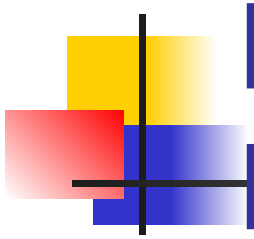
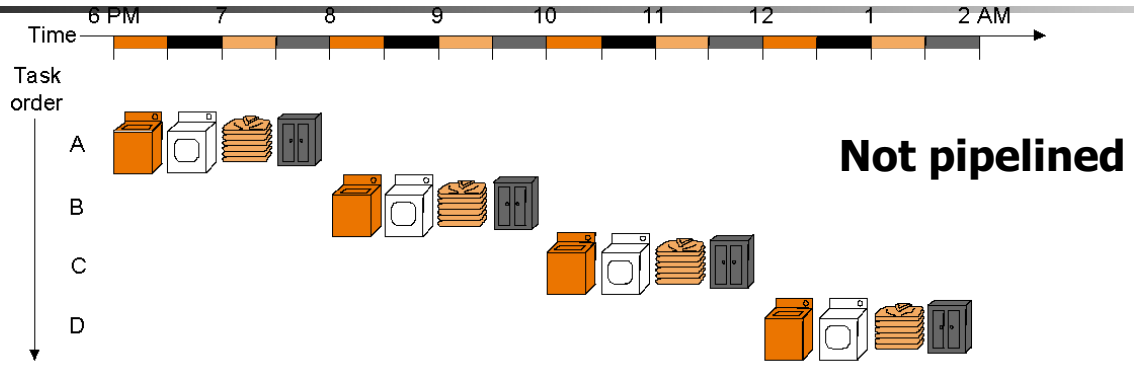


COD Ch. 6
Enhancing Performance with Pipelining

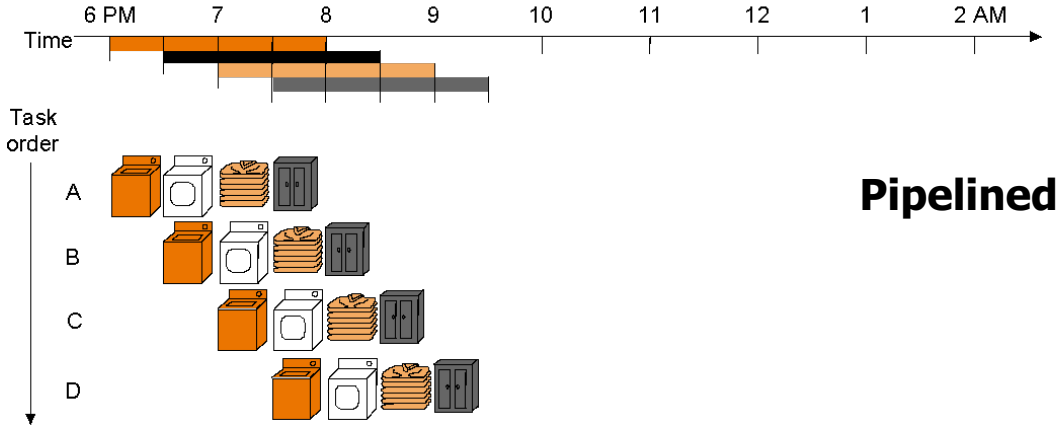
Pipelining

- Start work ASAP!! Do not waste time!



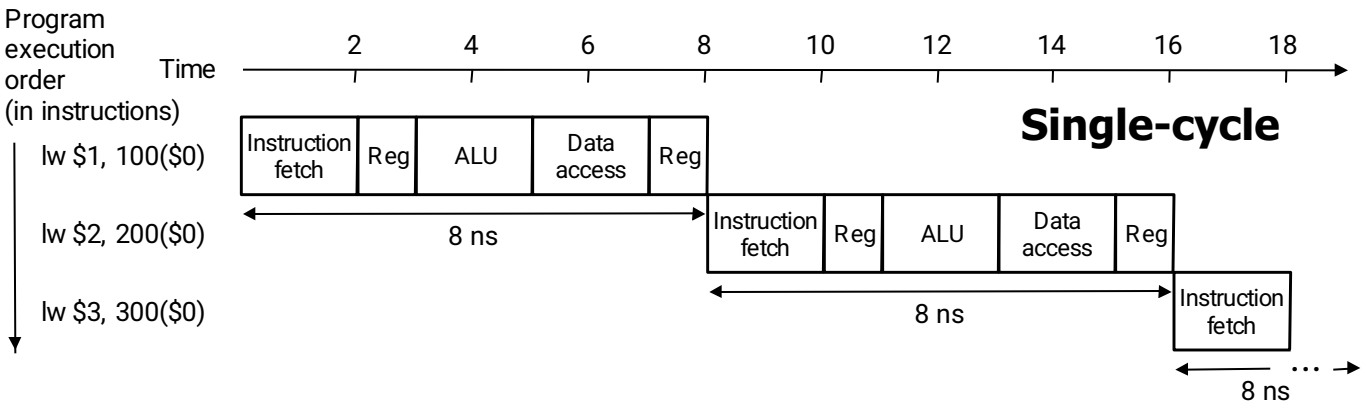
Not pipelined

Assume 30 min. each task – wash, dry, fold, store – and that separate tasks use separate hardware and so can be overlapped

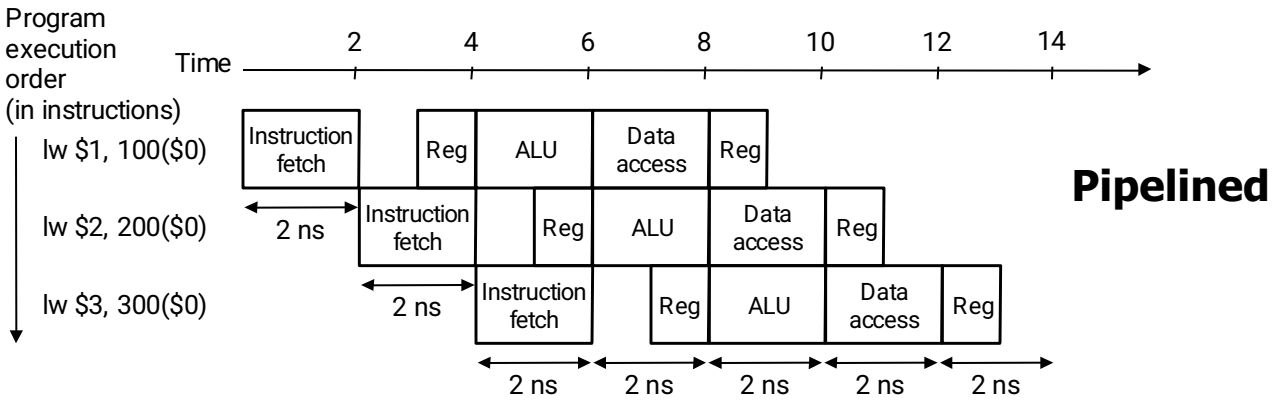


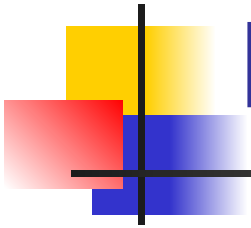
Pipelined

Pipelined vs. Single-Cycle Instruction Execution: the Plan



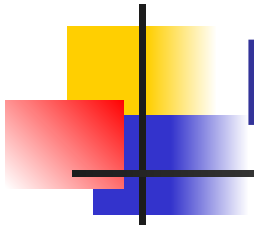
Assume 2 ns for memory access, ALU operation; 1 ns for register access: therefore, single cycle clock 8 ns; pipelined clock cycle 2 ns.





