CoE/ECE 0142 Computer Organization

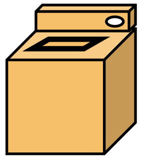
Pipelining

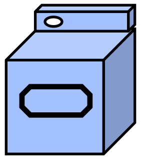
Instructor: Jun Yang

Slides are adapted from Zilles

A relevant question

- Assuming you’ve got:
  - One washer (takes 30 minutes)



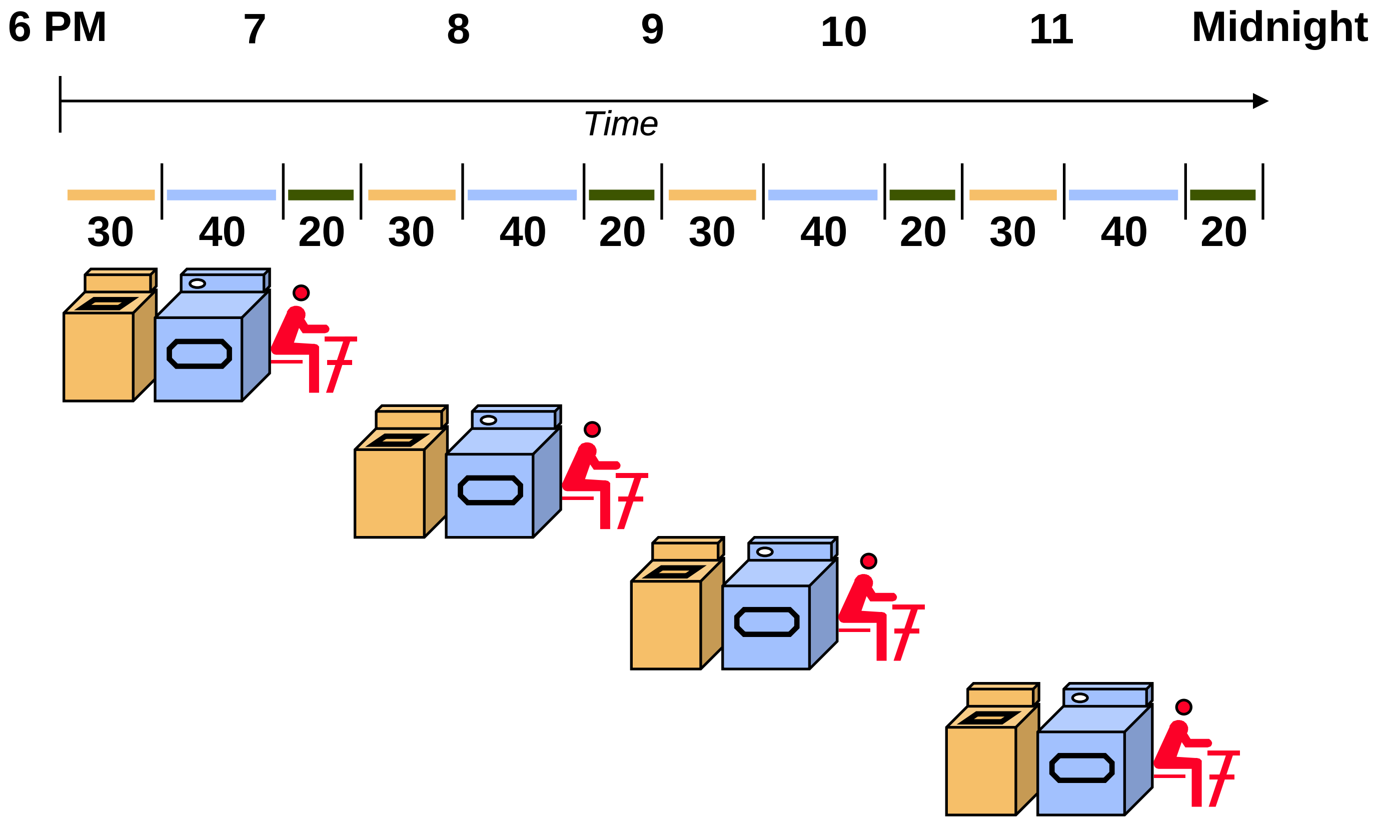


  - One drier (takes 40 minutes)

  - One “folder” (takes 20 minutes)

- It takes 90 minutes to wash, dry, and fold 1 load of laundry.
  - How long does 4 loads take?

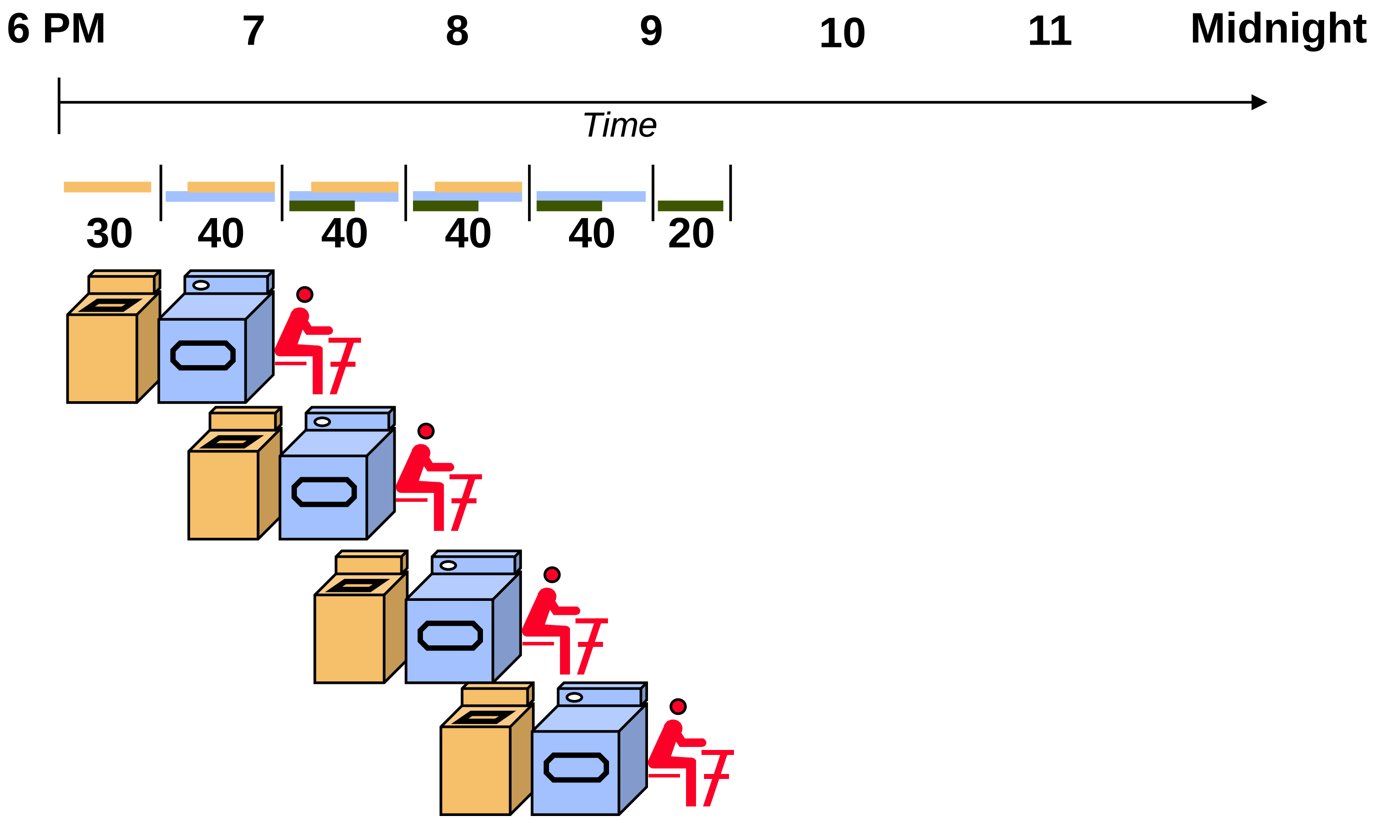
The slow way



- If each load is done sequentially it takes 6 hours

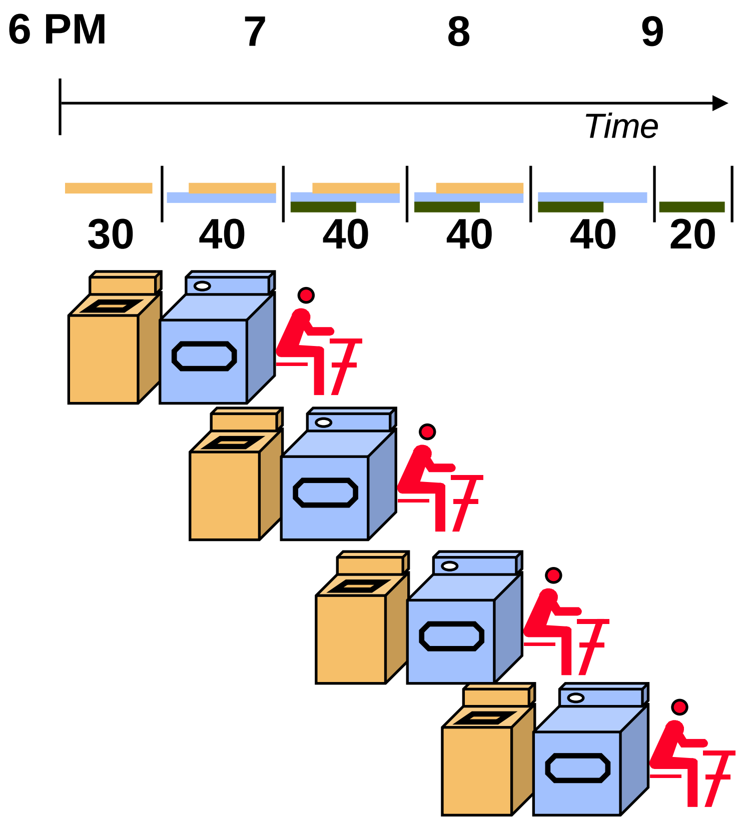
Laundry Pipelining

- ❑ **Start each load as soon as possible**
  - – **Overlap loads**



- ❑ **Pipelined laundry takes 3.5 hours**

Pipelining Lessons



- **Multiple tasks operating simultaneously using different resources**
- **Pipeline rate limited by slowest pipeline stage**
- **Unbalanced lengths of pipe stages reduces speedup**
- **Potential speedup = Number pipe stages**
- **Pipelining doesn’t help latency of single load, it helps throughput of entire workload**
- **Time to “fill” pipeline and time to “drain” it reduces speedup**

Pipelining

- Pipelining is a general-purpose efficiency technique
  - It is not specific to processors
- Pipelining is used in:
  - Assembly lines
  - Fast food restaurants
- Pipelining gives the best of both worlds and is used in just about every modern processor.

Instruction execution review

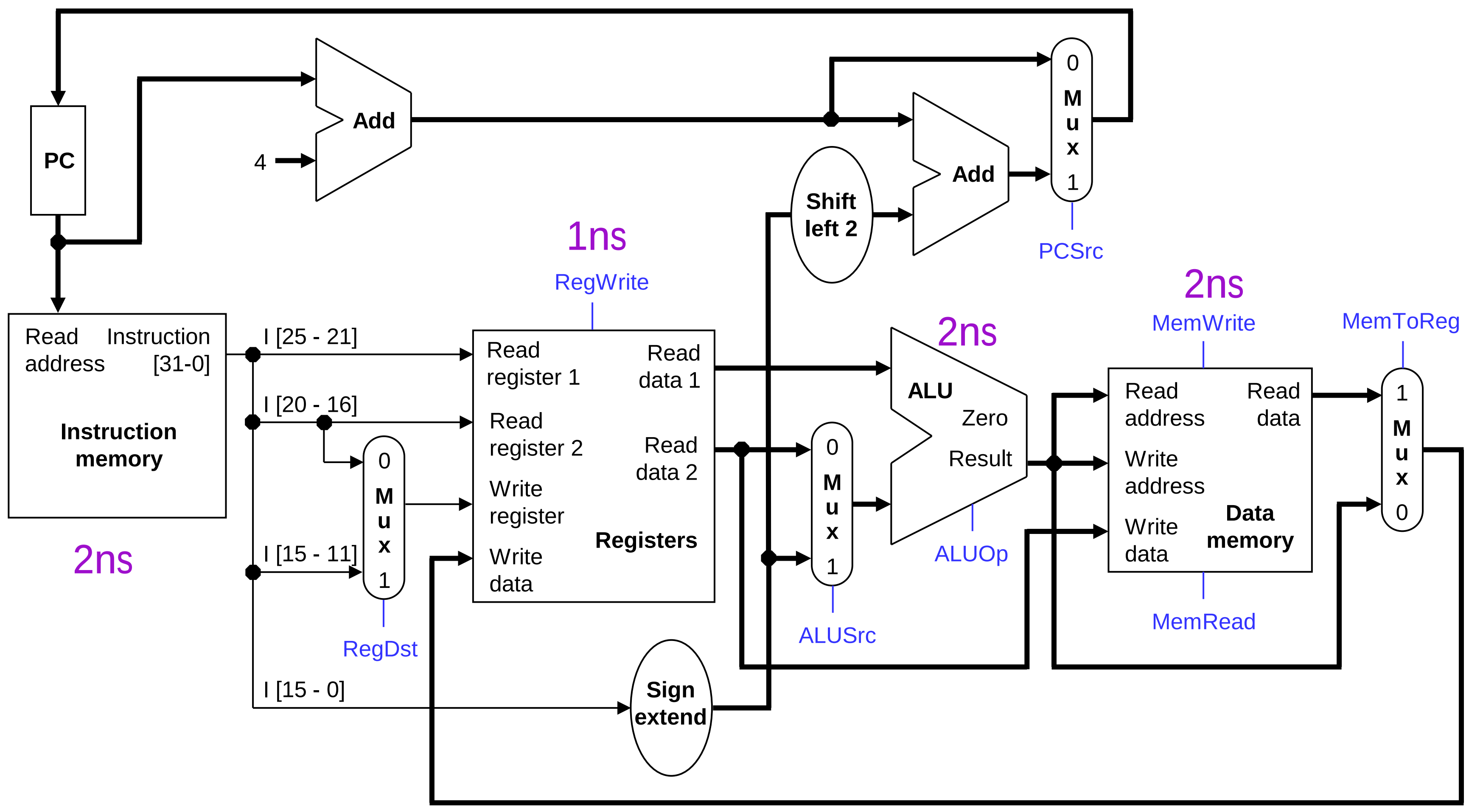
- Executing a MIPS instruction can take up to five steps.

Step	Name	Description
Instruction Fetch	IF	Read an instruction from memory.
Instruction Decode	ID	Read source registers and generate control signals.
Execute	EX	Compute an R-type result or a branch outcome.
Memory	MEM	Read or write the data memory.
Writeback	WB	Store a result in the destination register.

- However, as we saw, not all instructions need all five steps.

Instruction	Steps required				
beq	IF	ID	EX		
R-type	IF	ID	EX		WB
sw	IF	ID	EX	MEM	
lw	IF	ID	EX	MEM	WB

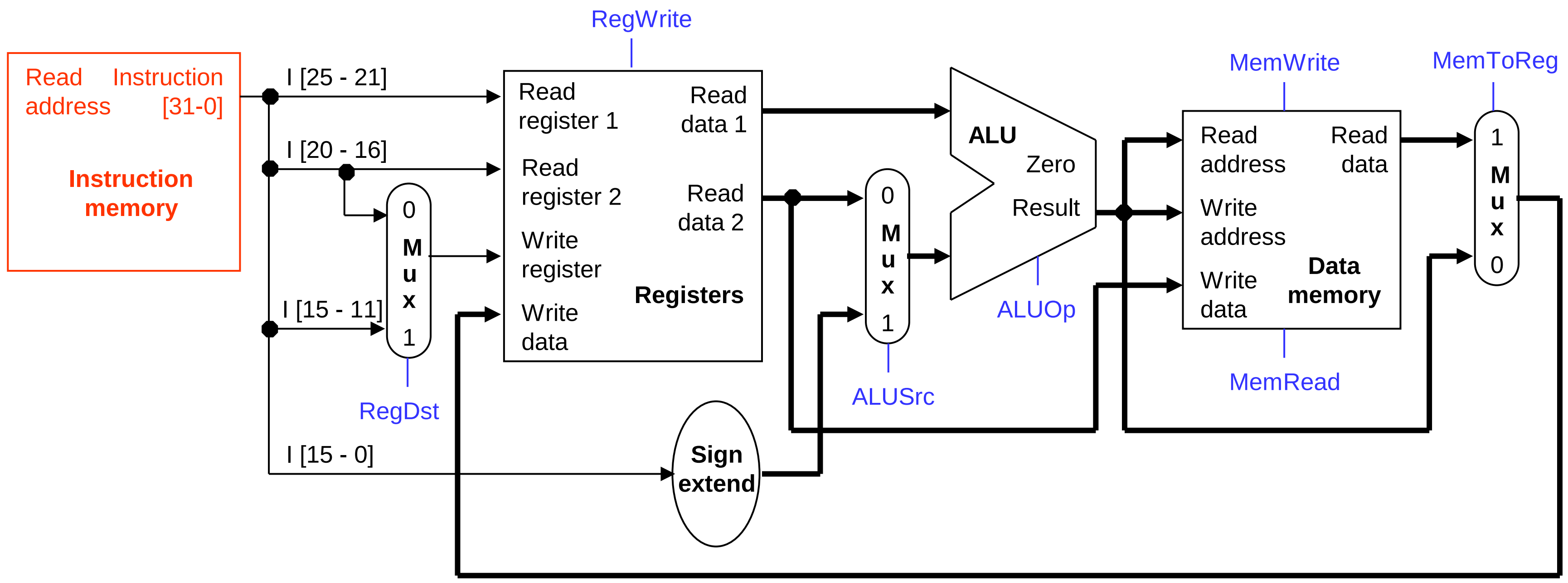
Single-cycle datapath diagram



- How long does it take to execute each instruction?

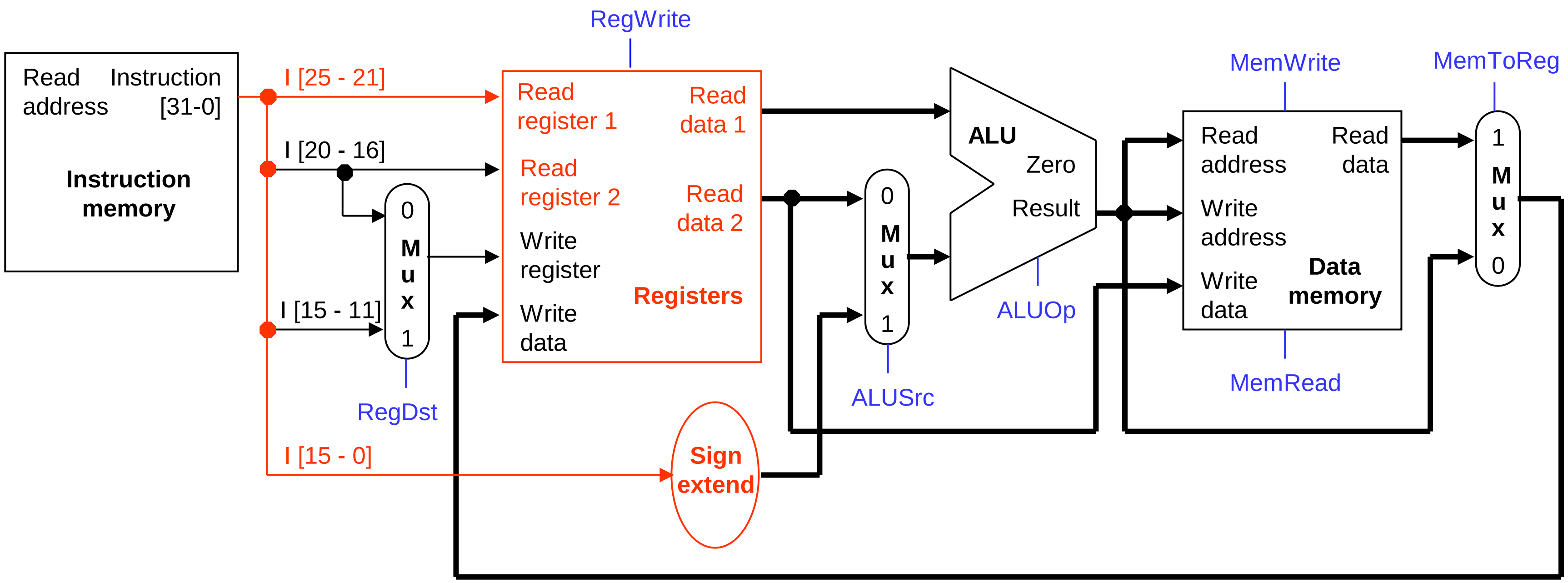
Review: Instruction Fetch (IF)

- Let’s quickly review how lw is executed in the single-cycle datapath.
- We’ll ignore PC incrementing and branching for now.
- In the Instruction Fetch (IF) step, we read the instruction memory.



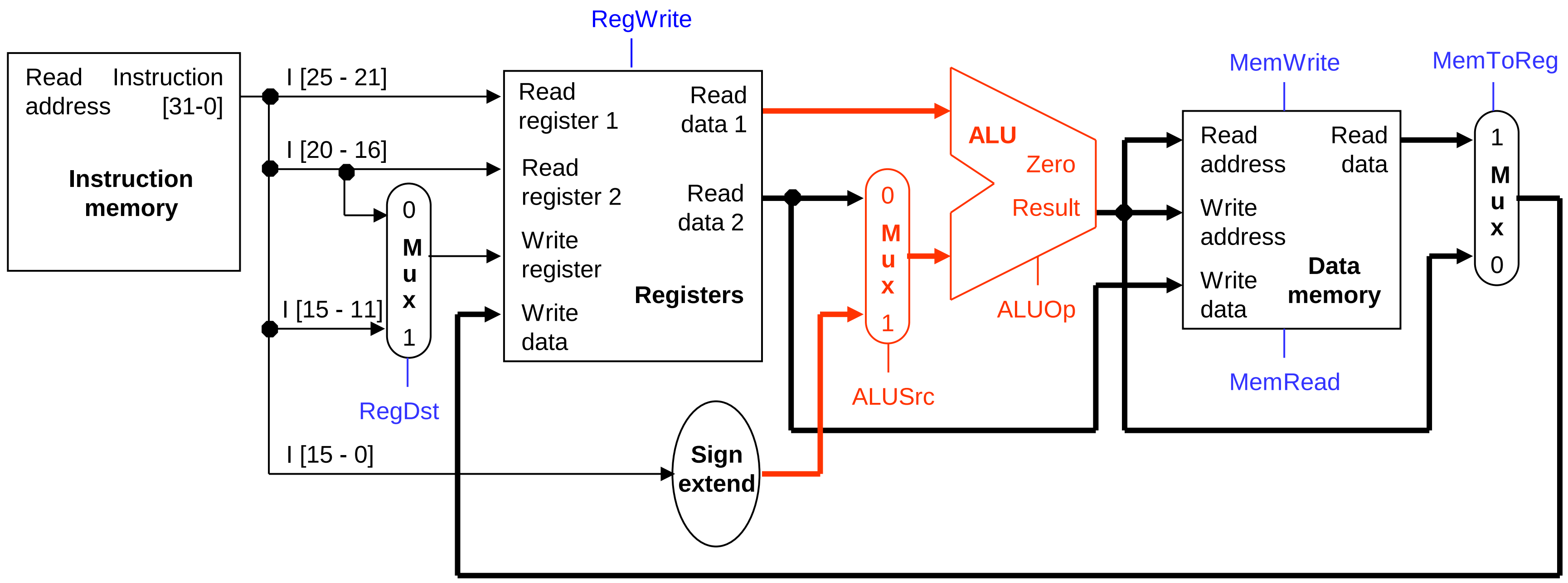
Instruction Decode (ID)

- The Instruction Decode (ID) step reads the source register from the register file.



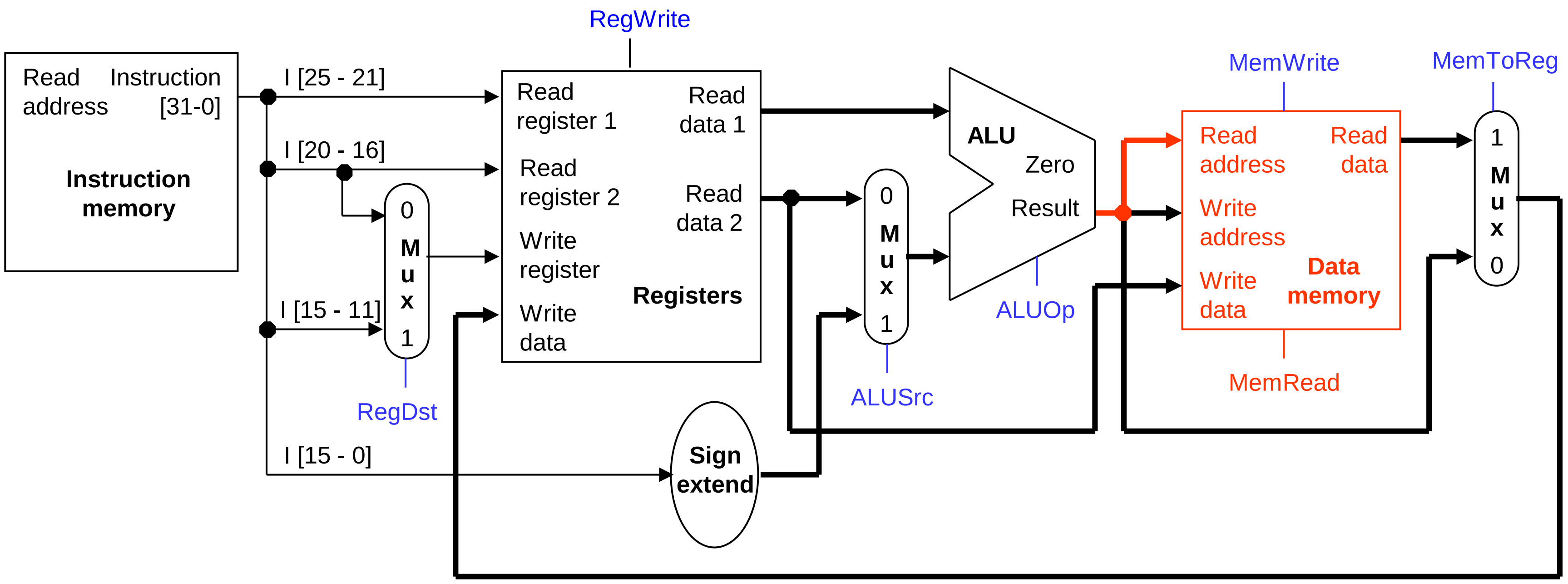
Execute (EX)

- The third step, Execute (EX), computes the effective memory address from the source register and the instruction's constant field.



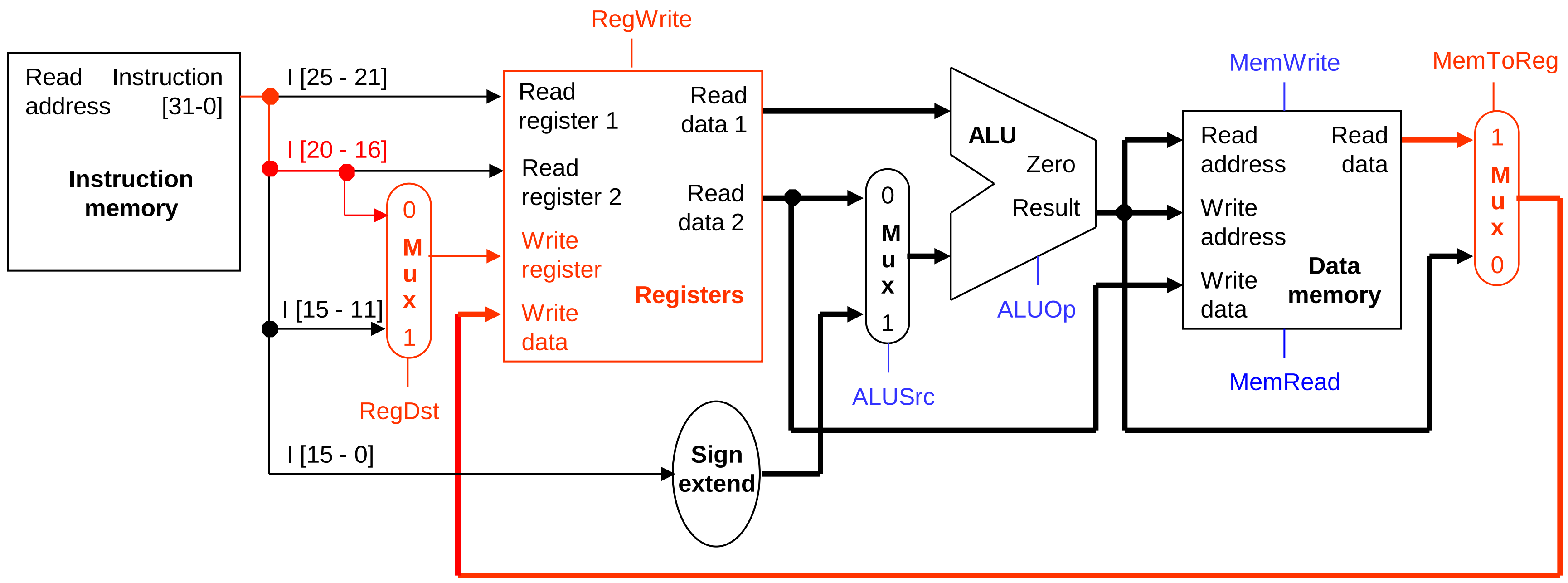
Memory (MEM)

- The Memory (MEM) step involves reading the data memory, from the address computed by the ALU.



Writeback (WB)

- Finally, in the Writeback (WB) step, the memory value is stored into the destination register.



