# Chapter 7. Basic Processing Unit & Pipelining

1. Some Fundamental Concepts
2. Execution of a complete Instruction
3. Pipelining: Basic concepts
4. Role of Cache memory
5. Pipeline Performance.

# Objectives

- How a processor executes instructions -Instruction Set
  - Processor (ISP).
  - The internal functional units of a processor and how they are interconnected-Central Processing Unit (CPU)
- A typical computing task consists of a series of steps specified by a sequence of machine instructions that constitute a program.
- How pipelining can improve throughput and reduce the overall execution time for a series of tasks.
- To identify and address hazards that can impact the smooth operation of a pipeline.

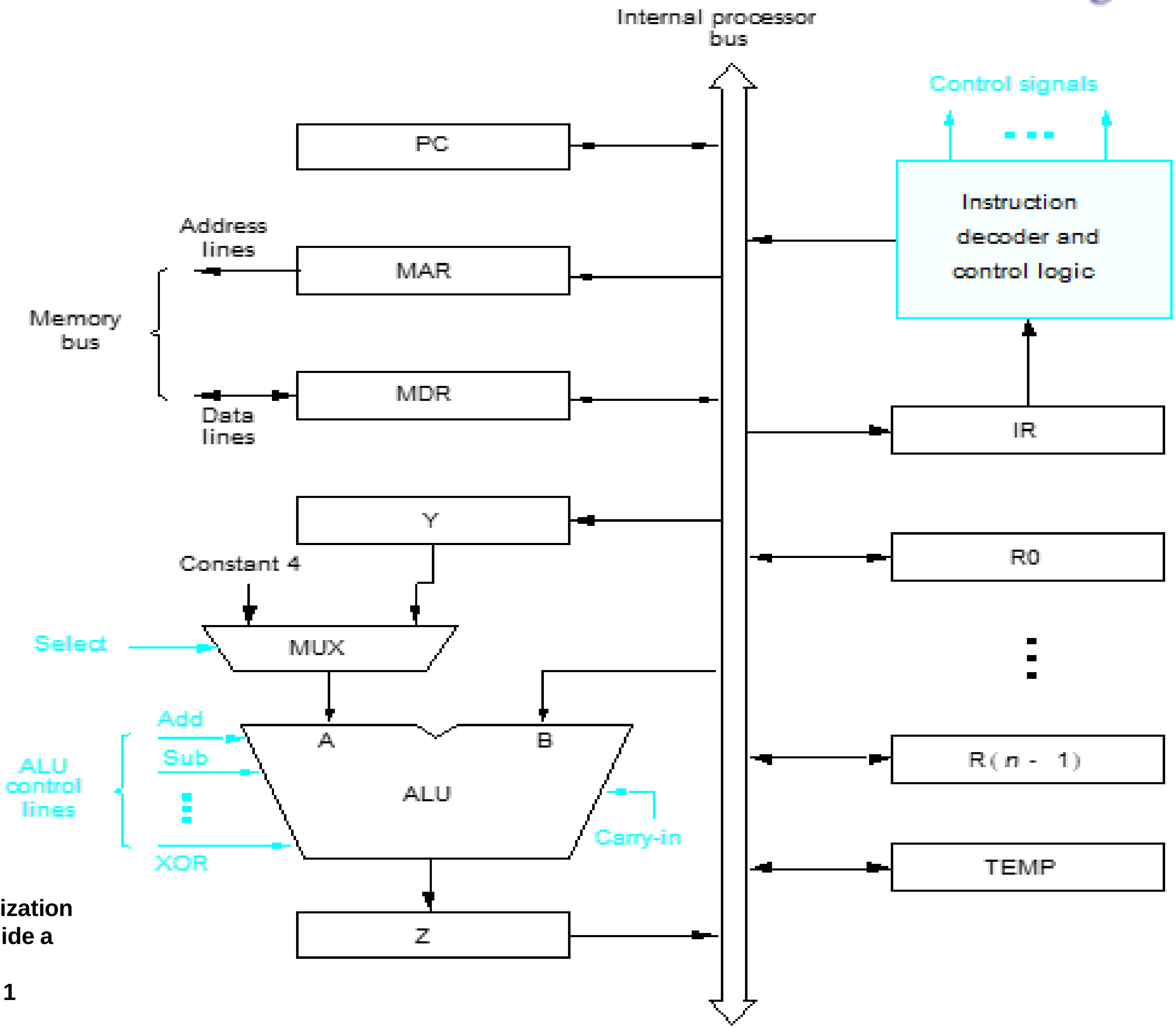
# 7.1 Some Fundamental Concepts

- Processor fetches one instruction at a time and perform the operation specified.
- Instructions are fetched from successive memory locations until a branch or a jump instruction is encountered.
- Processor keeps track of the address of the memory location containing the next instruction to be fetched using Program Counter (PC).
- Instruction Register (IR)

# Executing an Instruction

- Fetch the contents of the memory location pointed to by the PC. The contents of this location are loaded into the IR (fetch phase).
  - IR ← [[PC]]
- Assuming that the memory is byte addressable, increment the contents of the PC by 4 (fetch phase).
  - PC ← [PC] + 4
- Carry out the actions specified by the instruction in the IR (execution phase).

# Processor Organization



**MDR HAS TWO INPUTS AND TWO OUTPUTS**

**Fig 7.1**
**Single bus Organization of data path inside a processor**
**Ref: Module 1**

# Executing an Instruction

- Transfer a word of data from one processor register to another or to the ALU.
- Perform an arithmetic or a logic operation and store the result in a processor register.
- Fetch the contents of a given memory location and load them into a processor register.
- Store a word of data from a processor register into a given memory location.

