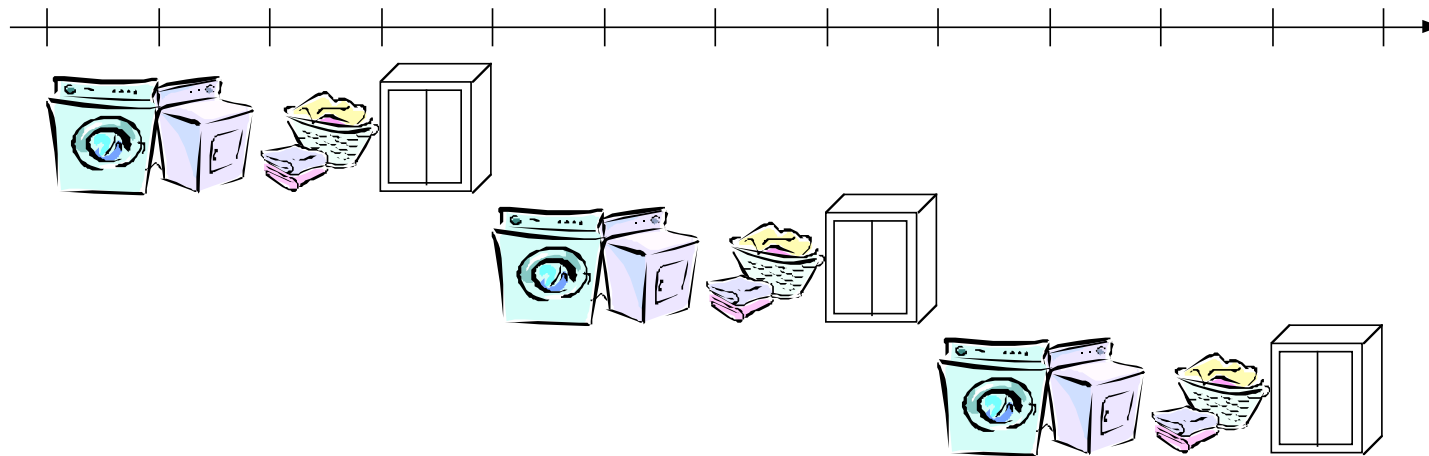


Pipelining

Read: Chapter 4, Sections 4.5 to 4.8 (4th edition)

- Laundry example: washing (30 minutes), drying (30 minutes), folding (30 minutes), “stashing” (30 minutes).
- If only one person’s wash, it takes 2 hours to complete.
- If several folks need to do laundry, can do in 2 hours each — sequential solution:



- But, the washer, dryer, “folder”, and “stasher” are *independent* units.

Pipeline basics:

- Pipelined laundry takes 3.5 hours for four loads:

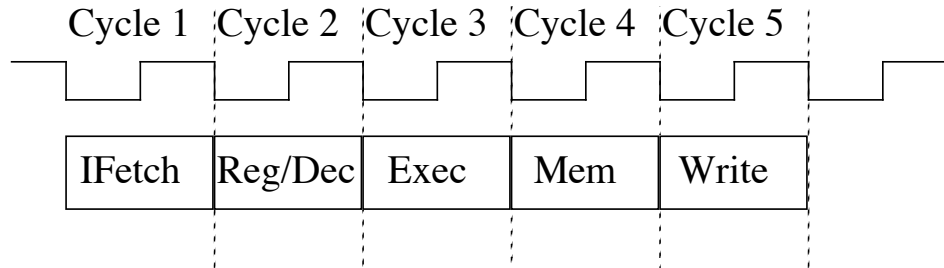


- Pipelining:
 - Does not help the *latency* of a single task — still takes 2 hours to do one person's laundry.
 - Does help the *throughput* of the entire work load — 3.5 hours vs. 8 hours.
 - *Multiple* tasks operating simultaneously, each using different resources.
 - Potential *speedup* = number of pipe stages.
 - Rate limited by slowest pipeline stage.
 - Unbalanced lengths of pipe stages reduces speedup.
 - Time to “fill” pipeline and time to “drain” it reduces speedup.

Pipeline basics (continued):

- Consider the load word instruction:

lw **\$s0, 0(\$t0)**



- **IFetch**: Instruction Fetch: get the instruction from memory.
- **Reg/Dec**: Fetch values from Registers and Decode the instruction.
- **Exec**: Execute; calculate the memory address from which to load the word.
- **Mem**: Read the word from Memory.
- **Write**: Write the word to the Register.

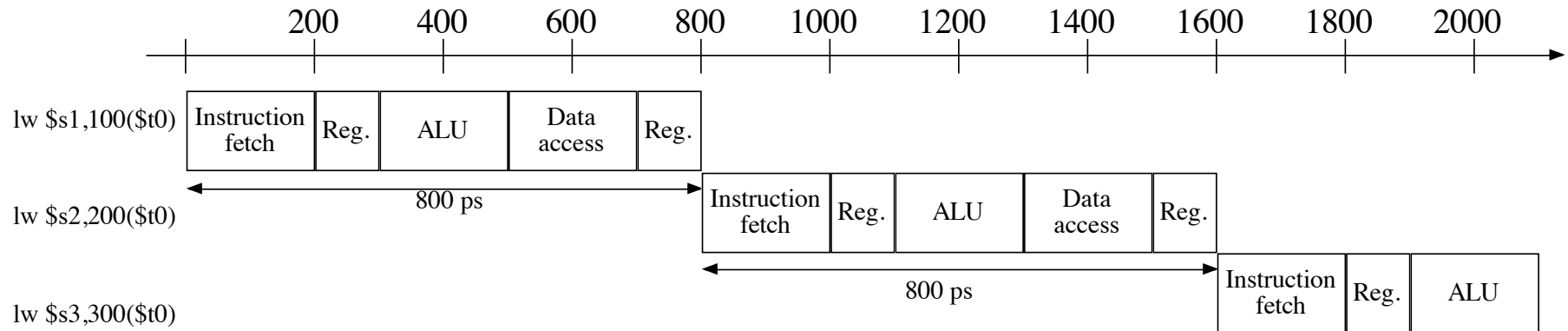
Pipeline basics (continued):

- A more realistic picture: Not all cycles take the same amount of time:
 - Memory access is slower.
 - ALU computation is slower.
 - Register access is faster.
- Figure 4.26, page 333 (4th edition):

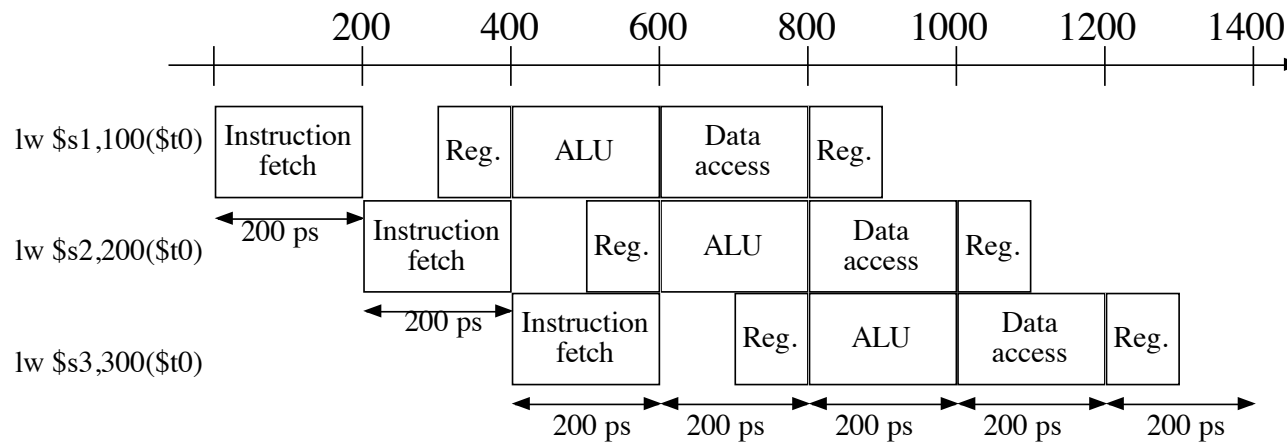
Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (lw)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (sw)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (add , sub , and , or , slt)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (beq)	200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

Pipeline basics (continued):

- Can improve performance by increasing the instruction throughput:
- 3 load word ops, 2.4 nanoseconds:

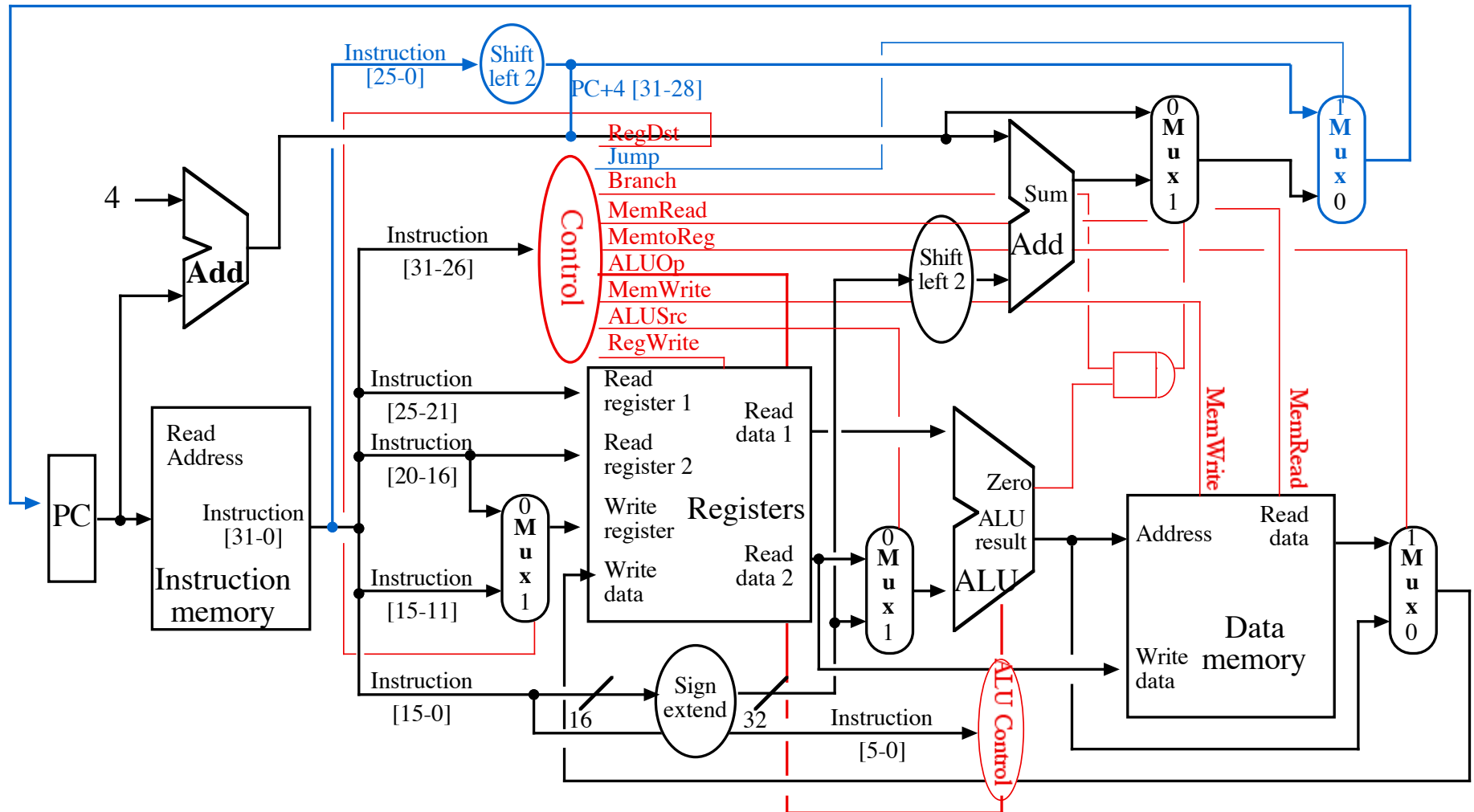


- Becomes 3 load word ops, 1.4 nanoseconds:



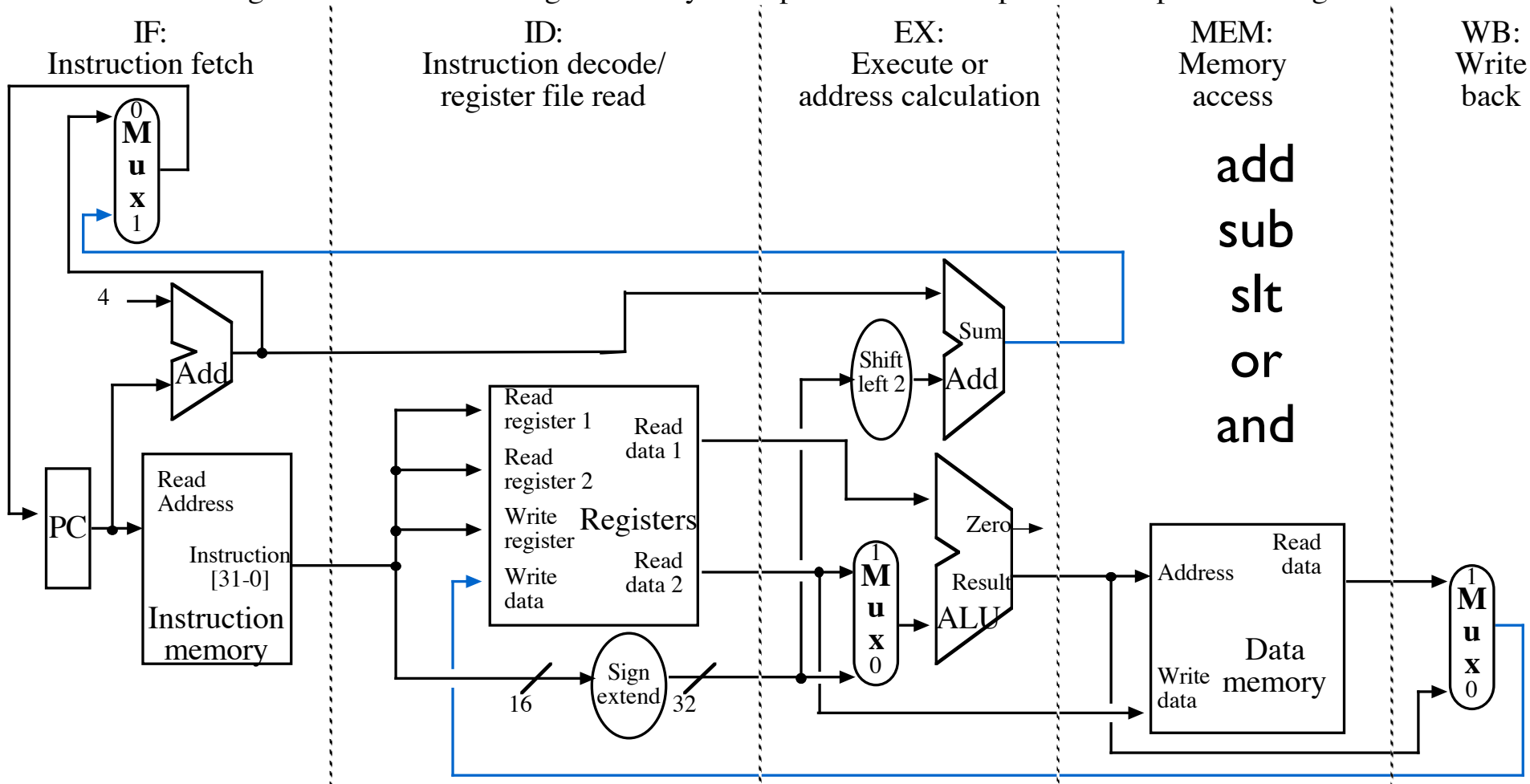
- Clock cycle time dependent on the slowest phases: 200 picoseconds in this case.

