Computer Organization and Structure

Bing-Yu Chen
National Taiwan University

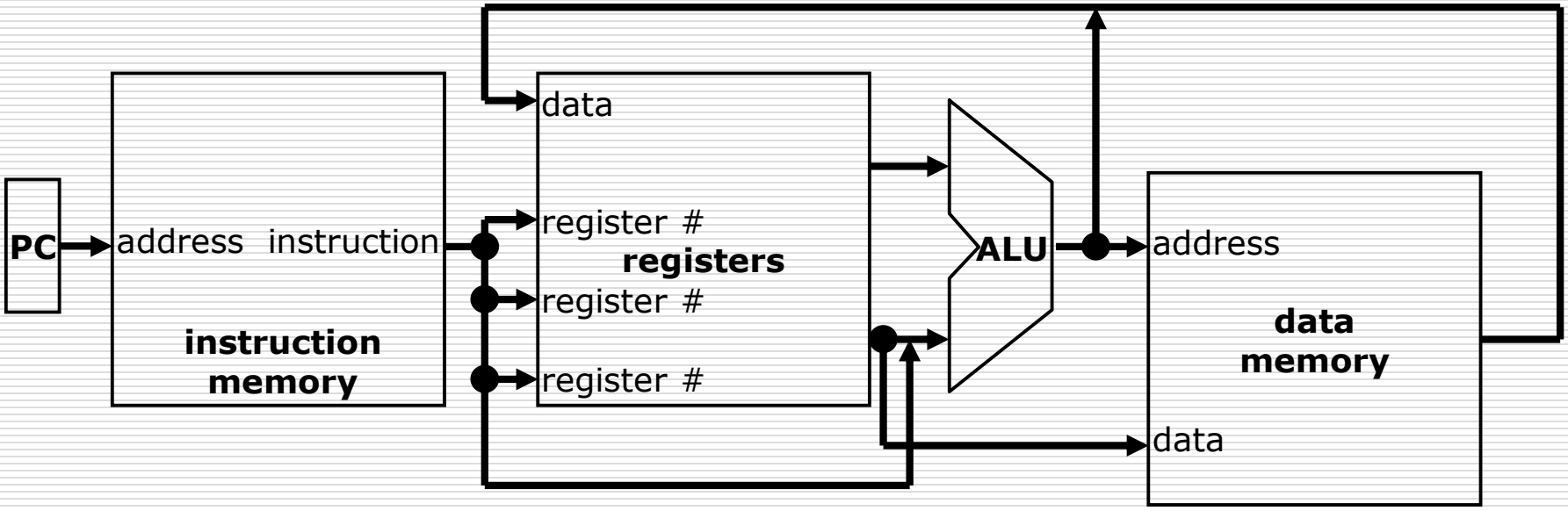
The Processor

- Logic Design Conventions
- Building a Datapath
- A Simple Implementation Scheme
- An Overview of Pipelining
- Pipelined Datapath and Control
- Data Hazards: Forwarding vs. Stalls
- Control Hazards
- Exceptions

Instruction Execution

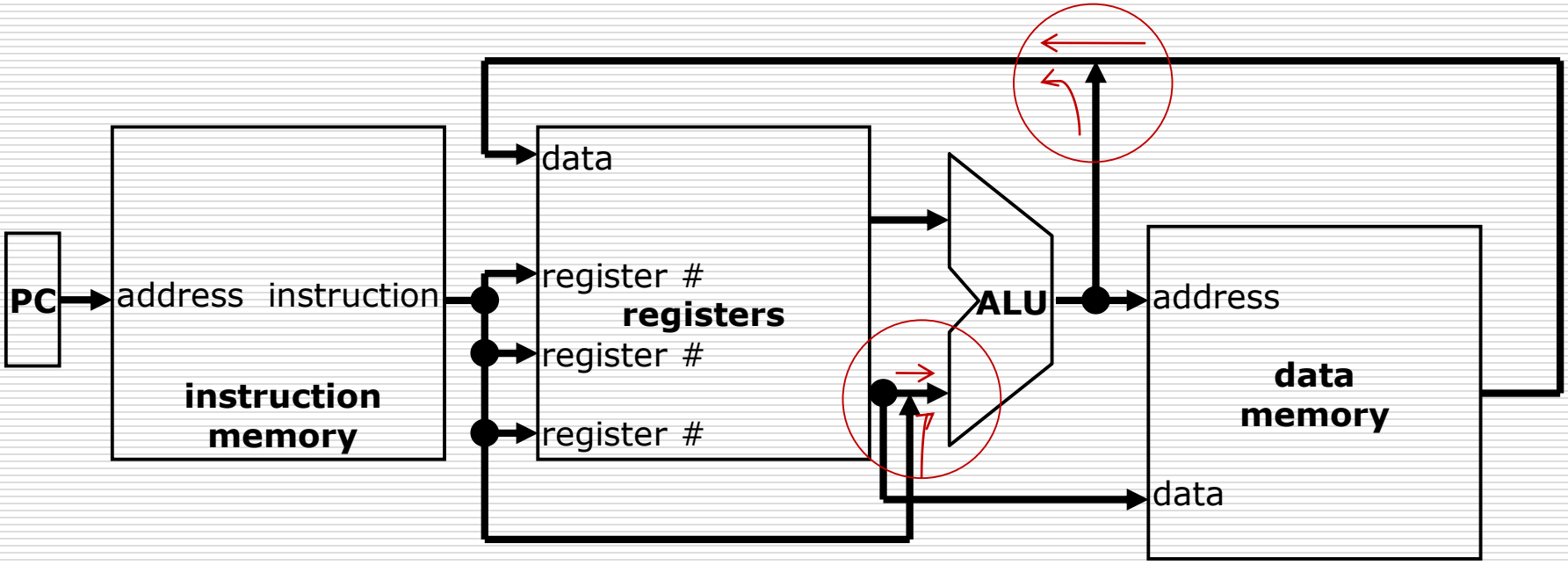
- PC $\rightarrow$ instruction memory, fetch instruction
- Register numbers $\rightarrow$ register file, read registers
- Depending on instruction class
  - Use ALU to calculate
    - Arithmetic result
    - Memory address for load/store
    - Branch target address
  - Access data memory for load/store
  - PC $\leftarrow$ target address or PC + 4

Abstract / Simplified View



- Two types of functional units:
  - elements that operate on data values (combinational)
  - elements that contain state (sequential)

Abstract / Simplified View



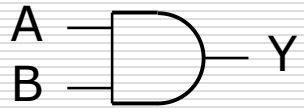
- Cannot just join wires together
  - Use multiplexers

Recall: Logic Design Basics

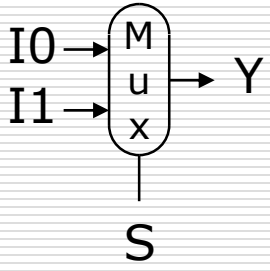
- Information encoded in binary
  - Low voltage = 0, High voltage = 1
  - One wire per bit
  - Multi-bit data encoded on multi-wire buses
- Combinational element
  - Operate on data
  - Output is a function of input
- State (sequential) elements
  - Store information

Recall: Combinational Elements

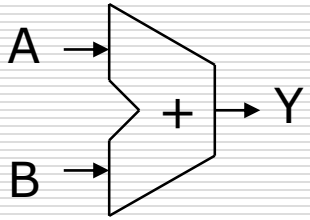
- AND-gate
  - Y = A & B



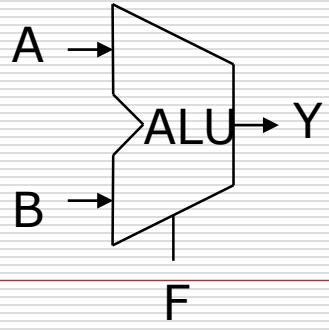
- Multiplexer
  - Y = S ? I1 : I0



- Adder
  - Y = A + B

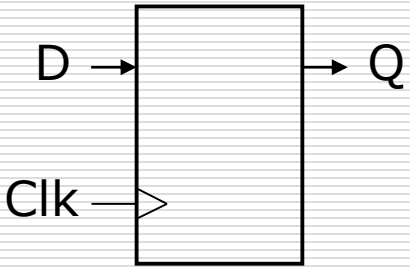


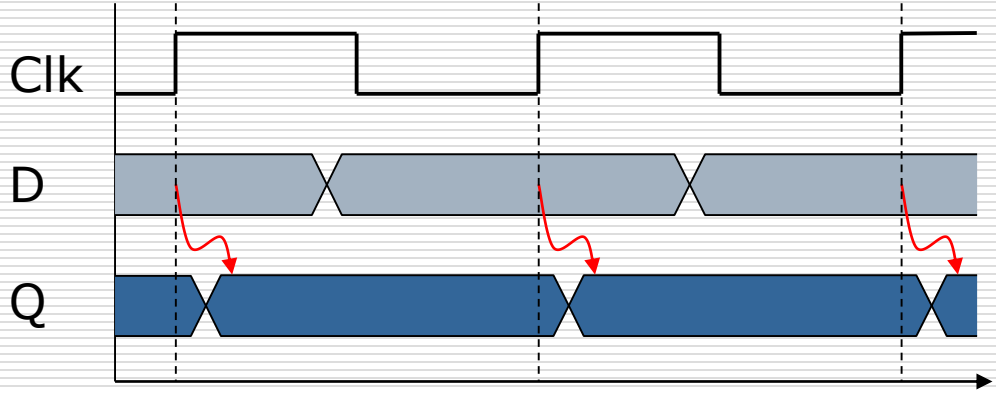
- ALU
  - Y = F(A, B)



Sequential Elements

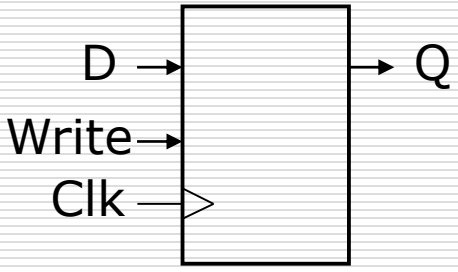
- Register: stores data in a circuit
  - Uses a clock signal to determine when to update the stored value
  - Edge-triggered: update when Clk changes from 0 to 1

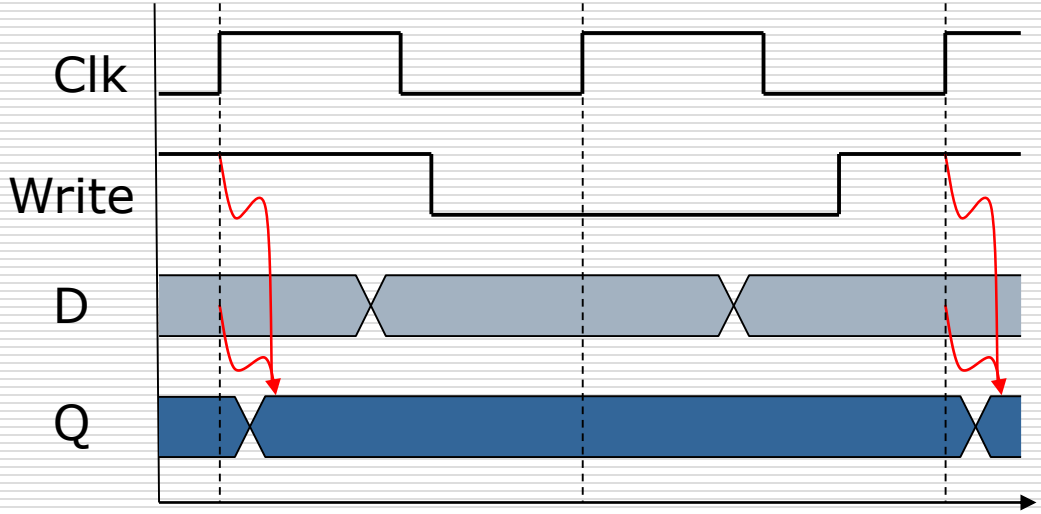




Sequential Elements

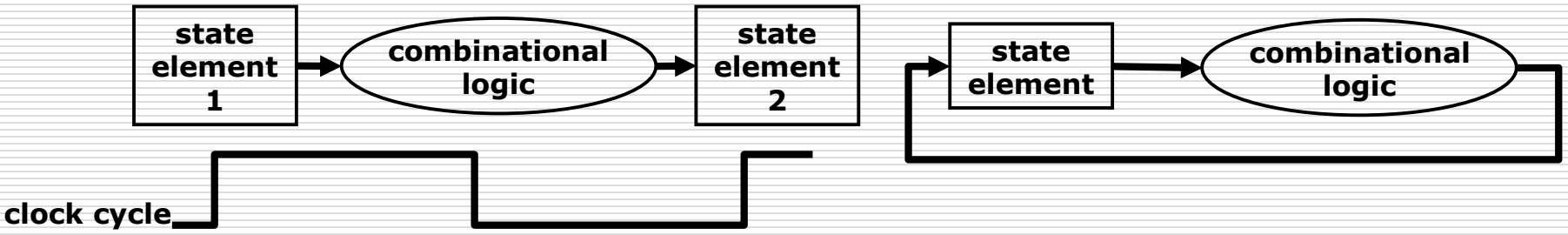
- Register with write control
  - Only updates on clock edge when write control input is 1
  - Used when stored value is required later





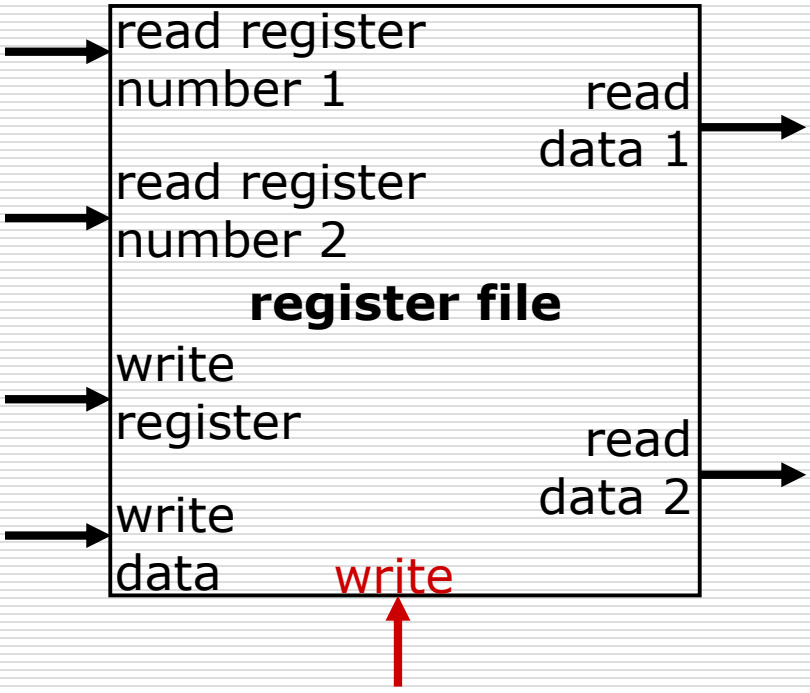
Clocking Methodology

- Combinational logic transforms data during clock cycles
  - Between clock edges
  - Input from state elements, output to state element
  - Longest delay determines clock period

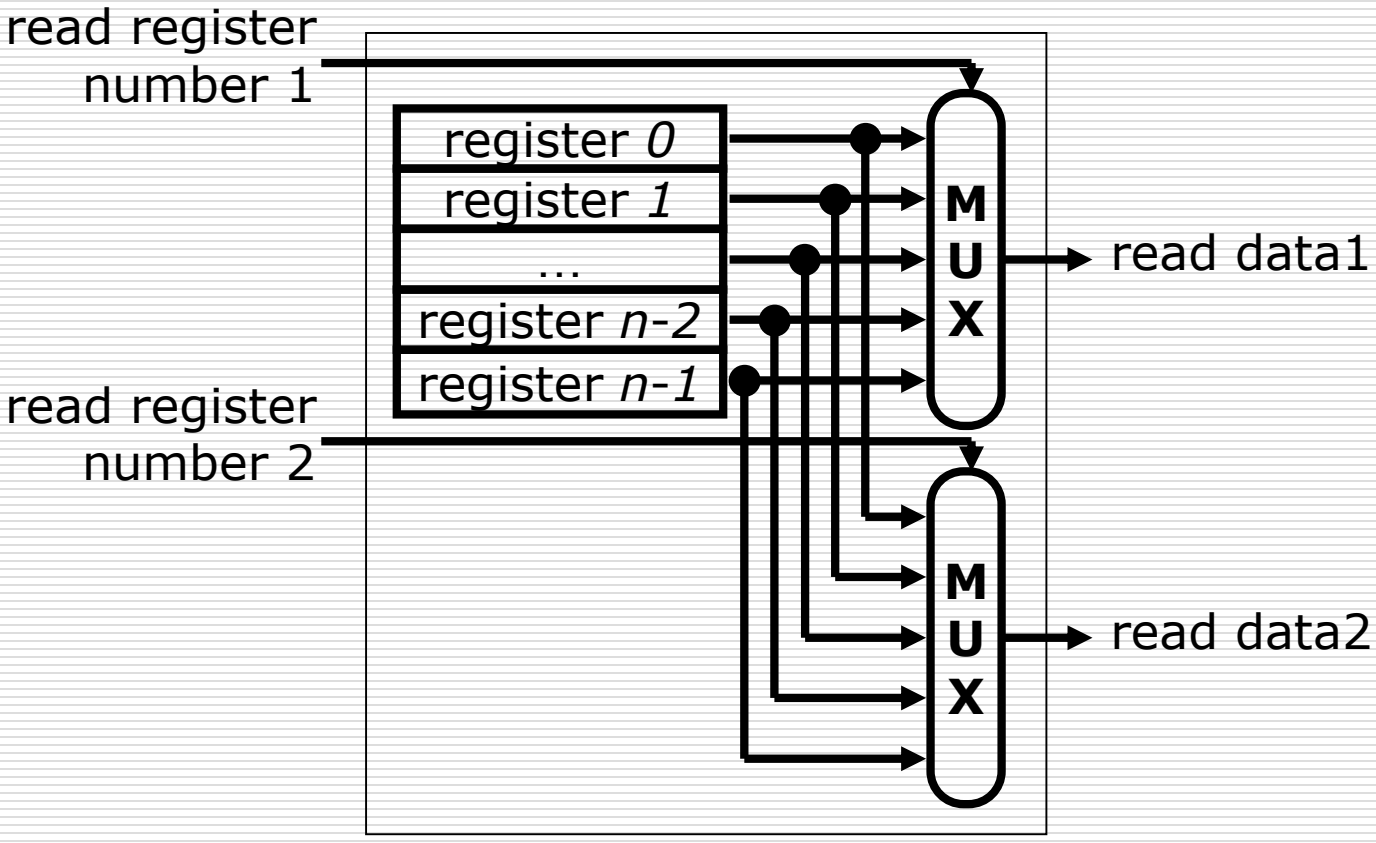


Register File

- built using D flip-flops

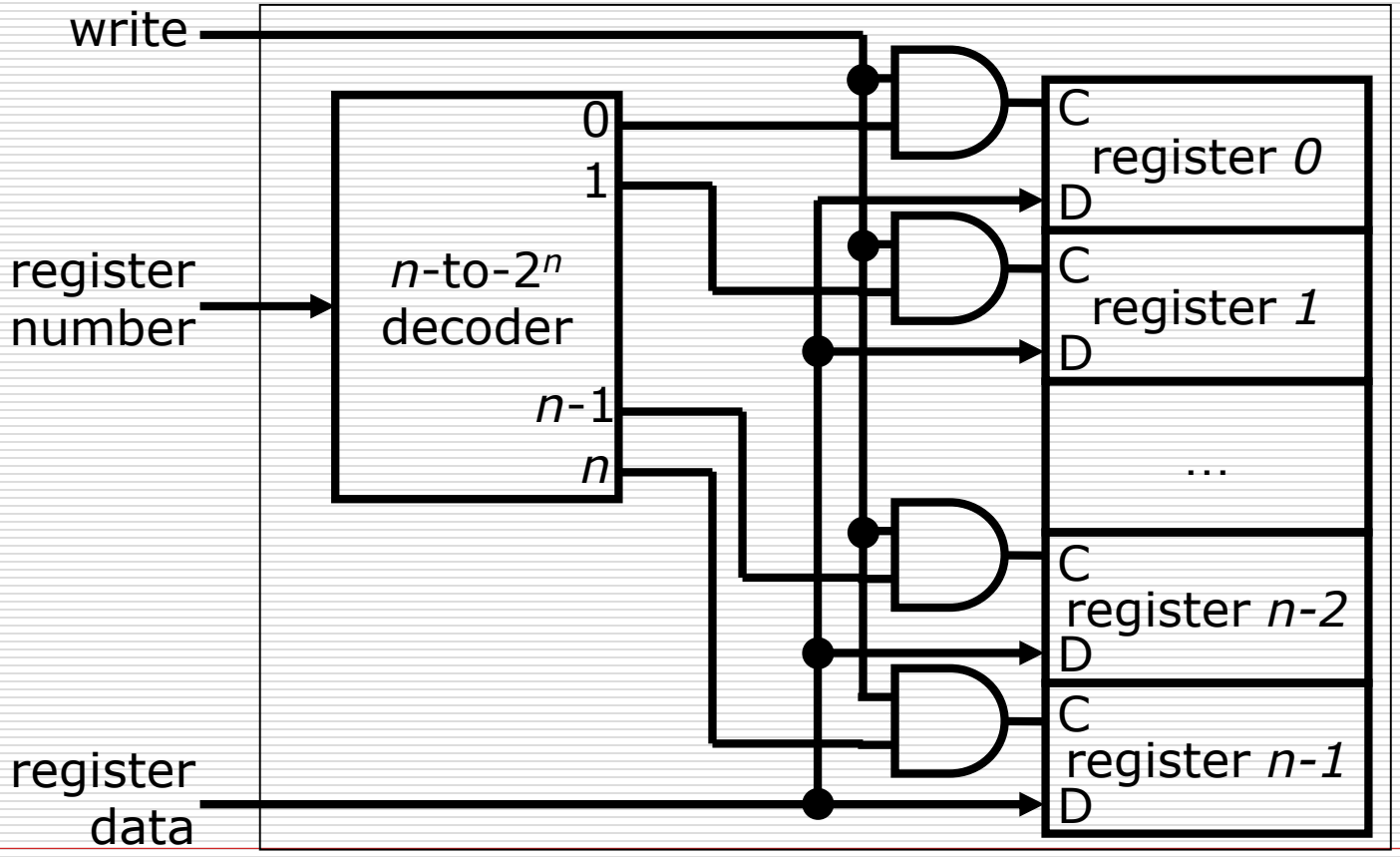


Register File

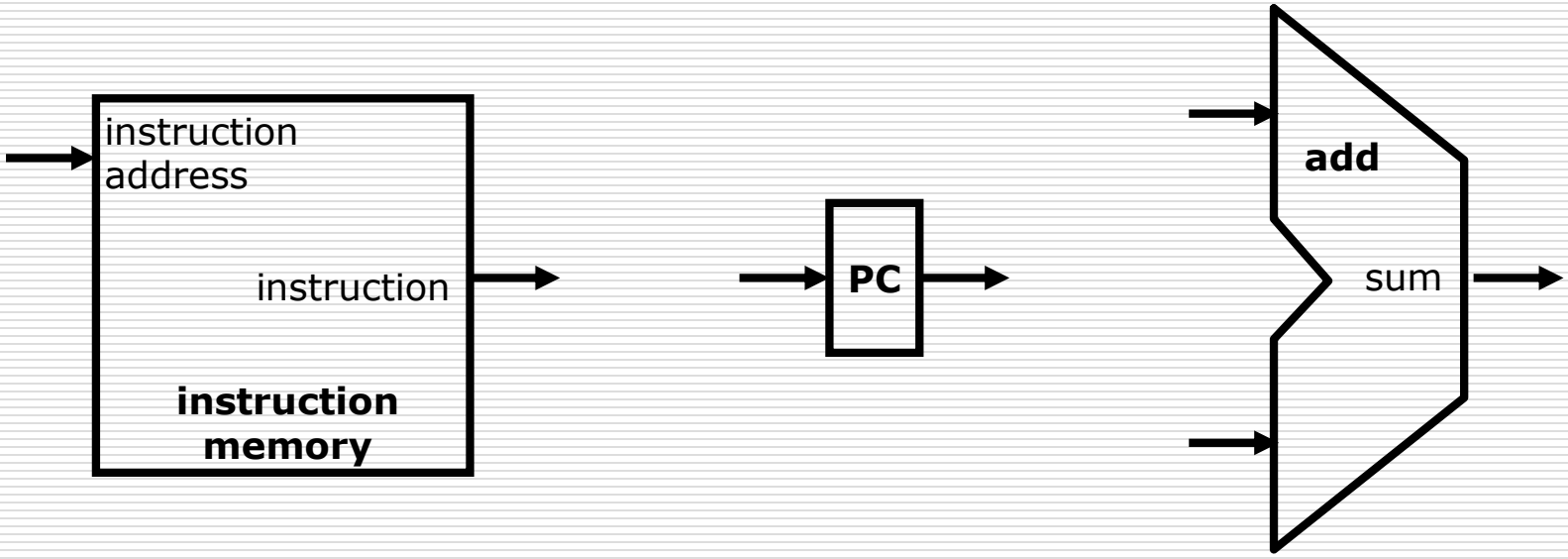


Register File

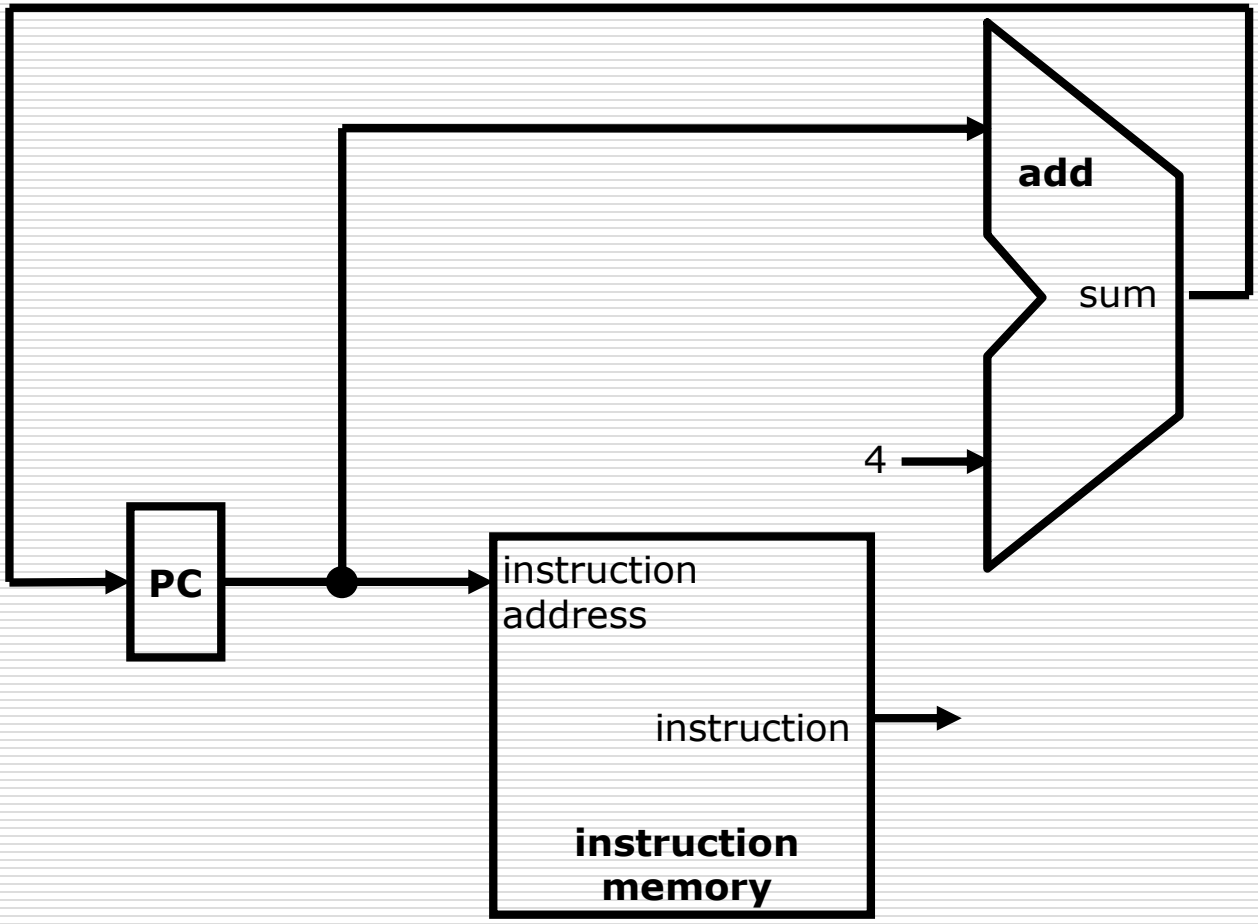
- Note: we still use the real clock to determine when to write



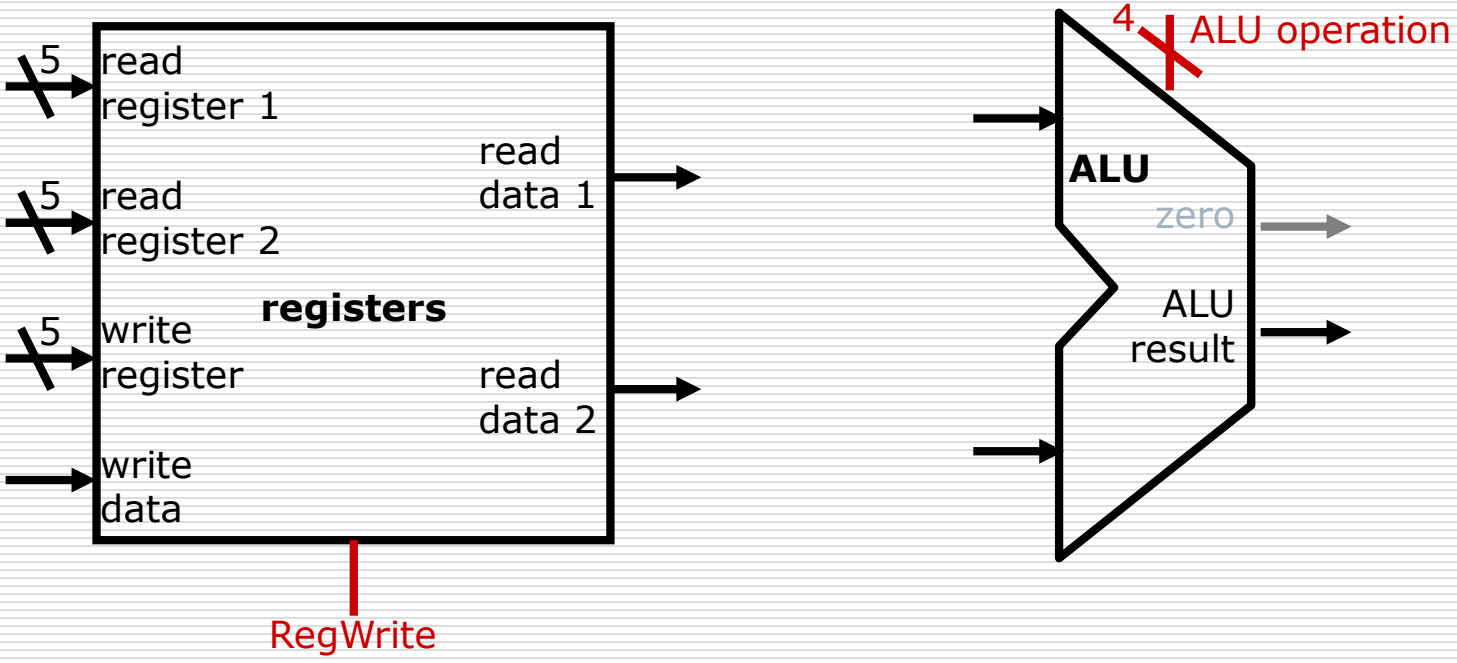
Instruction Fetch



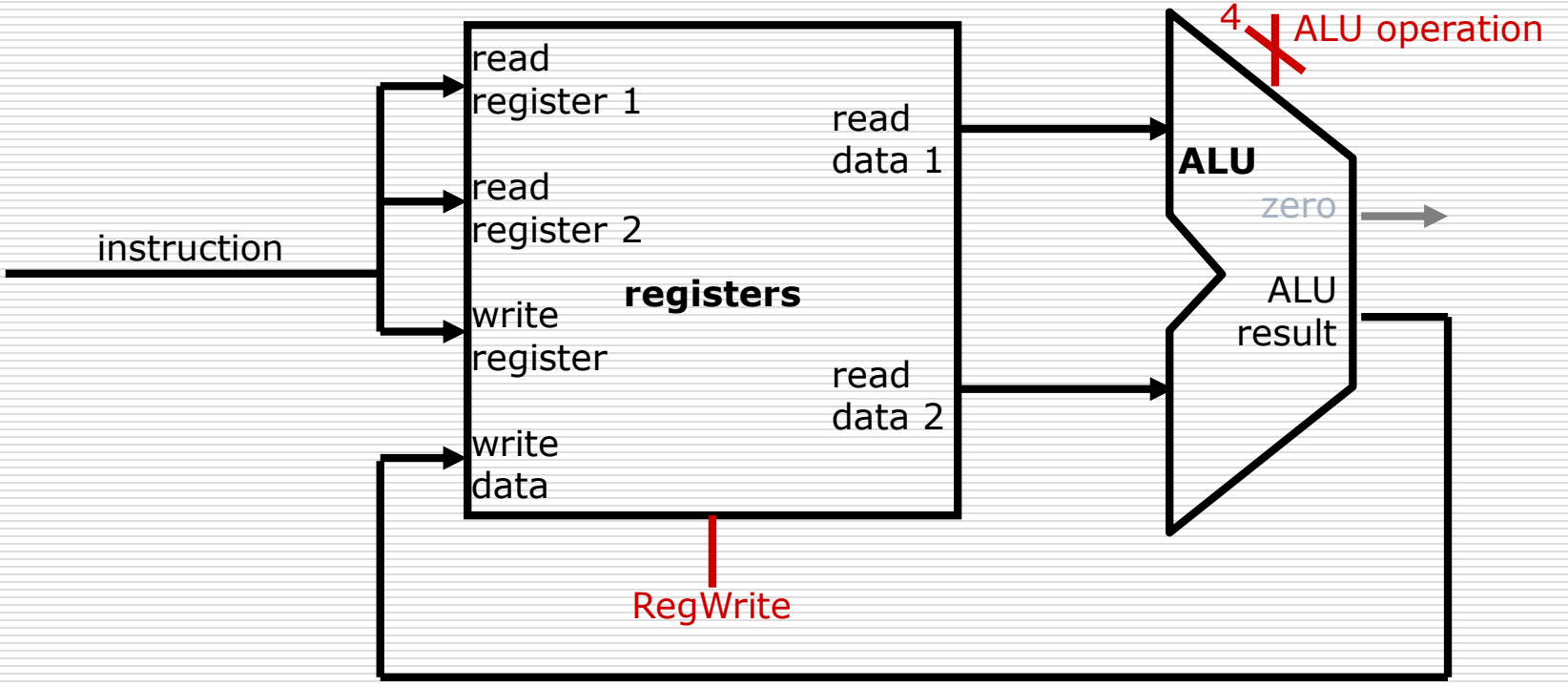
Instruction Fetch



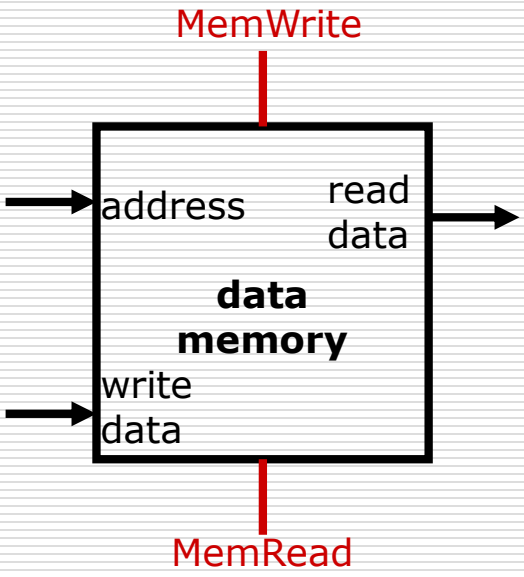
R-Format Instructions

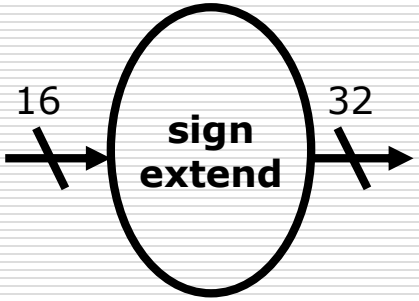


R-Format Instructions

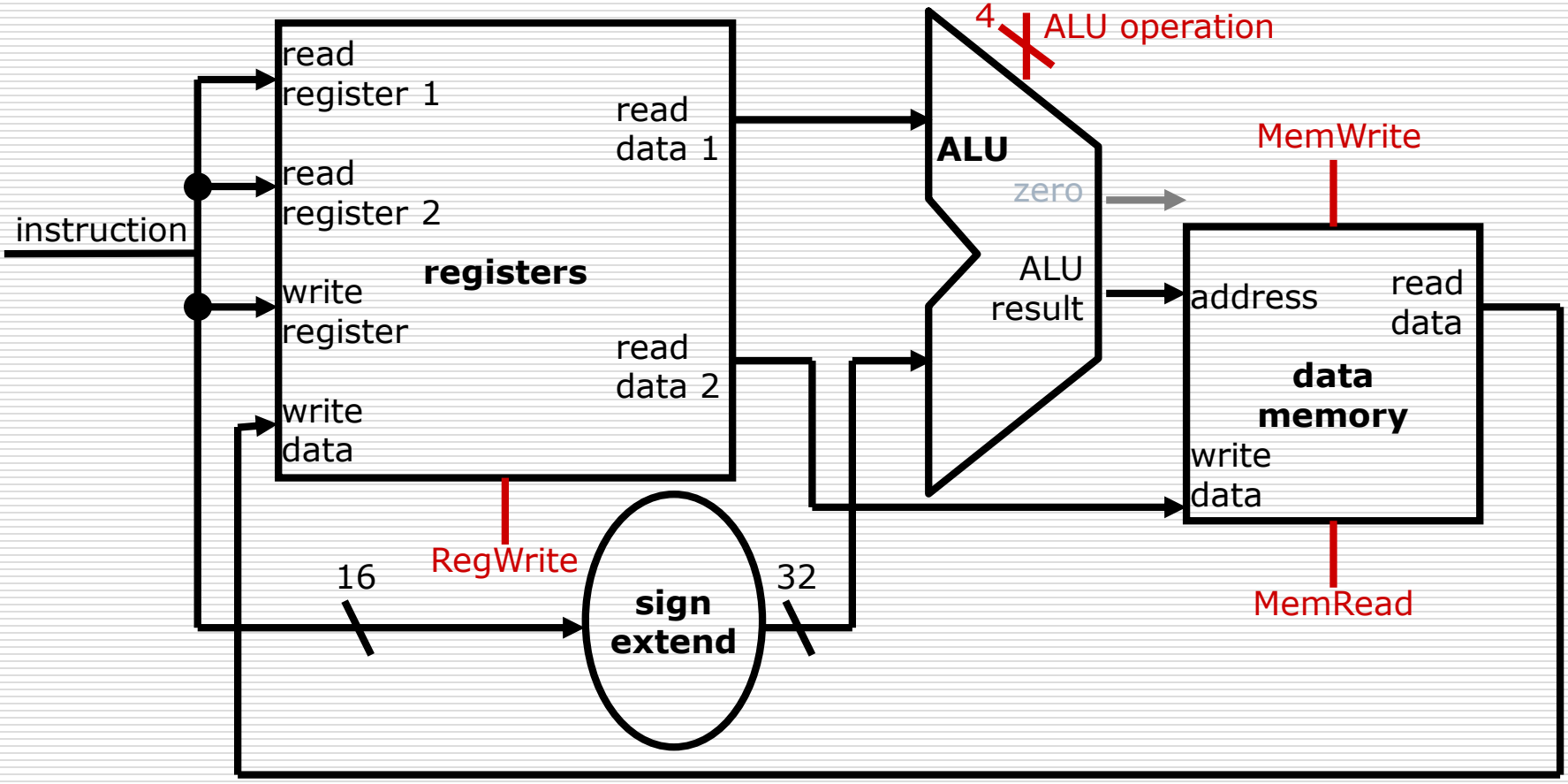


Load/Store Instructions

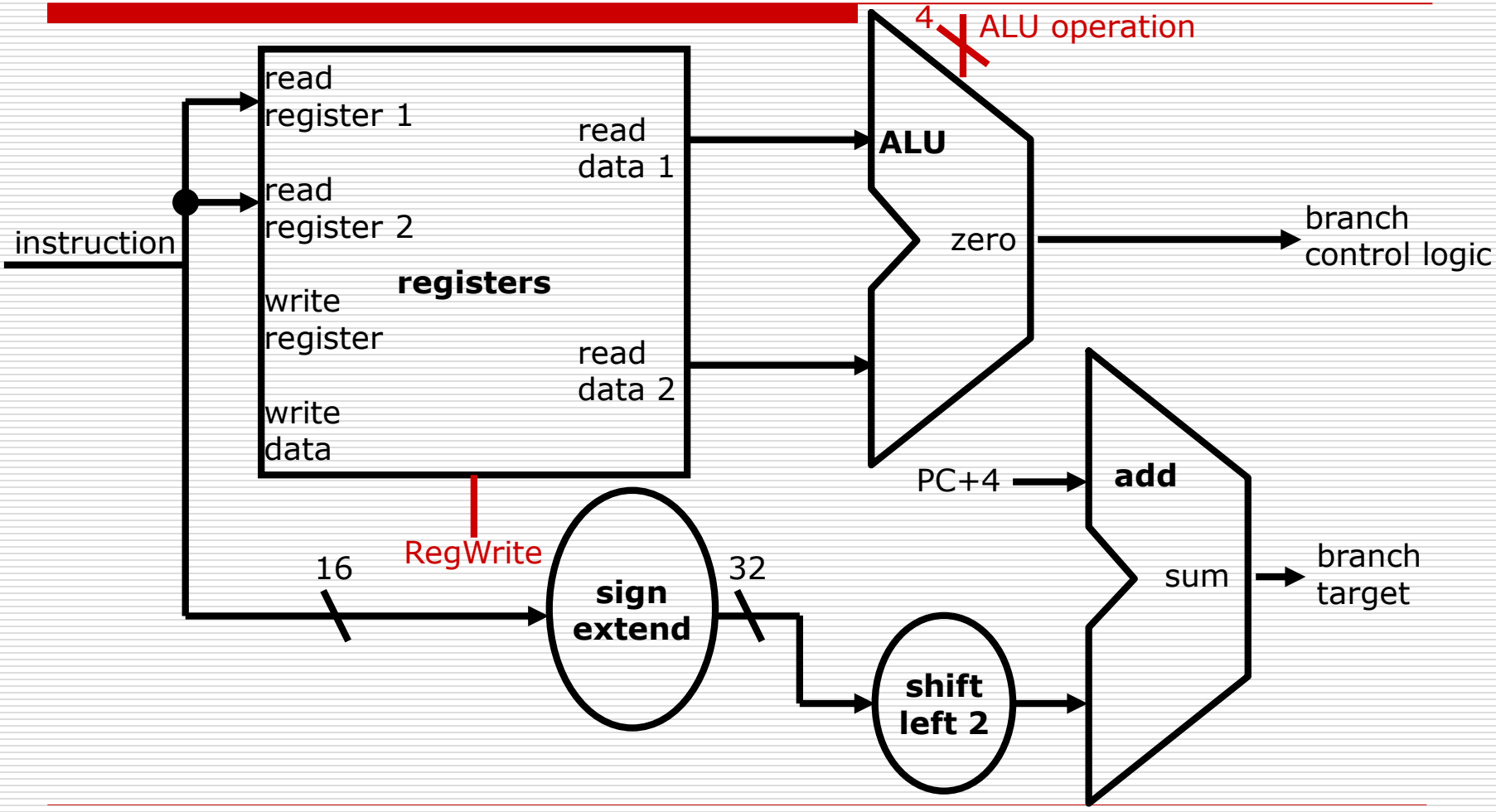




Load/Store Instructions



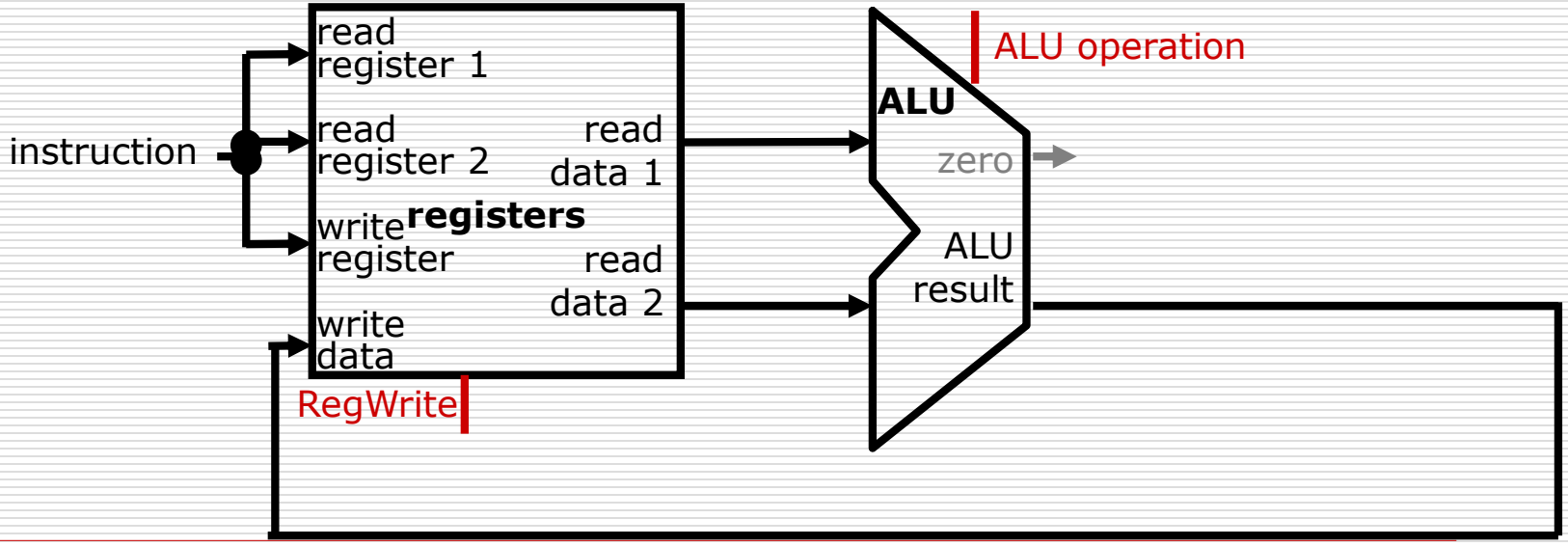
Branch Instructions



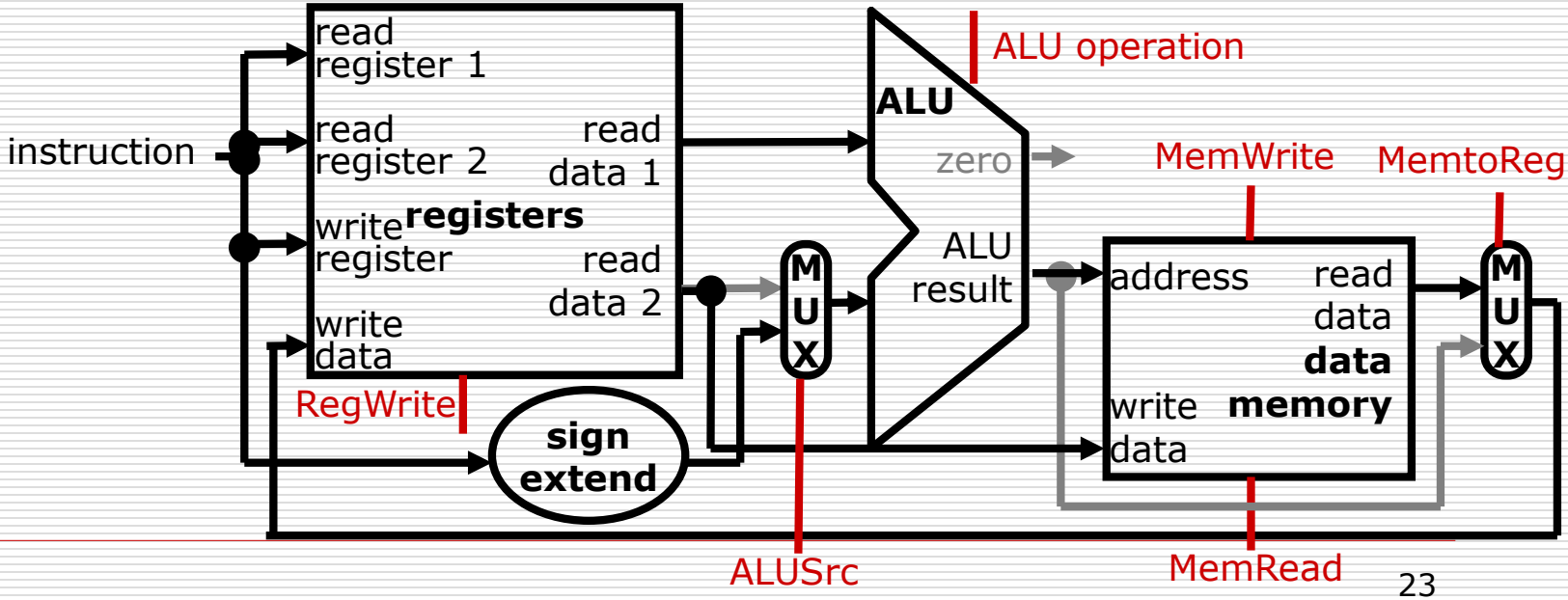
Composing the Elements

- First-cut data path does an instruction in one clock cycle
  - Each datapath element can only do one function at a time
  - Hence, we need separate instruction and data memories
- Use multiplexers where alternate data sources are used for different instructions

