Vel Tech
Rangarajan Dr. Sagunthala
R&D Institute of Science and Technology
(Deemed to be University Estd. u/s 3 of UGC Act, 1956)

Department of Computer Science and Engineering

COURSE CODE – TITLE: 1151CS110 – COMPUTER ORGANIZATION AND ARCHITECTURE

Course Instructor
Dr. M. Rajeev Kumar
Associate Professor (CSE)

UNIT III- Processor and Control Unit

CO Nos.	Course Outcome(s)	Level of learning domain (Based on revised Bloom’s)
CO3	Design a pipeline for consistent execution of instructions with minimum hazards.	K3

BASIC MIPS IMPLEMENTATION

Overview

- Instruction Set Processor (ISP)
- 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.
- An instruction is executed by carrying out a sequence of more rudimentary operations.

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).

$$IR \leftarrow [[PC]]$$

- Assuming that the memory is byte addressable, increment the contents of the PC by 4 (fetch phase).

$$PC \leftarrow [PC] + 4$$

- Carry out the actions specified by the instruction in the IR (execution phase).

Processor Organization

MDR HAS TWO INPUTS AND TWO OUTPUTS

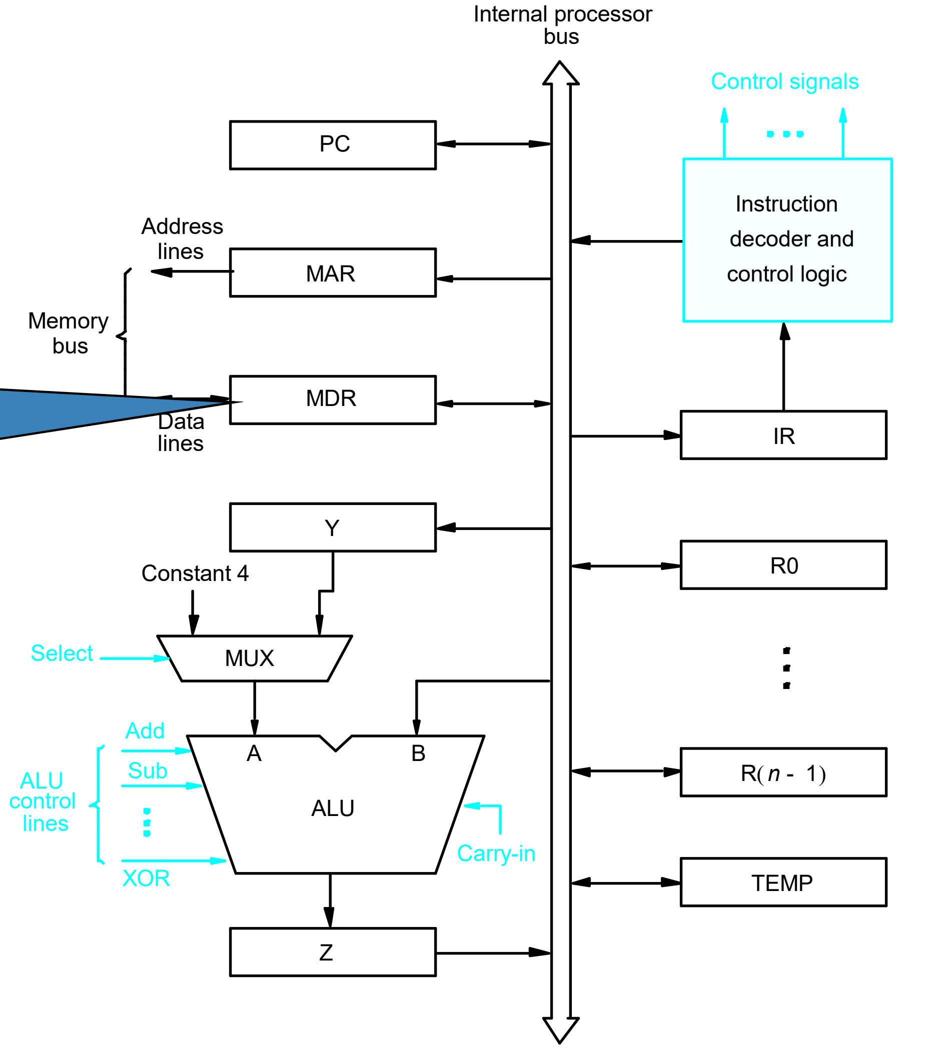
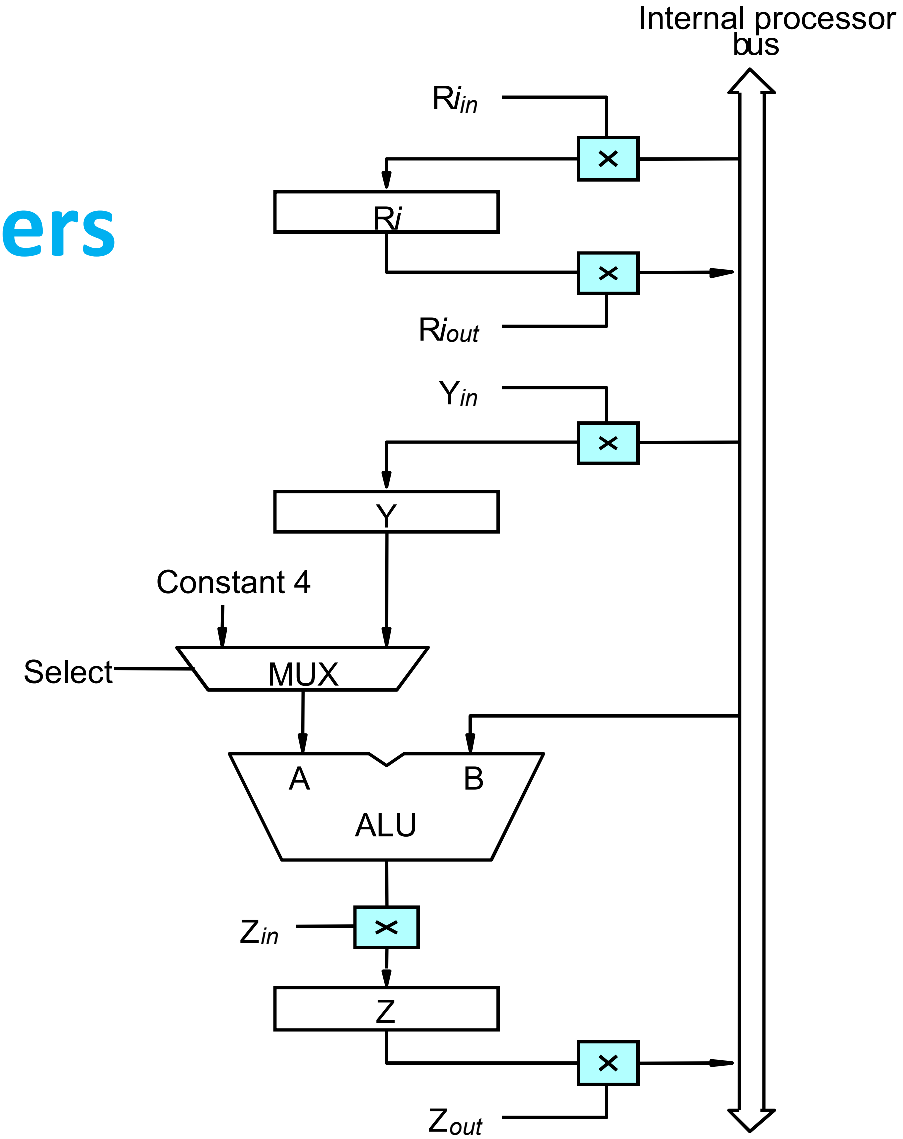


Fig. Single-Bus Datapath Organization

Executing an Instruction

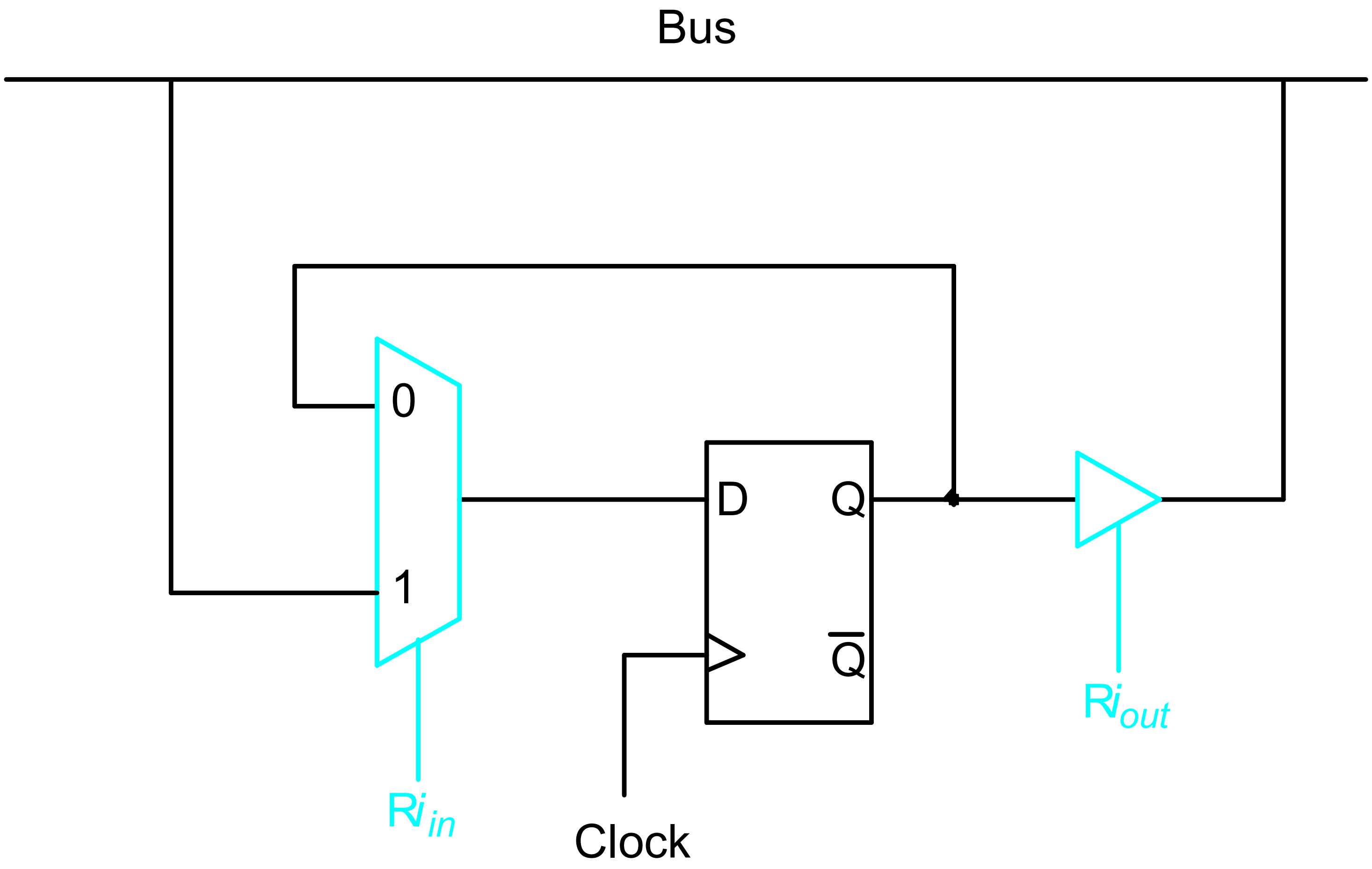
- Transfer a word of data from one processor register to another or to the ALU.
- Perform an arithmetic or a logic operation and store the result in a processor register.
- Fetch the contents of a given memory location and load them into a processor register.
- Store a word of data from a processor register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

