

Pipeline: Introduction

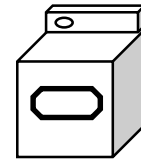
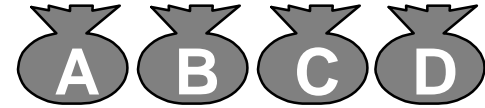
These slides are adapted from notes by Dr. David Patterson (UCB)

What is Pipelining?

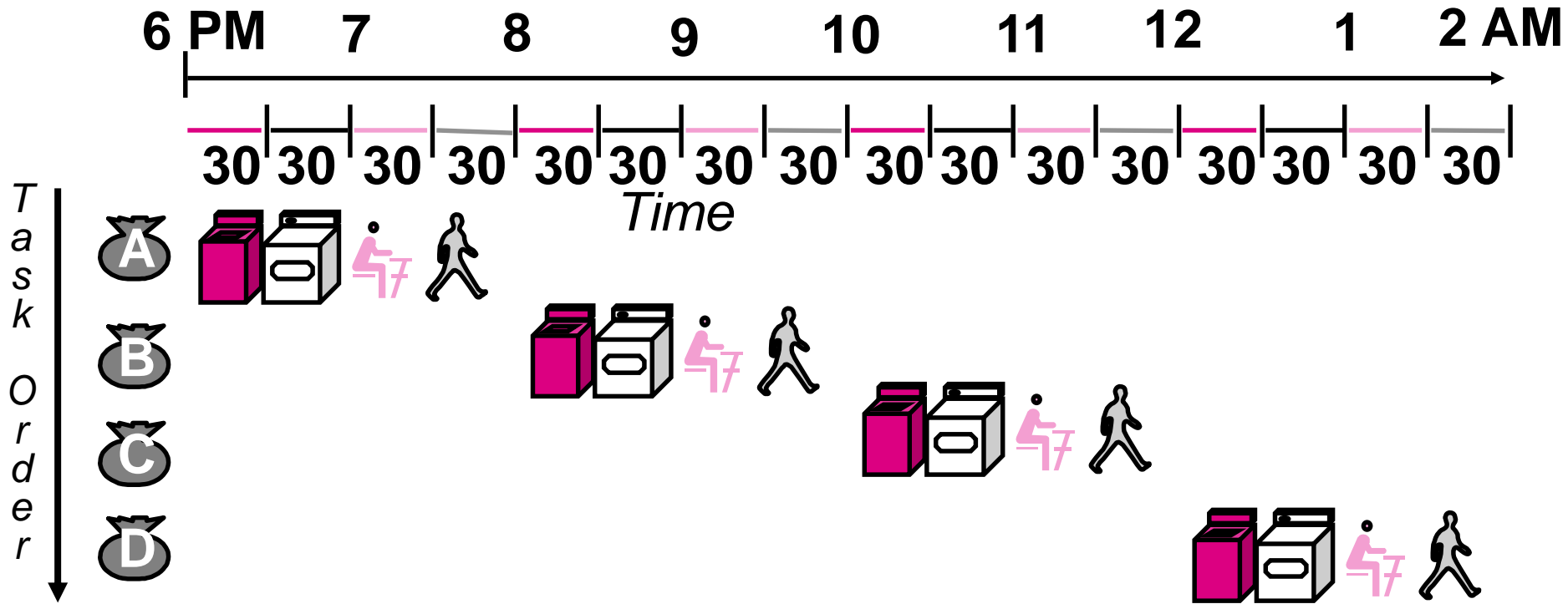
- A way of speeding up execution of instructions
- *Key idea:*
overlap execution of multiple instructions

The Laundry Analogy

- Anna, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold
- Washer takes 30 minutes
- Dryer takes 30 minutes
- “Folder” takes 30 minutes
- “Stasher” takes 30 minutes to put clothes into drawers

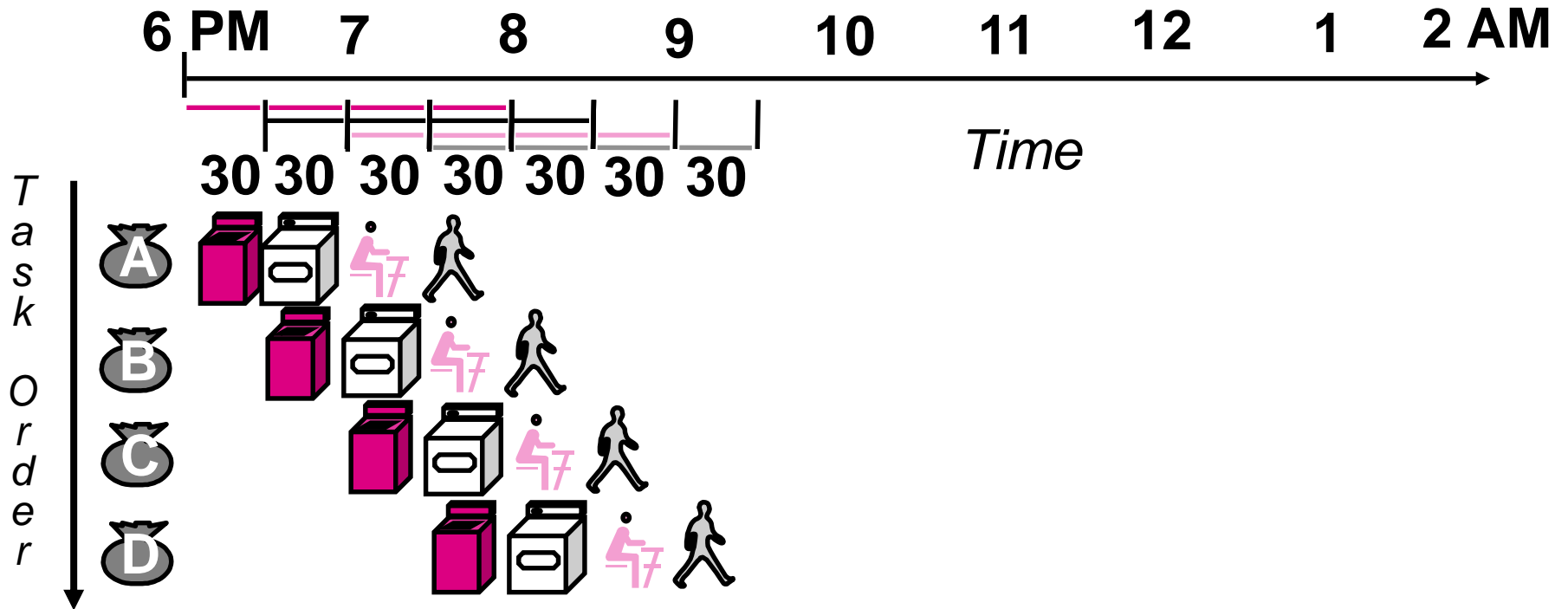


If we do laundry sequentially...



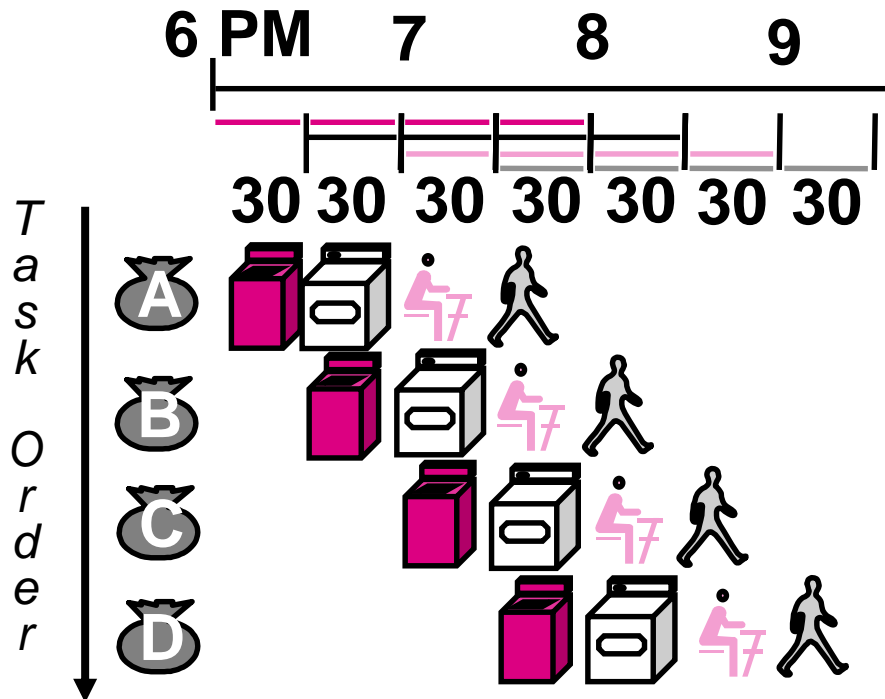
- Time Required: 8 hours for 4 loads

To Pipeline, We Overlap Tasks



- Time Required: 3.5 Hours for 4 Loads

To Pipeline, We Overlap Tasks



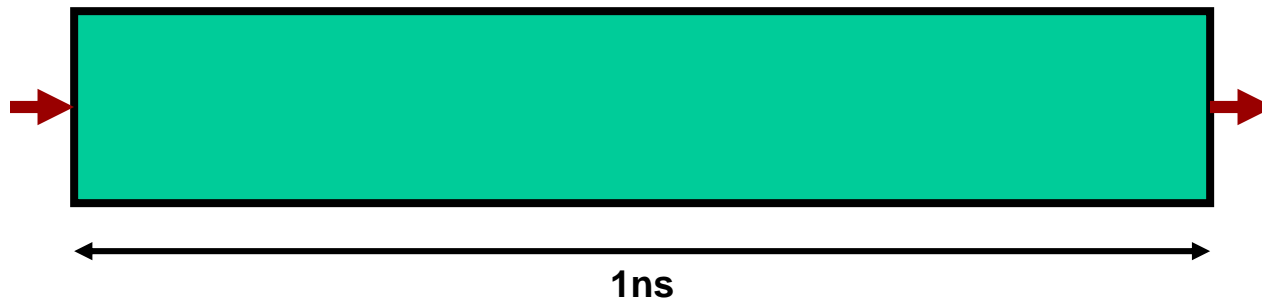
- Time*
- Does Pipelining help **latency** of single task? **No**
 - Does Pipelining help **throughput** of entire workload? **Yes**
 - Pipeline rate limited by ____?
the slowest pipeline stage
 - **Multiple** tasks operating simultaneously
 - Potential speedup = ? **Number of pipe stage**
 - Unbalanced lengths of pipe stages will ____ **reduces speedup**
 - Time to “**fill**” pipeline and time to “**drain**” it reduces speedup

Pipelining a Digital System

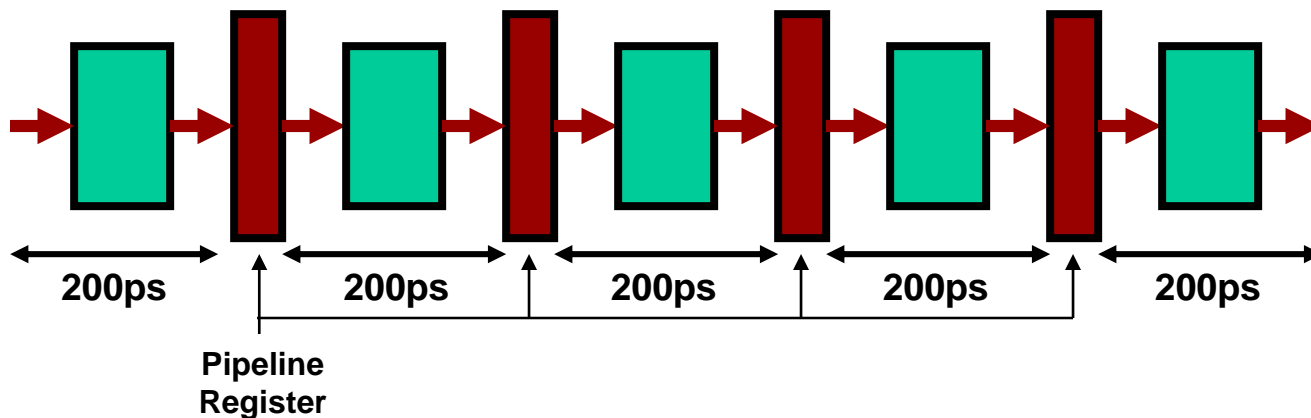
1 nanosecond = 10^{-9} second

1 picosecond = 10^{-12} second

- Key idea: break big computation up into pieces

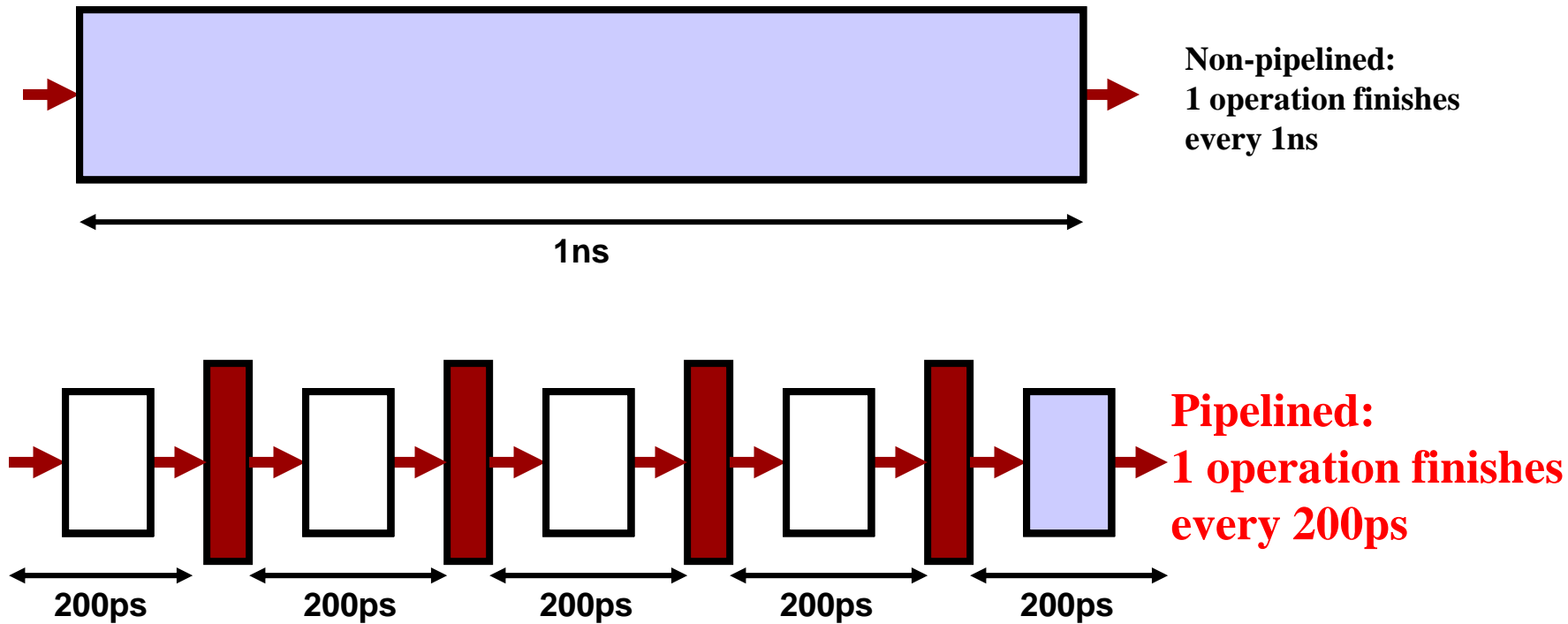


Separate each piece with a pipeline register



Pipelining a Digital System

- Why do this? Because it's faster for repeated computations



Comments about pipelining

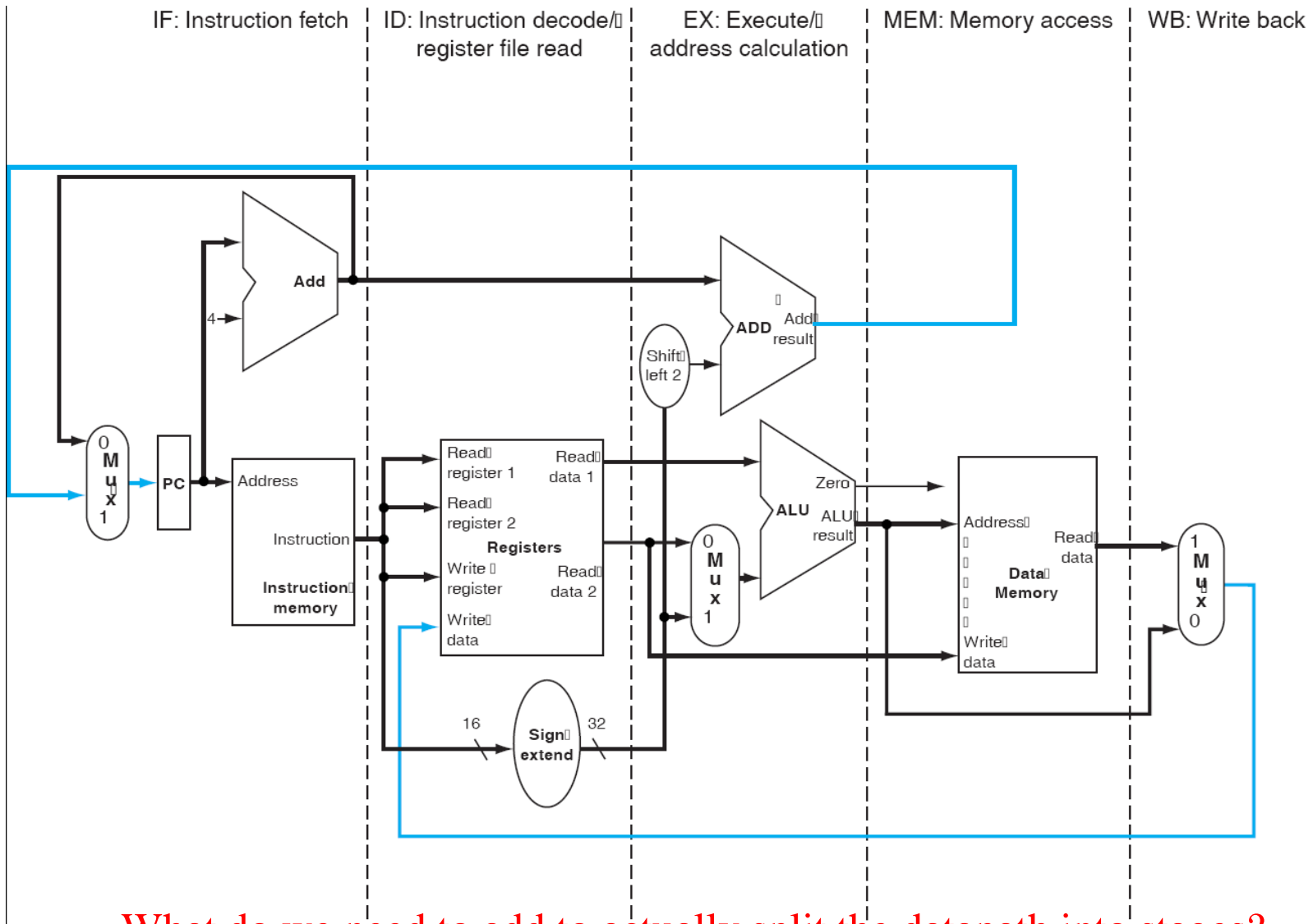
- Pipelining increases **throughput**, but not **latency**
 - Answer available every 200ps, BUT
 - A single computation still takes 1ns
- Limitations:
 - Computations must be divisible into stage size
 - ?

