Lab 7: Processor Optimization

Introduction

In this lab, you will improve the performance of the processor you designed in Lab 5. You will implement the same instruction set and use the same memory interface.

The processor from Lab 5 required many cycles to execute each instruction, so its IPC was less than 1. Its datapath was underutilized – while an instruction was being executed (for example, two registers are being added by your ALU), the other parts of the pipeline were laying dormant and not doing anything that cycle. However, it is possible to utilize more of your datapath every cycle. For example, while an instruction is being executed, you could be simultaneously fetching the next instruction from memory. This is called *pipelining* and it can increase the IPC of your processor.

As an example, Figure 1 shows the cycle-by-cycle execution of instructions in an unpipelined 3-cycle processor on the left (not necessarily the one from Lab 5), versus a pipelined one on the right, with operations belonging to the same instruction highlighted using the same colour. The unpipelined processor achieves an IPC of 0.33 whereas the pipelined one can achieve a steady-state IPC of 1 after an initial start-up period – a three-fold improvement. This is only the ideal case, and in practice the IPC will be lower for certain types and sequences of instructions.

#	Operations
1	o_pc_addr = PC PC = PC + 4
2	IR = i_pc_rddata
3	(execute instruction A)
4	o_pc_addr = PC PC = PC + 4
5	IR = i_pc_rddata
6	(execute instruction B)

#	Operations
1	o_pc_addr=PC PC=PC+4
2	o_pc_addr=PC PC=PC+4 IR=i_pc_rddata
3	o_pc_addr=PC PC=PC+4 IR=i_pc_rddata (execute A)
4	o_pc_addr=PC PC=PC+4 IR=i_pc_rddata (execute B)
5	o_pc_addr=PC PC=PC+4 IR=i_pc_rddata (execute C)

Figure 1: Example Unpipelined vs. Pipelined Execution

Your goal in this lab will be to pipeline your existing processor from Lab 5. Your modified design will be simulated in ModelSim using a testbench, running a suite of provided *microbenchmark* programs that measure the processor's IPC in different situations using carefully-selected sequences of instructions. **You must ensure that your design is hardware-synthesizable** (see instructions on *Using the* `harness` *Project with Quartus* below). No hardware will be actually tested on the DE1-SoC board.

In the last, optional part of the lab, you will have a chance to improve your processor's performance in a free-form way and compete with your classmates for bonus marks.

Description

For this part of the lab, your CPU's pipeline will have four stages. It takes one clock cycle for an instruction to move from one stage to the next, and unlike your CPU from Lab 5, multiple stages can be occupied with instructions simultaneously. Figure 2 gives an overview of the CPU's pipeline, showing the names of the stages and the hardware that is accessed (and operations that are performed) during each stage.

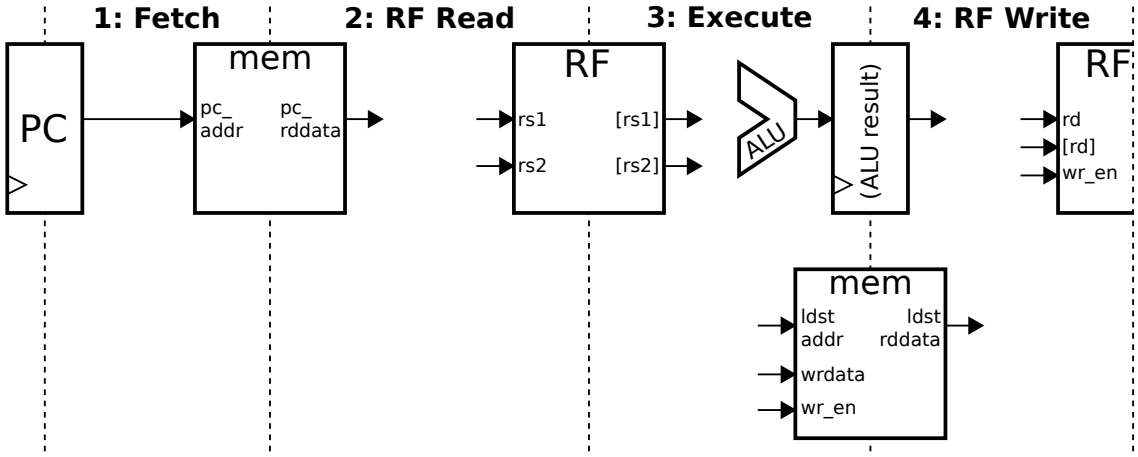


Figure 2: Pipelined processor datapath

Here's a detailed look at what each stage does:

1. **Fetch**: The PC is used to generate a read request to memory and is incremented by 4 (or overridden with a branch target address).
2. **Regfile Read (Decode)**: The 32-bit instruction word returning from memory is decoded into its parts and is used to initiate up to two reads to the register file for register operands `rs1` and `rs2`.
3. **Execute**: The operations specific to each instruction are performed here (everything except writing a result to an RF register). Arithmetic instructions calculate their result using the ALU. Branch and jump instructions may modify the PC. Load and store instructions generate a read or write to memory (simultaneously and independent of the instruction fetch from memory in Stage 1).
4. **Regfile Write (Writeback)**: If the instruction needs to write to the register file, it does so here. Note that the register being written to (called `rd`) can be a *completely different* one than the `rs1` or `rs2` being read at this time by a different instruction. This stage exists separately from Execute because for load instructions, an extra cycle is required to retrieve the data value to write into the RF.

Note that in Figure 2 some components show up twice (the memory and the register file), but are in fact just the same blocks redrawn twice for clarity, to better show their relationships to their pipeline stages. Other pipeline registers that you will need (ex. to retain `rd` until it is needed in stage 4) are not shown. The Fetch and Execute stages need to be able to operate simultaneously – if a load or store instruction is being executed in Stage 3, it must still be possible to fetch an instruction (at a different address) in the *same* clock cycle. The memory interface to your processor in this lab will be **dual-ported**. It's the same memory block, but allows two simultaneous accesses during the same clock cycle.

Table 1 shows the processor's signal interface for this lab, which are the same those you used in Lab 5. Memory Port 1 ('pc') is read-only and is used for fetching instructions, and Port 2 ('ldst') is used for loads and stores and can be read and written. The timing for each port's signals is identical to Lab 5. You can assume that reads and writes will always be accepted by the memory without stalling, and that read data always arrives one cycle after the address, read enable and byte enable are provided.

Signal	Direction	Size	Description
`clk`	input	1	Clock
`reset`	input	1	Active-high reset
`o_pc_addr`	output	32	Address (in bytes)
`o_pc_rd`	output	1	PC Read enable
`i_pc_rddata`	input	32	PC Read data
`o_pc_byte_en`	output	4	PC Byte enable
`o_ldst_addr`	output	32	Address (in bytes)
`o_ldst_rd`	output	1	Load/Store Read enable
`i_ldst_rddata`	input	32	Load/Store Read data
`o_ldst_wr`	output	1	Load/Store Write enable
`o_ldst_wrdata`	output	32	Load/Store Write data
`o_ldst_byte_en`	output	4	Load/Store Byte enable
`o_tb_regs`	output	32 x 32	Register values

Table 1: Processor signal interface

Part I: Basic Pipelined Processor

The implementation of the processor in this lab is more complicated than in Lab 5. It is similarly divided into multiple parts that build on each other. Note: you will not submit a version of your processor for each stage. You will sumbit one version that satisfies all the requirements and optionally a second version for the bonus (see Part V). In this first part, you will create an initial version of the pipelined processor that can correctly execute very simple programs consisting of only arithmetic instructions, `lui` and `auipc`. You can first implement this assuming independent instructions that do not read registers recently written to by an earlier instruction. In Part II you will add bypassing.

Project Setup

Start with your submission for Lab 5. In addition, there is a Quartus project called `harness` which is used to make sure that the Verilog/SystemVerilog you write is synthesizable on an FPGA. We discuss using the `harness` project further below.

Implementation

Write a control and datapath module for this new processor, and instantiate/connect them in the top-level `cpu` module. Use your Lab 5 code as a reference and starting point, especially the datapath. It might be helpful to reorganize your datapath by pipeline stage.

You will also need to add registers into your pipeline to 'remember' certain information from stage to stage. For example, Stage 4 still needs to know the register number being written to, and possibly the type of instruction as well. It must get these from Stage 3, which itself gets a copy from Stage 2 where the instruction word was actually read from memory.

A 'valid' register per-stage[1] will allow you to know if a particular stage is actually occupied or not, enabling/preventing that stage from making changes to registers, flags, or memory. For example, when the CPU is first reset, no instructions have been fetched yet, and Stages 2/3/4 are empty, and a zeroed valid flag for, say, Stage 4, will prevent a register file write during that cycle.

The control module for your CPU will require the most extensive re-write. Previously, you used a state machine, which can only be in one state at a time. This worked fine as a single thread of control for your multi-cycle CPU. For a pipelined CPU, multiple instructions exist in different phases of their execution simultaneously in your datapath. You can use multiple state machines, or ideally, abolish the use of state

[1] You are **not** required to use a valid register if you choose to implement this some other way.

machines altogether and control each pipeline stage independently based solely on the signals produced by the previous stage.

Testing

The testbench does two things: it ensures your processor's output is correct, and it measures its IPC to make sure its performance is high enough. When you run the testbench, it will give a Pass or Fail for both the correctness and performance categories. The goal of Part I is for your processor to be able to pass the first test only (`0_basic`) with an IPC of 1.0. This means your processor should complete one instruction every cycle in the steady state. If functional correctness fails, the testbench will print out the expected vs. observed values of each register.

At the bottom of `tb.sv` in the main `initial` block, you can comment out the other test cases to just test `0_basic` at first. After it passes, compile the `harness` Quartus project to verify that the Verilog/SystemVerilog code is hardware-synthesizable and causes no compilation errors or warnings about latches or combinational loops (see *Using the `harness` Project with Quartus*).

Part II: Dependent Instructions

In this part, you will enhance your processor to be able to handle sequences of dependent instructions. Consider this code:

```
ori  s0, zero, 1
addi s1, s0, 2
sub  s2, s0, s1
add  s0, s1, s1
```

The instructions are fetched sequentially and start moving forward in the pipeline. When the `ori` reaches Stage 4 (RF Write), the `addi` is in Stage 3 (Execute) and the `sub` is in Stage 2 (RF Read). If proper measures are not taken, this code will not execute correctly, because the `addi` and `sub` instructions will use the **old** value of `s0`, which the `ori` in Stage 4 hasn't had a chance to commit to the register file yet.

Stages 2 and 3 must be able to use the value of a register (`s0` in this case) from Stage 4 *before* it has been written to the register file. This value, which is connected to the Register File's write port, must be *forwarded* [2] to Stages 2 and 3. To do this, you need to create hardware that:

1. Recognizes when a register that's being read as an input in Stage 2, or Stage 3, is simultaneously being written to in Stage 4
2. Overrides Stage 2 and/or Stage 3's contents of `[rs1]` and/or `[rs2]` with Stage 4's `[rd]`.

After you add the forwarding logic, you should be able to pass the `1_arithdep` test. Again, verify that the Verilog/SystemVerilog code is hardware-synthesizable.

Part III: Branches

Branches here refer to any of the eight jump or branch instructions that can change the default control flow of the processor: `jal jalr beq bne blt bge bltu bgeu`. They pose a challenge for pipelining because it takes until the Execute stage to determine that an instruction *is* a branch instruction, whether or not that branch is taken, and what value the PC should be set to. Until that information is available, the Fetch and RF Read stages can only *guess* that the next two instructions are located at PC+4 and PC+8 [3]. This leap of faith is required if you have any hope of reaching an IPC of 1.

[2] This is sometimes referred to as bypassing. The two terms are equivalant.

[3] Modern processors use branch prediction to make better guesses. You can assume that instructions are not branches and fetch from PC+4.

For this lab, your Fetch stage can assume that the next PC is PC+4 unless the Execute stage knows for sure that it needed to be something else. This way, your processor can continue fetching 1 instruction per cycle and hoping that this assumption is correct. However, if there is a branch instruction in the Execute stage, and this branch ends up being taken, then that means that the two instructions behind the branch in the pipeline are *not actually the next 2 instructions*. In this scenario, you must make sure these two instructions never make it to the Execute stage. This will create an empty bubble of 2 cycles in your processor until the correct program flow is re-established.

A branch is considered 'taken' if it's either a jump (`jal jalr`), or if it's a conditional branch (`beq bne blt bge bltu bgeu`) **and** the condition is true. This changes the PC to something else than PC+4[4]. If the Execute stage contains a taken branch instruction, then at the end of that cycle:

1. The PC will be set to its correct value specified by the branch instruction and will generate a correct fetch *next* cycle. That means that the instruction being fetched (address being read from) next cycle will be the correct instruction to follow the branch.
2. Stage 2 will not be considered occupied/valid.
3. Stage 3 will not be considered occupied/valid.

If you used a 'valid' register for each pipeline stage, as suggested in Part I, then steps 2 and 3 should be straightforward. After modifying your processor to support branches, then you should be able to run `2_branch_nottaken` and `3_branch_taken`. Taken branches will degrade the IPC of the processor, and this is expected. The `3_branch_taken` test has a minimum expected IPC of 0.33 to reflect this.

Again, verify that the Verilog/SystemVerilog code is hardware-synthesizable.

Part IV: Loads/Stores

Finally, you will add support for load and store instructions. Make sure that you include forwarding logic for load and stores (and for jumps and branches).

You should now be able to run `4_memdep`. Again, verify that the Verilog/SystemVerilog code is hardware-synthesizable.

Part V: Competition (BONUS)

The bonus is **optional**. We recommend successfully completing parts I to IV before trying the bonus.

For the bonus, modify the architecture of your pipelined processor to improve its performance even further, and compete against your classmates. The top 10 performing designs in the class earn bonus marks. We will evaluate performance using two metrics:

- The average number of instructions per clock cycle (IPC) that the processor can execute
- The processor's clock frequency (f_{max})

To encourage designs that use architectural innovation to increase IPC we will determine your processor's performance as

$$score = f_{max} \cdot IPC^3$$

The IPC will be calculated as the geometric average over a set of test programs. The set of test programs will be different from the microbenchmarks in the tester. We will release some, but **not all**, of these programs for you to test with. Instructions on determining f_{max} for your processor using Quartus are further below.

[4]Don't worry about the case where the target PC specified by the branch *is* actually PC+4. Consider this a taken branch, and make an optimization in Part V if you wish.

You are free to modify the processor as you wish, including adding or removing pipeline stages. It must still correctly execute all the instructions and be able to be compiled in the Quartus `harness` project. You can not change the processor's signal interface. **Your bonus design is not required to pass the IPC performance requirements of benchmarks 0 to 4**. For example, a branch predictor might improve performance on average but degrade performance for one of the benchmarks – this is OK. Here are some possible things you can do to improve performance:

- Add more pipeline stages.
- Have fewer pipeline stages.
- Look for signals that can be calculated a pipeline stage earlier than they currently are.
- Try to predict the outcome of branches in a smarter way than "assume always not-taken".
- Find a way to execute more than one instruction simultaneously, for an IPC greater than 1.
- You know what the instruction is by Stage 2 – see if you can do any work there.
- Find the critical path of your circuit as reported by TimeQuest.

Using the `harness` Project with Quartus

We have provided a Quartus project for you which will allow you to check that your processor is hardware-synthesizable. Your processor **must** compile using the `harness` project. To check if your code is synthesizable:

1. Extract the Quartus project files from cpu_harness.zip and open the project [5].
2. Go to *Project* → *Add/Remove Files in Project...* and click the '...' icon to browse for files (see figure 3). Select the file(s) containing your processor implementation. The files should then be listed with the `harness` files. Click OK.

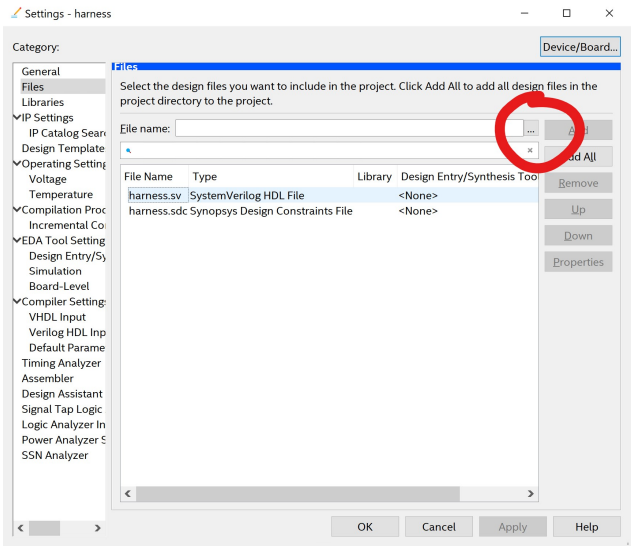


Figure 3: Adding files to Quartus project.

[5] On the U of T Windows machines, opening the project from the W drive works (in my experience). Quartus may have trouble with finding files in networked storage for some other locations.

3. Run *Analysis & Synthesis* then *Fitter (Place & Route)*.
4. Verify that the compilation was successful, that the code is hardware-synthesizable and causes no compilation errors or warnings about latches or combinational loops.

Determining Maximum Frequency (f_{max})

This section is only required for students trying the bonus.

To obtain your f_{max}, first compile your processor with the above steps. Then run *Timing Analysis*. From the compilation report select *Timing Analyzer → Slow 1100mV 85C Model → Fmax Summary*.

Improving Maximum Frequency

This section is only required for students trying the bonus.

To improve f_{max} you must decrease the critical path. The critical path is the path that contains the longest delay between a pair of registers. We provide here a brief overview of determining what the critical path is. It is up to you to determine how to decrease it. Usually by reducing the total amount of dependent logic on the critical path or by breaking up long stages into shorter parts.

1. From the compilation report select *Timing Analyzer → Slow 1100mV 85C Model → Timing Closure Recommendations*. This will display pairs source and sink registers which have the longest delay between them. Note: the names of these registers will be different from, but have some resemblance to, the names you used in your Verilog/SystemVerilog.
2. Click *Report recommendations for this path* beside the top entry in the table. Click OK on the pop-up titled *Report Timing Closure Recommendations*.
3. Here Quartus is telling you what the longest paths are and the cause. Also check out the other pages in the *Reports* section on the left side of Quartus.
4. Click *Reports → Timing Closure Recommendations → Detailed Per-Path Results*. Click *report timing* in the top row of the table and click *Report Timing* in the pop-up.
5. Under the *datapath* tab (*Data Arrival Path*) you can see a list of all the intermediate wires between the registers. These can help you determine what logic the signal is passing through between the registers.
6. Right click in the *Data Arrival Path* table and select *Locate Path → Locate in Technology Map Viewer*. This will show a pictoral representation of the path.

Netlist Viewer

This section is not required.

You may be interested in how your processor is translated into hardware. To view a schematic of your design click *Tools → Netlist Viewers → RTL Viewer* or *Technology Map Viewers (Post-Fitting)*. The first is more readable but has not been optimized, but the second is a better representation of the actual hardware that would be implemented.

Testing Your Processor

You must ensure that your design is hardware-synthesizable. Specifically, **it must be synthesizable using the Quartus `harness` project**. See instructions above on *Using the `harness` Project with Quartus*.

You can test your processor almost exactly as you tested your processor for lab 5. The testbench (tb.sv) is almost exactly the same and the programs are the same. **Note: the programs for this lab have**

different performance requirements than lab 5 which turns up in the hex files not being quite identical (they were compiled using the `-L7` flag).

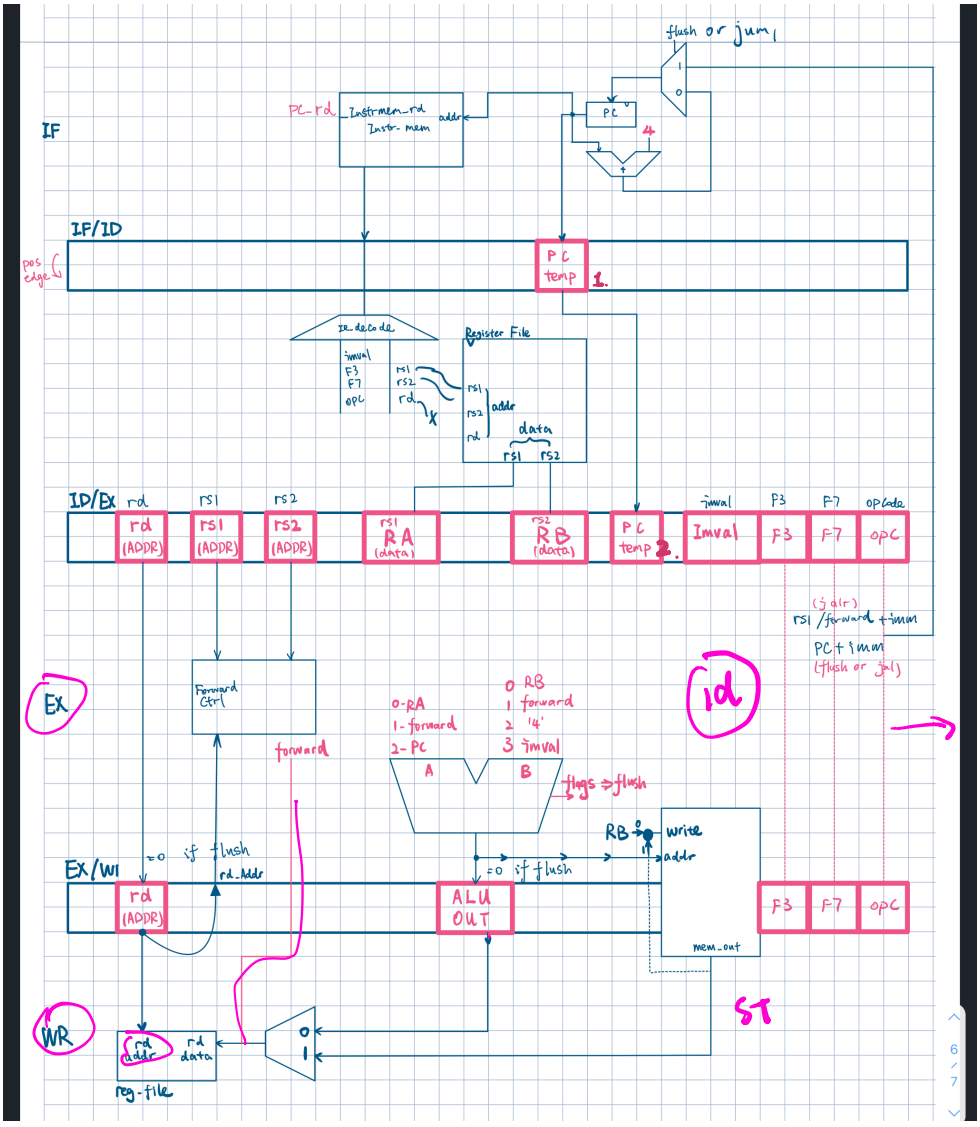
For students attempting the bonus we have released some of the programs that we will use to test your IPC (`7_gemm`, `9_dfs`, `10_sssp`). You can use these to test your processor's IPC, but for more accurate results we recommend you also create some more test programs of your own.

Submission

To help you move forward in stages this lab is split into 4 parts plus a bonus. These stages are just a suggestion so feel free to implement in whatever order you prefer.

For the automarker you will submit two files: part1.sv and part2.sv. The first is your pipelined processor implementing parts I to IV. The second is optional and is your processor for the bonus submission. If you choose not to try the bonus, you are not required to submit a `part2.sv` file. Submit using the following command: `submitece342s 7 part1.sv part2.sv`

Processor designs must successfully compile in the Quartus harness project to receive full marks. For the bonus, the 10 students with the highest performance figures will be awarded up to an additional 5% on their final course grades, depending on the performance of their processor.



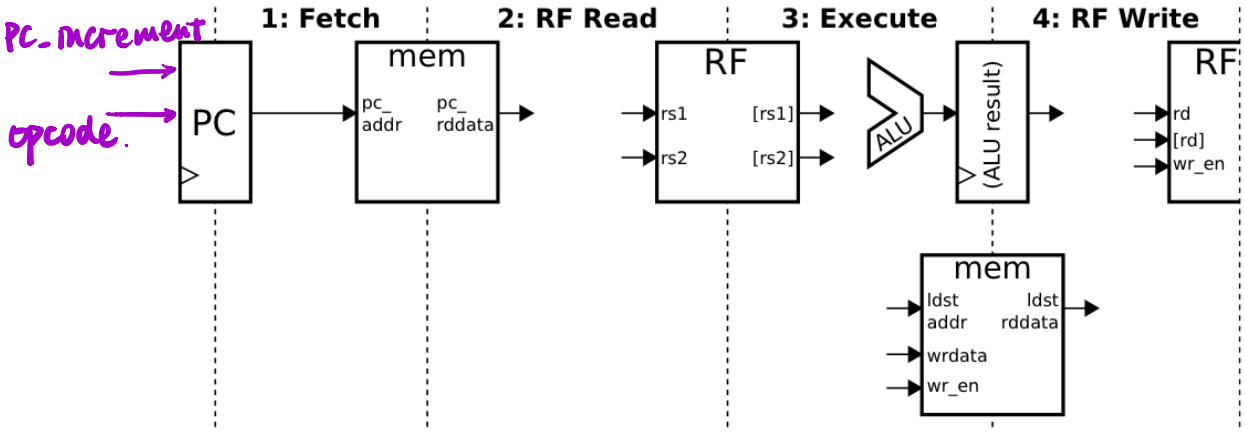
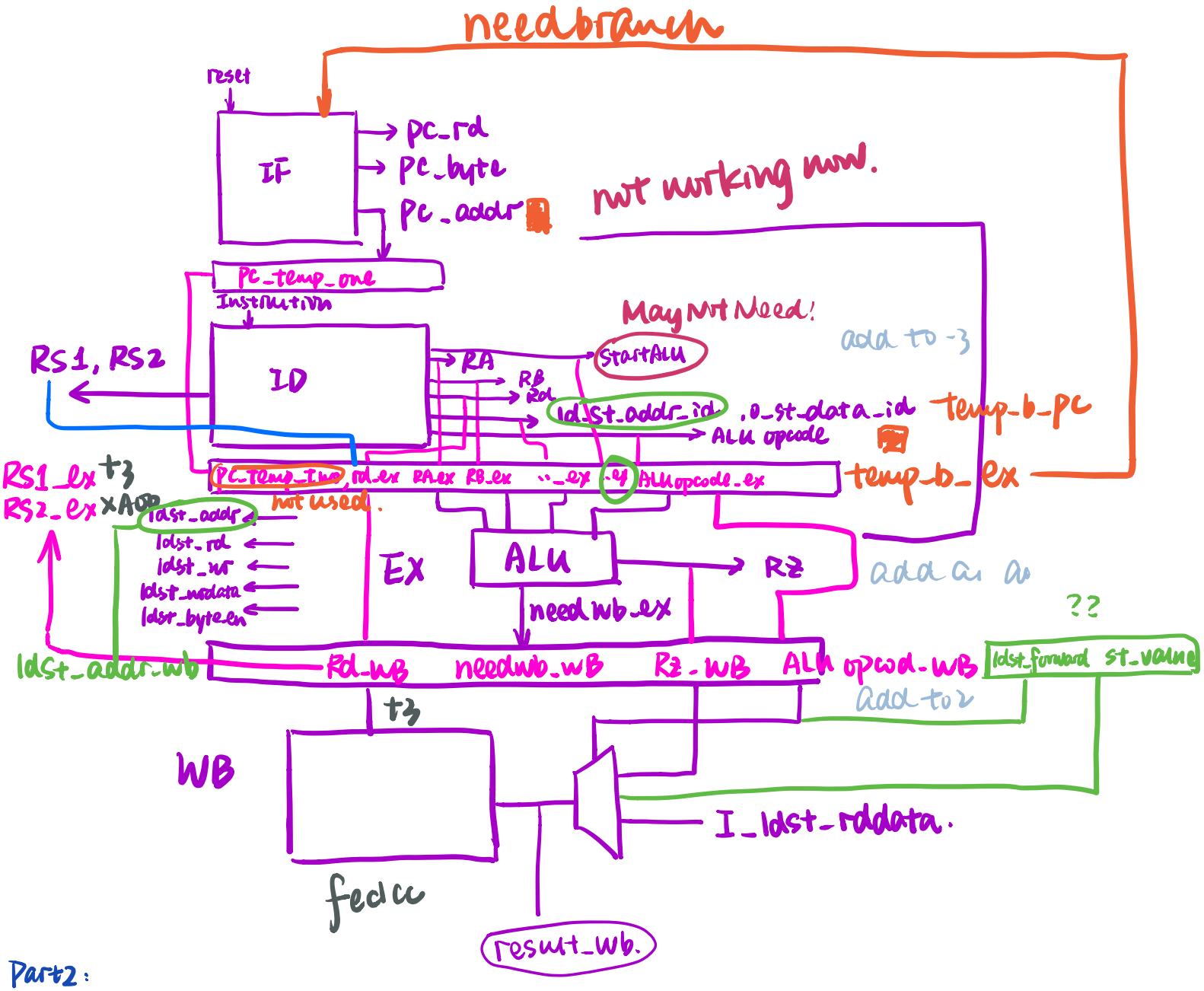