Pipelining: Keep in Mind

- Pipelining does not reduce latency of a single task, it increases throughput of entire workload
- Pipeline rate limited by longest stage
  - potential speedup = number pipe stages
  - unbalanced lengths of pipe stages reduces speedup
- Time to fill pipeline and time to drain it – when there is slack in the pipeline – reduces speedup

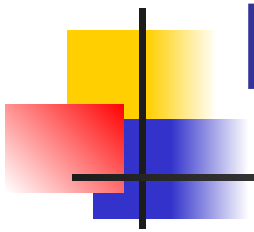


Example Problem

- *Problem: for the laundry fill in the following table when*
  1. *the stage lengths are 30, 30, 30 30 min., resp.*
  2. *the stage lengths are 20, 20, 60, 20 min., resp.*

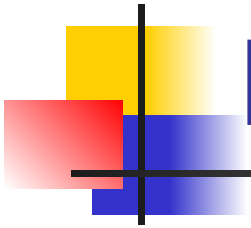
Person	Unpipelined finish time	Pipeline 1 finish time	Ratio unpipelined to pipeline 1	Pipeline 2 finish time	Ratio unpiplelined to pipeline 2
1					
2					
3					
4					
n					

- *Come up with a formula for pipeline speed-up!*



Pipelining MIPS

- What makes it easy with MIPS?
  - all instructions are same length
    - so fetch and decode stages are similar for all instructions
  - just a few instruction formats
    - simplifies instruction decode and makes it possible in one stage
  - memory operands appear only in load/stores
    - so memory access can be deferred to exactly one later stage
  - operands are aligned in memory
    - one data transfer instruction requires one memory access stage



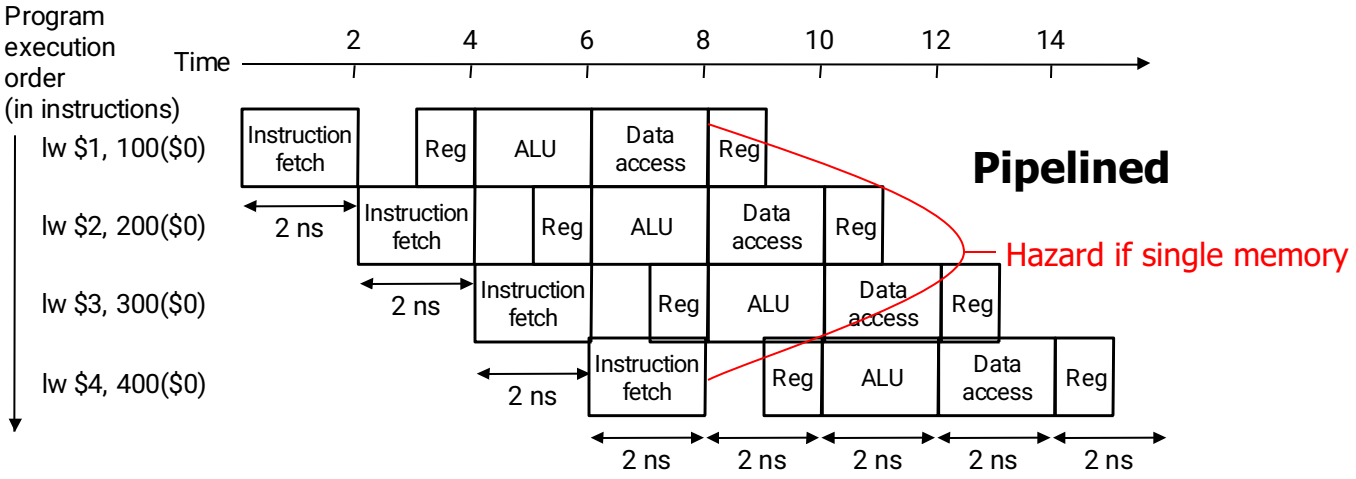
Pipelining MIPS

- What makes it hard?
  - structural hazards: different instructions, at different stages, in the pipeline want to use the same hardware resource
  - control hazards: succeeding instruction, to put into pipeline, depends on the outcome of a previous branch instruction, already in pipeline
  - data hazards: an instruction in the pipeline requires data to be computed by a previous instruction still in the pipeline

- Before actually building the pipelined datapath and control we first briefly examine these potential hazards individually…

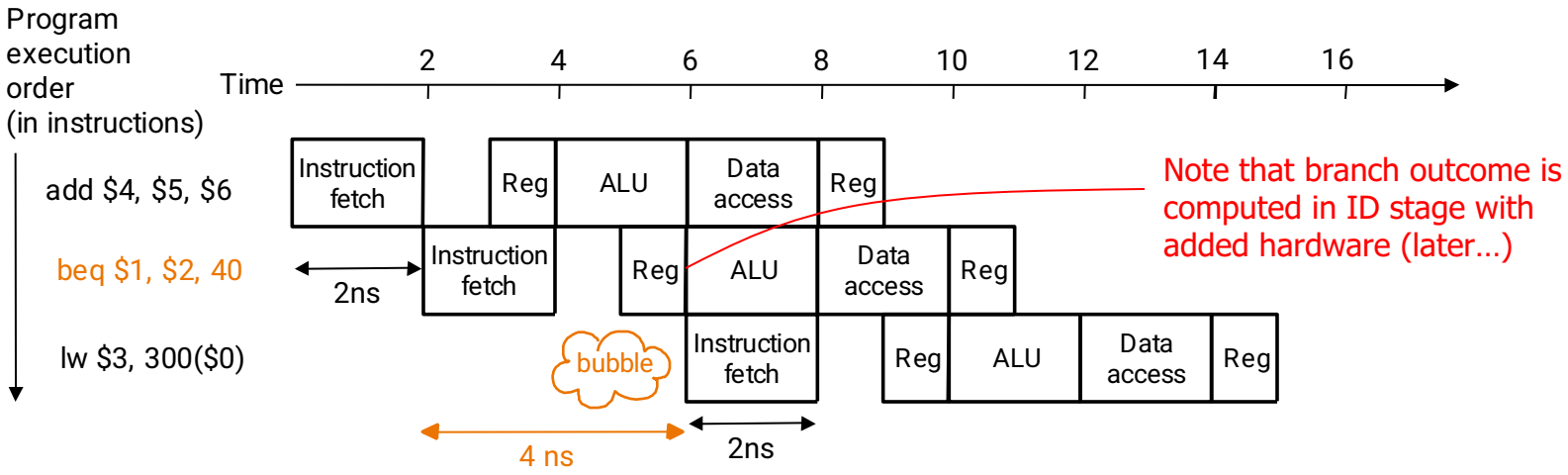
Structural Hazards

- Structural hazard: inadequate hardware to simultaneously support all instructions in the pipeline in the same clock cycle
- E.g., suppose single – not separate – instruction and data memory in pipeline below with one read port
  - then a structural hazard between first and fourth `lw` instructions



Control Hazards

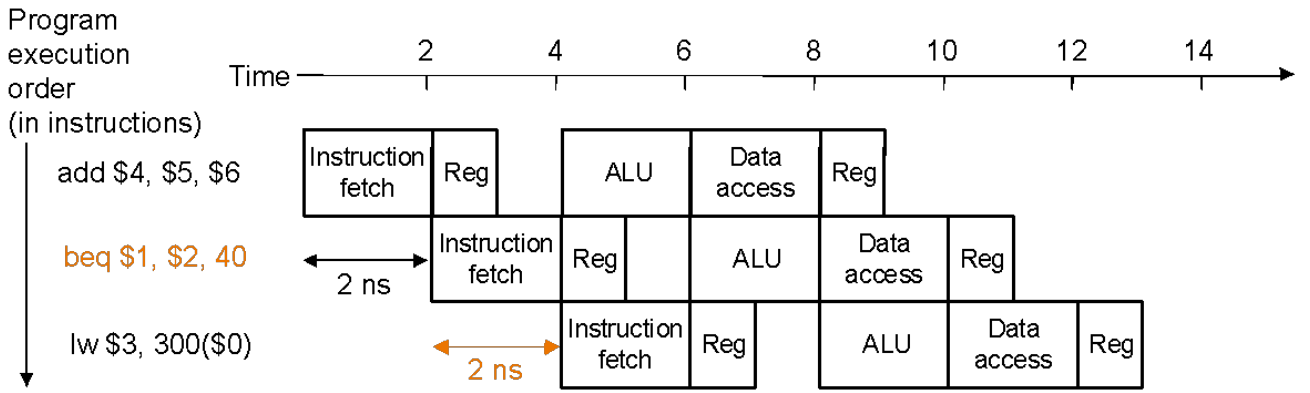
- Control hazard: need to make a decision based on the result of a previous instruction still executing in pipeline
- Solution 1 Stall the pipeline