Register Transfers



Register Transfers

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



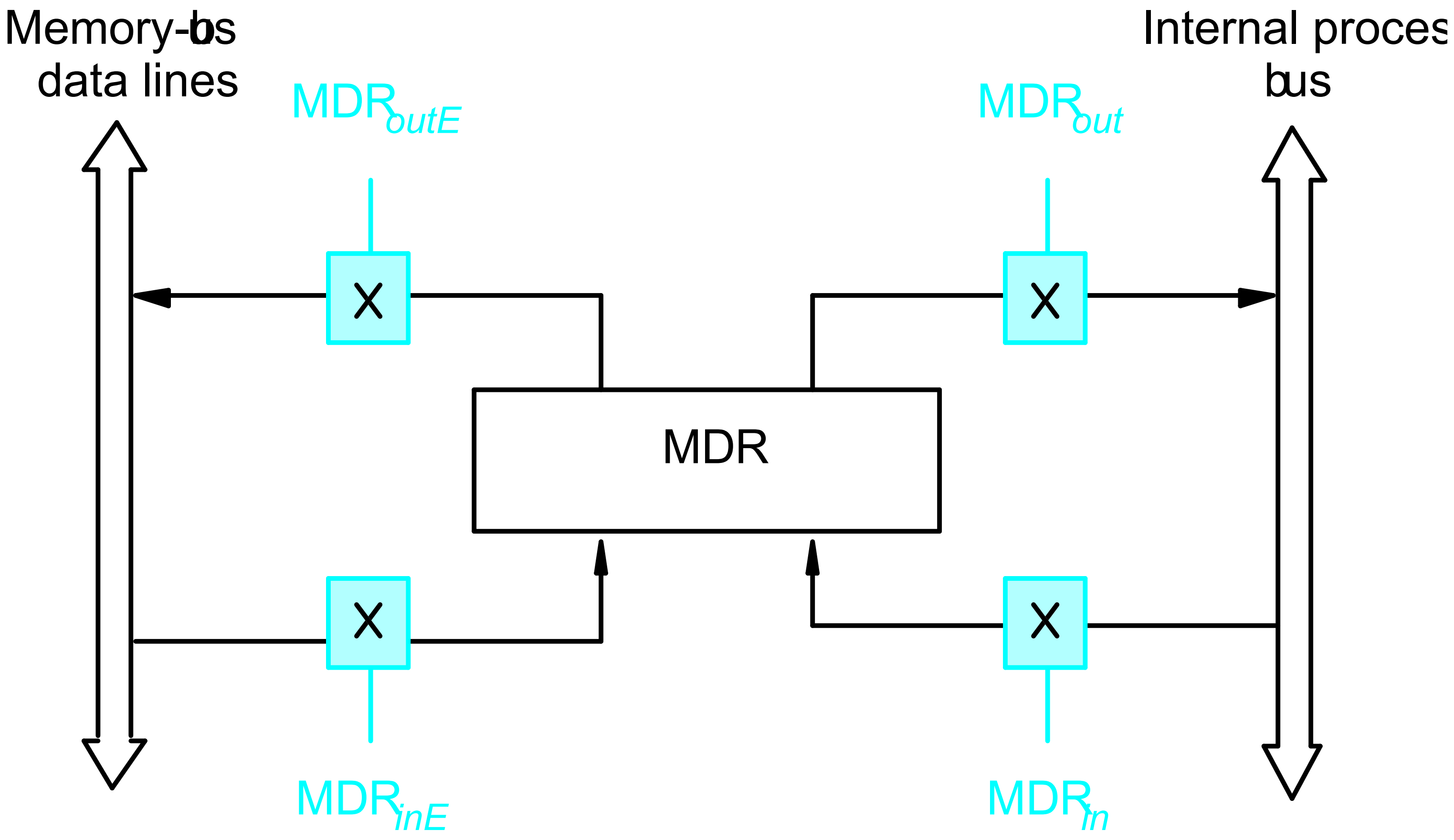
Performing an Arithmetic or Logic Operation

- 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?

1. R1out, Yin
2. R2out, SelectY, Add, Zin
3. Zout, R3in

Fetching a Word from Memory

- Address into MAR; issue Read operation; data into MDR.

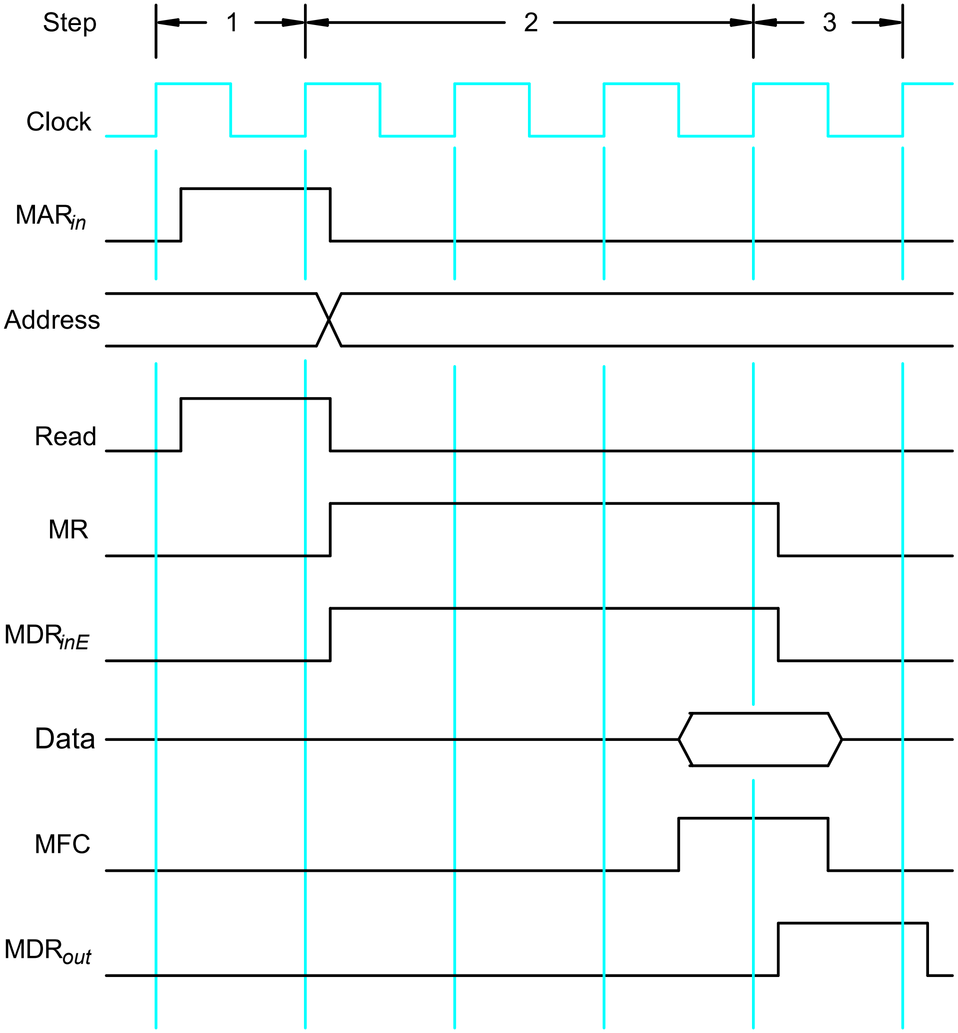


Fetching a Word from Memory *(Contd.)*

- 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]