## 7.1.1 Register Transfers

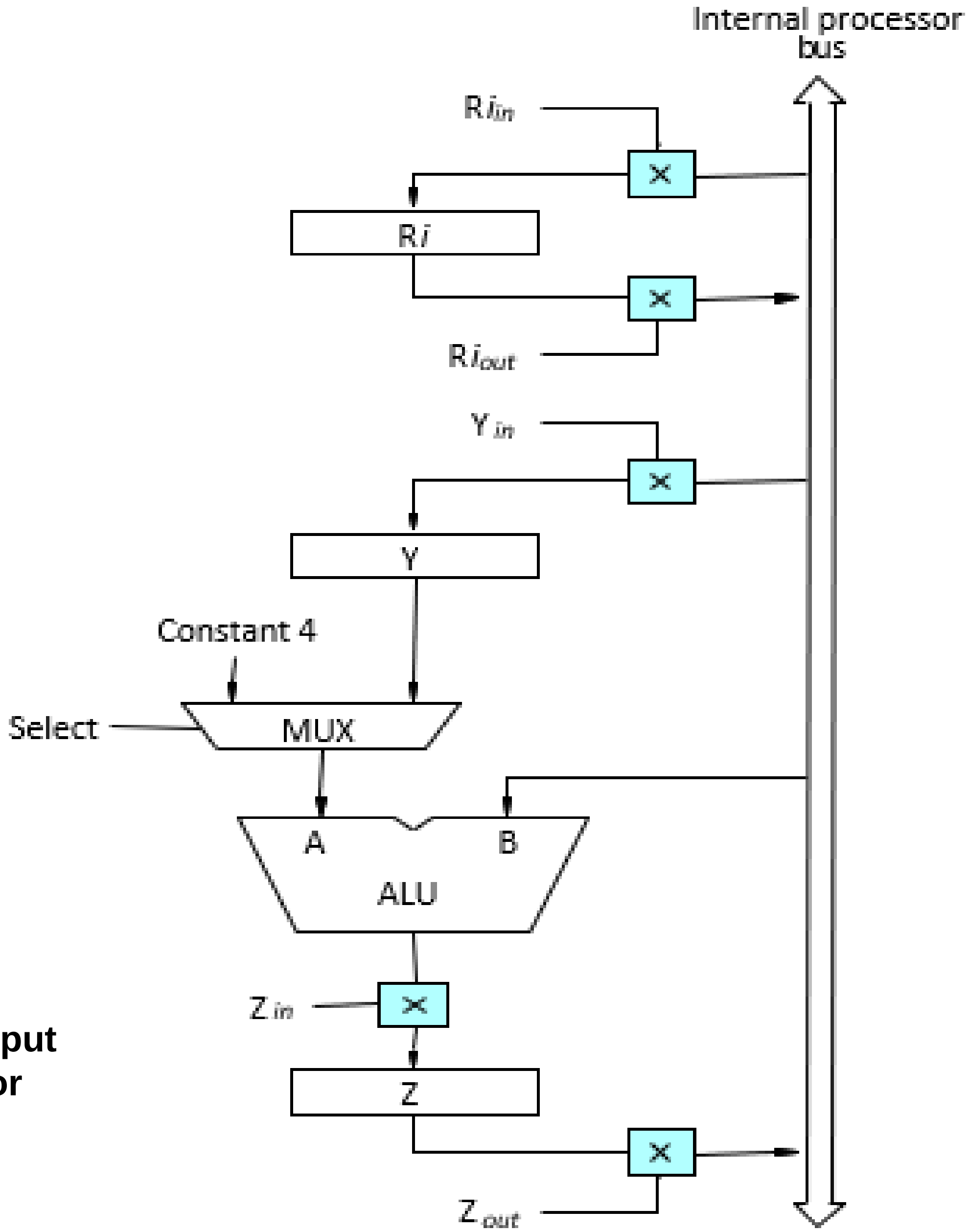


**Fig 7.2 Input and Out put gating for registers for fig 7.1**

# Register Transfers

$Ri_{in}$ is set to 1 , the data on the bus are loaded into Ri,

$Ri_{out}$ is set to 1 , the contents of register Ri are placed on the bus

$Ri_{out}$ is set to 0 , the bus can be used for transferring data from other registers

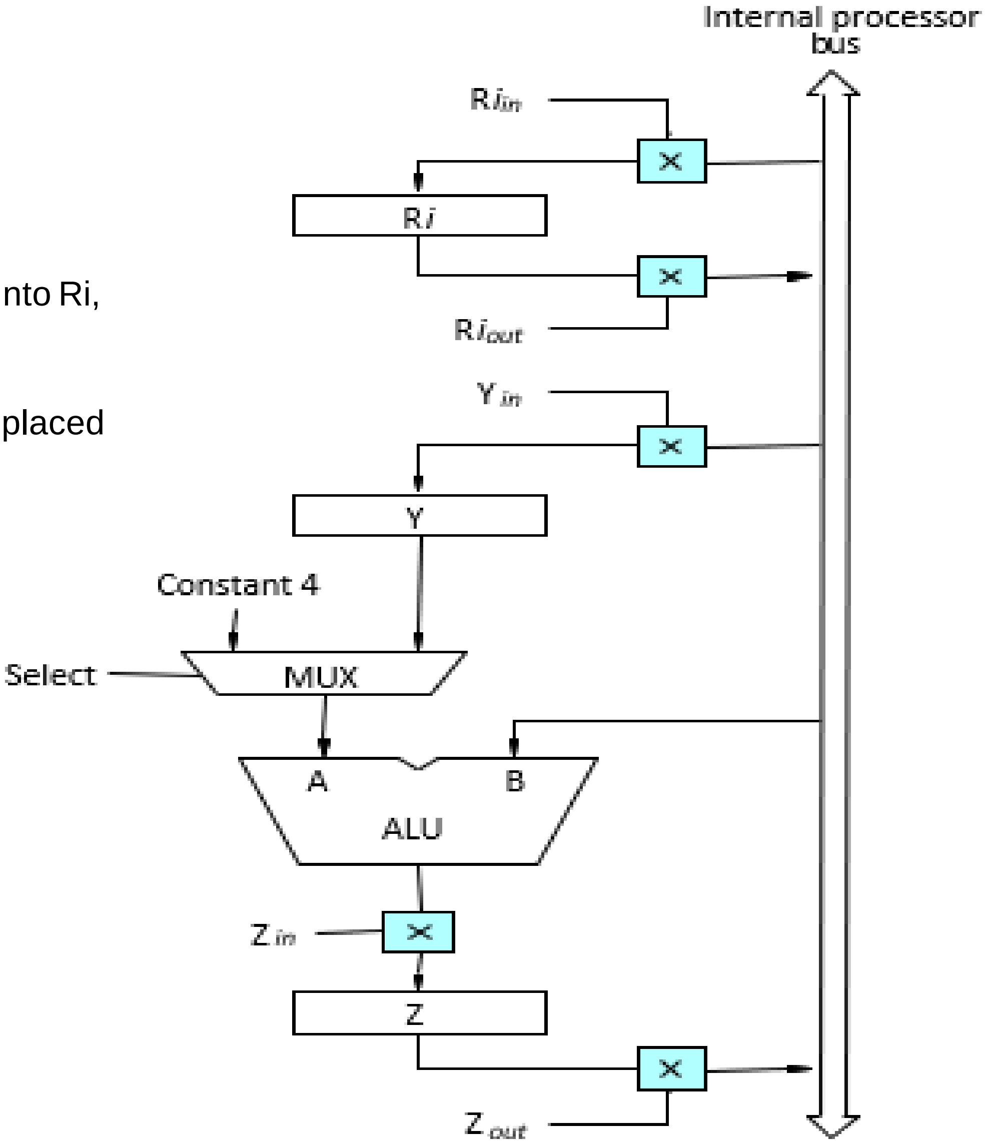


**Fig 7.2 Input and Out put gating for registers for fig 7.1**

# Register Transfers

- All operations and data transfers are controlled by the processor clock.

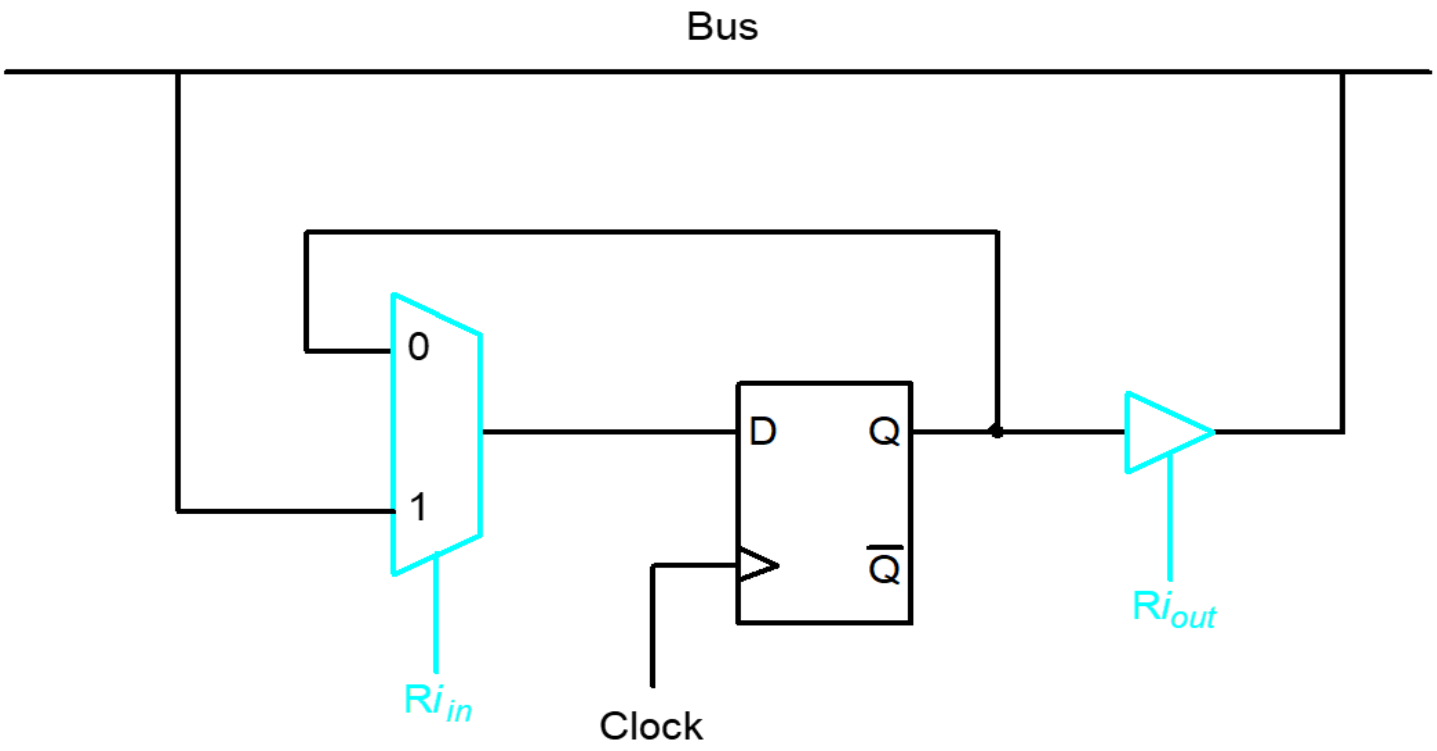


Figure 7.3. Input and output gting for one rgister bit.

