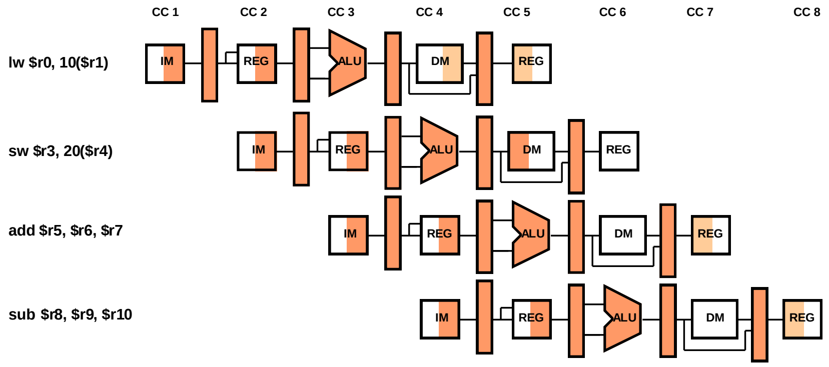
Computer Organization

Lecture 17 - Pipelined Processor Design 1

Reading: 4.5-4.6



Pipelining Outline

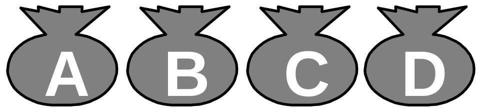
- Introduction
  - Defining Pipelining ◀
  - Pipelining Instructions
  - Hazards
- Pipelined Processor Design
  - Datapath
  - Control
- Advanced Pipelining
  - Superscalar
  - Dynamic Pipelining
  - Examples

What is Pipelining?

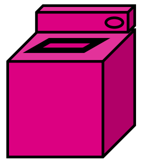
- A way of speeding up execution of instructions
- Key idea: overlap execution of multiple instructions
- Analogy: doing your laundry

The Laundry Analogy

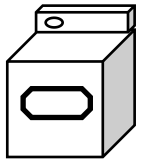
- Ann, Brian, Cathy, Dave each has one load of clothes to wash, dry, and fold



- Washer takes 30 minutes

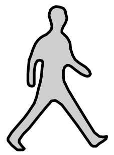


- Dryer takes 30 minutes

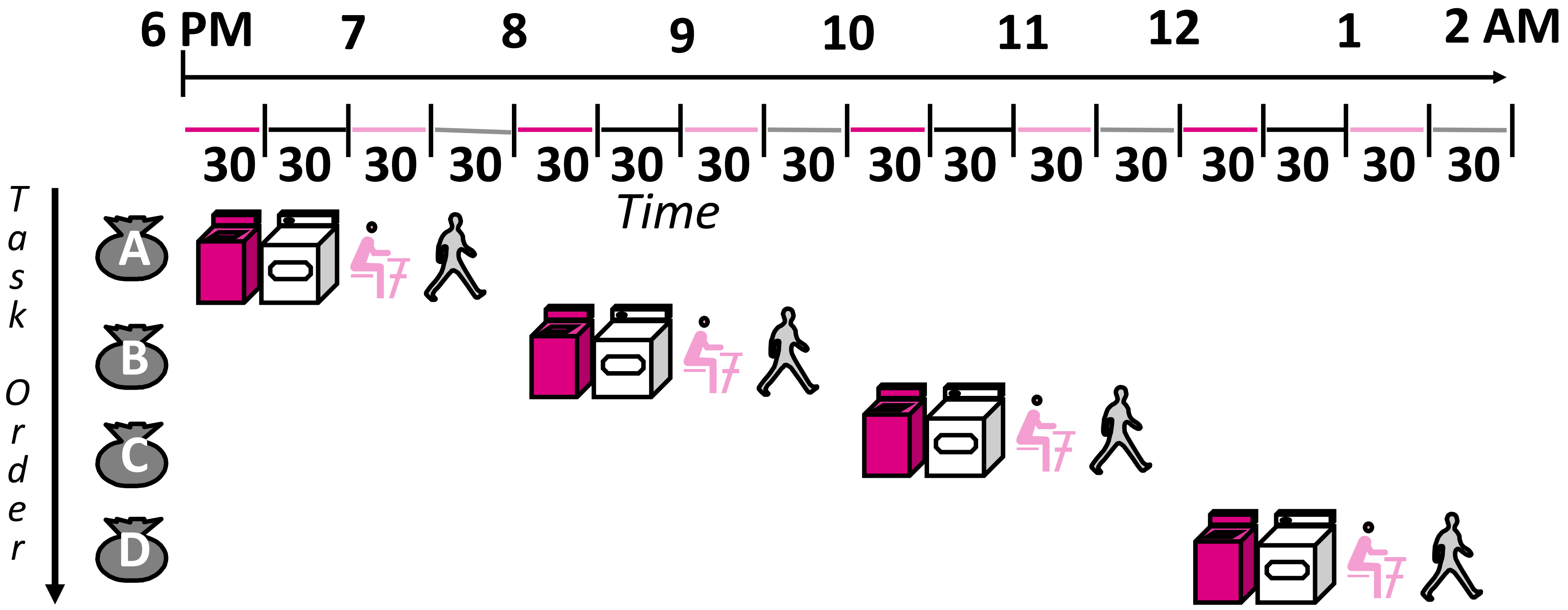


- “Folder” takes 30 minutes

- “Stasher” takes 30 minutes to put clothes into drawers

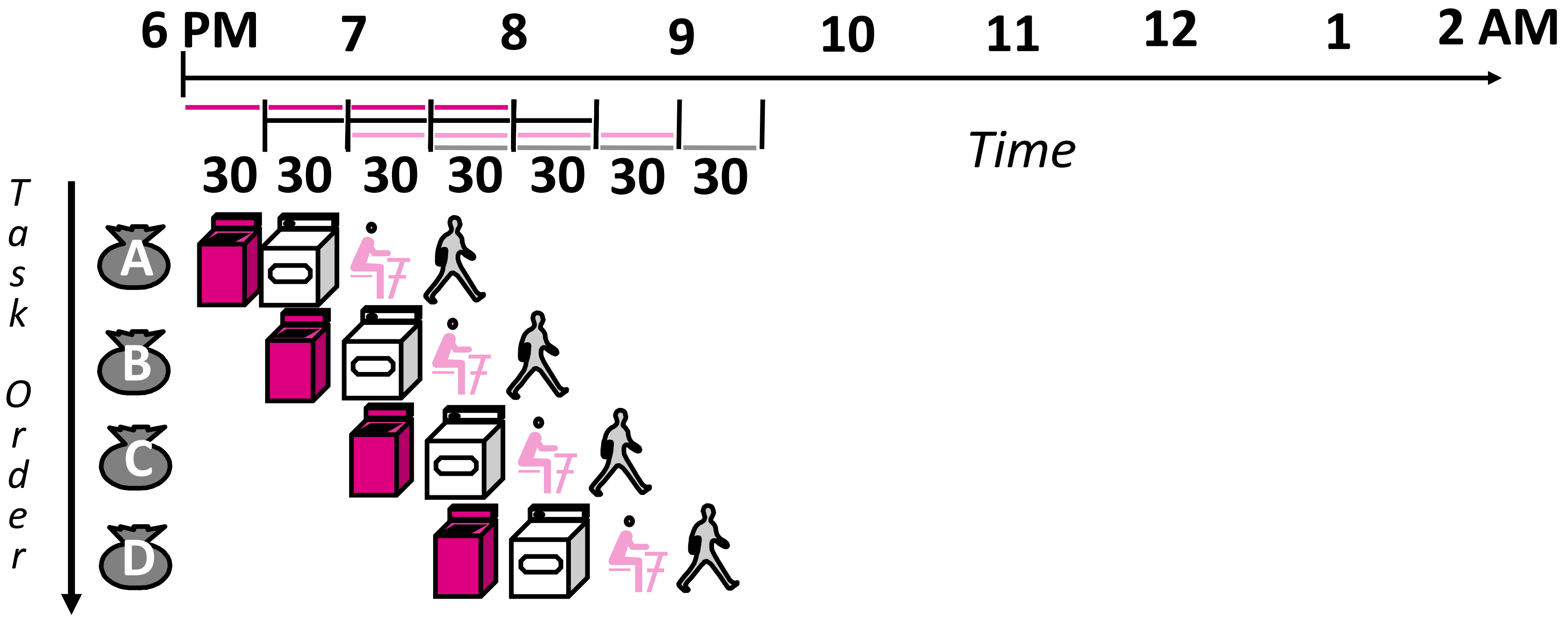


If we do laundry sequentially...



- Time Required: 8 hours for 4 loads

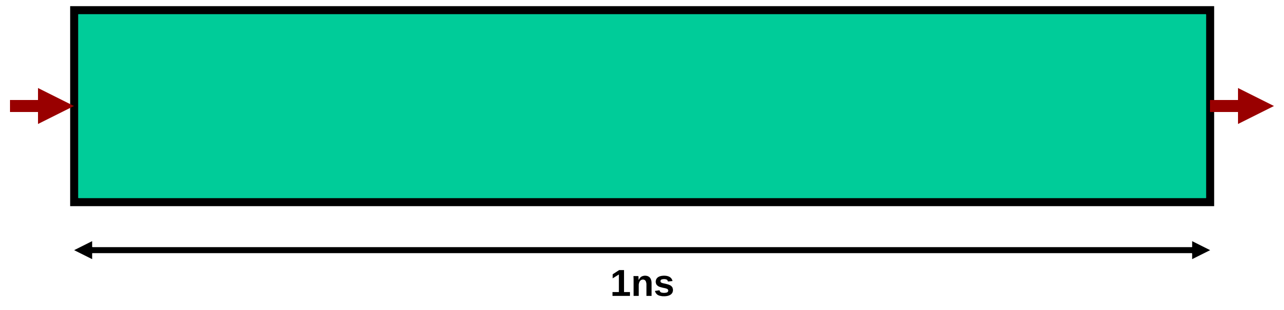
To Pipeline, We Overlap Tasks



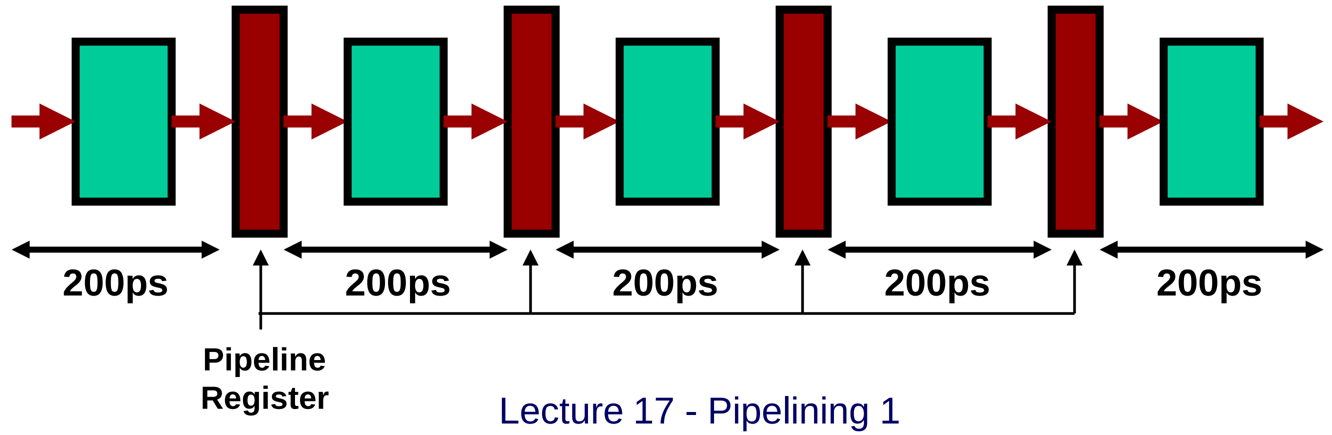
- **Time Required: 3.5 Hours for 4 Loads**
  - **Latency remains 2 hours**
  - **Throughput improves by factor of 2.3 (decreases for more loads)**