Timing

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



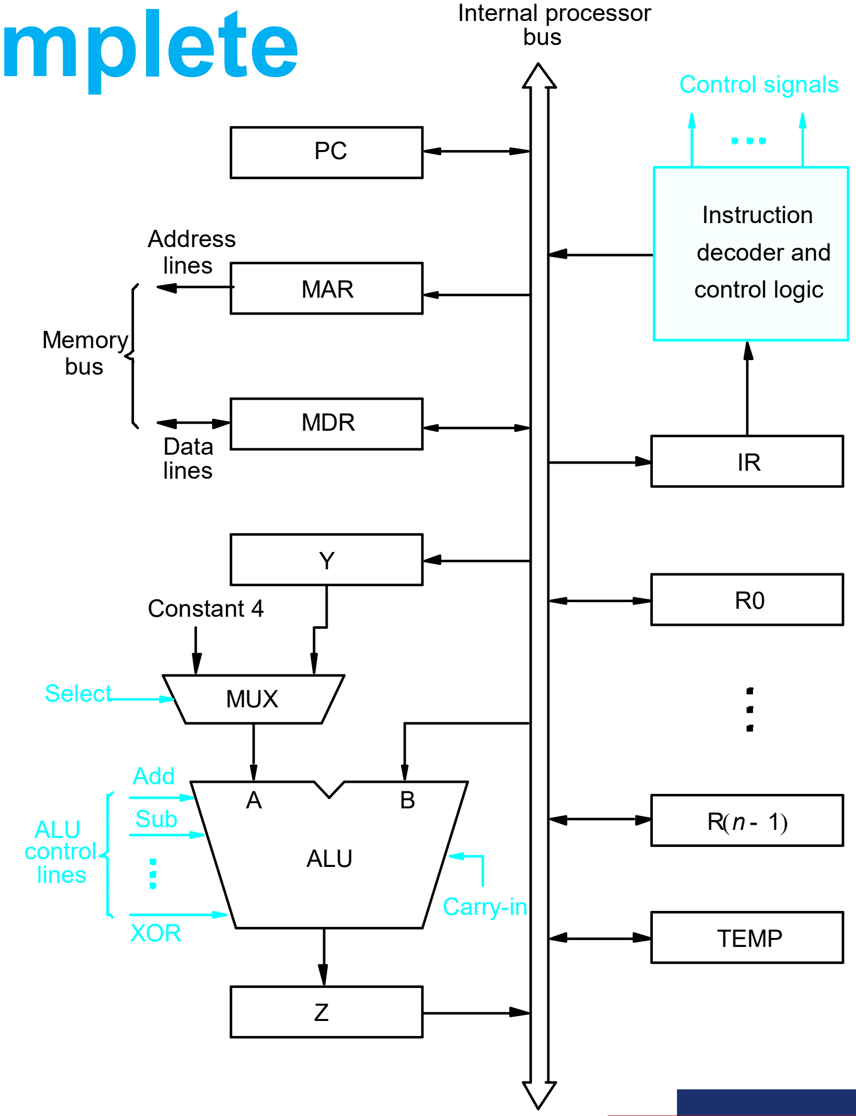
Execution of a Complete Instruction

- 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

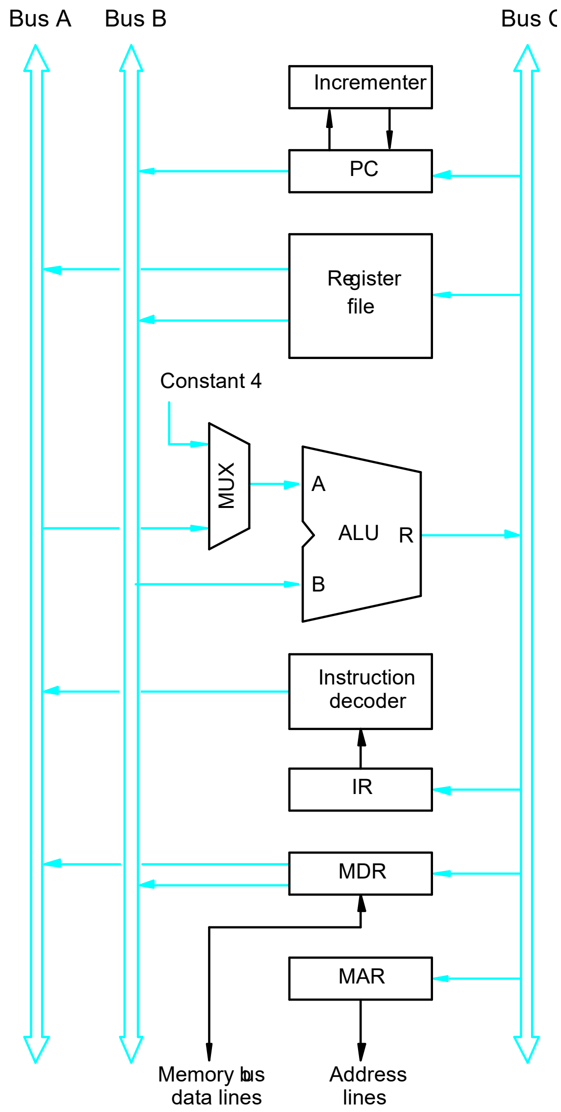
Execution of a Complete Instruction

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



Multiple-Bus Organization



Execution of a Branch Instruction

- 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 a Branch Instruction

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

Pipelining Overview

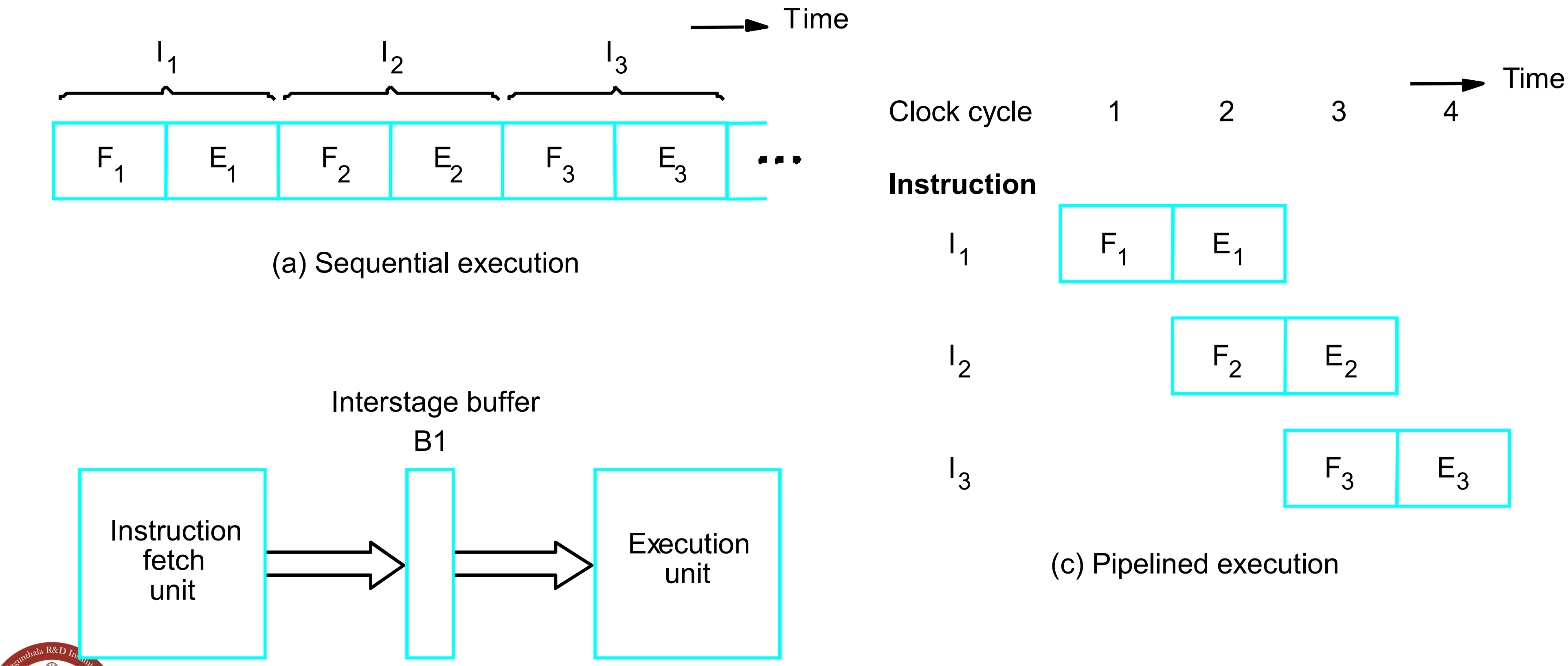
- Pipelining (Overlapping the execution of successive instructions) is widely used in modern processors.
- Pipelining improves system performance in terms of throughput.
- Pipelined organization requires sophisticated compilation techniques.

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.

Use the Idea of Pipelining in a Computer

Fetch + Execution



Clock cycle	1	2	3	4
Instruction				
I_1	F_1	E_1		
I_2		F_2	E_2	
I_3			F_3	E_3

(a) Sequential execution

(b) Hardware organization

(c) Pipelined execution

Figure. Basic idea of instruction pipelining.

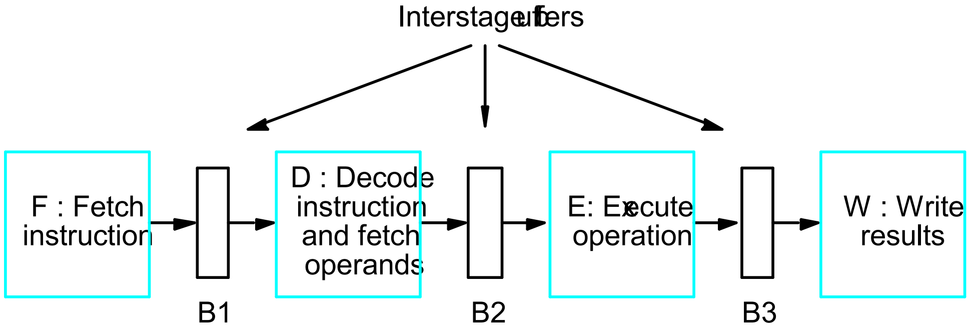
Use the Idea of Pipelining in a Computer