# Register Transfers

- All operations and data transfers are controlled by the processor clock.
- A two input multiplexer is used to select the data to the input of an edge triggered D F/F.
- When $Ri_{in}$ is equal to 1 , the multiplexer selects the data on the bus.
- This data will be loaded into the f/f at the rising edge of the clock.
- $Ri_{in}$ is equal to 0, the multiplexer feeds back the value currently stored in the F/F.
- The Q o/p of F/F is connected to the bus.

★ $Ri_{out}$ = 0 , disconnected state , when 1 , gate drives the bus to 0 or 1, depends on the value of Q

# 7.1.2 Performing an Arithmetic or Logic Operation

- ★ The ALU is a combinational circuit that has no internal storage.
- ALU gets the two operands from MUX and bus.
- The result is temporarily stored in register Z.
- What is the sequence of operations to add the contents of register R1 to those of R2 and store the result in R3?
  - R1out, Yin
  - R2out, SelectY, Add, Zin
  - Zout, R3in

# 7.1.3 Fetching a Word from Memory

- The response time of each memory access varies (cache miss, memory-mapped I/O,…).
- To accommodate this, the processor waits until it receives an indication that the requested operation has been completed (Memory-Function-Completed, MFC).
- Move (R1), R2
- MAR ← [R1]
- Start a Read operation on the memory bus
- Wait for the MFC response from the memory
- Load MDR from the memory bus
- R2 ← [MDR]

# Storing a word in memory

- Address is loaded into MAR
- Data to be written loaded into MDR.
- Write command is issued.
- Example: Move R2,(R1)

$R1_{out}$,$MAR_{in}$

$R2_{out}$,$MDR_{in}$,Write

$MDR_{outE}$, WMFC

# Timing

MAR ← [R1]

Assume MAR
is always available
on the address lines
of the memory bus.

Start a Read operation on the memory bus

Wait for the MFC response from the memory

Load MDR from the memory bus

R2 ← [MDR]

# Timing

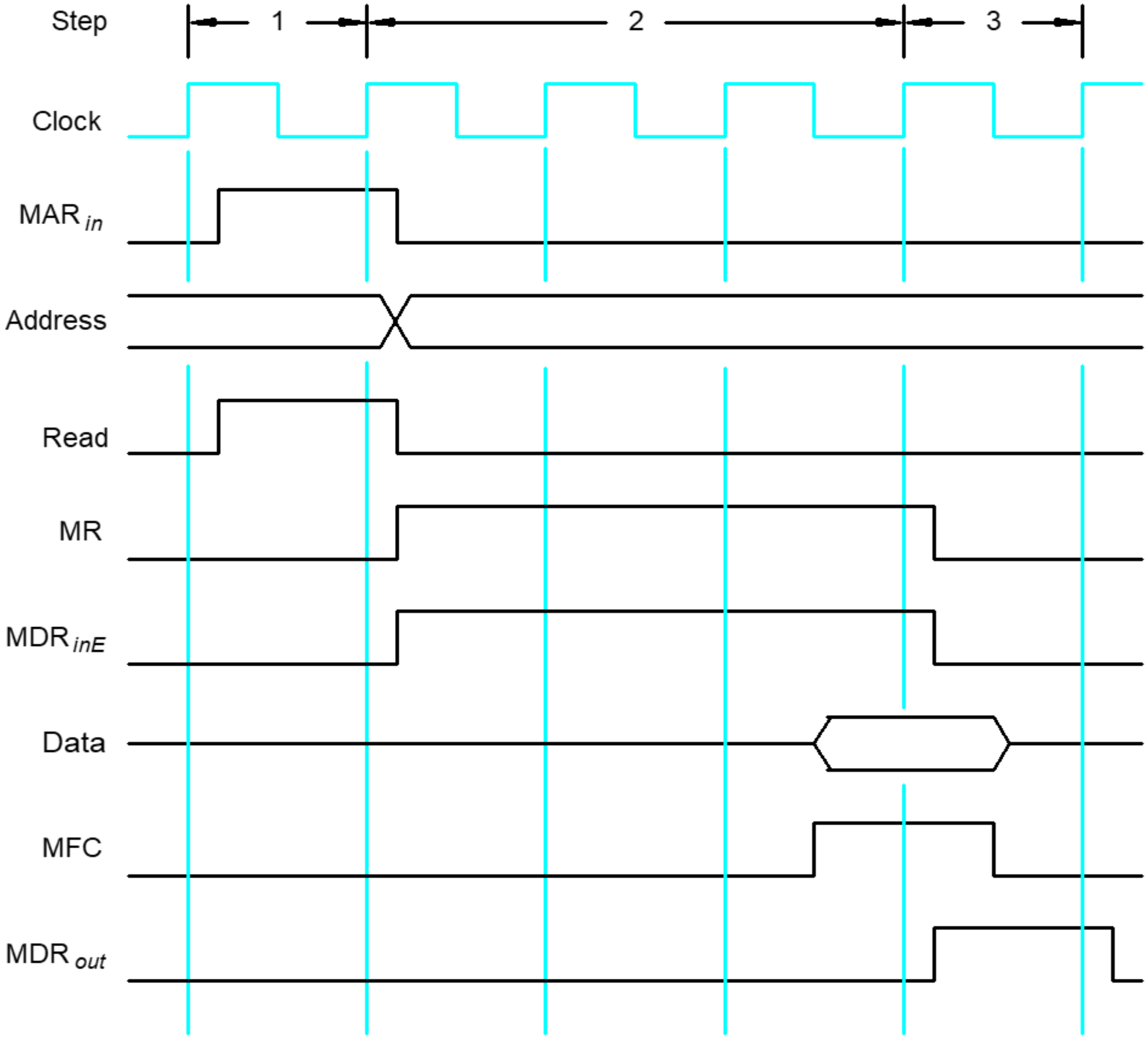


Figure 7.5. Timing of a memory Read operation.

# Execution of a Complete Instruction

- Add (R3), R1
- Fetch the instruction
- Fetch the first operand (the contents of the memory location pointed to by R3)
- Perform the addition
- Load the result into R1

# Execution of a Complete Instruction

Add (R3), R1

| Step | Action |
|---|---|
| 1 | $PC_{out}$ , $MAR_{in}$ , Read, Select4,Add, $Z_{in}$ |
| 2 | $Z_{out}$ , $PC_{in}$ , $Y_{in}$ , WMF C |
| 3 | $MDR_{out}$ , $IR_{in}$ |
| 4 | $R3_{out}$ , $MAR_{in}$ , Read |
| 5 | $R1_{out}$ , $Y_{in}$ , WMF C |
| 6 | $MDR_{out}$ , SelectY, Add, $Z_{in}$ |
| 7 | $Z_{out}$ , $R1_{in}$ , End |

Figure 7.6. Control sequence for execution of the instruction Add (R3),R1.

