Computer Architecture

Basic Design of Processor and Pipeline

Kyusik Chung

kchung@ssu.ac.kr

[Adapted from the lecture notes of Prof. Byung-gi Kim]

강의진도

주#	첫번째시간[사전녹화동영상]	교재내용	두번째시간[대면강의]	교재내용
1	강의1(컴퓨터구조소개)	ch1-part1	강의2(컴퓨터구조소개)	ch1-part2
2	강의3(ARM 프로세서구조)	ch2-part1	강의4(ARM 프로세서구조)	ch2-part1
3	강의5(ARM 프로그래밍)	ch2-part2	강의6(ARM 프로그래밍)	ch2-part2
4	강의7(ARM 프로그래밍예제)	ch2-part3	강의8(ARM 프로그래밍예제)	ch2-part3
5	Lab0(개발환경구축)		강의9(프로세서구조)	ch4-part1
6	Lab1(Factorial함수구현)		강의10(프로세서구조)	ch4-part1
7	Lab2(C pointer및stack overflow)		강의11(프로세서구조)	ch4-part1
8	Lab3(Bubble sorting구현)		중간고사	
9	Lab4(매트릭스곱셈구현)		강의12(프로세서구조)	ch4-part2
10	강의13(프로세서구조)	ch4-part2	강의14(메모리구조)	ch5-part1
11	강의15(메모리구조)	ch5-part1	강의16(메모리구조)	ch5-part1
12	강의17(메모리구조)	ch5-part2	강의18(메모리구조)	ch5-part2
13	강의19(메모리구조)	ch5-part2	강의20(병렬처리)	ch6-part1
14	강의21(병렬처리)	ch6-part2	강의22(병렬처리)	ch6-part2
15	강의23(최종정리)		기말고사	

파란색	사전녹화동영상
빨간색	대면강의
초록색	Lab(사전녹화동영상)

어셈블리프로그래밍 Lab

lab0. 어셈블리프로그래밍 개발환경 구축 및 디버거 사용법 학습

lab1. Factorial 함수를 어셈블리 코드로 구현 및 **test**

lab2. 디버거를 이용한 **C pointer** 동작 및 **stack overflow test**

lab3. Bubble sorting을 위한 어셈블리 코드 분석 및 **test**

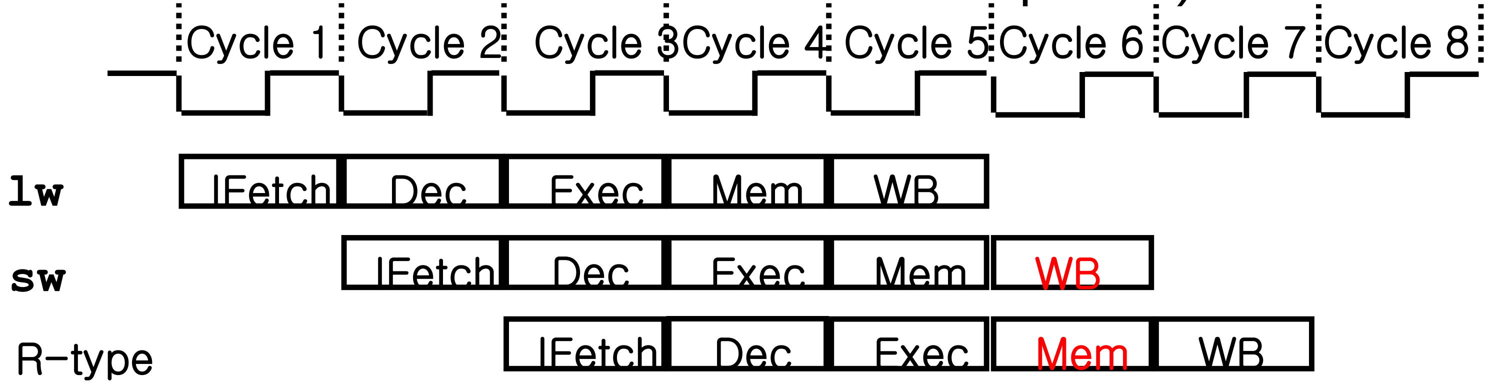
lab4. 매트릭스 곱셈을 어셈블리 코드로 구현 및 **test**

CPU에서의 성능향상

- **C** 코드를 보고 컴파일러가 최적화된 기계어코드를 만들었다고 하자.
  - 기계어 코드 총갯수 최소화/수행도중 메모리 참조(memory read 또는 memory write) 총 횟수 최소화
- **CPU** 입장에서 이 최적화 코드 빠르게 수행할 방법은 없을까 **?**
- (예) **Pipelining** 을 적용하면 약 **4**배가 빨라진다
- (예) 생성된 코드내 기계어(명령어)가 총 **100**개라고 하자
  - 한 명령어는 4단계(fetch-decode-execute-store)에 의해 수행. CPU 클럭 4개 소모
  - CPU가 4 GHZ 클럭을 사용한다고 가정. 하나의 클럭은 0.25 ns 임 (ch1-Part1-P43 참고). 한 명령어 수행시 0.25ns * 4 = 1ns 소모
  - 총 100개 명령어 수행하므로 총 수행시간은 1ns * 100 = 100 ns
  - Pipelining 적용시 100ns 대신 25.75ns에 수행가능. 어떻게 가능할까 ?

[복습]프로세서 성능 개선 기법 (Pipelining) 예

- **Start the next instruction before the current one has completed**
  - improves throughput - total amount of work done in a given time
  - instruction latency (execution time, delay time, response time - time from the start of an instruction to its completion) is *not* reduced



- **clock cycle (pipeline stage time) is limited by the slowest stage**
- **for some stages don't need the whole clock cycle (e.g., WB)**
- **for some instructions, some stages are wasted cycles (i.e., nothing is done during that cycle for that instruction**

Contents

4.1 Introduction

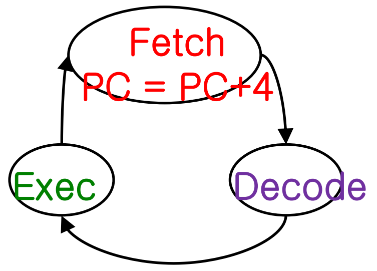
A Basic MIPS Implementation

- **We will examine two MIPS implementations**
  - A simplified version
  - A more realistic pipelined version
- **Simple subset, shows most aspects**
  - Memory-reference instructions(**I** format type: **load/store type**)
    - `lw, sw`
  - Arithmetic-logical instructions (**R format type**)
    - `add, sub, and, or, slt`
  - Branch instructions (**J** format type: **branch type**)
    - `beq, j`

R, I, J type에 대한 추가설명은 2장 Part3 P46 맨왼쪽 하단 참조

An Overview of the Implementation

- **The First 2 Steps of Every Instruction**
  1. The first step
     - Instruction fetch
       - Send PC to memory
       - Send READ signal to memory
     - PC = PC + 4
  2. The second step
     - Opcode decoding
     - Register prefetch



The Third Step

- **Memory-reference instructions**
  - Use ALU for an effective address calculation
- **Arithmetic-logical instructions**
  - Use ALU for the operation execution
- **Branch instructions**
  - Use ALU for comparison

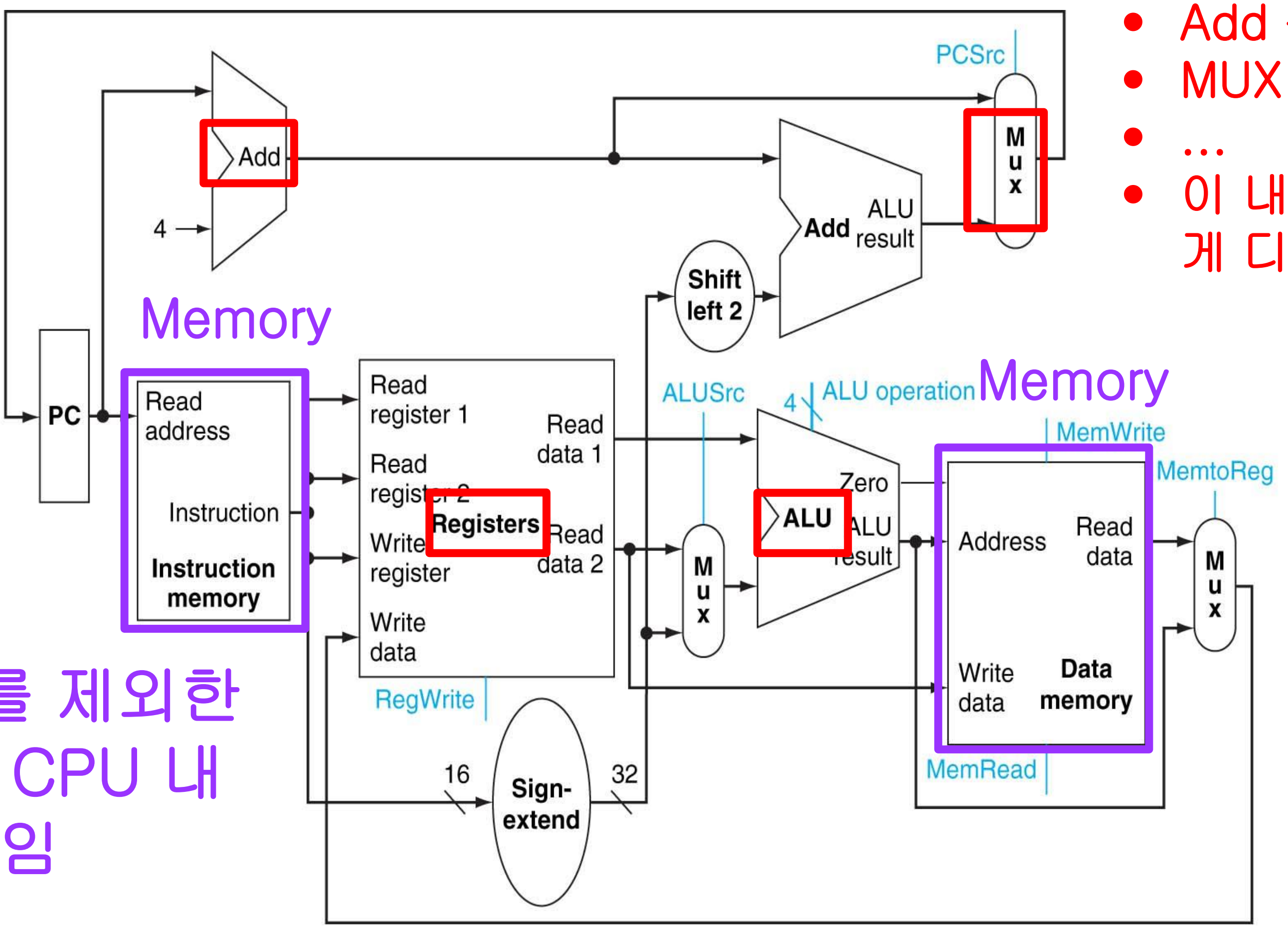
The Final Step

- **Memory-reference instructions**
  - **`Store`**: access the memory to write data
  - **`Load`**: access the memory to read data
- **Arithmetic-logical instructions**
  - Write the data from the ALU back into a register
- **Branch instructions**
  - Change PC based on the comparison

[복습] 1장 Part2 P7, 4장 - 컴퓨터 내부 구조 및 동작원리 이해

- Simple Datapath for MIPS Architecture

- ALU 동작방법 ?
- Register 동작방법?
- Add 동작방법 ?
- MUX 동작방법 ?
- …
- 이 내용을 공부하는 게 디지털회로설계임



Memory를 제외한 나머지는 CPU 내 datapath임

Basic Implementation of the MIPS Subset

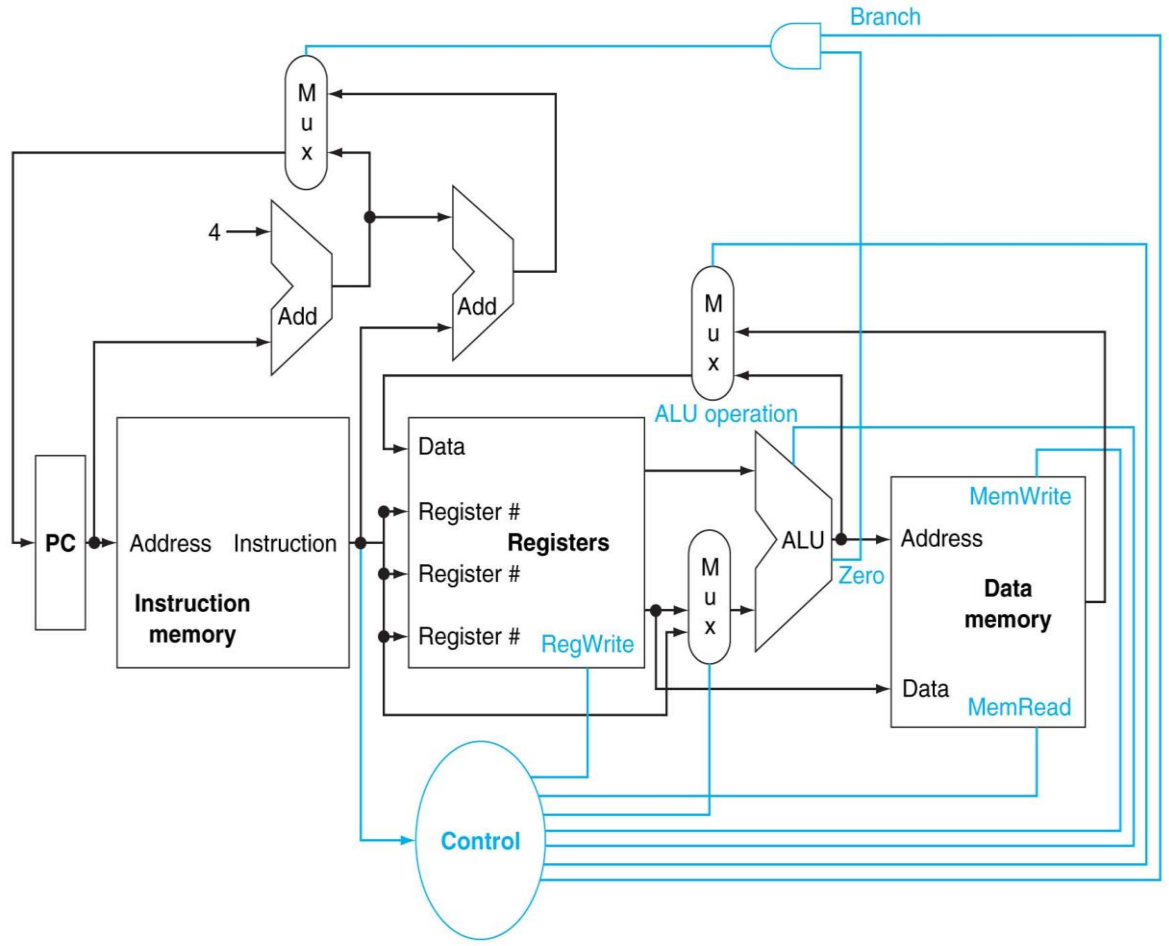


Figure 4.2

4.2 Logic Design Conventions

- **An edge triggered methodology**
- **Typical execution**
  - Read contents of some state elements
  - Send values through some combinational logic
  - Write results to one or more state elements

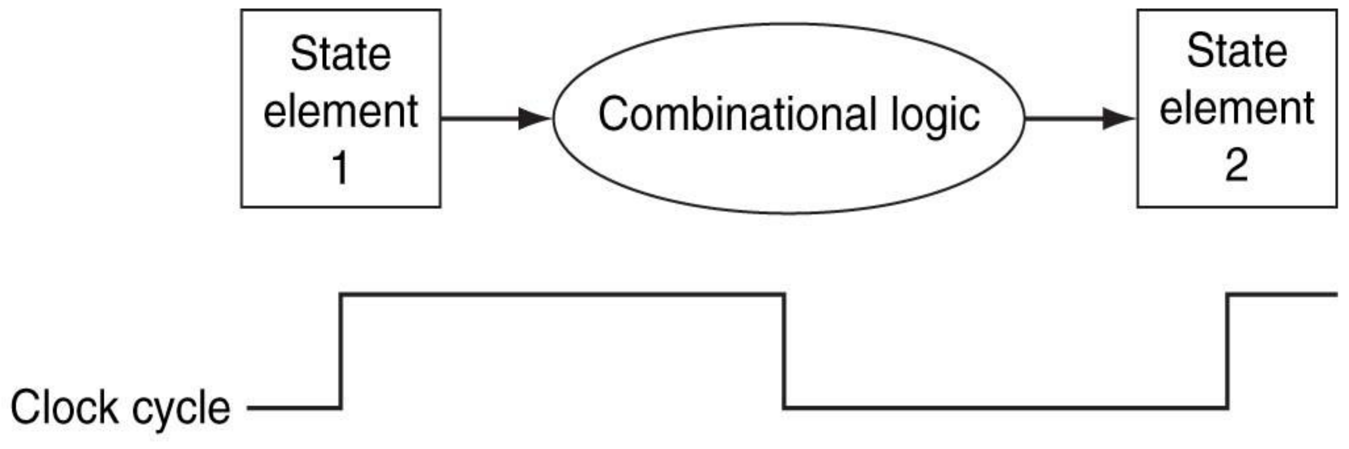
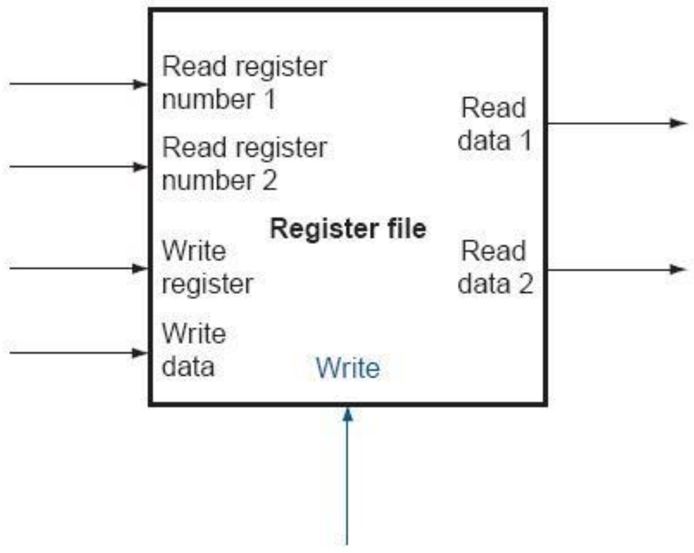


