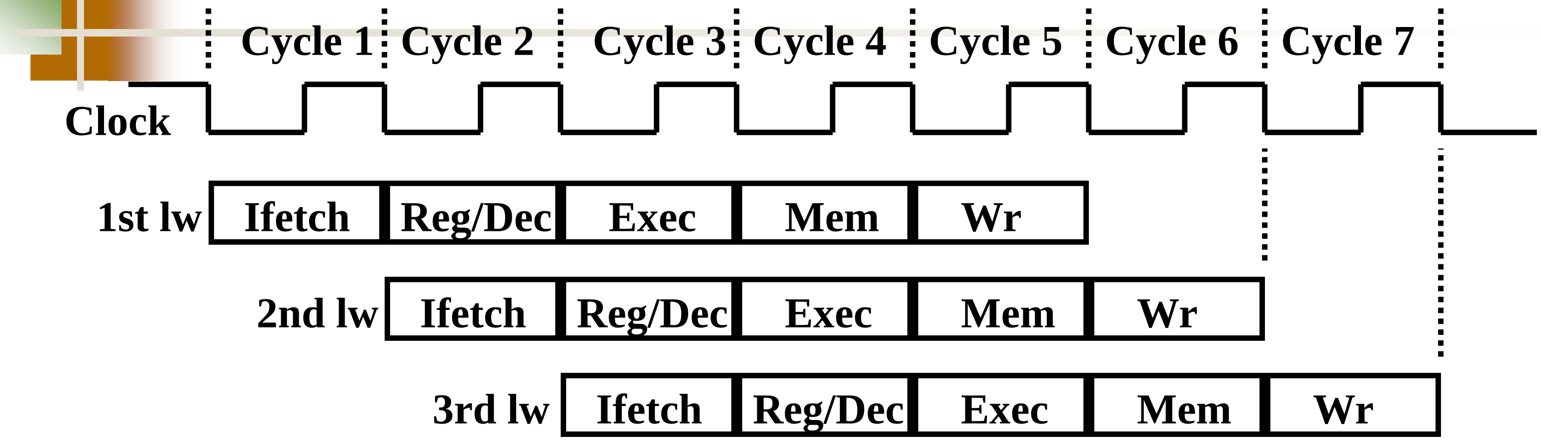
CSE331 – Computer Organization

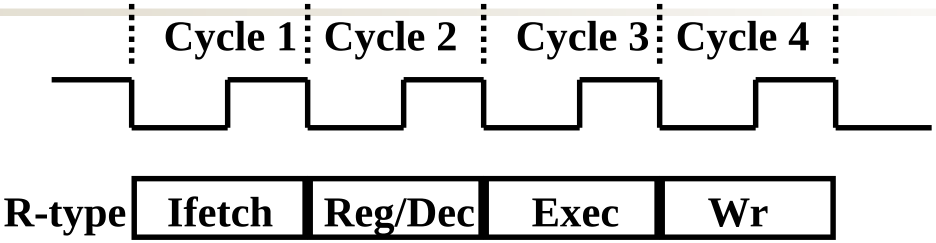
Lecture 10: A Pipelined Datapath Design

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

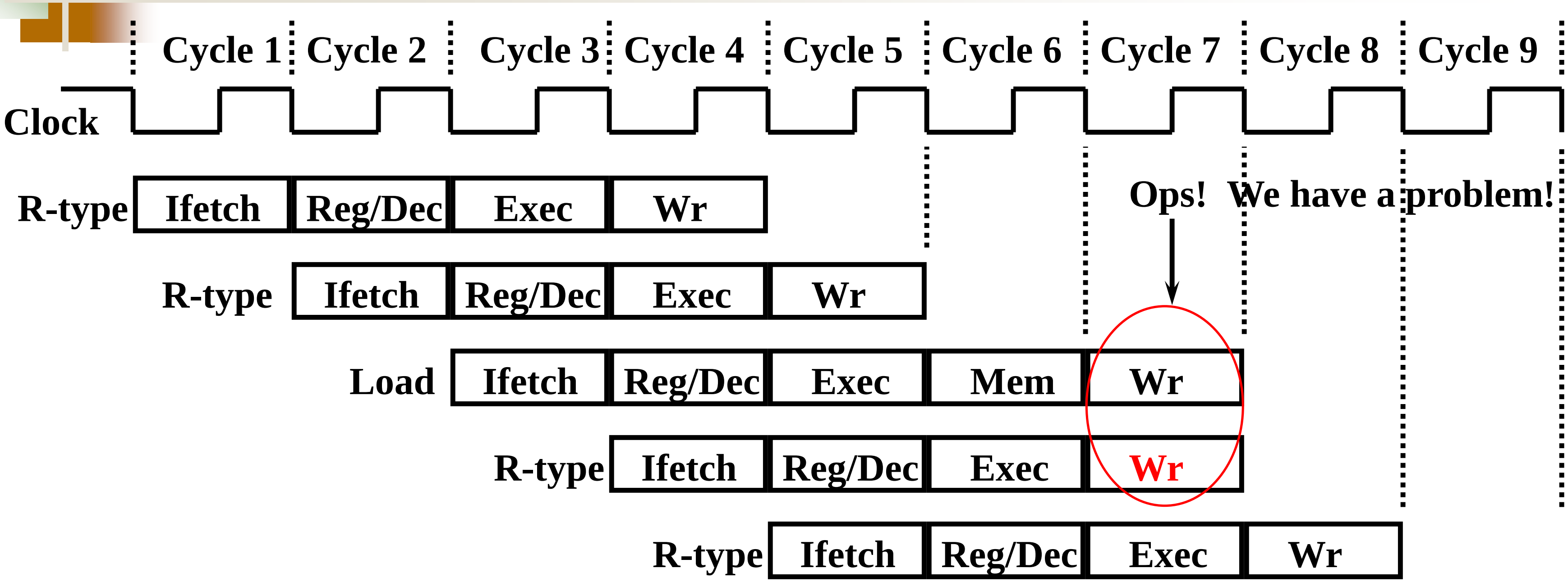
The Four Stages of R-type



	Cycle 1	Cycle 2	Cycle 3	Cycle 4
R-type	Ifetch	Reg/Dec	Exec	Wr

- 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



	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8	Cycle 9
Clock									
R-type	Ifetch	Reg/Dec	Exec	Wr					
R-type		Ifetch	Reg/Dec	Exec	Wr				
Load			Ifetch	Reg/Dec	Exec	Mem	Wr		
R-type				Ifetch	Reg/Dec	Exec	Wr		
R-type					Ifetch	Reg/Dec	Exec	Wr	

Ops! We have a problem!

- 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

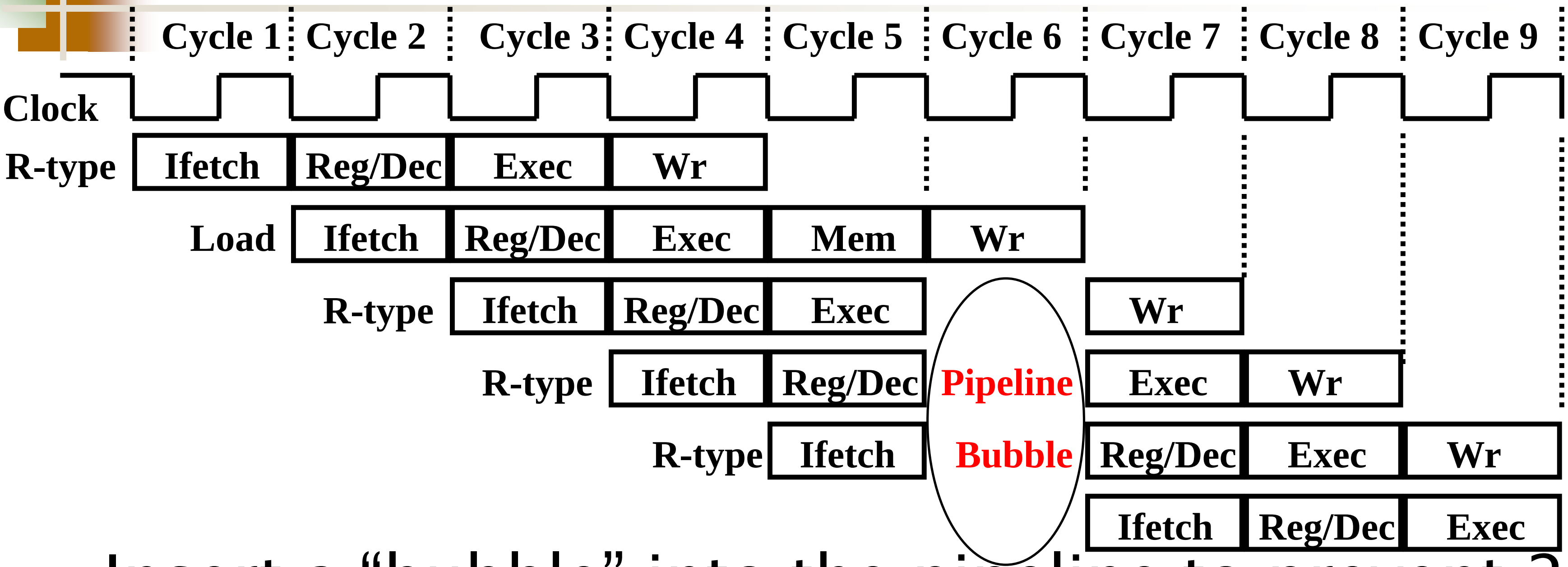
	1	2	3	4	5
Load	**Ifetch**	**Reg/Dec**	**Exec**	**Mem**	**Wr**

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

	1	2	3	4
R-type	**Ifetch**	**Reg/Dec**	**Exec**	**Wr**

  - 2 ways to solve this pipeline hazard.

