

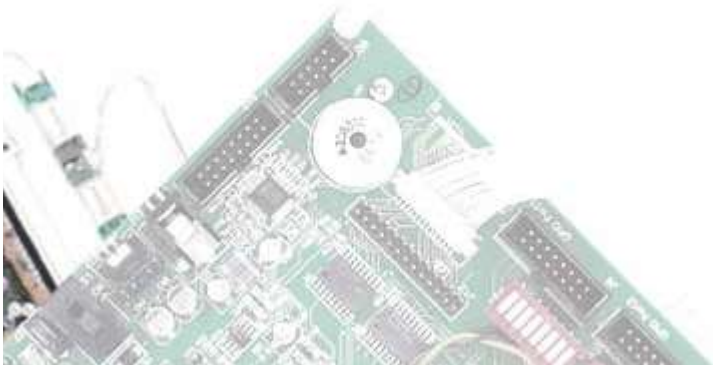


---

# Computer Architecture

```
INFOR_HEADER_T  
  
IF (FH_Open() != FH_SUCCESS) {  
    return(FTL_MEDIAERR);  
}  
  
/* SKIP THE BLOCK 0/1000 */  
for (current_block = 0; current_block < NO_OF_BLOCK; current_block++) {  
    FH_Erase(current_block);  
}  
...
```

## Ch. 4 The Processor (1)



# Combinational Logic Elements

## □ Stateless logic

- No embedded state (memory)
- Output fully dependent on inputs
- Any function possible
  - AND, OR, NAND, NOR, XOR, NOT...



AND



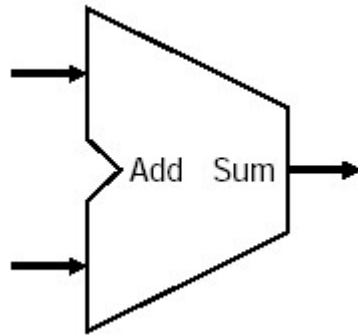
OR



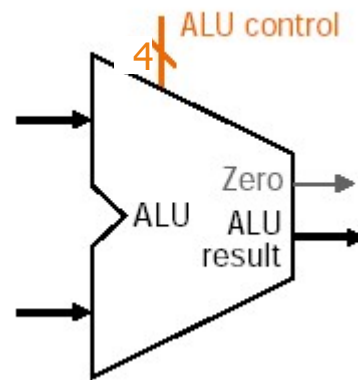
NOT

# Combinational Logic Elements

## □ Adder



## □ ALU

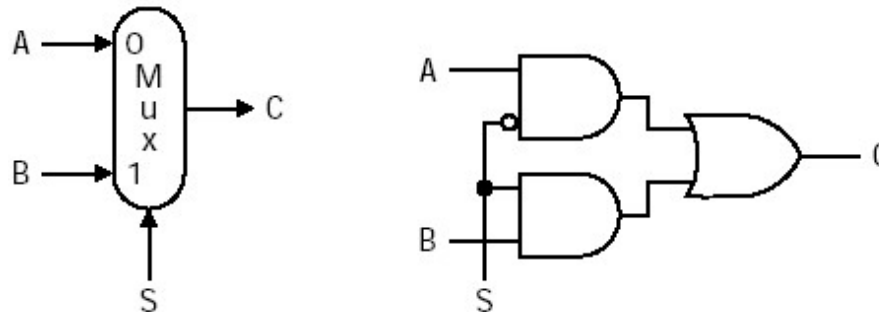


ALU control input	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set on less than

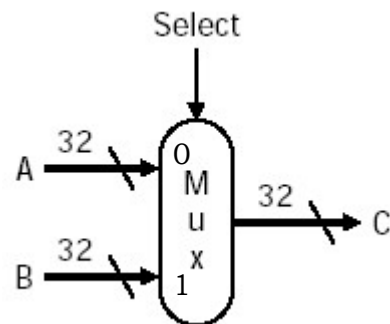
# Combinational Logic Elements

## ❑ Multiplexor

- A two-input multiplexor



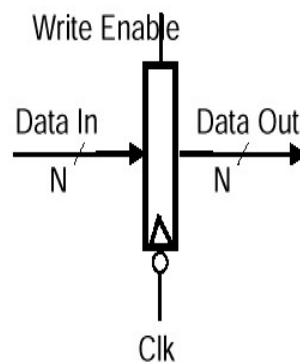
- A 32-bit wide 2-to-1 multiplexor



# Storage Elements (1)

## □ Register

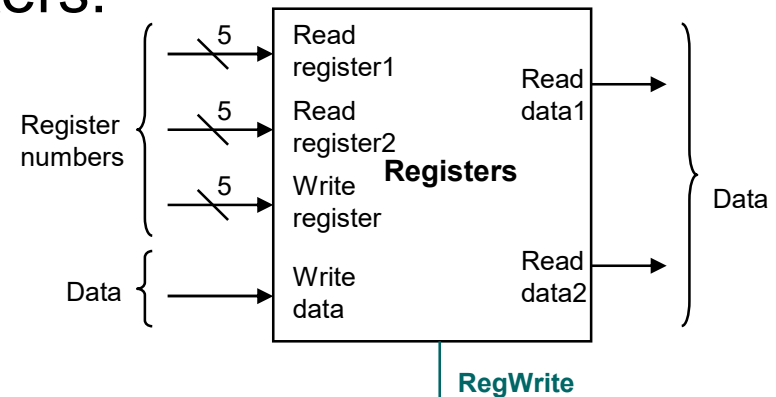
- Similar to the D Flip Flop except
  - N-bit input and output
  - Write Enable input
- Write Enable :
  - 0 : Data Out will not change
  - 1 : Data Out will become Data in
- Stored data changes only on falling clock edge!



# Storage Elements (2)

## ❑ Register File consists of 32 registers:

- Two 32-bit output busses:
  - Read data1 and Read data2
- One 32-bit input bus: Write data
- Register 0 hard-wired to value 0



## ❑ Register is selected by:

- Read register1 selects the register to put on Read data1
- Read register2 selects the register to put on Read data2
- Write register selects the register to be written via write data when  $\text{RegWrite} = 1$

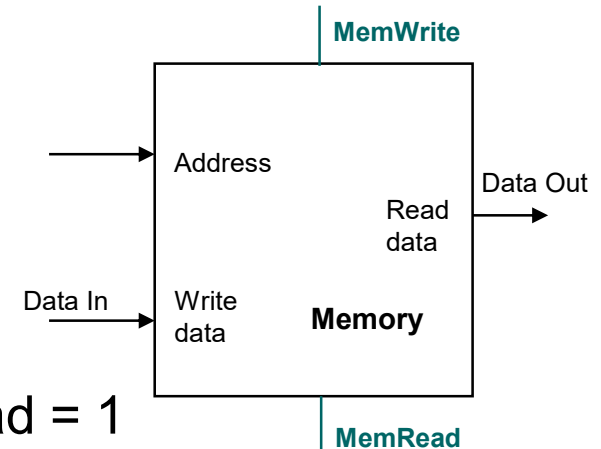
## ❑ Clock input (CLK)

- The CLK input is a factor only for write operation (data changes only on falling clock edge)

# Storage Elements (3)

- Memory has two busses:

- One output bus : Read data (Data Out)
- One input bus : Write data (Data In)



