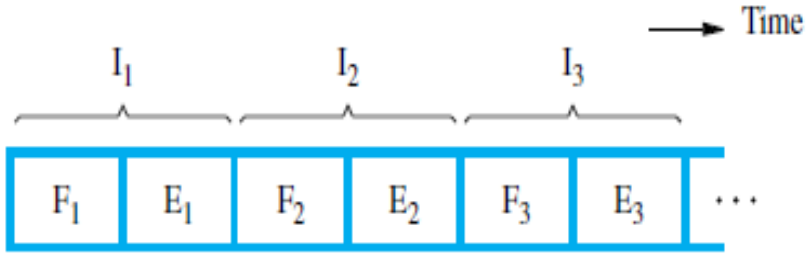
Module 3
Part 2

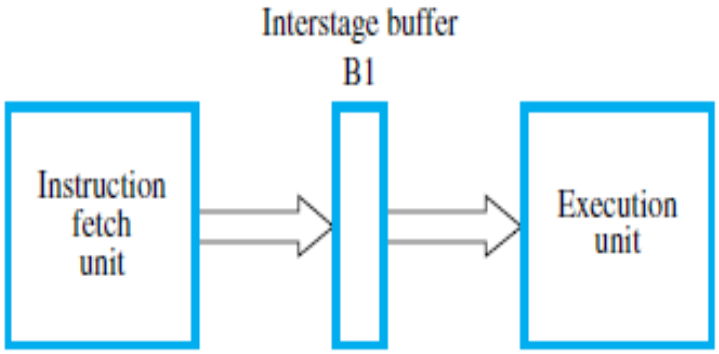
pipelining

Pipelining

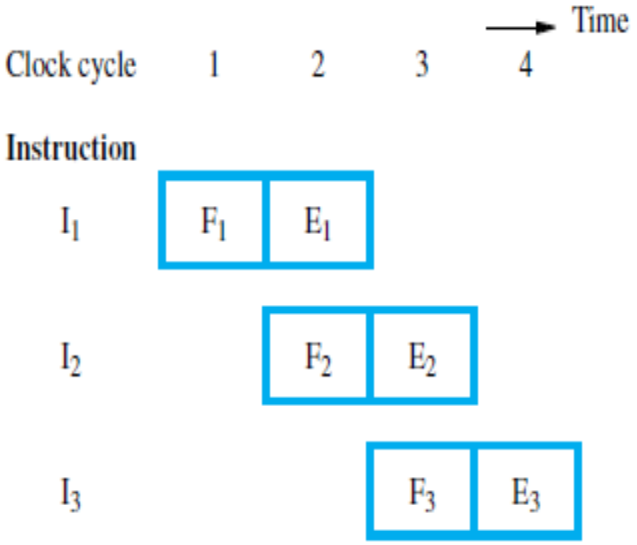
- Pipelining is a technique of decomposing a sequential process into sub operations, with each sub process being executed in a special dedicated segment that operates concurrently with all other segments.
- A pipeline can be visualized as a collection of processing segments through which binary information flows.
- Each segment performs partial processing dictated by the way the task is partitioned.
- The result obtained from the computation in each segment is transferred to the next segment in the pipeline.
- The final result is obtained after the data have passed through all segments.



(a) Sequential execution

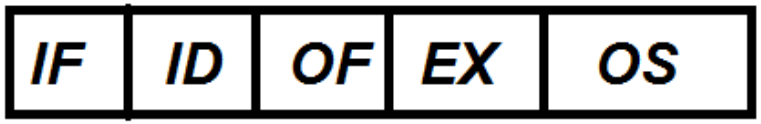


(b) Hardware organization

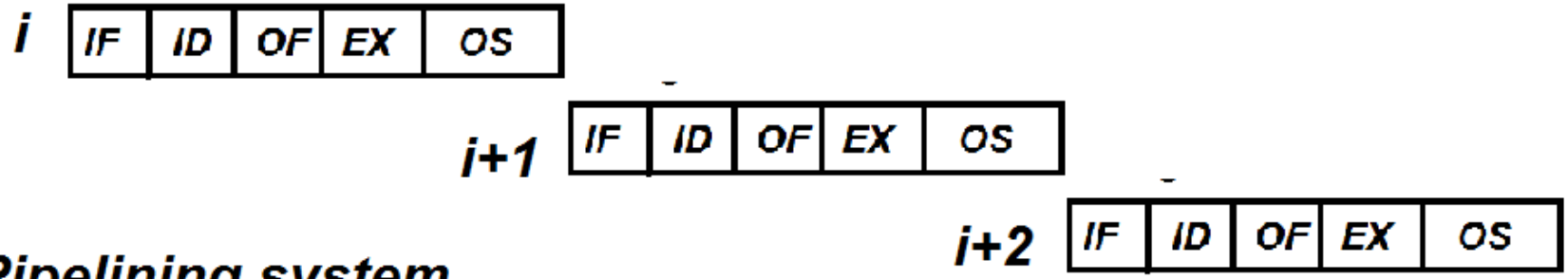


(c) Pipelined execution

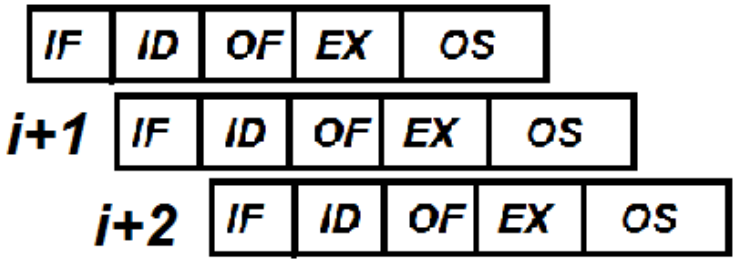
Stages of instruction execution



Non-pipelining system

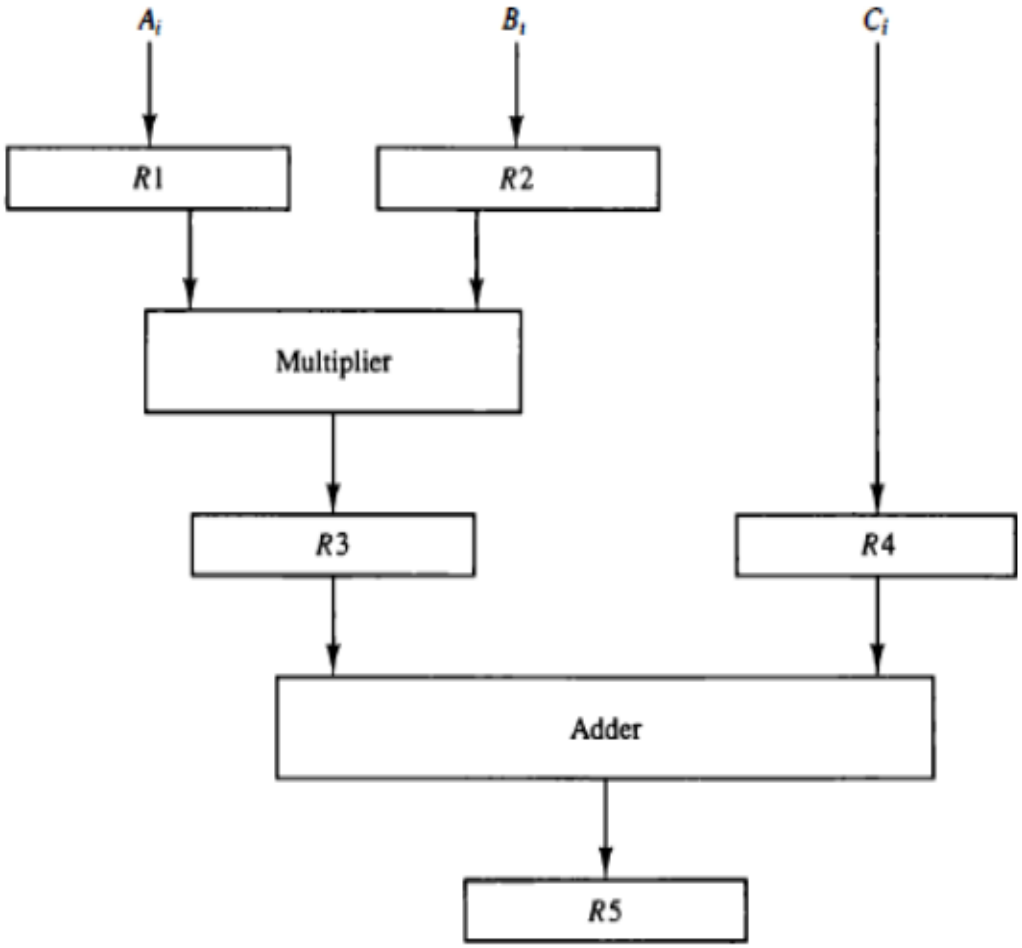


Pipelining system



Pipeline Organization

- Suppose we want to perform the combined multiply and add operation with a stream of numbers.
- Ai*Bi + Ci for i=1, 2, 3 ….7 Each sub operation is to implemented in a segment within a pipeline.
- Each segment has one or two registers and a combinational circuit as shown in fig.



