Design of the MIPS Processor

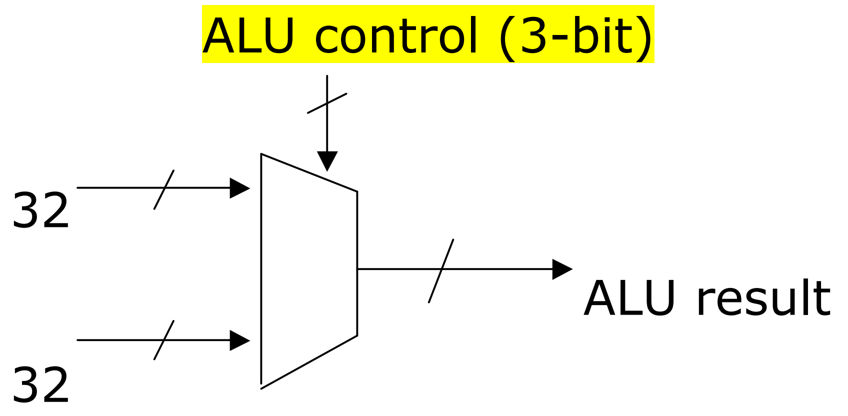
We will study the design of a simple version of MIPS that can support the following instructions:

- I-type instructions LW, SW
- R-type instructions, like ADD, SUB
- Conditional branch instruction BEQ
- J-type branch instruction J

The instruction formats

	6-bit	5-bit	5-bit	5-bit	5-bit	5-bit
LW	op	rs	rt	immediate		
SW	op	rs	rt	immediate		
ADD	op	rs	rt	rd	0	func
SUB	op	rs	rt	rd	0	func
BEQ	op	rs	rt	immediate		
J	op		address			

ALU control

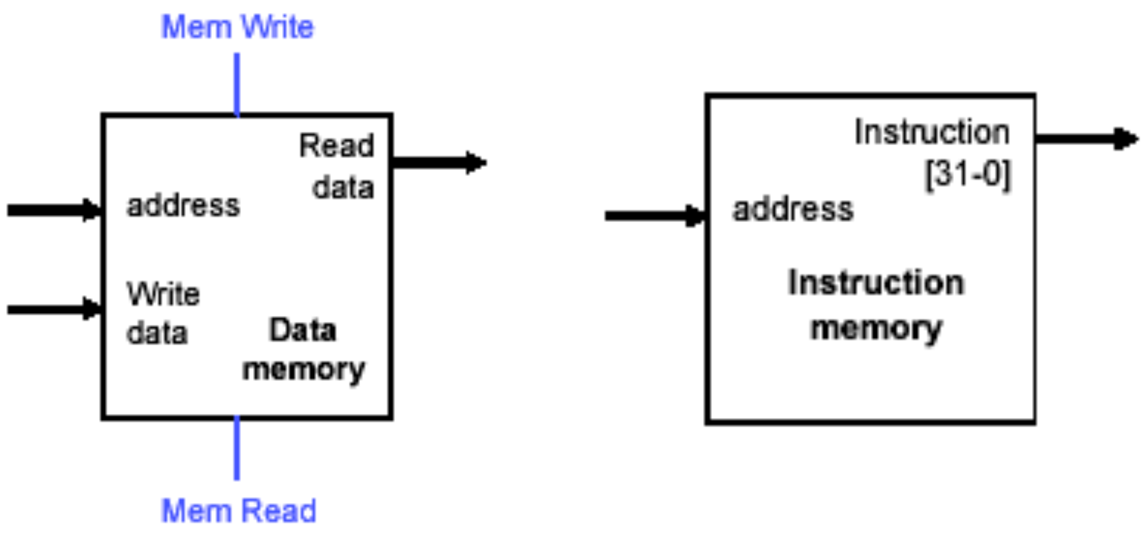


ALU control input	ALU function
000	AND
001	OR
010	add
110	sub
111	Set less than

How to generate the ALU control input? The control unit first generates this from the opcode of the instruction.

A single-cycle MIPS

We consider a simple version of MIPS that uses **Harvard architecture. Harvard architecture** uses separate memory for instruction and data.

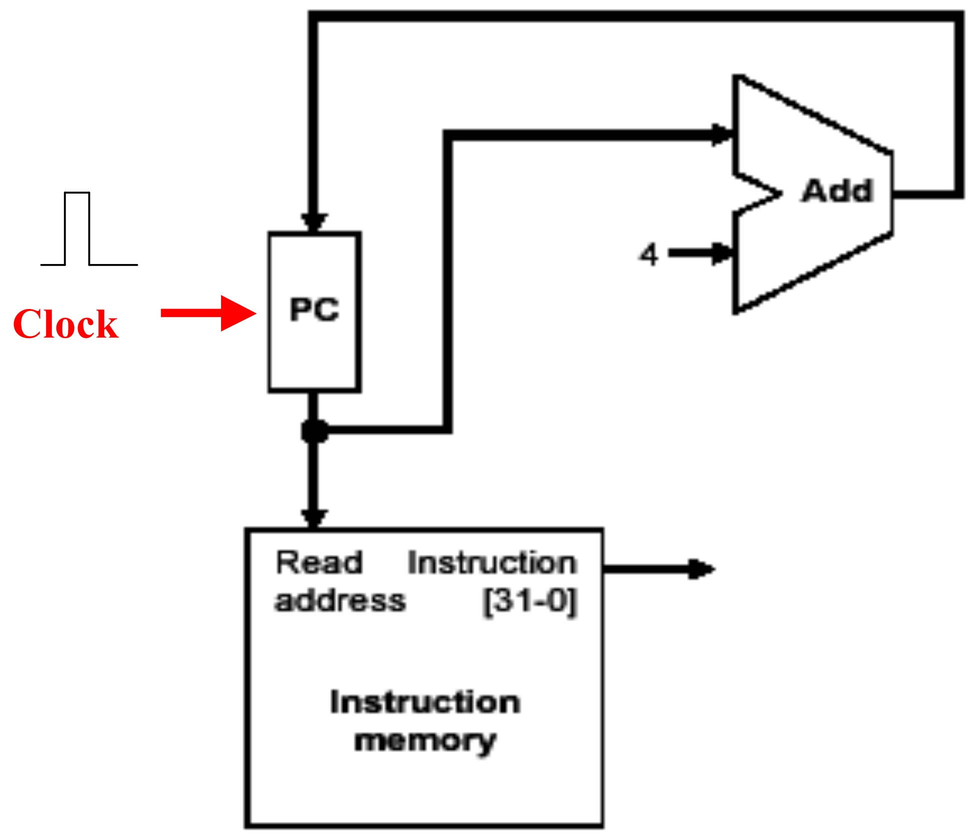


Instruction memory is read-only – a programmer cannot write into the instruction memory.