- Address selects

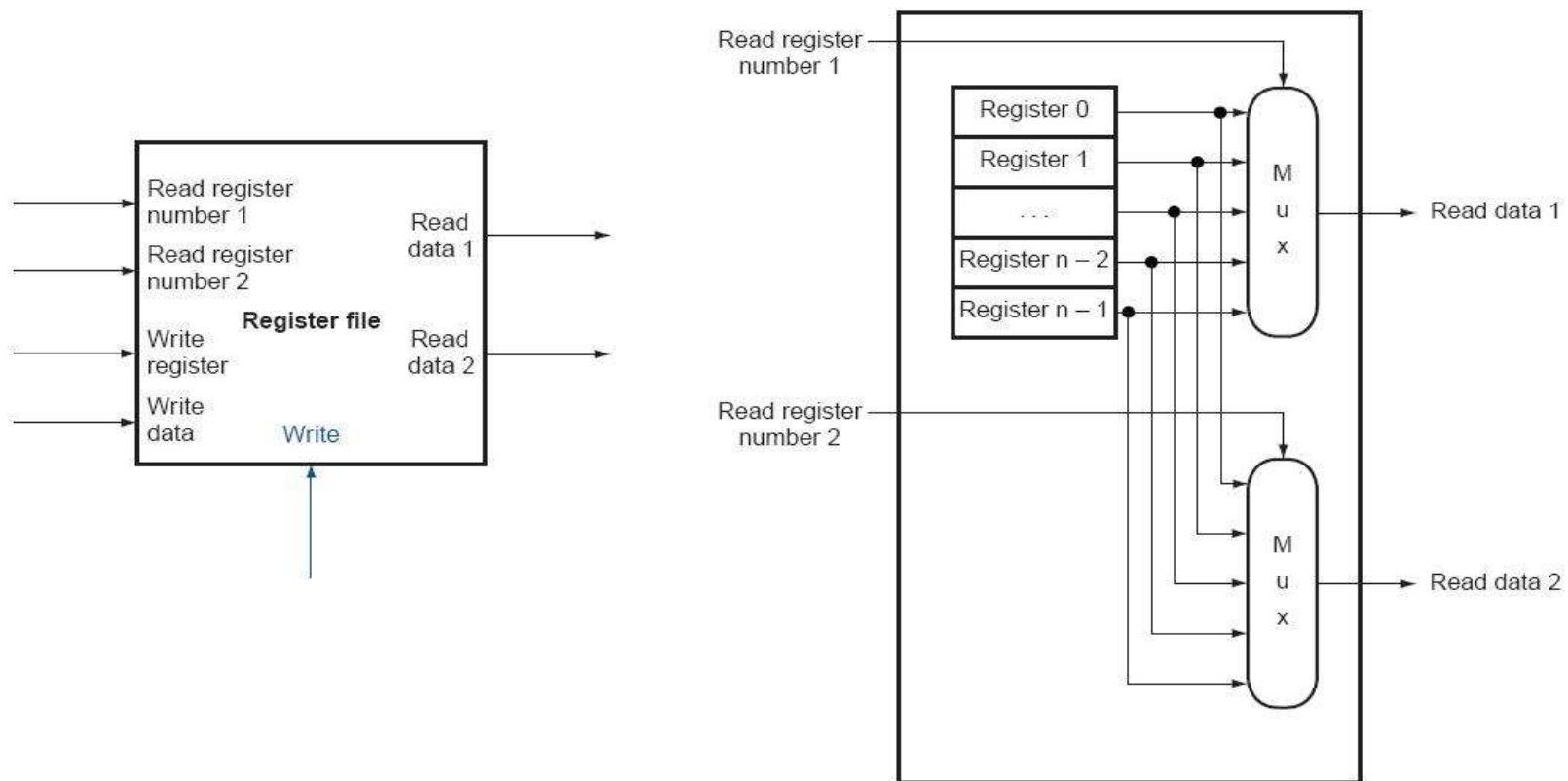
- the word to put on Data Out when MemRead = 1
- the word to be written via the Data In when MemWrite = 1

- Clock input (CLK)

- The CLK input is a factor only for write operation (data changes only on falling clock edge)

