Computer Organization and Design
The Hardware/Software Interface

Chapter 4 - Processor

Instructor: Dr. Feng Li

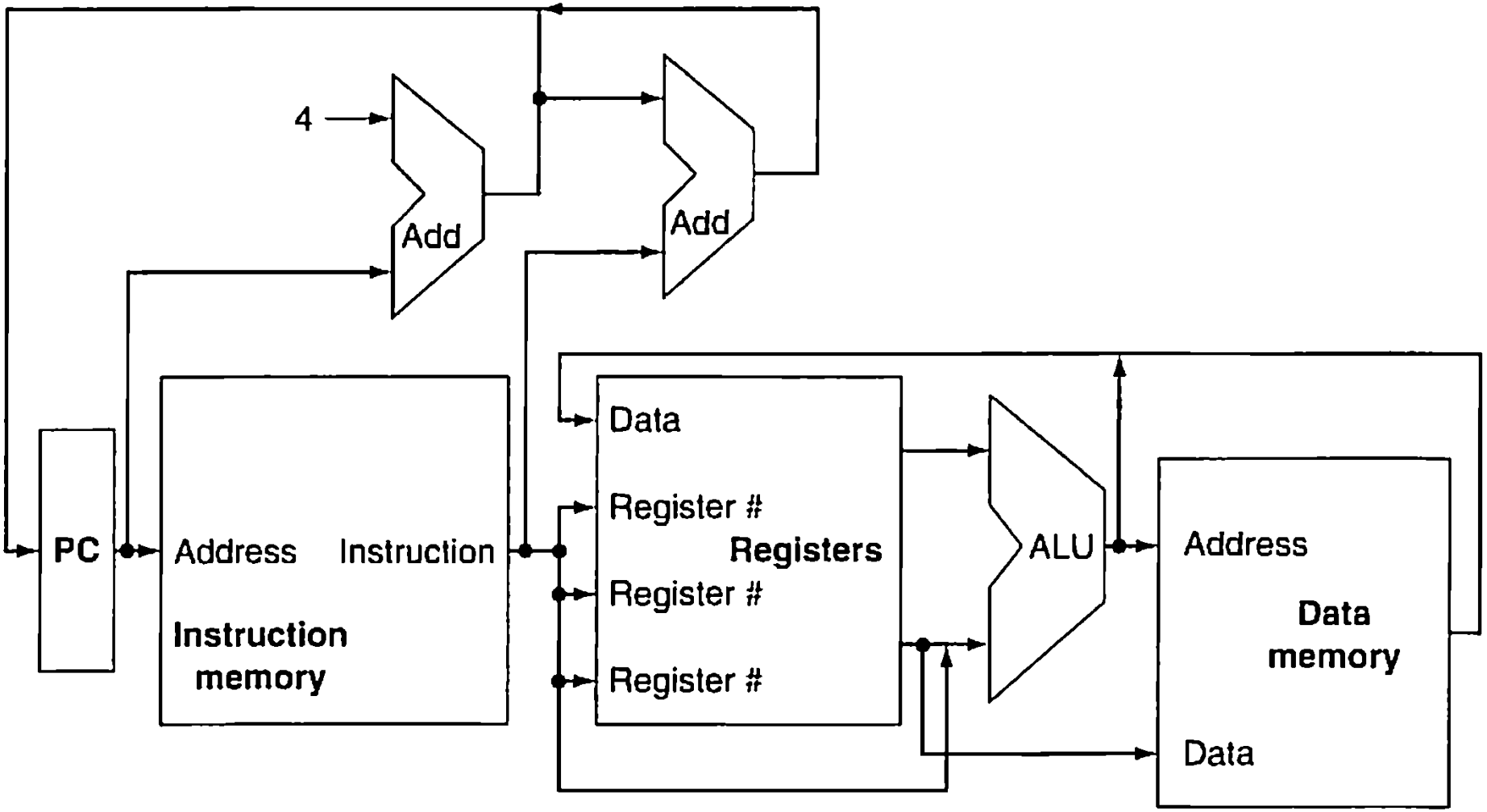
Chapter Four: The processor

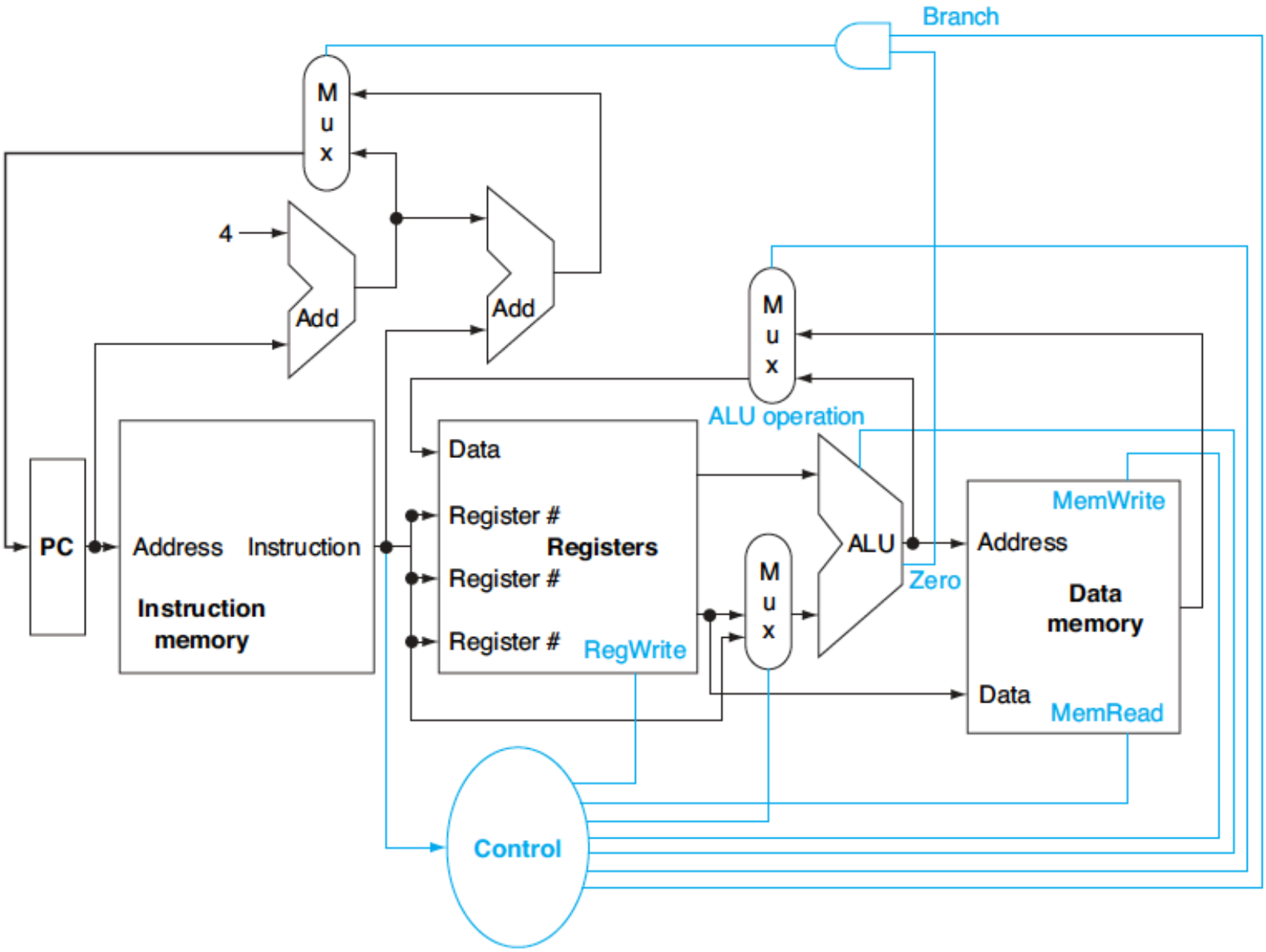
- 4.1 Introduction
- 4.2 Logic Design Conventions
- 4.3 Building a datapath
- 4.4 A Simple Implementation Scheme
- 4.5 An Overview of Pipelining
- 4.6 Pipelined Datapath and Control
- 4.7 Data Hazards: Forwarding versus Stalling
- 4.8 Control Hazards
- 4.9 Exceptions
- 4.10 Parallelism and Advanced Instruction-Level Parallelism
- … …

Introduction

- We'll look at an implementation of the MIPS
- Simplified to contain only:
  - memory-reference instructions: lw, sw
  - arithmetic-logical instructions: add, sub, and, or, slt
  - control flow instructions: beq, j
- An Overview of the implementation
  - For every instruction, the first two step are identical
    - Fetch the instruction from the memory
    - Decode and read the registers
  - Next steps depend on the instruction class
    - Memory-reference instructions, Arithmetic-logical instructions, branch instructions

An abstract view of the implementation of MIPS



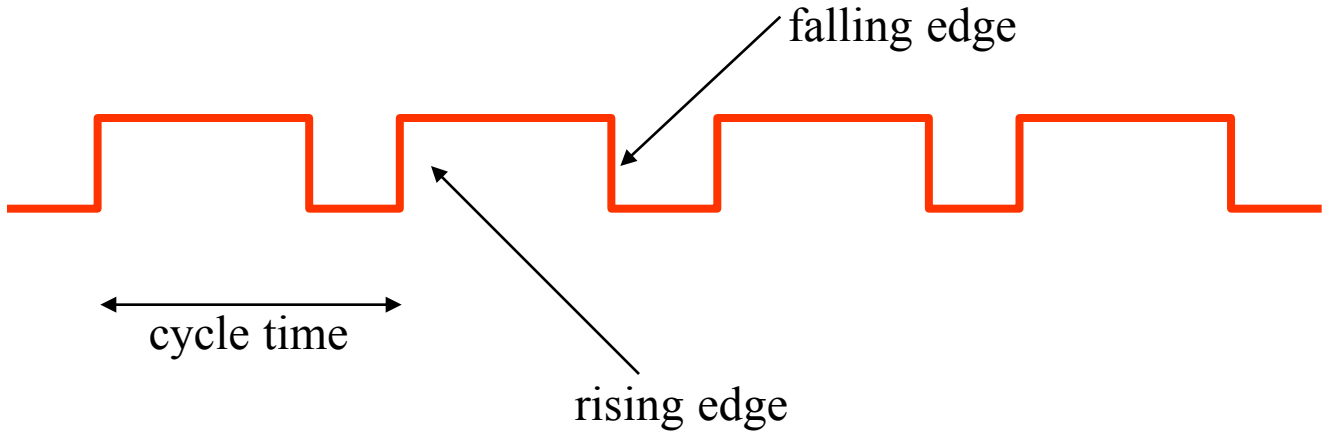

Branch
M
u
x
4
Add
Add
M
u
x
ALU operation
PC
Address
Instruction
Instruction
memory
Data
Register #
Registers
Register #
Register #
RegWrite
M
u
x
ALU
Zero
MemWrite
Address
Data
memory
Data
MemRead
Control

Logic Design Conventions

- Datapath elements consist of two types of logic elements
  - Combinational elements that operates on data values
  - State elements: elements that contain state
- The outputs of combinational elements depend only on the current input
- State elements have some internal storage, and the state can be maintained even when computers have no power
  - Two inputs: data value and clock
  - One output: the value that was written in an earlier clock cycle
  - E.g., D-type flip-flop, memories, registers

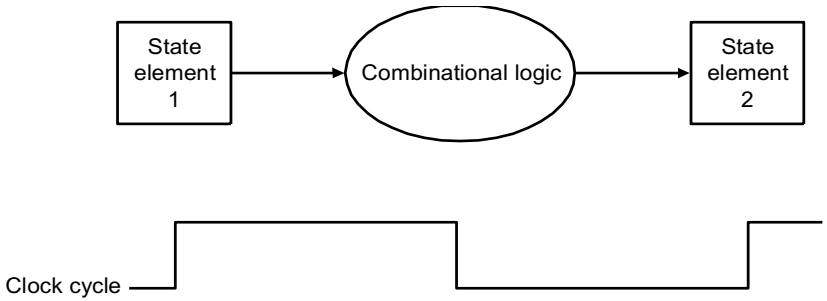
State Elements

- Clocks used in synchronous logic
  - ➢when should an element that contains state be updated?



Clocking methodology

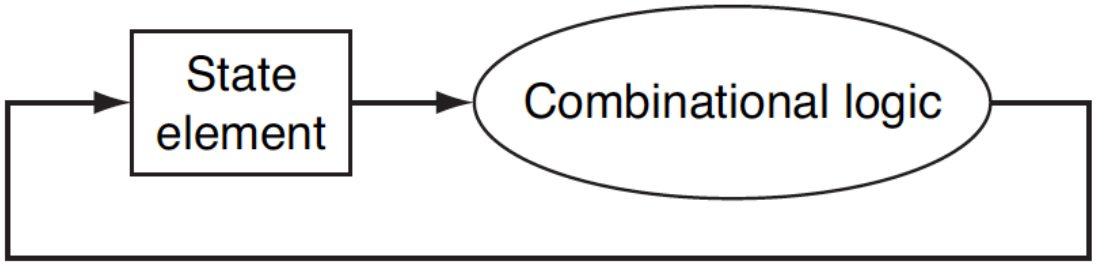
- Clocking methodology defines when signals can be read and when they can be written
- An edge-triggered clocking methodology
  - Any values stored in a sequential logic element are updated only on a clock edge, which is a quick transition from low to high or vice versa
- Typical execution:
  - read contents of some state elements,
  - send values through some combinational logic
  - write results to one or more state elements



All signals must propagate from state element 1, through the combinational logic, and to state element 2 in the time of one clock cycle.

- If a state element is not updated on every clock, then an explicit write control signal is required.
- Asserted: a signal is logically high
- Deasserted: a signal is logically low

- An edge-triggered methodology allows a state element to be read and written in the same clock cycle without creating a race that could lead to indeterminate data values



Building a datapath

- Datapath element
  - A unit used to operate on or hold data within a processor. In the MIPS implementation, the datapath elements include the instruction and data memories, the register file, the ALU and adders

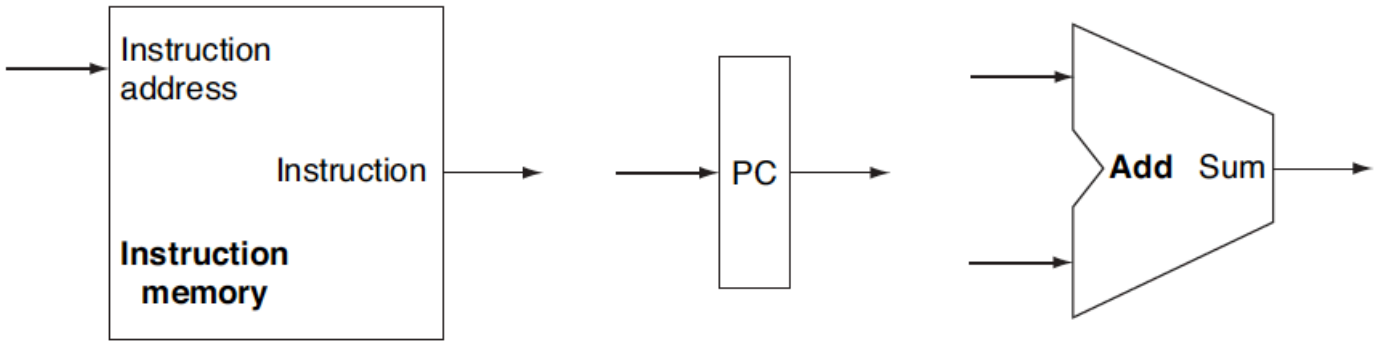
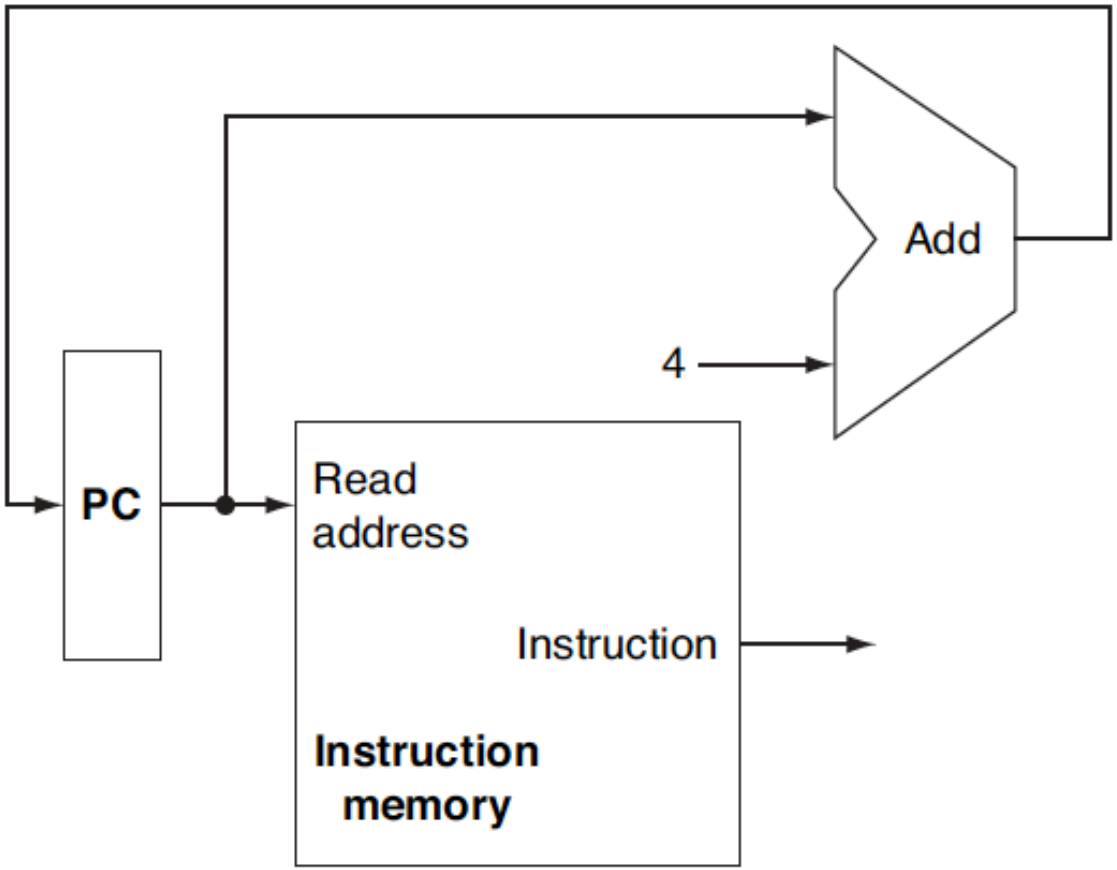
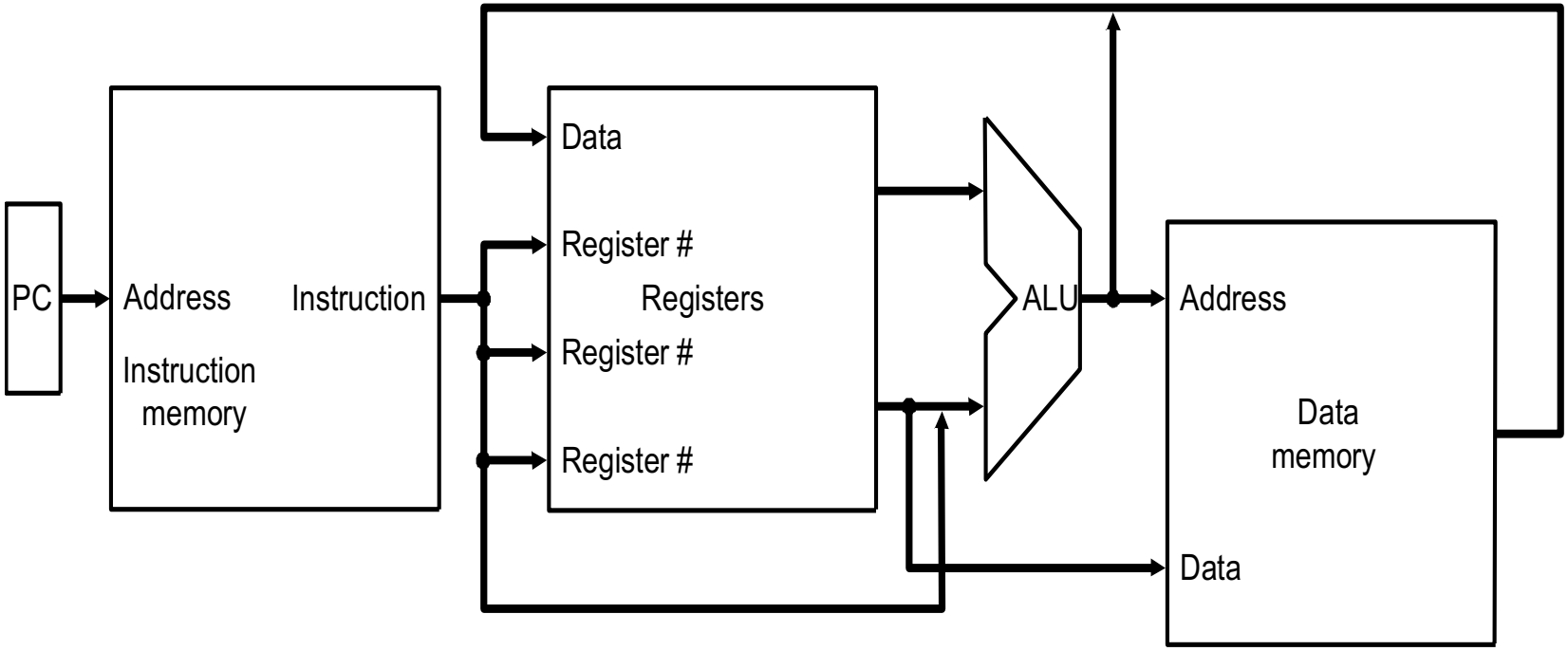


FIGURE 4.5 Two state elements are needed to store and access instructions, and an adder is needed to compute the next instruction address. The state elements are the instruction memory and the program counter. The instruction memory need only provide read access because the datapath does not write instructions. Since the instruction memory only reads, we treat it as combinational logic: the output at any time reflects the contents of the location specified by the address input, and no read control signal is needed. (We will need to write the instruction memory when we load the program; this is not hard to add, and we ignore it for simplicity.) The program counter is a 32-bit register that is written at the end of every clock cycle and thus does not need a write control signal. The adder is an ALU wired to always add its two 32-bit inputs and place the sum on its output.

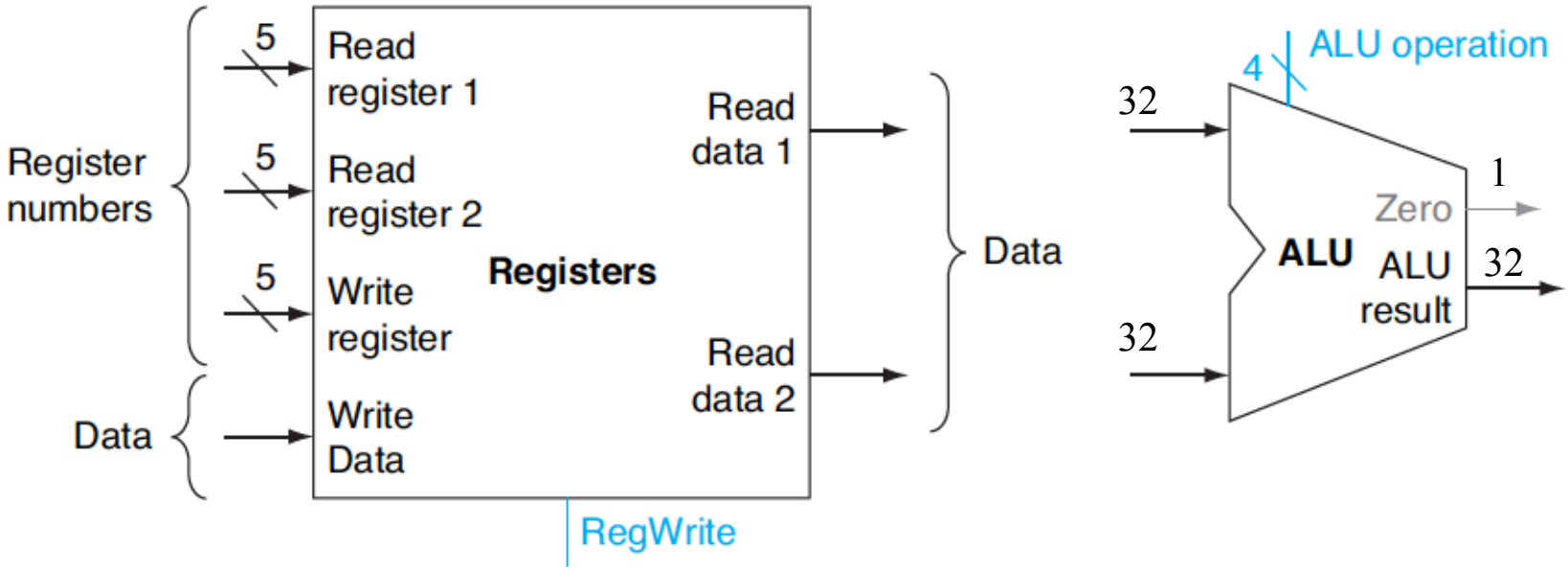
Instruction fetching unit




PC
Address
Instruction
Instruction memory
Data
Register #
Registers
Register #
Register #
ALU
Address
Data memory
Data

R-type (or arithmetic-logical) instructions

- Read two registers, perform an ALU operation on the contents of the registers, and write the result to a register

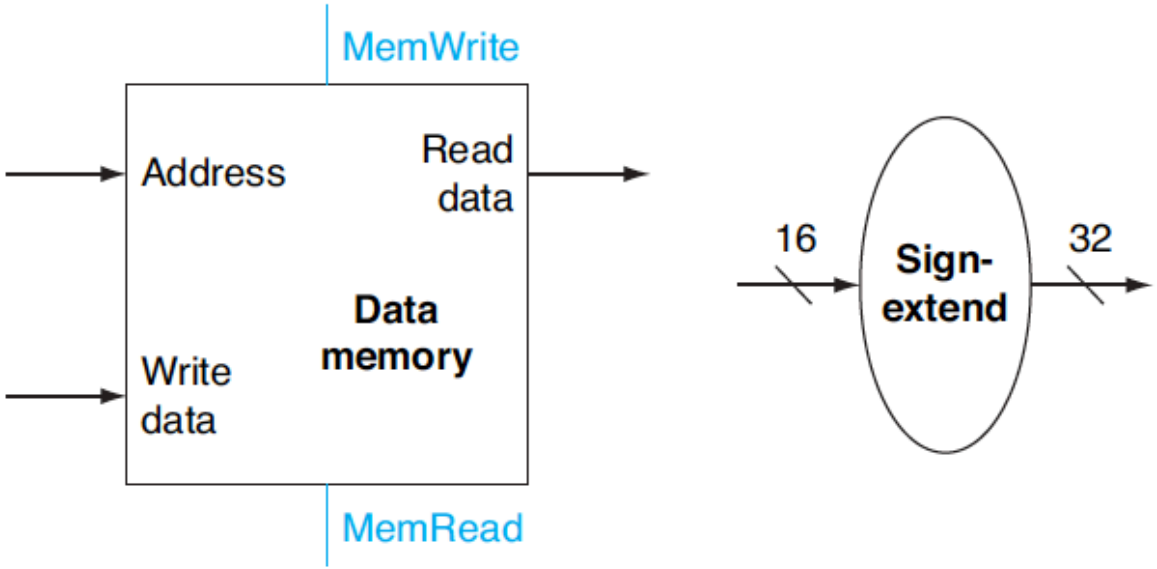


a. Registers

b. ALU

Memory-reference instructions

- lw $t1, offset_value ($t2)
- sw $t1, offset_value ($t2)



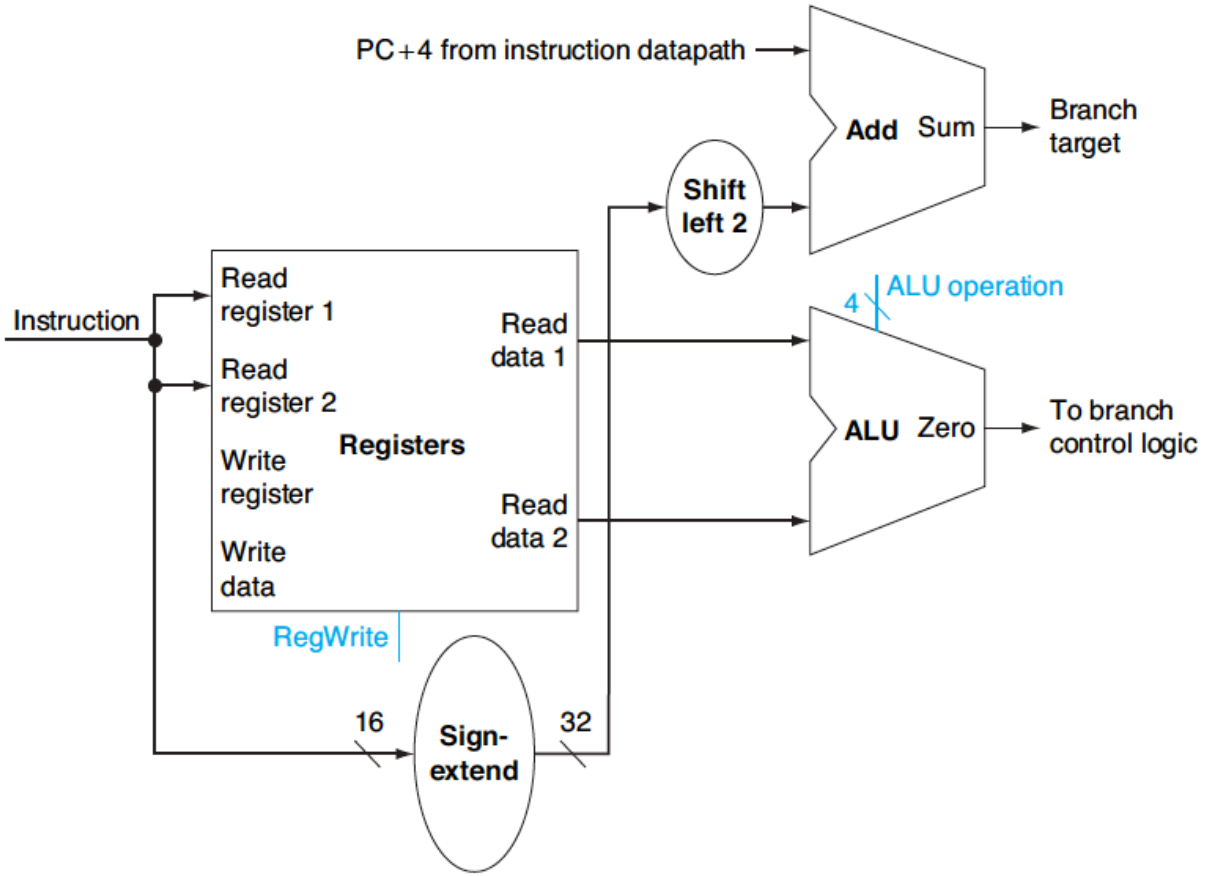
a. Data memory unit

b. Sign extension unit

Branch instruction

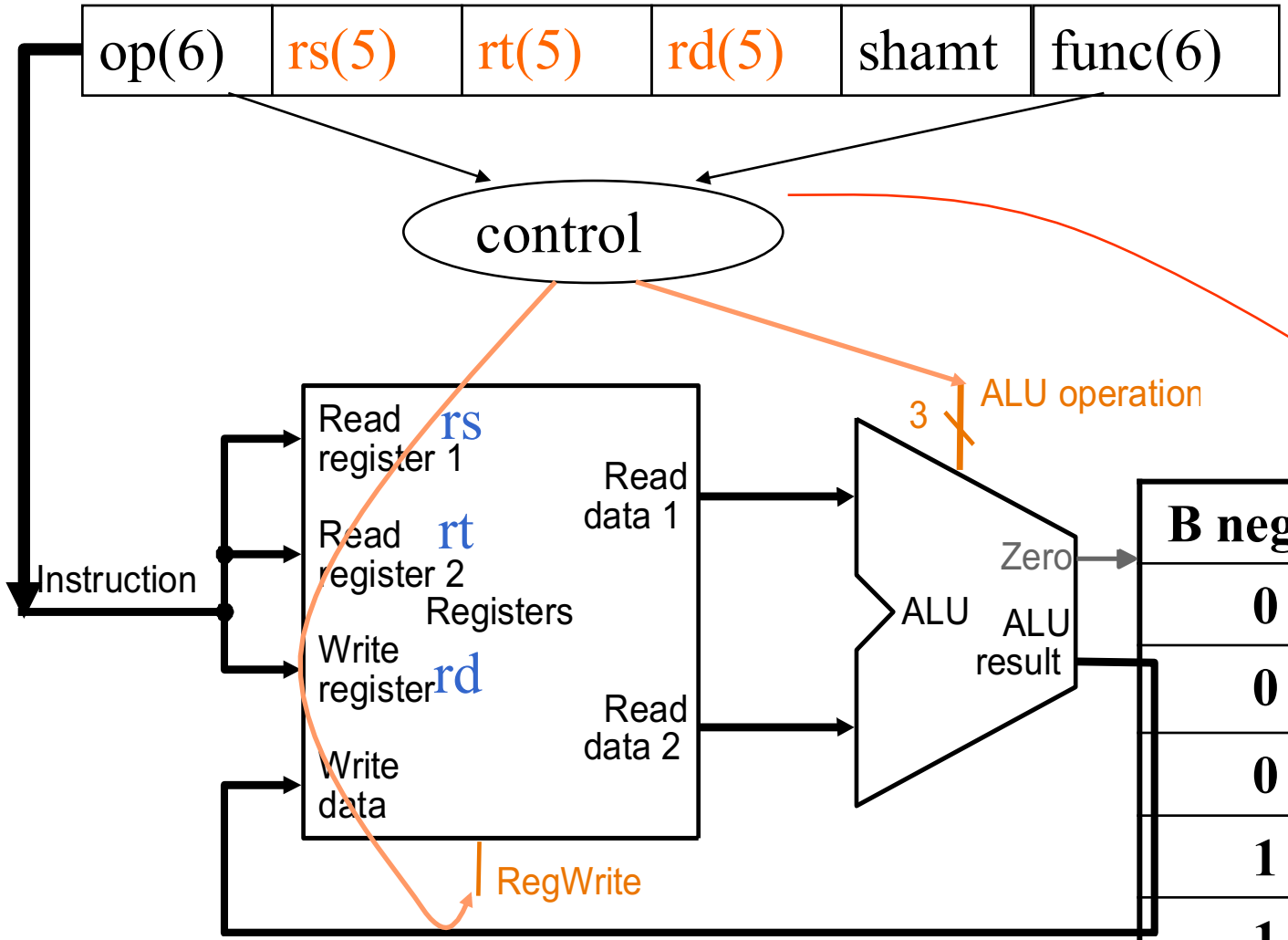
- beq instruction
  - Three operands: two registers that are compared for equality, and a 16-bit offset used to compute the branch target address relative to the branch instruction address
  - The base for branch address calculation is PC+4
  - The offset is based on word rather than byte, so the offset field should be shifted left 2 bits

- Compare the register contents to determine if the branch is taken or not
- Compute the branch target address