Figure 2: Pipelined processor datapath



Part 1:

x3 → gp
x5 → t0
x11 → a1
x14 → a4
x15 → a5
x28 → t3

x10 → a0

Part 3:

0
4
(jal) 8 ra = c
c
10
(jal) 14
18
1c
(jalr) 20
24
28
2c
(jalr) 30
34
38
3c
40
44
48
4c
50
54
58
5c
60
64
68
6c
70
74
78
7c
80
84
88
8c
90
94
98
90
a0

s7 = 0x20

ra = 34 jump to 20

s5 → x21
s6 → x22

a2-6 → x12-16

a2 = -4
a3 = -1
a4 = 0
a5 = 2
a6 = 3

	A	B
1	x0	zero
2	x1	ra
3	x2	sp
4	x3	gp
5	x4	tp
6	x5	t0
7	x6	t1
8	x7	t2
9	x8	s0
10	x9	s1
11	x10	a0
12	x11	a1
13	x12	a2
14	x13	a3
15	x14	a4
16	x15	a5
17	x16	a6
18	x17	a7
19	x18	s2
20	x19	s3
21	x20	s4
22	x21	s5
23	x22	s6
24	x23	s7
25	x24	s8
26	x25	s9
27	x26	s10
28	x27	s11
29	x28	t3
30	x29	t4
31	x30	t5
32	x31	t6
33		

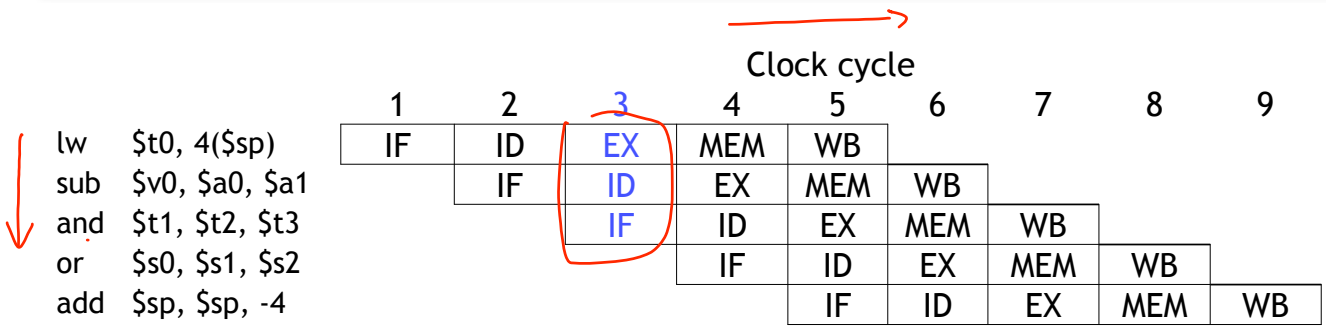
ldst

x2 sp

x21 s5-t6
-x31

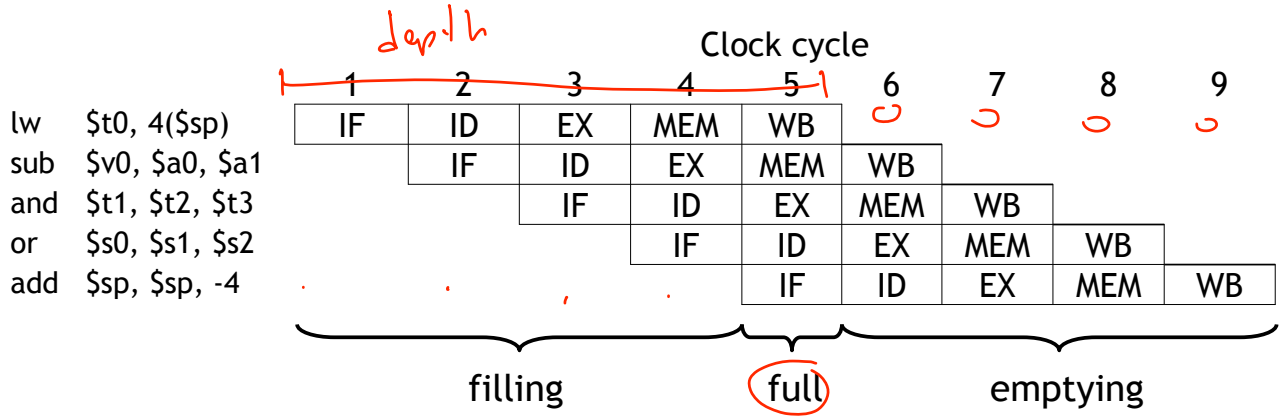
1470 ADD R型
1480
1520

A pipeline diagram



- 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" instruction is in its Execute stage.
  - Simultaneously, the "sub" is in its Instruction Decode stage.
  - Also, the "and" instruction is just being fetched.

Pipeline terminology



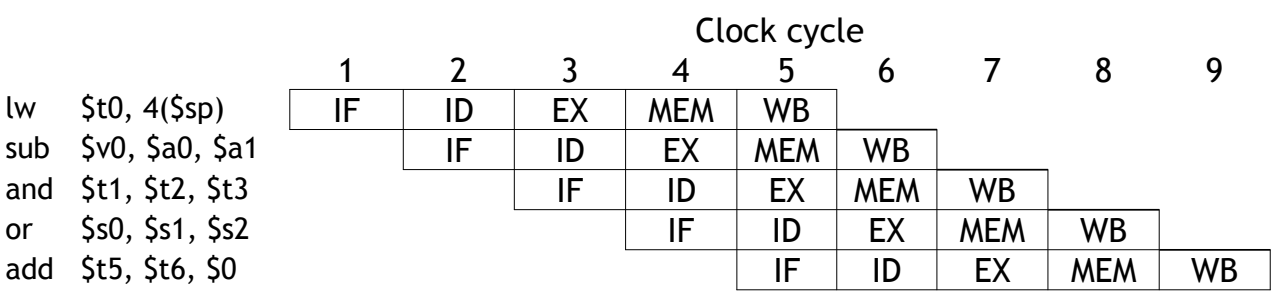
- The pipeline depth is the number of stages—in this case, five.
- In the first four cycles here, the pipeline is filling, since there are unused functional units.
- In cycle 5, the pipeline is full. Five instructions are being executed simultaneously, so all hardware units are in use.
- In cycles 6-9,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

Pipelined datapath and control

- Now we'll see a basic implementation of a pipelined processor.
  - The datapath and control unit share similarities with both the single-cycle and multicycle 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.



Pipelined Datapath

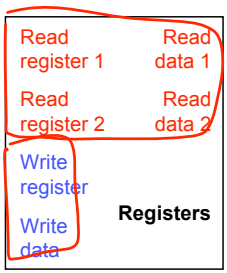
- The whole point of pipelining is to allow multiple instructions to execute at the same time.
- We may need to perform several operations in the same cycle.
  - Increment the PC and add registers at the same time.
  - Fetch one instruction while another one reads or writes data.

		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

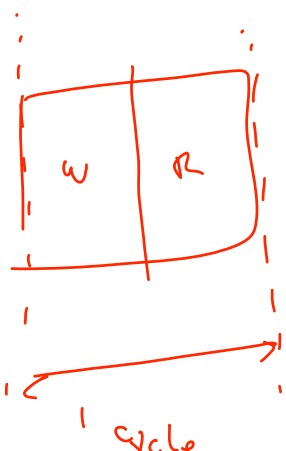
- Thus, like the single-cycle datapath, a pipelined processor will need to duplicate hardware elements that are needed several times in the same clock cycle.

One register file is enough

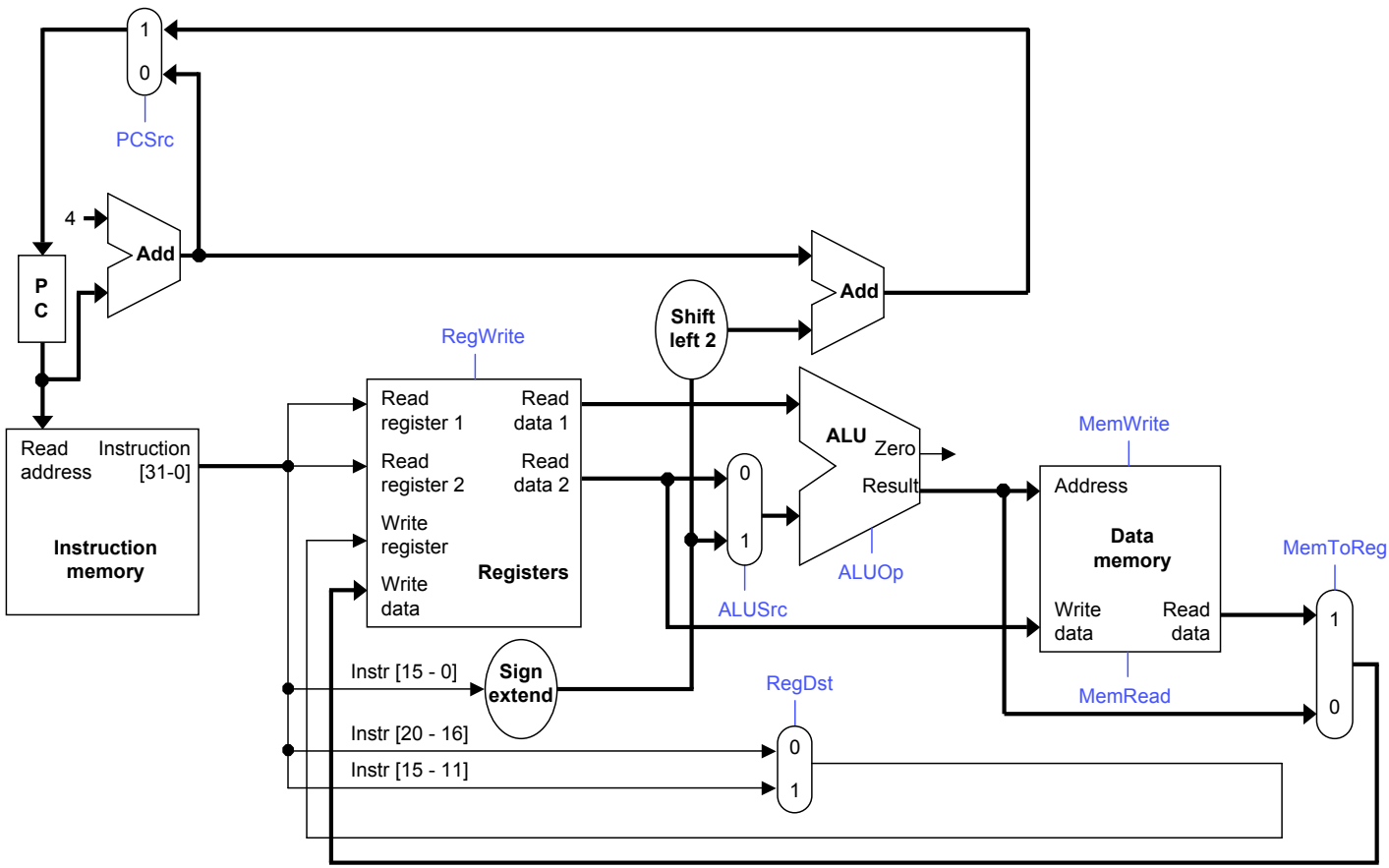
- We need only one register file to support both the ID and WB stages.



- Reads and writes go to separate ports on the register file.
- Writes occur in the first half of the cycle, reads occur in the second half.



Single-cycle datapath, slightly rearranged

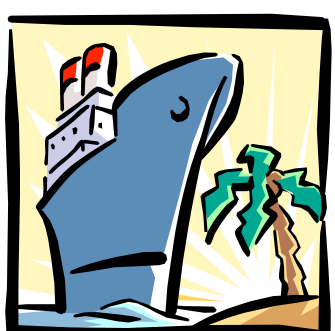


What's been changed?

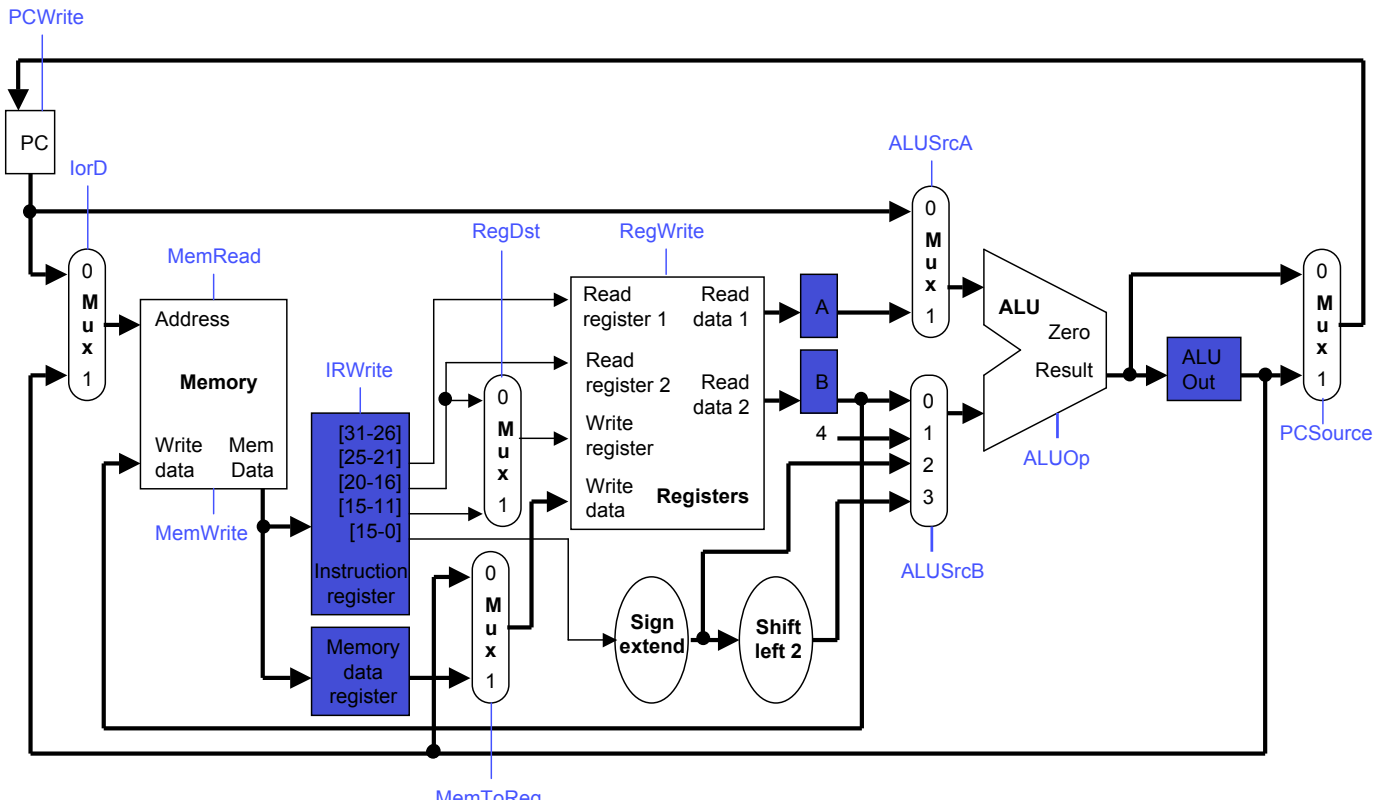
- Almost nothing! This is equivalent to the original single-cycle datapath.
  - There are separate memories for instructions and data.
  - There are two adders for PC-based computations and one ALU.
  - The control signals are the same.
- Only some cosmetic changes were made to make the diagram smaller.
  - 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.

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 added several intermediate registers to the multicycle datapath to preserve information between stages, as highlighted on the next slide.



Registers added to the multi-cycle



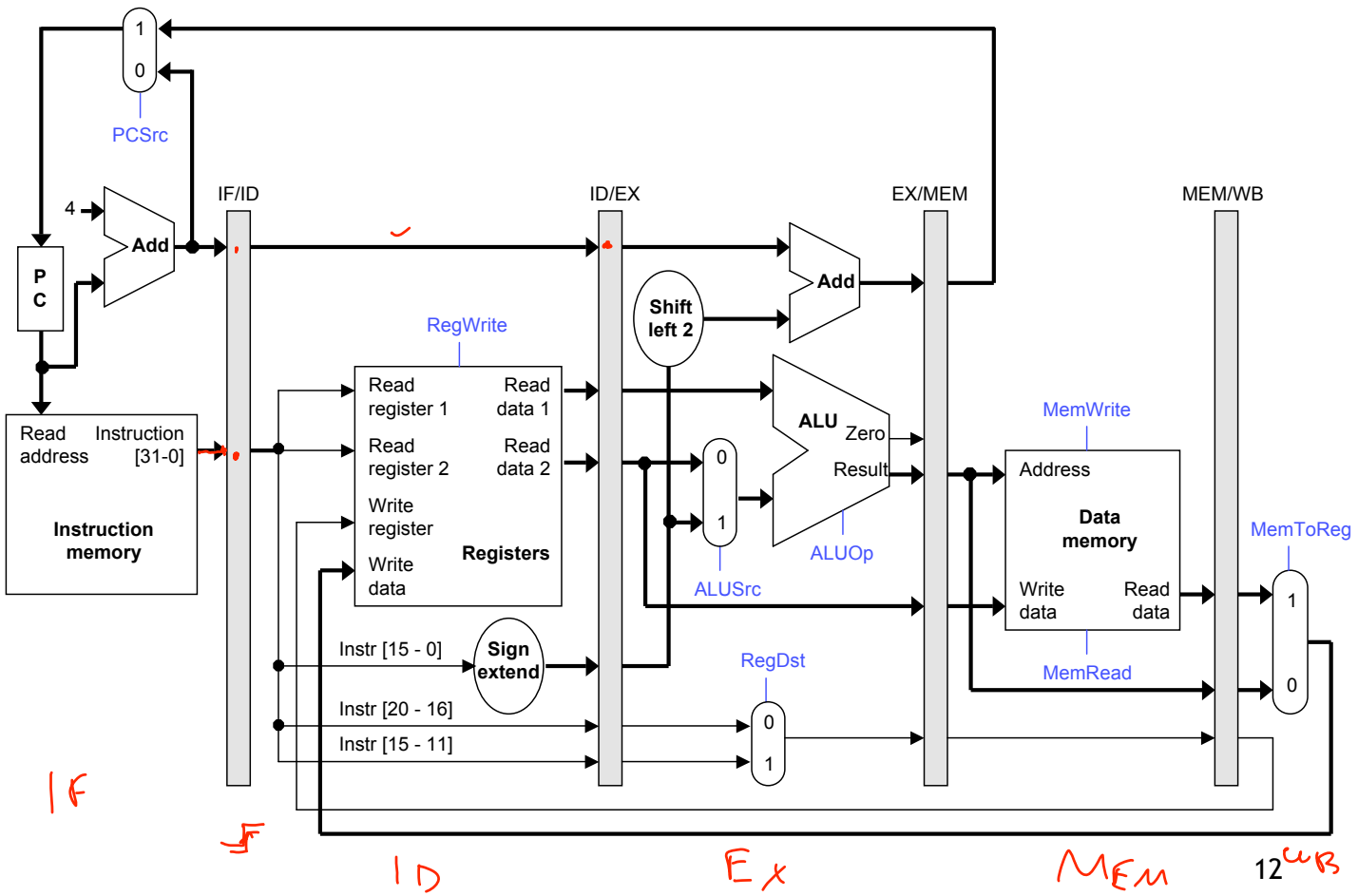
Pipeline registers

- We'll add intermediate registers to our pipelined datapath too.
- 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.

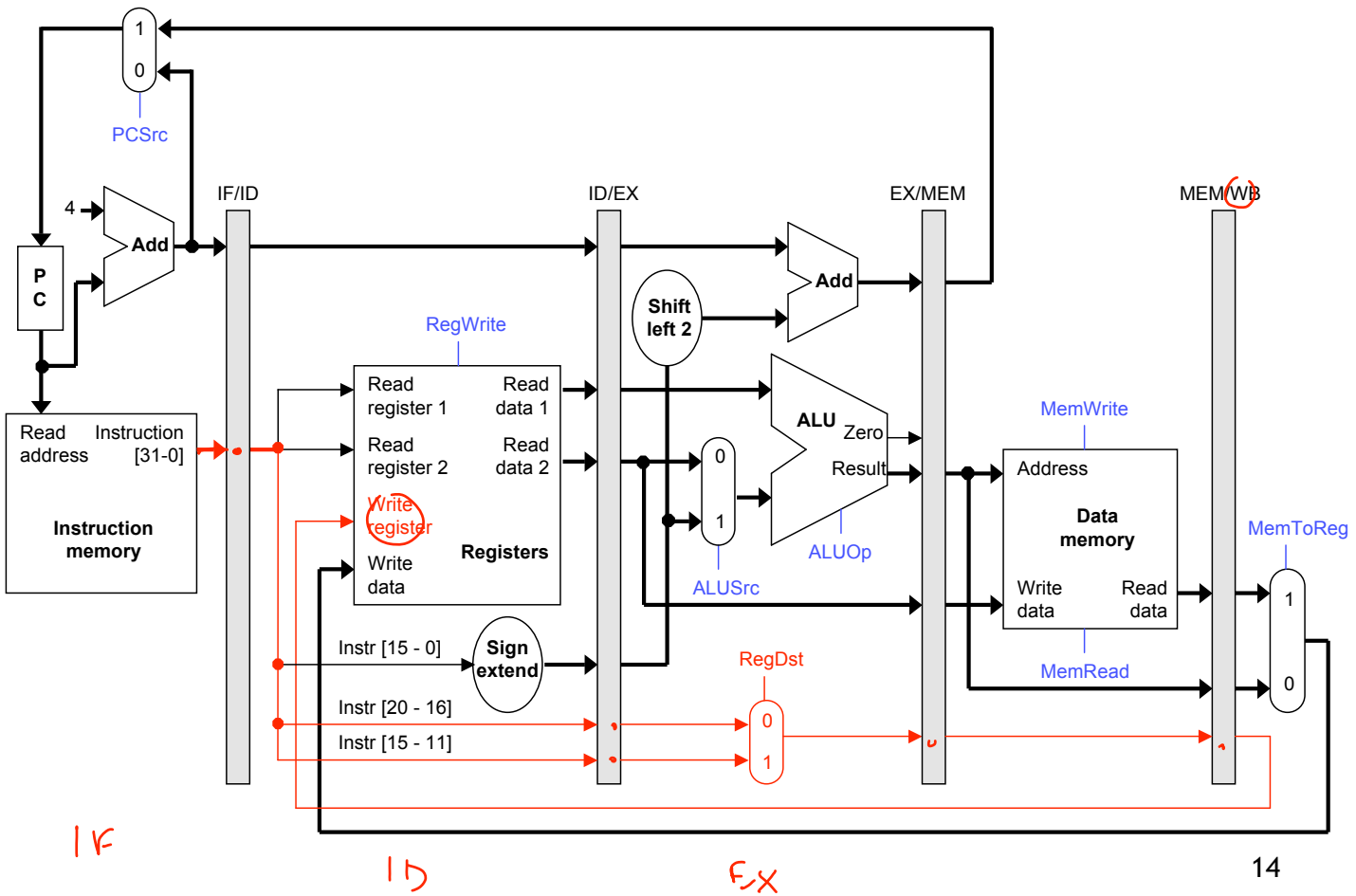
Pipelined datapath



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.
- Why can't we keep a single instruction register like we did in the multi-cycle data-path?

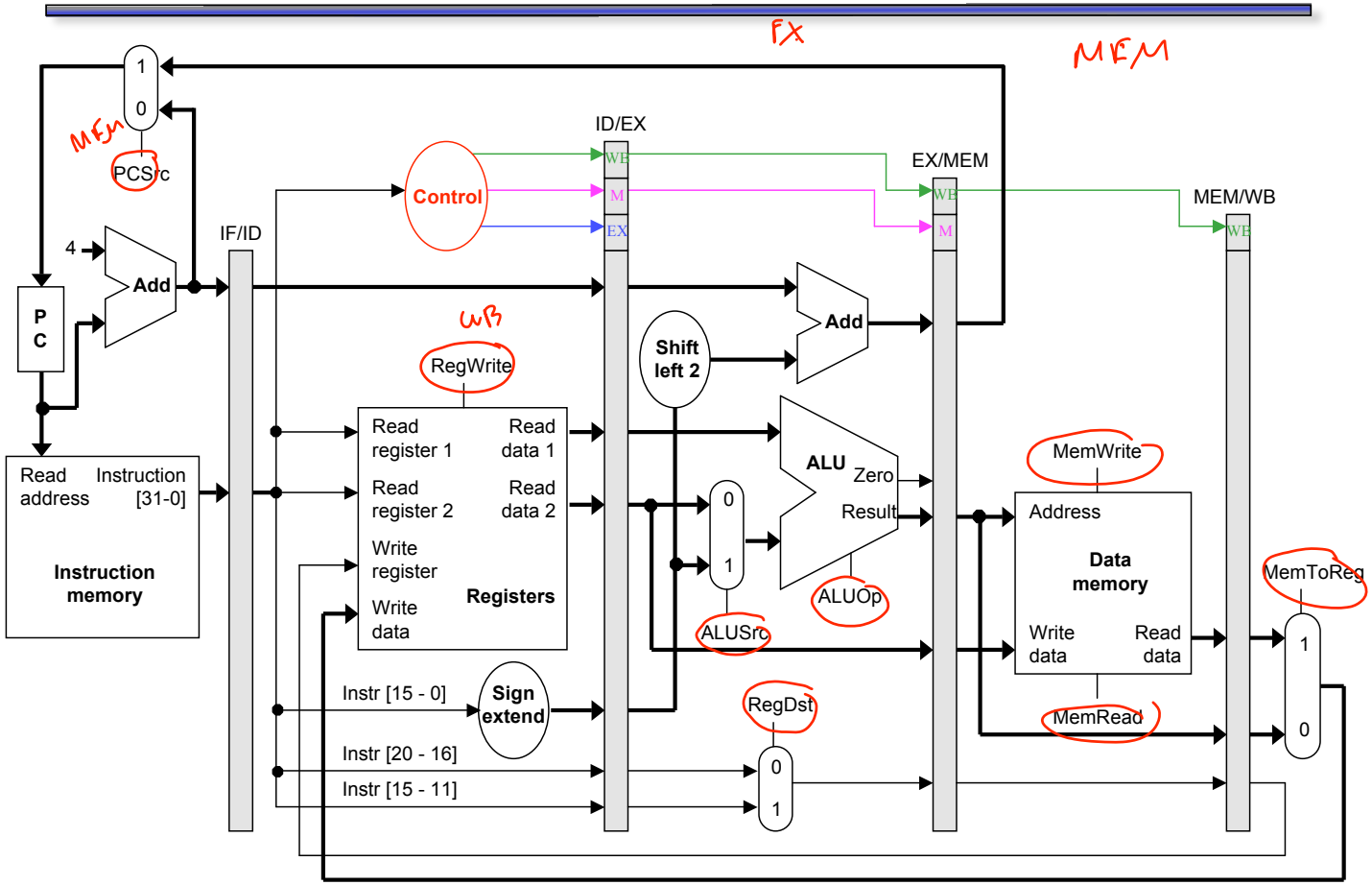
The destination register



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.