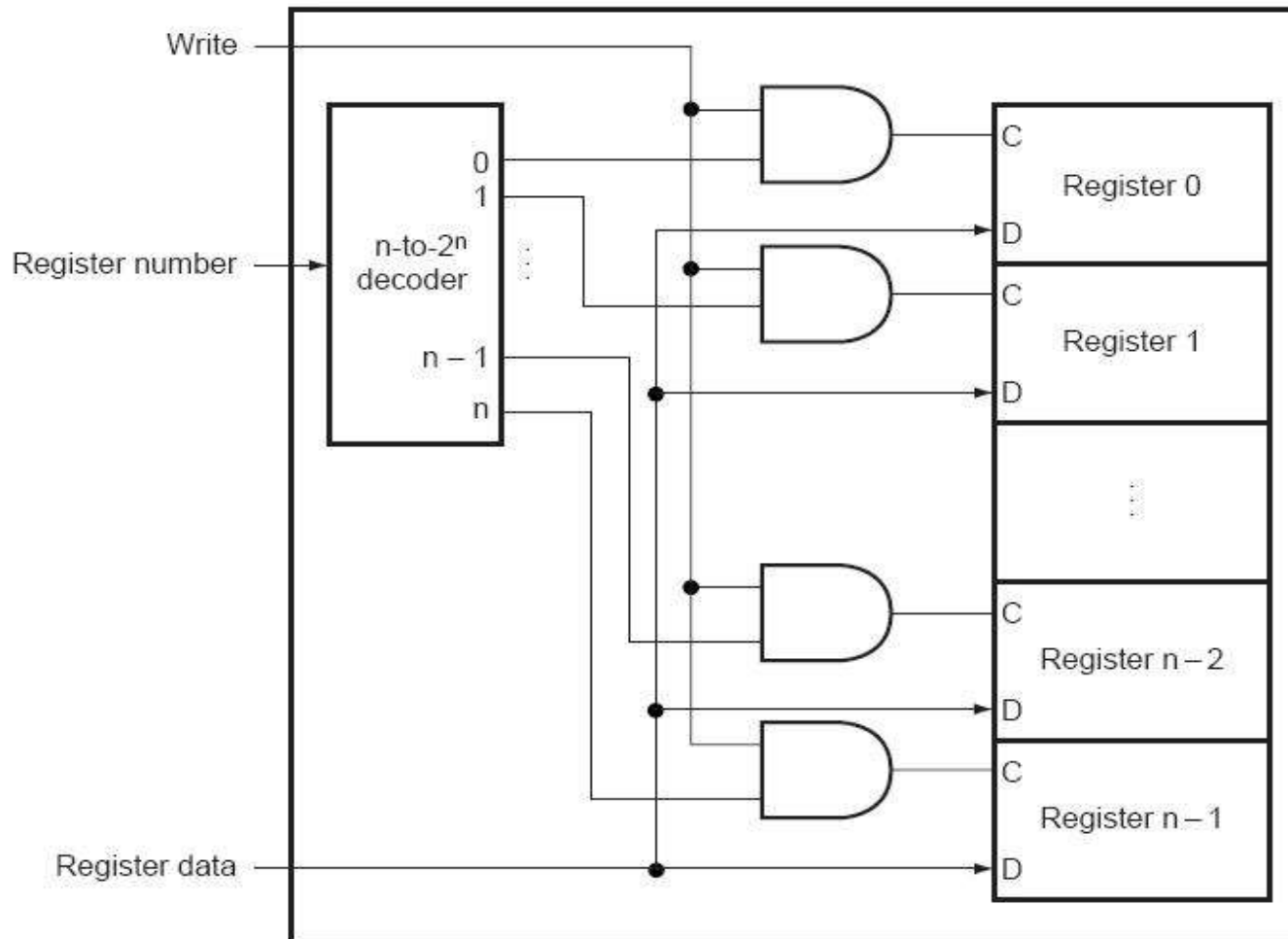
# Register File (1) : Output

- Built with D flip-flops





# Register File (2) : Input



# Instruction Execution (Steps) (1)



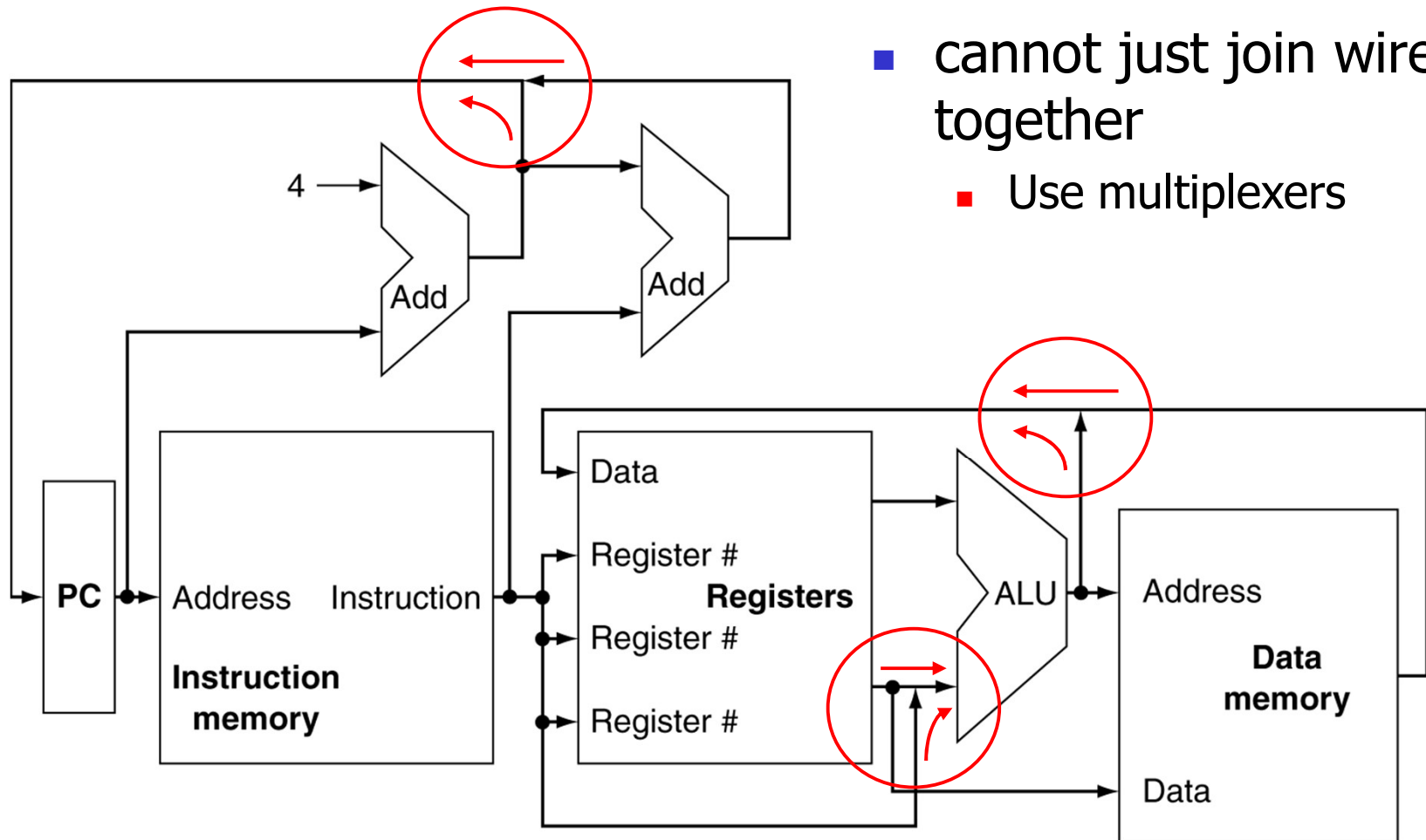
- ❑ First 2 Steps (of Every Instruction)
  - 1<sup>st</sup> step (Instruction fetch)
    - send PC to memory
    - send READ signal to memory
    - $PC = PC + 4$
  - 2<sup>nd</sup> step (Decode)
    - opcode decoding
    - register prefetch

# Instruction Execution (Steps) (2)



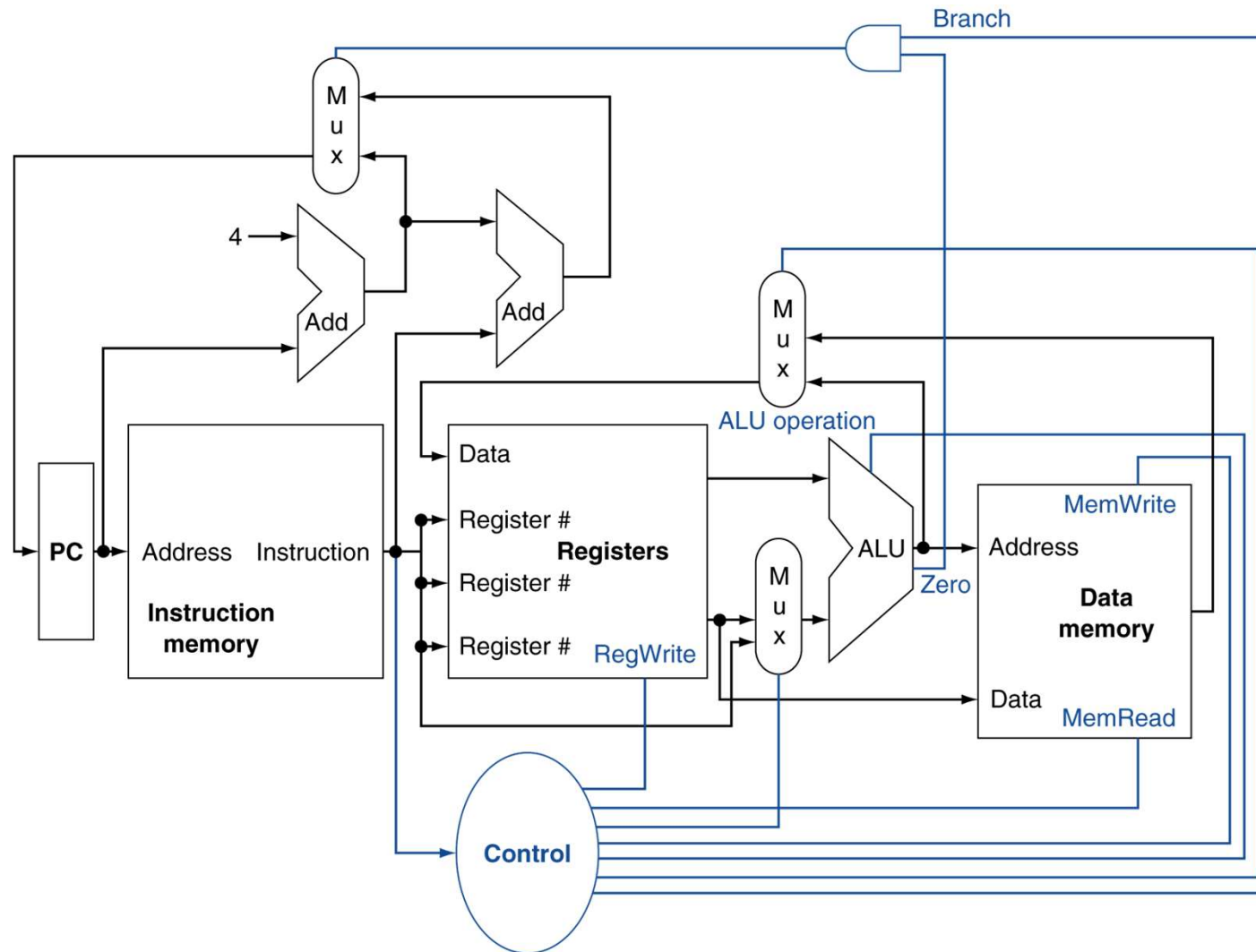
- ❑ 3<sup>rd</sup> step
  - Memory-reference instructions
    - Use ALU for an effective address calculation
  - Arithmetic-logical instructions
    - Use ALU for the operation execution
  - Branch instructions
    - Use ALU for comparison
- ❑ Final step
  - Memory-reference instructions
    - Store: access the memory to write data
    - Load: access the memory to read data
  - Arithmetic-logical instructions
    - Write the data from the ALU back into a register
  - Branch instructions
    - Change PC based on the comparison