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



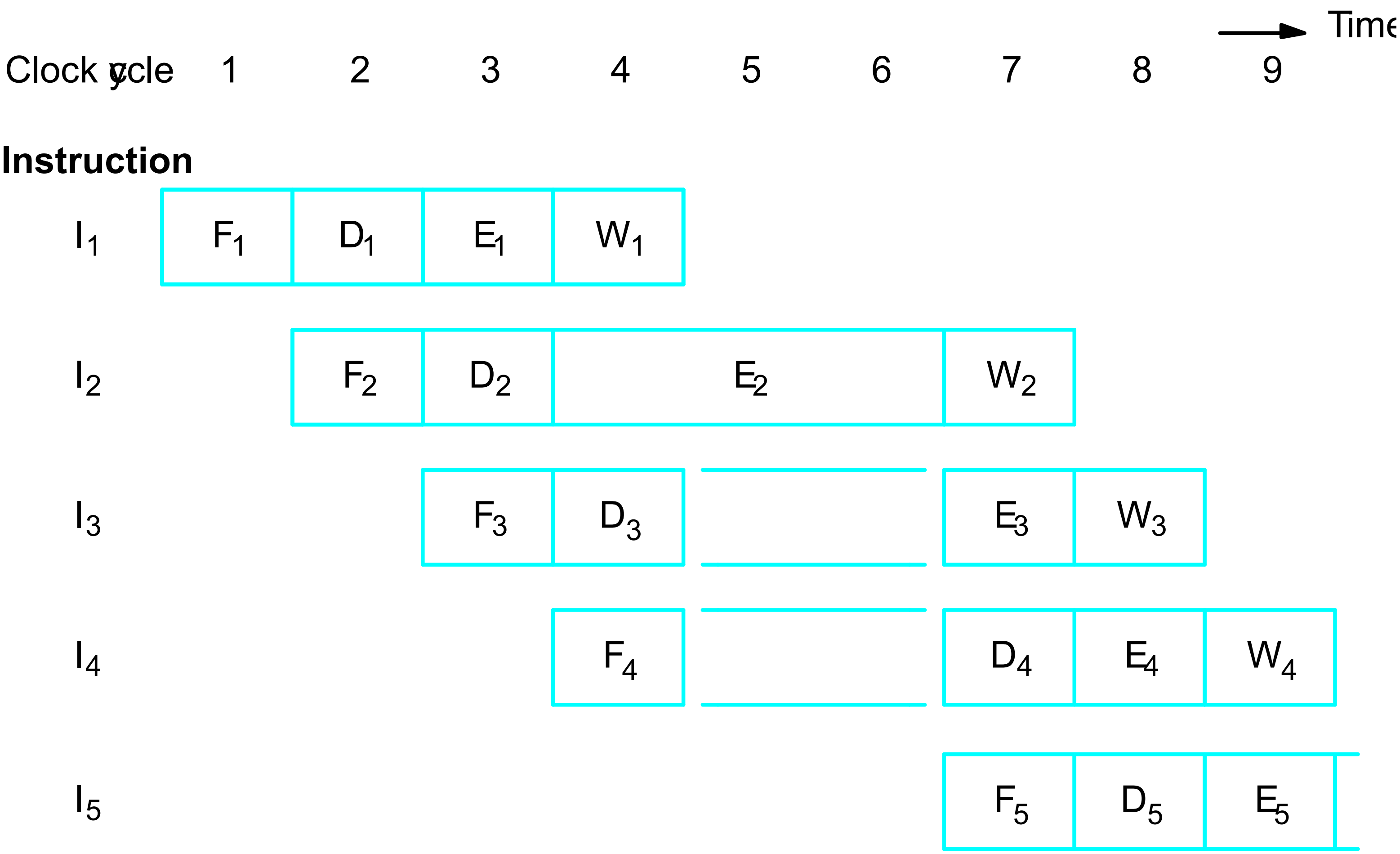
Role of Cache Memory

- 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



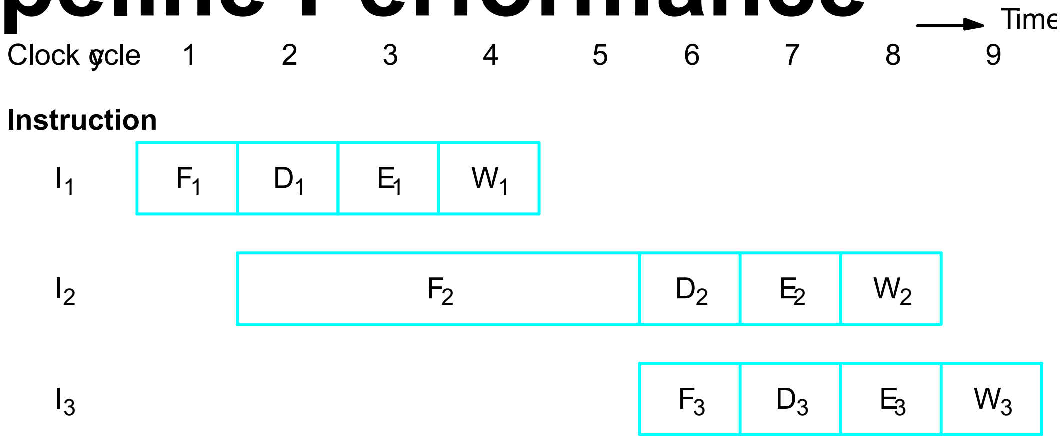
<table>
<tr><th>Clock cycle</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th></tr>
<tr><td><b>Instruction</b></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>I_1</td><td>F_1</td><td>D_1</td><td>E_1</td><td>W_1</td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>I_2</td><td></td><td>F_2</td><td>D_2</td><td colspan="3">E_2</td><td>W_2</td><td></td><td></td></tr>
<tr><td>I_3</td><td></td><td></td><td>F_3</td><td>D_3</td><td></td><td></td><td>E_3</td><td>W_3</td><td></td></tr>
<tr><td>I_4</td><td></td><td></td><td></td><td>F_4</td><td></td><td></td><td>D_4</td><td>E_4</td><td>W_4</td></tr>
<tr><td>I_5</td><td></td><td></td><td></td><td></td><td></td><td></td><td>F_5</td><td>D_5</td><td>E_5</td></tr>
</table>

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.

Pipeline Performance

Instruction hazard



→ 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	D_3	E_3	W_3

(a) Instruction execution steps in successive clock cycles

→ Time

Clock cycle	1	2	3	4	5	6	7	8	9
Stage									
F: Fetch	F_1	F_2	F_2	F_2	F_2	F_3			
D: Decode		D_1	idle	idle	idle	D_2	D_3		
E: Execute			E_1	idle	idle	idle	E_2	E_3	
W: Write				W_1	idle	idle	idle	W_2	W_3

(b) Function performed by each processor stage in successive clock cycles

Idle periods – stalls (bubbles)

Pipeline Performance

Load X(R1), R2

→ Time

Structural hazard

Clock cycle	1	2	3	4	5	6	7
Instruction							
I_1	F_1	D_1	E_1	W_1			
I_2 (Load)		F_2	D_2	E_2	M_2	W_2	
I_3			F_3	D_3	E_3		W_3
I_4				F_4	D_4		E_4
I_5						F_5	D_5

Pipeline Performance

- Again, pipelining does not result in individual instructions being executed faster; rather, it is the throughput that increases.
- Throughput is measured by the rate at which instruction execution is completed.
- Pipeline stall causes degradation in pipeline performance.
- We need to identify all hazards that may cause the pipeline to stall and to find ways to minimize their impact.

Data Hazards

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 ← 3 + A

  B ← 4 × A
- No hazard

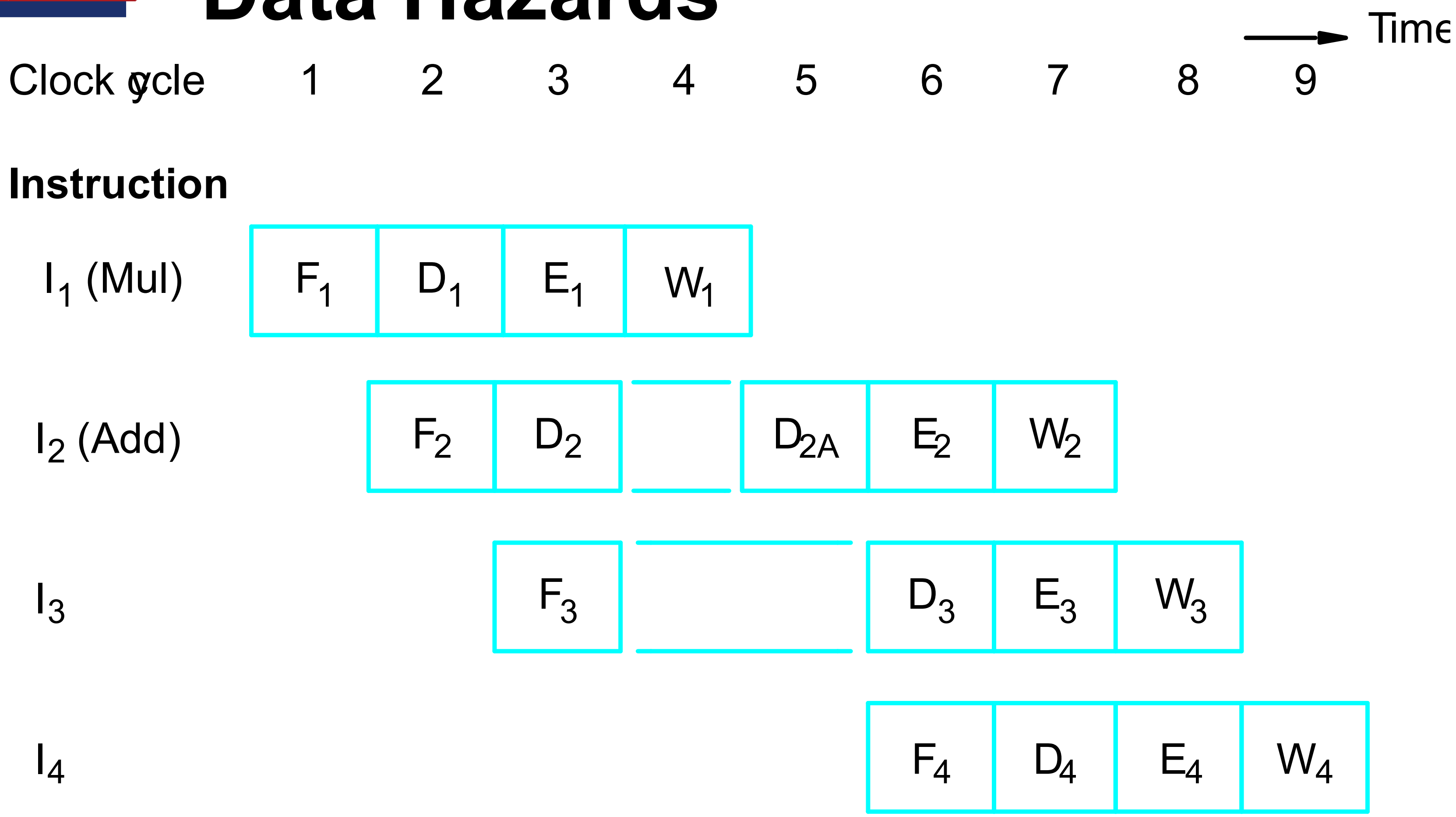
  A ← 5 × C

  B ← 20 + C
- When two operations depend on each other, they must be executed sequentially in the correct order.
- Another example:

  Mul R2, R3, R4

  Add R5, R4, R6

Data Hazards

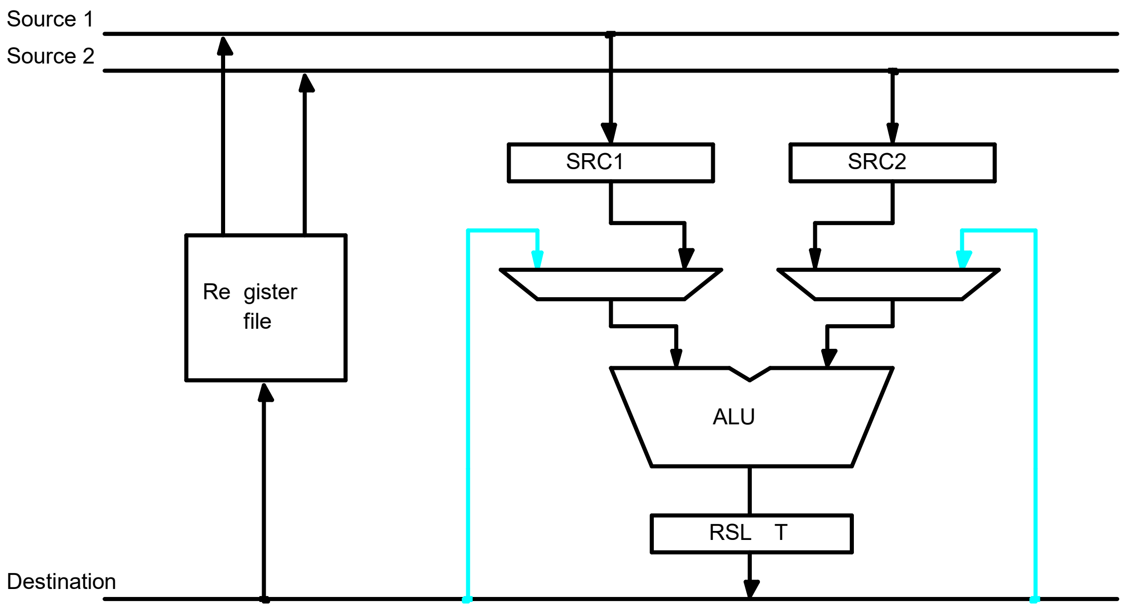


Clock cycle	1	2	3	4	5	6	7	8	9
Instruction									
I_1 (Mul)	F_1	D_1	E_1	W_1					
I_2 (Add)		F_2	D_2		D_{2A}	E_2	W_2		
I_3			F_3			D_3	E_3	W_3	
I_4						F_4	D_4	E_4	W_4

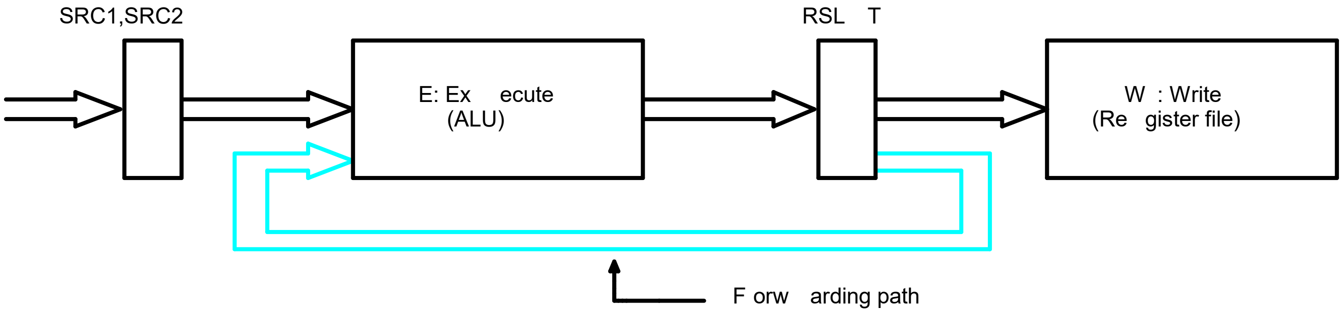
Figure. Pipeline stalled by data dependency between D_2 and W_1.