To read from the data memory, set Memory read =1

To write into the data memory, set Memory write =1

Instruction fetching



Each clock cycle fetches the instruction from the address specified by the PC, and increments PC by 4 at the same time.

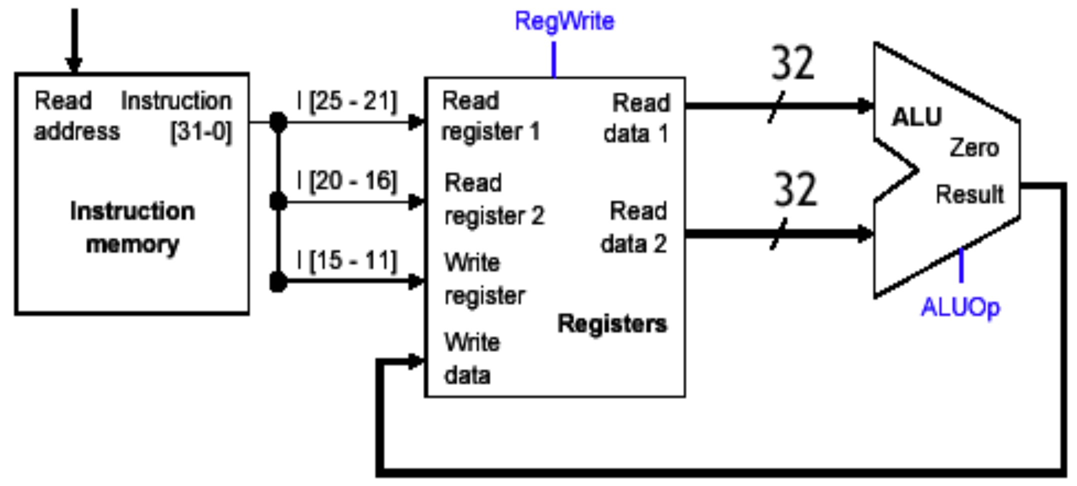
Executing R-type instructions

op	rs	rt	rd	shamt	func
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

add $s4, $t1, $t2

000000	01001	01010	10100	00000	1000000

This is the instruction format for the R-type instructions.



op	rs	rt	rd	shamt	func
31 26	25 21	20 16	15 11	10 6	5 0

Here are the steps in the execution of an R-type instruction:

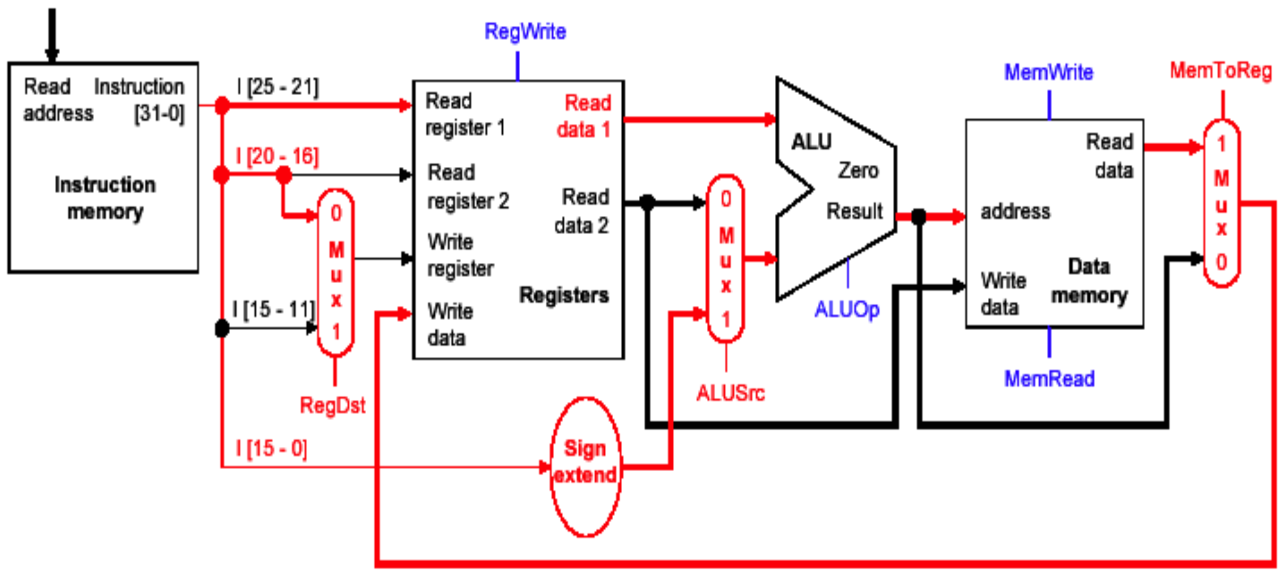
- ♦ Read instruction
- ♦ Read source registers rs and rt
- ♦ ALU performs the desired operation
- ♦ Store result in the destination register rd.

Q. Why should all these be completed in a single cycle?

Executing lw, sw instructions

These are I-type instructions.

lw $t0, -4($sp)	100011	11101	01000	1111 1111 1111 1100
	31 26	25 21	20 16	15 0
sw $a0, 16($sp)	101011	11101	00100	0000 0000 0001 0000
	op	rs	rt	address



Try to recognize the steps in the execution of lw and sw.

Design of the MIPS Processor (contd)

First, revisit the datapath for add, sub, lw, sw. We will augment it to accommodate the beq and j instructions.

