

# PIPELINED PROCESSOR (I)

---

Lecturer: Hui Annie Guo

[h.guo@unsw.edu.au](mailto:h.guo@unsw.edu.au)

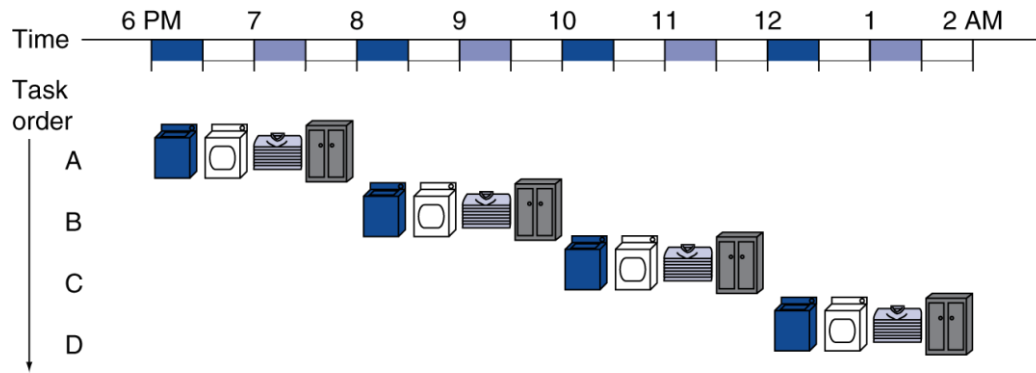
K17-501F

# Lecture overview

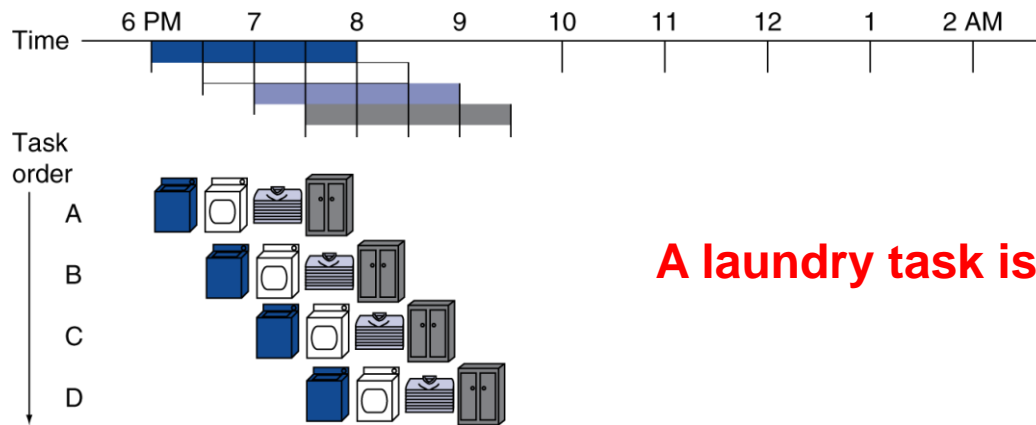
- **Topics**
  - **Pipelining datapath**
  - **Pipelining control**
- **Suggested reading**
  - **H&P Chapter 4.5 – 4.6**

# Pipelining: example

## • Pipelined laundry



- **Four loads:**
  - Speedup  
=  $8/3.5 = 2.3$



**A laundry task is pipelined into four stages**

# MIPS pipelined datapath


## Five stages:

1. **IF: Instruction fetch from memory**
2. **ID: Instruction decode & register read**
3. **EX: Execute operation or calculate address**
4. **MEM: Access memory operand**
5. **WB: Write result back to register**

# MIPS ISA is designed for pipelining

- **All instructions are 32-bits**
  - Easier to fetch in one cycle
  - cf. 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 an early stage, access memory in a later stage
    - Since ALU available
- **Alignment of memory operands**
  - Memory access takes “one cycle”
- **Most resources are available**

# Pipelined instruction execution

time(cc) 

Inst. #	1	2	3	4	5	6	7	8
Inst. i	IF	ID	EX	MEM	WB			
Inst. i+1		IF	ID	EX	MEM	WB		
Inst. i+2			IF	ID	EX	MEM	WB	
Inst. i+3				IF	ID	EX	MEM	WB

- In each cycle, new instruction fetched and begins 5 cycle execution
- Ideally, the pipeline completes one instruction every clock cycle
  - Namely, 5 times faster if  $T_{pipe} = (1/5) * T_{sc}$

pipelined processor

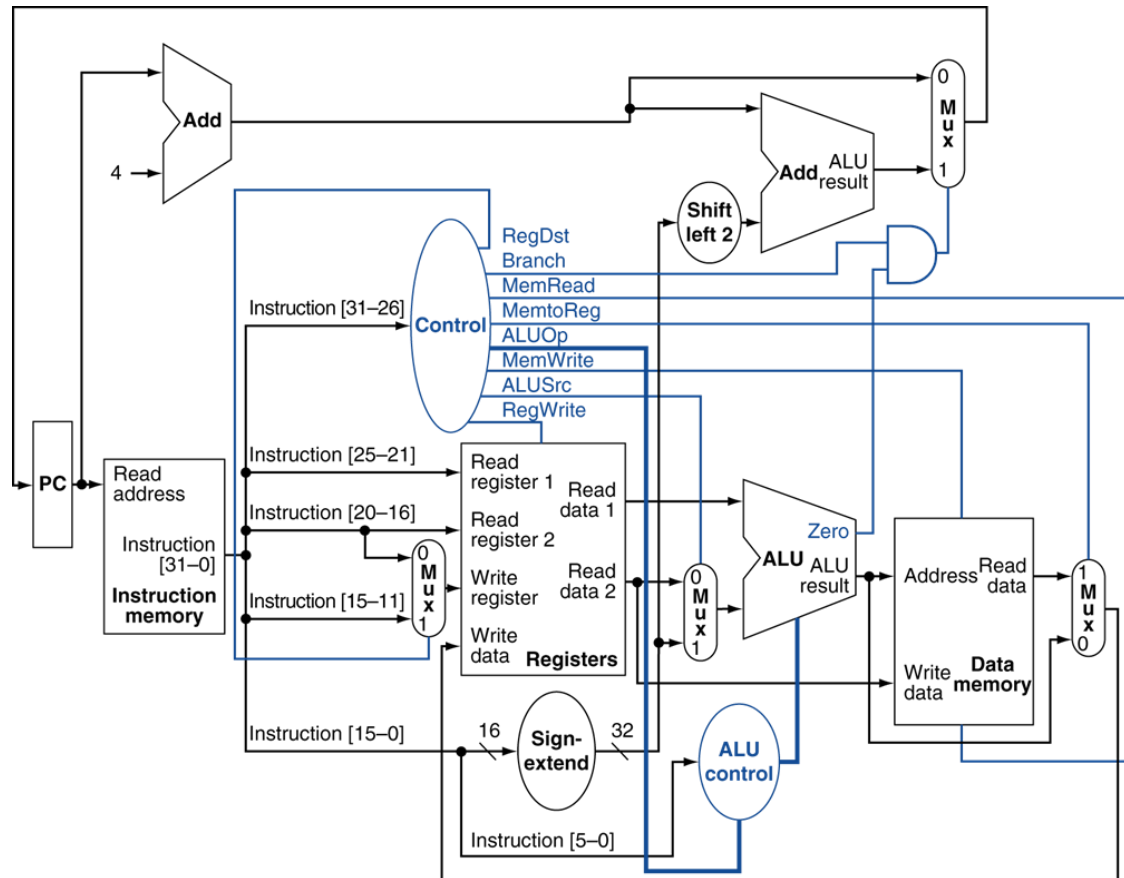
single-cycle processor

# Create pipeline stages (1)

- **Partition the single cycle datapath into sections. Each section forms a pipeline stage**
- **Considerations**
  - **Try to balance the stages for similar stage delays**
    - **To reduce the clock cycle time.**
  - **Try to make data flow in the same direction**
    - **To avoid pipeline hazards**
      - Will be covered later
  - **Make sure the correct data to be used for each instruction in the pipeline**
    - **For correct functions**

# Partition the SC datapath

- **Try to balance the stages for a small clock cycle time**
  - **The function of each section is already specified**  
IF, ID, Ex, Mem, WB
  - **But we can tune the partition for a better solution**

