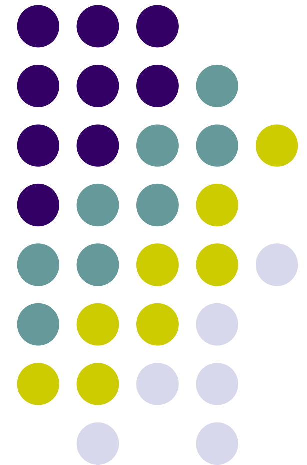
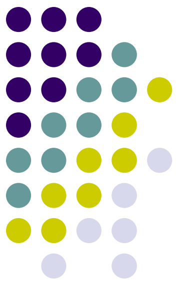
UNIT 3 - Basic Processing Unit

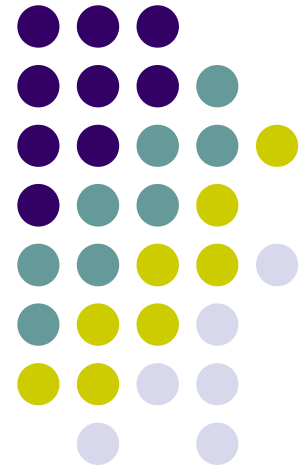


Overview

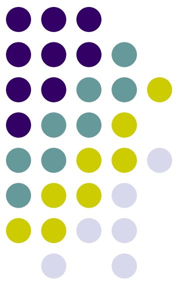


- Instruction Set Processor (ISP)
- Central Processing Unit (CPU)
- A typical computing task consists of a series of steps specified by a sequence of machine instructions that constitute a program.
- An instruction is executed by carrying out a sequence of more rudimentary operations.

Some Fundamental Concepts

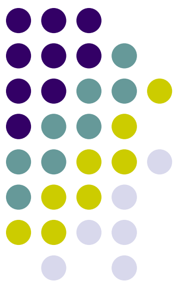


Fundamental Concepts



- 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).

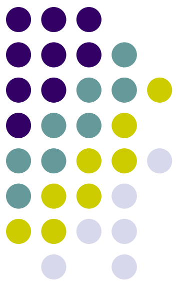
$$IR \leftarrow [[PC]]$$

- 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

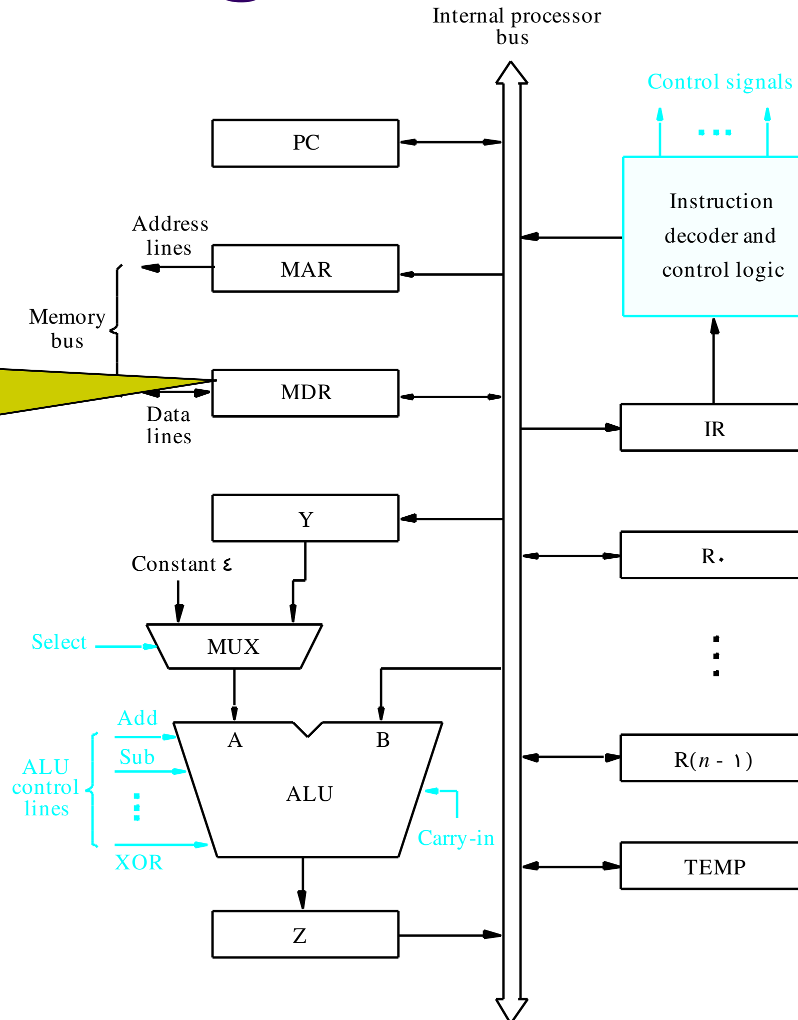
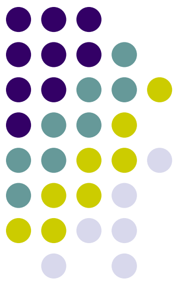


Figure 7.1. Single-bus organization of the datapath inside a processor.

Datapath

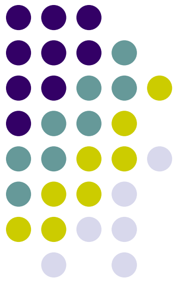
Textbook Page 413

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



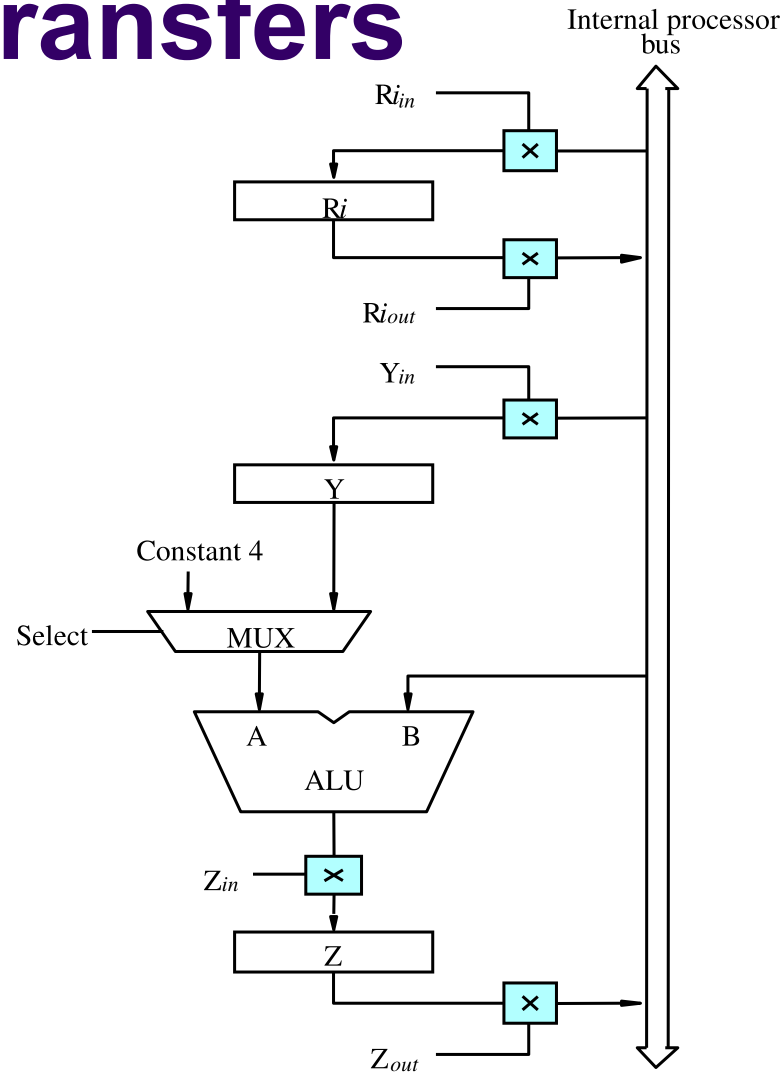
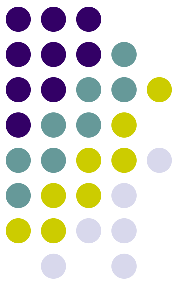


Figure 7.2. Input and output gating for the registers in Figure 7.1.



Register Transfers

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

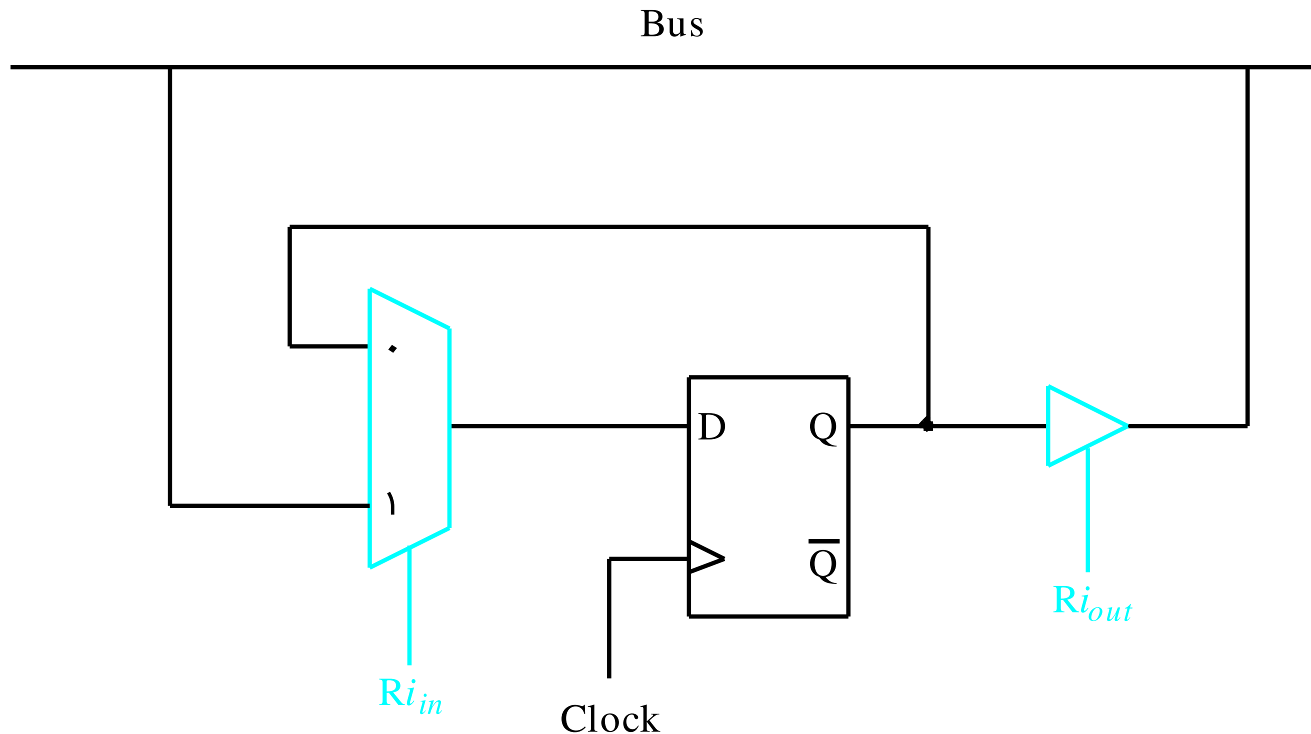
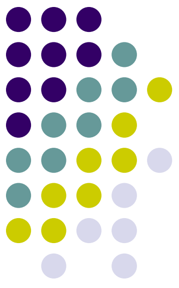
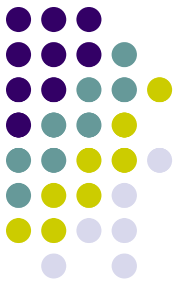


Figure 7.3. Input and output gating for one register bit.

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.

Memory-bus data lines

MDR_{outE}

MDR_{out}

Internal processor bus

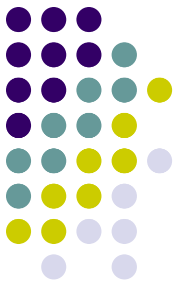
MDR

MDR_{inE}

MDR_{in}

Figure 7.4. Connection and control signals for register MDR.

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]

Timing

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

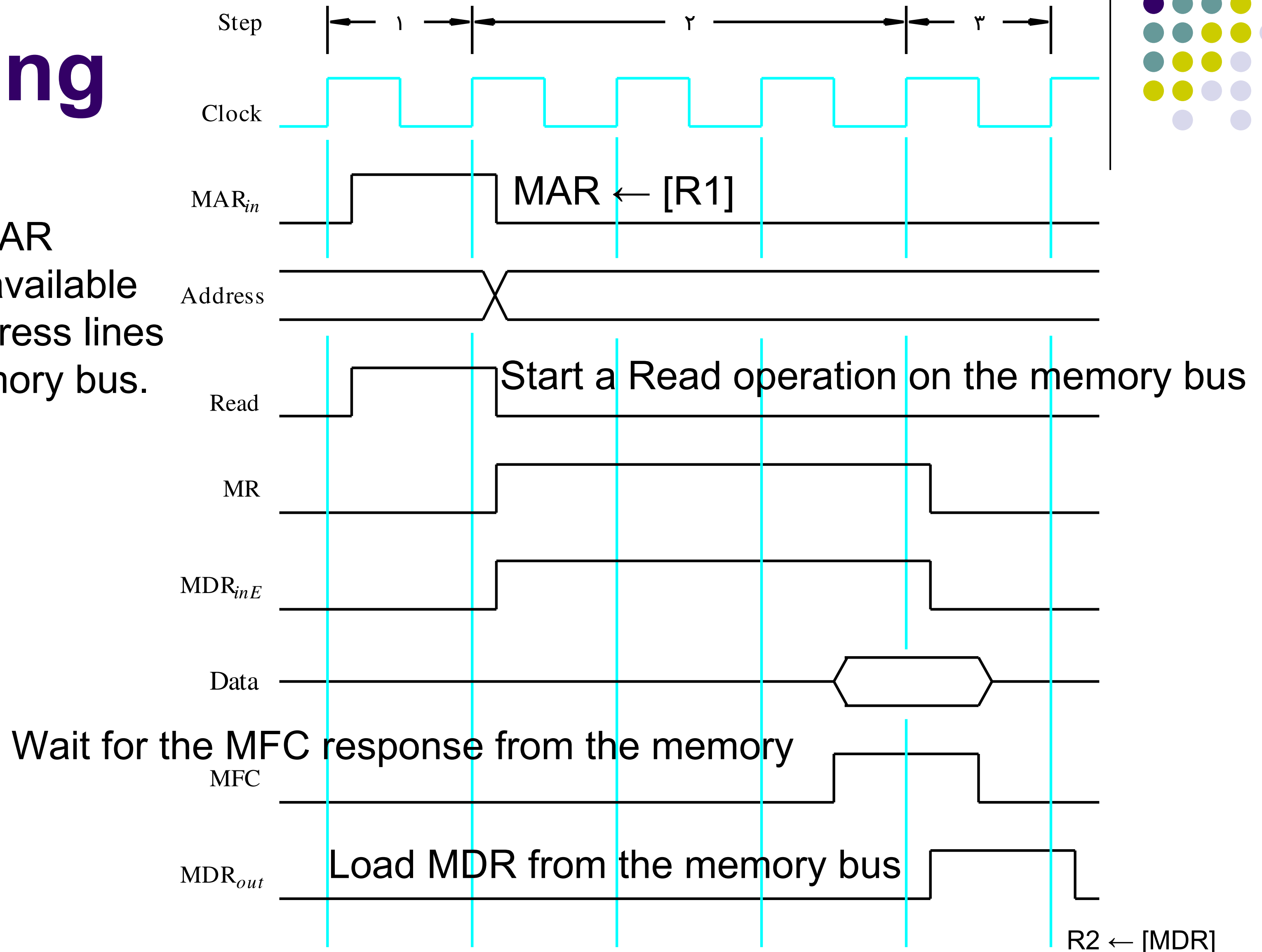
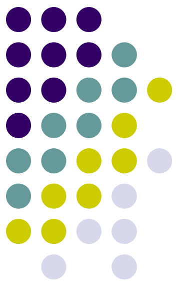


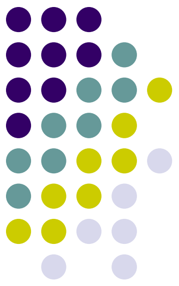
Figure ٧.٥. Timing of a memory Read operation.