Pipeline registers add overhead

Pipelining a Processor

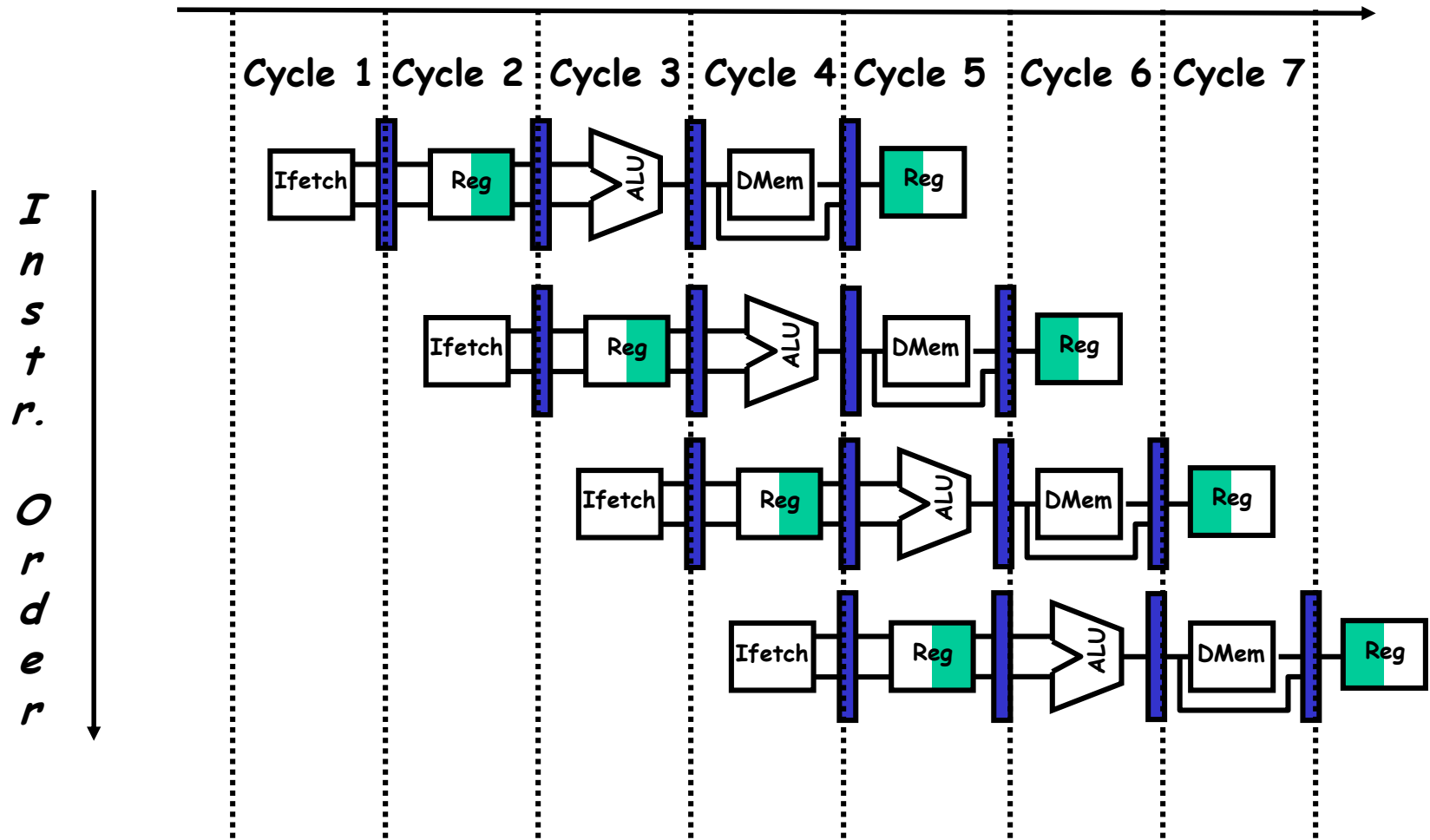
- Recall the 5 steps in instruction execution:
 1. Instruction Fetch (**IF**)
 2. Instruction Decode and Register Read (**ID**)
 3. Execution operation or calculate address (**EX**)
 4. Memory access (**MEM**)
 5. Write result into register (**WB**)
- Review: Single-Cycle Processor
 - All 5 steps done in a single clock cycle
 - Dedicated hardware required for each step

Review - Single-Cycle Processor



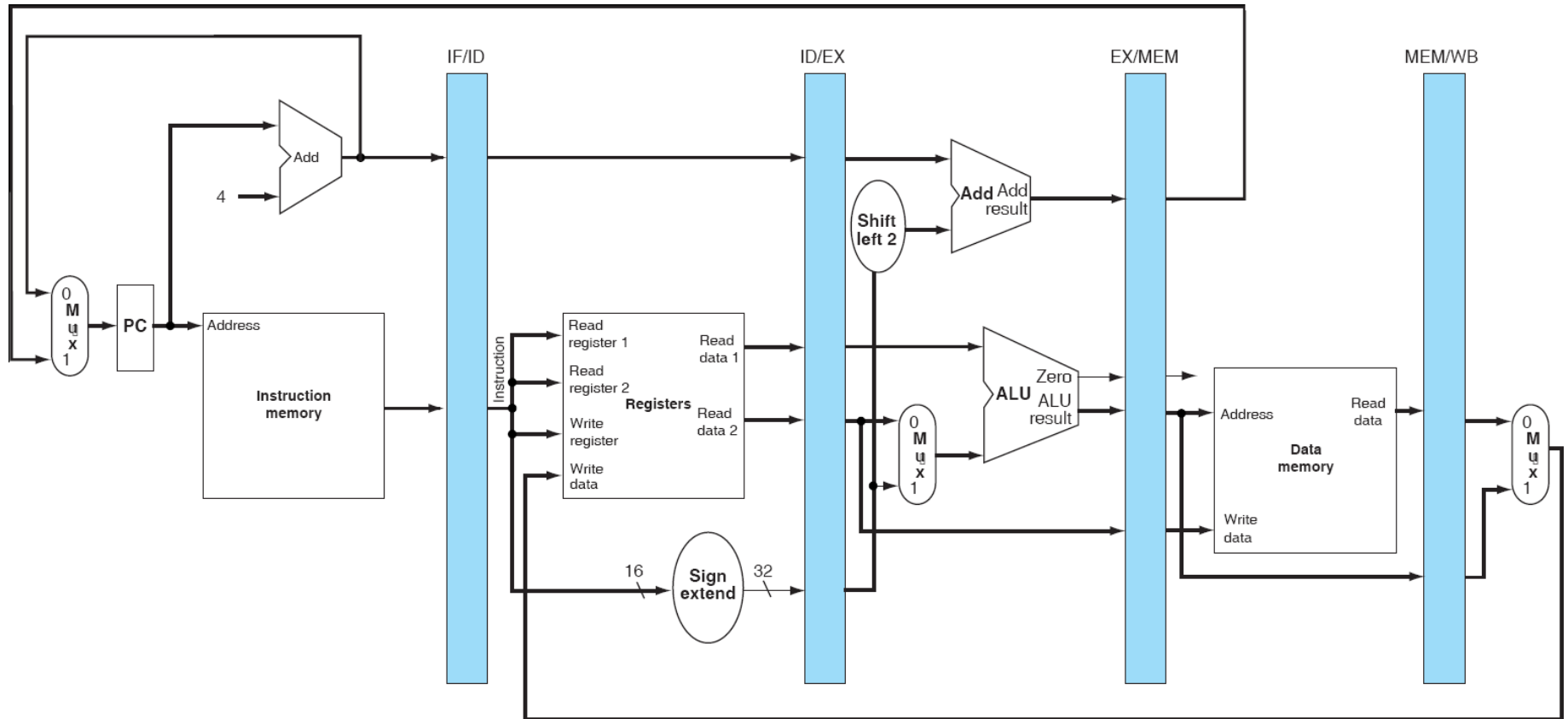
What do we need to add to actually split the datapath into stages?

The Basic Pipeline For MIPS



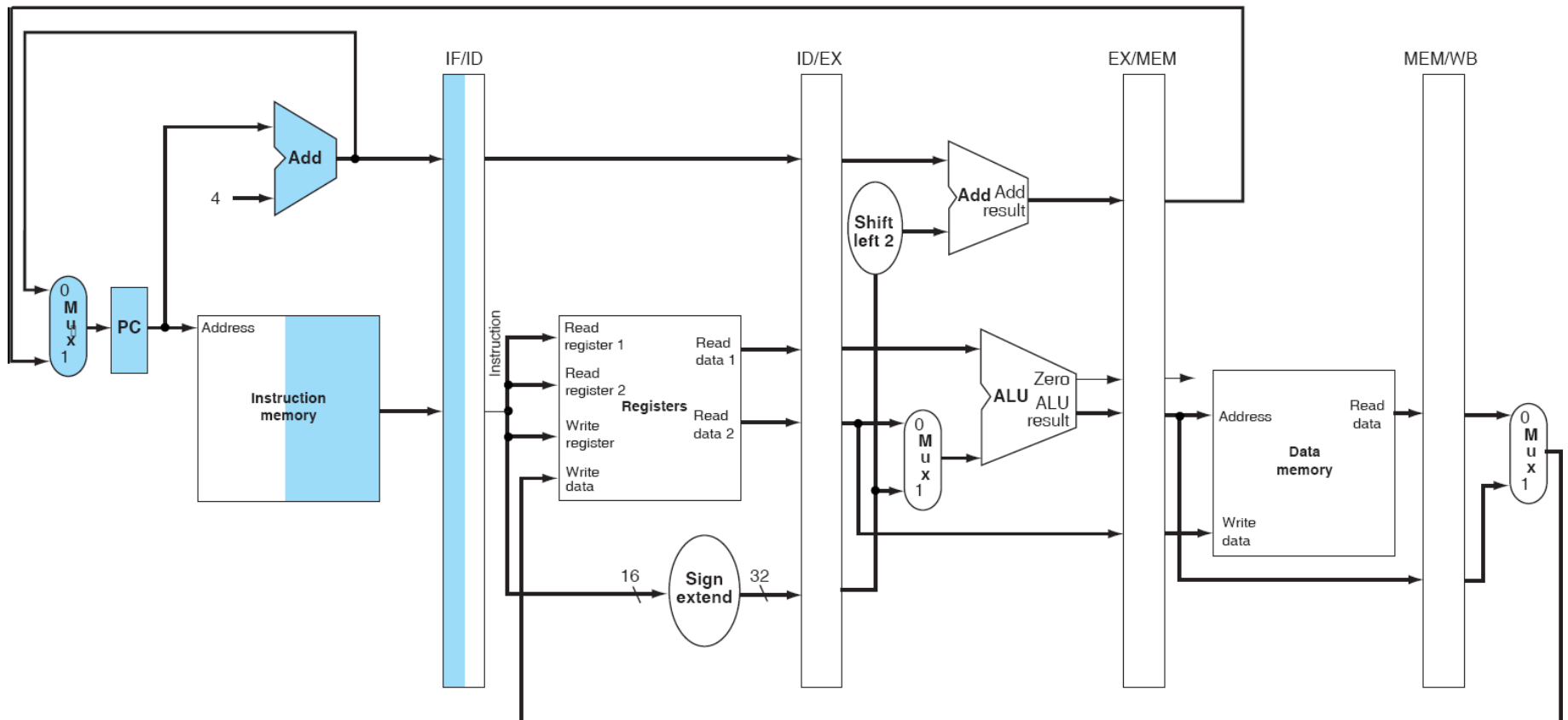
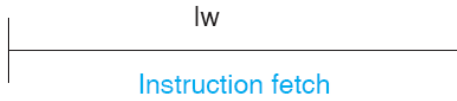
MIPS R3000 is a 32-bit architecture (RISC) (please refer to <http://eecs.harvard.edu/~cs161/notes/mips-part-I.pdf>)

Basic Pipelined Processor



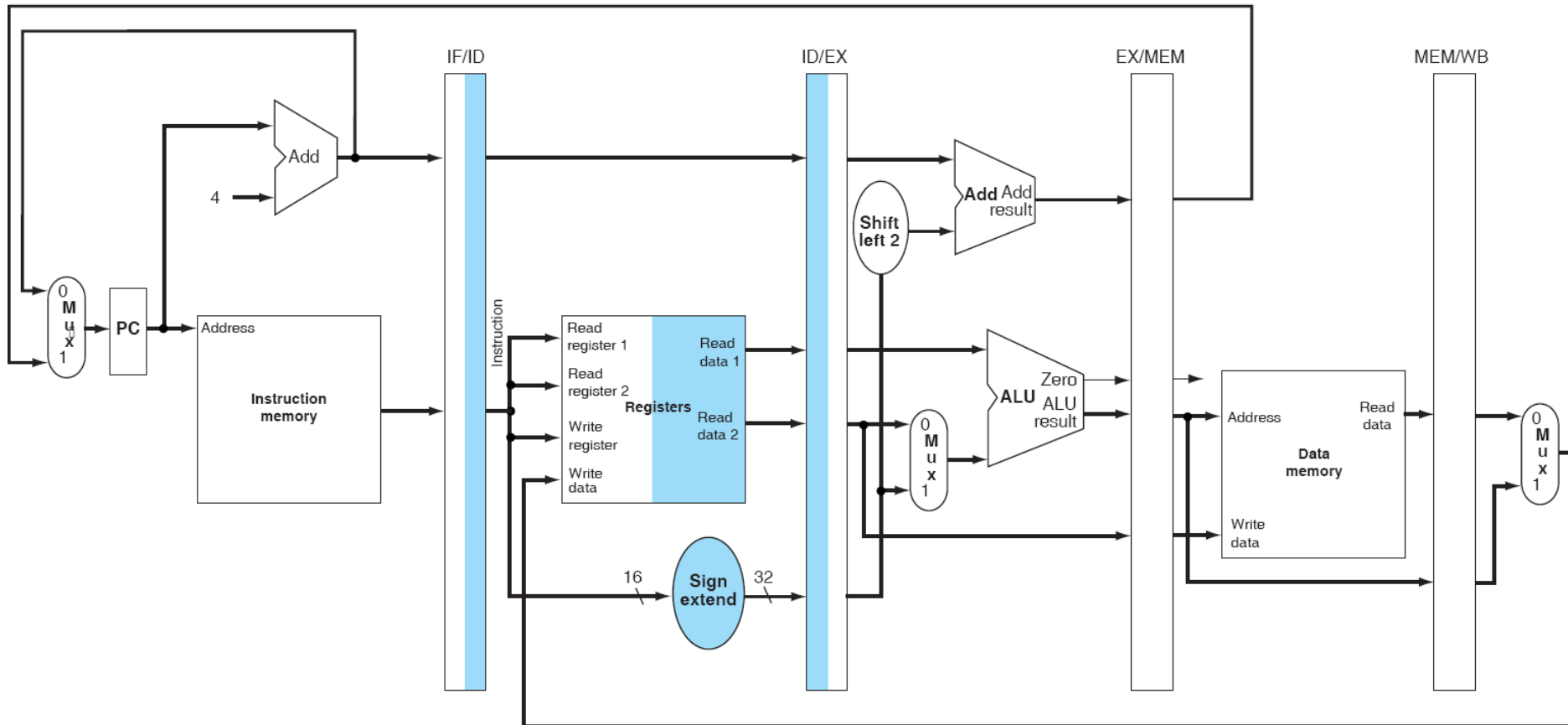
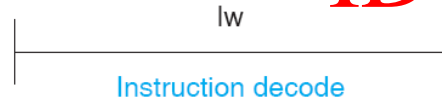
Pipeline example: lw