Execution of branch instructions

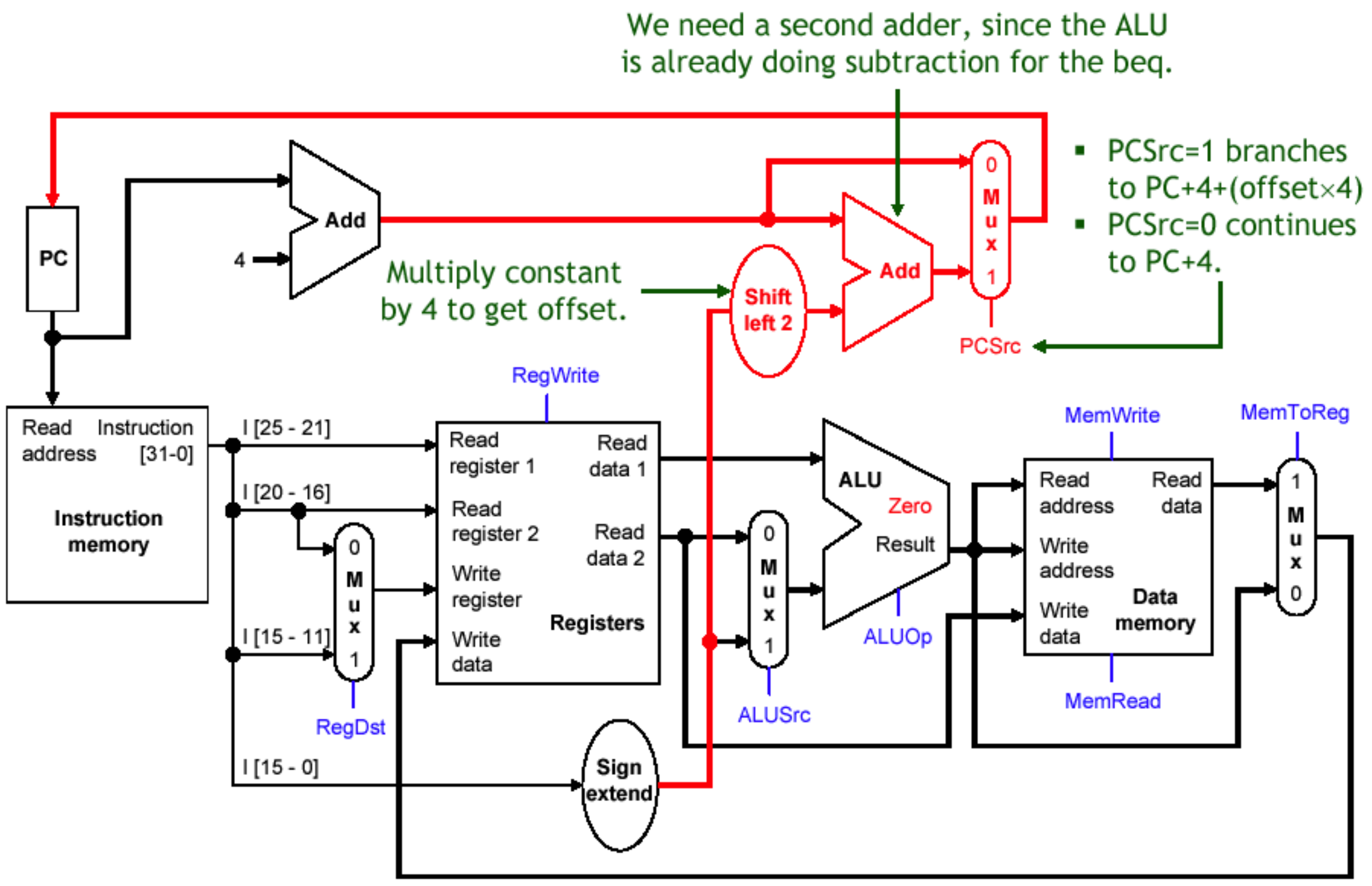
```
    beq $at, $zero, L
    add $v1, $v0, $zero
    add $v1, $v1, $v1
    j      somewhere
L:  add $v1, $v0, $v0
```

000100	00001	00000	0000 0000 0000 0011
op	rs	rt	address

Offset= 3x4=12

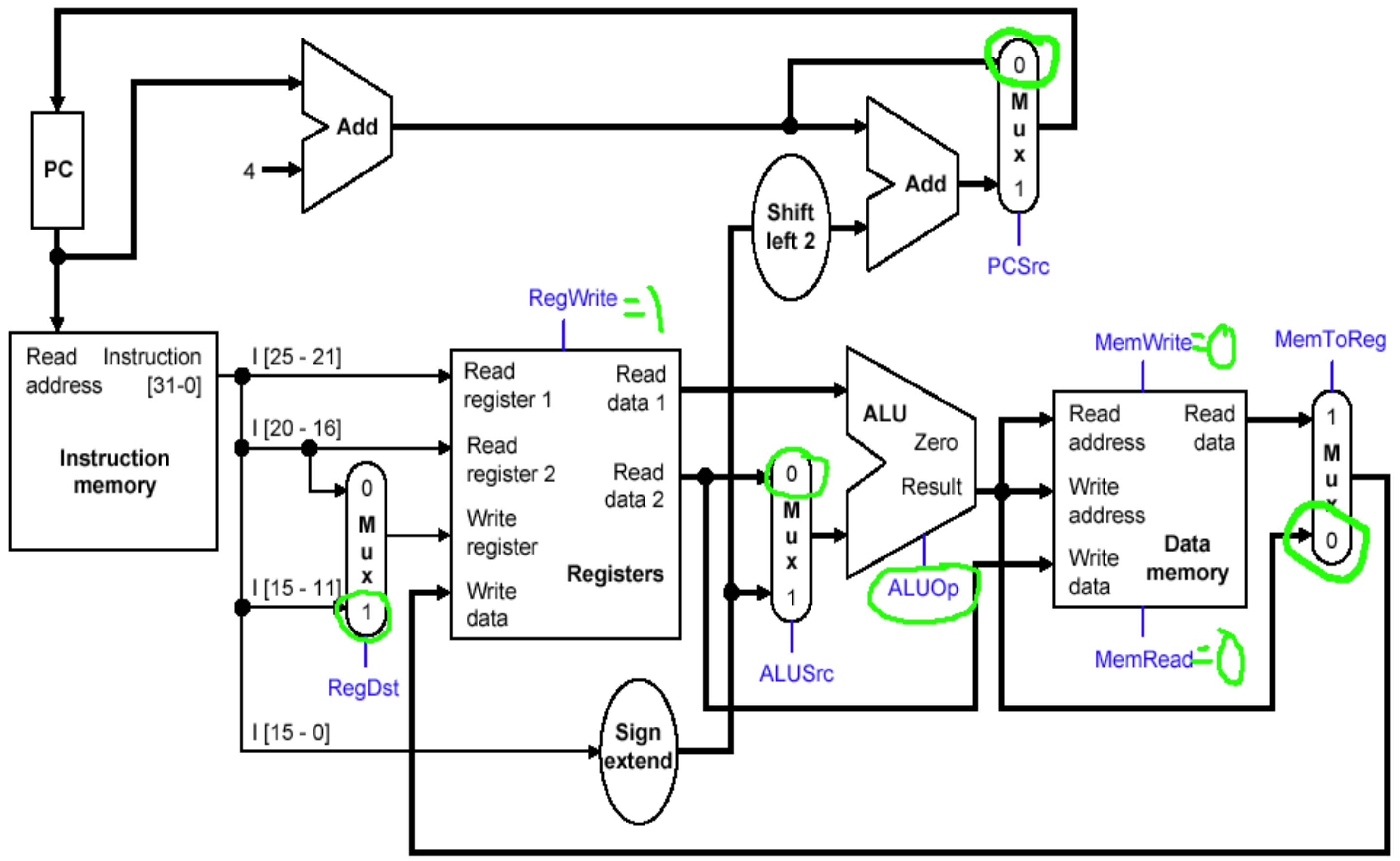
The offset must be added to the next PC to generate the target address for branch.