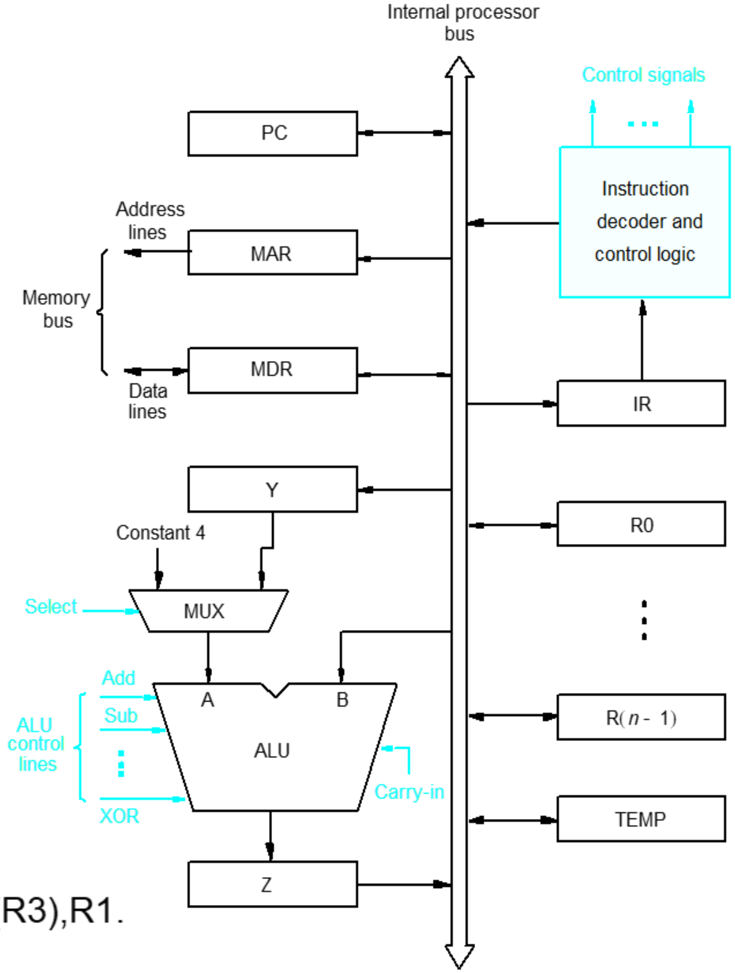


Figure 7.1. Single-bus organization of the datapath inside a proce

# Execution of Branch Instructions

- A branch instruction replaces the contents of PC with the branch target address, which is usually obtained by adding an offset X given in the branch instruction.
- The offset X is usually the difference between the branch target address and the address immediately following the branch instruction.
- Conditional branch

# Execution of Branch Instructions

| Step | Action |
|---|---|
| 1 | $PC_{out}$, $MAR_{in}$, Read, Select4, Add, $Z_{in}$ |
| 2 | $Z_{out}$, $PC_{in}$, $Y_{in}$, WMF C |
| 3 | $MDR_{out}$, $IR_{in}$ |
| 4 | Offset-field-of-$IR_{out}$, Add, $Z_{in}$ |
| 5 | $Z_{out}$, $PC_{in}$, End |

Figure 7.7. Control sequence for an unconditional branch instruction.

# Multiple-Bus Organization

- Add R4, R5, R6

| Step | Action |
|---|---|
| 1 | $PC_{out}$, R=B, $MAR_{in}$, Read, IncPC |
| 2 | WMF C |
| 3 | $MDR_{outB}$, R=B, $IR_{in}$ |
| 4 | $R4_{outA}$, $R5_{outB}$, SelectA, Add, $R6_{in}$, End |

Figure 7.9. Control sequence for the instruction. Add R4,R5,R6, for the three-bus organization in Figure 7.8.

# Quiz

- What is the control sequence for execution of the instruction

  Add R1, R2

  including the instruction fetch phase? (Assume single bus architecture)

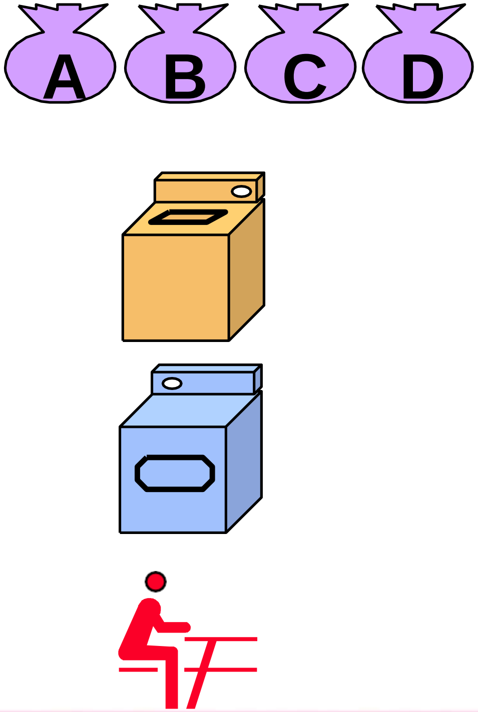
# Basic Concepts of Pipe Lining

# Making the Execution of Programs Faster

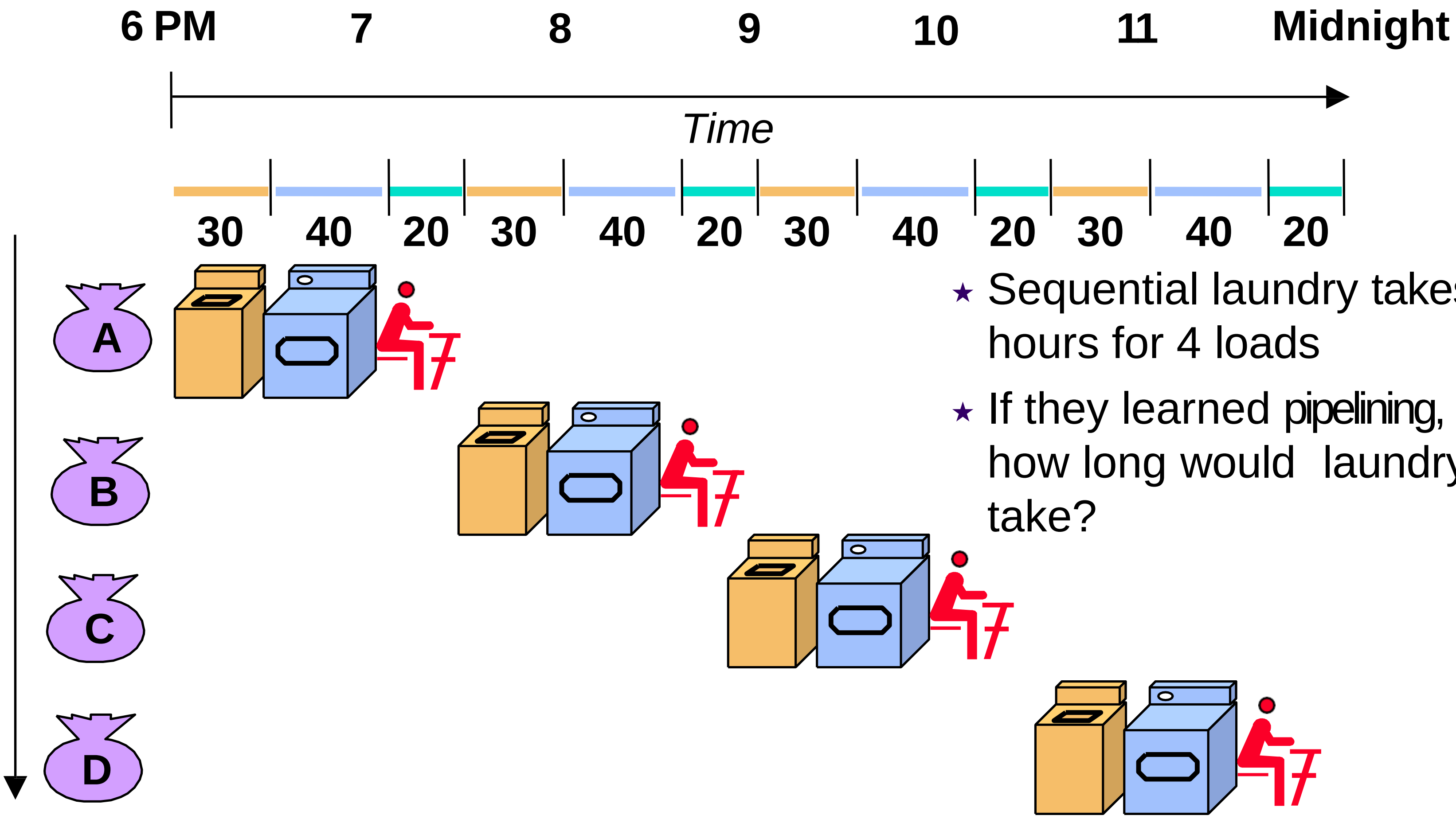
- Use faster circuit technology to build the processor and the main memory.
- Arrange the hardware so that more than one operation can be performed at the same time.
- In the latter way, the number of operations performed per second is increased even though the elapsed time needed to perform any one operation is not changed.