IF



Pipeline example: lw

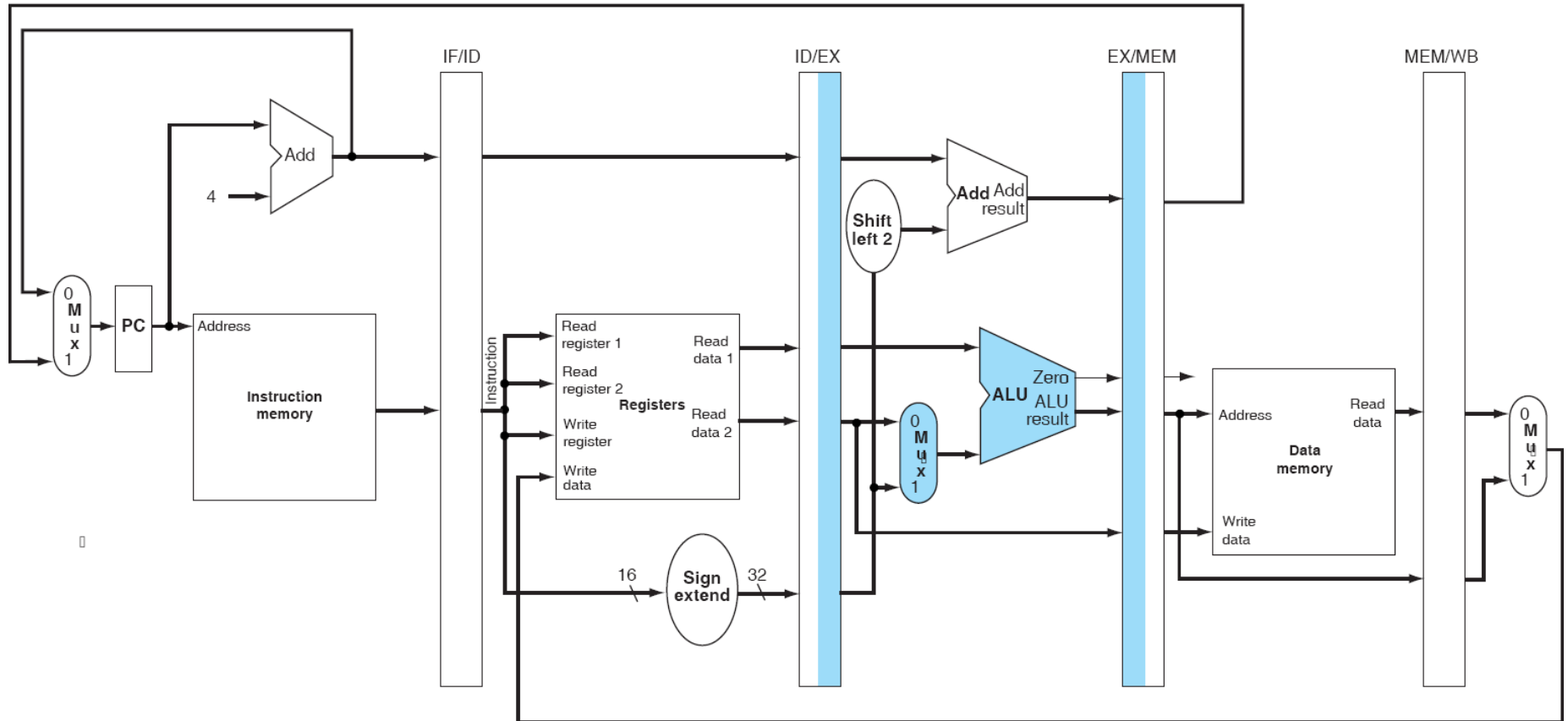
ID



Pipeline example: lw

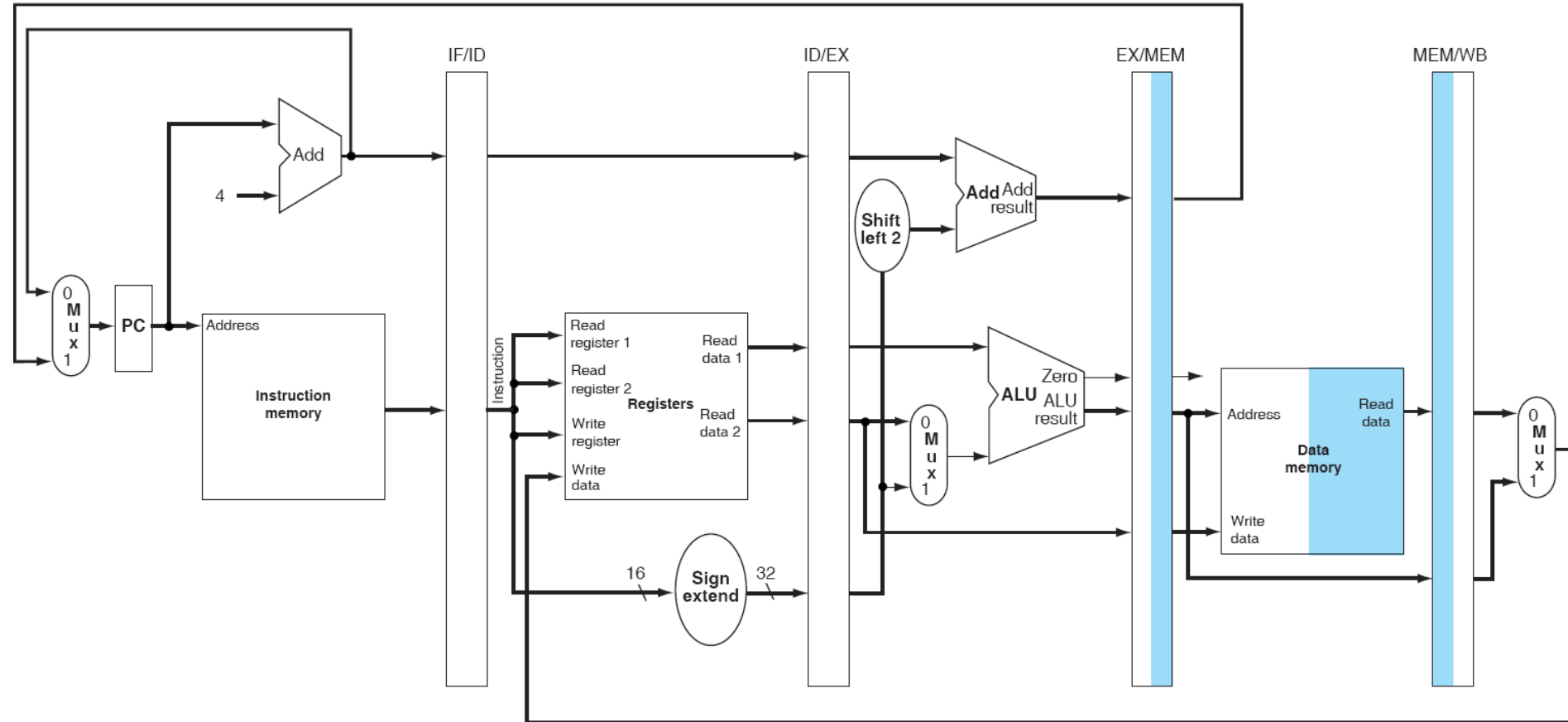
EX

lw
Execution



Pipeline example: lw

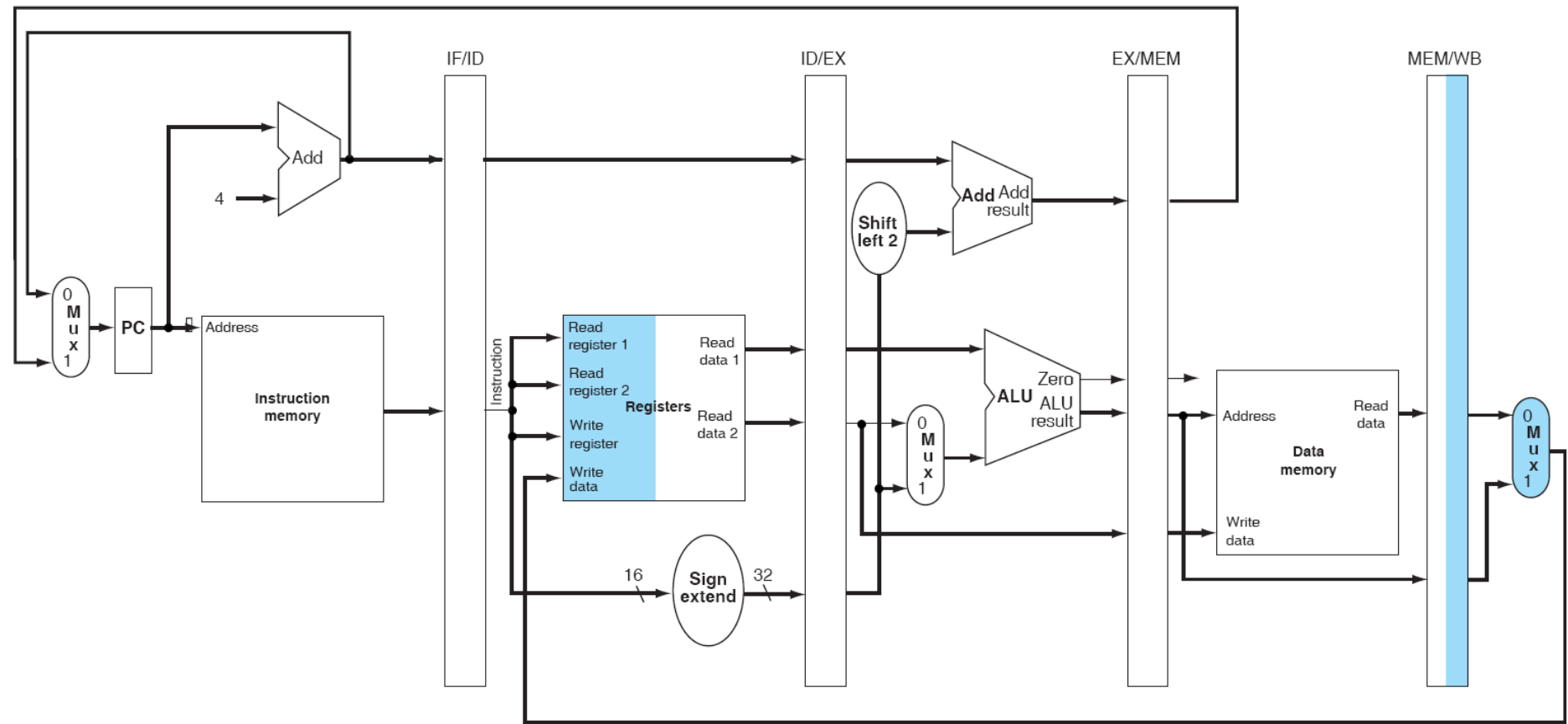
MEM



Pipeline example: lw

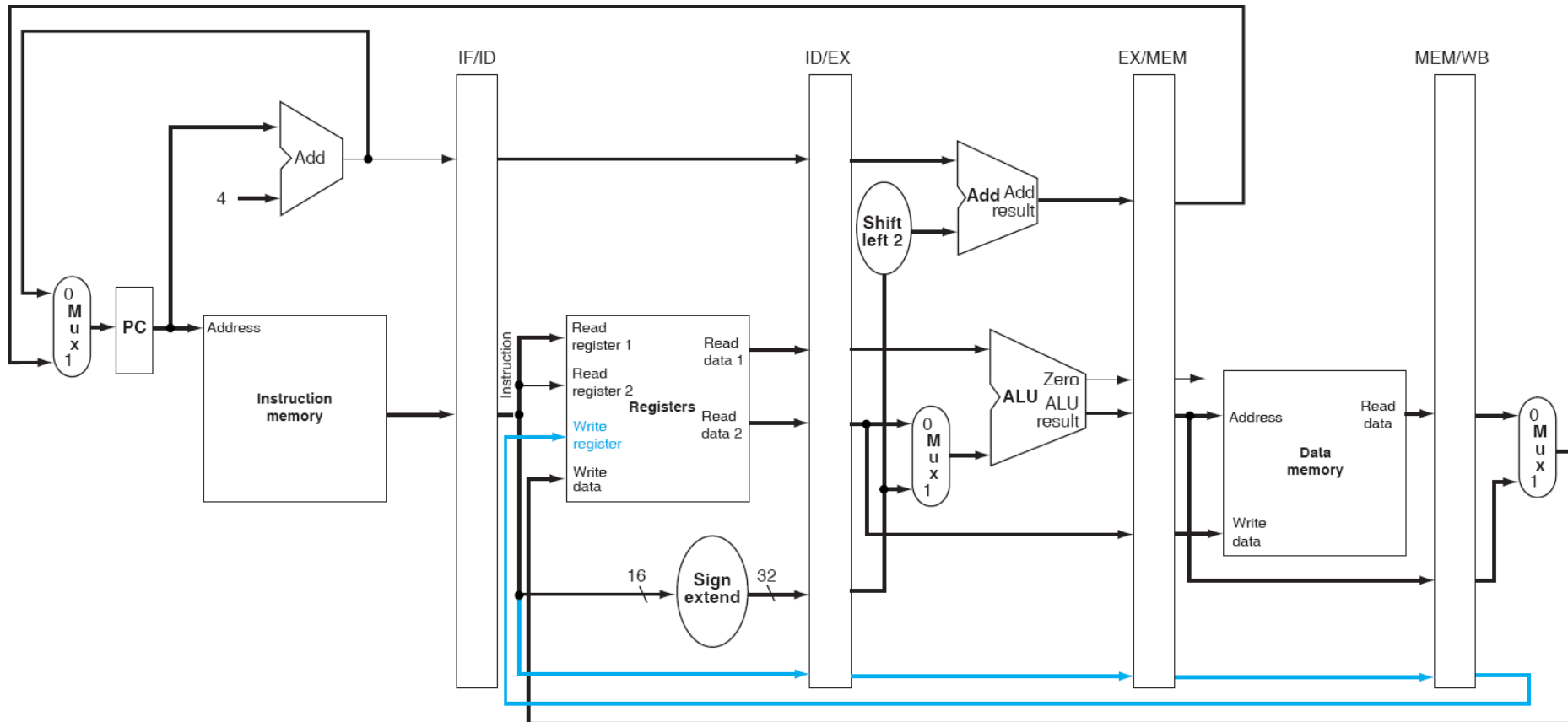
WB

lw
Write back



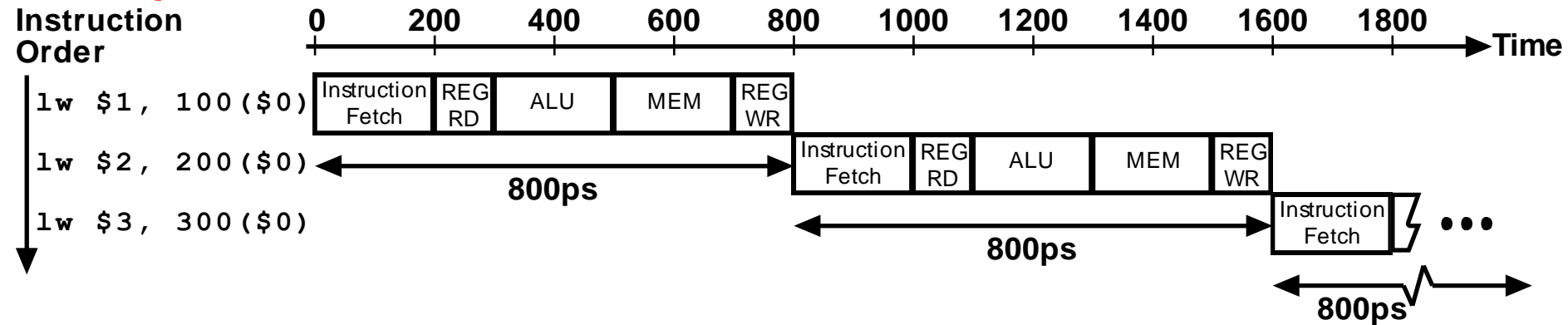
Can you find a problem?

Basic Pipelined Processor (Corrected)

