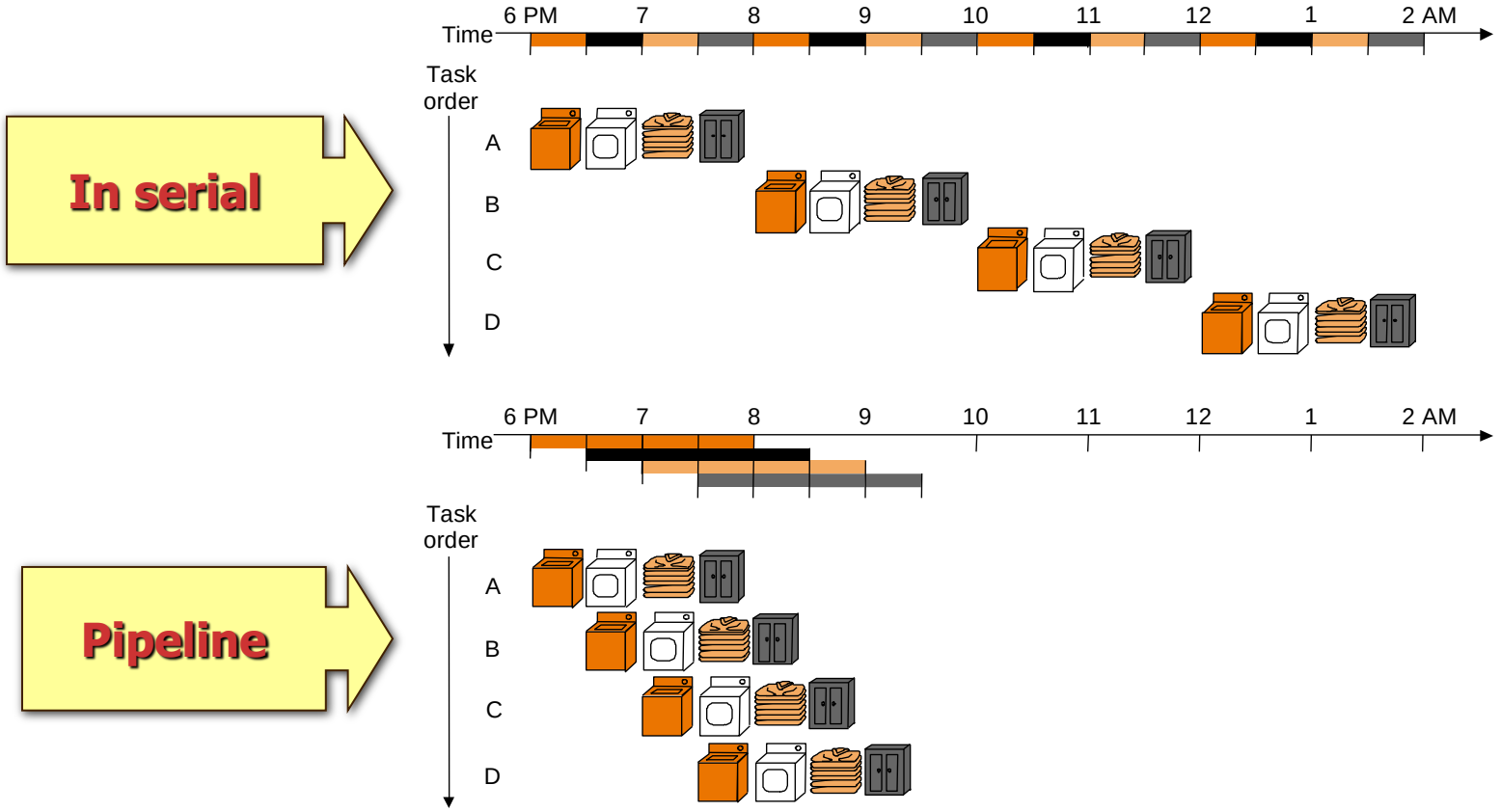
COMP2611: Computer Organization

The Pipelined Processor

Background

High-Performance Processors

- Two techniques for designing high-performance processors by exploiting parallelism:
  - **Multiprocessing: parallelism among multiple processors**
  - **Pipelining: implements parallelism between instructions on the same processor**



Pipelining Principles

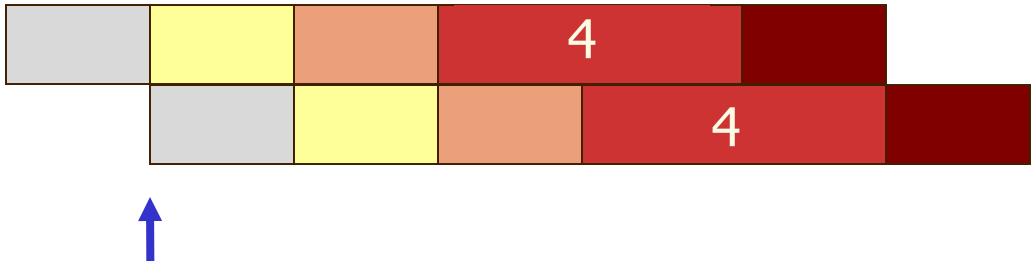
Key characteristics:

- **Multiple tasks** are processed simultaneously
- Ideally, these tasks should be **independent** of each other otherwise we need to make this the case
- Pipelining **does not help the latency** of a single task
- It **helps the throughput** of the entire workload
- Completion order in pipelined execution = that in sequential execution

How much can a pipeline improve?

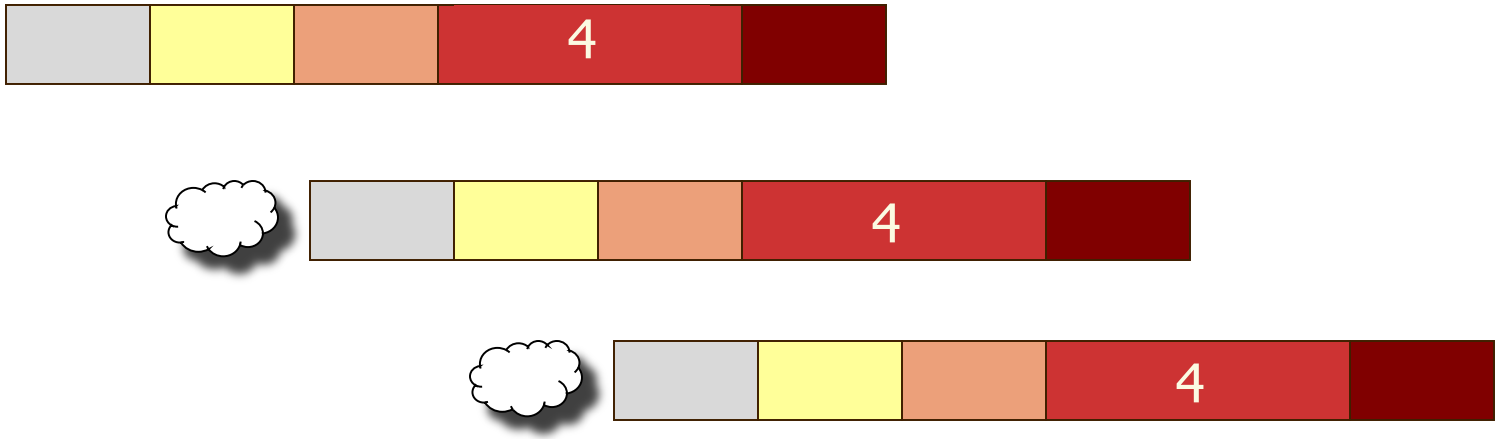
- **Potential speedup = number of pipeline stages**
- The pipeline rate is limited by the slowest pipeline stage
- Unbalanced lengths of pipeline stages can reduce speedup. Why?

Example: Bottleneck in Pipelining



- Can I align the pipeline stages as above?
- Answer: NO, because the tasks executing in parallel are not independent (task 4 overlaps task 4)

- The condition to align is to make sure NO OVERLAP of any stages?



