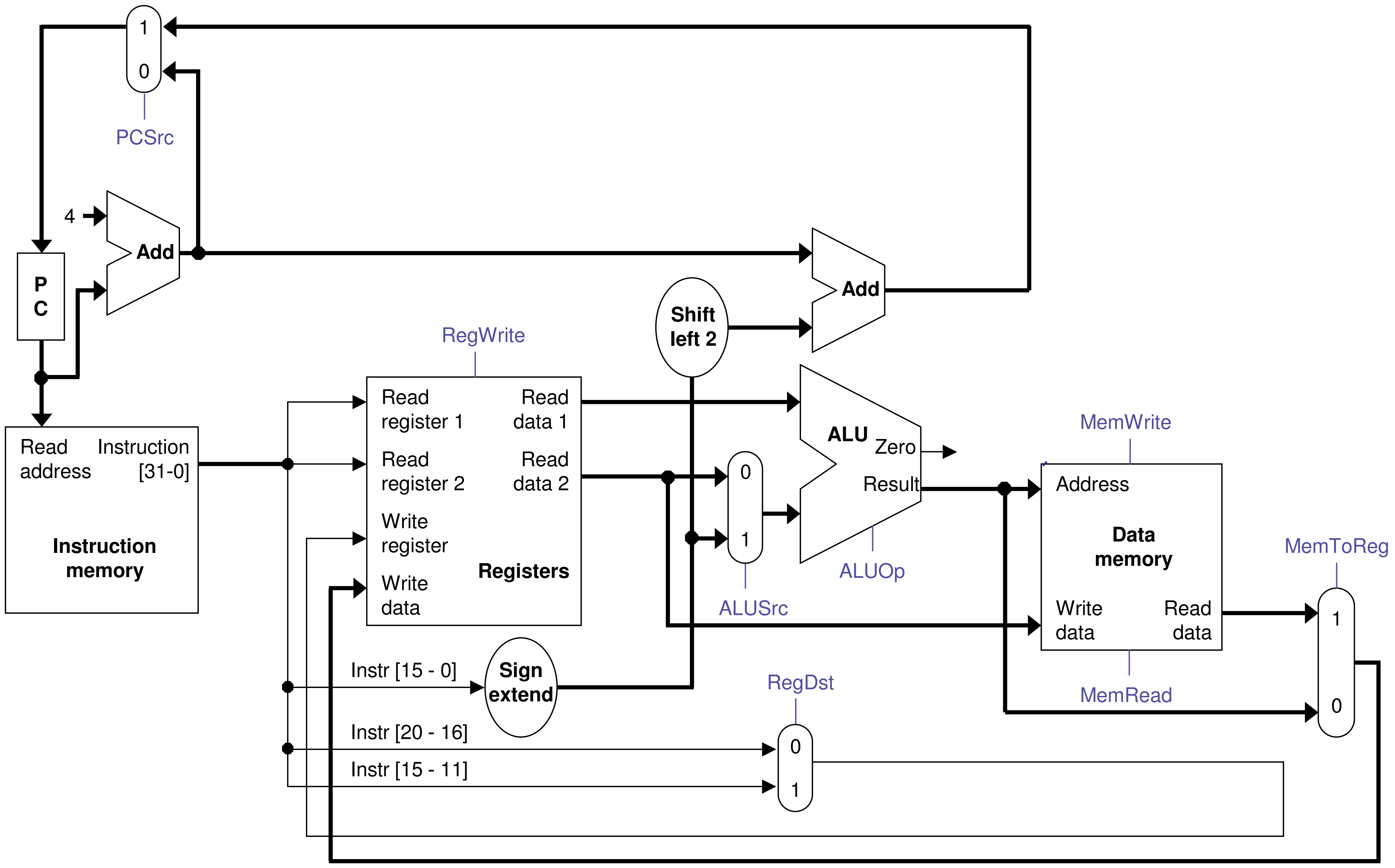
Single-cycle datapath, slightly rearranged



What’s been changed?

- Almost nothing! This is equivalent to the original single-cycle datapath.
  - There are separate memories for instructions and data.
  - There are two adders for PC-based computations and one ALU.
  - The control signals are the same.

- Only some cosmetic changes were made to make the diagram smaller.
  - A few labels are missing, and the muxes are smaller.
  - The data memory has only one Address input. The actual memory operation can be determined from the MemRead and MemWrite control signals.

- The datapath components have also been moved around in preparation for adding *pipeline registers*.

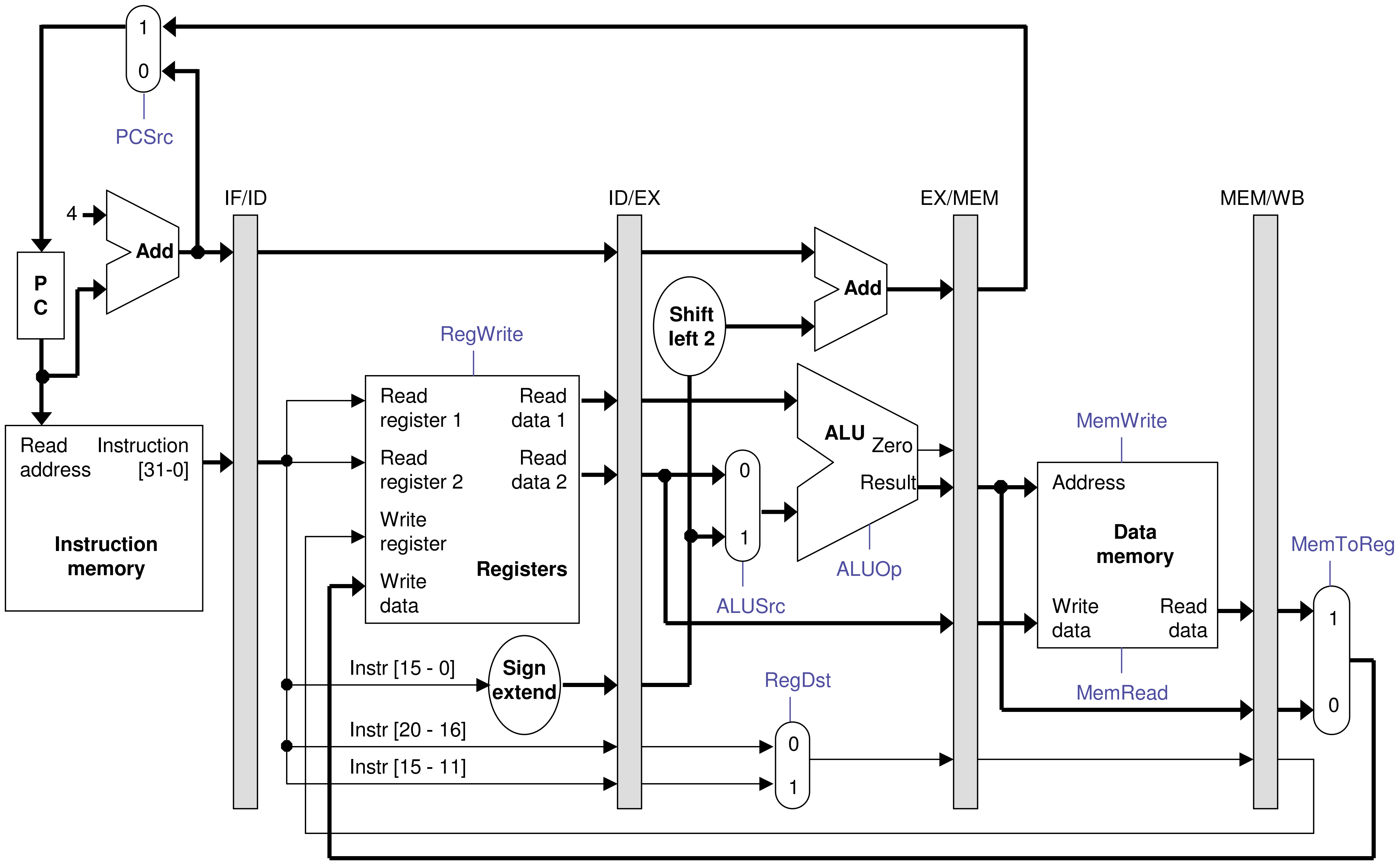
Pipeline registers

- In pipelining, we divide instruction execution into multiple cycles.
- Information computed during one cycle may be needed in a later cycle:
  - Instruction read in IF stage determines which registers are fetched in ID stage, what immediate is used for EX stage, and what destination register is for WB
  - Register values read in ID are used in EX and/or MEM stages
  - ALU output produced in EX is an effective address for MEM or a result for WB
- A lot of information to save!
  - Saved in intermediate registers called pipeline registers
- The registers are named for the stages they connect.

