Chapter 4. The Processor

Introduction

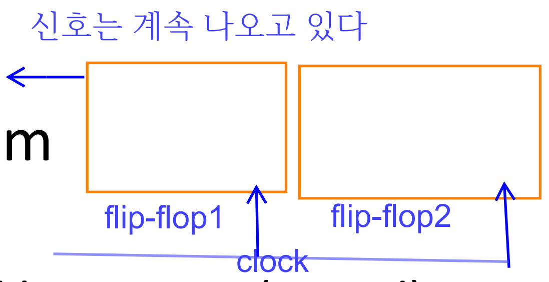
- What you will learn in this chapter
  - Principles and techniques used in implementing a processor
    - Simplified design CPU design
    - Pipelined design 핵심 (simplified를 배워야 pipelined를 알 수 있어서 배우는 것)
  - Implementation of subset of core instructions
    - lw, sw
    - add, sub, AND, OR, slt
    - beq, j
- Instruction Execution Instruction 수행
  - First two steps
    - Fetch instruction: address in PC fetch=메모리에 가서 가져온다 instruction fetch=pc레지스터에서 다음 실행할 주소를 계속 가져옴
    - Read registers: lw reads 1 register, others read 2 registers lw면 레지스터 1개 읽고, 나머지는 레지스터 2개읽어옴
  - Following steps are dependent on the instruction class
    - $\boxed{\text{Use ALU}}$
      - For arithmetic results
      - Memory address
      - Branch target address

add $t0, $t1, $t2랑
논리가 같음..
beq $t1, $t2, LABEL
(빼기해서 같은지 확인함)
jump 빼고 다 일관성있다

Logic Design Basics

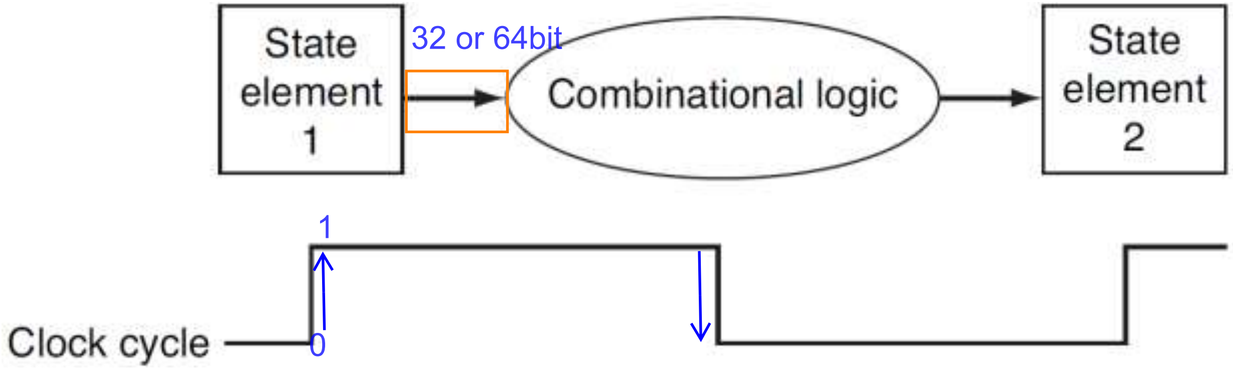
- Digital Logic Element types 차이점=저장이 가능한지 안한지
  - Combinational element
    - Operate on data
    - No internal storage
    - Output is a function of input
      - Given the same input, it always produce the same output
  - State (Sequential) element state를 저장함=stateful
    - State stored in internal storage
    - Two input and one output at minimum
      - Input: data value, clock
        - » clock determines when data should be written (stored)

다시.. 자료받기 그림있음
and, or, not 회로



Logic Design Basics

- Clocks
  - Timing of reads/writes must be controlled
    - Read/Write at the same time→ unpredictable output
- Edge-triggered clocking 0 => 1 or 1=>0으로 바뀌는 구간= clock이 튄다 =>state변함
  - transition from low→high or high→low
  - Data in a sequential logic element is updated only on a clock edge
  - Combinational logic takes input from state elements and outputs data to state elements

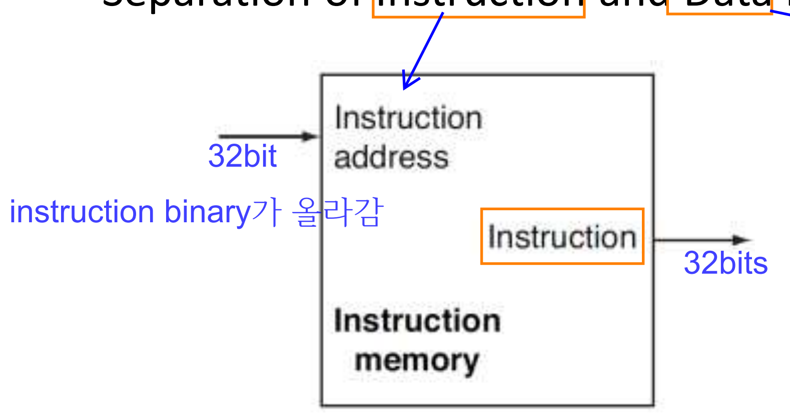


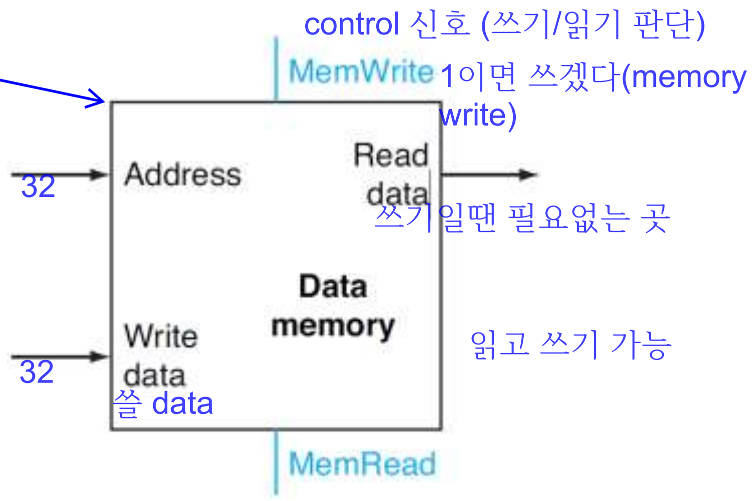
- Signal must propagate from SE1 to SE2 in 1 clock
- Optimal clock cycle length is determined by the combinational logic

Datapath Elements inside Processors

CPU 구성요소

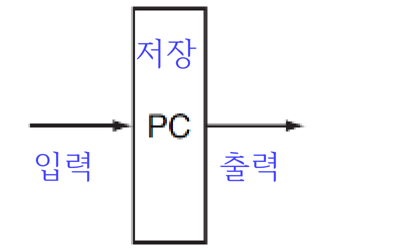
- Memory unit D-Ram 메모리같은 것의 구성
  - Separation of Instruction and Data memory





CPU에서 메모리를 주고 받는 interface(통로)

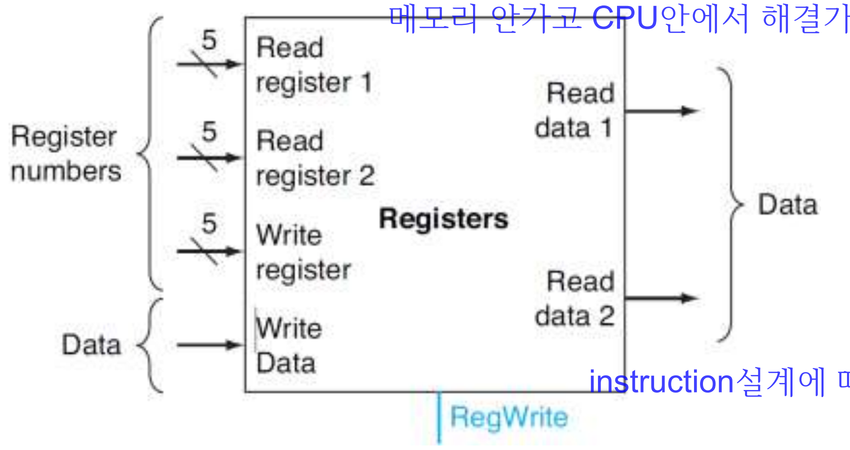
- PC



clock이 튈때까지 걸려있다가 튀면 출력하고 새로 들어옴

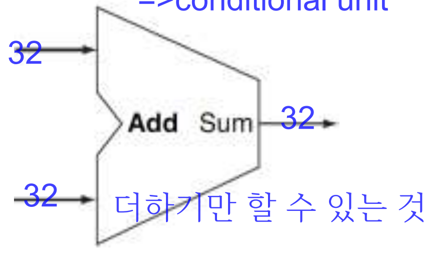
- Register file

register 많음..
interface를 하나 만들자
ex)읽고자하는 register몇 개 연결하면 read 나 write하는 interface
메모리 안가고 CPU안에서 해결가능



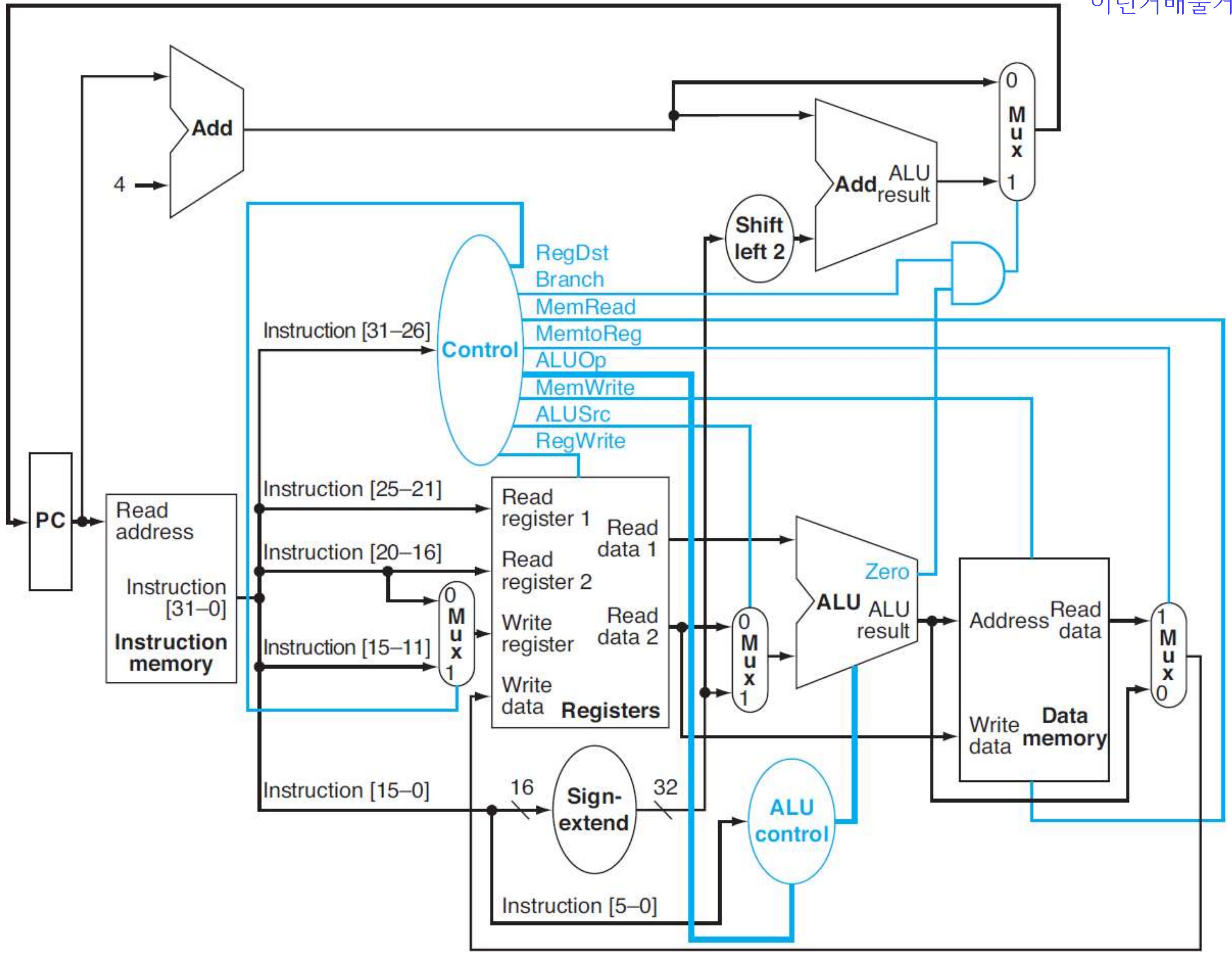
instruction설계에 따라 달라짐

- ALU 값이 안에 저장되지 않음(박스형태X) =>conditional unit



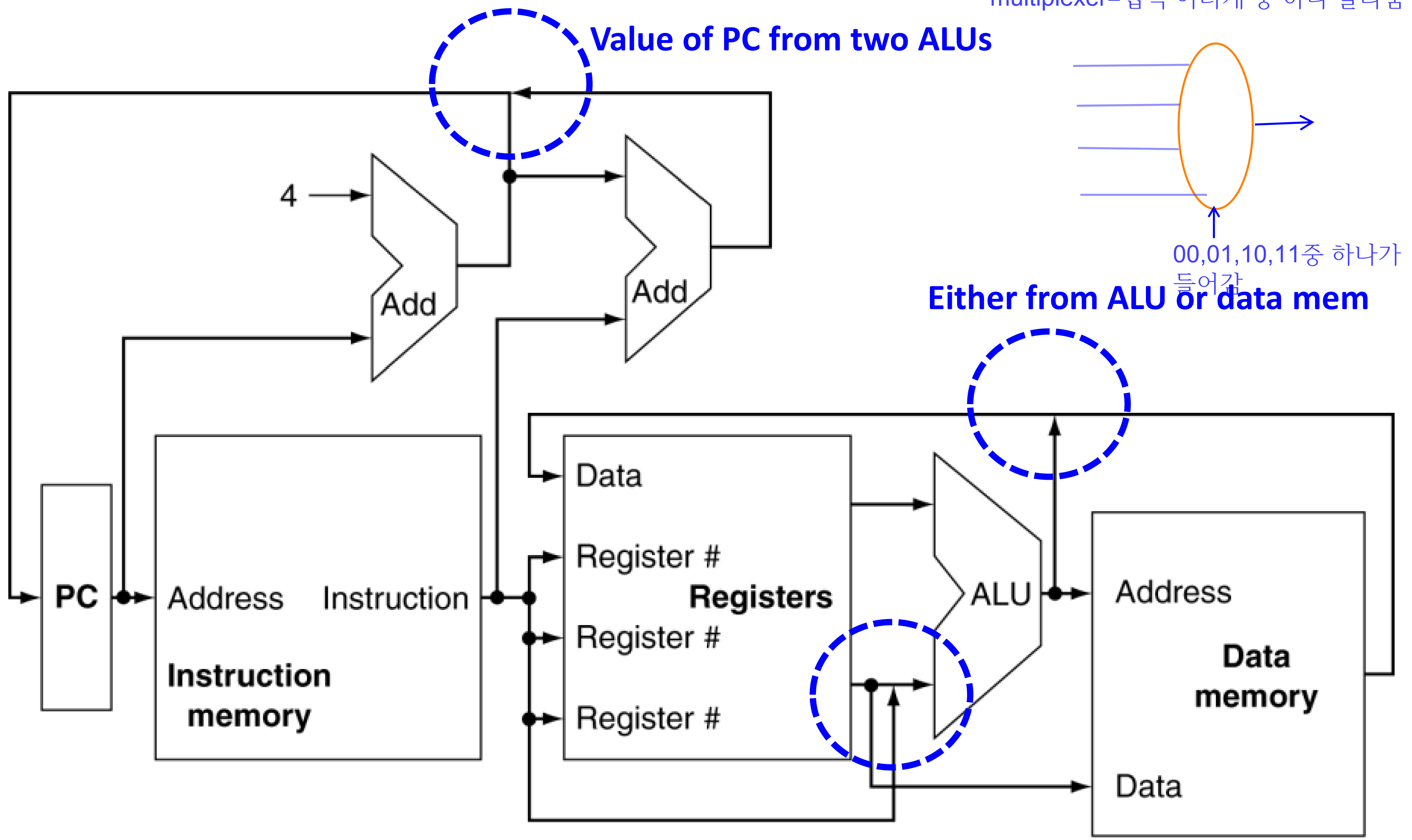
Completed CPU Architecture - nonpipelined

이런거배울거다



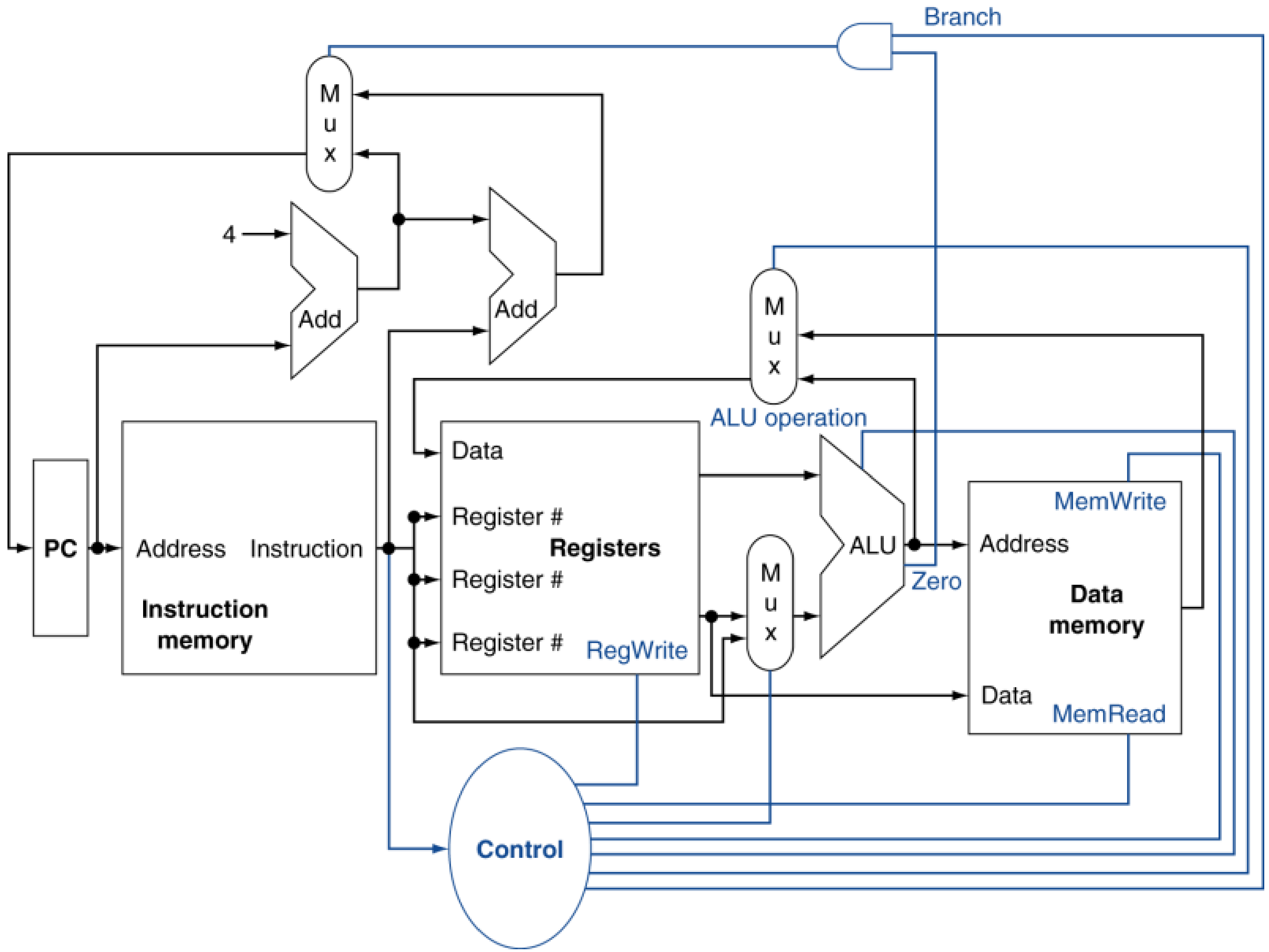
Abstract View of CPU Implementation

multiplexer=입력 여러개 중 하나 골라줌

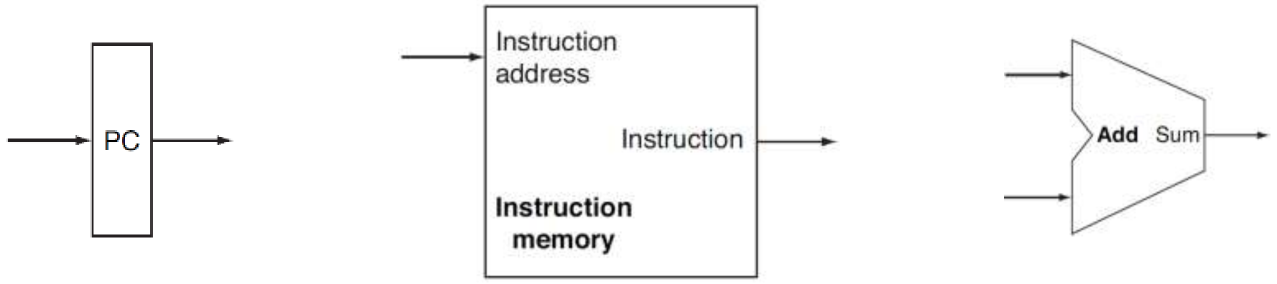


Either from registers of from immediate field

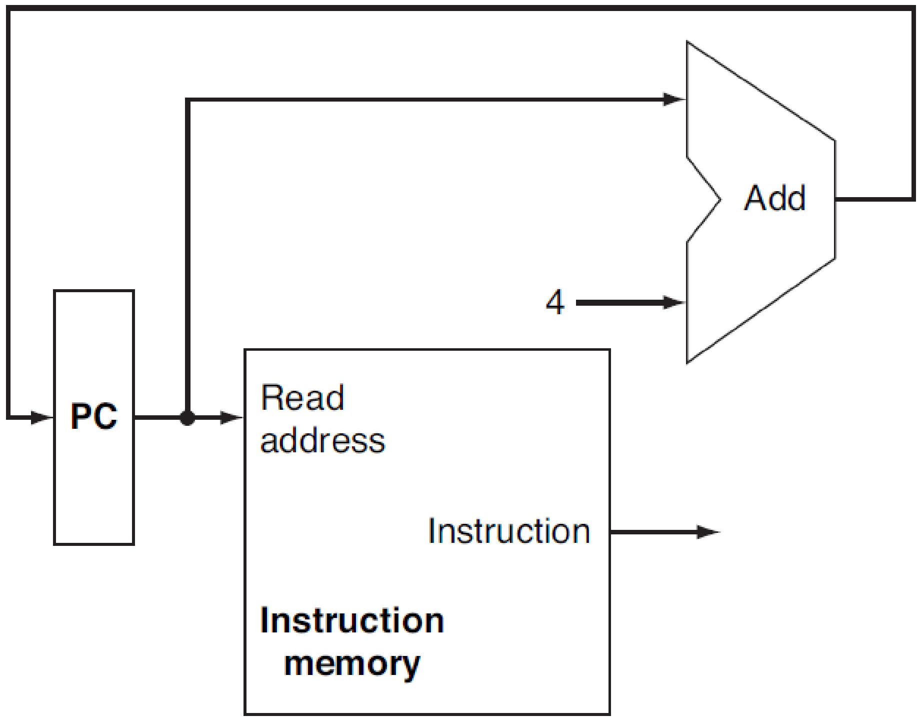
CPU Implementation with Control



Building Datapath: Fetching Instructions



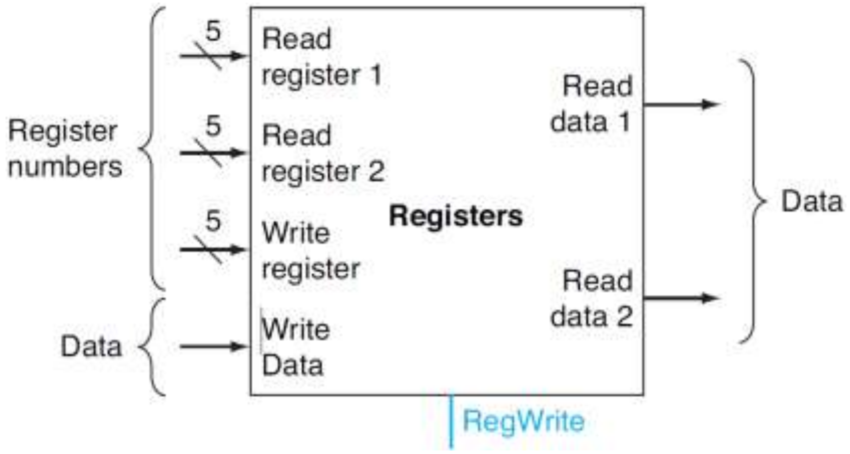
- Instruction execution
  - Fetch instruction from memory
  - Increment PC for the next instruction