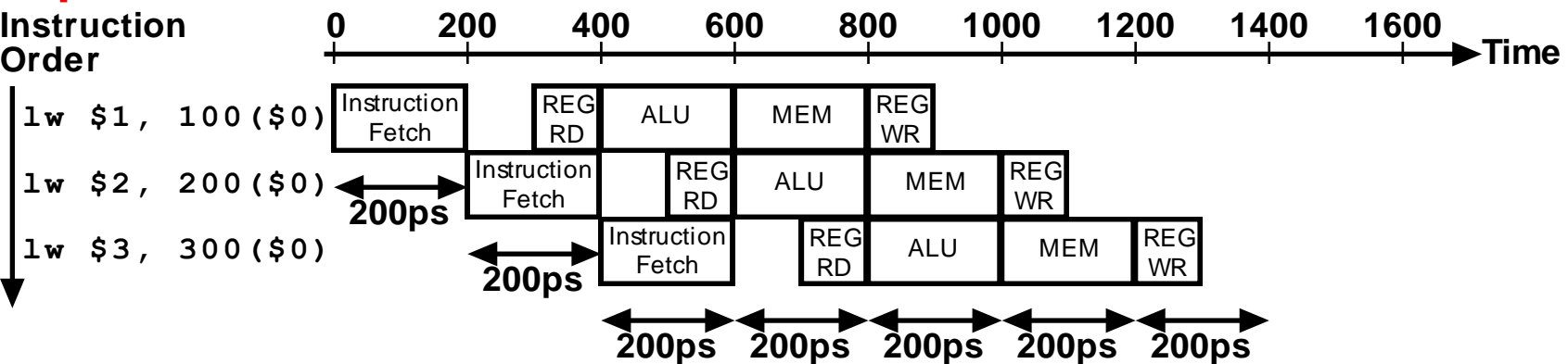


Single-Cycle vs. Pipelined Execution

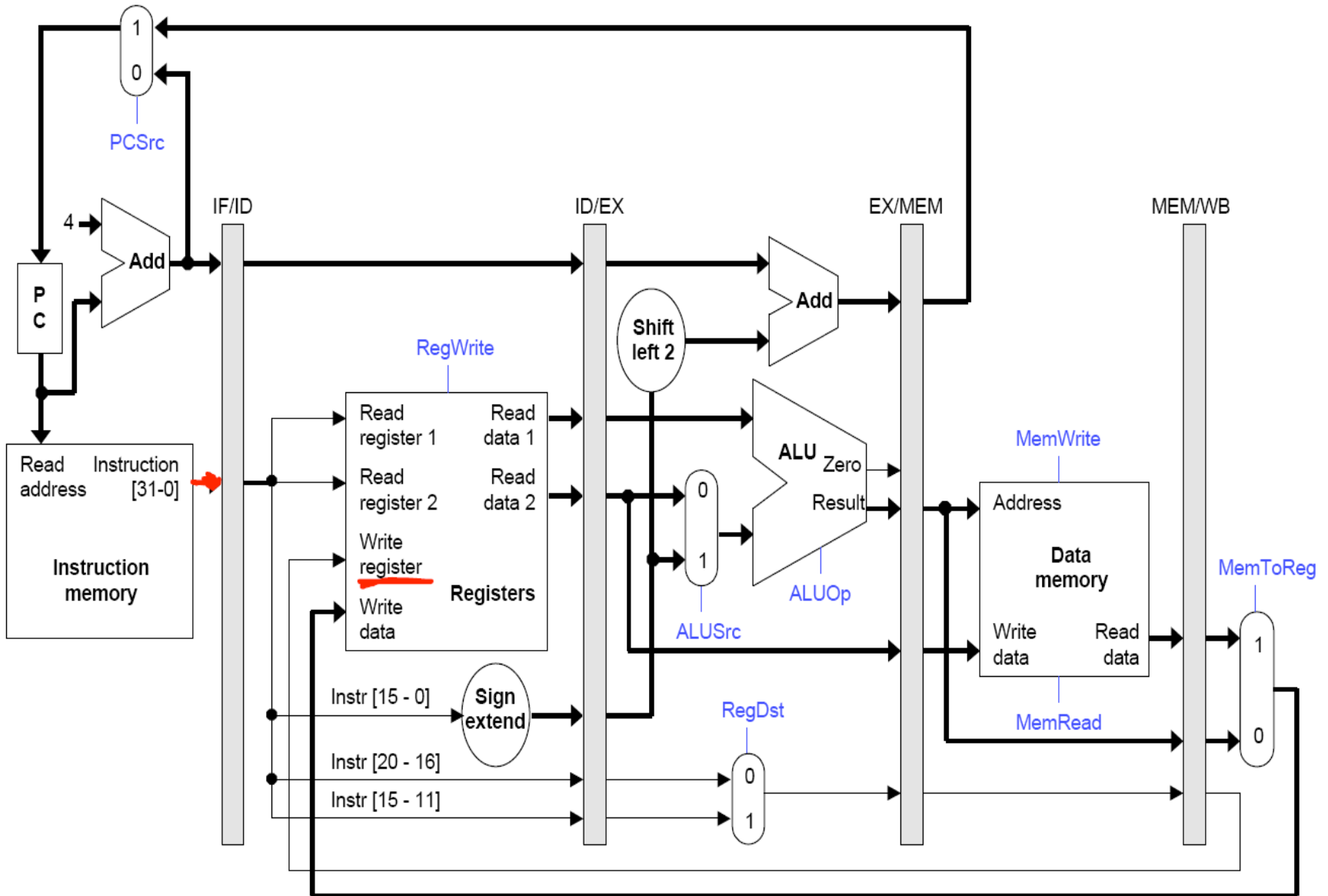
Non-Pipelined



Pipelined



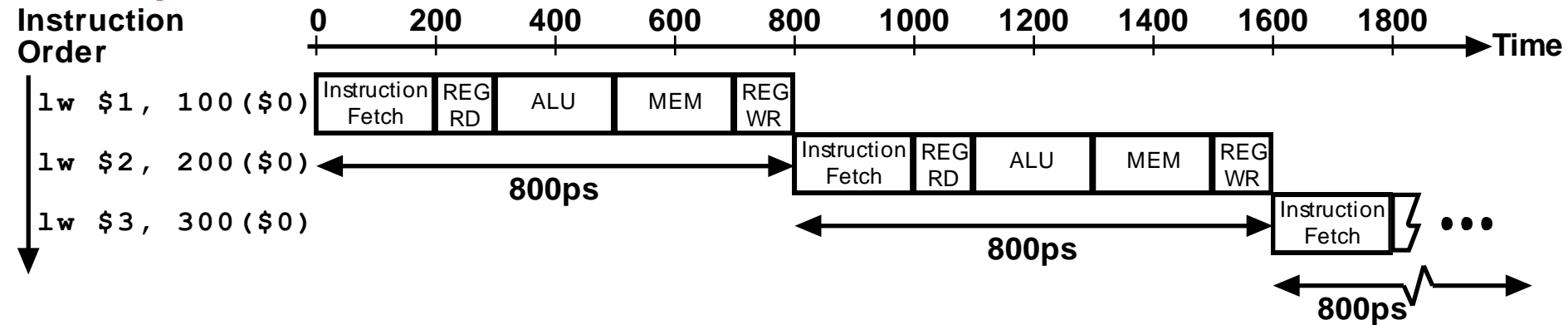
Pipelined datapath



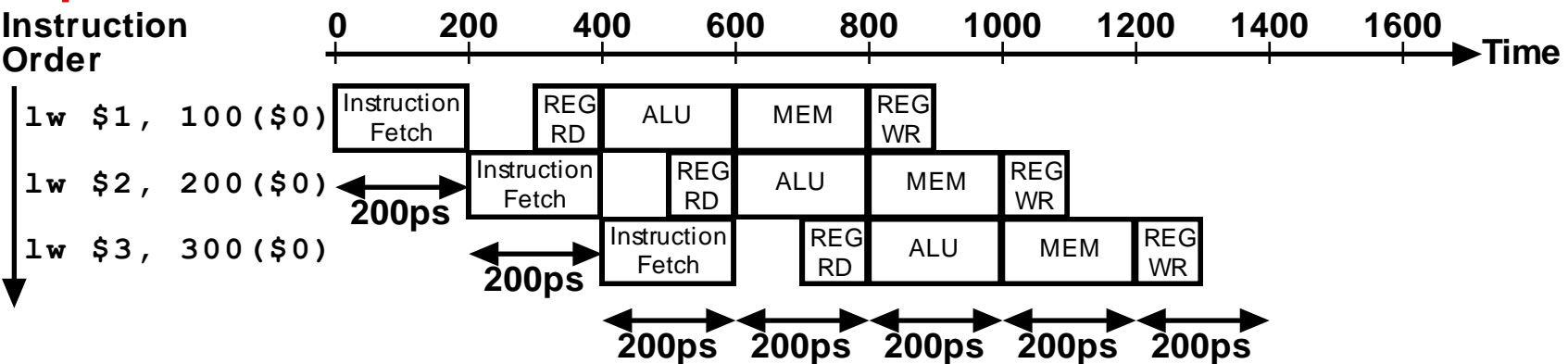
Pipeline: Hazards

Single-Cycle vs. Pipelined Execution

Non-Pipelined



Pipelined



Speedup

- Consider the unpipelined processor introduced previously. Assume that it has a 1 ns clock cycle and it uses 4 cycles for ALU operations and branches, and 5 cycles for memory operations, assume that the relative frequencies of these operations are 40%, 20%, and 40%, respectively. Suppose that due to clock skew and setup, pipelining the processor adds 0.2ns of overhead to the clock. Ignoring any latency impact, how much speedup in the instruction execution rate will we gain from a pipeline?

Average instruction execution time

$$\begin{aligned} &= 1 \text{ ns} * ((40\% + 20\%)*4 + 40\%*5) \\ &= 4.4\text{ns} \end{aligned}$$

Speedup from pipeline

$$\begin{aligned} &= \text{Average instruction time unpiplined} / \text{Average instruction time pipelined} \\ &= 4.4\text{ns} / 1.2\text{ns} = 3.7 \end{aligned}$$

Comments about Pipelining

- The good news
 - Multiple instructions are being processed at same time
 - This works because stages are isolated by registers
 - Best case speedup of N
- The bad news
 - Instructions interfere with each other - hazards
 - Example: different instructions may need the same piece of hardware (e.g., memory) in same clock cycle
 - Example: instruction may require a result produced by an earlier instruction that is not yet complete

Pipeline Hazards

- Limits to pipelining: Hazards prevent next instruction from executing during its designated clock cycle
 - Structural hazards: two different instructions use same h/w in same cycle
 - Data hazards: Instruction depends on result of prior instruction still in the pipeline
 - Control hazards: Pipelining of branches & other instructions that change the PC

Structural Hazards

- Attempt to use same resource twice at same time
- Example: Single Memory for instructions, data
 - Accessed by IF stage
 - Accessed at same time by MEM stage
- Solutions ?
 - Delay second access by one clock cycle
 - Provide separate memories for instructions, data
 - This is what the book does
 - This is called a “**Harvard Architecture**”
 - Real pipelined processors have separate **caches**

Pipelined Example - Executing Multiple Instructions

- Consider the following instruction sequence:

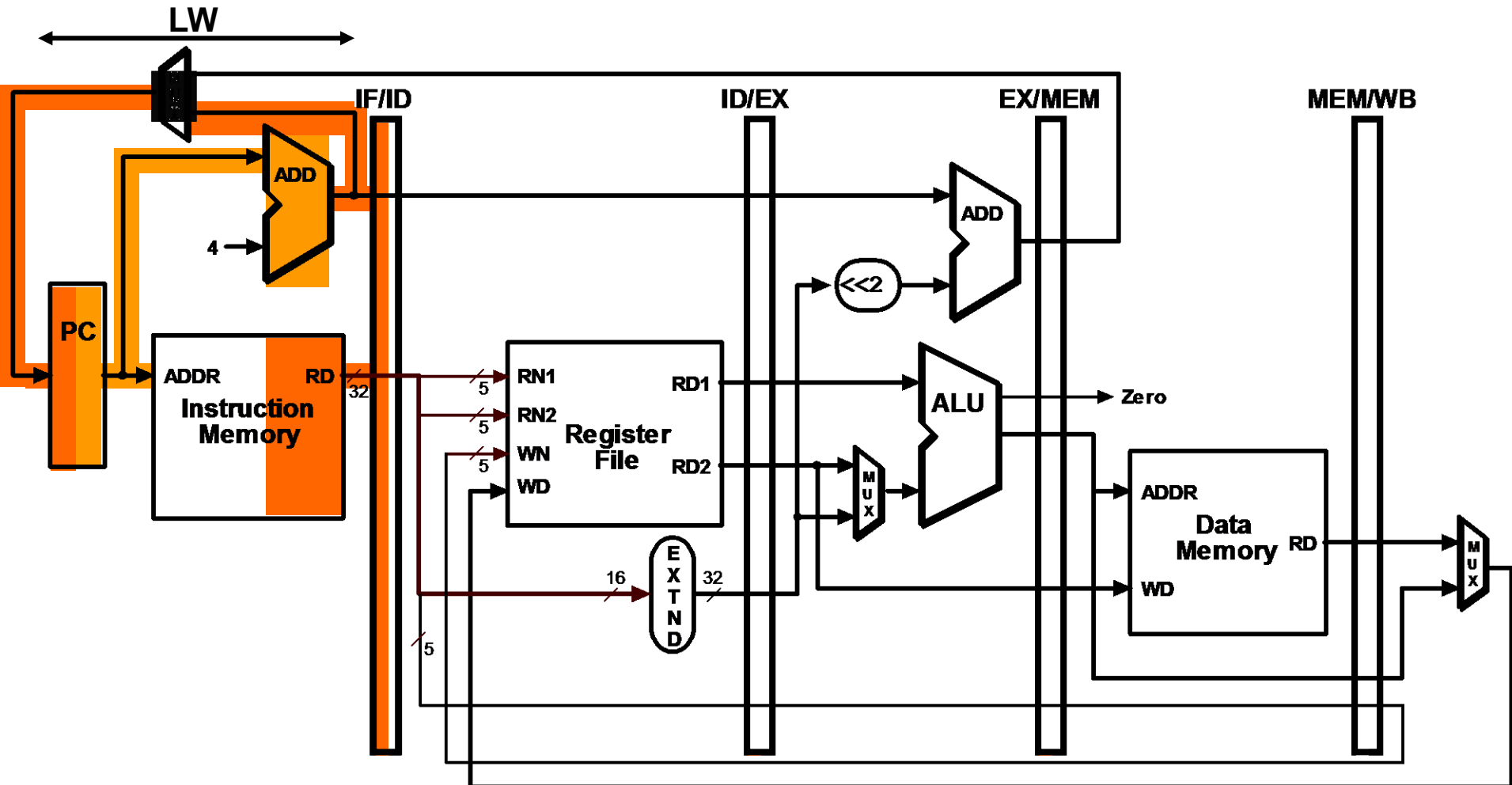
```
lw $r0, 10($r1)
```

```
sw $r3, 20($r4)
```

```
add $r5, $r6, $r7
```

