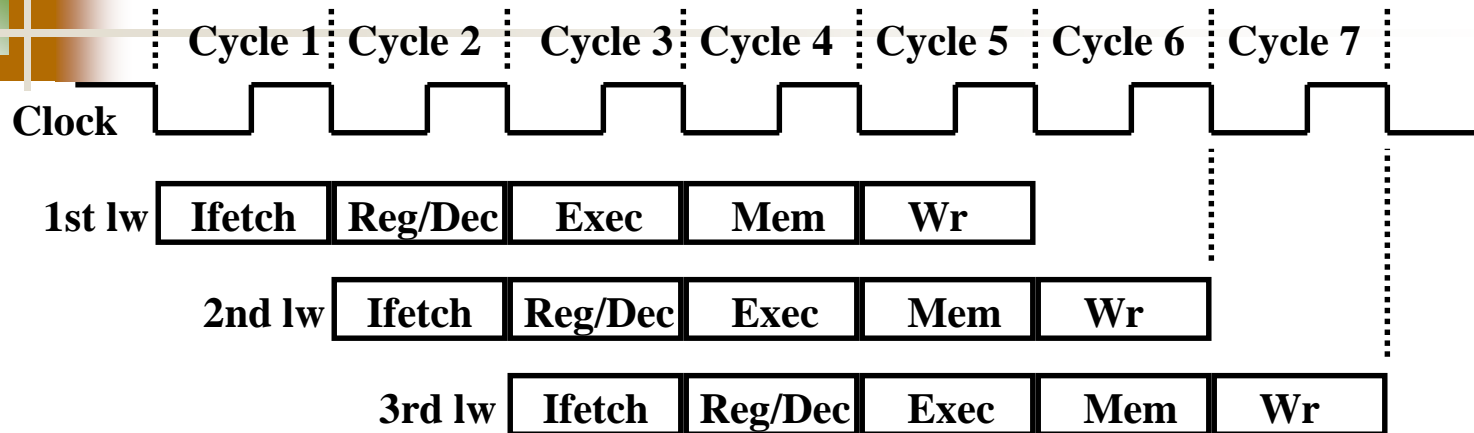


CSE331 – Computer Organization

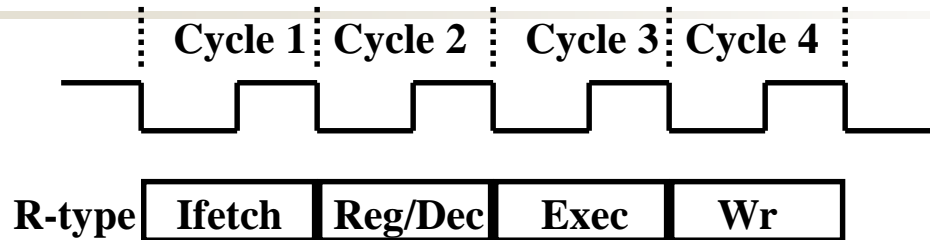
Lecture 10: A Pipelined Datapath Design

Pipelining the Load Instruction



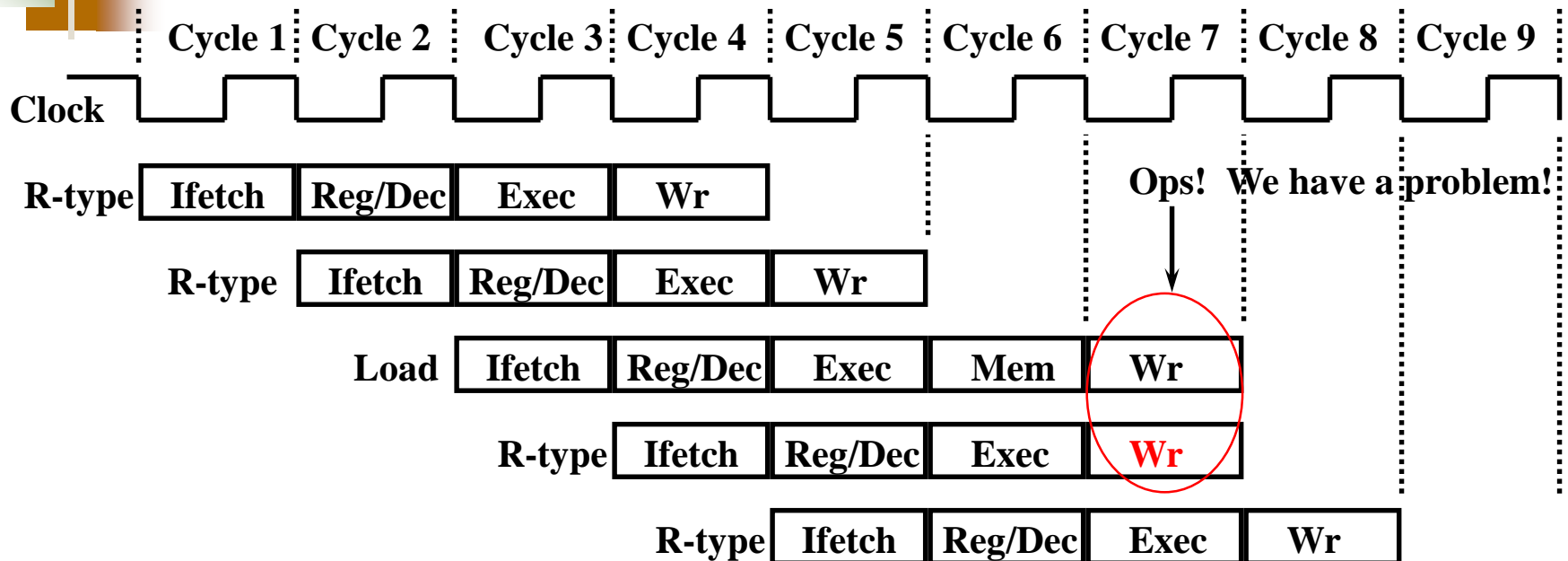
- ▶ The five independent functional units in the pipeline datapath are:
 - Instruction Memory for the **Ifetch** stage
 - Register File's Read ports (bus A and busB) for the **Reg/Dec** stage
 - ALU for the **Exec** stage
 - Data Memory for the **Mem** stage
 - Register File's **Write** port (bus W) for the **Wr** stage

The Four Stages of R-type



- ▶ **Ifetch: Instruction Fetch**
 - Fetch the instruction from the Instruction Memory
 - Update PC
- ▶ **Reg/Dec: Registers Fetch and Instruction Decode**
- ▶ **Exec:**
 - ALU operates on the two register operands
- ▶ **Wr: Write the ALU output back to the register file**

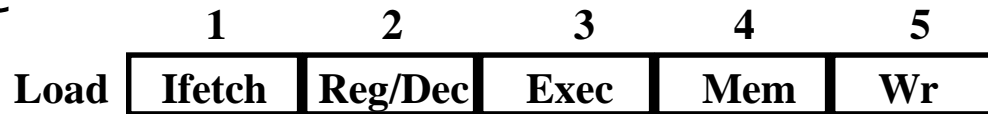
Pipelining the R-type and Load Instruction



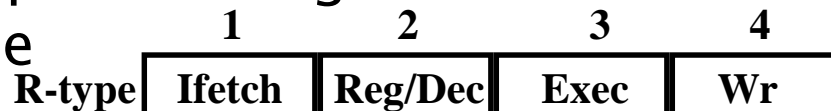
- ▶ We have pipeline conflict or structural hazard:
 - Two instructions try to write to the register file at the same time!
 - Only one write port

Important Observation

- ▶ Each functional unit can only be used **once** per instruction
- ▶ Each functional unit must be used at the **same** stage for all instructions:
 - Load uses Register File's Write Port during its **5th** stage

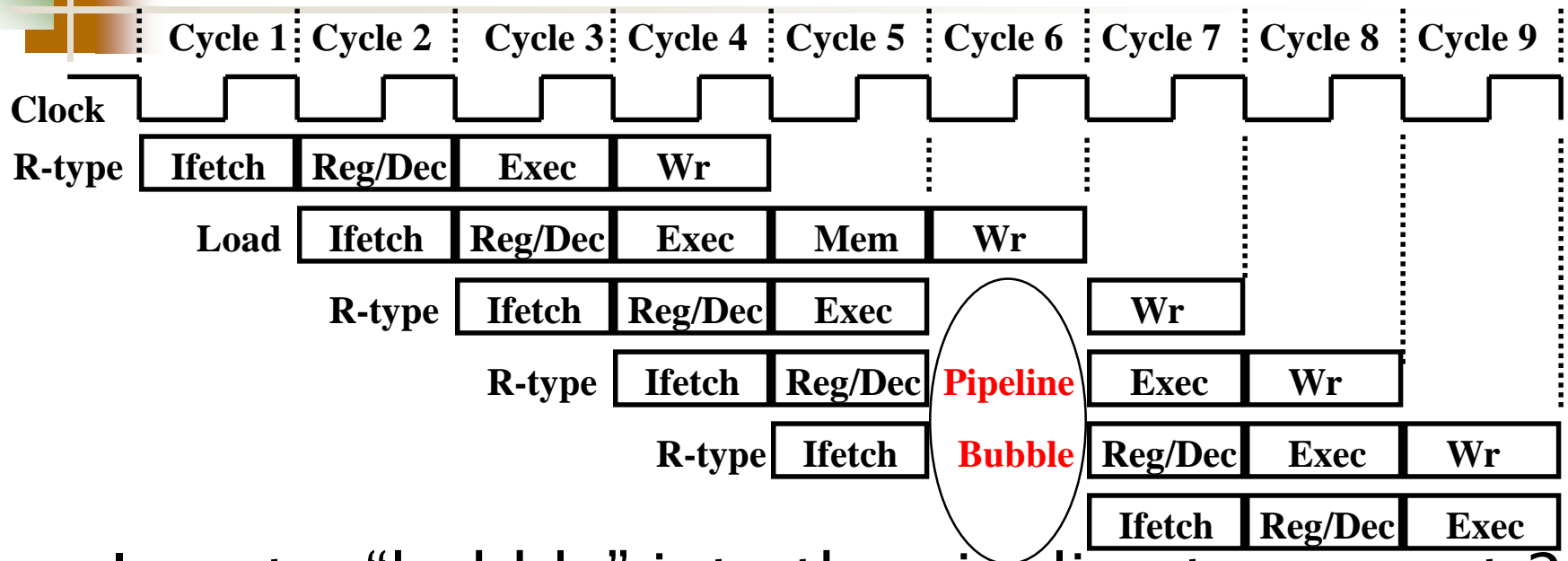


- R-type uses Register File's Write Port during its **4th** stage



- 2 ways to solve this pipeline hazard.