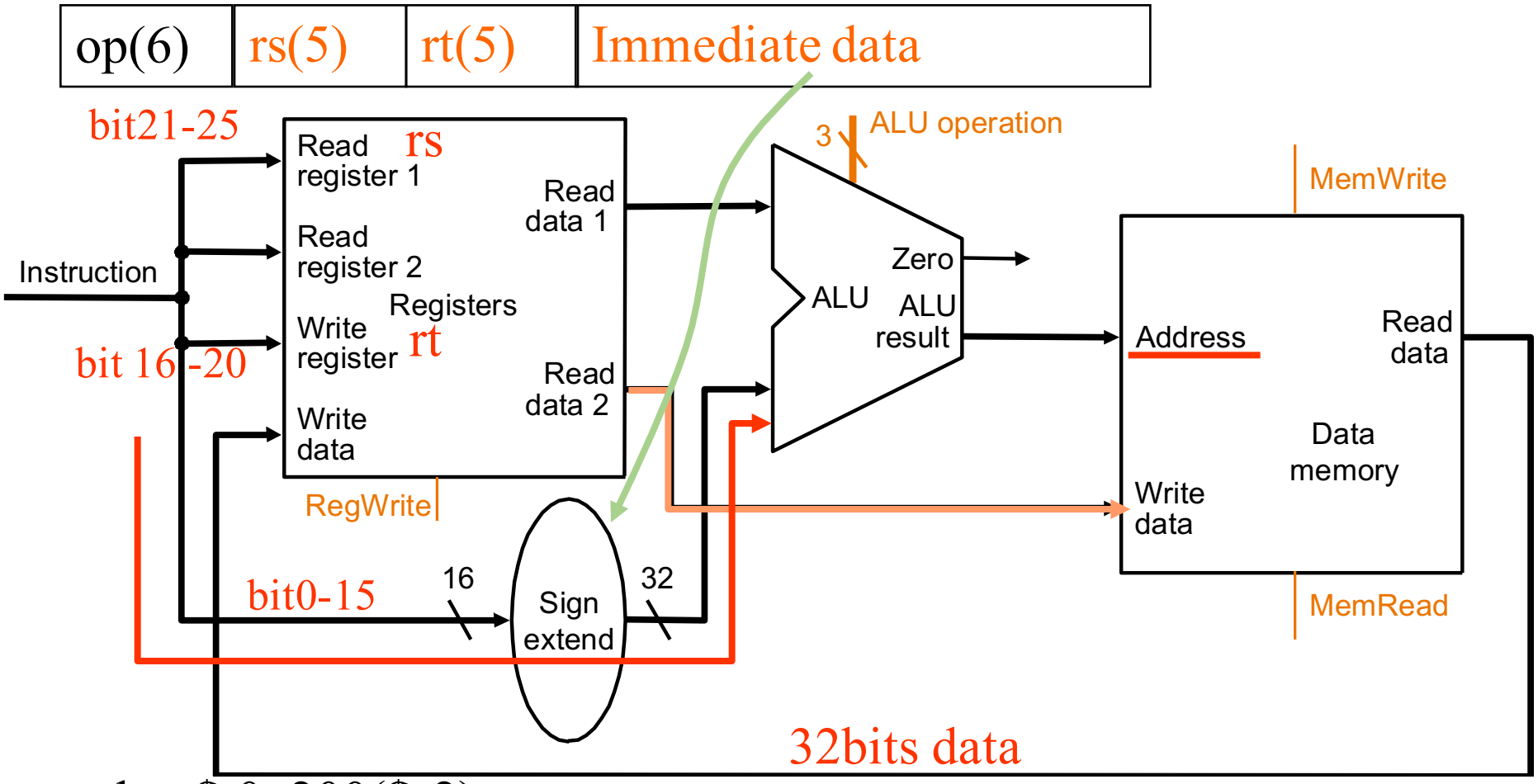
Implement the R-type instruction

R-instruction format：



B negate	op	function
0	00	and
0	01	Or
0	10	Add
1	10	Sub
1	11	Slt

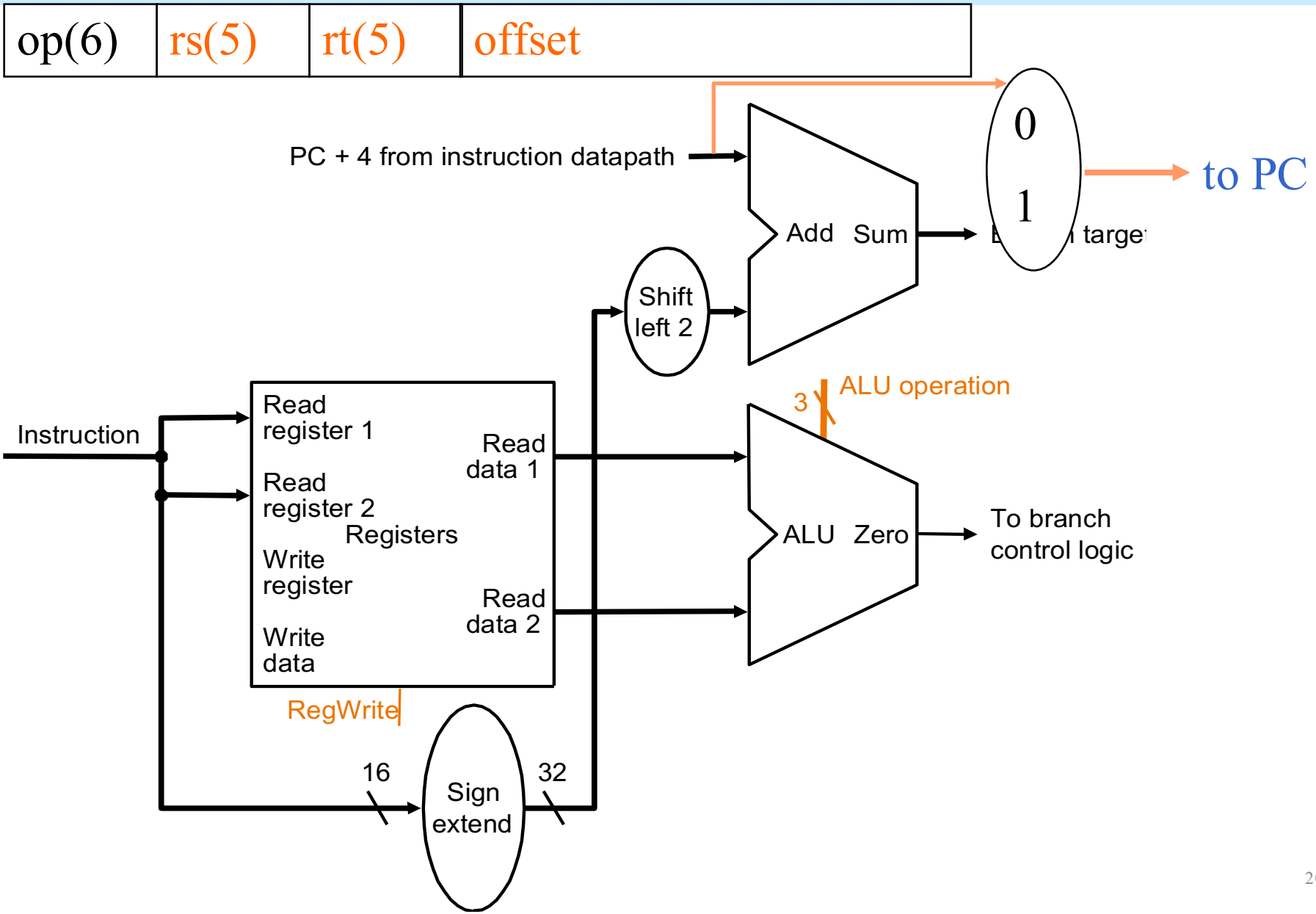
Implement the I type instruction



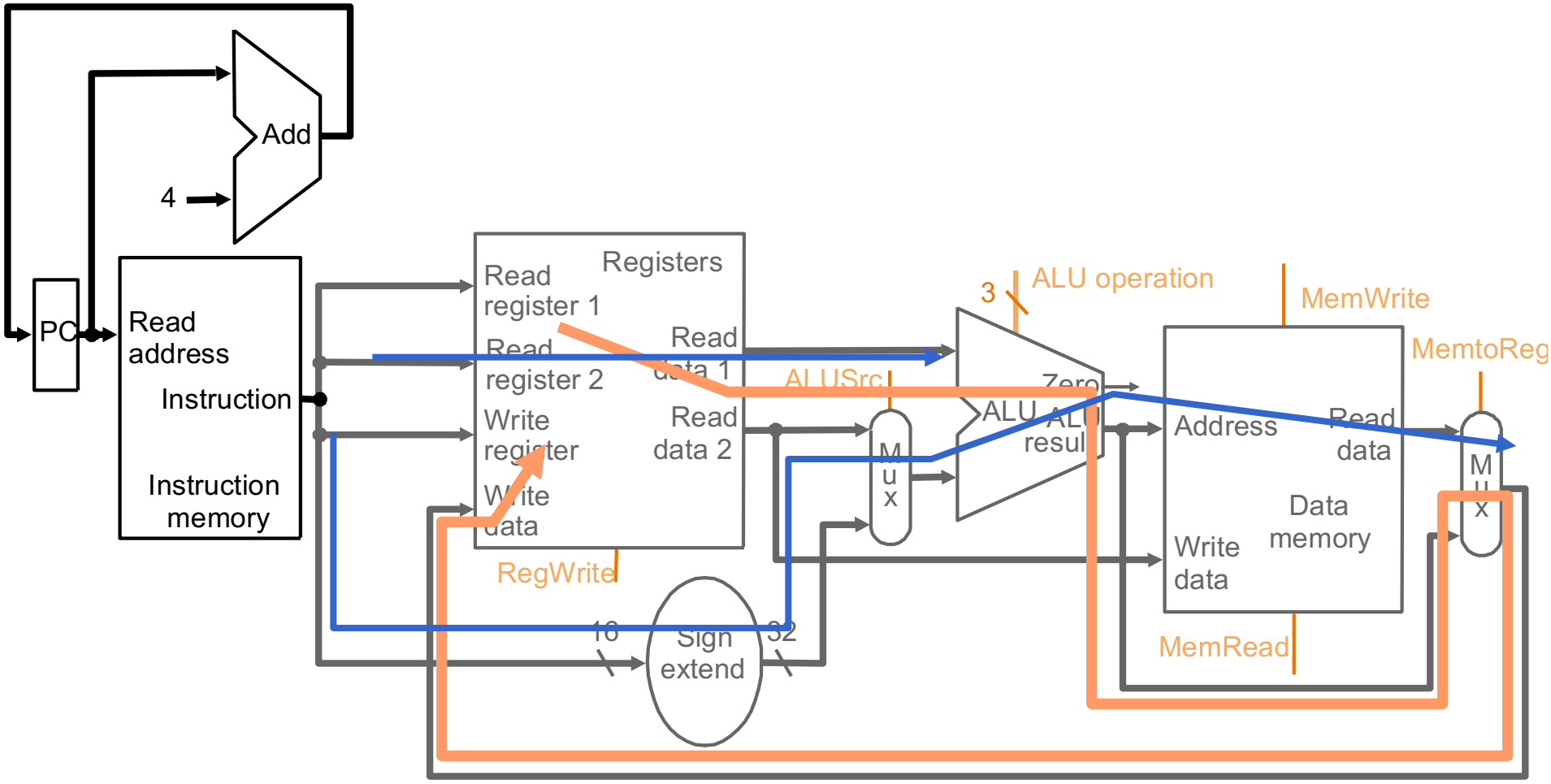
lw $t0, 200($s2)

if $s2=1000, it will load word in element number 1200 to $t0

Implementation of beq

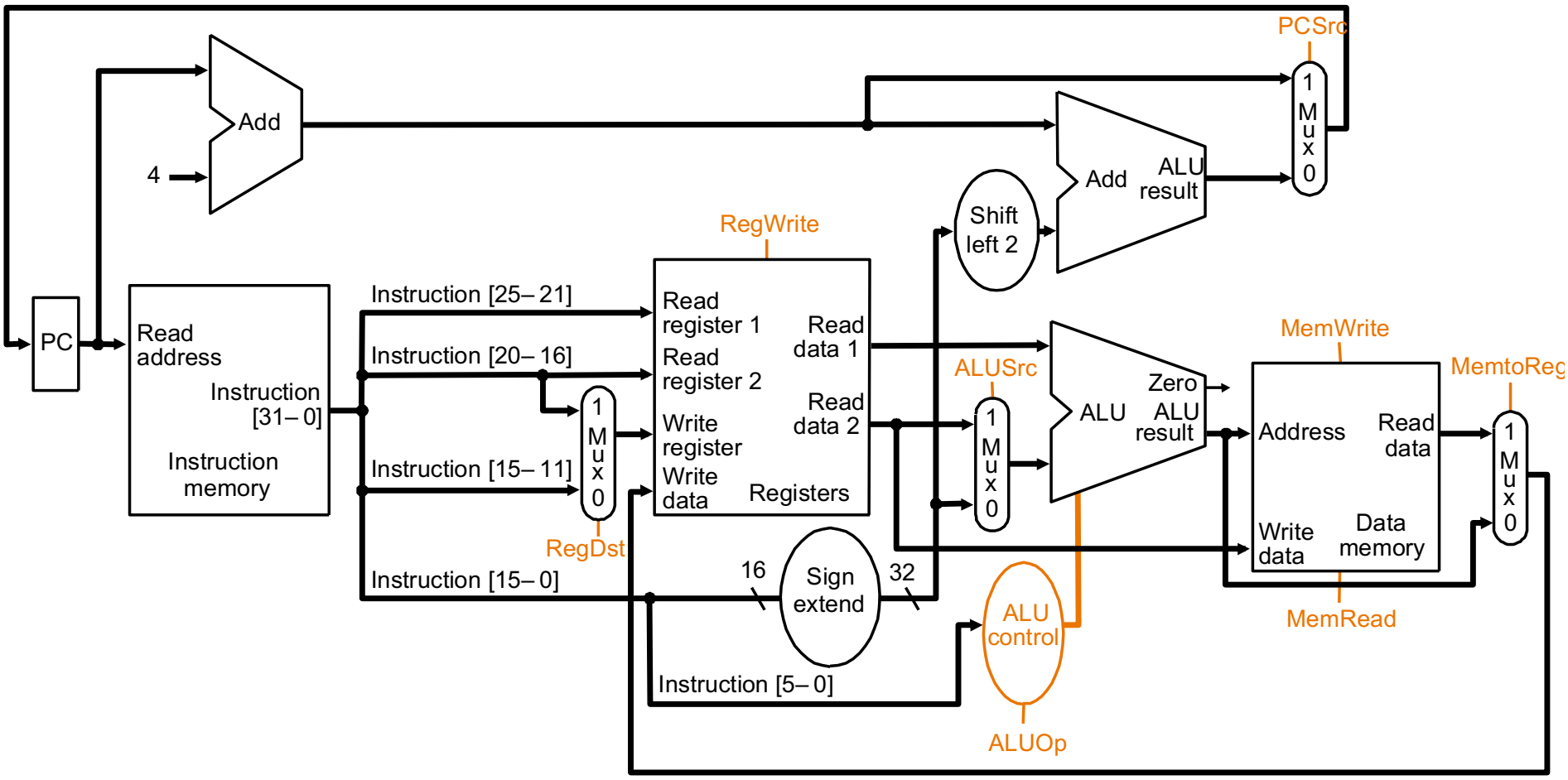


Combine the implementation R-type and I-type



Building the Datapath

- Use multiplexors to stitch them together



Note : control signals e.g. add $s0, $s1,$s2/ addi $s0,$s1,100

A simple implementation scheme

- Data path + control function
- Instructions
  - lw, sw
  - beq
  - add, sub, and, or, set on less than
  - j

ALU control

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

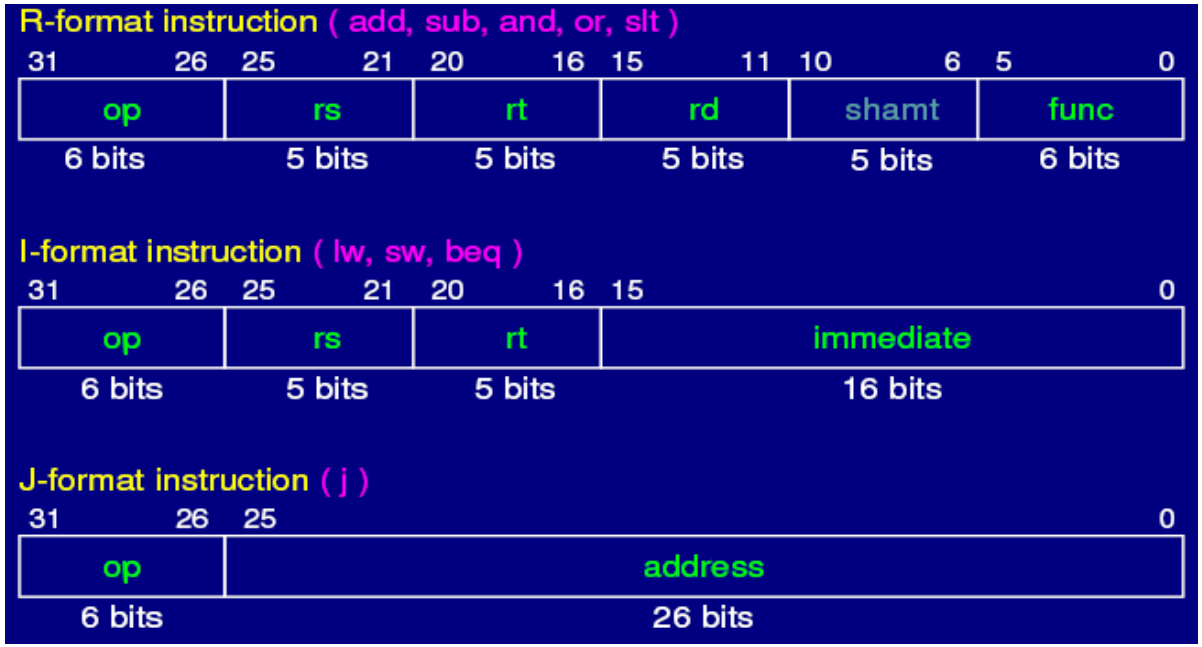
Depending on the instruction class, the ALU needs to perform one of these first five functions

- We need a small control unit
  - Input: the function field of the instruction and a 2-bit control field (i.e., ALUOp)
  - Ouput: 4-bit ALU control signal
- ALUOp indicates the operations that the ALU will perform
  - Addition (00) for load and store instructions
  - Subtraction (01) for beq
  - Operations encoded in the funct field (10)

Control

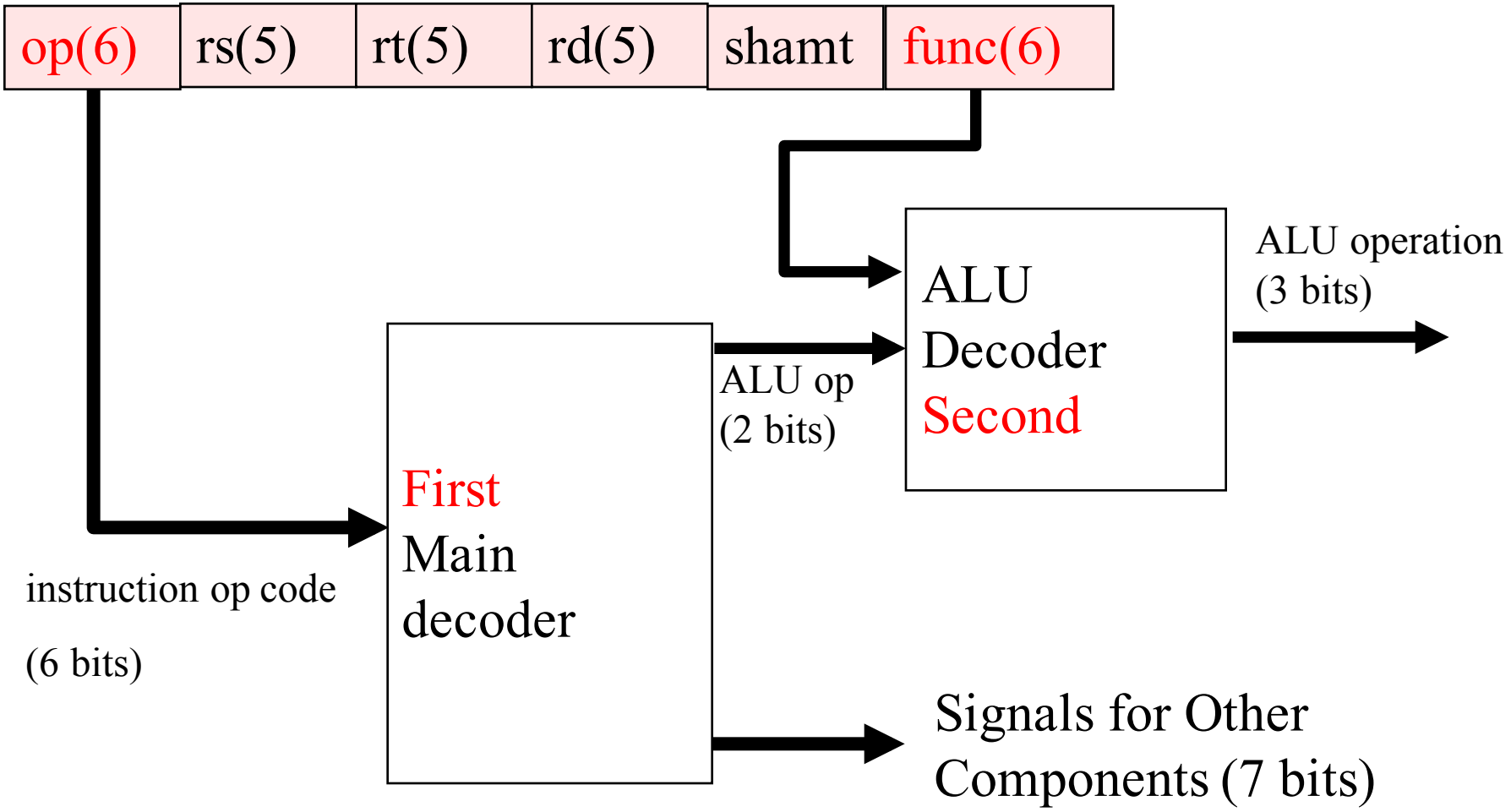
Analyze for cause and effect

- Information comes from the 32 bits of the instruction
- Selecting the operations to perform (ALU, read/write, etc.)
- Controlling the flow of data (multiplexor inputs)
- ALU's operation based on instruction type and function code



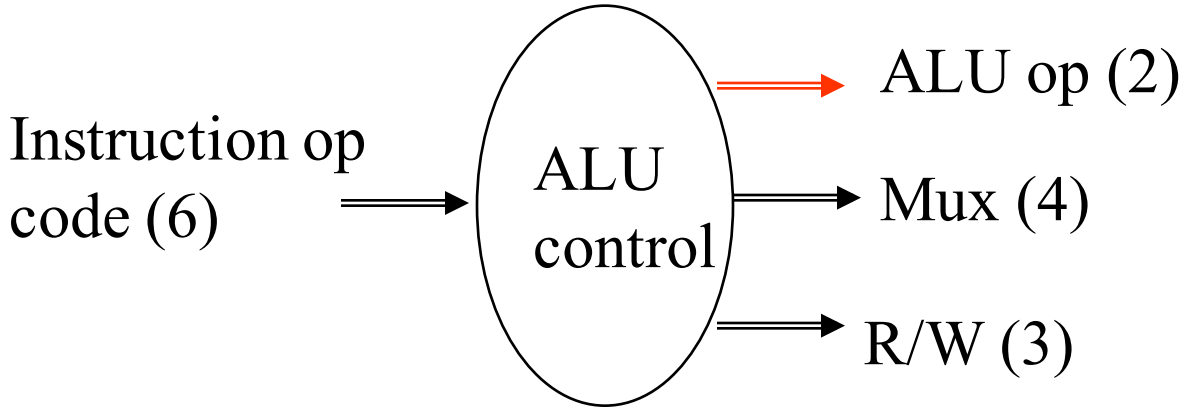
Scheme of Controller

- 2-level decoder



Designing the Main Control Unit (First level)

- Main Control Unit function
  - ALU op (2)
  - Divided 7 control signals into 2 groups
    - 4 Mux
    - 3 R/W



Designing the ALU decoder (Second level)

- How the ALU control bits are set depends on the ALUOp control bits and the different function codes for the R-type instructions

Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	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
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

Truth Table for ALU decoder

- Describe it using a truth table (can turn into gates):

ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	*210*
0	0	X	X	X	X	X	X	**010**
X	1	X	X	X	X	X	X	**110**
1	X	X	X	0	0	0	0	**010**
1	X	X	X	0	0	1	0	**110**
1	X	X	X	0	1	0	0	**000**
1	X	X	X	0	1	0	1	**001**
1	X	X	X	1	0	1	0	**111**

don't care

$$Operation_2 = ALU_{op0} + ALU_{op1}(\overline{F}_3\overline{F}_2F_1\overline{F}_0 + F_3\overline{F}_2F_1\overline{F}_0)$$

$$Operation_1 = \overline{ALU_{op1}\overline{F}_3F_2\overline{F}_1\overline{F}_0 + ALU_{op1}\overline{F}_3F_2\overline{F}_1F_0}$$

$$Operation_0 = ALU_{op1}\overline{F}_3F_2\overline{F}_1F_0 + ALU_{op1}F_3\overline{F}_2F_1\overline{F}_0$$

Designing the main control unit

- Identify the fields of an instruction
- Identify the control lines that are needed for the datapath

Main observations

- Opcode (bits 31:26)
- Registers rs (bits 25:21) and rt (bits 20:16) to be read
- Base register rs (bits 25:21) for load/store instructions
- Offset for branch equal, load, and store (bits 15:0)
- Destination register: rt (bits 20:16) for load instruction, rd (bits 15:11) for R-type instruction

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

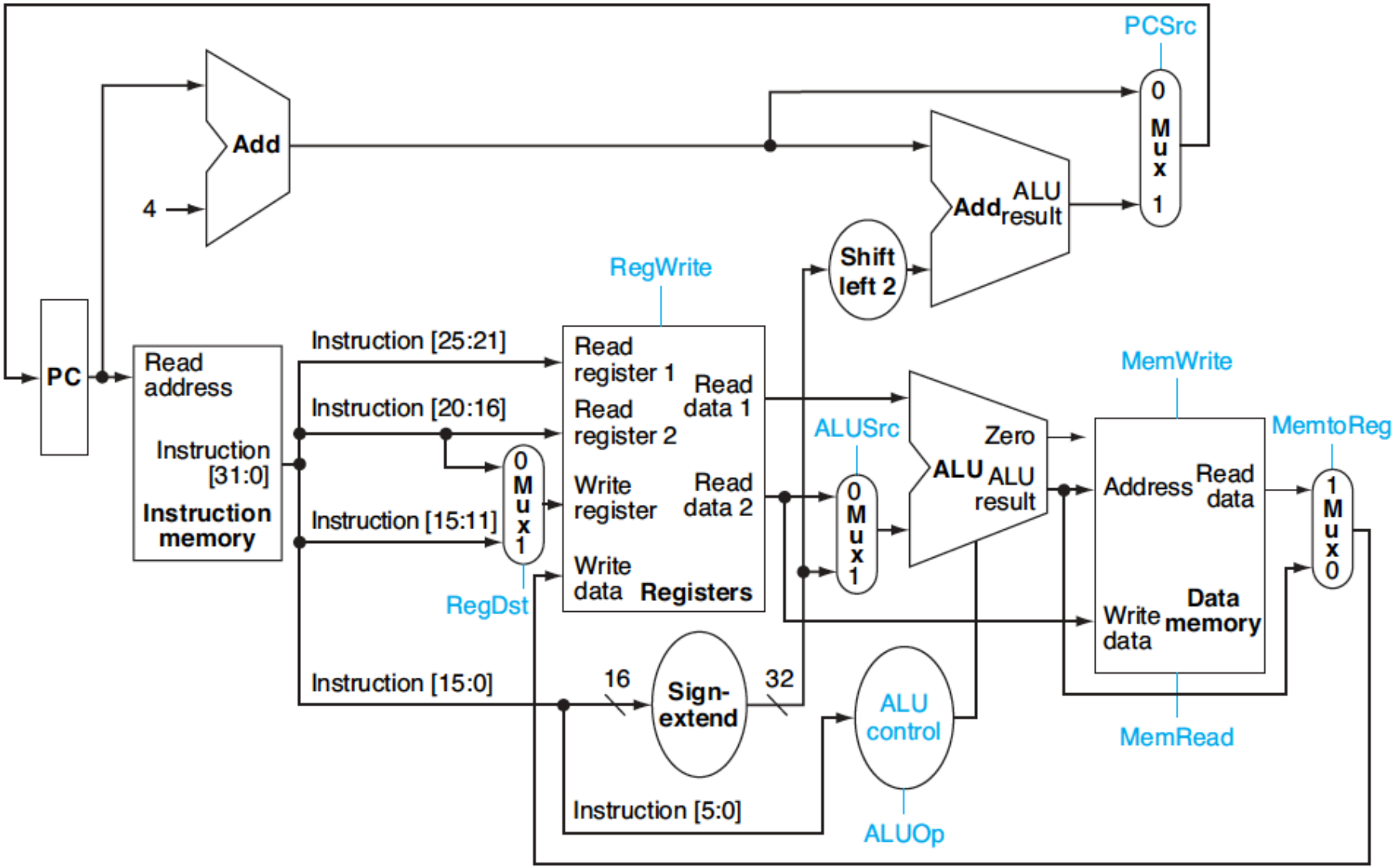
a. R-type instruction

Field	35 or 43	rs	rt	address
Bit positions	31:26	25:21	20:16	15:0

b. Load or store instruction

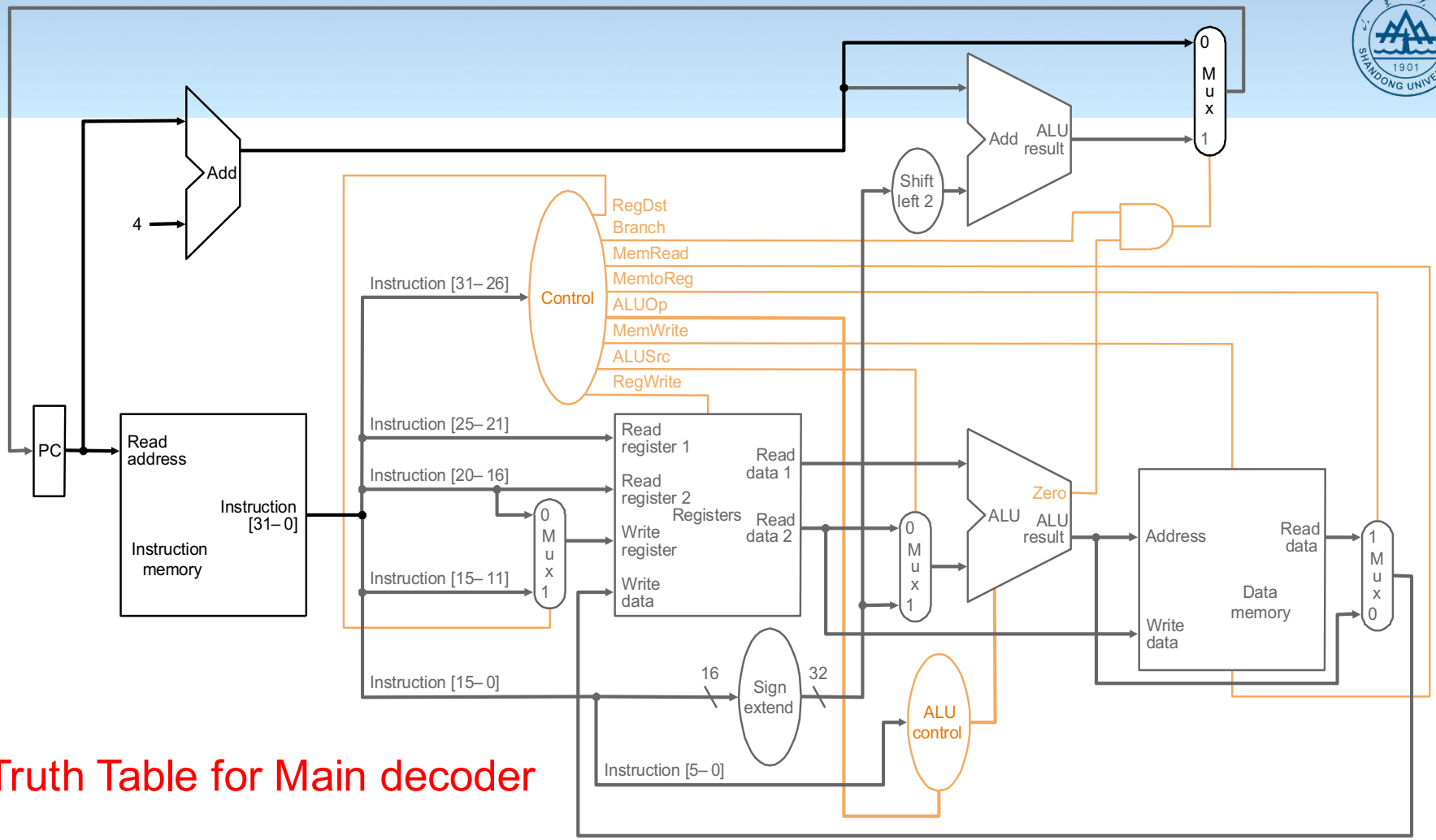
Field	4	rs	rt	address
Bit positions	31:26	25:21	20:16	15:0

c. Branch instruction


PCSrc
Add
4
0
M
u
x
1
ALU
Add result
Shift
left 2
RegWrite
PC
Read
address
Instruction
[31:0]
Instruction
memory
Instruction [25:21]
Instruction [20:16]
Instruction [15:11]
Instruction [15:0]
Instruction [5:0]
Read
register 1
Read
register 2
Write
register
Write
data
Read
data 1
Read
data 2
Registers
RegDst
ALUSrc
Zero
ALU
ALU
result
MemWrite
MemtoReg
Address
Read
data
Write
data
Data
memory
MemRead
16
Sign-
extend
32
ALU
control
ALUOp

The effect of each of the seven control signals

Signal name	Effect when deasserted(=0)	Effect when asserted(=1)
RegDst	The register destination number for the Write register comes from the rt field (bit 20:16)	The register destination number for the Write register comes from the rd field (bit 15:11)
RegWrite	None	Register destination input is written with the value on the Write data input
ALUScr	The second ALU operand come 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 calculates the value PC+4	The PC is replaced by the output of the adder that calculates 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 value on the Write data input.
MemtoReg	The value fed to register Write data input comes from the ALU	The value fed to the register Write data input comes from the data memory.

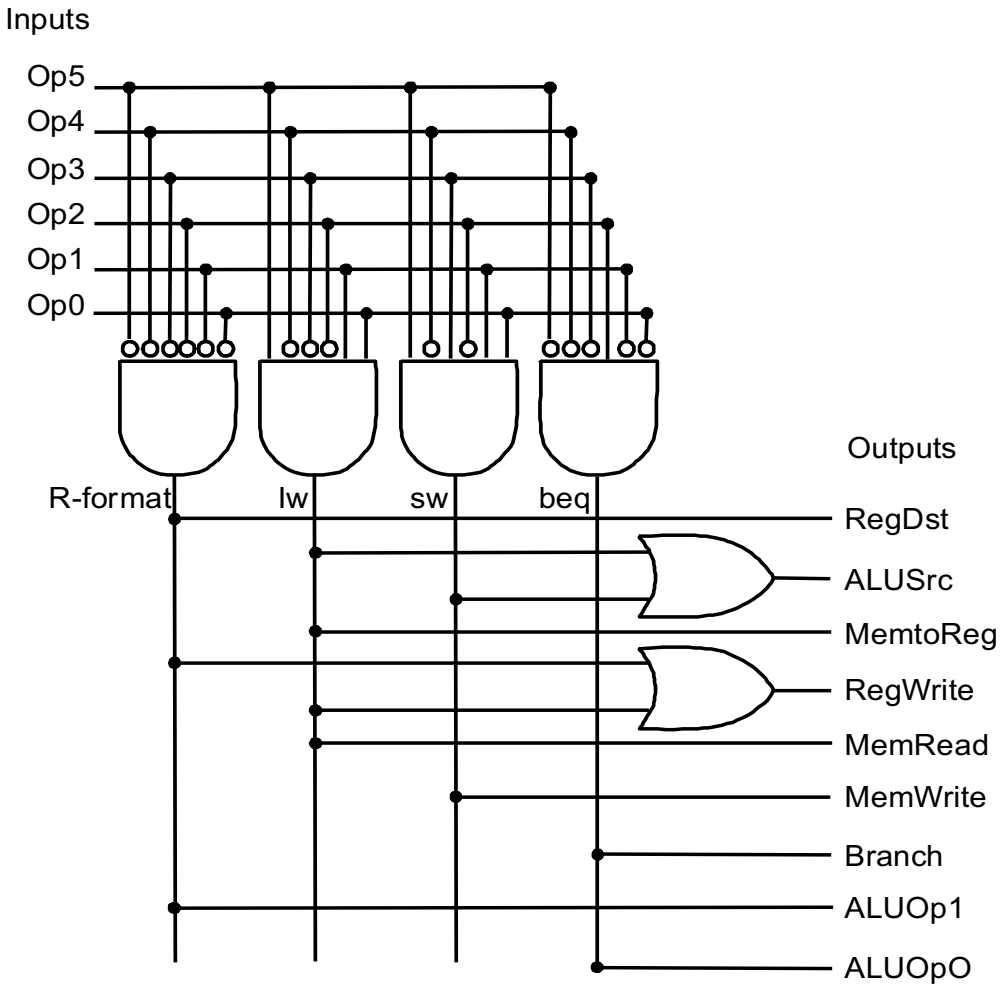


Truth Table for Main decoder

Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
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

Circuitry of main Controller

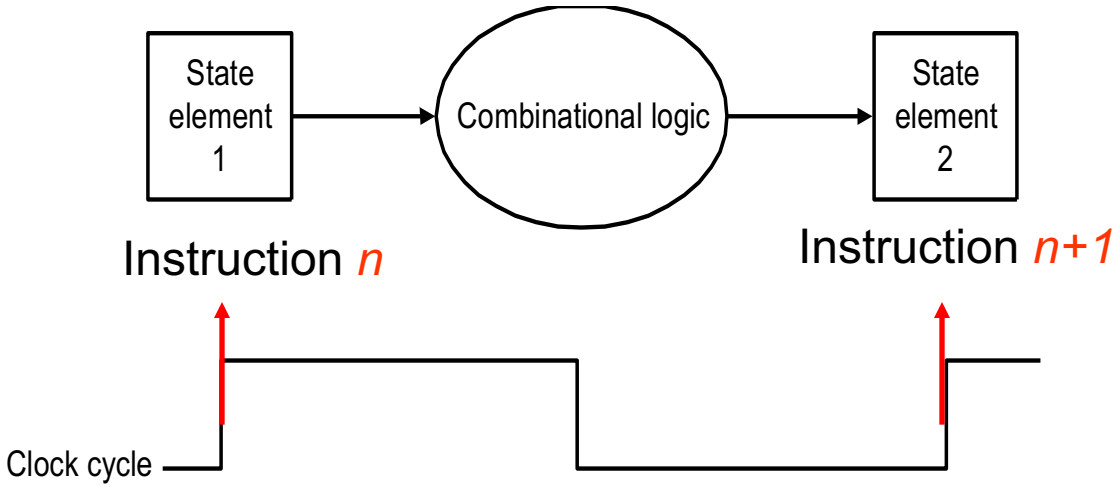
- Simple combinational logic (truth tables)



opcode	output	
000000	R-format	
100011	lw	
101011	sw	
000100	beq	

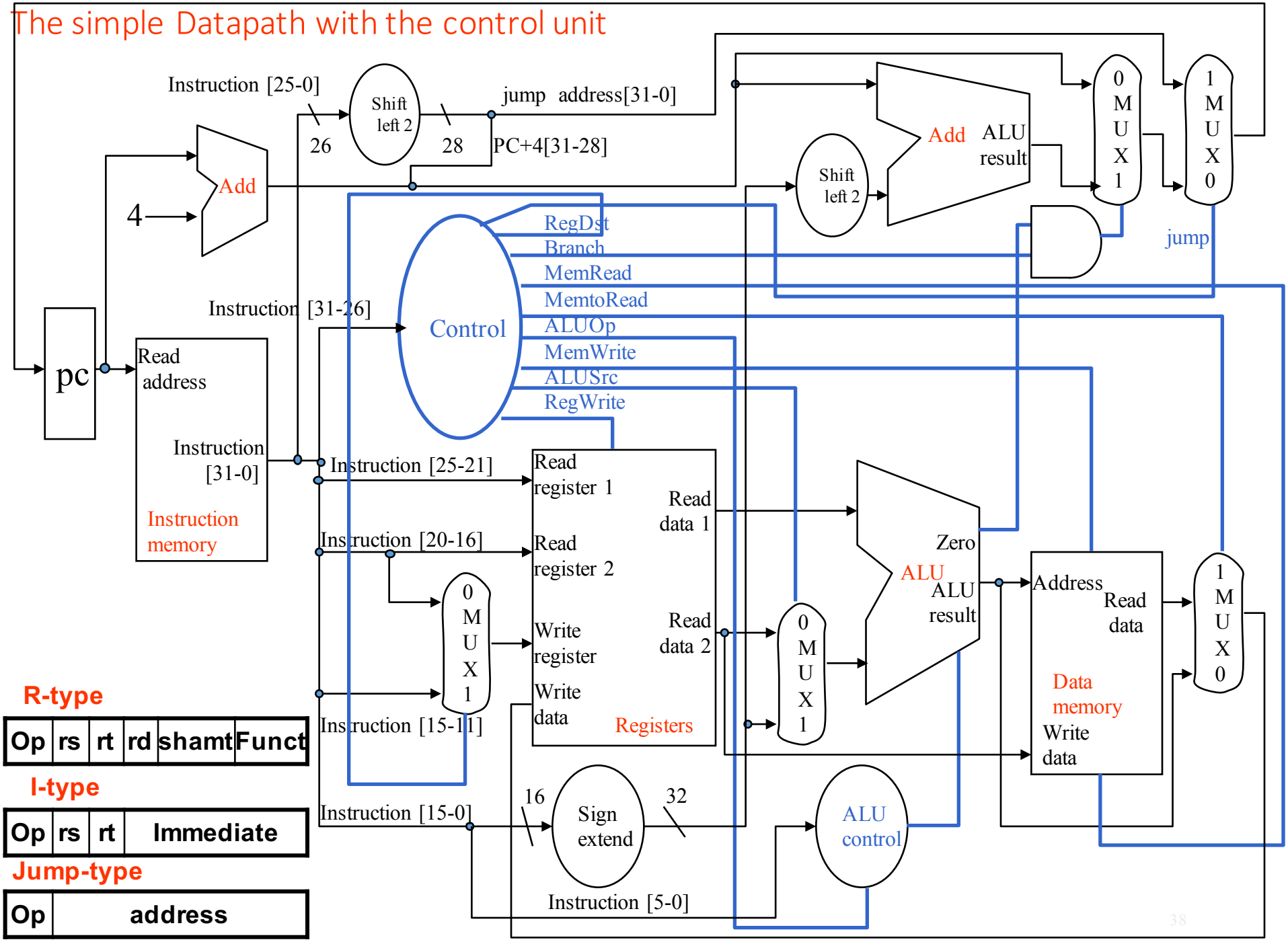
Our Simple Control Structure

- All of the logic is combinational
- We wait for everything to settle down, and the right thing to be done
  - ALU might not produce right answer? right away
  - we use write signals along with clock to determine when to write
- Cycle time determined by length of the longest path