Solution 1: Insert "Bubble" into the Pipeline

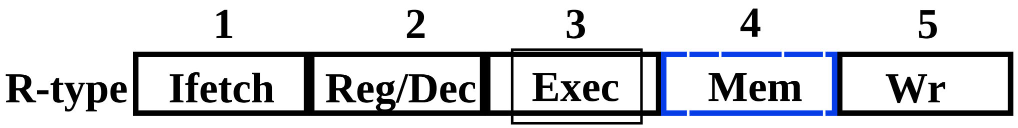


	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8	Cycle 9
Clock									
R-type	Ifetch	Reg/Dec	Exec	Wr					
Load		Ifetch	Reg/Dec	Exec	Mem	Wr			
R-type			Ifetch	Reg/Dec	Exec		Wr		
R-type				Ifetch	Reg/Dec	Pipeline	Exec	Wr	
R-type					Ifetch	Bubble	Reg/Dec	Exec	Wr
							Ifetch	Reg/Dec	Exec

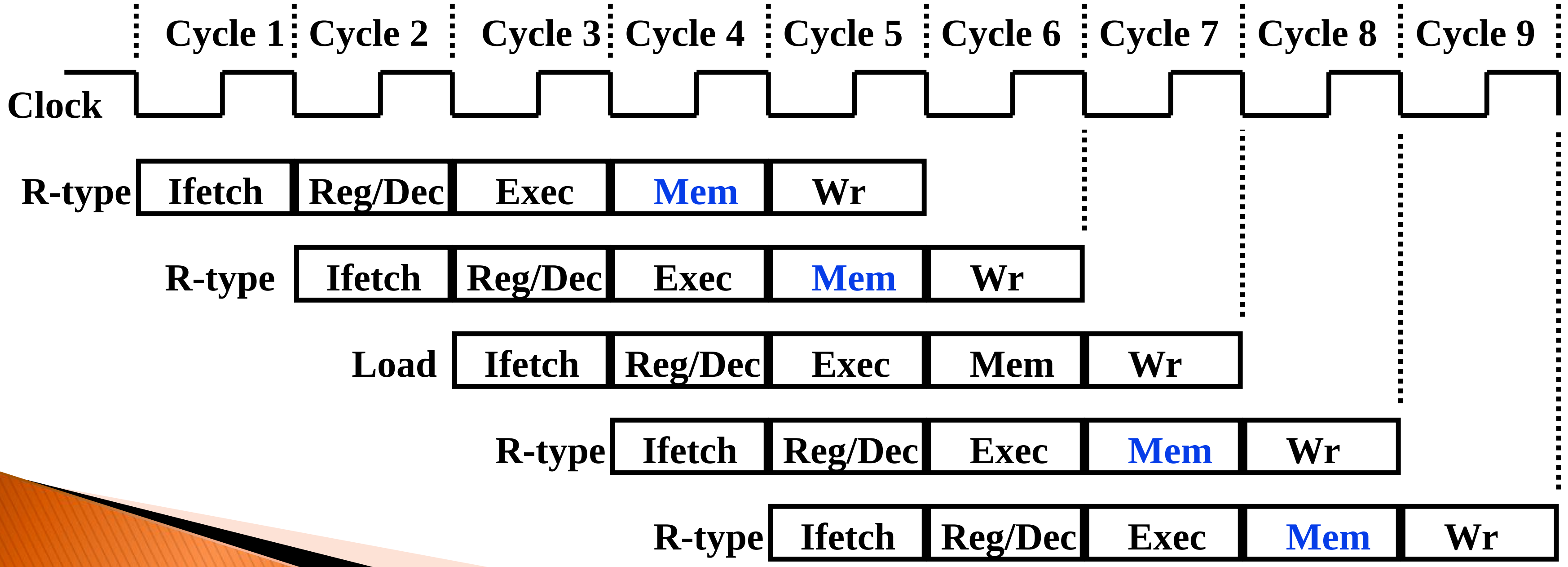
- Insert a "bubble" into the pipeline to prevent 2 writes at the same cycle
  - The control logic can be complex.
  - Lose instruction fetch and issue opportunity.
- No instruction is started in Cycle 6!

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.

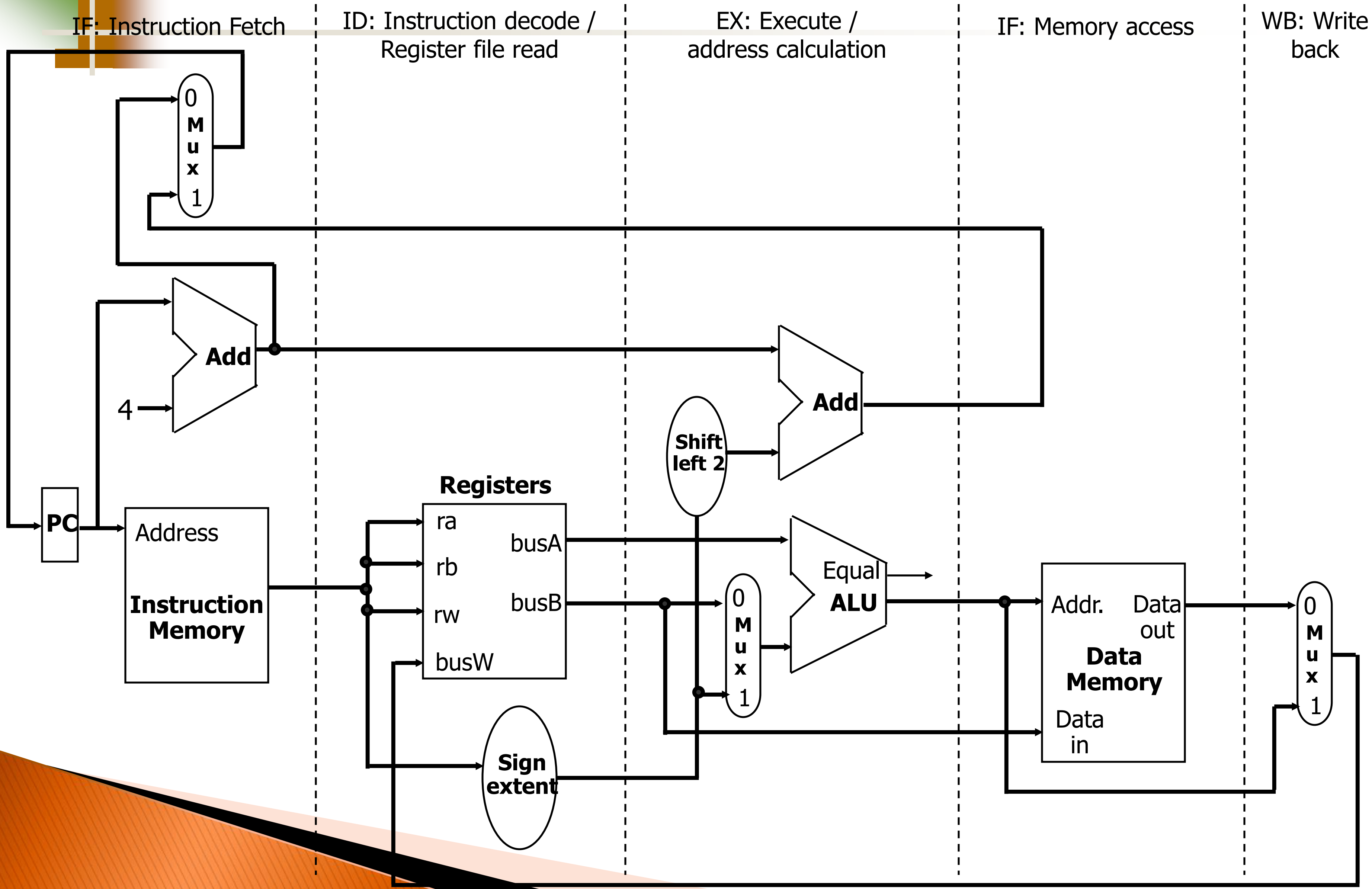


	1	2	3	4	5
R-type	Ifetch	Reg/Dec	Exec	Mem	Wr

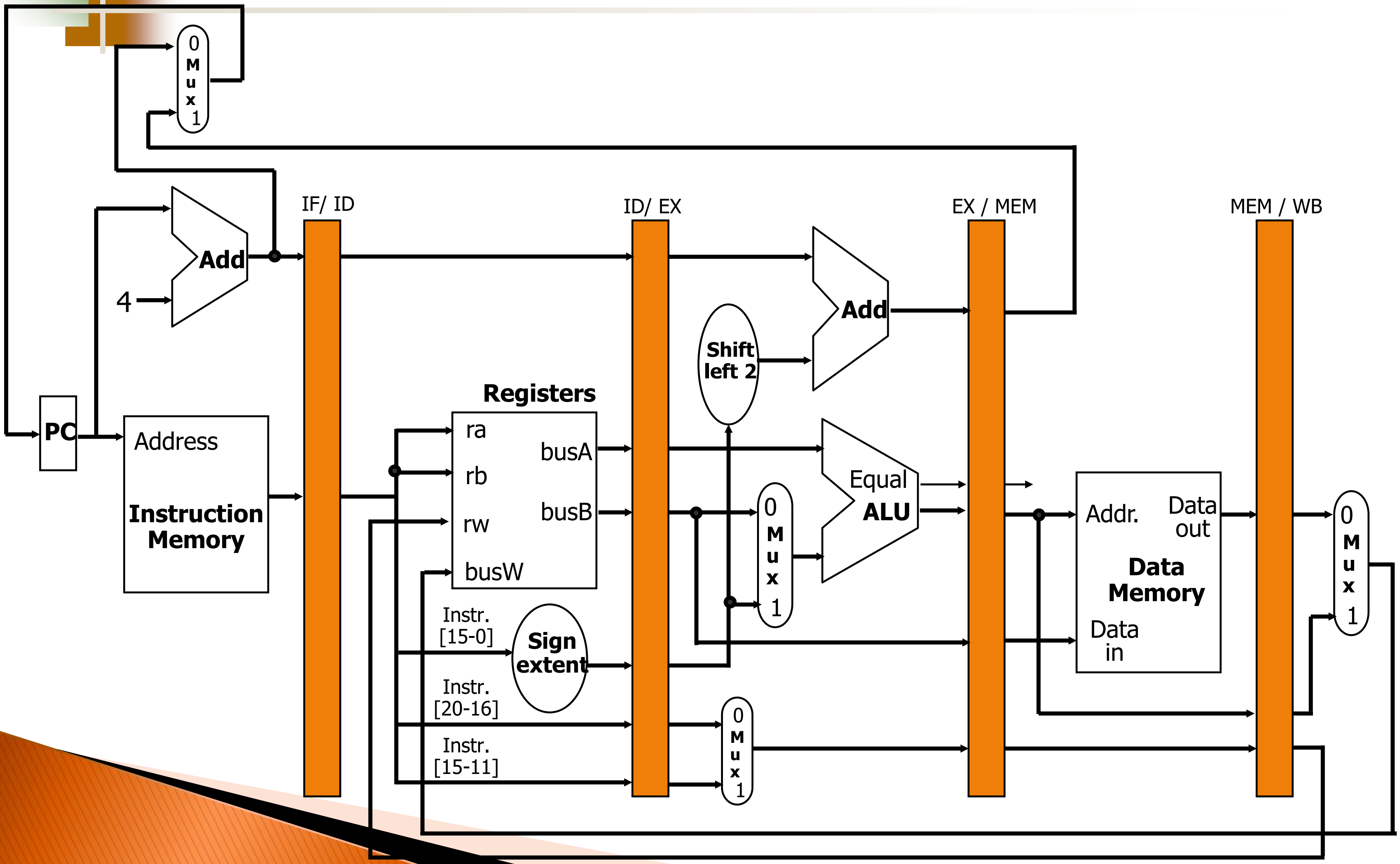


	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8	Cycle 9
Clock									
R-type	Ifetch	Reg/Dec	Exec	Mem	Wr				
R-type		Ifetch	Reg/Dec	Exec	Mem	Wr			
Load			Ifetch	Reg/Dec	Exec	Mem	Wr		
R-type				Ifetch	Reg/Dec	Exec	Mem	Wr	
R-type					Ifetch	Reg/Dec	Exec	Mem	Wr