Pipelined datapath and control

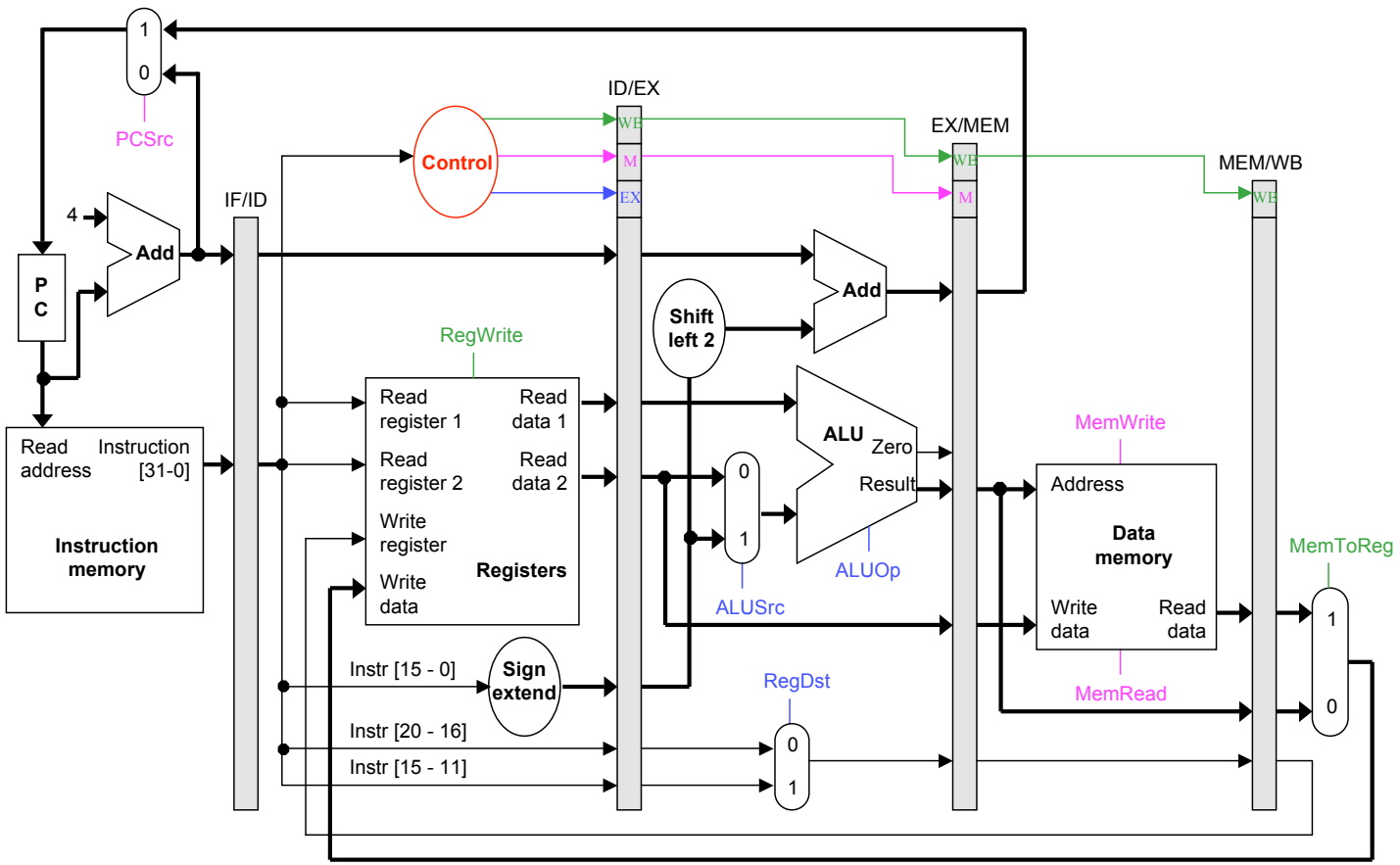


What about control signals?

- 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.

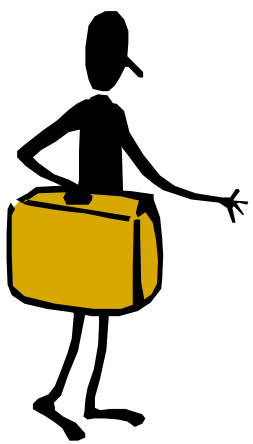
Stage	Control signals needed		
EX	ALUSrc	ALUOp	RegDst
MEM	MemRead	MemWrite	PCSrc
WB	RegWrite	MemToReg	

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.

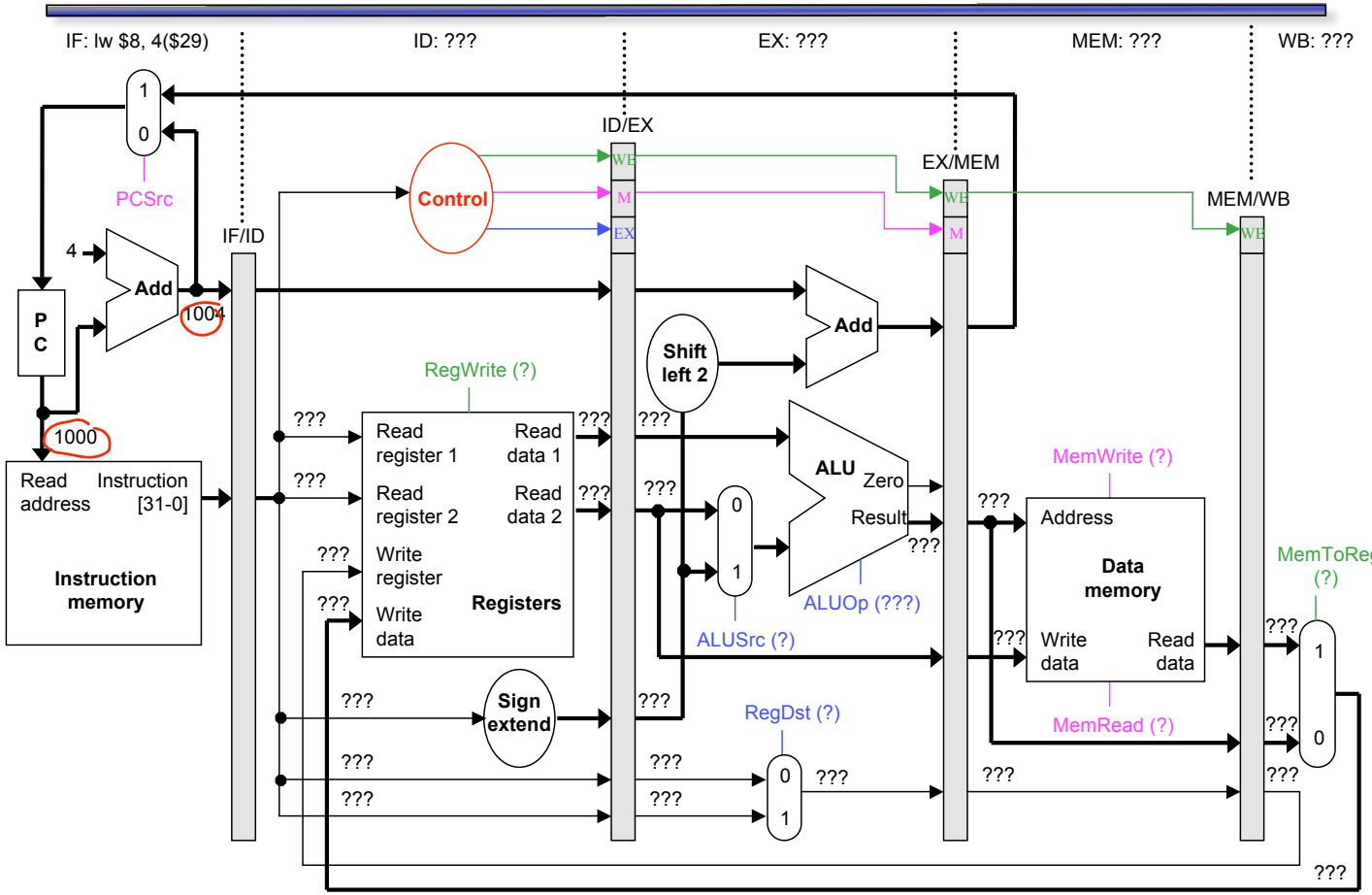
addresses in decimal

```
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
```

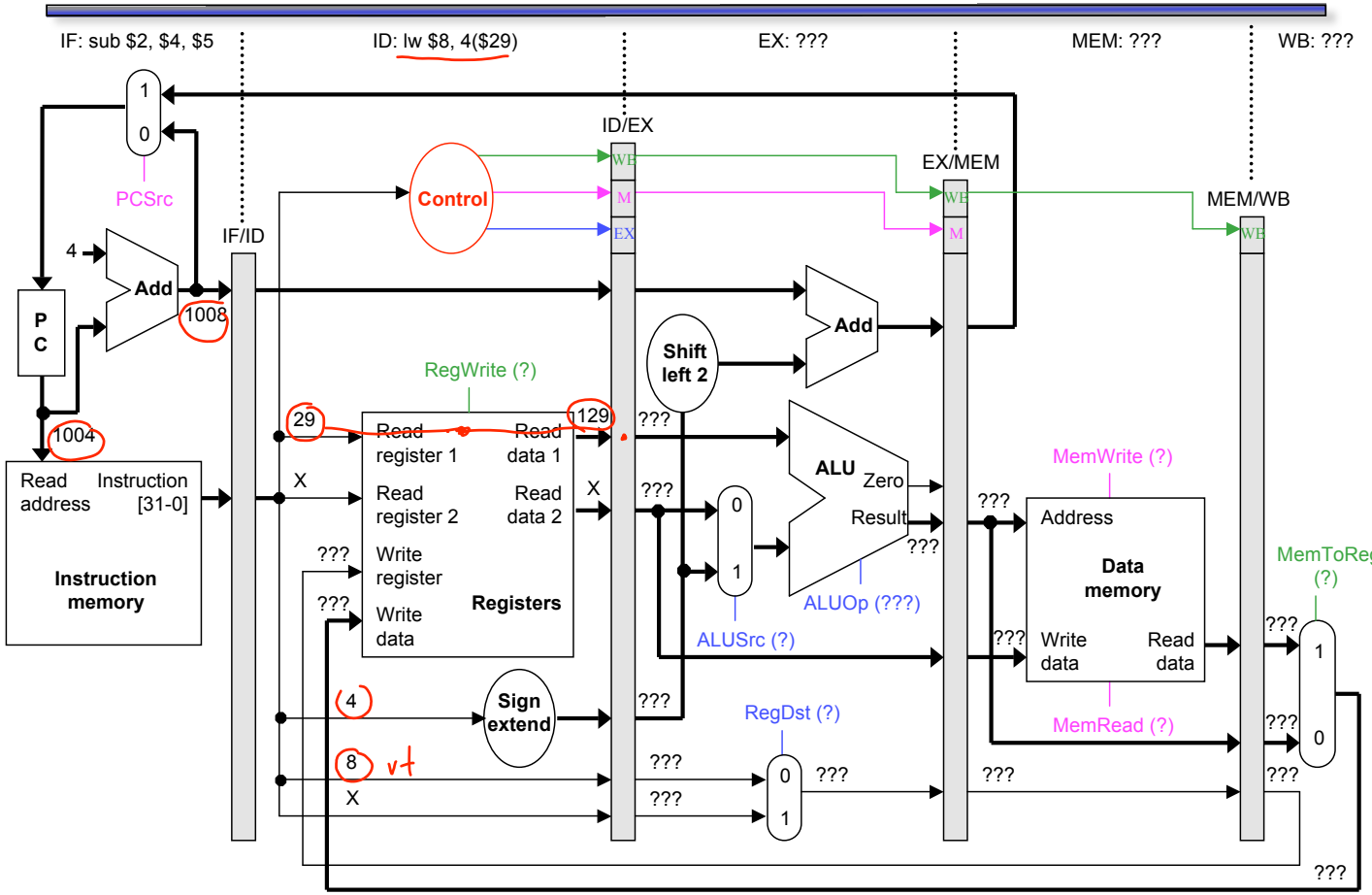
9 cycles
4 to fill
5 to complete

- 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.

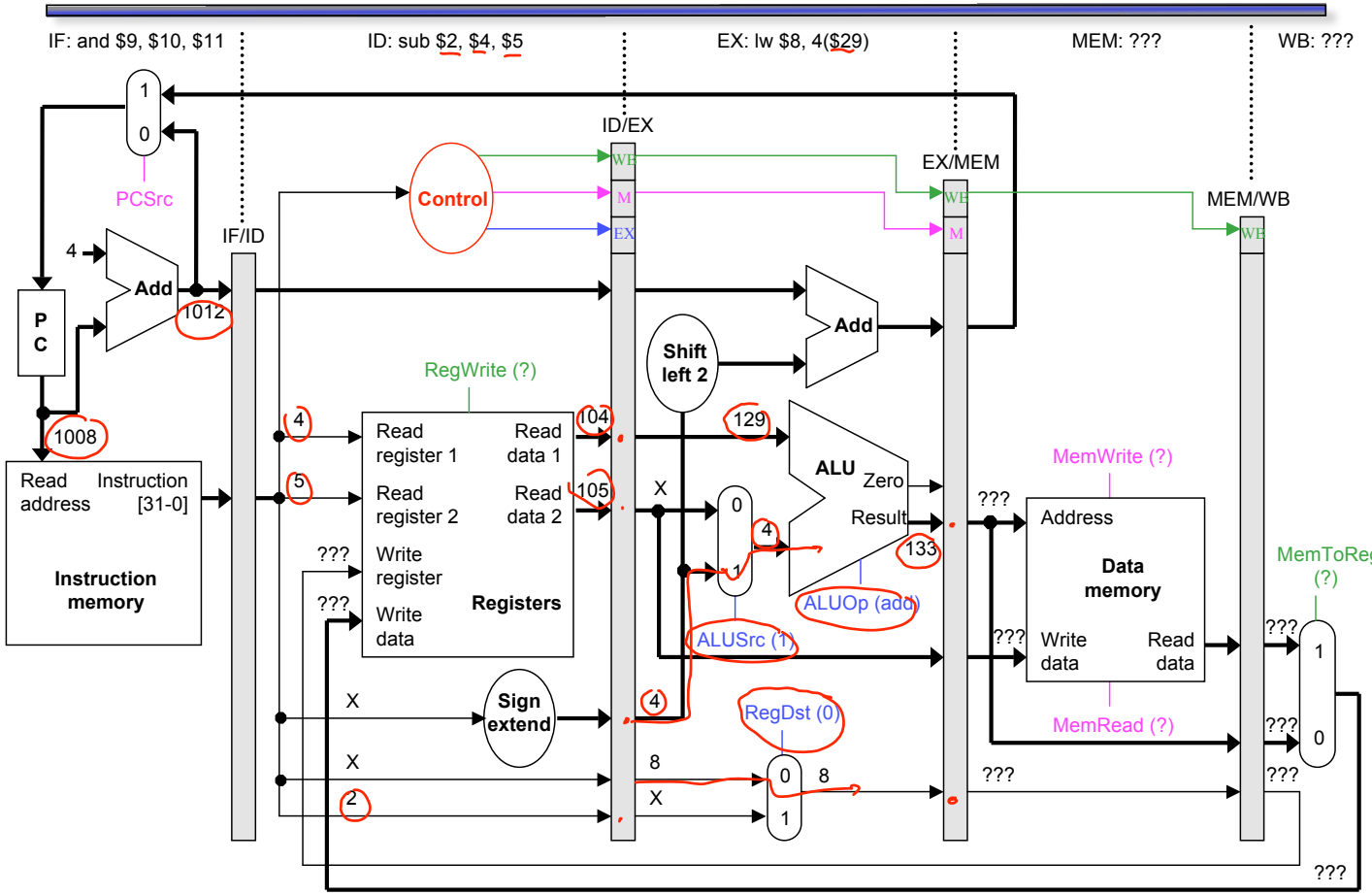
Cycle 1 (filling)



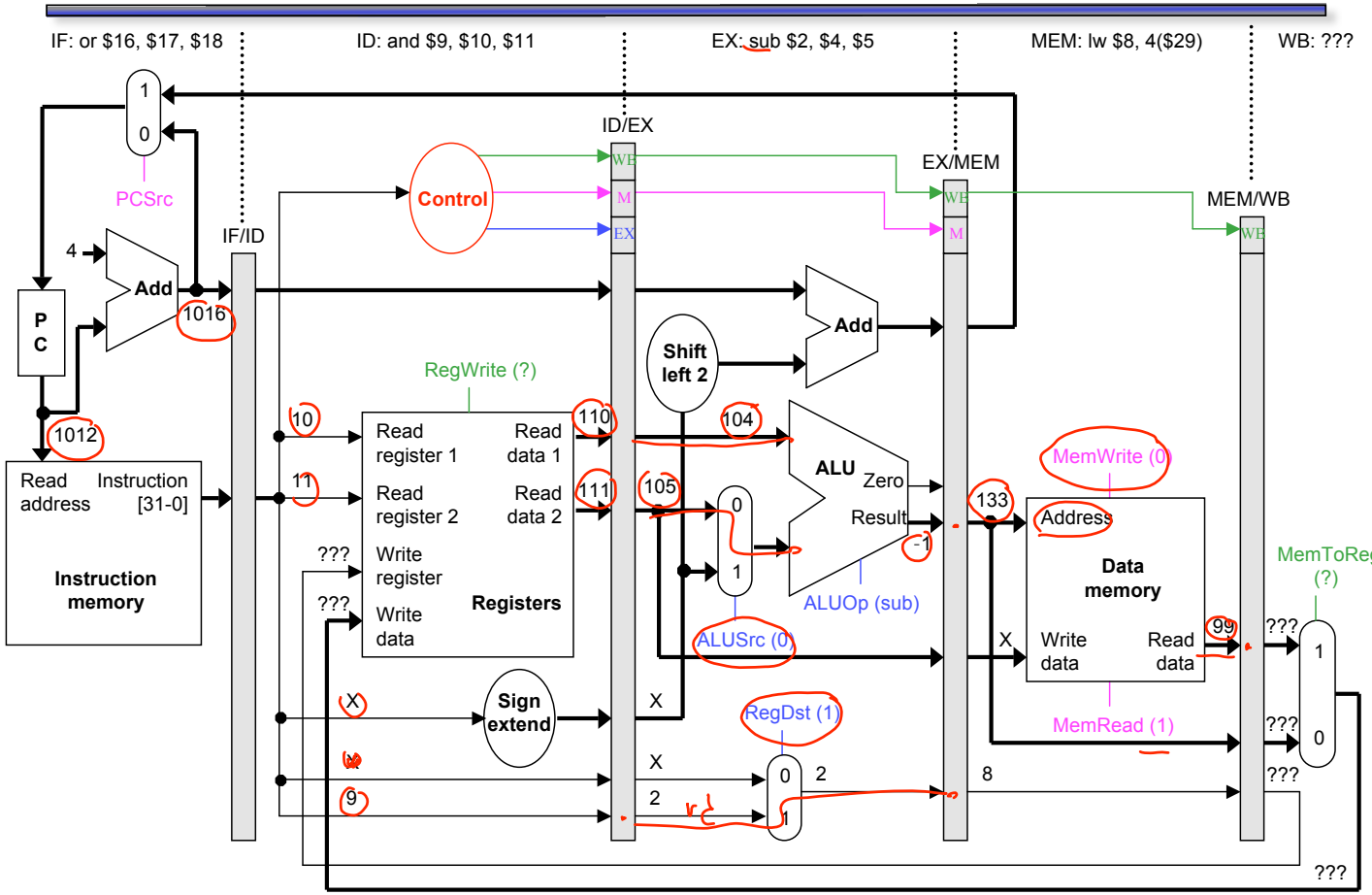
Cycle 2


IF: sub $2, $4, $5
ID: lw $8, 4($29)
EX: ???
MEM: ???
WB: ???
ID/EX
EX/MEM
MEM/WB
IF/ID
PCSrc
Control
WB
M
EX
Add
4
1008
P C
1004
Instruction memory
Read address
Instruction [31-0]
RegWrite (?)
29
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
1129
X
???
Shift left 2
ALU
Zero
Result
ALUOp (???)
ALUSrc (?)
MemWrite (?)
Address
Data memory
Write data
Read data
MemRead (?)
MemToReg (?)
Sign extend
4
8
RegDst (?)

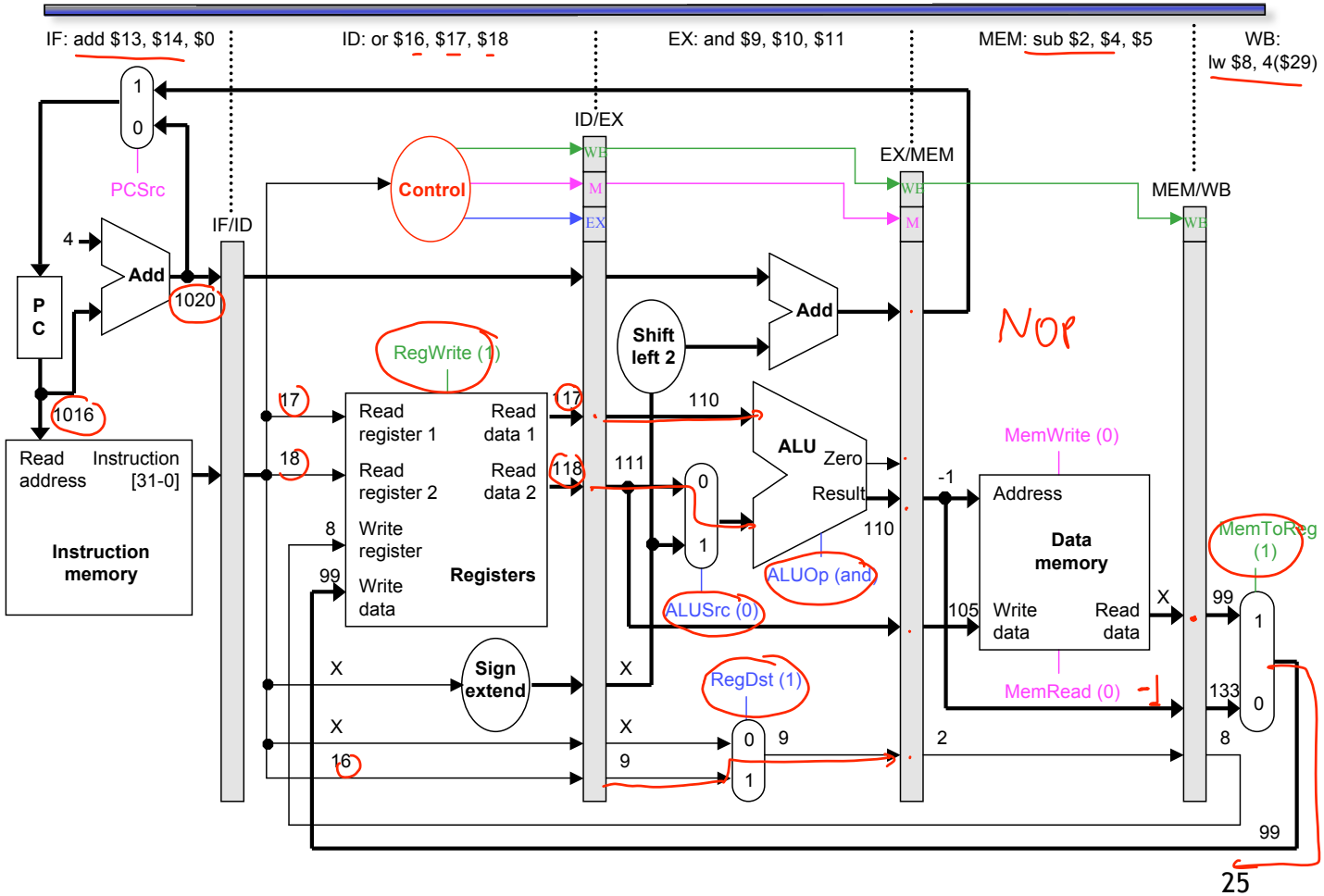
Cycle 3


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

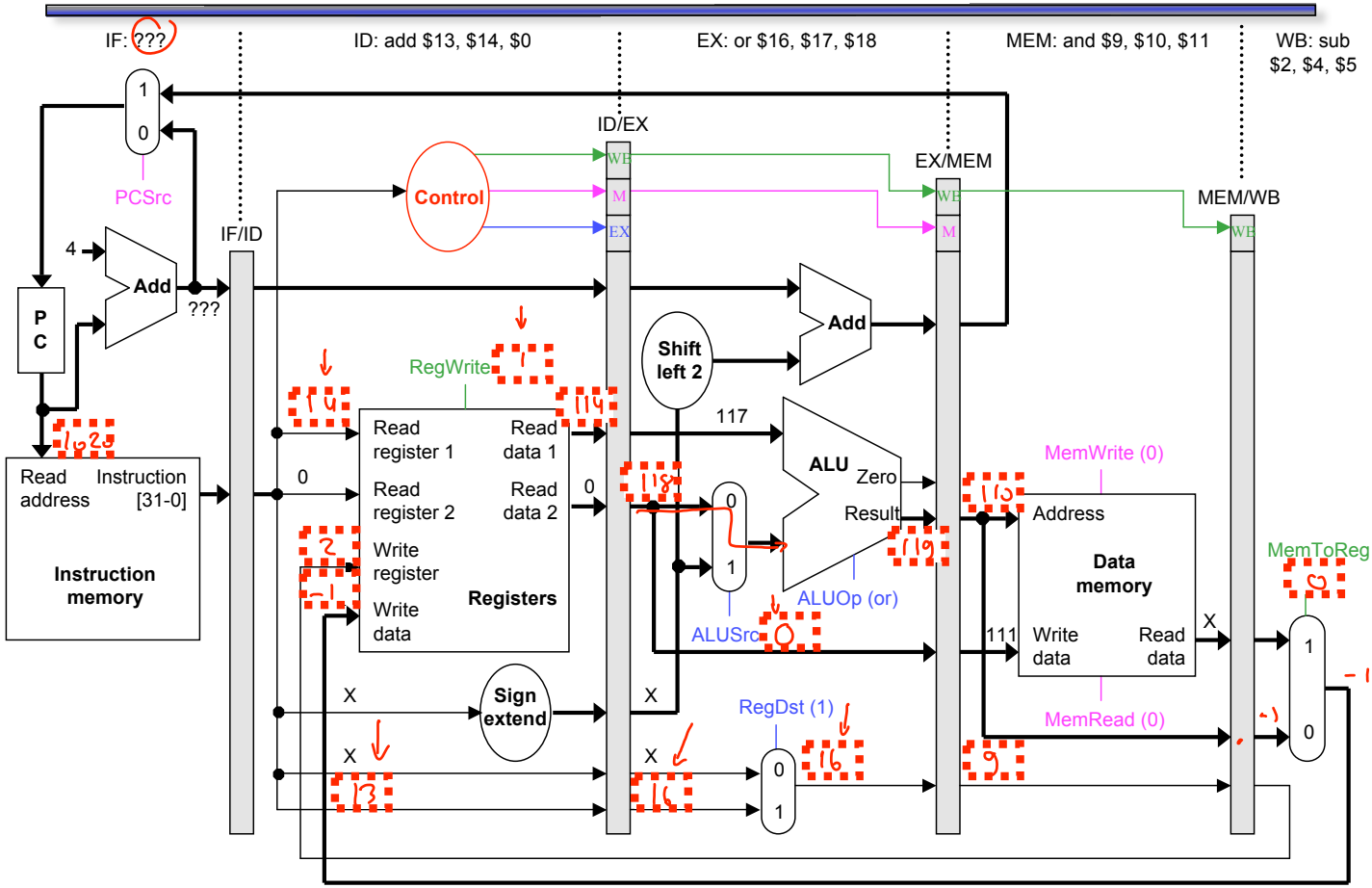
Cycle 4


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

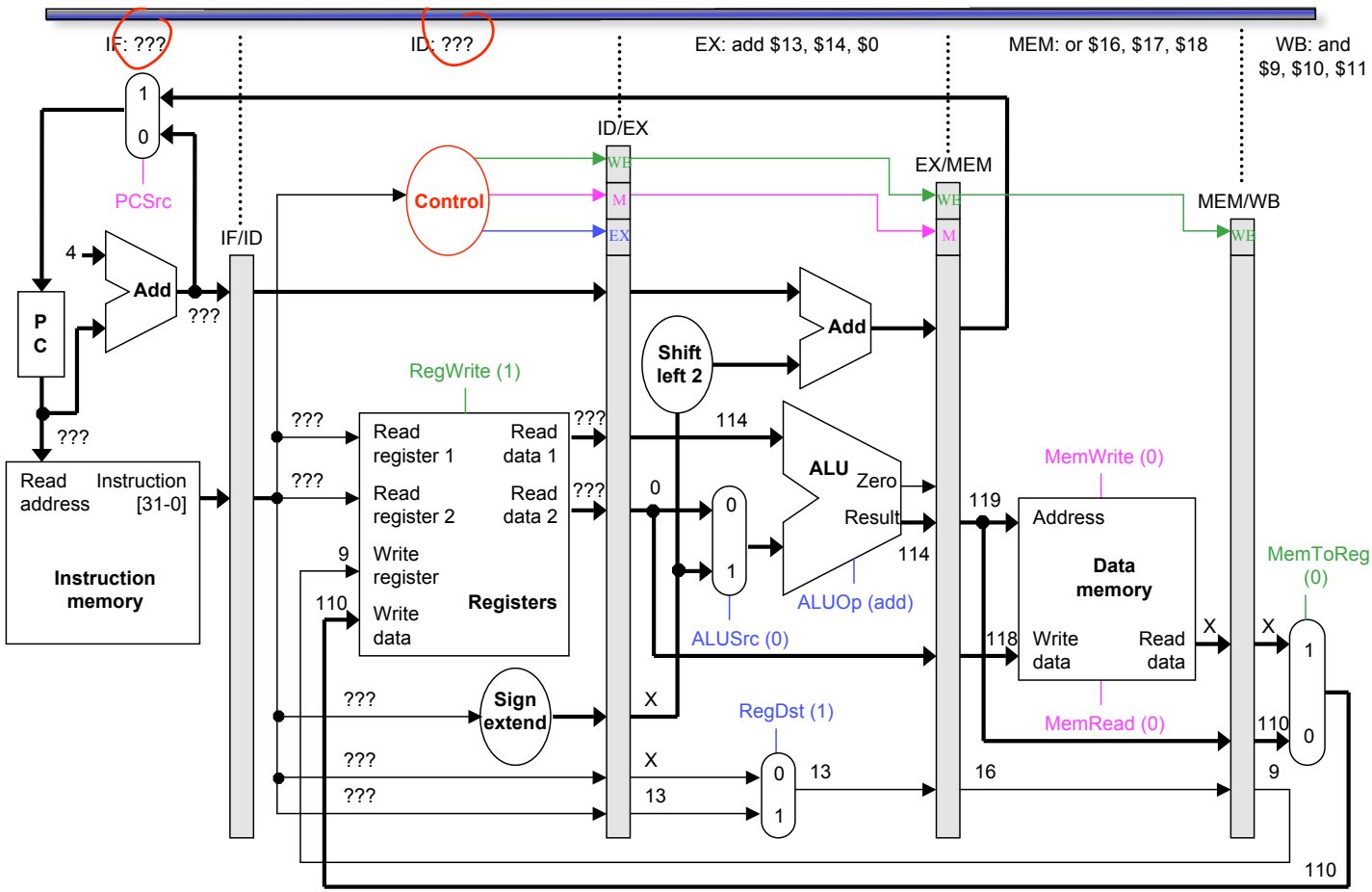
Cycle 5 (full)