Solution 1: Insert "Bubble" into the Pipeline

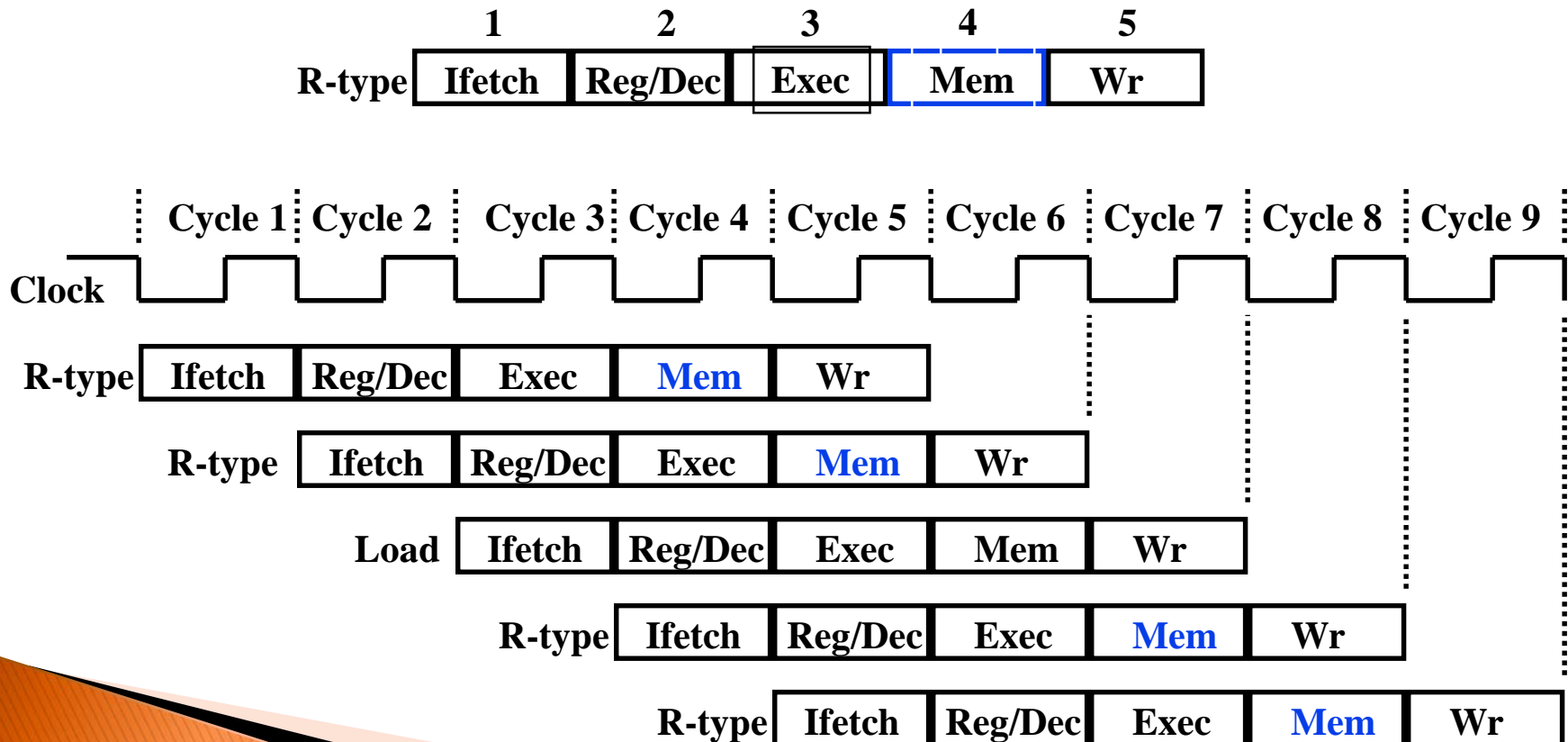


- ▶ Insert a "bubble" into the pipeline to prevent 2 writes at the same cycle
 - The control logic can be complex.
 - Lose instruction fetch and issue opportunity.
- ▶ No instruction is started in Cycle 6!

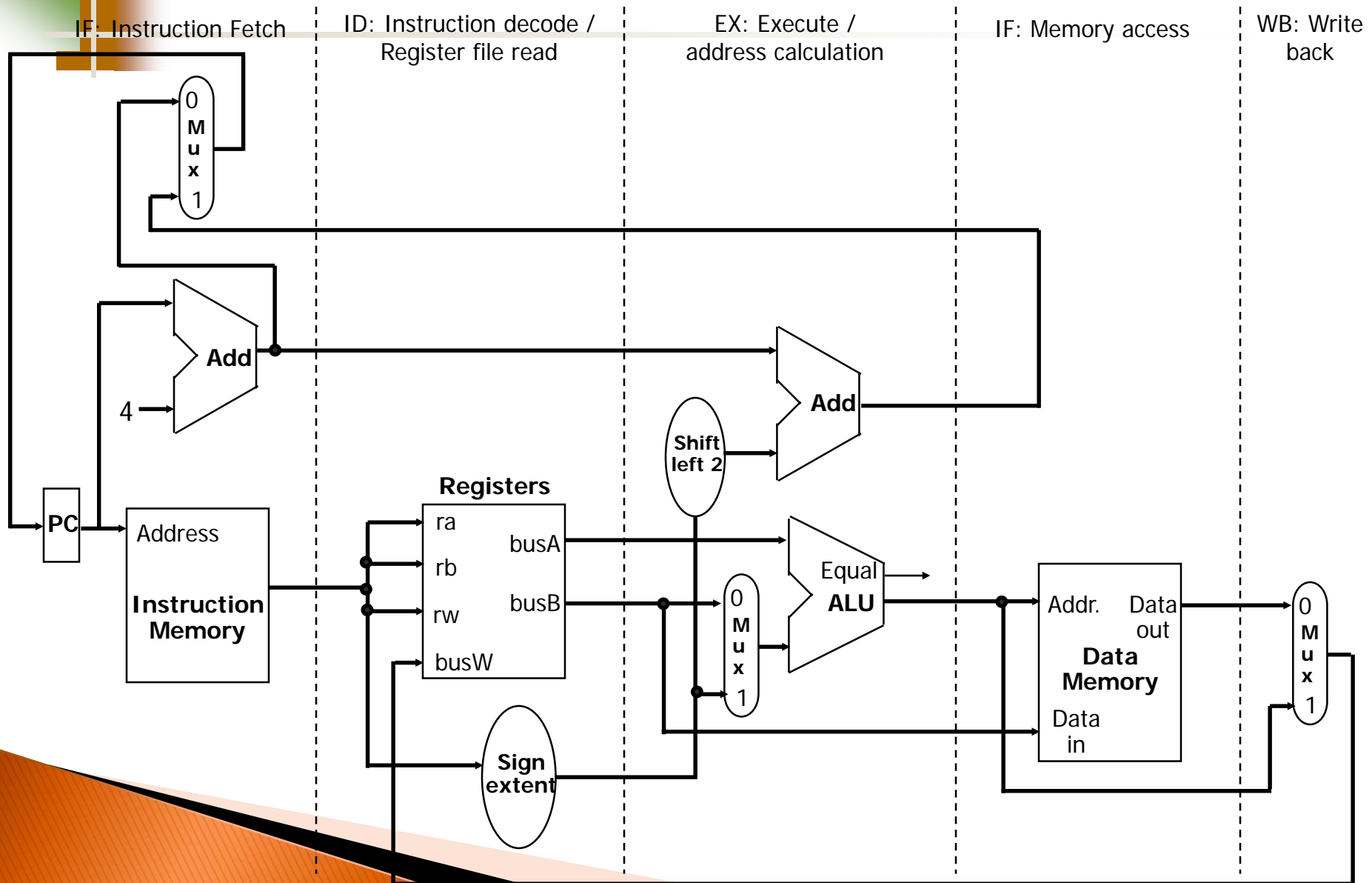
Solution 2: Delay R-type's Write by One Cycle

Delay R-type's register write by one cycle:

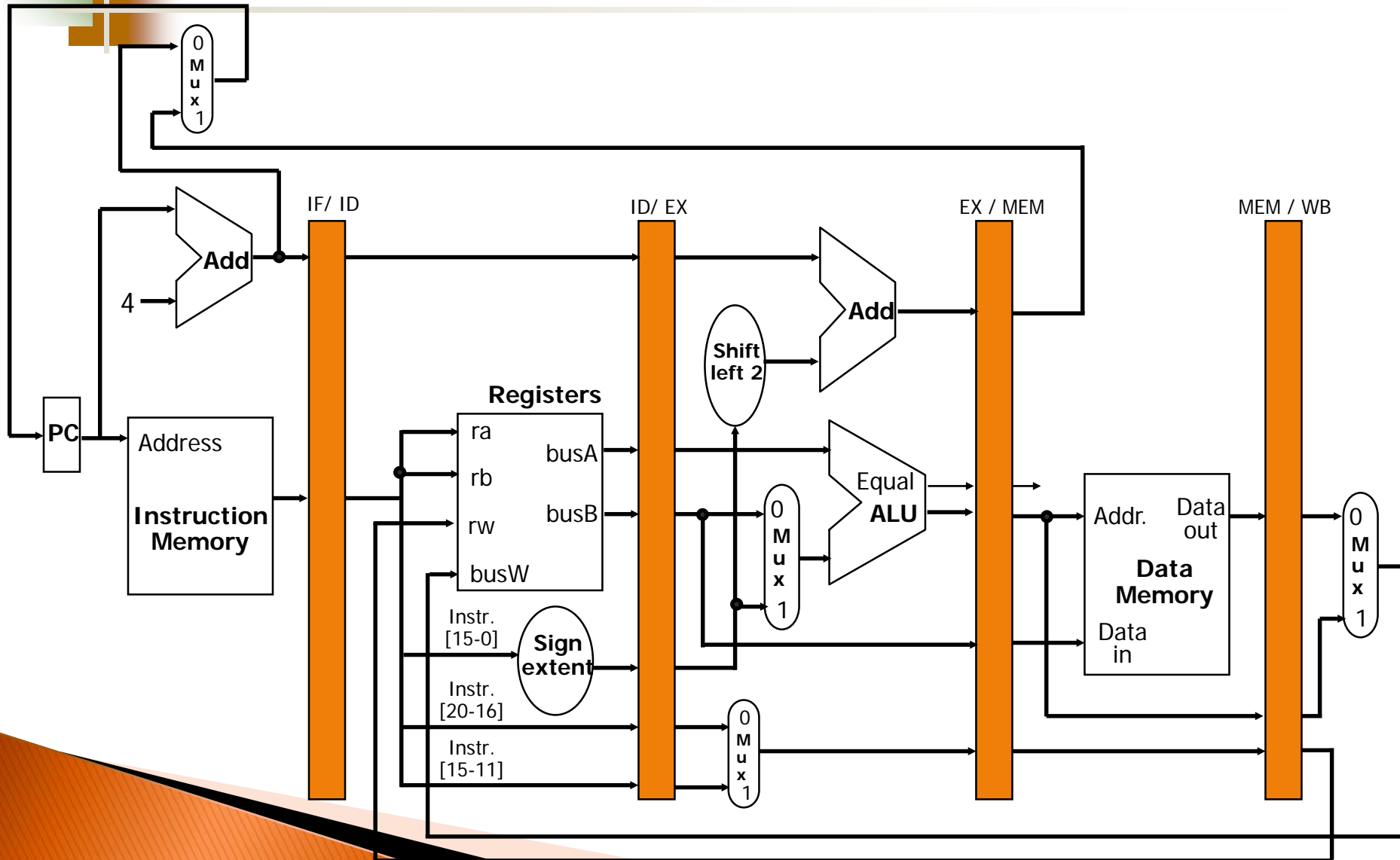
- Now R-type instructions also use Reg File's write port at Stage 5
- Mem stage is a **NOOP** stage: nothing is being done.