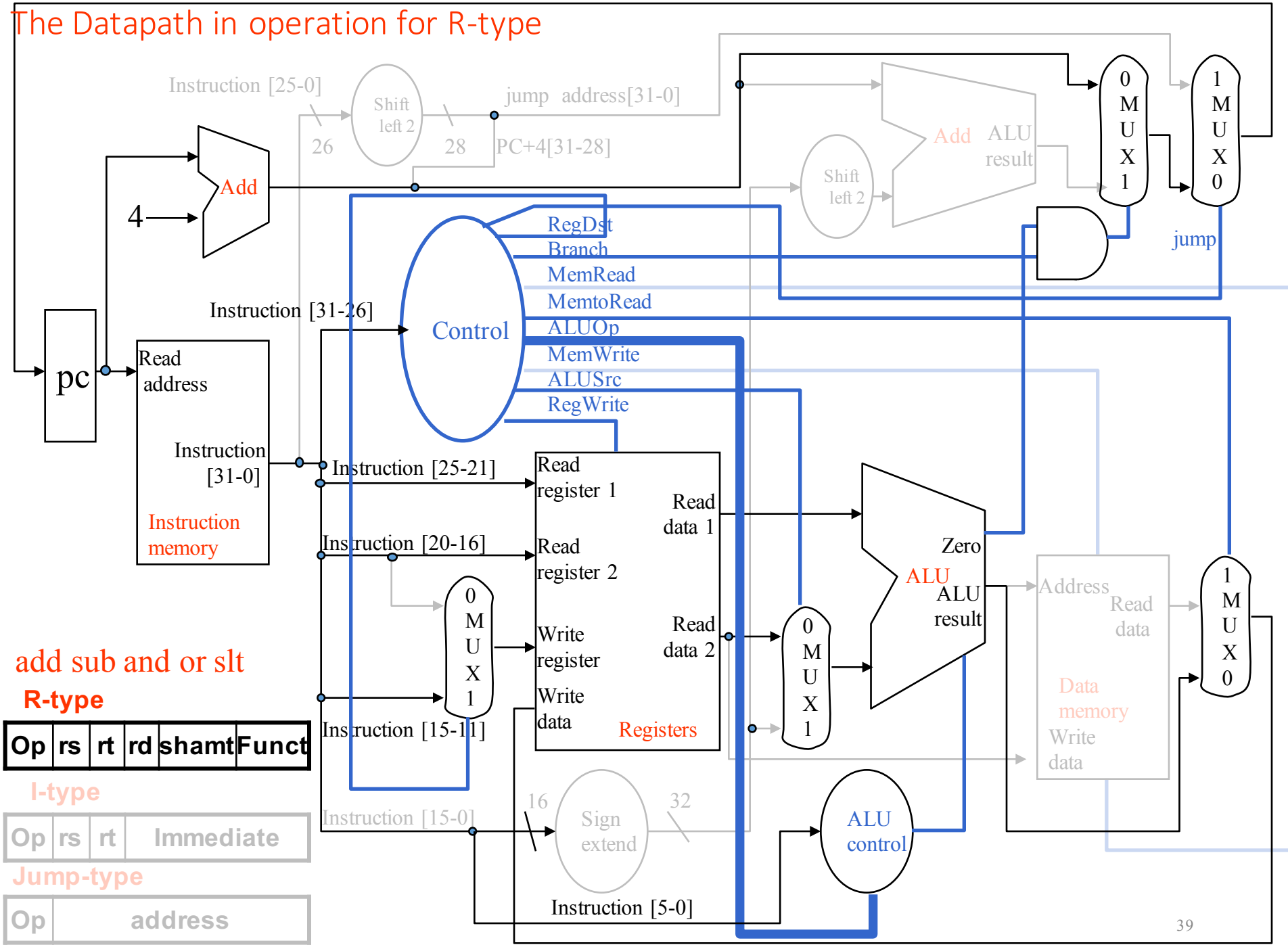


We are ignoring some details like setup and hold times

The simple Datapath with the control unit



The Datapath in operation for R-type



add sub and or slt

R-type

Op	rs	rt	rd	shamt	Funct

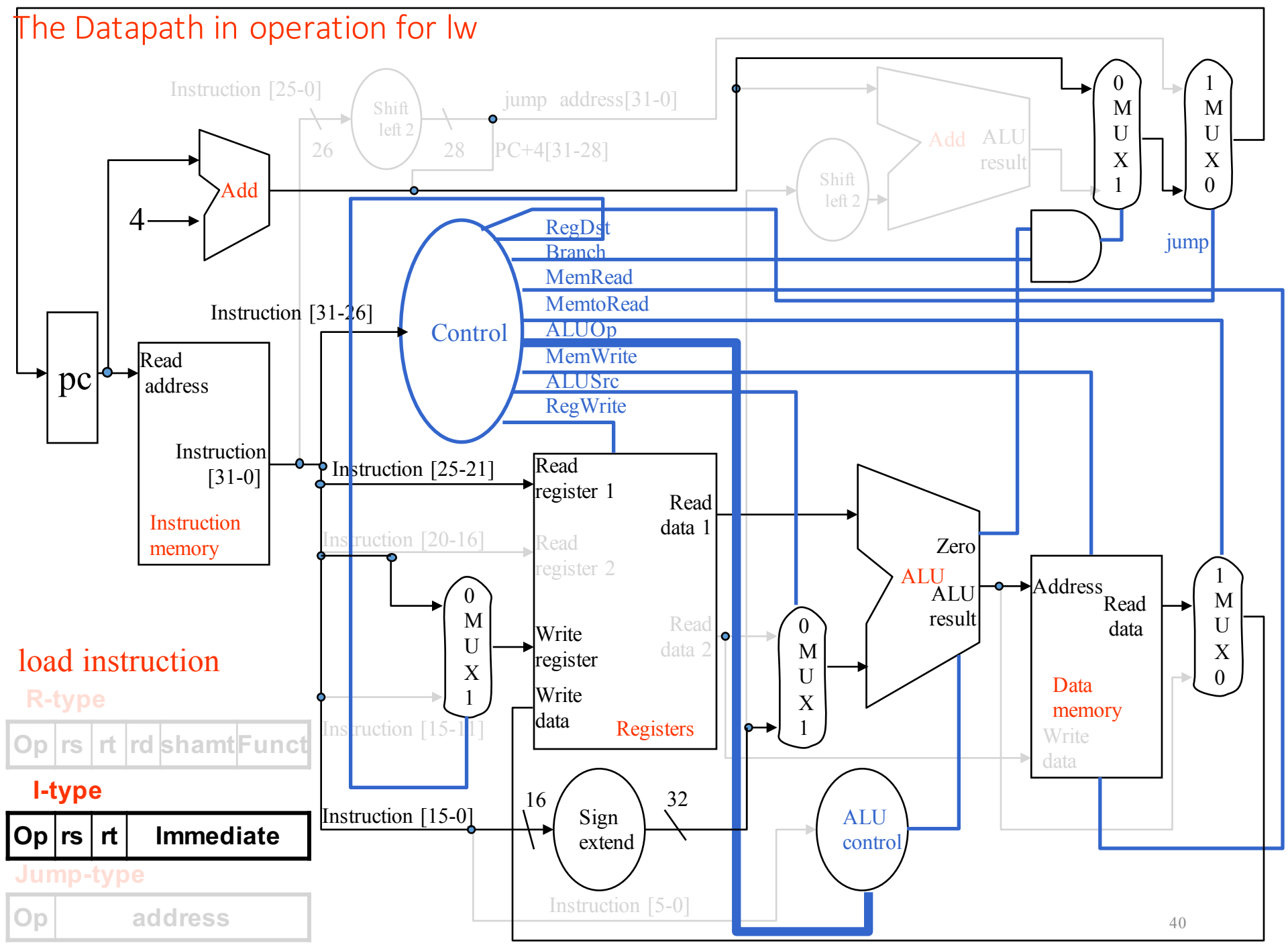
I-type

Op	rs	rt	Immediate

Jump-type

Op	address

The Datapath in operation for lw



load instruction

R-type

Op	rs	rt	rd	shamt	Funct

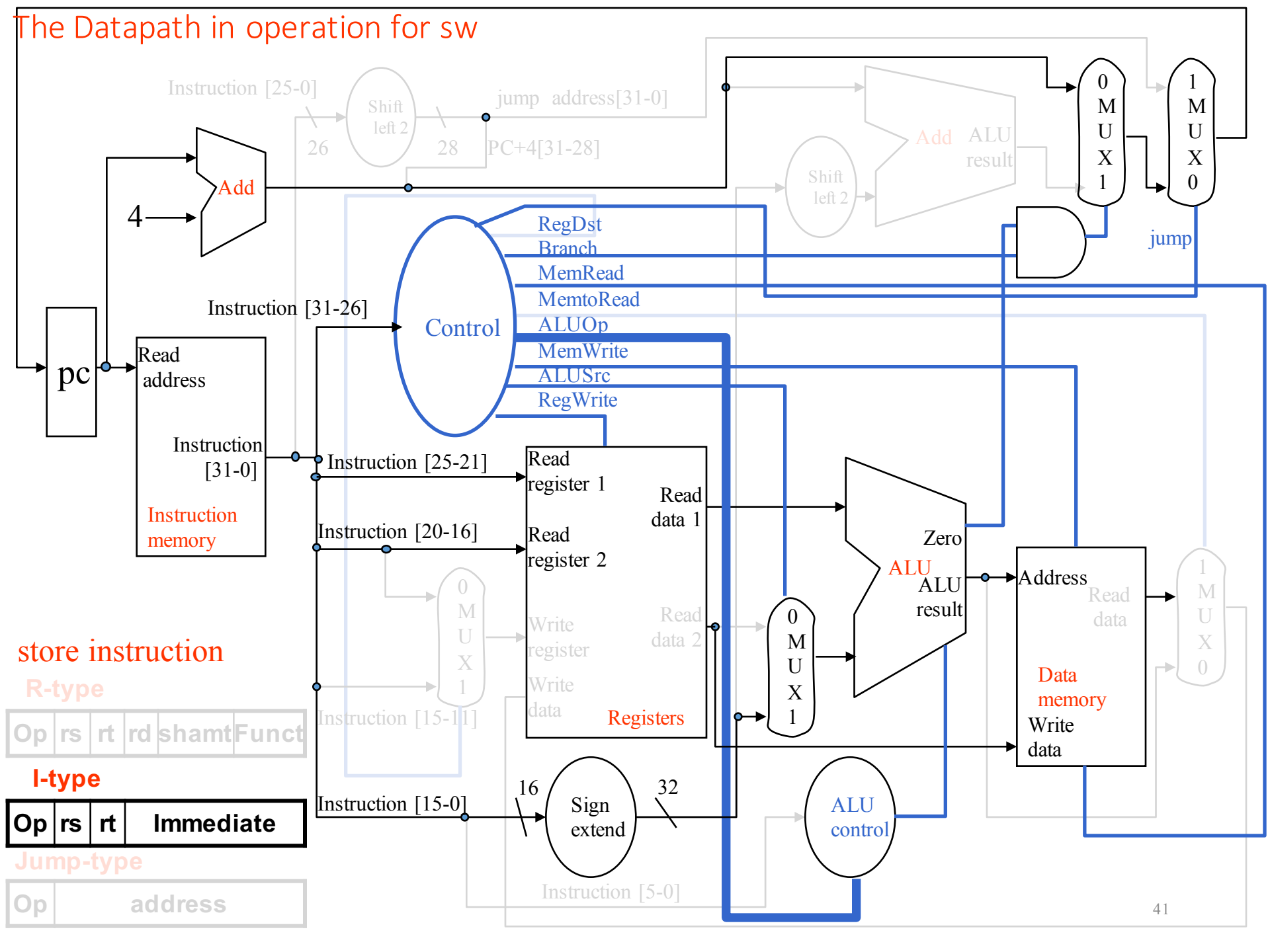
I-type

Op	rs	rt	Immediate

Jump-type

Op	address

The Datapath in operation for sw



store instruction

R-type

Op	rs	rt	rd	shamt	Funct

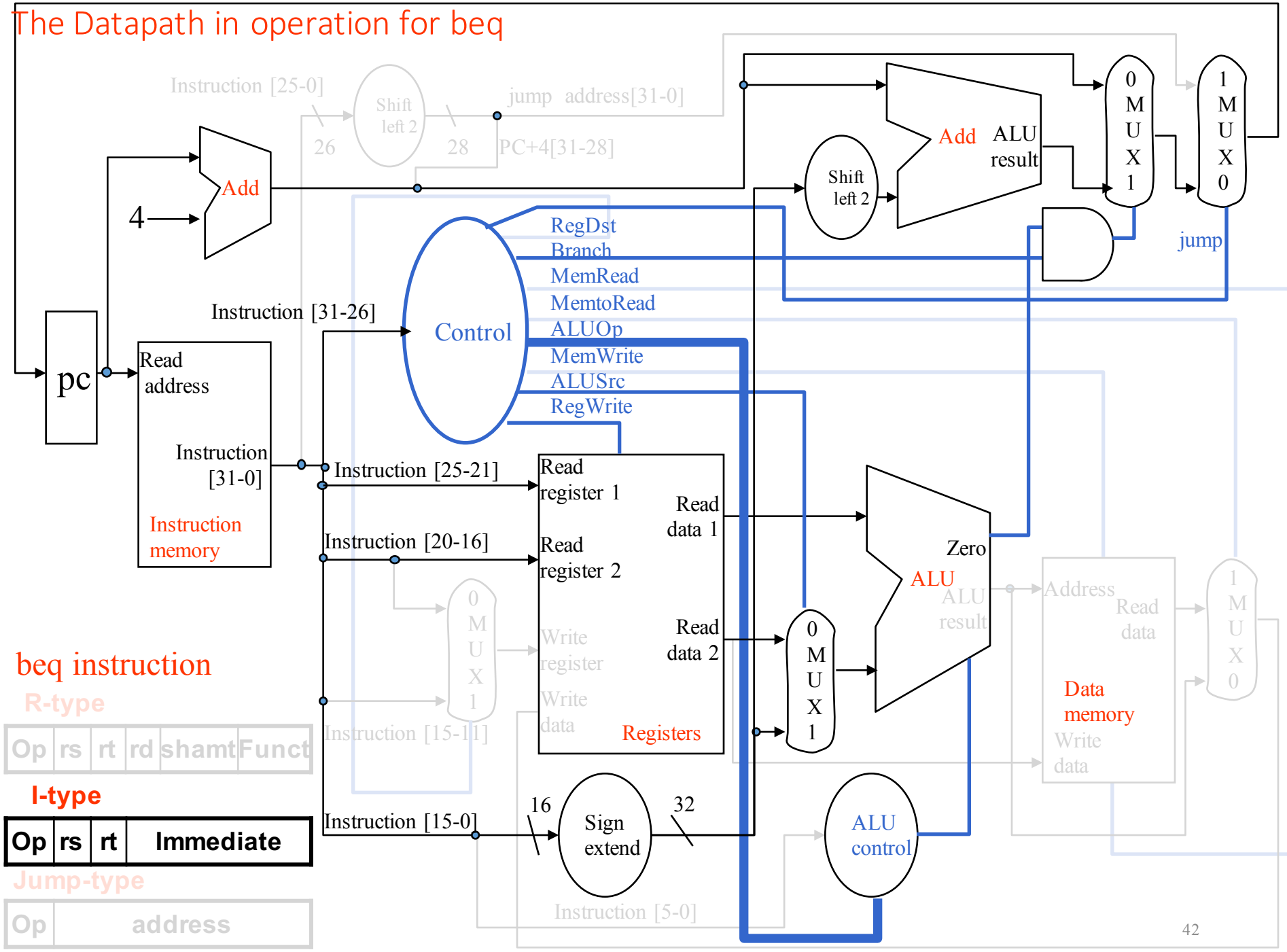
I-type

Op	rs	rt	Immediate

Jump-type

Op	address

The Datapath in operation for beq



beq instruction

R-type

Op	rs	rt	rd	shamt	Funct

I-type

Op	rs	rt	Immediate

Jump-type

Op	address

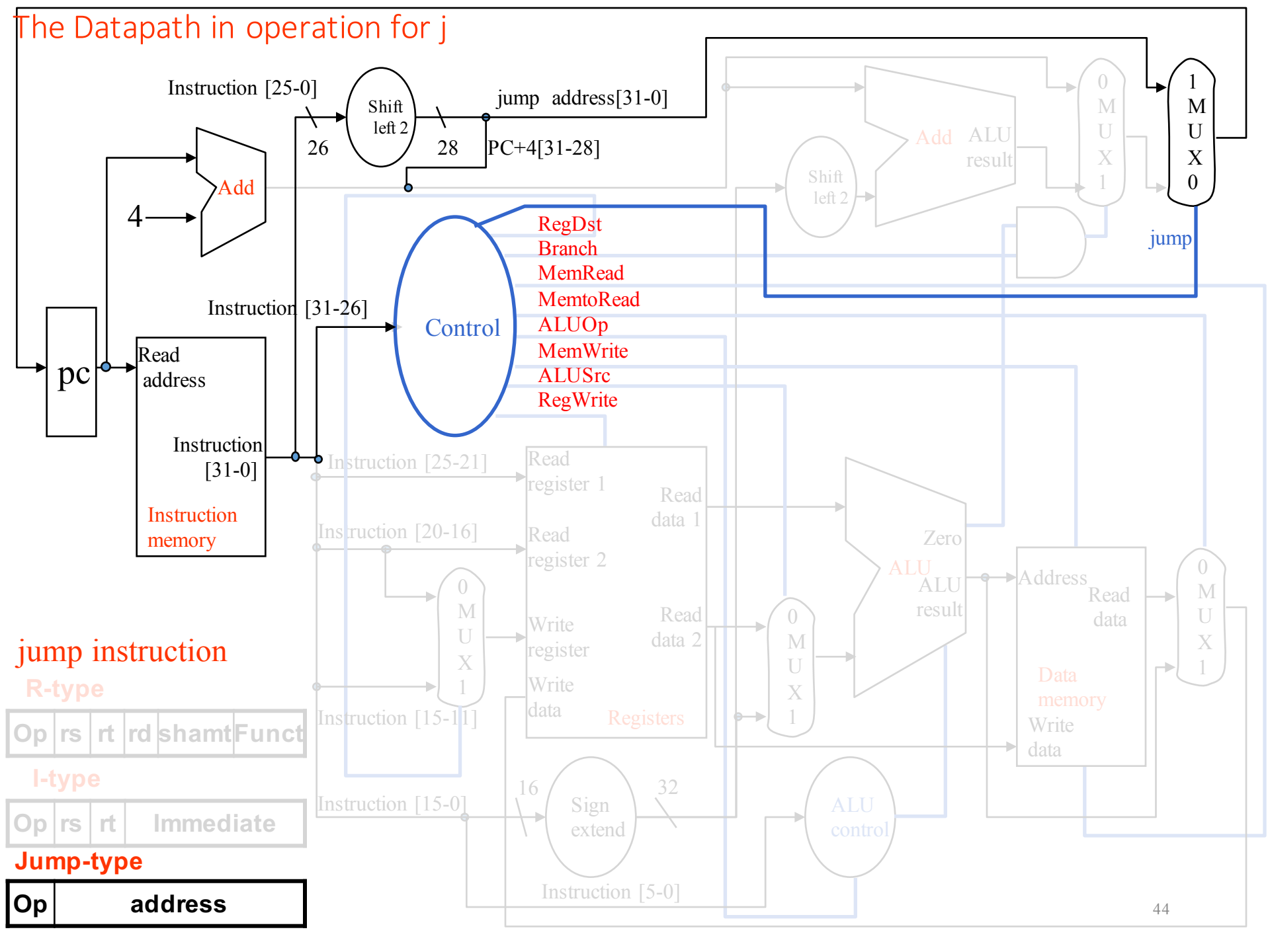
j instruction

- instruction format
  - j Label

(000010)2	26 bits address

- Implementation
  - The upper 4 bits of the current PC+4
  - The 26-bit immediate field of the jump instruction
  - The bits 00_{two}

The Datapath in operation for j



jump instruction

R-type

Op	rs	rt	rd	shamt	Funct

I-type

Op	rs	rt	Immediate

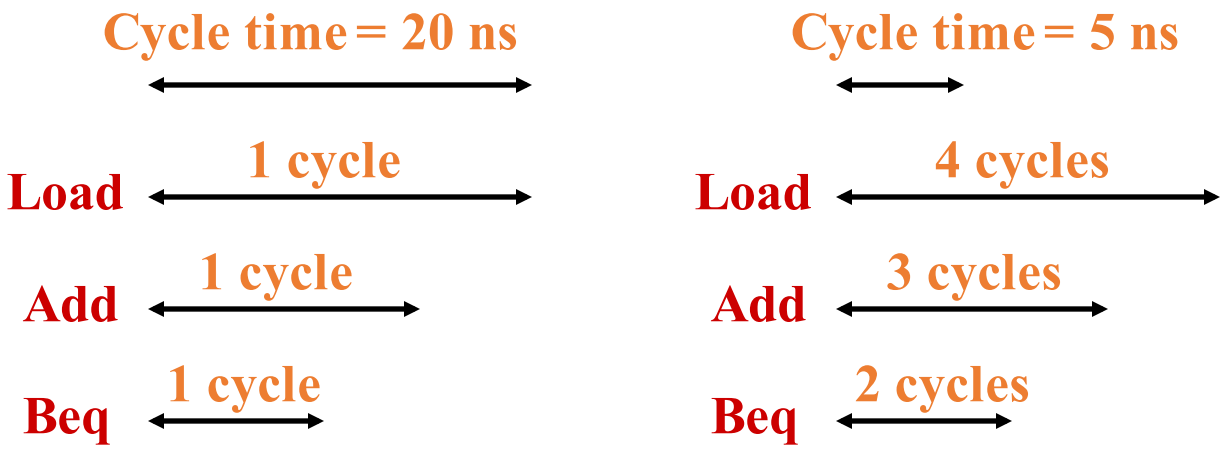
Jump-type

Op	address

Single Vs. Multi-Cycle Machine

- In this implementation, every instruction requires one cycle to complete ➔ cycle time = time taken for the slowest instruction

- If the execution was broken into multiple (faster) cycles, the shorter instructions can finish sooner



Single Cycle Problems

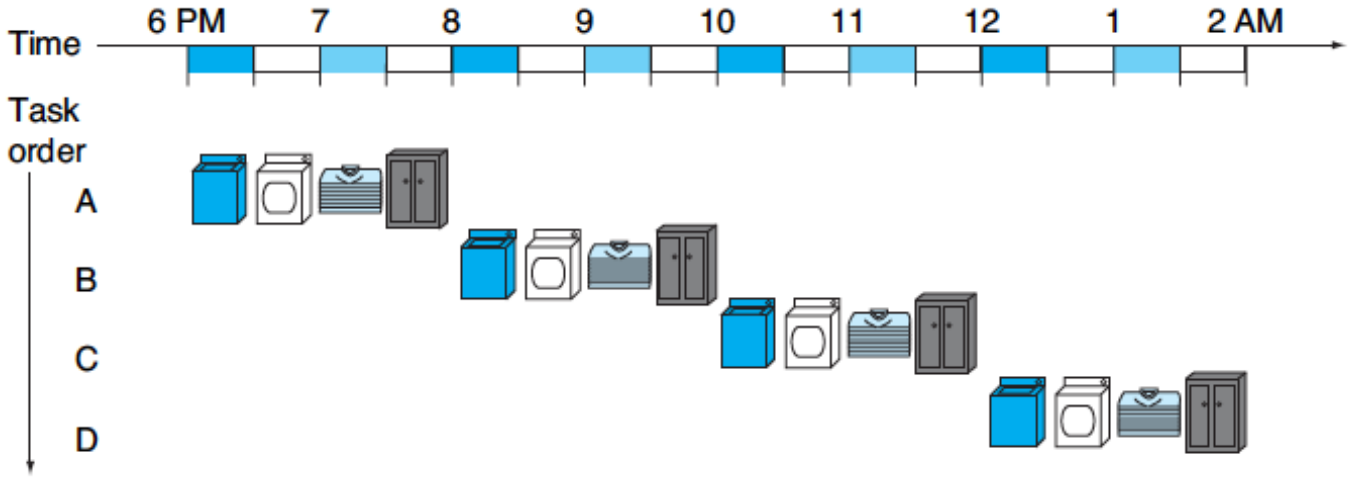
- what if we had a more complicated instruction like floating point?
  - If so, the waste of time will be more serious.
- The reason is the following:
  - Let's see the instruction 'mult'
  - This instruction needs to use the ALU repeatedly.

An Overview of Pipelining

- Pipelining is an implementation technique in which multiple instructions are overlapped in execution

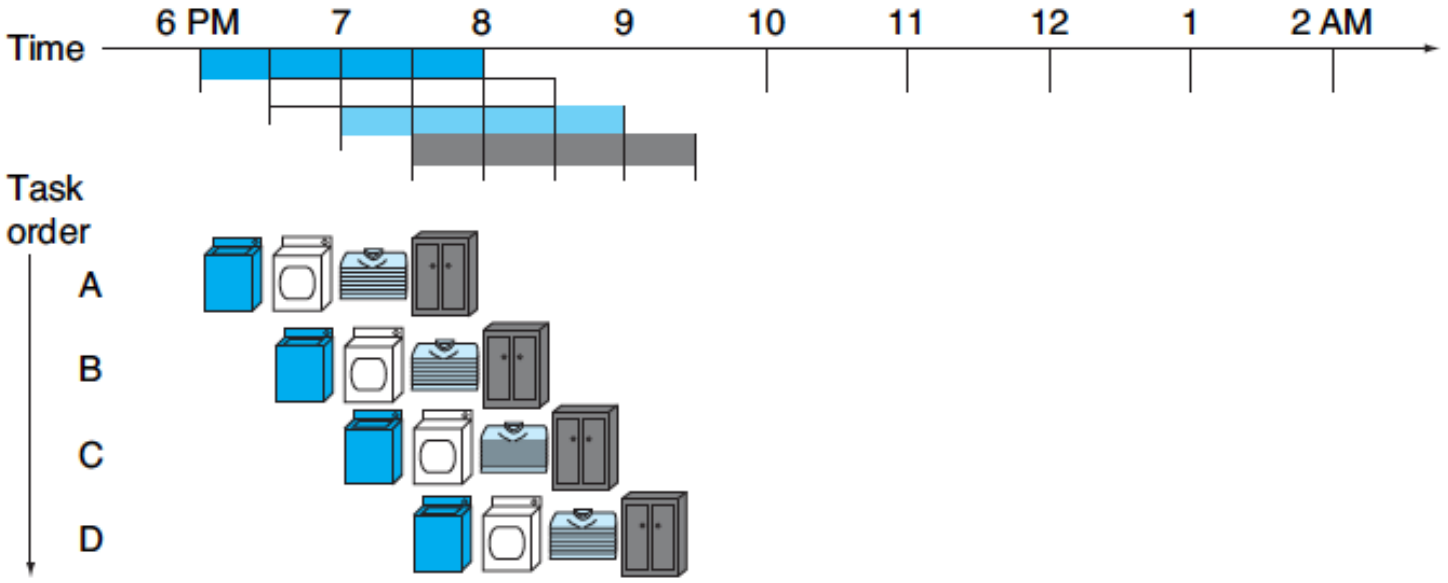
The laundry analogy for pipelining

- Place one dirty load of clothes in the washer
- When the washer is finished, place the wet load in the dryer
- When the dryer is finished, place the dry load on a table and fold
- When folding is finished, ask your roommate to put the clothes away

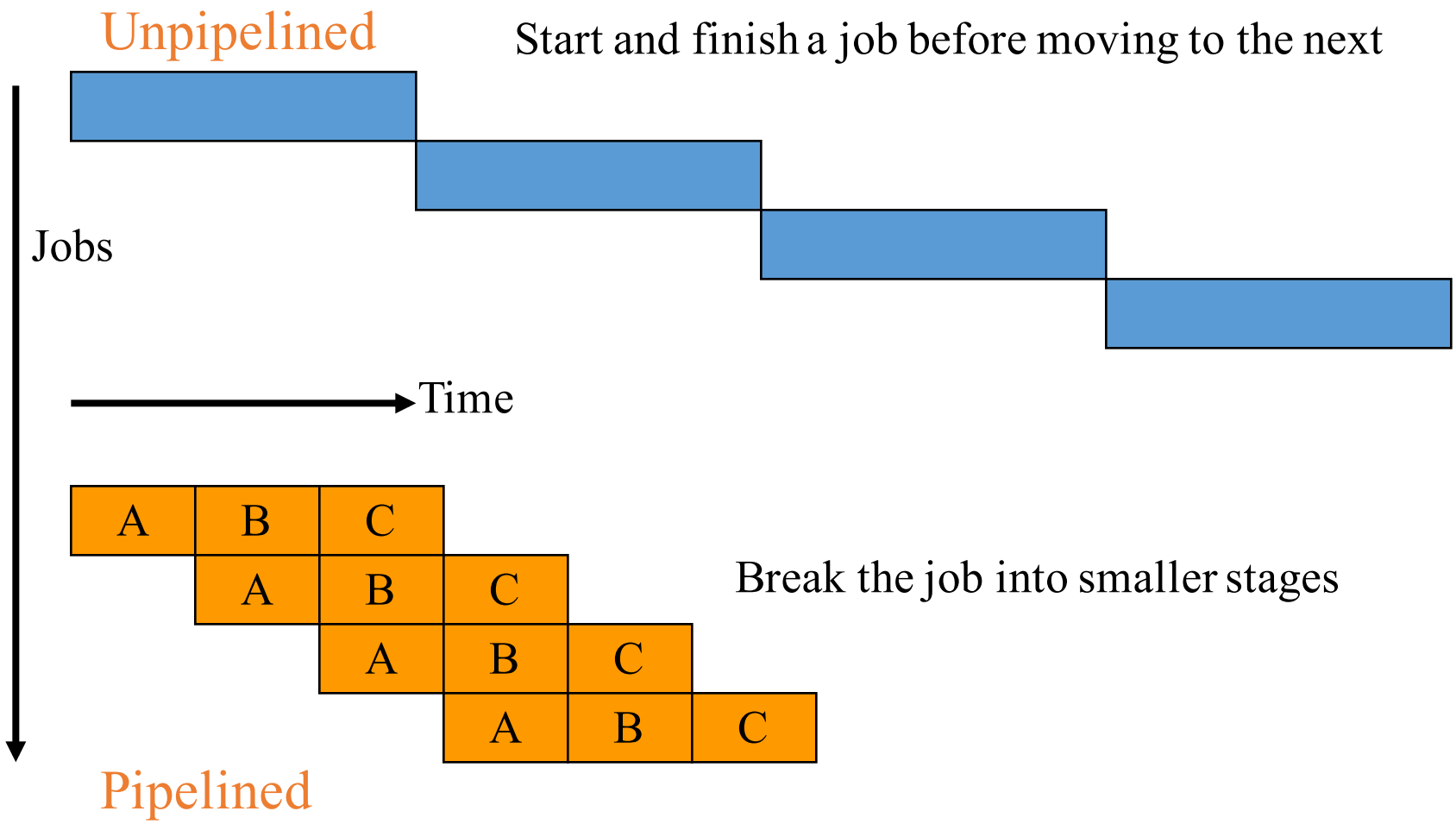


Pipelining paradox

- The time cost for handling a single dirty load is not shorter for pipelining
- The pipelining for many loads is faster, since everything is working in parallel, such that more loads are finished per hour



An Overview of Pipelining



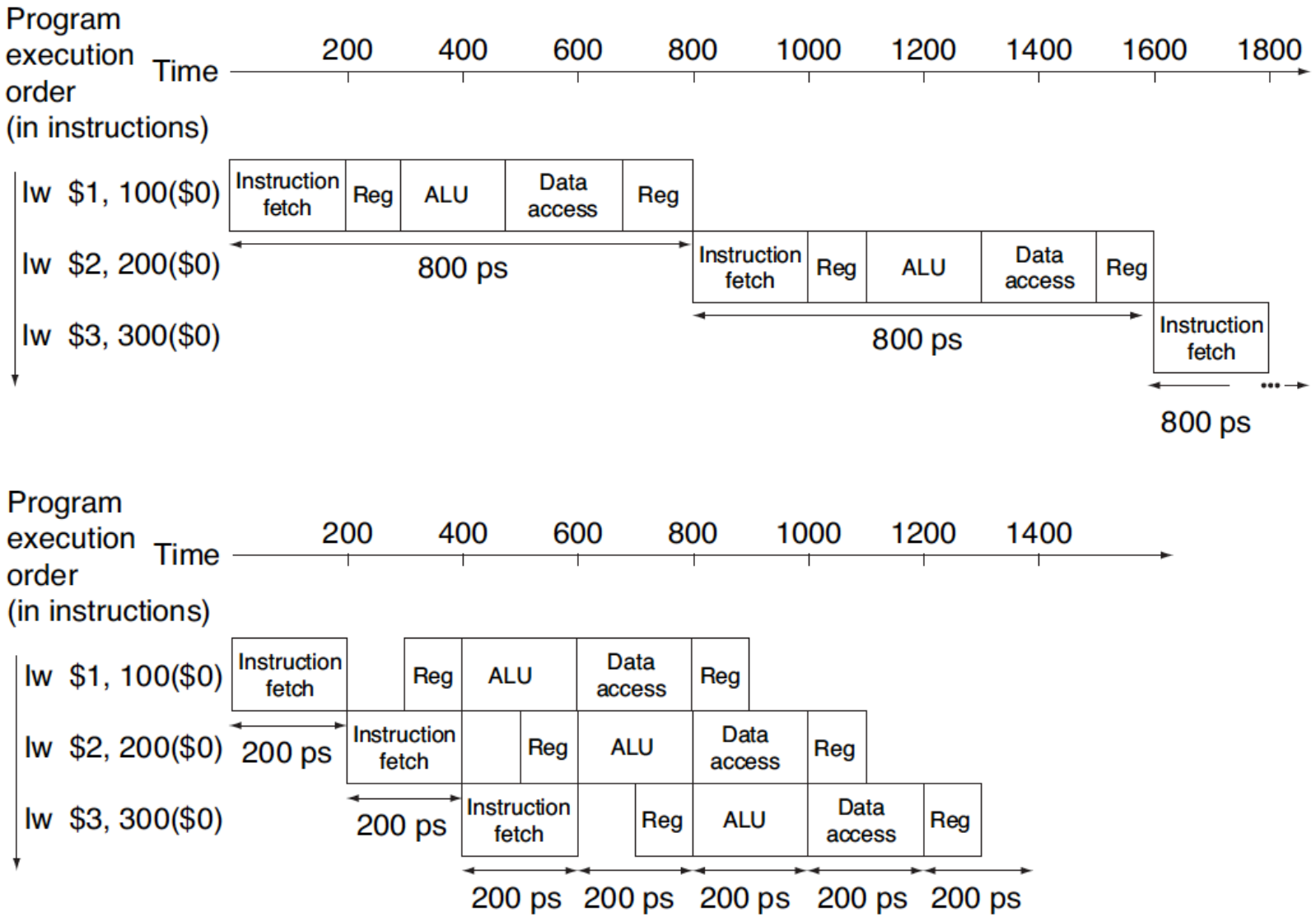
Applying pipelining to processors

- A MIPS instruction takes five steps
  - IF (Instruction Fetch): Fetch instruction from memory
  - ID (Instruction Decoding): Read registers while decoding the instruction
  - EX (ALU Execution): Execute the operation or calculate an address
  - MEM (Memory Access): Access an operand in data memory
  - WB (Write Back to Register): Write the result into a register

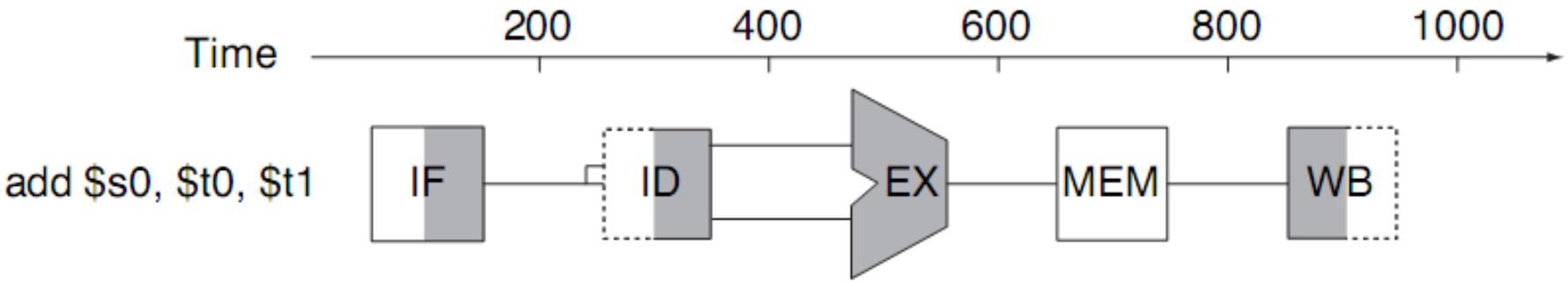
Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (`lw`)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (`sw`)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (`add`, `sub`, `AND`, `OR`, `slt`)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch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

FIGURE 4.26 Total time for each instruction calculated from the time for each component.
This calculation assumes that the multiplexors, control unit, PC accesses, and sign extension unit have no delay.

Single-cycle, nonpipelined execution versus pipelined execution



A 5-Stage Pipeline



- **IF: Instruction Fetch**
- **ID: Instruction Decoding**
- **EX: ALU Execution**
- **MEM: memory access**
- **WB: Write Back to Reg**

What would happen if we increased the number of instructions?

Fro example, 1 000 003 instructions.

Total execution time$_{\text{pipelined}}$ = 200 001 400 ps

Total execution time$_{\text{nonpipelined}}$= 800 002 400 ps

$$\frac{800\ 002\ 400\ ps}{200\ 001\ 400\ ps} \approx 4$$

Pipelining improves performance by increasing instruction throughput, as opposed to decreasing the execution time of an individual instruction.

Designing instruction sets for pipelining

- All MIPS instructions are the same length
- MIPS has only a few instruction formats, with the source register fields being located in the same place in each instruction
- Memory operands only appear in loads or stores in MIPS
- Operands must be aligned in memory

Pipeline hazards

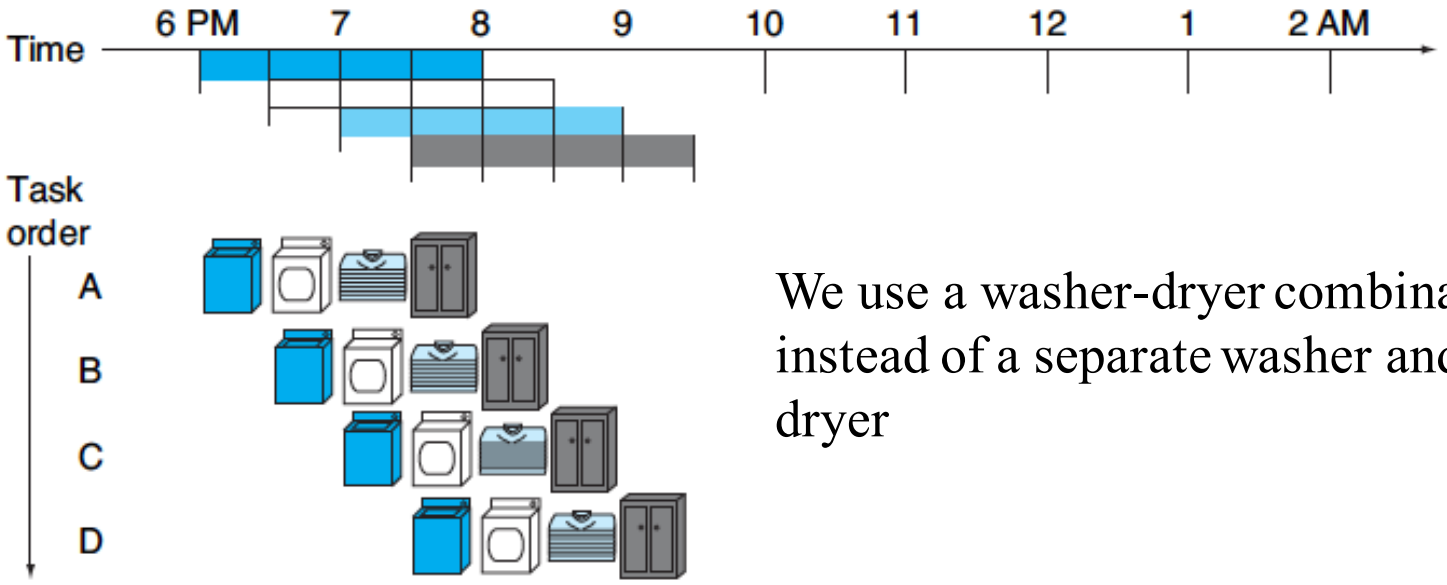
- Hazards: The next instruction cannot execute in the following clock cycle
  - Structural hazard
  - Data hazard
  - Control hazard

Hazards

- Structural hazards: different instructions in different stages (or the same stage) conflicting for the same resource
- Data hazards: an instruction cannot continue because it needs a value that has not yet been generated by an earlier instruction
- Control hazard: fetch cannot continue because it does not know the outcome of an earlier branch – special case of a data hazard – separate category because they are treated in different ways

Structure hazard

When a planned instruction cannot execute in the proper clock cycle because the hardware does not support the combination of instructions that are set to execute



We use a washer-dryer combination instead of a separate washer and dryer

Data hazard

When a planned instruction cannot execute in the proper clock cycle because data that is needed to execute the instruction is not yet available

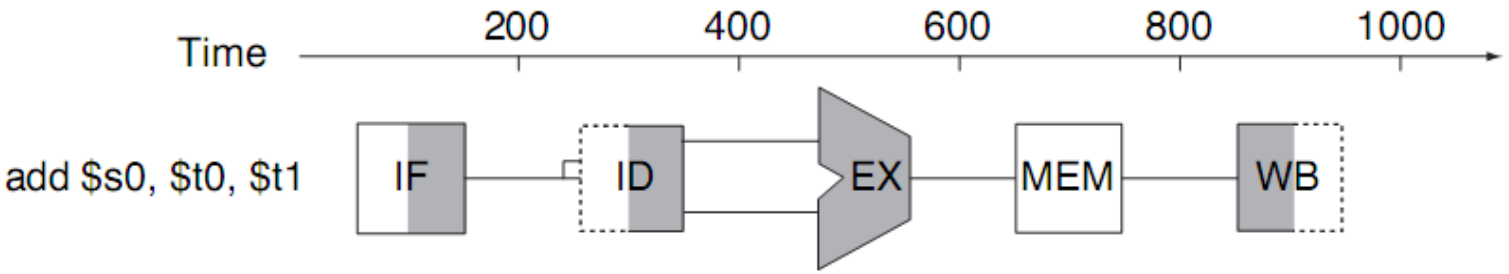
Example: add $s0, $t0, $t1

sub $t2, $s0, $t3

Solution: We do not have to wait for the instruction to complete before trying to resolve the data hazard. E.g., as soon as the ALU creates the sum for the add, we can supply it as an input for the subtract.