# Abstract view of the MIPS



- cannot just join wires together
- Use multiplexers

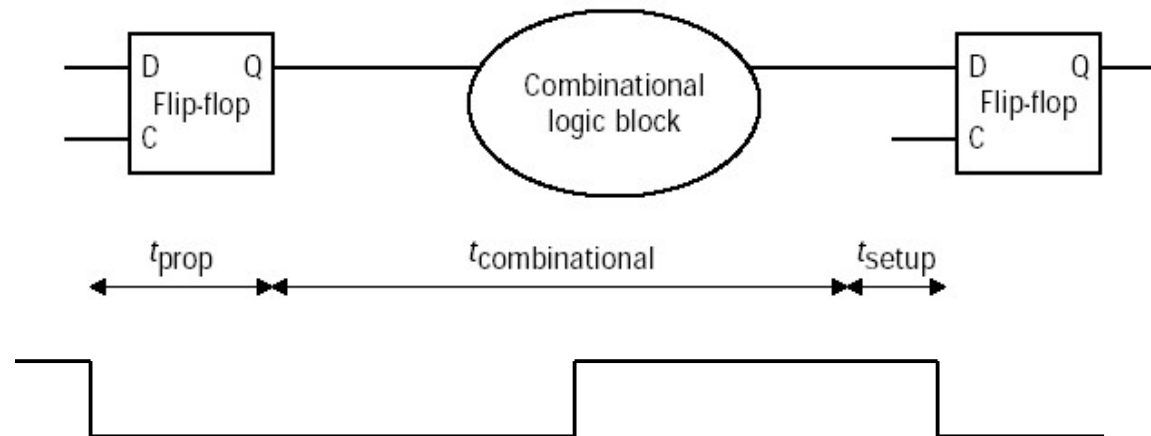
# Basic implementation of the MIPS subset



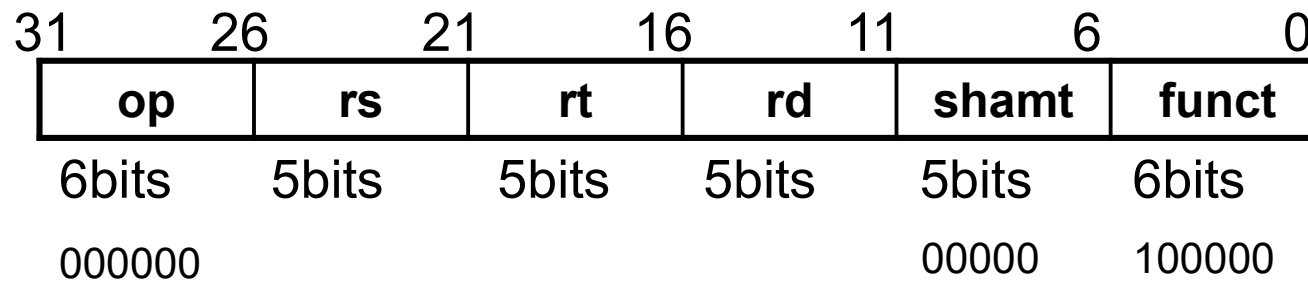
# Clocking

- All storage elements clocked by the same clock edge
  - Edge-triggered clocking (falling clock edge)
  - One clock period per clock cycle
  - Design always works if the clock is “slow enough”

$$\text{Cycle Time} = t_{\text{prop}} + t_{\text{combinational}} + t_{\text{setup}}$$



# add Instruction



□ add rd, rs, rt

$IR \leftarrow \text{mem}[PC];$

$R[rd] \leftarrow R[rs] + R[rt];$

$PC \leftarrow PC + 4;$

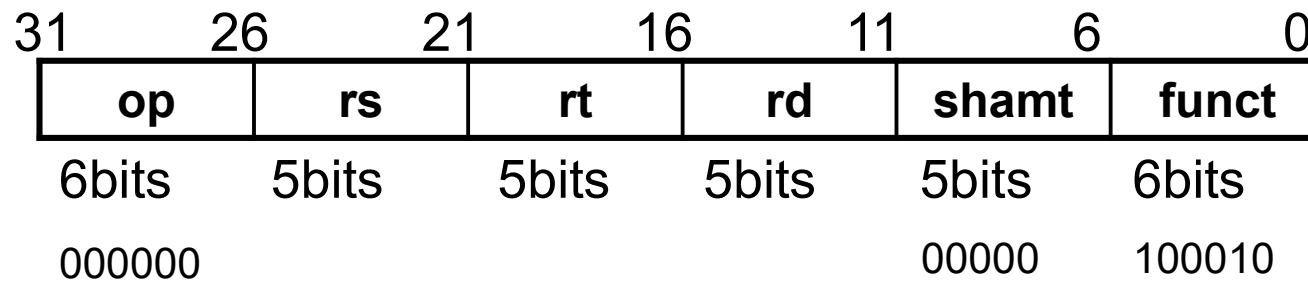
## RTL Description

Fetch instruction from memory

ADD instruction

Calculate next address

# sub Instruction



□ sub rd, rs, rt

$IR \leftarrow \text{mem}[PC];$

$R[rd] \leftarrow R[rs] + \sim R[rt] + 1;$

$PC \leftarrow PC + 4;$

## RTL Description

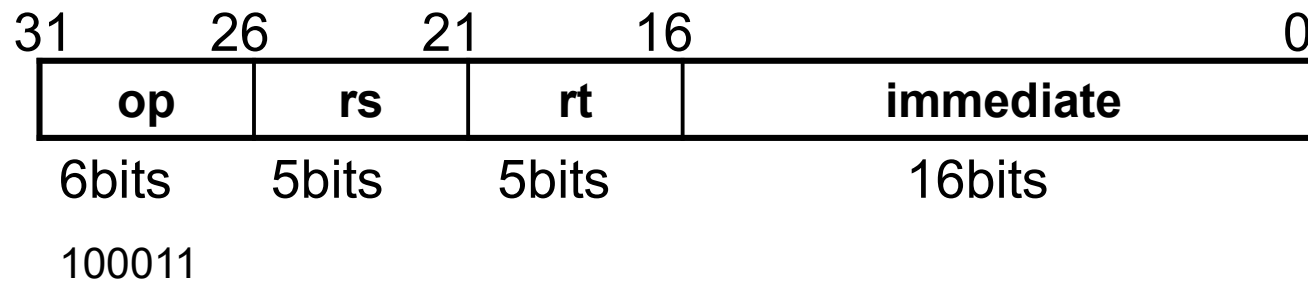
Fetch instruction from memory

SUB instruction

Calculate next address



# lw Instruction



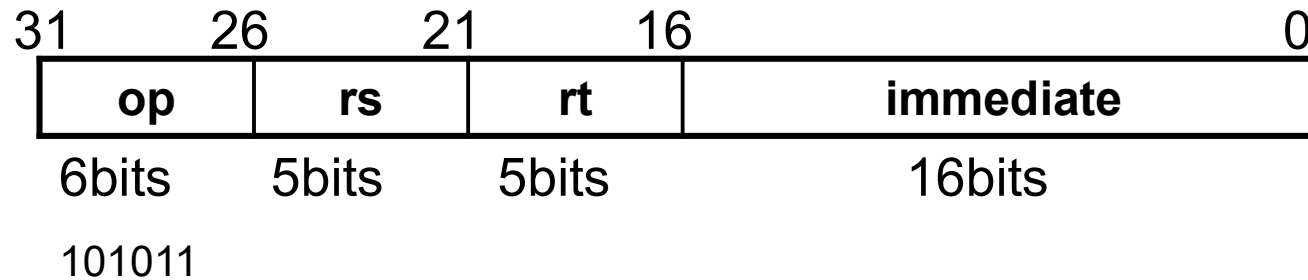
## □ lw rt, rs, imm16

$IR \leftarrow \text{mem}[PC];$   
 $\text{Addr} \leftarrow R[\text{rs}] + \text{SignExt}(\text{imm16});$   
 $R[\text{rt}] \leftarrow \text{Mem}[\text{Addr}];$   
 $PC \leftarrow PC + 4;$