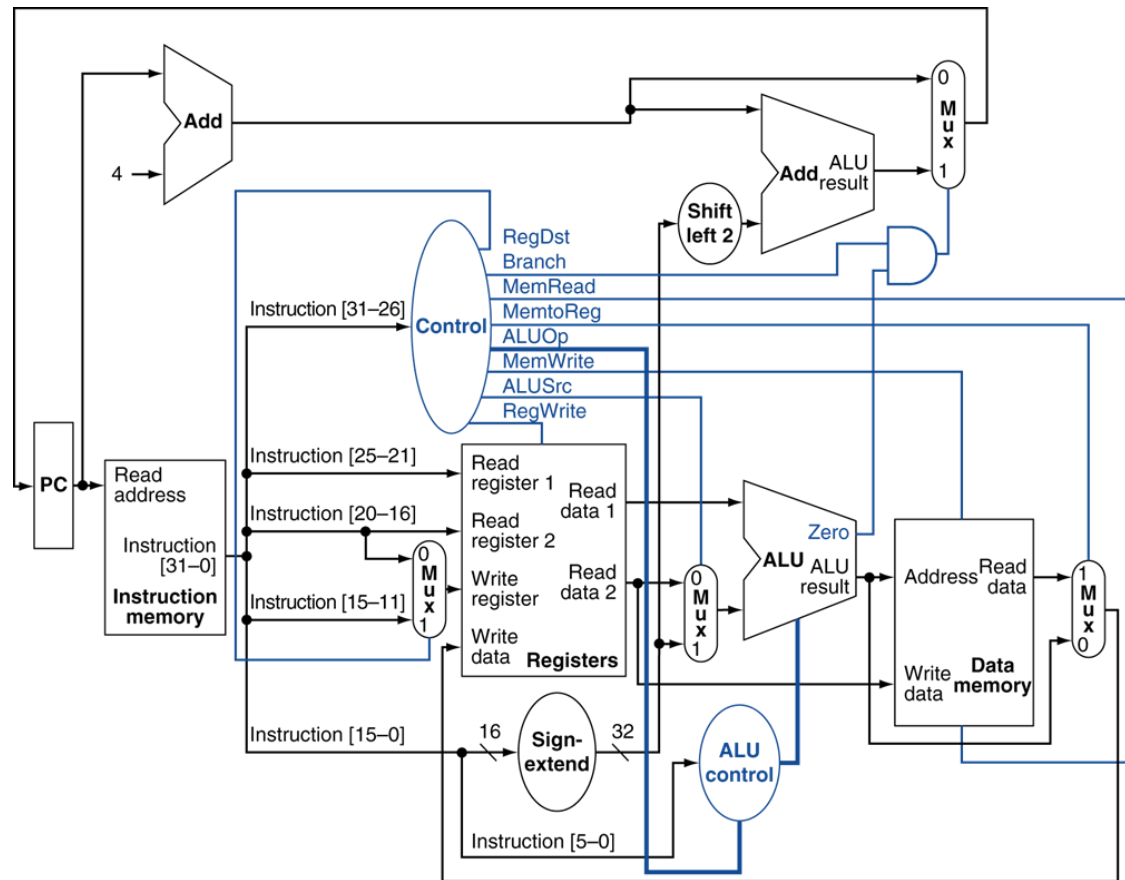




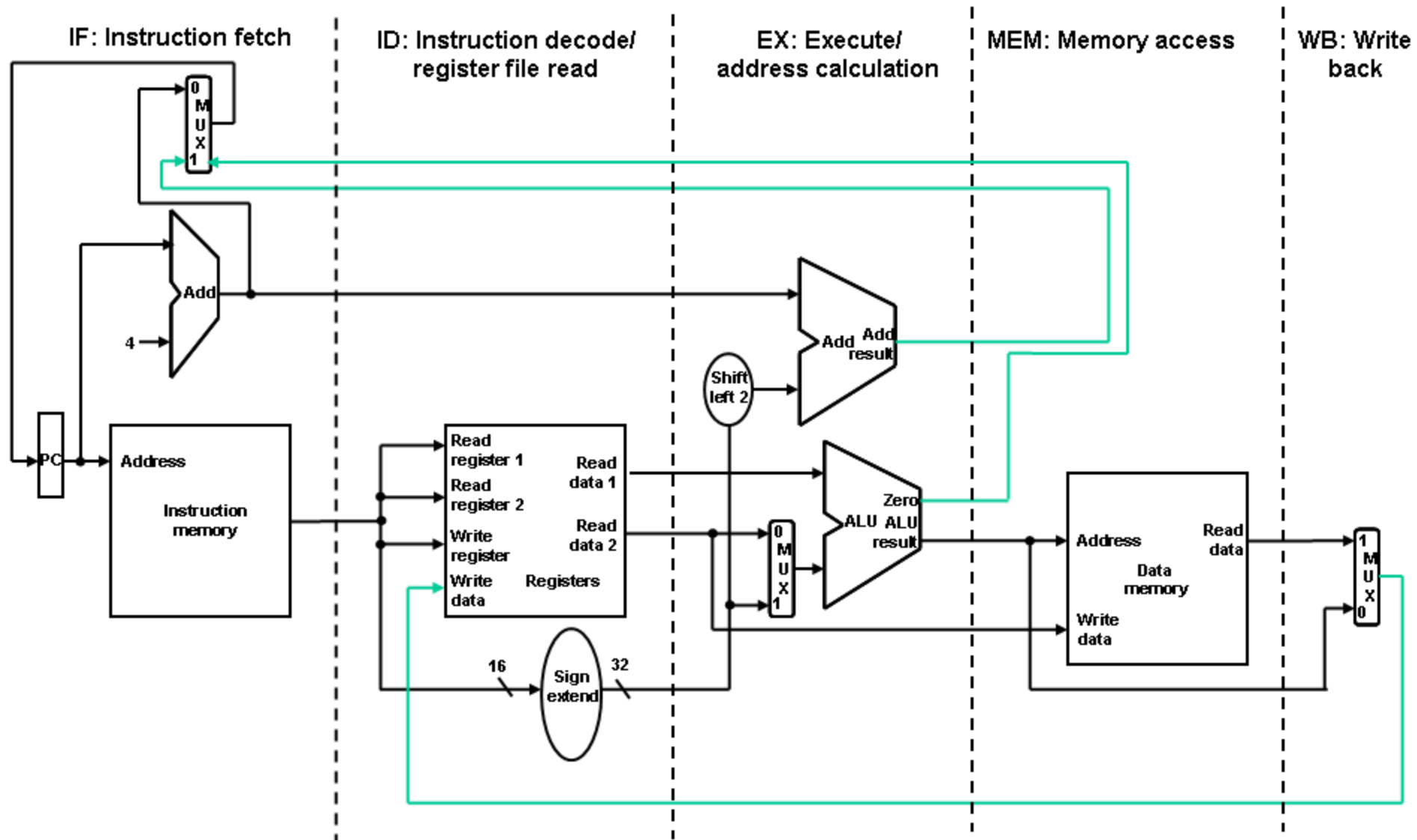
# Partition the SC datapath

- Try to make data flow in the same direction

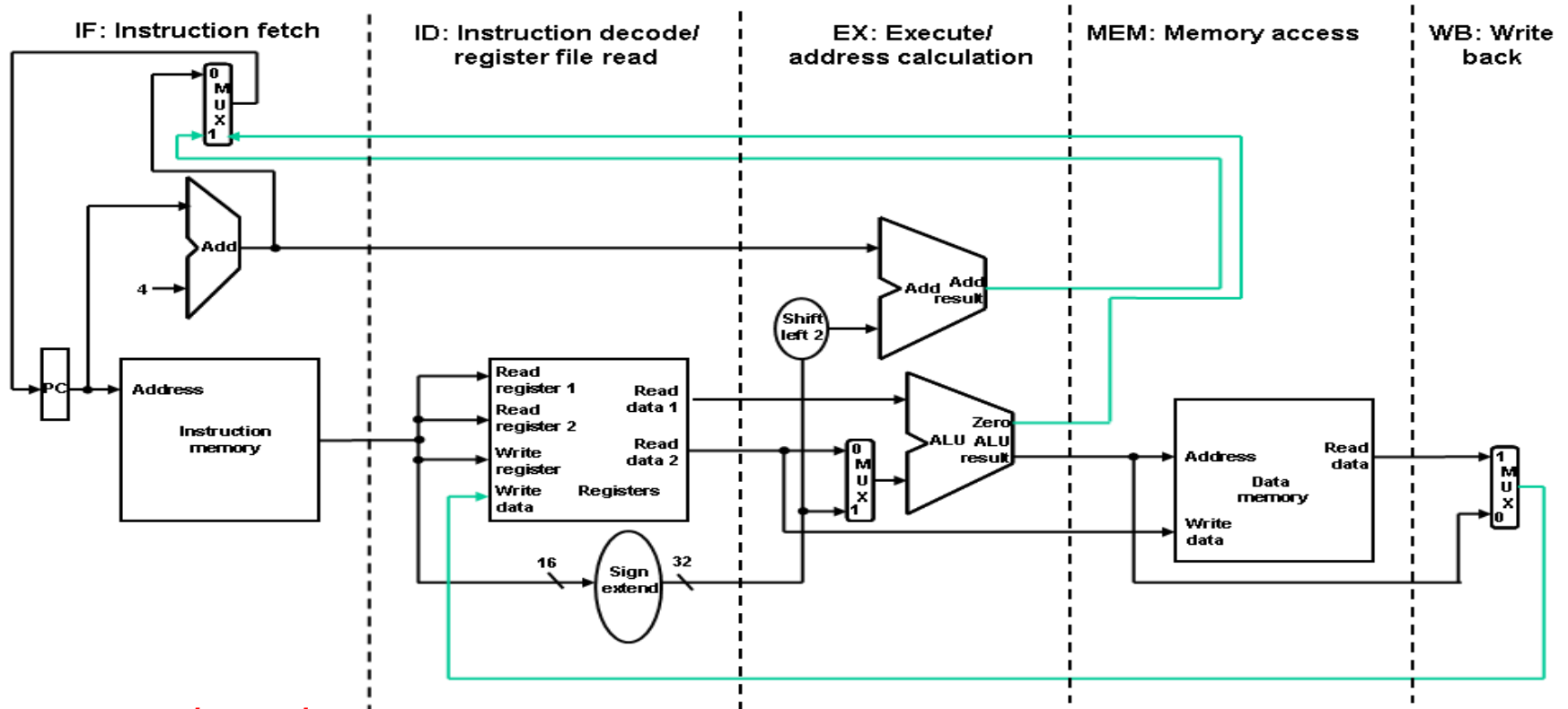
IF → ID → Ex → Mem → WB ?



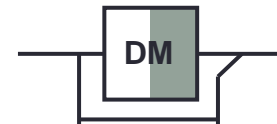
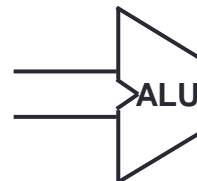
# Partitioned five stages



# Stage representation



*component based*

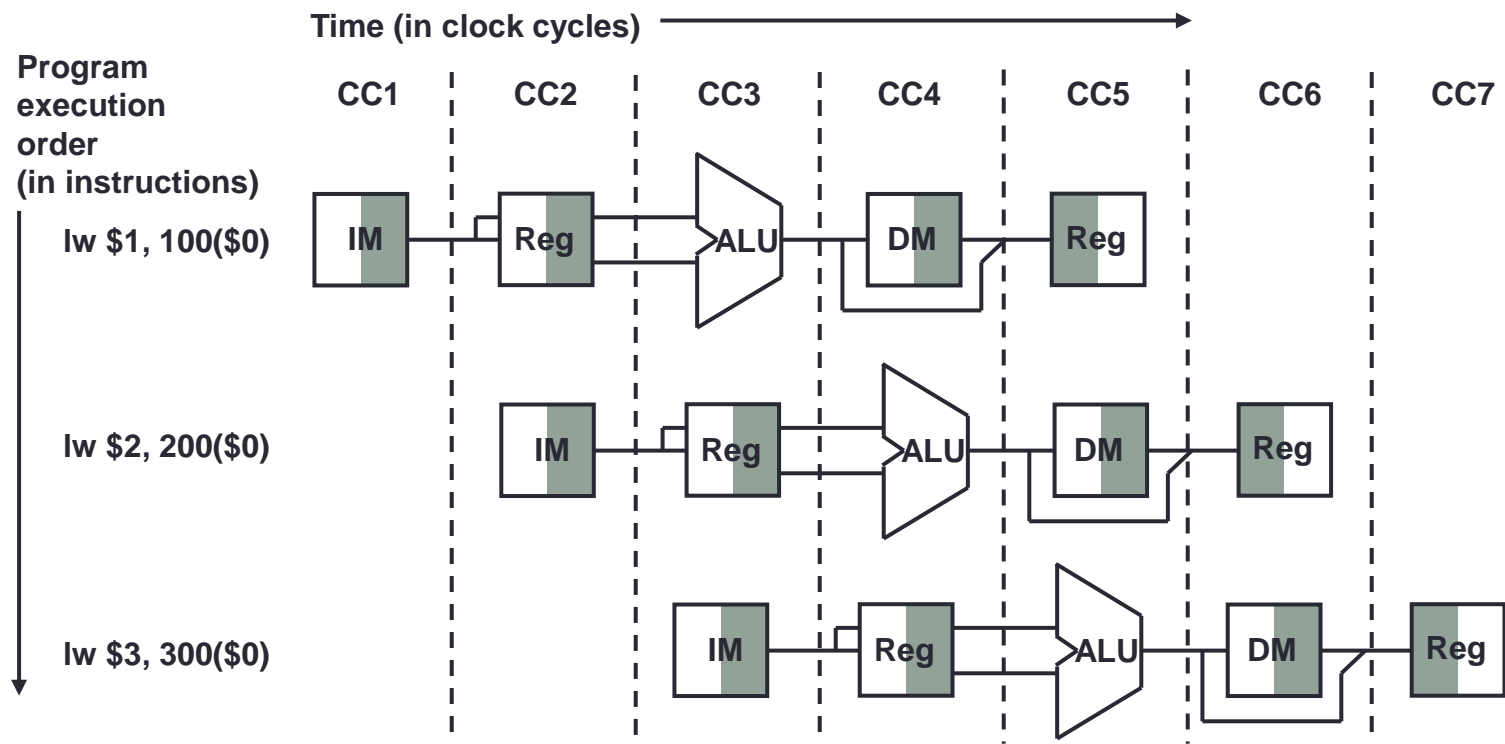


*operation based*



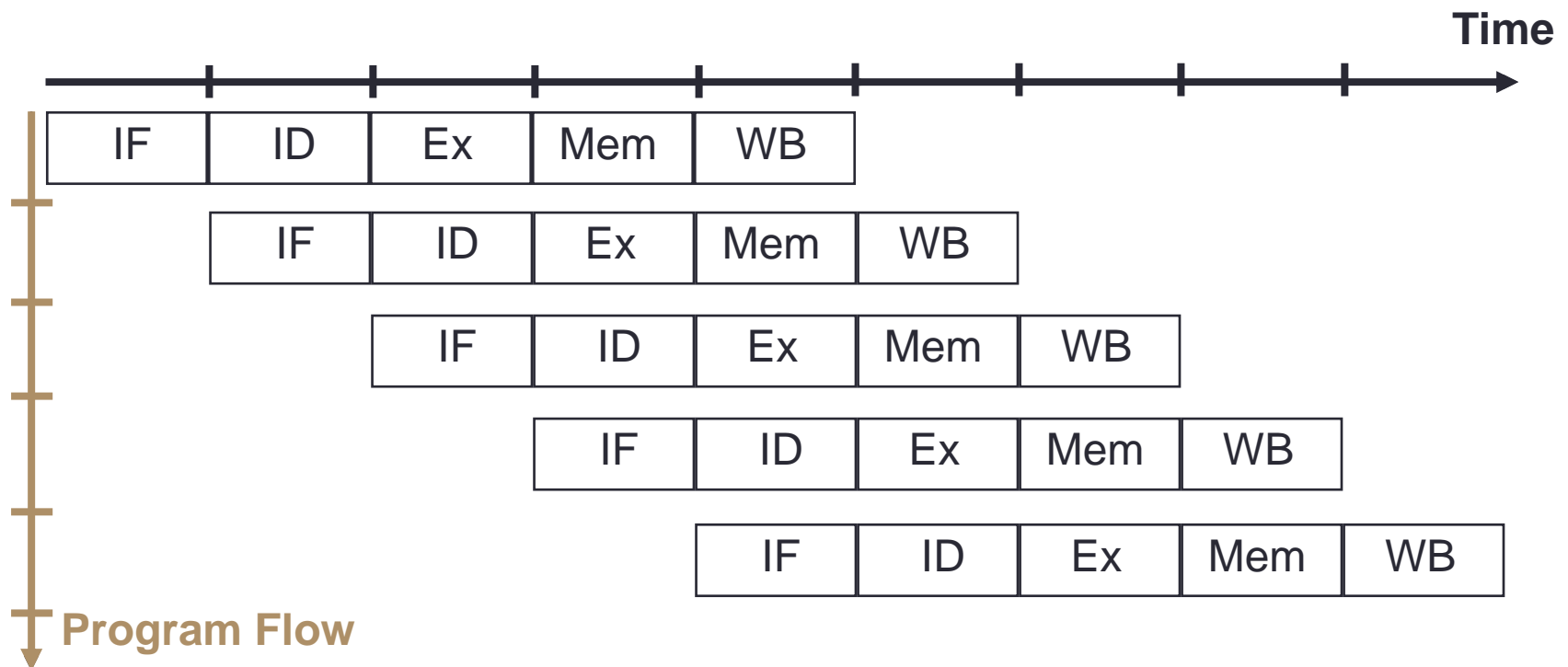
# Execution timing diagram

- **Example 1: using major component to represent each stage**
  - The **right half** of a register (memory) is highlighted as it is **read** and the **left half** is highlighted as it is **written**



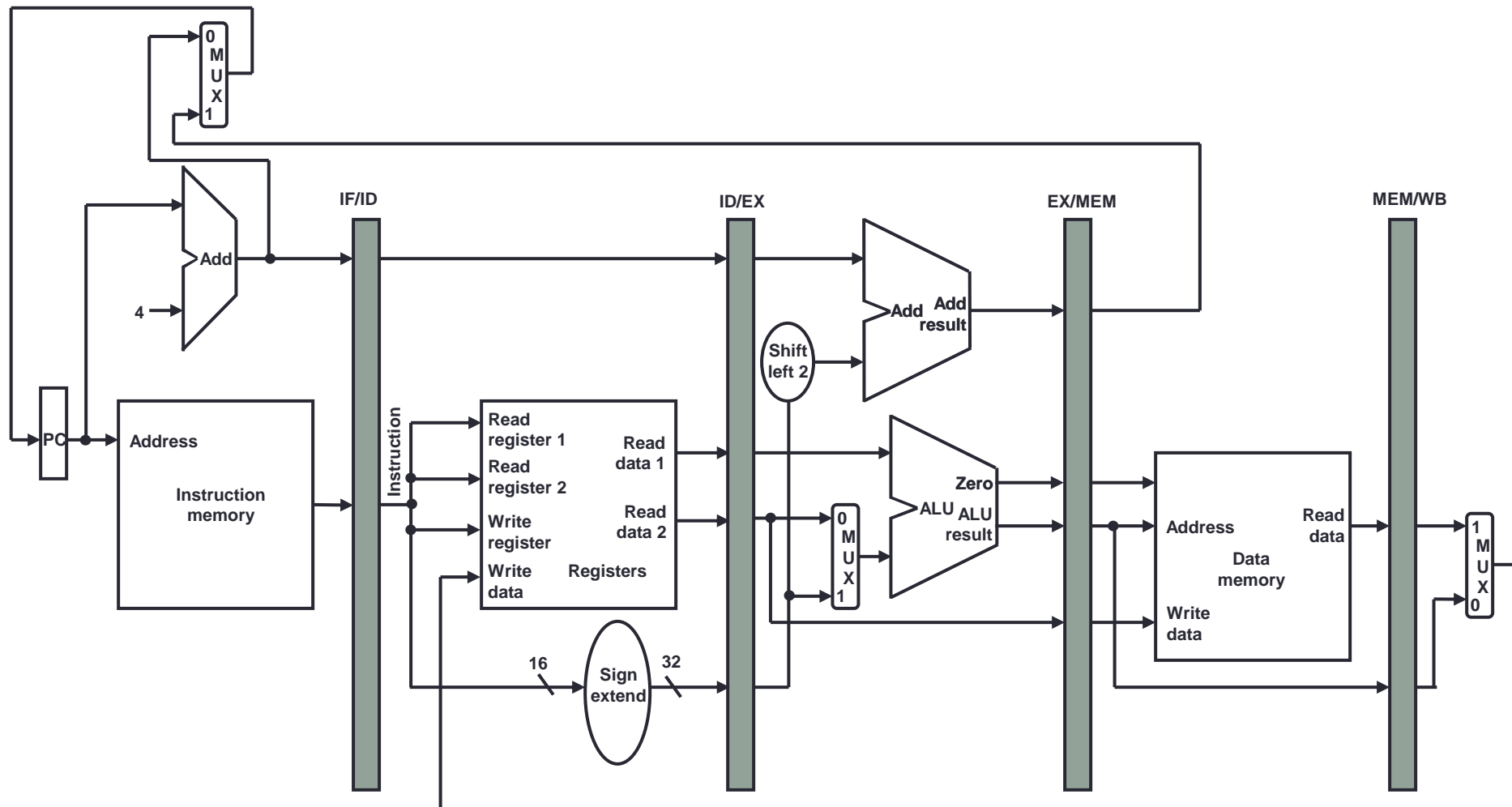
# Execution timing diagram

- Example 2: using operation to represent each stage**



# Create pipeline stages (2)

- To retain the values of an instruction to be used in other stages, the values must be saved in registers.



# Notes on the pipelined datapath

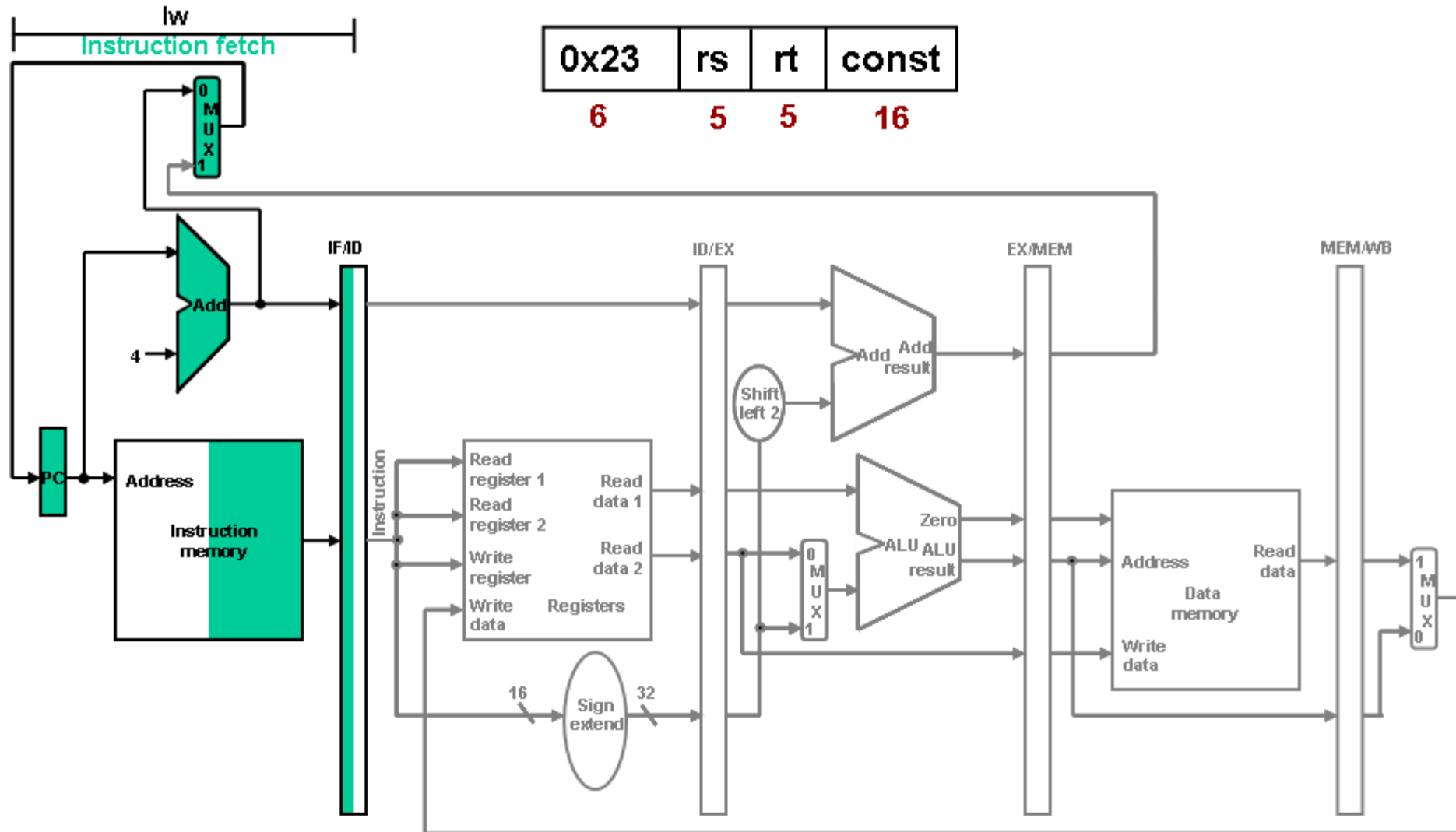
- The registers inserted between the pipeline stages are called pipeline stage register, or simply **pipeline register**.
- Each pipeline register is named after the two stages separated by that register
  - E.g. The register separating IF and ID stages is called the IF/ID register
- All instructions advance from one stage to the next stage
  - in one clock cycle

# Notes on the pipelined datapath (cont.)

- The write back stage does not need a separate register since the register file is updated at the end of that stage
  - A separate register would be redundant.
- An example of pipeline execution is given in the following slides.