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

Architecture



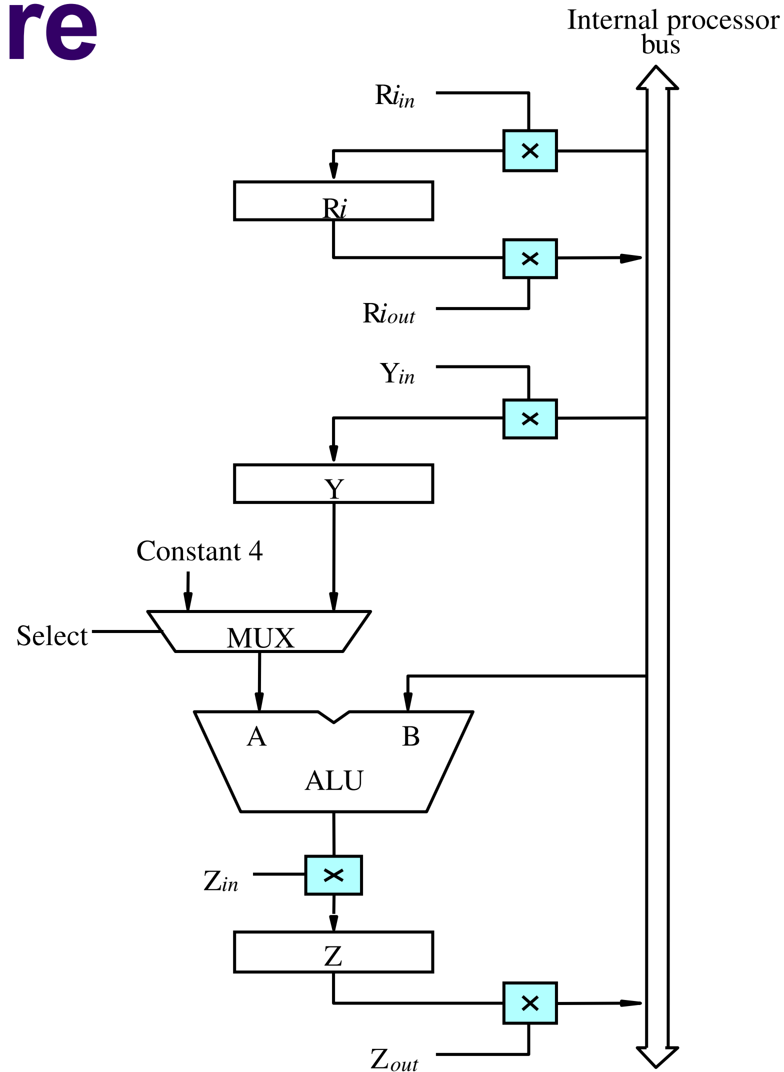


Figure 7.2. Input and output gating for the registers in Figure 7.1.

Execution of a Complete Instruction

Add (R3), R1

Step	Action
١	PC_{out} , MAR_{in} , Read, Select٤, Add, Z_{in}
٢	Z_{out} , PC_{in} , Y_{in} , WMF C
٣	MDR_{out} , IR_{in}
٤	R٣$_{out}$, MAR_{in} , Read
٥	R١$_{out}$, Y_{in} , WMF C
٦	MDR_{out} , SelectY, Add, Z_{in}
٧	Z_{out} , R١$_{in}$, End

Figure ٧.٦. Control sequence for execution of the instruction Add (R٣),R١.

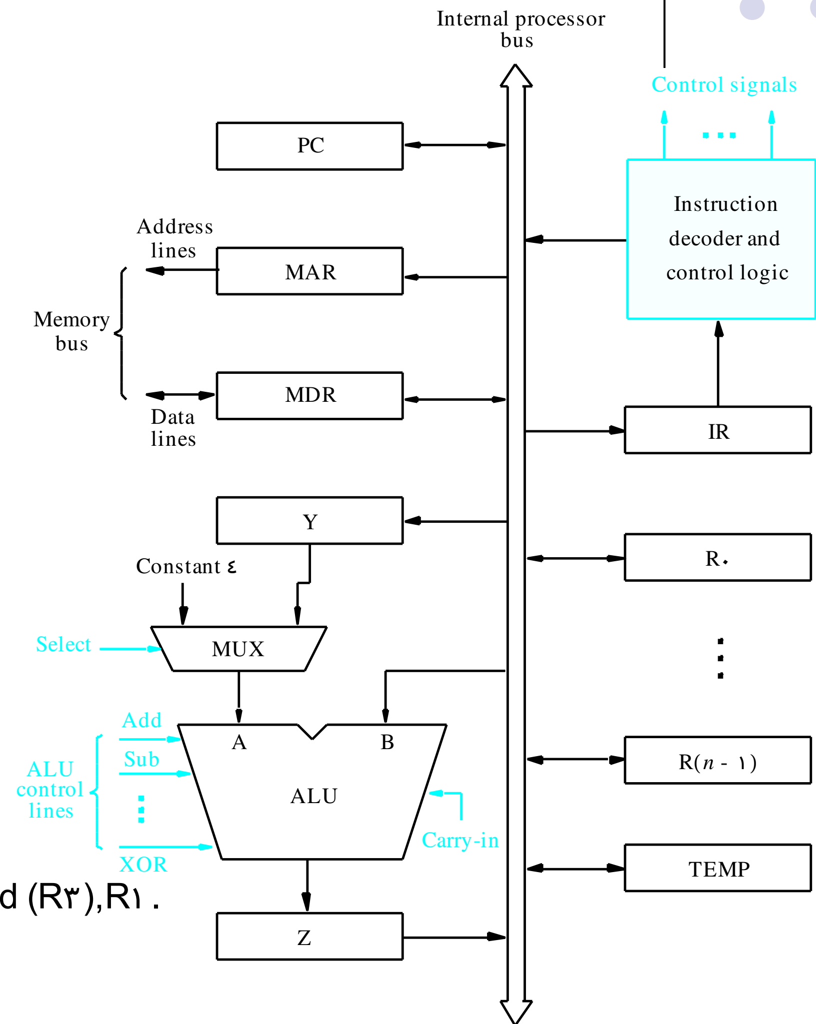
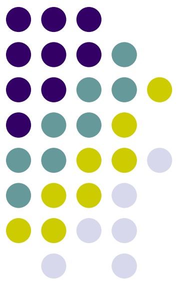


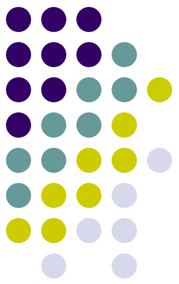
Figure ٧.١. Single-bus organization of the datapath inside a processor.

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

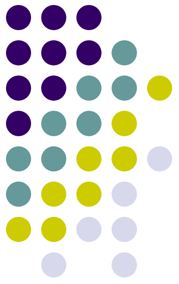
Execution of Branch Instructions



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



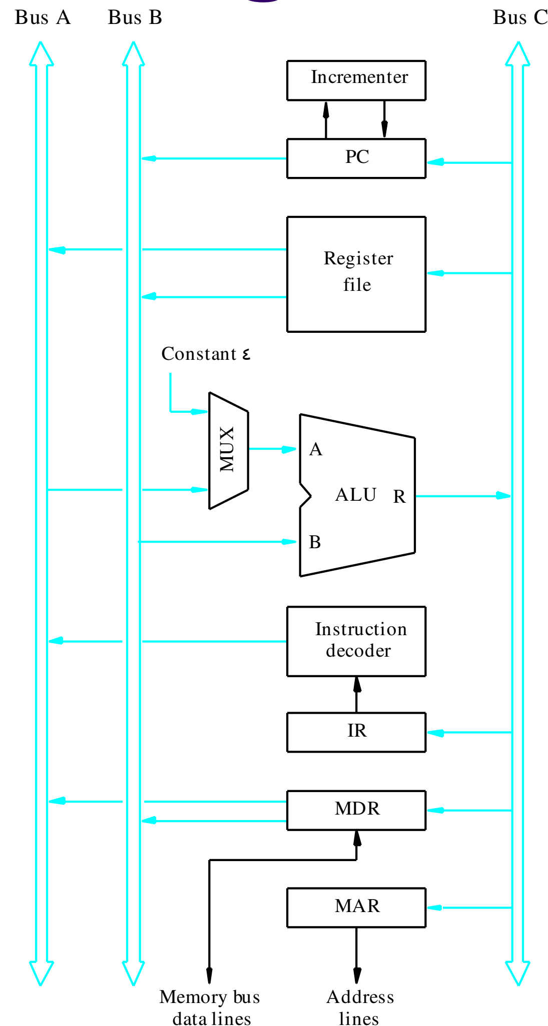
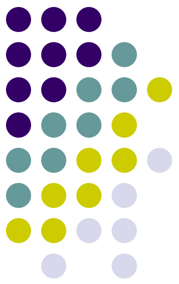


Figure 7.8. Three-bus organization of the datapath.

Multiple-Bus Organization



- Add R4, R5, R6

Step	Action
1	PC_{out}, R=B, MAR_{in}, Read, IncPC
2	WMFC
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)

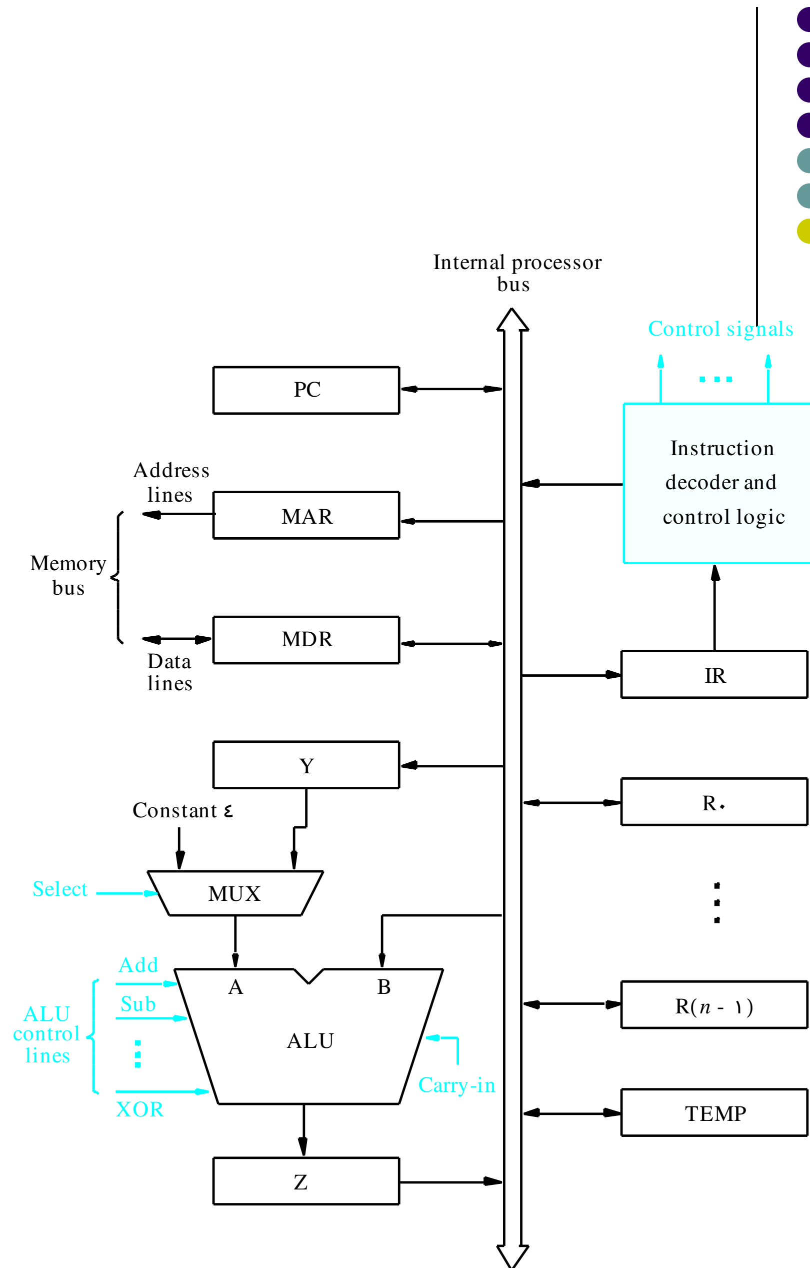
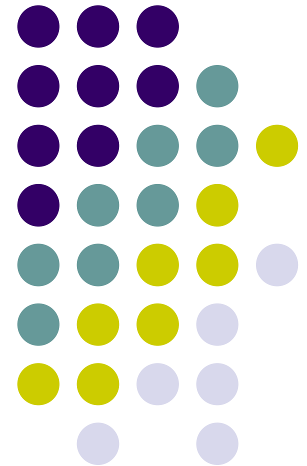
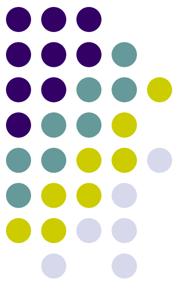


Figure 7.1. Single-bus organization of the datapath inside a processor.

Hardwired Control

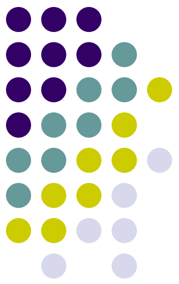


Overview



- To execute instructions, the processor must have some means of generating the control signals needed in the proper sequence.
- Two categories: hardwired control and microprogrammed control
- Hardwired system can operate at high speed; but with little flexibility.

Control Unit Organization



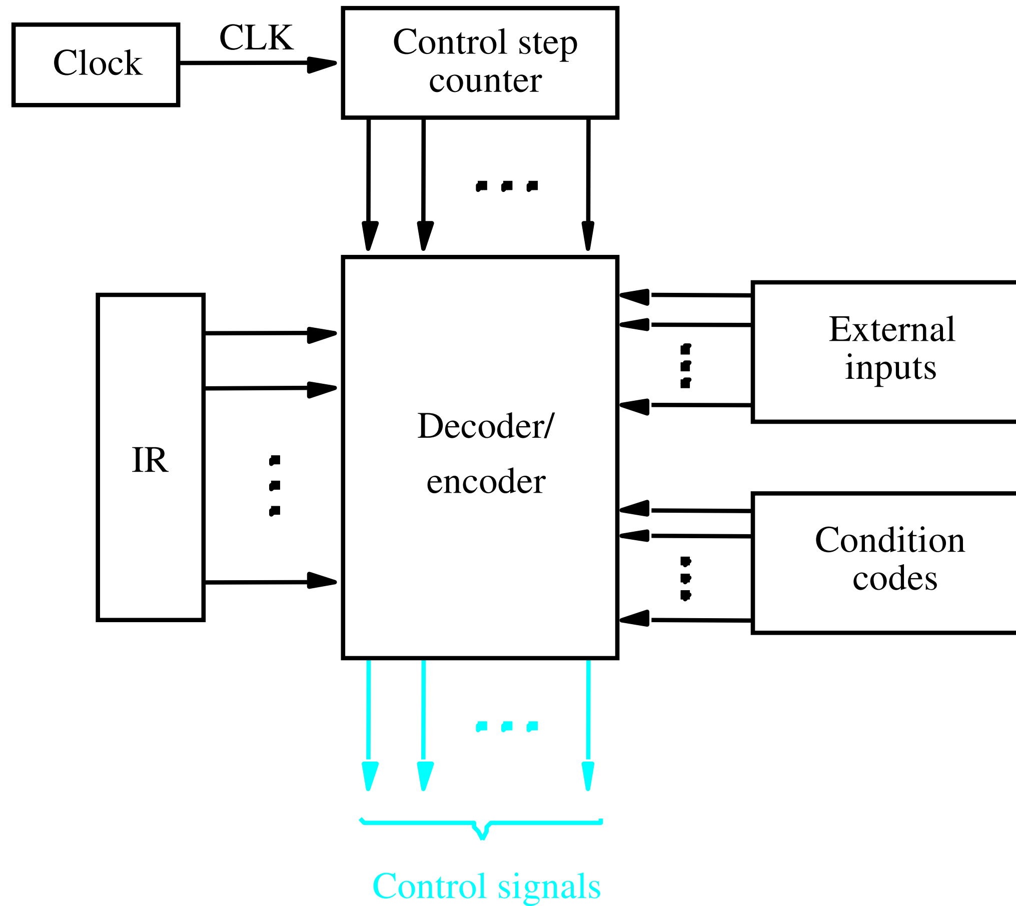
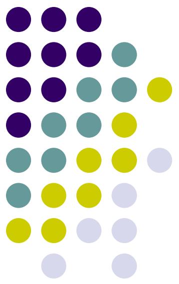


Figure 7.10. Control unit organization.

