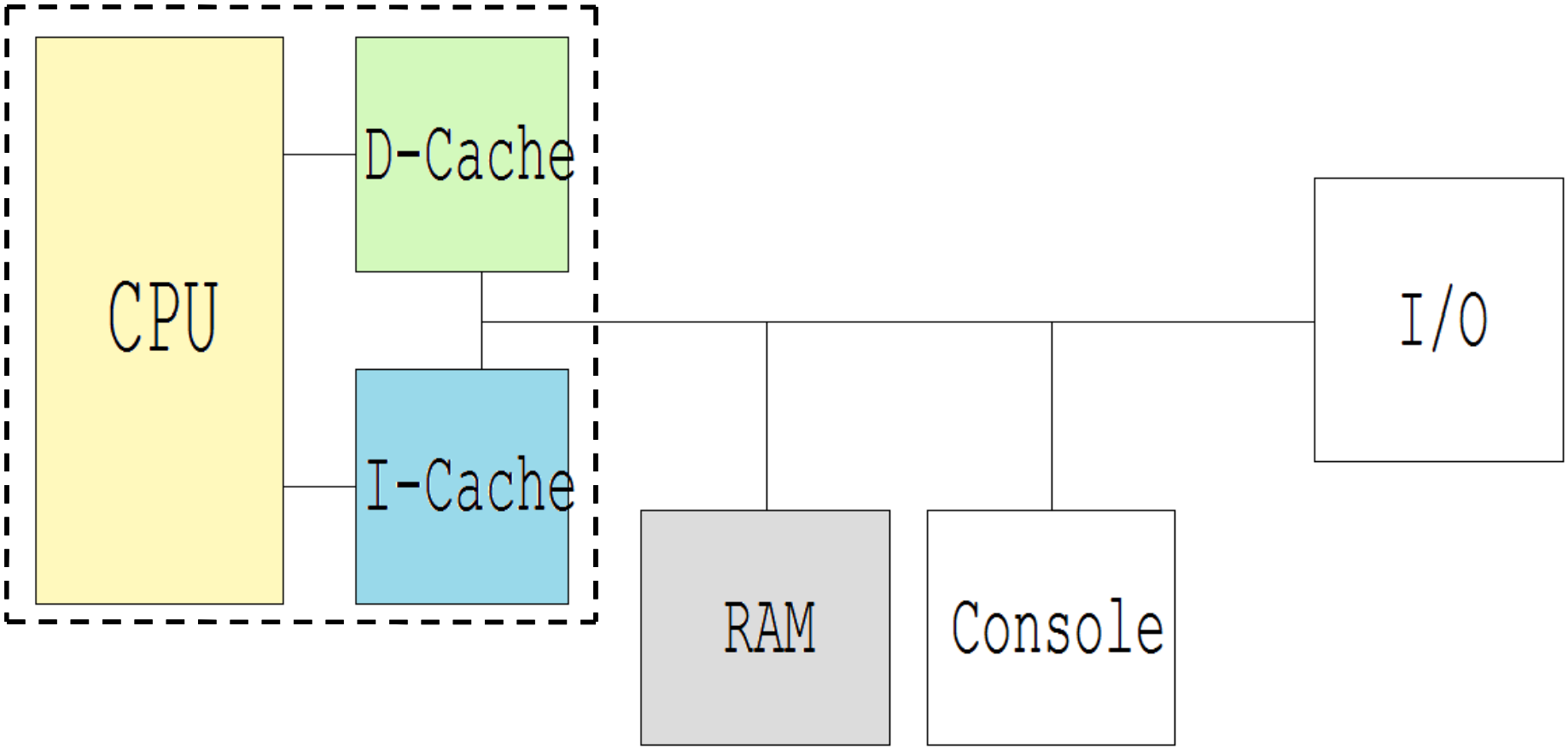
LAB3: IMPROVING MIPS PERFORMANCE WITH PIPELINING

Electrical and Computer Engineering
University of Cyprus

Previous Labs: MIPS single-cycle with Memory System



In this Lab: Pipelining

- You are expected to:
  - Understand the concept
  - Be familiar with the 5 MIPS pipeline stages
  - Understand the three pipeline hazards and their solutions.

PIPELINING

- Technique in which the execution of several instructions is *overlapped.*
- Each instruction is broken into several stages.
- Stages can operate concurrently PROVIDED WE HAVE SEPARATE RESOURCES FOR EACH STAGE! => each step uses a different functional unit.
- *Note*: execution time for a single instruction is NOT improved. **Throughput** of several instructions is improved.

Major Pipeline Benefit = Performance

- Ideal Performance
  - Time/instruction = non-piped-time/#stages
  - This is an asymptote of course, but +10% is commonly achieved

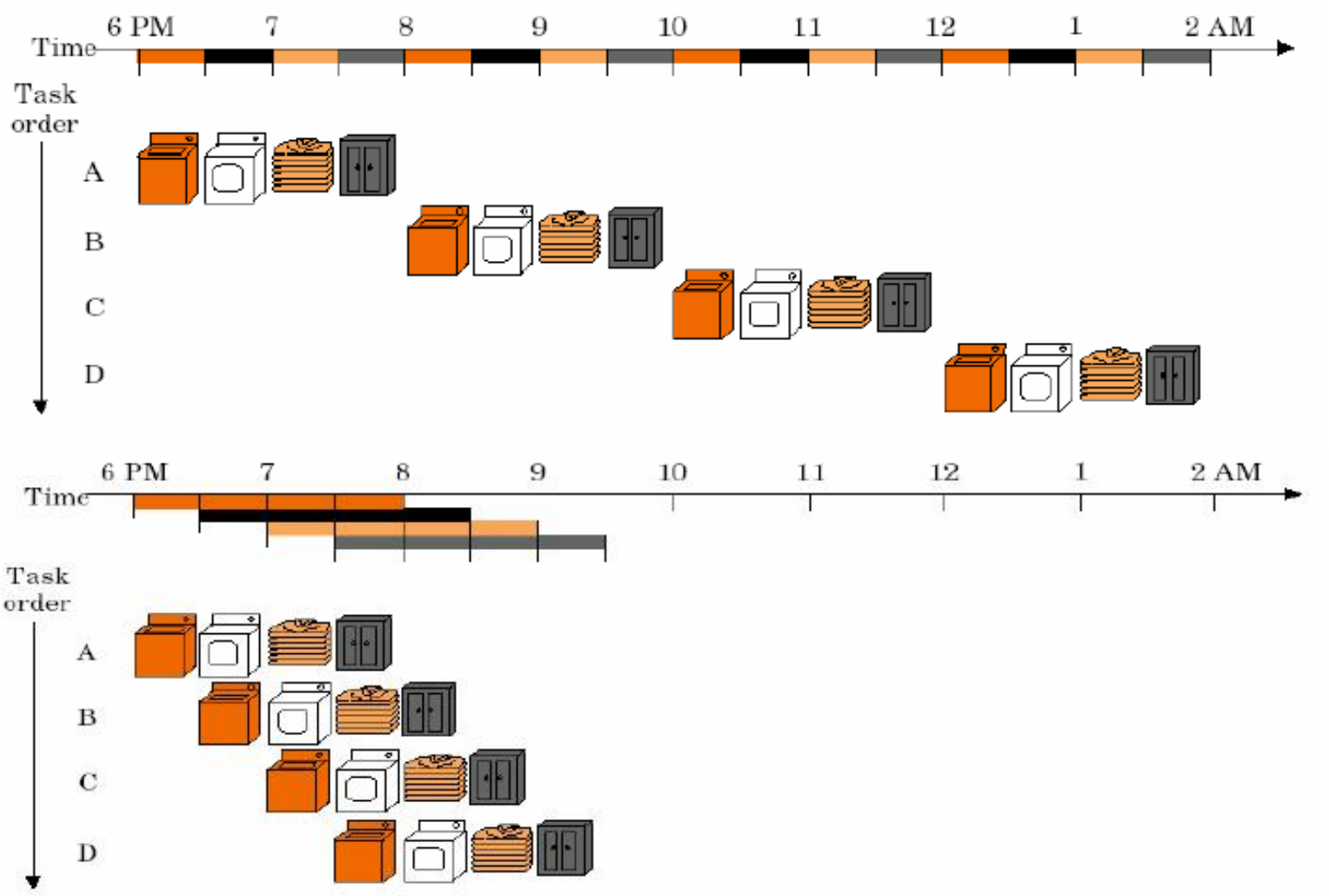
Two ways to view the performance mechanism

- Reduced CPI (i.e. non-piped to piped change)
  - Close to 1 instruction/cycle if you're lucky
- Reduced cycle-time (i.e. increasing pipeline depth)
  - Work split into more stages
  - Simpler stages result in faster clock cycles

Other Pipeline Benefits

- Completely hardware mechanism
- All modern machines are pipelined
  - This was the key technique to advancing performance in the 80's
  - In the 90's the move was to multiple pipelines
- Beware, no benefit is totally free/good

