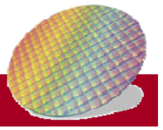
Chapter 4

Pipelined Processor

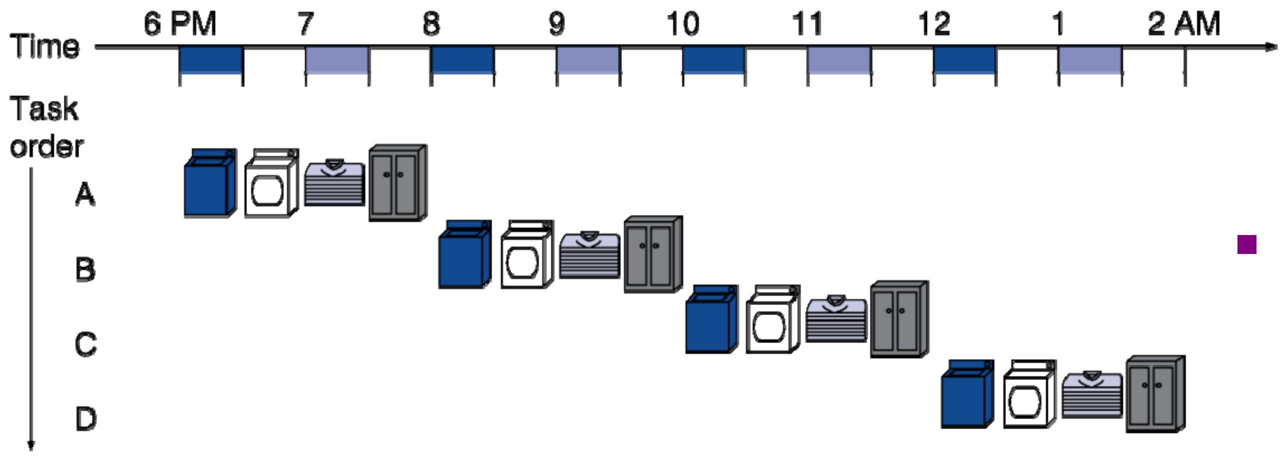


Outline

- A pipelined datapath
- Pipelined control
- Data hazards and forwarding
- Data hazards and stalls
- Branch hazards

Pipelining Overview

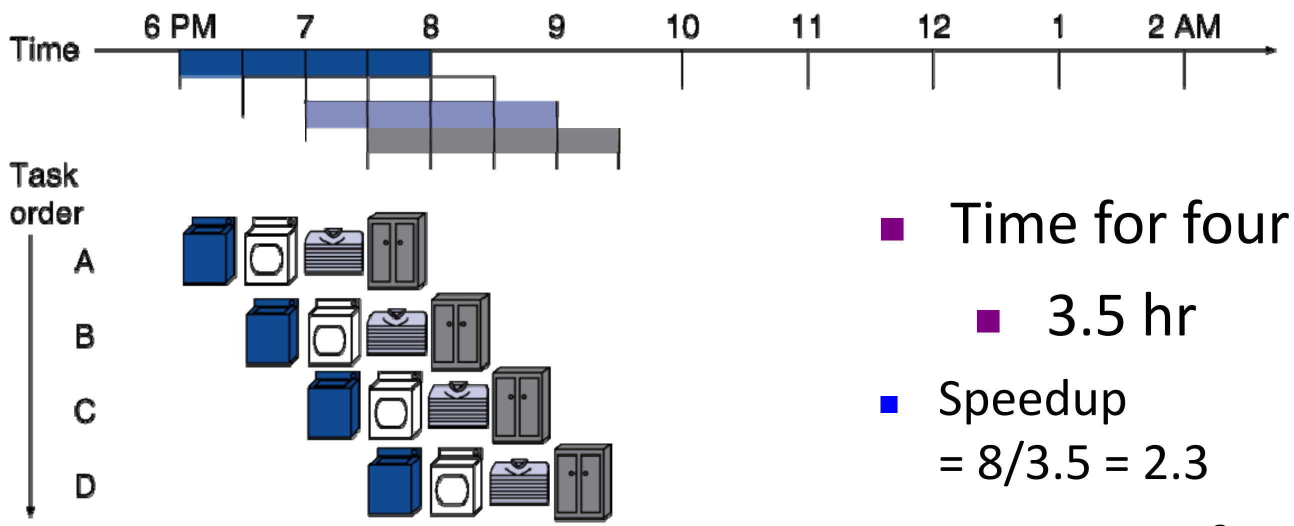
- Assume four steps to do laundry, each one take 30min
  - Place the clothes in the washer
  - Place the clothes in the dryer
  - Fold the clothes
  - Ask roommate to put the clothes away
- Nonpipelined laundry: one at a time



- Time for four load
  - 2 hr *4 = 8 hr

Pipelining Analogy

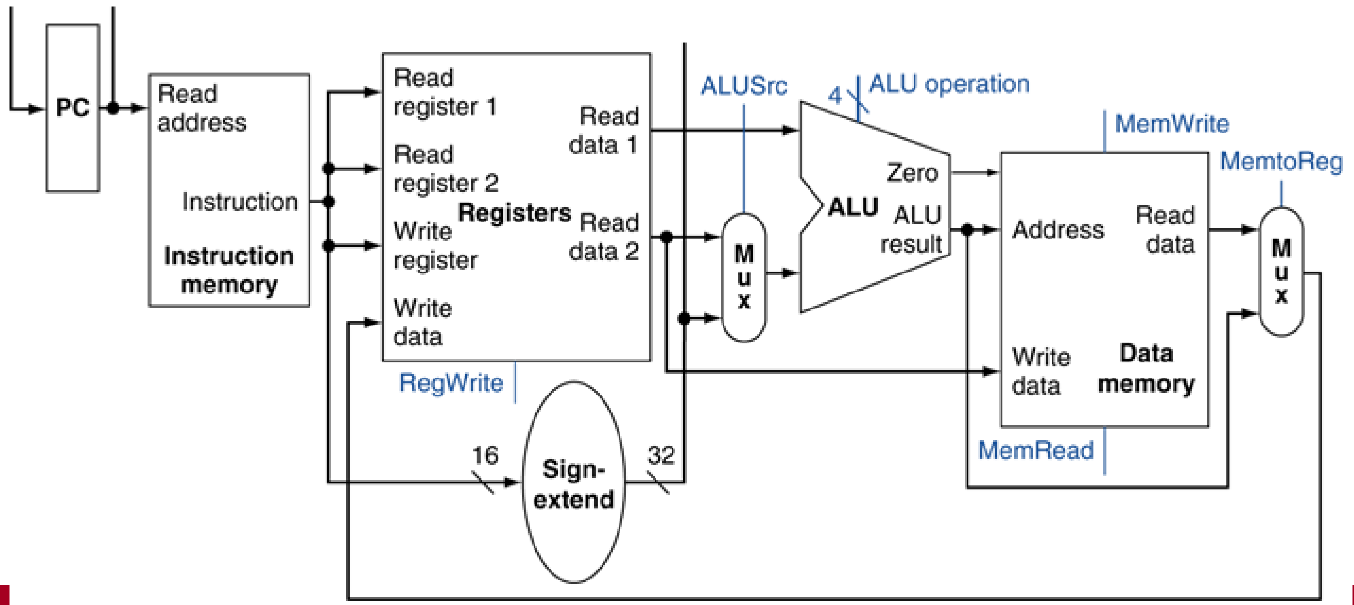
- Pipelined laundry: overlap the task
  - Parallelism improves performance



- Time for four loads
  - 3.5 hr
- Speedup
  = 8/3.5 = 2.3
- Non-stop for a lot of loads
  - Speedup
    = 2n/0.5n + 1.5 ≈ 4
    = number of stages

MIPS Pipeline

- Five stages, one step per stage
  1. IF: Instruction fetch from memory
  2. ID: Instruction decode & register read
  3. EX: Execute operation or calculate address
  4. MEM: Access memory operand (for lw and sw instruction)
  5. WB: Write result back to register



Pipeline Performance

- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages
- Time comparision between pipelined with single-cycle datapath

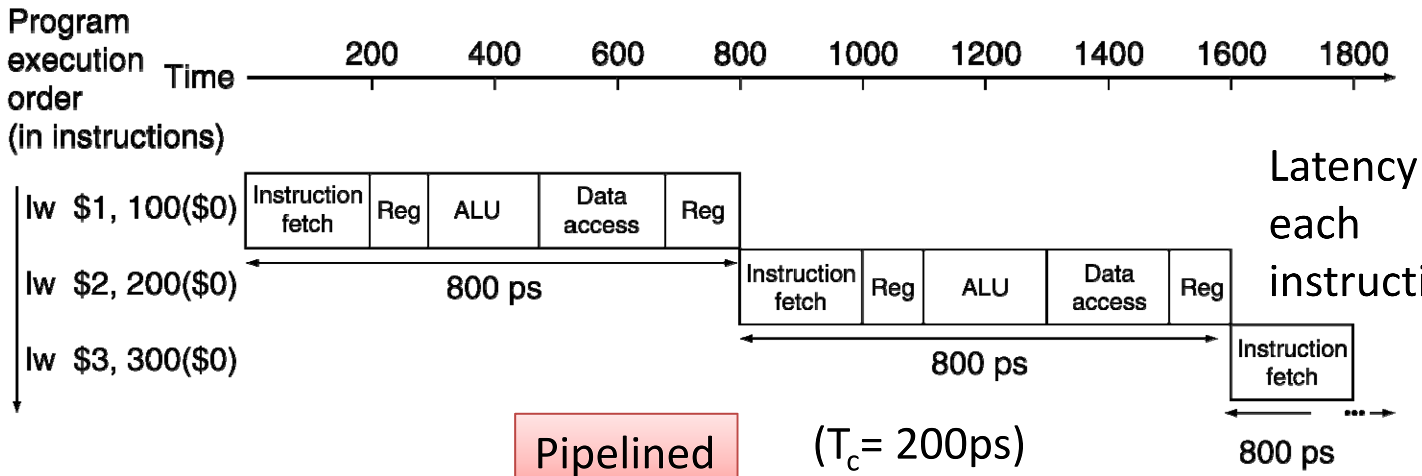
Longest data path: 800ps

Required time for single-cycle datapath

Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

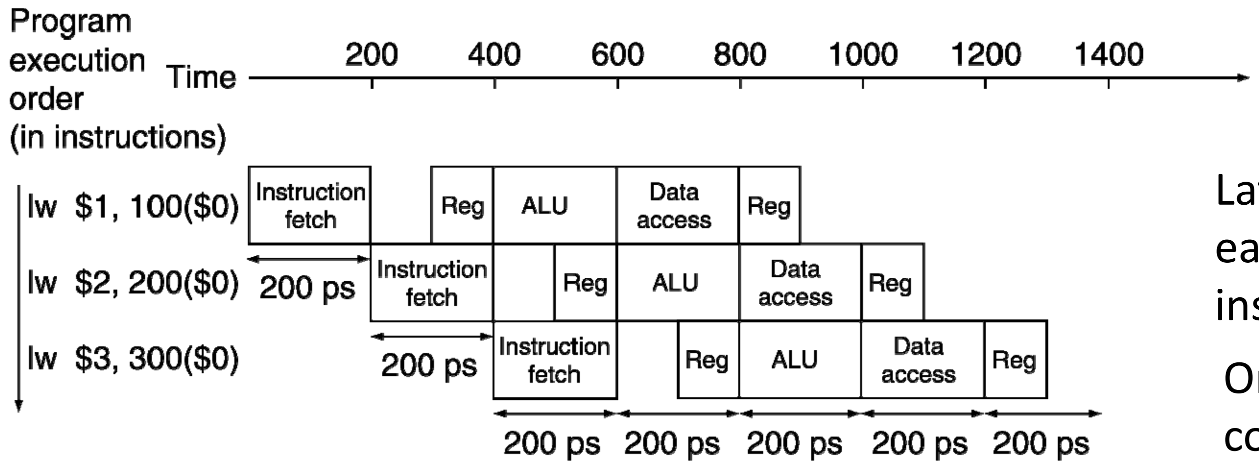
Pipeline Performance

Single-cycle (T$_c$= 800ps)



Latency for each instruction 800ps

Pipelined (T$_c$= 200ps)



Latency for each instruction 1000ps

One instruction is completed every 200ps

Pipeline Speedup

- Stage may be unbalanced: may waste time in a stage
- If all stages are balanced (all take the same time)

$$Time\ Between\ instructions_{pipelined} = \frac{Time\ between\ instruction_{nonpipelined}}{Number\ of\ Stages}$$

200=1000/5

- Speedup due to increased throughput
  - Latency (time for each instruction) does not decrease
  - Latency is 1000 ps, but one instruction is completed every 200ps

Pipelining and ISA Design

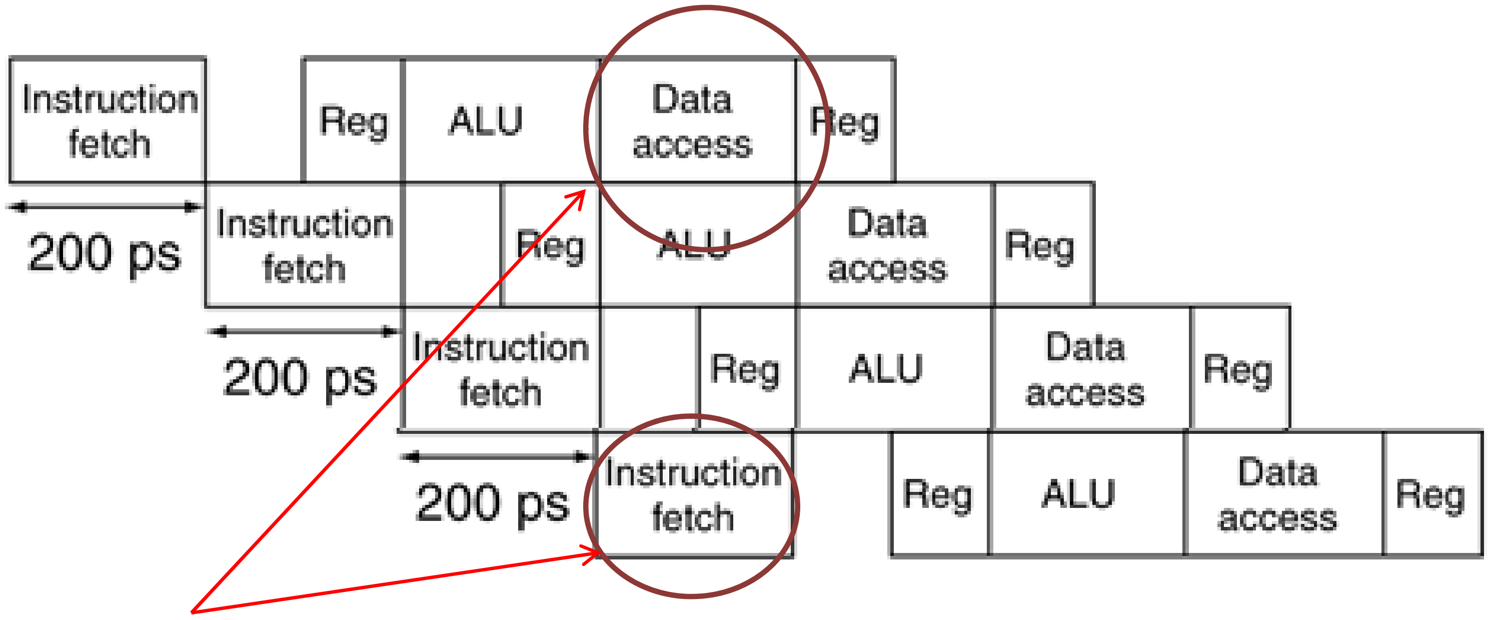
- MIPS ISA designed for pipelining
  - All instructions are 32-bits
    - Easier to fetch and decode in one cycle
    - c.f. x86: 1- to 17-byte instructions
  - Few and regular instruction formats
    - Can decode and read registers in one step
  - Load/store addressing
    - Can calculate address in 3rd stage, access memory in 4th stage
  - Alignment of memory operands
    - Memory access takes only one cycle

Pipeline Hazards

- Hazards are situations that prevent starting the next instruction in the next cycle
- Structure hazards
  - A required resource is busy. Need to wait until the resource is ready
  - E.g. instruction and data memory are separated to avoid structure hazards (see next slide)
- 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

