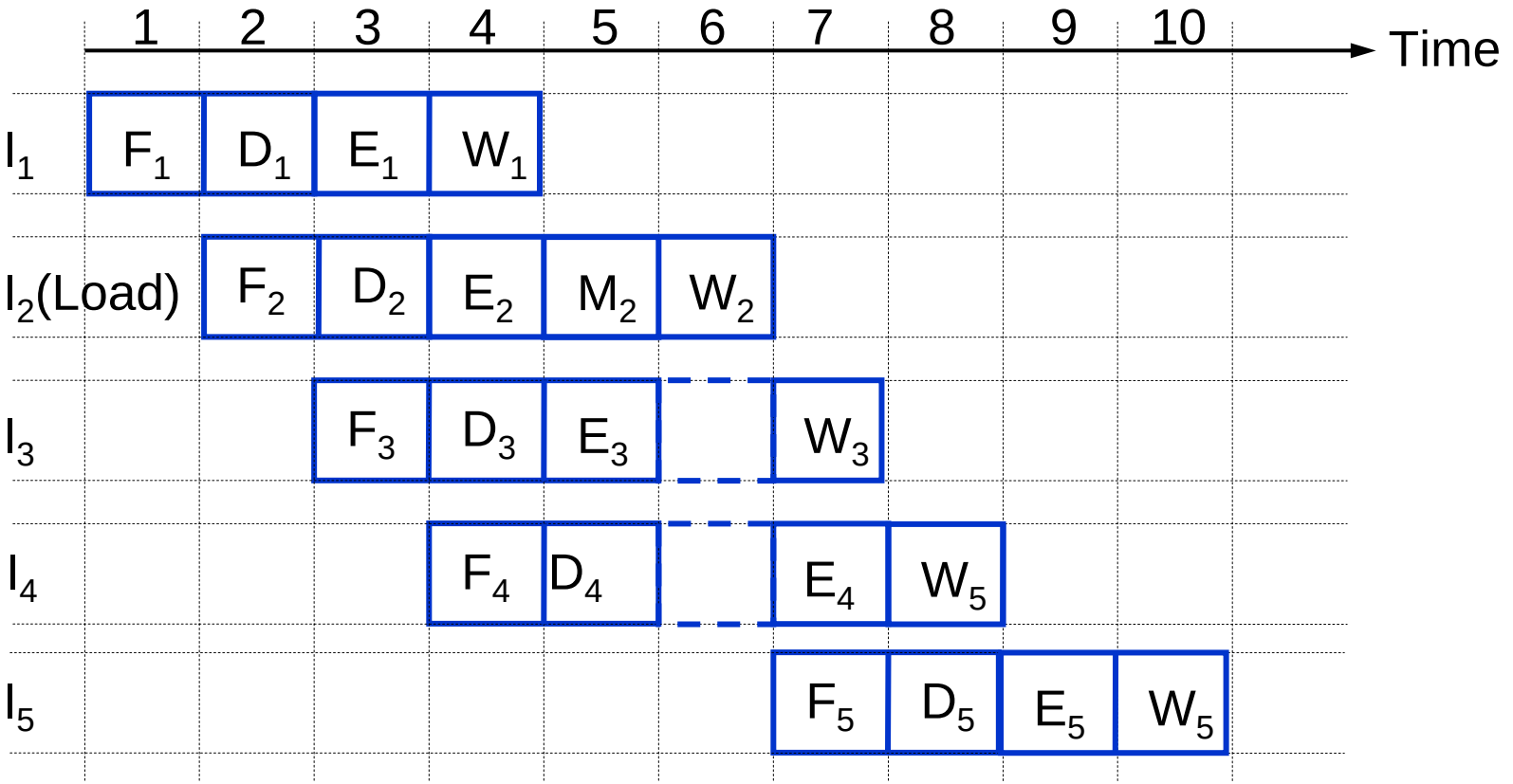


Structural Hazard

- Such idle periods shown in the last slide are called *stalls*. They are also often referred to as *bubbles* in the pipeline. Once created as a result of a delay in one of the pipeline stages, a bubble moves downstream until it reaches the last unit
- In pipelined operation, when two instructions require the use of a given hardware resource at the same time, the pipeline has a structural hazard
- The most common case in which this hazard may arise is in access to memory. One instruction may need to access memory as part of the Execute and Write stage while another instruction is being fetched