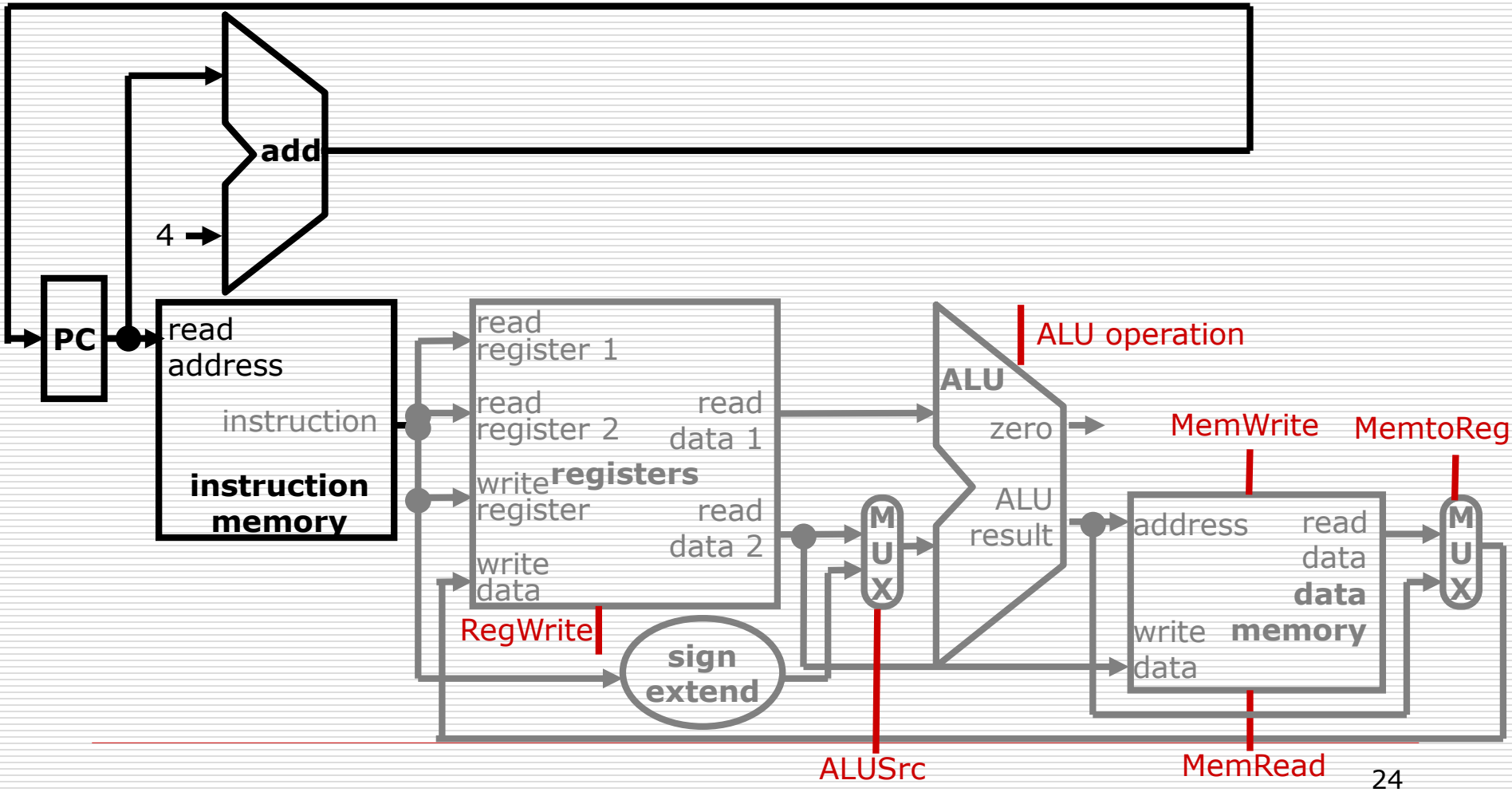
Building the Datapath



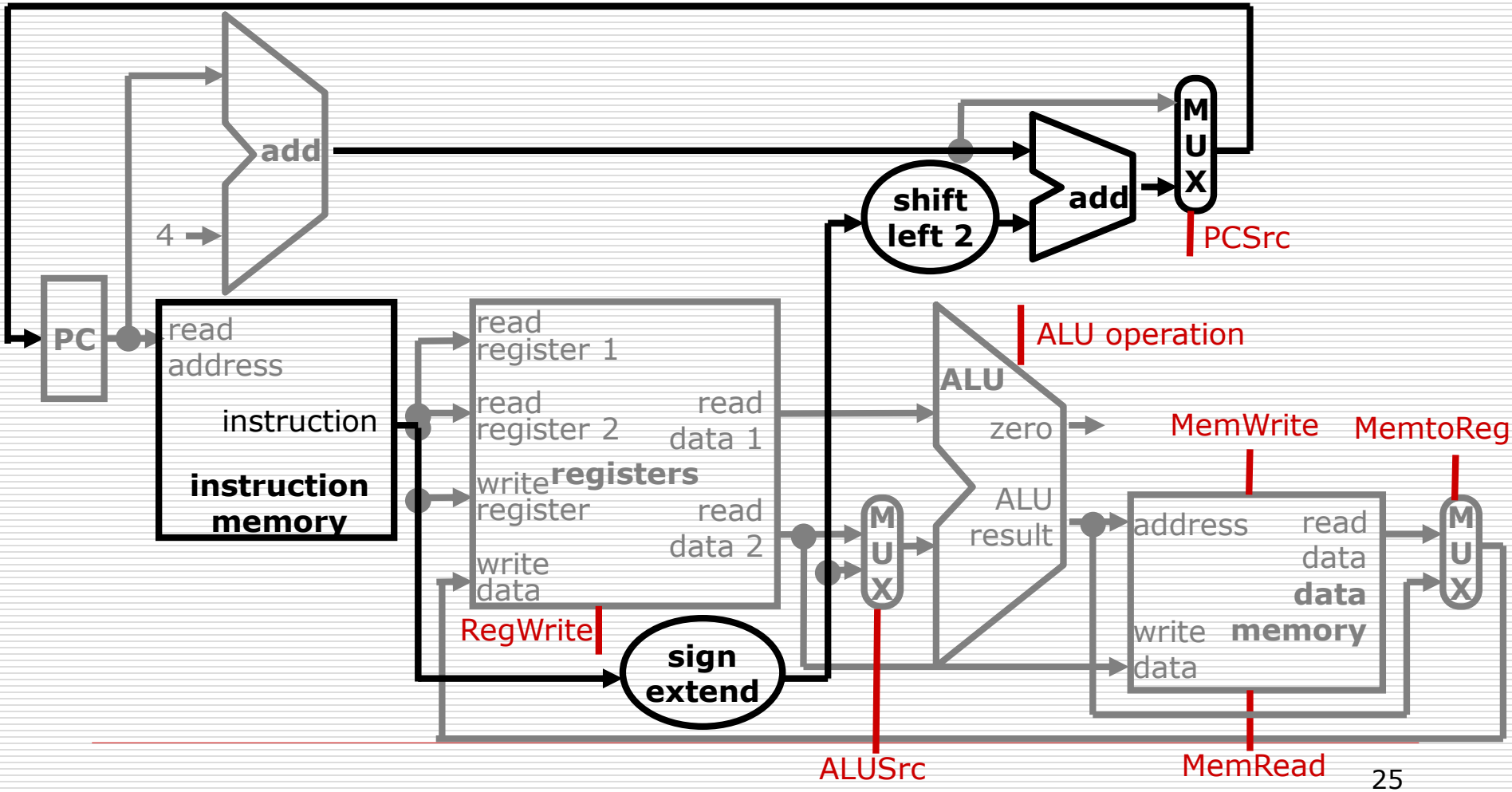
Building the Datapath



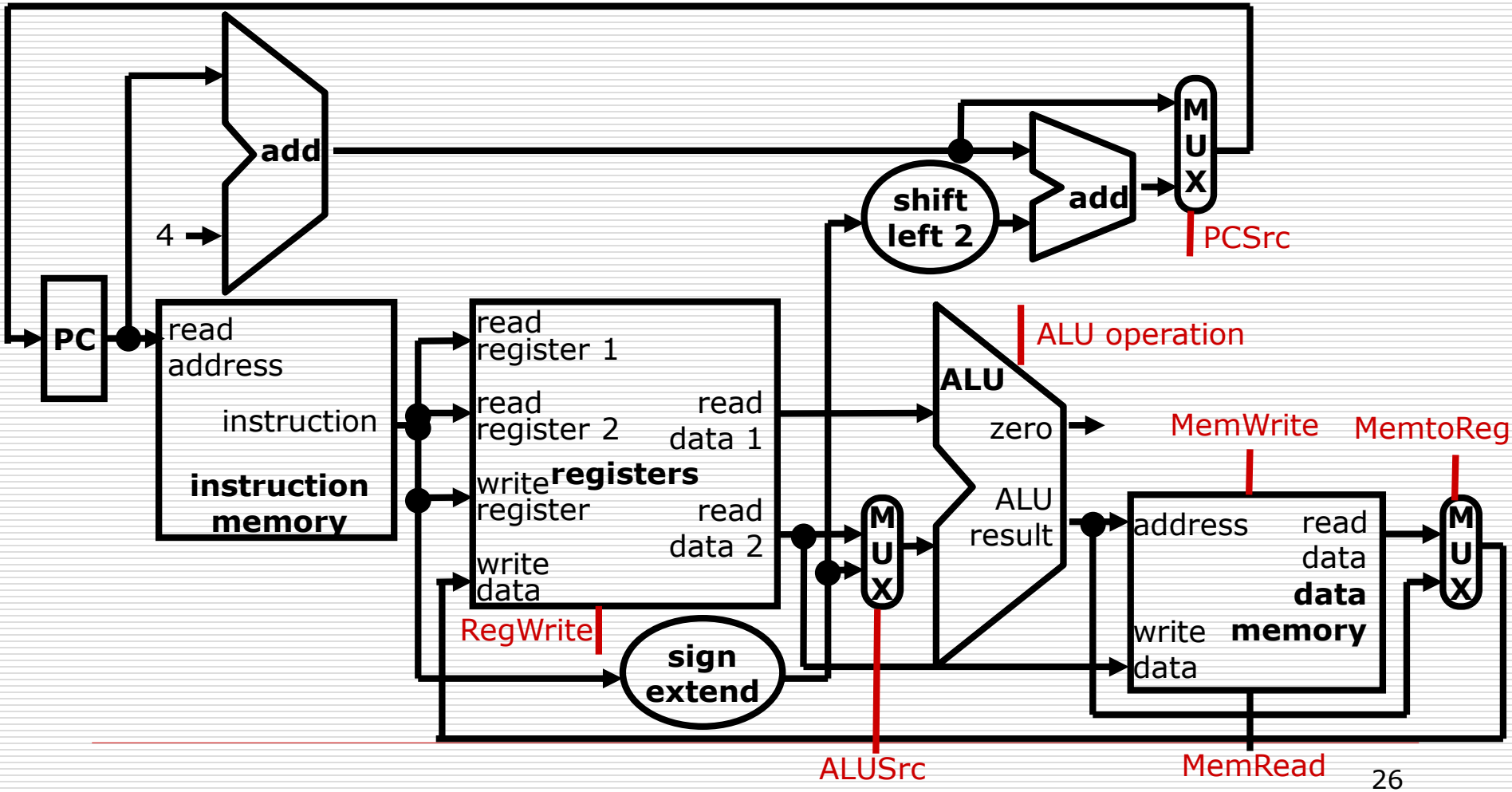
Building the Datapath



Building the Datapath



Building the Datapath



Control

- Selecting the operations to perform (ALU, r/w)
- Controlling the flow of data (multiplexor inputs)
- Information comes from the 32 bits of instruction

add $8, $17, $18

000000	10001	10010	01000	00000	100000
op	**rs**	**rt**	**rd**	**shamt**	**funct**

- ALU's operation based on instruction type and function code

ALU Control

- what should the ALU do with this instruction

  lw $1, 100($2)

35	2	1	100
op	**rs**	**rt**	**16 bit number**

- ALU control input

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

- why is the code for subtract 110 and not 011?

ALU Control

- must describe hardware to compute 4-bit ALU control input
  - given instruction type

    00 = lw, sw
    01 = beq,
    10 = arithmetic

    ALUOp computed from instruction type
  - function code for arithmetic
- describe it using a truth table (can turn into gates):

ALU Control

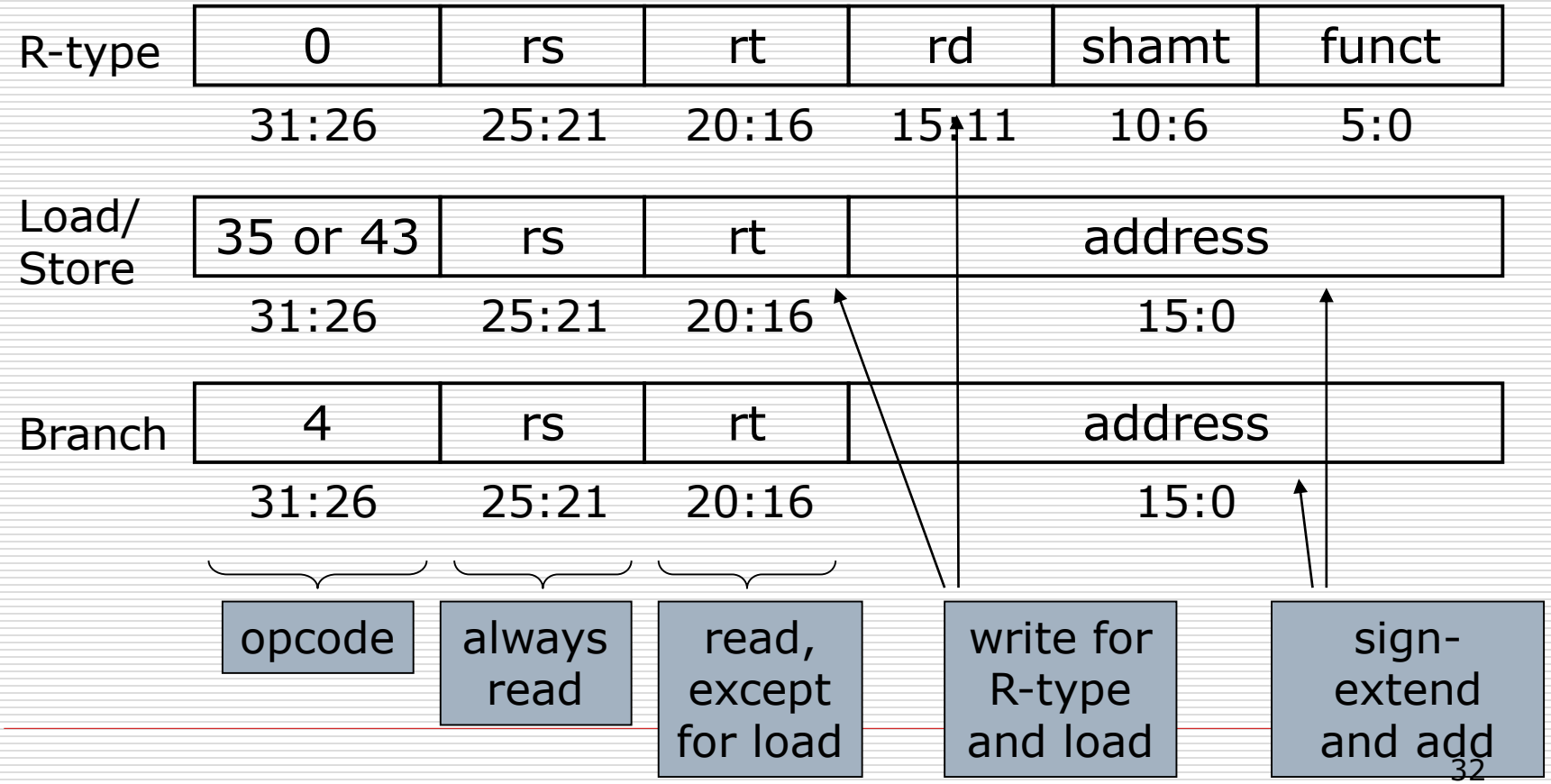
instruction opcode	ALUOp	Funct field	desired ALU action	ALU control input
LW	00	XXXXXX	add	0010
SW	00	XXXXXX	add	0010
Branch equal	01	XXXXXX	subtract	0110
R-type	10	100000	add	0010
R-type	10	100010	subtract	0110
R-type	10	100100	and	0000
R-type	10	100101	or	0001
R-type	10	101010	set on less than	0111

ALU Control

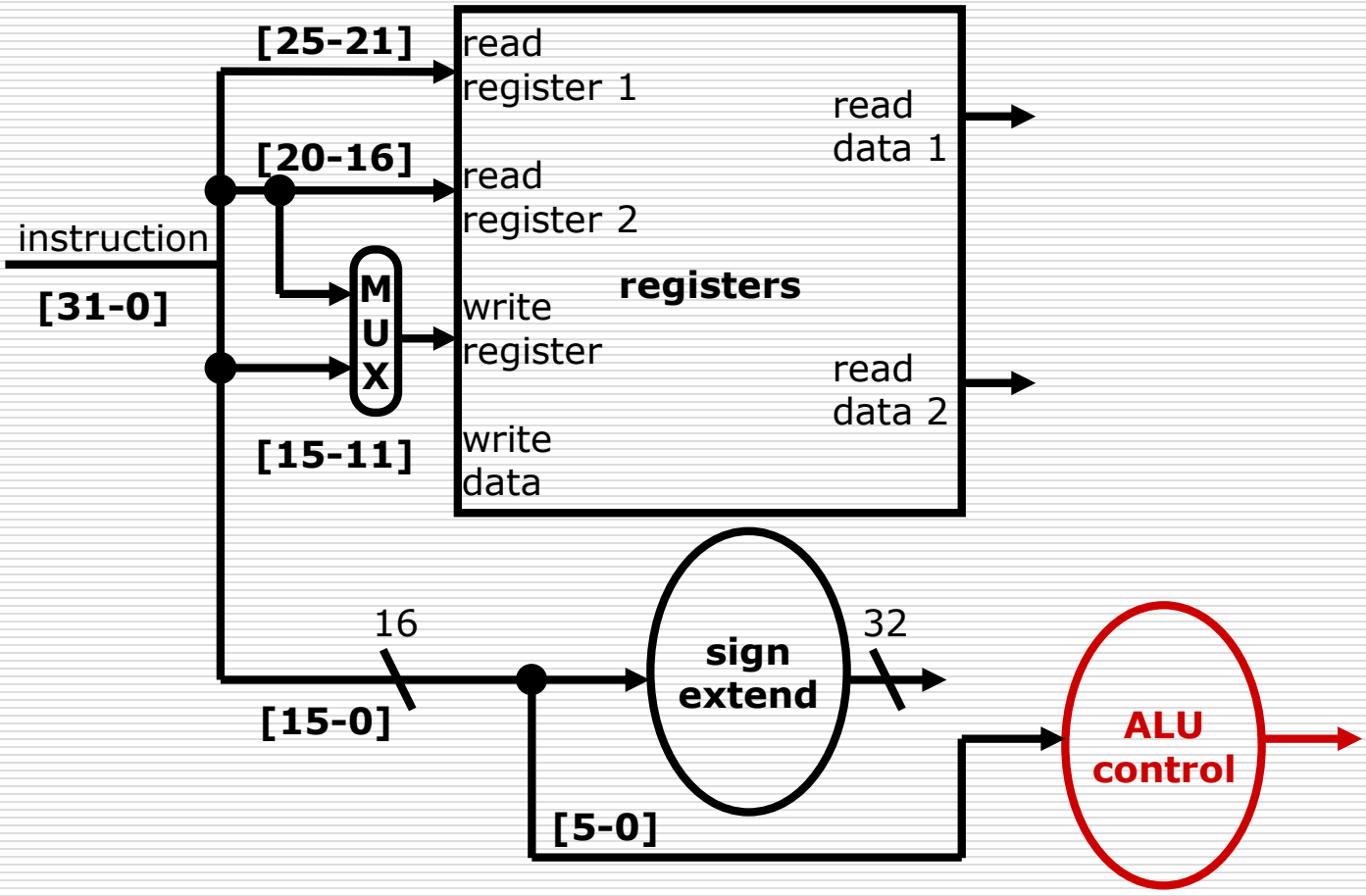
ALUOp		Funct field						operation
ALUOp1	ALUOp2	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	0110
1	X	X	X	0	0	0	0	0010
1	X	X	X	0	0	1	0	0110
1	X	X	X	0	1	0	0	0000
1	X	X	X	0	1	0	1	0001
1	X	X	X	1	0	1	0	0111

The Main Control Unit

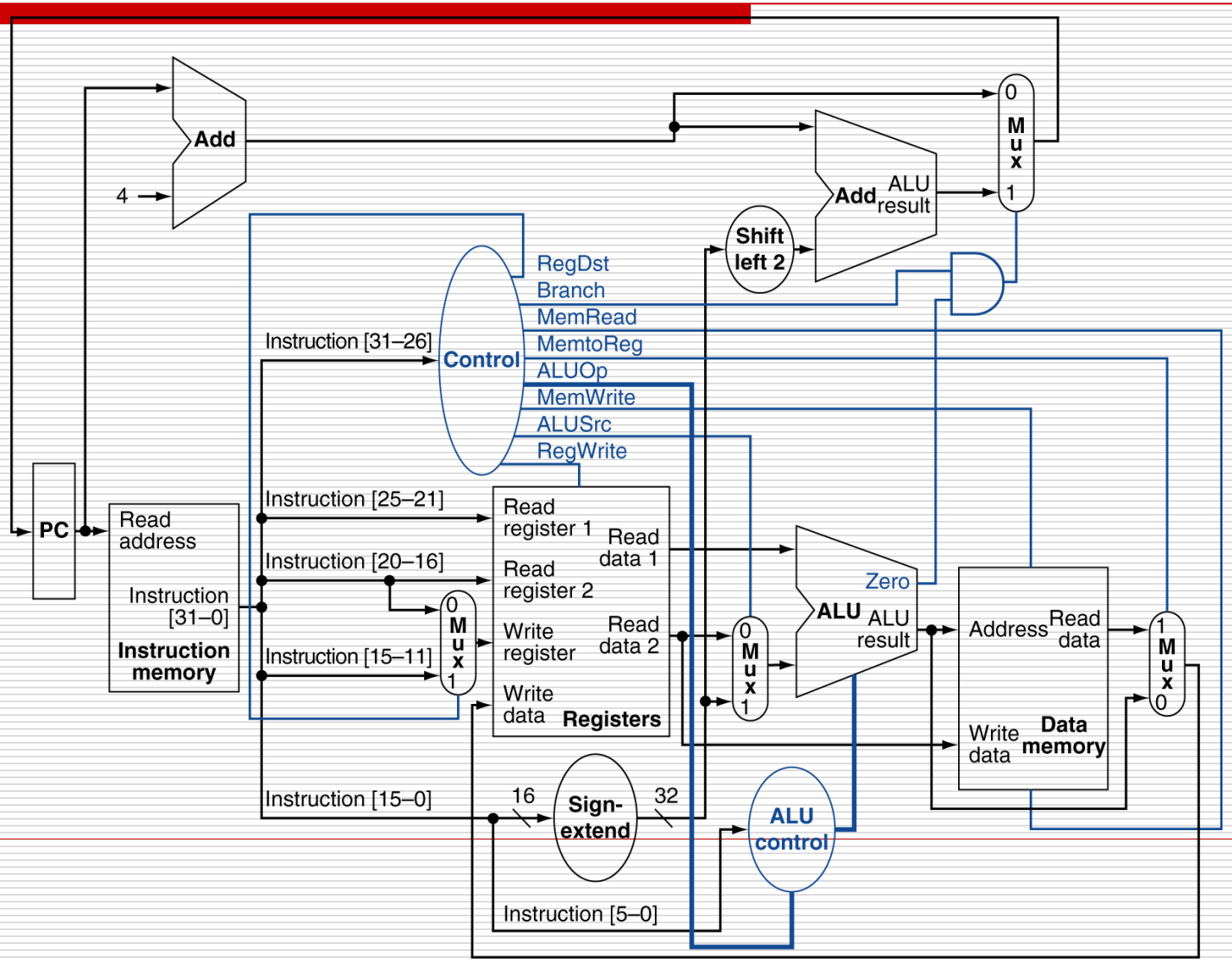
- Control signals derived from instruction



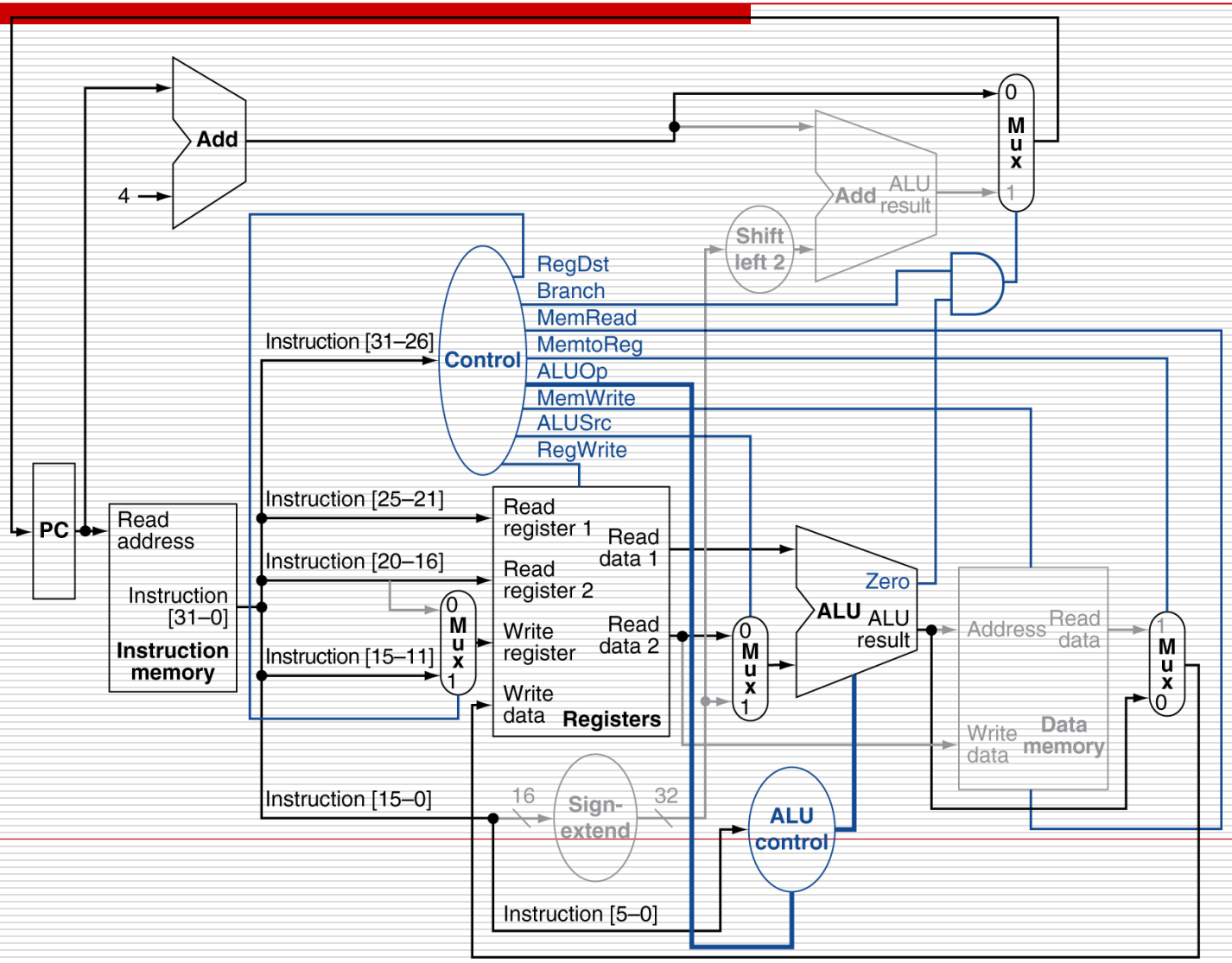
Building the Control



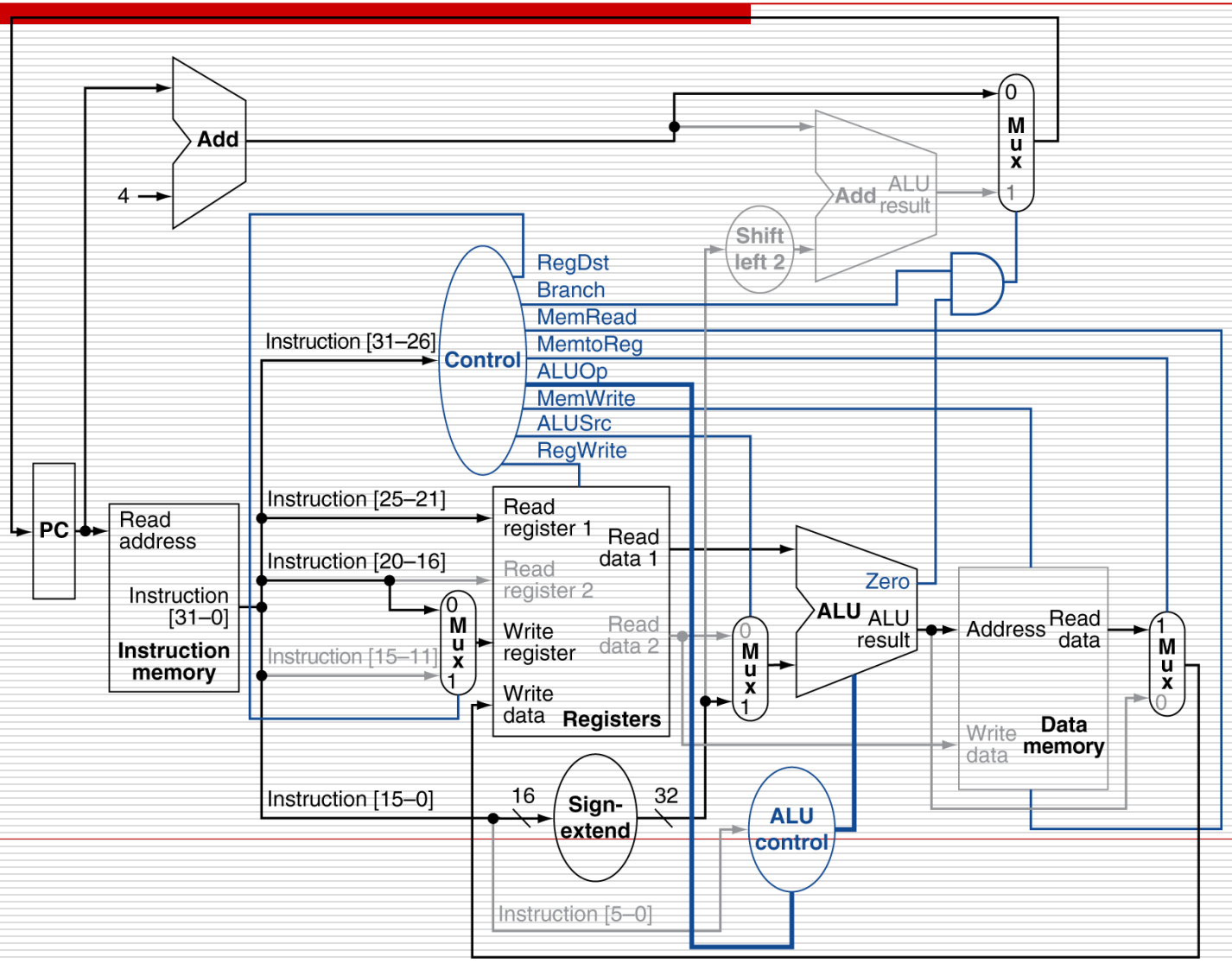
Datapath With Control



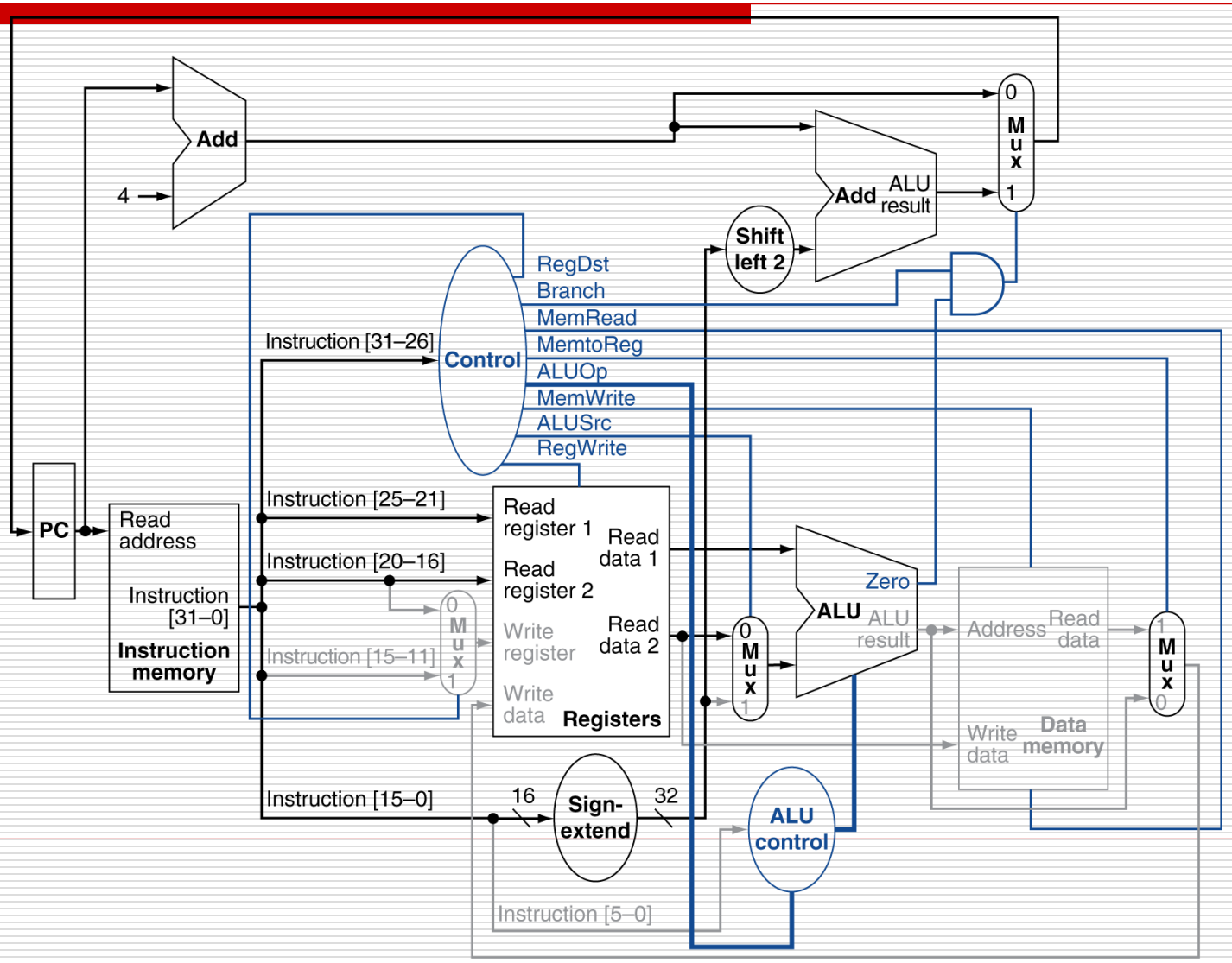
R-Type Instruction



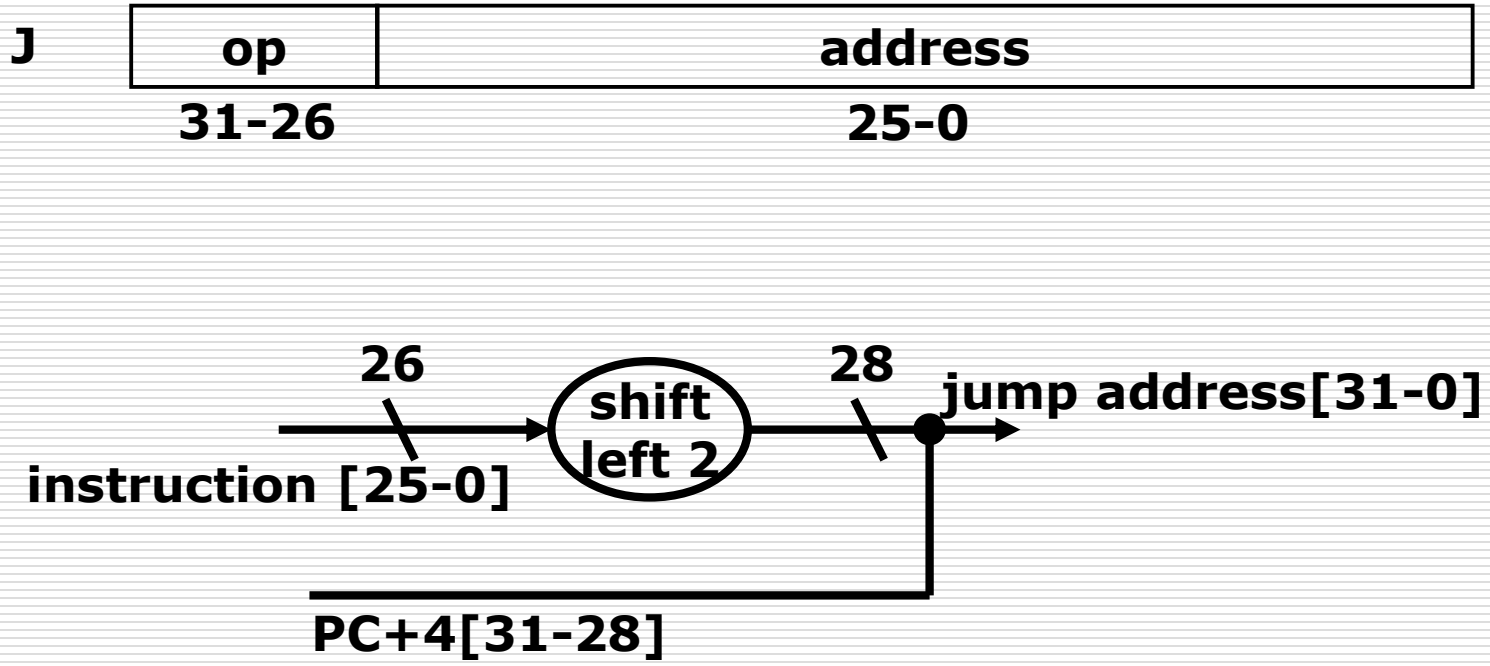
Load Instruction



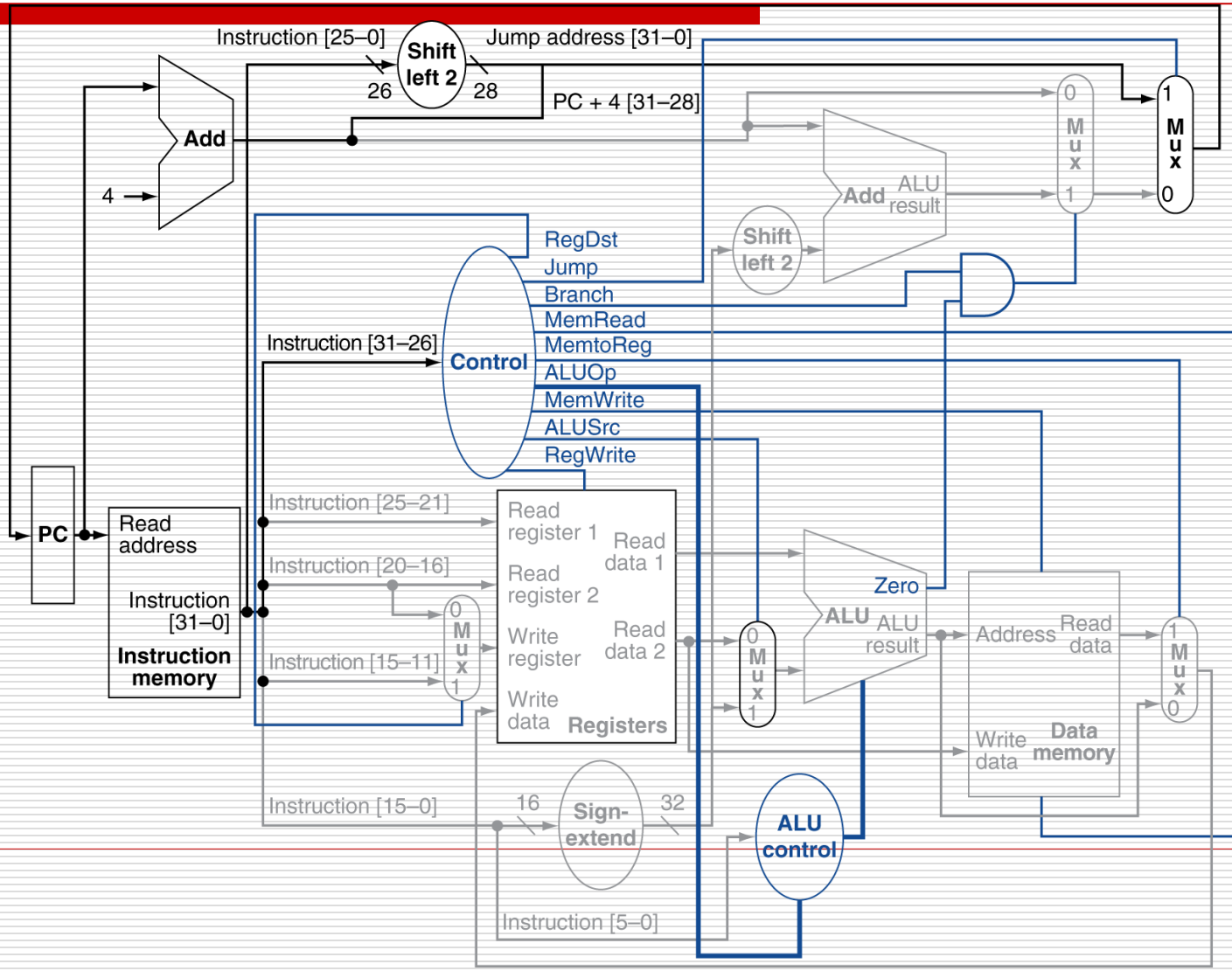
Branch-on-Equal Instruction



Implementing Jumps



Datapath With Jumps Added



Control

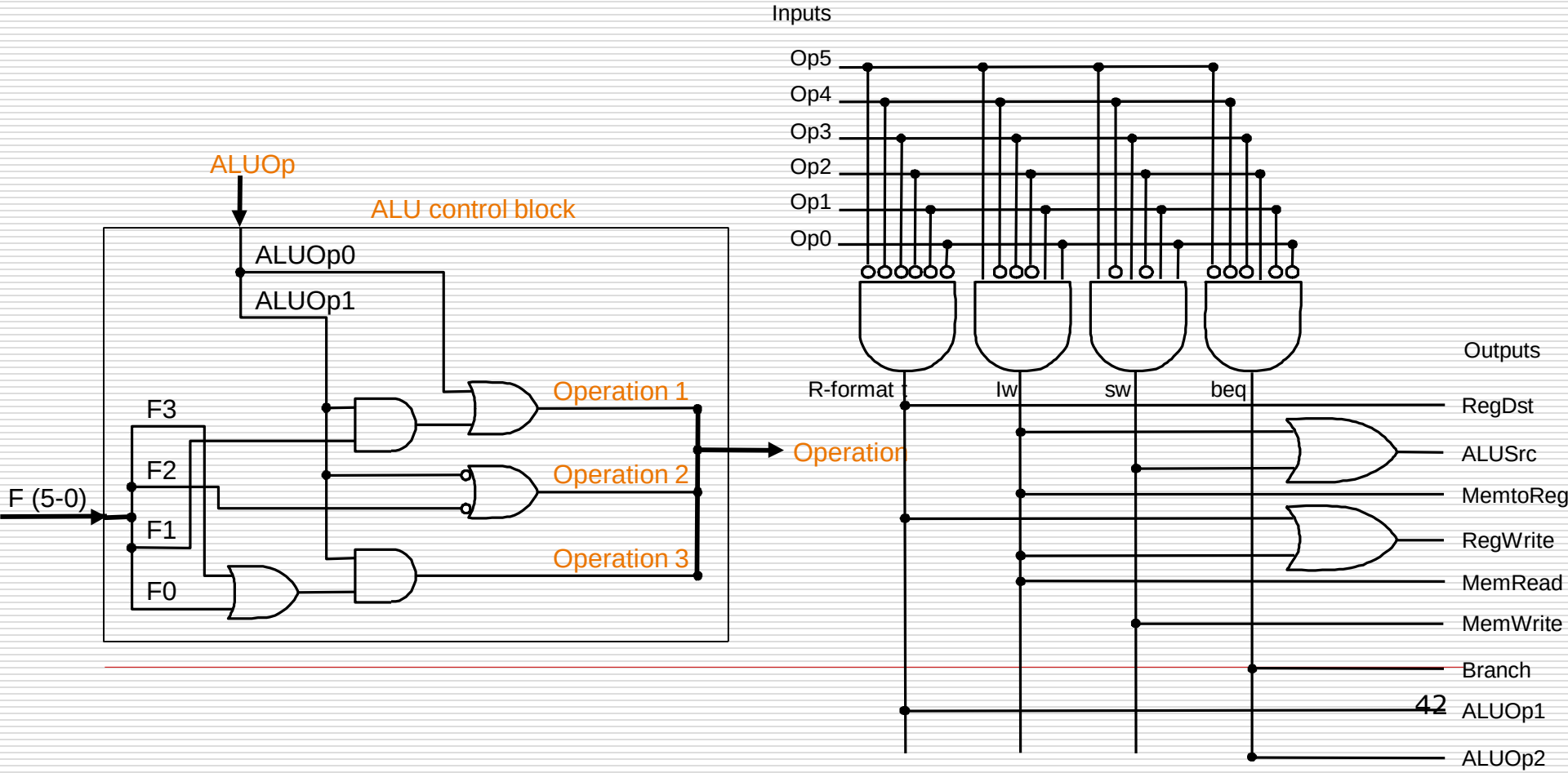
instruc-tion	Reg Dst	ALU Src	Mem to Reg	Reg Write	Mem Read	Mem Write	Branch	ALU Op1	ALU Op2
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