Operand Forwarding

- Instead of from the register file, the second instruction can get data directly from the output of ALU after the previous instruction is completed.
- A special arrangement needs to be made to “forward” the output of ALU to the input of ALU.



(a) Datapath



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

Handling Data Hazards in Software

- Let the compiler detect and handle the hazard:

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

- The compiler can reorder the instructions to perform some useful work during the NOP slots.

Side Effects

- The previous example is explicit and easily detected.
- Sometimes an instruction changes the contents of a register other than the one named as the destination.
- 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. (Example?)
- Example: conditional code flags:

```
Add  R1, R3
AddWithCarry  R2, R4
```

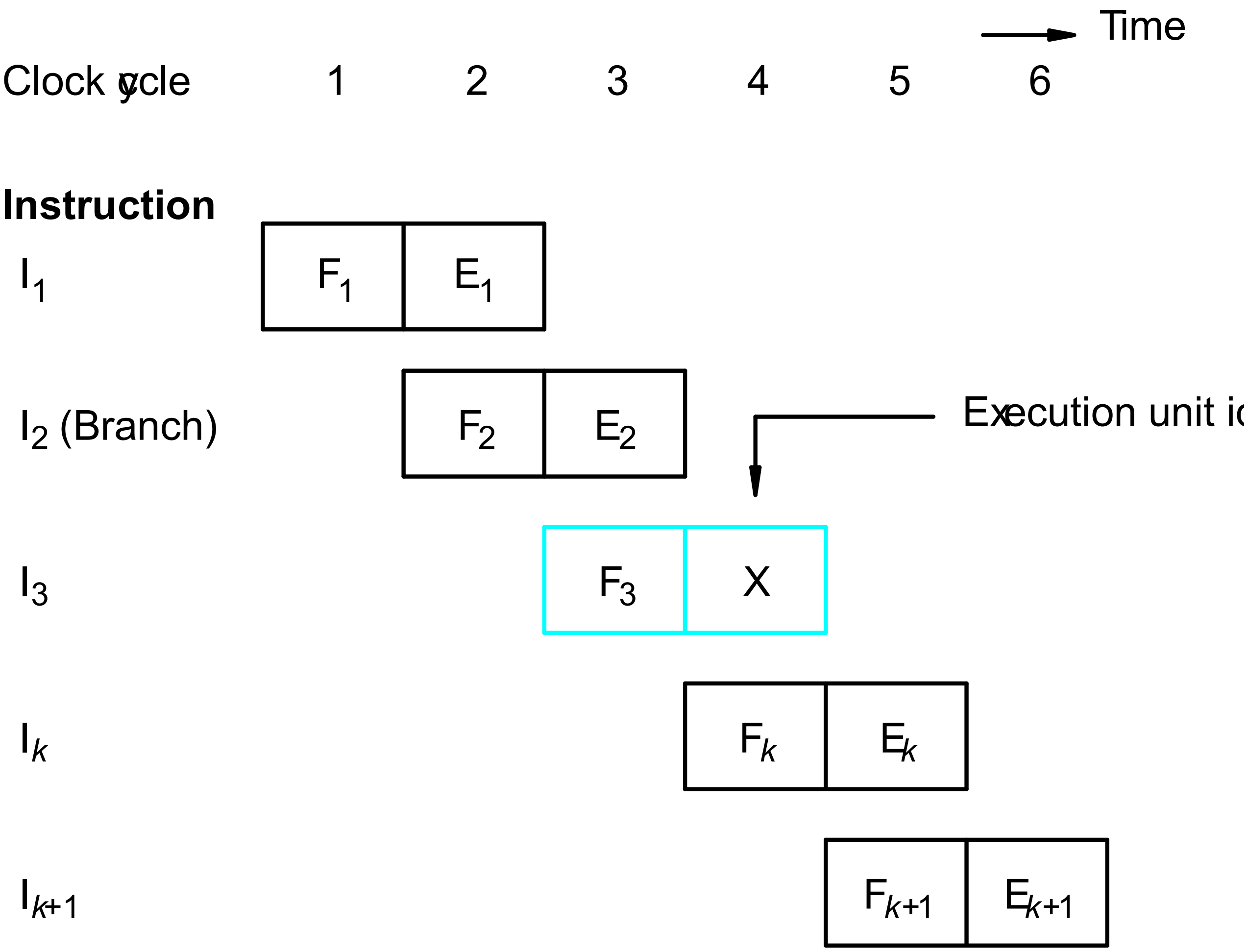
- Instructions designed for execution on pipelined hardware should have few side effects.

Instruction Hazards

Overview

- Whenever the stream of instructions supplied by the instruction fetch unit is interrupted, the pipeline stalls.
- Cache miss
- Branch

Unconditional Branches



Time

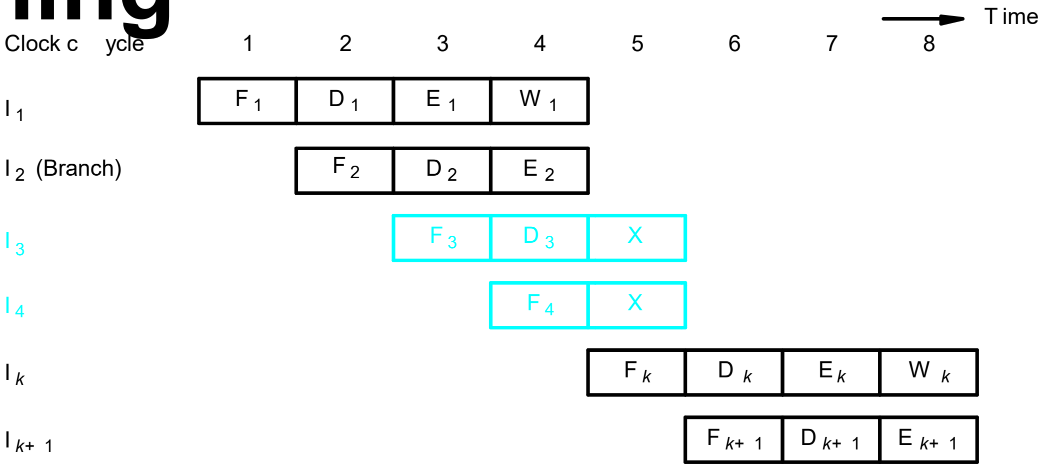
Execution unit i

Clock cycle	1	2	3	4	5	6
Instruction						
I_1	F_1	E_1				
I_2 (Branch)		F_2	E_2			
I_3			F_3	X		
I_k				F_k	E_k	
I_{k+1}					F_{k+1}	E_{k+1}

