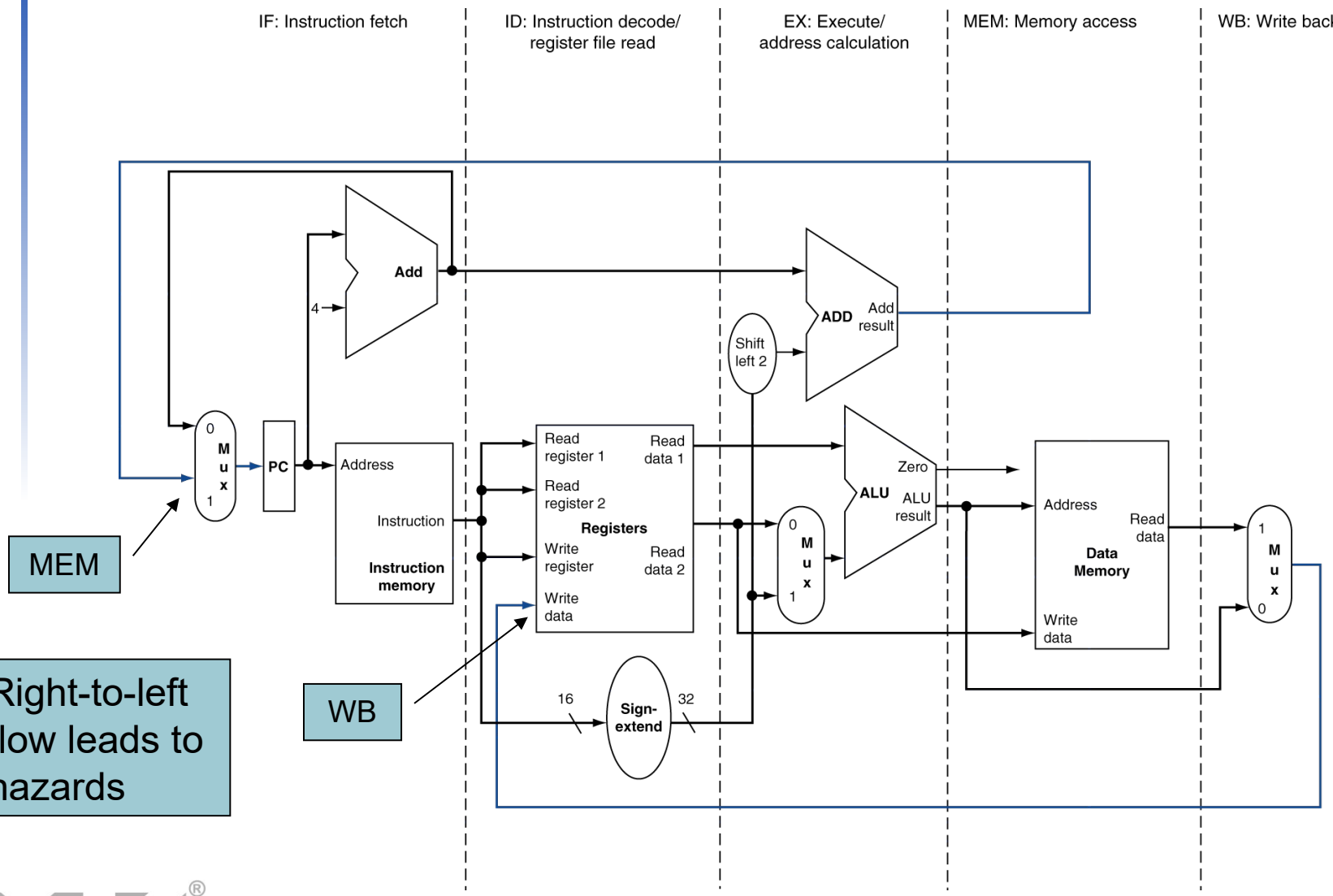
Chapter 4

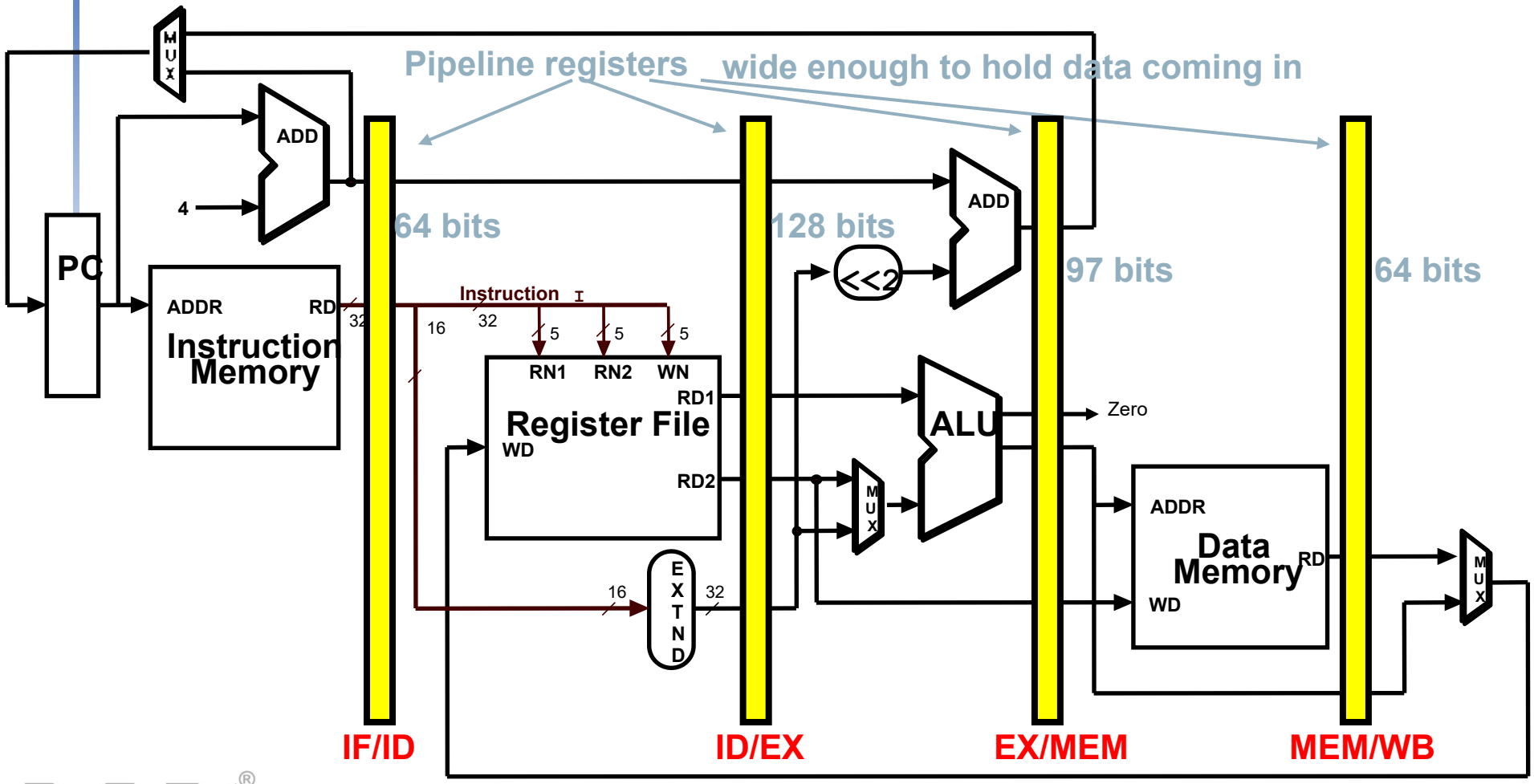
Pipelining Hazards

MIPS Pipelined Datapath



Right-to-left flow leads to hazards

Pipelined Datapath