Forwarding (or **bypassing**):

A method of resolving a data hazard by retrieving the missing data element from internal buffers rather than waiting for it to arrive from programmer-visible registers or memory



- IF: Instruction Fetch
- ID: Instruction Decoding
- EX: ALU Execution
- MEM: memory access
- WB: Write Back to Register

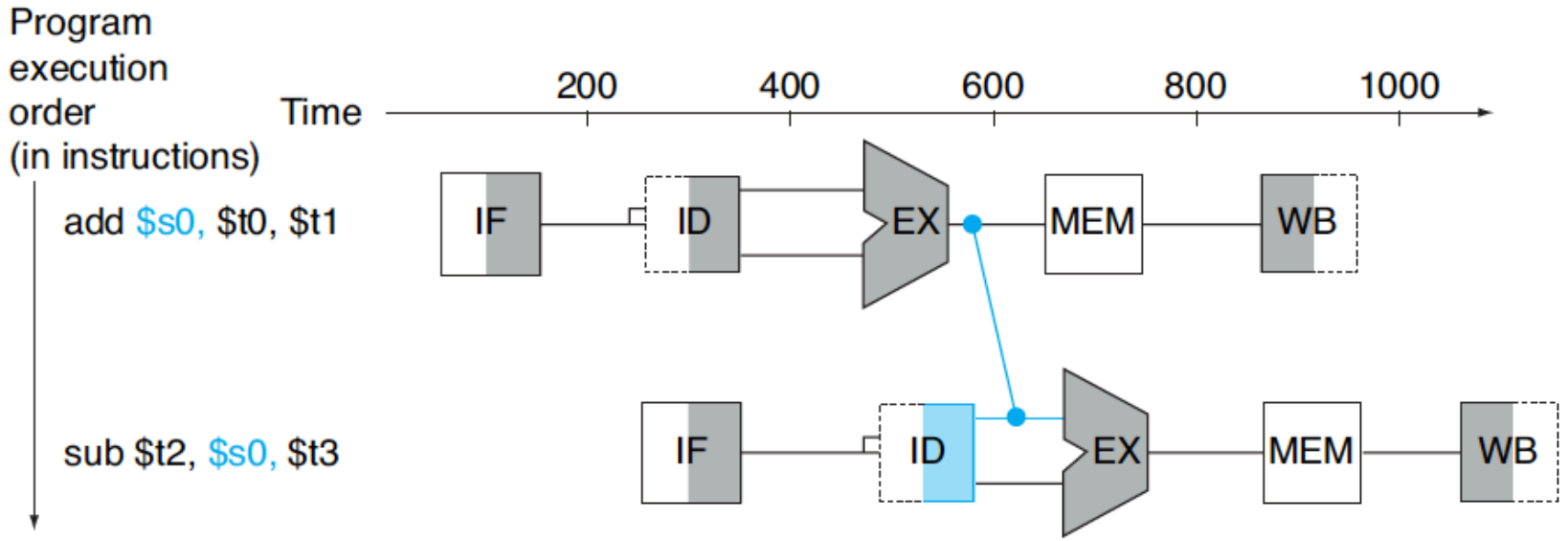
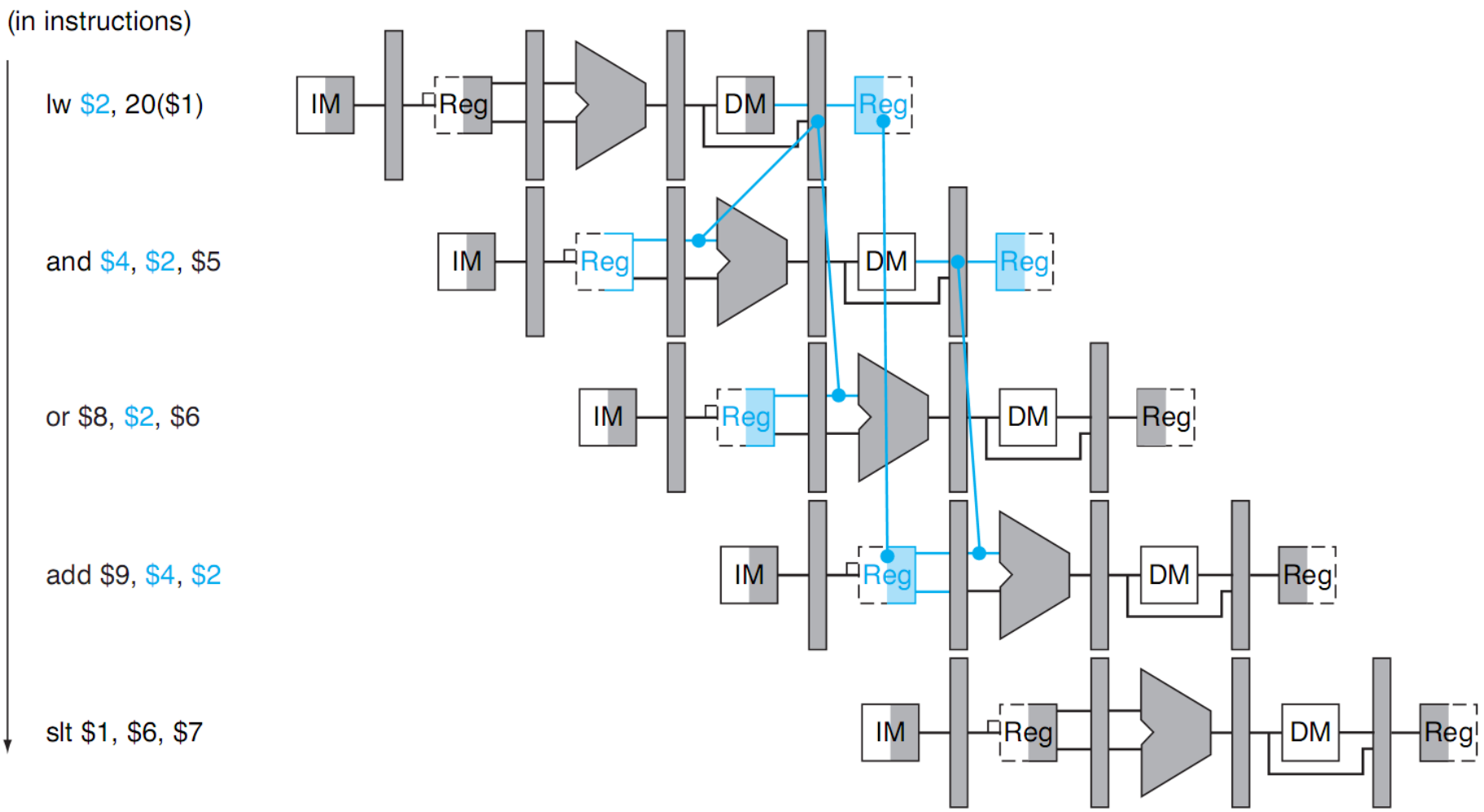
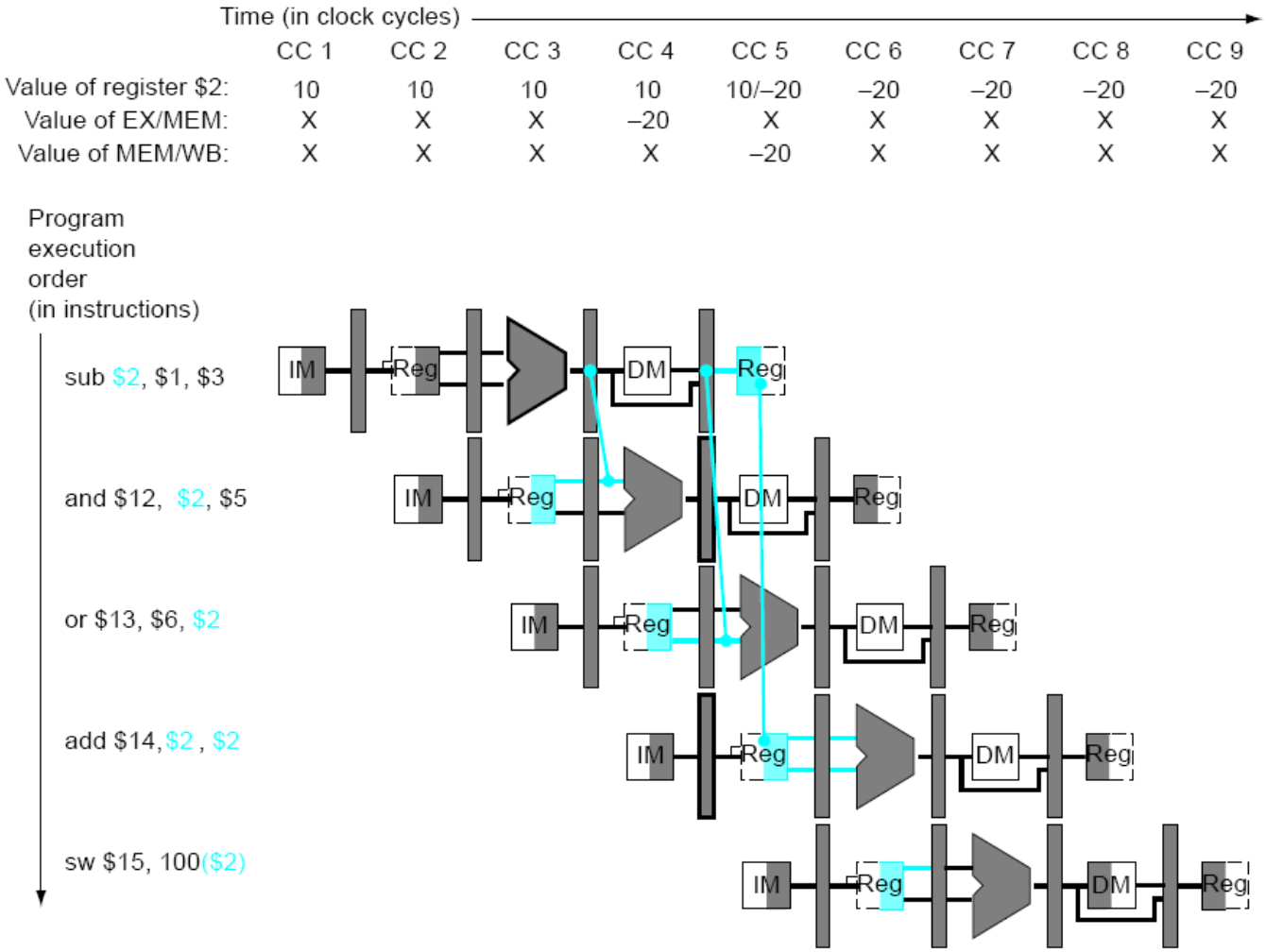


FIGURE 4.29 Graphical representation of forwarding. The connection shows the forwarding path from the output of the EX stage of `add` to the input of the EX stage for `sub`, replacing the value from register `$s0` read in the second stage of `sub`.

Data Hazards



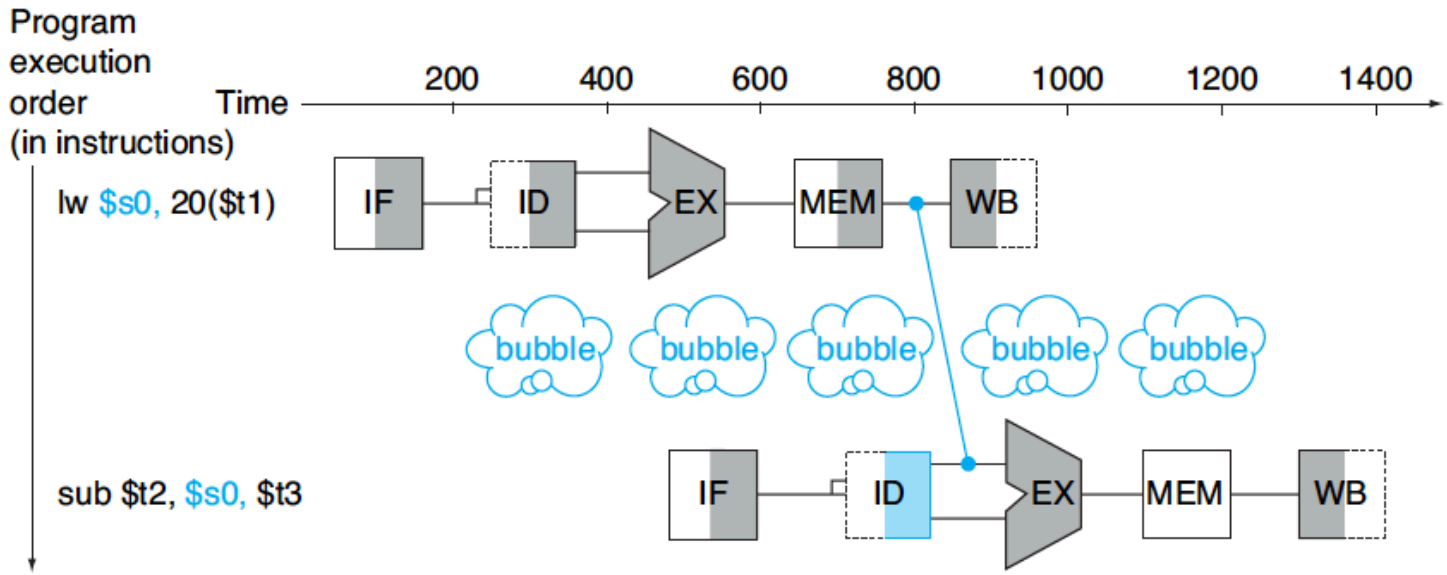
Bypassing



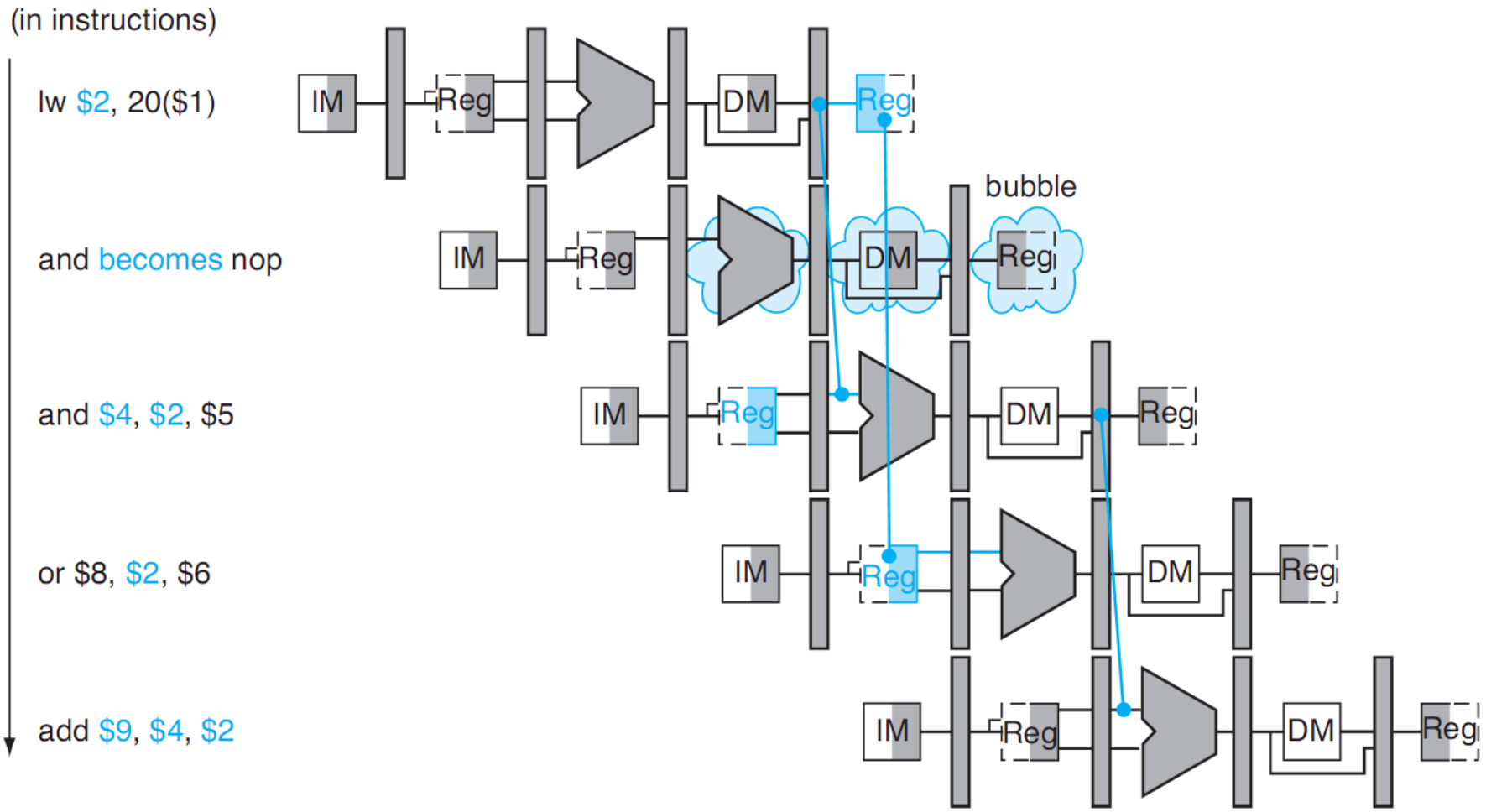
- **Some data hazard stalls can be eliminated: bypassing**

Load-use data hazard

- A specific form of data hazard in which the data being loaded by a load instruction has not yet become available when it is needed by another instruction
- Solution: Pipeline stall (also called "bubble")



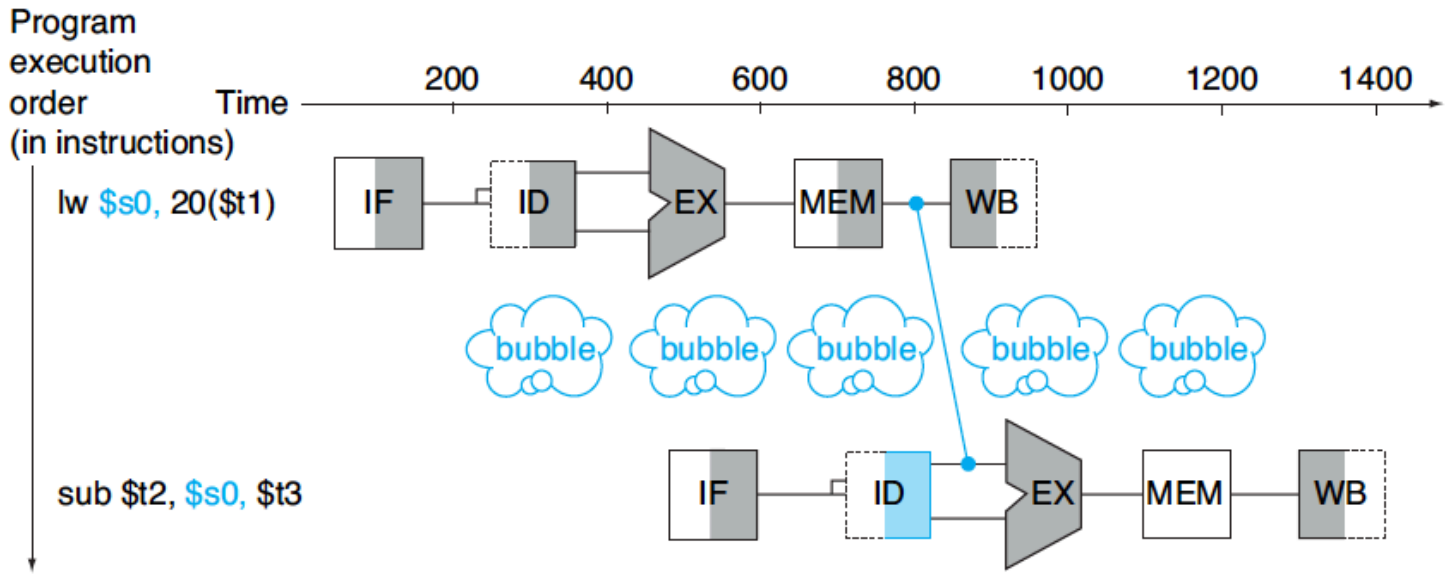
Example - bubble



A bubble is inserted beginning in clock cycle 4, by changing the and instruction to a nop.

Load-use data hazard

- A specific form of data hazard in which the data being loaded by a load instruction has not yet become available when it is needed by another instruction
- Solution: Pipeline stall (also called "bubble")



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

Assume that all variables are in memory and are addressable as offsets from $t0

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

- When a proper instruction cannot execute in the proper pipeline clock cycle because the instruction that was fetched is not the one that is needed; that is the flow of instruction addresses is not what the pipeline expected

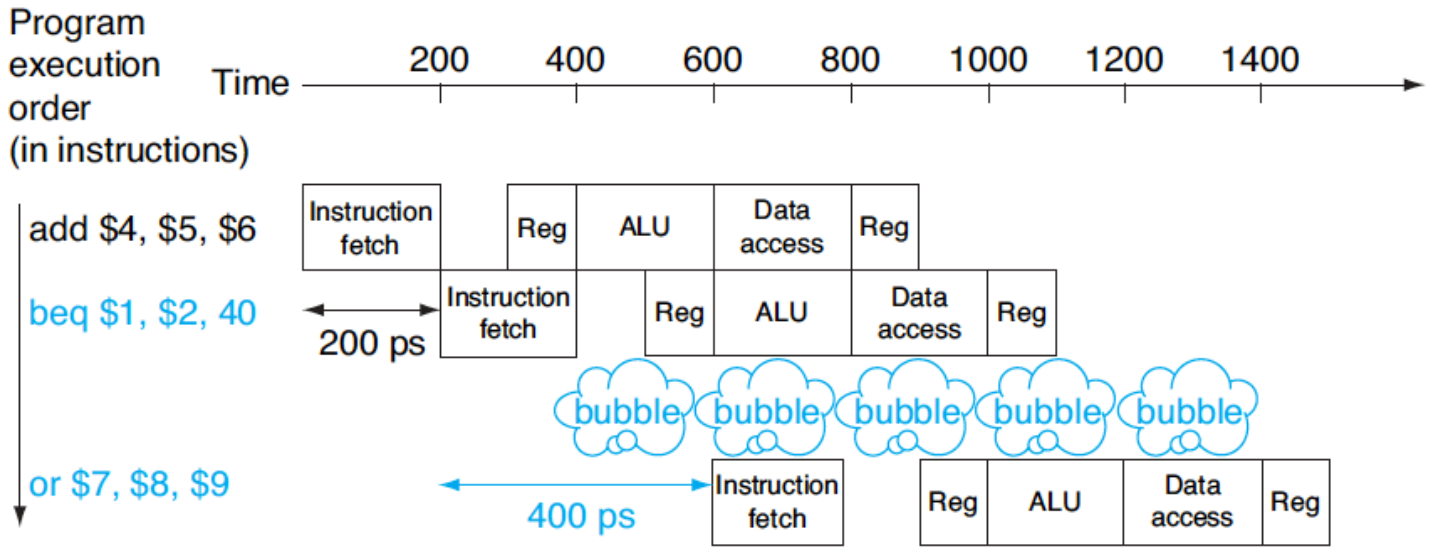
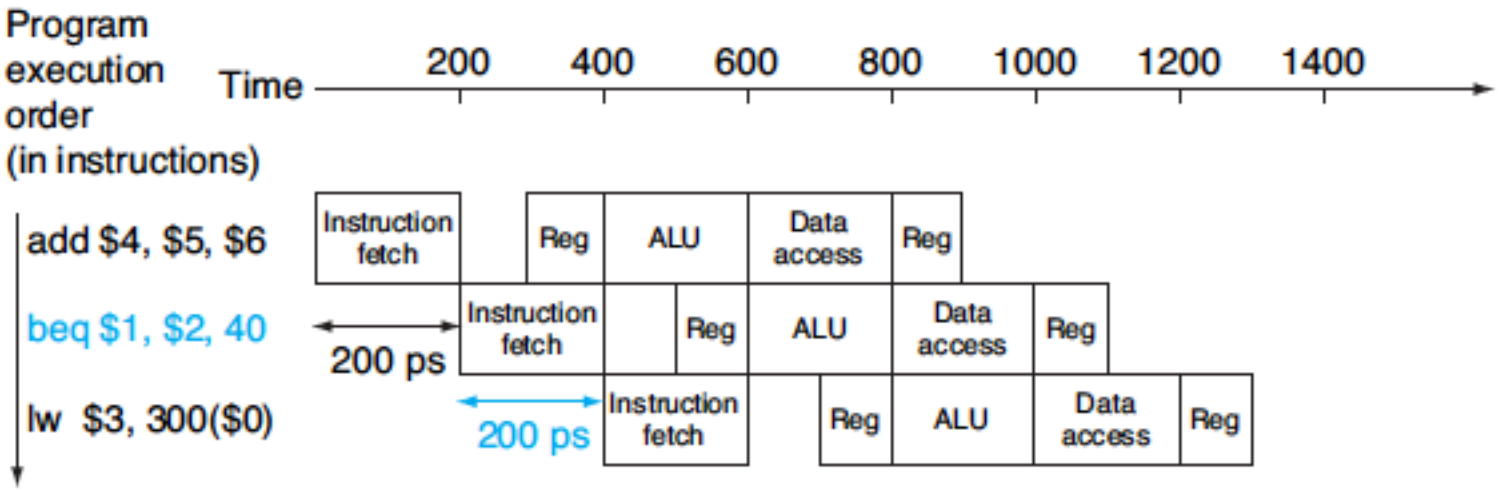
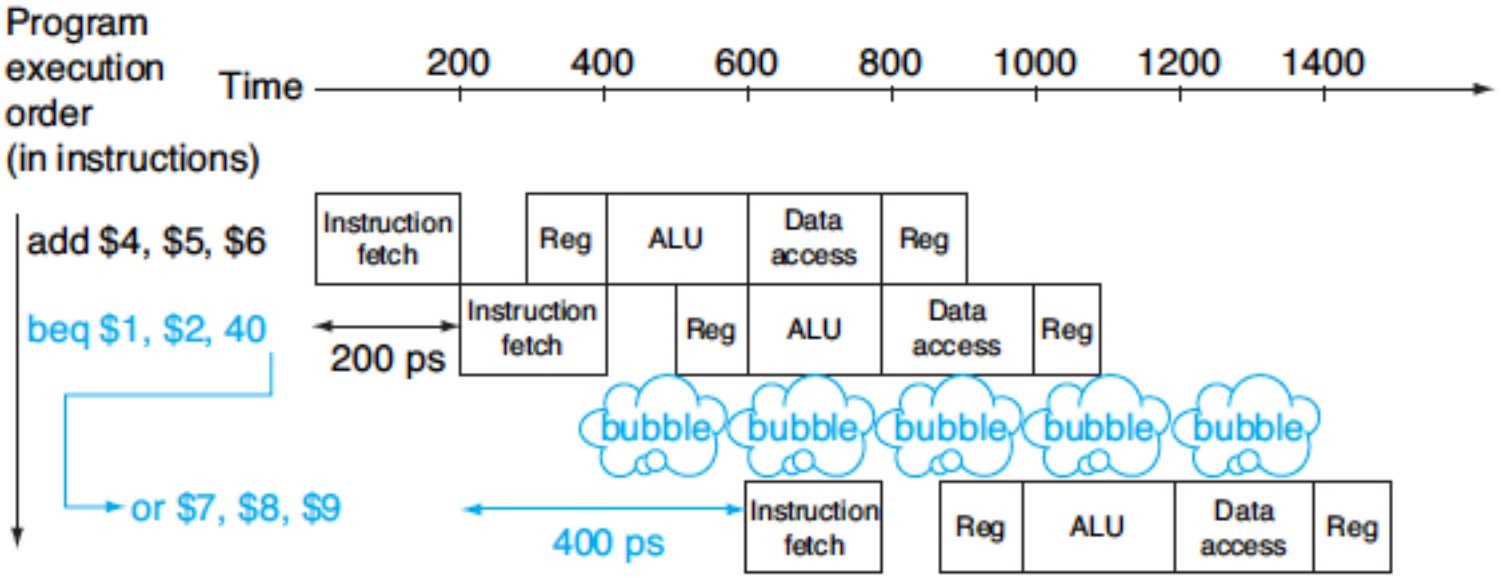


FIGURE 4.31 Pipeline showing stalling on every conditional branch as solution to control hazards. This example assumes the conditional branch is taken, and the instruction at the destination of the branch is the OR instruction. There is a one-stage pipeline stall, or bubble, after the branch. In reality, the process of creating a stall is slightly more complicated, as we will see in Section 4.8. The effect on performance, however, is the same as would occur if a bubble were inserted.

- Branch prediction
  - A method of resolving a branch hazard that assumes a given outcome for the branch and proceeds from that assumption rather than waiting to ascertain the actual outcome


Program execution order (in instructions)
Time
200
400
600
800
1000
1200
1400
add $4, $5, $6
beq $1, $2, 40
lw $3, 300($0)
Instruction fetch
Reg
ALU
Data access
Reg
200 ps
200 ps

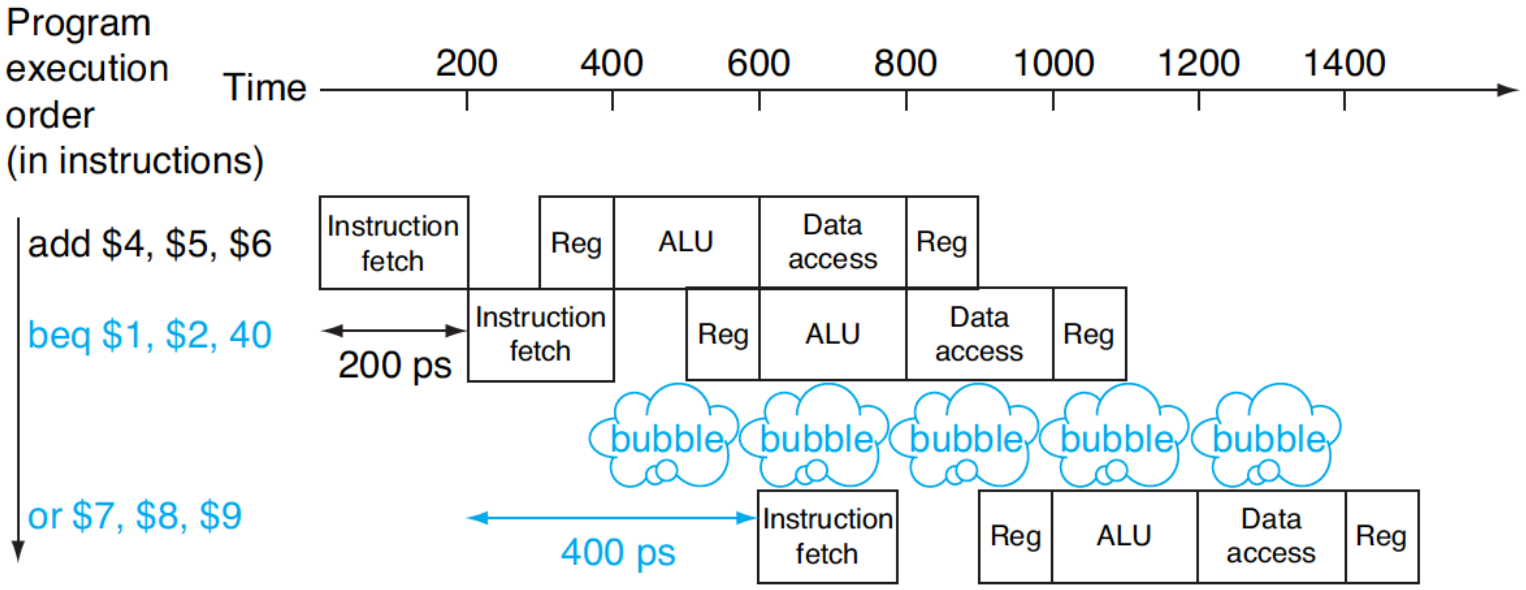

Program execution order (in instructions)
Time
200
400
600
800
1000
1200
1400
add $4, $5, $6
beq $1, $2, 40
or $7, $8, $9
Instruction fetch
Reg
ALU
Data access
Reg
200 ps
bubble
bubble
bubble
bubble
bubble
400 ps

Control hazard

- A more sophisticated version of branch predictor
  - Predict some branches as taken, while some as untaken
  - E.g., loops in a program
- Dynamic hardware predictor
  - Keeping a history for each branch for taken or untaken, and then using the recent past behavior to predict the future
  - When the guess is wrong, the pipeline control must ensure that the instruction following the wrongly guessed branch have no effect and must restart the pipeline from the proper branch address

Control hazard

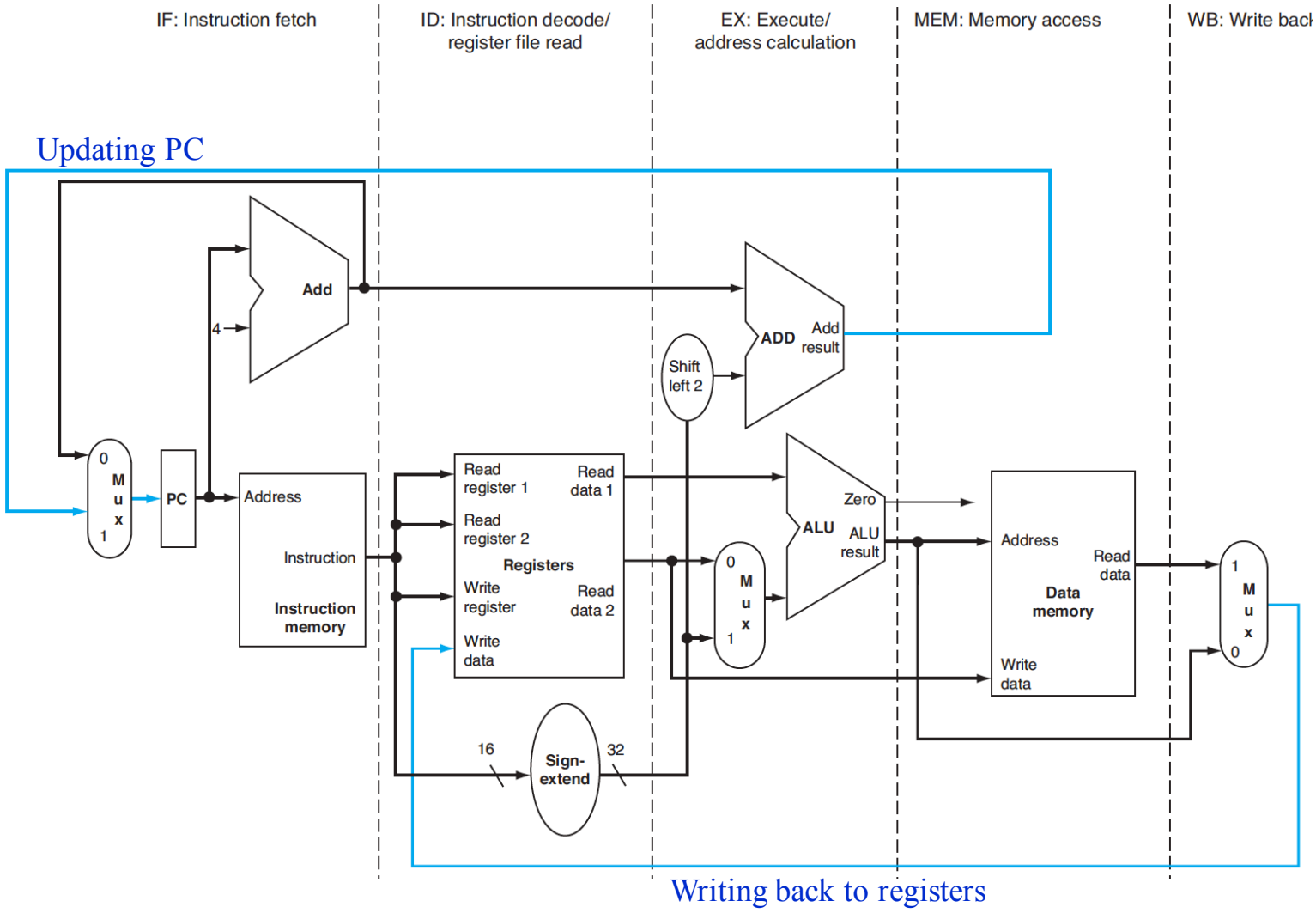
- Delayed branch
  - The delayed branch always executes the next sequential instruction, with the branch taking place after that one instruction delay
  - MIPS software will place an instruction immediately after the delayed branch instruction that is not affected by the branch, and a taken branch changes the address of the instruction that follows this safe instruction



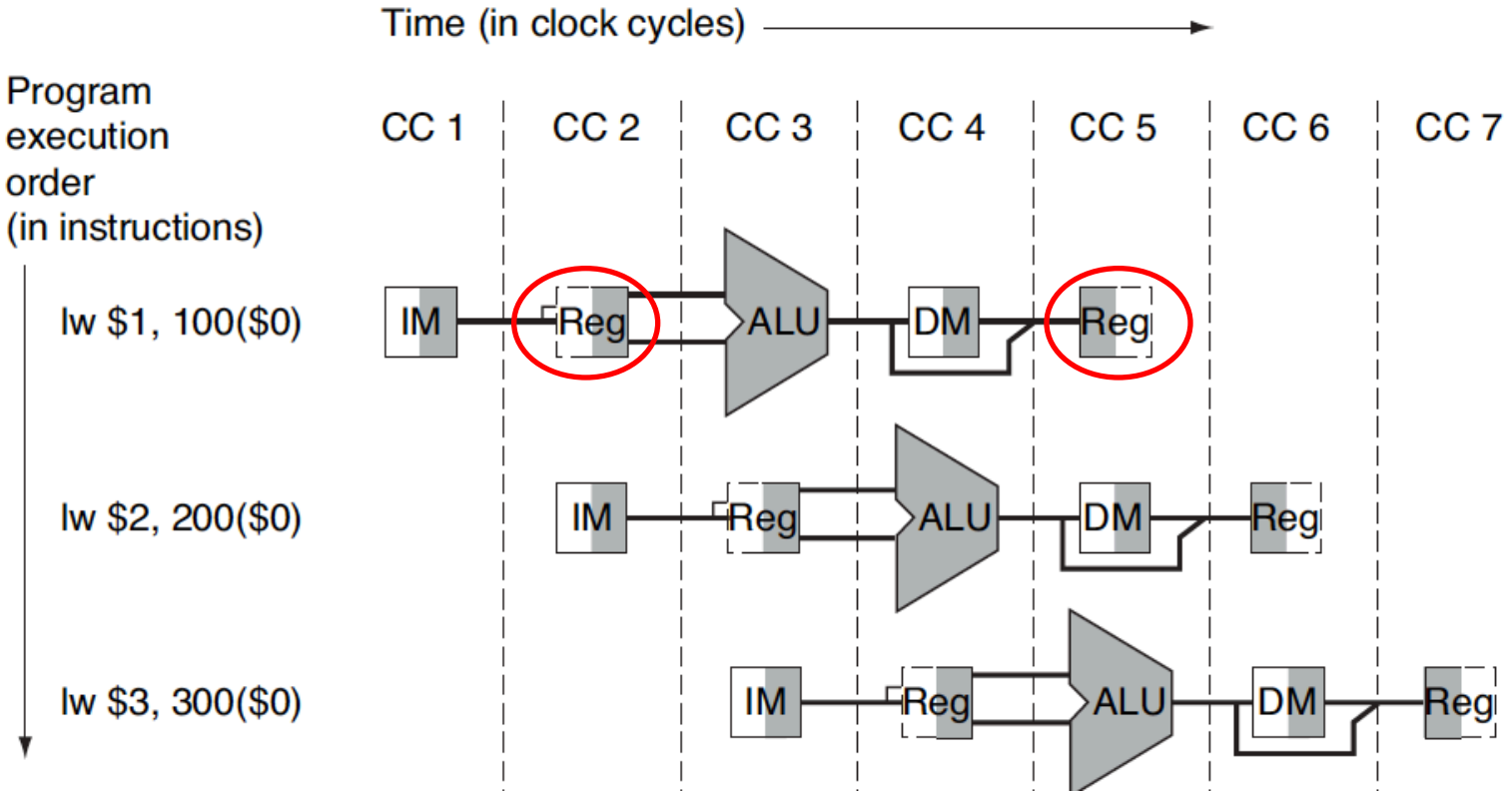
Big picture of pipeline

- Pipelining increases the number of simultaneously executing instructions and the rate at which instructions are started and completed.
- Pipelining does not reduce the time it takes to complete an individual instruction, so-called the latency