The MIPS Pipeline

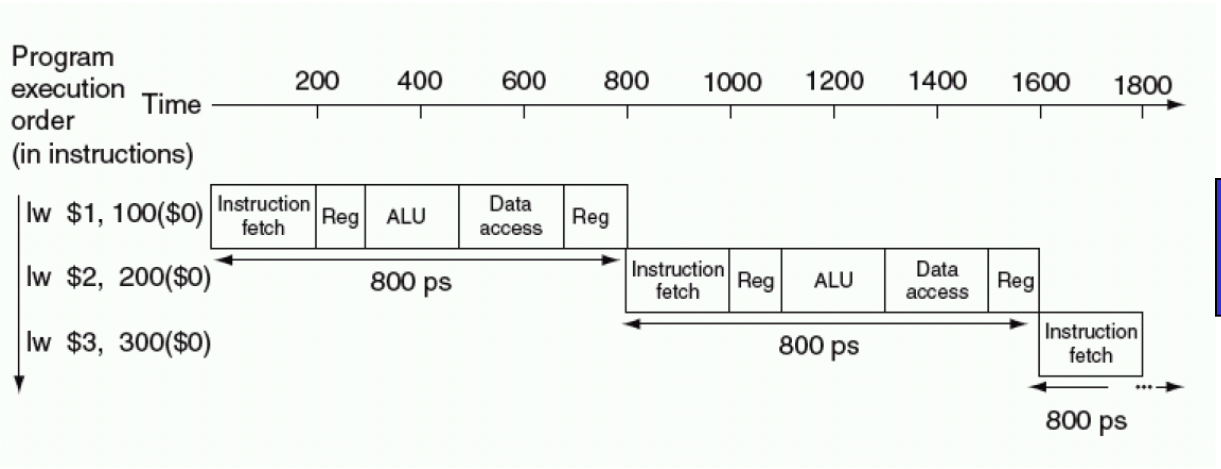
Pipeline Performance Example

- Assume we require:
  - 100 picoseconds for register read or write
  - 200 picoseconds for all other stages

Instruction	Instruction fetch	Register read	ALU op	Memory access	Register write	Total time
Load Word (lw)	200ps	100 ps	200ps	200ps	100 ps	800ps
Store Word (sw)	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
Branch (beq)	200ps	100 ps	200ps			500ps

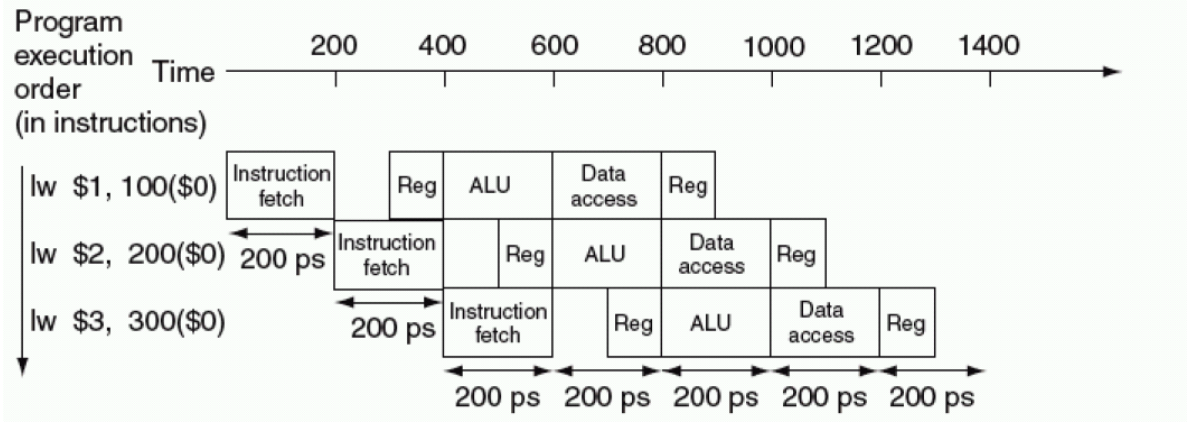
Pipelined vs. Non-pipelined

- Single-cycle, non-pipelined execution:



Total execution time = 2400 ps

- Pipelined execution



Total execution time =1400 ps

The Performance Benefit

- The instruction **delays** in the example:
  - 800 ps (single cycle datapath)
  - 1000 ps (pipelined datapath)
- The instruction **throughputs** in the example:
  - 1 instruction per 800 ps (single cycle datapath)
  - 1 instruction per 200 ps (long time average for pipelined datapath)
- Pipelining does not improve the latency of a single instruction, it improves the throughput of the system (i.e., the datapath)

- In general (ideally), if we have a N stage pipeline:
  - We need N-1 cycles to fill the pipeline,
  - Then one instruction will finish per cycle
  - So the throughput is: Clock Rate x IC/(IC + N - 1), with IC >> N

The Performance Benefit

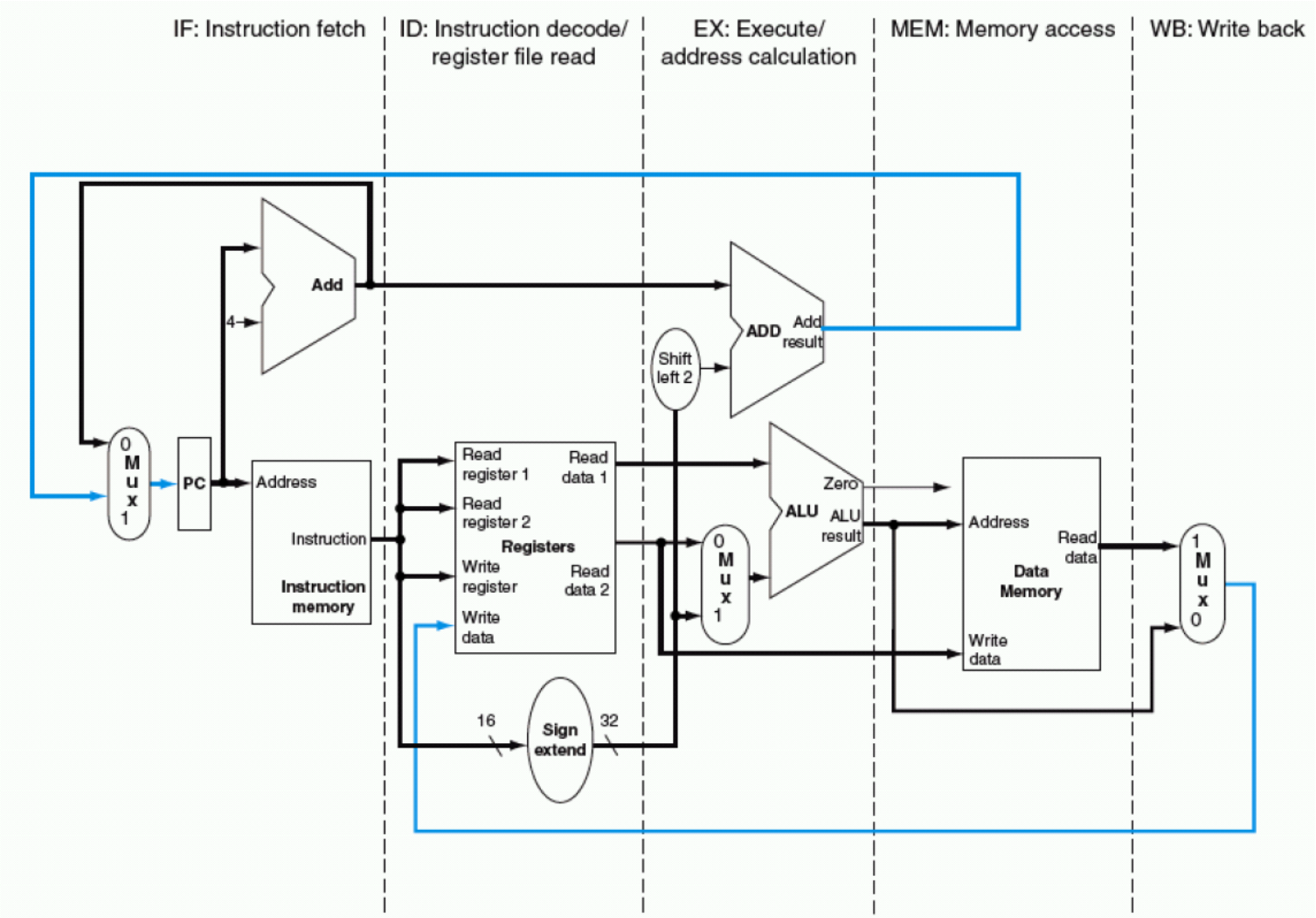
- Pipeline speed is limited by the most time consuming pipeline stage, as this stage determines the duration of a clock cycle (why?)
- A well balanced pipeline is one such that each stage takes the same amount of time to execute.
- When the pipeline is well balanced :

$$\text{Time between instructions}_{\text{pipelined}} = \frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of stages}}$$

- If the pipeline is well balanced, the **speedup** equals to Number of pipeline stages (a.k.a depth of the pipeline).

MIPS Pipeline

- Basic idea: take a single-cycle datapath and separate it into 5 pieces. Each piece responsible for a single instruction execution stage.



MIPS ISA for Pipelining

ISA design affects the complexity of pipeline implementation.