- R1 through R5 are registers that receive new data with every clock pulse. The multiplier and adder are combinational circuits. The sub operations performed in each segment of the pipeline are as follows:
- R1<- Ai R2<-Bi Input Ai and Bi
- R3<-R1*R2 R4<-Ci multiply and input Ci add Ci to product
- R5<-R3+R4
- The five registers are loaded with new data every clock pulse.

Clock Pulse Number	Segment 1		Segment 2		Segment 3
	$R1$	$R2$	$R3$	$R4$	$R5$
1	A_1	B_1	—	—	—
2	A_2	B_2	$A_1 * B_1$	C_1	—
3	A_3	B_3	$A_2 * B_2$	C_2	$A_1 * B_1 + C_1$
4	A_4	B_4	$A_3 * B_3$	C_3	$A_2 * B_2 + C_2$
5	A_5	B_5	$A_4 * B_4$	C_4	$A_3 * B_3 + C_3$
6	A_6	B_6	$A_5 * B_5$	C_5	$A_4 * B_4 + C_4$
7	A_7	B_7	$A_6 * B_6$	C_6	$A_5 * B_5 + C_5$
8	—	—	$A_7 * B_7$	C_7	$A_6 * B_6 + C_6$
9	—	—	—	—	$A_7 * B_7 + C_7$

- The first clock pulse transfers A1 and B1 into R1 and R2. The second clock pulse transfers the product of R1 and R2 into R3 and C1 into R4. The same clockpulse transfers A2 and B2 into R1 and R2. The third clock pulse operates on all three segments simultaneously. It places A3 and B3 into R1 and R2, transfers the product of R1 and R2 into R3, transfers C2 into R4, and places the sum of R3 and R4 into R5. It takes three clock pulses to fill up the pipe and retrieve the first output from R5. From there on, each clock produces a new output and moves the data one step down the pipeline. This happens as long as new input data flow into the system.

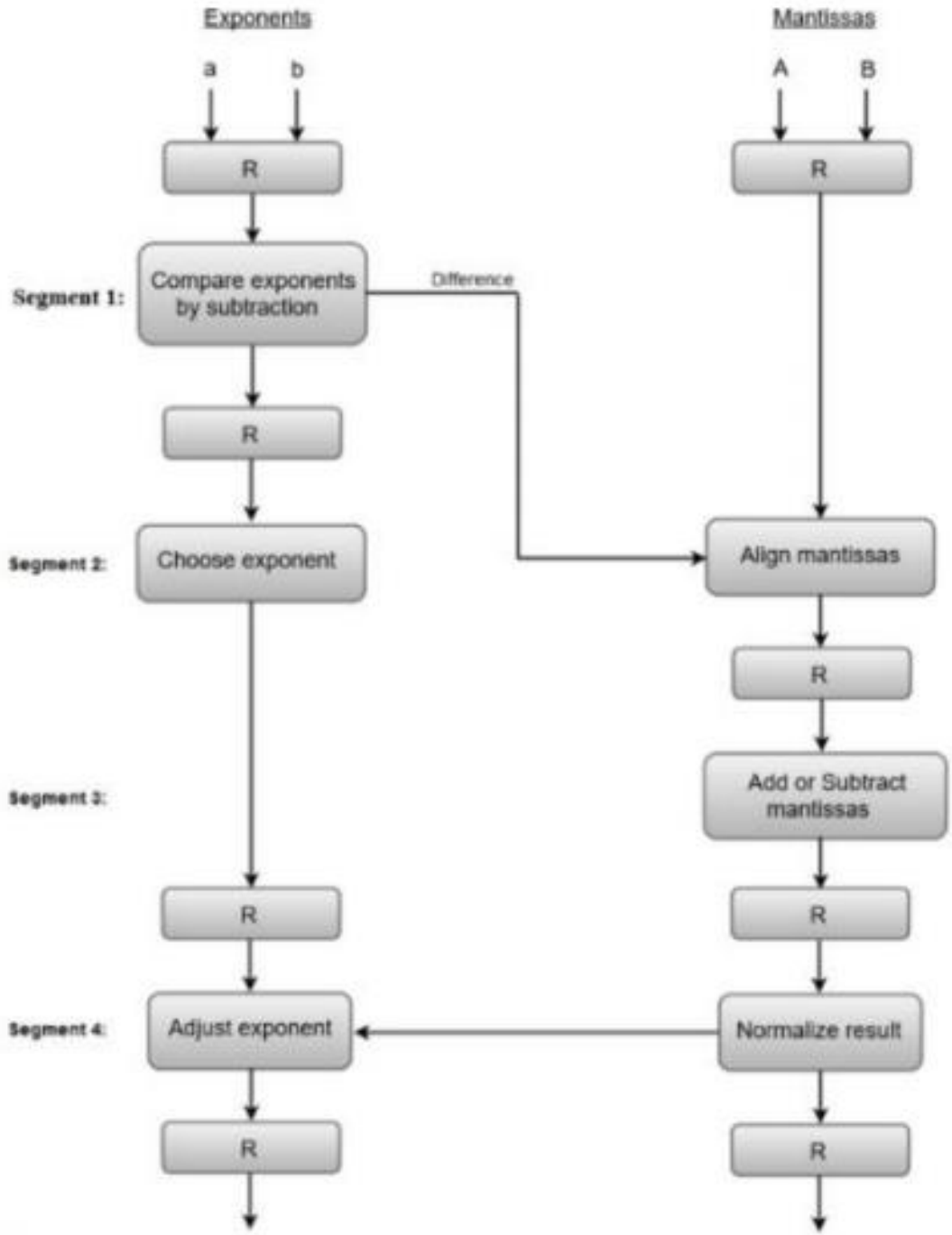
Classification of pipeline processor

1. Arithmetic pipeline
2. Instruction pipeline

- **Arithmetic pipeline**
- An *arithmetic pipeline* generally breaks an arithmetic operation into multiple arithmetic steps.
- So in arithmetic pipeline, an arithmetic operation like multiplication, addition, etc. can be divided into series of steps that can be executed one by one in stages in Arithmetic Logic Unit (ALU).

- **Floating point addition using arithmetic pipeline :**
  The following sub operations are performed in this case:
- Compare the exponents.
- Align the mantissas.
- Add or subtract the mantissas.
- Normalize the result

- The inputs to the floating point adder pipeline are two normalized floating point binary numbers.
- X=A*2^a Y=B*2^b


Exponents
Mantissas
a
b
A
B
R
R
Segment 1:
Compare exponents by subtraction
Difference
R
Segment 2:
Choose exponent
Align mantissas
R
Segment 3:
Add or Subtract mantissas
R
R
Segment 4:
Adjust exponent
Normalize result
R
R

$X = m \times r^{e}$

m- mantissa

r- radix

e- exponent

$X = 0.9143 \times 10^{3}$

$Y = 0.5429 \times 10^{2}$

step1: compare exponent

x=3, y=2 , larger is chosen =>3

step 2:allign mantissa

$X = 0.9143 \times 10^{3}$

$Y = 0.05429 \times 10^{3}$

step3:Add mantissas

$X+Y = 0.968$

step4:Normalise result

0.968×10^{3}

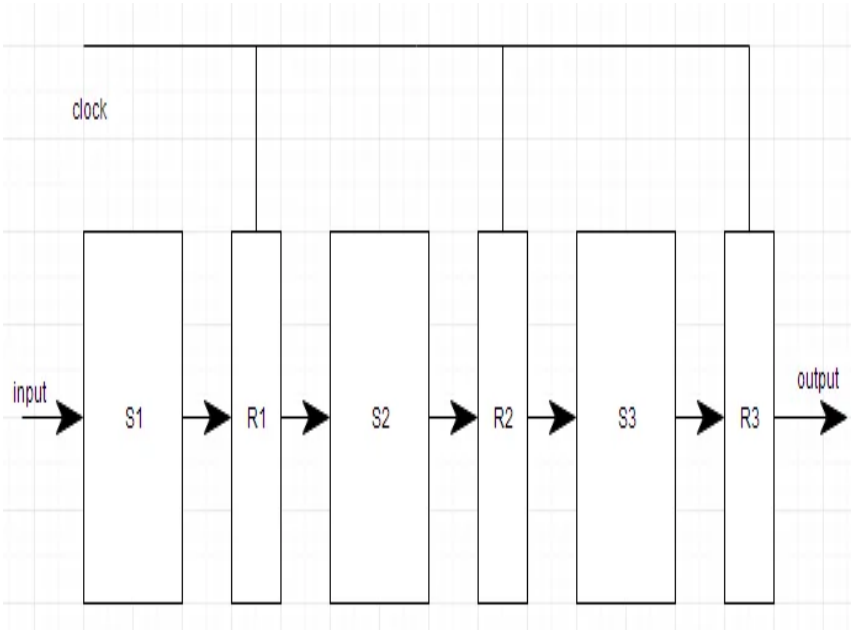
- First of all the two exponents are compared and the larger of two exponents is chosen as the result exponent. The difference in the exponents then decides how many times we must shift the smaller exponent to the right. Then after shifting of exponent, both the mantissas get aligned. Finally the addition of both numbers take place followed by normalisation of the result in the last segment.