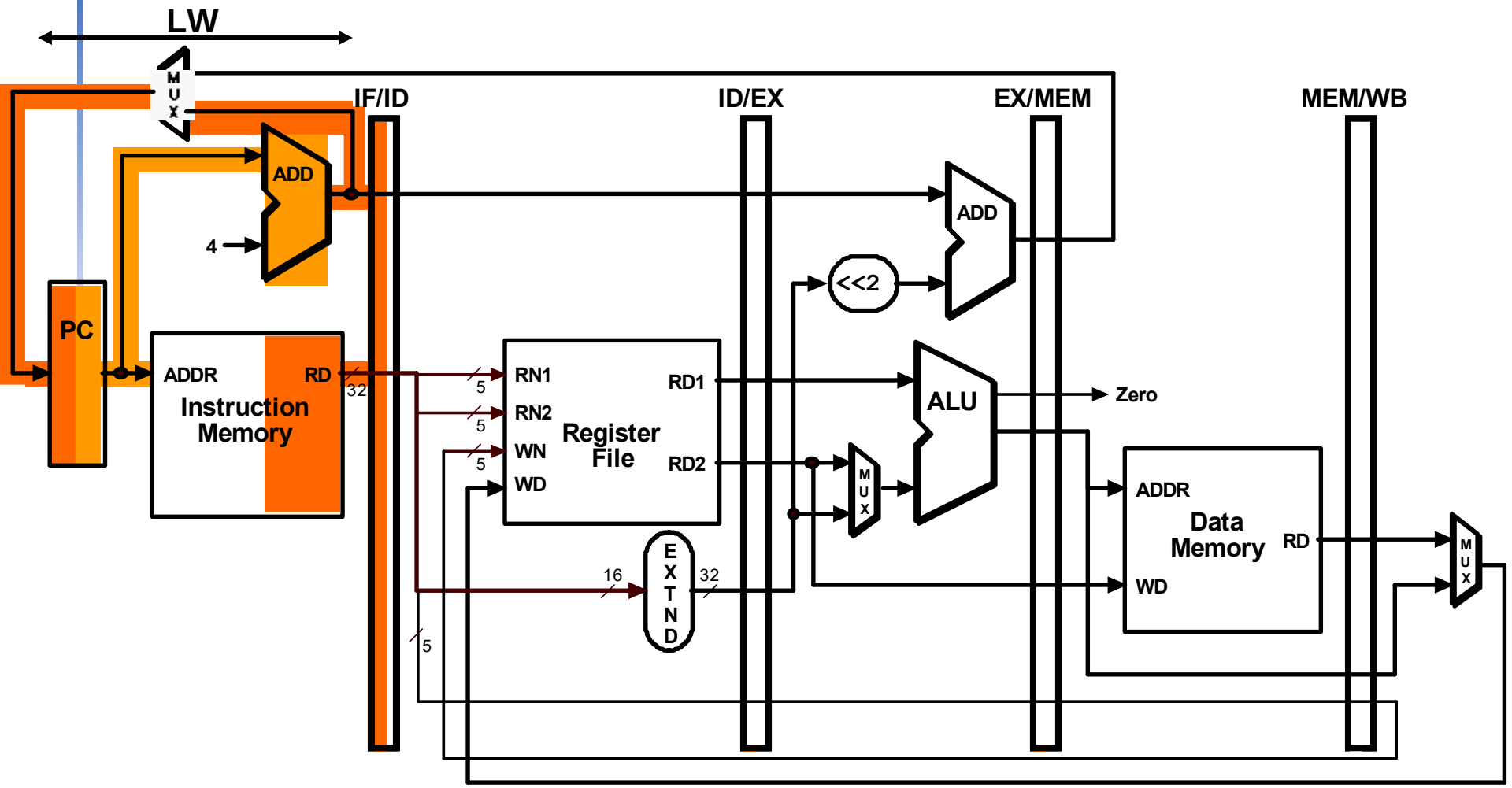


Pipelined Example

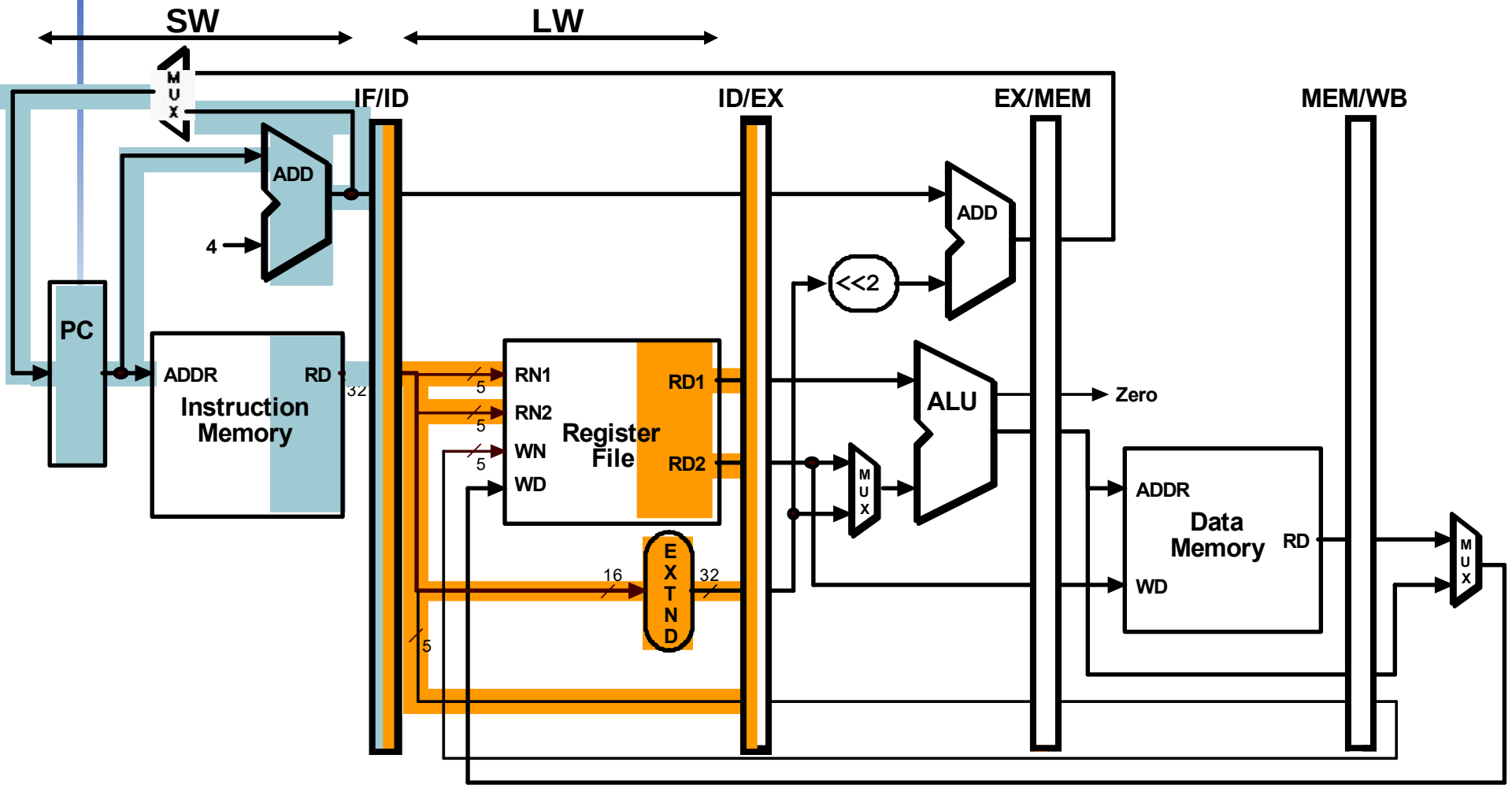
- Consider the following instruction sequence:

```
lw  $t0,  10($t1)
sw  $t3, 20($t4)
add $t5,  $t6,  $t7
sub $t8,  $t9,  $t10
```