An Example of a Structural Hazard

- Load X(R1), R2



Data Hazards

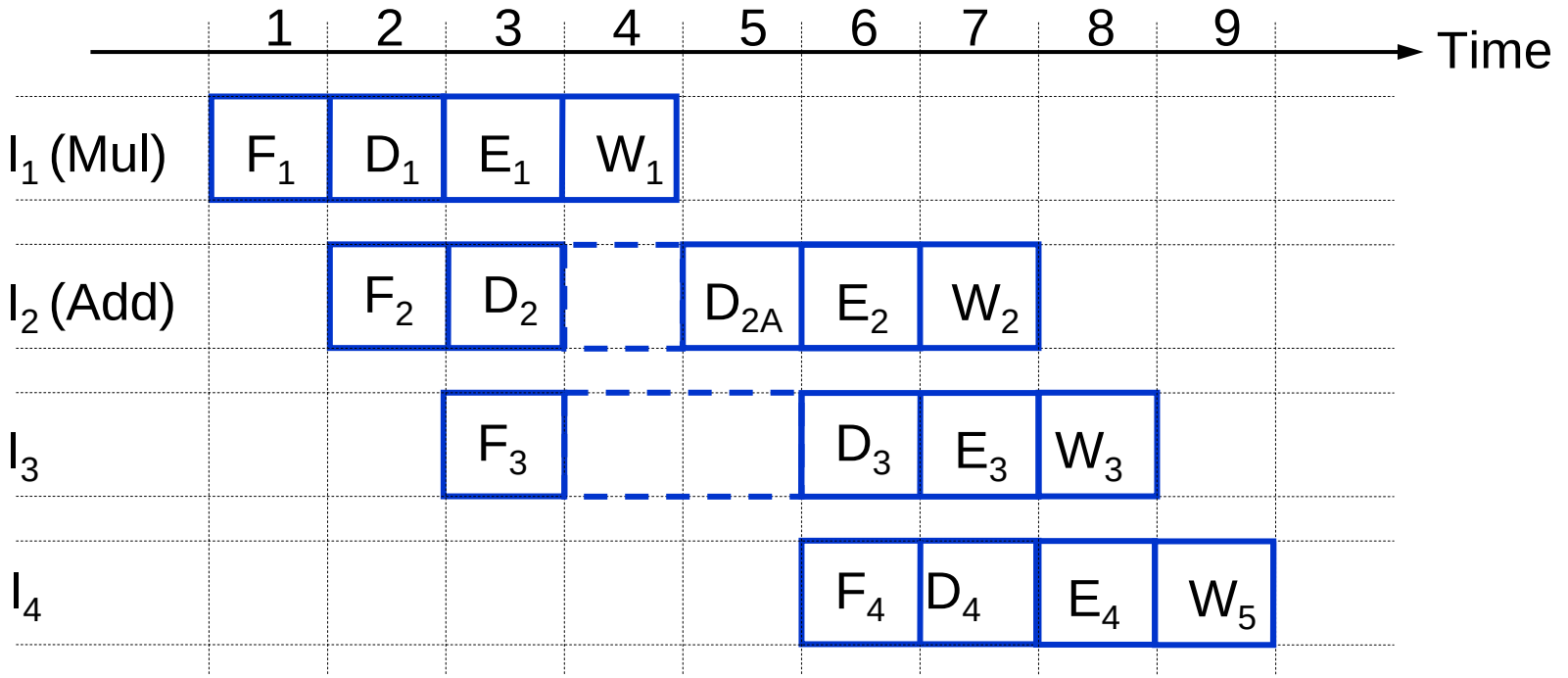
- Consider a program that contains two instructions, I_1 followed by I_2. When this program is executed in a pipeline, the execution of I_2 can begin before the execution of I_1 is completed. This means that the results generated by I_1 may not be available for use by I_2
- Assume that A=5, and consider the following two operations:
  - A←3+A
  - B←4xA
  - When these operations are performed in the order given, the result is B=32. But if they are performed concurrently, the value of A used in computing B would be the original value, 5, leading to an incorrect result

Data Hazards

- On the other hand, the two operations
  - A←5xC
  - B←20+C
  - can be performed concurrently, because these operations are independent
- These two examples illustrate a basic constraint that must be enforced to guarantee correct results.
- When two operations depend on each other, they must be performed sequentially in the correct order