Detailed Block Description



Clock

CLK

Control step counter

Reset

Step decoder

$T_{۱}$ $T_{۲}$ ... T_n

IR

Instruction decoder

$INS_{۱}$

$INS_{۲}$

INS_m

Encoder

External inputs

Condition codes

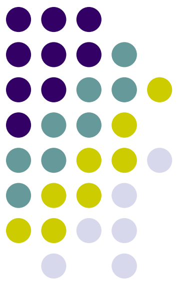
Run

End

Control signals

Figure ۷.۱۱. Separation of the decoding and encoding functions.

Generating Z_{in}



- $Z_{in} = T_1 + T_6 \cdot ADD + T_4 \cdot BR + \ldots$

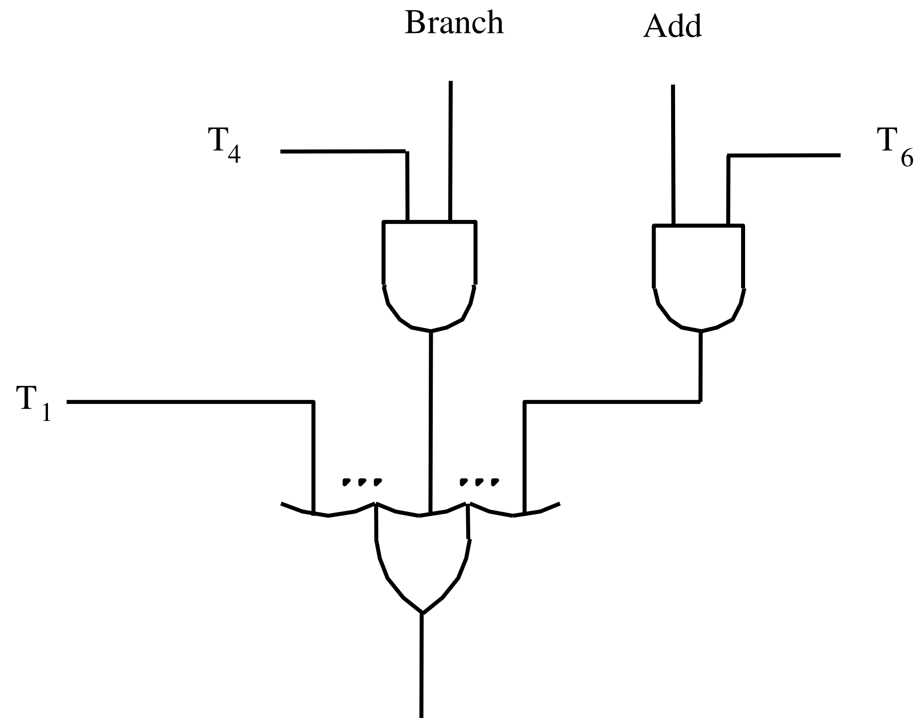
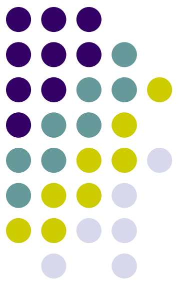


Figure 7.12. Generation of the Z_{in} control signal for the processor in Figure 7.1.

Generating End

- End = $T_7 \cdot ADD + T_5 \cdot BR + (T_5 \cdot N + T_4 \cdot \overline{N}) \cdot BRN + \ldots$



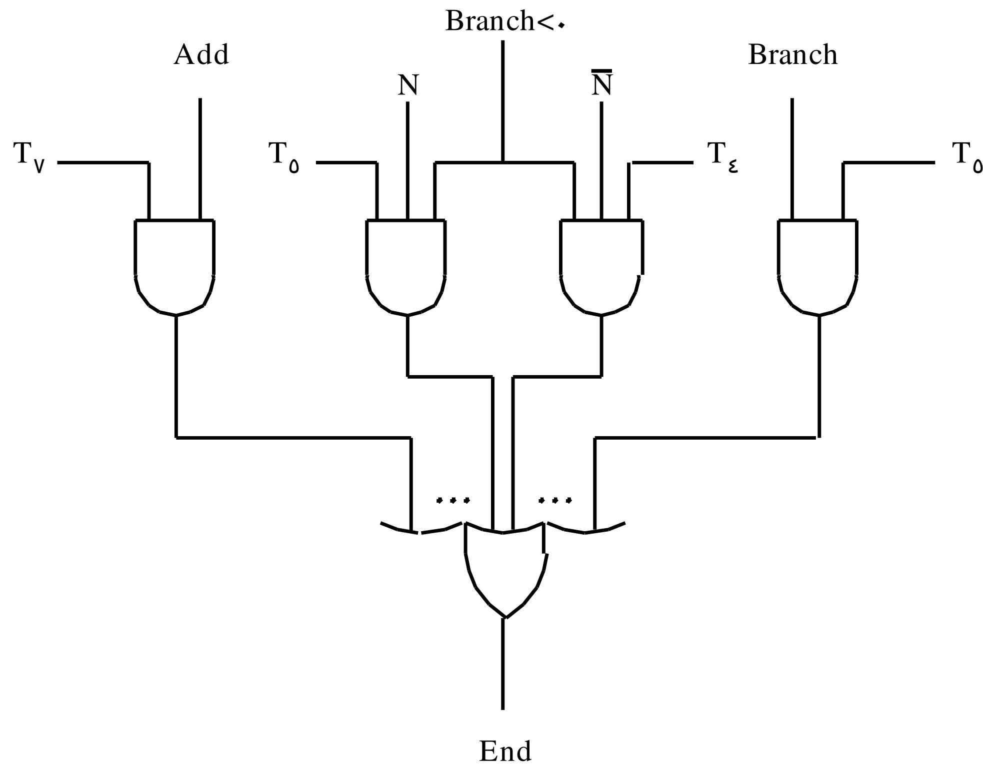
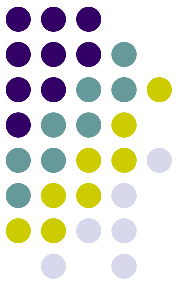


Figure ٧.١٣. Generation of the End control signal.

A Complete Processor



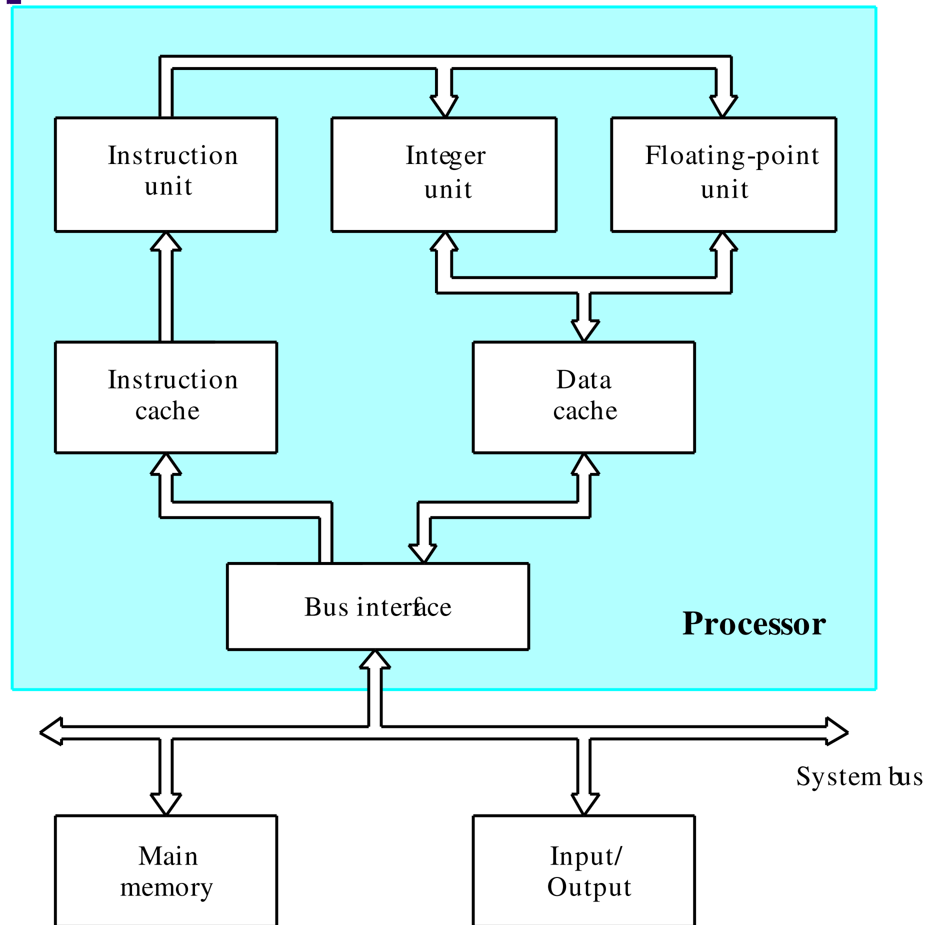
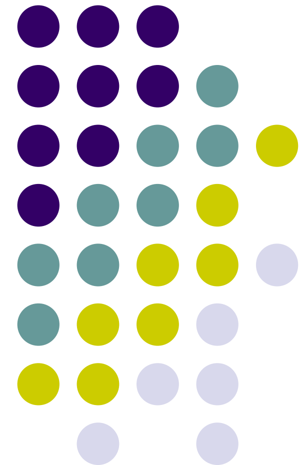
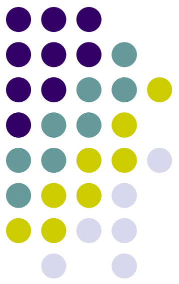


Figure ٧.١٤. Block diagram of a complete processor.

Microprogrammed Control





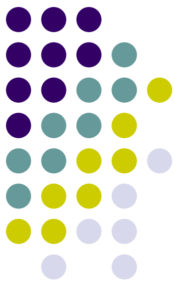
Overview

- Control signals are generated by a program similar to machine language programs.
- Control Word (CW); microroutine; microinstruction

Micro - instruction	٠٠	PC_{in}	PC_{out}	MAR_{in}	Read	MDR_{out}	IR_{in}	Y_{in}	Select	Add	Z_{in}	Z_{out}	R١$_{out}$	R١$_{in}$	R٣$_{out}$	WMFC	End	٠٠
١		٠	١	١	١	٠	٠	٠	١	١	١	٠	٠	٠	٠	٠	٠	
٢		١	٠	٠	٠	٠	٠	١	٠	٠	٠	١	٠	٠	٠	١	٠	
٣		٠	٠	٠	٠	١	١	٠	٠	٠	٠	٠	٠	٠	٠	٠	٠	
٤		٠	٠	١	١	٠	٠	٠	٠	٠	٠	٠	٠	٠	١	٠	٠	
٥		٠	٠	٠	٠	٠	٠	١	٠	٠	٠	٠	١	٠	٠	١	٠	
٦		٠	٠	٠	٠	١	٠	٠	٠	١	١	٠	٠	٠	٠	٠	٠	
٧		٠	٠	٠	٠	٠	٠	٠	٠	٠	٠	١	٠	١	٠	٠	١	

Figure ٧.١٥ An example of microinstructions for Figure ٧.٦.

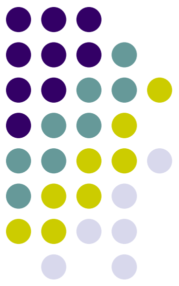
Overview



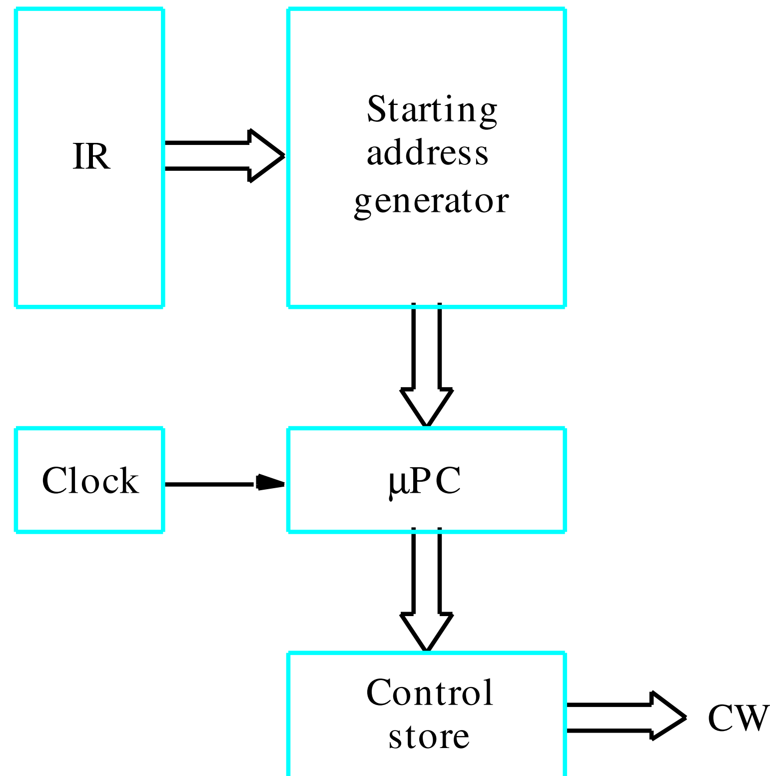
Step	Action
١	PC_{out} , MAR_{in} , Read, Select٤, Add, Z_{in}
٢	Z_{out} , PC_{in} , Y_{in} , WMF C
٣	MDR_{out} , IR_{in}
٤	$R٣_{out}$, MAR_{in} , Read
٥	$R١_{out}$, Y_{in} , WMF C
٦	MDR_{out} , SelectY, Add, Z_{in}
٧	Z_{out} , $R١_{in}$, End

Figure ٧.٦. Control sequence for execution of the instruction Add (R٣),R١ .

Overview



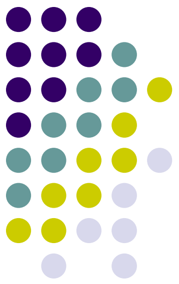
- Control store



One function cannot be carried out by this simple organization.

Figure ٧.١٦. Basic organization of a microprogrammed control unit.

Overview

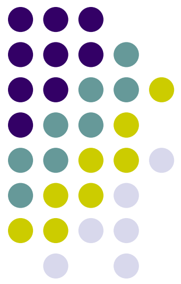


- The previous organization cannot handle the situation when the control unit is required to check the status of the condition codes or external inputs to choose between alternative courses of action.
- Use conditional branch microinstruction.

Address	Microinstruction
0	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
1	Z_{out}, PC_{in}, Y_{in}, WMFC
2	MDR_{out}, IR_{in}
3	Branch to starting address of appropriate microroutine
....	
25	If N=0, then branch to microinstruction 0
26	Offset-field-of-IR_{out}, SelectY, Add, Z_{in}
27	Z_{out}, PC_{in}, End

Figure 7.17. Microroutine for the instruction Branch<0.

Overview



IR

Starting and branch address generator

External inputs

Condition codes

Clock

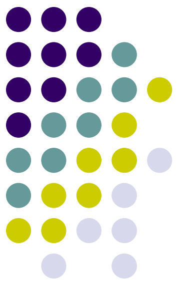
μ PC

Control store

CW

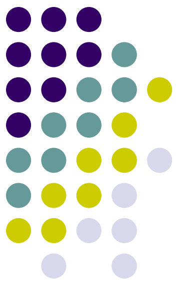
Figure 7.18. Organization of the control unit to allow conditional branching in the microprogram.

Microinstructions



- A straightforward way to structure microinstructions is to assign one bit position to each control signal.
- However, this is very inefficient.
- The length can be reduced: most signals are not needed simultaneously, and many signals are mutually exclusive.
- All mutually exclusive signals are placed in the same group in binary coding.