Control

	signal	R-format	lw	sw	beq
inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
outputs	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

Control

- Simple combinational logic (truth tables)



Performance Issues

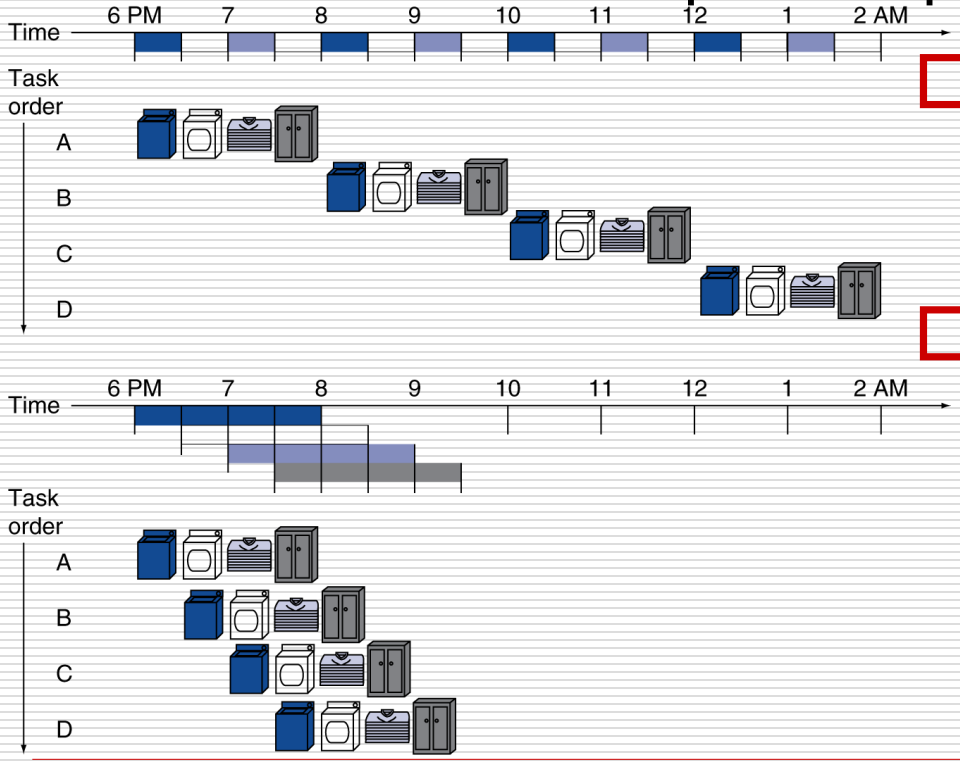
- Longest delay determines clock period
  - Critical path: load instruction
  - Instruction memory $\rightarrow$ register file $\rightarrow$ ALU $\rightarrow$ data memory $\rightarrow$ register file
- Not feasible to vary period for different instructions
- Violates design principle
  - Making the common case fast
- We will improve performance by pipelining

Performance of Single-Cycle Machines

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

Pipelining Analogy

- Pipelined laundry: overlapping execution
  - Parallelism improves performance

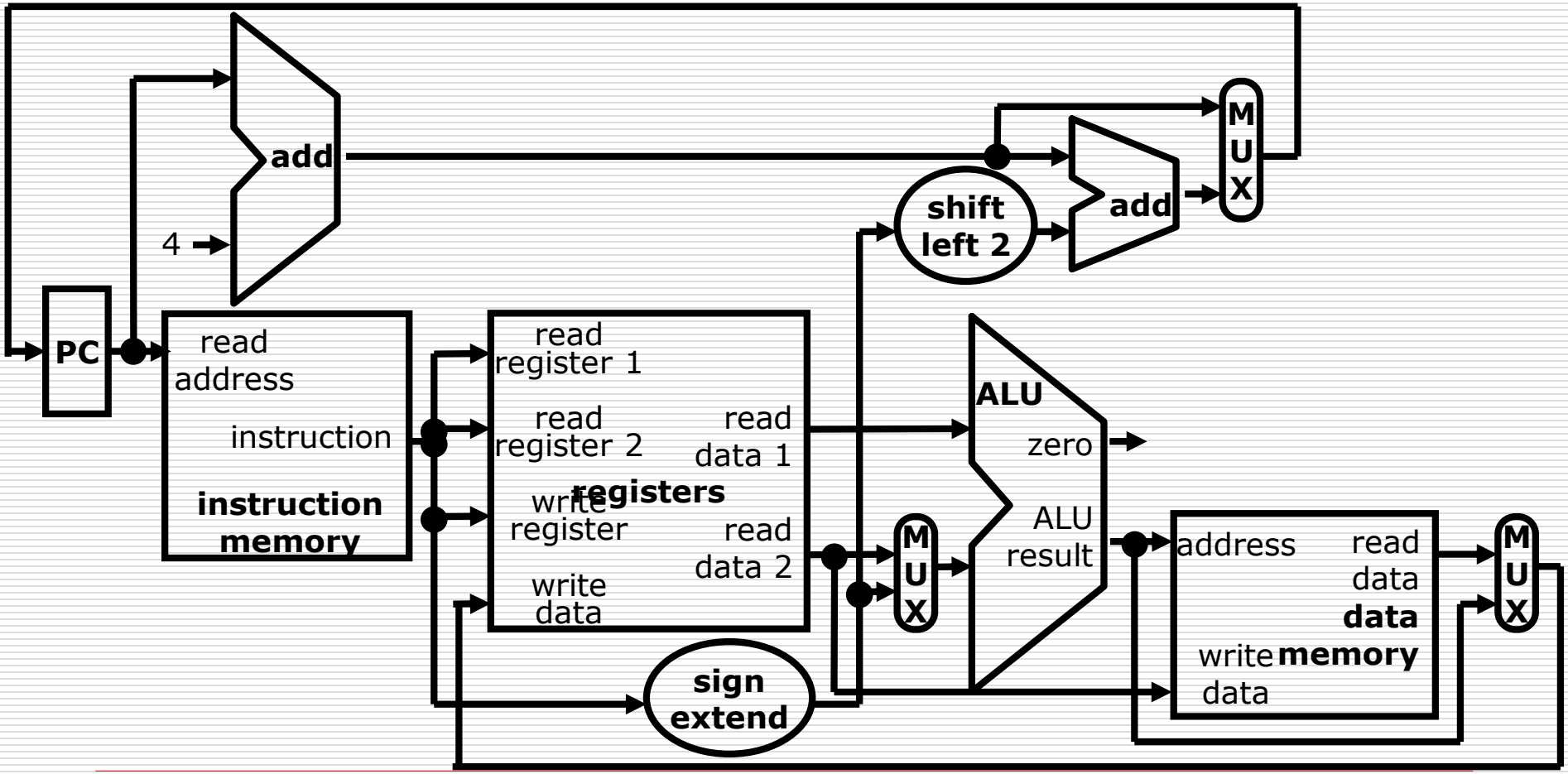


- Four loads:
  - Speedup = 8/3.5 = 2.3
- Non-stop:
  - Speedup = 2n/(0.5n+1.5) ≈ 4 = number of stages

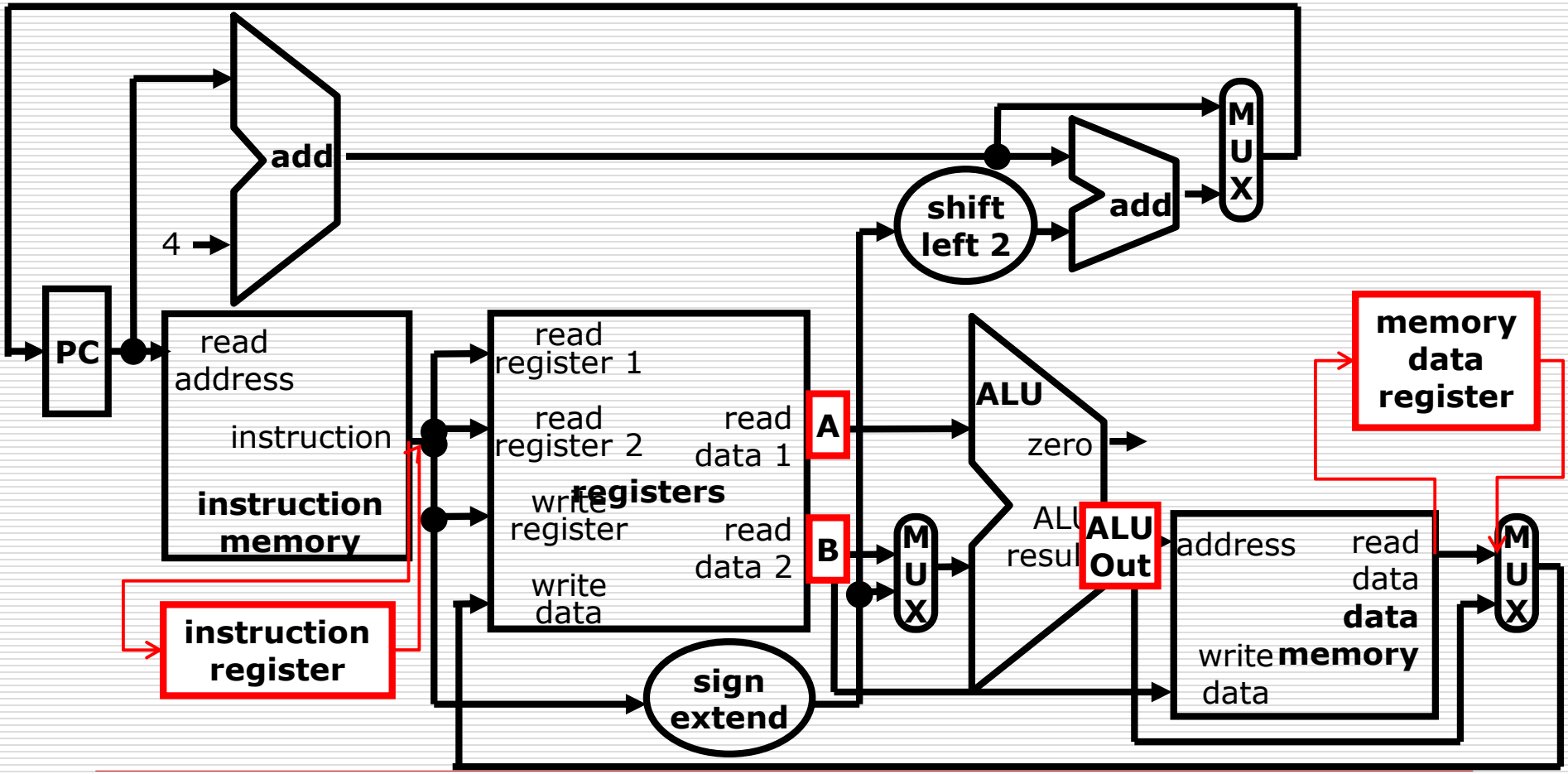
MIPS Pipeline

- Five stages, one step per stage
  1. IF: Instruction fetch from memory
  2. ID: Instruction decode & register read
  3. EX: Execute operation or calculate address
  4. MEM: Access memory operand
  5. WB: Write result back to register

Recall:
Single Cycle Implementation



Toward Pipeline Implementation



Step 1: Instruction Fetch

- use PC to get instruction and put it in the Instruction Register.
- increment the PC by 4 and put the result back in the PC.
- can be described succinctly using RTL "Register-Transfer Language"

```
IR = Memory[PC];
PC = PC + 4;
```

Can we figure out the values of the control signals?
What is the advantage of updating the PC now?

Step 2: Instruction Decode & Register Fetch

- read registers rs and rt in case we need them
- compute the branch address in case the instruction is a branch
- RTL:

```
A = Reg[IR[25-21]];
B = Reg[IR[20-16]];
ALUOut = PC + (sign-extend(IR[15-0]) << 2);
```

- We aren't setting any control lines based on the instruction type
  - we are busy "decoding" it in our control logic

Step 3: Execution Step (instruction dependent)

- ALU is performing one of three functions, based on instruction type
- Memory Reference:
  ALUOut = A + sign-extend(IR[15-0]);
- R-type:
  ALUOut = A op B;
- Branch:
  if (A==B) PC = ALUOut;
- Jump:
  PC = PC[31-28] || (IR[25-0] << 2)

Step 4: Memory-Access

- Loads and stores access memory

```
MDR = Memory[ALUOut];
        or
Memory[ALUOut] = B;
```

Step 5: Write-back Step (R-Type or Loads)

- Loads access memory

  Reg[IR[20-16]]= MDR;

- R-type instructions finish

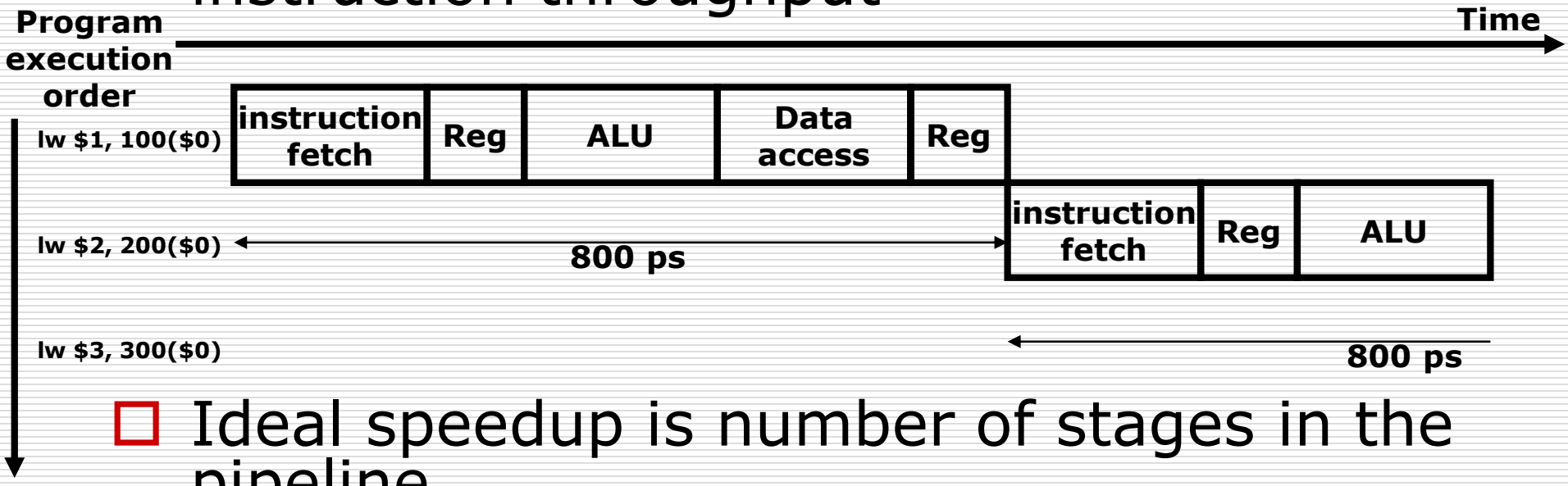
  Reg[IR[15-11]] = ALUOut;

Summary

step	R-type	memory reference	branch	jump
1	IR=Memory[PC] PC=PC+4			
2	A=Reg[IR[25-21]] B=Reg[IR[20-16]] ALUOut=PC+(sign-extend(IR[15-0])<<2)			
3	ALUOut=A op B	ALUOut=A+sign-extend (IR[15-0])	if (A==B) PC=ALUOut	PC=PC[31-28]|| (IR[25-0]<<2)
4		lw:MDR=Memory[ALUOut] or sw:Memory[ALUOut]=B		
5	Reg[IR[15-11]]= ALUOut	lw:Reg[IR[20-16]]=MDR		

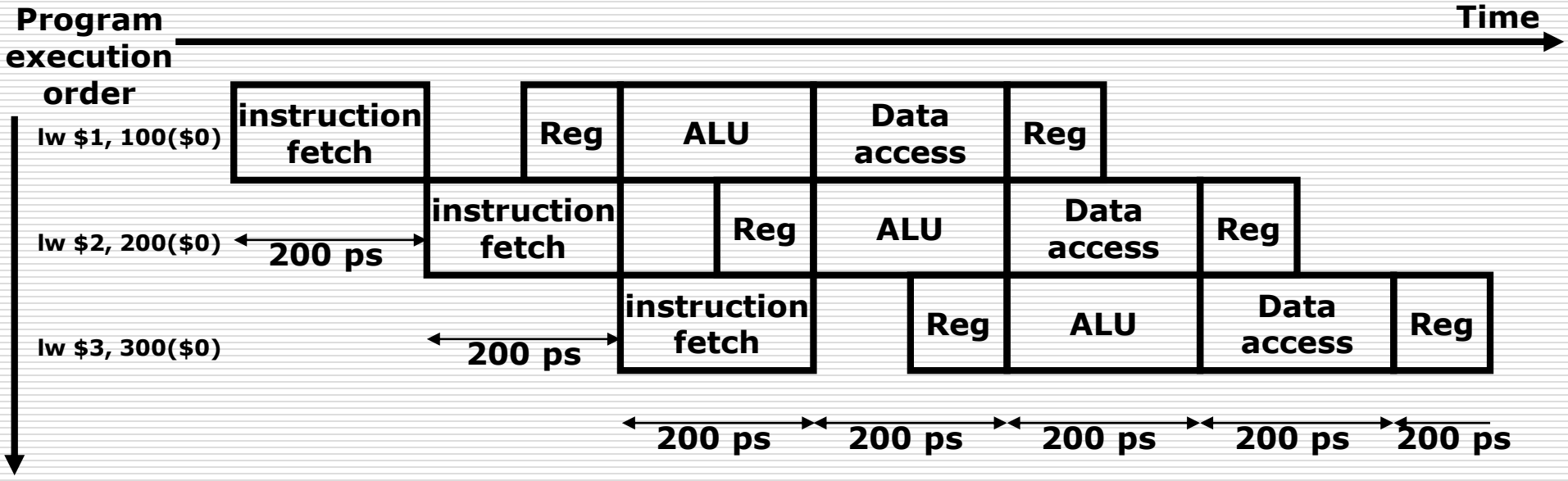
Non-Pipelining

- Improve performance by increasing instruction throughput



- Ideal speedup is number of stages in the pipeline.
- Do we achieve this?

Pipelining


Program execution order
Time
lw $1, 100($0)
lw $2, 200($0)
lw $3, 300($0)
instruction fetch
Reg
ALU
Data access
Reg
200 ps
200 ps
200 ps
200 ps
200 ps
200 ps

Pipeline Speedup

- If all stages are balanced
  - i.e., all take the same time
  - $\text{Time between instructions}_{\text{pipelined}} = \frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of stages}}$
- If not balanced, speedup is less
- Speedup due to increased throughput
  - Latency (time for each instruction) does not decrease

Hazards

- Situations that prevent starting the next instruction in the next cycle
- Structure hazards
  - A required resource is busy
- Data hazard
  - Need to wait for previous instruction to complete its data read/write
- Control hazard
  - Deciding on control action depends on previous instruction

Structure Hazards

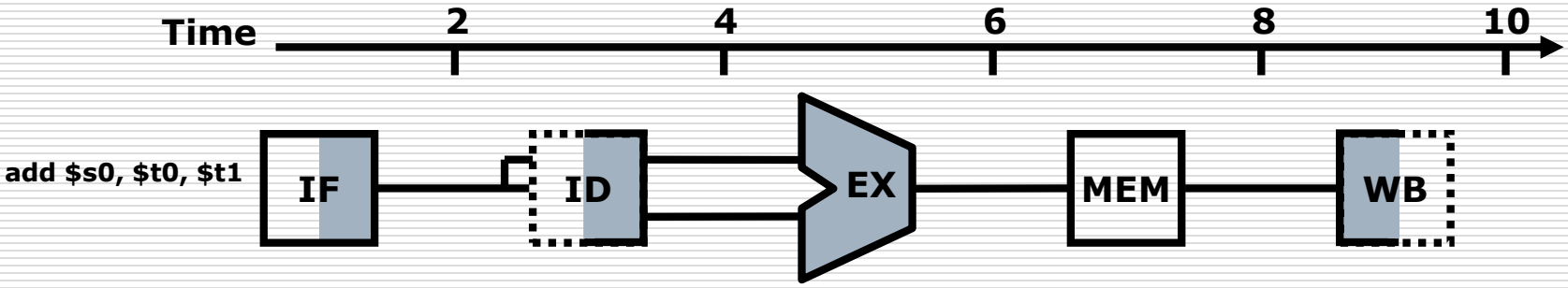
- Conflict for use of a resource
- In MIPS pipeline with a single memory
  - Load/store requires data access
  - Instruction fetch would have to stall for that cycle
    - Would cause a pipeline “bubble”
- Hence, pipelined datapaths require separate instruction/data memories
  - Or separate instruction/data caches

Data Hazards

- An instruction depends on completion of data access by a previous instruction
  - add $s0, $t0, $t1
  - sub $t2, $s0, $t3

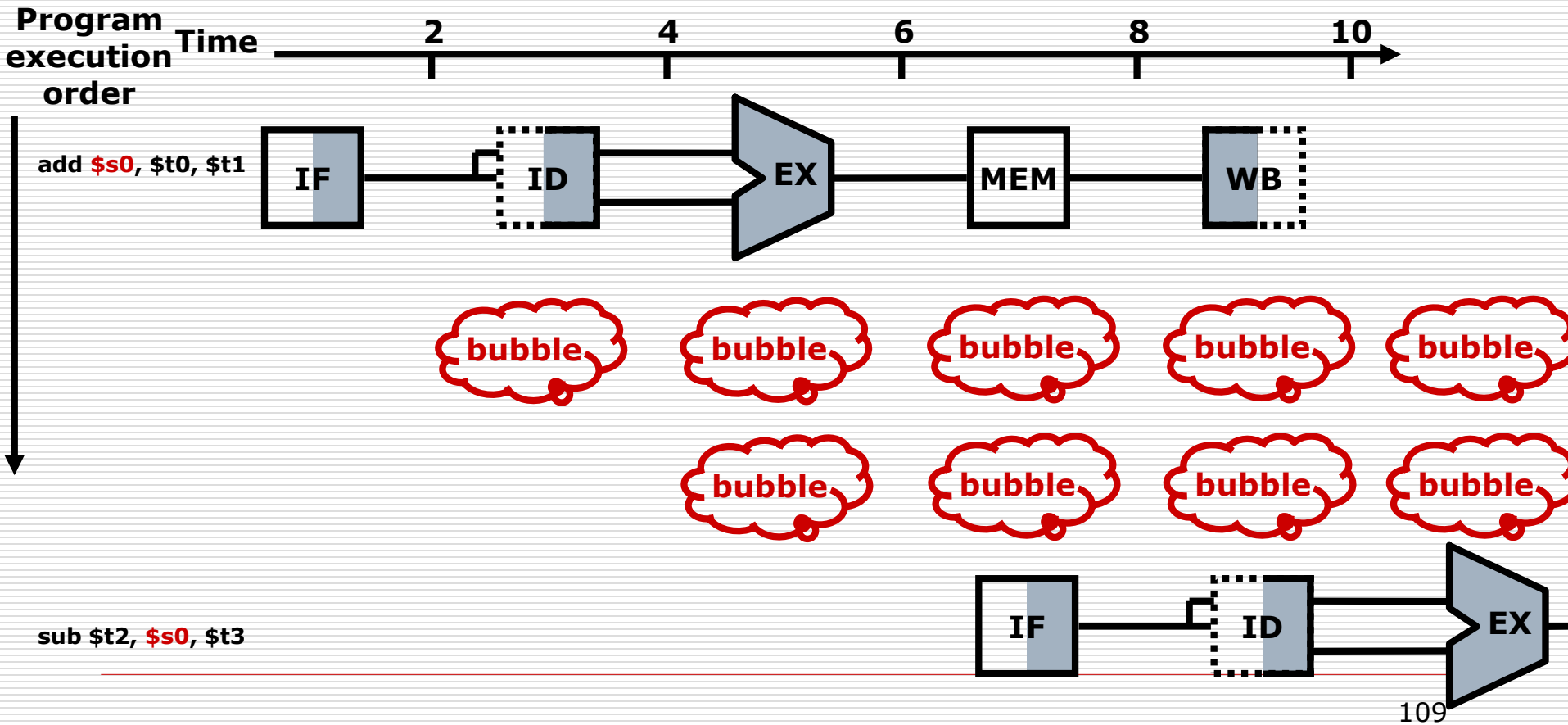
- Solution
  - Stalling
  - Forwarding (a.k.a Bypassing)

Graphically Representing Pipelines

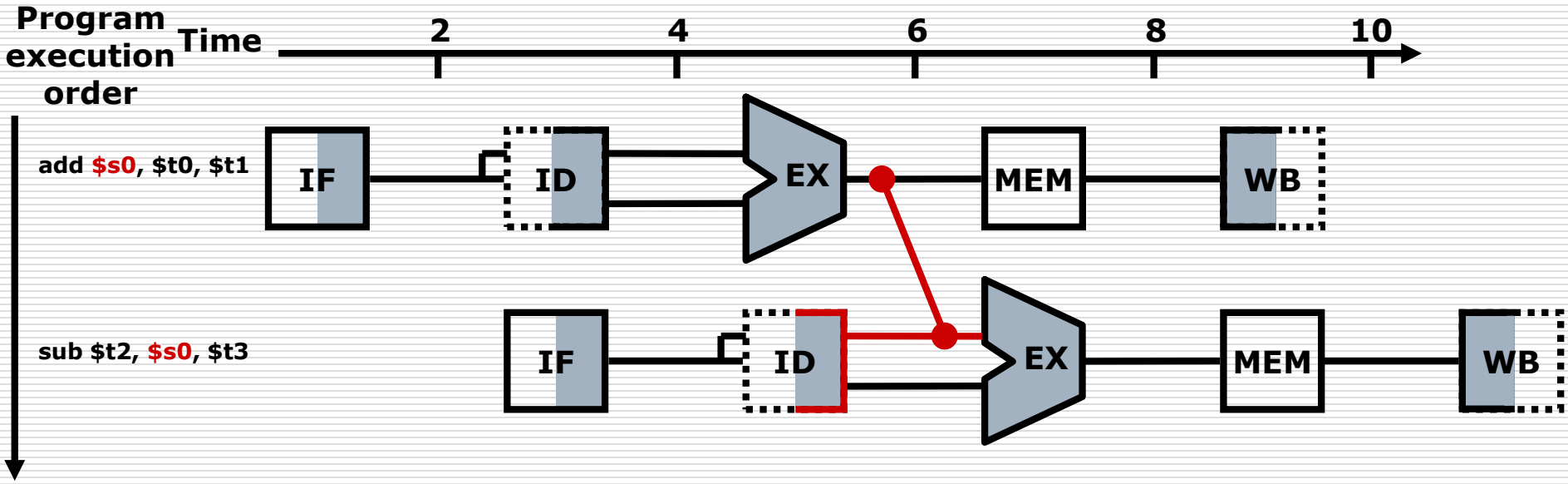


- shading on right half means “READ”
- shading on left half means “WRITE”
- white background means “NOT USED”
- dotted line means always “NO READ” and “NO WRITE” on ID and WB

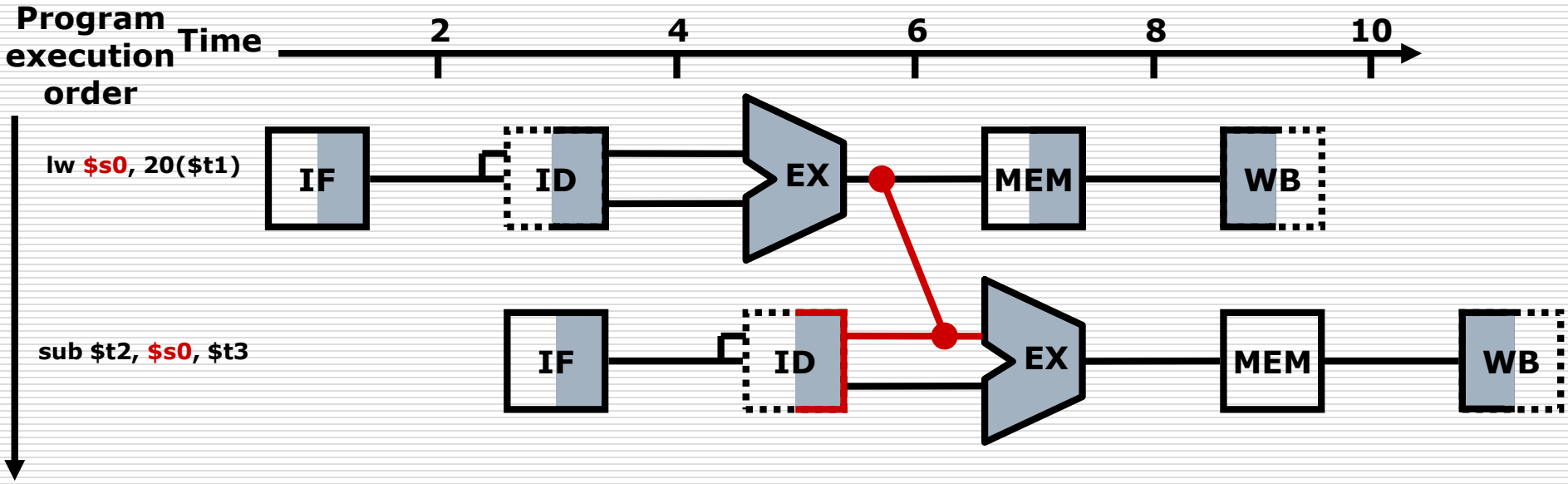
Stalling


Program execution order
Time
2
4
6
8
10
add $s0, $t0, $t1
IF
ID
EX
MEM
WB
bubble
bubble
bubble
bubble
bubble
bubble
bubble
bubble
bubble
bubble
sub $t2, $s0, $t3
IF
ID
EX

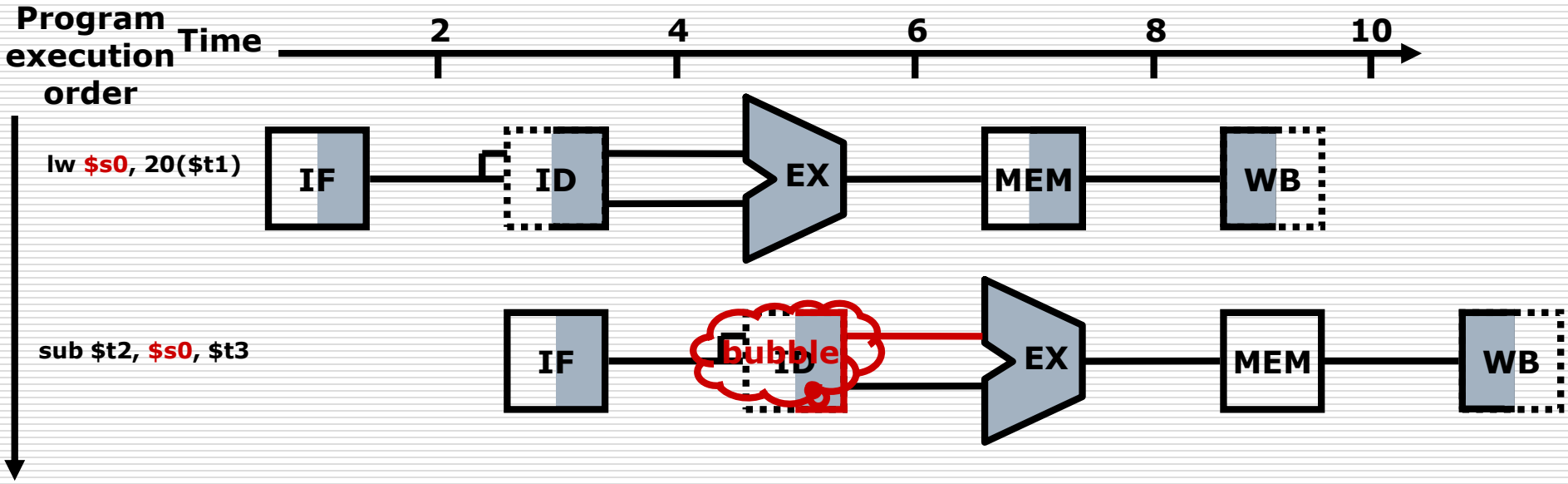
Forwarding


Program execution order
Time
2
4
6
8
10
add $s0, $t0, $t1
IF
ID
EX
MEM
WB
sub $t2, $s0, $t3
IF
ID
EX
MEM
WB

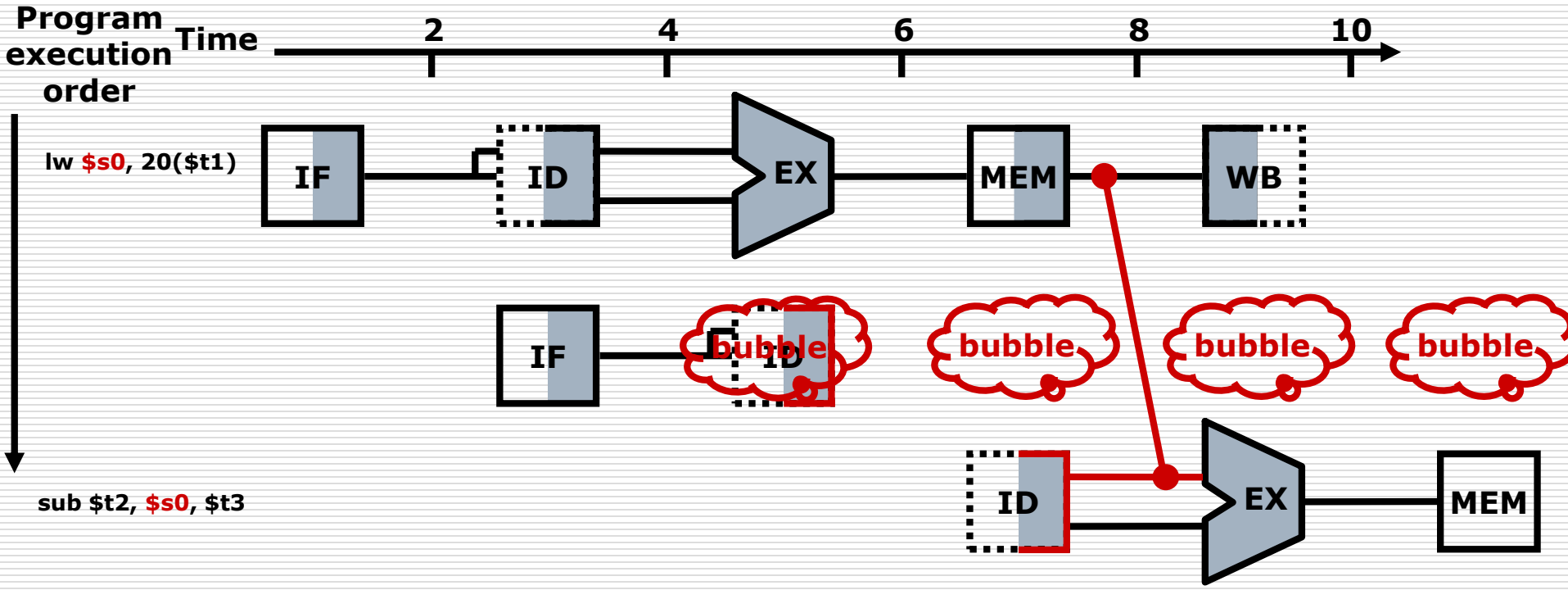
Load-Use Data Hazard



Load-Use Data Hazard



Load-Use Data Hazard



Code Scheduling to Avoid Stalls

- Try and avoid stalls! e.g., reorder these instructions:

```
lw $t0, 0($t1)
lw $t2, 4($t1)
sw $t2, 0($t1)
sw $t0, 4($t1)
```

- Add a "branch delay slot"
  - the next instruction after a branch is always executed
  - rely on compiler to "fill" the slot with something useful
- Superscalar: start more than one instruction in the same cycle

Code Scheduling to Avoid Stalls

```
                                # reg $t1 has the address of v[k]
lw      $t0, 0($t1)     # reg $t0 (temp) = v[k]
lw      $t2, 4($t1)     # reg $t2 = v[k+1]
sw      $t2, 0($t1)     # v[k] = reg $t2
sw      $t0, 4($t1)     # v[k+1] = reg $t0 (temp)
```

reorder

```
                                # reg $t1 has the address of v[k]
lw      $t0, 0($t1)     # reg $t0 (temp) = v[k]
lw      $t2, 4($t1)     # reg $t2 = v[k+1]
sw      $t0, 4($t1)     # v[k+1] = reg $t0 (temp)
sw      $t2, 0($t1)     # v[k] = reg $t2
```

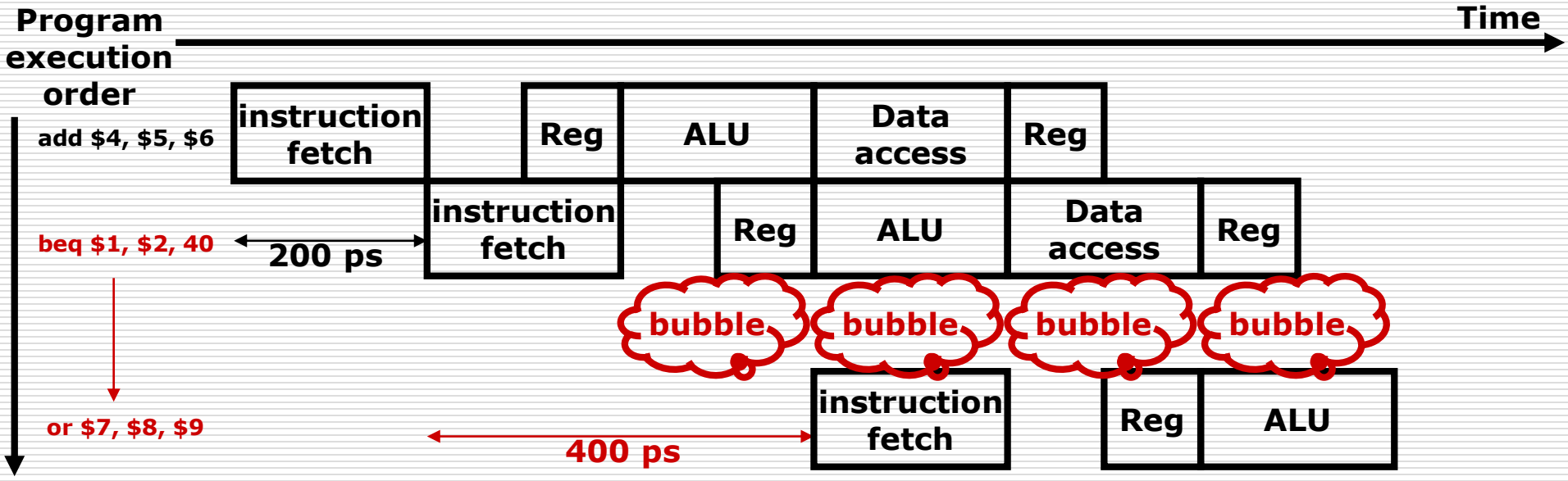
Control Hazards

- Branch determines flow of control
  - Fetching next instruction depends on branch outcome
  - Pipeline cannot always fetch correct instruction
    - Still working on ID stage of branch
- In MIPS pipeline
  - Need to compare registers and compute target early in the pipeline
  - Add hardware to do it in ID stage

Solutions of Control Hazards

- stall
  - certainly works, but is slow
- predict
  - does not slow down the pipeline when you are correct, otherwise redo
- delayed decision = delayed branch