Pipelining a Digital System

- Key idea: break big computation up into pieces

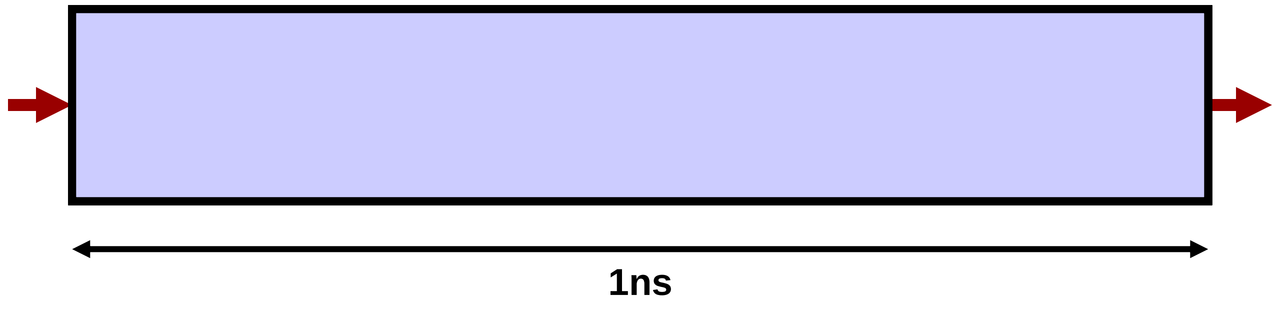


- Separate each piece with a pipeline register

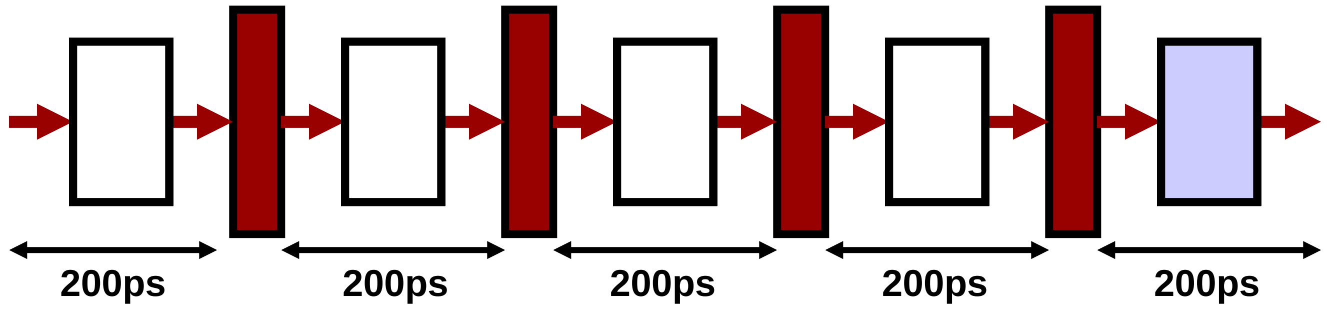


Pipelining a Digital System

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



Non-pipelined:
1 operation finishes every 1ns



Pipelined:
1 operation finishes every 200ps

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

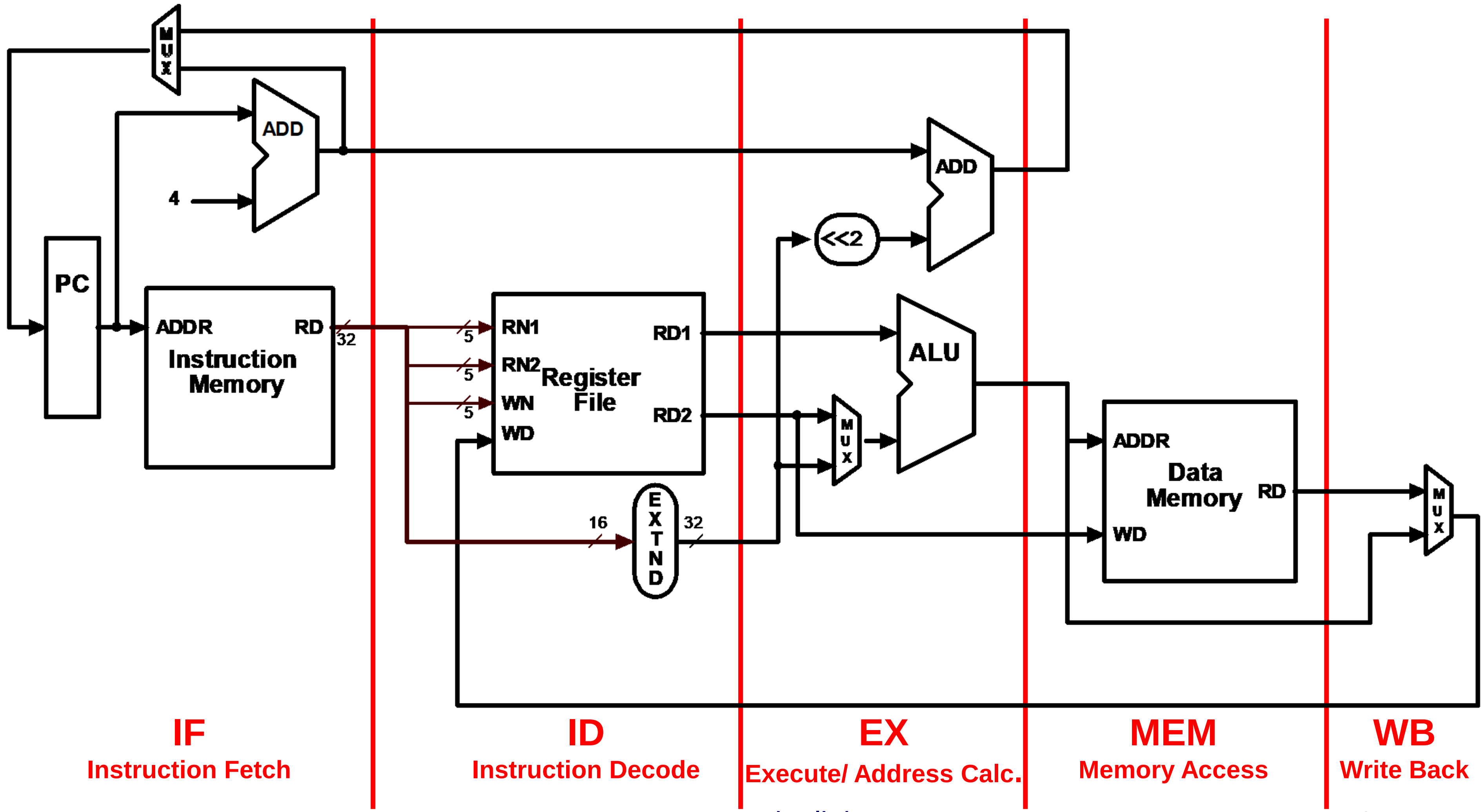
Pipelining Outline

- Introduction
  - Defining Pipelining
  - Pipelining Instructions ◀
  - Hazards
- Pipelined Processor Design
  - Datapath
  - Control
- Advanced Pipelining
  - Superscalar
  - Dynamic Pipelining
  - Examples

Pipelining a Processor

- Recall the 5 steps in instruction execution:
  1. Instruction Fetch
  2. Instruction Decode and Register Read
  3. Execution operation or calculate address
  4. Memory access
  5. Write result into register
- Review: Single-Cycle Processor
  - All 5 steps done in a single clock cycle
  - Dedicated hardware required for each step
- What happens if we break execution into multiple cycles, but keep the extra hardware?

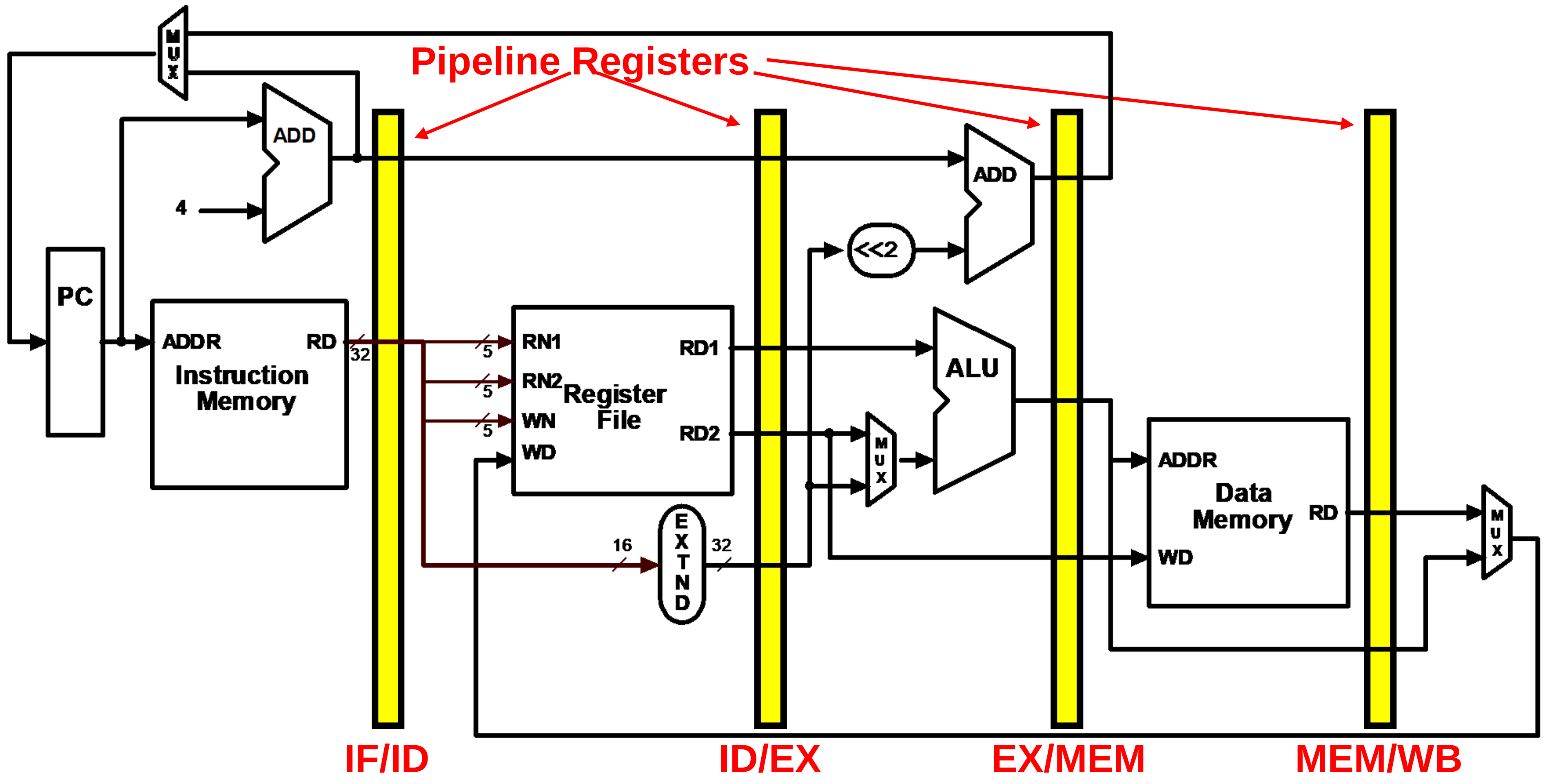
Review - Single-Cycle Processor



Pipelining - Key Idea

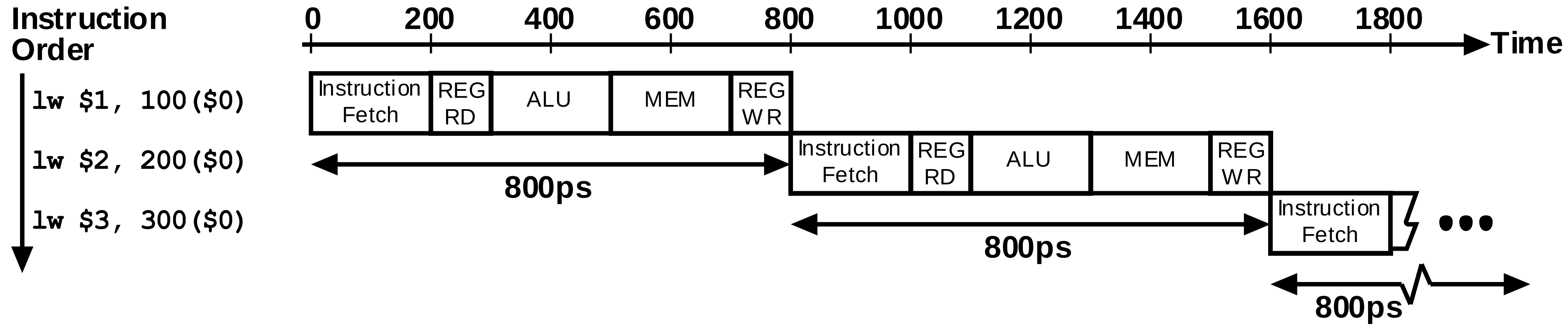
- **Question:** **What happens if we break execution into multiple cycles, but keep the extra hardware?**
- **Answer:** **in the best case, we can start executing a new instruction on each clock cycle - this is pipelining**
- **Pipelining stages:**
  - **IF - Instruction Fetch**
  - **ID - Instruction Decode**
  - **EX - Execute / Address Calculation**
  - **MEM - Memory Access (read / write)**
  - **WB - Write Back (results into register file)**