Instruction Pipeline

- A stream of instructions can be executed by overlapping fetch, decode and execute phases of an instruction cycle. This type of technique is used to increase the throughput of the computer system.
- An instruction pipeline reads instruction from the memory while previous instructions are being executed in other segments of the pipeline.
- Thus we can execute multiple instructions simultaneously. The pipeline will be more efficient if the instruction cycle is divided into segments of equal duration.

- In the most general case computer needs to process each instruction in following sequence of steps:
- Fetch the instruction from memory (FI)
- Decode the instruction (DA)
- Calculate the effective address
- Fetch the operands from memory (FO)
- Execute the instruction (EX)
- Store the result in the proper place

- In pipeline system, each segment consists of an input register followed by a combinational circuit. The register is used to hold data and combinational circuit performs operations on it. The output of combinational circuit is applied to the input register of the next segment.



- Each step is executed in a particular segment, and there are times when different segments may take different times to operate on the incoming information. Moreover, there are times when two or more segments may require memory access at the same time, causing one segment to wait until another is finished with the memory.
- The organization of an instruction pipeline will be more efficient if the instruction cycle is divided into segments of equal duration. One of the most common examples of this type of organization is a **Four-segment instruction pipeline.**
-

- A **four-segment instruction** pipeline combines two or more different segments and makes it as a single one. For instance, the decoding of the instruction can be combined with the calculation of the effective address into one segment.

Segment 1:

- The instruction fetch segment can be implemented using first in, first out (FIFO) buffer.

Segment 2:

- The instruction fetched from memory is decoded in the second segment, and eventually, the effective address is calculated in a separate arithmetic circuit.

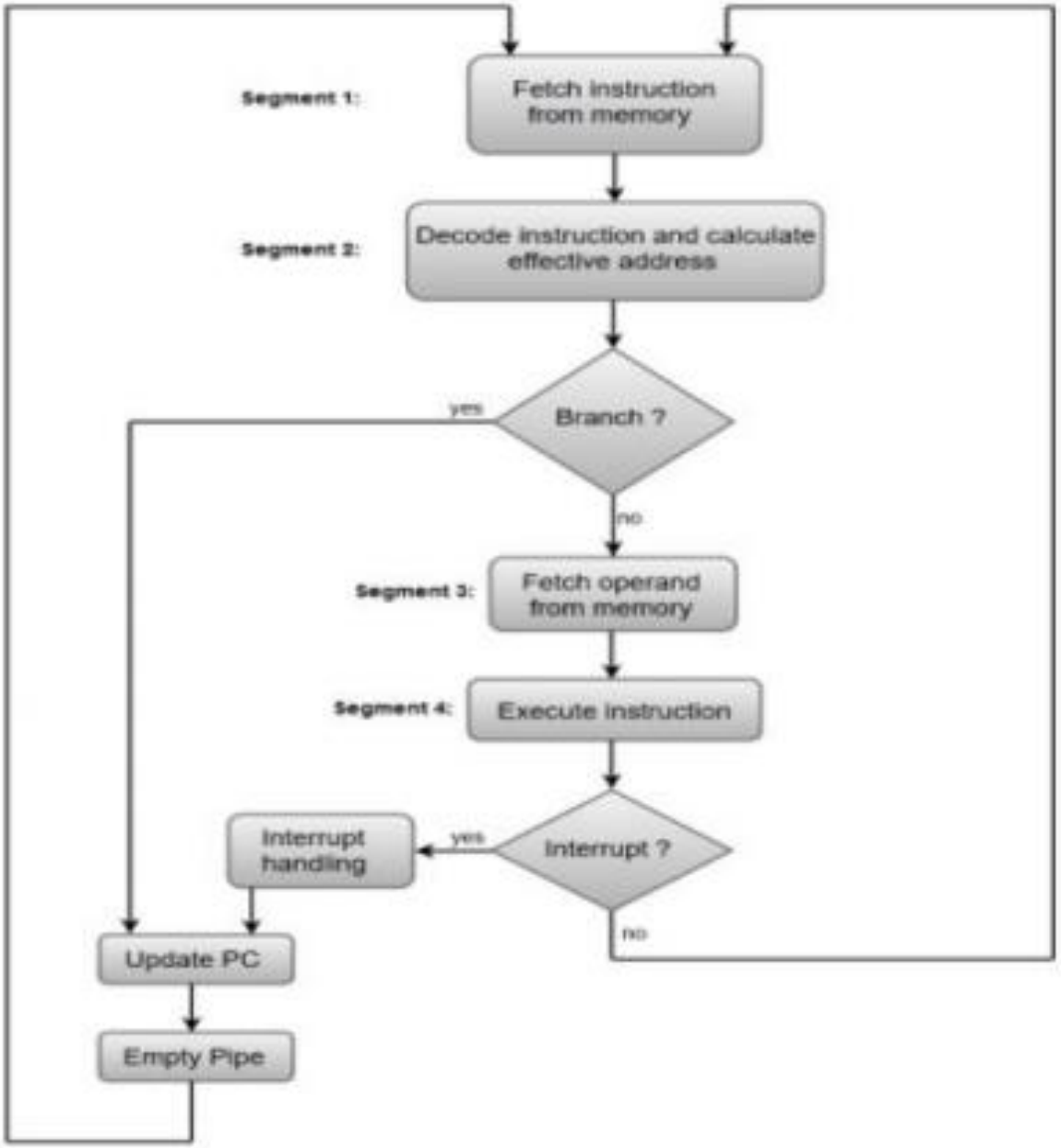
Segment 3:

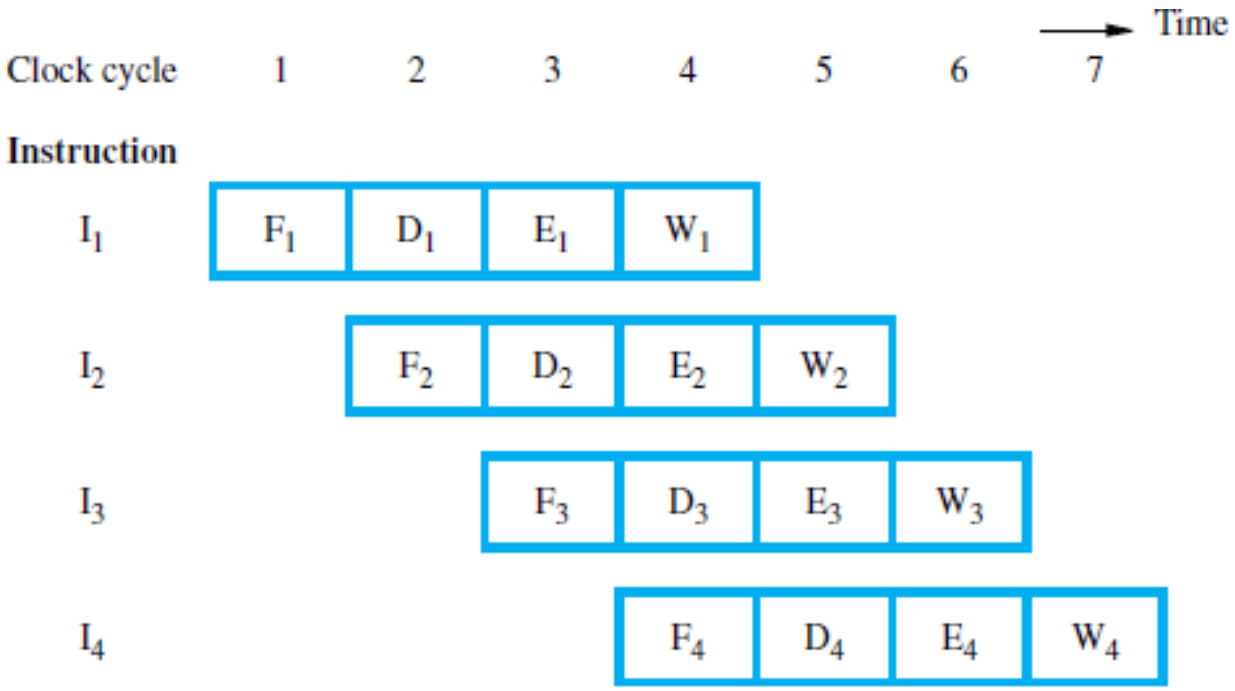
- An operand from memory is fetched in the third segment.

