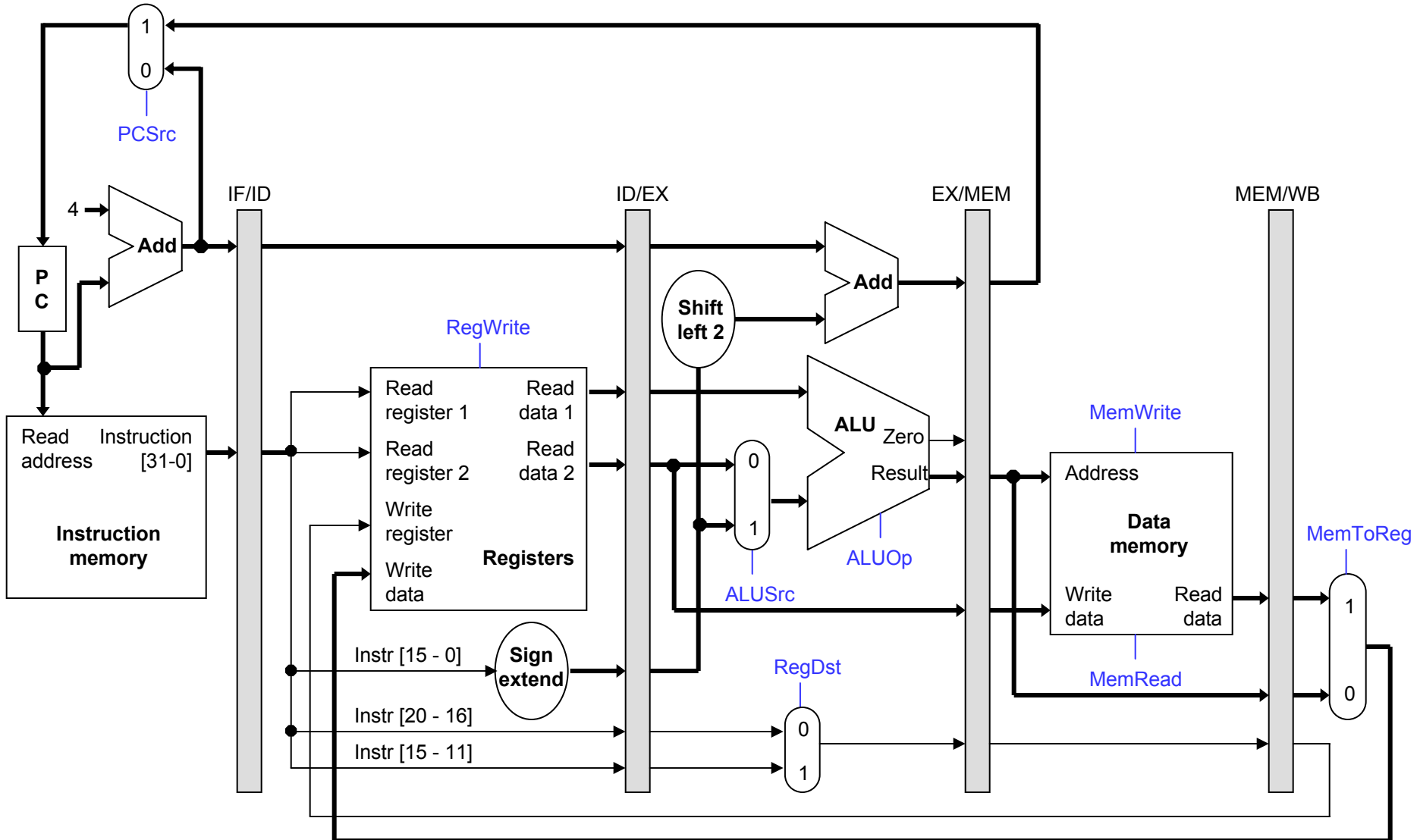


# Pipelined datapath

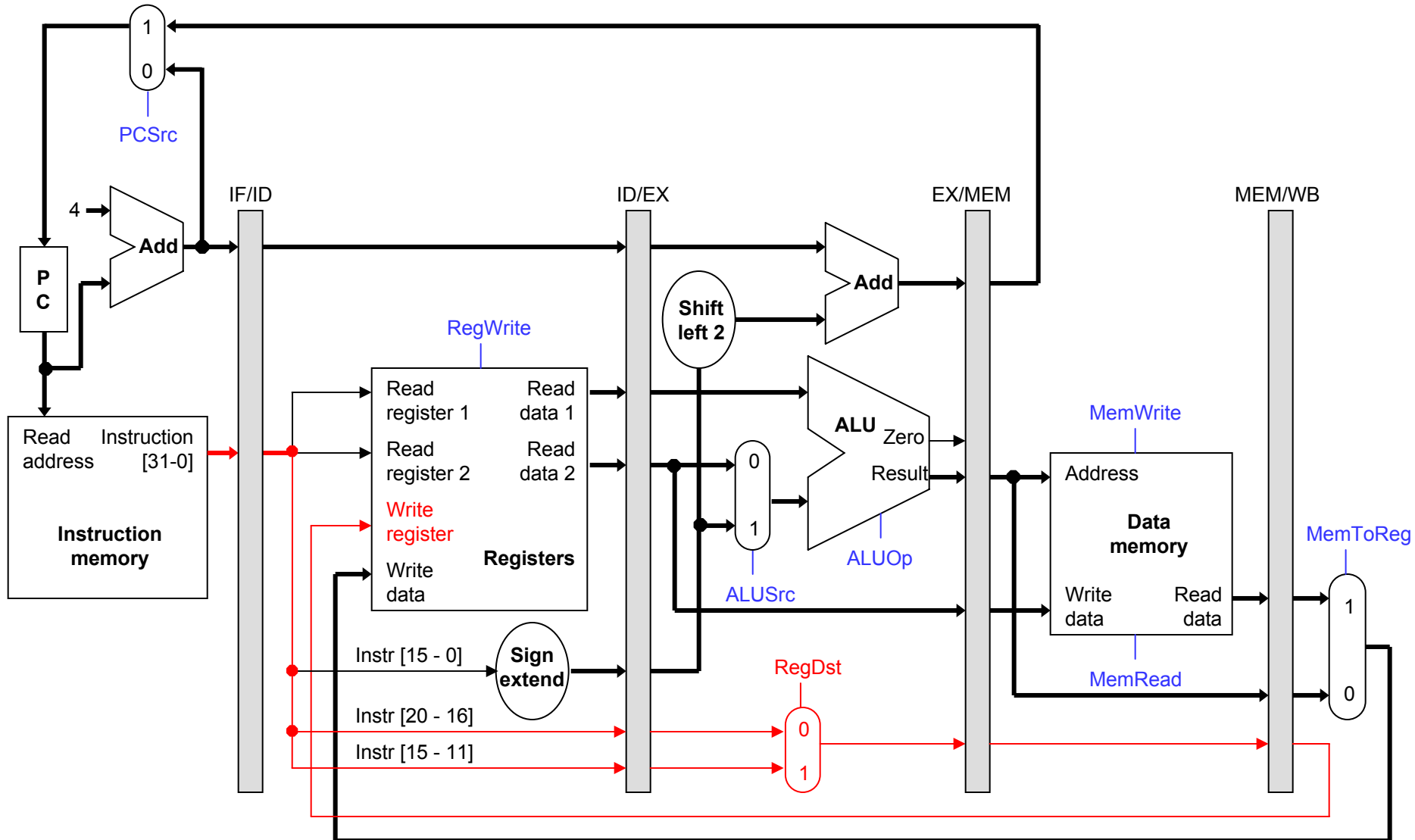


# Propagating values forward

---

- ❑ Any data values required in later stages must be propagated through the pipeline registers.
- ❑ The most extreme example is the **destination register**.
  - The rd field of the instruction word, retrieved in the first stage (IF), determines the destination register. But that register isn't updated until the *fifth* stage (WB).
  - Thus, the rd field must be passed through all of the pipeline stages, as shown in red on the next slide.