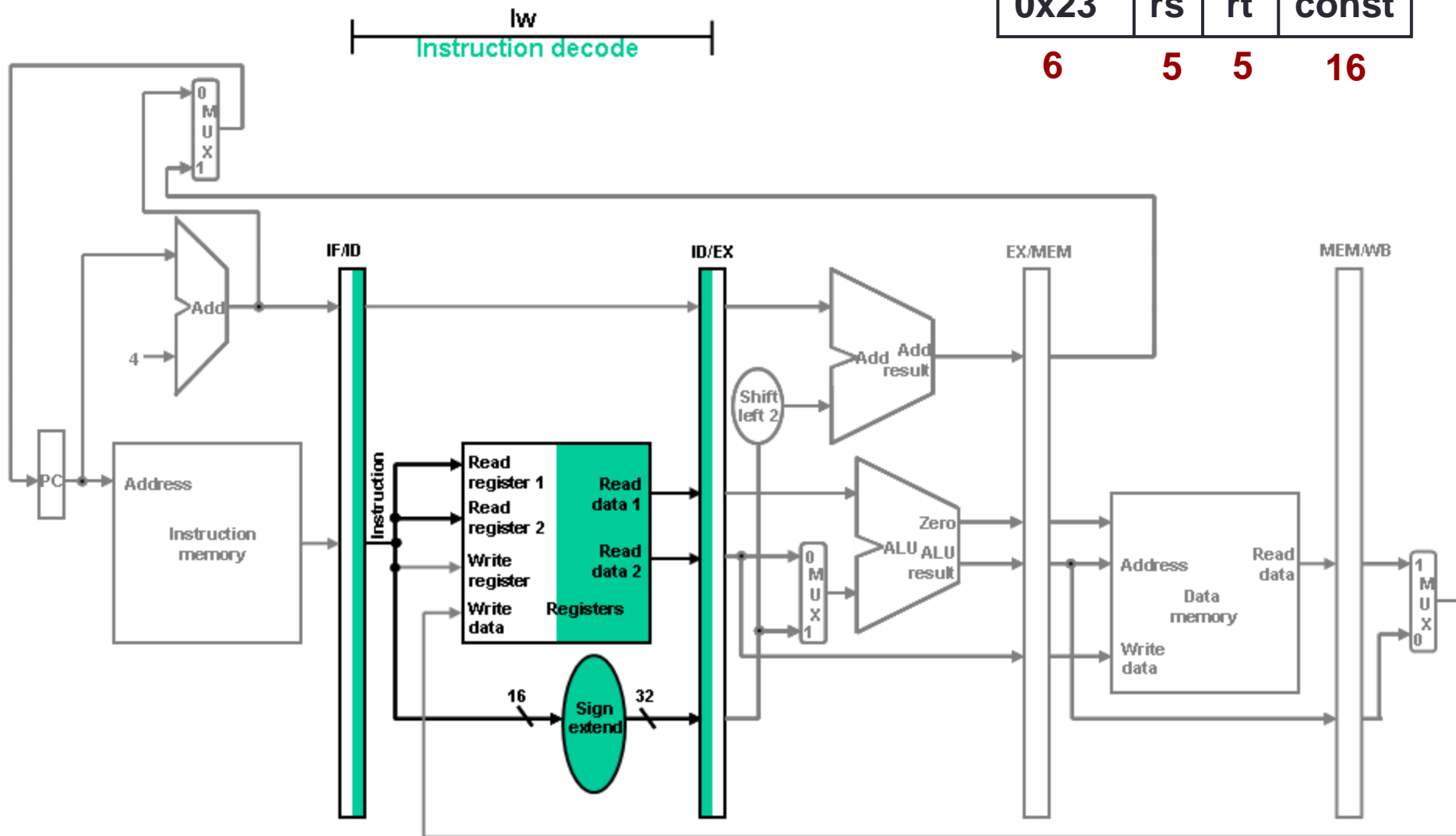


# Pipelined **LW** execution: **IF** stage

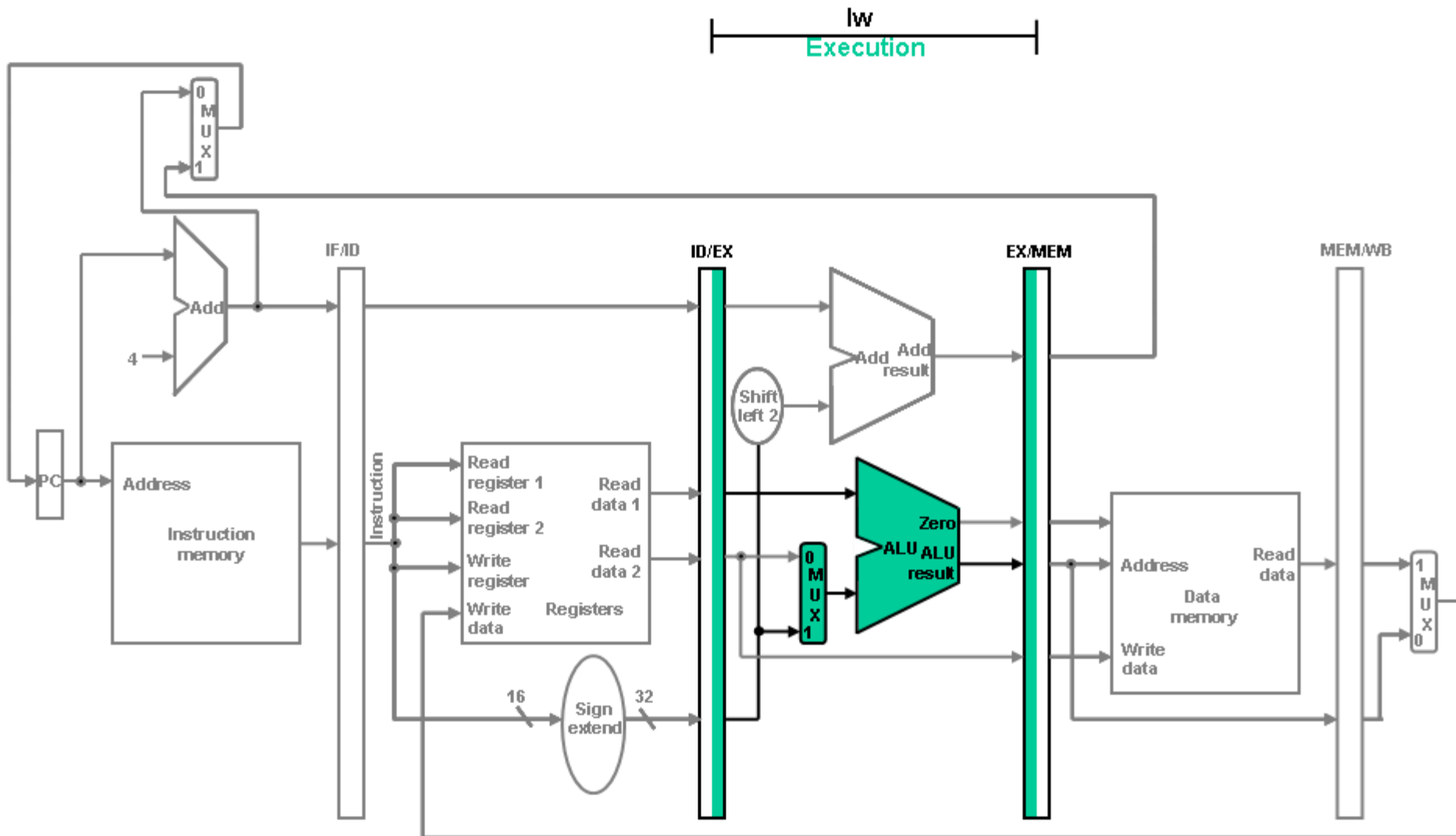


# Pipelined **LW** execution: ID stage

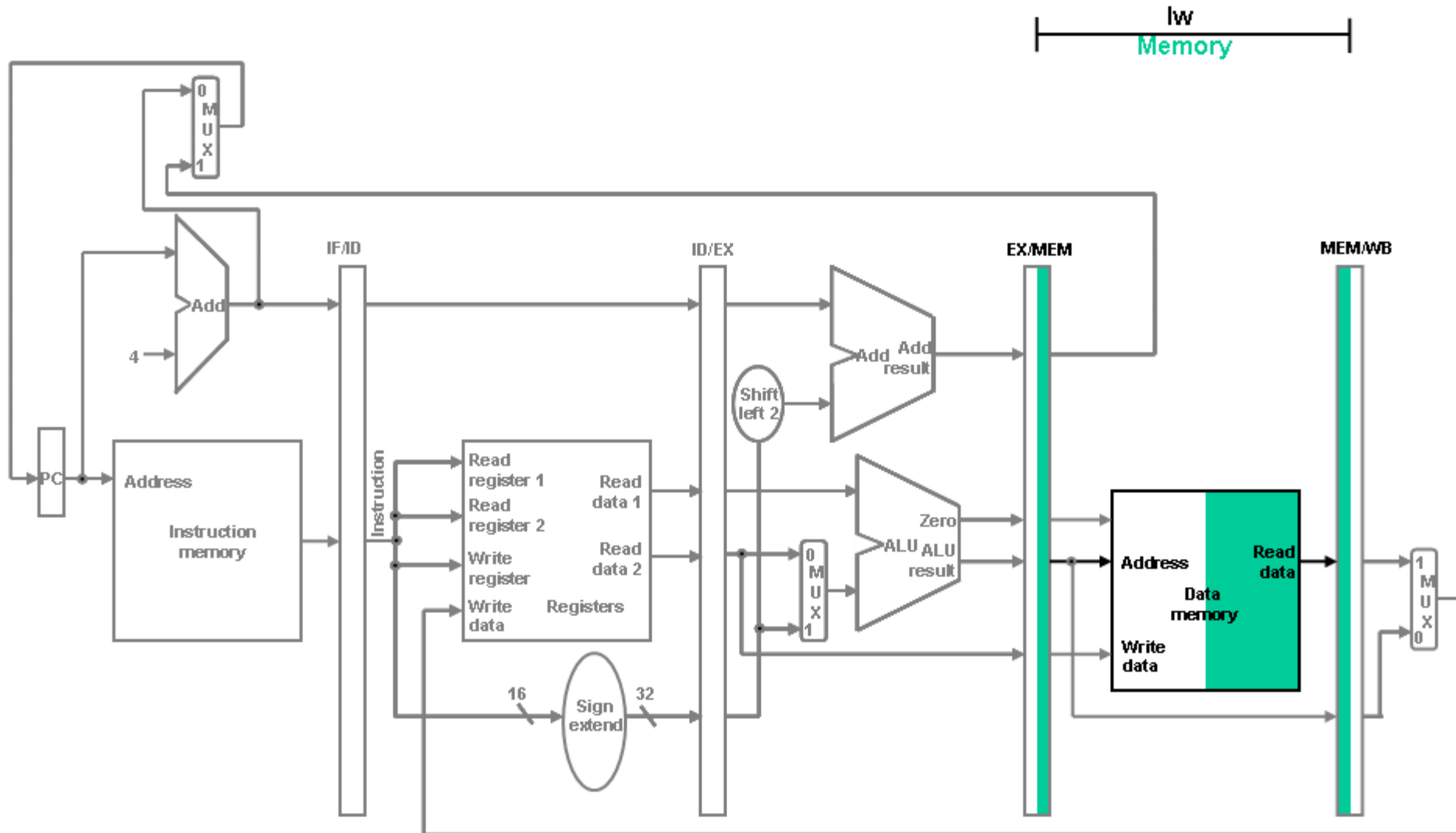
0x23	rs	rt	const
6	5	5	16



# Pipelined **LW** execution: **Ex** stage

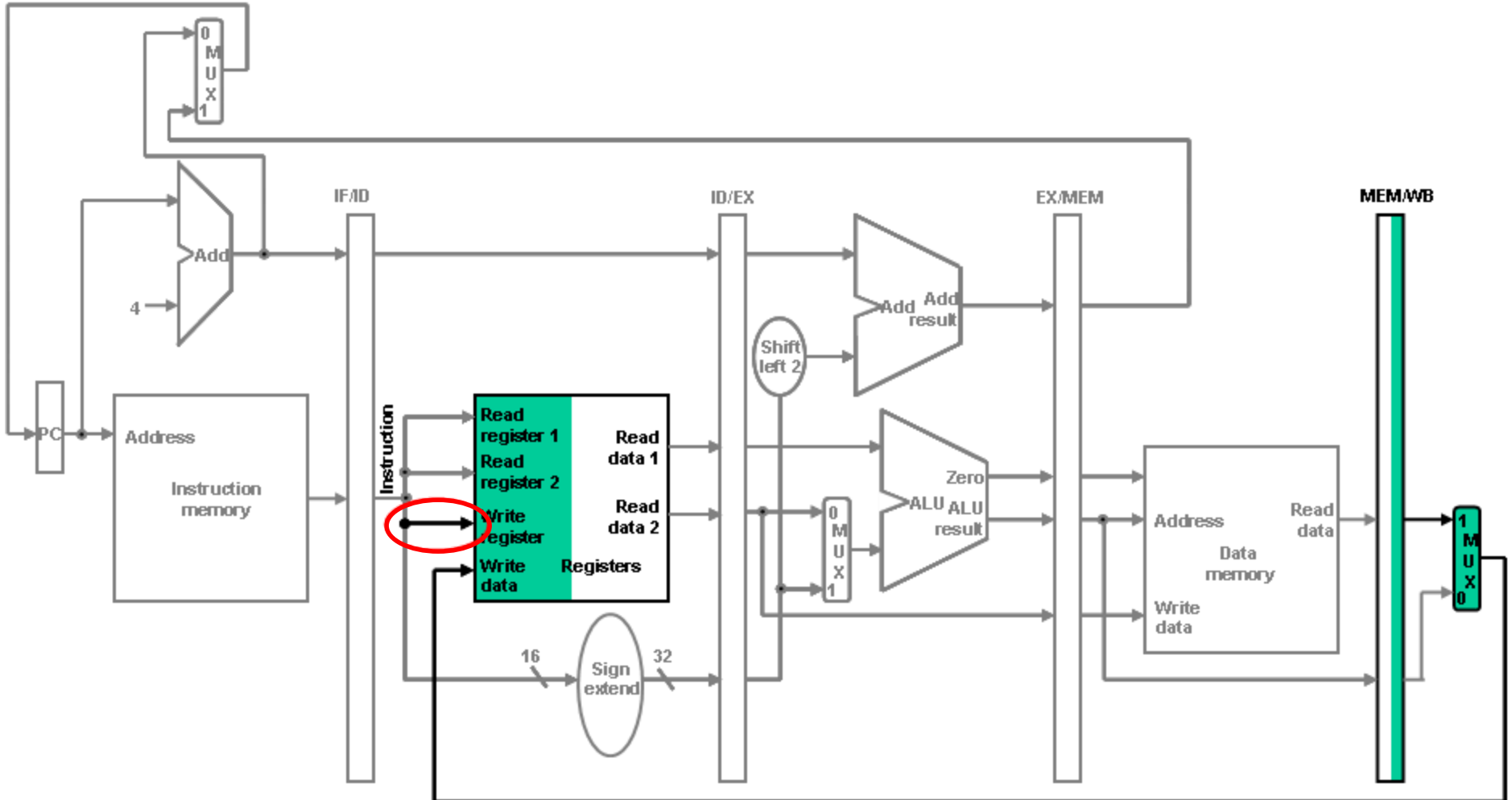


# Pipelined **LW** execution: **Mem** stage



# Pipelined **LW** Execution: **WB** stage

lw  
Write back



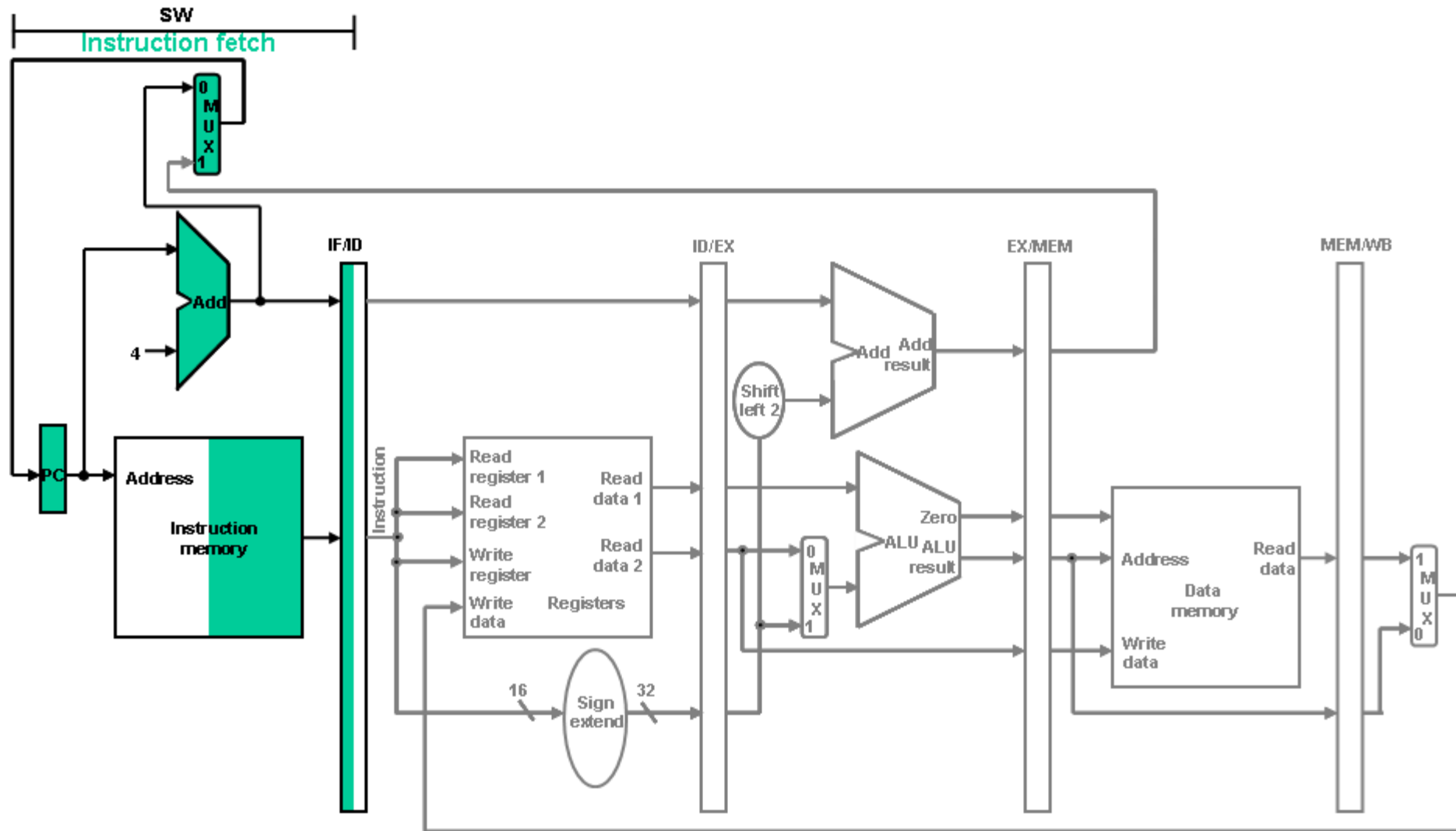
# Notes on pipelined **LW** execution

- The previous five slides show the active portions of the datapath as a load instruction progresses through the pipeline
- The operations of the five stages are as follows:
  - *Instruction fetch*
    - The instruction is read from memory pointed by PC and is stored in IF/ID. PC is incremented by 4 and written back ready for the next instruction. The new PC value is stored in IF/ID for instructions such as BEQ.