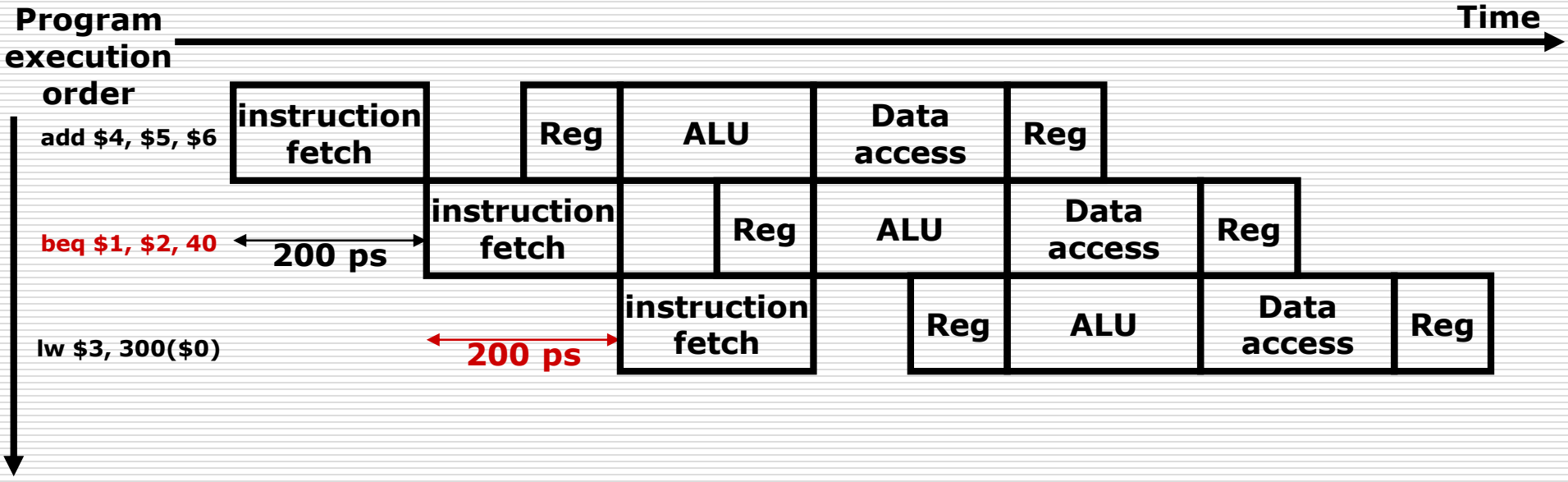
Stall on Branch



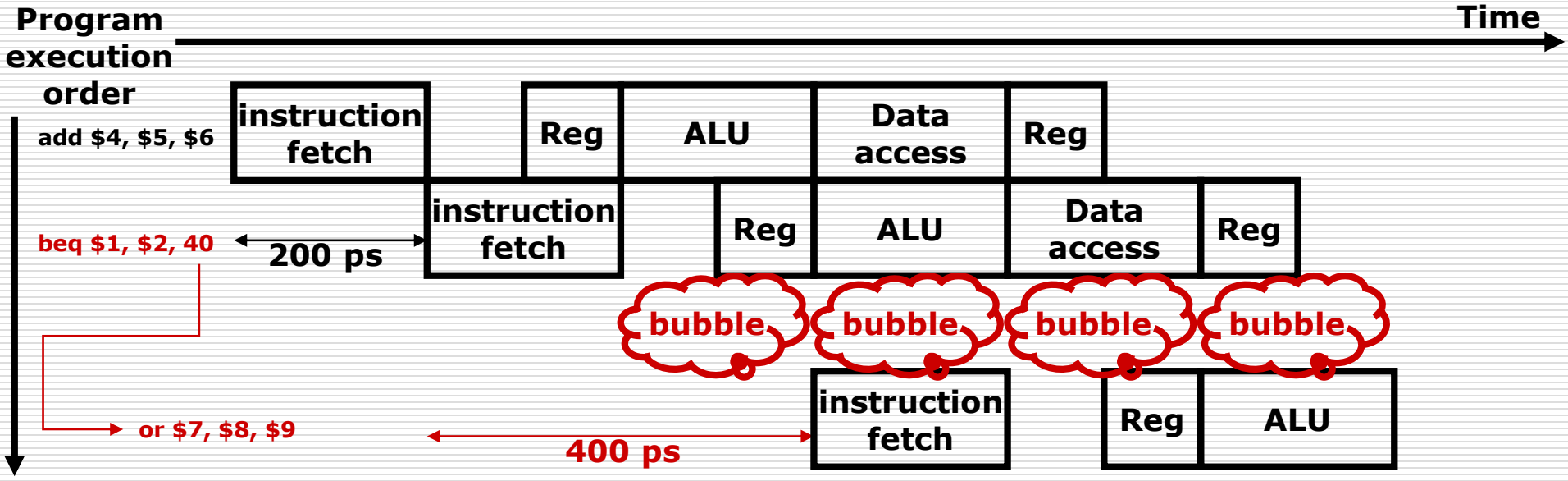
Branch Prediction

- Longer pipelines cannot readily determine branch outcome early
  - Stall penalty becomes unacceptable
- Predict outcome of branch
  - Only stall if prediction is wrong
- In MIPS pipeline
  - Can predict branches not taken
  - Fetch instruction after branch, with no delay

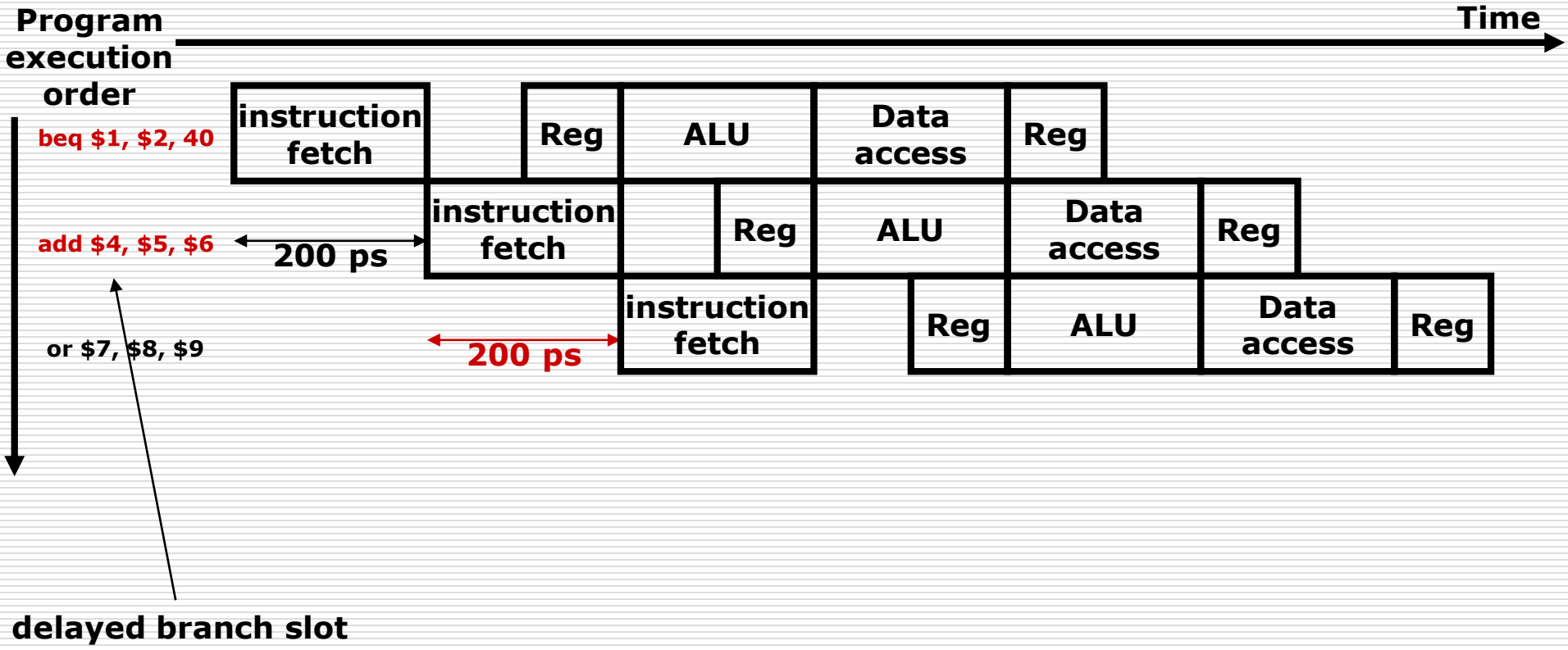
Predict - branch will be untaken



Predict - failed



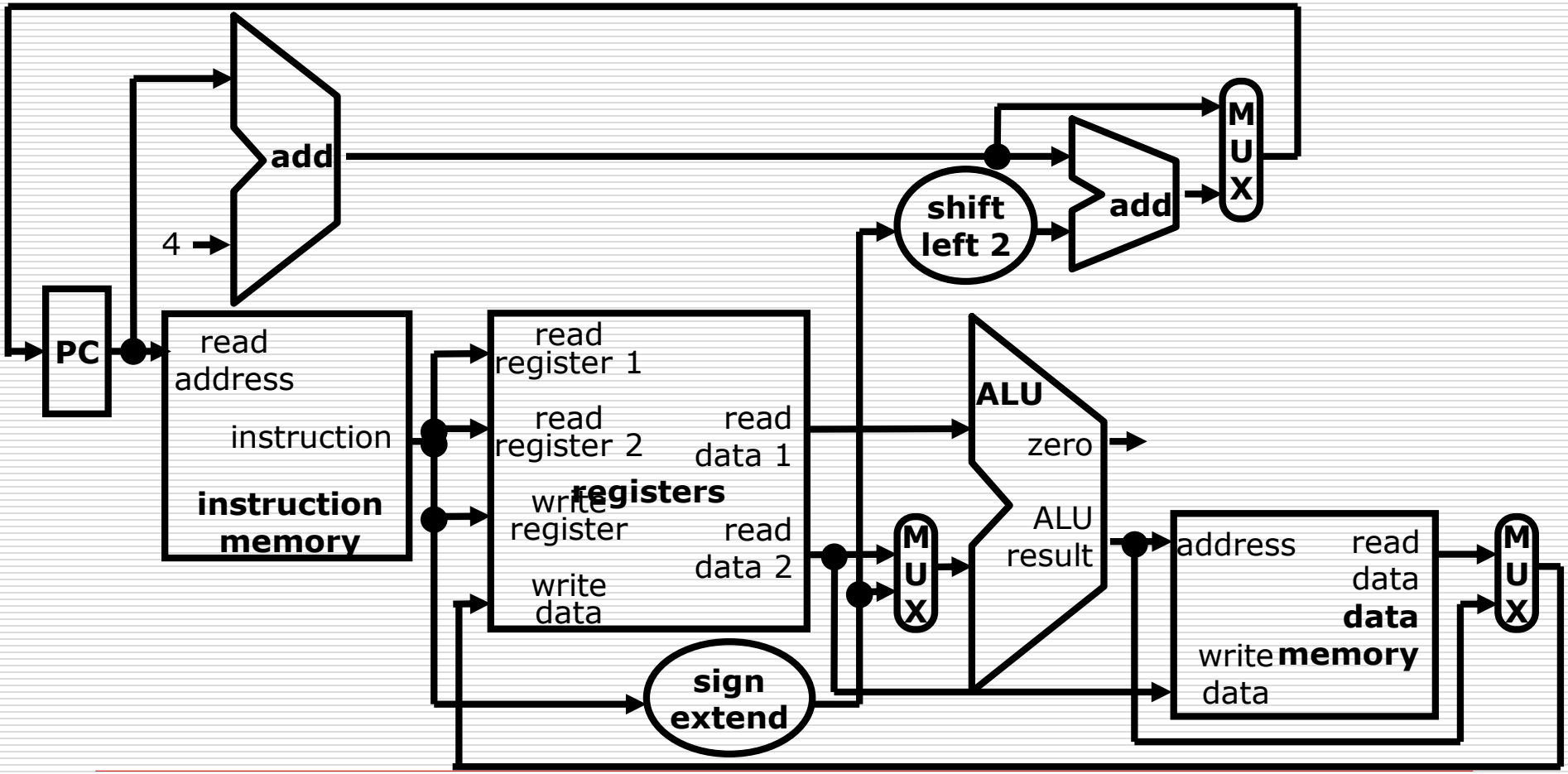
Delayed Branch



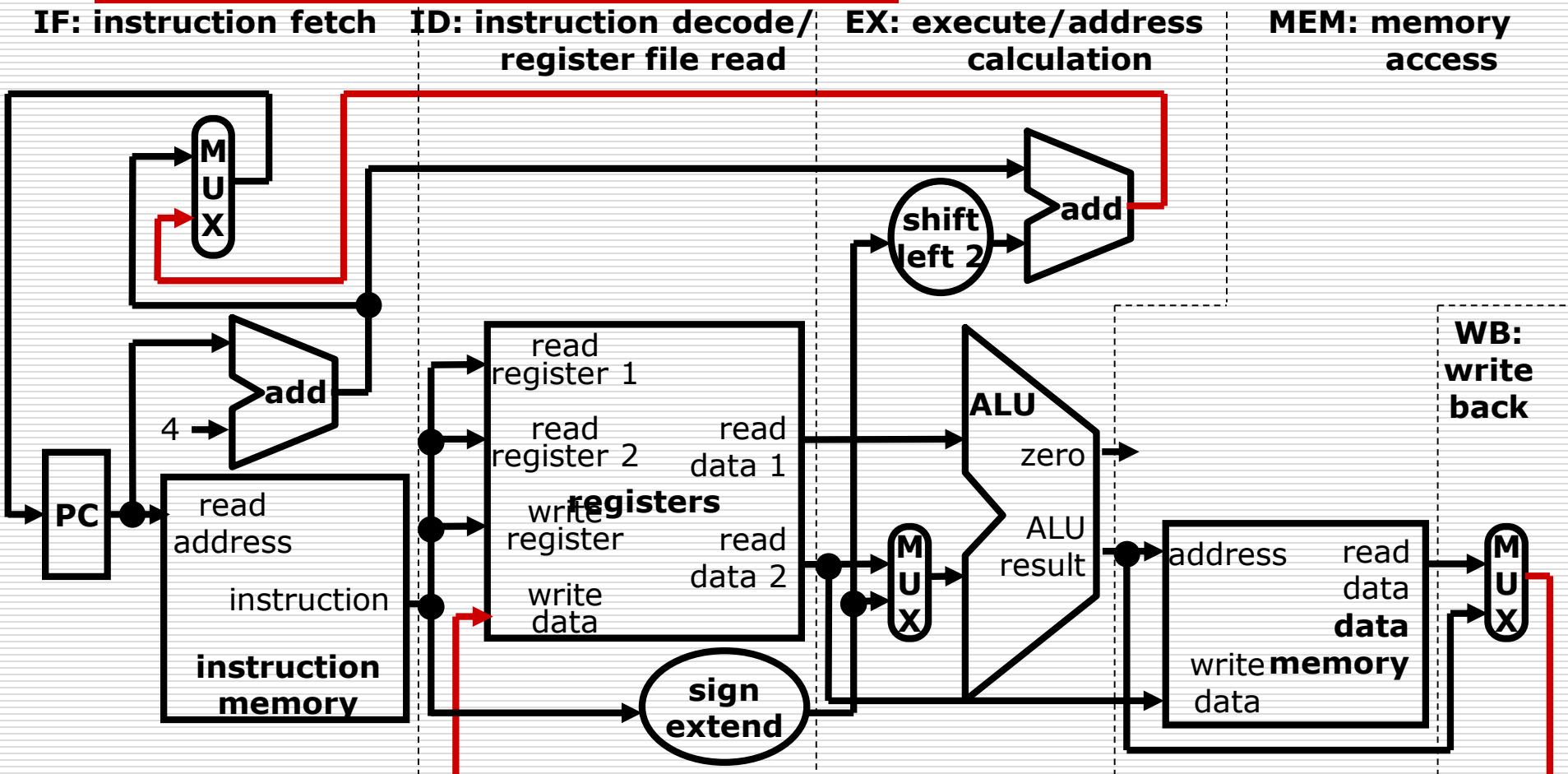
More-Realistic Branch Prediction

- Static branch prediction
  - Based on typical branch behavior
  - Example: loop and if-statement branches
    - Predict backward branches taken
    - Predict forward branches not taken
- Dynamic branch prediction
  - Hardware measures actual branch behavior
    - e.g., record recent history of each branch
  - Assume future behavior will continue the trend
    - When wrong, stall while re-fetching, and update history

Recall:
Single Cycle Implementation

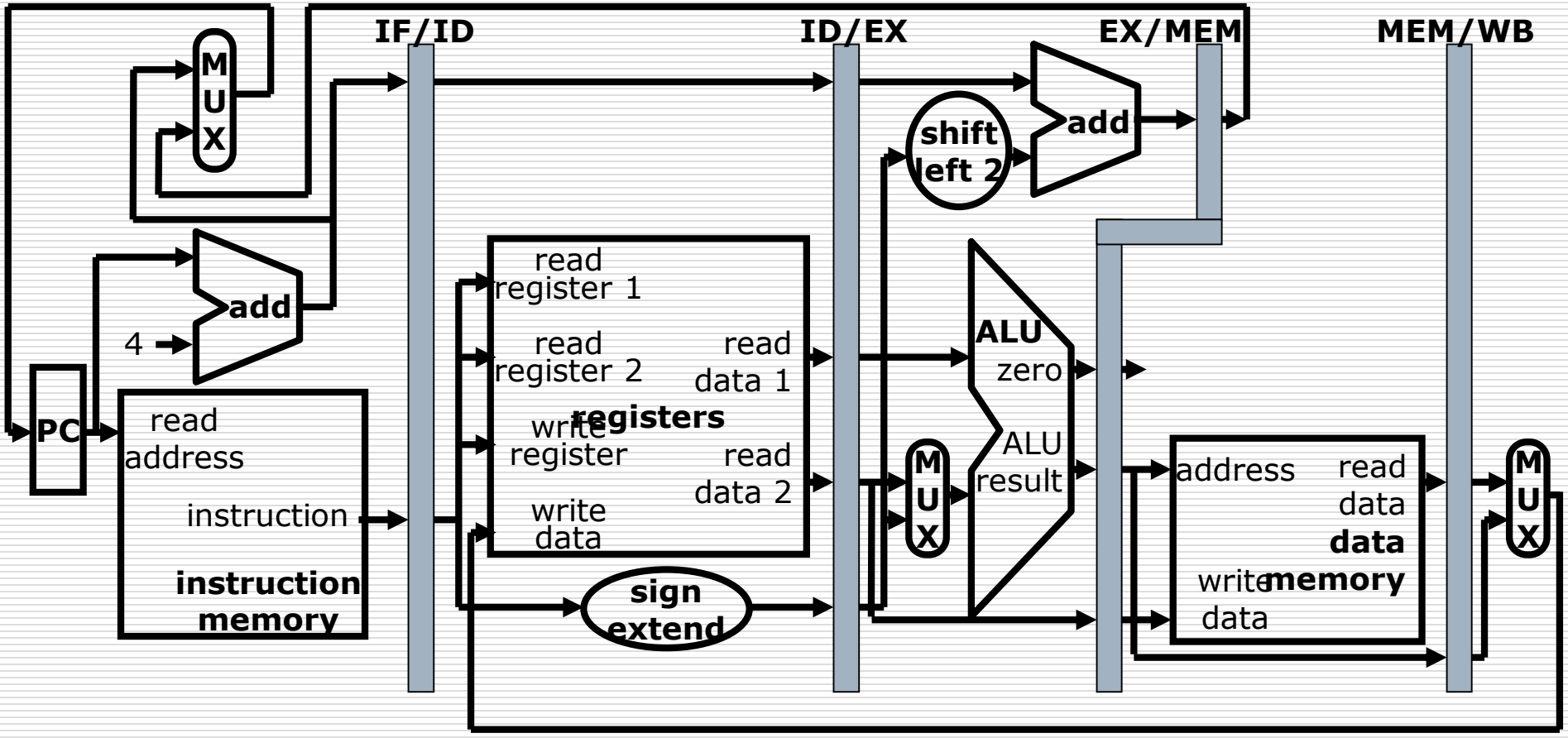


Pipelined Datapath



What do we need to add to actually split the datapath into stages?

Pipeline registers

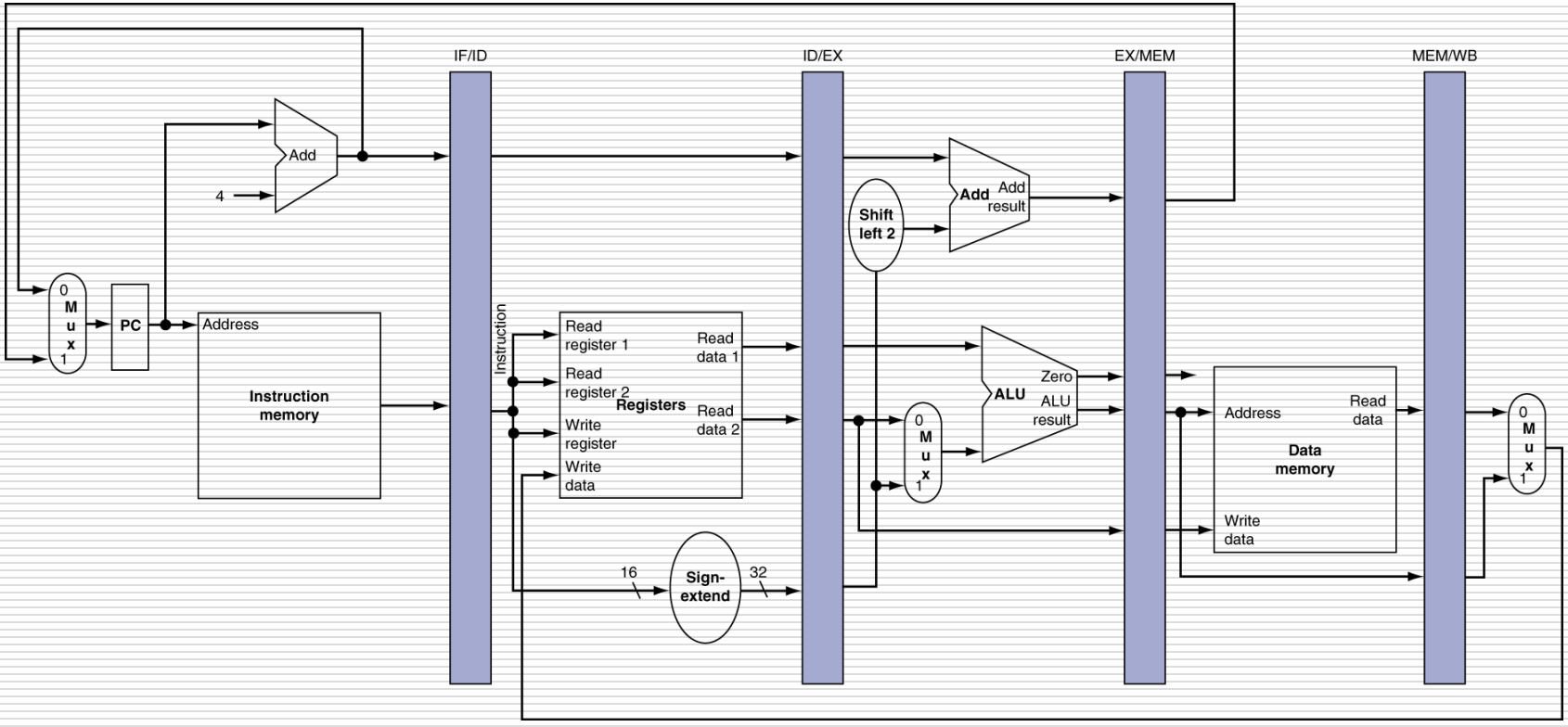


Can you find a problem even if there are no dependencies?
What instructions can we execute to manifest the problem?

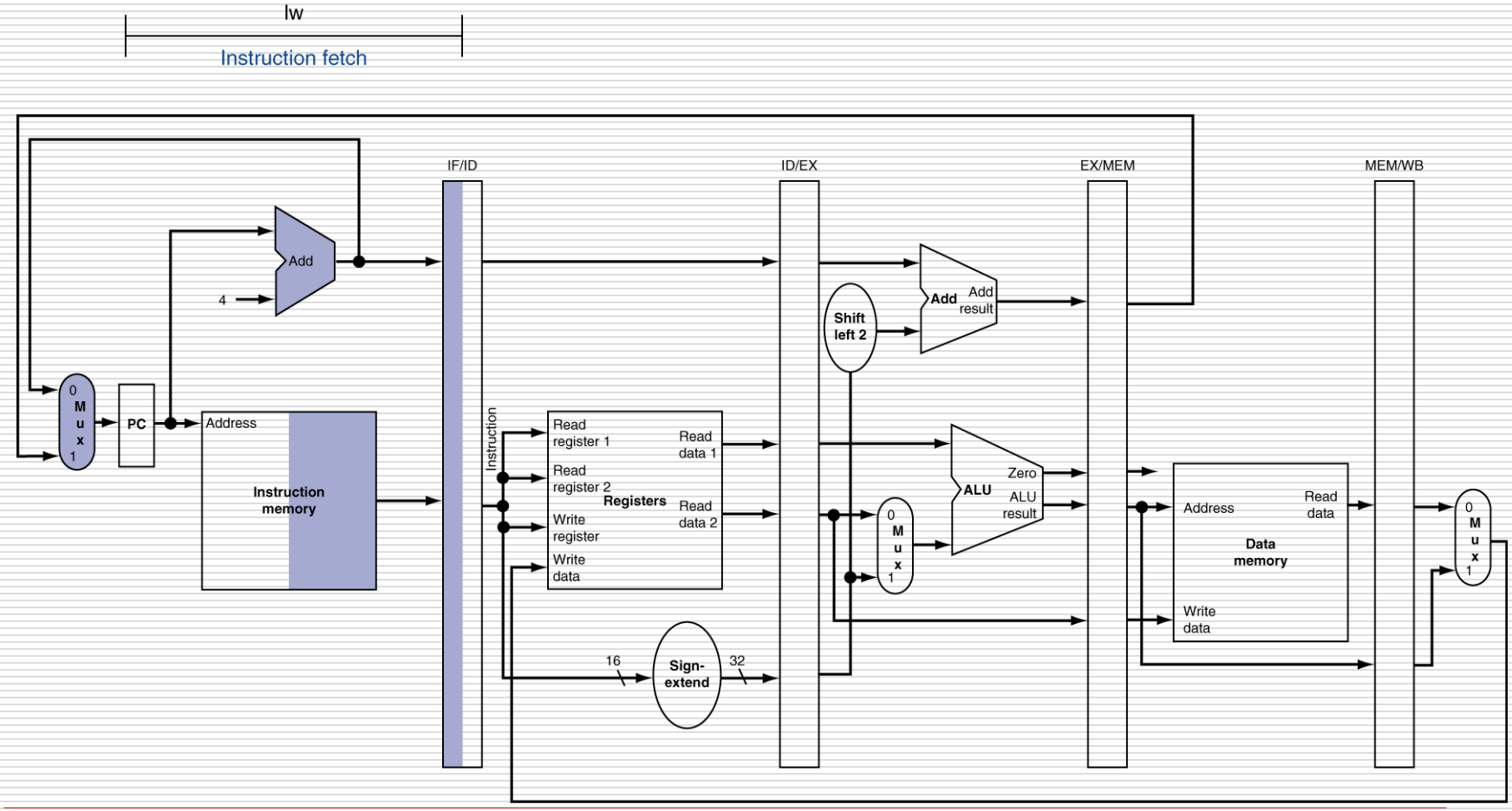
Pipeline Operation

- Cycle-by-cycle flow of instructions through the pipelined datapath
  - "Single-clock-cycle" pipeline diagram
    - Shows pipeline usage in a single cycle
    - Highlight resources used
  - c.f. "multi-clock-cycle" diagram
    - Graph of operation over time
- We will look at "single-clock-cycle" diagrams for load & store

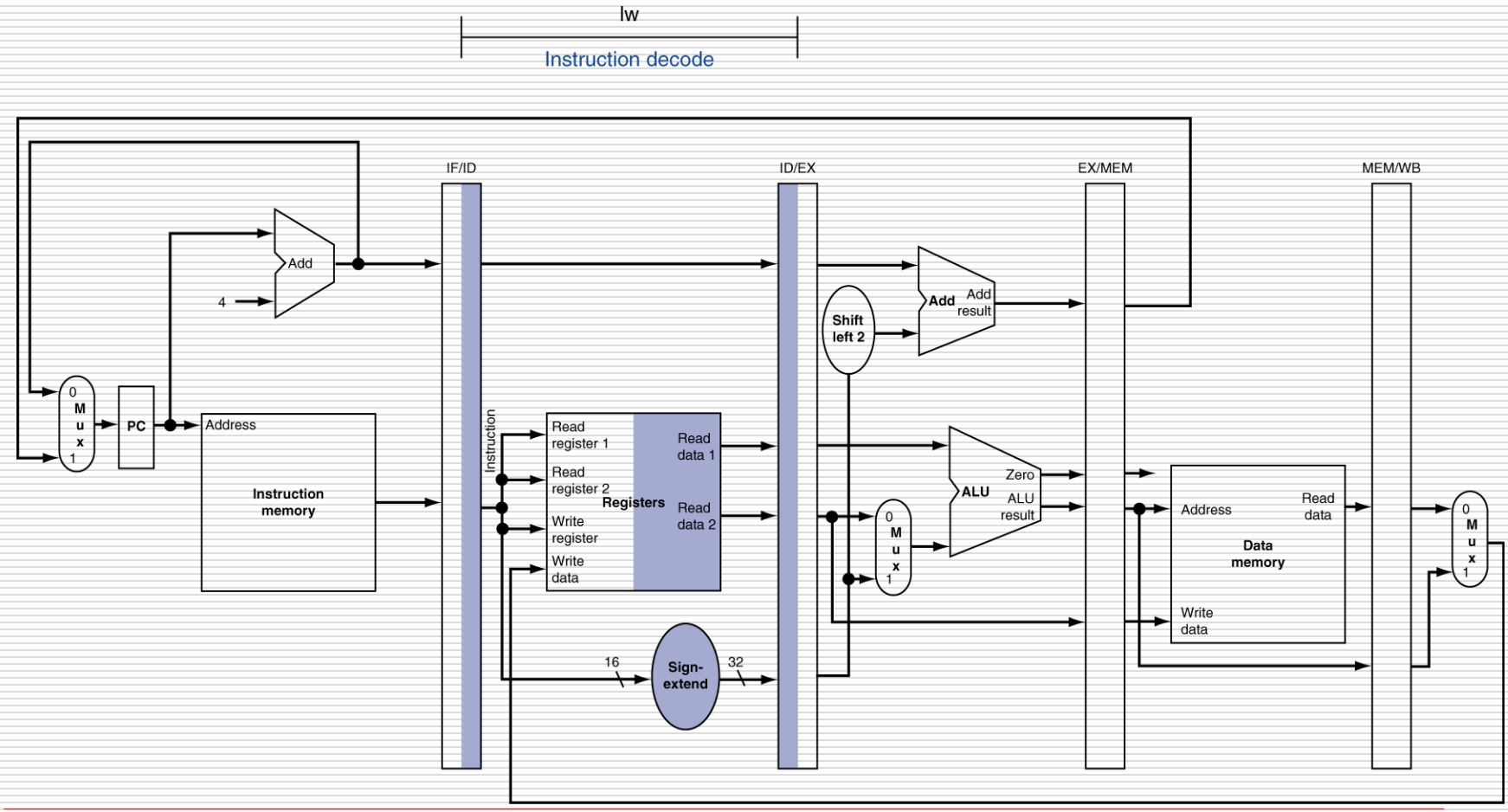
Original Pipelined Datapath



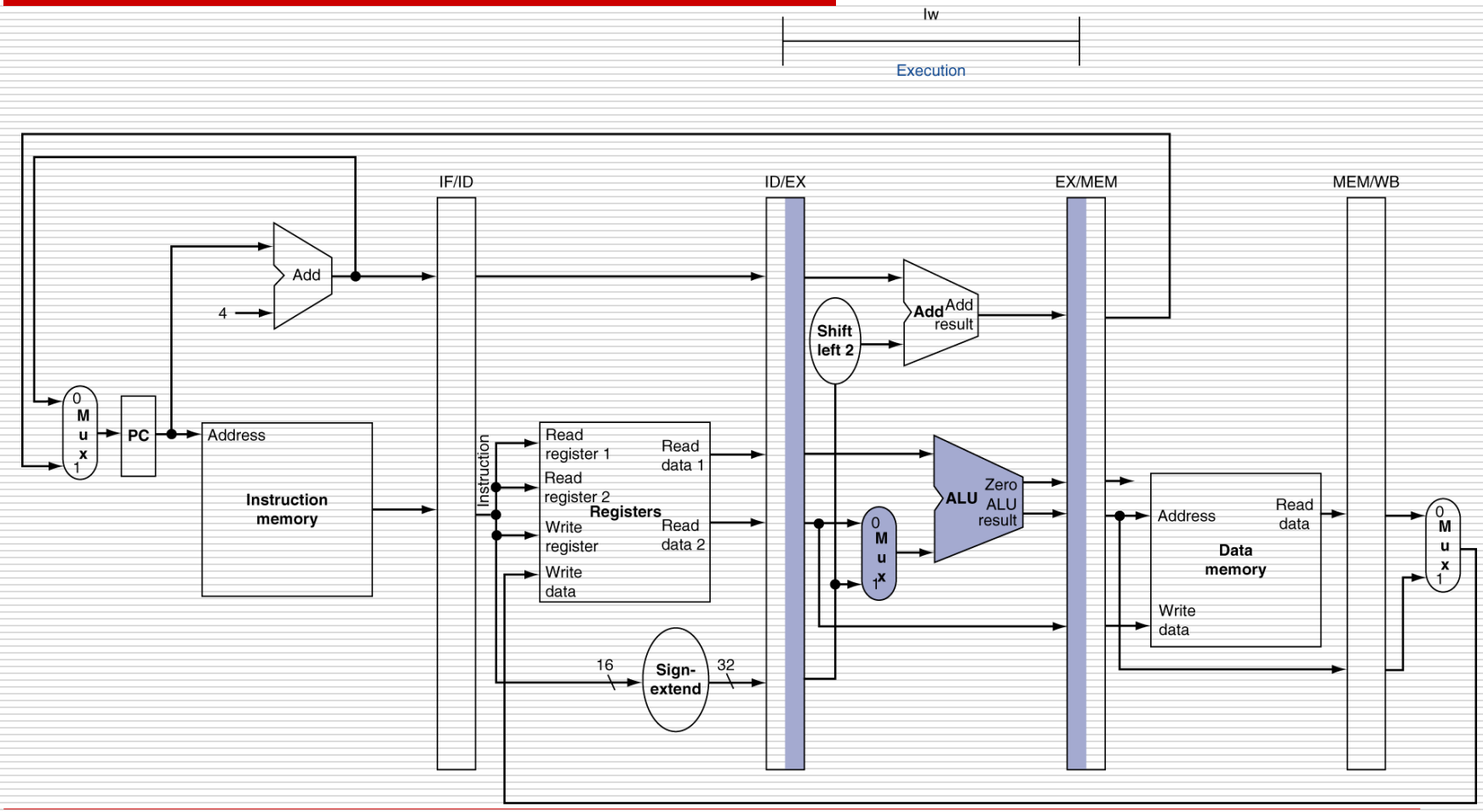
IF for Load, Store, …



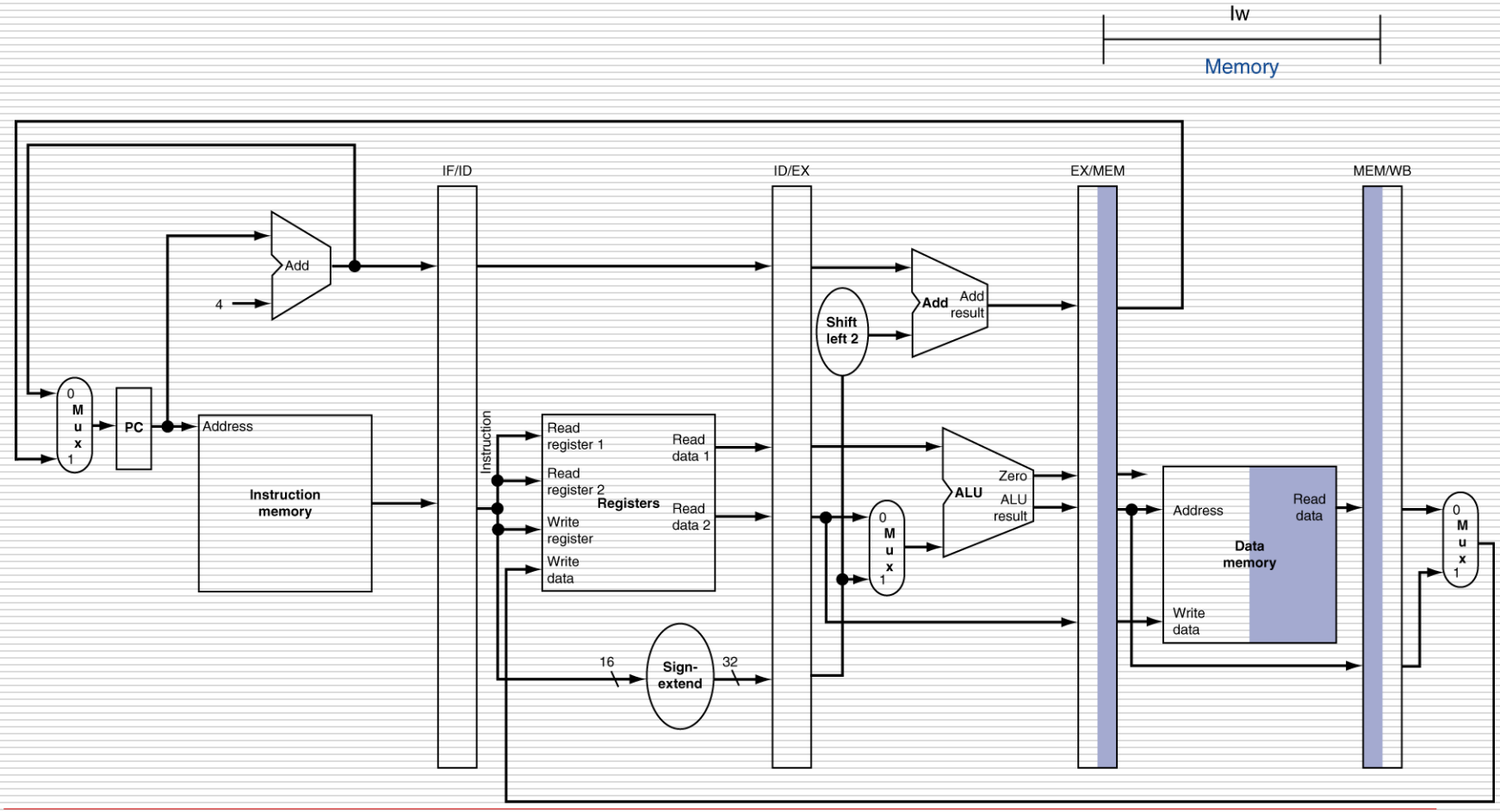
ID for Load, Store, …



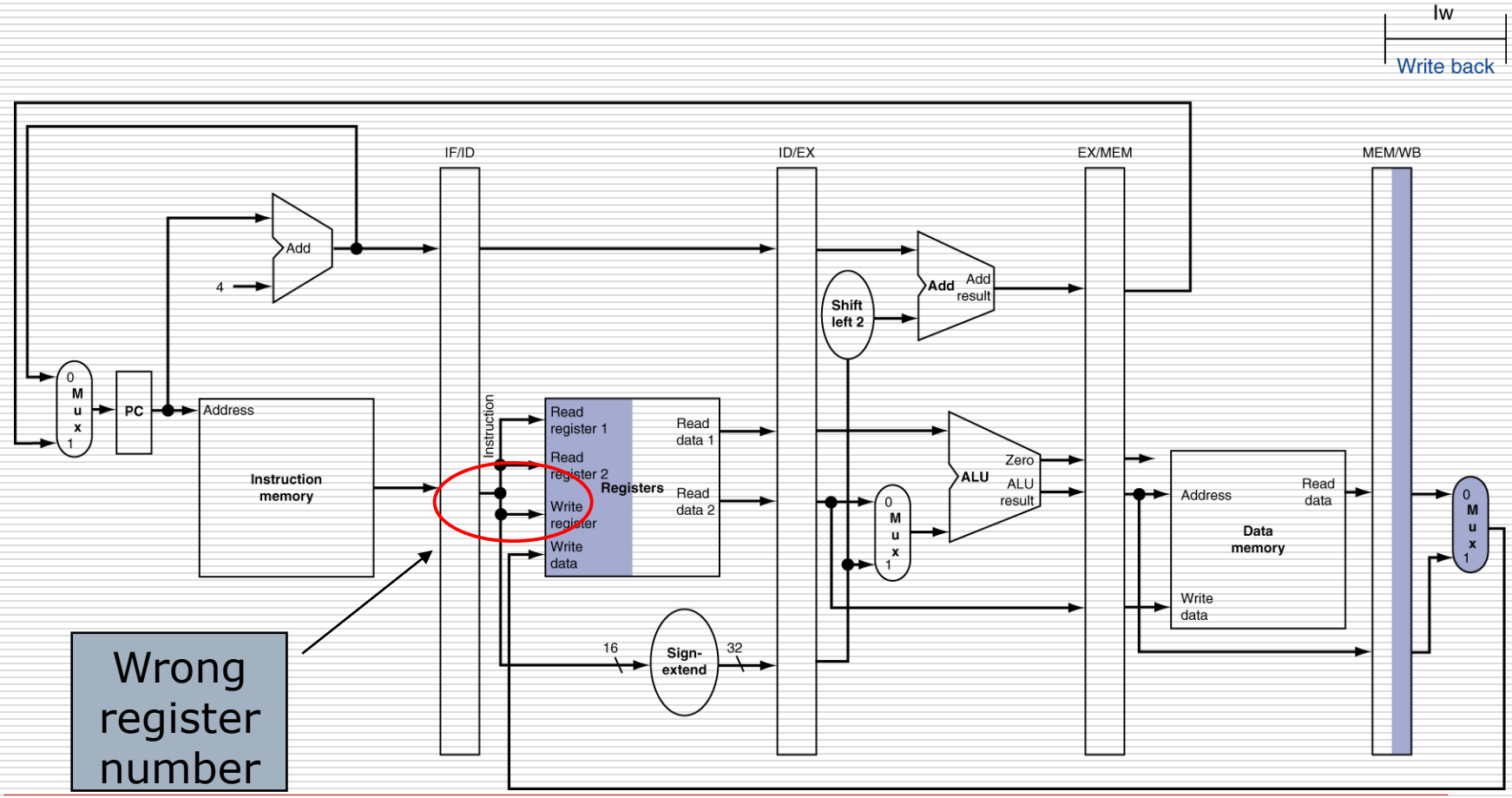
EX for Load


lw
Execution
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
M
u
x
0
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
16
Sign-extend
32
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Read data
Data memory
Write data

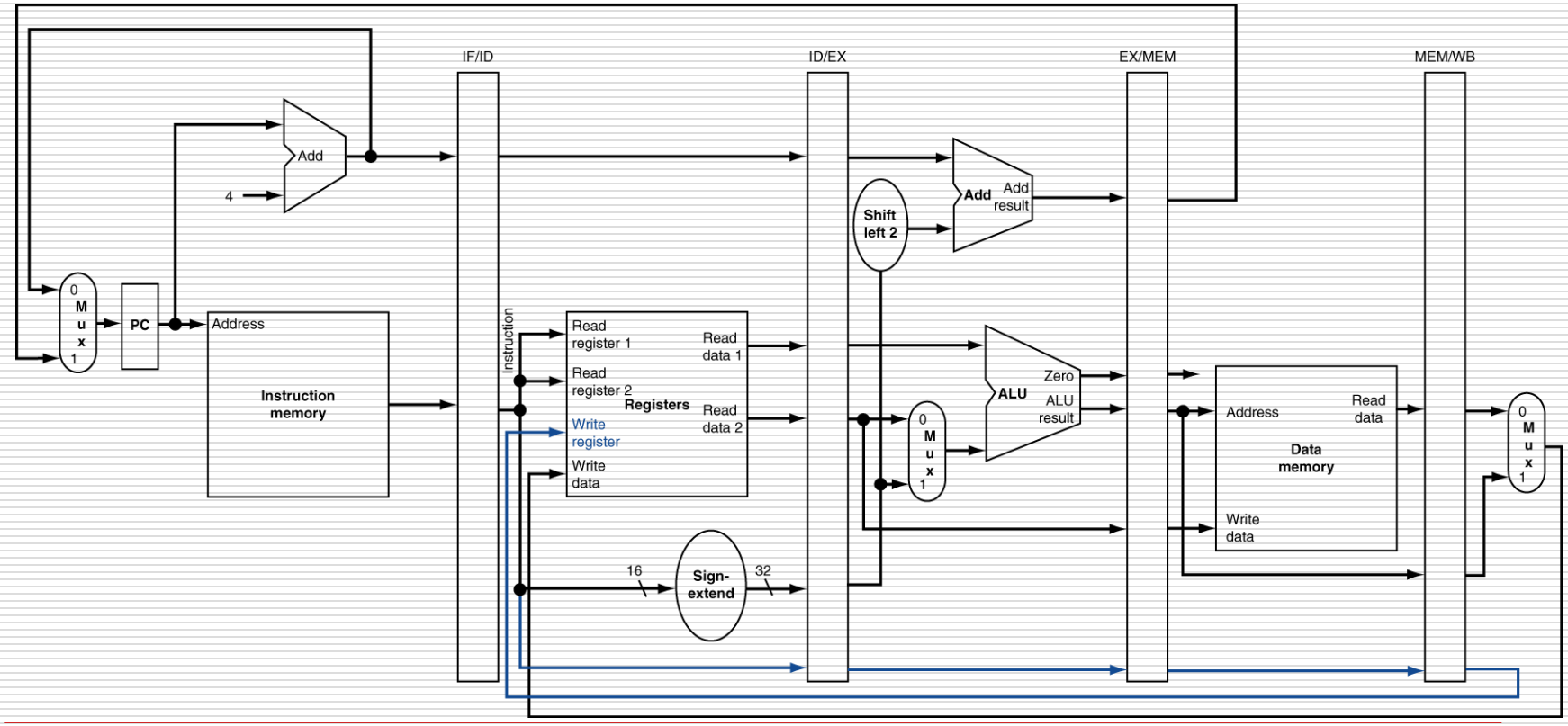
MEM for Load


lw
Memory
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
M u x
0
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
16
Sign-extend
32
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Data memory
Read data
Write data