Data Hazards

- For example, the two instructions
  - Mul R2, R3, R4
  - Add R5, R4, R6

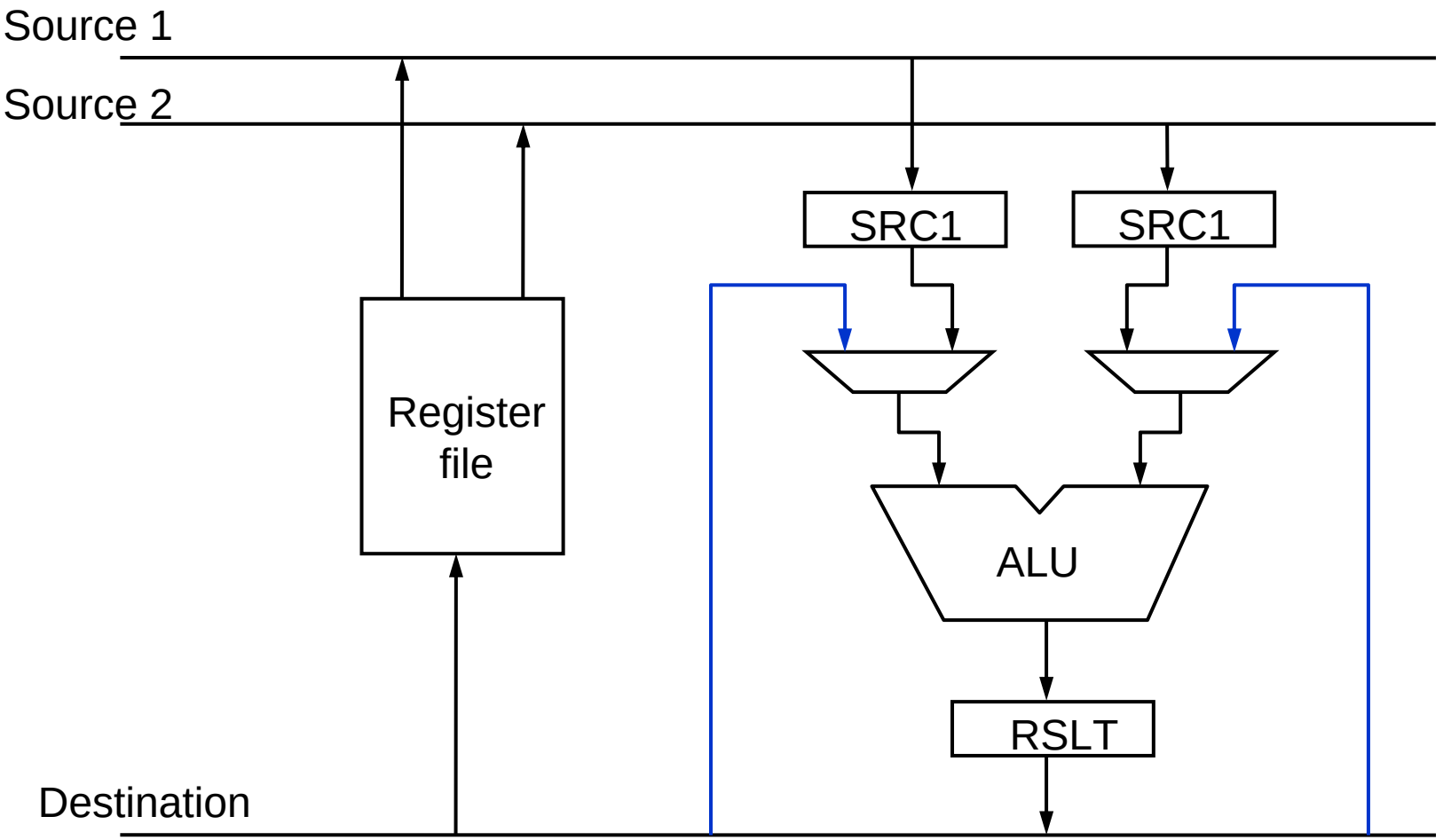


The pipeline schedule

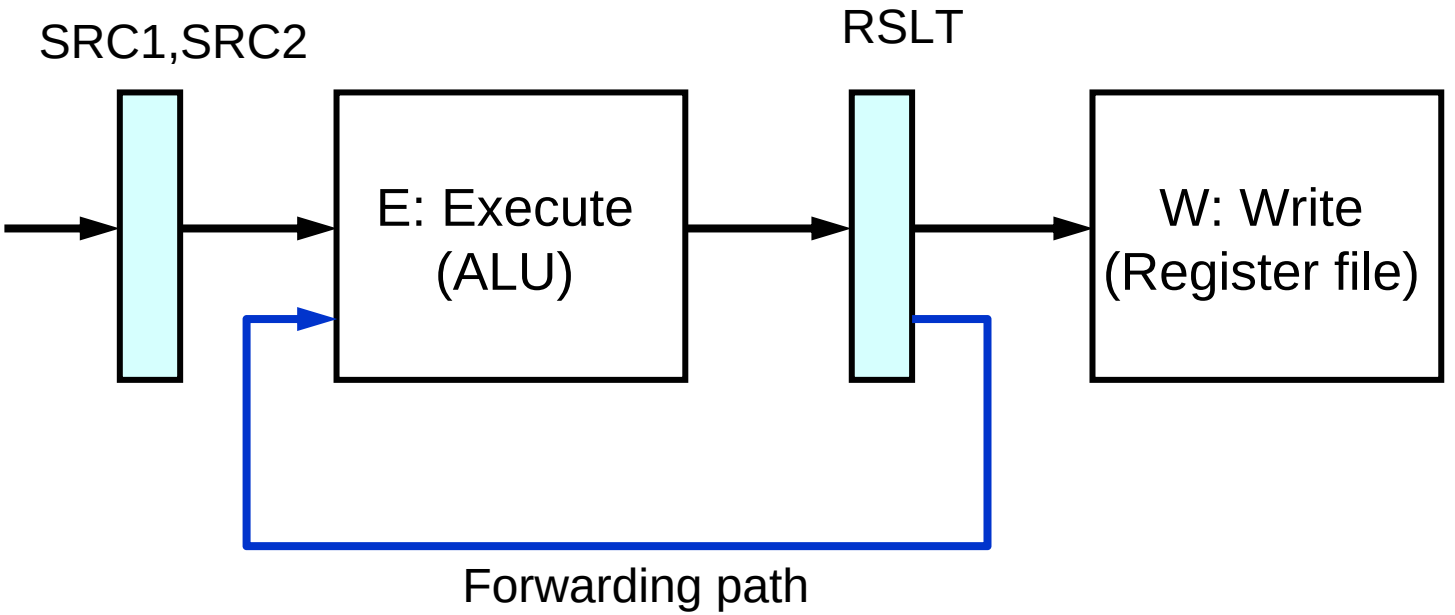
Operand Forwarding in Datapath

- The data hazard just described arises because one instruction, instruction I_2, is waiting for data to be written in the register file. However, these data are available at the output of the output of the ALU once the Execute stage completes step E_1.
- Hence, the delay can be reduced, or possibly eliminated, if we arrange for the result of instruction I_1 to be forwarded directly for use in step E_2