    - We store all data that may be needed by subsequent stages.

# Notes on pipelined **LW** execution (cont.)

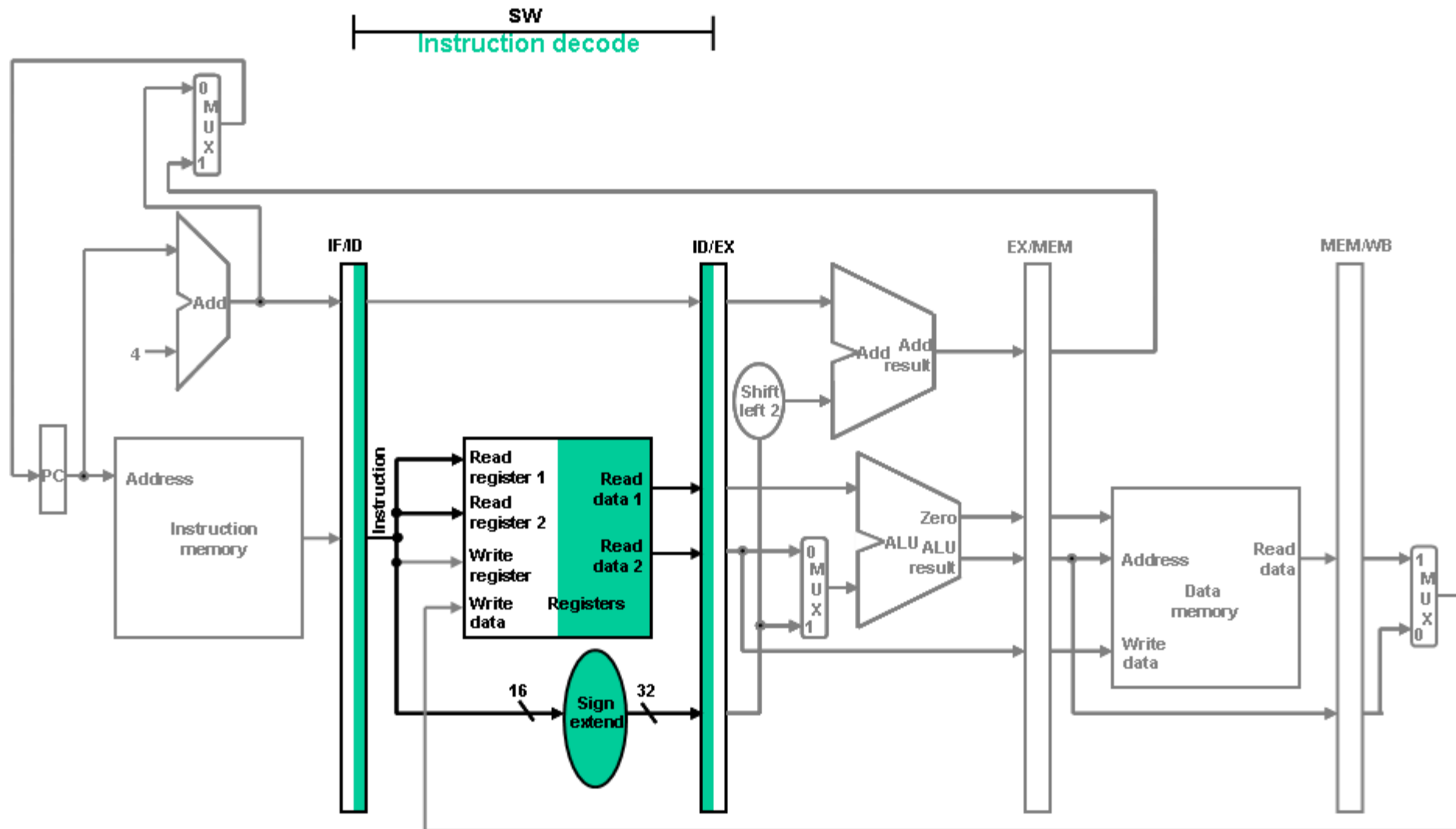
- *Instruction decode and register file read*
  - The immediate field is retrieved from IF/ID and sign extended;
  - The two source registers are read;
  - All three values are stored in ID/EX along with everything else that may be needed by the instruction in the next clock cycle.
- *Execution or address calculation*
  - The value in register 1 and the sign-extended immediate value from ID/EX are added by ALU and the result is placed in the EX/MEM register.
- *Memory access*
  - Data memory is read using the address from EX/MEM; The read data is loaded into the MEM/WB register.
- *Write back*
  - The data from MEM/WB is written to the register file.

# Pipelined SW execution: IF stage

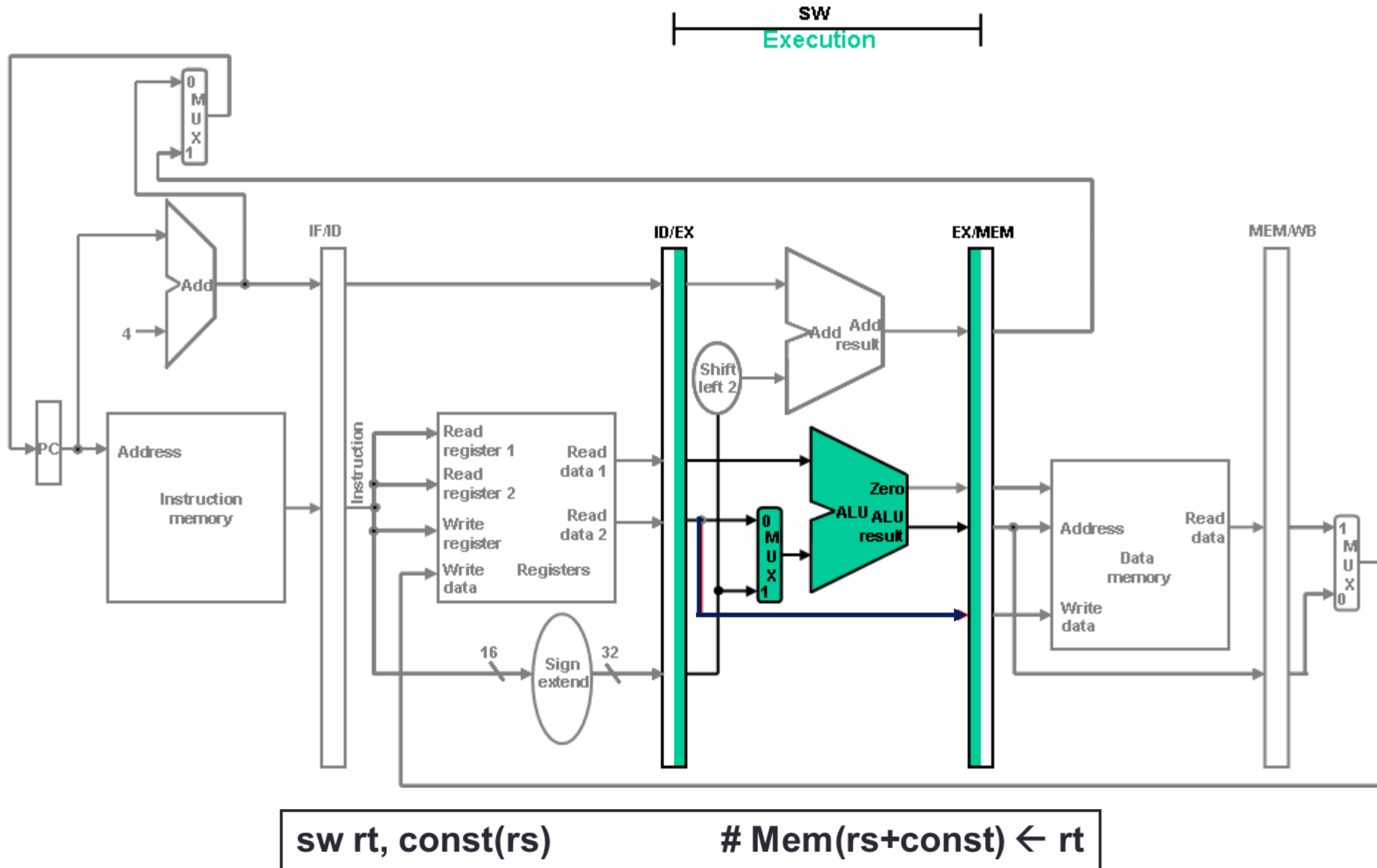




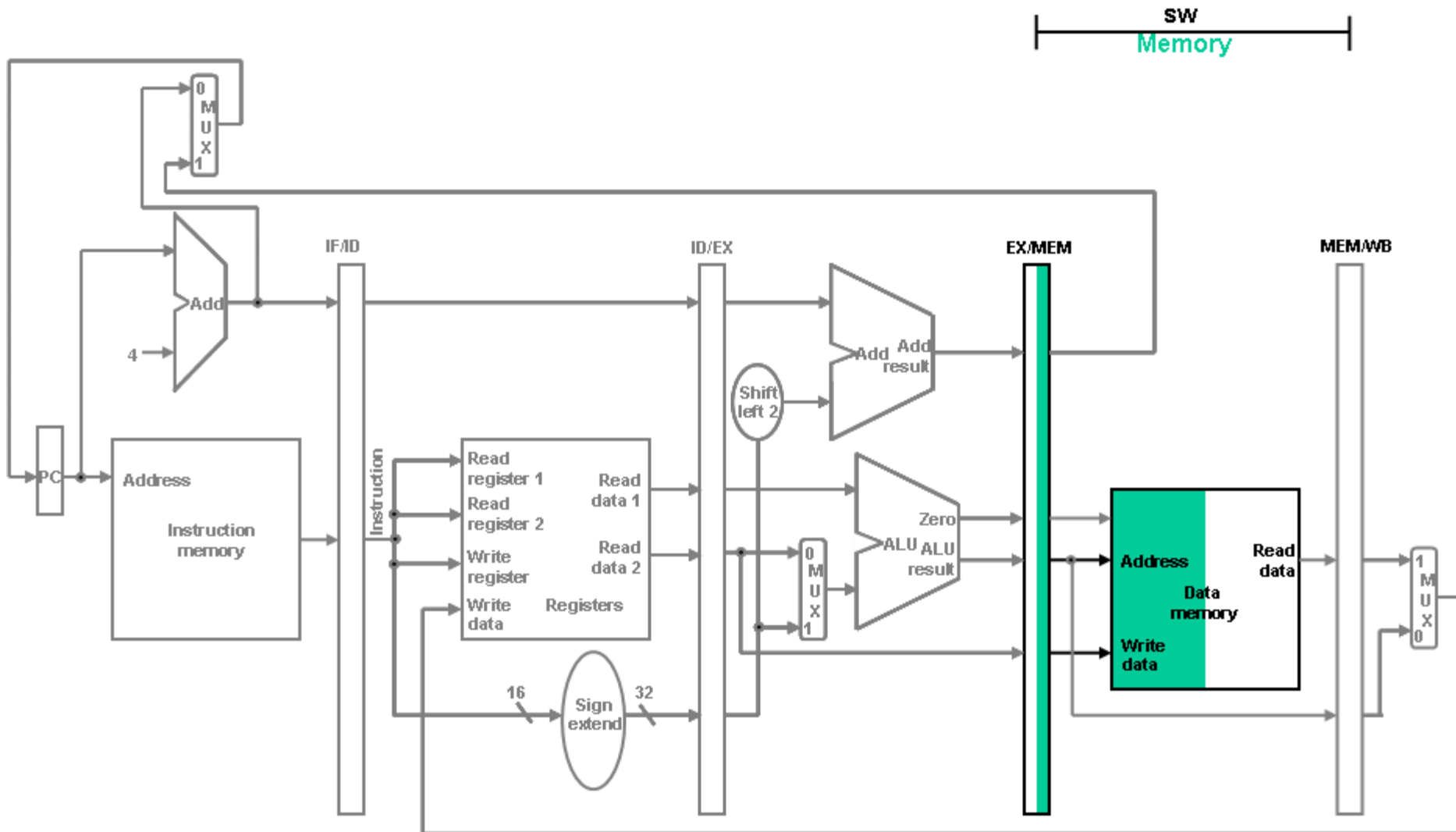
# Pipelined **SW** execution: **ID** stage