IF/ID   ID/EX   EX/MEM   MEM/WB

- No register is needed after the WB stage, because after WB the instruction is done

Pipelined datapath



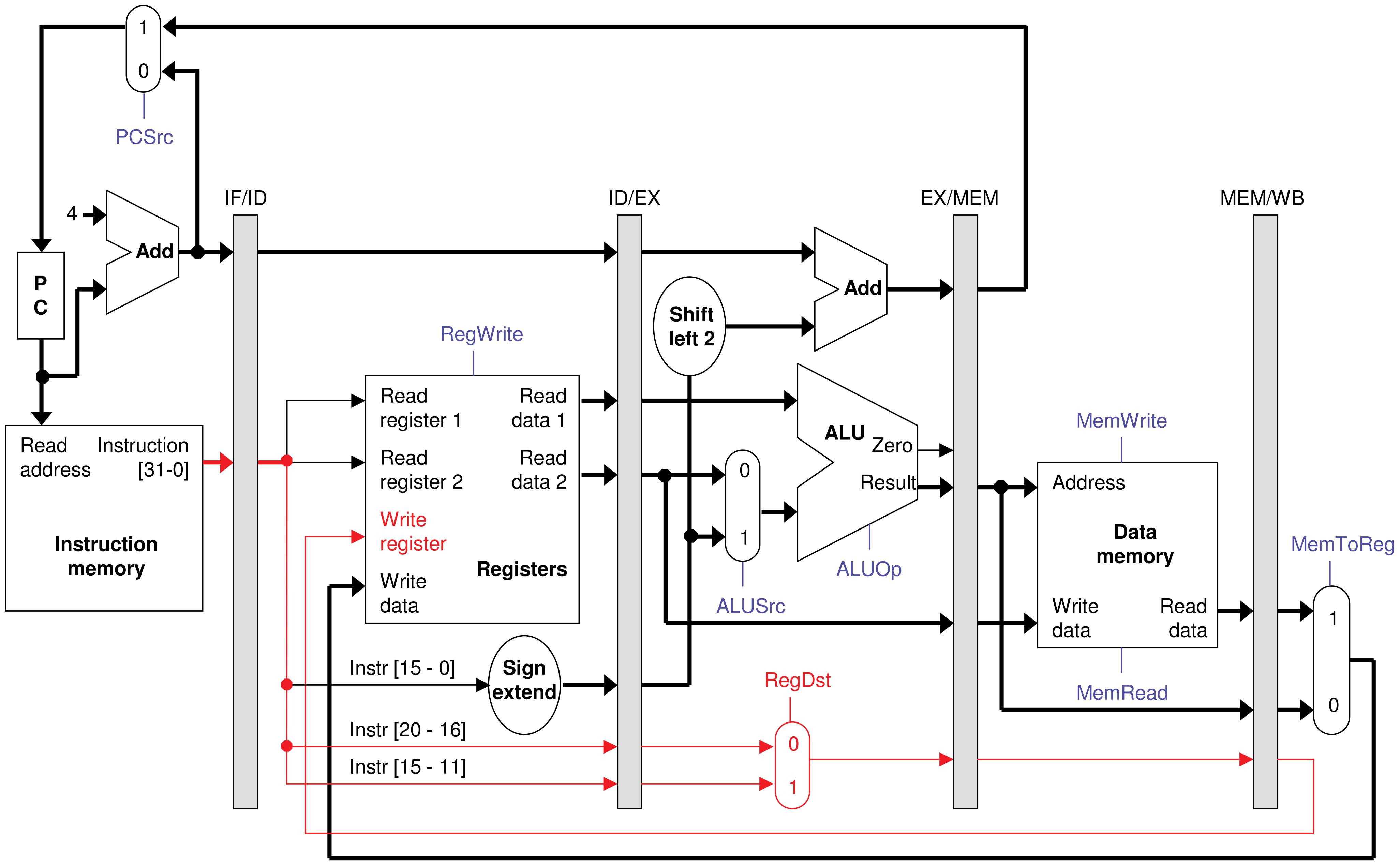
Propagating values forward

- Data values required *later* propagated through the pipeline registers

- The most extreme example is the destination register (`rd` or `rt`)
  - It is retrieved in IF, but isn't updated until the WB
  - Thus, it must be passed through *all* pipeline stages, as shown in red on the next slide

- Notice that we can't keep a single "instruction register," because the pipelined machine needs to fetch a new instruction every clock cycle

The destination register



What about control signals?

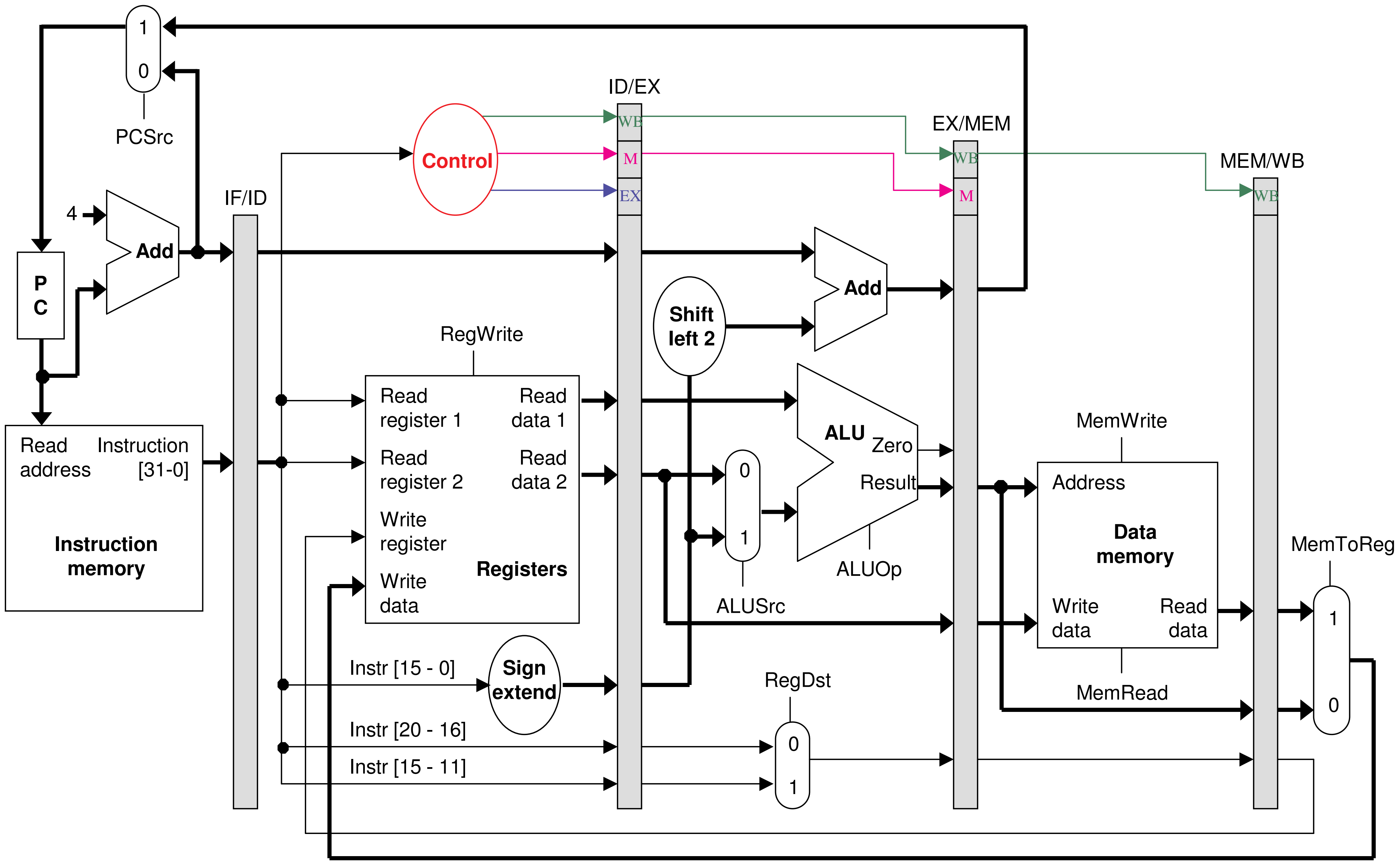
- Control signals generated similar to the single-cycle processor
  - in the ID stage, the processor decodes the instruction fetched in IF and produces the appropriate control values

- Some of the control signals will not be needed until later stages
  - These signals must be propagated through the pipeline until they reach the appropriate stage
  - We just pass them in the pipeline registers, along with the data

- Control signals can be categorized by the pipeline stage that uses them

Stage	Control signals needed		
EX	ALUSrc	ALUOp	RegDst
MEM	MemRead	MemWrite	PCSrc
WB	RegWrite	MemToReg	

Pipelined datapath and control



Notes about the diagram

- The control signals are grouped together in the pipeline registers, just to make the diagram a little clearer

- Not all of the registers have a write enable signal
  - the datapath fetches one instruction per cycle, so the PC must also be updated on each clock cycle – including a write enable for the PC would be redundant
  - similarly, the pipeline registers are also written on every cycle, so no explicit write signals are needed