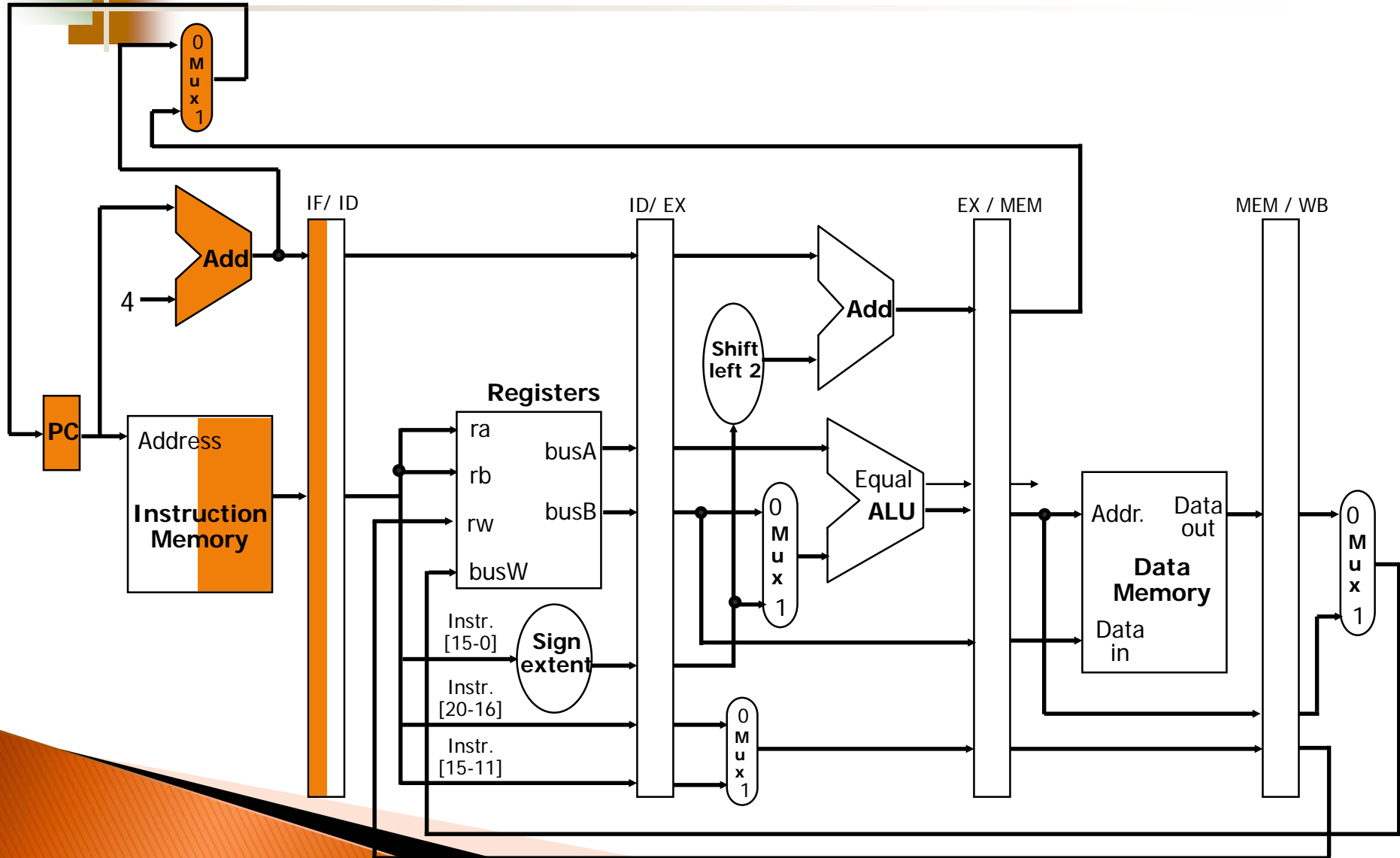


Single Cycle Datapath

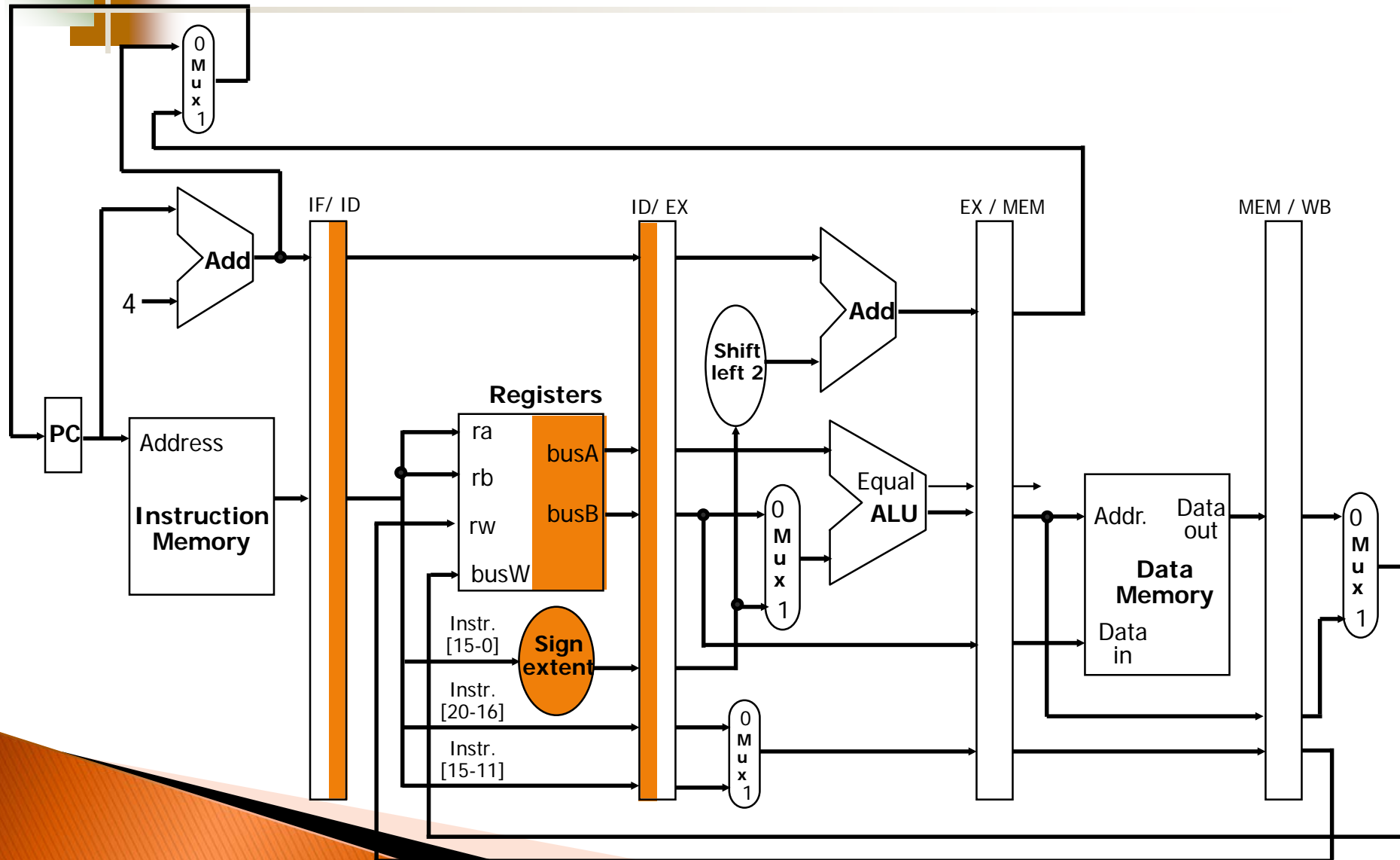


Pipelined Version of the Datapath

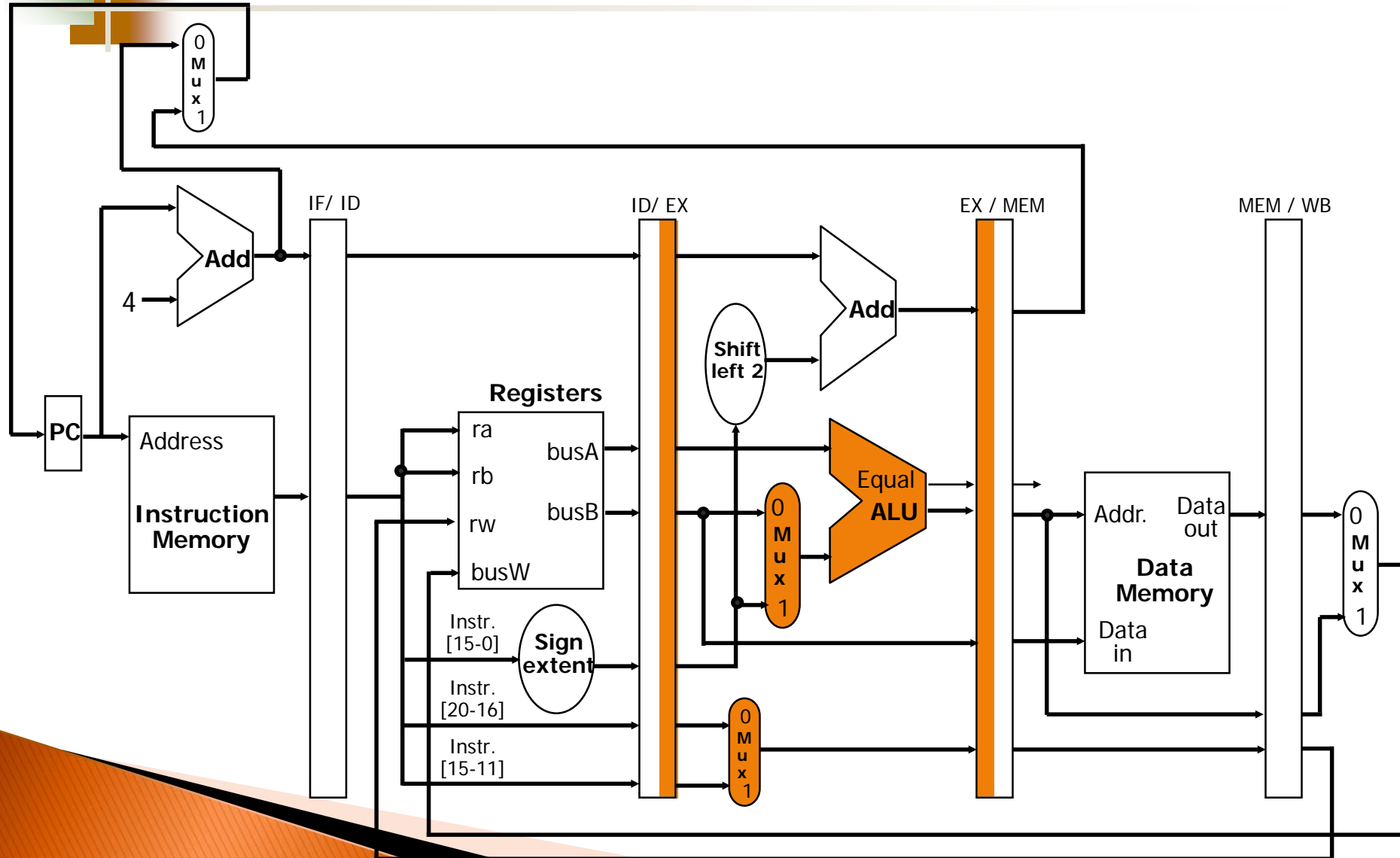