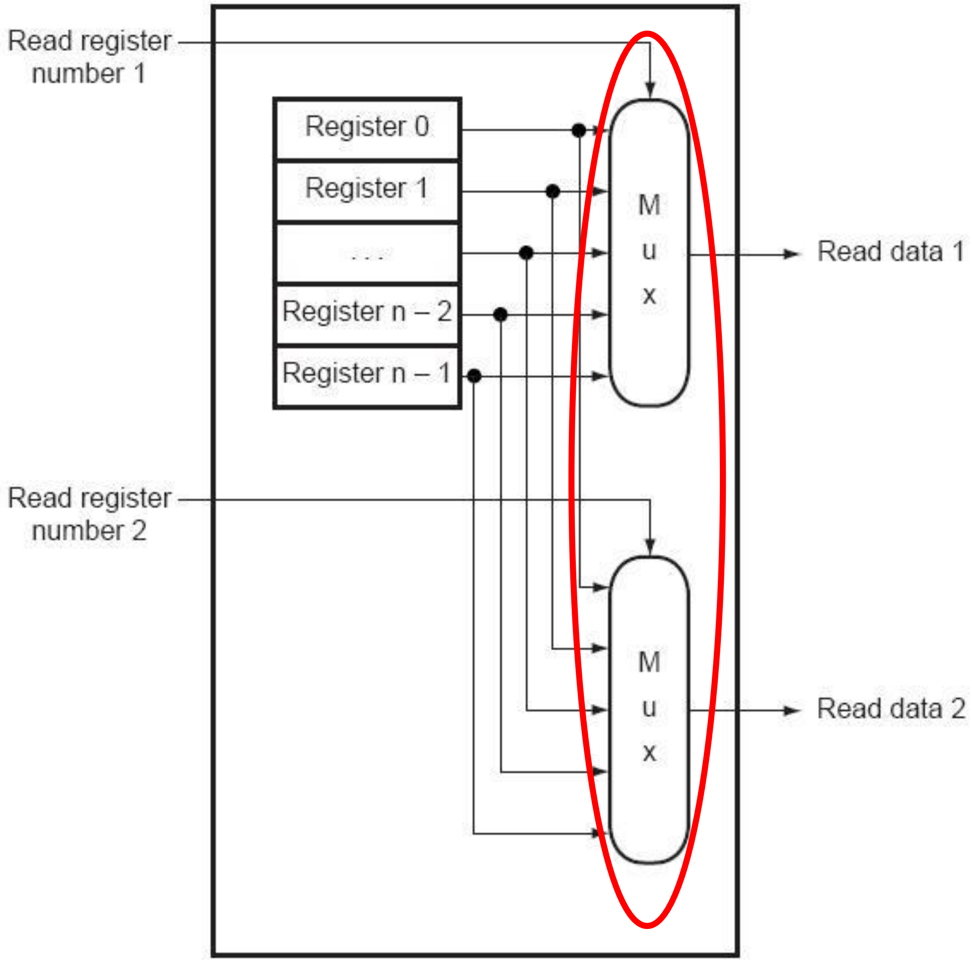
Figure 4.3

Register File - Output

[복습]1장 Part2 P27 참고

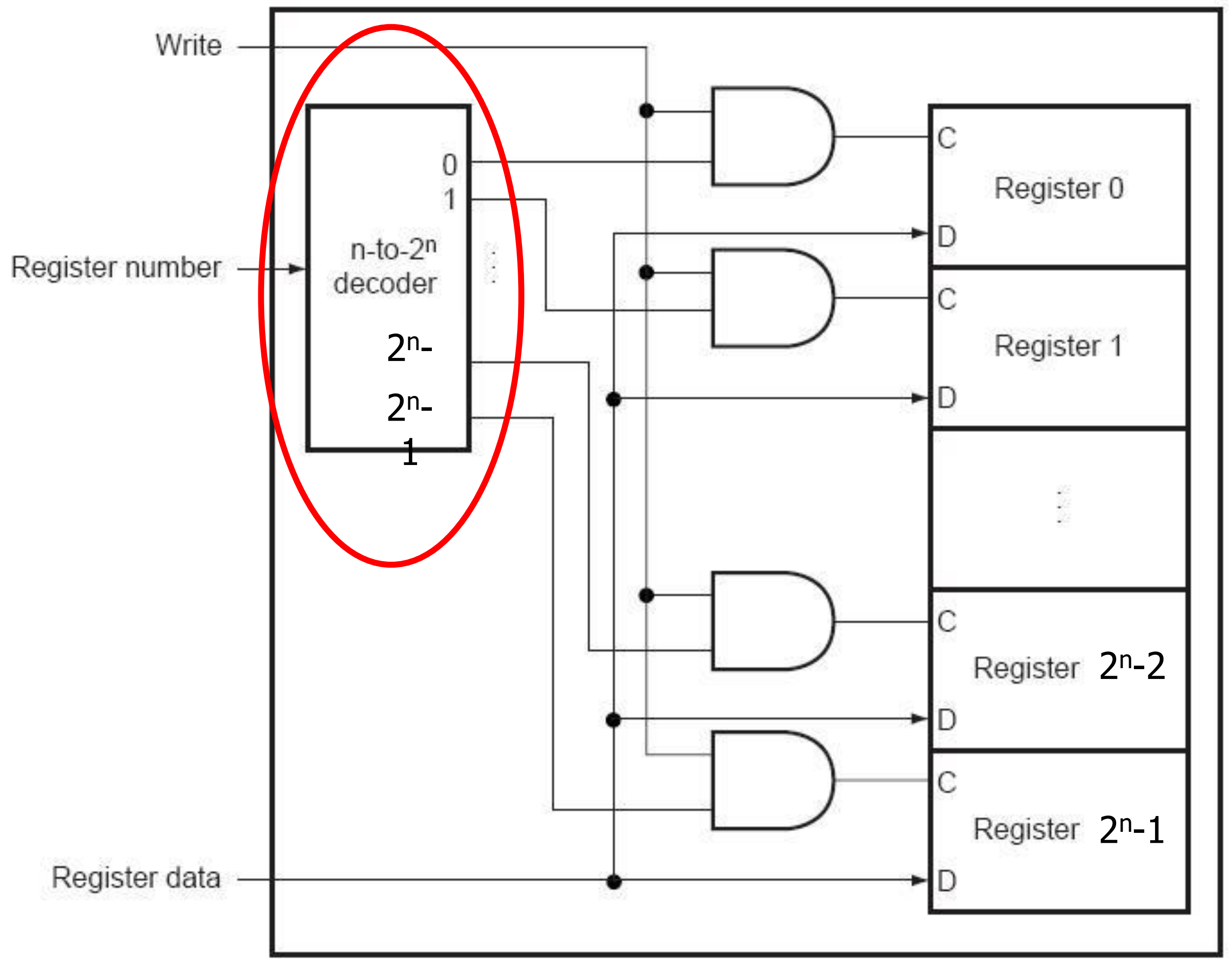
- **Built using D flip-flops**





Register File - Input

[복습]1장 Part2 P27 참고



4.3 Building a Datapath

- **Datapath elements for instruction fetch**
  - Instruction memory: Store the instructions of a program
  - Program Counter (PC): Store the address of the instruction
  - Adder: Increment the PC to the address of the next instruction

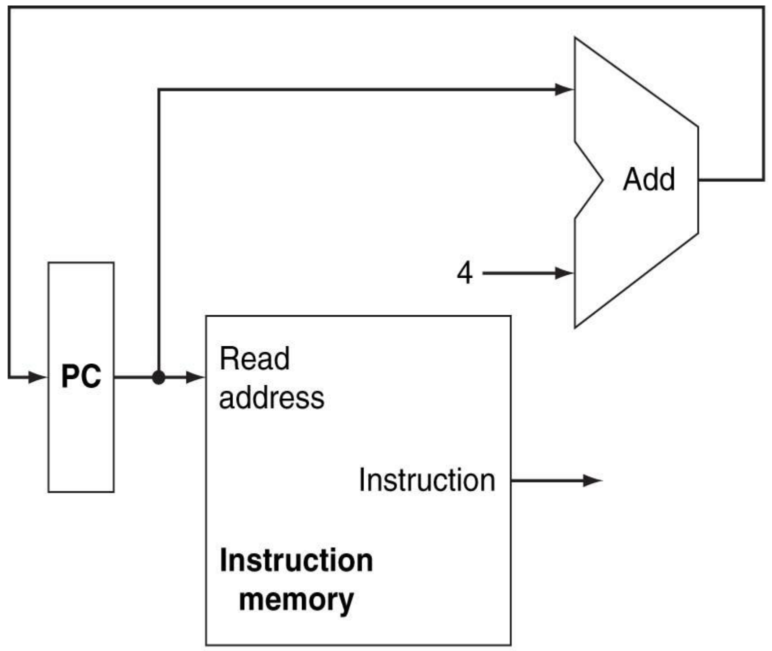


Figure 4.6

Executing R-format Instructions

- **Step 3**

  Use ALU for the opcode execution
- **Step 4**

  Write the data from the ALU back into a register
- **Major components**

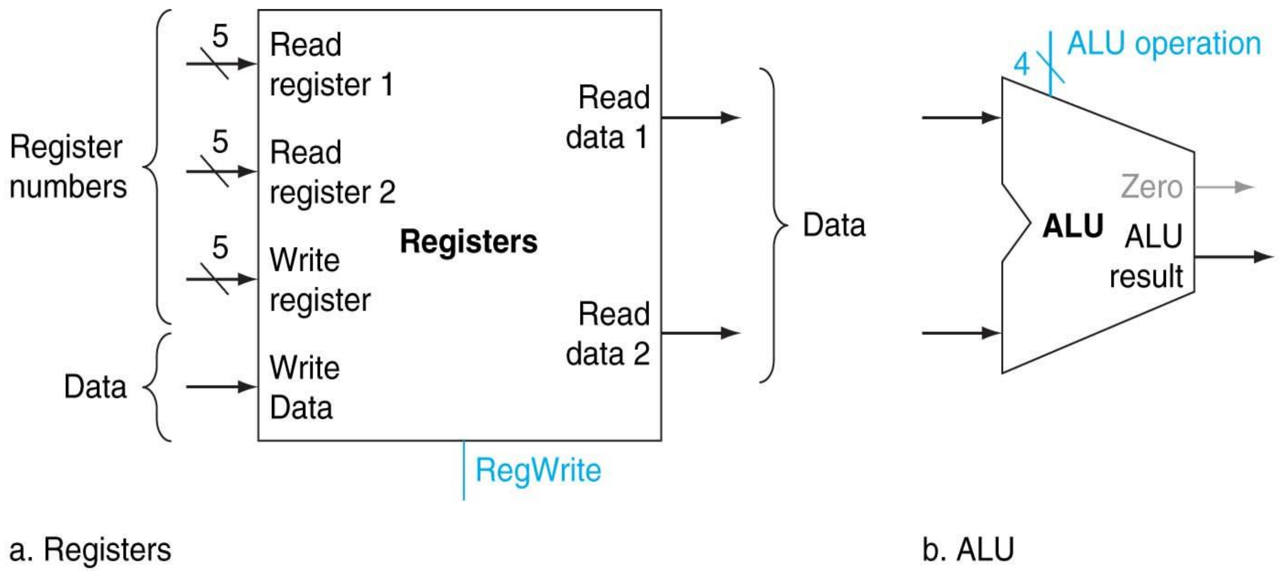
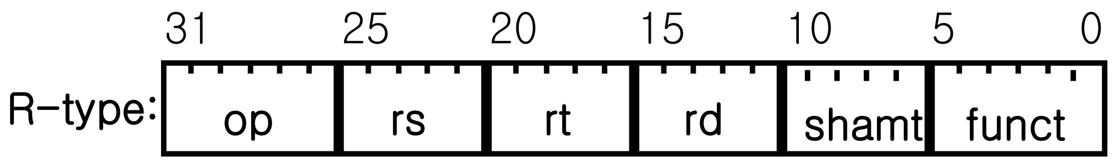
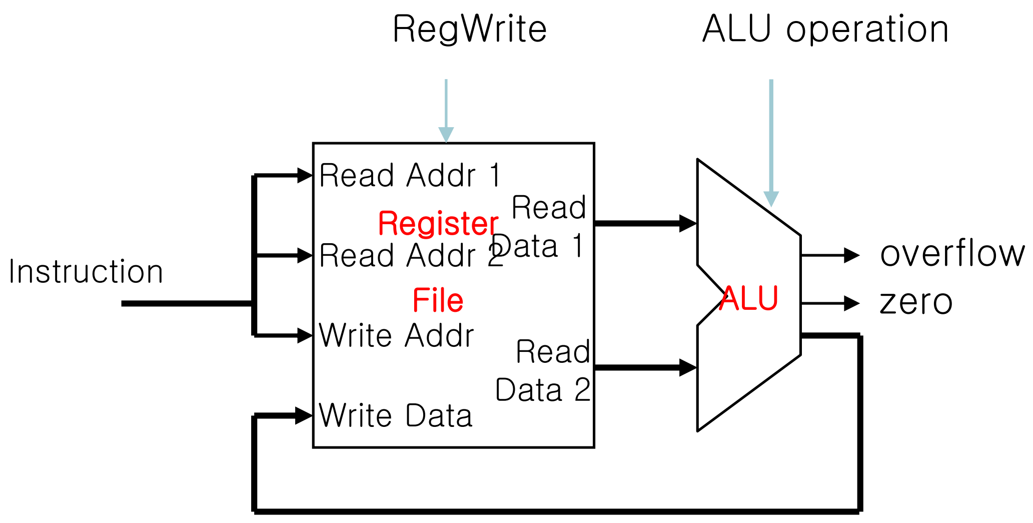


Figure 4.7

Datapath for R-format Instructions





- Note that Register File is not written every cycle (e.g. **`sw`**), so we need an explicit write control signal for the Register File

Executing Memory Reference Instructions

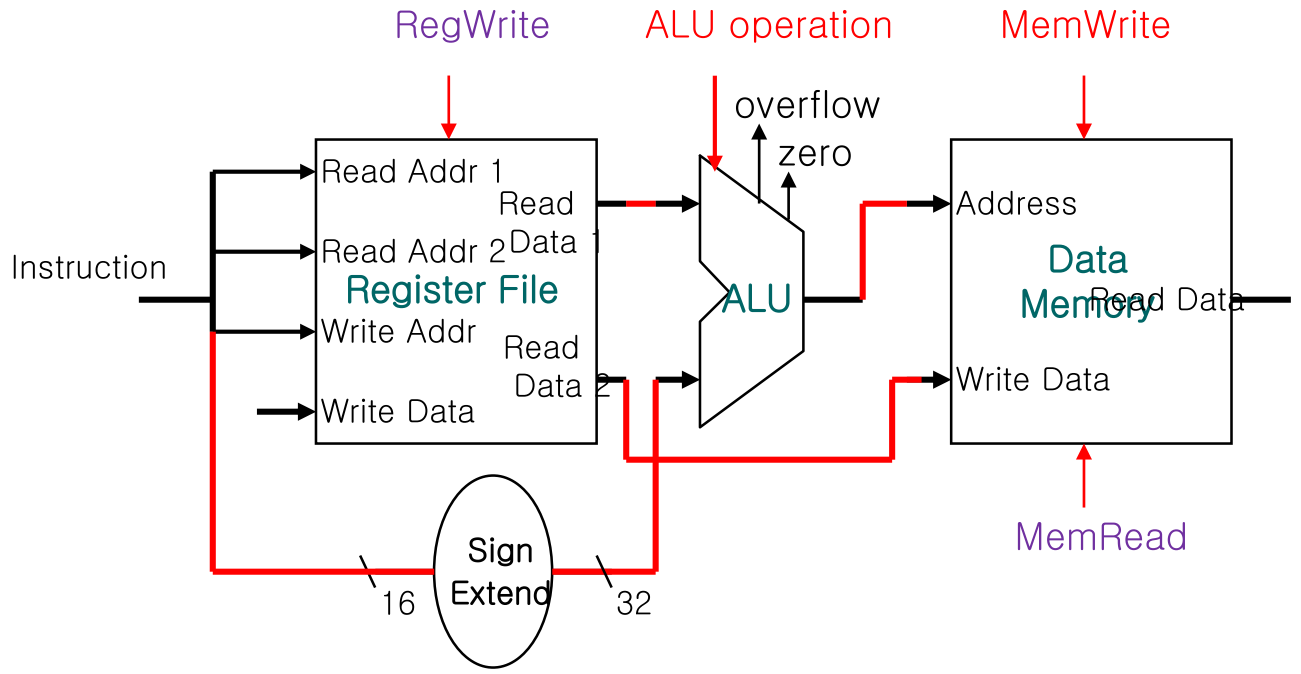
- **Step 3**

  Compute a memory address by adding the base register to the sign-extended 16-bit offset

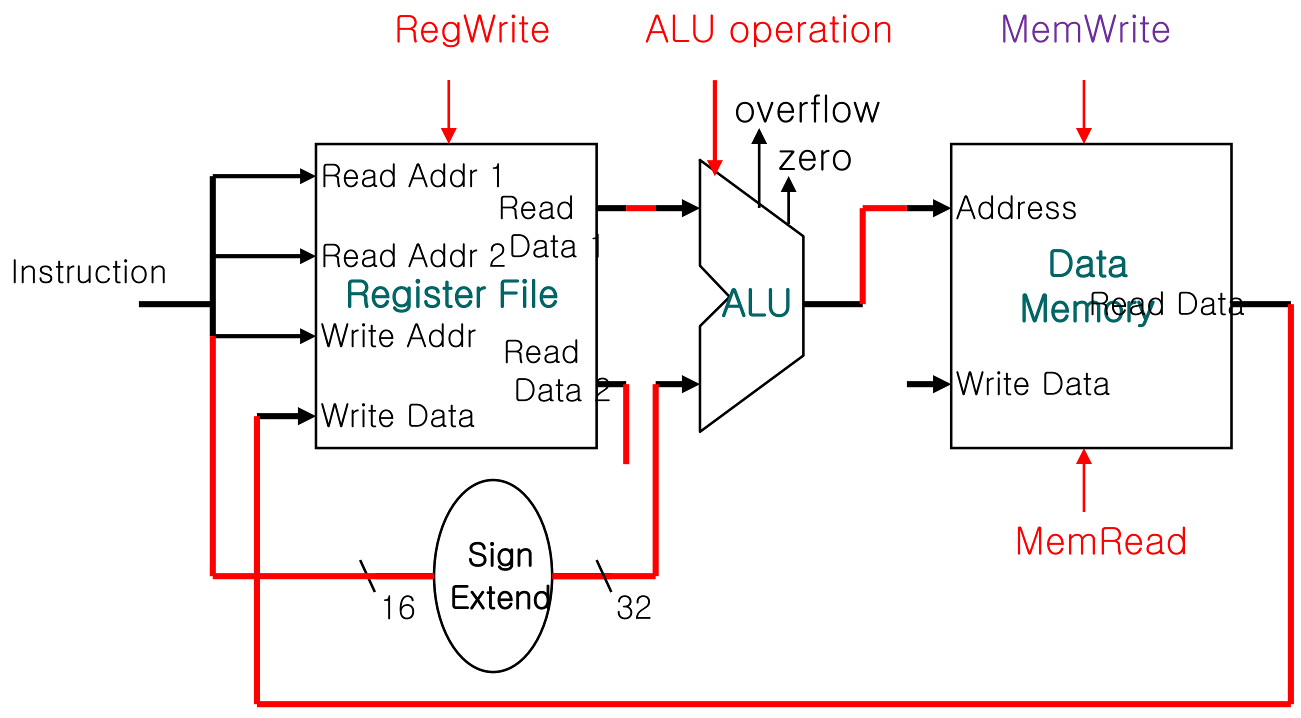
- **Step 4**
  - **`sw $9,offset_value($10)`**
    - Write **`$9`** into memory
  - **`lw $9,offset_value($10)`**
    - Read a value from memory
    - Write it into **`$9`**

- **Major components**
  - Sign extension unit and data memory

Datapath for store Instructions



Datapath for load Instructions



Executing Branch Instructions

- **Step 3**

  Use ALU for comparison

  Compute branch target address by adding the sign-extended offset to the PC
- **Step 4**