Basic Pipelined Processor

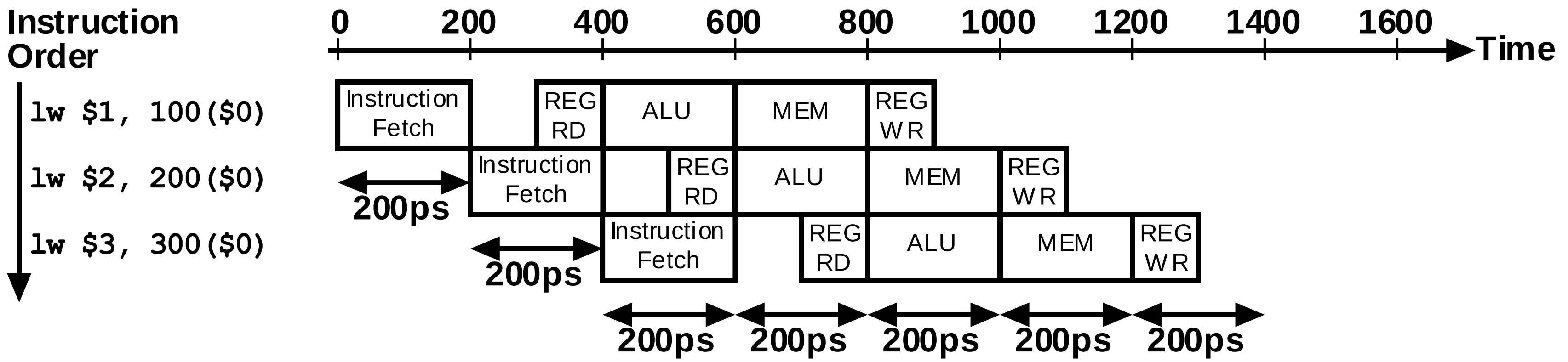


Single-Cycle vs. Pipelined Execution

Non-Pipelined



Pipelined



Comments about Pipelining

- The good news
  - Multiple instructions are being processed at same time
  - This works because stages are isolated by registers
  - For N stages, 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
  - Worst case: must suspend execution - stall
    - Wait until needed hardware is available OR
    - Wait until needed data is available

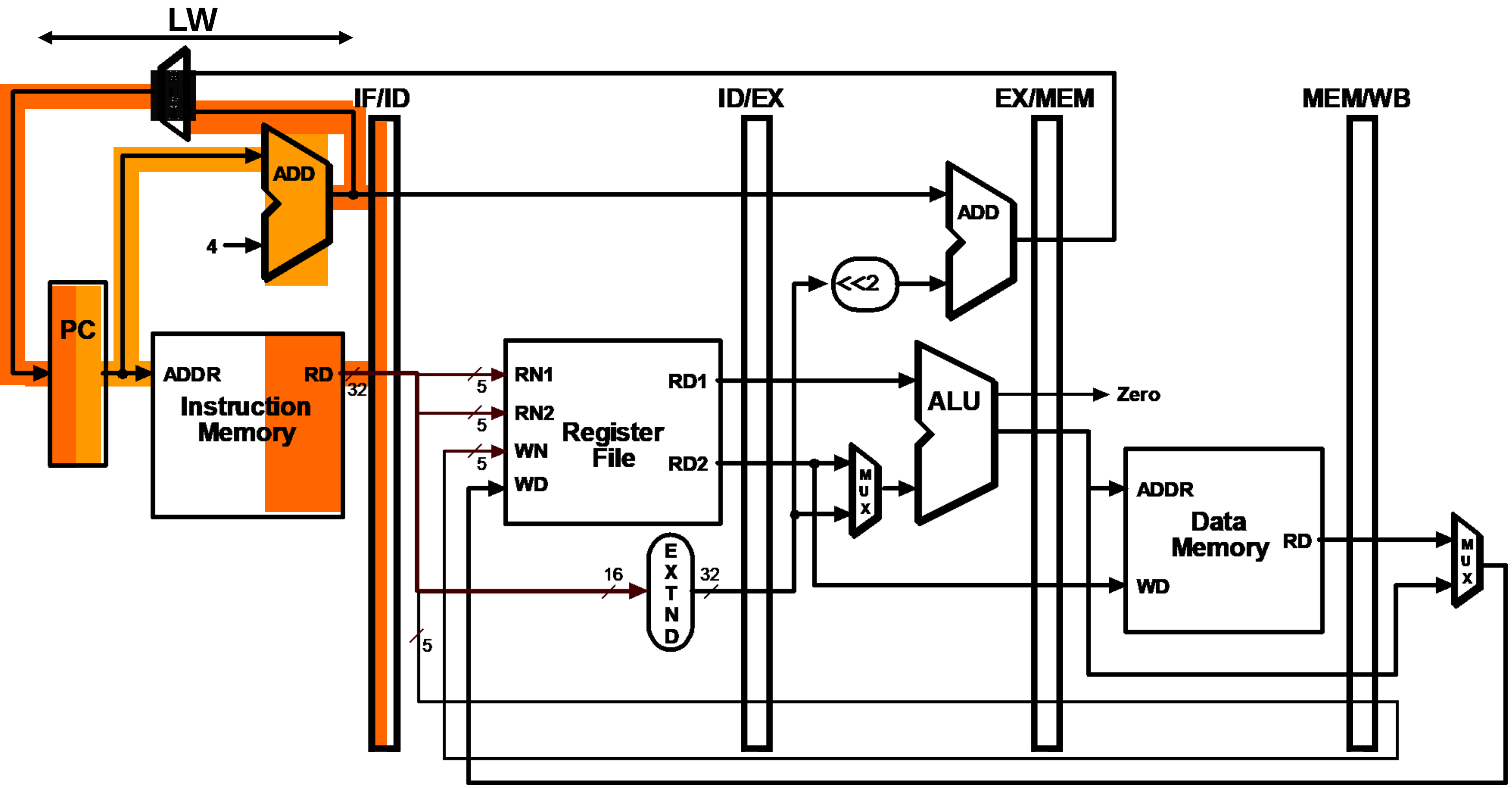
Pipelined Example - Executing Multiple Instructions

- **Consider the following instruction sequence:**

```
lw $r0, 10($r1)
sw $r3, 20($r4)
add $r5, $r6, $r7
sub $r8, $r9, $r10
```

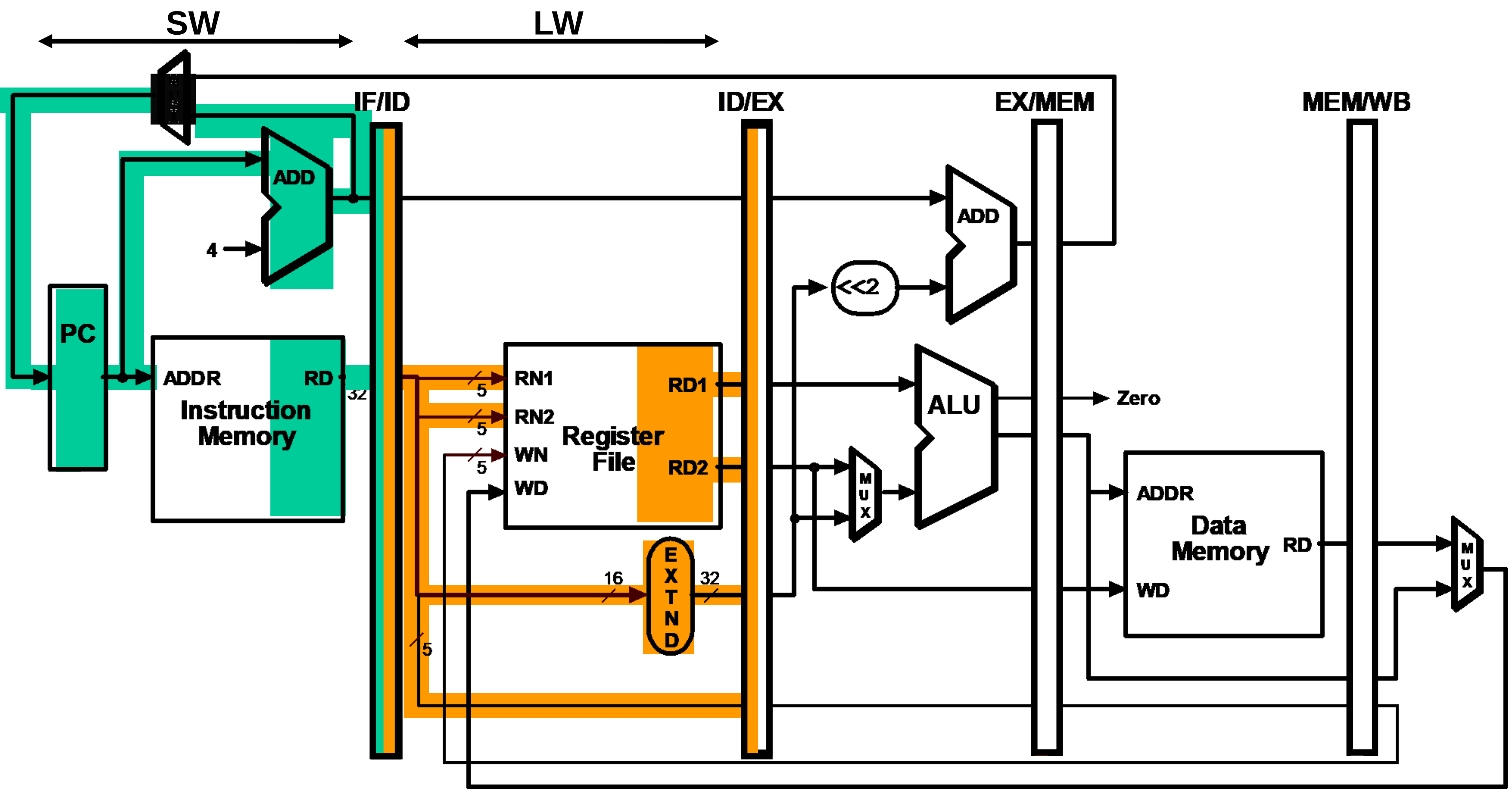
Executing Multiple Instructions Clock Cycle 1

```
lw $r0, 10($r1)
sw $r3, 20($r4)
add $r5, $r6, $r7
sub $r8, $r9, $r10
```