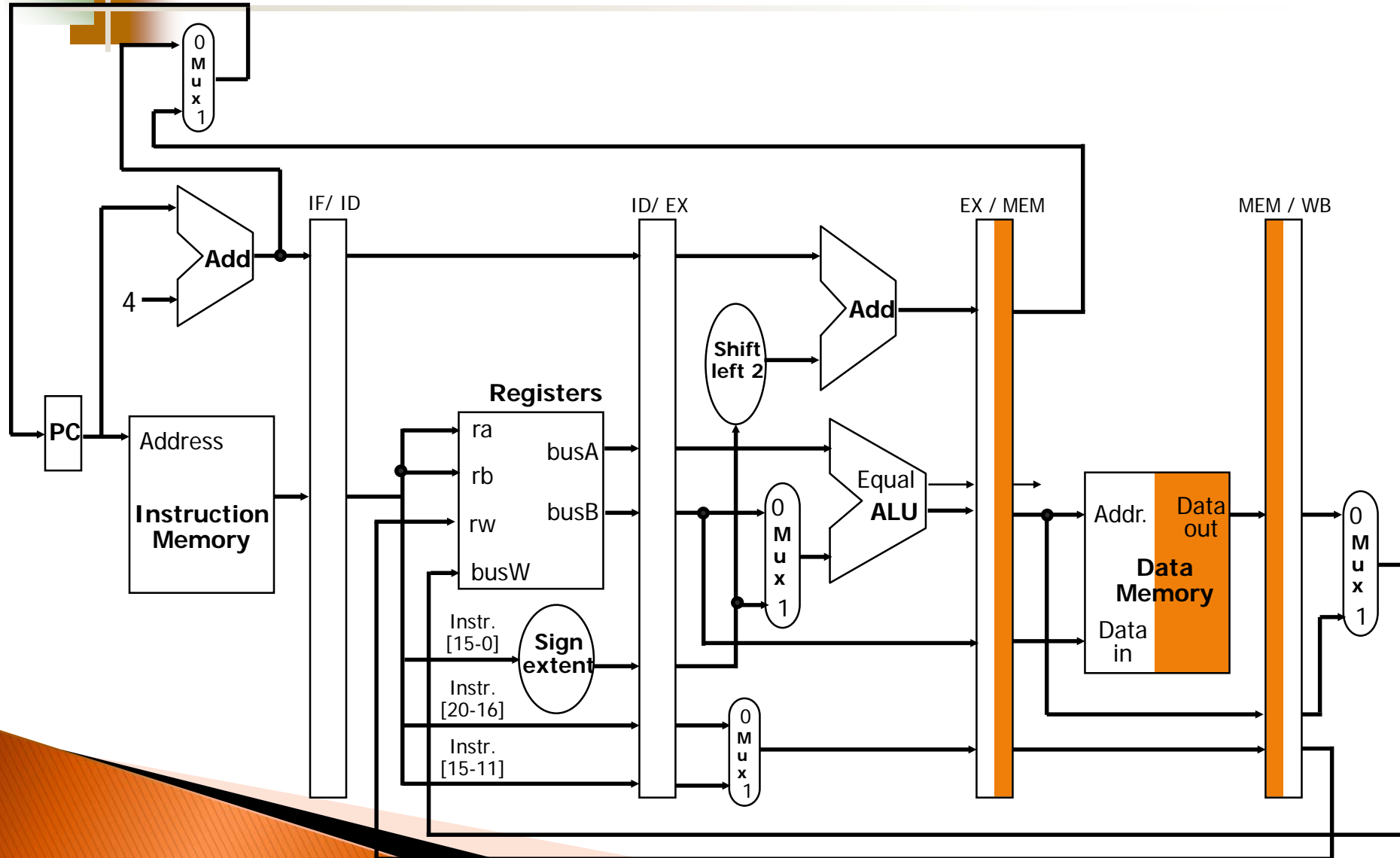
ID: The second pipe stage of a load instruction



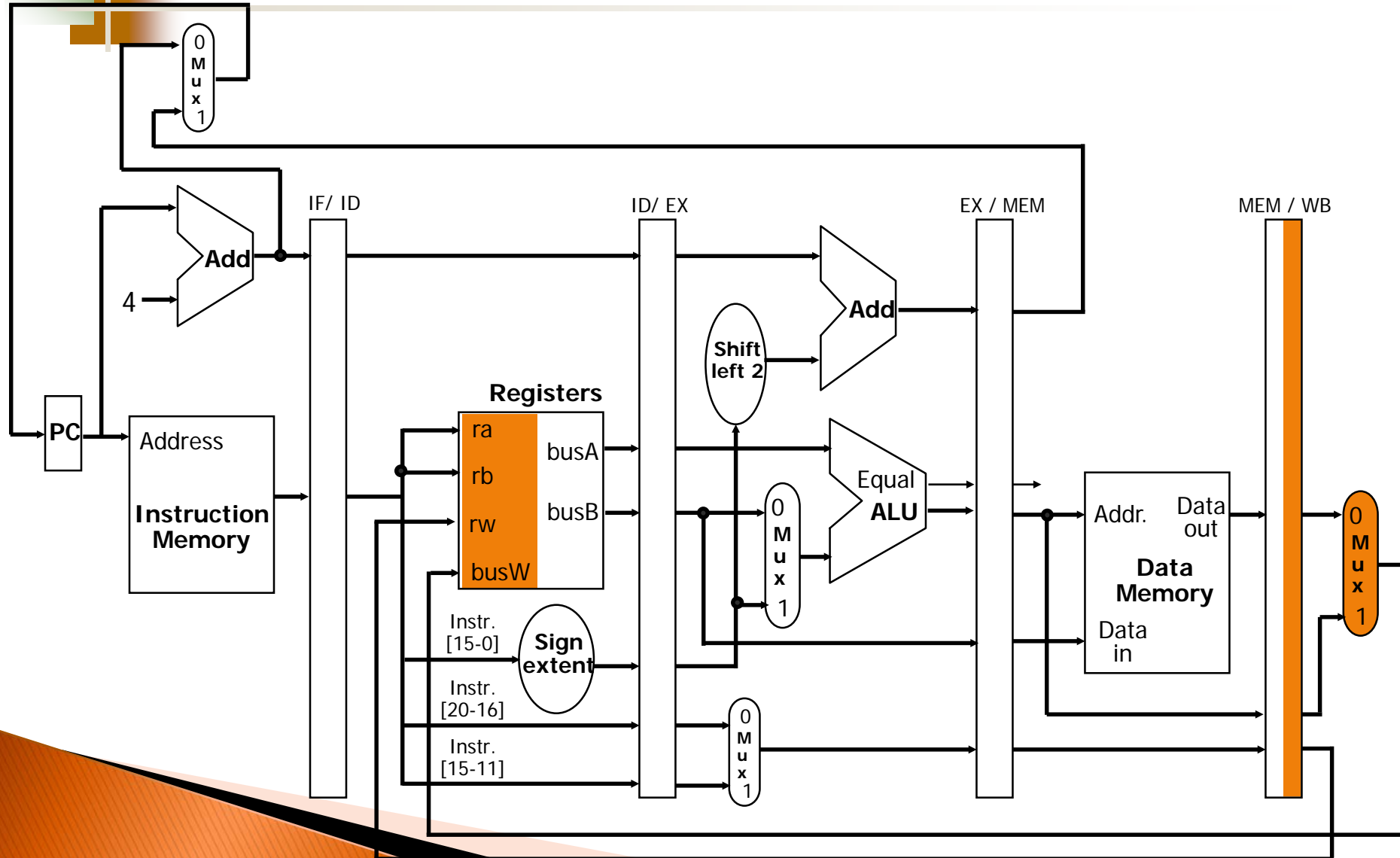
EX: The third pipe stage of a load instruction