An example execution sequence

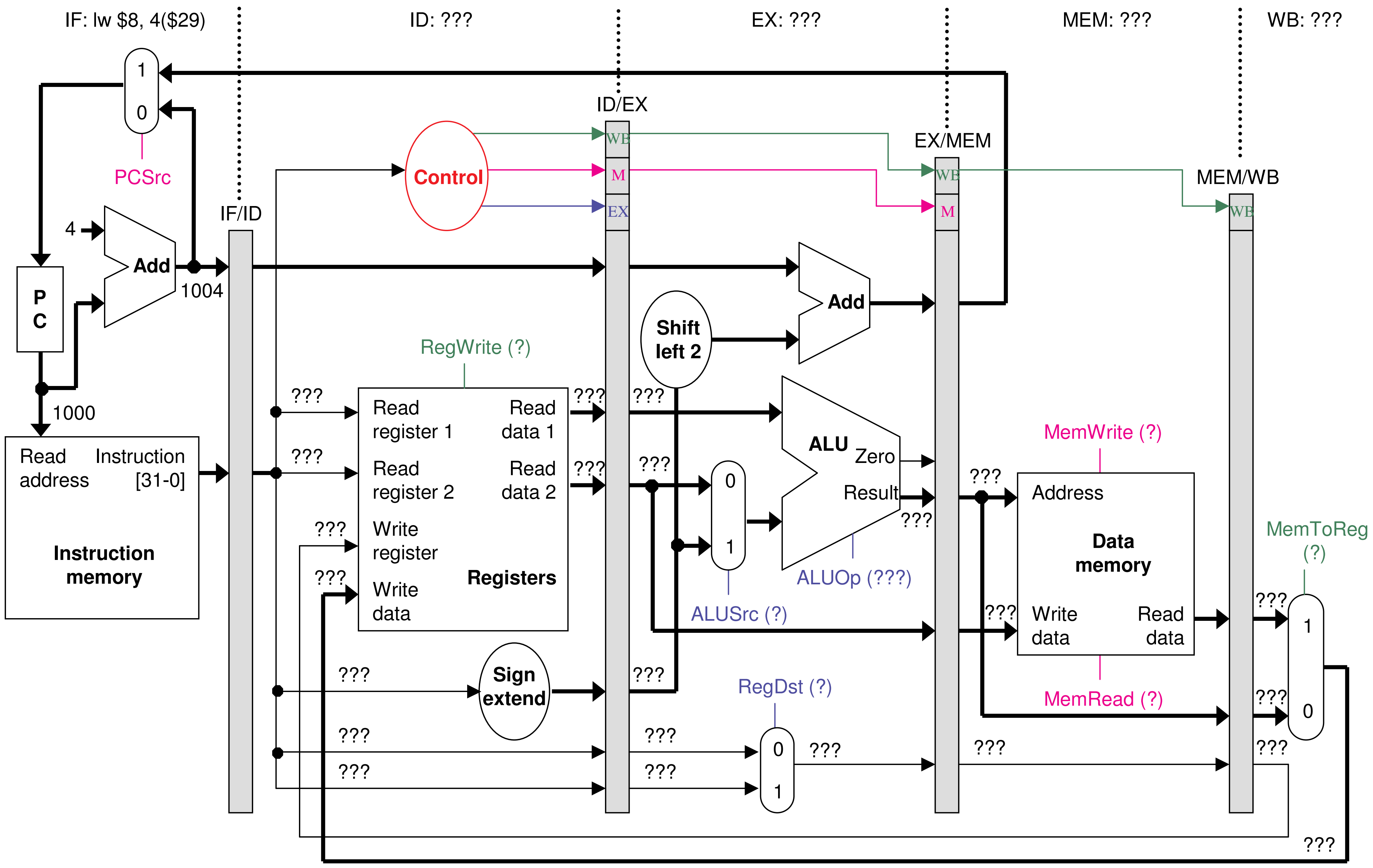
- Here's a sample sequence of instructions to execute.

addresses in decimal

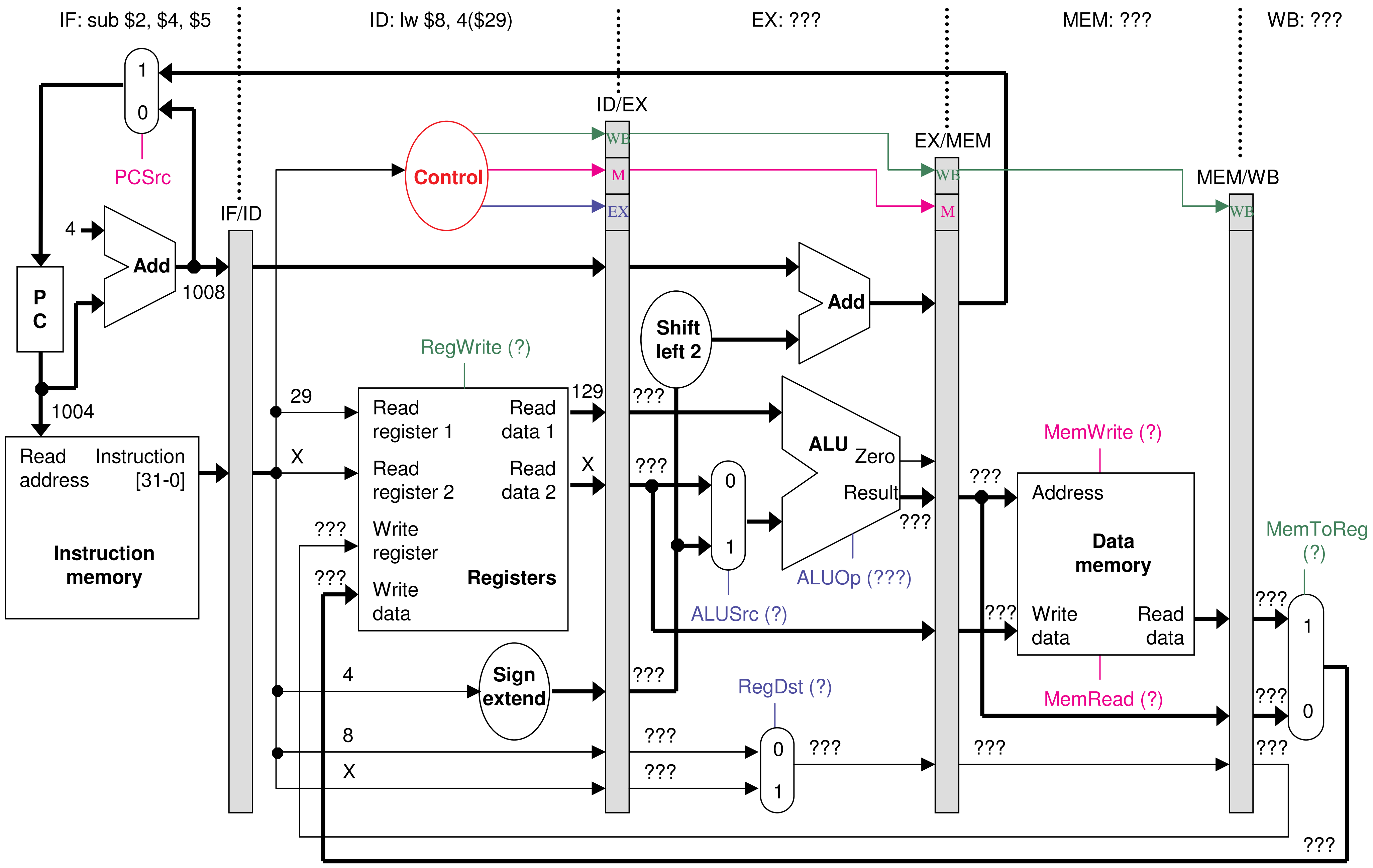
```
1000:  lw  $8, 4($29)
1004:  sub $2, $4, $5
1008:  and $9, $10, $11
1012:  or  $16, $17, $18
1016:  add $13, $14, $0
```

- We'll make some assumptions, just so we can show actual data values.
  - Each register contains its number plus 100. For instance, register $8 contains 108, register $29 contains 129, and so forth.
  - Every data memory location contains 99.
- Our pipeline diagrams will follow some conventions.
  - An X indicates values that aren't important, like the constant field of an R-type instruction.
  - Question marks ??? indicate values we don't know, usually resulting from instructions coming before and after the ones in our example.