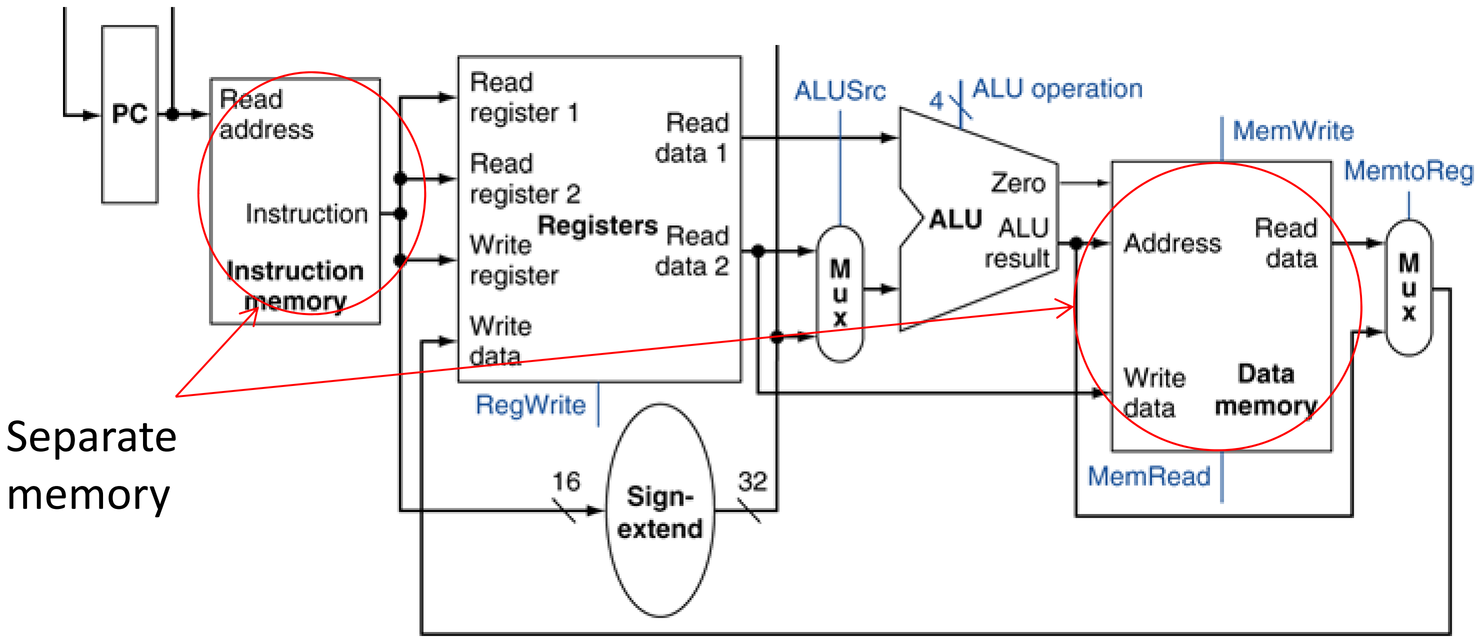
- Conflict for use of a resource
- In MIPS pipeline with a single memory
  - Load/store and Instruction fetch both require memory access
  - *Stall* will occur if both happen in the same cycle
  - Structure Hazards



Conflict , both need to access memory at the same time

Solution for Structure Hazards

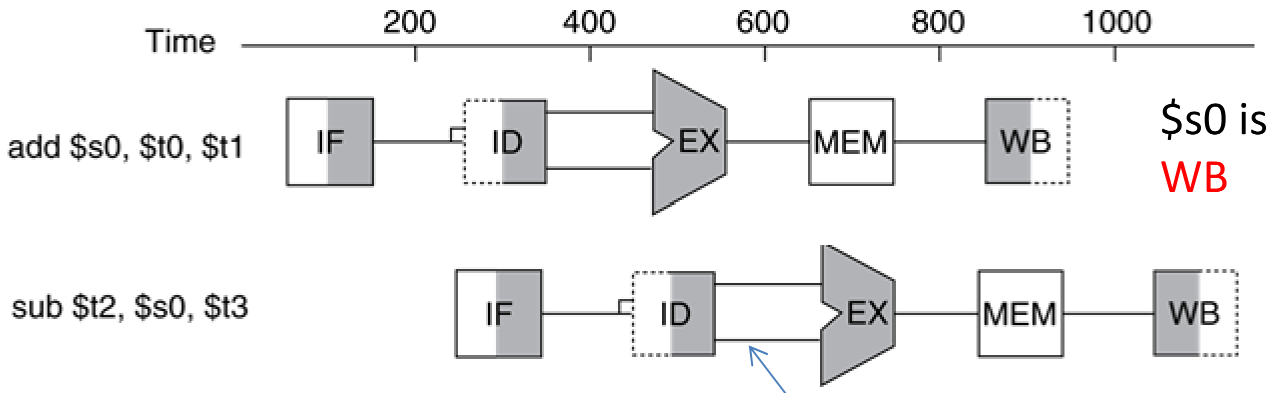
- Hence, pipelined datapaths require separate instruction/data memories
  - Or separate instruction/data caches



Data Hazards

- An instruction depends on completion of data access by a previous instruction

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



$s0 is ready after WB

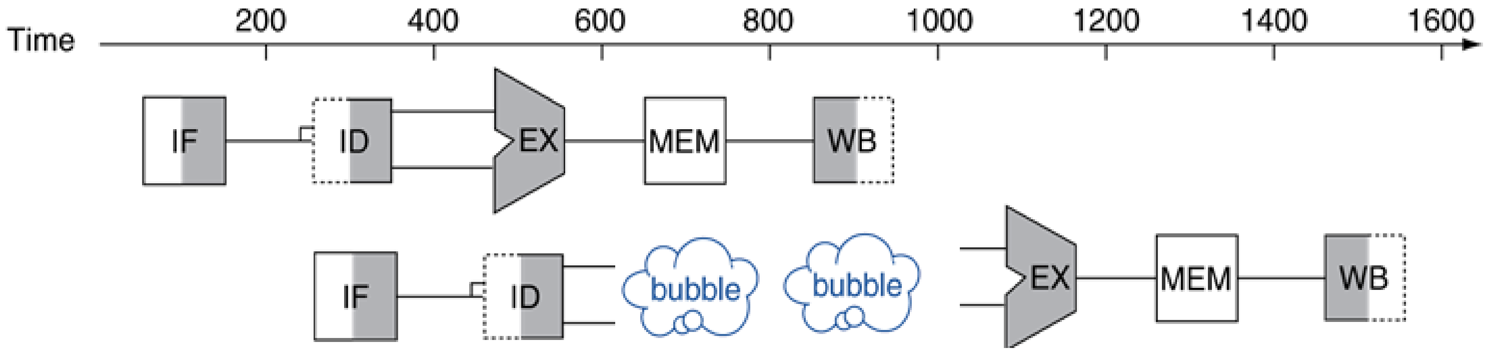
$s0 is incorrect, so the result is incorrect

Data Hazards causes stall

- Instruction stall for cycles – waste cycle time
  - Also called pipeline "bubble"

add $s0, $t0, $t1

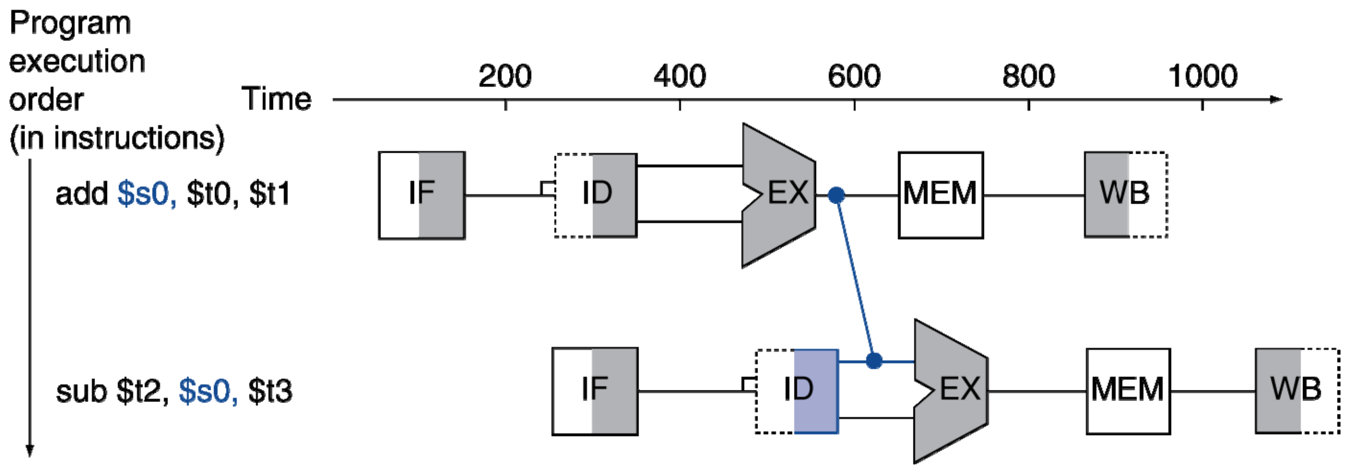
sub $t2, $s0, $t3



2 bubbles(nop) are added
Can we have a better solution?

Data Hazard Solution 1 - Forwarding

- Use result when it is computed
  - Don't wait for it to be stored in a register
  - Requires extra connections in the datapath
  - Also call bypassing

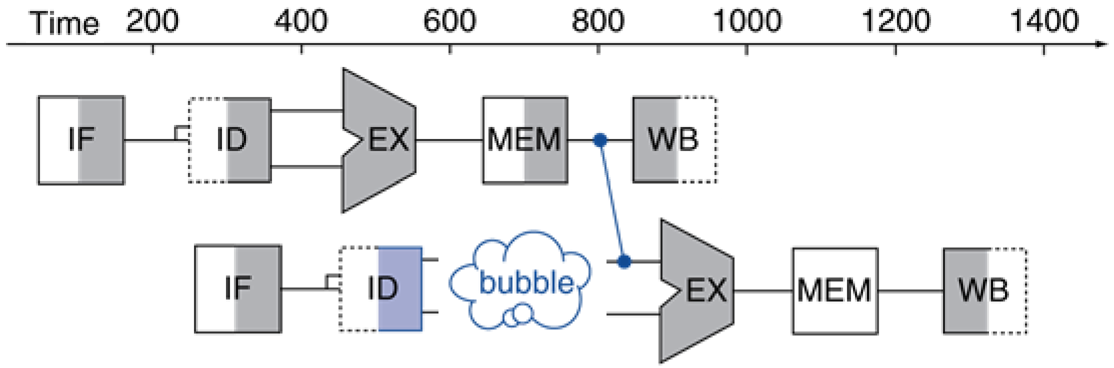


Load-Use Data Hazard

- Forwarding can't always avoid stalls
  - If value is not ready when needed
  - Can't forward backward in time!
  - For example, the lw instruction, data is ready after memory access

lw $s0, 20($t1)

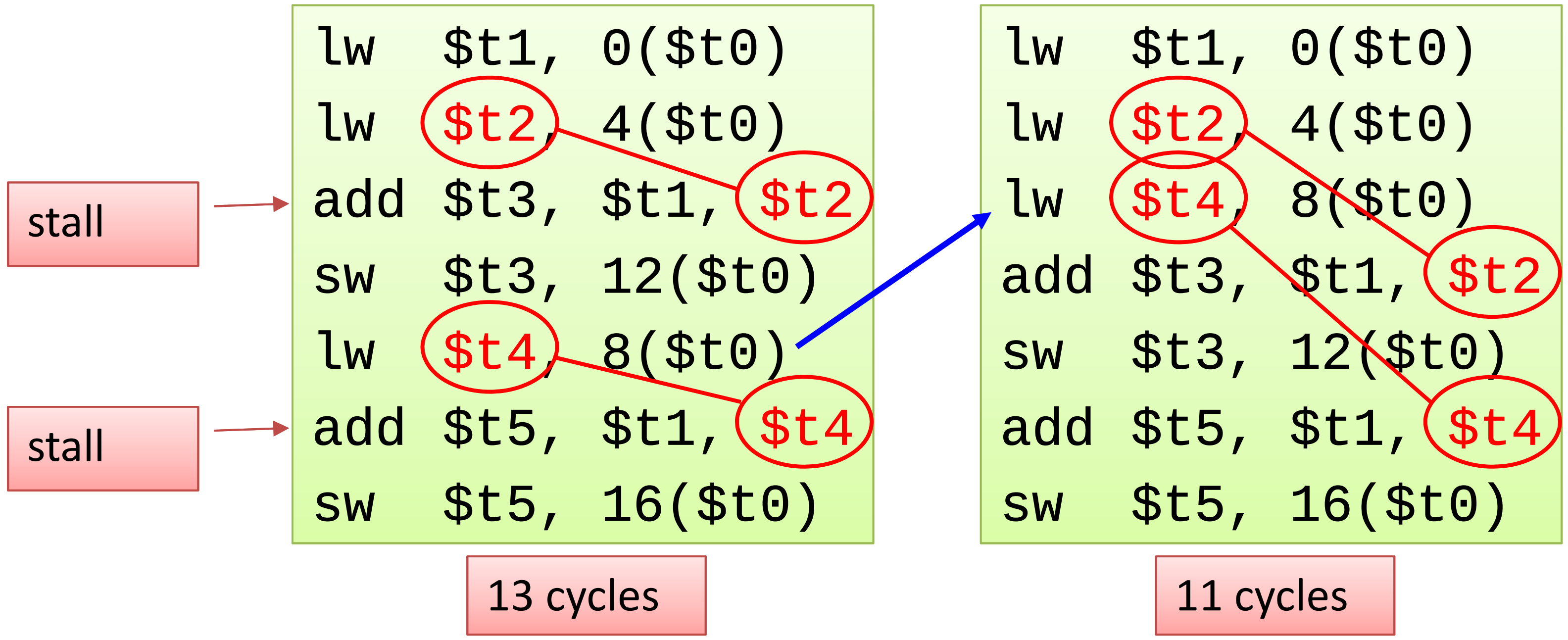
sub $t2, $s0, $t3



One extra cycle is necessary

Data hazard solution 2: Code Scheduling

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



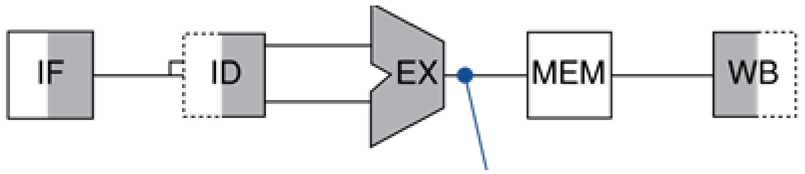
```
lw   $t1, 0($t0)
lw   $t2, 4($t0)
add  $t3, $t1, $t2
sw   $t3, 12($t0)
lw   $t4, 8($t0)
add  $t5, $t1, $t4
sw   $t5, 16($t0)
```

```
lw   $t1, 0($t0)
lw   $t2, 4($t0)
lw   $t4, 8($t0)
add  $t3, $t1, $t2
sw   $t3, 12($t0)
add  $t5, $t1, $t4
sw   $t5, 16($t0)
```

Control Hazards

- Branch determines flow of control
  - Fetching next instruction depends on branch outcome
  - Result of Beq is known at the EX stage, but still working on ID stage of branch
    - Can't determine the next instruction

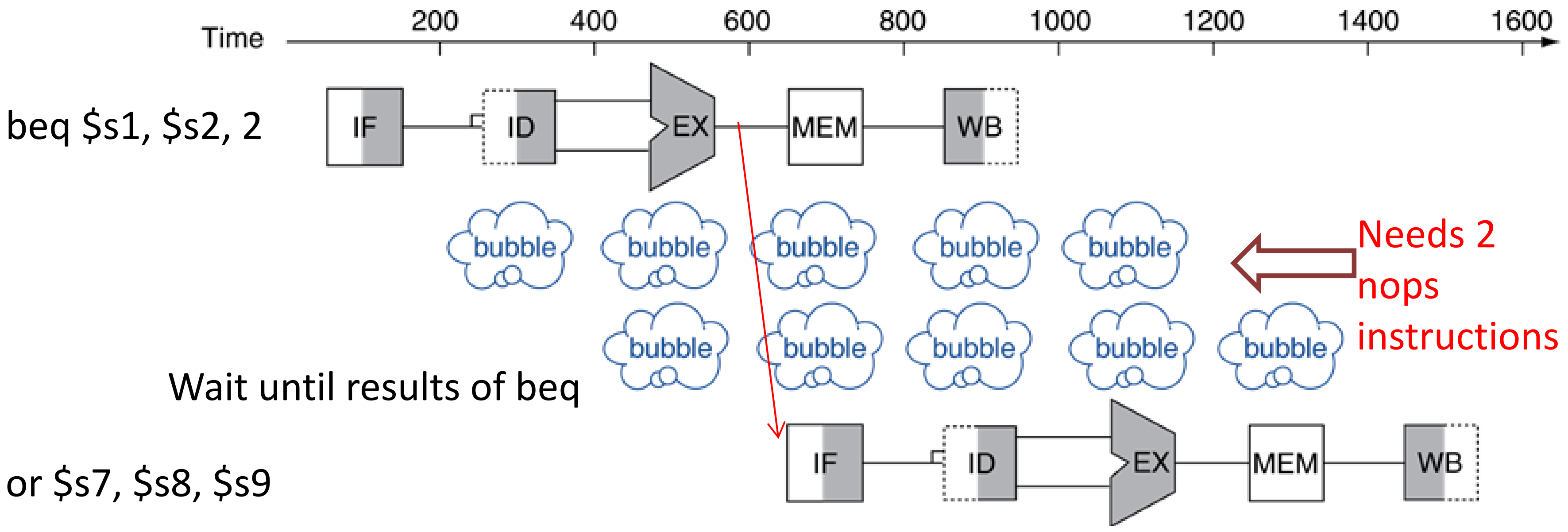
```
beq $s3, $s4, Else
```



Result of Beq is determined over here.