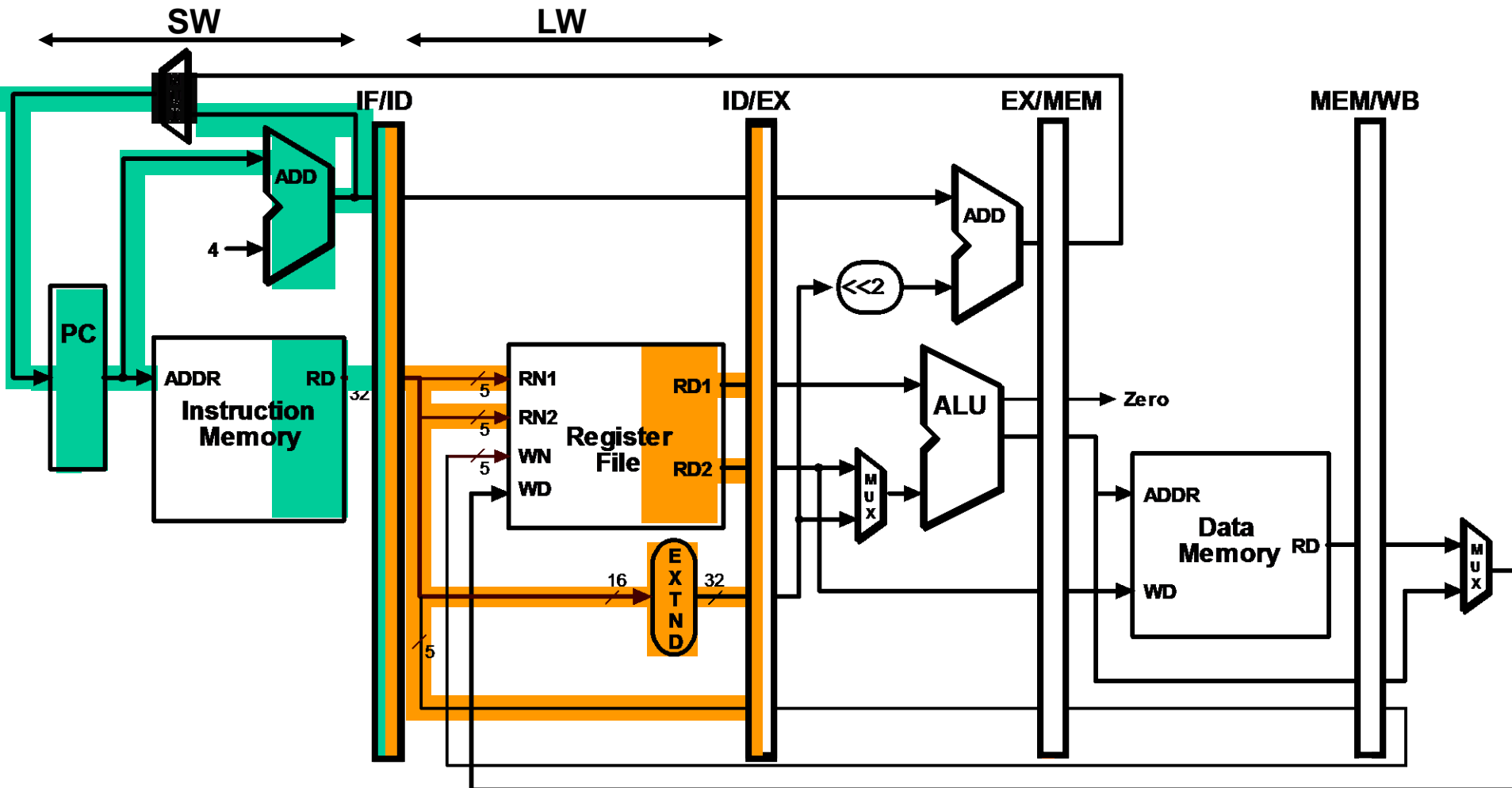
```
sub $r8, $r9, $r10
```

Executing Multiple Instructions Clock Cycle 1

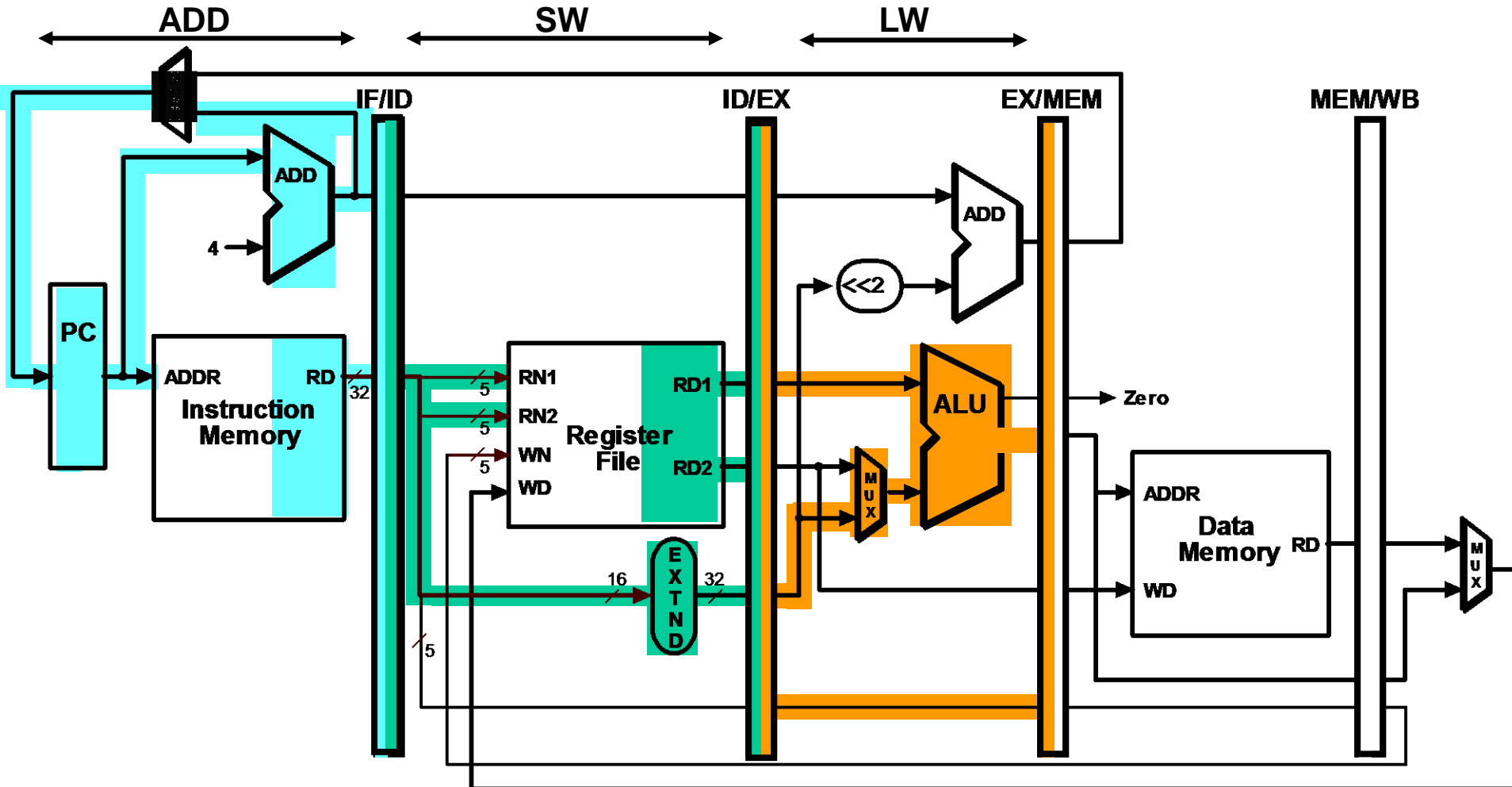


Executing Multiple Instructions

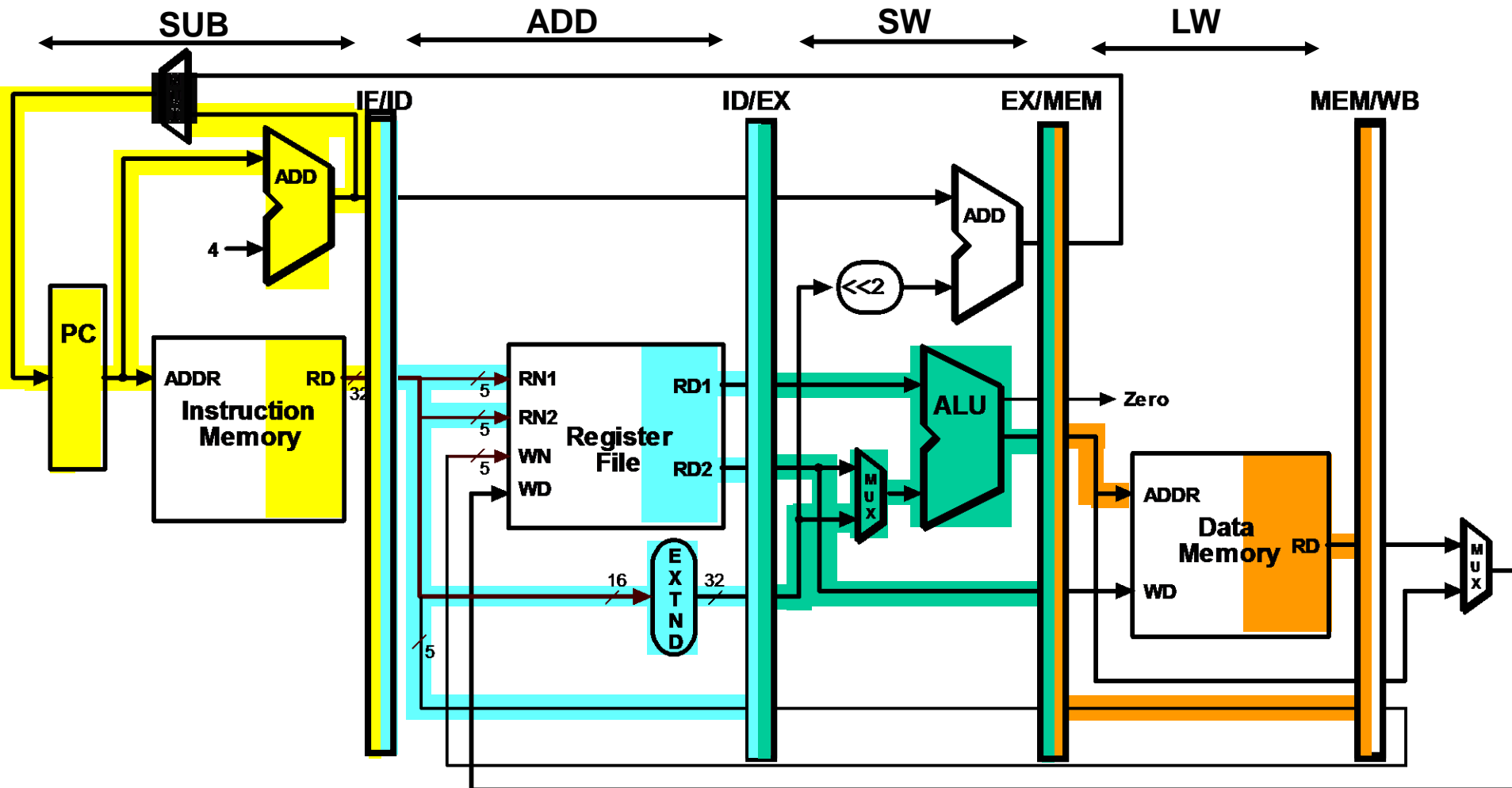
Clock Cycle 2



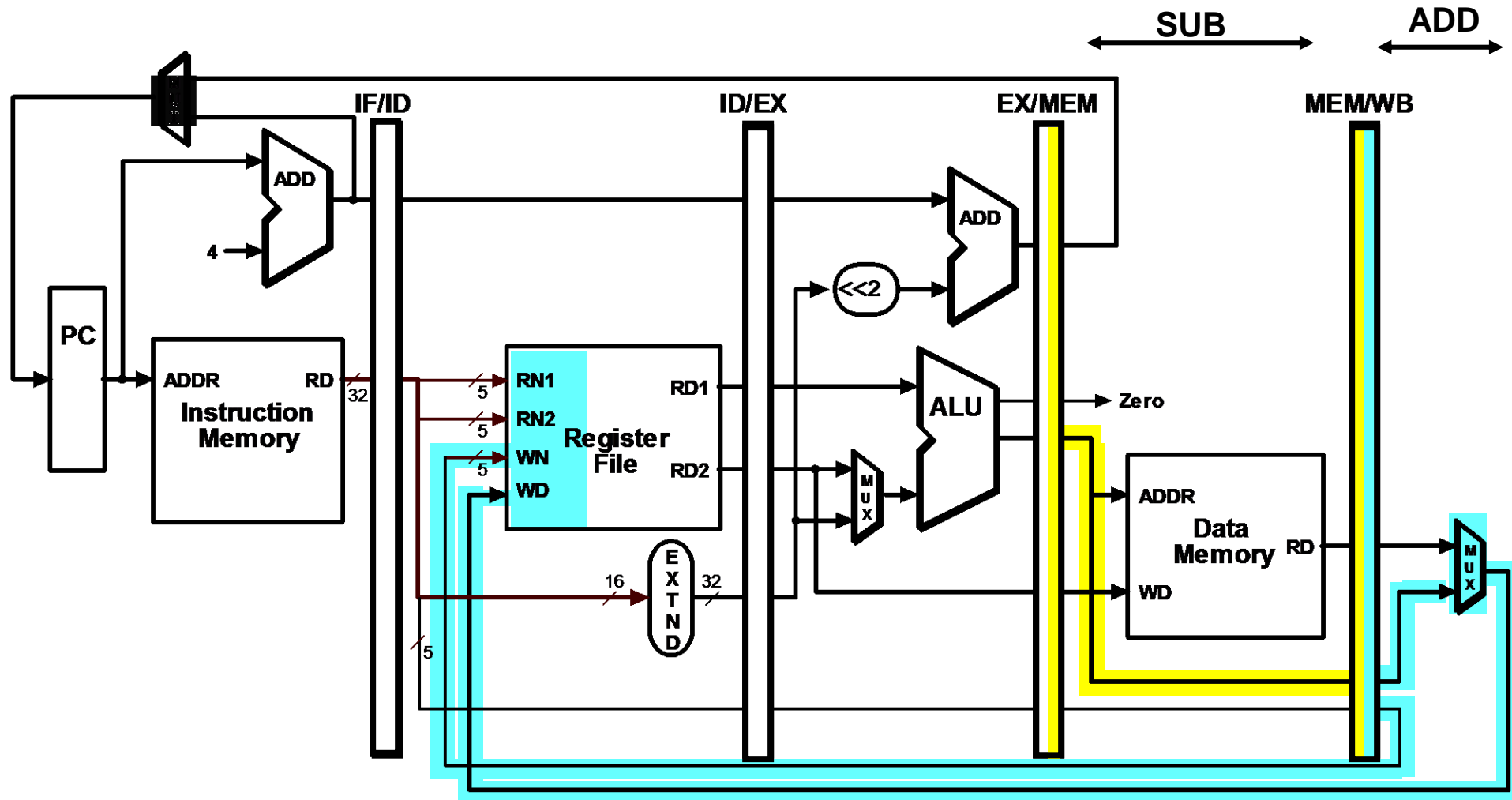
Executing Multiple Instructions Clock Cycle 3



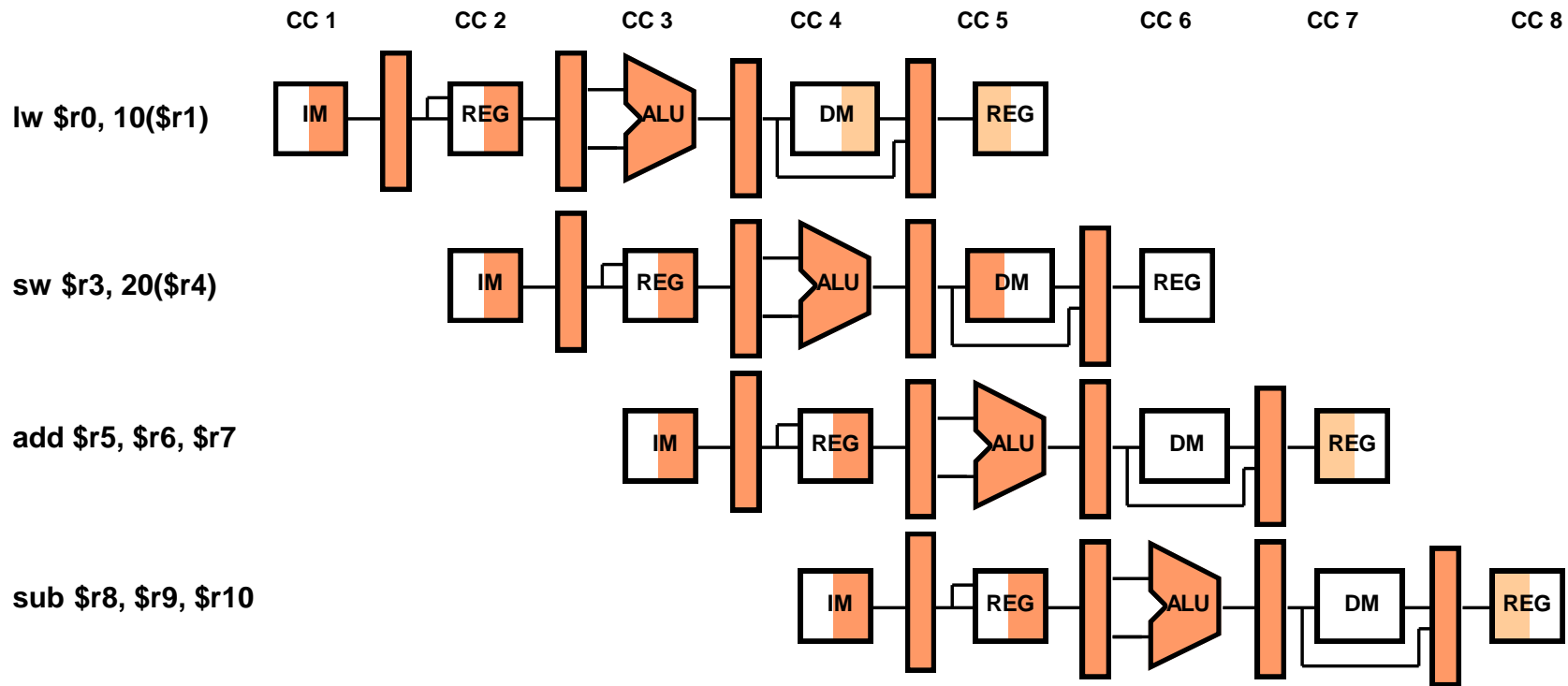
Executing Multiple Instructions Clock Cycle 4



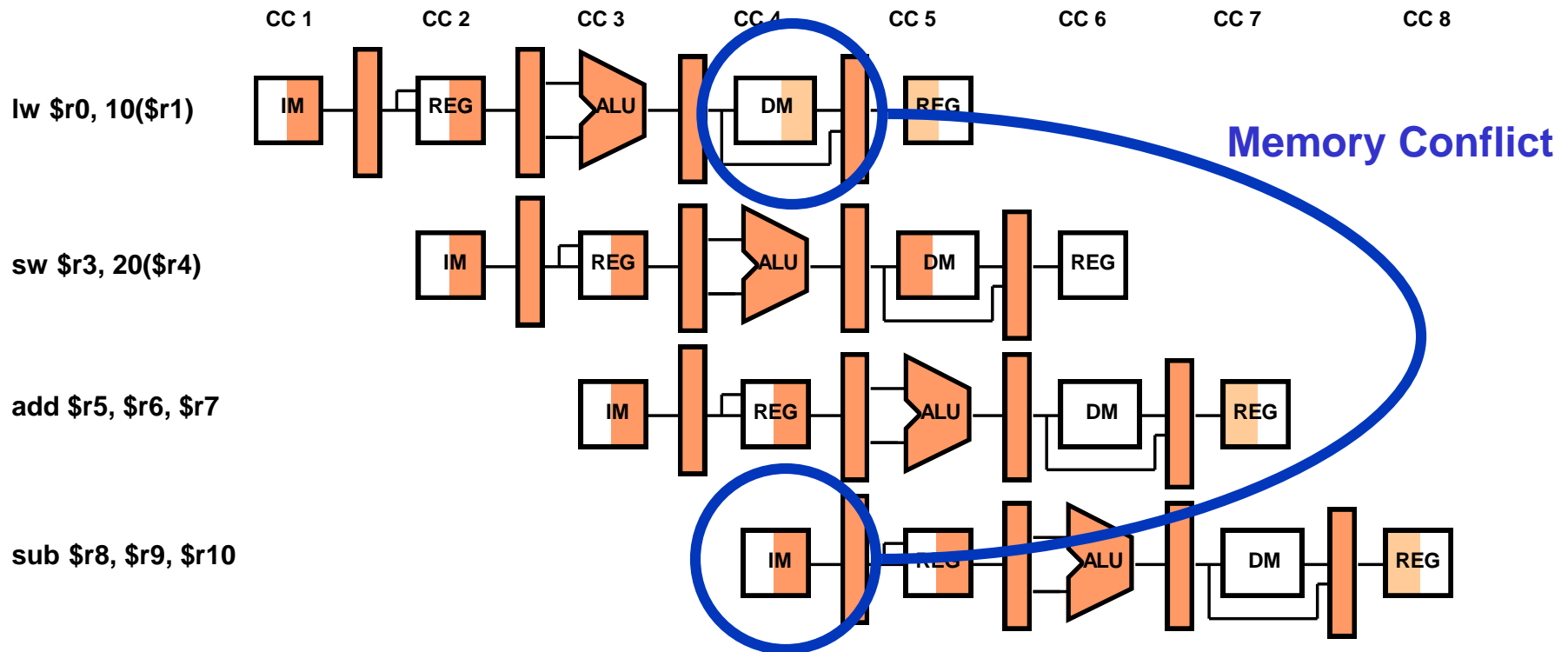
Executing Multiple Instructions Clock Cycle 7



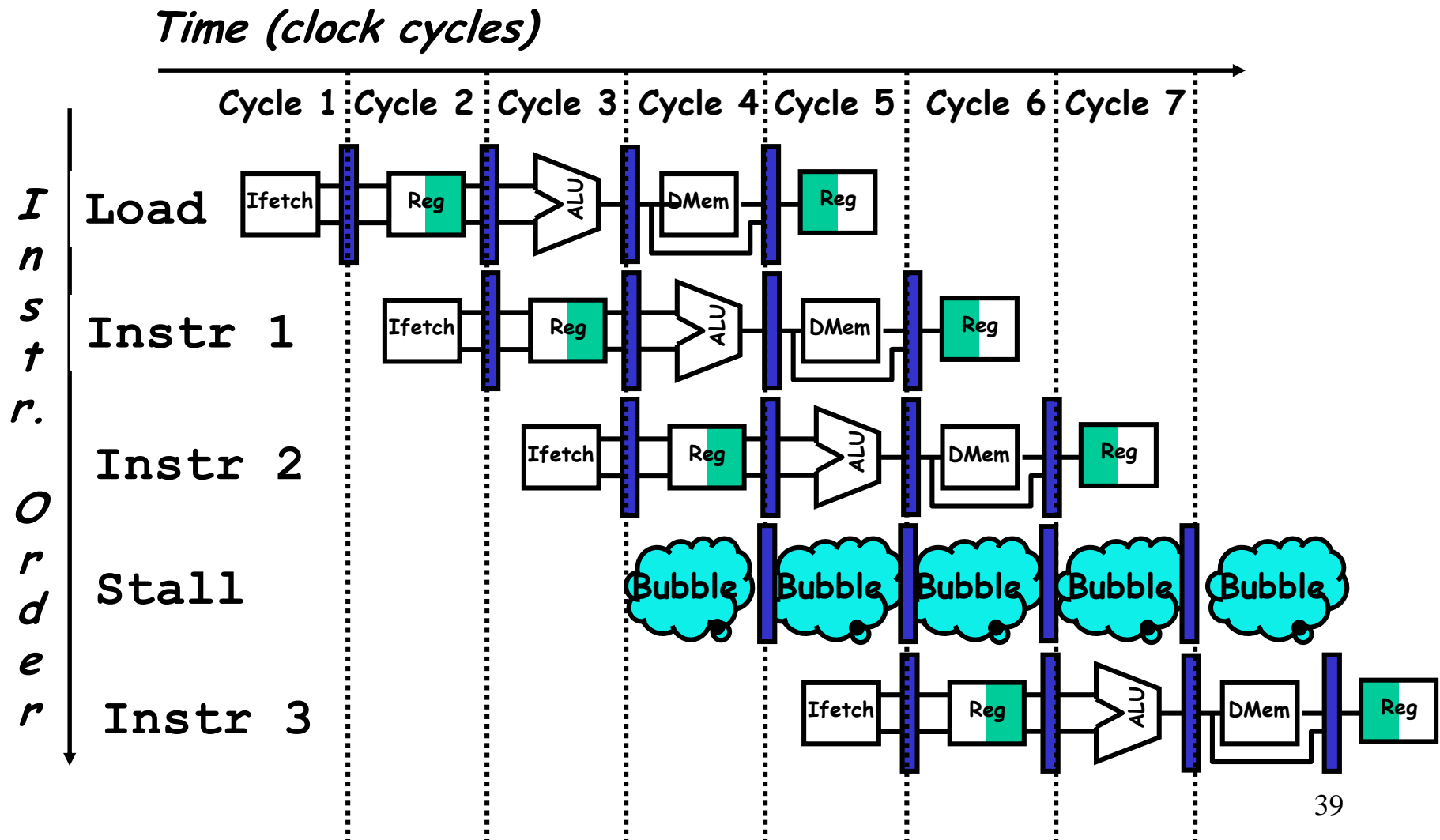
Alternative View - Multicycle Diagram



Alternative View - Multicycle Diagram



One Memory Port Structural Hazards



Structural Hazards

Some Common Structural Hazards:

- Memory:
 - we've already mentioned this one.
- Floating point:
 - Since many floating point instructions require many cycles, it's easy for them to interfere with each other.
- Starting up more of one type of instruction than there are resources.
 - For instance, the PA-8600 can support two ALU + two load/store instructions per cycle - that's how much hardware it has available.