Cycle 1 (filling)



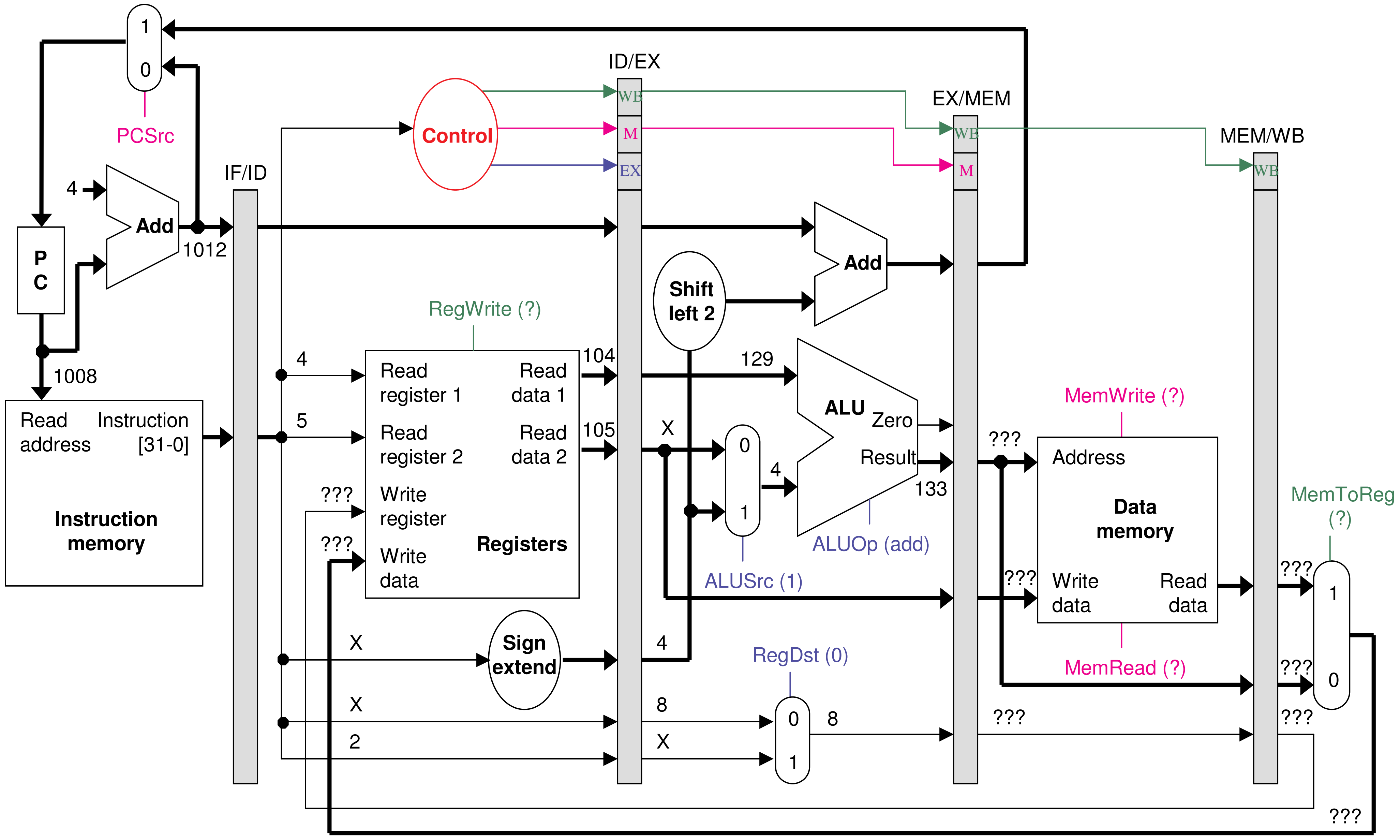
Cycle 2



IF: sub $2, $4, $5
ID: lw $8, 4($29)
EX: ???
MEM: ???
WB: ???
1
0
PCSrc
IF/ID
ID/EX
EX/MEM
MEM/WB
Control
WB
M
EX
WB
M
WB
4
Add
1008
P C
1004
Add
Shift left 2
RegWrite (?)
29
X
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
129
X
???
???
???
???
???
Read address
Instruction [31-0]
Instruction memory
ALU
Zero
Result
???
0
1
ALUSrc (?)
ALUOp (???)
MemWrite (?)
Address
Data memory
Write data
Read data
MemRead (?)
???
???
MemToReg (?)
???
???
???
1
0
4
Sign extend
???
8
X
???
???
RegDst (?)
0
1
???
???
???

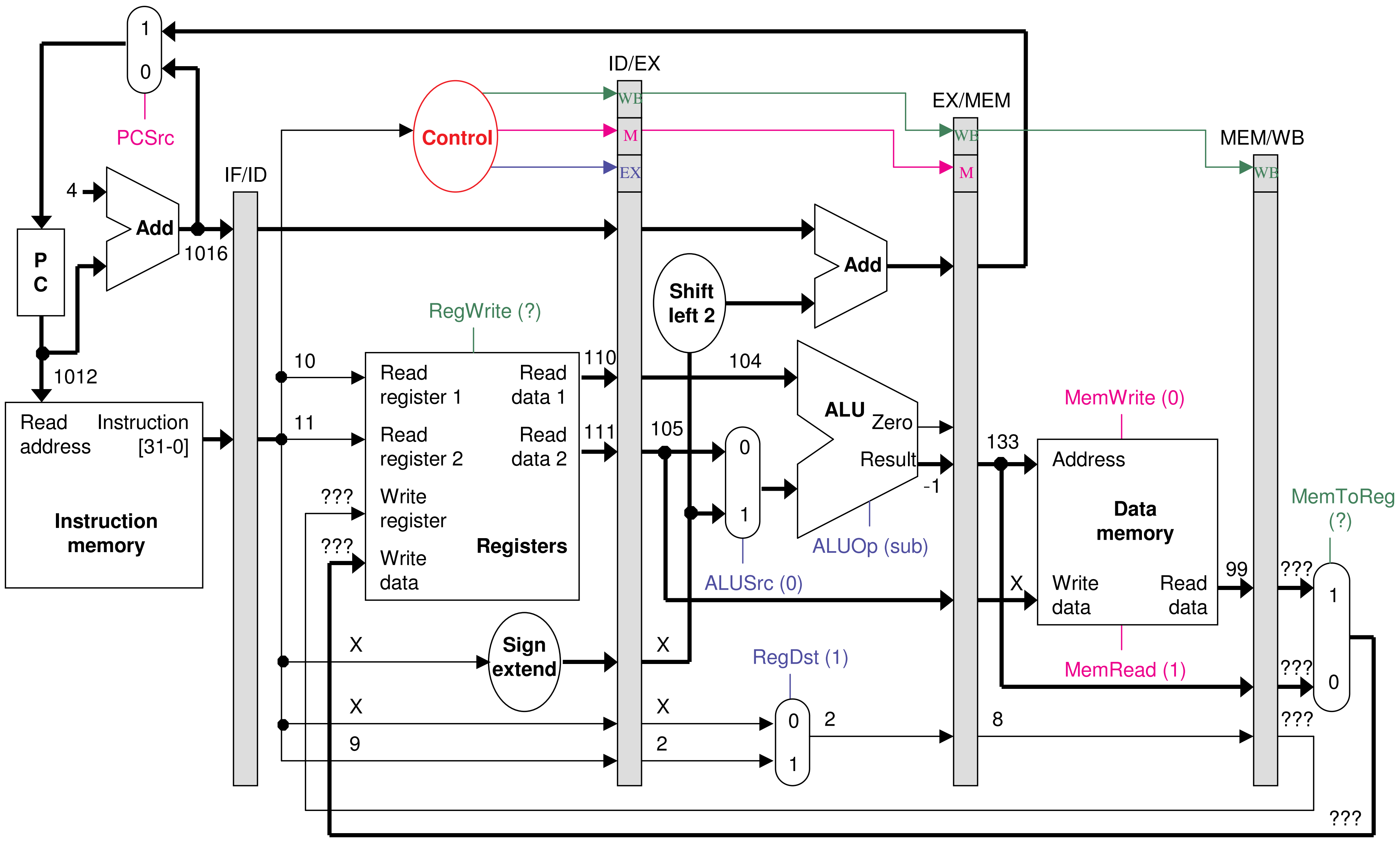
Cycle 3

IF: and $9, $10, $11   ID: sub $2, $4, $5   EX: lw $8, 4($29)   MEM: ???   WB: ???