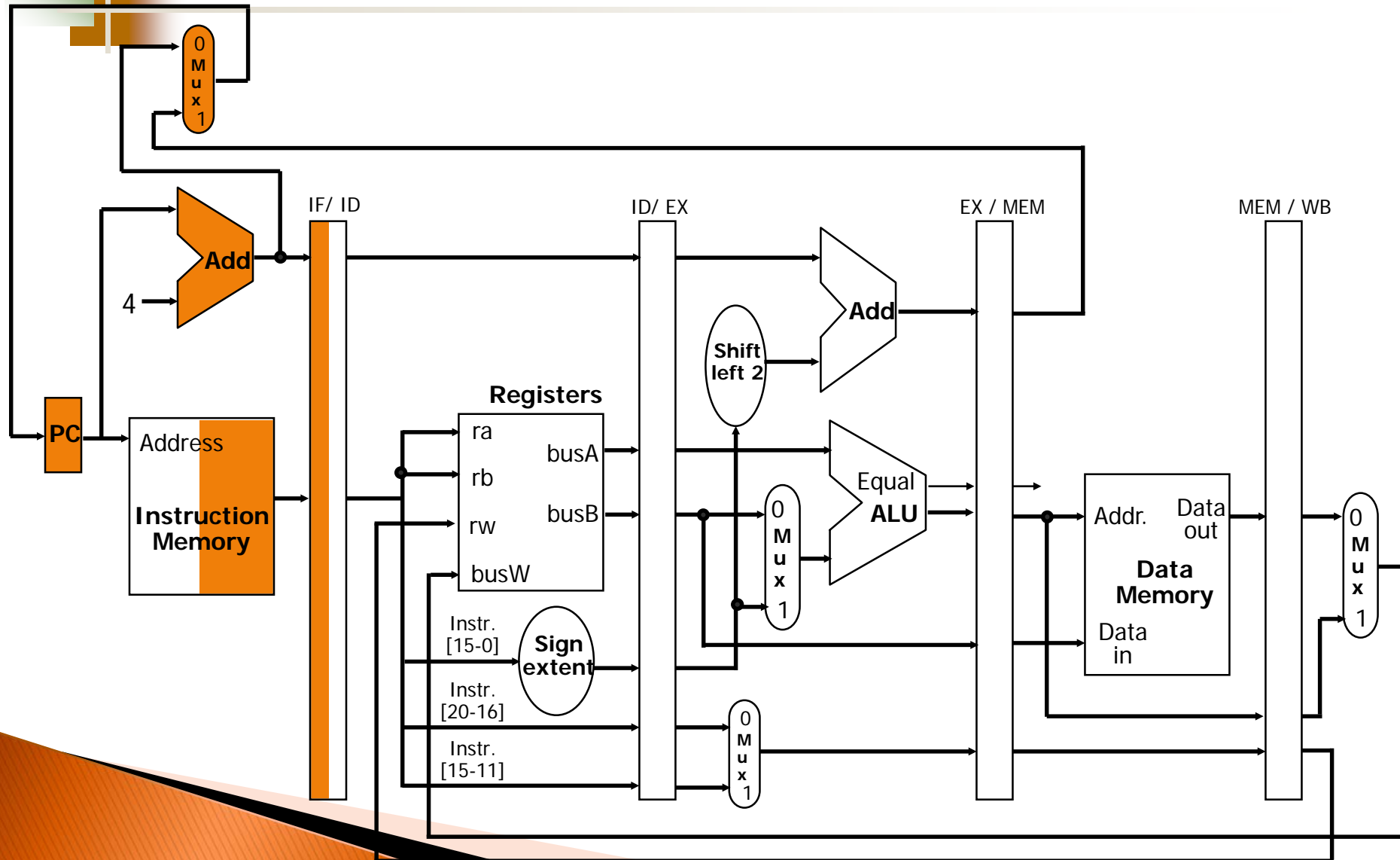
MEM: The fourth pipe stage of a load instruction



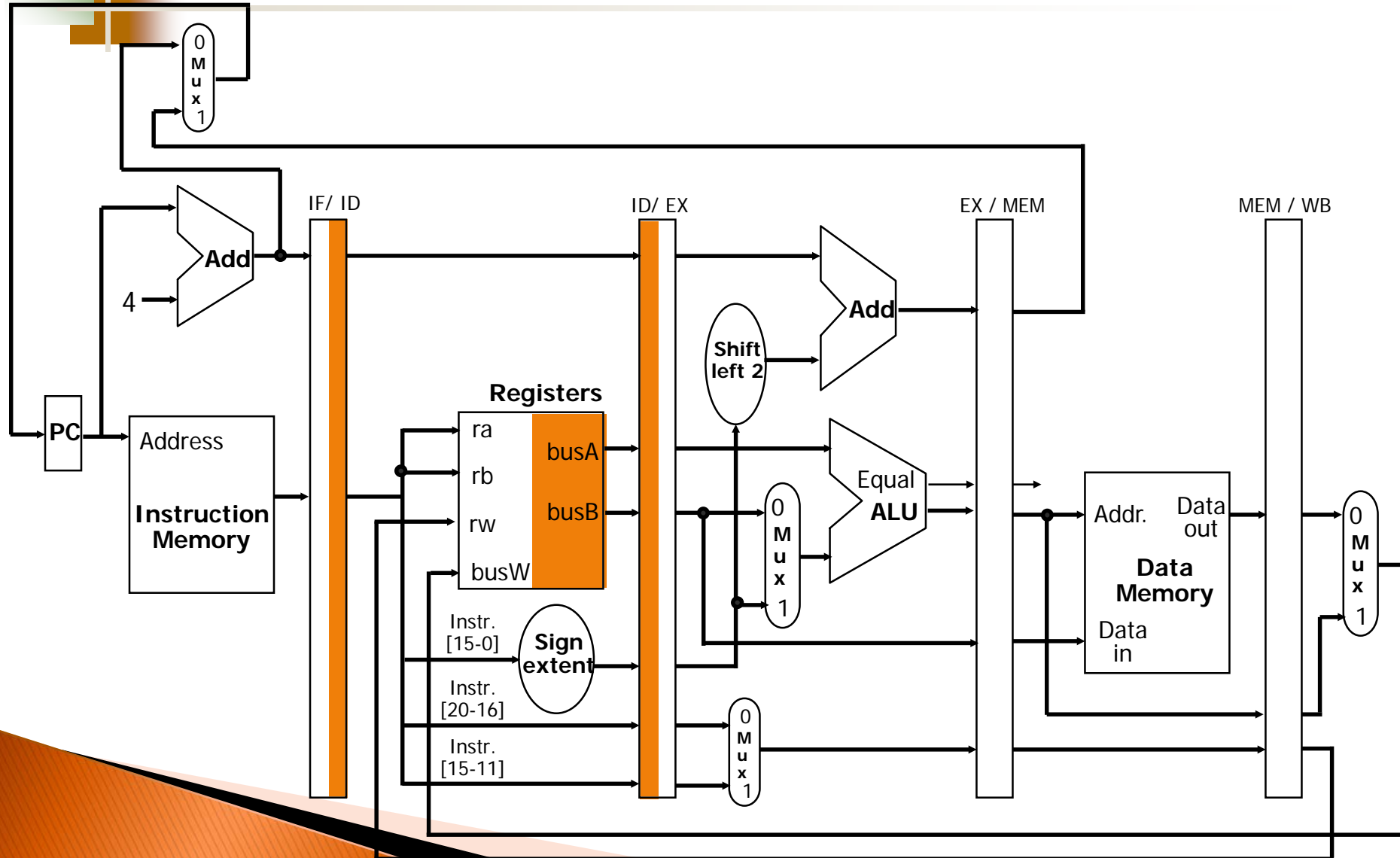
WB: The fifth pipe stage of a load instruction



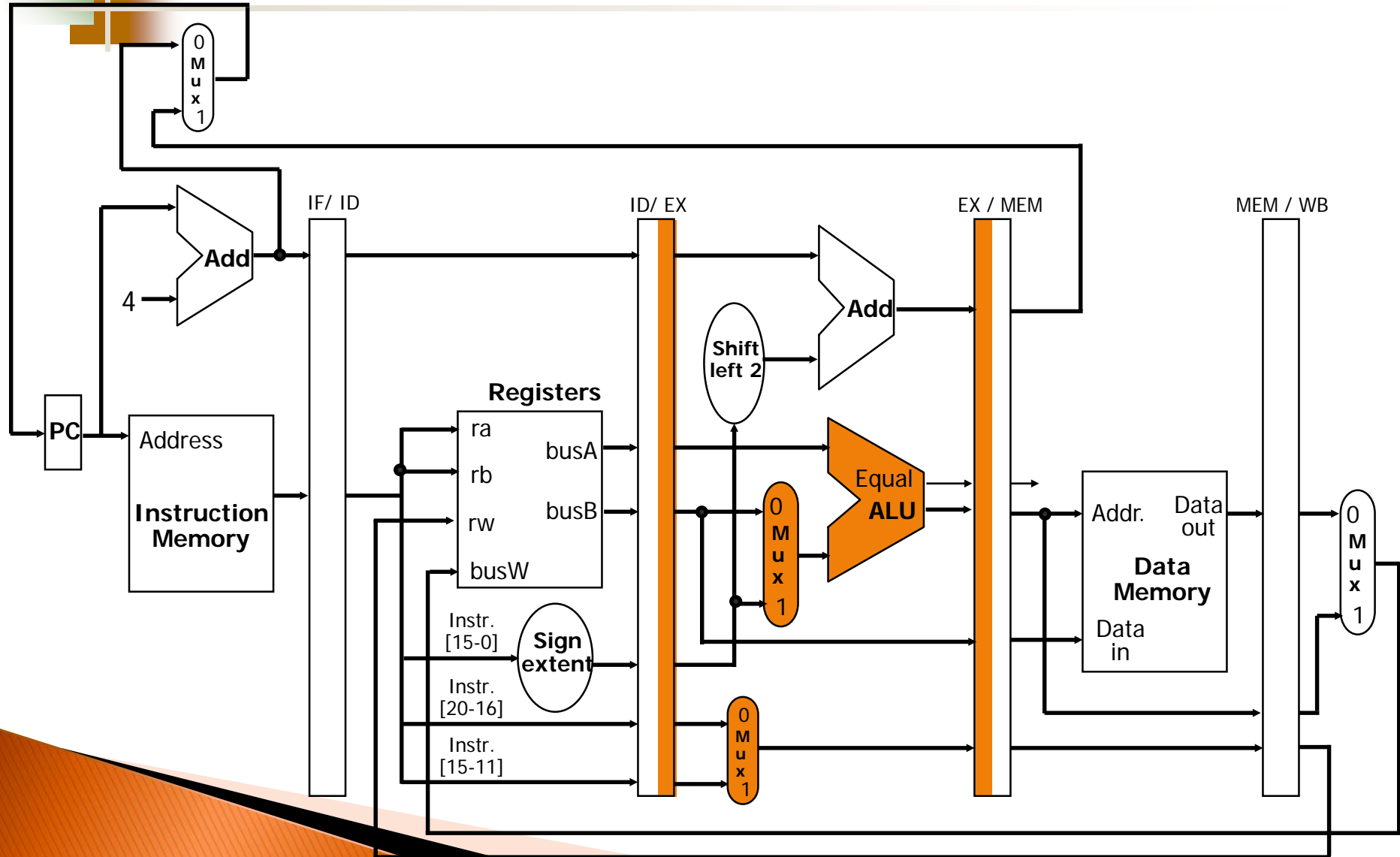
IF: The first pipe stage of a R-type instruction