Segment 4:

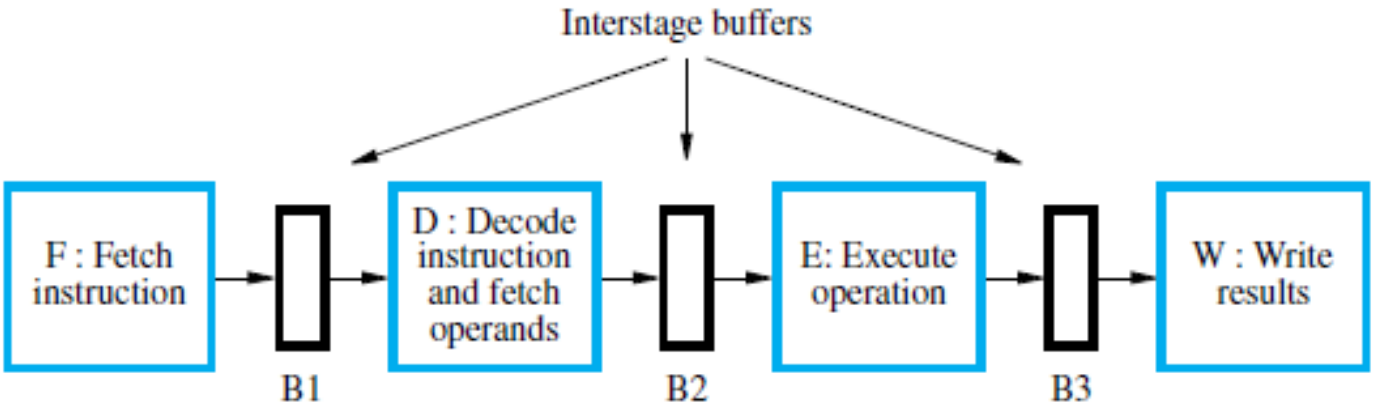
- The instructions are finally executed in the last segment of the pipeline organization.

Four Segment Instruction Pipeline





(a) Instruction execution divided into four steps



(b) Hardware organization

The operation of the instruction pipeline.
The time in the horizontal axis is divided into steps of equal duration.

Step:		1	2	3	4	5	6	7	8	9	10	11	12	13
Instruction:	1	FI	DA	FO	EX									
	2		FI	DA	FO	EX								
(Branch)	3			FI	DA	FO	EX							
	4				FI	-	-	FI	DA	FO	EX			
	5					-	-	-	FI	DA	FO	EX		
	6									FI	DA	FO	EX	
	7										FI	DA	FO	EX

- It is assumed that the processor has separate instruction and data memories so that the operation in FI and FO can proceed at the same time.
- In the absence of a branch instruction, each segment operates on different instructions. Thus, in step 4, instruction 1 is being executed in segment EX; the operand for instruction 2 is being fetched into segment FO; instruction 3 is being decoded in segment DA; and instruction 4 is being fetched from memory in segment FI.
- Assume now this instruction 3 is a branch instruction. As soon as this instruction is decoded in segment DA in step 4, the transfer from FI to DA of the other instructions are halted until the branch instruction is executed in step 6.

PIPELINE CONFLICTS

- Resource conflicts caused by access to memory by two segments at the same time. Most of these conflicts can be resolved by using separate instruction and data memories.
- Data dependency conflicts arise when an instruction depends on the result of a previous instruction, but this result is not yet available.
- Branch difficulties arise from branch and other instructions that change the value of PC.

Pipeline Hazards

- ***Pipeline Hazards*** are situations that prevent the next instruction in the instruction stream from executing in its designated clock cycle
- Hazards reduce the performance from the ideal speedup gained by pipelining
- Three types of hazards
  - ***Structural hazards***
    - Arise from resource conflicts when the hardware can't support all possible combinations of overlapping instructions
  - ***Data hazards***
    - Arise when an instruction depends on the results of a previous instruction in a way that is exposed by overlapping of instruction in pipeline
  - ***Control hazards***
    - Arise from the pipelining of branches and other instructions that change the PC (Program Counter)

Pipeline Hazards

- Hazards in pipeline can make the pipeline to ***stall***
- Eliminating a hazard often requires that some instructions in the pipeline to be allowed to proceed while others are delayed
  - When an instruction is stalled, instructions issued *latter* than the stalled instruction are stopped, while the ones issued *earlier* must continue
- No new instructions are fetched during the stall

DATA HAZARDS

- We must ensure that the results obtained when instructions are executed in a pipelined processor are identical to those obtained when the same instructions are executed sequentially.

Hazard occurs

- $A \leftarrow 3 + A$
- $B \leftarrow 4 \times A$

No hazard

- $A \leftarrow 5 \times C$
- $B \leftarrow 20 + C$
- When two operations depend on each other, they must be executed sequentially in the correct order.

EG2:

- Mul R2, R3, R4
- Add R5, R4, R6

Mul R2,R3,R4
Add R5,R4,R6 give rise to a data dependency

- The result of the multiply instruction is placed into register R4, which in turn is one of the two source operands of the Add instruction.
- Assuming that the multiply operation takes one clock cycle to complete, execution would proceed as shown in Figure. As the Decode unit decodes the Add instruction in cycle 3, it realizes that R4 is used as a source operand. Hence, the D step of that instruction cannot be completed until the W step of the multiply instruction has been completed. Completion of step D2 must be delayed to clock cycle 5, and is shown as step D2A in the figure. Instruction I3 is fetched in cycle 3, but its decoding must be delayed because step D3 cannot precede D2. Hence, pipelined execution is stalled for two cycles.

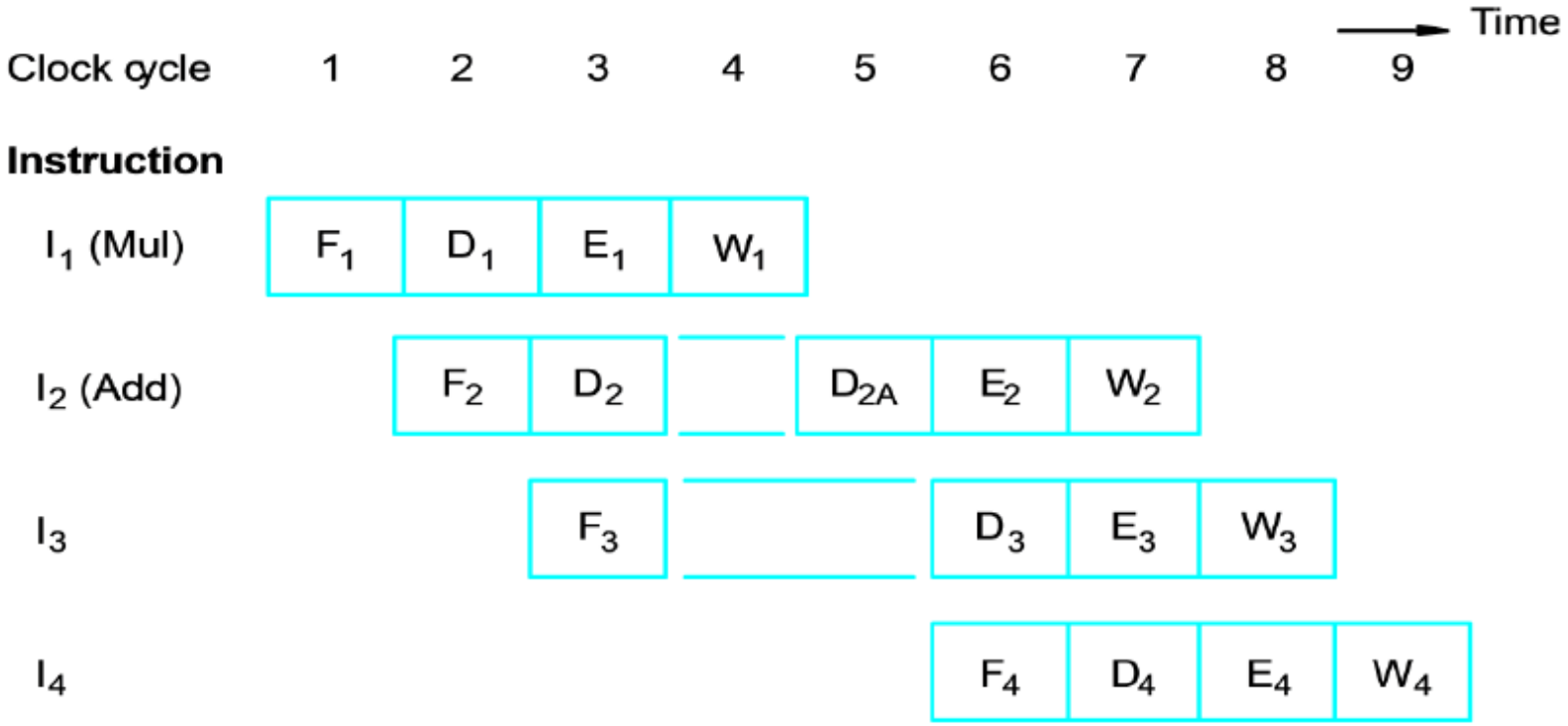
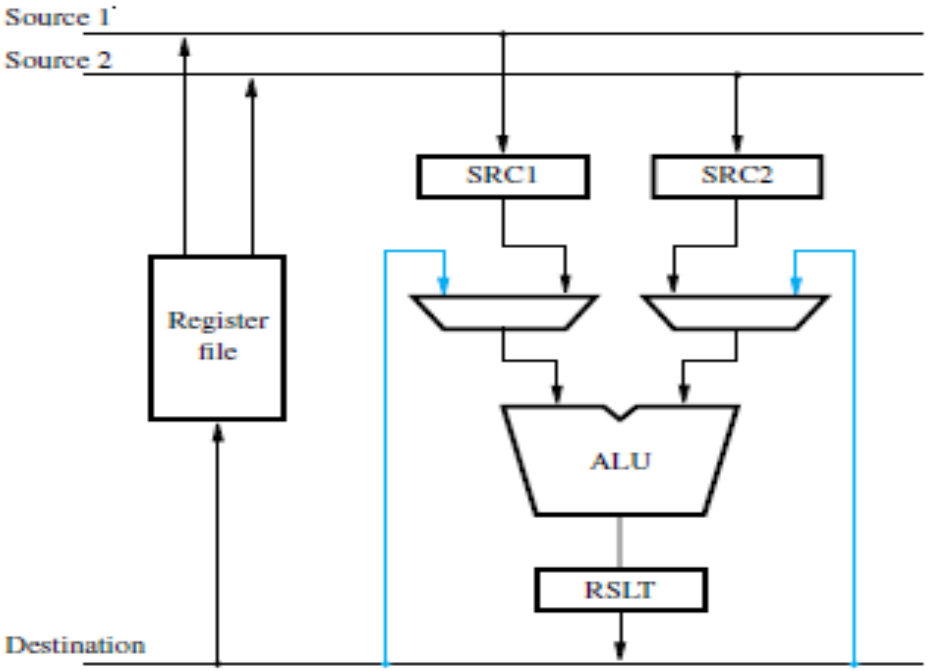