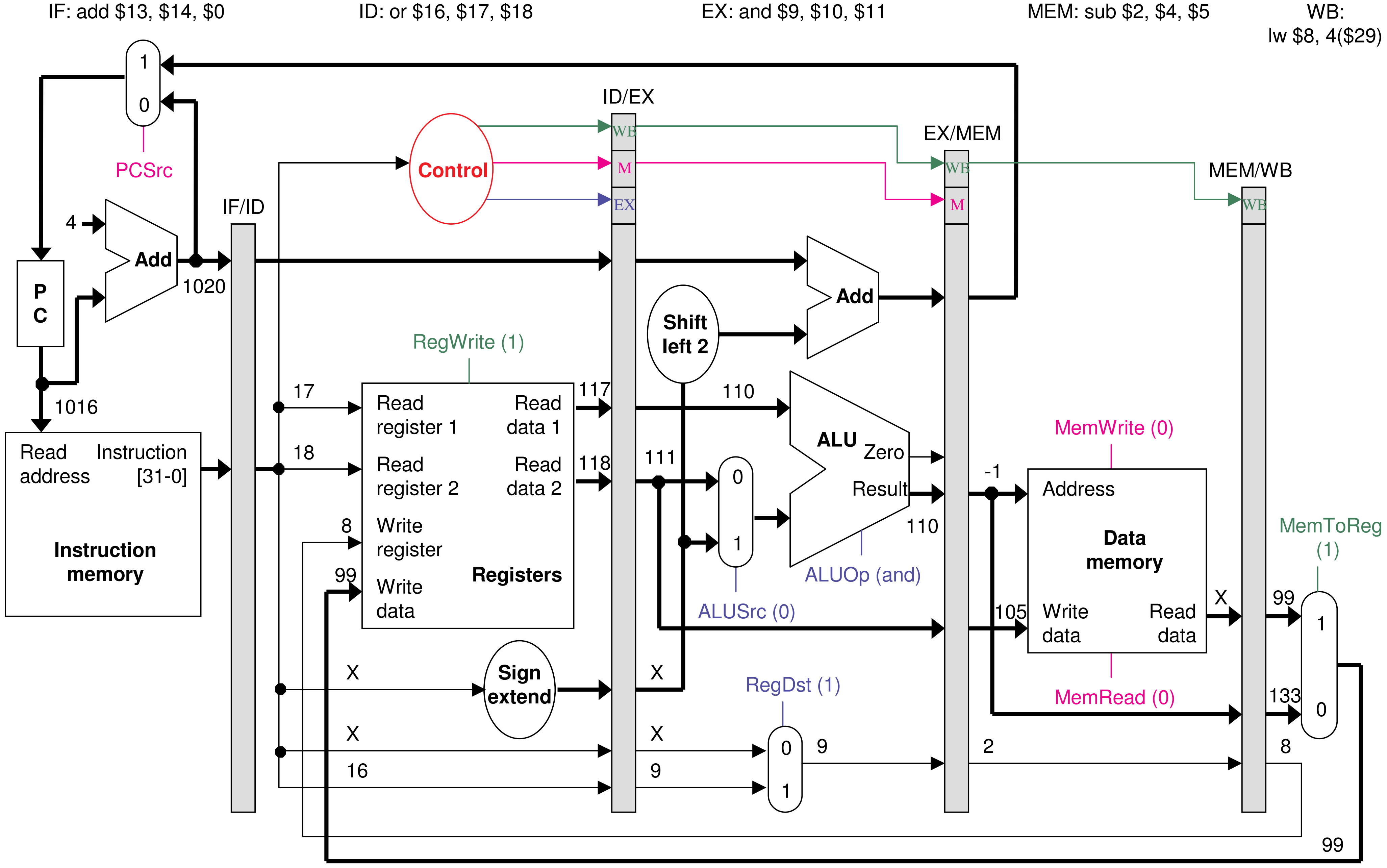


Cycle 4

IF: or $16, $17, $18 ID: and $9, $10, $11 EX: sub $2, $4, $5 MEM: lw $8, 4($29) WB: ???

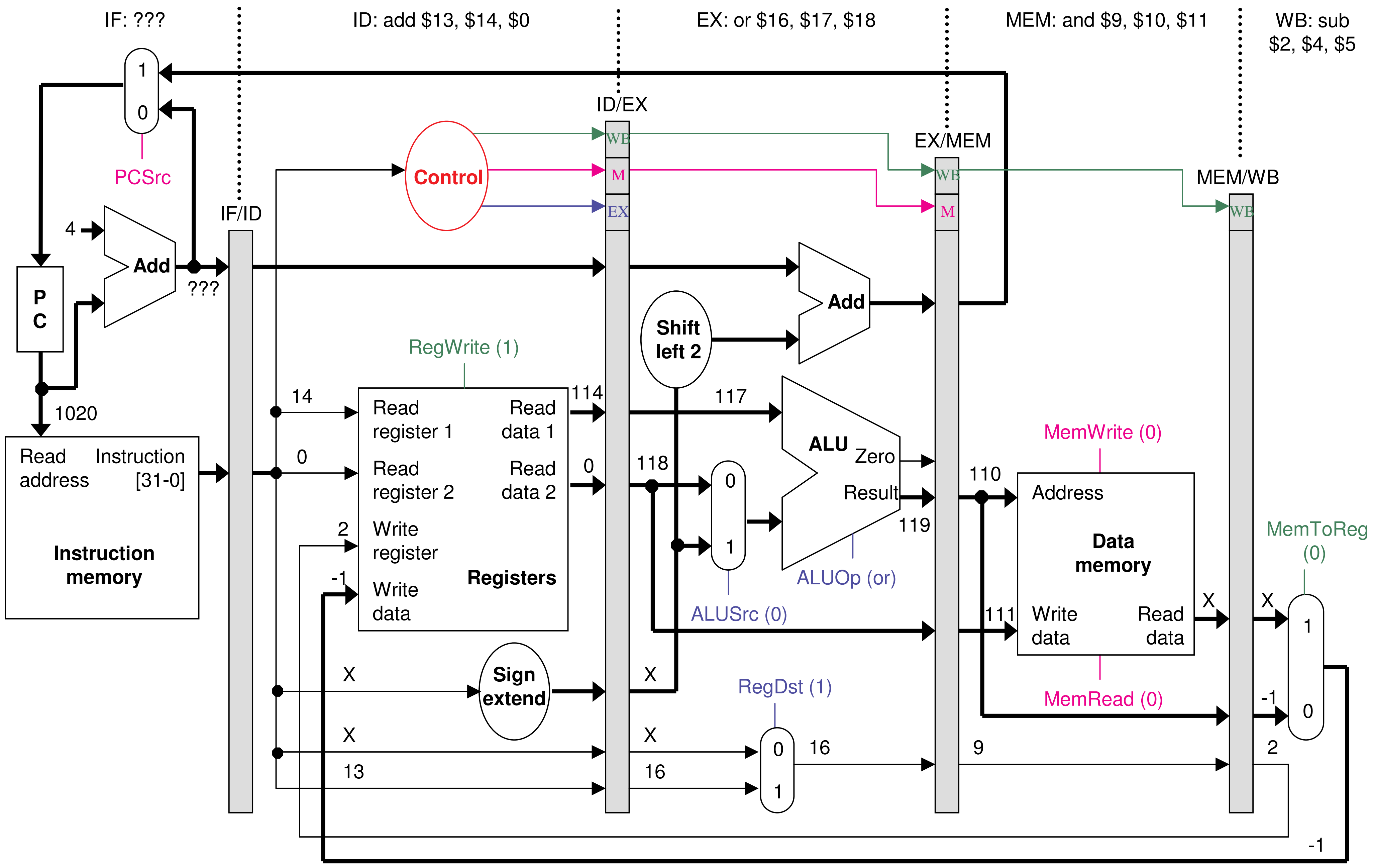


Cycle 5 (full)