Single-cycle Implementation (continued):



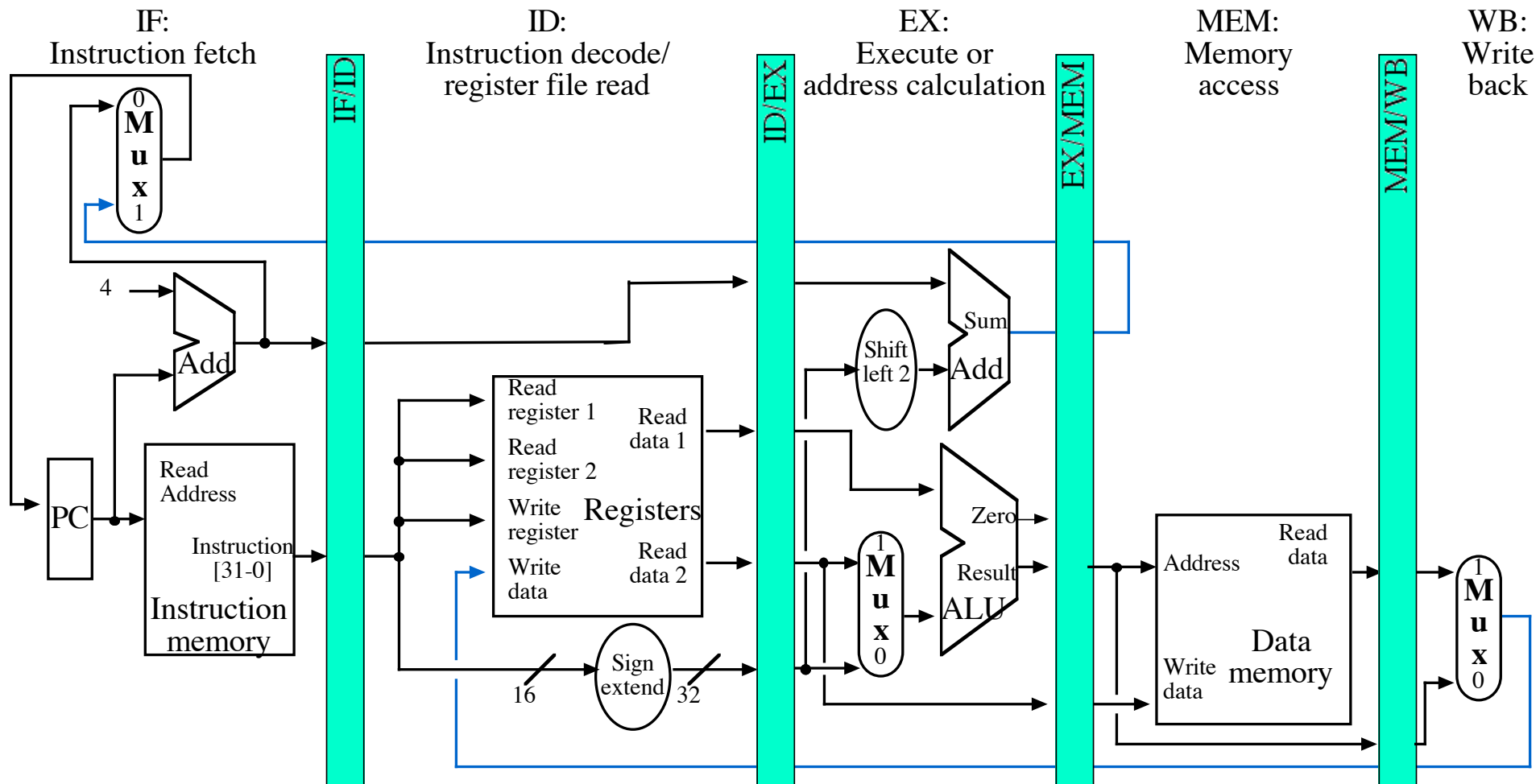
Building a Pipelined datapath:

- Need to re-arrange some items from single-clock cycle implementation to split the data path into stages:



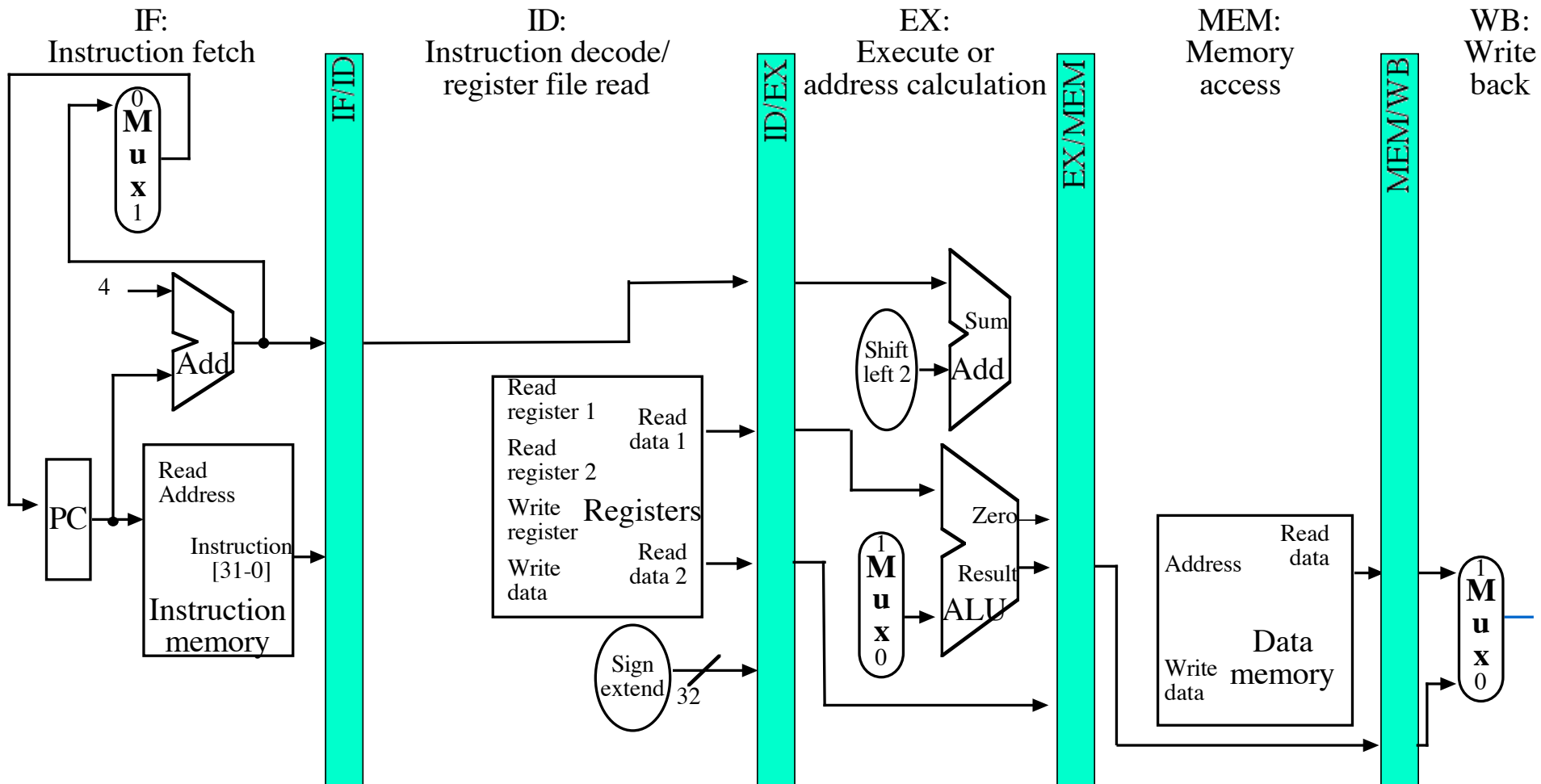
Building a Pipelined datapath (continued):

- Add pipeline registers in-between each pipeline stage.



Building a Pipelined datapath (continued):

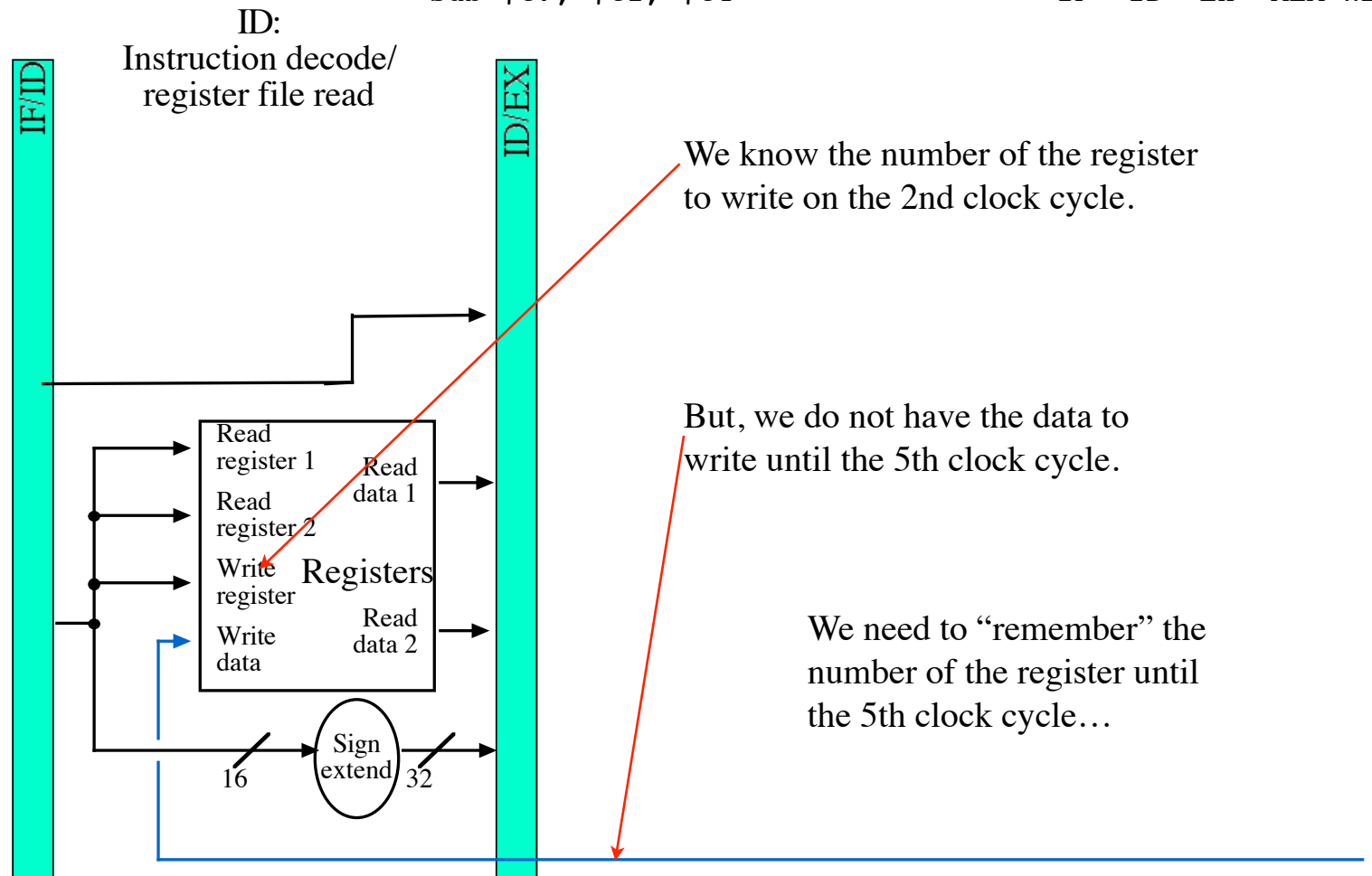
- How big is each pipeline register? How many bits are in each? (We'll need more bits before we are done...)



Building a Pipelined datapath (continued):

- Problem:

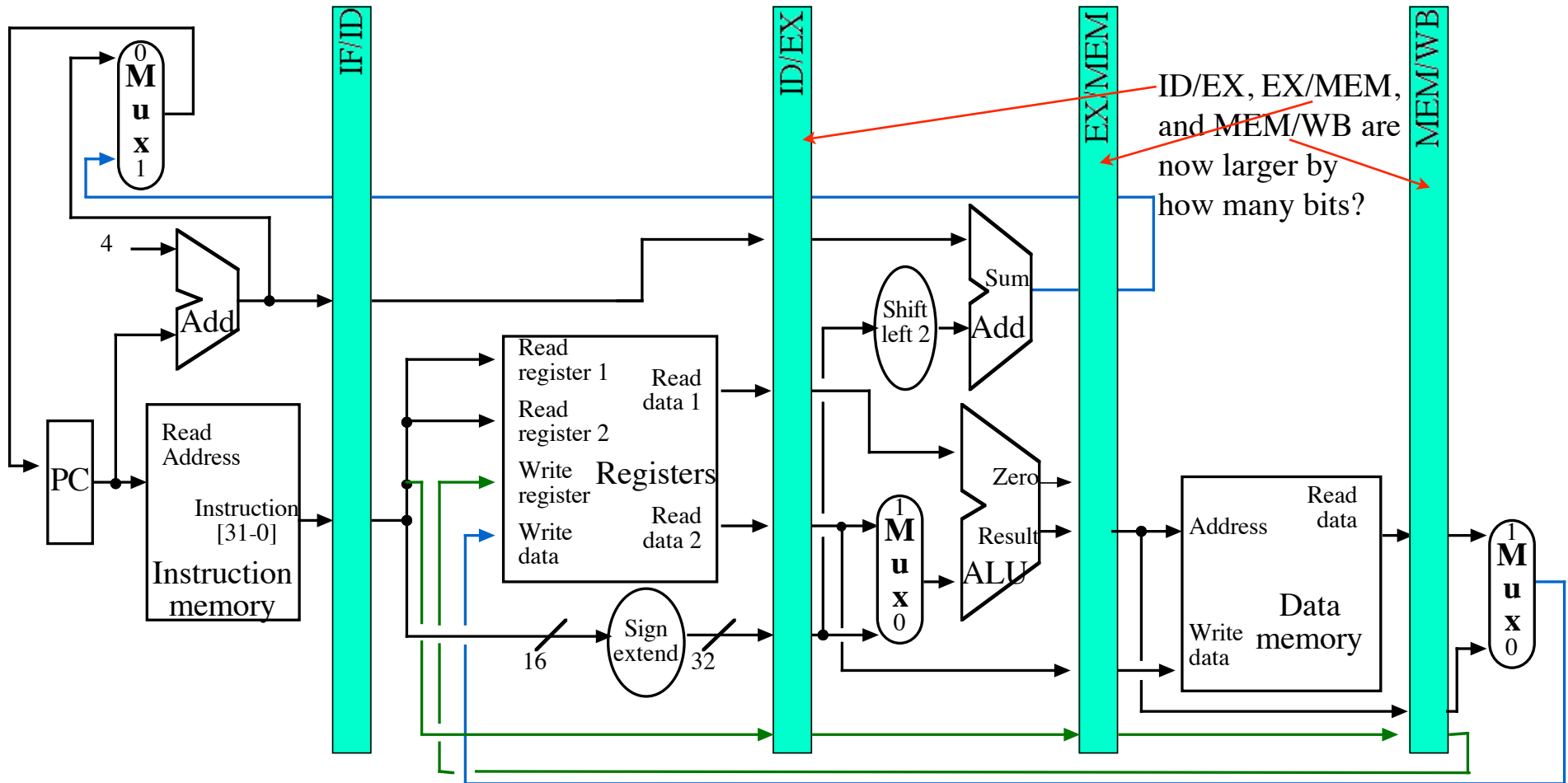
and \$t1, \$t2, \$t3	IF	ID	EX	MEM	WB				
or \$t4, \$t7, \$t2		IF	ID	EX	MEM	WB			
slt \$t5, \$t5, \$t7			IF	ID	EX	MEM	WB		
add \$t6, \$t3, \$t2				IF	ID	EX	MEM	WB	
sub \$t7, \$t1, \$t4					IF	ID	EX	MEM	WB



and \$t1, \$t2, \$t3	IF	ID	EX	MEM	WB				
or \$t4, \$t7, \$t2		IF	ID	EX	MEM	WB			
slt \$t5, \$t5, \$t7			IF	ID	EX	MEM	WB		
add \$t6, \$t3, \$t2				IF	ID	EX	MEM	WB	
sub \$t7, \$t1, \$t4					IF	ID	EX	MEM	WB

Building a Pipelined datapath (continued):

- The write register value is stored in the ID/EX register on cycle 2, then in EX/MEM on cycle 3, then in MEM/WB on cycle 4. The value is finally used on cycle 5.

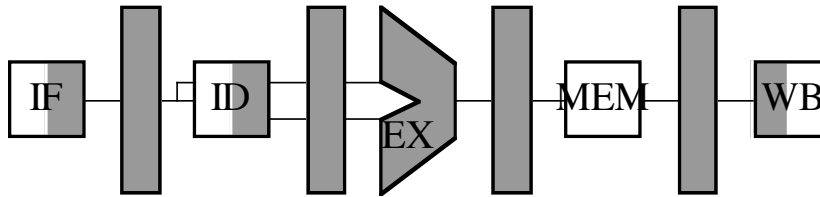






Building a Pipelined datapath (continued):

- What makes pipelining easy?
 - All instructions are the same length.
 - Only a few instruction formats (MIPS uses three: R-type, I-type, J-type)
 - Memory operands appear only in loads and stores.
- What makes it hard?
 - Structural hazards: suppose we have only one memory.
 - Data hazards: an instruction depends on a previous instruction.
 - Control hazards: need to worry about branch instructions.
- We'll build a simple pipeline and look at (some of) these issues.
- (Time permitting) We'll talk about modern processors and what really makes it hard:
 - Exception handling.
 - Trying to improve performance with out-of-order execution, etc.