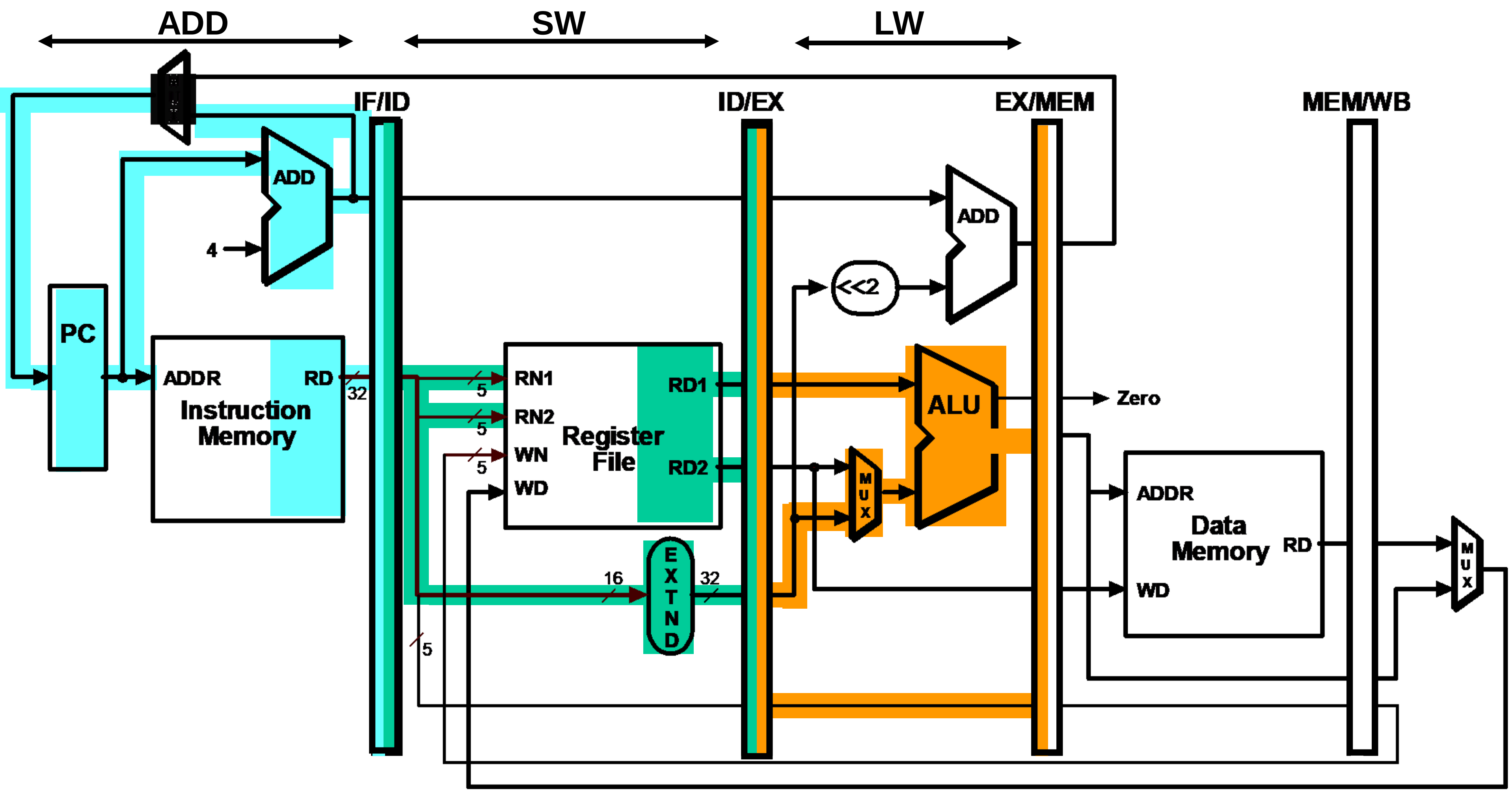
Executing Multiple Instructions Clock Cycle 2

```
lw $r0, 10($r1)
sw $r3, 20($r4)
add $r5, $r6, $r7
sub $r8, $r9, $r10
```



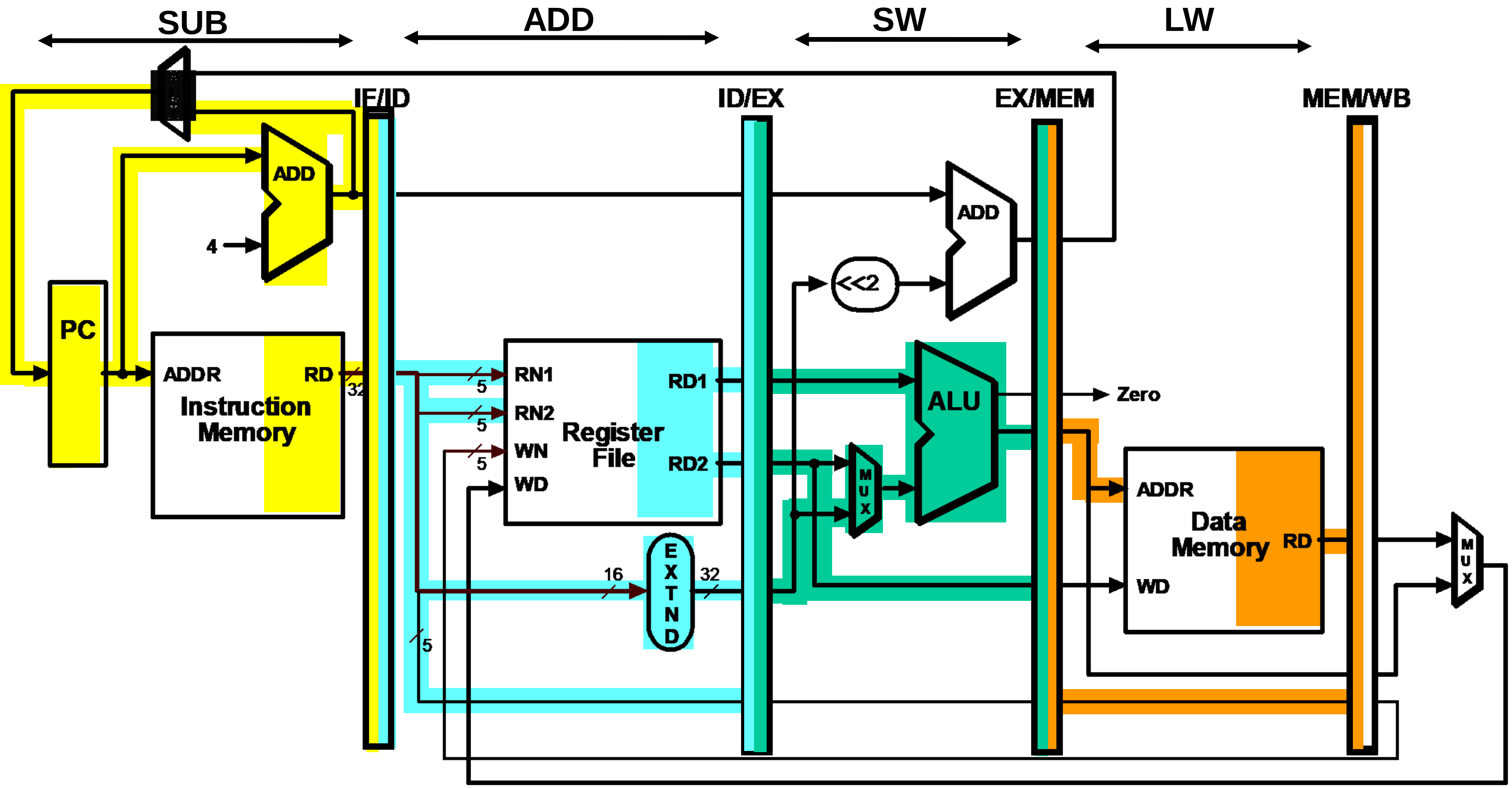
Executing Multiple Instructions Clock Cycle 3

```
lw $r0, 10($r1)
sw $r3, 20($r4)
add $r5, $r6, $r7
sub $r8, $r9, $r10
```



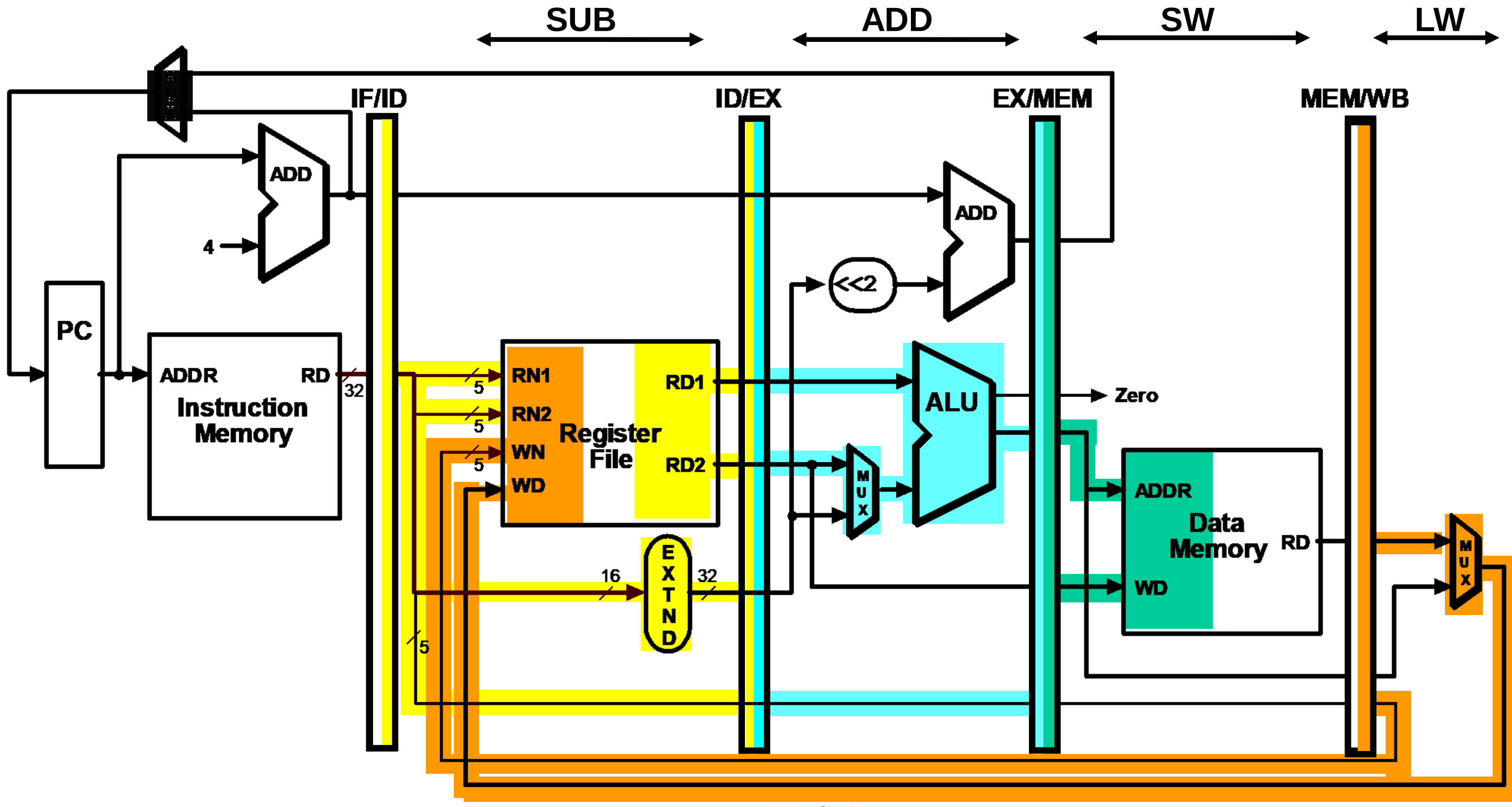
Executing Multiple Instructions Clock Cycle 4

```
lw $r0, 10($r1)
sw $r3, 20($r4)
add $r5, $r6, $r7
sub $r8, $r9, $r10
```



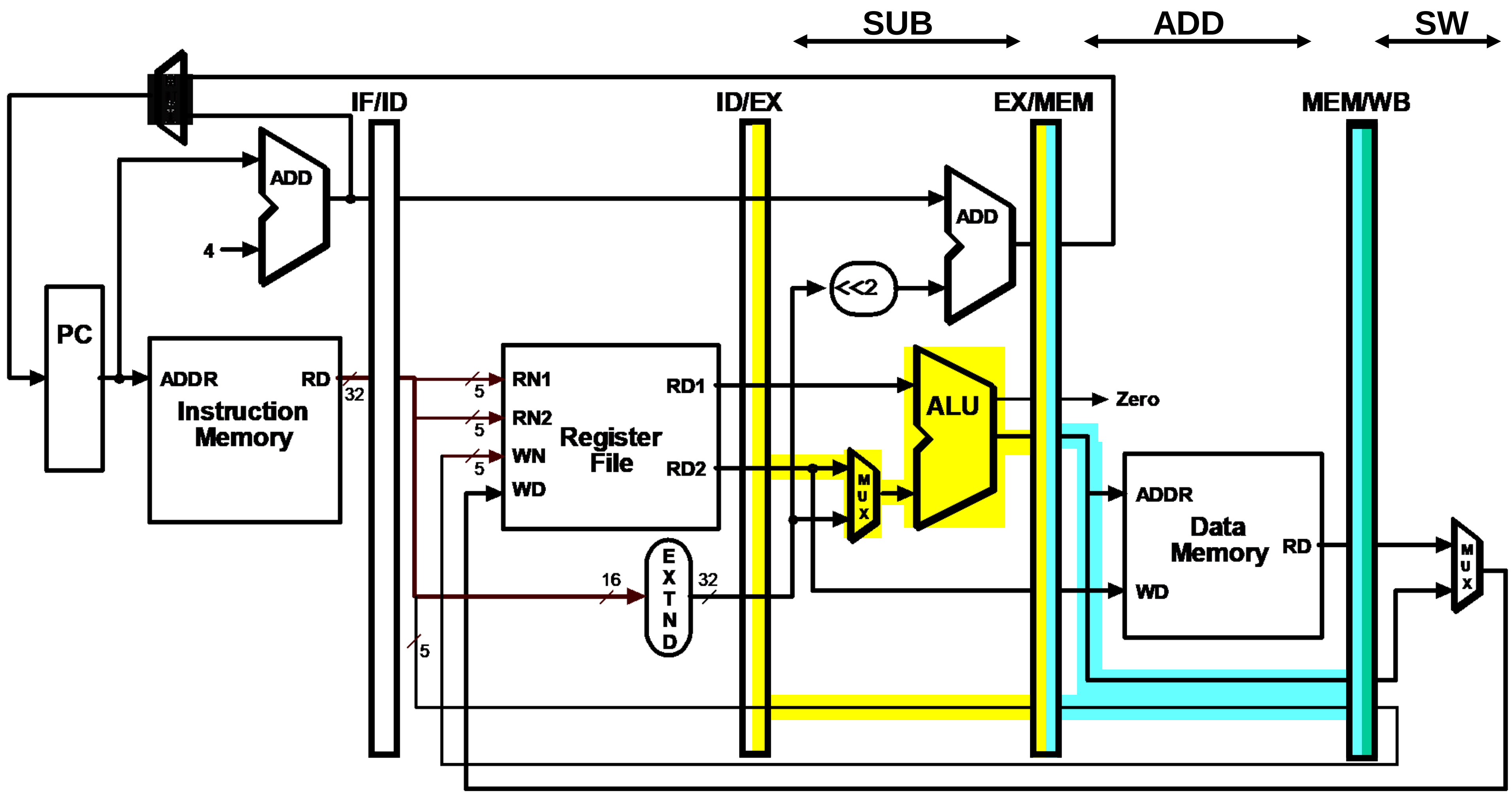
Executing Multiple Instructions Clock Cycle 5

```
lw $r0, 10($r1)
sw $r3, 20($r4)
add $r5, $r6, $r7
sub $r8, $r9, $r10
```