Dealing with Structural Hazards

Stall

- low cost, simple
- Increases CPI
- use for rare case since stalling has performance effect

Pipeline hardware resource

- useful for multi-cycle resources
- good performance
- sometimes complex e.g., RAM

Replicate resource

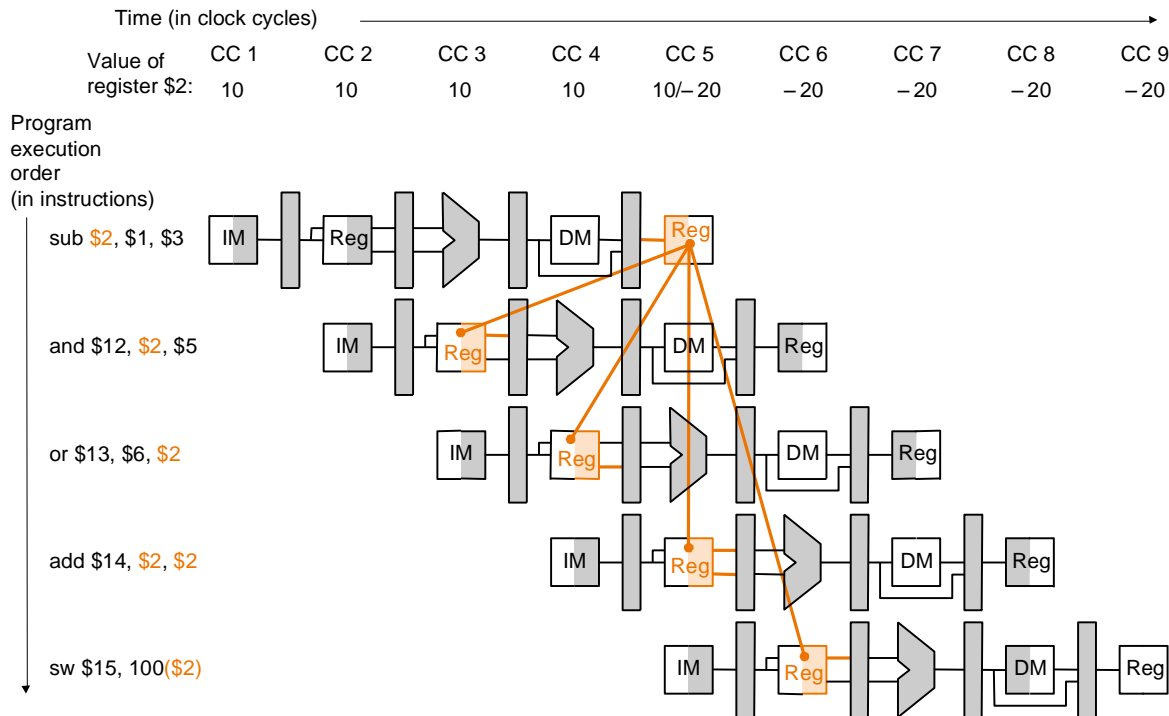
- good performance
- increases cost (+ maybe interconnect delay)
- useful for cheap or divisible resources

Structural Hazards

- Structural hazards are reduced with these rules:
 - Each instruction uses a resource at most once
 - Always use the resource in the same pipeline stage
 - Use the resource for one cycle only
- Many RISC ISA's designed with this in mind
- Sometimes very complex to do this.
 - For example, memory of necessity is used in the IF and MEM stages.

Data Hazards

- Data hazards occur when data is used before it is stored



The use of the result of the SUB instruction in the next two instructions causes a data hazard, since the register is not written until after those instructions read it.

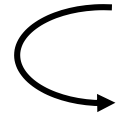
Data Hazards

Execution Order is:

Instr_I
Instr_J

Read After Write (RAW)

Instr_J tries to read operand before Instr_I writes it

 I: **add** **r1**, r2, r3
J: **sub** r4, **r1**, r3

- Caused by a “**Dependence**” (in compiler nomenclature). This hazard results from an actual need for communication.

Data Hazards

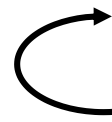
Execution Order is:

Instr_i
Instr_j

Write After Read (WAR)

Instr_j tries to write operand before Instr_i reads it

- Gets wrong operand

 I: **sub** r4, **r1**, r3
J: **add** **r1**, r2, r3
K: **mul** r6, r1, r7

- Called an “**anti-dependence**” by compiler writers. This results from reuse of the name “**r1**”.
- Can’t happen in MIPS 5 stage pipeline because:
 - All instructions take 5 stages, and
 - Reads are always in stage 2, and
 - Writes are always in stage 5

Data Hazards

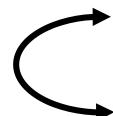
Execution Order is:

Instr_I
Instr_J

Write After Write (WAW)

Instr_J tries to write operand *before* Instr_I writes it

- Leaves wrong result (Instr_I not Instr_J)



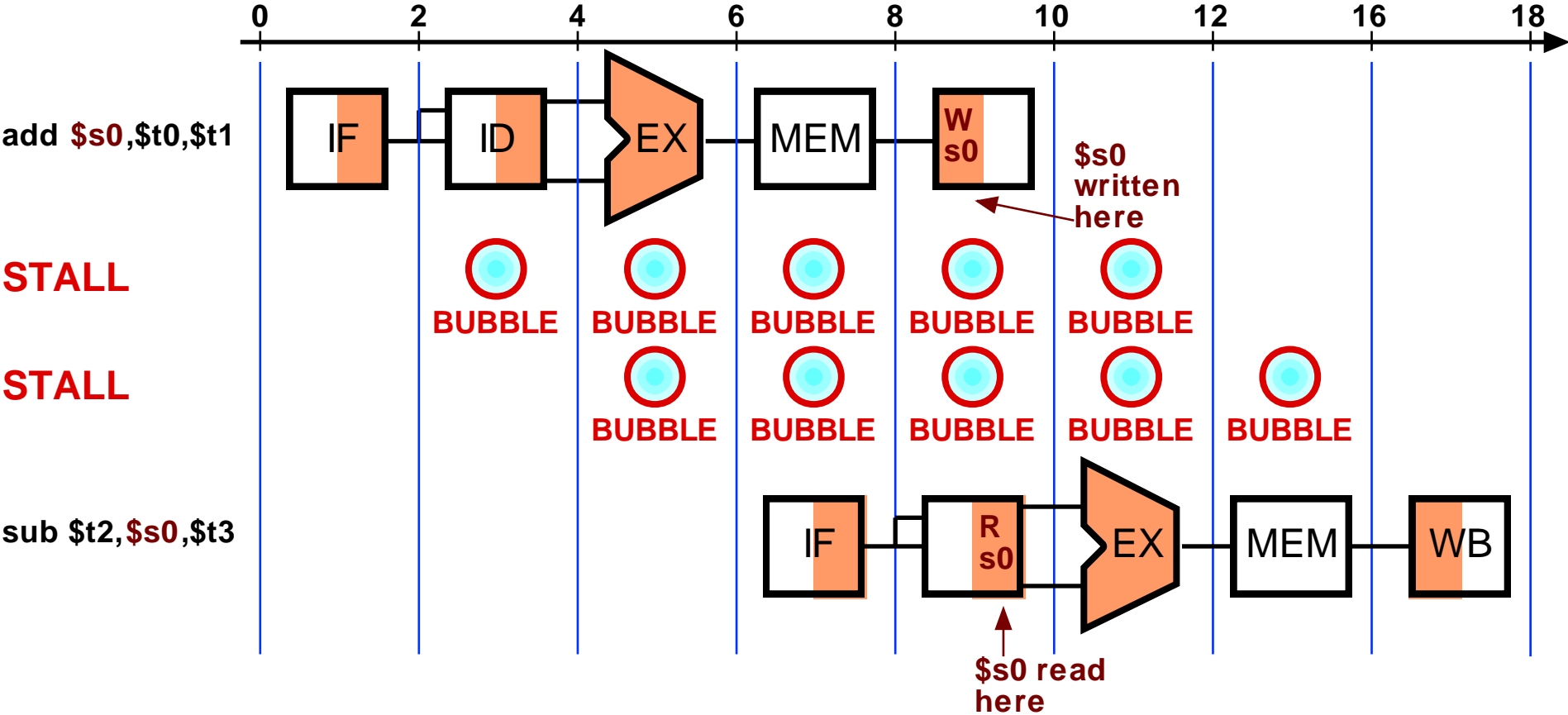
```
I:  sub  r1 , r4 , r3
J:  add  r1 , r2 , r3
K:  mul  r6 , r1 , r7
```

- Called an “**output dependence**” by compiler writers
This also results from the reuse of name “**r1**”.
- Can’t happen in MIPS 5 stage pipeline because:
 - All instructions take 5 stages, and
 - Writes are always in stage 5
- Will see WAR and WAW in later more complicated pipes

Data Hazards

- Solutions for Data Hazards
 - Stalling
 - Forwarding:
 - connect new value directly to next stage
 - Reordering

Data Hazard - Stalling



Data Hazards - Stalling

Simple Solution to RAW

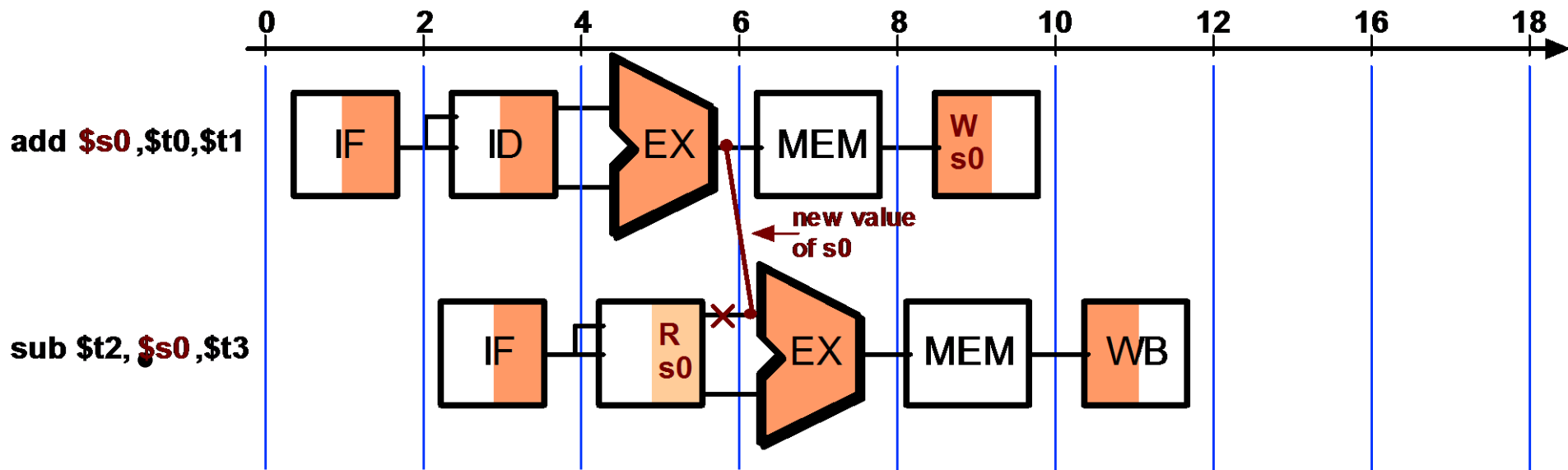
- Hardware detects RAW and stalls
- Assumes register written then read each cycle
 - + low cost to implement, simple
 - reduces IPC
- Try to minimize stalls

Minimizing RAW stalls

- Bypass/forward/shortcircuit (We will use the word “forward”)
- Use data before it is in the register
 - + reduces/avoids stalls
 - complex
- Crucial for common RAW hazards

Data Hazards - Forwarding

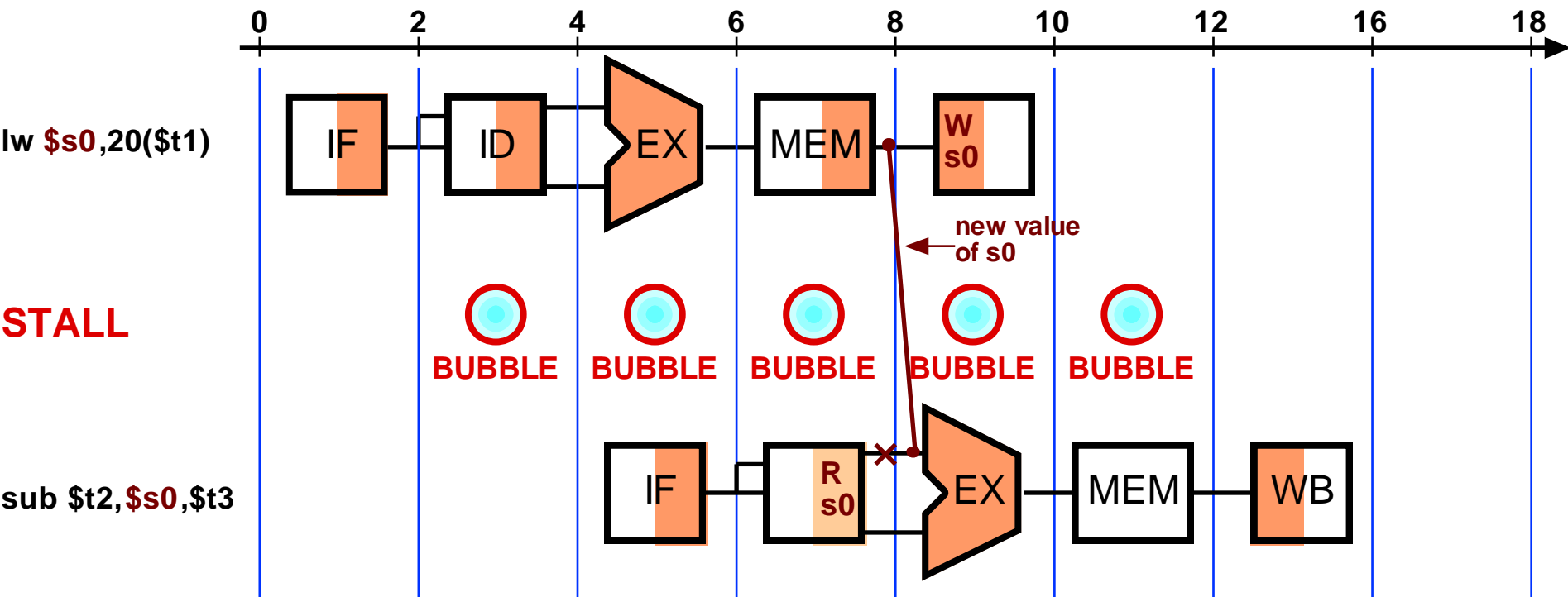
- Key idea: connect new value directly to next stage
- Still read s0, but ignore in favor of new result



Problem: what about load instructions?

Data Hazards - Forwarding

- STALL still required for load - data avail. after MEM
- MIPS architecture calls this delayed load, initial implementations required compiler to deal with this



Data Hazards

This is another representation of the stall.

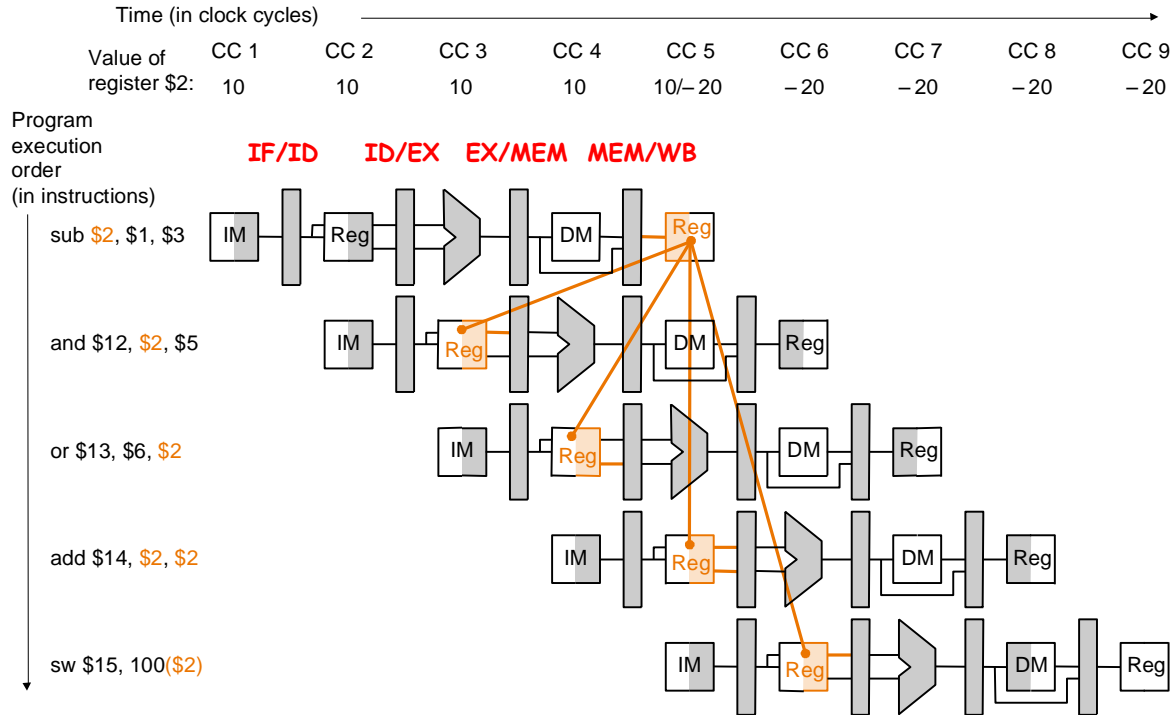
LW	R1, 0(R2)	IF	ID	EX	MEM	WB			
SUB	R4, R1, R5		IF	ID	EX	MEM	WB		
AND	R6, R1, R7			IF	ID	EX	MEM	WB	
OR	R8, R1, R9				IF	ID	EX	MEM	WB

LW	R1, 0(R2)	IF	ID	EX	MEM	WB				
SUB	R4, R1, R5		IF	ID	stall	EX	MEM	WB		
AND	R6, R1, R7			IF	stall	ID	EX	MEM	WB	
OR	R8, R1, R9				stall?	IF	ID	EX	MEM	WB

Why “OR R8, R1, R9” needs to be stalled in “IF” stage? This is because if not, there will be two instructions (itself and “AND R6, R1, R7”) both reading register R1 in the same cycle, which is not allowed. For a register, two instructions can simultaneously access it but it must be “one read and one write”, rather than, two writes.