Operand Forwarding in Datapath



Operand Forwarding in a Pipelined Processor

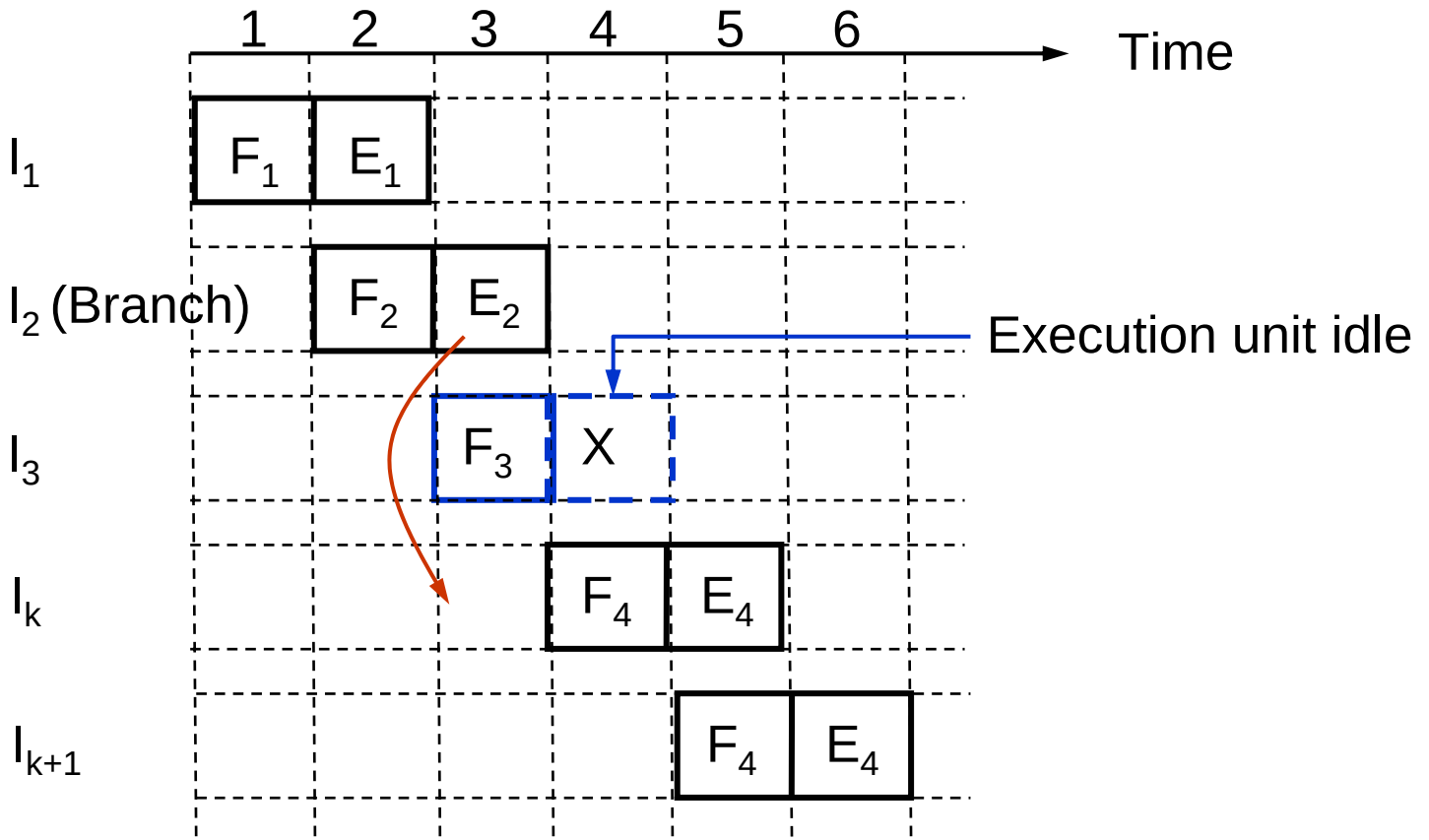


Handling Data Hazards in Software

- An alternative approach is to leave the task of detecting data dependencies and dealing with them to the software.
- In this case, the compiler can introduce two-cycle delay needed between instruction I1 and I2 by inserting NOP (No-operation) instructions, as follows:
  - I_1: Mul R2, R3, R4
  - NOP
  - NOP
  - I_2: Add R5, R4, R6