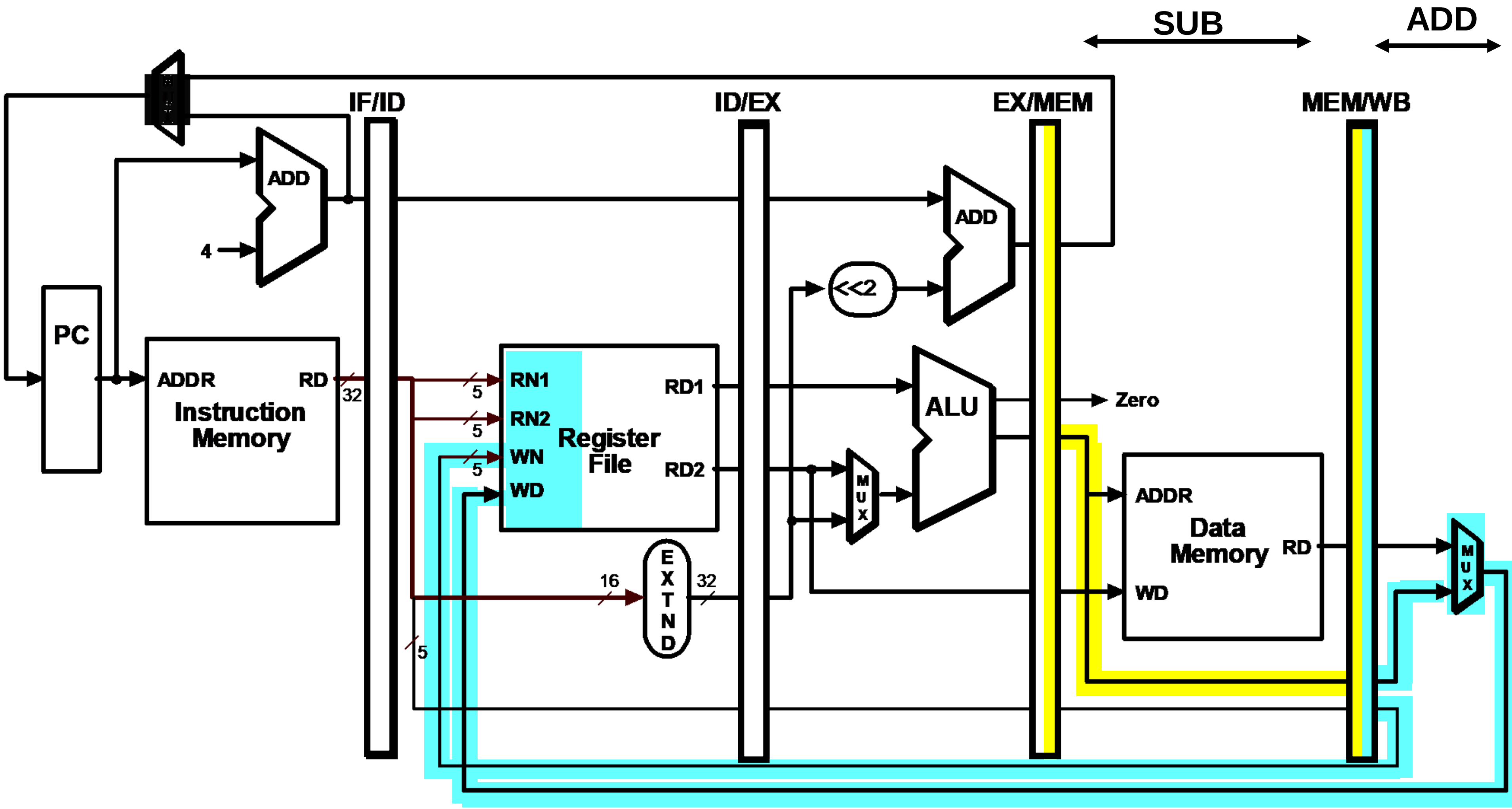
Executing Multiple Instructions Clock Cycle 6

```
lw $r0, 10($r1)
sw $r3, 20($r4)
add $r5, $r6, $r7
sub $r8, $r9, $r10
```



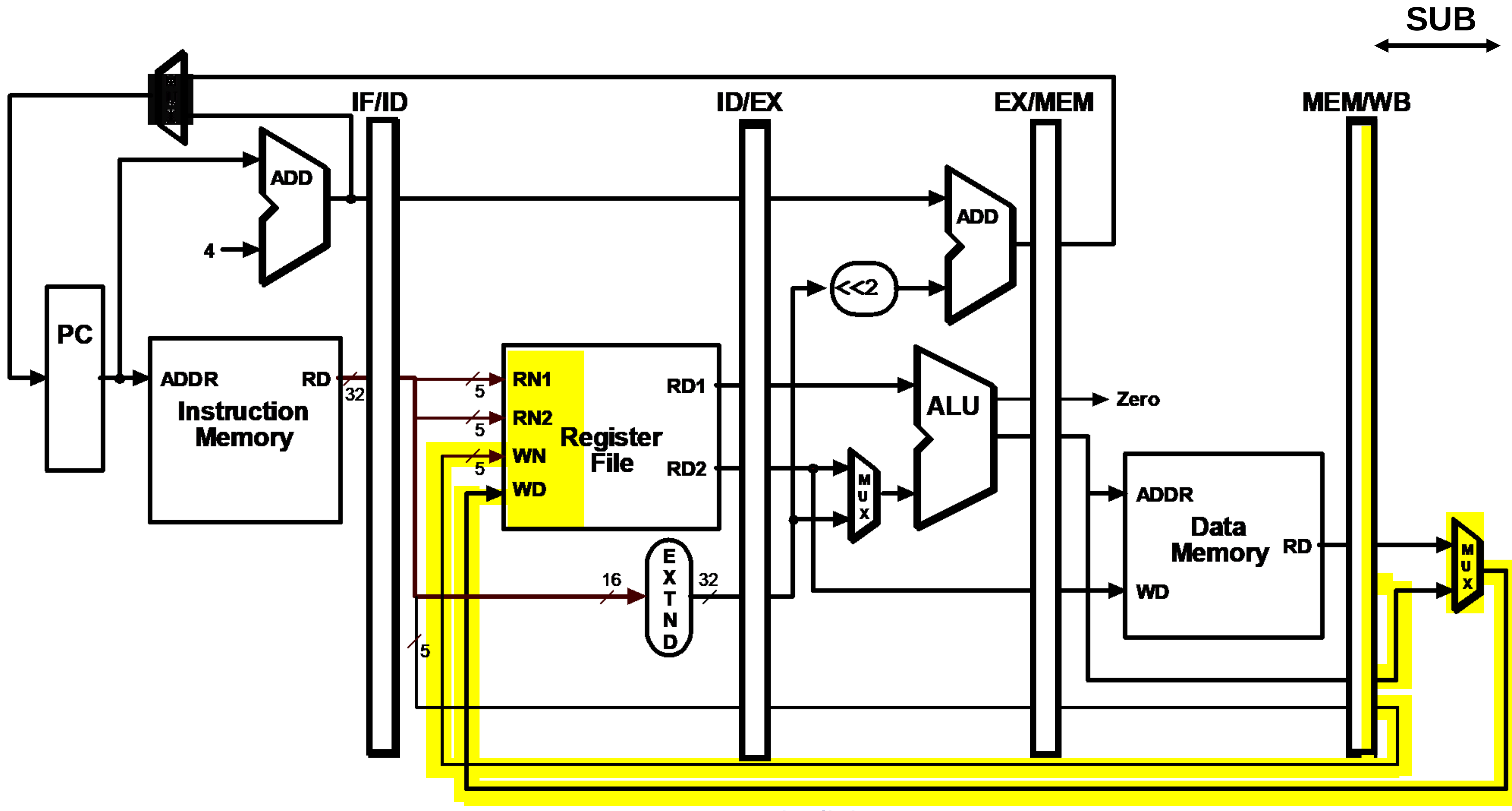
Executing Multiple Instructions Clock Cycle 7

```
lw $r0, 10($r1)
sw $r3, 20($r4)
add $r5, $r6, $r7
sub $r8, $r9, $r10
```

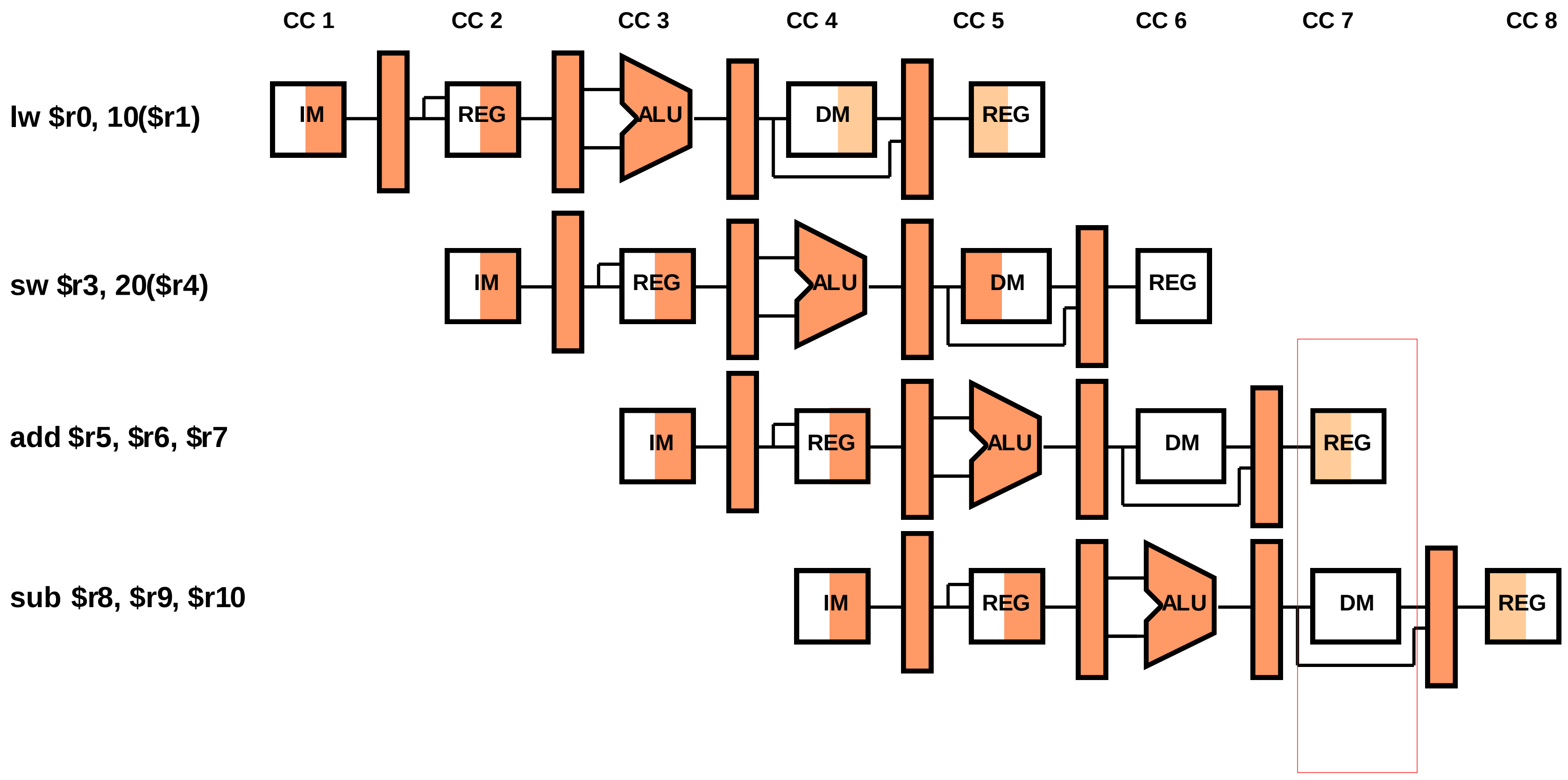


Executing Multiple Instructions Clock Cycle 8

```
lw $r0, 10($r1)
sw $r3, 20($r4)
add $r5, $r6, $r7
sub $r8, $r9, $r10
```