A bunch of lazy functional units

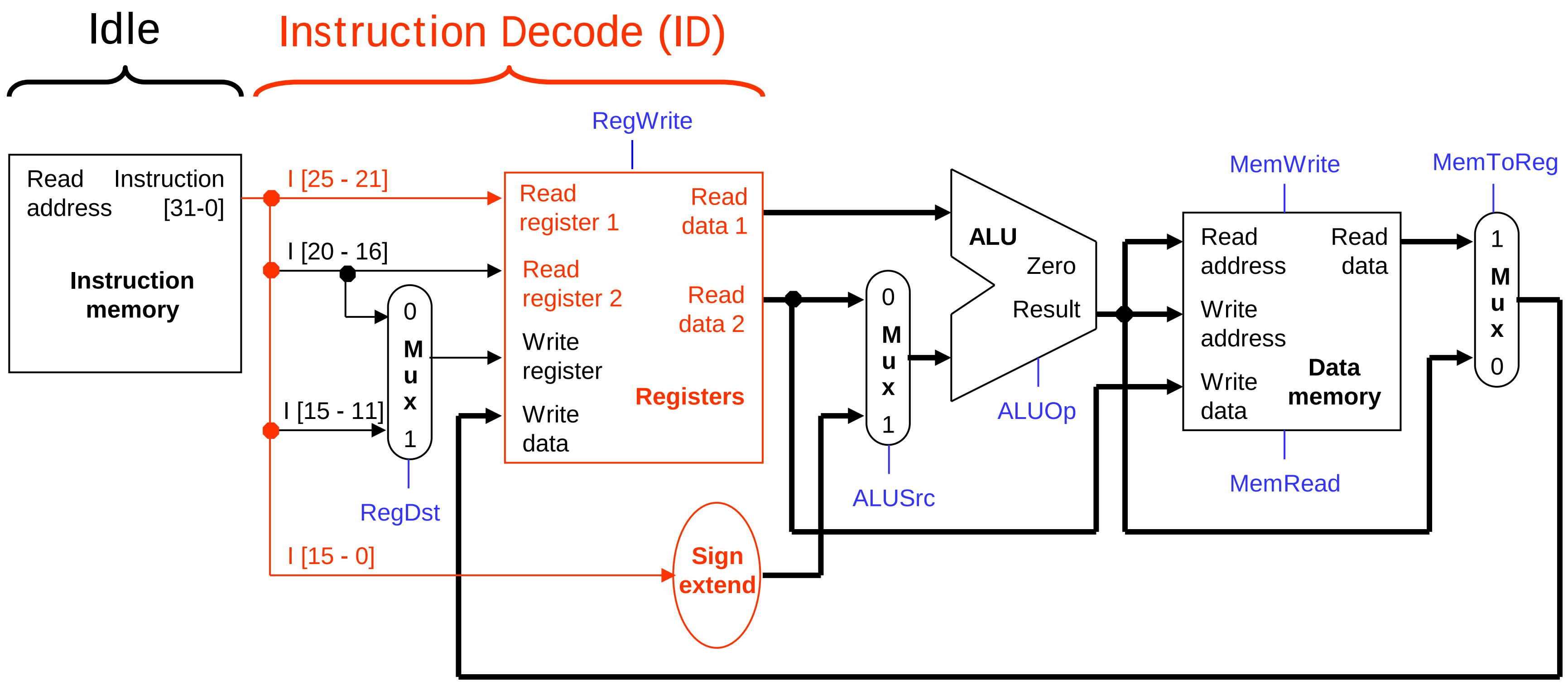
- Notice that each execution step uses a different functional unit.

- In other words, the main units are idle for most of the 8ns cycle!
  - The instruction RAM is used for just 2ns at the start of the cycle.
  - Registers are read once in ID (1ns), and written once in WB (1ns).
  - The ALU is used for 2ns near the middle of the cycle.
  - Reading the data memory only takes 2ns as well.

- That's a lot of hardware sitting around doing nothing.

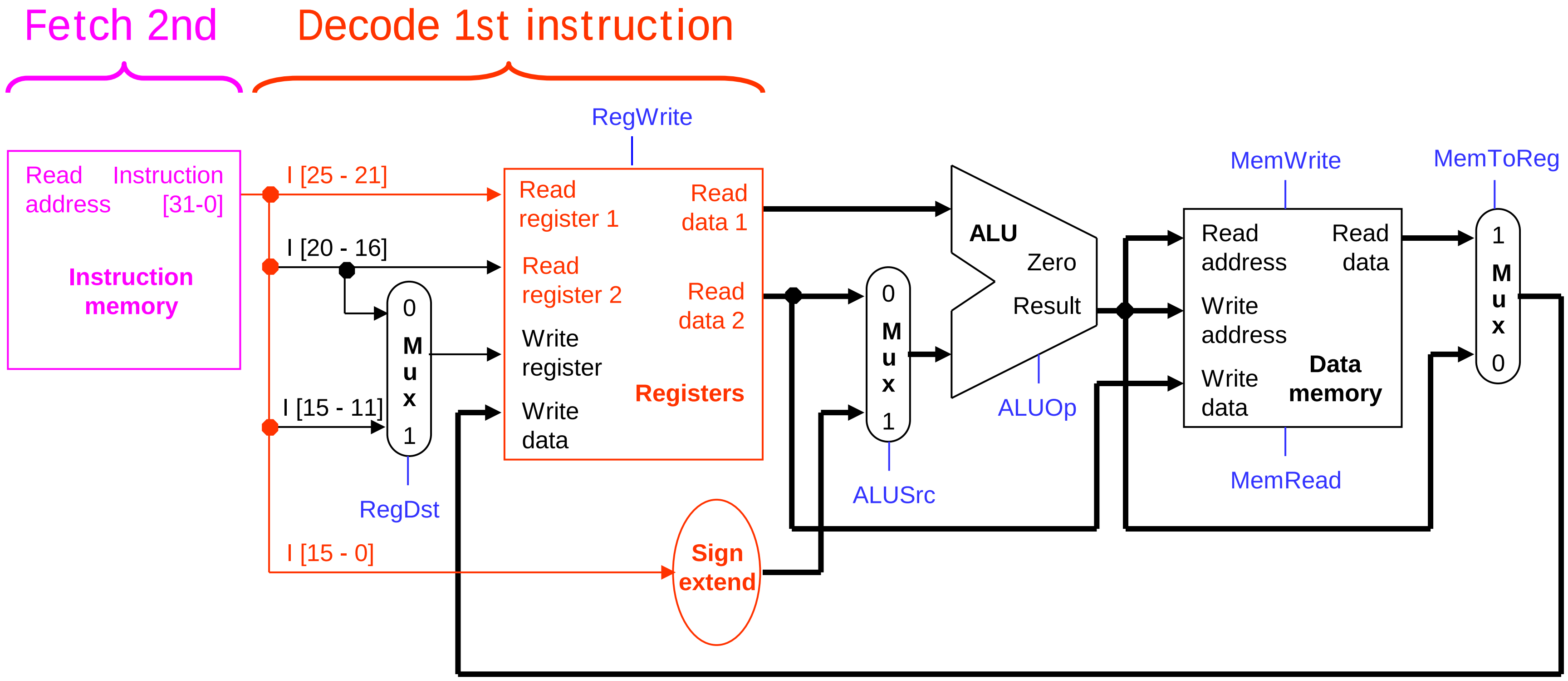
Putting those slackers to work

- We shouldn't have to wait for the entire instruction to complete before we can re-use the functional units.
- For example, the instruction memory is free in the Instruction Decode step as shown below, so...



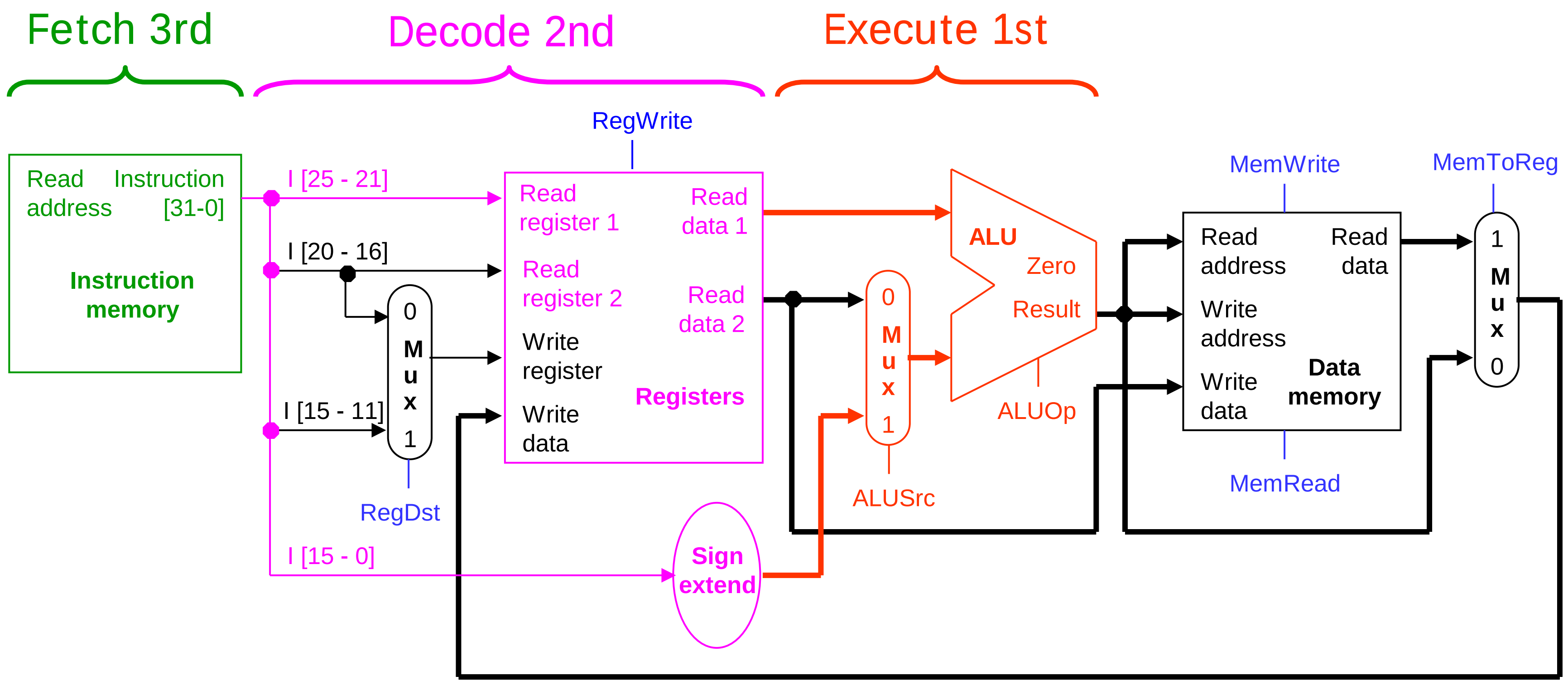
Decoding and fetching together

- Why don't we go ahead and fetch the *next* instruction while we're decoding the first one?



Executing, decoding and fetching

- Similarly, once the first instruction enters its Execute stage, we can go ahead and decode the second instruction.
- But now the instruction memory is free again, so we can fetch the third instruction!

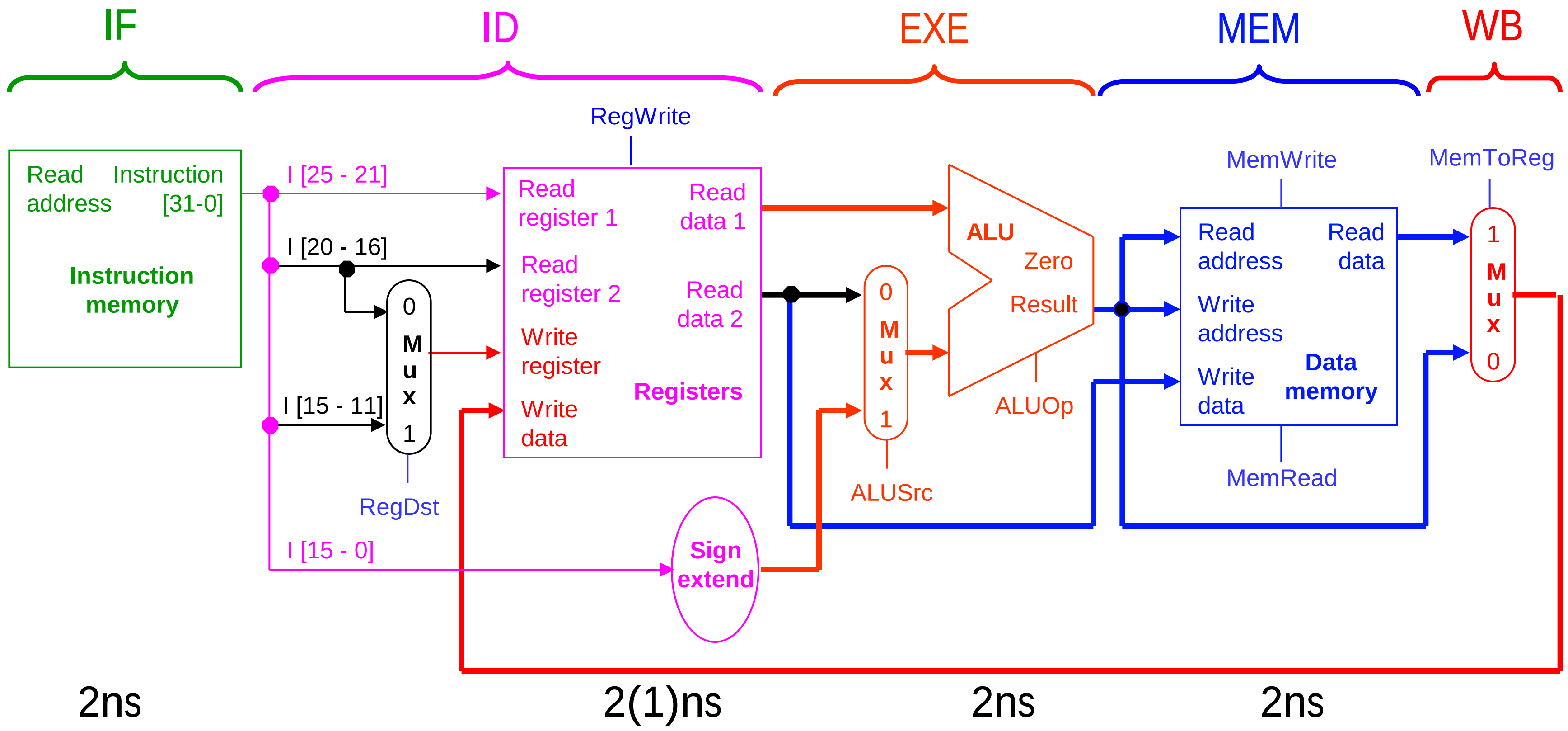


Making Pipelining Work

- We'll make our pipeline 5 stages long, to handle load instructions
  - Stages are: IF, ID, EX, MEM, and WB
- We want to support executing 5 instructions simultaneously: one in each stage.

Break datapath into 5 stages

- Each stage has its own functional units.
- Each stage can execute in 2ns



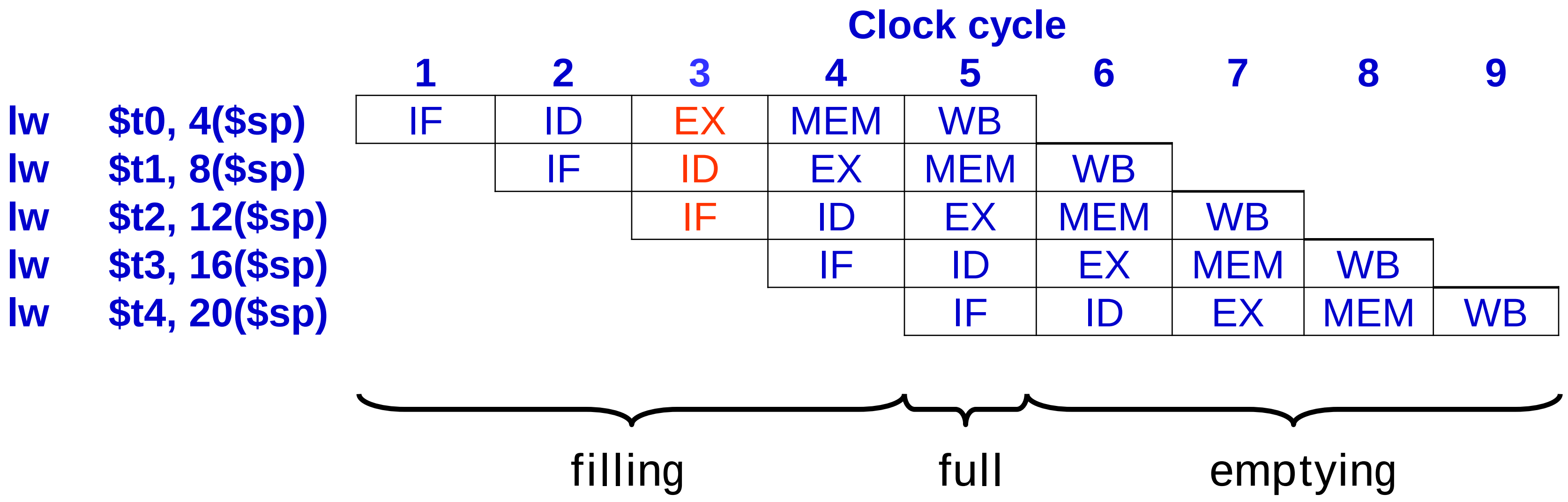
Pipelining Loads

		Clock cycle								
		1	2	3	4	5	6	7	8	9
lw	$t0, 4($sp)	IF	ID	EX	MEM	WB				
lw	$t1, 8($sp)		IF	ID	EX	MEM	WB			
lw	$t2, 12($sp)			IF	ID	EX	MEM	WB		
lw	$t3, 16($sp)				IF	ID	EX	MEM	WB	
lw	$t4, 20($sp)					IF	ID	EX	MEM	WB

- **A pipeline diagram shows the execution of a series of instructions.**
  - The instruction sequence is shown vertically, from top to bottom.
  - Clock cycles are shown horizontally, from left to right.
  - Each instruction is divided into its component stages. (We show five stages for every instruction, which will make the control unit easier.)

- **This clearly indicates the overlapping of instructions. For example, there are three instructions active in the third cycle above.**
  - The “lw $t0” instruction is in its Execute stage.
  - Simultaneously, the “lw $t1” is in its Instruction Decode stage.
  - Also, the “lw $t2” instruction is just being fetched.

Pipelining terminology



	Clock cycle 1	2	3	4	5	6	7	8	9
lw $t0, 4($sp)	IF	ID	EX	MEM	WB				
lw $t1, 8($sp)		IF	ID	EX	MEM	WB			
lw $t2, 12($sp)			IF	ID	EX	MEM	WB		
lw $t3, 16($sp)				IF	ID	EX	MEM	WB	
lw $t4, 20($sp)					IF	ID	EX	MEM	WB

- The **pipeline depth** is the number of stages—in this case, five.
- In the first four cycles here, the pipeline is **filling**, since there are unused functional units.
- In cycle 5, the pipeline is **full**. Five instructions are being executed simultaneously, so all hardware units are in use.
- In cycles 6-9, the pipeline is **emptying**.

Pipelining Performance

		Clock cycle								
		1	2	3	4	5	6	7	8	9
lw	$t0, 4($sp)	IF	ID	EX	MEM	WB				
lw	$t1, 8($sp)		IF	ID	EX	MEM	WB			
lw	$t2, 12($sp)			IF	ID	EX	MEM	WB		
lw	$t3, 16($sp)				IF	ID	EX	MEM	WB	
lw	$t4, 20($sp)					IF	ID	EX	MEM	WB

$\underbrace{\text{cycles 1–4}}_{\text{filling}}$

- **Execution time on ideal pipeline:**
  - time to fill the pipeline + one cycle per instruction
  - How long for N instructions?

- **Compared to single-cycle design, how much faster is pipelining for N=1000 ?**

Pipeline Datapath: Resource Requirements

		Clock cycle								
		1	2	3	4	5	6	7	8	9
lw	$t0, 4($sp)	IF	ID	EX	MEM	WB				
lw	$t1, 8($sp)		IF	ID	EX	MEM	WB			
lw	$t2, 12($sp)			IF	ID	EX	MEM	WB		
lw	$t3, 16($sp)				IF	ID	EX	MEM	WB	
lw	$t4, 20($sp)					IF	ID	EX	MEM	WB

- **We need to perform several operations in the same cycle.**
  - Increment the PC and add registers at the same time.
  - Fetch one instruction while another one reads or writes data.

- **What does that mean for our hardware?**

Pipelining other instruction types

- **R-type instructions only require 4 stages: IF, ID, EX, and WB**
  - We don't need the MEM stage
- **What happens if we try to pipeline loads with R-type instructions?**

Clock cycle

		1	2	3	4	5	6	7	8	9
add	$sp, $sp, -4	IF	ID	EX	WB					
sub	$v0, $a0, $a1		IF	ID	EX	WB				
lw	$t0, 4($sp)			IF	ID	EX	MEM	WB		
or	$s0, $s1, $s2				IF	ID	EX	WB		
lw	$t1, 8($sp)					IF	ID	EX	MEM	WB

  - Load uses Register File's Write Port during its 5th (cycle 7) stage
  - R-type uses Register File's Write Port during its 4th (cycle 7) stage

A solution: Insert NOP stages

- **Enforce uniformity**
  - Make all instructions take 5 cycles.
  - Make them have the same stages, in the same order
    - Some stages will do nothing for some instructions

R-type	IF	ID	EX	NOP	WB

Clock cycle

		1	2	3	4	5	6	7	8	9
add	$sp, $sp, -4	IF	ID	EX	NOP	WB				
sub	$v0, $a0, $a1		IF	ID	EX	NOP	WB			
lw	$t0, 4($sp)			IF	ID	EX	MEM	WB		
or	$s0, $s1, $s2				IF	ID	EX	NOP	WB	
lw	$t1, 8($sp)					IF	ID	EX	MEM	WB

- Stores and Branches have NOP stages, too…

store	IF	ID	EX	MEM	NOP
branch	IF	ID	EX	NOP	NOP

What we have so far

- Pipelining attempts to maximize instruction throughput by overlapping the execution of multiple instructions.
- Pipelining offers amazing speedup.
  - In the best case, one instruction finishes on every cycle, and the speedup is equal to the pipeline depth.
- The pipeline datapath is much like the single-cycle one, but with added pipeline registers
  - Each stage needs its own functional units
- Next we'll see the datapath and control, and walk through an example execution.

Pipelined Datapath and Control

- We'll see a basic implementation of a pipelined processor.
  - The datapath and control unit share similarities with the single-cycle implementations that we already saw.
  - An example execution highlights important pipelining concepts.
- In future lectures, we'll discuss several complications of pipelining that we're hiding from you for now.

Pipelining Concepts

- **A pipelined processor allows multiple instructions to execute at once, and each instruction uses a different functional unit in the datapath.**
- **This increases throughput, so programs can run faster.**
  - **One instruction can finish executing on every clock cycle, and simpler stages also lead to shorter cycle times.**

		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

Pipelined Datapath

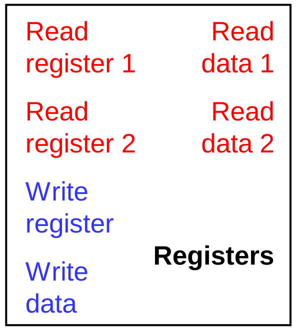
- The whole point of pipelining is to allow multiple instructions to execute at the same time.
- We may need to perform several operations in the same cycle.
  - Increment the PC and add registers at the same time.
  - Fetch one instruction while another one reads or writes data.

		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

- Thus, like the single-cycle datapath, a pipelined processor will need to duplicate hardware elements that are needed several times in the same clock cycle.

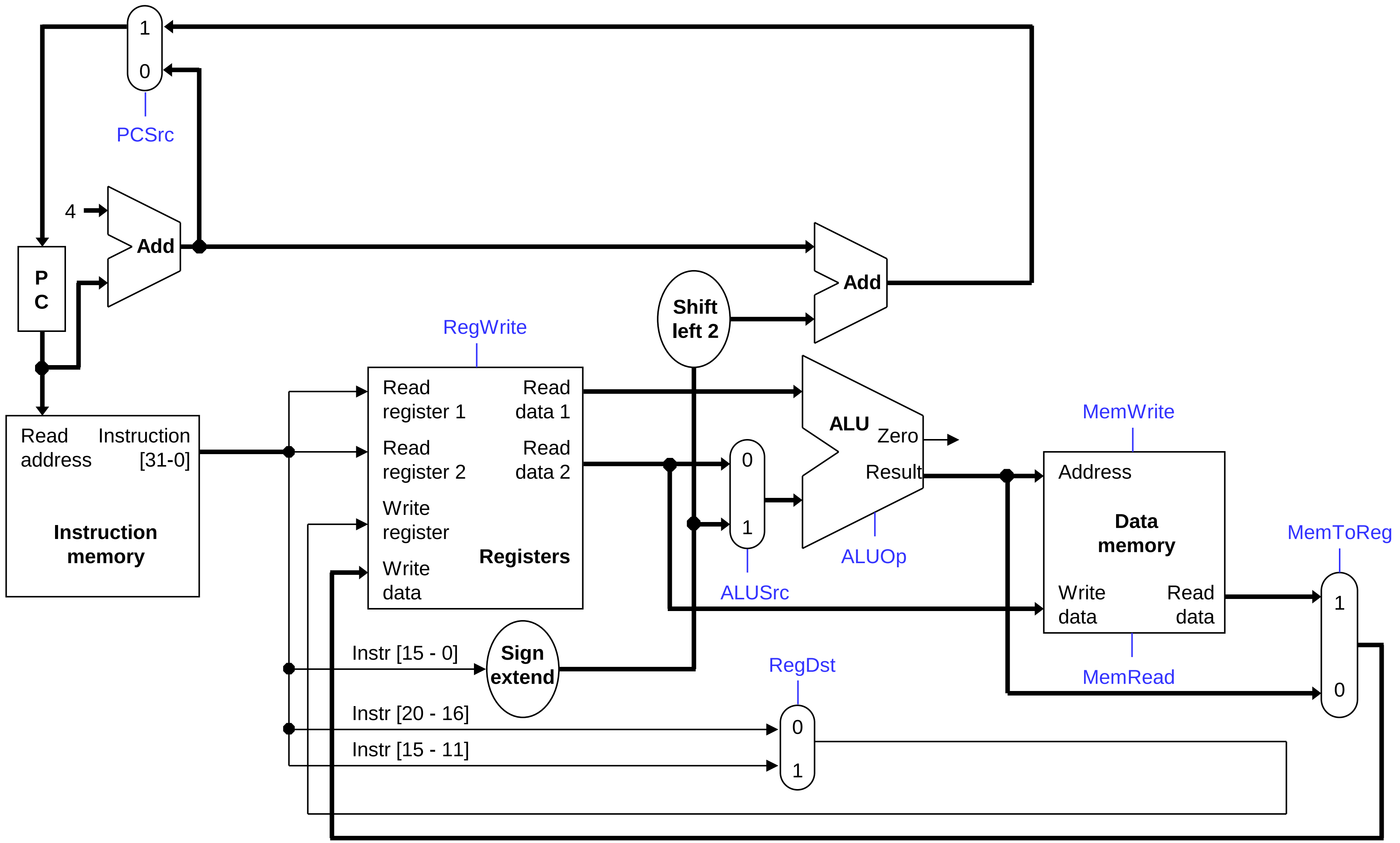
One register file is enough

- We need only one register file to support both the ID and WB stages.



- Reads and writes go to separate ports on the register file.
- We already took advantage of this property in our single-cycle CPU.

Single-cycle datapath, slightly rearranged



Multiple cycles

- **In pipelining, we also divide instruction execution into multiple cycles.**
- **Information computed during one cycle may be needed in a later cycle.**
  - The instruction read in the IF stage determines which registers are fetched in the ID stage, what constant is used for the EX stage, and what the destination register is for WB.
  - The registers read in ID are used in the EX and/or MEM stages.
  - The ALU output produced in the EX stage is an effective address for the MEM stage or a result for the WB stage.
- **We need to add several intermediate registers to datapath to preserve information between stages.**

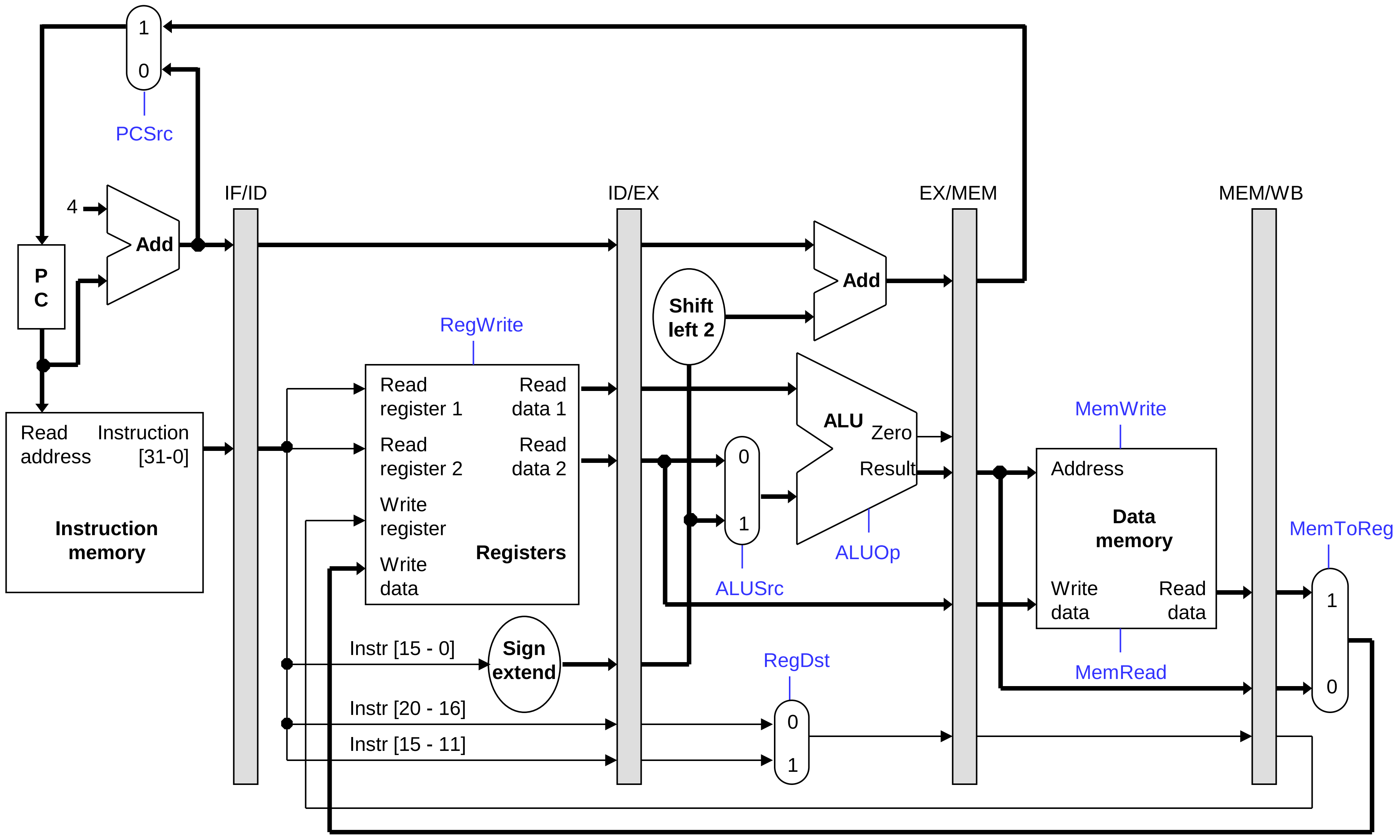
Pipeline registers

- There's a lot of information to save, however. We'll simplify our diagrams by drawing just one big pipeline register between each stage.
- 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.