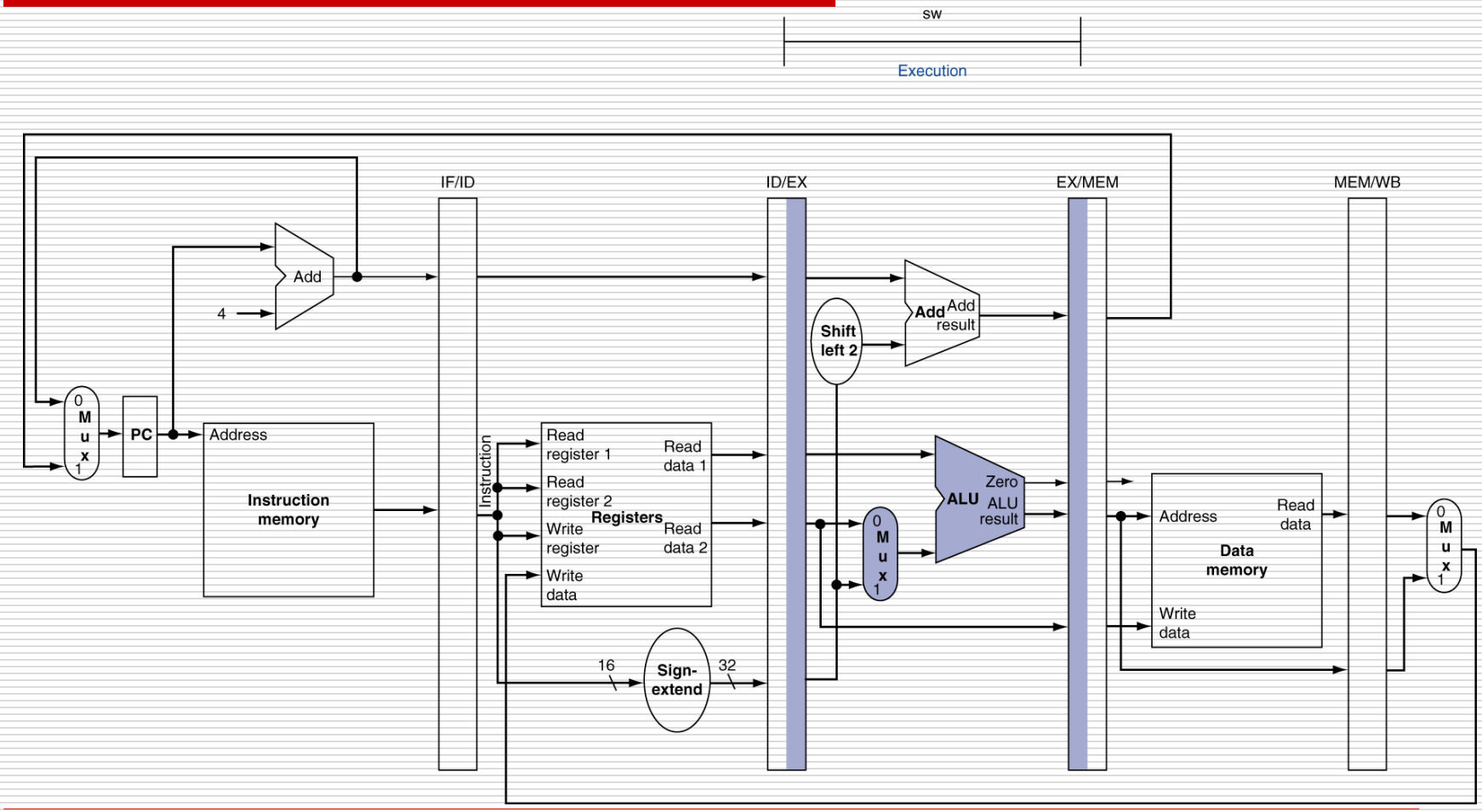
WB for Load


lw
Write back
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
M u x
0
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Read data
Data memory
Write data
16
Sign-extend
32
Wrong register number

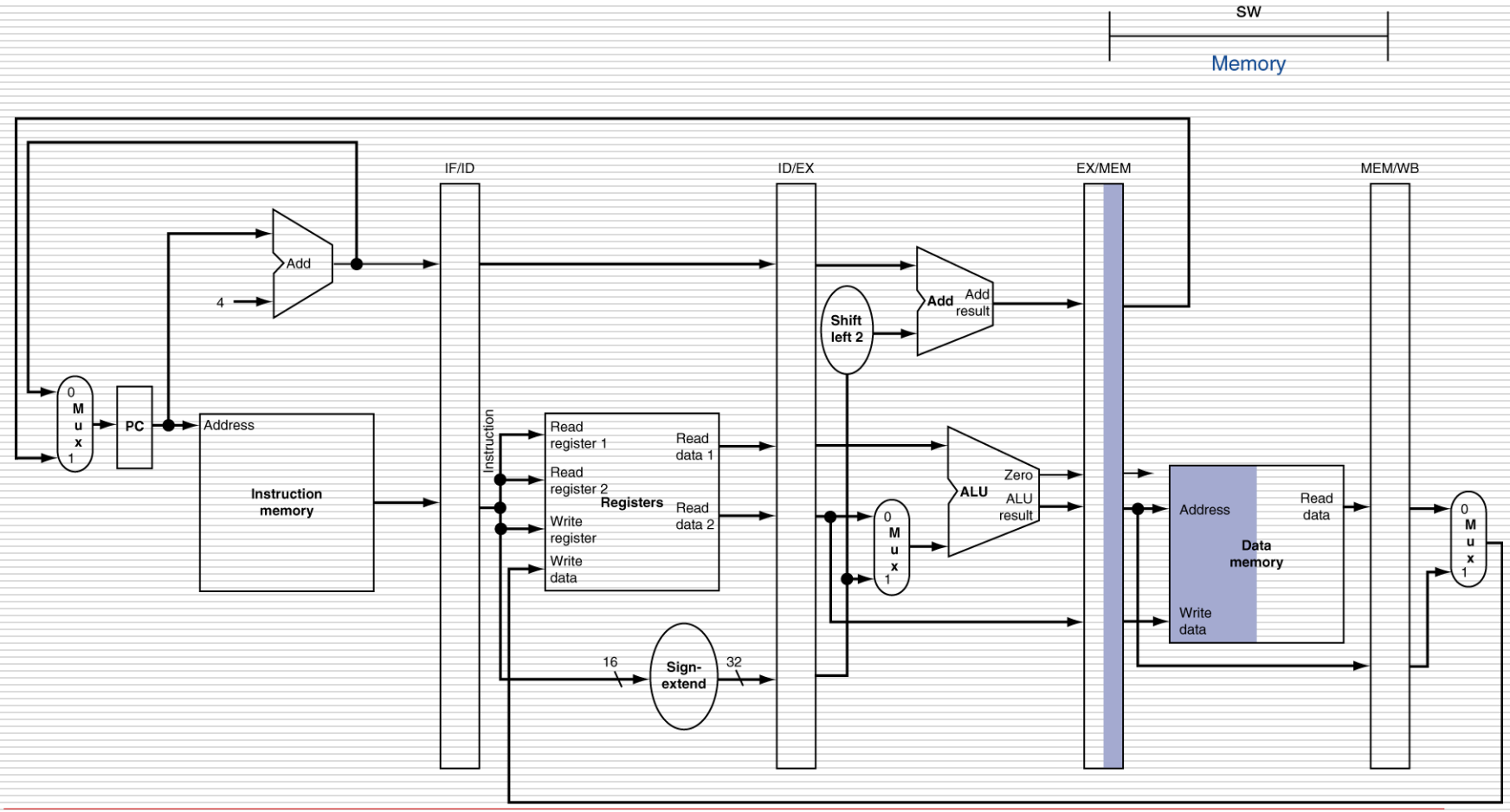
Corrected Datapath for Load



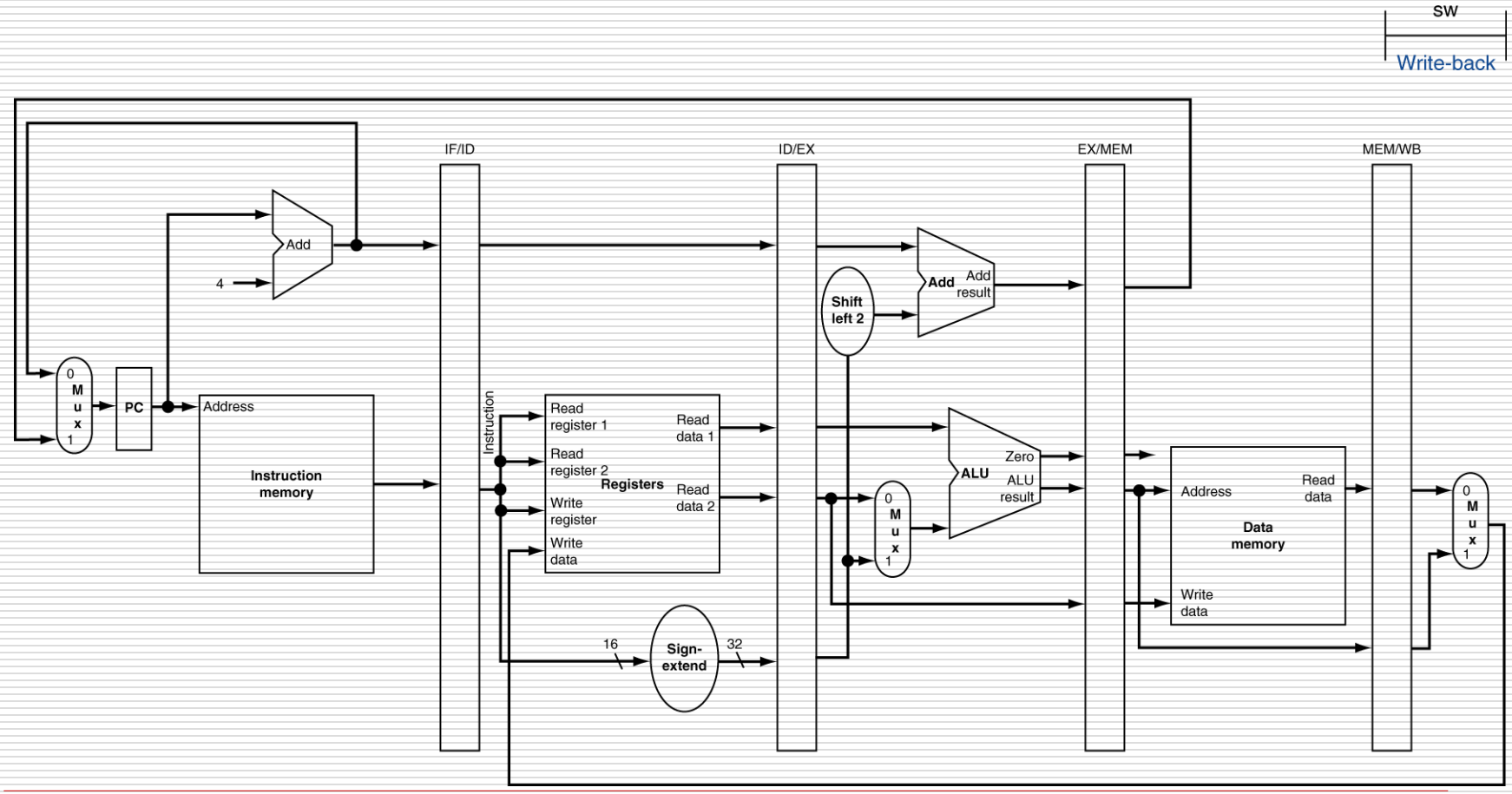
EX for Store


sw
Execution
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
M
u
x
0
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Write register
Write data
Registers
Read data 1
Read data 2
16
Sign-extend
32
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Read data
Data memory
Write data

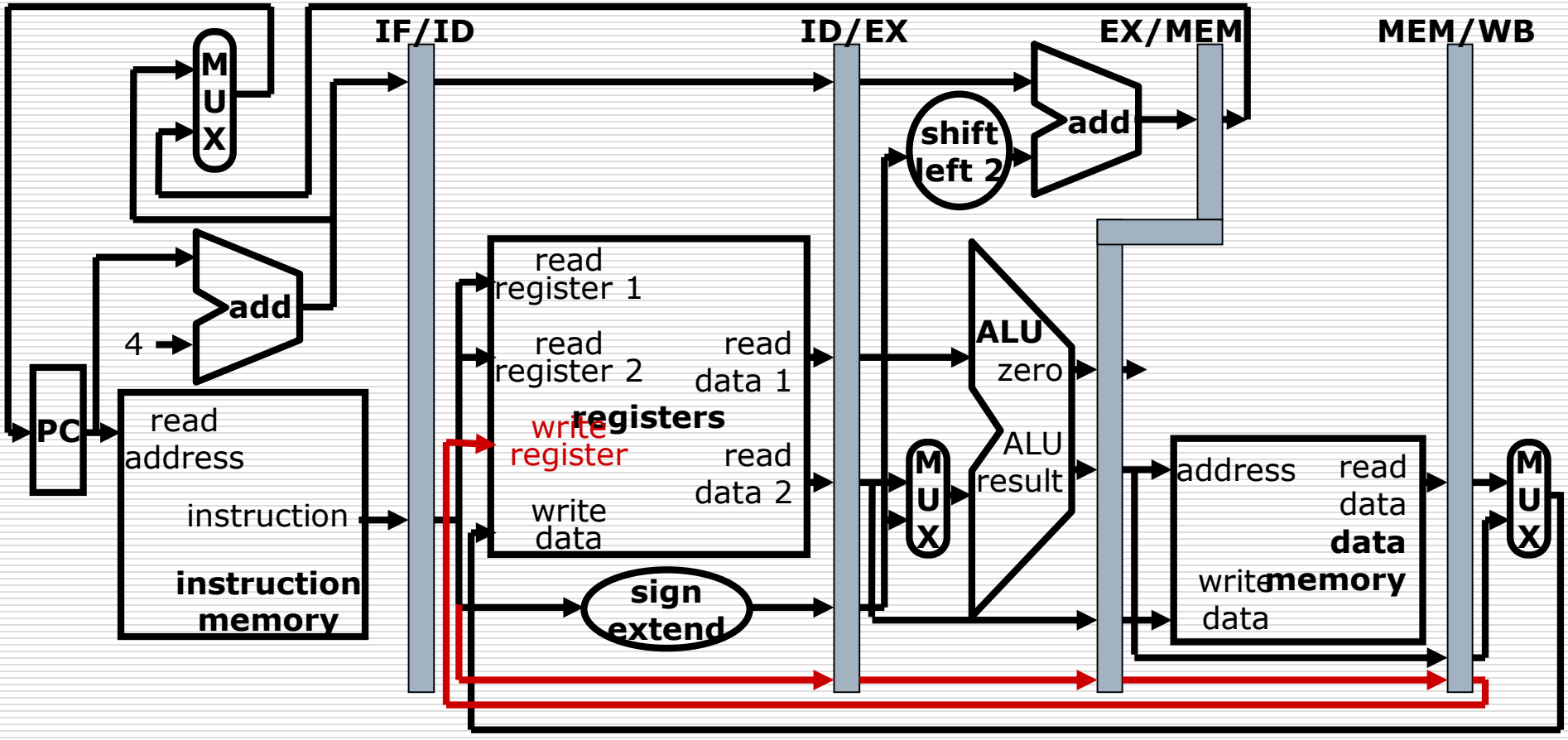
MEM for Store



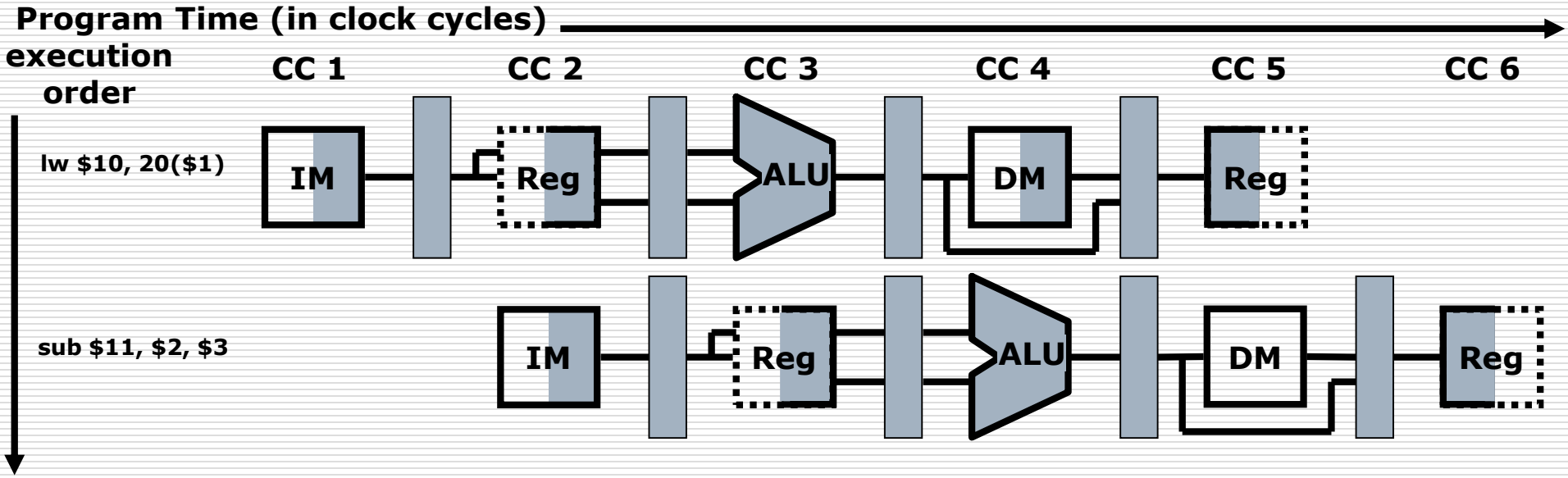
WB for Store


SW
Write-back
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
M
u
x
0
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
16
Sign-extend
32
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Data memory
Read data
Write data

Corrected Datapath
- single-clock-cycle pipeline diagram

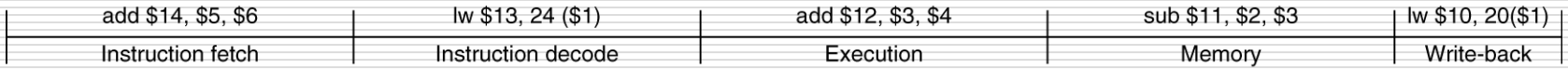


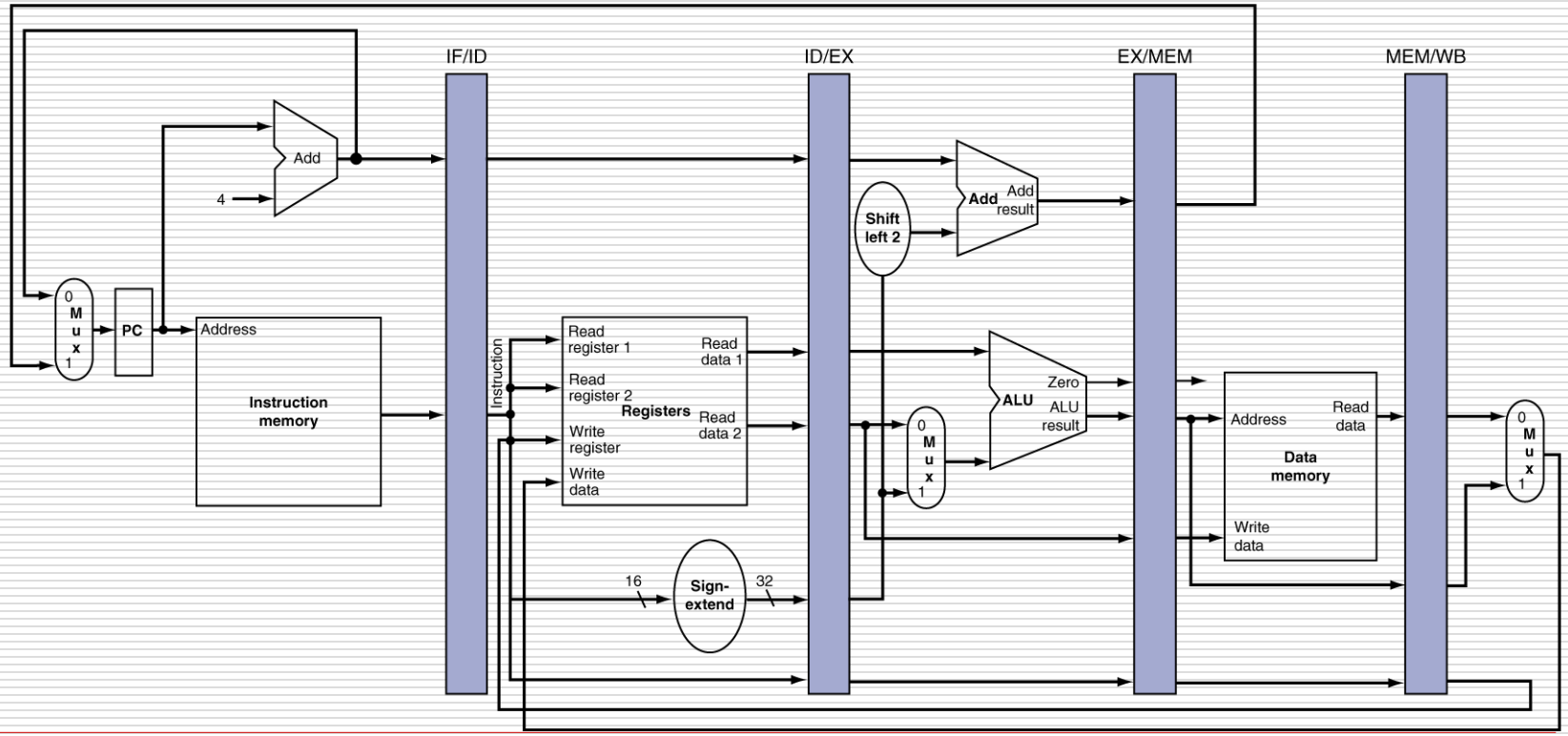
Multi-Cycle Pipeline Diagram



- Can help with answering questions like:
  - how many cycles does it take to execute this code?
  - what is the ALU doing during cycle 4?
  - use this representation to help understand datapaths

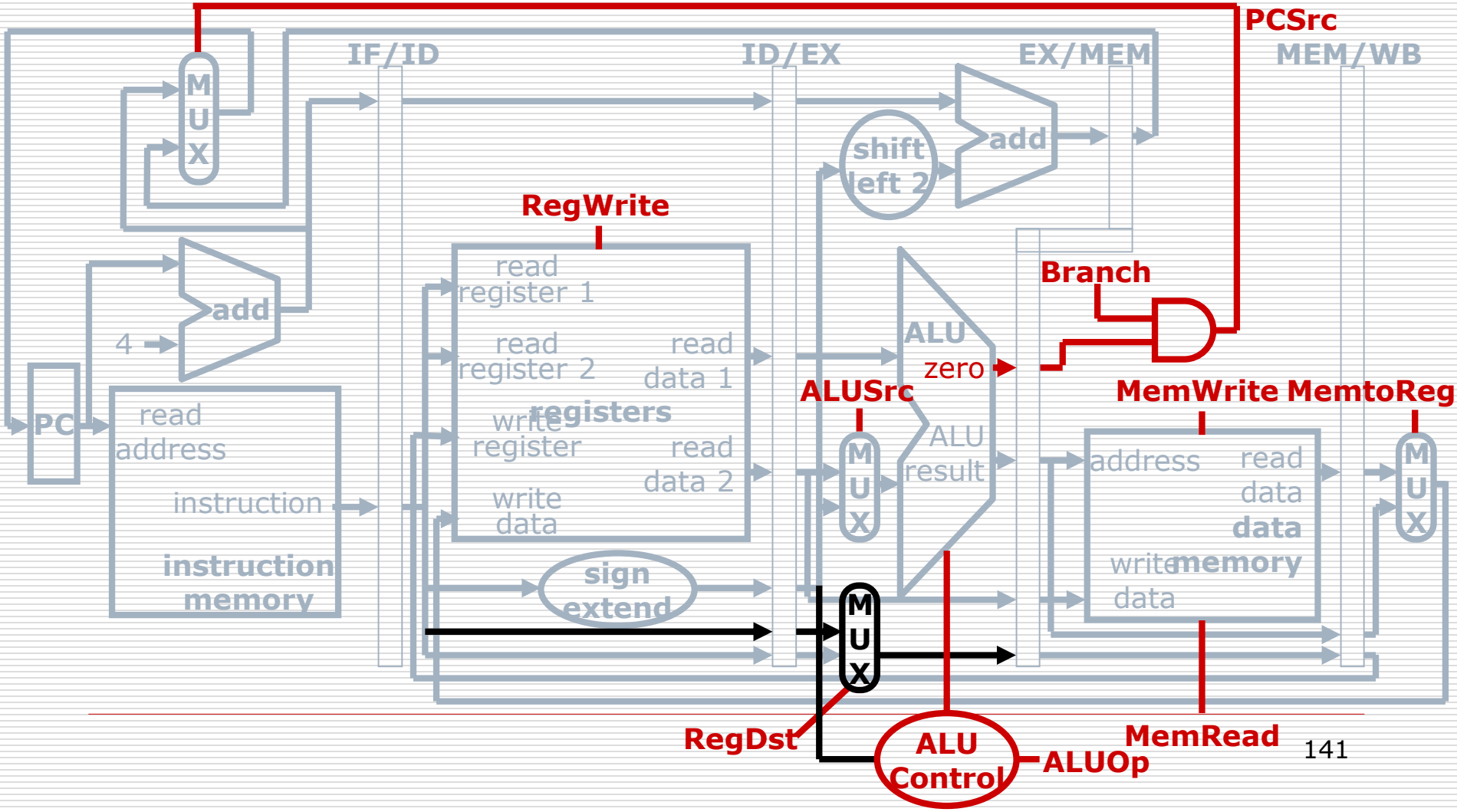
Single-Cycle Pipeline Diagram





- State of pipeline in a given cycle

Pipelined Control



Pipelined Control

- We have 5 stages.
  What needs to be controlled in each stage?
  - Instruction Fetch and PC Increment
  - Instruction Decode / Register Fetch
  - Execution
  - Memory Stage
  - Write Back
- How would control be handled in an automobile plant?
  - a fancy control center telling everyone what to do?
  - should we use a finite state machine?

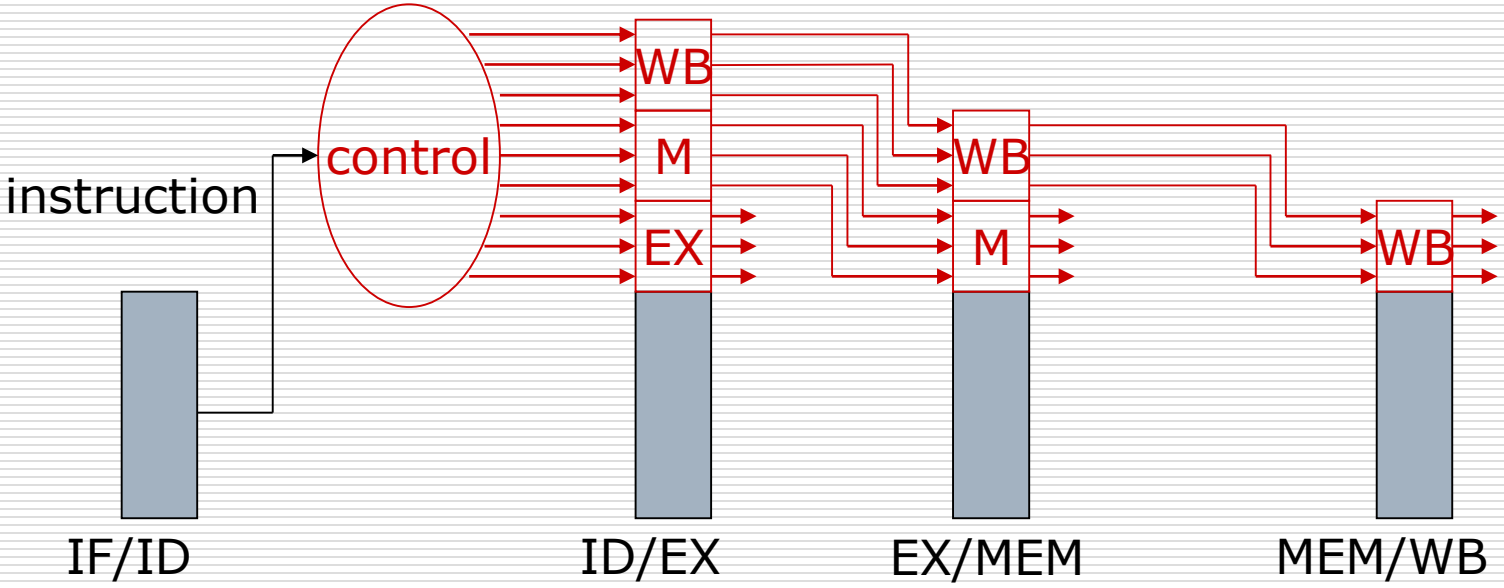
Pipelined Control

- Pass control signals along just like the data

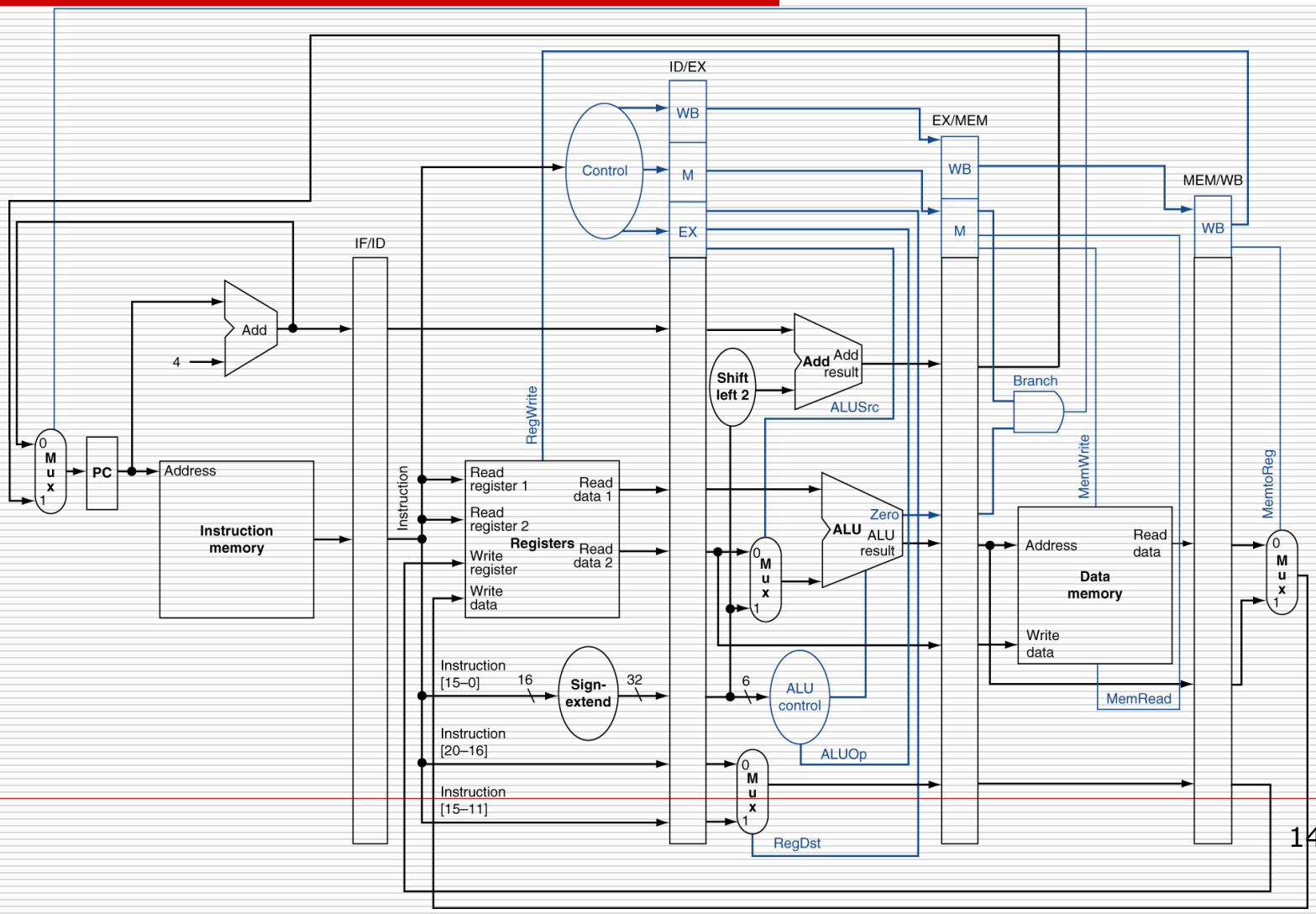
	Execution/Address calculation stage control lines				Memory access stage control lines			stage control lines	
	Reg Dst	ALU Op1	ALU Op0	ALU Src	Branch	Mem Read	Mem Write	Reg Write	Memto Reg
R-format	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X

Pipelined Control

- Control signals derived from instruction
  - As in single-cycle implementation

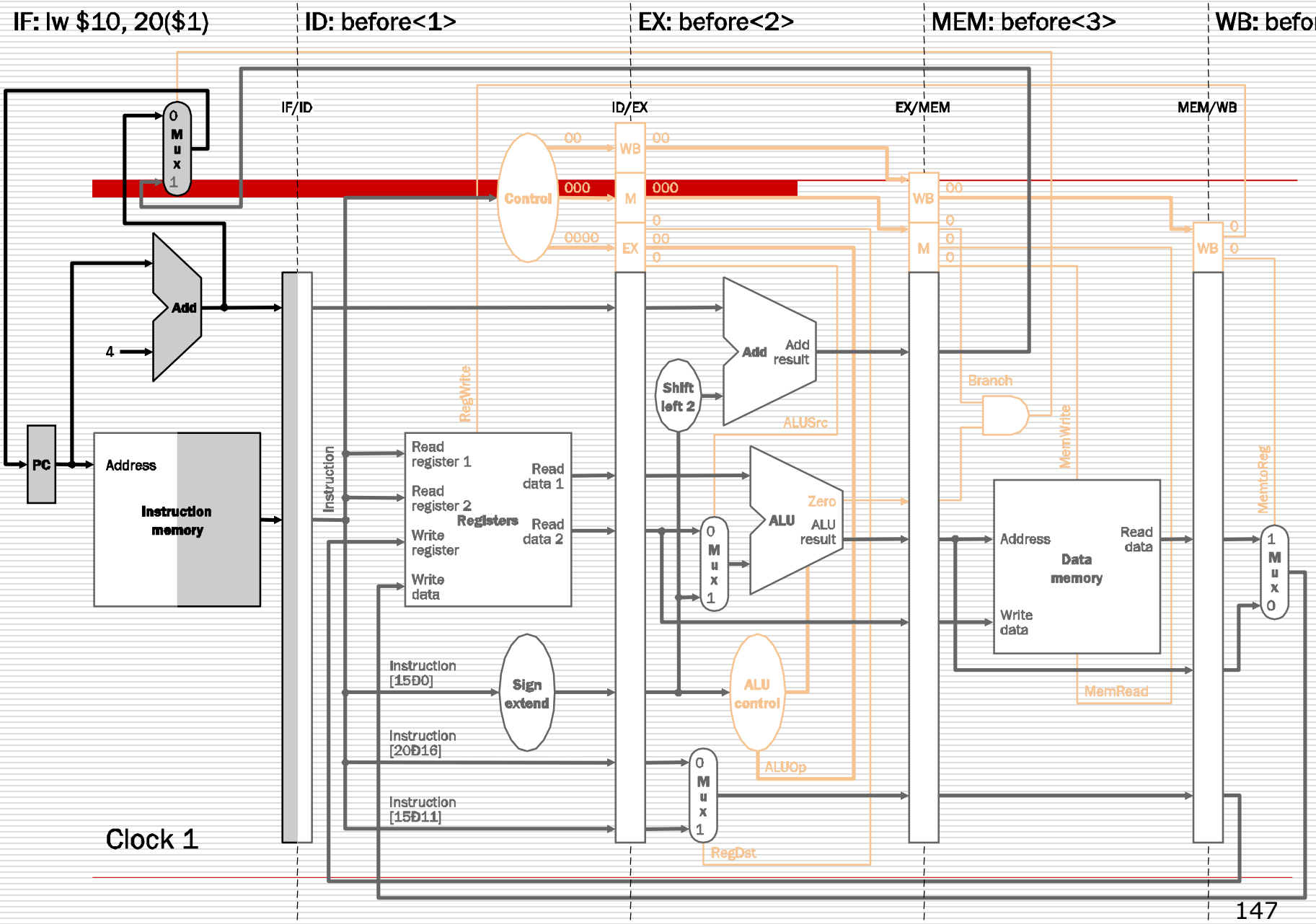


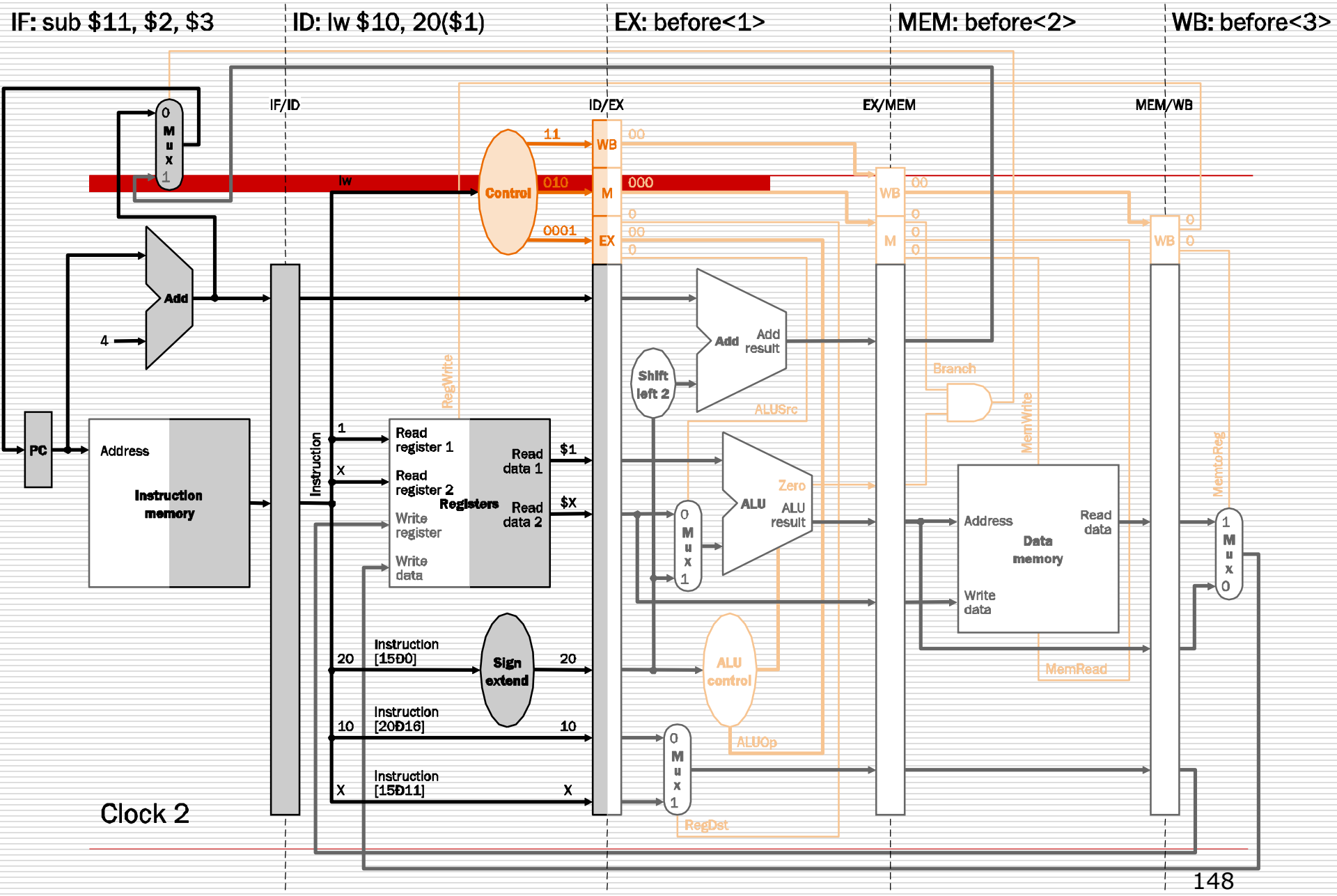
Pipelined Control

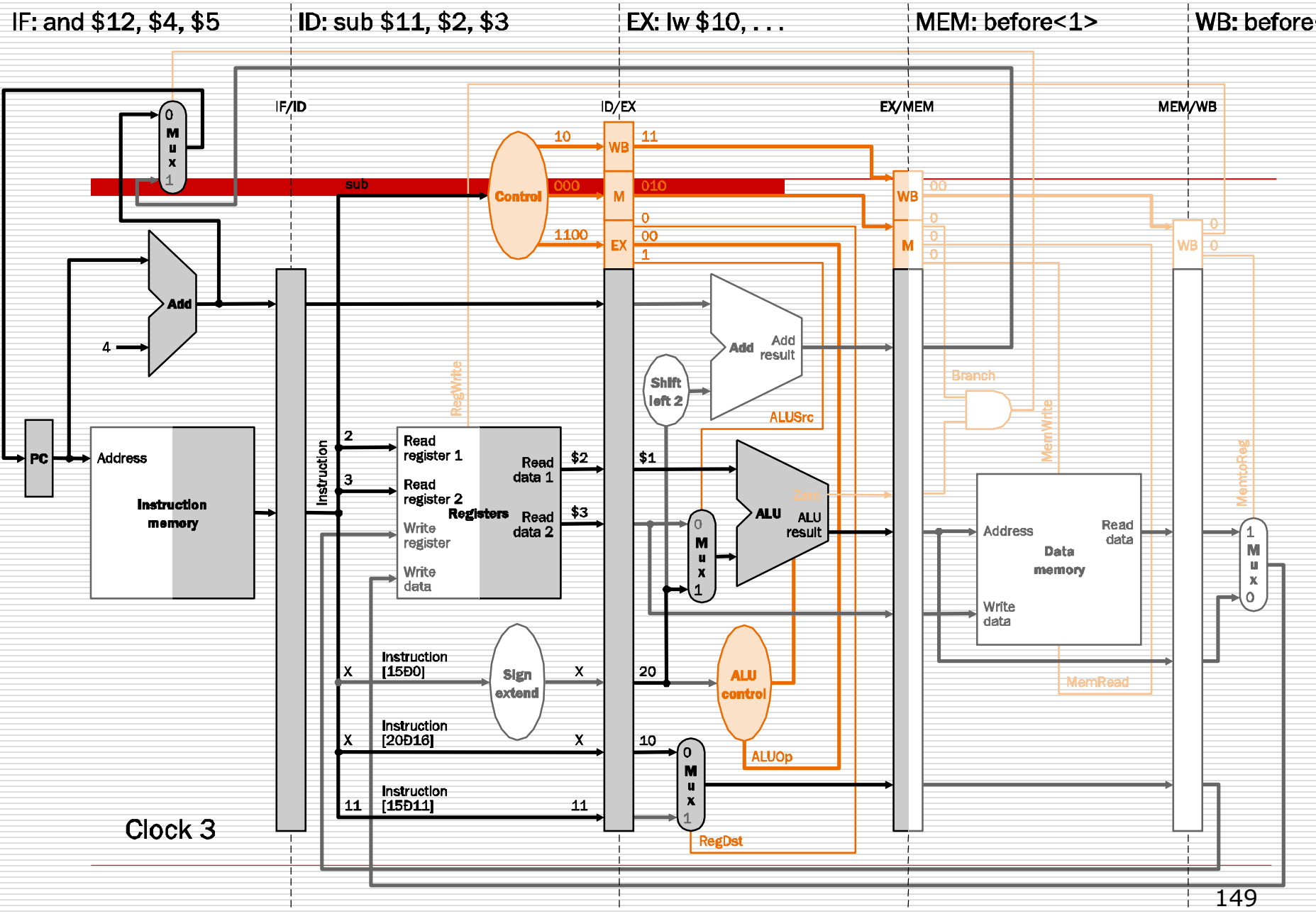


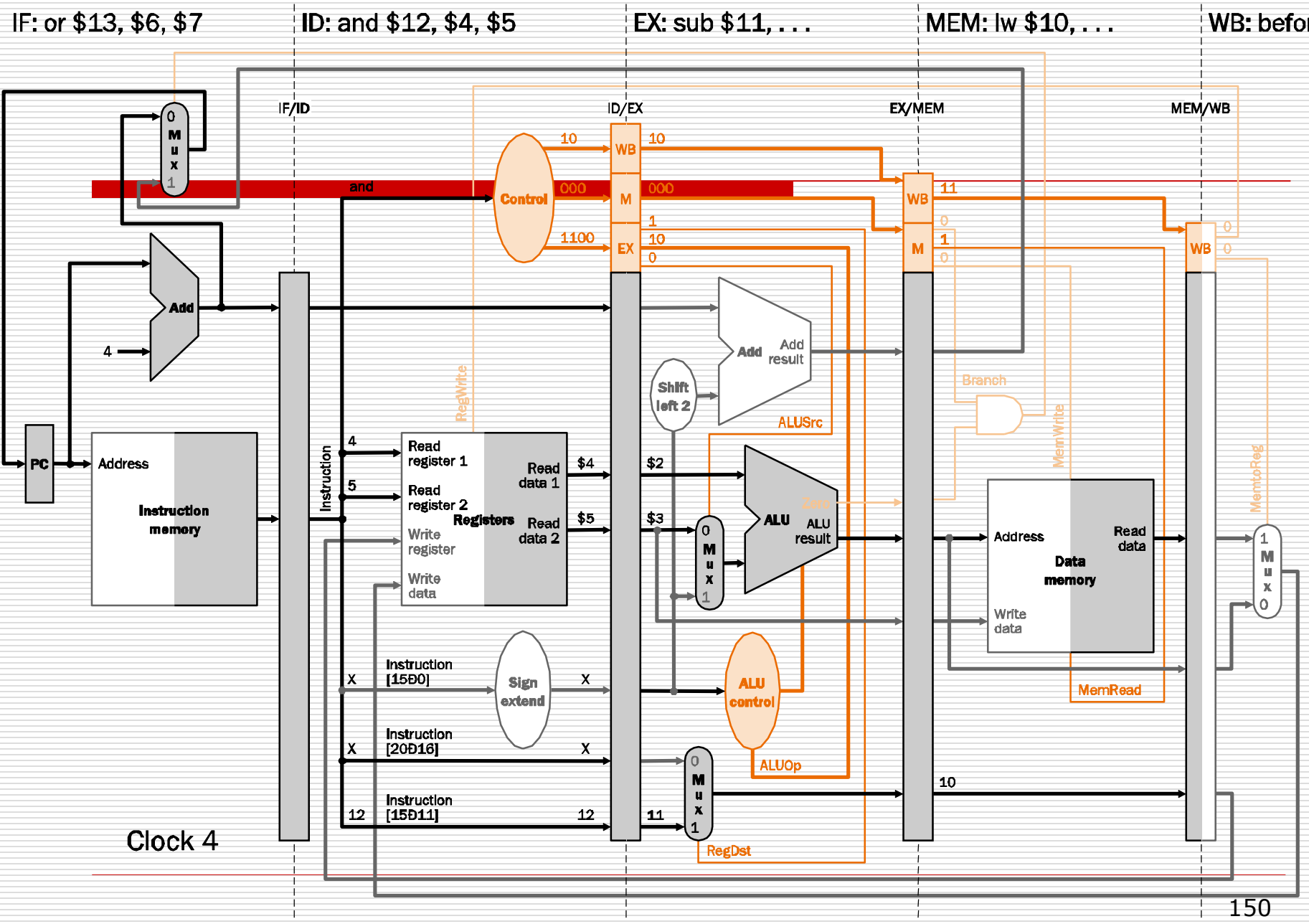
Example

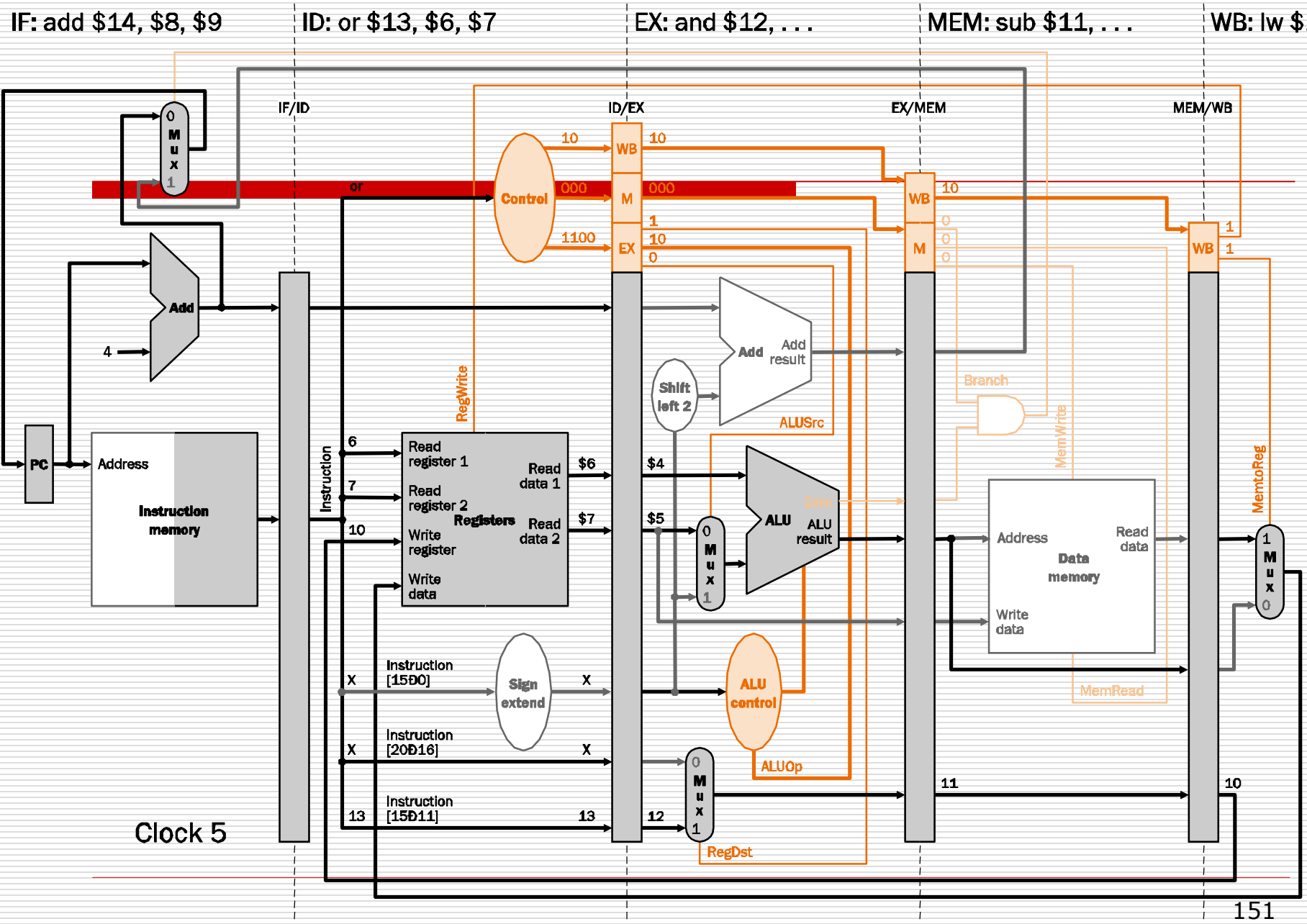
- lw $10, 20($1)
- sub $11, $2, $3
- and $12, $4, $5
- or $13, $6, $7
- add $14, $8, $9