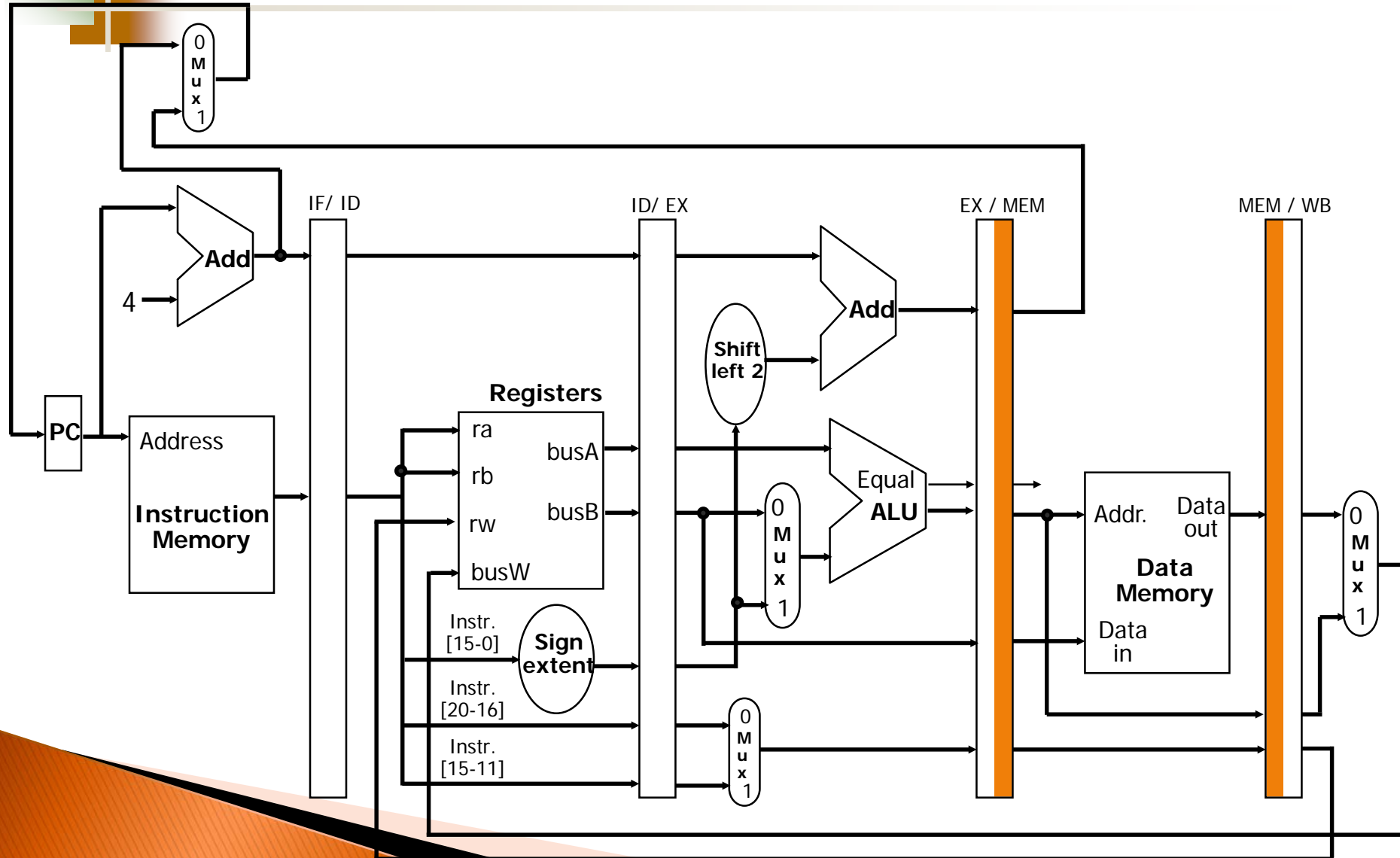
ID: The second pipe stage of a R-type instruction



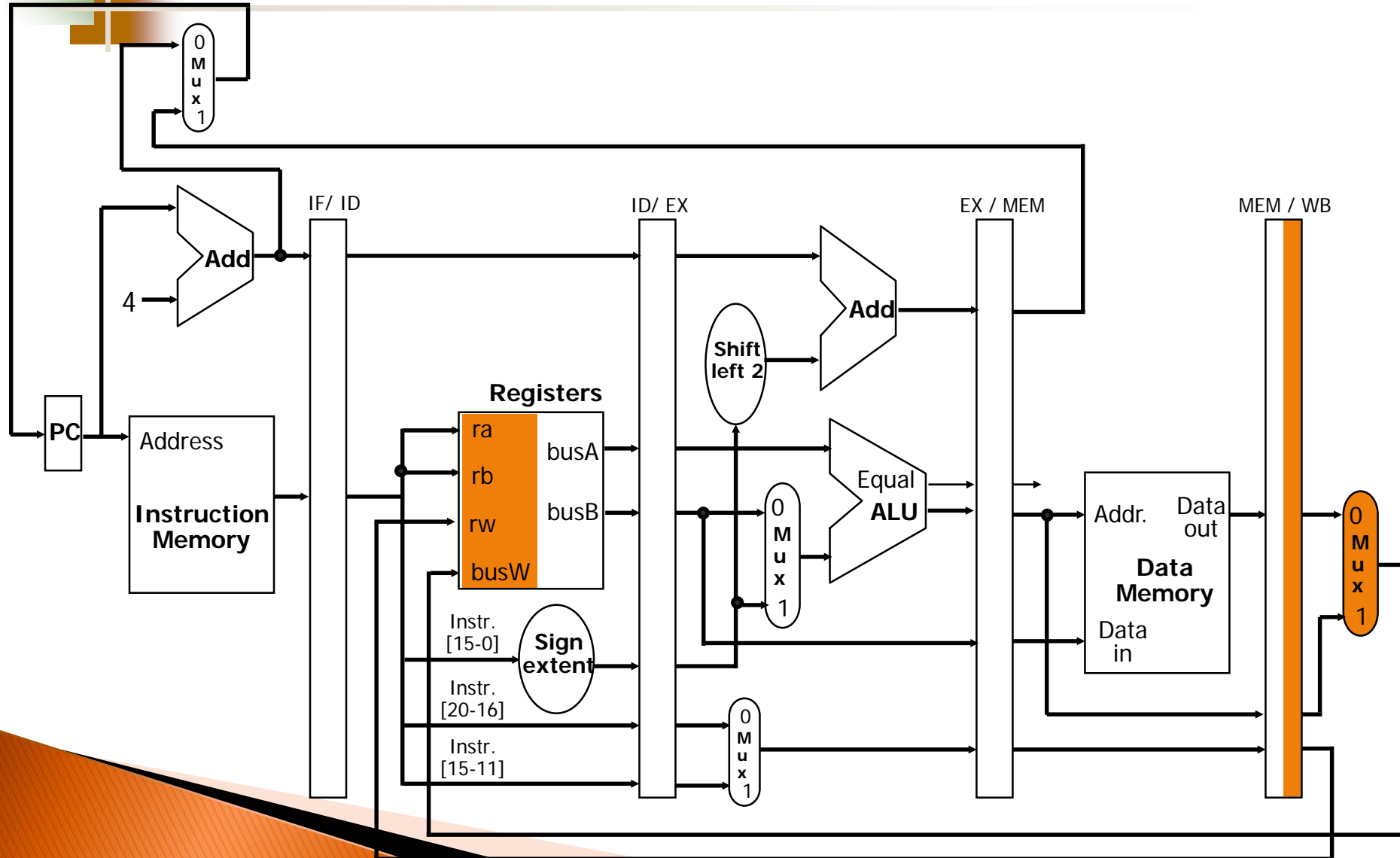
EX: The third pipe stage of a R-type instruction