Pipelined datapath and control

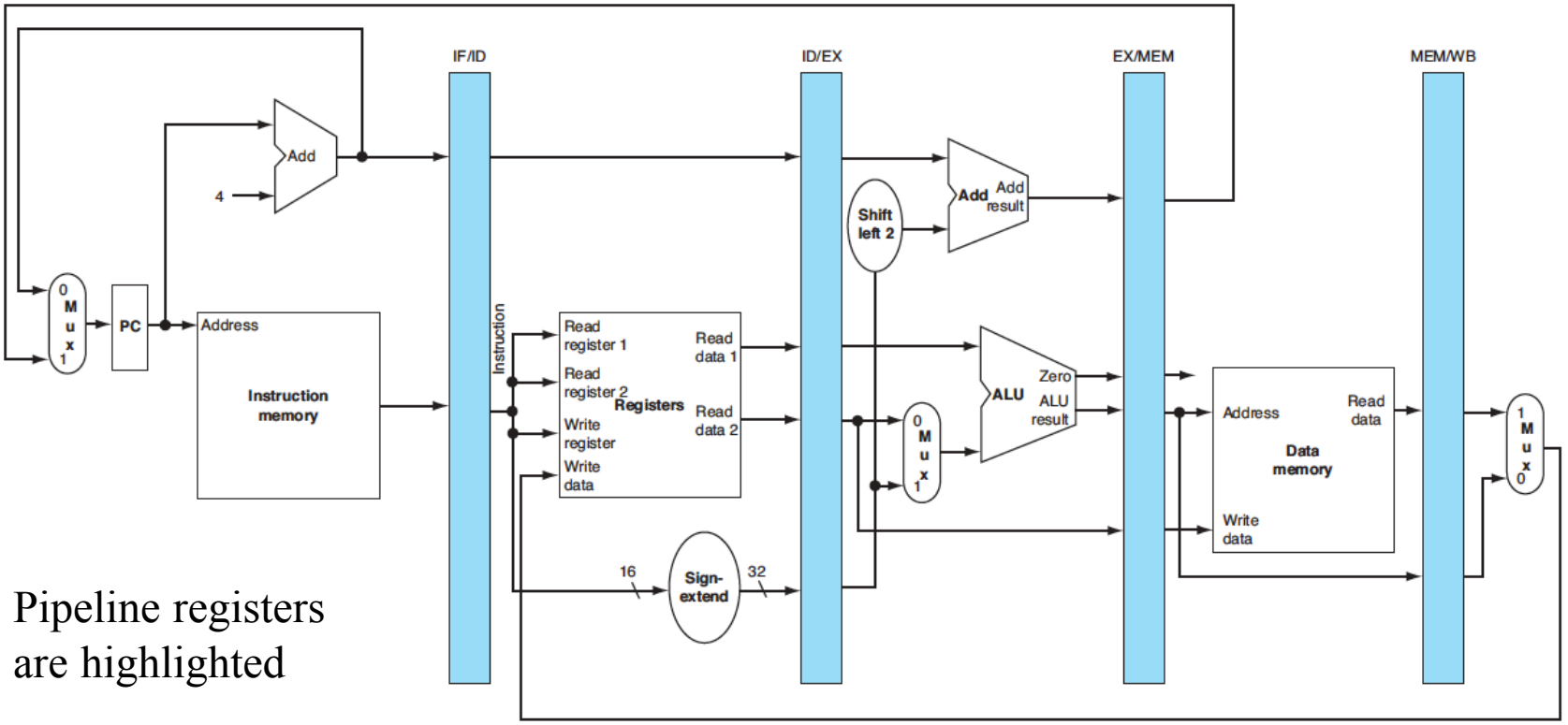


- One way to show what happens in pipelined execution is to pretend that each instruction has its own datapath, and then to place these datapaths on a timeline to show their relationship



- **IM**: The instruction memory and the PC in the instruction fetch stage
- **Reg**: The register file and sign extender in the instruction decode/ register file read stage, etc
- **DM**: Data memory access

- If we add some registers to hold data, portions of a single data path can be shared during instruction execution
- All instructions advance during each clock cycle from one pipeline register to the next
- No pipeline register at the end of the write-back stage
- PC can be thought of as a visible pipeline register



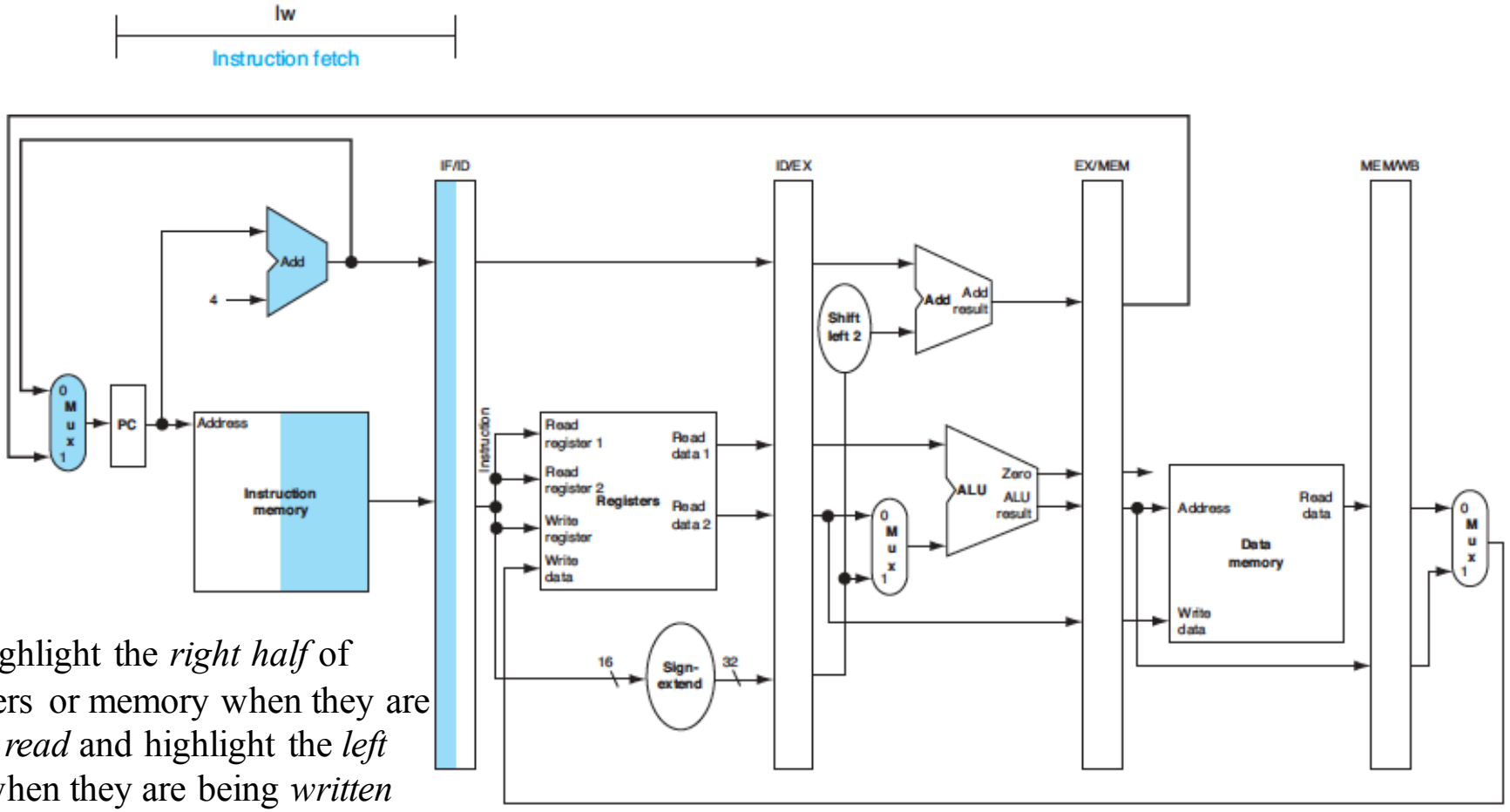
Pipeline registers are highlighted

64 bits 128 bits 97 bits 64bits

Example: **lw** instruction

Instruction fetch

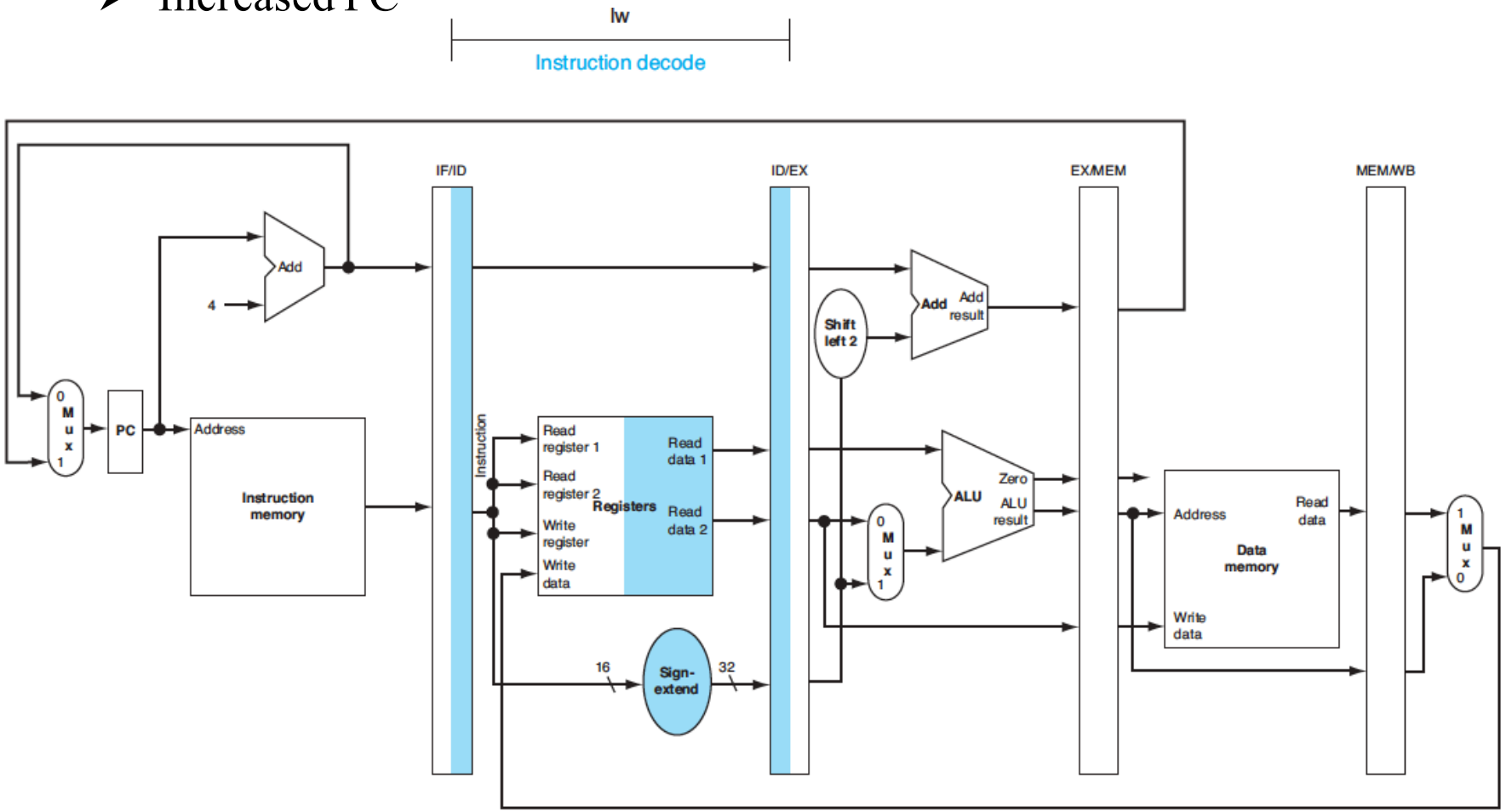
- Fetch the instruction addressed by PC, and save it in IF/ID pipeline register
- Increase PC by 4 and write it back to PC
- The increased address is also saved in the IF/ID pipeline register



We highlight the *right half* of registers or memory when they are being *read* and highlight the *left half* when they are being *written*

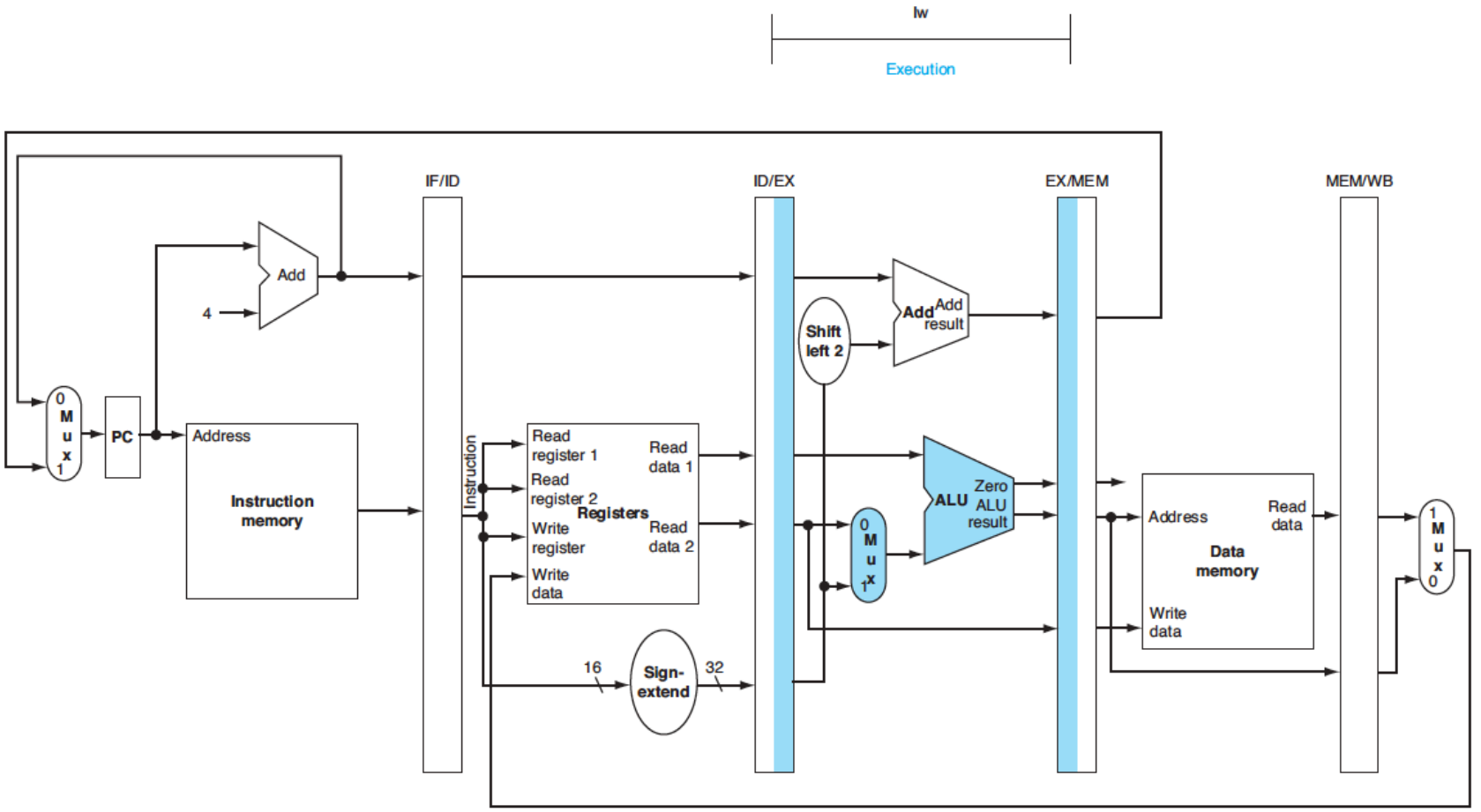
Instruction decode and register file read

- The following three values are stored in ID/EX pipeline register
  - 16-bit immediate field
  - Two register numbers
  - Increased PC



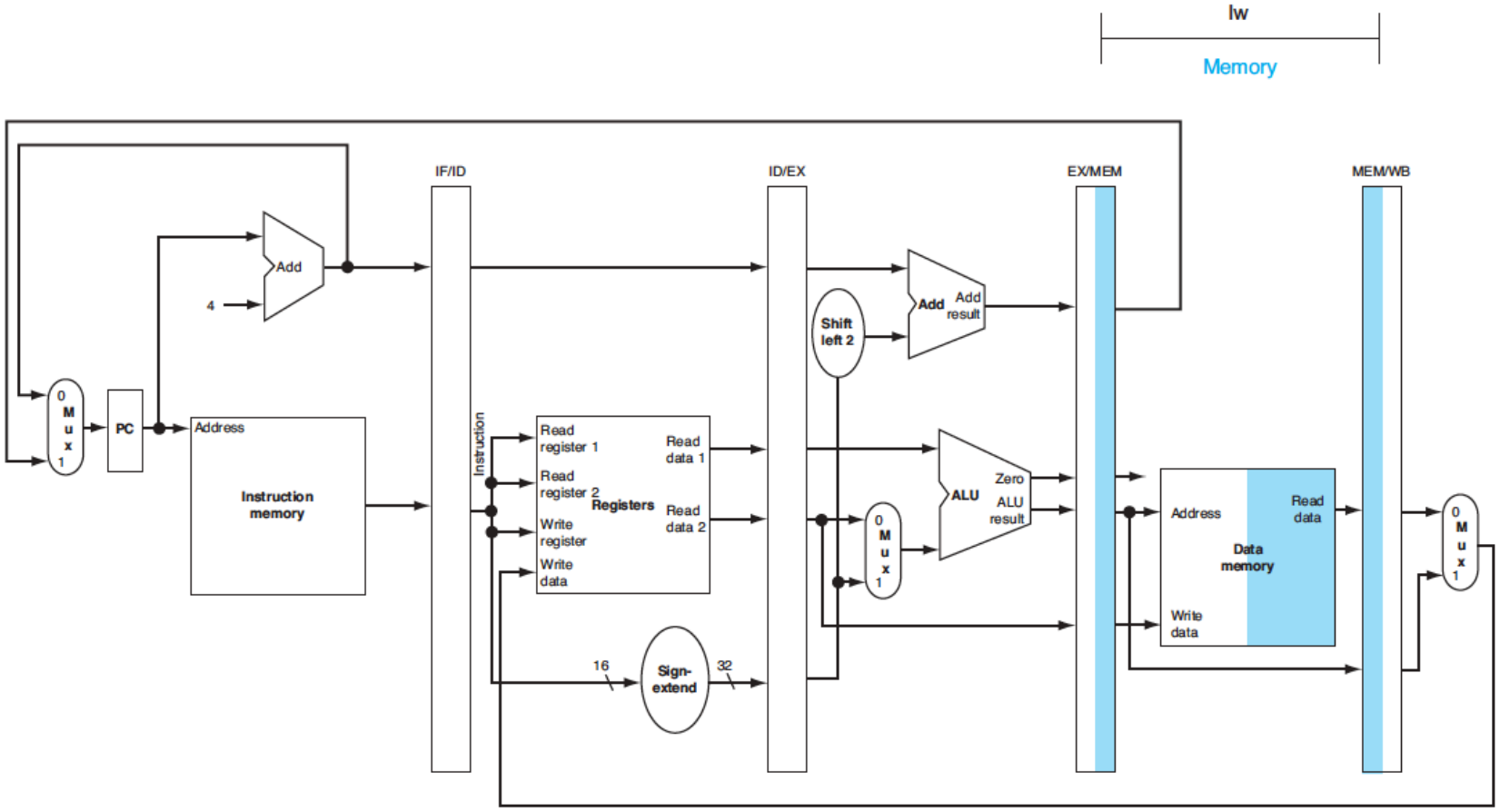
Execute or address calculation

- Reads contents of register 1
- Sign-extend the immediate
- Add the above two values in ALU
- Save the sum in EX/MEM pipeline register



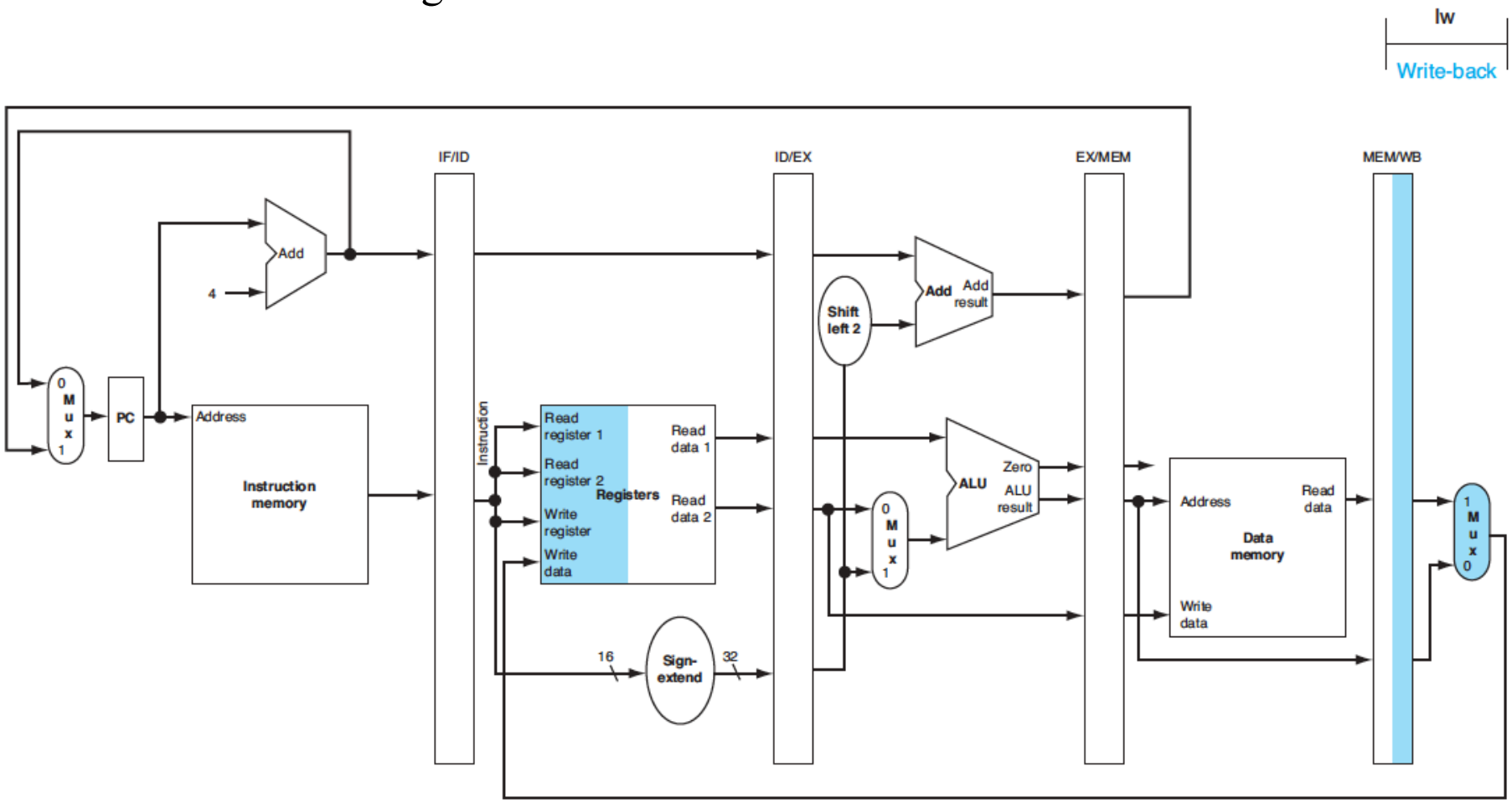
Memory access

- Read the data memory using the address from the EX/MEM register
- Load the data into MEM/WB pipeline register



Write-back

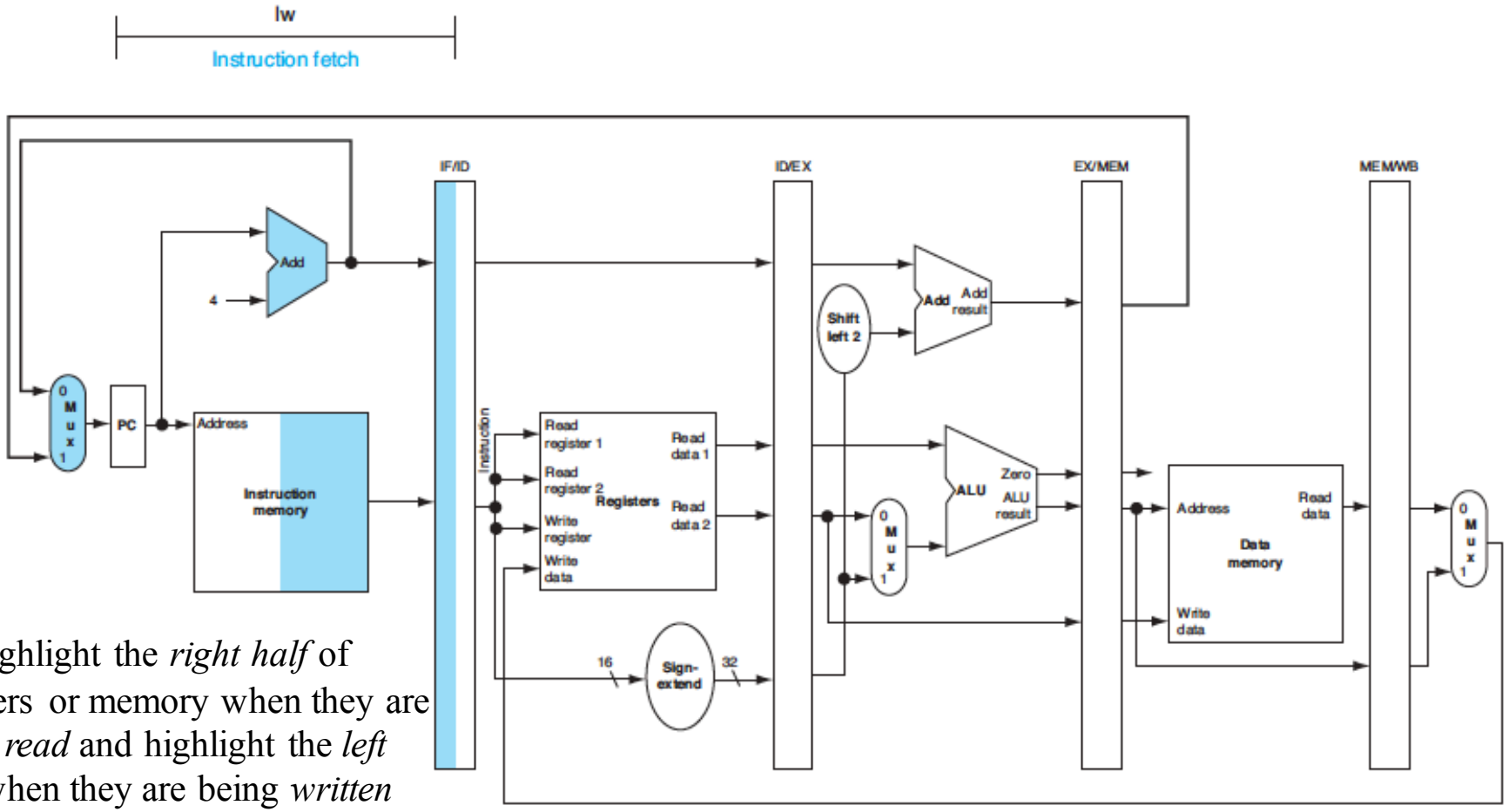
- Read the data from MEM/WB pipeline register
- Write it into the register file



Example: sw instruction

Instruction fetch

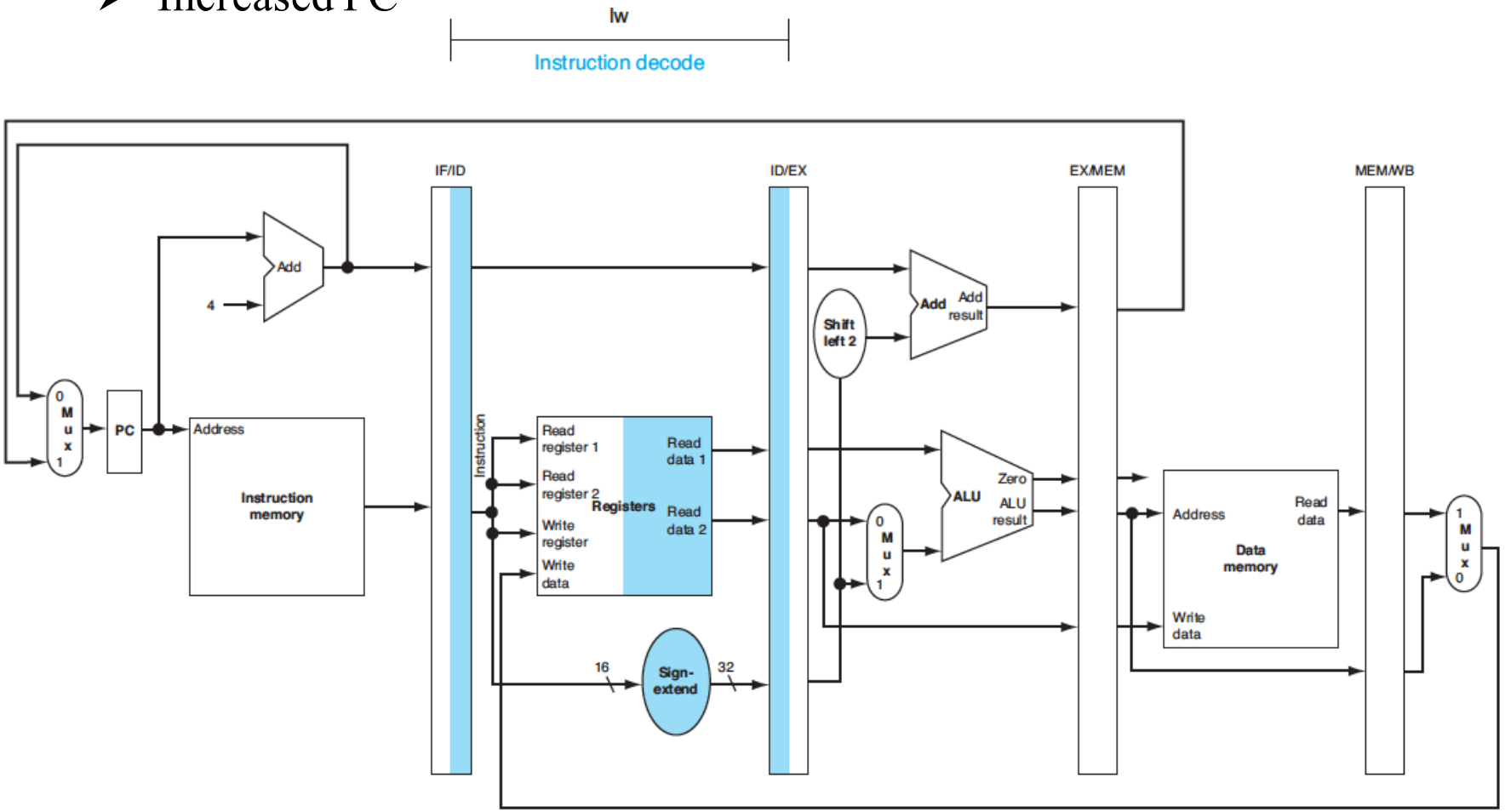
- Fetch the instruction addressed by PC, and save it in IF/ID pipeline register
- Increase PC by 4 and write it back to PC
- The increased address is also saved in the IF/ID pipeline register



We highlight the *right half* of registers or memory when they are being *read* and highlight the *left half* when they are being *written*

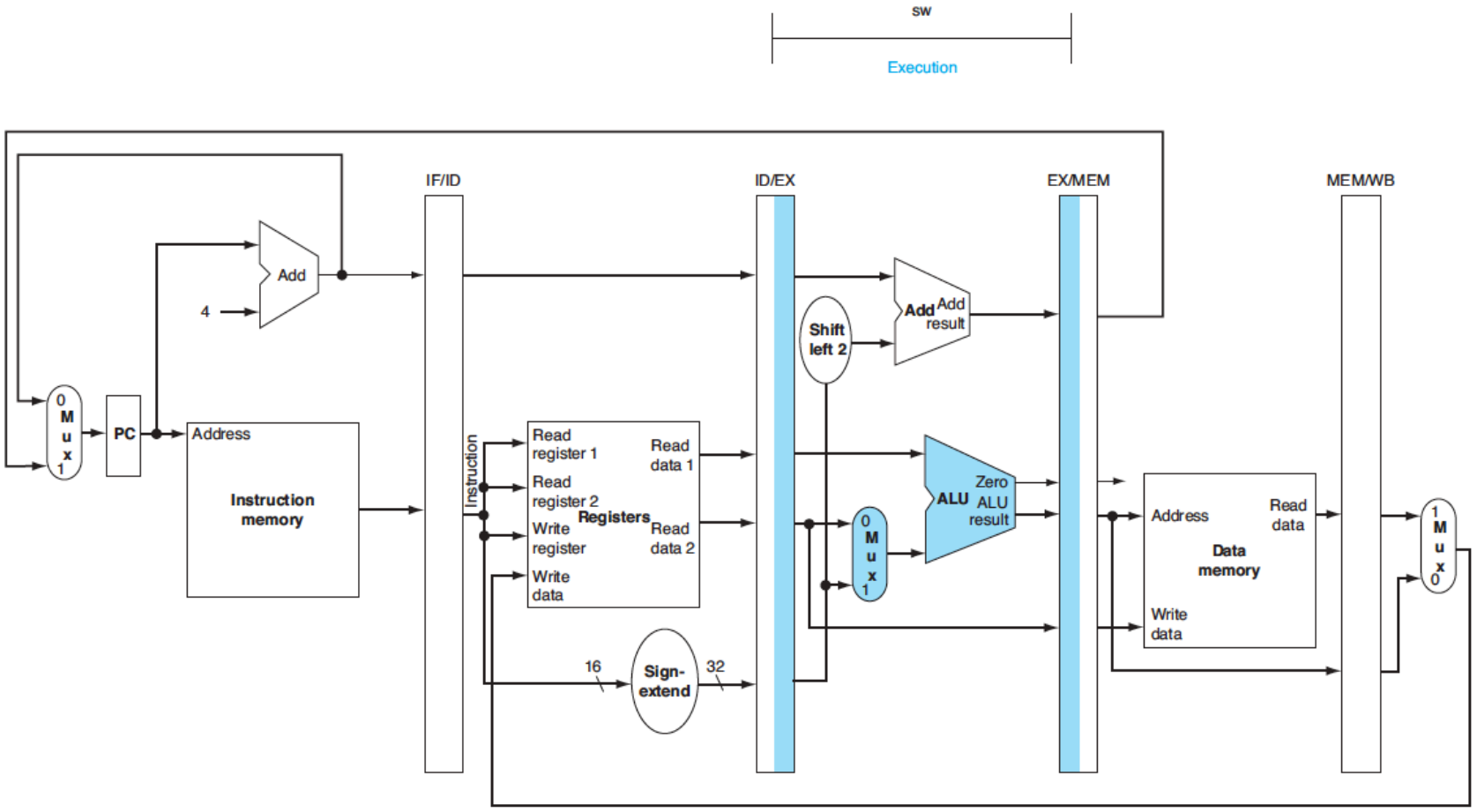
Instruction decode and register file read

- The following three values are stored in ID/EX pipeline register
  - 16-bit immediate field
  - Two register numbers
  - Increased PC



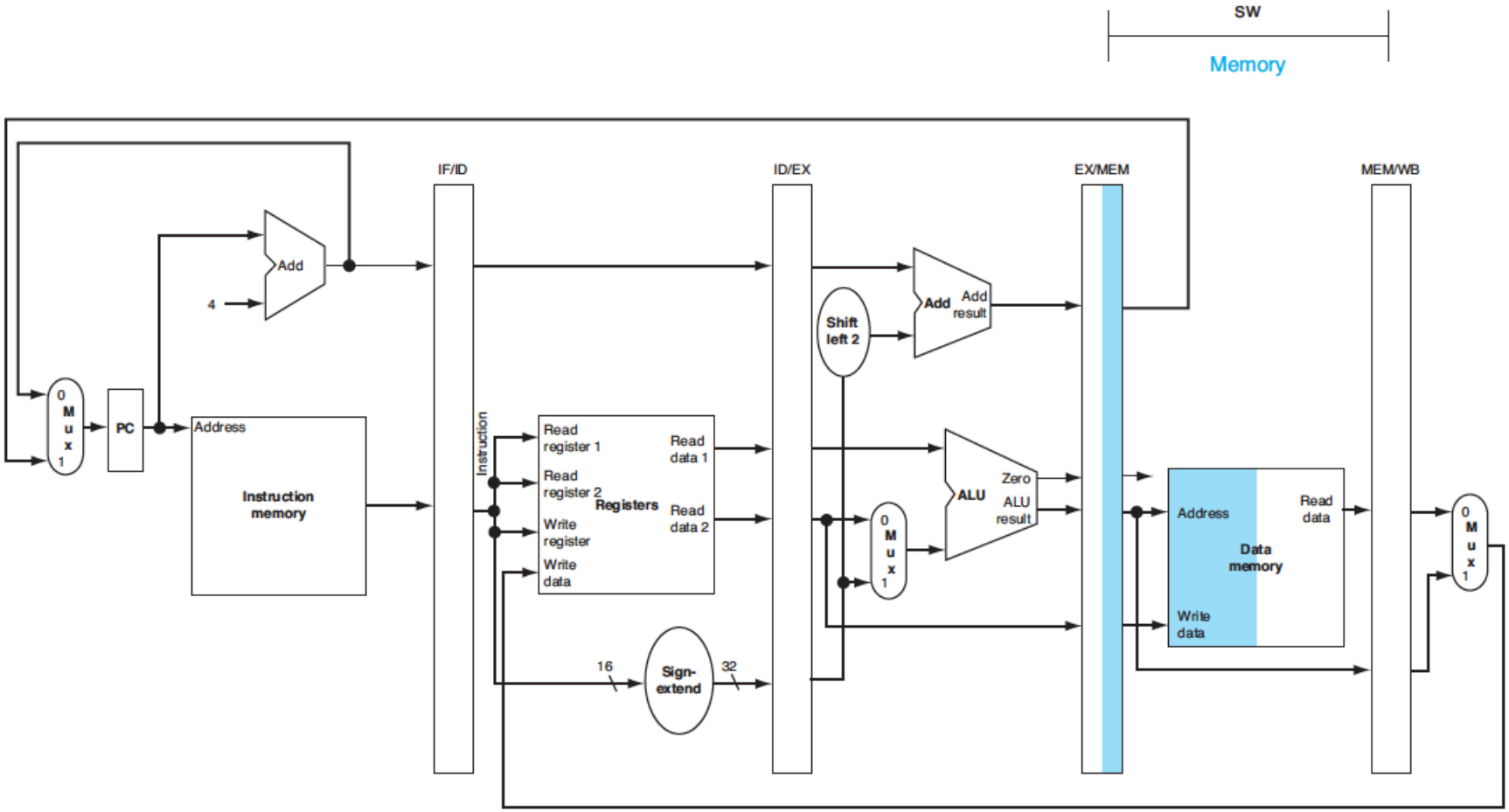
Execute or address calculation

- Reads contents of register 2
- Sign-extend the immediate
- Add the above two values in ALU
- Save the sum in EX/MEM pipeline register



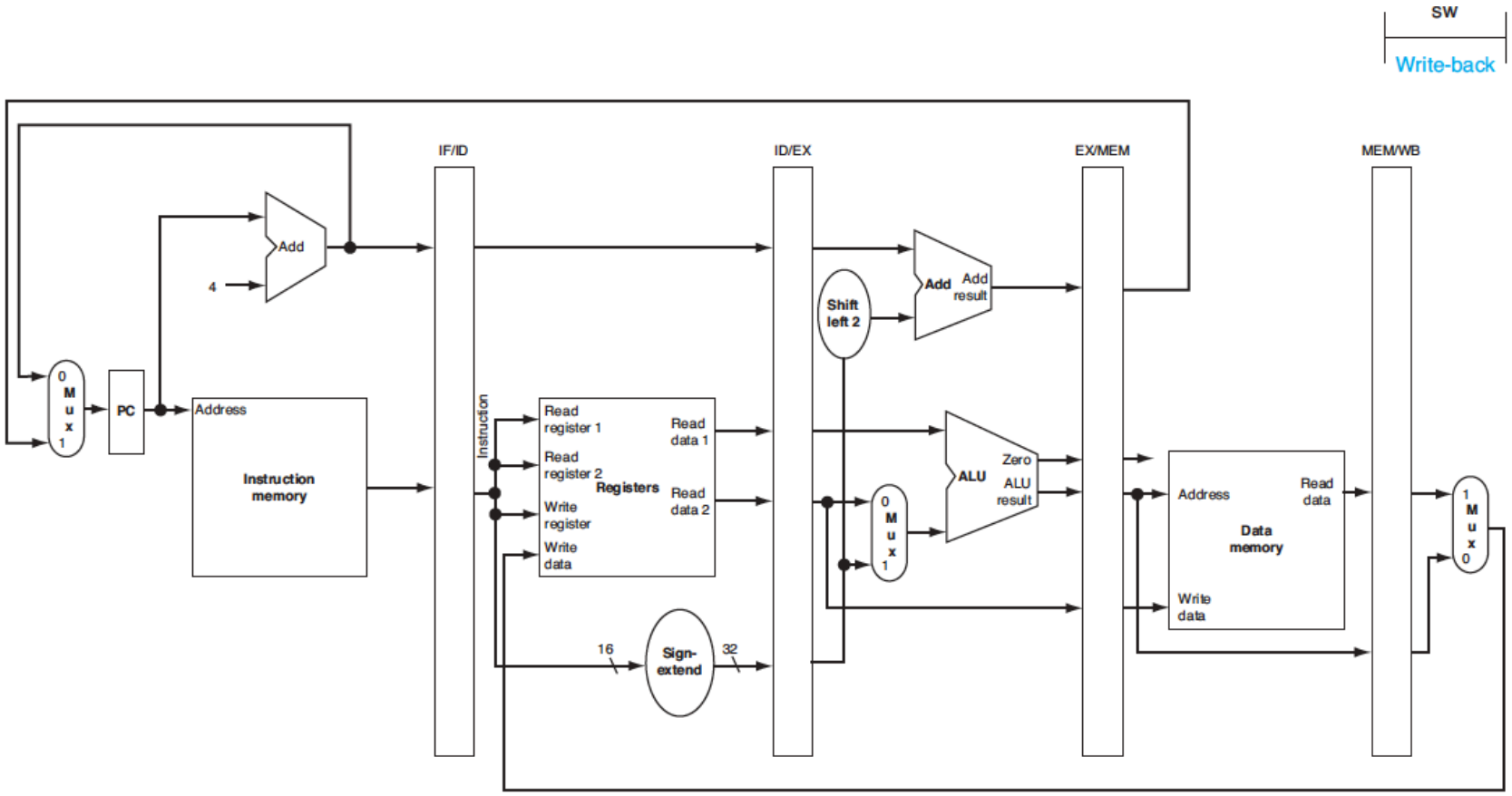
Memory access

- Write the data to the memory according to the address calculated earlier



Write-back

- Do nothing

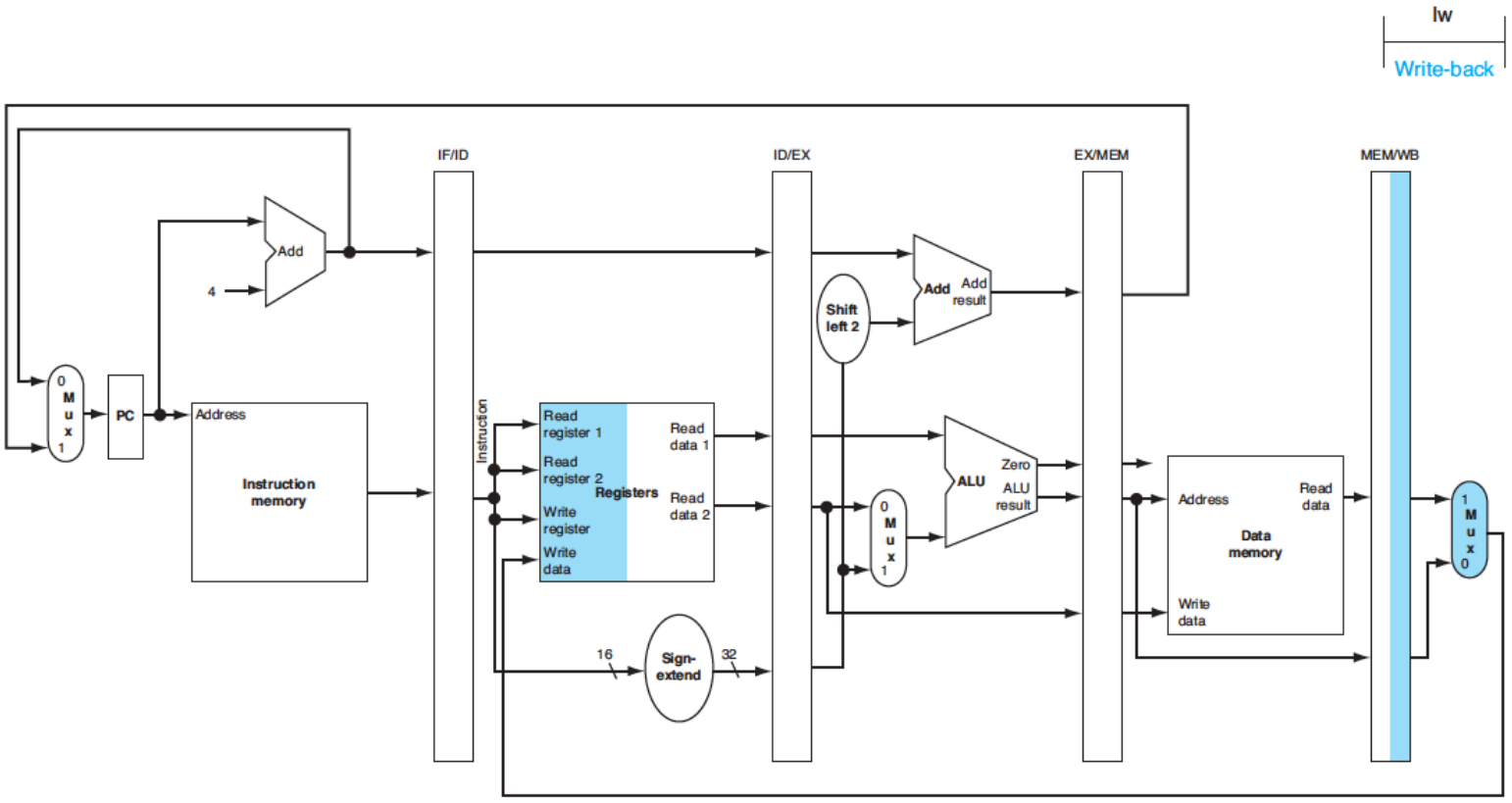


What we learn?