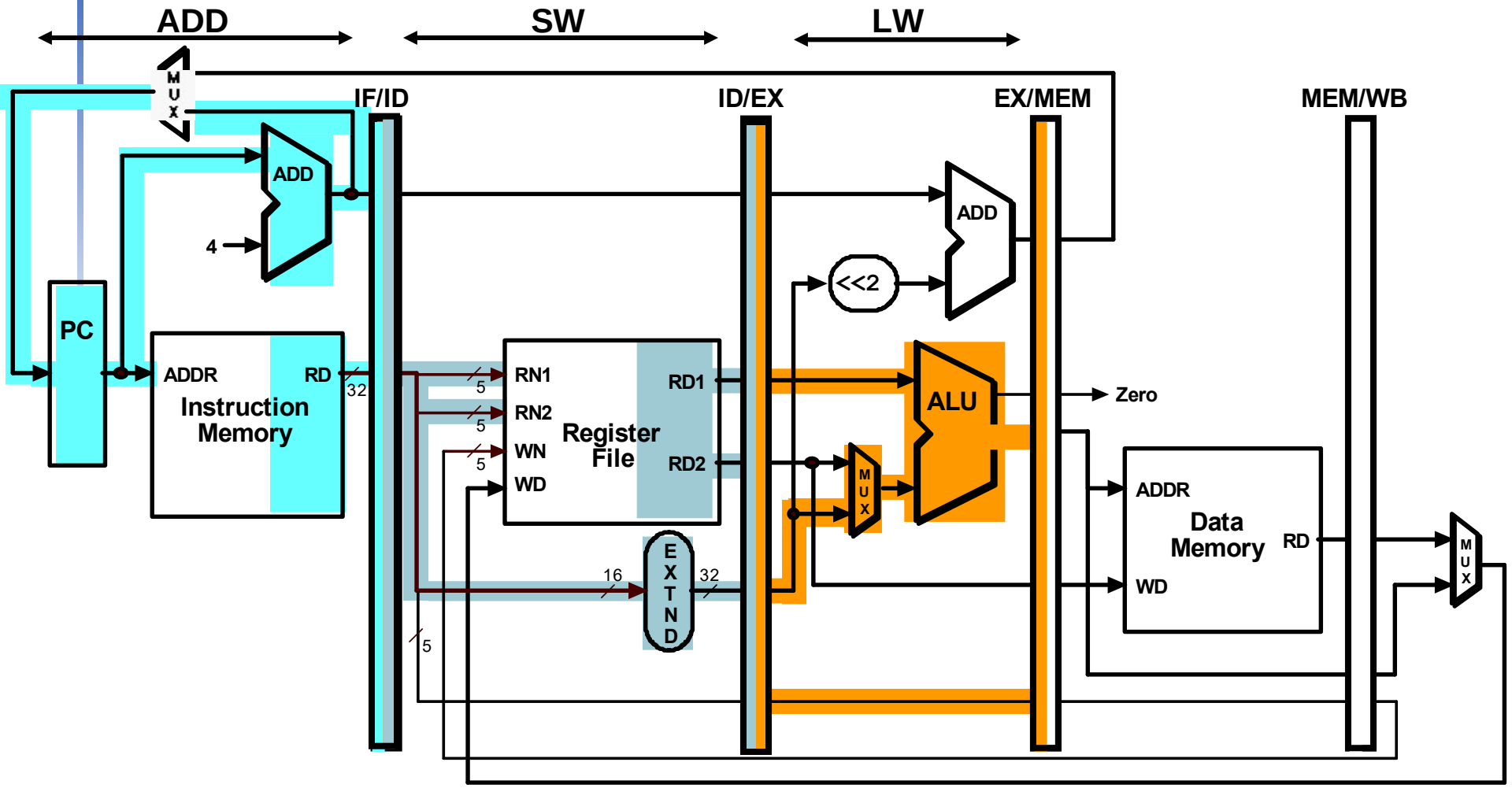
Clock Cycle 1



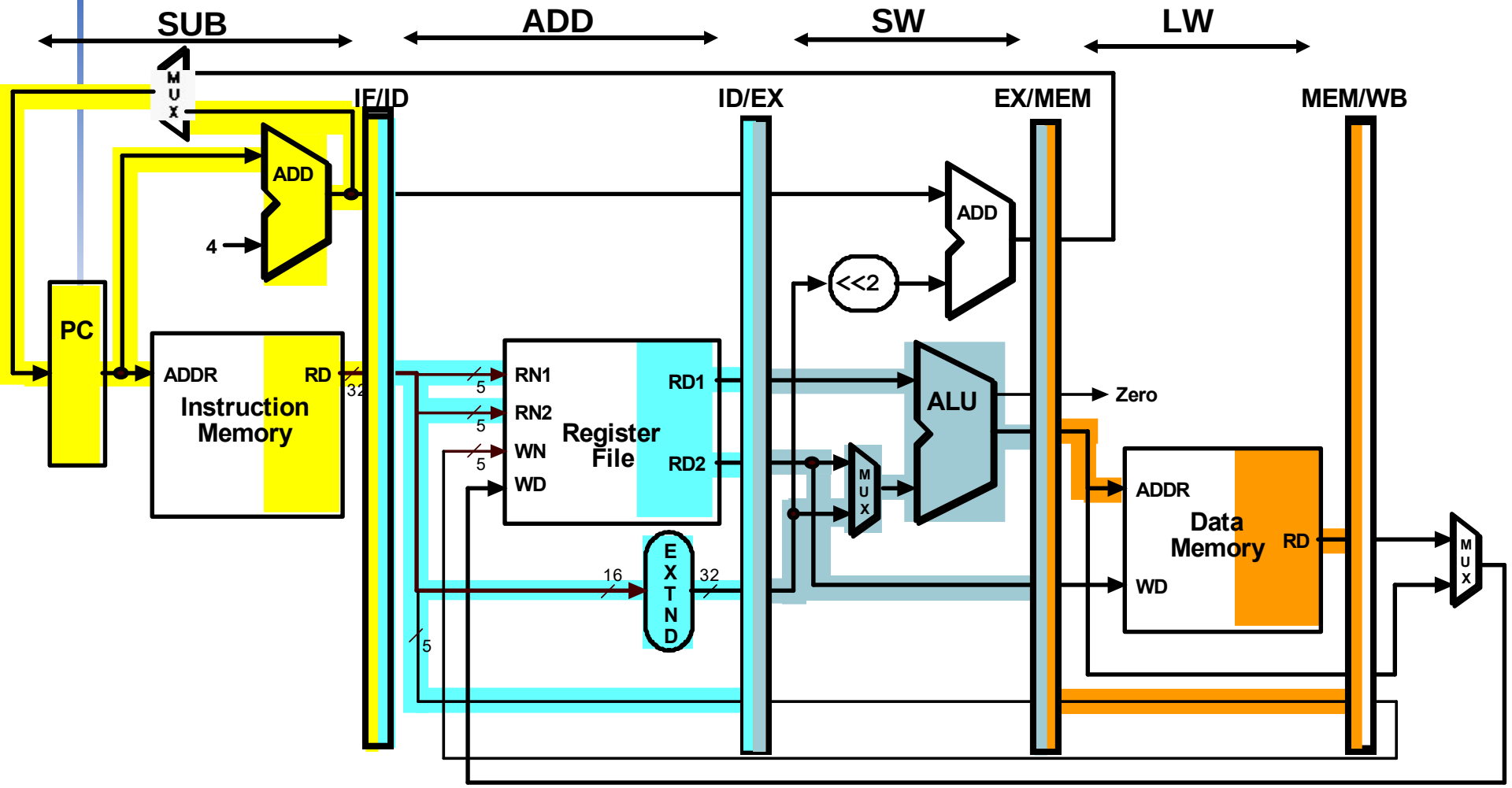
Clock Cycle 2



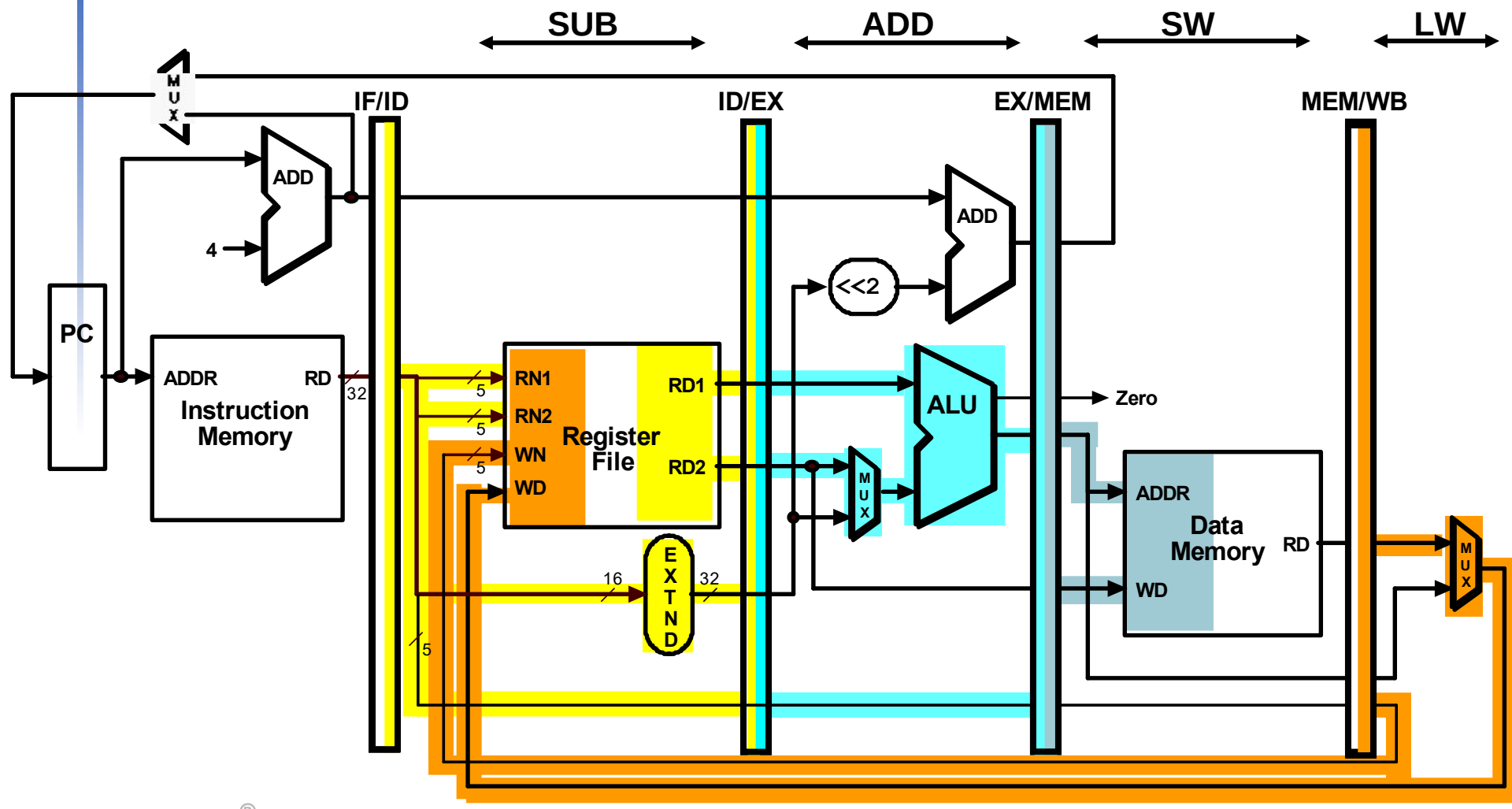
