LECTURE 8

Pipelining:
Datapath and Control

PIPELINED DATAPATH

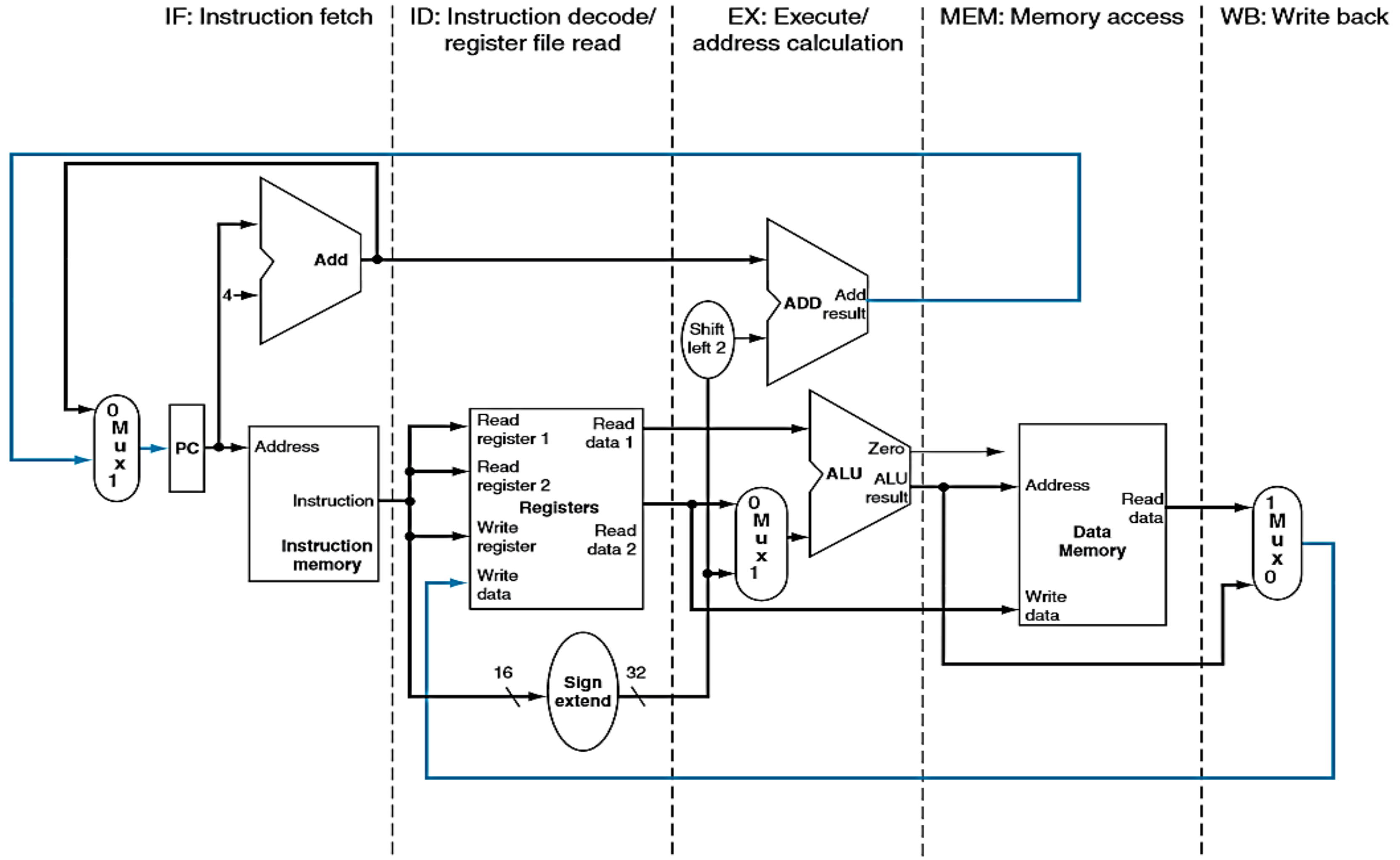
As with the single-cycle and multi-cycle implementations, we will start by looking at the datapath for pipelining.

We already know that pipelining involves breaking up instructions into five stages:

- IF – Instruction Fetch
- ID – Instruction Decode
- EX – Execution
- MEM – Memory Access
- WB – Write Back

We will start by taking a look at the single-cycle datapath, divided into stages.

PIPELINED DATAPATH



PIPELINED DATAPATH

As we can see, each of the steps maps nicely in order onto the single-cycle datapath.

Instruction fields and data generally move from left-to-right as they progress through each stage.

The two exceptions are:

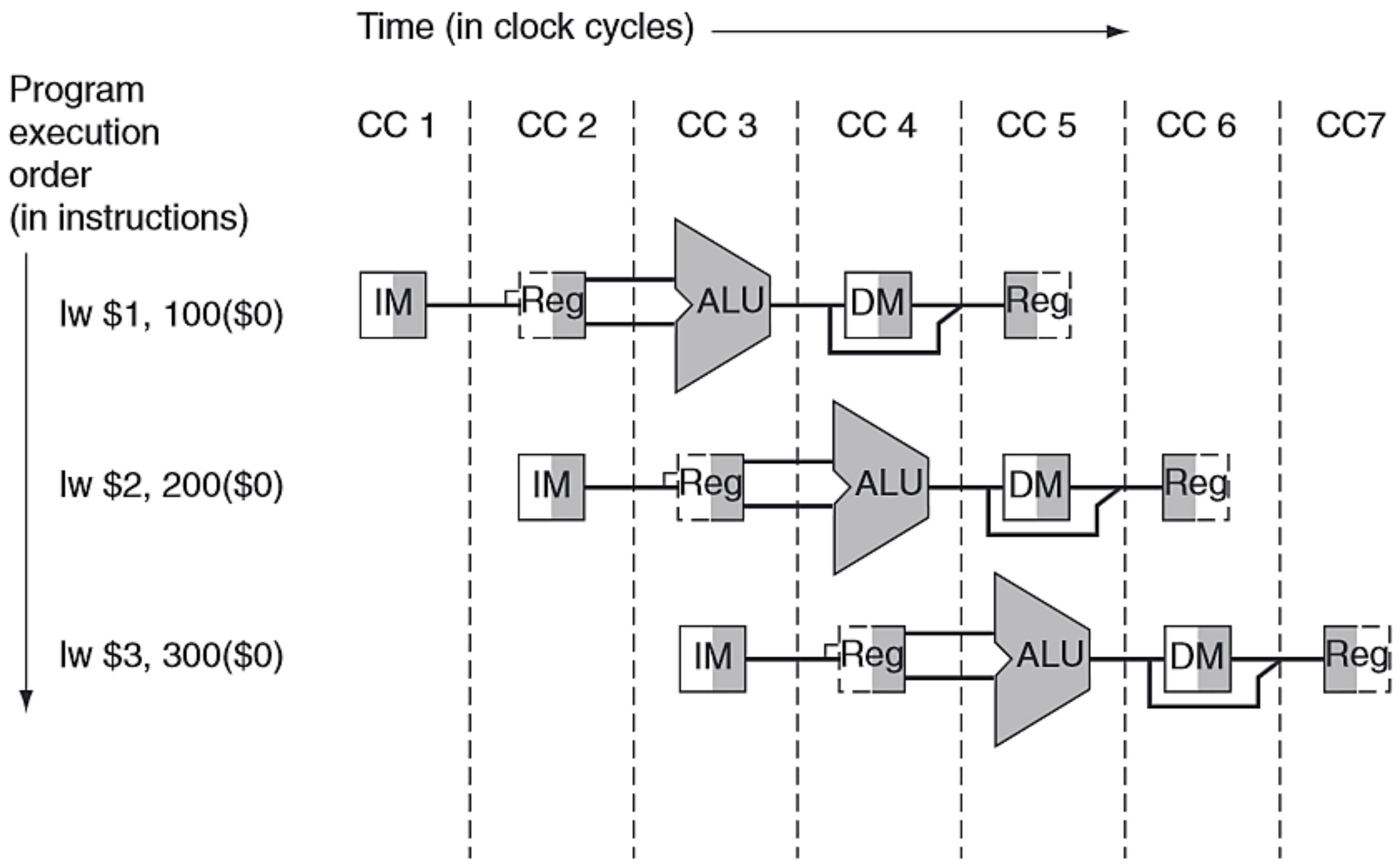
- The WB stage places the result back into the register file in the middle of the datapath → leads to ***data hazards***.
- The selection of the next value of the PC – either the incremented PC or the branch address → leads to ***control hazards***.

PIPELINED DATAPATH

One way to visualize pipelining is to consider the execution of each instruction independently, as if it has the datapath all to itself.

We can place these datapaths on a timeline to see their relationship.

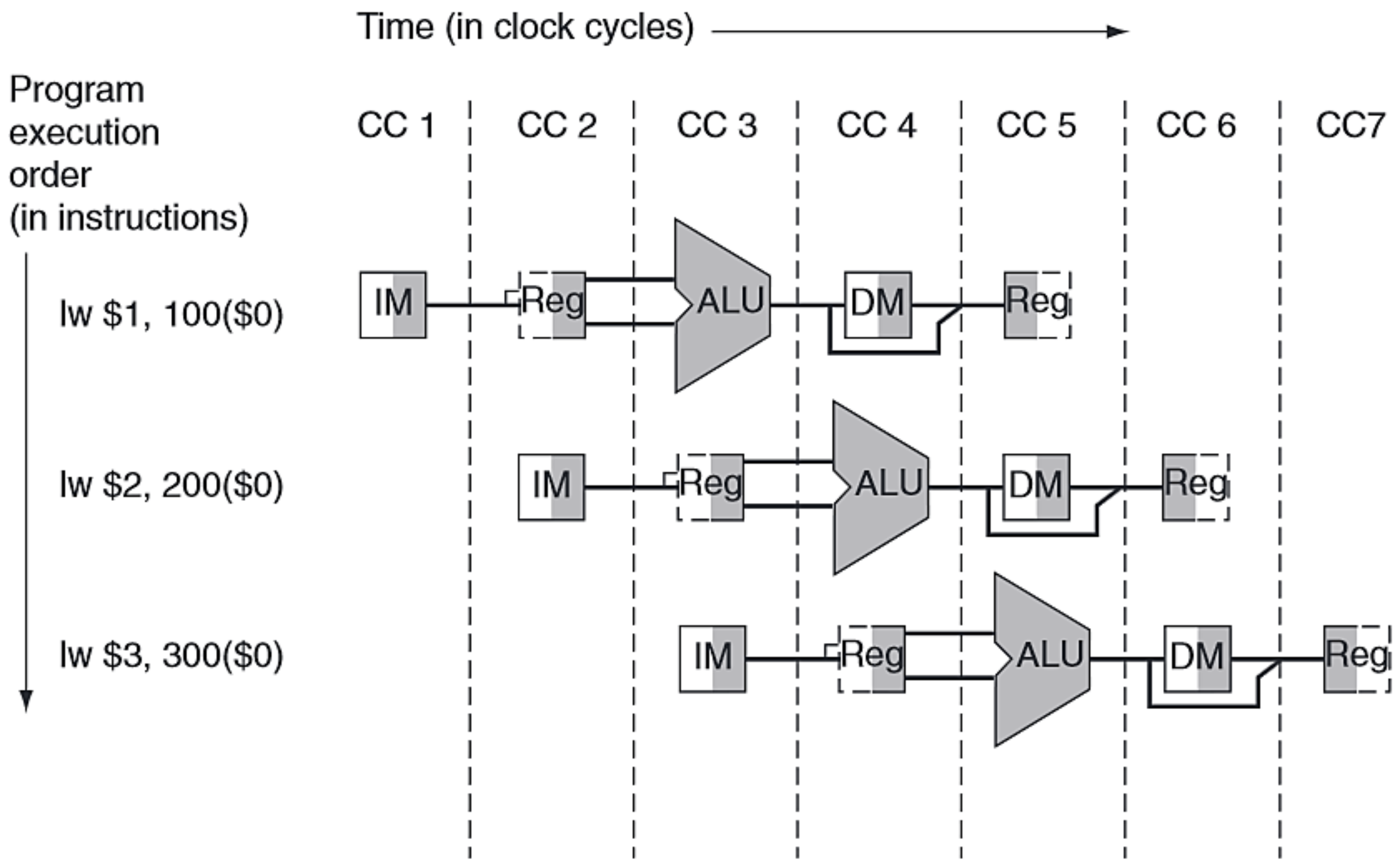
The stages are represented by the datapath element being used, shaded according to use.



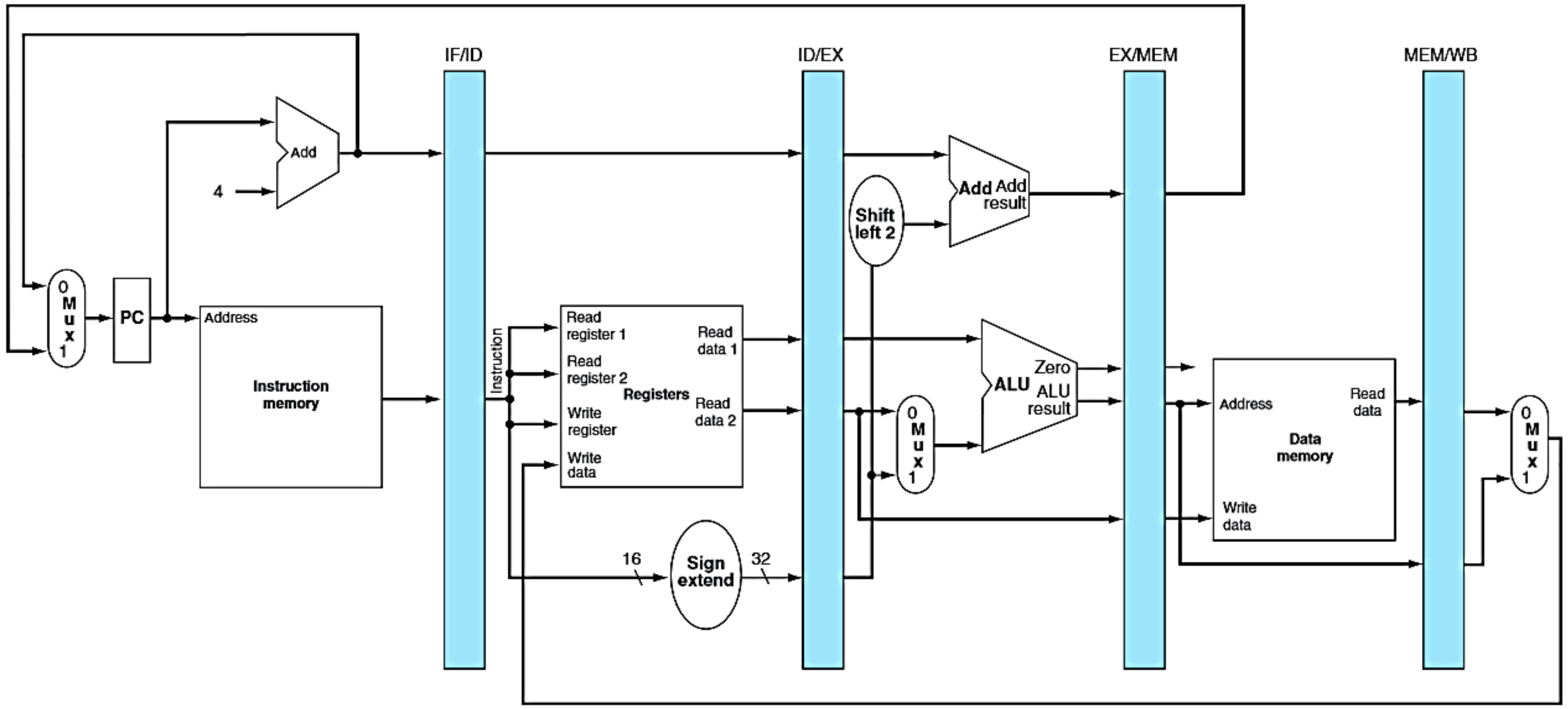
PIPELINED DATAPATH

In reality, these instructions are not executing in their own datapaths, they share a datapath.

The first instruction uses instruction memory in its IF stage in cycle 1. Then, in cycle 2, the second instruction uses instruction memory for its own IF stage. For this to work, we need to add registers to store data between cycles.



PIPELINED DATAPATH



PIPELINED DATAPATH

The previous slide shows the addition of **pipeline registers** (in blue) which are used to hold data between cycles.

Following our laundry analogy, these might be like **baskets** between the washer, dryer, etc that hold a clothing load between steps.

During each cycle, an instruction advances from one pipeline register to the next pipeline register. Note that the registers are labeled by the stages that they separate.

Pipeline registers are **as wide as necessary to hold all of the data passed into them**. For instance, *IF/ID is 64 bits wide because it must hold a **32-bit instruction** and a **32-bit PC+4** result.*

PIPELINED DATAPATH FOR LOAD WORD

Let's walk through the datapath using the load word instruction as an example. Load word is a good instruction to start with because it is active in every stage of the pipelined datapath.

Note that in the following datapaths, the **right** half of registers or memory are shaded when they are being ***read***. The **left** half is shaded when they are being ***written***.

```
lw $rt, immed($rs)
```

31-26	**25-21**	**20-16**	**15-0**
opcode	rs	rt	immed

The load word instruction adds `immed` to the contents of `$rs` to obtain the address in memory whose contents are written to `$rt`.

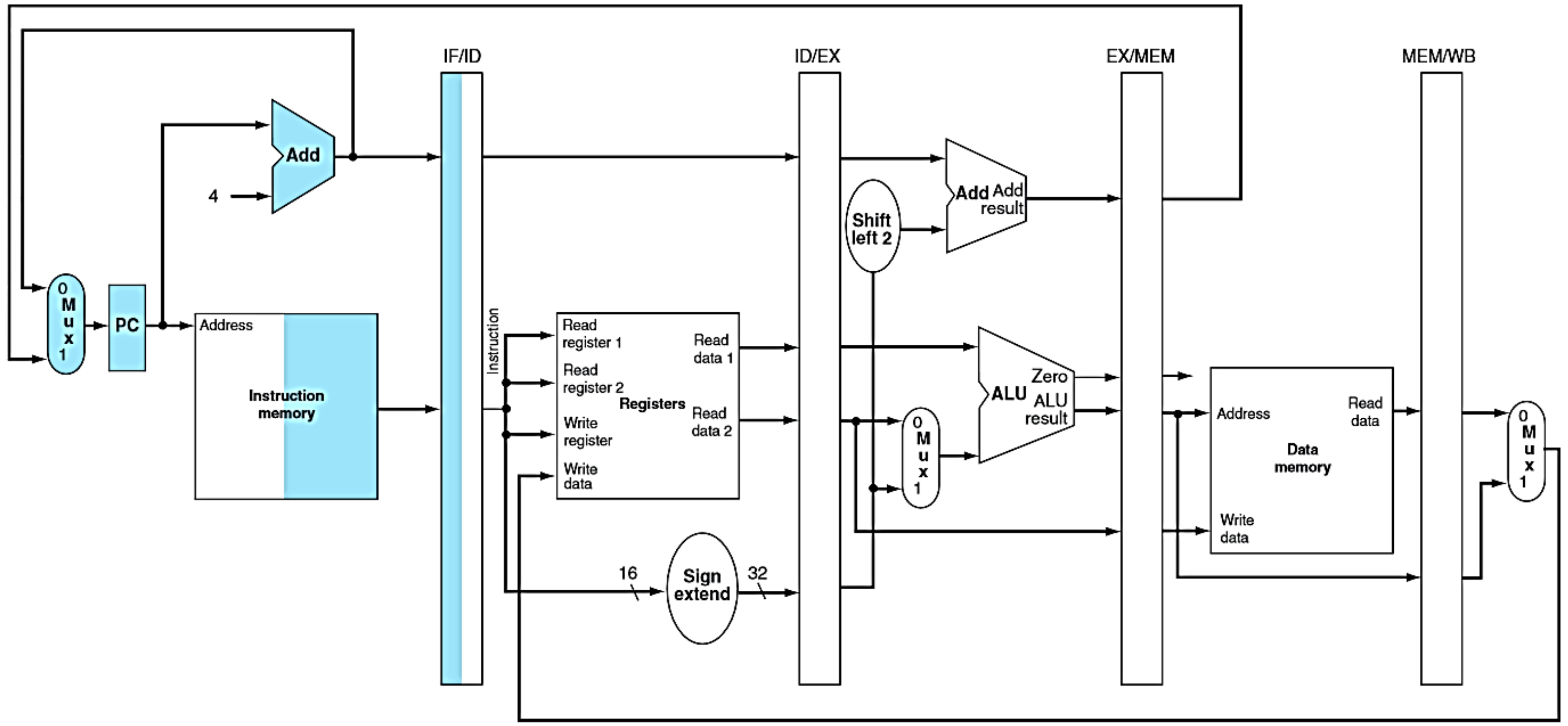
PIPELINED DATAPATH FOR LOAD WORD

Instruction Fetch (IF)

- The instruction is read from memory using the contents of PC and **placed in the IF/ID register.**
- The PC address is **incremented by 4** and **written back to the PC register,** as well as **placed in the IF/ID register** in case the instruction needs it later.

Note: the datapath does not know that we are performing a load word at this point so it forwards the PC+4 value just in case.

PIPELINED DATAPATH FOR LOAD WORD: IF

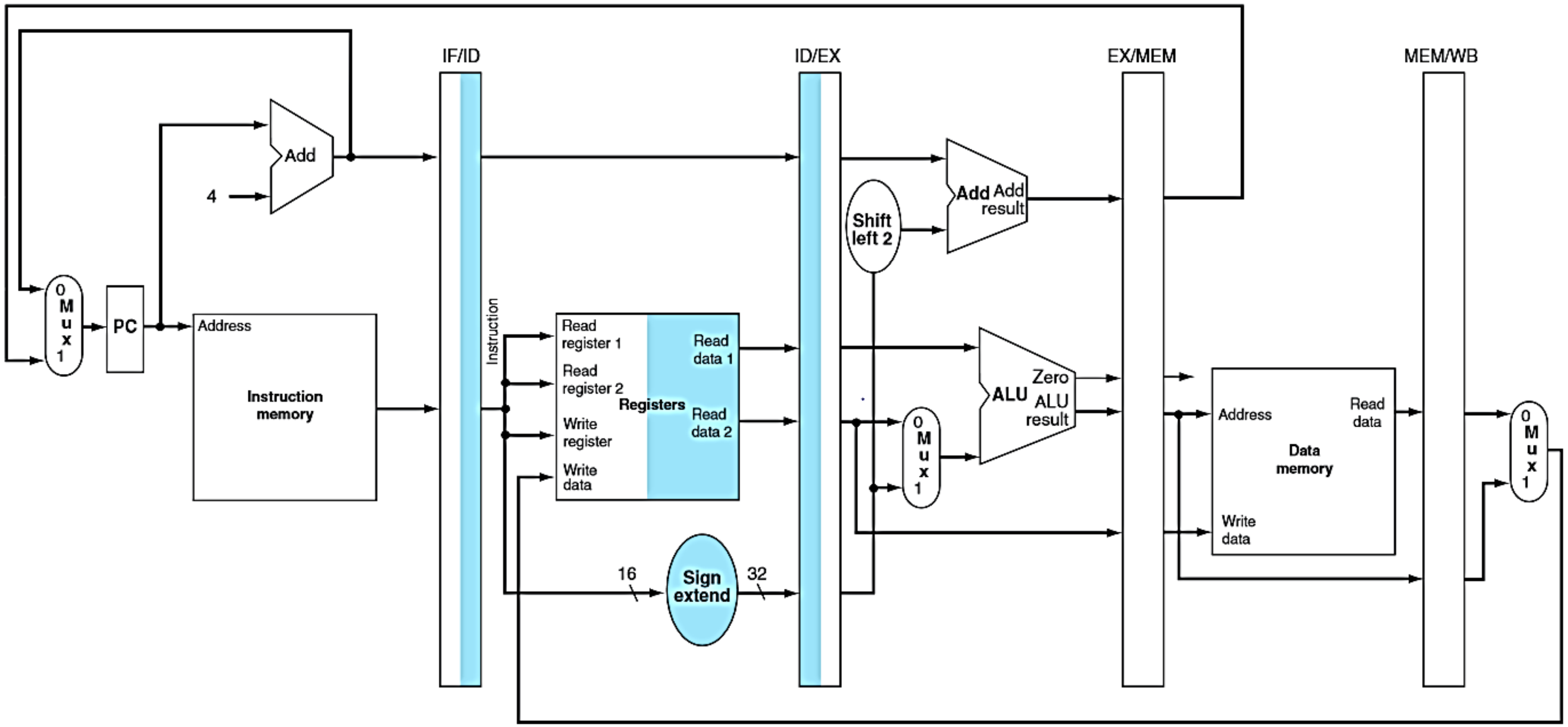


PIPELINED DATAPATH FOR LOAD WORD

Instruction Decode and Register File Read (ID):

- The registers **$rs and $rt are read from the register file** and **stored in the ID/EX pipeline register.** Remember, we don't know what the instruction is yet.
- The 16-bit immediate field **is sign-extended to 32-bits** and stored in the ID/EX pipeline register.
- The **PC+4 value is copied from the IF/ID register into the ID/EX register** in case the instruction needs it later.