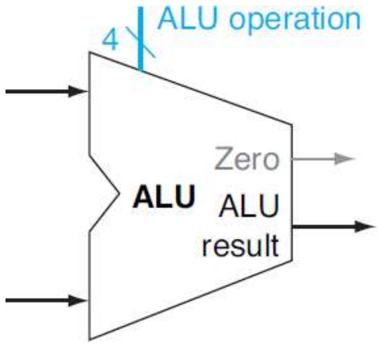


Decoding and Executing Instructions

- R-type instruction
  - Read two registers
  - Perform arithmetic/logical operation
  - Write the result to a register

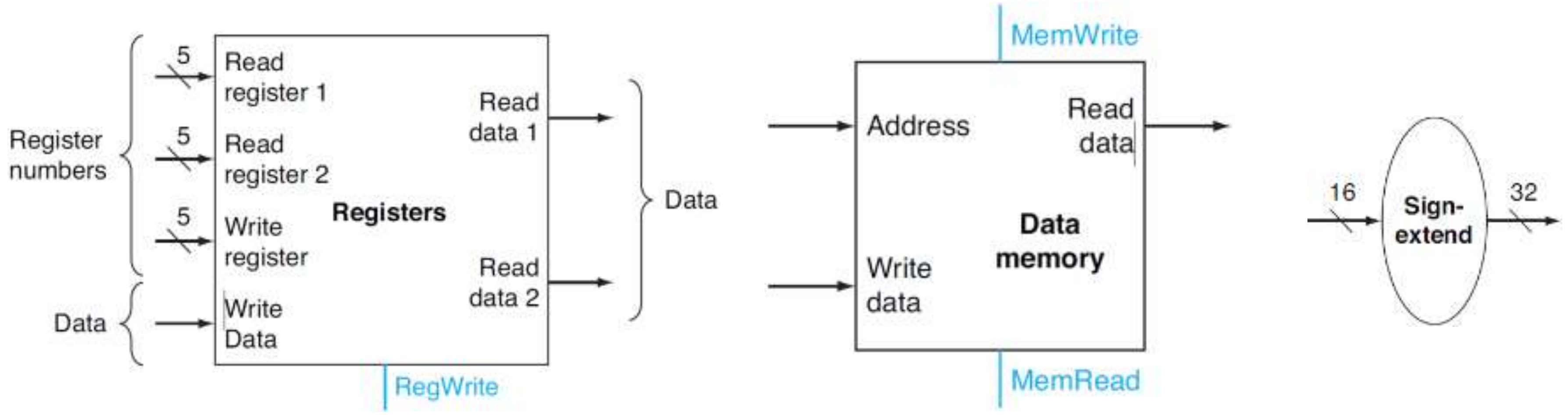
- Register file
  - set of registers that can be accessed by register number (and data)
  - Reading: two register number and two data output
    - Always output data of two register numbers in the read ports
  - Writing: one register number and data to be written
    - controlled by the write control signal





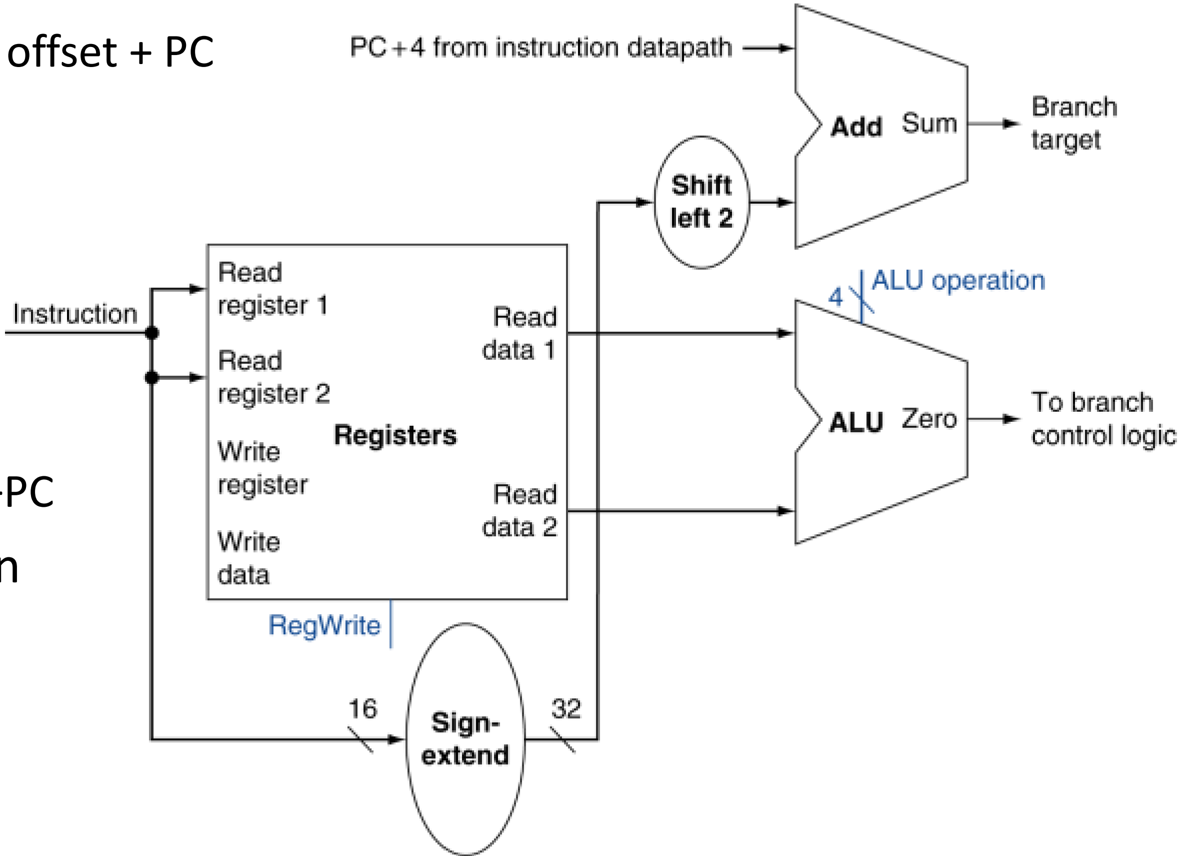
Decoding and Executing Instructions

- Load/Store instructions
  - lw: read from memory, write into the register file
  - sw: read from register, write to memory
  - sign-extend
    - 16-bit offset field to 32-bit signed value



Decoding and Executing Instructions

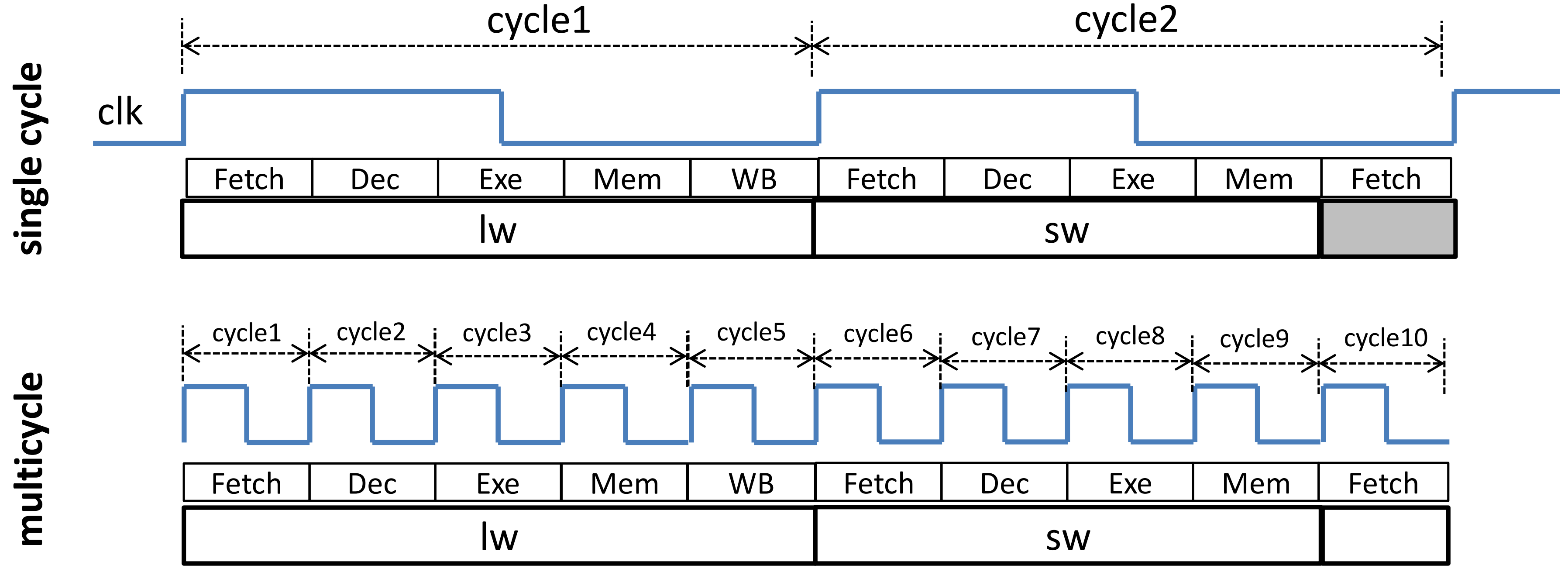
- Branch instructions
  - two registers for comparing
  - branch target address, 16-bit offset
  - Need to compute the branch target address
    - shift left 2 bits of the offset
    - add sign-extended offset + PC



- When branch is taken:
  - Replace the PC to offset+PC
- When branch is not taken
  - PC+4

Creating a Single Datapath

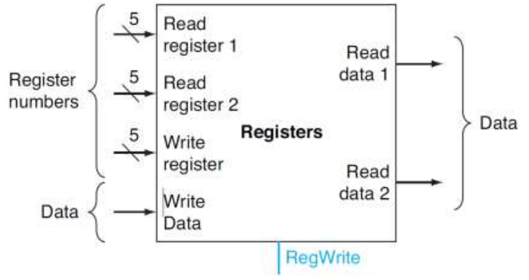
- Execute in one single cycle

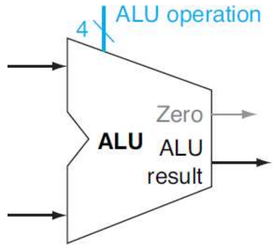


  - No element can be used more than once
    - If needed, it must be duplicated
      - Separate memory for instruction and data memory
- Share elements between instruction types
  - multiple input to the elements
    - multiplexor and control

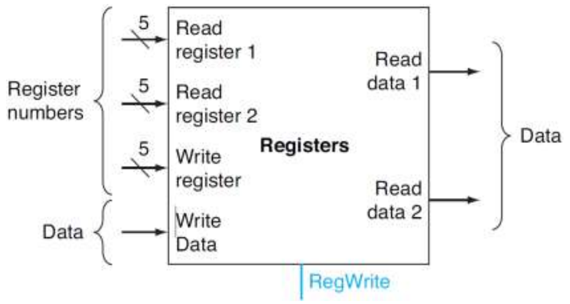
Creating a Single Datapath

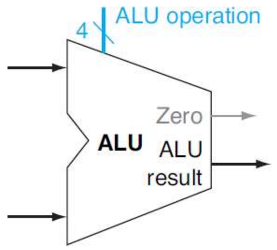
- Supporting the Arithmetic-logical (R-type) instructions
  - add $t3, $t1, $t2

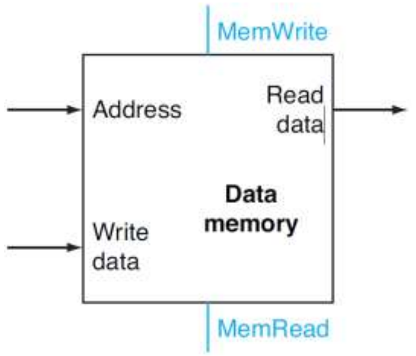


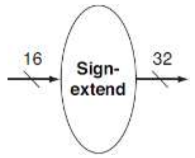


- Supporting the memory (I-type) instructions
  - lw $t1, 0($s1), sw $t1, 0($s1)

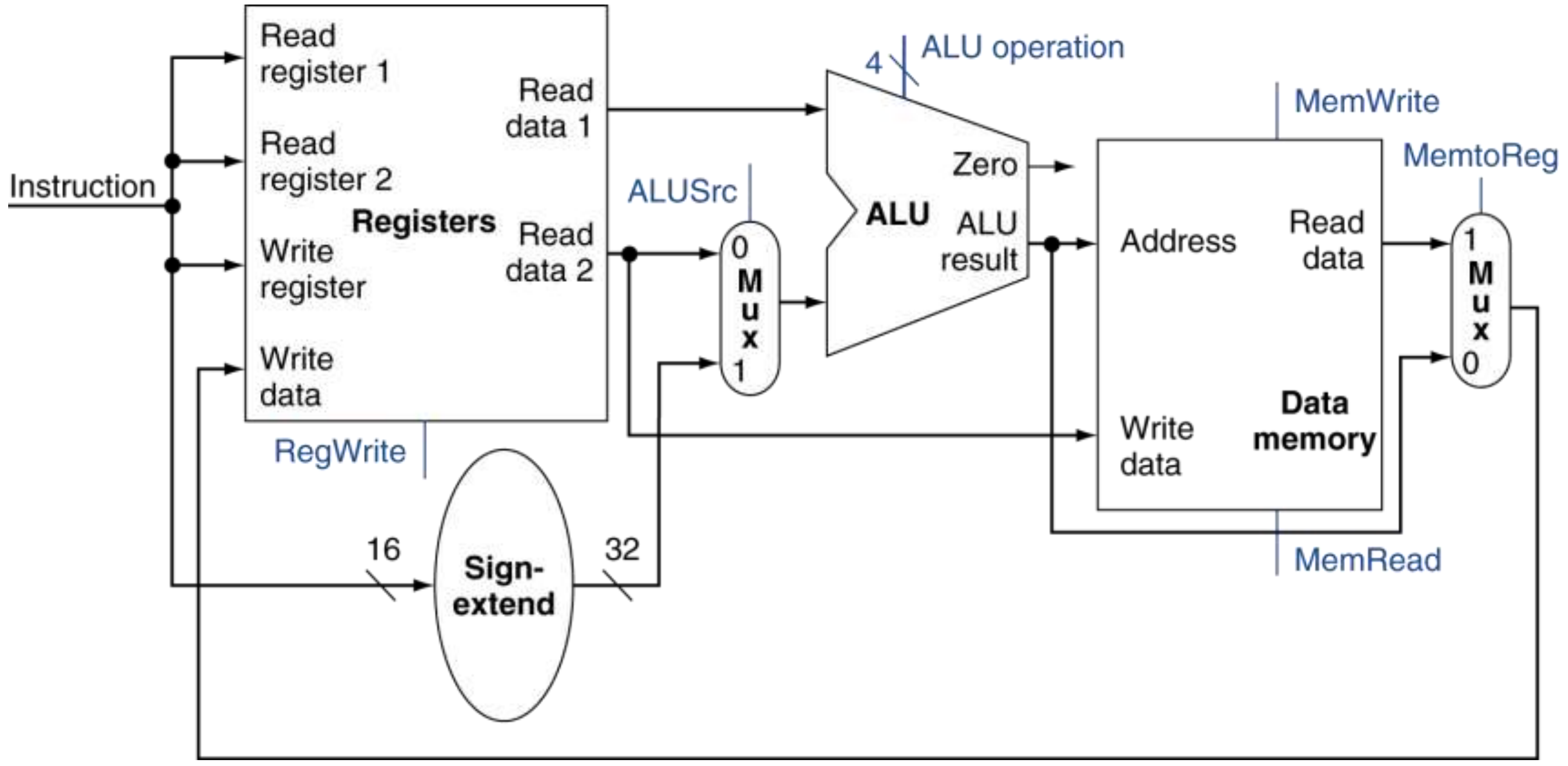






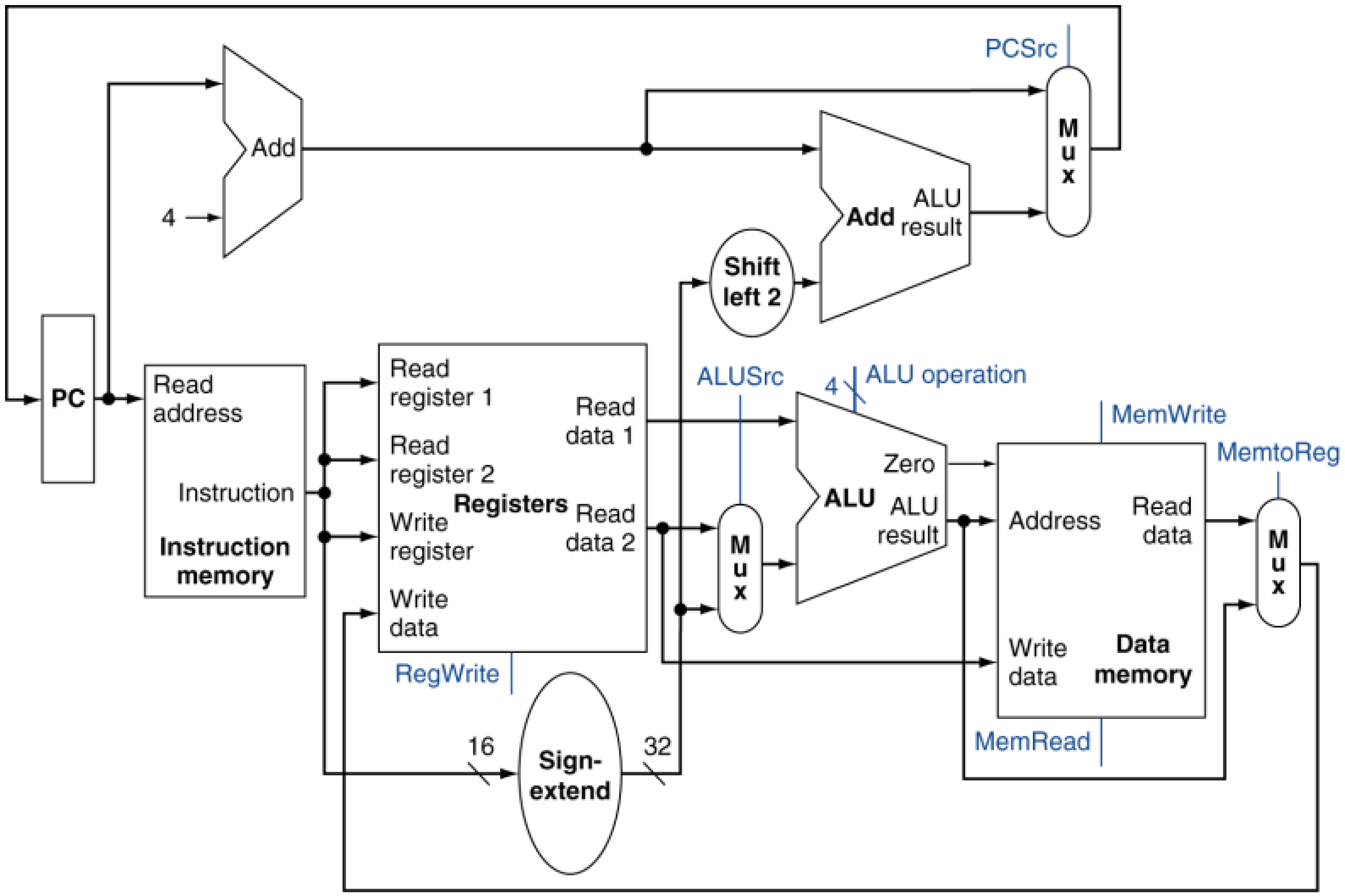


Combined datapath for R-type and lw/sw



Complete Datapath

- R-type + lw/sw + beq



ALU Control

- ALU operations

ALU control	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set-on-less-than
1100	NOR