# The destination register

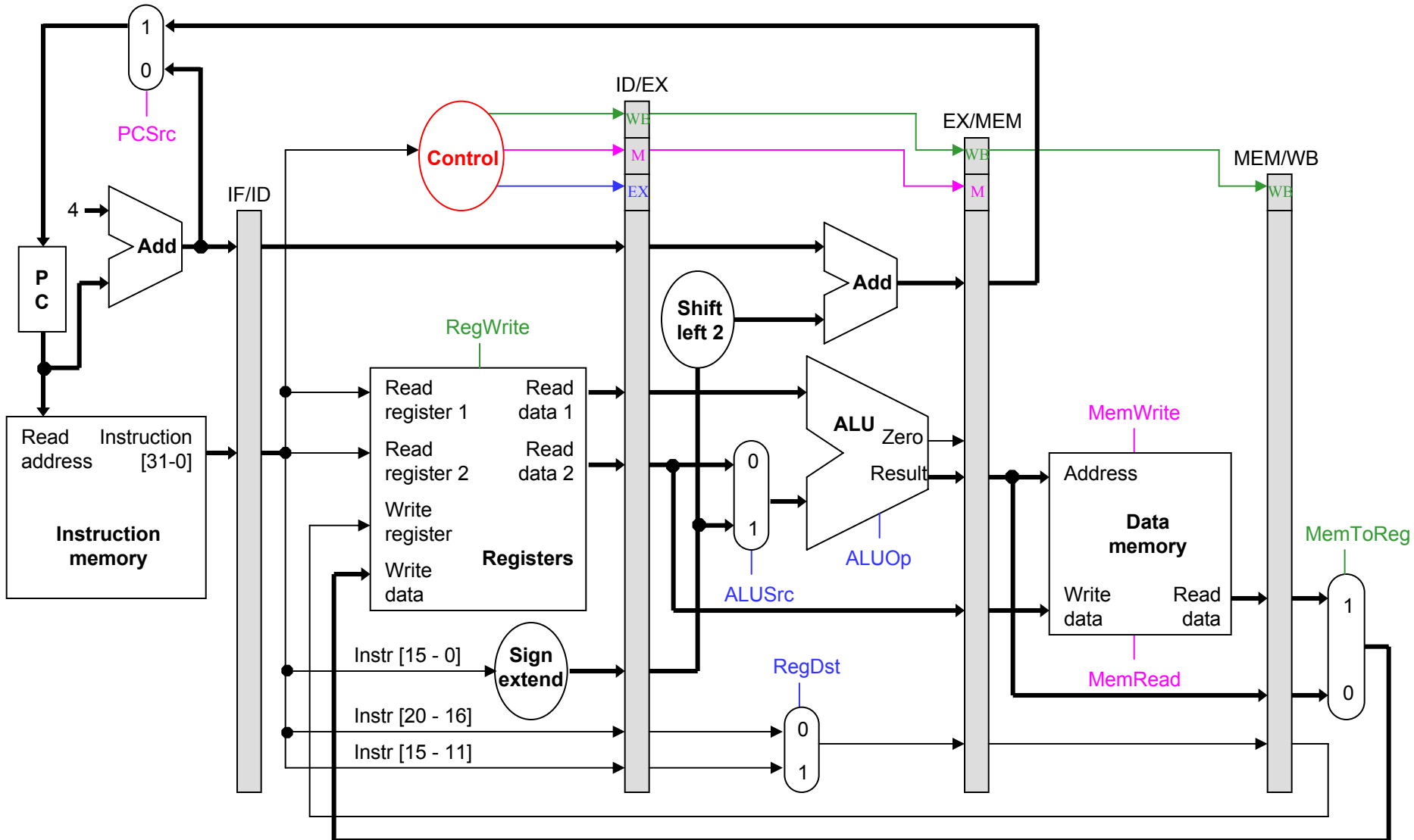


# What about control signals?

- ❑ The control signals are generated in the same way as in the single-cycle processor—after an instruction is fetched, the processor decodes it and produces the appropriate control values.
- ❑ But just like before, some of the control signals will not be needed until some later stage and clock cycle.
- ❑ These signals must be propagated through the pipeline until they reach the appropriate stage. We can just pass them in the pipeline registers, along with the other 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.**
  - **Because the datapath fetches one instruction per cycle, 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.

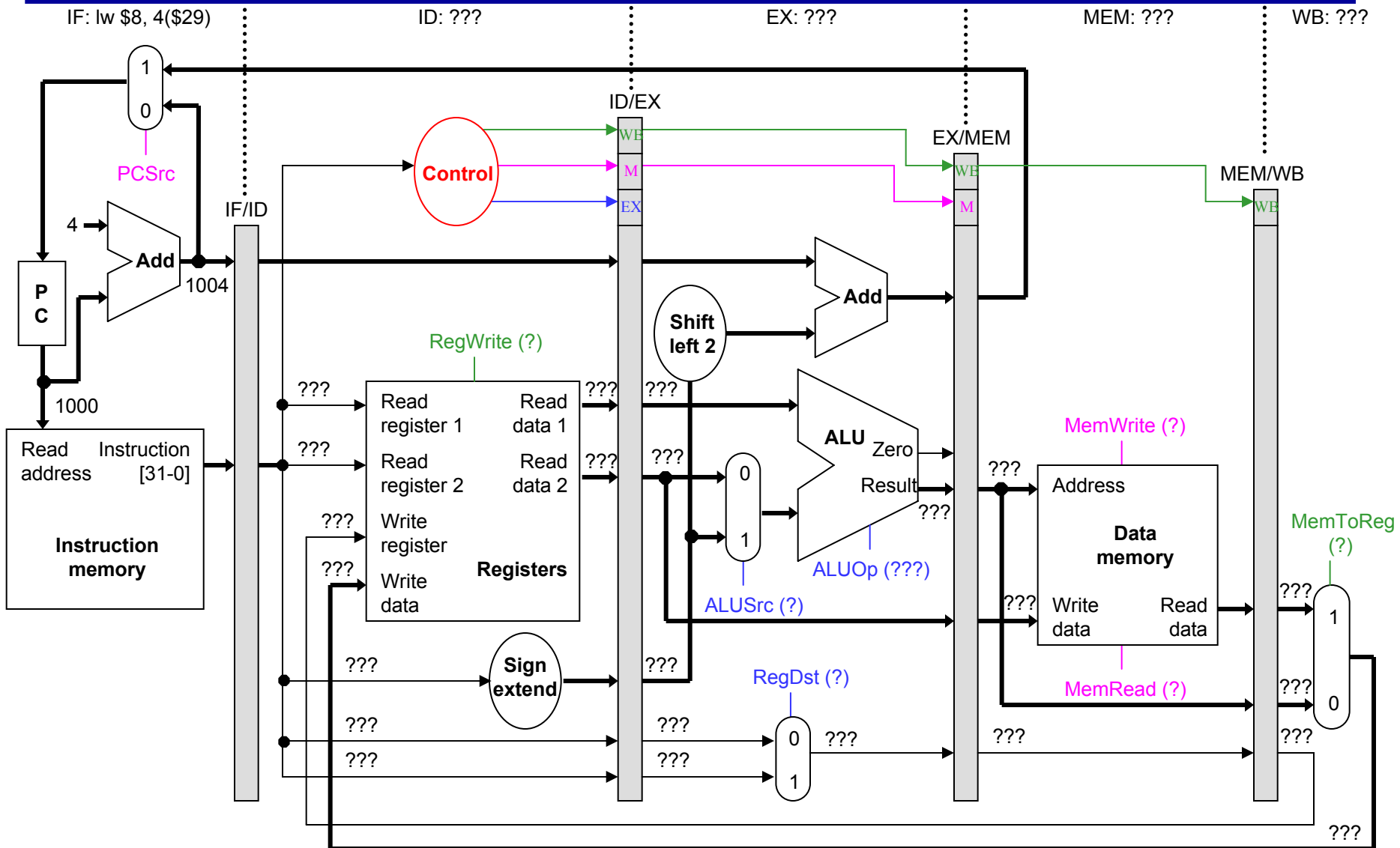
addresses  
in decimal



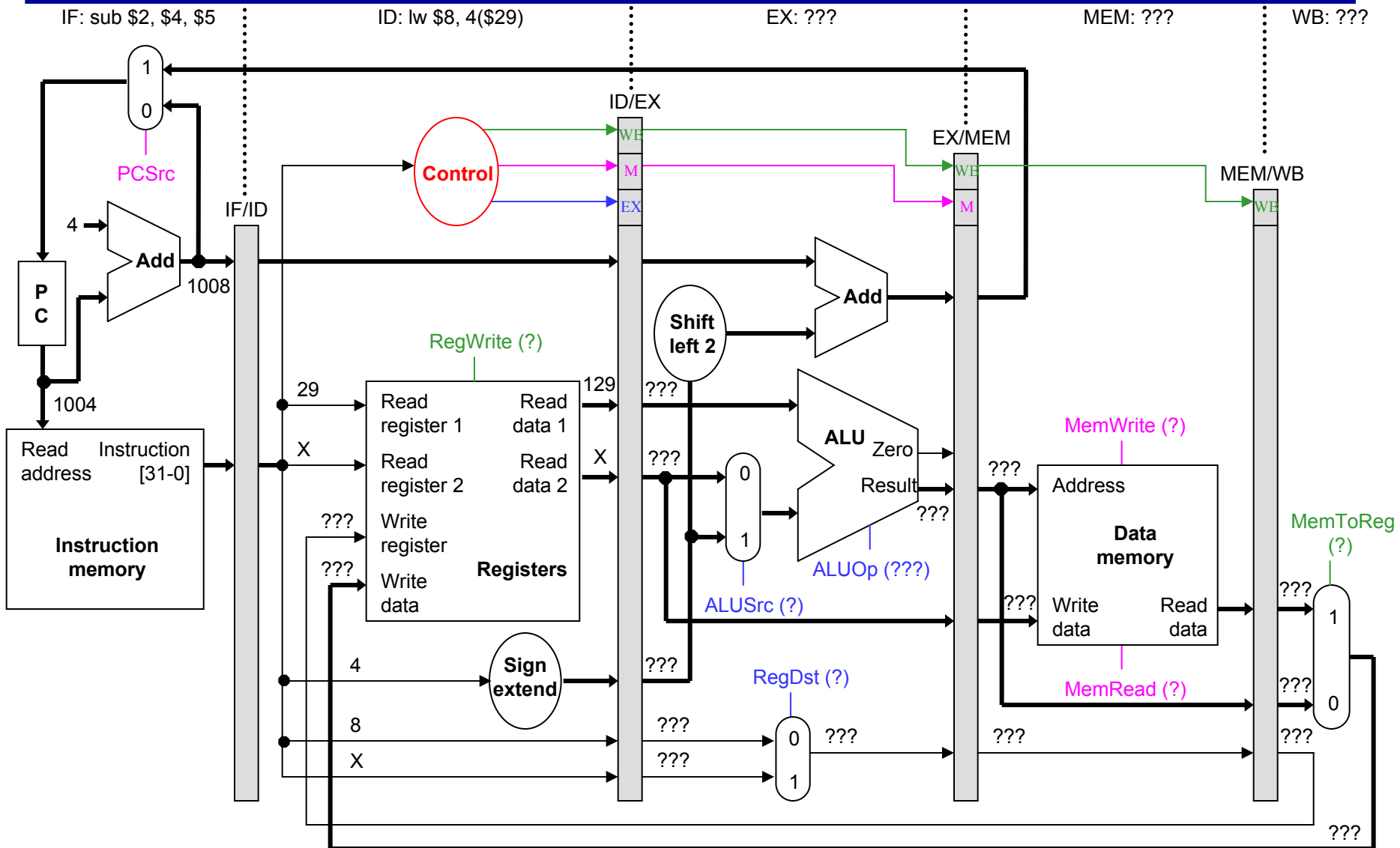
```
1000: lw $8, 4($29)
1004: sub $2, $4, $5
1008: and $9, $10, $11
1012: or $16, $17, $18
1016: add $13, $14, $0
```

- 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.