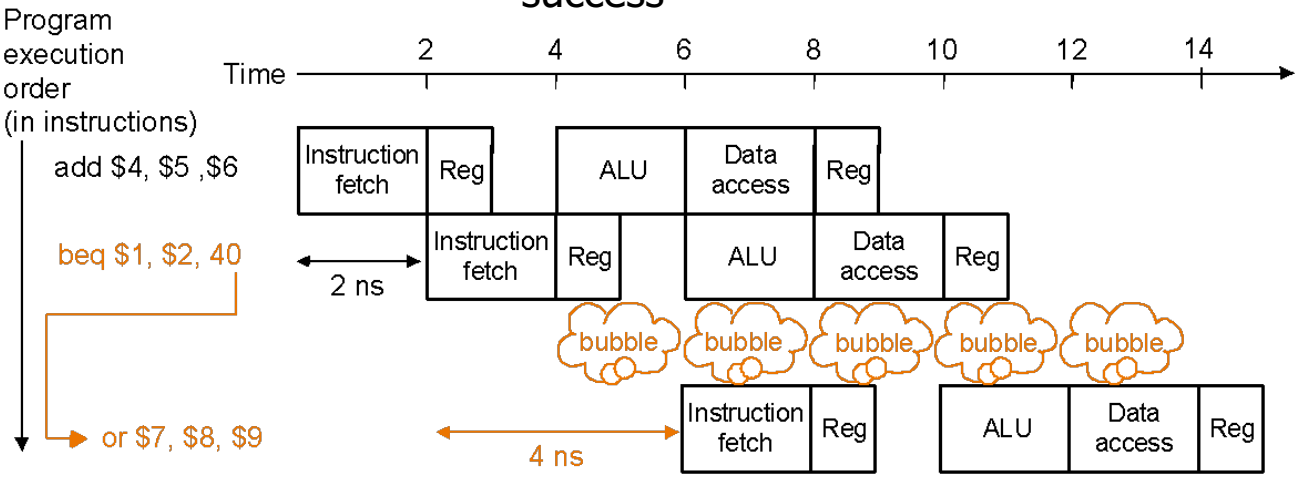
Pipeline stall

Control Hazards

- Solution 2 Predict branch outcome
  - e.g., predict branch-not-taken :



Prediction success

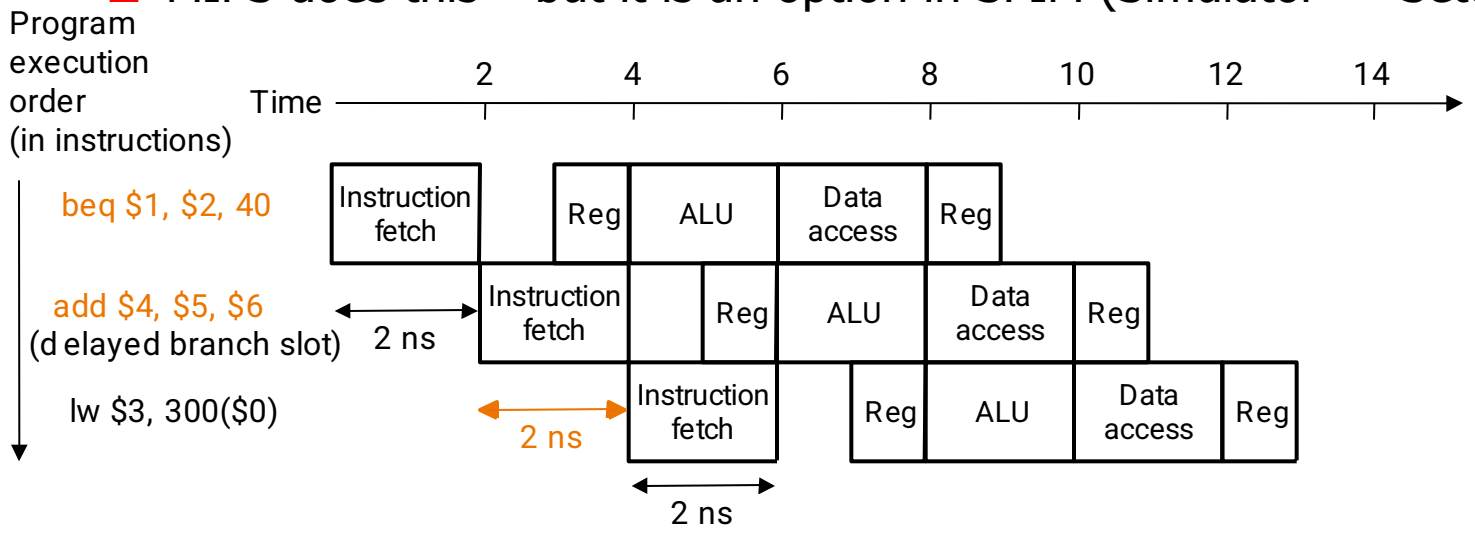


Prediction failure: undo (=flush)

```
lw
```

Control Hazards

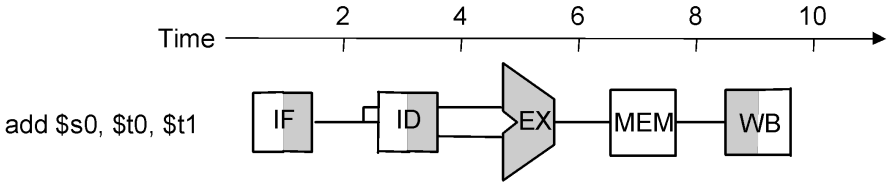
- Solution 3 Delayed branch: always execute the sequentially next statement with the branch executing after one instruction delay – compiler's job to find a statement that can be put in the slot that is independent of branch outcome
  - MIPS does this – but it is an option in SPIM (Simulator -> Settings)



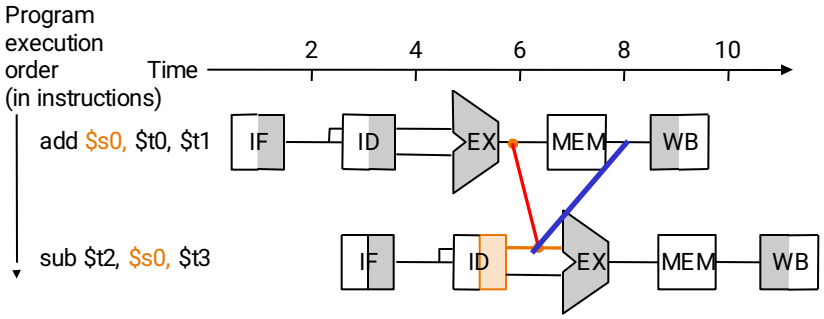
Delayed branch `beq` is followed by `add` that is independent of branch outcome

Data Hazards

- Data hazard: instruction needs data from the result of a previous instruction still executing in pipeline
- Solution Forward data if possible…



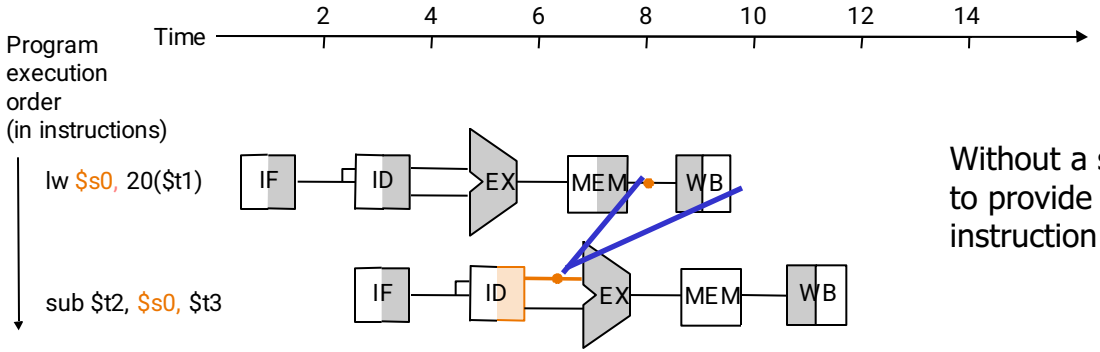
Instruction pipeline diagram: shade indicates use – left=write, right=read