  - lw, sw: add
  - R-type: depends on the 6-biy funct field
  - branch: subtract

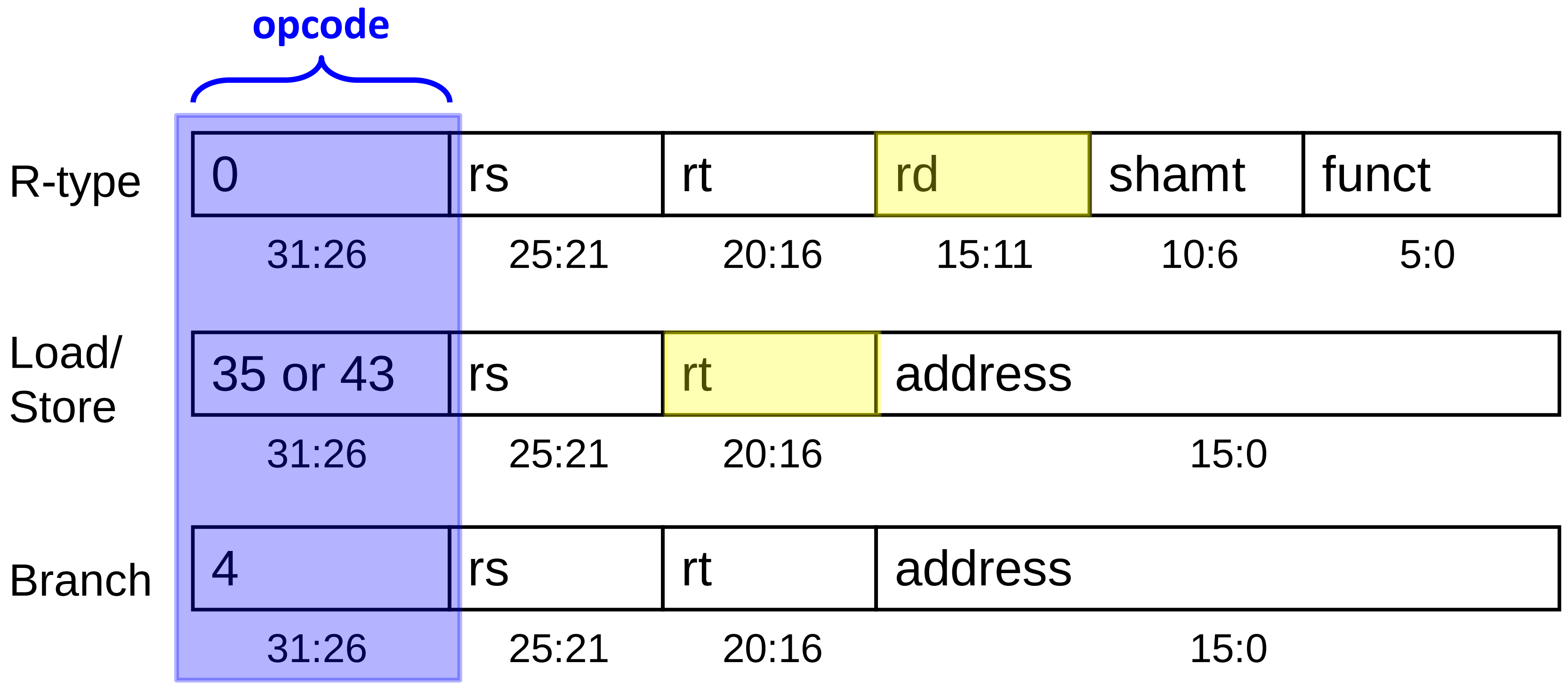
ALU Control

- ALU Control: Small control unit that sends control signal to ALU
  - 2 inputs: ALUOp generated from main ‘control’ unit and funct field from instruction
    - multilevel control
  - output: 4 bit ALU control code

opcode	ALUOp	Operation	funct	ALU function	ALU control
lw	00	load word	XXXXXX	add	0010
sw	00	store word	XXXXXX	add	0010
beq	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
		subtract	100010	subtract	0110
		AND	100100	AND	0000
		OR	100101	OR	0001
		set-on-less-than	101010	set-on-less-than	0111

Main Control Unit

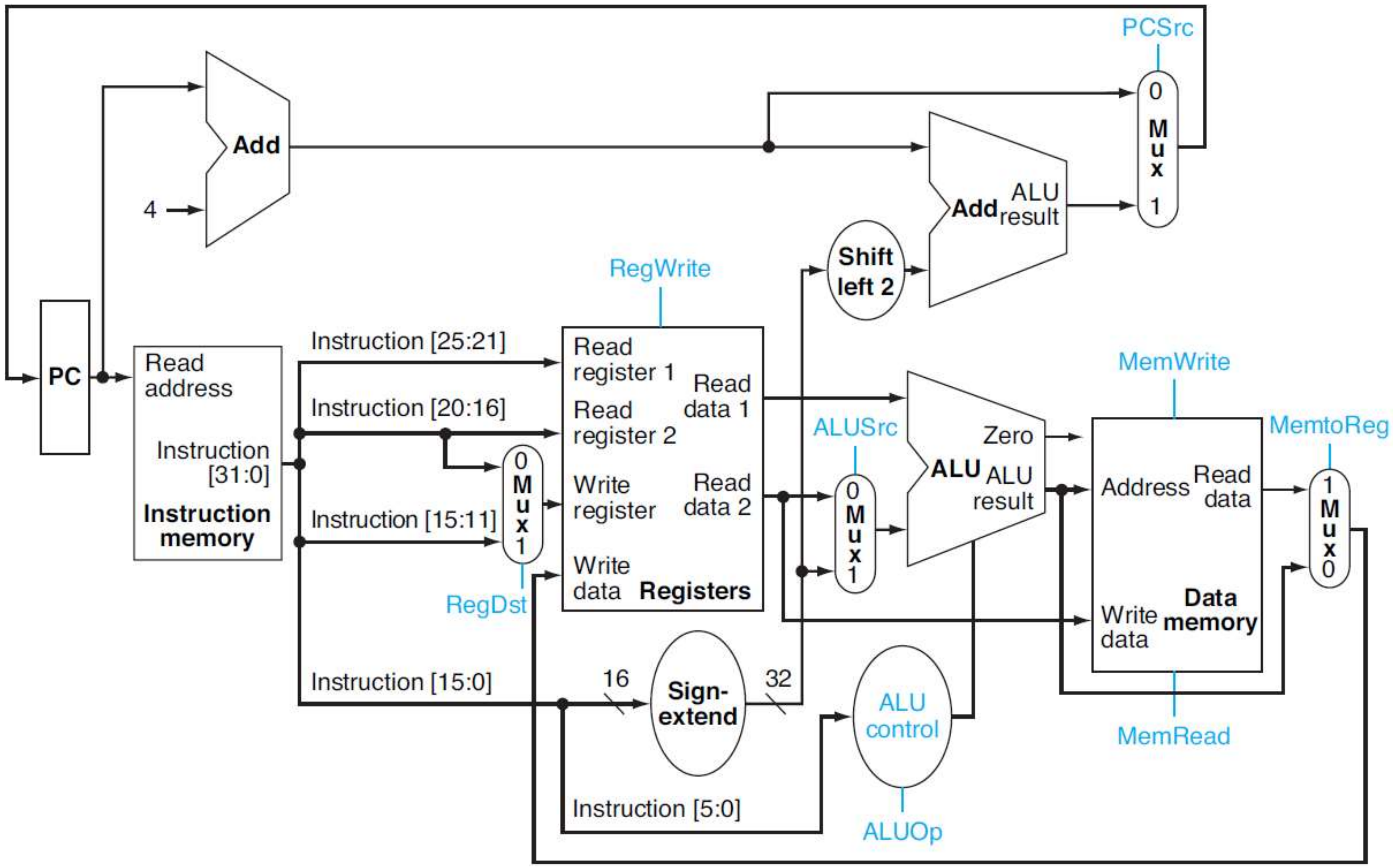
- Deriving control signals from instruction



  - two registers always at rs and rt
  - base register for lw, sw always in rs
  - 16-bit offset always in 15:0
  - destination register – for lw: rt, for R-type: rd

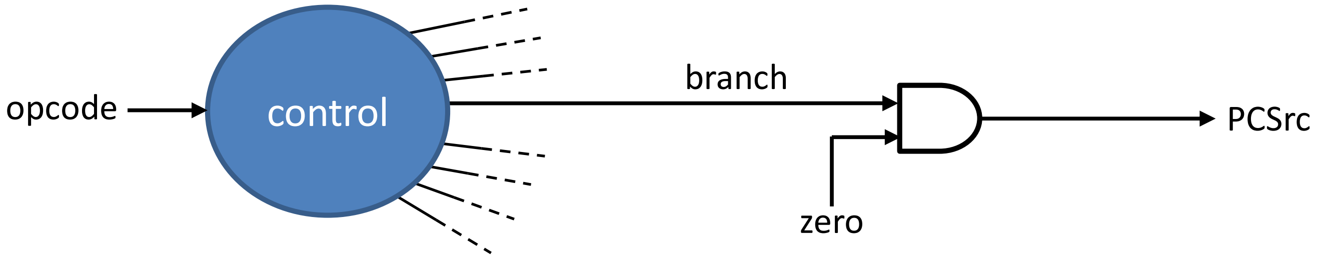
Updated Datapath

- All necessary mux and control lines identified



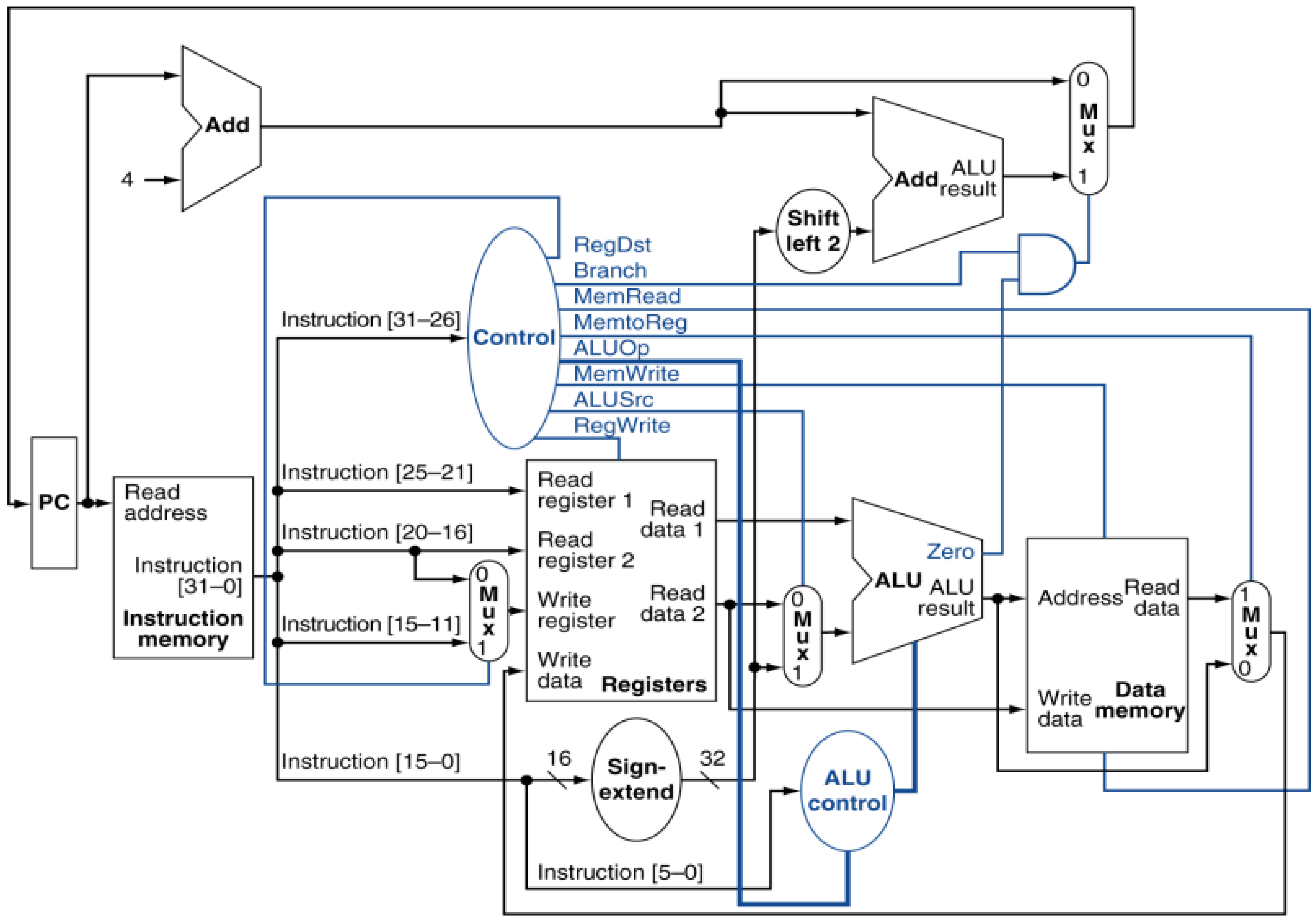
Main Control Unit

- Total 7 single-bit control and 2-bit ALUop control signal
- All (except one) controls can be derived from the instruction (opcode)
  - Exception: PCSrc
    - PCSrc: set when it is the beq instruction and 0 output from ALU
    - Need to 'AND' signal from control and the zero from ALU