PIPELINED DATAPATH FOR LOAD WORD: ID

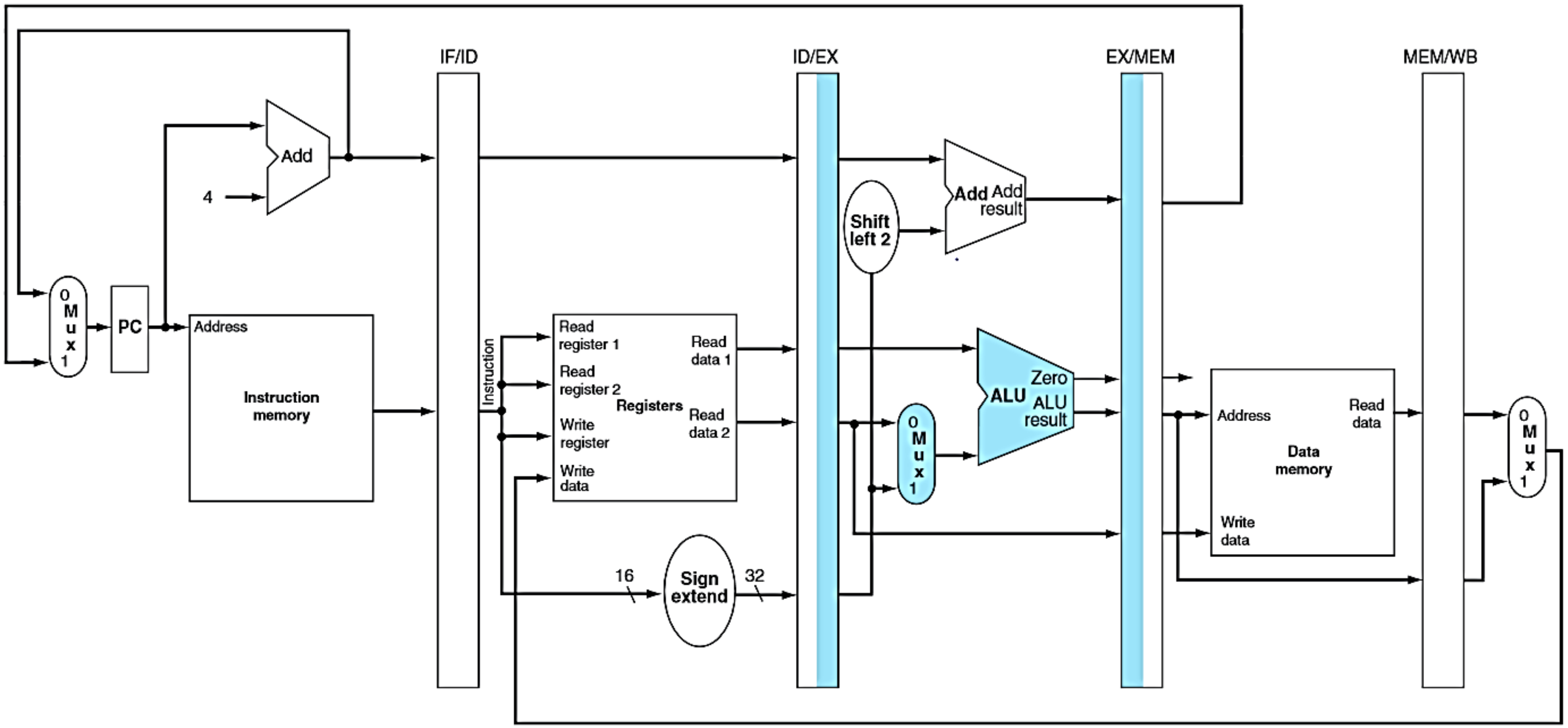


PIPELINED DATAPATH FOR LOAD WORD

Execute or Address Calculation (EX):

- From the ID/EX pipeline register, take the contents of **$rs and the sign-extended immediate field as inputs to the ALU**, which performs an add operation. The sum is **placed in the EX/MEM pipeline register.**

PIPELINED DATAPATH FOR LOAD WORD: EX

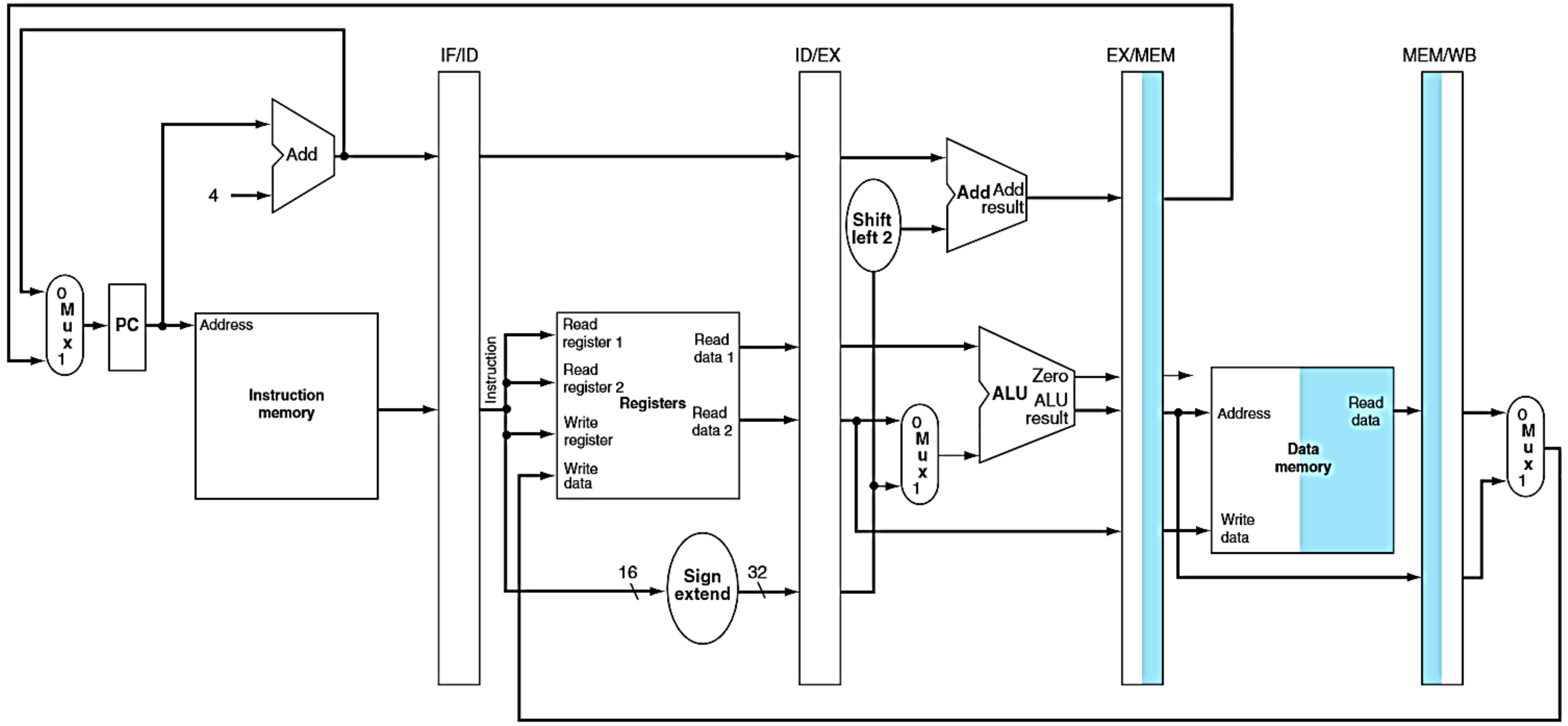


PIPELINED DATAPATH FOR LOAD WORD

Memory Access (MEM):

- Take the **address stored in the EX/MEM pipeline** register and use it to access data memory. The **data read from memory is stored in the MEM/WB** pipeline register.

PIPELINED DATAPATH FOR LOAD WORD: MEM

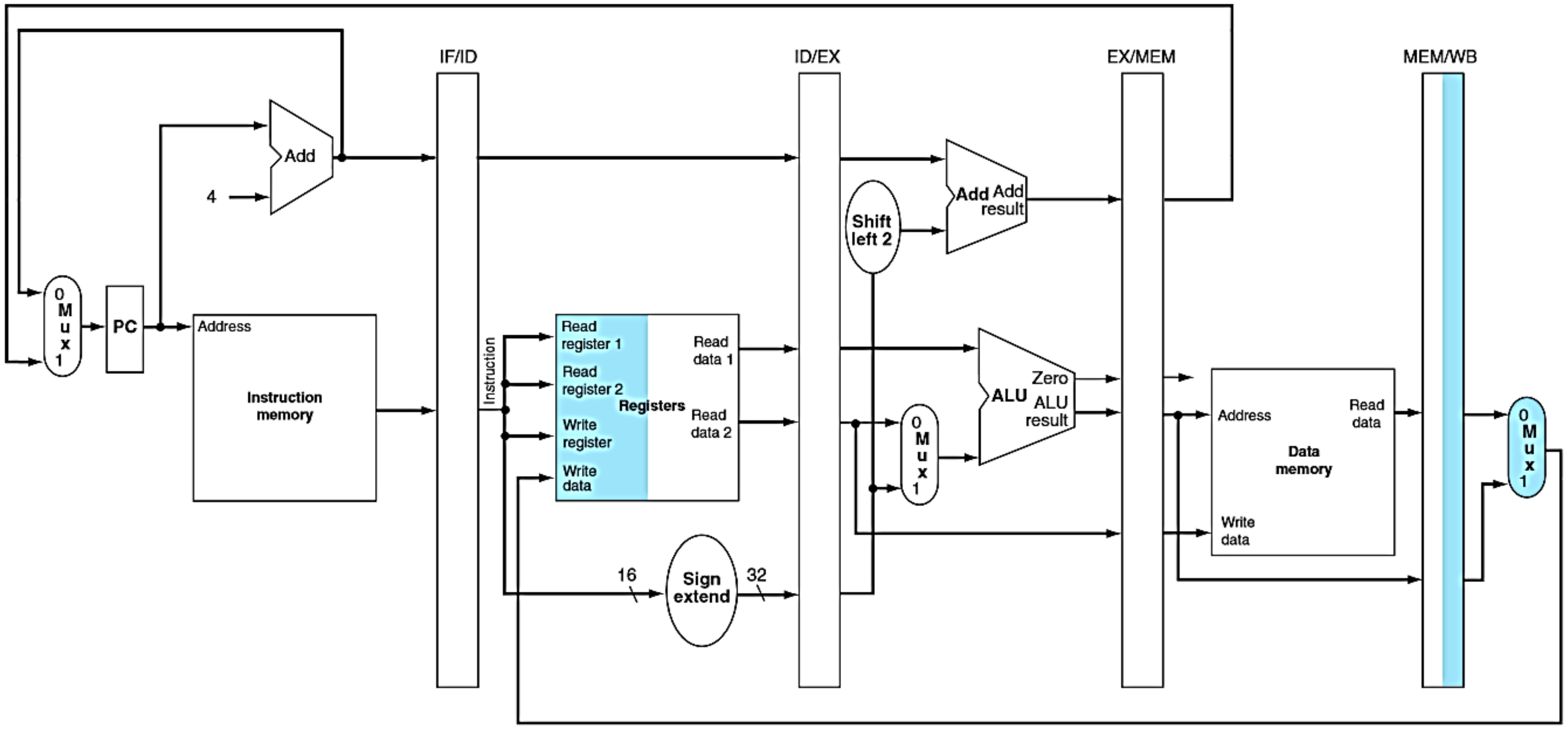


PIPELINED DATAPATH FOR LOAD WORD

Write Back (WB):

- Read the data from the MEM/WB register and **write it back to the register file** in the middle of the datapath.

PIPELINED DATAPATH FOR LOAD WORD: WB



PIPELINED DATAPATH FOR LOAD WORD

It’s important to note that any information we need will have to be **passed from pipeline register to pipeline register** while instruction executes.

Because the instructions share the elements, we cannot assume anything from a previous cycle is still there. We must carry the data with us as we move along the data path.

So by now you should be wondering about the last step. **How does the WB stage know which register to write to?** *The IF/ID pipeline register should no longer contain the necessary instruction field* – it’s already been overwritten by three other instructions at this point.

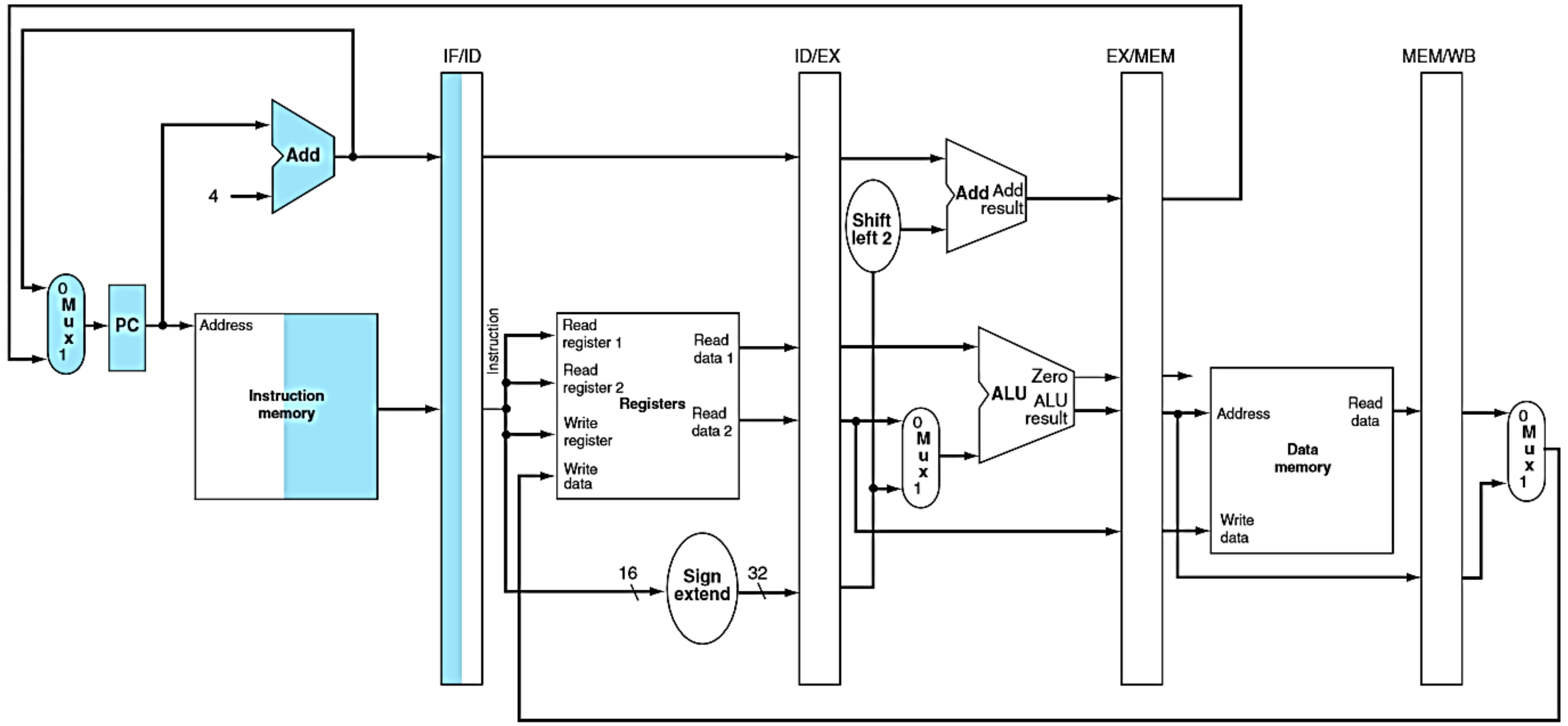
We’ll see the solution soon, but let’s look at store word first.

PIPELINED DATAPATH FOR STORE WORD

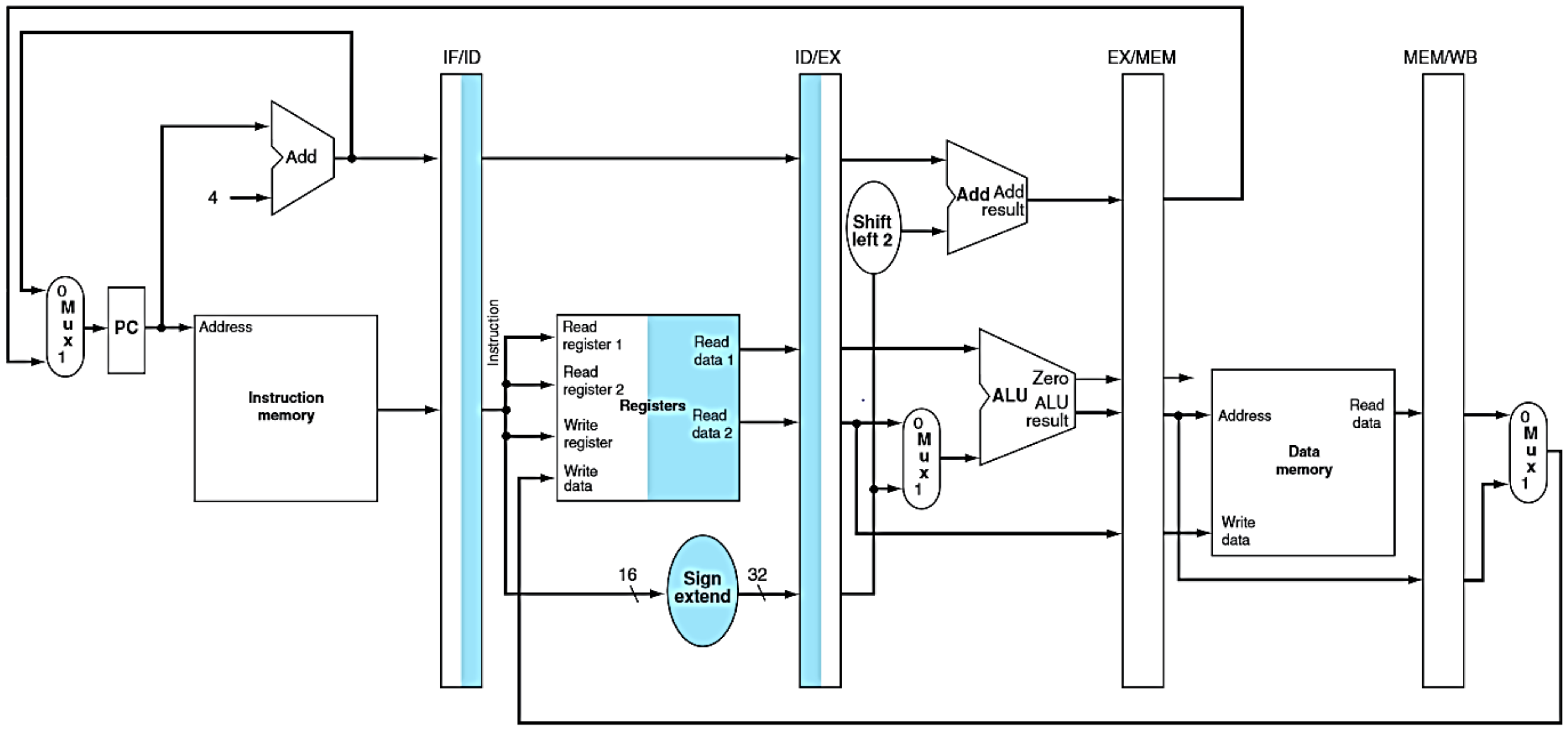
The IF and ID stages are identical for all instructions.

At these stages of execution, we still do not know which instruction we are actually executing so we perform general operations that either apply to all instructions or may speculatively apply to a select few instructions.

PIPELINED DATAPATH FOR STORE WORD: IF



PIPELINED DATAPATH FOR STORE WORD: ID

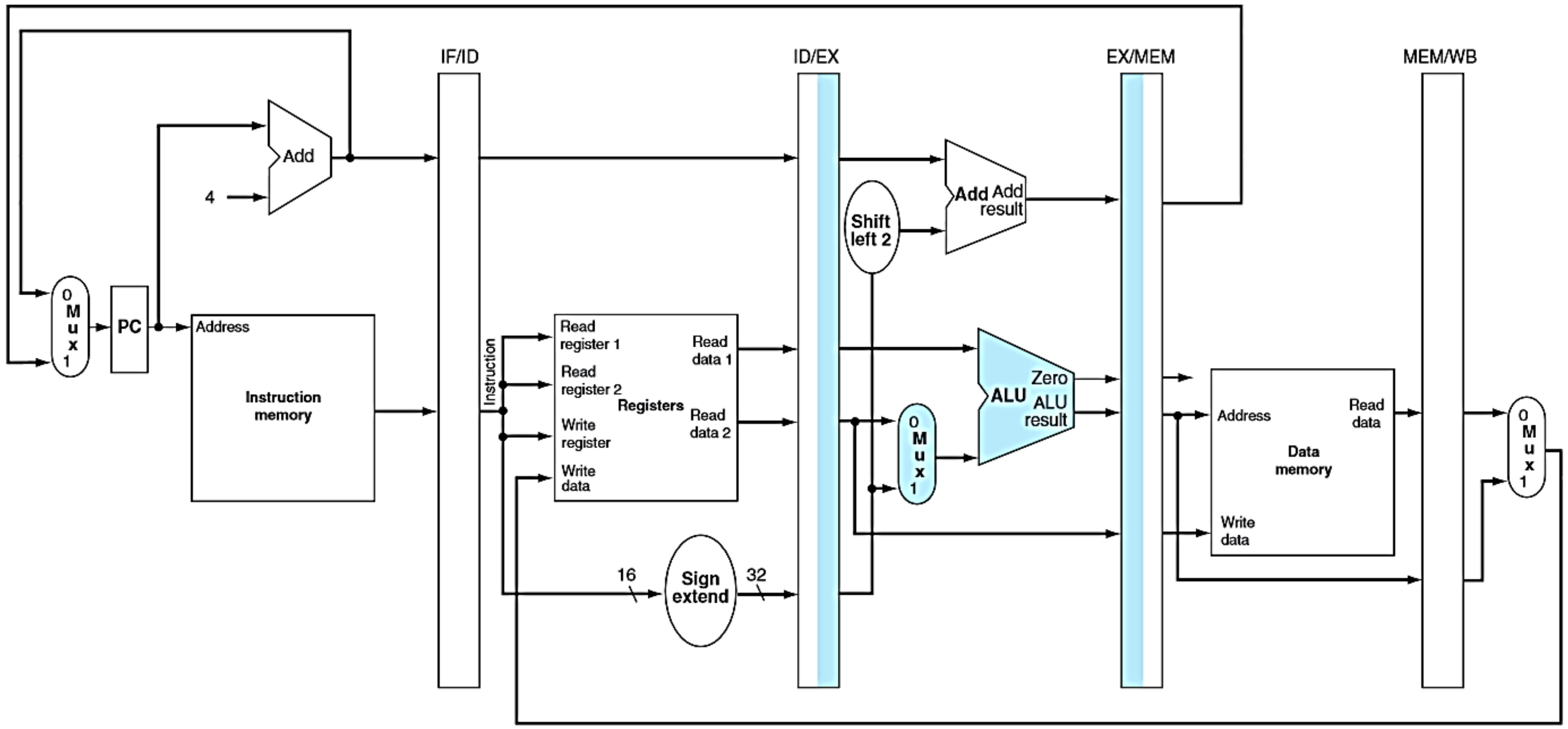


PIPELINED DATAPATH FOR STORE WORD

Execute or Address Calculation (EX):

- From the ID/EX pipeline register, take the contents of $rs and the sign-extended immediate field as inputs to the ALU, which performs an add operation. The sum is placed in the EX/MEM pipeline register.
- The contents of **the $rt register are copied from the ID/EX pipeline register** to the EX/MEM pipeline register.