- The information from one stage to another should be placed in the pipeline registers; otherwise, the information would be lost when the next instruction enters the pipeline stage
- Each logical component of the datapath should be used only within a single pipeline stage; otherwise, we would have a structural hazard

A bug ?

How can we find the register to which we write the data back?



A revised pipeline control

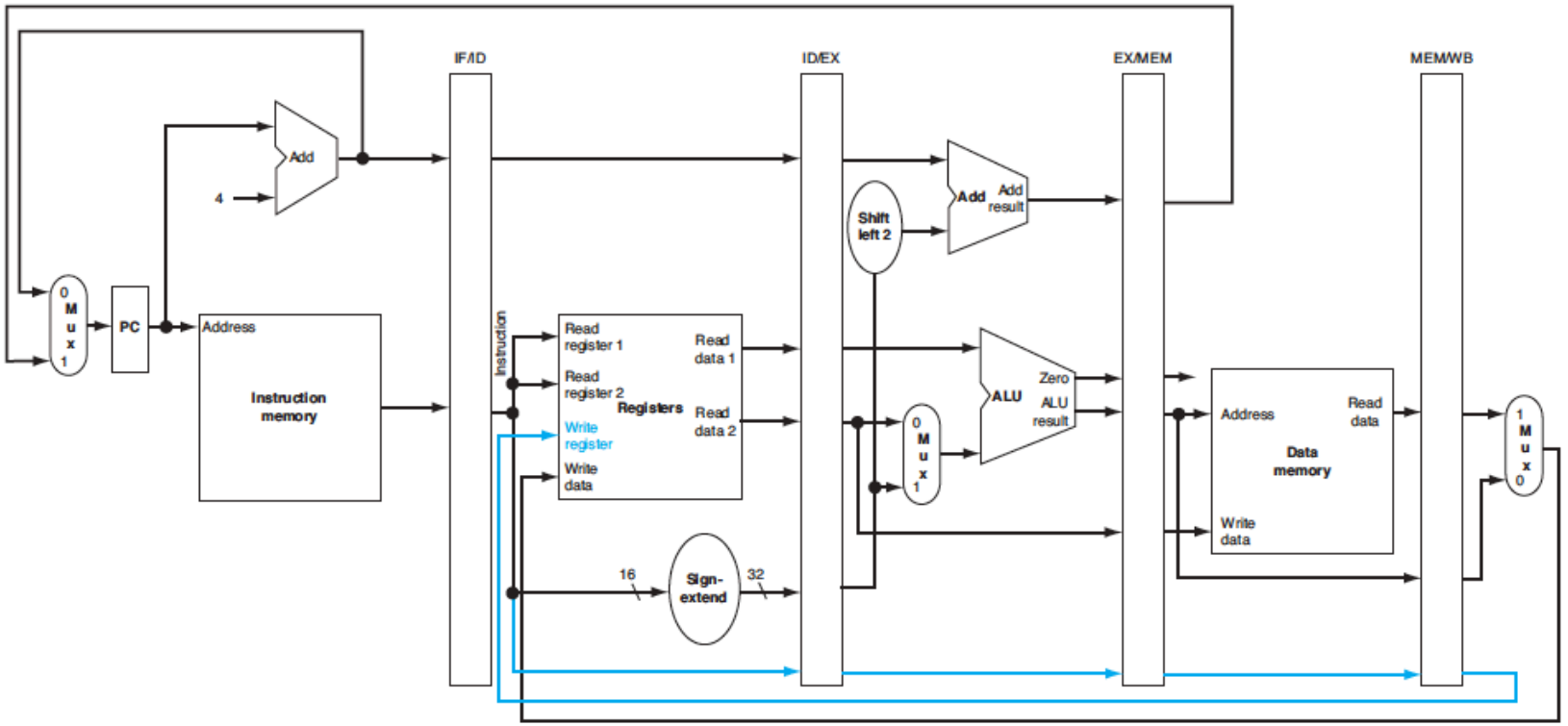
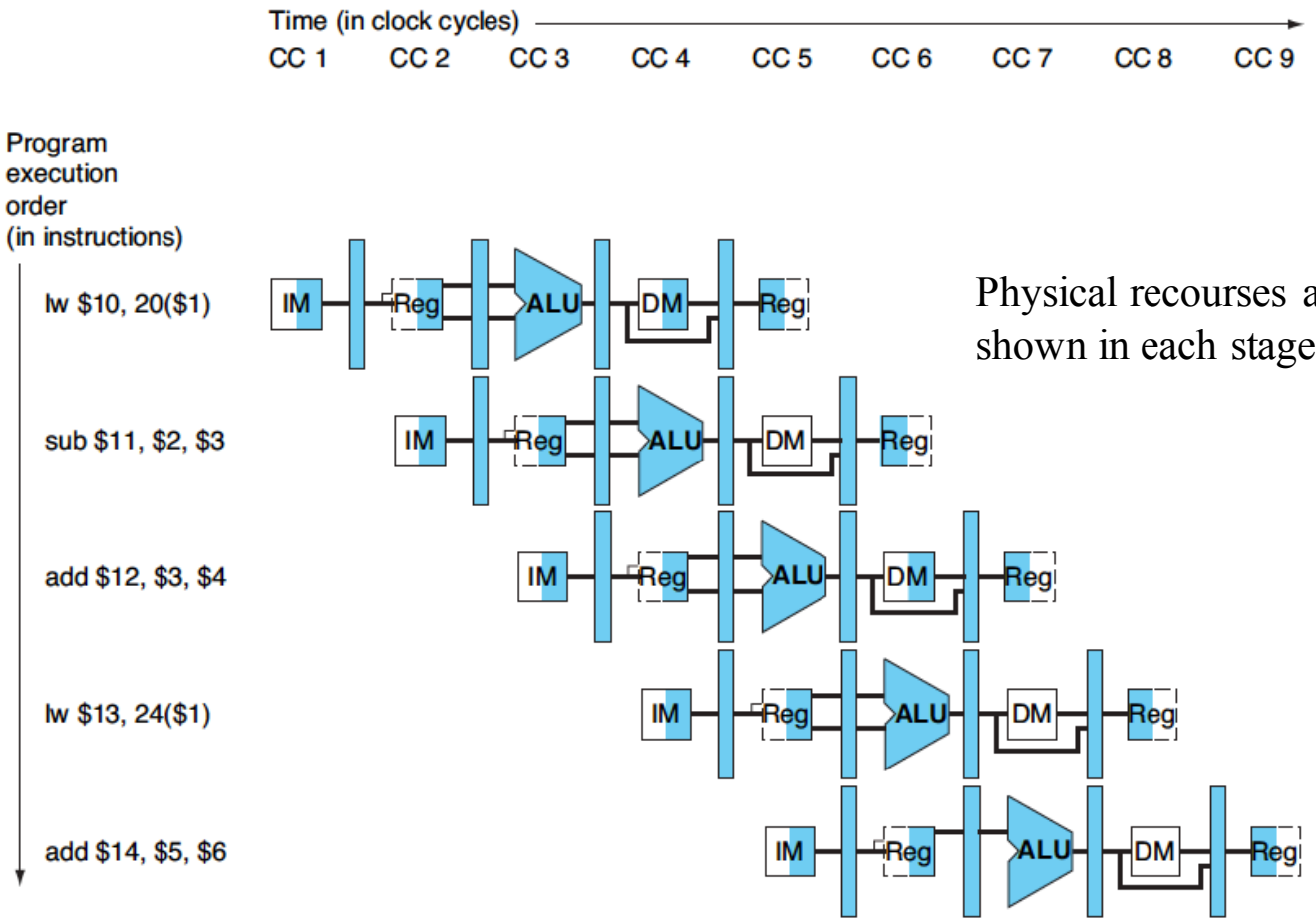


FIGURE 4.41 The corrected pipelined datapath to handle the load instruction properly. The write register number now comes from the MEM/WB pipeline register along with the data. The register number is passed from the ID pipe stage until it reaches the MEM/WB pipeline register, adding five more bits to the last three pipeline registers. This new path is shown in color.

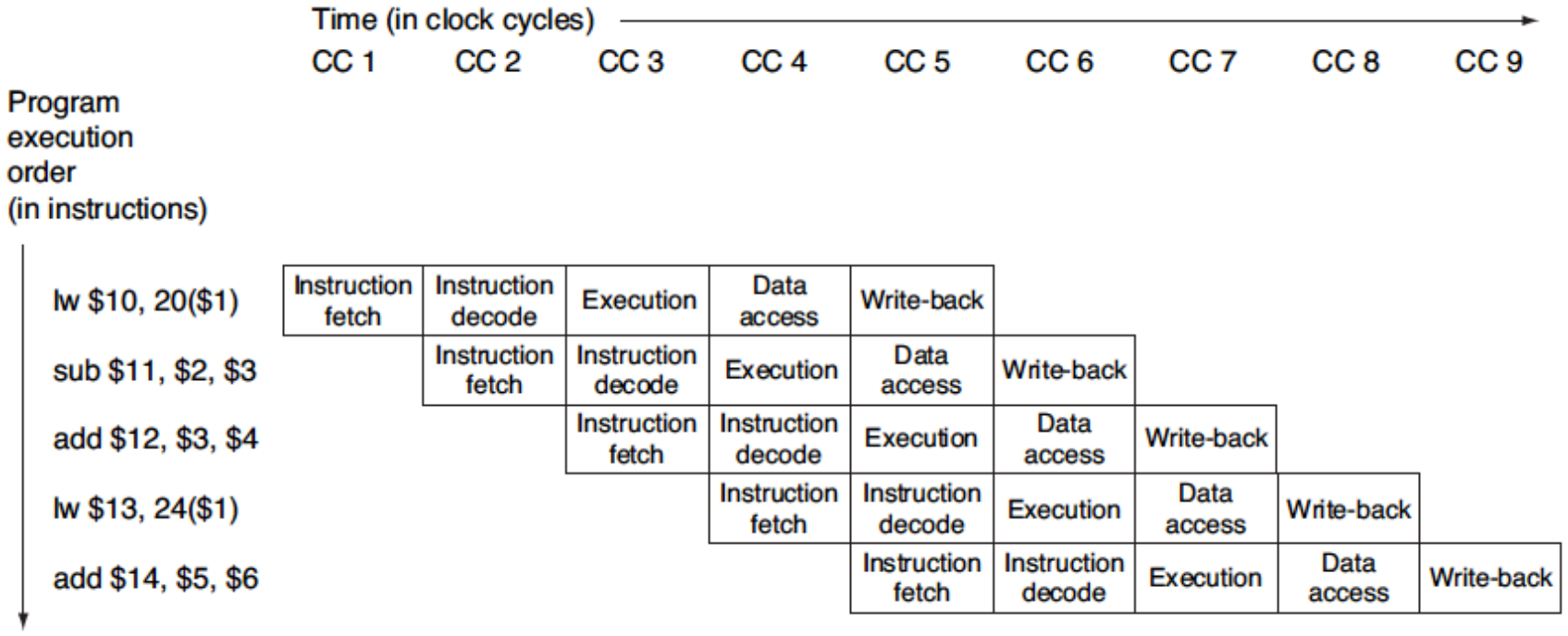
Graphically representing pipelines

Multiple-clock-cycle pipeline diagram



Graphically representing pipelines

A more traditional version of multiple-clock-cycle pipeline diagram



Graphically representing pipelines

- Single-clock-cycle pipeline diagrams show the state of the entire data path during a single clock cycle

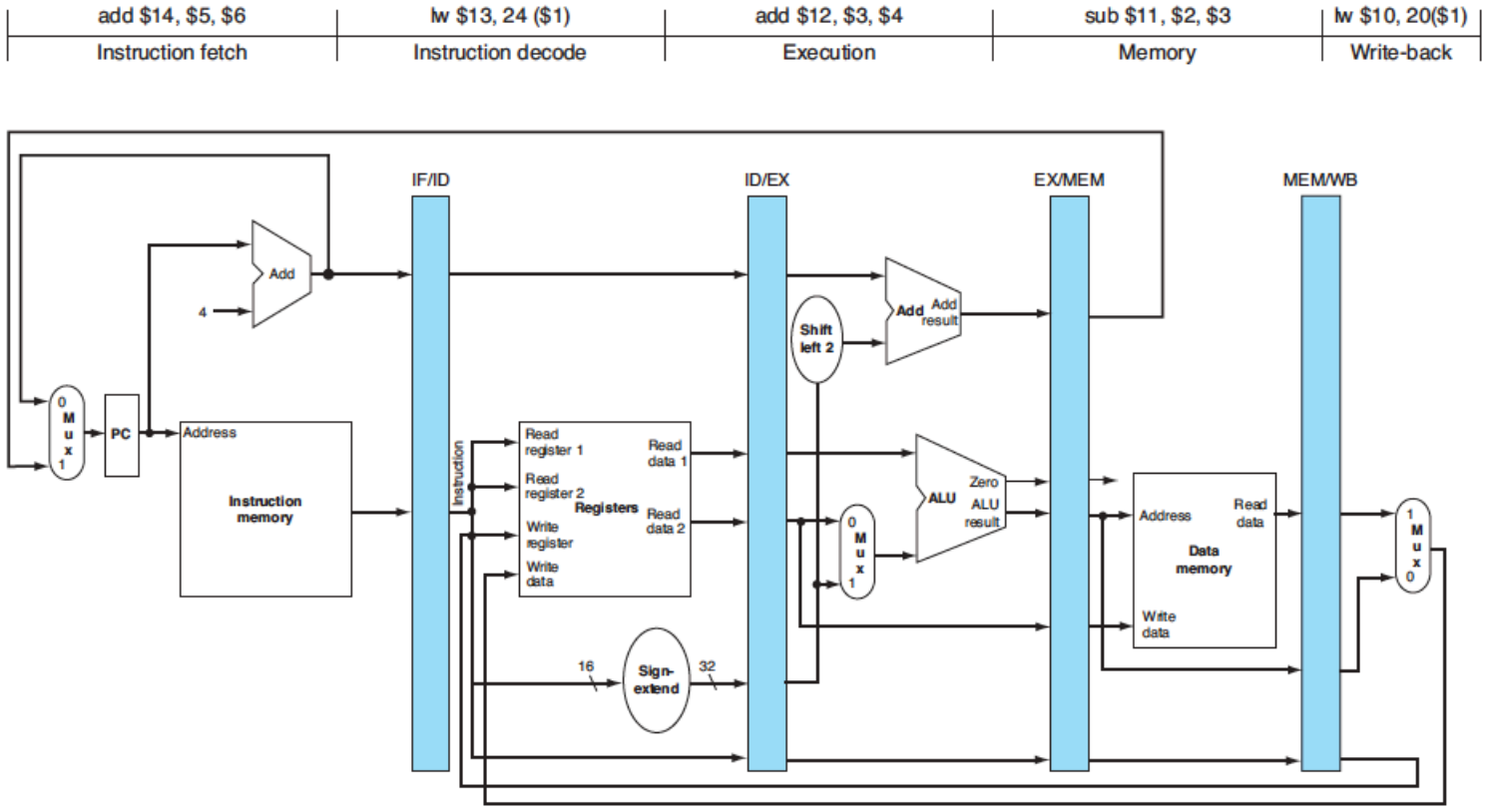
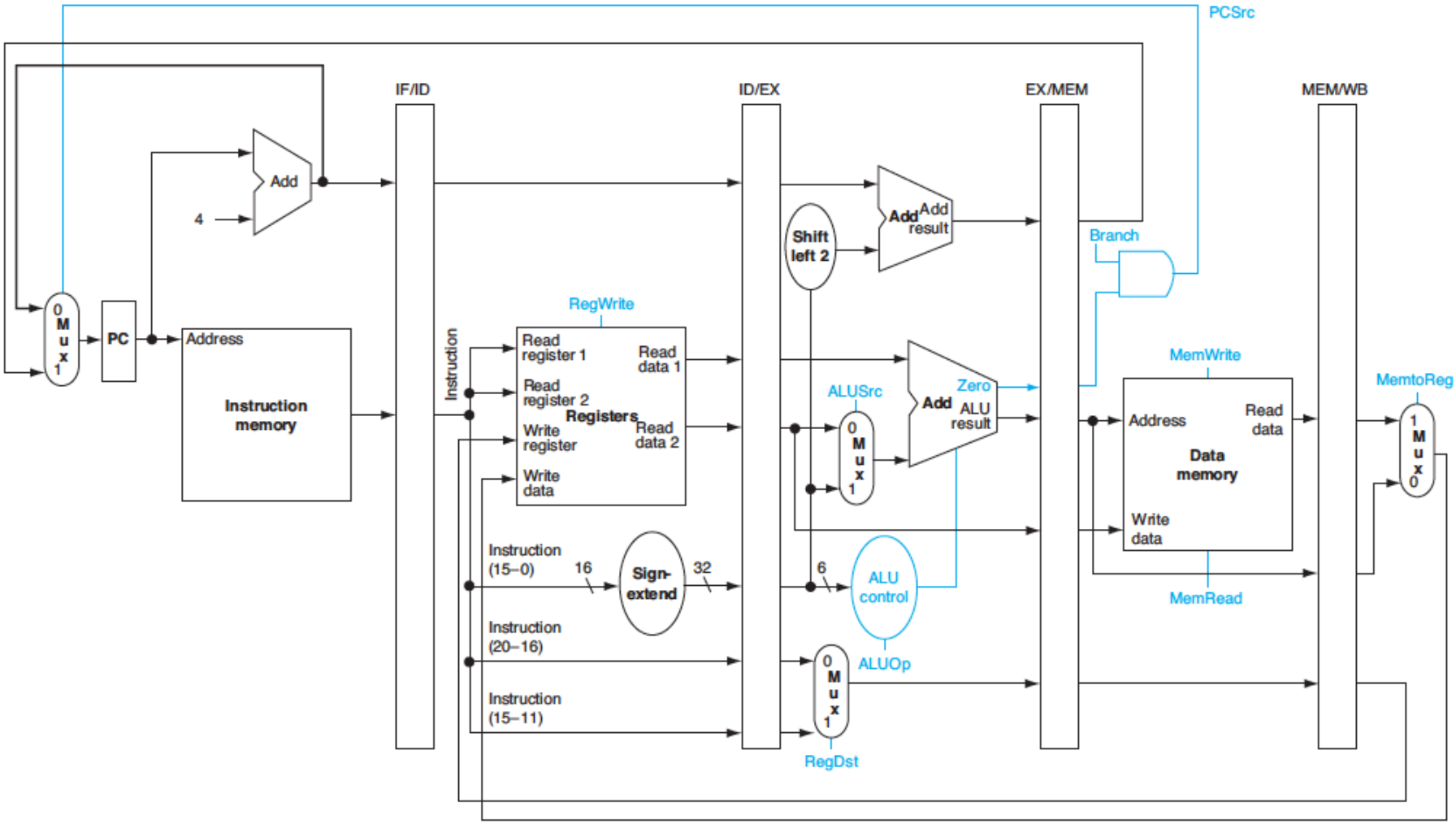


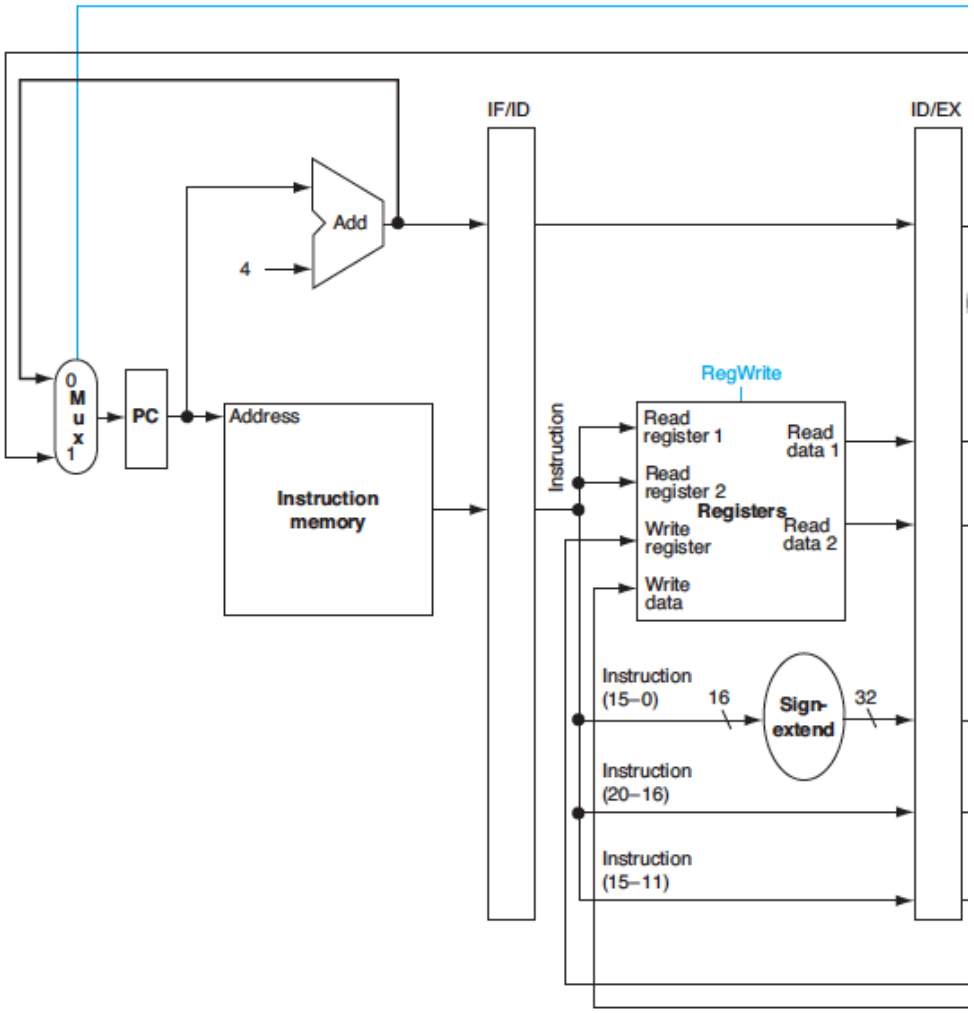
FIGURE 4.45 The single-clock-cycle diagram corresponding to clock cycle 5 of the pipeline in Figures 4.43 and 4.44. As you can see, a single-clock-cycle figure is a vertical slice through a multiple-clock-cycle diagram.

Pipelined control



Five groups of control signals

- Instruction fetch
  - Read instruction memory and write the PC
  - Always asserted
- Instruction decode/register file read
  - No optional control line



Five groups of control signals

- Execution/address calculation
  - RegDst, ALUOp, ALUSrc

Instruction opcode	ALUOp	Instruction operation	Function code	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch 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

Signal name	Effect when deasserted (0)	Effect when asserted (1)
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)
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

Five groups of control signals

- Memory access
  - Branch, MemRead, and MemWrite
  - PCsrc selects the next sequential address unless control asserts Branch and the ALU result was 0

Signal name	Effect when deasserted (0)	Effect when asserted (1)
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
PCSrc	The PC is replaced by the output of the address that computes the value of PC+4	The PC is replaced by the output of the adder that calculates the branch target

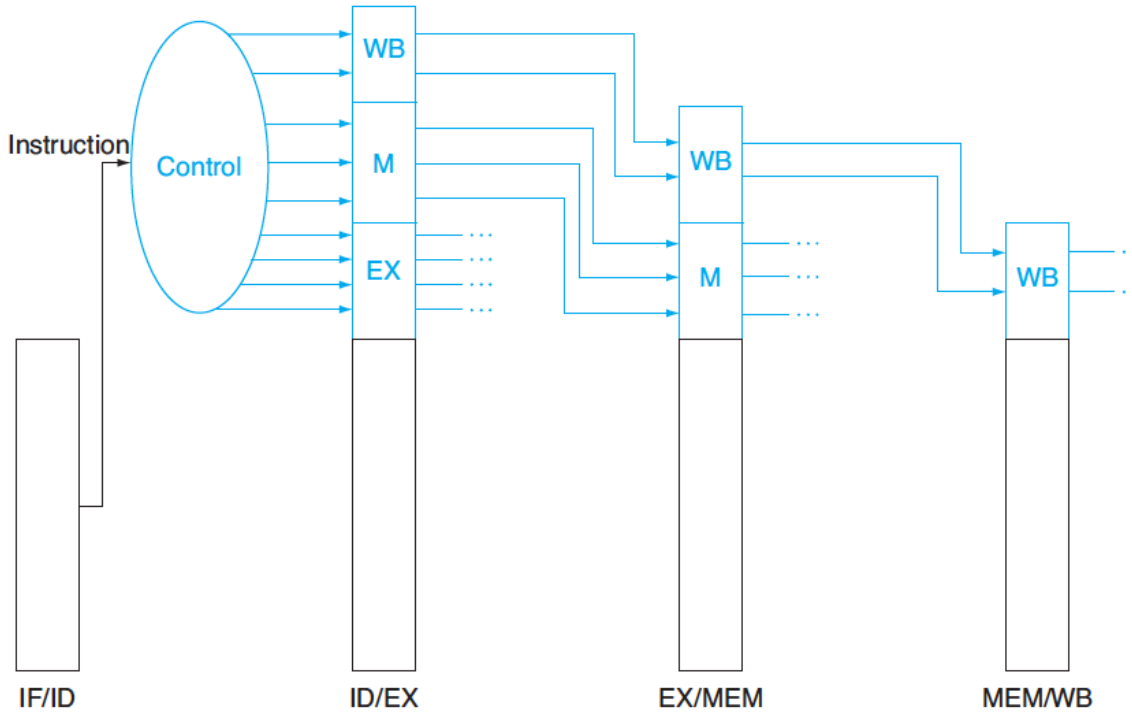
Five groups of control signals

- Write-back
  - MemtoReg, RegWrite

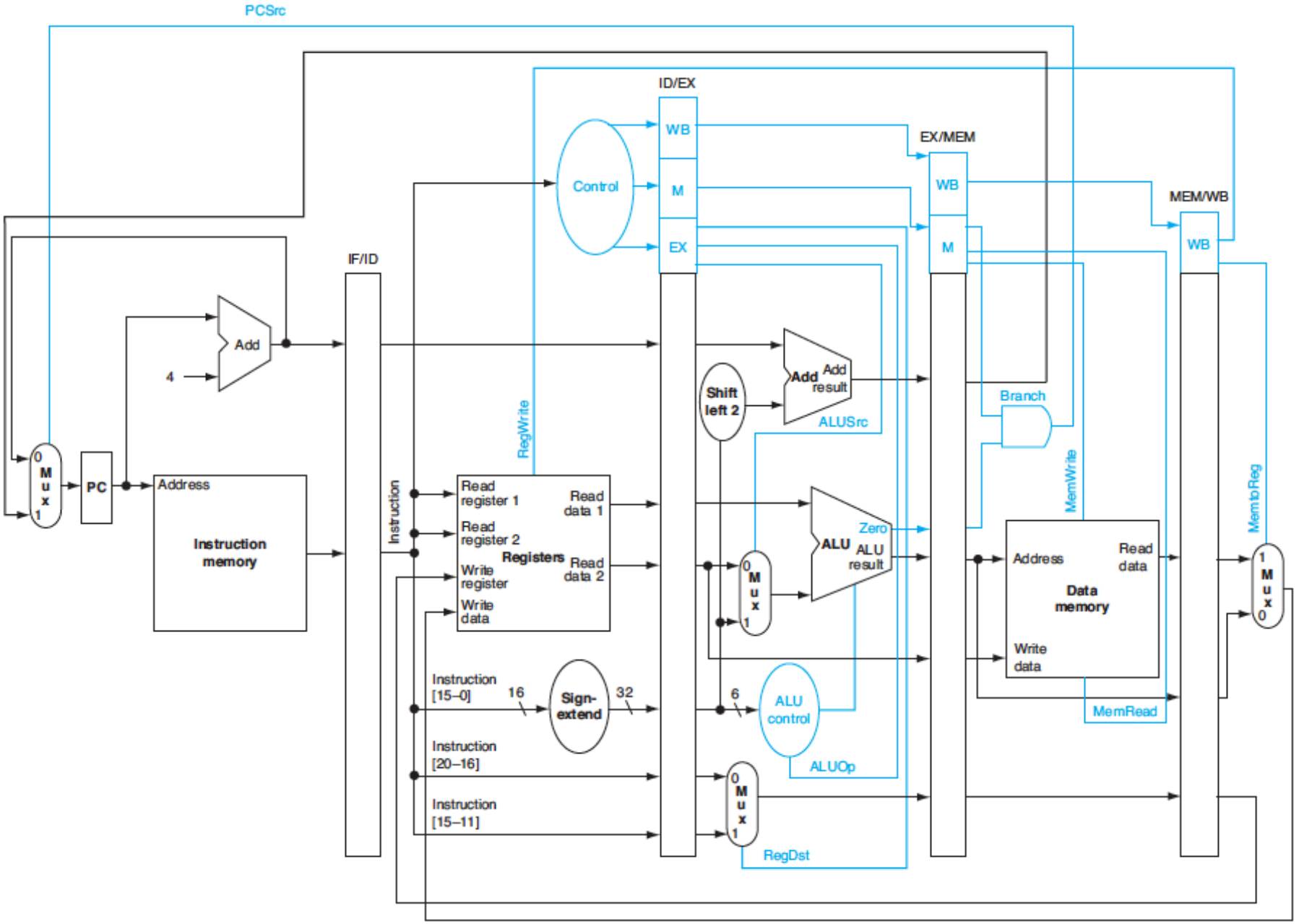
Signal name	Effect when deasserted (0)	Effect when asserted (1)
MemtoReg	The value fed to the register Write data input comes from the ALU	The value fed to the register Write data input comes from the data memory
RegWrite	None	The register on the Write register input is written with the value on the Write data input

Implementing the control

- Control implementation is to set the control signals
- Extending the pipeline register to store the control settings



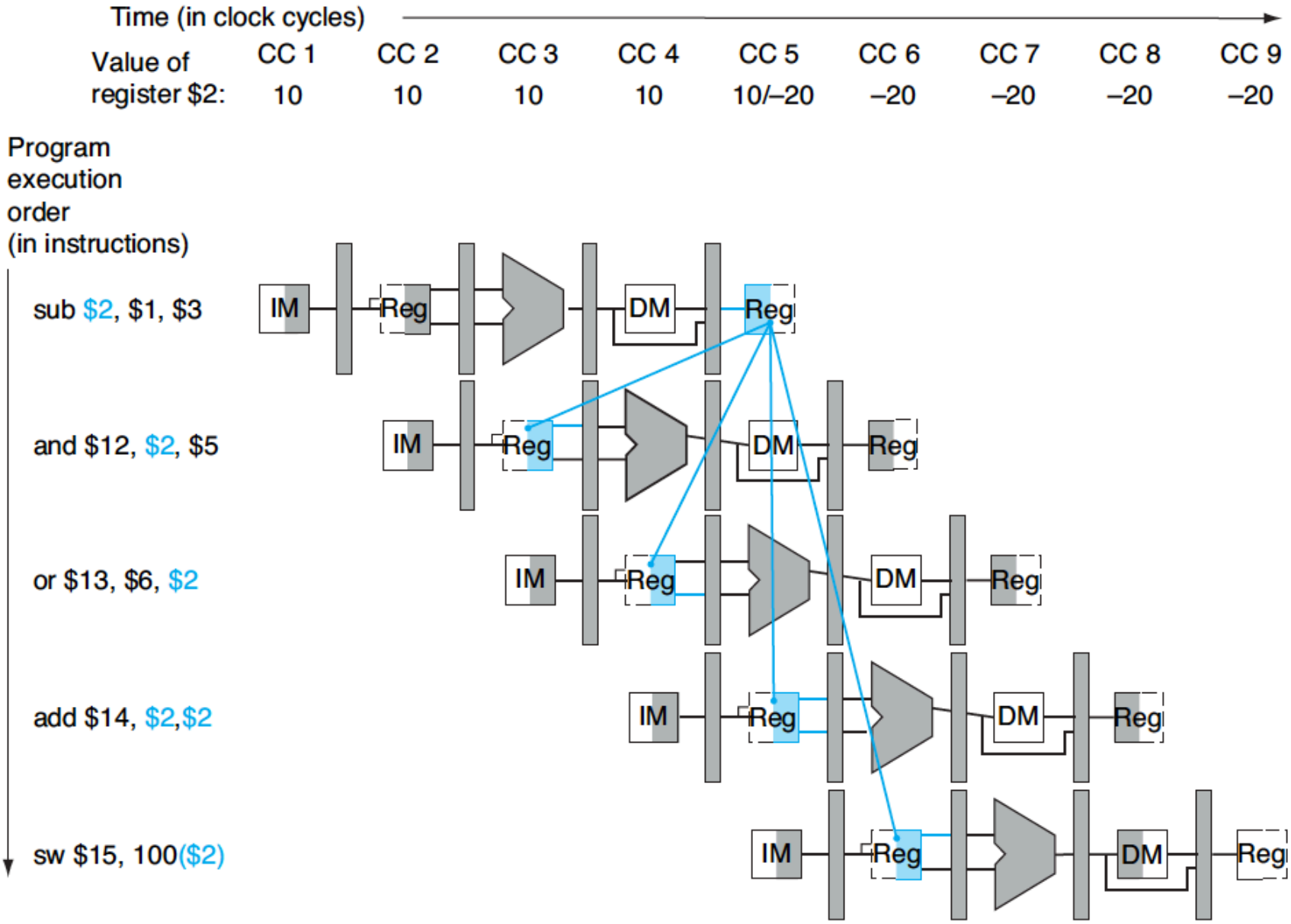
Note that four of the nine control lines are used in the EX phase, with the remaining five control lines passed on to the EX/MEM pipeline register extended to hold the control lines; three are used during the MEM stage, and the last two are passed to MEM/WB for use in the WB stage.