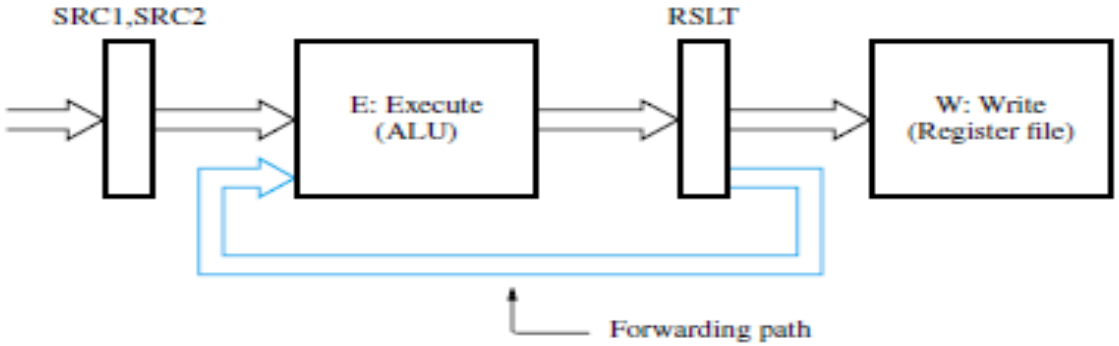
Figure 8.6. Pipeline stalled by data dependency between D_2 and W_1.

OPERAND FORWARDING

- The data hazard just described arises because one instruction, instruction I2 in above figure, is waiting for data to be written in the register file.
- However, these data are available at the output of the ALU once the Execute stage completes step E1. Hence, the delay can be reduced, or possibly eliminated, if we arrange for the result of instruction I1 to be forwarded directly for use in step E2.



(a) Datapath



(b) Position of the source and result registers in the processor pipeline

Operand forwarding in a pipelined processor.

- In this arrangement registers SRC1, SRC2, and RSLT have been added. These registers constitute the inter stage buffers needed for pipelined operation, as illustrated in Figure *b.*
- *With* reference to Figure 8.2*b, registers SRC1 and SRC2 are part of buffer B2 and RSLT is* part of B3.
- The data forwarding mechanism is provided by the blue connection lines.
- The two multiplexers connected at the inputs to the ALU allow the data on the destination bus to be selected instead of the contents of either the SRC1 or SRC2 register.

The operations performed in each clock cycle are as follows:

- After decoding instruction I2 and detecting the data dependency, a decision is made to use data forwarding.
- The operand not involved in the dependency, register R2, is read and loaded in register SRC1 in clock cycle 3.
- In the next clock cycle, the product produced by instruction I1 is available in register RSLT, and because of the forwarding connection, it can be used in step E2.
- Hence, execution of I2 proceeds without interruption.

HANDLING DATA HAZARDS IN SOFTWARE

- In Fig, we assumed the data dependency is discovered by the hardware while the instruction is being decoded.
- The control hardware delays reading register R4 until cycle 5, thus introducing a 2-cycle stall unless operand forwarding is used.
- 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 the two-cycle delay needed between instructions I1 and I2 by inserting NOP (No-operation) instructions, as follows:

I1: Mul R2,*R3,R4*
NOP
NOP
I2: Add R5,*R4,R6*

- If the responsibility for detecting such dependencies is left entirely to the software, the compiler must insert the NOP instructions to obtain a correct result. This possibility illustrates the close link between the compiler and the hardware.
- A particular feature can be either implemented in hardware or left to the compiler.
- Leaving tasks such as inserting NOP instructions to the compiler leads to simpler hardware.
- Being aware of the need for a delay, the compiler can attempt to reorder instructions to perform useful tasks in the NOP slots, and thus achieve better performance.
- On the other hand, the insertion of NOP instructions leads to larger code size. Also, it is often the case that a given processor architecture has several hardware implementations, offering different features.
- NOP instructions inserted to satisfy the requirements of one implementation may not be needed and, hence, would lead to reduced performance on a different implementation.

SIDE EFFECTS

- The data dependencies encountered in the preceding examples are explicit and easily detected because the register involved is named as the destination in instruction I1 and as a source in I2.

- Sometimes an instruction changes the contents of a register other than the one named as the destination. An instruction that uses an autoincrement or autodecrement addressing mode is an example.

- In addition to storing new data in its destination location, the instruction changes the contents of a source register used to access one of its operands. All the precautions needed to handle data dependencies involving the destination location must also be applied to the registers affected by an autoincrement or autodecrement operation.
- When a location other than one explicitly named in an instruction as a destination operand is affected, the instruction is said to have a ***side effect.***

- For example, stack instructions, such as push and pop, produce similar side effects because they implicitly use the autoincrement and autodecrement addressing modes.
- Another possible side effect involves the condition code flags, which are used by instructions such as conditional branches and add-with-carry.

- Suppose that registers R1 and R2 hold a double-precision integer number that we wish to add to another double precision number in registers R3 and R4. This may be accomplished as follows:

  Add R1,*R3*

  AddWithCarry R2,*R4*

- An implicit dependency exists between these two instructions through the carry flag.
- This flag is set by the first instruction and used in the second instruction, which performs the operation