# Traditional Pipeline Concept

- ⋆Laundry Example
- ⋆Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold
- ⋆Washer takes 30 minutes
- ⋆Dryer takes 40 minutes
- ⋆“Folder” takes 20 minutes

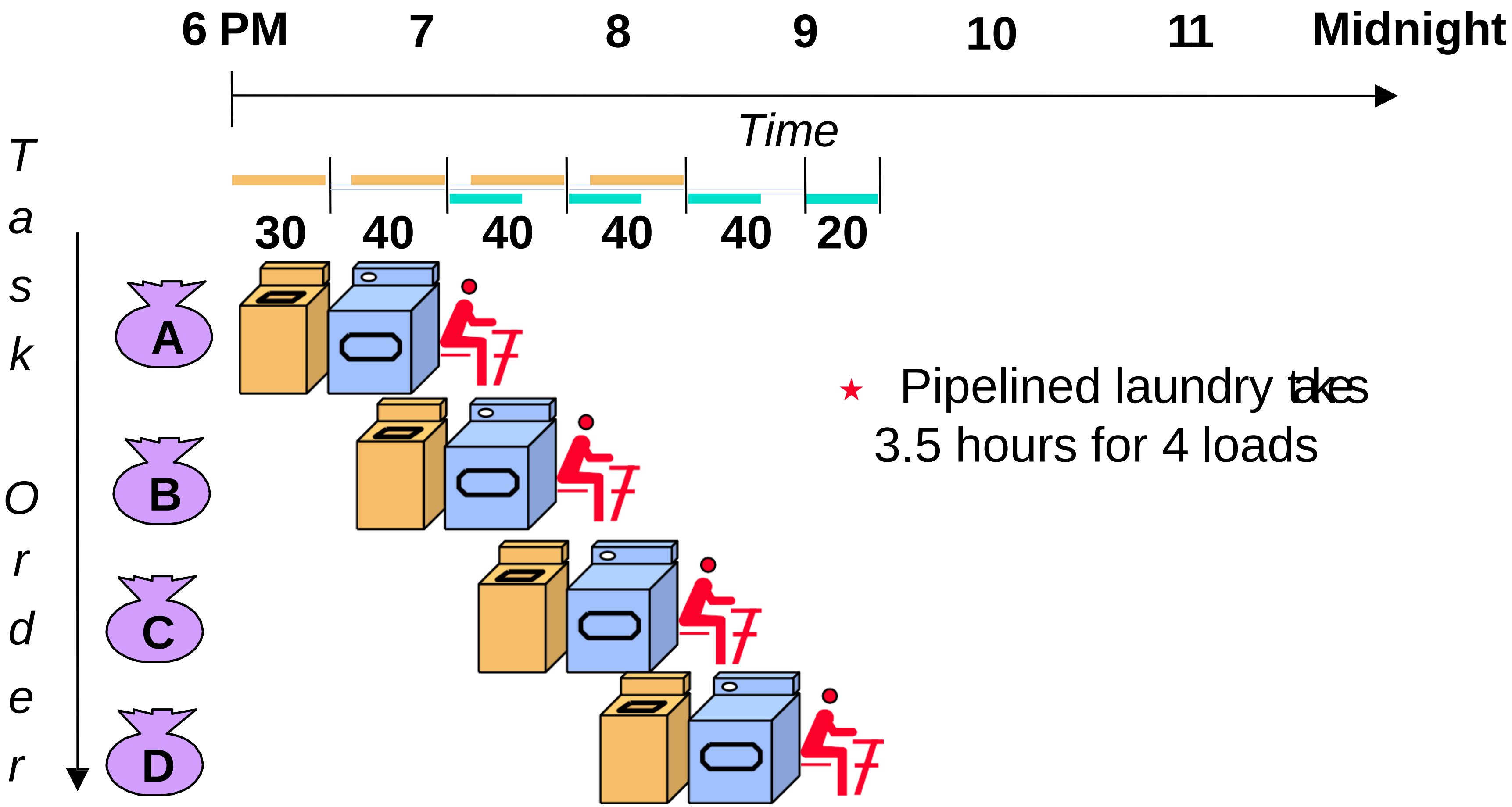


# Traditional Pipeline Concept



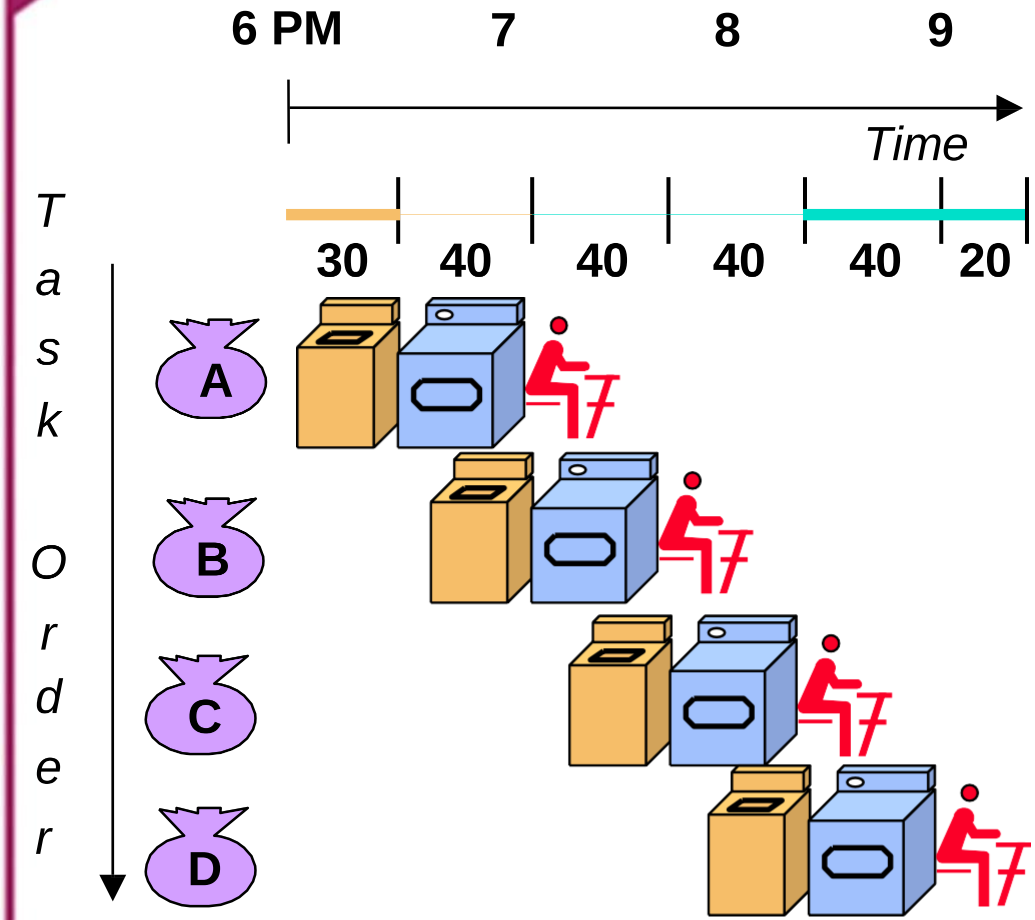
- Sequential laundry takes 6 hours for 4 loads
- If they learned pipelining, how long would laundry take?