PIPELINED DATAPATH FOR STORE WORD: EX



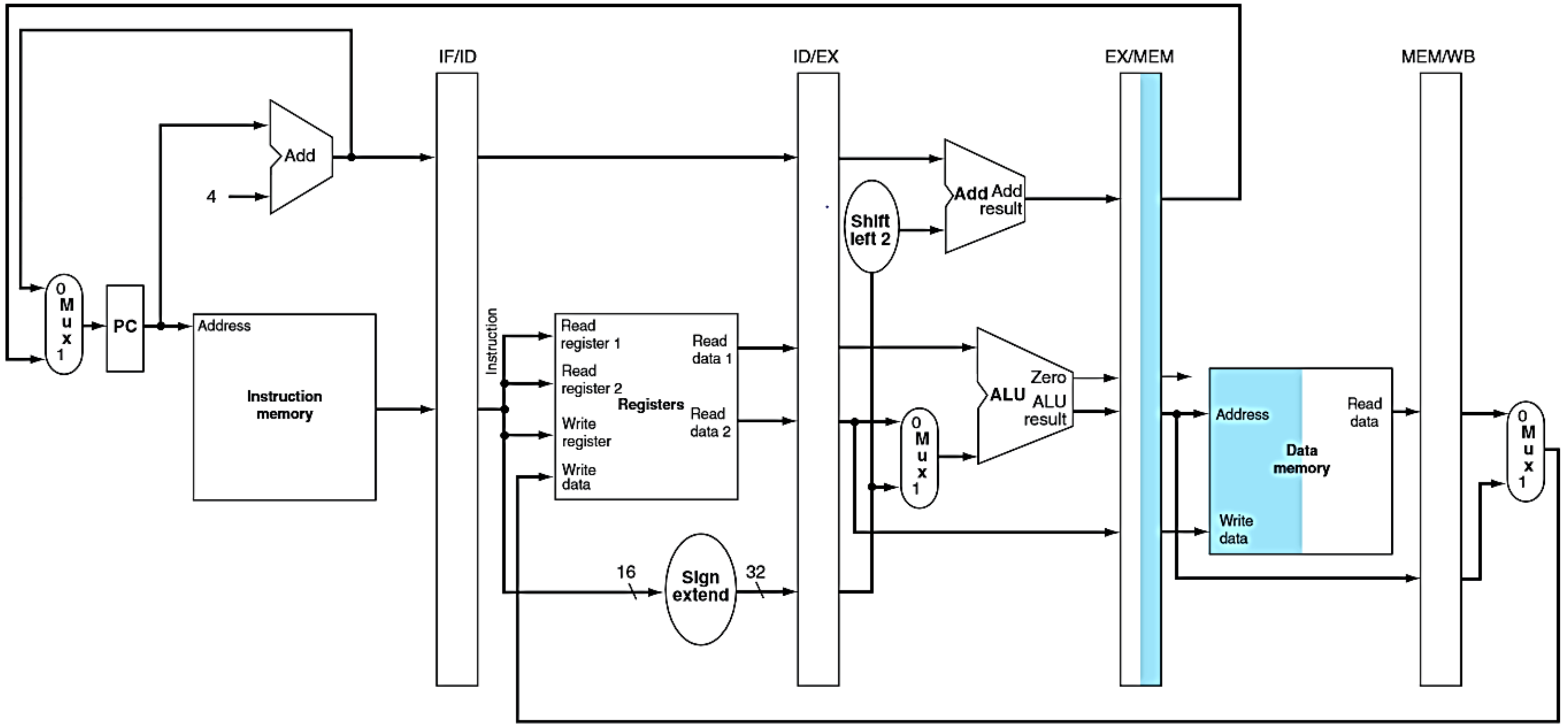
PIPELINED DATAPATH FOR STORE WORD

Memory Access (MEM):

- Take the address stored in the EX/MEM pipeline register and use it to access data memory.
- Take the contents of $rt from the EX/MEM pipeline and write it to data memory at the address specified.

Note that in order to use the $rt field in the MEM stage, we have to carry it with us from the ID stage. This means copying it in every pipeline register until we use it.

PIPELINED DATAPATH FOR STORE WORD: MEM

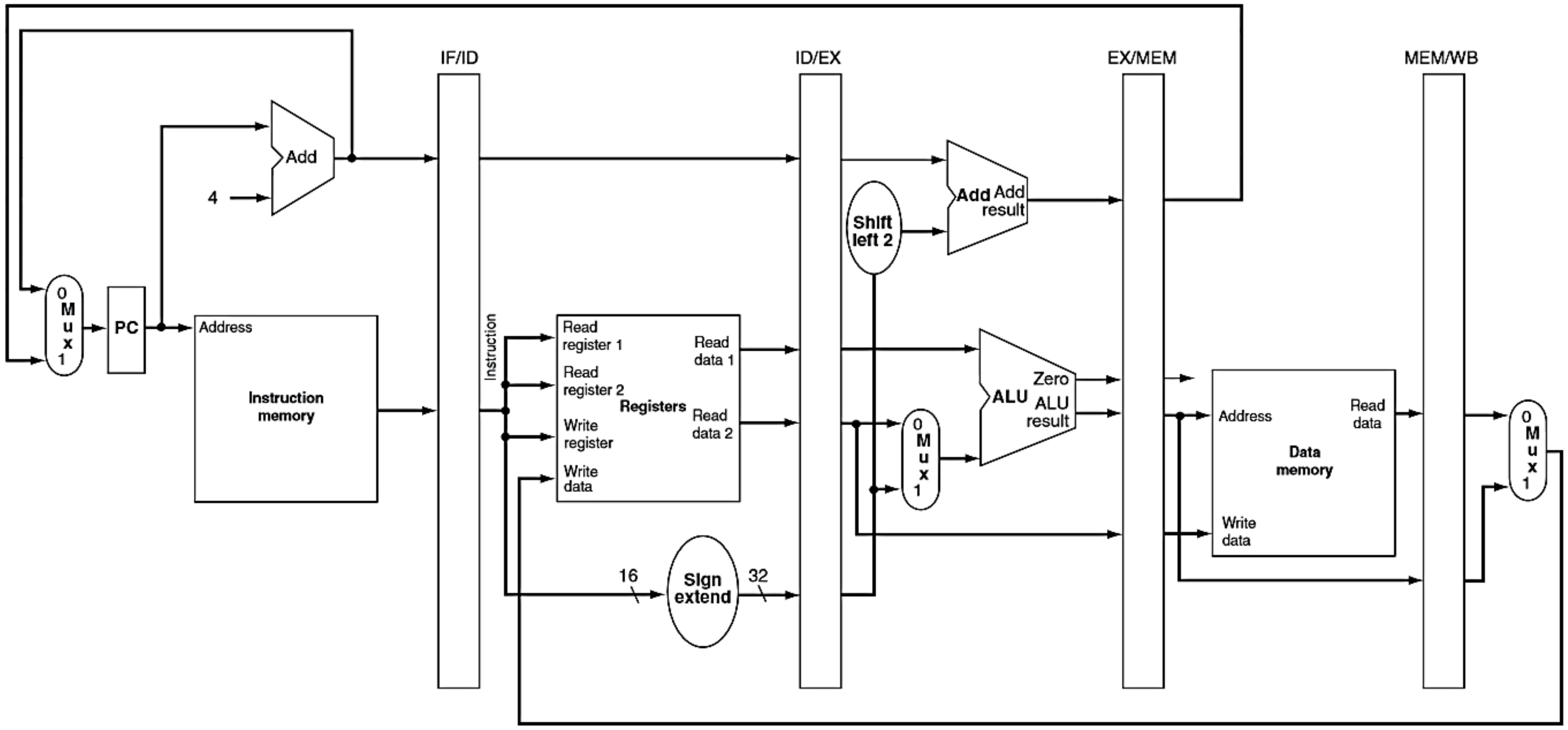


PIPELINED DATAPATH FOR STORE WORD

Write Back (WB):

- We’re already done with the store word instruction so we don’t have to do anything.

PIPELINED DATAPATH FOR STORE WORD: WB



PIPELINED DATAPATH

In both the load and store word datapaths, we can notice another important point: *every datapath element is used in only one pipeline stage*. If this wasn't the case, we'd create a **structural hazard**.

Let's say we used the ALU from the EX stage to both increment PC in the IF stage and perform operations in the EX stage. This is fine for a single instruction, but what happens if we're executing multiple instructions and one is currently in the IF stage and another is in the EX stage? Which one gets to use the ALU?

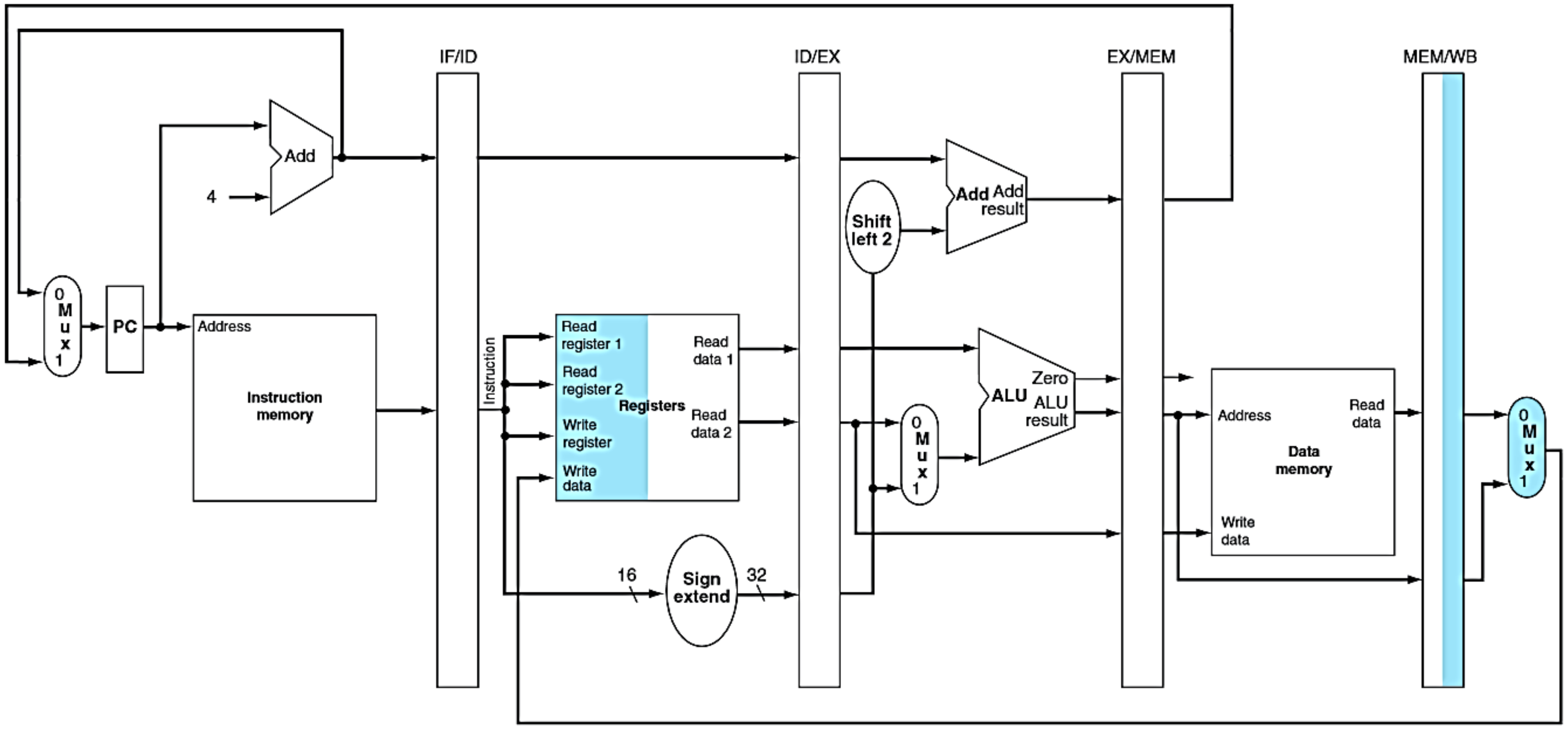
Clearly, we'd have an issue. So, we add **redundancy** to our datapath in order to gain the speed improvements from pipelining.

PIPELINED DATAPATH FOR LOAD WORD

Ok, so let’s look back at load word’s WB stage.

Remember that we have a problem? We don’t know to which register we need to write back. What’s the solution?

PIPELINED DATAPATH FOR LOAD WORD: WB



PIPELINED DATAPATH FOR LOAD WORD

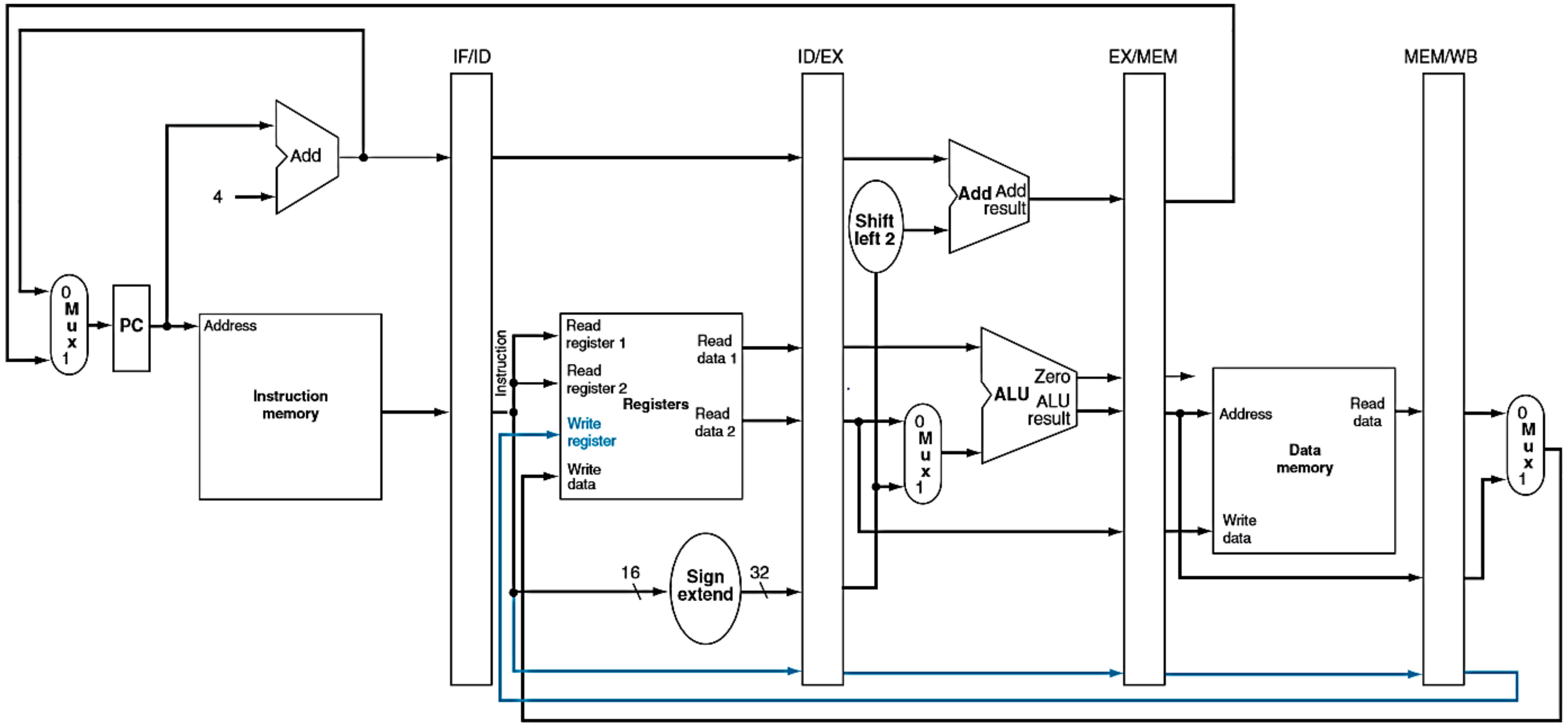
Ok, so let's look back at load word's WB stage.

Remember that we have a problem? We don't know to which register we need to write back. What's the solution?

That's right! ***Carry the information through each stage using the pipeline registers****. To do this, we'll modify the datapath a little bit.*

Now, we'll pass the write register number from the MEM/WB pipeline register along with the data. This register number is initially discovered in the ID stage and must be passed through the pipeline registers until we need it in the WB stage.

PIPELINED DATAPATH FOR LOAD WORD: WB



PIPELINED DATAPATH

Consider the 5 instruction sequence below.

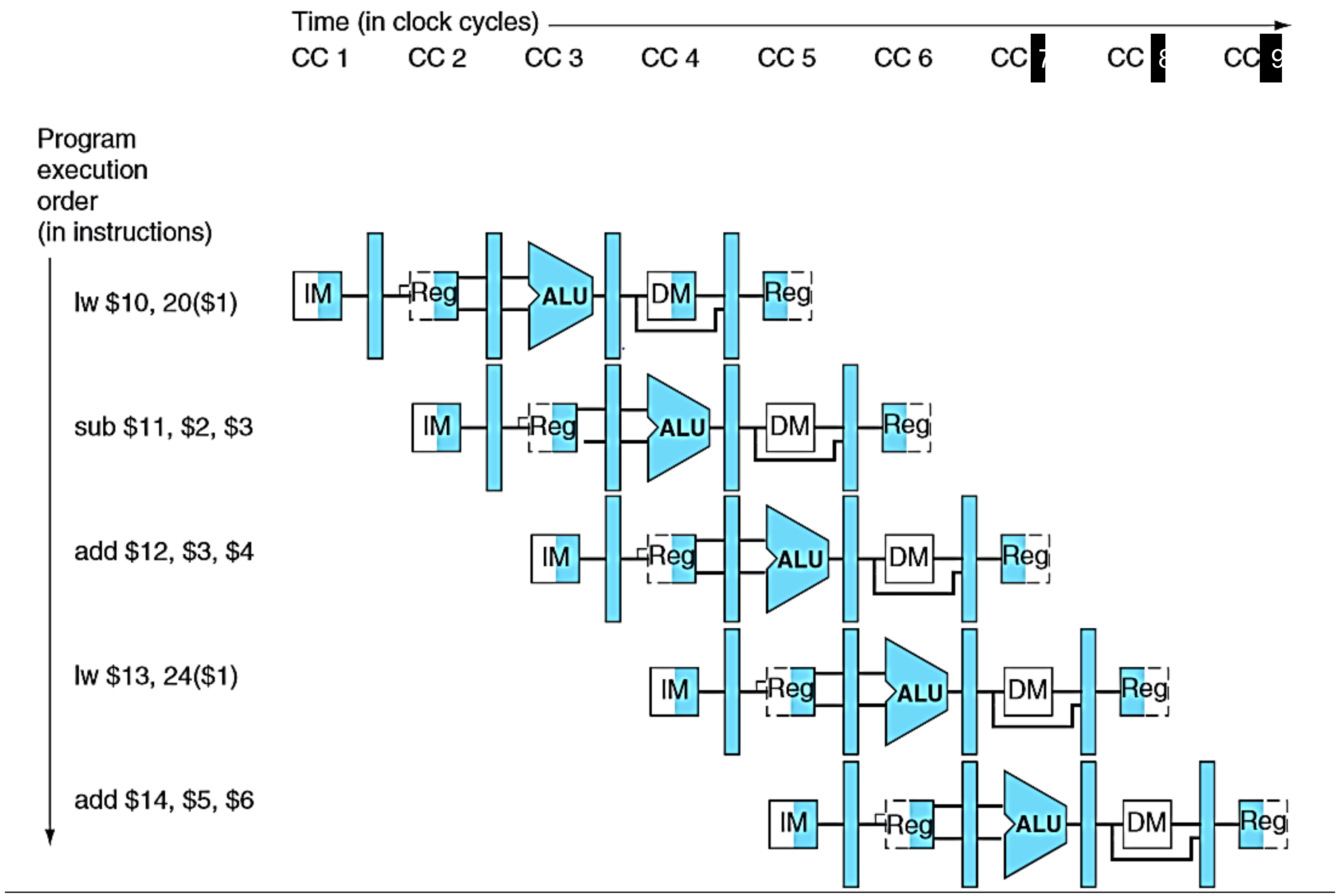
```
lw   $10, 20($1)
sub  $11, $2, $3
add  $12, $3, $4
lw   $13, 24($1)
add  $14, $5, $6
```

First, take note of the fact that we have no data hazards since no instruction depends on data from a previous instruction. We also have no control hazards since we are not branching or jumping. So, we can safely pipeline without stalls.

PIPELINED DATAPATH

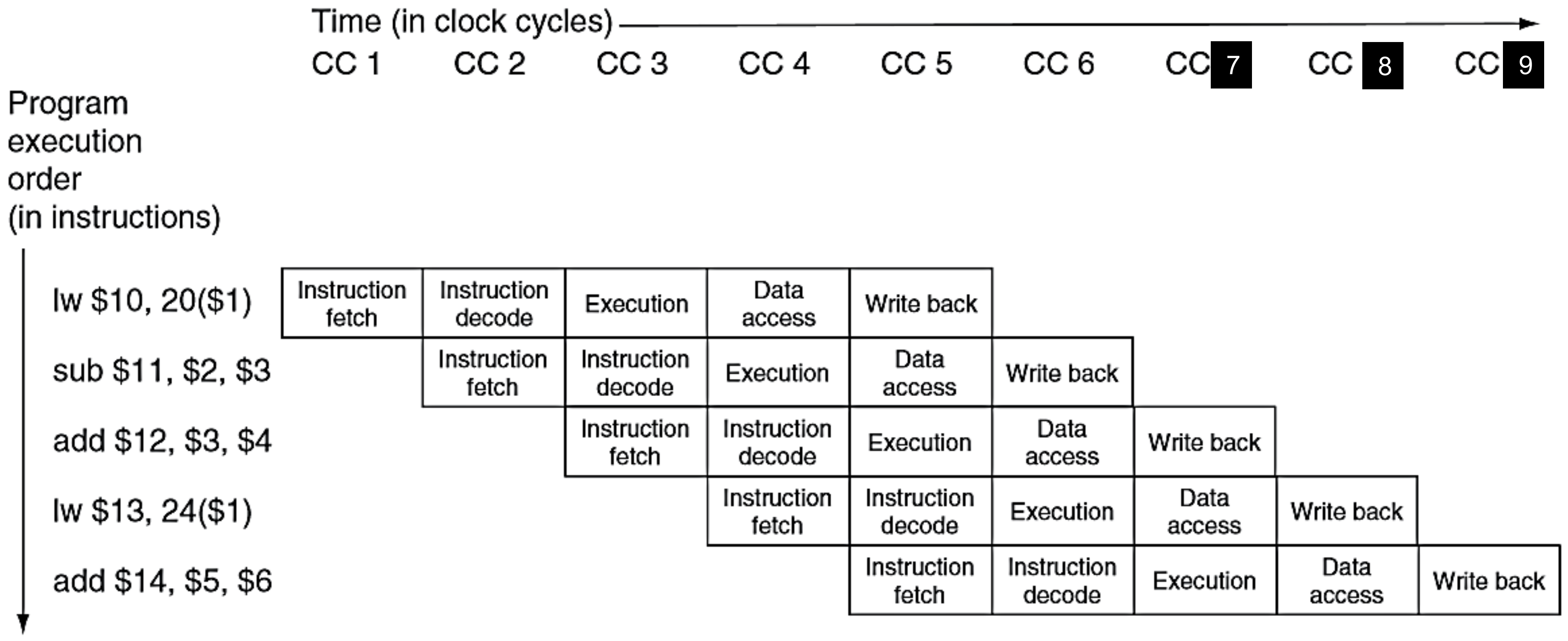
We can start by diagramming the individual datapaths used by every instruction.

This allows us to see which stage each instruction is executing in a given clock cycle.



PIPELINED DATAPATH

We can also chart the stages themselves rather than the elements being used in the cycle.