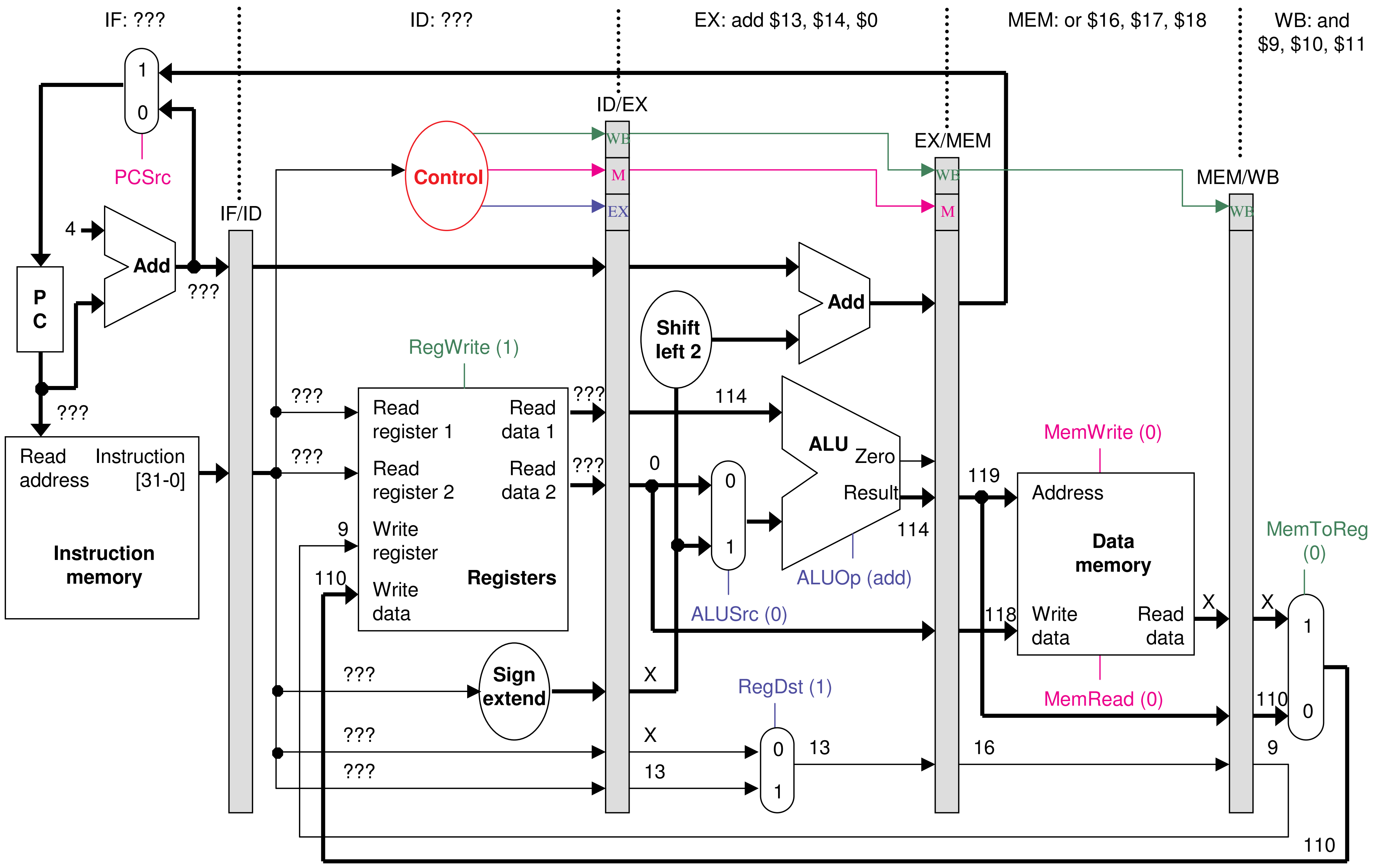


IF: add $13, $14, $0
ID: or $16, $17, $18
EX: and $9, $10, $11
MEM: sub $2, $4, $5
WB: lw $8, 4($29)
1
0
PCSrc
4
Add
1020
PC
1016
IF/ID
Control
ID/EX
WB
M
EX
EX/MEM
WB
M
MEM/WB
WB
Add
Shift left 2
RegWrite (1)
17
18
8
99
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
117
118
110
111
0
1
ALU
Zero
Result
110
ALUOp (and)
ALUSrc (0)
-1
MemWrite (0)
Address
Data memory
Write data
Read data
105
X
MemRead (0)
MemToReg (1)
99
1
133
0
8
99
Read address
Instruction [31-0]
Instruction memory
X
Sign extend
X
X
X
16
9
RegDst (1)
0
1
9
2

Cycle 6 (emptying)

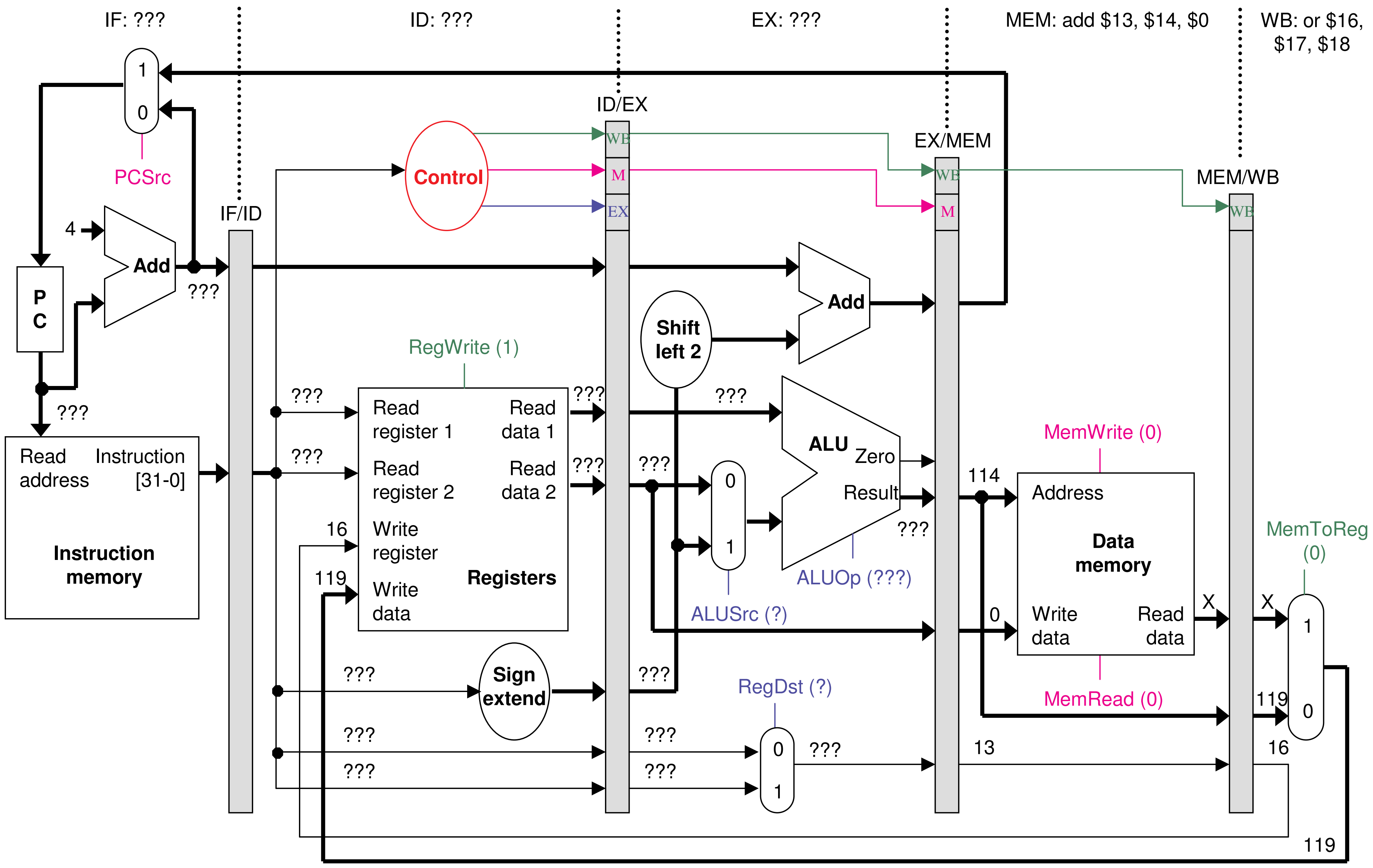


Cycle 7



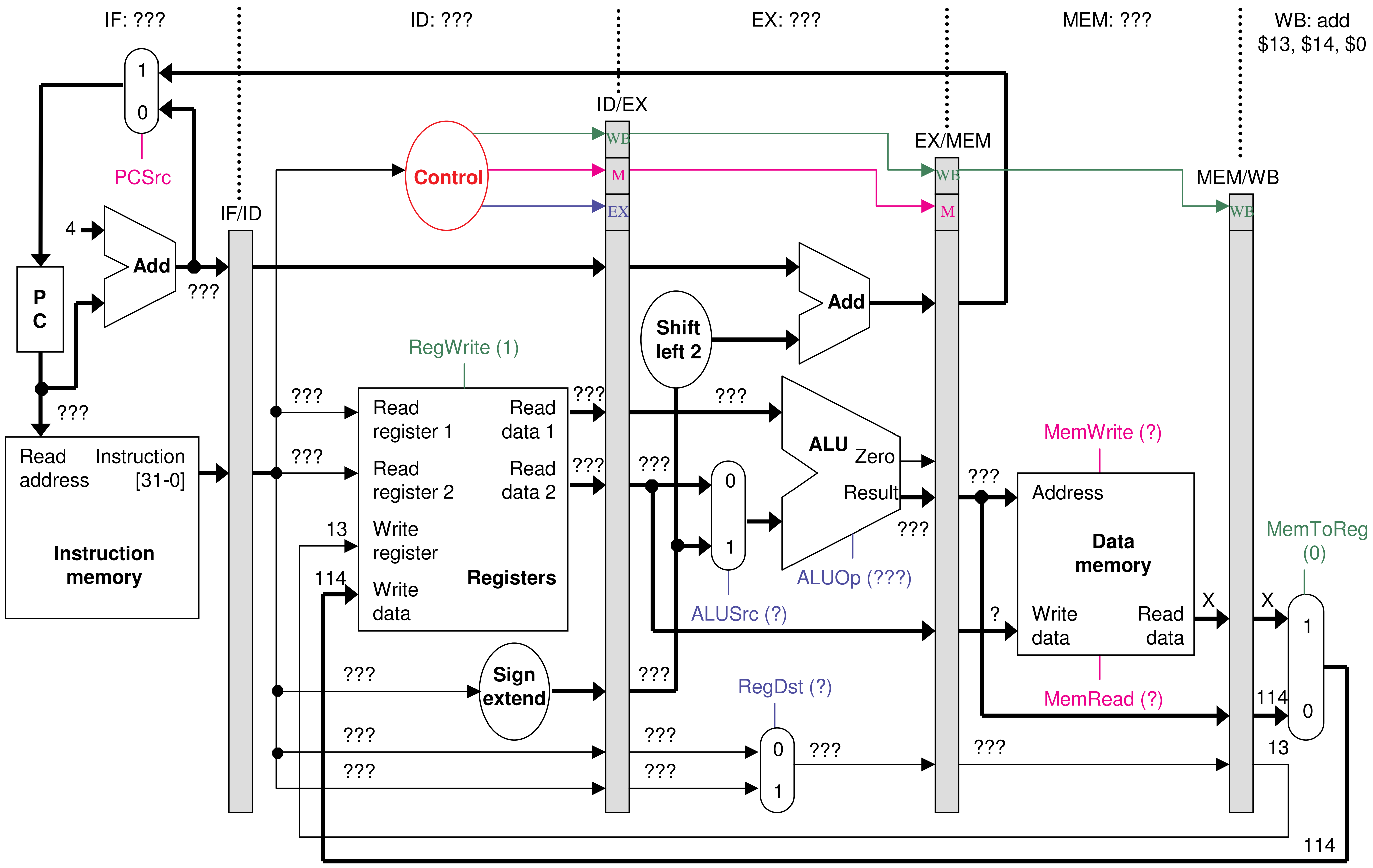
IF: ???
ID: ???
EX: add $13, $14, $0
MEM: or $16, $17, $18
WB: and $9, $10, $11
1
0
PCSrc
ID/EX
EX/MEM
MEM/WB
IF/ID
Control
WB
M
EX
WB
M
WB
4
Add
???
P C
Add
Shift left 2
RegWrite (1)
???
???
???
???
???
114
0
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
9
110
Read address
Instruction [31-0]
Instruction memory
ALU
Zero
Result
114
0
1
ALUSrc (0)
ALUOp (add)
MemWrite (0)
119
Address
Data memory
118
Write data
Read data
MemRead (0)
MemToReg (0)
X
X
1
0
110
???
Sign extend
X
???
X
???
13
RegDst (1)
0
1
13
16
9
110