  Change PC based on the comparison

- **Major components**
  - Shift-left-2 unit
  - Separate adder

Datapath for Branch Instruction

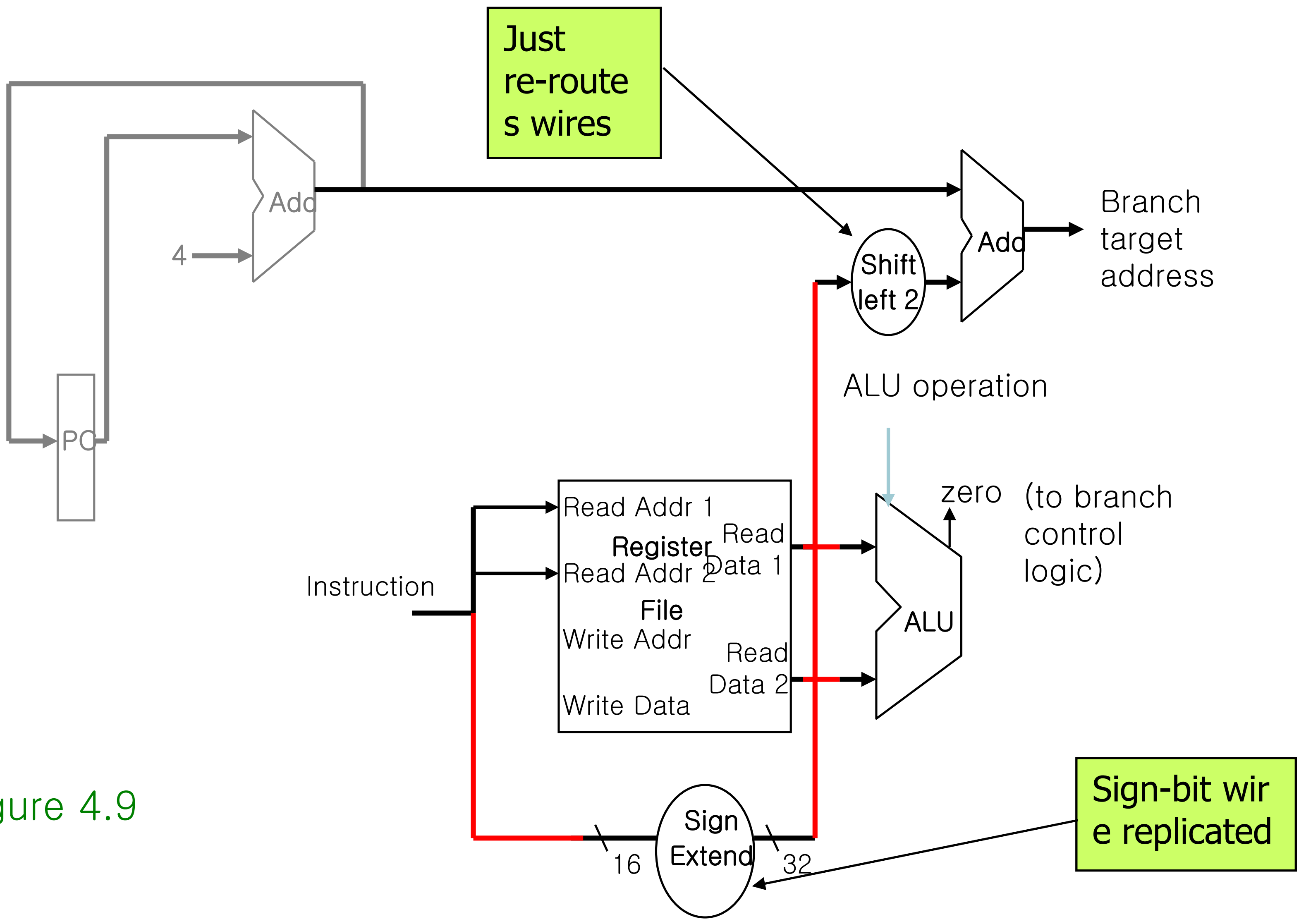
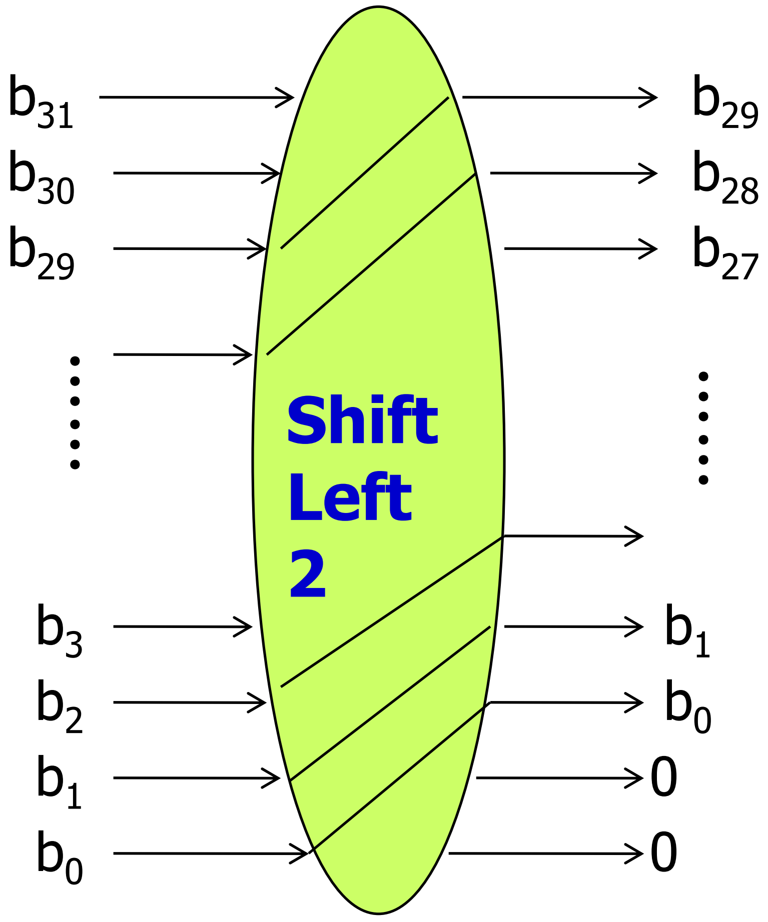
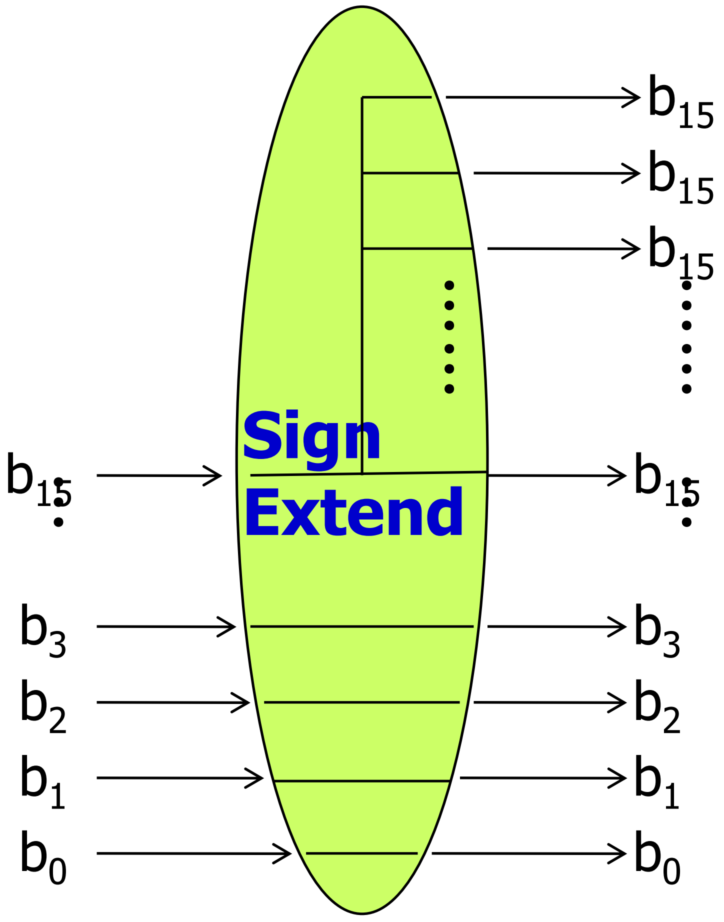


Figure 4.9

Units Implementation





Creating a Single Datapath

- **The simplest datapath**
  - Single-cycle datapath (↔ Multi-cycle datapath)
    - Execute all instructions in one clock cycle
    - No datapath resource can be used more than once per instruction.
    - Any element needed more than once must be duplicated.
    - Separate instruction and data memories
  - Sharing a datapath between two different instruction types
    - Use multiplexer

Example: Building a Datapath

- **Combine the arithmetic-logical instruction datapath and the memory instruction datapath.**

[Answer]

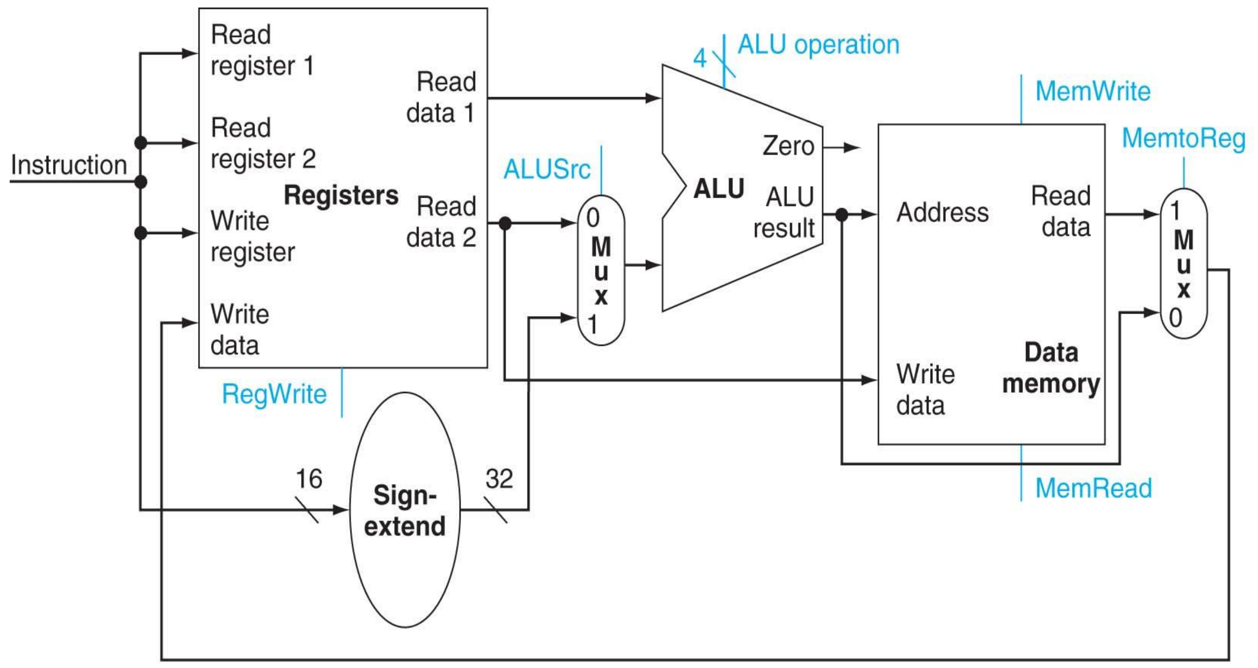


Figure 4.10

Simple Datapath for MIPS Architecture

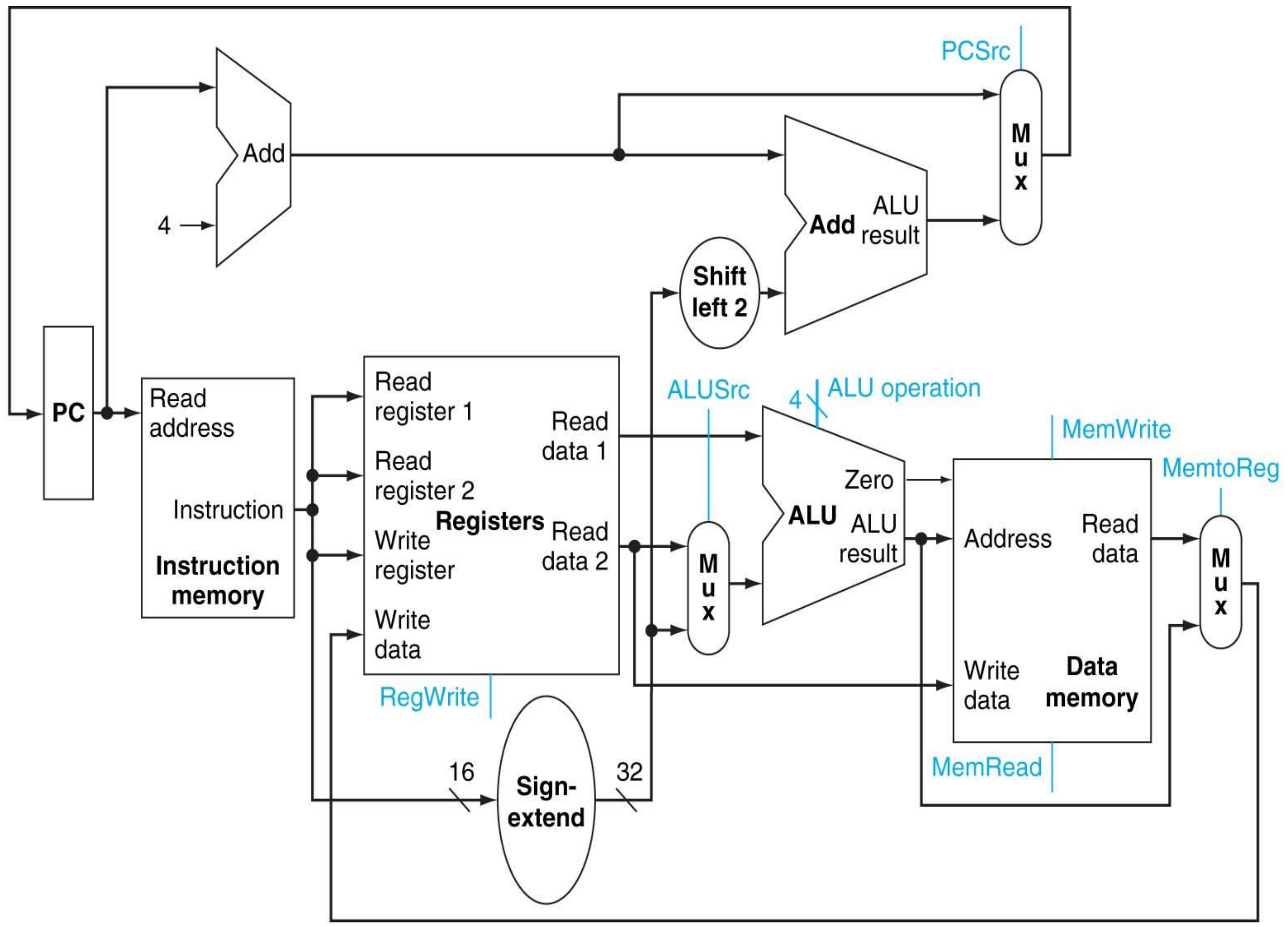


Figure 4.1

4.4 A Simple Implementation Scheme (Control Unit)

The ALU Control

ALU에서 덧셈, 뺄셈, ... 중 어느 기능을 수행할지 제어하는 단자

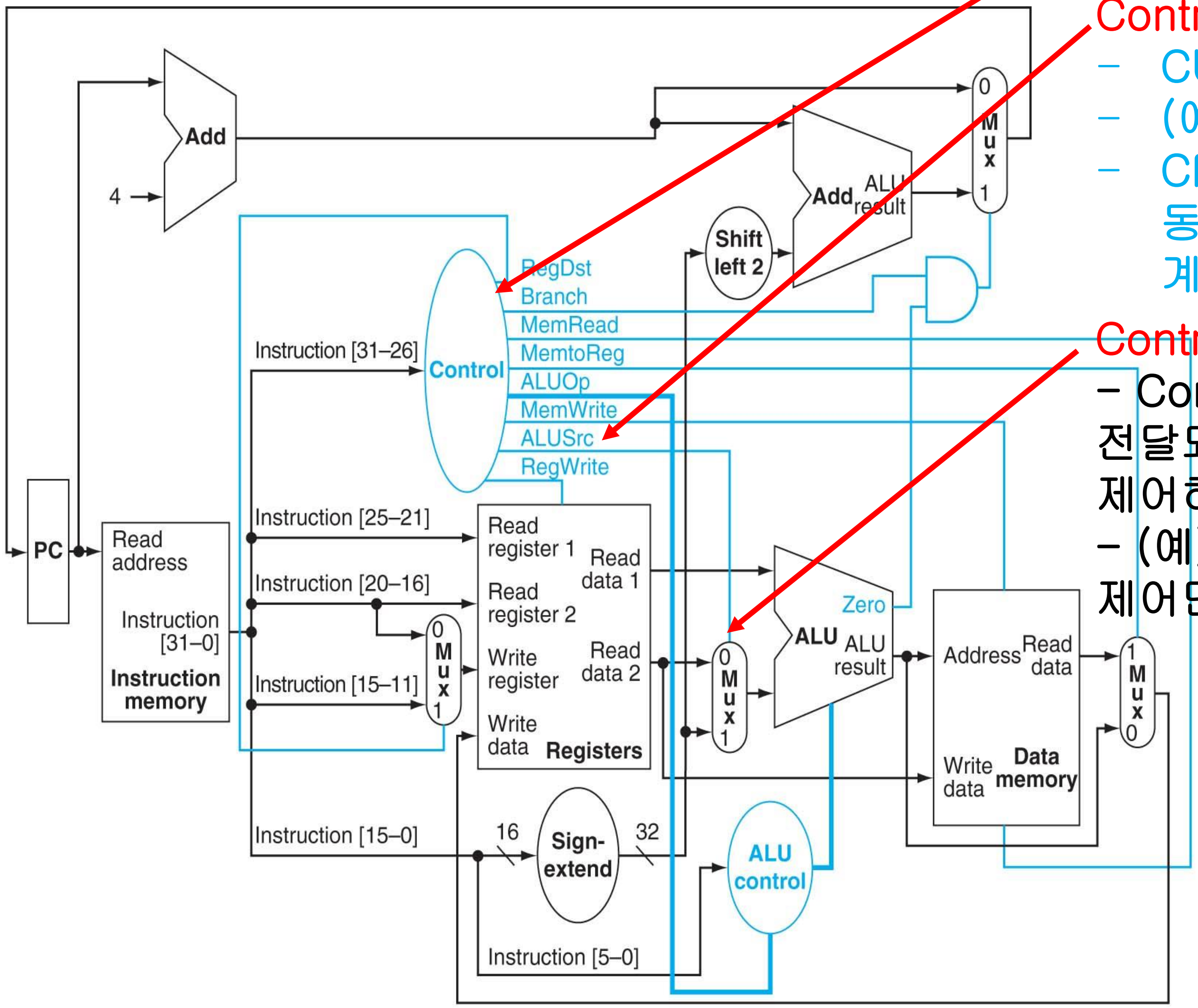
Instruction opcode	ALU Op	Instruction operation	Function field	Desired ALU action	ALU operation
lw	00	load word	XXXXXX	add	0010
sw	00	store word	XXXXXX	add	0010
beq	01	branch on equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	and	0000
R-type	10	OR	100101	or	0001
R-type	10	set on less than	101010	set on less than	0111

ALU를 사용하는 명령어중 R-type 명령어, load & store 명령어, branch 명령어중 어느 type인지 지정

R-type 명령어 경우 ALU에서 하는 동작의 종류를 지정

Figure 4.12

Main Control and ALU Control



Control unit(CU)

Control signal

- CU의 output
- (예) ALUSrc
- CPU 한 클럭 동안 0 또는 1을 계속 유지함

Control point

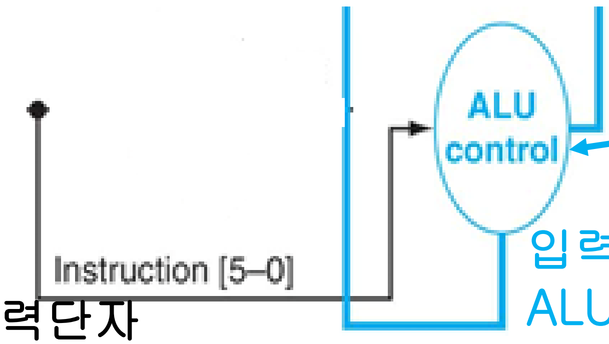
– Control signal이 전달되어 회로를 제어하게 되는 부분

– (예) MUX의 제어단자

Figure 4.17

Truth Table for the ALU Control Bits

앞장 그림 하단부분



출력단자
ALU Operation

ALU control 설계 ?