MIPS ISA is designed for pipelining

- **All instruction are of the same length** (32-bit)

  Easy to fetch one instruction in first stage of the pipeline and decode it in the second

- **It has just a few similar instruction formats**

  With the source register fields being located in the same place in all instructions, 2nd stage can read the register file while decoding the type of instruction just fetched

- **Memory operands only appear in loads and stores**

  We can use the execute stage to calculate the memory address and then access memory in the following stage

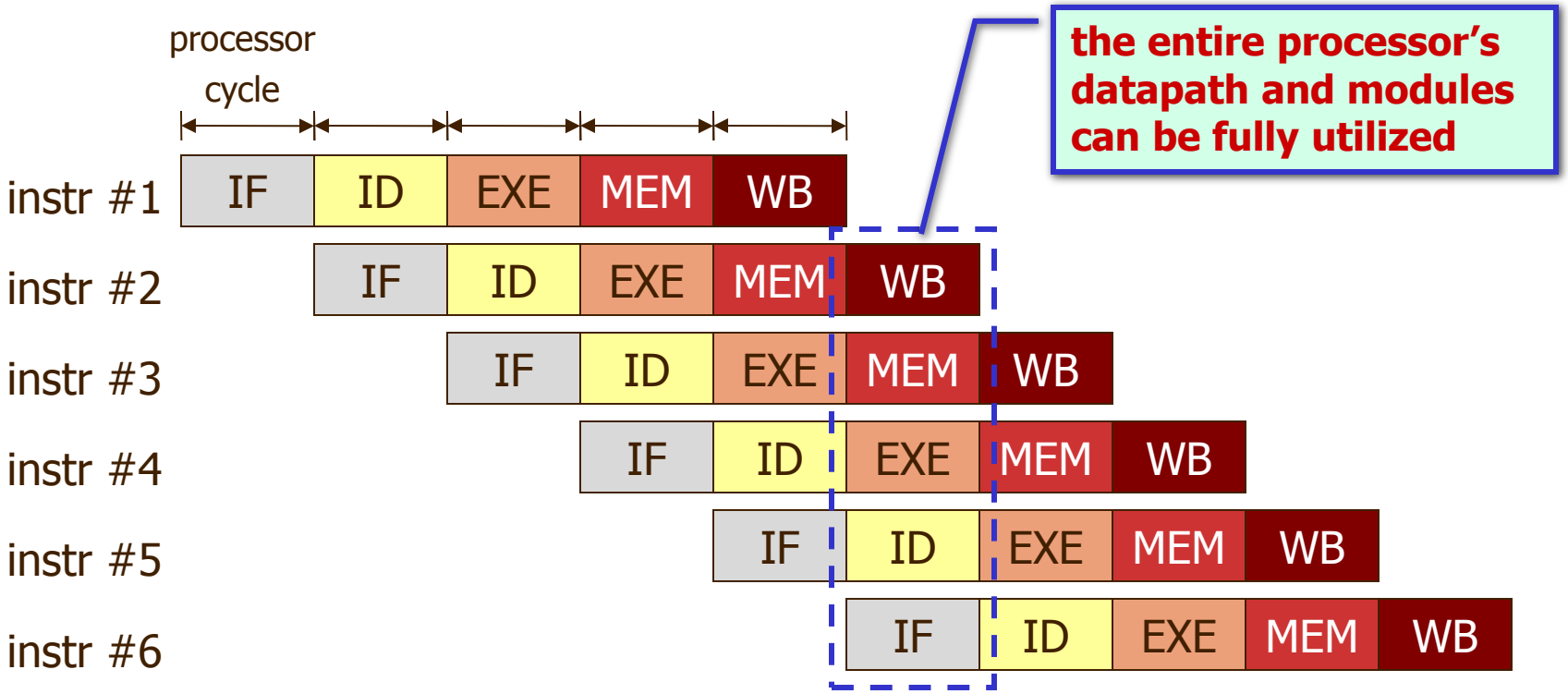
- **Alignment of memory operands on word boundaries**

  We need not worry about a single data transfer instruction requiring two memory accesses; the data can be transferred between processor and memory in a single pipeline stage

Pipelining Instructions

In pipelined processor,

- Each instruction takes multiple steps
- Each step is independent of each other and takes different datapath



- At each cycle, one instruction is fetched and sent to the processor
- Ideally, after pipeline is fully filled, one instruction completes each cycle

MIPS Pipeline Stages

Execution of each instruction is broken into 5 stages: (in the order of execution)

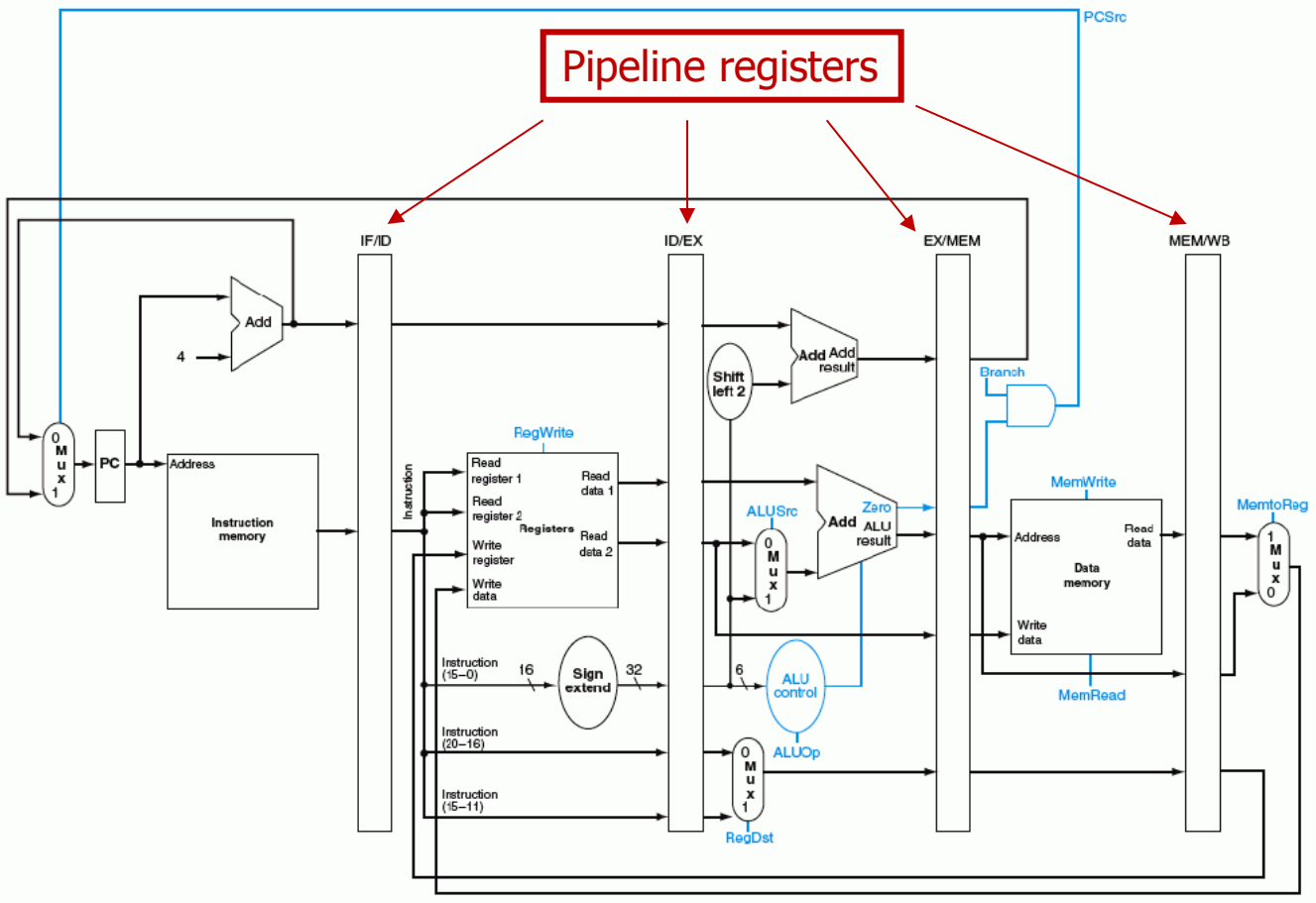
- **IF** : Fetch the instruction from memory
- **ID** : Instruction decode & register read
- **EX** : Perform ALU operation
- **MEM** : Memory access (if necessary)
- **WB** : Write result back to register

Each stage uses a different hardware unit and takes one clock cycle to complete.

- Instructions can **co-exist** in the datapath if all of them are in different stages of execution from one another

Pipeline Registers

- Additional **pipeline registers** are needed
  - Located between the stages, i.e. **IF/ID**, **ID/EX**, **EX/MEM**, **MEM/WB**)
  - Hold information produced in the previous cycle