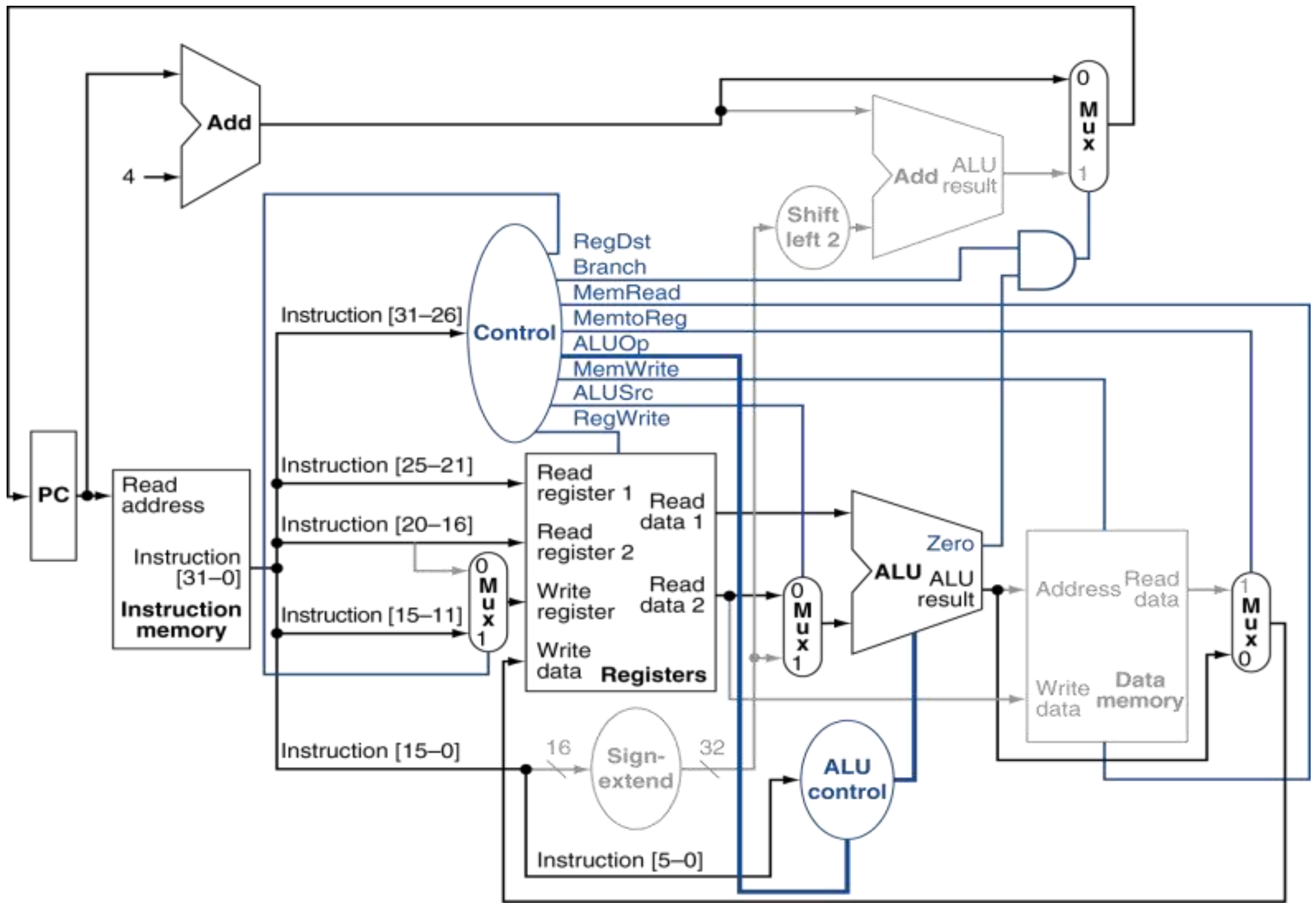


Instruction	RegDst	ALUSrc	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

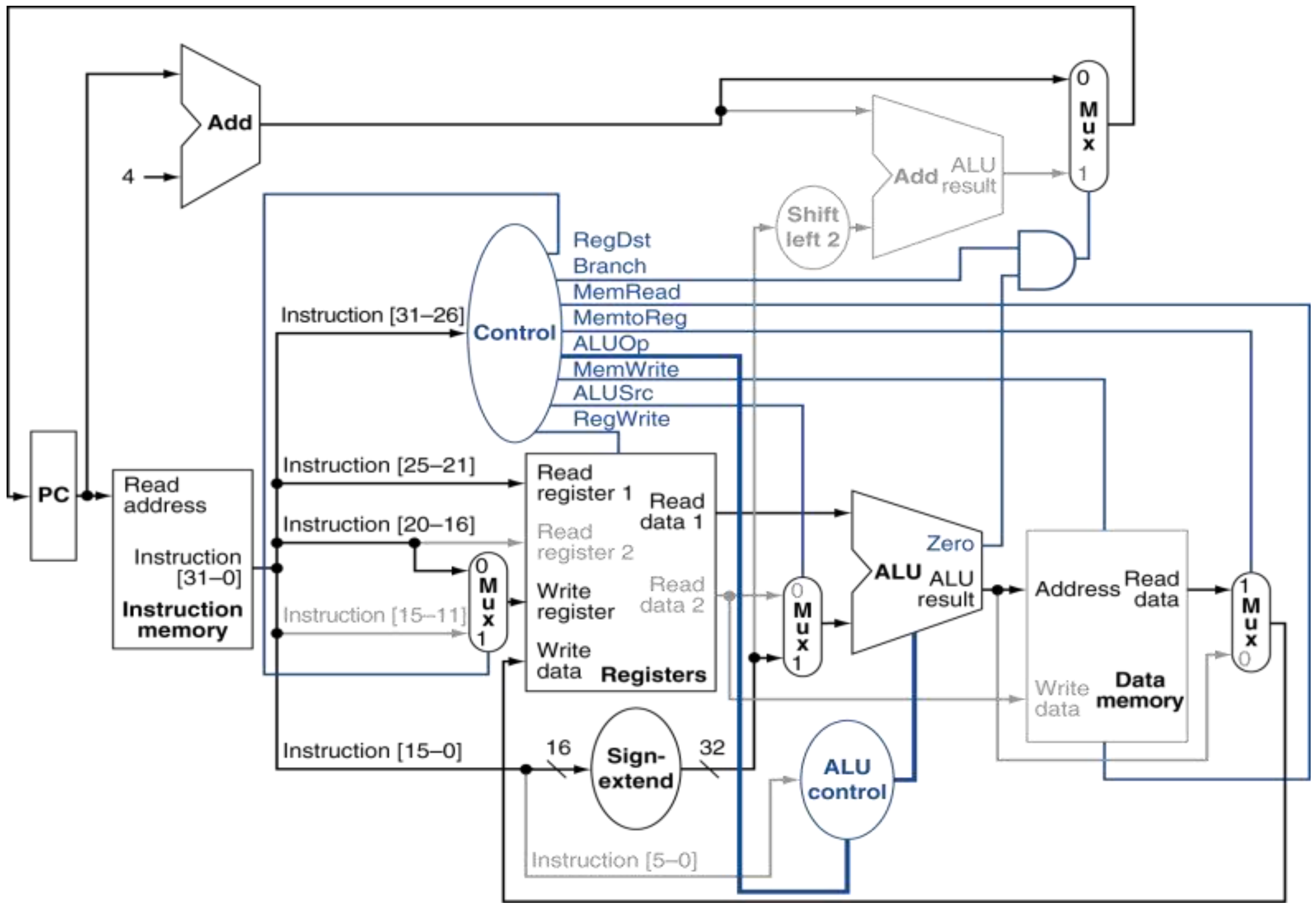
Datapath with Full Control



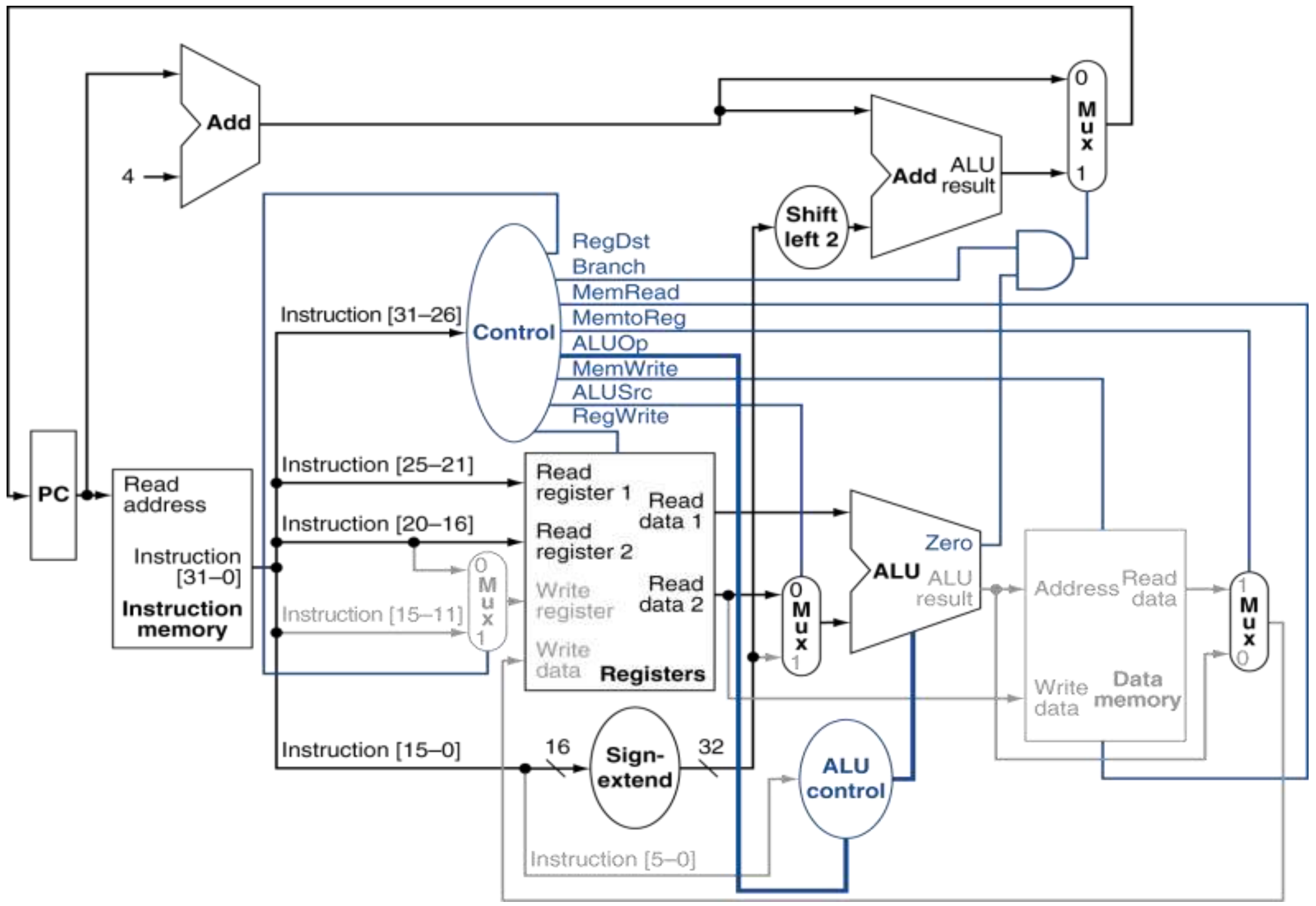
Datapath Operation for R-type



Datapath Operation for Load



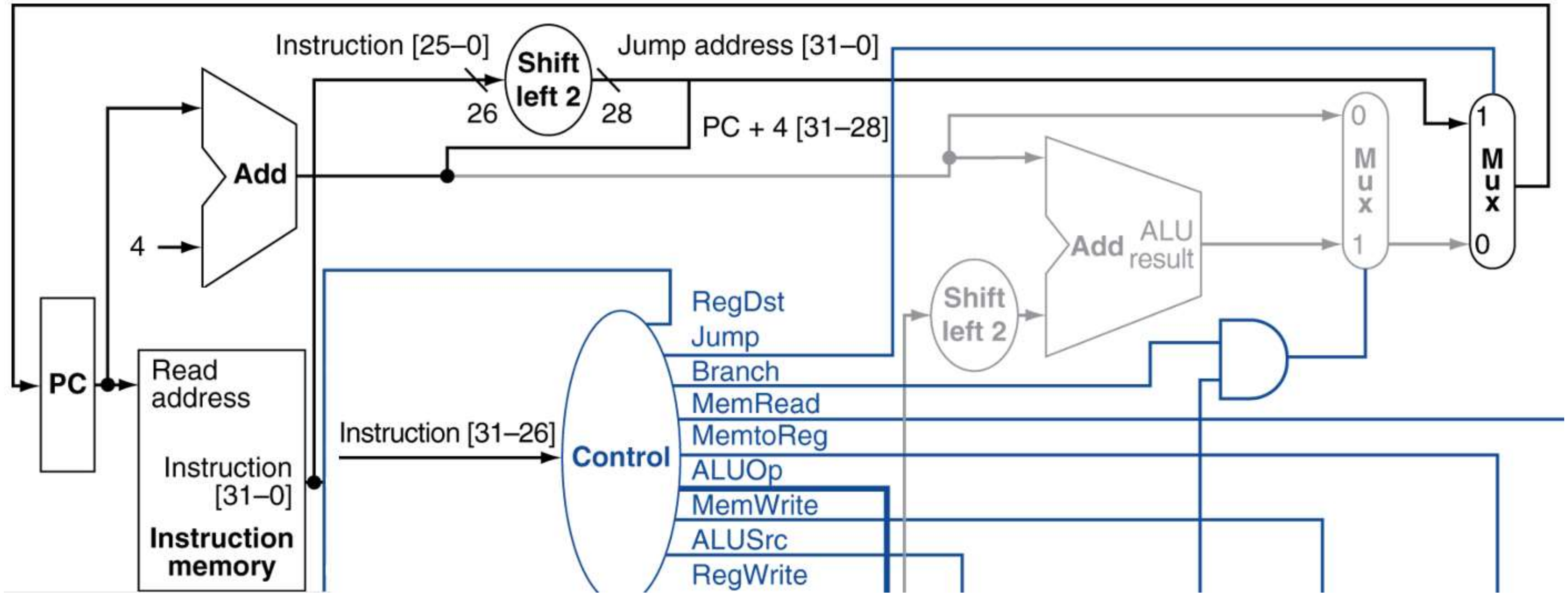
Datapath Operation for Branch



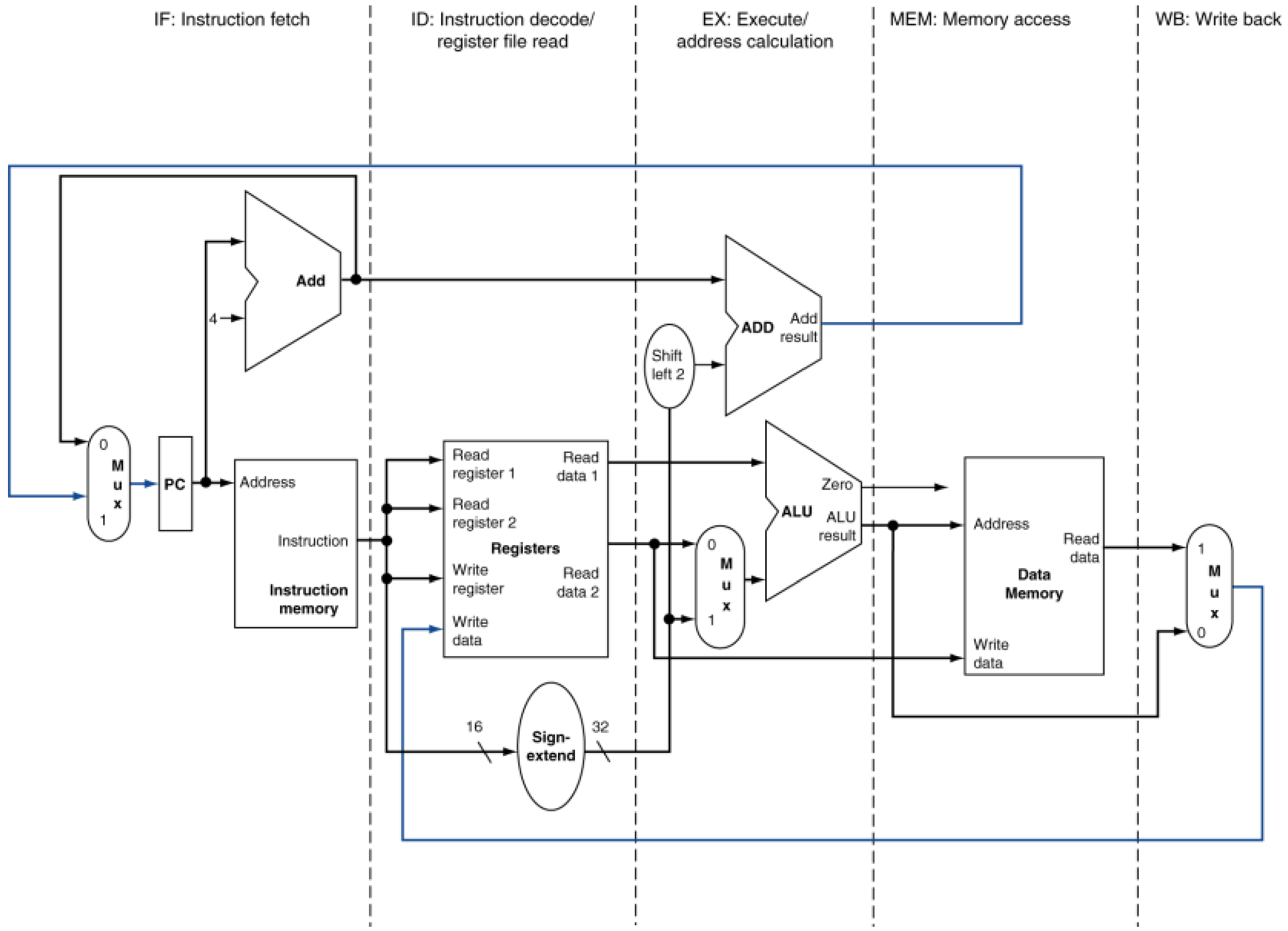
Handling Jump Instruction

Jump	2	address
	31:26	25:0

- Constructing complete 32 bit address
  - 4 bits from PC + 26 bits + 00

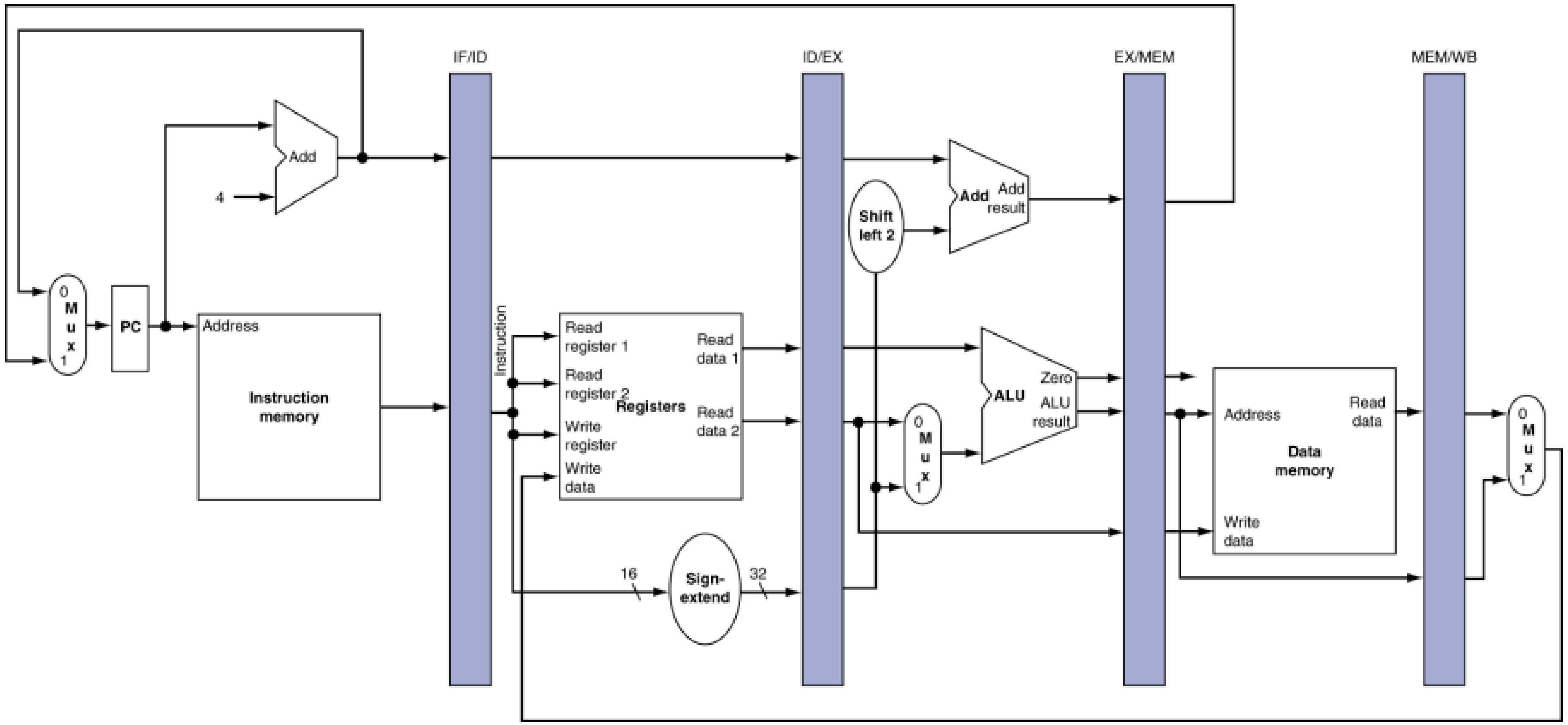


Instruction Execution Stages



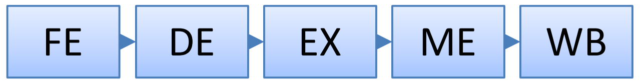
Instruction Execution - Multicycle

- Registers between Stages

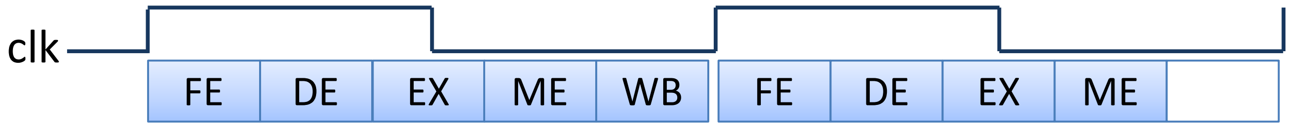


Pipelining

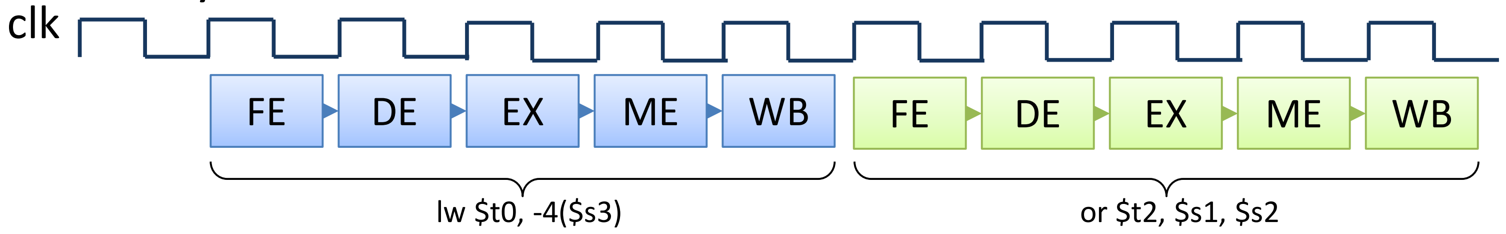
- Instruction Execution Steps



- Single cycle execution

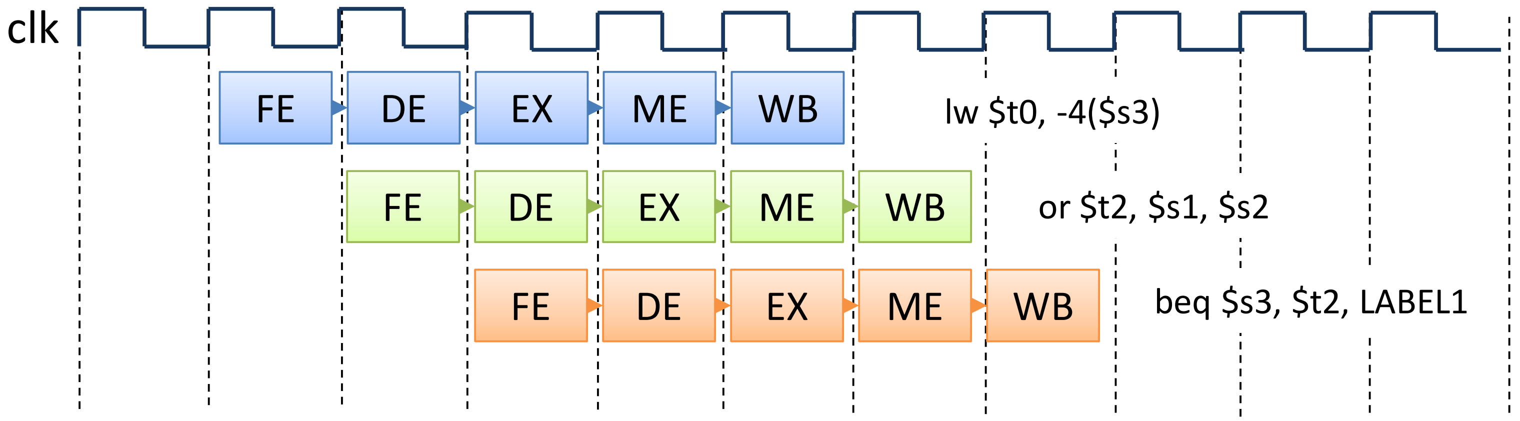


- Multi cycle execution



  - Each step uses only part of the CPU H/W (components)

- Pipelined execution



Pipeline Performance

- Performance improvement
  - Theoretic throughput improvement = # of pipeline stages
- Performance comparison

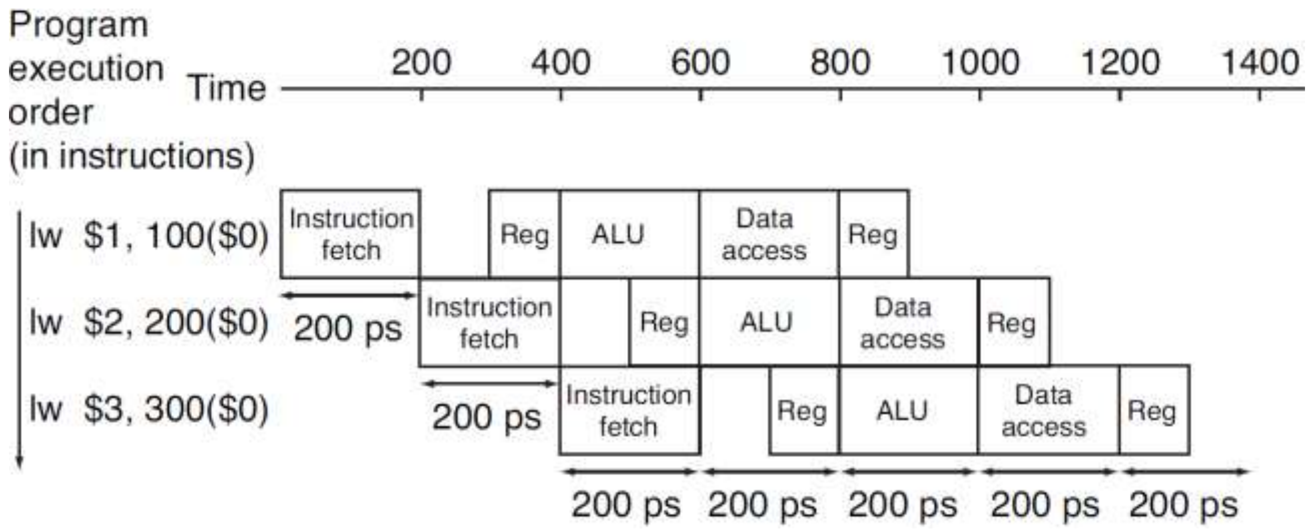
Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

  - In the single-cycle design
    - Cycle must be long enough to allow for the slowest instruction: lw = 800ps
    - 3 lw instructions: 800ps x 3 = 2400ps = 2.4 ns
  - In the pipelined design
    - Cycle must be long enough to allow for the slowest stage: 200 ps
    - 200ps x 3 = 600ps

Pipeline Performance

Speed up계산시 Single cycle 버전과 비교하려면 800과 비교해야되는 것 아닌가? 1000 아니고?
그럼 이론적속도향상5배는 안나오는데

- Theoretic speed up upper bound
  - ideal case: equal stage length
  - no pipeline overhead
- In the example of 3 lw instructions,
  - total pipelined time = 1000ps + 200ps + 200ps = 1400ps



  - With more instructions, true speed up is reached
- Pipelining increases the instruction throughput

Pipeline Hazards

- Hazards
  - In pipeline, there is a situation in which the next instruction cannot execute in the next cycle
- Type of hazards
  - Structural Hazards
    - H/w cannot support specific combination of instructions
    - resource conflict
      - ex) what if there was only one memory unit?
  - Data Hazards
    - Data needed in the following instruction is not ready

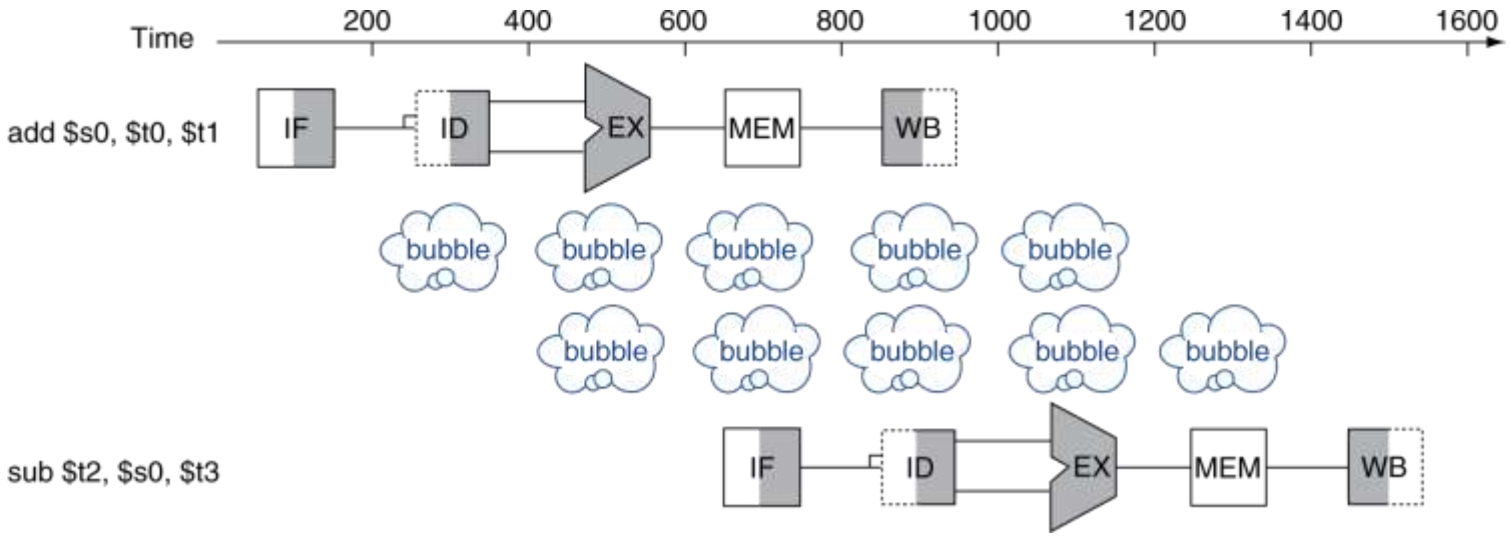
```
add $s0, $t0, $t1
sub $t2, $s0, $t3
```