MEM: The fourth pipe stage of a R-type instruction

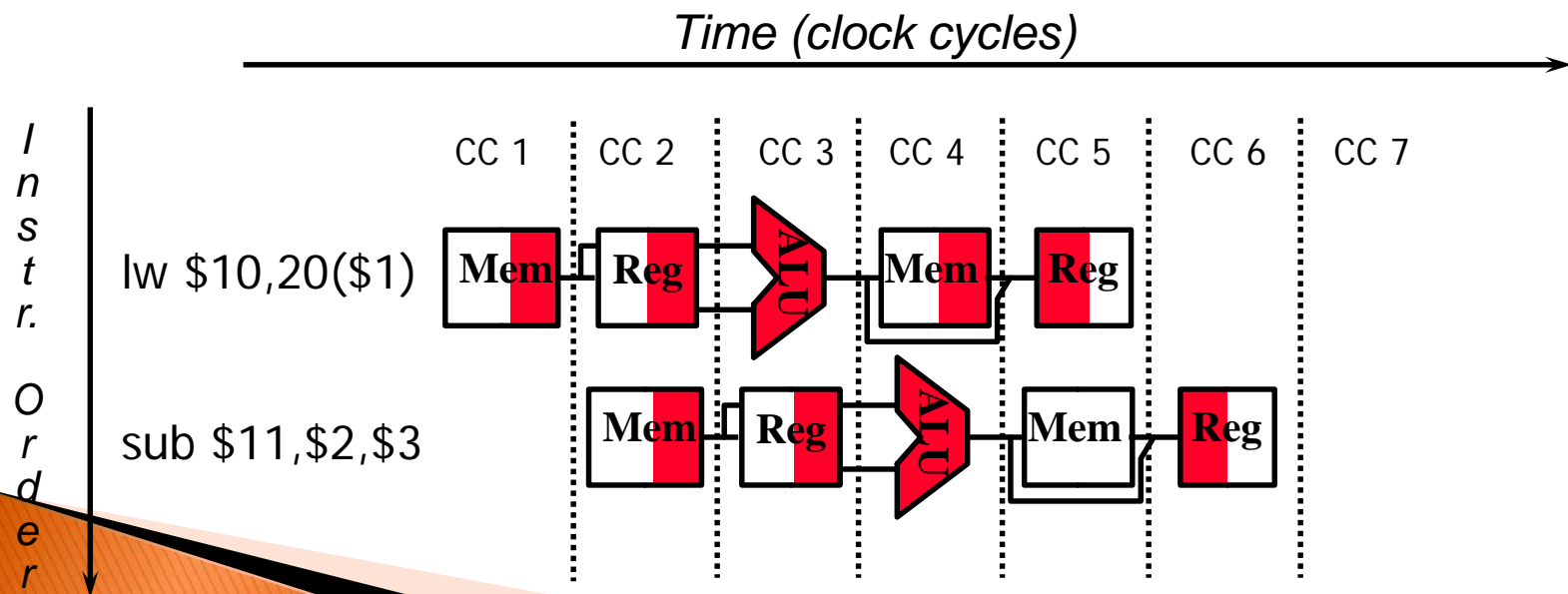


WB: The fifth pipe stage of a R-type instruction

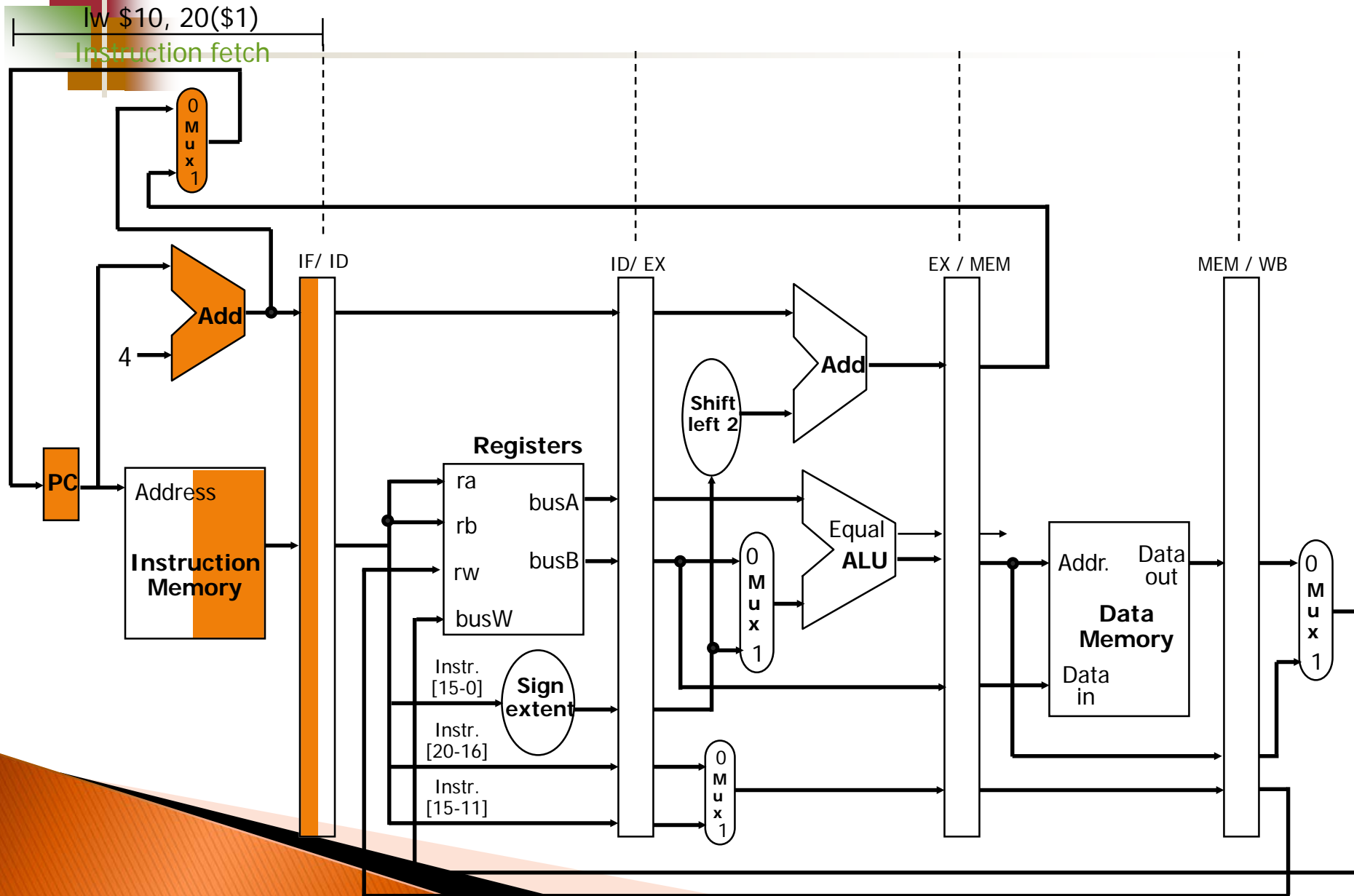


An Example to Clarify Pipelining

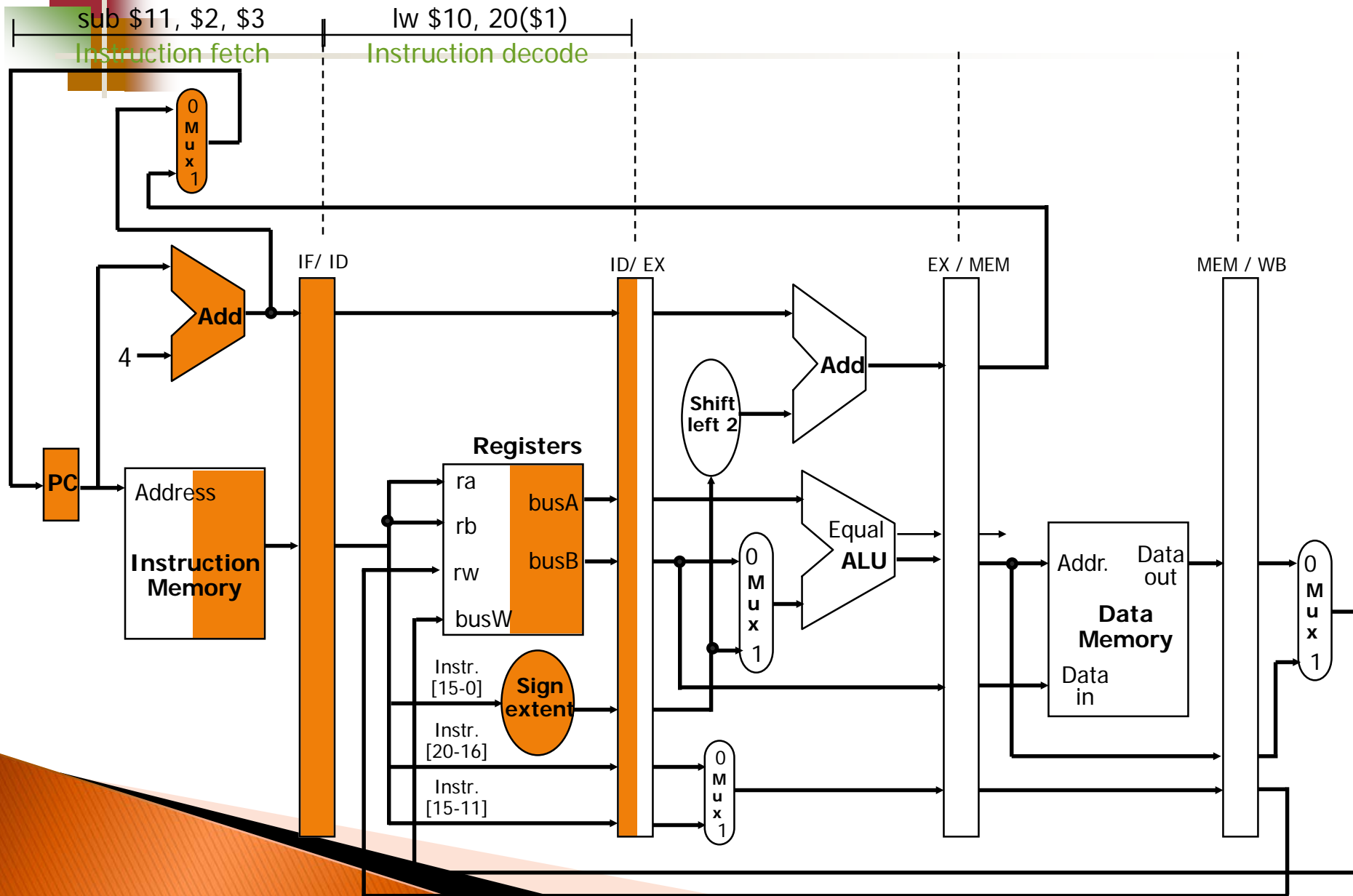
- ▶ Since many instructions are simultaneously executing in a single cycle datapath, it can be difficult to understand.
- ▶ The following code will be examined:
 lw \$10, 20(\$1)
 sub \$11, \$2, \$3