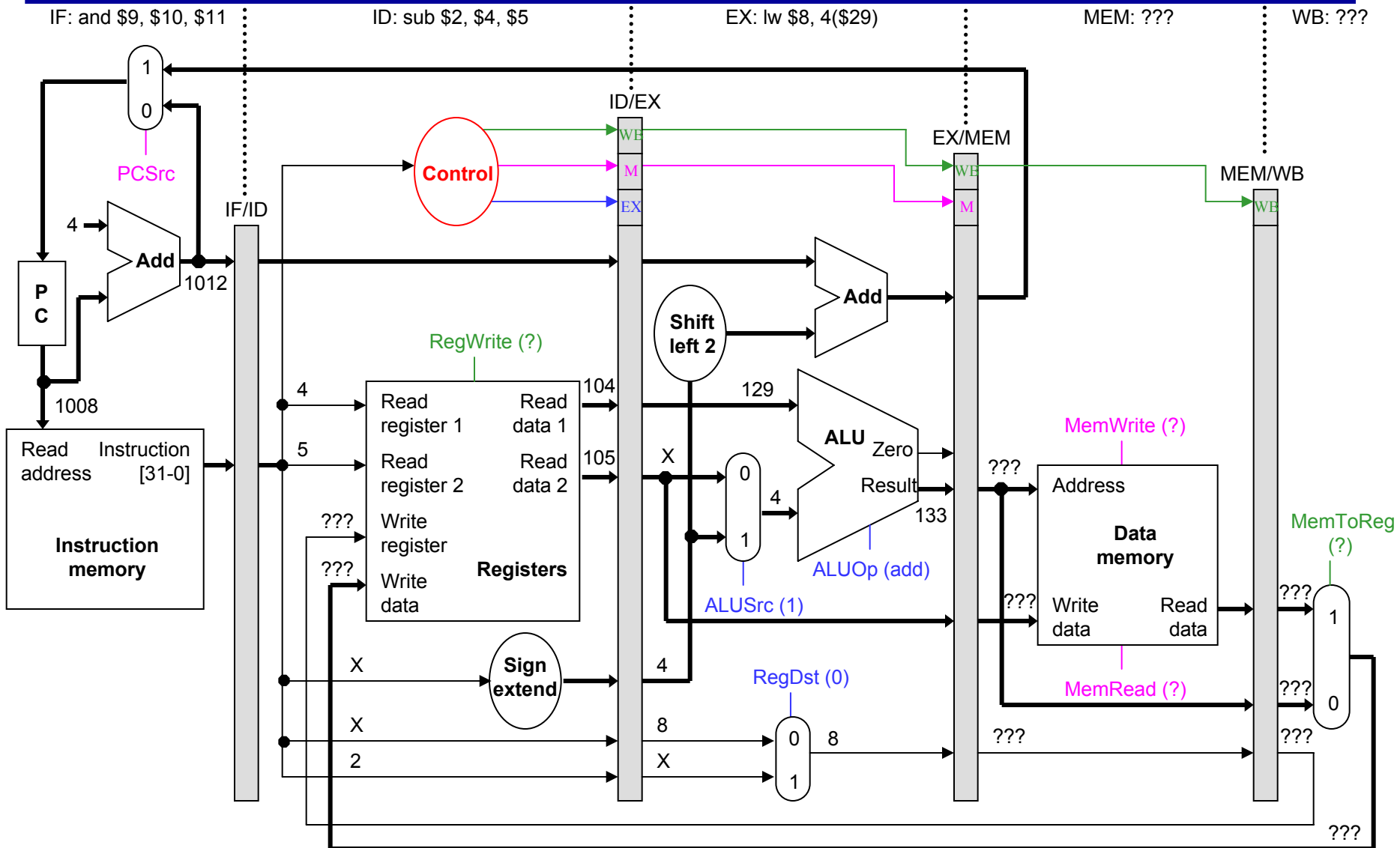
# Cycle 1 (filling)