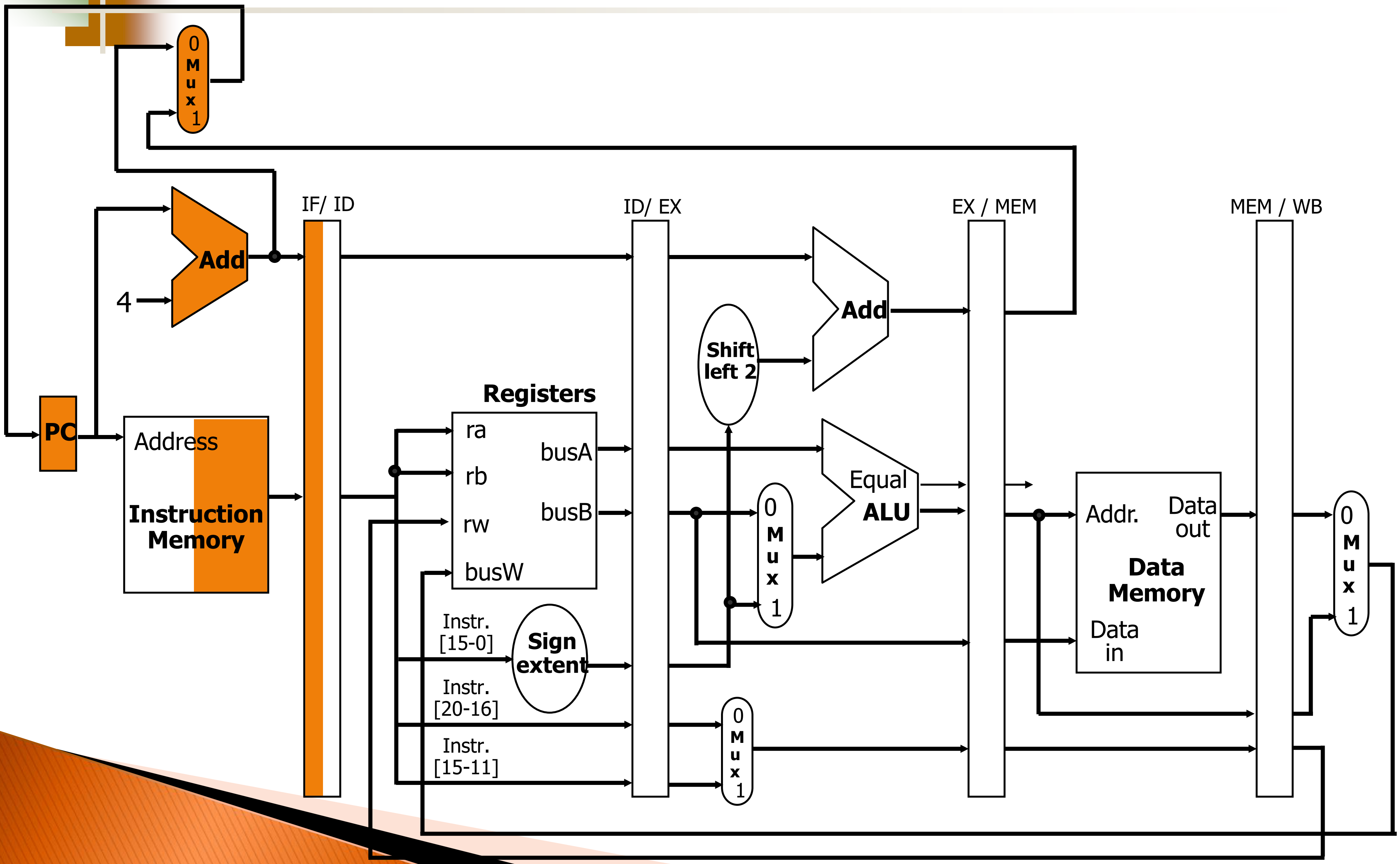
Single Cycle Datapath



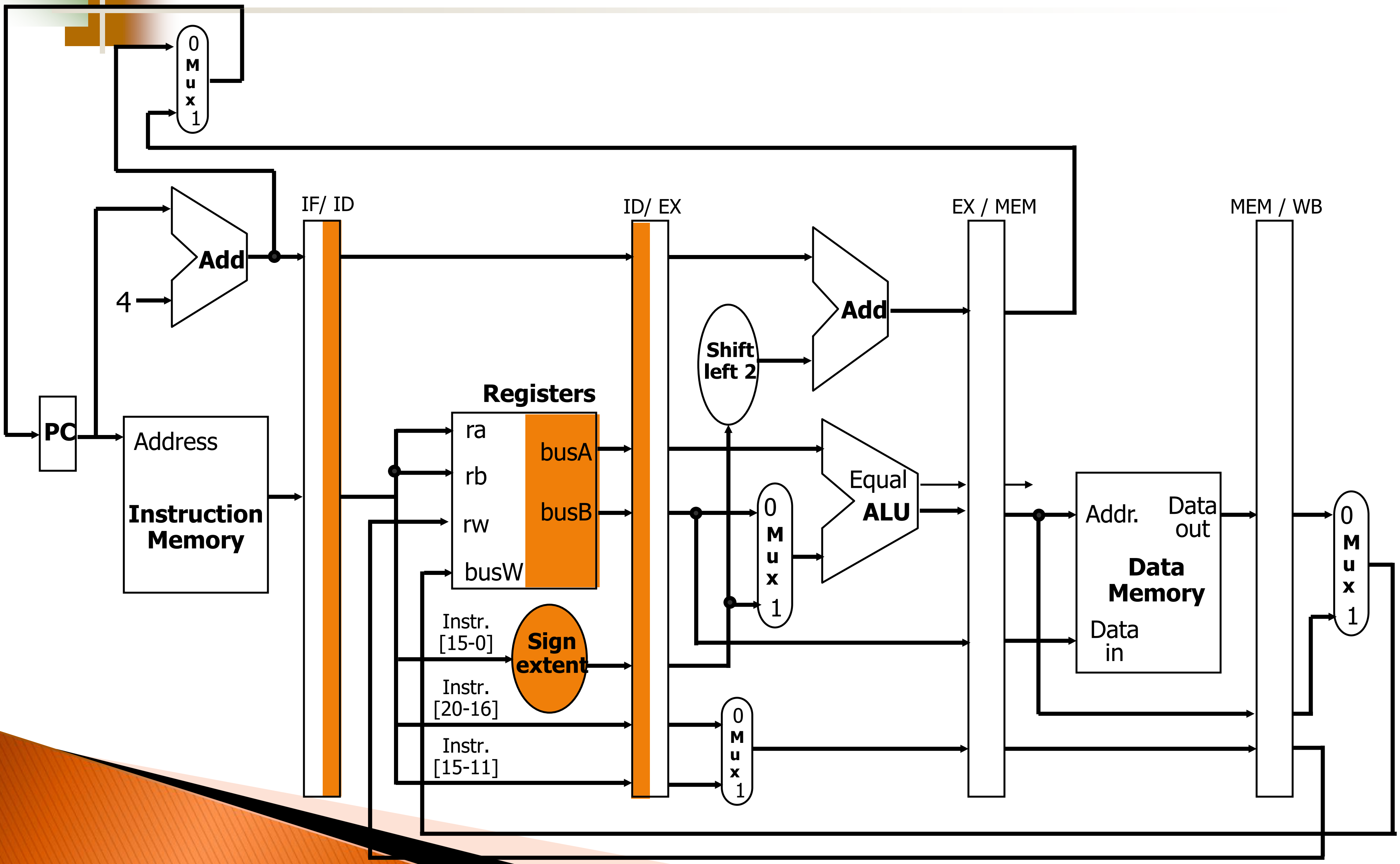
Pipelined Version of the Datapath



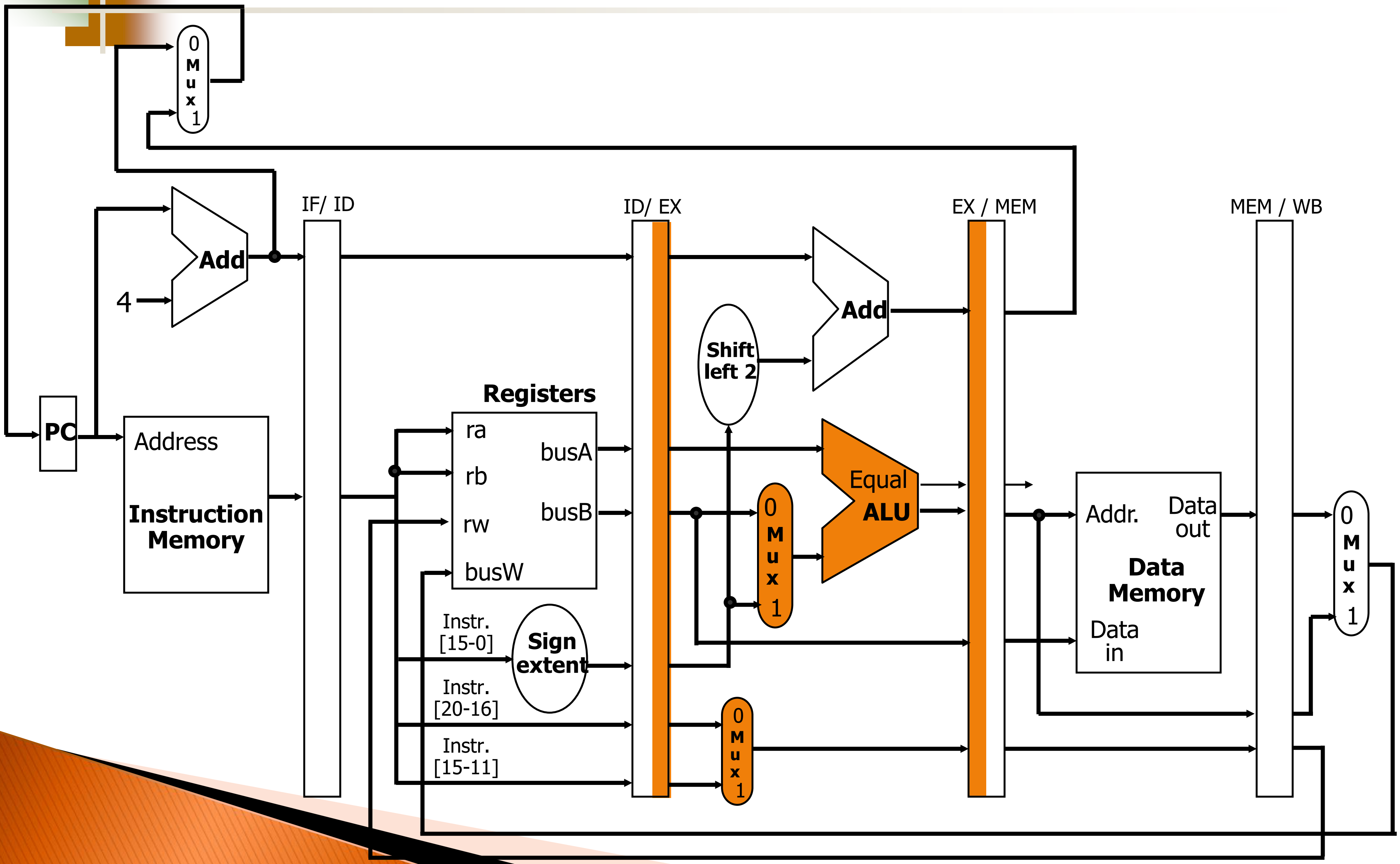
IF: The first pipe stage of a load instruction