  - Control Hazards
    - branch hazards

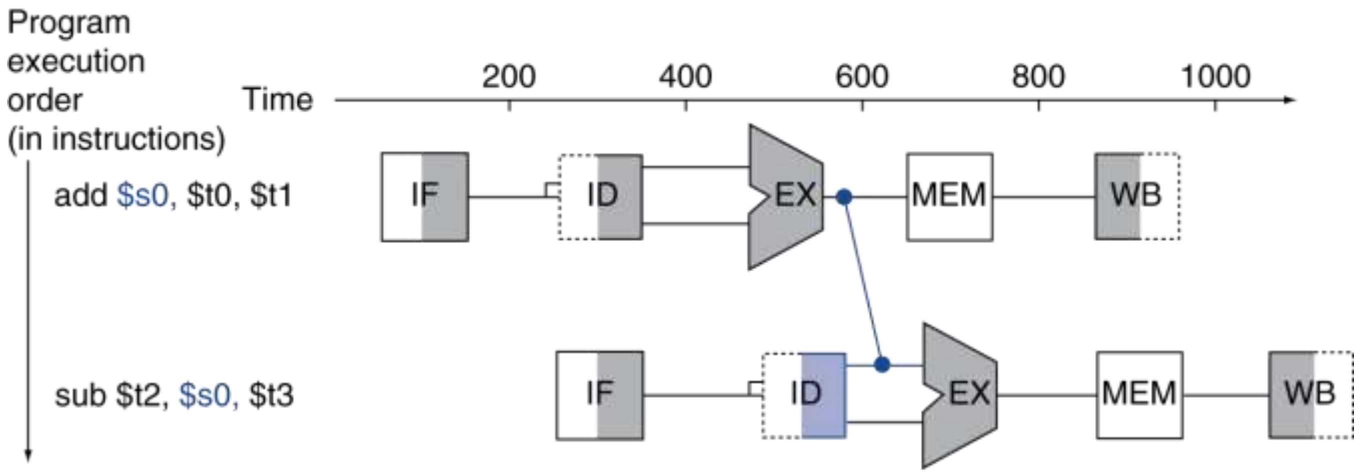
Data Hazards

- Instruction depends on the result of previous instruction

```
add $s0, $t0, $t1
sub $t2, $s0, $t3
```

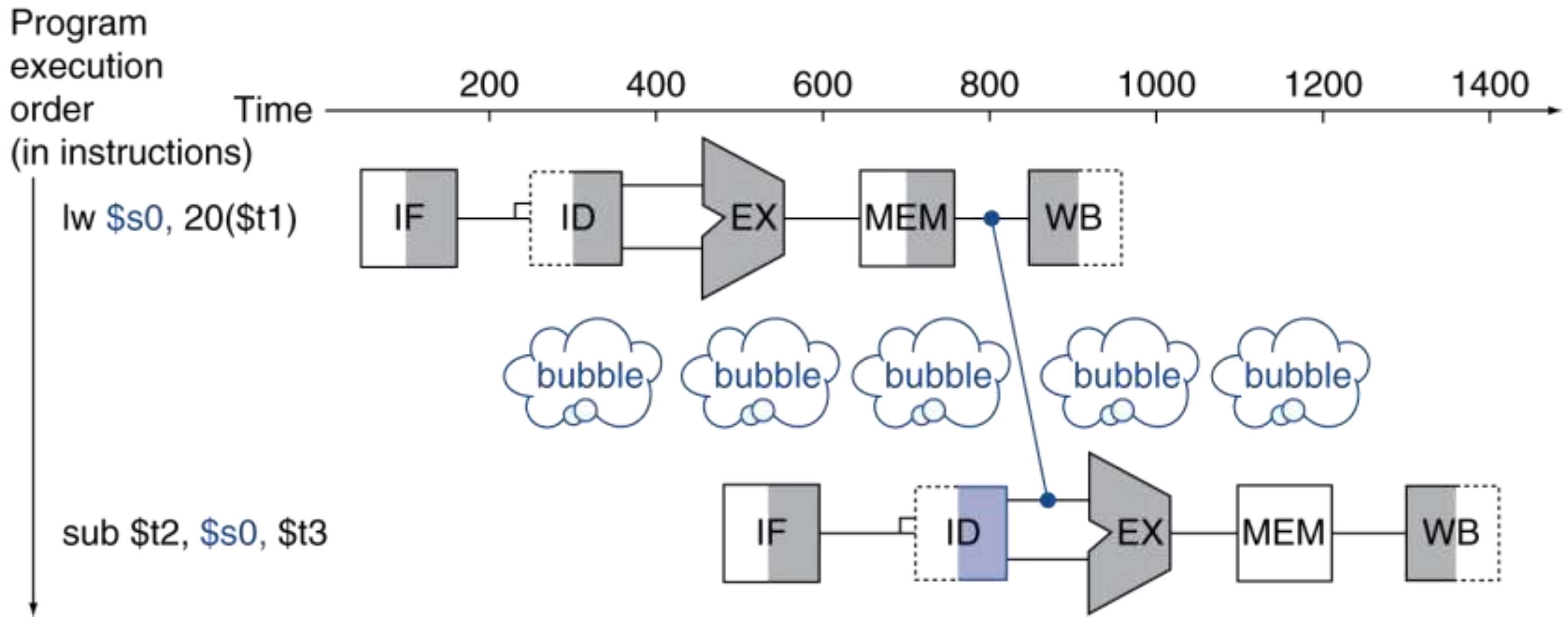


- Forwarding



Data Hazards

- load-use data hazard
  - Even with forwarding, pipeline stall (bubble) is unavoidable



  - What else can we do about the load-use data hazard?

Data Hazards

- Code Reordering

a = b + e ;
c = b + f ;

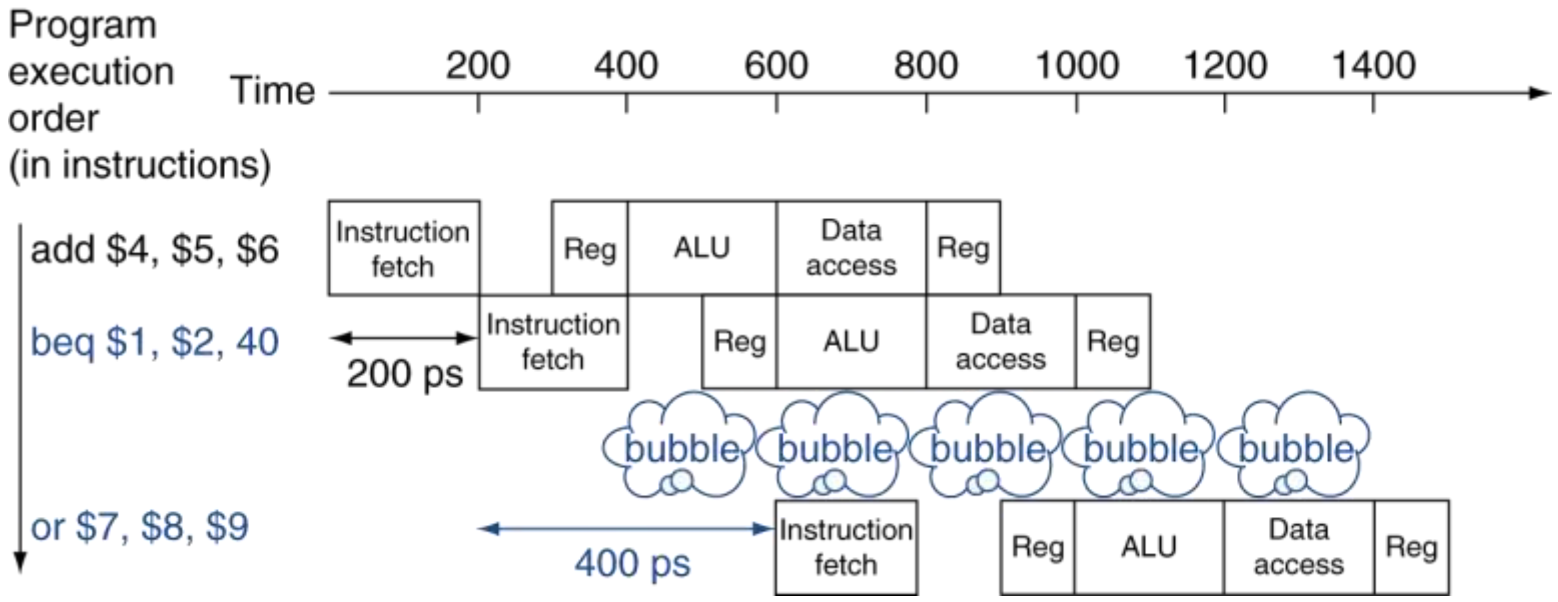
```
        lw $t1, 0($t0)
        lw $t2, 4($t0)
hazard
        add $t3, $t1,$t2
        sw $t3, 12($t0)
        lw $t4, 8($t0)
hazard
        add $t5, $t1,$t4
        sw $t5, 16($t0)
```

```
lw $t1, 0($t0)
lw $t2, 4($t0)
lw $t4, 8($t0)
add $t3, $t1,$t2
sw $t3, 12($t0)
add $t5, $t1,$t4
sw $t5, 16($t0)
```

Control Hazards

- Hazard in which next instruction is not known until the EX stage of current instruction is reached
  - branch instruction
  - Unable to fetch the correct instruction
- Solutions

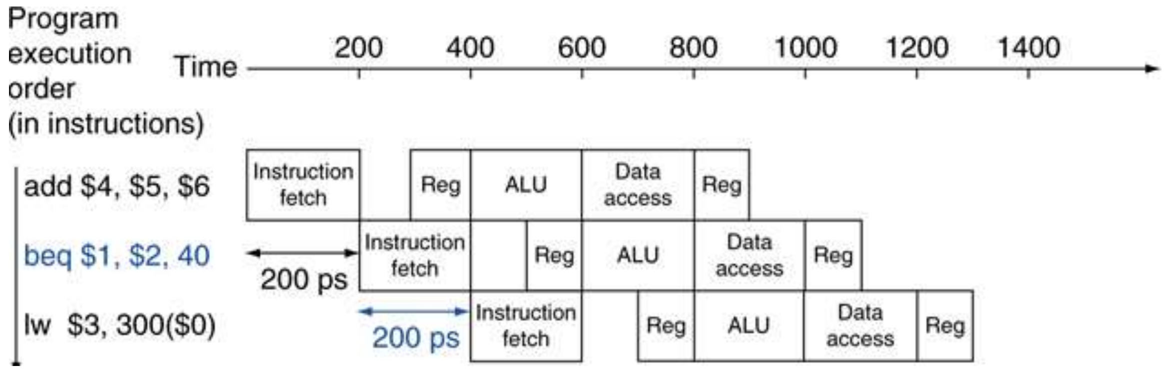
  (i) stall (ii) extra H/W (iii) any other option?



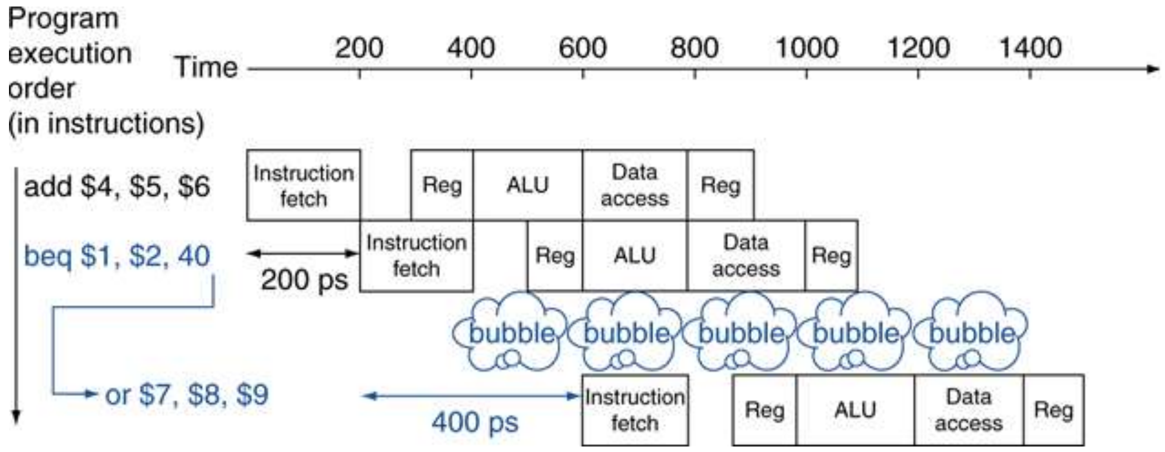
Control Hazards

- Branch Prediction
  - Stall only when prediction is wrong
  - Simple Prediction ↔ Dynamic Prediction
    - Predict always as branch not-taken/taken

Correct Prediction

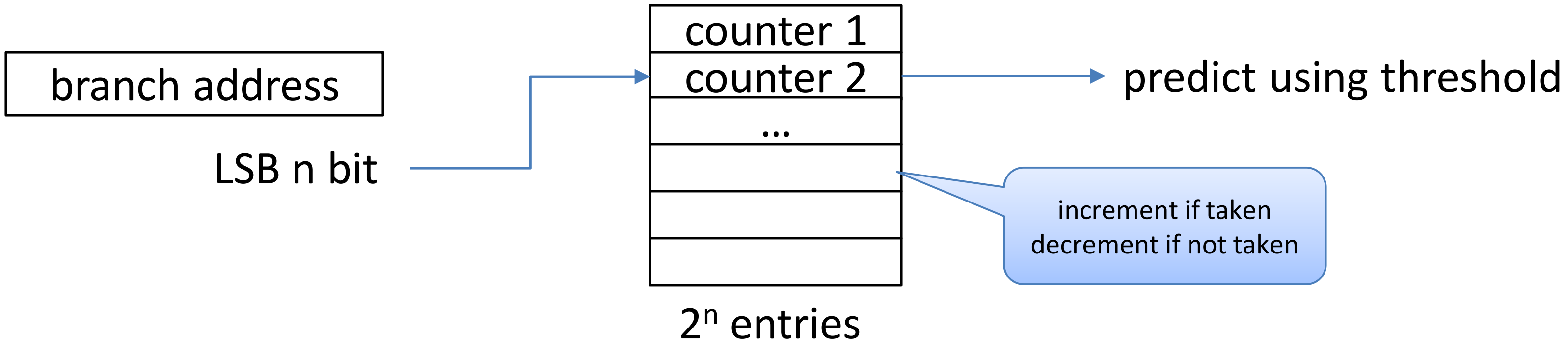


Incorrect Prediction



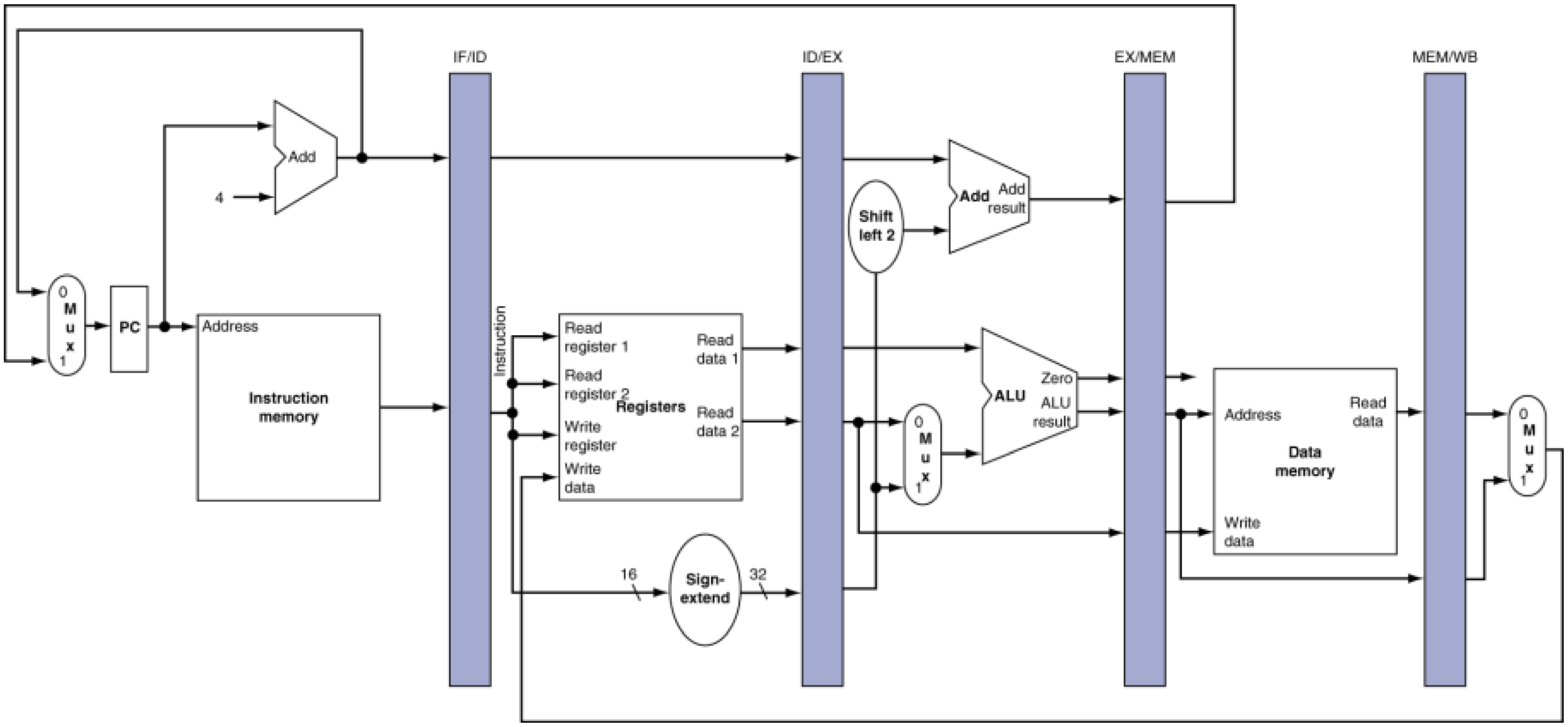
Control Hazards

- Branch instruction: 20% on average
- Static branch prediction
  - Uses information gathered before execution
  - Ex) bottom of the loop: more likely that branch will be taken
  - It relies only on the typical behavior
    - Does not differentiate branch instructions
- Dynamic branch prediction
  - Uses information gathered at runtime
  - Keep a history of recent branch decision
    - Modern CPU predicts with 90% accuracy

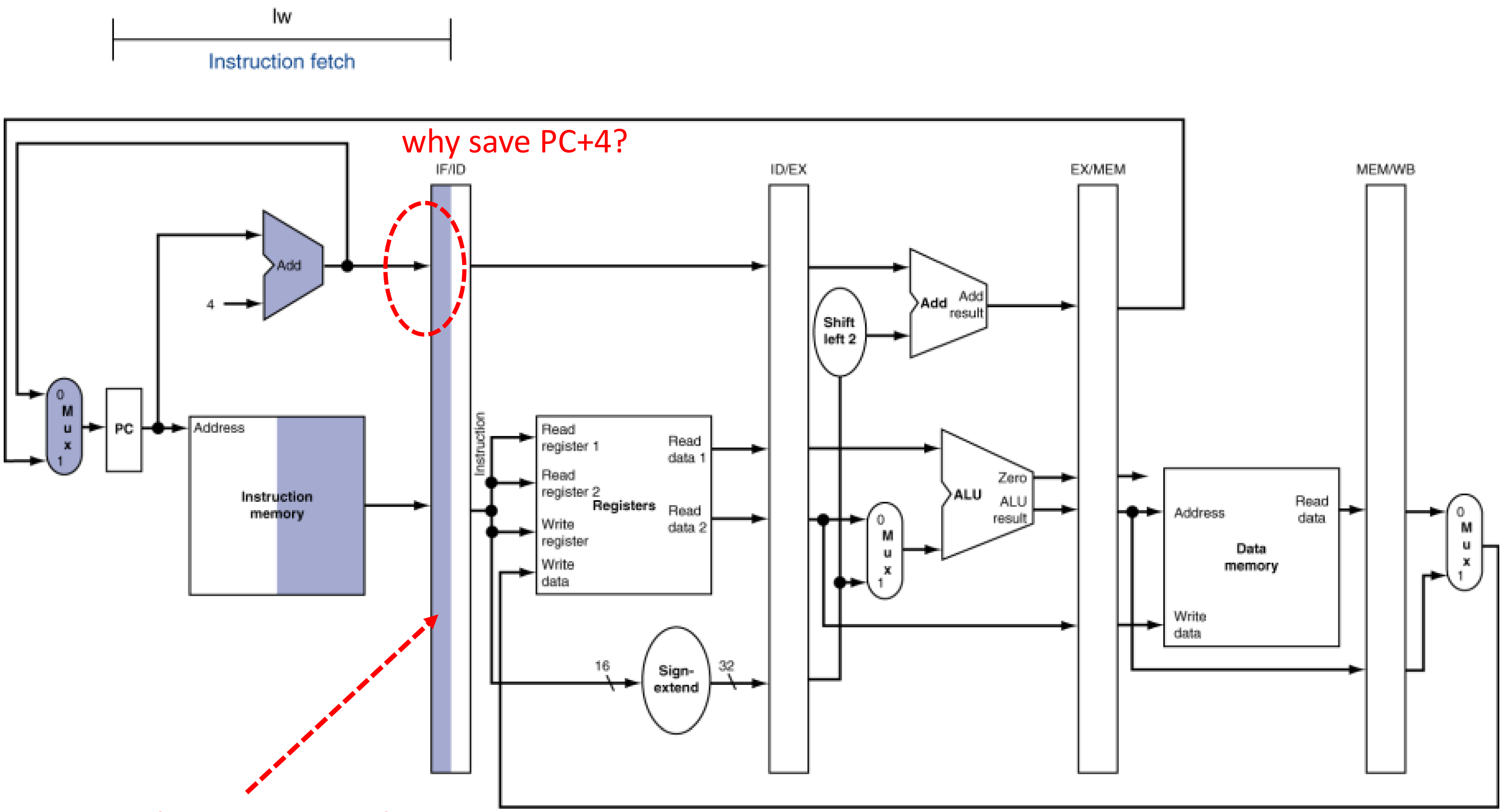


Pipelined Datapath

- Pipeline registers between Stages
- Pipeline registers hold information produced in previous cycle