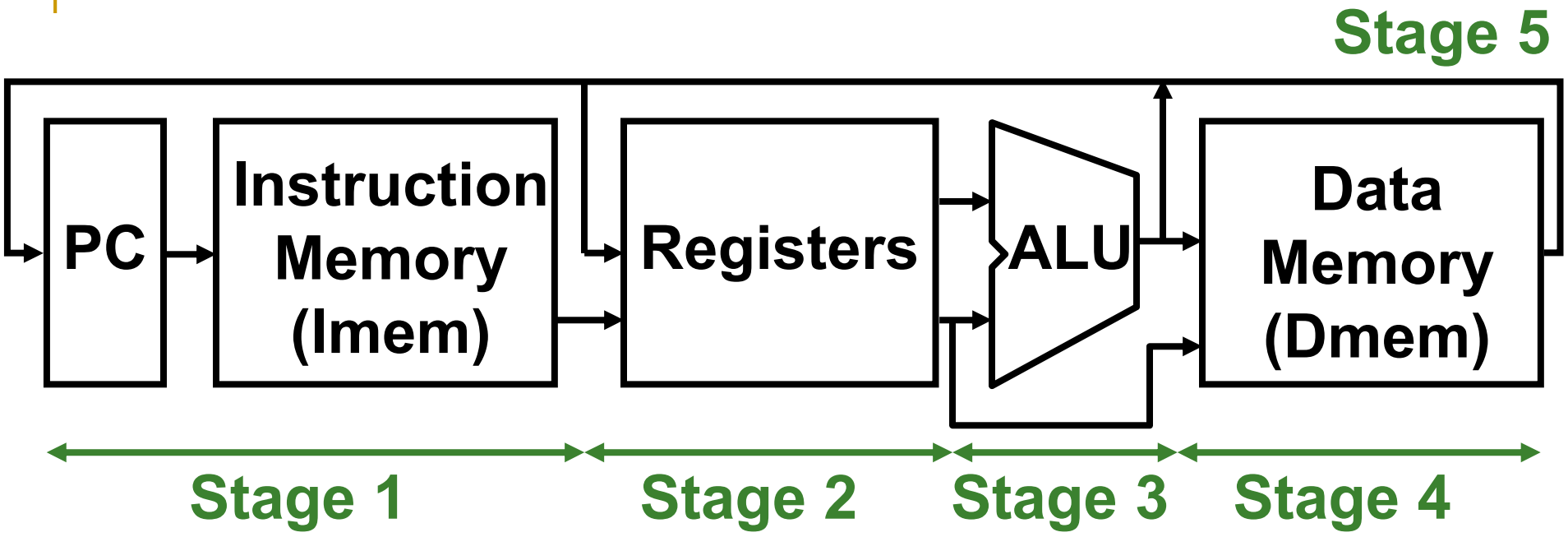
Laundry analogy to pipelining



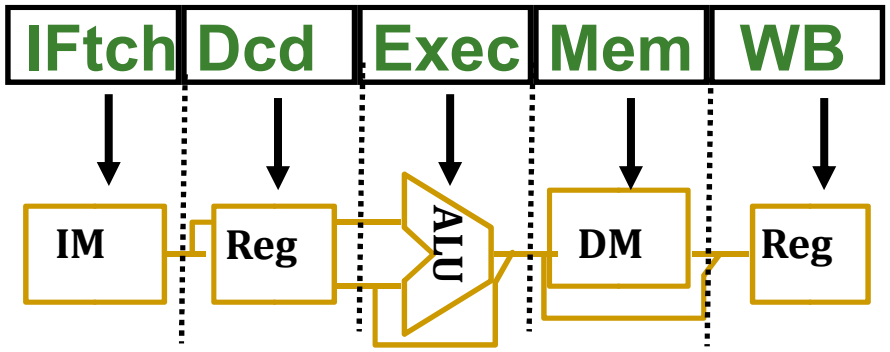
Implementation of a RISC (Unpipelined, Multicycle)

- Implementation of an integer subset of a RISC architecture that takes at most 5 clock cycles.
  - Instruction Fetch (IF)
  - Instruction Decode/Register Fetch (ID)
  - Execution/Effective Address Calculation (EX)
  - Memory Access (MEM)
  - Write-Back (WB)