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)
ID/EX
EX/MEM
MEM/WB
IF/ID
PCSrc
Control
WB
M
EX
Add
4
1020
P C
1016
Instruction memory
Read address
Instruction [31-0]
RegWrite (1)
Shift left 2
Add
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
17
18
8
99
118
111
110
ALU
Zero
Result
ALUOp (and)
ALUSrc (0)
MemWrite (0)
-1
Address
Data memory
Write data
Read data
105
MemRead (0)
MemToReg (1)
133
Sign extend
X
16
RegDst (1)
9
2
99
25

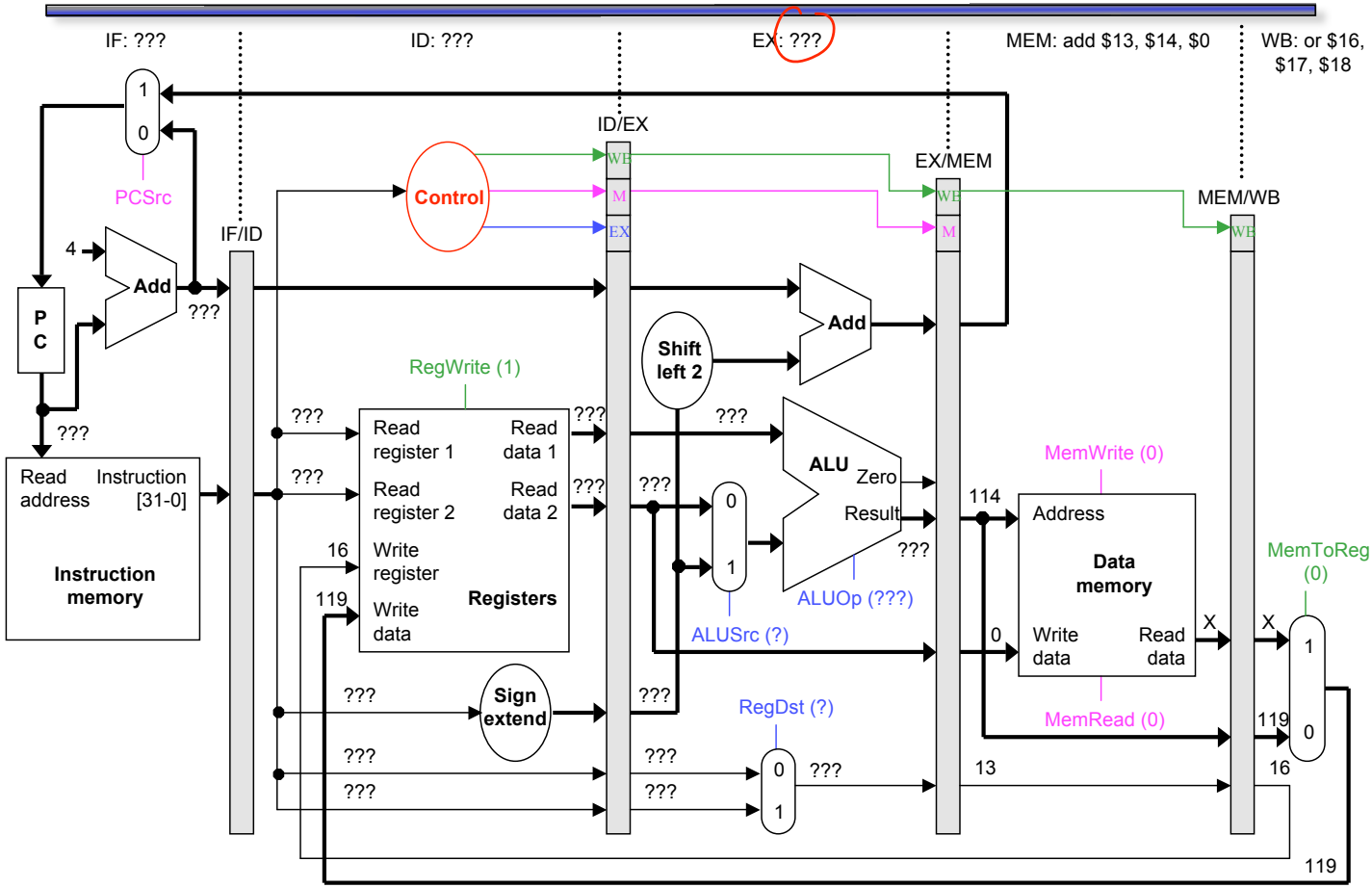
Cycle 6 (emptying)



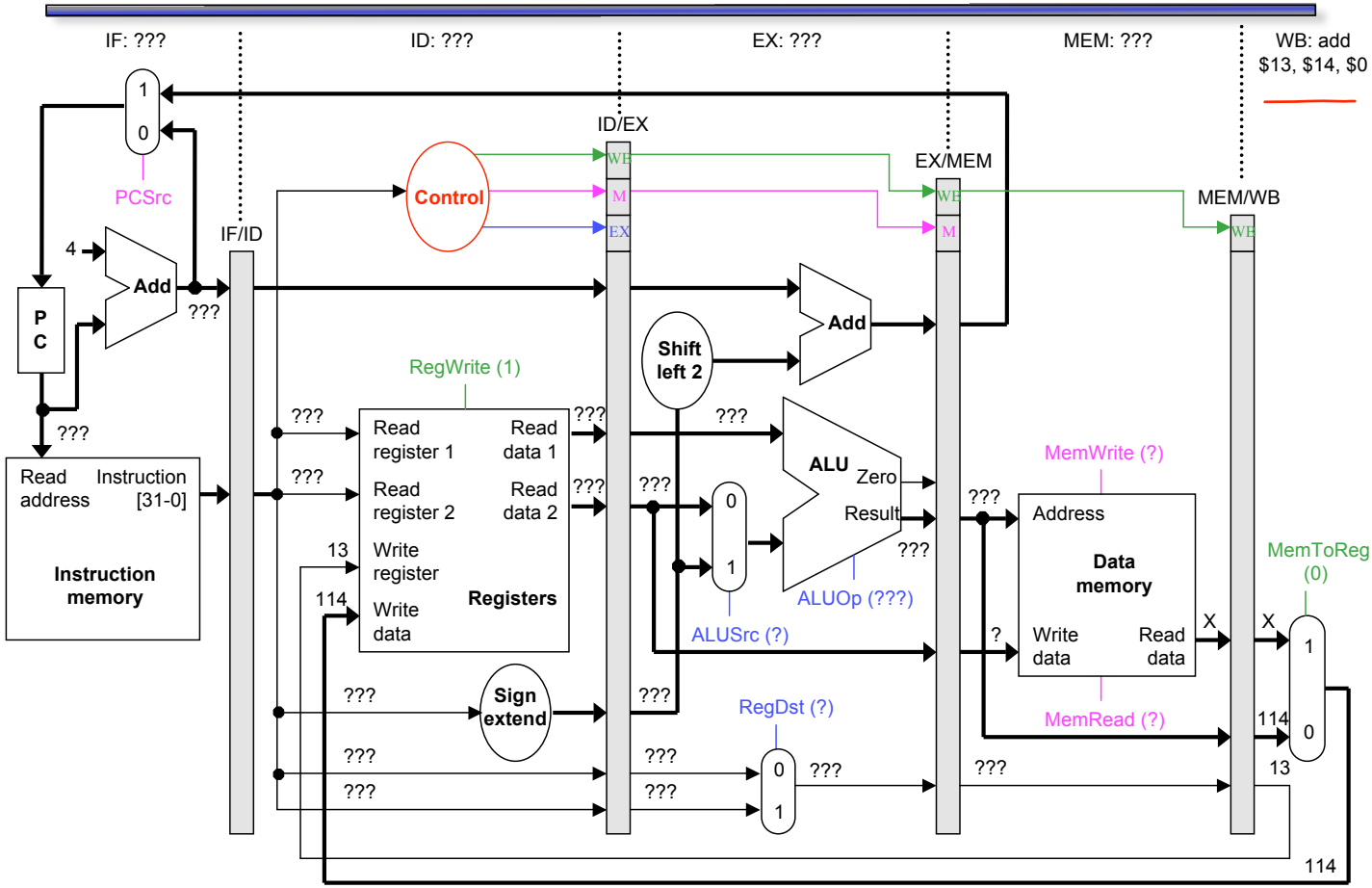
Cycle 7


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

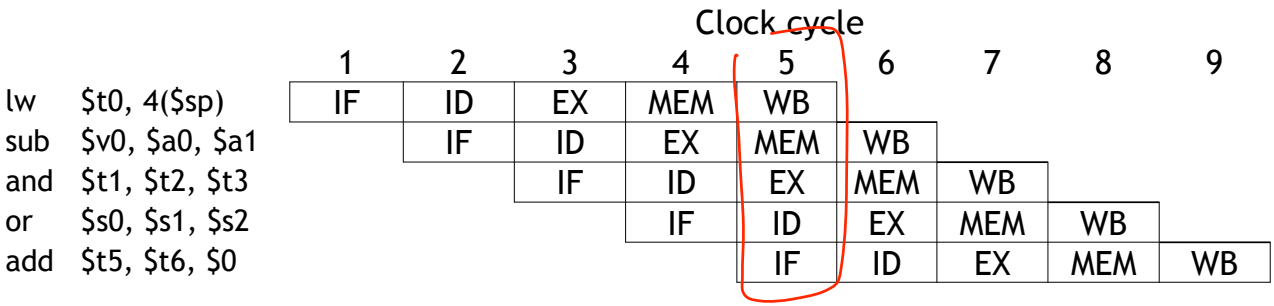
Cycle 8


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

Cycle 9


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

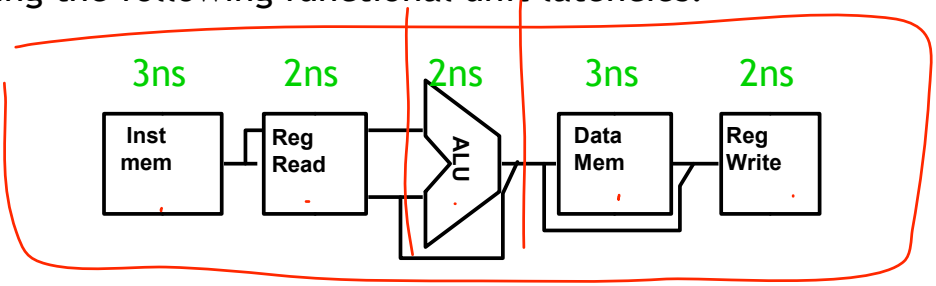
That's a lot of diagrams there



- 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.
- Try to understand this example or the similar one in the book at the end of Section 6.3.

Performance Revisited

- Assuming the following functional unit latencies:



- What is the cycle time of a single-cycle implementation?
  - What is its throughput?

12ns

1 / 12ns

- What is the cycle time of a ideal pipelined implementation?
  - What is its steady-state throughput?

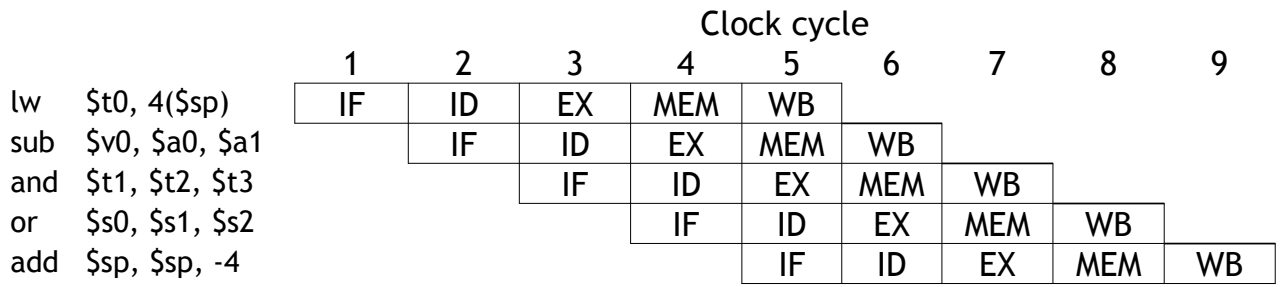
3ns

- How much faster is pipelining?

12ns / 3ns = 4 x

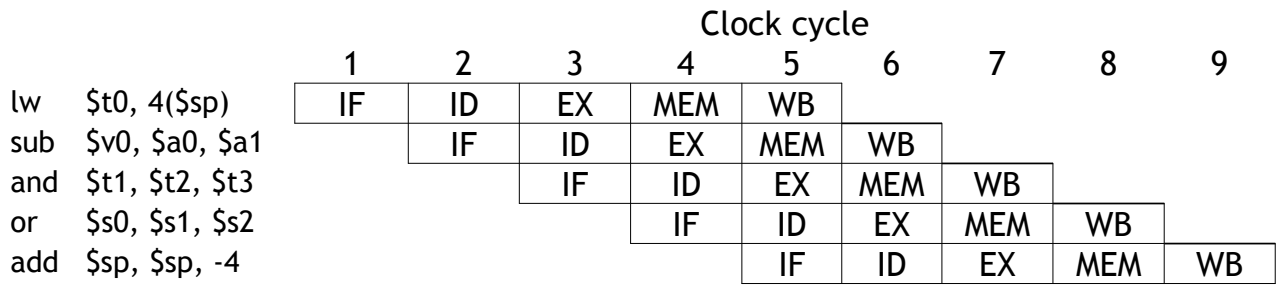
Speedup

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.

Summary

- The pipelined datapath combines ideas from the single and multicycle processors that we saw earlier.
  - It 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.

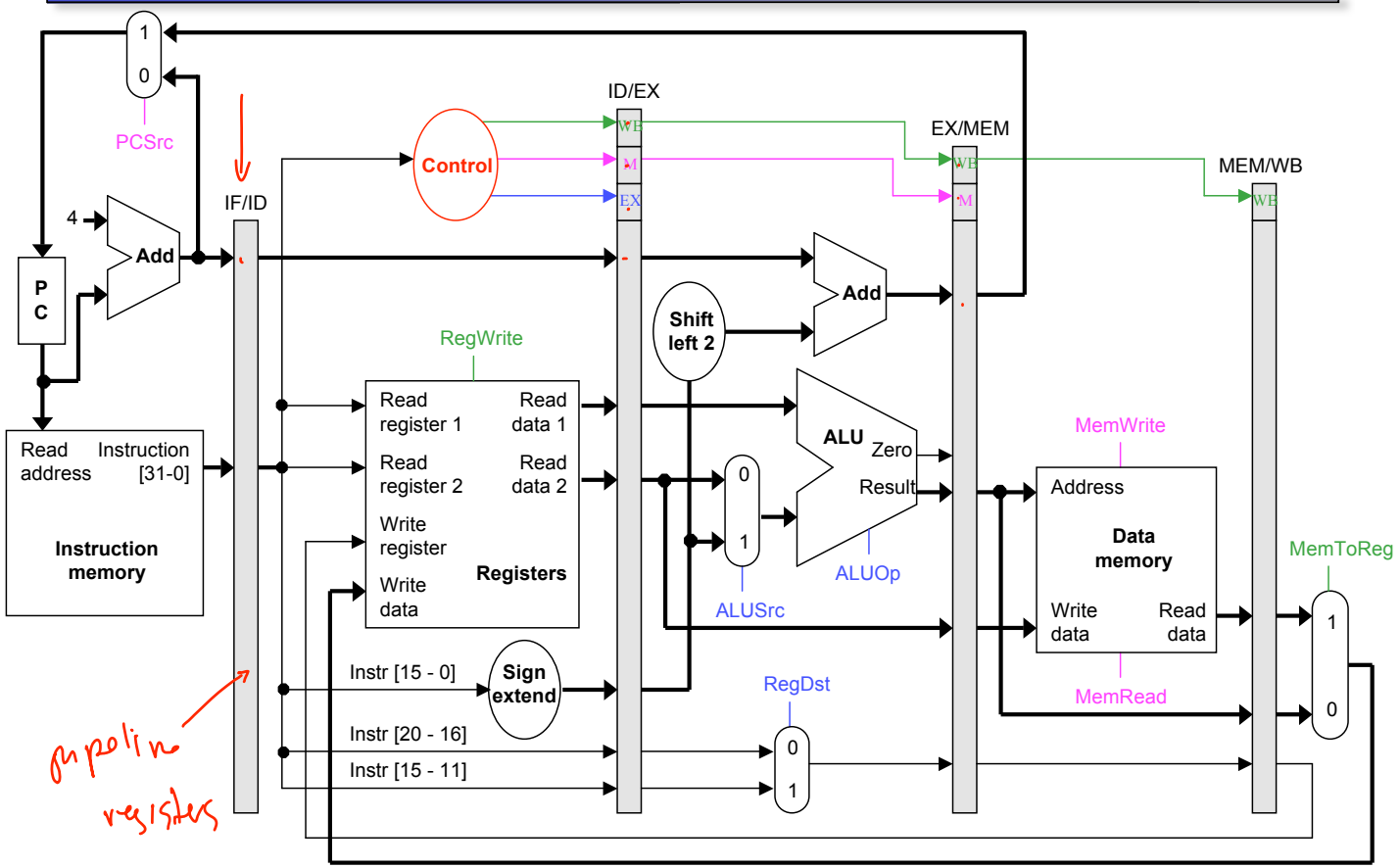
- Next lecture, we'll start talking about Hazards.

Forwarding

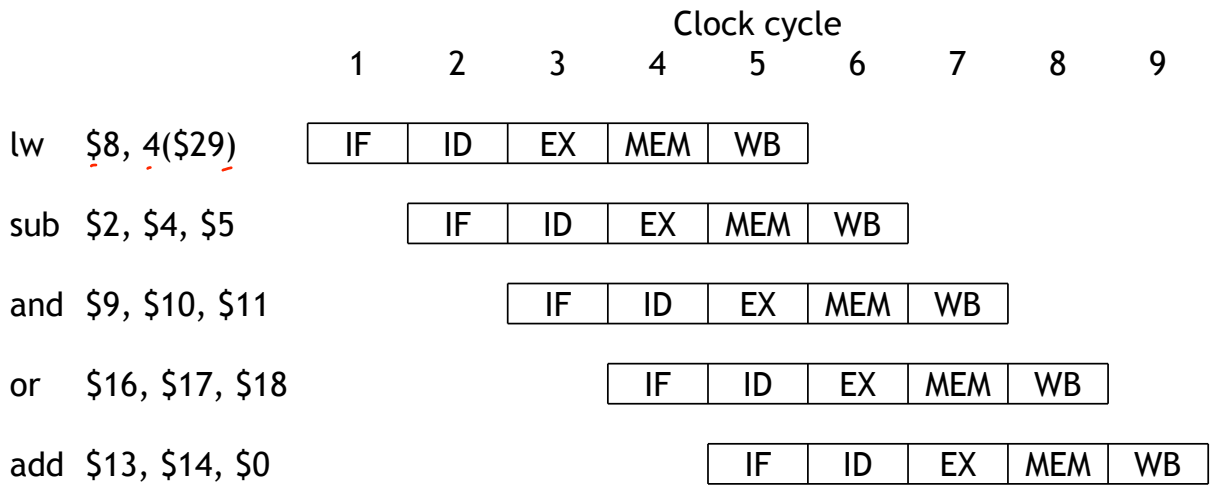
- Now, we'll introduce some problems that data hazards can cause for our pipelined processor, and show how to handle them with forwarding.

Pick-up hand-out !

The pipelined datapath



Pipeline diagram review



- This diagram shows the execution of an ideal code fragment.
  - Each instruction needs a total of five cycles for execution.
  - One instruction begins on every clock cycle for the first five cycles.
  - One instruction completes on each cycle from that time on.

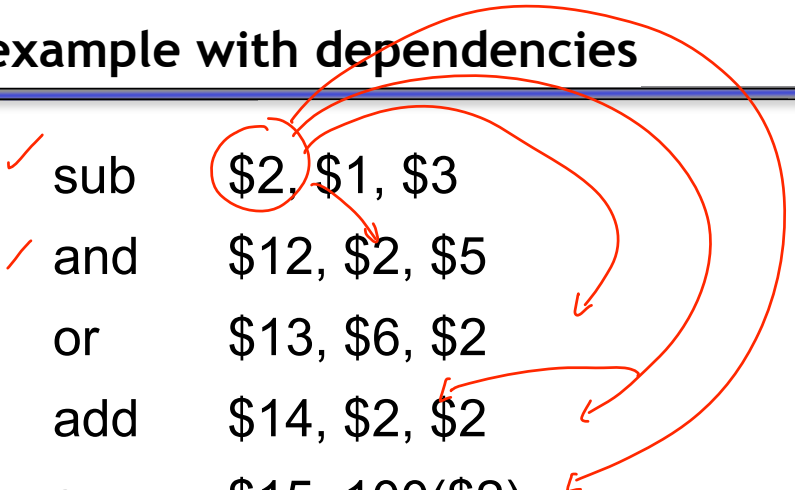
Our examples are too simple

- 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



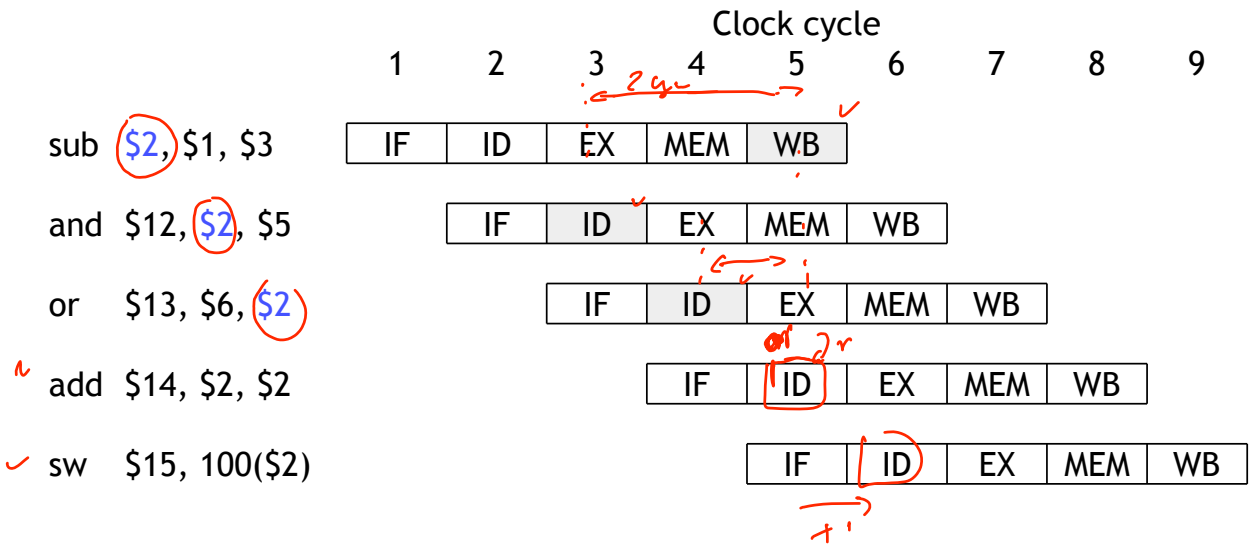
```
sub   $2, $1, $3
and   $12, $2, $5
or    $13, $6, $2
add   $14, $2, $2
sw    $15, 100($2)
```

An example with dependencies

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

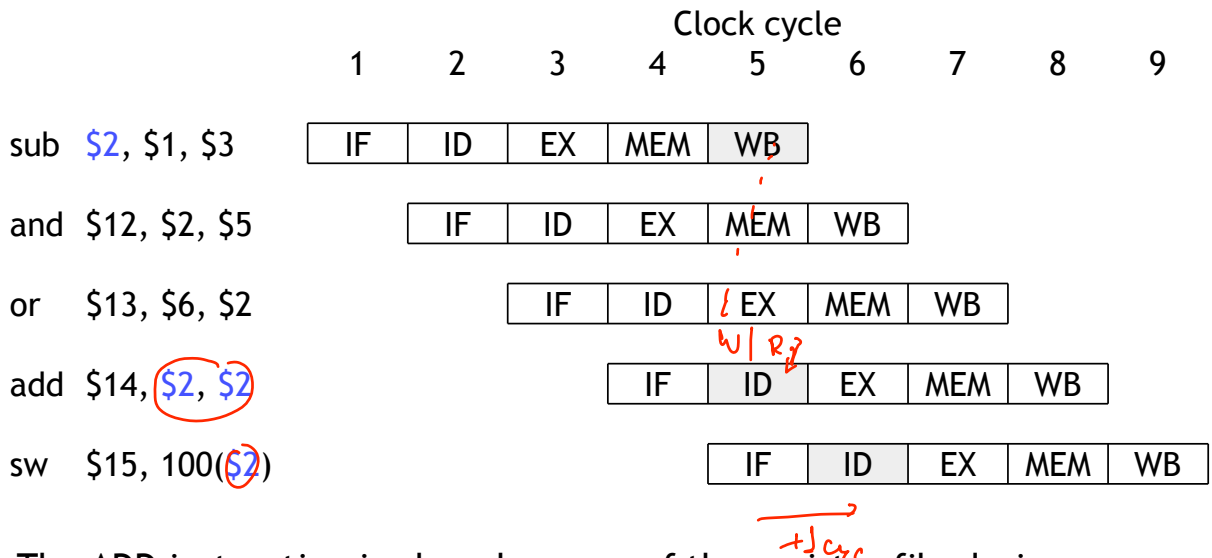
- There are several dependencies in this new code fragment.
  - The first instruction, SUB, stores a value into $2.
  - That register is used as a source in the rest of the instructions.
- This is not a problem for the single-cycle and multicycle datapaths.
  - Each instruction is executed completely 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