# Traditional Pipeline Concept



- ★ Pipelined laundry takes 3.5 hours for 4 loads

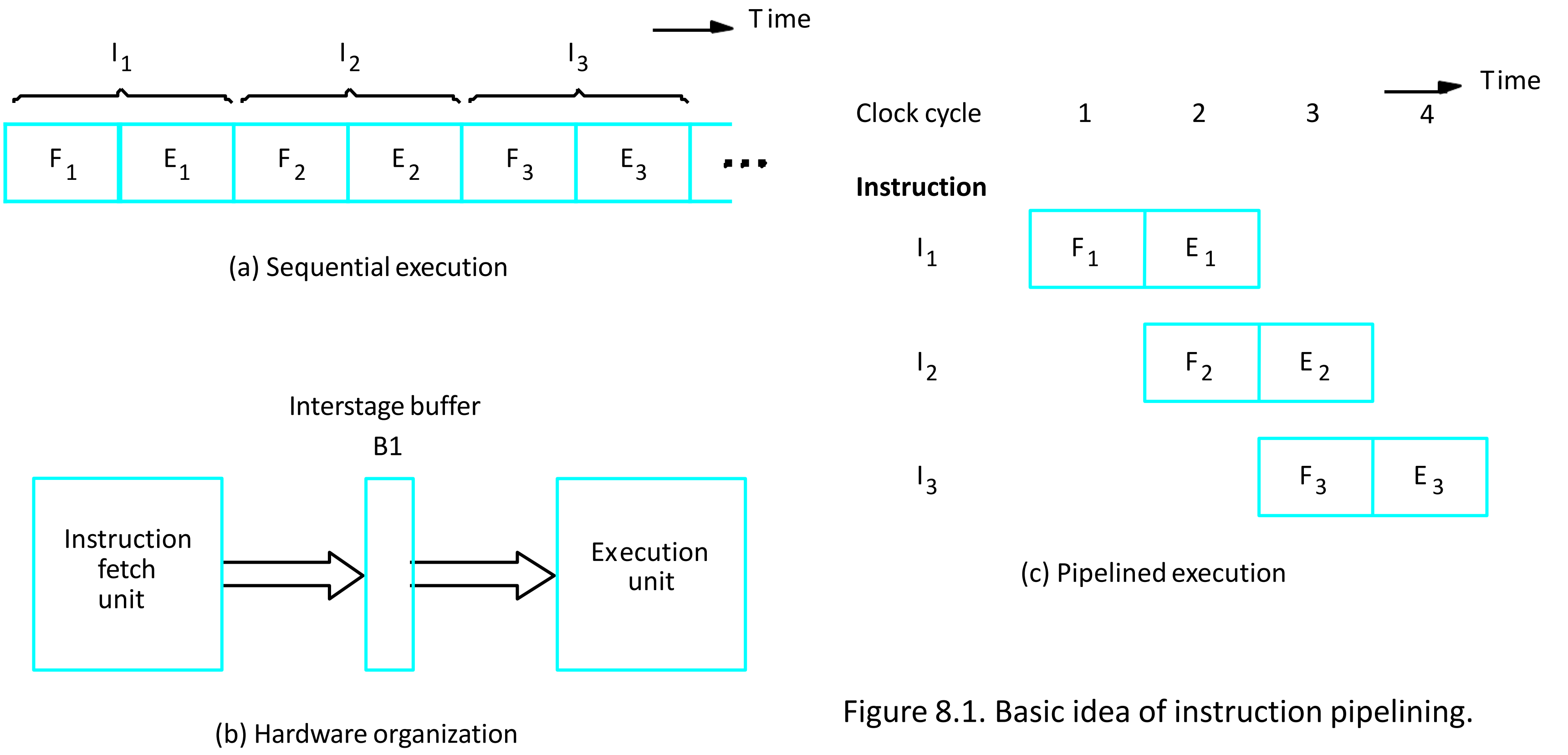
# Traditional Pipeline Concept



- ⋆ Pipelining doesn’t help latency of single task, it helps throughput of entire workload
  - ⋆ Pipeline rate limited by slowest pipeline stage
- ⋆ Multiple tasks operating simultaneously using different resources
- ⋆ Potential speedup = Number pipe stages
- ⋆ Unbalanced lengths of pipe stages reduces speedup
- ⋆ Time to “fill” pipeline and time to “drain” it reduces speedup
- ⋆ Stall for Dependences

# Use the Idea of Pipelining in a Computer

Fetch + Execution



(a) Sequential execution

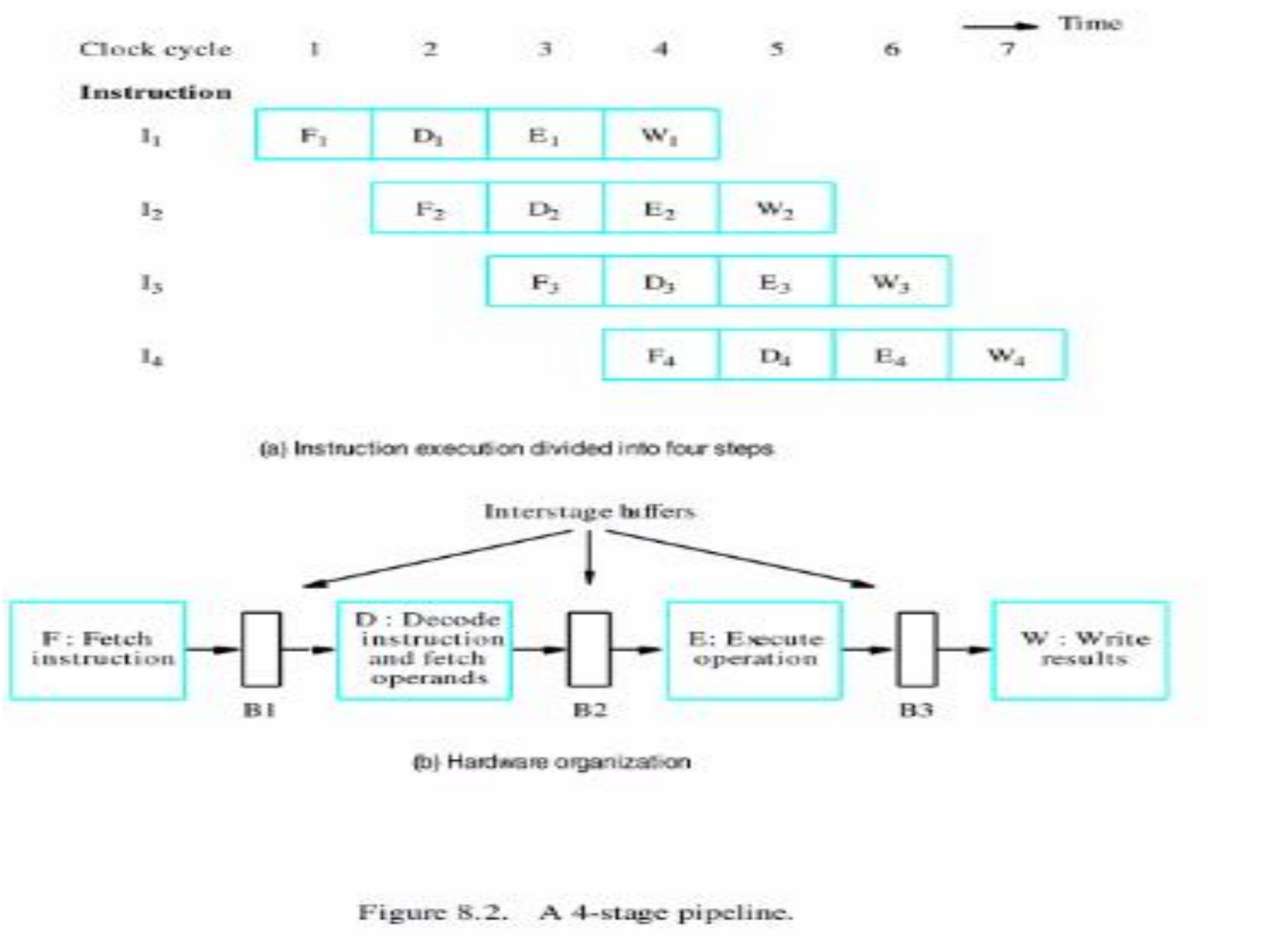
(b) Hardware organization

(c) Pipelined execution

Figure 8.1. Basic idea of instruction pipelining.