Branches Timing

- Branch penalty

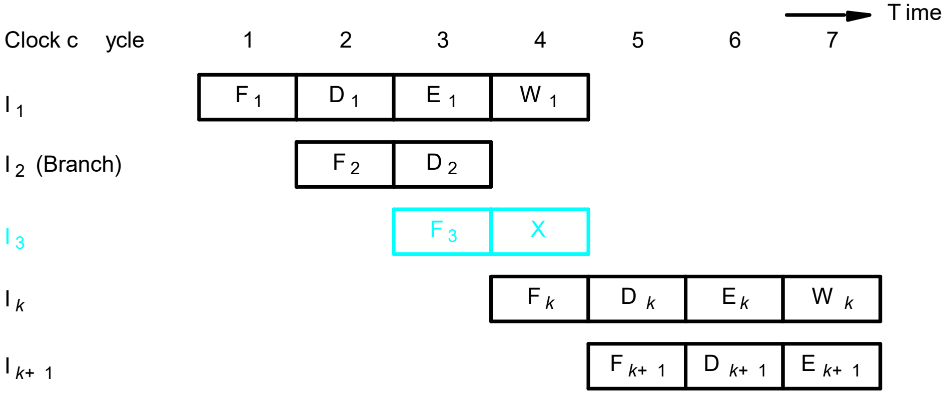
- Reducing the penalty



Time

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

(a) Branch address computed in Execute stage



Time

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

(b) Branch address computed in Decode stage

Instruction Queue and Prefetching

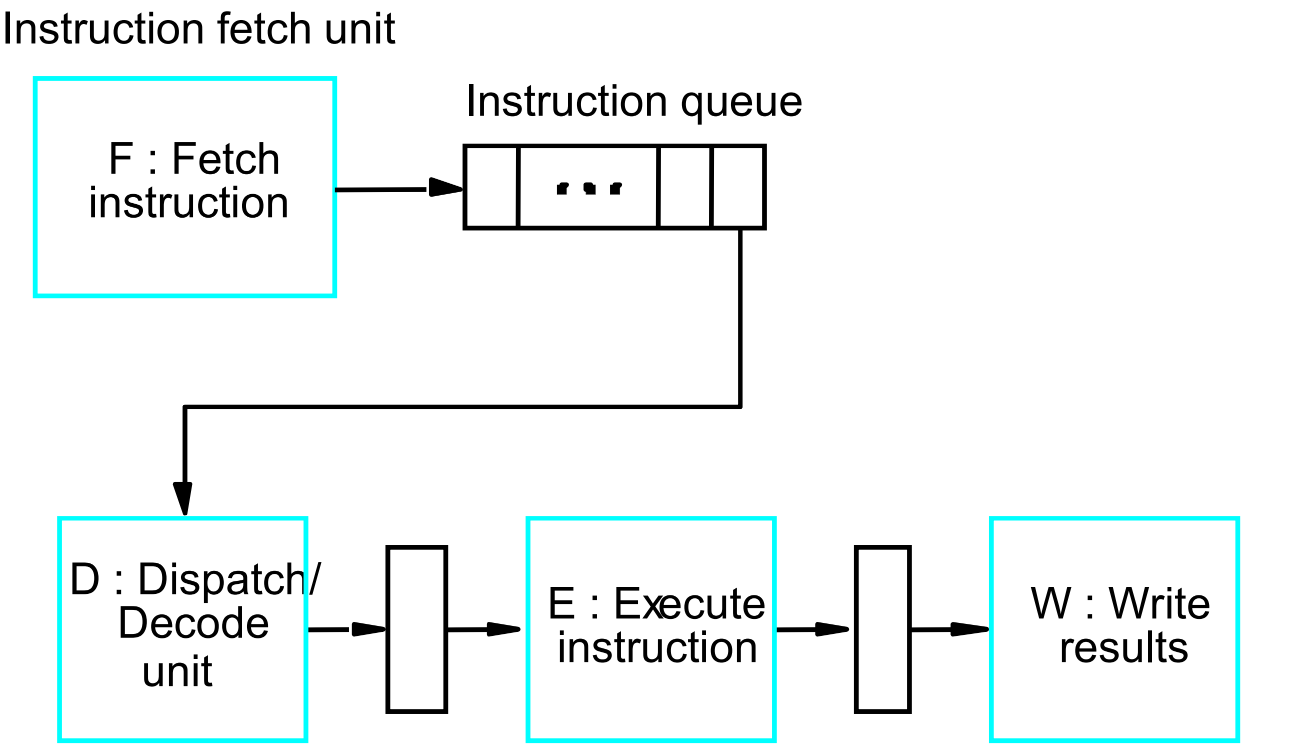
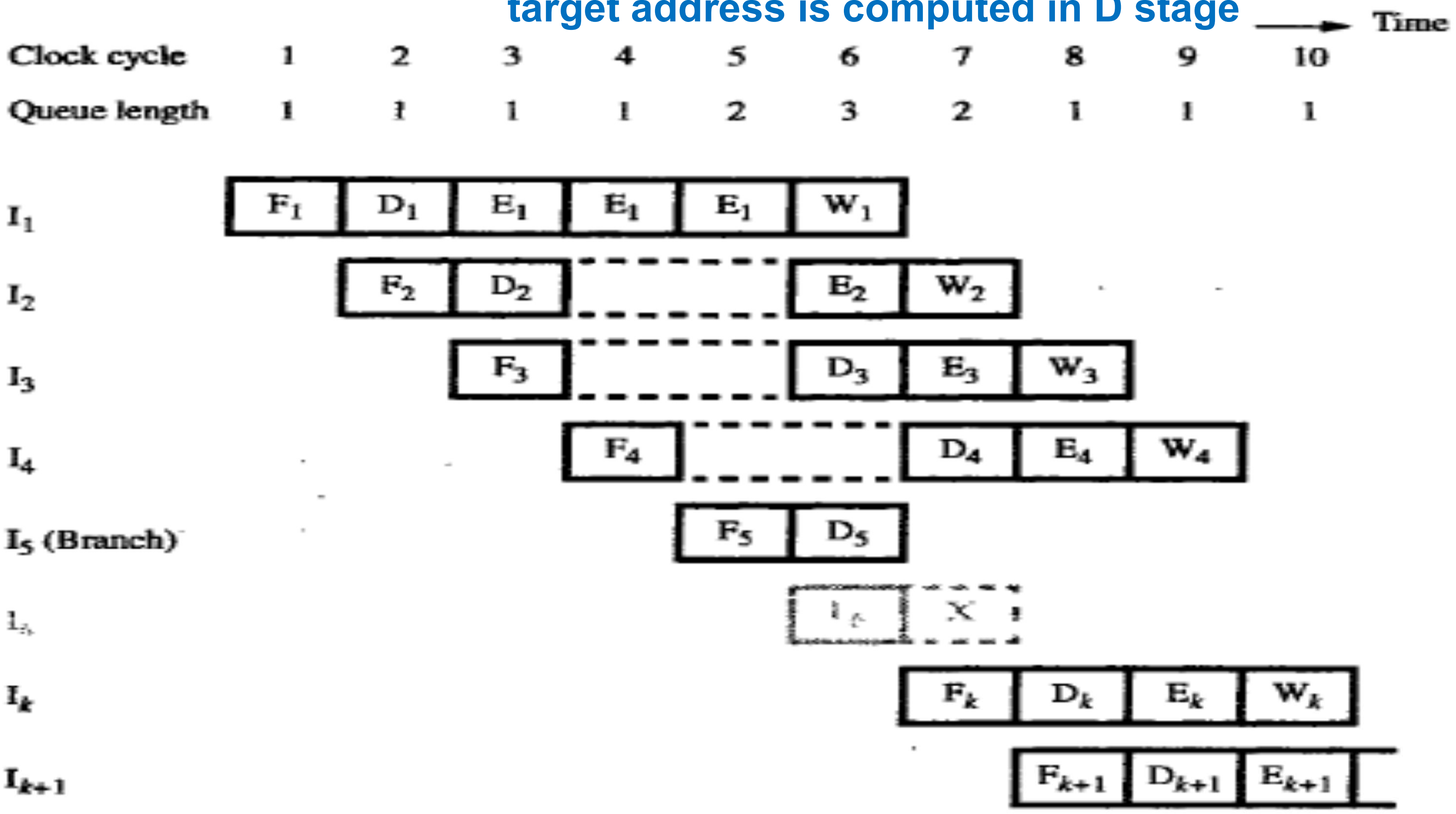


Figure. Use of an instruction queue in the hardware organization

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



Clock cycle	1	2	3	4	5	6	7	8	9	10
Queue length	1	1	1	1	2	3	2	1	1	1
I_1	F_1	D_1	E_1	E_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	
I_5 (Branch)					F_5	D_5				
I_6						F_6	X			
I_k							F_k	D_k	E_k	W_k
I_{k+1}								F_{k+1}	D_{k+1}	E_{k+1}

Conditional Braches

- 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 represent about 20% of the dynamic instruction count of most programs.

Delayed Braches

- 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 place useful instructions in these slots.
- The effectiveness of the delayed branch approach depends on how often it is possible to reorder instructions.