Instruction Hazards-Unconditional Branch

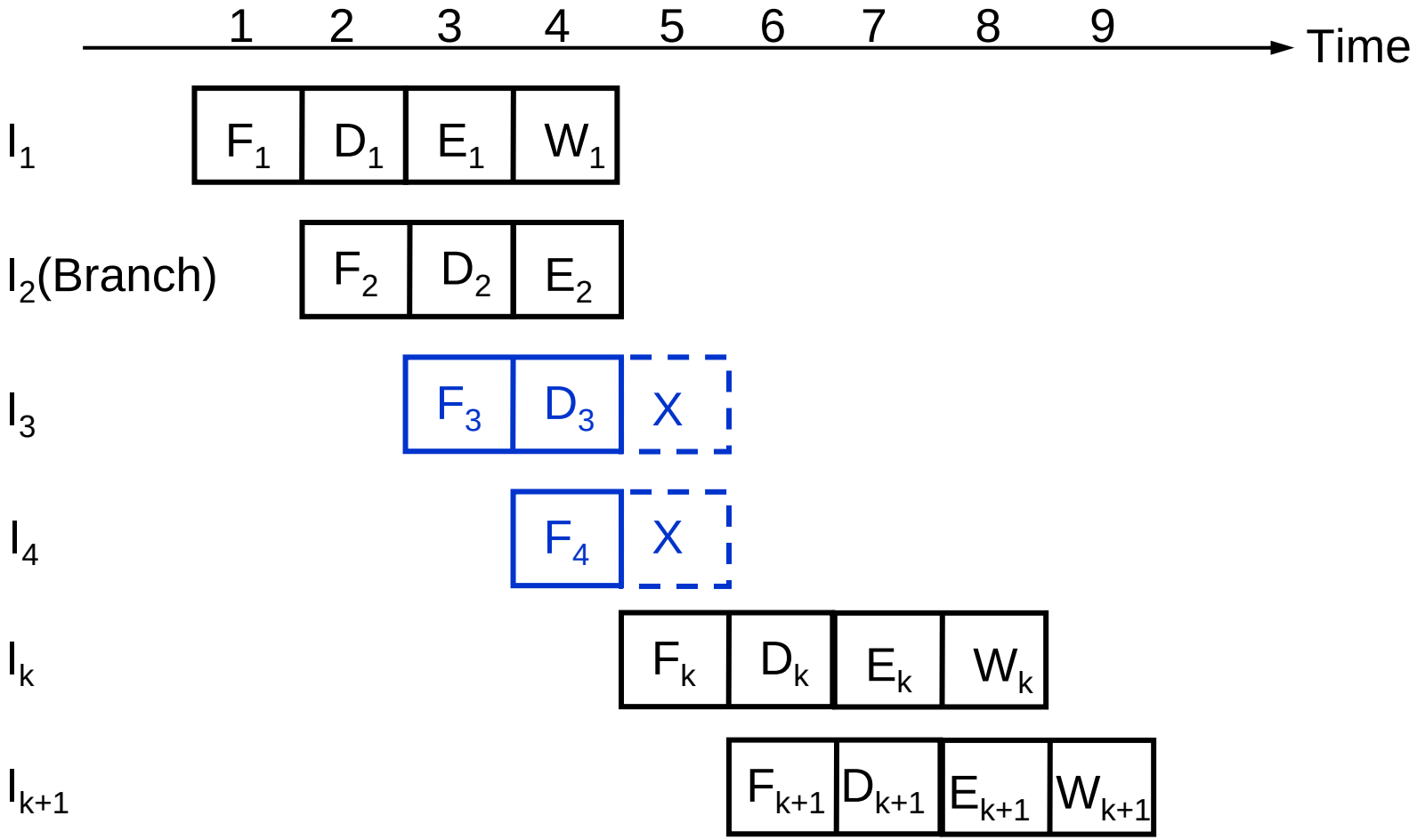
- Unconditional branches



- The time lost as a result of a branch instruction is referred to as the branch penalty