# Pipelined **SW** execution: **Ex** stage

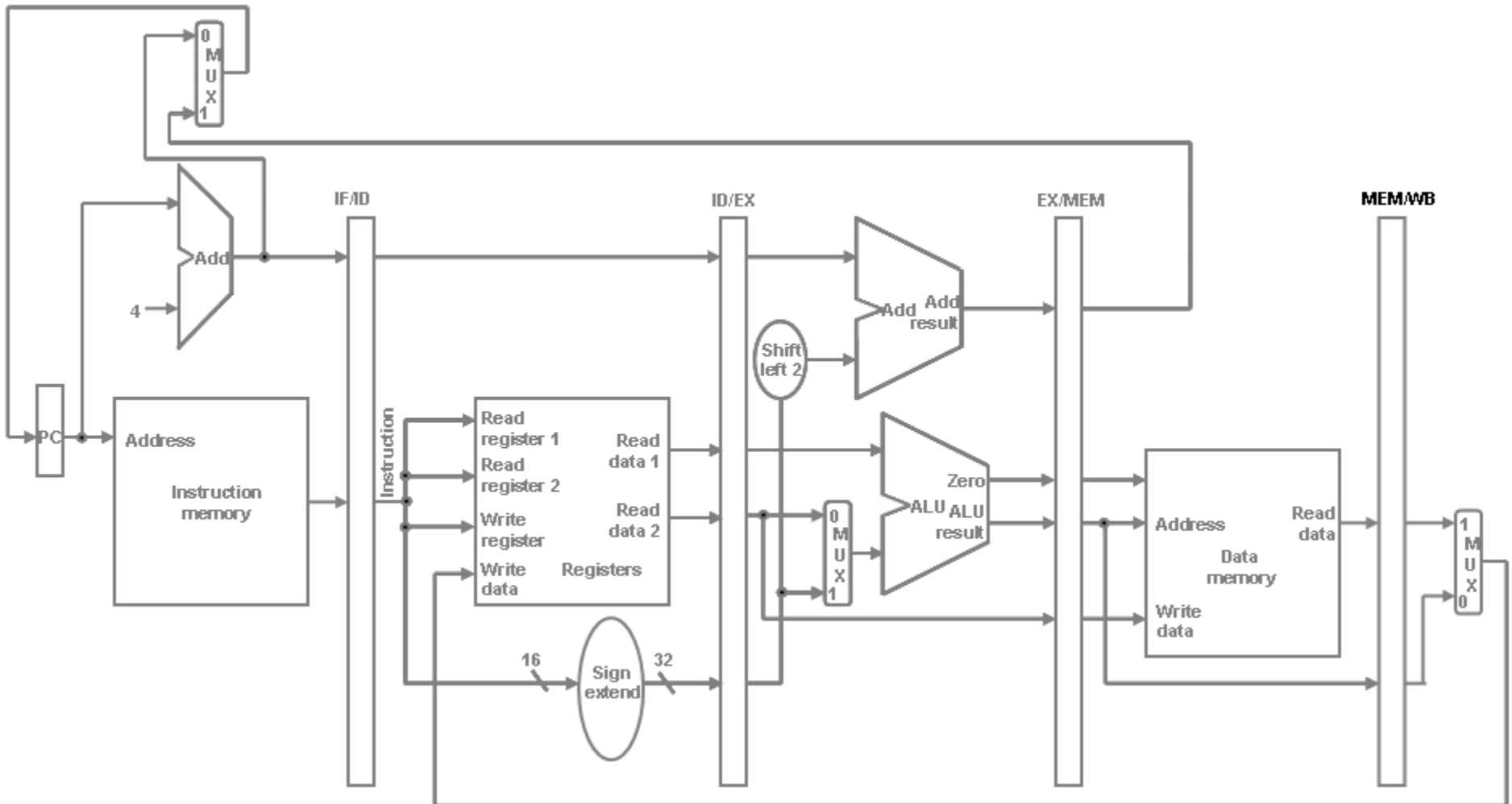


# Pipelined **SW** execution: **Mem** stage



# Pipelined **SW** execution: **WB** stage

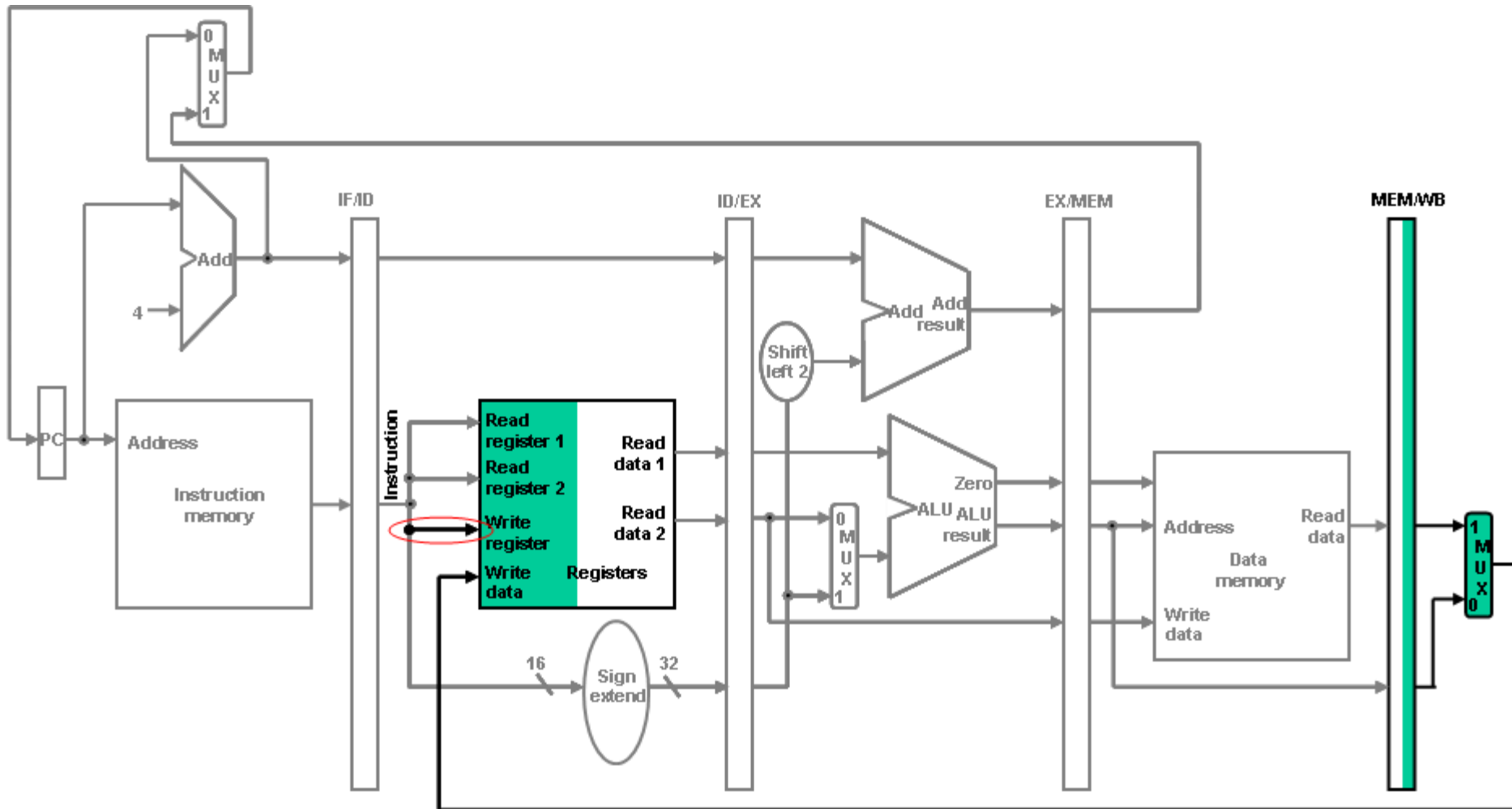
Write back



# Notes on **SW** instruction execution

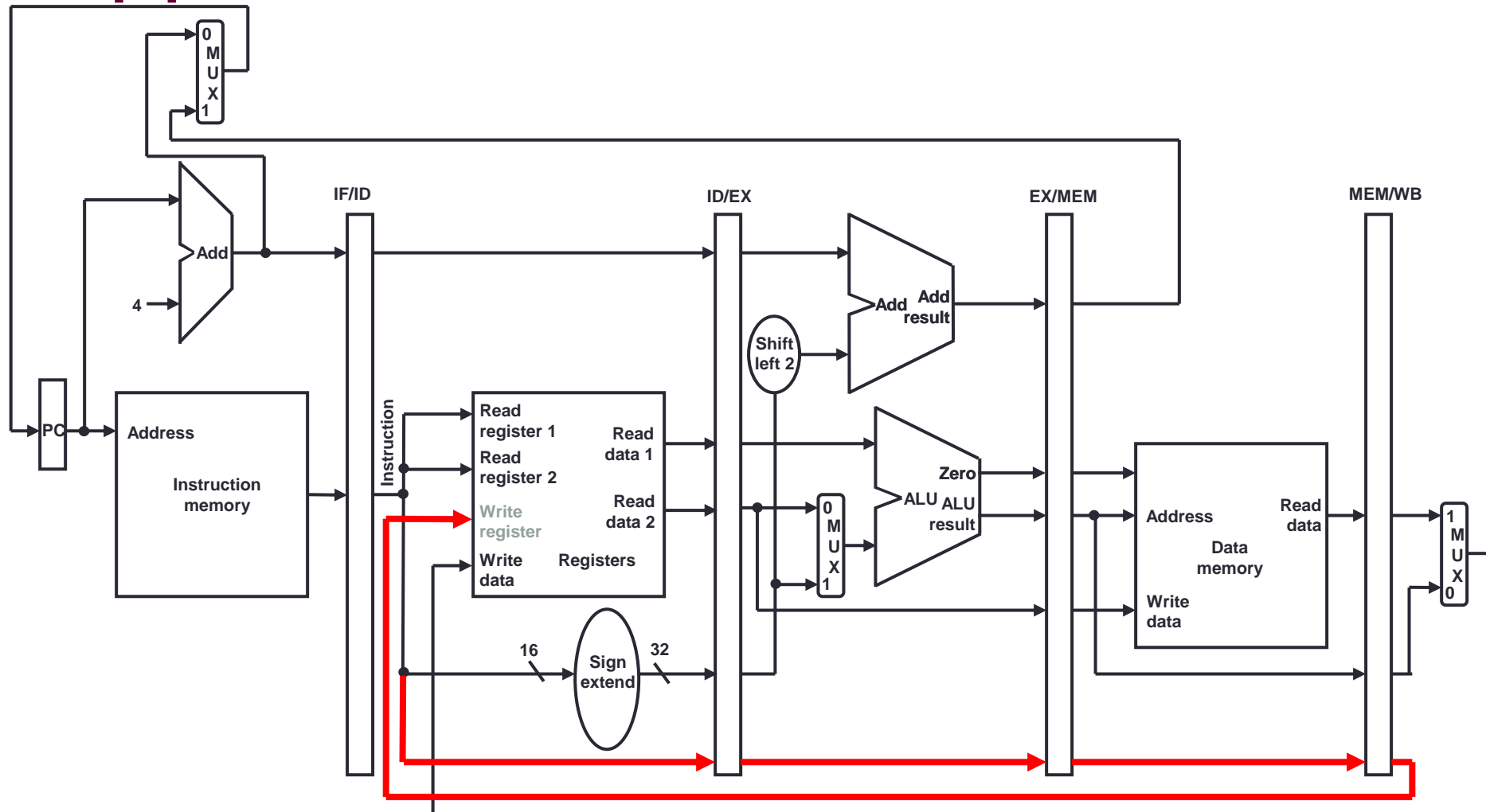
- **The first three stages are identical to those for the load instruction, but the last two stages operate differently**
- ***Memory access***
  - The data is written to memory. In order to make the data available during the Mem stage it had to be placed into the EX/MEM register during the EX stage.
- ***Write back***
  - For this instruction, nothing happens in the write back stage.

# Bug in the write back stage?



# Corrected design

- Carry the destination register info through the pipeline as well!