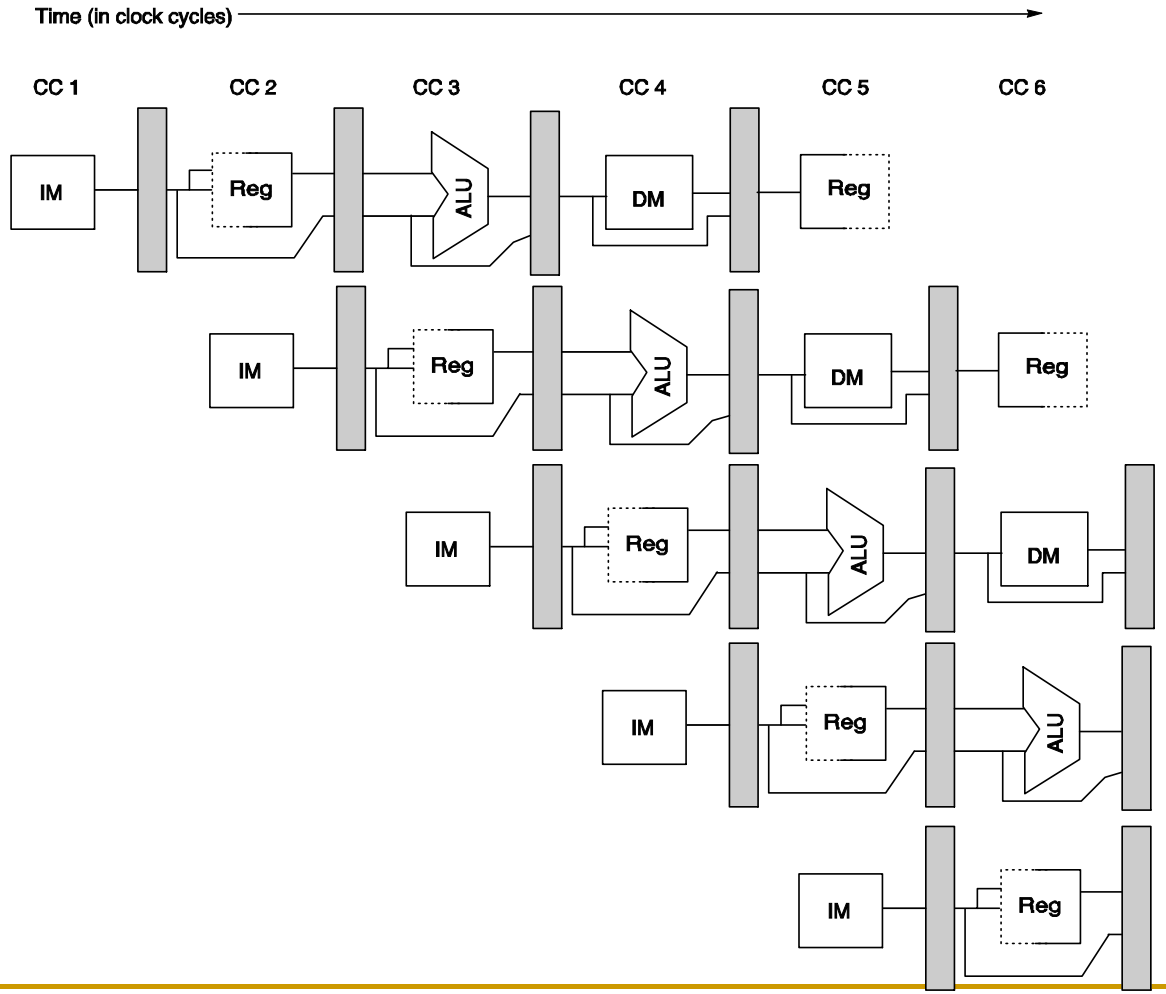
Review: Single-cycle Datapath for MIPS



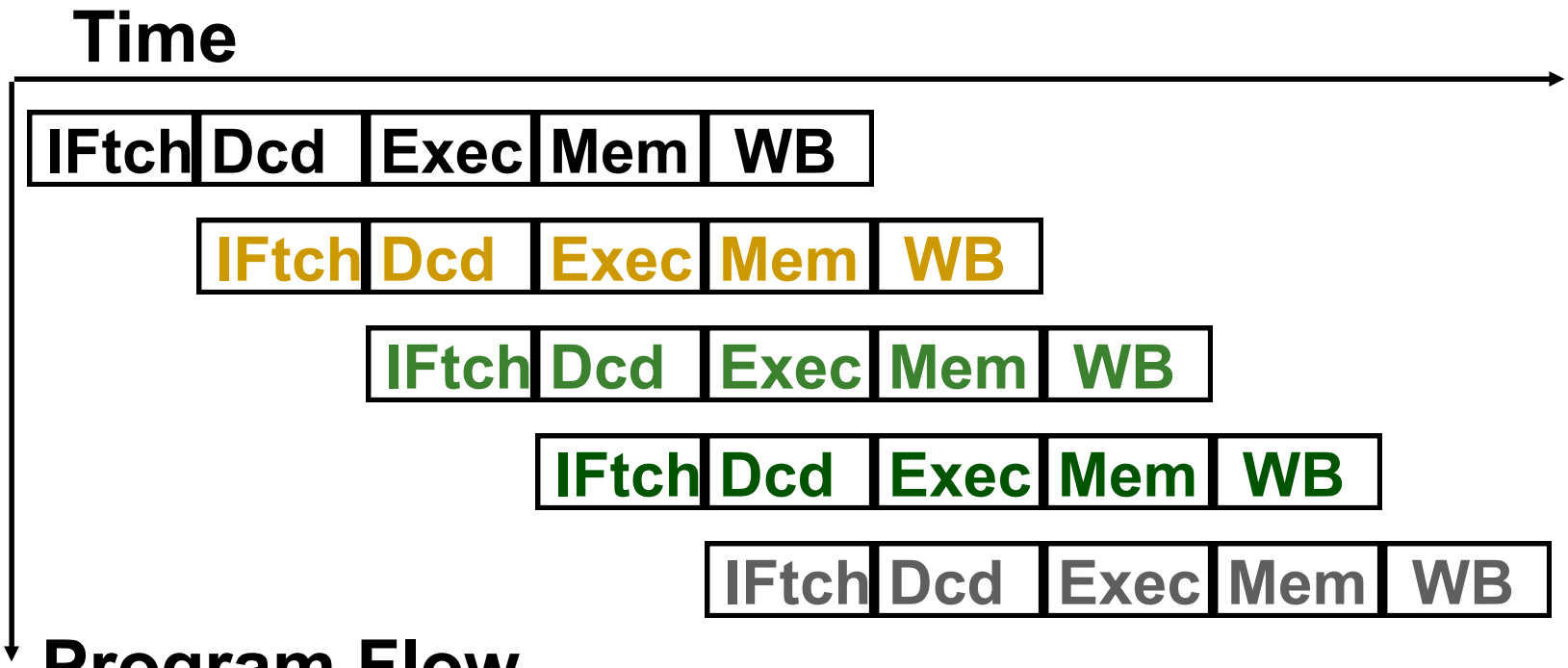
- Use datapath figure to represent pipeline



Classic 5 Stage Pipeline for a RISC Processor



Pipelined Execution Representation



- To simplify pipeline, every instruction takes same number of steps, called stages
- One clock cycle per stage

Important Pipeline Characteristics

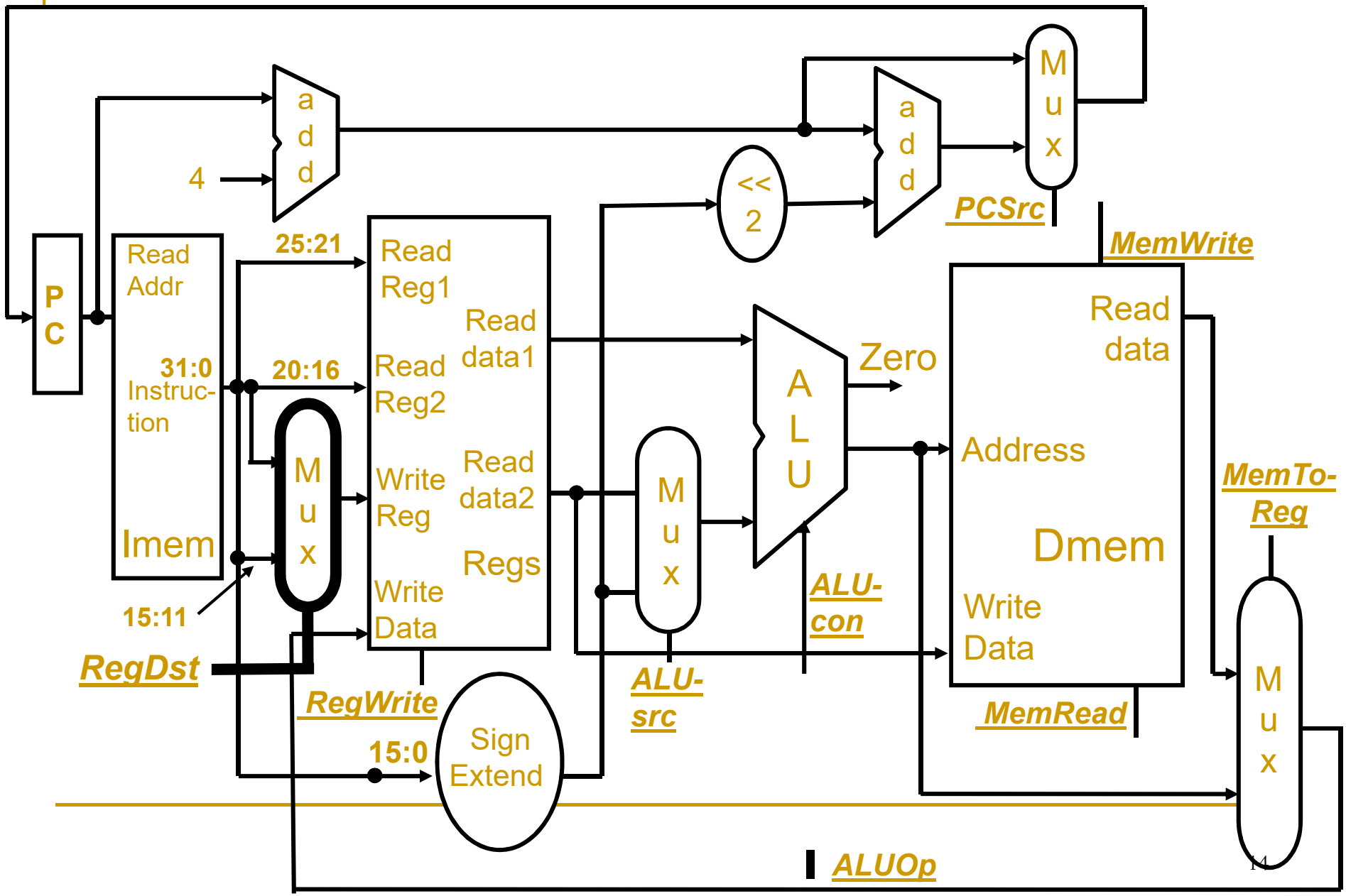
- **Latency**
  - Time it takes an instruction to go through the pipe
  - Latency = # stages * stage-delay
  - Dominant feature if there are a lot of exceptions…
- **Throughput**
  - Determined by the rate at which instructions can start/finish
  - Dominant feature if there are few exceptions

Pipelining Lessons

- Pipelining doesn't help latency (execution time) of single task, it helps throughput of entire workload
- Multiple tasks operating simultaneously using different resources
- *Potential* speedup = Number of pipe stages
- Time to "fill" pipeline and time to "drain" it reduces speedup

- Pipeline rate limited by slowest pipeline stage
- Unbalanced lengths of pipe stages also reduces speedup