PCSrc
ID/EX
EX/MEM
MEM/WB
IF/ID
WB
M
EX
Control
Add
4
M u x
0
1
PC
Address
Instruction memory
Instruction
RegWrite
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Instruction [15–0]
16
Sign-extend
32
Instruction [20–16]
Instruction [15–11]
Shift left 2
Add
Add result
ALUSrc
ALU
Zero
ALU result
6
ALU control
ALUOp
RegDst
Branch
MemWrite
MemRead
MemtoReg
Address
Read data
Data memory
Write data

Data hazard: forwarding vs stalling

- An example

```
sub    $2, $1,$3      # Register $2 written by sub
and    $12,$2,$5      # 1st operand($2) depends on sub
or     $13,$6,$2      # 2nd operand($2) depends on sub
add    $14,$2,$2      # 1st($2) & 2nd($2) depend on sub
sw     $15,100($2)    # Base ($2) depends on sub
```


Time (in clock cycles)
Value of register $2:
CC 1 CC 2 CC 3 CC 4 CC 5 CC 6 CC 7 CC 8 CC 9
10 10 10 10 10/–20 –20 –20 –20 –20
Program execution order (in instructions)
sub $2, $1, $3
and $12, $2, $5
or $13, $6, $2
add $14, $2,$2
sw $15, 100($2)
IM
Reg
DM

A more precise notation of dependences

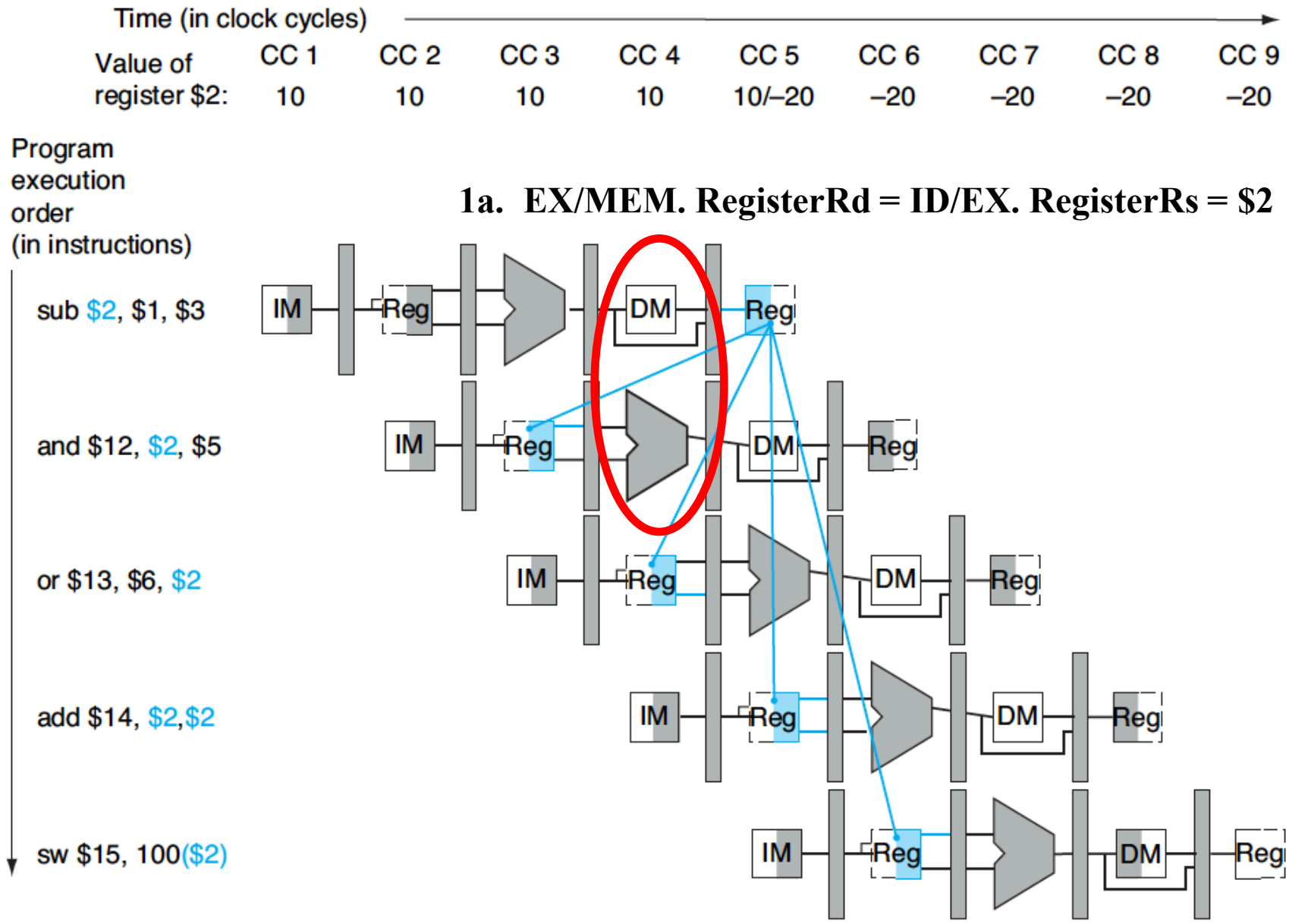
1a. EX/MEM.RegisterRd = ID/EX.RegisterRs

1b. EX/MEM.RegisterRd = ID/EX.RegisterRt

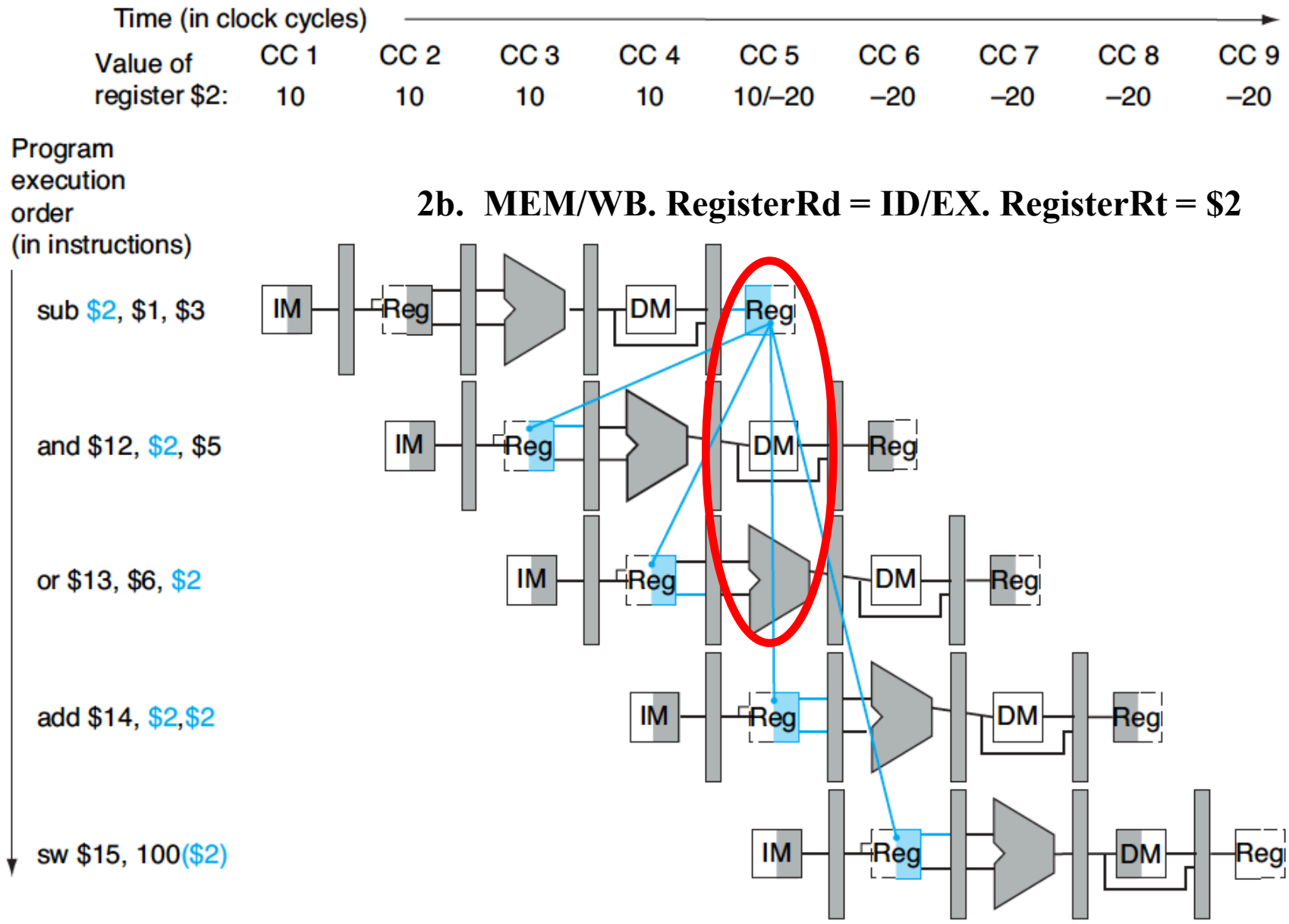
2a. MEM/WB.RegisterRd = ID/EX.RegisterRs

2b. MEM/WB.RegisterRd = ID/EX.RegisterRt

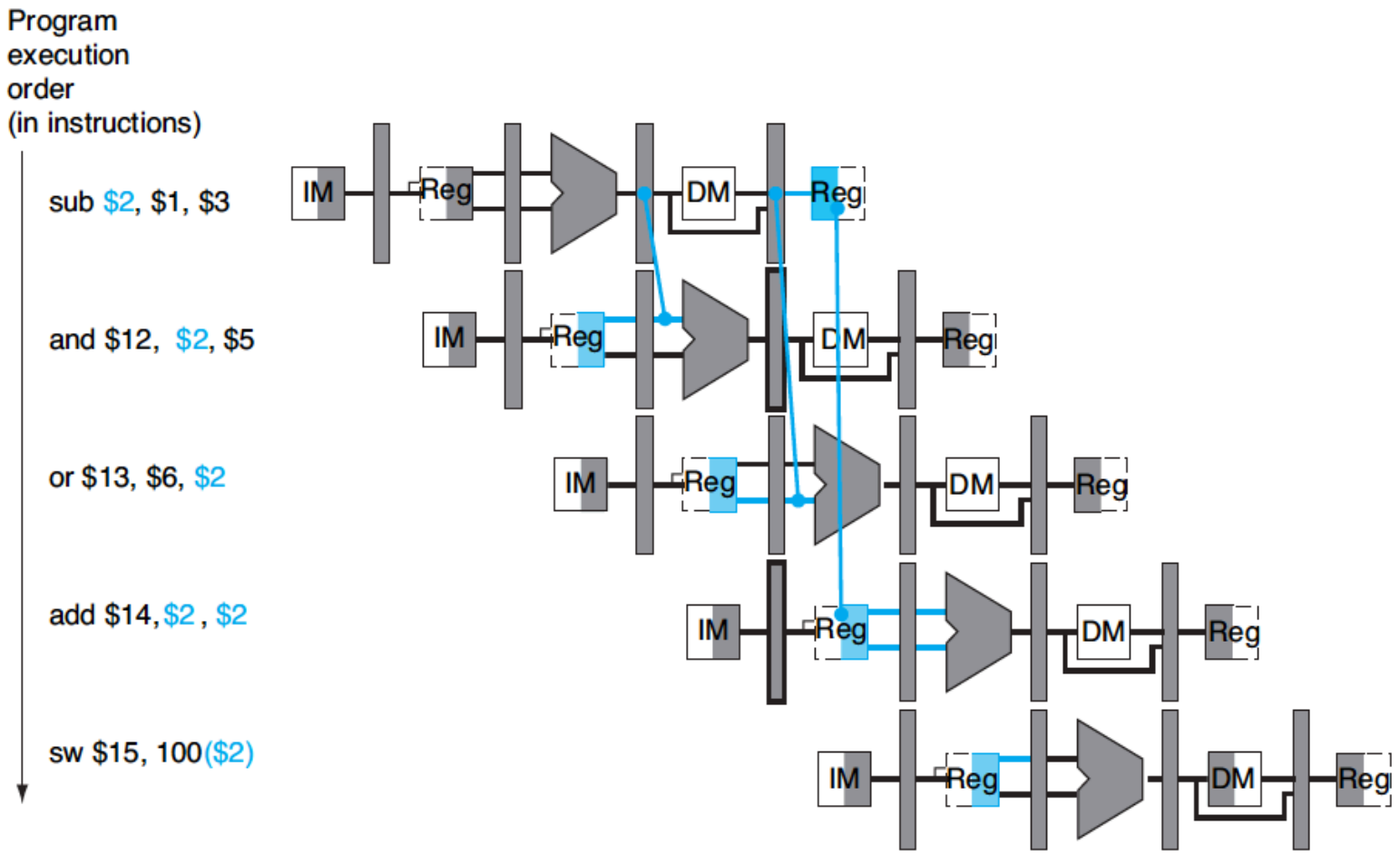
- *E.g., "ID/EX. RegisterRs" refers to the number of one register whose value is found in the pipeline register ID/EX*
- *The first part of the name is the name of the pipeline register*
- *The second part of the name is the name of the field in that register*



1a. EX/MEM. RegisterRd = ID/EX. RegisterRs = $2


Time (in clock cycles)
Value of register $2:
CC 1 10
CC 2 10
CC 3 10
CC 4 10
CC 5 10/–20
CC 6 –20
CC 7 –20
CC 8 –20
CC 9 –20
Program execution order (in instructions)
2b. MEM/WB. RegisterRd = ID/EX. RegisterRt = $2
sub $2, $1, $3
and $12, $2, $5
or $13, $6, $2
add $14, $2,$2
sw $15, 100($2)
IM
Reg
DM

	Time (in clock cycles)								
	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
Value of register $2:	10	10	10	10	10/–20	–20	–20	–20	–20
Value of EX/MEM:	X	X	X	–20	X	X	X	X	X
Value of MEM/WB:	X	X	X	X	–20	X	X	X	X



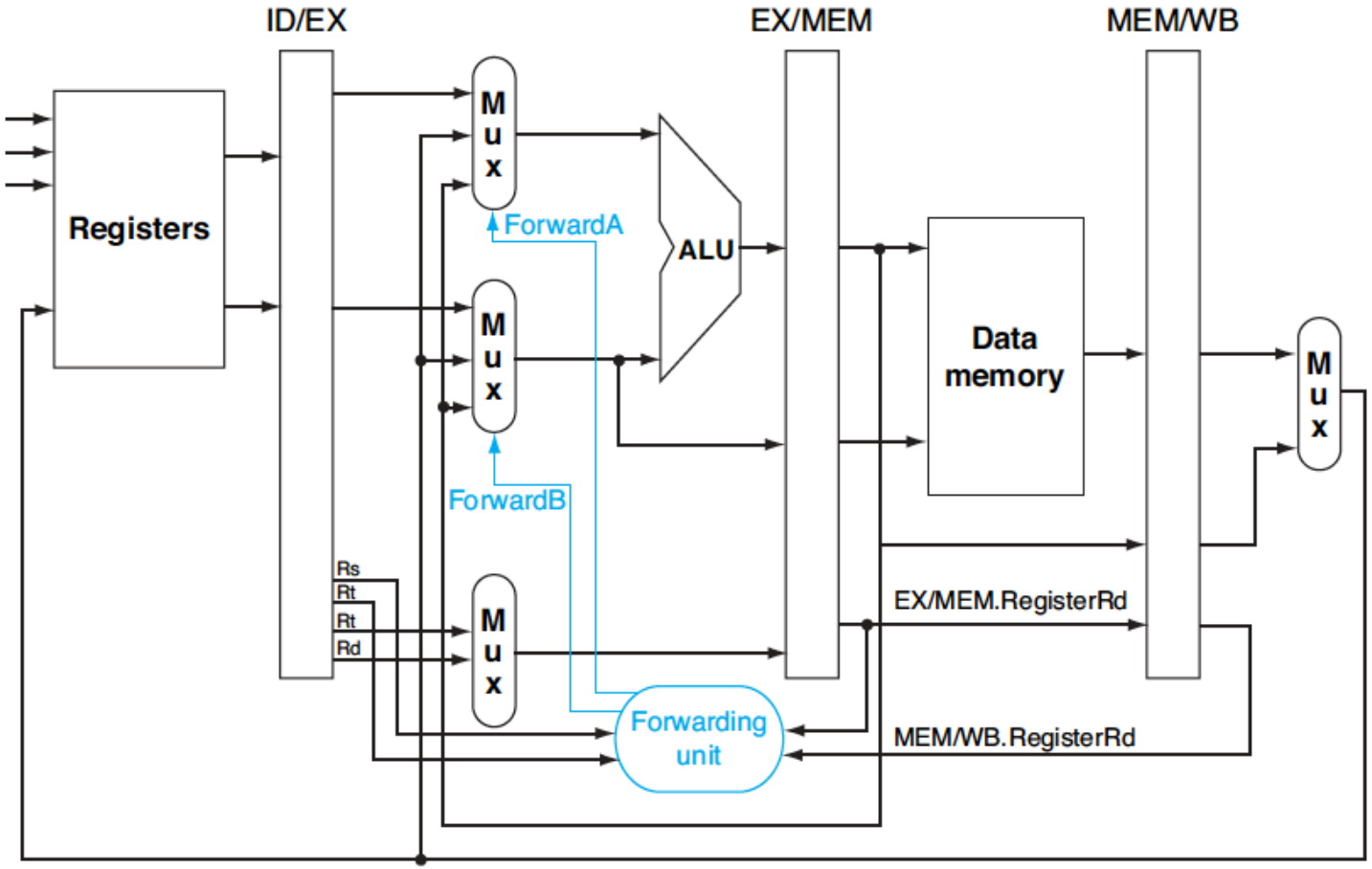
- The above policy may be inaccurate when the instruction does not write registers such that it would forward when it shouldn't
  - Examining the WB control field of the pipeline register during the EX and MEM stages determines whether RegWrite is asserted
- In MIPS, $0 should always yield an operand of 0. What if an instruction has $0 as its destination (e.g., sll $0, $1, 2)
  - We have to avoid forwarding its possibly nonzero result value

1a. EX/MEM.RegisterRd = ID/EX.RegisterRs
1b. EX/MEM.RegisterRd = ID/EX.RegisterRt

EX/MEM. RegisterRd ≠ 0

2a. MEM/WB.RegisterRd = ID/EX.RegisterRs
2b. MEM/WB.RegisterRd = ID/EX.RegisterRt

MEM/WB. RegisterRd ≠ 0



Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

1. *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
```

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

2. *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
```

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

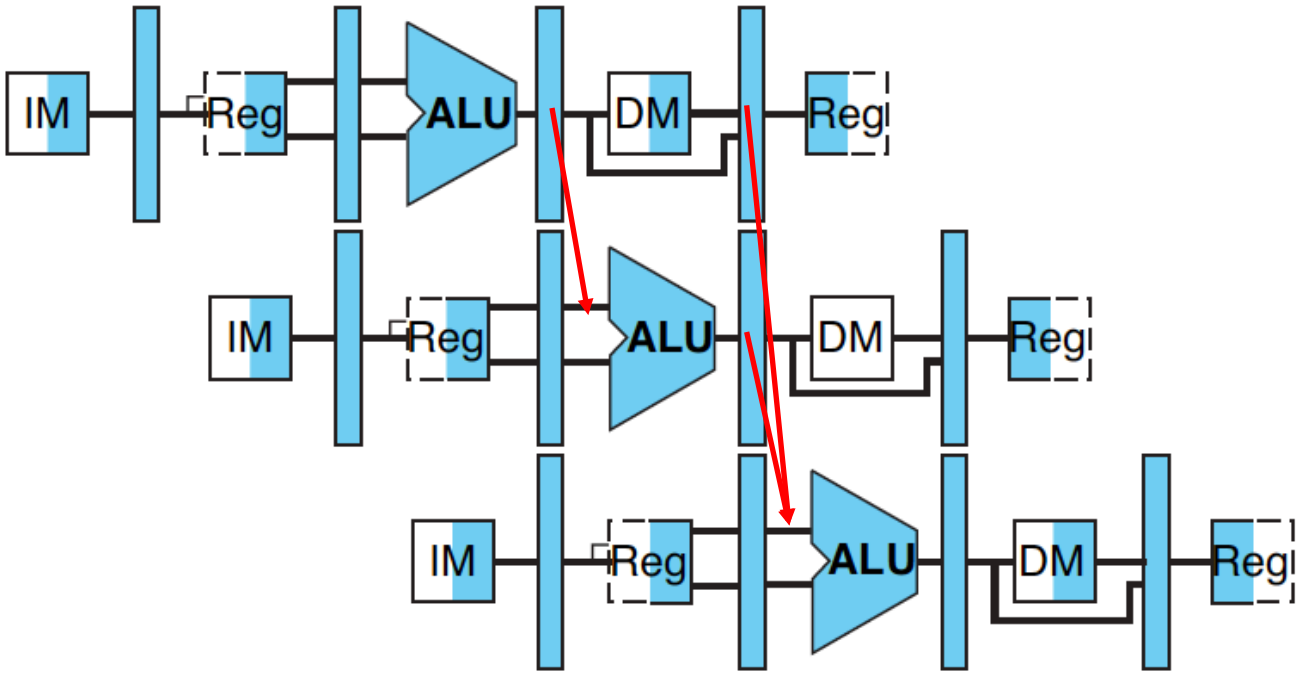
Another potential data hazard can occur when there is a conflict between the result of the WB stage instruction and the MEM stage instruction – which should be forwarded?

- *Don't even try to forward from MEM/WB to EX; if there is already forwarding of more recent result from EX/MEM.*

add $1, $1, $2

add $1, $1, $3

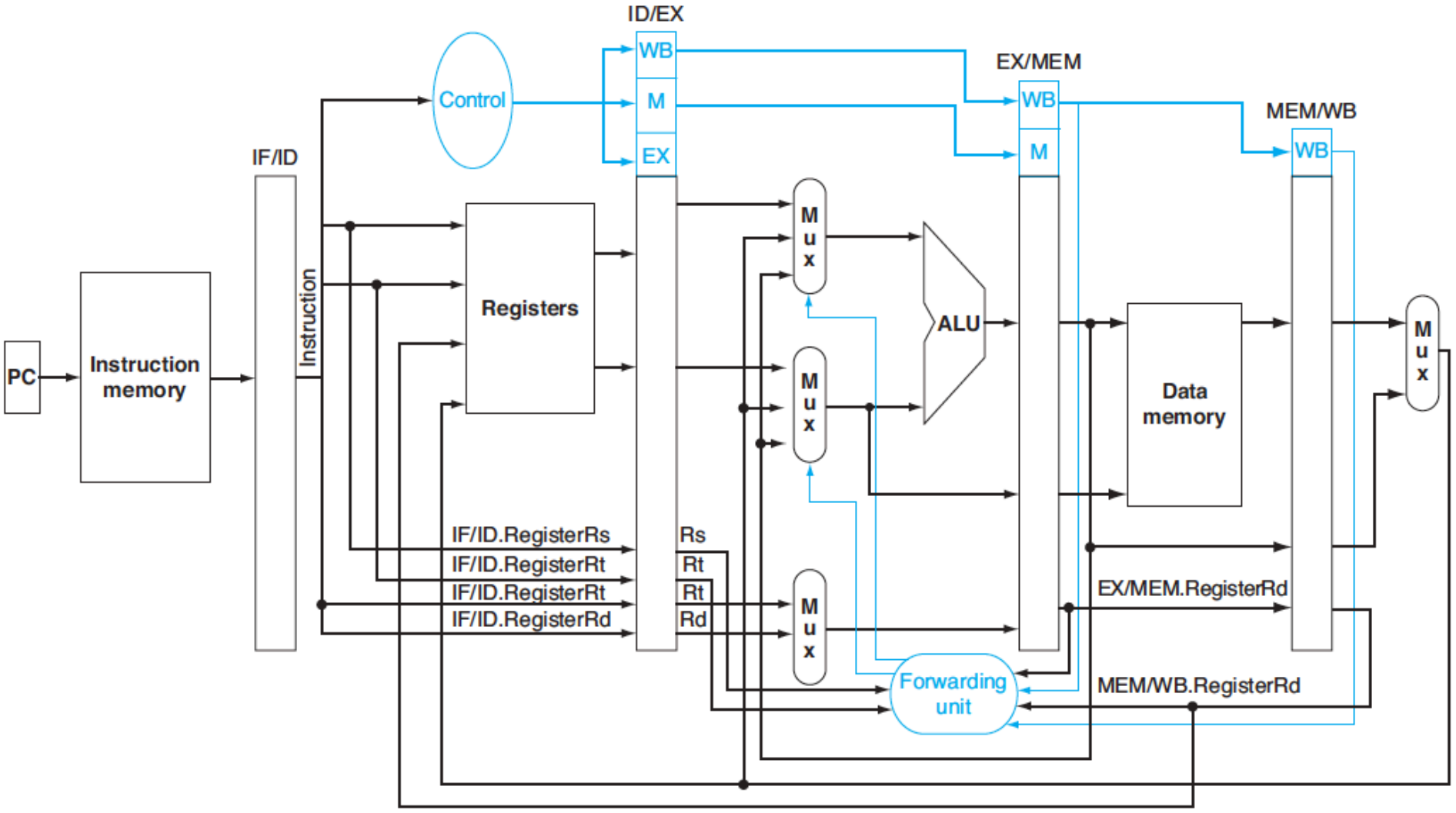
add $1, $1, $4



if (MEM/WB. RegWrite
and (MEM/WB. RegisterRd ≠ 0)
and not (EX/MEM. RegWrite and (EX/MEM. RegisterRd ≠ 0)
and (EX/MEM. RegisterRd = ID/EX. RegisterRs))
and (MEM/WB. RegisterRd = ID/EX. RegisterRs)) ForwardA = 01

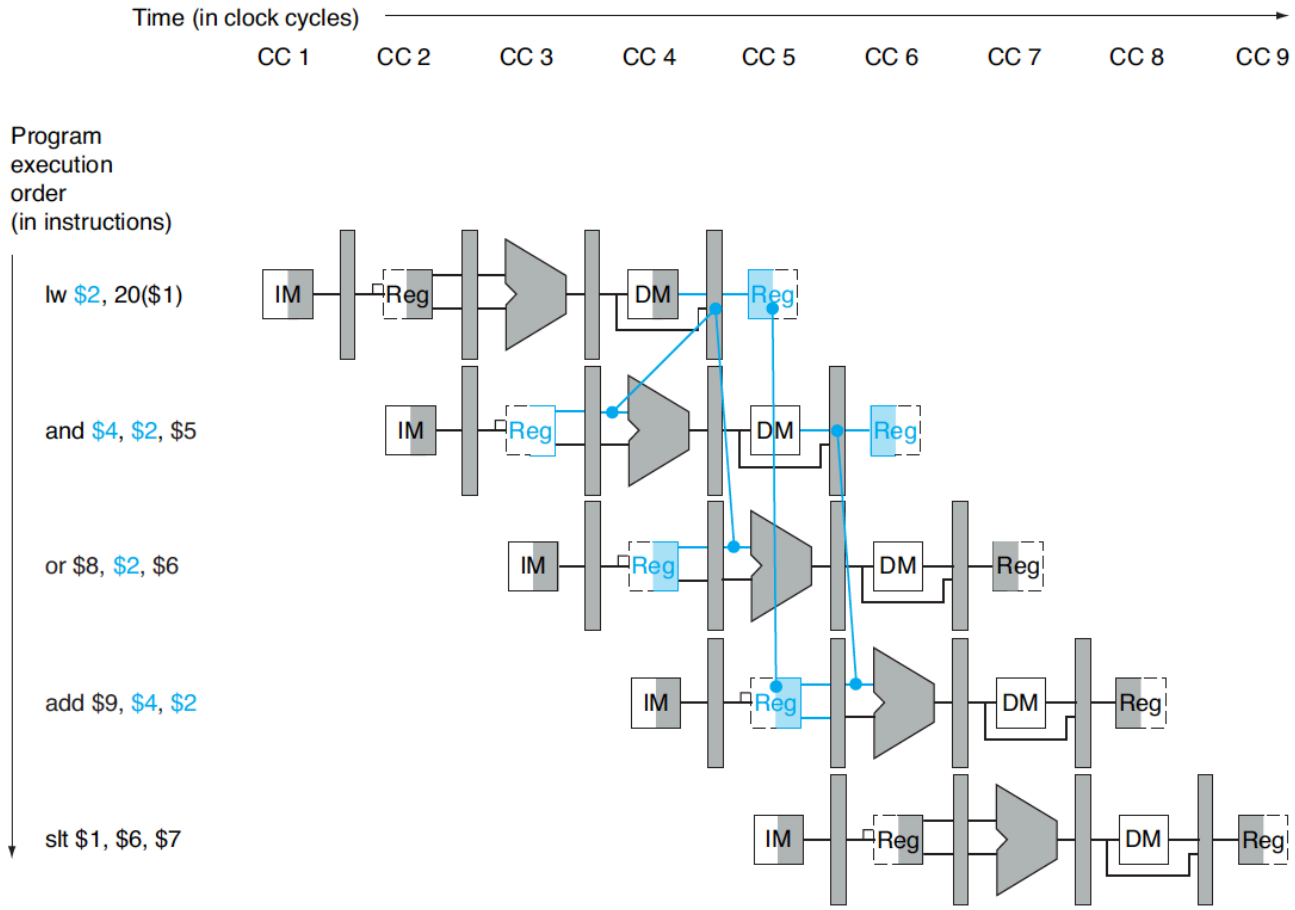
if (MEM/WB. RegWrite
and (MEM/WB. RegisterRd ≠ 0)
and not (EX/MEM. RegWrite and (EX/MEM. RegisterRd ≠ 0)
and (EX/MEM. RegisterRd = ID/EX. RegisterRt))
and (MEM/WB. RegisterRd = ID/EX. RegisterRt)) ForwardB = 01

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.


ID/EX
EX/MEM
MEM/WB
WB
M
EX
Control
IF/ID
PC
Instruction memory
Instruction
Registers
M u x
ALU
Data memory
IF/ID.RegisterRs
IF/ID.RegisterRt
IF/ID.RegisterRt
IF/ID.RegisterRd
Rs
Rt
Rt
Rd
EX/MEM.RegisterRd
MEM/WB.RegisterRd
Forwarding unit

Data Hazards and Stalls

Since the dependence between the load and the following instruction (and) goes backward in time, this hazard cannot be solved by forwarding. Hence, this combination must result in a stall by the hazard detection unit.



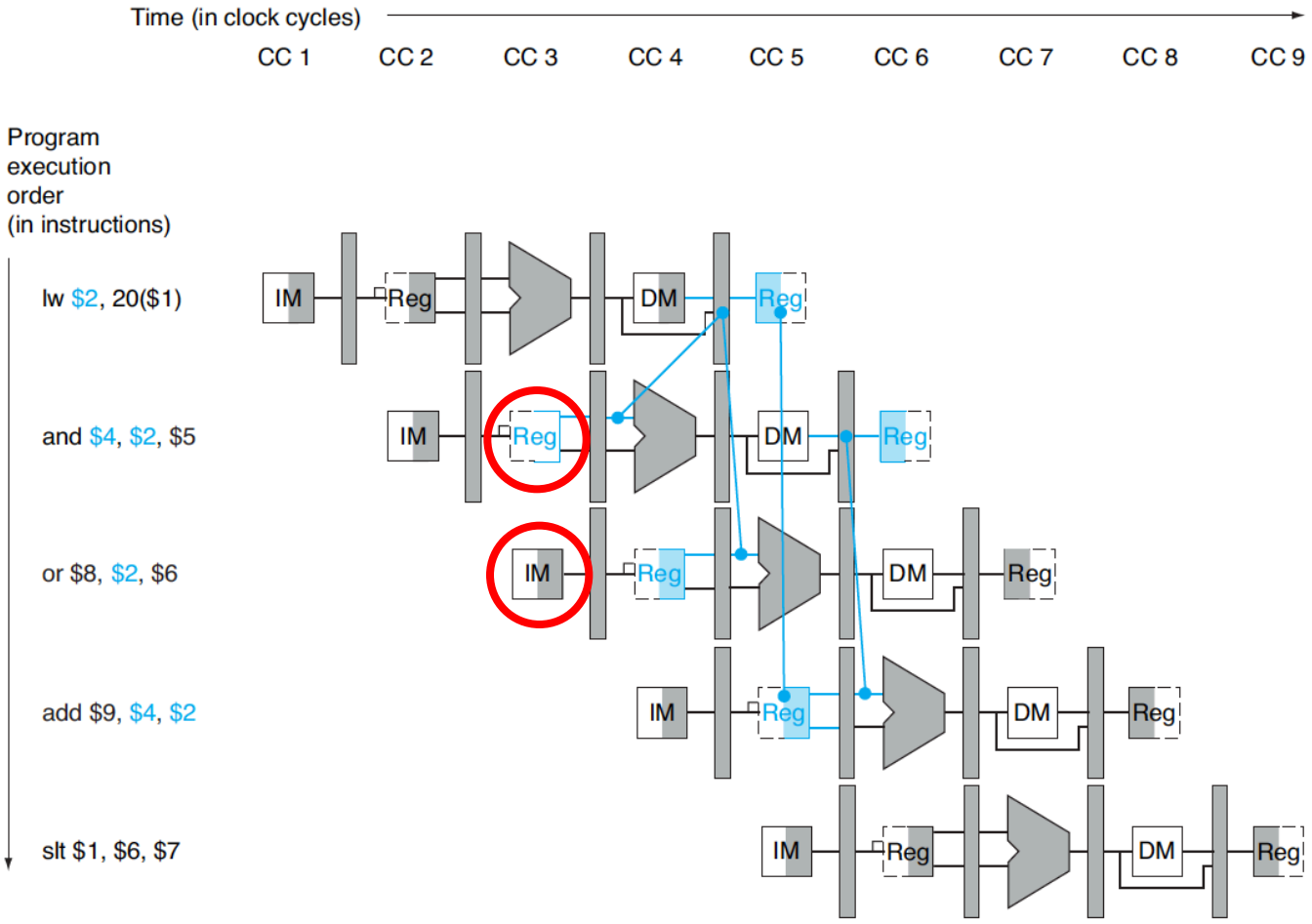
Hazard detection unit

It operates during the ID stage so that it can insert the stall between the load and its use.

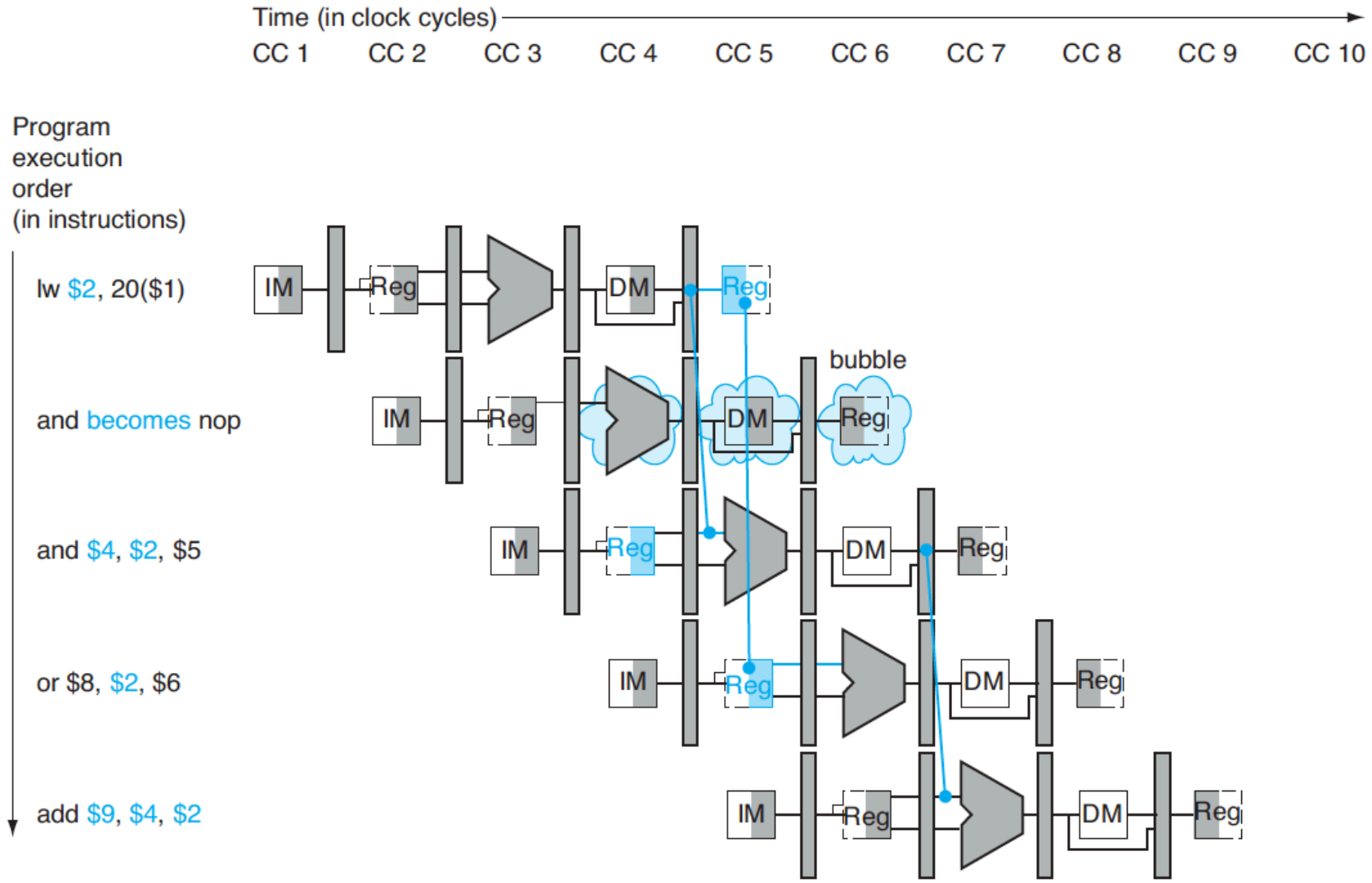
```
if (ID/EX.MemRead and                                          Line 1
   ((ID/EX.RegisterRt = IF/ID.RegisterRs) or                   Line 2
     (ID/EX.RegisterRt = IF/ID.RegisterRt)))                   Line 3
      stall the pipeline                                       Line 4
```

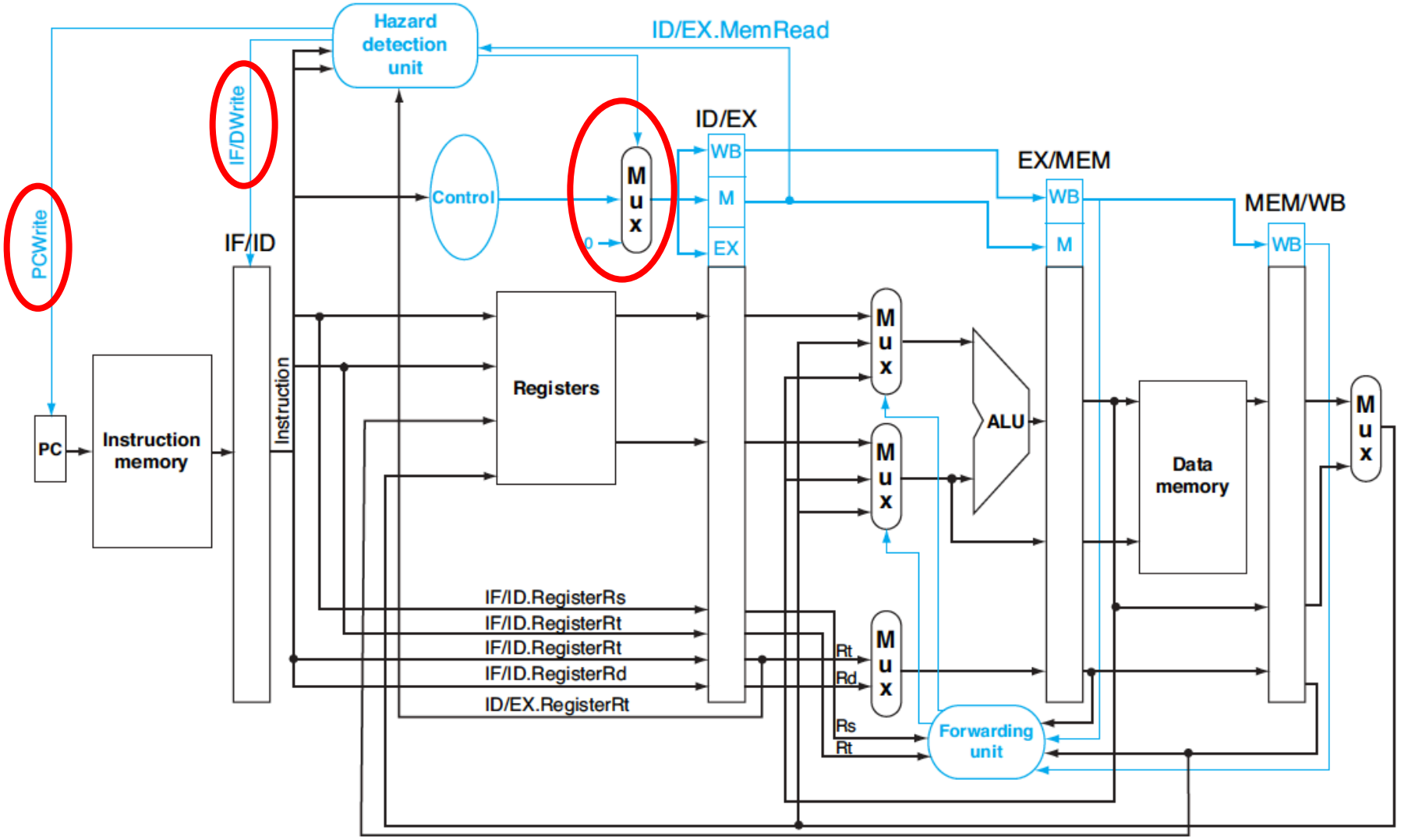
- Line 1: Check if the instruction is a load
- Line 2 and Line 3: Check if the destination register field of the load instruction in the EX stage matches either the source register of the instruction in the ID stage
- Lin 4: The instruction stalls one clock cycle

- If the instruction in ID stage is stalled, then the instruction in the IF stage must also be stalled; otherwise, the fetched instruction would be lost
- How to stall an instruction?
  - Preventing the PC register and the IF/ID pipeline register from changing
  - The back half of the pipeline (starting with the EX stage) must be performed with no effect

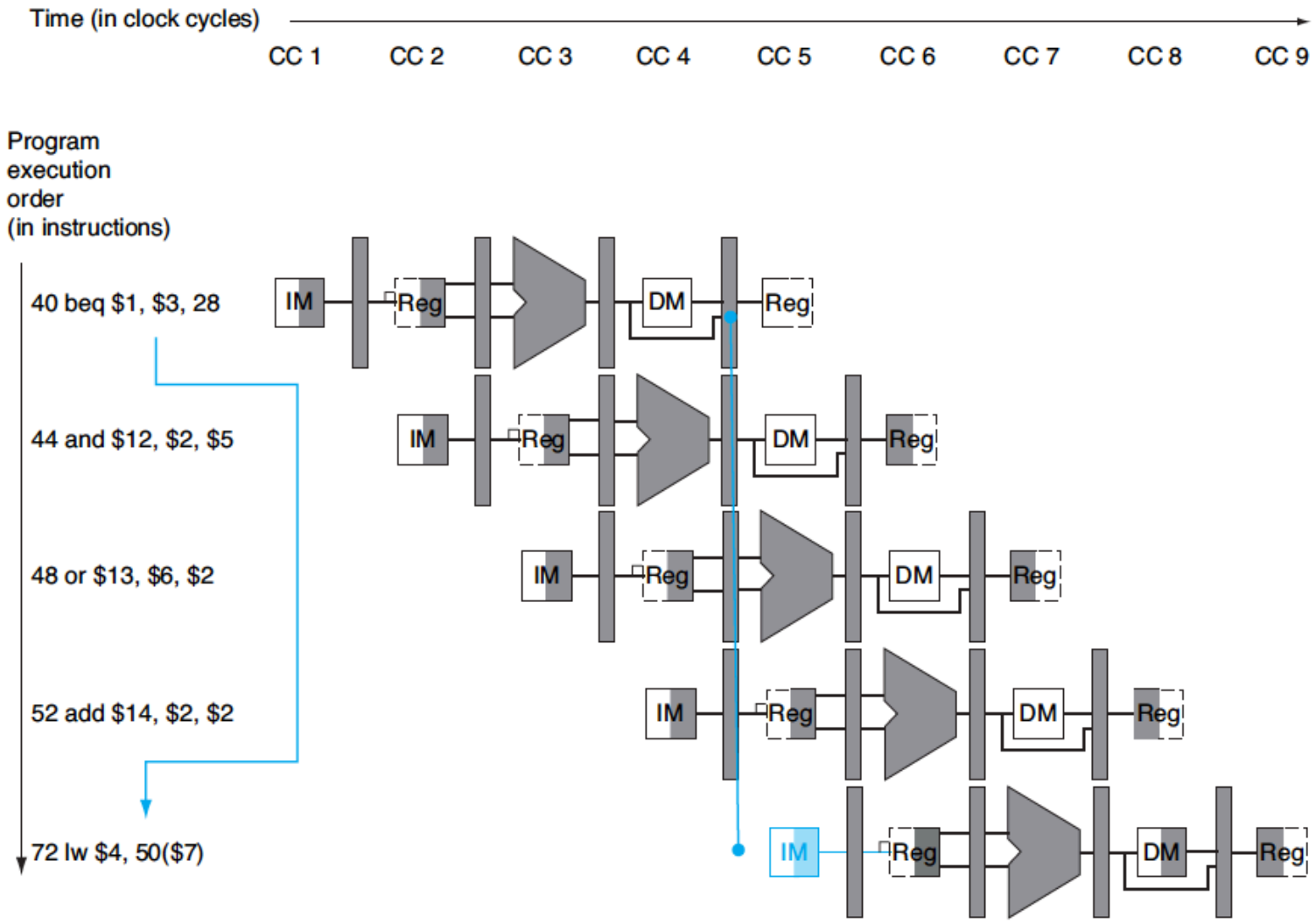


Changing the EX, MEM, and WB control fields of the ID/EX pipeline register to 0, which will result in a **nop** instruction.