Partial Format for the Microinstructions



Microinstruction

F١	F٢	F٣	F٤	F٥	

F١ (٤ bits)	F٢ (٣ bits)	F٣ (٣ bits)	F٤ (٤ bits)	F٥ (٢ bits)
٠٠٠٠: No transfer	٠٠٠: No transfer	٠٠٠: No transfer	٠٠٠٠: Add	٠٠: No action
٠٠٠١: PC$_{out}$	٠٠١: PC$_{in}$	٠٠١: MAR$_{in}$	٠٠٠١: Sub	٠١: Read
٠٠١٠: MDR$_{out}$	٠١٠: IR$_{in}$	٠١٠: MDR$_{in}$	⋮	١٠: Write
٠٠١١: Z$_{out}$	٠١١: Z$_{in}$	٠١١: TEMP$_{in}$		
٠١٠٠: R٠$_{out}$	١٠٠: R٠$_{in}$	١٠٠: Y$_{in}$	١١١١: XOR	
٠١٠١: R١$_{out}$	١٠١: R١$_{in}$		١٦ ALU functions	
٠١١٠: R٢$_{out}$	١١٠: R٢$_{in}$			
٠١١١: R٣$_{out}$	١١١: R٣$_{in}$			
١٠١٠: TEMP$_{out}$				
١٠١١: Offset$_{out}$				

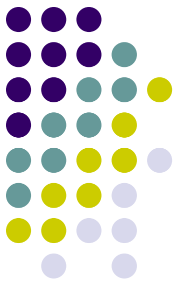
F٦	F٧	F٨	···

F٦ (١ bit)	F٧ (١ bit)	F٨ (١ bit)
٠: SelectY	٠: No action	٠: Continue
١: Select٤	١: WMFC	١: End

What is the price paid for this scheme?

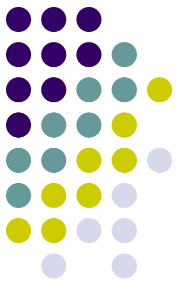
Figure ٧.١٩. An example of a partial format for field-encoded microinstructions.

Further Improvement

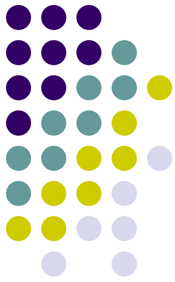


- Enumerate the patterns of required signals in all possible microinstructions. Each meaningful combination of active control signals can then be assigned a distinct code.
- Vertical organization
- Horizontal organization

Microprogram Sequencing



- If all microprograms require only straightforward sequential execution of microinstructions except for branches, letting a μPC governs the sequencing would be efficient.
- However, two disadvantages:
  - Having a separate microroutine for each machine instruction results in a large total number of microinstructions and a large control store.
  - Longer execution time because it takes more time to carry out the required branches.
- Example: Add src, Rdst
- Four addressing modes: register, autoincrement, autodecrement, and indexed (with indirect forms).



000 MAR ← [PC]; Read; Z ← [PC] + 4 ← Start

001 PC, Y ← [Z]; WMFC

002 IR ← [MDR]

003 Branch{InstDec, OR}

Microroutines for other instructions

Indexed

161 MAR ← [PC]; Read Z ← [PC] + 4

162 PC ← [Z]; WMFC

163 Y ← [MDR]

164 Z ← [Y] + [Rsrc]

165 MAR ← [Z]; Read

166 Branch{170, OR}; WMFC

Autodecrement

141 Z ← [Rsrc] − 4

142 MAR, Rsrc ← [Z]; Read

143 Branch{170, OR}; WMFC

Autoincrement

121 MAR ← [Rsrc]; Read Z ← [Rsrc] + 4

122 Rsrc ← [Z]

123 Branch{170, OR}; WMFC

Register indirect

111 MAR ← [Rsrc]; Read

112 Branch{171}; WMFC

α

170 MAR ← [MDR]; Read; WMFC

171 Y ← [MDR]

Register direct

101 Y ← [Rsrc]

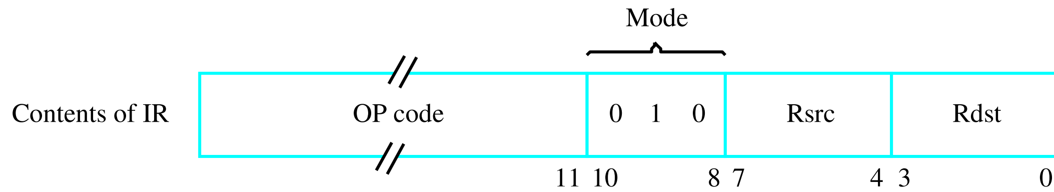
102 Branch{172}

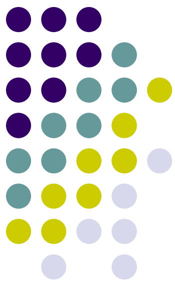
172 Z ← [Y] + [Rdst]

173 Rdst ← [Z] → End

Figure 7.20. Flowchart of a microprogram for the Add src,Rdst instruction.

- Bit-ORing
- Wide-Branch Addressing
- WMFC



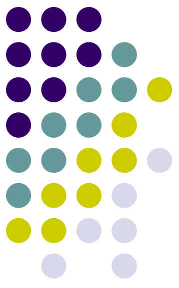


Address (octal)	Microinstruction
000	PC_{out}, MAR_{in}, Read, Select4, Add, Z_{in}
001	Z_{out}, PC_{in}, Y_{in}, WMFC
002	MDR_{out}, IR_{in}
003	μBranch $\{\mu PC \leftarrow 101$ (from Instruction decoder);
	$\mu PC_{5,4} \leftarrow [IR_{10,9}]$; $\mu PC_3 \leftarrow [\overline{IR_{10}}] \cdot [\overline{IR_9}] \cdot [IR_8]\}$
121	$Rsrc_{out}$, MAR_{in}, Read, Select4, Add, Z_{in}
122	Z_{out}, $Rsrc_{in}$
123	μBranch $\{\mu PC \leftarrow 170; \mu PC_0 \leftarrow [\overline{IR_8}]\}$, WMFC
170	MDR_{out}, MAR_{in}, Read, WMFC
171	MDR_{out}, Y_{in}
172	$Rdst_{out}$, SelectY, Add, Z_{in}
173	Z_{out}, $Rdst_{in}$, End

Figure 7.21. Microinstruction for Add (Rsrc)+,Rdst.

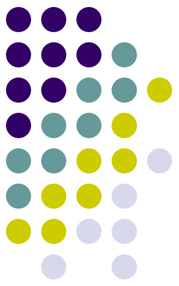
Note: Microinstruction at location 170 is not executed for this addressing mode.

Microinstructions with Next-Address Field



- The microprogram we discussed requires several branch microinstructions, which perform no useful operation in the datapath.
- A powerful alternative approach is to include an address field as a part of every microinstruction to indicate the location of the next microinstruction to be fetched.
- Pros: separate branch microinstructions are virtually eliminated; few limitations in assigning addresses to microinstructions.
- Cons: additional bits for the address field (around 1/6)

Microinstructions with Next-Address Field



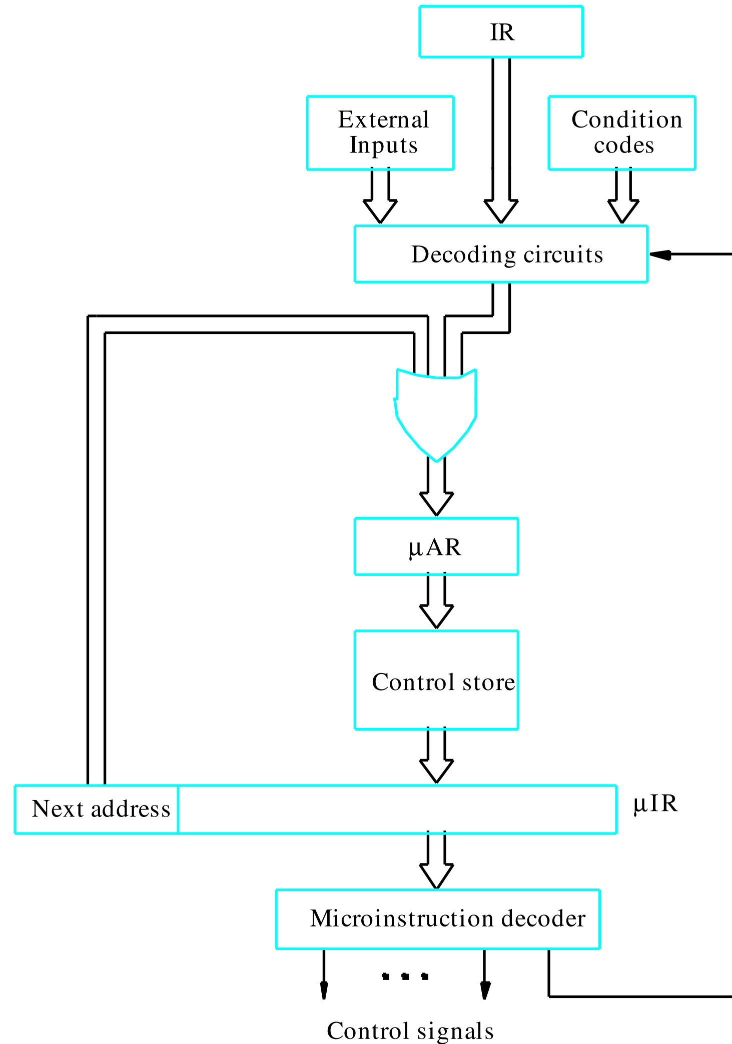


Figure ٧.٢٢. Microinstruction-sequencing organization.

Microinstruction

F٠	F١	F٢	F٣

F٠ (٨ bits)	F١ (٣ bits)	F٢ (٣ bits)	F٣ (٣ bits)
Address of next microinstruction	٠٠٠: No transfer ٠٠١: PC_{out} ٠١٠: MDR_{out} ٠١١: Z_{out} ١٠٠: $Rsrc_{out}$ ١٠١: $Rdst_{out}$ ١١٠: $TEMP_{out}$	٠٠٠: No transfer ٠٠١: PC_{in} ٠١٠: IR_{in} ٠١١: Z_{in} ١٠٠: $Rsrc_{in}$ ١٠١: $Rdst_{in}$	٠٠٠: No transfer ٠٠١: MAR_{in} ٠١٠: MDR_{in} ٠١١: $TEMP_{in}$ ١٠٠: Y_{in}

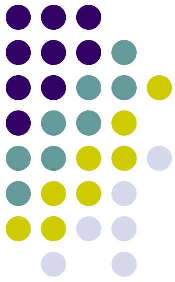
F٤	F٥	F٦	F٧

F٤ (٤ bits)	F٥ (٢ bits)	F٦ (١ bit)	F٧ (١ bit)
٠٠٠٠: Add ٠٠٠١: Sub ⋮ ١١١١: XOR	٠٠: No action ٠١: Read ١٠: Write	٠: SelectY ١: Select٤	٠: No action ١: WMFC

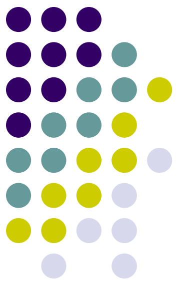
F٨	F٩	F١٠

F٨ (١ bit)	F٩ (١ bit)	F١٠ (١ bit)
٠: NextAdrs ١: InstDec	٠: No action ١: OR_{mode}	٠: No action ١: OR_{indsrc}

Figure ٧.٢٣. Format for microinstructions in the example of Section ٧.٥.٣.

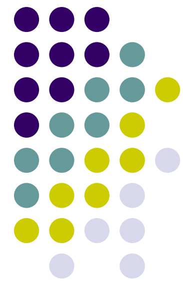


Implementation of the Microroutine



Octal address	F٠	F١	F٢	F٣	F٤	F٥	F٦	F٧	F٨	F٩	F١٠
٠ ٠ ٠	٠ ٠ ٠ ٠ ٠ ٠ ٠ ١	٠ ٠ ١	٠ ١ ١	٠ ٠ ١	٠ ٠ ٠ ٠	٠ ١	١	٠	٠	٠	٠
٠ ٠ ١	٠ ٠ ٠ ٠ ٠ ٠ ١ ٠	٠ ١ ١	٠ ٠ ١	١ ٠ ٠	٠ ٠ ٠ ٠	٠ ٠	٠	١	٠	٠	٠
٠ ٠ ٢	٠ ٠ ٠ ٠ ٠ ٠ ١ ١	٠ ١ ٠	٠ ١ ٠	٠ ٠ ٠	٠ ٠ ٠ ٠	٠ ٠	٠	٠	٠	٠	٠
٠ ٠ ٣	٠ ٠ ٠ ٠ ٠ ٠ ٠ ٠	٠ ٠ ٠	٠ ٠ ٠	٠ ٠ ٠	٠ ٠ ٠ ٠	٠ ٠	٠	٠	١	١	٠
١ ٢ ١	٠ ١ ٠ ١ ٠ ٠ ١ ٠	١ ٠ ٠	٠ ١ ١	٠ ٠ ١	٠ ٠ ٠ ٠	٠ ١	١	٠	٠	٠	٠
١ ٢ ٢	٠ ١ ١ ١ ١ ٠ ٠ ٠	٠ ١ ١	١ ٠ ٠	٠ ٠ ٠	٠ ٠ ٠ ٠	٠ ٠	٠	١	٠	٠	١
١ ٧ ٠	٠ ١ ١ ١ ١ ٠ ٠ ١	٠ ١ ٠	٠ ٠ ٠	٠ ٠ ١	٠ ٠ ٠ ٠	٠ ١	٠	١	٠	٠	٠
١ ٧ ١	٠ ١ ١ ١ ١ ٠ ١ ٠	٠ ١ ٠	٠ ٠ ٠	١ ٠ ٠	٠ ٠ ٠ ٠	٠ ٠	٠	٠	٠	٠	٠
١ ٧ ٢	٠ ١ ١ ١ ١ ٠ ١ ١	١ ٠ ١	٠ ١ ١	٠ ٠ ٠	٠ ٠ ٠ ٠	٠ ٠	٠	٠	٠	٠	٠
١ ٧ ٣	٠ ٠ ٠ ٠ ٠ ٠ ٠ ٠	٠ ١ ١	١ ٠ ١	٠ ٠ ٠	٠ ٠ ٠ ٠	٠ ٠	٠	٠	٠	٠	٠

Figure ٧.٢٤. Implementation of the microroutine of Figure ٧.٢١ using a next-microinstruction address field. (See Figure ٧.٢٣ for encoded signals.)