Time (in clock cycles)

Program execution order (in instructions)

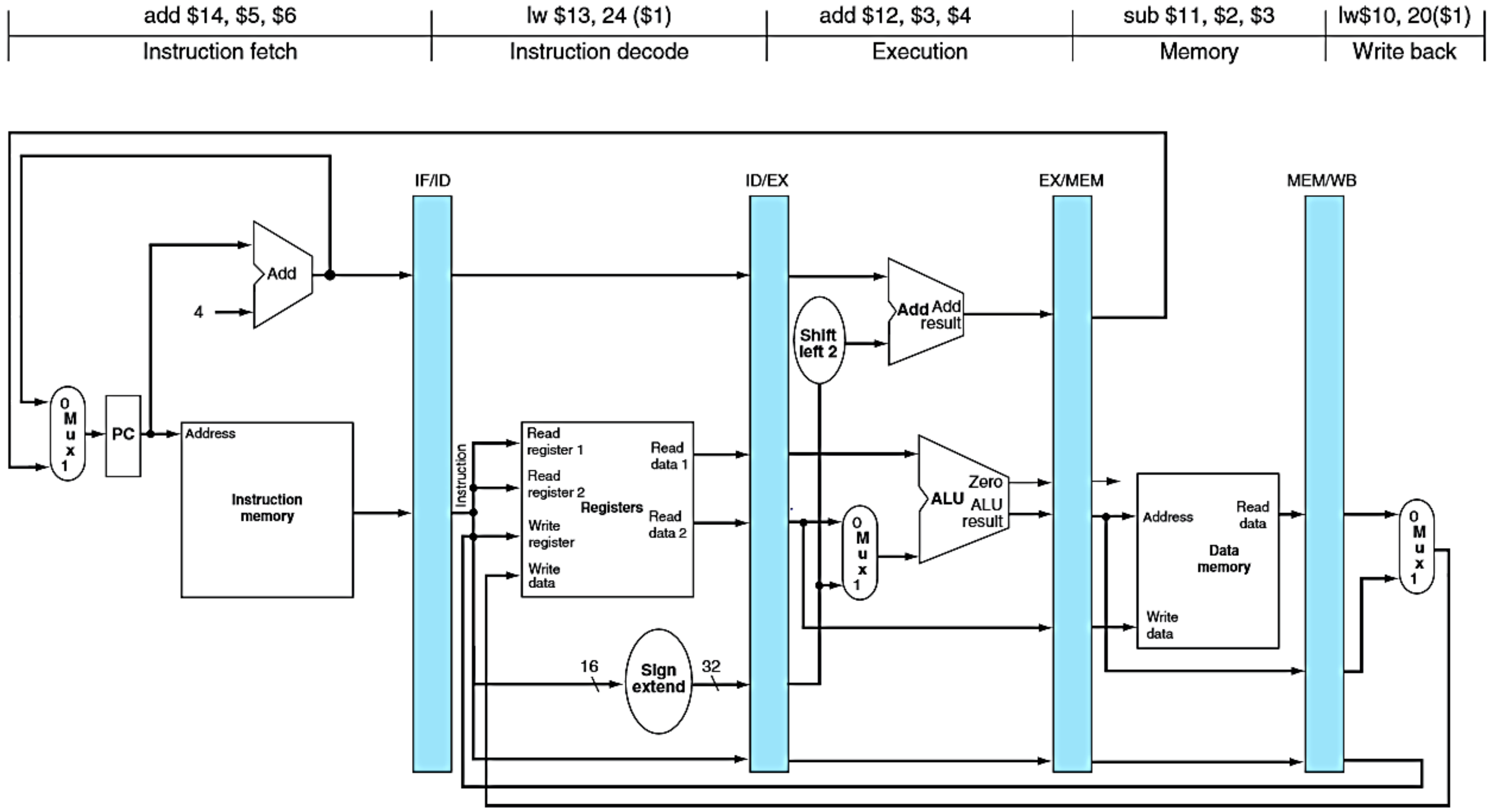
	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
lw $10, 20($1)	Instruction fetch	Instruction decode	Execution	Data access	Write back				
sub $11, $2, $3		Instruction fetch	Instruction decode	Execution	Data access	Write back			
add $12, $3, $4			Instruction fetch	Instruction decode	Execution	Data access	Write back		
lw $13, 24($1)				Instruction fetch	Instruction decode	Execution	Data access	Write back	
add $14, $5, $6					Instruction fetch	Instruction decode	Execution	Data access	Write back

PIPELINED DATAPATH

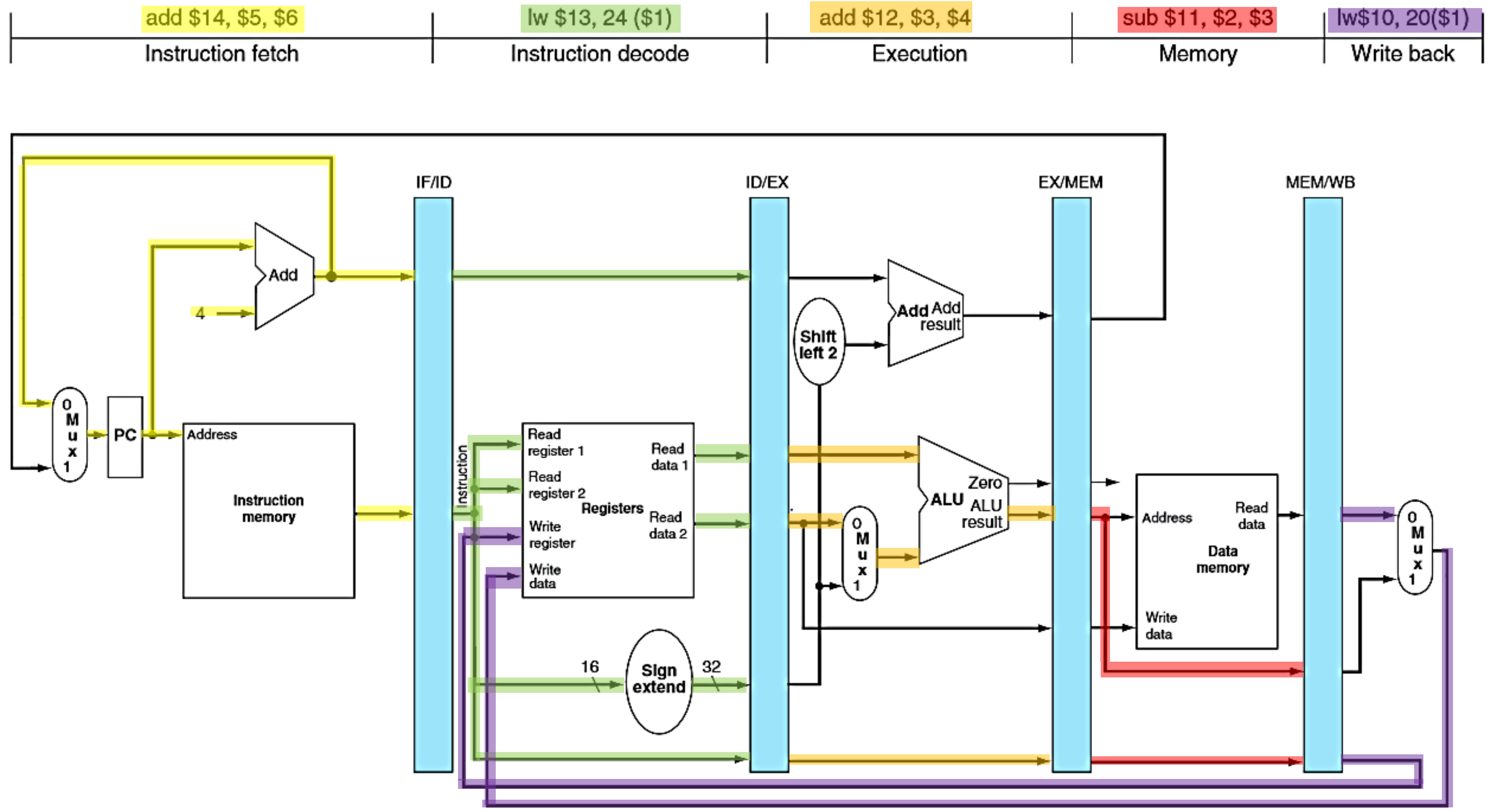
On the next slide, we show a single datapath as it is being used by the 5 instructions at once.

The cycle depicted is **cycle 5**. In this cycle, the first load word is executing its WB stage while the last add instruction is executing its IF stage.

PIPELINED DATAPATH



PIPELINED DATAPATH



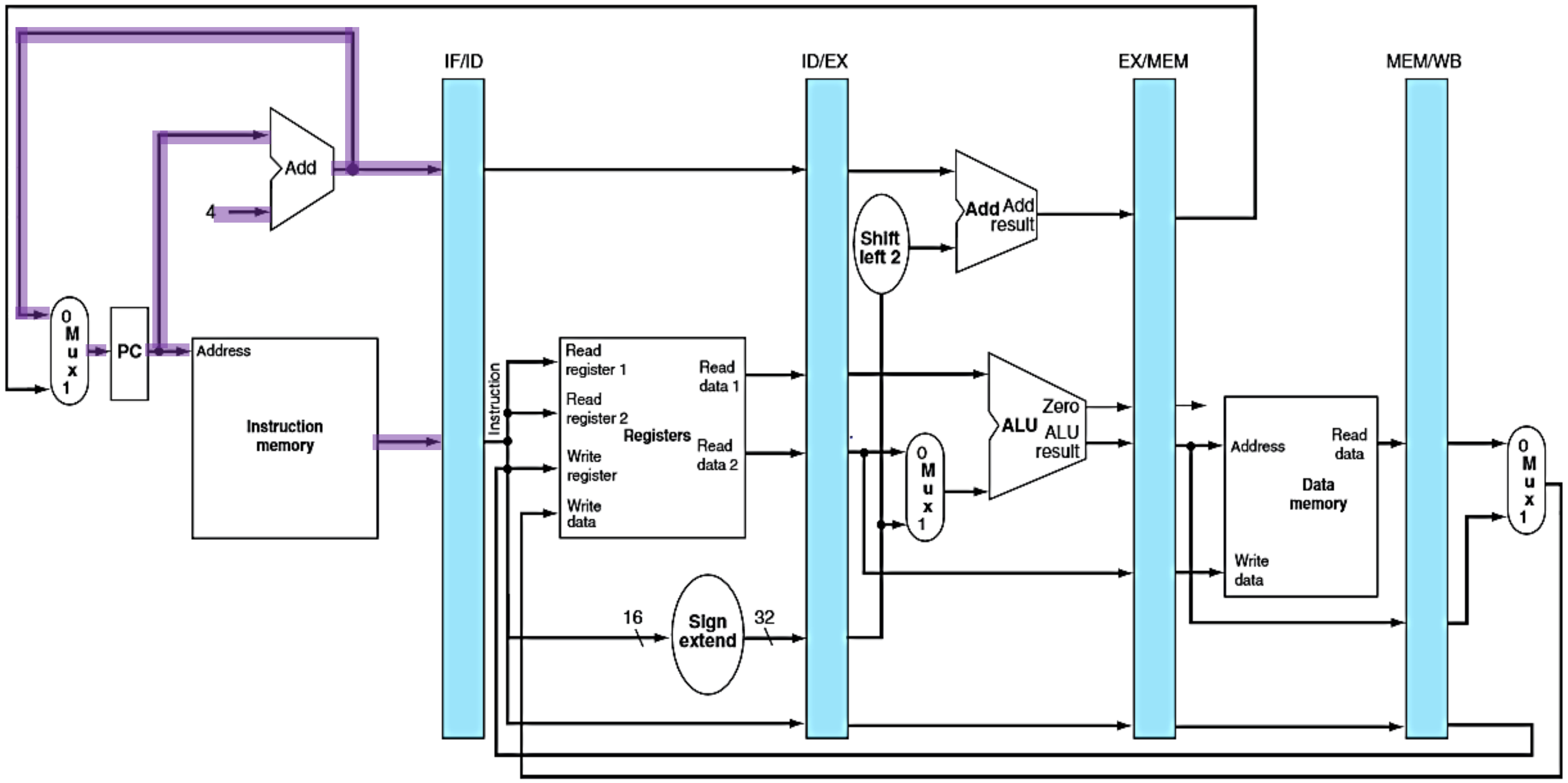
PIPELINED DATAPATH

In the following slides, we walk through all 9 cycles required to fully complete the 5 instructions in our sample sequence.

We will start with clock cycle 1 and highlight the relevant datapath lines according to the instruction being executed in each particular stage.

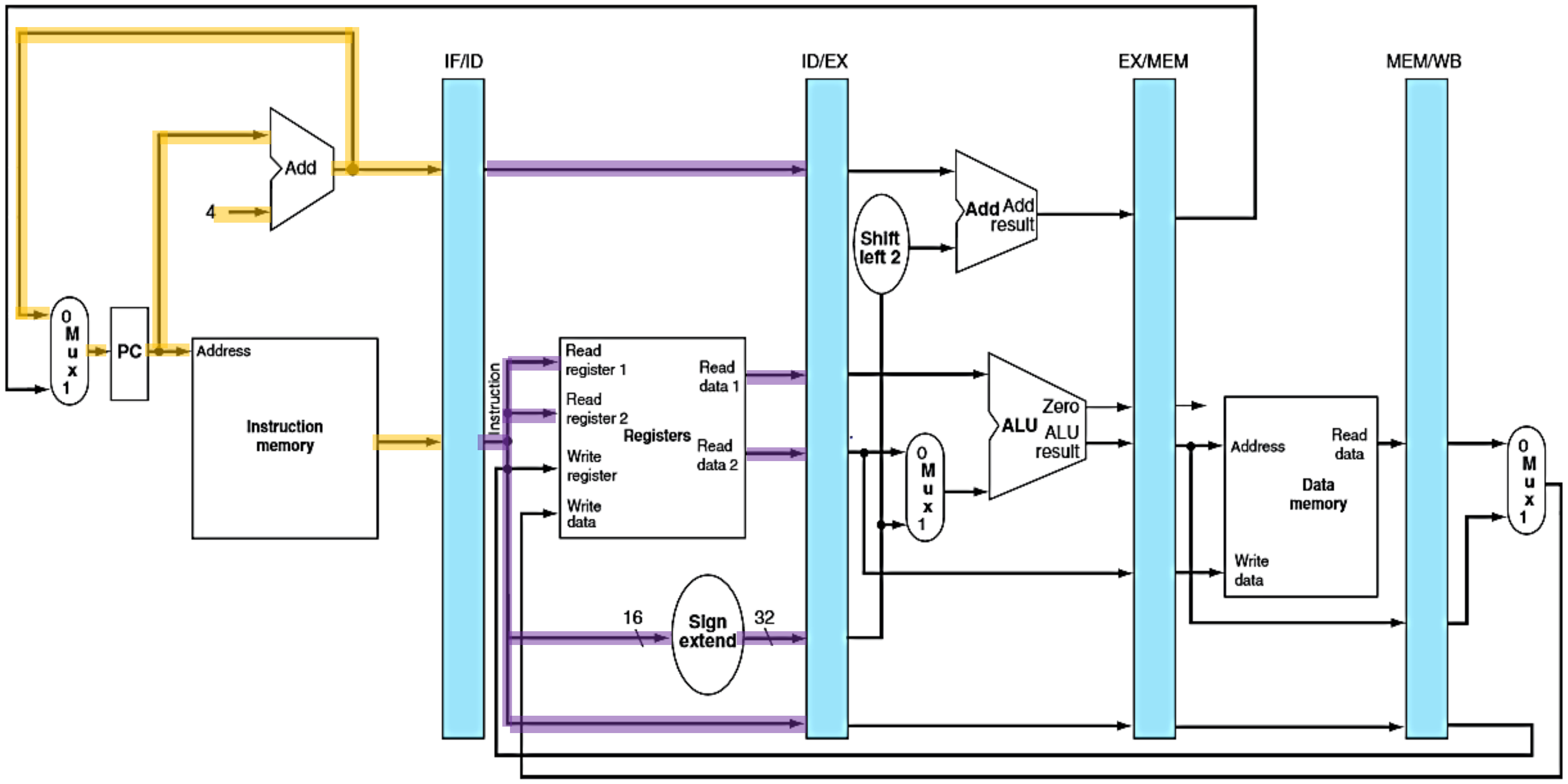
PIPELINED DATAPATH

```
lw $10, 20($1)
```



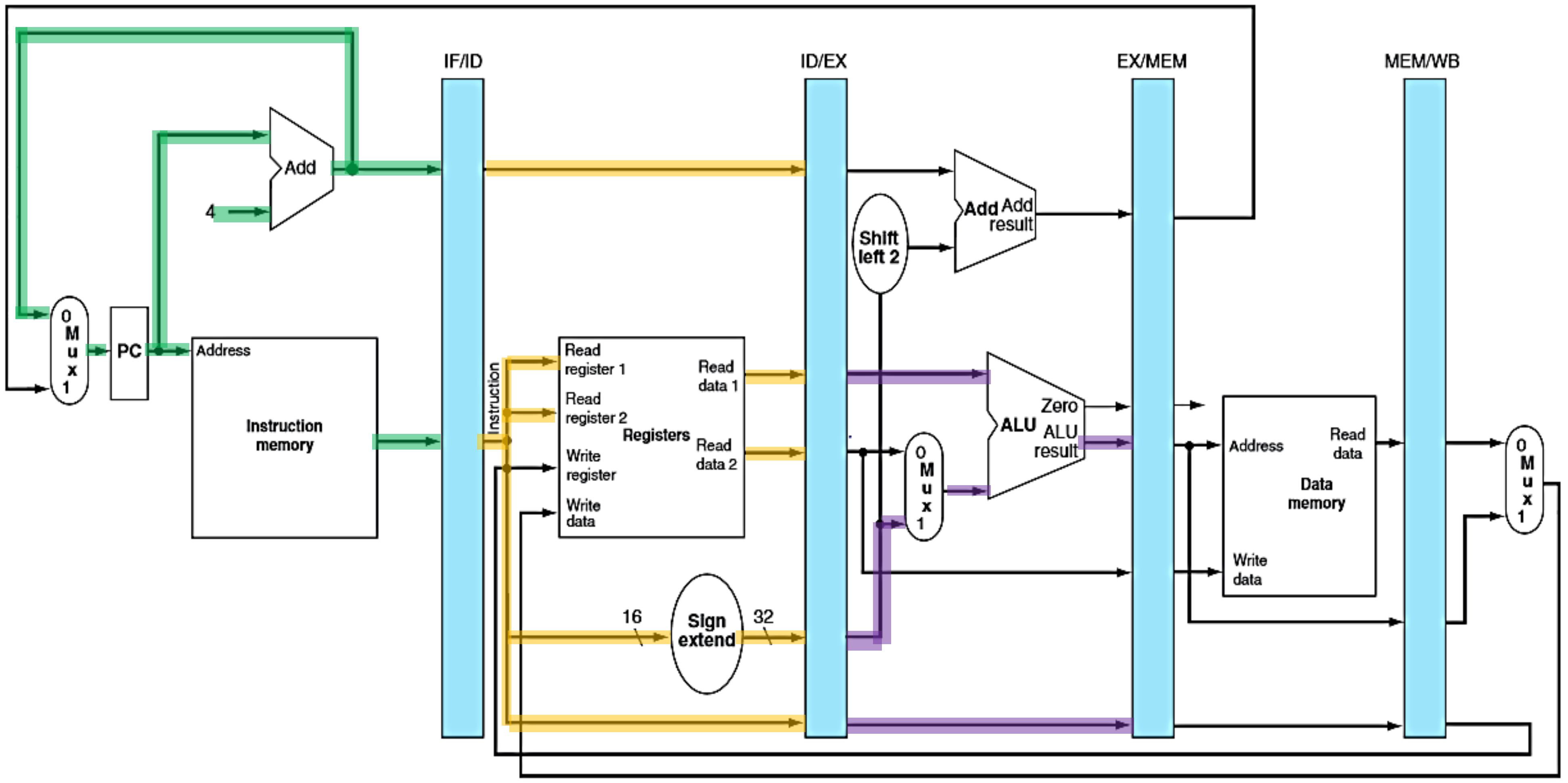
PIPELINED DATAPATH

sub $11,$2,$3   lw $10, 20($1)



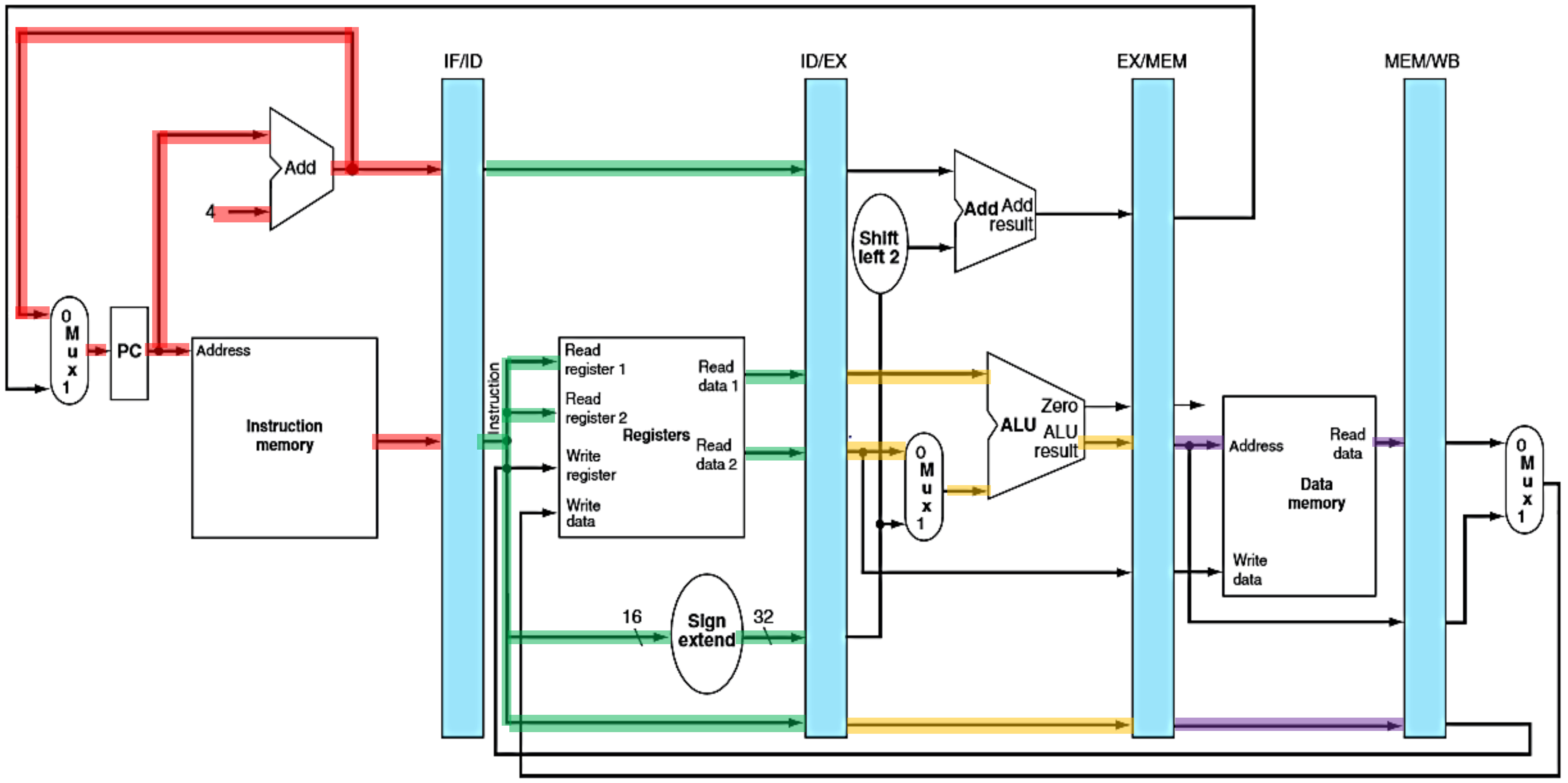
PIPELINED DATAPATH

add $12,$3,$4 sub $11,$2,$3 lw $10, 20($1)