## RTL Description

Fetch instruction from memory  
Compute memory address  
Load data into register  
Calculate next address

# sw Instruction

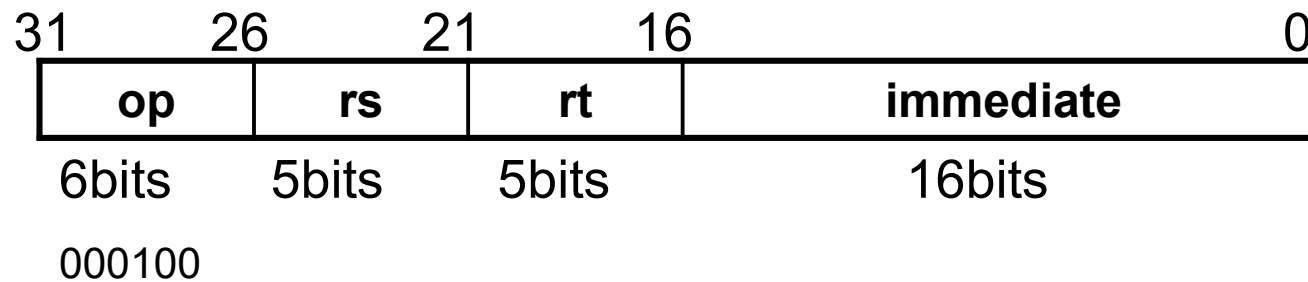


- **sw** rt, rs, imm16  
IR  $\leftarrow$  mem[PC];  
Addr  $\leftarrow$  R[rs] + SignExt(imm16);  
Mem[Addr]  $\leftarrow$  R[rt];  
PC  $\leftarrow$  PC + 4;

## RTL Description

Fetch instruction from memory  
Compute memory address  
Store data into memory  
Calculate next address

# beq Instruction



## □ beq rt, rs, imm16

IR  $\leftarrow$  mem[PC];

Cond  $\leftarrow$  R[rs] +  $\sim$ R[rt] + 1;

if (Cond == 0) then

PC  $\leftarrow$  PC + 4 + (SignExt(imm16)  $\ll$  2);  
(Branch if equal)

else

PC  $\leftarrow$  PC + 4;

(Fall through otherwise)