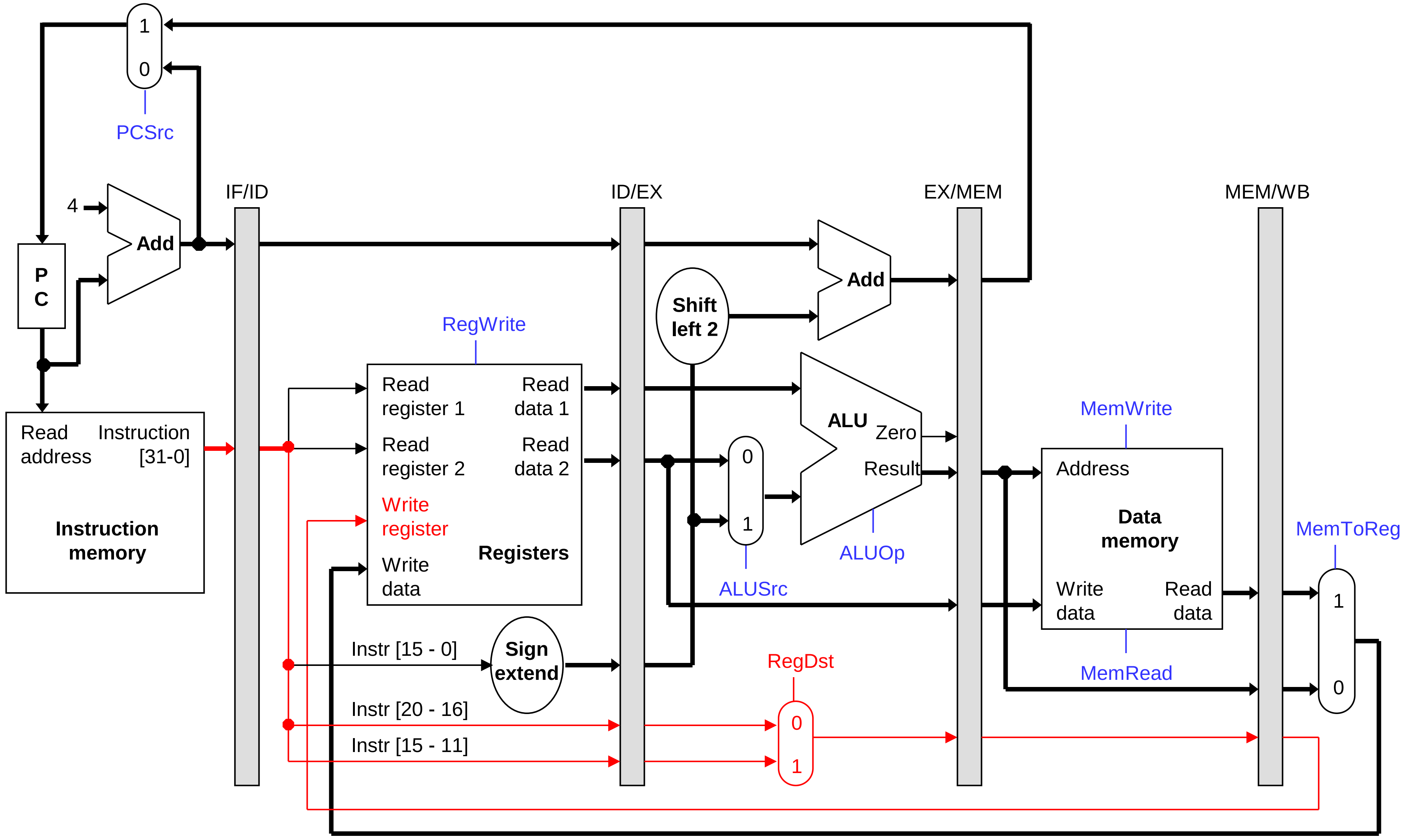
Pipelined datapath



Propagating values forward

- **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

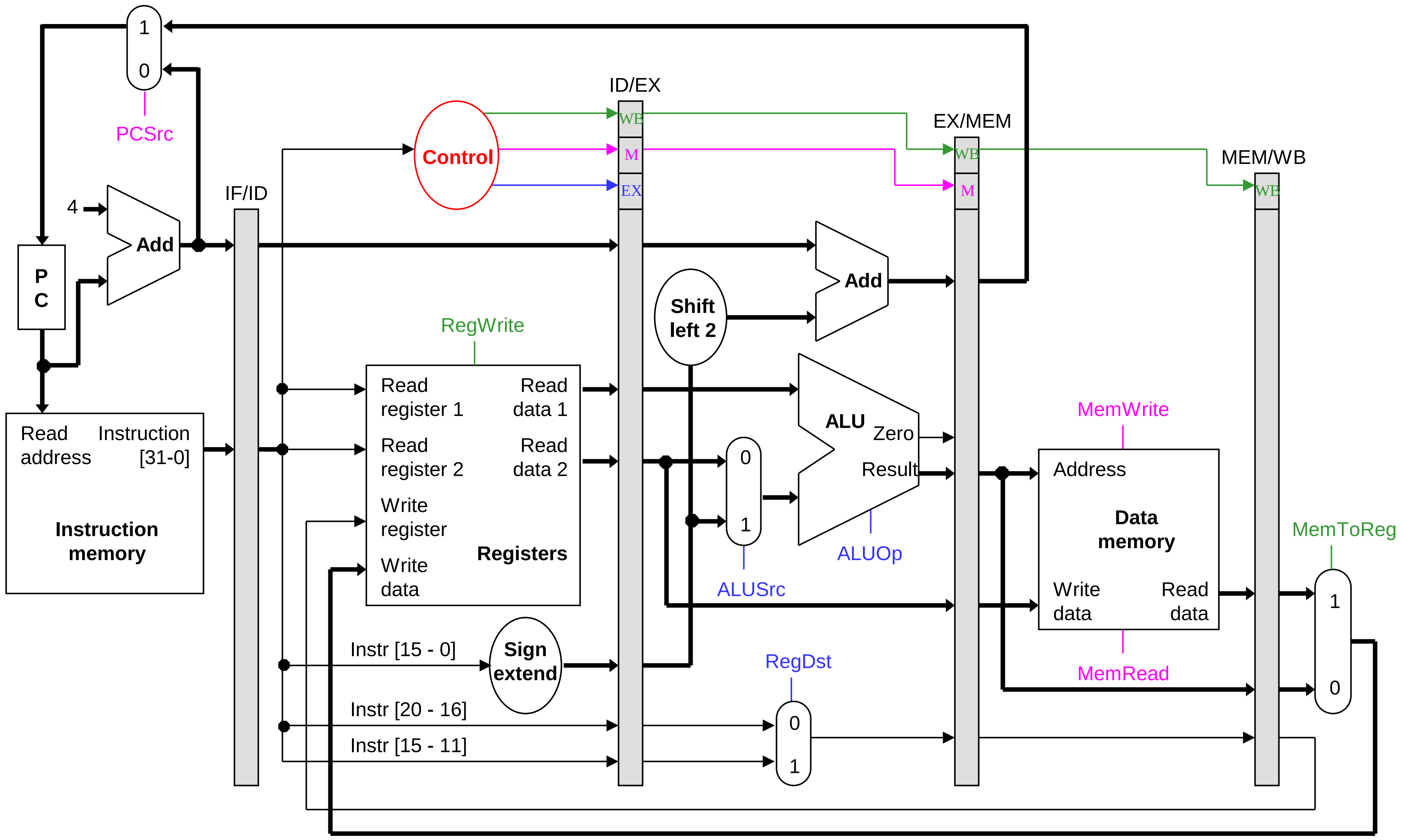


What about control signals?

- 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

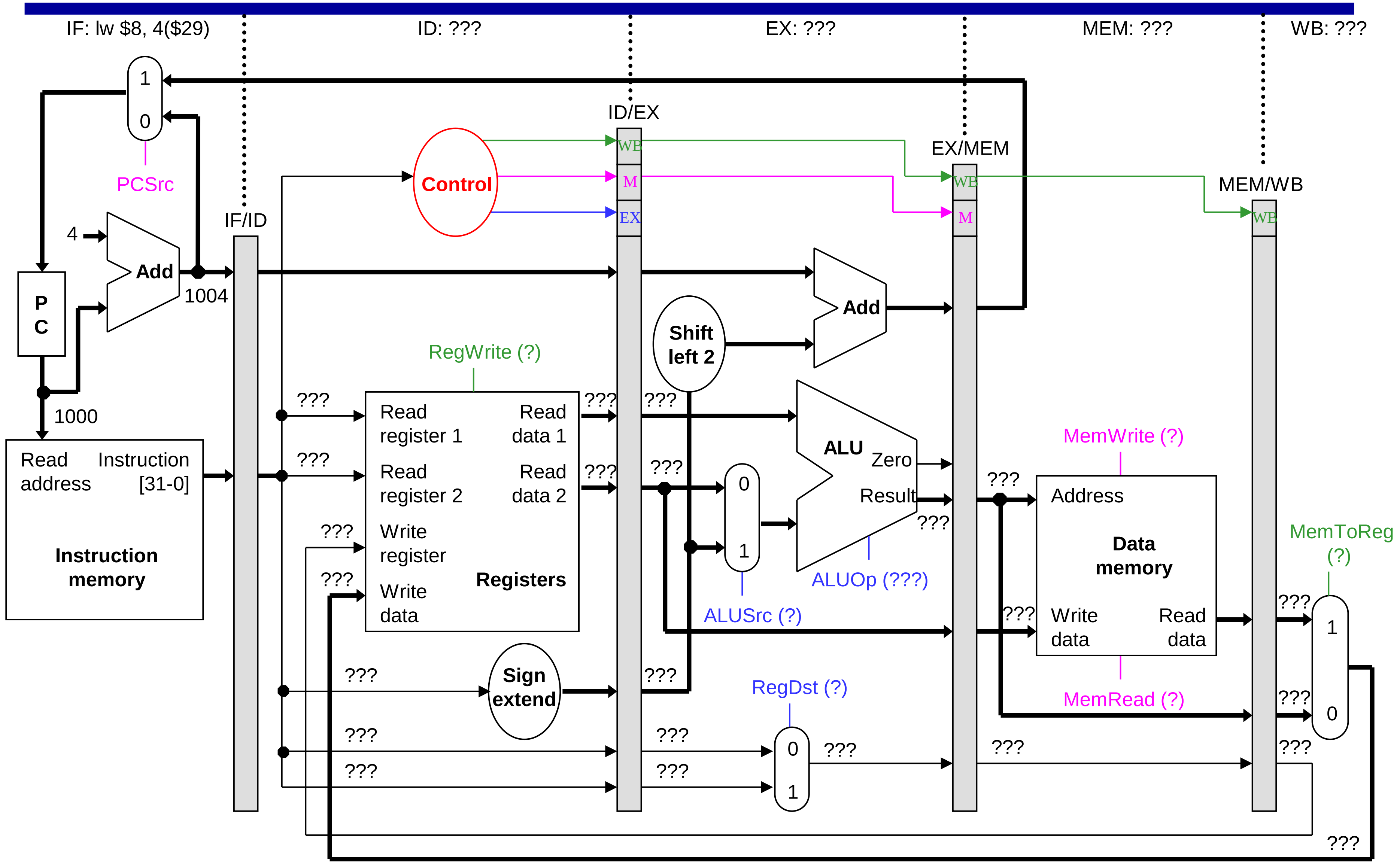
- Here’s a sample sequence of instructions to execute.

addresses in decimal

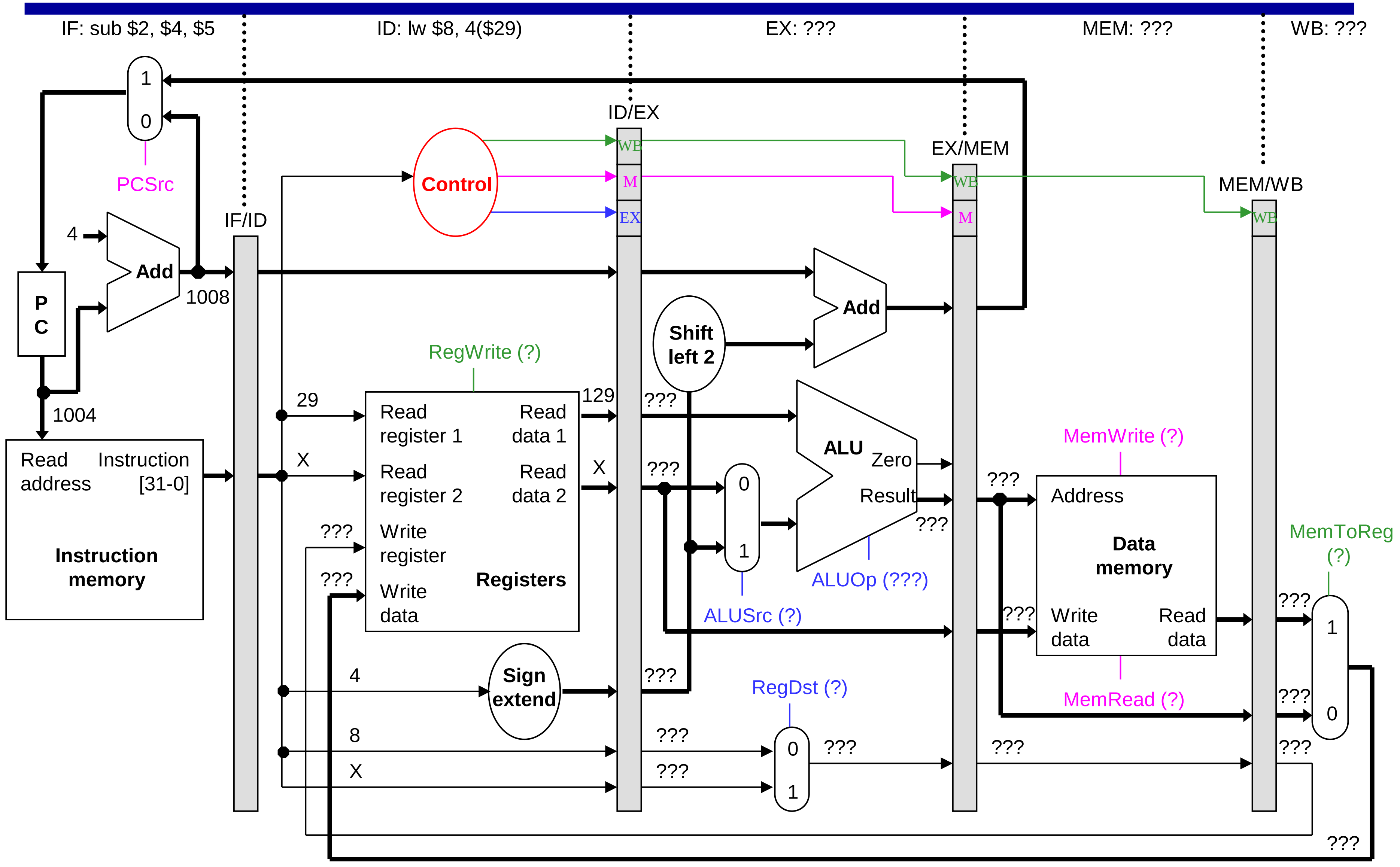
```
1000:   lw   $8, 4($29)
1004:   sub  $2, $4, $5
1008:   and  $9, $10, $11
1012:   or   $16, $17, $18
1016:   add  $13, $14, $0
```

- 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)

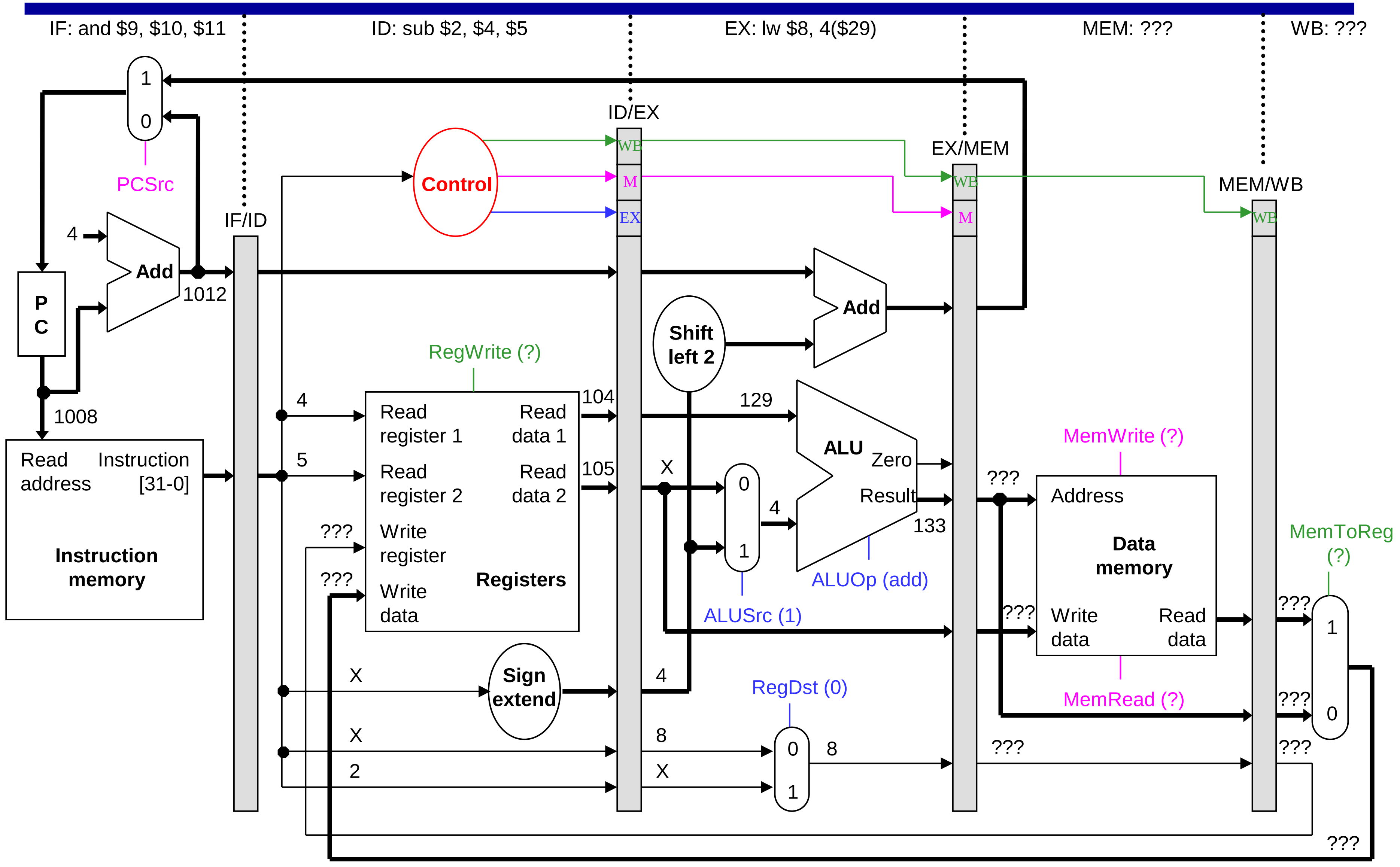


Cycle 2



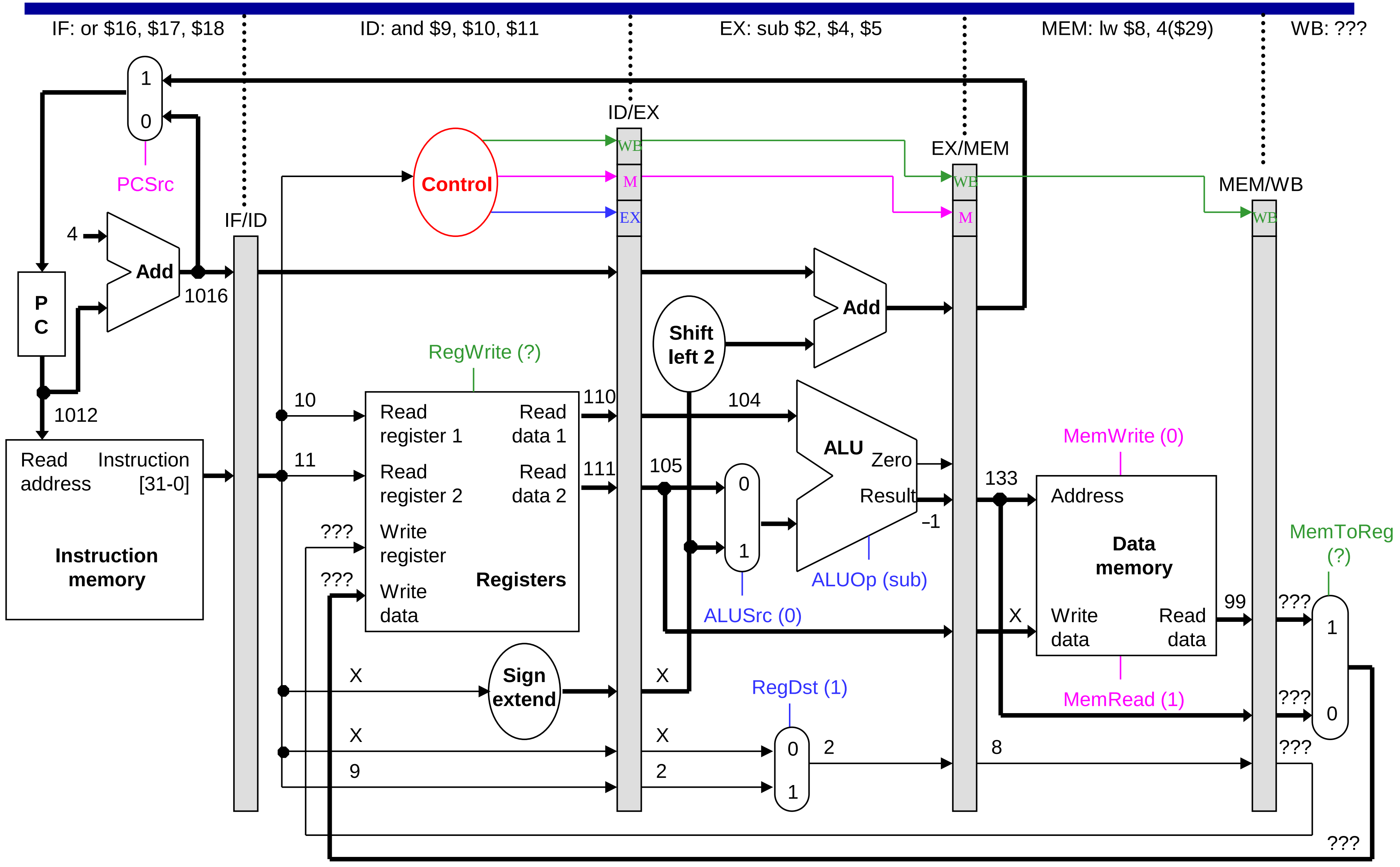
IF: sub $2, $4, $5
ID: lw $8, 4($29)
EX: ???
MEM: ???
WB: ???
1
0
PCSrc
IF/ID
ID/EX
EX/MEM
MEM/WB
Control
WB
M
EX
WB
M
WB
4
Add
1008
P C
1004
Read address
Instruction [31-0]
Instruction memory
RegWrite (?)
29
X
???
???
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
129
X
Shift left 2
Add
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

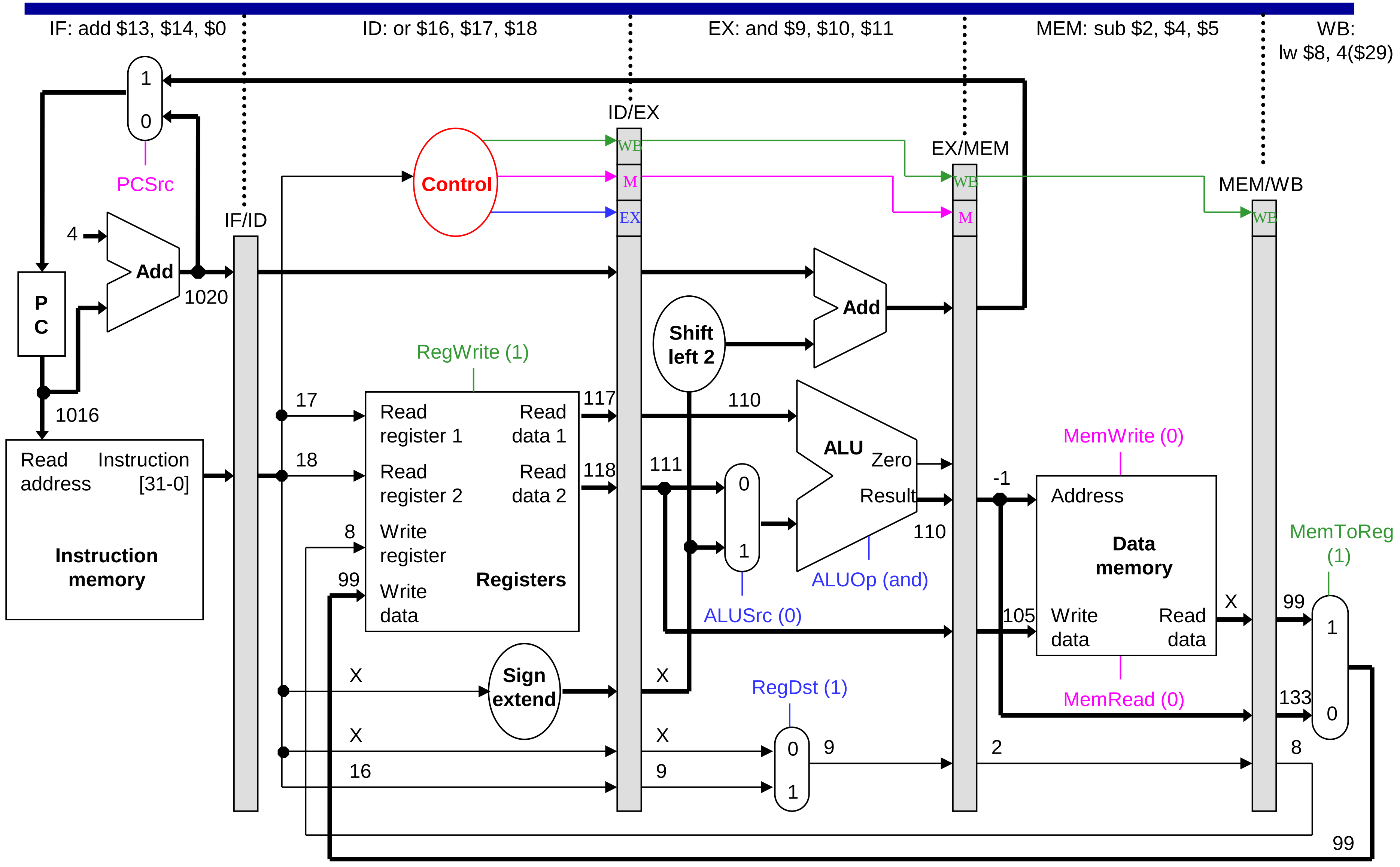


IF: and $9, $10, $11
ID: sub $2, $4, $5
EX: lw $8, 4($29)
MEM: ???
WB: ???
PCSrc
1
0
4
Add
1012
P C
1008
Read address
Instruction [31-0]
Instruction memory
IF/ID
Control
ID/EX
WB
M
EX
EX/MEM
MEM/WB
RegWrite (?)
4
5
???
???
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
104
105
Shift left 2
Add
129
X
ALU
Zero
Result
133
ALUOp (add)
ALUSrc (1)
MemWrite (?)
Address
Data memory
Write data
Read data
MemRead (?)
MemToReg (?)
Sign extend
X
X
2
8
X
RegDst (0)
8
???