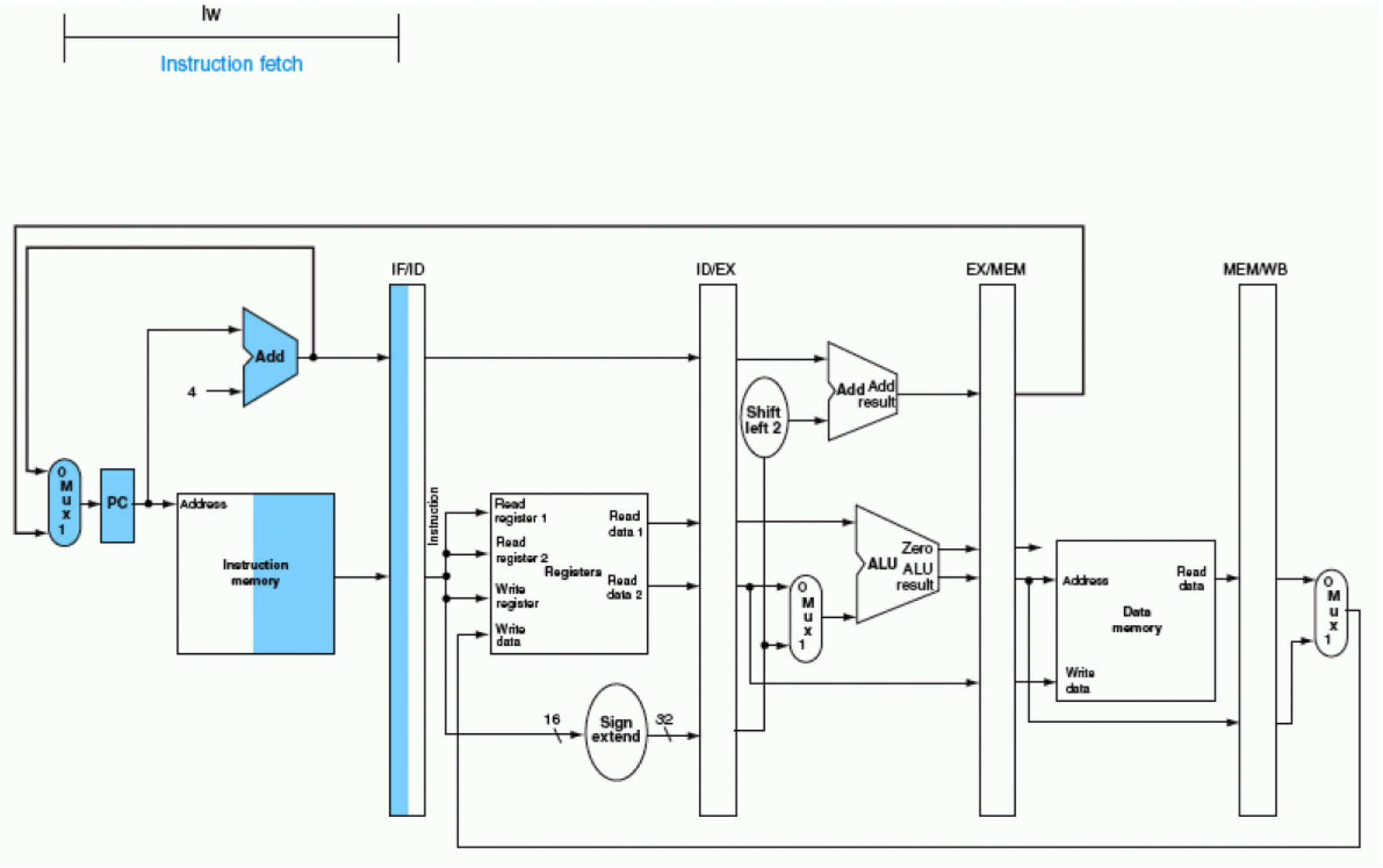


Pipeline Diagram

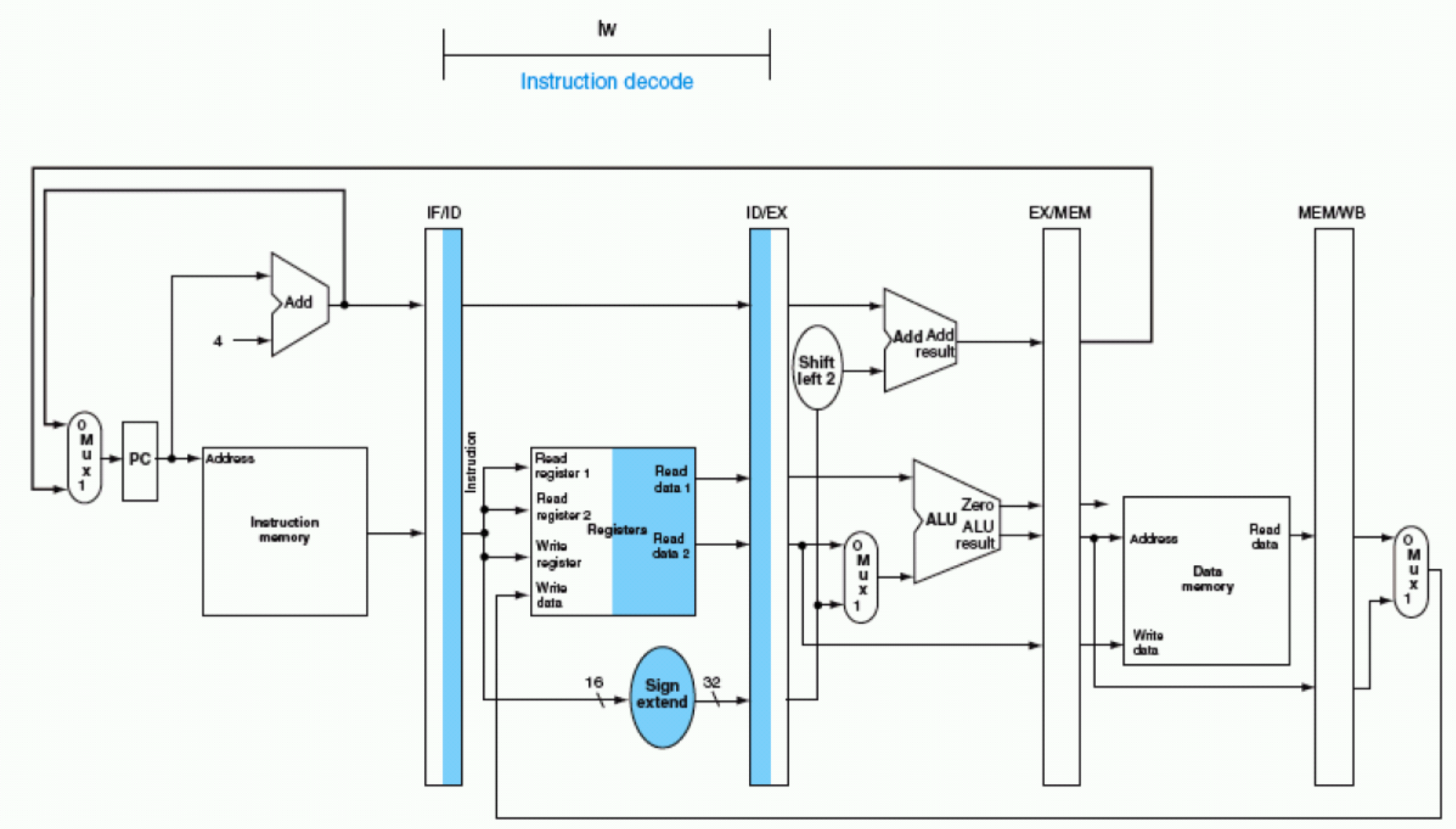
- Every clock cycle, many instructions are simultaneously executing in a single datapath

- Two common ways of showing the pipeline operations
  - **Single-clock-cycle pipeline diagram** : shows the pipeline usage in a single cycle, highlight the resources used.
    - An example of lw execution is shown in following pages
  - **Multi-clock-cycle pipeline diagram** : shows the graph of operation over time.

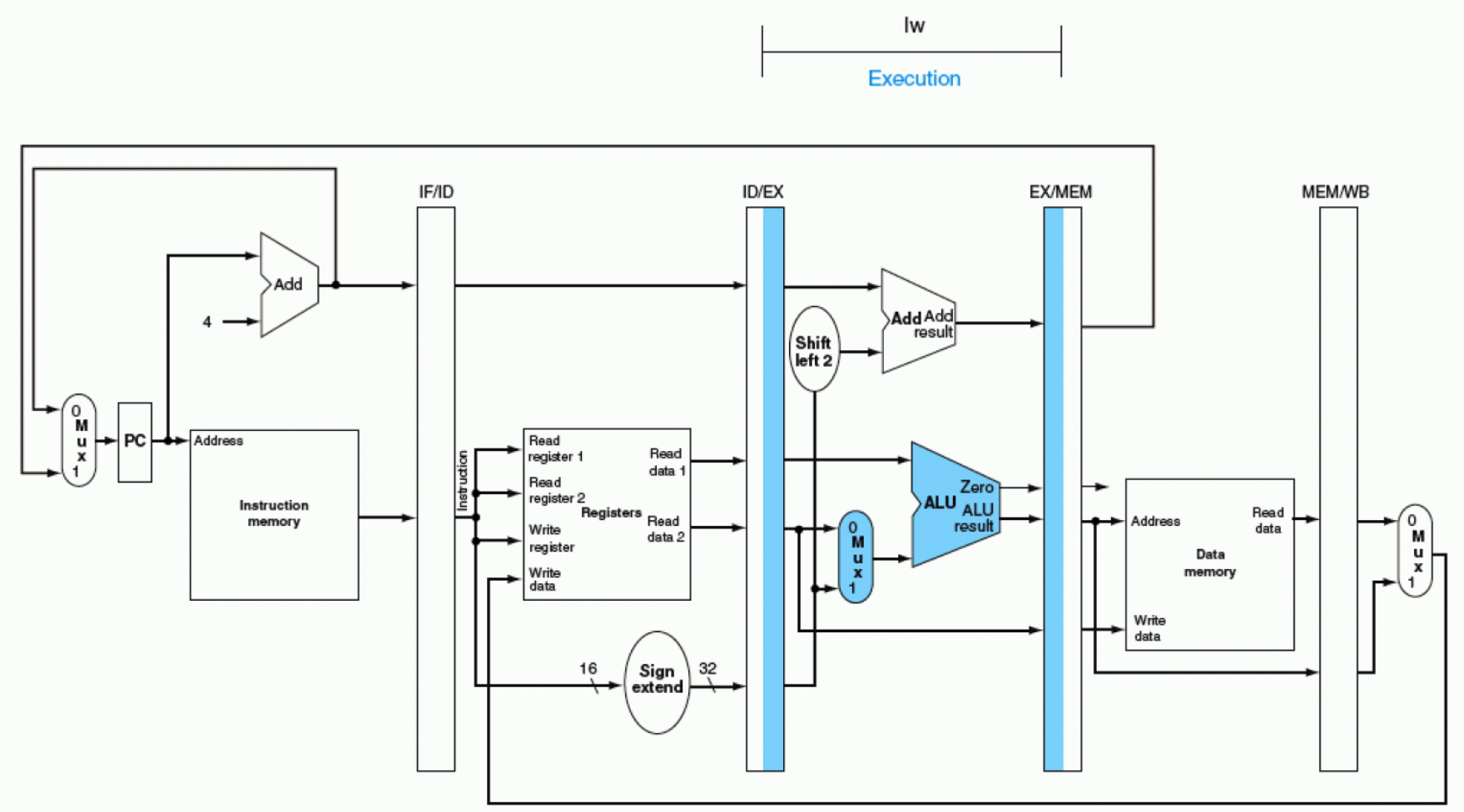
Single clock cycle diagram: IF stage of lw