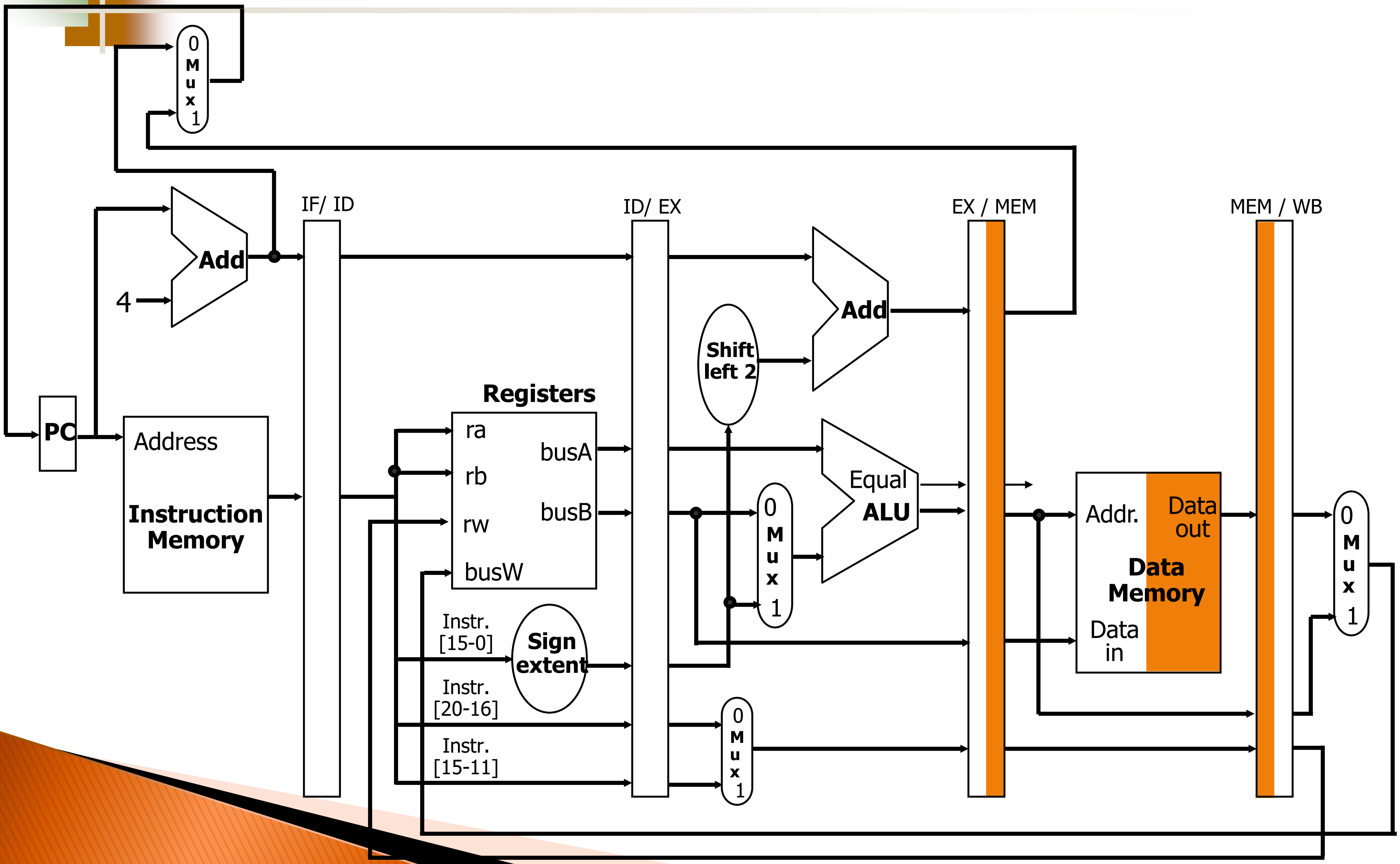
ID: The second pipe stage of a load instruction



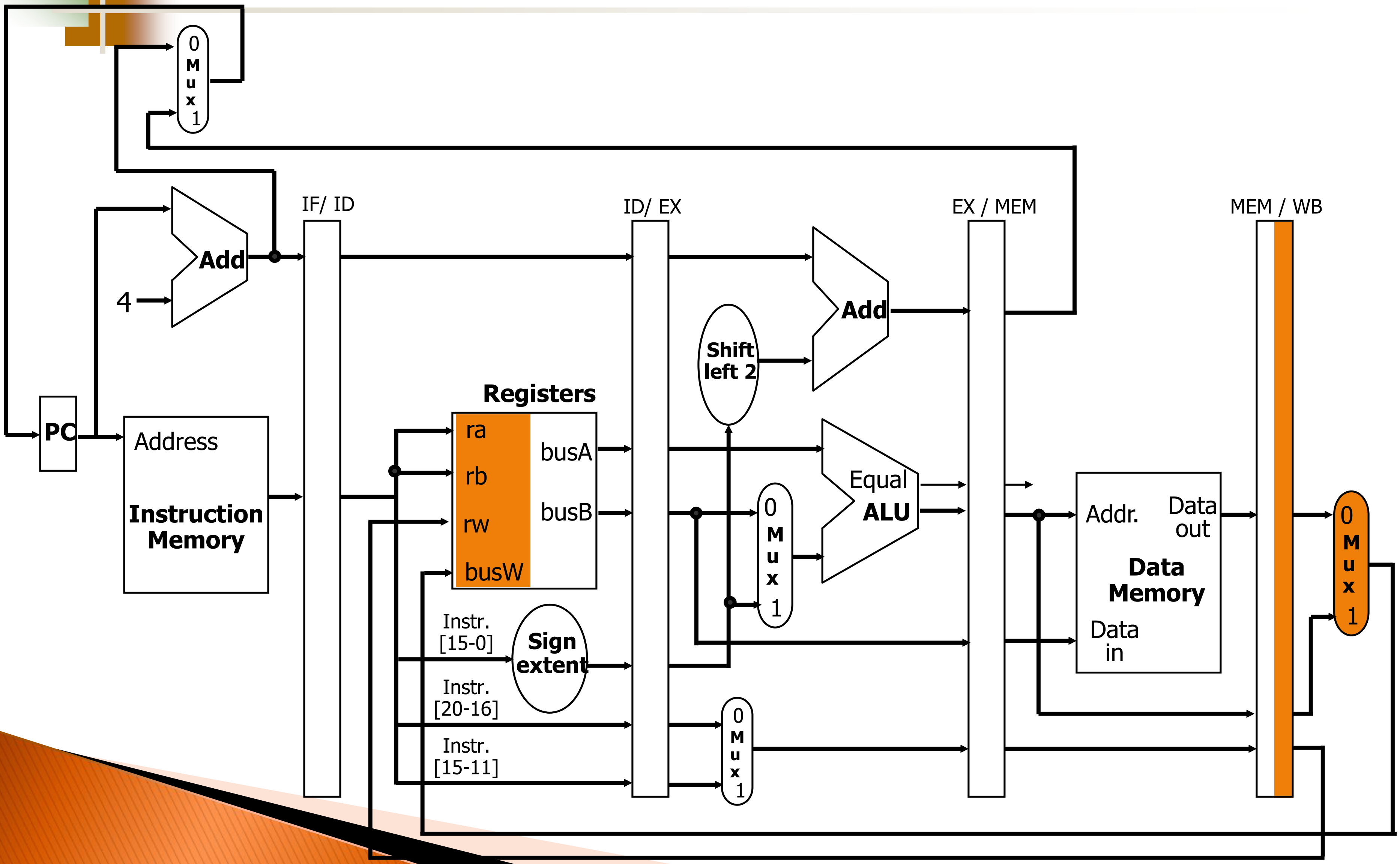
EX: The third pipe stage of a load instruction