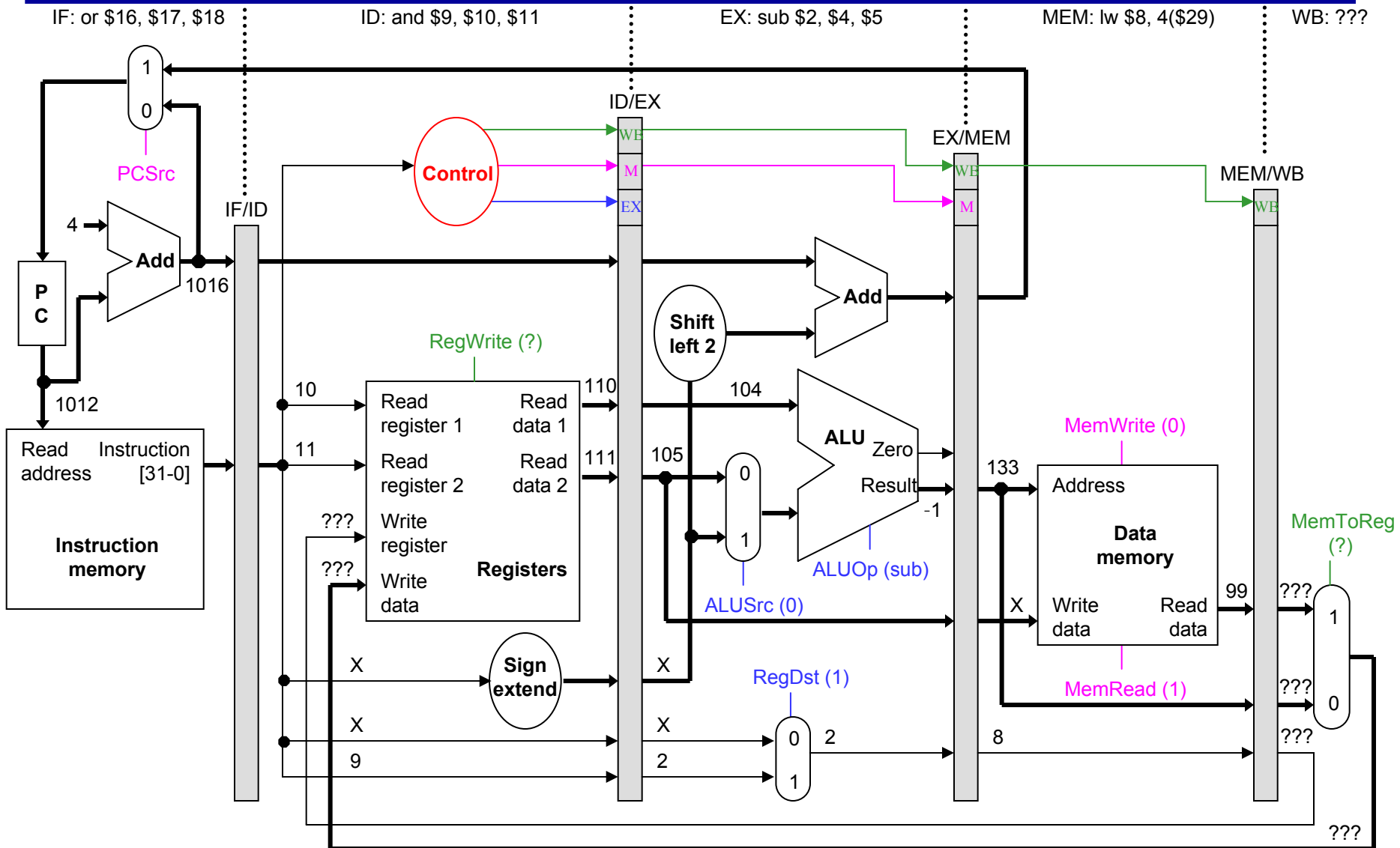
# Cycle 2



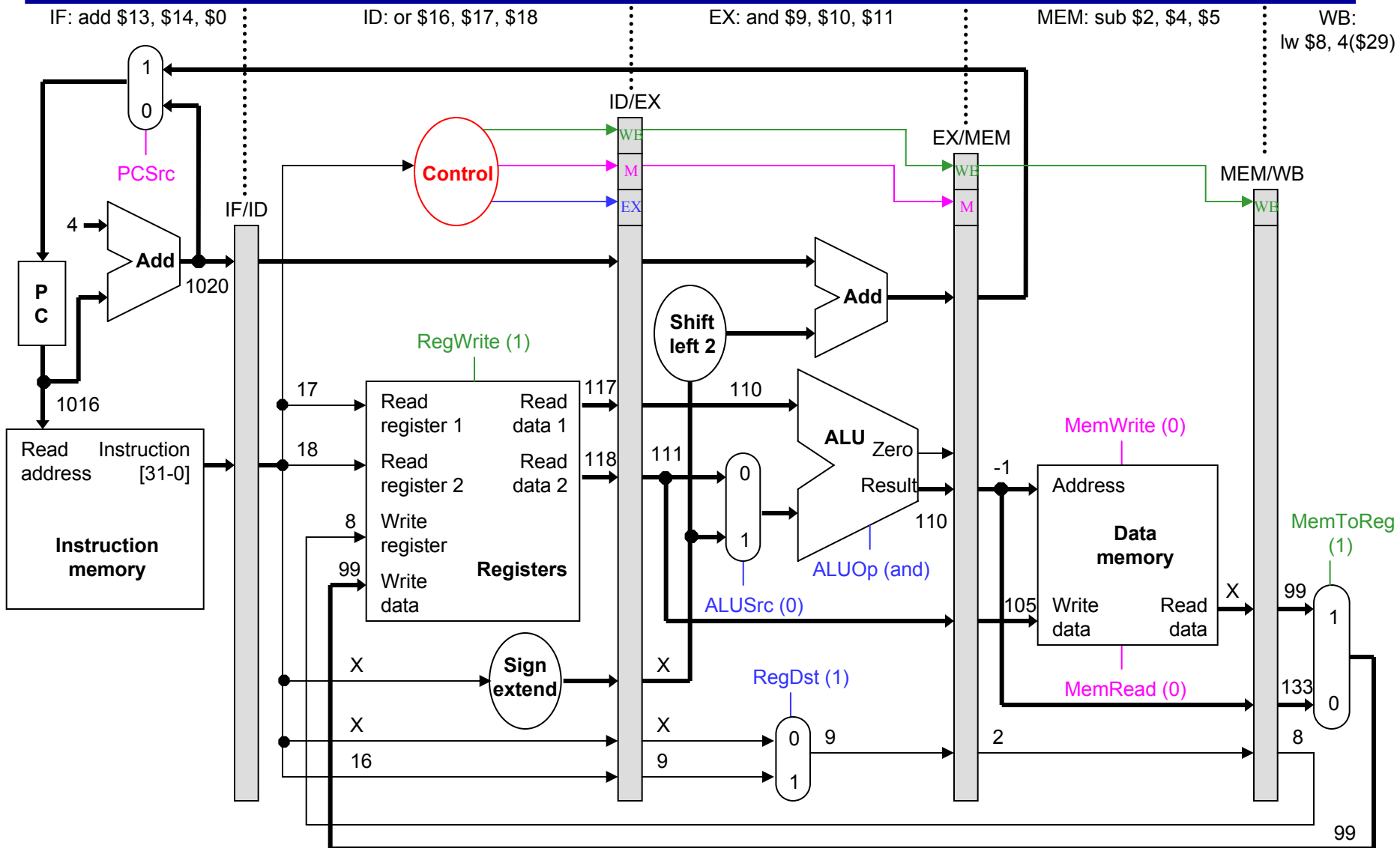
# Cycle 3



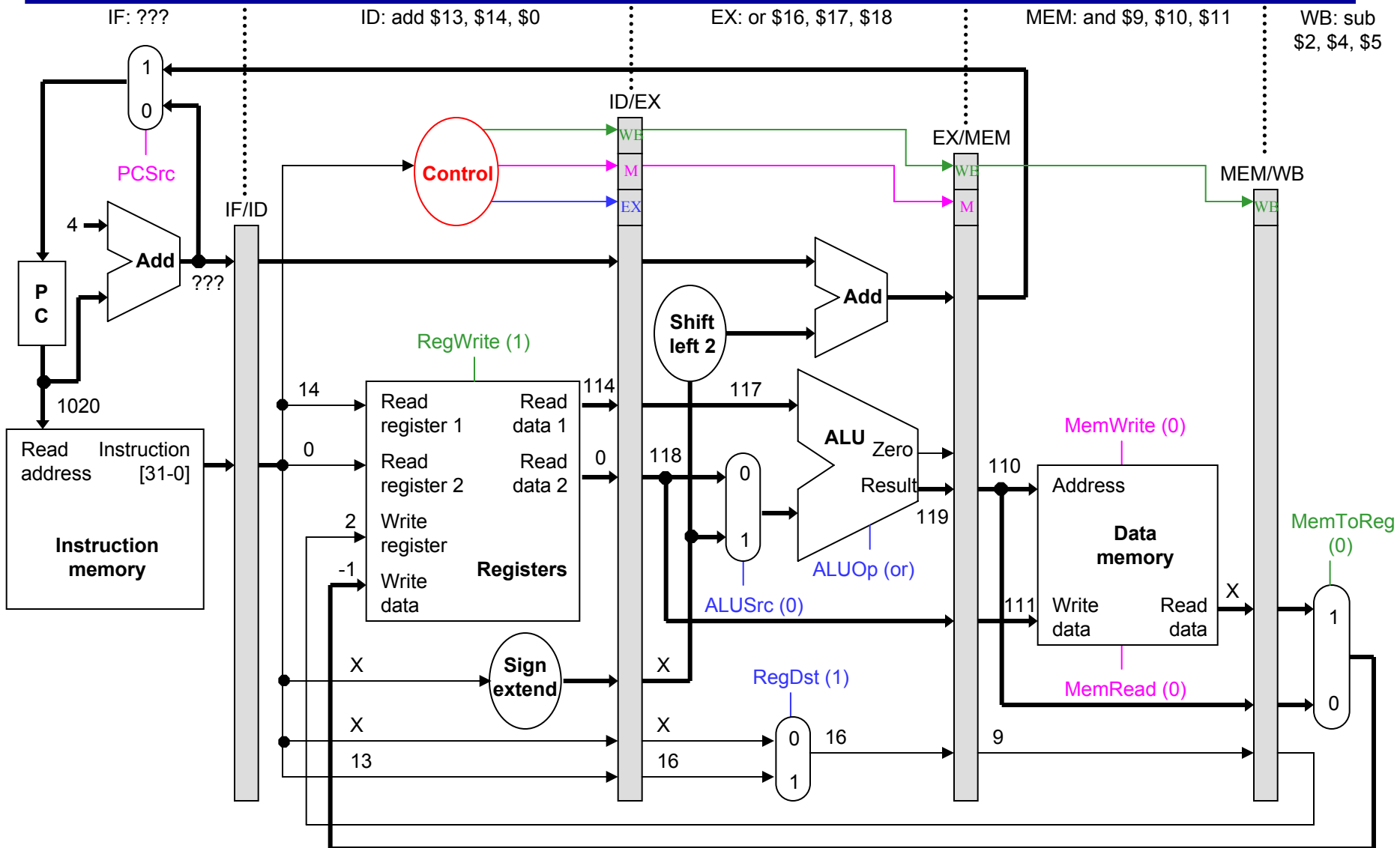
# Cycle 4



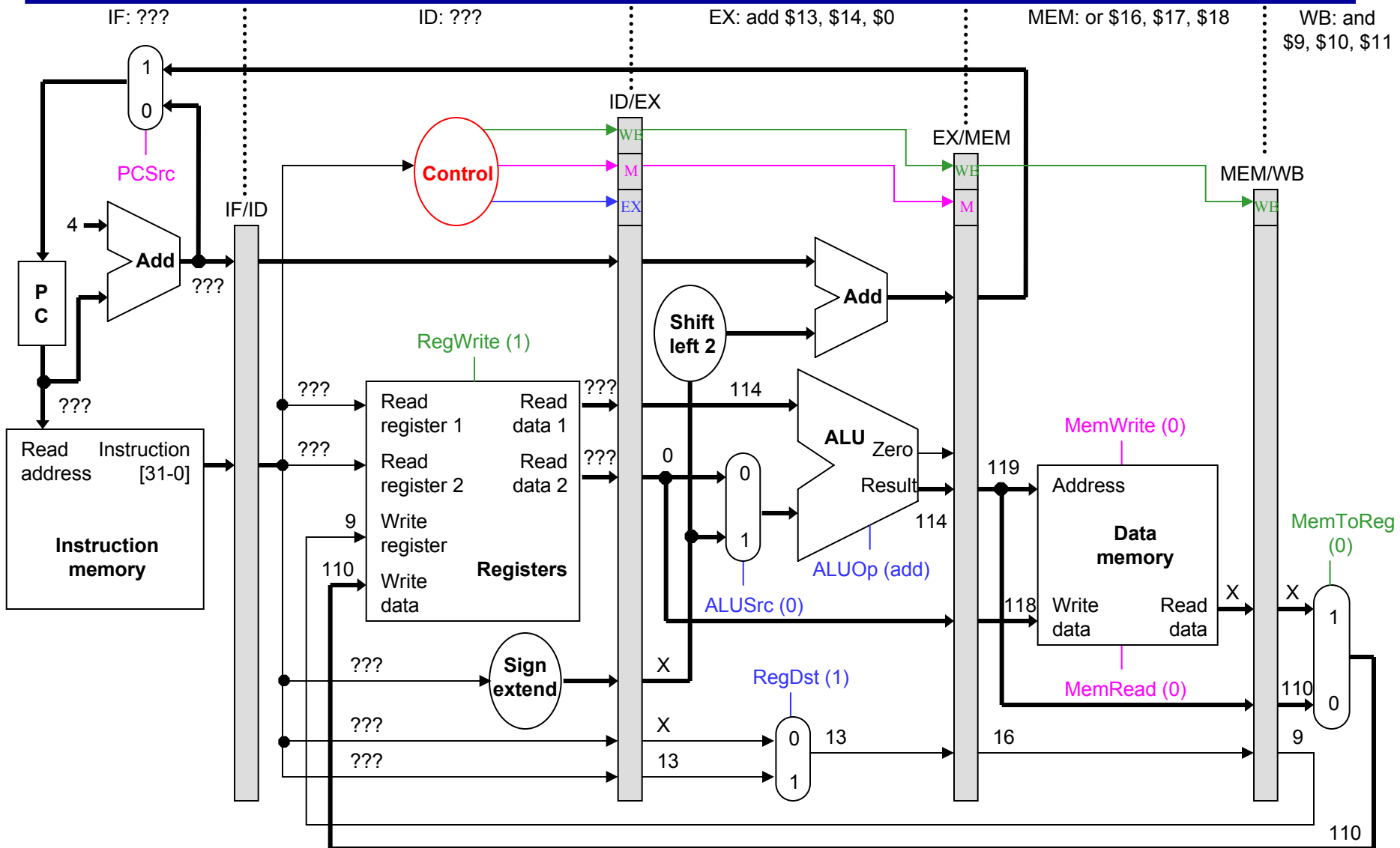
# Cycle 5 (full)