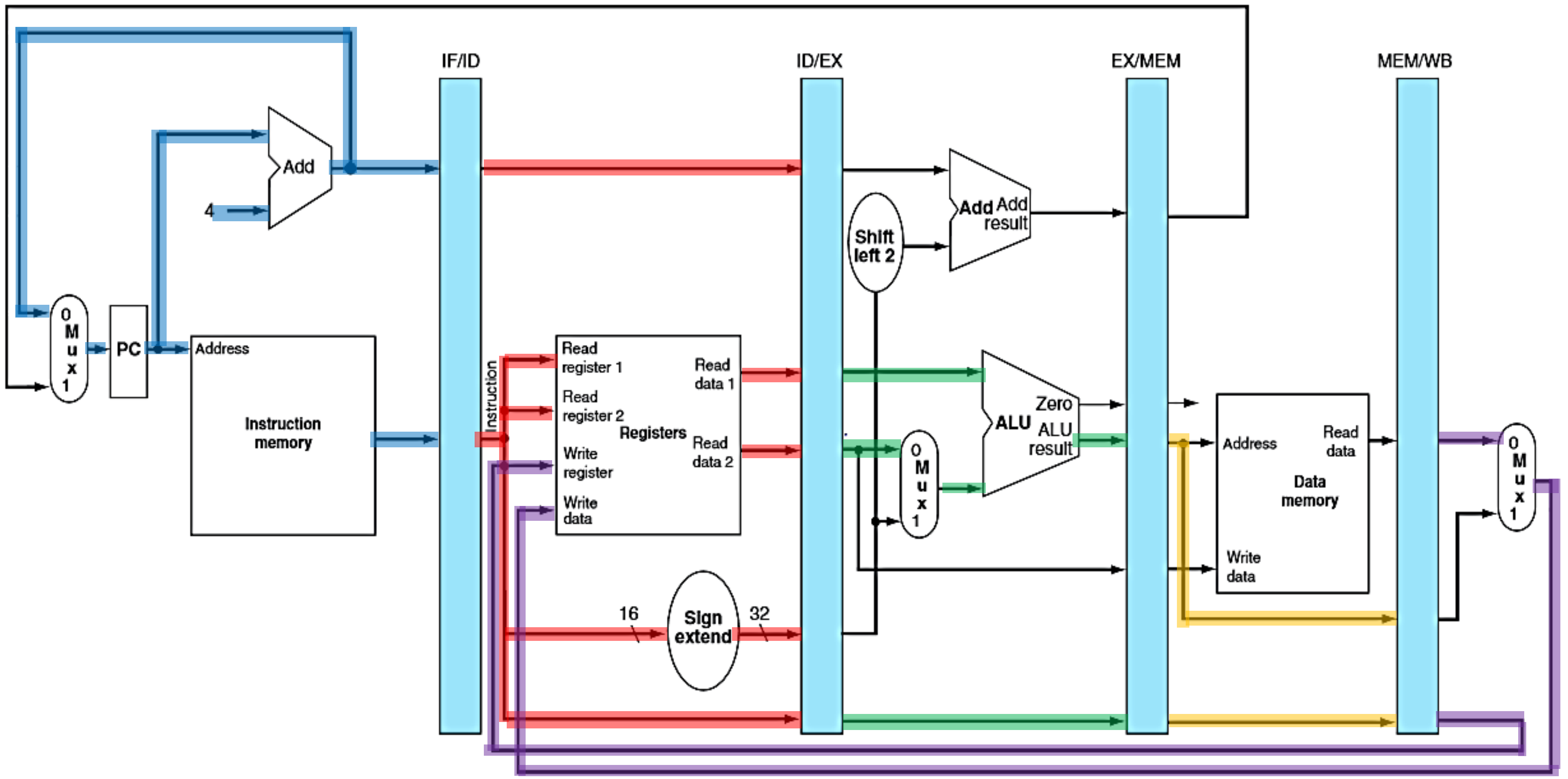
PIPELINED DATAPATH

lw $13, 24($1)   add $12,$3,$4   sub $11,$2,$3   lw $10, 20($1)



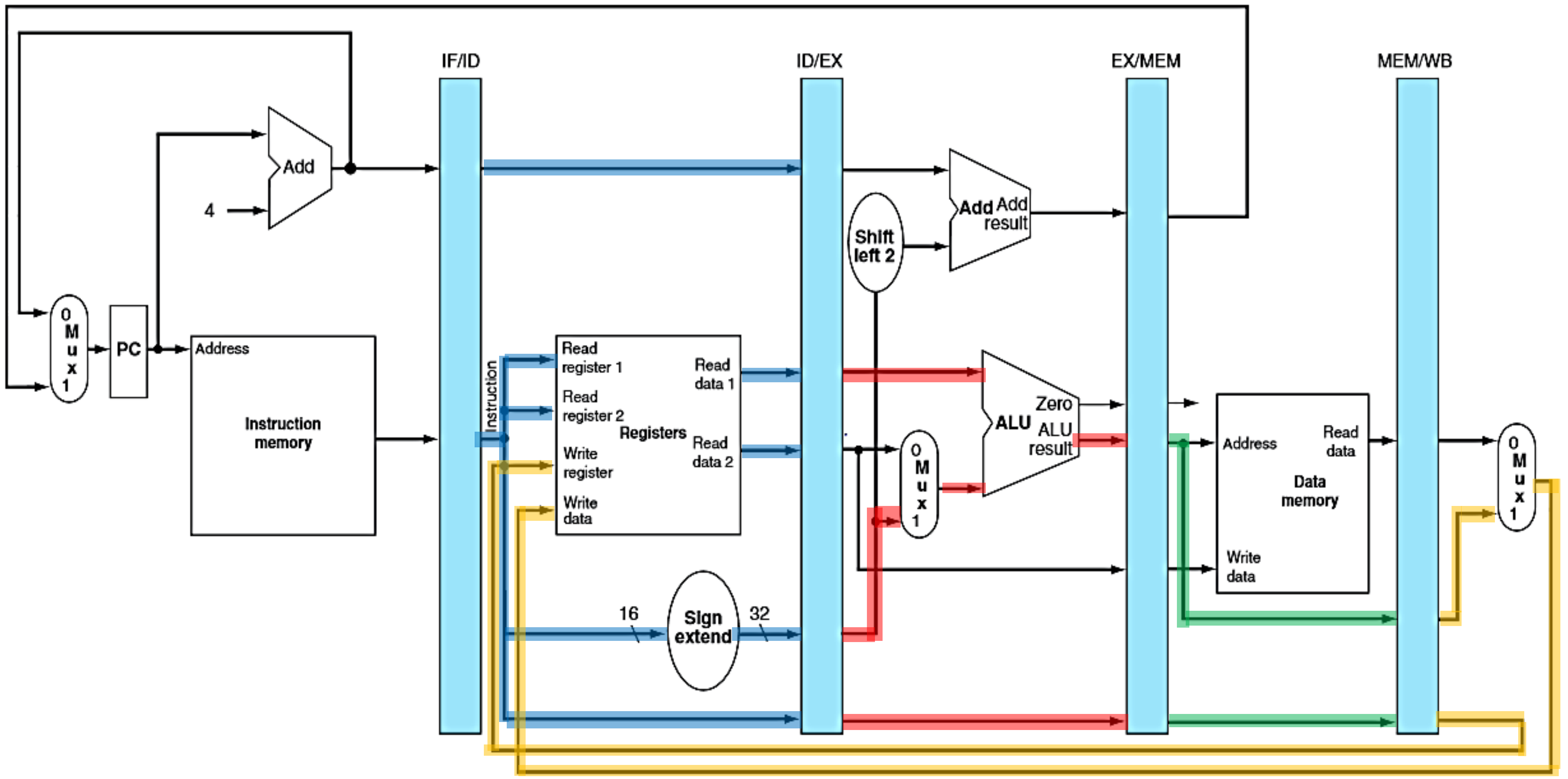
PIPELINED DATAPATH

`add $14,$5,$6` `lw $13, 24($1)` `add $12,$3,$4` `sub $11,$2,$3` `lw $10, 20($1)`



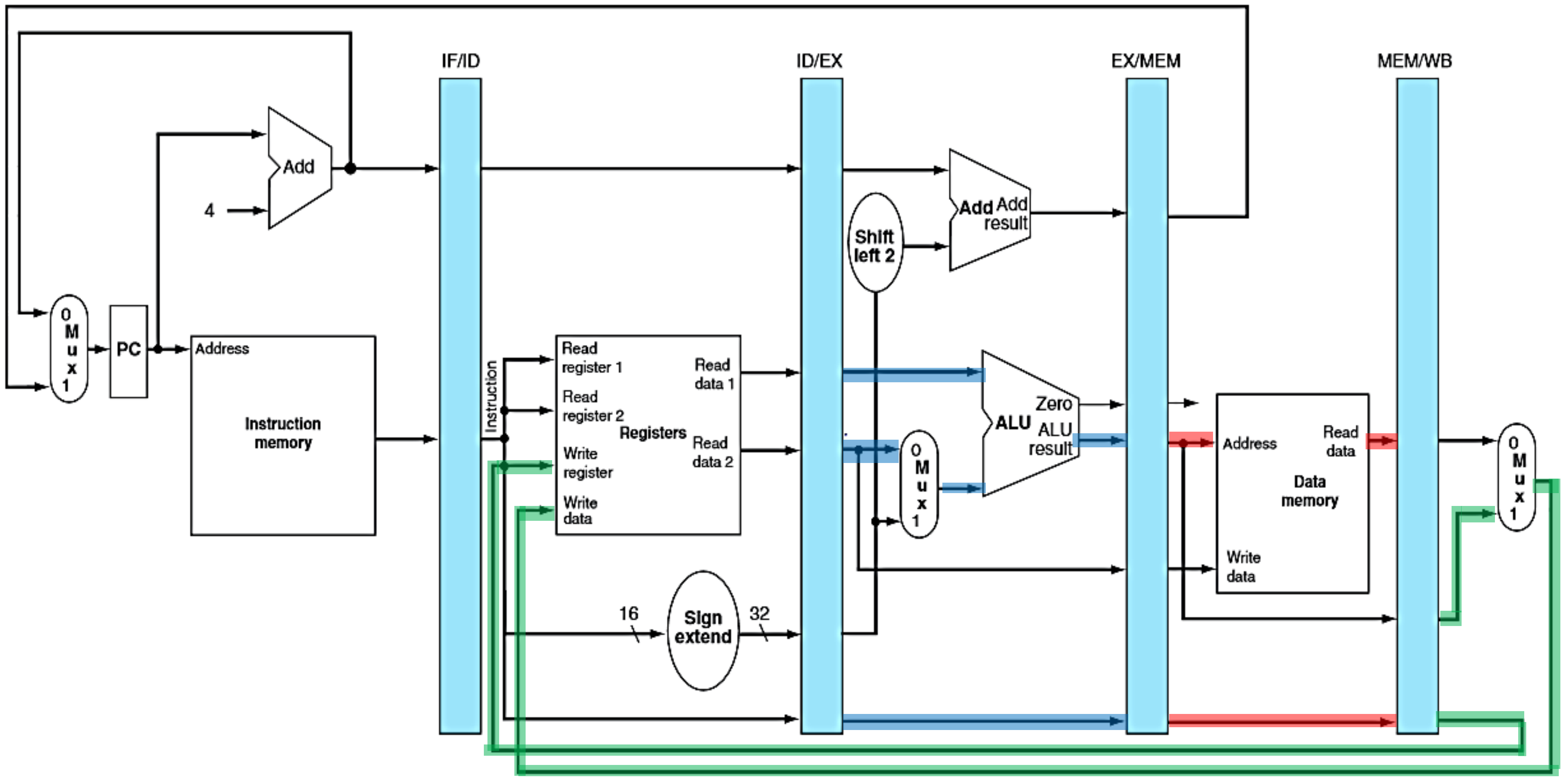
PIPELINED DATAPATH

add $14,$5,$6 lw $13, 24($1) add $12,$3,$4 sub $11,$2,$3



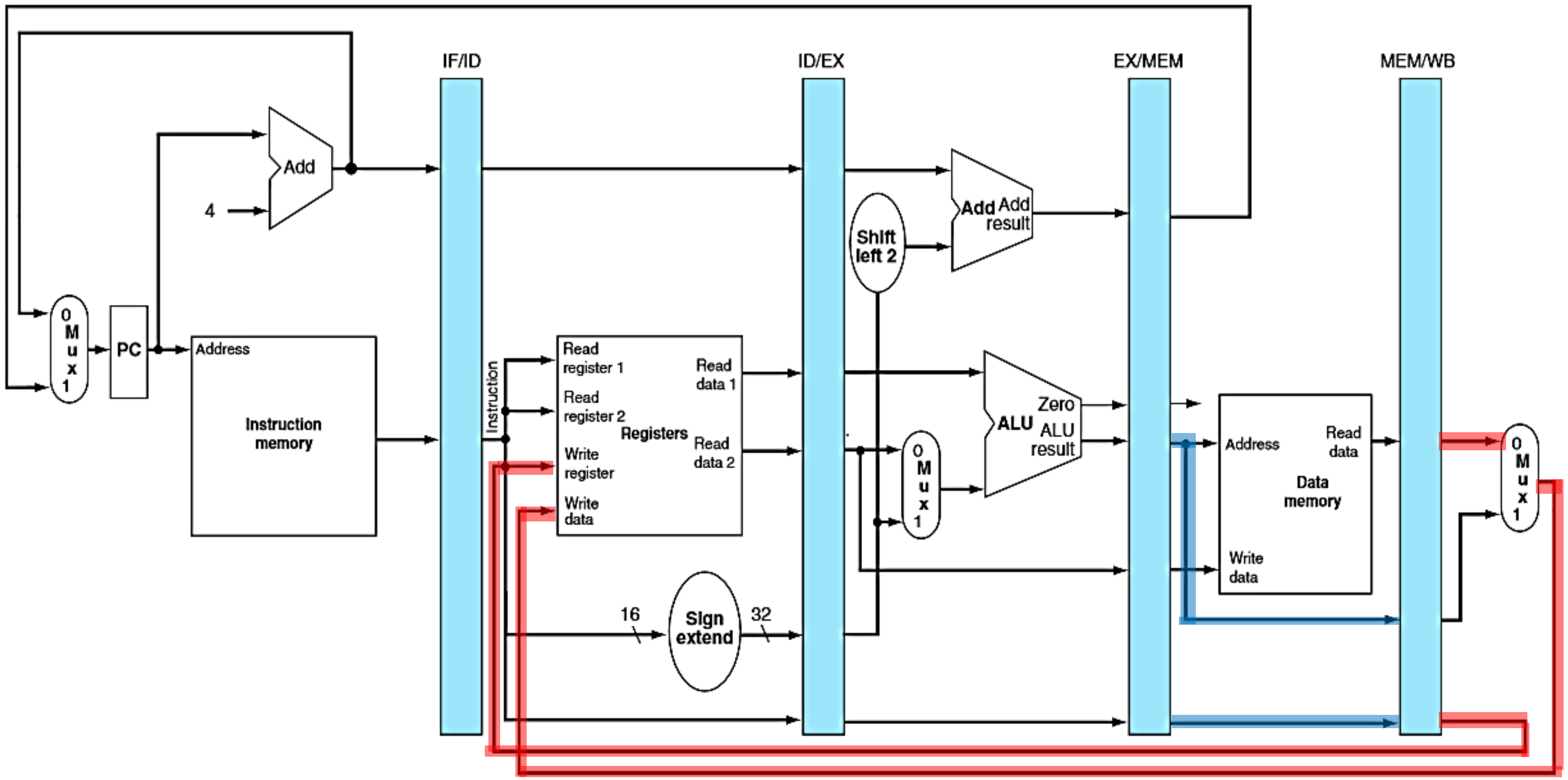
PIPELINED DATAPATH

add $14,$5,$6 lw $13, 24($1) add $12,$3,$4



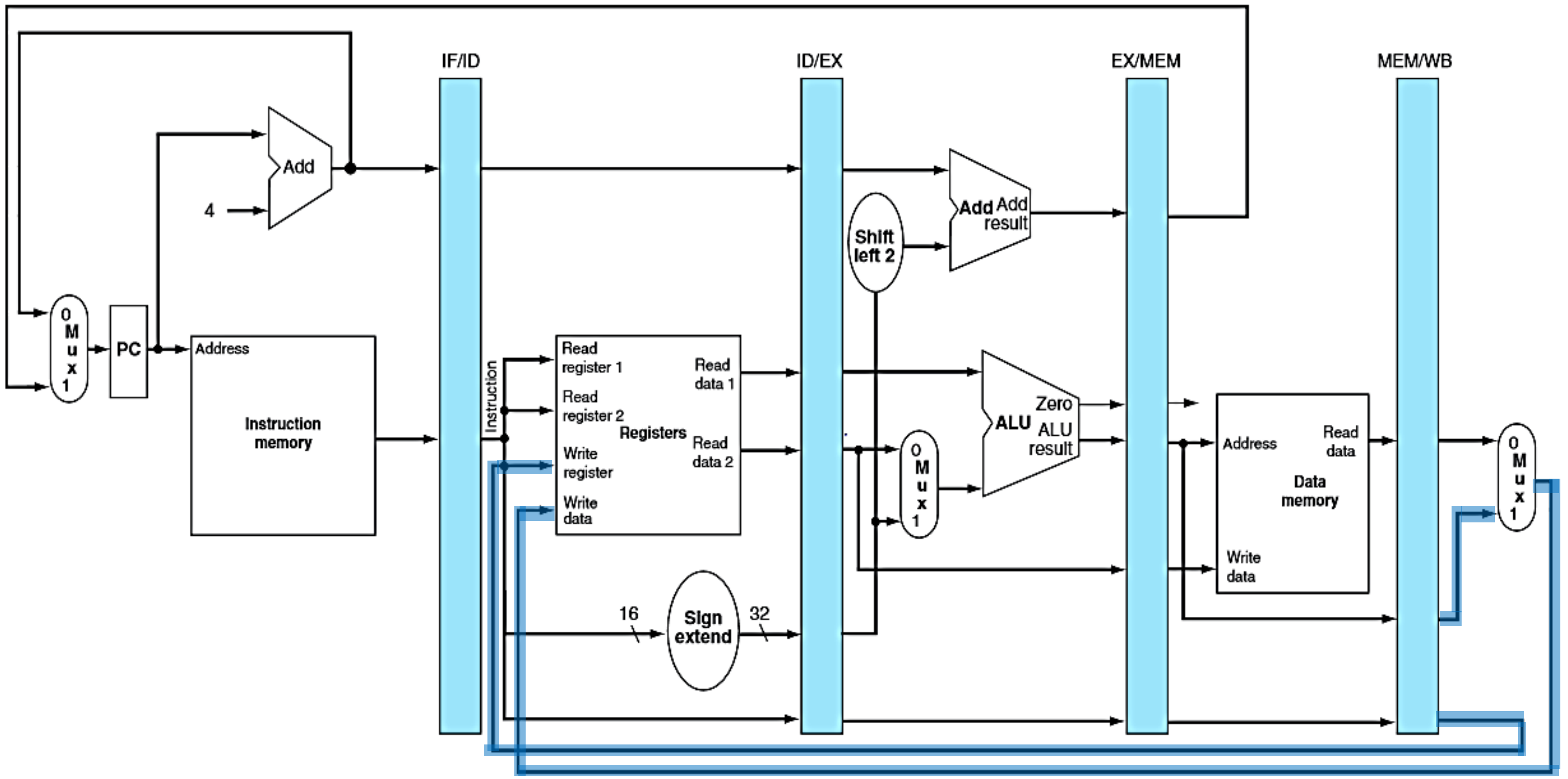
PIPELINED DATAPATH

add $14,$5,$6 lw $13, 24($1)



PIPELINED DATAPATH

`add $14,$5,$6`

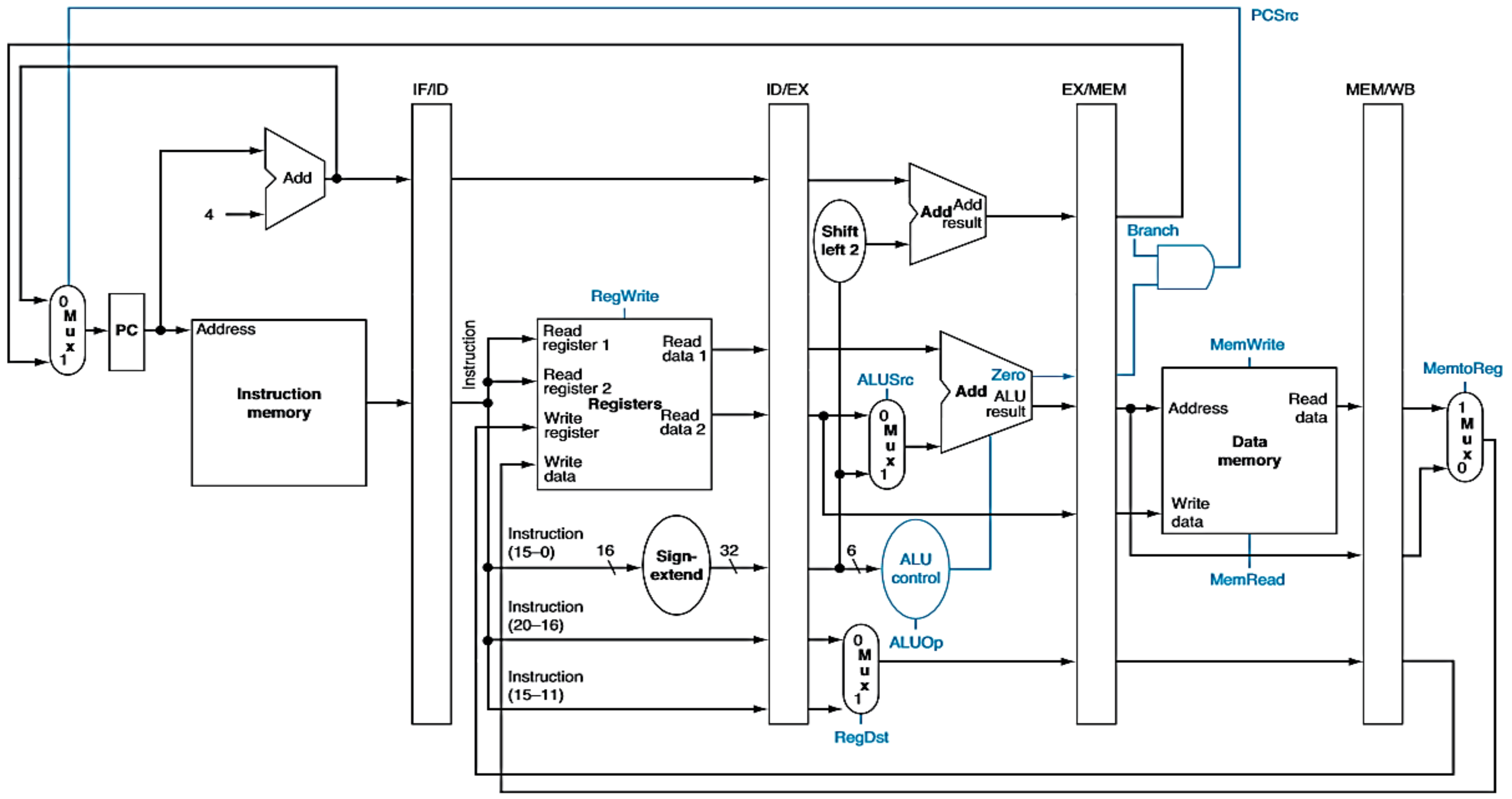


PIPELINED CONTROL

Now, let’s add control to our pipelined datapath. We’ll start with a simple control scheme and deal with pipeline hazards later.

A lot of the control logic is borrowed from the single-cycle and multi-cycle implementations.

PIPELINED CONTROL



PIPELINED CONTROL

Let's remind ourselves of the roles of these control lines.

Opcode	ALU op	Operation	Funct	ALU action	ALU Control Input
lw	00	Load word	N/A	add	0010
sw	00	Store word	N/A	add	0010
beq	01	Branch equal	N/A	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	slt	0111

PIPELINED CONTROL

1-Bit Signal Name	Effect When Deasserted	Effect When Asserted
RegDst	The register file destination number for the Write register comes from the rt field.	The register file destination number for the Write register comes from the rd field.
RegWrite	None	Write register is written with the value of the Write data input.
ALUSrc	The first ALU operand is Read Data 2.	The first ALU operand is sign-ext immediate field.
PCSrc	PC is replaced by PC+4.	PC is replaced by branch target.
MemRead	None	Content of memory at the location specified by the Address input is put on the Memory data output.
MemWrite	None	The Write Data input is written to the Address input.
MemtoReg	The value fed to the register file is from the ALU.	The value fed to the register file input comes from Memory.

PIPELINED CONTROL

Let's remind ourselves of the roles of these control lines. Notice now that we're grouping the control lines based on their relevant stage.