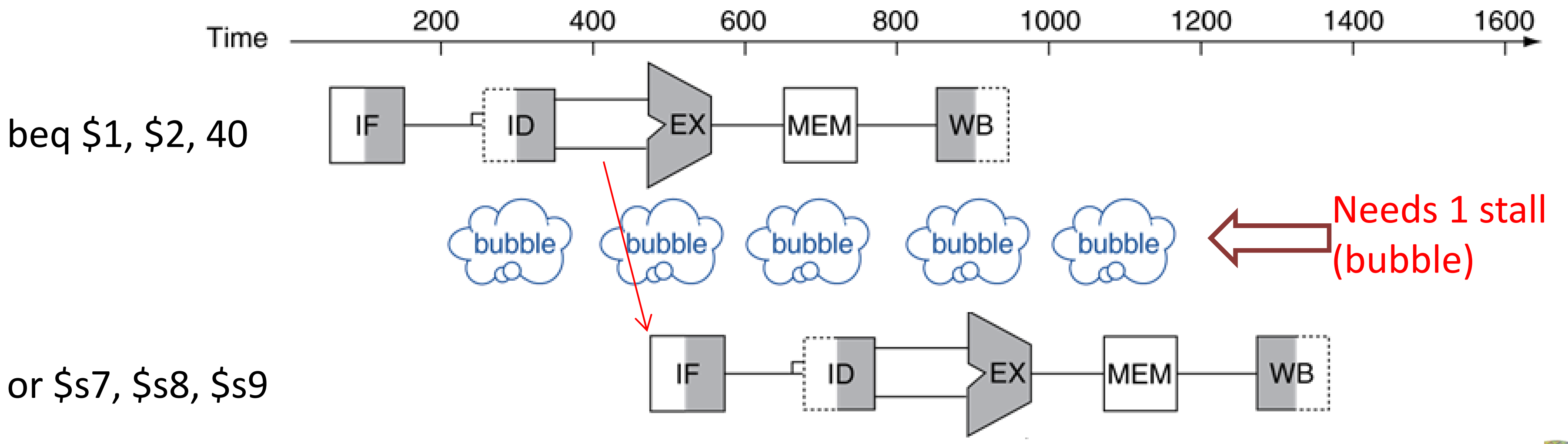
Control Hazard causes stalls (bubble)

- Branch determines flow of control
  - Fetching next instruction depends on branch outcome
  - Result of Beq is known at the EX stage
- Wait until branch outcome determined before fetching next instruction



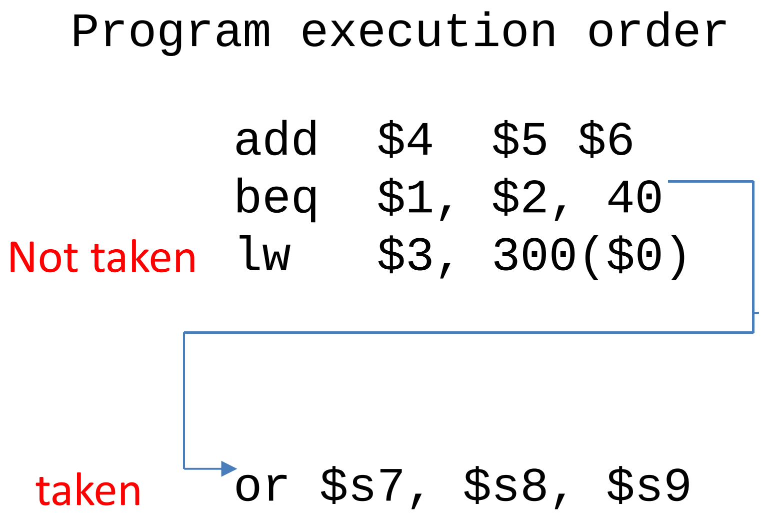
Control Hazard Solution 1: Add hardware in ID stage

- Compare registers and compute target early in the pipeline (Details are in § 4.8)
- Extra hardware to test registers, calculate the branch address, and update the PC during the ID stage of the pipeline => stall is reduced to only one



Control Hazard Solution 2: Branch Prediction

- Predict outcome of branch
  - Correct prediction, no bubble is added
  - Incorrect prediction, clean incorrect and add bubble
- Example: predict branches not taken
  - Fetch instruction after branch, with no delay
  - If prediction is correct, no stall
  - If prediction is incorrect, ignore the incorrect result, and load correct instruction



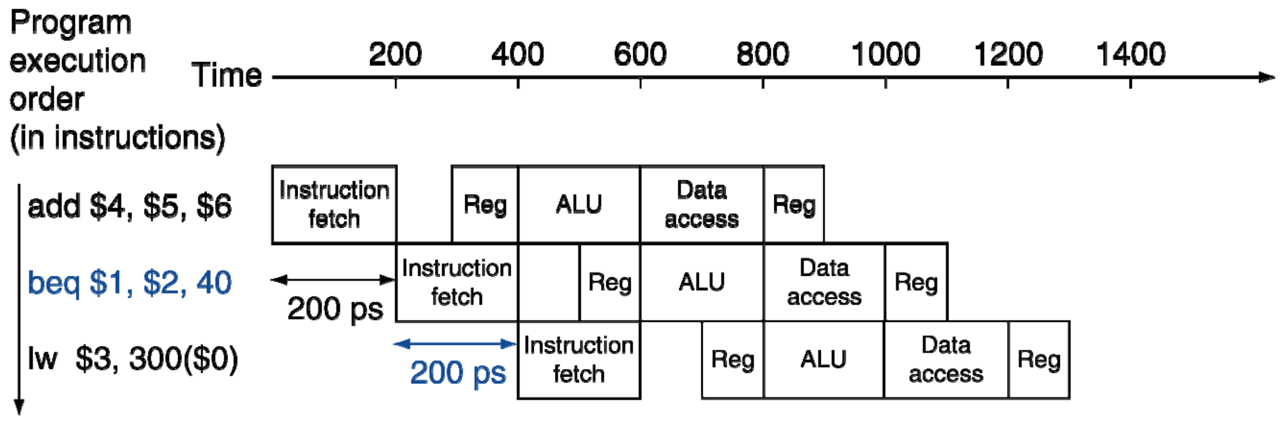
```
add   $4   $5 $6
beq   $1,  $2, 40
lw    $3,  300($0)

or $s7, $s8, $s9
```

See next slide for figures, and more details in § 4.8

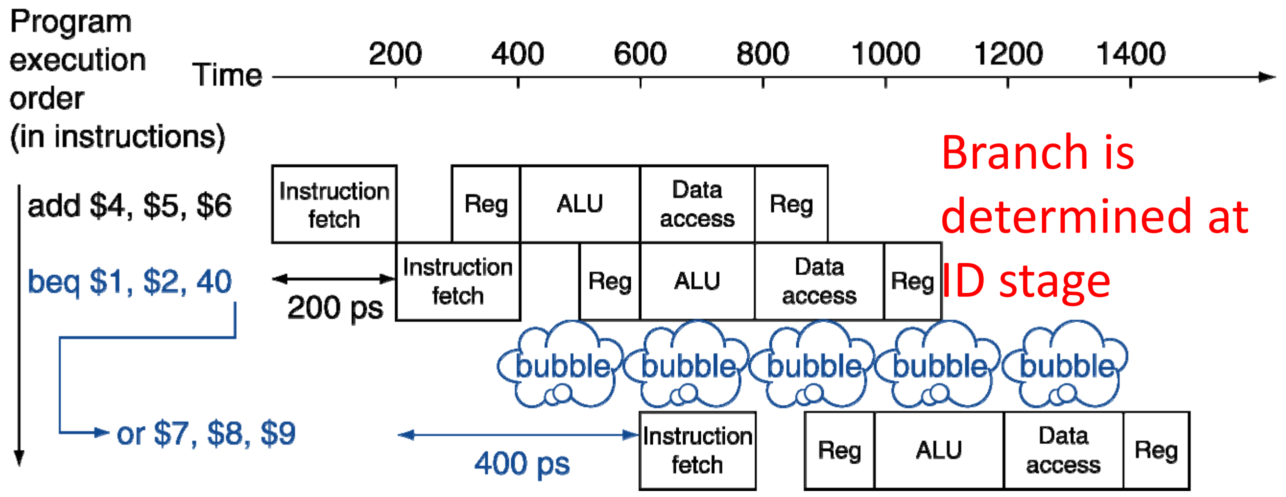
MIPS with Predict Not Taken

Correct Prediction



No bubble

Incorrect Prediction



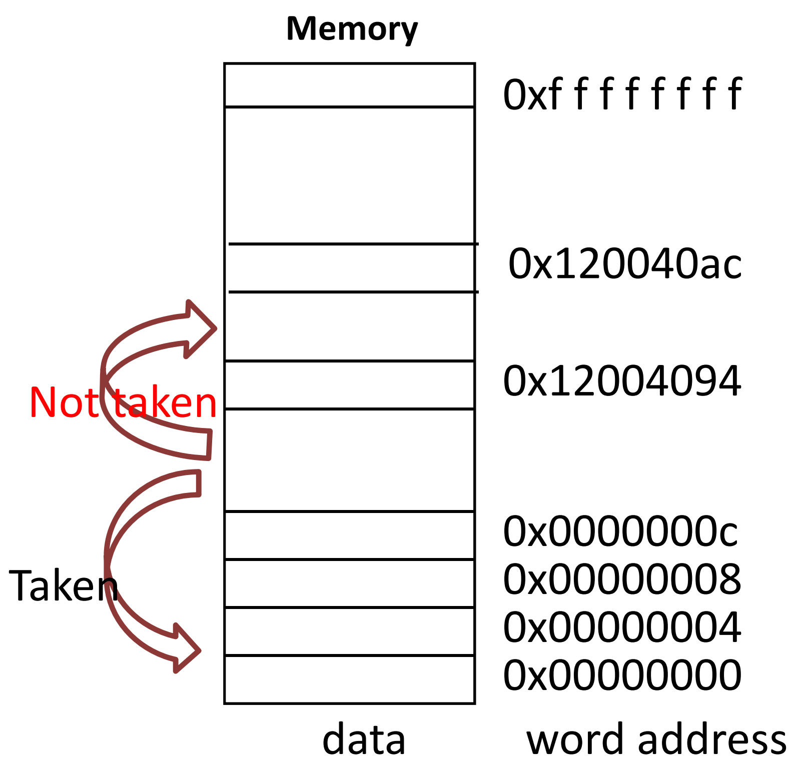
Branch is determined at ID stage

one bubble

Improve Branch Prediction Accuracy

- Static branch prediction
  - Based on typical branch behavior
  - Example: loop and if-statement branches
    - backward branches => predict taken
    - forward branches =>not taken
- Dynamic branch prediction
  - Hardware measures actual branch behaviors
  - Perdition based on recent histories of each branch

Taken, Taken, Taken, Taken



What is the next prediction?
Taken for Not Taken ?

Control Hazard Solution 3: Delayed Branch

- Idea: Instruction after the branch is always executed → Insert an instruction that must be executed →

  stall is removed

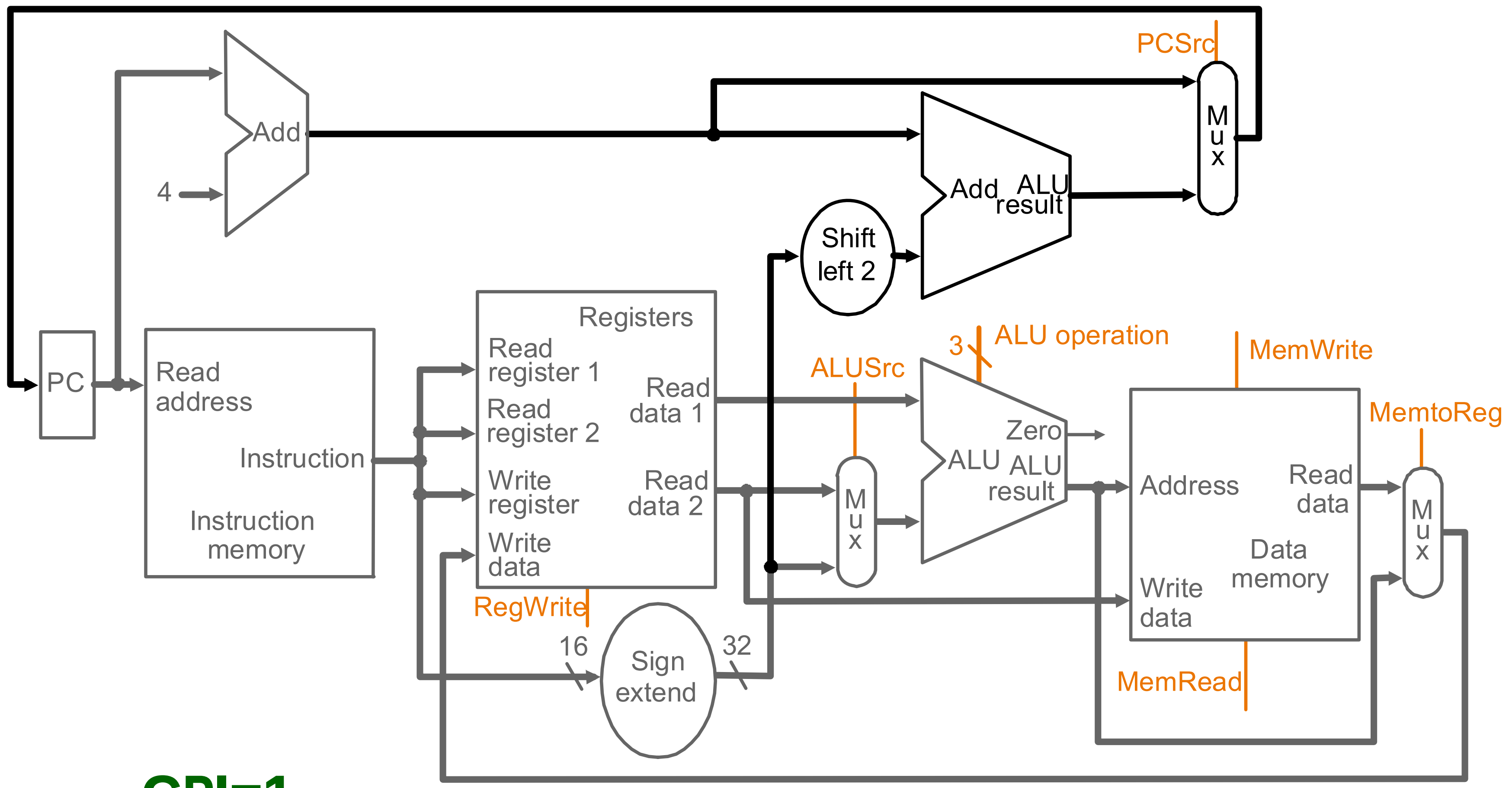
- One common case is reorder the instruction before the branch and the branch instruction

```
add   $4   $5  $6                 beq   $1,  $2,  40
beq   $1,  $2,  40      ⇒         add   $4   $5  $6
lw    $3,  300($0)                lw    $3,  300($0)
```

- Hidden from programmer
  – Assembler automatically arrange instructions.