Alternative View - Multicycle Diagram



Hazards

Pipelining Outline

- Introduction
  - Defining Pipelining
  - Pipelining Instructions
  - Hazards ◀
- Pipelined Processor Design
  - Datapath
  - Control
- Advanced Pipelining
  - Superscalar
  - Dynamic Pipelining
  - Examples

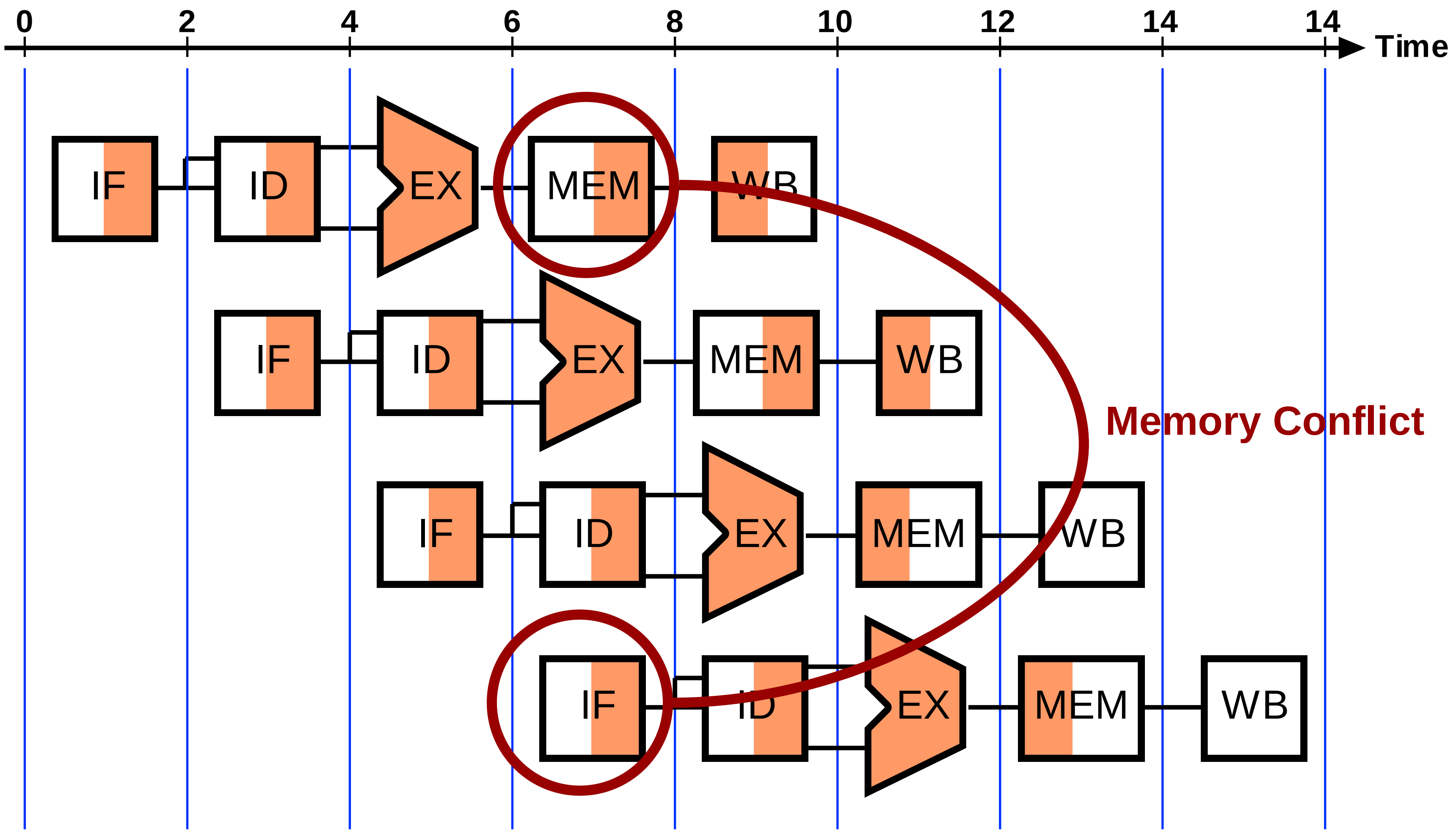
Pipeline Hazards

- Where one instruction cannot immediately follow another
- Types of hazards
  - Structural hazards - attempt to use same resource twice
  - Control hazards - attempt to make decision before condition is evaluated
  - Data hazards - attempt to use data before it is ready
- Hazards can be resolved by waiting ("stalling")

Structural Hazards

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

Example Structural Hazard - Single Memory



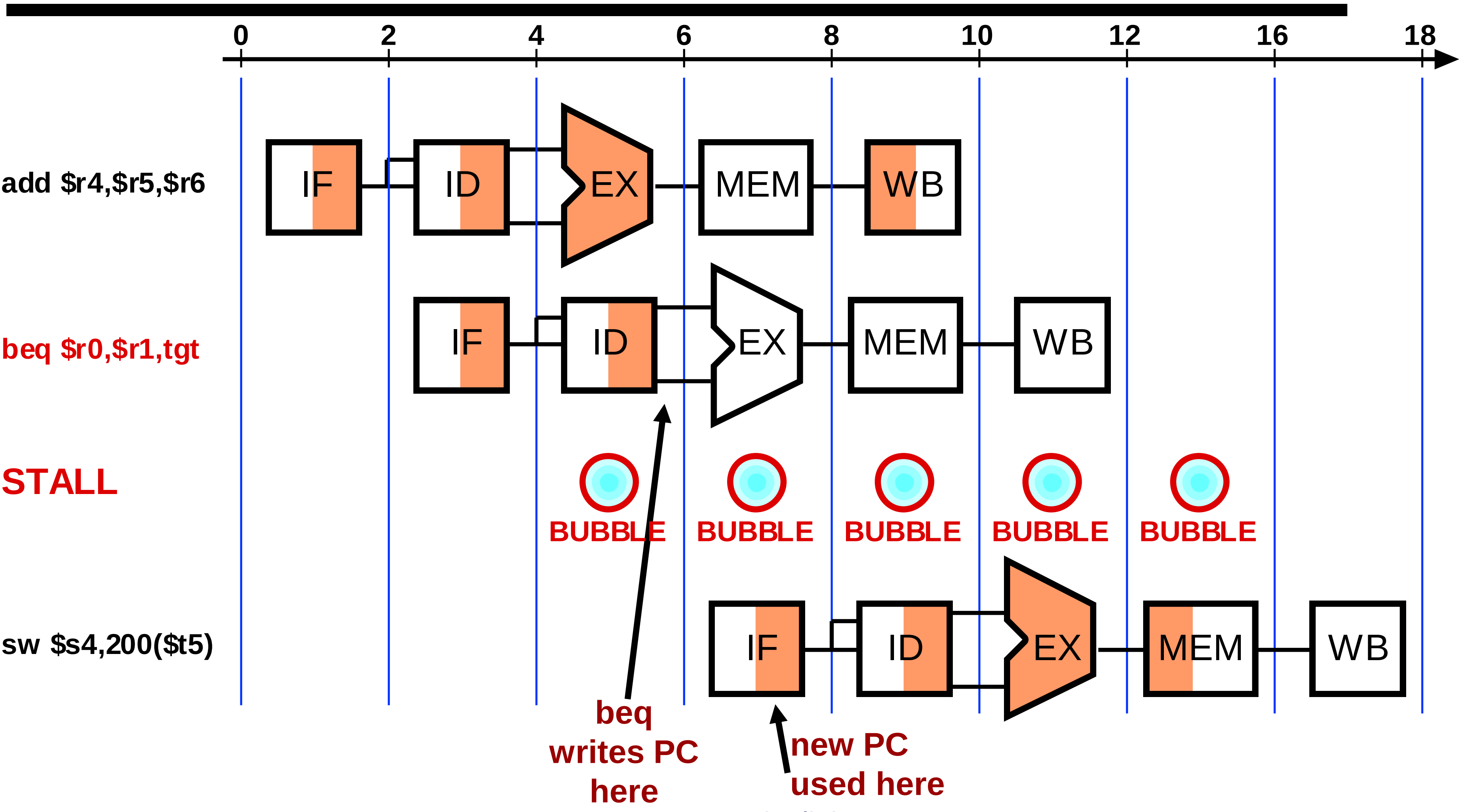
Control Hazards

- Attempt to make a decision before condition is evaluated
- Example: `beq $s0, $s1, offset`
- Assume we add hardware to second stage to:
  - Compare fetched registers for equality
  - Compute branch target
- This allows branch to be taken at end of second clock cycle
- But, this still means result is not ready when we want to load the next instruction!

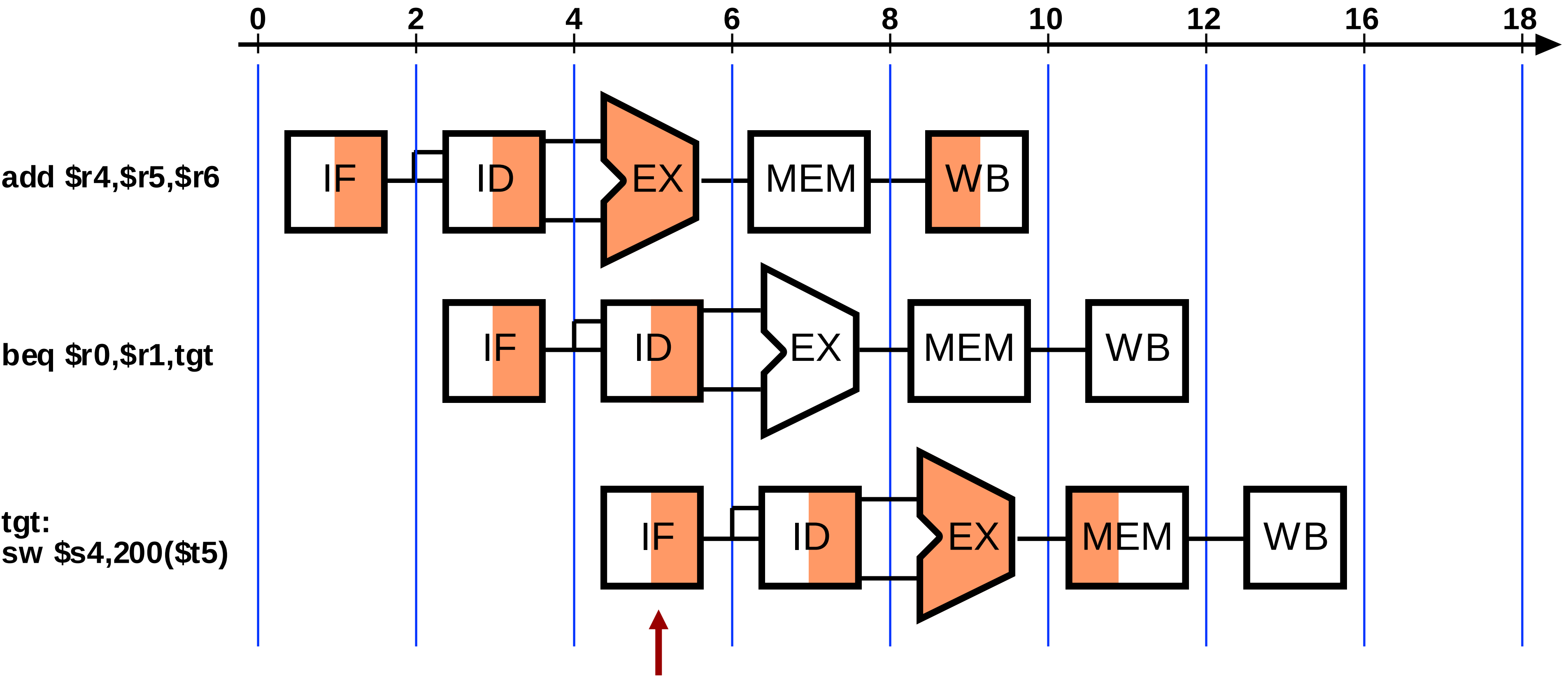
Control Hazard Solutions

- **Stall** - stop loading instructions until result is available
- **Predict** - assume an outcome and continue fetching (undo if prediction is wrong)
- **Delayed branch** - specify in architecture that following instruction is always executed