# Use the Idea of Pipelining in a Computer

Fetch + Decode
+ Execution + Write



→ Time

| Clock cycle | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| **Instruction** | | | | | | | |
| $I_1$ | $F_1$ | $D_1$ | $E_1$ | $W_1$ | | | |
| $I_2$ | | $F_2$ | $D_2$ | $E_2$ | $W_2$ | | |
| $I_3$ | | | $F_3$ | $D_3$ | $E_3$ | $W_3$ | |
| $I_4$ | | | | $F_4$ | $D_4$ | $E_4$ | $W_4$ |

(a) Instruction execution divided into four steps

(b) Hardware organization

Figure 8.2. A 4-stage pipeline.

Textbook page: 457

# Role of Cache Memory

- ★ Each pipeline stage is expected to complete in one clock cycle.
- ★ The clock period should be long enough to let the slowest pipeline stage to complete.
- ★ Faster stages can only wait for the slowest one to complete.
- ★ Since main memory is very slow compared to the execution, if each instruction needs to be fetched from main memory, pipeline is almost useless.
- ★ Fortunately, we have cache.

# Pipeline Performance

- The potential increase in performance resulting from pipelining is proportional to the number of pipeline stages.
- However, this increase would be achieved only if all pipeline stages require the same time to complete, and there is no interruption throughout program execution.
- Unfortunately, this is not true.

# Pipeline Performance

- The previous pipeline is said to have been stalled for two clock cycles.
- Any condition that causes a pipeline to stall is called a hazard.
- Data hazard – any condition in which either the source or the destination operands of an instruction are not available at the time expected in the pipeline. So some operation has to be delayed, and the pipeline stalls.
- Instruction (control) hazard – a delay in the availability of an instruction causes the pipeline to stall.
- Structural hazard – the situation when two instructions require the use of a given hardware resource at the same time.

# Data Hazard

- Ensure that the results obtained when instructions are executed in pipeline processor identical to those obtained when the same instructions are executed sequentially.
- Hazard Occurs

  A←3+ A
- No Hazard

  A←5 * C

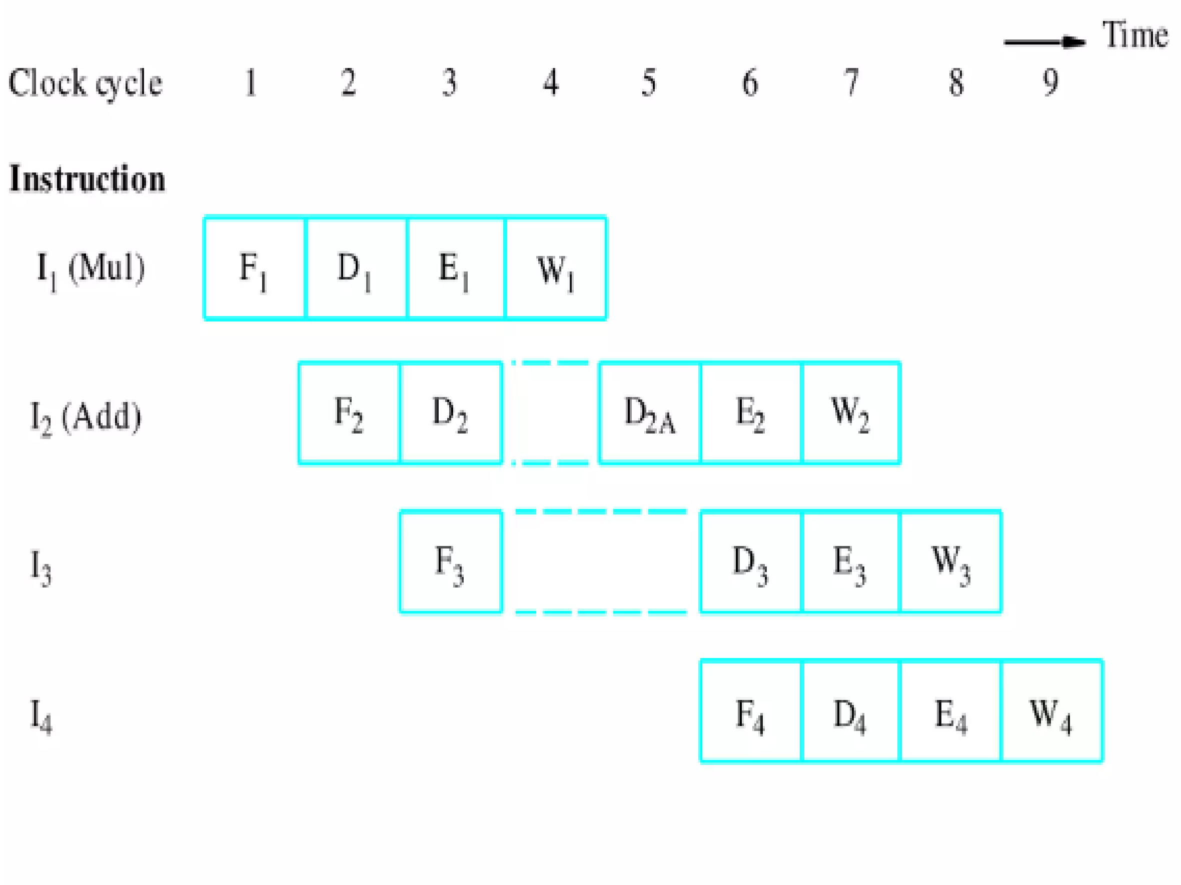
  B←20 + C
- When two operations are dependent, they should execute sequentially in correct order.

# Data Hazard

- Example:

  Mul R2,R3,R4

  Add  R5,R4,R6



| Clock cycle | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| **Instruction** | | | | | | | | | |
| $I_1$ (Mul) | $F_1$ | $D_1$ | $E_1$ | $W_1$ | | | | | |
| $I_2$ (Add) | | $F_2$ | $D_2$ | | $D_{2A}$ | $E_2$ | $W_2$ | | |
| $I_3$ | | | $F_3$ | | | $D_3$ | $E_3$ | $W_3$ | |
| $I_4$ | | | | | | $F_4$ | $D_4$ | $E_4$ | $W_4$ |

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

# Data Dependency Solutions

- Hardware Interlocks-is a circuit that detect instructions whose source operands are destination of instructions.
- Operand Forwarding-Uses a special h/w to  detect conflict and avoid it by using data through special path between pipeline segments.
- Delayed Load-compiler for computer is designed to reorder the instructions to delay the loading of the conflicting data by inserting No- Operation

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

# Instructional Hazard

- One of the major problem in operating the instruction pipeline is the occurrence of branch instruction .
- 1-Unconditional Branch-always change the sequential program flow by loading the program counter with target address.
- 2-Condtional Branch-the control select the target instruction if condition is satisfied or the next sequential instruction if condition is not satisfied.