Cycle 8



IF: ???
ID: ???
EX: ???
MEM: add $13, $14, $0
WB: or $16, $17, $18
1
0
PCSrc
IF/ID
ID/EX
EX/MEM
MEM/WB
Control
WB
M
EX
WB
M
WB
4
Add
???
P C
???
Read address
Instruction [31-0]
Instruction memory
Add
Shift left 2
RegWrite (1)
???
???
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
16
119
???
???
???
???
???
0
1
ALUSrc (?)
ALU
Zero
Result
???
ALUOp (???)
114
0
MemWrite (0)
Address
Data memory
Write data
Read data
MemRead (0)
X
X
MemToReg (0)
1
0
119
16
13
???
Sign extend
???
???
???
???
???
RegDst (?)
0
1
???
119

Cycle 9



IF: ???
ID: ???
EX: ???
MEM: ???
WB: add $13, $14, $0
ID/EX
EX/MEM
MEM/WB
IF/ID
PCSrc
1
0
Control
WB
M
EX
WB
M
WB
4
Add
???
P C
Add
Shift left 2
RegWrite (1)
???
???
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
13
114
???
???
???
???
ALU
Zero
Result
???
0
1
ALUSrc (?)
ALUOp (???)
Read address
Instruction [31-0]
Instruction memory
MemWrite (?)
Address
Data memory
Write data
Read data
MemRead (?)
???
?
X
X
MemToReg (0)
1
0
114
13
Sign extend
???
???
???
???
???
???
RegDst (?)
0
1
???
???
114

That’s a lot of diagrams there

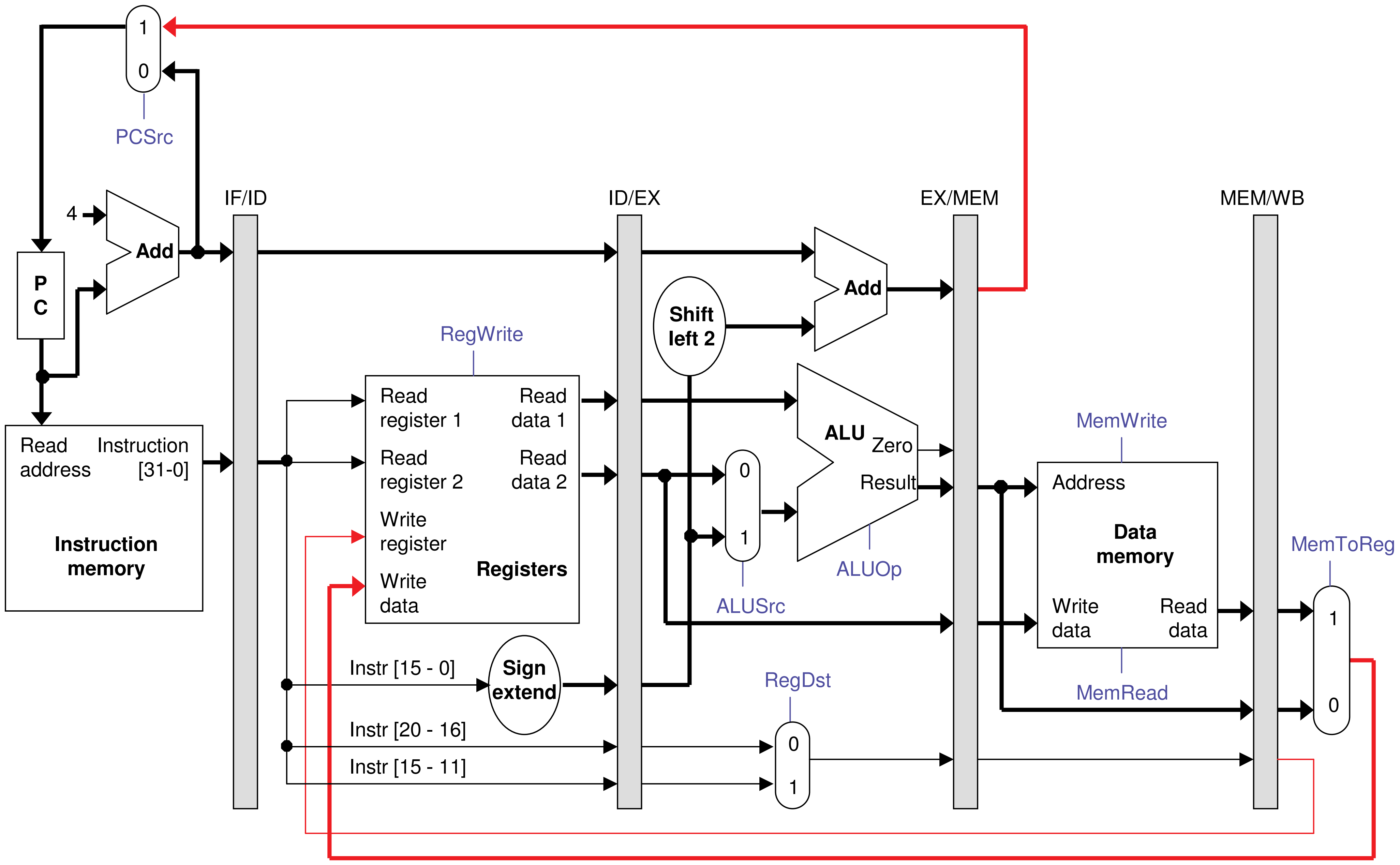
		Clock cycle								
		1	2	3	4	5	6	7	8	9
lw	$t0, 4($sp)	IF	ID	EX	MEM	WB				
sub	$v0, $a0, $a1		IF	ID	EX	MEM	WB			
and	$t1, $t2, $t3			IF	ID	EX	MEM	WB		
or	$s0, $s1, $s2				IF	ID	EX	MEM	WB	
add	$t5, $t6, $0					IF	ID	EX	MEM	WB

- Compare the last nine slides with the pipeline diagram above.
  - You can see how instruction executions are overlapped.
  - Each functional unit is used by a *different* instruction in each cycle.
  - The pipeline registers save control and data values generated in previous clock cycles for later use.
  - When the pipeline is full in clock cycle 5, all of the hardware units are utilized. This is the ideal situation, and what makes pipelined processors so fast.
- Try to understand this example or the similar one in the book at the end of Section 6.3.

Instruction set architectures and pipelining

- The MIPS instruction set was designed especially for easy pipelining.
  - All instructions are 32-bits long, so the instruction fetch stage just needs to read one word on every clock cycle.
  - Fields are in the same position in different instruction formats—the opcode is always the first six bits, rs is the next five bits, etc. This makes things easy for the ID stage.
  - MIPS is a register-to-register architecture, so arithmetic operations cannot contain memory references. This keeps the pipeline shorter and simpler.
- Pipelining is harder for older, more complex instruction sets.
  - If different instructions had different lengths or formats, the fetch and decode stages would need extra time to determine the actual length of each instruction and the position of the fields.
  - With memory-to-memory instructions, additional pipeline stages may be needed to compute effective addresses and read memory *before* the EX stage.