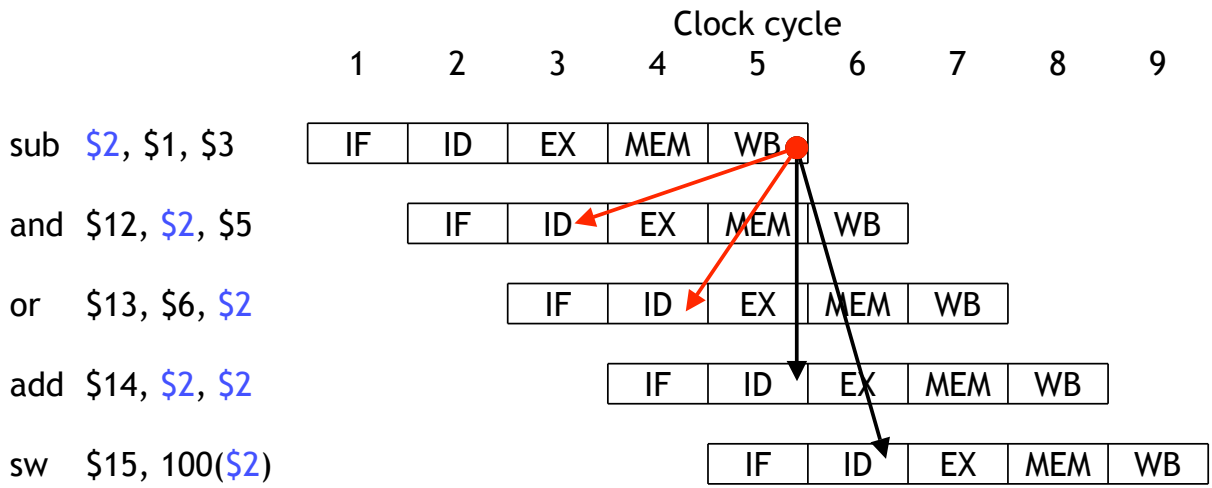
- The SUB instruction does not write to register $2 until clock cycle 5. This causes two data hazards in our current pipelined datapath.
  - The AND reads register $2 in cycle 3. Since SUB hasn't modified the register yet, this will be the *old* value of $2, not the new one.
  - Similarly, the OR instruction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

Things that are okay



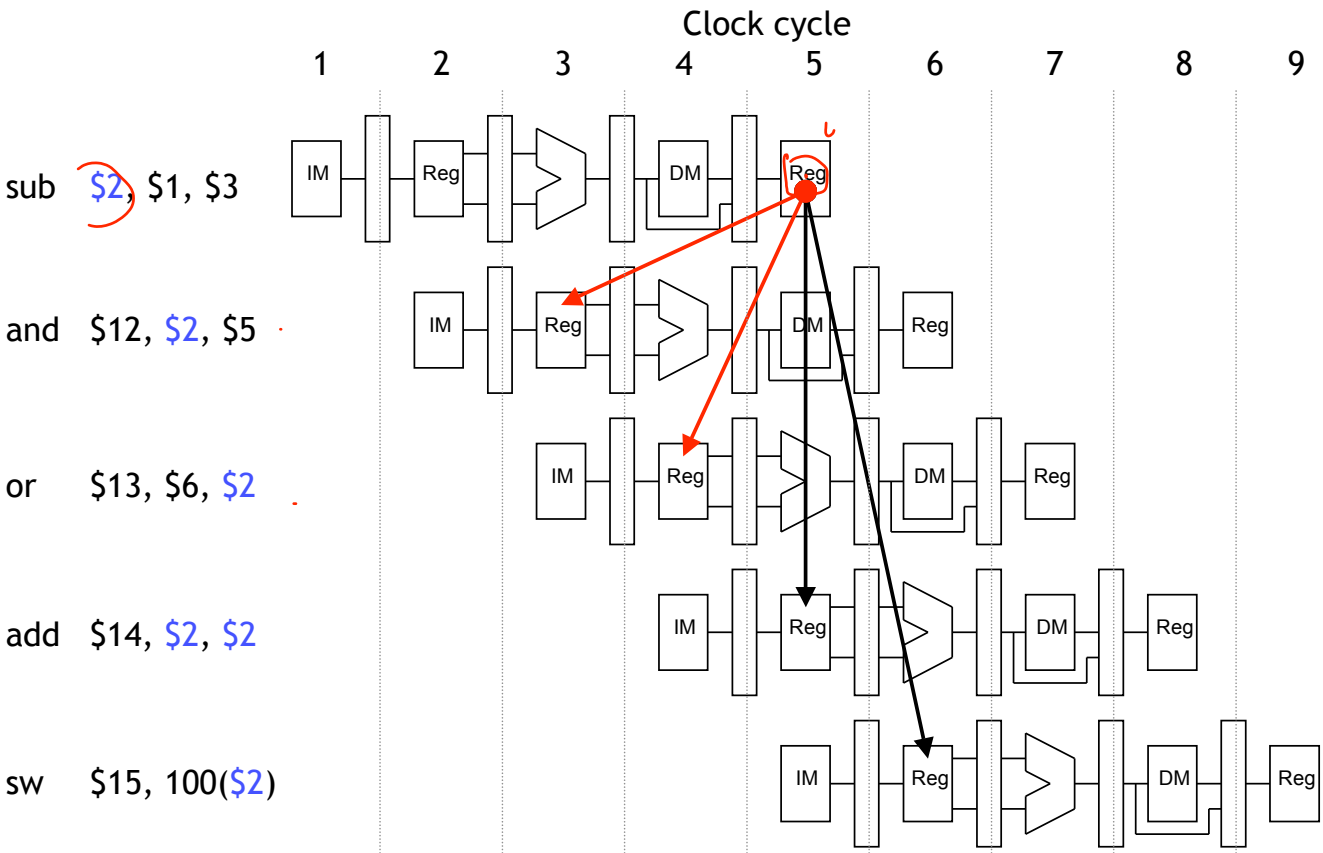
- The ADD instruction is okay, because of the register file design.
  - Registers are written at the beginning of a clock cycle.
  - The new value will be available by the end of that cycle.
- The SW is no problem at all, since it reads $2 after the SUB finishes.

Dependency arrows



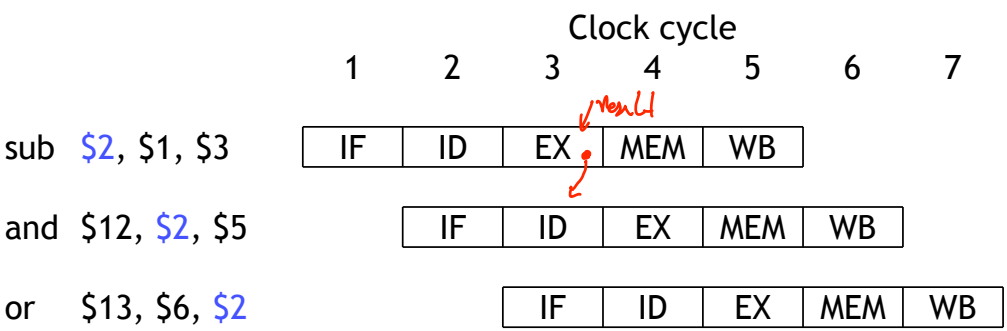
- Arrows indicate the flow of data between instructions.
  - The tails of the arrows show when register $2 is written.
  - The heads of the arrows show when $2 is read.
- Any arrow that points backwards in time represents a data hazard in our basic pipelined datapath. Here, hazards exist between instructions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

A fancier pipeline diagram



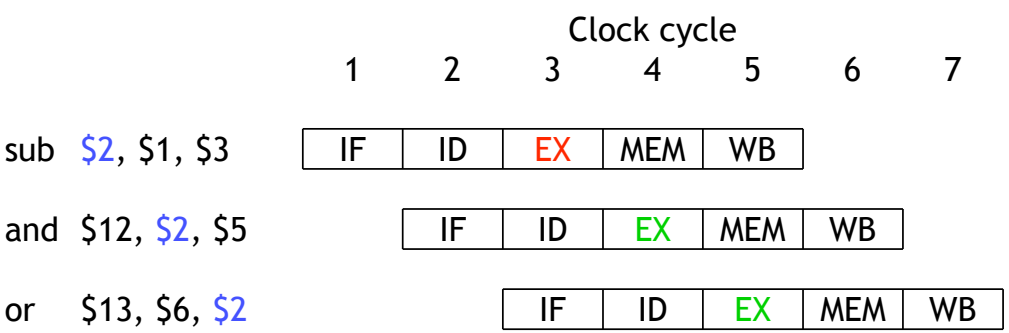
A more detailed look at the pipeline

- We have to eliminate the hazards, so the AND and OR instructions in our example will use the correct value for register $2.
- When is the data is actually produced and consumed?
- What can we do?



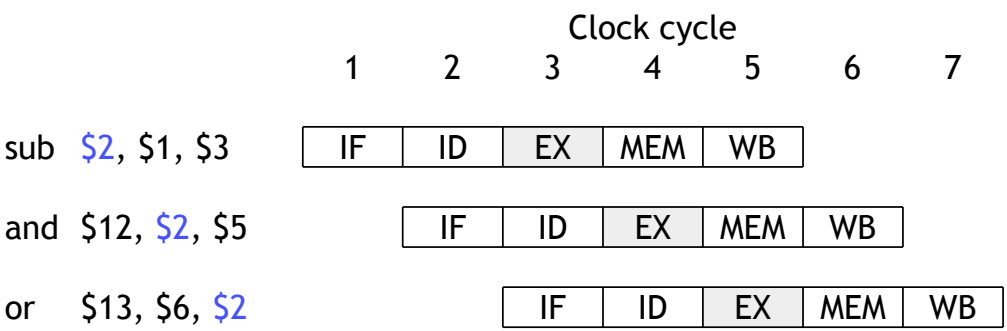
A more detailed look at the pipeline

- 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.



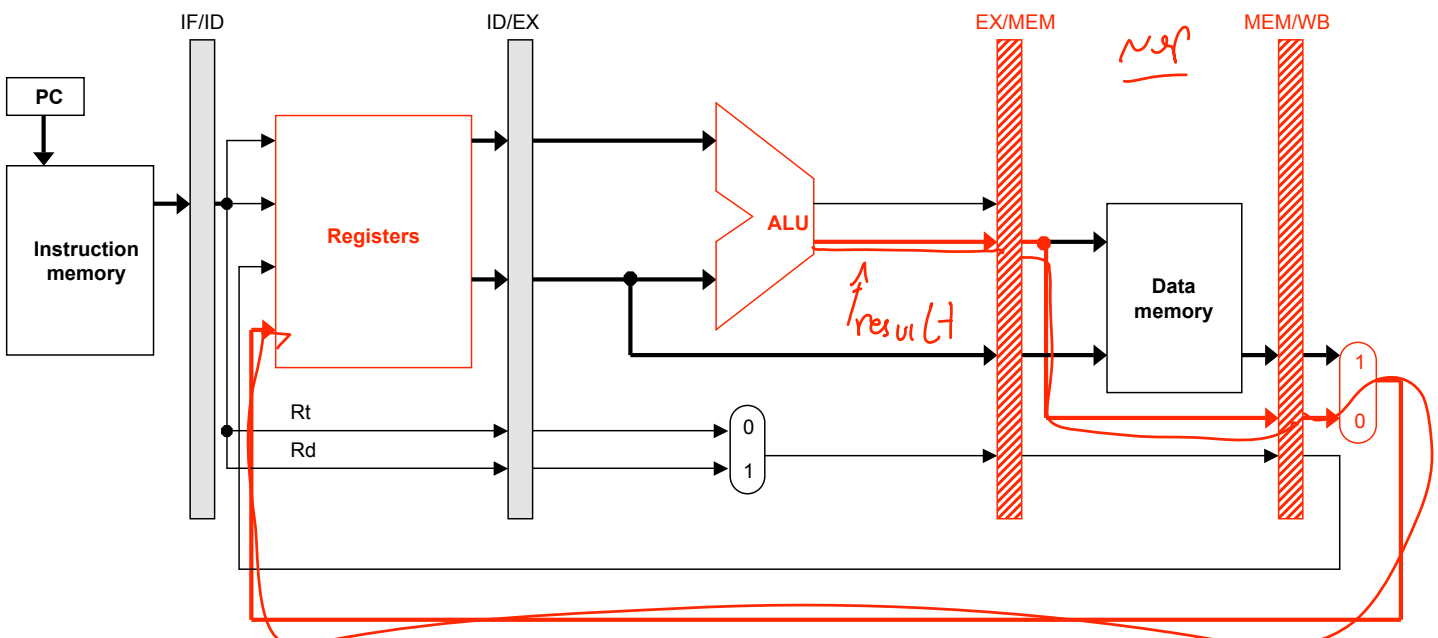
Bypassing the register file

- 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.



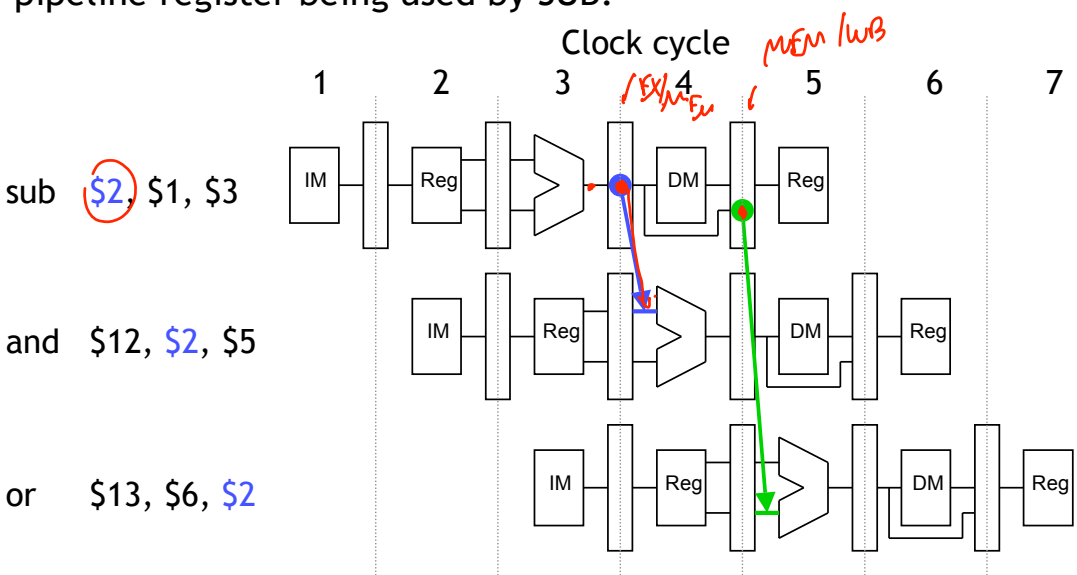
Where to find the ALU result

- 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.



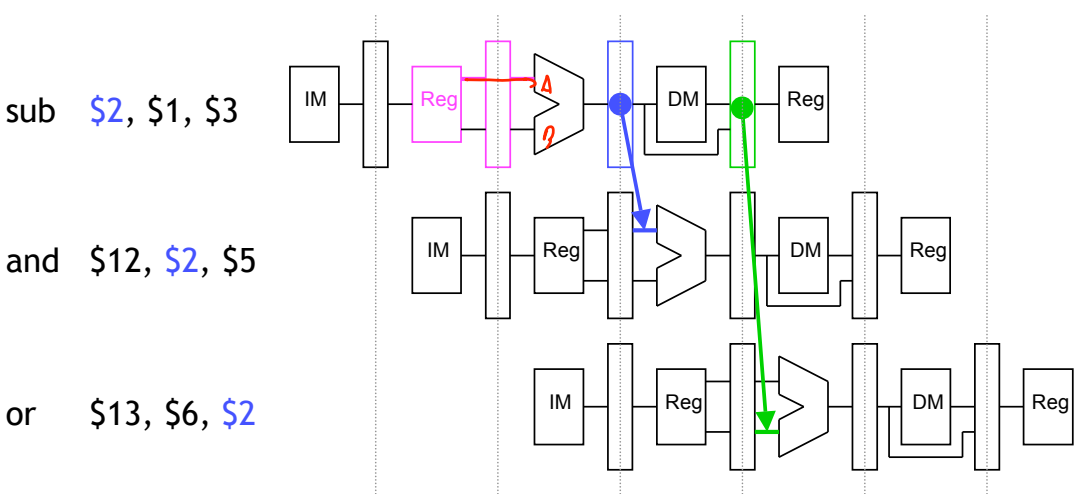
Forwarding

- 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.

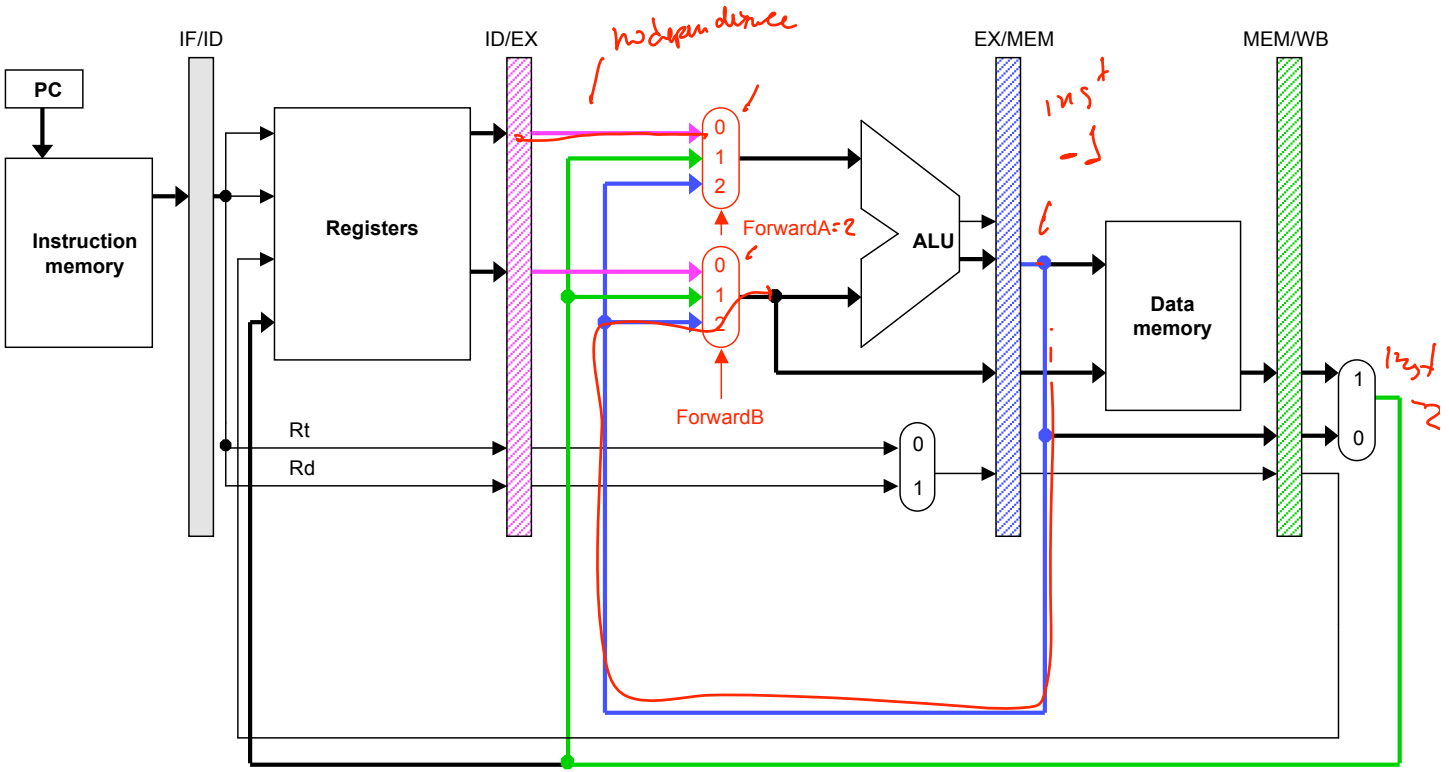


Outline of forwarding hardware

- 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.

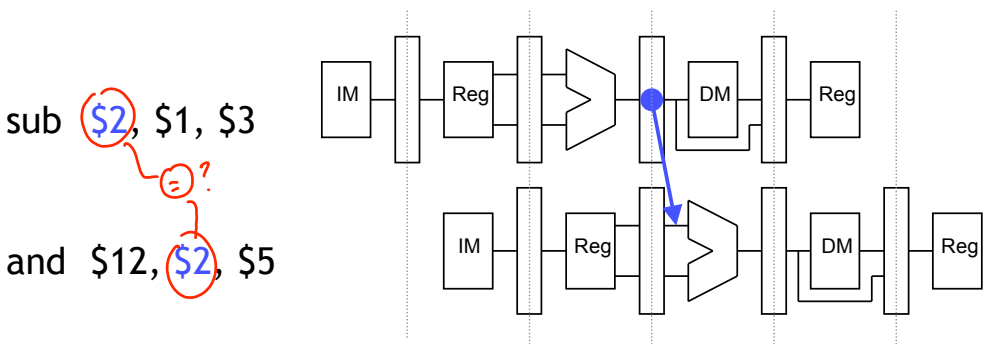


Simplified datapath with forwarding muxes



Detecting EX/MEM data hazards

- So how can the hardware determine if a hazard exists?

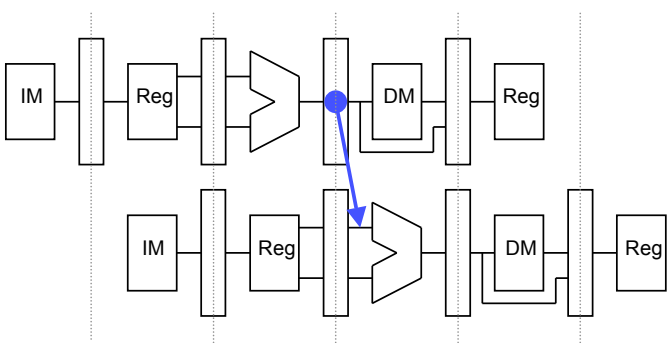


Detecting EX/MEM data hazards

- 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.RegisterRt refers to the rt field stored in the ID/EX pipeline.

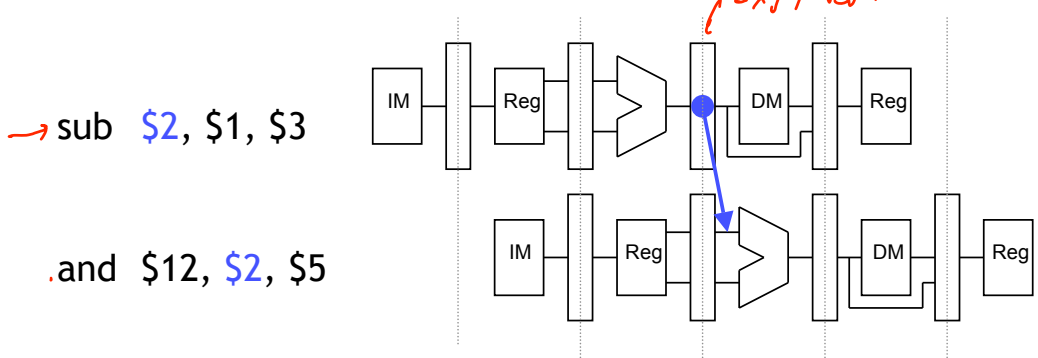
EX/MEM data hazard equations

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

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

- The second ALU source is similar.

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

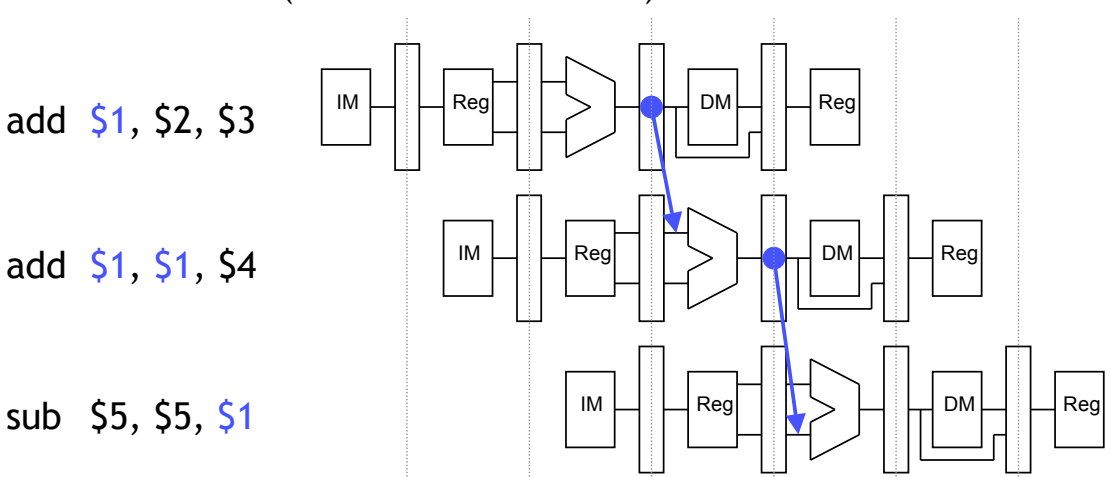


Detecting MEM/WB data hazards

- 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.