The modified version of MIPS



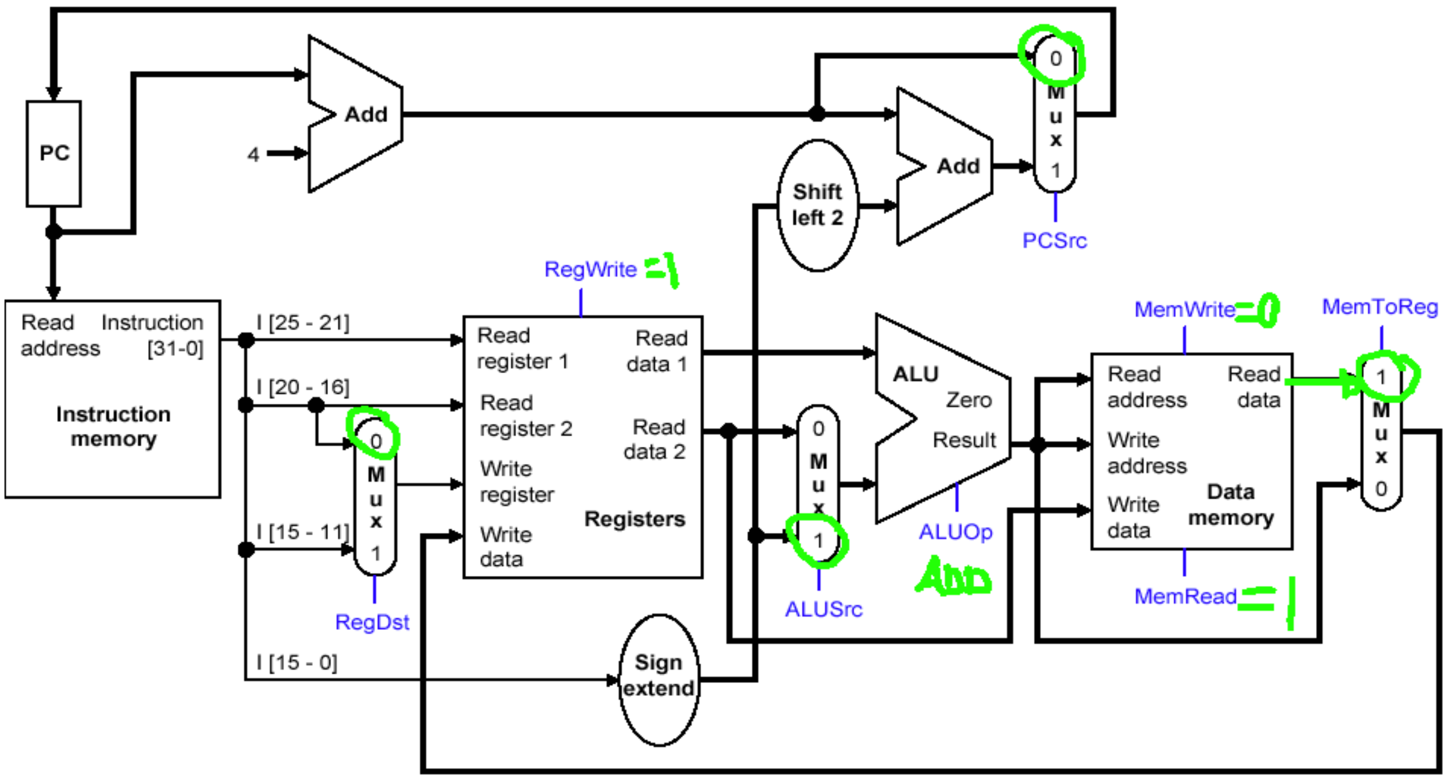
The final datapath for single cycle MIPS. Find out which paths the signal follow for lw, sw, add and beq instructions

Executing R-type instructions



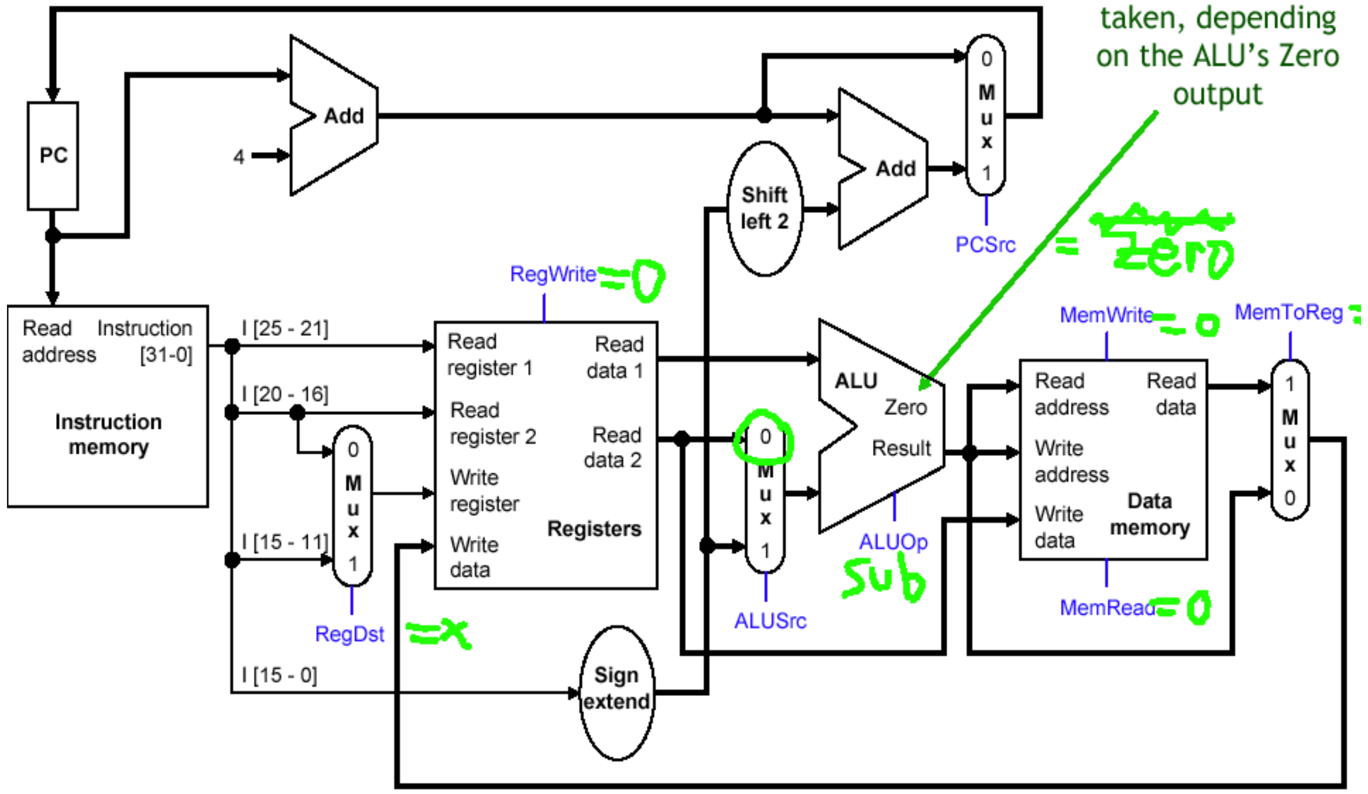
The ALUop will be determined by the value of the opcode field and the function field of the instruction word