# 1-Unconditional Branch

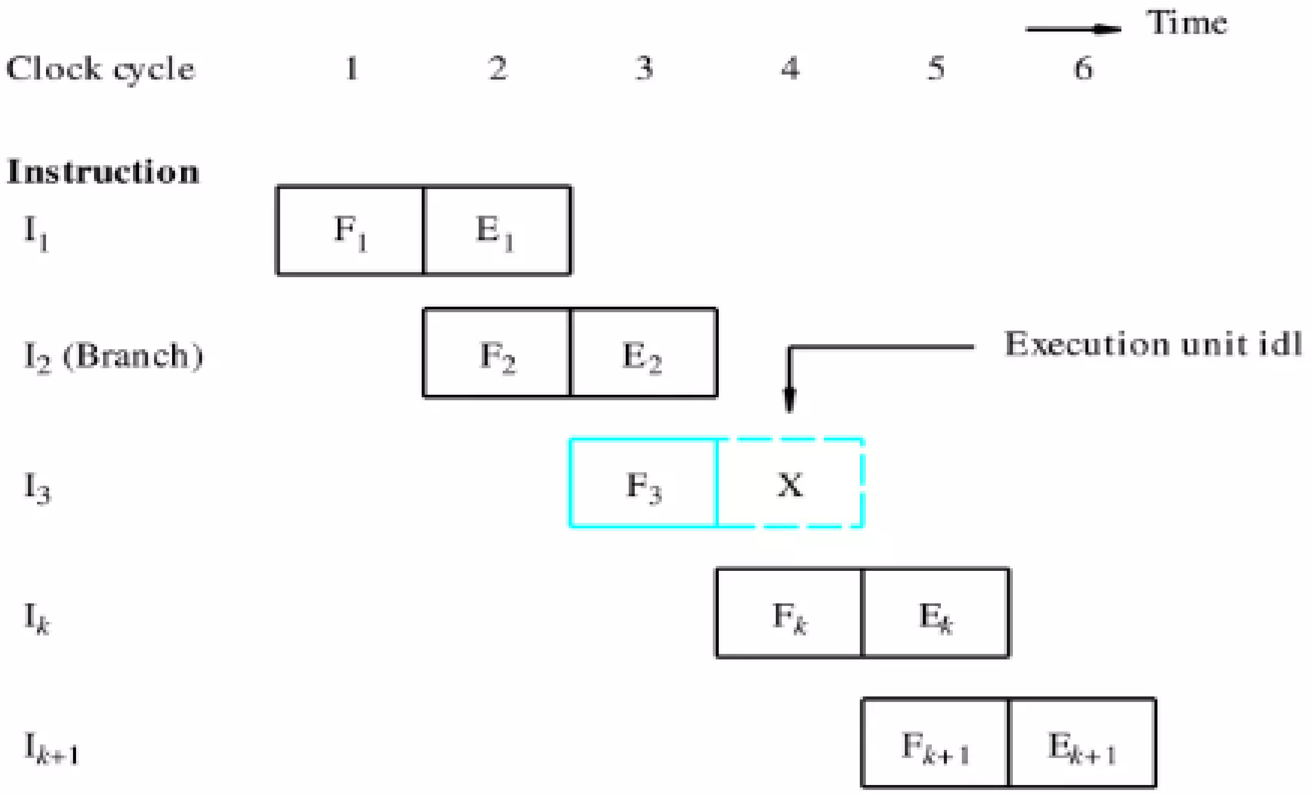


Figure 8.8. An idle cycle caused by a branch instruction.

# Unconditional Branches

- The time lost as result of  branch instruction is referred as branch penalty.
- The previous example I3 instruction is wrongly fetched and branch target address k will discard the I3.
- The fetch unit has dedicated H/W  which identify the branch target address as soon as possible after an instruction is fetched.

# Instruction Queue & Fetching

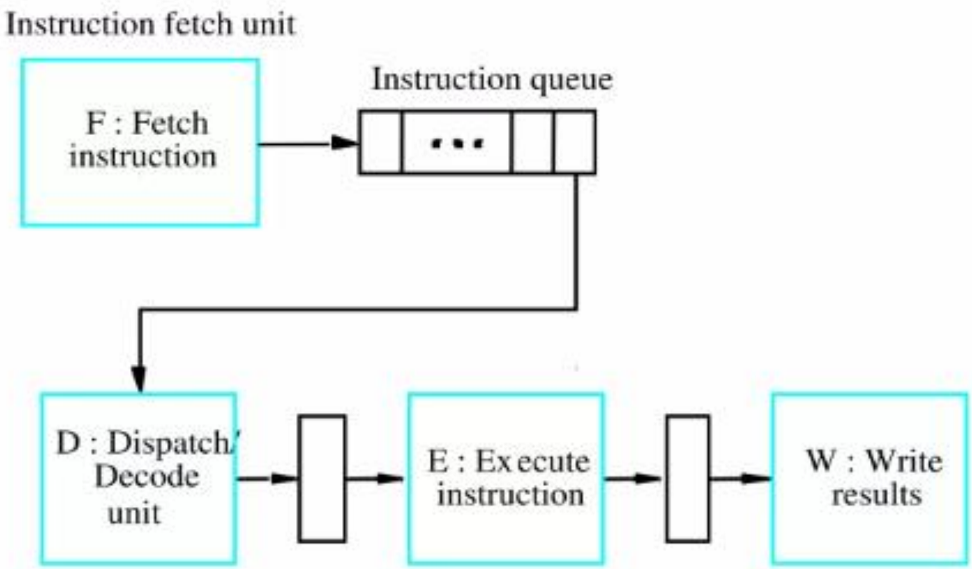
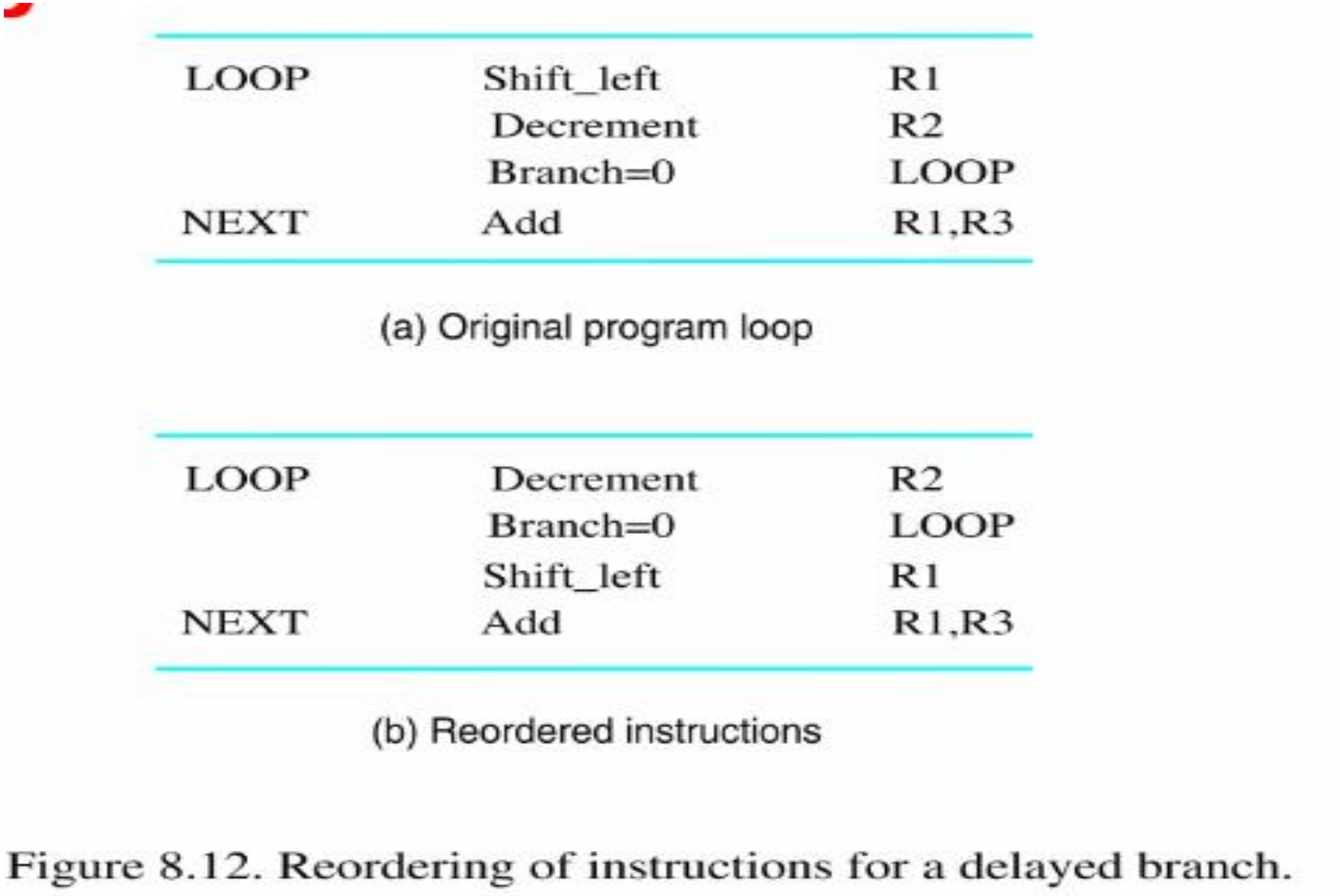


Figure 8.10. Use of an instruction queue in the hardware organization of Figure 8.2*b*.

# 2-Conditional Branch

- A conditional branch instruction introduces the added hazard cause by the dependency of branch condition on a result of previous instruction.
- The decision to branch cannot be made until the execution of that instruction has  been completed.

# Delayed Branch



```
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

Figure 8.12. Reordering of instructions for a delayed branch.