Representing Pipelines:

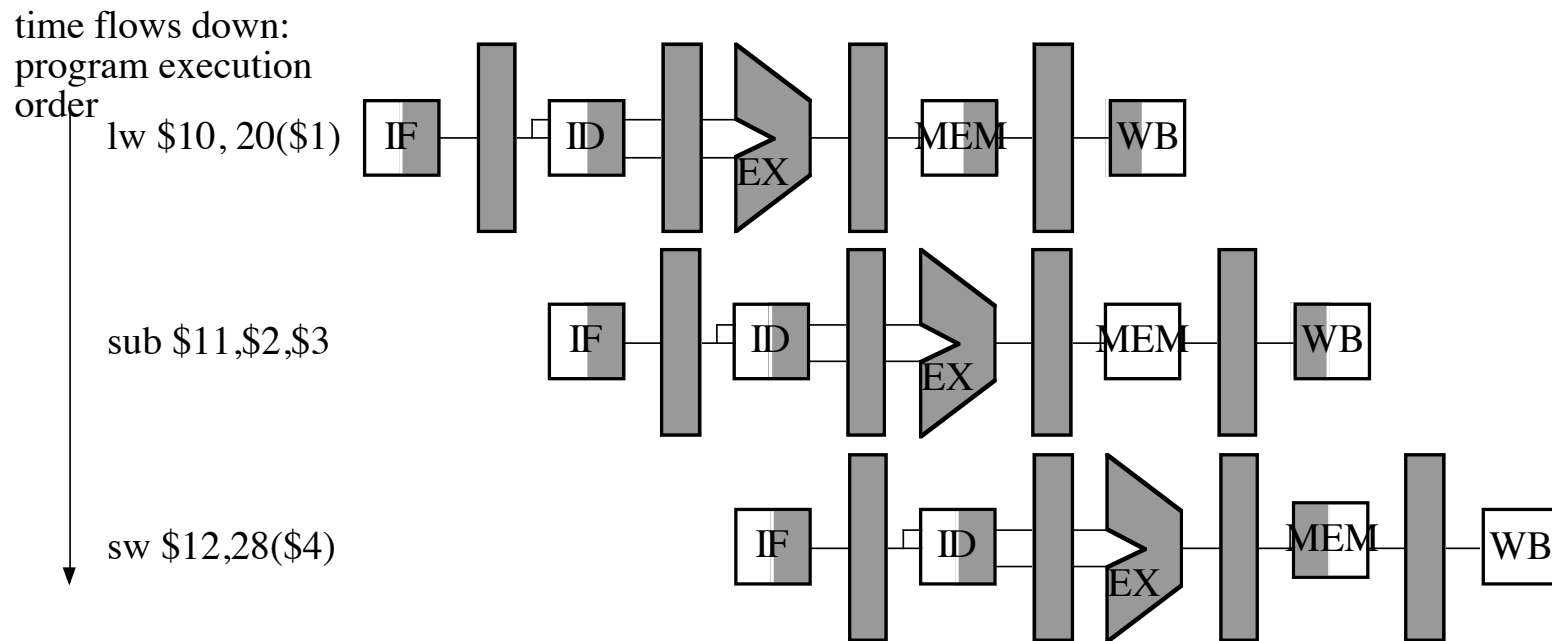
- Simplified drawing to enable us to talk about multiple instructions executing in sequence.
- Example: the add command:



- Full shading indicates combinational unit that is active on that stage: 
- The shading on the right side indicates the unit is being READ on that clock cycle: 
 - Reads occur at the end of the clock cycle.
- The shading on the left side indicates the unit is being WRITTEN on that clock cycle: 
 - Writes occur at the beginning of the clock cycle.
- No shading indicates the unit is not being used on that clock cycle: 
- IF ID EX MEM WB
- IF ID EX MEM WB

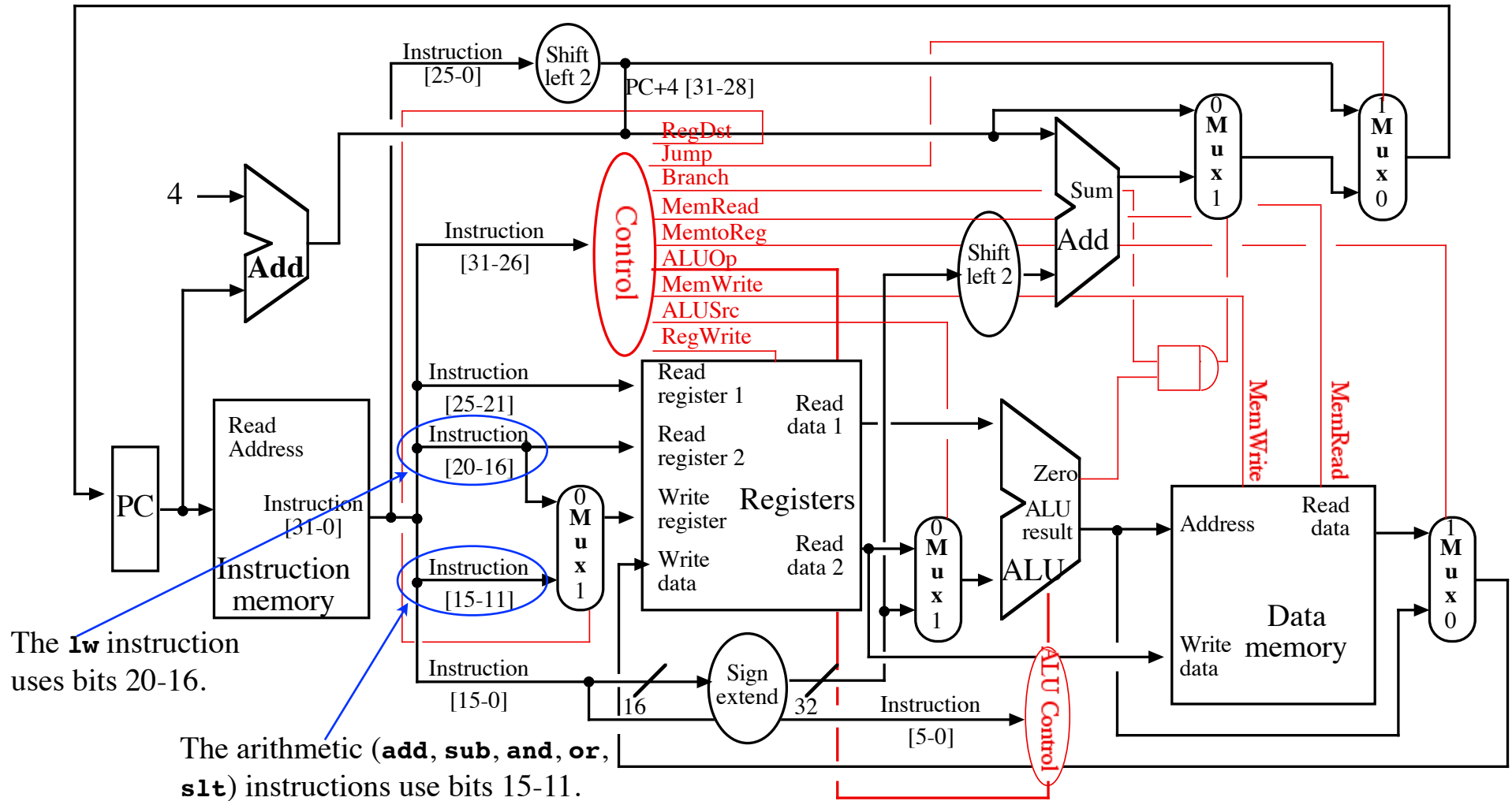
Representing Pipelines (continued):

- Can help with answering questions such as:
 - How many clock cycles does it take to execute this code?
 - What is the ALU doing during clock cycle 4? What else is happening during clock cycle 4?
- Can use this representation to help understand datapaths through the CPU.



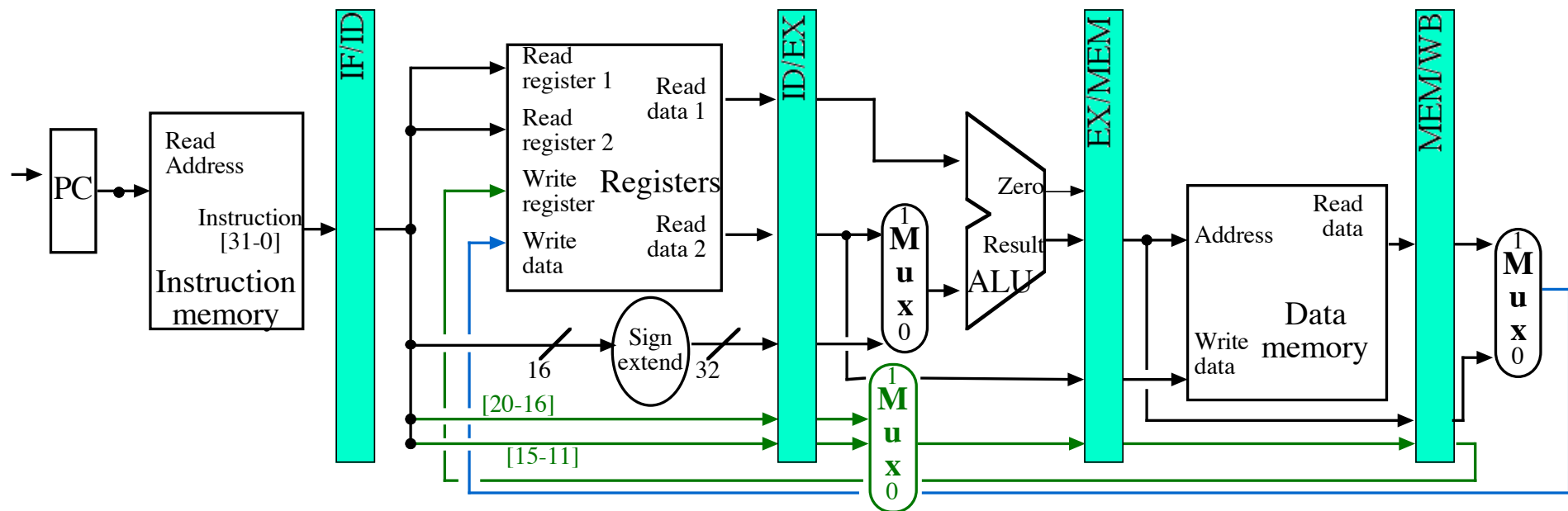
Pipeline Control:

- Control wires are (more or less) the same ones we used before (as per the single-clock cycle implementation).



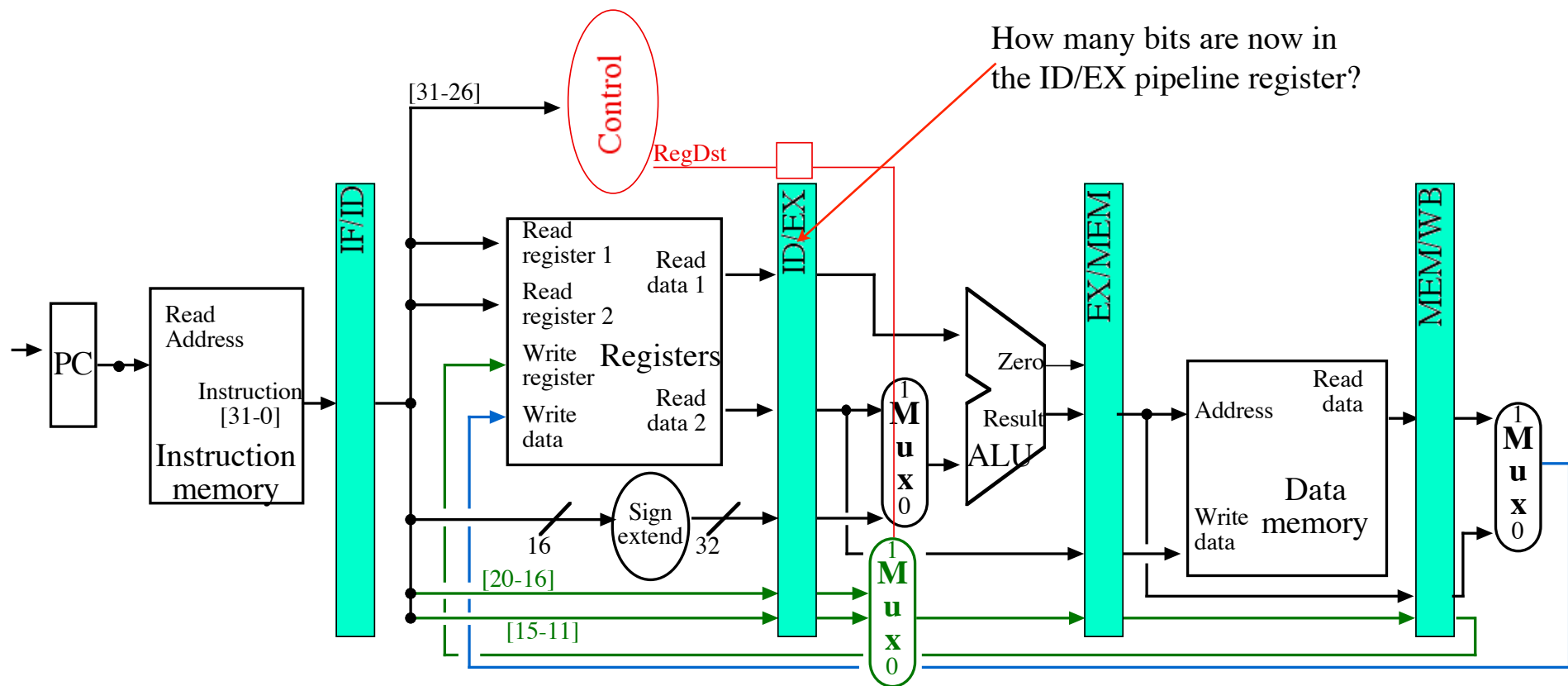
Pipeline Control (continued):

- A specific example: the number of the register to which the result is written.
 - The **lw** instruction uses bits 20-16.
 - The arithmetic (**add**, **sub**, **and**, **or**, **slt**) instructions use bits 15-11.
- We do not know which set of bits to use until the end of the second clock cycle.
 - Therefore, the control wire cannot be turned on or off until the third clock cycle.
 - The multiplexor has to be in the third clock cycle portion of the CPU.

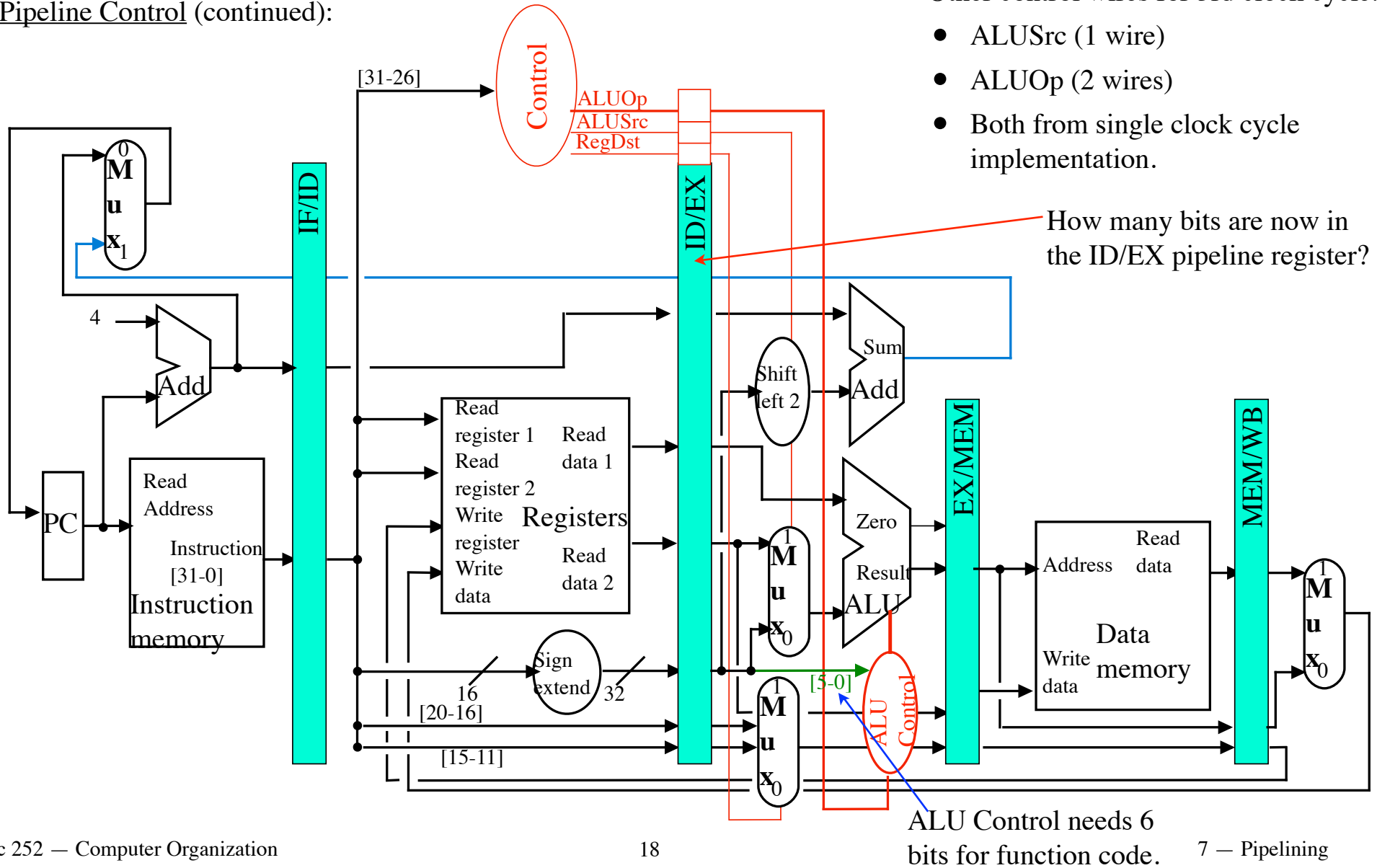


Pipeline Control (continued):

- The second clock cycle has the Control unit. It gets the opcode from wires 31-26.
- Control turns the RegDst control wire on or off.
- Store the RegDst control wire in the ID/EX pipeline register for use during the 3rd clock cycle.



Pipeline Control (continued):

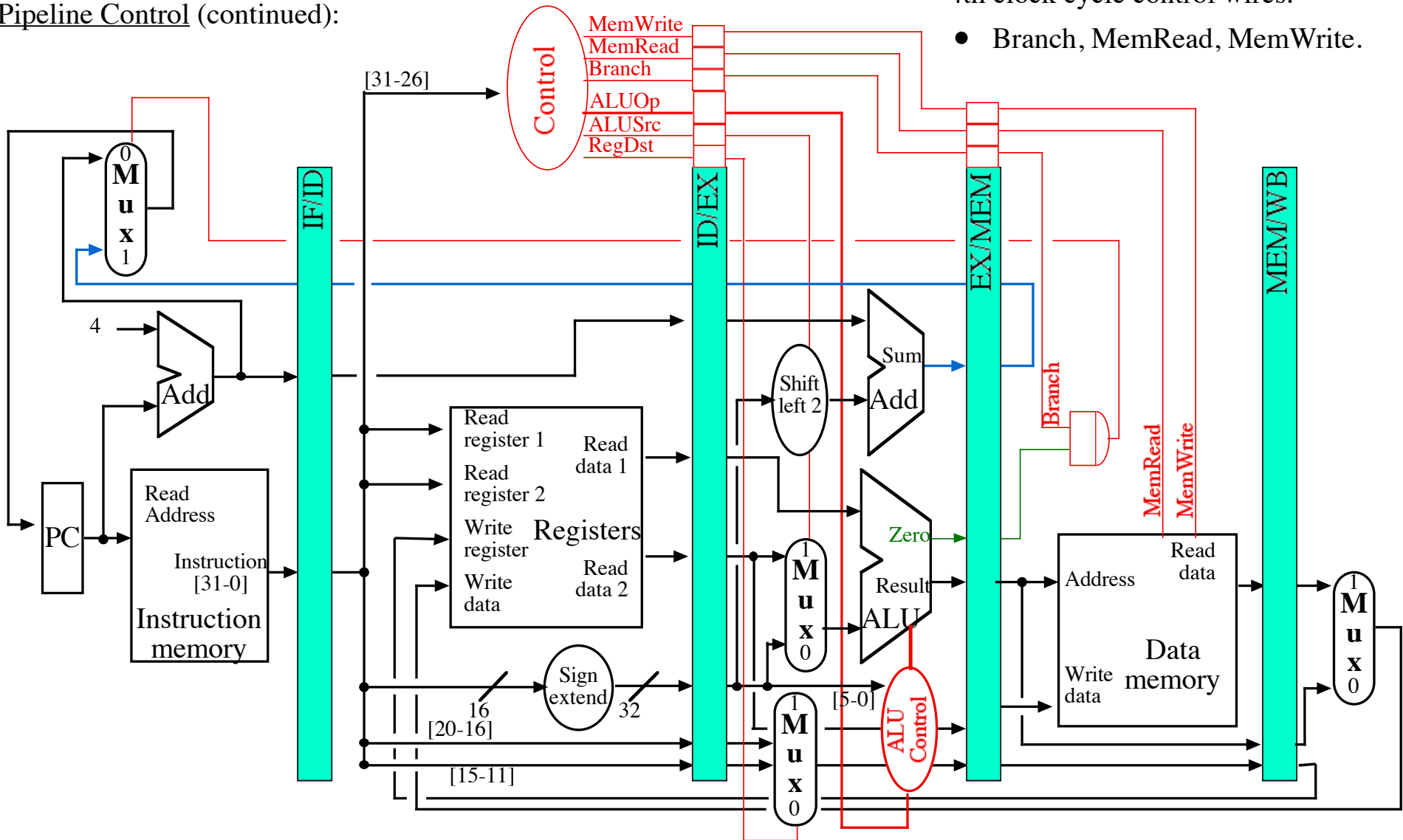


- Other control wires for 3rd clock cycle:
 - ALUSrc (1 wire)
 - ALUOp (2 wires)
 - Both from single clock cycle implementation.