Delayed Braches

```
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. Reordering of instructions for a delayed branch.

Delayed Braches

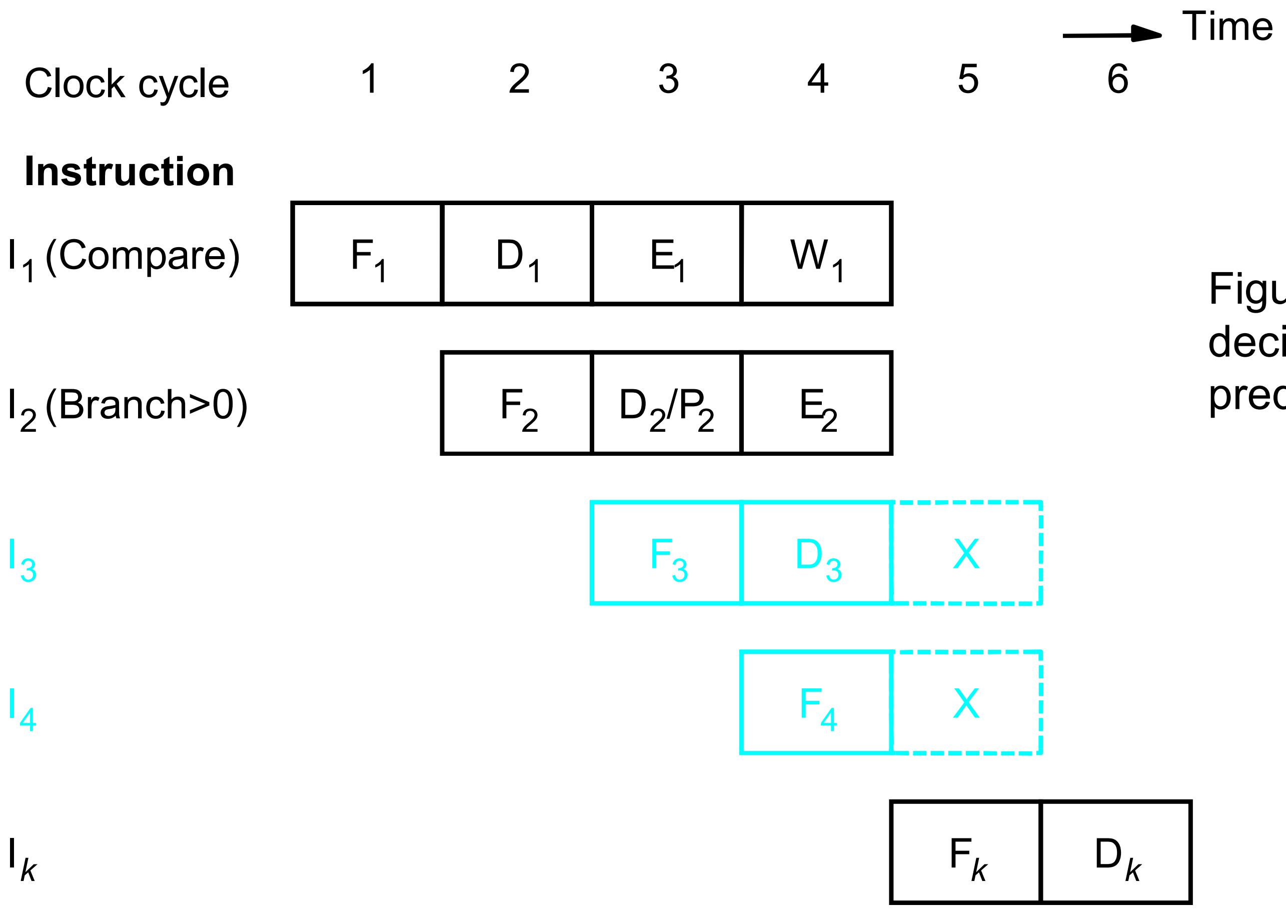
→ Time

Clock cycle	1	2	3	4	5	6	7	8
Instruction								
Decrement	F	E						
Branch		F	E					
Shift (delay slot)			F	E				
Decrement (Branch taken)				F	E			
Branch					F	E		
Shift (delay slot)						F	E	
Add (Branch not taken)							F	E

Braches Prediction

- To predict whether or not a particular branch will be taken.
- Simplest form: assume branch will not take place and continue to fetch instructions in sequential address order.
- Until the branch is evaluated, instruction execution along the predicted path must be done on a speculative basis.
- **Speculative execution:** instructions are executed before the processor is certain that they are in the correct execution sequence.
- Need to be careful so that no processor registers or memory locations are updated until it is confirmed that these instructions should indeed be executed.

Incorrectly Predicted Branch



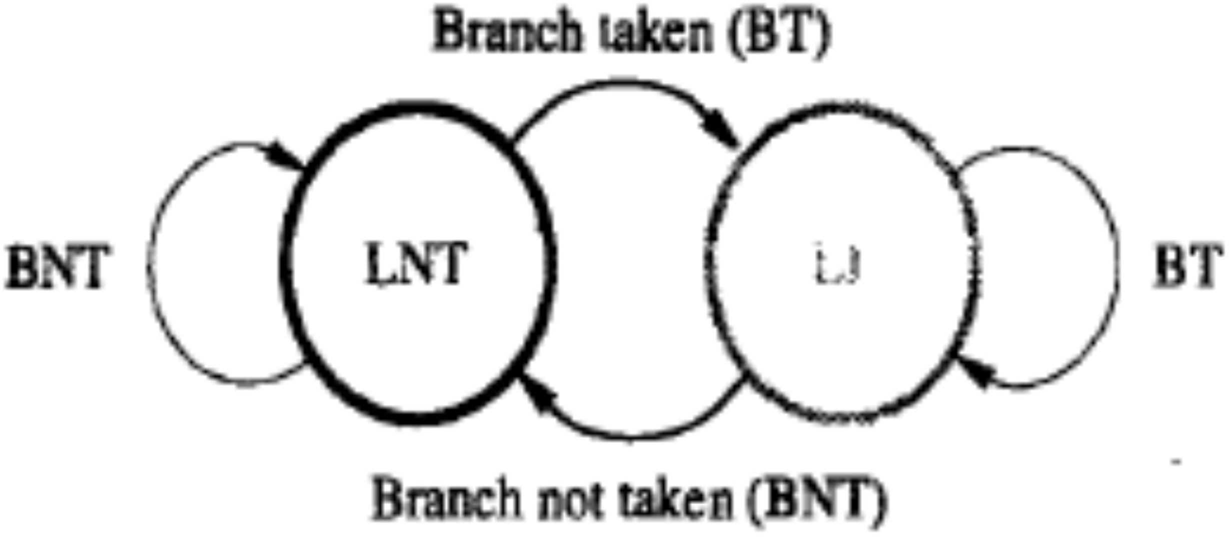
Clock cycle	1	2	3	4	5	6
Instruction						
I_1 (Compare)	F_1	D_1	E_1	W_1		
I_2 (Branch>0)		F_2	D_2/P_2	E_2		
I_3			F_3	D_3	X	
I_4				F_4	X	
I_k					F_k	D_k

Figure. Timing when a branch decision has been incorrectly predicted as not taken.

Dynamic Branch Prediction

LT: Branch is likely to be taken

LNT: Branch is likely not to be taken



(a) A 2-state algorithm

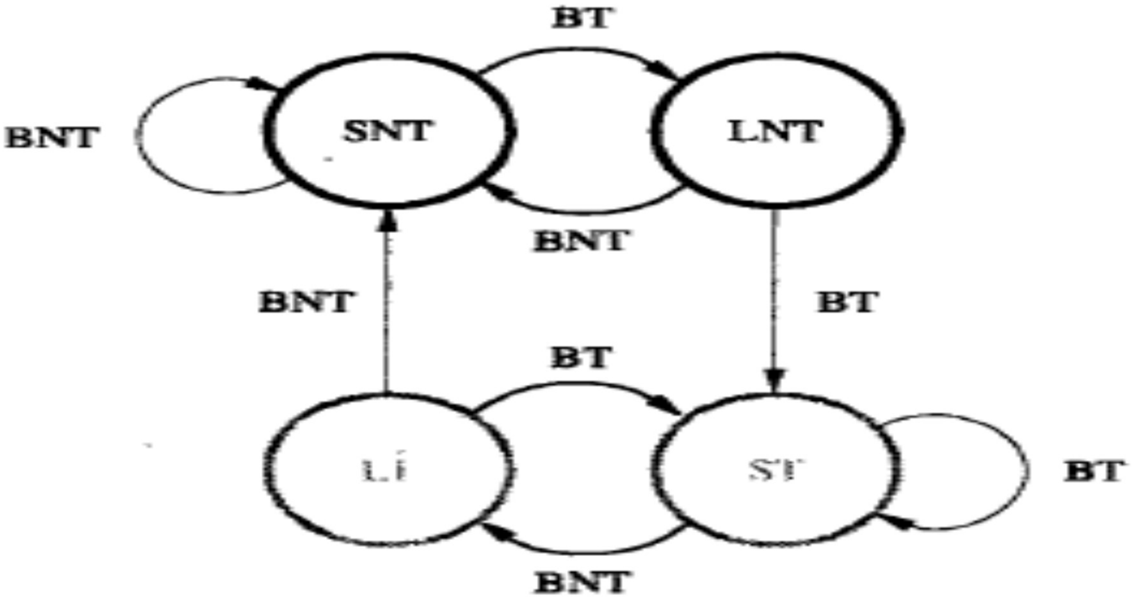
Dynamic Branch Prediction *(Contd.,)*

ST: Strongly likely to be taken

LT: Likely to be taken

LNT: Likely not to be taken

SNT: Strongly likely not to be taken



(b) A 4-state algorithm

Branch Prediction

- Better performance can be achieved if we arrange for some branch instructions to be predicted as taken and others as not taken.
- Use hardware to observe whether the target address is lower or higher than that of the branch instruction.
- Let compiler include a branch prediction bit.
- So far the branch prediction decision is always the same every time a given instruction is executed – static branch prediction.

Influence on Instruction Sets

Overview

- Some instructions are much better suited to pipeline execution than others.
- Addressing modes
- Conditional code flags

Addressing Modes

- Addressing modes include simple ones and complex ones.
- In choosing the addressing modes to be implemented in a pipelined processor, we must consider the effect of each addressing mode on instruction flow in the pipeline:
  - ➢ Side effects
  - ➢ The extent to which complex addressing modes cause the pipeline to stall
  - ➢ Whether a given mode is likely to be used by compilers

Recall

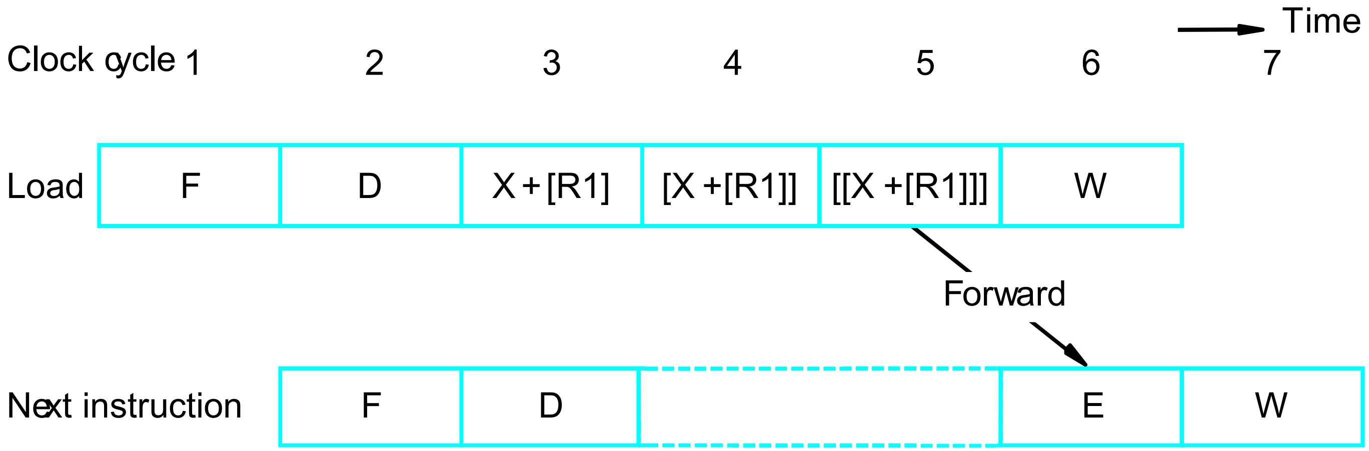
Load X(R1), R2

→ Time

Clock cycle	1	2	3	4	5	6	7
Instruction							
I_1	F_1	D_1	E_1	W_1			
I_2 (Load)		F_2	D_2	E_2	M_2	W_2	
I_3			F_3	D_3	E_3		W_3
I_4				F_4	D_4		E_4
I_5						F_5	D_5

Complex Addressing Mode

Load (X(R1)), R2



Clock cycle	1	2	3	4	5	6	7
Load	F	D	X + [R1]	[X +[R1]]	[[X +[R1]]]	W	
Next instruction		F	D			E	W

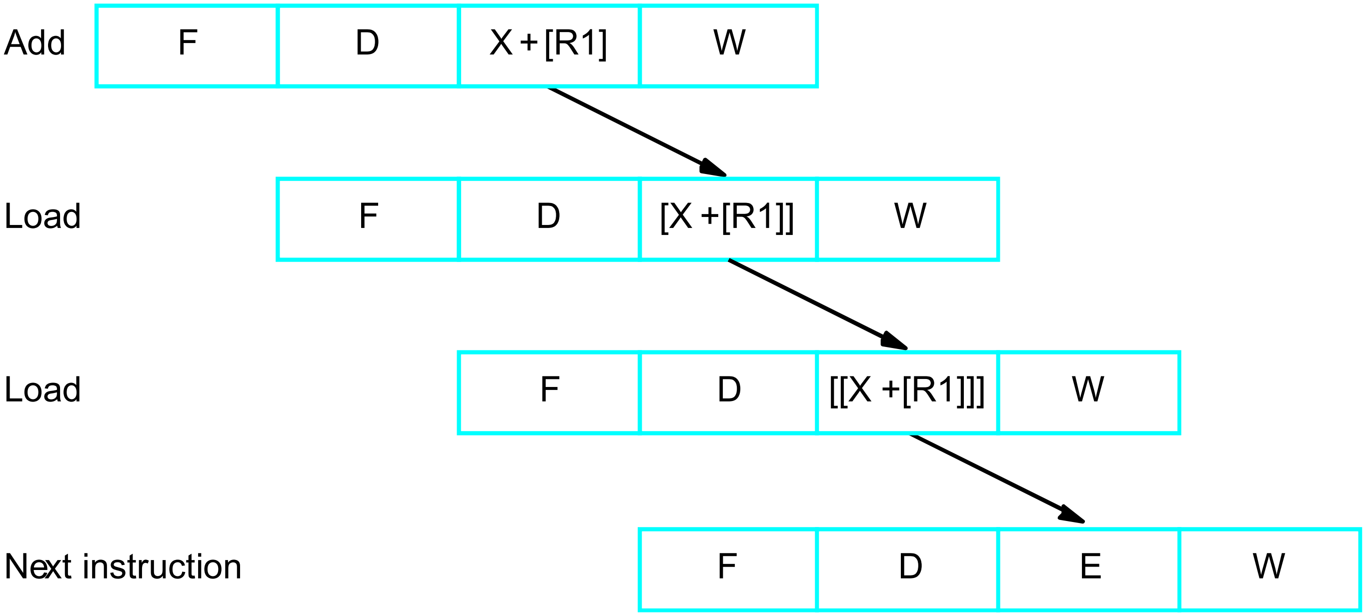
Time

Forward

(a) Complex addressing mode

Simple Addressing Mode

Add #X, R1, R2
Load (R2), R2
Load (R2), R2



Add	F	D	X + [R1]	W			
Load		F	D	[X +[R1]]	W		
Load			F	D	[[X +[R1]]]	W	
Next instruction				F	D	E	W

(b) Simple addressing mode

Addressing Modes

- In a pipelined processor, complex addressing modes do not necessarily lead to faster execution.
- **Advantage:** reducing the number of instructions / program space
- **Disadvantage:** cause pipeline to stall / more hardware to decode / not convenient for compiler to work with
- **Conclusion:** complex addressing modes are not suitable for pipelined execution.
- Good addressing modes should have:
  - ➢ Access to an operand does not require more than one access to the memory
  - ➢ Only load and store instruction access memory operands
  - ➢ The addressing modes used do not have side effects
- Register, register indirect, index

Conditional Codes

- If an optimizing compiler attempts to reorder instruction to avoid stalling the pipeline when branches or data dependencies between successive instructions occur, it must ensure that reordering does not cause a change in the outcome of a computation.
- The dependency introduced by the condition-code flags reduces the flexibility available for the compiler to reorder instructions.

Conditional Codes

```
Add              R1,R2
Compare          R3,R4
Branch=0         . . .
```

(a) A program fragment