# Key points

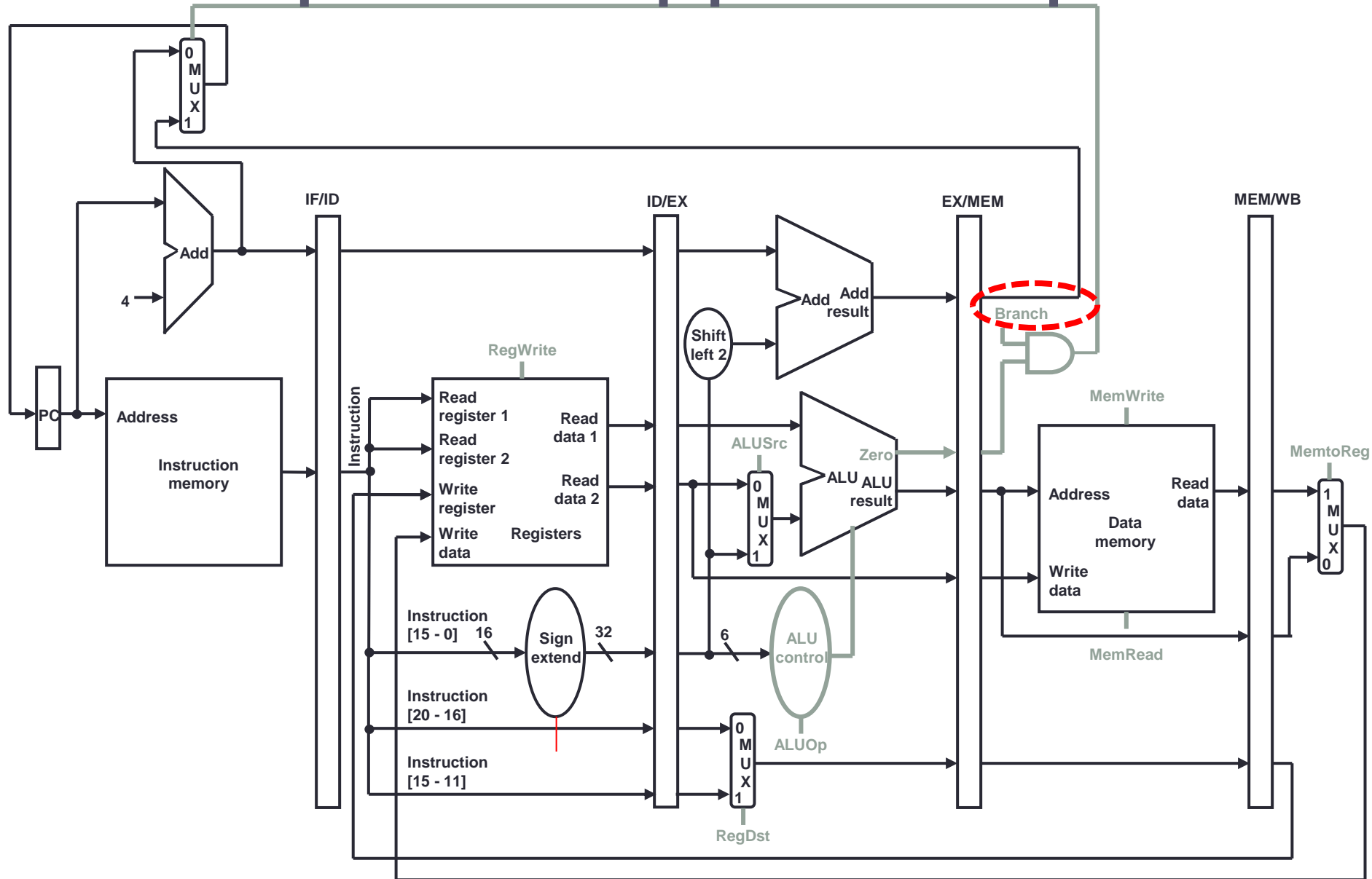
- To pass something from an early pipe stage to a later pipe stage, the information must be placed in the pipeline register; otherwise the information will be lost when the next instruction enters that pipeline stage.
- Each logical component of the datapath – such as instruction memory, register read ports, ALU, data memory, and register write port – can be used only within a *single* pipeline stage; otherwise we would have a ***structural hazard***.
  - To be covered later



# Pipelined control

- **Add control to the pipelined datapath just like we did to the single cycle datapath:**
  1. **Label control points**
  2. **Determine the control settings for each stage for all instructions**
  3. **Design control logic to implement the control**
    - **To handle sequencing, control inputs are pipelined together with data and intermediate results**

# Control points in the pipelined datapath

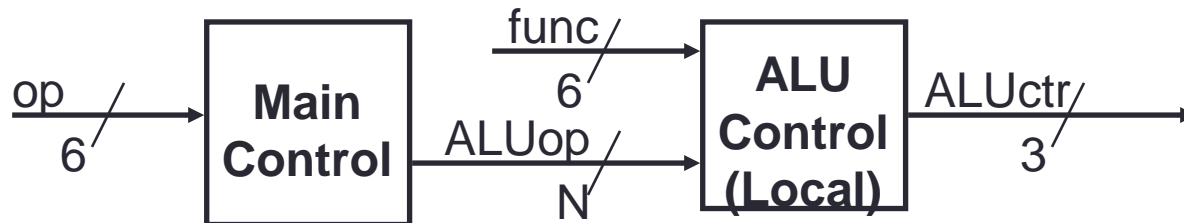


# Notes on pipeline control

- Since each control signal is associated with a component that is active only in one pipeline stage, we group control signals for stages
  - **IF stage**
    - No control signals needed
  - **ID stage**
    - ExtOp
  - **Ex stage**
    - RegDst, ALUOp, and ALUSrc
  - **Mem stage**
    - Branch, MemRead, and MemWrite
  - **Write back**
    - MemtoReg and RegWrite

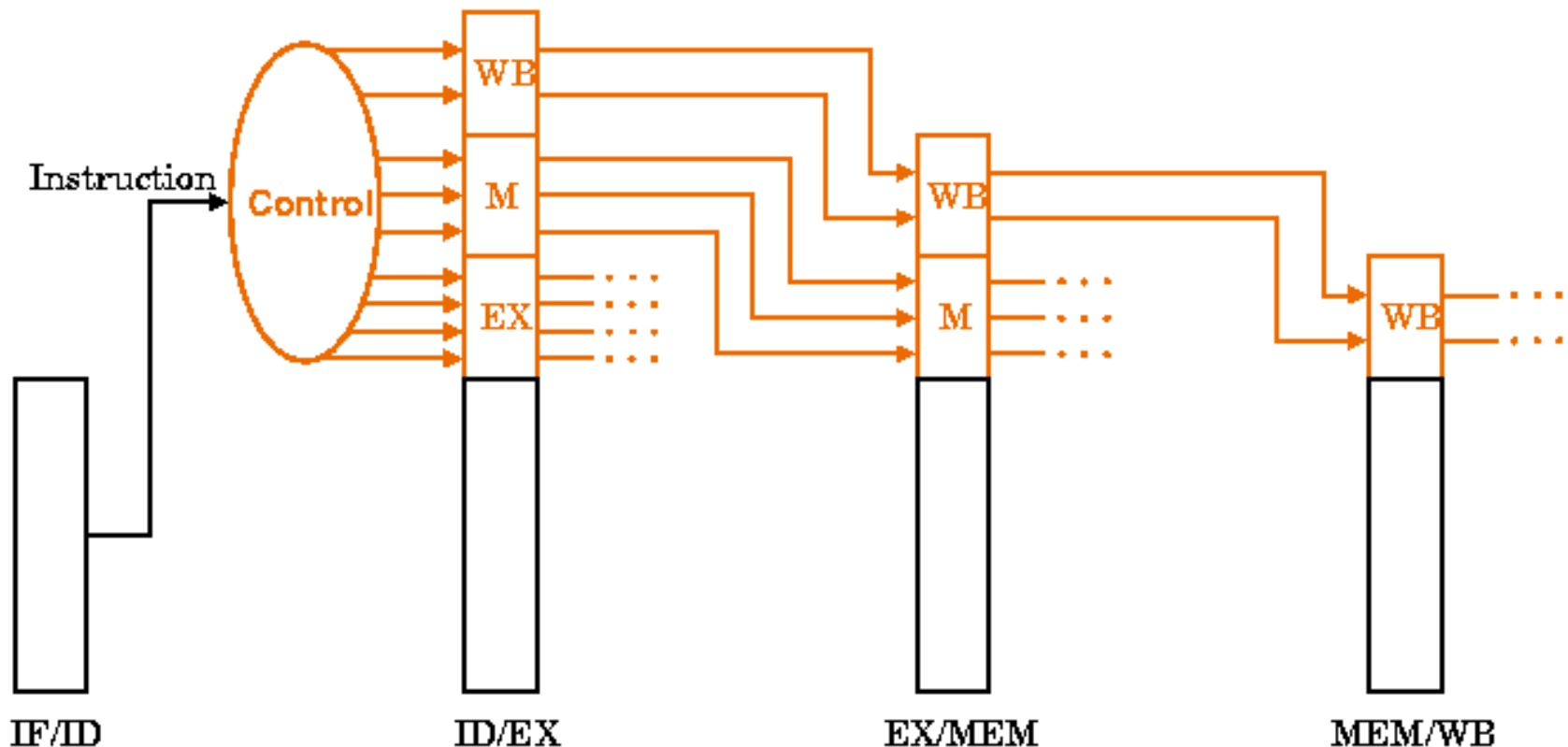
# Recall: Control Unit of single cycle processor

- We can use the same design for pipelined datapath

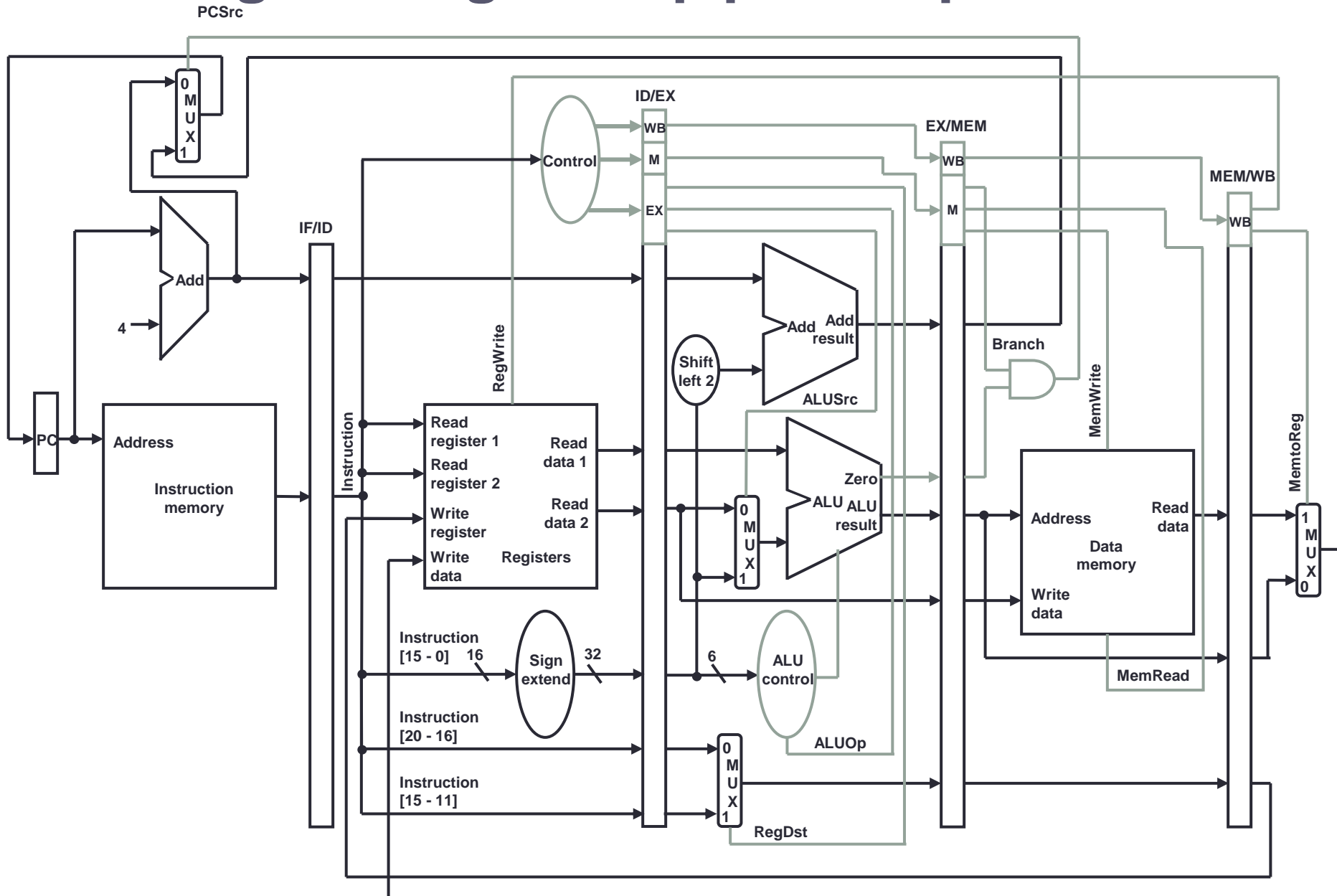


# Pipelining the control signals

- The control signals are generated in ID stage
- These control signals are then used in the following stages as the instruction moves down the pipeline



# Putting it all together: pipelined processor



# Example of pipeline execution

- **How are the following instructions executed?**

```
lw  $10, 20($1)
sub $11, $2, $3
and $12, $4, $5
or  $13, $6, $7
add $14, $8, $9
```

# Execution example: clock cycle 1/9

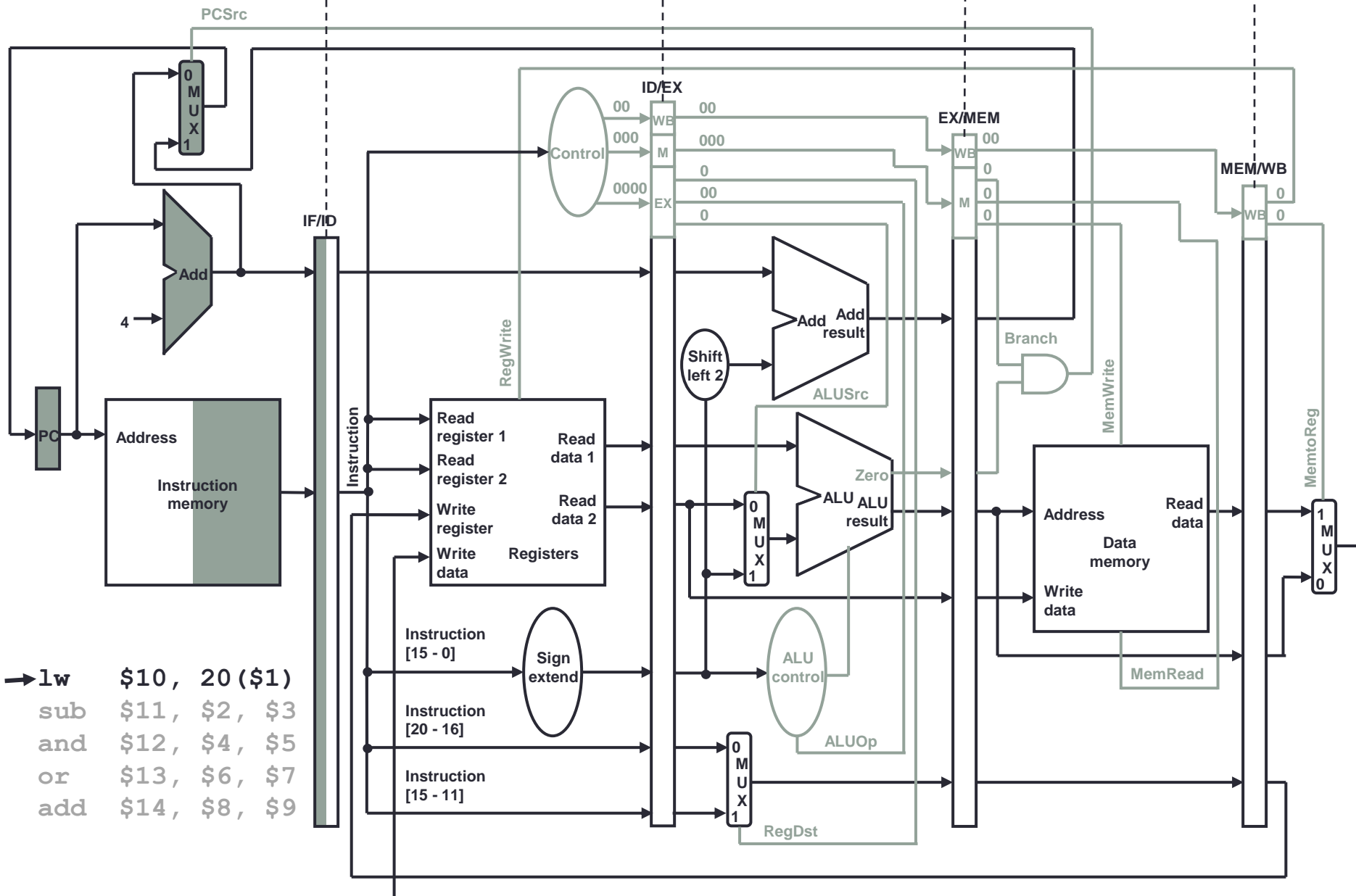
IF: lw \$10, 20(\$1)

ID: before &lt;1&gt;

EX: before &lt;2&gt;

MEM: before &lt;3&gt;

WB: b &lt;4&gt;