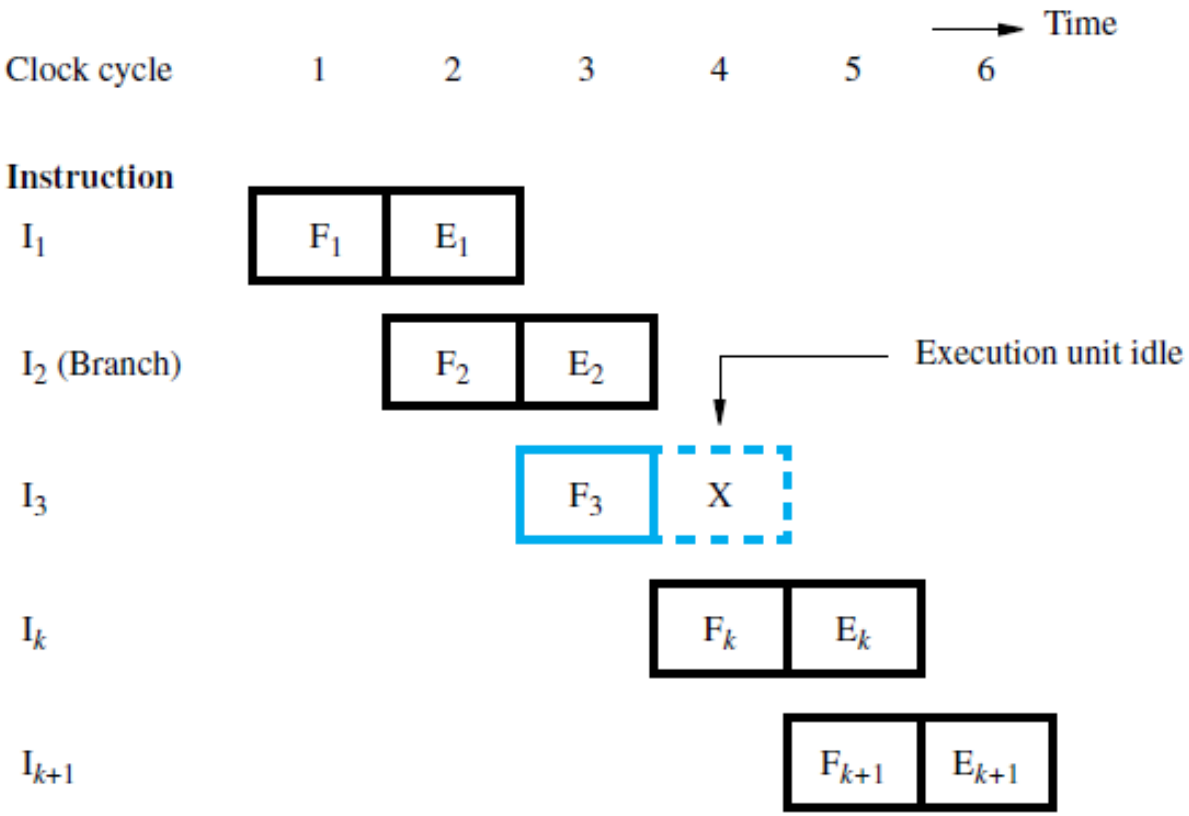
  R4 ← [R2] + [R4] + carry

- Instructions that have side effects give rise to multiple data dependencies, which lead to a substantial increase in the complexity of the hardware or software needed to resolve them.
- For this reason, instructions designed for execution on pipelined hardware should have few side effects.

INSTRUCTION HAZARDS

- The purpose of the instruction fetch unit is to supply the execution units with a steady stream of instructions.
- Whenever this stream is interrupted, the pipeline stalls, as the case of a cache miss and branch instruction.

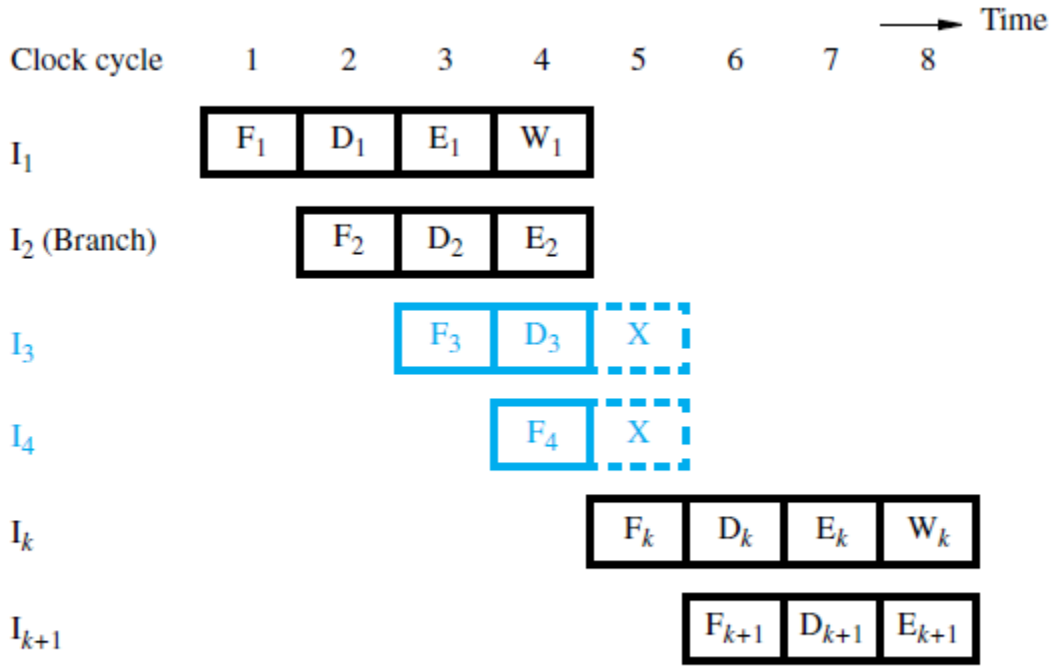
UNCONDITIONAL BRANCHING



An idle cycle caused by a branch instruction.

- Figure shows a sequence of instructions being executed in a two-stage pipeline. Instructions I1 to I3 are stored at successive memory addresses, and I2 is a branch instruction. Let the branch target be instruction I*k* .
- *In clock cycle 3, the fetch operation* for instruction I3 is in progress at the same time that the branch instruction is being decoded and the target address computed.
- In clock cycle 4, the processor must discard I3, which has been incorrectly fetched, and fetch instruction I*k* .
- *In the meantime, the* hardware unit responsible for the Execute (E) step must be told to do nothing during that clock period. Thus, the pipeline is stalled for one clock cycle.

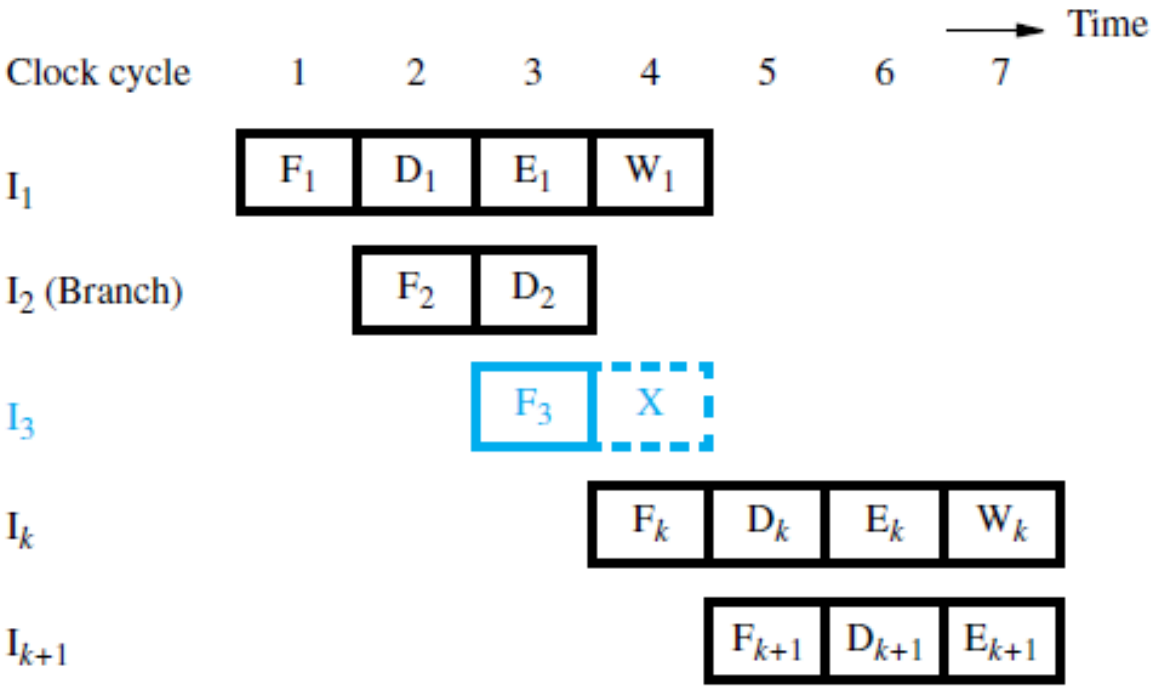
The time lost as a result of a branch instruction is often referred to as the *branch penalty*. *In above figure , the branch penalty is one clock cycle. For a longer pipeline,* the branch penalty may be higher.



(a) Branch address computed in Execute stage

We have assumed that the branch address is computed in step E2. Instructions I3 and I4 must be discarded, and the target instruction, I*k* , *is fetched in clock cycle 5. Thus, the branch penalty is two clock* cycles.