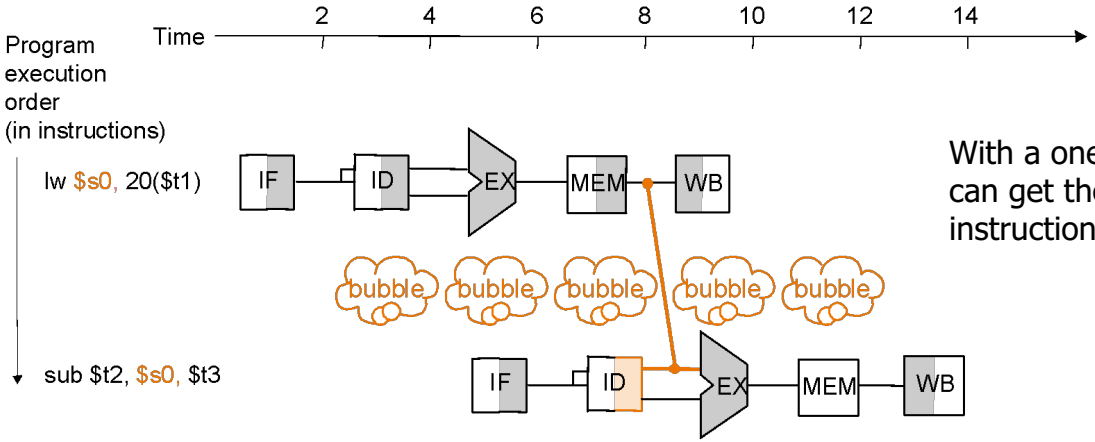
Without forwarding – blue line – data has to go back in time; with forwarding – red line – data is available in time

Data Hazards

- Forwarding may not be enough
  - e.g., if an R-type instruction following a load uses the result of the load – called load-use data hazard



Without a stall it is impossible to provide input to the `sub` instruction in time



With a one-stage stall, forwarding can get the data to the `sub` instruction in time

Reordering Code to Avoid Pipeline Stall (Software Solution)

- Example:

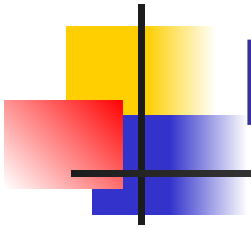
```
lw $t0, 0($t1)
lw $t2, 4($t1)
sw $t2, 0($t1)
sw $t0, 4($t1)
```

Data hazard

- Reordered code:

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

Interchanged

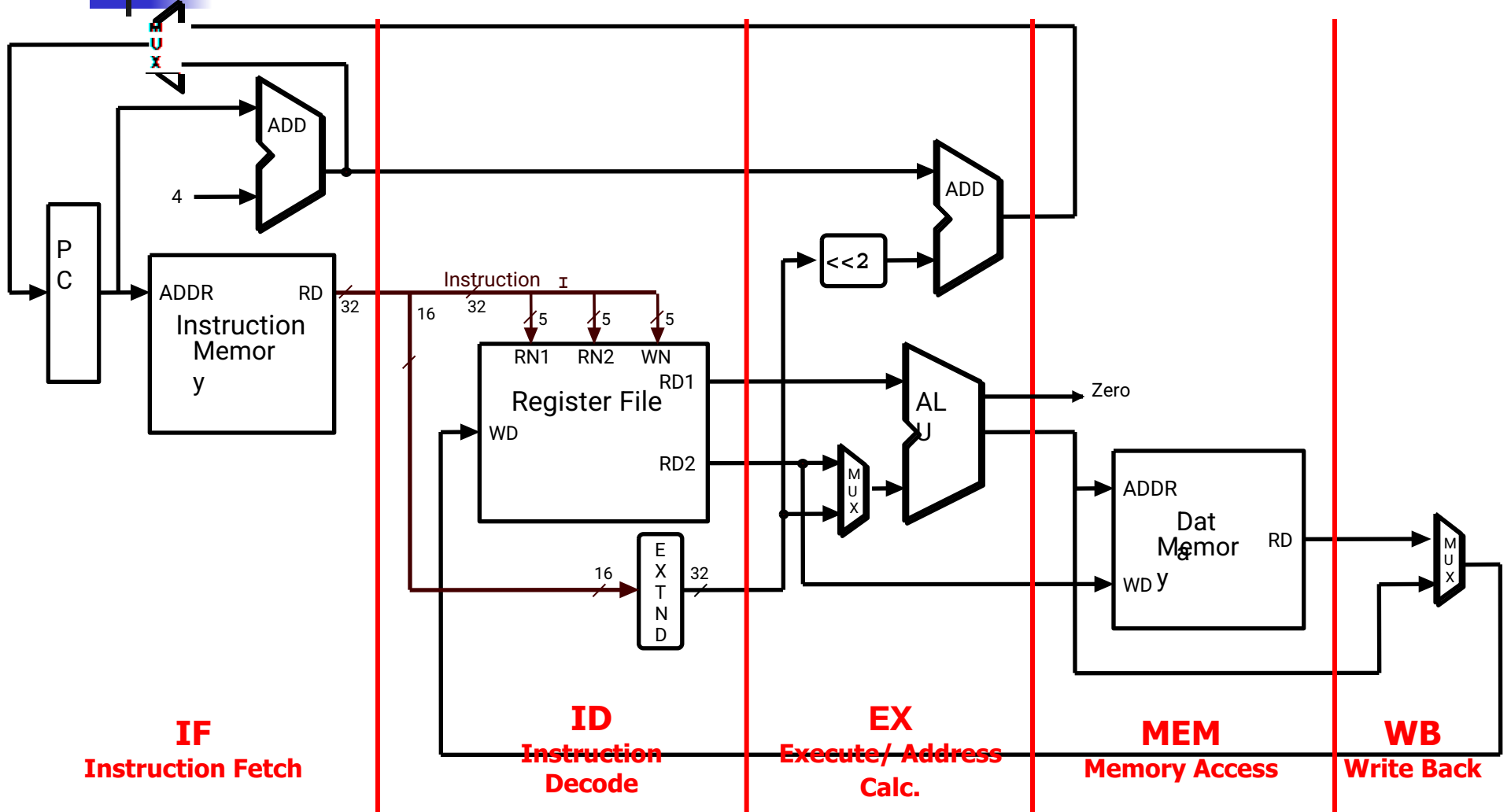


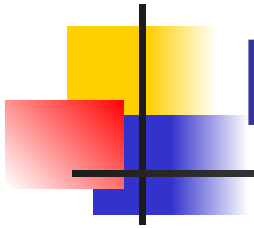
Pipelined Datapath

- We now move to actually building a pipelined datapath
- First recall the 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)
- Review: single-cycle processor
  - all 5 steps done in a single clock cycle
  - dedicated hardware required for each step

- *What happens if we break the execution into multiple cycles, but keep the extra hardware?*

Review - Single-Cycle Datapath "Steps"