Single Clock Cycle Diagram: Clock Cycle 3



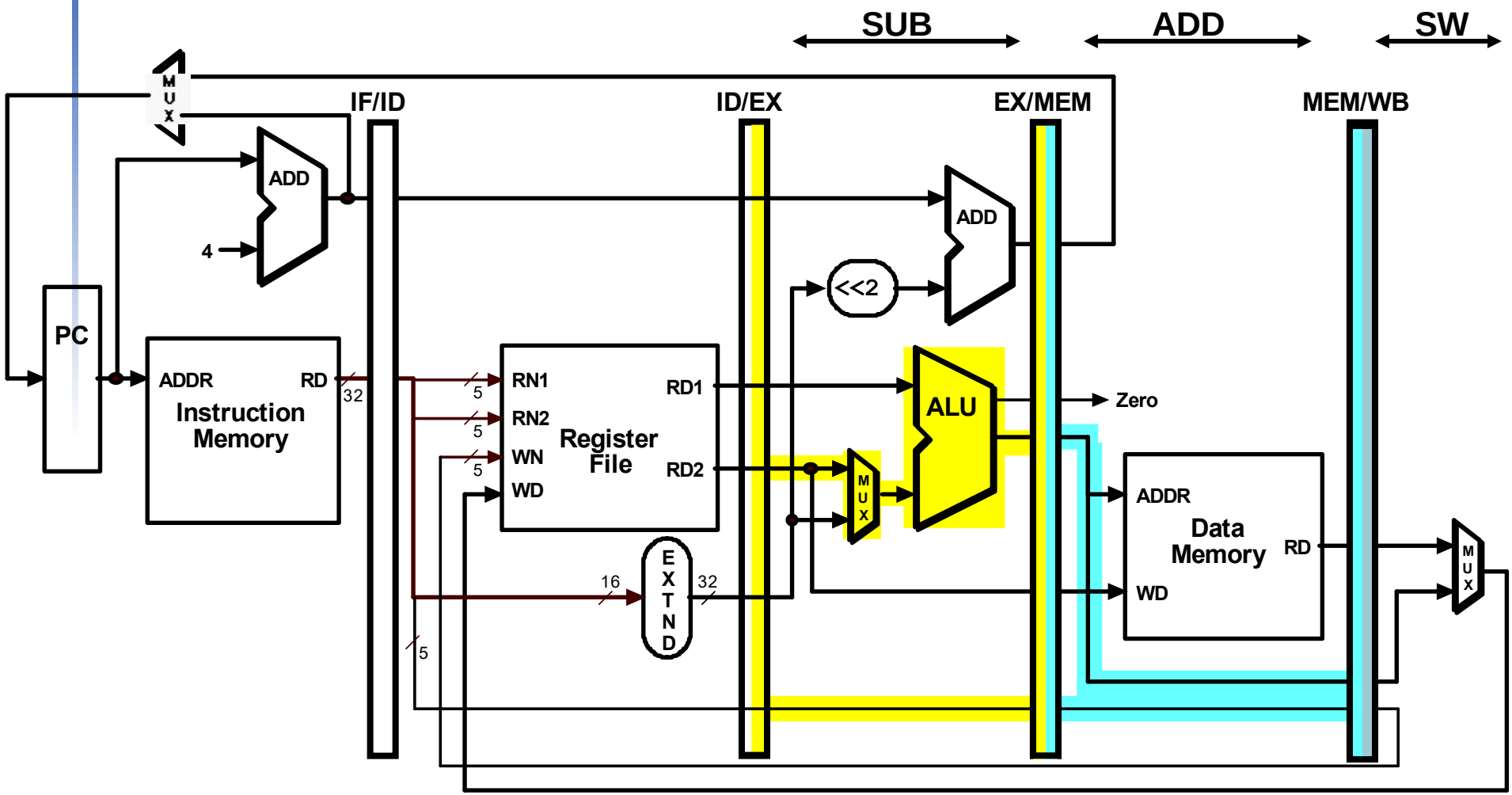
Single Clock Cycle Diagram: Clock Cycle 4