Forwarding

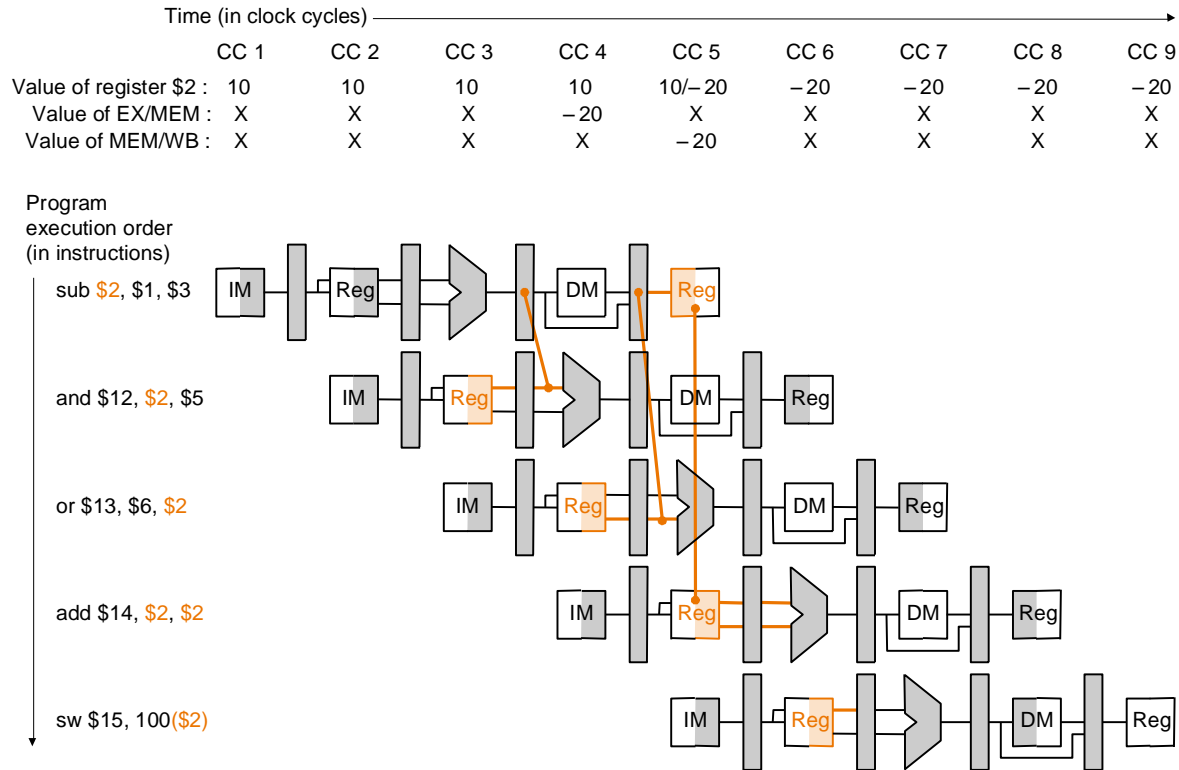
Key idea: connect data internally before it's stored



How would you design the forwarding?

Data Hazard Solution: Forwarding

- Key idea: connect data internally before it's stored



Assumption:

- The register file forwards values that are read and written during the same cycle.

Data Hazard Summary

- Three types of data hazards
 - RAW (MIPS)
 - WAW (not in MIPS)
 - WAR (not in MIPS)
- Solution to RAW in MIPS
 - Stall
 - Forwarding
 - Detection & Control
 - A stall is needed if read a register after a load instruction that writes the same register.
 - Reordering

Control Hazards

A *control hazard* is when we need to find the destination of a branch, and can't fetch any new instructions until we know that destination.

A branch is either

- **Taken:** $PC \leq PC + 4 + Imm$
- **Not Taken:** $PC \leq PC + 4$

Control Hazards

Control Hazard on Branches

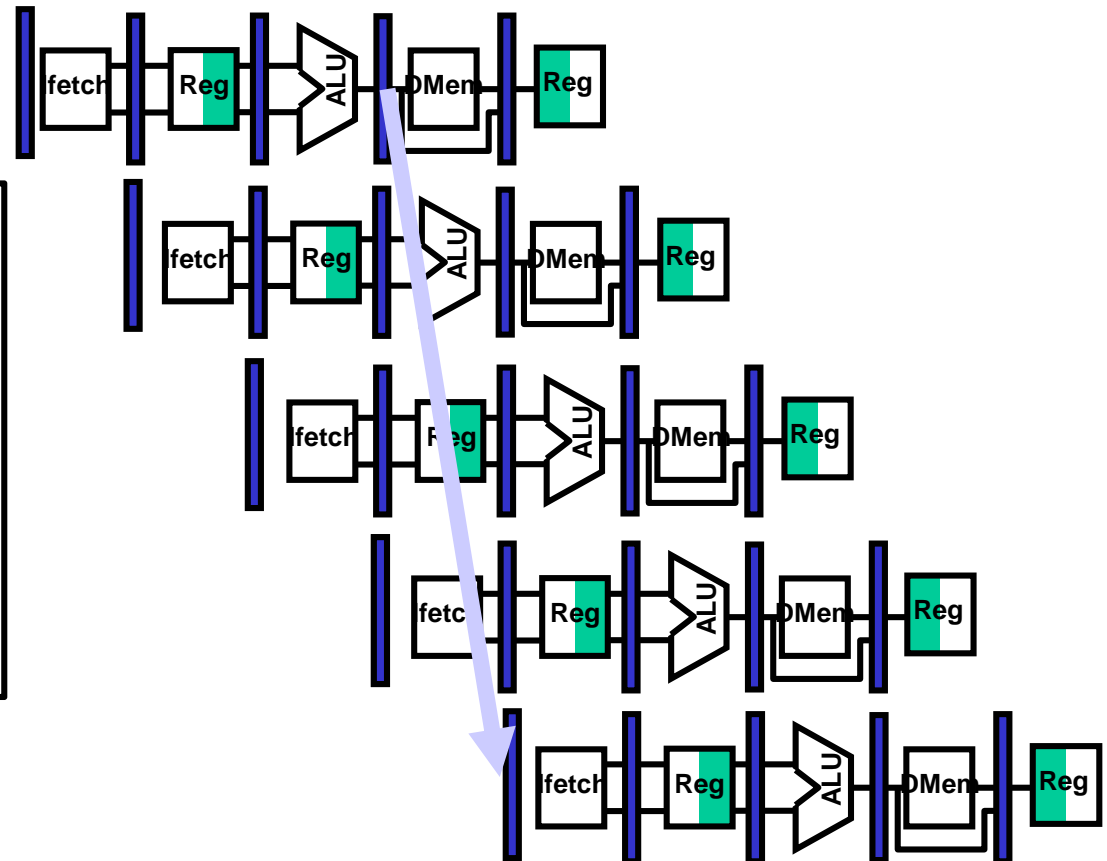
10: **beq r1,r3,36**

14: and r2,r3,r5

18: or r6,r1,r7

22: add r8,r1,r9

36: xor r10,r1,r11

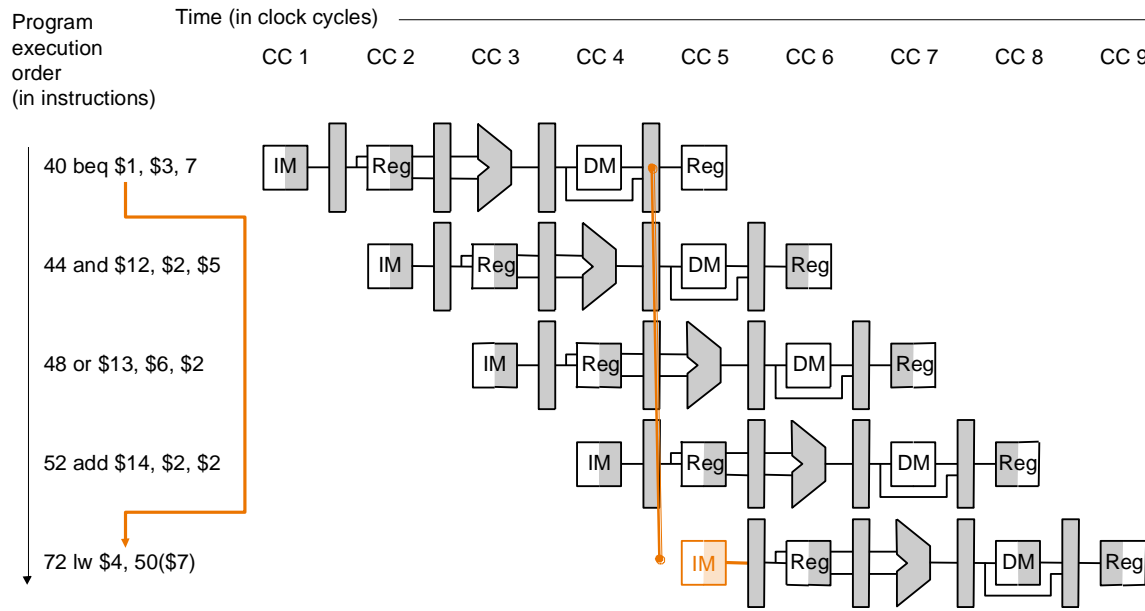


The penalty when branch taken is 3 cycles!

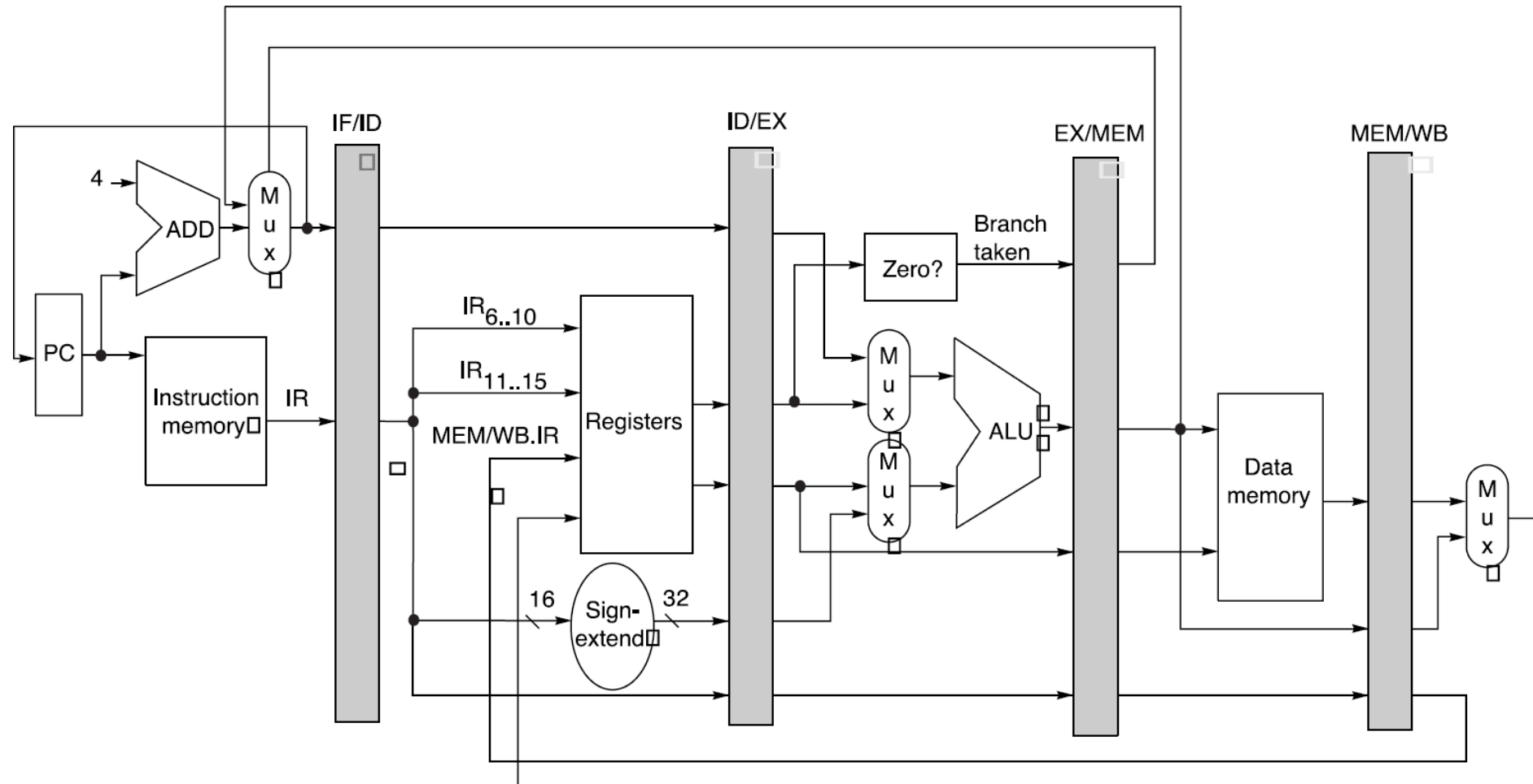
Three Stage Stall

Branch Hazards

- Just stalling for each branch is not practical
Common assumption: branch not taken
- When assumption fails: flush three instructions



Basic Pipelined Processor

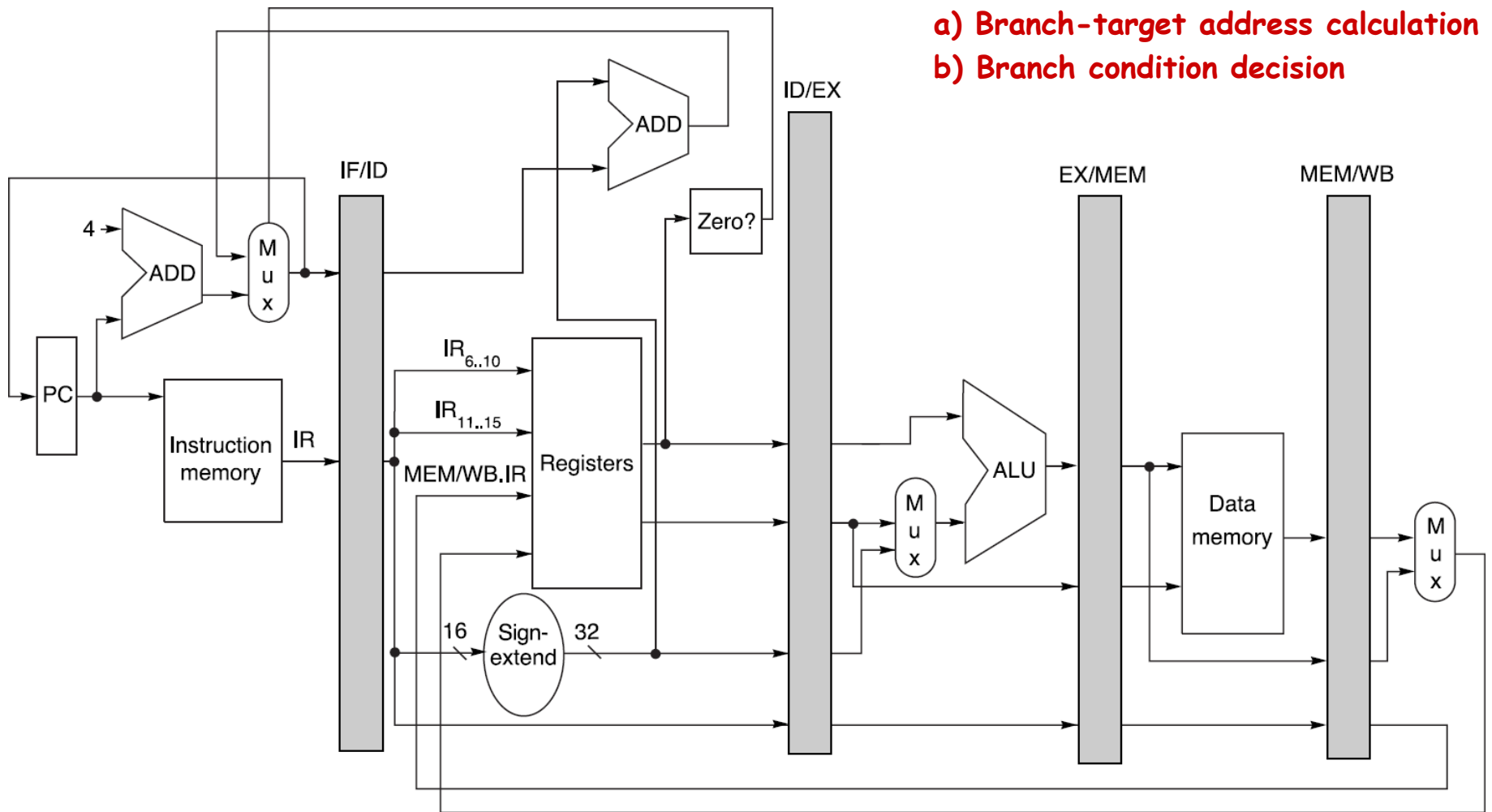


In our original Design, branches have a penalty of 3 cycles

Reducing Branch Delay

Move following to ID stage

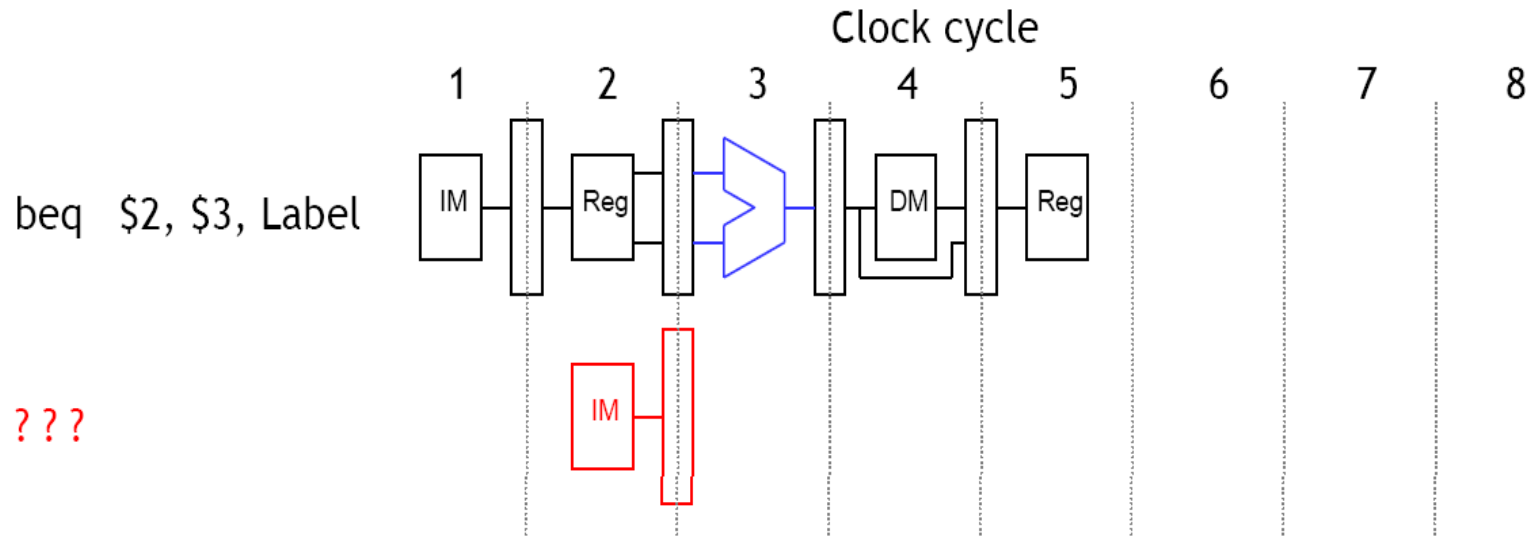
- a) Branch-target address calculation
- b) Branch condition decision



Reduced penalty (1 cycle) when branch take!