MEM/WB hazard equations

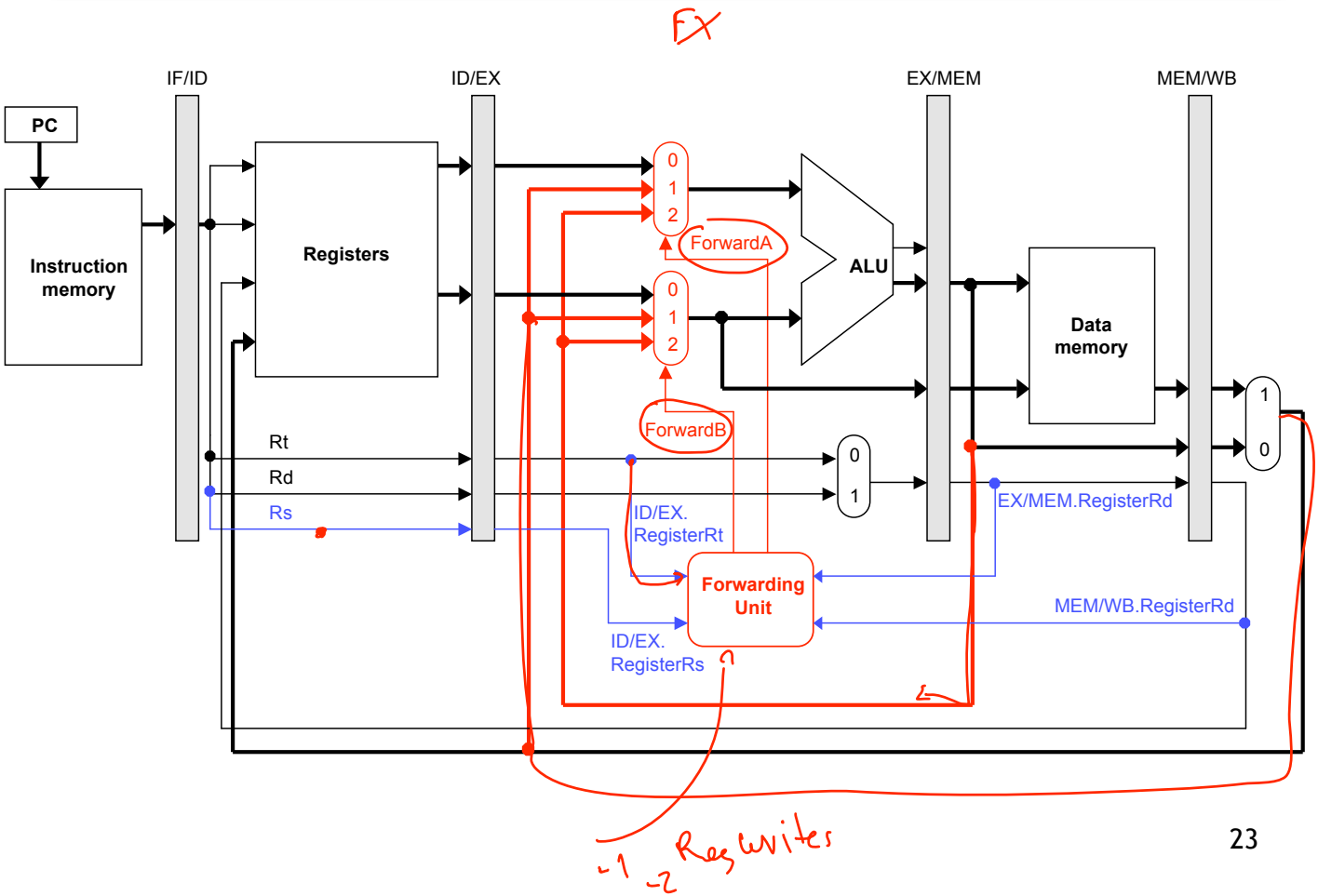
- Here is an equation for detecting and handling MEM/WB hazards for the first ALU source.

  if (MEM/WB.RegWrite = 1
  and MEM/WB.RegisterRd = ID/EX.RegisterRs
  and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRs or EX/MEM.RegWrite = 0)
  then ForwardA = 1

- The second ALU operand is handled similarly.

  if (MEM/WB.RegWrite = 1
  and MEM/WB.RegisterRd = ID/EX.RegisterRt
  and (EX/MEM.RegisterRd ≠ ID/EX.RegisterRt or EX/MEM.RegWrite = 0)
  then ForwardB = 1

Simplified datapath with forwarding



The forwarding unit

- The forwarding unit has several control signals as inputs.

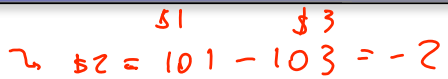
ID/EX.RegisterRs	EX/MEM.RegisterRd	MEM/WB.RegisterRd
ID/EX.RegisterRt	EX/MEM.RegWrite	MEM/WB.RegWrite

(The two RegWrite signals are not shown in the diagram, but they come from the control unit.)

- The fowarding 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

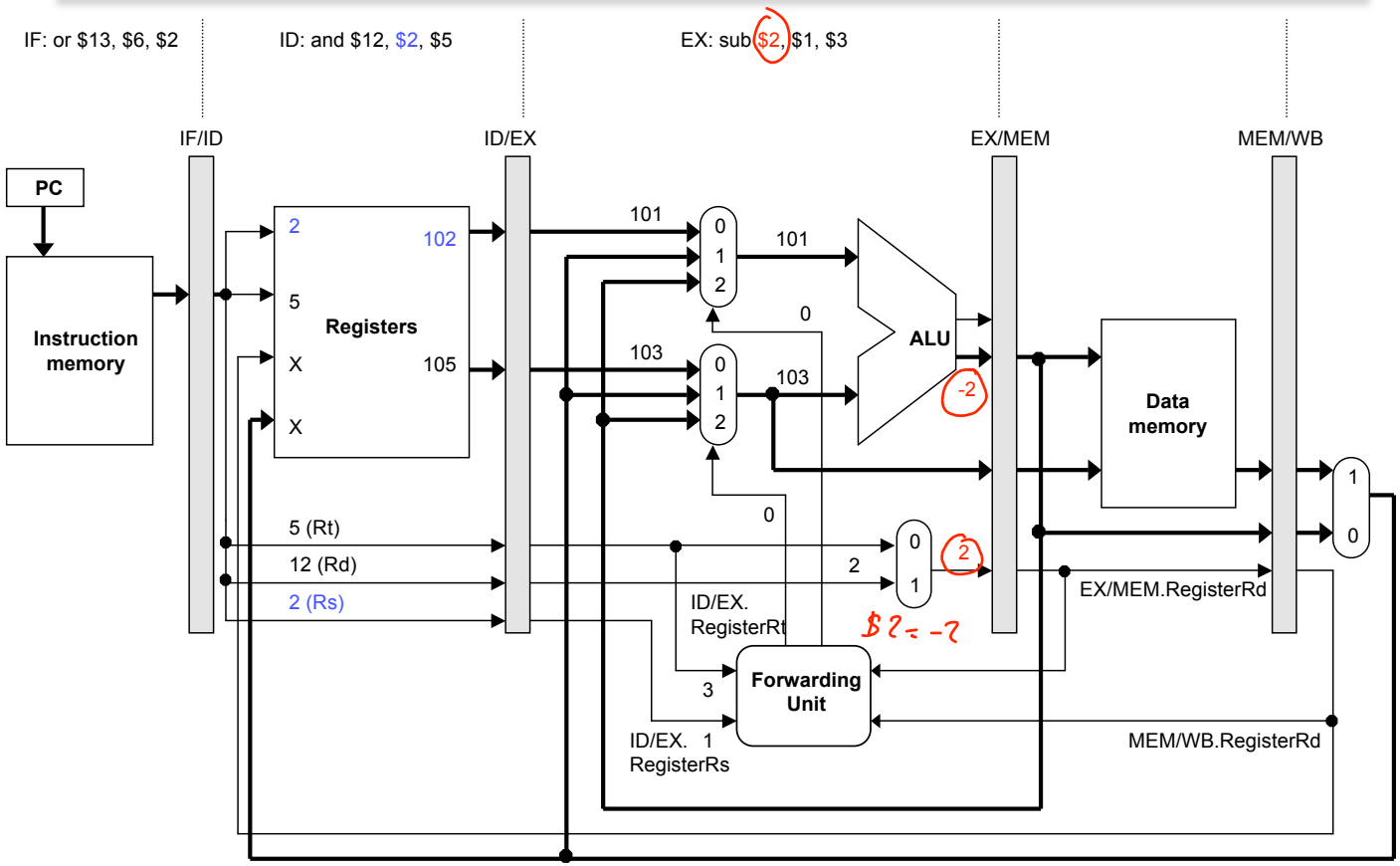
```
sub   $2, $1, $3
and   $12, $2, $5
or    $13, $6, $2
add   $14, $2, $2
sw    $15, 100($2)
```



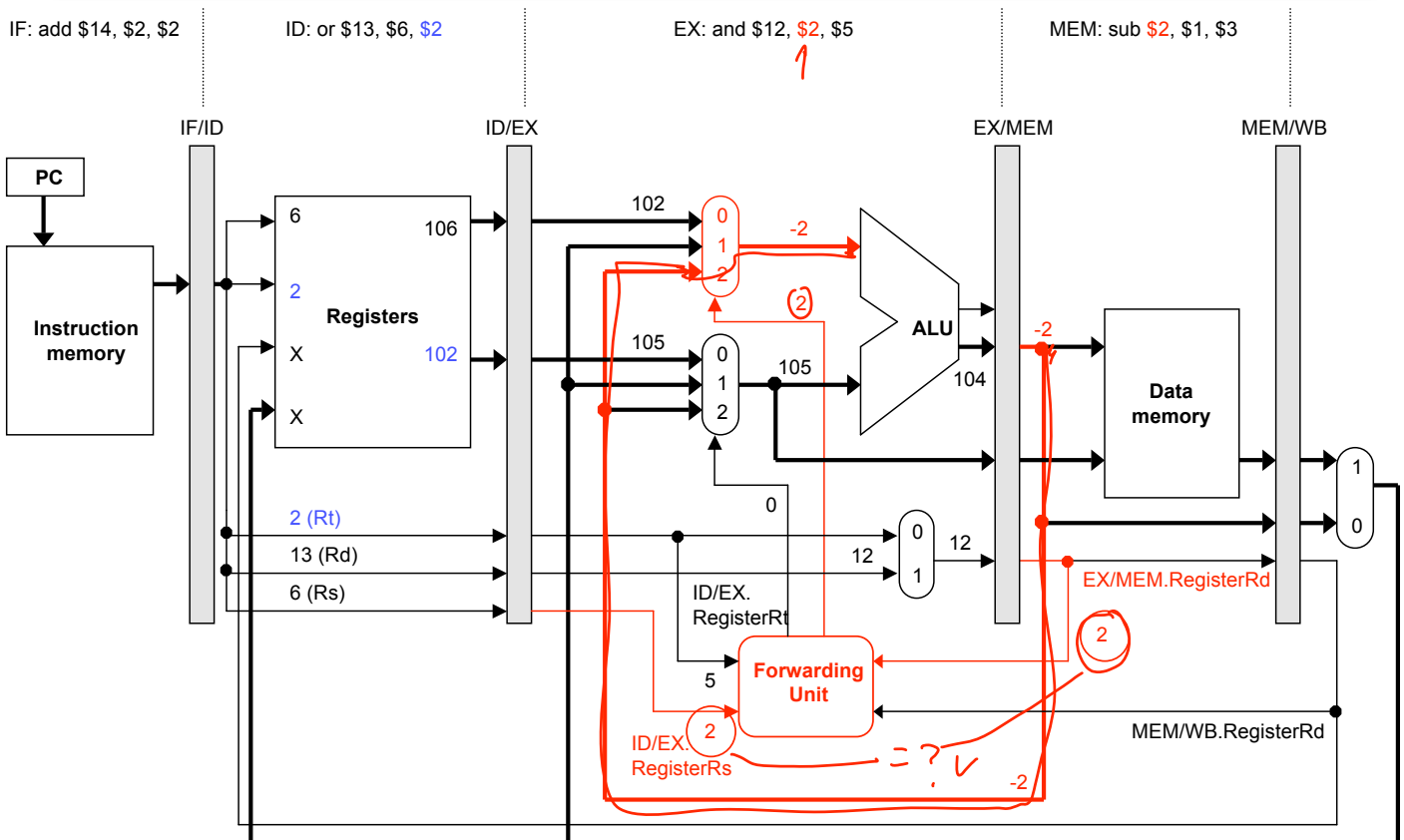
- Assume again each register initially contains its number plus 100.
  - After the first instruction, $2 should contain -2 (101 - 103).
  - The other instructions should all use -2 as one of their operands.

- 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.

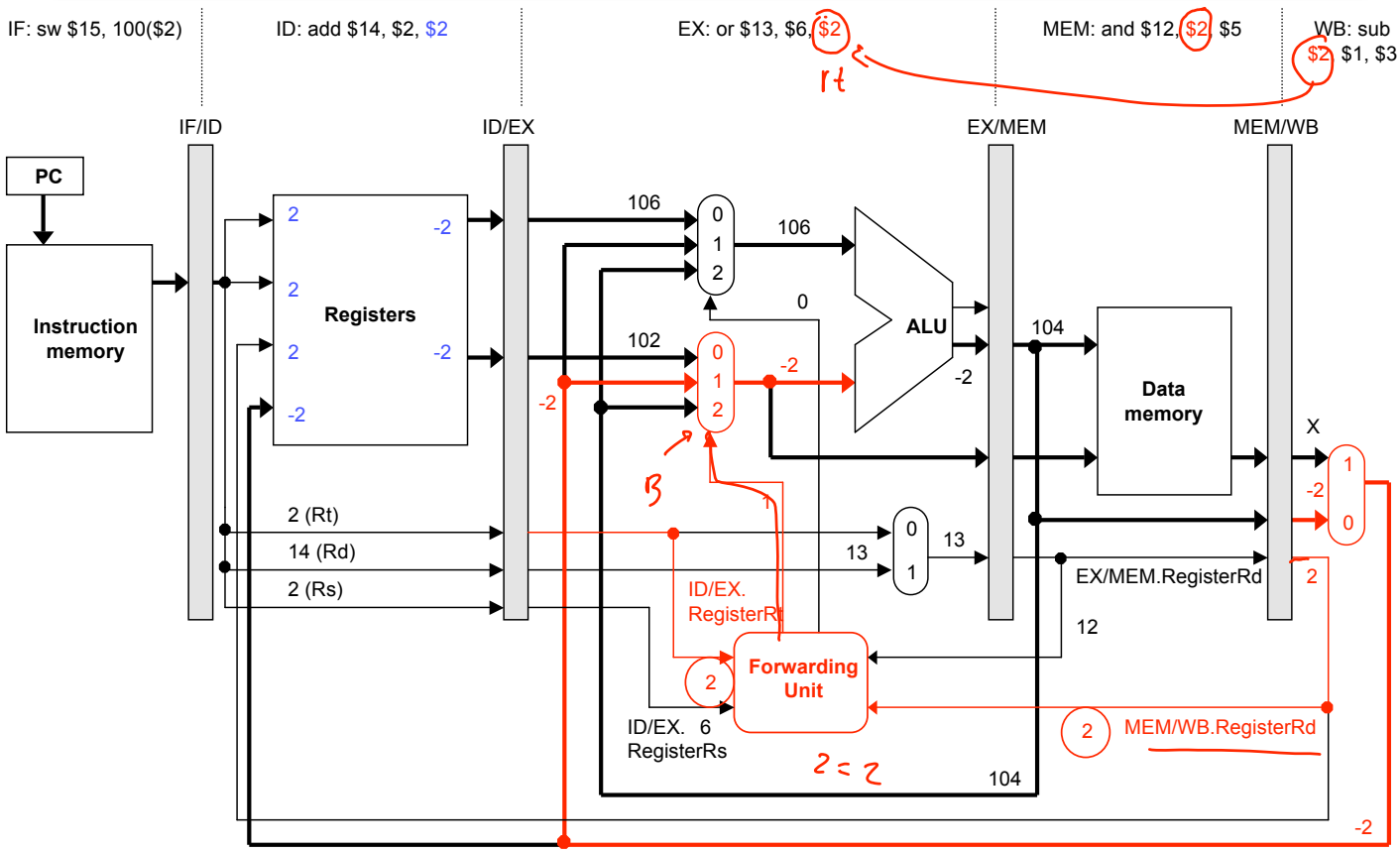
Clock cycle 3



Clock cycle 4: forwarding $2 from EX/MEM



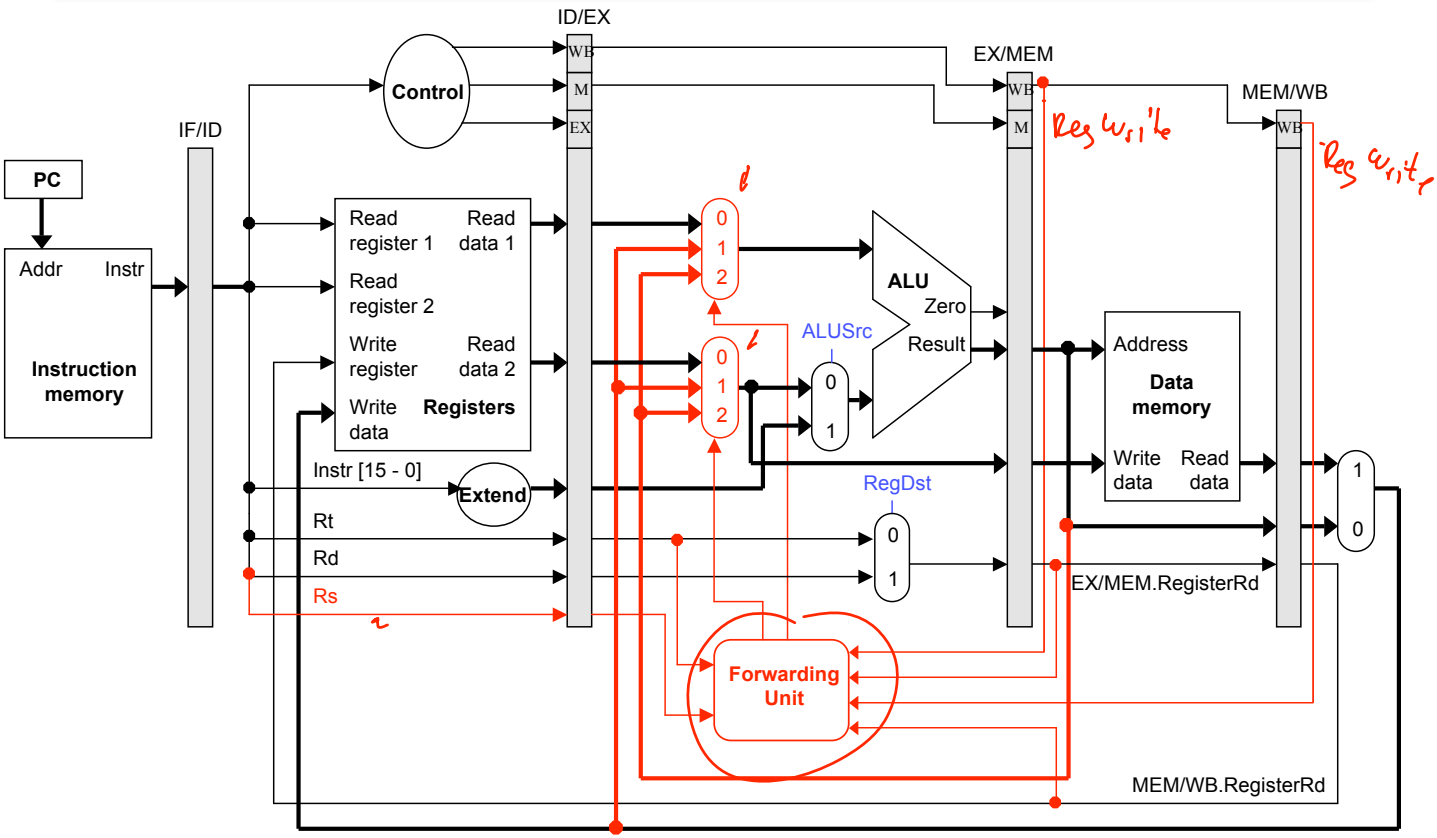
Clock cycle 5: forwarding $2 from MEM/WB



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.

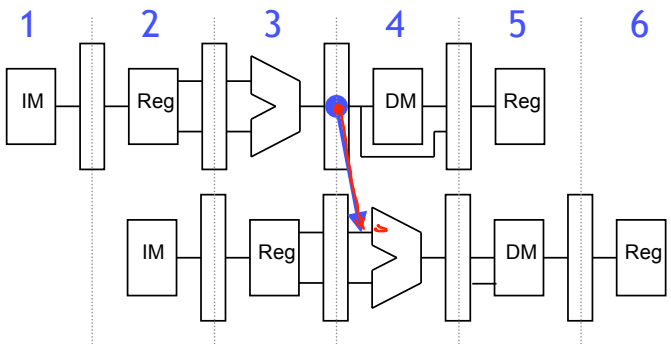
Complete pipelined datapath…so far



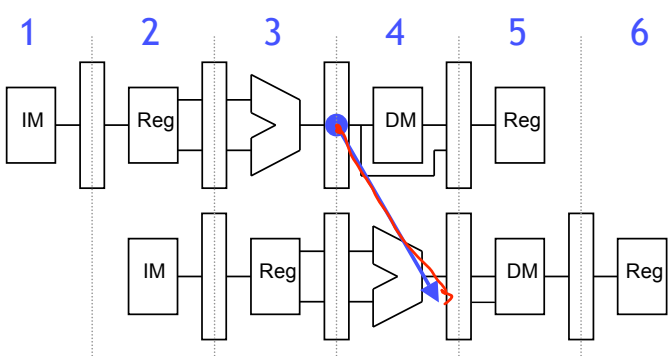
What about stores?

- Two “easy” cases:

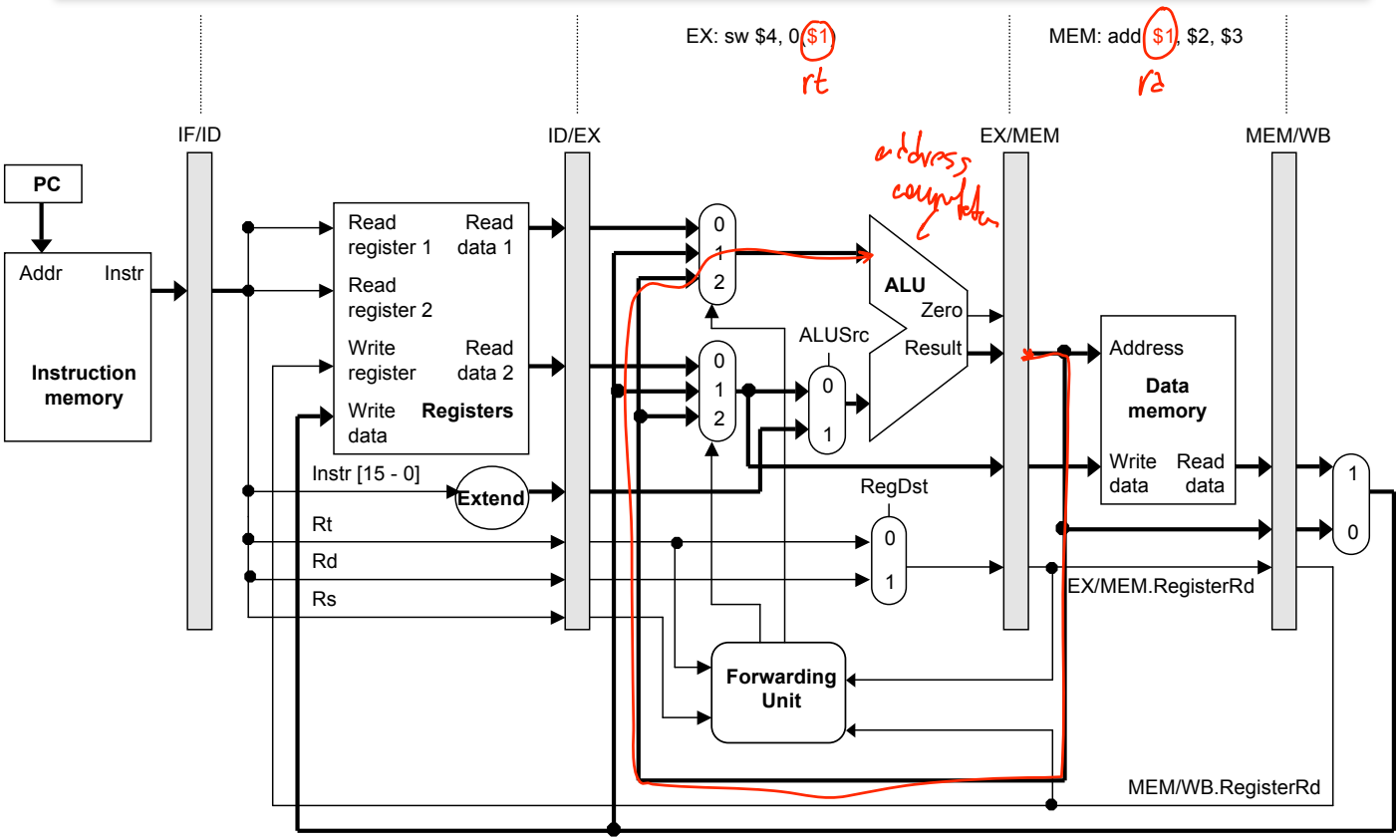
```
add $1, $2, $3
sw  $4, 0($1)
```



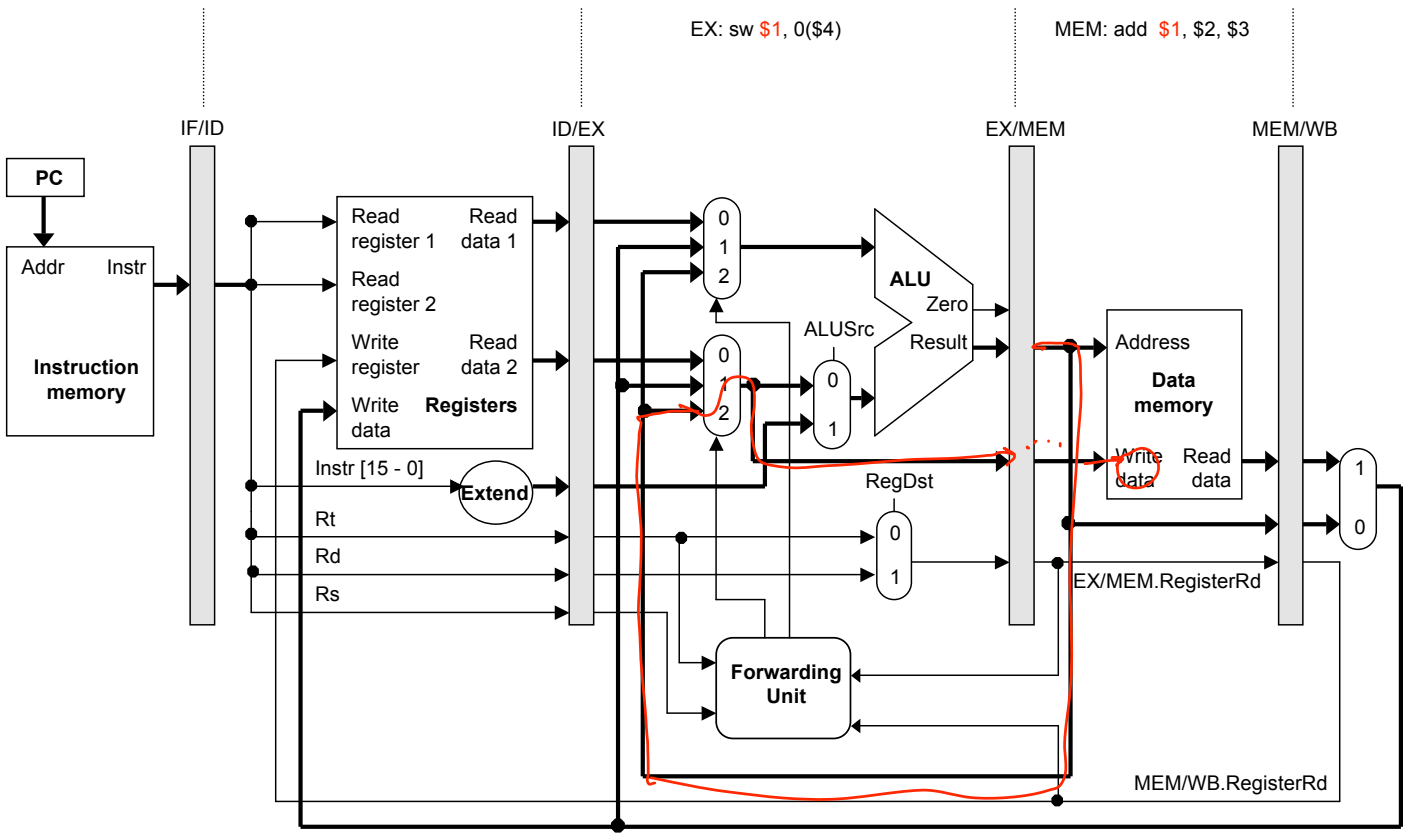

```
add $1, $2, $3
sw  $1, 0($4)
```



Store Bypassing: Version 1

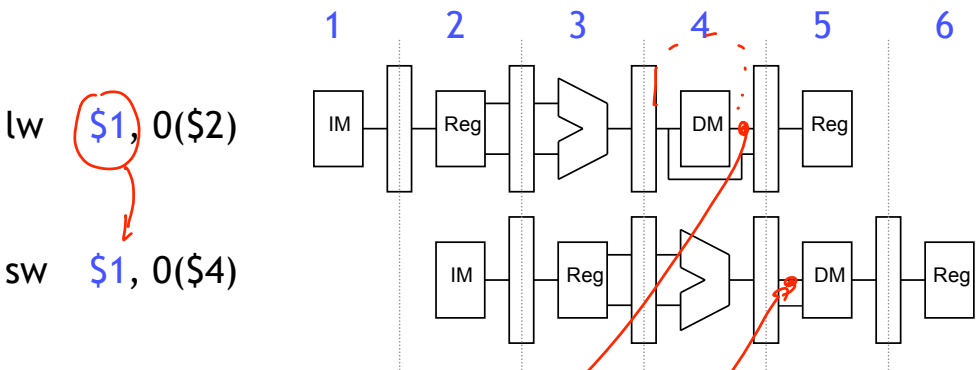


Store Bypassing: Version 2



What about stores?