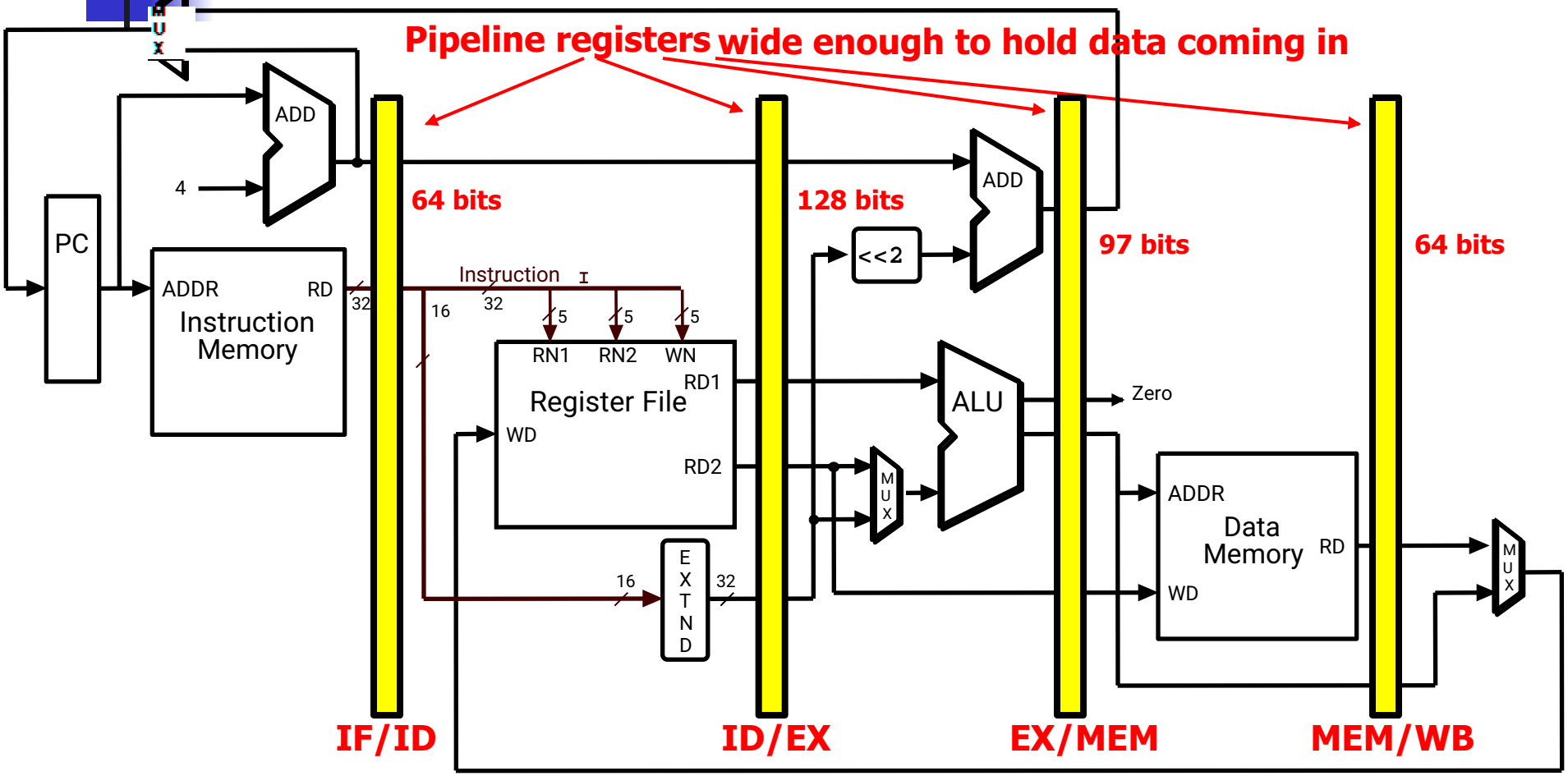


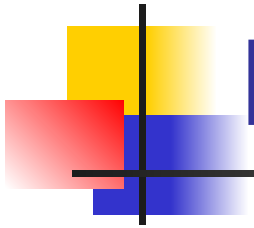


Pipelined Datapath – Key Idea

- *What happens if we break the execution into multiple cycles, but keep the extra hardware?*
  - Answer: *We may be able to start executing a new instruction at each clock cycle* - pipelining
- …but we shall need extra registers to hold data between cycles – pipeline registers

Pipelined Datapath



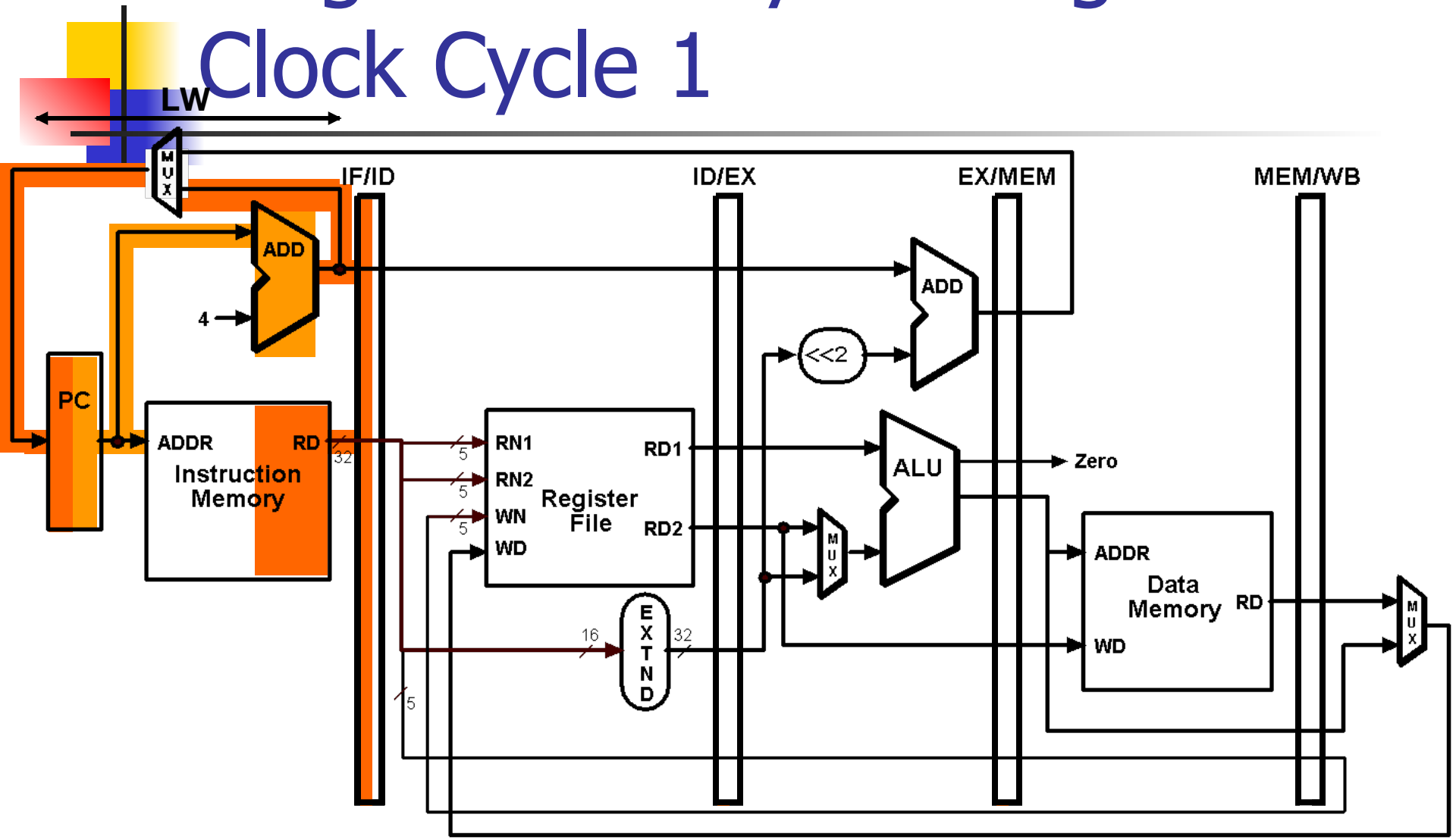


Pipelined Example

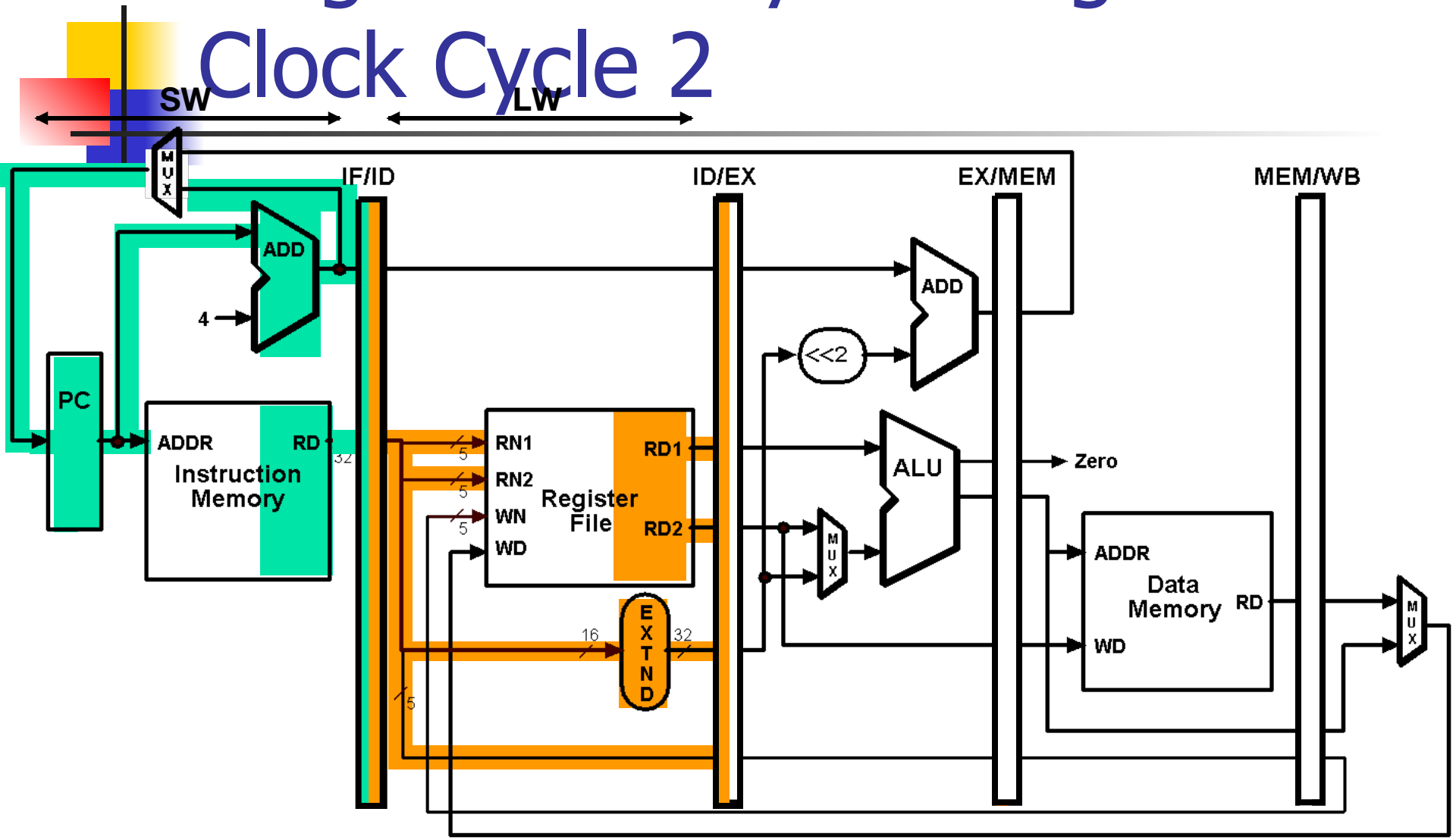
- Consider the following instruction sequence:

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

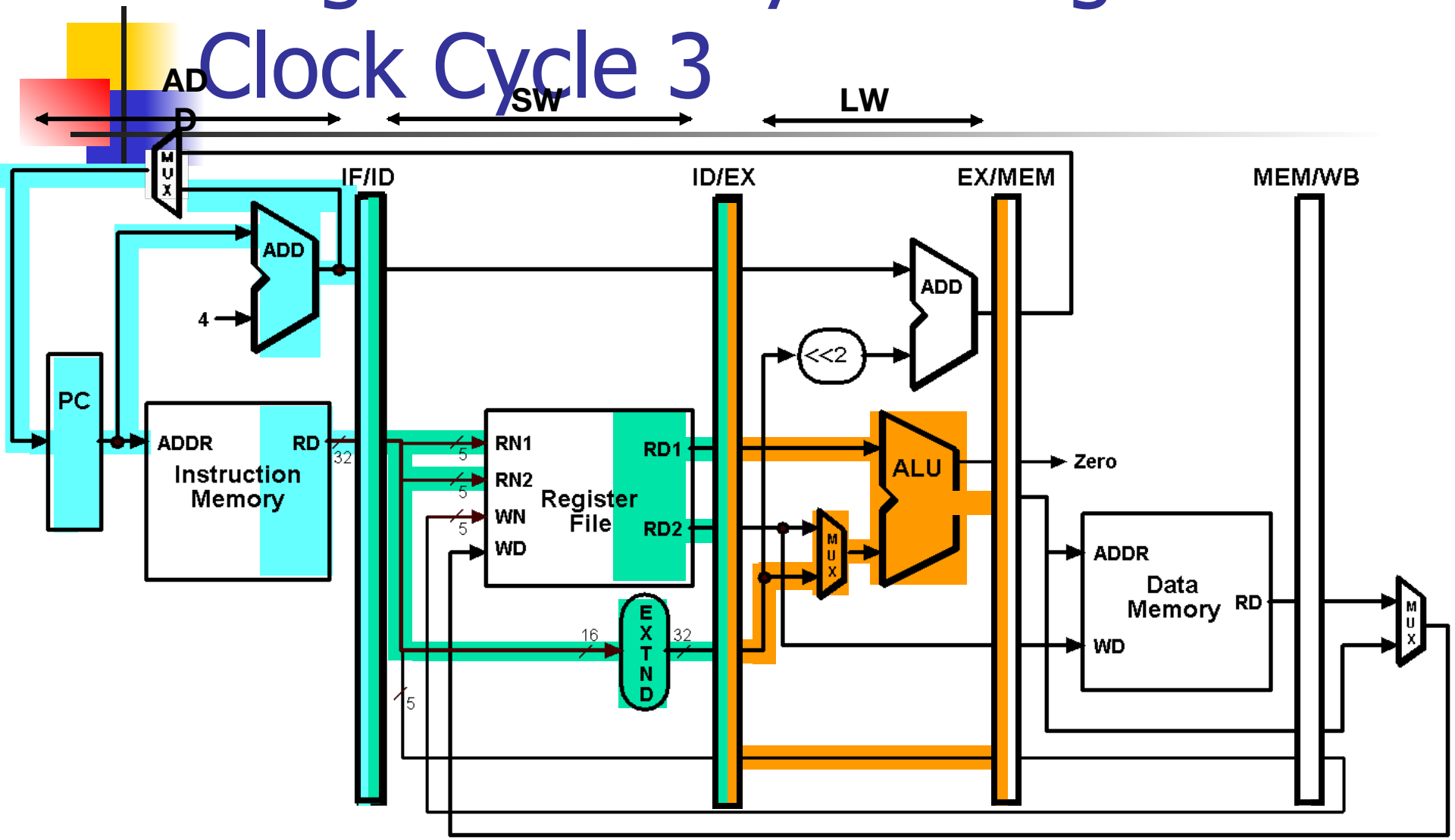
Single-Clock-Cycle Diagram: Clock Cycle 1



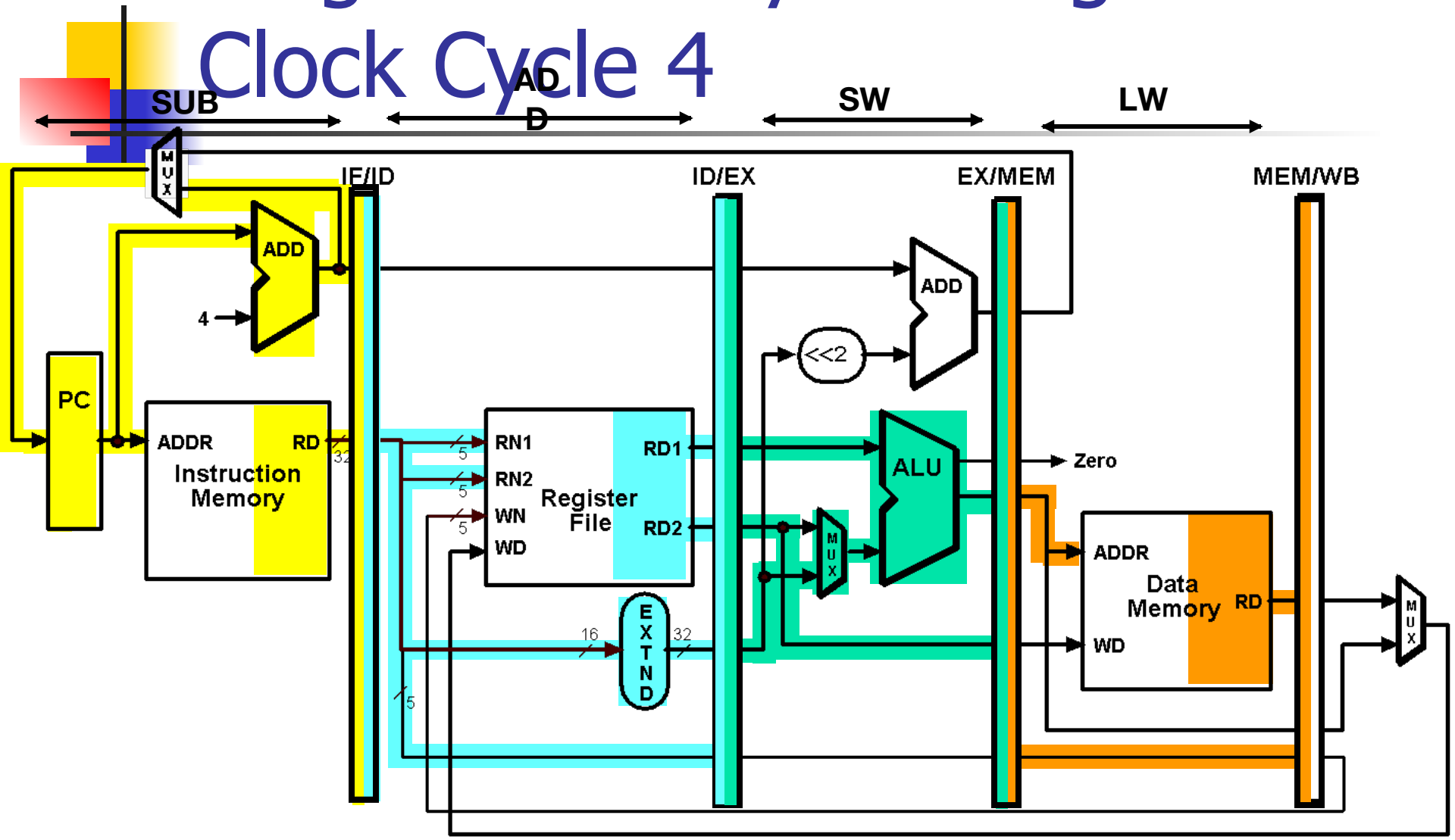
Single-Clock-Cycle Diagram: Clock Cycle 2