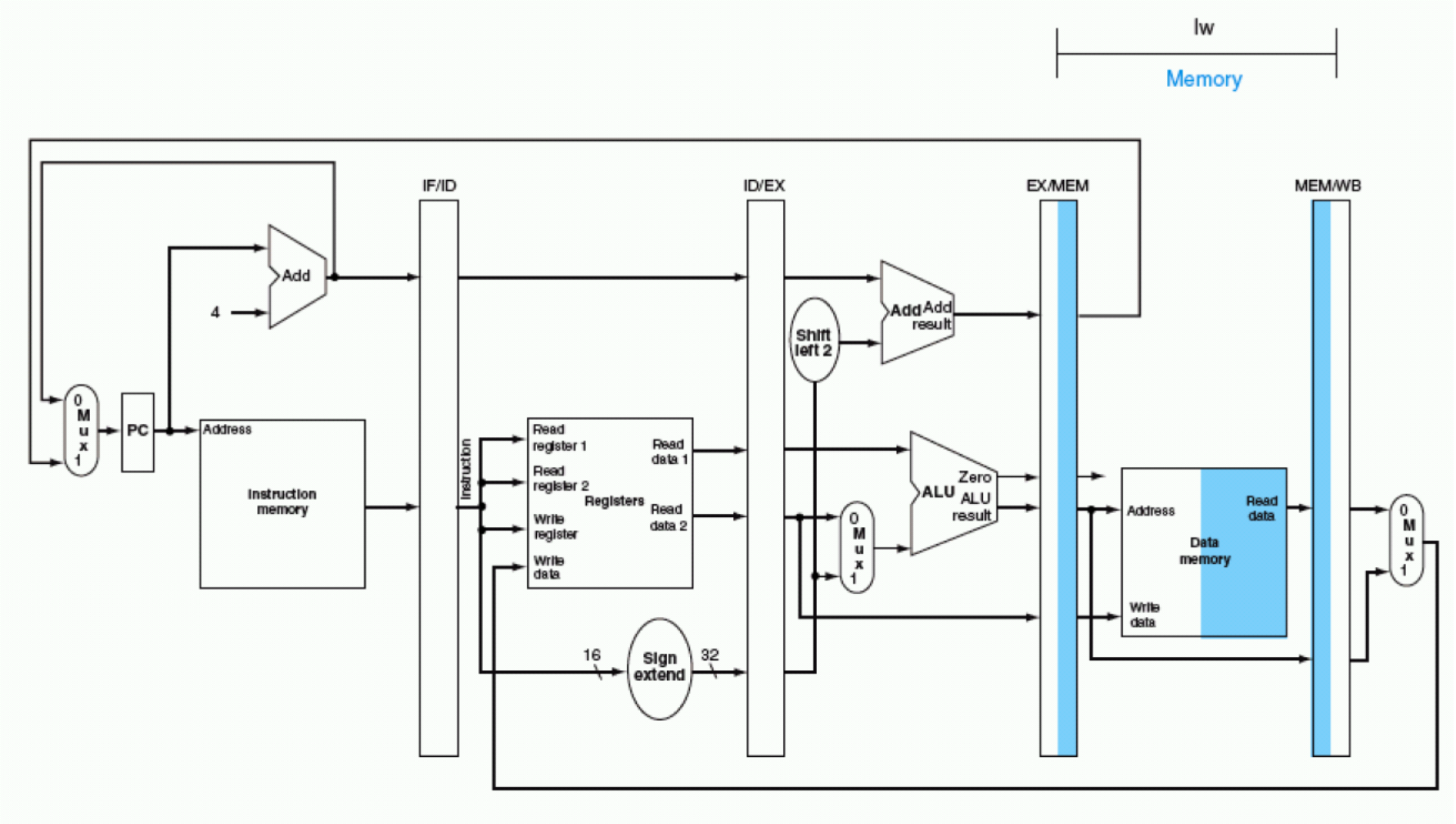
Single clock cycle diagram: ID stage of lw



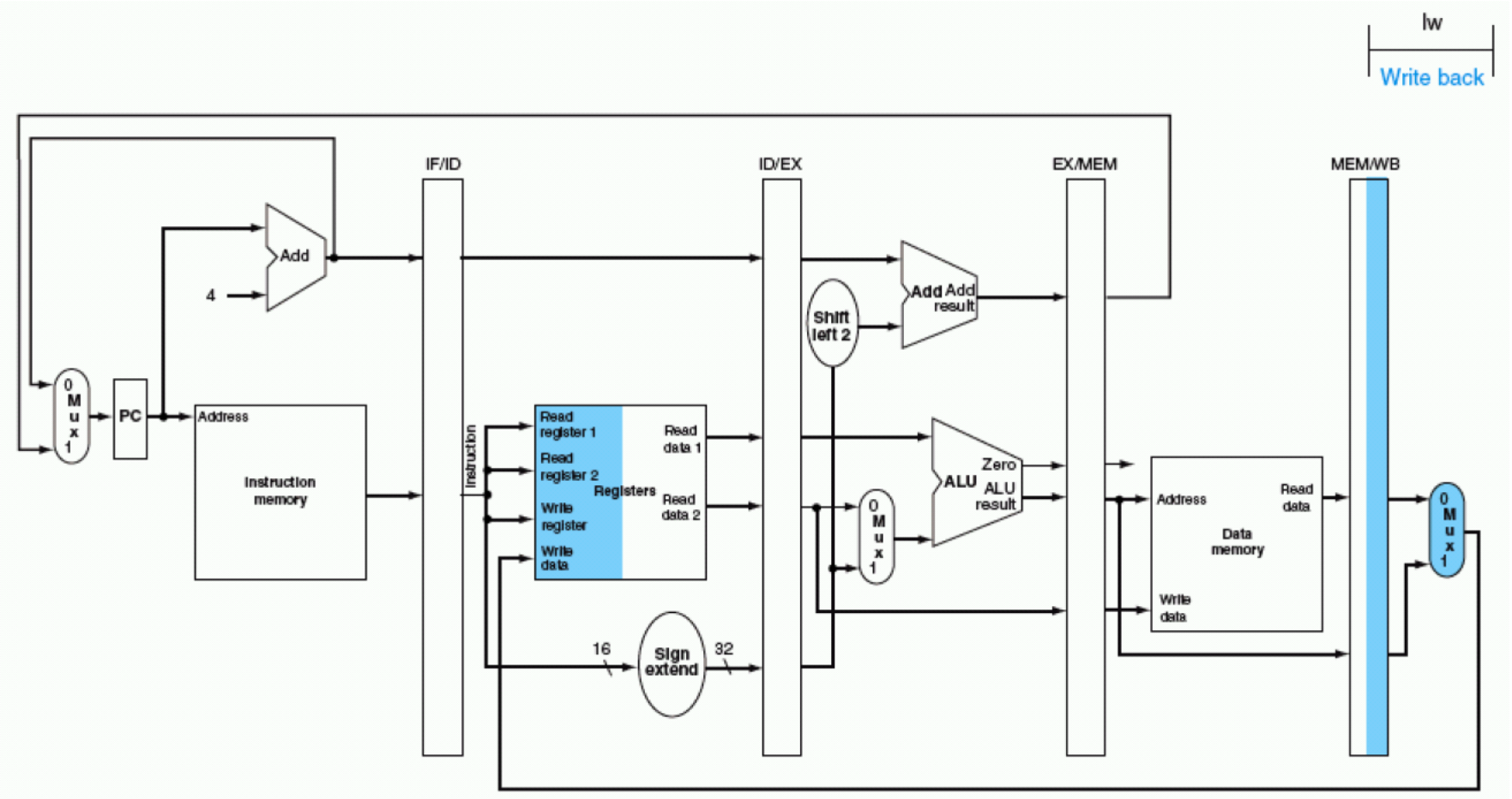
Single clock cycle diagram: EXE stage of lw