Executing LW instruction



Executing beq instruction

The branch may or may not be taken, depending on the ALU's Zero output



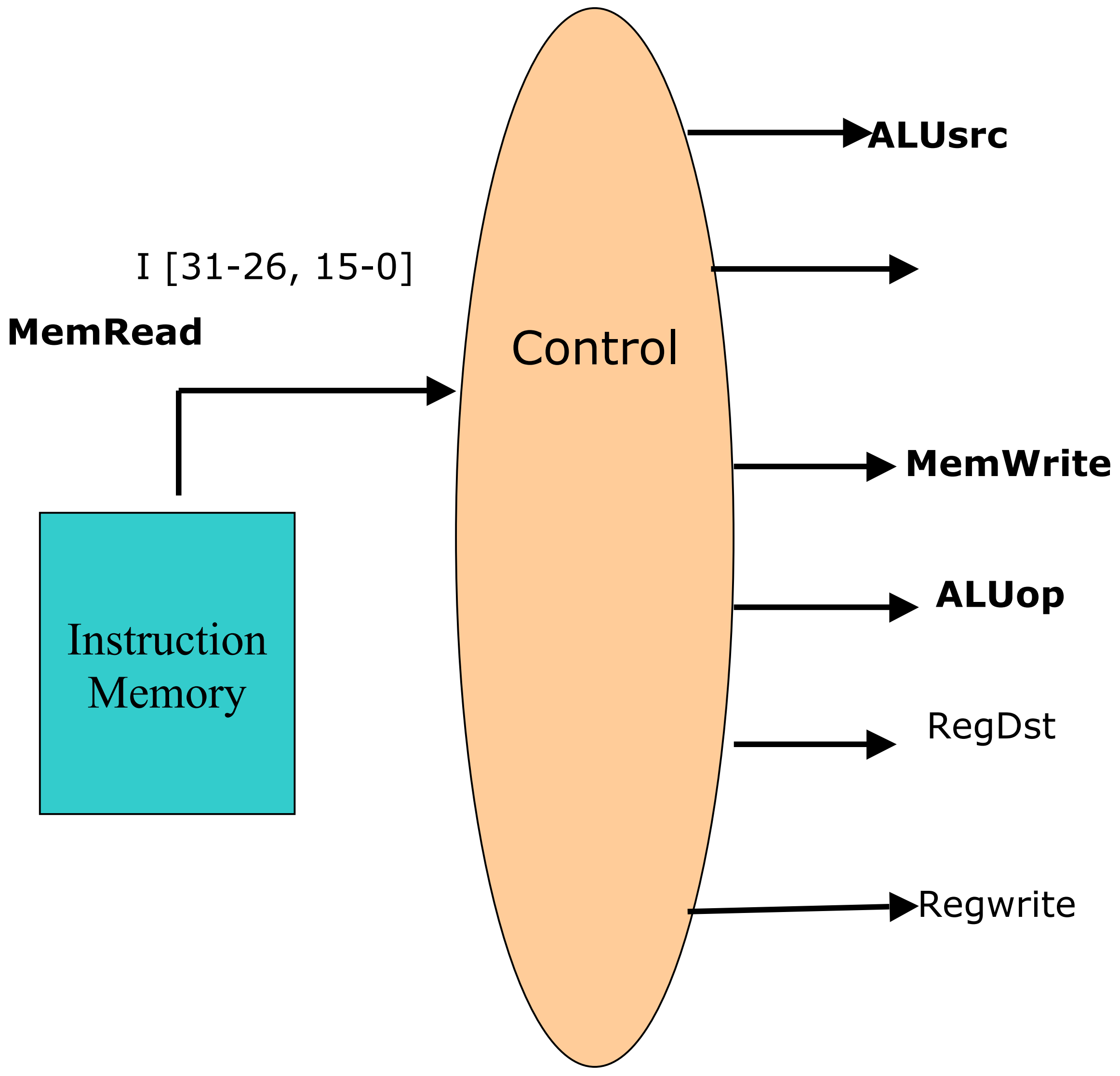
Control signal table

This table summarizes what control signals are needed to execute an instruction. The set of control signals vary from one instruction to another.

Operation	RegDst	RegWrite	ALUSrc	ALUOp	MemWrite	MemRead	MemToReg
add	1	1	0	010	0	0	0
sub	1	1	0	110	0	0	0
and	1	1	0	000	0	0	0
or	1	1	0	001	0	0	0
slt	1	1	0	111	0	0	0
lw	0	1	1	010	0	1	1
sw	X	0	1	010	1	0	X
beq	X	0	0	110	0	0	X

How to implement the control unit? Recall how to convert a truth table into a logical circuit! The control unit implements the above truth table.

The Control Unit



All control signals are not shown here

1-cycle implementation is not used

Why? Because the length of the clock cycle will always be determined by the slowest operation (lw, sw) even if the data memory is not used.