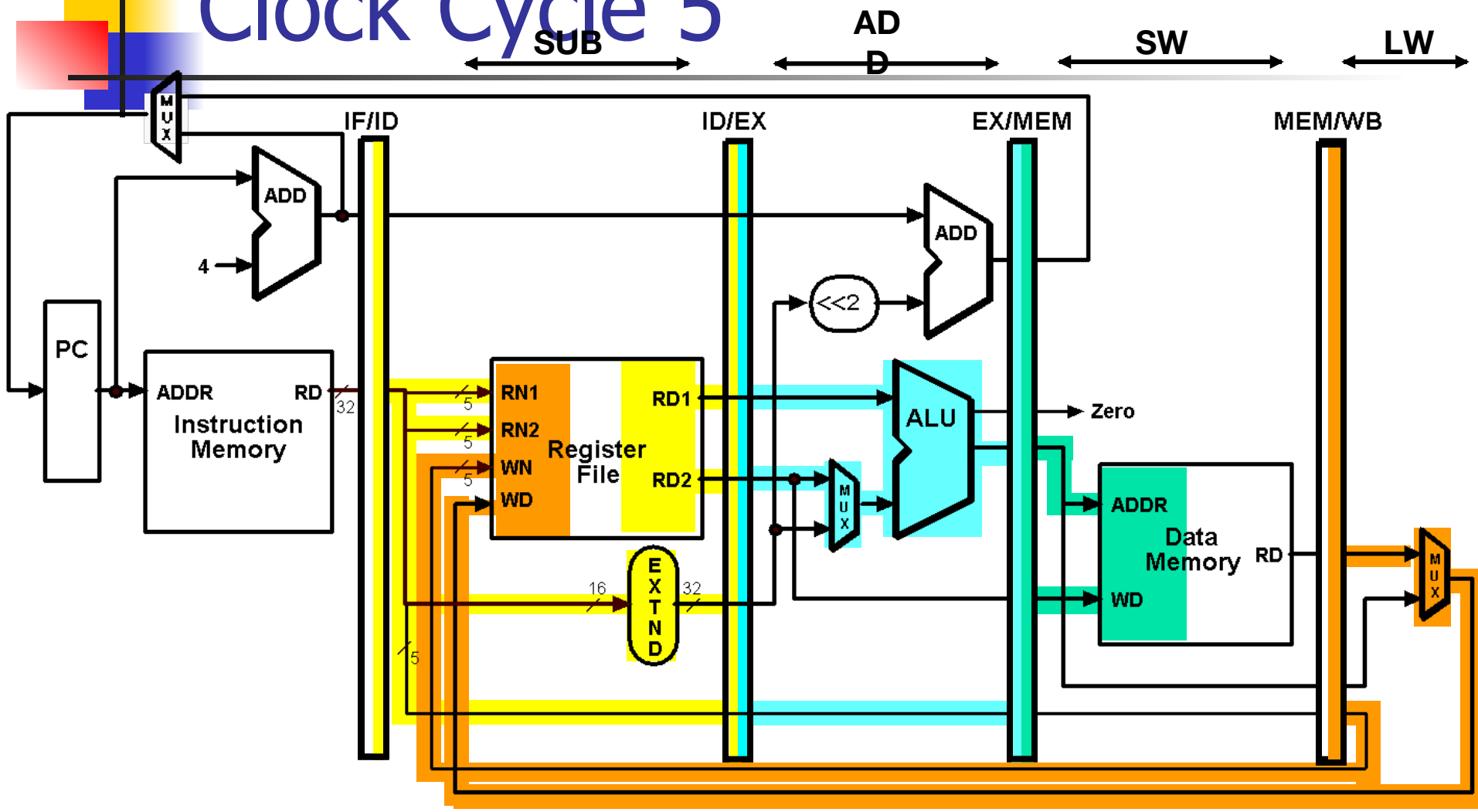
Single-Clock-Cycle Diagram: Clock Cycle 3



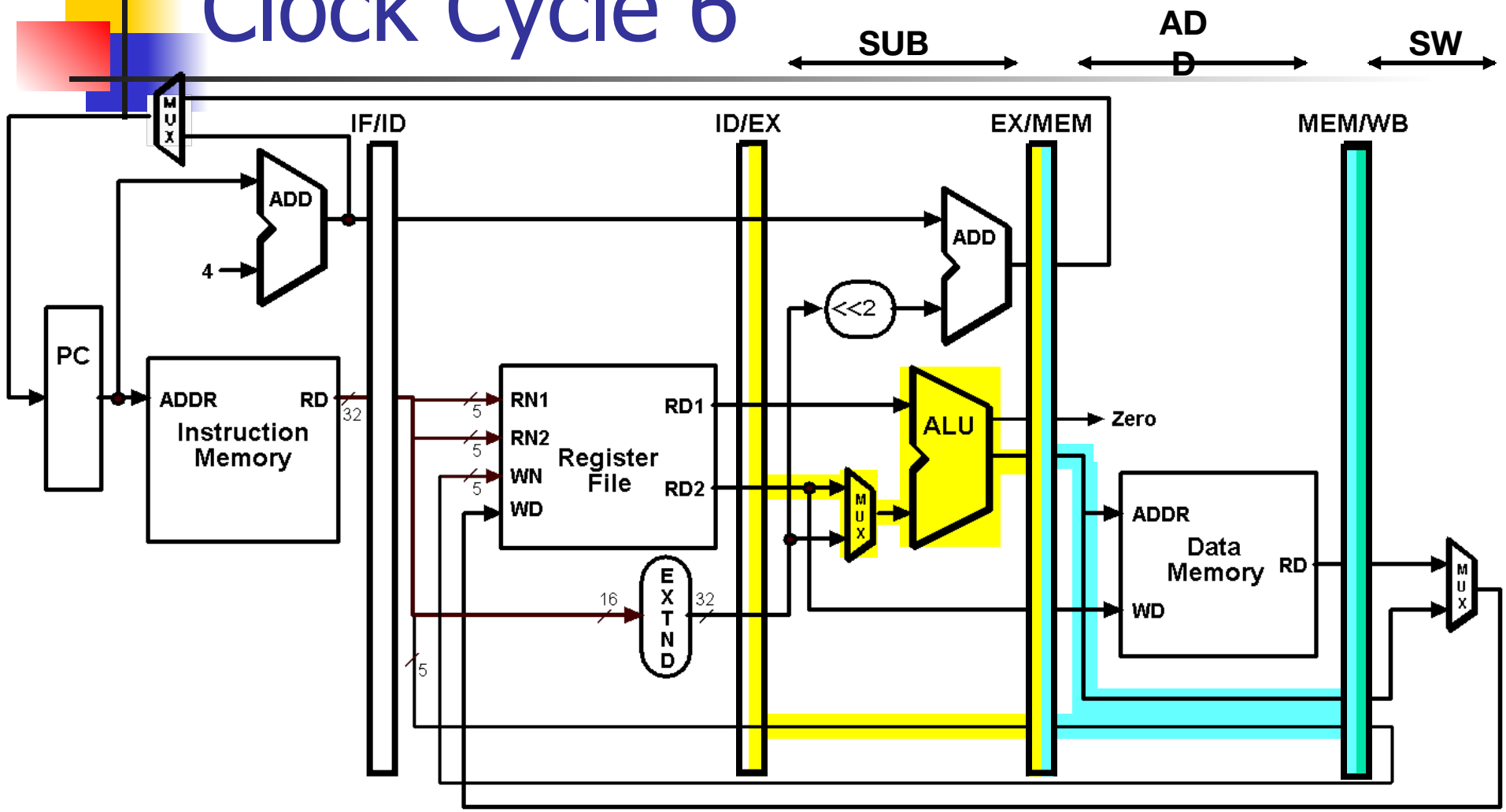
Single-Clock-Cycle Diagram: Clock Cycle 4