Control Hazard - Stall

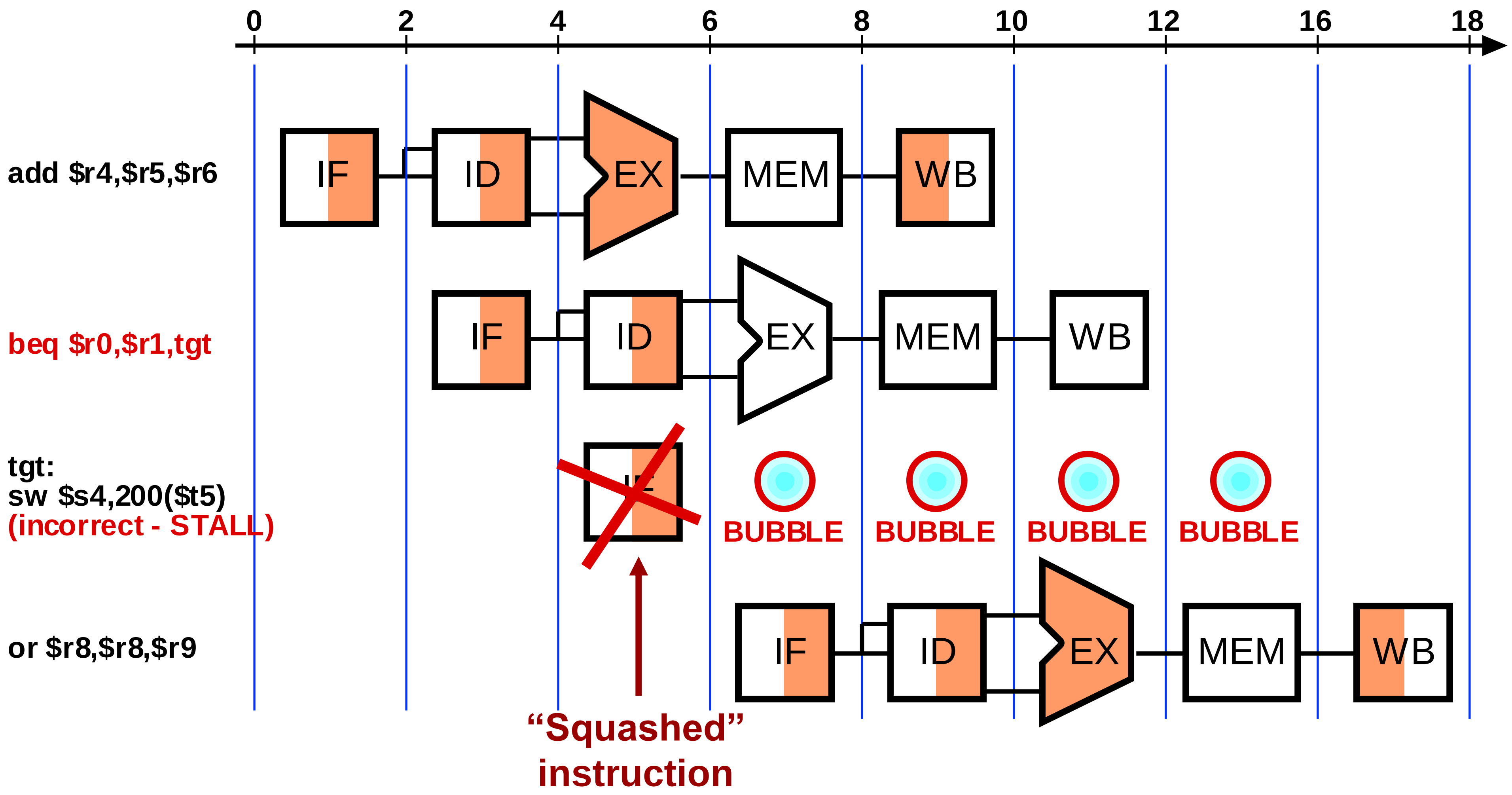


Control Hazard - Correct Prediction



Fetch assuming branch taken

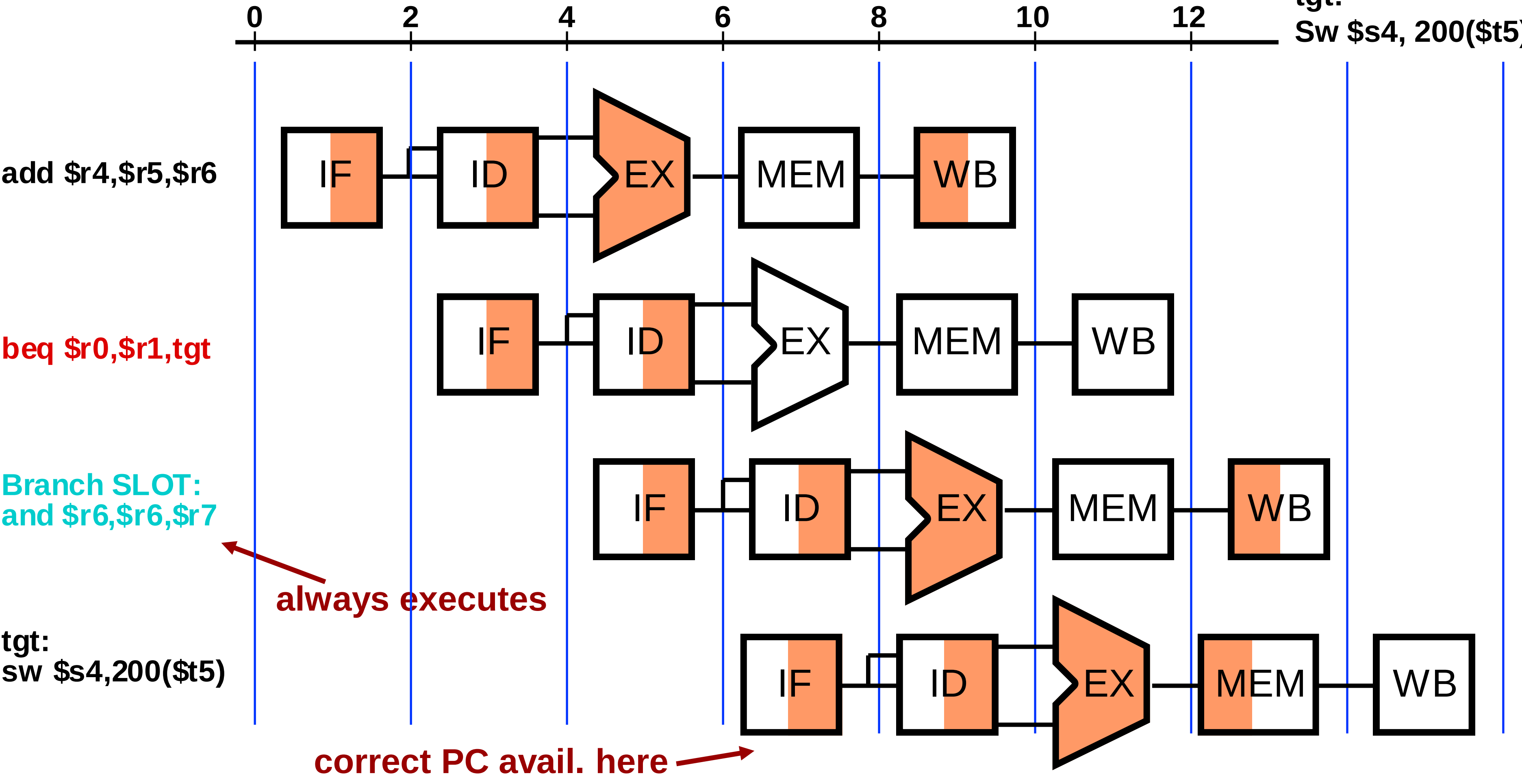
Control Hazard - Incorrect Prediction



Control Hazard - Delayed Branch

add $r4, $r5, $r6
and $s6, $s6, $r7
beq $r0, $r1, tgt

tgt:
Sw $s4, 200($t5)



Summary - Control Hazard Solutions

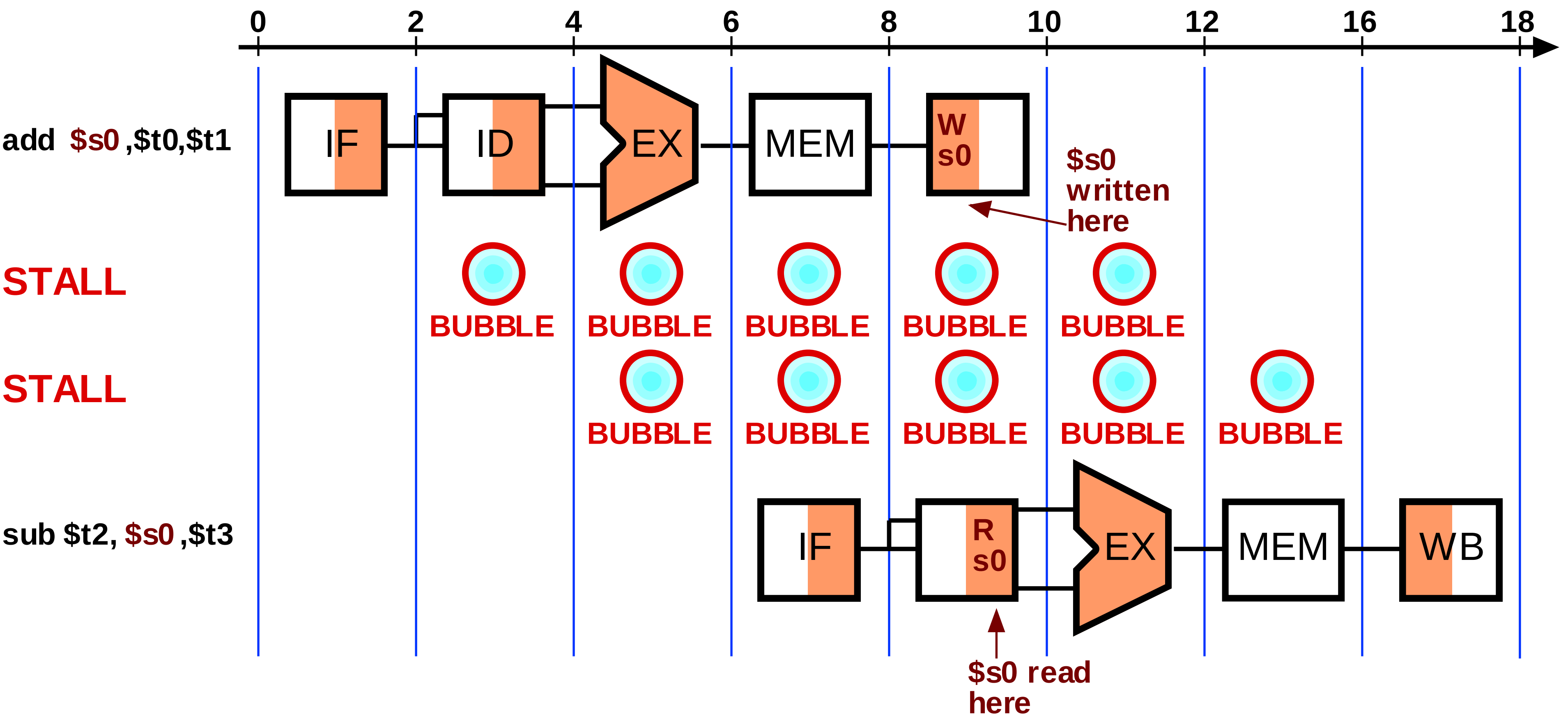
- **Stall** - stop fetching instr. until result is available
  - Significant performance penalty
  - <u>Hardware required to stall</u>
- **Predict** - assume an outcome and continue fetching (undo if prediction is wrong)
  - Performance penalty only when guess is wrong
  - <u>Hardware required to "squash" instructions</u>
- **Delayed branch** - specify in architecture that following instruction is always executed
  - Compiler <u>re-orders instructions into delay slot</u>
  - Insert "NOP" (no-op) operations when can't use (~50%)
    - This is how original MIPS worked
  - But, rationale (이론근거) may not hold up as technology changes