Single Cycle Datapath



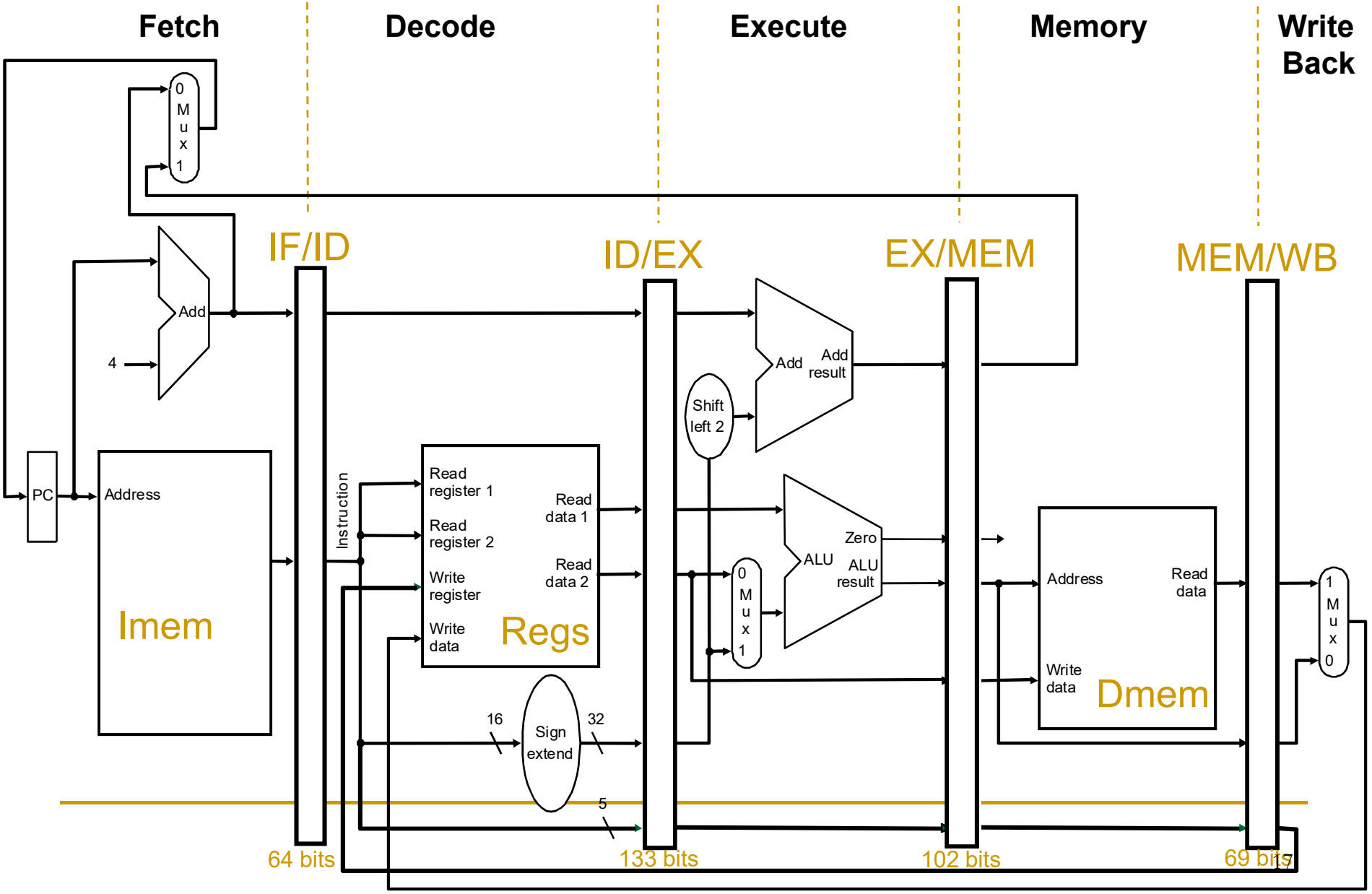
Required Changes to Datapath

- Introduce registers to separate 5 stages by putting IF/ID, ID/EX, EX/MEM, and MEM/WB registers in the datapath.
- Next PC value is computed in the 3rd step, but we need to bring in next instruction in the next cycle.
- Branch address is computed in 3rd stage. With pipeline, the PC value has changed! Must carry the PC value along with instruction.

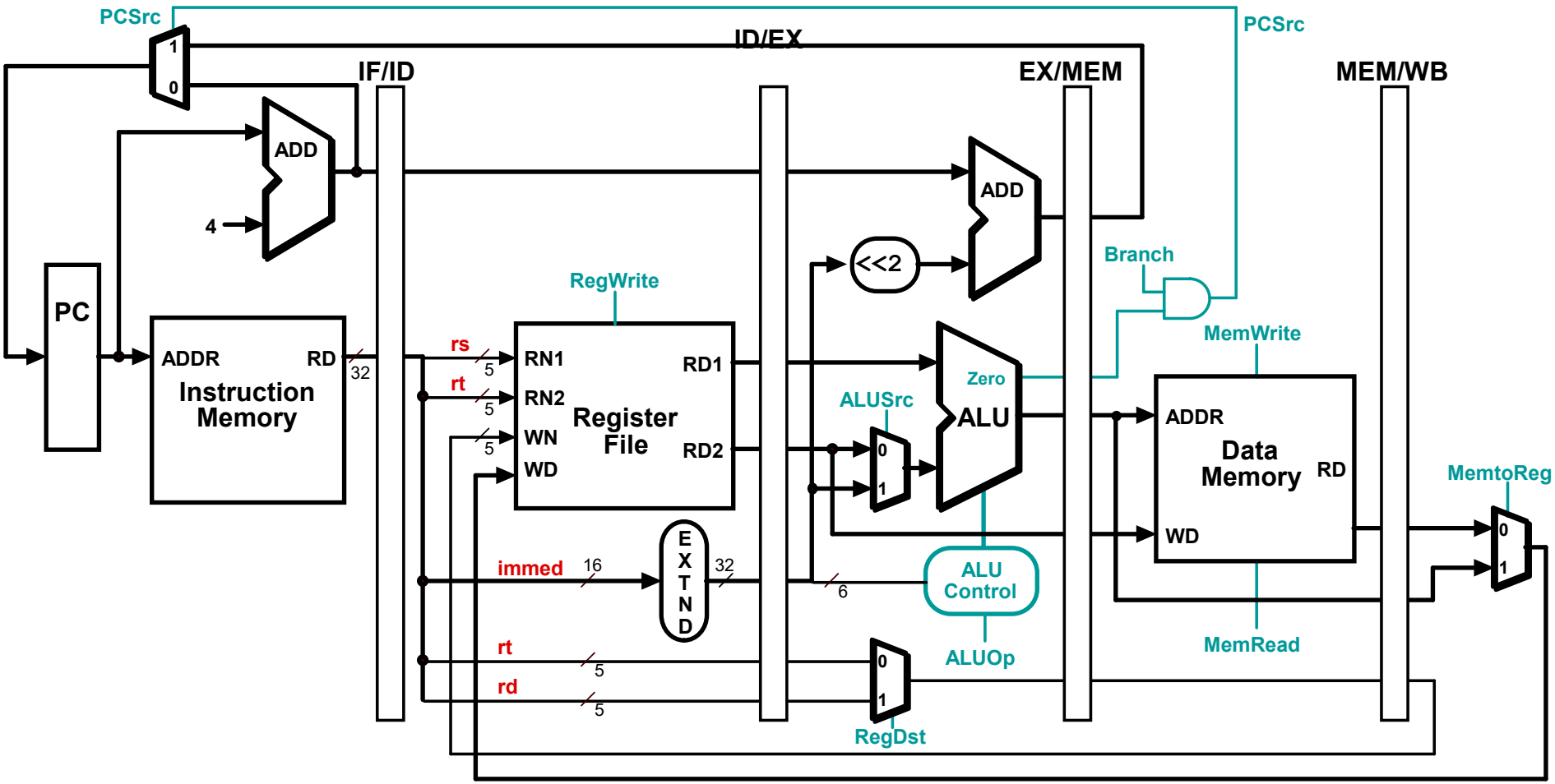
Changes to Datapath (cont.)

- For lw instruction, we need write register address at stage 5. But the IR is now occupied by another instruction! So, we must carry the IR destination field as we move along the stages. See connection in fig.

Pipelined Datapath (with Pipeline Regs)

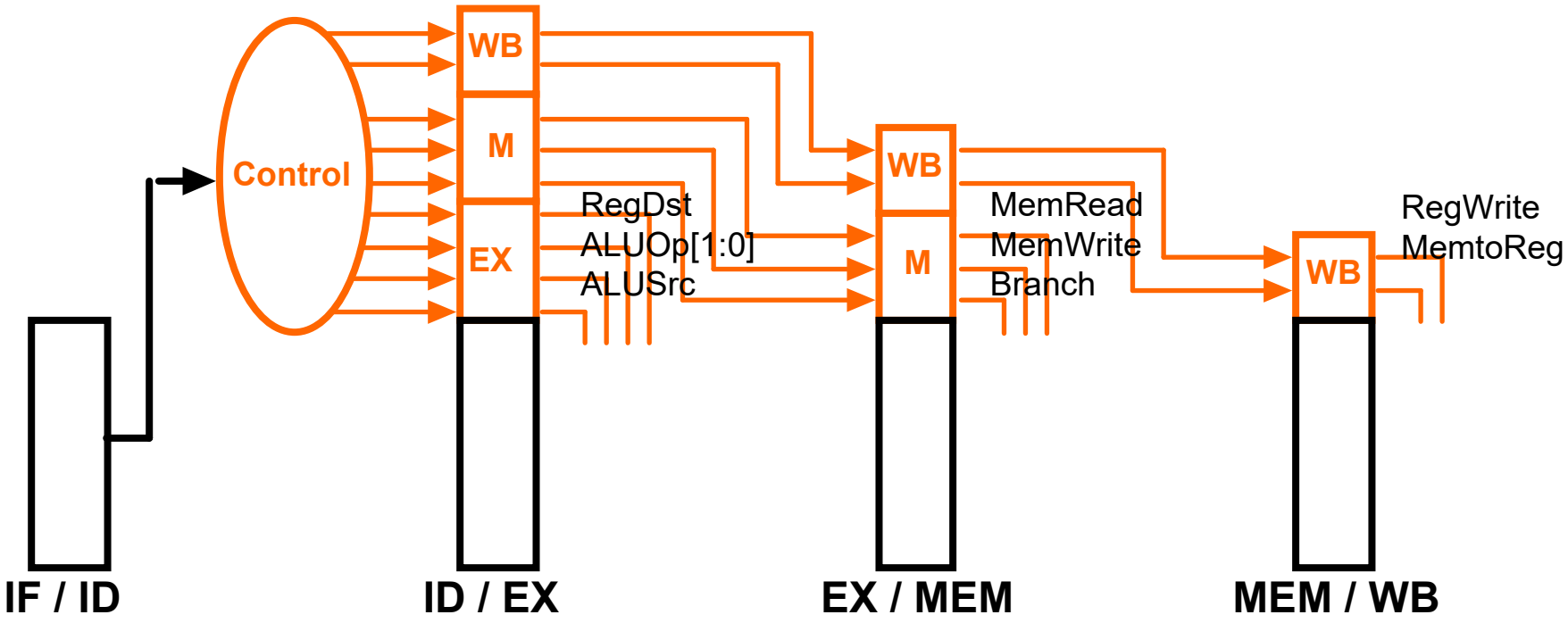


Pipelined Datapath (with Control Signals)