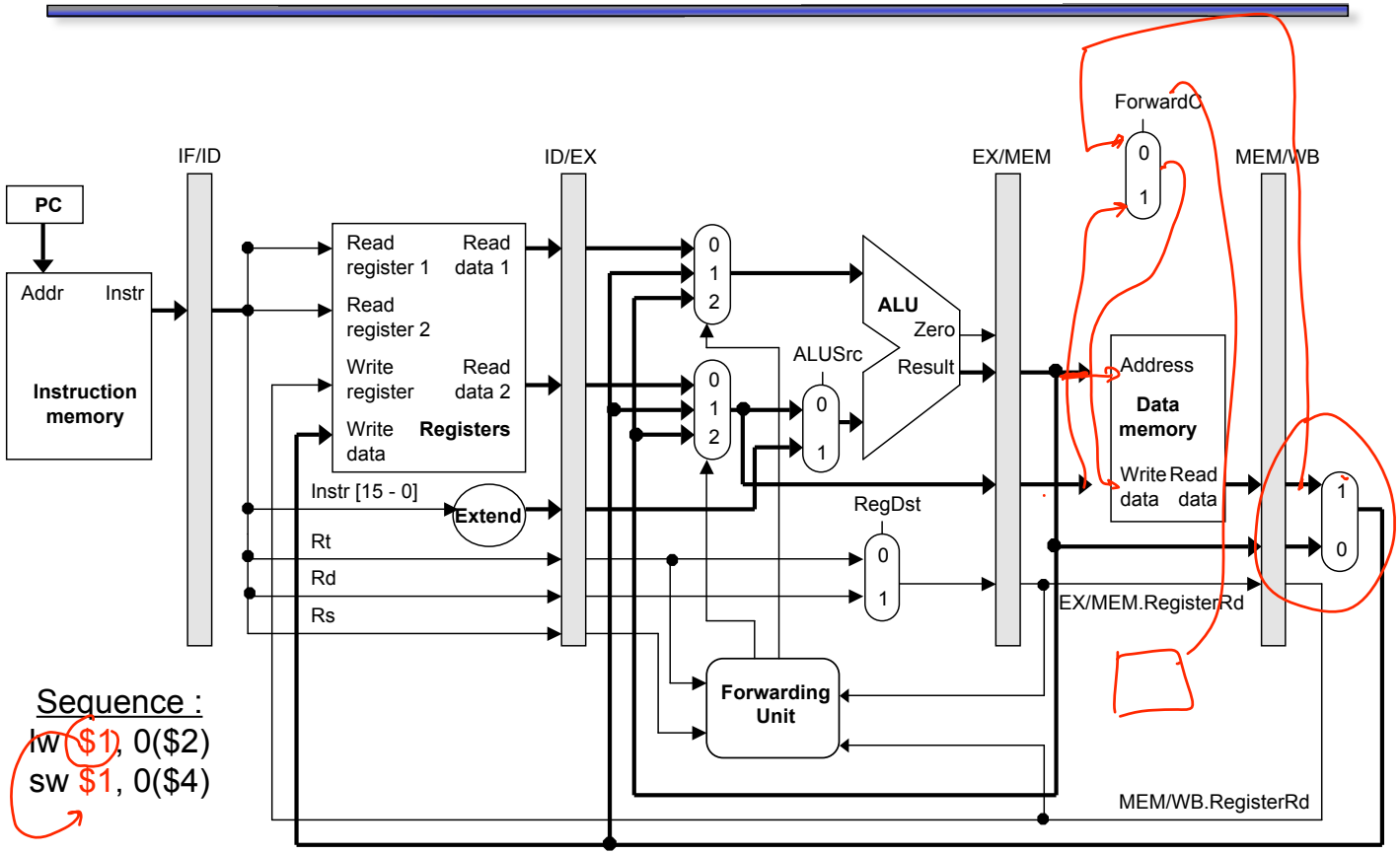
- A harder case:



- 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 6.4 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

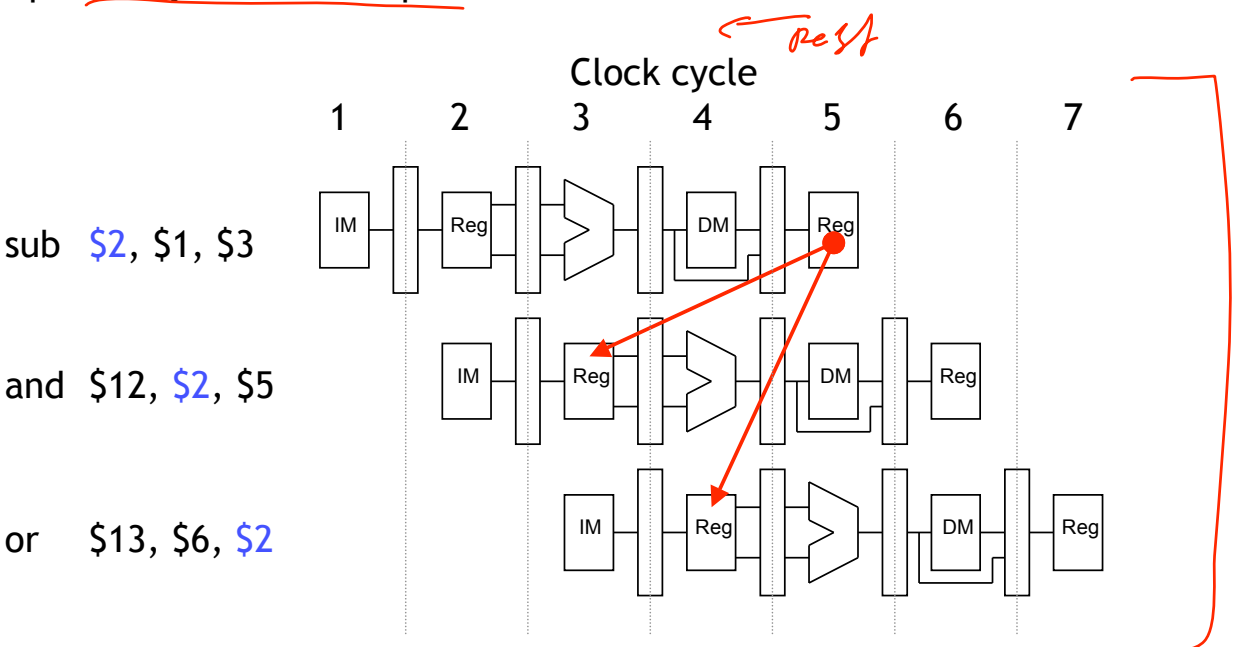
- In real code, most instructions are dependent upon other ones.
  - This can lead to data hazards in our original pipelined datapath.
  - Instructions can't write back to the register file soon enough for the next two instructions to read.
- Forwarding eliminates data hazards involving arithmetic instructions.
  - 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

- So far, we have 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.

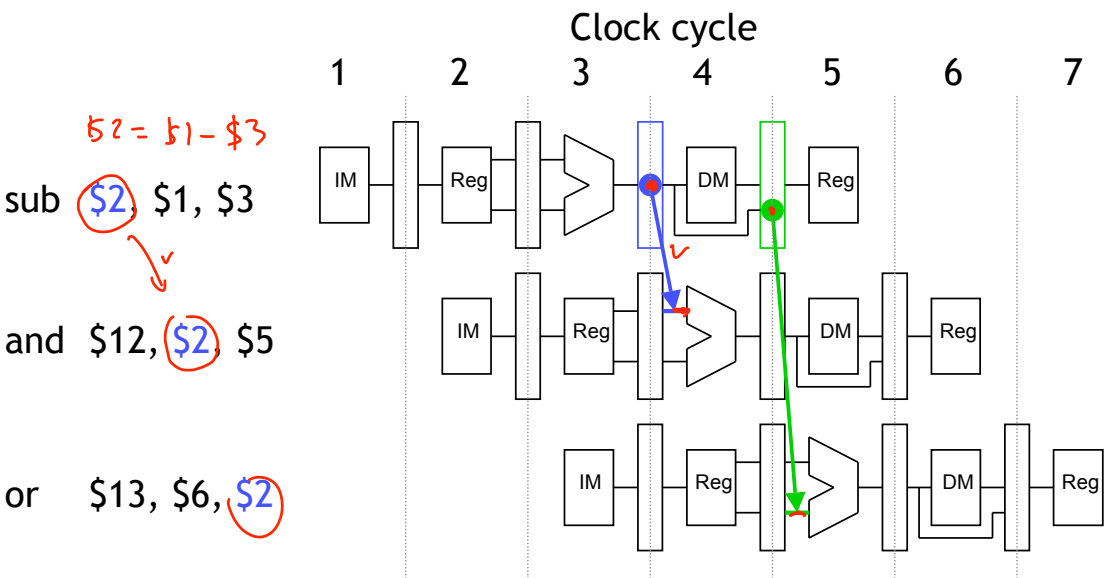
Data hazard review

- 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.



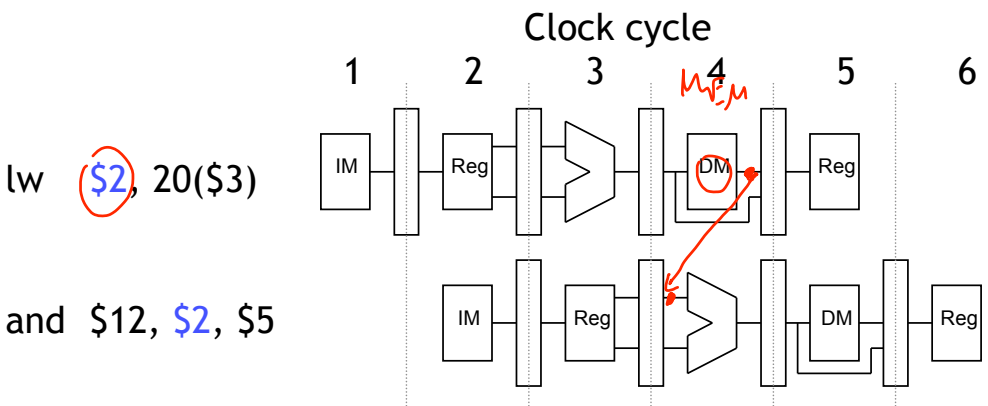
Forwarding

- 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.



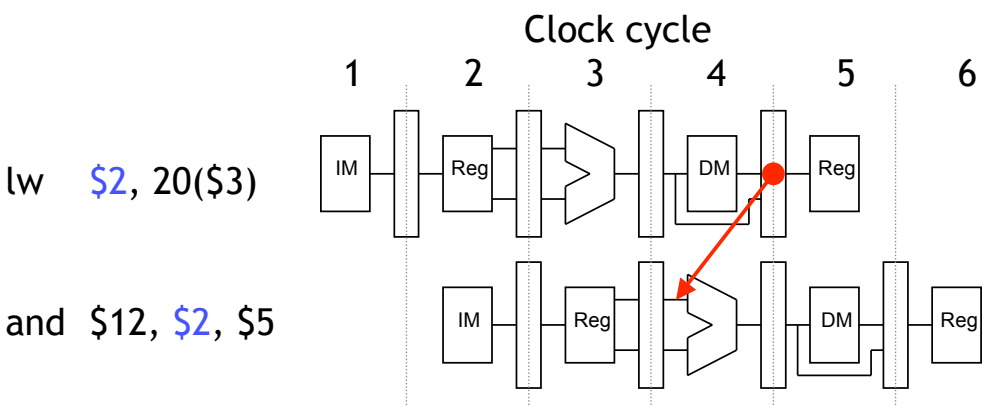
What about loads?

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



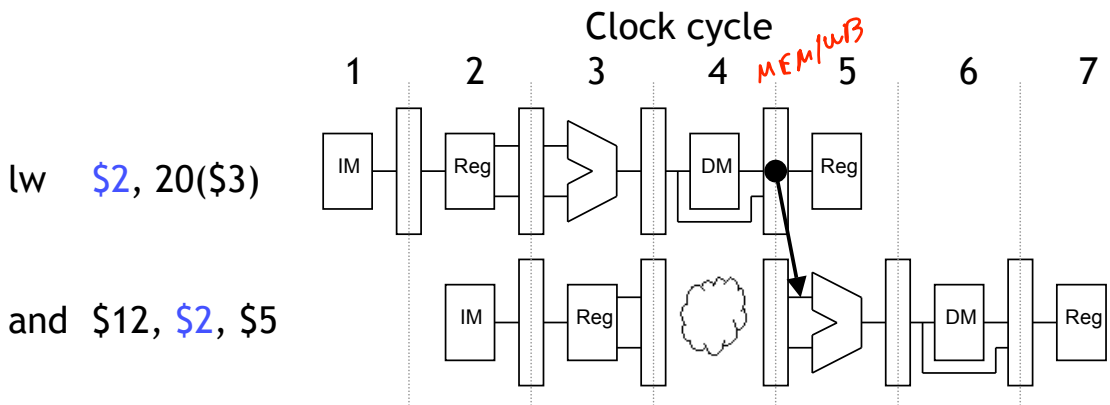
What about loads?

- 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

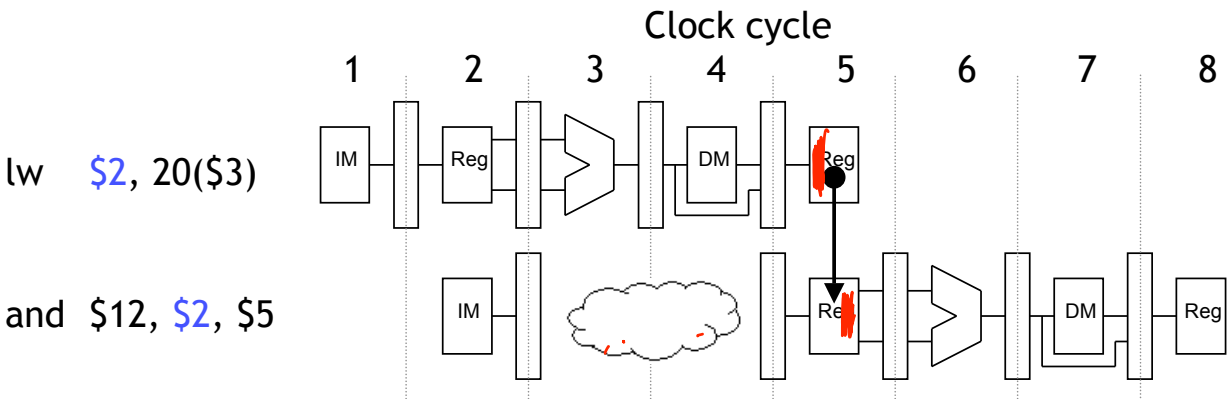
- The easiest solution is to stall the pipeline.
- We could delay the AND instruction by introducing a one-cycle delay into the pipeline, sometimes called a bubble.



- 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

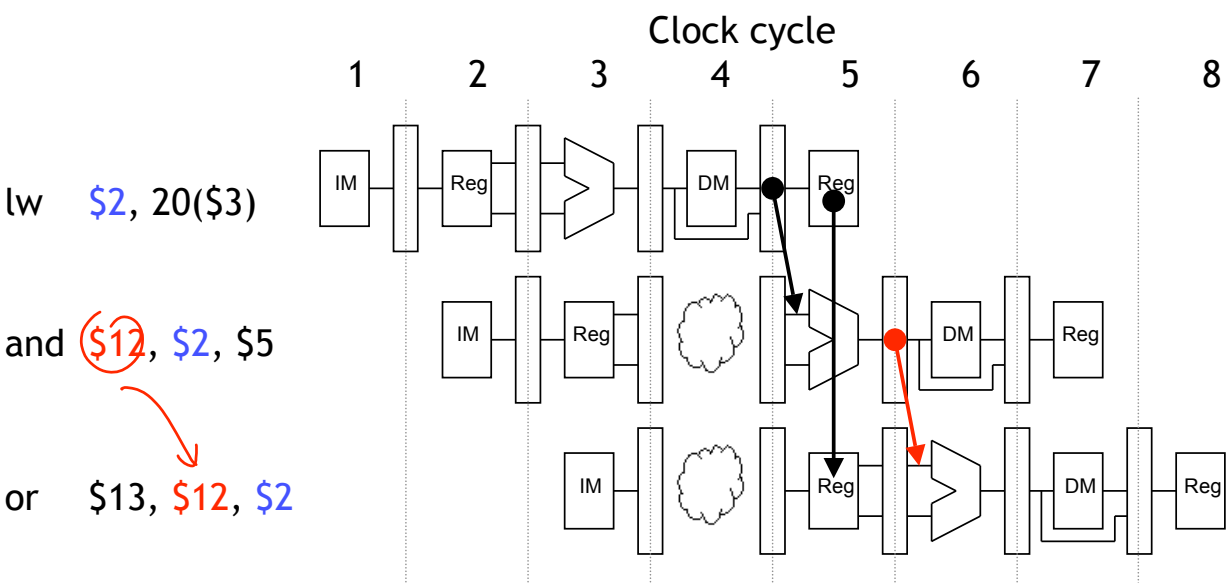
- Without forwarding, we'd have to stall for *two* cycles to wait for the LW instruction's writeback stage.



- 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.

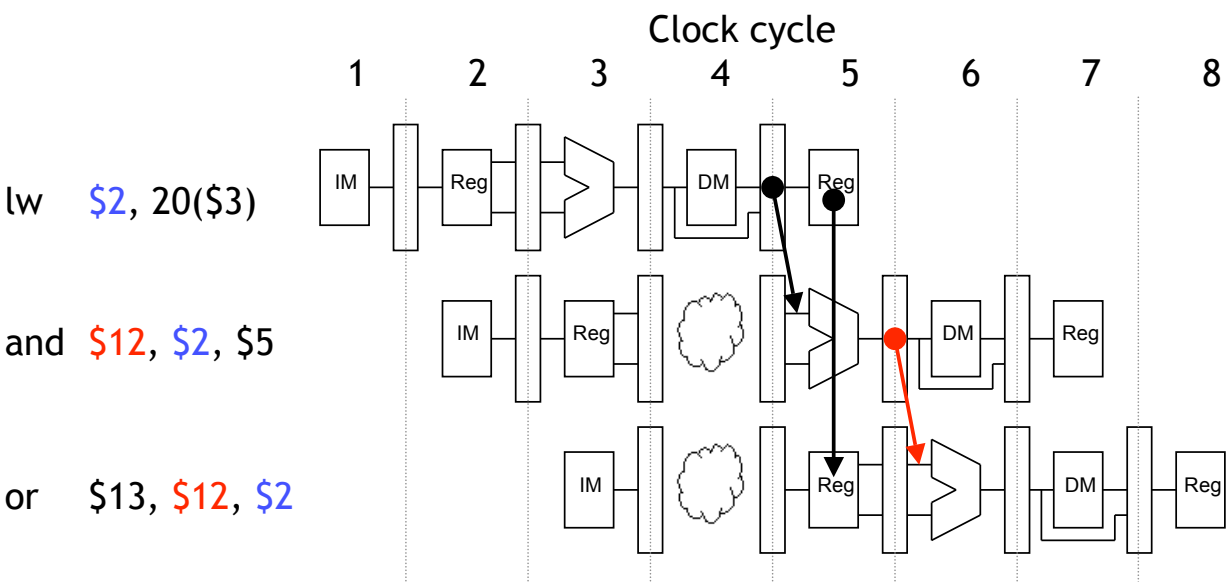
Stalling delays the entire pipeline

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



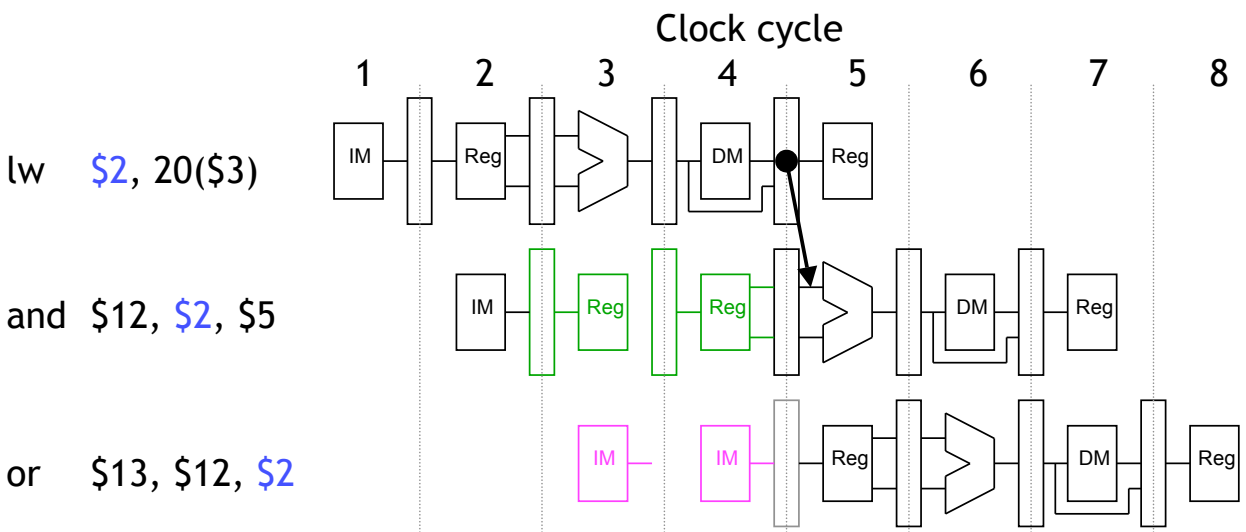
Stalling delays the entire pipeline

- 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

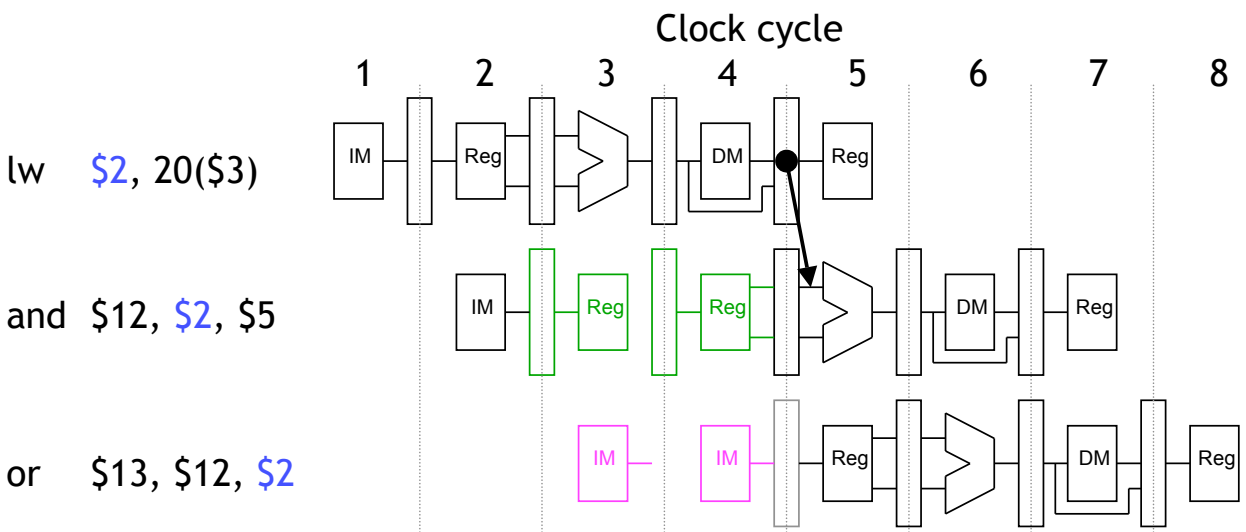
- One way to implement a stall is to force the two instructions after LW to pause and remain in their ID and IF stages for one extra cycle.



- 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.

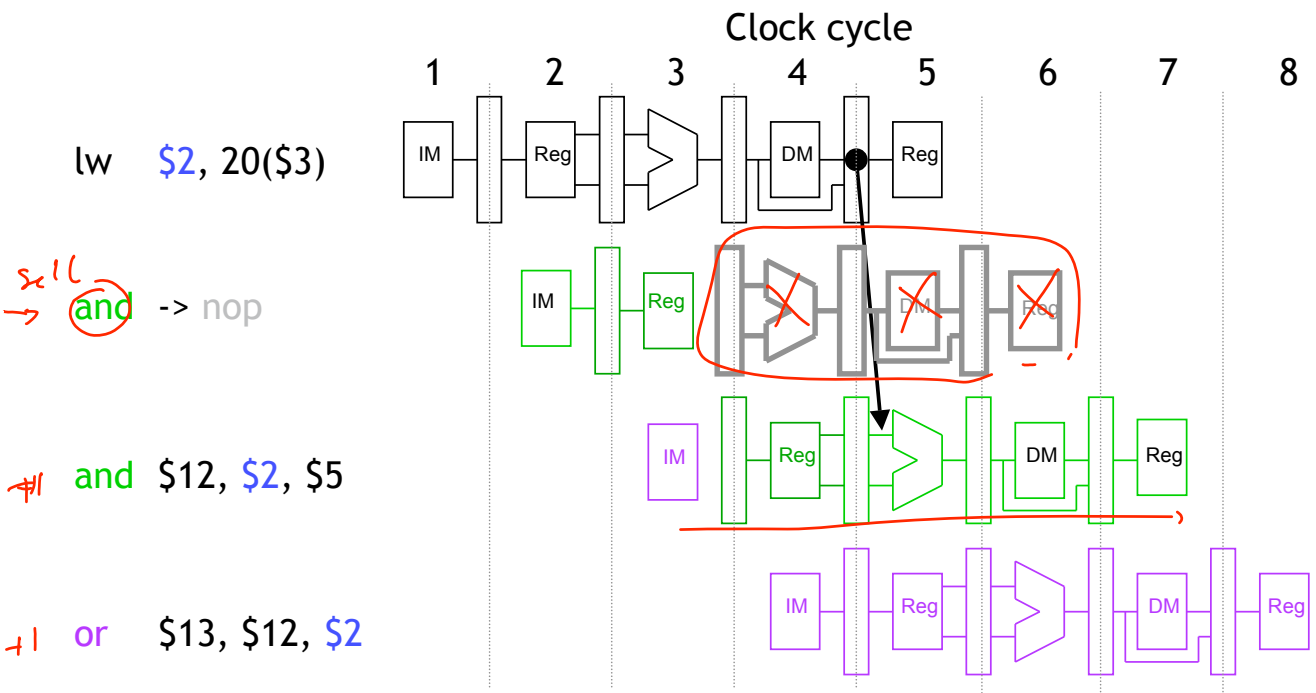
What about EXE, MEM, WB

- 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 instruction into the pipeline

Detecting stalls

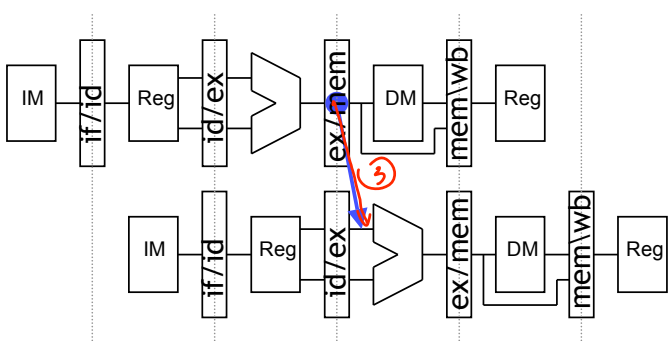
- Detecting stall is much like detecting data hazards.