Data Hazards

- Attempt to use data before it is ready
- Solutions
  - Stalling - wait until result is available
  - Forwarding - make data available inside datapath
  - Reordering instructions - use compiler to avoid hazards
- Examples:

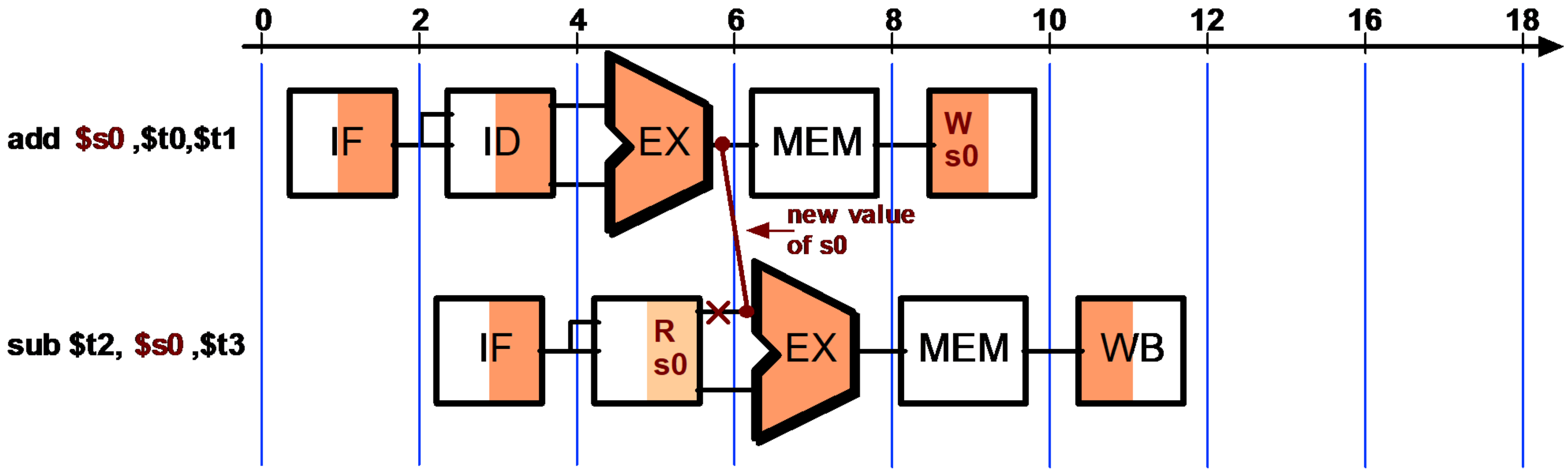
```
add $s0, $t0, $t1 ; $s0 = $t0+$t1
sub $t2, $s0, $t3 ; $t2 = $s0-$t2

lw $s0, 0($t0)      ; $s0 = MEM[$t0]
sub $t2, $s0, $t3 ; $t2 = $s0-$t2
```

Data Hazard - Stalling



Data Hazards - Forwarding

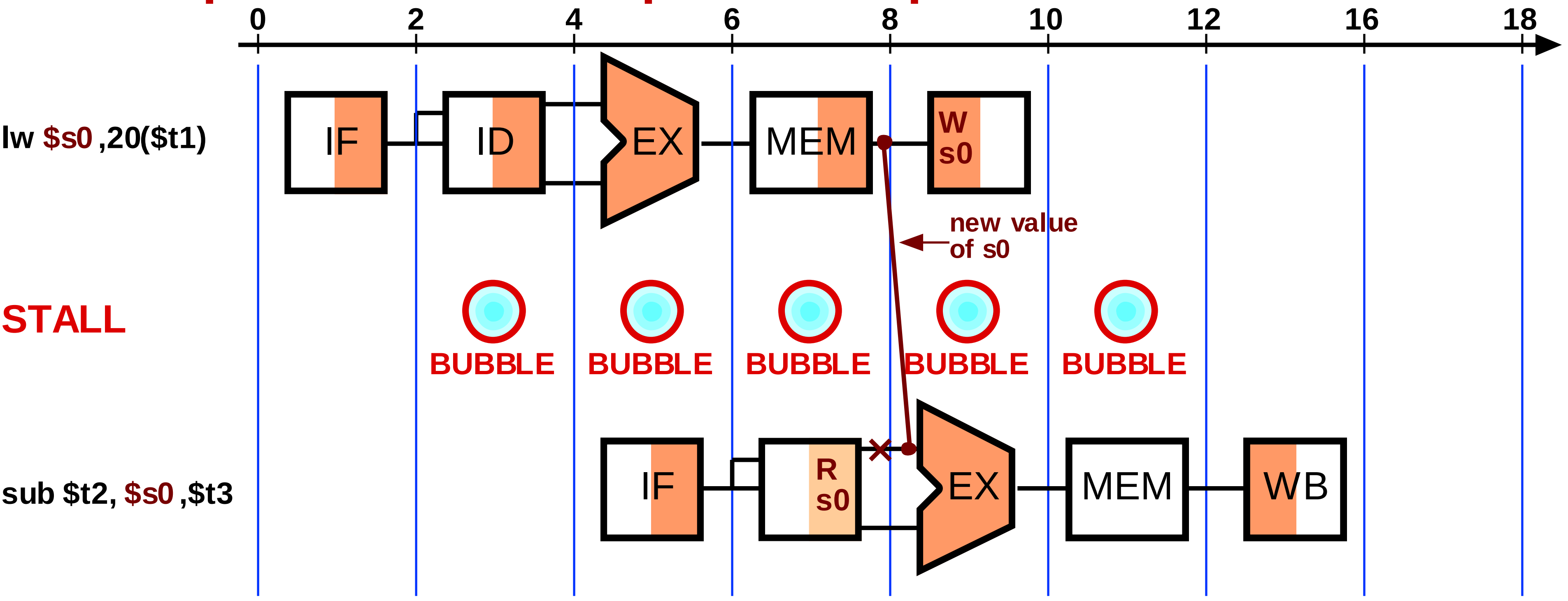
- Key idea: connect new value directly to next stage
- Still read s0, but ignore in favor of new result



- Problem: what about load instructions?

Data Hazards - Forwarding

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



Data Hazards - Reordering Instructions

- Assuming we have data forwarding, what are the hazards in this code?

```
lw $t0, 0($t1)
lw $t2, 4($t1)
sw $t2, 0($t1)
sw $t0, 4($t1)
```

- Reorder instructions to remove hazard:

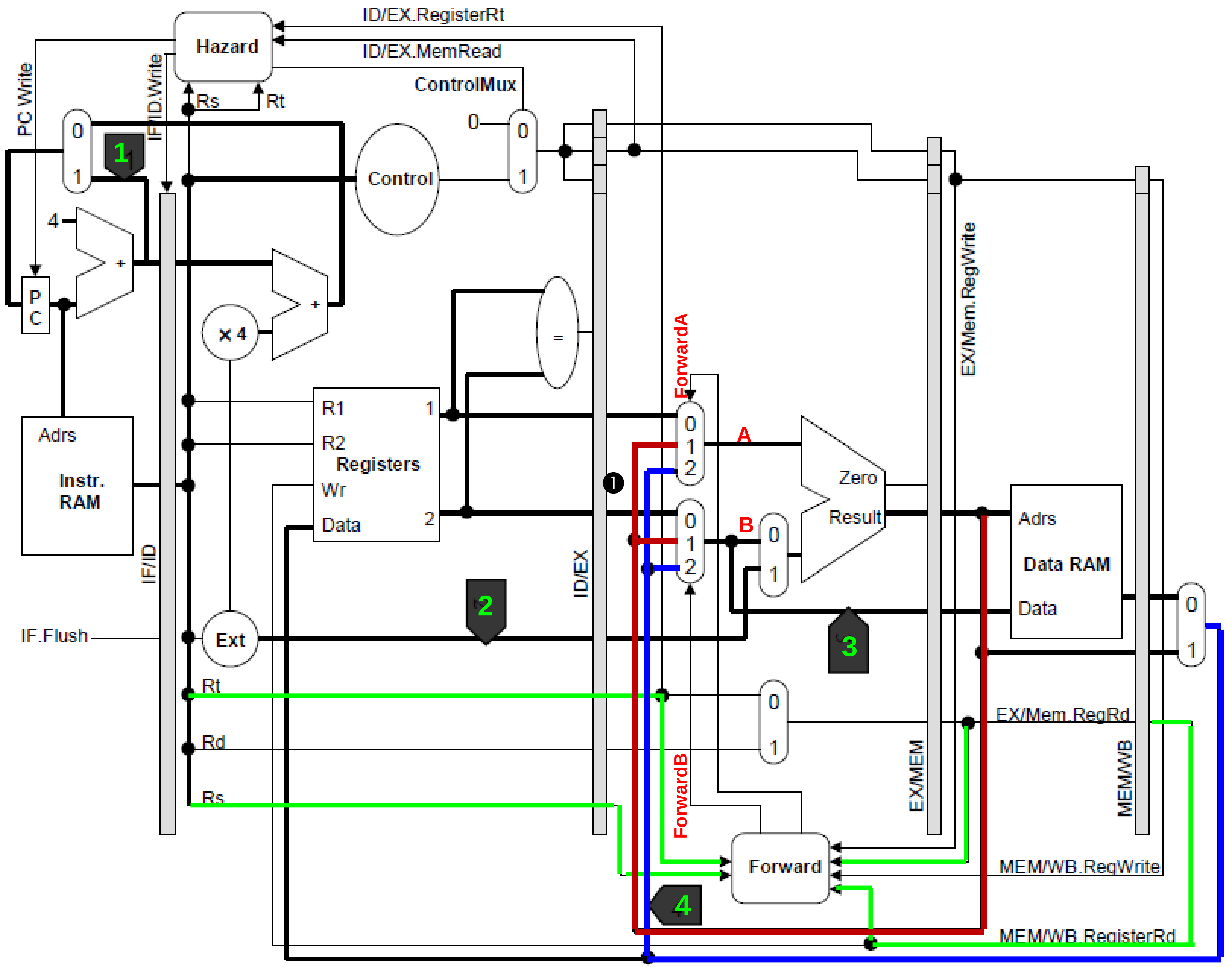
```
lw $t0, 0($t1)
lw $t2, 4($t1)
sw $t0, 4($t1)
sw $t2, 0($t1)
```

Summary - Pipelining Overview

- Pipelining increase <u>throughput</u> (but not latency)
- Hazards limit performance
  - Structural hazards
  - Control hazards
  - Data hazards

Pipelining Outline - Coming Up

- Introduction
  - Defining Pipelining
  - Pipelining Instructions
  - Hazards
- Pipelined Processor Design ◀
  - Datapath
  - Control
- Advanced Pipelining
  - Superscalar
  - Dynamic Pipelining
  - Examples



Hazard
ID/EX.RegisterRt
ID/EX.MemRead
ControlMux
PC Write
IF/ID.Write
Rs
Rt
0
1
1
Control
0
0
1
4
+
P C
+
× 4
=
EX/Mem.RegWrite
ForwardA
Adrs
Instr. RAM
R1
R2
Registers
Wr
Data
1
2
0
1
2
A
Zero
Result
IF/ID
ID/EX
0
1
2
B
0
1
Adrs
Data RAM
Data
2
3
0
1
IF.Flush
Ext
Rt
Rd
Rs
0
1
EX/Mem.RegRd
ForwardB
EX/MEM
MEM/WB
Forward
MEM/WB.RegWrite
4
MEM/WB.RegisterRd