§4.6 Pipelined Datapath and Control

Recap: A Single-Cycle Processor



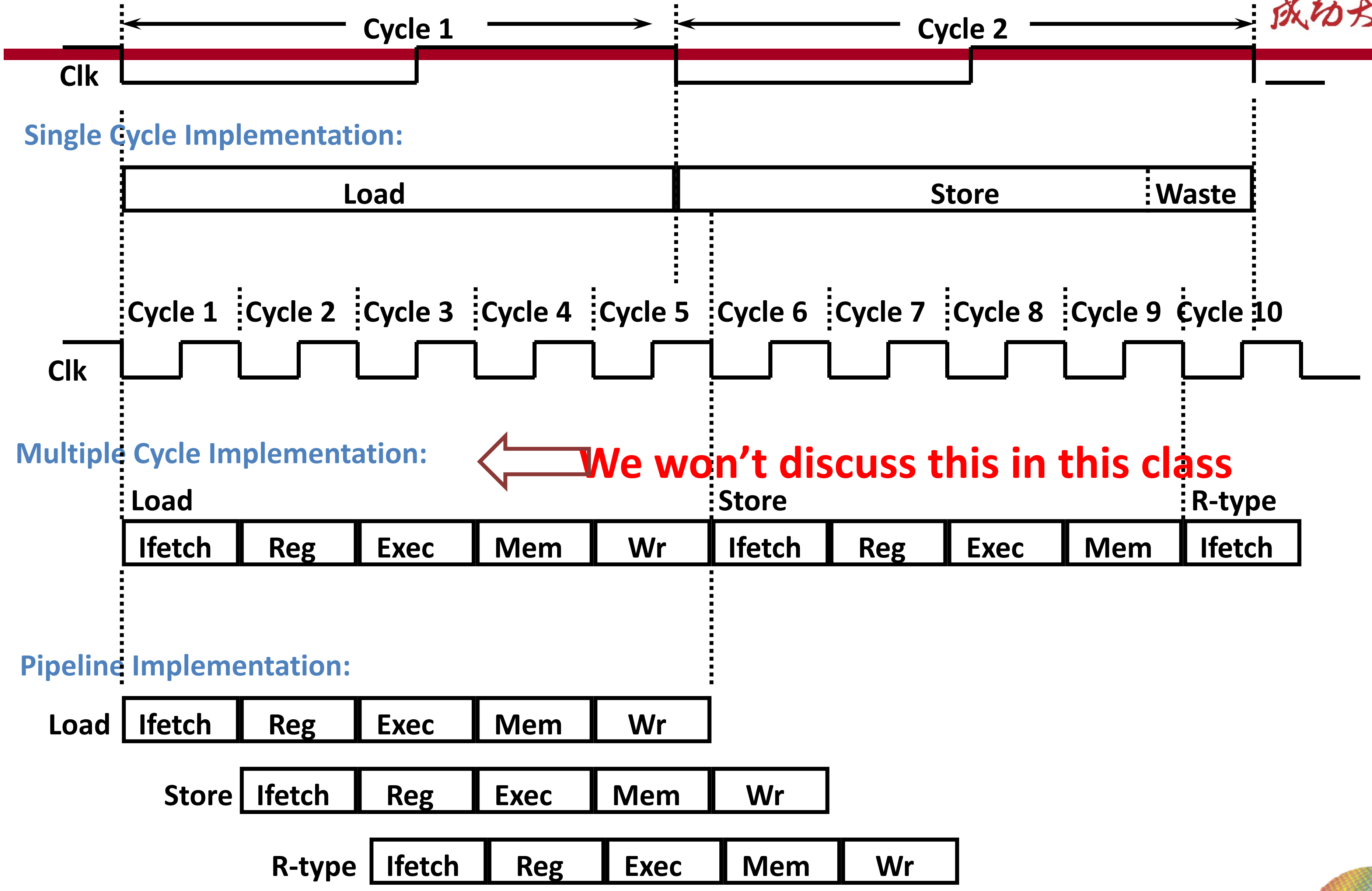
CPI=1

Single-Cycle Design Problems

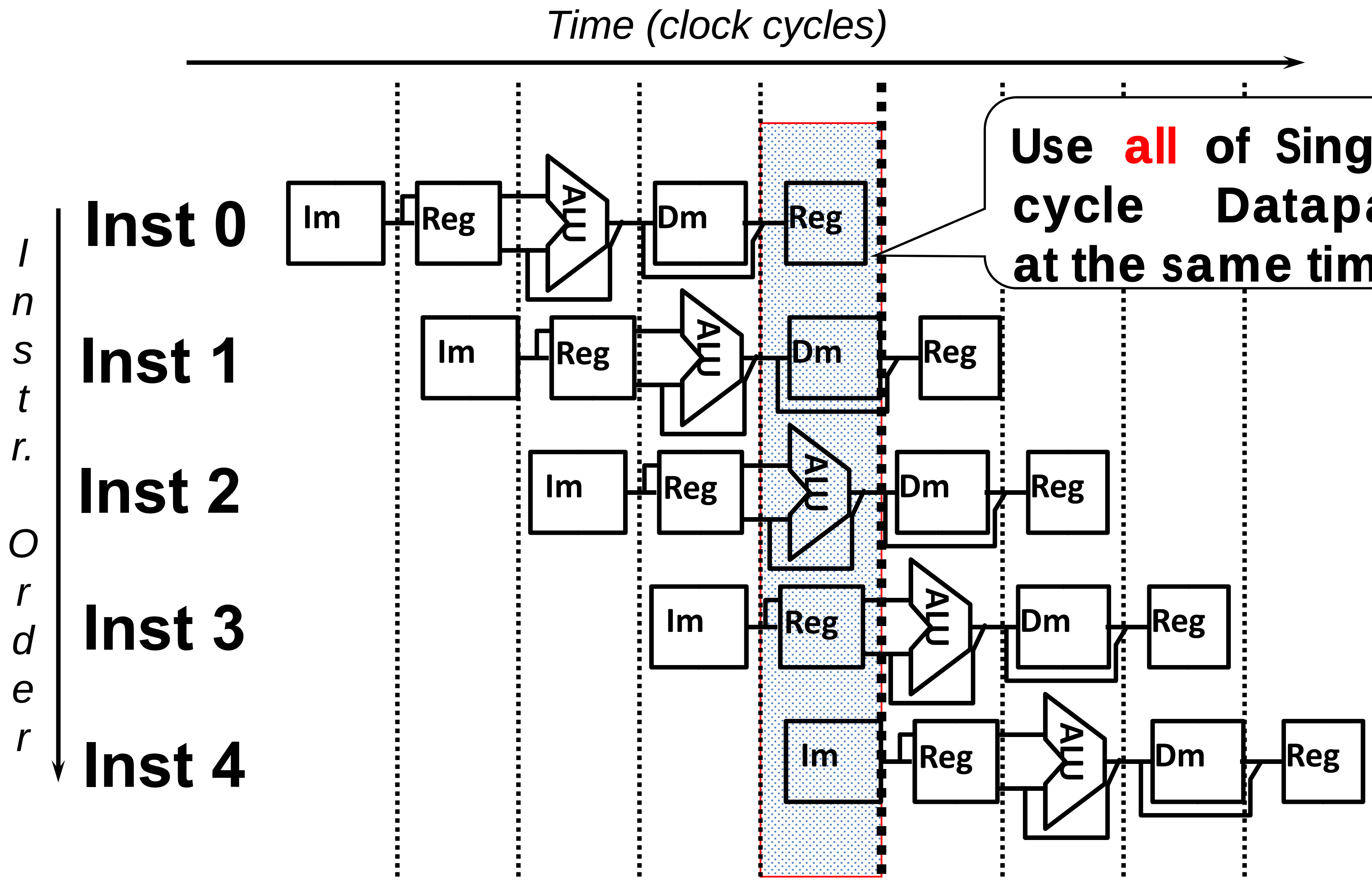
- Fixed-period clock
  - CPI = 1
  - cycle time determined by length of longest instruction path (lw)
    - but some instructions could run in a shorter clock cycle
      - *waste of time*
      - consider if we have more complicated instructions like floating point!
  - resources used more than once in the same cycle need to be duplicated
    - waste of hardware and chip area

Use pipelined design to fix the problem

Single-, Multi-Cycle, vs. Pipeline



Why Pipeline? Because the Resources are There!

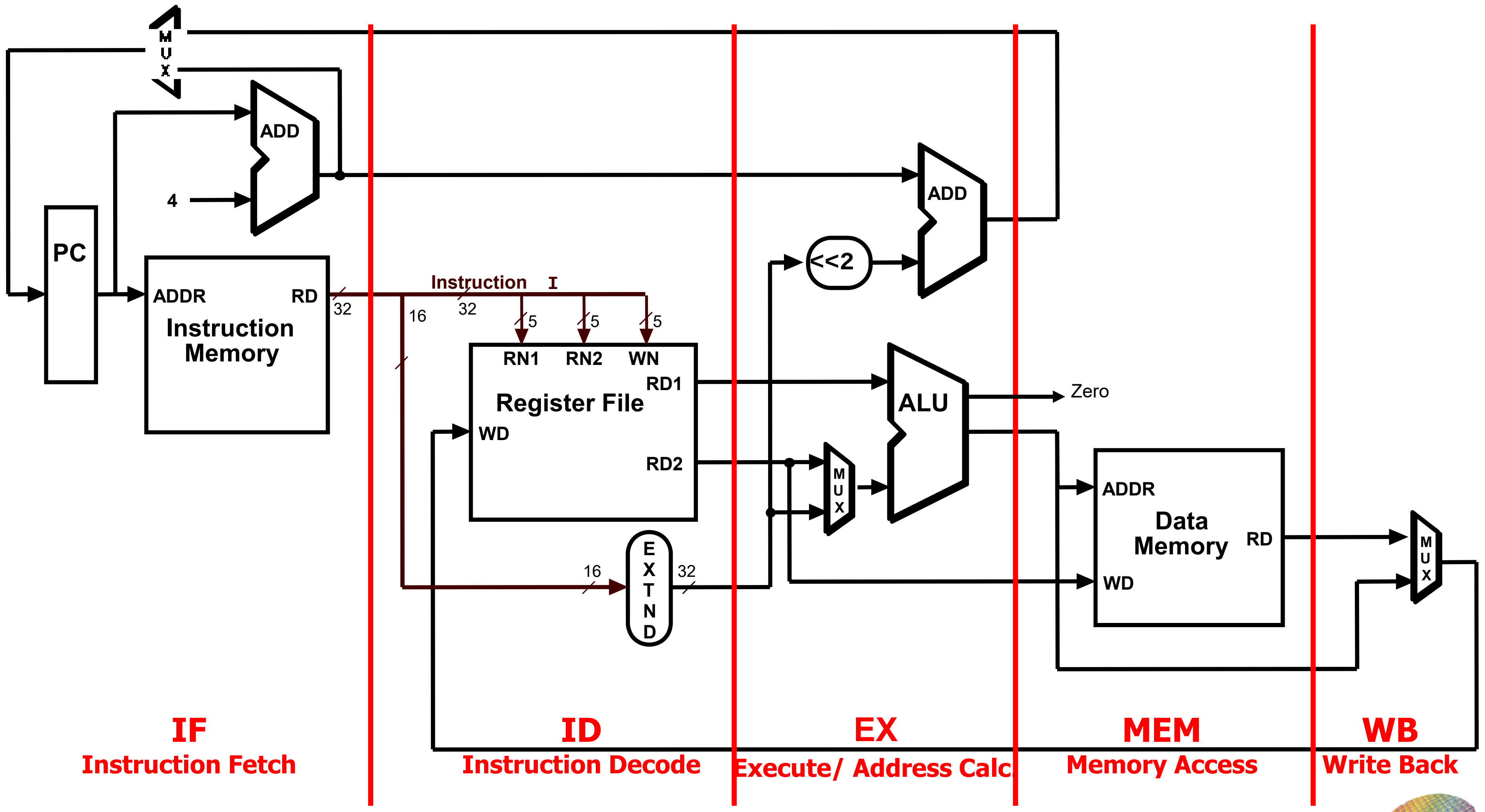


Use **all** of Single-cycle Datapath at the same time

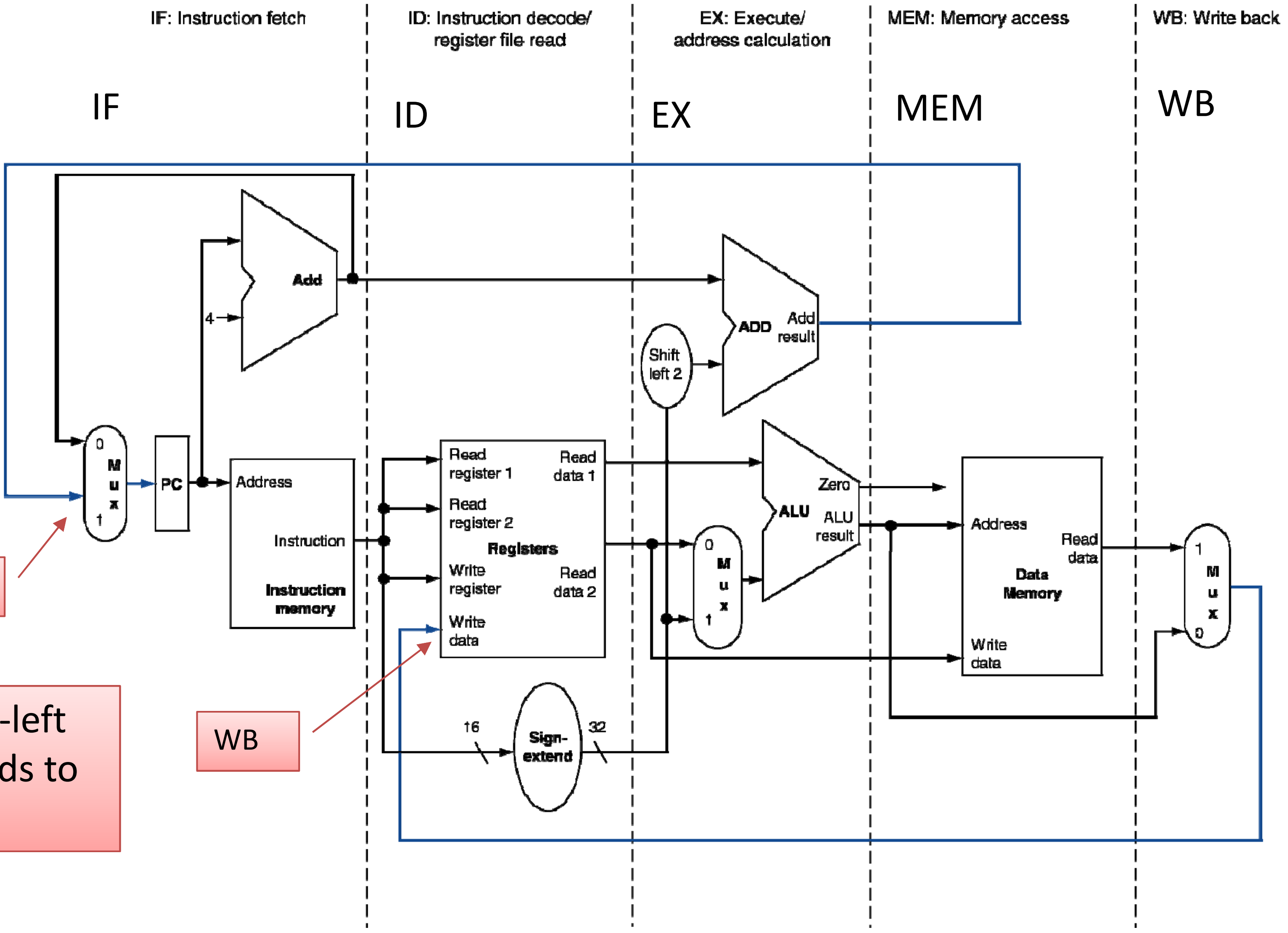
Recall: 5 steps in Instruction Execution

1. Instruction Fetch & PC Increment (IF)
2. Instruction Decode and Register Read (ID)
3. Execution or calculate address (EX)
4. Memory access (MEM)
5. Write result into register (WB)

Recall “Single-Cycle” Datapath



MIPS Pipelined Datapath – 5 stages



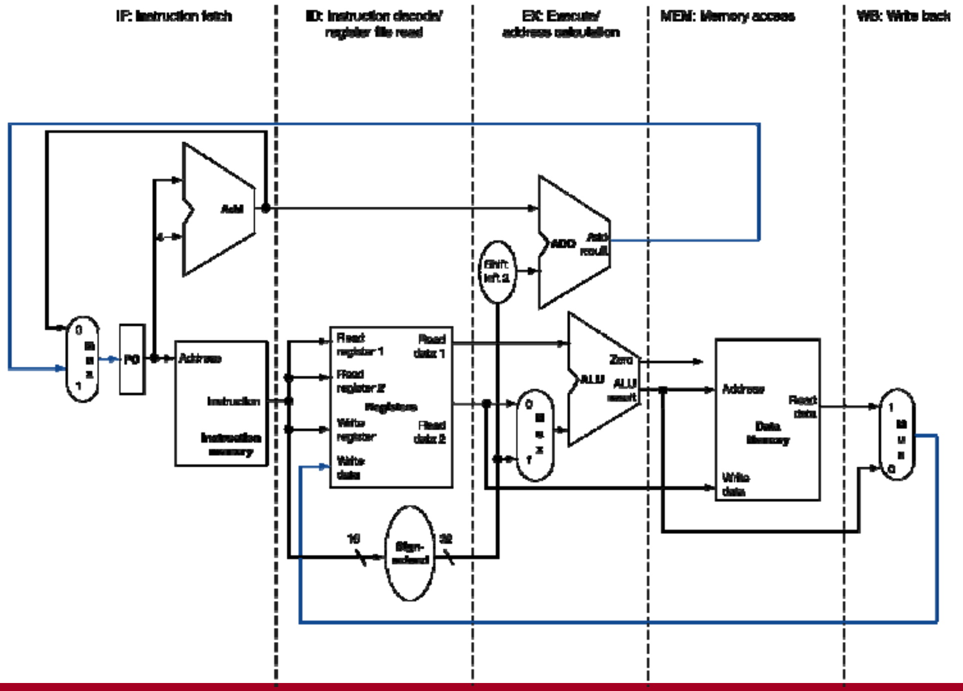
MEM

WB

Right-to-left flow leads to hazards

Pipelined Datapath

- Break instructions into the following *potential* execution steps
  1. Instruction fetch and PC increment (IF)
  2. Instruction decode and register fetch (ID)
  3. Execution, memory address computation, or branch completion (EX)
  4. Memory access or R-type instruction completion (MEM)
  5. Memory read completion (WB)



Not every instruction need 5 steps!