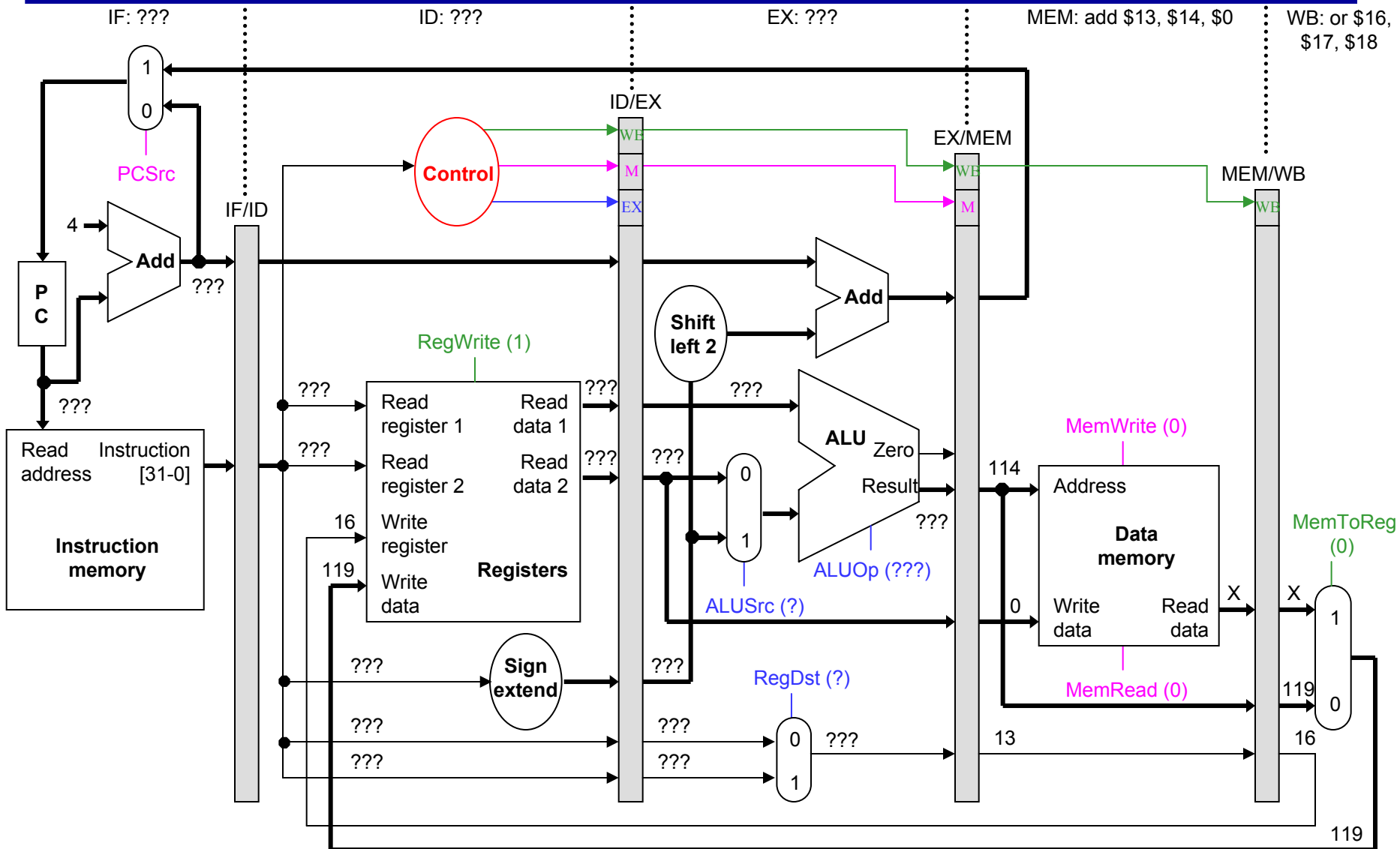
# Cycle 6 (emptying)



# Cycle 7

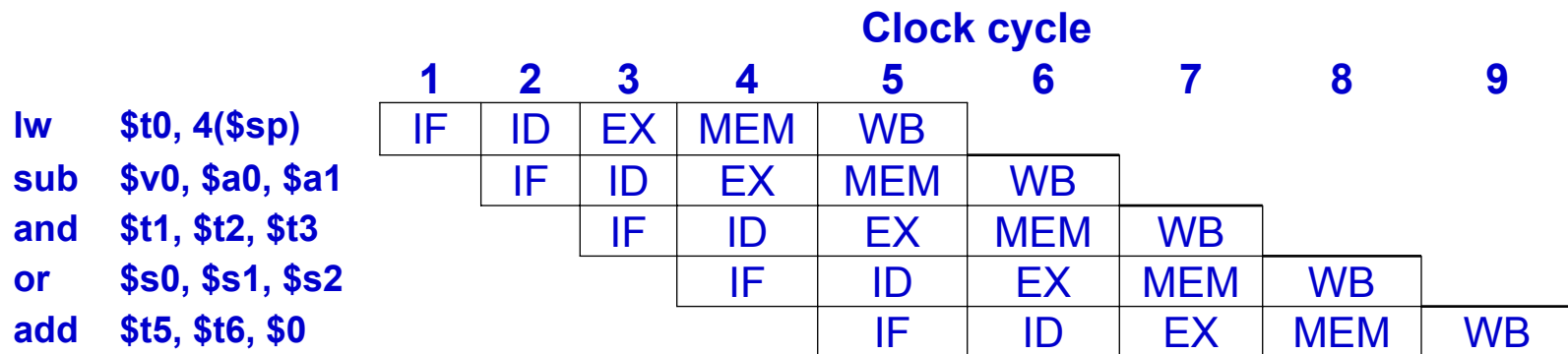


# Cycle 8





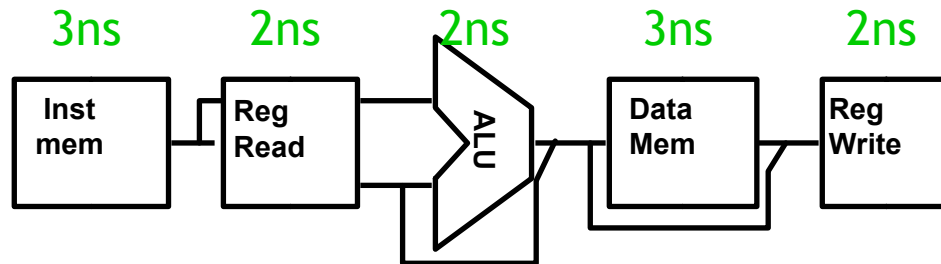
# That's a lot of diagrams there



- ❑ 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.

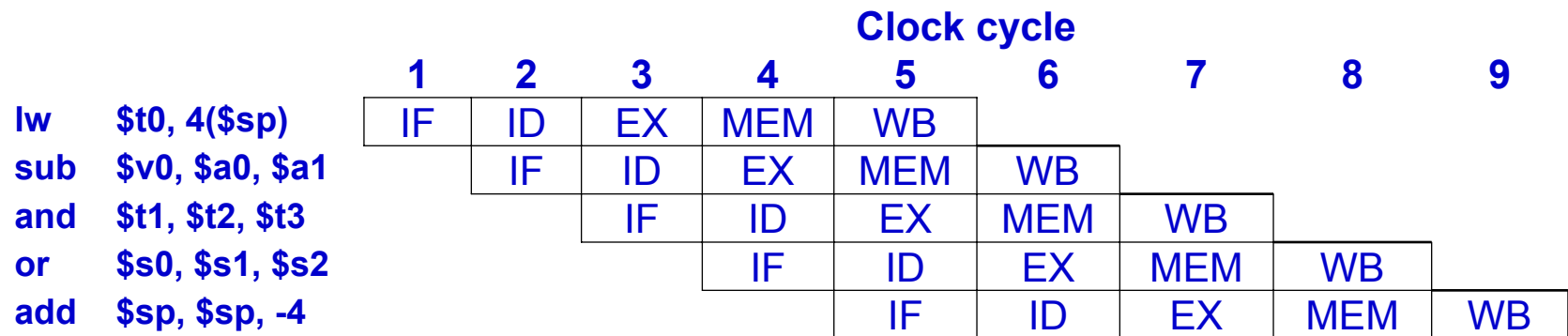
# Performance Revisited

- Assuming the following functional unit latencies:



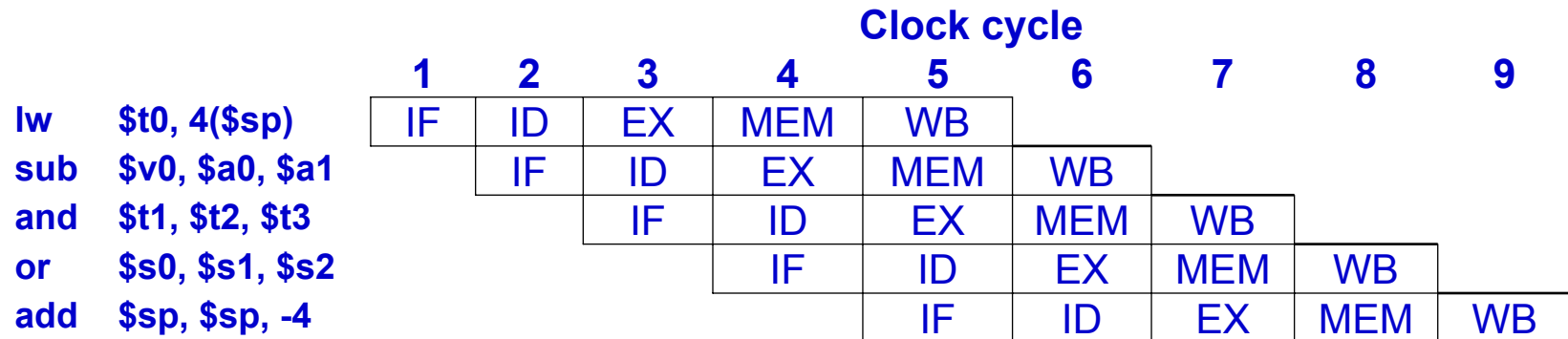
- What is the cycle time of a **single-cycle implementation**?
  - What is its throughput?
- What is the cycle time of an **ideal pipelined implementation**?
  - What is its steady-state throughput?
- How much faster is pipelining?

# Ideal speedup



- ❑ In our pipeline, we can execute up to five instructions simultaneously.
  - This implies that the maximum speedup is 5 times.
  - In general, the **ideal speedup** equals the pipeline depth.
- ❑ Why was our speedup on the previous slide “only” 4 times?
  - The pipeline stages are imbalanced: a register file and ALU operations can be done in 2ns, but we must stretch that out to 3ns to keep the ID, EX, and WB stages synchronized with IF and MEM.
  - Balancing the stages is one of the many hard parts in designing a pipelined processor.

