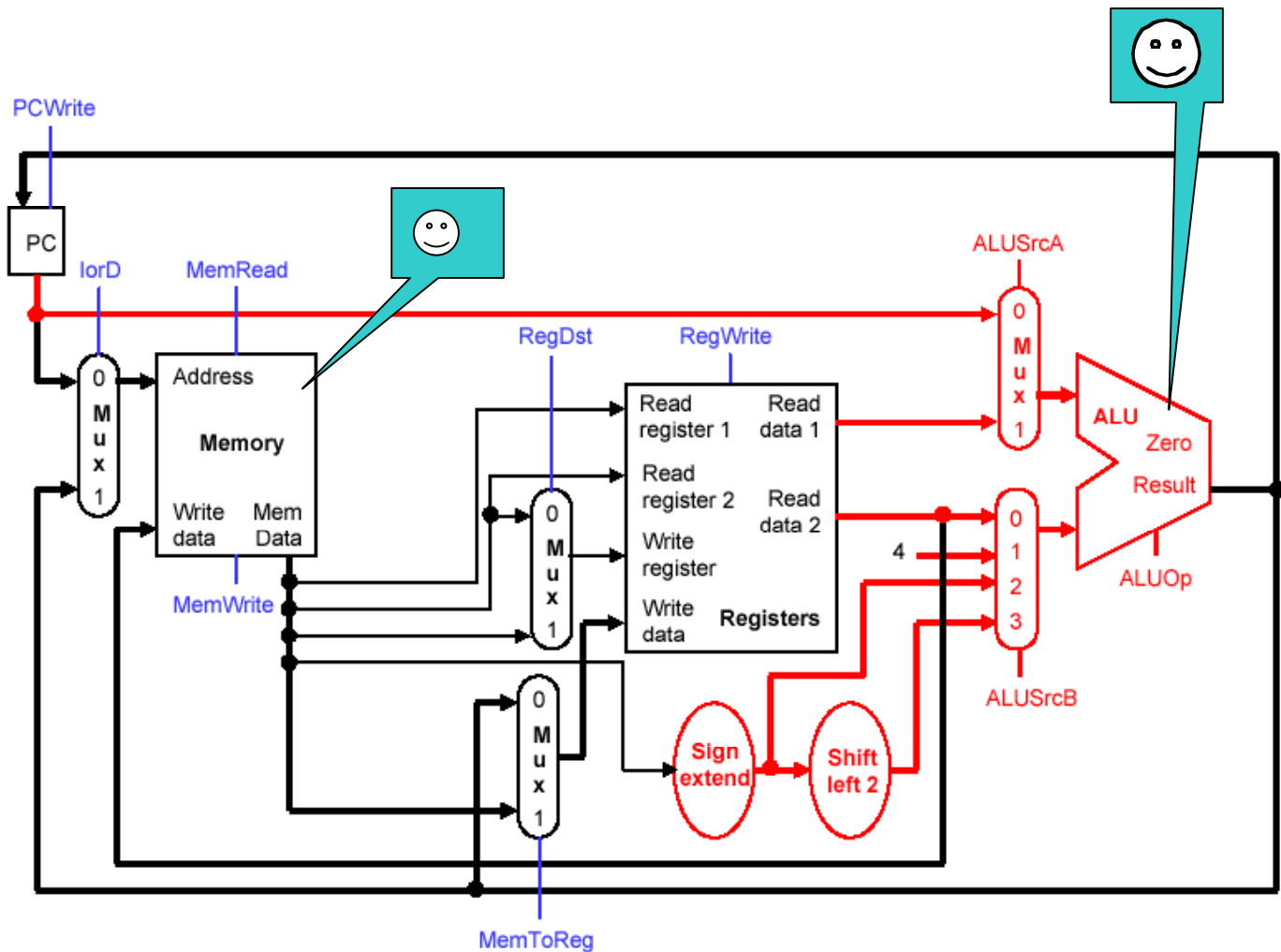


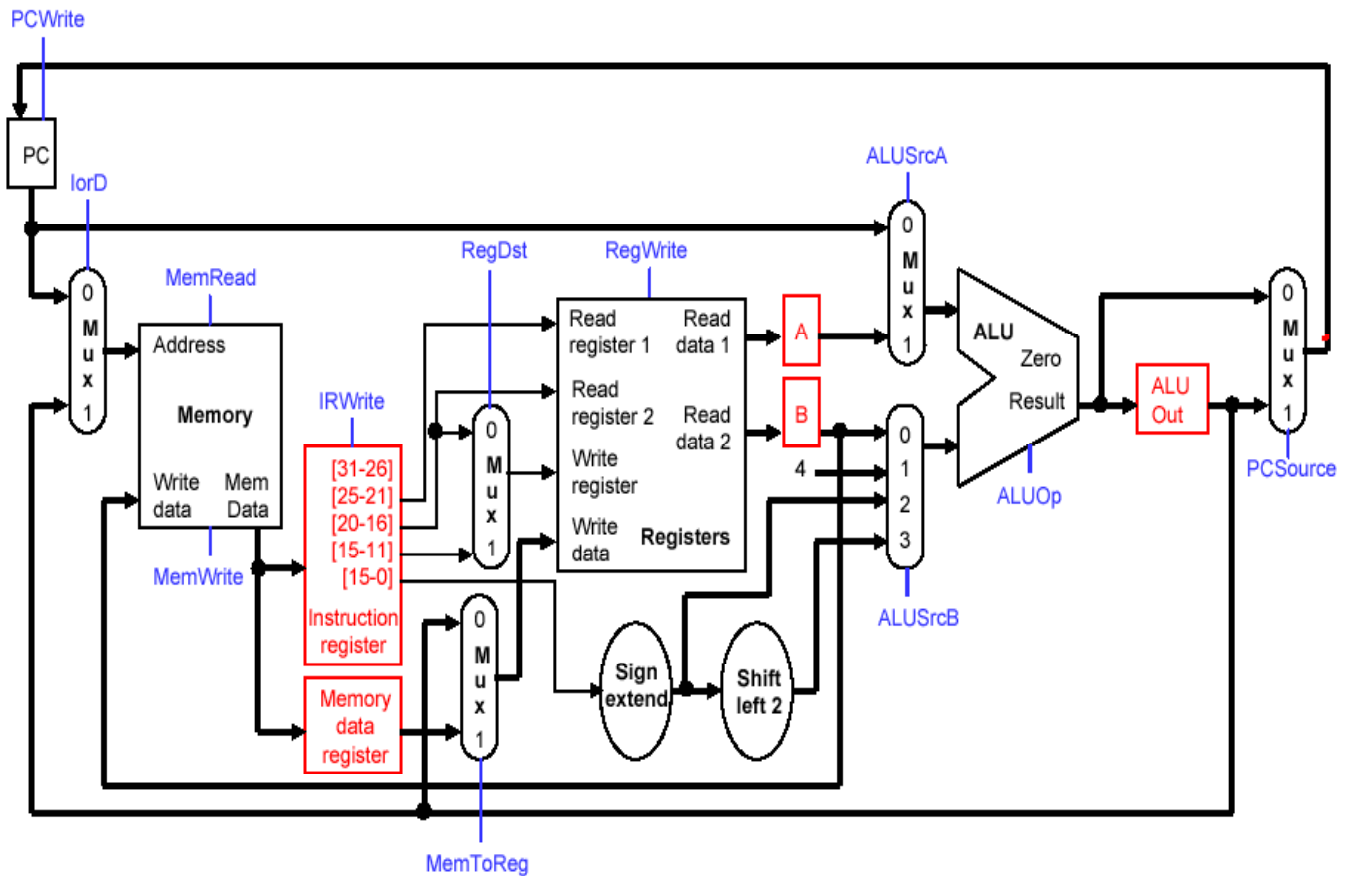
# Multi-cycle implementation of MIPS



Note that we have *eliminated two adders*, and used only *one memory unit* (so it is Princeton architecture) that contains both instructions and data. It is **not essential to have a single memory** unit, but it shows an **alternative design of the datapath**.