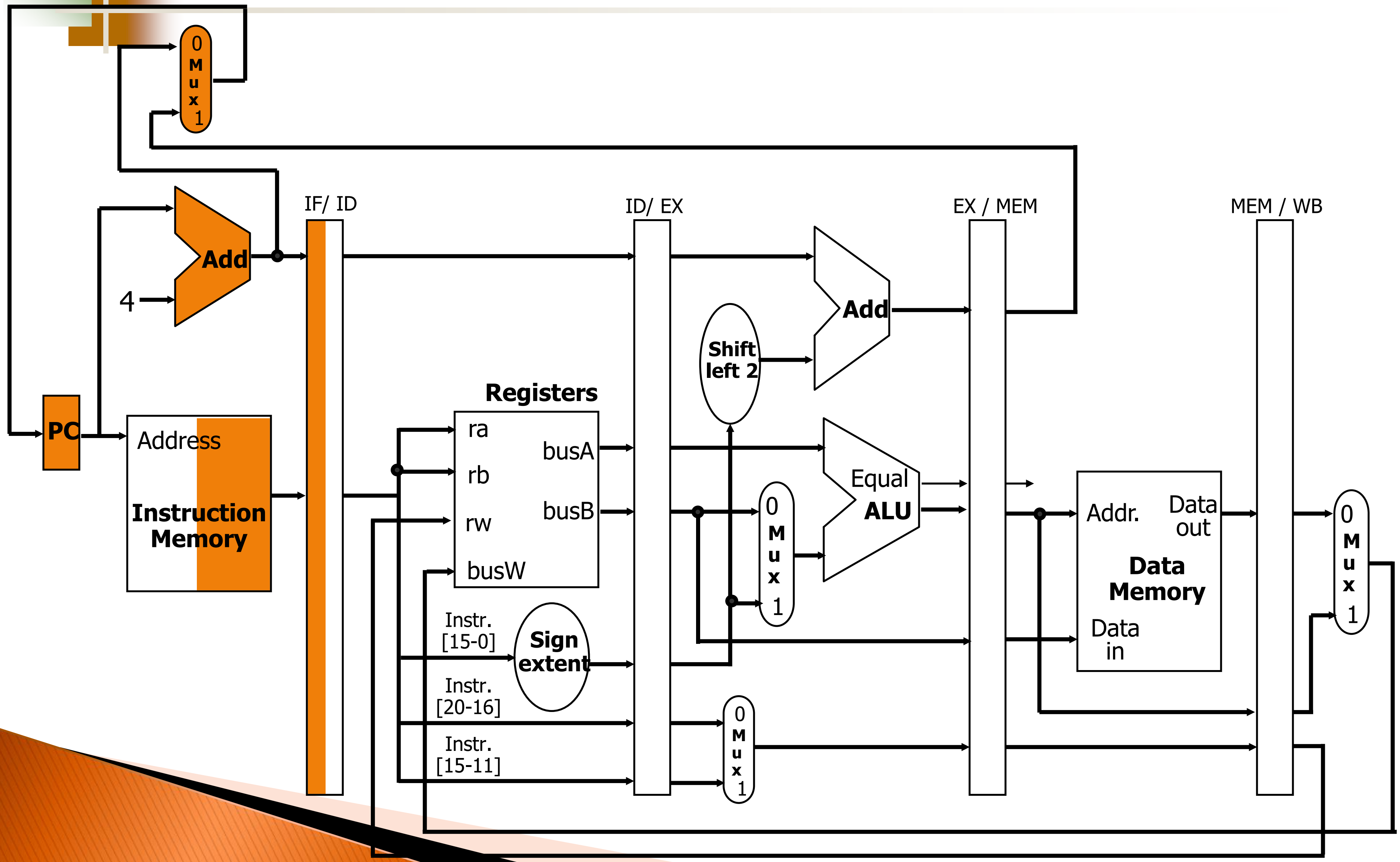
MEM: The fourth pipe stage of a load instruction



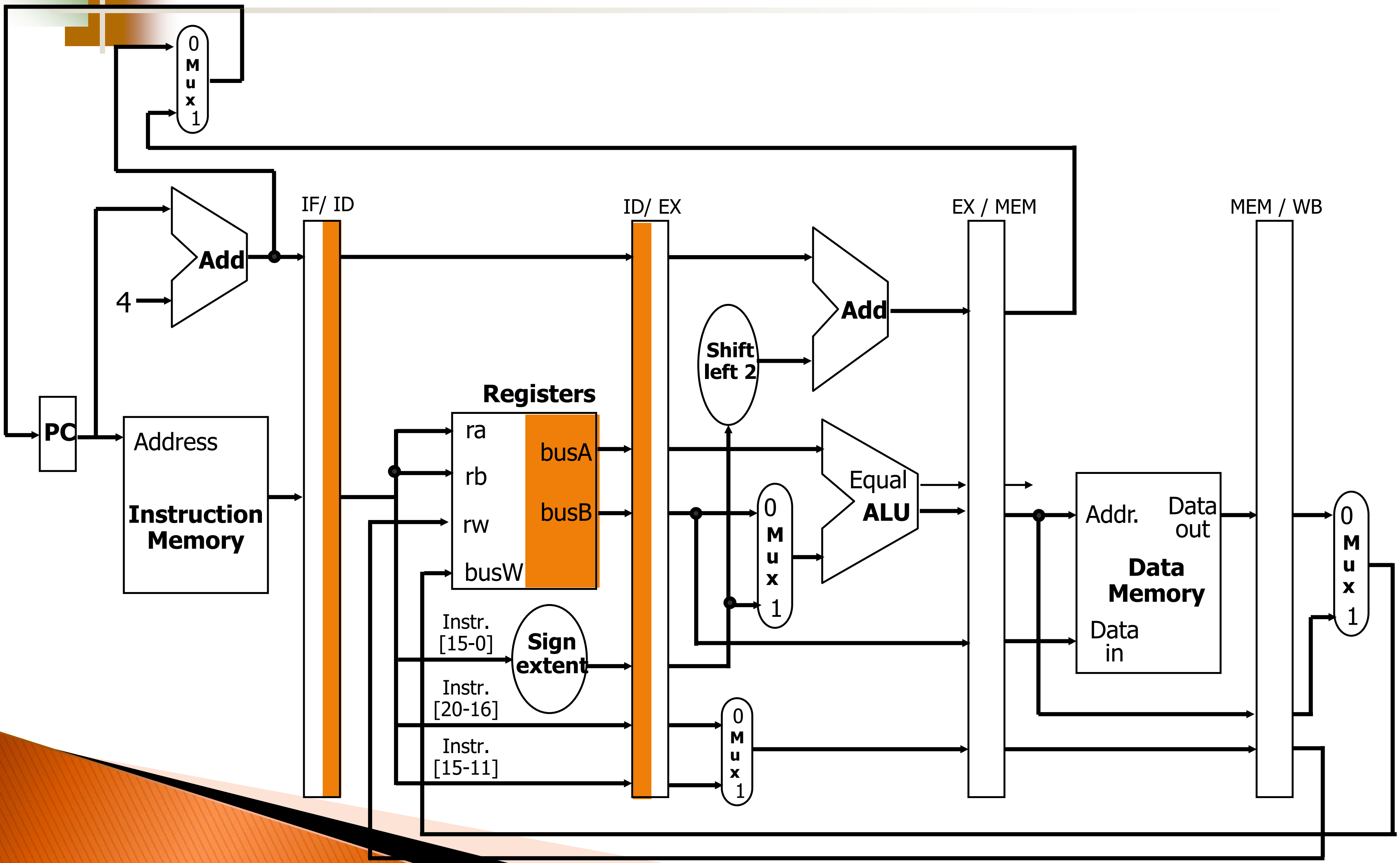
WB: The fifth pipe stage of a load instruction



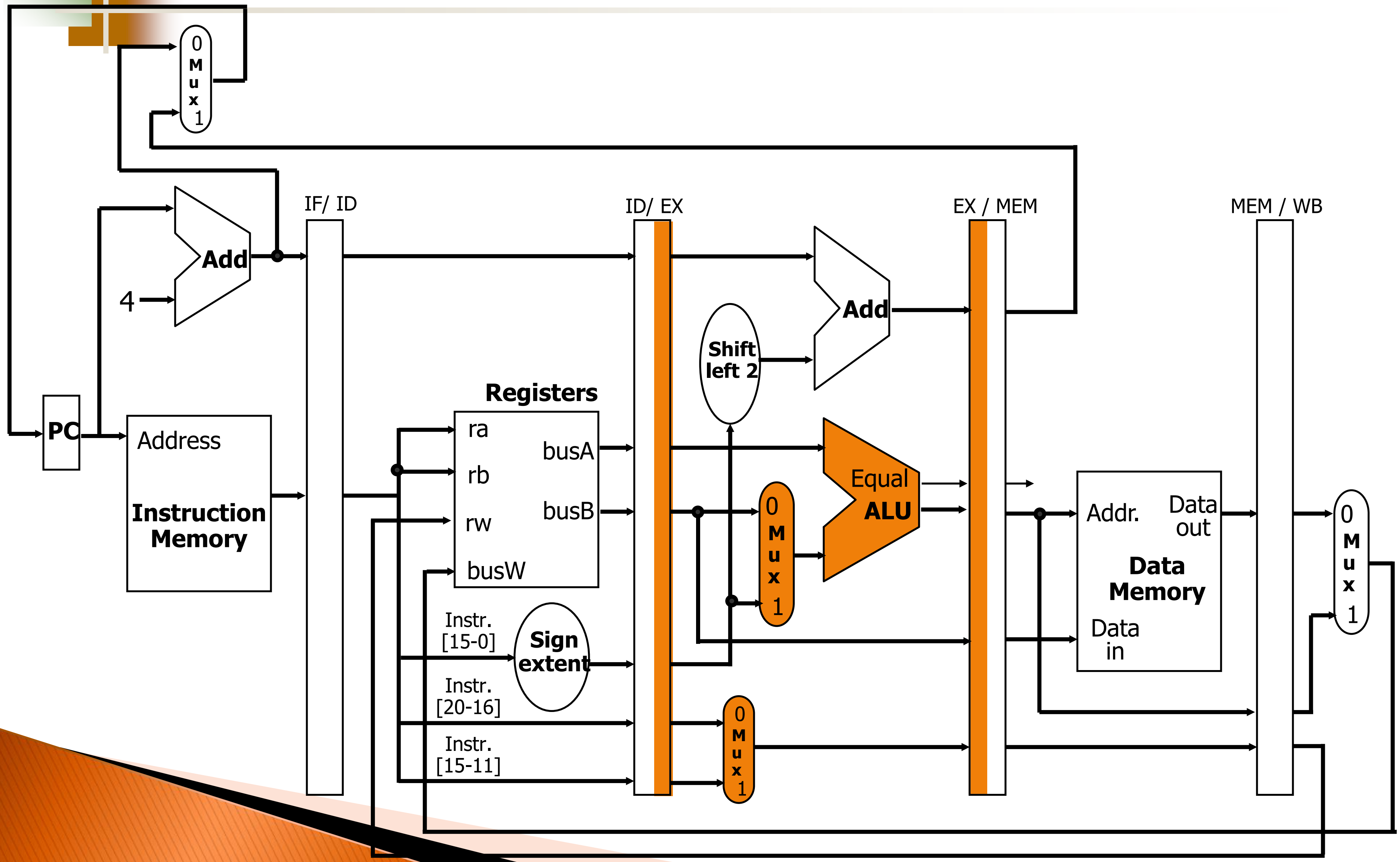
IF: The first pipe stage of a R-type instruction