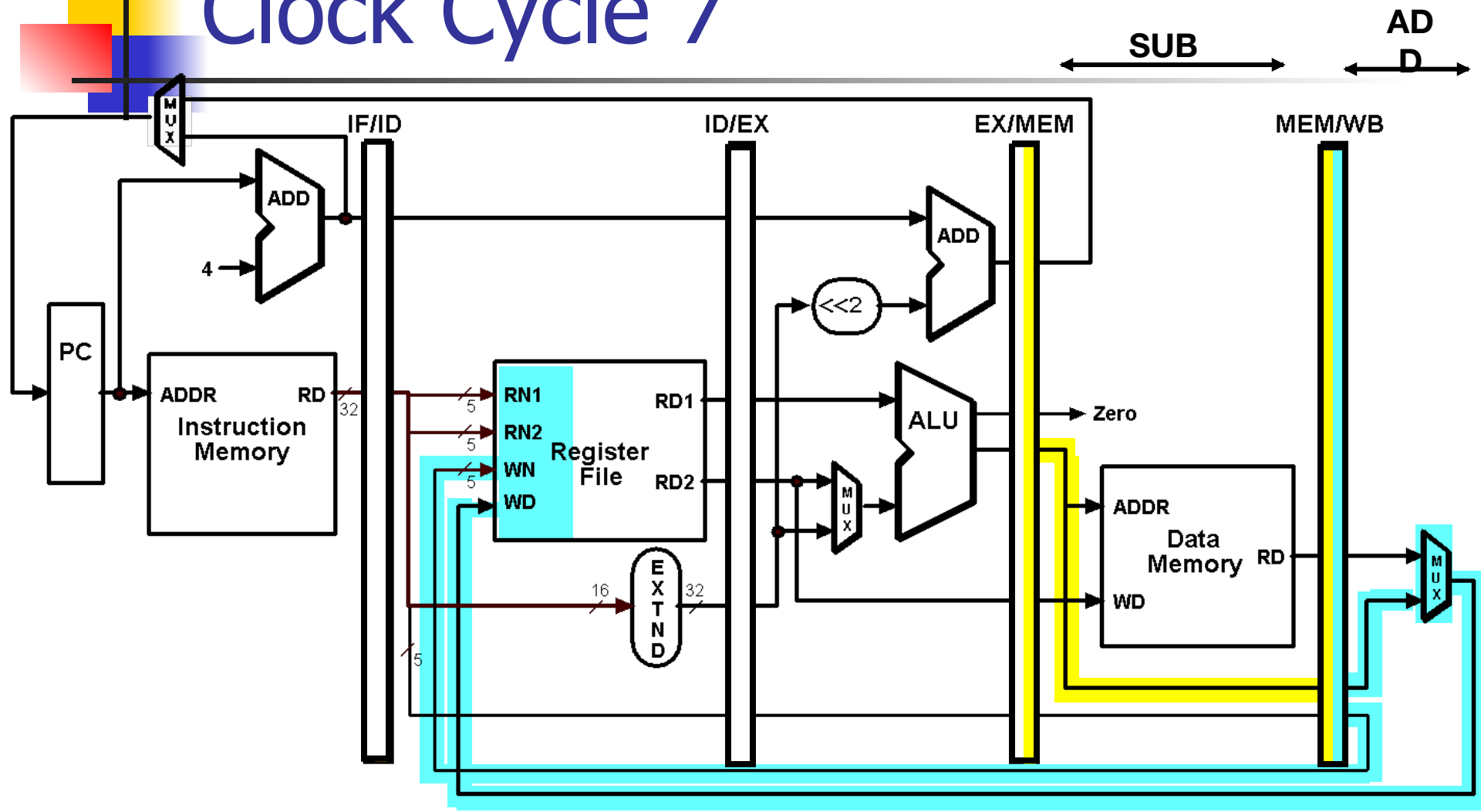
Single-Clock-Cycle Diagram: Clock Cycle 5



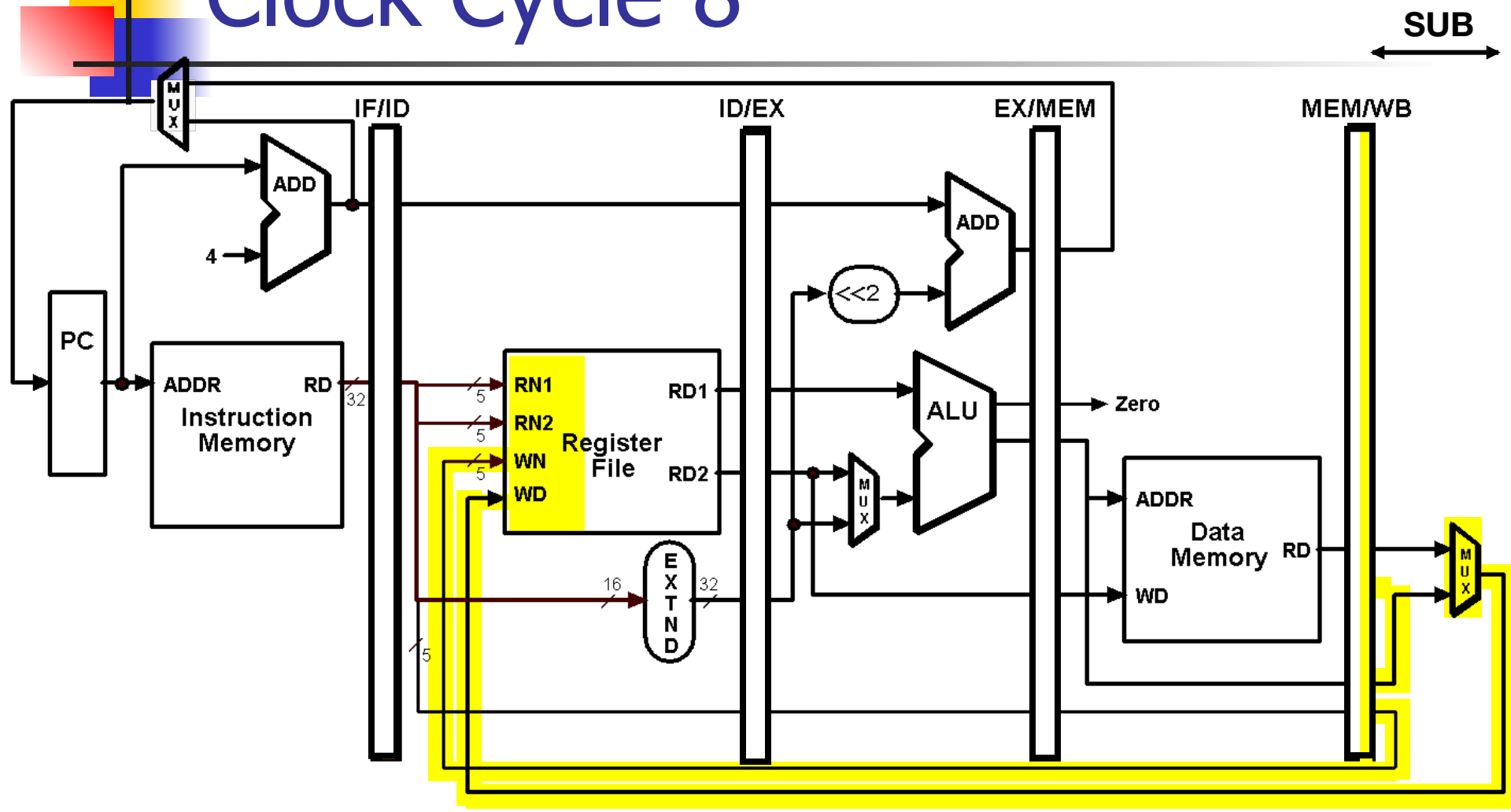
Single-Clock-Cycle Diagram: Clock Cycle 6



Single-Clock-Cycle Diagram: Clock Cycle 7



Single-Clock-Cycle Diagram: Clock Cycle 8



Alternative View – Multiple-Clock-Cycle Diagram