입력단자
ALUOp1, ALUOp2

입력단자
F5, F4, F3, F2, F1, F0
(P32의 R-type 맨마지막
funct 5:0 에 해당)

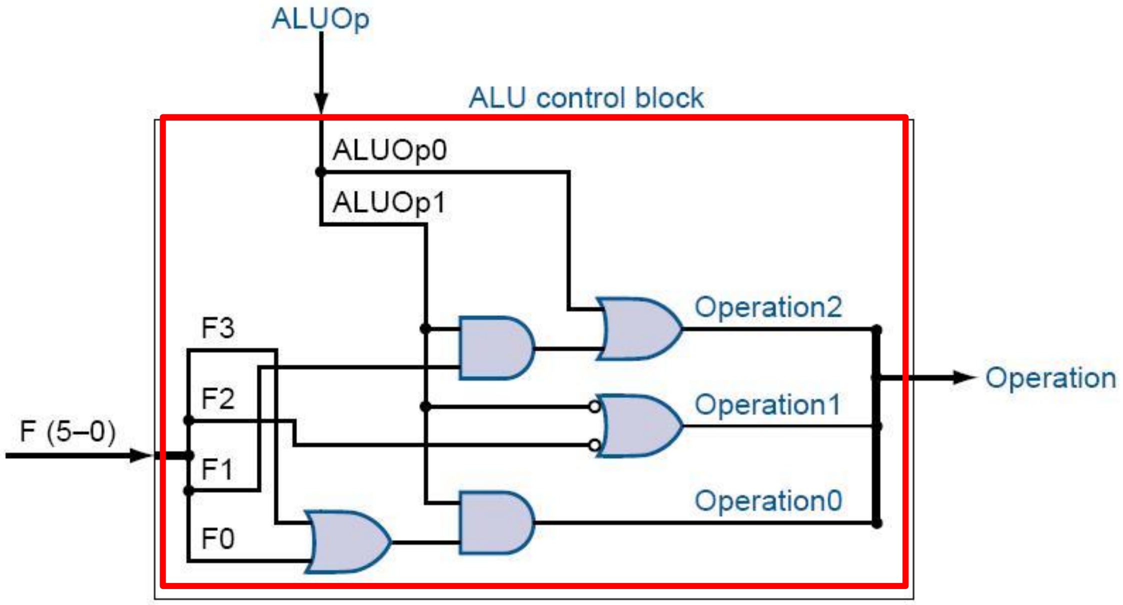
Operation3=0

ALUOp		Function field						ALU Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	x	x	x	x	x	x	0 0 1 0
x	1	x	x	x	x	x	x	0 1 1 0
1	x	x	x	0	0	0	0	0 0 1 0
1	x	x	x	0	0	1	0	0 1 1 0
1	x	x	x	0	1	0	0	0 0 0 0
1	x	x	x	0	1	0	1	0 0 0 1
1	x	x	x	1	0	1	0	0 1 1 1

Operation2=ALUOp0 + ALUOp1 · F1

Implementation of ALU Control

Operation3 = 0
Operation2 = ALUOp0 + ALUOp1 · F1
Operation1 = ALUOp1' + F2'
Operation0 = ALUOp1 · (F0 + F3)



Designing the Main Control Unit

- Control signals derived from instruction

현재 수행중인 명령어코드를 보고 CU는 control signal 값(0또는 1)들을 결정

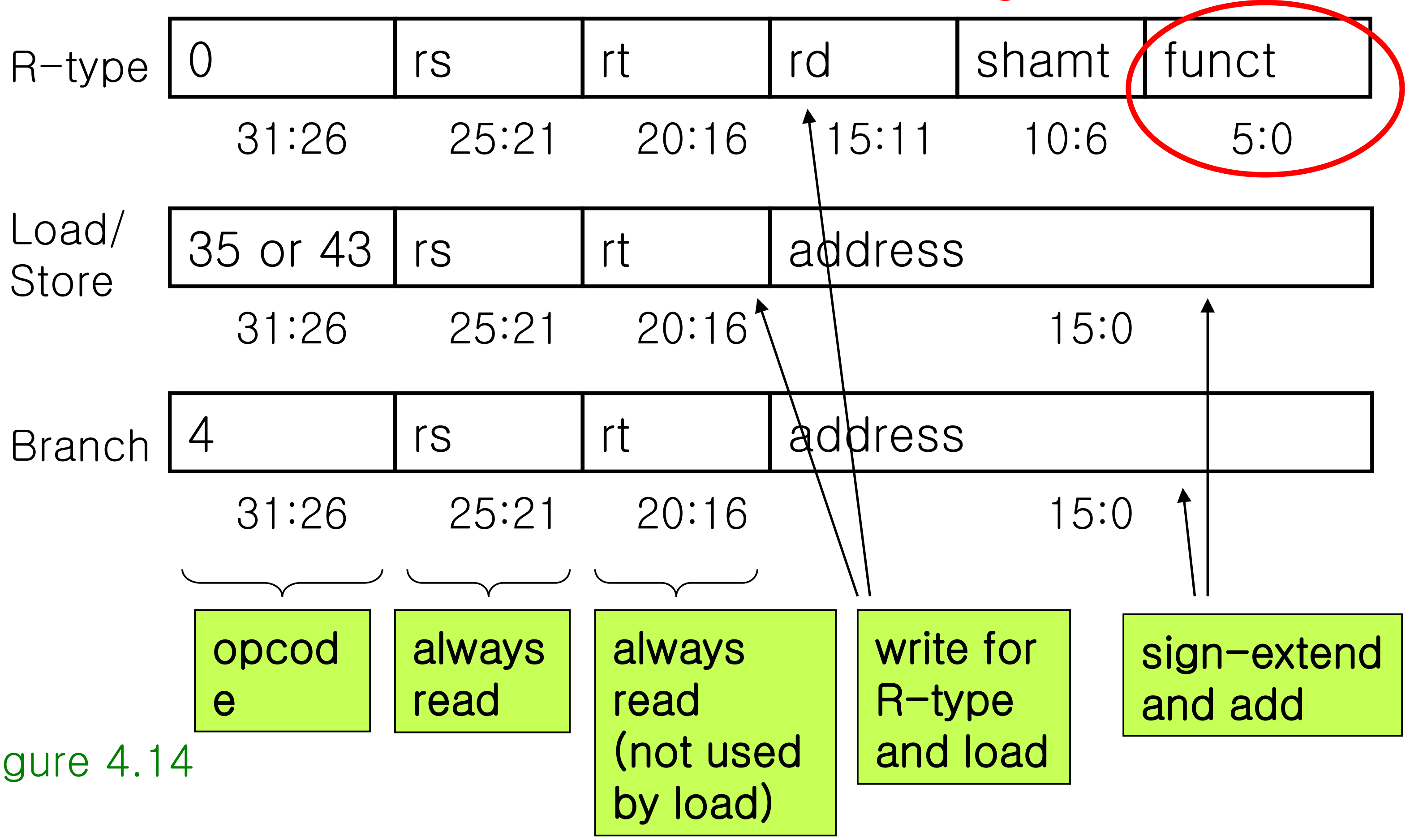


Figure 4.14

Datapath with Control Lines

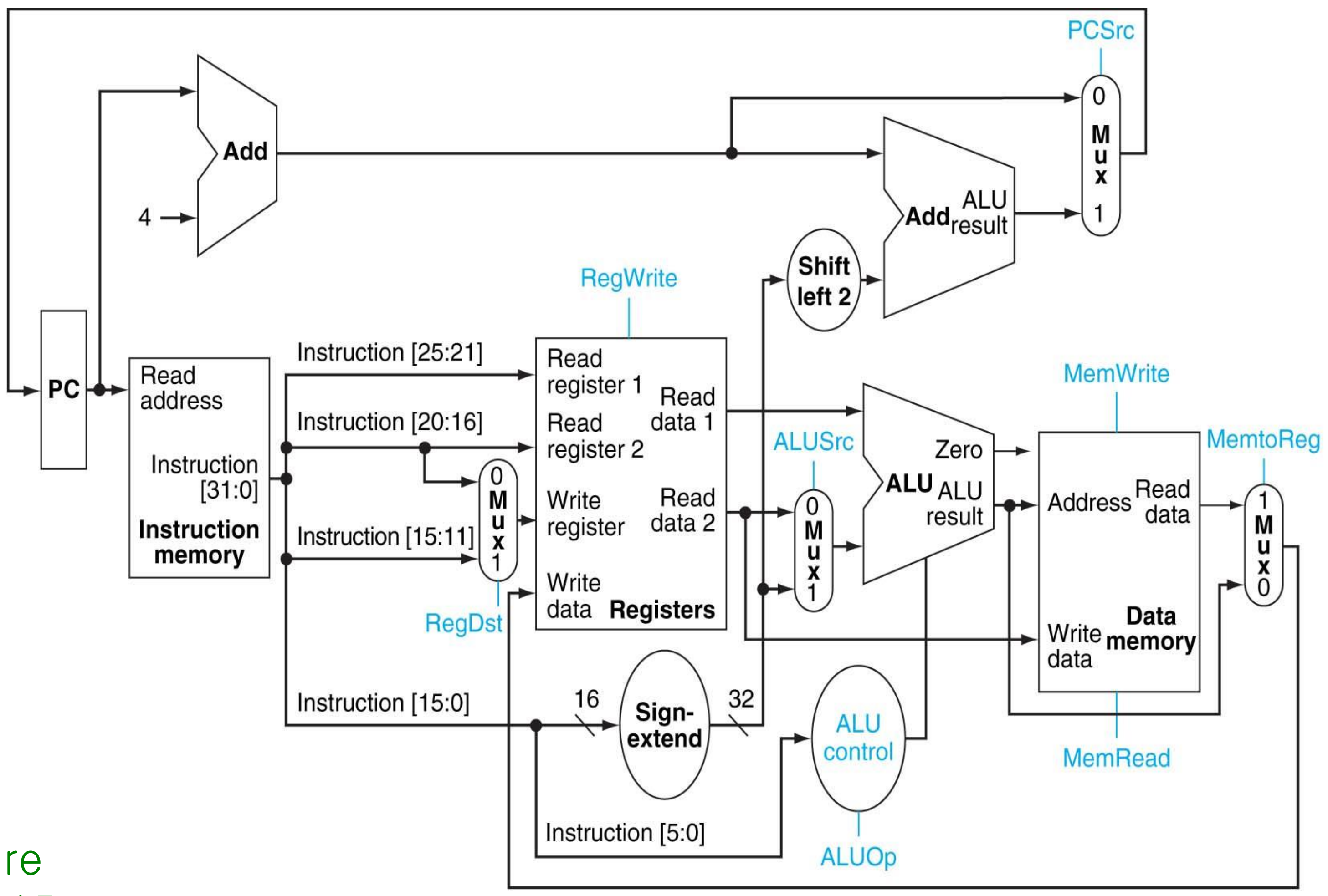


Figure 4.15

The Function of the Control Signals

Signal name	Effect when deasserted	Effect when asserted
RegDst	The register destination number for the Write register comes from the rt field (bits 20:16).	The register destination number for the Write register comes from the rd field (bits 15:11).
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, lower 16 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

Figure 4.16

Control Unit

- **PCSrc = Branch · Zero (from ALU)**

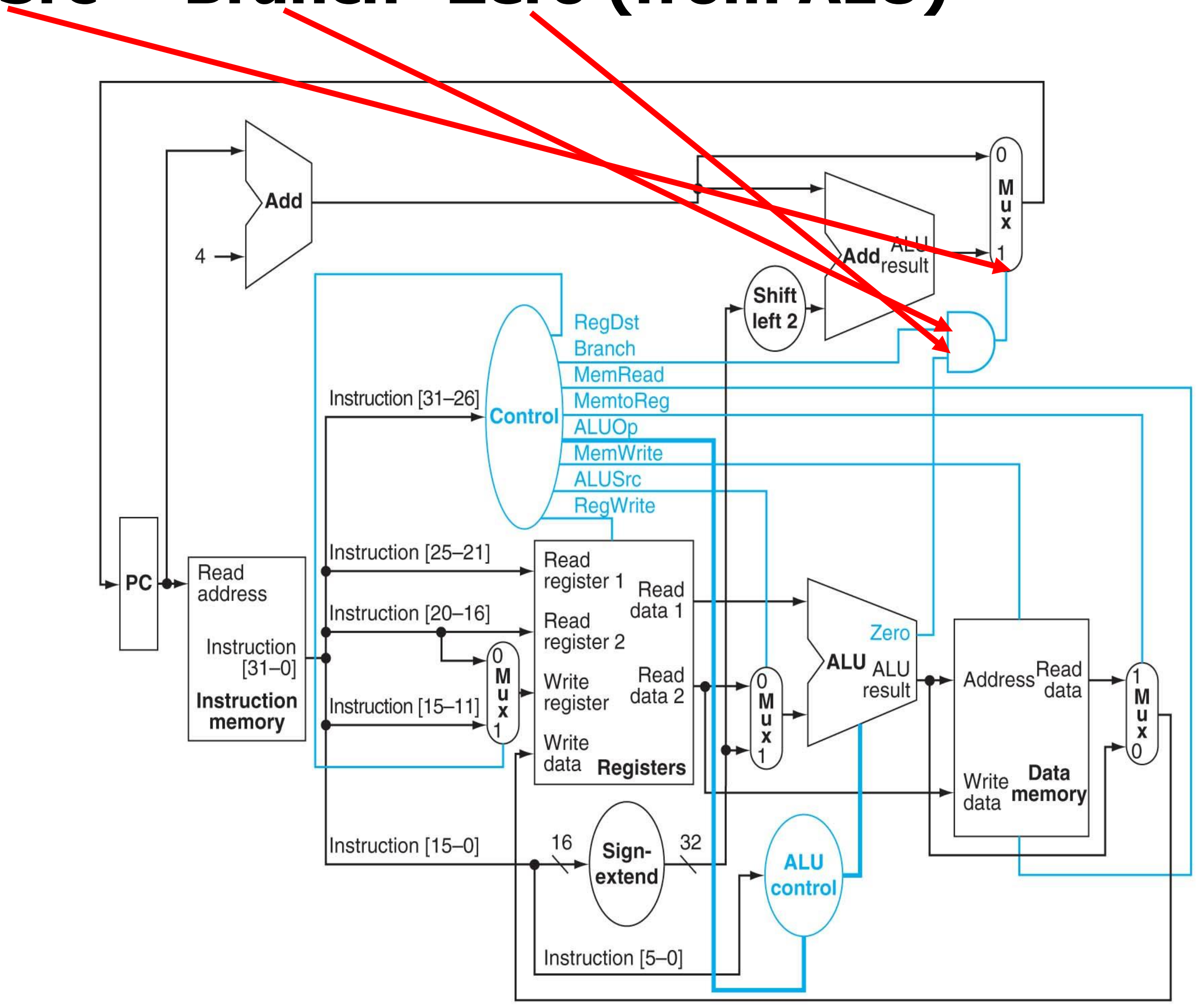


Figure 4.17

Execution of an R-type Instruction

- e.g. `add $t1,$t2,$t3`

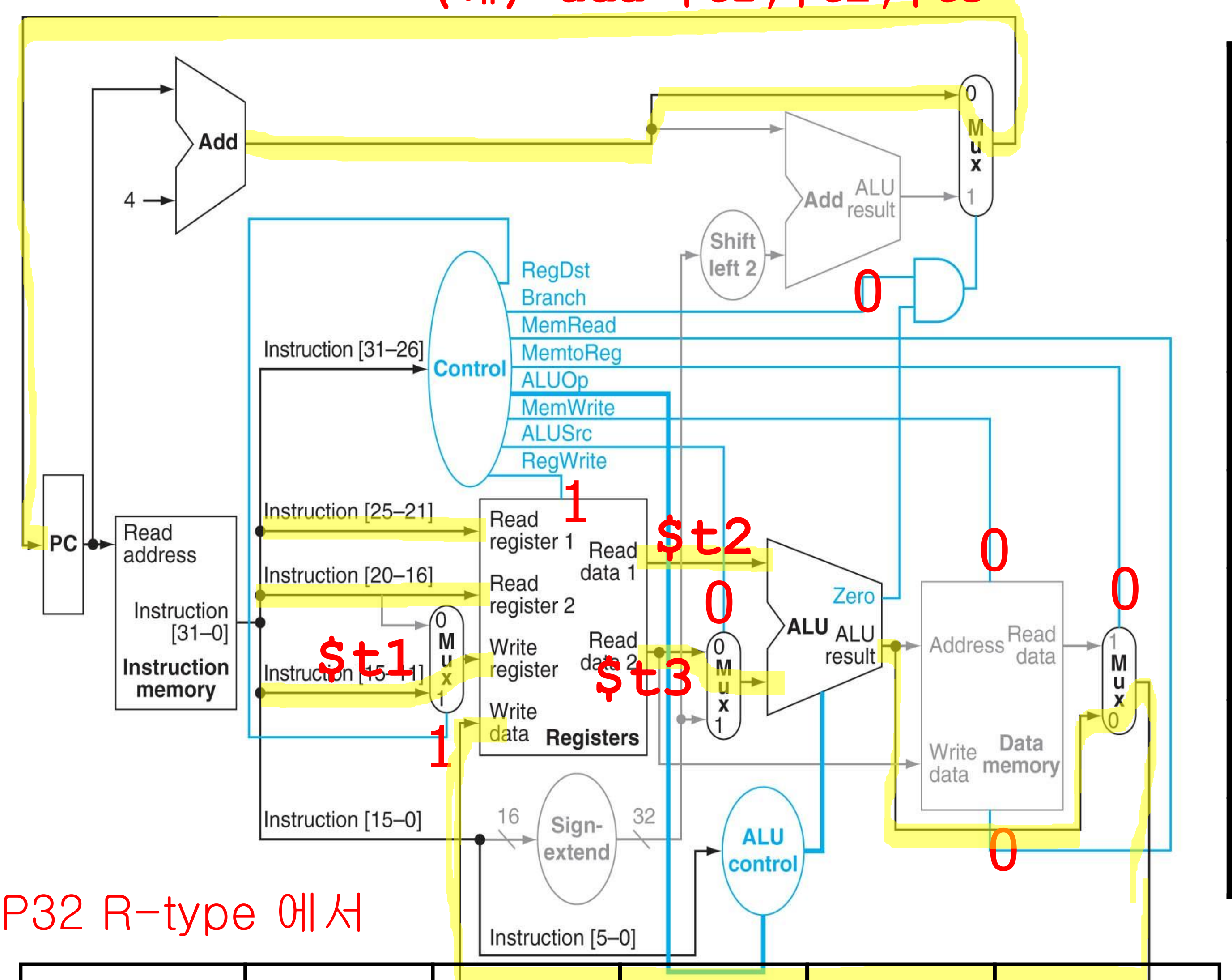
1. Fetch instruction and increment PC.
2. Read two registers (`$t2` and `$t3`) and generate control signals in the main control unit.
3. ALU operates on the data, using the function code.
4. ALU result is written into `$t1`.