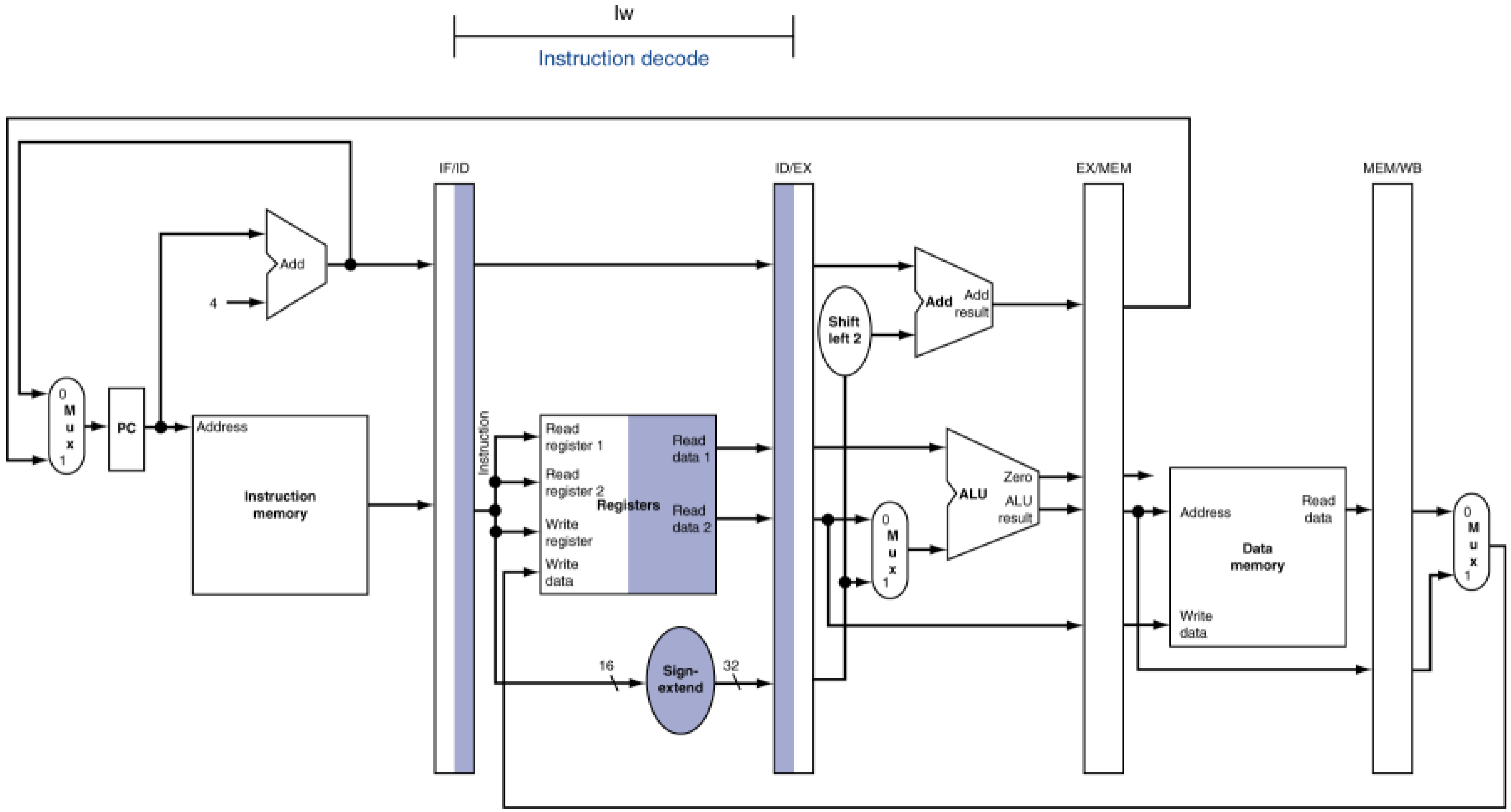
IF stage for LW instruction



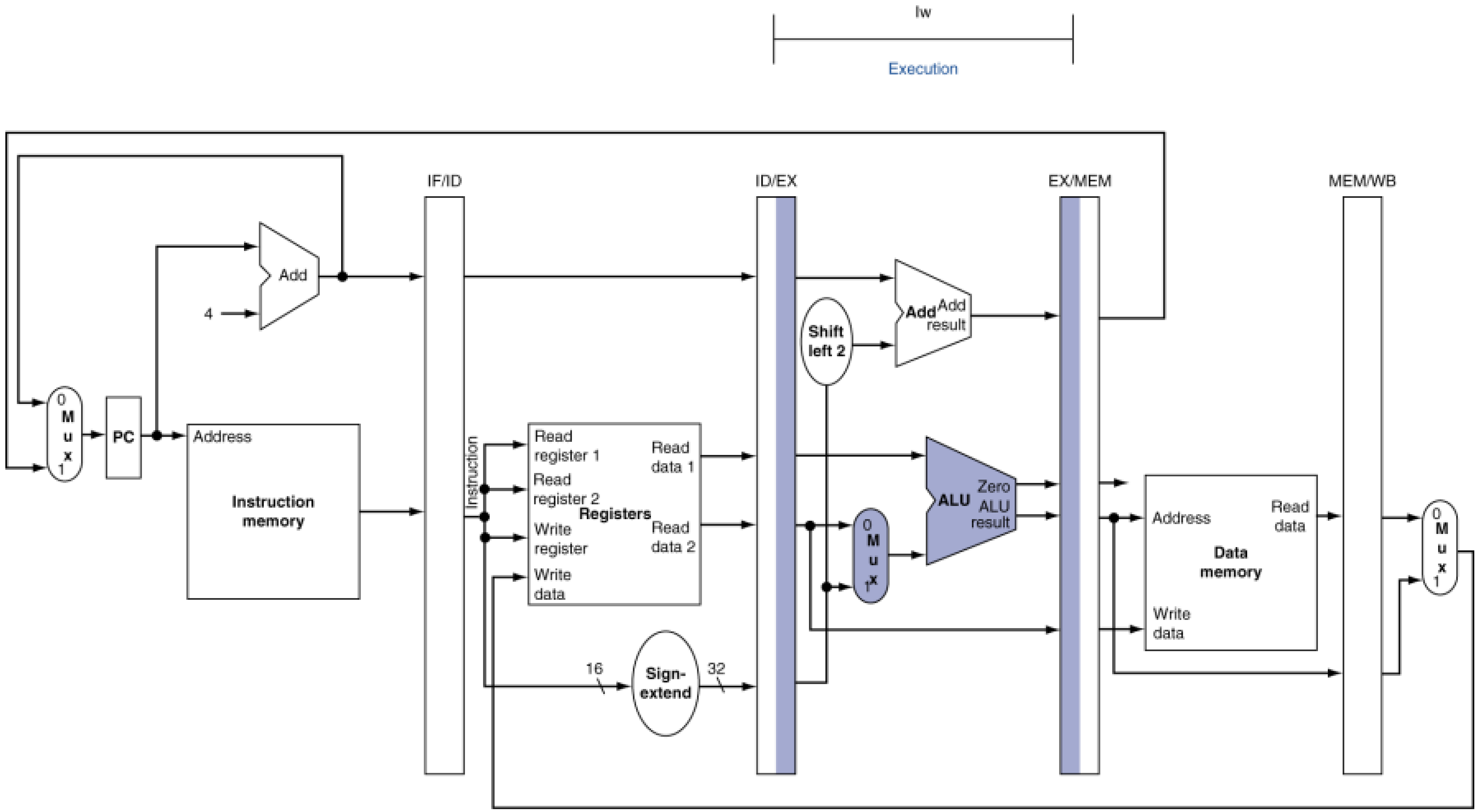
why save PC+4?

Instruction not yet written to the pipeline register

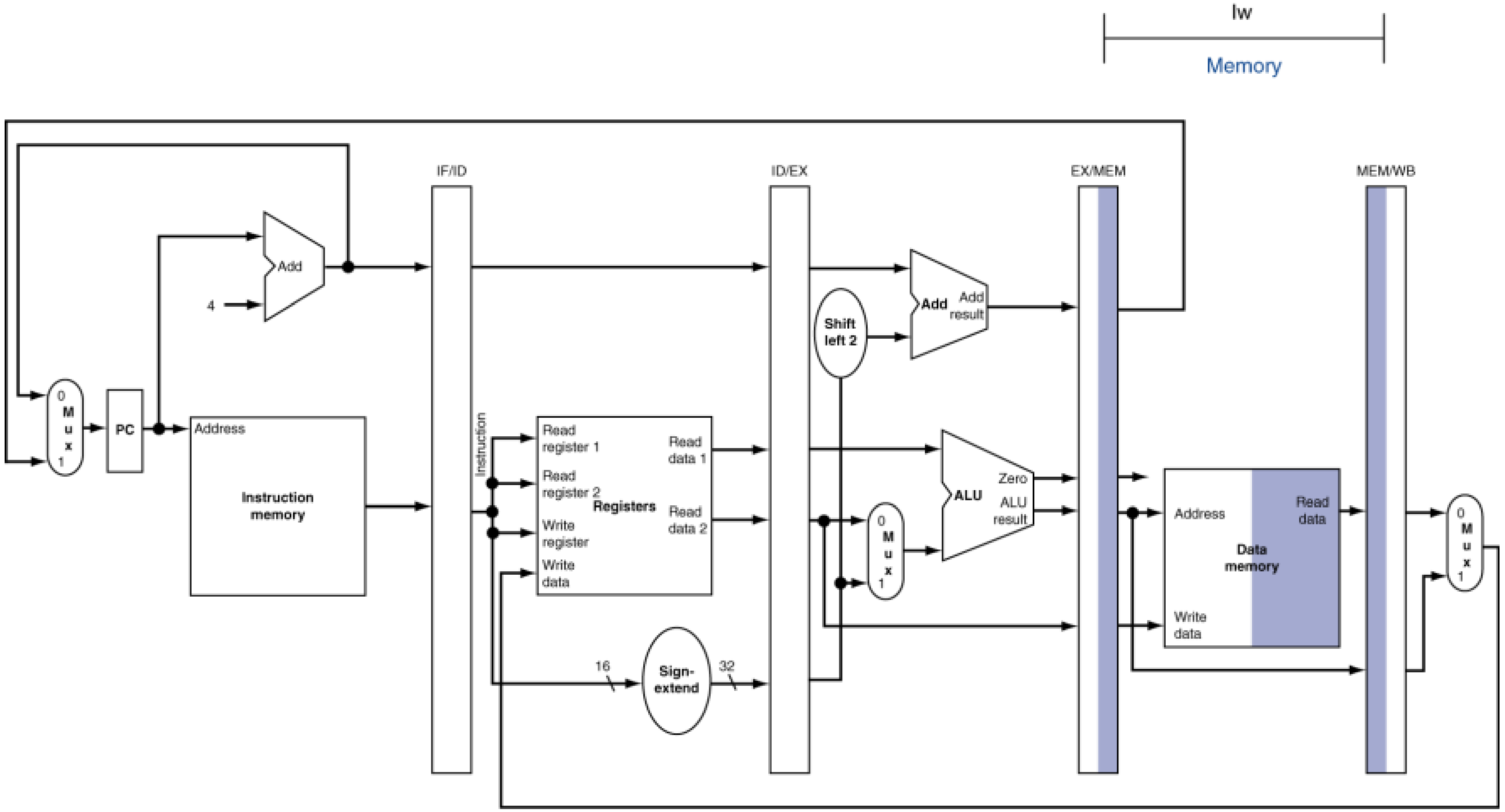
ID stage for LW instruction



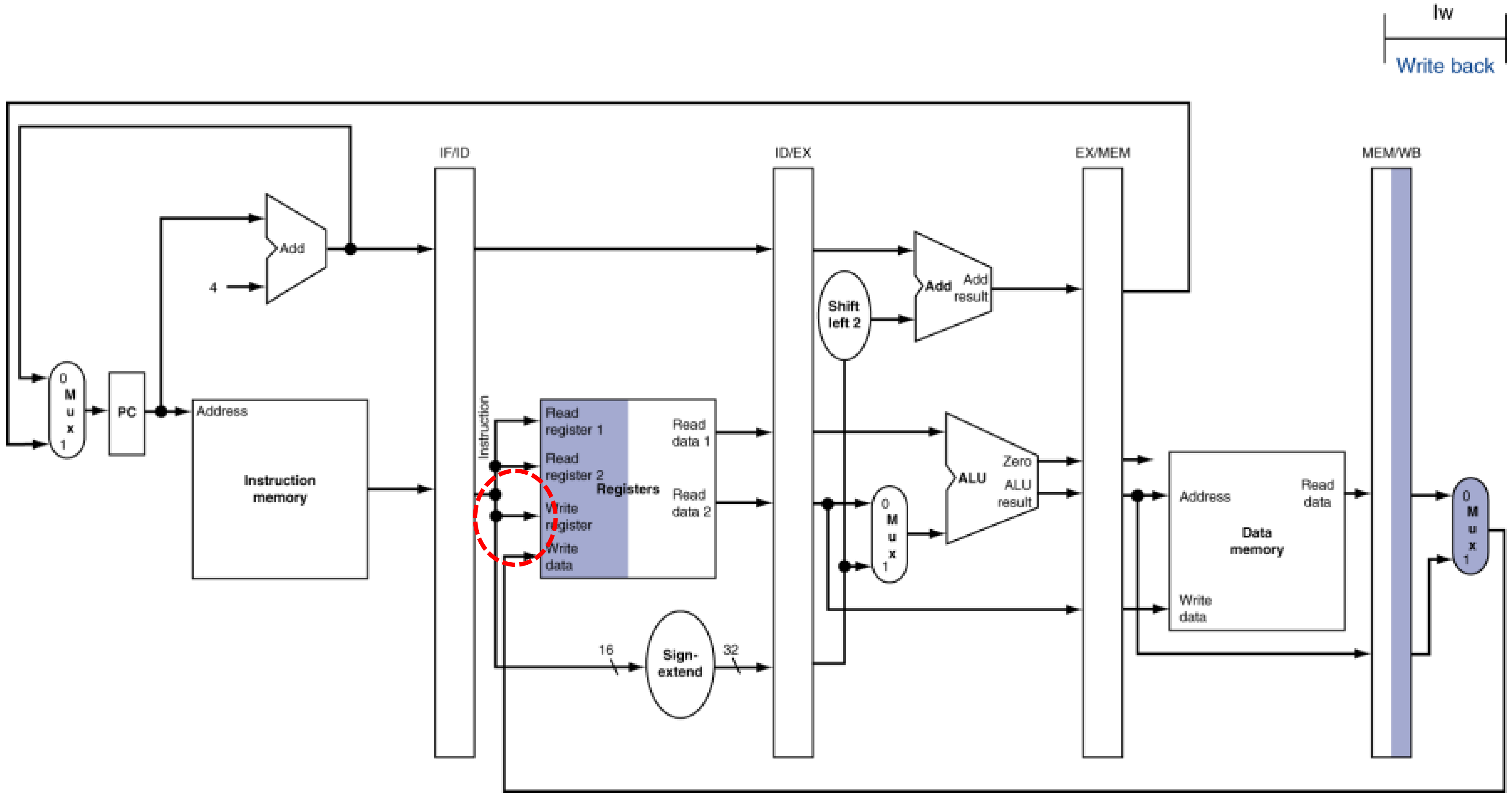
EX stage for LW instruction



MEM stage for LW instruction

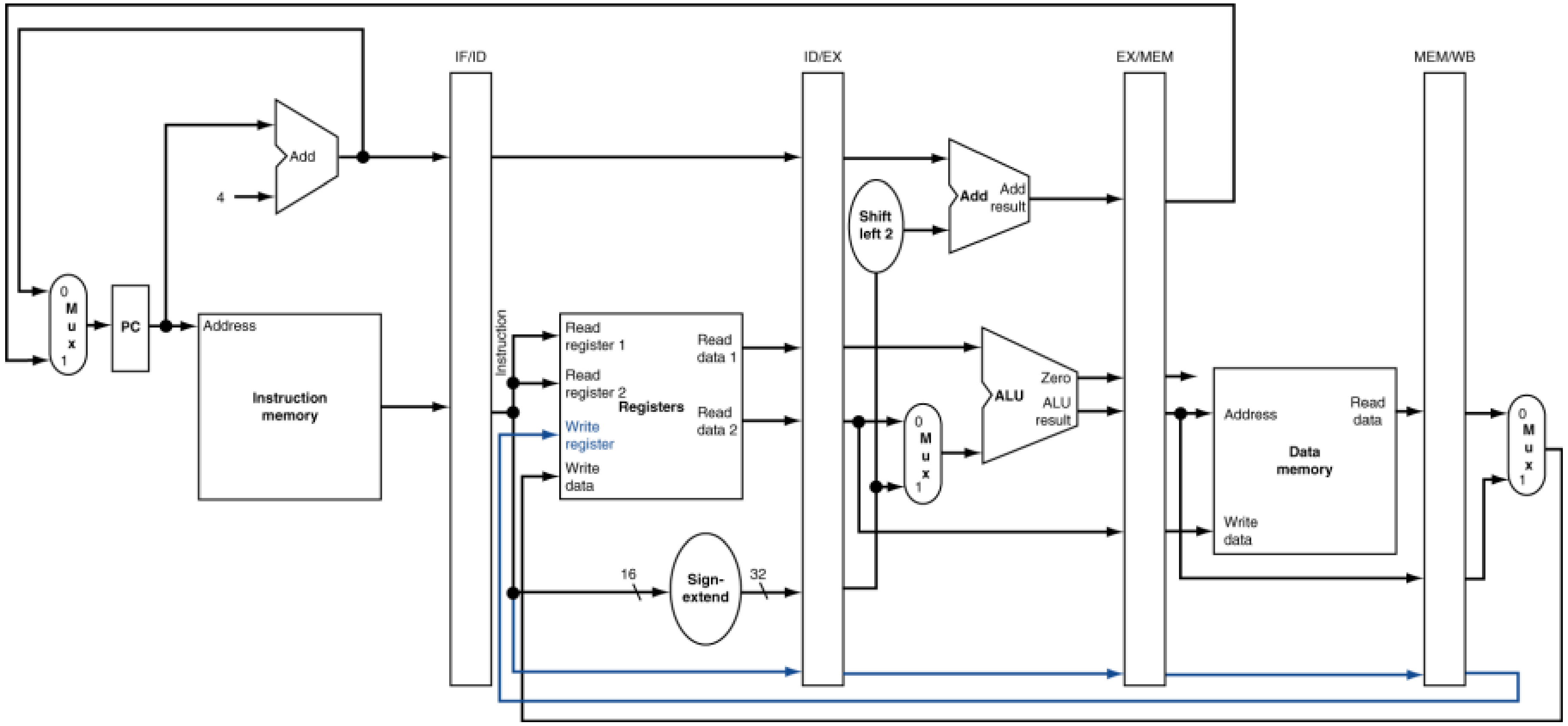


WB stage for LW instruction (with Error)