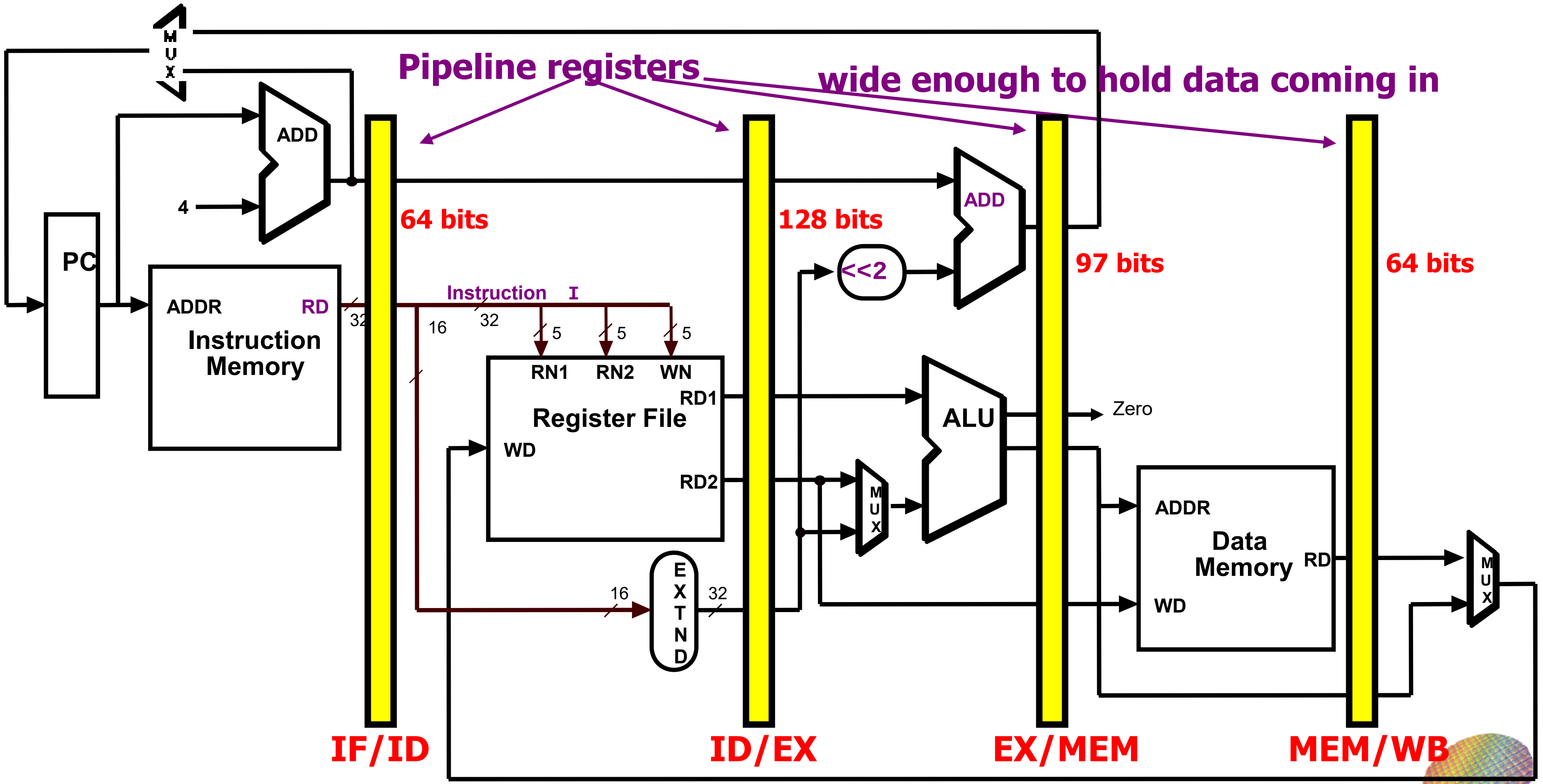
R-type need 4 steps,

lw needs 5 steps,

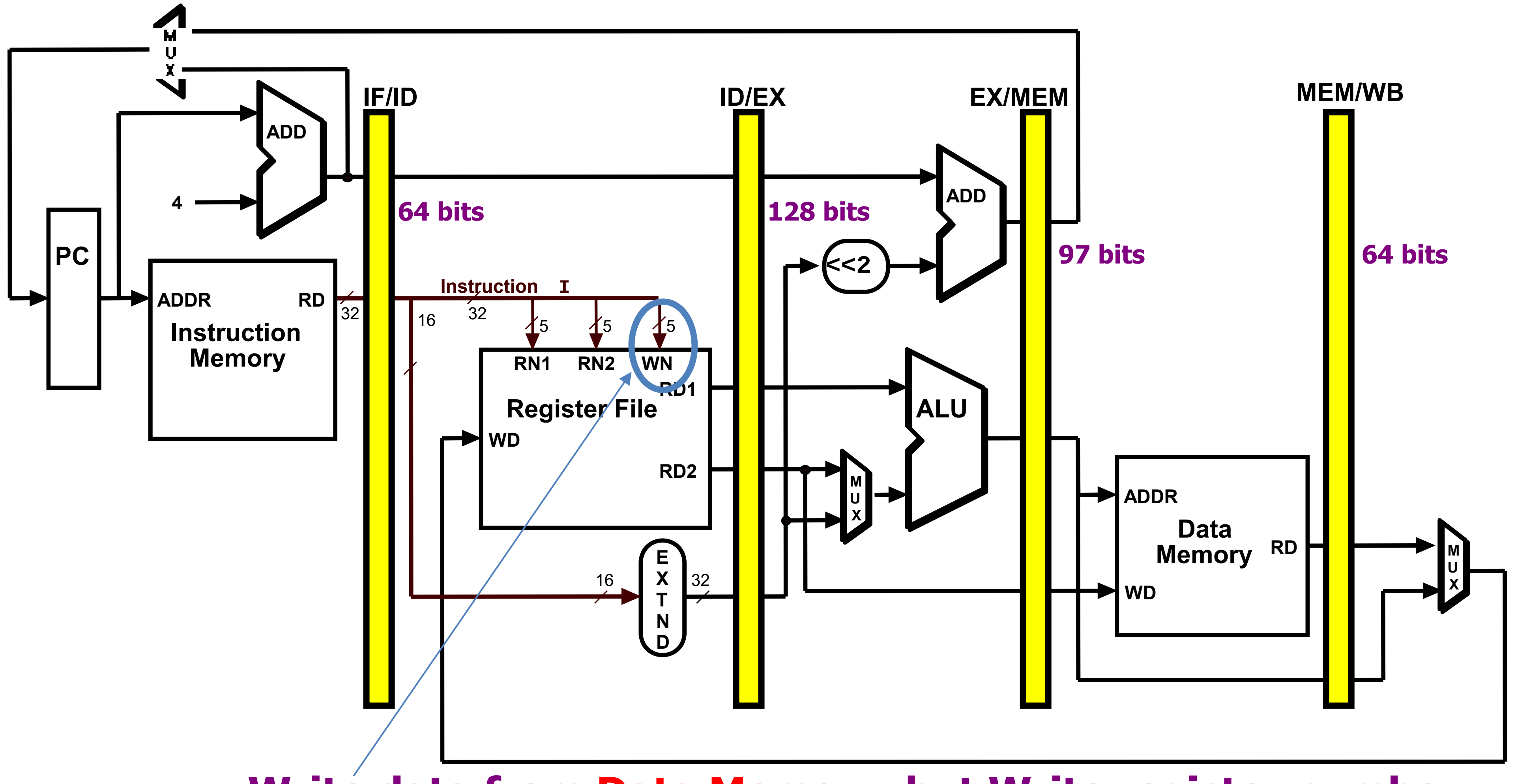
branch needs 3 steps

Pipelined Datapath

- Need registers between stages
  - To hold information produced in previous cycle
  - Data pass to next stage only during clock transition

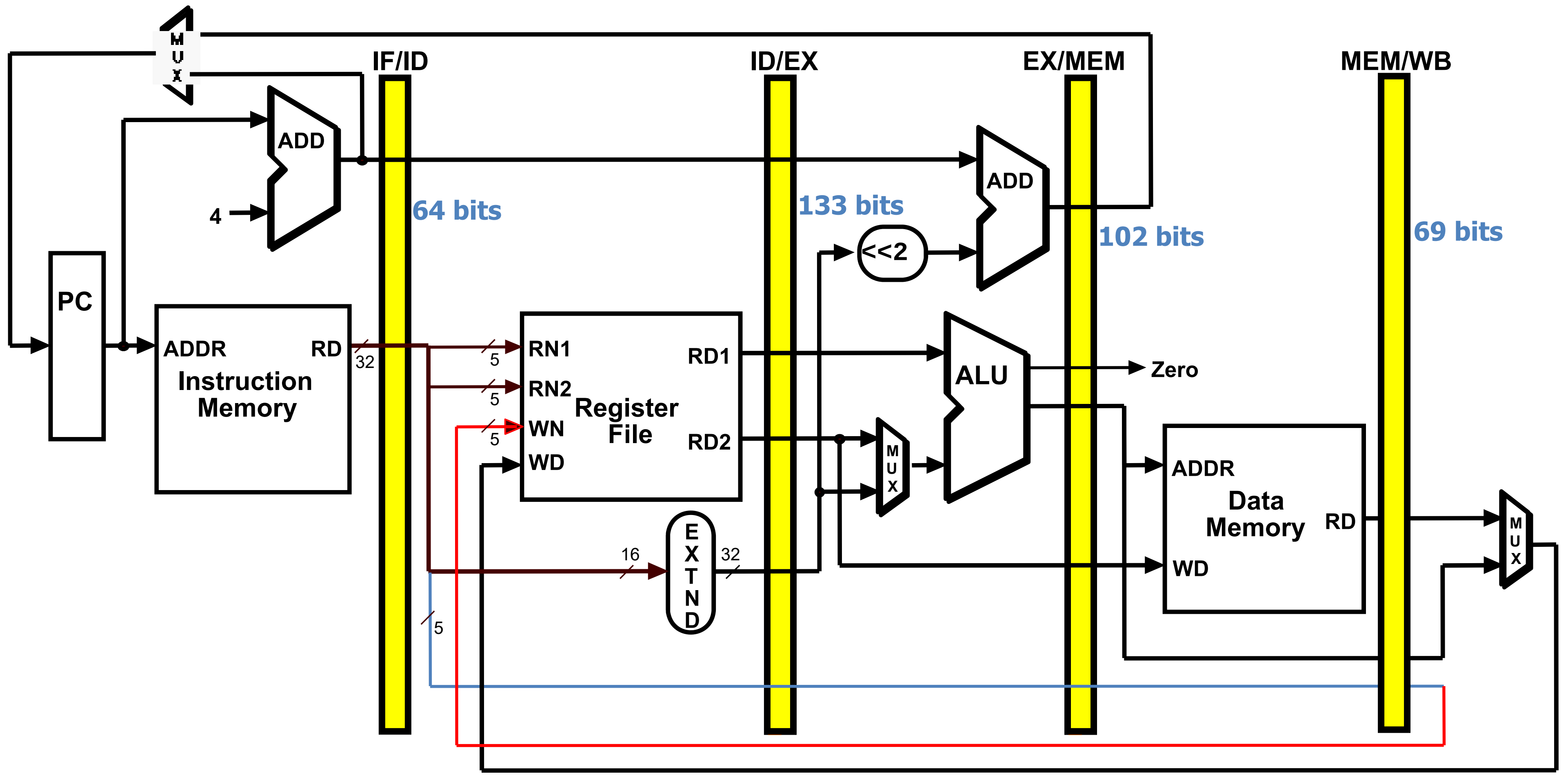


Bug in the Datapath



Write data from Data Memory, but Write register number comes from another instruction in the latter stage!

Corrected Datapath



Destination register number is also passed through ID/EX, EX/MEM and MEM/WB registers, which are now wider by 5 bits

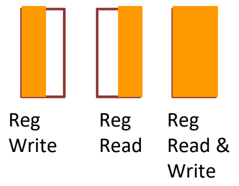
Pipeline Operation

- Cycle-by-cycle flow of instructions through the pipelined datapath
  - "Single-clock-cycle" pipeline diagram
    - Shows pipeline usage in a single cycle
    - Highlight resources used
  - c.f. "multi-clock-cycle" diagram
    - Graph of operation over time
- We'll look at "single-clock-cycle" diagrams for load & store

Pipelined Example

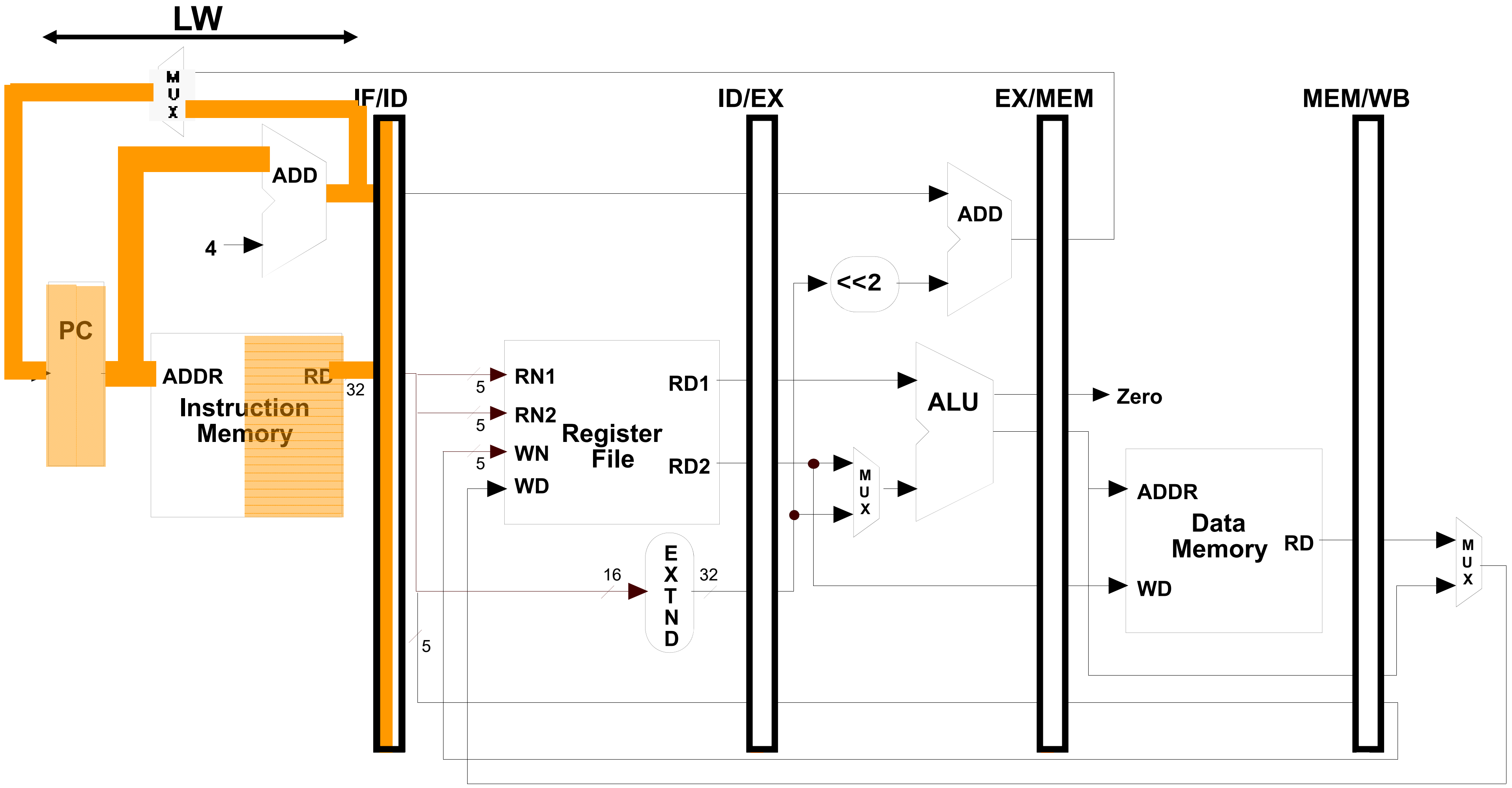
- Consider the following instruction sequence:

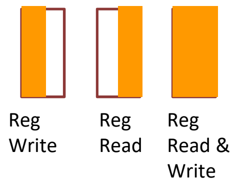
```
lw  $t0,  10($t1)
sw  $t3, 20($t4)
add $t5,  $t6,  $t7
sub $t8,  $t9,  $t10
```



Single-Clock-Cycle Diagram: Clock Cycle 1

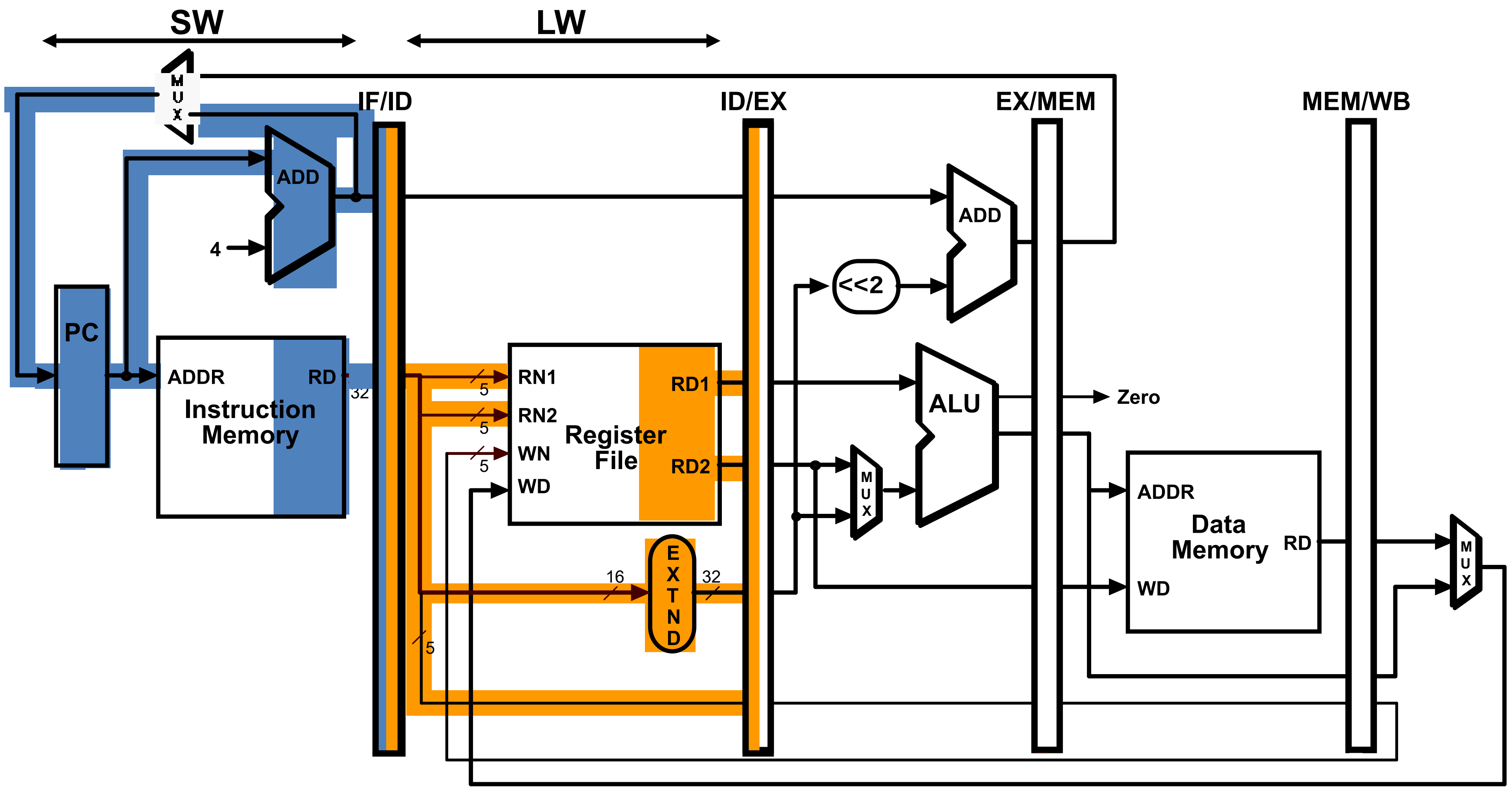
lw $t0, 10($t1)
sw $t3, 20($t4)
add $t5, $t6, $t7
sub $t8, $t9, $t10

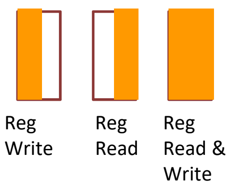




Single-Clock-Cycle Diagram: Clock Cycle 2

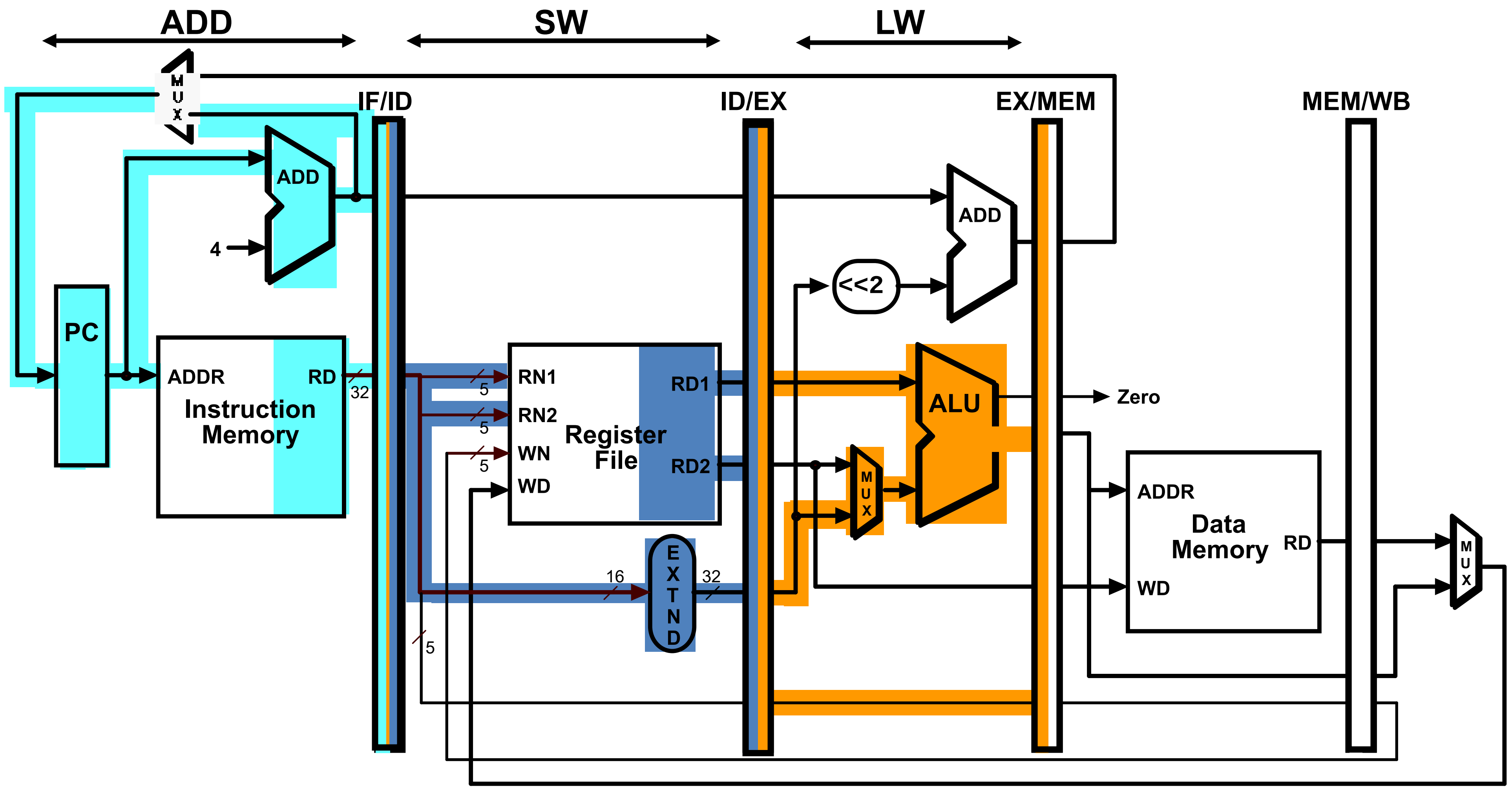
lw $t0, 10($t1)
sw $t3, 20($t4)
add $t5, $t6, $t7
sub $t8, $t9, $t10

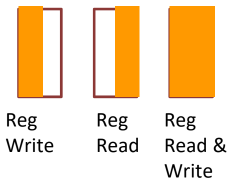




Single-Clock-Cycle Diagram: Clock Cycle 3

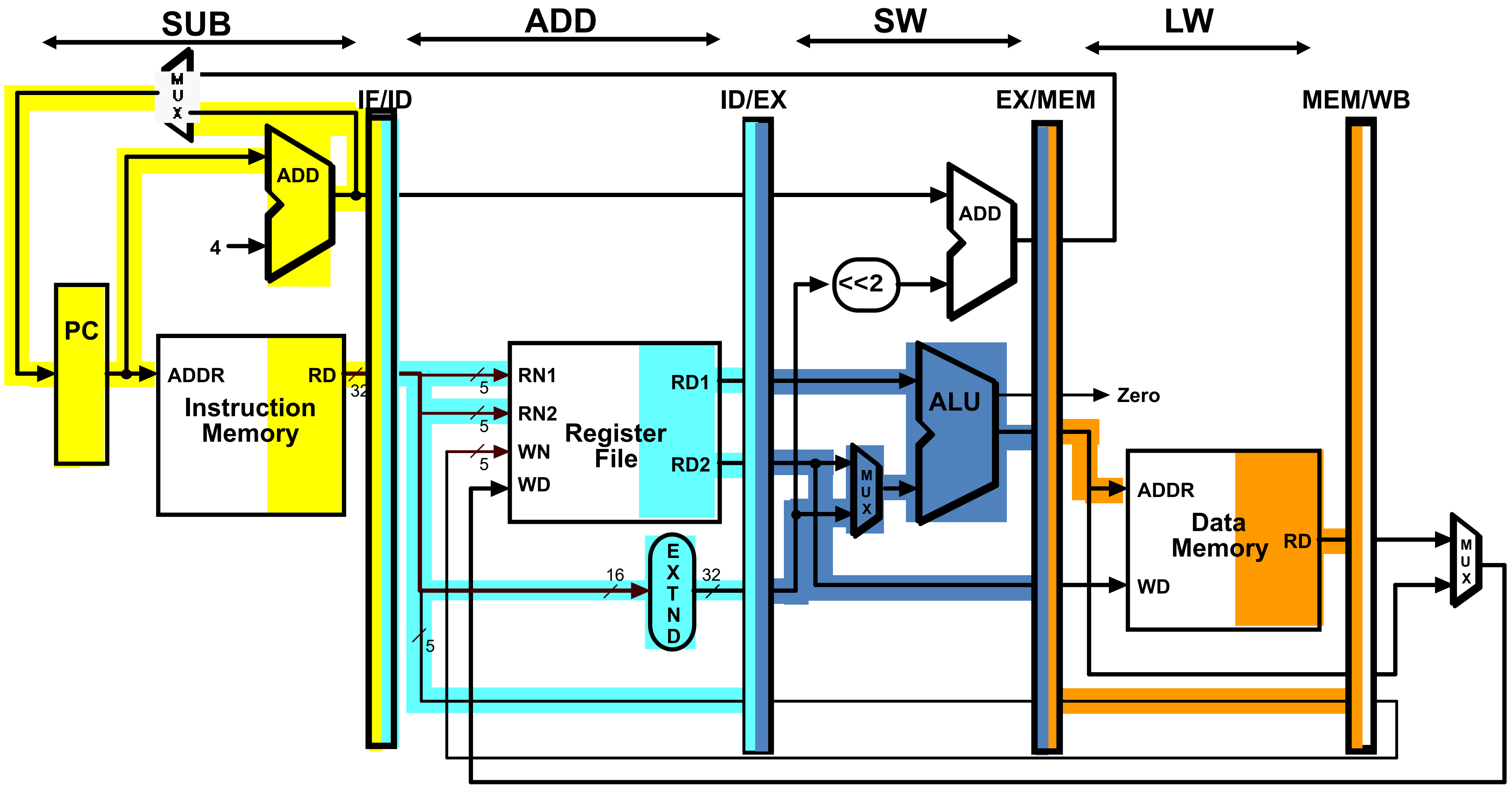
lw $t0, 10($t1)
sw $t3, 20($t4)
add $t5, $t6, $t7
sub $t8, $t9, $t10