## RTL Description

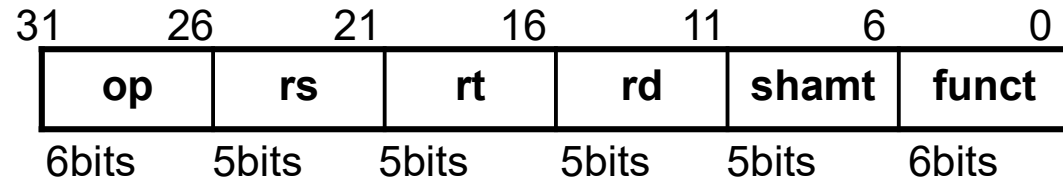
Fetch instruction from memory

Compute conditional Cond

# Instruction Format Summary

## □ Add, sub, and, or

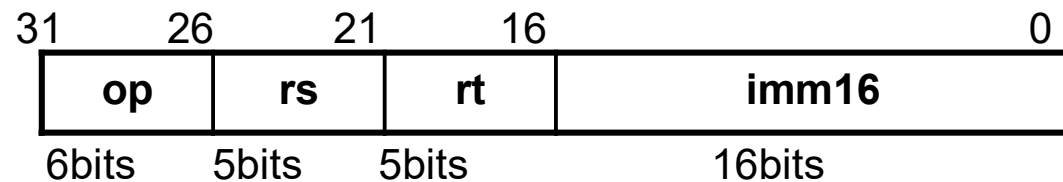
- add rd, rs, rt
- sub rd, rs, rt



R-format Instruction

## □ Load, store

- lw rt, rs, imm16
- sw rt, rs, imm16



I-format Instruction

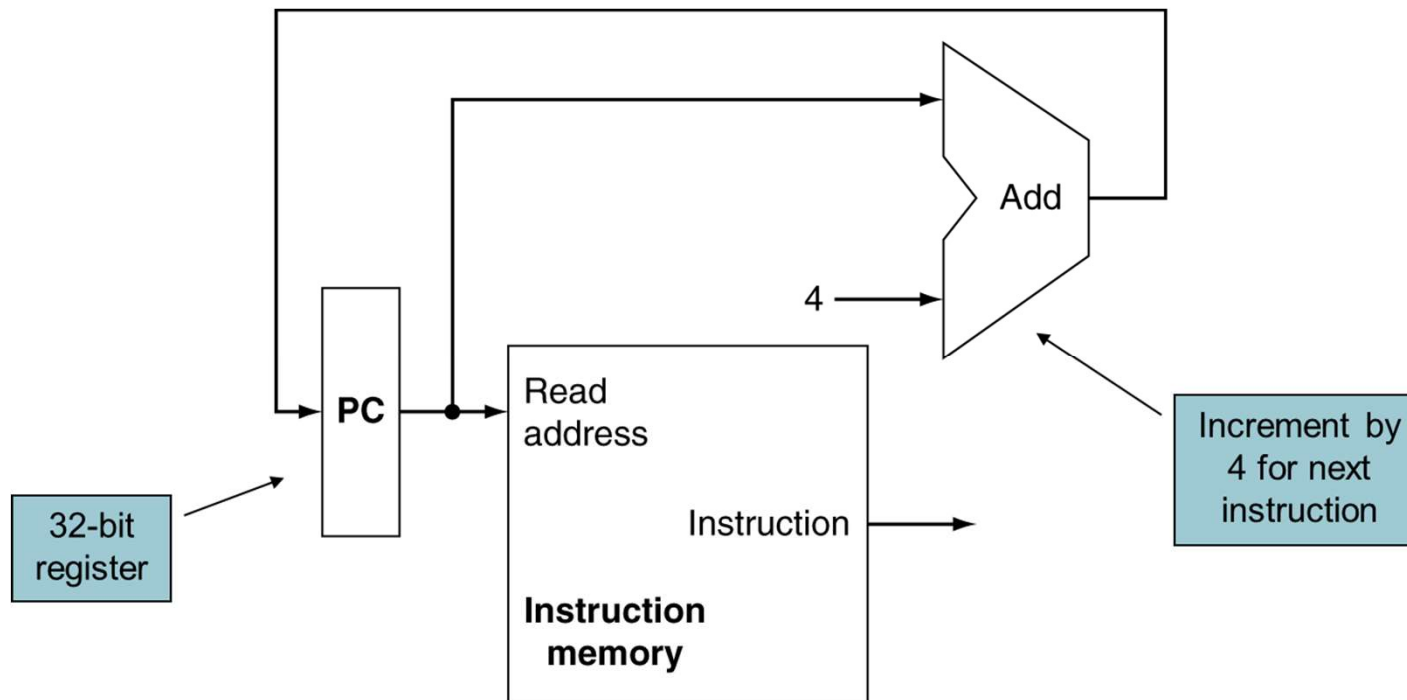
## □ Branch

- beq rt, rs, imm16

# PC-related Datapath

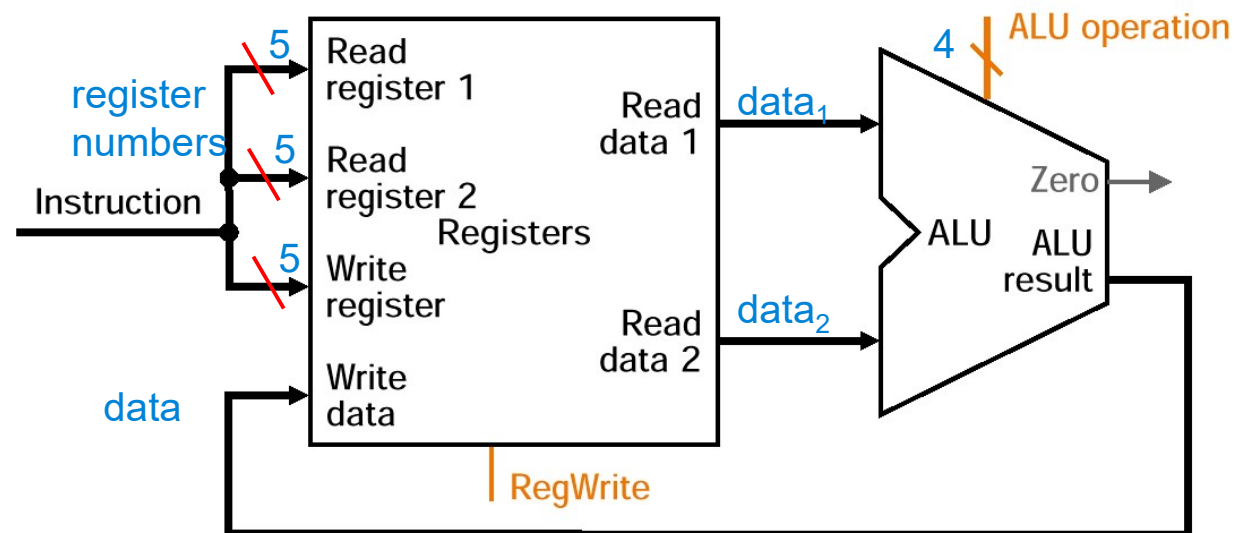
## □ Datapath elements for **instruction fetch**

- **Instruction memory** : Store the instructions of a program
- **Program Counter (PC)** : Store the address of the instruction
- **Adder** : Increment the PC to the address of the next instruction

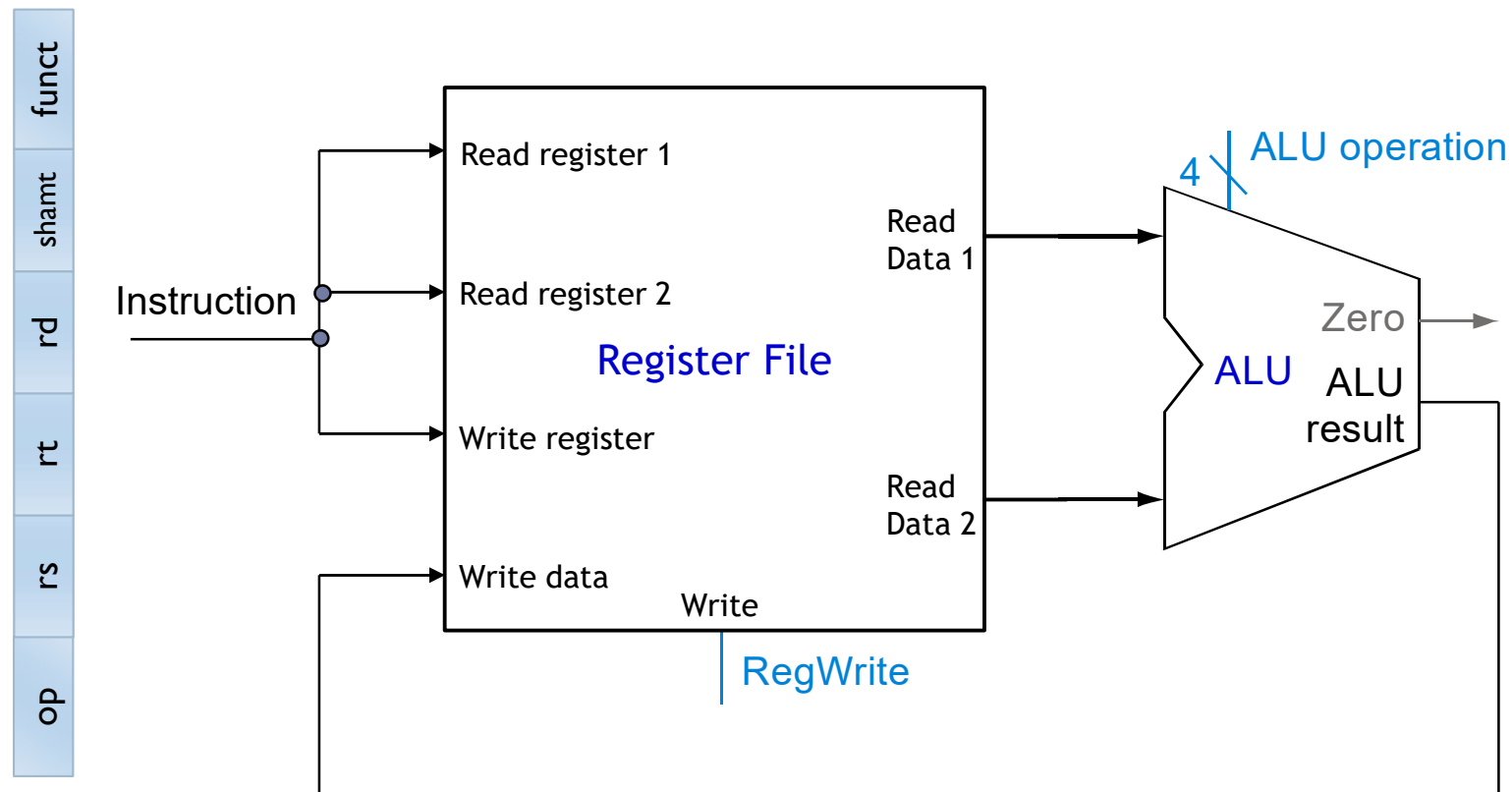


# Datapath for R-format Instructions (1)

- ❑ Step 3
  - Use ALU for the opcode execution
- ❑ Step 4
  - Write the data from the ALU back into a register
- ❑ Major components