P28에서 R-type
add 명령어의 경우
funct =100000

0	rs	rt	rd	shamt	funct
31:26	25:21	20:16	15:11	10:6	5:0

Datapath for an R-type Instruction

(예) add $t1,$t2,$t3



P32 R-type 에서

RegDst	1
ALUSrc	0
MemtoReg	0
RegWrite	1
MemRead	0
MemWrite	0
Branch	0
ALUOp	10

Figure 4.19

0	rs	rt	rd	shamt	funct
31:26	25:21	20:16	15:11	10:6	5:0

Finalizing the Control

Input		Output									
mnemonic	opcode	RegDst	ALUSrc	MemetoReg	RegWrite	memRead	MemWrite	Branch	ALUOp1	ALUOp2	Jump
R-type	000000	1	0	0	1	0	0	0	1	0	0
`lw`	100011	0	1	1	1	1	0	0	0	0	0
`sw`	101011	X	1	X	0	0	1	0	0	0	0
`beq`	000100	X	0	X	0	0	0	1	0	1	0
`j`	000010	X	X	X	0	0	0	X	X	X	1
`addi`	001000										

Modified Figure 4.22

Implementation of the Control

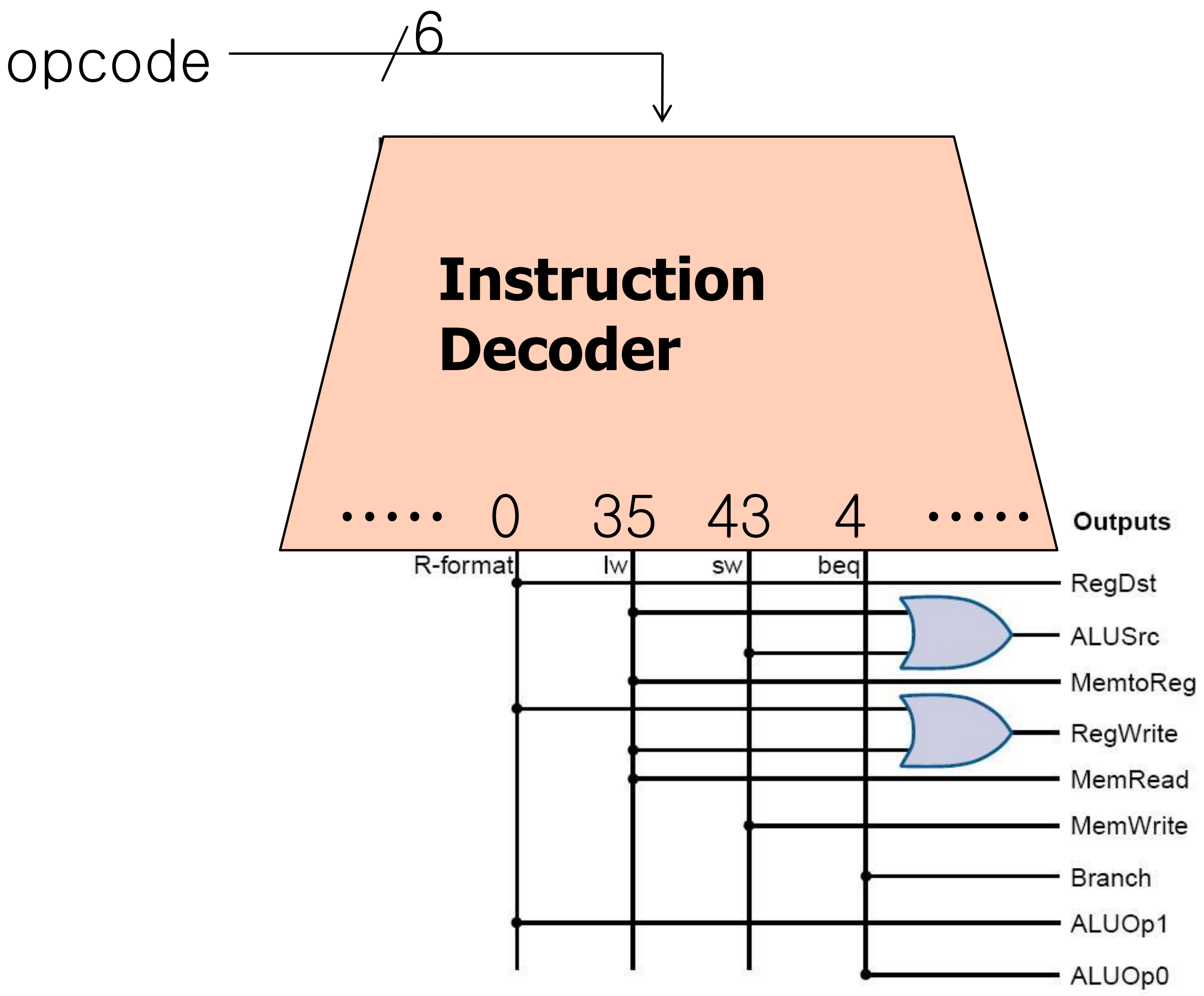


Figure C.2.5

Why a Single-Cycle Implementation Is Not Used Today

- CPI = 1
- Clock cycle is determined by the longest possible path
- Critical path
  - Load instruction
  - instruction memory → register file → ALU → data memory → register file
- Not feasible to vary period for different instructions
- Violates design principle
  - Making the common case fast
- Performance can be improved by pipelining

Problems with a Single-Cycle Implementation

- Too long a clock cycle with floating-point unit or complex instruction set
- Implementation techniques that reduce common case delay but do not reduce worst-case cycle time cannot be used.
- Some functional units must be duplicated.

Inefficient both in performance and in hardware cost.

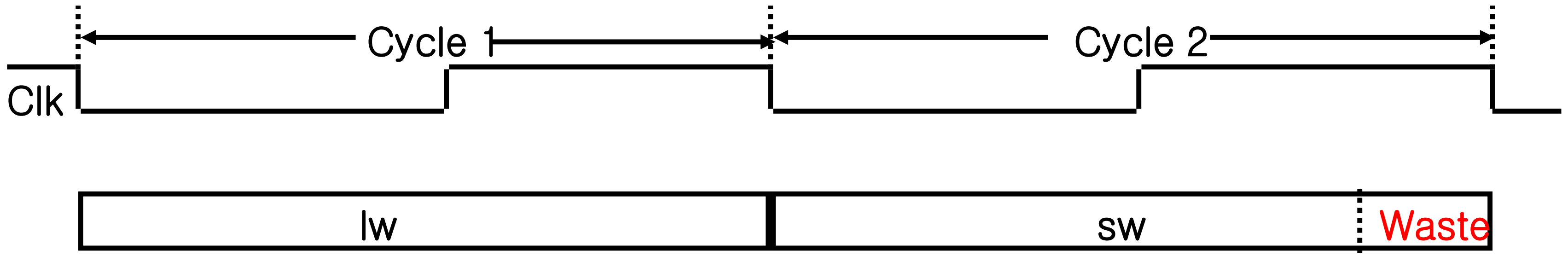
- **Multicycle datapath**
  - Shorter clock cycle
  - Multiple clock cycles for each instruction

Multicycle Datapath

- Each step in the execution will take 1 clock cycle
- Different number of clock cycles for the different instructions
- Sharing functional units within the execution of a single instruction

Single Cycle Disadvantages & Advantages

- Uses the clock cycle inefficiently – the clock cycle must be timed to accommodate the slowest instruction
  - especially problematic for more complex instructions like floating point multiply



- May be wasteful of area since some functional units (e.g., adders) must be duplicated since they can not be shared during a clock cycle
- but, is simple and easy to understand

Multicycle Implementation

Step	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps

1	fetch	IR ← Memory[PC]
		PC ← PC + 4		

2	decode	A ← Reg[IR[25:21]]
		B ← Reg[IR[20:16]]
		ALUOut ← PC + (sign-extend(IR[15:0]) << 2)		

execute 3	ALUOut ← A op B	ALUOut ← A + sign-extend(IR[15:0])	If(A == B) PC ← ALUOut	PC ← PC[31:28] || (IR[25:0]<<2)
store 4	Reg[IR[15:11]] ← ALUOut	Load: MDR←Memory[ALUOut] or Store: Memory[ALUOut] ← B		
5		Load : Reg[IR[20:16]] ← MDR		

Multicycle Datapath and Control

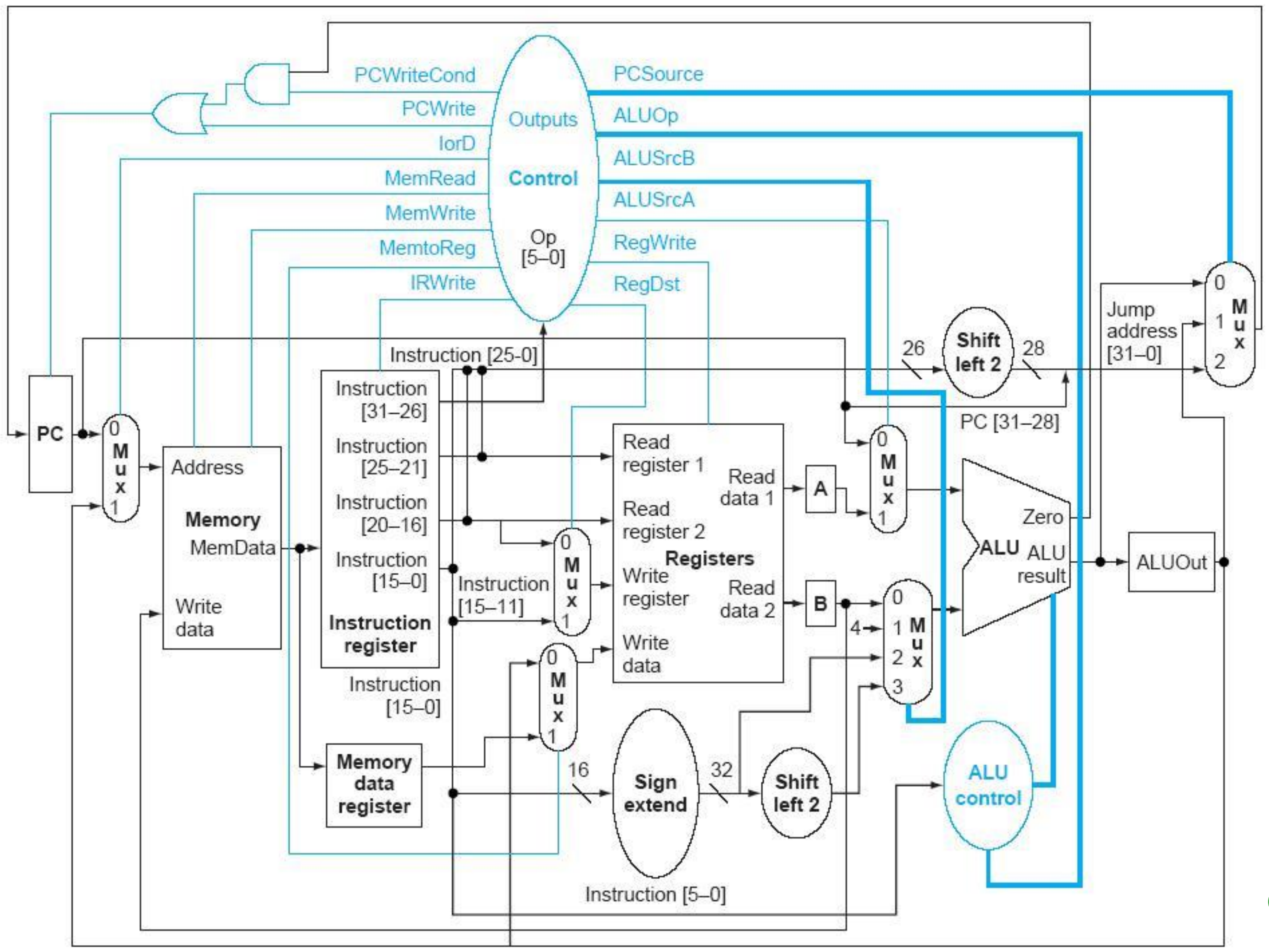


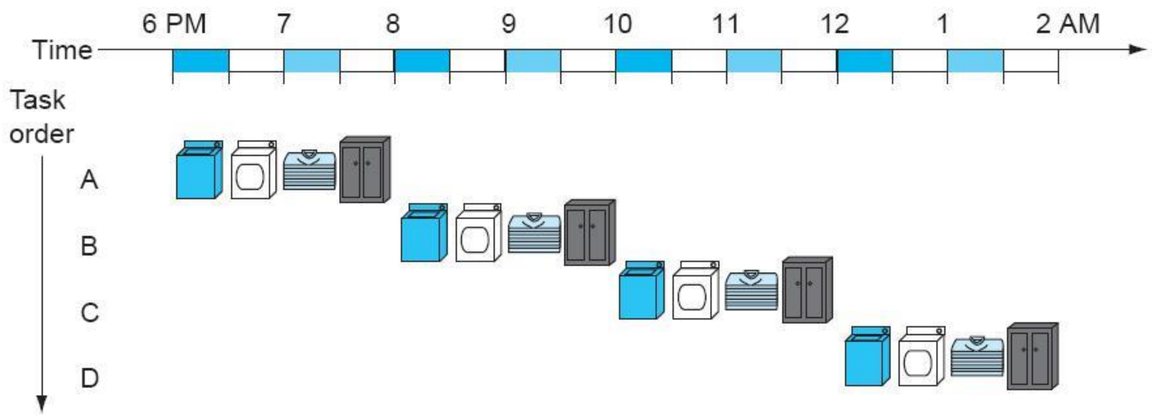
Figure 5.28 of 3rd edition

4.5 An Overview of Pipelining

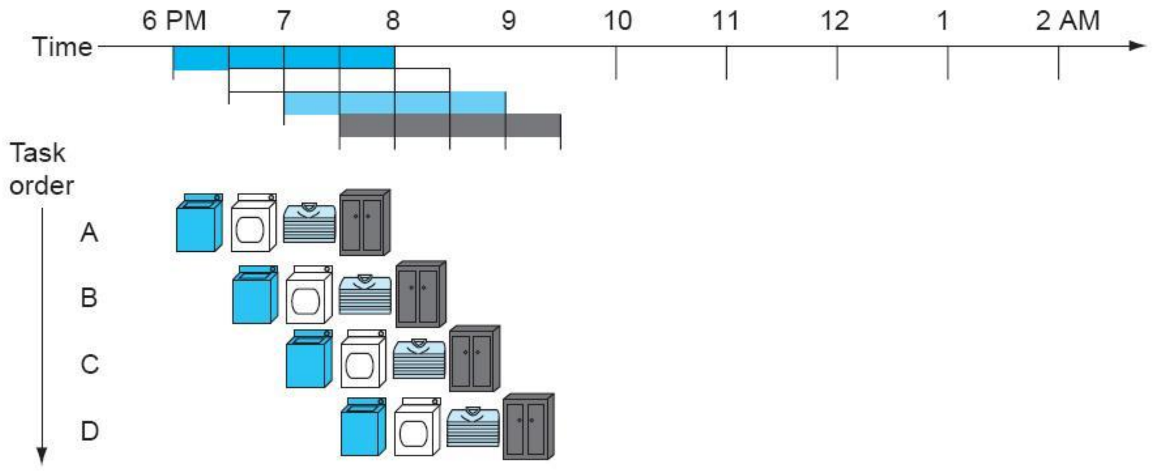
- **Pipelining**
  - Implementation technique in which multiple instructions are overlapped in execution
  - Exploits parallelism among the instructions in a sequential instruction stream
  - Improves instruction **throughput** rather than individual instruction **execution time**

The Laundry Analogy for Pipelining

Figure 4.25



8 hours for 4 tasks



3.5 hours for 4 tasks

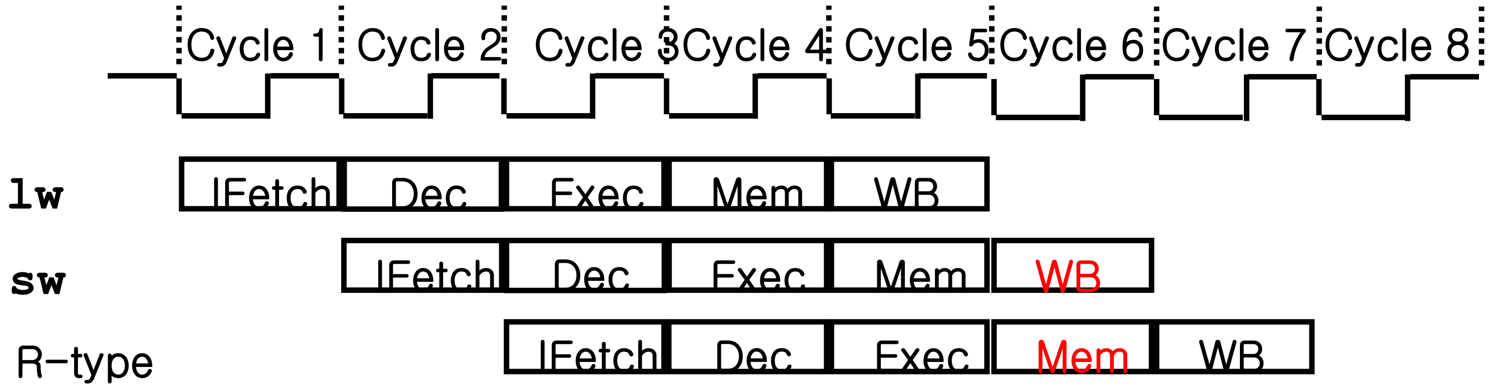
speedup = 8/3.5 = 2.3

5 Stages of Instruction Execution

1. Fetch instruction.(Ifetch)
2. Read registers while decoding the instruction.(Decode)
3. Execute the operation or calculate an address.(Exec)
4. Access an operand in data memory.(Memory)
5. Write the result into a register. (Write Back)

A Pipelined MIPS Processor

- Start the next instruction before the current one has completed
  - improves throughput - total amount of work done in a given time
  - instruction latency (execution time, delay time, response time - time from the start of an instruction to its completion) is *not* reduced

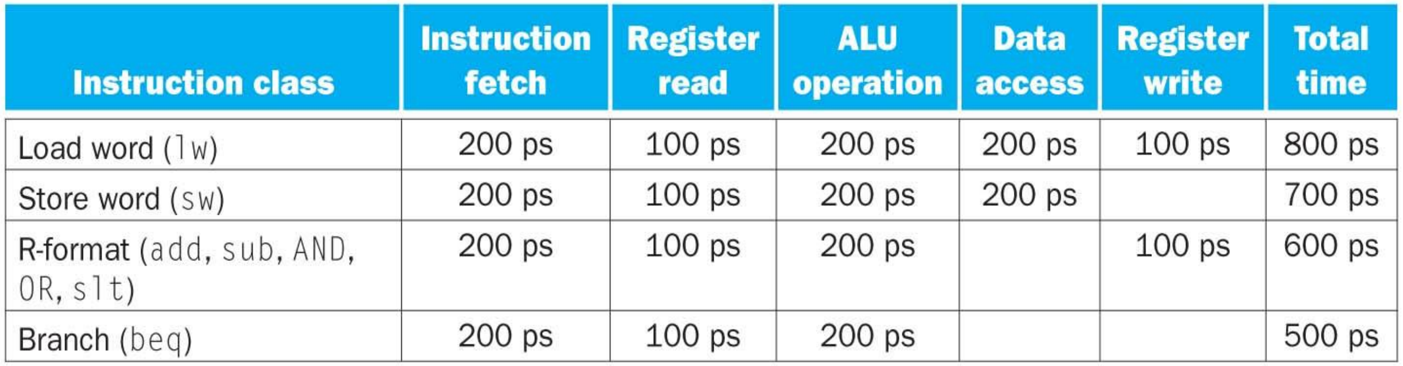


- clock cycle (pipeline stage time) is limited by the slowest stage
- for some stages don't need the whole clock cycle (e.g., WB)
- for some instructions, some stages are wasted cycles (i.e., nothing is done during that cycle for that instruction

Example: Single-Cycle vs. Pipelined

- Operation times for the major functional units
  - Memory access and ALU operation: 200 ps
  - Register file read or write: 100 ps
- **Compare the average time between instructions of a single-cycle implementation to a pipelined implementation.**

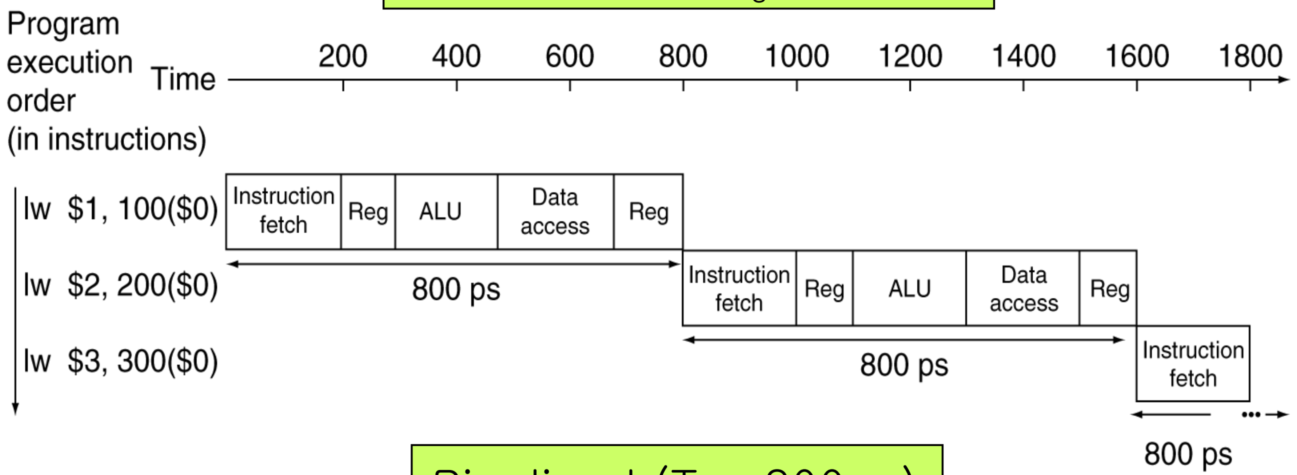
[Answer]



Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (lw)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (sw)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (add, sub, AND, OR, slt)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (beq)	200 ps	100 ps	200 ps			500 ps

Figure 4.26

Single-cycle (T_c= 800ps)



Pipelined (T_c= 200ps)