## Intermediate registers are necessary

In each cycle, a fraction of the instruction is executed. Note the red boxes that are the intermediate registers:



## Five stages of instruction execution

- Cycle 1. Instruction fetch and PC increment
- Cycle 2. Reading sources from the register file
- Cycle 3. Performing an ALU computation
- Cycle 4. Reading or writing (data) memory
- Cycle 5. Storing data back to the register file

## Why intermediate registers?

**Sometimes we need the output of a functional unit in a later clock cycle during the execution of an instruction.**

(Example: The **instruction word** fetched in stage 1 determines the destination of the **register write** in stage 5. The **ALU result** for an address computation in stage 3 is needed as the memory address for **lw** or **sw** in stage 4.)

**These outputs must be stored in intermediate registers for future use. Otherwise they will be lost by the next clock cycle.**

(Instruction read in stage 1 is saved in **Instruction register**. Register file outputs from stage 2 are saved in **registers A** and **B**. The **ALU output** will be stored in a register **ALUout**. Any data fetched from memory in stage 4 is kept in the **Memory data register MDR**.)

# The Five Cycles of MIPS

## (Instruction Fetch)

IR := Memory[PC]

PC := PC + 4

## (Instruction decode and Register fetch)

A := Reg[IR[25:21]], B := Reg[IR[20:16]]

ALUout := PC + sign-extend(IR[15:0])

## (Execute | Memory address | Branch completion)

Memory reference: ALUout := A + IR[15:0]

R-type (ALU): ALUout := A op B

Branch: if A = B then PC := ALUout

## (Memory access | R-type completion)

LW: MDR := Memory[ALUout]

SW: Memory[ALUout] := B

R-type: Reg[IR[15:11]] := ALUout

## (Writeback)

LW: Reg[[20:16]] := MDR

# Instruction execution review

---

- ❑ Executing a MIPS instruction can take up to five steps.

Step	Name	Description
Instruction Fetch	IF	Read an instruction from memory.
Instruction Decode	ID	Read source registers and generate control signals.
Execute	EX	Compute an R-type result or a branch outcome.
Memory	MEM	Read or write the data memory.
Writeback	WB	Store a result in the destination register.

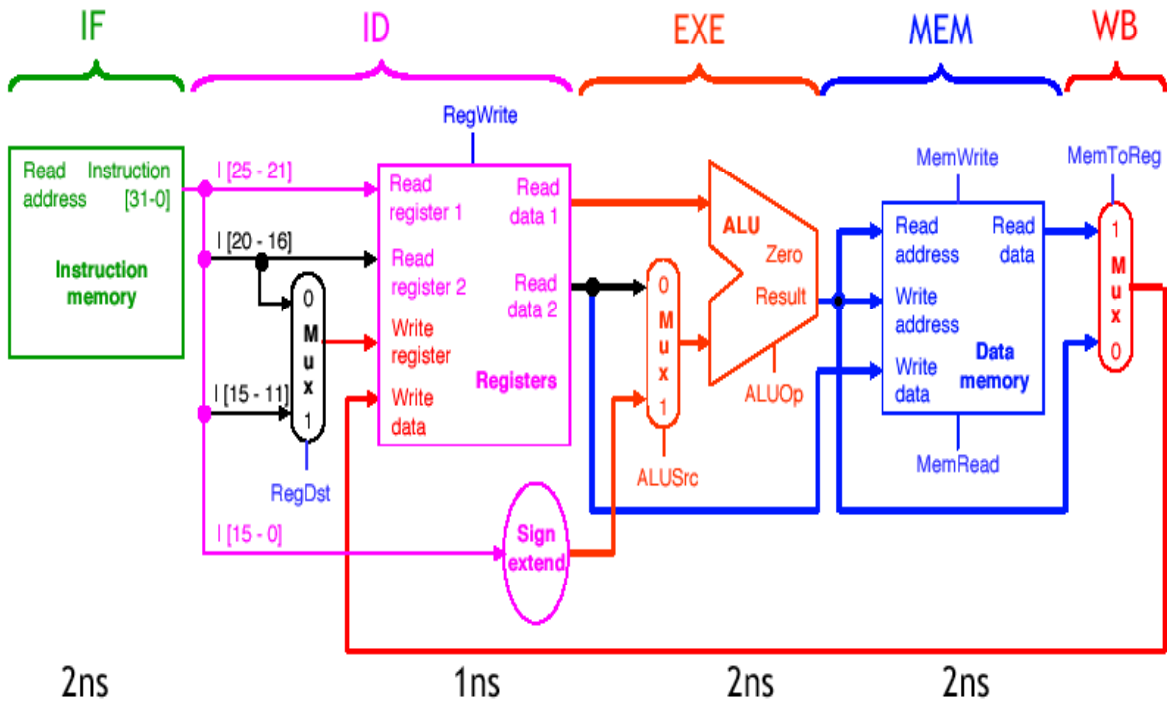
- ❑ However, as we saw, not all instructions need all five steps.