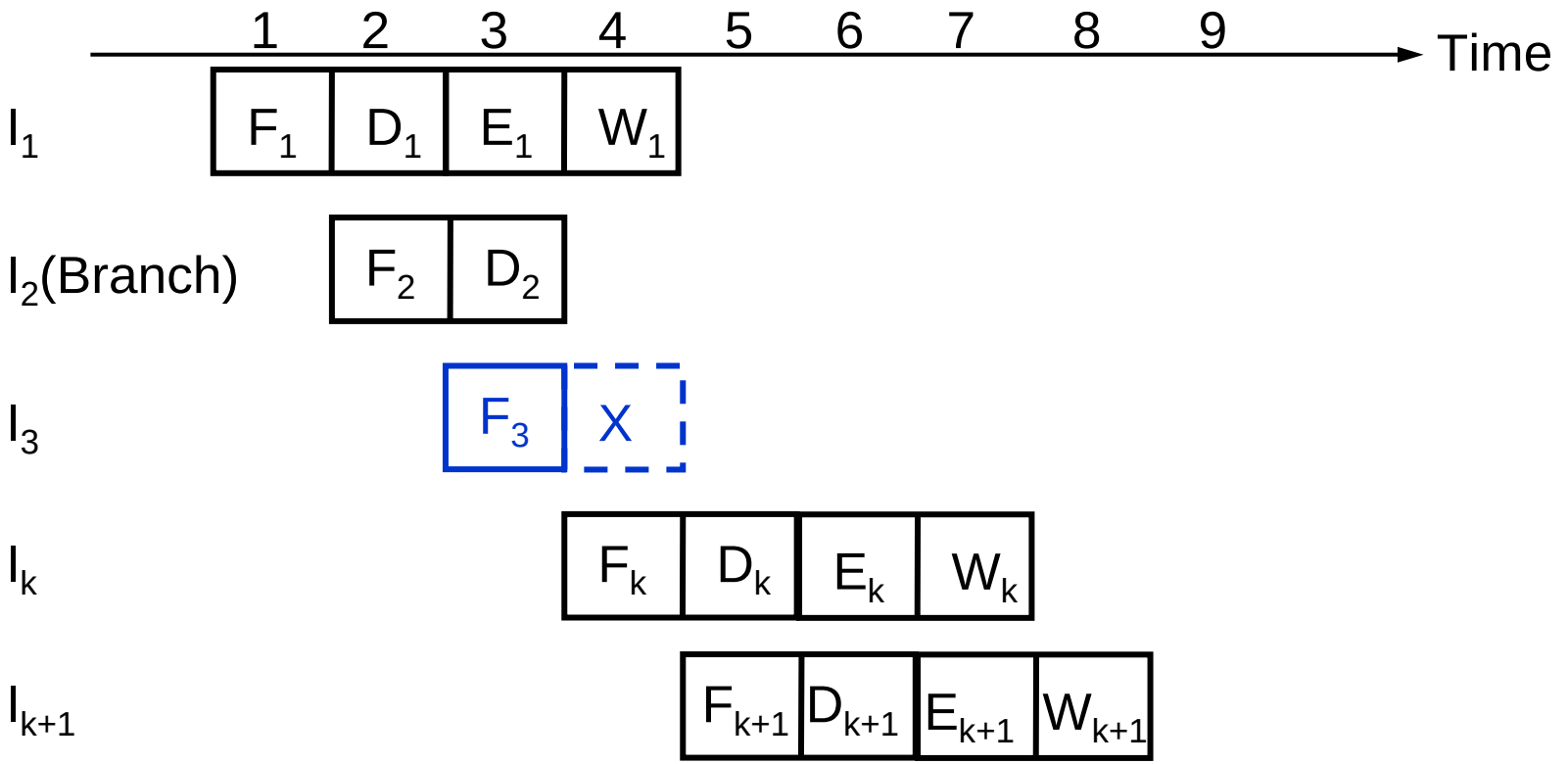
Branch Penalty

- For a longer pipeline, the branch penalty may be higher



Branch Penalty Reduction

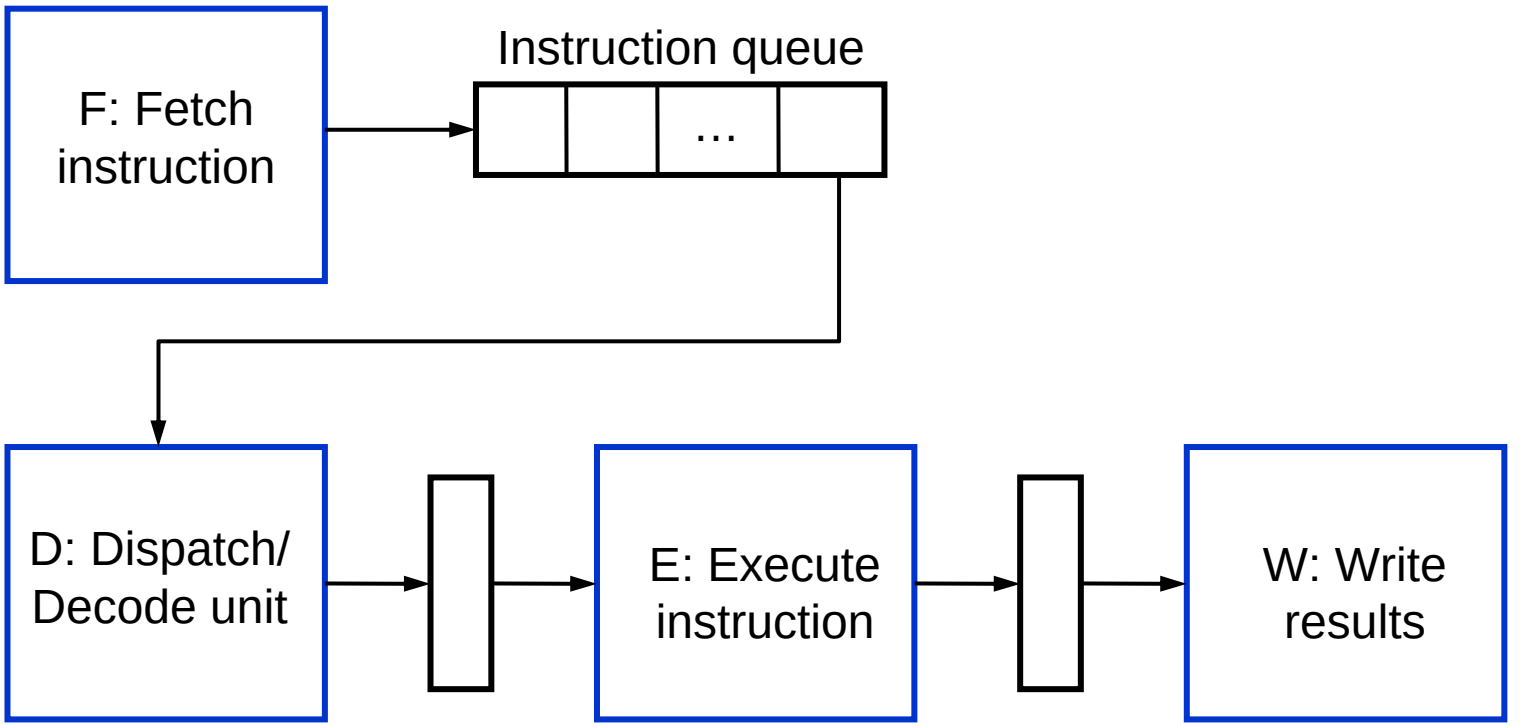
- Reducing the branch penalty requires the branch address to be computed earlier in the pipeline. Typically, the instruction fetch unit has dedicated hardware to identify a branch instruction an compute the branch target address as quickly as possible after an instruction is fetched