	EX				MEM			WB	
Instr.	**RegDst**	**ALUOp1**	**ALUOp2**	**ALUSrc**	**Branch**	**MemRead**	**MemWrite**	**RegWrite**	**MemToReg**
R	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

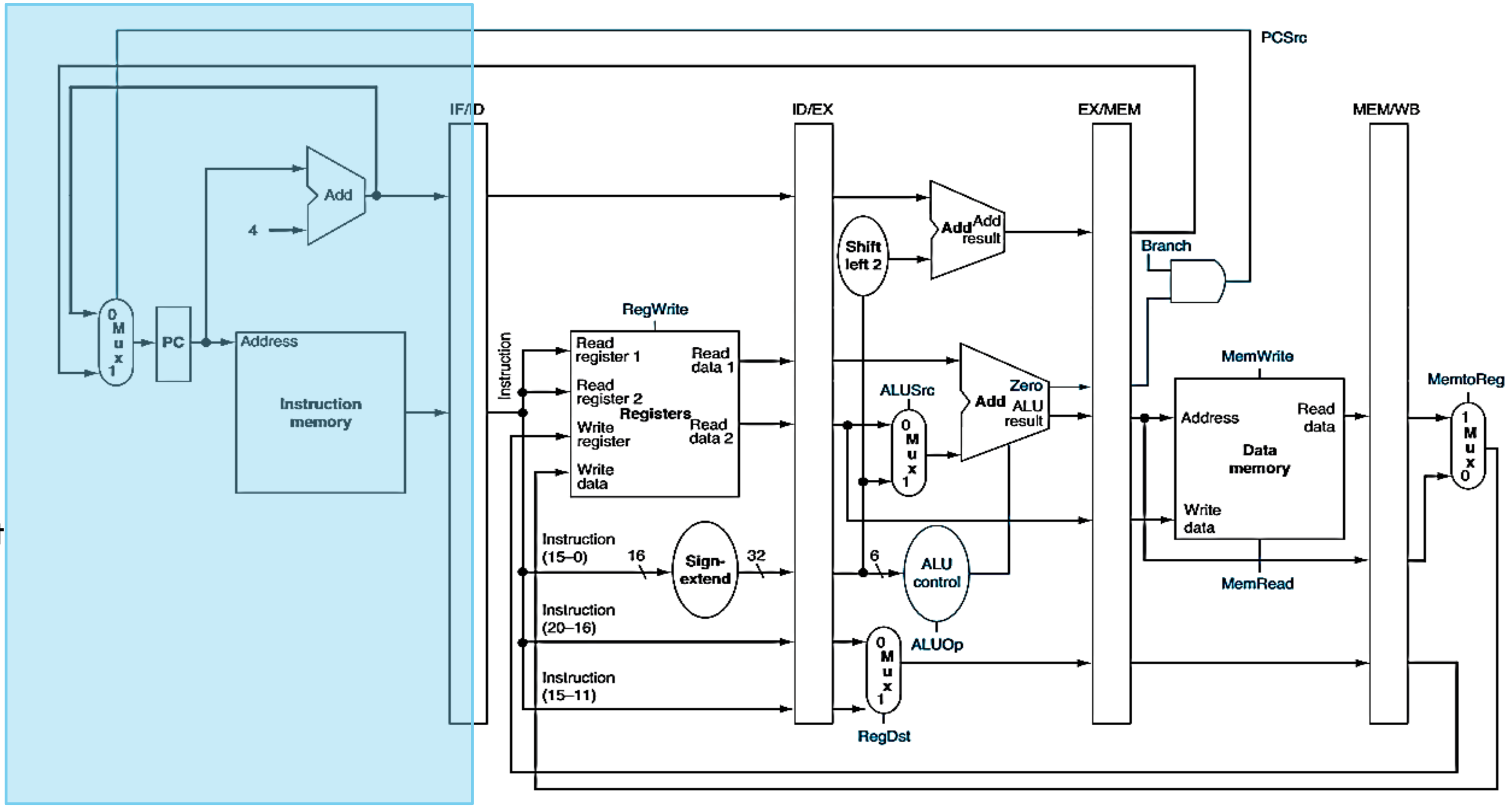
Unlike the multi-cycle implementation, we assume that the **PC is written on every clock cycle** so there is no explicit control signal for writing PC. We also assume that the pipeline registers are written on every clock cycle.

As we just saw in the previous slide, we can handle “controlling” 5 instructions at once by **dividing the control signals into 5 groups** – one for each pipeline stage.

PIPELINED CONTROL: IF

The only relevant control signals in this stage are the control signals to read instruction memory and write to PC.

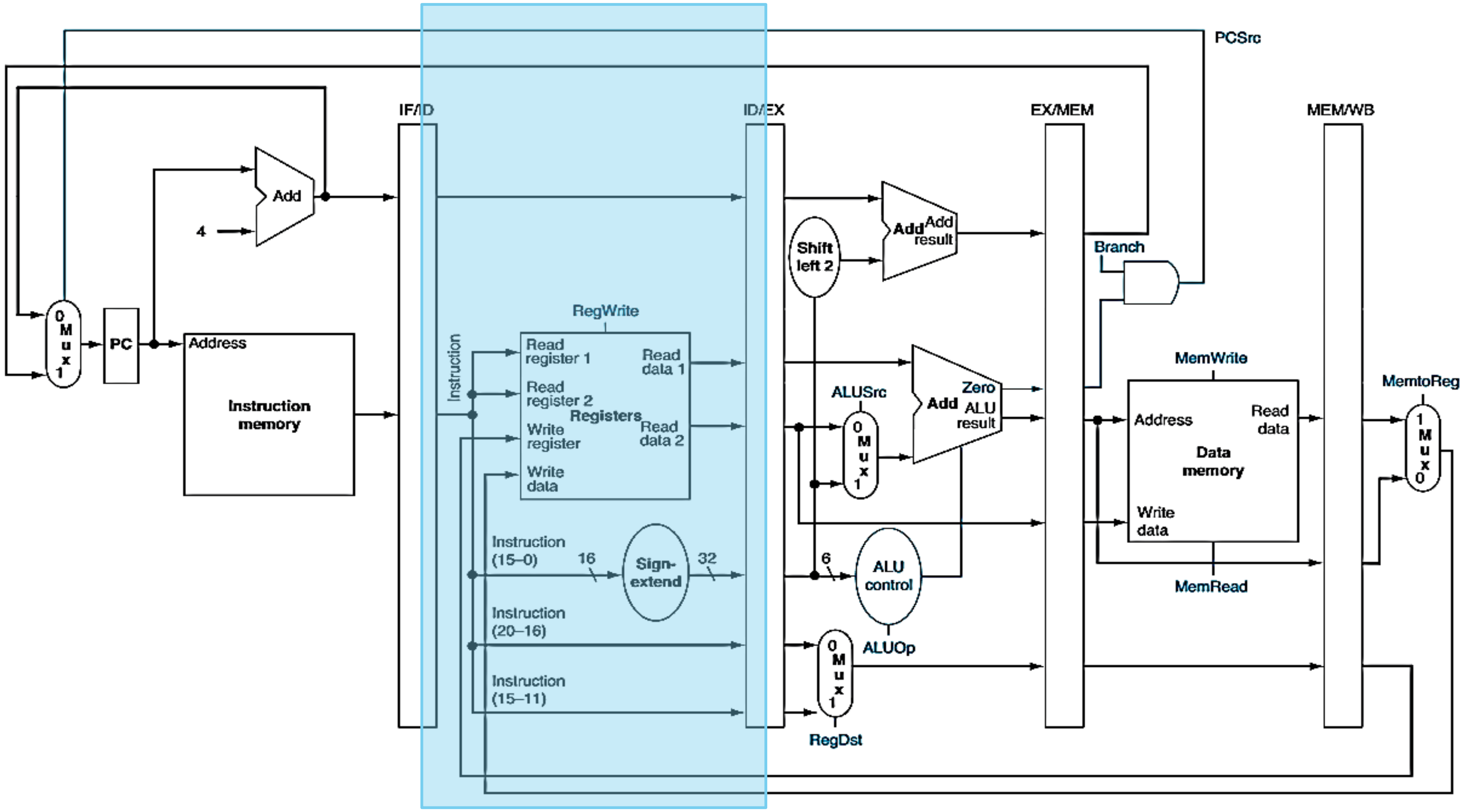
These control signals are always asserted however so we don't need to worry about them.



PIPELINED CONTROL: ID

As in the previous stage, the same actions take place during every cycle so we do not have to be concerned with any optional control lines.

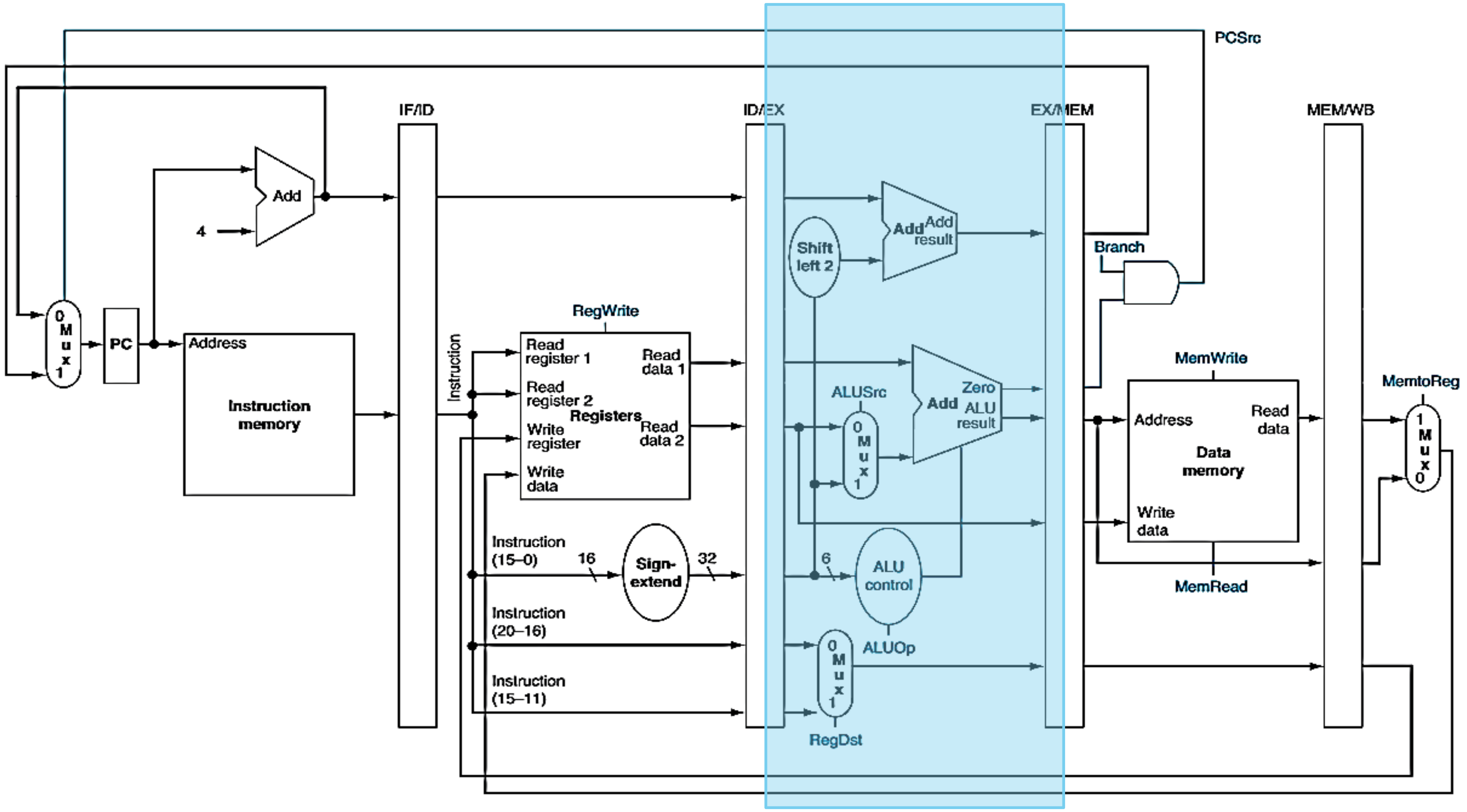
Notice the **RegWrite** signal physically resides in this component of the datapath but it is a control line of **the WB stage**.



PIPELINE CONTROL: EX

The relevant control signals in this stage are:

- RegDst
- ALUOp
- ALUSrc

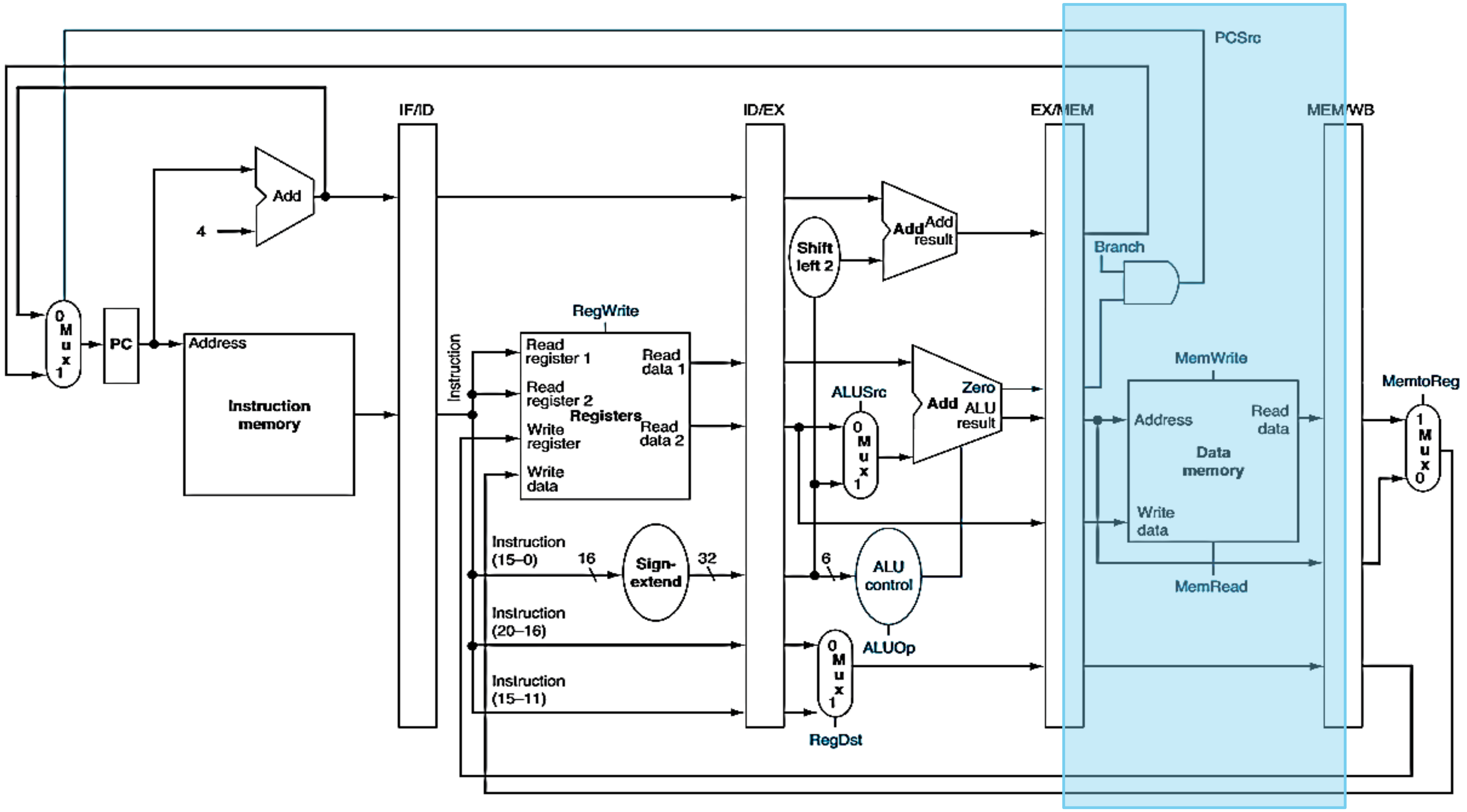


PIPELINE CONTROL: MEM

The relevant control signals in this stage are:

- **Branch**
- **MemRead**
- **MemWrite**

PCSrc is also computed in this stage but it is **a product of two other control signals,** so we don't worry about it when designing our Control unit.

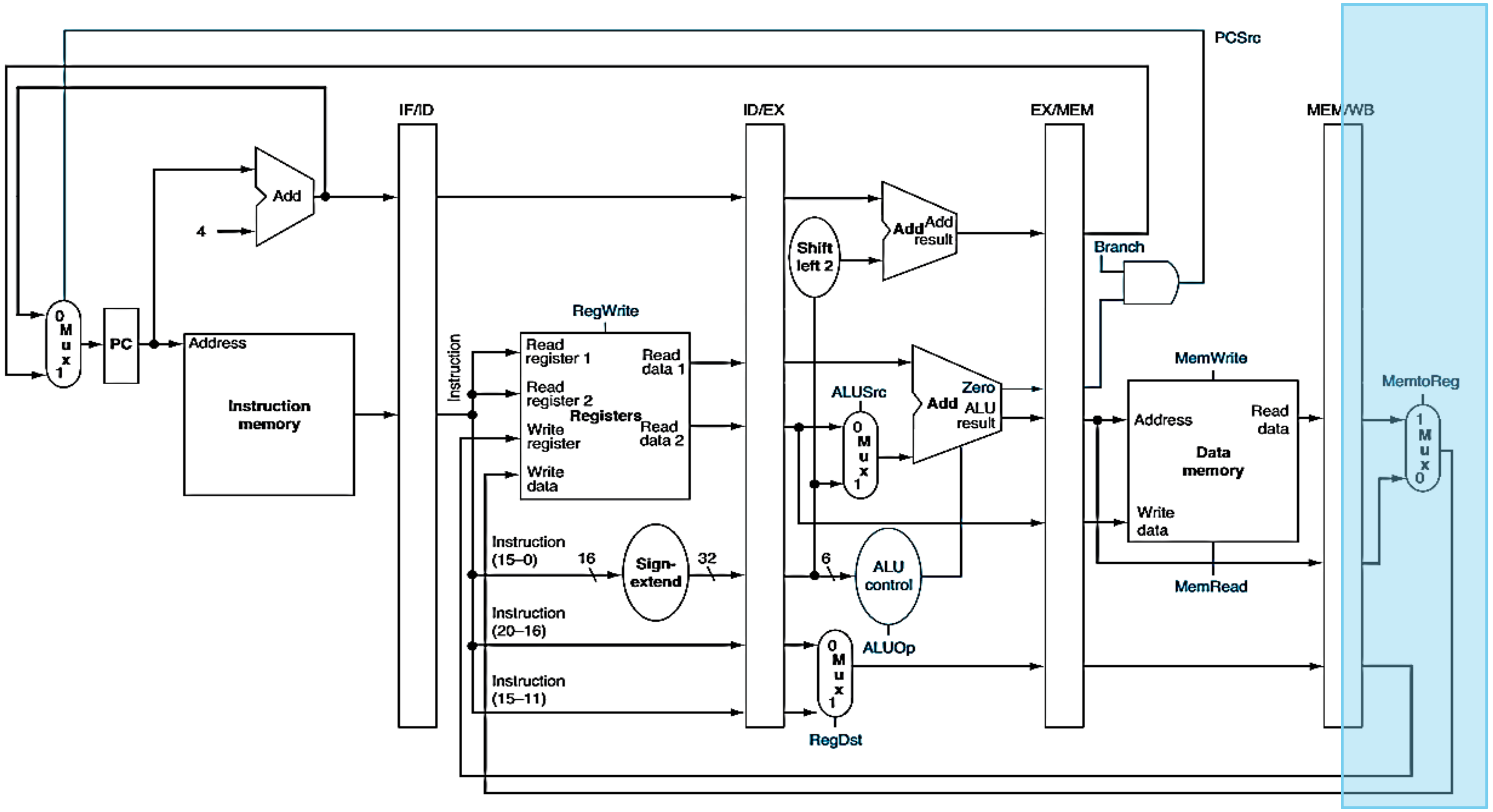


PIPELINE CONTROL: WB

The relevant control signals in this stage are:

- **MemtoReg**
- **RegWrite**

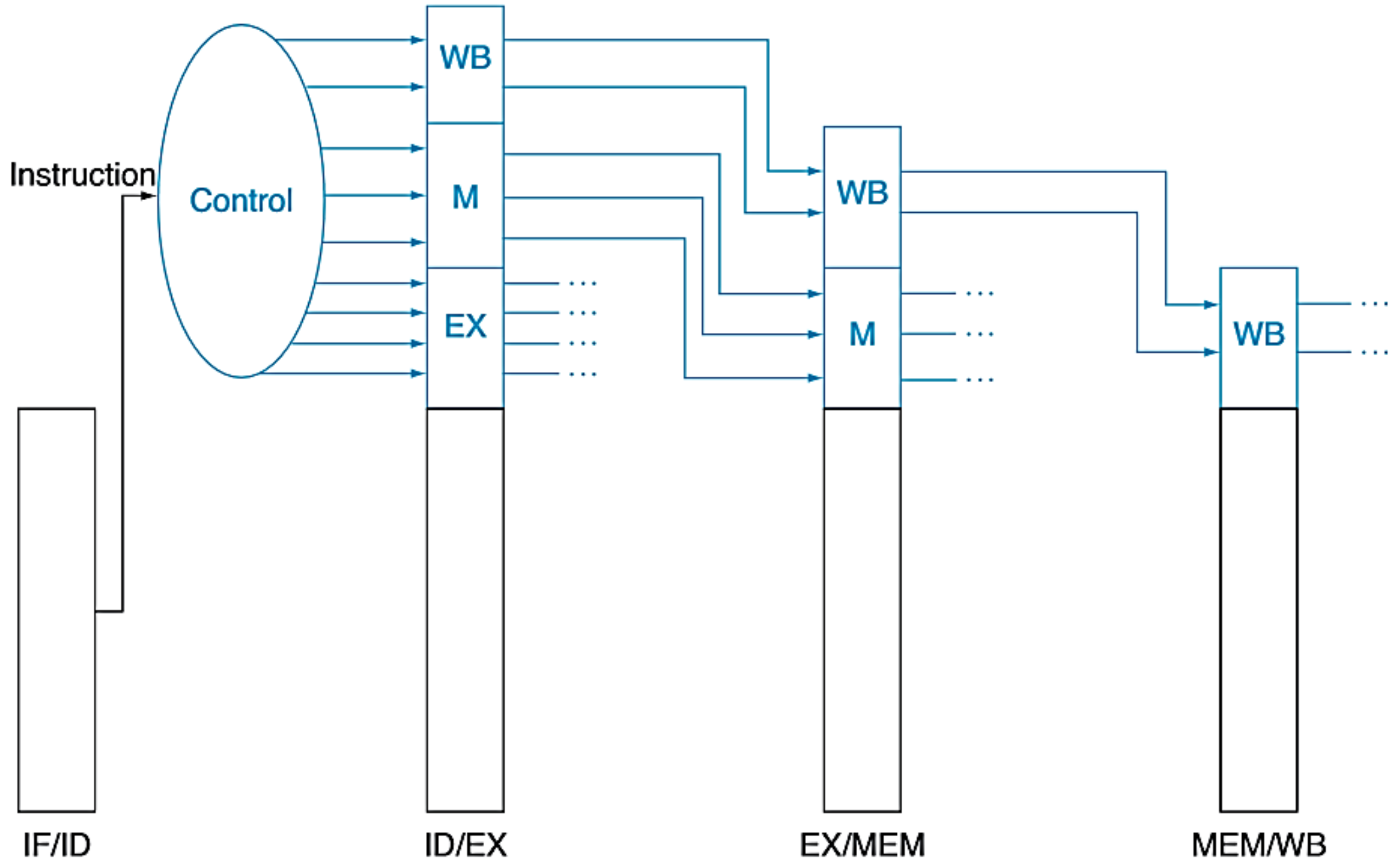
Note that RegWrite is our one exception. It actually appears in ID's phase of the pipeline but it is a control signal for the WB stage.



PIPELINED CONTROL

As before, we need to **preserve the context of an instruction as it moves through the datapath.**

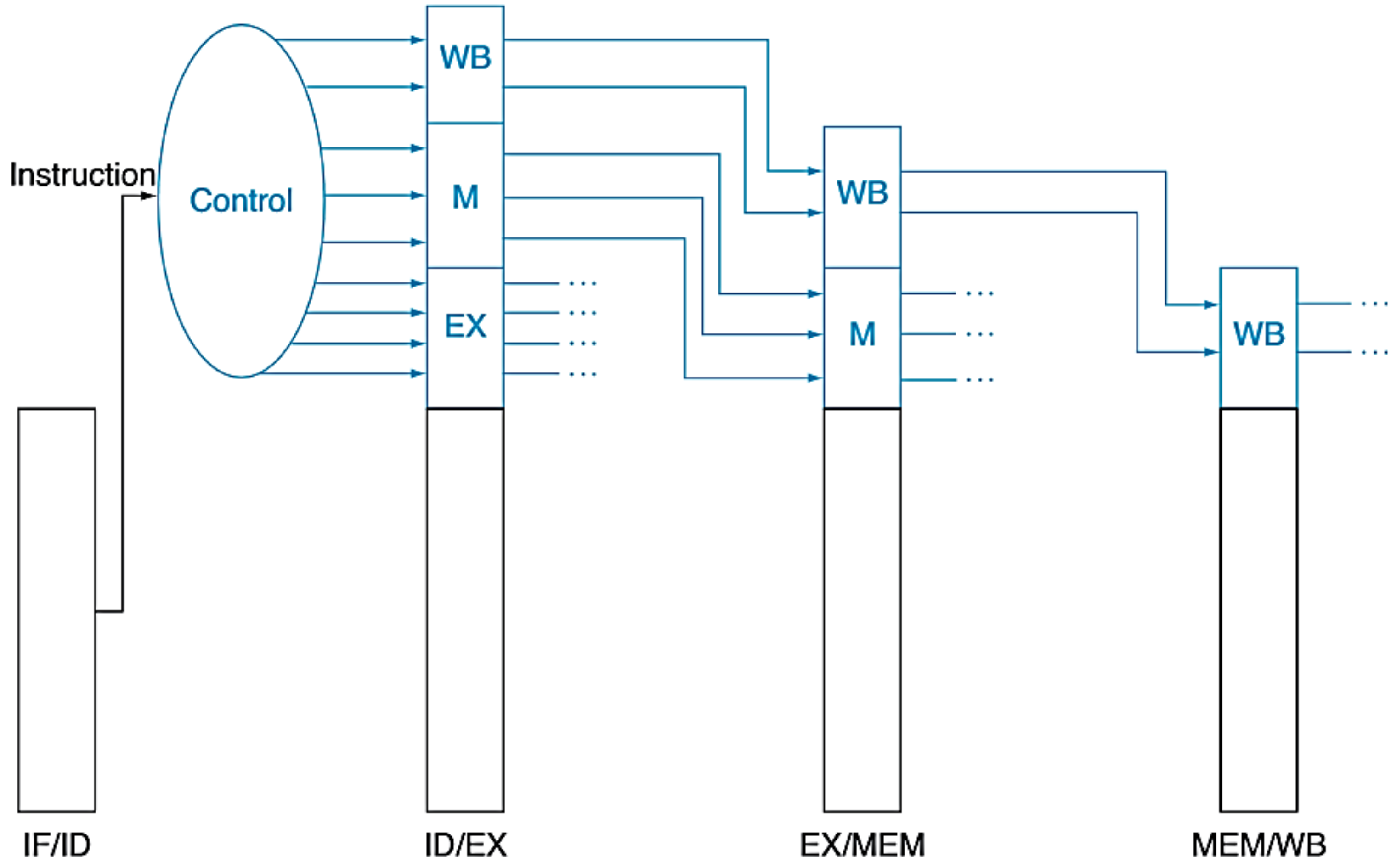
We cannot reach back in the datapath for some information about how to execute the instruction – it will have already been overwritten.