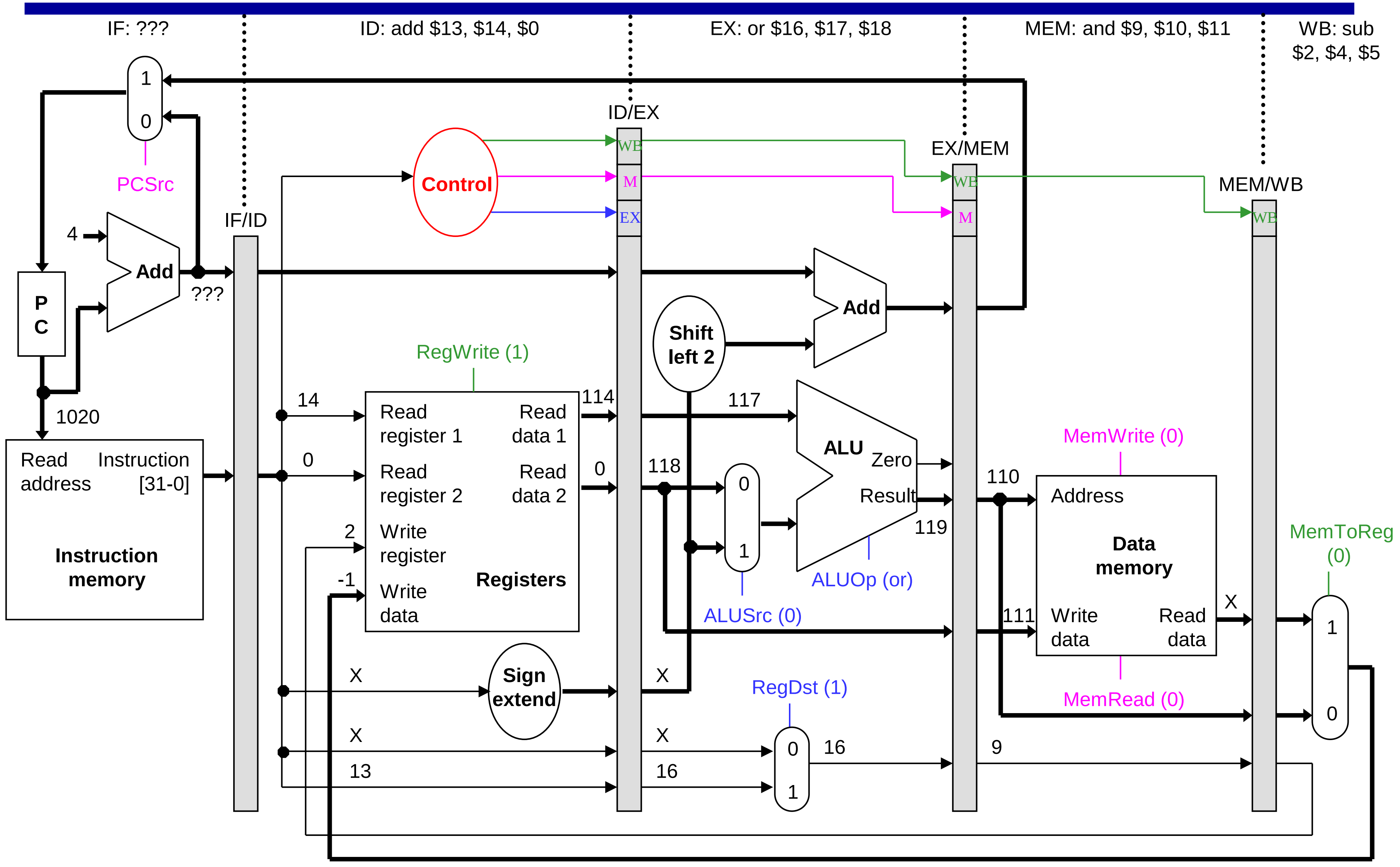
Cycle 4


IF: or $16, $17, $18
ID: and $9, $10, $11
EX: sub $2, $4, $5
MEM: lw $8, 4($29)
WB: ???
1
0
PCSrc
IF/ID
ID/EX
EX/MEM
MEM/WB
Control
WB
M
EX
WB
M
WB
4
Add
P C
1016
1012
Read address
Instruction [31-0]
Instruction memory
RegWrite (?)
10
11
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
???
???
110
111
104
105
Shift left 2
Add
ALU
Zero
Result
-1
ALUSrc (0)
ALUOp (sub)
0
1
133
MemWrite (0)
Address
Data memory
Write data
Read data
MemRead (1)
X
99
MemToReg (?)
???
???
???
???
Sign extend
X
X
X
X
9
2
RegDst (1)
0
1
2
8

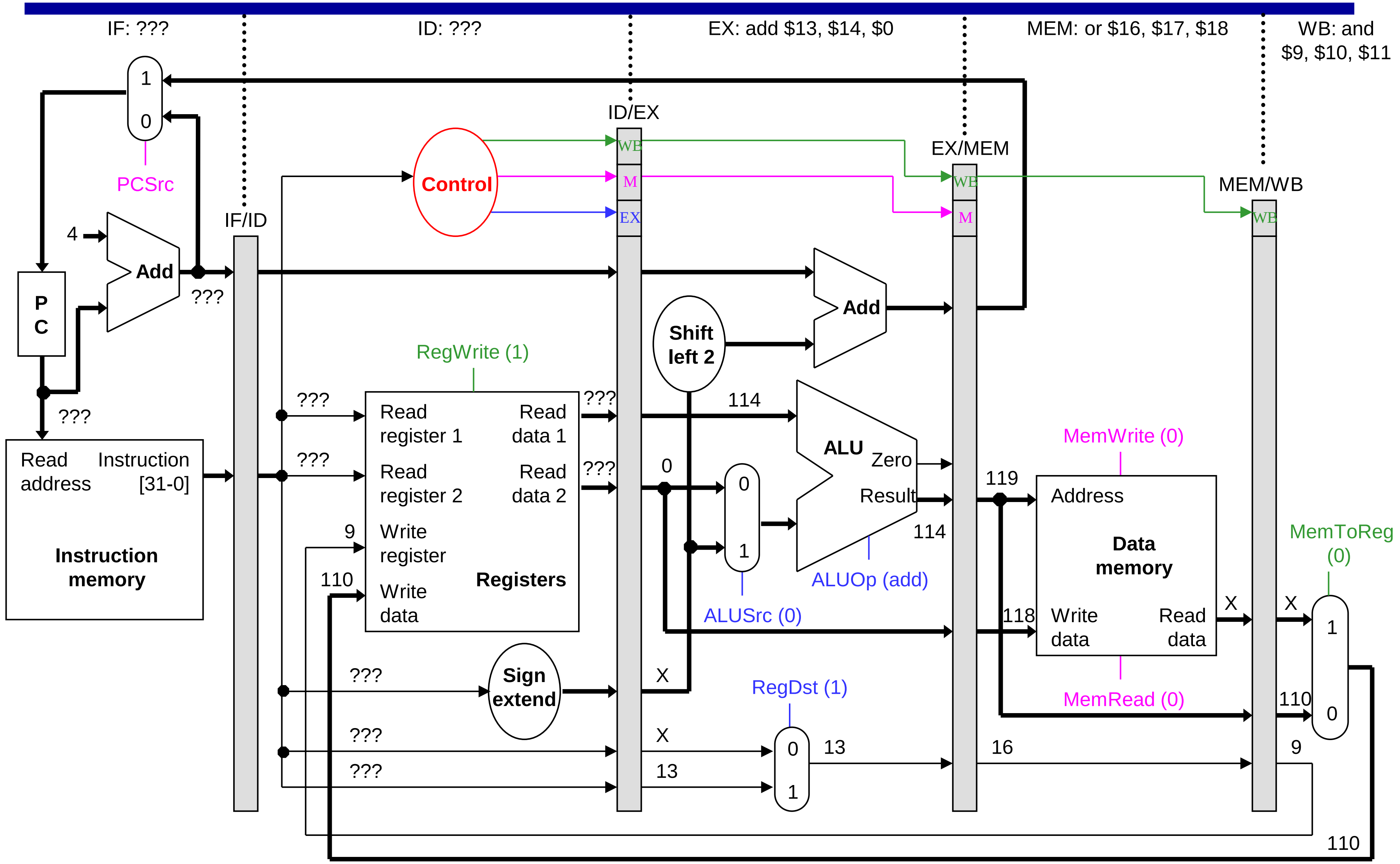
Cycle 5 (full)


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
IF/ID
ID/EX
EX/MEM
MEM/WB
Control
WB
M
EX
WB
M
WB
4
Add
1020
P C
1016
Read address
Instruction [31-0]
Instruction memory
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
Shift left 2
Add
110
111
0
1
ALUSrc (0)
ALU
Zero
Result
110
ALUOp (and)
-1
MemWrite (0)
Address
Data memory
Write data
Read data
105
MemRead (0)
X
99
133
MemToReg (1)
1
0
X
Sign extend
X
X
X
16
9
RegDst (1)
0
1
9
2
8
99

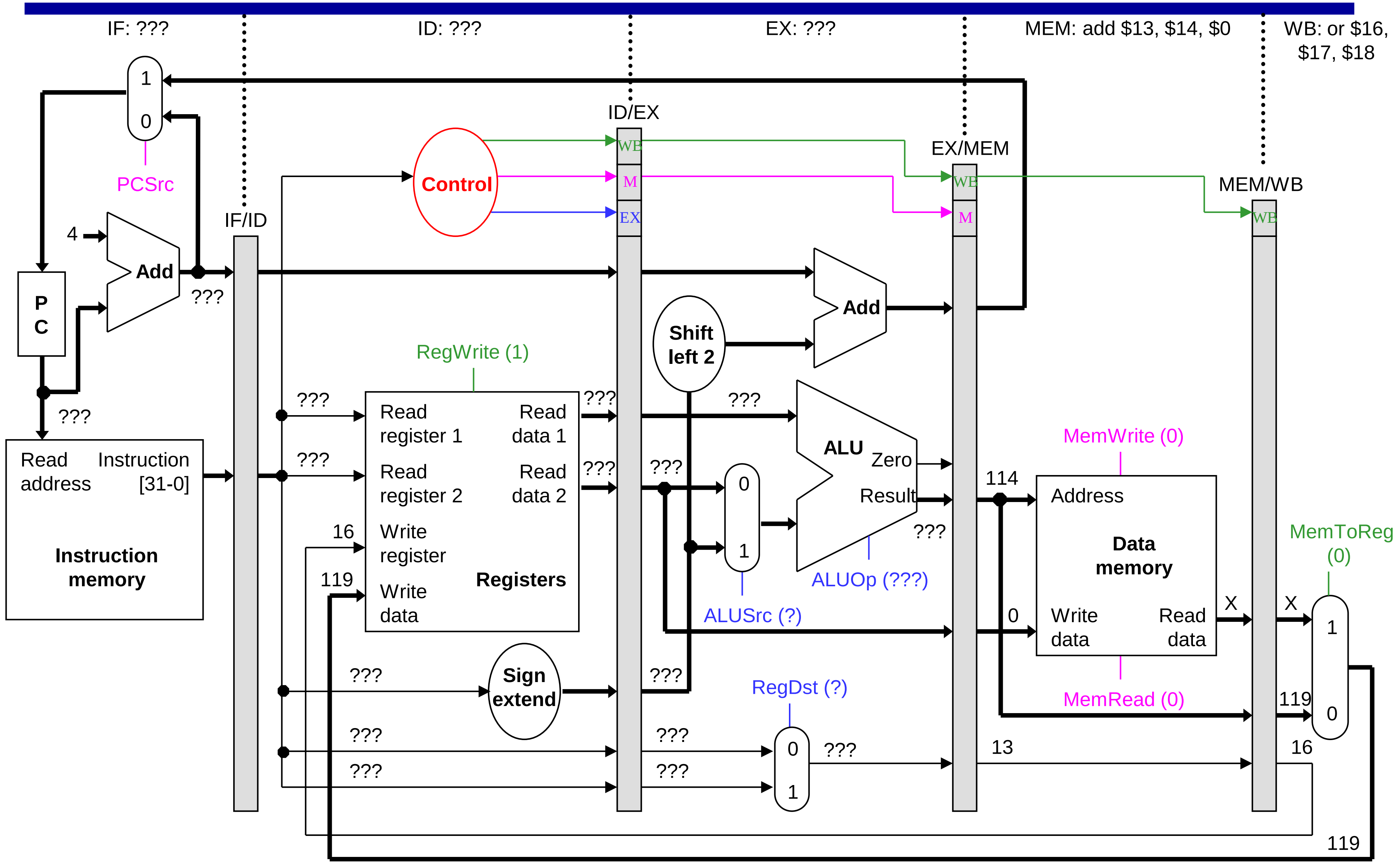
Cycle 6 (emptying)



Cycle 7

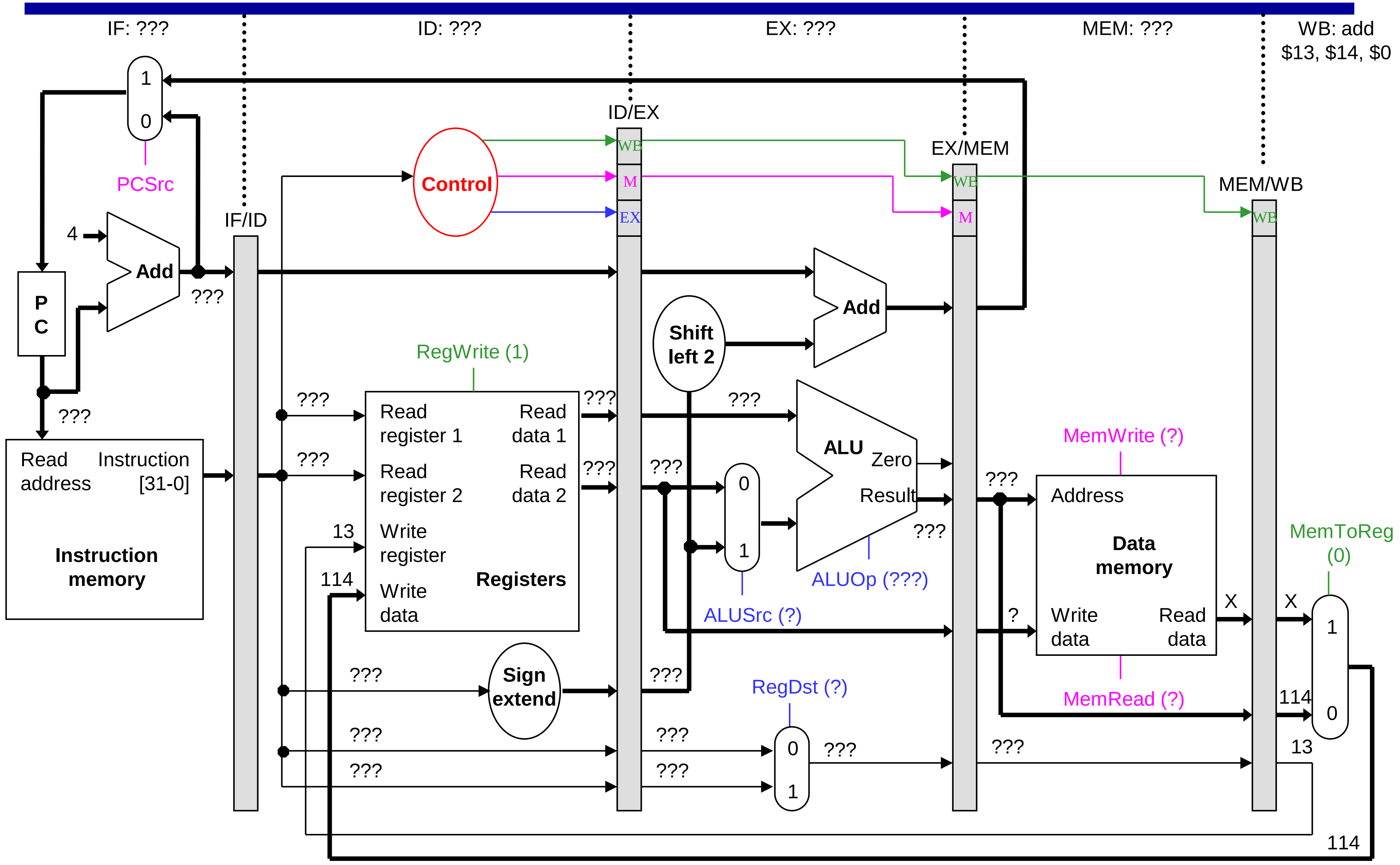

IF: ???
ID: ???
EX: add $13, $14, $0
MEM: or $16, $17, $18
WB: and $9, $10, $11
IF/ID
ID/EX
EX/MEM
MEM/WB
PCSrc
Control
WB
M
EX
4
Add
???
P C
Read address
Instruction [31-0]
Instruction memory
RegWrite (1)
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
9
110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

Cycle 8



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

Cycle 9



IF: ???
ID: ???
EX: ???
MEM: ???
WB: add $13, $14, $0
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
RegWrite (1)
???
???
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
13
114
???
???
Shift left 2
Add
???
???
0
1
ALU
Zero
Result
???
ALUSrc (?)
ALUOp (???)
MemWrite (?)
???
Address
Data memory
Write data
Read data
?
X
X
MemRead (?)
MemToReg (0)
1
0
114
13
114
Sign extend
???
???
???
???
???
???
RegDst (?)
0
1
???
???

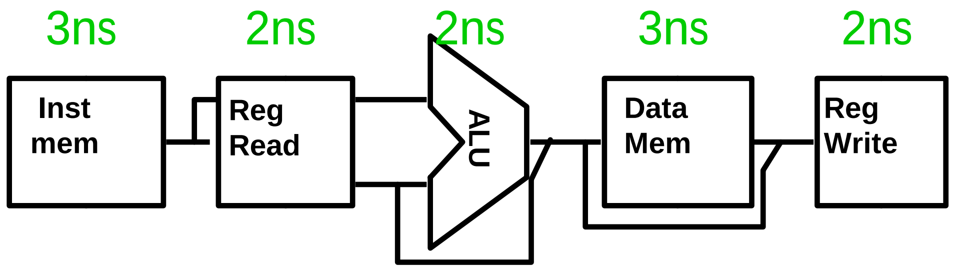
That’s a lot of diagrams there

		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.**

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

		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	$sp, $sp, -4					IF	ID	EX	MEM	WB

- **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

						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, $s2**				IF	ID	EX	MEM	WB	
add	**$sp, $sp, -4**					IF	ID	EX	MEM	WB

- **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.

Summary so far

- The pipelined datapath uses multiple memories and ALUs.
  - Instruction execution is split into several stages.
- 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.

So far, our examples are too simple

```
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)
```

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

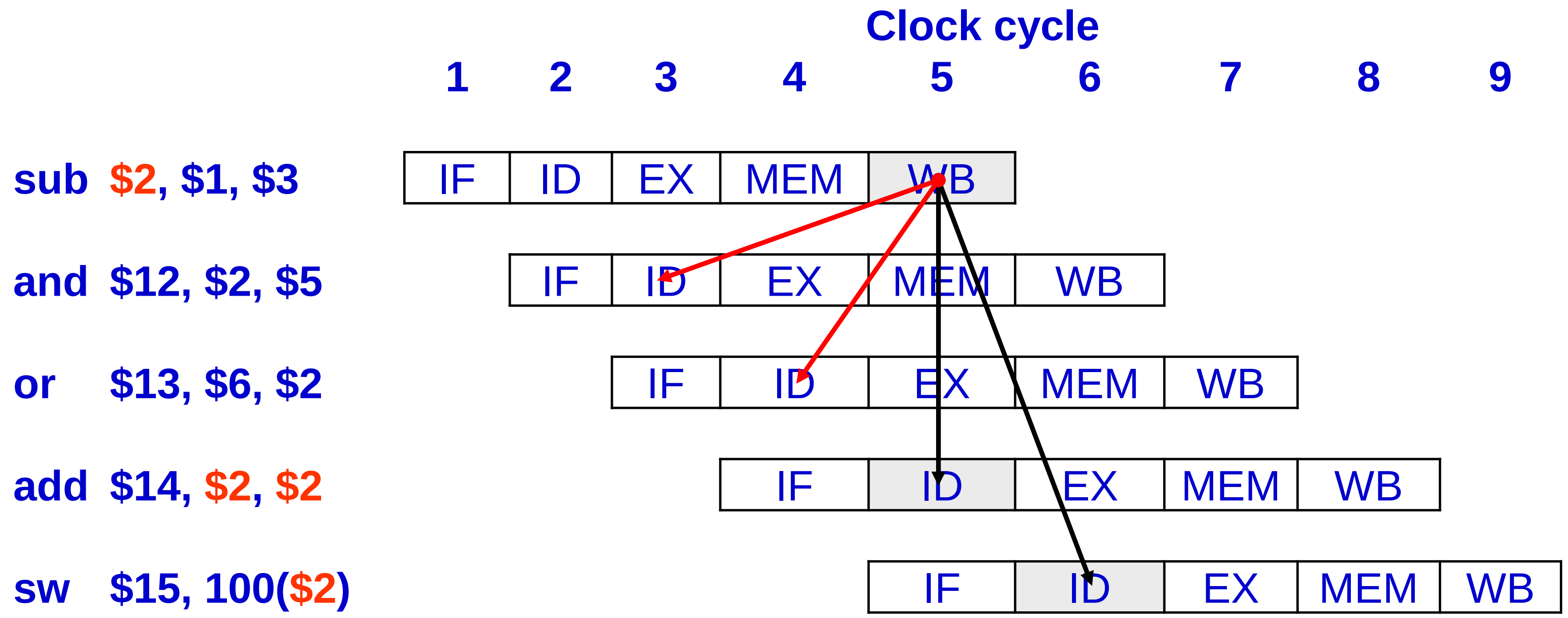
- The SUB instruction does not write to register $2 until clock cycle 5. This causes two data hazards in our current pipelined datapath.
  – The AND reads register $2 in cycle 3. Since SUB hasn't modified the register yet, this will be the *old* value of $2, not the new one.
  – Similarly, the OR instruction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

Things that are okay

	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

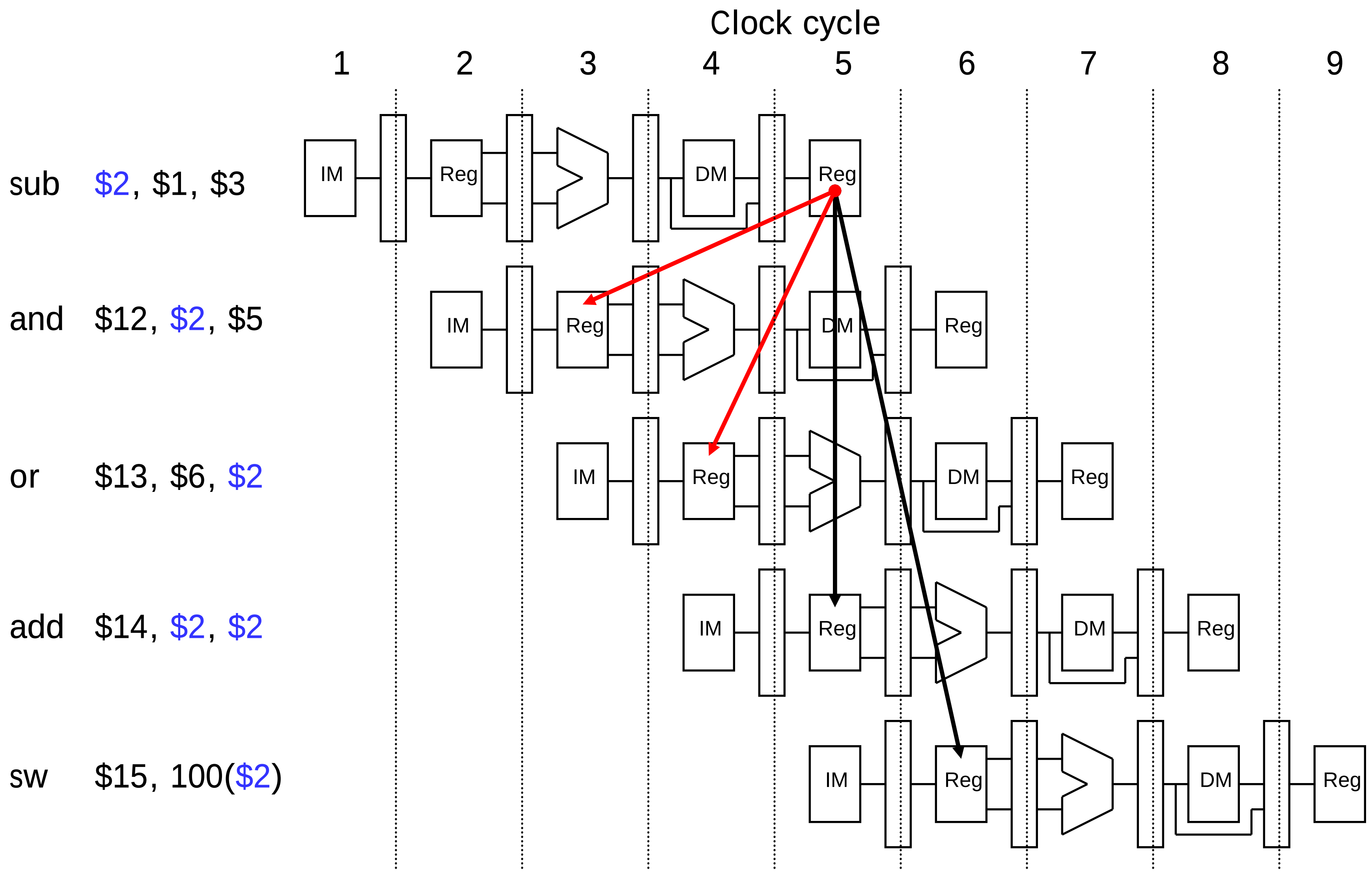
- **The ADD instruction is okay, because of the register file design.**
  - **Registers are written at the beginning of a clock cycle.**
  - **The new value will be available by the end of that cycle.**
- **The SW is no problem at all, since it reads $2 after the SUB finishes.**

Dependency arrows



- 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.

A fancier pipeline diagram



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.

Clock cycle

	1	2	3	4	5	6	7
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

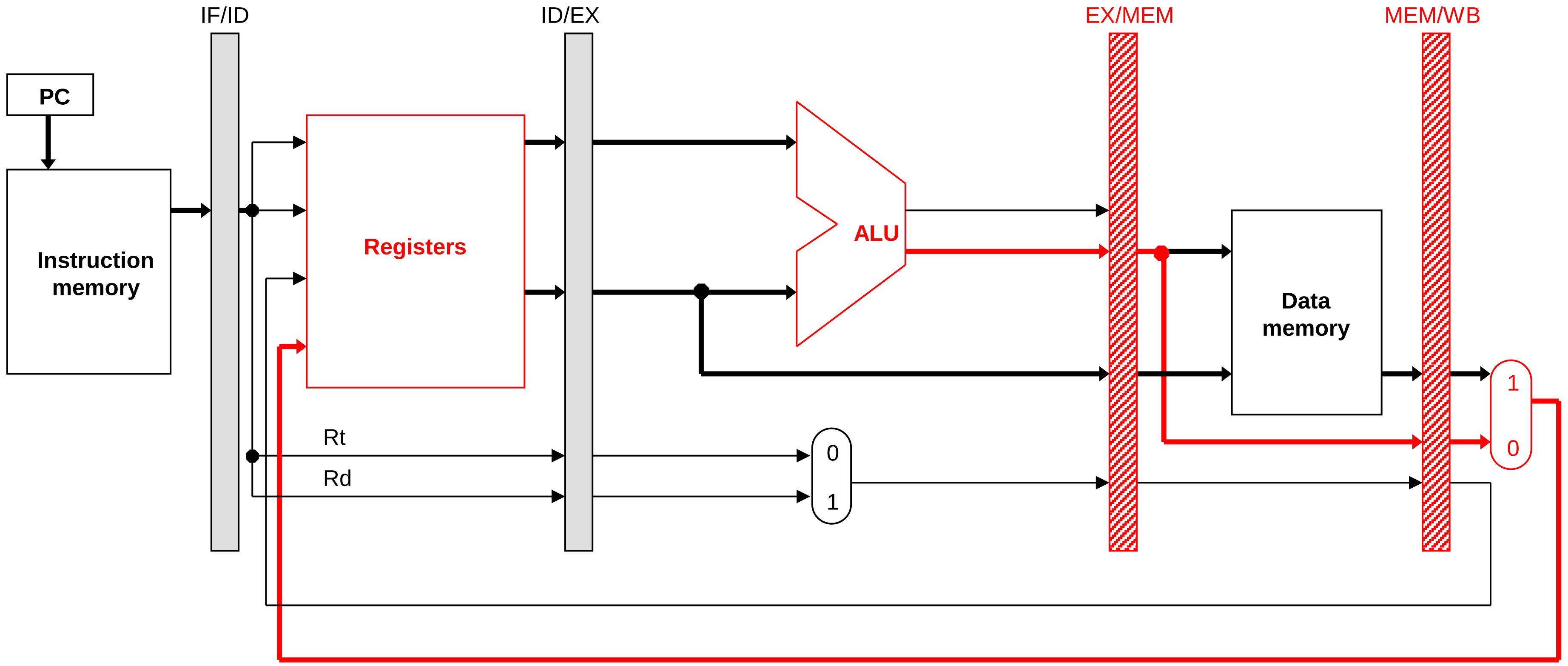
Bypassing the register file

- 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.

	Clock cycle						
	1	2	3	4	5	6	7
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

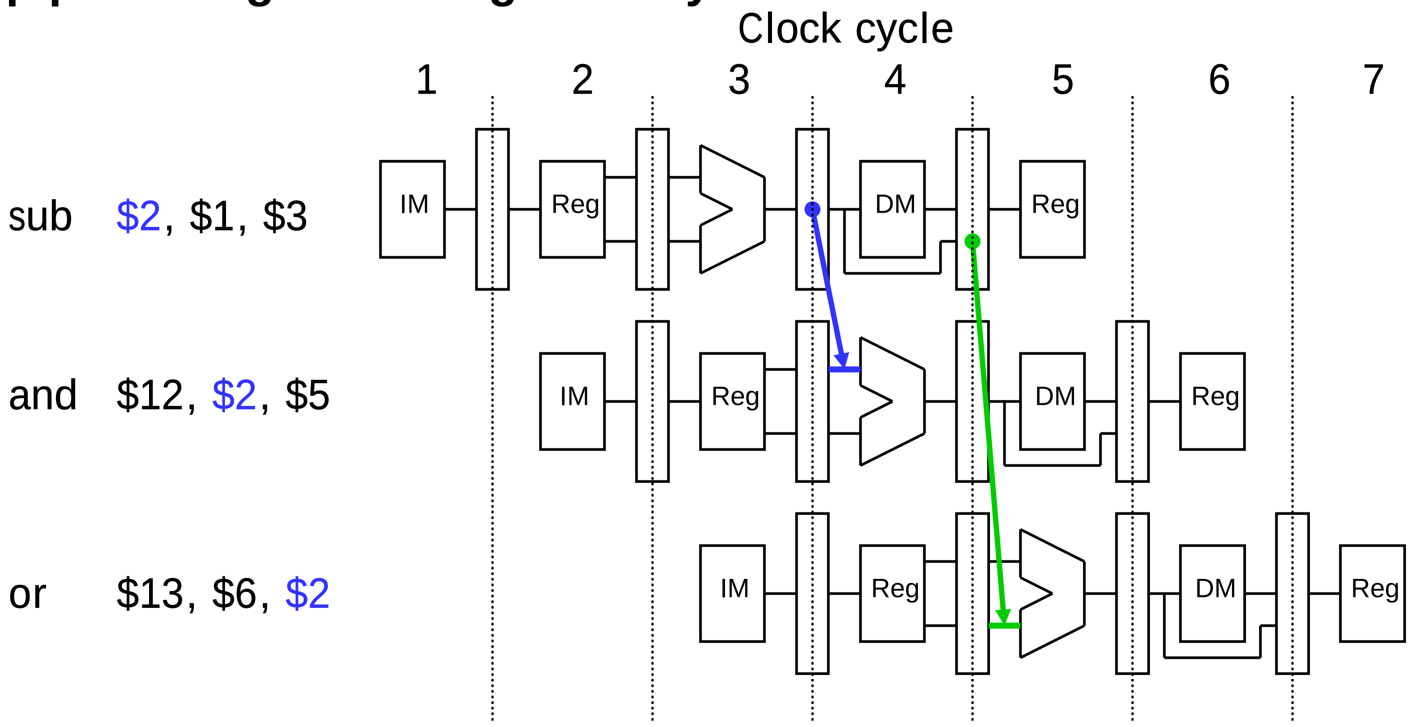
Where to find the ALU result

- 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.



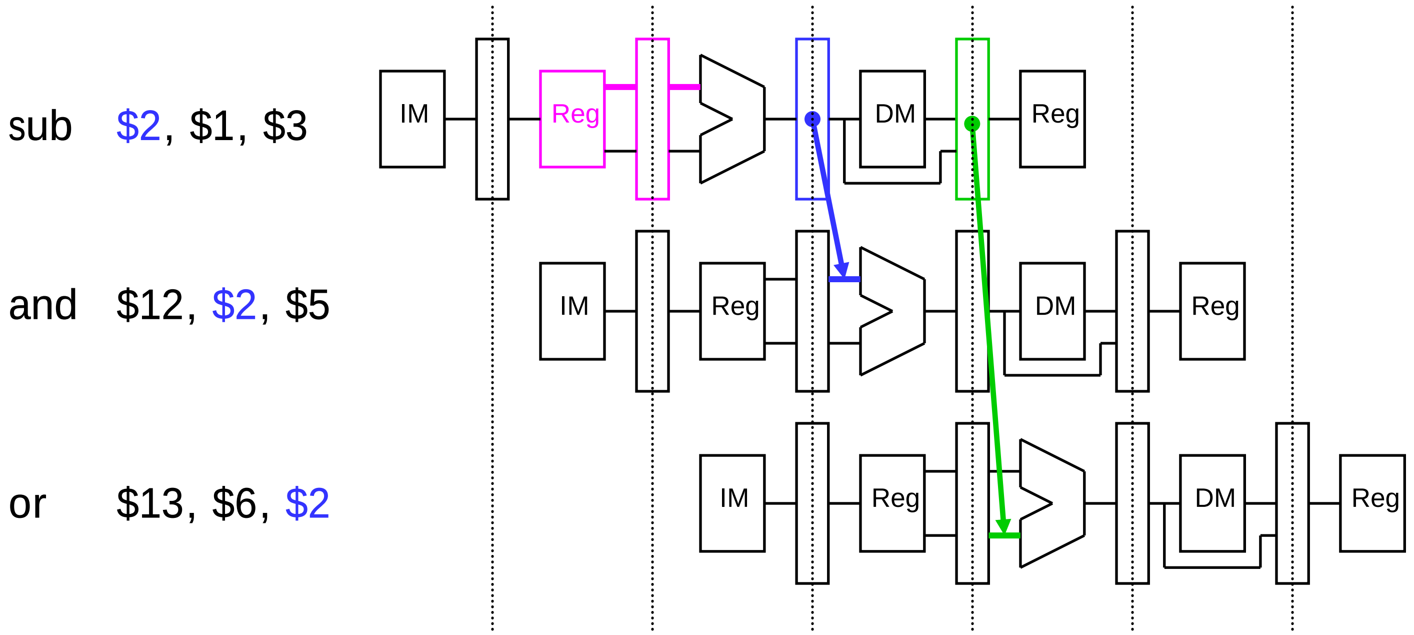
Forwarding

- 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.