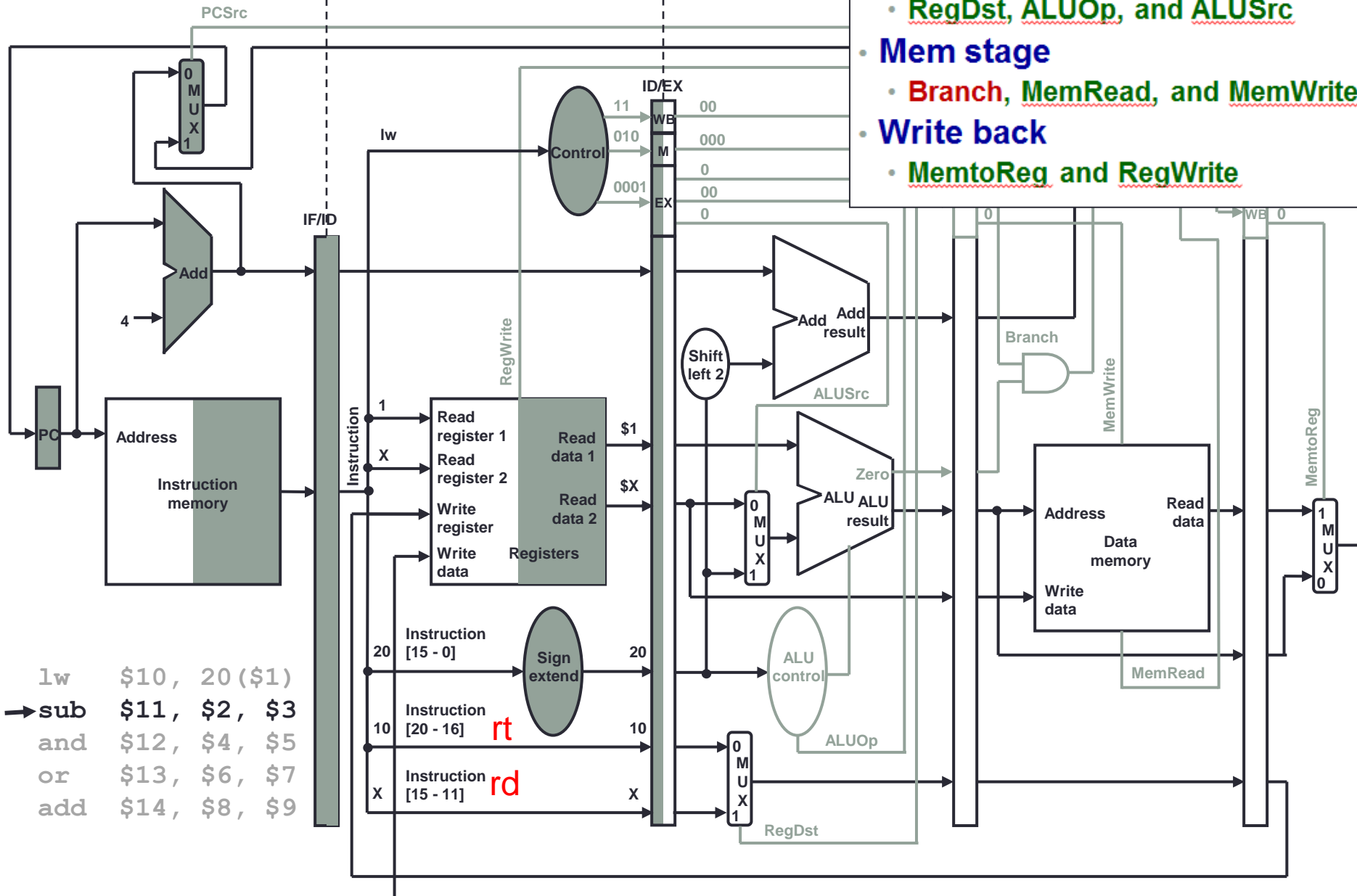
# Execution example: clock cycle

IF: sub \$11, \$2, \$3

ID: lw \$10, 20(\$1)

EX: before &lt;1&gt;

- **IF stage and ID stage**
  - No control signals needed.
- **Ex stage**
  - RegDst, ALUOp, and ALUSrc
- **Mem stage**
  - Branch, MemRead, and MemWrite
- **Write back**
  - MemtoReg and RegWrite



COMP3211/COMP9211 Week 3-1

# Execution example: clock cycle 3/9

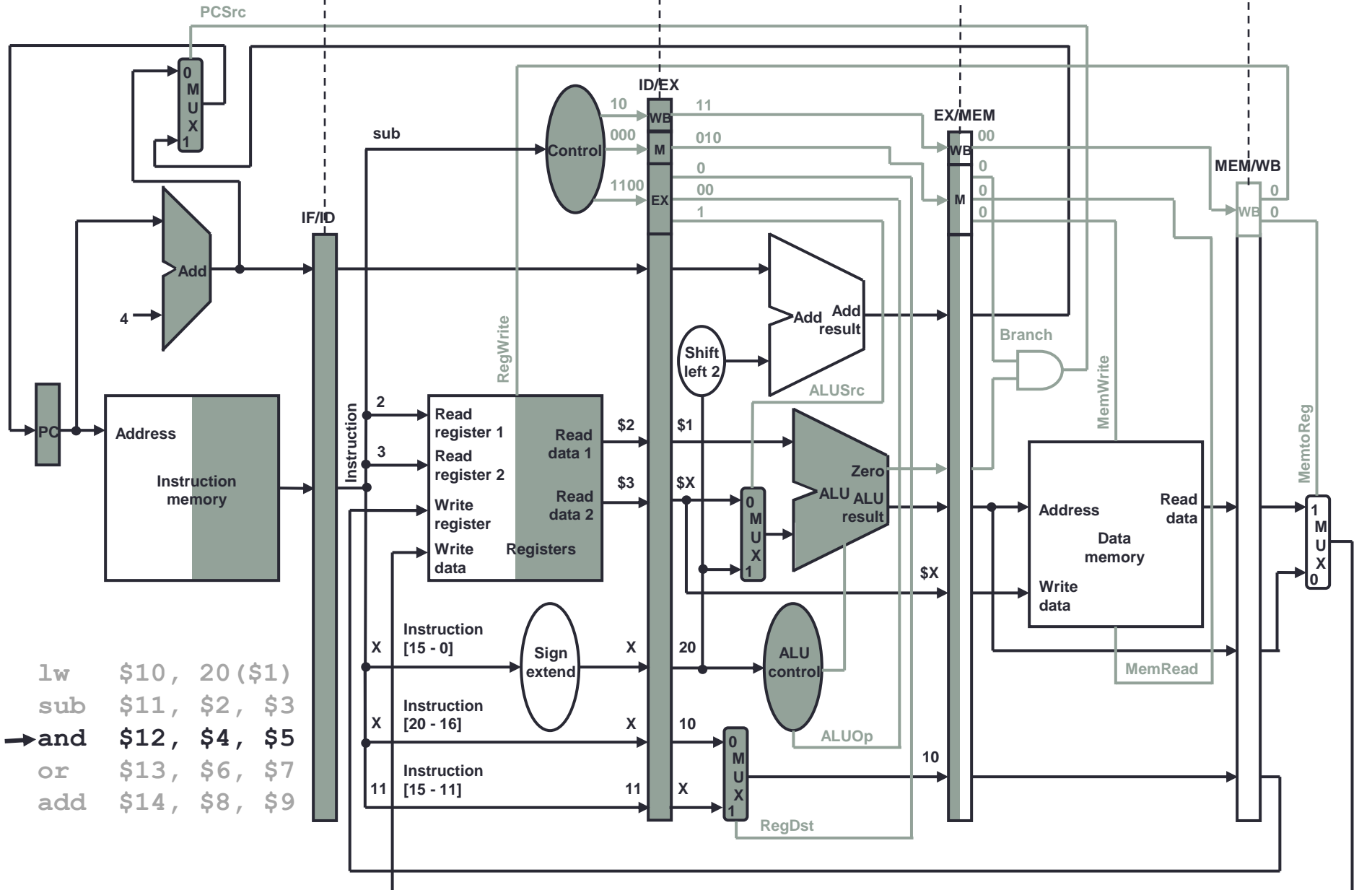
**IF: and \$12, \$4, \$5**

**ID: sub \$11, \$2, \$3**

**EX: lw \$10, 20(\$1)**

**MEM: before <1>**

**WB: b <2>**



# Execution example: clock cycle 4/9

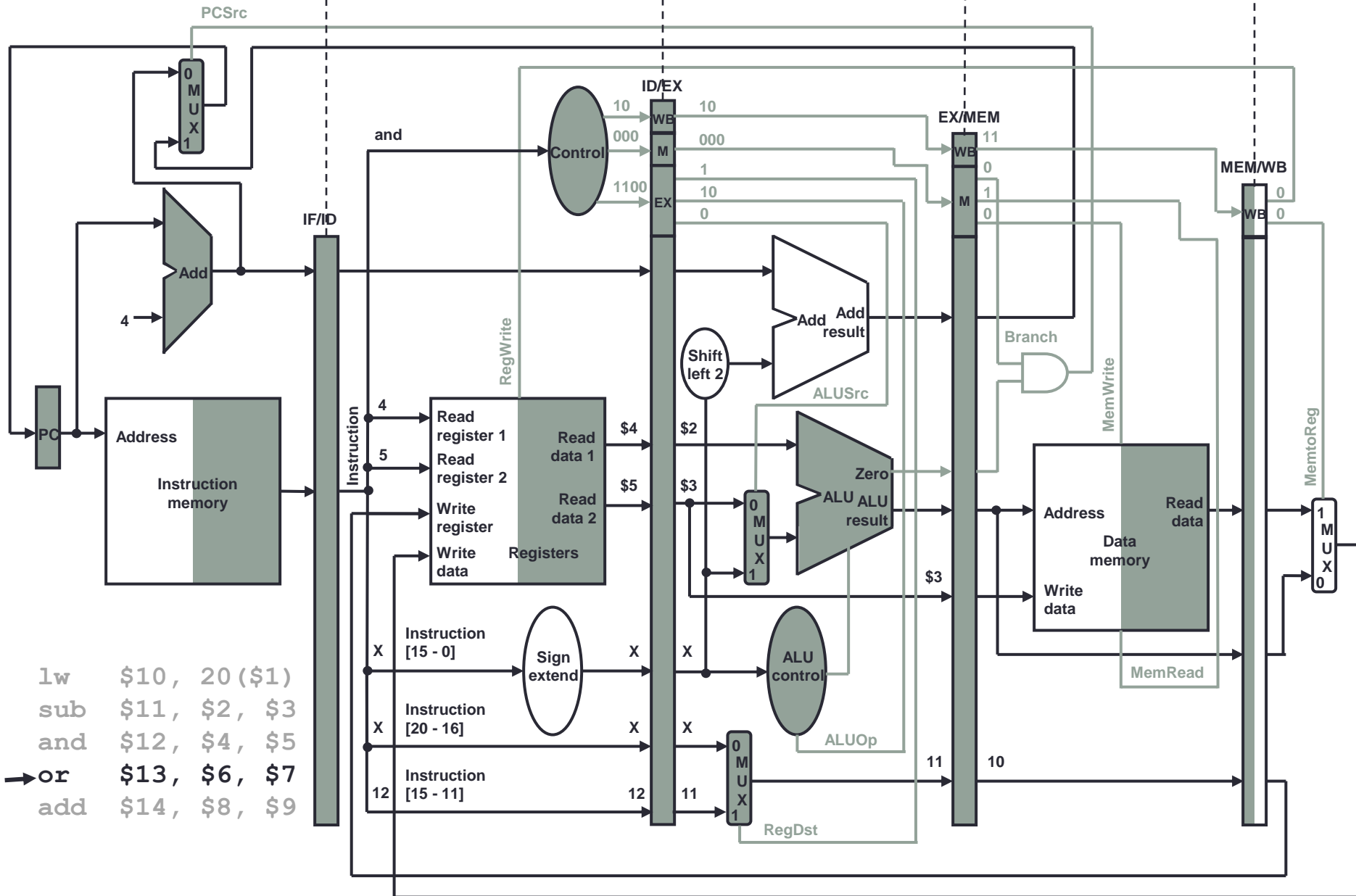
IF: or \$13, \$6, \$7

ID: and \$12, \$4, \$5

EX: sub \$11, \$2, \$3

MEM: lw \$10, 20(\$1)

WB: b &lt;1&gt;



# Execution example: clock cycle 5/9

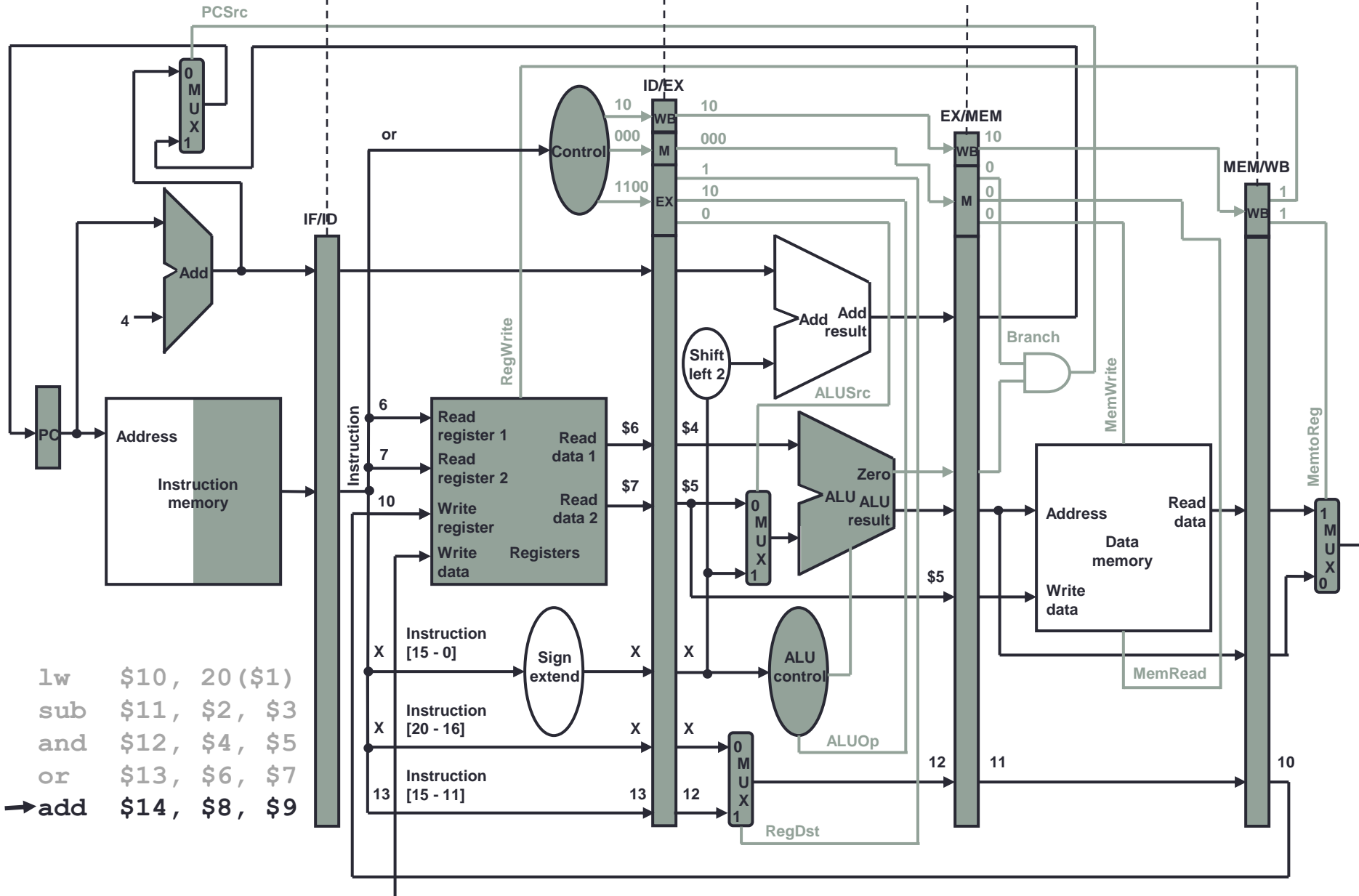
IF: add \$14, \$8, \$9

ID: or \$13, \$6, \$7

EX: and \$12, \$4, \$5

MEM: sub \$11, \$2, \$3

WB: lw...