Instruction Queue and Prefetching

- Either a cache miss or branch instruction stalls the pipeline for one or more clock cycles. To reduce the effect of these interruptions, many processors employ sophisticated fetch units that can fetch instructions before they are needed and put them in a queue.
- Typically, the instruction queue can store several instructions. A separate unit, which we call the dispatch unit, takes instructions from the front of the queue and sends them to the execution unit.

Instruction Queue and Prefetching



When the pipeline stalls because of a data hazard, for example, the dispatch unit is not able to issue instructions from the instruction queue. However, the fetch unit continues to fetch instructions and add them to the queue. Conversely, if there is a delay in fetching instructions because of a branch or a cache miss, the dispatch unit continues to issue instructions from the instruction queue.

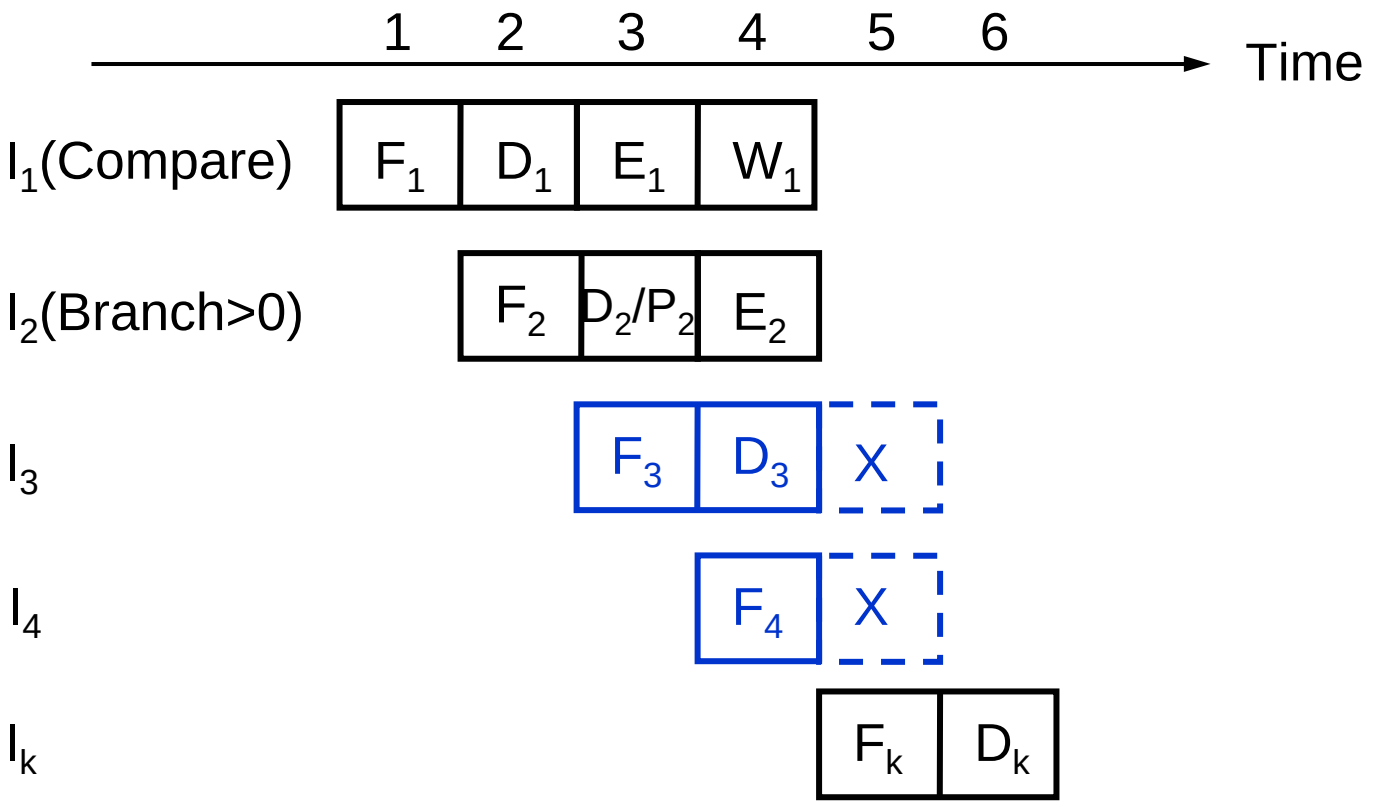
Conditional Branches

- A conditional branch instruction introduces the added hazard caused by the dependency of the branch condition on the result of a preceding instruction. The decision to branch cannot be made until the execution of that instruction has been completed
- Branch instructions occur frequently. In fact, they represent about 20% of the dynamic instruction count of most programs. (The dynamic count is the number of instruction executions, taking into account the fact that some program instructions are executed many times because of loops.)