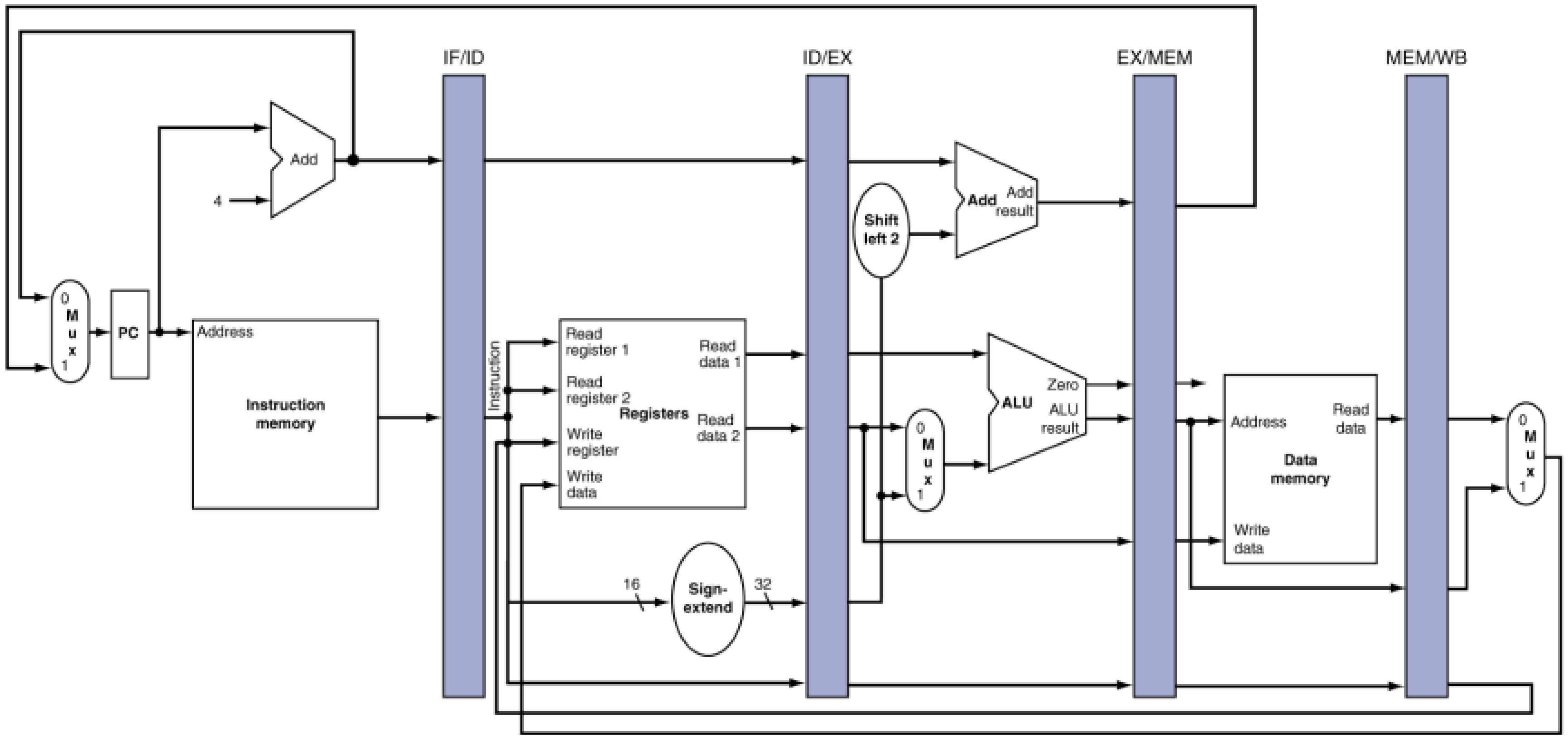
Correct Datapath

- Write register number must come from MEM/WB pipeline register

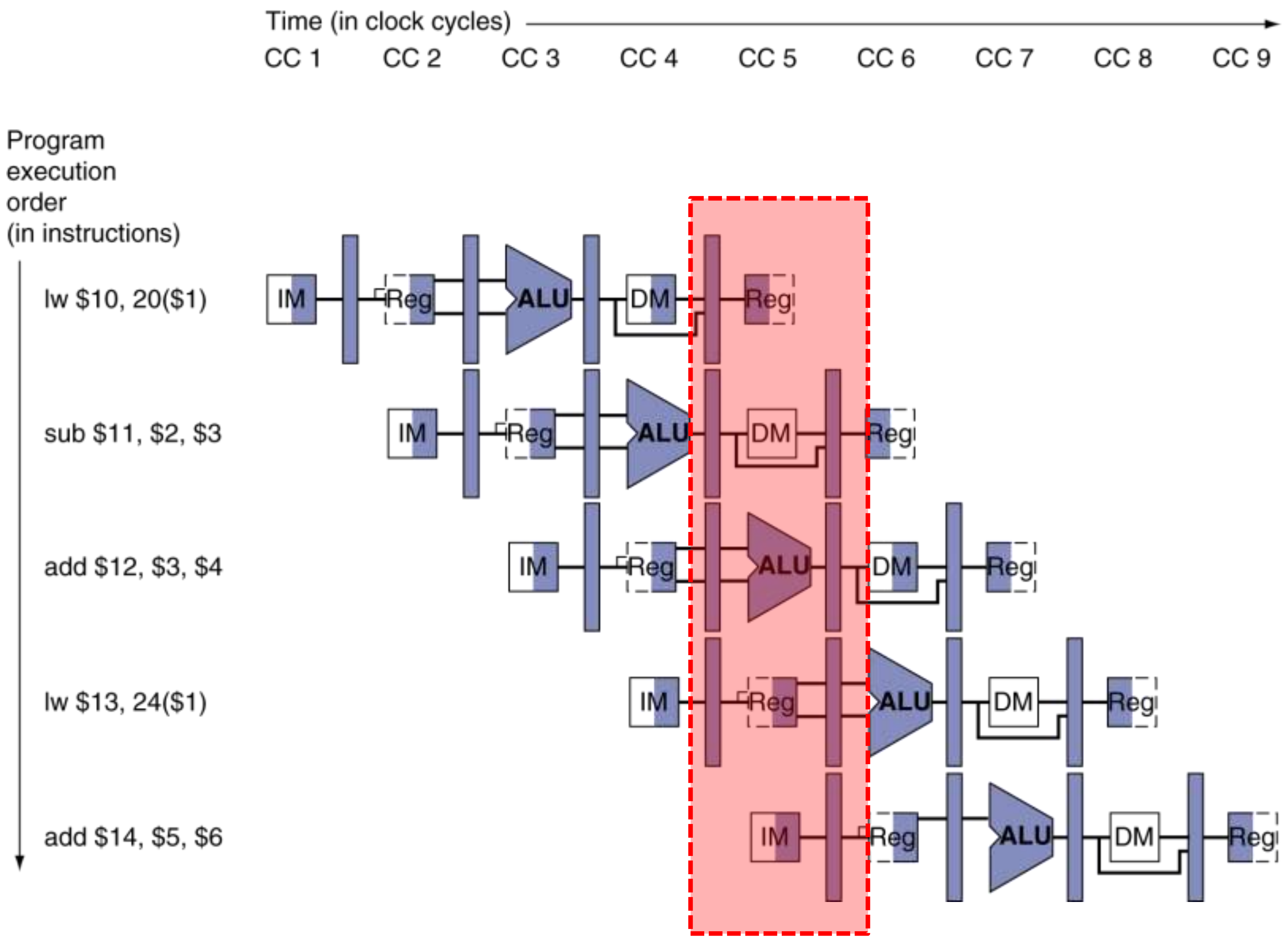


Single-cycle Pipeline Diagram

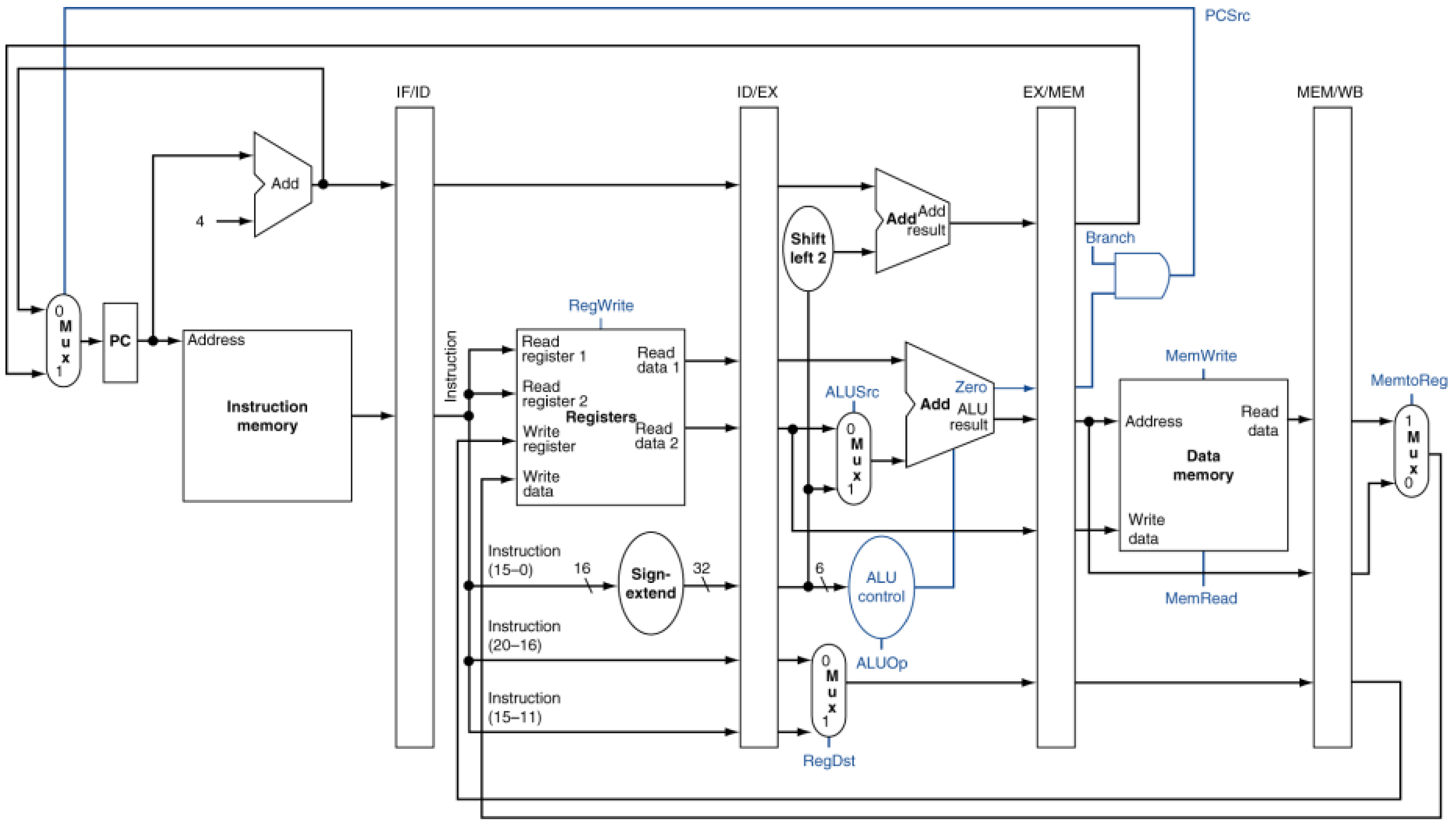
add $14, $5, $6	lw $13, 24 ($1)	add $12, $3, $4	sub $11, $2, $3	lw $10, 20($1)
Instruction fetch	Instruction decode	Execution	Memory	Write-back



Multi-cycle Pipeline Diagram



Pipeline Control Signals

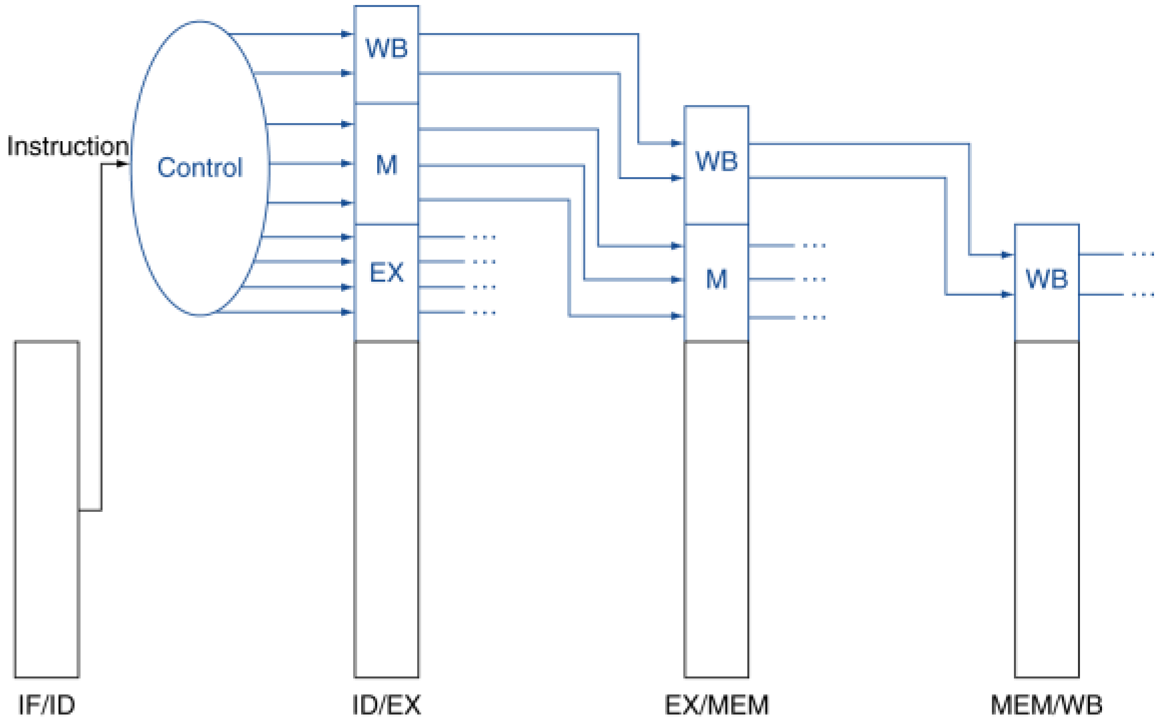


Pipeline Control

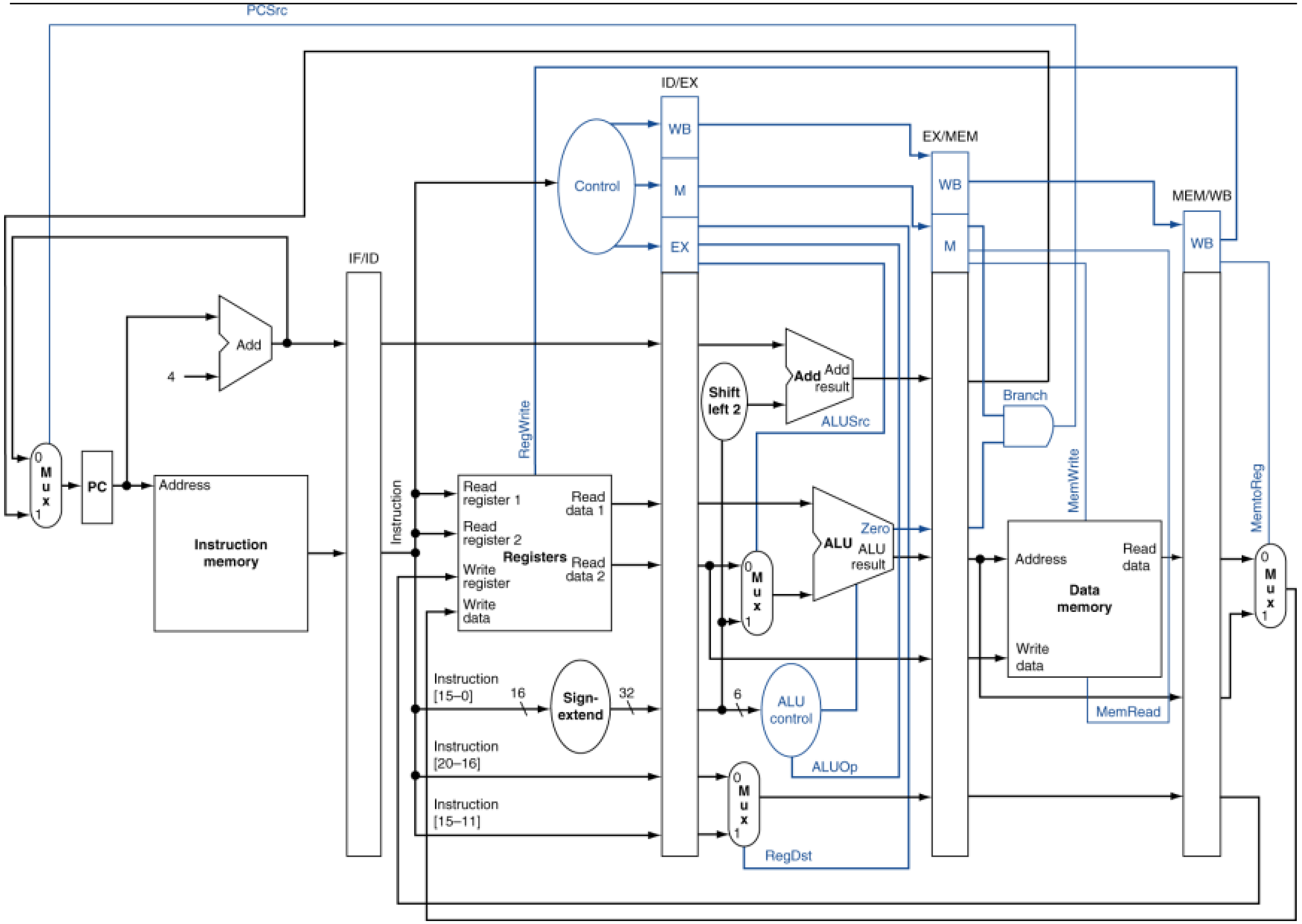
- Control lines groups by stages

Instruction	Execution/address calculation stage control lines				Memory access stage control lines			Write-back stage control lines	
	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

- Pipeline registers are extended to include control signals
  - Control lines start at EX stage



Pipelined Datapath with Control



Data Hazard

- Example instruction sequence

```
sub $2, $1,$3
and $12,$2,$5
or  $13,$6,$2
add $14,$2,$2
sw  $15,100($2)
```

these 4 instructions are dependent on the 1st instruction

time

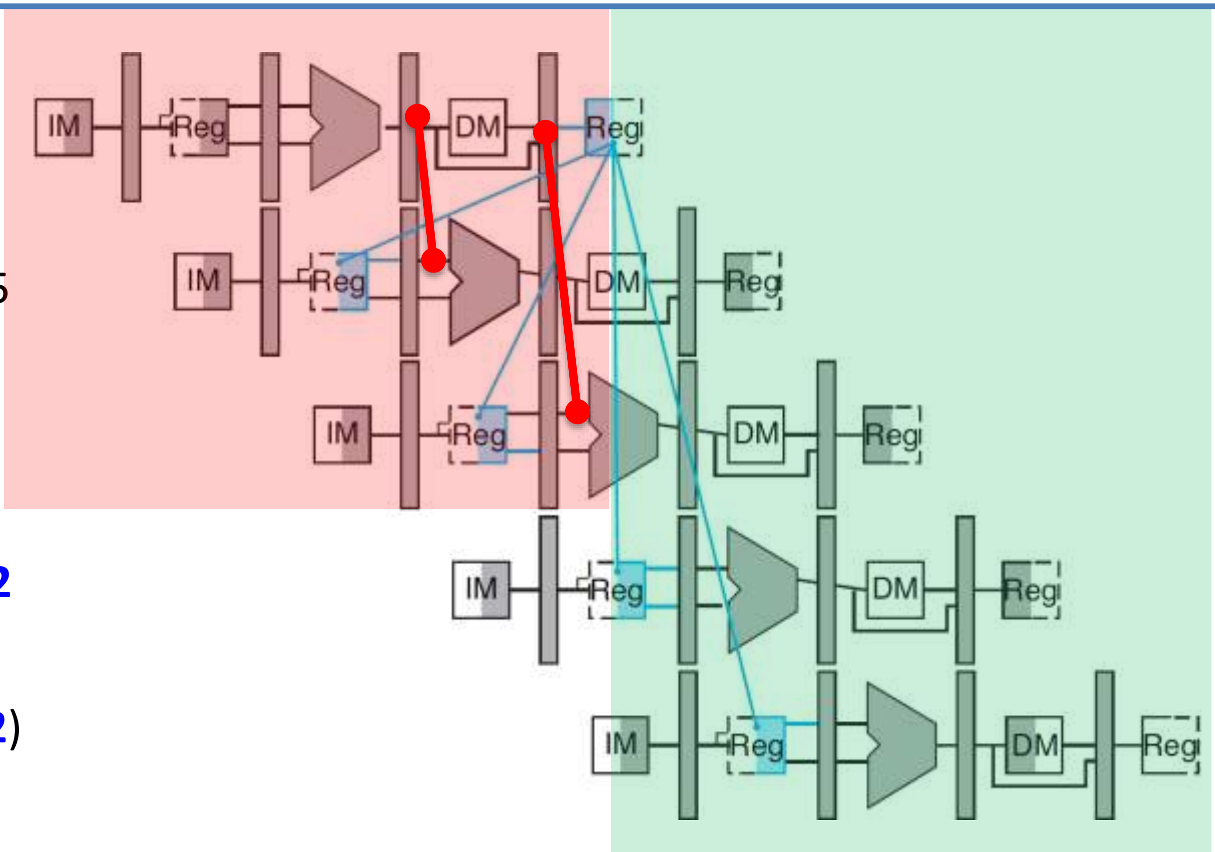
sub **$2**, $1, $3

and $12, **$2**, $5

or $13, $6, **$2**

add $14, $2, **$2**

sw $15, 100(**$2**)



Dependence Detection

- Notations
  - Pipeline Registers
    - IF/ID, ID/EX, EX/MEM, MEM/WB
  - Fields in the pipeline registers
    - ID/EX.RegisterRs
- Detection
  - Pass register numbers across the stages
- Data hazard is detected when:
  - EX/MEM.RegisterRd = ID/EX.RegisterRs
  - EX/MEM.RegisterRd = ID/EX.RegisterRt
  - MEM/WB.RegisterRd = ID/EX.RegisterRs
  - MEM/WB.RegisterRd = ID/EX.RegisterRt

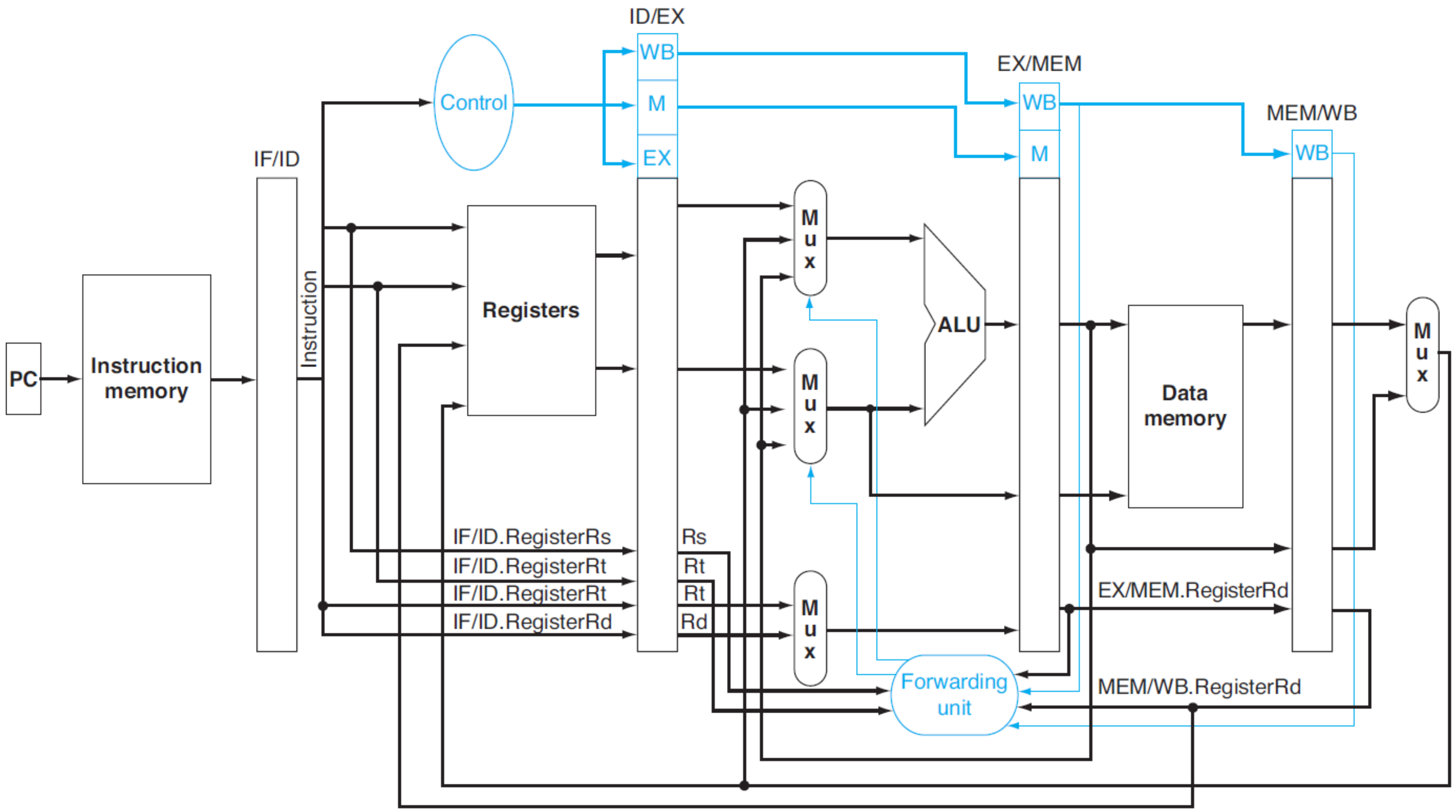
Dependence Detection

- Refinements
  - Checking register numbers is not accurate
    - Not all instructions write data to rd register
    - Solution
      - Check EX/MEM.RegWrite, MEM/WB.RegWrite
  - Safeguard $0 register

```
sll $0, $t3, 5
add $s2, $0, $s1
```

    - EX/MEM.RegisterRd ≠ 0
    - MEM/WB.RegisterRd ≠ 0
- **Forwarding unit** implements this logic