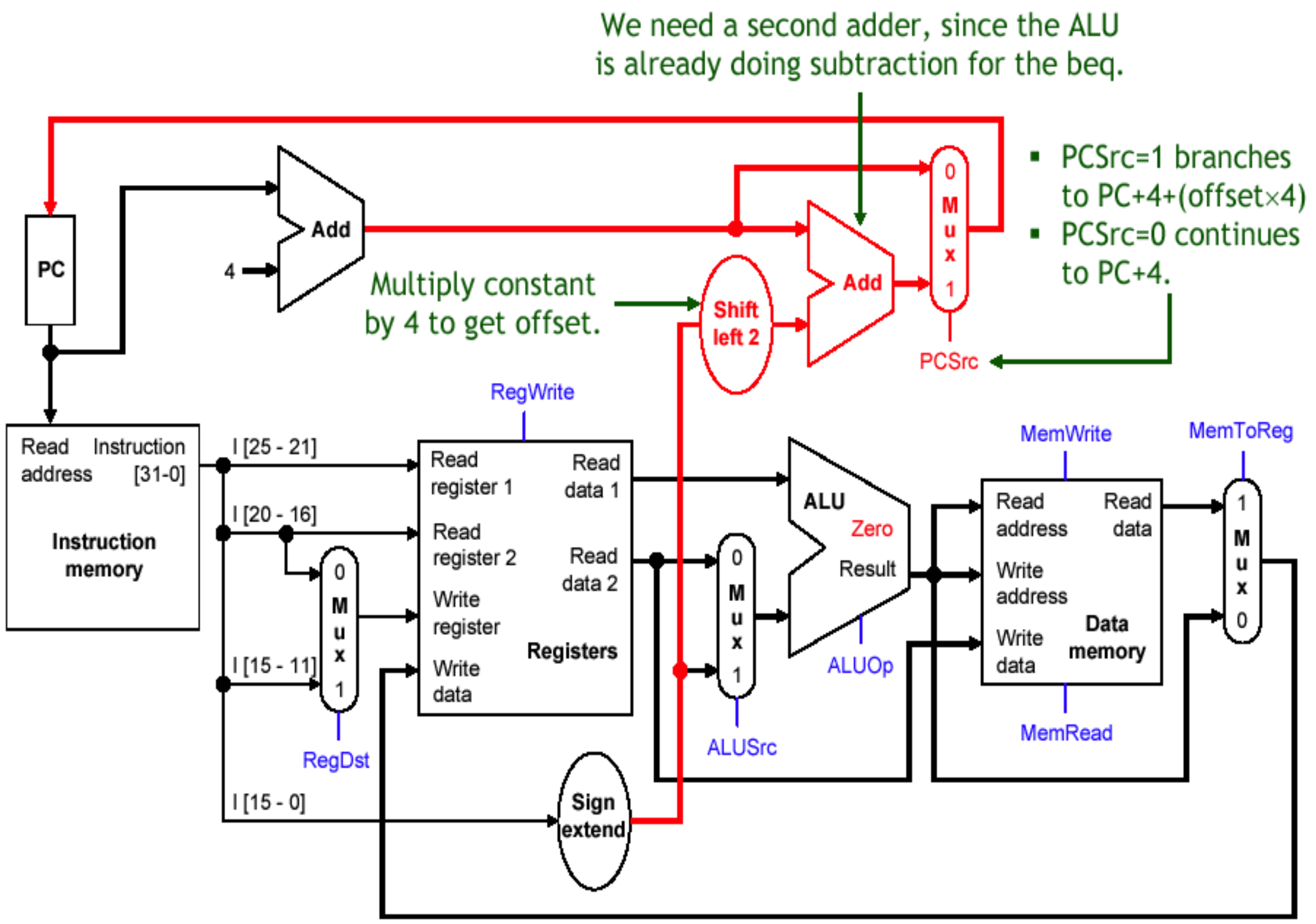
Practical implementations use multiple cycles per instruction, which fixes some shortcomings of the 1-cycle implementation.

- Faster instructions (R-type) are not held back by the slower instructions (lw, sw)
- The clock cycle time can be decreased, i.e. faster clock can be used
- Eventually simplifies the implementation of pipelining, the universal speed-up technique.

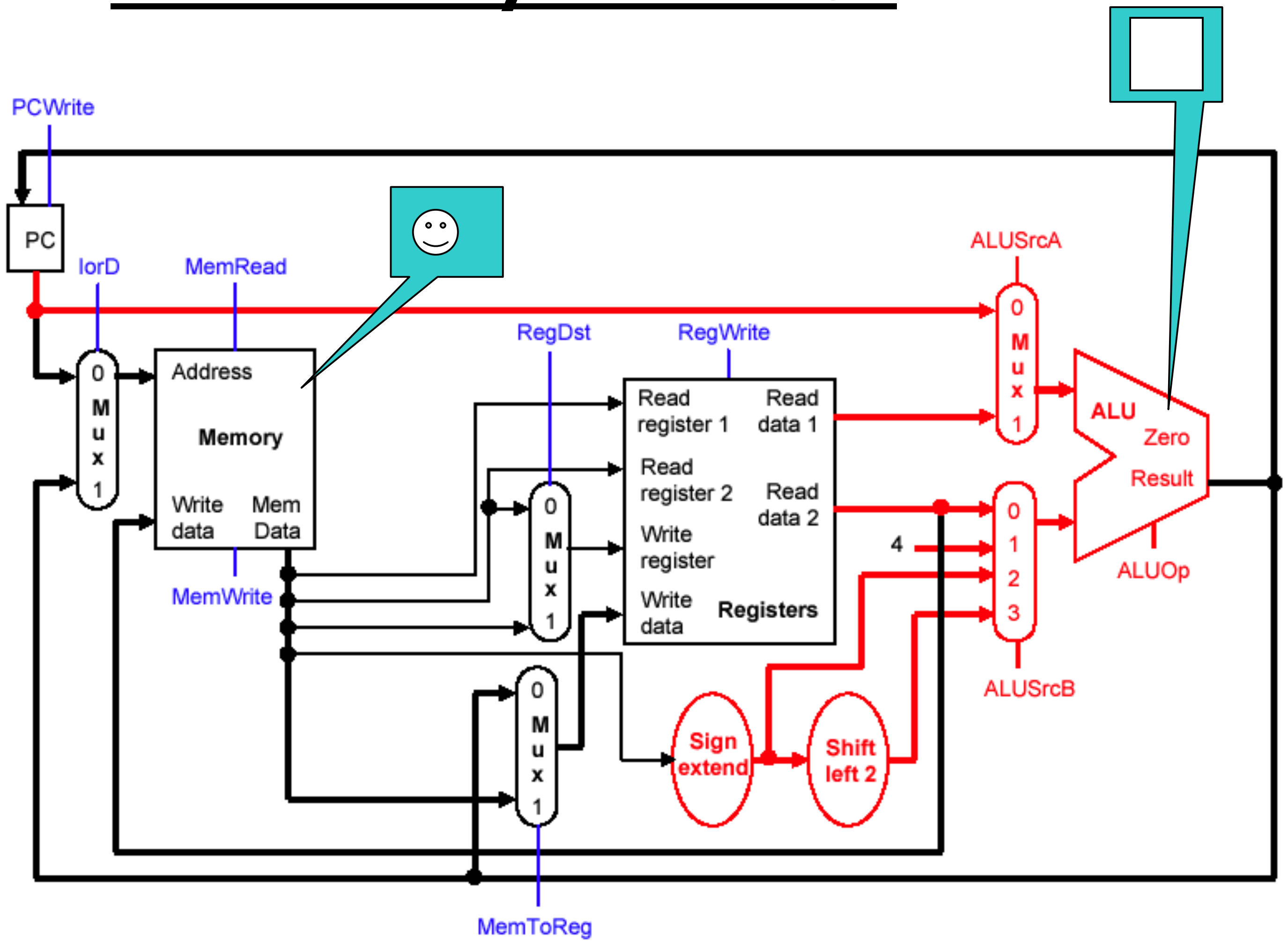
This requires some changes in the datapath