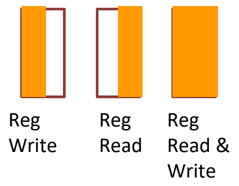




Single-Clock-Cycle Diagram: Clock Cycle 4

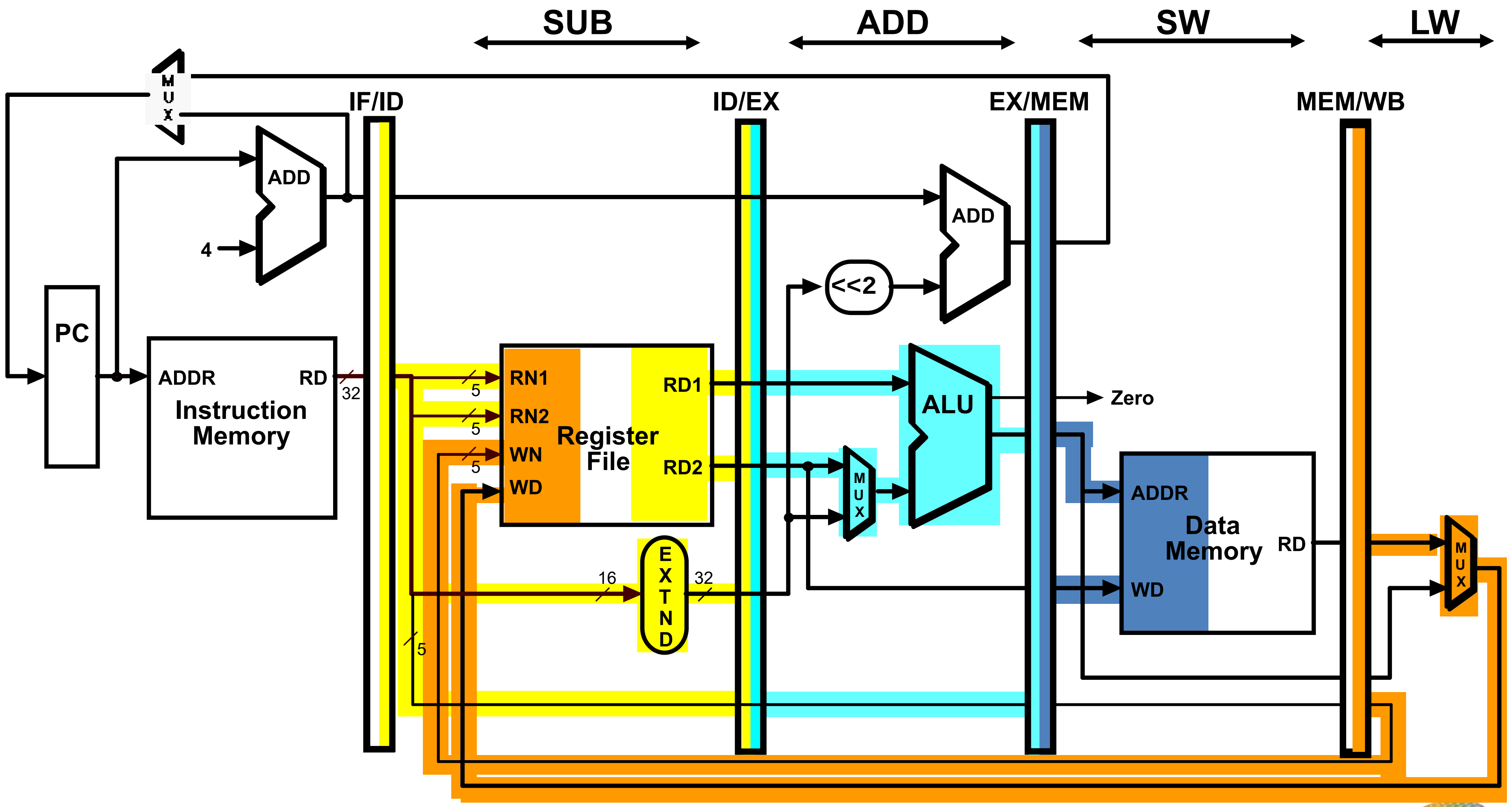
lw $t0, 10($t1)
sw $t3, 20($t4)
add $t5, $t6, $t7
sub $t8, $t9, $t10

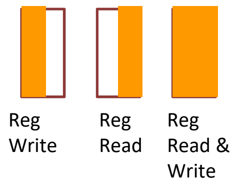




Single-Clock-Cycle Diagram: Clock Cycle 5

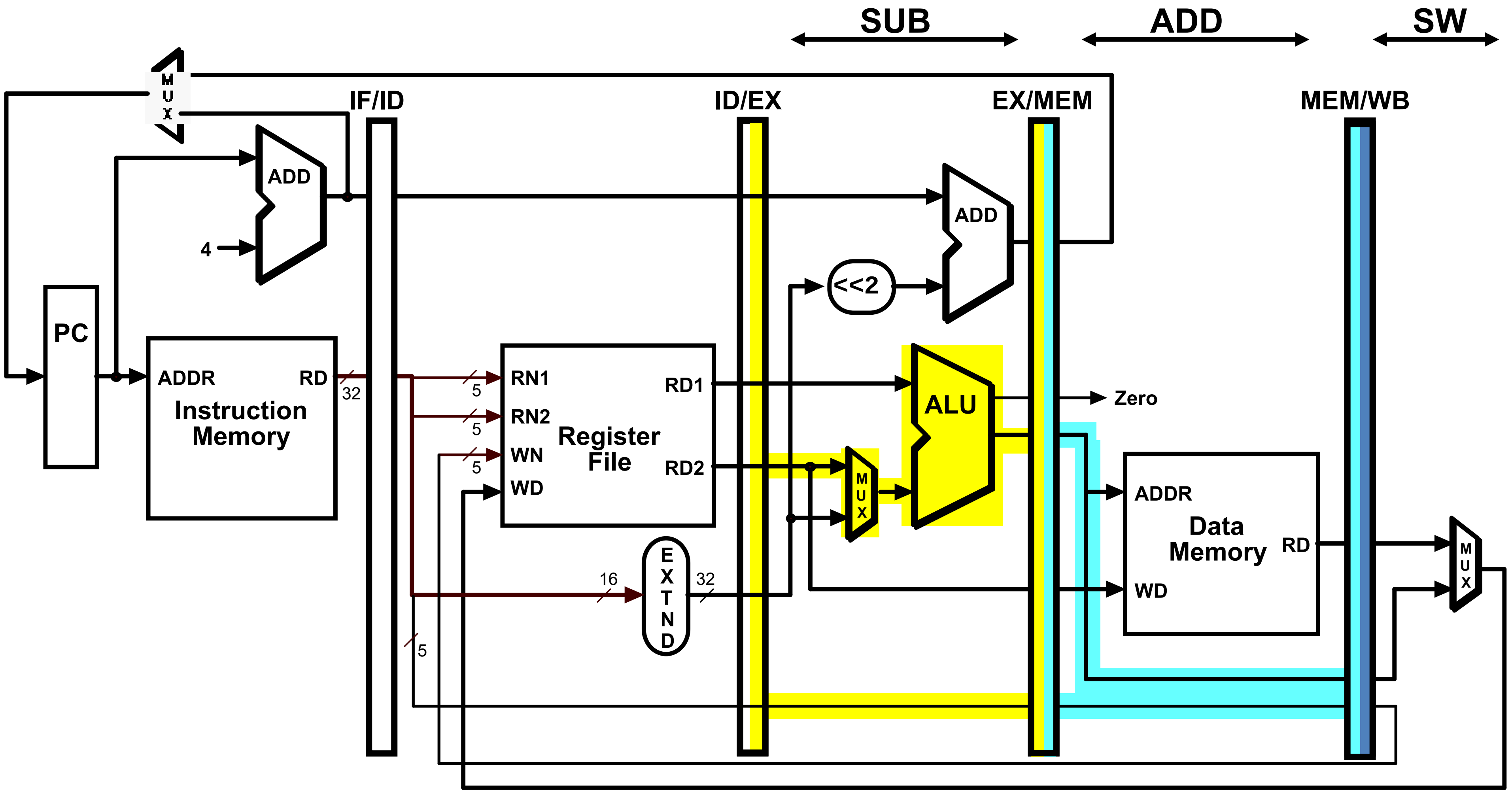
lw $t0, 10($t1)
sw $t3, 20($t4)
add $t5, $t6, $t7
sub $t8, $t9, $t10

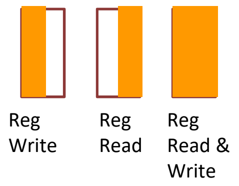




Single-Clock-Cycle Diagram: Clock Cycle 6

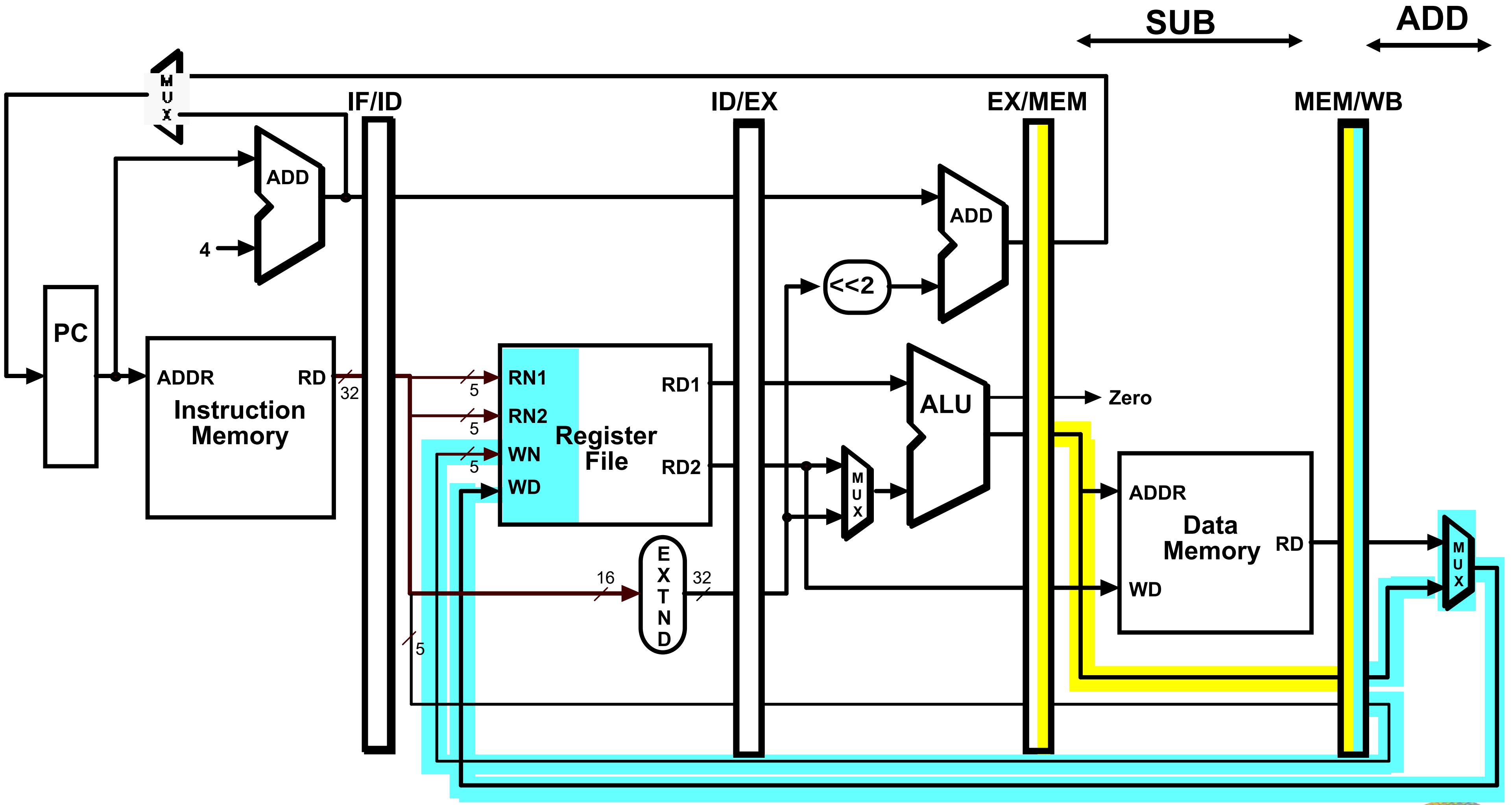
lw $t0, 10($t1)
sw $t3, 20($t4)
add $t5, $t6, $t7
sub $t8, $t9, $t10





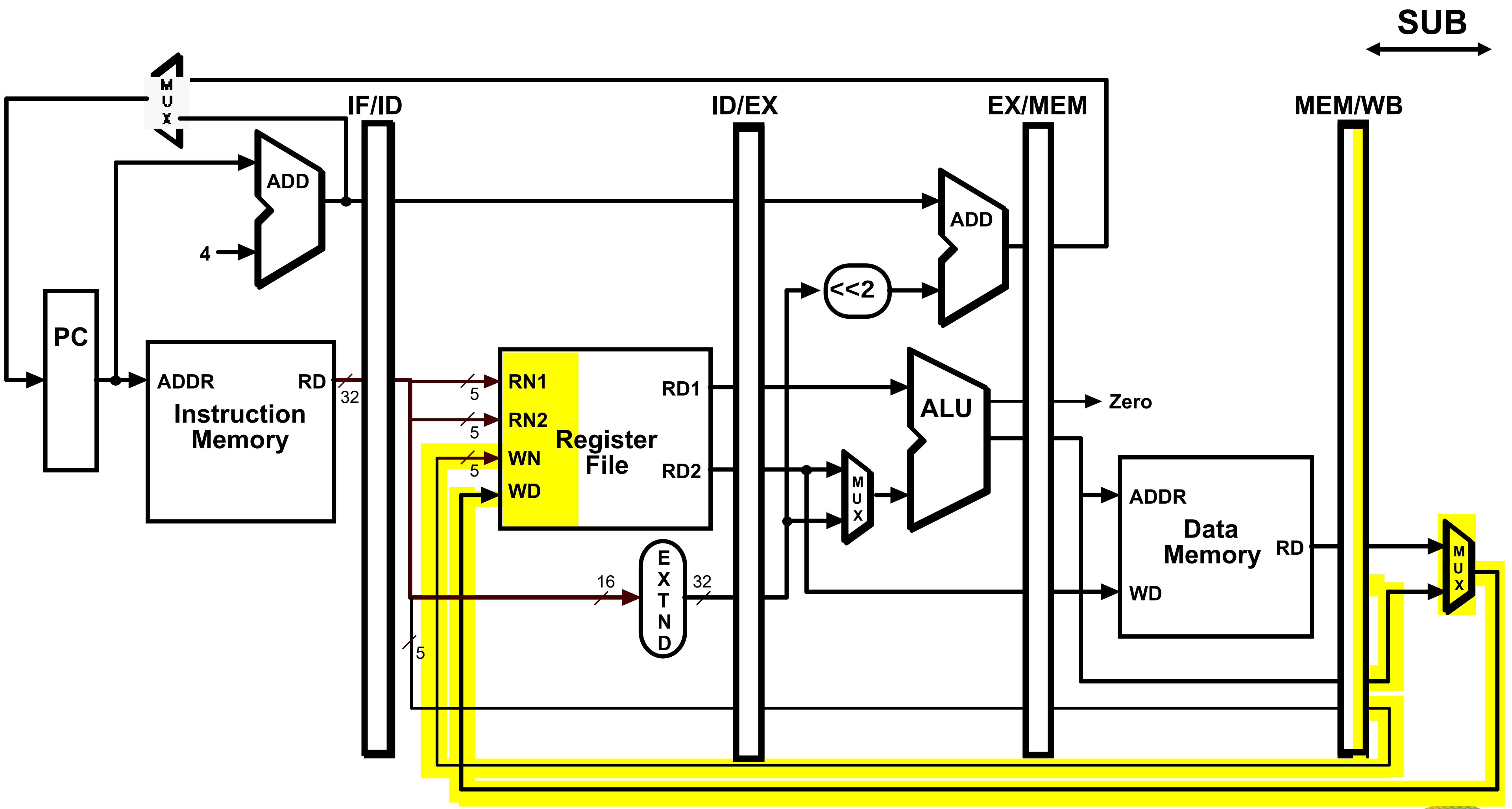
Single-Clock-Cycle Diagram: Clock Cycle 7

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



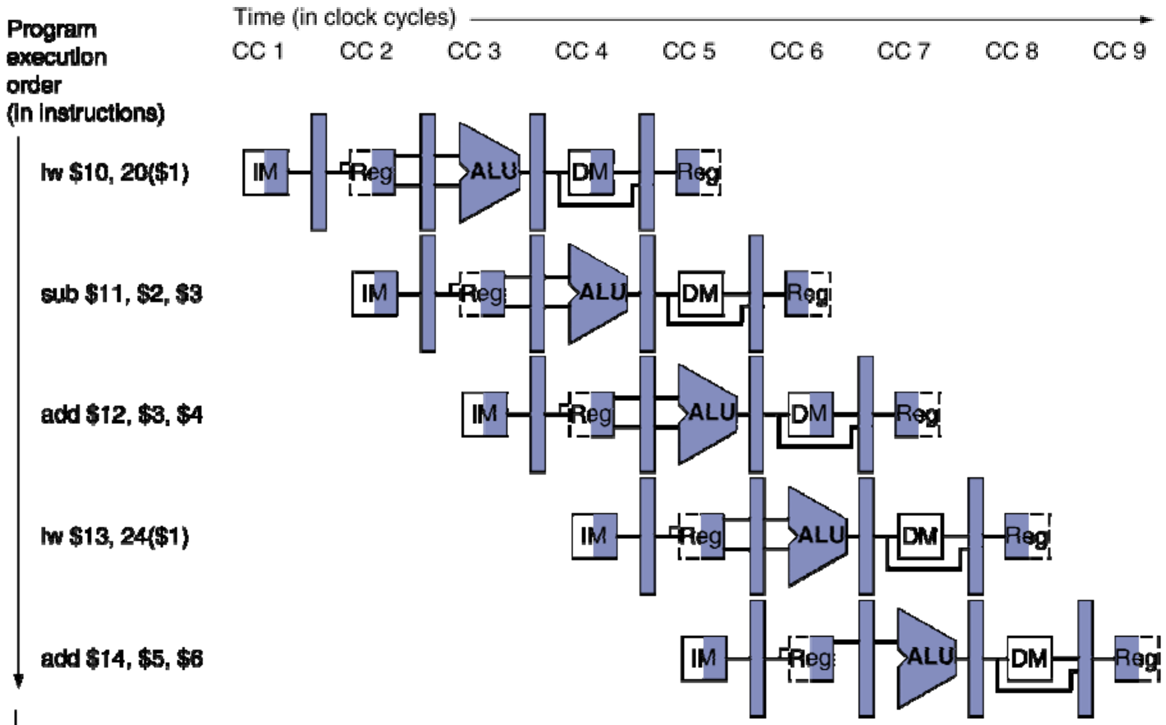
Single-Clock-Cycle Diagram: Clock Cycle 8