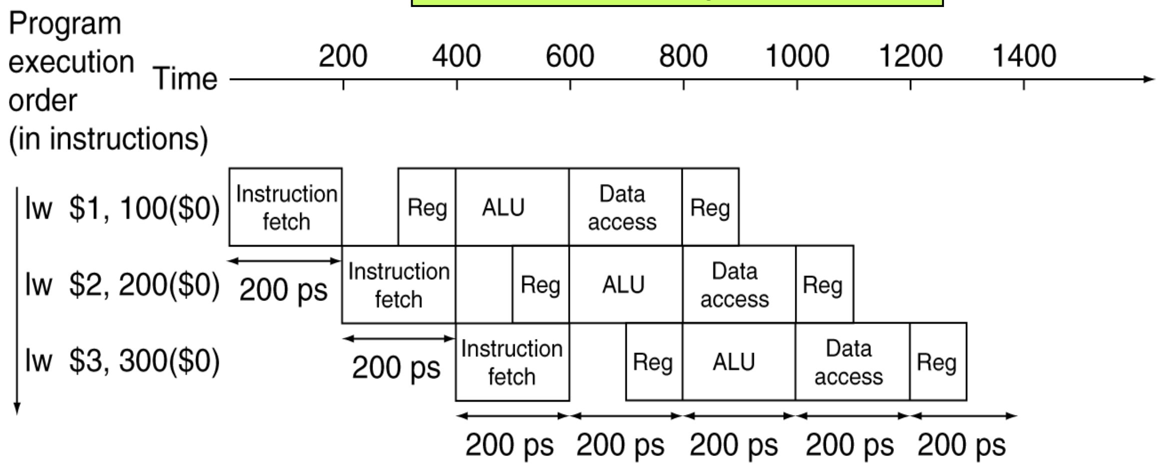


Figure 4.27

Pipeline Performance

- Clock cycle is determined by the time required for the slowest pipe stage.
- With perfectly balanced pipeline stages,

$$\text{Time between instructions}_{\text{pipelined}} = \frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of pipe stages}}$$

- Speedup of *k*-stage pipeline with clock cycle time=*t*
  - For n instructions,

n개 명령어를 순차적으로 수행할 경우 소요시간

$$\text{speedup} = \frac{n \times k \times t}{(k-1) \times t + n \times t} = \frac{n \times k \times t}{k \times t + (n-1) \times t}$$

  - For infinite number of instructions (i.e. $n \rightarrow \infty$),

speedup = *k*

n개 명령어를 파이프라이닝 방식으로 수행할 경우 소요시간

4.6 Pipelined Datapath and Control

- **5 stages of instruction pipeline**
  - IF: Instruction fetch
  - ID: Instruction decode and register file read
  - EX: Execution and address calculation
  - MEM: Data memory access
  - WB: Write back

5 Steps of MIPS Datapath

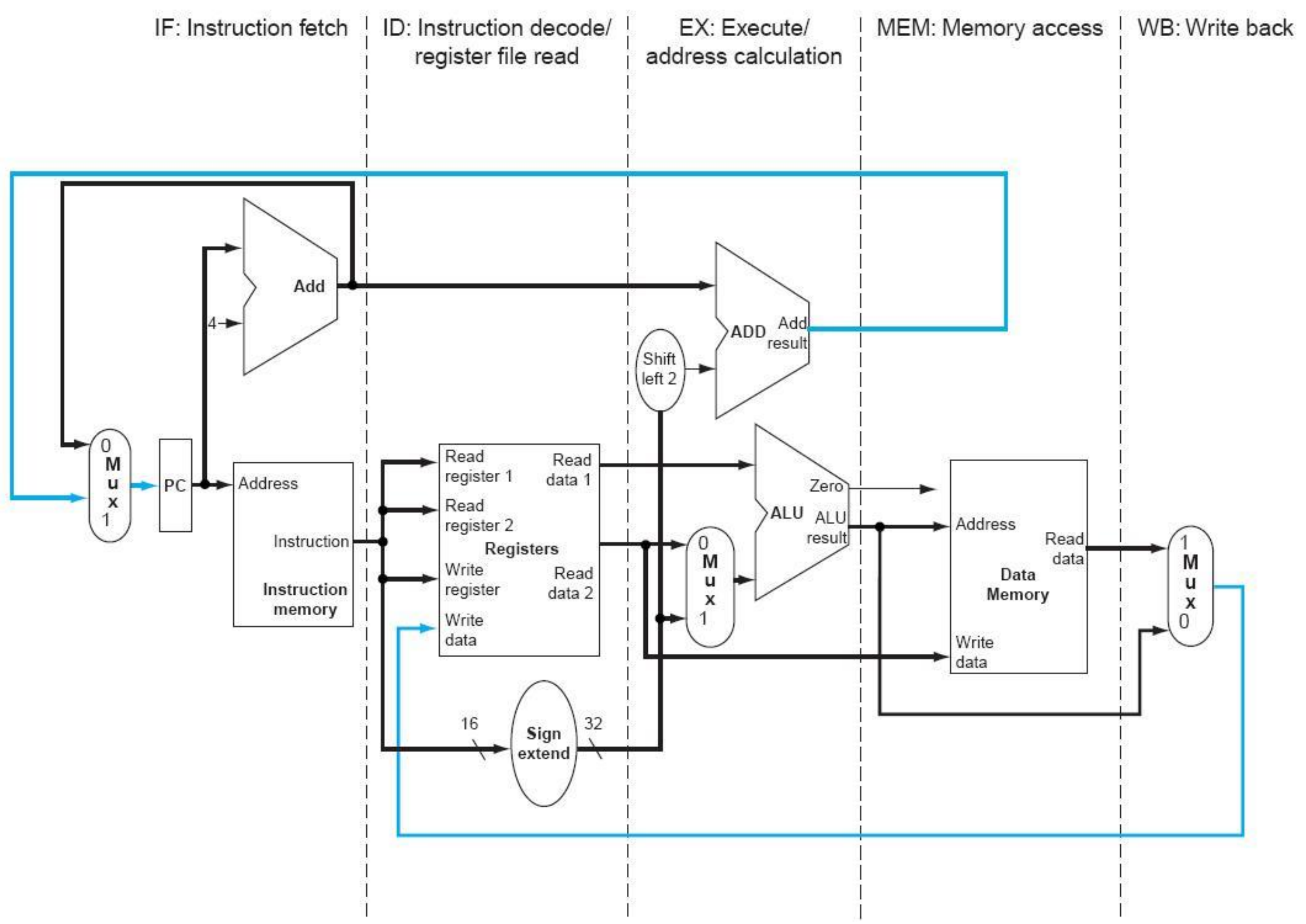


Figure 4.33

Pipelined Execution

- **2 exceptions to the left-to-right flow of instructions**
  - Write-back stage: Send ALU result back to the register file
    ⇒ Data hazard (성능저하요인 - 4장 Part2에서 공부)
  - Next PC select: Incremented PC or the branch address from MEM stage ⇒ Control hazard(성능저하요인- 4장 Part2에서 공부)

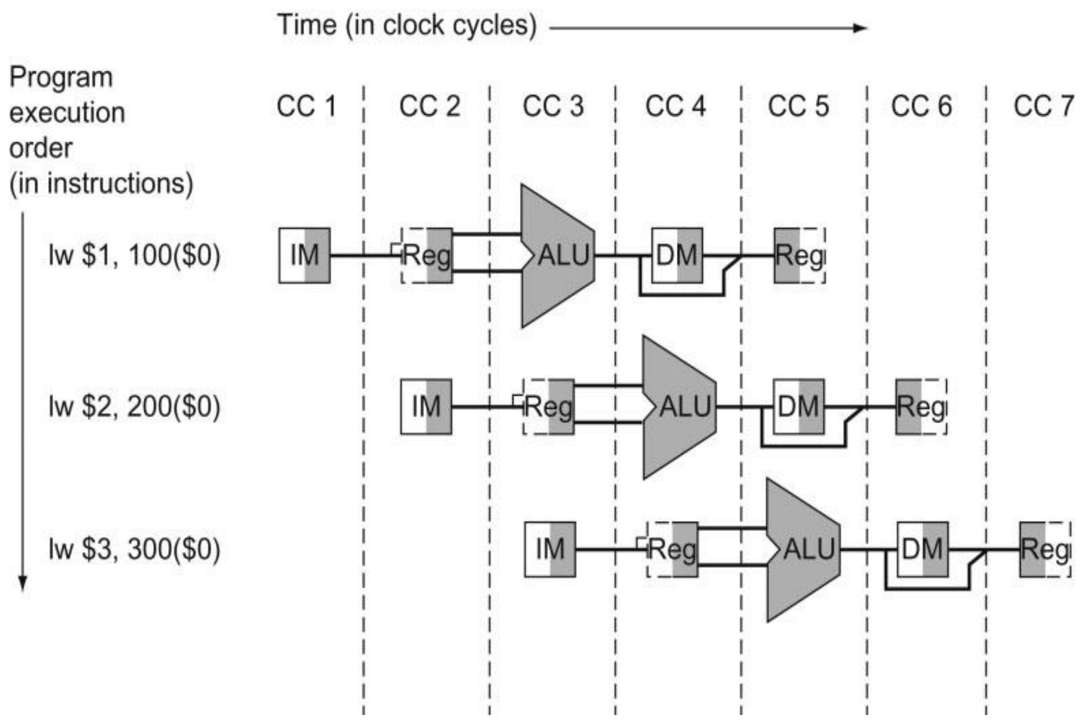
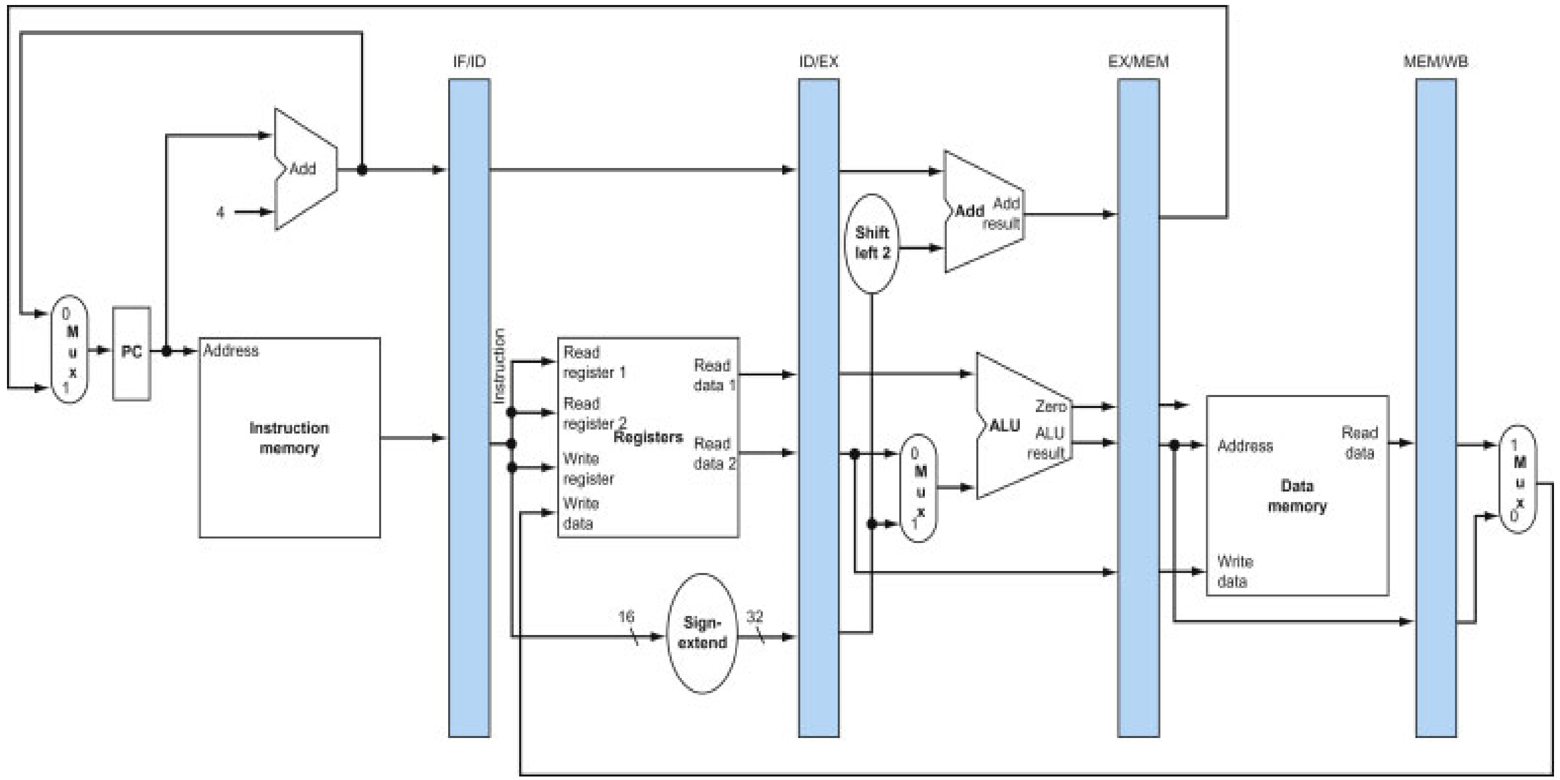


Figure 4.34

Pipelined Version of the Datapath

- **Pipeline register**
  - Separation of the two stages
  - Hold information produced in previous cycle

Figure 4.35



Graphically Representing Pipelines

- **Multiple-clock-cycle pipeline diagram**
  - Showing resource usage

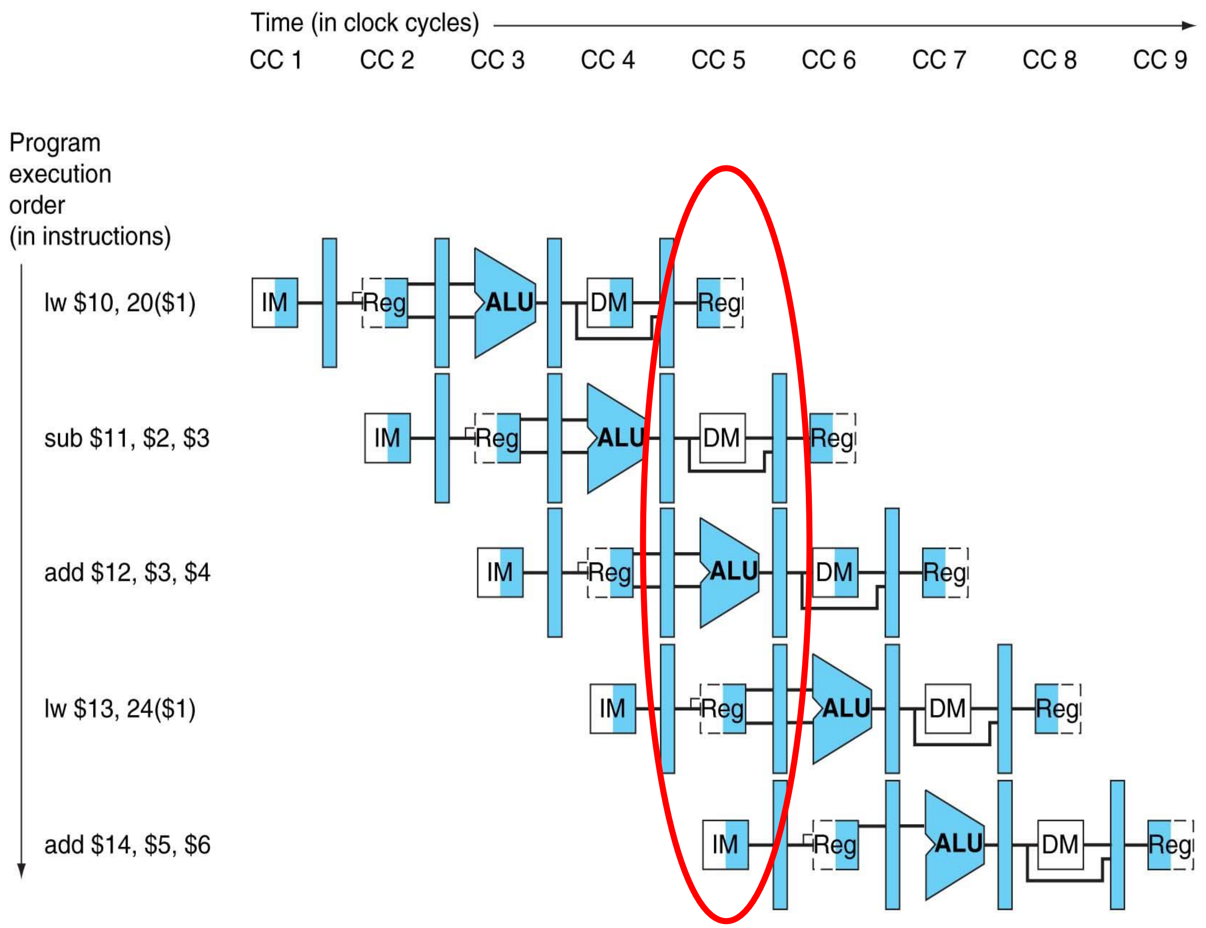


Figure 4.43

Traditional Multi-Cycle Pipeline Diagram

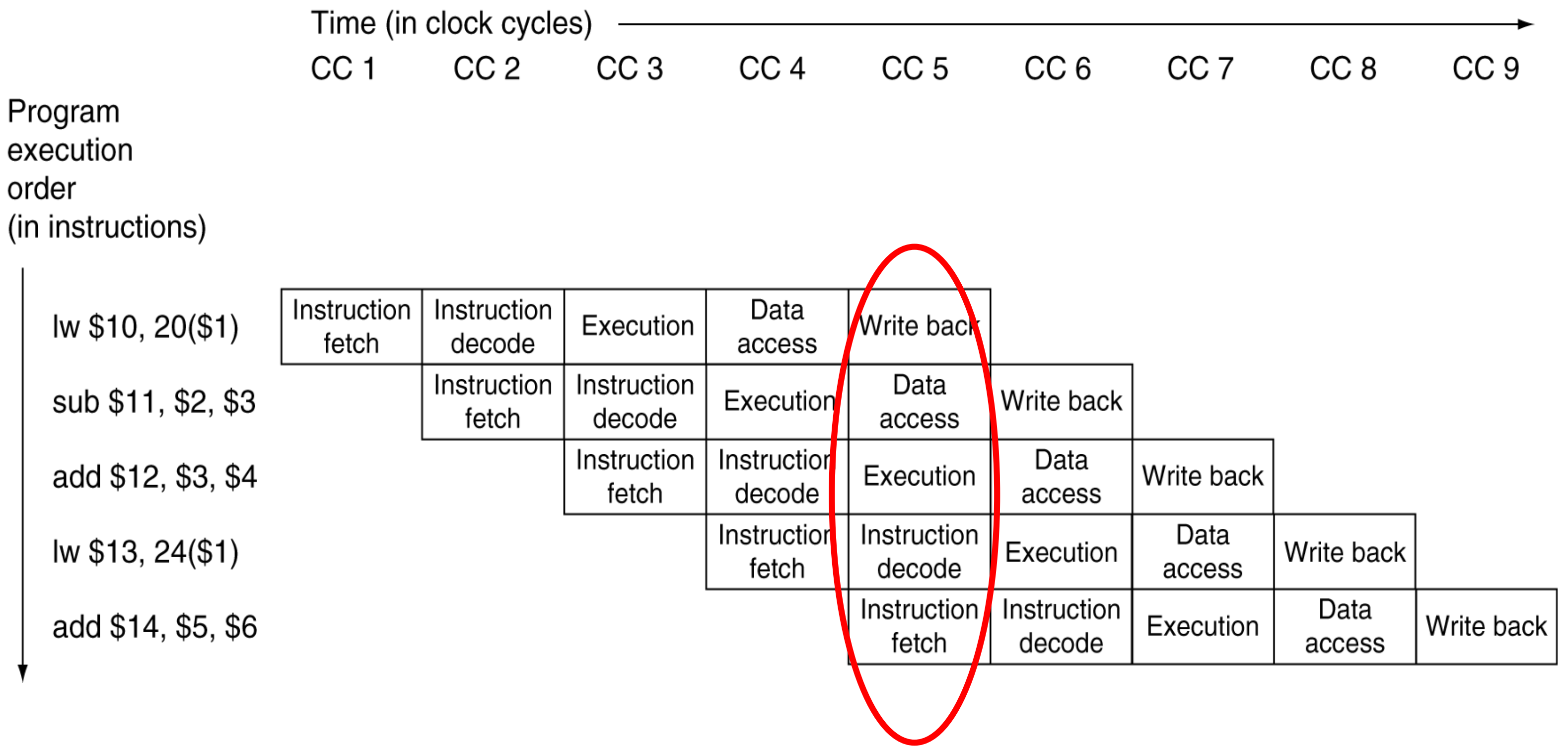


Figure 4.44

Pipelined Version of the Datapath

- **Pipeline register**
  - Separation of the two stages
  - Hold information produced in previous cycle