Forwarding Architecture

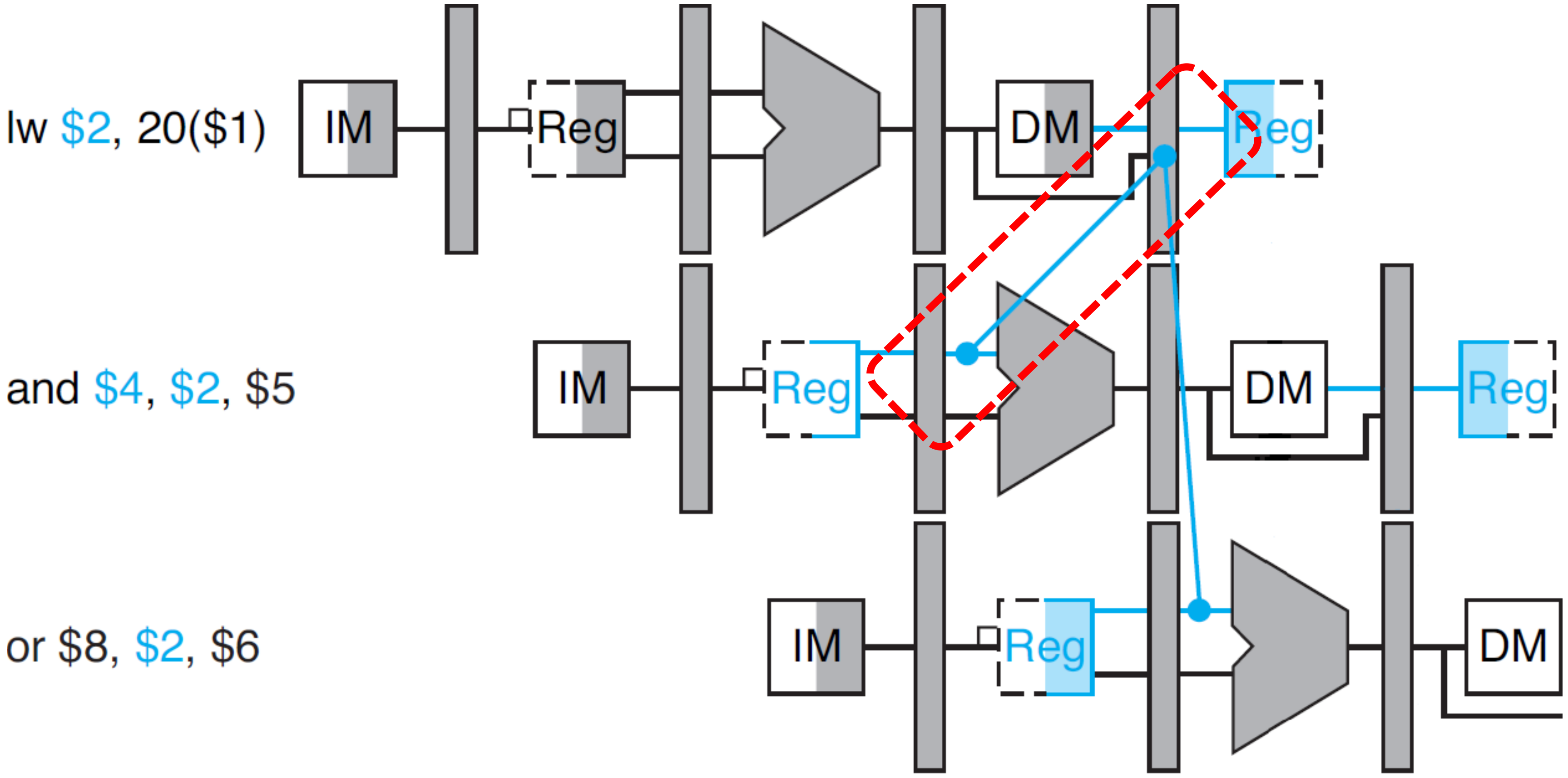


Forwarding Conditions

- EX hazard
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
        and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 10
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
        and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 10
- MEM hazard
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
        and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 01
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
        and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 01

Load-use Data Hazard

- Combination of load with the instruction that reads the output of the load
  - Forwarding cannot be applied
  - Need to insert bubble → **hazard detection unit** needed



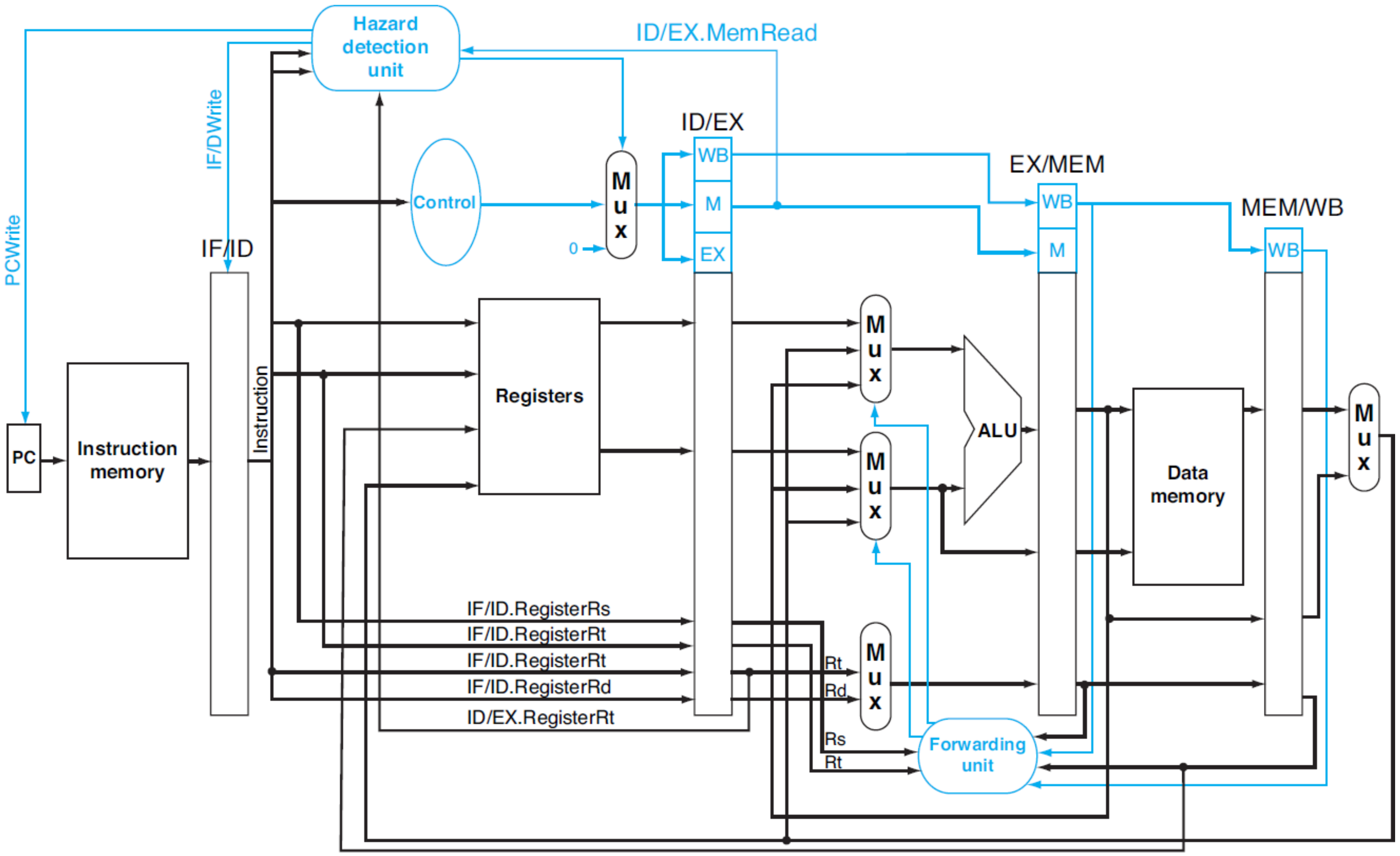
Hazard Detection Unit

- Detect load-use hazard at ID stage
- Detection condition
  - ID/EX.MemRead and $\rightarrow$ **test if it is a load instruction**
    ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
    (ID/EX.RegisterRt = IF/ID.RegisterRt))
  - Insert bubble if condition is true
- Forwarding unit handles data forwarding later

Stalling a Pipeline (Inserting a bubble)

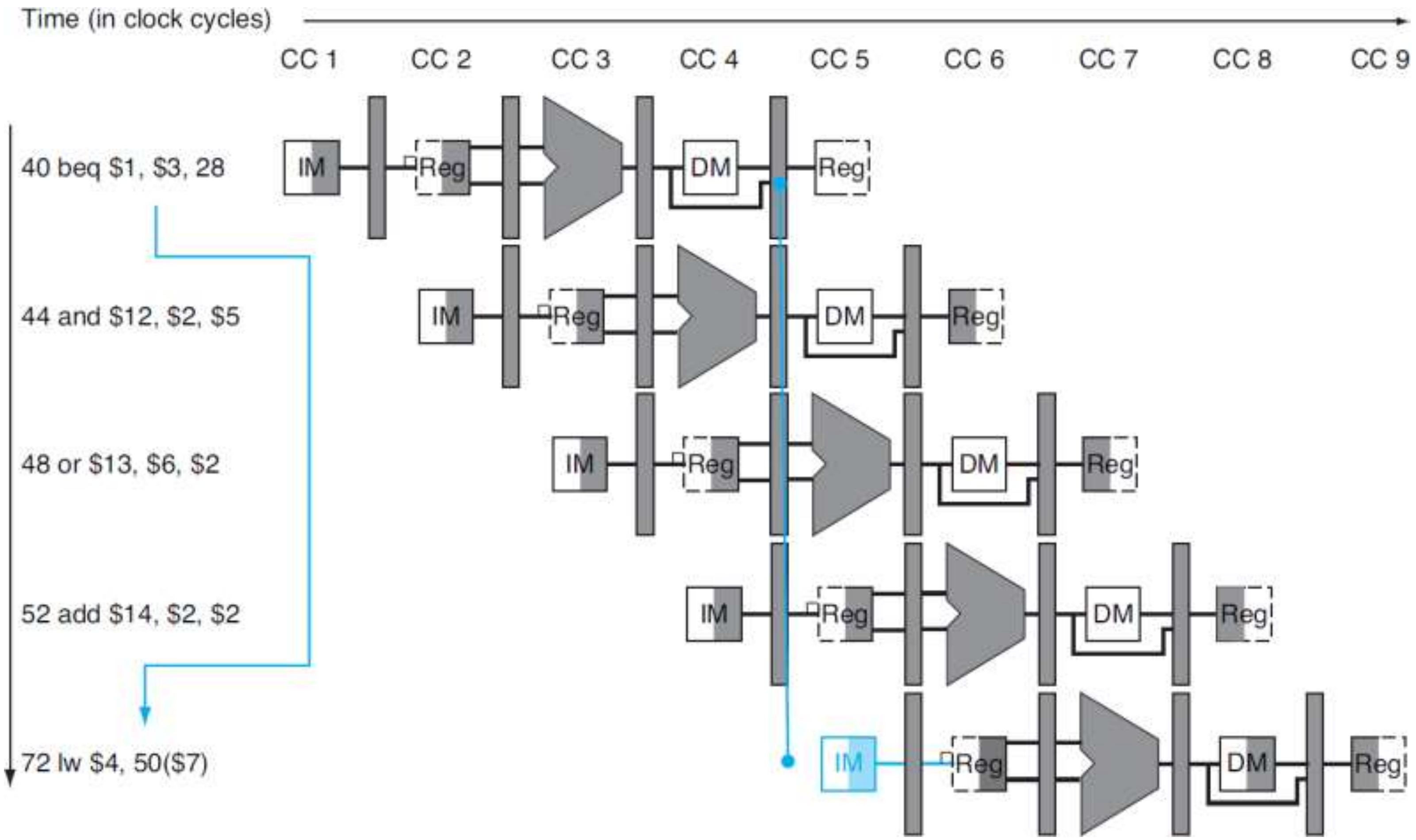
- Stalling requires two actions
  - Prevent PC register and IF/ID from being updated (+4)
    - Next cycle will fetch the same instruction
  - Insert nops (no-operation)
    - Insert all-zero control signals to ID/EX
      - No values are written to any state elements

Hazard Detection Unit with Controls

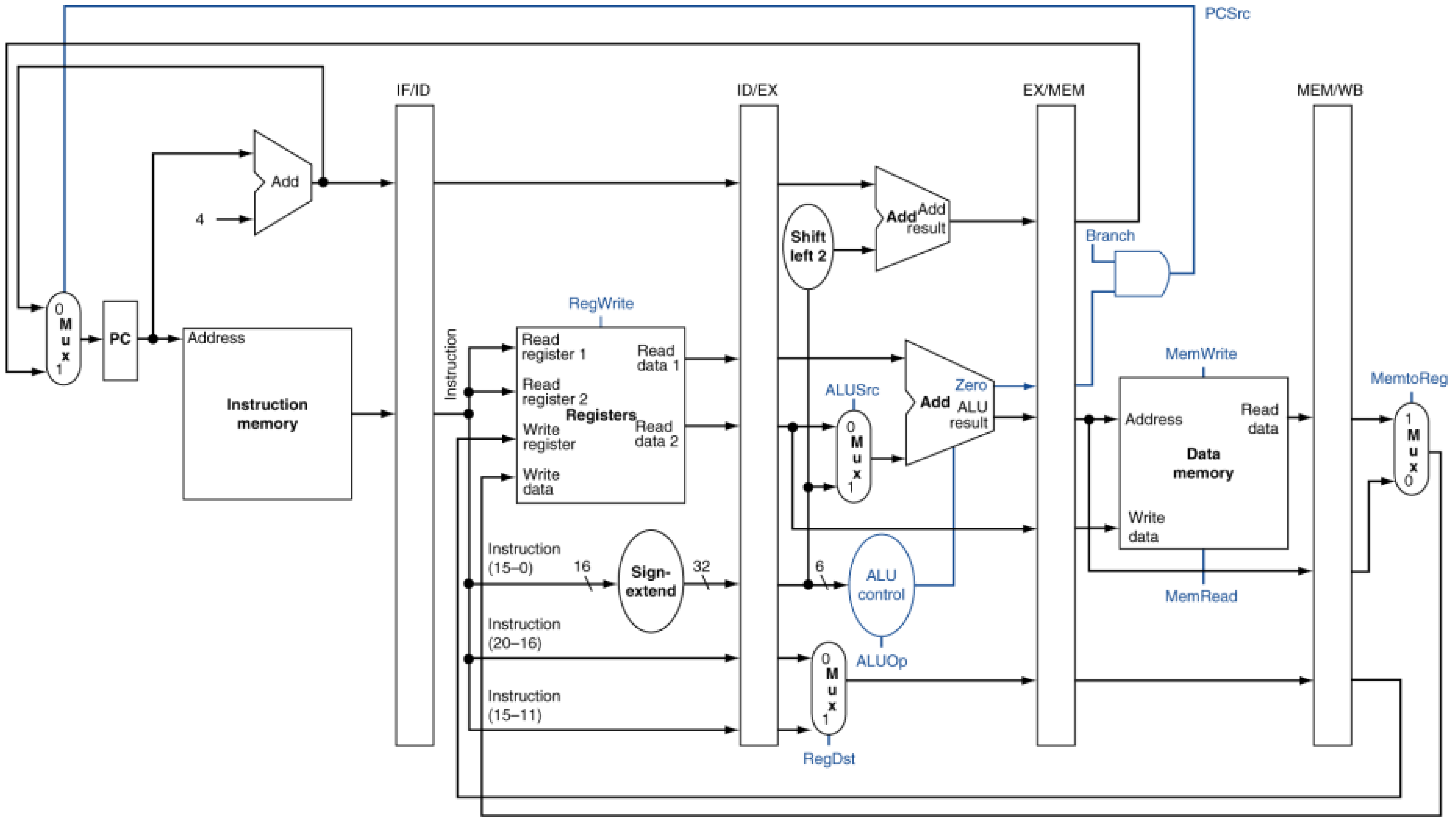


Control Hazards

- Branch-not-taken method
  - Need to discard instructions if prediction is wrong
    - discard 3 instructions in ID and IF stage
  - Flush: Change the control of ID, IF stages



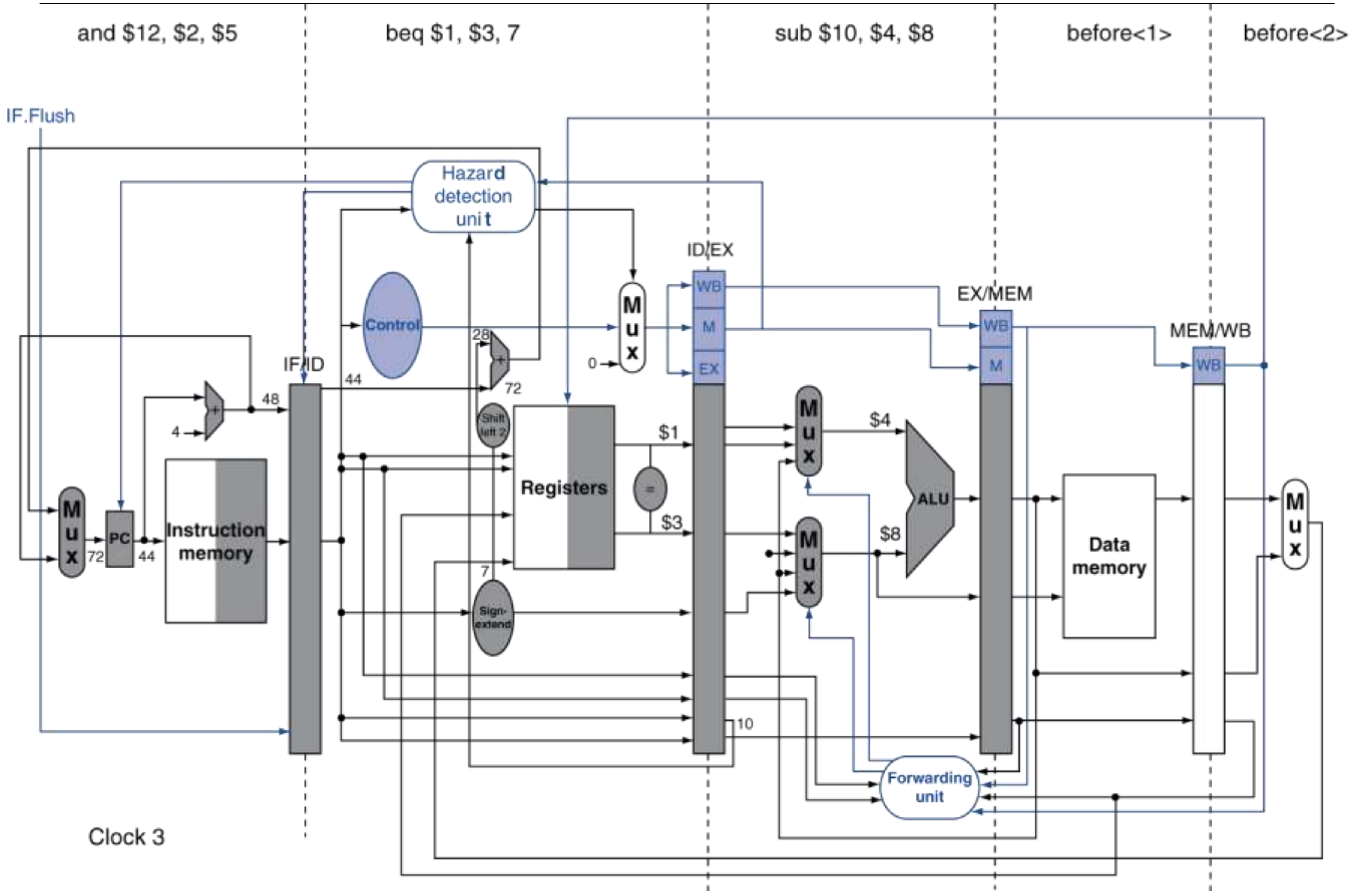
How Many Stalls Are Needed for Control Hazard?



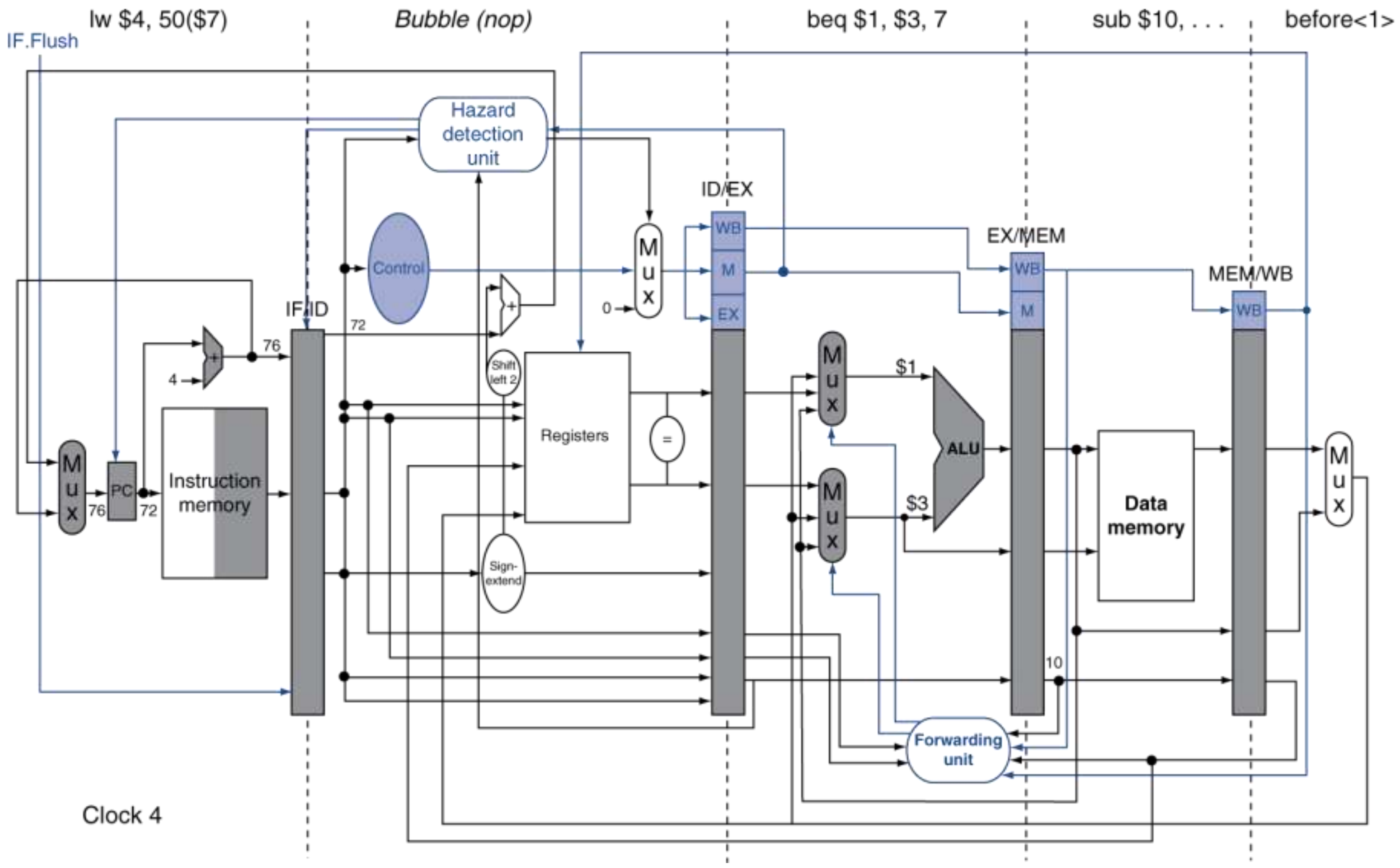
Reducing the Penalty of Branch

- Move the branch decision to ID stage
  - Most instructions use simple test. Let’s do simple equality test.
    - For more complex branch decision, separate branch instruction is created
- Moving up the branch decision to ID stage
  - Branch target: Move adder from EX to ID
  - Branch decision
    - Need additional forwarding and hazard detection logic
    - Need stalls
      - beq, add combination: 1 stall
      - beq, lw combination: 2 stall

Branch Taken – Before



Branch Taken - After



IF.Flush
lw $4, 50($7)
Bubble (nop)
beq $1, $3, 7
sub $10, . . .
before<1>
Hazard detection unit
ID/EX
EX/MEM
MEM/WB
IF/ID
Control
WB
M
EX
Mux
0
72
76
4
PC
Instruction memory
Shift left 2
Registers
=
Sign-extend
$1
$3
ALU
Data memory
10
Forwarding unit
Clock 4