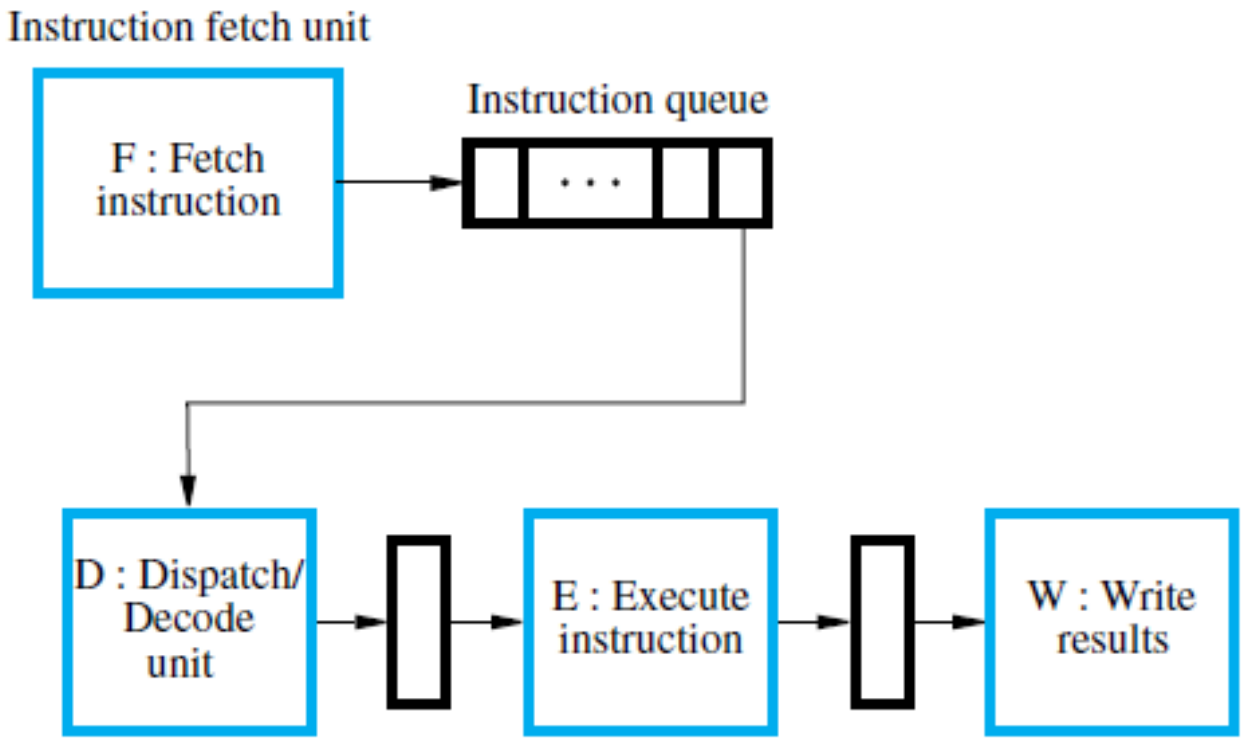
- Reducing the branch penalty requires the branch address to be computed earlier in the pipeline. Typically, the instruction fetch unit has dedicated hardware to identify a branch instruction and compute the branch target address as quickly as possible after an instruction is fetched.
- With this additional hardware, both of these tasks can be performed in step D2, leading to the sequence of events shown in Figure (*b). In this* case, the branch penalty is only one clock cycle.



(b) Branch address computed in Decode stage

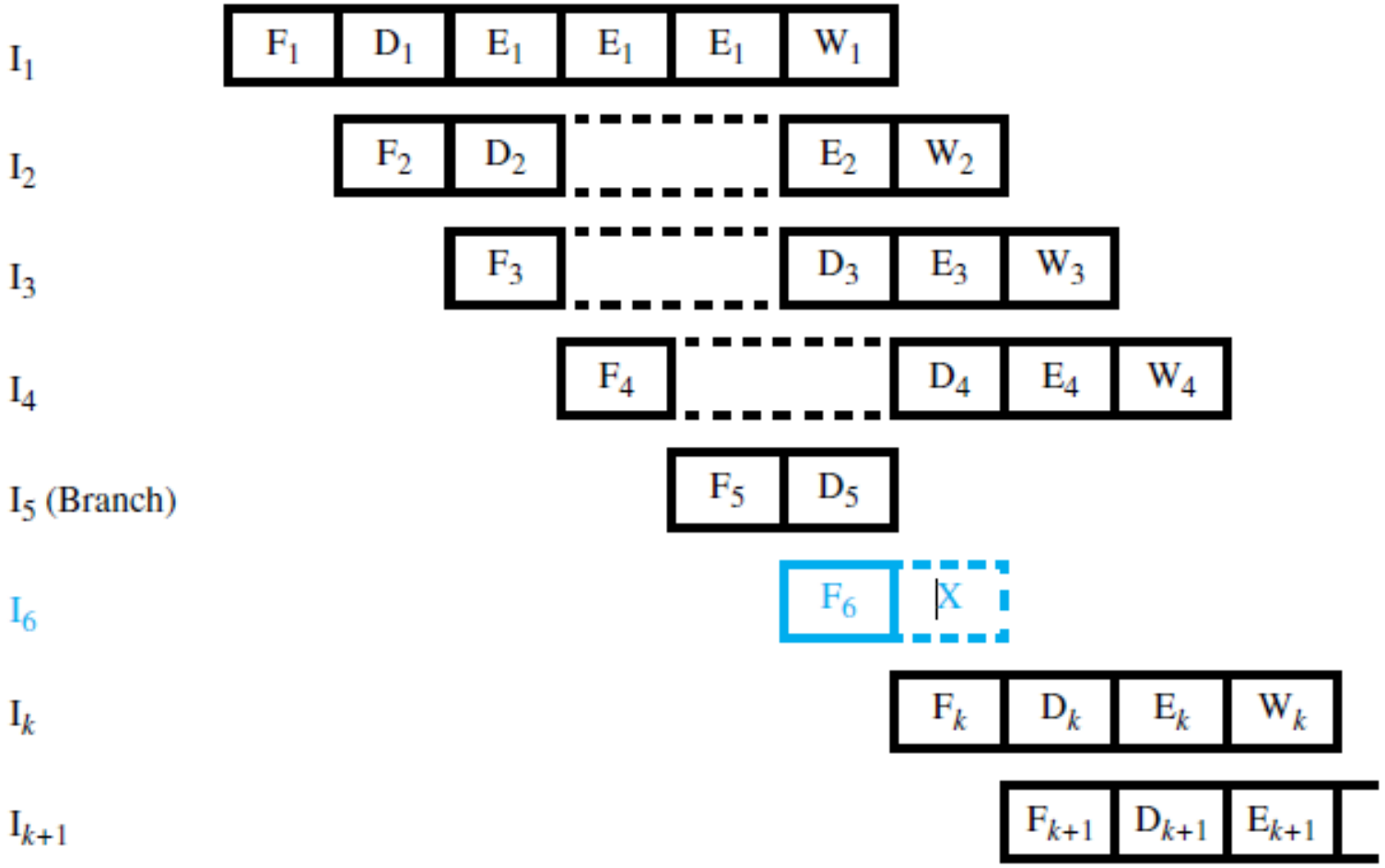
Instruction Queue and Prefetching

- Either a cache miss or a 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.



Use of an instruction queue in the hardware organization

Clock cycle	1	2	3	4	5	6	7	8	9	10
Queue length	1	1	1	1	2	3	2	1	1	1



Branch timing in the presence of an instruction queue. Branch target address is computed in the D stage.

- We have assumed that initially the queue contains one instruction. Every fetch operation adds one instruction to the queue and every dispatch operation reduces the queue length by one. Hence, the queue length remains the same for the first four clock cycles.
- Suppose that instruction I1 introduces a 2-cycle stall. Since space is available in the queue, the fetch unit continues to fetch instructions and the queue length rises to 3 in clock cycle 6.

- Instruction I5 is a branch instruction. Its target instruction, I*k , is fetched in cycle 7,* and instruction I6 is discarded.
- The branch instruction would normally cause a stall in cycle 7 as a result of discarding instruction I6.
- Instead, instruction I4 is dispatched from the queue to the decoding stage. After discarding I6, the queue length drops to 1 in cycle 8.
- The queue length will be at this value until another stall is encountered.