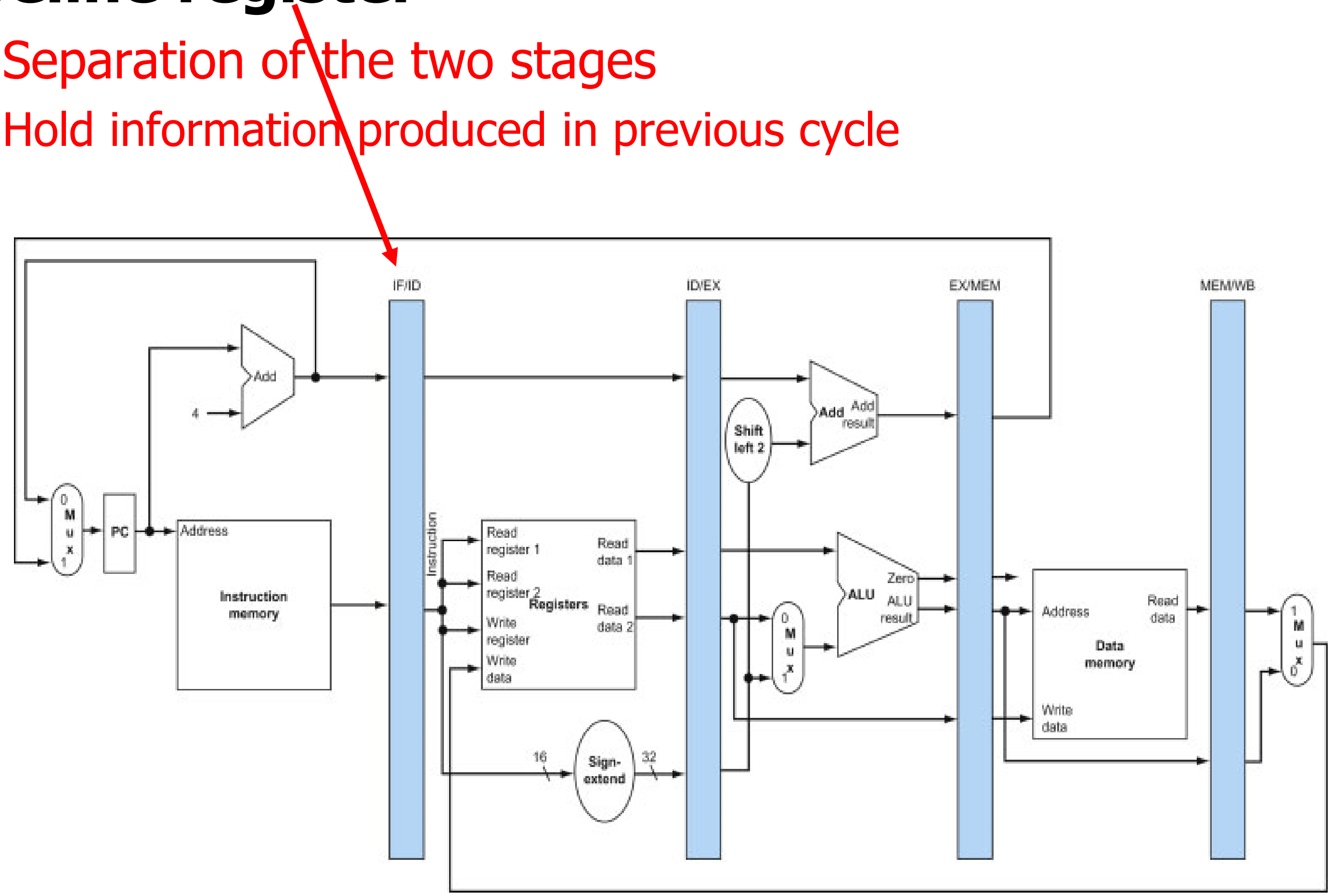


Figure 4.35

`lw` Instruction in IF Stage

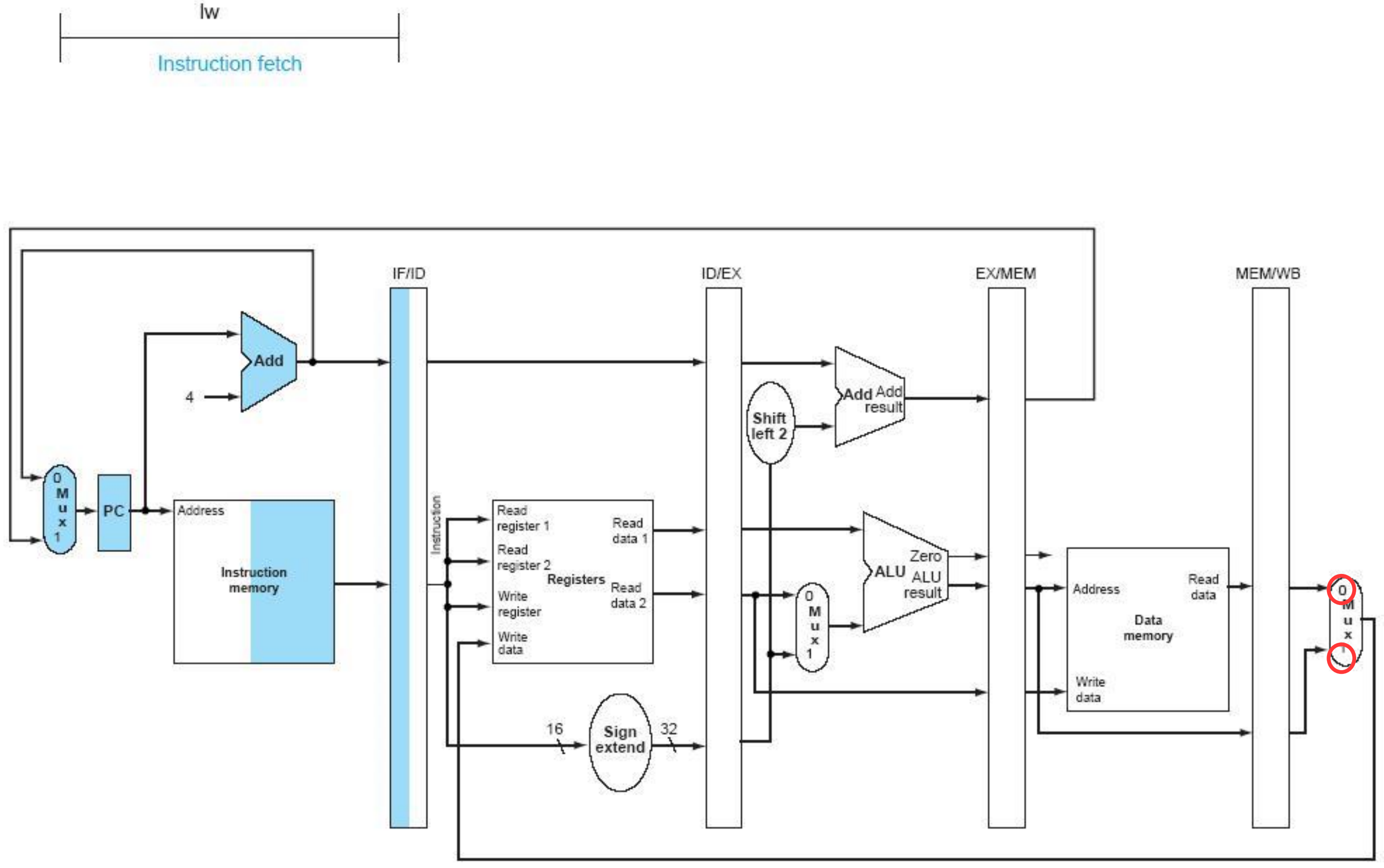


Figure 4.36(a)

`lw` Instruction in ID Stage

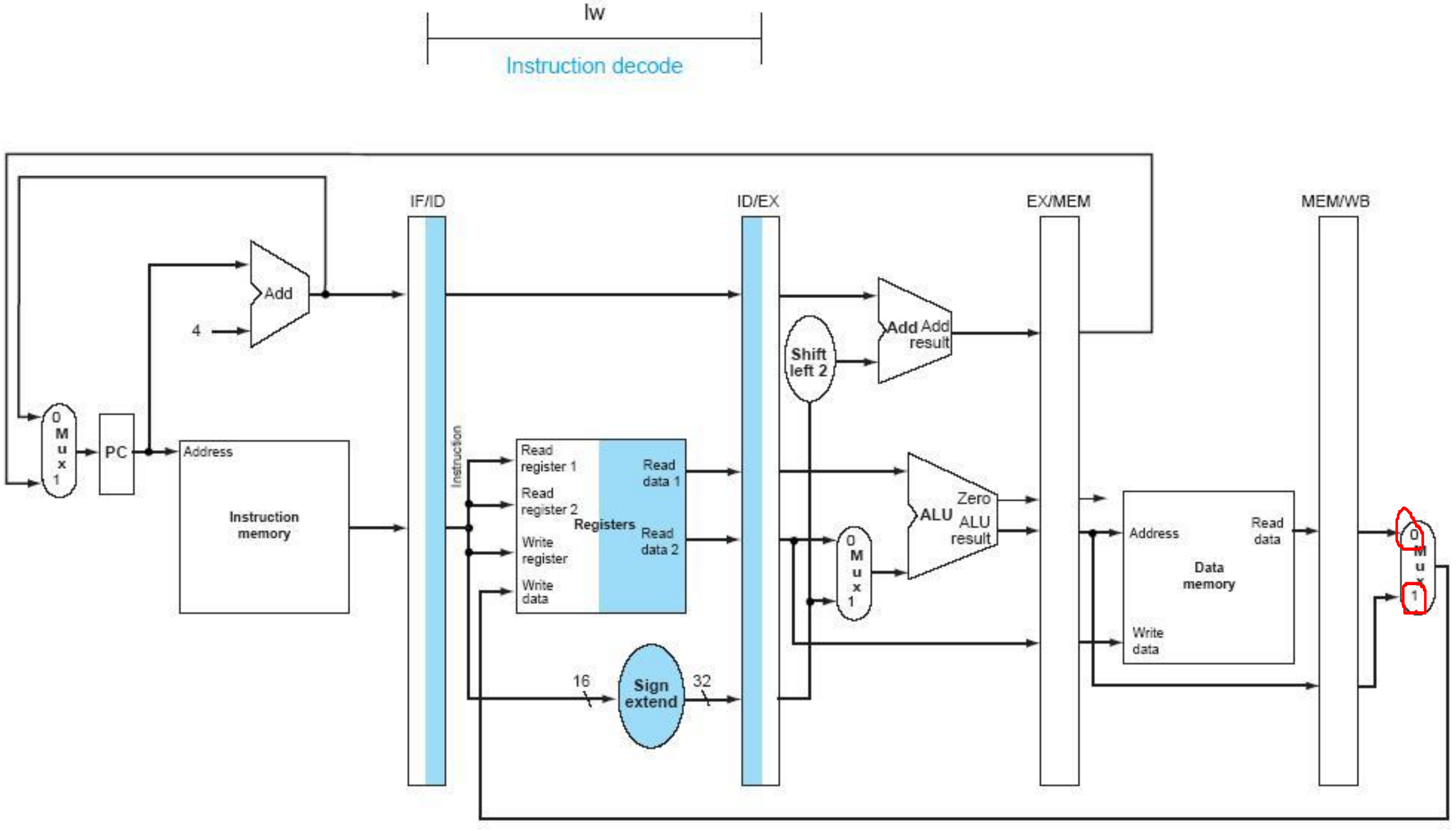


Figure 4.36(b)

`lw` Instruction in EX Stage

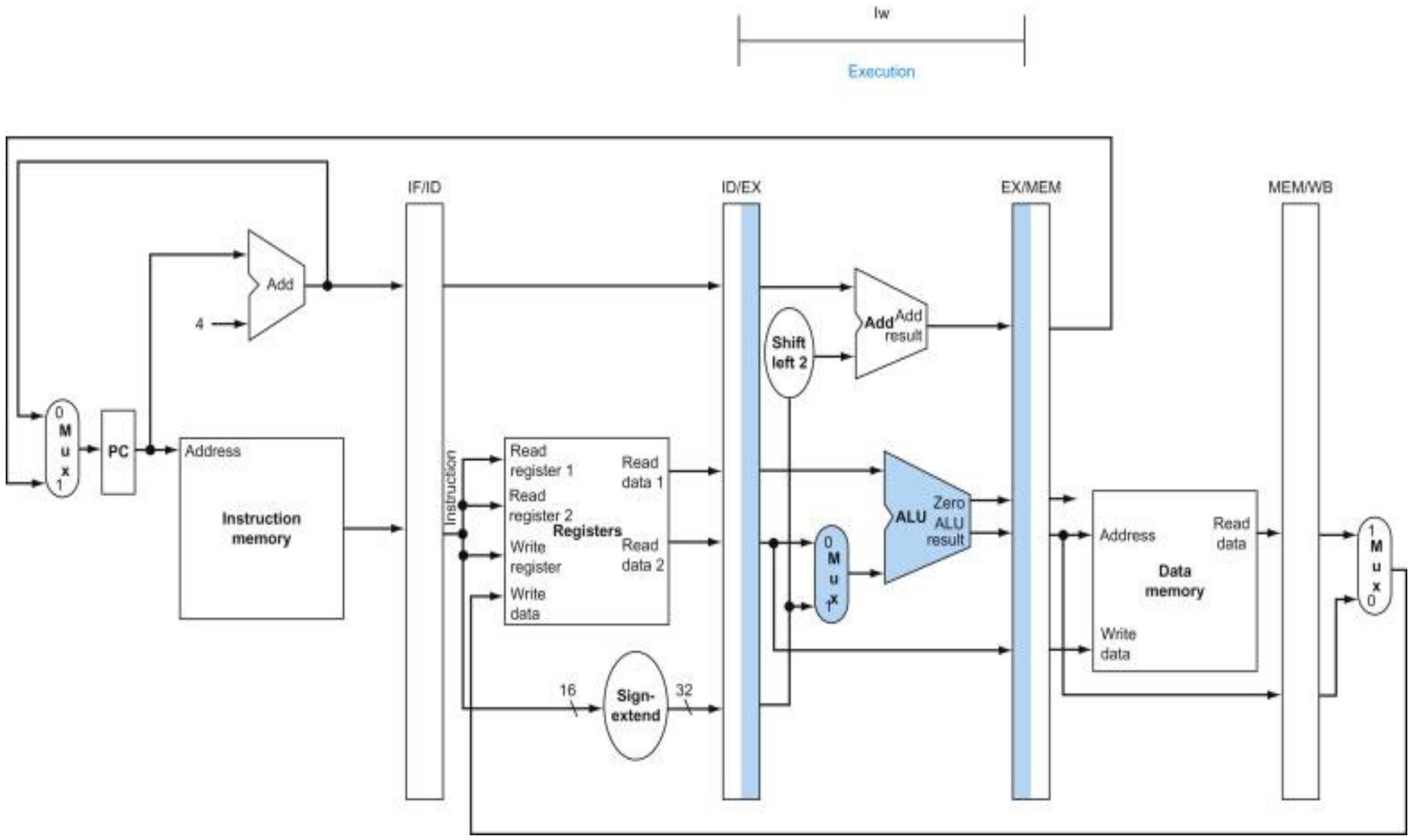


Figure 4.37

lw Instruction in MEM Stage

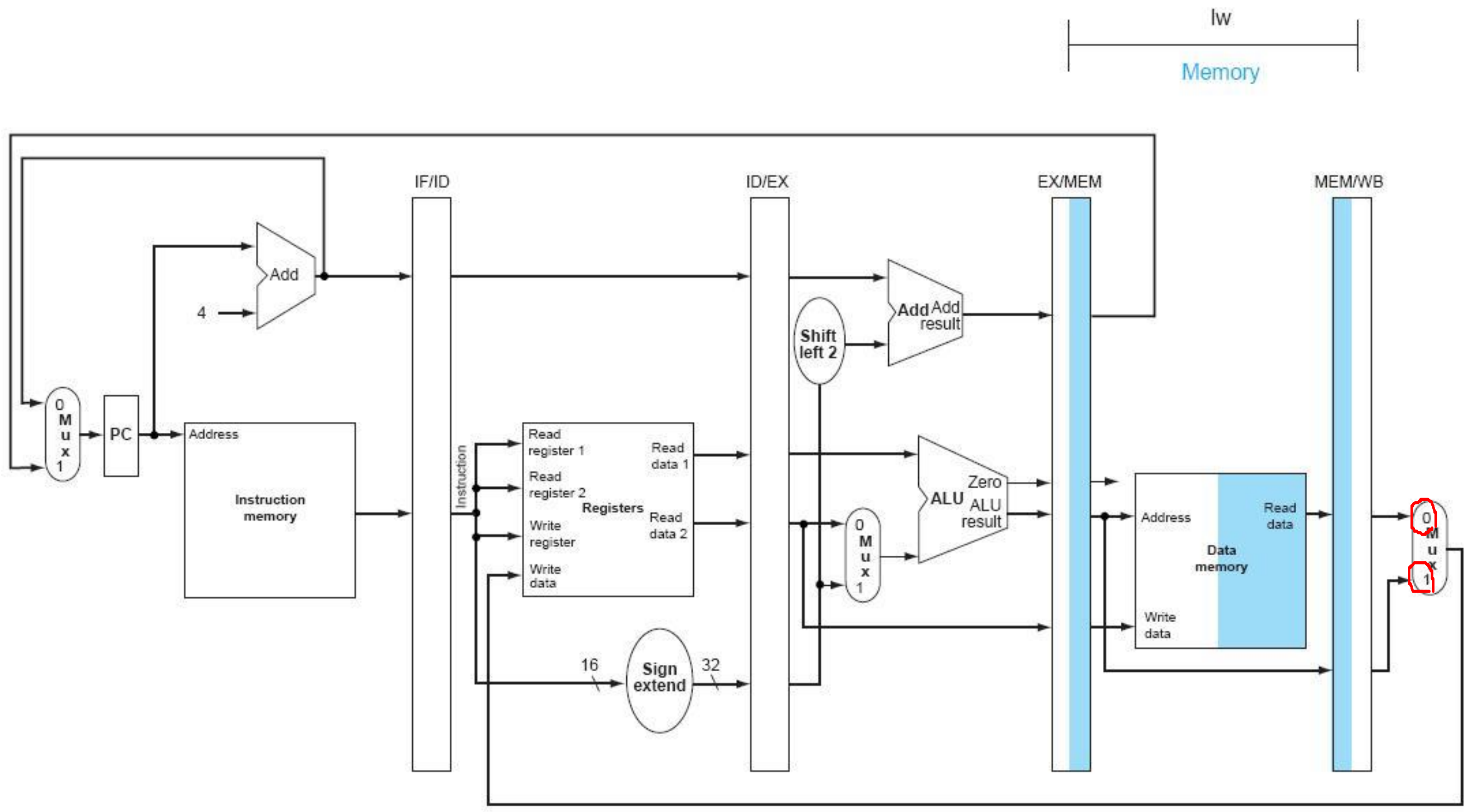


Figure 4.38(a)