```
Compare          R3,R4
Add              R1,R2
Branch=0         . . .
```

(b) Instructions reordered

Figure. Instruction reordering.

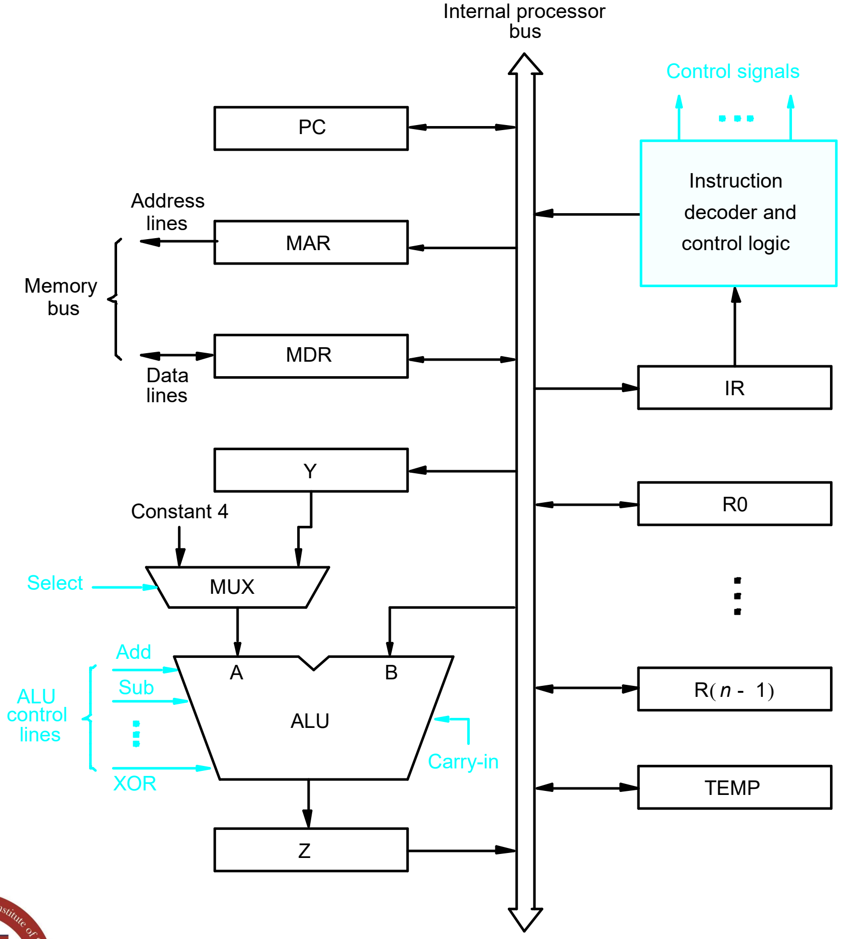
Conditional Codes

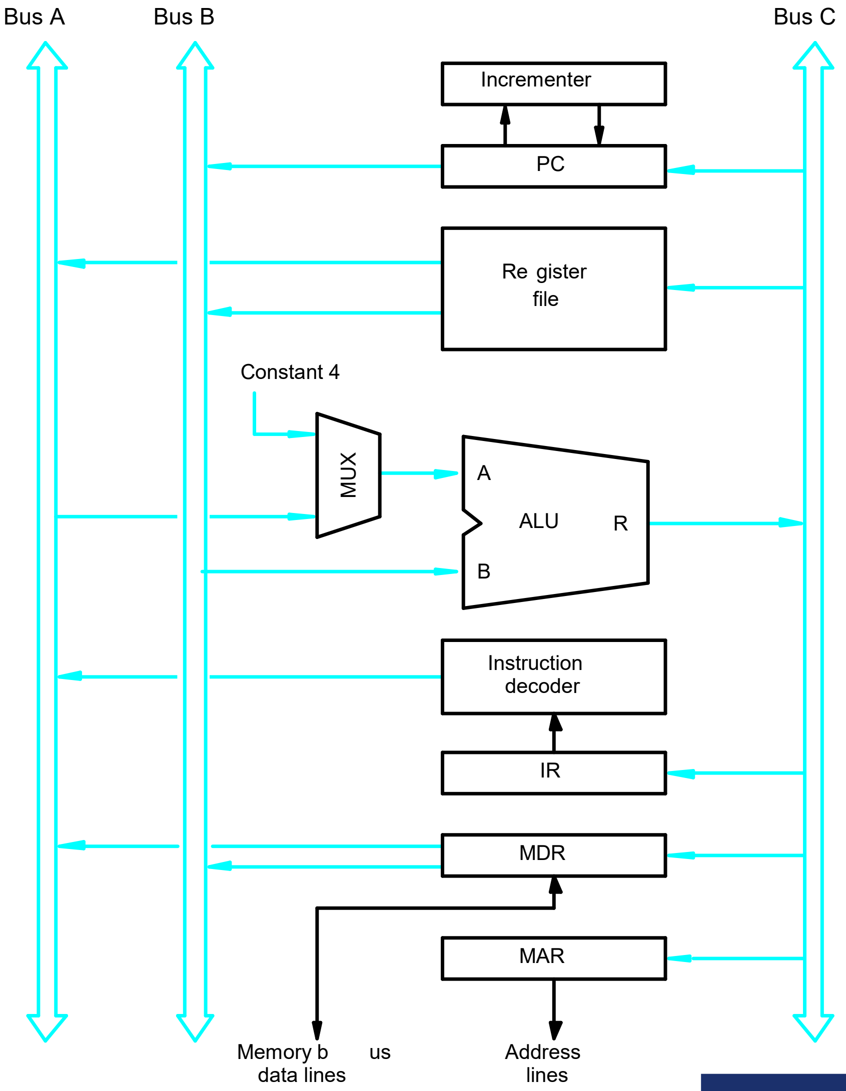
Two conclusion:

- To provide flexibility in reordering instructions, the condition-code flags should be affected by as few instruction as possible.
- The compiler should be able to specify in which instructions of a program the condition codes are affected and in which they are not.

Datapath and Control Considerations

Original Design

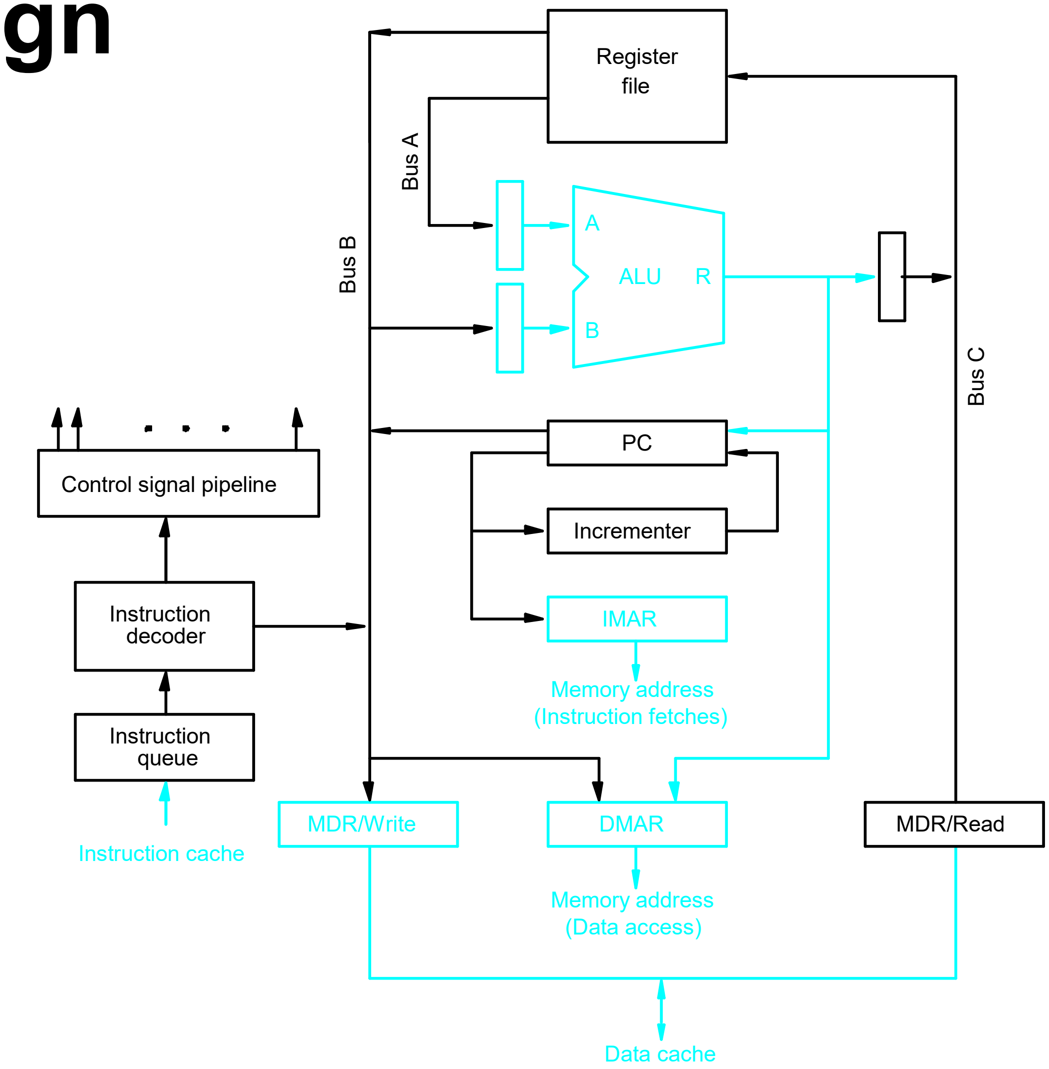




Pipelined Design

- Separate instruction and data caches
- PC is connected to IMAR
- DMAR
- Separate MDR
- Buffers for ALU
- Instruction queue
- Instruction decoder output

- Reading an instruction from the instruction cache
- Incrementing the PC
- Decoding an instruction
- Reading from or writing into the data cache
- Reading the contents of up to two regs
- Writing into one register in the reg file
- Performing an ALU operation



Exception Handling

Exception

- “Unexpected” events requiring change in flow of control
  - Different ISAs use the terms differently
- Exceptions
  - Arises within the CPU
  - E.g undefined opcode, overflow, system call,…

Interrupts

- From an external I/O controller or device
- Dealing with them without sacrificing performance is hard

Types of Exception

- I/O device request
- Invoking an OS service from a user program
- Tracing instruction execution
- Breakpoint
- Integer arithmetic overflow
- FP arithmetic anomaly
- Page fault
- Misaligned memory access
- Memory protection violation
- Using an undefined or unimplemented instruction
- Hardware malfunctions
- Power failure

Characteristics of Exceptions

- Synchronous vs Asynchronous
  - Occurs in same place with the same data and memory allocation – Synchronous
  - Devices external to the CPU and memory cause asynchronous exceptions
    - Can be handled after the current instruction and hence easier
- User requested vs Coerced
  - User requested – Eg. Breakpoints
    - Predictable
    - Can be handled after the current instruction
  - Coerced exception are by hardware not under the control of the user program
    - Harder to handle
- User maskable vs unmaskable
- Within vs between instructions
- Resume vs Terminate

Exceptions in a Pipeline

- Another form of control hazard
- Consider overflow on add in E stage
  - Add R1, R2
    - Prevent R2 from being written into
    - Complete previous instructions
    - Flush ADD and subsequent instructions
    - Handle the exception
- Similar to mispredicted branch
  - Use much of the same hardware

Handling Exceptions in a Pipeline

- Force a trap instruction into the pipeline on the next F (Instruction Fetch)
- Until the trap is taken, turn off all writes for the faulting instruction and for all instructions that follow in the pipeline
  - Done by placing zeros in the latches
  - Prevents any state changes till the exception is handled
- The exception-handling routine saves PC of the faulting instruction to return from the exception later
  - Multiple PCs needed with delayed branching