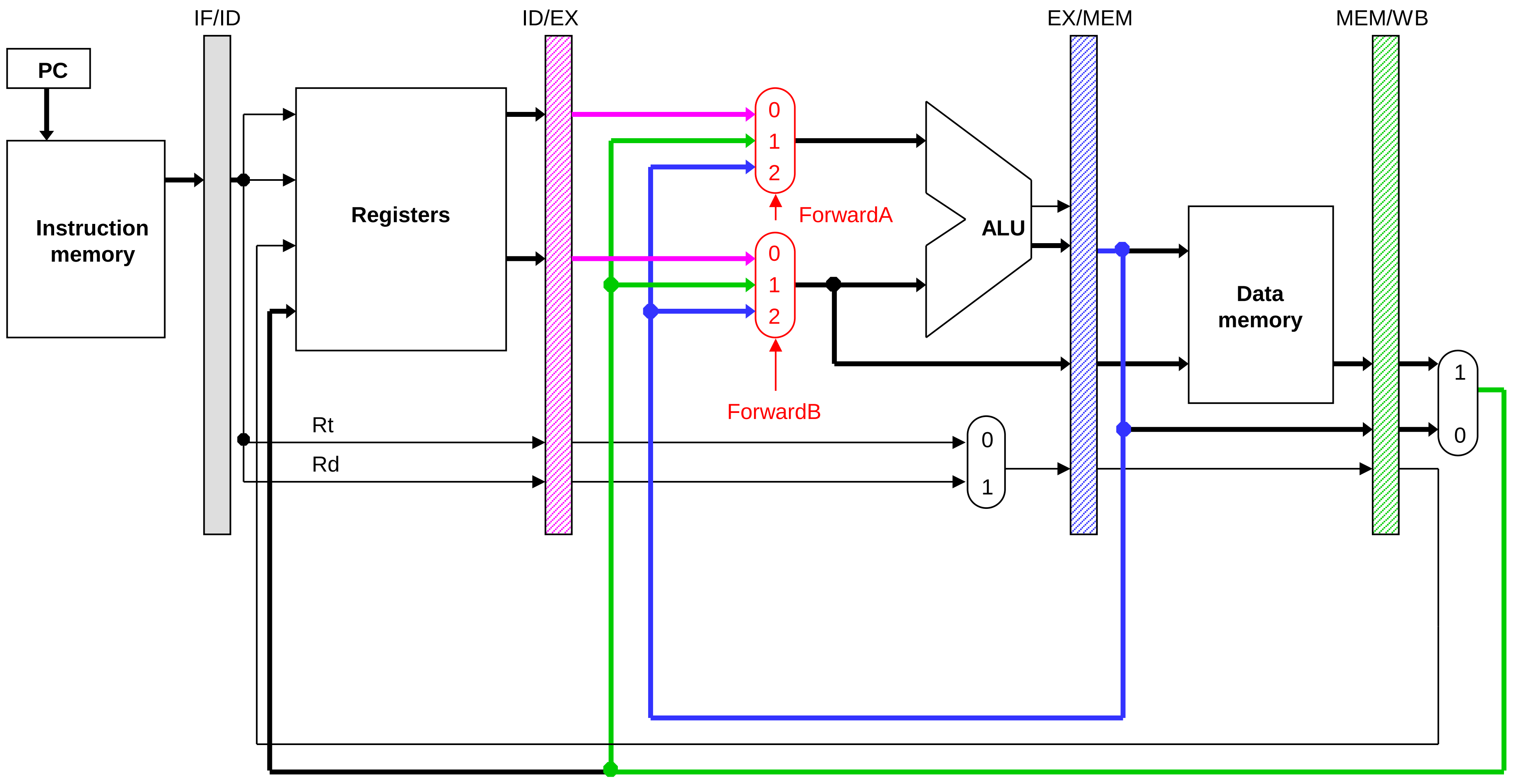


Outline of forwarding hardware

- 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

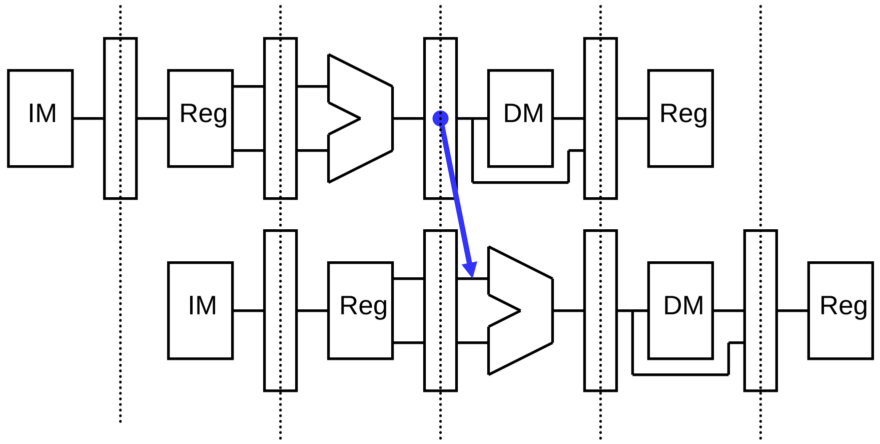


Detecting EX/MEM data hazards

- 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.

sub $2, $1, $3

and $12, $2, $5



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

EX/MEM data hazard equations

- The first ALU source comes from the pipeline register when necessary.

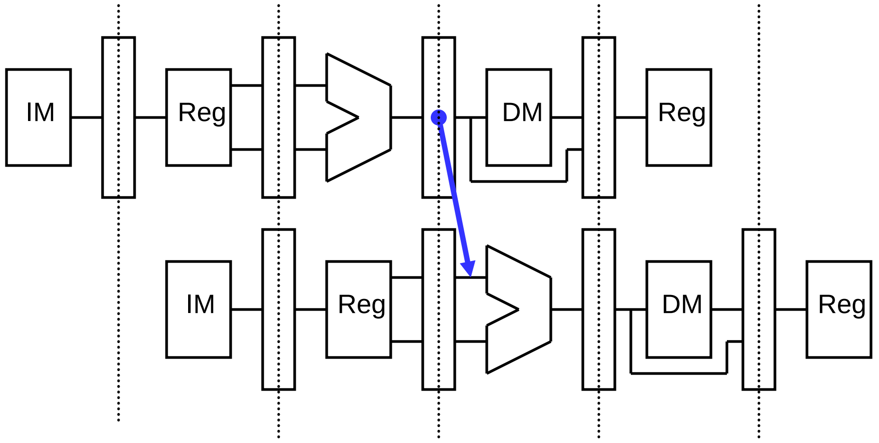
```
if (EX/MEM.RegWrite = 1
    and EX/MEM.Rd = ID/EX.Rs)
then ForwardA = 2
```

- The second ALU source is similar.

```
if (EX/MEM.RegWrite = 1
    and EX/MEM.Rd = ID/EX.Rt)
then ForwardB = 2
```

sub $2, $1, $3

and $12, $2, $5



Detecting MEM/WB data hazards

- A MEM/WB hazard may occur between an instruction in the EX stage and the instruction from *two* cycles ago.
- One new problem is if a register is updated twice in a row.

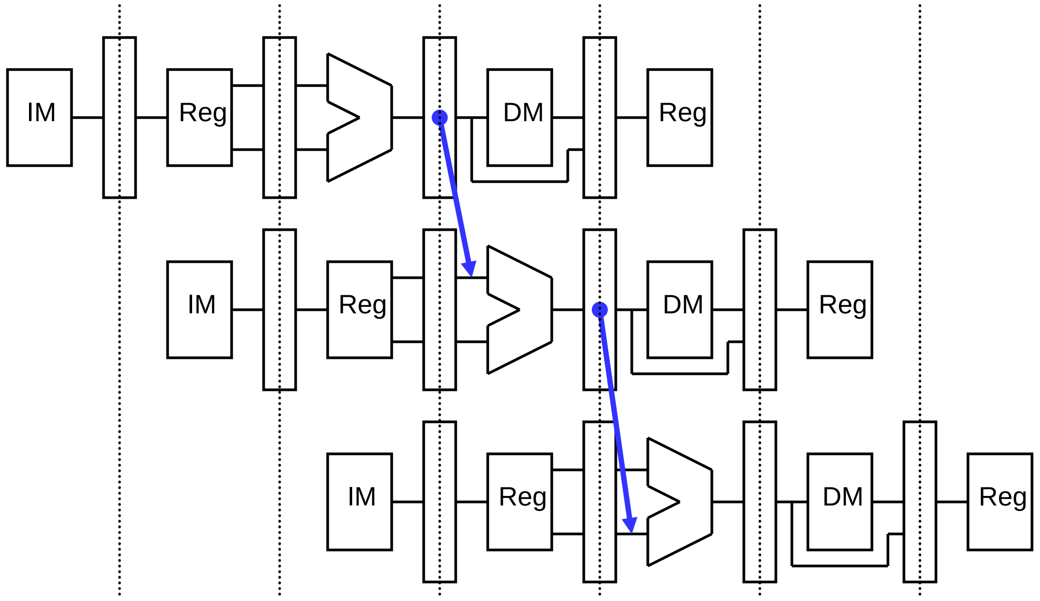
```
add   $1, $2, $3
add   $1, $1, $4
sub   $5, $5, $1
```

- Register $1 is written by *both* of the previous instructions, but only the most recent result (from the second ADD) should be forwarded.

add $1, $2, $3

add $1, $1, $4

sub $5, $5, $1



MEM/WB hazard equations

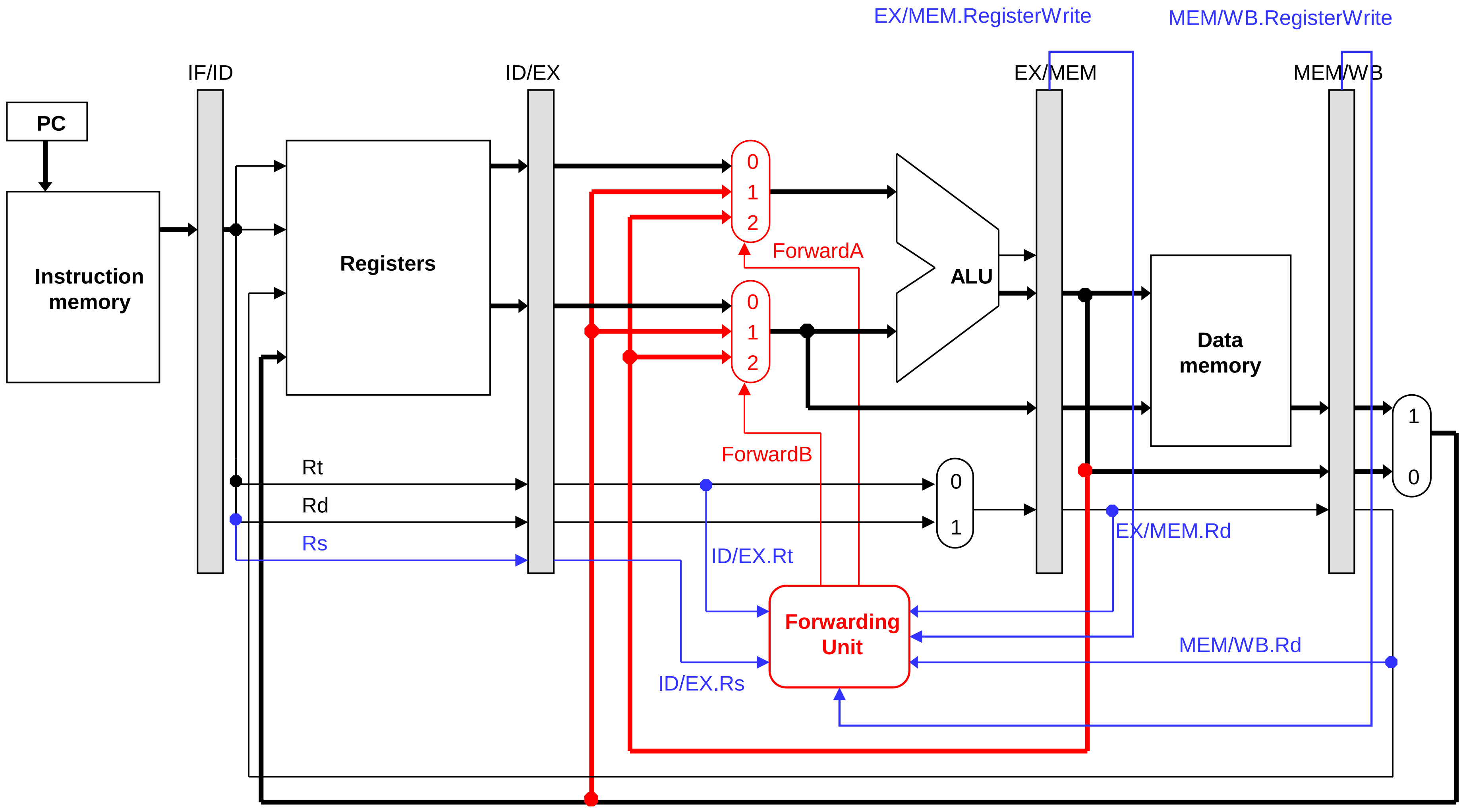
- Here is an equation for detecting and handling MEM/WB hazards for the first ALU source.

```
if (MEM/WB.RegWrite = 1
    and MEM/WB.Rd = ID/EX.Rs
    and (EX/MEM.Rd ≠ ID/EX.Rs or EX/MEM.RegWrite = 0)
then ForwardA = 1
```

- The second ALU operand is handled similarly.

```
if (MEM/WB.RegWrite = 1
    and MEM/WB.Rd = ID/EX.Rt
    and (EX/MEM.Rd ≠ ID/EX.Rt or EX/MEM.RegWrite = 0)
then ForwardB = 1
```

Simplified datapath with forwarding



The forwarding unit

- The forwarding unit has several control signals as inputs.

ID/EX.Rs	EX/MEM.Rd	MEM/WB.Rd
ID/EX.Rt	EX/MEM.RegWrite	MEM/WB.RegWrite

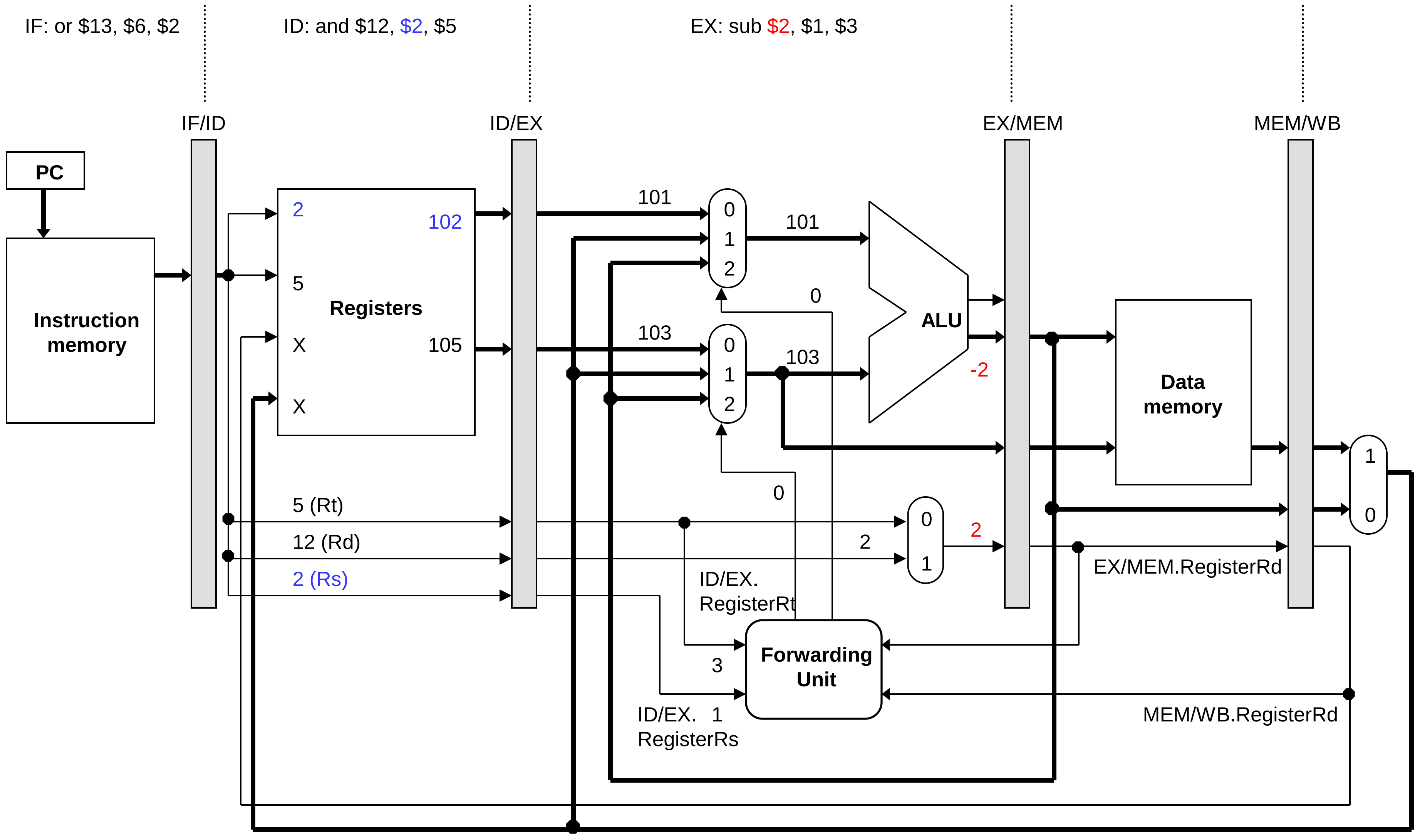
- The forwarding unit outputs are selectors for the ForwardA and ForwardB multiplexers attached to the ALU. These outputs are generated from the inputs using the equations on the previous pages.
- Some new buses route data from pipeline registers to the new muxes.

Example

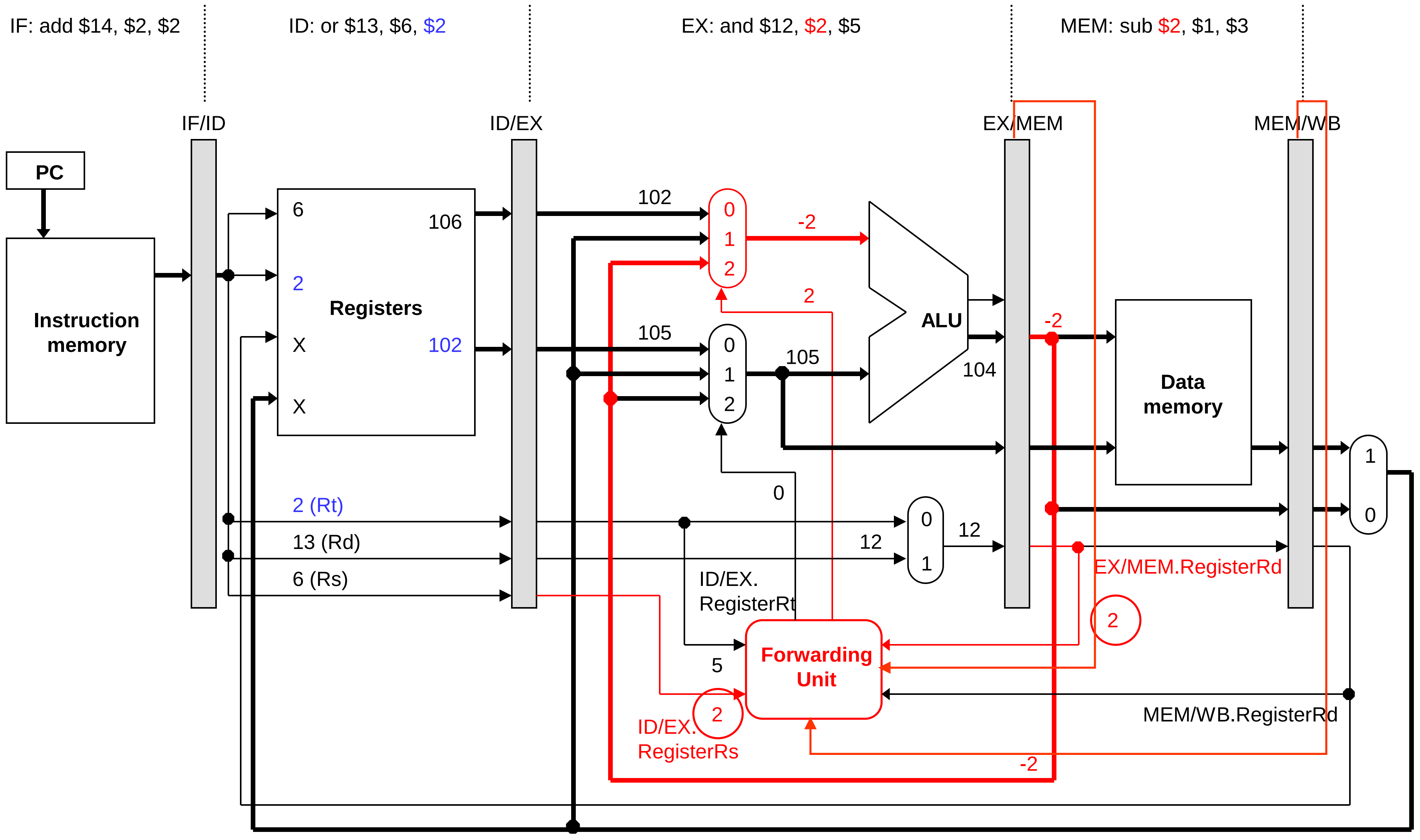
```
sub    $2, $1, $3
and    $12, $2, $5
or     $13, $6, $2
add    $14, $2, $2
sw     $15, 100($2)
```

- Assume again each register initially contains its number plus 100.
  - After the first instruction, $2 should contain -2 (101 - 103).
  - The other instructions should all use -2 as one of their operands.
- We'll try to keep the example short.
  - Assume no forwarding is needed except for register $2.
  - We'll skip the first two cycles, since they're the same as before.

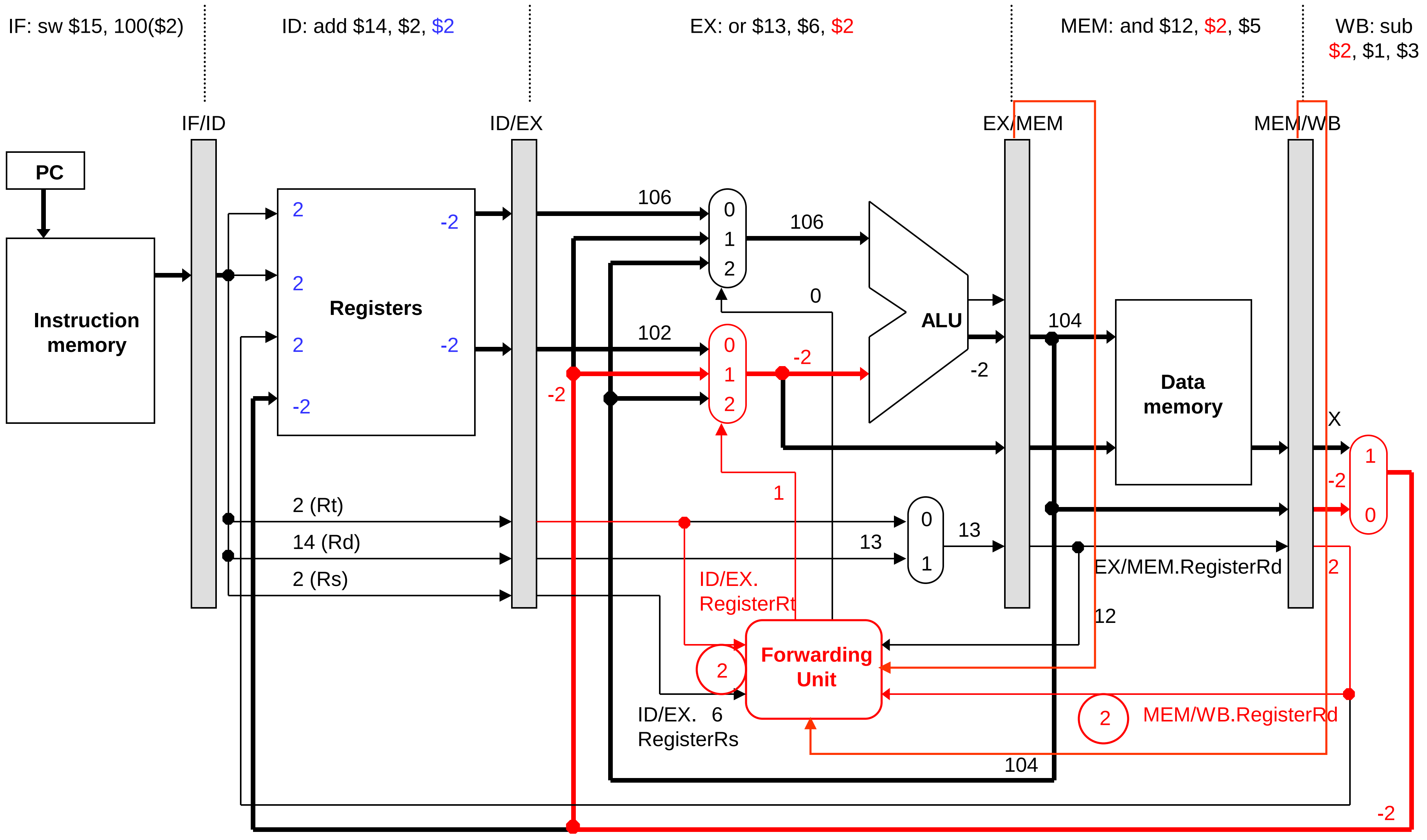
Clock cycle 3



Clock cycle 4: forwarding $2 from EX/MEM



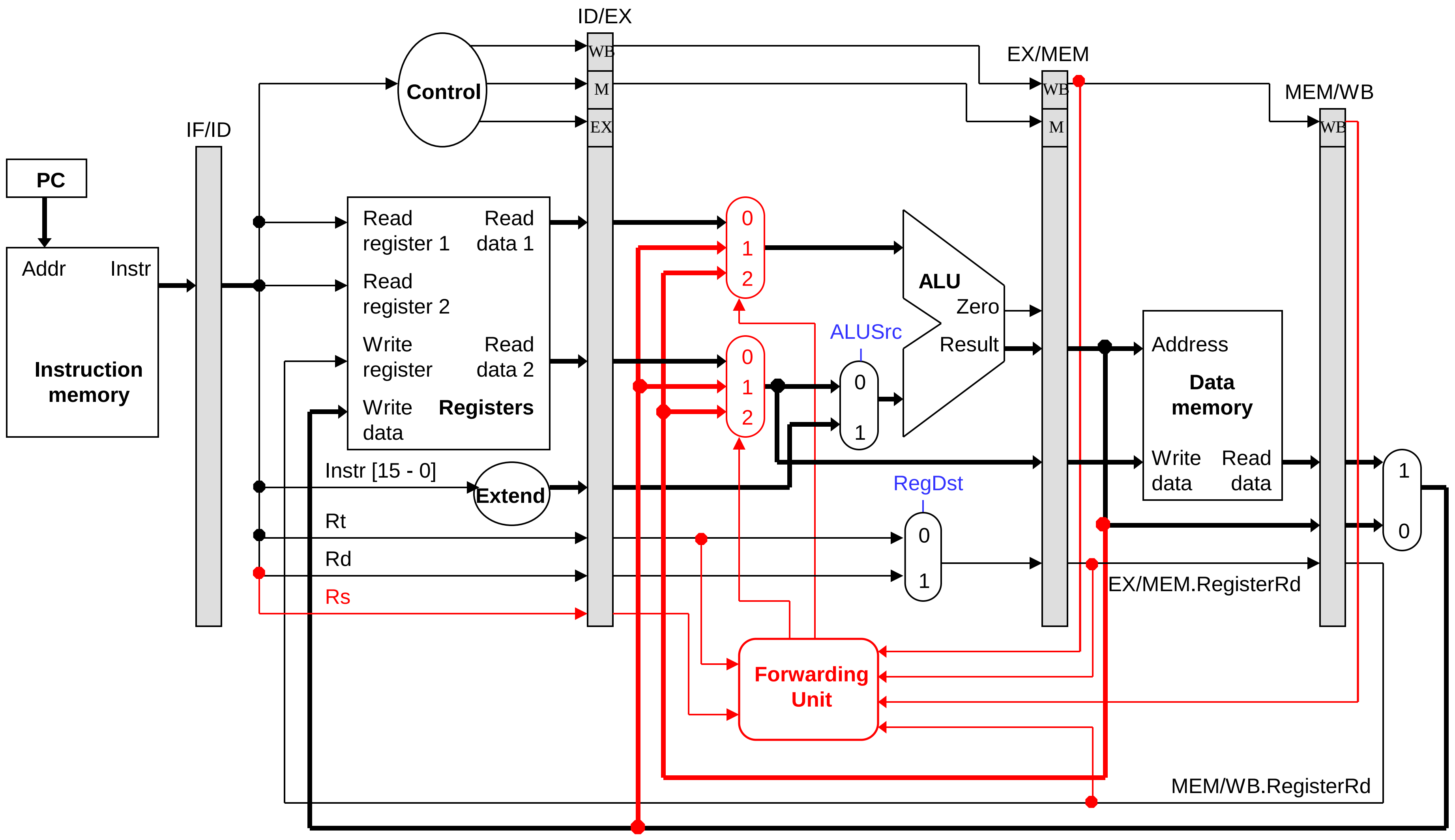
Clock cycle 5: forwarding $2 from MEM/WB



Lots of data hazards

- **The first data hazard occurs during cycle 4.**
  - The forwarding unit notices that the ALU’s first source register for the AND is also the destination of the SUB instruction.
  - The correct value is forwarded from the EX/MEM register, overriding the incorrect old value still in the register file.
- **A second hazard occurs during clock cycle 5.**
  - The ALU’s second source (for OR) is the SUB destination again.
  - This time, the value has to be forwarded from the MEM/WB pipeline register instead.
- **There are no other hazards involving the SUB instruction.**
  - During cycle 5, SUB writes its result back into register $2.
  - The ADD instruction can read this new value from the register file in the same cycle.