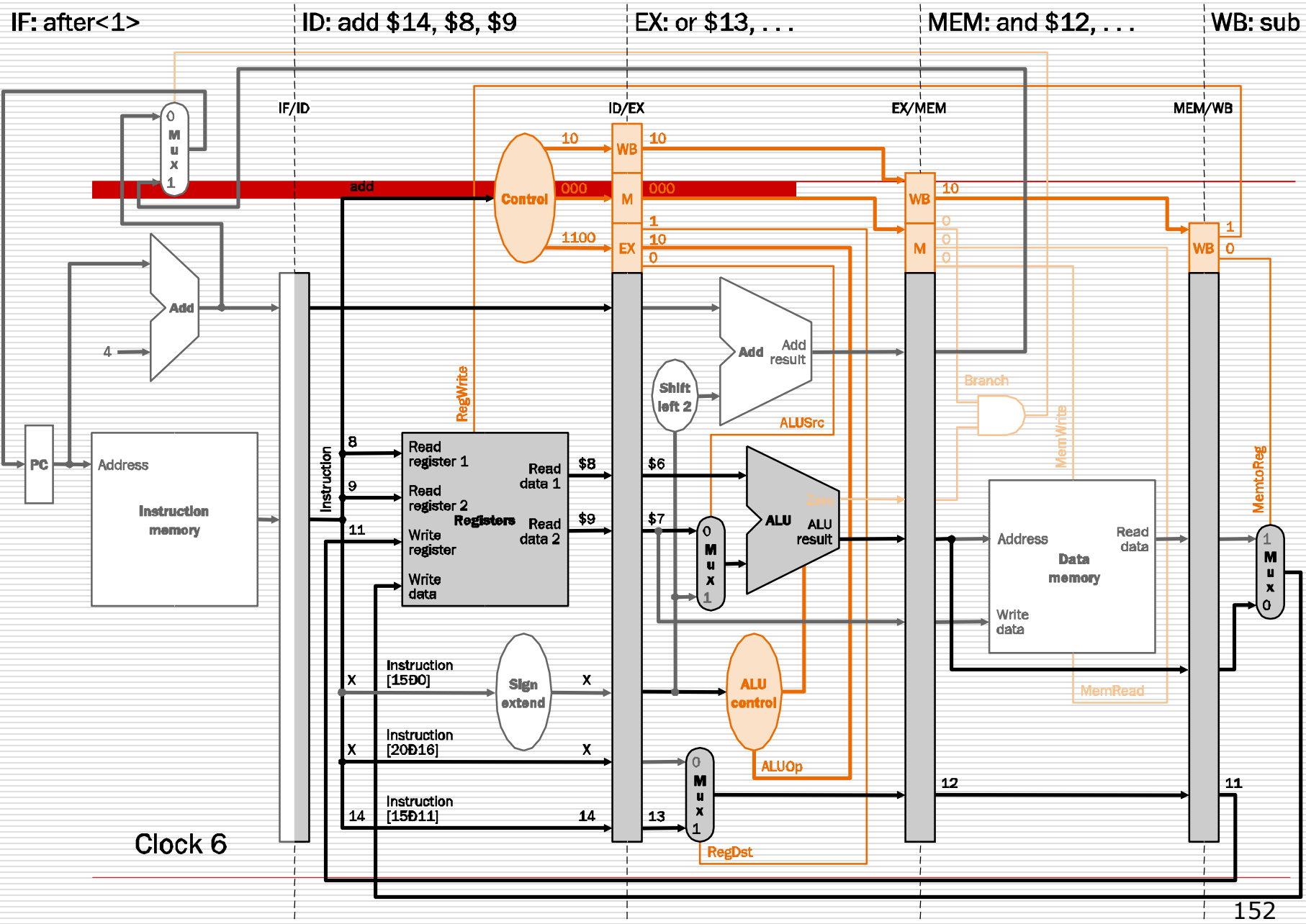

IF: lw $10, 20($1)
ID: before<1>
EX: before<2>
MEM: before<3>
WB: before<4>
IF/ID
ID/EX
EX/MEM
MEM/WB
0 M u x 1
Add
4
PC
Address
Instruction memory
Instruction
Control
000
0000
WB
M
EX
00
0
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
RegWrite
Add
Add result
Shift left 2
ALUSrc
Zero
ALU
ALU result
Branch
MemWrite
MemtoReg
Address
Data memory
Read data
Write data
MemRead
1 M u x 0
Instruction [15Đ0]
Sign extend
Instruction [20Đ16]
Instruction [15Đ11]
ALU control
ALUOp
RegDst
Clock 1

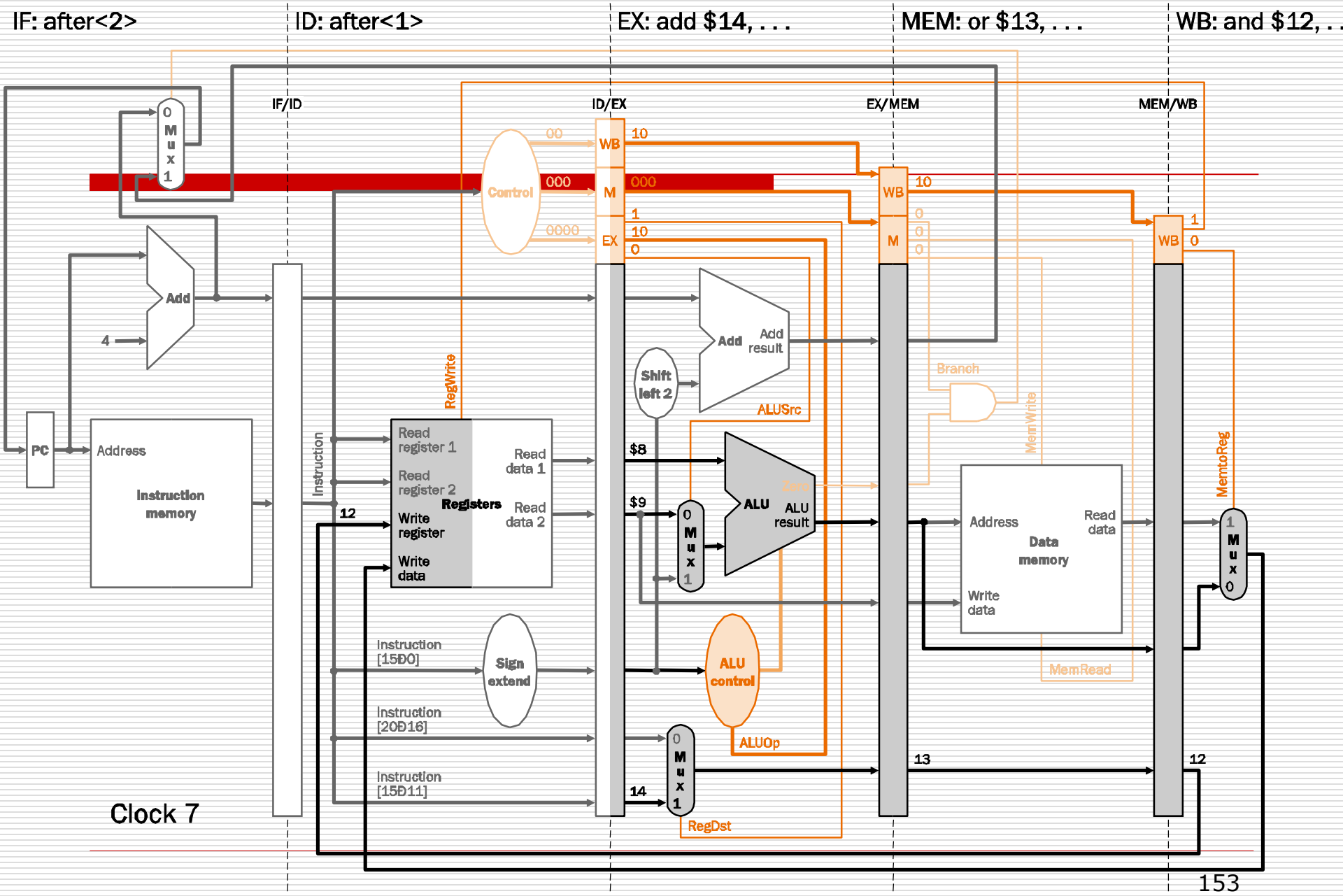

IF: sub $11, $2, $3
ID: lw $10, 20($1)
EX: before<1>
MEM: before<2>
WB: before<3>
IF/ID
ID/EX
EX/MEM
MEM/WB
M u x
0
1
Add
4
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
$1
$X
RegWrite
lw
Control
11
010
0001
WB
M
EX
000
00
Add
Add result
Shift left 2
ALUSrc
Zero
ALU
ALU result
ALU control
ALUOp
RegDst
Branch
MemWrite
MemRead
MemtoReg
Address
Data memory
Read data
Write data
Instruction [15Đ0]
Instruction [20Đ16]
Instruction [15Đ11]
Sign extend
20
10
X
Clock 2

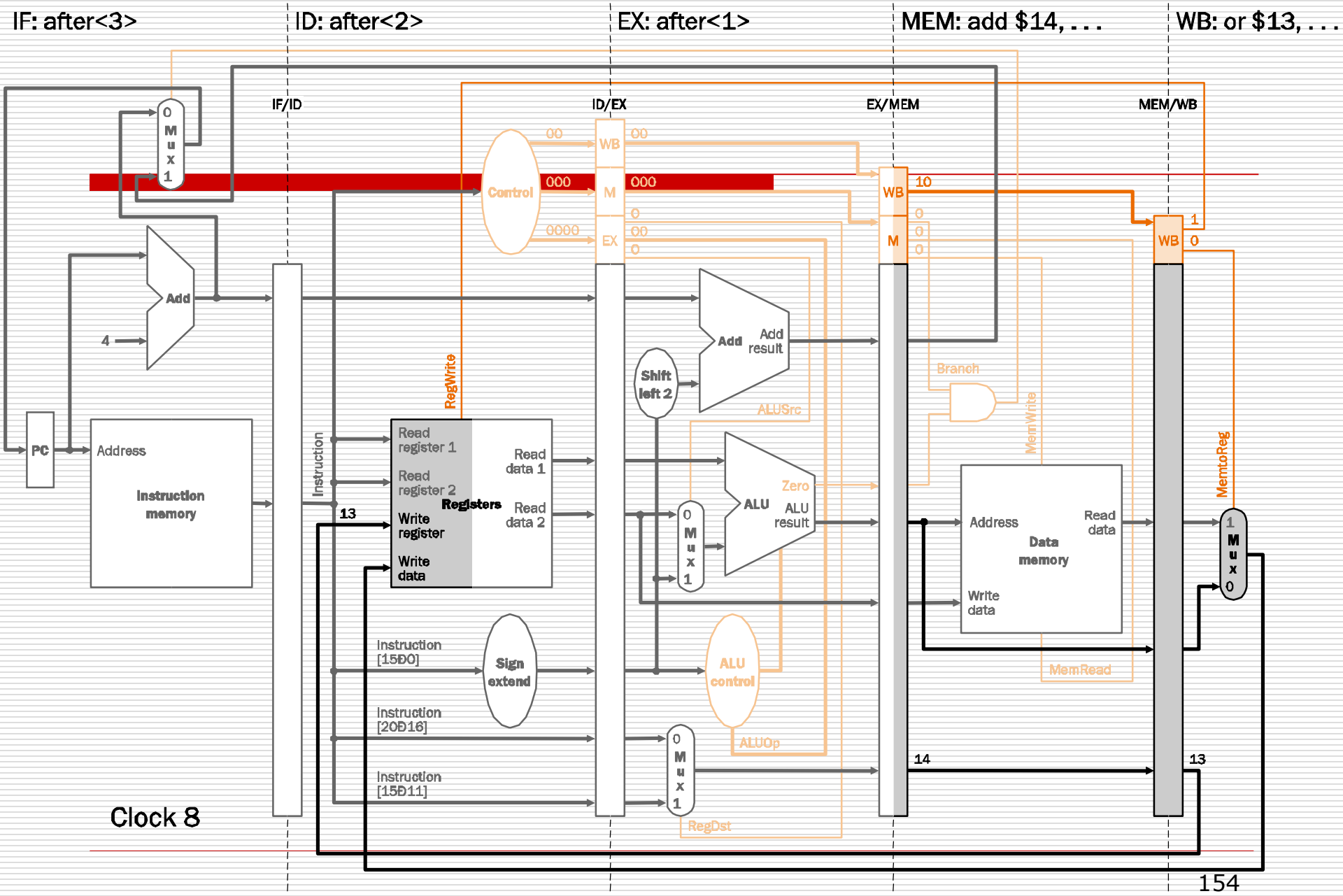

IF: and $12, $4, $5
ID: sub $11, $2, $3
EX: lw $10, . . .
MEM: before<1>
WB: before<2>
IF/ID
ID/EX
EX/MEM
MEM/WB
sub
Control
10
000
1100
WB
M
EX
11
010
00
1
0
Add
4
PC
Address
Instruction memory
Instruction
2
3
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
$2
$3
RegWrite
Add
Add result
Shift left 2
ALUSrc
$1
ALU
ALU result
Zero
Branch
MemWrite
MemtoReg
Address
Data memory
Read data
Write data
MemRead
Instruction [15Đ0]
Instruction [20Đ16]
Instruction [15Đ11]
Sign extend
X
20
10
11
ALU control
ALUOp
RegDst
Clock 3

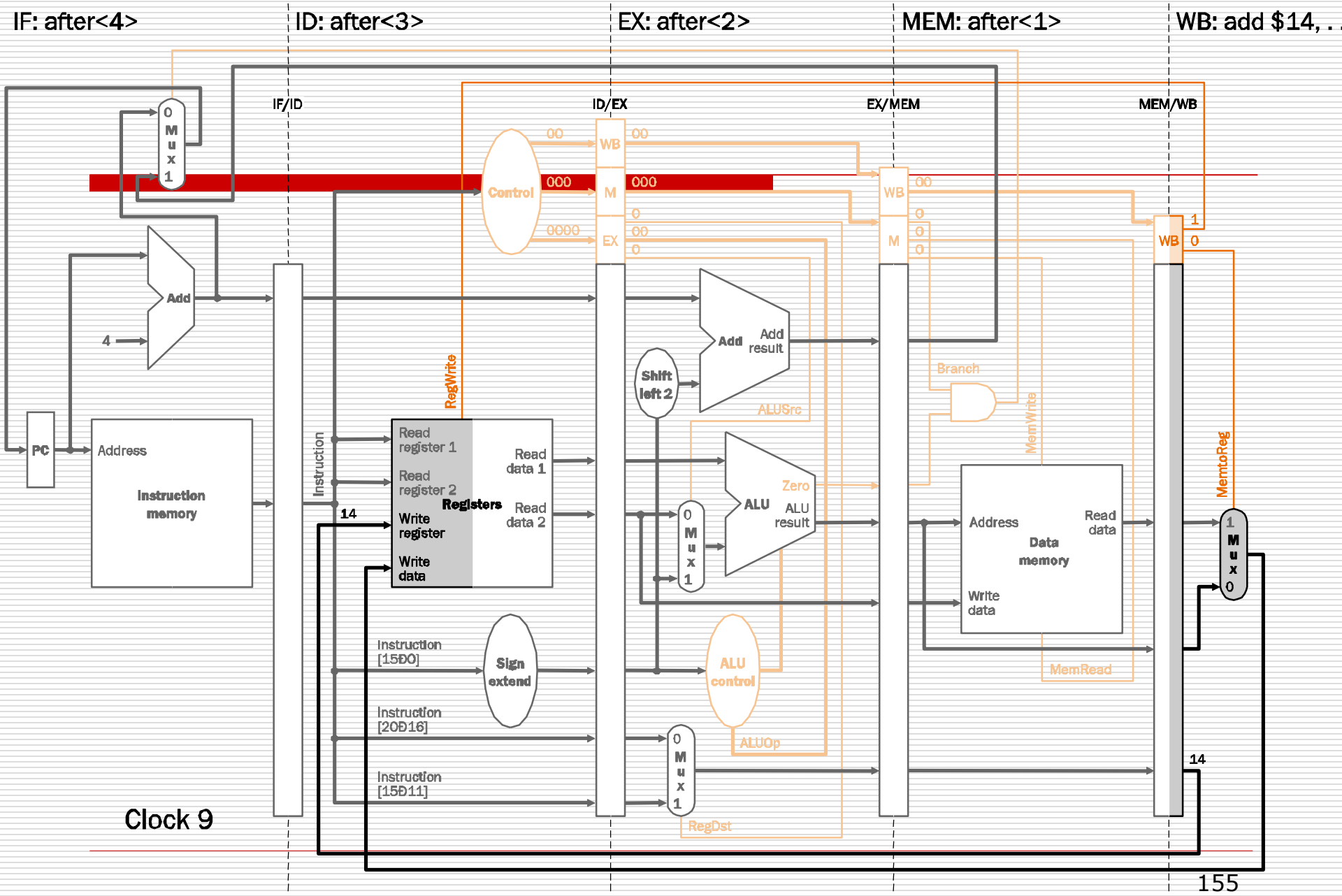

IF: or $13, $6, $7
ID: and $12, $4, $5
EX: sub $11, . . .
MEM: lw $10, . . .
WB: before<1>
IF/ID
ID/EX
EX/MEM
MEM/WB
and
Control
10
000
1100
WB
M
EX
Add
4
PC
Address
Instruction memory
Instruction
RegWrite
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
$4
$5
Shift left 2
Add result
ALUSrc
$2
$3
ALU
ALU result
Zero
Branch
MemWrite
MemtoReg
Address
Data memory
Read data
Write data
MemRead
Instruction [15Đ0]
Instruction [20Đ16]
Instruction [15Đ11]
Sign extend
ALU control
ALUOp
RegDst
Clock 4
150


IF: add $14, $8, $9
ID: or $13, $6, $7
EX: and $12, . . .
MEM: sub $11, . . .
WB: lw $10, . . .
IF/ID
ID/EX
EX/MEM
MEM/WB
Clock 5
PC
Address
Instruction memory
Add
4
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
RegWrite
Control
WB
M
EX
Shift left 2
Add result
ALUSrc
ALU
ALU result
Zero
Branch
MemWrite
Address
Data memory
Read data
Write data
MemRead
MemtoReg
Instruction [15Đ0]
Instruction [20Đ16]
Instruction [15Đ11]
Sign extend
ALU control
ALUOp
RegDst
Mux
151


IF: after<1>
ID: add $14, $8, $9
EX: or $13, . . .
MEM: and $12, . . .
WB: sub $11, . .
IF/ID
ID/EX
EX/MEM
MEM/WB
0 M u x 1
add
Control
10
000
1100
WB
M
EX
10
000
1
10
0
WB
M
10
0
0
0
WB
1
0
Add
4
PC
Address
Instruction memory
Instruction
8
9
11
RegWrite
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
$8
$9
Add
Add result
Shift left 2
ALUSrc
$6
$7
ALU
ALU result
Zero
Branch
MemWrite
MemtoReg
Address
Data memory
Read data
Write data
MemRead
1 M u x 0
Instruction [15Đ0]
Instruction [20Đ16]
Instruction [15Đ11]
X
X
X
X
Sign extend
ALU control
ALUOp
14
14
13
12
11
RegDst
Clock 6


IF: after<2>
ID: after<1>
EX: add $14, . . .
MEM: or $13, . . .
WB: and $12, . . .
IF/ID
ID/EX
EX/MEM
MEM/WB
M u x
Add
4
PC
Address
Instruction memory
Instruction
RegWrite
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
12
Control
00
000
0000
WB
M
EX
10
1
0
Add
Add result
Shift left 2
ALUSrc
$8
$9
ALU
ALU result
Zero
Instruction [15Đ0]
Sign extend
Instruction [20Đ16]
Instruction [15Đ11]
ALU control
ALUOp
14
RegDst
Branch
MemWrite
Address
Data memory
Read data
Write data
MemRead
13
MemtoReg
Clock 7


IF: after<3>
ID: after<2>
EX: after<1>
MEM: add $14, . . .
WB: or $13, . . .
IF/ID
ID/EX
EX/MEM
MEM/WB
Clock 8
PC
Address
Instruction memory
Add
4
Instruction
RegWrite
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
13
Control
000
0000
WB
M
EX
00
Shift left 2
Add
Add result
ALUSrc
Zero
ALU
ALU result
Instruction [15Ð0]
Sign extend
Instruction [20Ð16]
Instruction [15Ð11]
ALU control
ALUOp
RegDst
Branch
MemWrite
MemRead
Address
Data memory
Read data
Write data
MemtoReg
14
10
1
0


IF: after<4>
ID: after<3>
EX: after<2>
MEM: after<1>
WB: add $14, . .
IF/ID
ID/EX
EX/MEM
MEM/WB
M u x
0
1
Add
4
PC
Address
Instruction memory
Instruction
14
RegWrite
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Control
00
000
0000
WB
M
EX
Shift left 2
Add
Add result
ALUSrc
Zero
ALU
ALU result
Branch
MemWrite
MemtoReg
Address
Data memory
Read data
Write data
MemRead
Instruction [15Ð0]
Instruction [20Ð16]
Instruction [15Ð11]
Sign extend
ALU control
ALUOp
RegDst
Clock 9

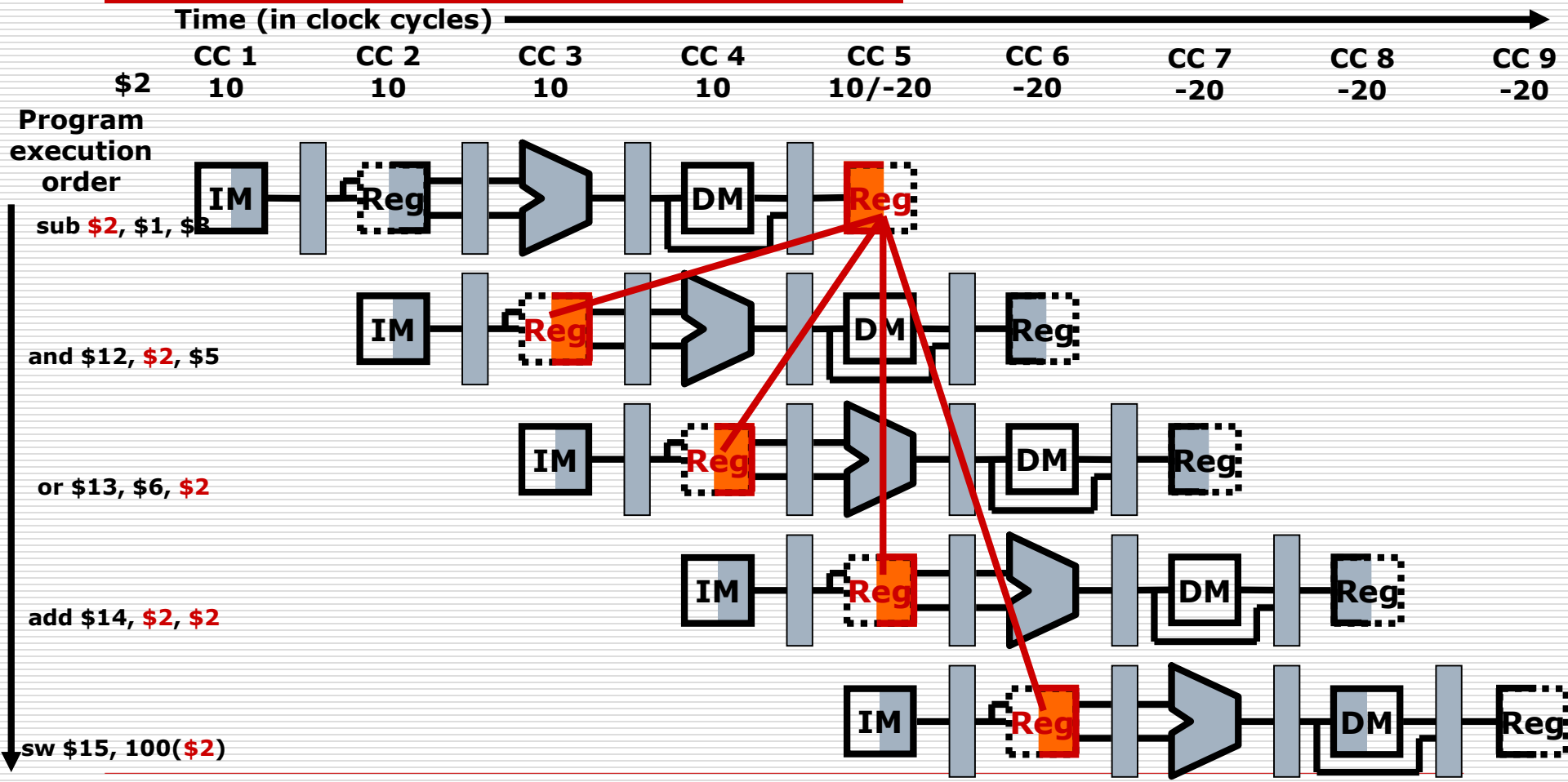
Data Hazards in ALU Instructions

- Problem with starting next instruction before first is finished
  - dependencies that "go backward in time" are data hazards

- How about the following example?

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

Dependencies



Stalling

- Have compiler guarantee no hazards
- Where do we insert the “nops” ?

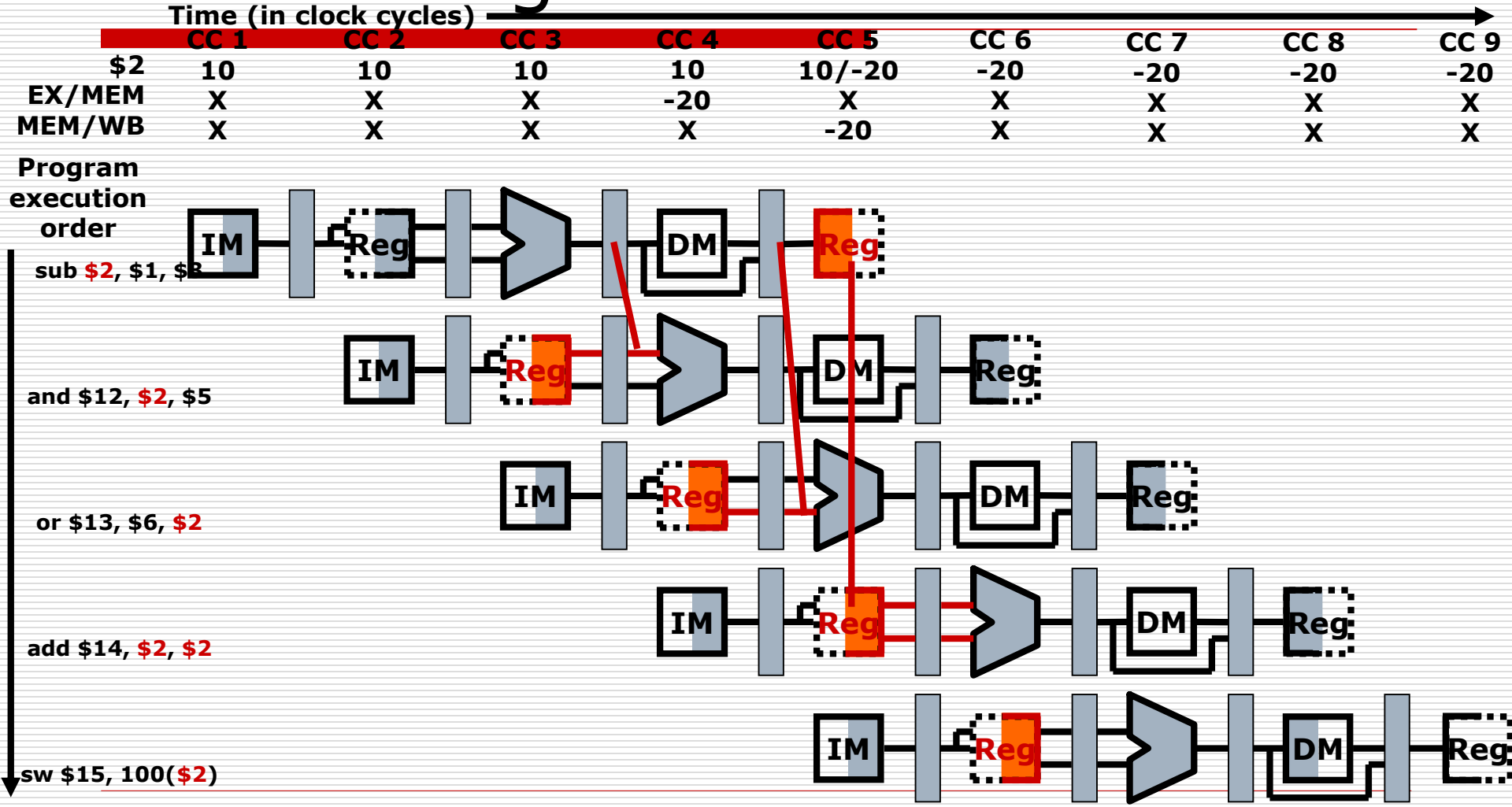
```
sub   $2, $1, $3
nop
nop
and   $12, $2, $5
or    $13, $6, $2
add   $14, $2, $2
sw    $15, 100($2)
```

- Problem: this really slows us down!

Forwarding

- Use temporary results, don't wait for them to be written
  - register file forwarding to handle read/write to same register
  - ALU forwarding

Forwarding


Time (in clock cycles)
CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
CC 7
CC 8
CC 9
$2 10 10 10 10 10/-20 -20 -20 -20 -20
EX/MEM X X X -20 X X X X X
MEM/WB X X X X -20 X X X X
Program execution order
sub $2, $1, $3
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15, 100($2)
IM
Reg
DM

Forwarding Unit – EX Hazard

- if (EX/MEM.RegWrite
  and (EX/MEM.RegisterRd≠0)
  and (EX/MEM.RegisterRd=ID/EX.RegisterRs)
  then ForwardA=10

- if (EX/MEM.RegWrite
  and (EX/MEM.RegisterRd≠0)
  and (EX/MEM.RegisterRd=ID/EX.RegisterRt)
  then ForwardB=10

Forwarding Unit – MEM Hazard

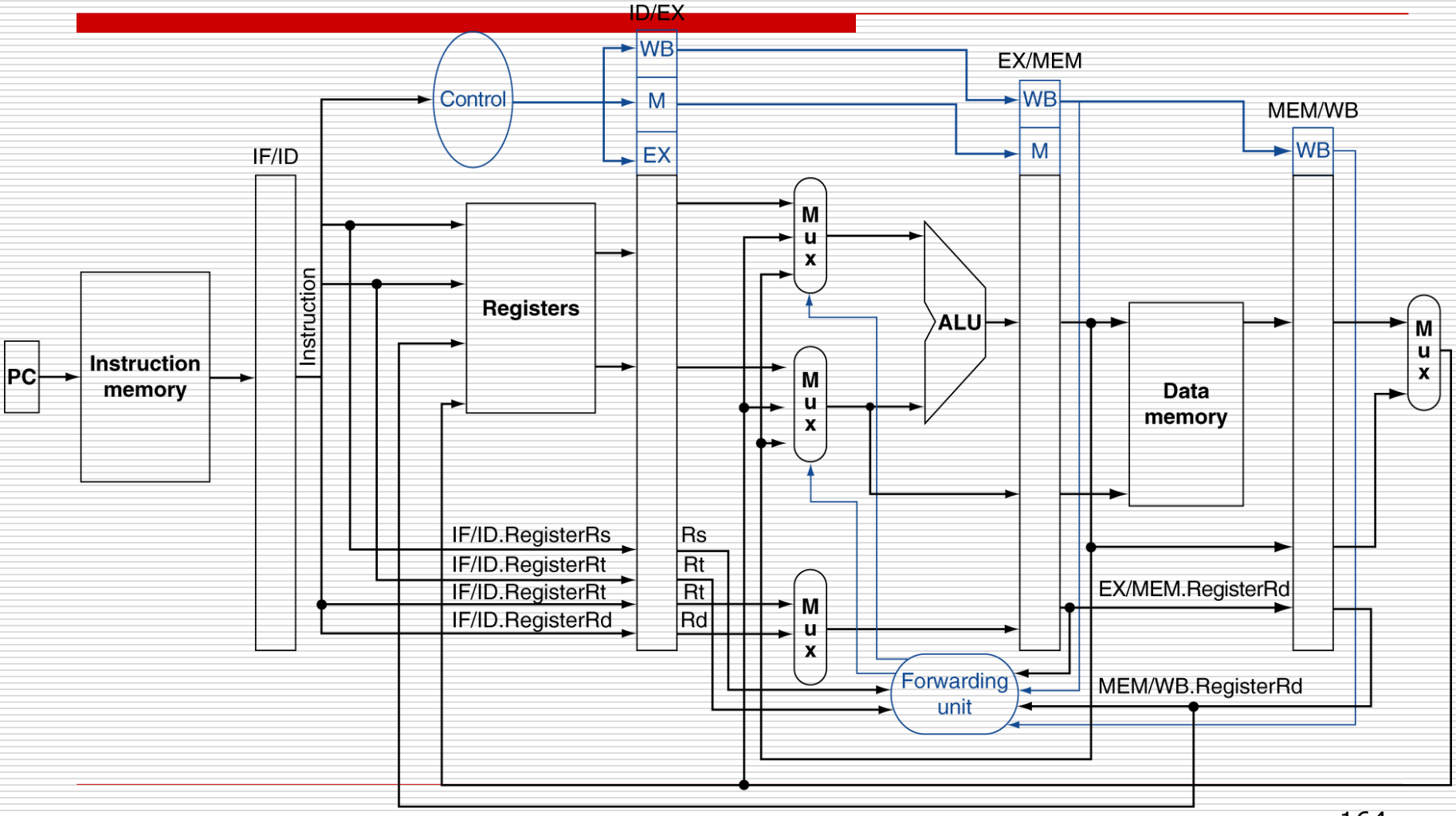
- if (MEM/WB.RegWrite
  and (MEM/WB.RegisterRd≠0)
  and (EX/MEM.RegisterRd≠ID/EX.RegisterRs)
  and (MEM/WB.RegisterRd=ID/EX.RegisterRs))
  then ForwardA=01

- if (MEM/WB.RegWrite
  and (MEM/WB.RegisterRd≠0)
  and (EX/MEM.RegisterRd≠ID/EX.RegisterRt)
  and (MEM/WB.RegisterRd=ID/EX.RegisterRt))
  then ForwardB=01

Control Values

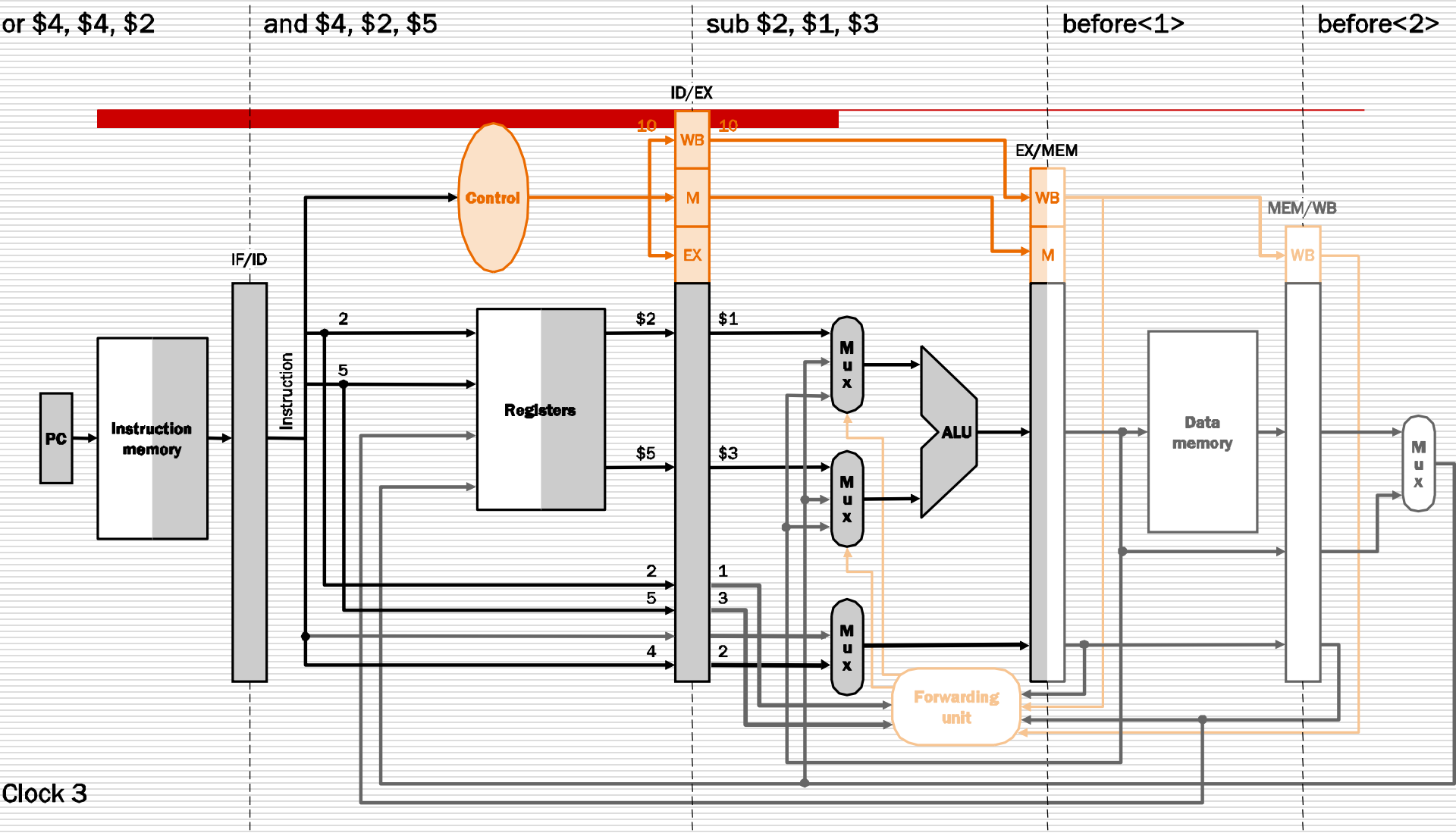
mux control	source	explanation
ForwardA=00	**ID/EX**	1st ALU operand comes from the register file
ForwardA=10	**EX/MEM**	1st ALU operand is forwarded from the prior ALU result
ForwardA=01	**MEM/WB**	1st ALU operand is forwarded from data memory or an earlier ALU result
ForwardB=00	**ID/EX**	2nd ALU operand comes from the register file
ForwardB=10	**EX/MEM**	2nd ALU operand is forwarded from the prior ALU result
ForwardB=01	**MEM/WB**	2nd ALU operand is forwarded from data memory or an earlier ALU result