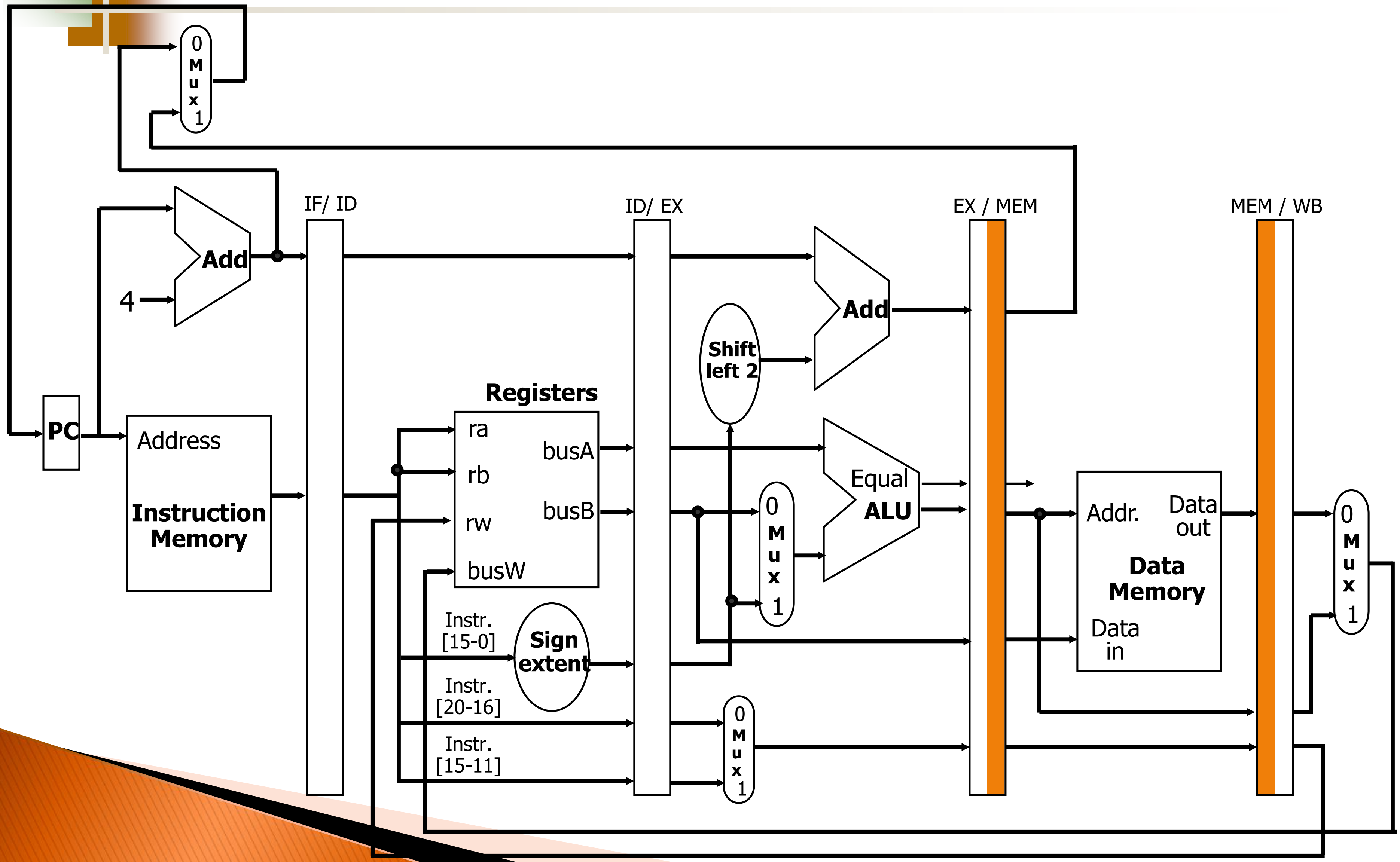
ID: The second pipe stage of a R-type instruction



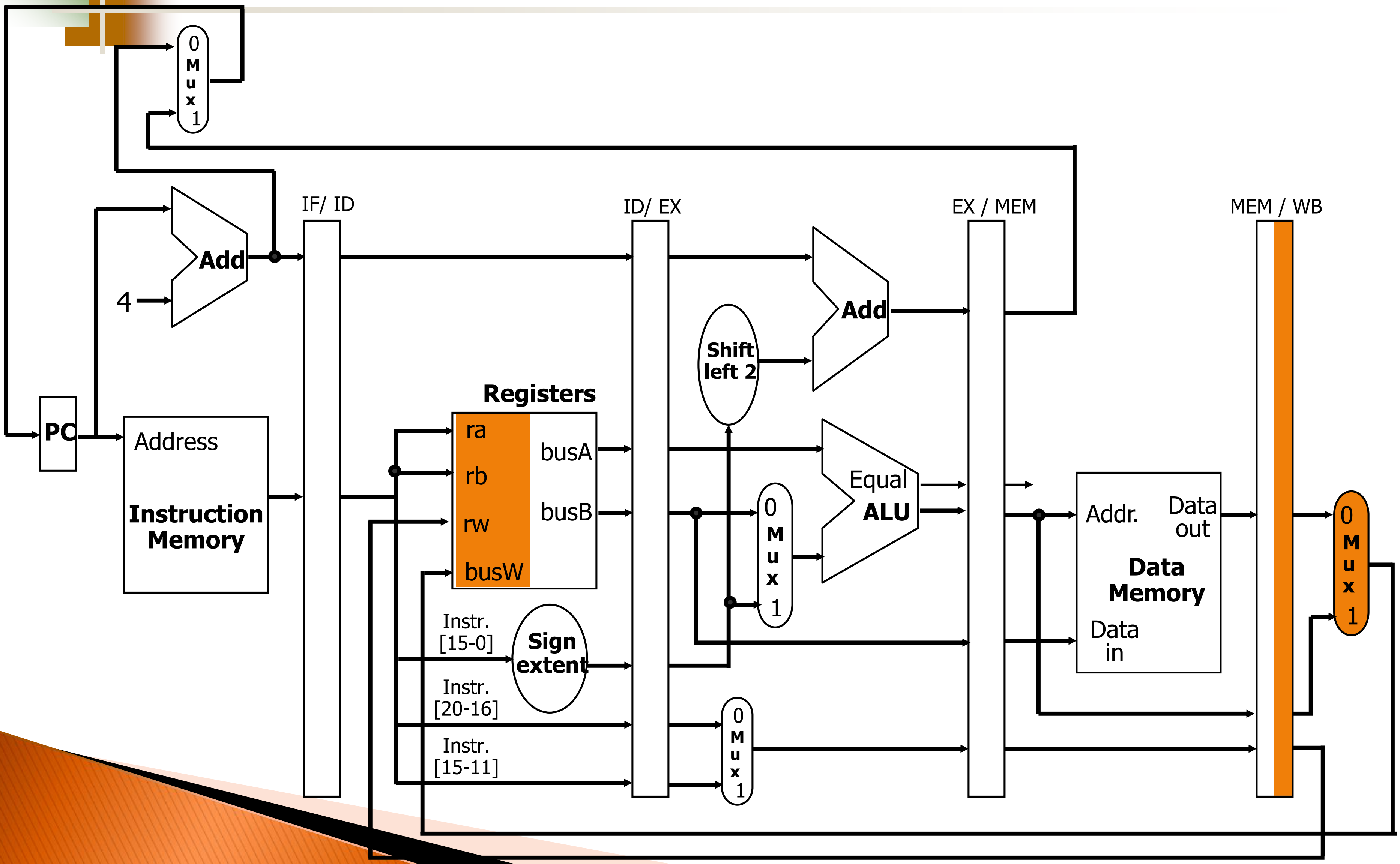
EX: The third pipe stage of a R-type instruction



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



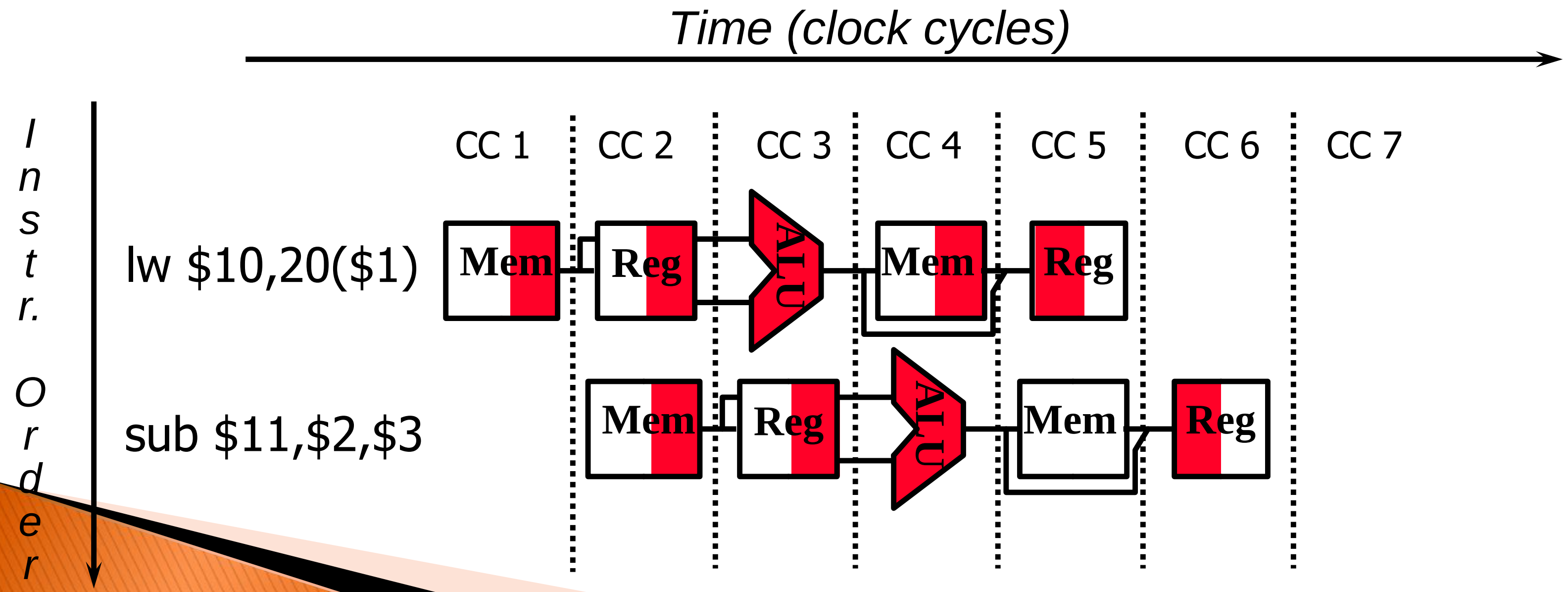
WB: The fifth pipe stage of a R-type instruction



An Example to Clarify Pipelining

- Since many instructions are simultaneously are executing in a single cycle datapath, it can be difficult to understand.
- The following code will be examined:

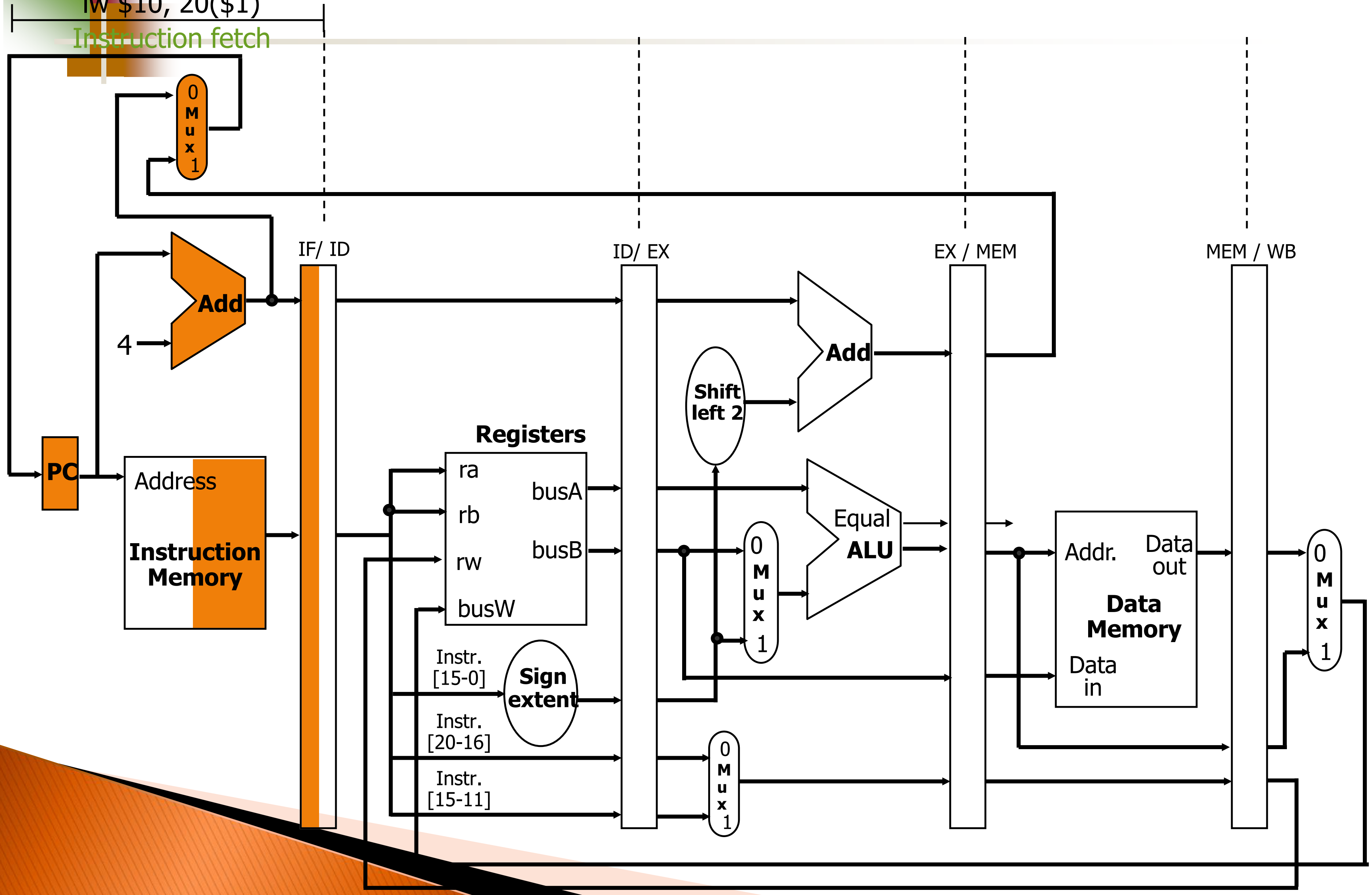
```
lw   $10, 20($1)
sub $11, $2, $3
```



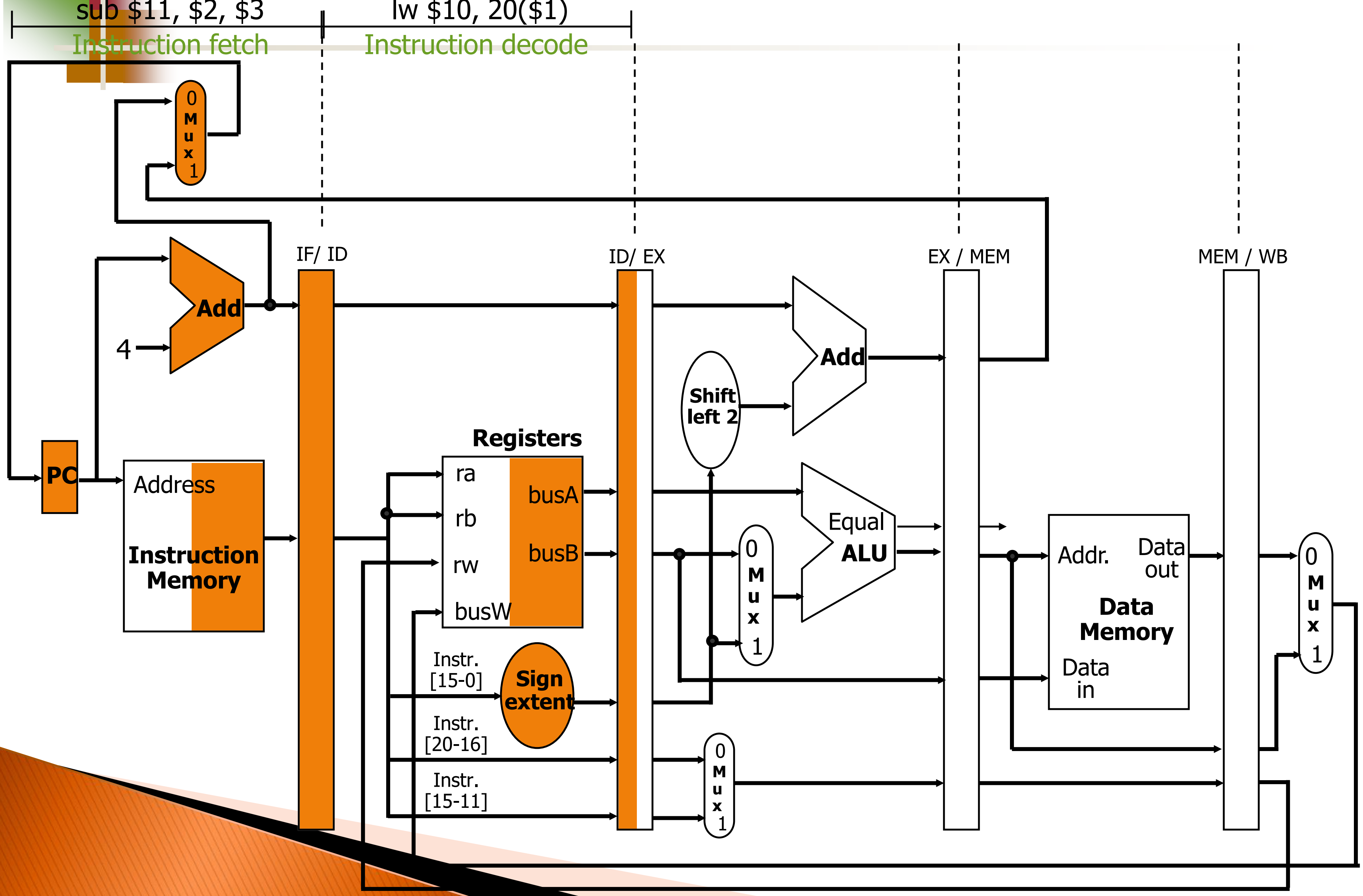
Clock 1

lw $10, 20($1)