PIPELINED CONTROL

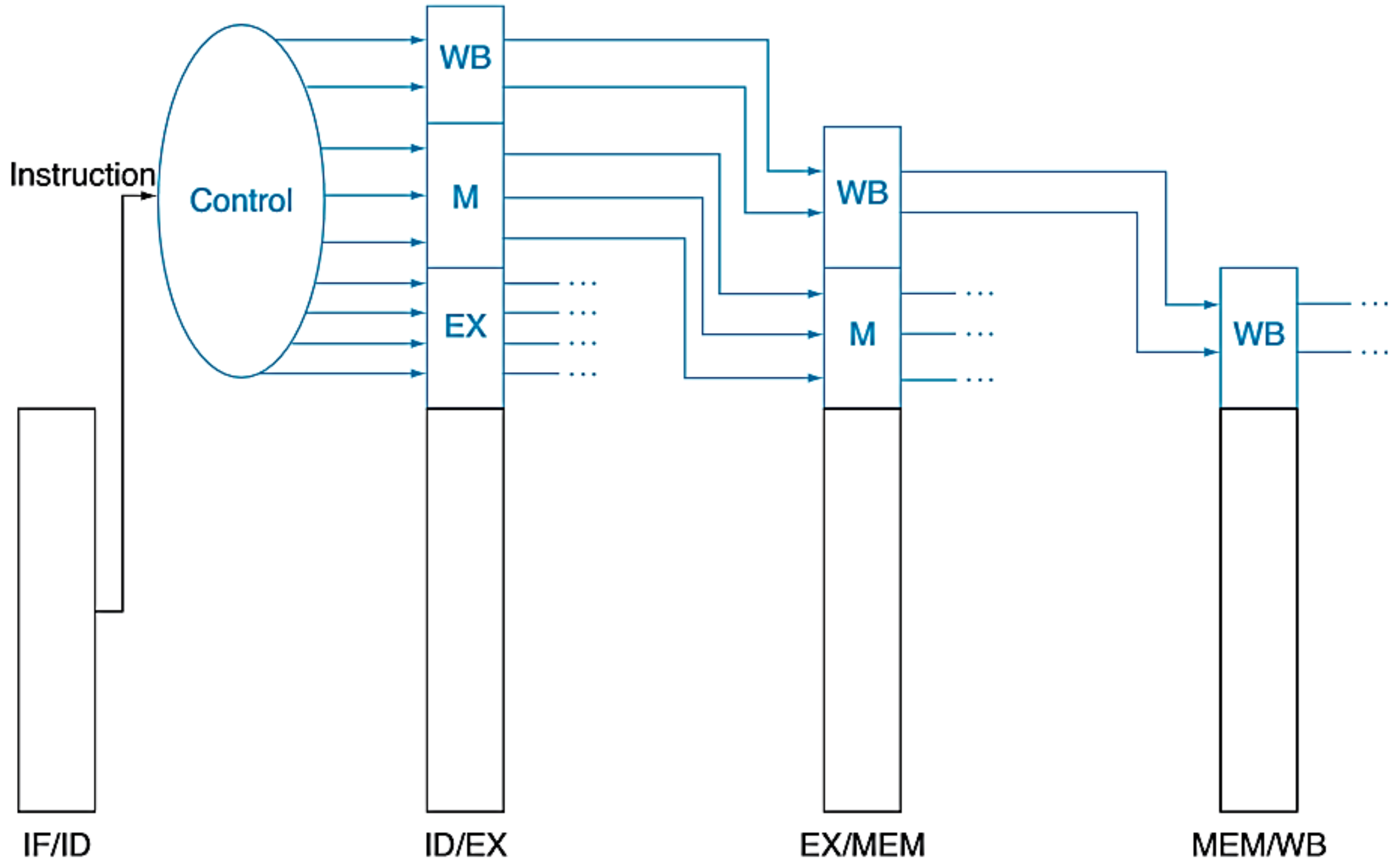
We must pass along the control line values for **the last 3 stages**. Remember, the first 2 stages are universal.

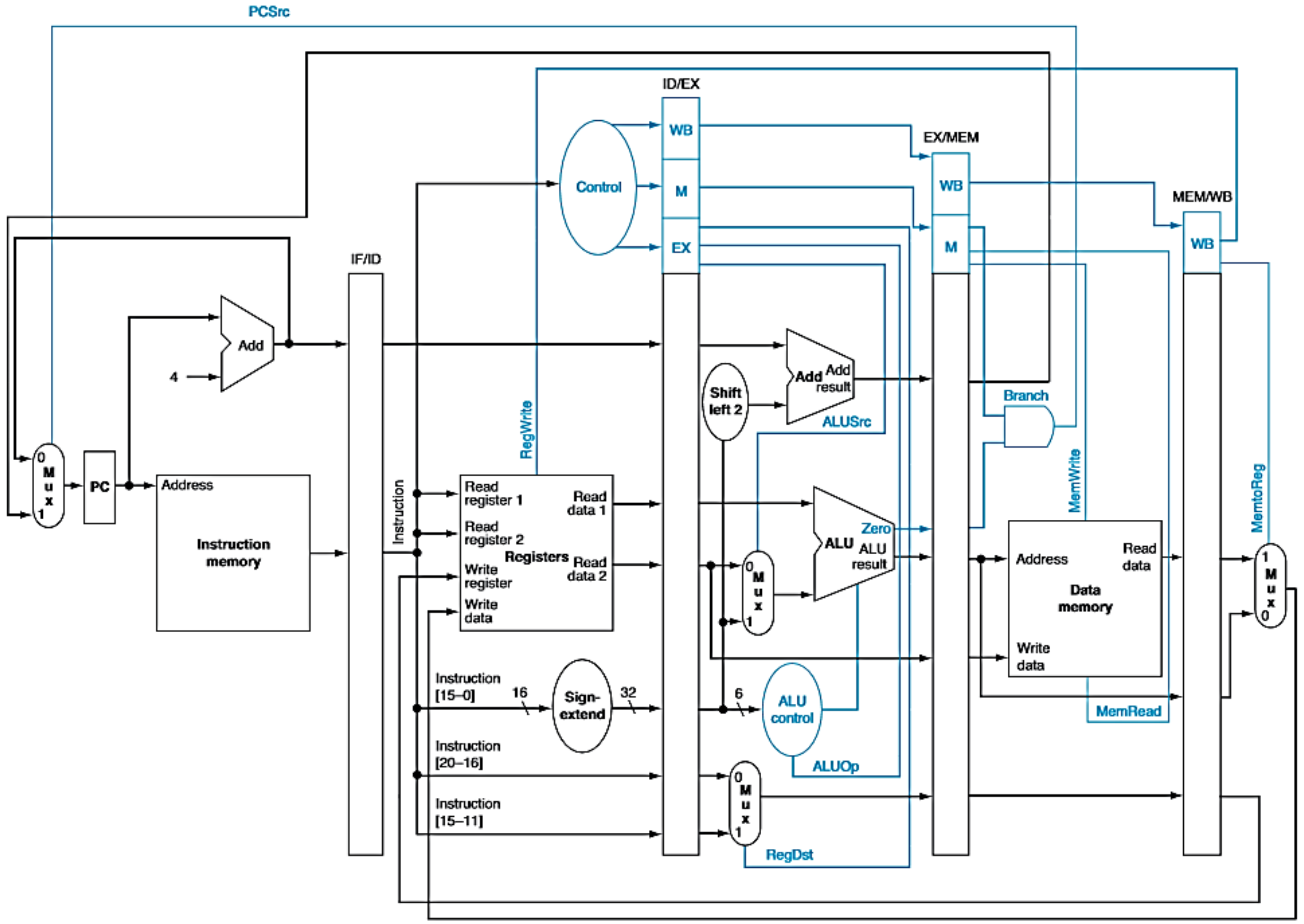
For example, the RegWrite and MemtoReg values, decoded after the second stage, must be passed through the EX and MEM stages to the WB stage.



PIPELINED CONTROL

Our pipeline registers have been extended to handle the control information as well.



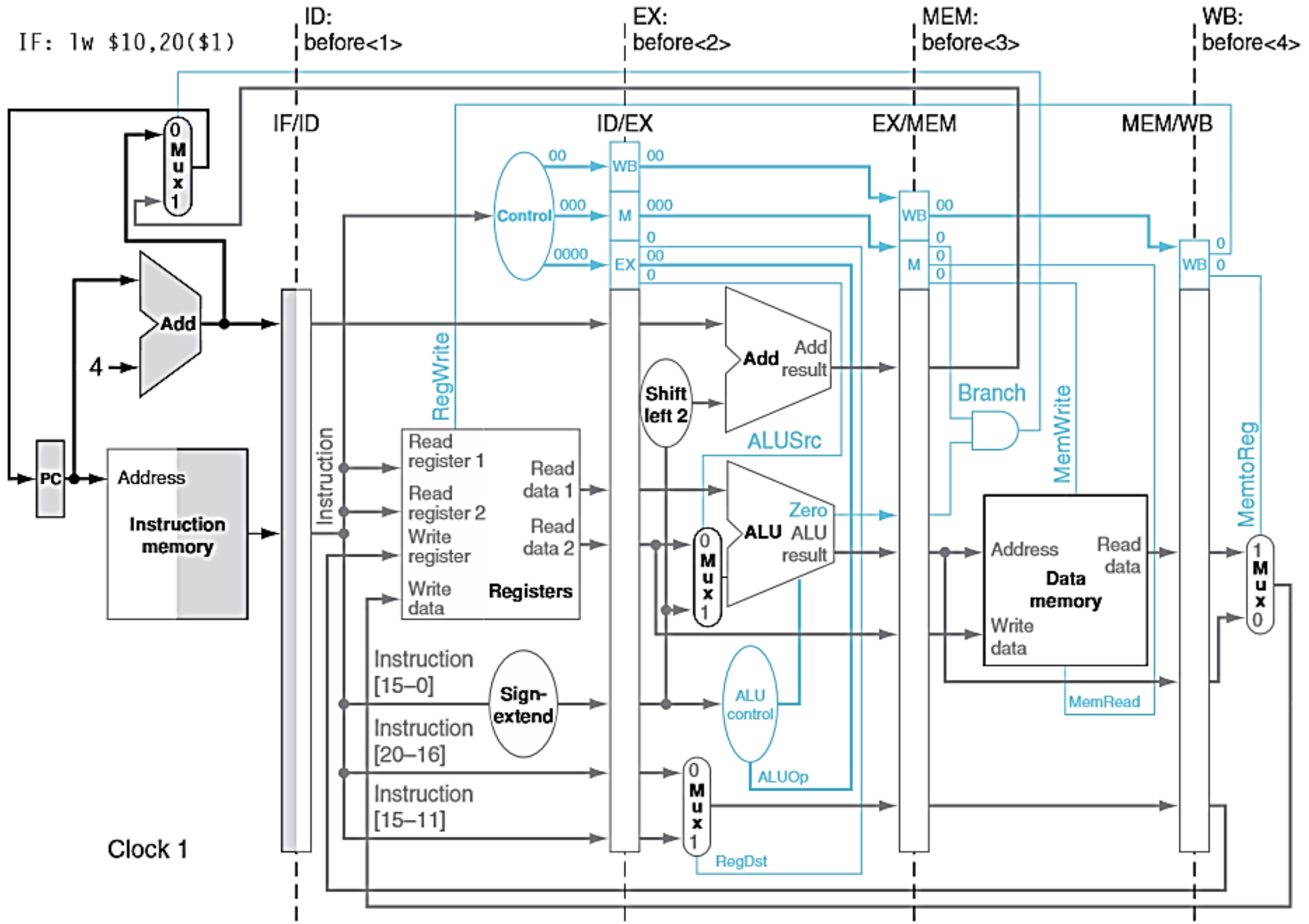

PCSrc
ID/EX
EX/MEM
MEM/WB
IF/ID
Control
WB
M
EX
WB
M
WB
Add
4
Add
Add result
Shift left 2
ALUSrc
Branch
RegWrite
MemWrite
MemtoReg
0 Mux 1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Write register
Write data
Registers
Read data 1
Read data 2
Zero
ALU
ALU result
0 Mux 1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

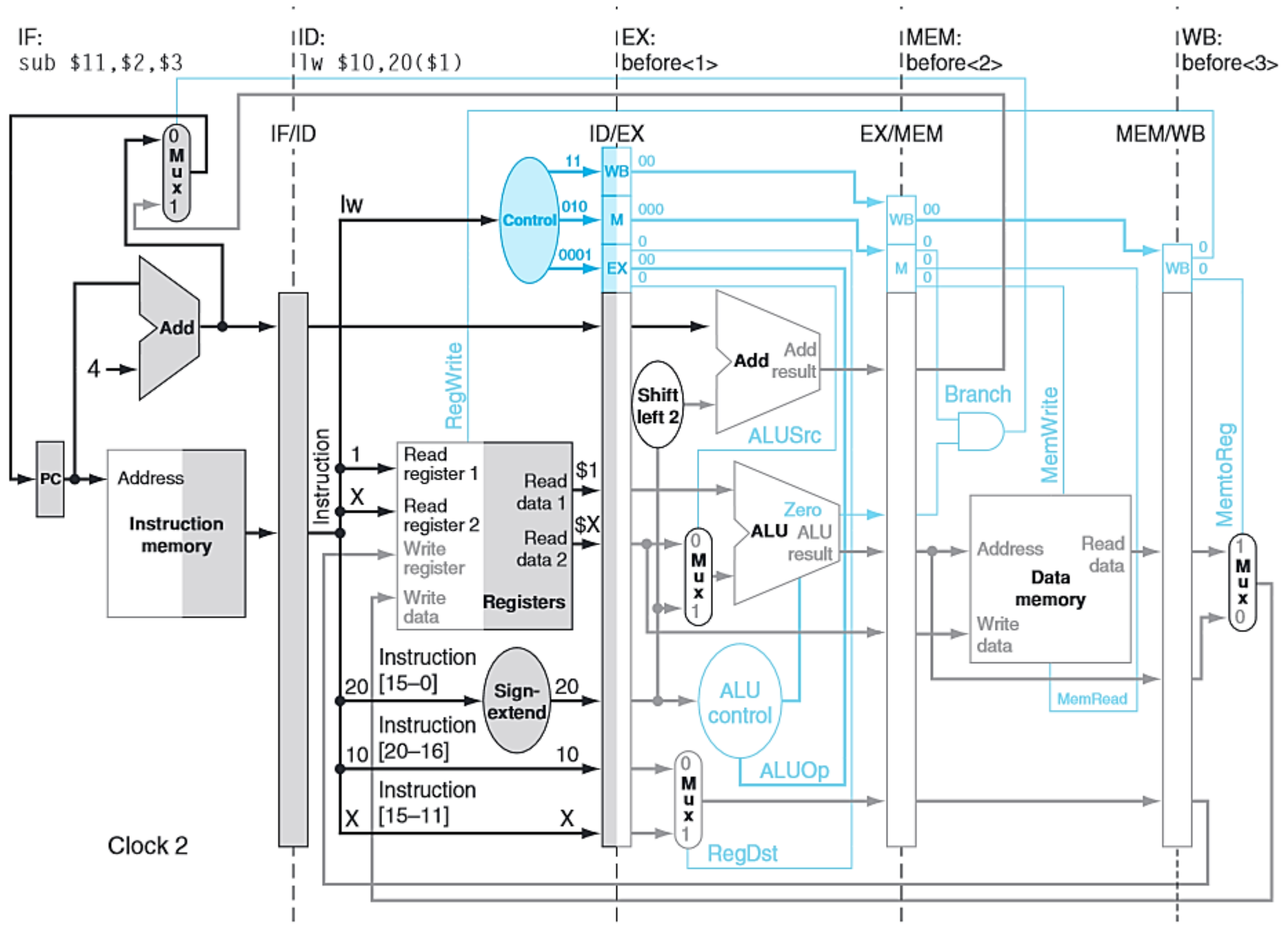
PIPELINED CONTROL

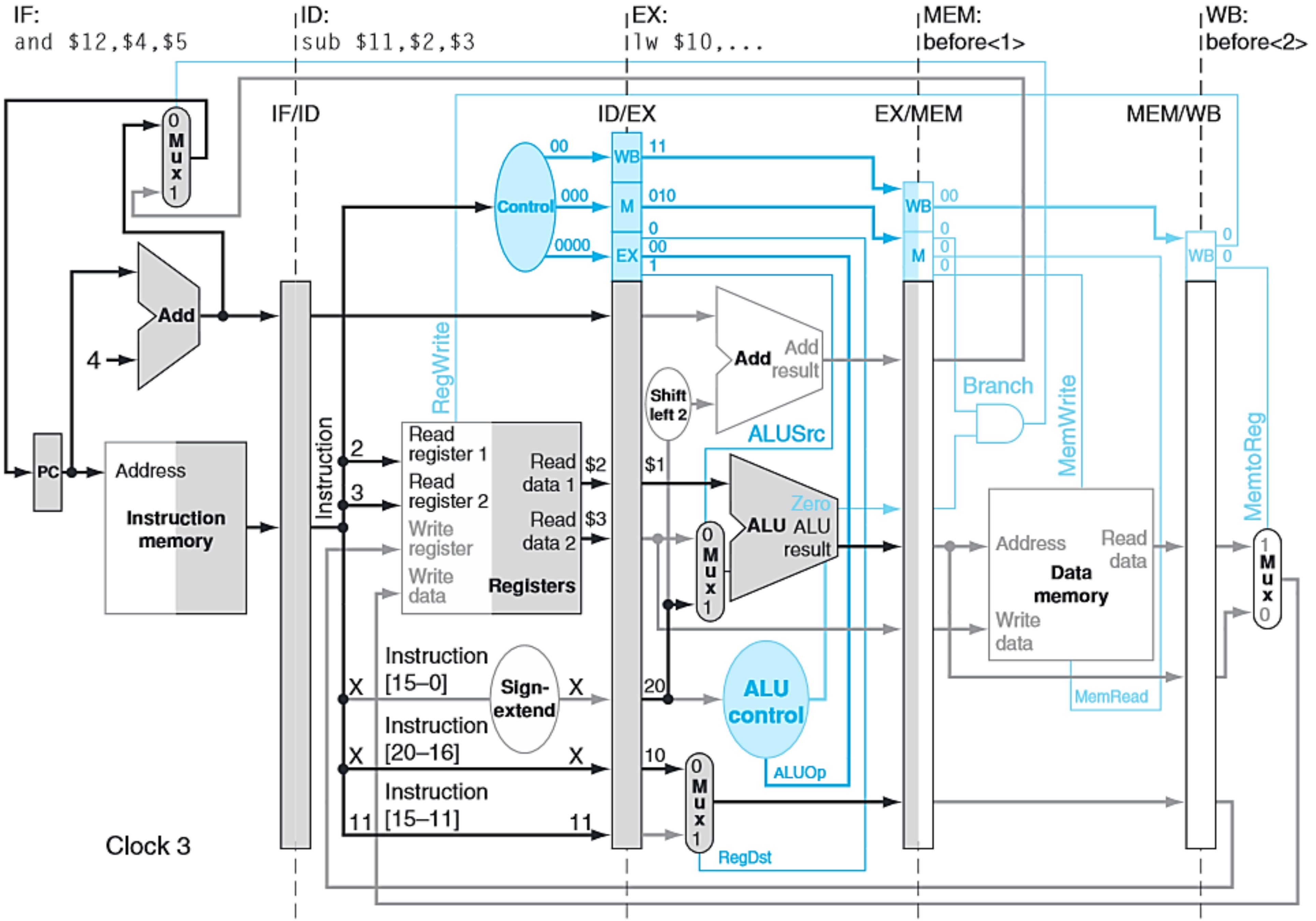
Consider the following instructions.