# The pipelining paradox



- ❑ Pipelining does *not* improve the **execution time** of any single instruction. Each instruction here actually takes *longer* to execute than in a single-cycle datapath (15ns vs. 12ns)!
- ❑ Instead, pipelining increases the **throughput**, or the amount of work done per unit time. Here, several instructions are executed together in each clock cycle.
- ❑ The result is improved execution time for a *sequence* of instructions, such as an entire program.

# 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.

# So far, 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, as we saw last time.
- ❑ 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)

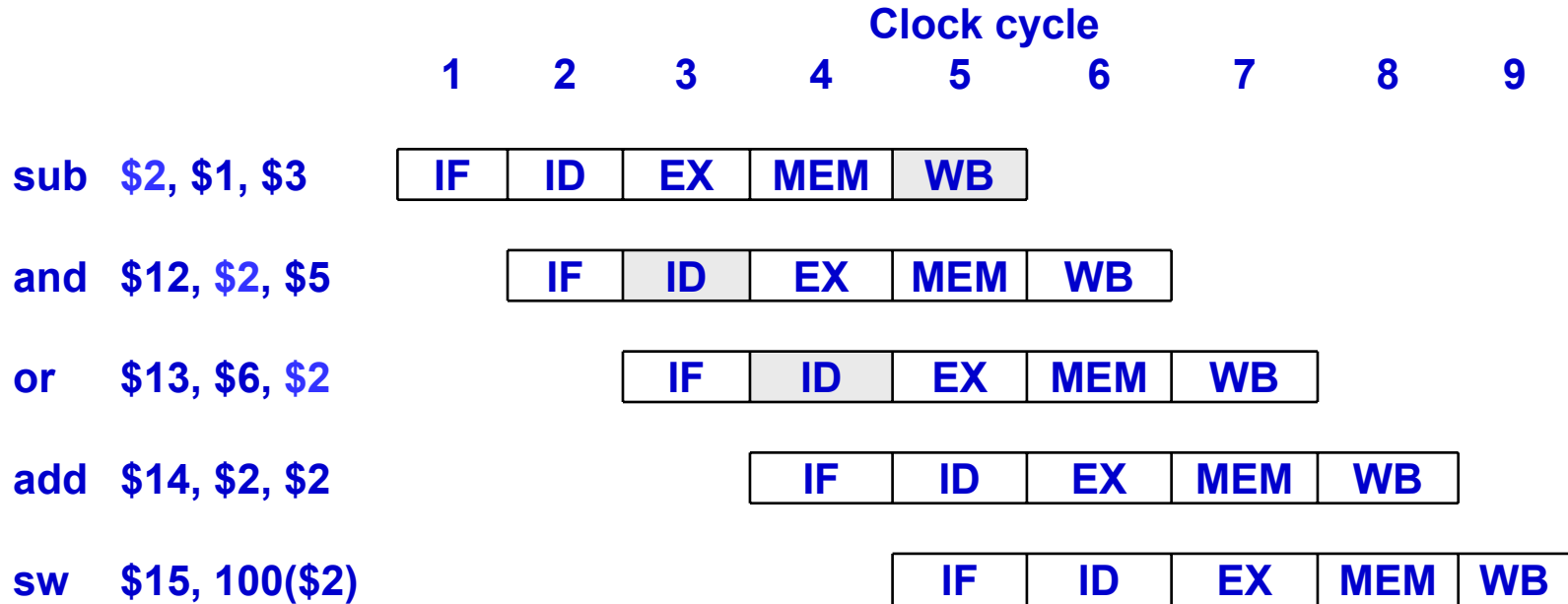
# An example with dependencies

---

```
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 and multicycle datapaths.
  - 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



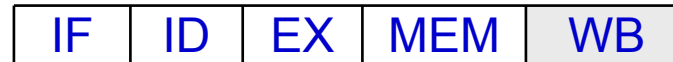
- ❑ The SUB instruction does not write to register \$2 until clock cycle 5. This causes two **data hazards** in our current pipelined datapath.
  - The AND reads register \$2 in cycle 3. Since SUB hasn't modified the register yet, this will be the *old* value of \$2, not the new one.
  - Similarly, the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

# Things that are okay

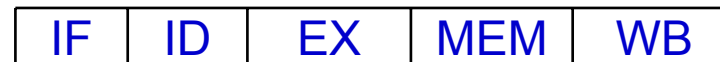
Clock cycle

1 2 3 4 5 6 7 8 9

sub \$2, \$1, \$3



and \$12, \$2, \$5



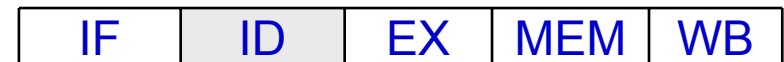
or \$13, \$6, \$2



add \$14, \$2, \$2



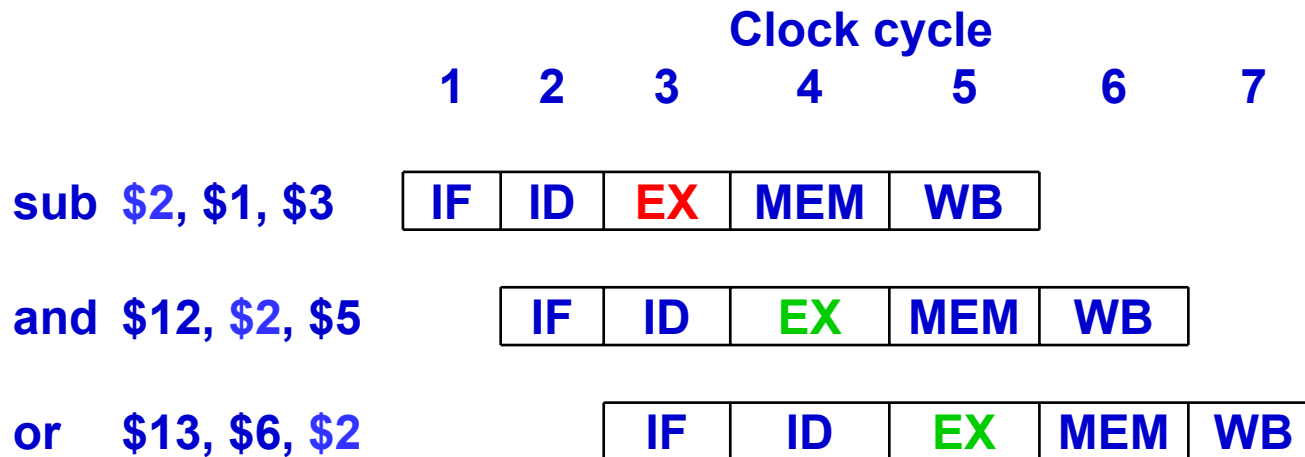