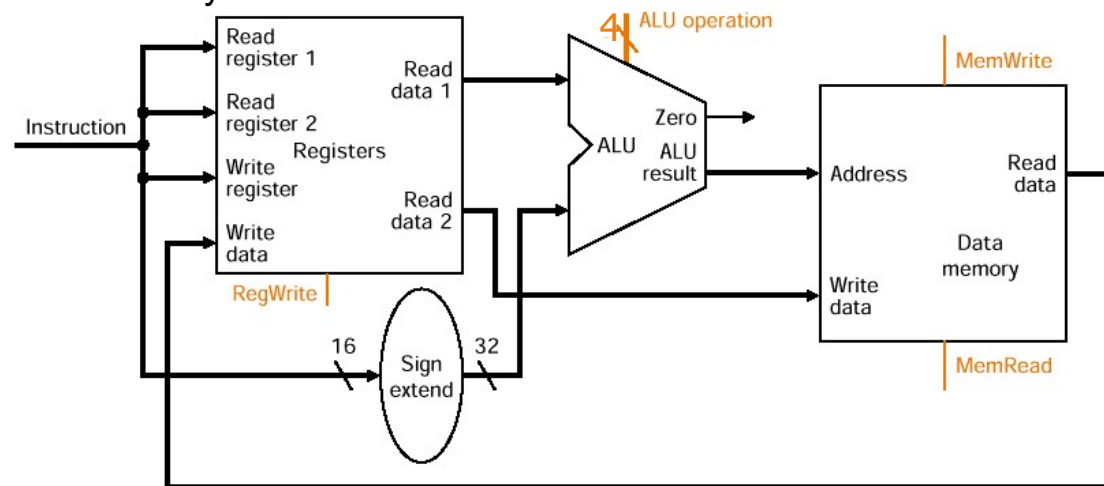


# Datapath for R-format Instructions (2)



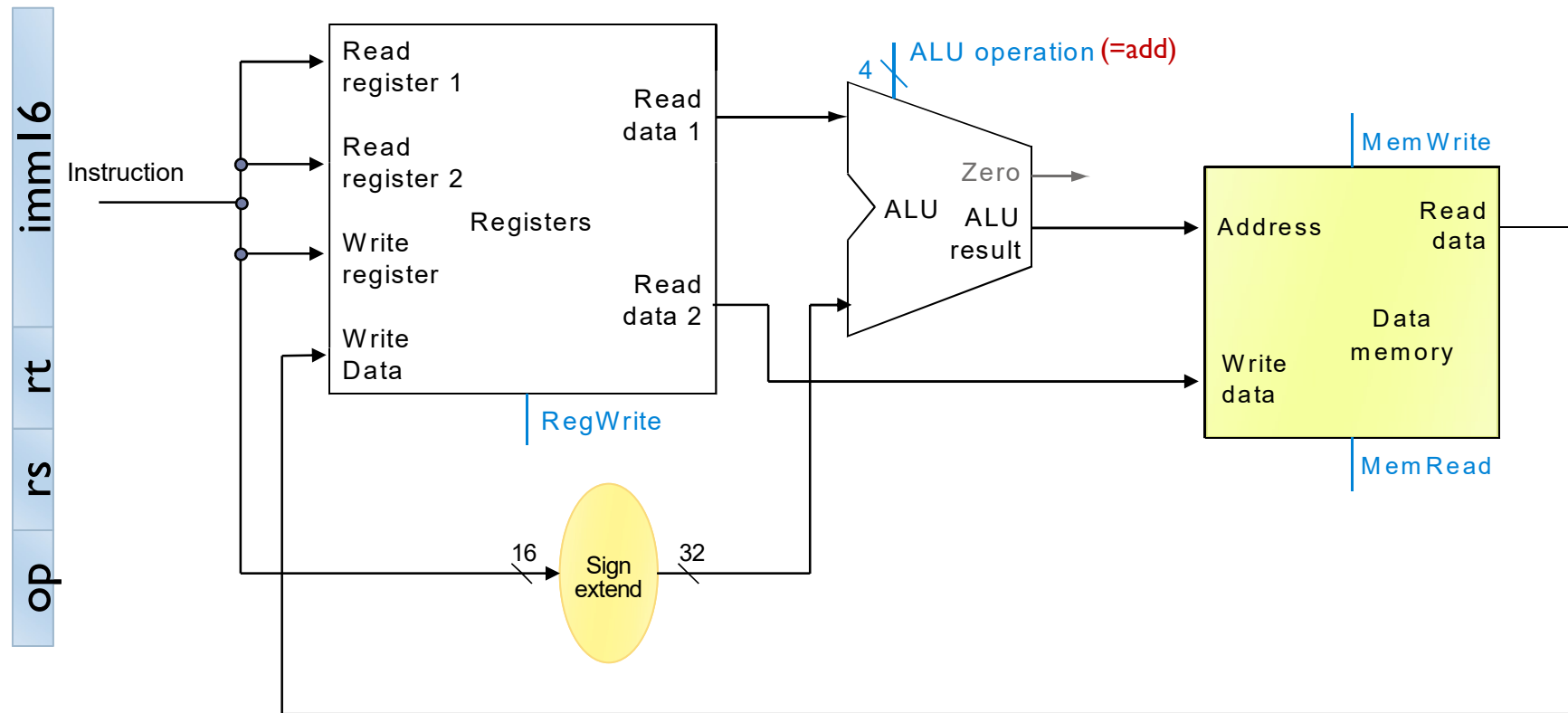
# Datapath for a Load or Store (1)

- Step 3
  - Compute a memory address by adding the base register to the 16-bit offset
- Step 4
  - `sw $t1,offset_value($t2)`
    - Write \$t1 into memory
  - `lw $t1,offset_value($t2)`
    - Read a value from memory
    - Write it into \$t1
- Major components
  - Sign extension unit and data memory





# Datapath for a Load or Store (2)



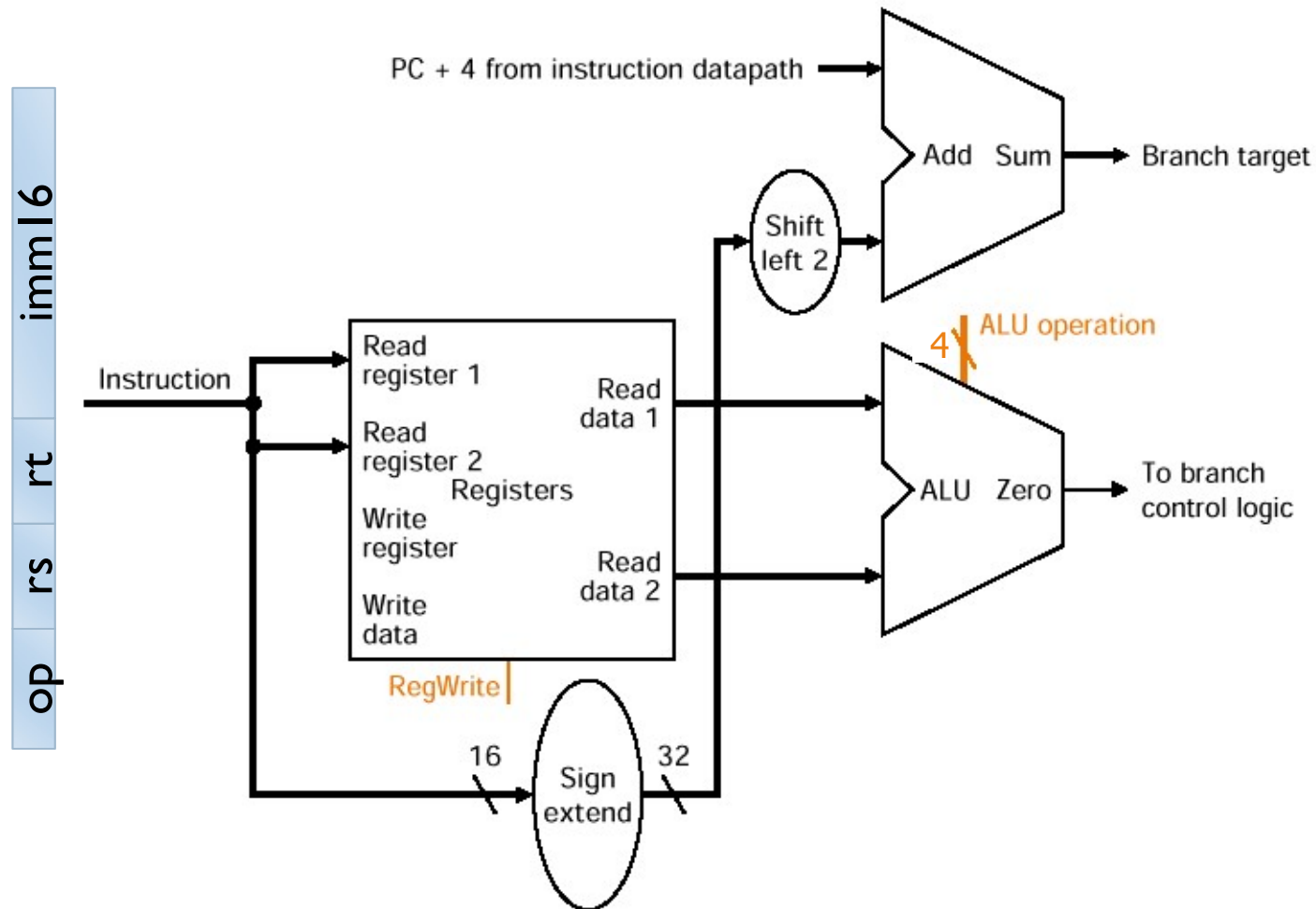
# Datapath for a Branch (1)