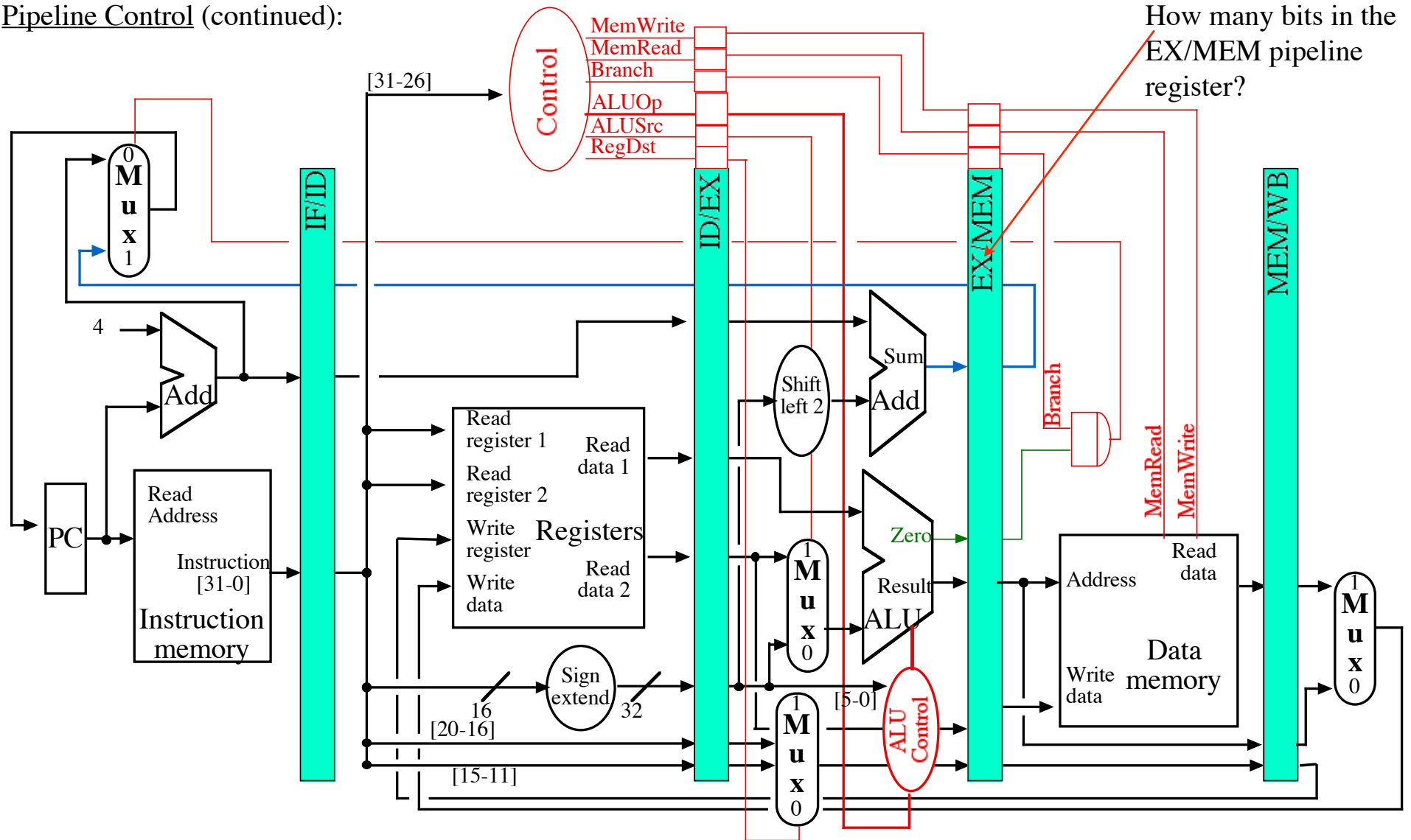
How many bits are now in the ID/EX pipeline register?

Pipeline Control (continued):

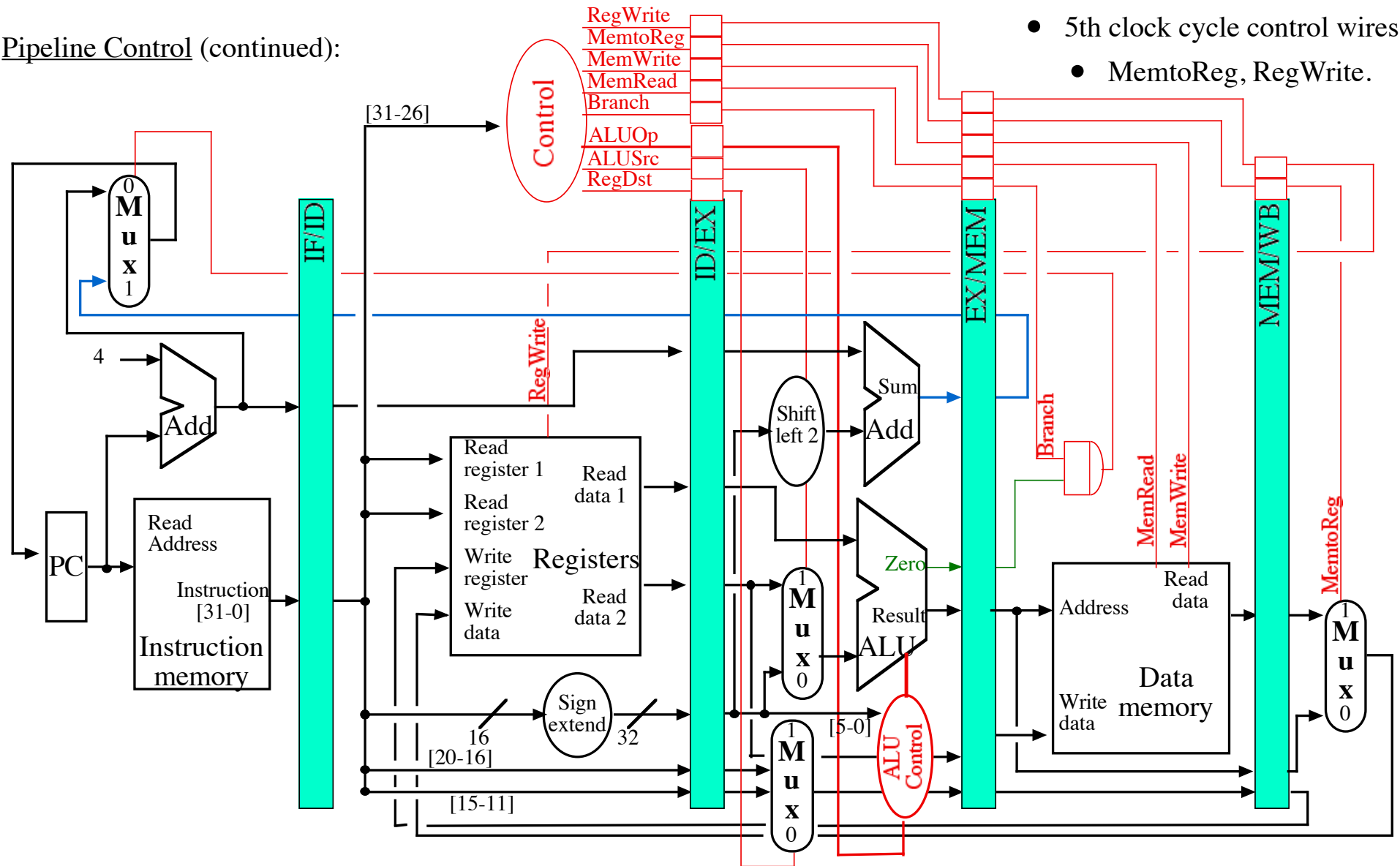
- 4th clock cycle control wires:
- Branch, MemRead, MemWrite.



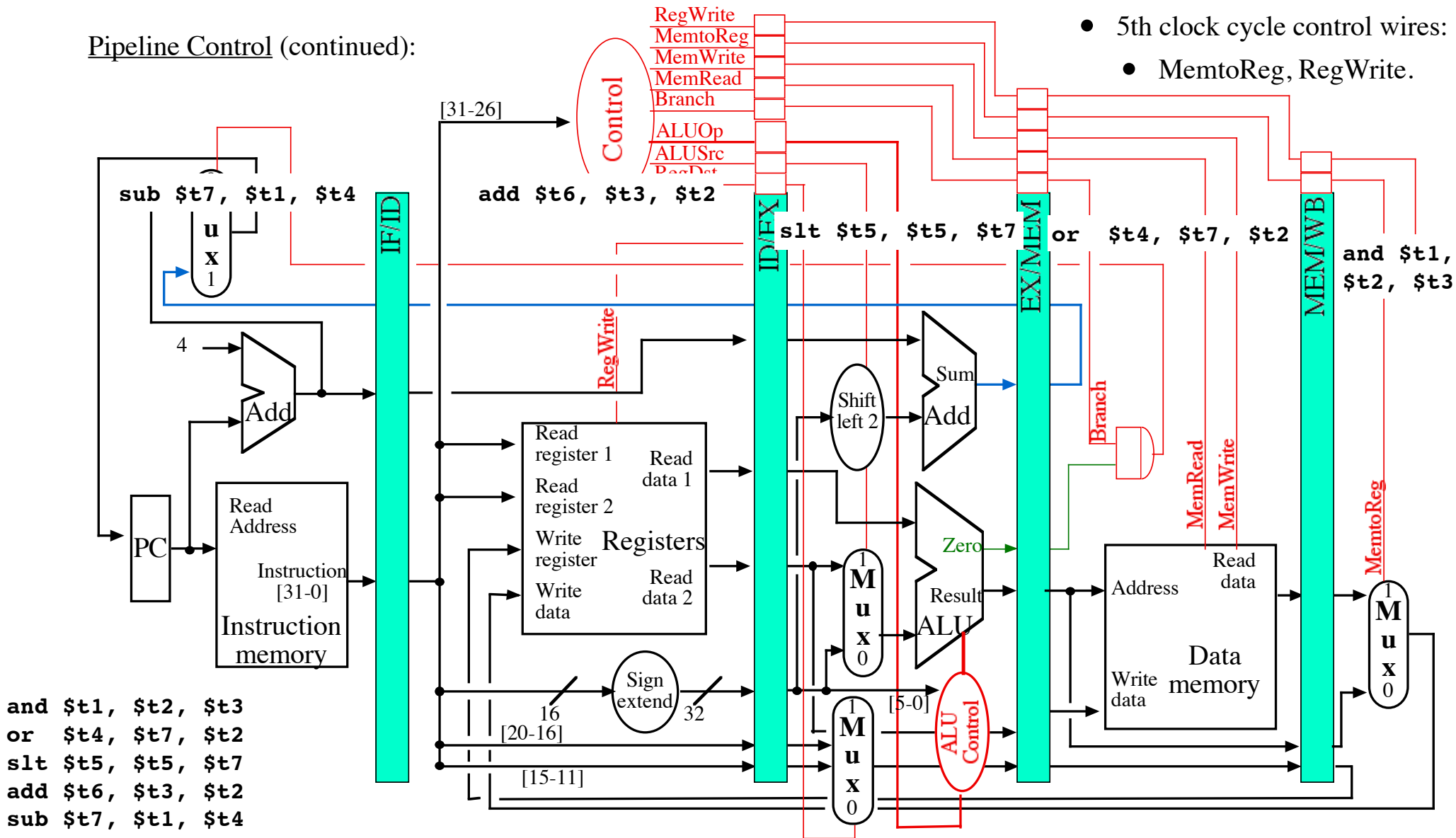
Pipeline Control (continued):



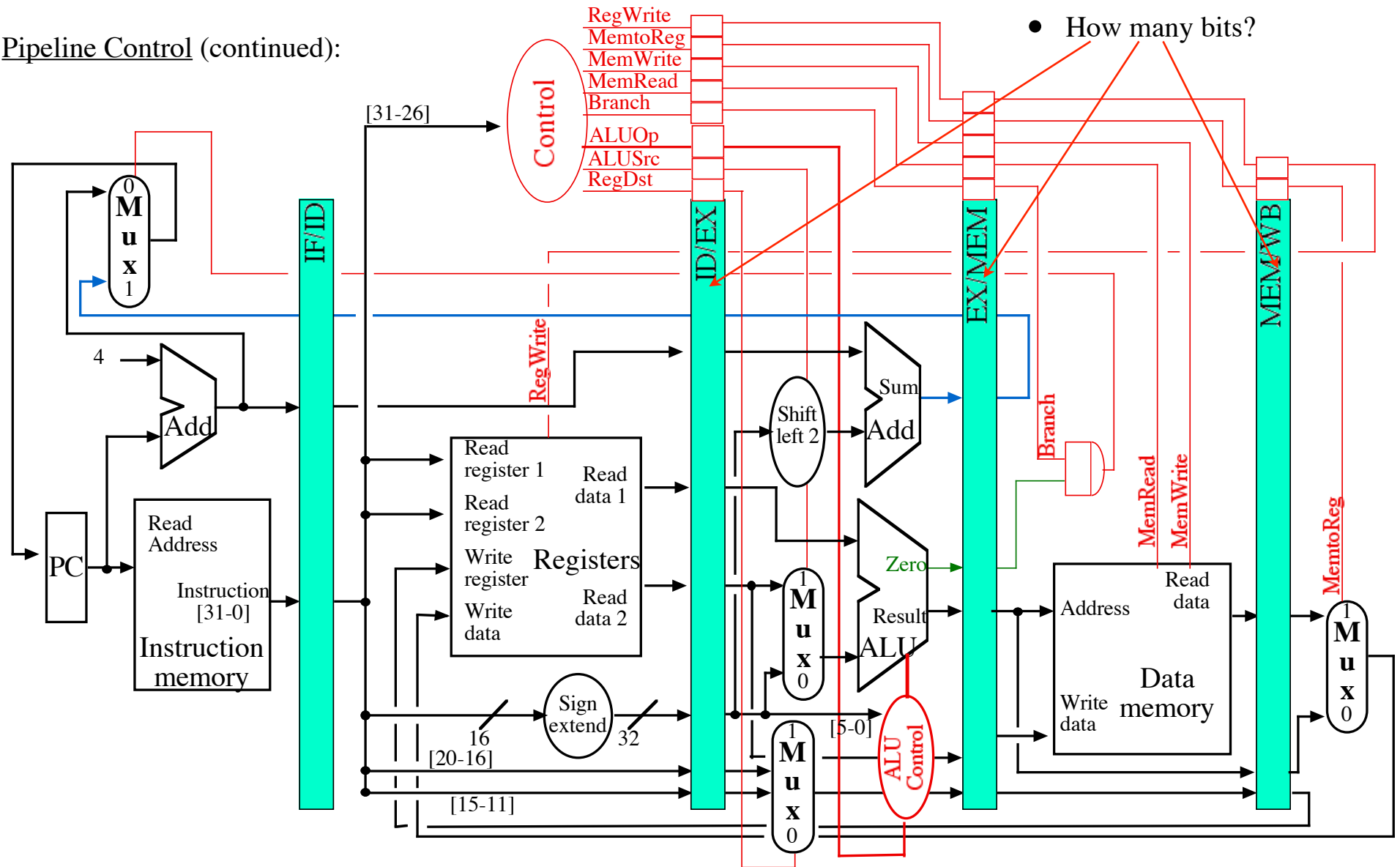
Pipeline Control (continued):



Pipeline Control (continued):



Pipeline Control (continued):



Pipeline Control (continued):

- What are the settings needed for each control line on each stage of the pipeline?