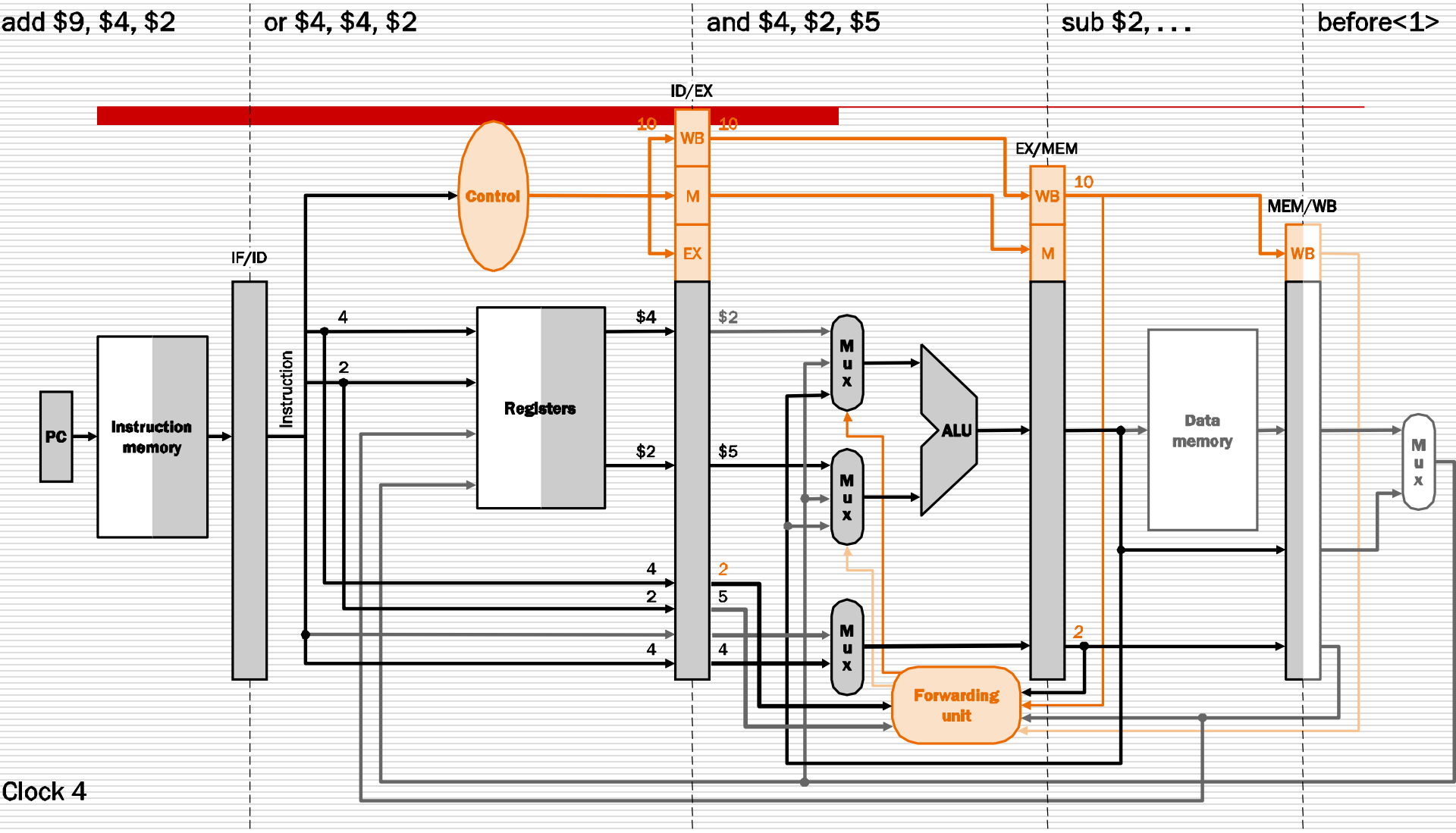
Forwarding Paths

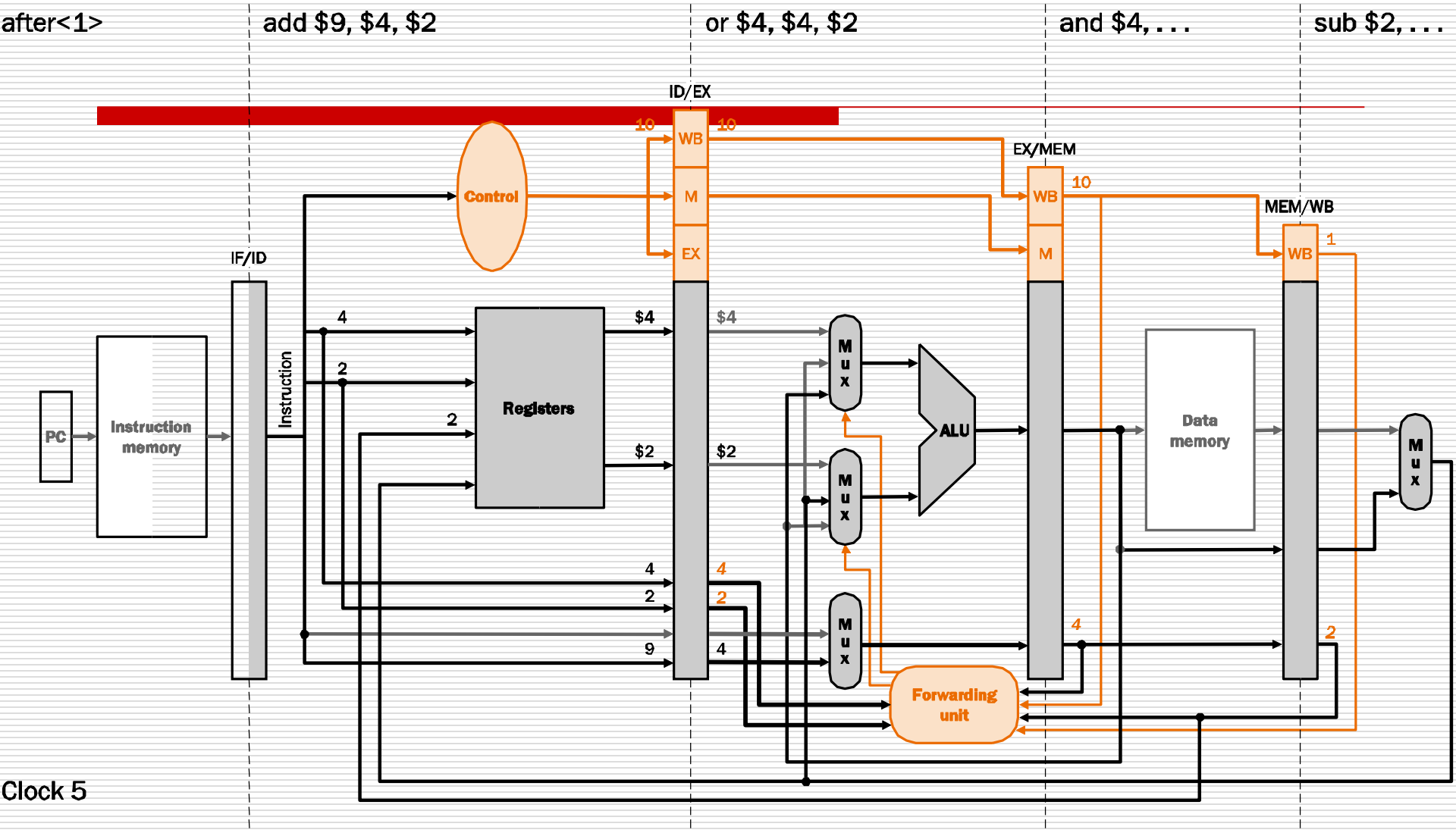


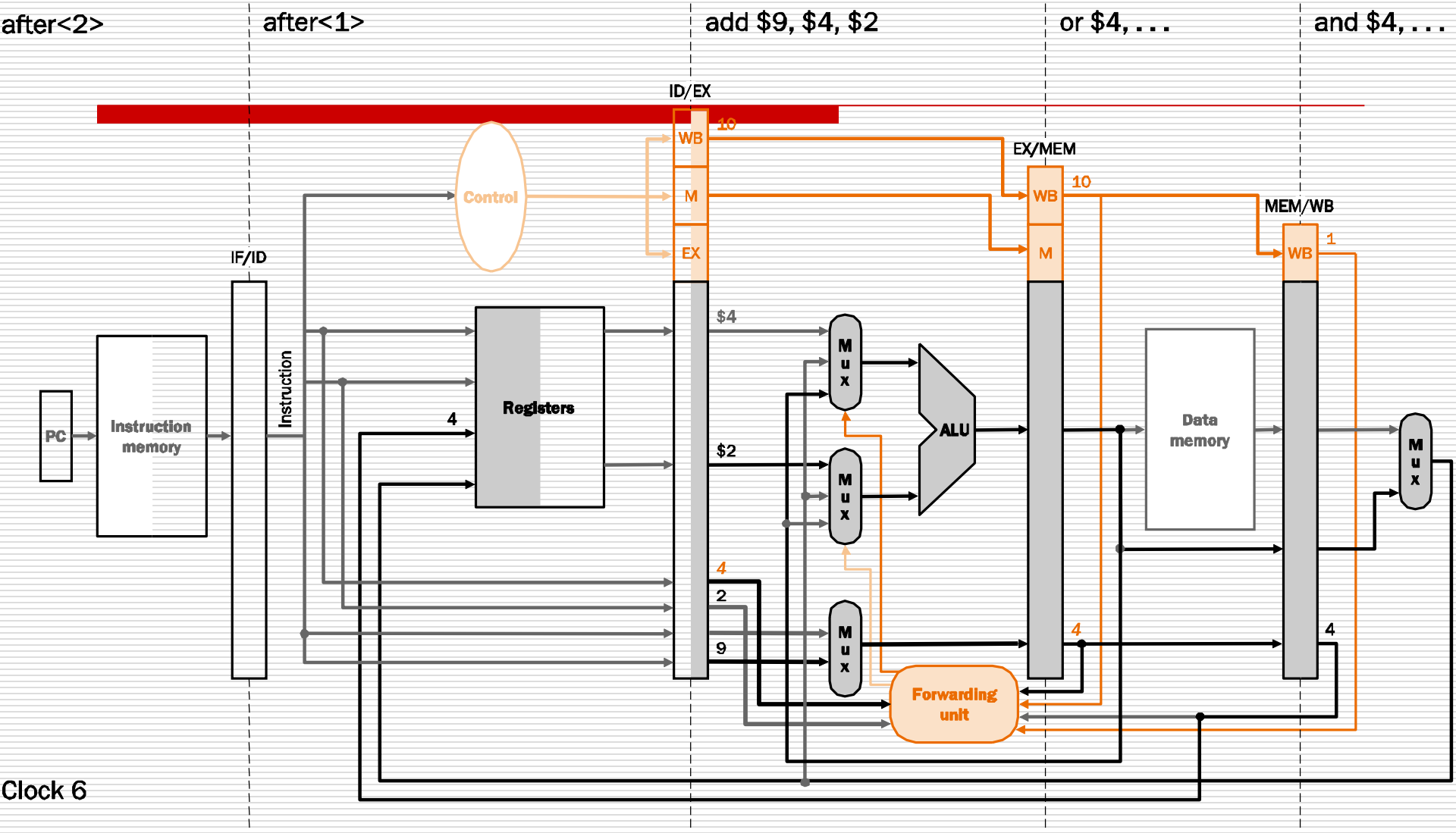
Example

- sub $2, $1, $3
- and $4, $2, $5
- or $4, $4, $2
- add $9, $4, $2


or $4, $4, $2
and $4, $2, $5
sub $2, $1, $3
before<1>
before<2>
ID/EX
10
WB
10
M
EX
Control
EX/MEM
WB
M
MEM/WB
WB
IF/ID
PC
Instruction memory
Instruction
2
5
Registers
$2
$5
$1
$3
M u x
ALU
Data memory
M u x
2
5
4
1
3
2
M u x
Forwarding unit
Clock 3


add $9, $4, $2
or $4, $4, $2
and $4, $2, $5
sub $2, . . .
before<1>
ID/EX
EX/MEM
MEM/WB
IF/ID
10
WB
M
EX
Control
PC
Instruction memory
Instruction
4
2
Registers
$4
$2
$2
$5
4
2
4
2
5
4
Mux
ALU
Data memory
Forwarding unit
Clock 4

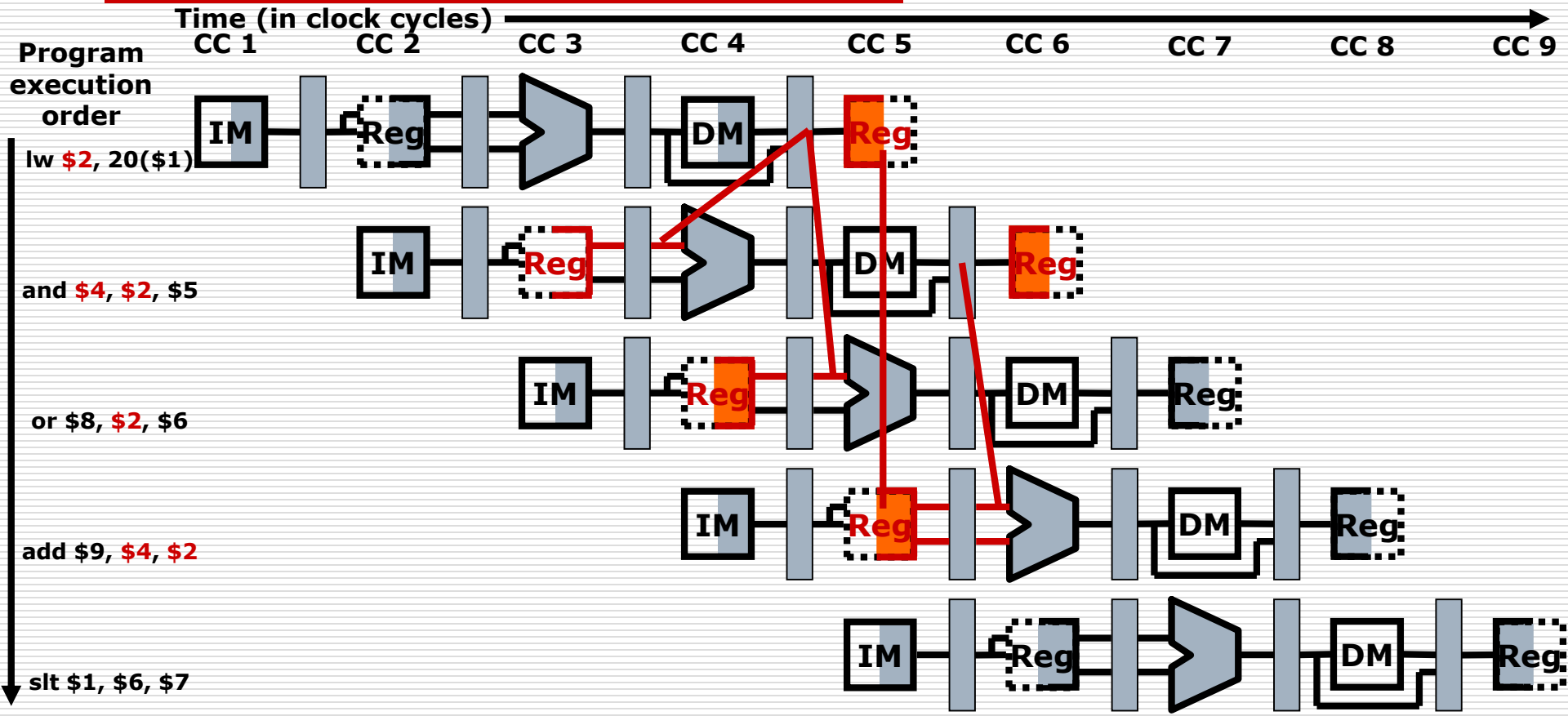

after<1>
add $9, $4, $2
or $4, $4, $2
and $4, . . .
sub $2, . . .
ID/EX
EX/MEM
MEM/WB
IF/ID
Control
WB
M
EX
10
10
10
1
PC
Instruction memory
Instruction
Registers
4
2
2
$4
$2
$4
$2
4
2
9
4
4
2
4
2
M u x
M u x
M u x
ALU
Data memory
M u x
Forwarding unit
Clock 5


after<2>
after<1>
add $9, $4, $2
or $4, . . .
and $4, . . .
ID/EX
EX/MEM
MEM/WB
WB
M
EX
10
Control
IF/ID
Instruction
PC
Instruction memory
Registers
4
$4
$2
4
2
9
Mux
ALU
Data memory
Forwarding unit
1
Clock 6

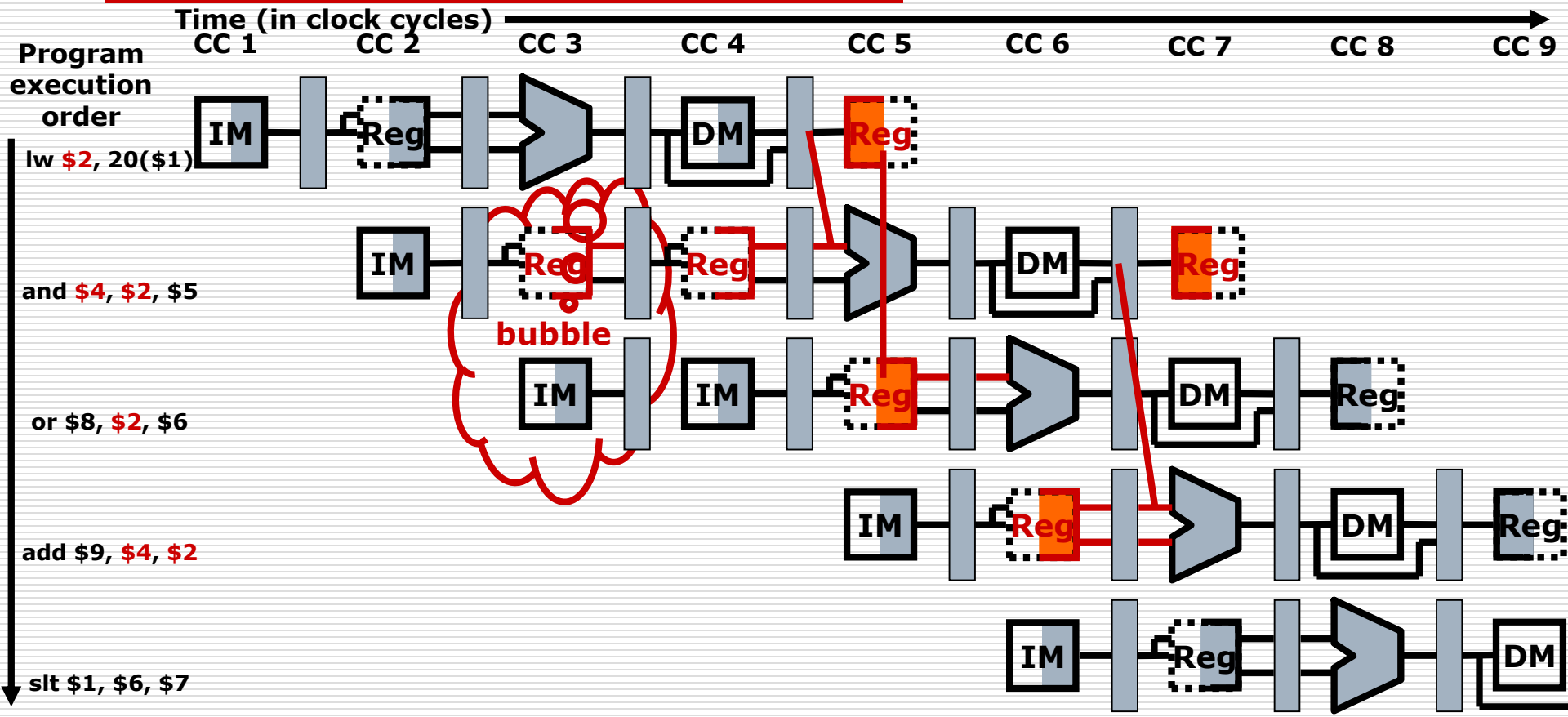
Can't always forward

- Load word can still cause a hazard:
  - an instruction tries to read a register following a load instruction that writes to the same register.
- Thus, we need a hazard detection unit to “stall” the load instruction

Load-Use Data Hazard



Stalling



we can stall the pipeline by keeping an instruction in the same stage

Load-Use Hazard Detection

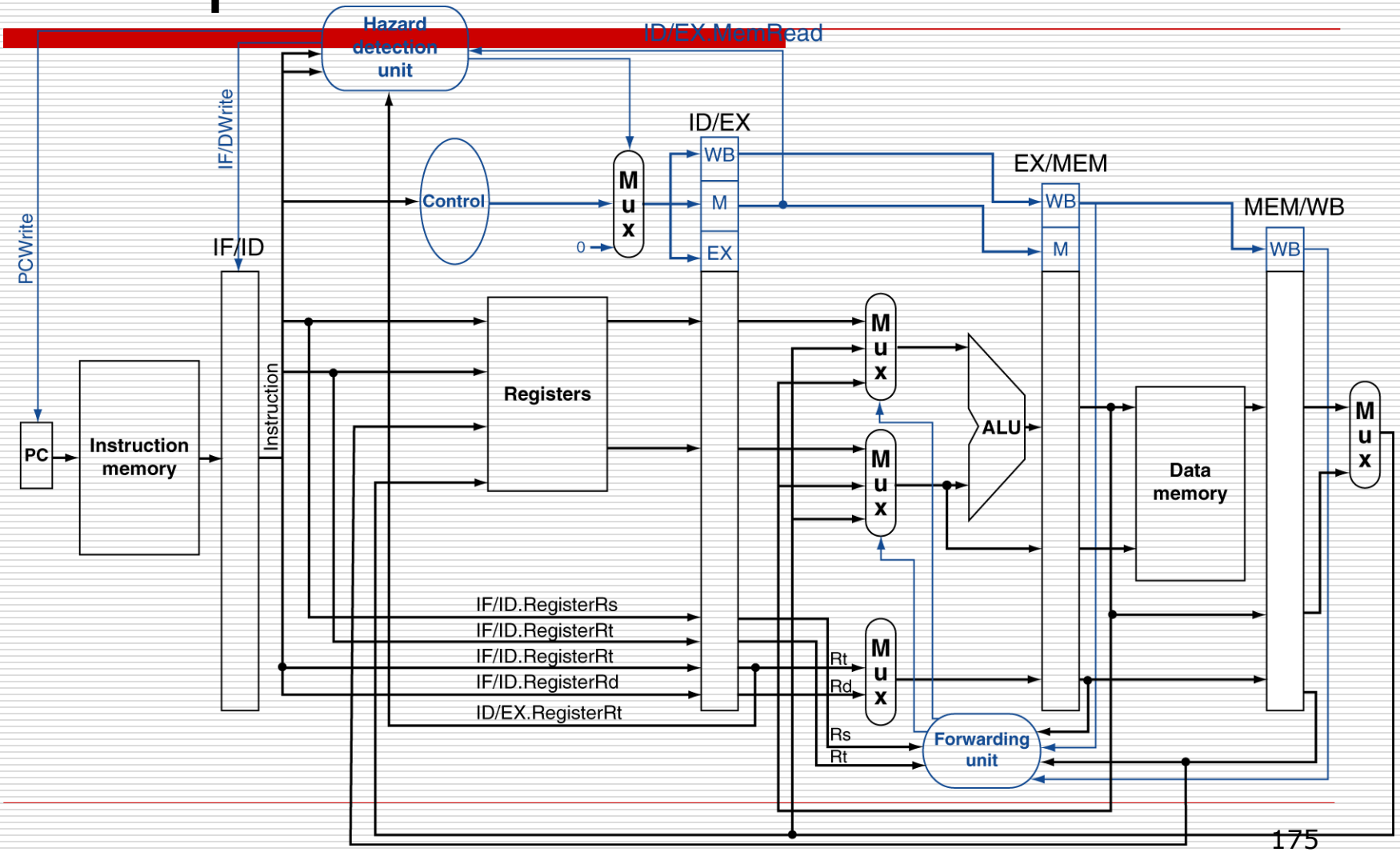
- Stall by letting an instruction that won't write anything go forward

- if (ID/EX.MemRead
  and ((ID/EX.RegisterRt=IF/ID.RegisterRs)
  or (ID/EX.RegisterRt=IF/ID.RegisterRt)))
  then stall the pipeline

How to Stall the Pipeline

- Force control values in ID/EX register to 0
  - EX, MEM and WB do nop (no-operation)
- Prevent update of PC and IF/ID register
  - Using instruction is decoded again
  - Following instruction is fetched again
  - 1-cycle stall allows MEM to read data for lw
    - Can subsequently forward to EX stage

Datapath with Hazard Detection



Example

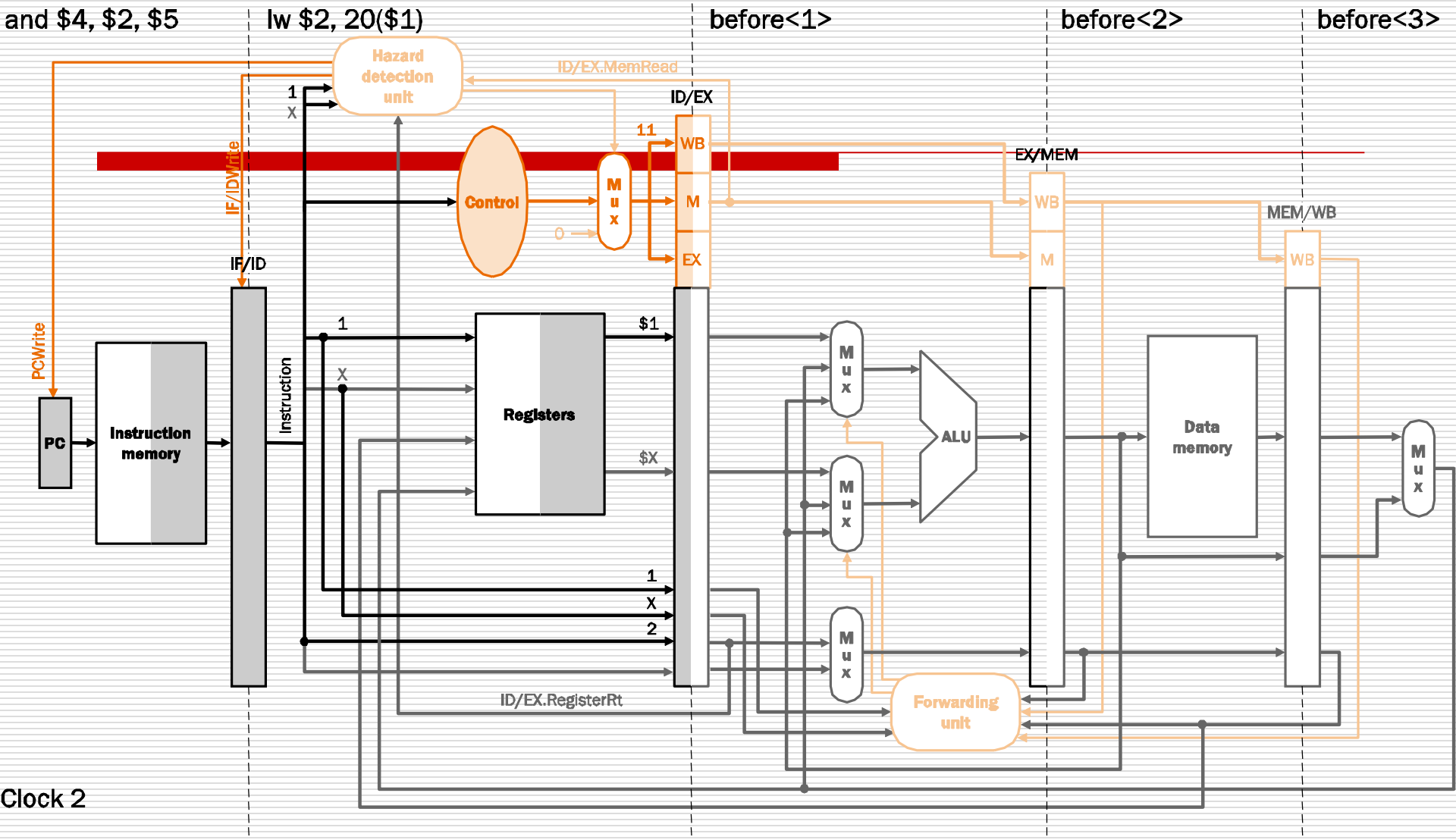
- lw $2, 20($1)
- and $4, $2, $5
- or $4, $4, $2
- add $9, $4, $2

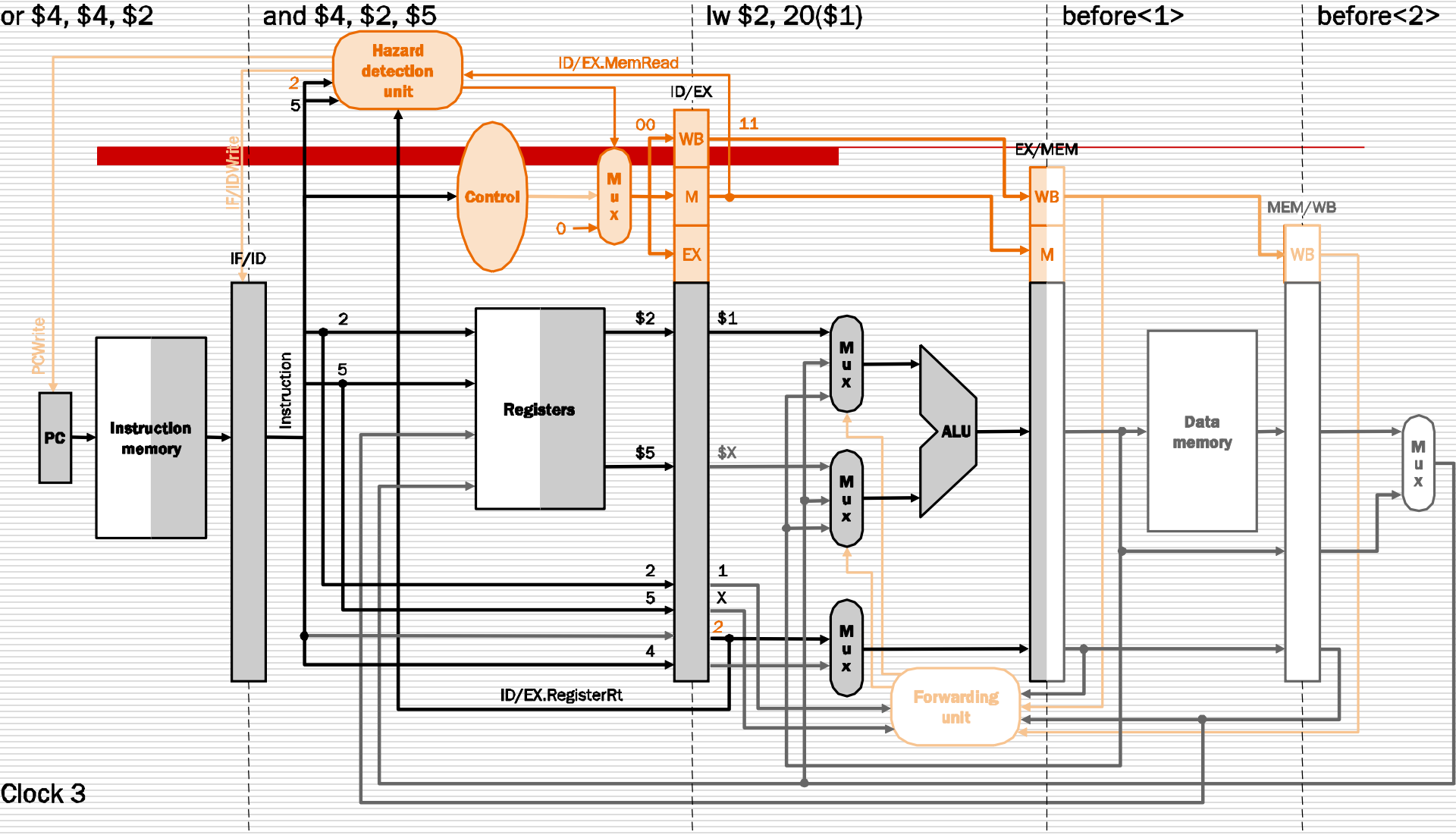
Example

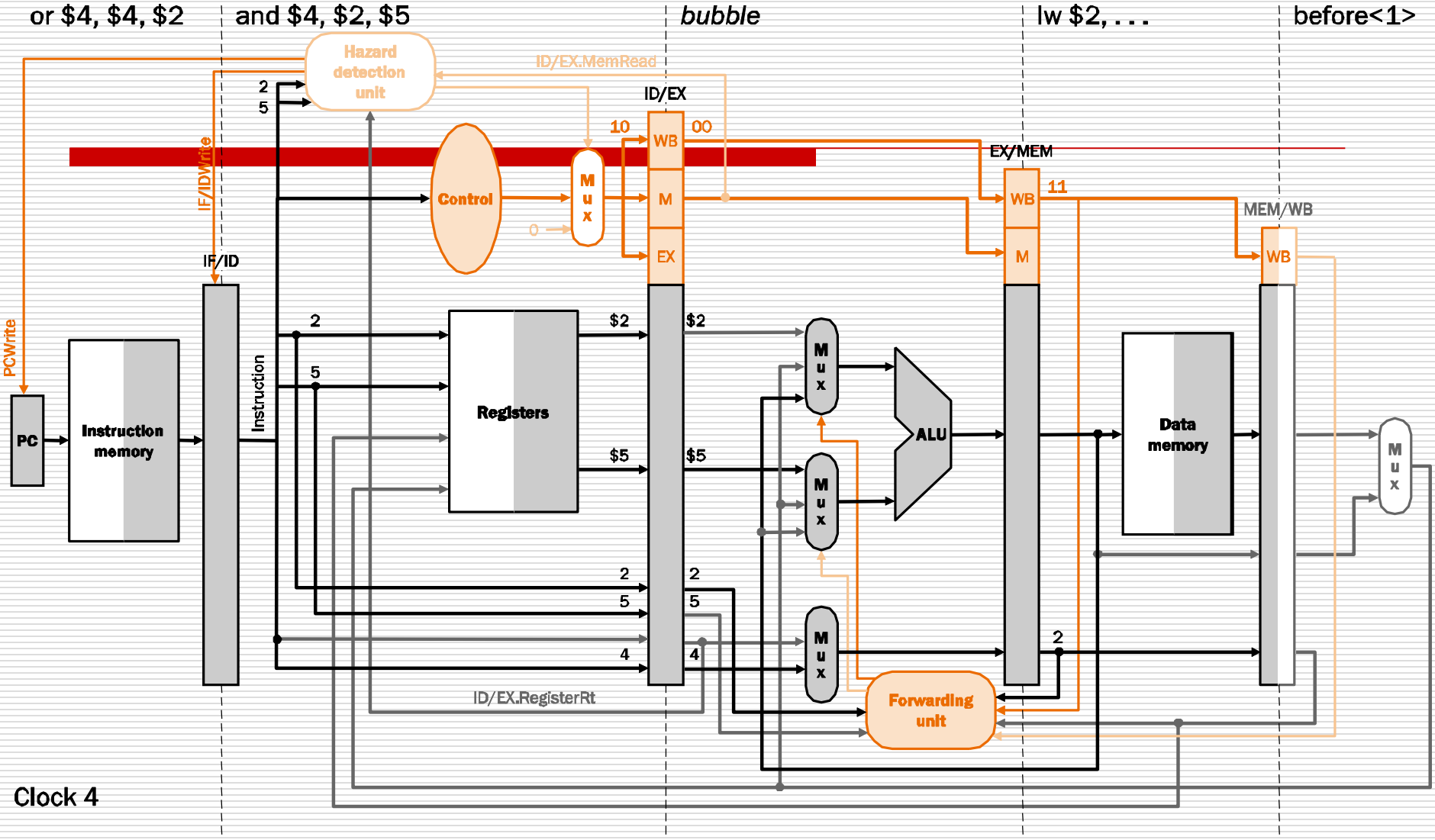
- lw $2, 20($1)

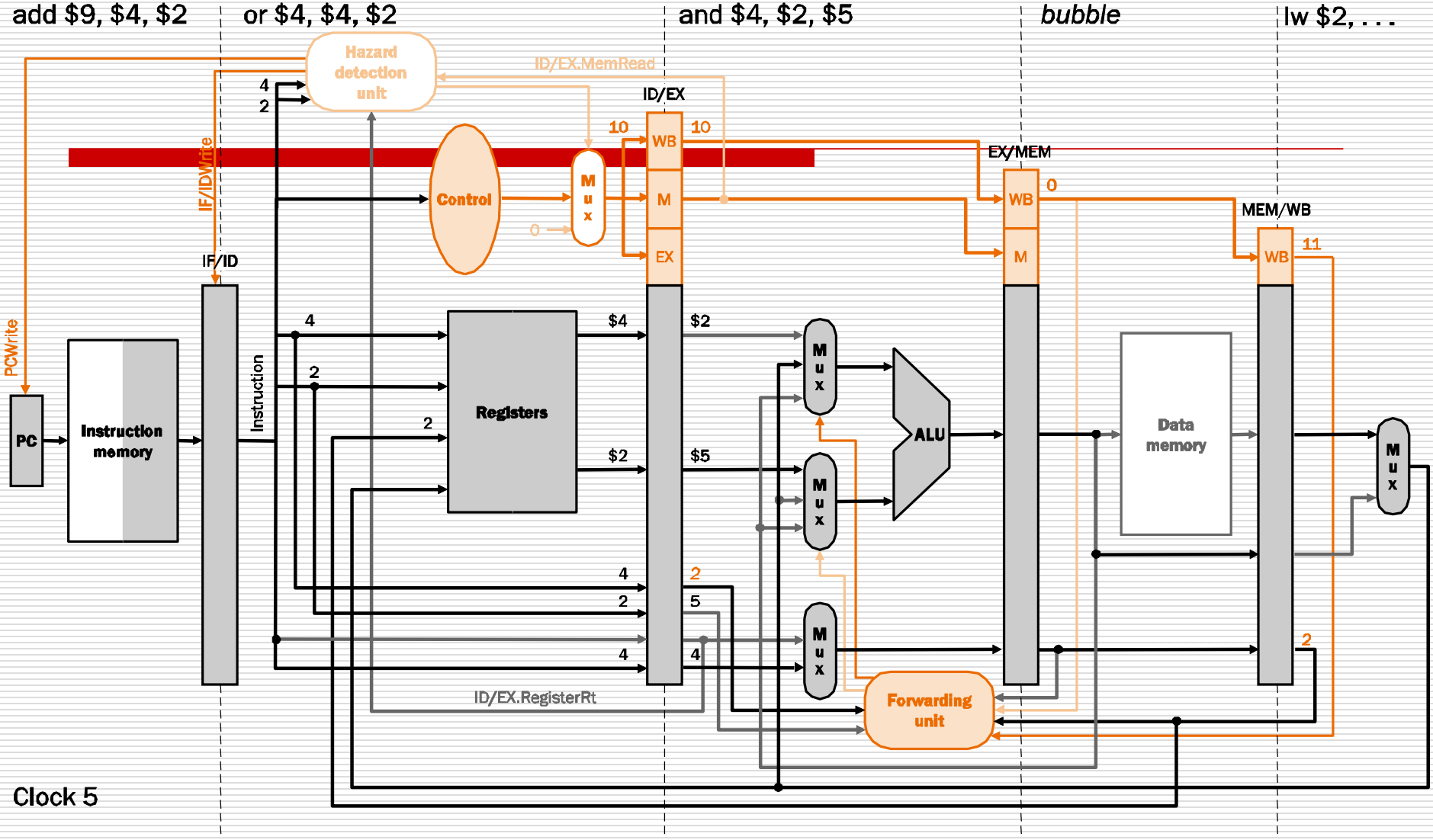
  stall

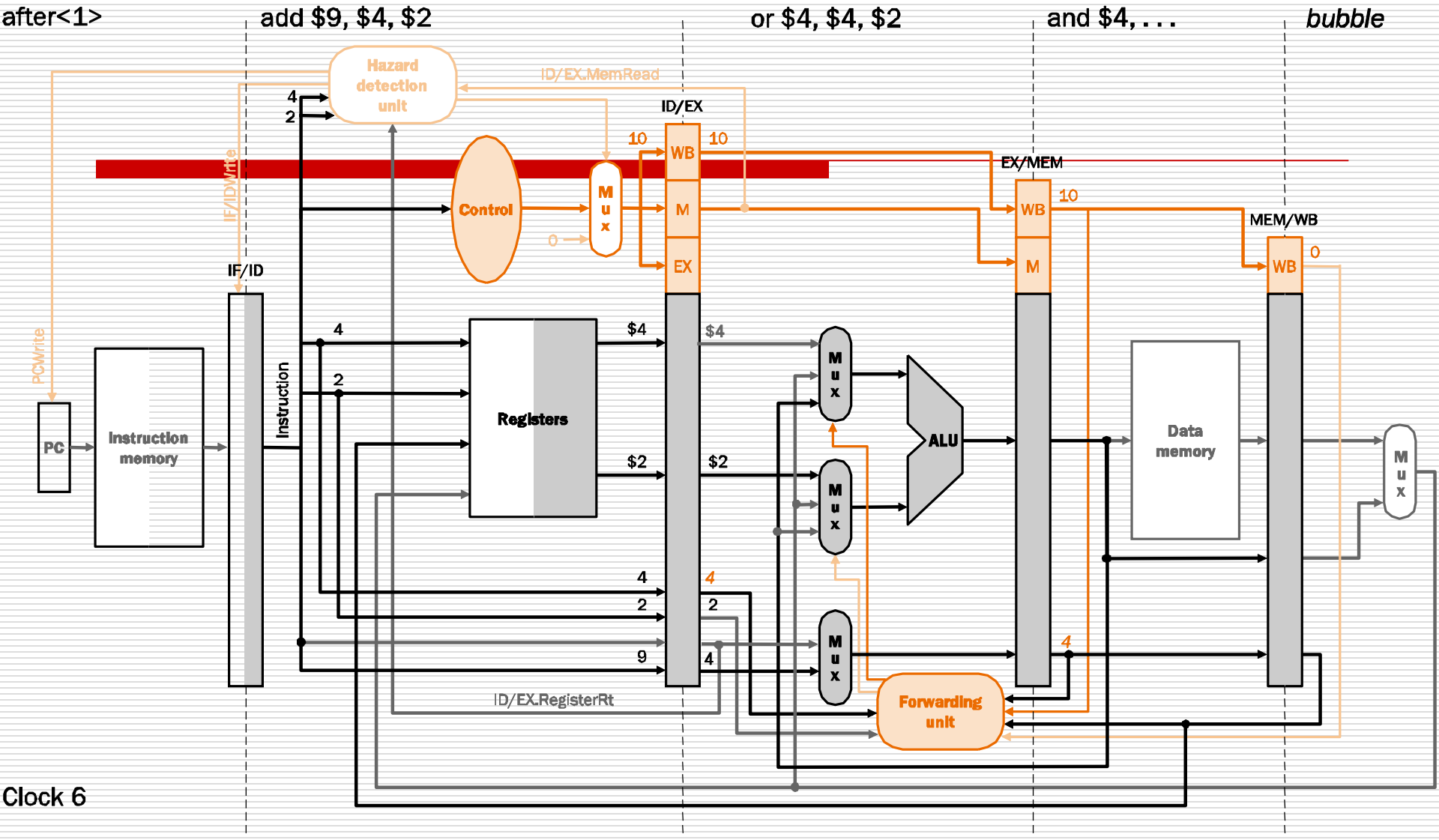
- and $4, $2, $5
- or $4, $4, $2
- add $9, $4, $2

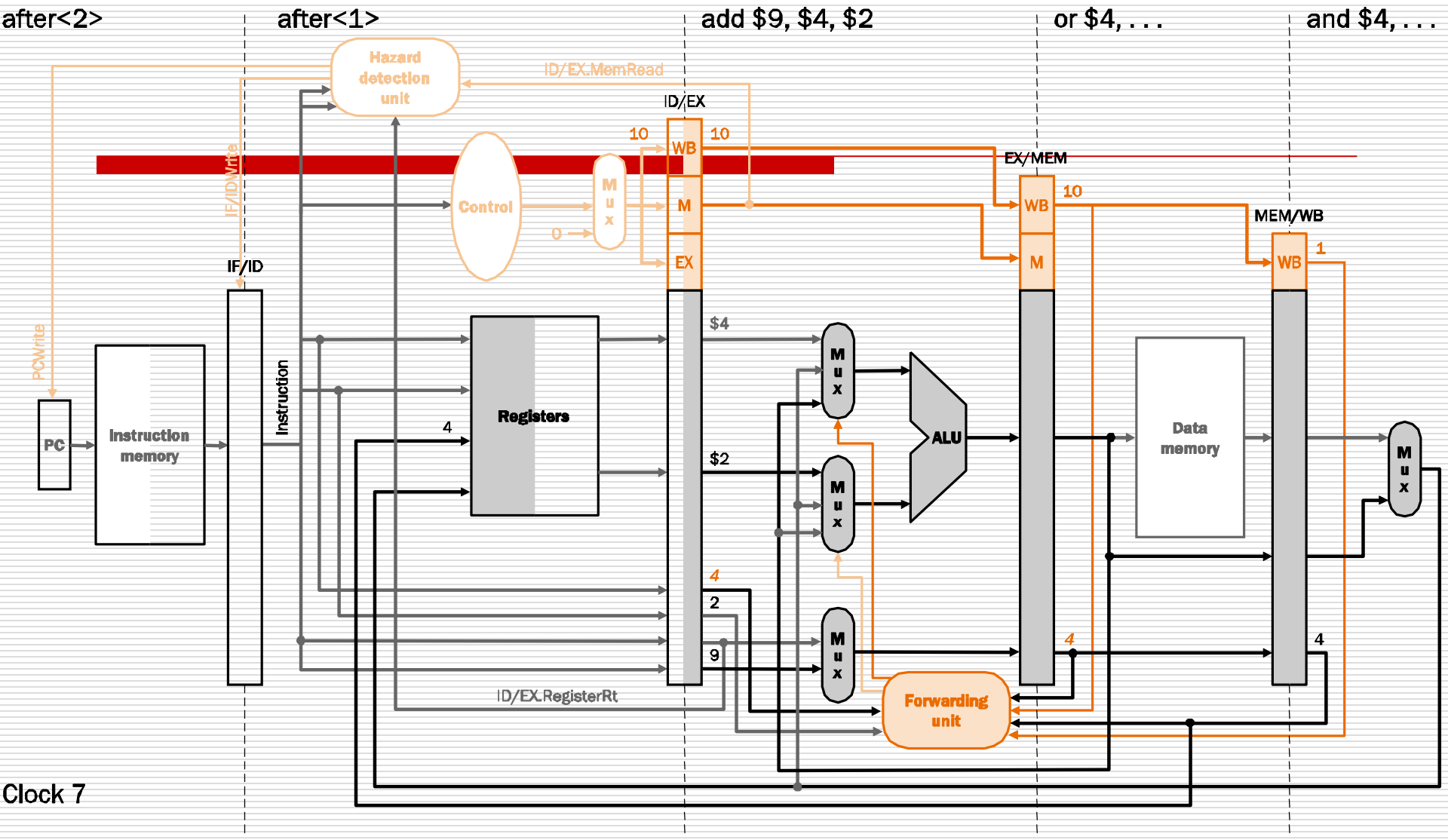

and $4, $2, $5
lw $2, 20($1)
before<1>
before<2>
before<3>
Hazard detection unit
ID/EX.MemRead
ID/EX
11
WB
M
EX
Control
Mux
0
1
X
IF/IDWrite
IF/ID
EX/MEM
WB
M
MEM/WB
WB
PCWrite
PC
Instruction memory
Instruction
1
X
Registers
$1
$X
1
X
2
Mux
ALU
Data memory
Mux
ID/EX.RegisterRt
Forwarding unit
Clock 2


or $4, $4, $2
and $4, $2, $5
lw $2, 20($1)
before<1>
before<2>
Hazard detection unit
ID/EX.MemRead
ID/EX
00
11
WB
M
EX
Control
M u x
0
EX/MEM
MEM/WB
IF/ID
IF/IDWrite
PCWrite
PC
Instruction memory
Instruction
Registers
$2
$1
$5
$X
ALU
Data memory
Forwarding unit
ID/EX.RegisterRt
Clock 3


or $4, $4, $2
and $4, $2, $5
bubble
lw $2, . . .
before<1>
Hazard detection unit
ID/EX.MemRead
ID/EX
EX/MEM
MEM/WB
WB
M
EX
Control
Mux
IF/IDWrite
PCWrite
IF/ID
PC
Instruction memory
Instruction
Registers
ALU
Data memory
Forwarding unit
ID/EX.RegisterRt
Clock 4


add $9, $4, $2
or $4, $4, $2
and $4, $2, $5
bubble
lw $2, . . .
Hazard detection unit
ID/EX.MemRead
ID/EX
EX/MEM
MEM/WB
IF/IDWrite
PCWrite
Control
WB
M
EX
Mux
IF/ID
PC
Instruction memory
Instruction
Registers
ALU
Data memory
Forwarding unit
ID/EX.RegisterRt
Clock 5


after<1>
add $9, $4, $2
or $4, $4, $2
and $4, . . .
bubble
Hazard detection unit
ID/EX.MemRead
ID/EX
EX/MEM
MEM/WB
IF/IDWrite
PCWrite
Control
WB
M
EX
Mux
IF/ID
Instruction
PC
Instruction memory
Registers
ALU
Data memory
Forwarding unit
ID/EX.RegisterRt
Clock 6


after<2>
after<1>
add $9, $4, $2
or $4, . . .
and $4, . . .
Hazard detection unit
ID/EX.MemRead
ID/EX
EX/MEM
MEM/WB
IF/IDWrite
PCWrite
Control
WB
M
EX
10
0
1
IF/ID
Instruction
PC
Instruction memory
Registers
4
$4
$2
2
9
ALU
Data memory
Forwarding unit
ID/EX.RegisterRt
Clock 7