$R۱۵_{in}$ $R۱۵_{out}$ ... $R۰_{in}$ $R۰_{out}$

Decoder

Decoder

IR | Rsrc | Rdst

External inputs

Condition codes

Decoding circuits

$InstDec_{out}$

OR_{mode}

OR_{indsrc}

μAR

Control store

Next address | F۱ | F۲ | ... | F۸ | F۹ | F۱۰

$Rdst_{out}$

$Rdst_{in}$

$Rsrc_{out}$

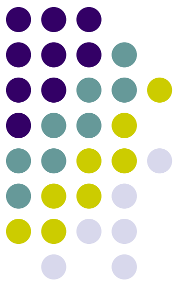
$Rsrc_{in}$

Microinstruction decoder

Other control signals

Figure ۷.۲۵. Some details of the control-signal-generating circuitry.

bit-ORing



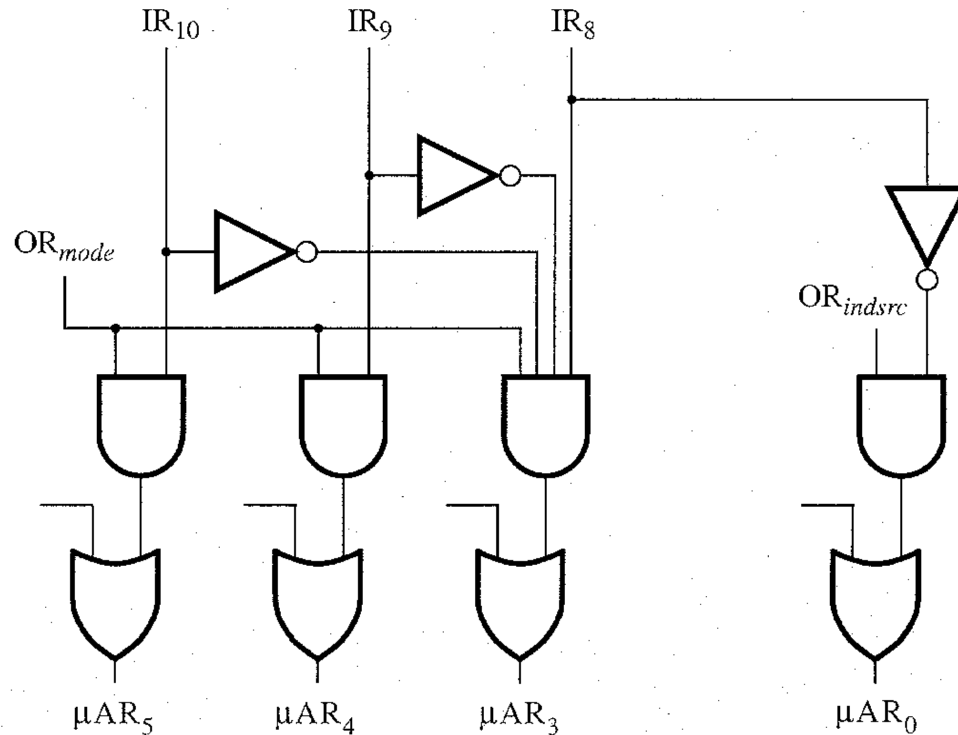
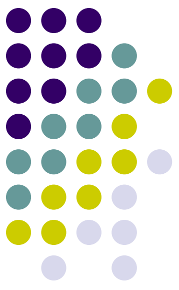


Figure 7.26. Control circuitry for bit-ORing

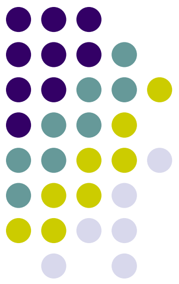
(part of the decoding circuits in Figure 7.25).

Prefetching



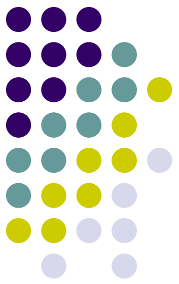
- One drawback of Micro Programmed control is that it leads to slower operating speed because of the time it takes to fetch microinstructions from control store
- Faster operation is achieved if the next microinstruction is prefetched while the current one is executing
- In this way execution time is overlapped with fetch time

Prefetching – Disadvantages



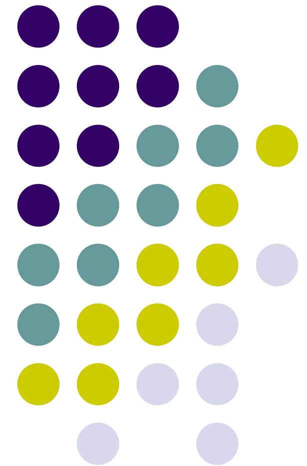
- Sometimes the status flag & the result of the currently executed microinstructions are needed to know the next address
- Thus there is a probability of wrong instructions being prefetched
- In this case fetch must be repeated with the correct address

Emulation

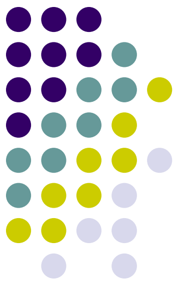


- Emulation allows us to replace obsolete equipment with more up-to-date machines
- It facilitates transitions to new computer systems with minimal disruption
- It is the easiest way when machines with similar architecture are involved

Pipelining

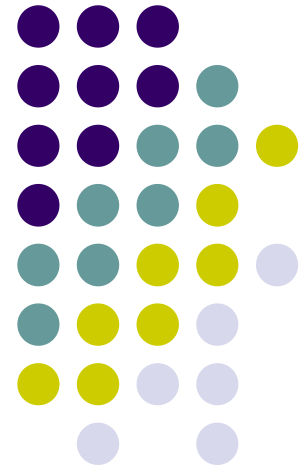


Overview

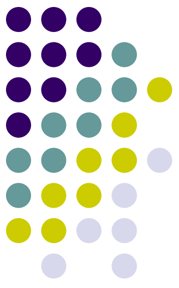


- Pipelining is widely used in modern processors.
- Pipelining improves system performance in terms of throughput.
- Pipelined organization requires sophisticated compilation techniques.

Basic Concepts

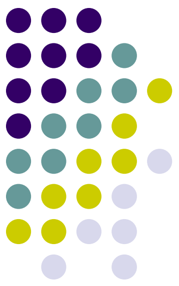


Making the Execution of Programs Faster

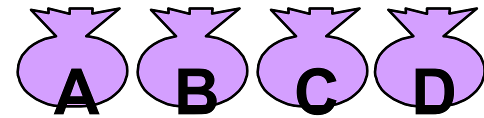


- Use faster circuit technology to build the processor and the main memory.
- Arrange the hardware so that more than one operation can be performed at the same time.
- In the latter way, the number of operations performed per second is increased even though the elapsed time needed to perform any one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

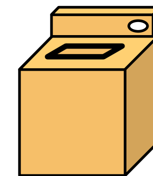
Traditional Pipeline Concept



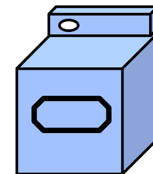
- Laundry Example
- Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold



- Washer takes 30 minutes

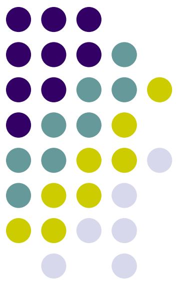


- Dryer takes 40 minutes



- “Folder” takes 20 minutes

Traditional Pipeline Concept



6 PM 7 8 9 10 11 Midnight

Time

30 40 20 30 40 20 30 40 20 30 40 20

A

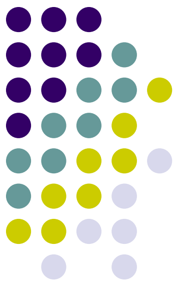
B

C

D

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

Traditional Pipeline Concept



6 PM 7 8 9 10 11 Midnight

Time

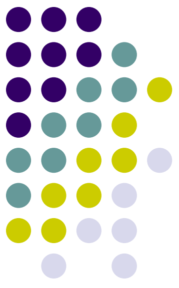
Task Order

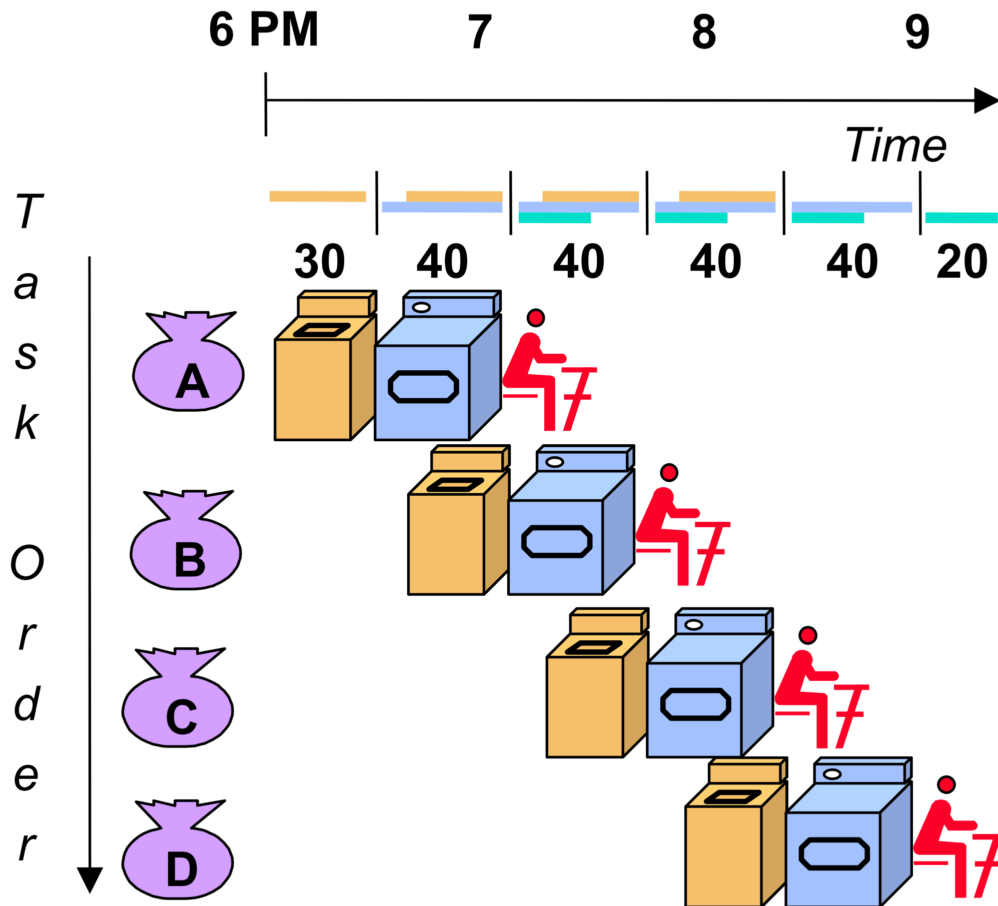
30 40 40 40 40 20

A B C D

- Pipelined laundry takes 3.5 hours for 4 loads

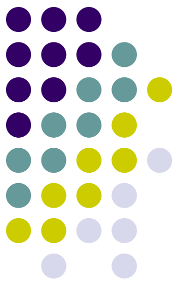
Traditional Pipeline Concept



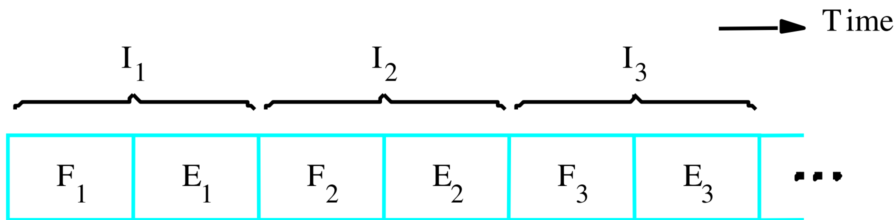


- Pipelining doesn't help latency of single task, it helps throughput of entire workload
- Pipeline rate limited by slowest pipeline stage
- Multiple tasks operating simultaneously using different resources
- Potential speedup = Number pipe stages
- Unbalanced lengths of pipe stages reduces speedup
-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

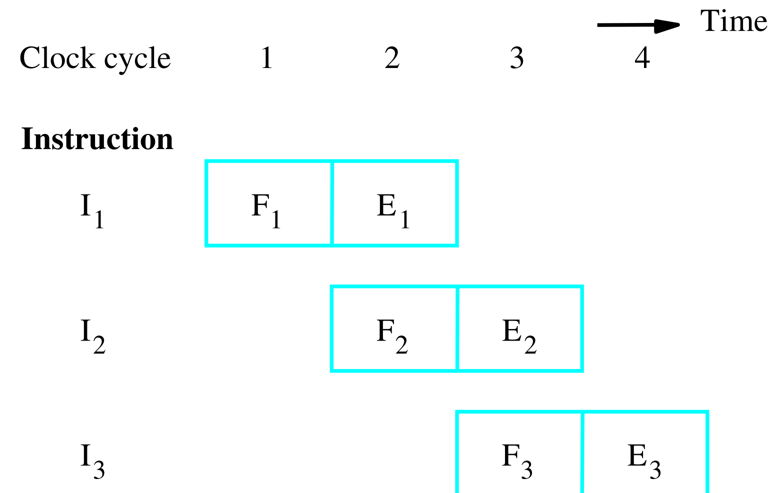
Use the Idea of Pipelining in a Computer



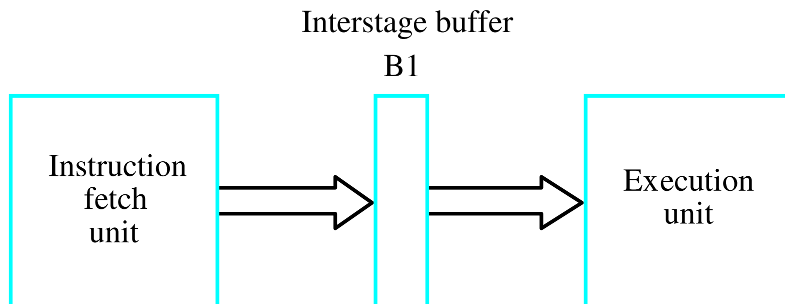
Fetch + Execution



(a) Sequential execution



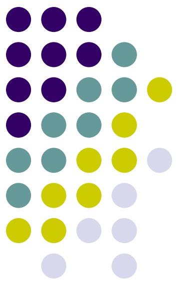
(c) Pipelined execution



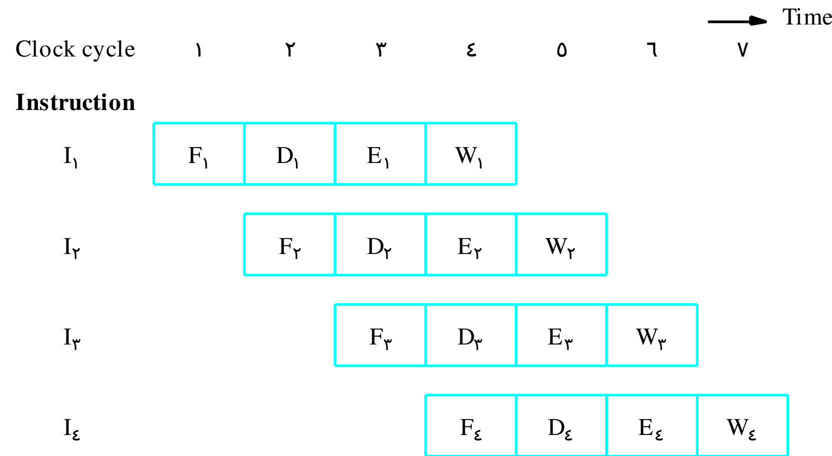
(b) Hardware organization

Figure 8.1. Basic idea of instruction pipelining.

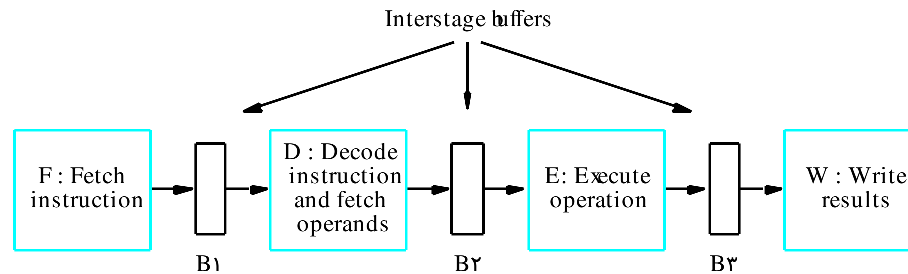
Use the Idea of Pipelining in a Computer



Fetch + Decode
+ Execution + Write



(a) Instruction execution divided into four steps

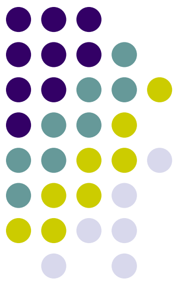


(b) Hardware organization

Textbook page: 457

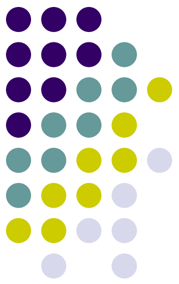
Figure ٨.٢. A ٤-stage pipeline.