Instruction	Execution/Address Calculation 3rd Stage				Memory Access 4th Stage			Write-back 5th Stage	
	RegDst	ALUOp1	ALUOp0	ALUSrc	Branch	MemRead	MemWrite	RegWrite	MemtoReg
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

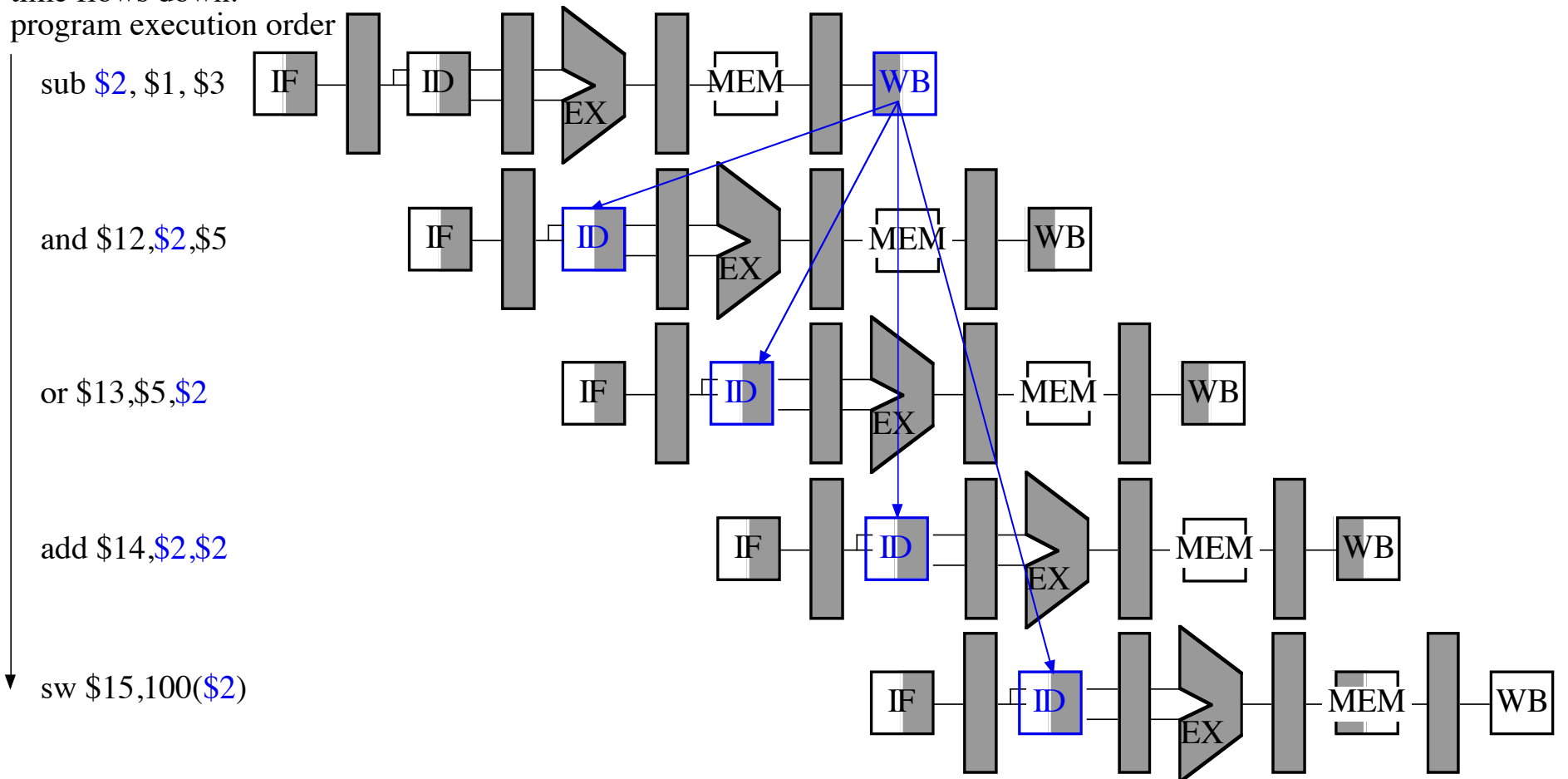
- Only need to look at stages 3, 4, and 5.
 - Why not 1 and 2?
- X = “does not care”
- Taken from the single-clock cycle, but re-arranged to fit the stages for pipelining.
 - See figure 4.49, page 360 (4th edition).

Data Hazards and Forwarding:

- Problems can arise when the next instruction is started before the current one is finished.
- Dependencies that “go backward in time” are data hazards.

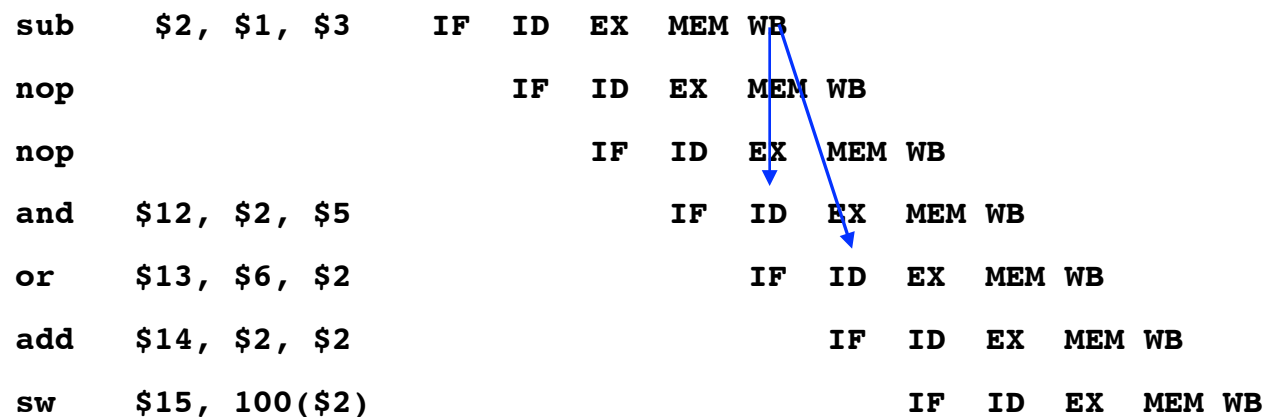
time flows down:

program execution order



Data Hazards and Forwarding (continued):

- Solution: All assembly languages have a “no operation” instruction.
 - **nop** in MIPS.
 - The CPU does nothing on a **nop** operation.
 - Have the compiler insert **nop**’s as needed so the CPU “waits” for the value to be written into \$2.
 - Where do we insert the **nop**’s?

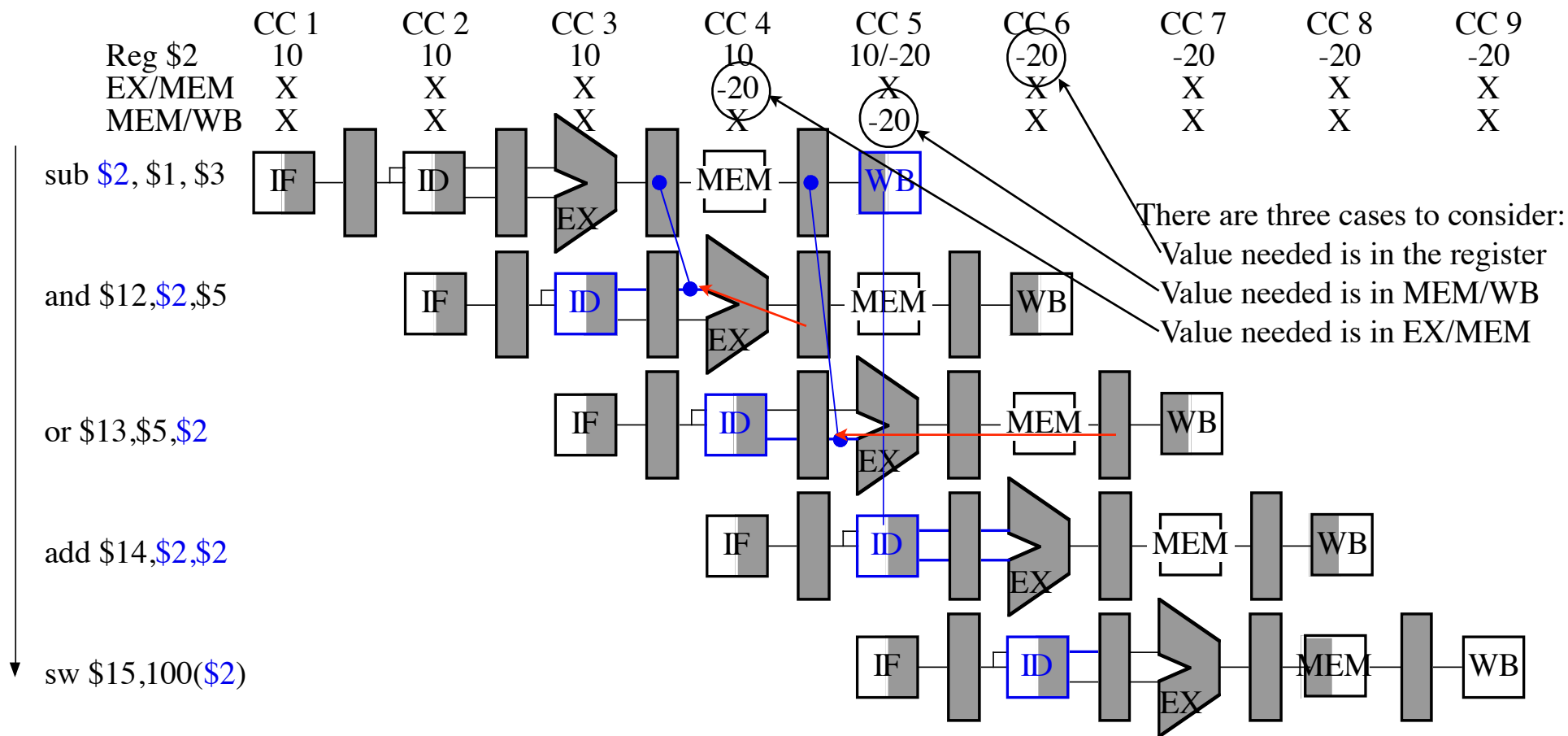


- What is wrong with this approach?

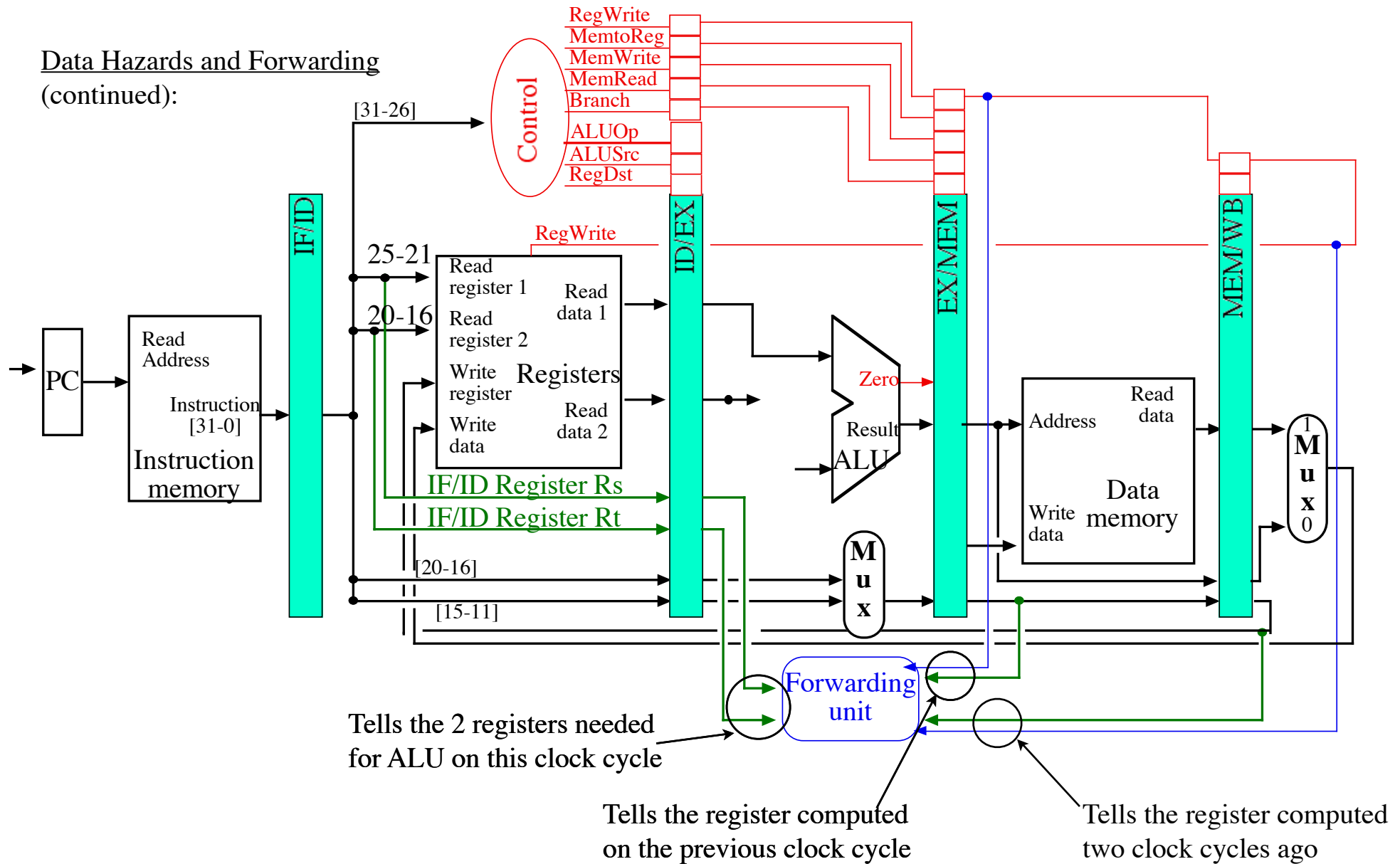


Data Hazards and Forwarding (continued):

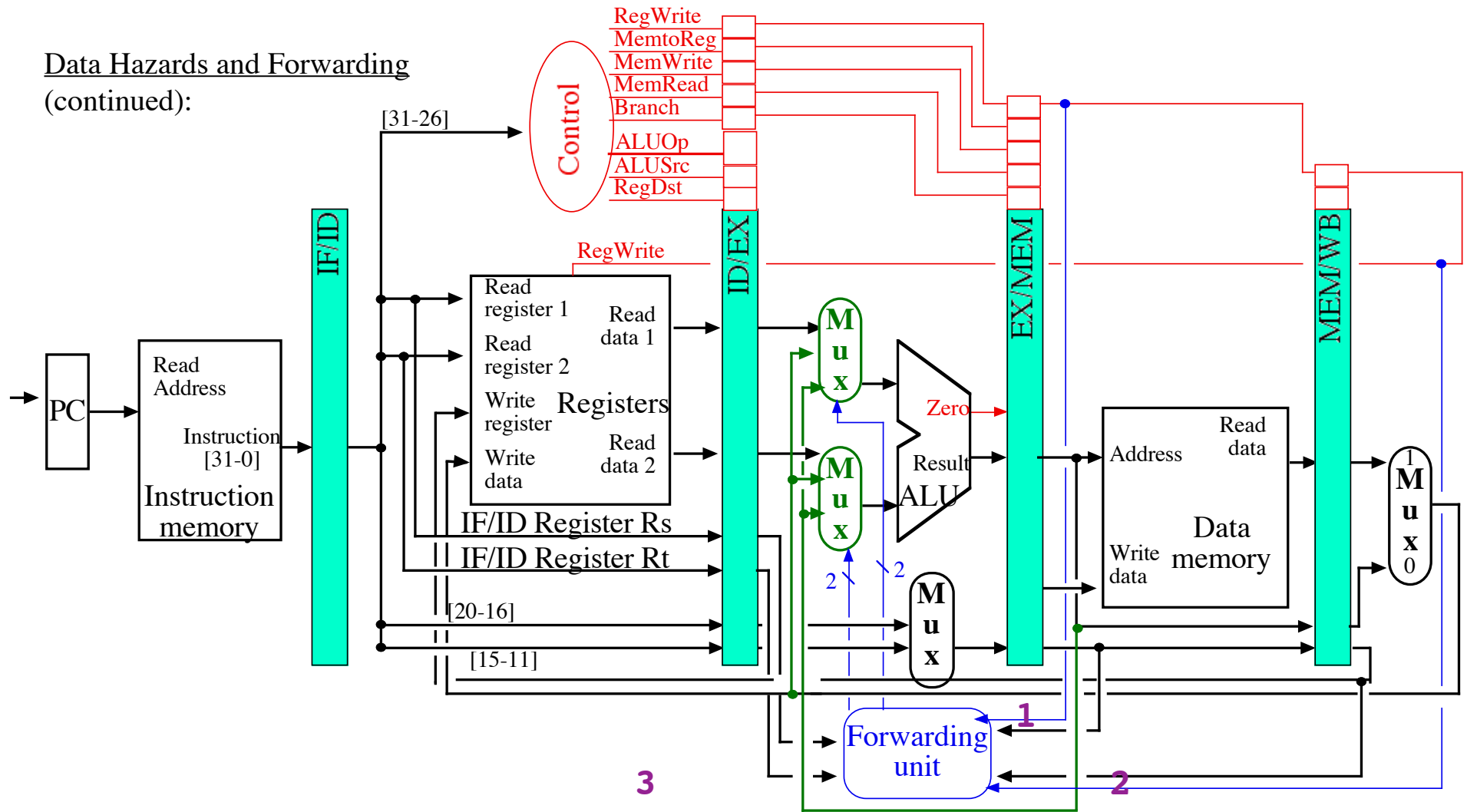
- The value we need is present in a pipeline register — Use it instead of waiting.
- Register file forwarding to handle read/write to same register:



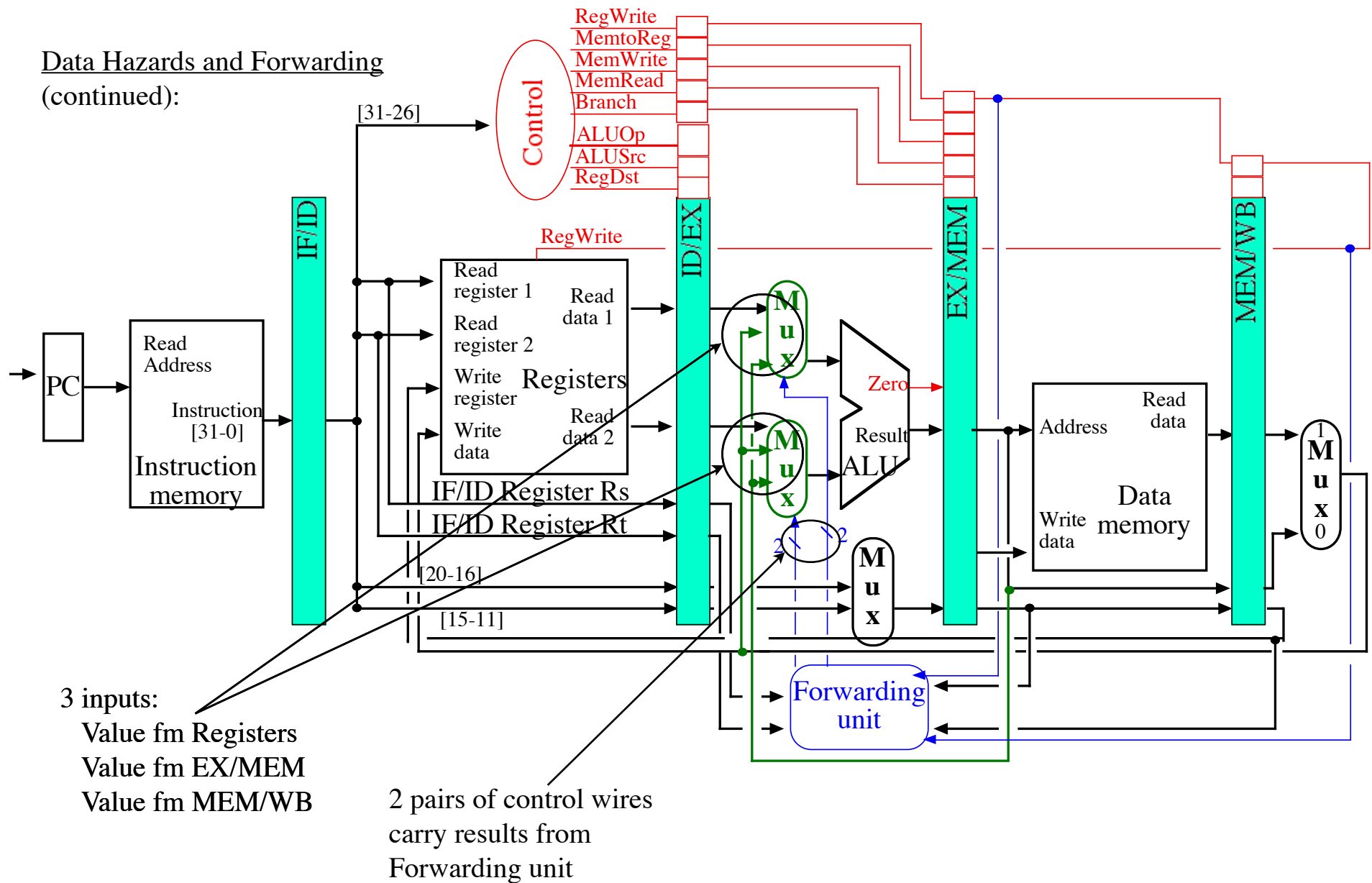
Data Hazards and Forwarding (continued):



Data Hazards and Forwarding (continued):

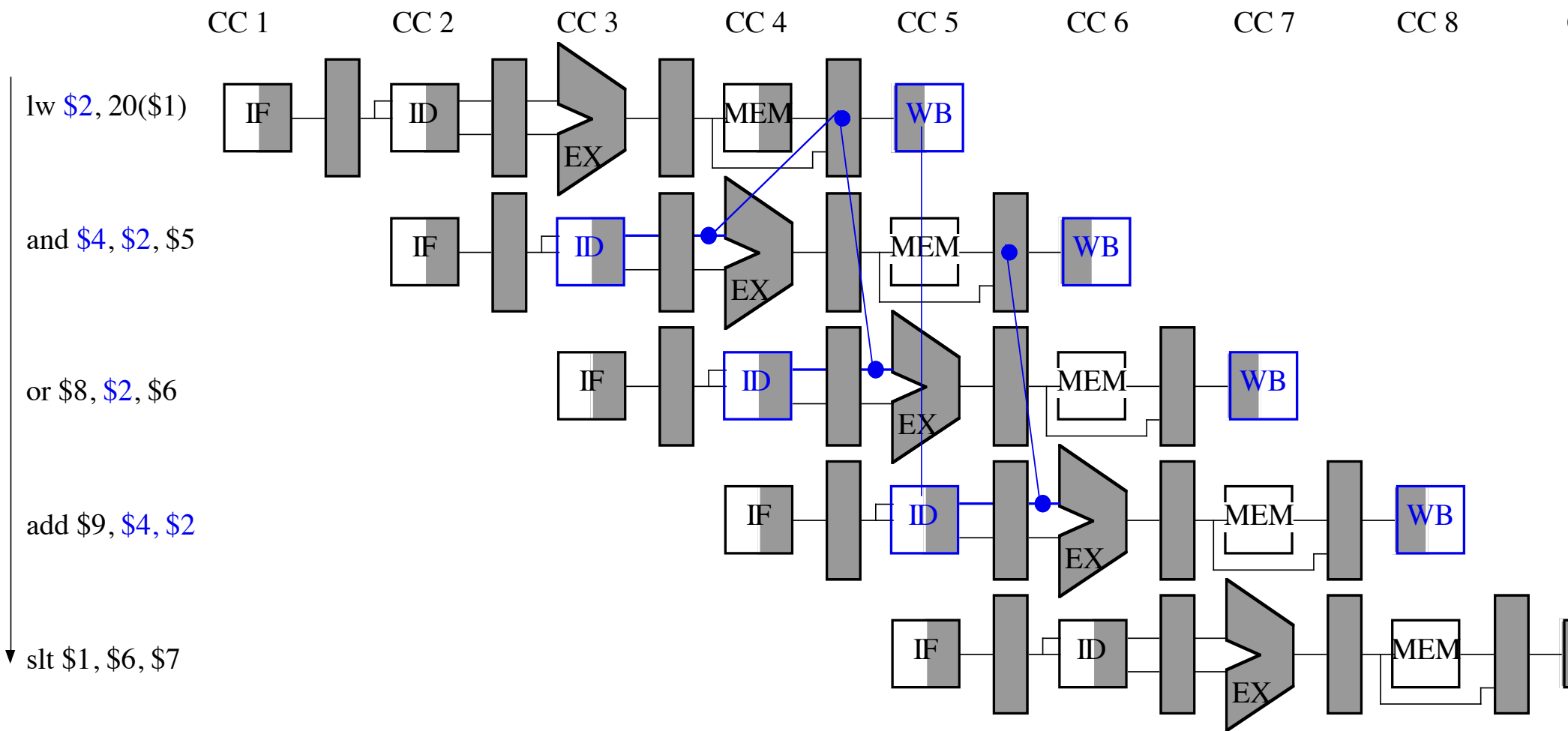


Data Hazards and Forwarding (continued):



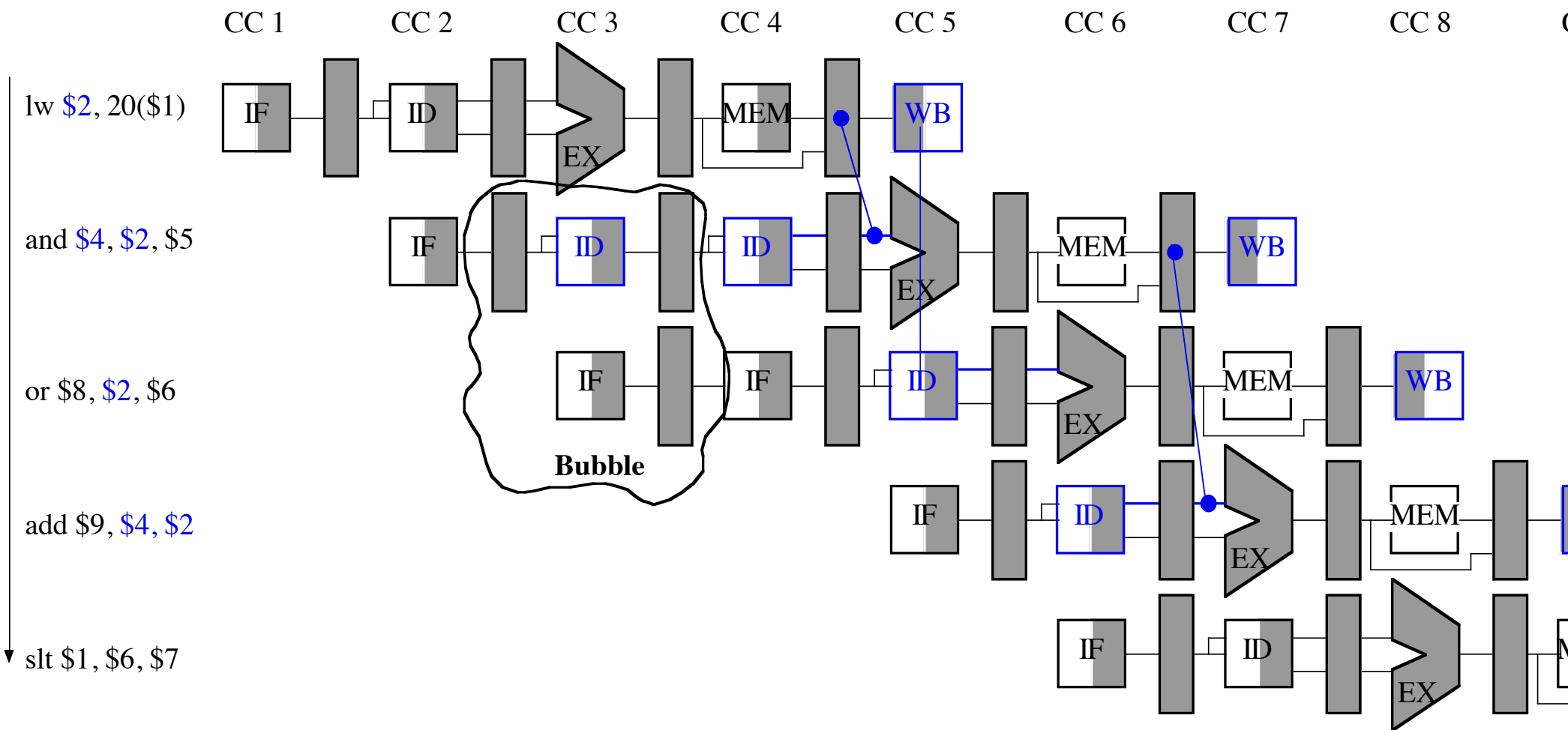
Data Hazards — Stalls:

- Load word can still cause a hazard — try to read register following a load instruction that writes that register.



Data Hazards — Stalls (continued):

- Stall the pipeline by keeping an instruction in the same stage.



op	rs	rt	rd	shamt	funct
----	----	----	----	-------	-------

MemRead, to determine if memory will be read

op	rs	rt	16 bit number
----	----	----	---------------

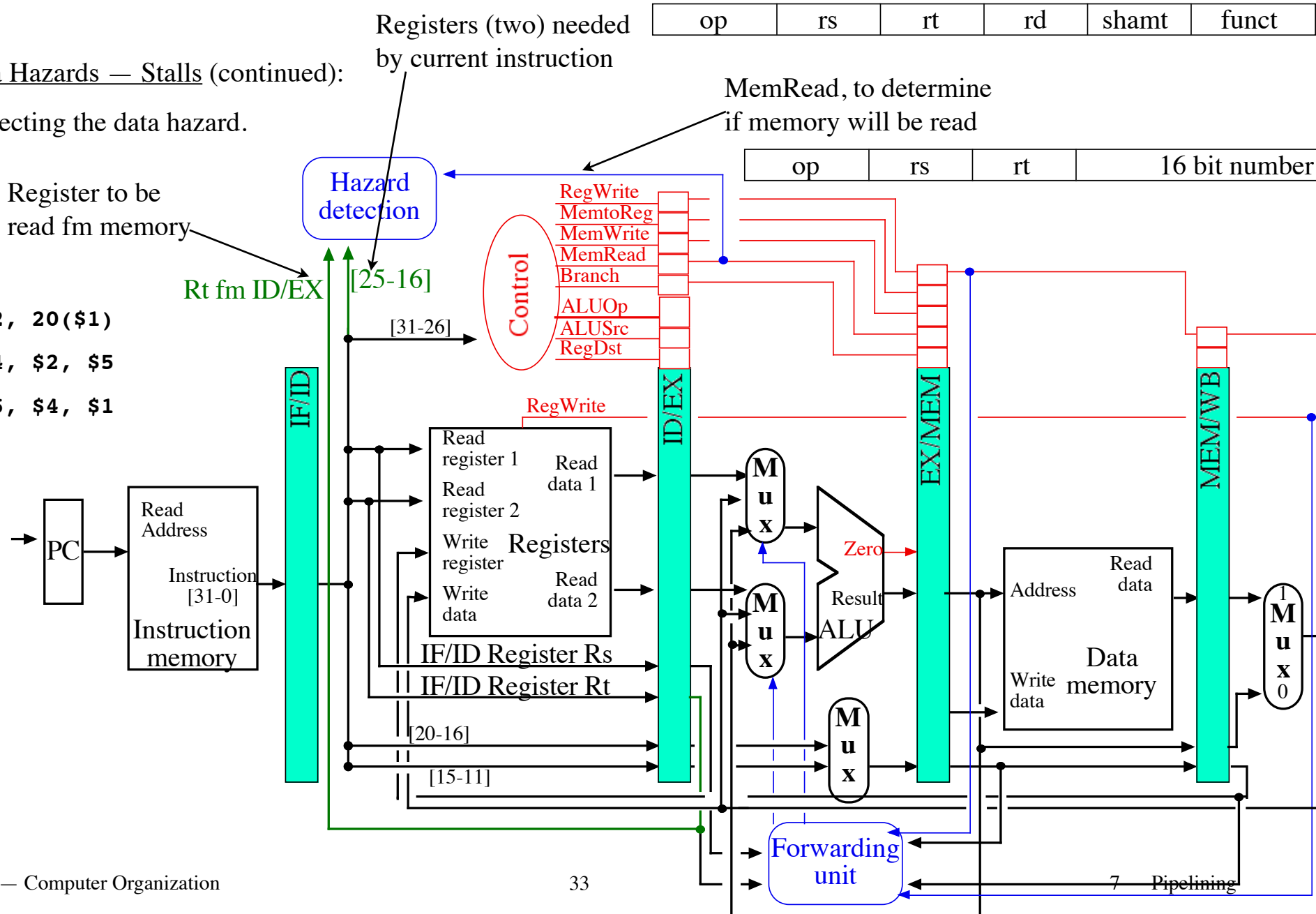
Data Hazards — Stalls (continued):

- Detecting the data hazard.

Register to be read from memory

Rt from ID/EX

```
lw $2, 20($1)
and $4, $2, $5
or $5, $4, $1
```

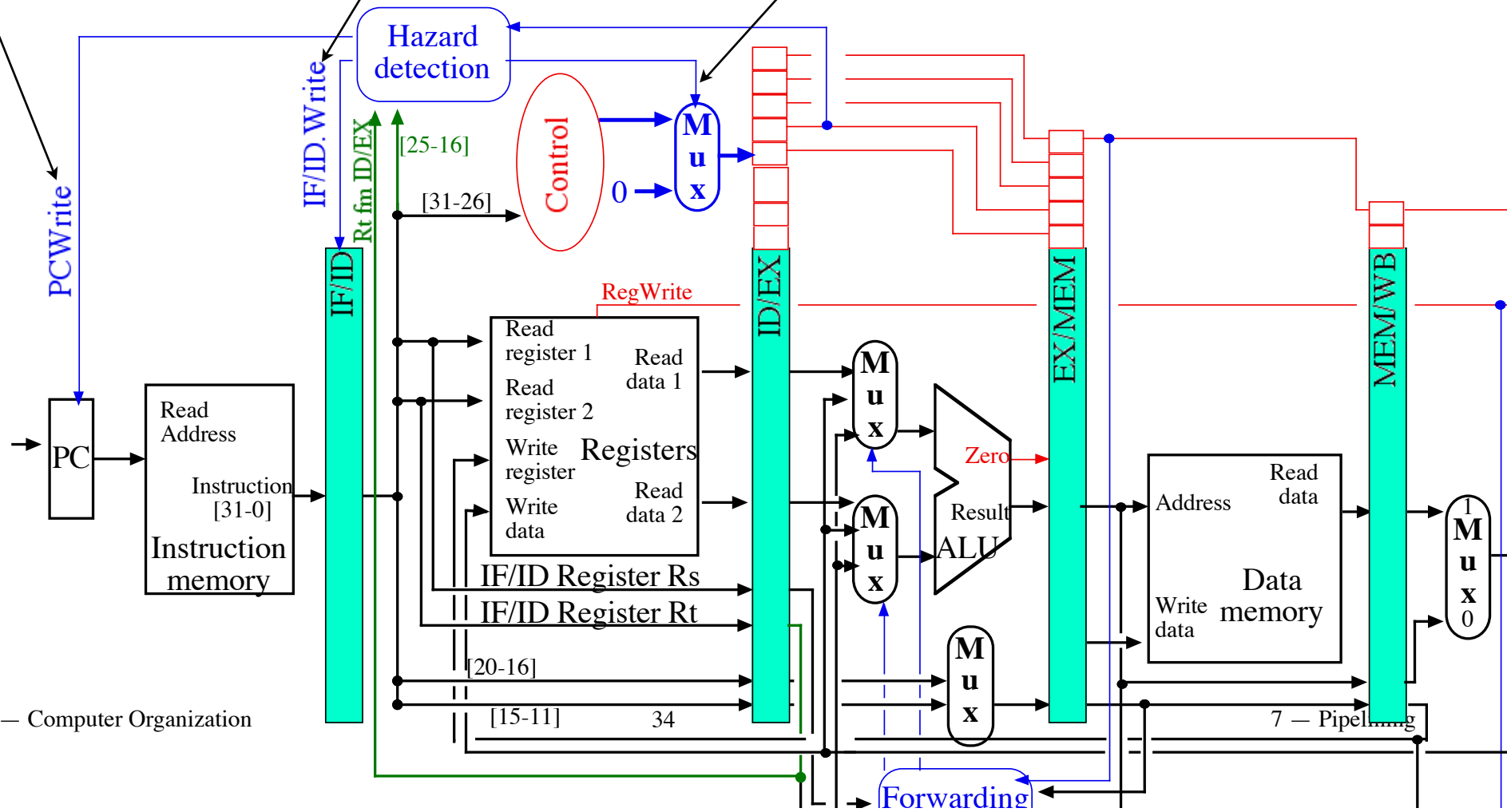


Data Hazards — Stalls (continued):