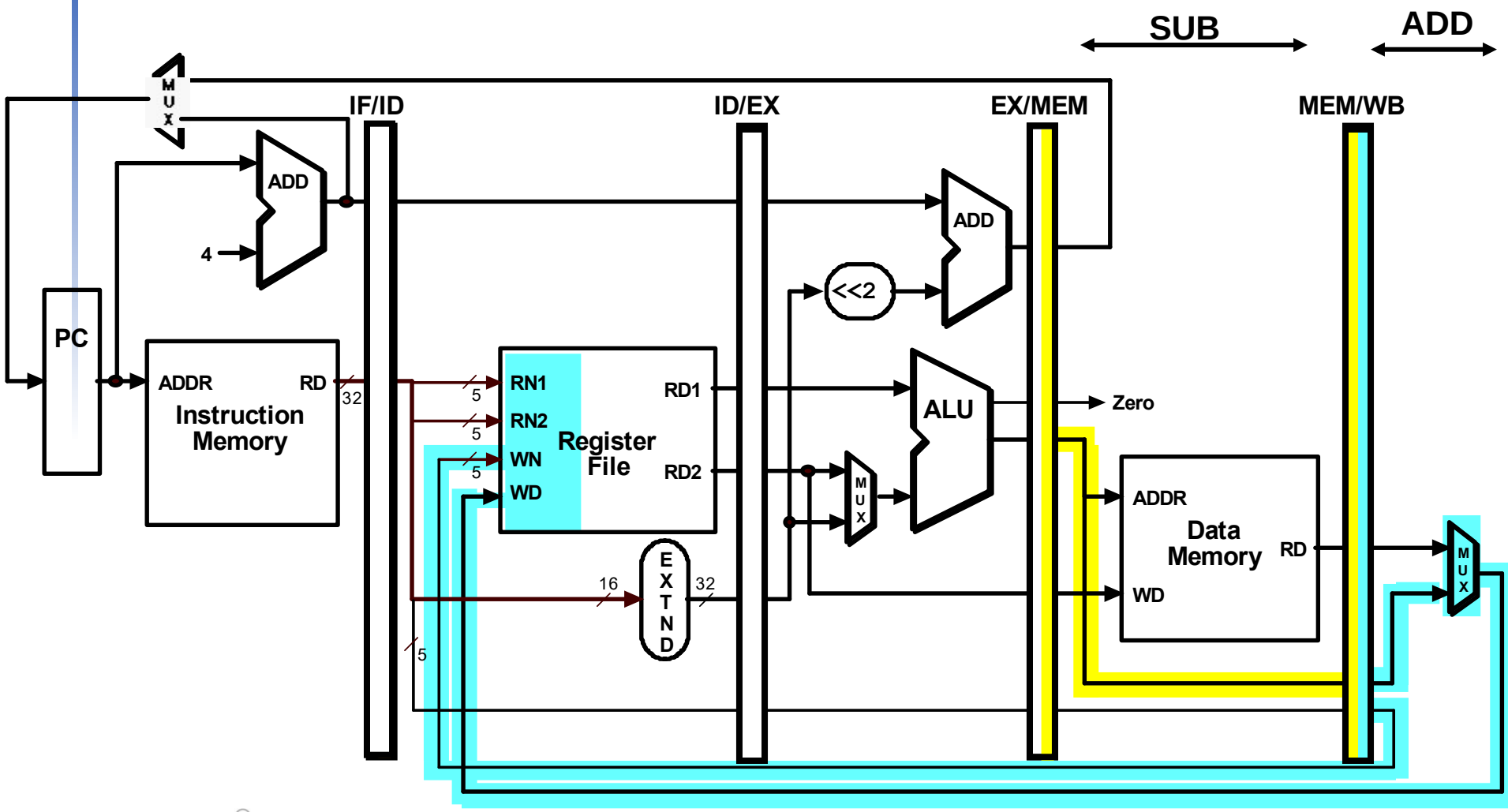
Single Clock Cycle Diagram: Clock Cycle 5



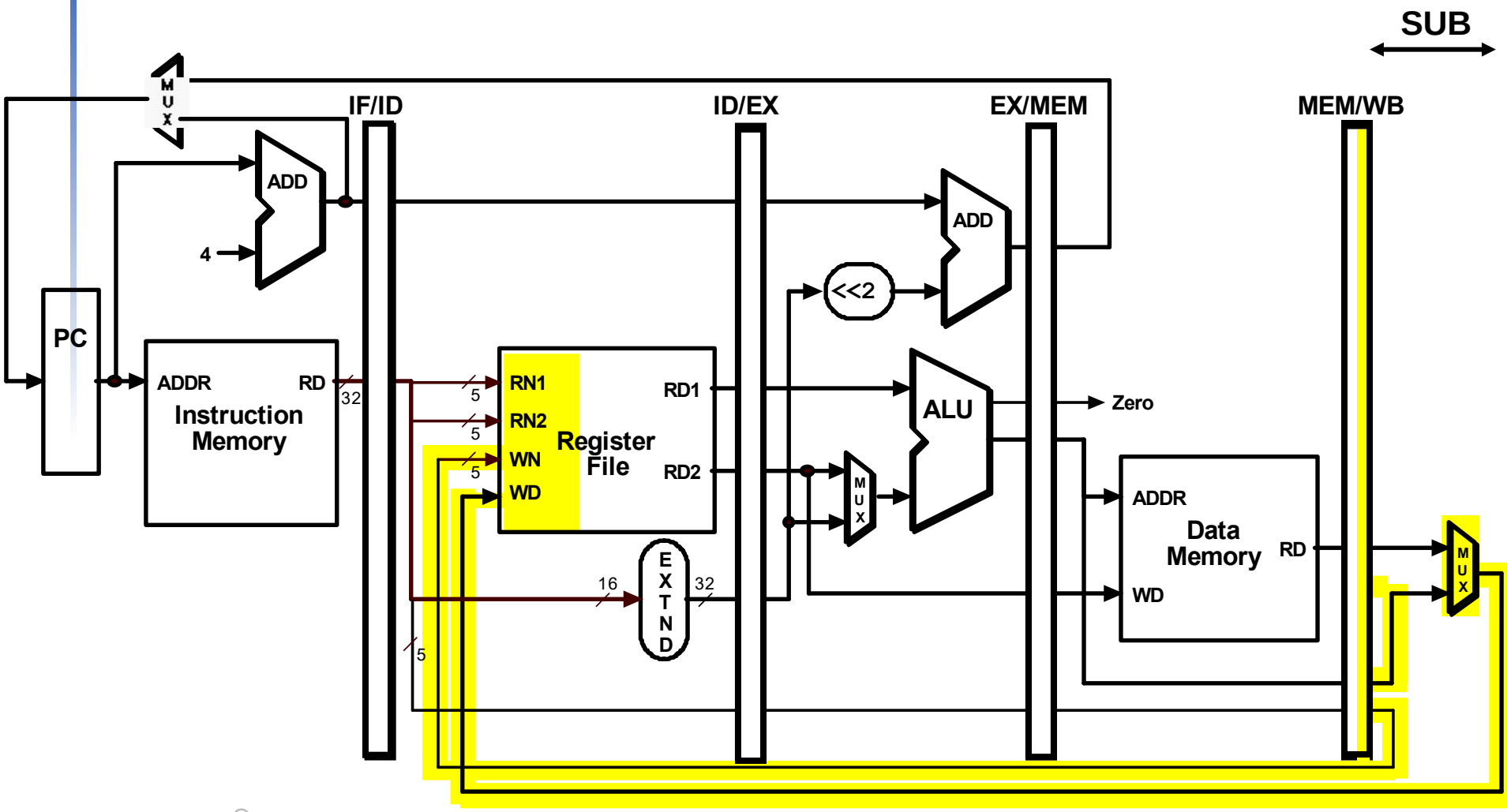
Single Clock Cycle Diagram: Clock Cycle 6