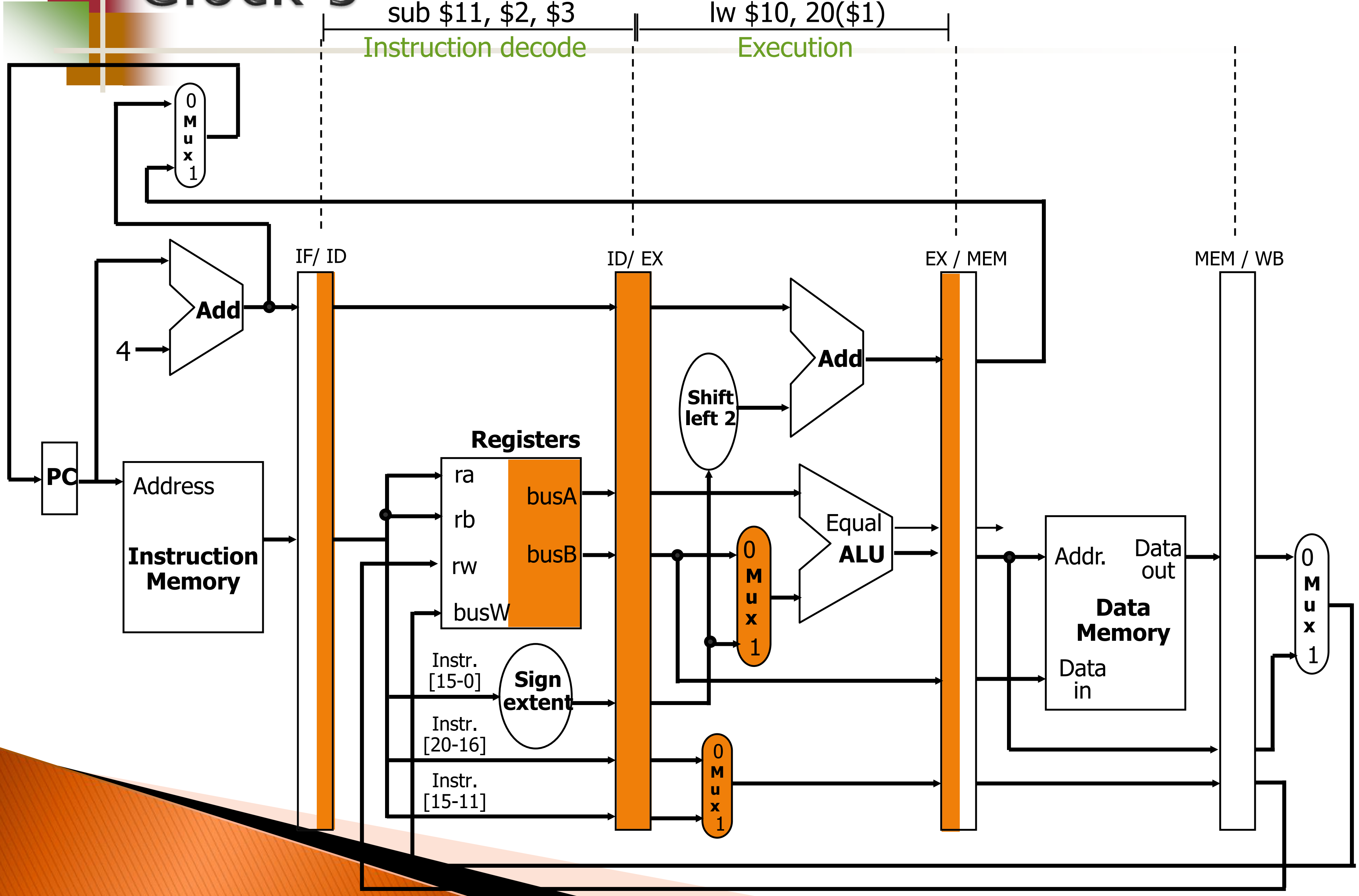
Instruction fetch



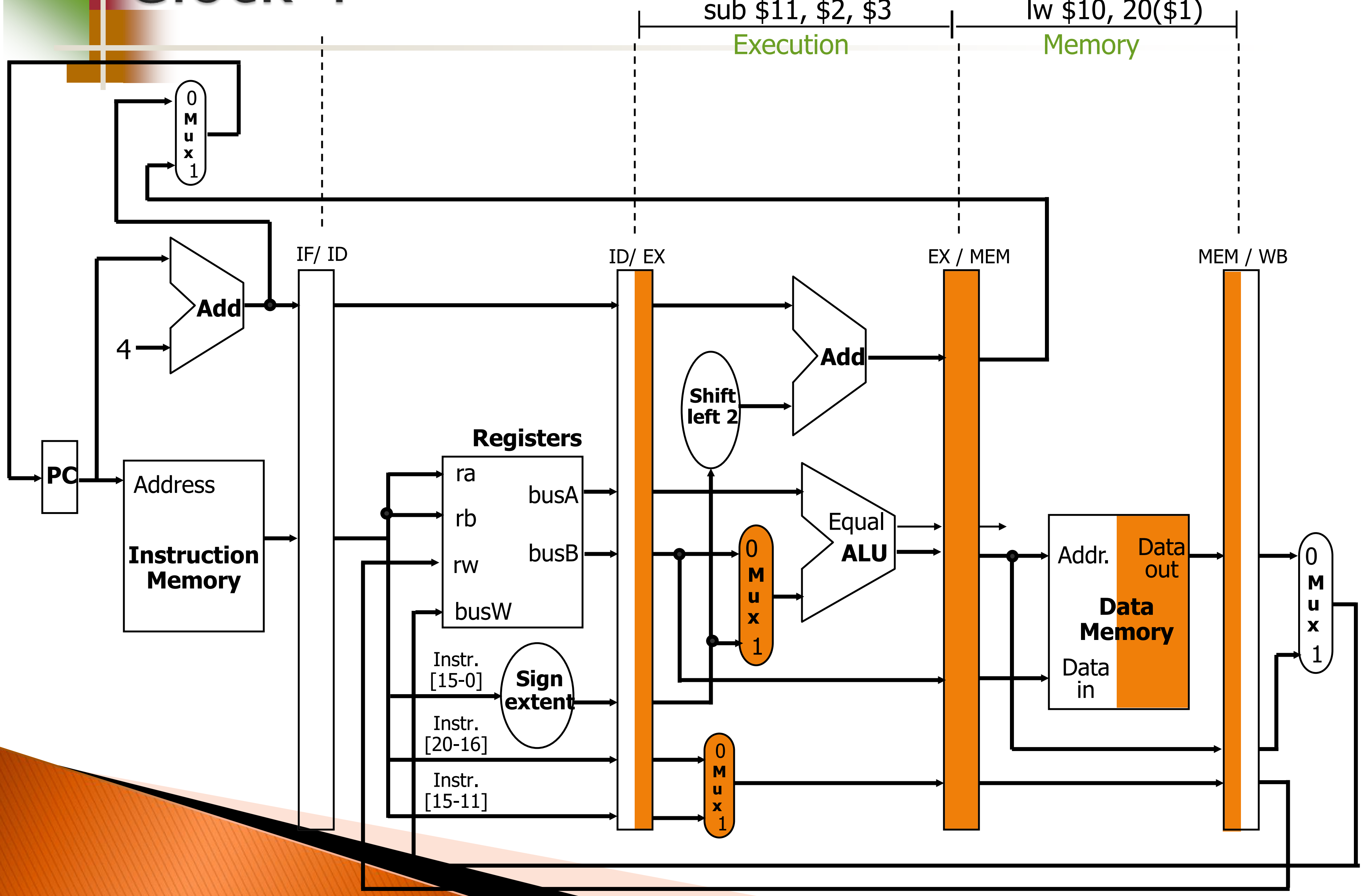
Clock 2


sub $11, $2, $3
lw $10, 20($1)
Instruction fetch
Instruction decode
0
M
u
x
1
IF/ ID
ID/ EX
EX / MEM
MEM / WB
Add
4
PC
Address
Instruction Memory
Registers
ra
rb
rw
busW
busA
busB
Instr. [15-0]
Instr. [20-16]
Instr. [15-11]
Sign extent
Shift left 2
Add
Equal
ALU
0
M
u
x
1
0
M
u
x
1
Addr.
Data out
Data Memory
Data in
0
M
u
x
1

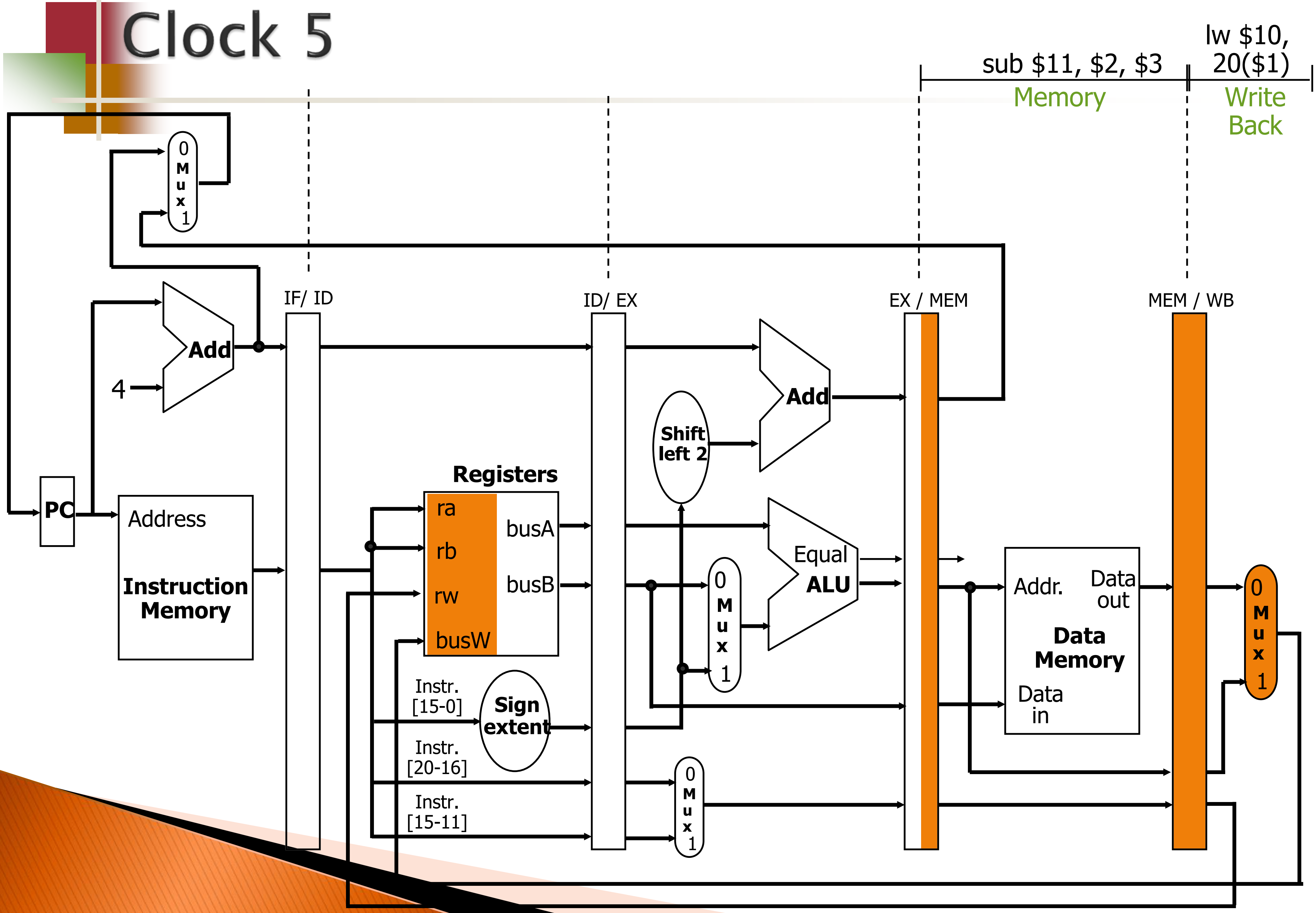
Clock 3


sub $11, $2, $3
Instruction decode
lw $10, 20($1)
Execution
0
M
u
x
1
IF/ ID
ID/ EX
EX / MEM
MEM / WB
Add
4
Add
Shift left 2
Registers
PC
Address
Instruction Memory
ra
rb
rw
busW
busA
busB
0
M
u
x
1
Equal
ALU
Addr.
Data out
Data Memory
Data in
0
M
u
x
1
Instr. [15-0]
Sign extent
Instr. [20-16]
Instr. [15-11]
0
M
u
x
1

Clock 4


sub $11, $2, $3
Execution
lw $10, 20($1)
Memory
0
M
u
x
1
IF/ ID
ID/ EX
EX / MEM
MEM / WB
Add
4
PC
Address
Instruction Memory
Registers
ra
rb
rw
busW
busA
busB
Shift left 2
Add
Equal
ALU
0
M
u
x
1
Instr. [15-0]
Sign extent
Instr. [20-16]
Instr. [15-11]
0
M
u
x
1
Addr.
Data out
Data Memory
Data in
0
M
u
x
1

Clock 5


sub $11, $2, $3
Memory
lw $10, 20($1)
Write Back
0
M
u
x
1
IF/ ID
ID/ EX
EX / MEM
MEM / WB
Add
4
Add
Shift left 2
PC
Address
Instruction Memory
Registers
ra
rb
rw
busW
busA
busB
Equal
ALU
0
M
u
x
1
Addr.
Data out
Data Memory
Data in
0
M
u
x
1
Instr. [15-0]
Sign extent
Instr. [20-16]
Instr. [15-11]
0
M
u
x
1

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.