Single clock cycle diagram: MEM stage of lw

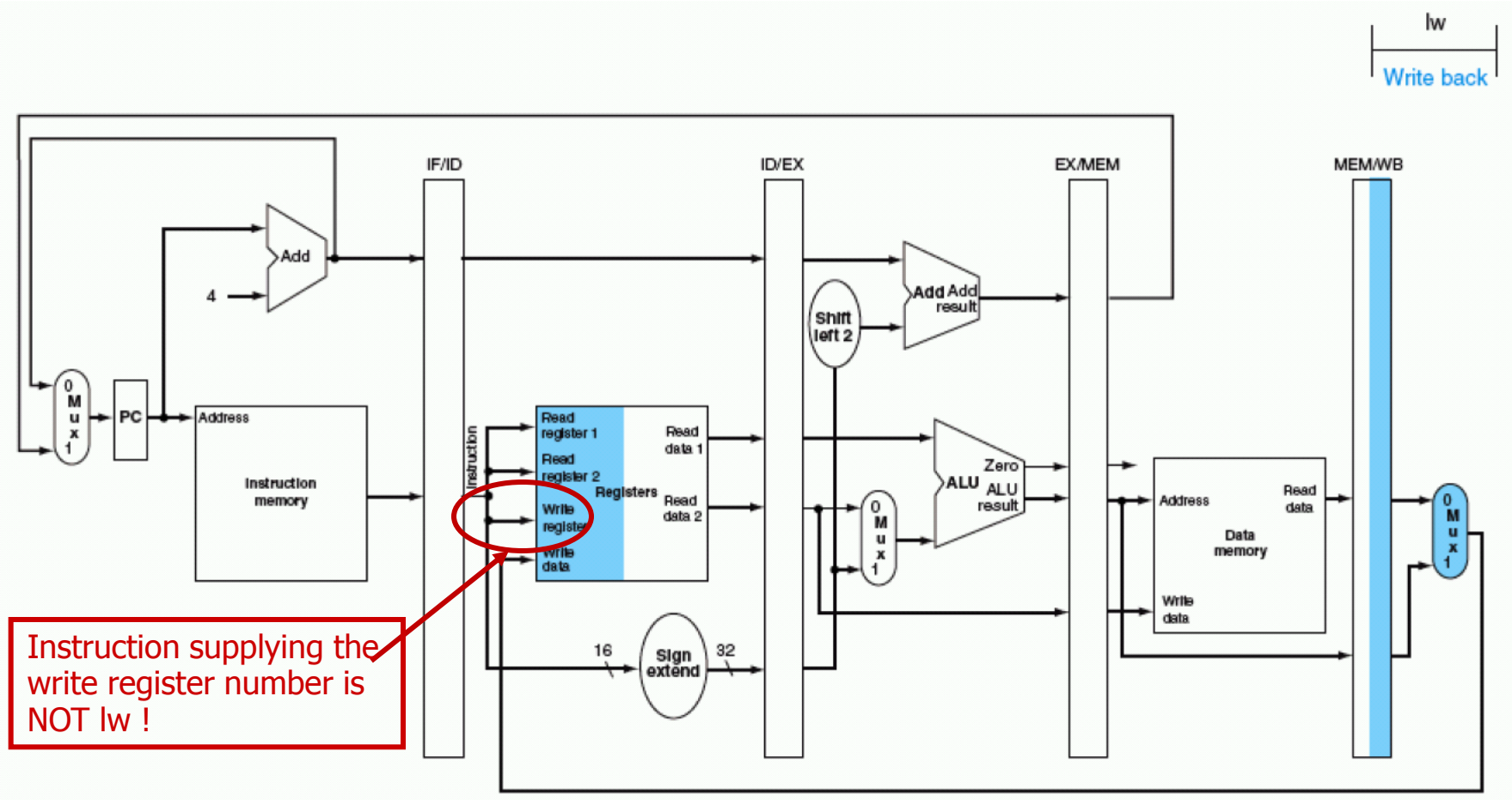


Single clock cycle diagram: WB stage of lw



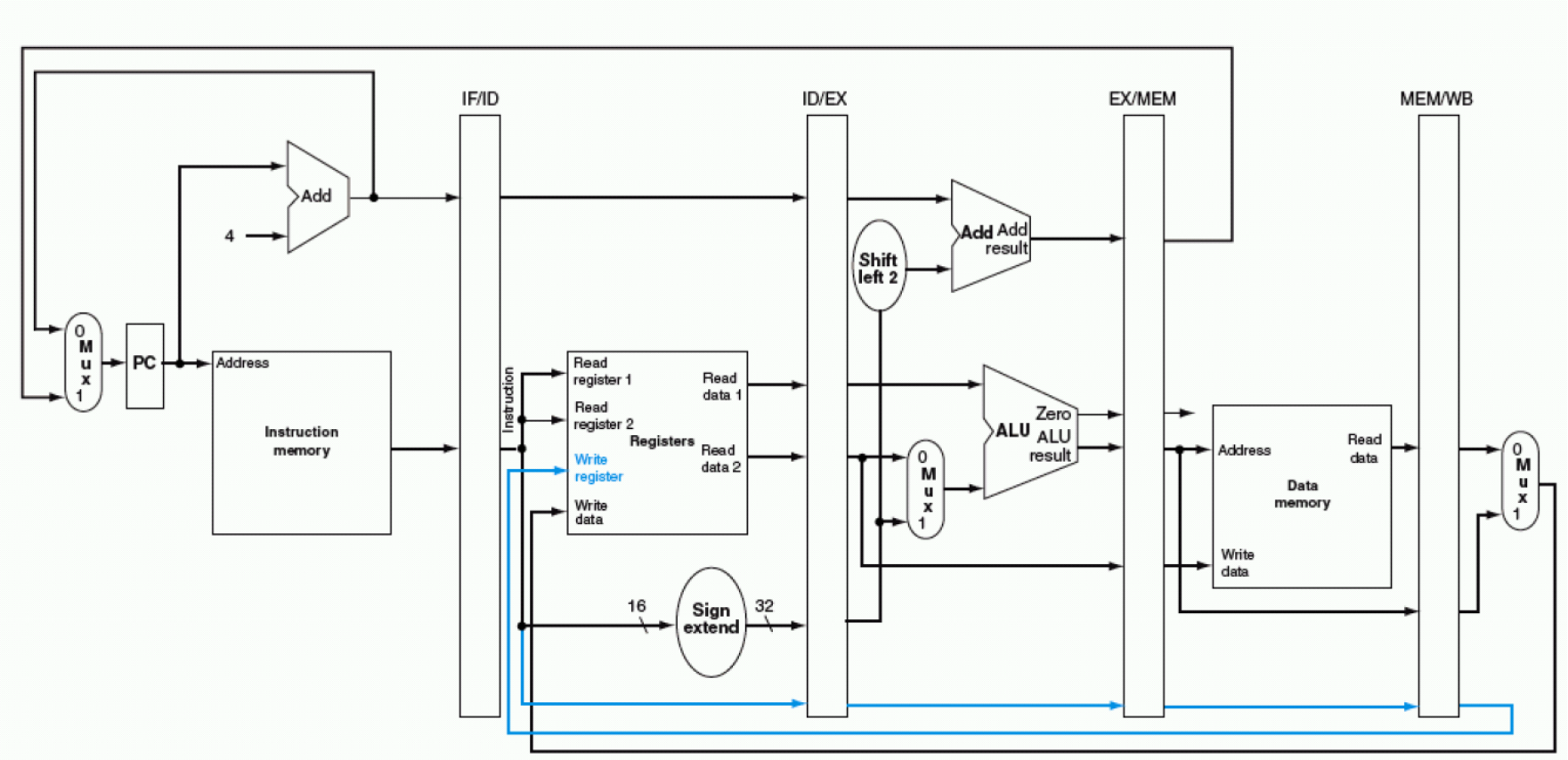
Single clock cycle diagram: WB stage of lw

- There is a problem with the WB stage of lw!



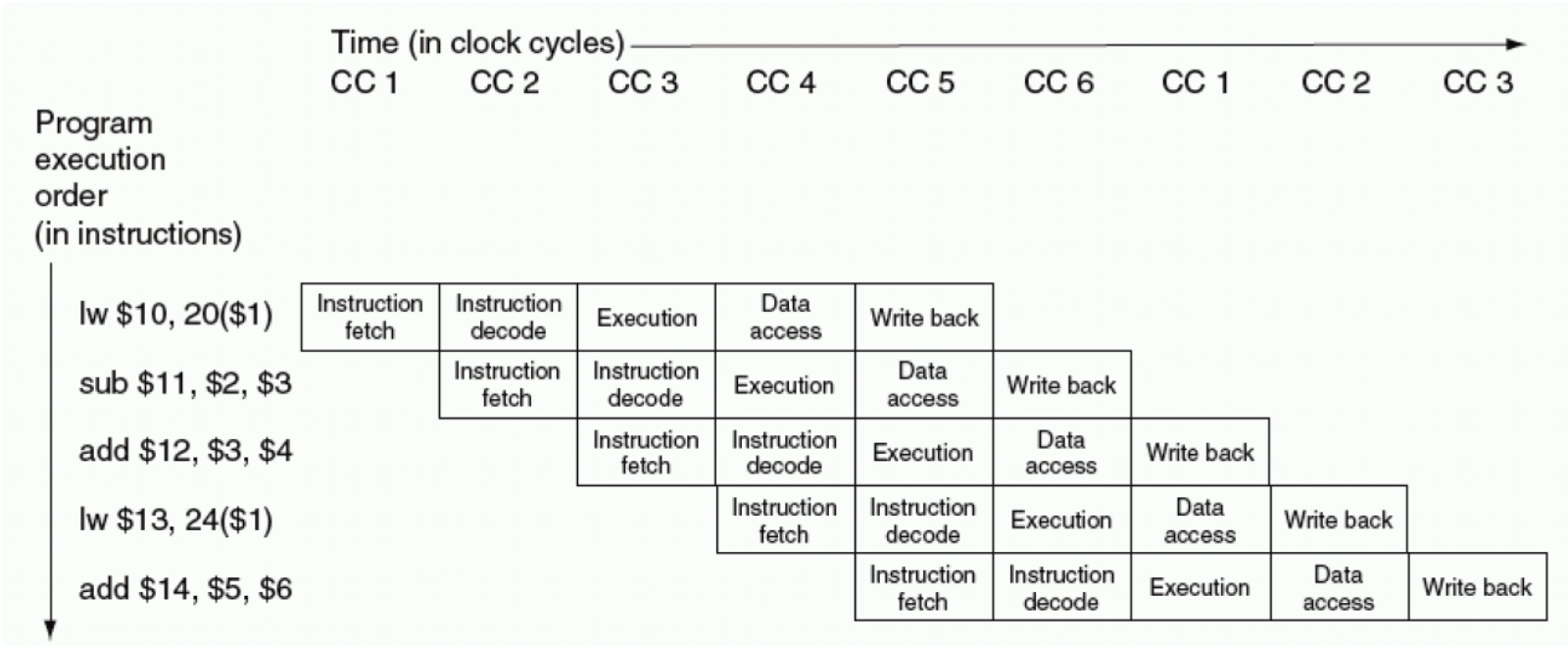
The corrected Datapath for lw

- To solve this problem: the "write register" information is forwarded from the MEM/WB pipeline registers.