Note how everything goes left to right, except ...



Our examples are too simple

- Here is the example instruction sequence used to illustrate pipelining on the previous page.

```
lw   $8, 4($29)
sub  $2, $4, $5
and  $9, $10, $11
or   $16, $17, $18
add  $13, $14, $0
```

- The instructions in this example are independent.
  - Each instruction reads and writes completely different registers.
  - Our datapath handles this sequence easily.
- But most sequences of instructions are *not* independent!

An example with dependencies

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

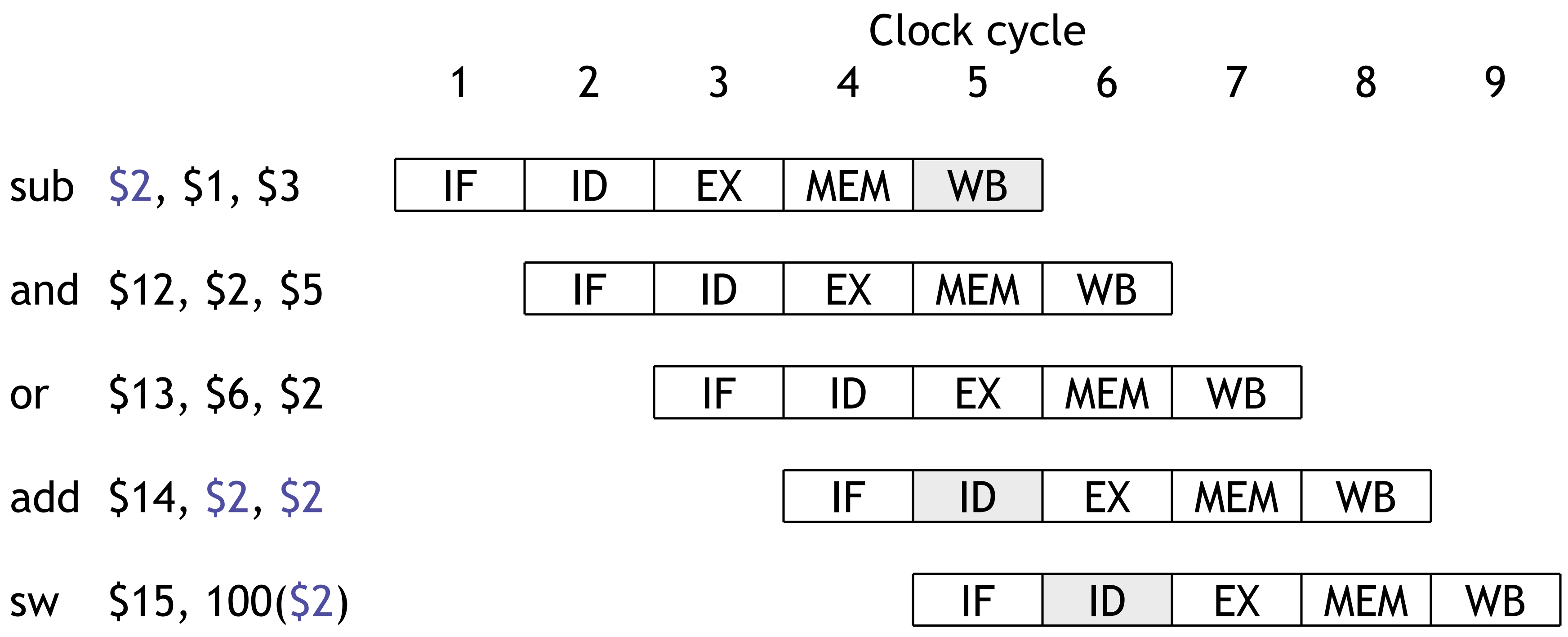
- There are several dependencies in this new code fragment.
  - The first instruction, SUB, stores a value into $2.
  - That register is used as a source in the rest of the instructions.
- This is not a problem for the single-cycle datapath.
  - Each instruction is executed completely before the next one begins.
  - This ensures that instructions 2 through 5 above use the new value of $2 (the sub result), just as we expect.
- How would this code sequence fare in our pipelined datapath?

Data hazards in the pipeline diagram

	Clock cycle								
	1	2	3	4	5	6	7	8	9
sub $2, $1, $3	IF	ID	EX	MEM	WB				
and $12, $2, $5		IF	ID	EX	MEM	WB			
or $13, $6, $2			IF	ID	EX	MEM	WB		
add $14, $2, $2				IF	ID	EX	MEM	WB	
sw $15, 100($2)					IF	ID	EX	MEM	WB

- The SUB instruction does not write to register $2 until clock cycle 5. This causes two data hazards in our current pipelined datapath.
  - The AND reads register $2 in cycle 3. Since SUB hasn't modified the register yet, this will be the *old* value of $2, not the new one.
  - Similarly, the OR instruction uses register $2 in cycle 4, again before it's actually updated by SUB.

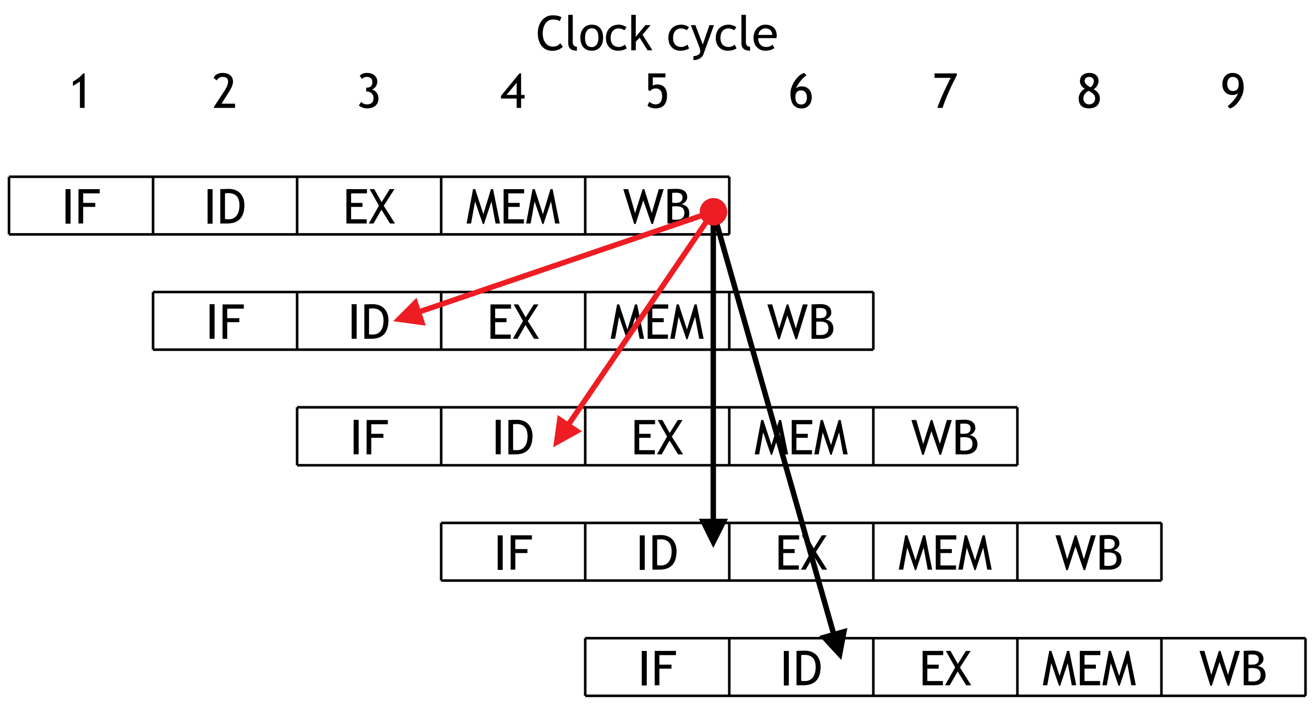
Things that are okay



- The ADD instruction is okay, because of the register file design.
  - Registers are written at the beginning of a clock cycle.
  - The new value will be available by the end of that cycle.
- The SW is no problem at all, since it reads $2 after the SUB finishes.

Dependency arrows

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



- Arrows indicate the flow of data between instructions.
  - The tails of the arrows show when register $2 is written.
  - The heads of the arrows show when $2 is read.
- Any arrow that points backwards in time represents a data hazard in our basic pipelined datapath. Here, hazards exist between instructions 1 & 2 and 1 & 3.

Summary

- The pipelined datapath extends the single-cycle processor that we saw earlier to improve instruction throughput.
  - Instruction execution is split into several stages.
  - Multiple instructions flow through the pipeline simultaneously.

- Pipeline registers propagate data and control values to later stages.

- The MIPS instruction set architecture supports pipelining with uniform instruction formats and simple addressing modes.

- Dependencies between instructions are called Hazards.

- We’ll look at hazards more carefully next week, and in MP 5.