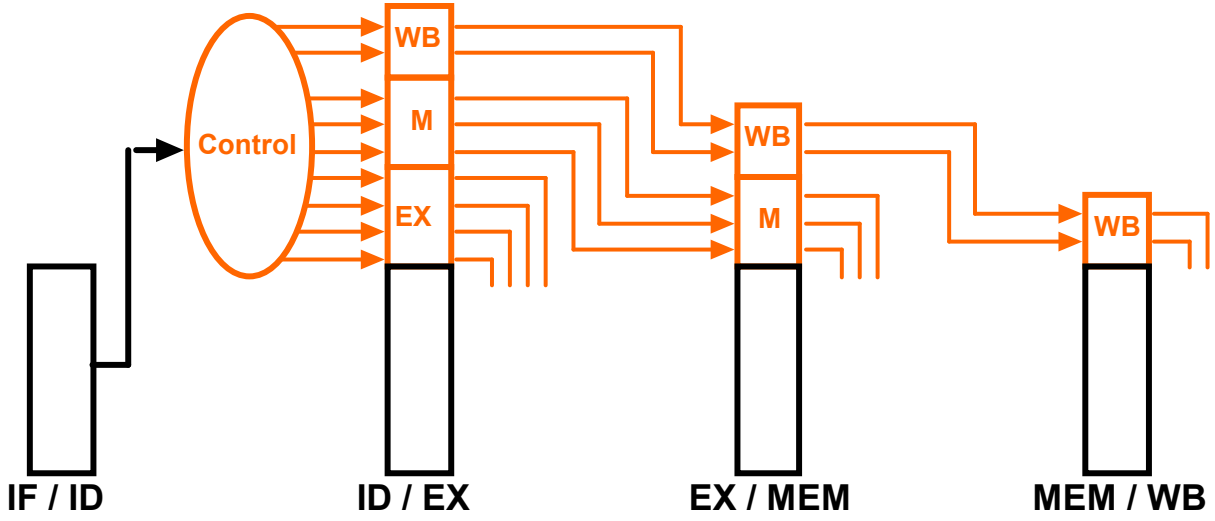
Control for Pipelined Datapath

- Basic approach: build on single-cycle control
  - Place control unit in ID stage
  - Pass control signals to following stages

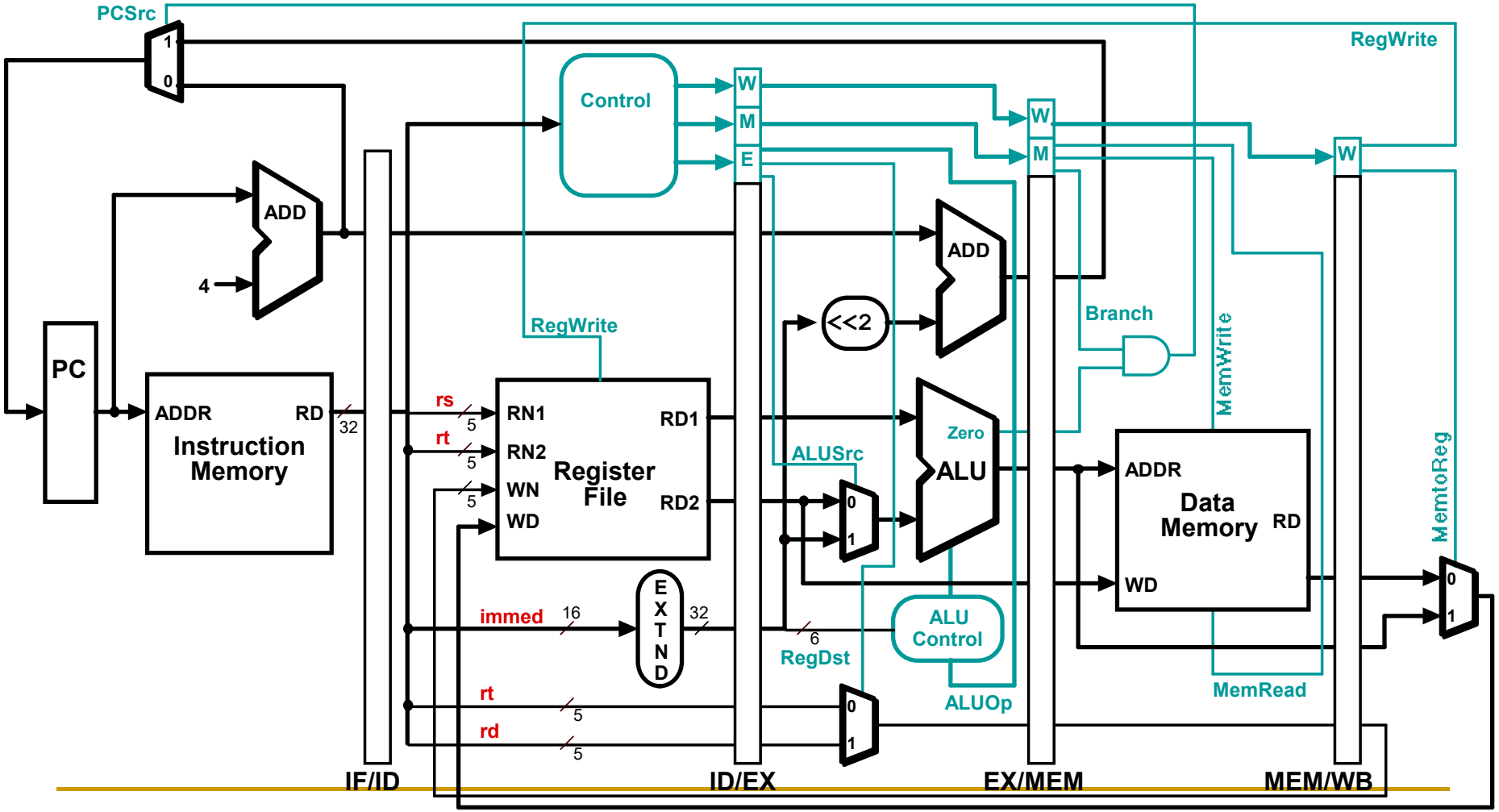


Control for Pipelined Datapath

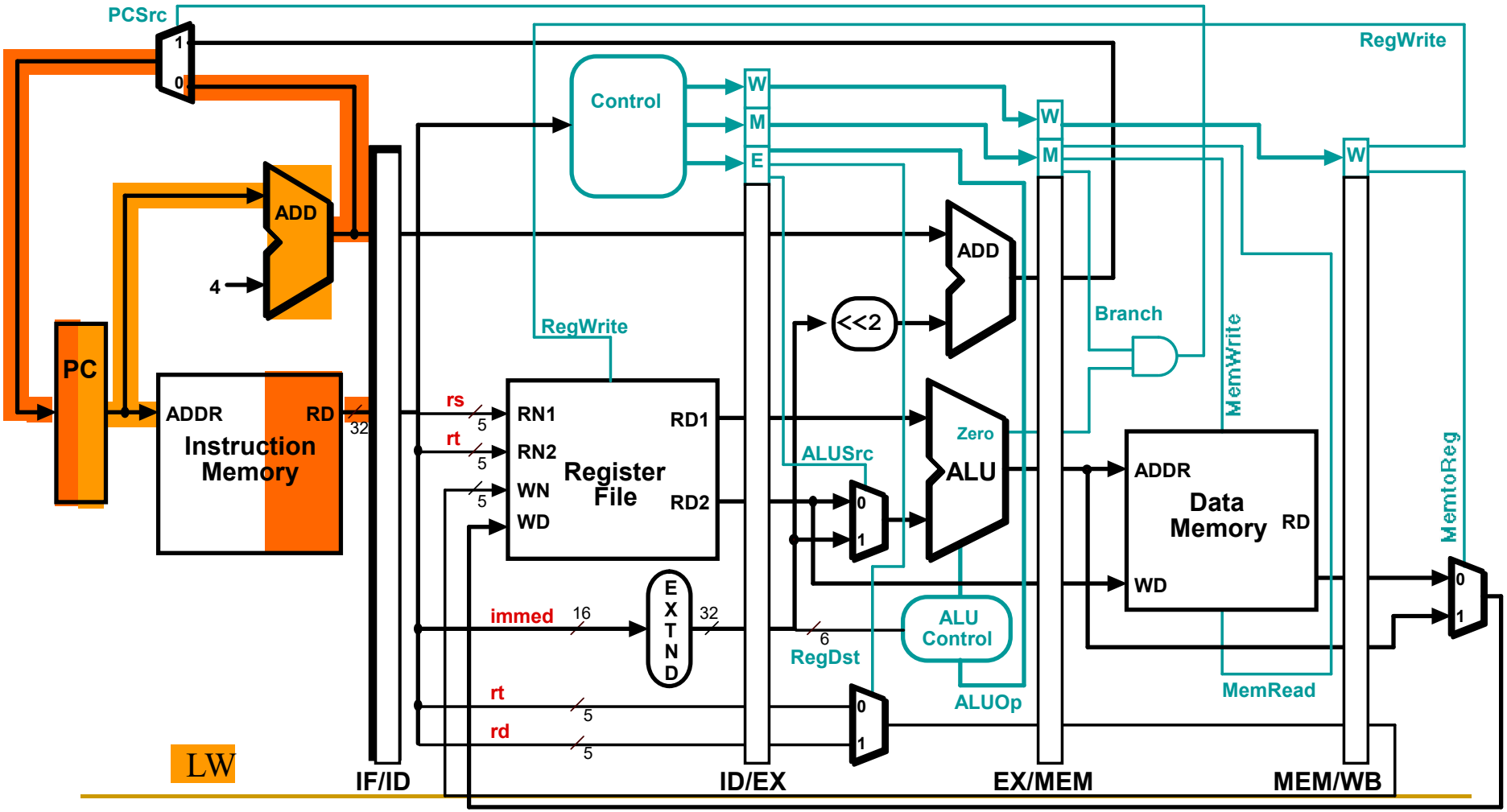
Instruction	Execution/Address Calculation stage control lines				Memory access stage control lines			Write-back stage control lines	
	Reg Dst	ALU Op1	ALU Op0	ALU Src	Branch	Mem Read	Mem Write	Reg write	Mem to Reg
R-format	1	1	0	0	0	0	0	1	0
`lw`	0	0	0	1	0	1	0	1	1
`sw`	X	0	0	1	0	0	1	0	X
`beq`	X	0	1	0	1	0	0	0	X