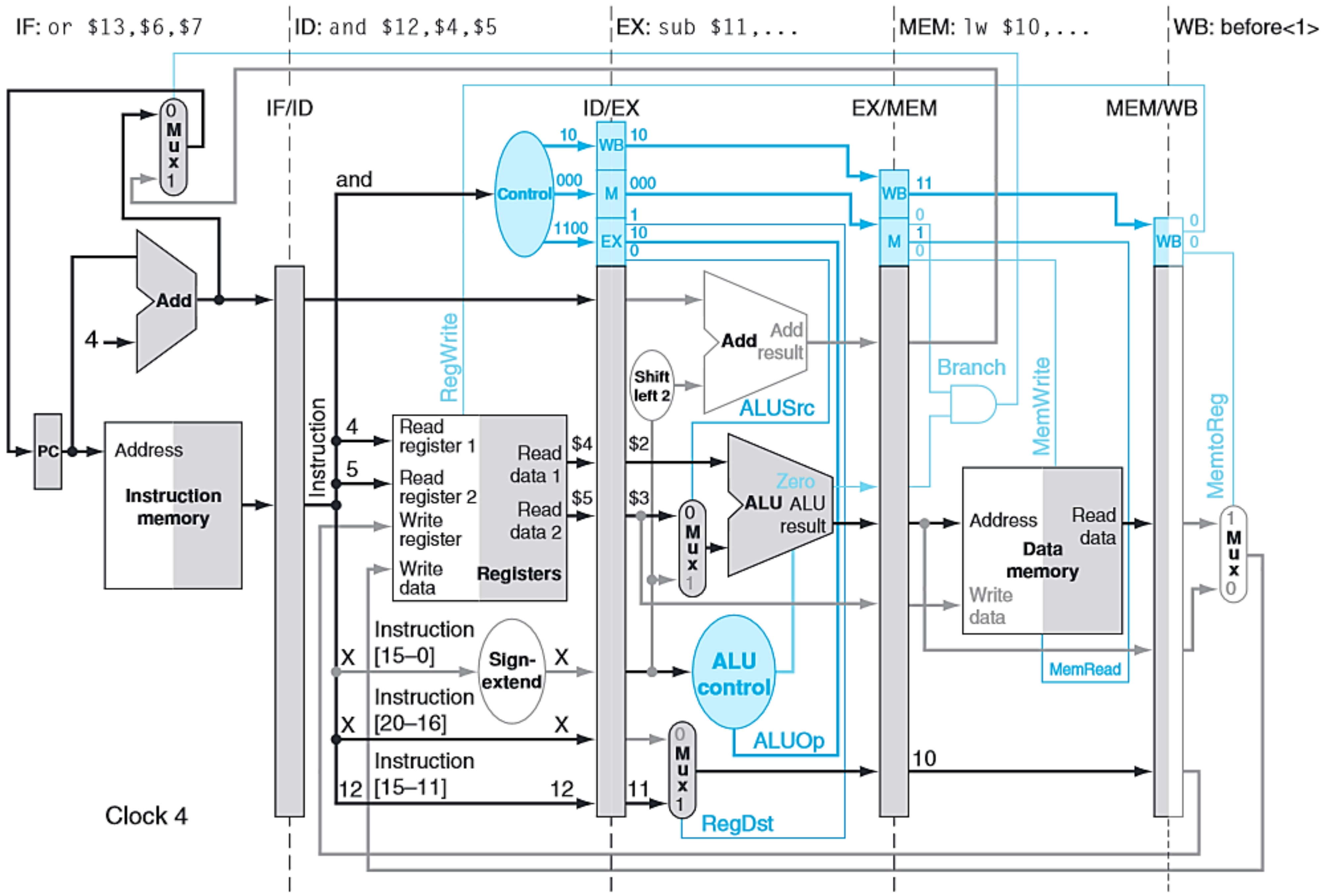
```
lw   $10, 20($1)
sub  $11, $2, $3
and  $12, $4, $5
or   $13, $6, $7
add  $14, $8, $9
```

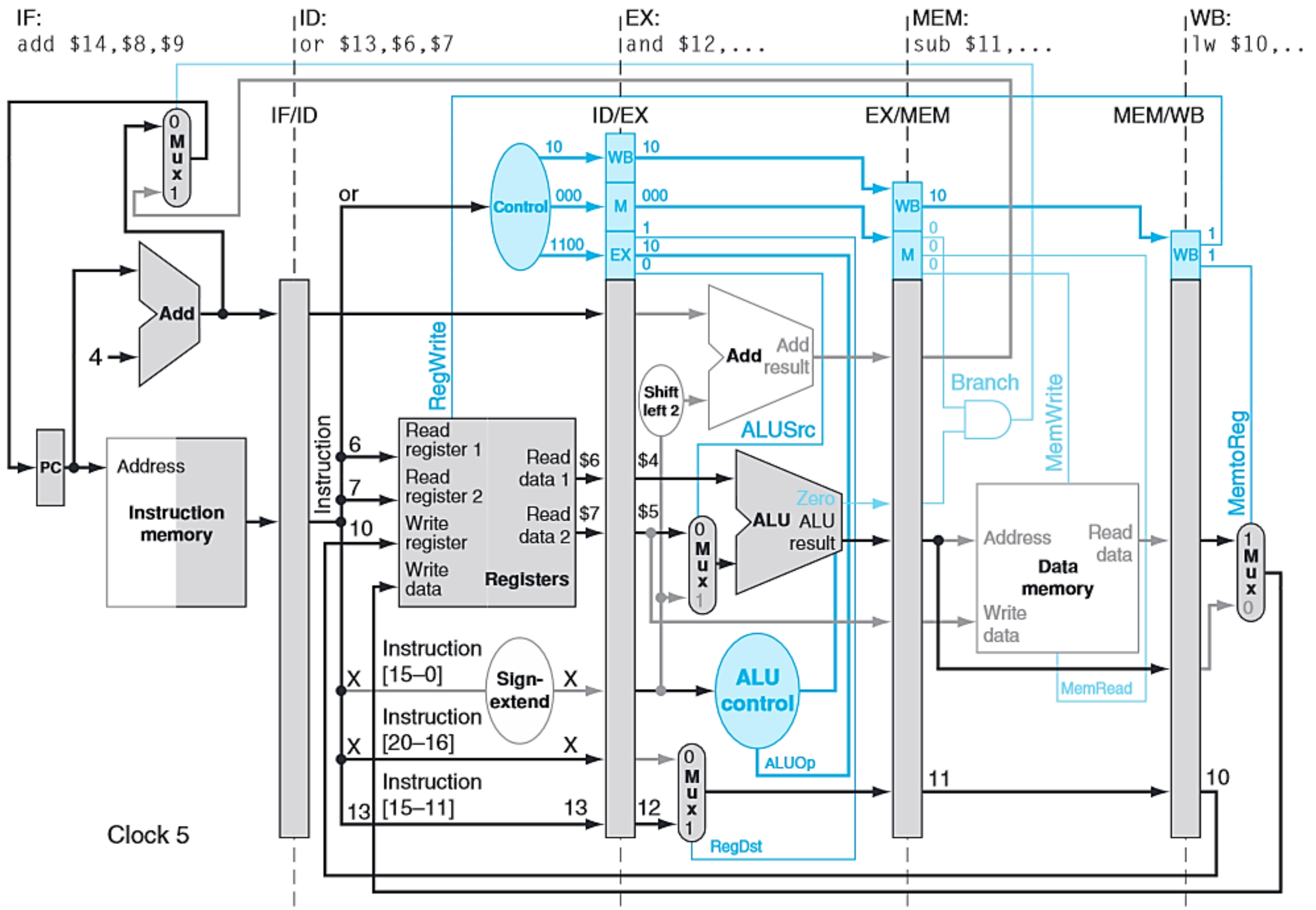
Let’s examine, in all the detail, how these simple instructions are executed in a pipelined datapath. Note again that we have no control or data hazards. Each slide following will show the datapath state in a subsequent clock cycle.

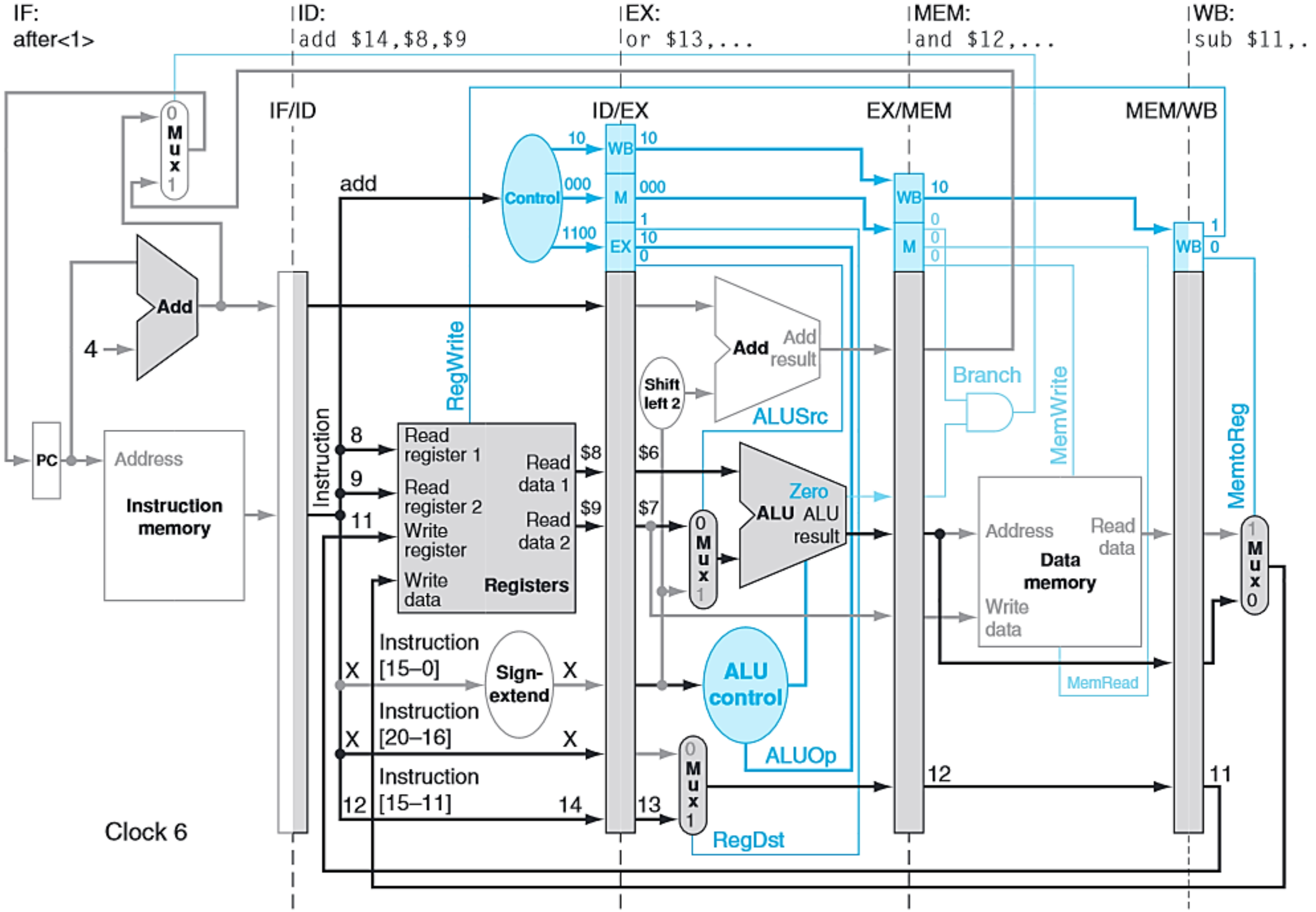

IF: lw $10,20($1)
ID: before<1>
EX: before<2>
MEM: before<3>
WB: before<4>
IF/ID
ID/EX
EX/MEM
MEM/WB
0 Mux 1
Add
4
PC
Address
Instruction memory
Instruction
Control
00
000
0000
WB
M
EX
0
RegWrite
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
Instruction [15–0]
Instruction [20–16]
Instruction [15–11]
Sign-extend
Shift left 2
Add
Add result
ALUSrc
Zero
ALU
ALU result
ALU control
ALUOp
RegDst
Branch
MemWrite
Address
Read data
Data memory
Write data
MemRead
MemtoReg
1 Mux 0
Clock 1

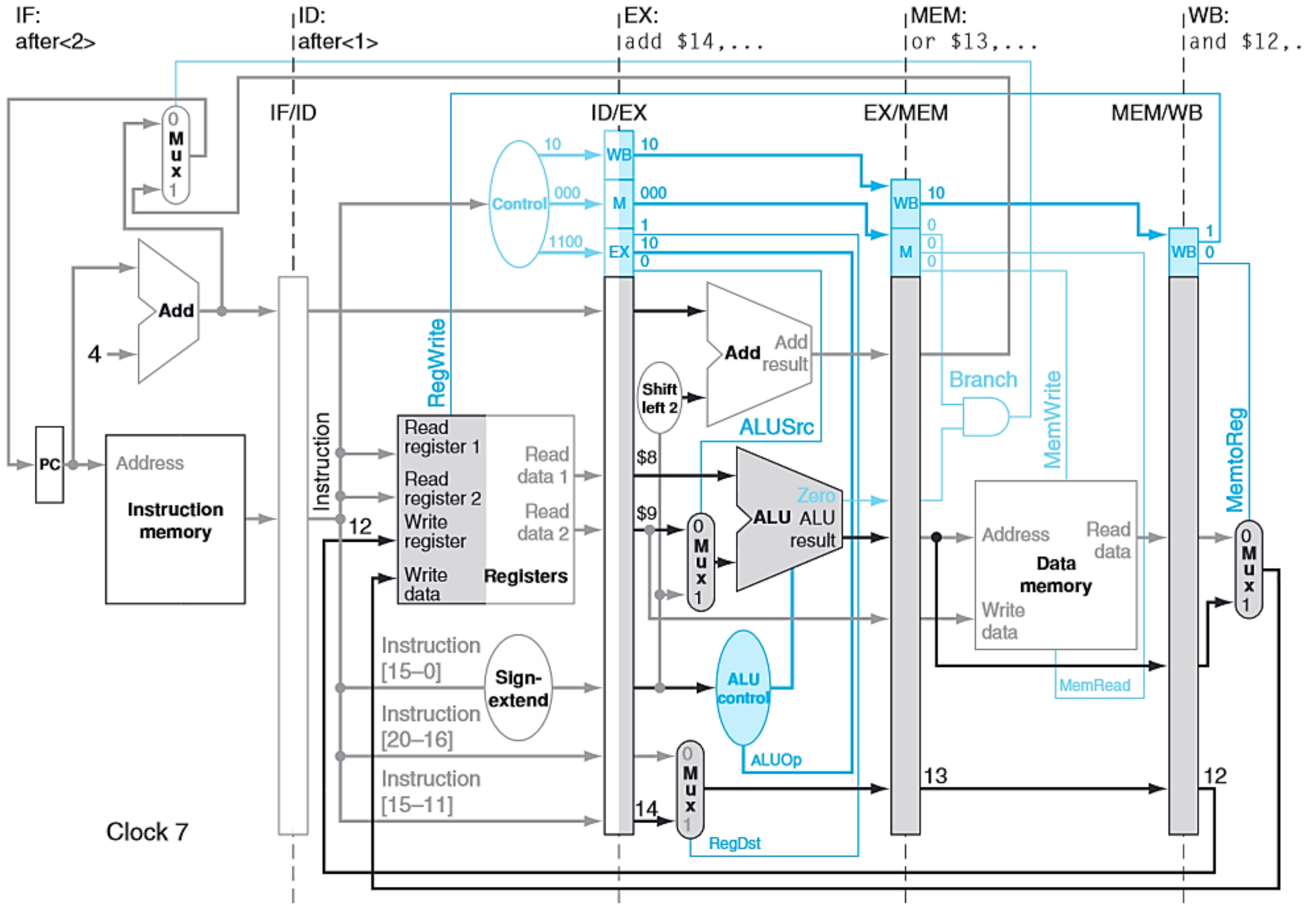

IF:
sub $11,$2,$3
ID:
lw $10,20($1)
EX:
before<1>
MEM:
before<2>
WB:
before<3>
IF/ID
ID/EX
EX/MEM
MEM/WB
Control
lw
11
010
0001
WB
M
EX
00
000
0
00
0
Mux
Add
4
PC
Address
Instruction memory
Instruction
RegWrite
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
$1
$X
1
X
Shift left 2
Add result
ALUSrc
Zero
ALU
ALU result
ALU control
ALUOp
Branch
MemWrite
Address
Read data
Data memory
Write data
MemRead
MemtoReg
Sign-extend
Instruction [15–0]
20
Instruction [20–16]
10
Instruction [15–11]
X
RegDst
Clock 2

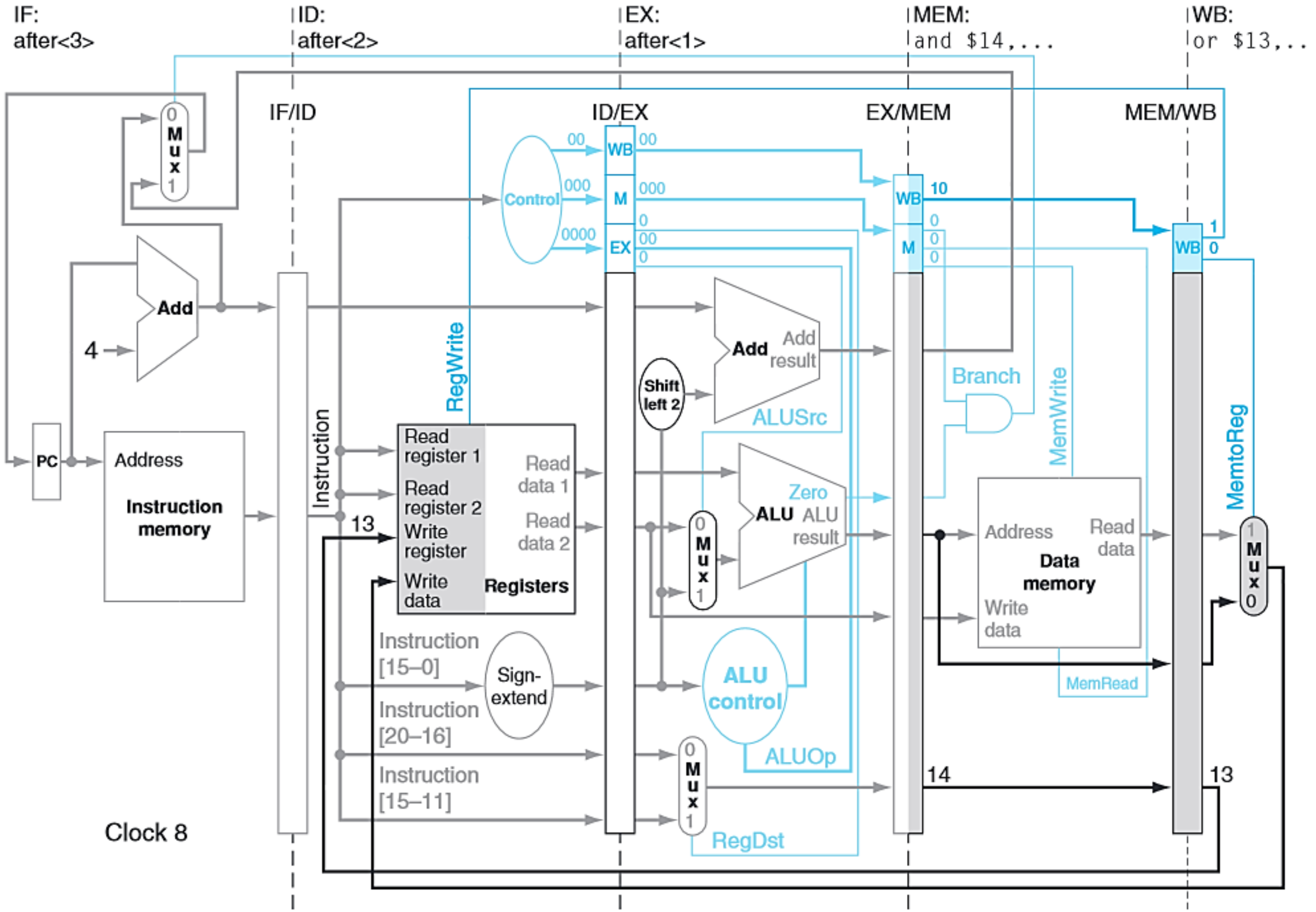

IF:
and $12,$4,$5
ID:
sub $11,$2,$3
EX:
lw $10,...
MEM:
before<1>
WB:
before<2>
IF/ID
ID/EX
EX/MEM
MEM/WB
Control
WB
M
EX
00
000
0000
11
010
0
00
1
0
RegWrite
Add
4
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
2
3
$2
$3
$1
Shift left 2
Add
Add result
ALUSrc
Zero
ALU
ALU result
Mux
Branch
MemWrite
Address
Read data
Data memory
Write data
MemRead
MemtoReg
Instruction [15–0]
Sign-extend
Instruction [20–16]
Instruction [15–11]
X
11
20
10
ALU control
ALUOp
RegDst
Clock 3

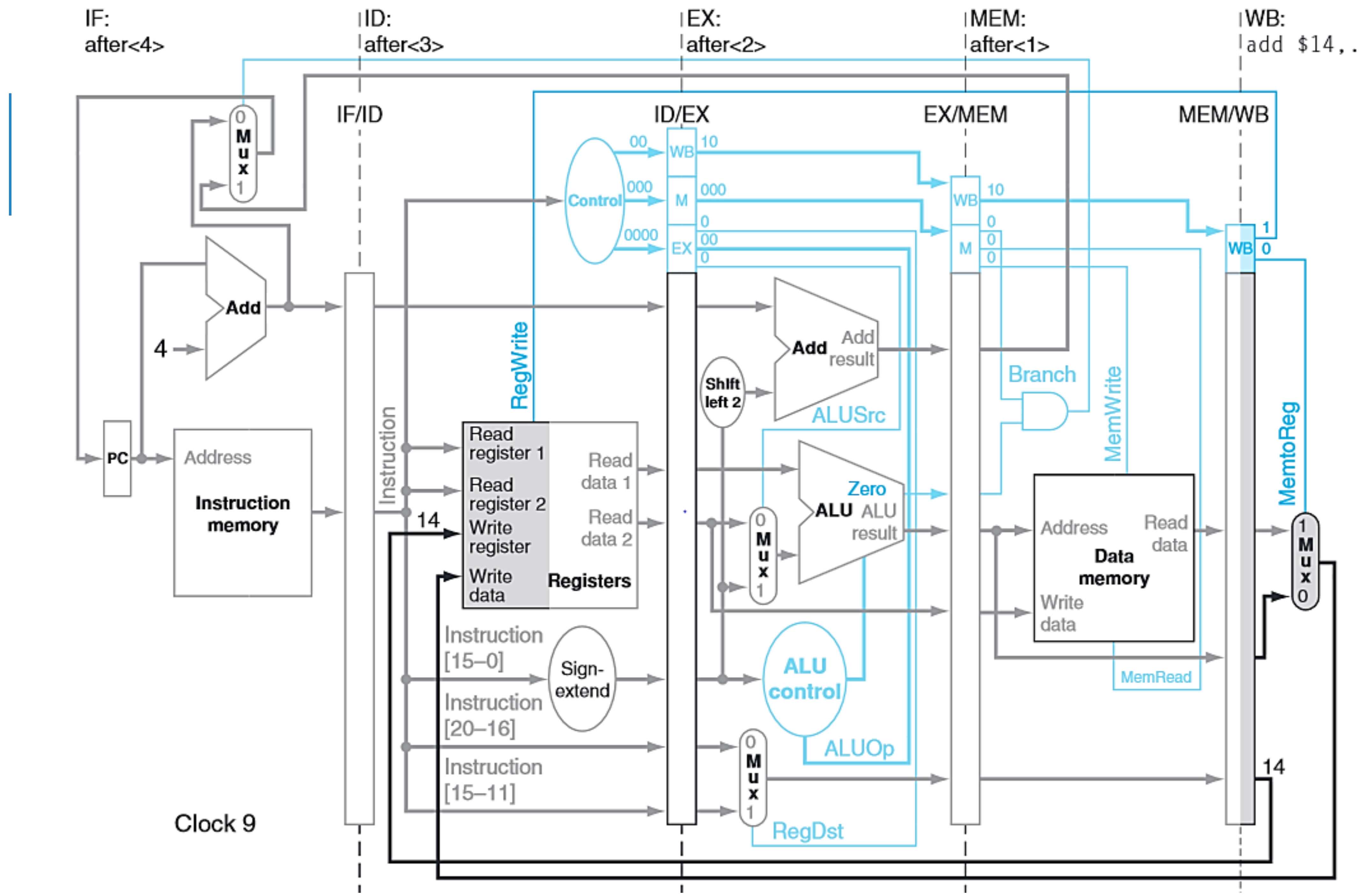

IF: or $13,$6,$7
ID: and $12,$4,$5
EX: sub $11,...
MEM: lw $10,...
WB: before<1>
IF/ID
ID/EX
EX/MEM
MEM/WB
Mux
Add
4
PC
Address
Instruction memory
Instruction
and
Control
10
000
1100
WB
M
EX
11
0
1
RegWrite
4
5
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
$4
$5
$2
$3
Shift left 2
Add
Add result
ALUSrc
Zero
ALU
ALU result
ALU control
ALUOp
Instruction [15–0]
Sign-extend
Instruction [20–16]
Instruction [15–11]
X
12
11
RegDst
Branch
MemWrite
Address
Read data
Data memory
Write data
MemRead
MemtoReg
Clock 4


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
Control
WB
M
EX
RegWrite
Branch
MemWrite
MemRead
MemtoReg
ALUSrc
ALUOp
RegDst
Zero
Add
Add result
Shift left 2
ALU
ALU result
ALU control
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
Sign-extend
Instruction [15–0]
Instruction [20–16]
Instruction [15–11]
Data memory
Read data
Mux
Clock 5


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
Mux
Add
4
PC
Address
Instruction memory
Instruction
add
Control
WB
M
EX
10
000
1100
1
0
RegWrite
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
8
9
11
$8
$9
$6
$7
Shift left 2
Add
Add result
ALUSrc
Zero
ALU
ALU result
ALU control
ALUOp
Instruction [15–0]
Instruction [20–16]
Instruction [15–11]
Sign-extend
X
12
14
13
RegDst
Branch
MemWrite
Address
Read data
Data memory
Write data
MemRead
MemtoReg
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
Mux
Add
4
PC
Address
Instruction memory
Instruction
Control
10
000
1100
WB
M
EX
RegWrite
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
12
Instruction [15–0]
Sign-extend
Instruction [20–16]
Instruction [15–11]
Shift left 2
Add
Add result
ALUSrc
$8
$9
ALU
ALU result
Zero
ALU control
ALUOp
14
RegDst
Branch
MemWrite
Address
Read data
Data memory
Write data
MemRead
13
MemtoReg
12
Clock 7


IF:
after<3>
ID:
after<2>
EX:
after<1>
MEM:
and $14,...
WB:
or $13,..
IF/ID
ID/EX
EX/MEM
MEM/WB
Mux
Add
4
PC
Address
Instruction memory
Instruction
Control
WB
M
EX
RegWrite
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
13
Instruction [15–0]
Sign-extend
Instruction [20–16]
Instruction [15–11]
Shift left 2
Add result
ALUSrc
Zero
ALU
ALU result
ALU control
ALUOp
RegDst
Branch
MemWrite
Data memory
Read data
Write data
MemRead
MemtoReg
14
Clock 8


IF:
after<4>
ID:
after<3>
EX:
after<2>
MEM:
after<1>
WB:
add $14,.
IF/ID
ID/EX
EX/MEM
MEM/WB
Control
WB
M
EX
00
000
0000
10
000
0
00
0
10
0
0
0
1
0
Mux
Add
4
PC
Address
Instruction memory
Instruction
RegWrite
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
14
Instruction [15–0]
Instruction [20–16]
Instruction [15–11]
Sign-extend
Shift left 2
Add
Add result
ALUSrc
Zero
ALU
ALU result
ALU control
ALUOp
RegDst
Branch
MemWrite
Address
Read data
Data memory
Write data
MemRead
MemtoReg
14
Clock 9