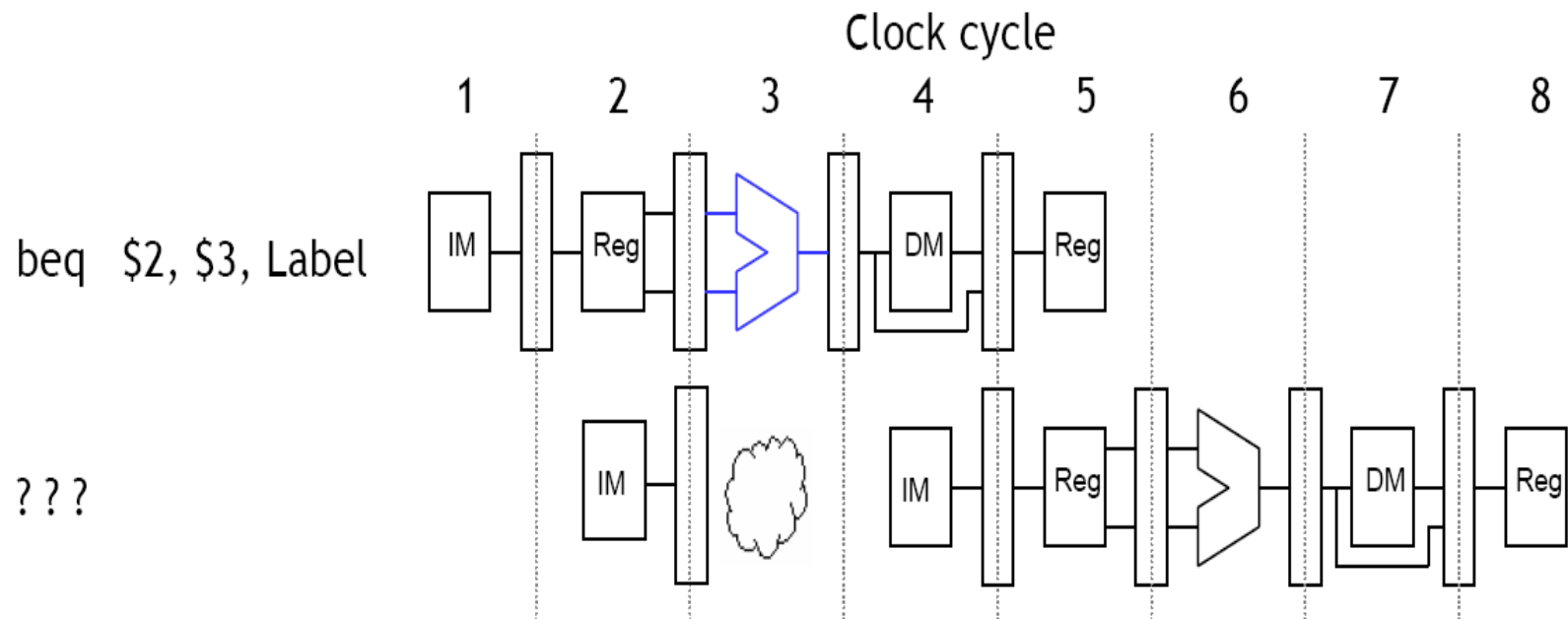
Branches

- 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

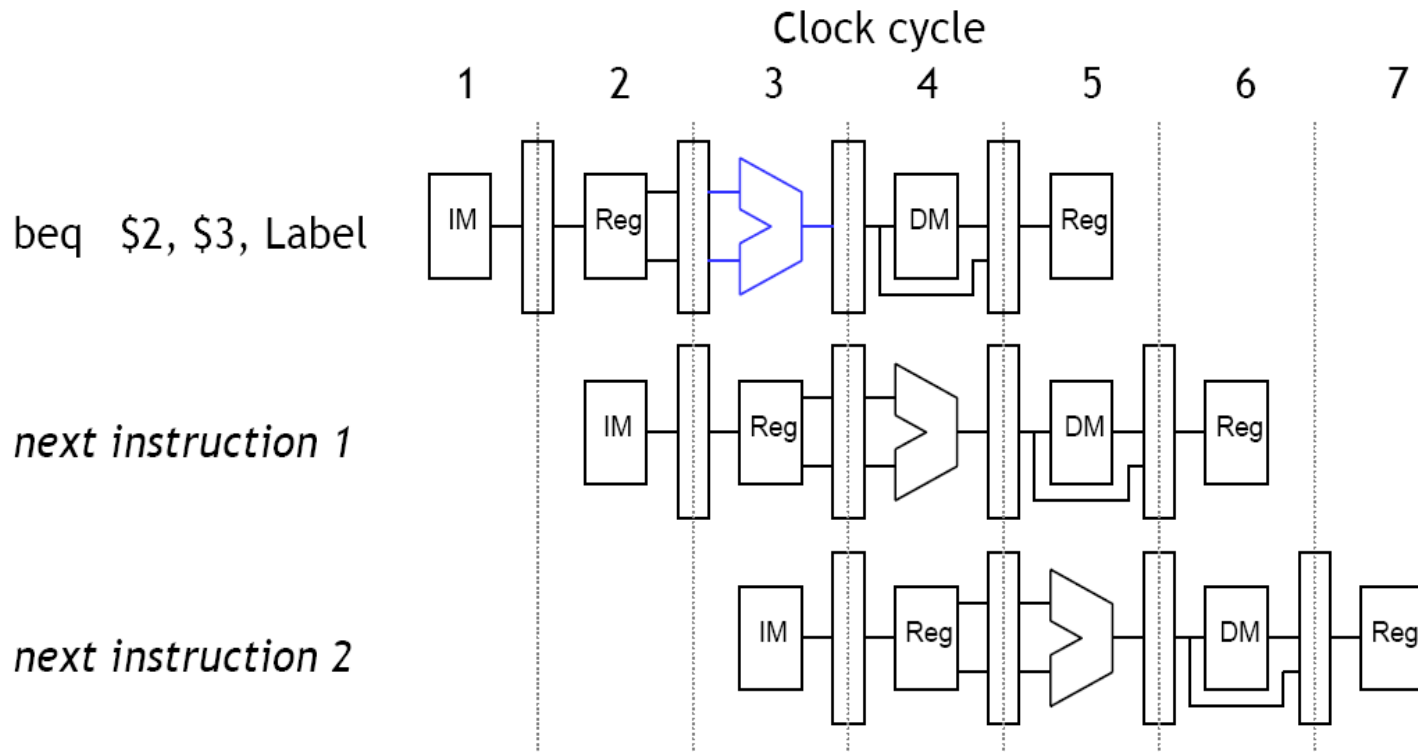
- Again, stalling is always one possible solution.



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

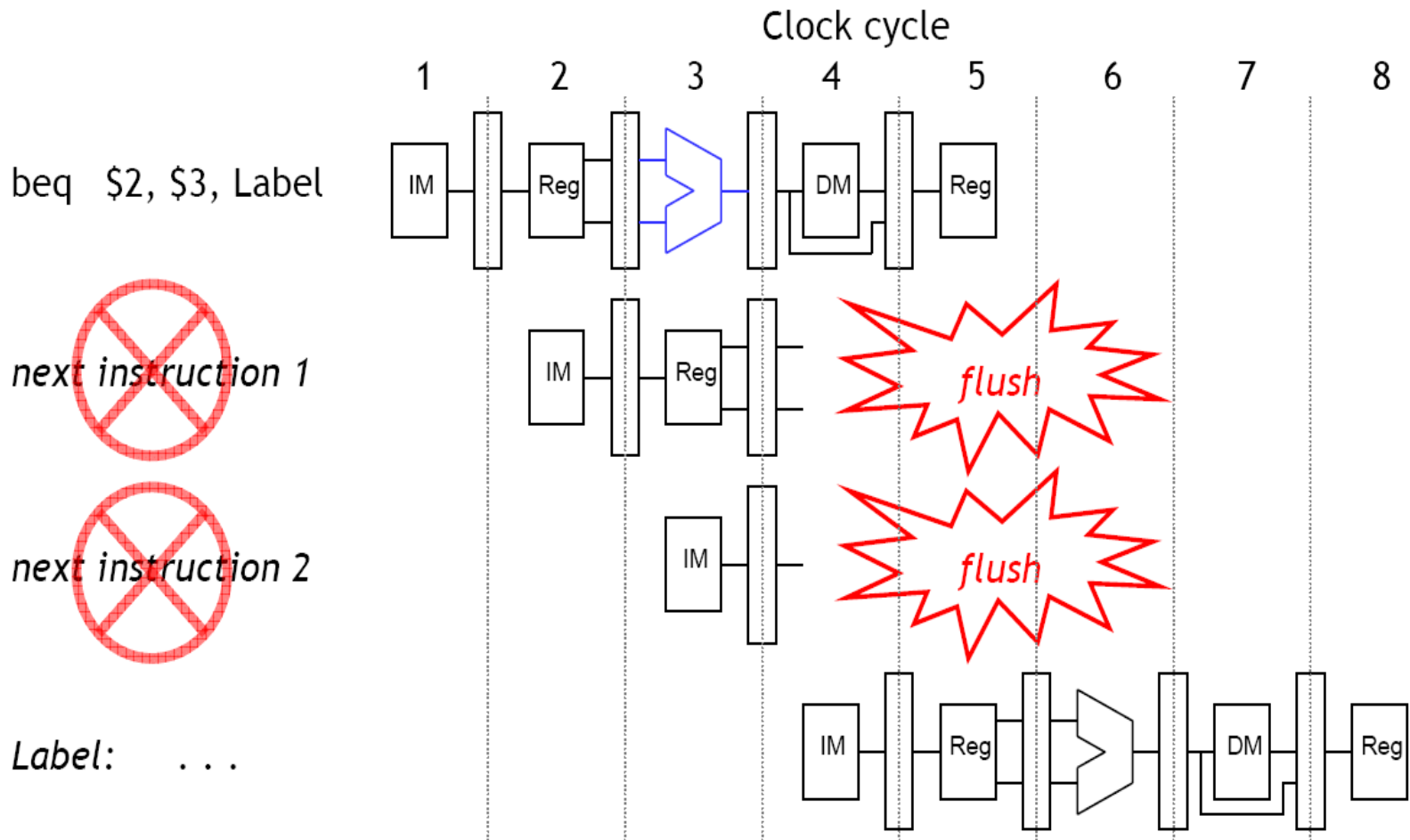
Branch prediction

- 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.

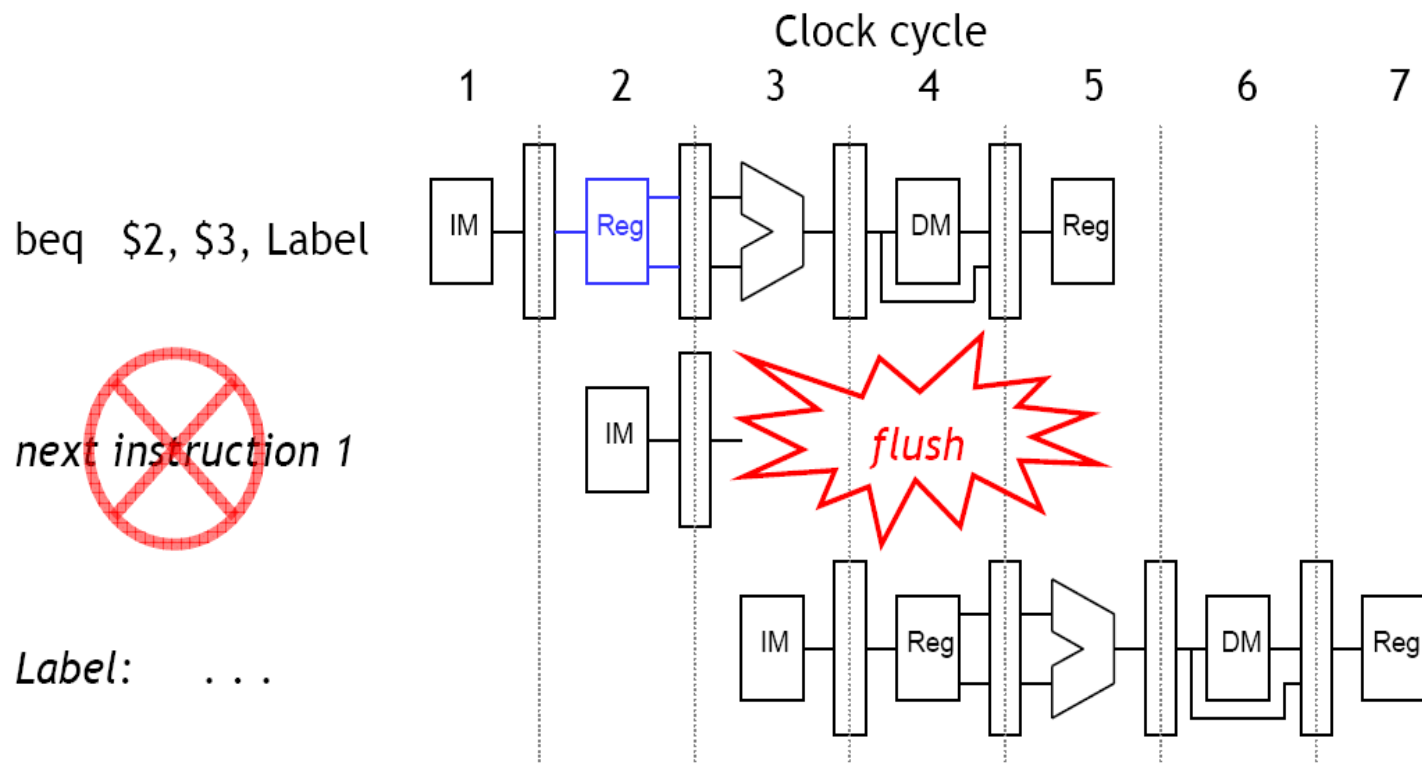


Performance gains and losses

- 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.

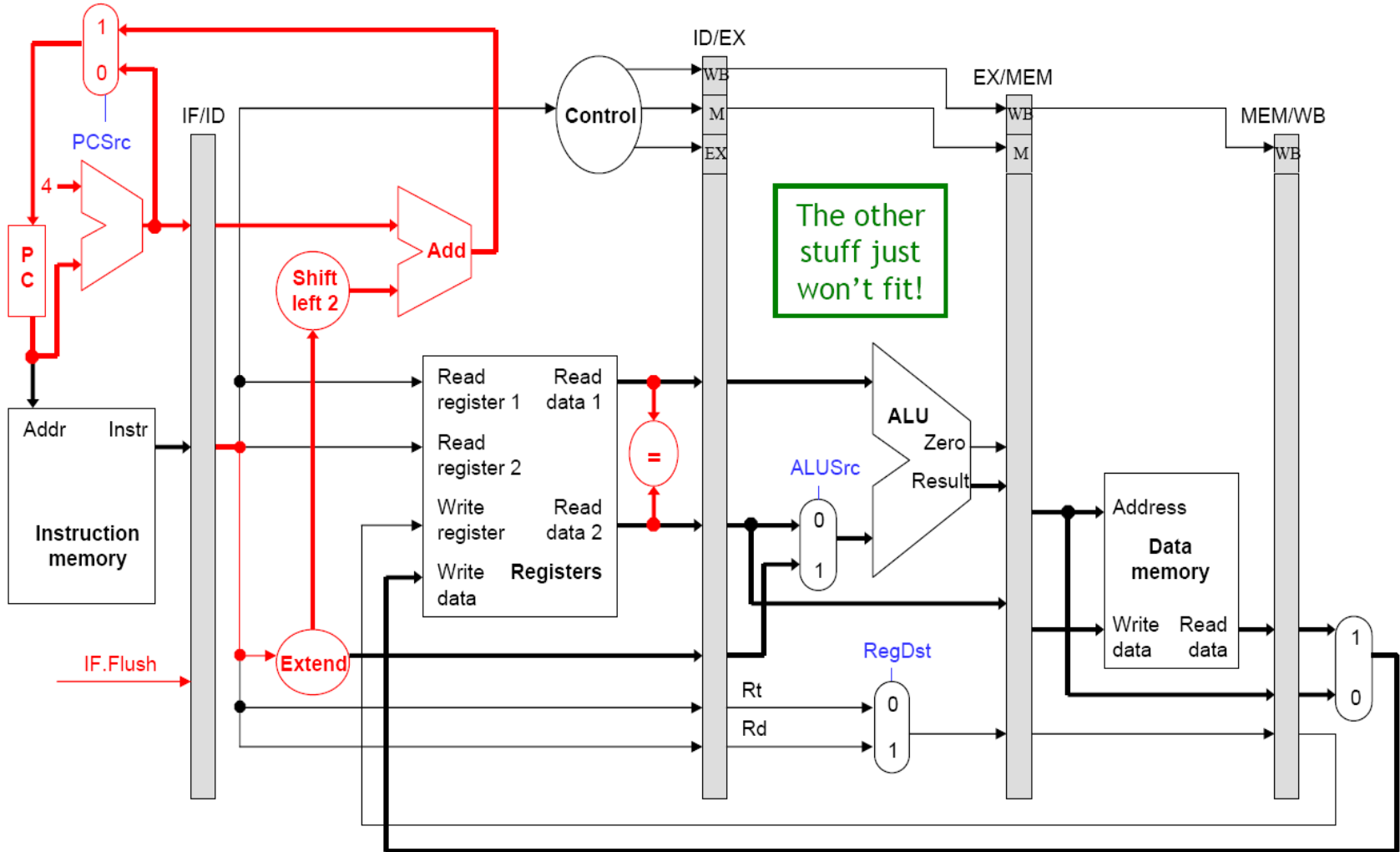


Implementing flushes

- 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



Branch Behavior in Programs

- Based on SPEC benchmarks on DLX
 - Branches occur with a frequency of 14% to 16% in integer programs and 3% to 12% in floating point programs.
 - About 75% of the branches are forward branches
 - 60% of forward branches are taken
 - 80% of backward branches are taken
 - 67% of all branches are taken
- Why are branches (especially backward branches) more likely to be taken than not taken?

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.