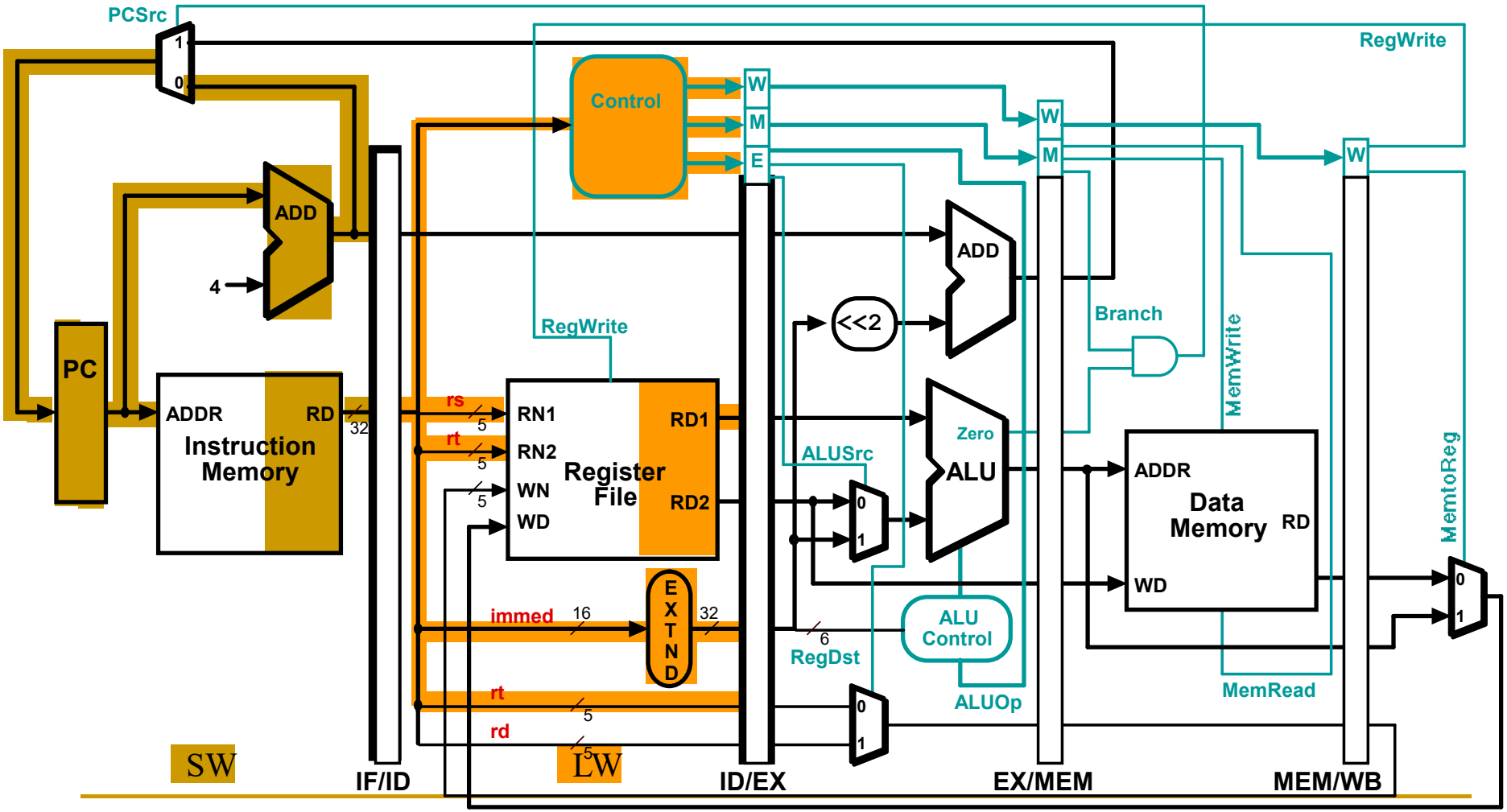
Datapath and Control Unit



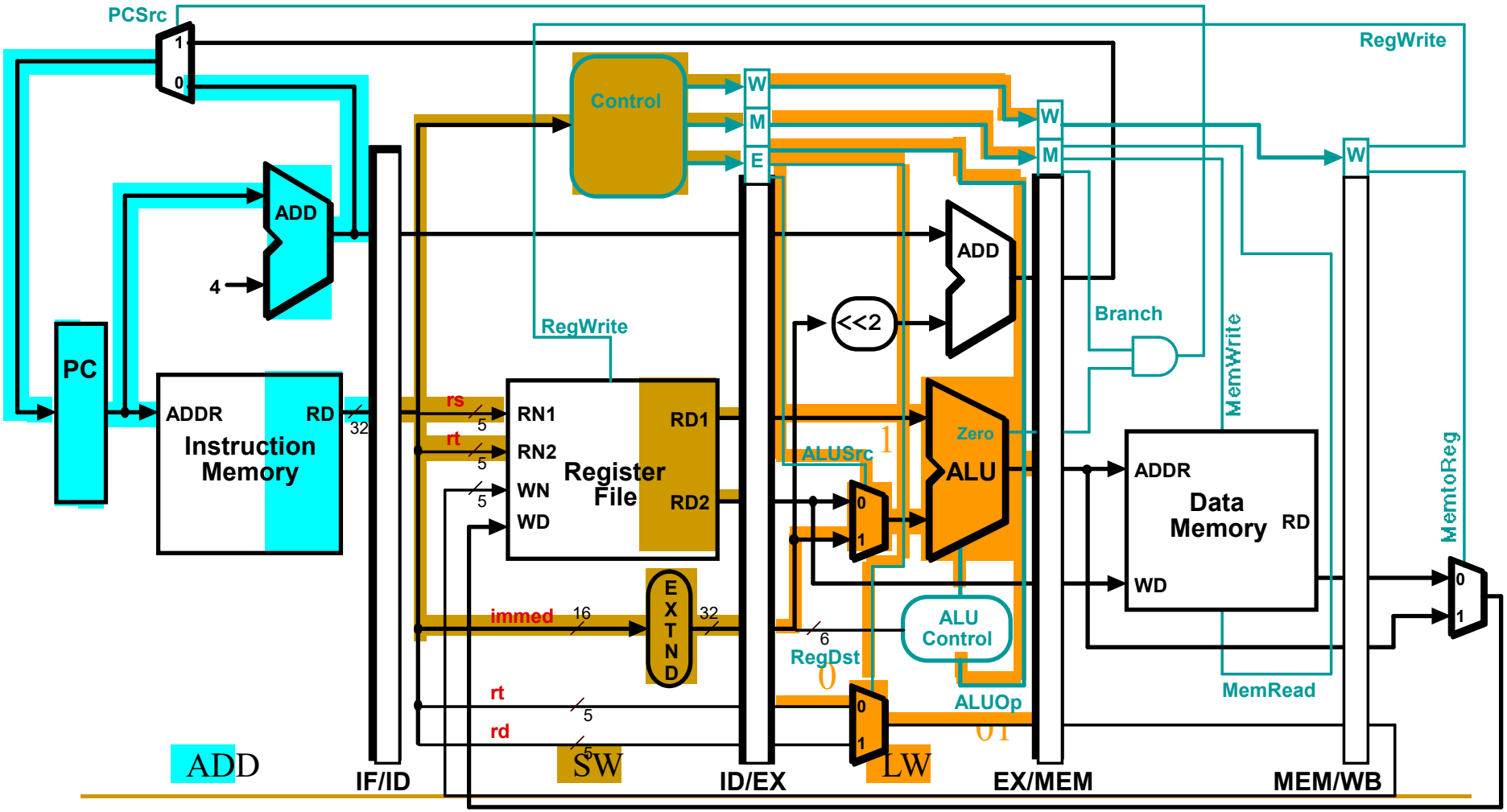
Tracking Control Signals - Cycle 1



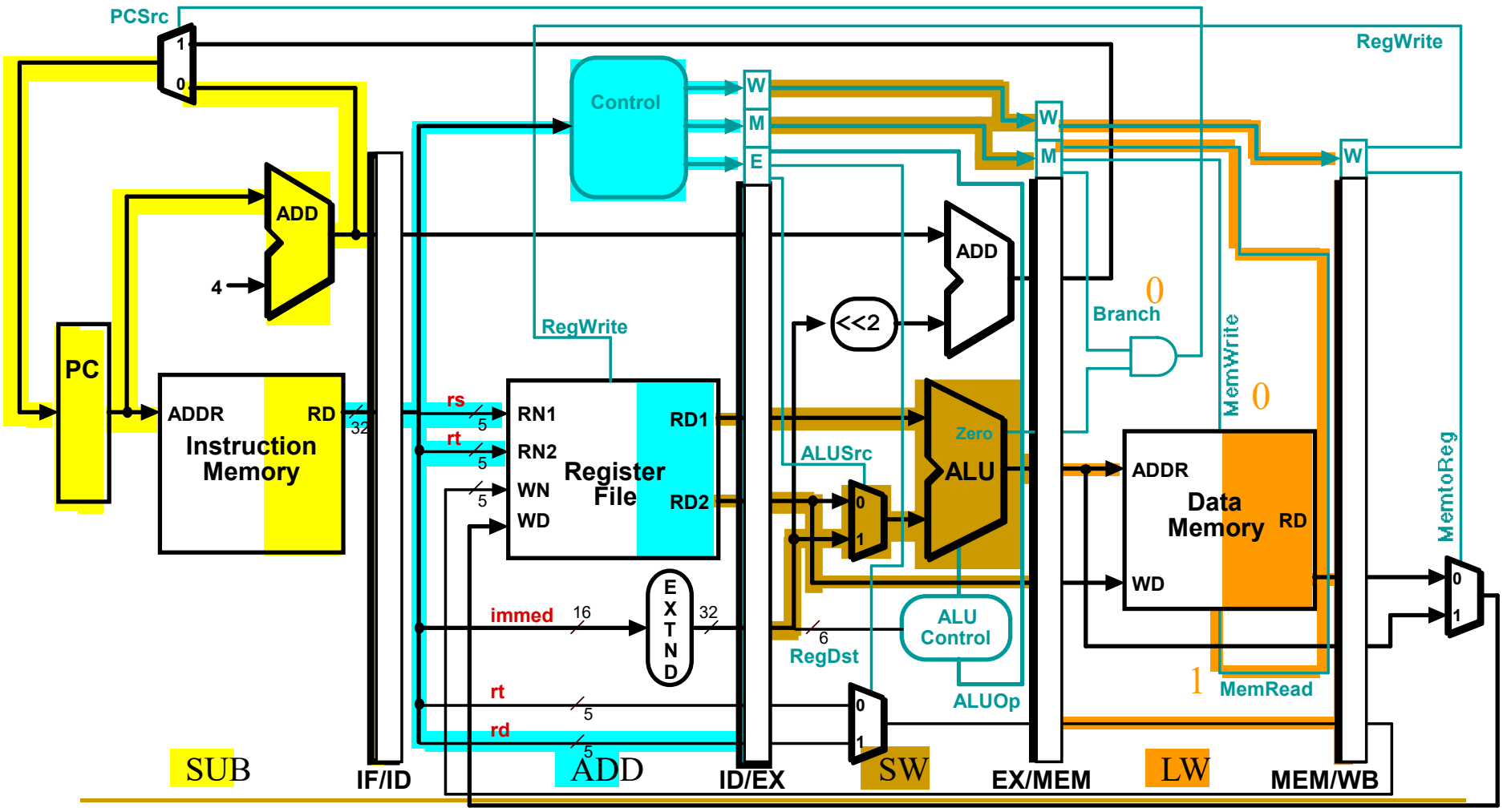
Tracking Control Signals - Cycle 2



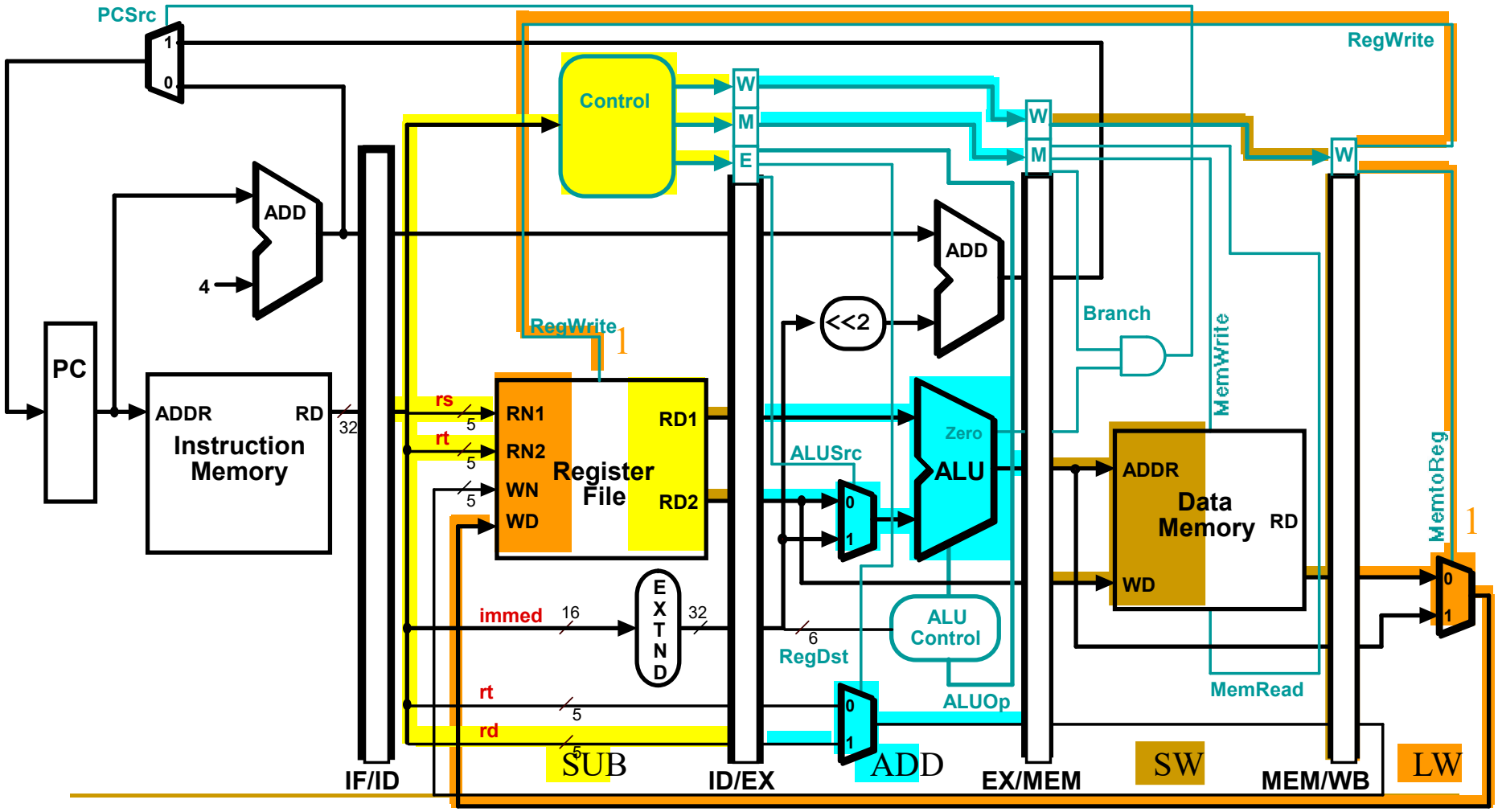
Tracking Control Signals - Cycle 3



Tracking Control Signals - Cycle 4



Tracking Control Signals - Cycle 5



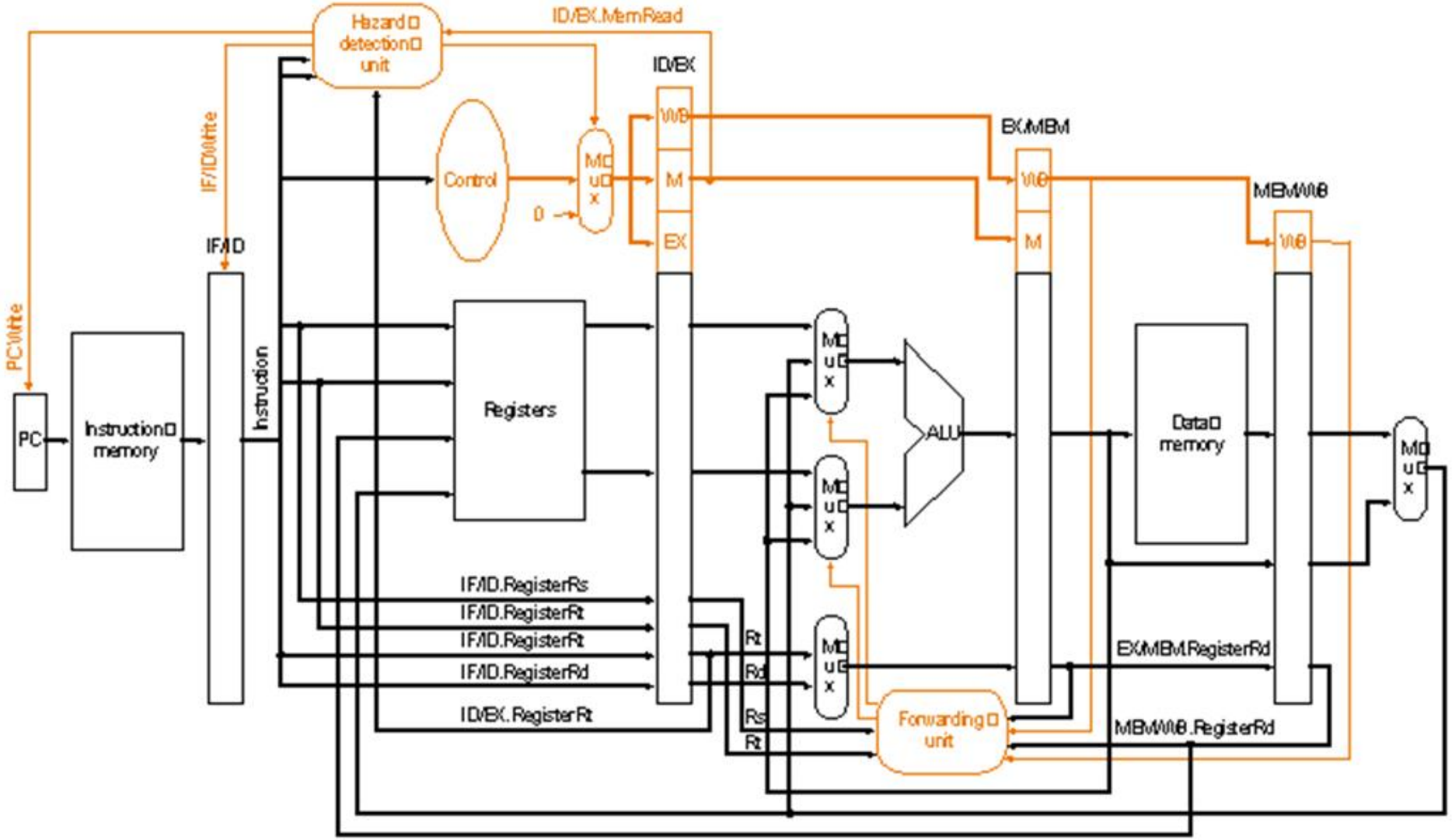
Can pipelining get us into trouble?

- Yes: Pipeline Hazards
- Hazards occur because data required for executing the current instruction may not be available.
  - Structural hazards: attempt to use the same resource two different ways at the same time
  - Control hazards: attempt to make a decision before condition is evaluated
    - branch instructions
    - interrupts

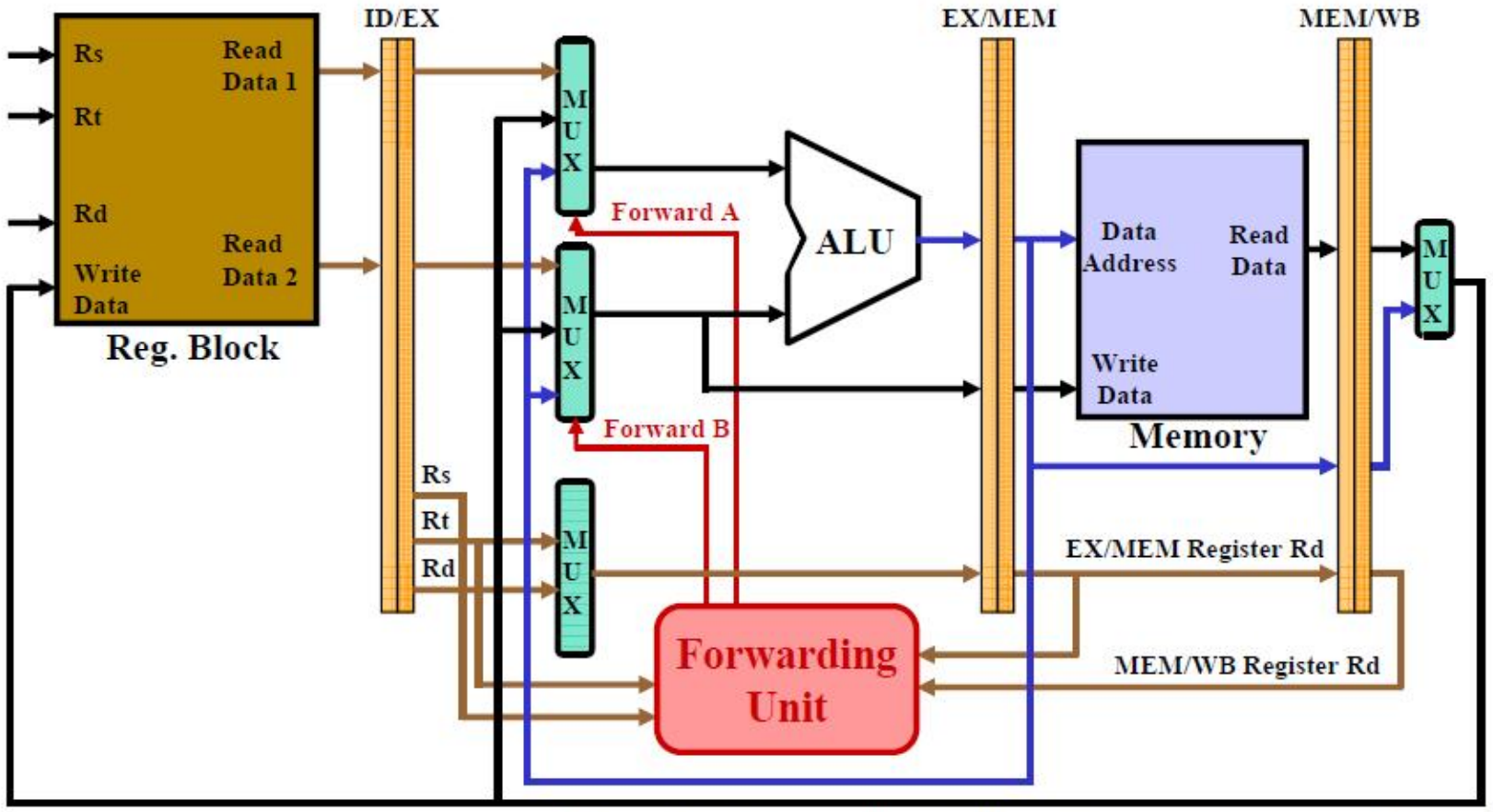
Pipelining troubles (cont.)

- Data hazards occur when an instruction needs register contents for an arithmetic/ logical/memory instruction, before they are ready.
  - instruction depends on result of prior instruction still in the pipeline
- Can always resolve hazards by waiting:
  - pipeline control must detect the hazard