Role of Cache Memory



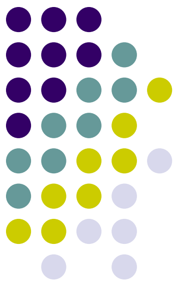
- Each pipeline stage is expected to complete in one clock cycle.
- The clock period should be long enough to let the slowest pipeline stage to complete.
- Faster stages can only wait for the slowest one to complete.
- Since main memory is very slow compared to the execution, if each instruction needs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

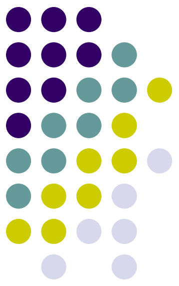


Time →

<table>
<tr><th>Clock cycle</th><th>١</th><th>٢</th><th>٣</th><th>٤</th><th>٥</th><th>٦</th><th>٧</th><th>٨</th><th>٩</th></tr>
<tr><td>Instruction</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>I١</td><td>F١</td><td>D١</td><td>E١</td><td>W١</td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>I٢</td><td></td><td>F٢</td><td>D٢</td><td colspan="3">E٢</td><td>W٢</td><td></td><td></td></tr>
<tr><td>I٣</td><td></td><td></td><td>F٣</td><td>D٣</td><td></td><td></td><td>E٣</td><td>W٣</td><td></td></tr>
<tr><td>I٤</td><td></td><td></td><td></td><td>F٤</td><td></td><td></td><td>D٤</td><td>E٤</td><td>W٤</td></tr>
<tr><td>I٥</td><td></td><td></td><td></td><td></td><td></td><td></td><td>F٥</td><td>D٥</td><td>E٥</td></tr>
</table>

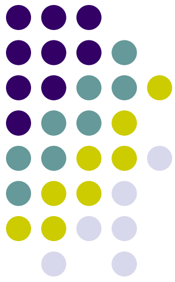
Figure ٨.٣. Effect of an execution operation taking more than one clock cycle.

Pipeline Performance

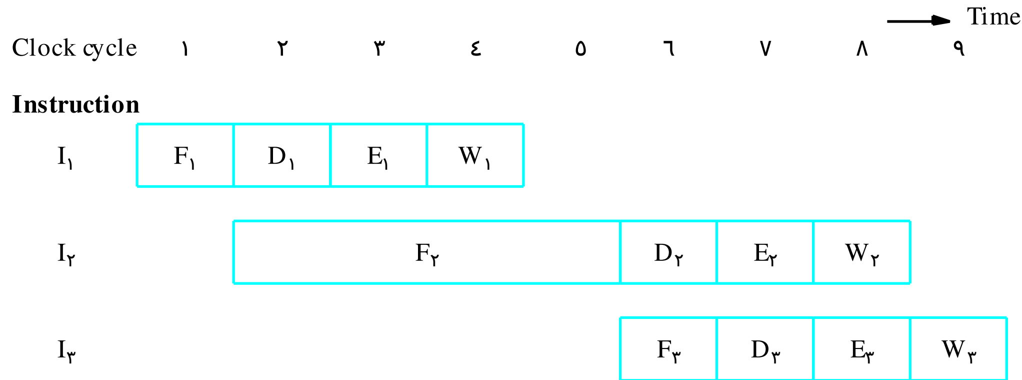


- The previous pipeline is said to have been stalled for two clock cycles.
- Any condition that causes a pipeline to stall is called a hazard.
- Data hazard – any condition in which either the source or the destination operands of an instruction are not available at the time expected in the pipeline. 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.

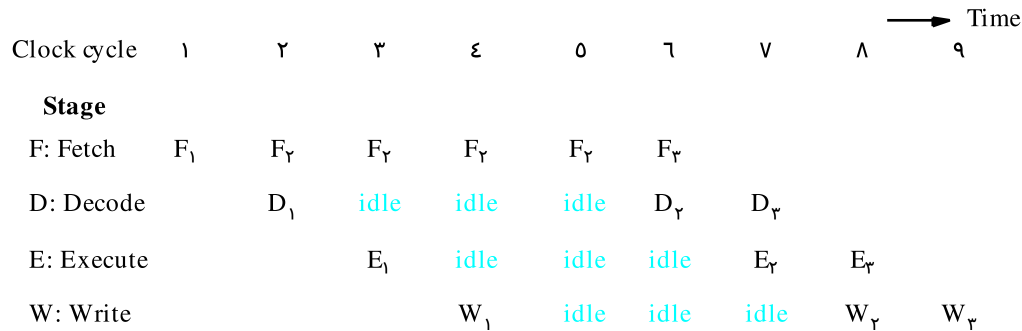
Pipeline Performance



Instruction hazard



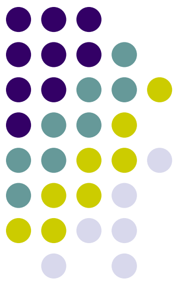
(a) Instruction execution steps in successive clock cycles



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

Idle periods – stalls (bubbles)

Figure ٨.٤. Pipeline stall caused by a cache miss in F٢.



Pipeline Performance

Structural hazard

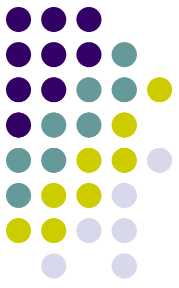
Load X(R1), R2

Time →

Instruction \ Clock cycle	١	٢	٣	٤	٥	٦	٧
I١	F١	D١	E١	W١			
I٢ (Load)		F٢	D٢	E٢	M٢	W٢	
I٣			F٣	D٣	E٣		W٣
I٤				F٤	D٤		E٤
I٥						F٥	D٥

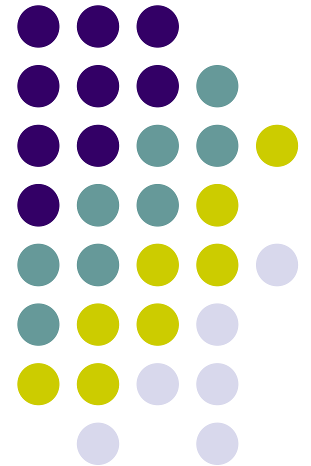
Figure ٨.٥. Effect of a Load instruction on pipeline timing.

Pipeline Performance

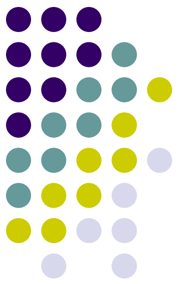


- Again, pipelining does not result in individual instructions being executed faster; rather, it is the throughput that increases.
- Throughput is measured by the rate at which instruction execution is completed.
- Pipeline stall causes degradation in pipeline performance.
- We need to identify all hazards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

  $A \leftarrow 3 + A$

  $B \leftarrow 4 \times A$
- No hazard

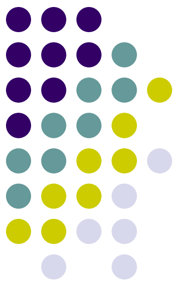
  $A \leftarrow 5 \times C$

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

  Mul R2, R3, R4

  Add R5, R4, R6

Data Hazards

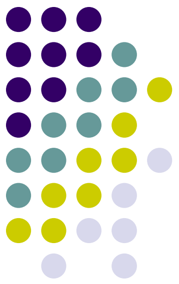


Time →

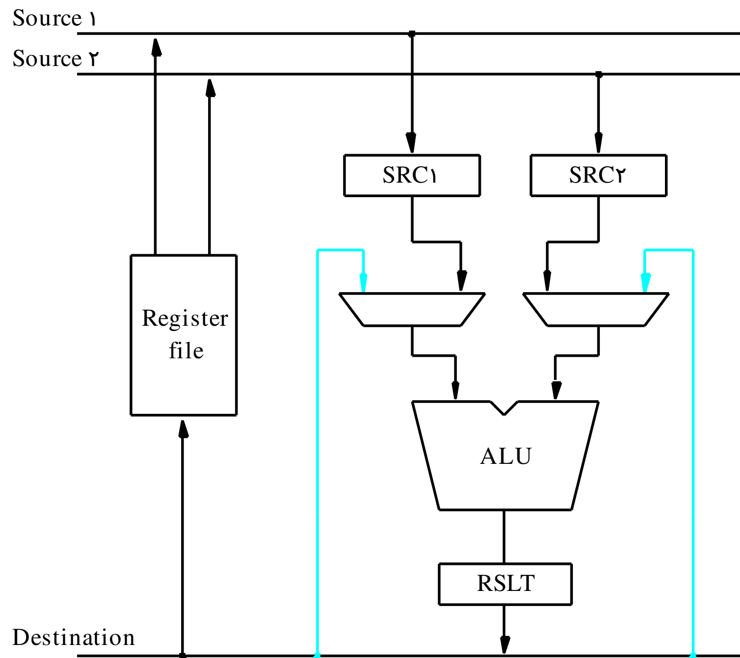
Clock cycle	١	٢	٣	٤	٥	٦	٧	٨	٩
Instruction									
$I_{١}$ (Mul)	$F_{١}$	$D_{١}$	$E_{١}$	$W_{١}$					
$I_{٢}$ (Add)		$F_{٢}$	$D_{٢}$		$D_{٢A}$	$E_{٢}$	$W_{٢}$		
$I_{٣}$			$F_{٣}$			$D_{٣}$	$E_{٣}$	$W_{٣}$	
$I_{٤}$						$F_{٤}$	$D_{٤}$	$E_{٤}$	$W_{٤}$

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

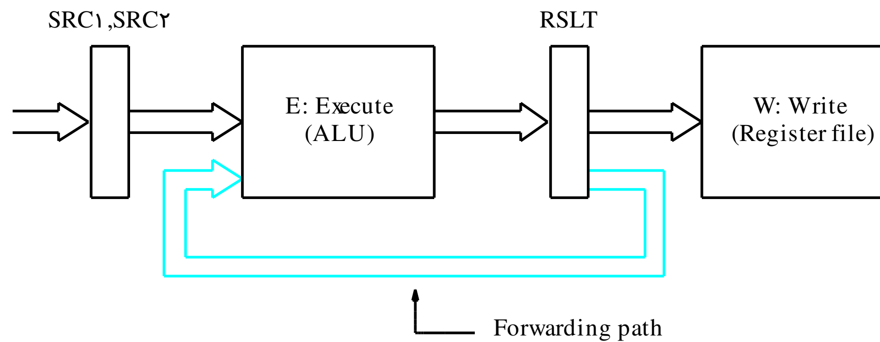
Operand Forwarding



- 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.

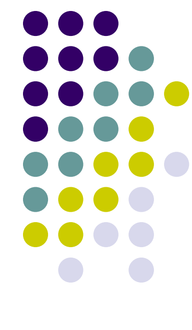


(a) Datapath

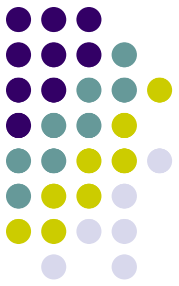


(b) Position of the source and result registers in the processor pipeline

Figure ۸.۷. Operand forwarding in a pipelined processor.