Instruction	Steps required				
beq	IF	ID	EX		
R-type	IF	ID	EX		WB
sw	IF	ID	EX	MEM	
lw	IF	ID	EX	MEM	WB

We will now study the implementation of a **pipelined version** of MIPS. We utilize the five stages of implementation for this purpose.

# Break datapath into 5 stages

- ❑ Each stage has its own functional units.
- ❑ Each stage can execute in 2ns
  - Just like the multi-cycle implementation



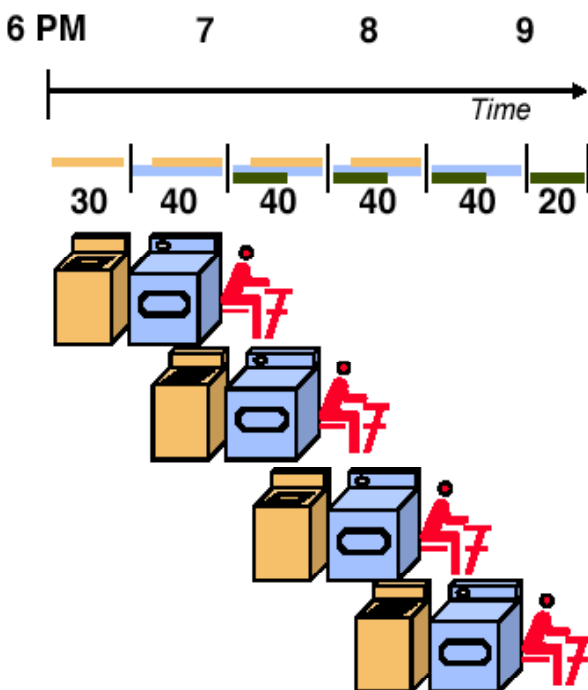
**The PC is not shown here, but can easily be added.**

Also, the buffer between the stages is not shown

The implementation of pipelining becomes “simpler” when you use **separate instruction memory** and **data memory** (We will explain it later). So we go back to our original Harvard architecture.

# Pipelined MIPS

**Why pipelining?** While a typical instruction takes 3-4 cycles (i.e. 3-4 CPI), a pipelined processor targets 1 CPI (and gets close to it).



- Pipelining doesn't help **latency** of single load, it helps **throughput** of entire workload
- Pipeline rate limited by **slowest** pipeline stage
- Multiple** tasks operating simultaneously using different resources
- Potential speedup = **Number pipe stages**
- Unbalanced lengths of pipe stages reduces speedup
- Time to **"fill"** pipeline and time to **"drain"** it reduces speedup

Pipelining in a laundromat -- Washer takes 30 minutes -- Dryer takes 40 minutes -- Folding takes 20 minutes. How does the laundromat example help with speeding up MIPS?