Single Clock Cycle Diagram: Clock Cycle 7

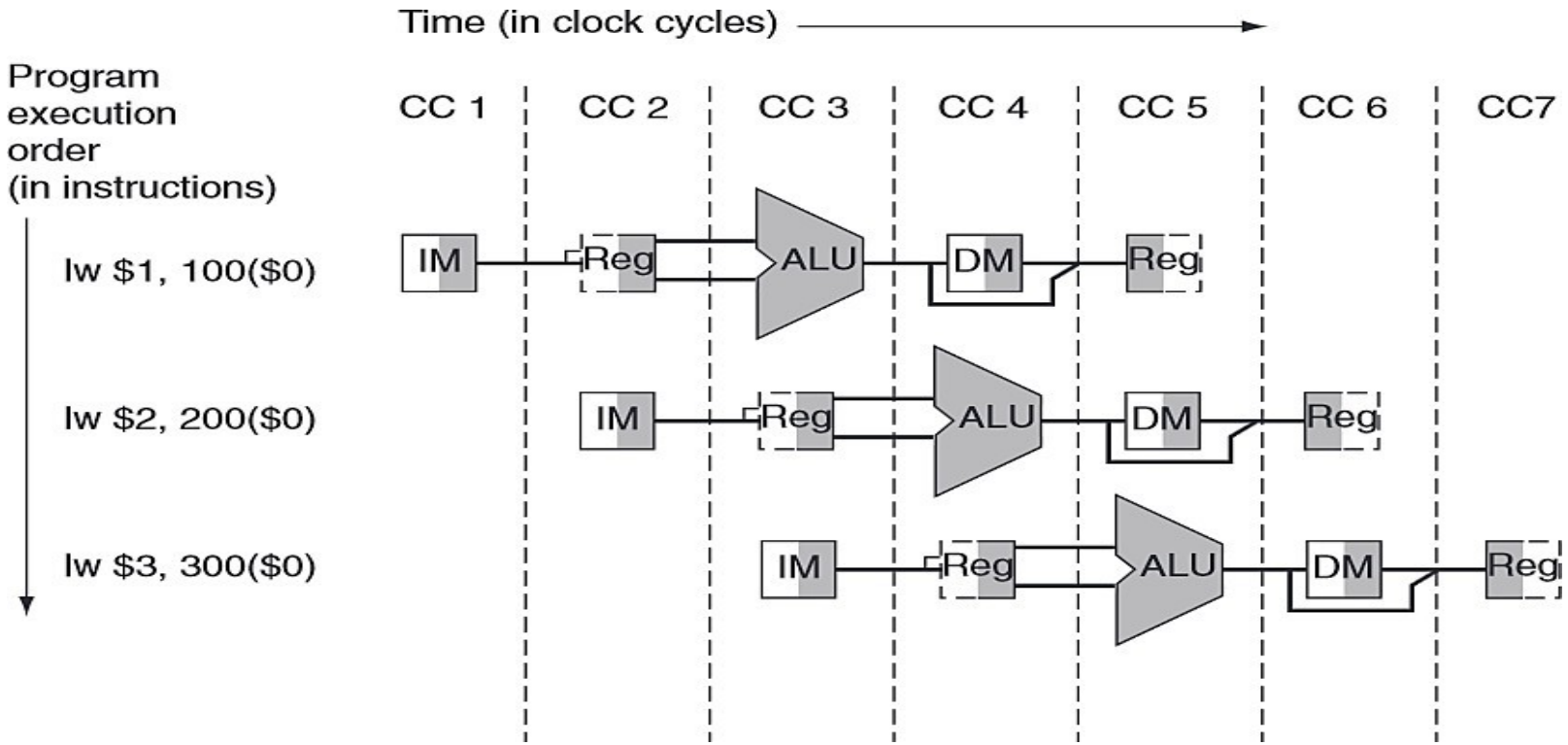


Clock Cycle 8



Pipeline behavior

- Pipeline takes new instruction into its IF Stage in every clock cycle



Hazards

- Situations that prevent starting the next instruction in the next cycle
  - **Dependencies**: relationships between instructions that prevent one instruction from being moved past another.
- Structure hazards
  - A required resource is busy
- Data hazard
  - Need to wait for previous instruction to complete its data read/write
- Control hazard
  - Deciding on control action depends on previous instruction

Structure Hazards

- A **structural hazard** occurs when a planned instruction cannot execute in the proper clock cycle because the hardware cannot support the particular combination of instructions that are set to execute in the given clock cycle.

- In the laundry analogy, a structural hazard might occur if we used a combo washer/dryer instead of separate washer and dryer machines.

Structure Hazards

- Conflict for use of a resource or inadequate hardware to simultaneously support all instructions in the pipeline in the same clock cycle
- **In MIPS pipeline with a single memory**
  - Load/store requires data access
  - Instruction fetch would have to *stall* for that cycle
    - Would cause a pipeline “bubble”
- Hence, pipelined data paths require separate instruction/data memories

PIPELINE HAZARDS

A **structural hazard** occurs when a planned instruction cannot execute in the proper clock cycle because the hardware cannot support the particular combination of instructions that are set to execute in the given clock cycle.

Imagine the following instructions are executed over 8 clock cycles. Notice how in cycle 4, we have a MEM and IF phase executing. If there is only one single memory unit, we will have a structural hazard.

cycle	1	2	3	4	5	6	7	8
lw	IF	ID	EX	MEM	WB			
lw		IF	ID	EX	MEM	WB		
lw			IF	ID	EX	MEM	WB	
lw				IF	ID	EX	MEM	WB

Data Hazards

A **data hazard** occurs when a planned instruction cannot execute in the proper clock cycle because the data that is needed is not yet available.

Consider the following instructions:

```
add    $s0, $t0, $t1
sub    $t2, $s0, $t3
```

cycle	1	2	3	4	5	6
add	IF	ID	EX	MEM	WB	
sub		IF	ID	EX	MEM	WB

Here the current instruction depends on completion of data access by a previous instruction

Data Hazards

The **add** instruction does not write its results to the register file until the fifth stage (cycle 5).

However, the **sub** instruction will need the updated value of ***$s0*** in its second stage (cycle 3).

add ***$s0***, $t0, $t1

sub $t2, ***$s0***, $t3

cycle	**1**	**2**	**3**	**4**	**5**	**6**
add	IF	ID	EX	MEM	WB	
sub		IF	ID	EX	MEM	WB

Data Hazards

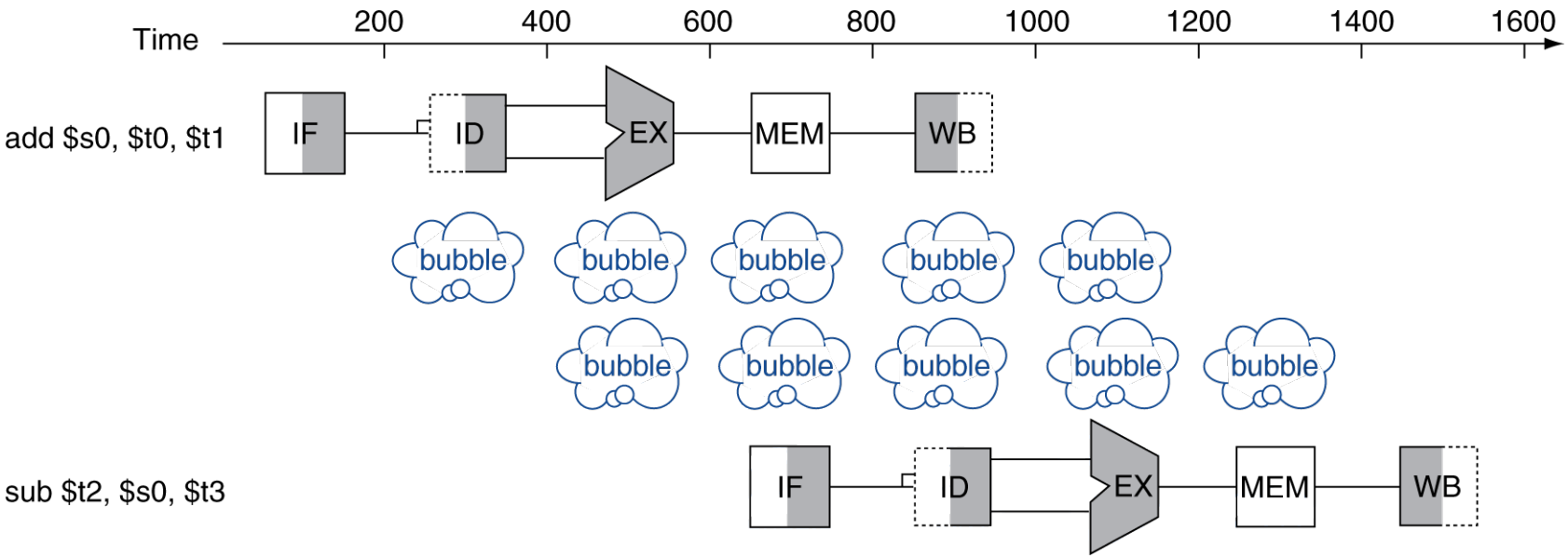
The worst case solution involves stalling the sub instruction for 3 cycles. While this resolves our dependency issue, it's not ideal. As a note, it is part of the compiler's job to identify dependencies like this and reorder instructions if possible. But we cannot rely on that solution either.

```
add     $s0, $t0, $t1
sub     $t2, $s0, $t3
```