Handling Data Hazards in Software

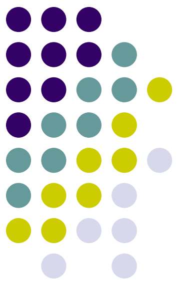


- Let the compiler detect and handle the hazard:

```
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

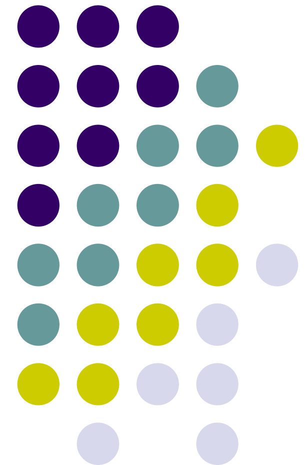


- The previous example is explicit and easily detected.
- Sometimes an instruction changes the contents of a register other than the one named as the destination.
- When a location other than one explicitly named in an instruction as a destination operand is affected, the instruction is said to have a side effect. (Example?)
- Example: conditional code flags:

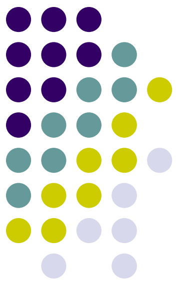
  Add R1, R3

  AddWithCarry R2, R4
- Instructions designed for execution on pipelined hardware should have few side effects.

Instruction Hazards

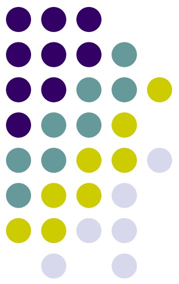


Overview



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

Unconditional Branches



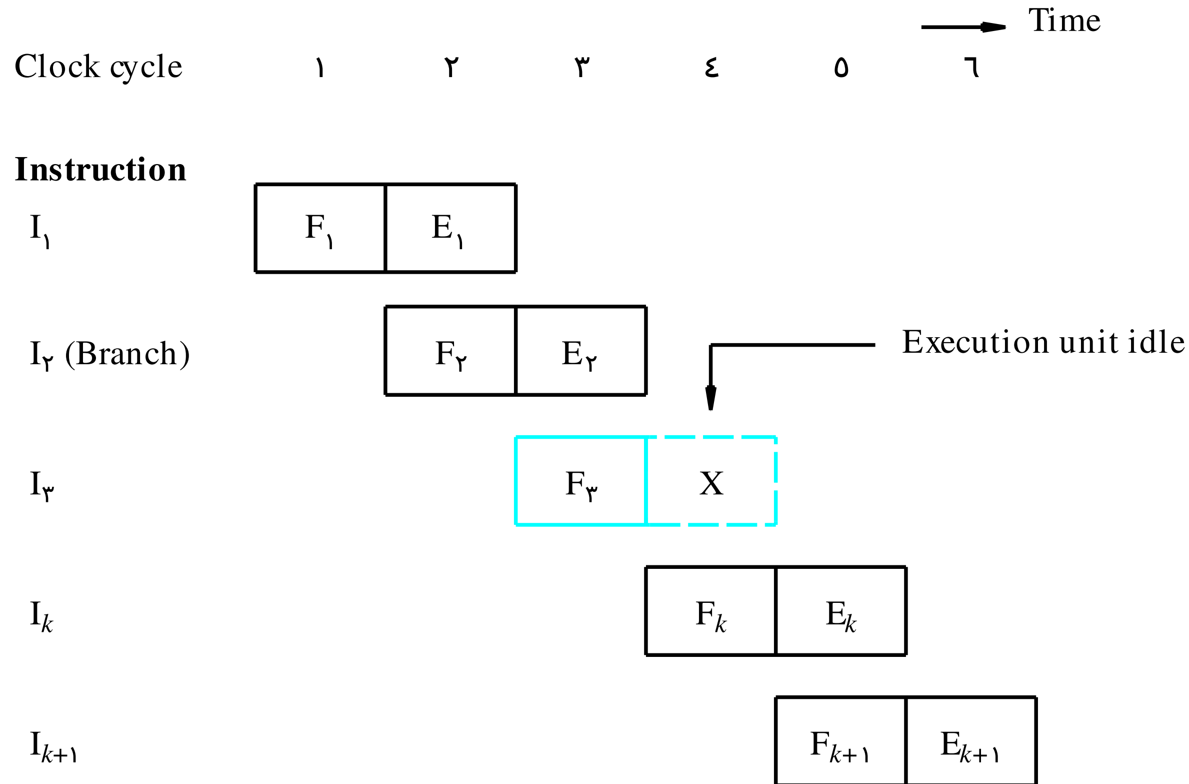
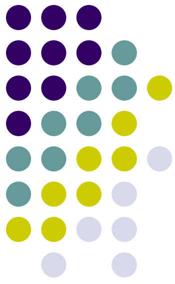


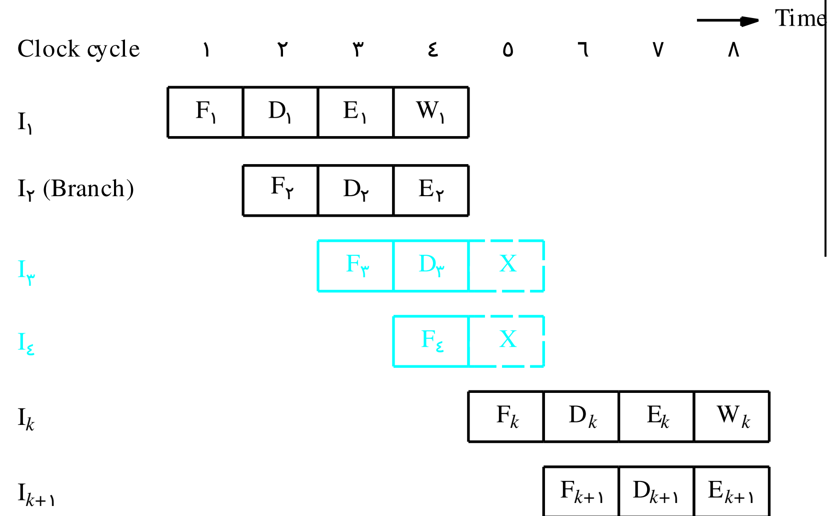
Figure ٨.٨. An idle cycle caused by a branch instruction.

Branch Timing

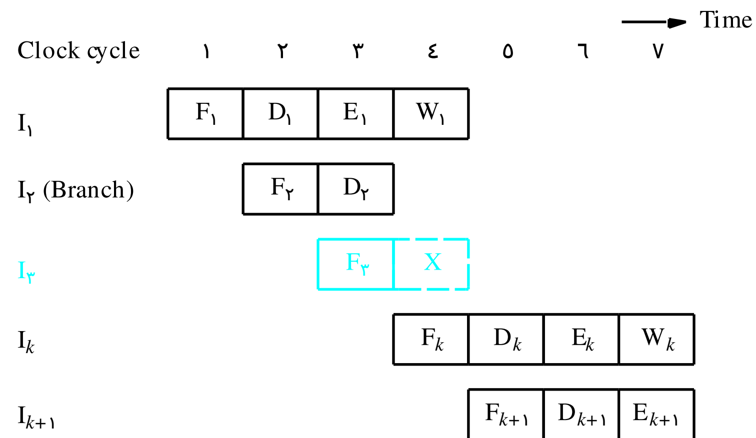


- Branch penalty

- Reducing the penalty



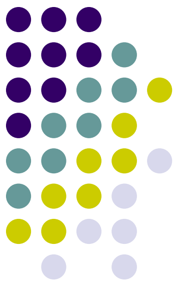
(a) Branch address computed in Execute stage



(b) Branch address computed in Decode stage

Figure ٨.٩. Branch timing.

Instruction Queue and Prefetching



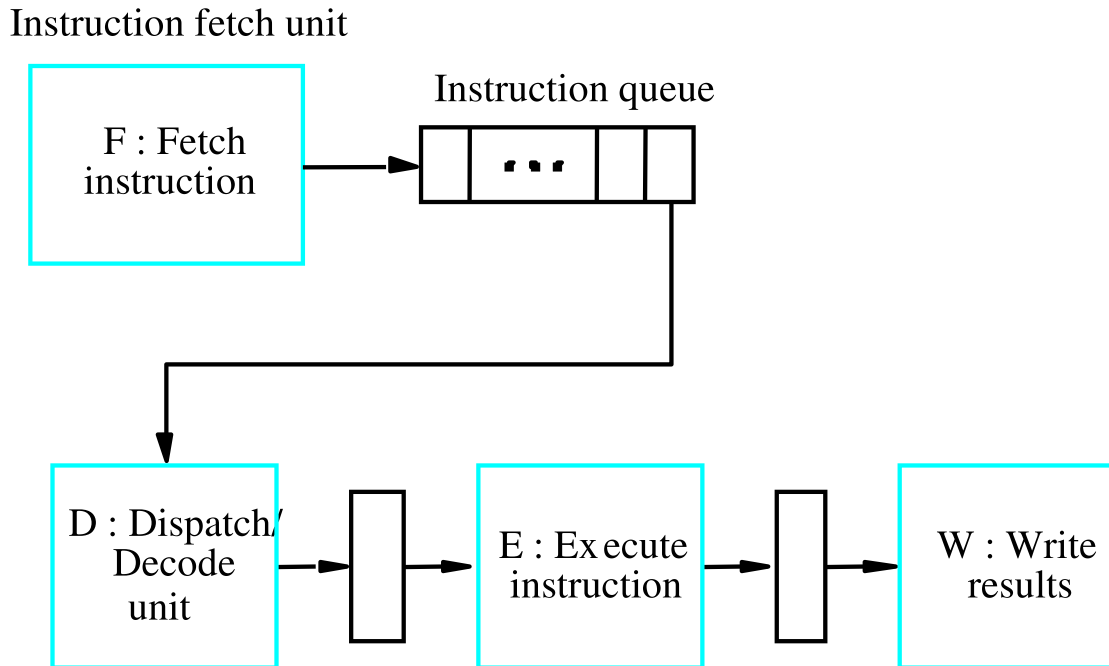
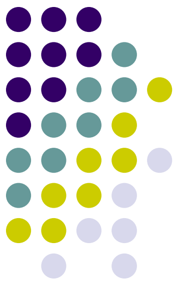


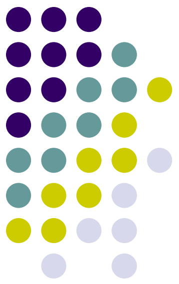
Figure 8.10. Use of an instruction queue in the hardware organization of Figure 8.2*b*.

Conditional Braches



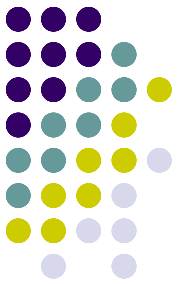
- A conditional branch instruction introduces the added hazard caused by the dependency of the branch condition on the result of a preceding instruction.
- The decision to branch cannot be made until the execution of that instruction has been completed.
- Branch instructions represent about 20% of the dynamic instruction count of most programs.

Delayed Branch



- 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 Branch



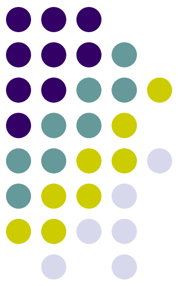
LOOP	Shift_left	R1
	Decrement	R2
	Branch=0	LOOP
NEXT	Add	R1,R3

(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.



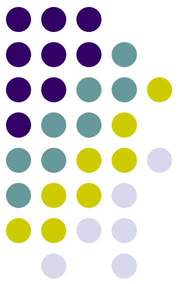
Delayed Branch

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

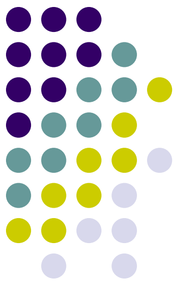
Figure 8.13. Execution timing showing the delay slot being filled during the last two passes through the loop in Figure 8.12.

Branch 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



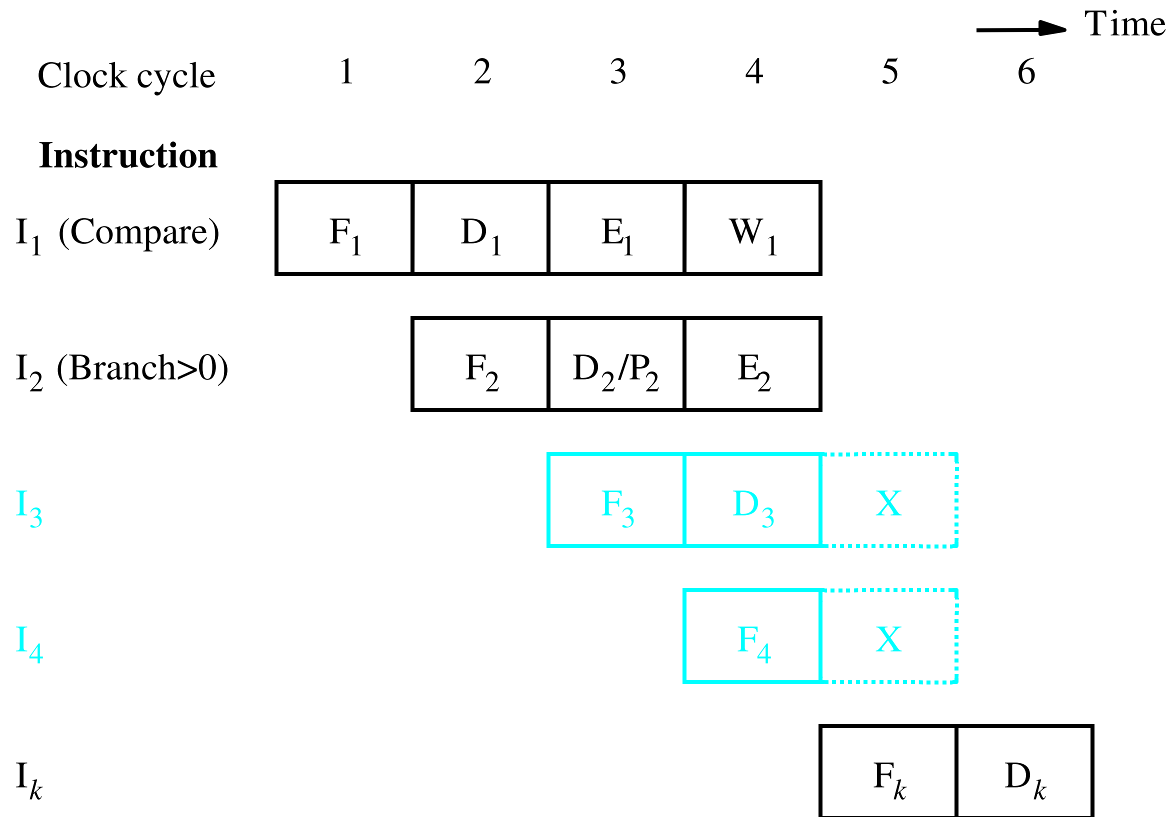
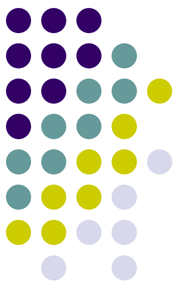


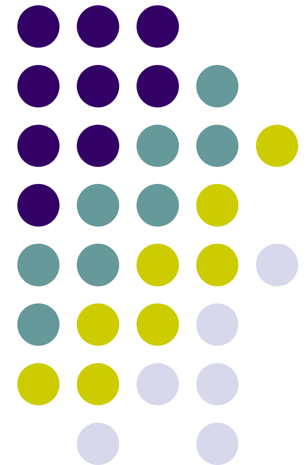
Figure 8.14. Timing when a branch decision has been incorrectly predicted as not taken.

Branch Prediction

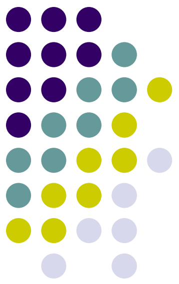


- Better performance can be achieved if we arrange for some branch instructions to be predicted as taken and others as not taken.
- Use hardware to observe whether the target address is lower or higher than that of the branch instruction.
- Let compiler include a branch prediction bit.
- So far the branch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

Influence on Instruction Sets

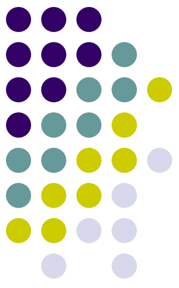


Overview

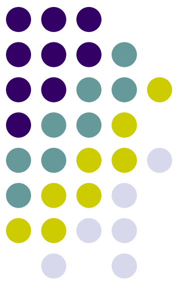


- Some instructions are much better suited to pipeline execution than others.
- Addressing modes
- Conditional code flags

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

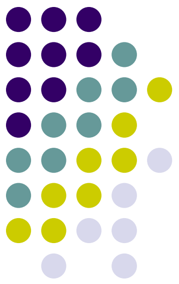
→ Time

Clock cycle	١	٢	٣	٤	٥	٦	٧
Instruction							
I١	F١	D١	E١	W١			
I٢ (Load)		F٢	D٢	E٢	M٢	W٢	
I٣			F٣	D٣	E٣		W٣
I٤				F٤	D٤		E٤
I٥						F٥	D٥

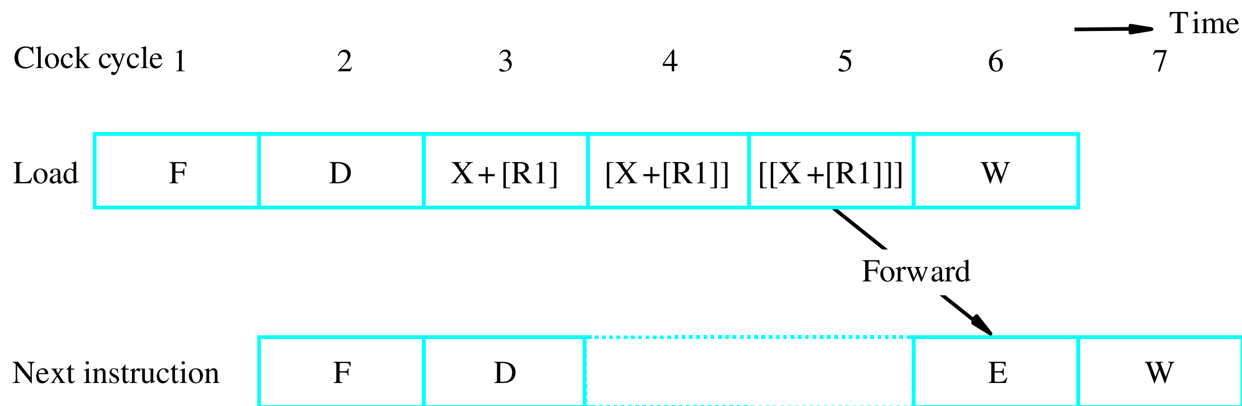
Load (R1), R2

Figure ٨.٥. Effect of a Load instruction on pipeline timing.