Add Control wire:
Determine if PC does
or does not change

Add control wire:
Determine if new
instruction is written
to IF/ID

Add Multiplexor:
What control wires are sent to later cycles



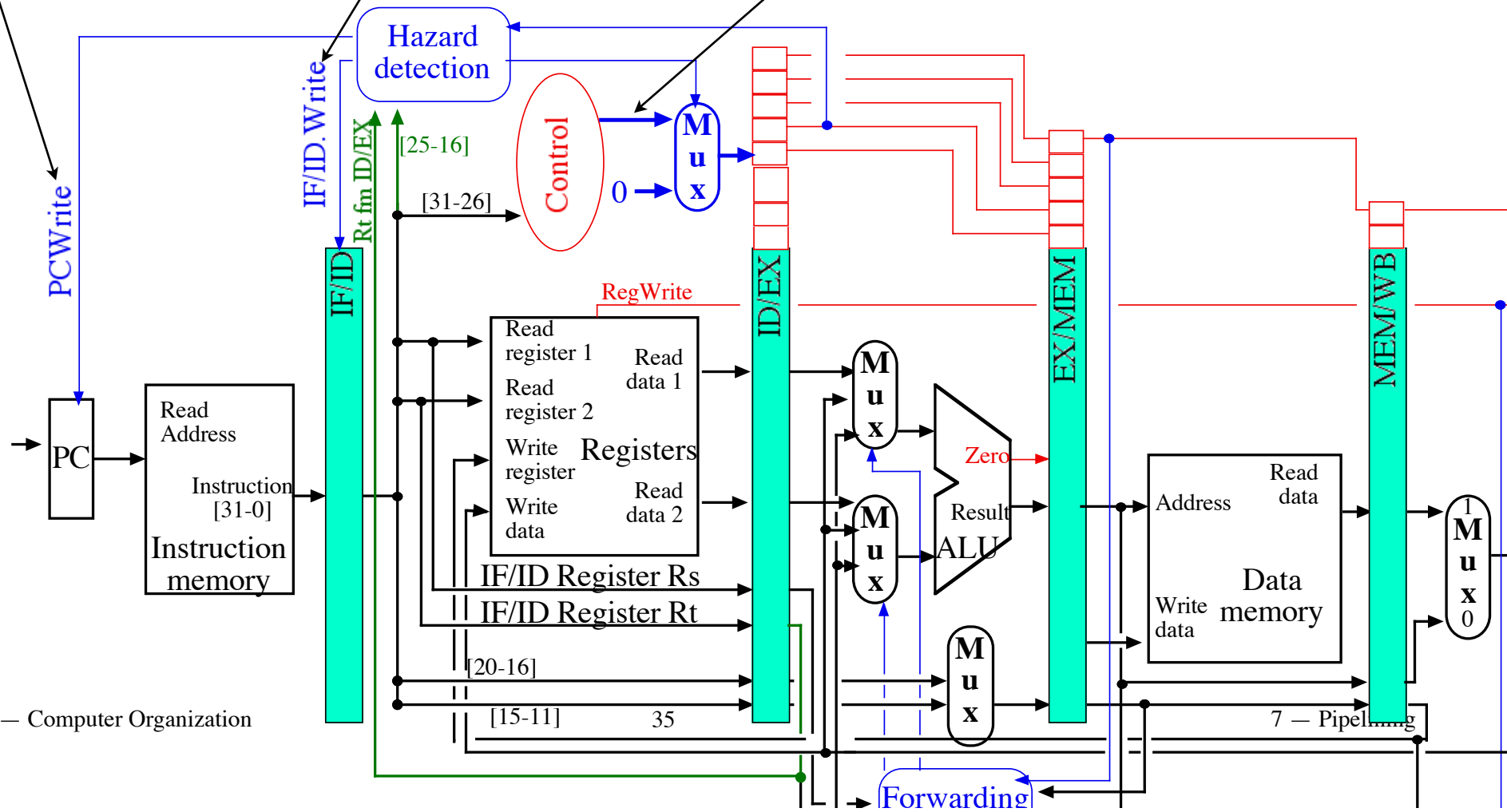
Data Hazards — Stalls (continued):

No stall necessary!

Choose: PCWrite On
Change PC for next instruction

Choose: IF/ID.Write On
Write instruction in IF/ID

Choose:
Control wire values to be passed to later cycles



Data Hazards — Stalls (continued):

Choose: PCWrite Off

PC stays the same to re-read instruction

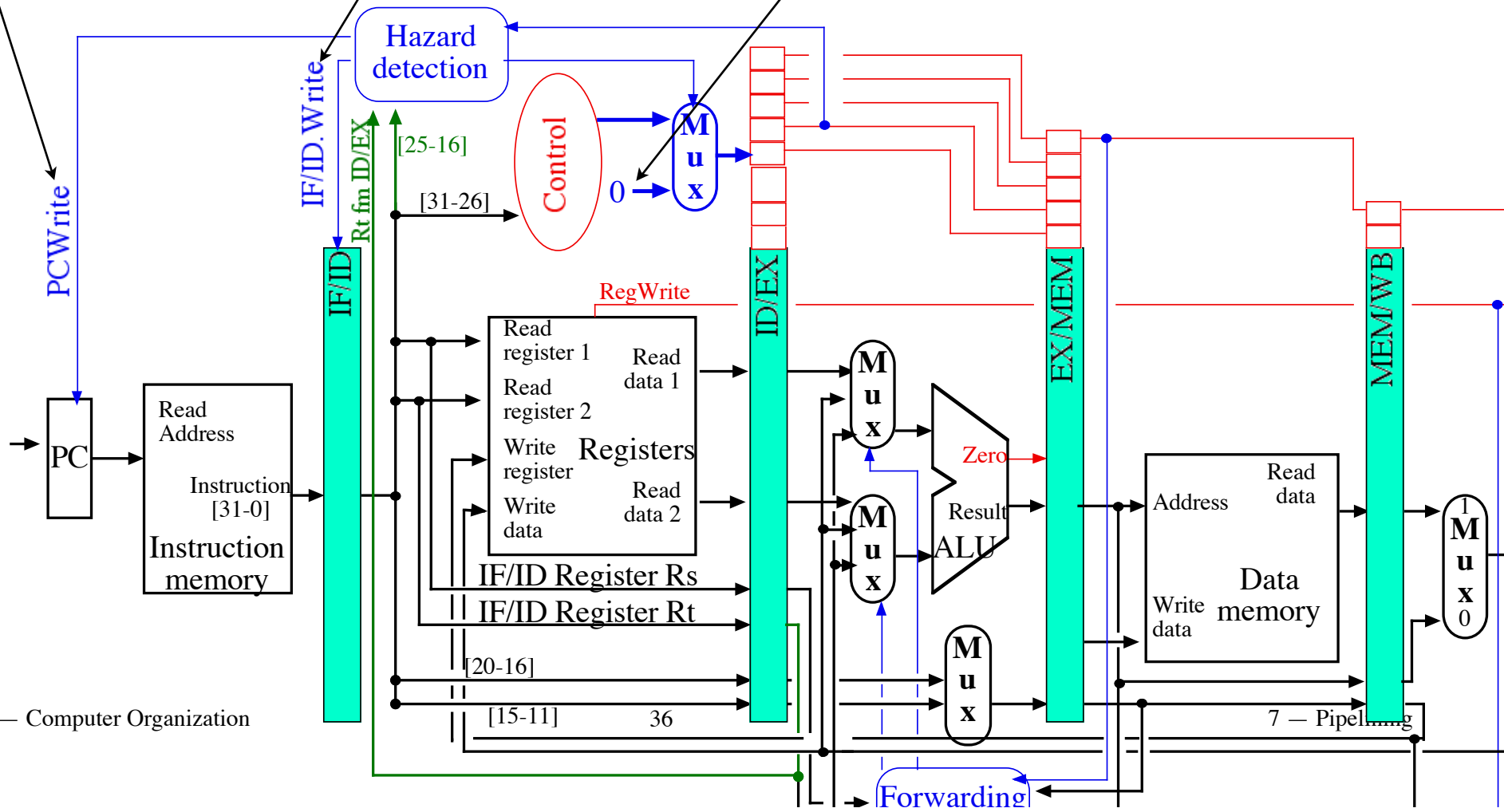
Choose: IF/ID.Write Off

Leave previous instruction in IF/ID

Choose:

All control wires turned off for stall

Stall is needed!

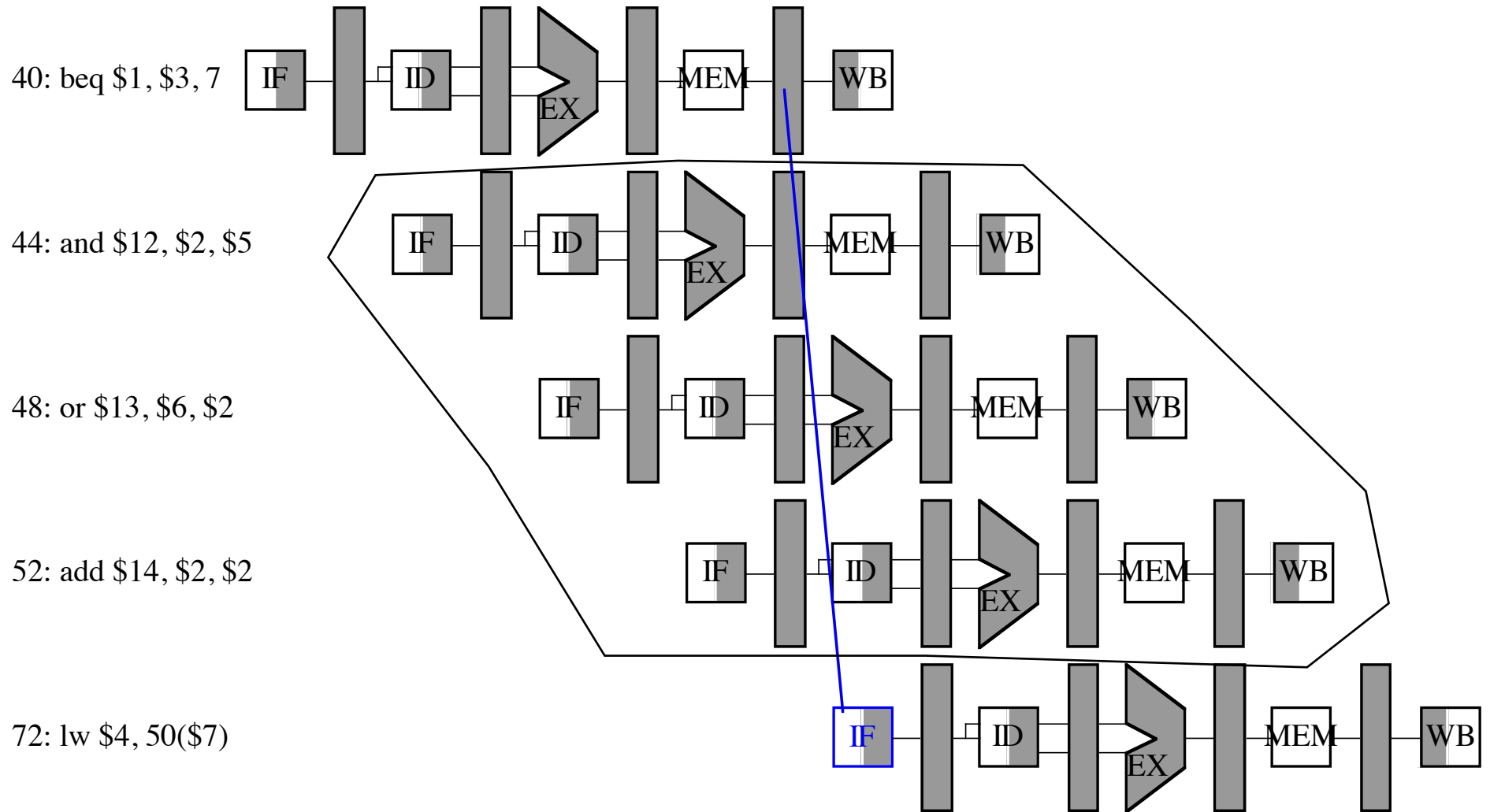


Branch Hazards:

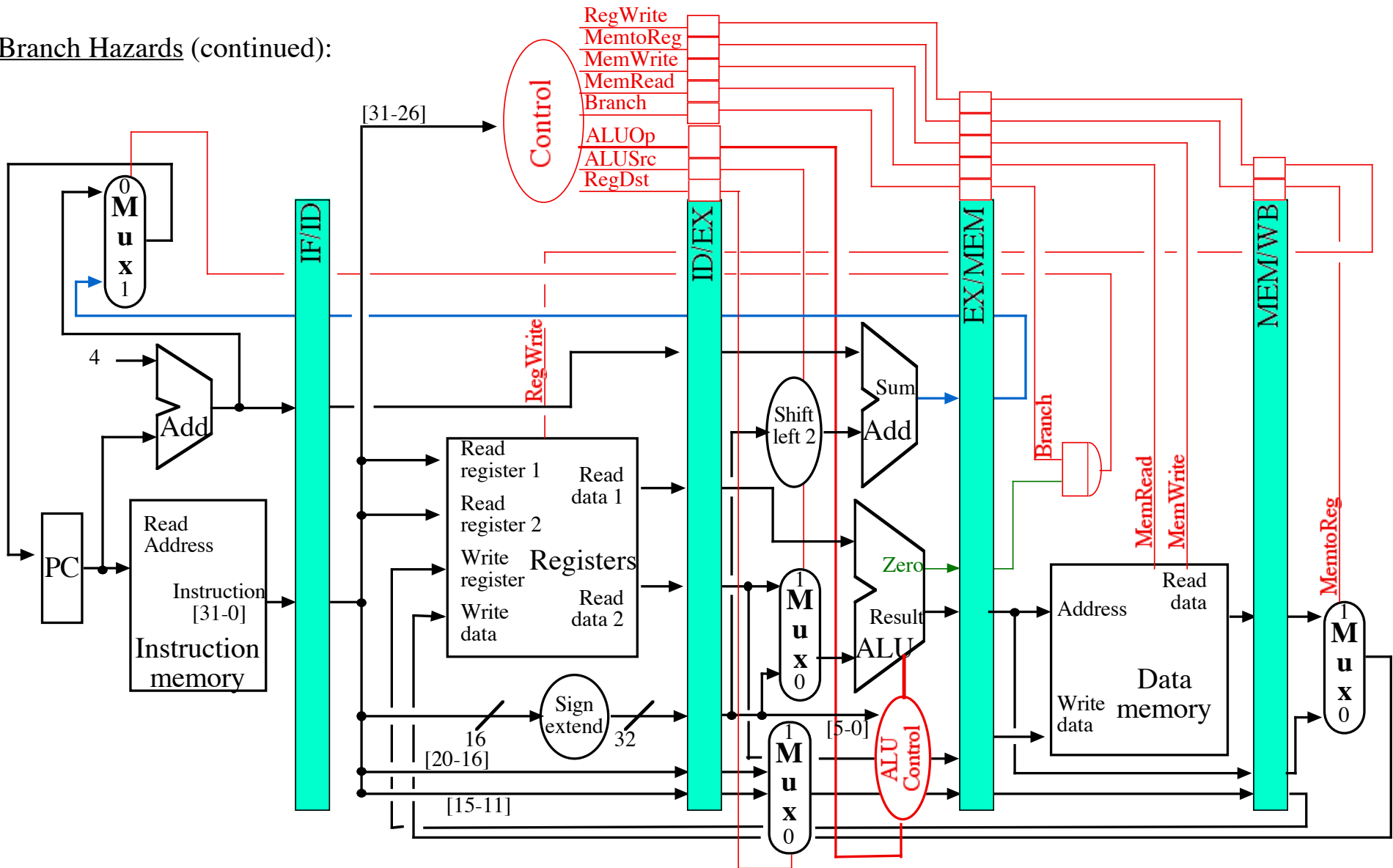
- Pages 339-343 of Section 4.5 and Section 4.8 (4th edition)
- For **beq**, the current design:
 - 2nd clock cycle: Fetches the registers and decodes the **beq**.
 - 3rd clock cycle:
 - **ALU** subtracts to determine if two registers are equal
 - Special-purpose **Add** computes branch address
 - 4th clock cycle: If equal, branch occurs.
- Consider:

Branch Hazards (continued):

- The **and**, **or**, **add** instructions are already started by the time the **beq** changes the PC.