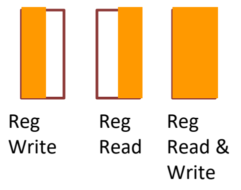
lw $t0, 10($t1)
sw $t3, 20($t4)
add $t5, $t6, $t7
sub $t8, $t9, $t10



Multi-Cycle Pipeline Diagram

- Two ways to represent pipeline





Resource usage is shown

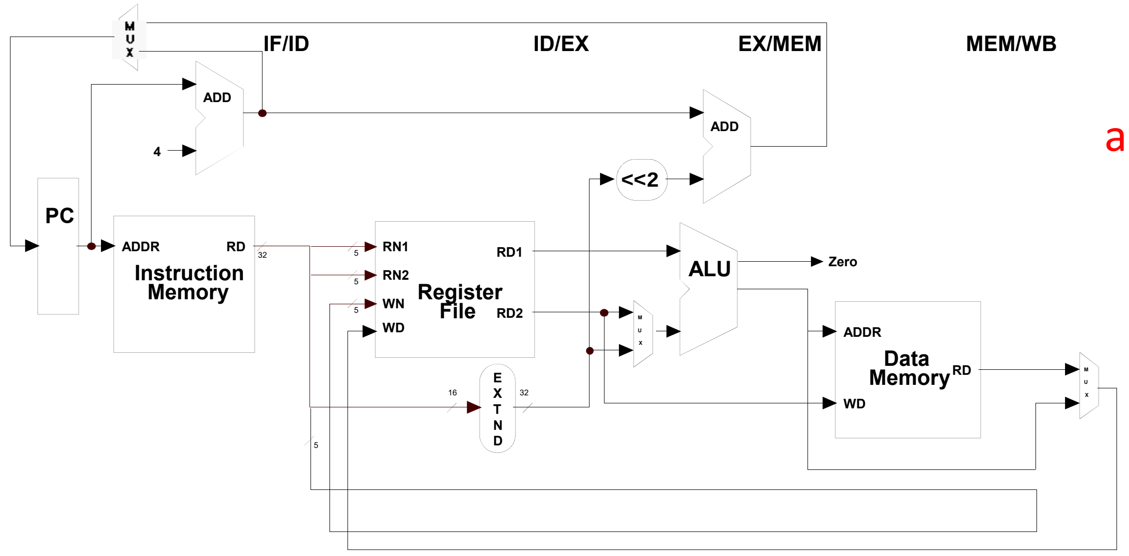
lw $10, 20($1)	Instruction fetch	Instruction decode	Execution	Data access	Write back				
sub $11, $2, $3		Instruction fetch	Instruction decode	Execution	Data access	Write back			
add $12, $3, $4			Instruction fetch	Instruction decode	Execution	Data access	Write back		
lw $13, 24($1)				Instruction fetch	Instruction decode	Execution	Data access	Write back	
add $14, $5, $6					Instruction fetch	Instruction decode	Execution	Data access	Write back

Traditional

Why the execution of an R-type instruction uses 5 stages?

Observation

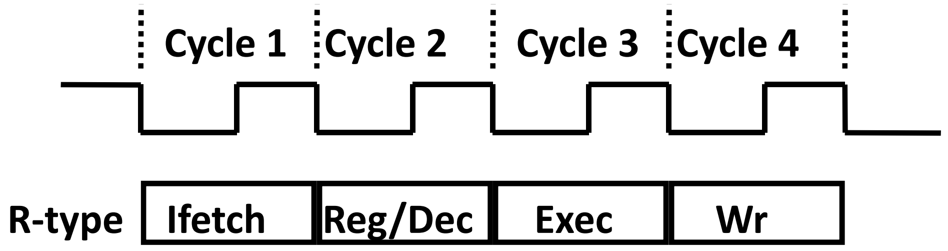
- R-type instruction, do not need to write to data memory
  - Only IF, ID, EXE, WB stages are used
- why the execution of an R-type instruction uses 5 stages?



add $t5, $t6, $t7

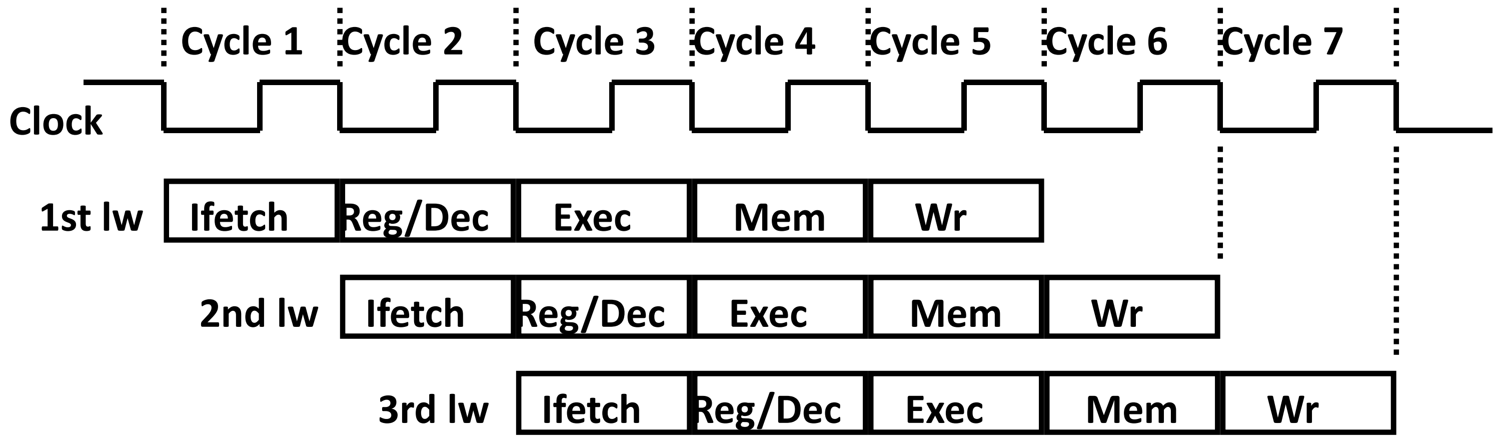
do not need to write to data memory

Supposed R-type Instructions only use 4 stages in pipeline



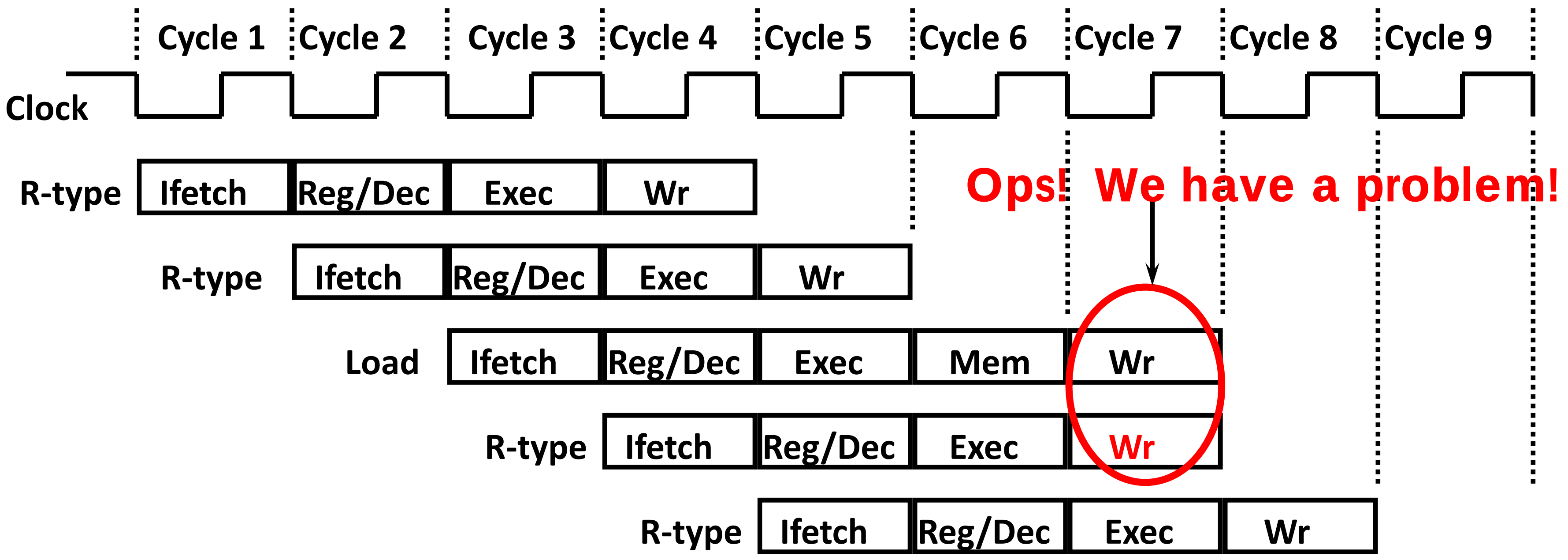
- IF: fetch the instruction from the Instruction Memory
- ID: registers fetch and instruction decode
- EX: ALU operates on the two register operands
- WB: write ALU output back to the register file

Now Load has 5 stages



- 5 functional units in the pipeline datapath are:
  - Instruction Memory for the Fetch stage
  - Register File's Read ports (busA and busB) for the Reg/Dec stage
  - ALU for the Exec stage
  - Data Memory for the MEM stage
  - Register File's Write port (busW) for the WB stage

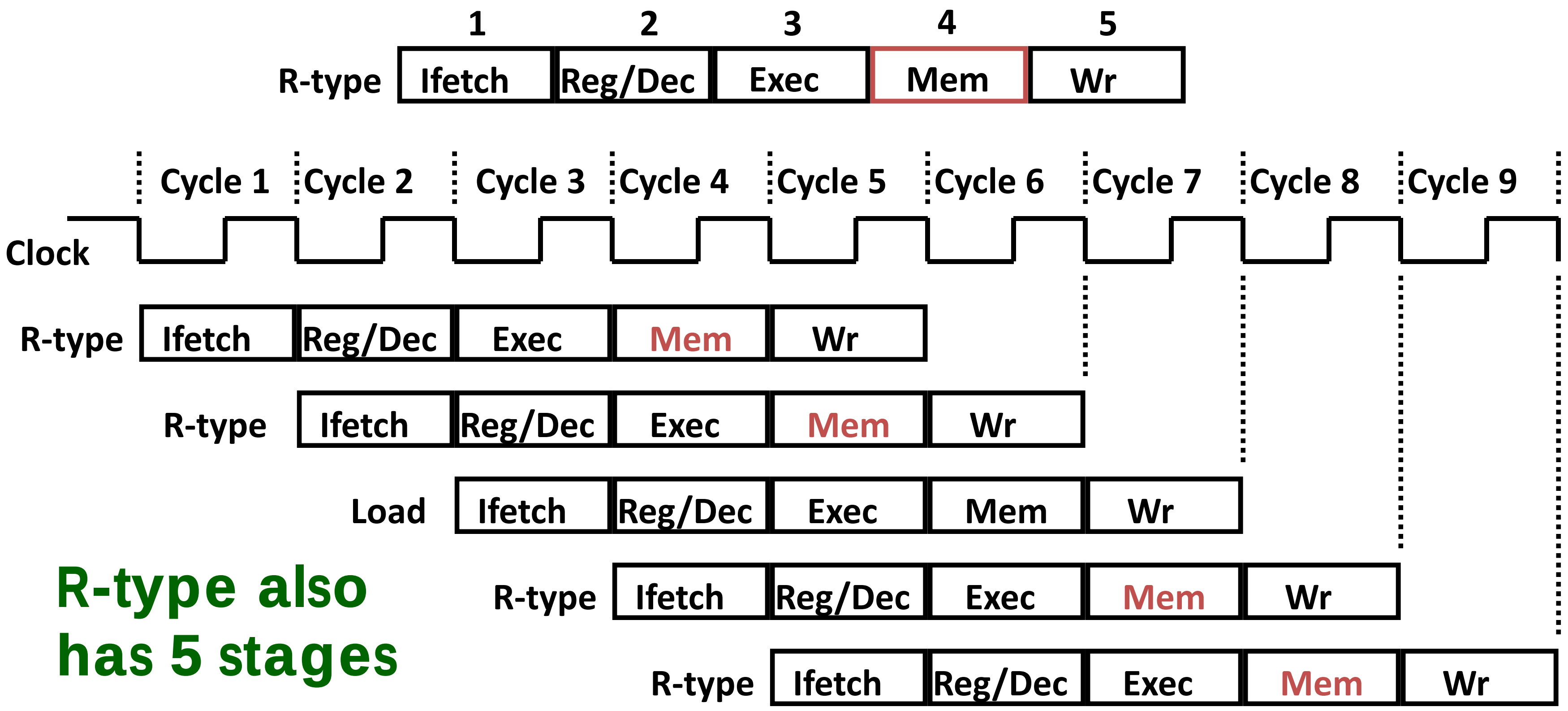
Pipelining with R-type and `load`



- We have a *structural hazard*:
  - Two instructions try to write to the register file at the same time!
  - Only one write port

Solution: Delay R-type's Write

- Delay R-type's register write by one cycle:
  - R-type also use Reg File's write port at Stage 5
  - MEM is a NOP stage: nothing is being done.



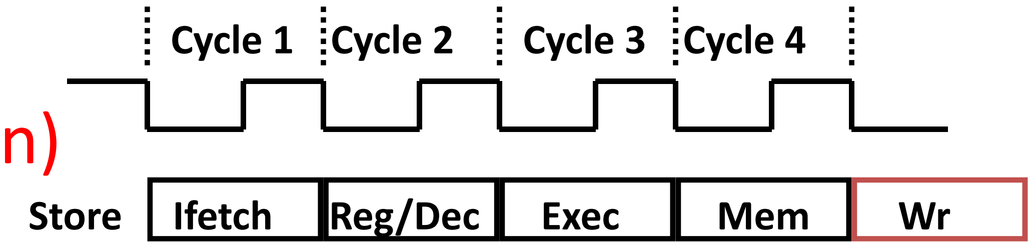
R-type also has 5 stages

SW

- Similar situation applies to SW, originally only 4 stages are needed
  - IF: fetch the instruction from the Instruction Memory
  - ID: registers fetch and instruction decode
  - EX: calculate the memory address
  - MEM: write the data into the Data Memory

Add an extra stage:

- WB: NOP (No operation)



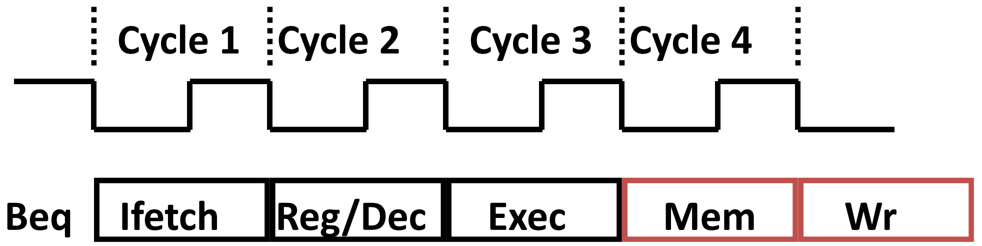
BEQ

Similar situation applies to BEQ instruction, originally only 3 stages are needed

- IF: fetch the instruction from the Instruction Memory
- ID: registers fetch and instruction decode
- EX: compare the two register operand, select correct branch target address, and latch into PC

Add two extra stages:

- MEM: NOP
- WB: NOP



Backup slides

- Assume 5 stages: IF, ID, EX, MEM, WB , list the active stages in the following instructions

lw $t0, 10($t1)	IF, ID, EX, MEM, WB
sw $t3, 20($t4)	IF, ID, EX, MEM
add $t5, $t6, $t7	IF, ID, EX,, WB
sub $t8, $t9, $t10	IF, ID, EX,, WB
Beq $t0, $t1, EXIT	IF, ID, EX,