cycle	1	2	3	4	5	6	7	8	9
add	IF	ID	EX	MEM	WB				
sub					IF	ID	EX	MEM	WB

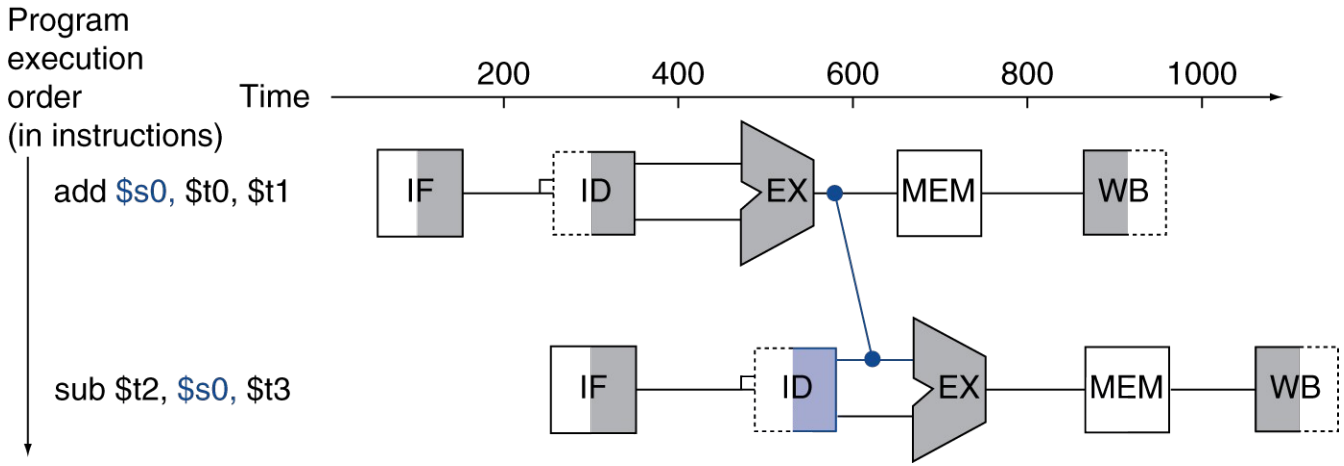
Data Hazards

- Optimization: writing to the register to the first half of the clock and reading from the register second half of the clock
  - `add $s0, $t0, $t1`
    `sub $t2, $s0, $t3`



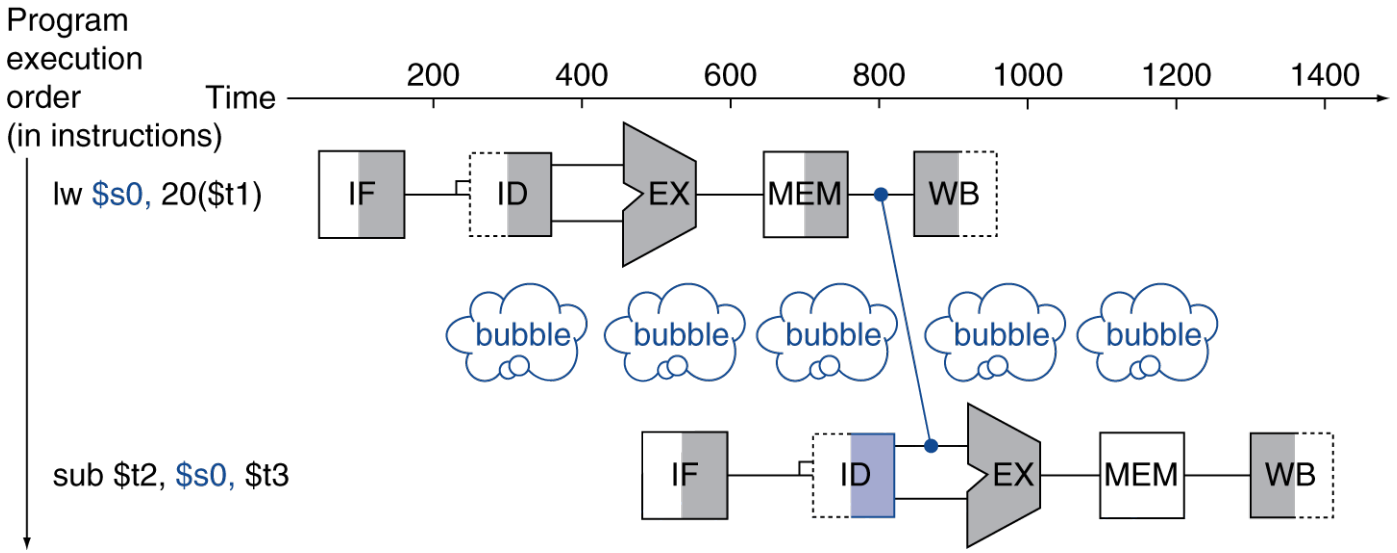
Forwarding (aka Bypassing)

- Another solution for the problem is known as **forwarding** (or **bypassing**). This method involves retrieving the data from internal buffers rather than waiting for the data to be updated in the register file or data memory
  - Because the result of the add operation is available at the end of the EX stage, we can grab its value for use in the subsequent instruction.
  - Requires extra connections in the data path



Load-Use Data Hazard

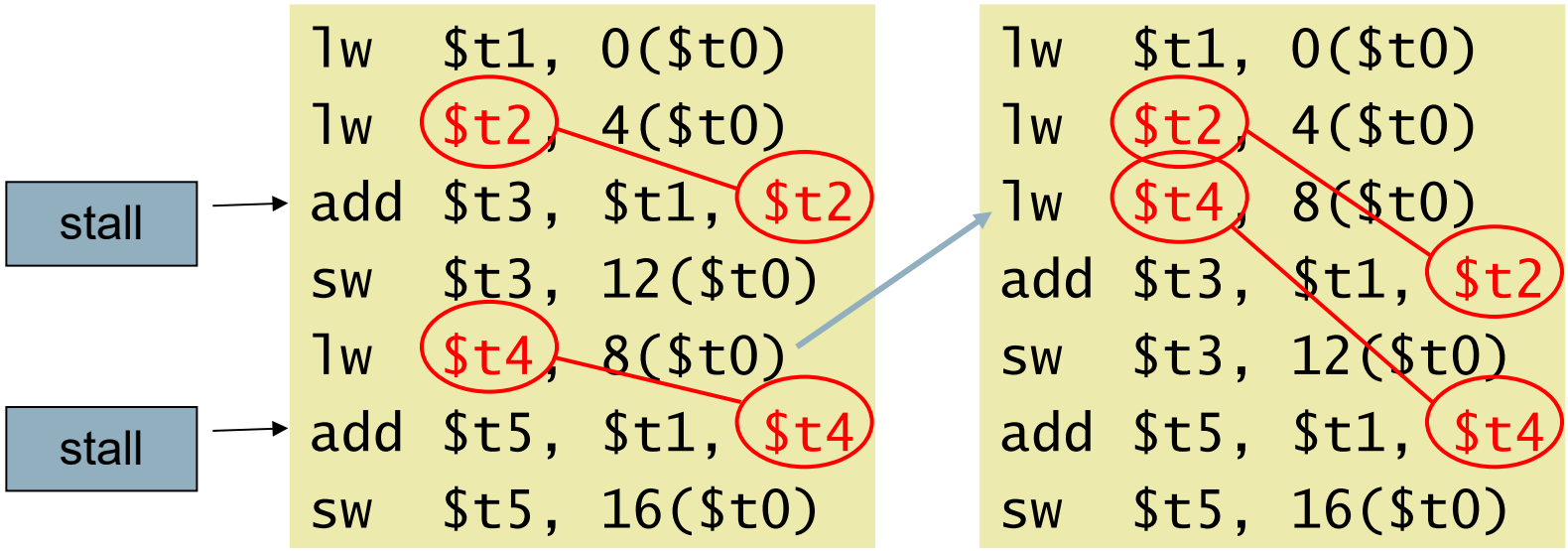
- We cannot always prevent all stalls with forwarding. Consider the following instructions.



Even if we use forwarding, the new contents of $s0 are only available after load word's MEM stage. So we'll have to stall the sub instruction one cycle.

Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction
- C code for `A = B + E; C = B + F;`



Code Scheduling to Avoid Stalls

If we are using a pipelined processor with forwarding, we have the following stages executing in each cycle:

cycle	1	2	3	4	5	6	7	8	9	10	11	12	13
lw $t1,0($t0)	IF	ID	EX	MEM	WB								
lw $t2,4($t0)		IF	ID	EX	MEM	WB							
add $t3,$t1,$t2				IF	ID	EX	MEM	WB					
sw $t3,12($t0)					IF	ID	EX	MEM	WB				
lw $t4,8($t0)						IF	ID	EX	MEM	WB			
add $t5,$t1,$t4								IF	ID	EX	MEM	WB	

Code Scheduling to Avoid Stalls

The reordering allows us to execute the program in two fewer cycles than before.

cycle	1	2	3	4	5	6	7	8	9	10	11	12	13
lw $t1,0($t0)	IF	ID	EX	MEM	WB								
lw $t2,4($t0)		IF	ID	EX	MEM	WB							
lw $t4,8($t0)			IF	ID	EX	MEM	WB						
add $t3,$t1,$t2				IF	ID	EX	MEM	WB					
sw $t3,12($t0)					IF	ID	EX	MEM	WB				
add $t5,$t1,$t4						IF	ID	EX	MEM	WB			
sw $t5,16($t0)							IF	ID	EX	MEM	WB		

Control Hazards

The third type of hazard, a **control hazard** (or **branch hazard**), occurs when the flow of instruction addresses is not known at the time that the next instruction must be loaded. Let's say we have the following instructions.

```
beq     $t0, $t1, L1
sub     $t2, $s0, $t3
```

We have a problem: we do not know what the next instruction should be until the end of the third cycle. But we're automatically fetching the next instruction in the second cycle.

cycle	1	2	3	4	5	6
beq	IF	ID	EX	MEM	WB	
sub		IF	ID	EX	MEM	WB

Control hazards

- Problem: The processor does not know soon enough
  - whether or not a conditional branch should be taken.
  - the target address of a transfer-of-control instruction.
- Solutions:
  - **Stall** until the necessary information becomes available.
  - **Predict** the outcome and act accordingly.

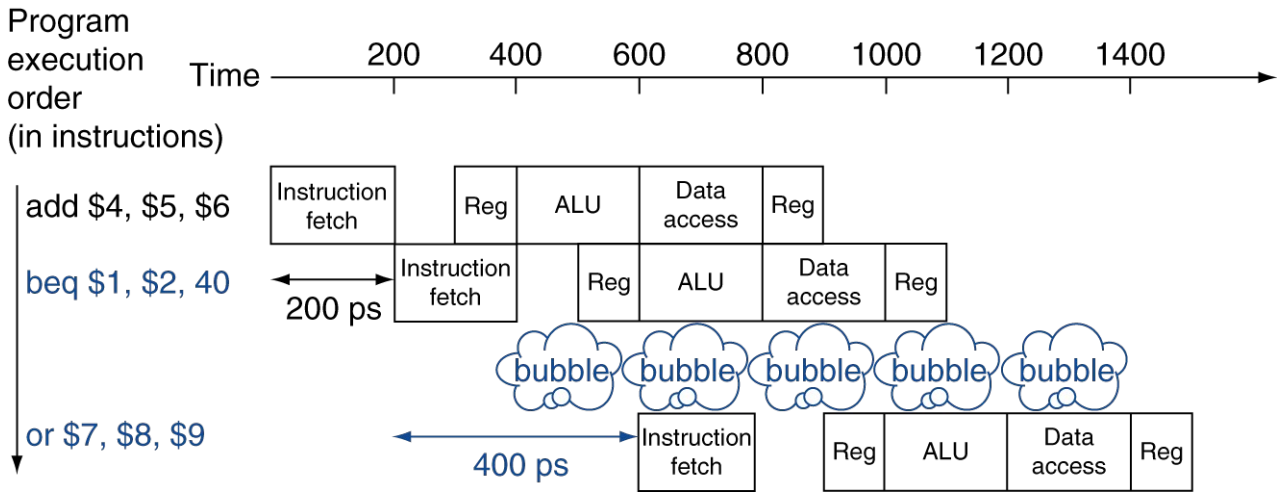
- If we stall until the branch target is known, we will always incur a penalty for stalling.
- However, if we predict that the branch is not taken and act accordingly, we will only incur a penalty when the branch actually is taken.

Control Hazards

- Branch determines flow of control
  - Fetching next instruction depends on branch outcome
  - Pipeline can’t always fetch correct instruction
    - Still working on ID stage of branch
- In MIPS pipeline
  - Need to compare registers and compute target early in the pipeline
  - **Add hardware to do it in ID stage**

Stall on Branch

- Wait until branch outcome determined before fetching next instruction

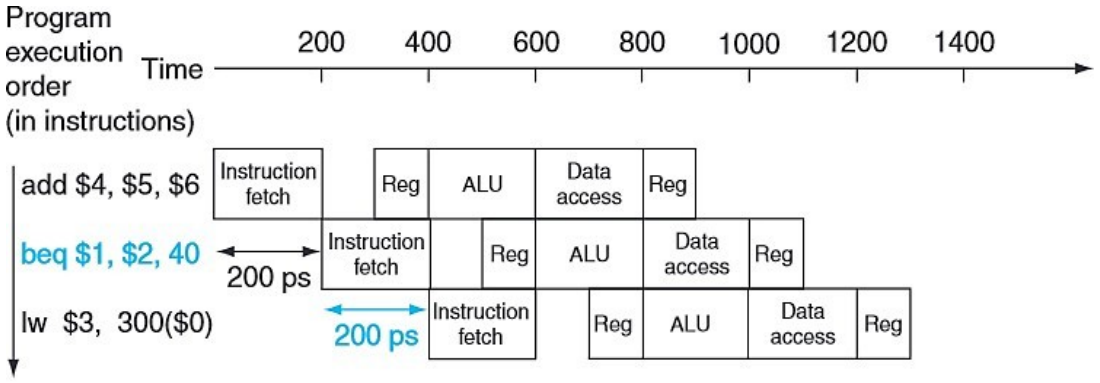


Branch Prediction

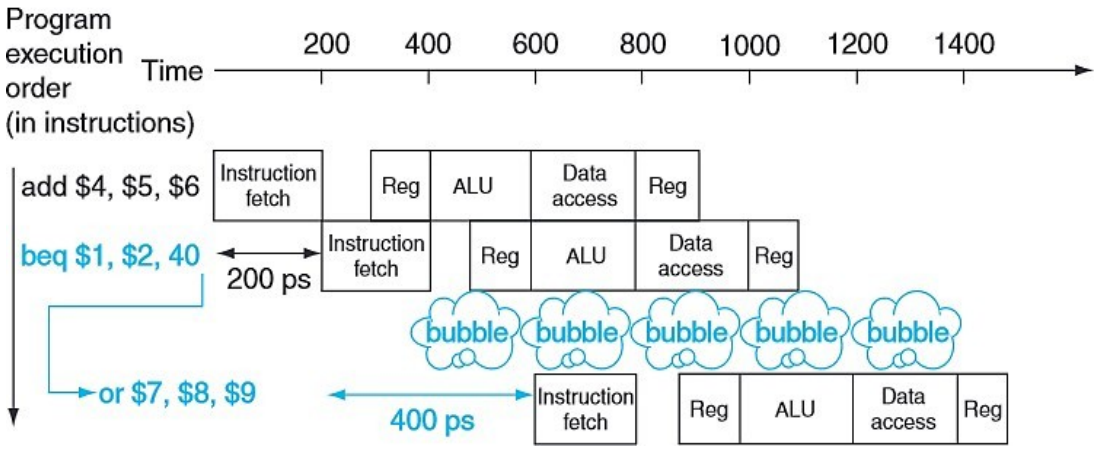
- Longer pipelines can't readily determine branch outcome early
  - Stall penalty becomes unacceptable
- **Predict outcome of branch**
  - Only stall if prediction is wrong
- In MIPS pipeline
  - Can predict branch outcome as **not taken**
  - Fetch instruction after branch, with no delay

MIPS with Predict Not Taken

When predicting that a branch is not taken, we proceed as normal. If the branch is not taken, there is no issue.



If the branch is taken, however, we incur a stalling penalty. This penalty can vary but even in a highly optimized pipeline we will have to essentially "stall" for a cycle.



Pipeline Summary

The BIG Picture

- Pipelining improves performance by increasing instruction throughput
  - Executes multiple instructions in parallel
  - Each instruction has the same latency
- Subject to hazards
  - Structure, data, control
- Pipelining is a technique that exploits parallelism among the instructions in a sequential instruction stream.