Branch Hazards (continued):



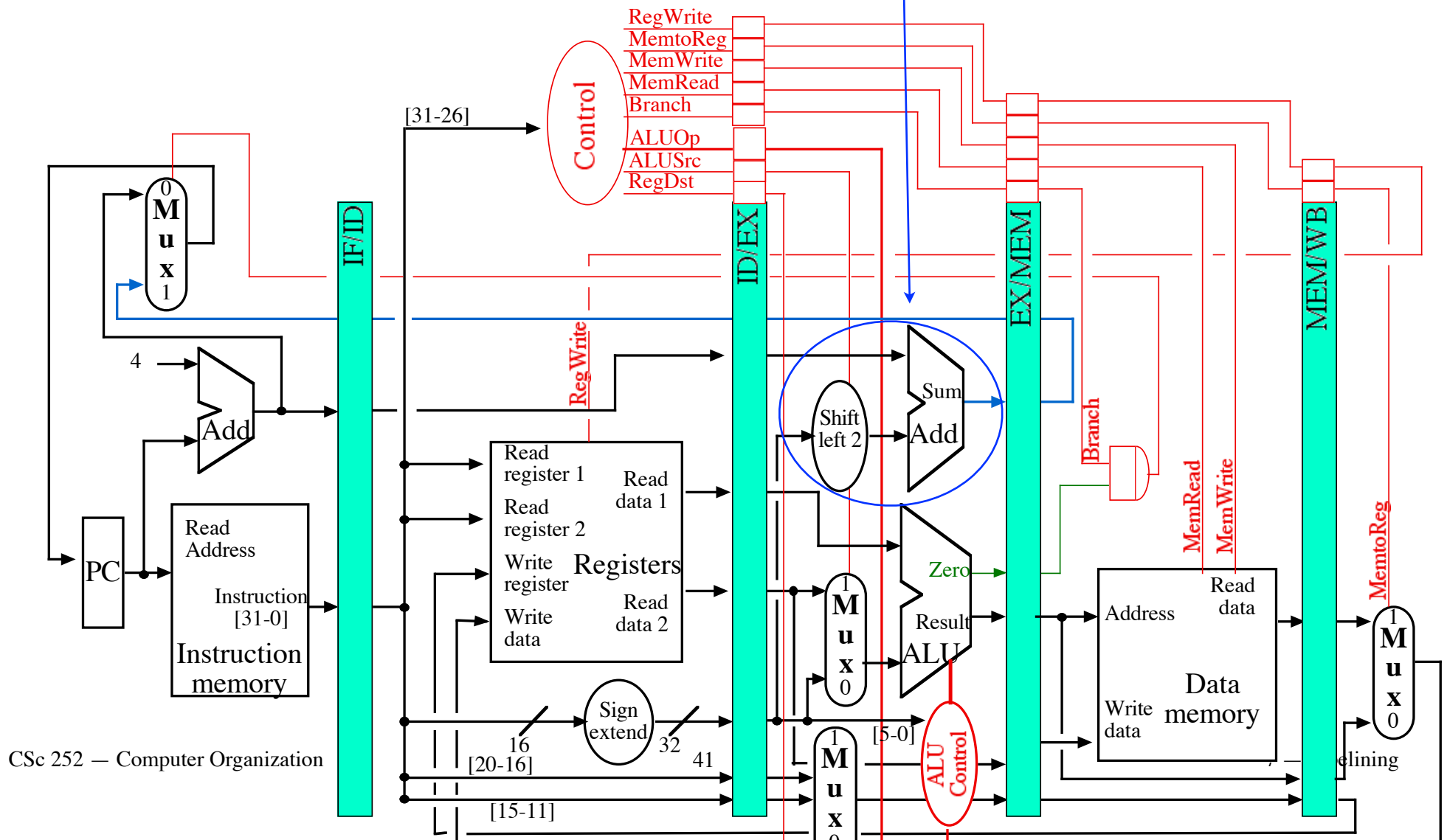
Branch Hazards (continued):

- Need to “predict” whether the **beq** is true or false.
 - Is this just a 50/50 guess?
 - Sometimes not:
 - ??
- Still, the prediction will not be correct all the time.
 - Need to reduce the loss when **beq** is true.

Branch Hazards (continued):

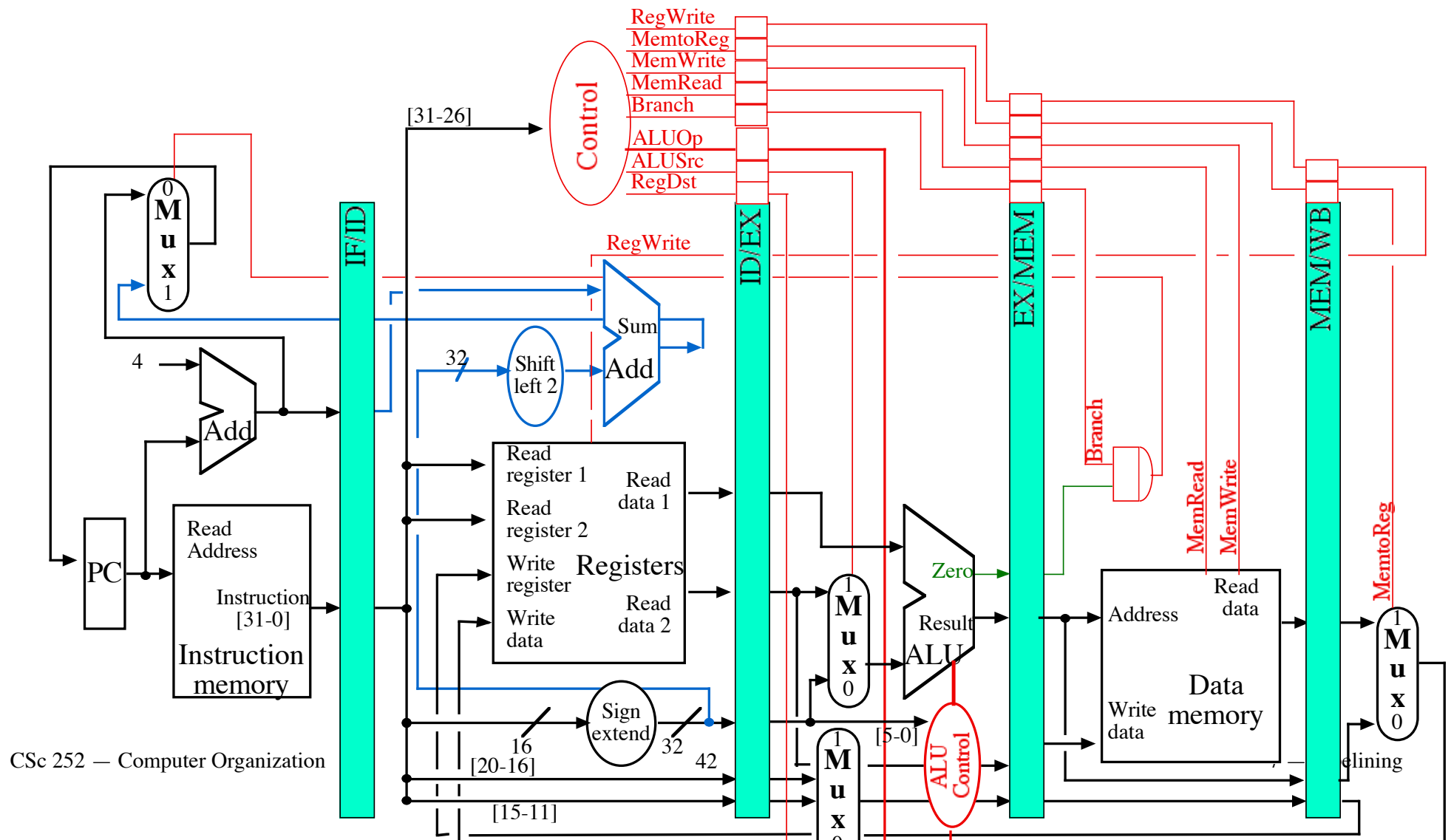
- Can we make the **beq** “faster” — use fewer clock cycles?

Move address computation to 2nd cycle.



Branch Hazards (continued):

- Compute the address while getting values from the registers.

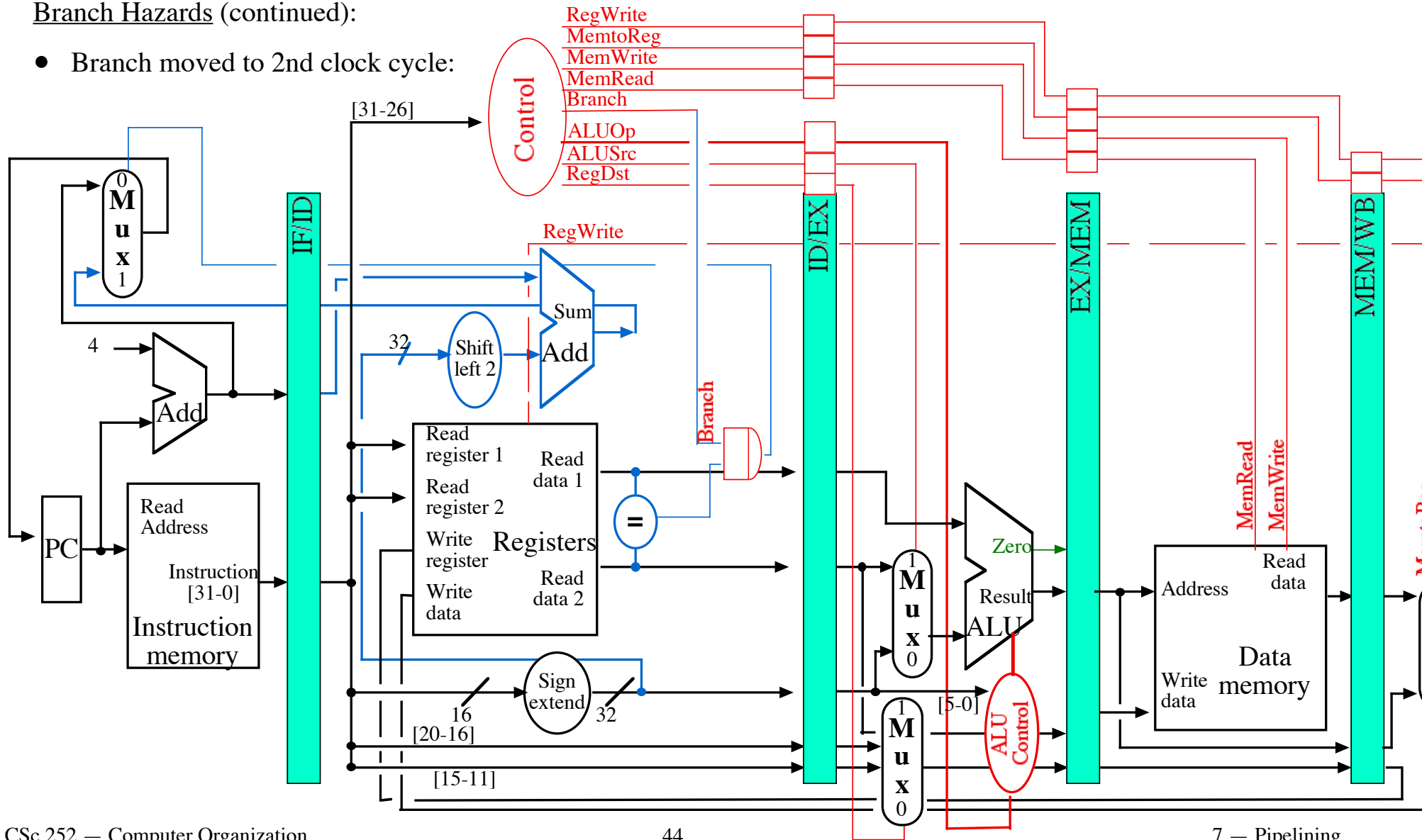


Branch Hazards (continued):

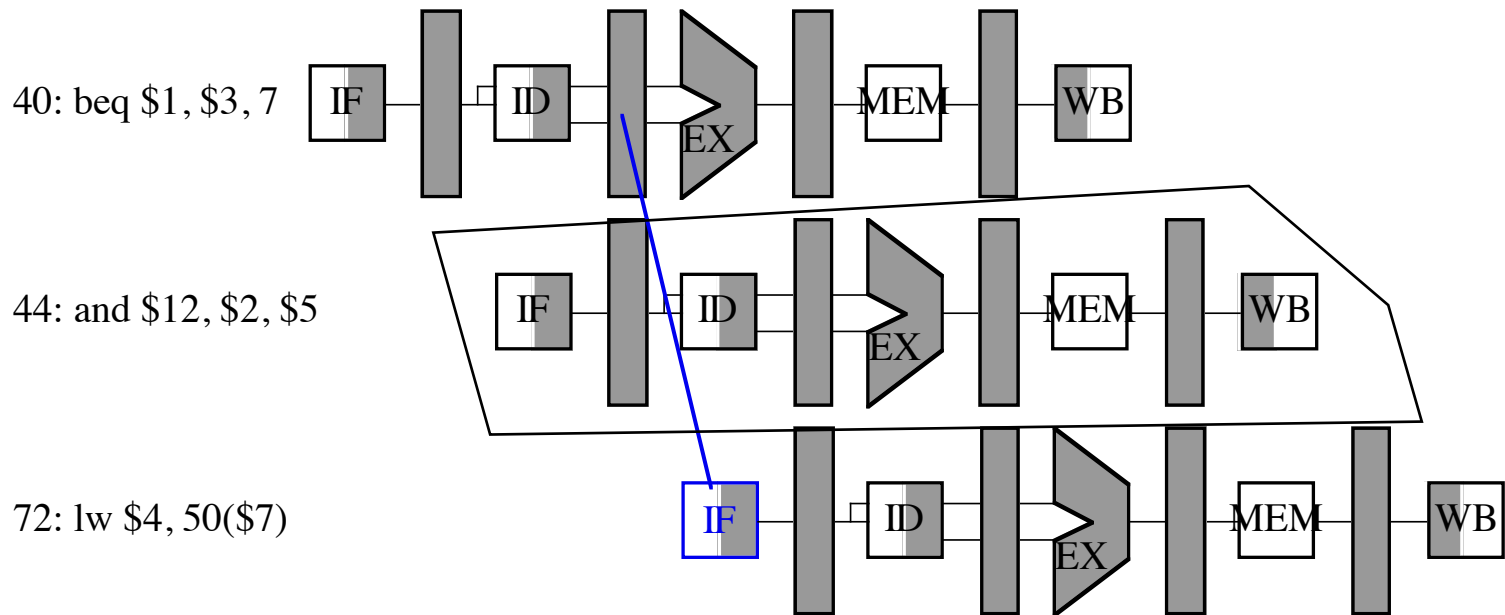
- How to compare the two register contents?
 - Not possible to put another ALU into the second clock cycle. Why?
 - Need a faster way to compare:

Branch Hazards (continued):

- Branch moved to 2nd clock cycle:



Branch Hazards (continued):



add \$9, \$7, \$8
sub \$4, \$7, \$1
beq \$1, \$3, ...
slt \$5, \$7, \$9

changes to

add \$9, \$7, \$8
beq \$1, \$3, ...
sub \$4, \$7, \$1
slt \$5, \$7, \$9

Branch Hazards (continued):

- There are still problems:

```
for ( i = 0; i < 10; i++ ) {
    x = 2 * 3.14159 * i;
    printf(x);
}
```

```
add    $1, $7, $8
sub    $1, $7, $1
beq    $1, $3, ...
nop
slt    $5, $7, $9
```

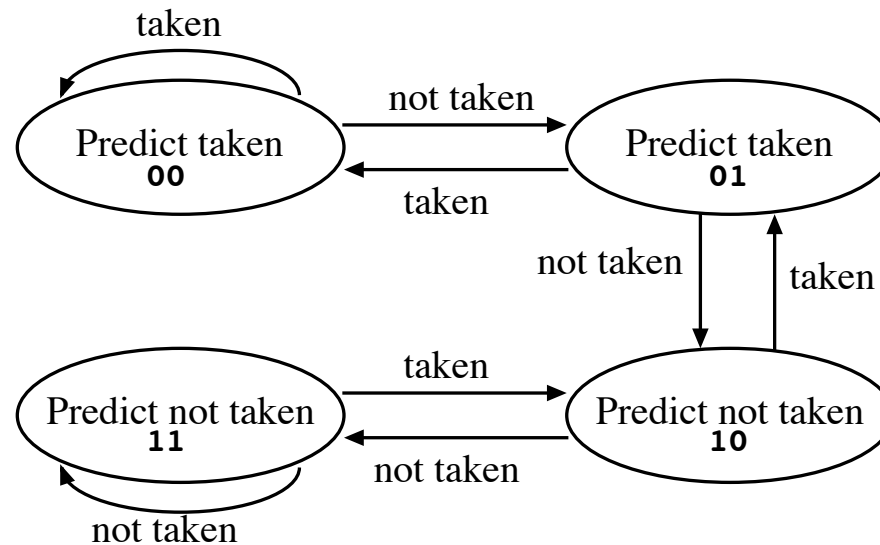
```
temp = 2 * 3.14159;
for ( i = 0; i < 10; i++ ) {
    x = x + temp * i;
    // printf(x);
}
```

Branch Hazards (continued):

- Rather than always assume “branch not taken”:
 - Remember what happened the last time the branch was executed.
 - Simple technique:
 - Store the PC for the branch and a single bit that indicates what happened the last time.
 - But, this can lead to problems:
 - Consider a loop executed 10 times.
 - Only correct 80% of the time. Why?

Branch Hazards (continued):

- Better approach: “Remember” two bits of information: (Figure 4.63, page 381)
 - Deterministic Finite State (DFS) Diagram:



Summary:

- Improving Performance:
 - Try and avoid stalls — re-order instructions when possible.
 - Can be done in hardware, or by the compiler.
 - Often by both!
 - Branch prediction to reduce problems with branch hazards.
 - Superscalar: start more than one instruction in the same cycle.
- Dynamic Scheduling:
 - Hardware tries to find instructions to execute.
 - Out-of-order execution is possible.
 - Speculative execution and dynamic branch prediction.
- You have the background you need to learn more.
 - And, there is more to learn about this!

Controlling Hardware:

- Reading: Appendix D, Sections D.1 and D.2.
- The MIPS ALU has 4 control wires and 6 operations that it performs.
 - MIPS defines NOR, which was not used by the subset we covered.
 - NOR requires the extra bit; the left-most Operation bit shown below
- Figure 4.12, page 317.

Instruction opcode	ALUOp		Instruction operation	Funct field						Operation
	ALUOp1	ALUOp0		F5	F4	F3	F2	F1	F0	
lw	0	0	load word	X	X	X	X	X	X	00 10
sw	0	0	store word	X	X	X	X	X	X	00 10
beq	0	1	branch equal	X	X	X	X	X	X	01 10
R-type	1	0	add	1	0	0	0	0	0	00 10
R-type	1	0	subtract	1	0	0	0	1	0	01 10
R-type	1	0	AND	1	0	0	1	0	0	00 00
R-type	1	0	OR	1	0	0	1	0	1	00 01
R-type	1	0	set less than	1	0	1	0	1	0	01 11

Controlling Hardware (continued):

- When is **operation2** a 1?

ALUOp		Function code fields					
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0
0	1	X	X	X	X	X	X
1	X	X	X	X	X	X	X

Instruction opcode	ALUOp		Instruction operation	Funct field						Operation
	ALUOp1	ALUOp0		F5	F4	F3	F2	F1	F0	
lw	0	0	load word	X	X	X	X	X	X	00 10
sw	0	0	store word	X	X	X	X	X	X	00 10
beq	0	1	branch equal	X	X	X	X	X	X	01 10
R-type	1	0	add	1	0	0	0	0	0	00 10
R-type	1	0	subtract	1	0	0	0	1	0	01 10
R-type	1	0	AND	1	0	0	1	0	0	00 00
R-type	1	0	OR	1	0	0	1	0	1	00 01
R-type	1	0	set less than	1	0	1	0	1	0	01 11