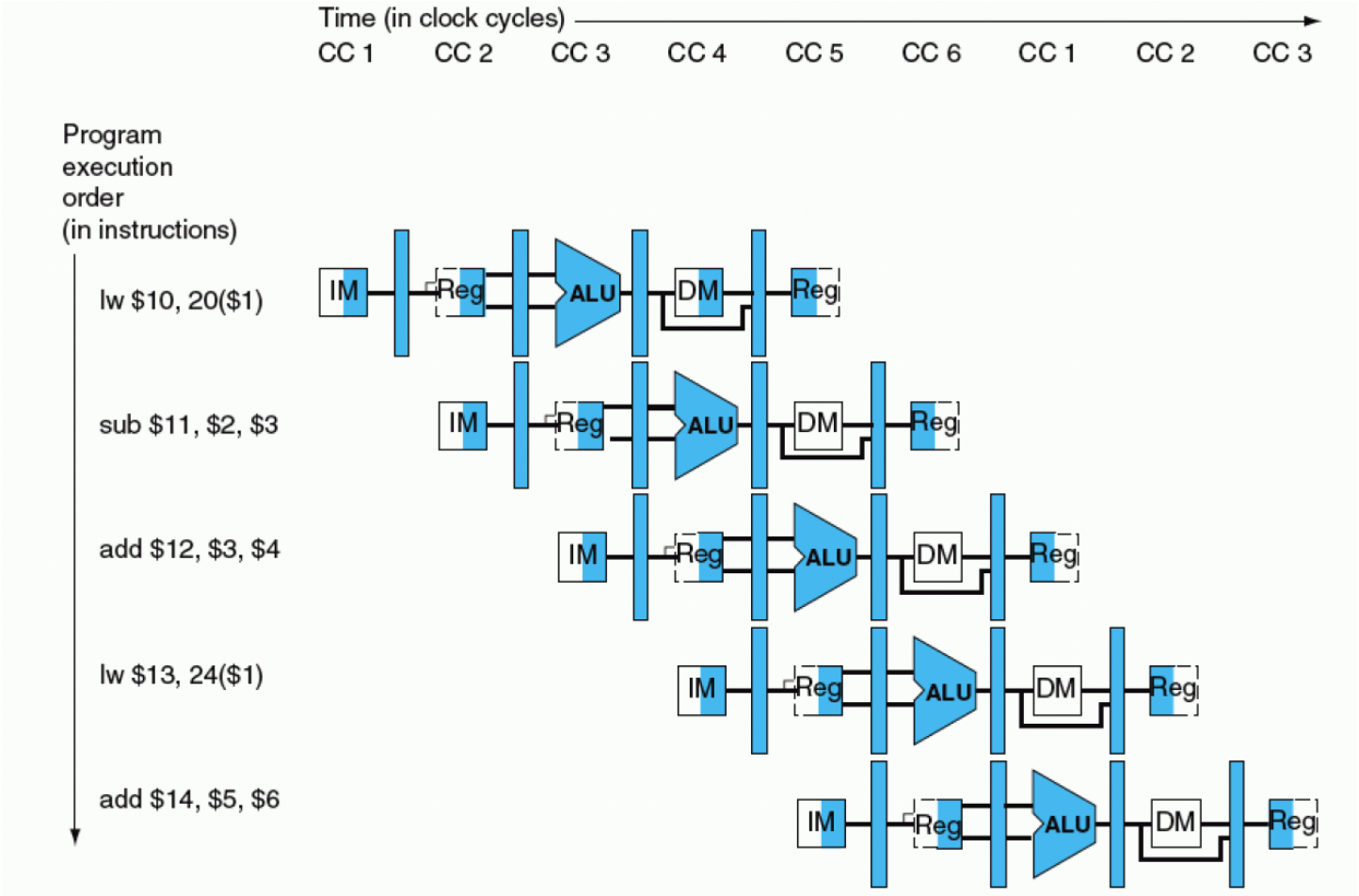
Multi-clock-cycle pipeline diagram : traditional view

- The following diagram shows the execution of a series of instructions.

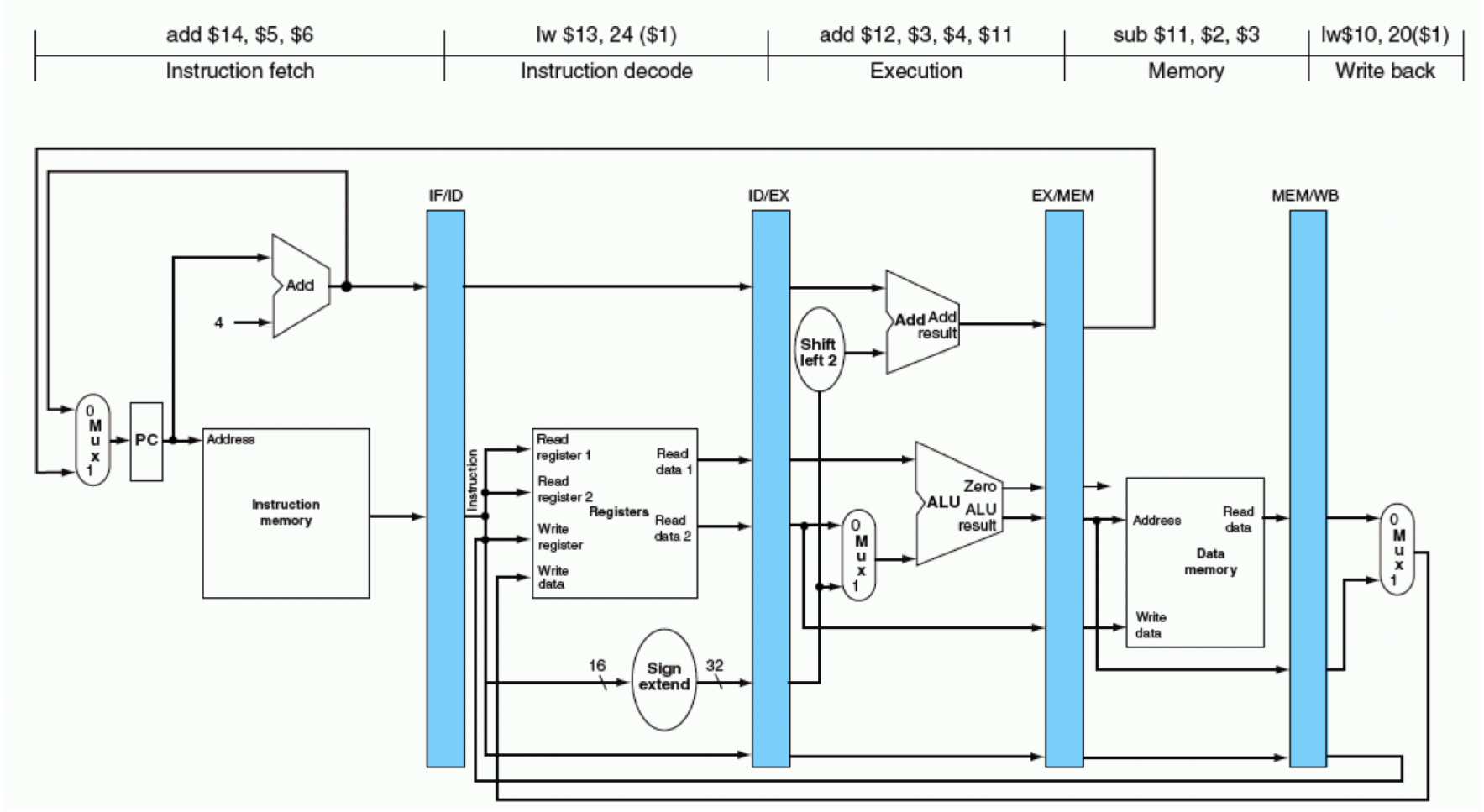


Multi-clock-cycle pipeline diagram: graphical view

- The multi-clock-cycle form showing the hardware utilizations.