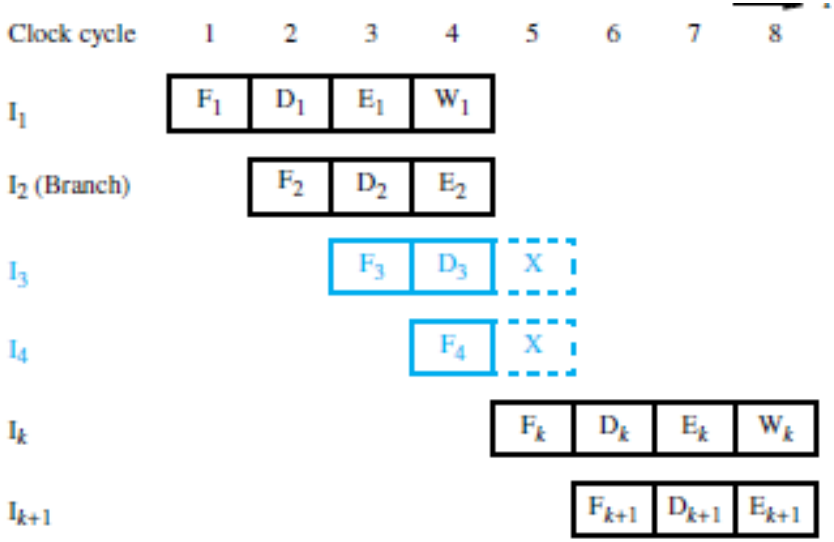
- Instructions I1, I2, I3, I4, and I*k* *complete execution in successive clock cycles.* ***Hence, the*** *branch instruction* does not increase the overall execution time.
- This is because the instruction fetch unit has executed the branch instruction (by computing the branch address) concurrently with the execution of other instructions. This technique is referred to as ***branch folding.***

CONDITIONAL BRANCHES AND BRANCH PREDICTION

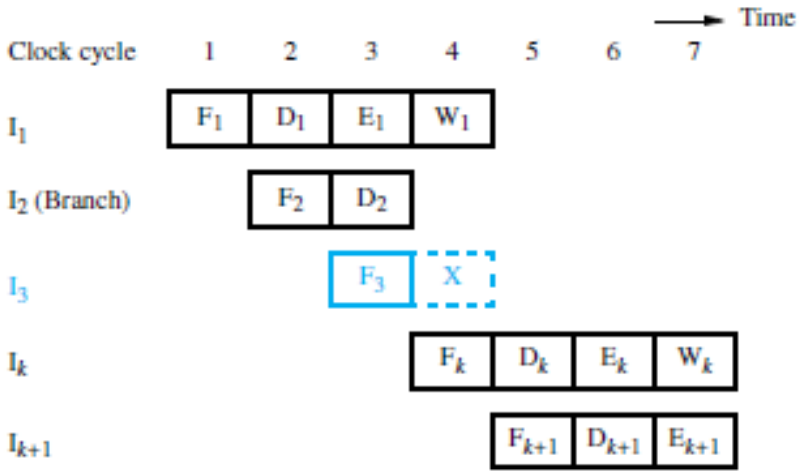
- A conditional branch instruction introduces the added hazard caused by the dependency of the branch condition on the result of a preceding instruction. The decision to branch cannot be made until the execution of that instruction has been completed.
- Branch instructions occur frequently. Because of the branch penalty, this large percentage would reduce the gain in performance expected from pipelining.
- Branch instructions can be handled in several ways to reduce their negative impact on the rate of execution of instructions.

Delayed Branch

- In Figure , the processor fetches instruction I3 before it determines whether the current instruction, I2, is a branch instruction. When execution of I2 is completed and a branch is to be made, the processor must discard I3 and fetch the instruction at the branch target. The location following a branch instruction is called a *branch delay slot*.
- There may be more than one branch delay slot, depending on the time it takes to execute a branch instruction. For example, there are two branch delay slots in Figure *a and* one delay slot in Figure *b.*



(a) Branch address computed in Execute stage



(b) Branch address computed in Decode stage

- A technique called *delayed branching* can minimize the penalty incurred as a result of conditional branch instructions. The idea is simple. The instructions in the delay slots are always fetched. Therefore, we would like to arrange for them to be fully executed whether or not the branch is taken.
- The objective is to be able to place useful instructions in these slots. If no useful instructions can be placed in the delay slots, these slots must be filled with NOP instructions.
- This situation is exactly the same as in the case of data dependency.

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

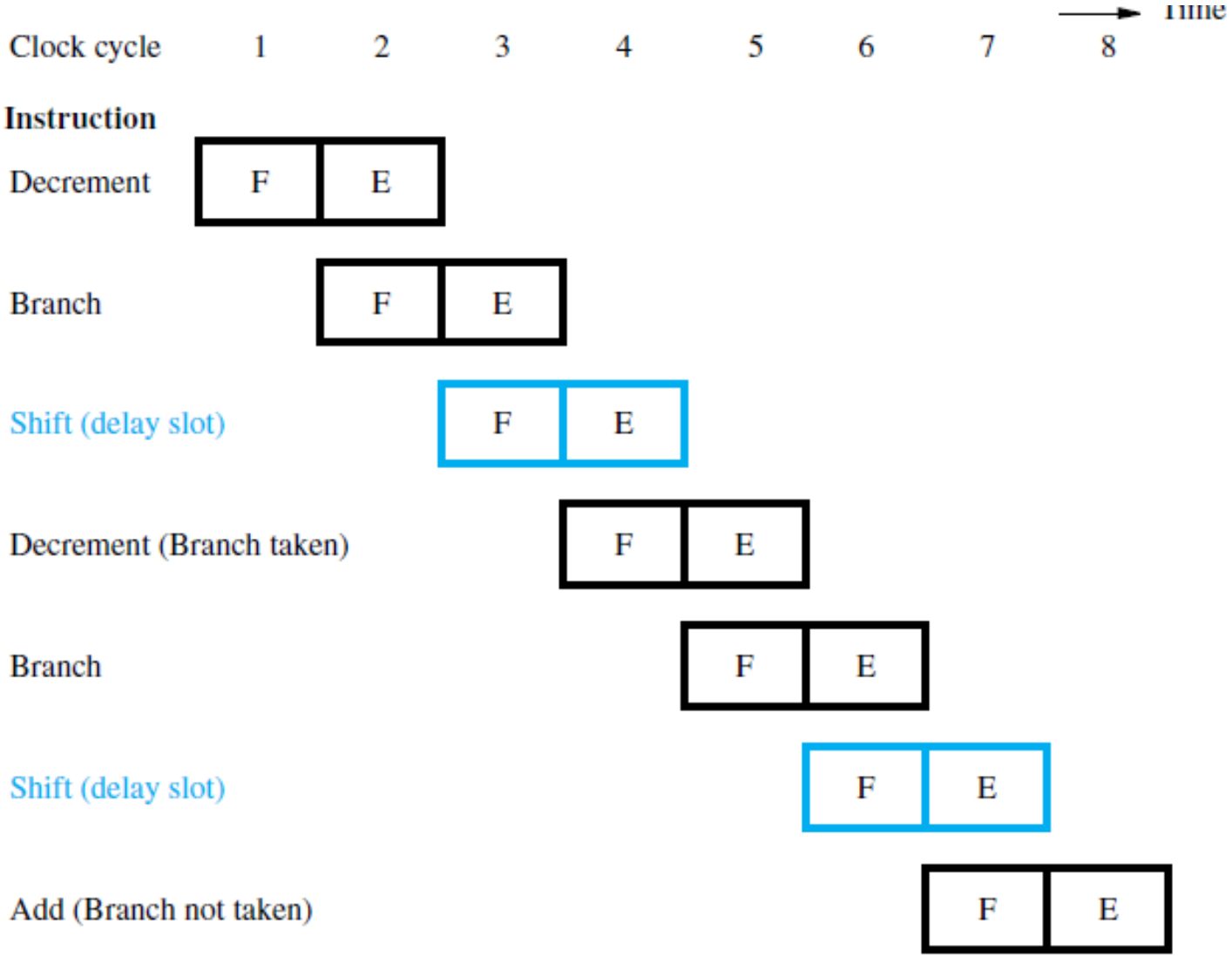
(a) Original program loop

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

(b) Reordered instructions

Reordering of instructions for a delayed branch.

- After evaluating the branch condition, the processor fetches the instruction at LOOP or at NEXT, depending on whether the branch condition is true or false, respectively.
- In either case, it completes execution of the shift instruction.



Execution timing showing the delay slot being filled during the last two passes through the loop in Figure *b*.

- *Register R2 is used as* a counter to determine the number of times the contents of register R1 are shifted left. For a processor with one delay slot, the instructions can be reordered as shown in Figure. *The shift instruction is fetched while the branch instruction is being* executed. After evaluating the branch condition, the processor fetches the instruction at LOOP or at NEXT, depending on whether the branch condition is true or false, respectively. In either case, it completes execution of the shift instruction.
- Pipelined operation is not interrupted at any time, and there are no idle cycles. Logically, the program is executed as if the branch instruction were placed after the shift instruction.
- Branching takes place one instruction later than where the branch instruction appears in the instruction sequence in the memory, hence the name **"delayed branch."**

Branch Prediction

- Another technique for reducing the branch penalty associated with conditional branches is to attempt to predict whether or not a particular branch will be taken.
- 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 registers or memory locations are updated until it is confirmed that these instructions should indeed be executed.
- If the branch decision indicates otherwise, the instructions and all their associated data in the execution units must be purged, and the correct instructions fetched and executed.
- A decision on which way to predict the result of the branch may be made in hardware by observing whether the target address of the branch is lower than or higher than the address of the branch instruction.
- A more flexible approach is to have the compiler decide whether a given branch instruction should be predicted taken or not taken. The branch prediction decision is always the same every time a given instruction is executed. Any approach that has this characteristic is called *static branch prediction.*

Dynamic Branch Prediction

- The objective of branch prediction algorithms is to reduce the probability of making a wrong decision, to avoid fetching instructions that eventually have to be discarded.
- 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.

THANK YOU