`lw` Instruction in WB Stage

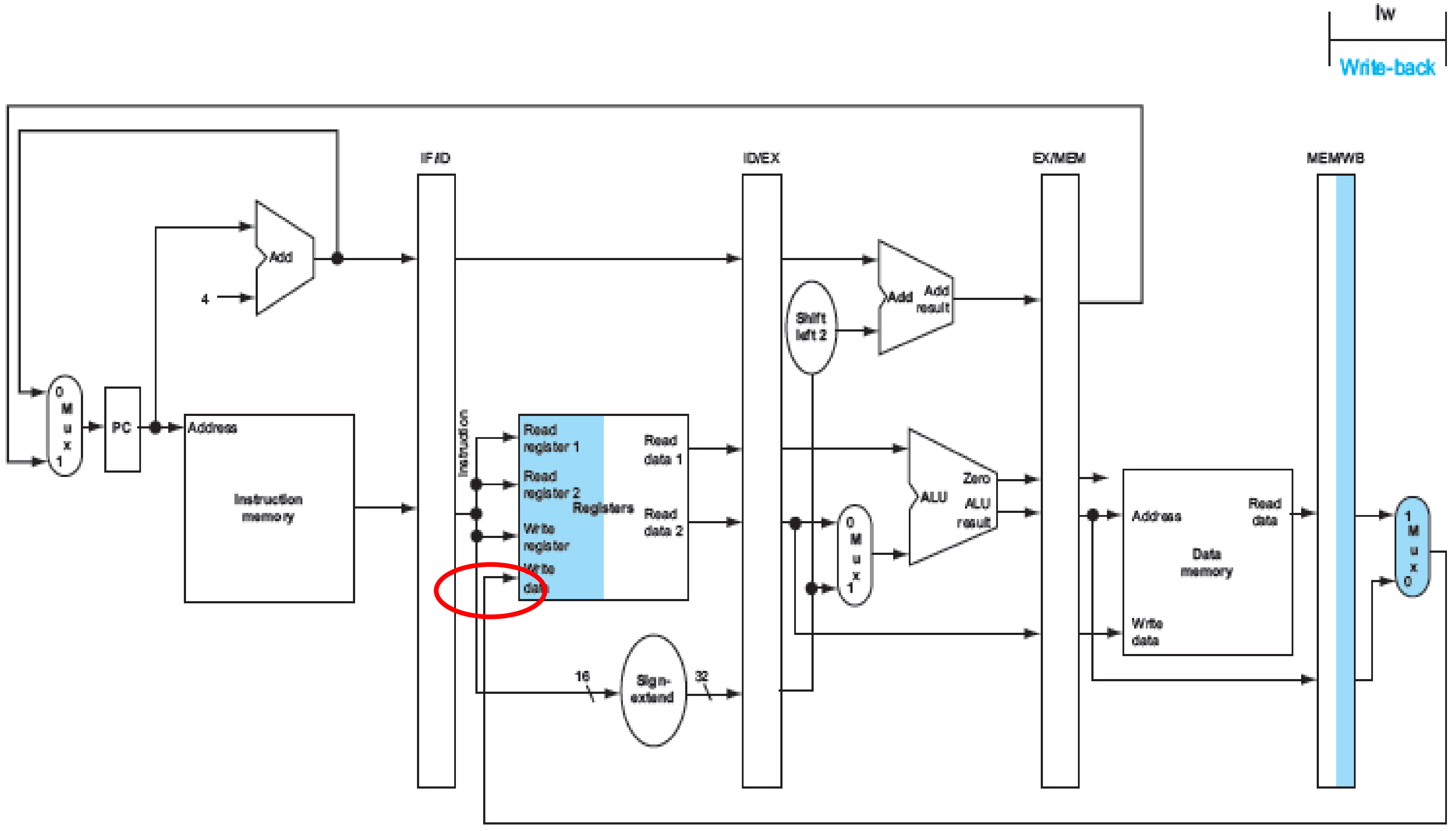


Figure 4.38(b)

Corrected Pipelined Datapath

- `load` **instruction**
  - Write register number: Should be preserved until WB stage

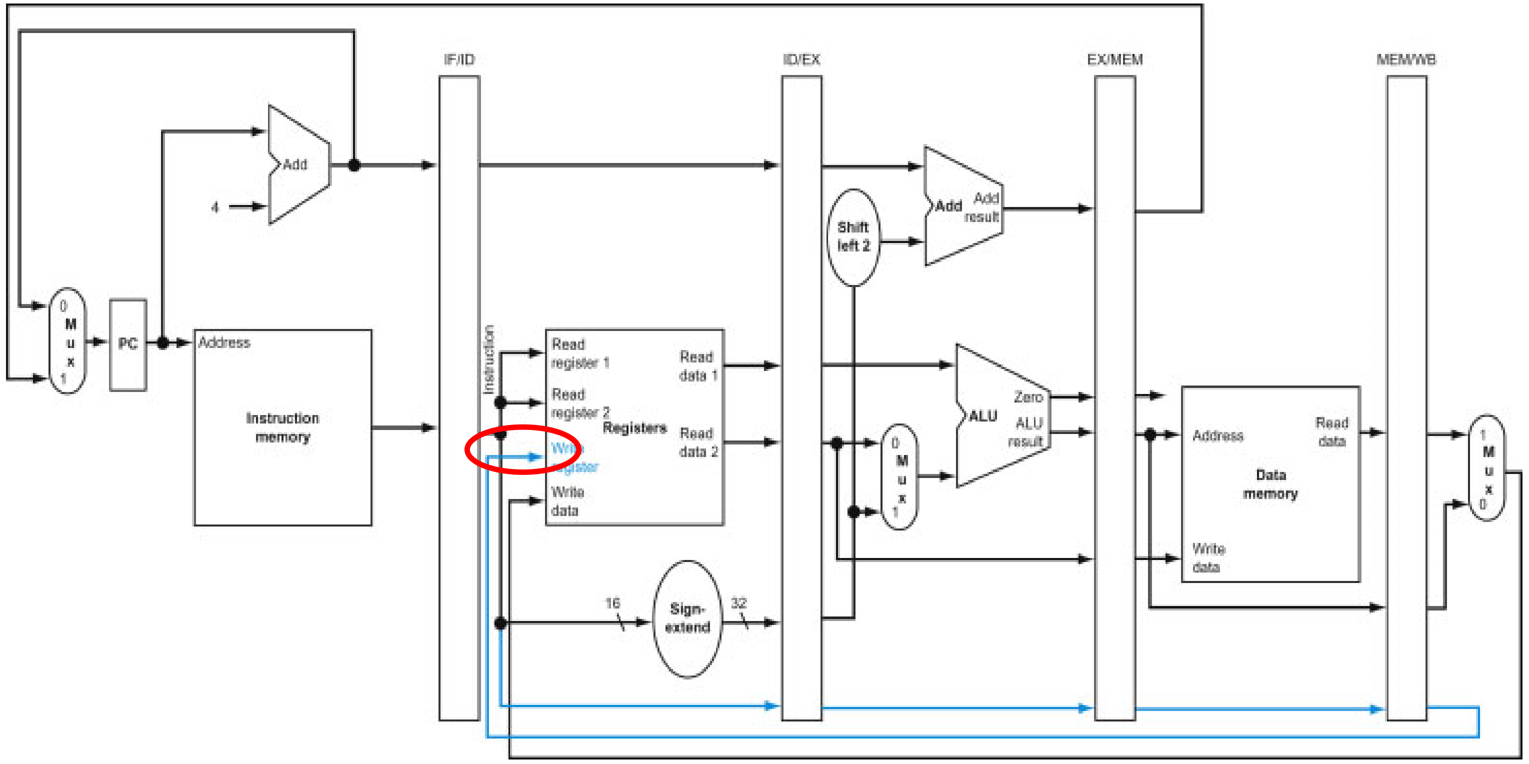


Figure 4.41

Pipelined Control

- **Pipelined datapath with control signals**

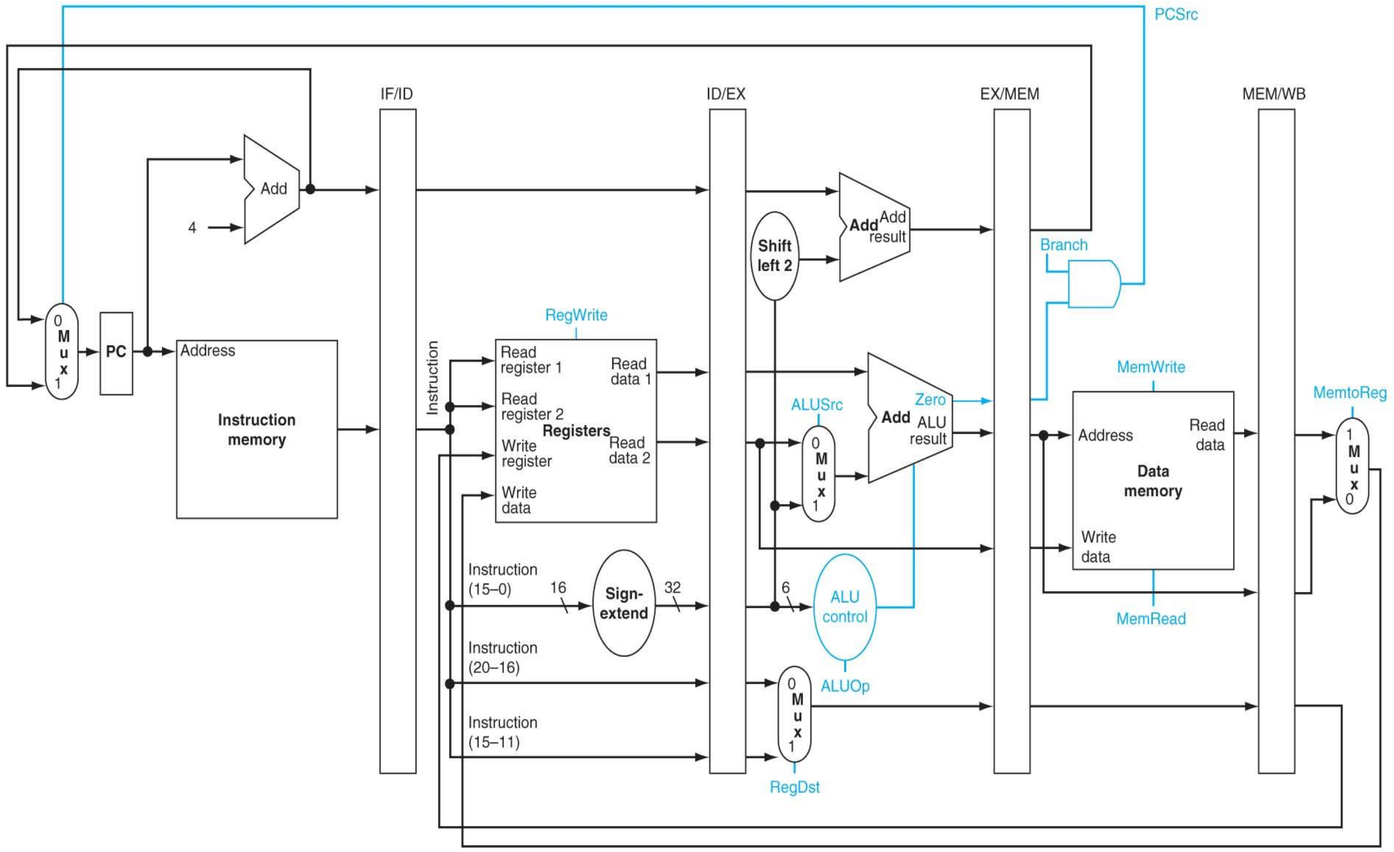
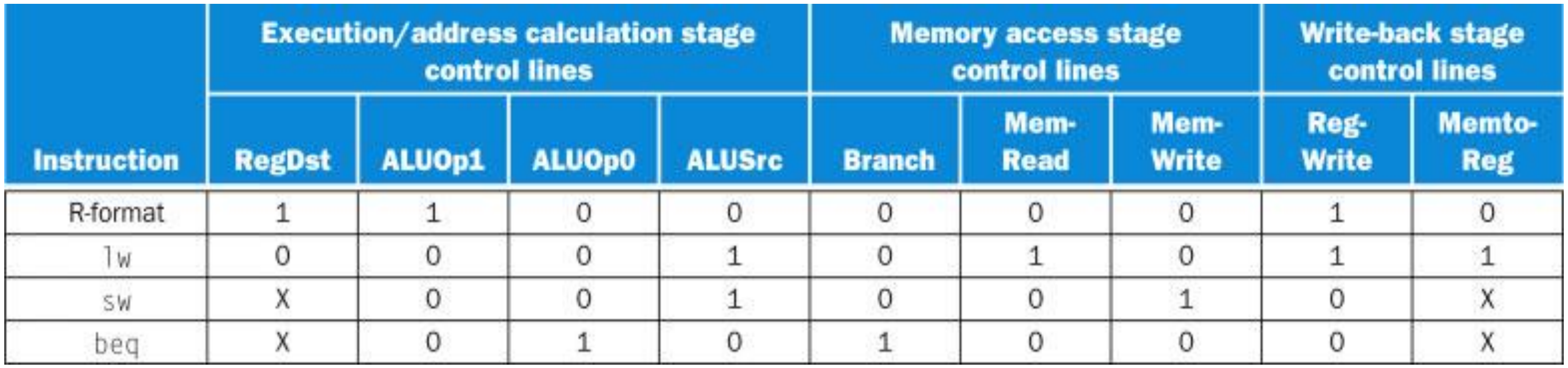


Figure 4.46

Control Signals

1. IF : nothing (the same thing happens at every clock cycle)
2. ID : nothing
3. EX : RegDst, ALUOp, ALUSrc
4. MEM : Branch(for `beq`), MemRead(for `lw`), MemWrite(for `sw`)
5. WB : MemtoReg, RegWrite



	Execution/address calculation stage control lines				Memory access stage control lines			Write-back stage control lines	
Instruction	RegDst	ALUOp1	ALUOp0	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
R-format	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X

Figure 4.49

Control Signals for Each Stage

- Control signals derived from instruction

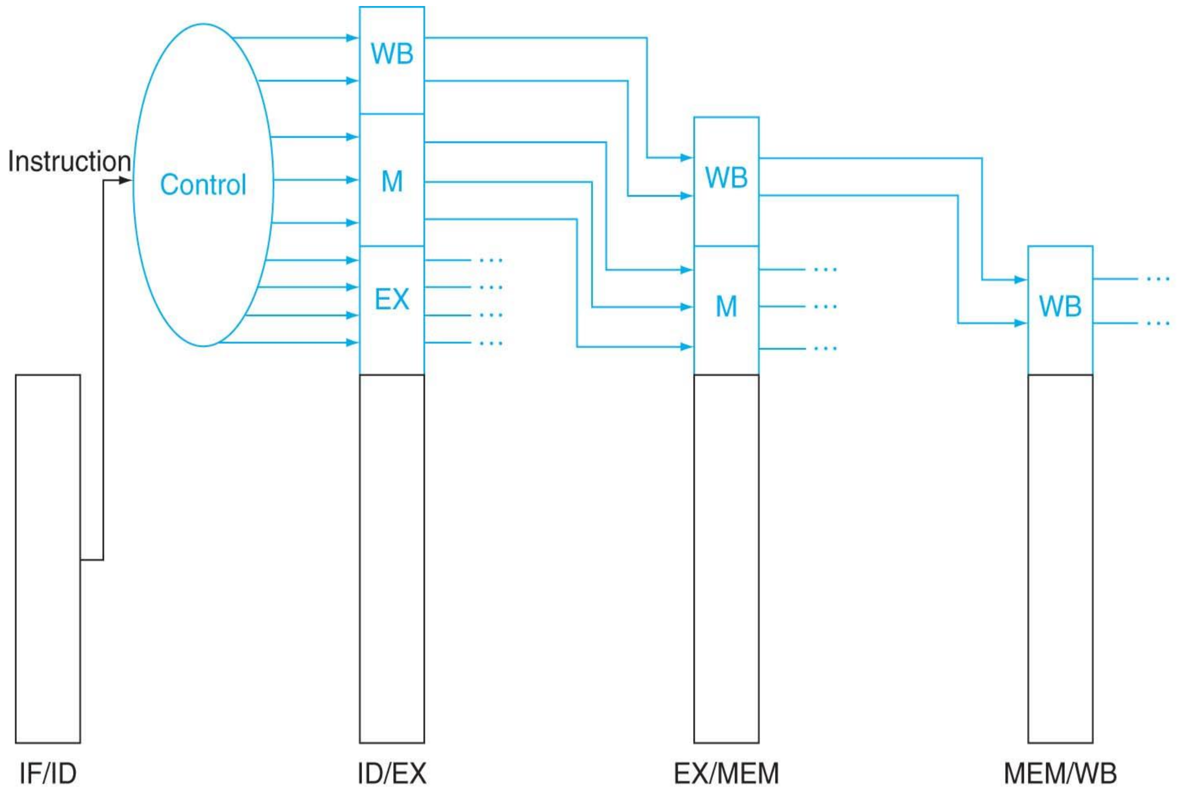


Figure 4.50

Complete Pipelined Datapath

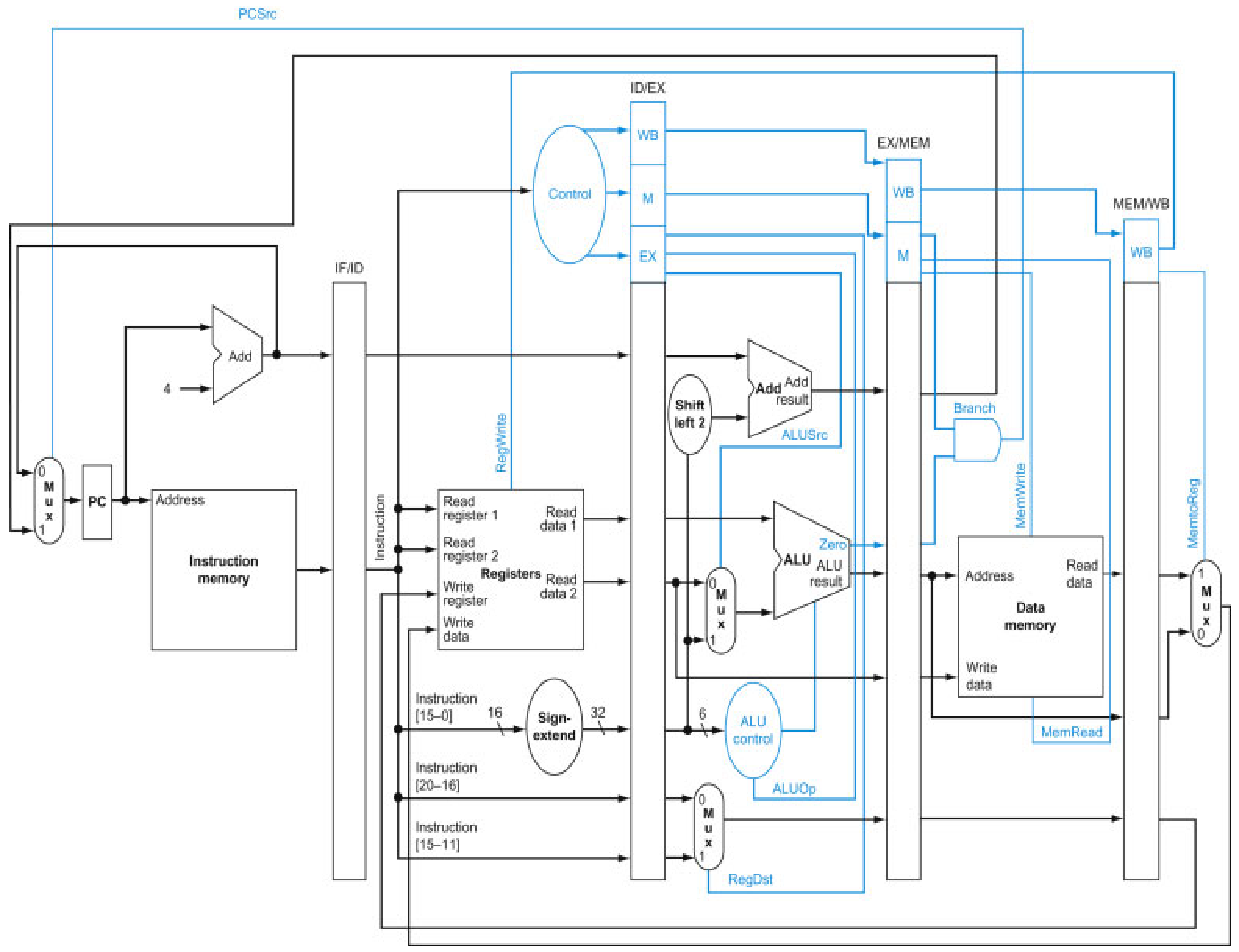


Figure 4.51