Complex Addressing Mode

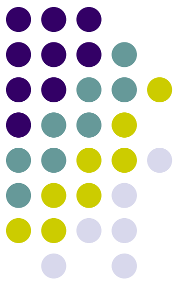


Load (X(R1)), R2

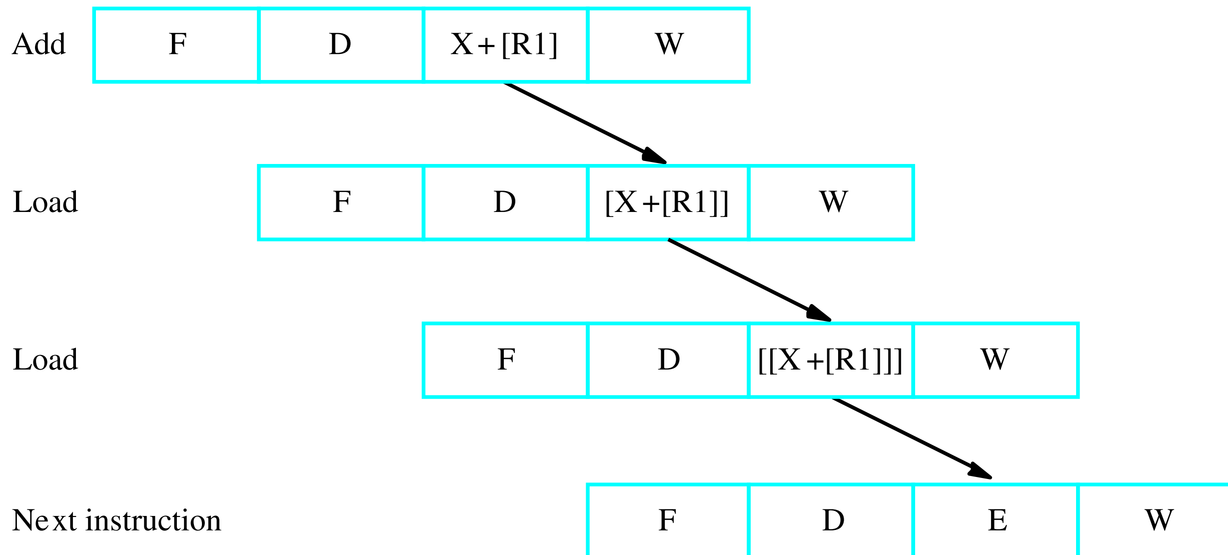


(a) Complex addressing mode

Simple Addressing Mode

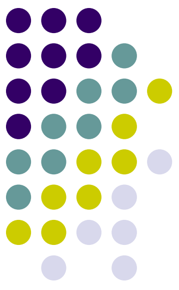


Add #X, R1, R2
Load (R2), R2
Load (R2), R2



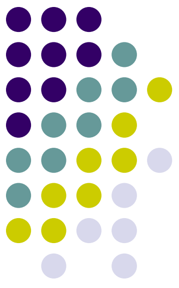
(b) Simple addressing mode

Addressing Modes



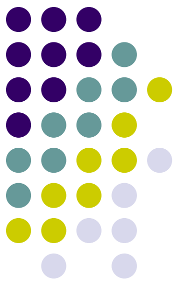
- In a pipelined processor, complex addressing modes do not necessarily lead to faster execution.
- Advantage: reducing the number of instructions / program space
- Disadvantage: cause pipeline to stall / more hardware to decode / not convenient for compiler to work with
- Conclusion: complex addressing modes are not suitable for pipelined execution.

Addressing Modes



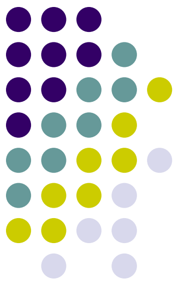
- Good addressing modes should have:
  - Access to an operand does not require more than one access to the memory
  - Only load and store instruction access memory operands
  - The addressing modes used do not have side effects
- Register, register indirect, index

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	. . .

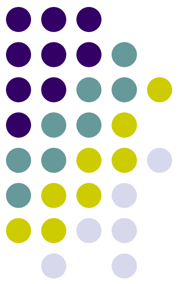
(a) A program fragment

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

(b) Instructions reordered

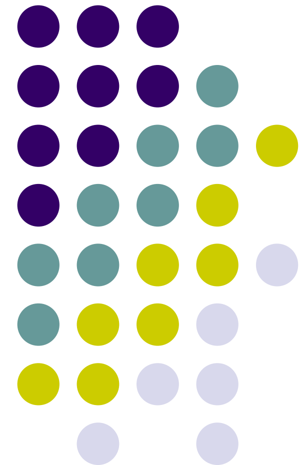
Figure 8.17. Instruction reordering.

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

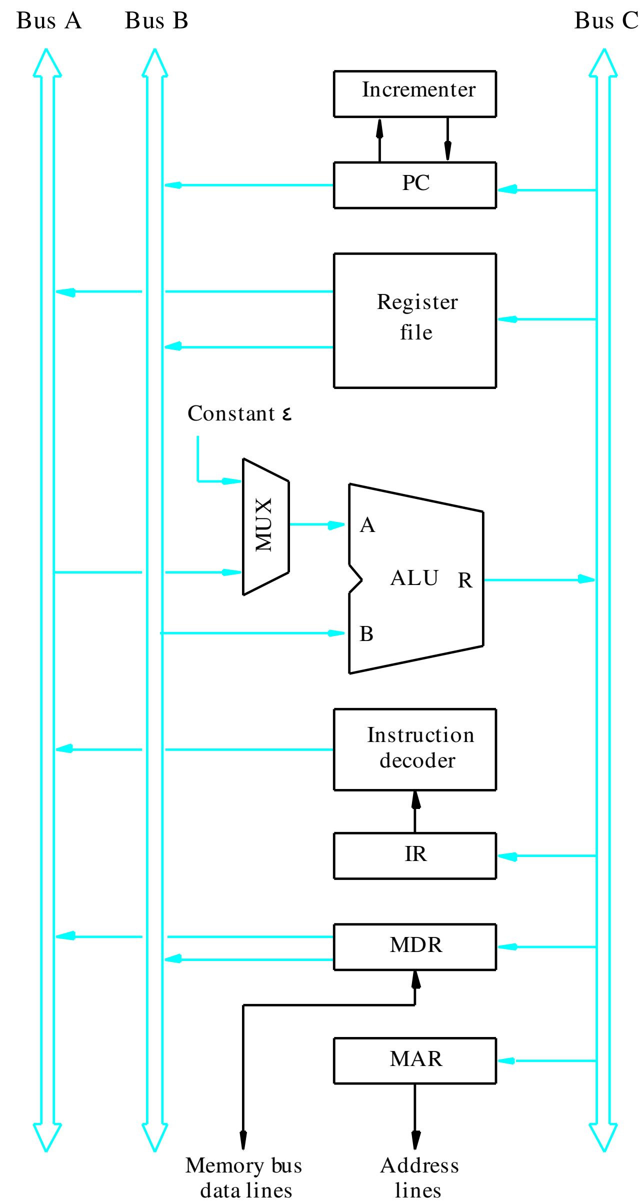
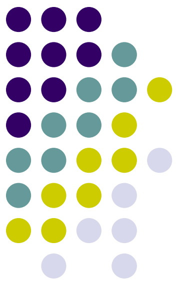
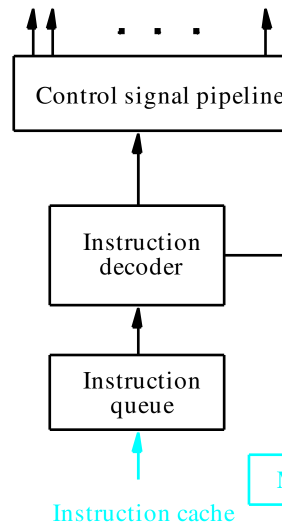


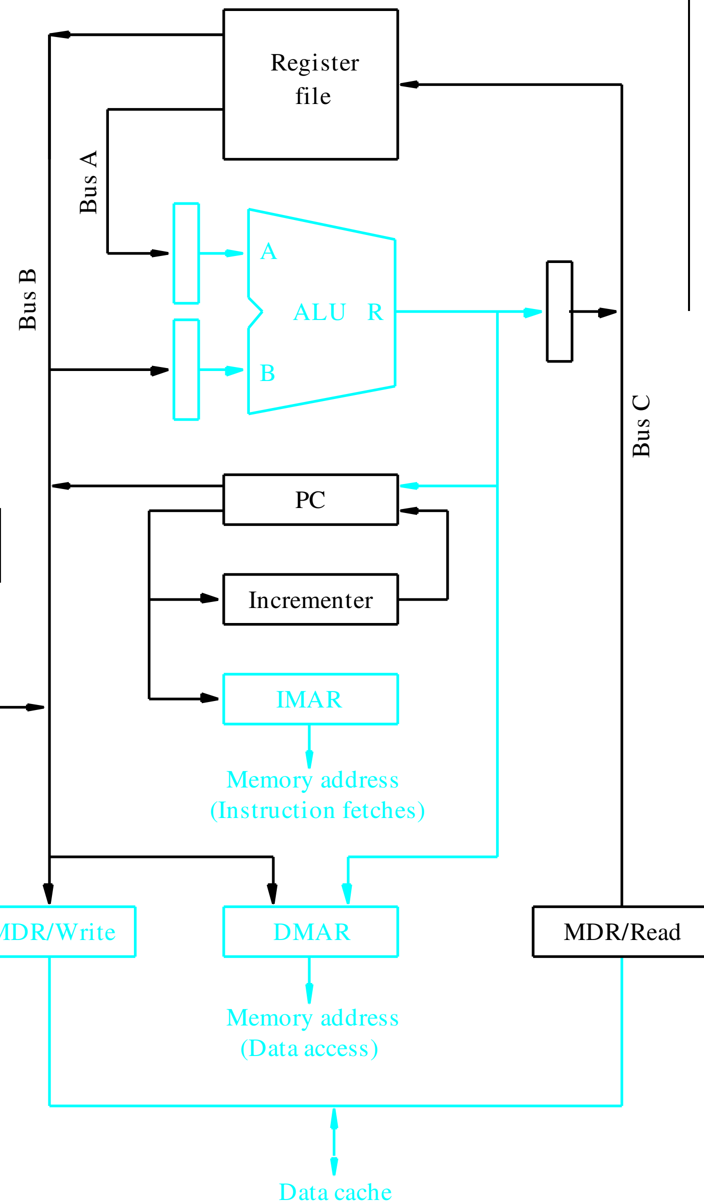
Figure ٧.٨. Three-b us organization of the datapath.

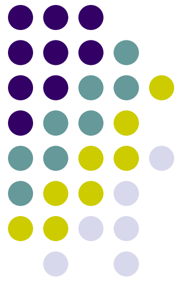


Pipelined Design

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



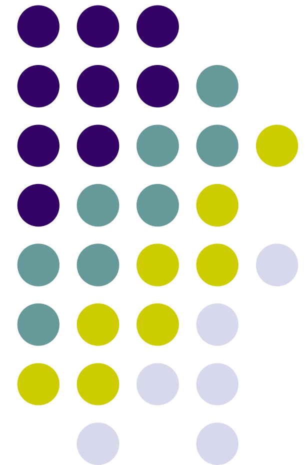




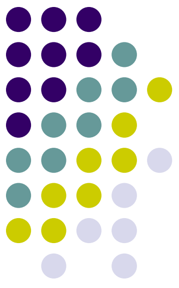
- Reading an instruction from the instruction cache
- Incrementing the PC
- Decoding an instruction
- Reading from or writing into the data cache
- Reading the contents of up to two regs
- Writing into one register in the reg file
- Performing an ALU operation

Figure ۸.۱۸. Datapath modified for pipelined execution, with interstage buffers at the input and output of the ALU.

Superscalar Operation

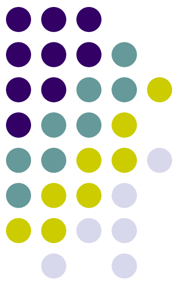


Overview



- The maximum throughput of a pipelined processor is one instruction per clock cycle.
- If we equip the processor with multiple processing units to handle several instructions in parallel in each processing stage, several instructions start execution in the same clock cycle – multiple-issue.
- Processors are capable of achieving an instruction execution throughput of more than one instruction per cycle – superscalar processors.
- Multiple-issue requires a wider path to the cache and multiple execution units.

Superscalar



F : Instruction fetch unit

Instruction queue

Dispatch unit

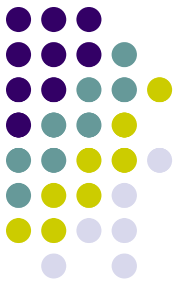
Floating-point unit

Integer unit

W : Write results

Figure ٨.١٩. A processor with two execution units.

Timing



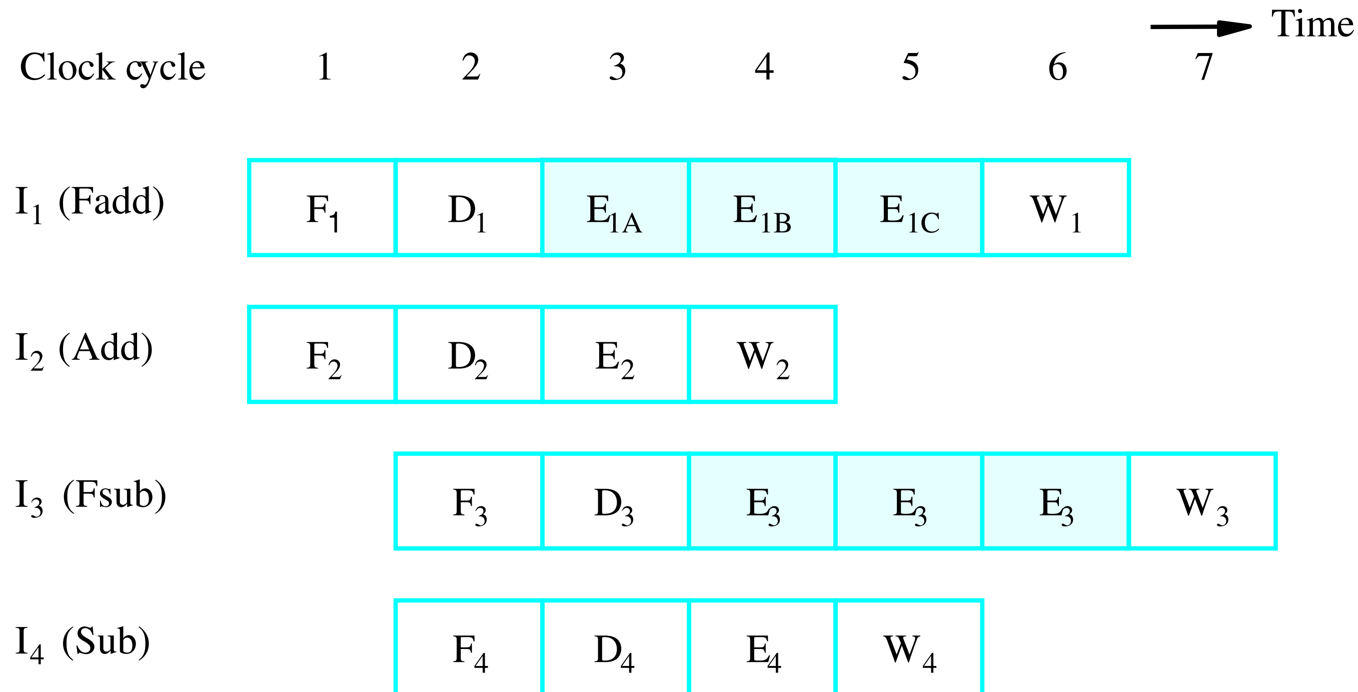
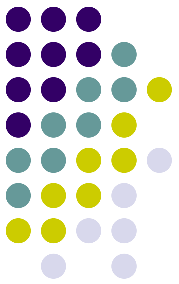
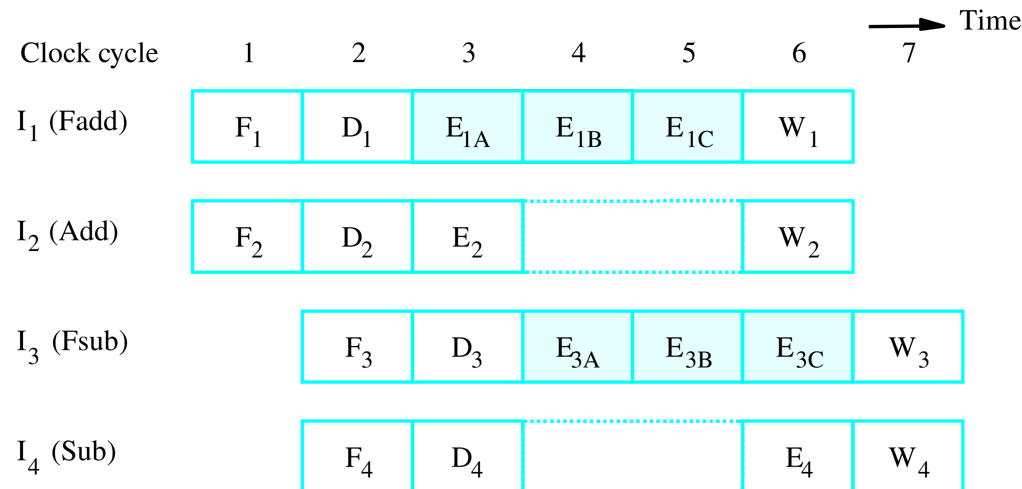


Figure 8.20. An example of instruction execution flow in the processor of Figure 8.19, assuming no hazards are encountered.



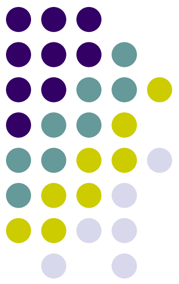
Out-of-Order Execution

- Hazards
- Exceptions
- Imprecise exceptions
- Precise exceptions

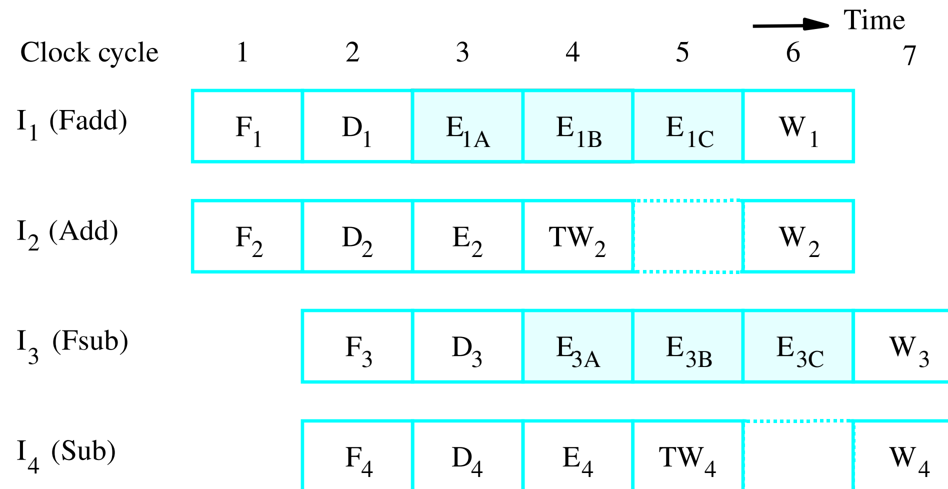


(a) Delayed write

Execution Completion



- It is desirable to used out-of-order execution, so that an execution unit is freed to execute other instructions as soon as possible.
- At the same time, instructions must be completed in program order to allow precise exceptions.
- The use of temporary registers
- Commitment unit



(b) Using temporary registers