Branch Prediction

- The simplest form of branch prediction is to assume that the branch will not take place and to continue to fetch instructions in sequential address order. Until the branch condition is evaluated, instruction execution along the predicted path must be done on a speculative basis.
- Speculative execution means that instructions are executed before the processor is certain that they are in the correct execution sequence. Hence, care must be taken that no processor register or memory locations are updated until it is confirmed that these instructions should indeed be executed.

An Example of Branch Prediction



The results of the compare operation are available at the end of cycle 3. Assuming that they are forwarded immediately to the instruction fetch unit, the branch condition is evaluated in cycle 4.

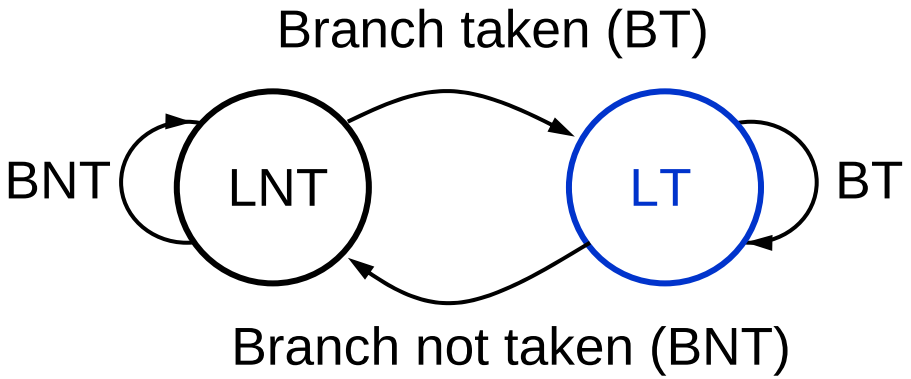
Dynamic Branch Prediction

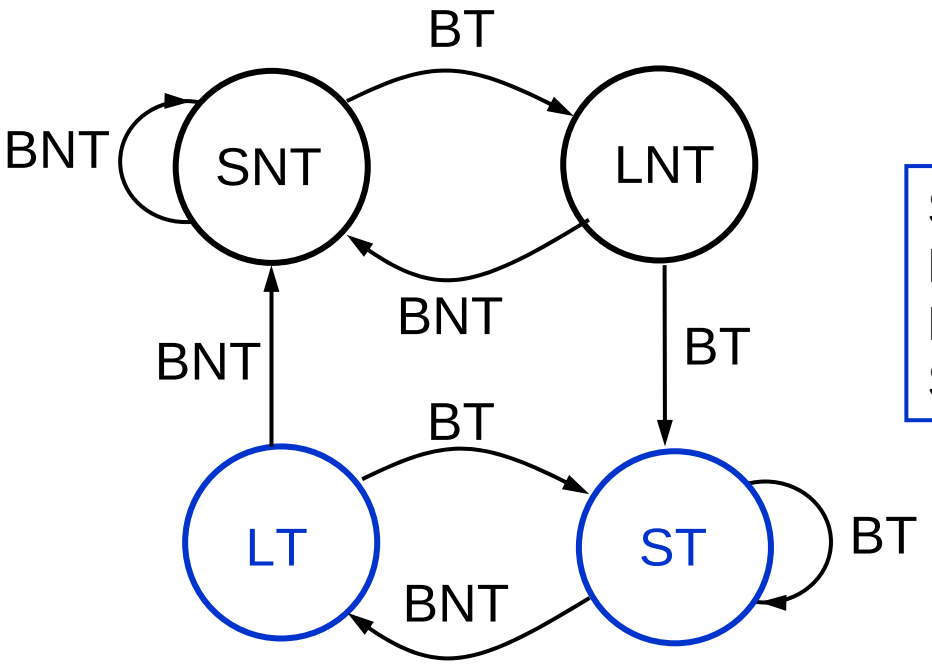
- The branch prediction decision is always the same every time a given instruction is executed, this is called *static branch prediction*
- The prediction decision may changed depending on execution history is called *dynamic branch prediction*
- In dynamic branch prediction schemes, the processor hardware assesses the likelihood of a given branch being taken by keeping track of branch decisions every time that instruction is executed

Dynamic Branch Prediction

- The simplest form, the execution history used in predicting the outcome of a given instruction is the result of the most recent execution of that instruction. The processor assumes that the next time the instruction is executed, the result is likely to be the same.
- Hence, the algorithm may be described by a two-state machine. The two states are
  - LT: Branch is likely to be taken
  - LNT: Branch is likely not to be taken

Dynamic Branch Prediction





ST: Strongly likely to be taken
LT: Likely to be taken
LNT: Likely not to be taken
SNT: Strongly likely not to be taken