Multi-cycle implementation of MIPS

First, revisit the 1-cycle version



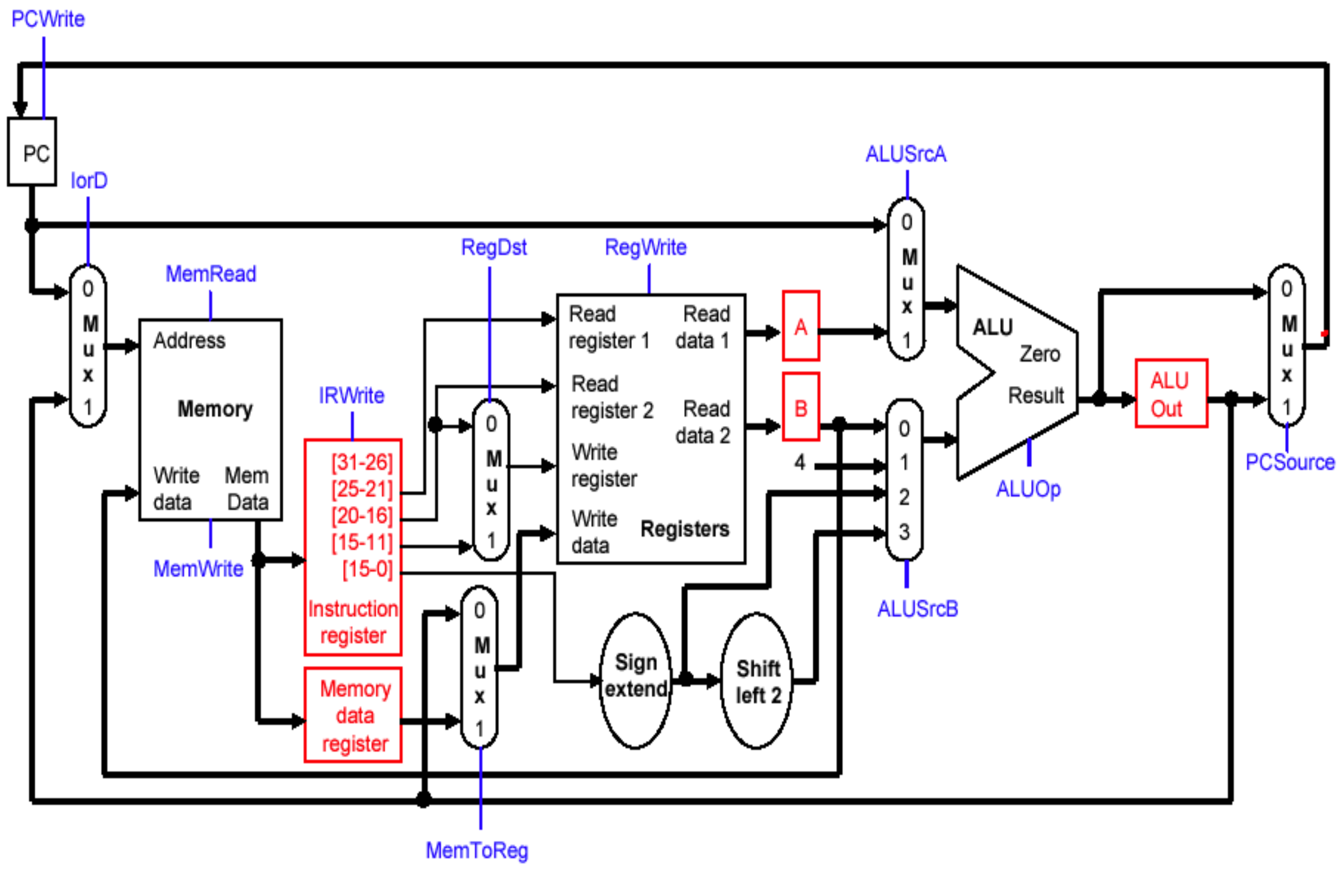
The multi-cycle version



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



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 after 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)