Stalls and Performance

- Stalls reduce performance
  - But are required to get correct results
- Compiler can arrange code to avoid hazards and stalls
  - Requires knowledge of the pipeline structure

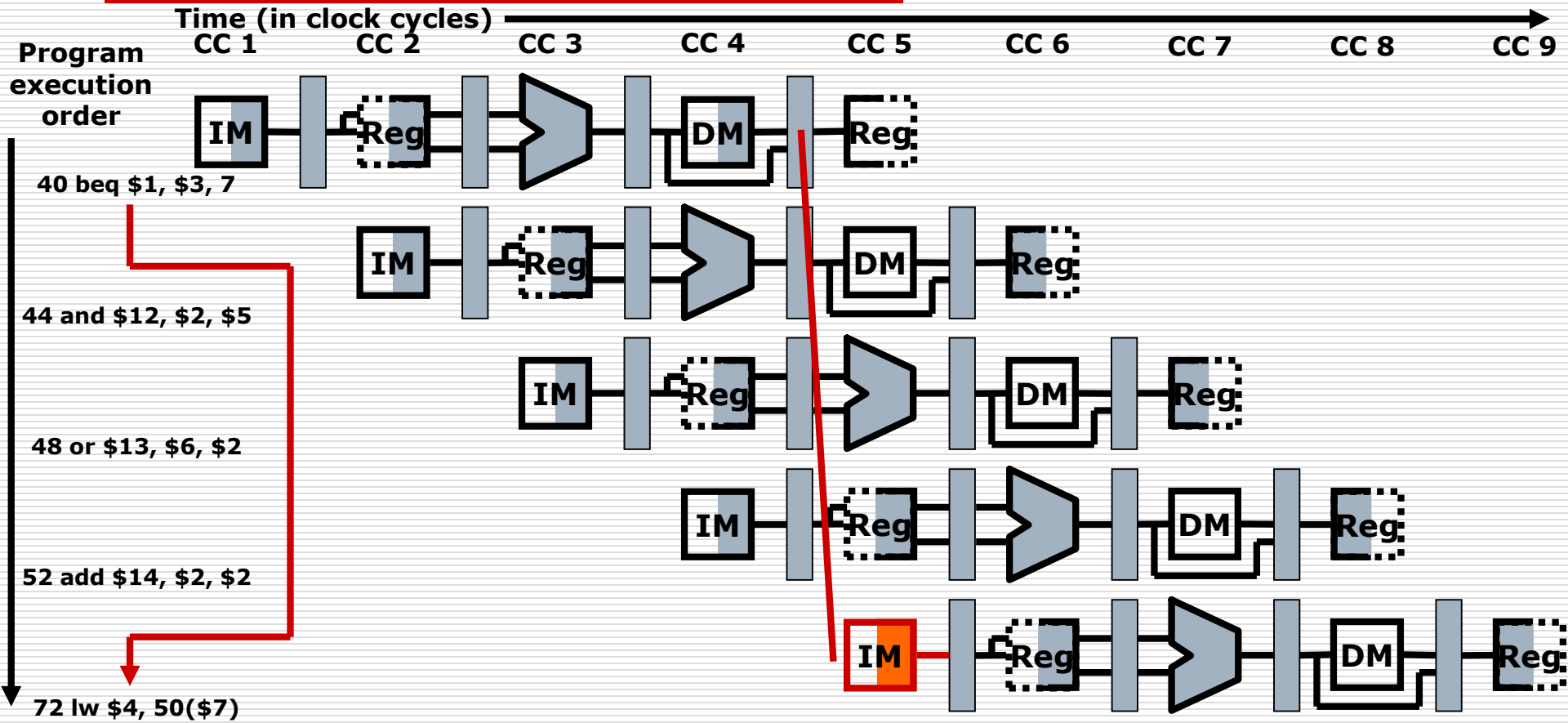
Branch Hazards

- When we decide to branch, other instructions are in the pipeline!
- We are predicting “branch not taken”
  - need to add hardware for flushing instructions if we are wrong

Example of Branch Hazards

```
36   sub  $10, $4, $8
40   beq  $1, $3, 7       # PC-relative
44   and  $12, $2, $5     #  branch to
48   or   $13, $2, $6     #  40+4+7*4
52   add  $14, $4, $2
56   slt  $15, $6, $7
...
72   lw   $4, 50($7)
```

Branch Hazards

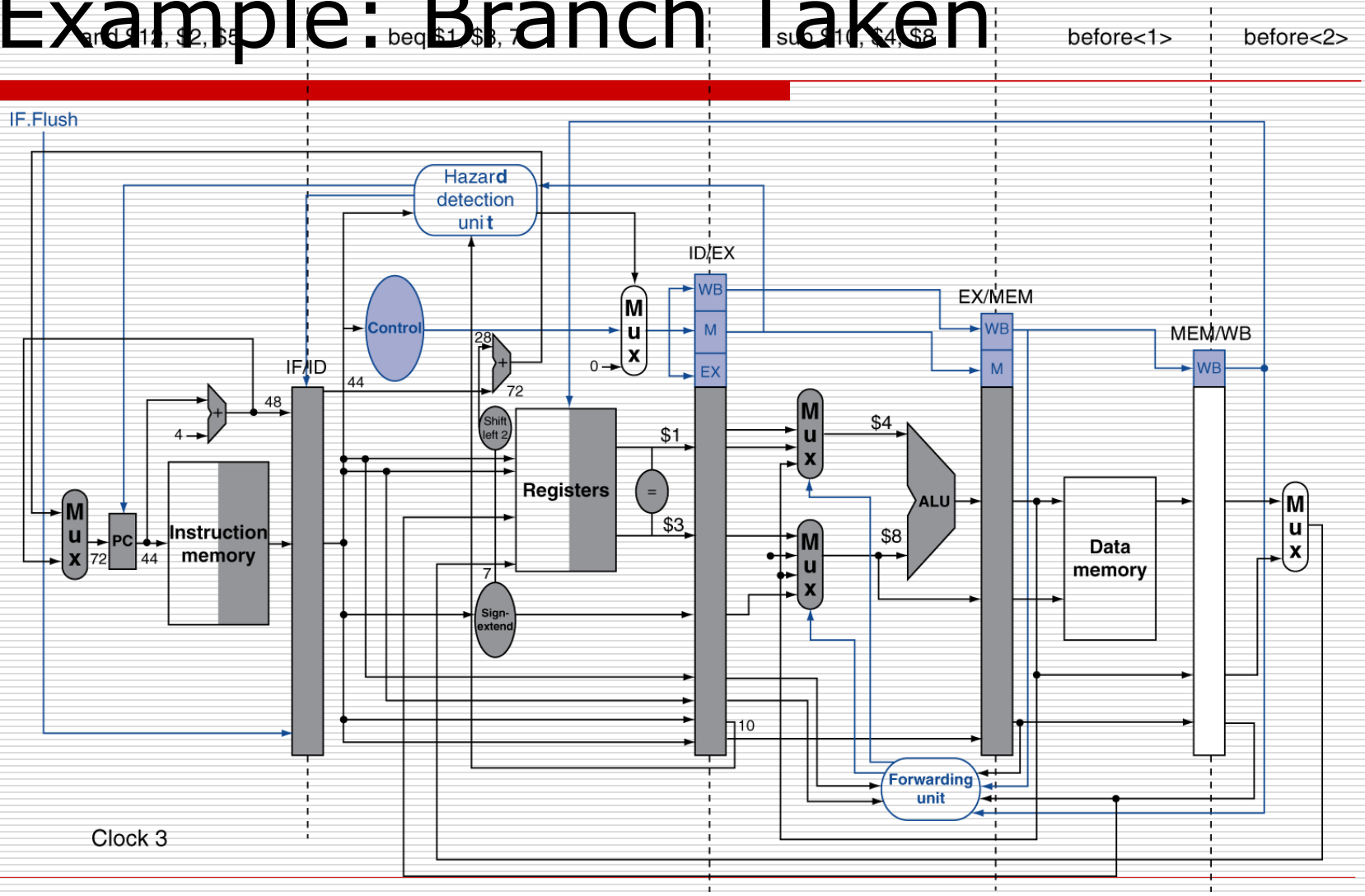


Reducing Branch Delay

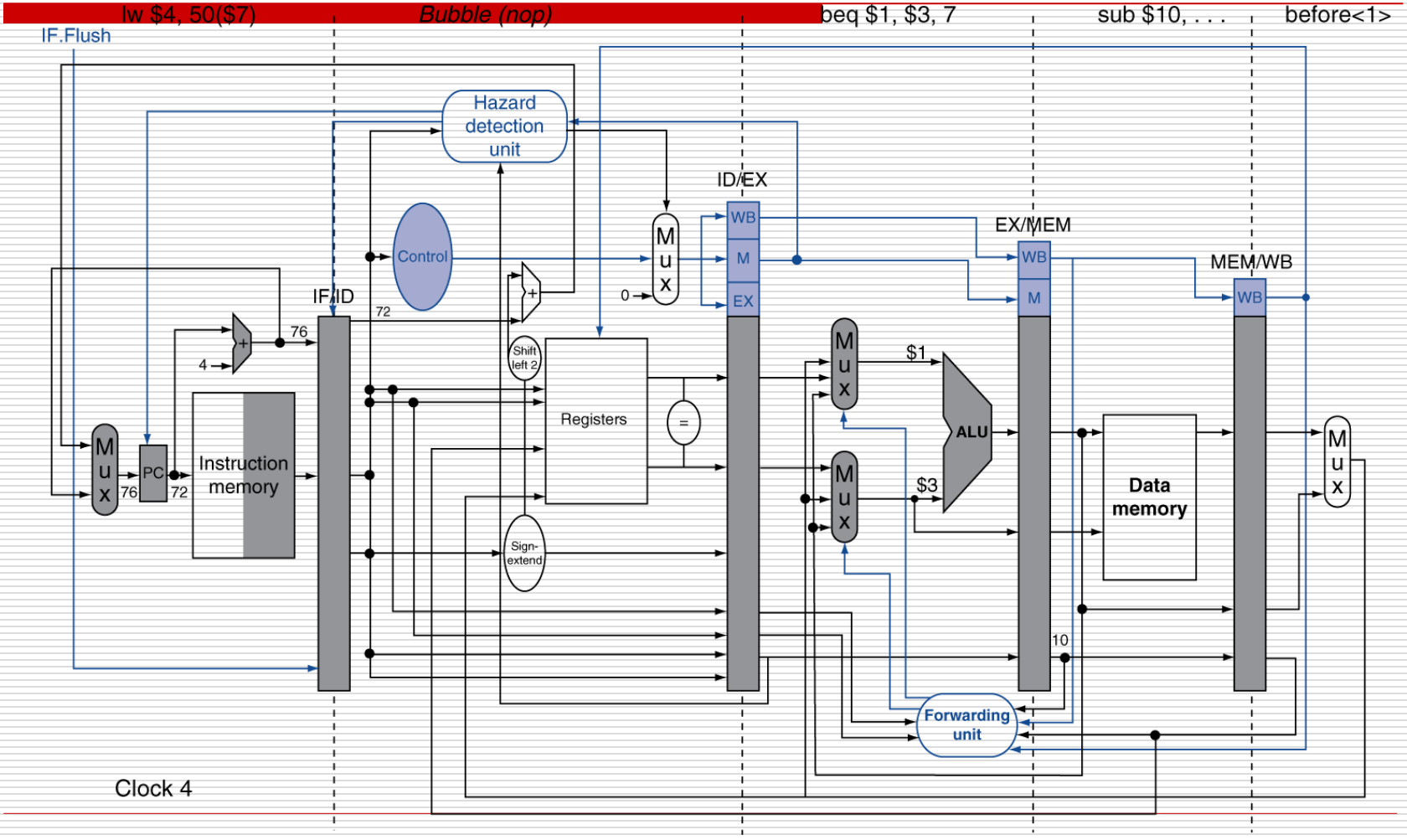
- Move hardware to determine outcome to ID stage
  - Target address adder
  - Register comparator
- Example: branch taken

```
36:   sub   $10, $4, $8
40:   beq   $1,  $3, 7
44:   and   $12, $2, $5
48:   or    $13, $2, $6
52:   add   $14, $4, $2
56:   slt   $15, $6, $7
      ...
72:   lw    $4, 50($7)
```

Example: Branch Taken



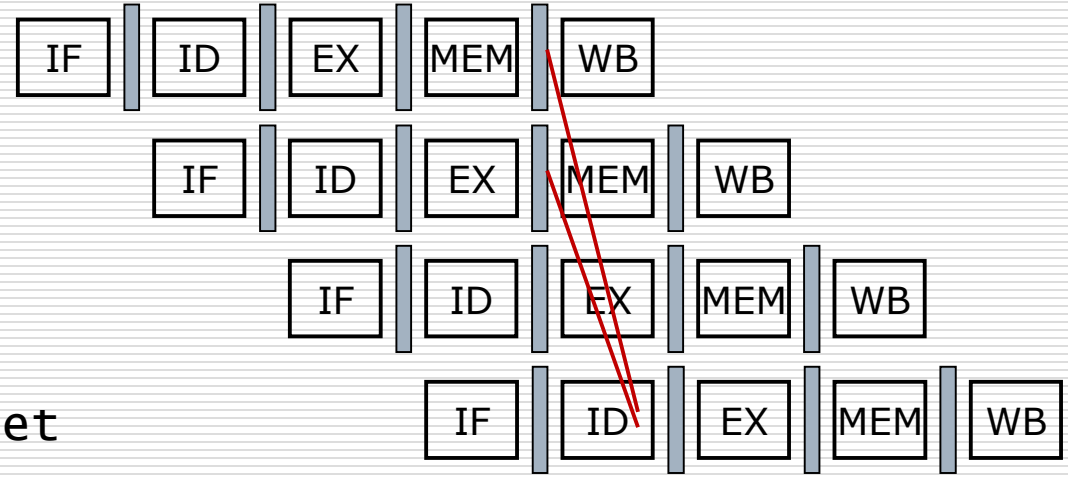
Example: Branch Taken



Data Hazards for Branches

- If a comparison register is a destination of 2nd or 3rd preceding ALU instruction

```
add $1, $2, $3
add $4, $5, $6
…
beq $1, $4, target
```

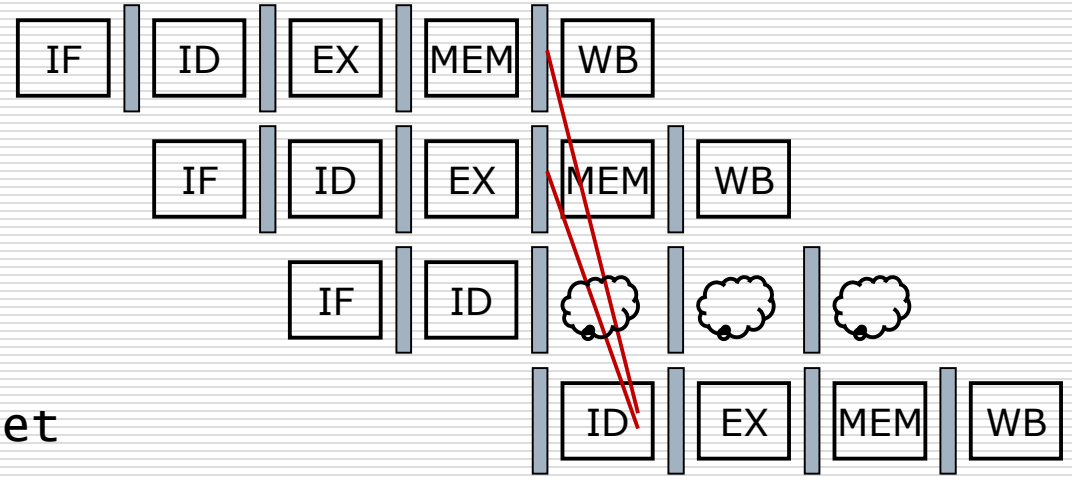


- Can resolve using forwarding

Data Hazards for Branches

- If a comparison register is a destination of preceding ALU instruction or 2nd preceding load instruction
  - Need 1 stall cycle

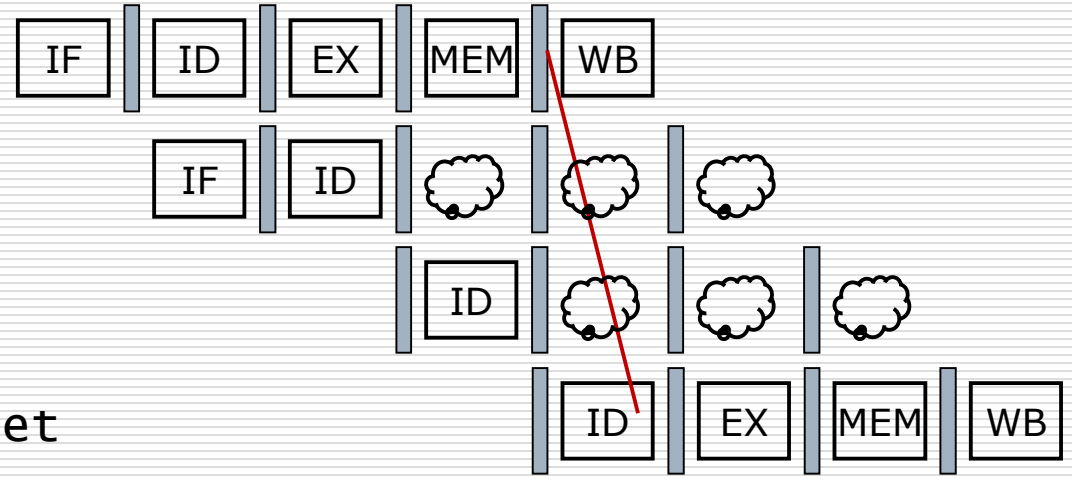
```
lw  $1, addr
add $4, $5, $6
beq stalled
beq $1, $4, target
```



Data Hazards for Branches

- If a comparison register is a destination of immediately preceding load instruction
  - Need 2 stall cycles

```
lw  $1, addr
beq stalled
beq stalled
beq $1, $0, target
```

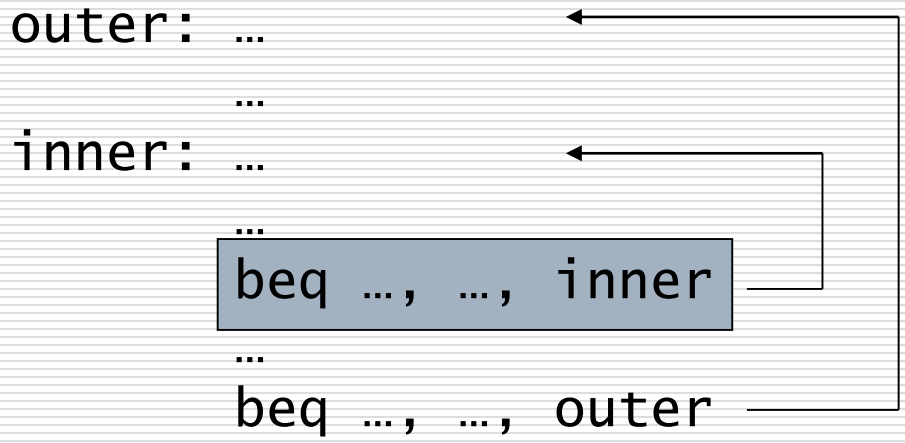


Dynamic Branch Prediction

- In deeper and superscalar pipelines, branch penalty is more significant
- Use dynamic prediction
  - Branch prediction buffer (aka branch history table)
  - Indexed by recent branch instruction addresses
  - Stores outcome (taken/not taken)
  - To execute a branch
    - Check table, expect the same outcome
    - Start fetching from fall-through or target
    - If wrong, flush pipeline and flip prediction

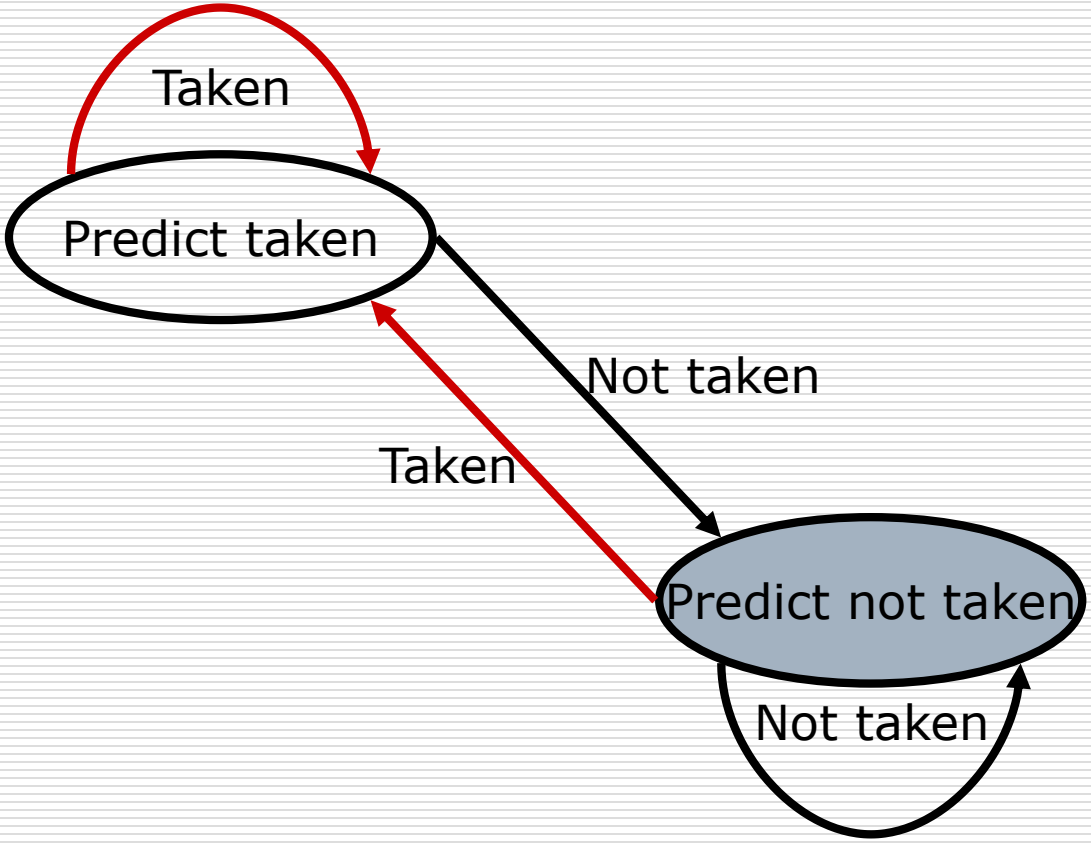
1-Bit Predictor: Shortcoming

- Inner loop branches mispredicted twice!



  - Mispredict as taken on last iteration of inner loop
  - Then mispredict as not taken on first iteration of inner loop next time around

1-Bit Predictor: Shortcoming



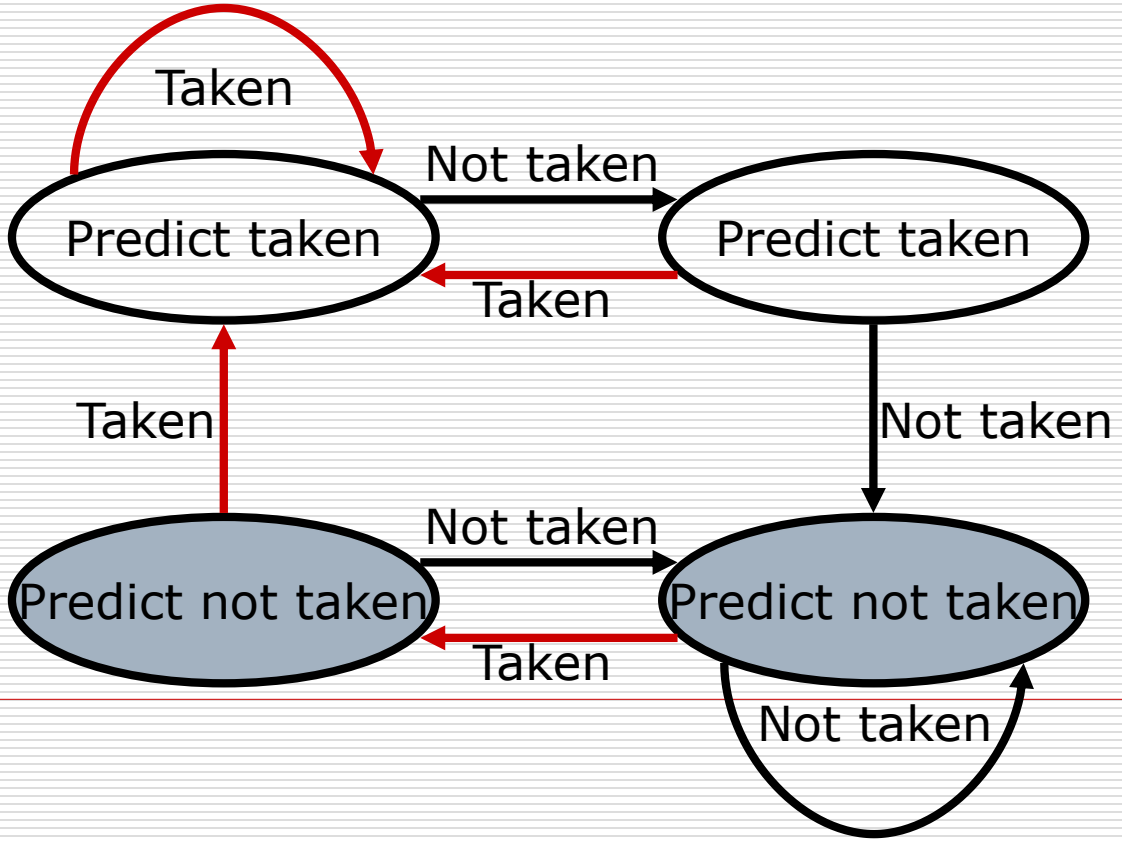
Loops and Prediction

- Consider a loop branch that branches 9 times in a row, then is not taken once. What is the prediction accuracy for this branch, assuming the prediction bit for this branch remains in the 1-bit prediction buffer?

- Answer: 80%, WHY?

2-Bit Predictor

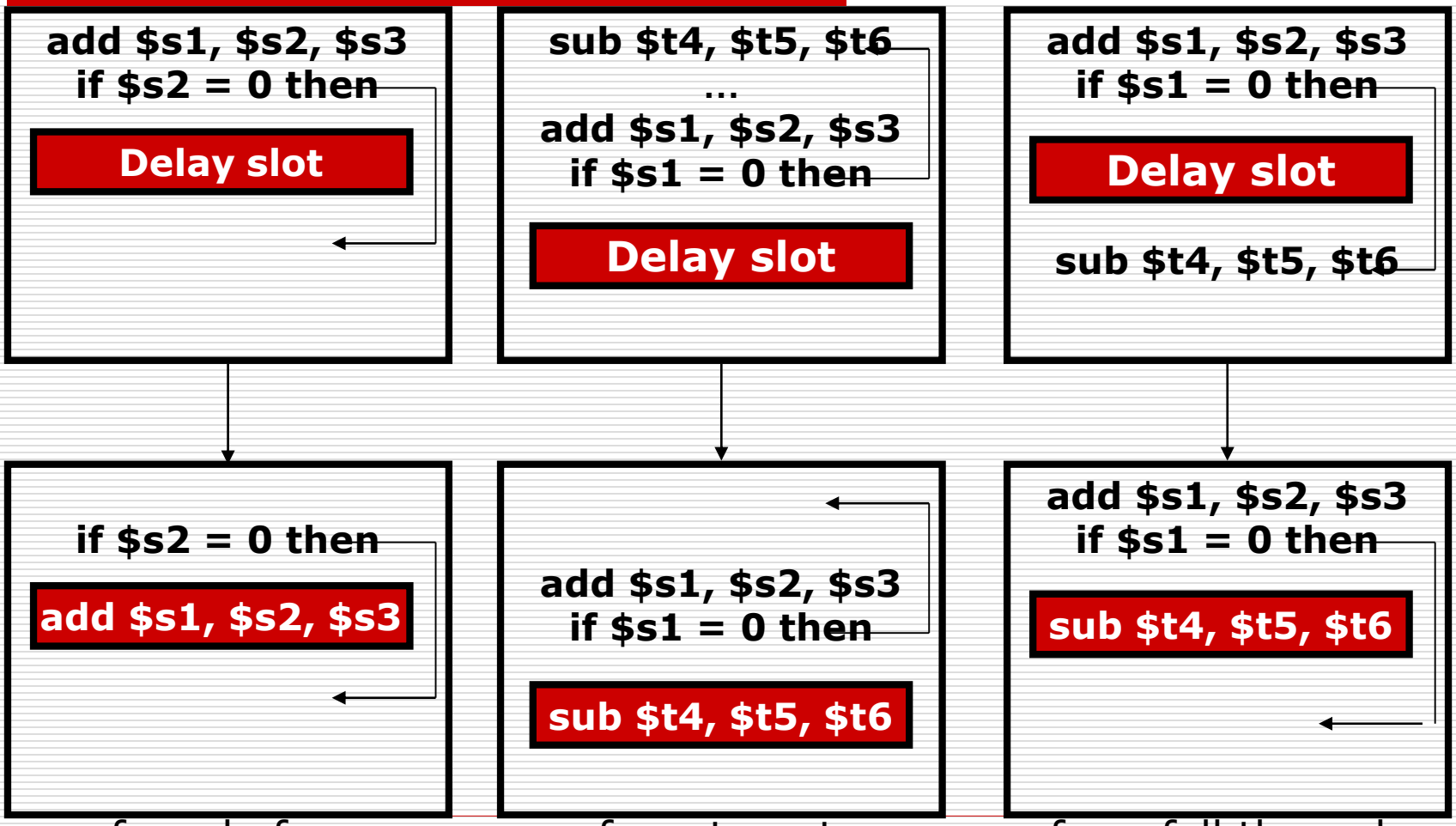
- Only change prediction on two successive mispredictions



Calculating the Branch Target

- Even with predictor, still need to calculate the target address
  - 1-cycle penalty for a taken branch
- Branch target buffer
  - Cache of target addresses
  - Indexed by PC when instruction fetched
    - If hit and instruction is branch predicted taken, can fetch target immediately

Scheduling the Branch Delay Slot



Comparing Performance of Several Control Schemes

- assume the operation times:
  - memory units: 200ps
  - ALU and adders: 100ps
  - register file: 50ps

- clock cycle time of single-cycle datapath
  - 200+50+100+200+50=600ps

Comparing Performance of Several Control Schemes

- the clock cycles of pipelined design:
  - loads: 1 or 2
    - 1 for no load-use dependence
    - 2 for load-use dependence
    - average = 1.5
  - stores: 1
  - ALU instructions: 1
  - branches: 1 or 2
    - 1 for predicted correctly
    - 2 for not predicted correctly
    - average = 1.25
  - jumps: 2

- assume the instruction mix:
  - loads: 25%
  - stores: 10%
  - ALU: 52%
  - branches: 11%
  - jumps: 2%

- average CPI of pipelined design
  - 0.25x1.5+0.1x1+0.52x1+0.11x1.25+0.02x2=1.17

Comparing Performance of Several Control Schemes

- average instruction time of
- single-cycle datapath
  - 600ps
- pipelined design
  - 200x1.17=234ps

Exceptions and Interrupts

- "Unexpected" events requiring change in flow of control
  - Different ISAs use the terms differently
- Exception
  - Arises within the CPU
    - e.g., undefined opcode, overflow, syscall, …
- Interrupt
  - From an external I/O controller
- Dealing with them without sacrificing performance is hard

Handling Exceptions

- In MIPS, exceptions managed by a System Control Coprocessor (CP0)
- Save PC of offending (or interrupted) instruction
  - In MIPS: Exception Program Counter (EPC)
- Save indication of the problem
  - In MIPS: Cause register
  - We'll assume 1-bit
    - 0 for undefined opcode, 1 for overflow
- Jump to handler at 8000 00180

An Alternate Mechanism

- Vectored Interrupts
  - Handler address determined by the cause
- Example:
  - Undefined opcode: C000 0000
  - Overflow: C000 0020
  - …: C000 0040
- Instructions either
  - Deal with the interrupt, or
  - Jump to real handler

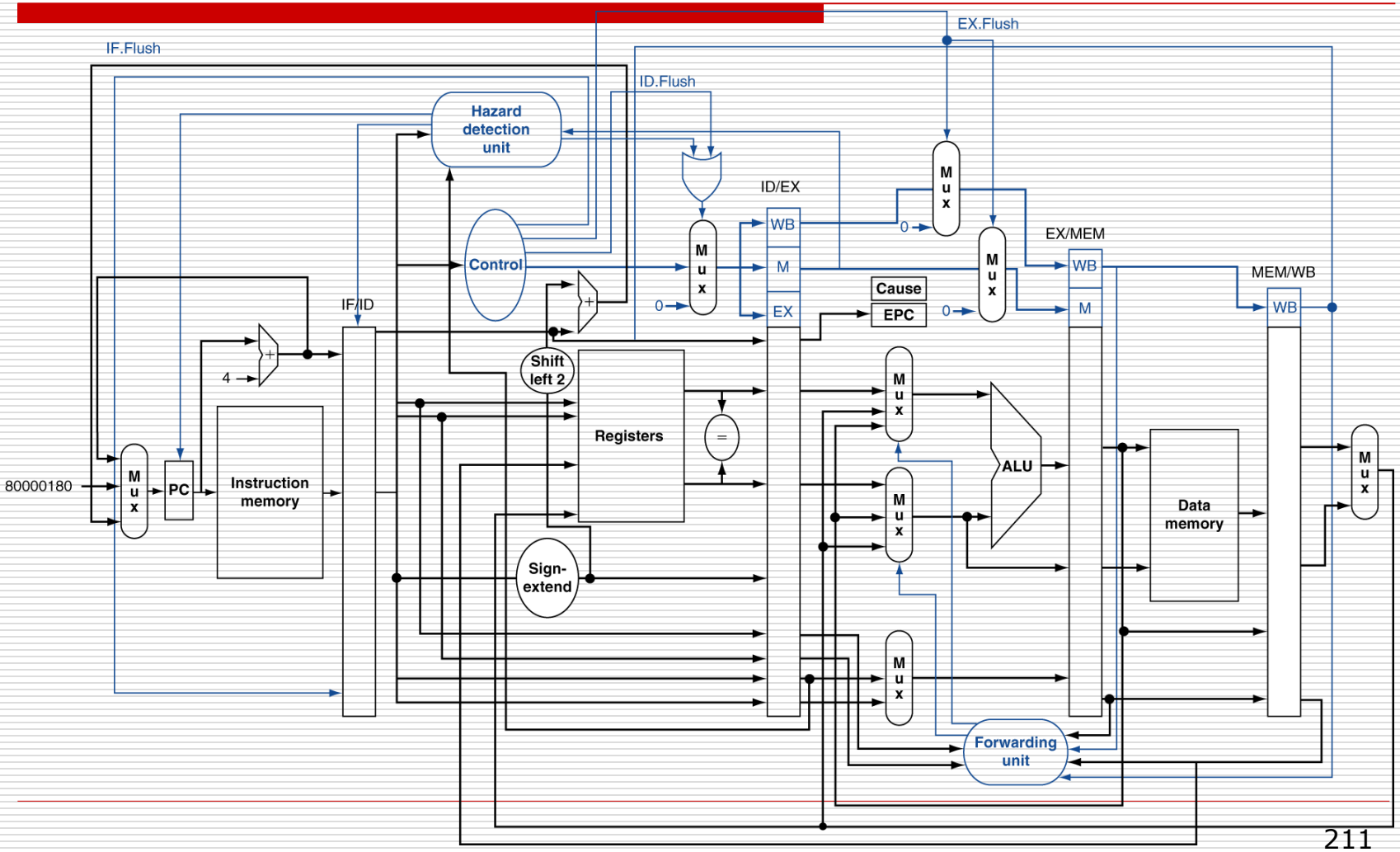
Handler Actions

- Read cause, and transfer to relevant handler
- Determine action required
- If restartable
  - Take corrective action
  - use EPC to return to program
- Otherwise
  - Terminate program
  - Report error using EPC, cause, …

Exceptions in a Pipeline

- Another form of control hazard
- Consider overflow on add in EX stage
  - add $1, $2, $1
  - Prevent $1 from being clobbered
  - Complete previous instructions
  - Flush add and subsequent instructions
  - Set Cause and EPC register values
  - Transfer control to handler
- Similar to mispredicted branch
  - Use much of the same hardware

Pipeline with Exceptions



Exception Properties

- Restartable exceptions
  - Pipeline can flush the instruction
  - Handler executes, then returns to the instruction
    - Refetched and executed from scratch
- PC saved in EPC register
  - Identifies causing instruction
  - Actually PC + 4 is saved
    - Handler must adjust

Exception Example

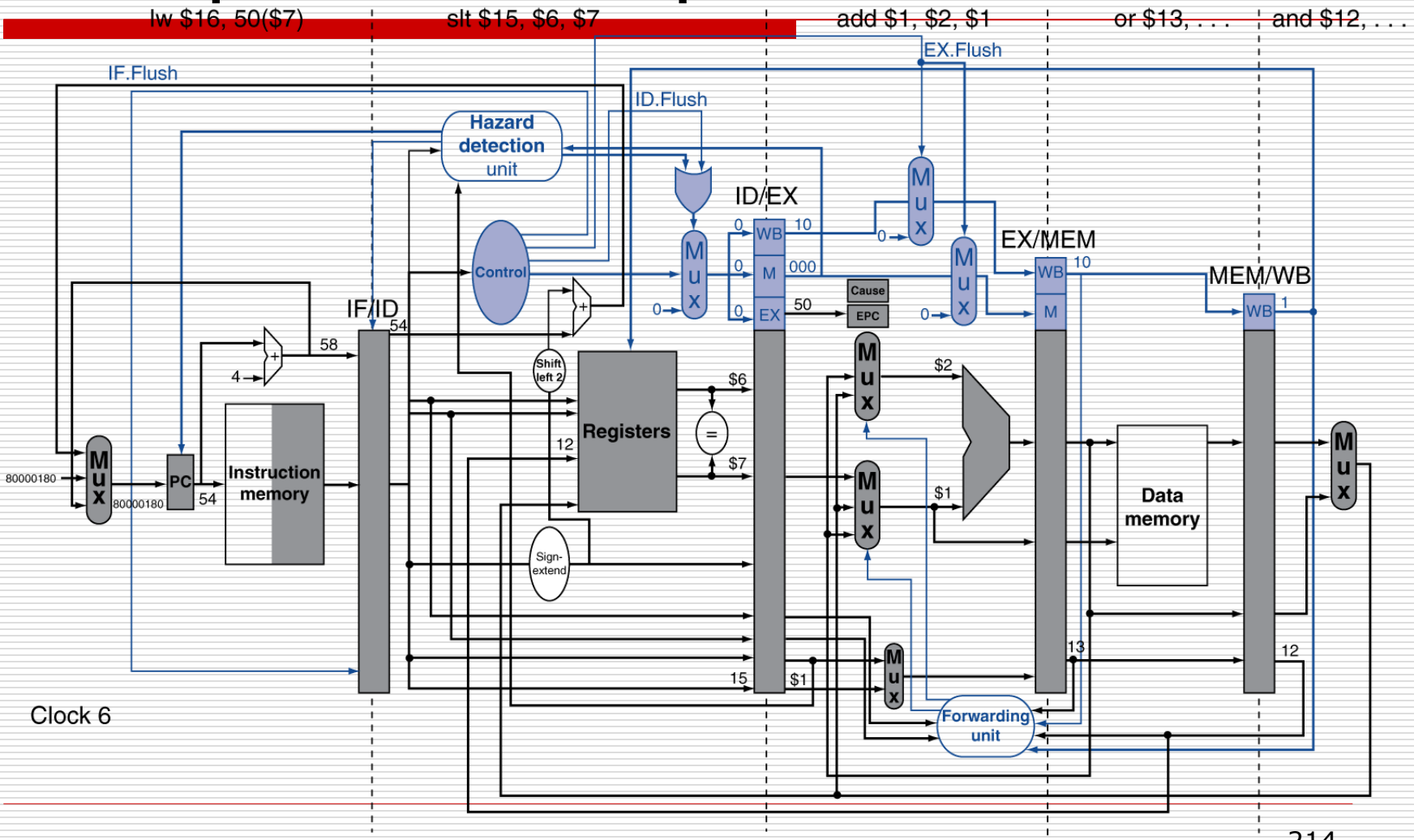
- Exception on `add` in

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

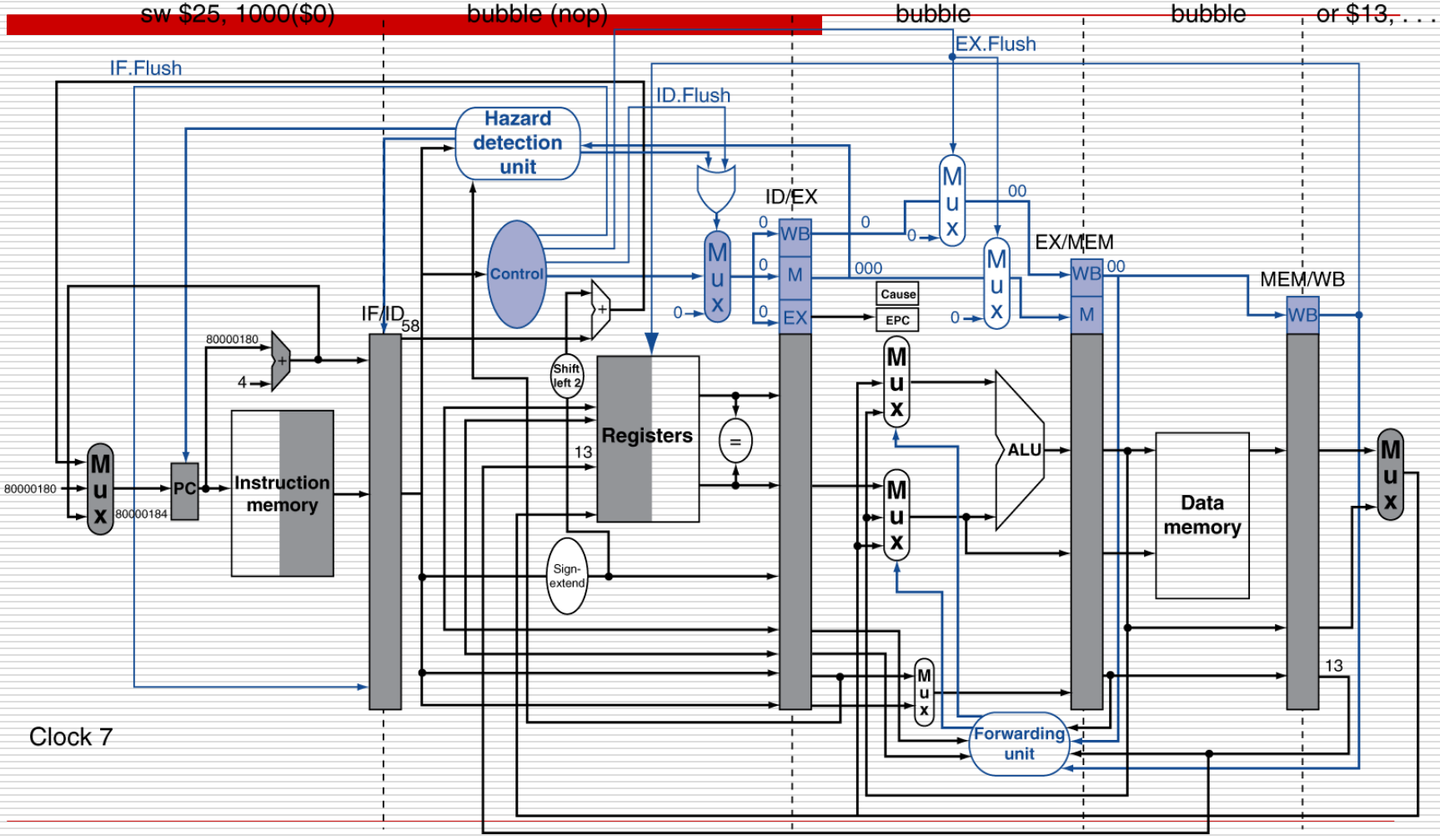
- Handler

```
80000180   sw   $25, 1000($0)
80000184   sw   $26, 1004($0)
…
```

Exception Example



Exception Example



Multiple Exceptions

- Pipelining overlaps multiple instructions
  - Could have multiple exceptions at once
- Simple approach: deal with exception from earliest instruction
  - Flush subsequent instructions
  - "Precise" exceptions
- In complex pipelines
  - Multiple instructions issued per cycle
  - Out-of-order completion
  - Maintaining precise exceptions is difficult!

Imprecise Exceptions

- Just stop pipeline and save state
  - Including exception cause(s)
- Let the handler work out
  - Which instruction(s) had exceptions
  - Which to complete or flush
    - May require “manual” completion
- Simplifies hardware, but more complex handler software
- Not feasible for complex multiple-issue out-of-order pipelines