- 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

older: sub $2, $1, $3

newer: and $12, $2, $5

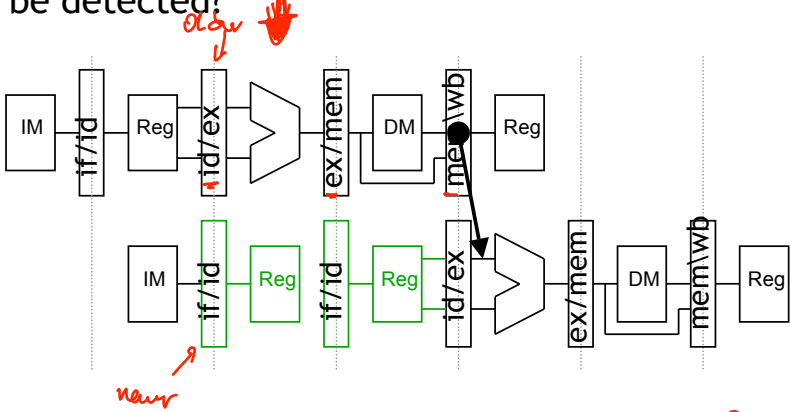


Detecting Stalls, cont.

- When should stalls be detected?

① lw $2, 20($3)

② and $12, $2, $5



- What is the stall condition?

① is a load
2's source is = 1's dest.

if (ID/EX.MEM_READ &&
((ID/EX.RS == IF/ID.RS) ||
(ID/EX.RT == IF/ID.RT))
)
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

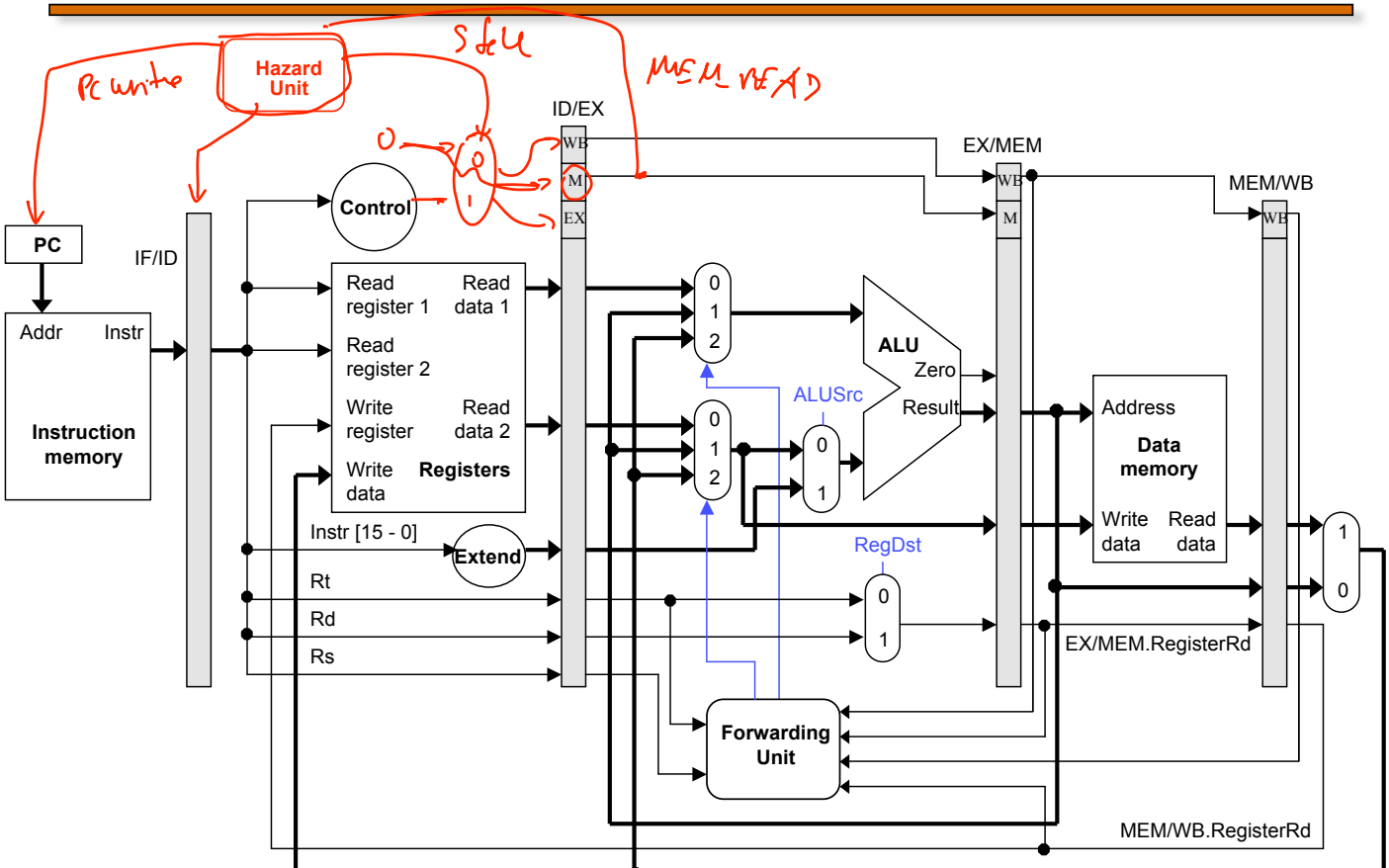
...and the LW destination is one of the current source registers.

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

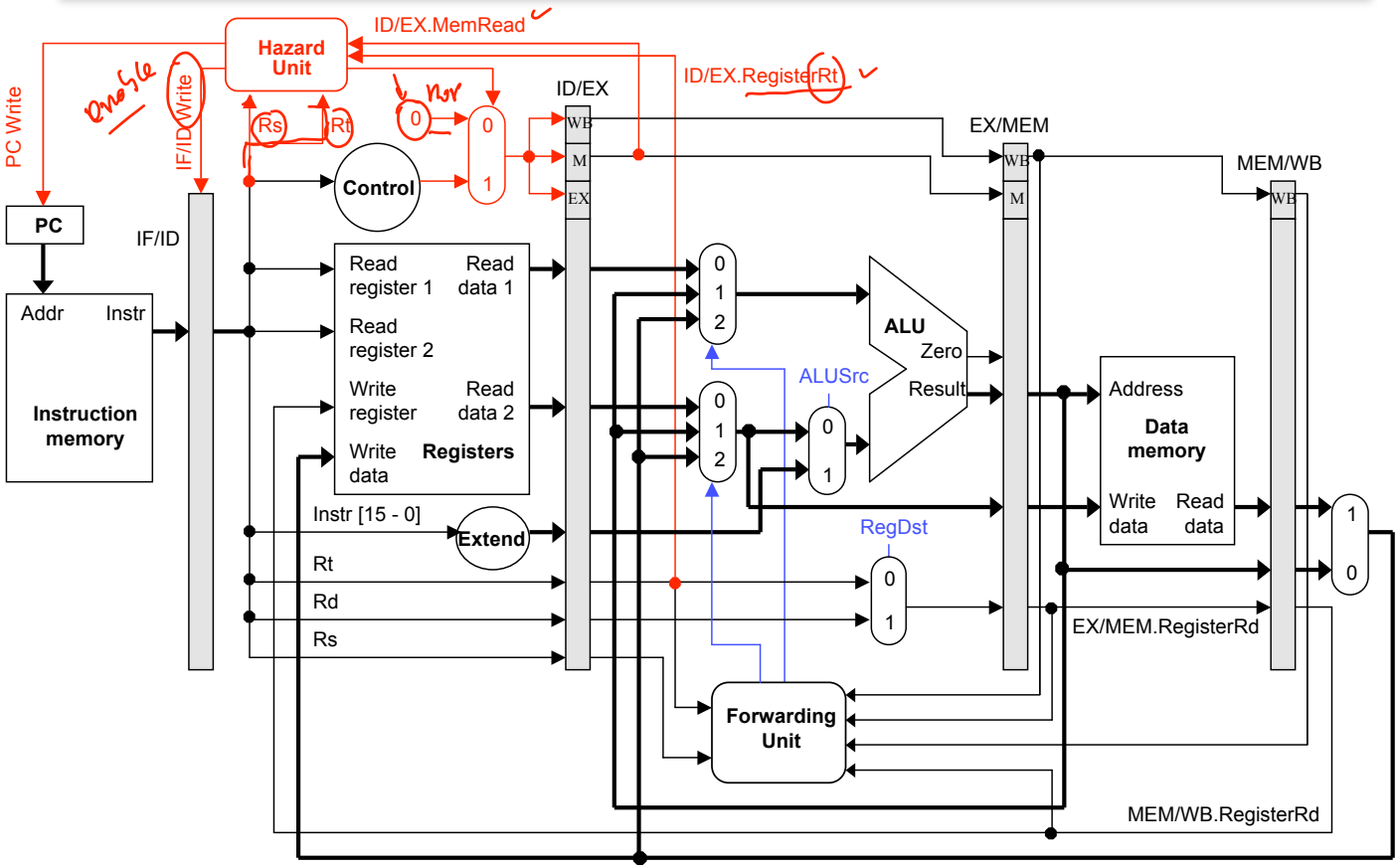
- The complete test for stalling is the conjunction of these two conditions.

```
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

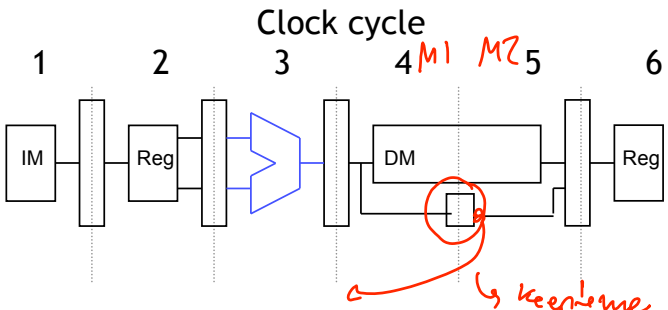


The hazard detection unit

- 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

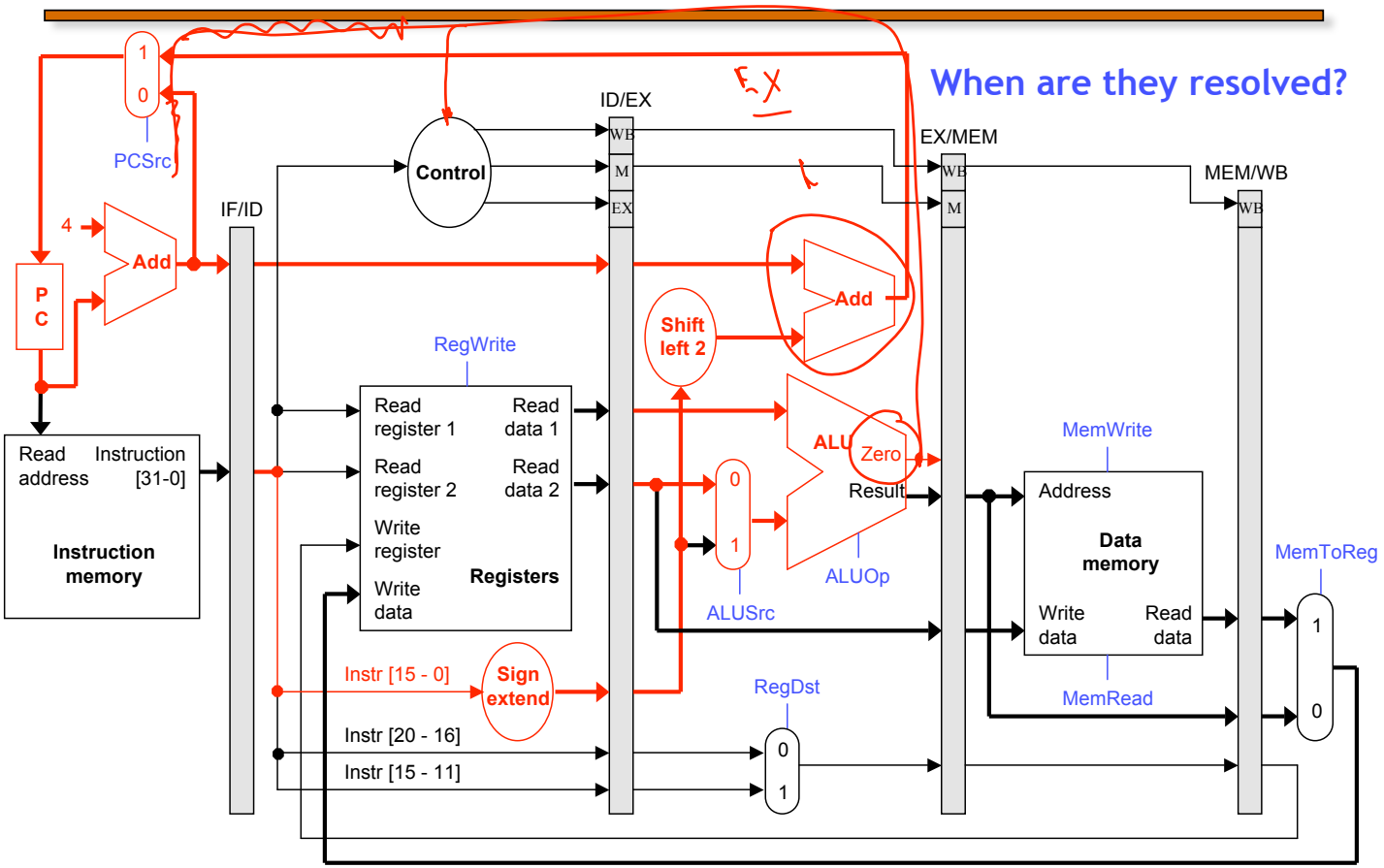
- What if data memory access was so slow, we wanted to pipeline it over 2 cycles?



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

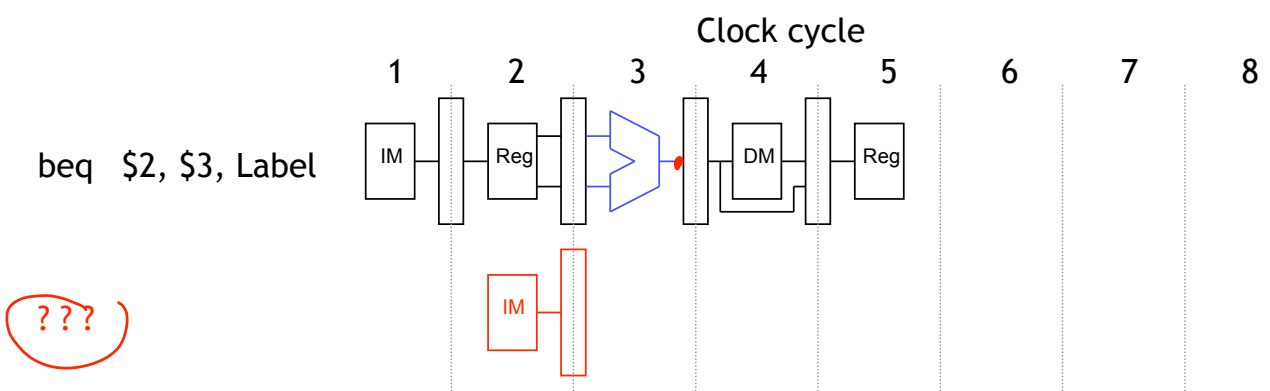
```
lw      r13, 0(r11)
add     r7, r8, r9
add     r15, r7, r13
```

Branches in the original pipelined datapath



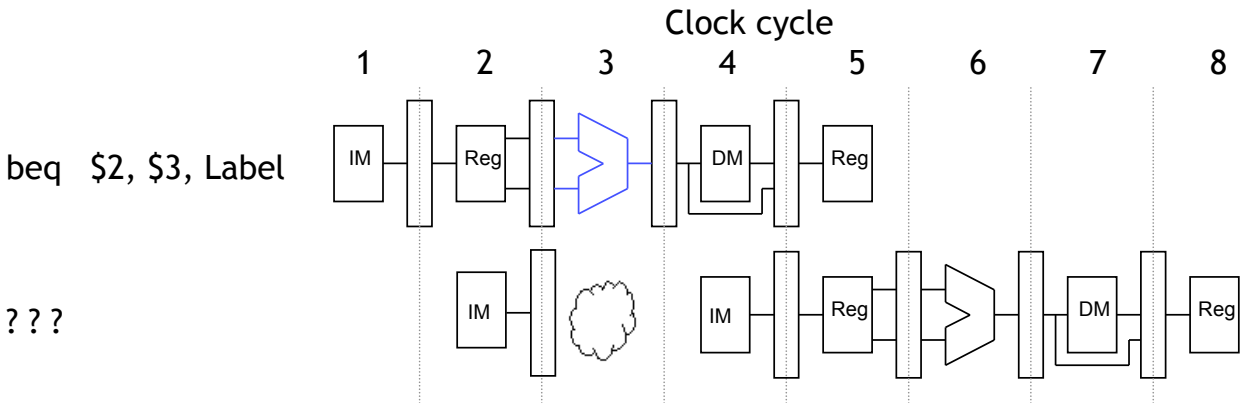
Branches

- 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

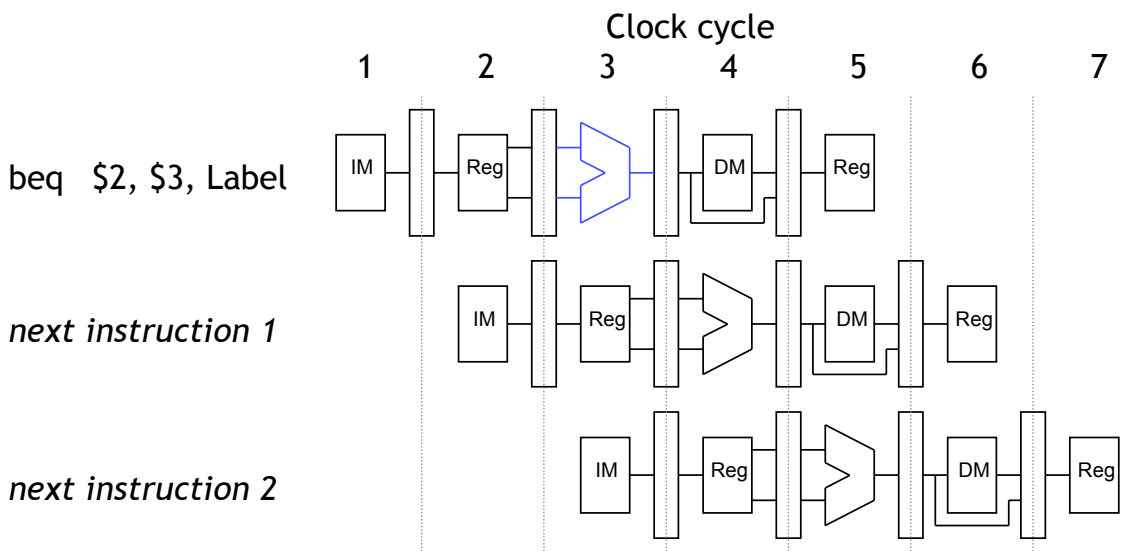
- Again, stalling is always one possible solution.



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

Branch prediction

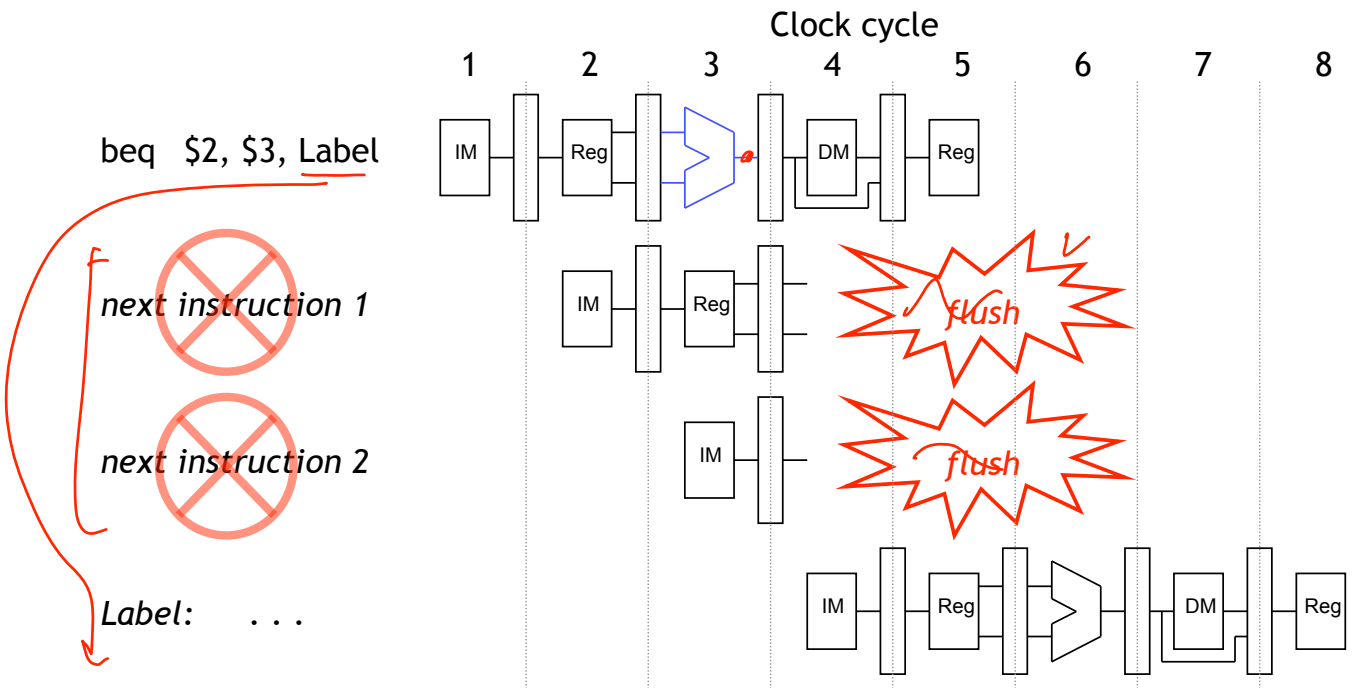
- Another approach is to *guess* whether or not the branch is taken.
  - In terms of hardware, it's easier to assume the branch is *not* taken.
  - This way we just increment the PC and continue execution, as for normal instructions.
- If we're correct, then there is no problem and the pipeline keeps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



Label

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.

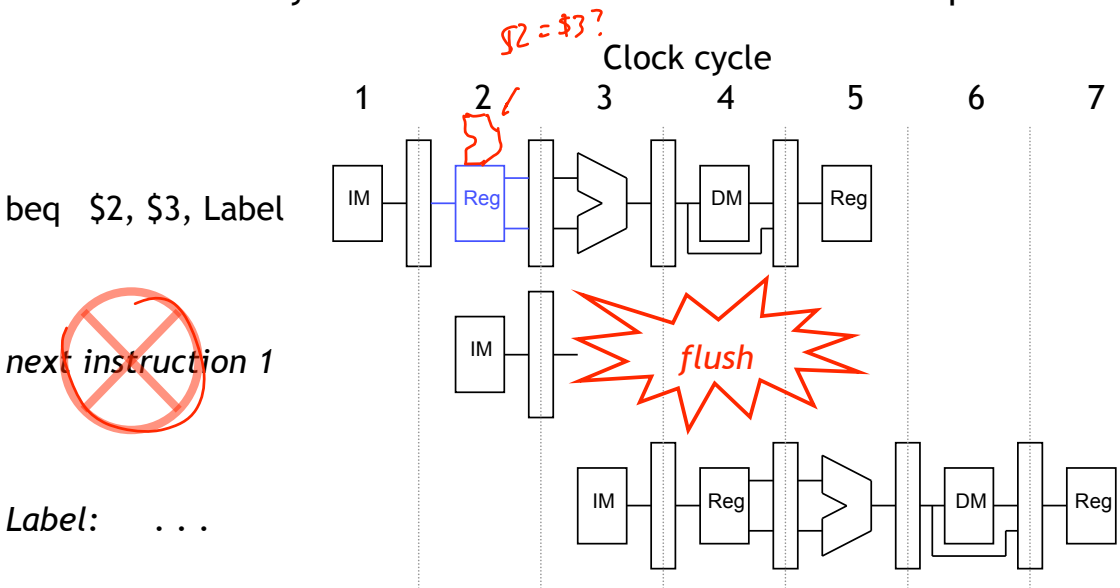


Performance gains and losses

- 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

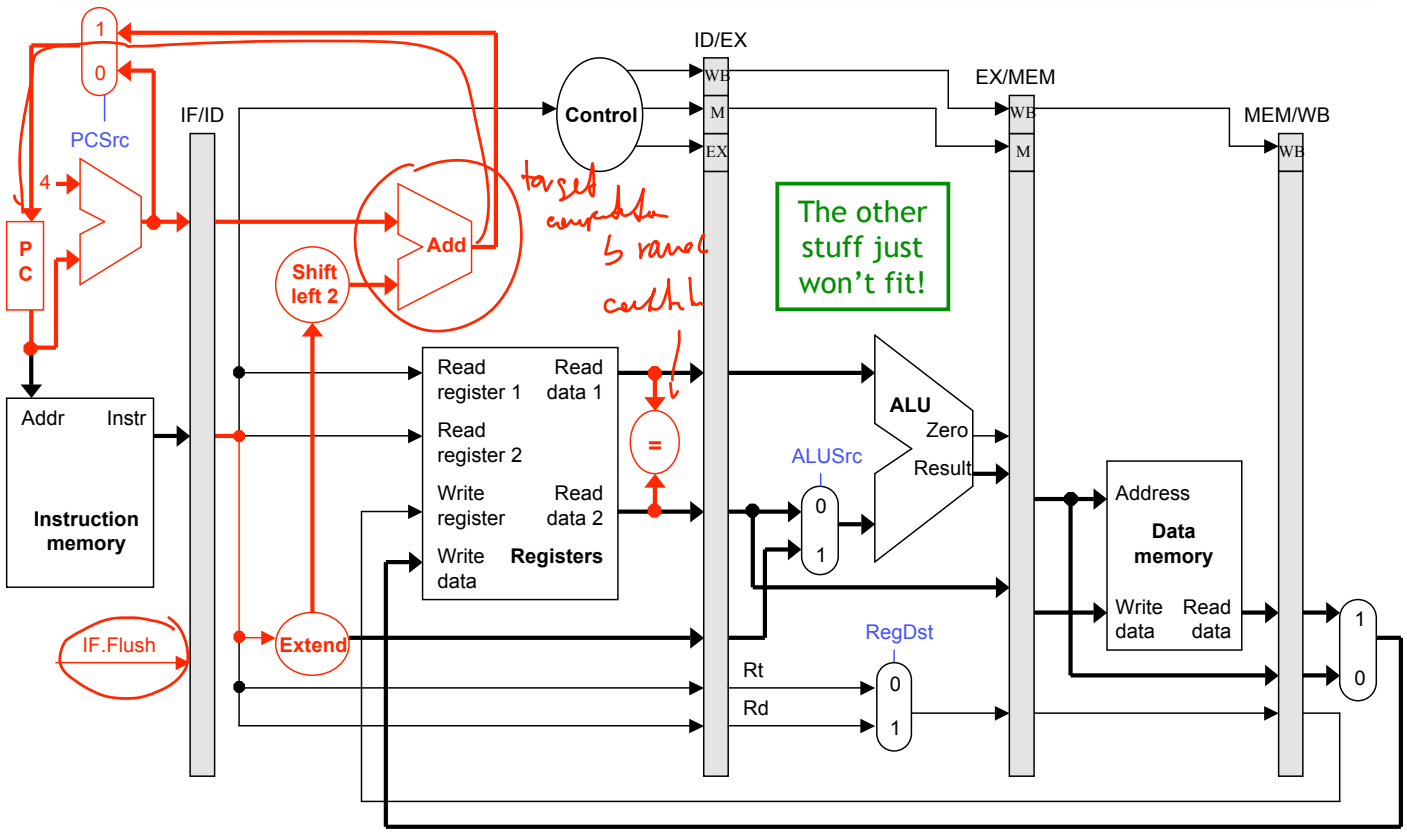
- We can actually decide the branch a little earlier, in ID instead of EX.
  - Our sample instruction set has only a BEQ.
  - We can add a small comparison circuit to the ID stage, after the source registers are read.
- Then we would only need to flush one instruction on a misprediction.



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



Timing

- If no prediction:

```
IF   ID   EX   MEM  WB
     IF   IF   ID     EX   MEM WB    --- lost 1 cycle
```

- If prediction:
  - If Correct

```
IF   ID  EX   MEM WB
     IF  ID   EX    MEM   WB    -- no cycle lost
```

  - If Misprediction:

```
IF   ID  EX   MEM  WB
     IF0 IF1  ID    EX   MEM  WB   --- 1 cycle lost
```

↑ fancy ID stage and Prediction

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.