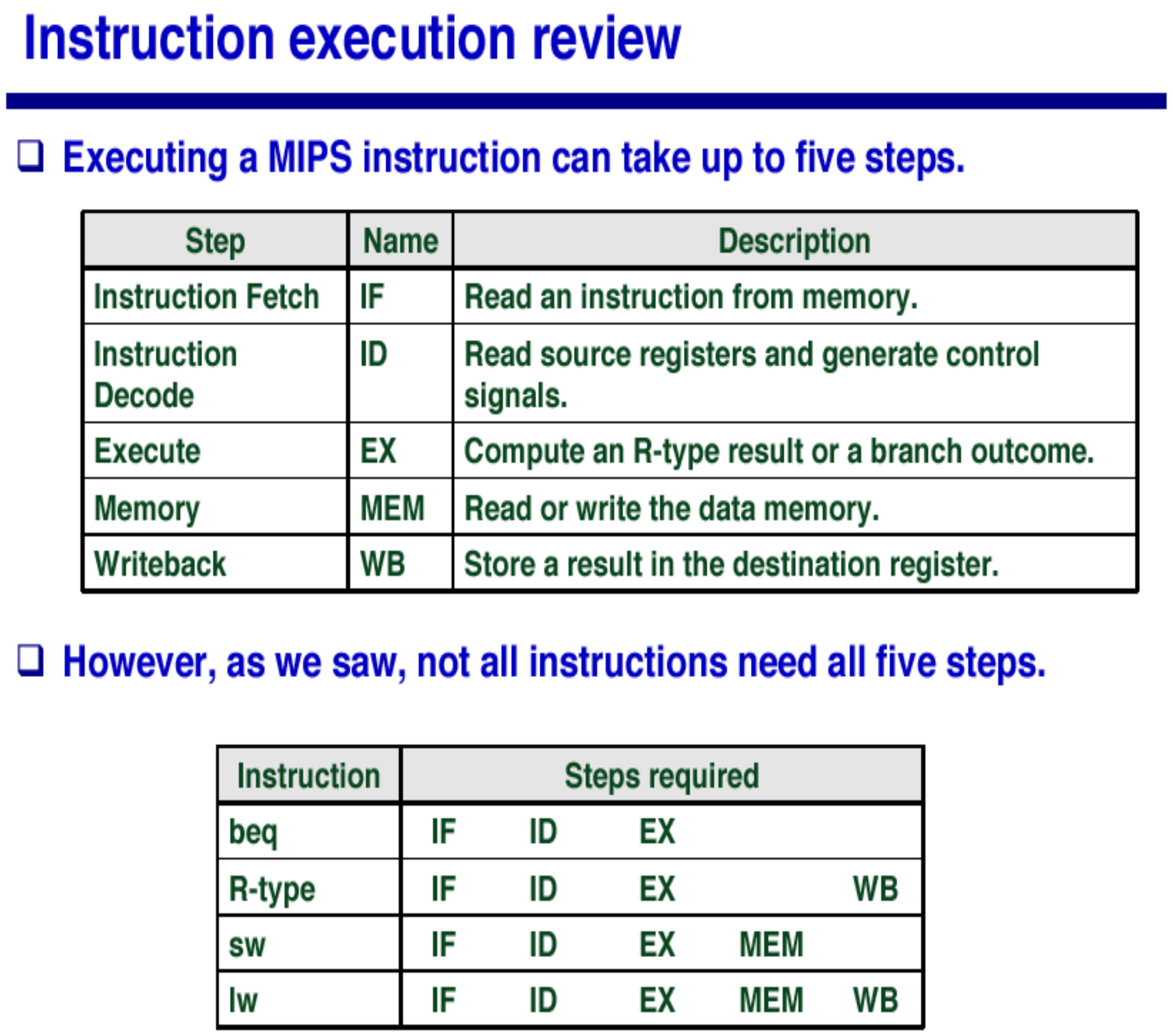
LW: MDR:= Memory[ALUout]

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.

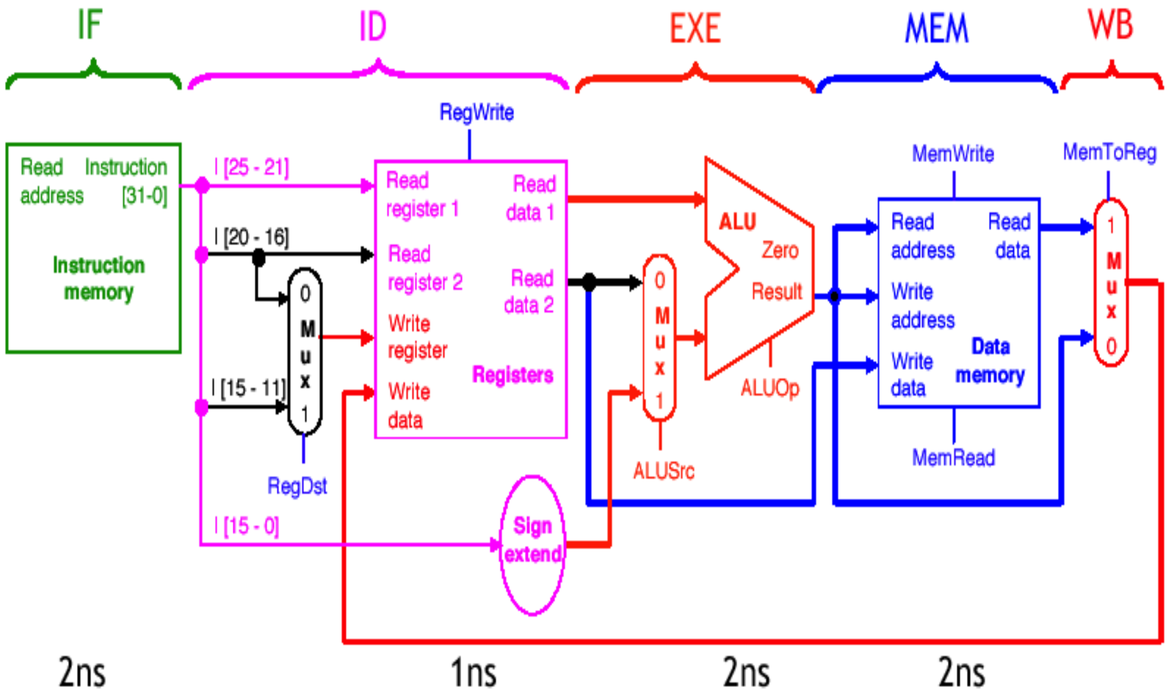
- However, as we saw, not all instructions need all five steps.

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



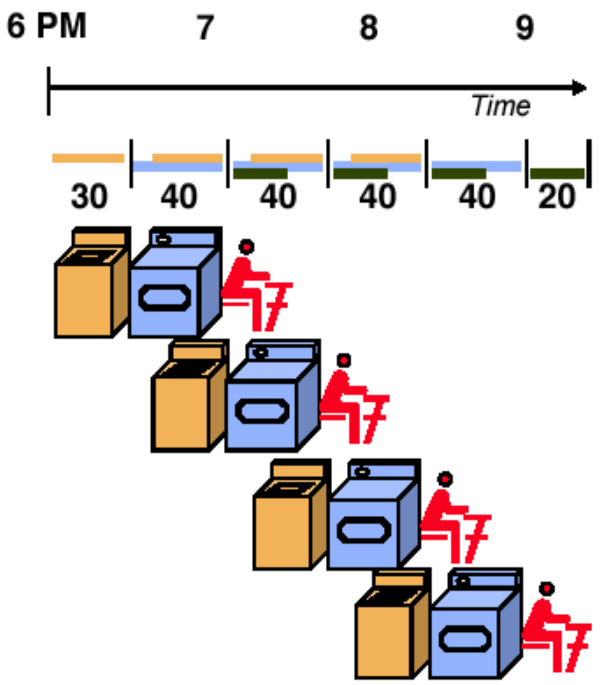
The PC is not shown here, but can easily be added.

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?