Complete pipelined datapath...so far

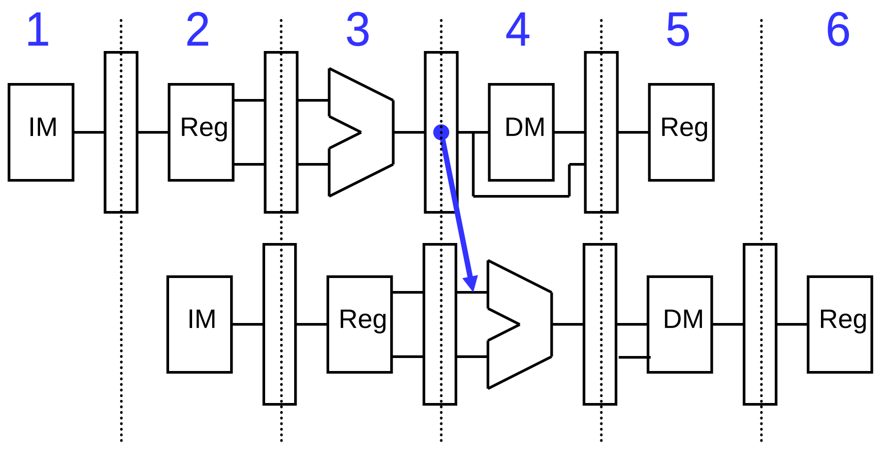


What about stores?

- Two “easy” cases:

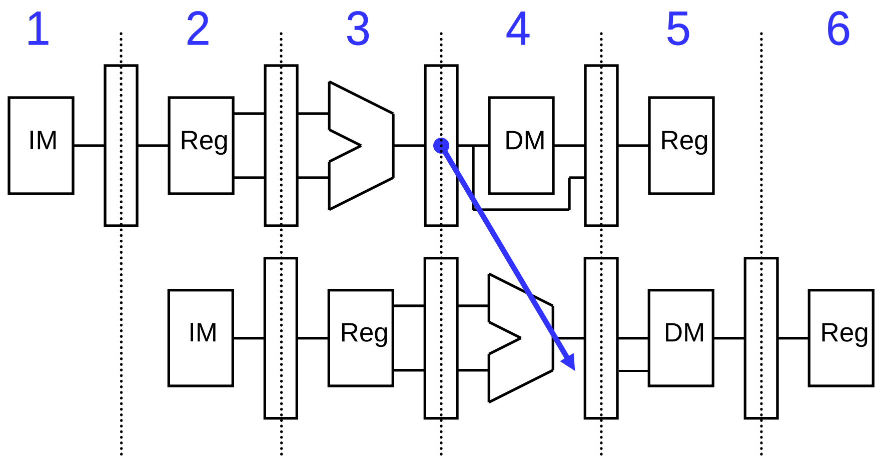
add $1, $2, $3

sw $4, 0($1)

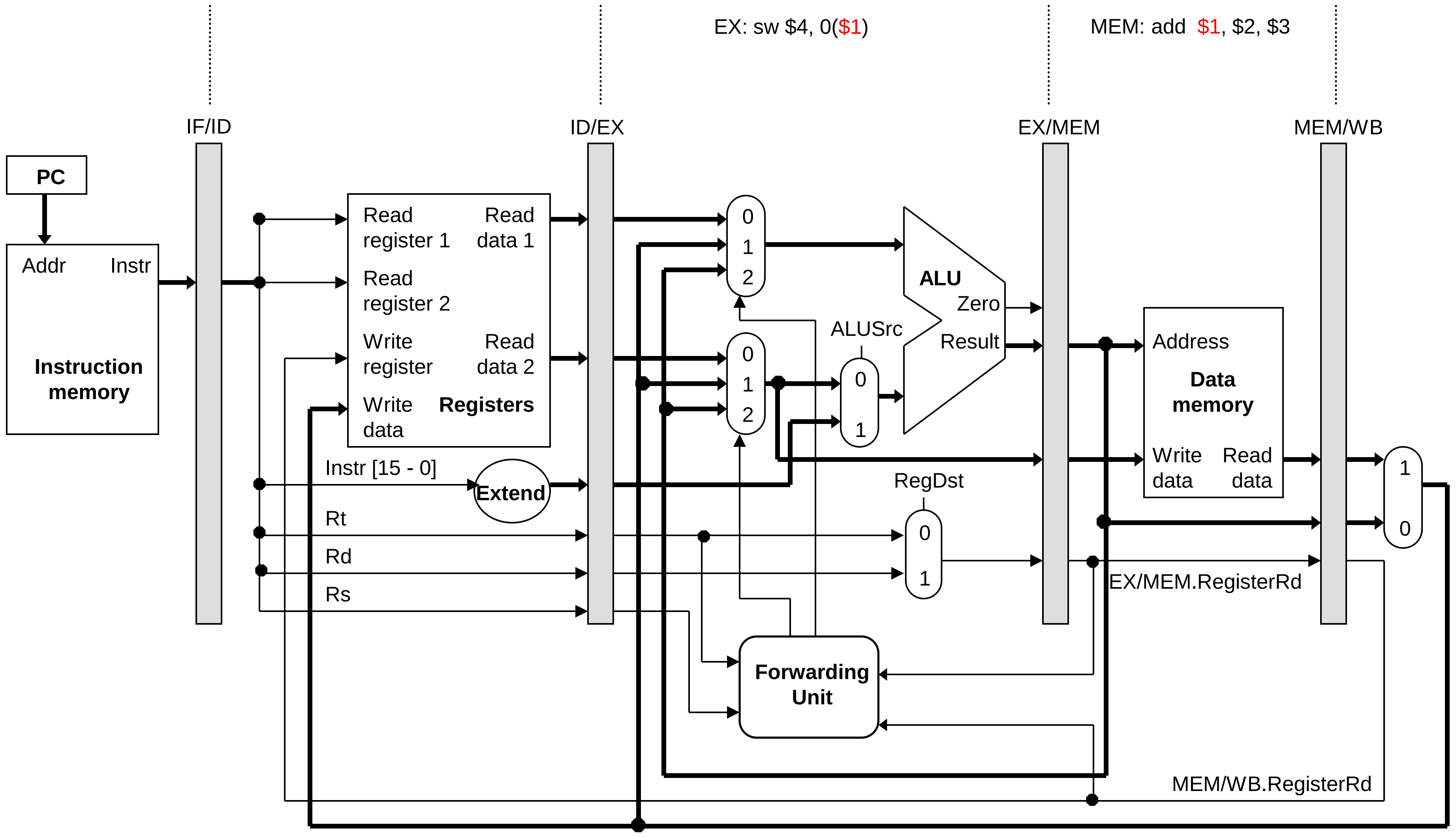


add $1, $2, $3

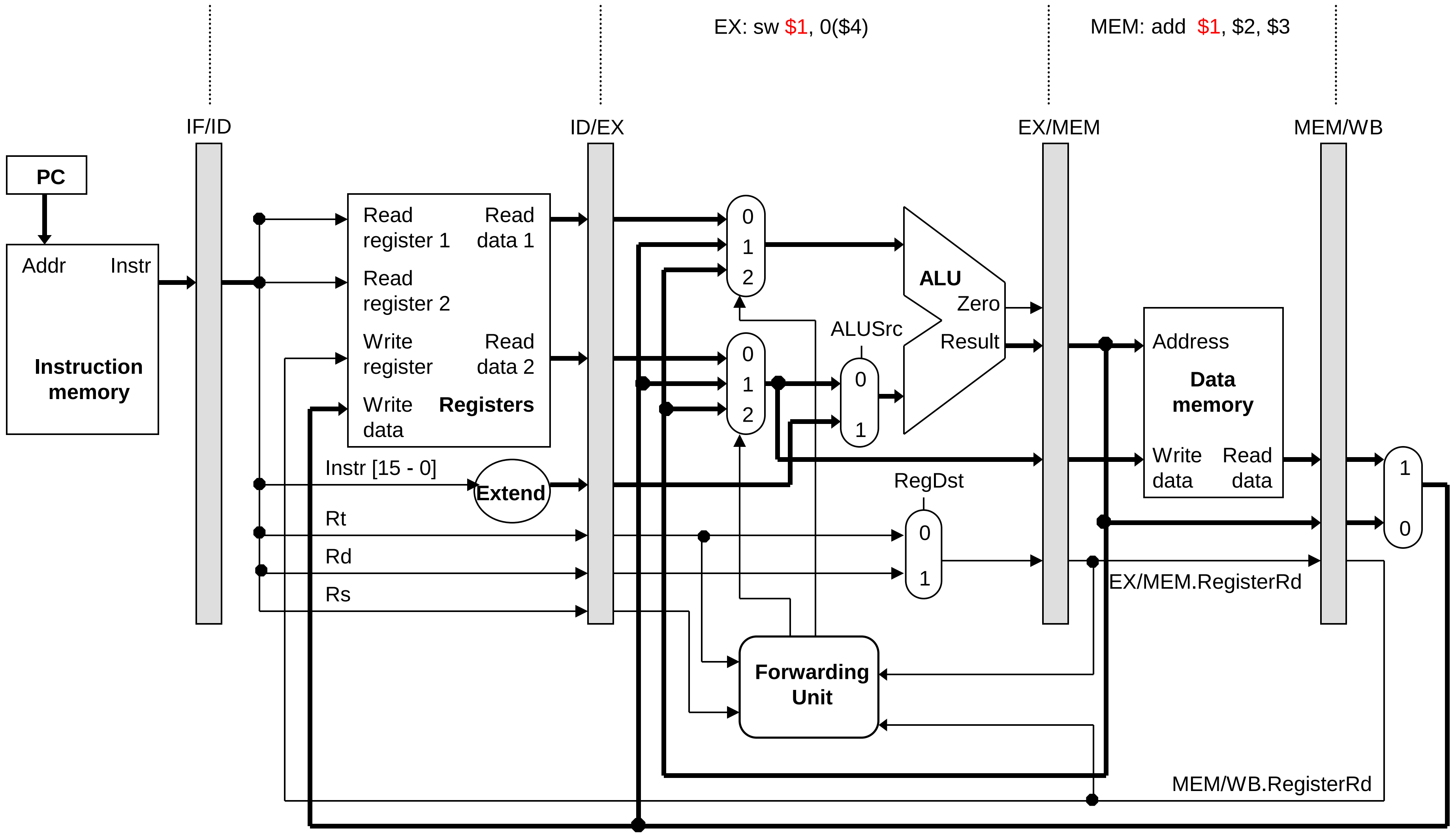
sw $1, 0($4)



Store Bypassing: Version 1

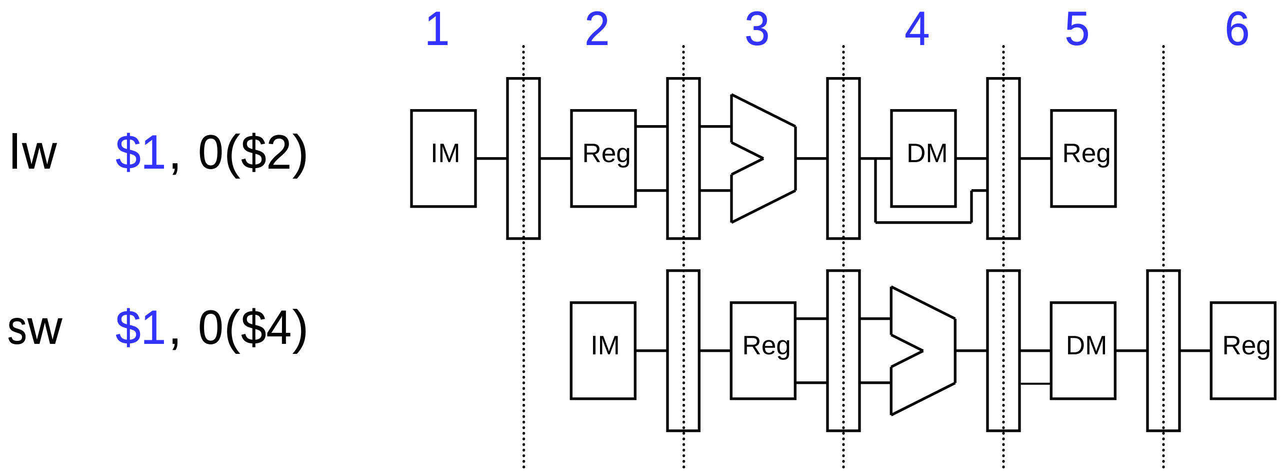


Store Bypassing: Version 2



What about stores?

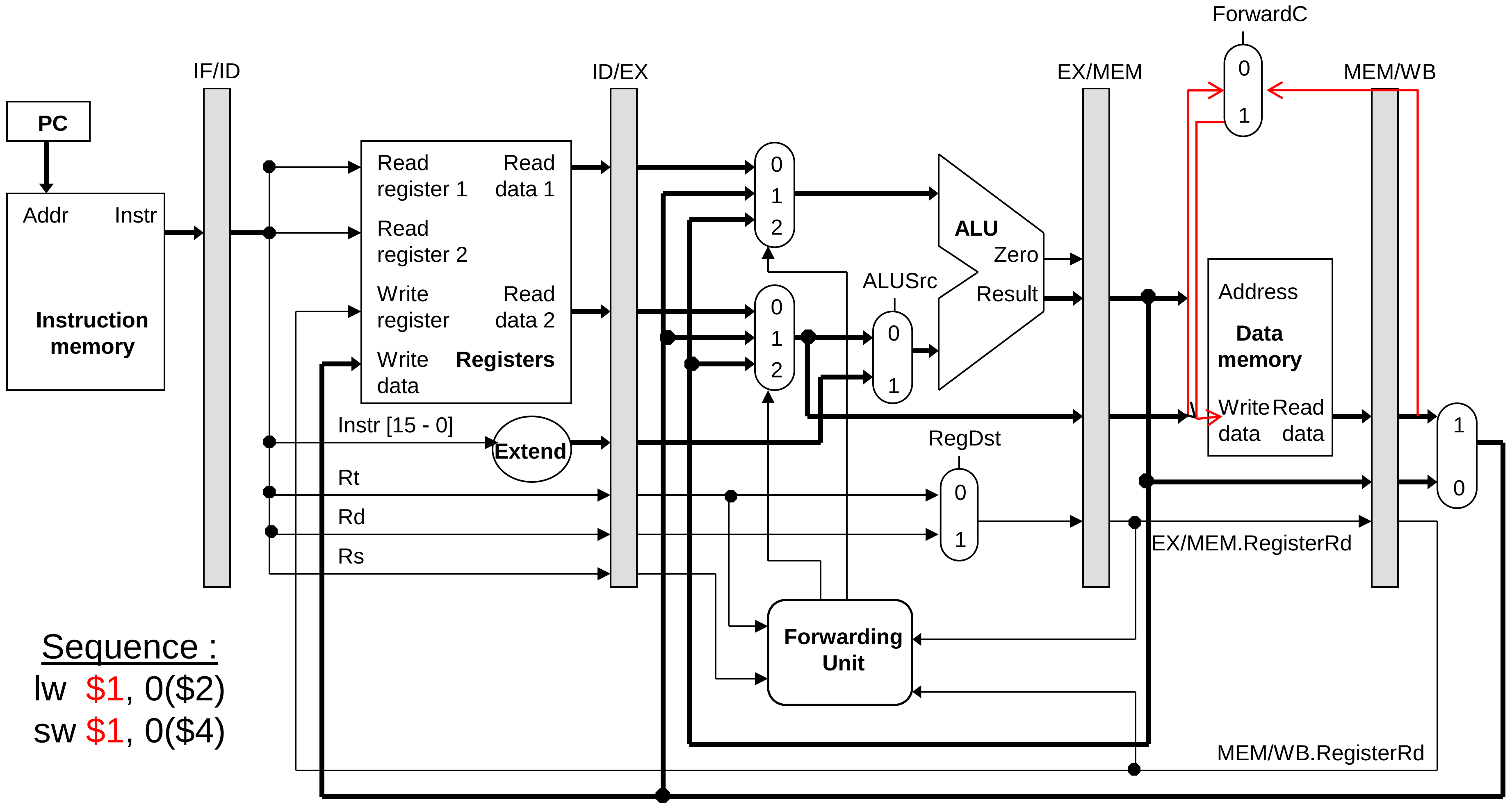
- **A harder case:**



- **In what cycle is:**
  - **The load value available?**
  - **The store value needed?**

- **What do we have to add to the datapath?**

Load/Store Bypassing: Extend the Datapath



Sequence :
lw $1, 0($2)
sw $1, 0($4)

Miscellaneous comments

- Each MIPS instruction writes to at most one register.
  - This makes the forwarding hardware easier to design, since there is only one destination register that ever needs to be forwarded.
- Forwarding is especially important with deep pipelines like the ones in all current PC processors.
- Section 4.8 of the textbook has some additional material not shown here.
  - Their hazard detection equations also ensure that the source register is not $0, which can never be modified.
  - There is a more complex example of forwarding, with several cases covered. Take a look at it!

Summary

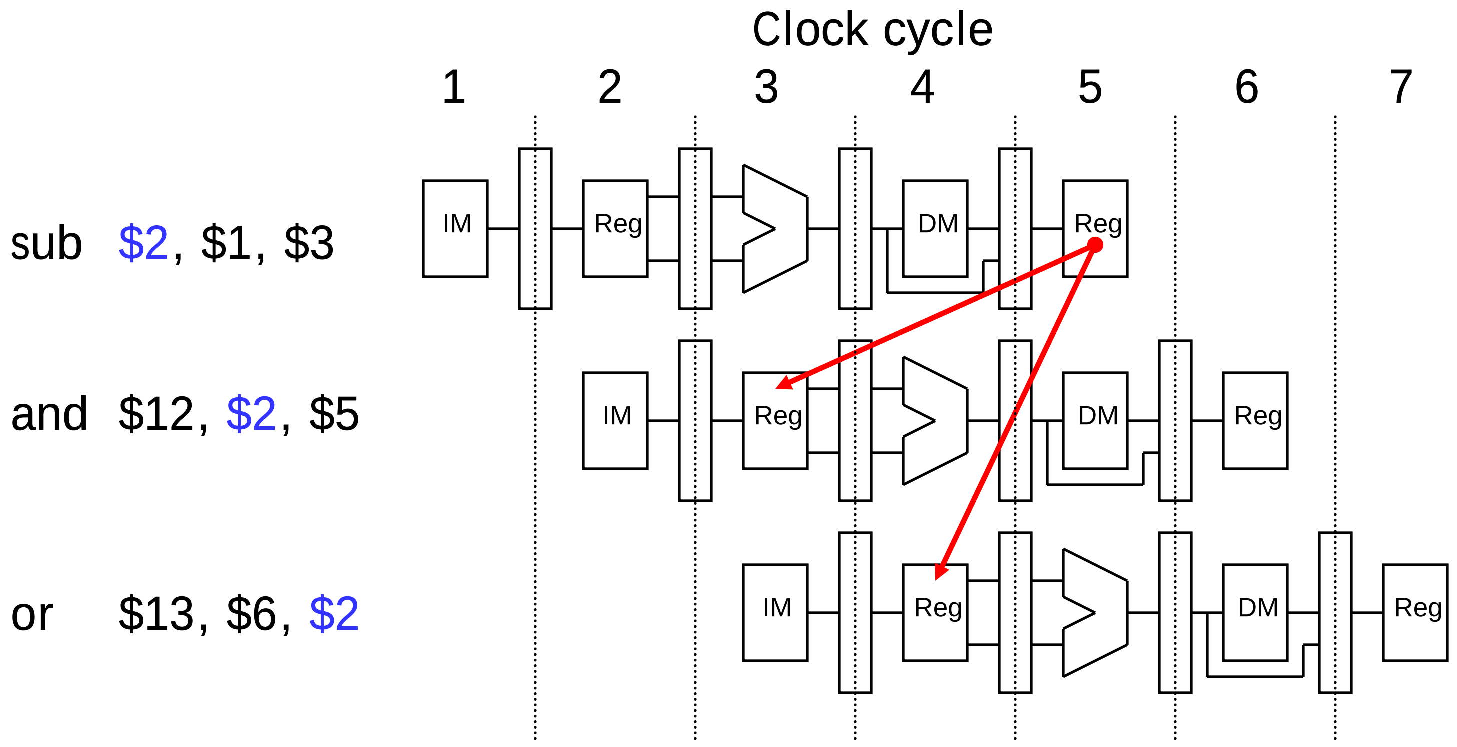
- In real code, most instructions are dependent upon other ones.
  - This can lead to data hazards in our original pipelined datapath.
  - Instructions can't write back to the register file soon enough for the next two instructions to read.
- Forwarding eliminates data hazards involving arithmetic instructions.
  - The forwarding unit detects hazards by comparing the destination registers of previous instructions to the source registers of the current instruction.
  - Hazards are avoided by grabbing results from the pipeline registers *before* they are written back to the register file.
- Next we'll finish up pipelining.
  - Forwarding can't save us in some cases involving lw.
  - We still haven't talked about branches for the pipelined datapath.

Stalls and flushes

- Last time, we discussed data hazards that can occur in pipelined CPUs if some instructions depend upon others that are still executing.
  - Many hazards can be resolved by forwarding data from the pipeline registers, instead of waiting for the writeback stage.
  - The pipeline continues running at full speed, with one instruction beginning on every clock cycle.
- Now we’ll see some real limitations of pipelining.
  - Forwarding may not work for data hazards from load instructions.
  - Branches affect the instruction fetch for the next clock cycle.
- In both of these cases we may need to slow down, or stall, the pipeline.

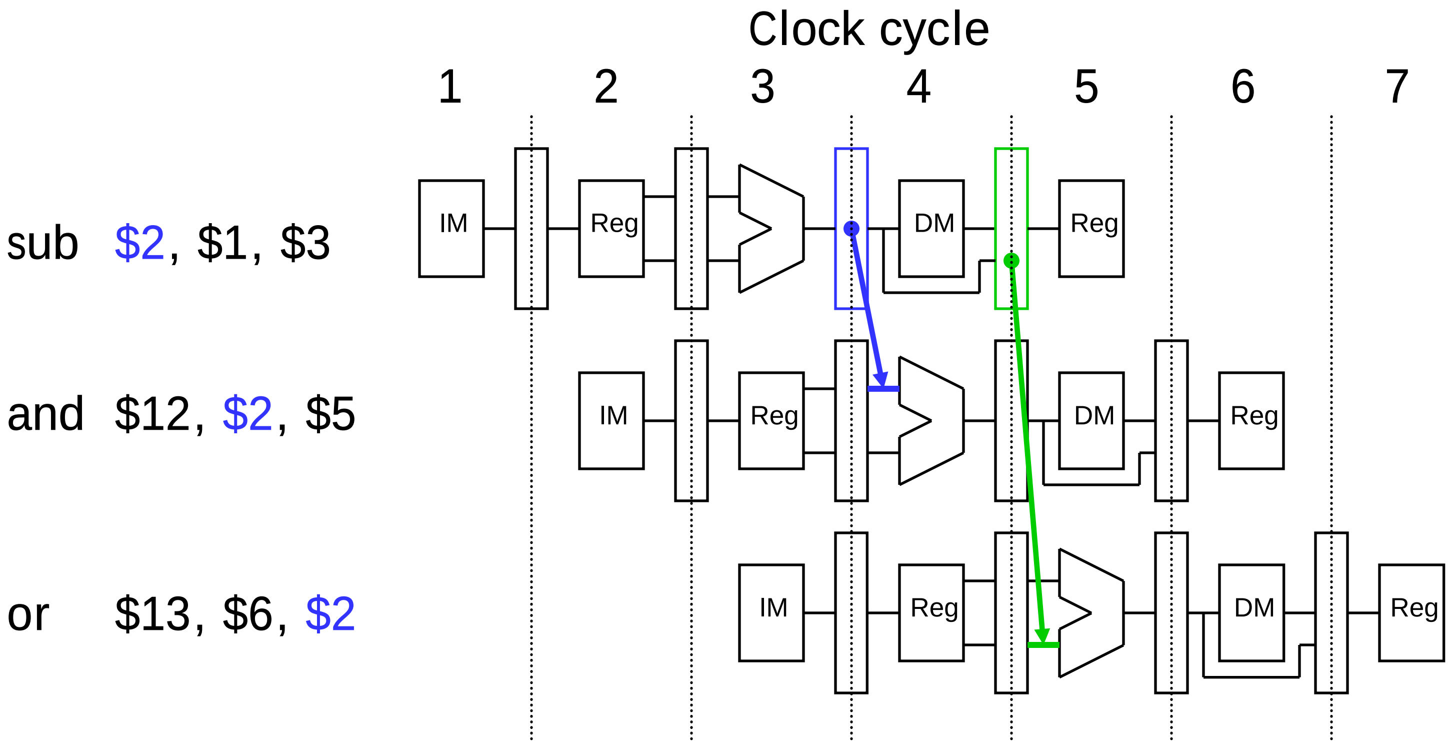
Data hazard review

- A data hazard arises if one instruction needs data that isn't ready yet.
  - Below, the AND and OR both need to read register $2.
  - But $2 isn't updated by SUB until the fifth clock cycle.
- Dependency arrows that point backwards indicate hazards.



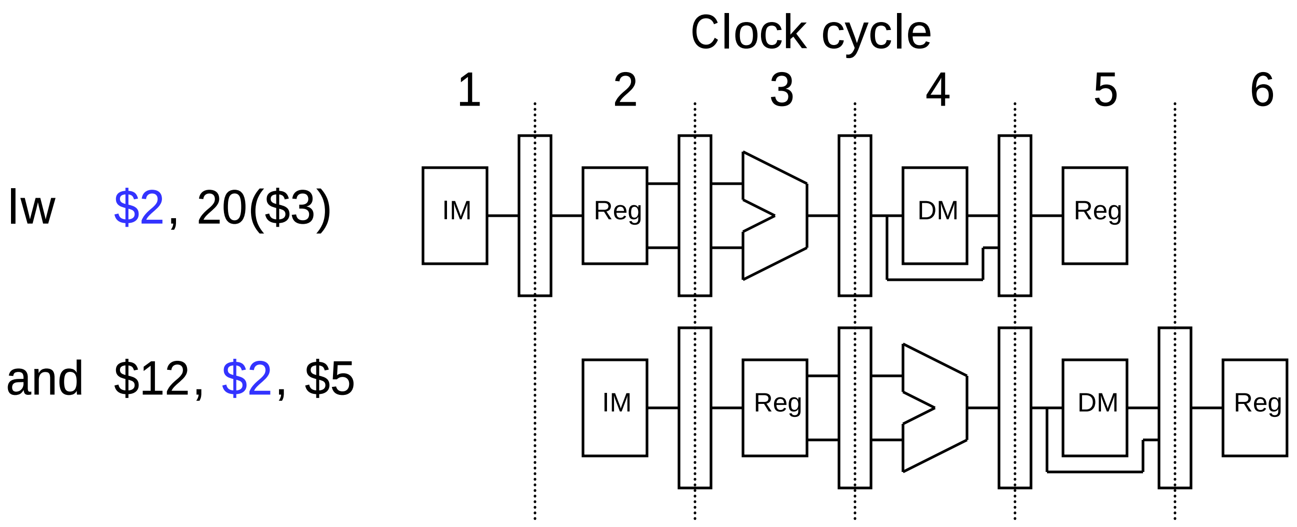
Forwarding to the rescue!

- The desired value ($1 - $3) has actually already been computed—it just hasn't been written to the registers yet.
- Forwarding allows other instructions to read ALU results directly from the pipeline registers, without going through the register file.



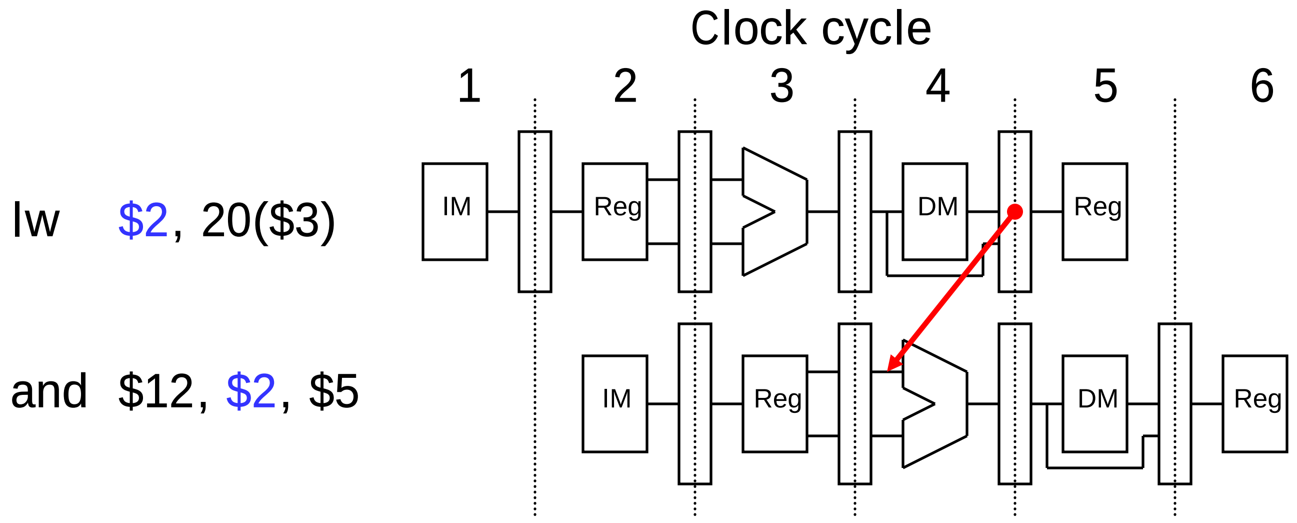
What about loads?

- Imagine if the first instruction in the example was LW instead of SUB.
  - How does this change the data hazard?