  - take action (or delay action) to resolve hazards

Hazard detection & Forwarding Units



Forwarding: A solution to Data Hazards



After David A. Patterson and John L. Hennessy, *Computer Organization and Design*, 2nd Edition

Stalls

- Forwarding will not always solve the problems of data hazards.
- For example, suppose an add instruction follows a load word (lw), and the add involves the register that receives the memory data.
- In this case, forwarding will not work.
- The reason is that the data must be read from memory, and so it will not be available until the end of the MEM cycle. Thus the required data is not available for a forward, and the add instruction. if it proceeds, will process the wrong data.
- A solution to this problem is the stall.
- A stall halts the instruction awaiting data, while the key instruction (a lw in this case) proceeds to the end of the MEM cycle, after which the desired data is available to the add.

Other Problems With Branches

- A remaining problem is what to do about instructions following a branch.
- Even assuming forwarding and stalls, the branch/no branch decision is not made until the third stage.
- This means that in the MIPS pipeline, two following instructions will enter the pipe before the branch/no branch decision is made.
- What if:
  - The following instructions were for the case of “branch taken” and the branch was not taken.
  - The following instructions were for “branch not taken” and it was taken.
- In either case, the wrong instructions are in the pipe and they must be eliminated (“flushed”).
- How can this problem be prevented?

Control Hazard Approach

- One approach is to always assume the branch is(or is not) taken:
- Say we assume the branch is never taken. Then if the instruction in ALU/EX is a branch, the instructions in IF and ID/RF will be those in the “not taken” program line (branch determination is made in ALU/EX).
  - It this assumption is correct, the pipeline will continue to flow without delay.
  - When the branch is taken, instructions in IF and ID/RF must be “flushed,” usually by changing the “op” code of those instructions to a “nop” and letting them proceed to the end of the pipe.

Summary

- Pipelining is a fundamental concept in computers/nature
  - Multiple instructions in flight
  - Limited by length of longest stage
  - Latency vs.Throughput
- Hazards gum up the works