Single-clock-cycle diagram in CC5



The Pipeline operation

- **Ideally**
  - One stage begins in every cycle.
  - One stage completes in each cycle.
  - Each instruction takes 5 cycles
- In each clock cycle, several instructions are active.
- Different stages are executing different instructions.
- **Difficulty**:
  - How to generate the control signals ?
  - we need to set the control signals for each pipeline stage for each instruction.

Pipelined control

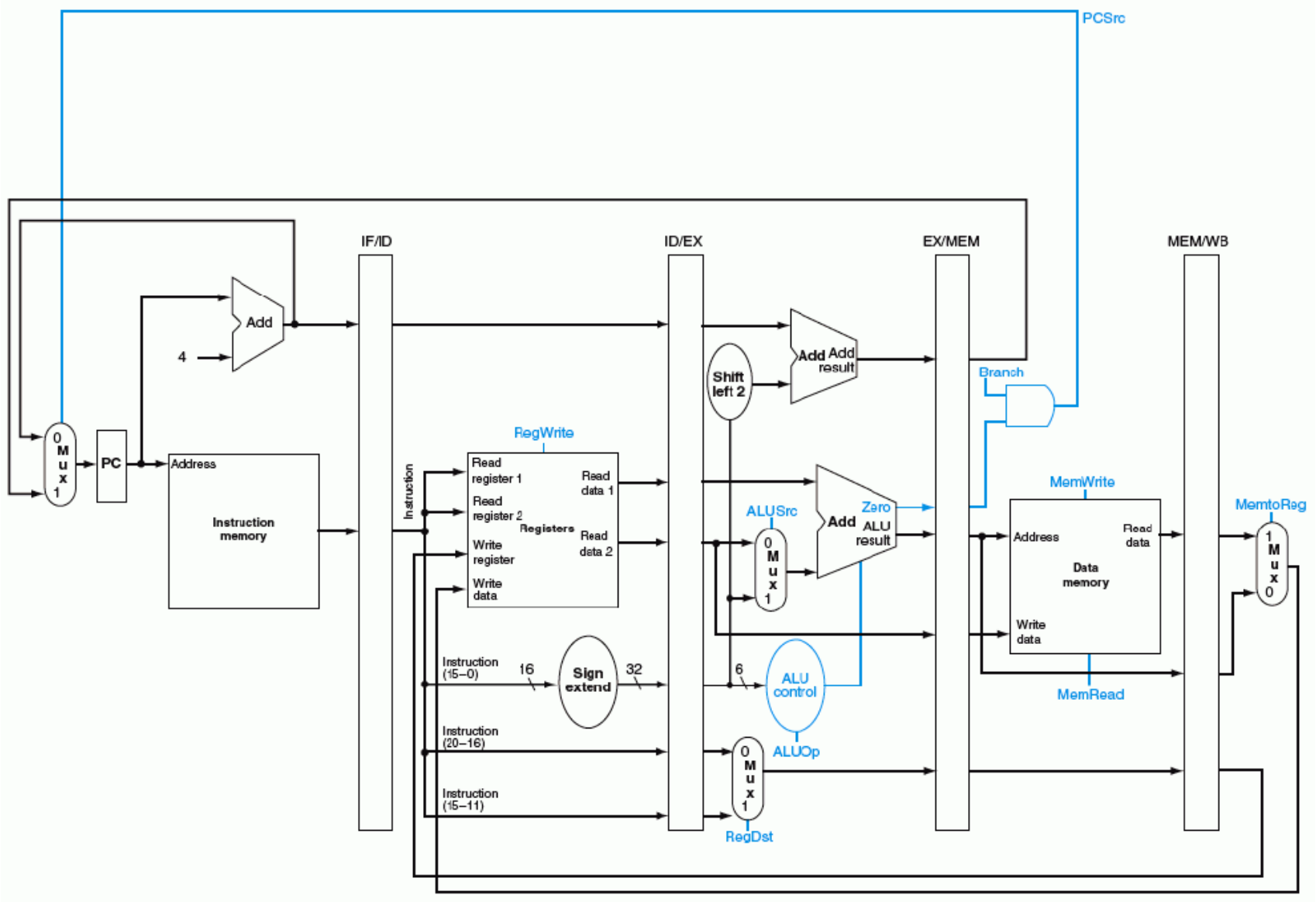
Pipelined Control?

- Let's start with a simple design that views the problem in a **greatly simplified way**.

- Use the **same controls as the single-cycle datapath**, but **pipeline** them so that the correct control signals are supplied for each stage of the instruction.
  - **Pass control signals together with the instruction through the pipeline**.

- Temporarily ignore data dependence related problems (**Hazards**), and will provide solutions to this problem later.

Control Signals Identified



Control Signals For Each Stage

The control signals required by each stage are grouped

Stage 1: **Instruction fetch (IF)** – no control signals, the instruction is read from the instruction memory and PC is updated to PC+4.

Stage 2: **Instruction decode and register read (ID)** – no control signals, instruction is decoded and source operands are read from register file.

Stage 3: **Execute (EX)**– *RegDst*, *ALUOp*, and *ALUSrc*.

Stage 4: **Memory Access (MEM)**– *Branch*, *MemRead*, and *MemWrite*

Stage 5: **Write Back (WB)** –*MemToReg* and *RegWrite*

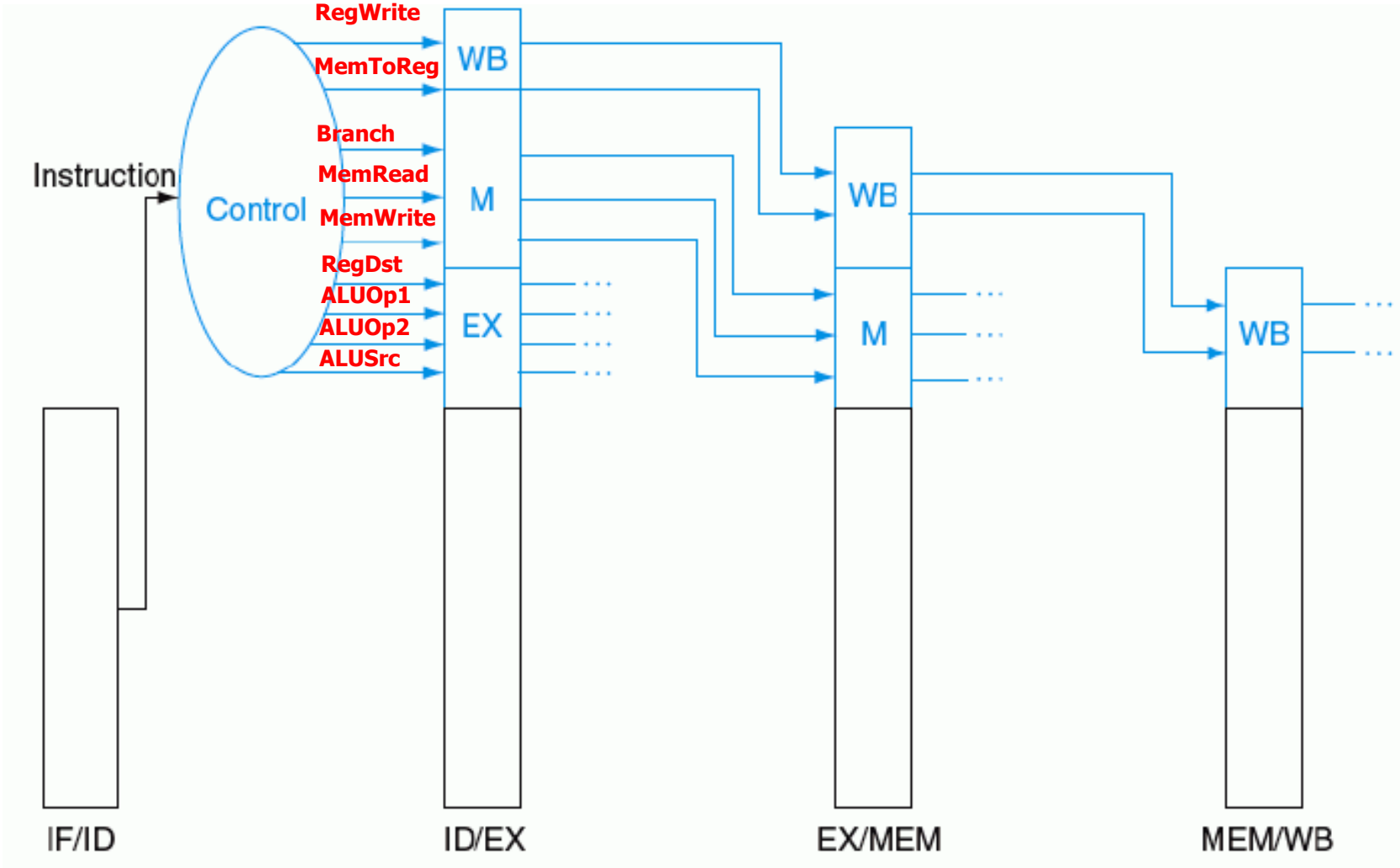
Control Signals For Each Stage

- The group of control signals and their values for different classes of instructions:

Instructions	EX				MEM			WB	
	RegDst	ALUOp1	ALUOp2	ALUSrc	Branch	MemRead	MemWrite	RegWrite	MemToReg
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

The Pipelined Control: passing the controls

- Control signals are passed to the next stage only if they are required



The Pipelined Control: the complete datapath