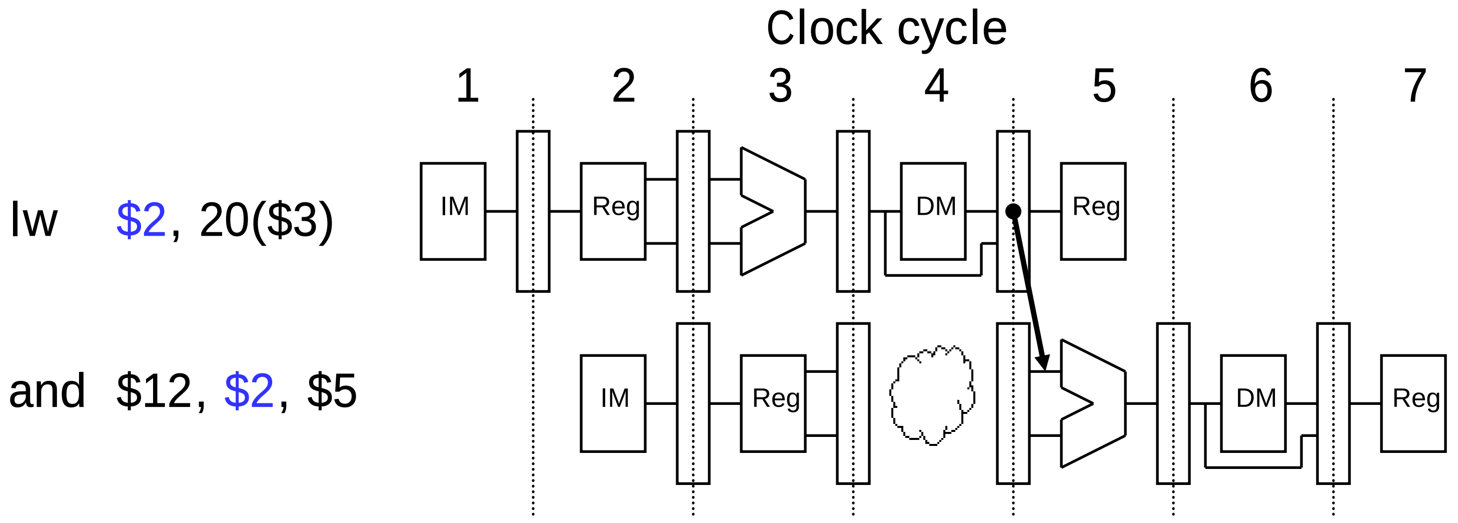
What about loads?

- Imagine if the first instruction in the example was LW instead of SUB.
  - The load data doesn't come from memory until the *end* of cycle 4.
  - But the AND needs that value at the *beginning* of the same cycle!
- This is a "true" data hazard—the data is not available when we need it.



Stalling

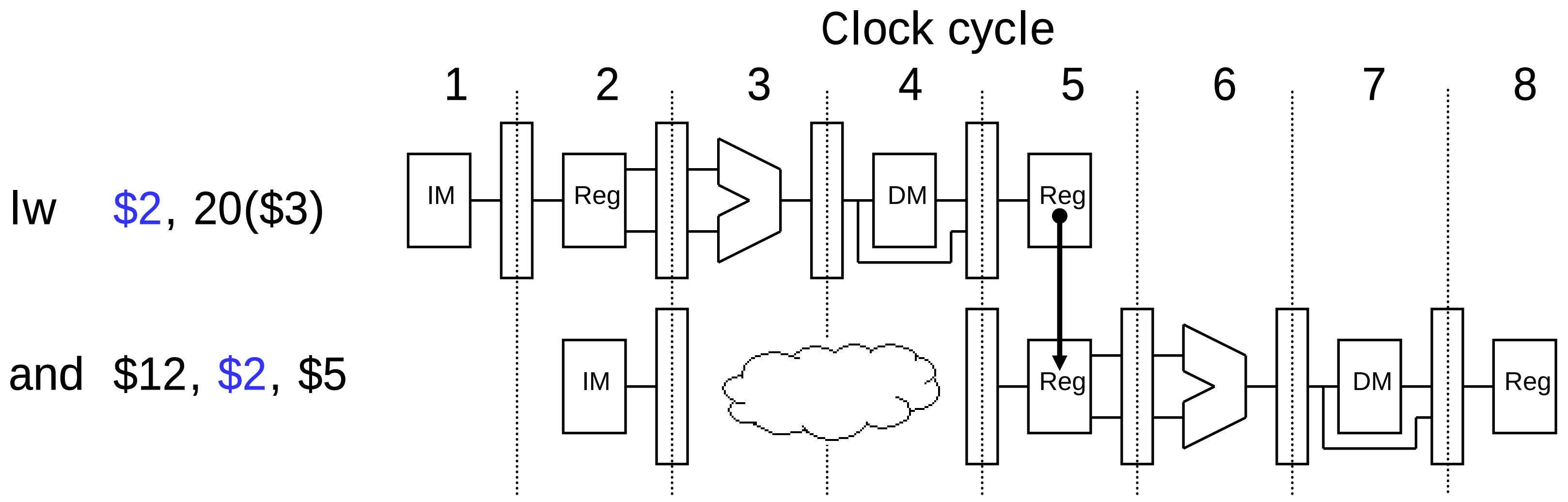
- The easiest solution is to stall the pipeline.
- We could delay the AND instruction by introducing a one-cycle delay into the pipeline, sometimes called a bubble.



- Notice that we're still using forwarding in cycle 5, to get data from the MEM/WB pipeline register to the ALU.

Stalling and forwarding

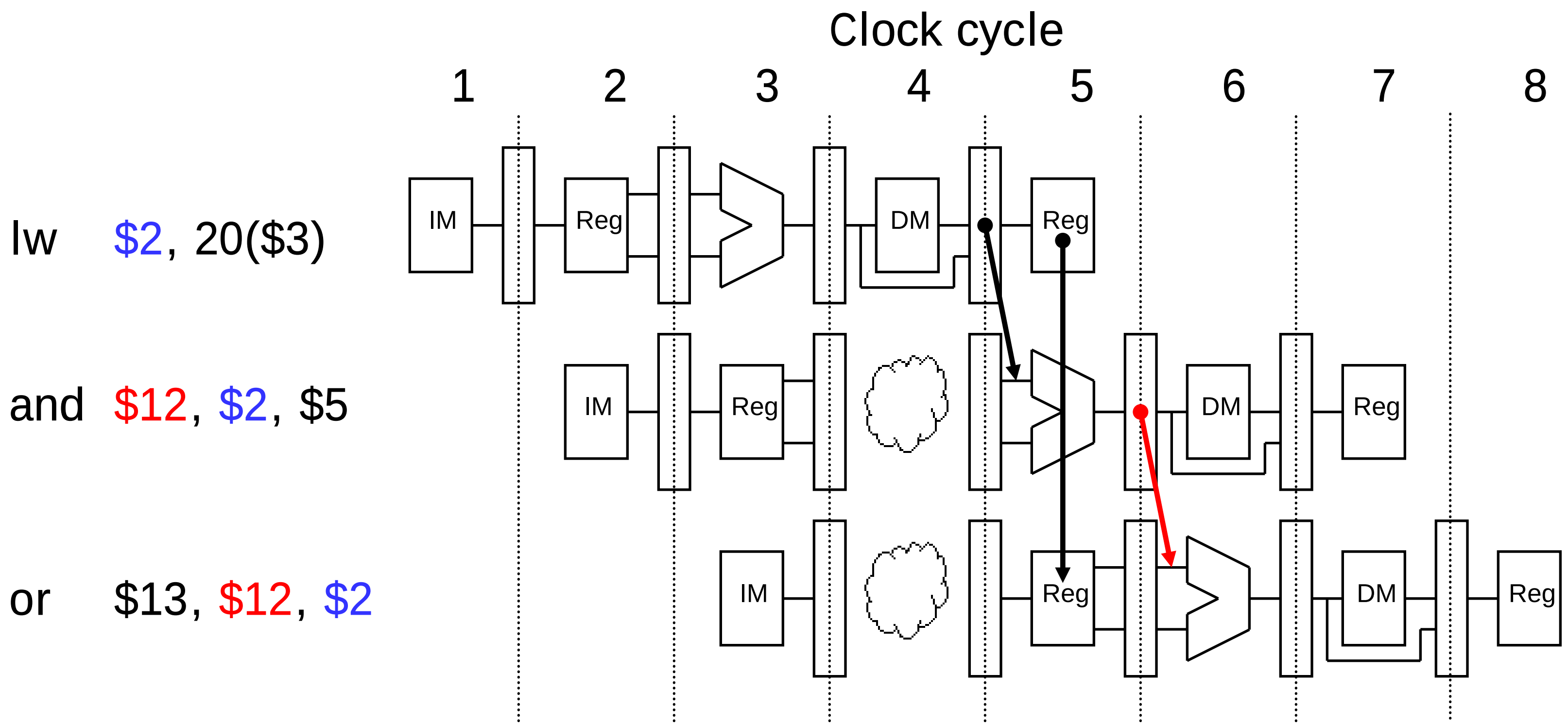
- Without forwarding, we'd have to stall for *two* cycles to wait for the LW instruction's writeback stage.



- In general, you can always stall to avoid hazards—but dependencies are very common in real code, and stalling often can reduce performance by a significant amount.

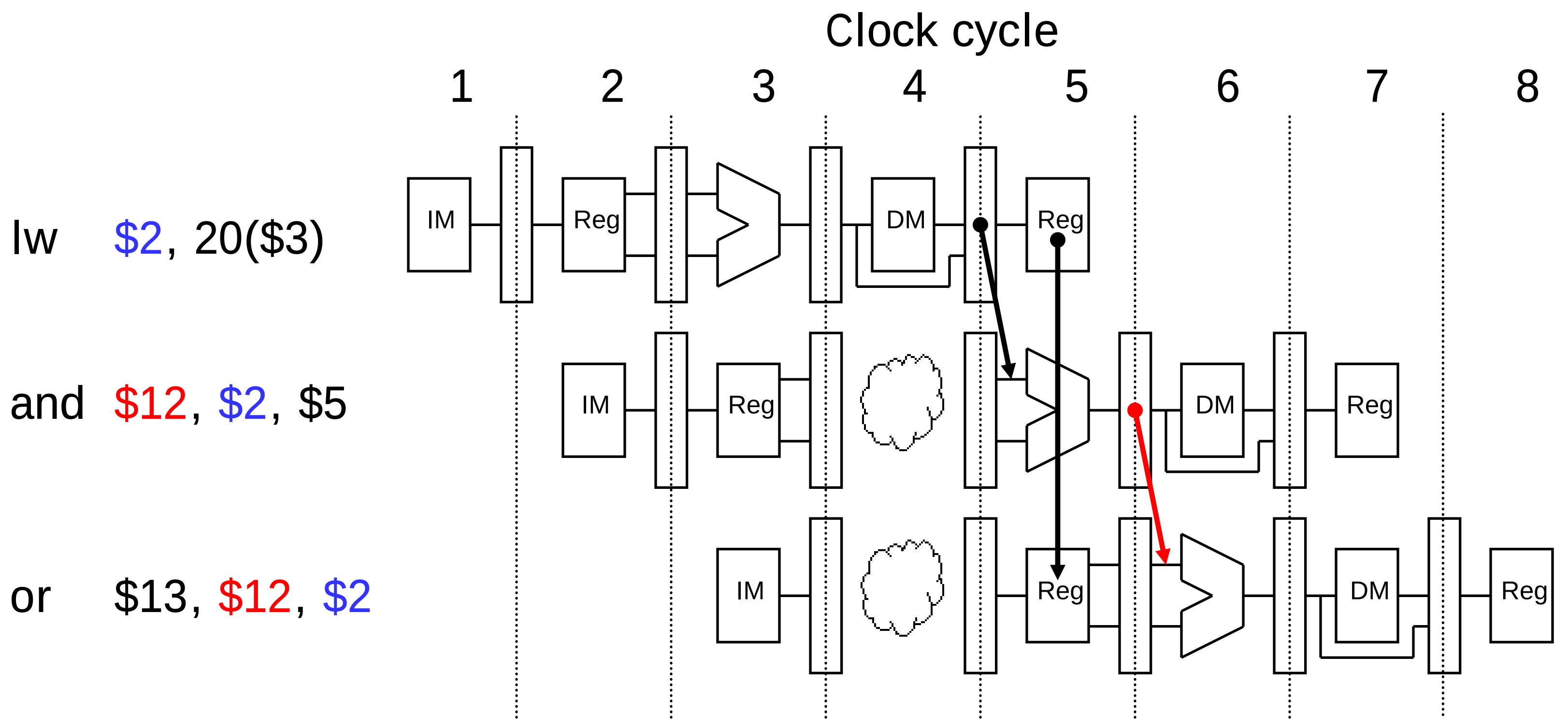
Stalling delays the entire pipeline

- If we delay the second instruction, we'll have to delay the third one too.
  - Why? (two reasons)



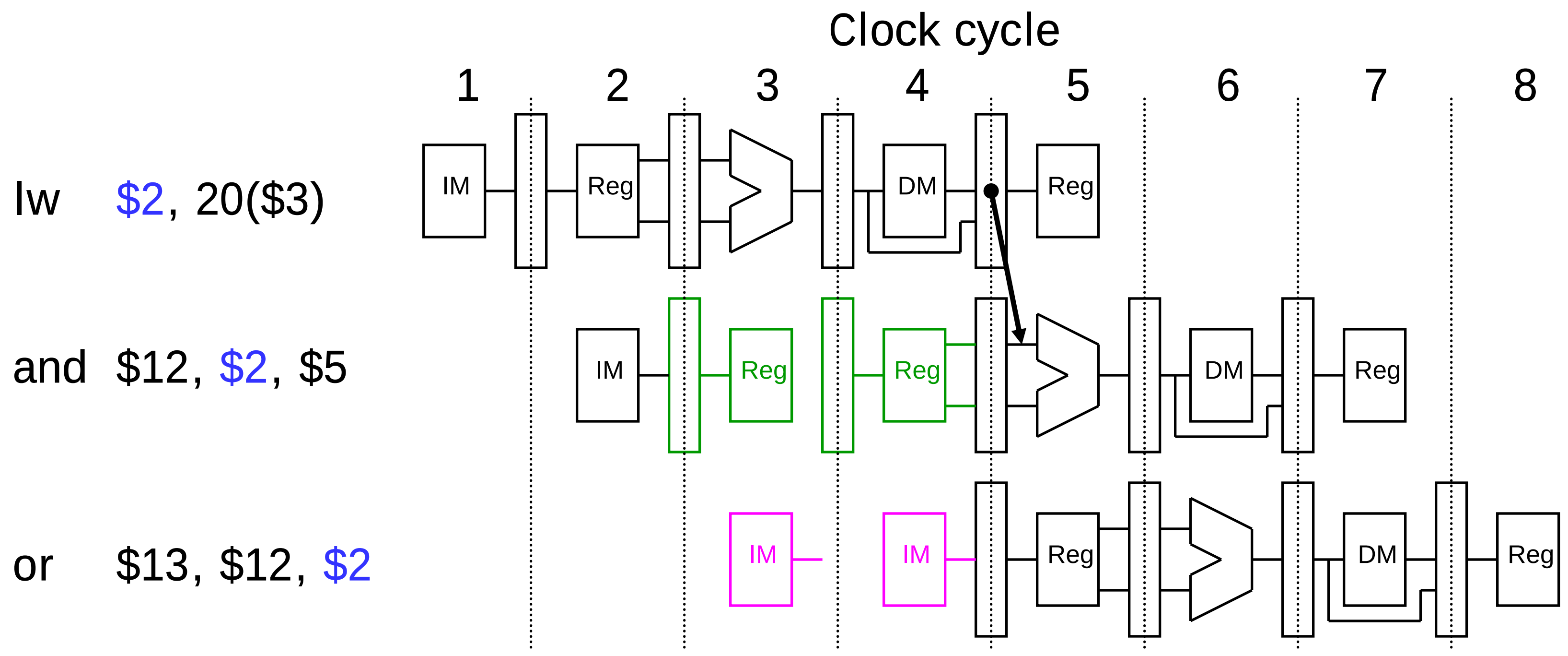
Stalling delays the entire pipeline

- If we delay the second instruction, we'll have to delay the third one too.
  - This is necessary to make forwarding work between AND and OR.
  - It also prevents problems such as two instructions trying to write to the same register in the same cycle.



Implementing stalls

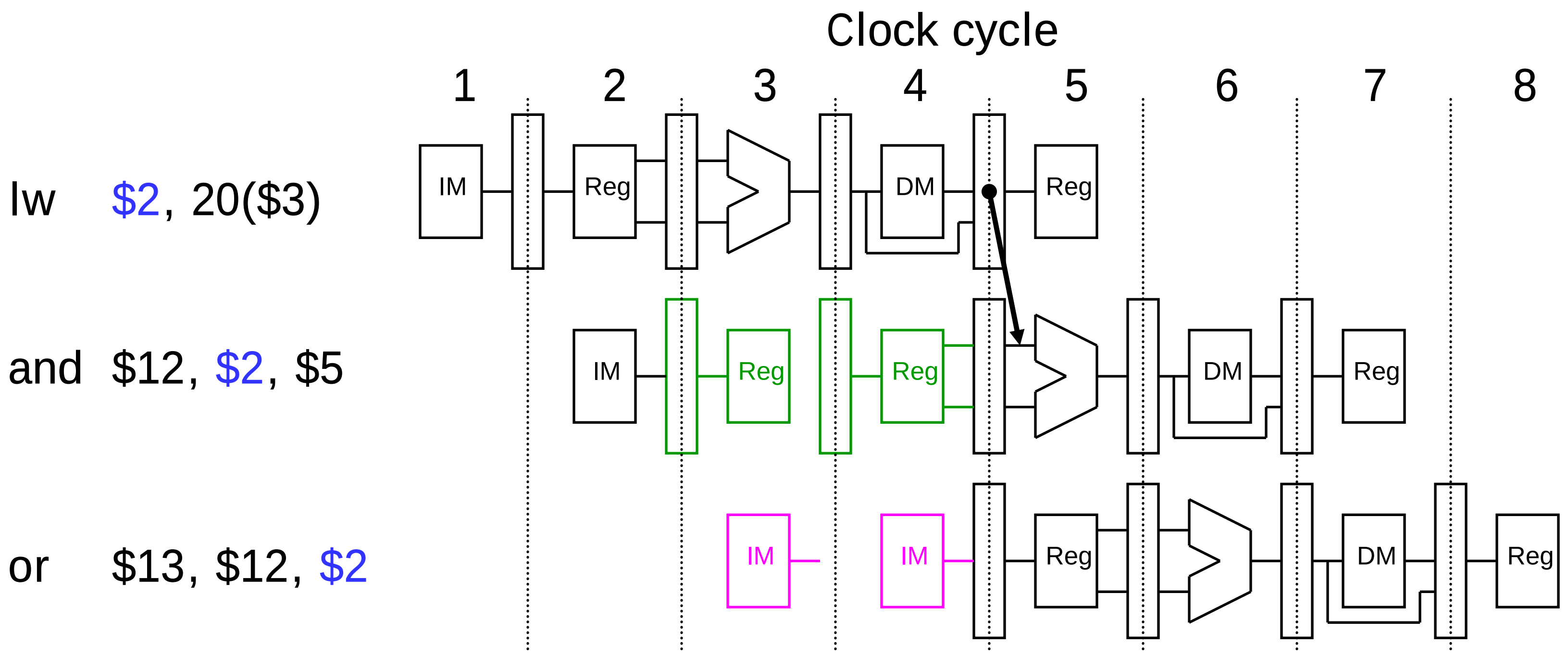
- One way to implement a stall is to force the two instructions after LW to pause and remain in their ID and IF stages for one extra cycle.



- This is easily accomplished.
  – Don’t update the PC, so the current IF stage is repeated.
  – Don’t update the IF/ID register, so the ID stage is also repeated.

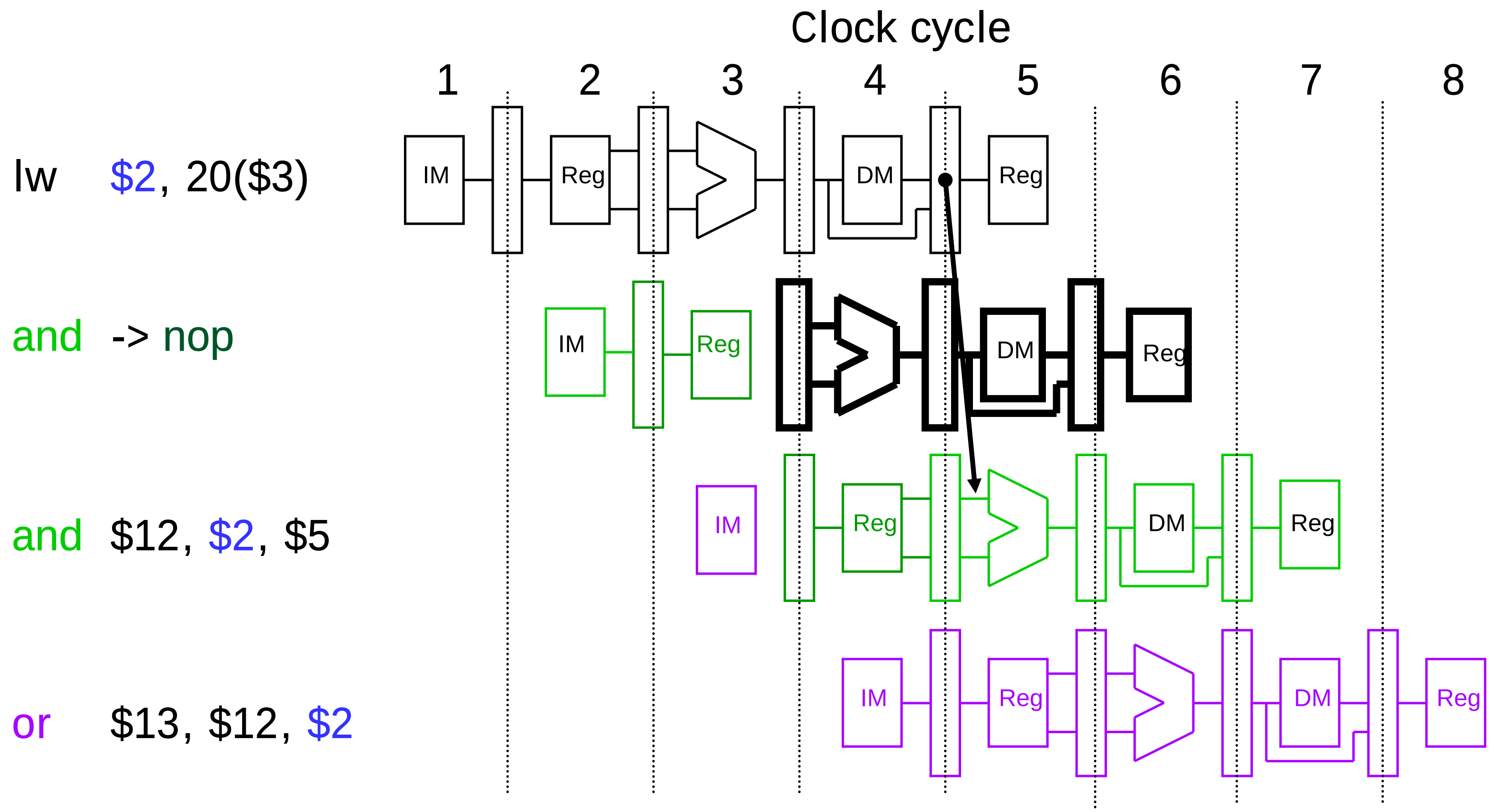
What about EXE, MEM, WB

- But what about the ALU during cycle 4, the data memory in cycle 5, and the register file write in cycle 6?



- Those units aren't used in those cycles because of the stall, so we can set the EX, MEM and WB control signals to all 0s.

Stall = Nop conversion



- The effect of a load stall is to insert an empty or nop (“no operation”) instruction into the pipeline

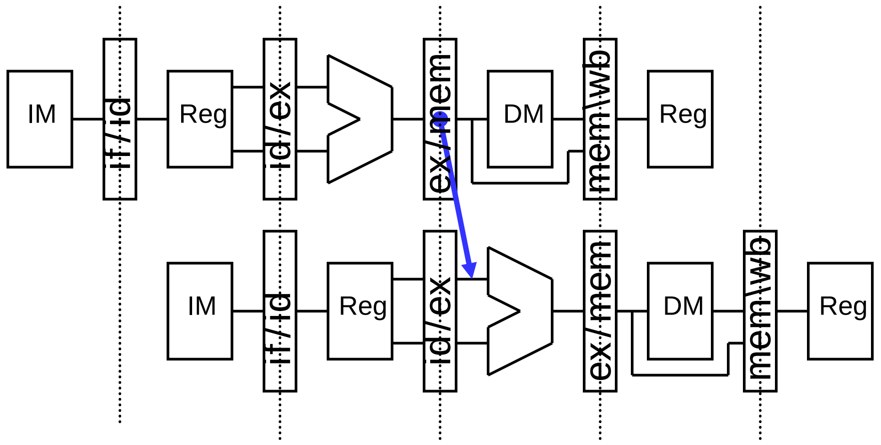
Detecting stalls

- Detecting stall is much like detecting data hazards.
- Recall the format of hazard detection equations:

```
if (EX/MEM.RegWrite = 1
    and EX/MEM.RegisterRd = ID/EX.RegisterRs)
then Bypass Rs from EX/MEM stage latch
```

sub $2, $1, $3

and $12, $2, $5

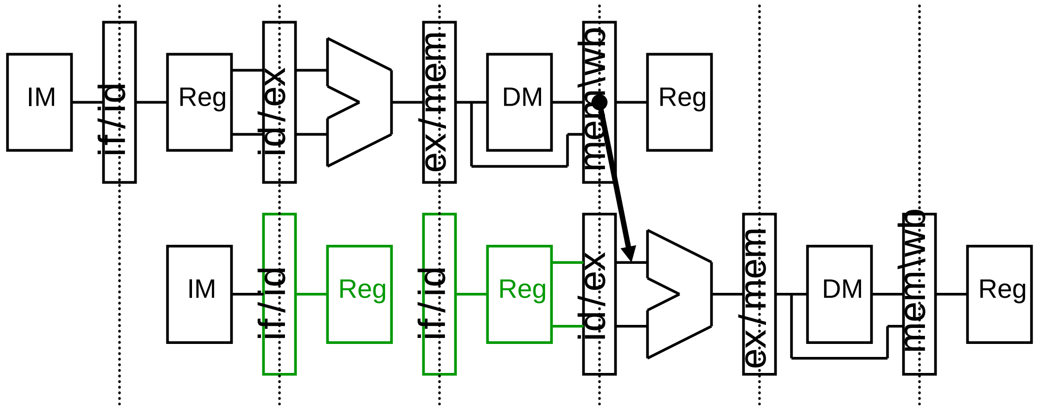


Detecting Stalls, cont.

- When should stalls be detected?

lw $2, 20($3)

and $12, $2, $5



- What is the stall condition?

if (

)

then stall

Detecting stalls

- We can detect a load hazard between the current instruction in its ID stage and the previous instruction in the EX stage just like we detected data hazards.
- A hazard occurs if the previous instruction was LW...

  ID/EX.MemRead = 1

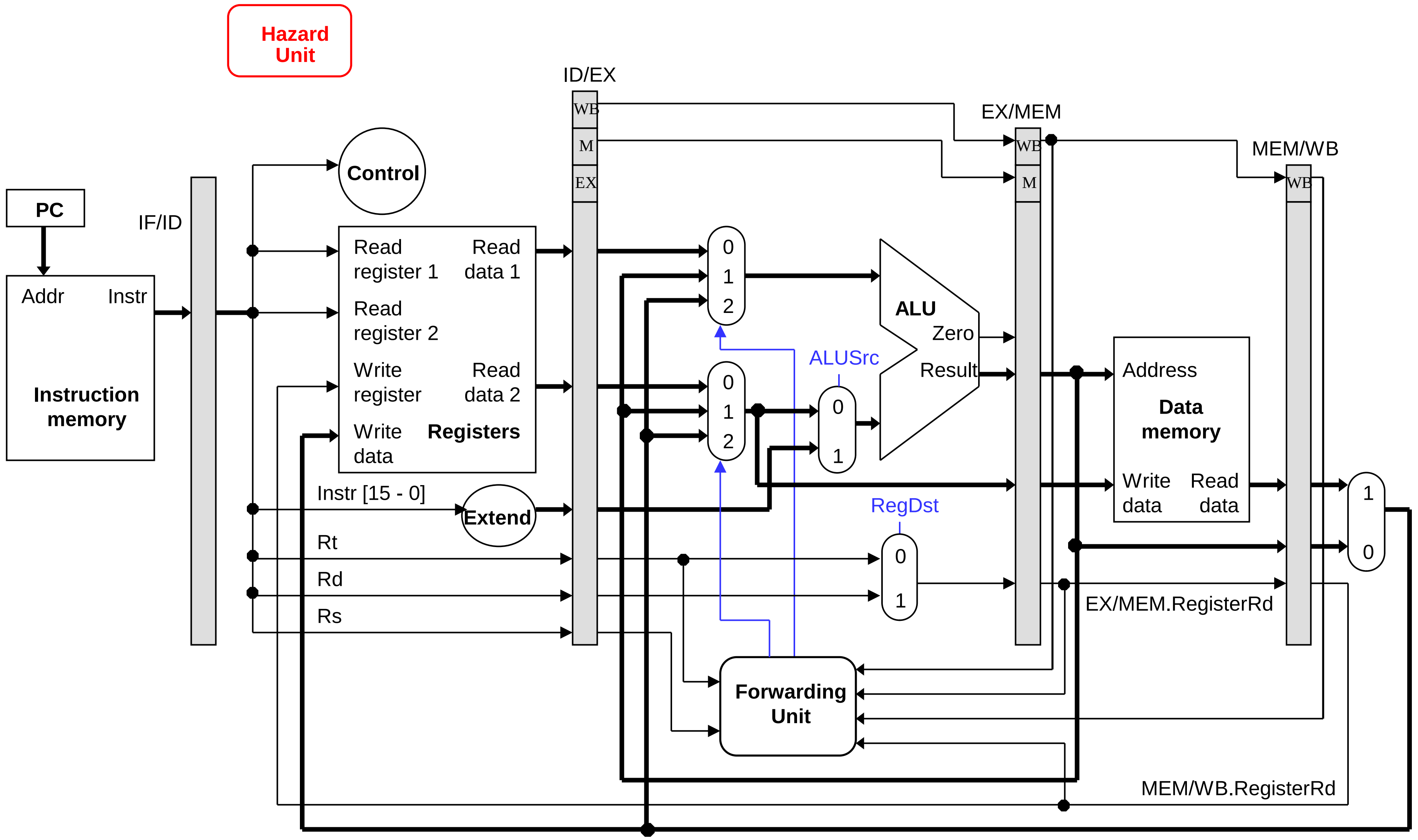
  ...and the LW destination is one of the current source registers.