# Execution example: clock cycle 6/9

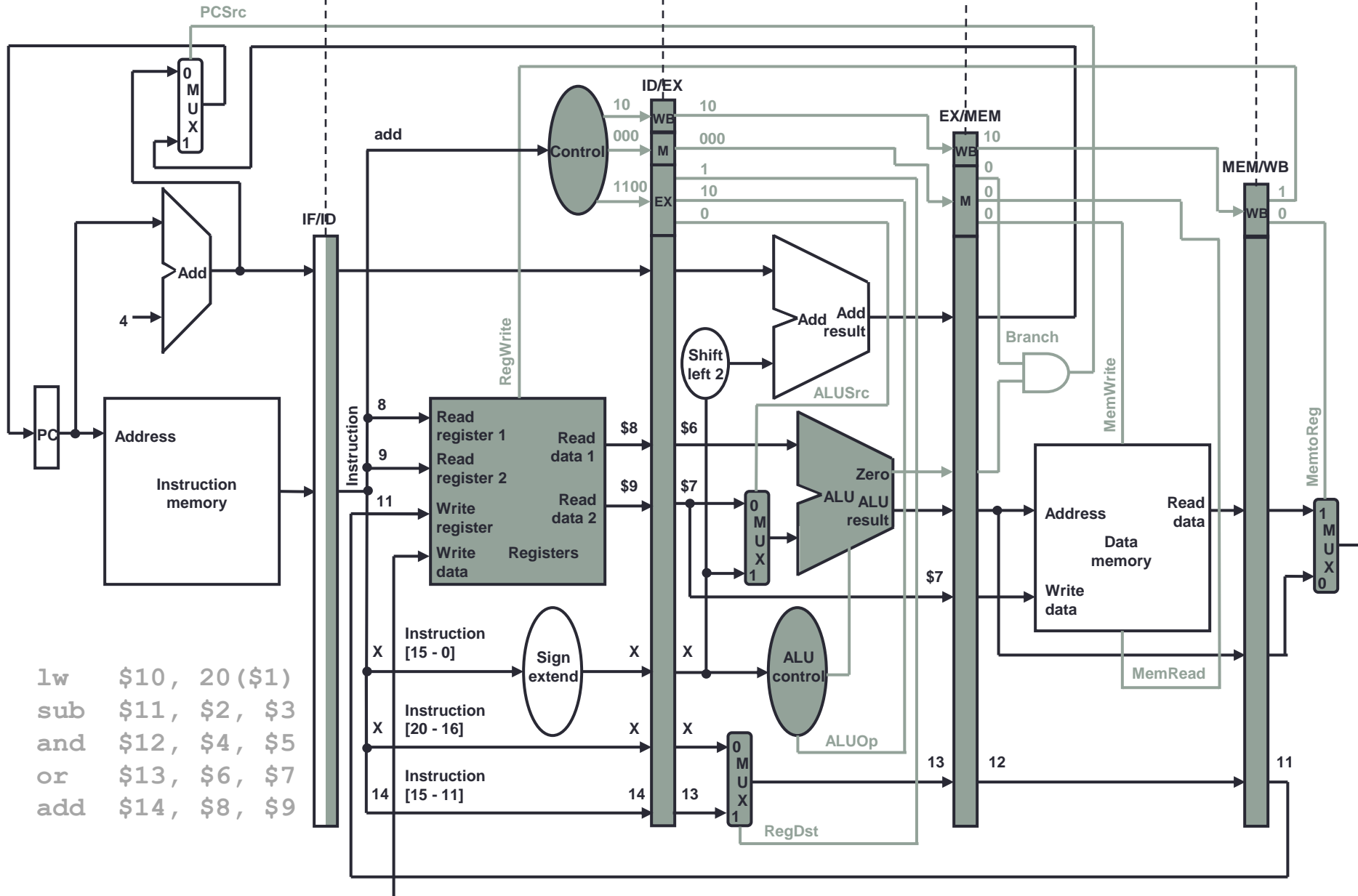
IF: after &lt;1&gt;

ID: add \$14, \$8, \$9

EX: or \$13, \$6, \$7

MEM: and \$12, \$4, \$5

WB: sub...



# Execution example: clock cycle 7/9

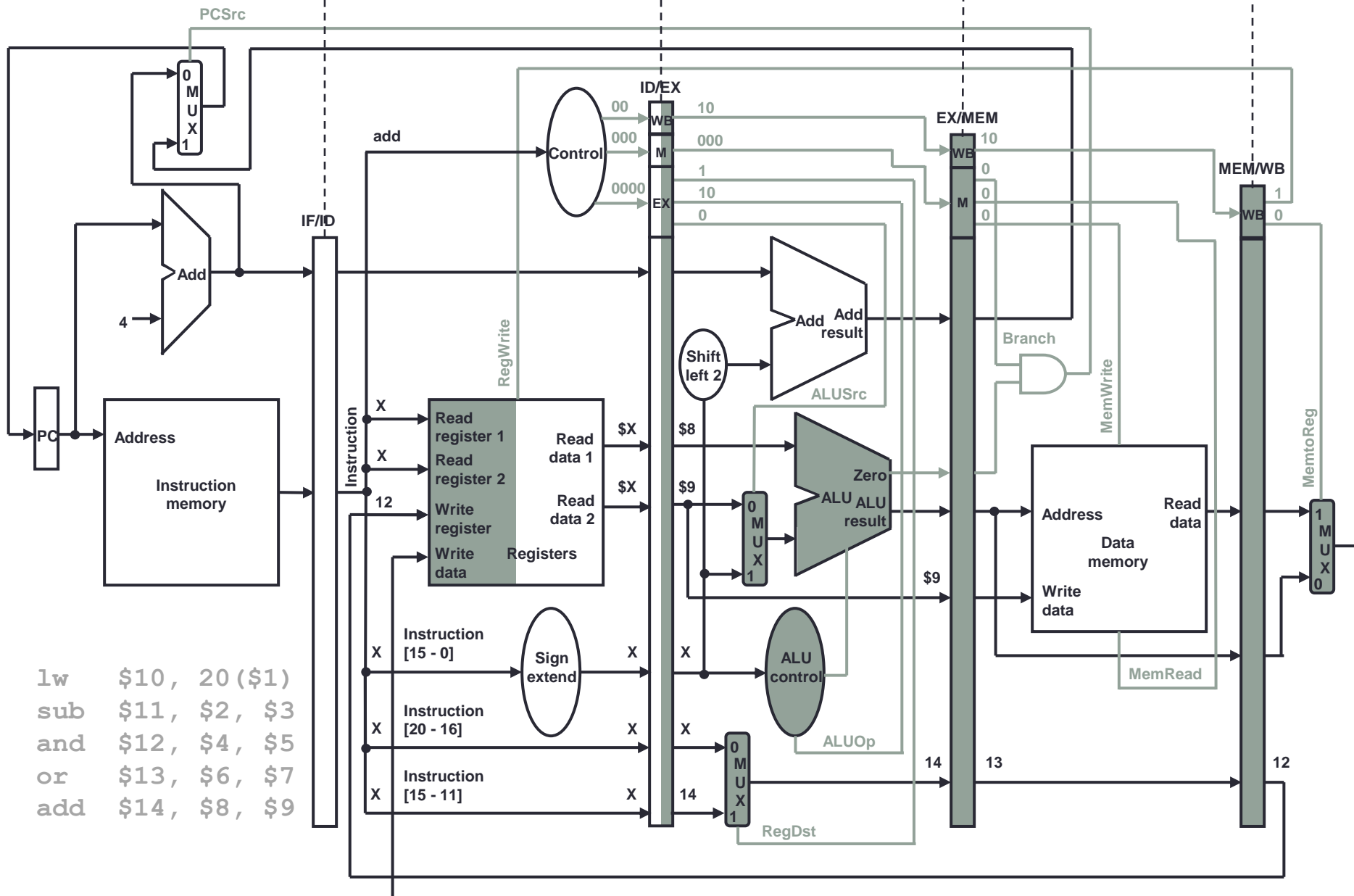
IF: after &lt;2&gt;

ID: after &lt;1&gt;

EX: add \$14, \$8, \$9

MEM: or \$13, \$6, \$7

WB: and...



# Execution example: clock cycle 8/9

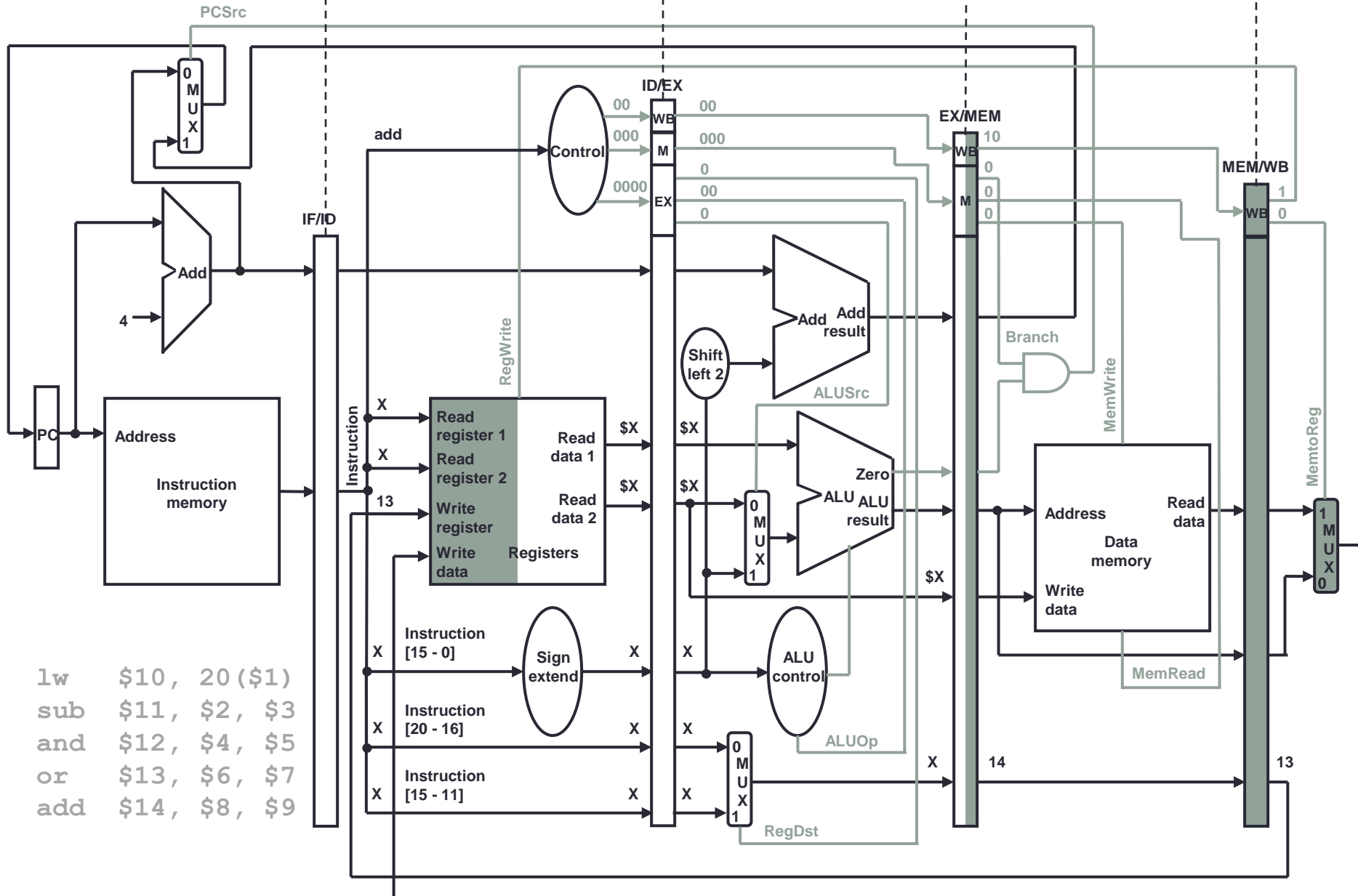
IF: after &lt;3&gt;

ID: after &lt;2&gt;

EX: after &lt;1&gt;

MEM: add \$14, \$8, \$9

WB: or...



# Execution example: clock cycle 9/9

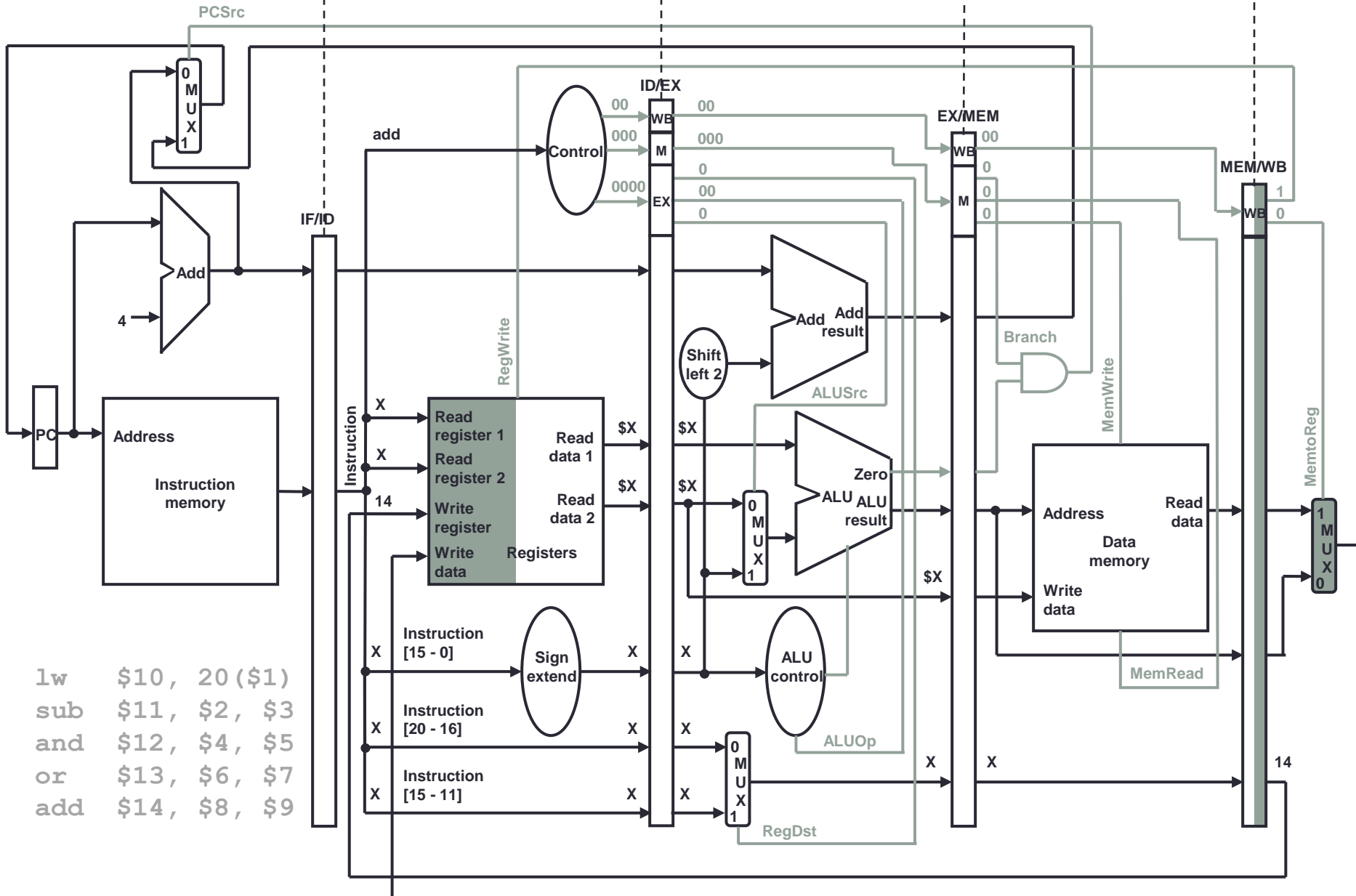
IF: after &lt;4&gt;

ID: after &lt;3&gt;

EX: after &lt;2&gt;

MEM: after &lt;1&gt;

WB: add...





# In-class exercise

**Given stage delays as below**

- 100ps for register read or write
- 200ps for other stages

**Compare the performance of the pipelined datapath with**

- single-cycle datapath
- multi-cycle datapath

Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time used
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 speedup

- **If all stages are balanced**
  - **i.e. equal stage delay**

$$\text{speedup} = \frac{\text{exec\_time}_{\text{non-pipe}}}{\text{exec\_time}_{\text{pipe}}} \\ = \text{no. of stages}$$

- **If not balanced, speedup is less**
- **Speedup is due to increased throughput**
  - **Latency (time for each instruction) does not decrease**
- **Any other issues?**