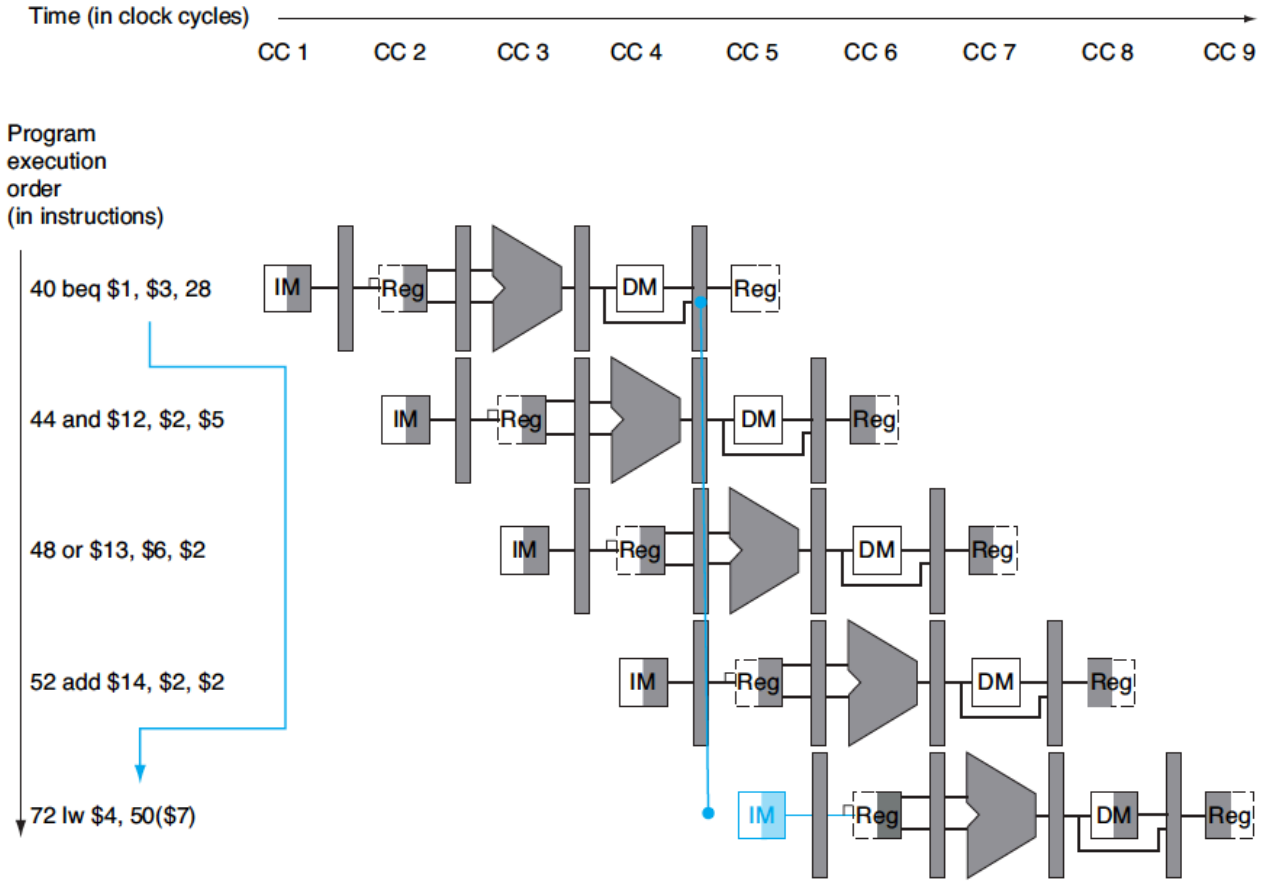

Hazard detection unit
ID/EX.MemRead
IF/DWrite
PCWrite
Control
M u x
0
ID/EX
WB
M
EX
EX/MEM
WB
M
MEM/WB
WB
IF/ID
PC
Instruction memory
Instruction
Registers
M u x
ALU
Data memory
M u x
M u x
M u x
IF/ID.RegisterRs
IF/ID.RegisterRt
IF/ID.RegisterRt
IF/ID.RegisterRd
ID/EX.RegisterRt
Rt
Rd
Rs
Rt
Forwarding unit

Control hazard



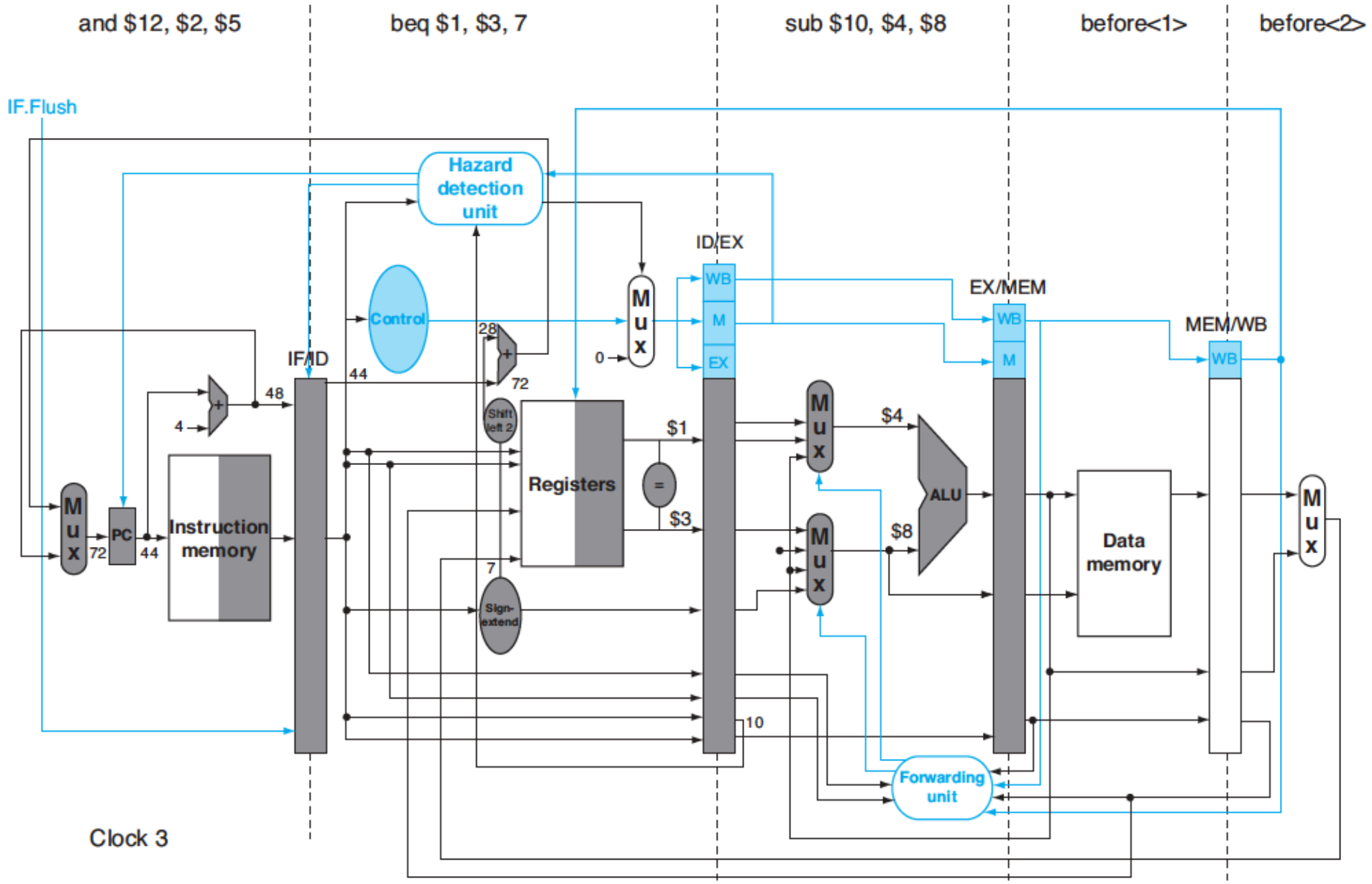
Assume Branch Not Taken

- Predict that the branch will not be taken and thus continue execution down the sequential instruction stream
- What if we make a wrong prediction?
  - Discard the instruction that are being fetched and decoded
  - Execution continues at the branch target

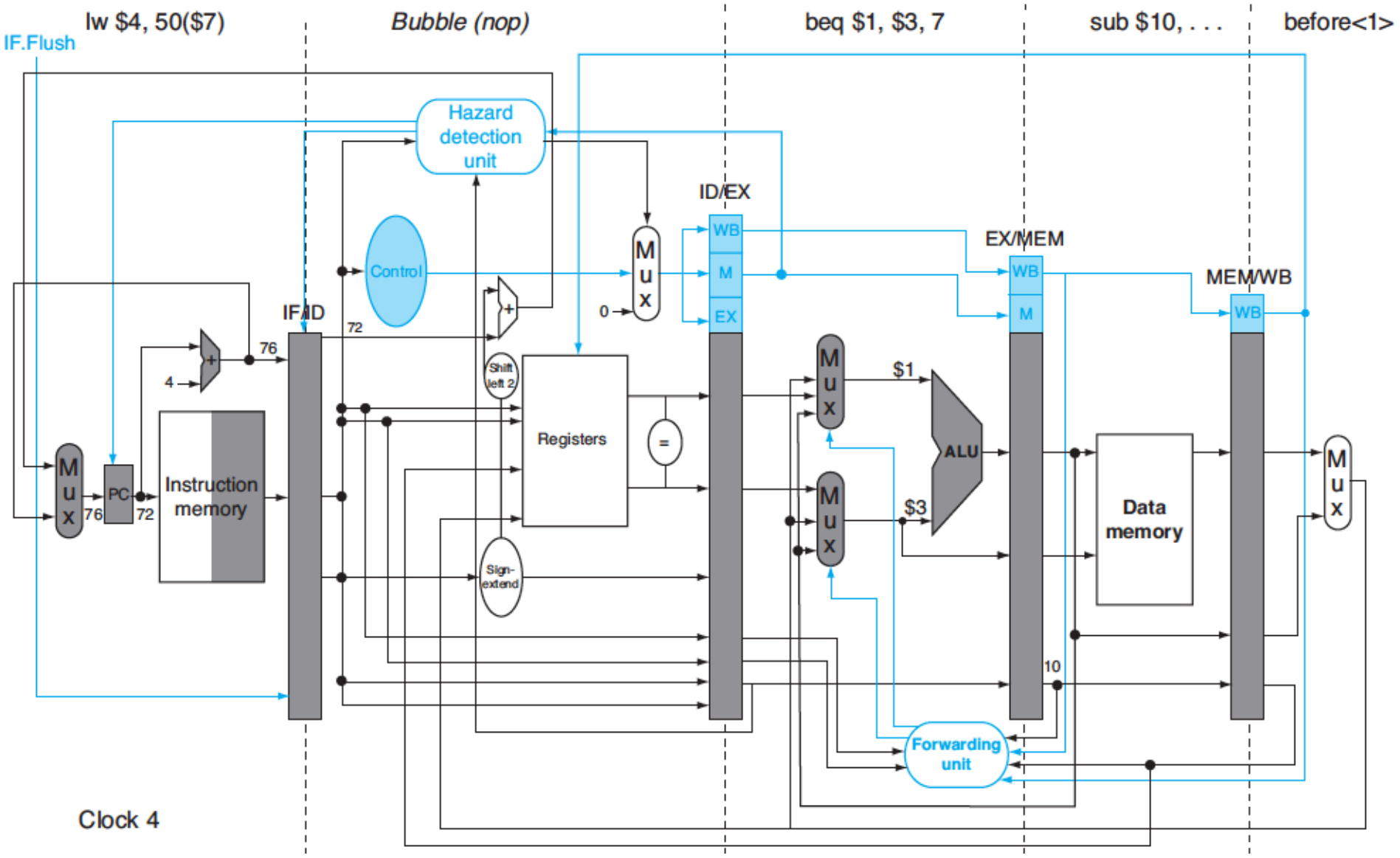


Reducing the Delay of Branches

- Reduce the cost of the taken branch
  - Computing the branch target address: move the branch adder from the EX stage to the ID stage
  - Evaluating the branch decision (comparing the two registers read during the ID stage to see if they are equal): first XORing their respective bits and then ORing all the results



The ID stage of clock cycle 3 determines that a branch must be taken, so it selects 72 as the next PC address and zeros the instruction fetched for the next clock cycle



Clock cycle 4 shows the instruction at location 72 being fetched and the single bubble or nop instruction in the pipeline as a result of the taken branch

Dynamic branch prediction

- Prediction of branches at runtime using runtime information
  - Look up the address of the instruction to see if a branch was taken the last time this instruction was executed, and, if so, to begin fetching new instructions from the same place as the last time.
  - Branch prediction buffer (branch history table): A small memory that is indexed by the lower portion of the address of the branch instruction and that contains one or more bits indicating whether the branch was recently taken or not

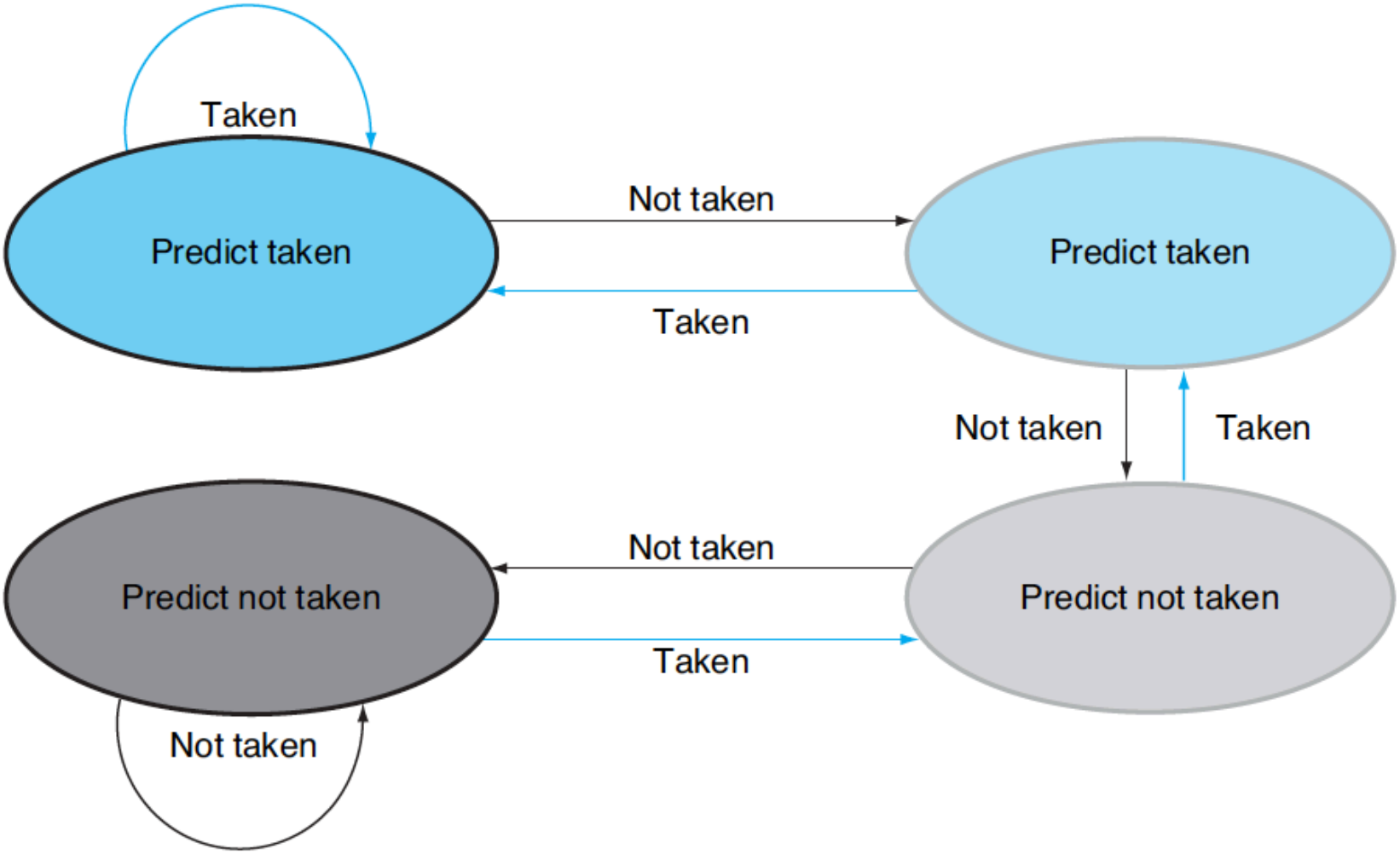
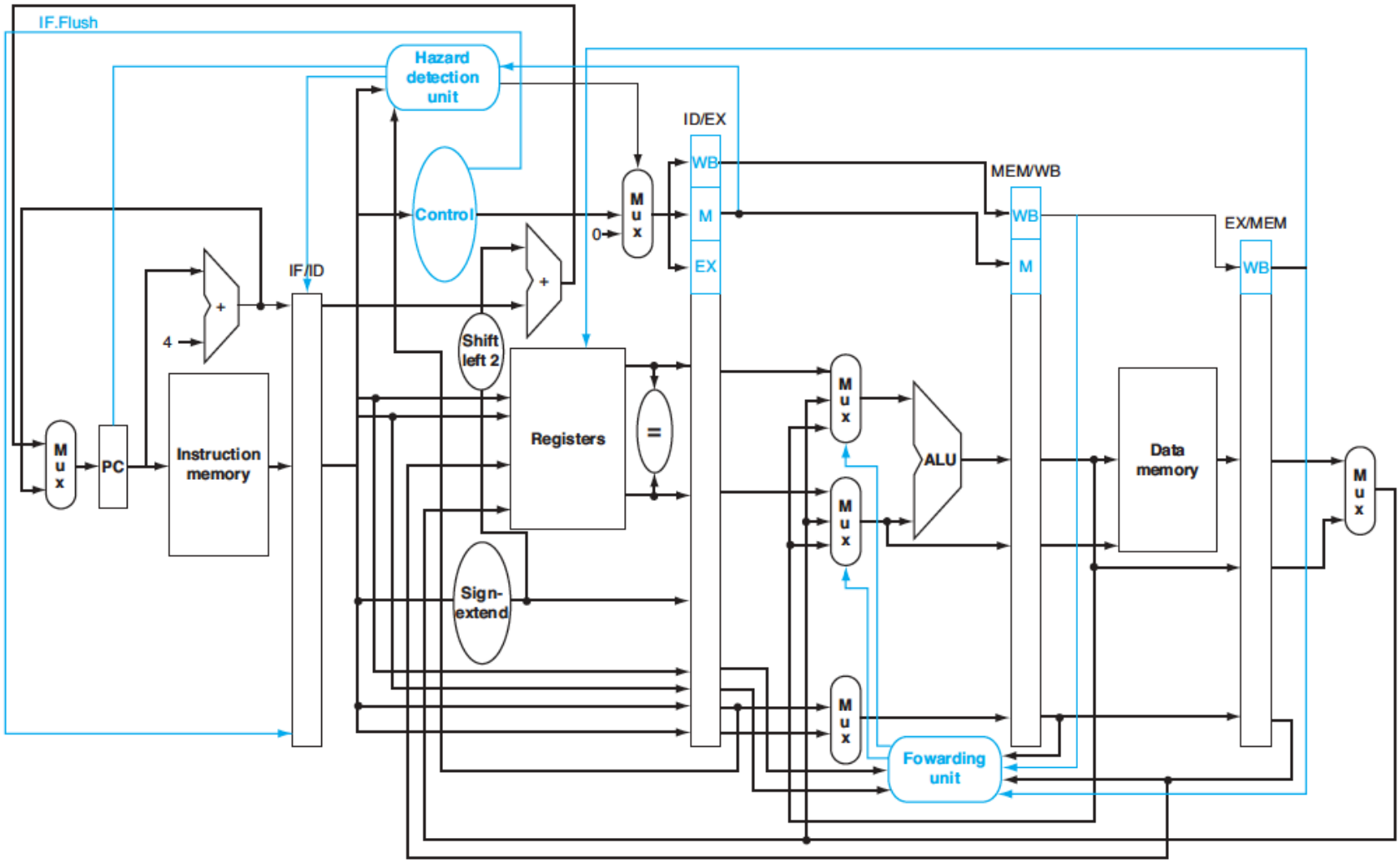


FIGURE 4.63 The states in a 2-bit prediction scheme. By using 2 bits rather than 1, a branch that strongly favors taken or not taken—as many branches do—will be mispredicted only once. The 2 bits are used to encode the four states in the system. The 2-bit scheme is a general instance of a counter-based predictor, which is incremented when the prediction is accurate and decremented otherwise, and uses the midpoint of its range as the division between taken and not taken.

Pipeline Summary



Exceptions

- Exception: Any unexpected change in control flow without distinguishing whether the cause is internal or external
- Interruption: An exception that comes from outside of the processor

Type of event	From where?	MIPS terminology
I/O device request	External	Interrupt
Invoke the operating system from user program	Internal	Exception
Arithmetic overflow	Internal	Exception
Using an undefined instruction	Internal	Exception
Hardware malfunctions	Either	Exception or interrupt

How exceptions are handled in the MIPS architecture

- Two types of exceptions in our current MIPS implementation
  - Execution of an undefined instruction
  - An arithmetic overflow
- A basic action that must be performed when an exception occurs
  - Save the address of the offending instruction in the 32-bit *exception program counter* (EPC)
- Actions taken to deal with exceptions
  - Providing some service to user program
  - Taking predefined action in response to an overflow (or stopping the execution of the program)
  - Reporting an error
- When the above actions are done
  - Terminating the program or continue its execution using the EPC to return to where the program is interrupted

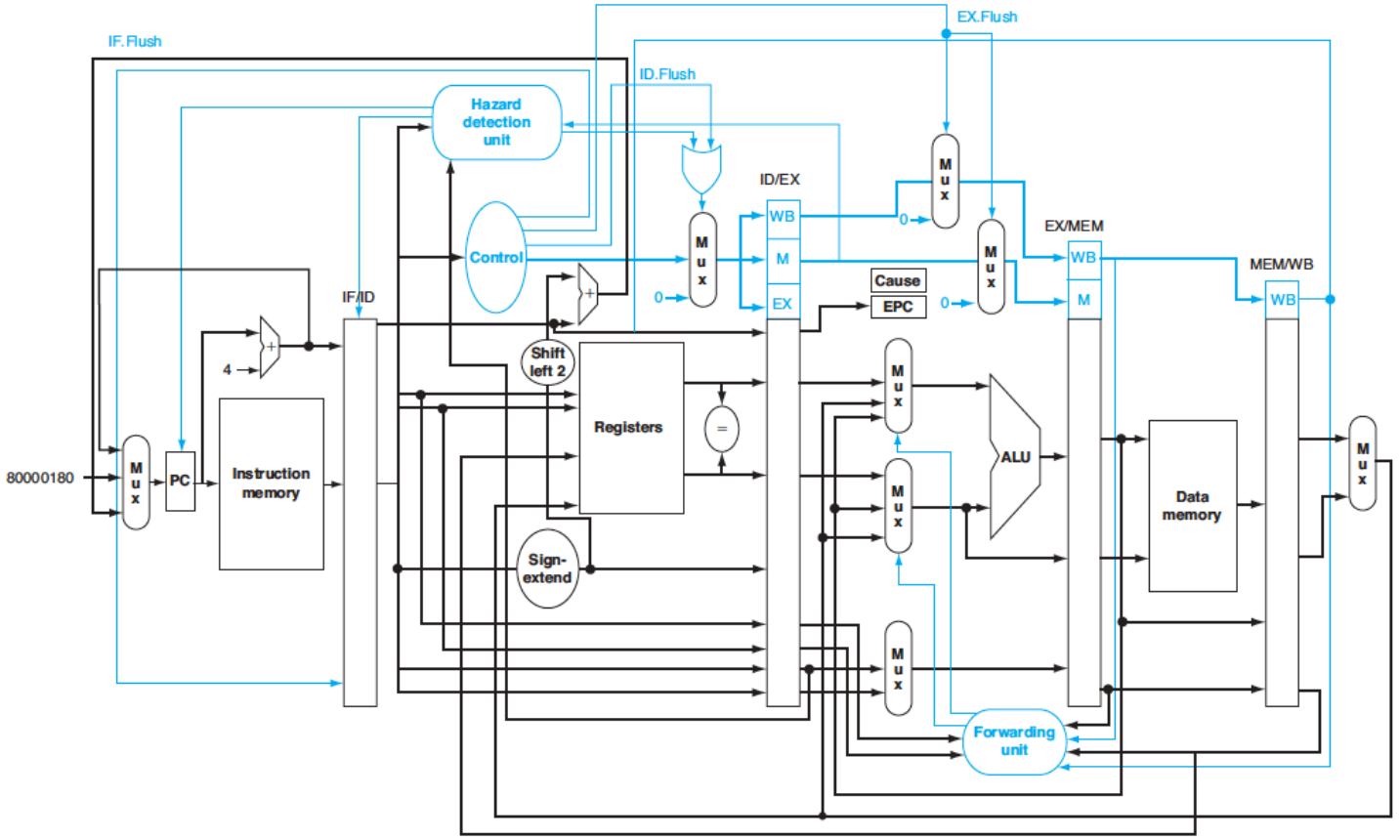
- To take proper actions to handle exceptions, the operating system must know the reason for the exception
  - 32-bit Cause register used by MIPS (a status register holding a field that indicates the reason for the exception)
  - Vectored interrupts (an interrupt for which the address to which control is transferred is determined by the cause the exception)

Exception type	Exception vector address (in hex)
Undefined instruction	$8000\ 0000_{hex}$
Arithmetic overflow	$8000\ 0180_{hex}$

- When the exception is not vectored, a single entry point ($8000\ 0180_{16}$) for all exceptions should be used, and the operating system decodes the status register to find the cause

Exceptions in a pipelined implementation

- In pipeline, exceptions can be treated as another form of control hazard
  - Detecting exception in EX stage
  - Flushing the instructions which are in the stages of IF, ID, and EX
  - Saving the address of the offending instruction in the EPC

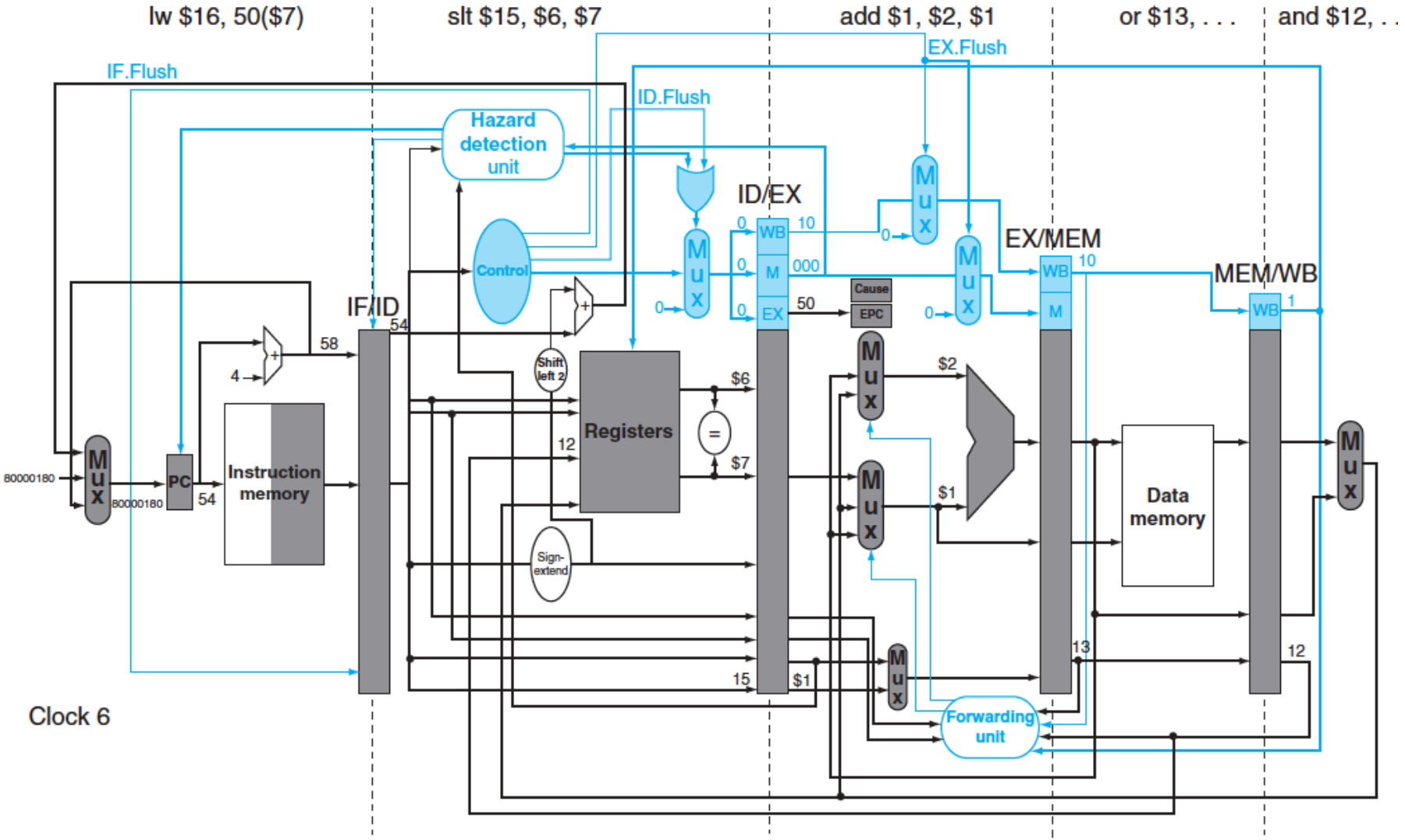


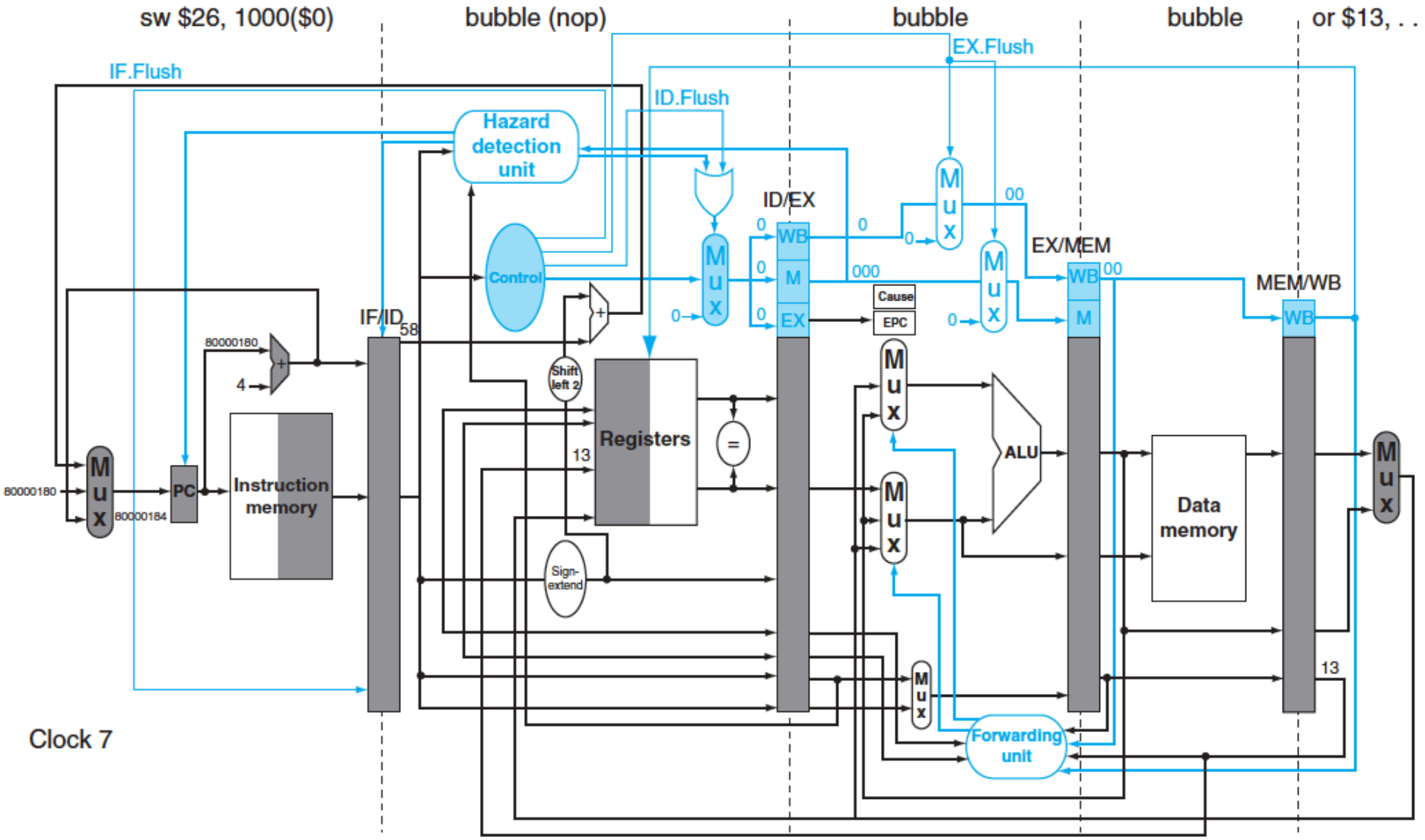
Given this instruction sequence

```
40hex  sub   $11, $2, $4
44hex  and   $12, $2, $5
48hex  or    $13, $2, $6
4Chex  add    $1, $2, $1
50hex  slt   $15, $6, $7
54hex  lw    $16, 50($7)
...
```

Assume the instruction to be invoked on an exception begin like this

```
80000180hex  sw      $26, 1000($0)
80000184hex  sw      $27, 1004($0)
...
```


lw $16, 50($7)
slt $15, $6, $7
add $1, $2, $1
or $13, . . .
and $12, . .
IF.Flush
ID.Flush
EX.Flush
Hazard detection unit
Control
ID/EX
EX/MEM
MEM/WB
IF/ID
WB
M
EX
Cause
EPC
Shift left 2
Registers
Instruction memory
Data memory
Sign-extend
Forwarding unit
PC
Mux
80000180
Clock 6


sw $26, 1000($0)
bubble (nop)
bubble
bubble
or $13, . .
IF.Flush
EX.Flush
ID.Flush
Hazard detection unit
ID/EX
EX/MEM
MEM/WB
Control
IF/ID
58
WB
M
EX
Cause
EPC
000
00
80000180
4
PC
80000184
Instruction memory
Shift left 2
Registers
13
=
Sign-extend
ALU
Data memory
Forwarding unit
Mux
Clock 7

Parallelism via Instructions

- Instruction-Level Parallelism: Pipelining exploits the potential parallelism among instructions.
  - Increasing the depth of the pipeline to overlap more instructions
  - Replicating the internal components of the computer so that it can launch multiple instructions in every pipeline stage (also called "*multiple issue*")
- Multiple issue
  - Static multiple issue: An approach to implementing a multiple-issue processor where many decisions are made by the compiler before execution
  - Dynamic multiple issue: An approach to implementing a multiple-issue processor where many decisions are made during execution by the processor (also called "*superscalar*")

Implementing multiple-issue pipeline

- Packaging instruction into issue slots
  - How many instruction can be issued in a given clock cycle?
  - Which instruction can be issued in a given clock cycle?
- Dealing with data and control hazards
  - In static issue processor, the compiler handles some (or all) of the consequences of data/control hazards
  - In dynamic issue processor, hardware techniques operating at execution time are used to alleviate at least some classes of hazards

Static multiple issue

- Static multiple-issue processors package instructions and deal with hazards through compilers
  - Instructions are packaged into issue packets each of which can be executed in one clock cycle, such that each multiple issue can be considered as a single instruction allowing several operations in certain predefined fields (so-called *Very Long Instruction* Word or *VLIW*)
  - Most static issue processors rely on compilers to take on some responsibility for handling data and control hazards

Dynamic multiple-issue processors (Superscalar)

- In the simplest superscalar processor, instructions issue in order, and the processor decides how many instructions can issue in a given clock cycle
- Compiler is still needed to schedule instruction to move dependences apart and thereby improve the instruction issue rate
- Different from VLIW processors,
  - The code, whether scheduled or not, is guarantted by the harware to execute correctly
  - Compiled code will always run correctly independent of the issue rate or pipeline structure

Thanks !