  ID/EX.RegisterRt = IF/ID.RegisterRs
  or
  ID/EX.RegisterRt = IF/ID.RegisterRt

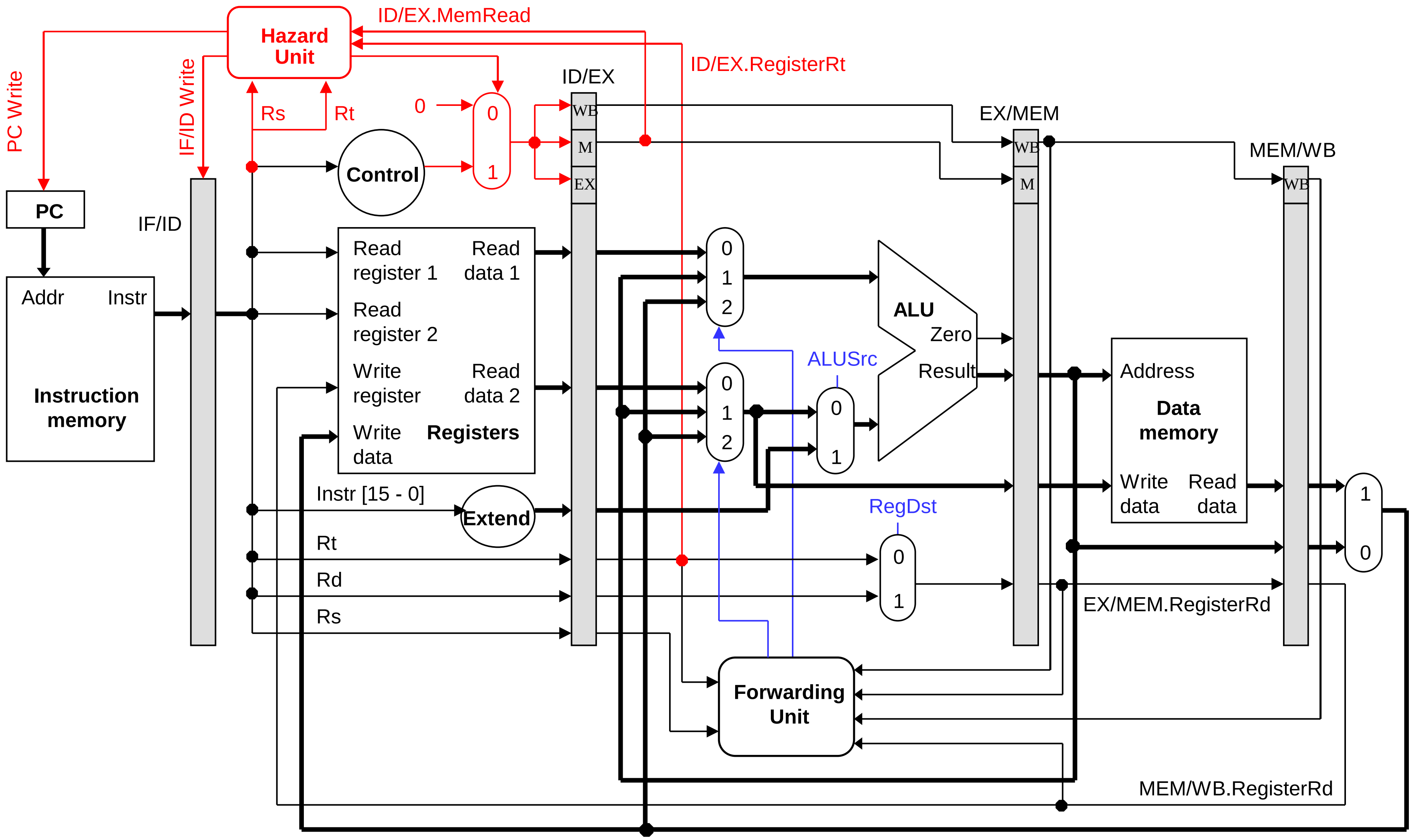
- The complete test for stalling is the conjunction of these two conditions.

```
if (ID/EX.MemRead = 1 and
     ( ID/EX.RegisterRt = IF/ID.RegisterRs or
       ID/EX.RegisterRt = IF/ID.RegisterRt))
then stall
```

Adding hazard detection to the CPU



Adding hazard detection to the CPU

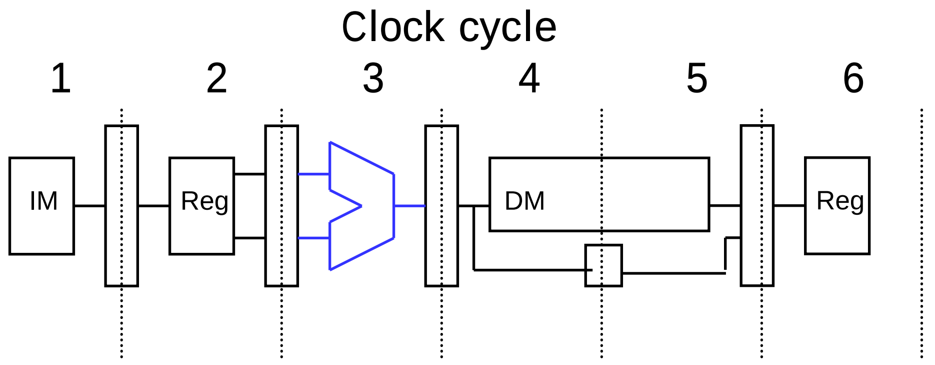


The hazard detection unit

- The hazard detection unit's inputs are as follows.
  - IF/ID.RegisterRs and IF/ID.RegisterRt, the source registers for the current instruction.
  - ID/EX.MemRead and ID/EX.RegisterRt, to determine if the previous instruction is LW and, if so, which register it will write to.
- By inspecting these values, the detection unit generates three outputs.
  - Two new control signals PCWrite and IF/ID Write, which determine whether the pipeline stalls or continues.
  - A mux select for a new multiplexer, which forces control signals for the current EX and future MEM/WB stages to 0 in case of a stall.

Generalizing Forwarding/Stalling

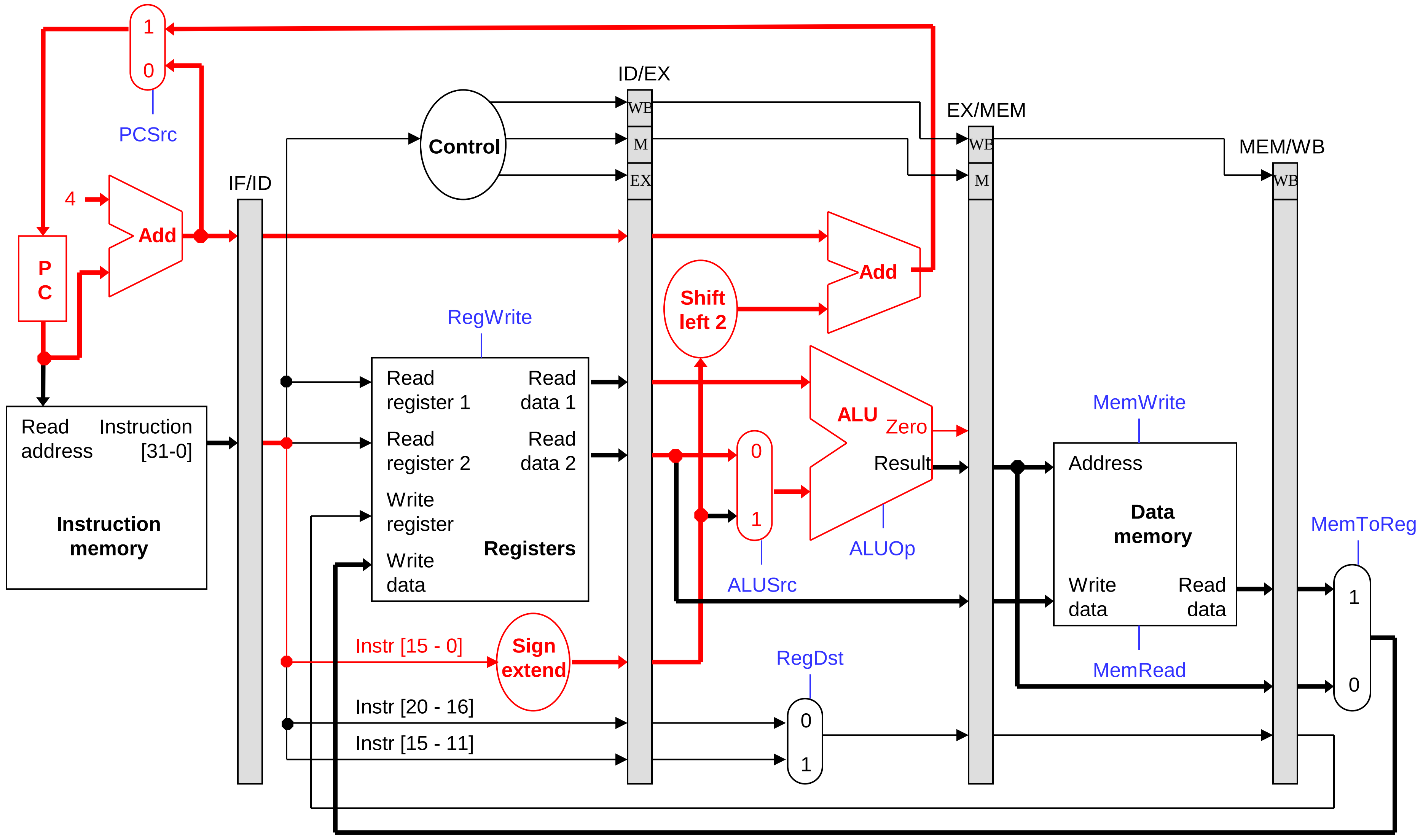
- What if data memory access was so slow, we wanted to pipeline it over 2 cycles?



- How many bypass inputs would the muxes in EXE have?
- Which instructions in the following require stalling and/or bypassing?

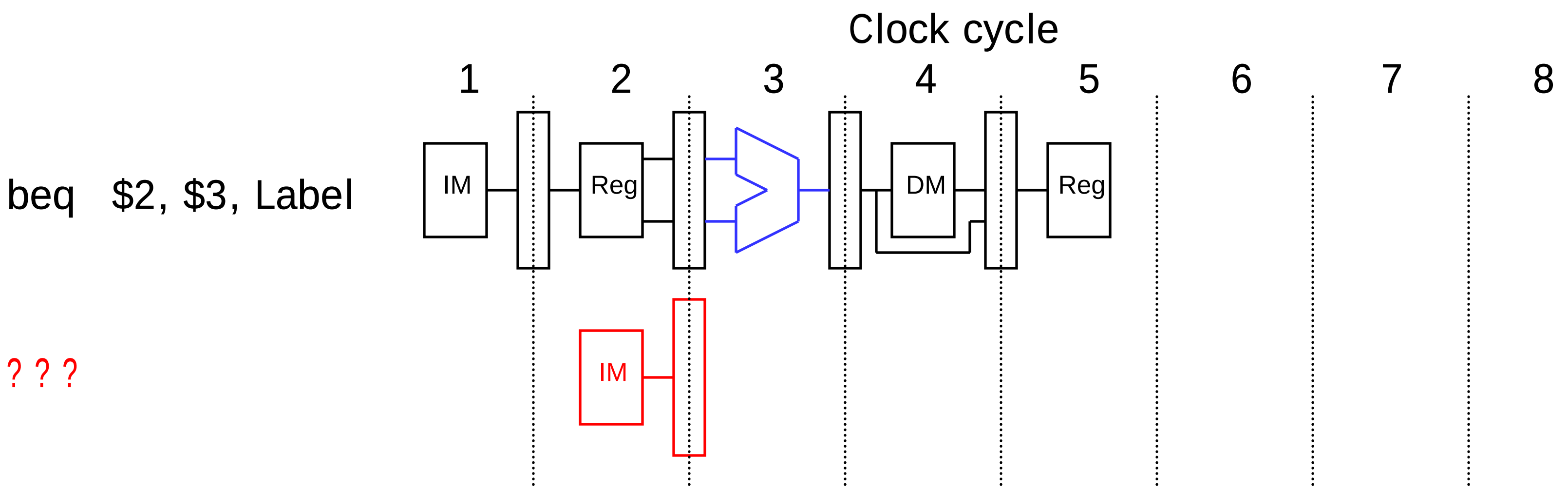
```
lw      r13, 0(r11)
add     r7, r8, r13
add     r15, r7, r13
```


Branches in the original pipelined datapath



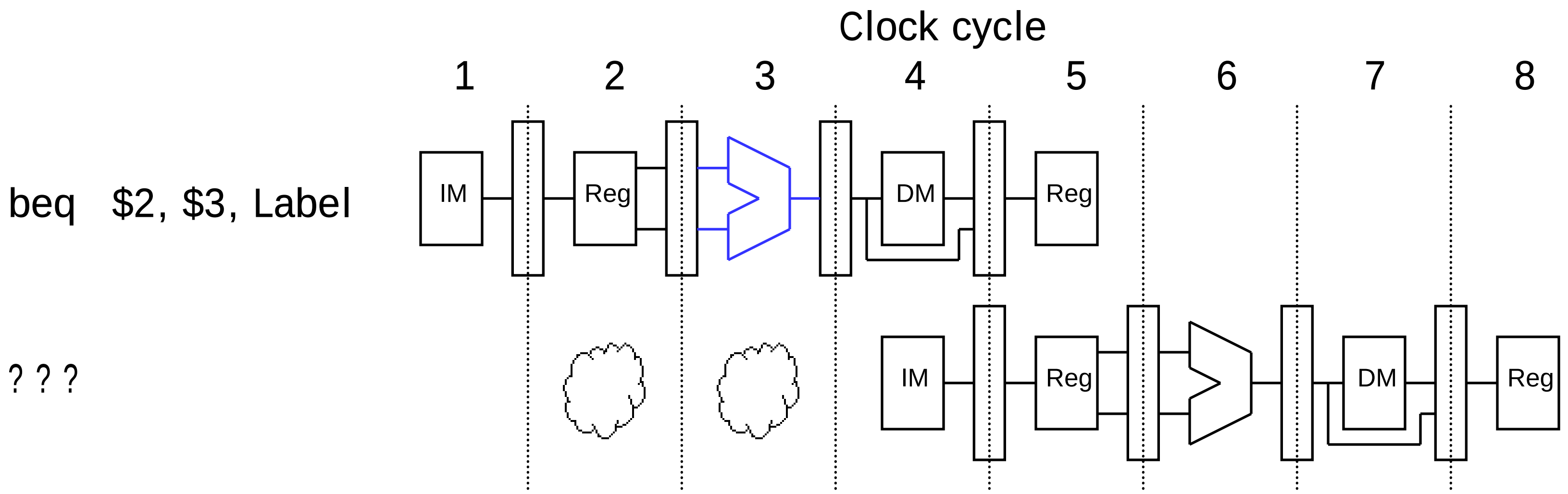
Branches

- Most of the work for a branch computation is done in the EX stage.
  - The branch target address is computed.
  - The source registers are compared by the ALU, and the Zero flag is set or cleared accordingly.
- Thus, the branch decision cannot be made until the end of the EX stage.
  - But we need to know which instruction to fetch next, in order to keep the pipeline running!
  - This leads to what's called a control hazard.



Stalling is one solution

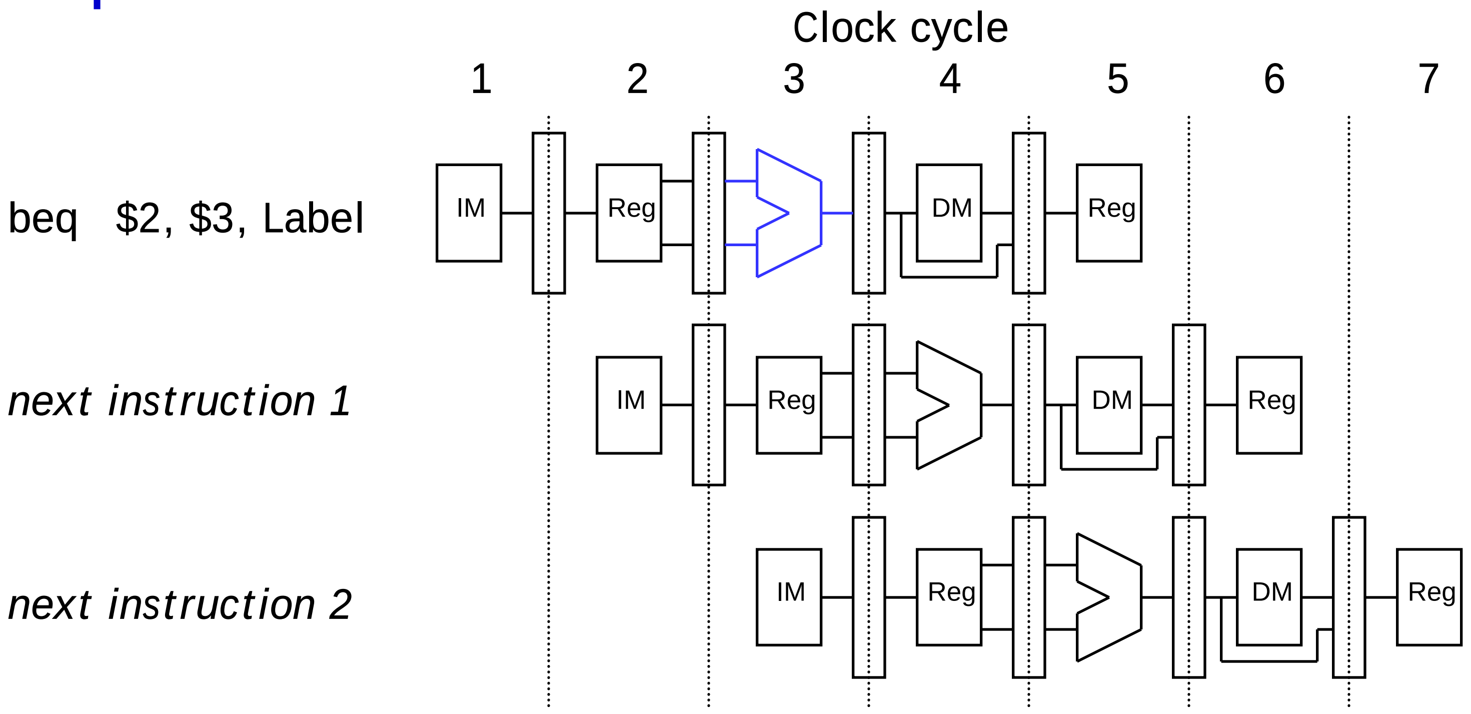
- Again, stalling is always one possible solution.



- Here we just stall until cycle 4, after we do make the branch decision.

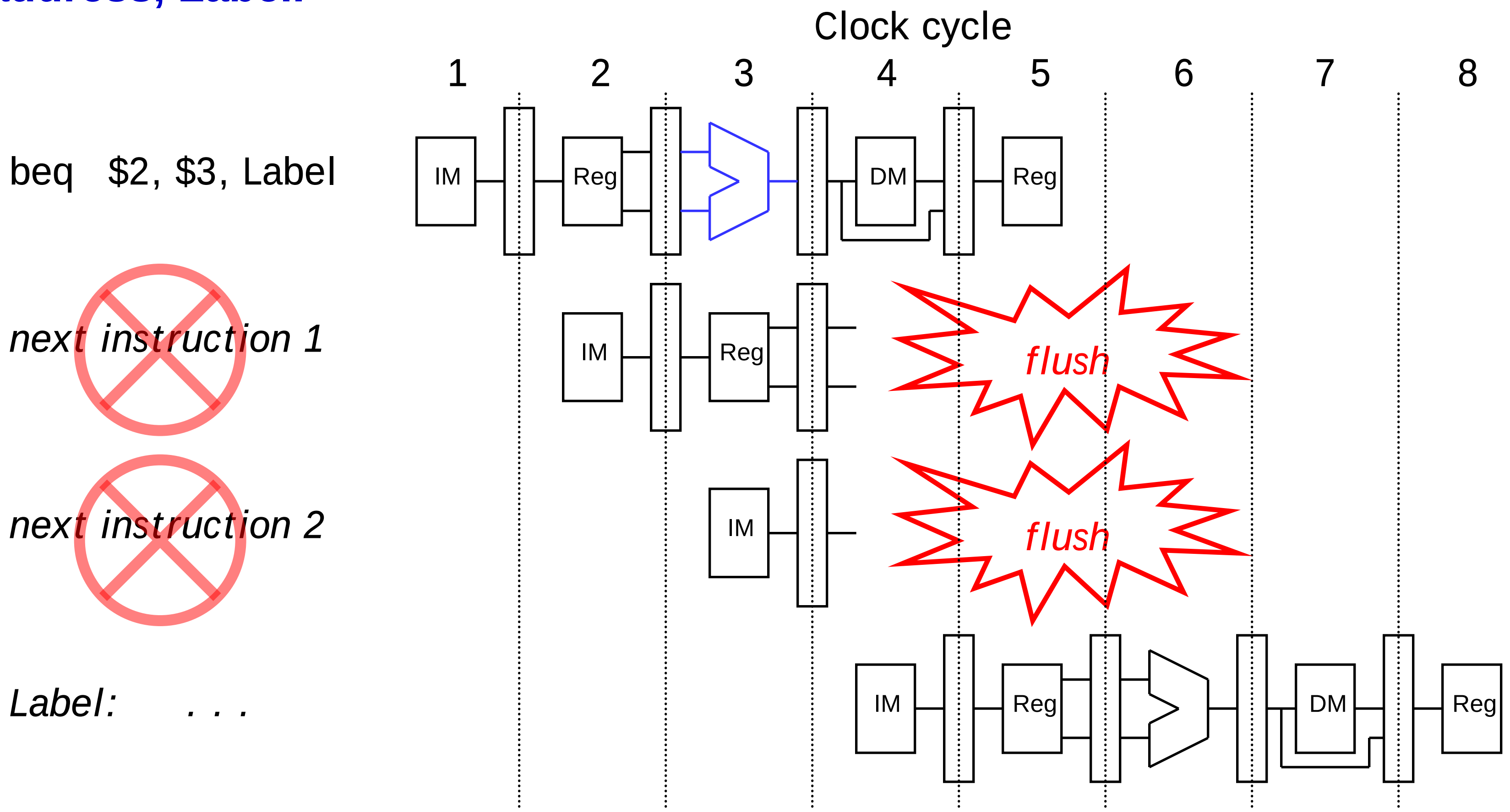
Branch prediction

- Another approach is to *guess* whether or not the branch is taken.
  - In terms of hardware, it's easier to assume the branch is *not* taken.
  - This way we just increment the PC and continue execution, as for normal instructions.
- If we're correct, then there is no problem and the pipeline keeps going at full speed.



Branch misprediction

- If our guess is wrong, then we would have already started executing two instructions incorrectly. We'll have to discard, or **flush**, those instructions and begin executing the right ones from the branch target address, Label.

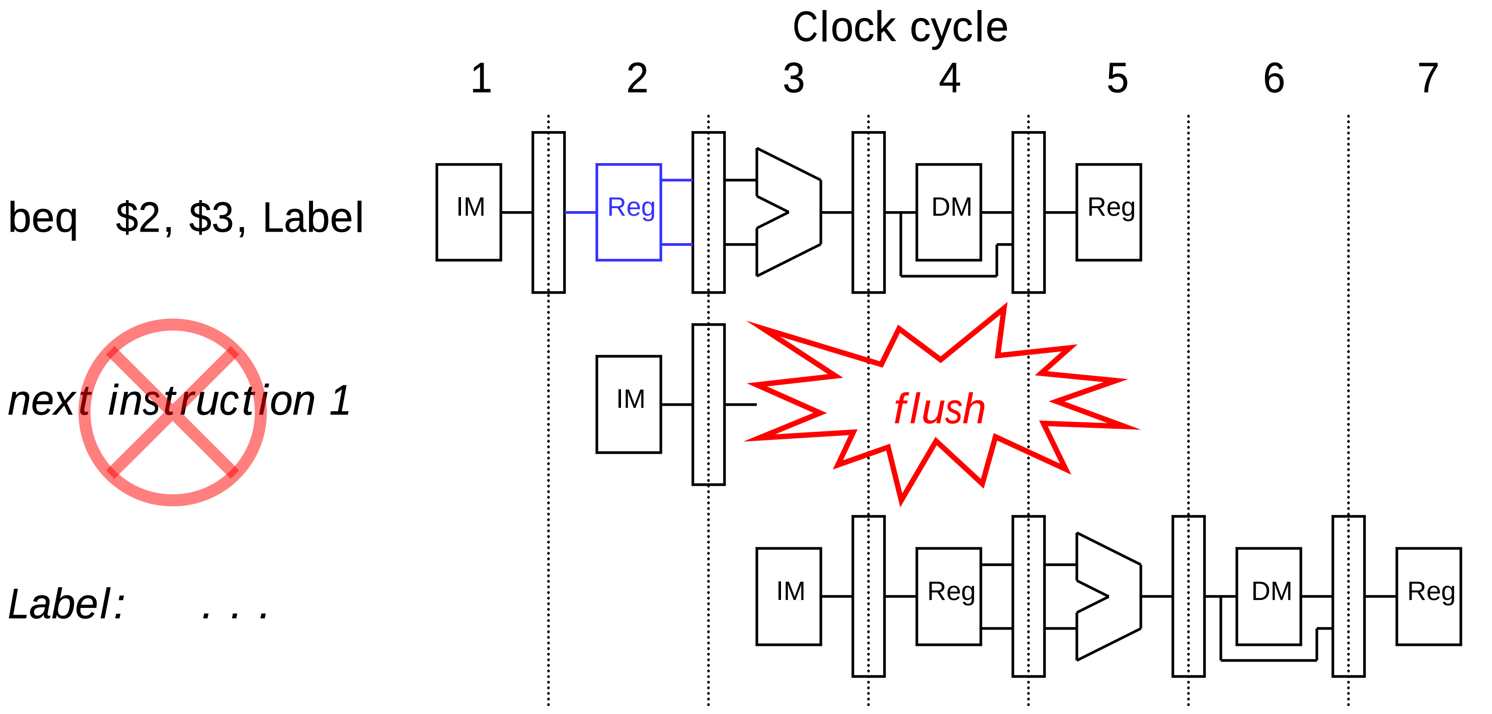


Performance gains and losses

- Overall, branch prediction is worth it.
  - Mispredicting a branch means that two clock cycles are wasted.
  - But if our predictions are even just occasionally correct, then this is preferable to stalling and wasting two cycles for *every* branch.
- All modern CPUs use branch prediction.
  - Accurate predictions are important for optimal performance.
  - Most CPUs predict branches dynamically—statistics are kept at run-time to determine the likelihood of a branch being taken.
- The pipeline structure also has a big impact on branch prediction.
  - A longer pipeline may require more instructions to be flushed for a misprediction, resulting in more wasted time and lower performance.
  - We must also be careful that instructions do not modify registers or memory before they get flushed.

Implementing branches

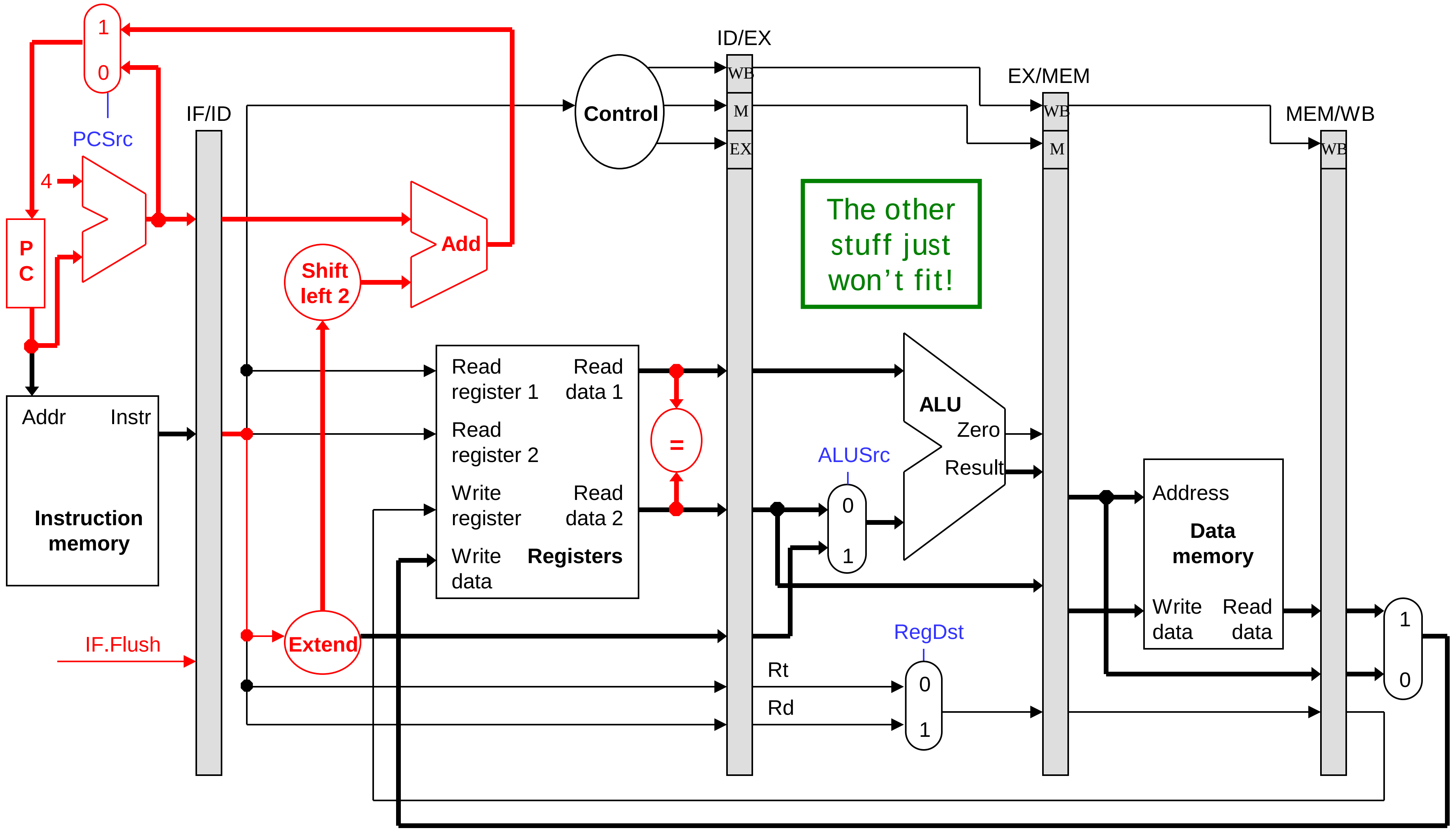
- We can actually decide the branch a little earlier, in ID instead of EX.
  - Our sample instruction set has only a BEQ.
  - We can add a small comparison circuit to the ID stage, after the source registers are read.
- Then we would only need to flush one instruction on a misprediction.



Implementing flushes

- We must flush one instruction (in its IF stage) if the previous instruction is BEQ and its two source registers are equal.
- We can flush an instruction from the IF stage by replacing it in the IF/ID pipeline register with a harmless nop instruction.
  - MIPS uses sll $0, $0, 0 as the nop instruction.
  - This happens to have a binary encoding of all 0s: 0000 .... 0000.
- Flushing introduces a bubble into the pipeline, which represents the one-cycle delay in taking the branch.
- The IF.Flush control signal shown on the next page implements this idea, but no details are shown in the diagram.

Branching *without* forwarding and load stalls



The other stuff just won't fit!

Summary

- Three kinds of hazards conspire to make pipelining difficult.
- Structural hazards result from not having enough hardware available to execute multiple instructions simultaneously.
  - These are avoided by adding more functional units (e.g., more adders or memories) or by redesigning the pipeline stages.
- Data hazards can occur when instructions need to access registers that haven’t been updated yet.
  - Hazards from R-type instructions can be avoided with forwarding.
  - Loads can result in a “true” hazard, which must stall the pipeline.
- Control hazards arise when the CPU cannot determine which instruction to fetch next.
  - We can minimize delays by doing branch tests earlier in the pipeline.
  - We can also take a chance and predict the branch direction, to make the most of a bad situation.