sw \$15, 100(\$2)



- ❑ 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.

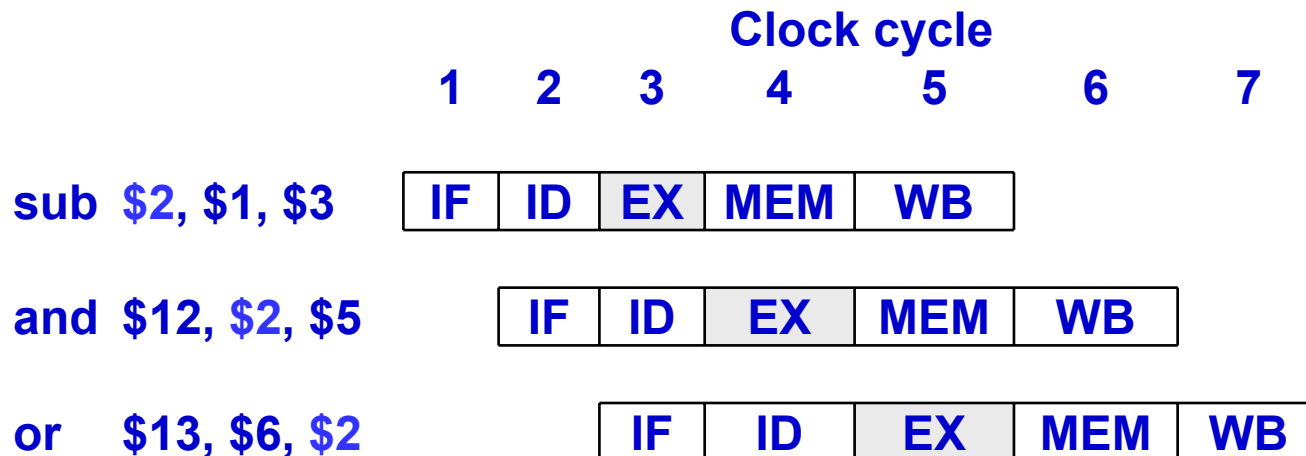
# A more detailed look at the pipeline

- ❑ We have to eliminate the hazards, so the AND and OR instructions in our example will use the correct value for register \$2.
- ❑ Let's look at when the data is actually produced and consumed.
  - The SUB instruction produces its result in its EX stage, during cycle 3 in the diagram below.
  - The AND and OR need the new value of \$2 in their EX stages, during clock cycles 4-5 here.



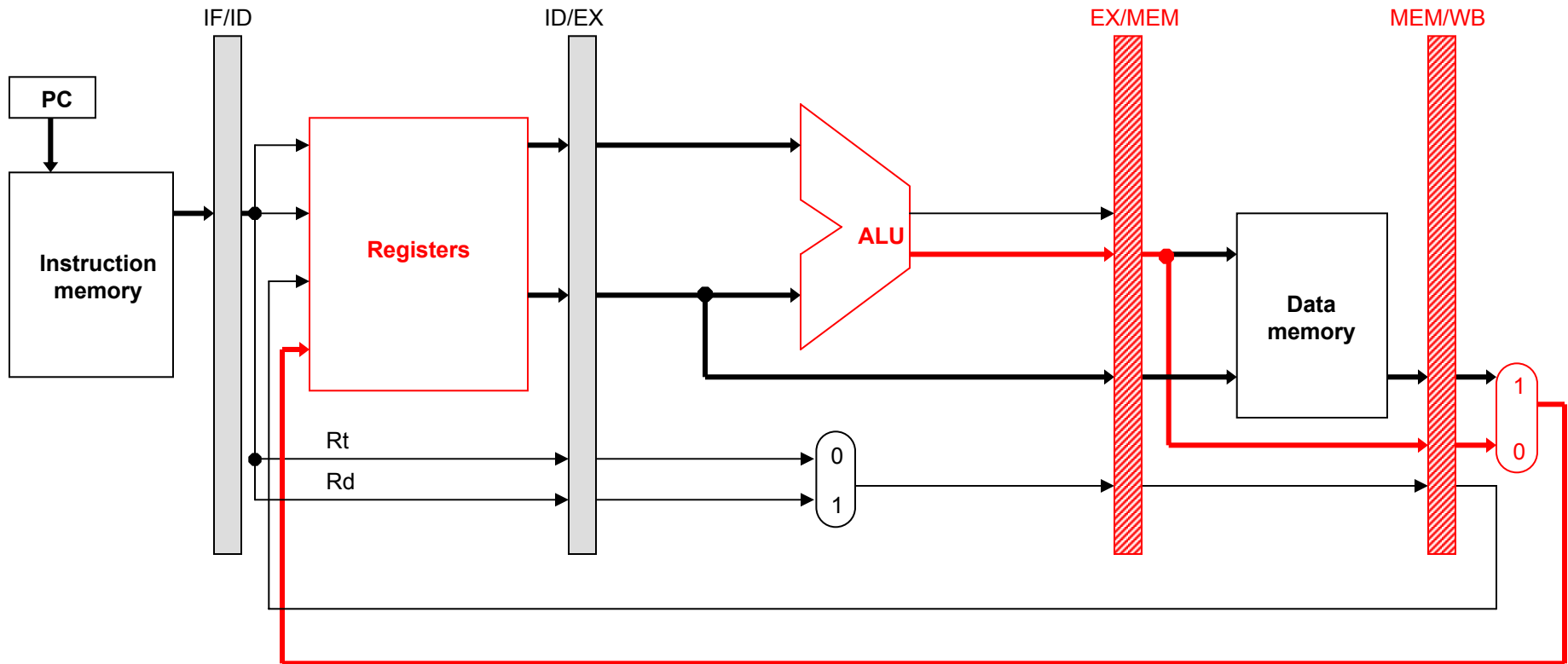
# Bypassing the register file

- ❑ The actual result \$1 - \$3 is computed in clock cycle 3, *before* it's needed in cycles 4 and 5.
- ❑ If we could somehow bypass the writeback and register read stages when needed, then we can eliminate these data hazards.
  - Today we'll focus on hazards involving arithmetic instructions.
  - Next time, we'll examine the lw instruction.
- ❑ Essentially, we need to pass the ALU output from SUB directly to the AND and OR instructions, without going through the register file.



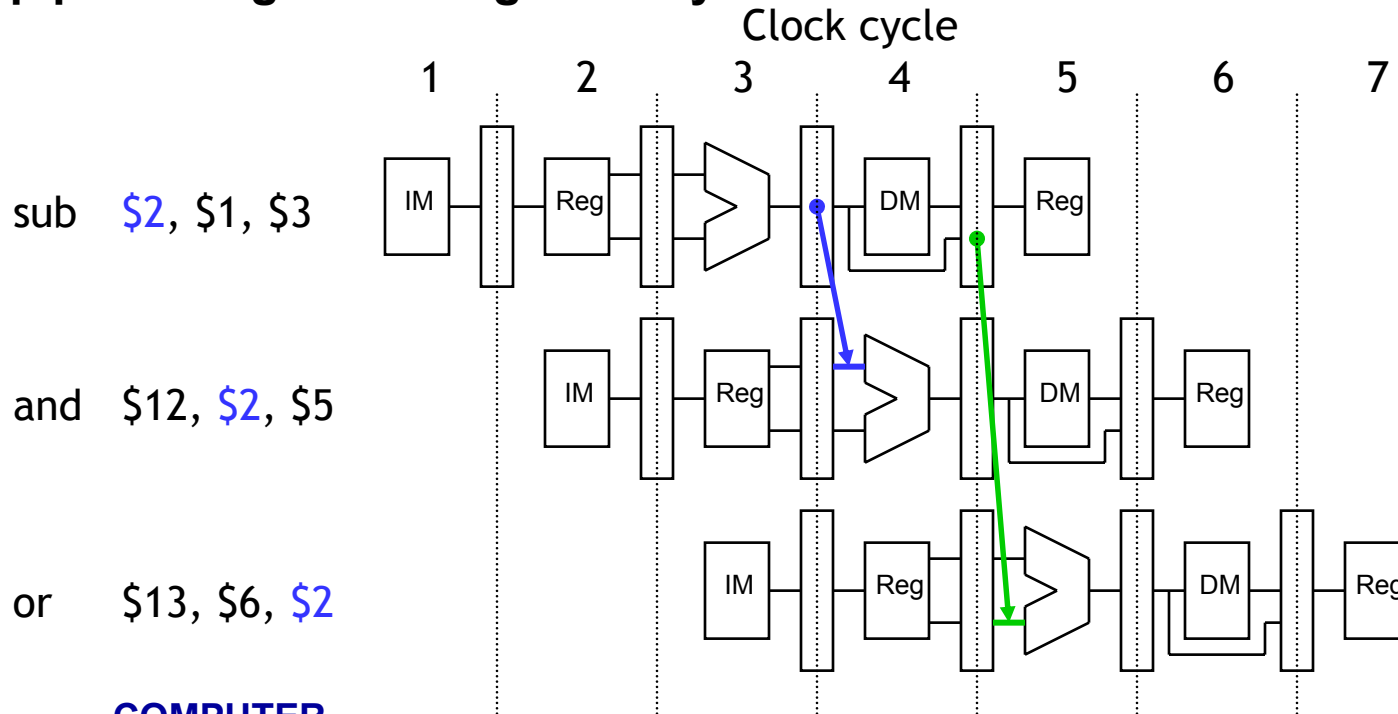
# Where to find the ALU result

- ❑ The ALU result generated in the EX stage is normally passed through the pipeline registers to the MEM and WB stages, before it is finally written to the register file.
- ❑ This is an abridged diagram of our pipelined datapath.



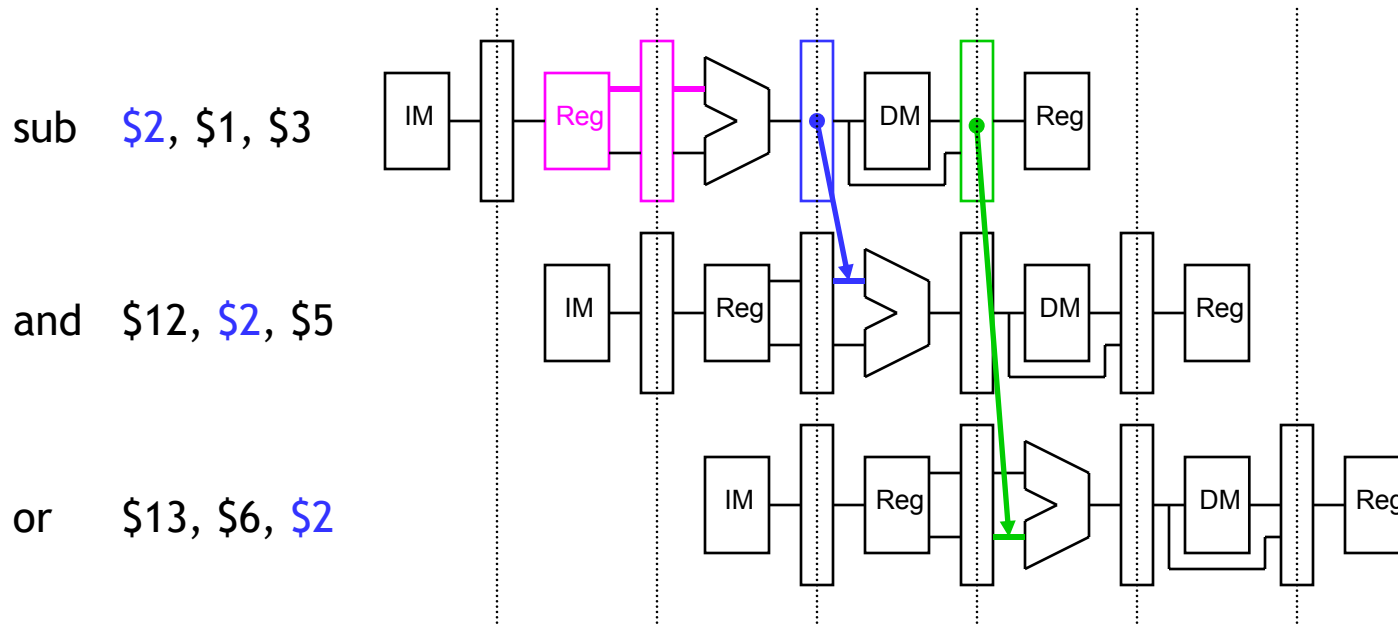
# Forwarding

- Since the pipeline registers already contain the ALU result, we could just **forward** that value to subsequent instructions, to prevent data hazards.