Clock 1



Clock 2



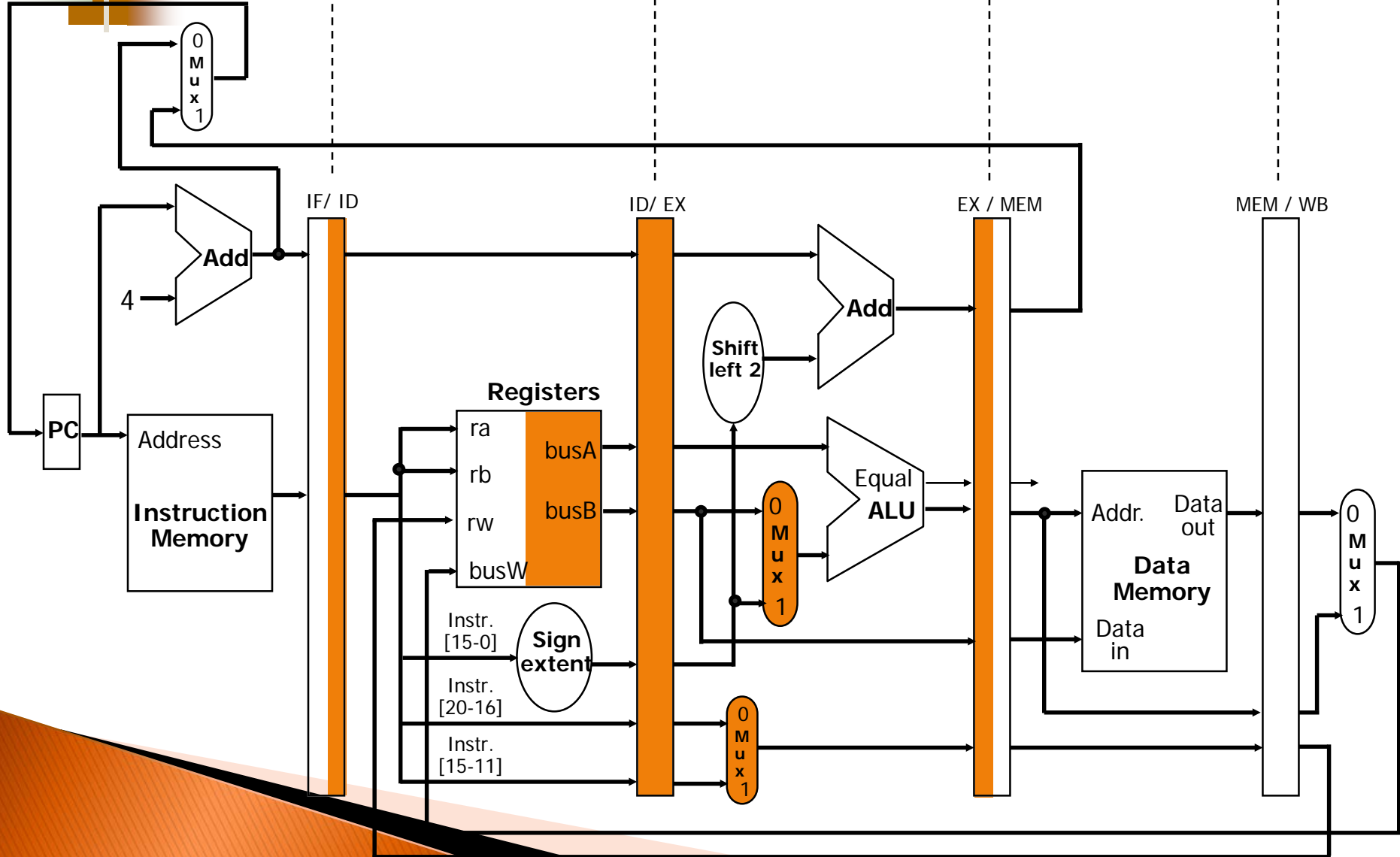
Clock 3

sub \$11, \$2, \$3

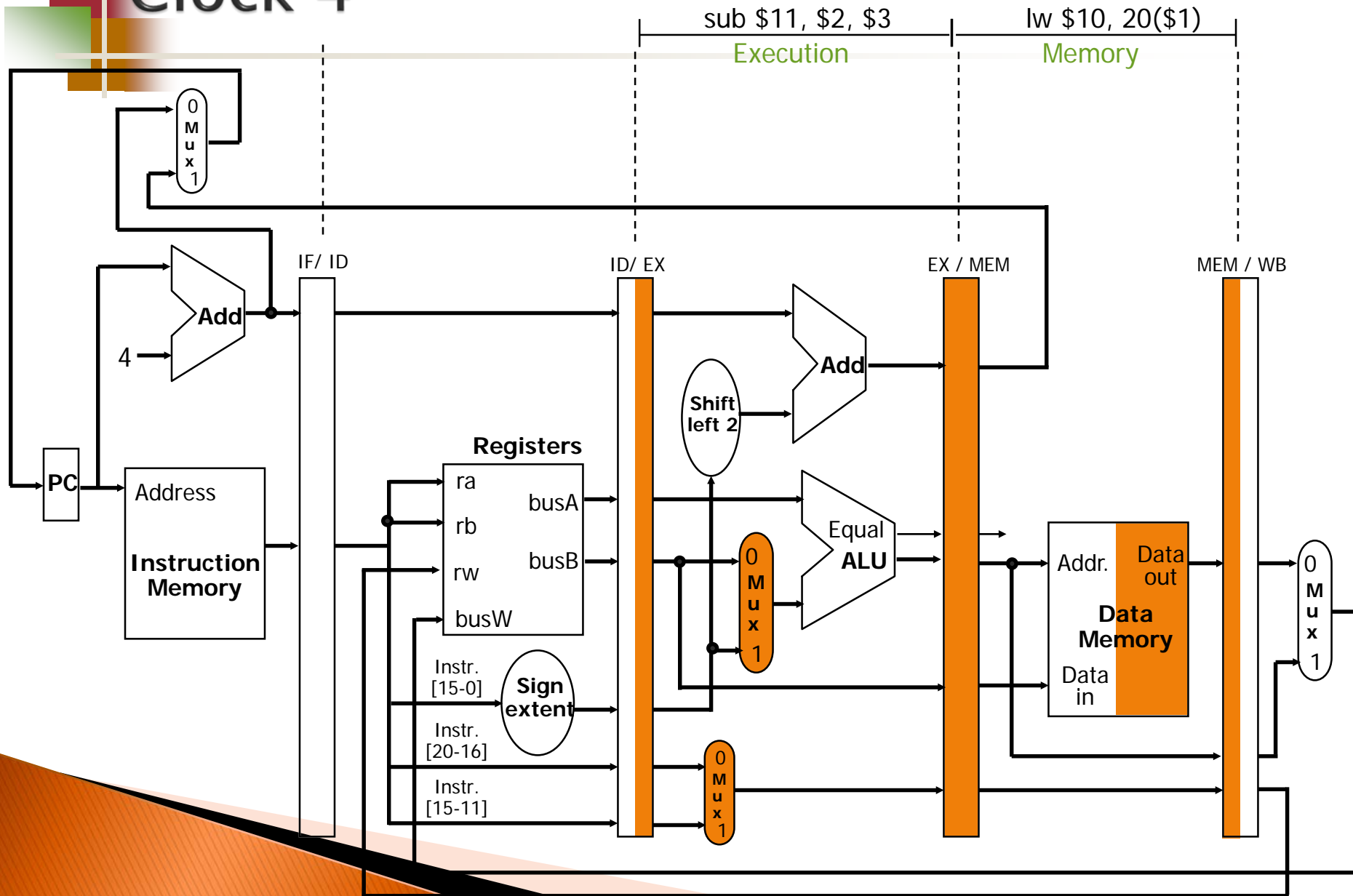
lw \$10, 20(\$1)

Instruction decode

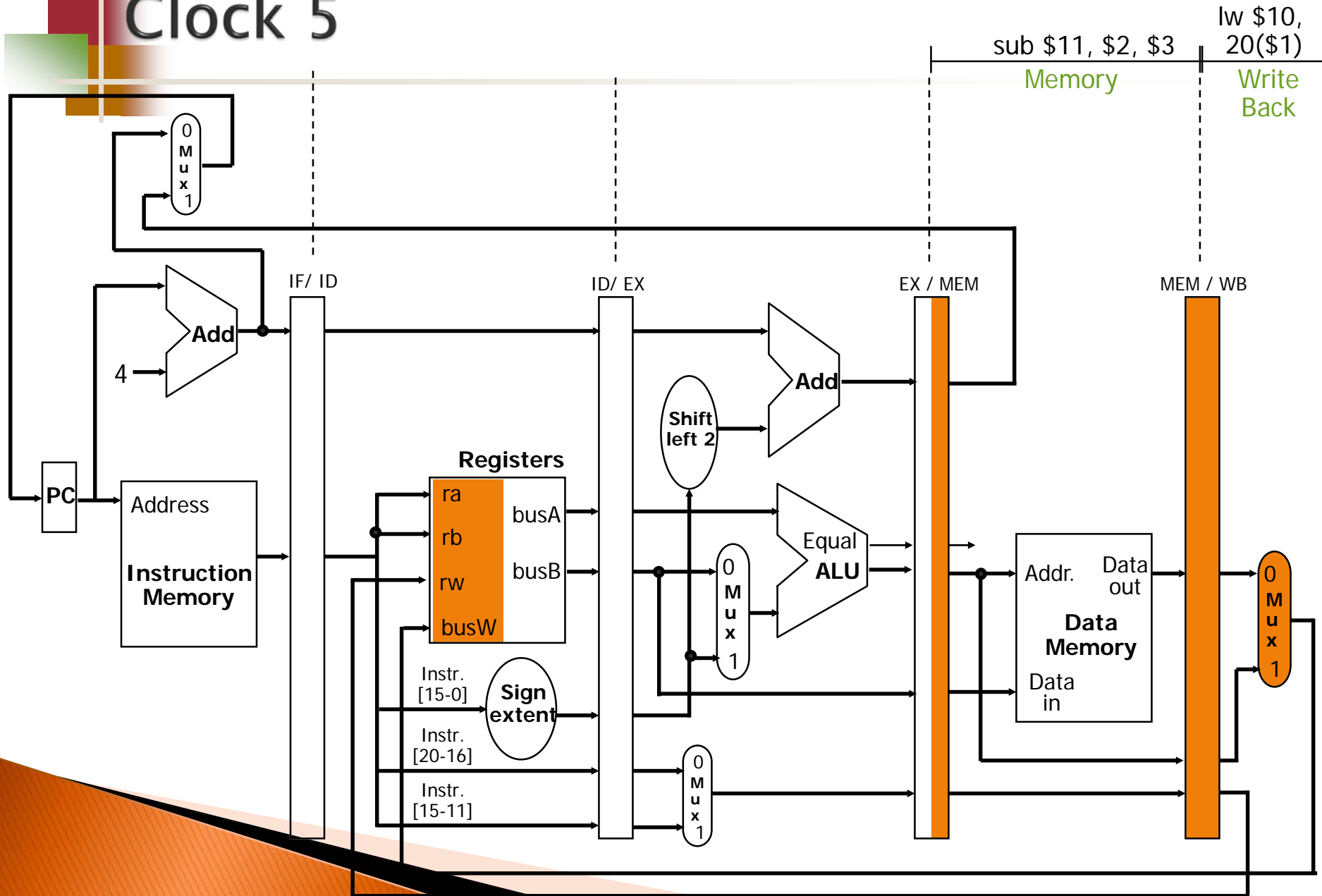
Execution



Clock 4



Clock 5





Summary: Pipelining

▶ What makes it easy

- all instructions are the same length
- just a few instruction formats
- memory operands appear only in loads and stores

▶ What makes it hard?

- structural hazards: suppose we had only one memory
- control hazards: need to worry about branch instructions
- data hazards: an instruction depends on a previous instruction

▶ Pipelining is a fundamental concept

- multiple steps using distinct resources

▶ The modern processors really makes it hard:

- exception handling
 - trying to improve performance with out-of-order execution, etc.
- 