  - In clock cycle 4, the AND instruction can get the value \$1 – \$3 from the **EX/MEM** pipeline register used by sub.
  - Then in cycle 5, the OR can get that same result from the **MEM/WB** pipeline register being used by SUB.

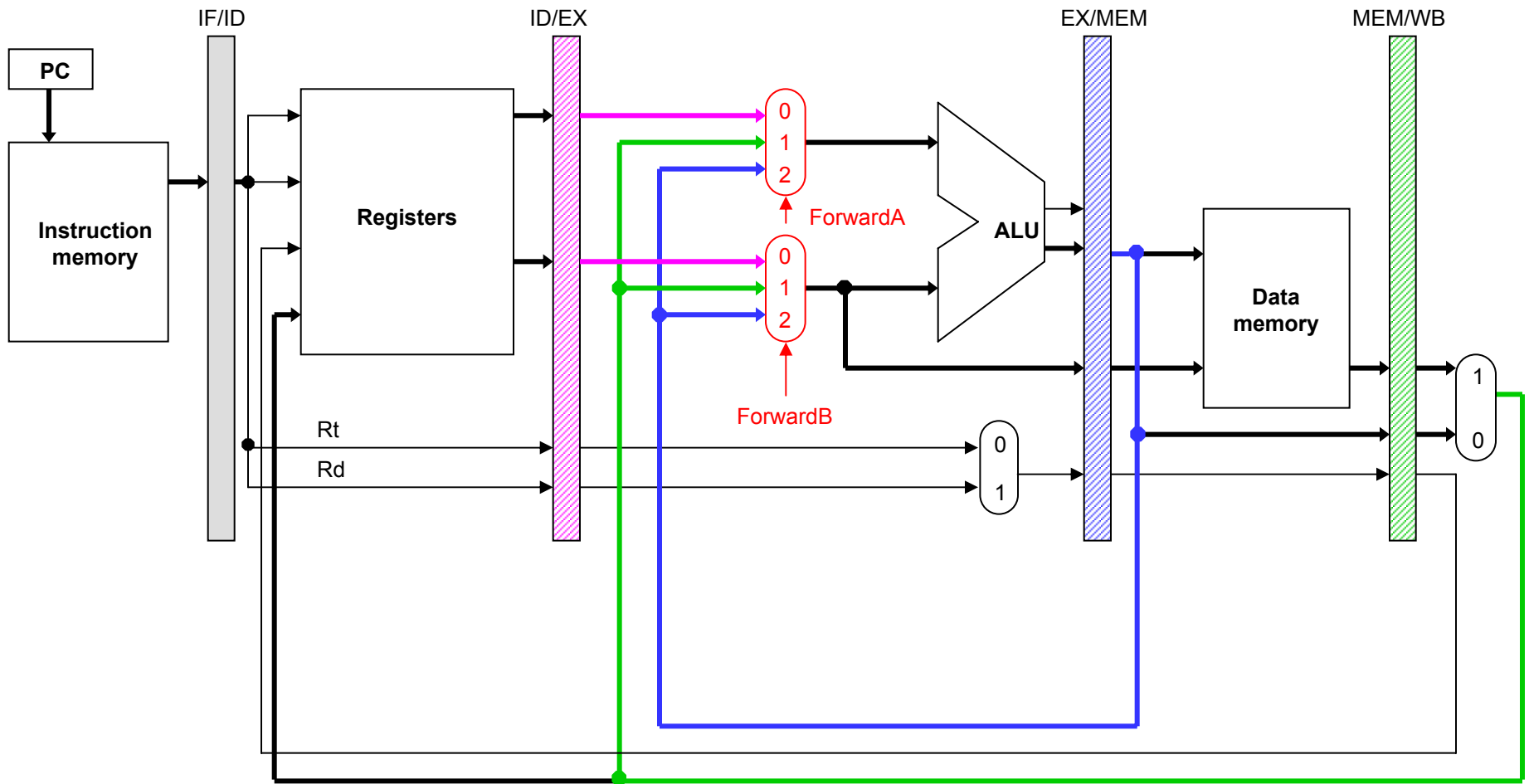


# Outline of forwarding hardware

- A **forwarding unit** selects the correct ALU inputs for the EX stage.
  - If there is no hazard, the ALU's operands will come from the **register file**, just like before.
  - If there is a hazard, the operands will come from either the **EX/MEM** or **MEM/WB** pipeline registers instead.
- The ALU sources will be selected by two new multiplexers, with control signals named **ForwardA** and **ForwardB**.

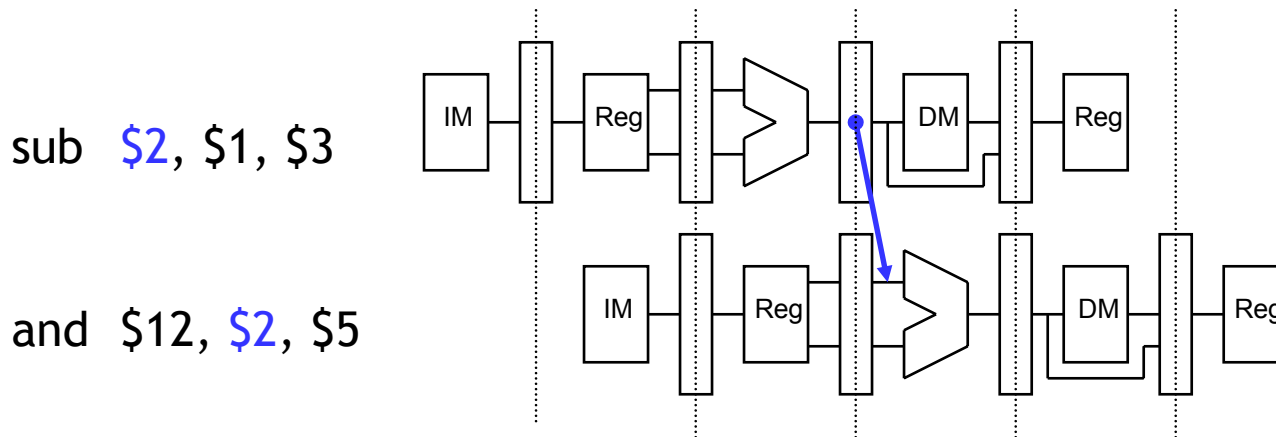


# Simplified datapath with forwarding muxes



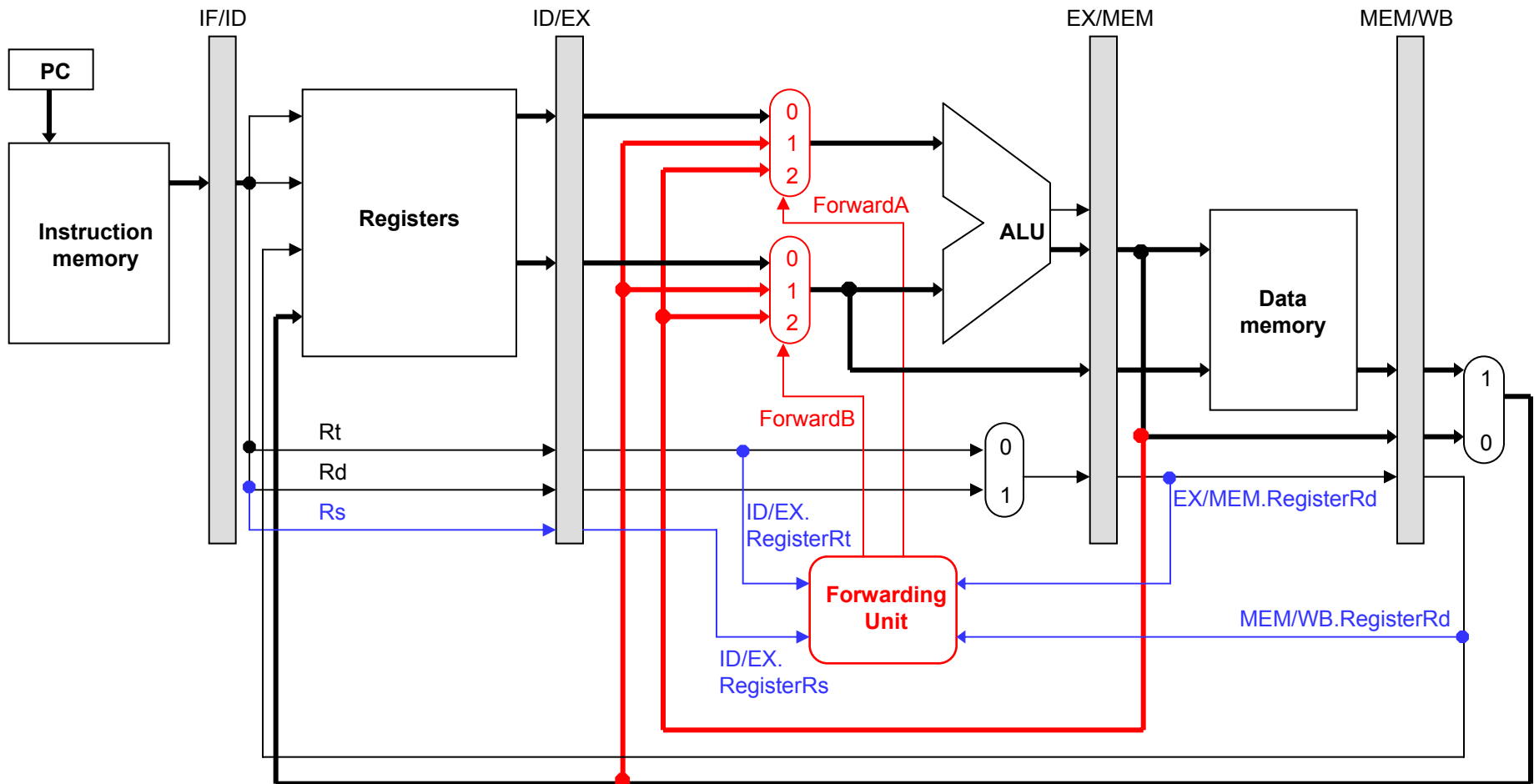
# Detecting EX/MEM data hazards

- ❑ So how can the hardware determine if a hazard exists?
- ❑ An **EX/MEM hazard** occurs between the instruction currently in its EX stage and the previous instruction if:
  1. The previous instruction will write to the register file, *and*
  2. The destination is one of the ALU source registers in the EX stage.
- ❑ There is an EX/MEM hazard between the two instructions below.



- ❑ Data in a pipeline register can be referenced using a class-like syntax. For example, ID/EX.RegisterRt refers to the rt field stored in the ID/EX pipeline.

# Simplified datapath with forwarding



---

**The End**  
**Good Luck on Your Final !**