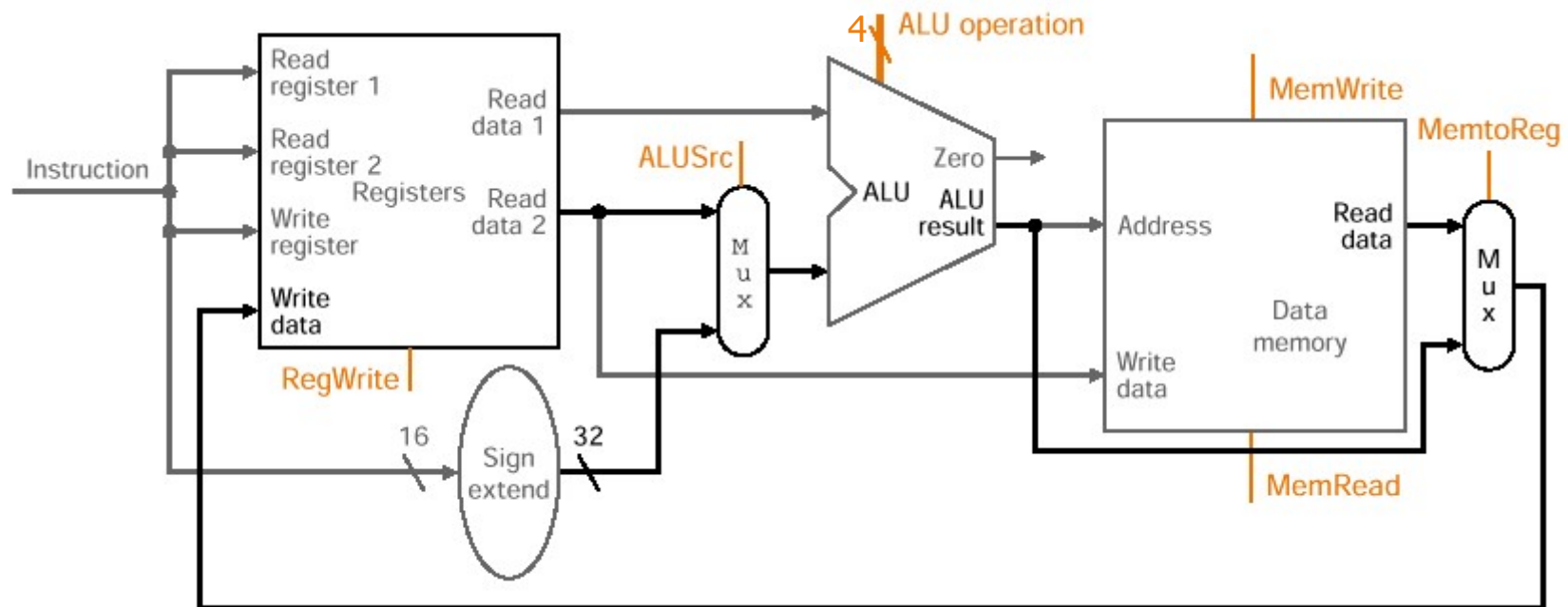
---

- ❑ Step 3
  - Use ALU for comparison
  - Compute branch target address by adding the sign-extended offset to the PC
- ❑ Step 4
  - Change PC based on the comparison
- ❑ Major components
  - Shift-left-2 unit
  - Separate adder

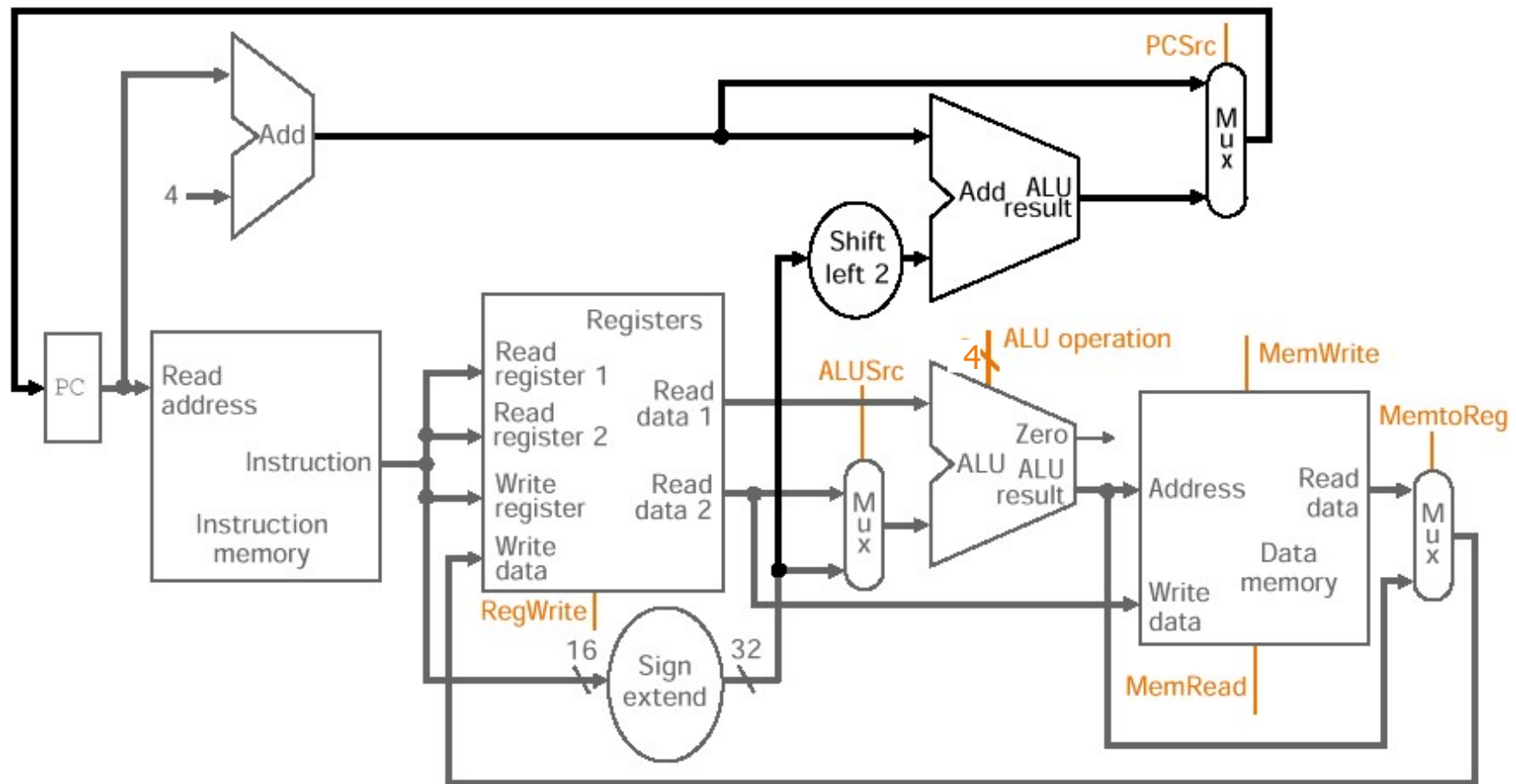
# Datapath for a Branch (2)



# R-format + (Load or Store)



# R-format + (Load or Store) + Branch



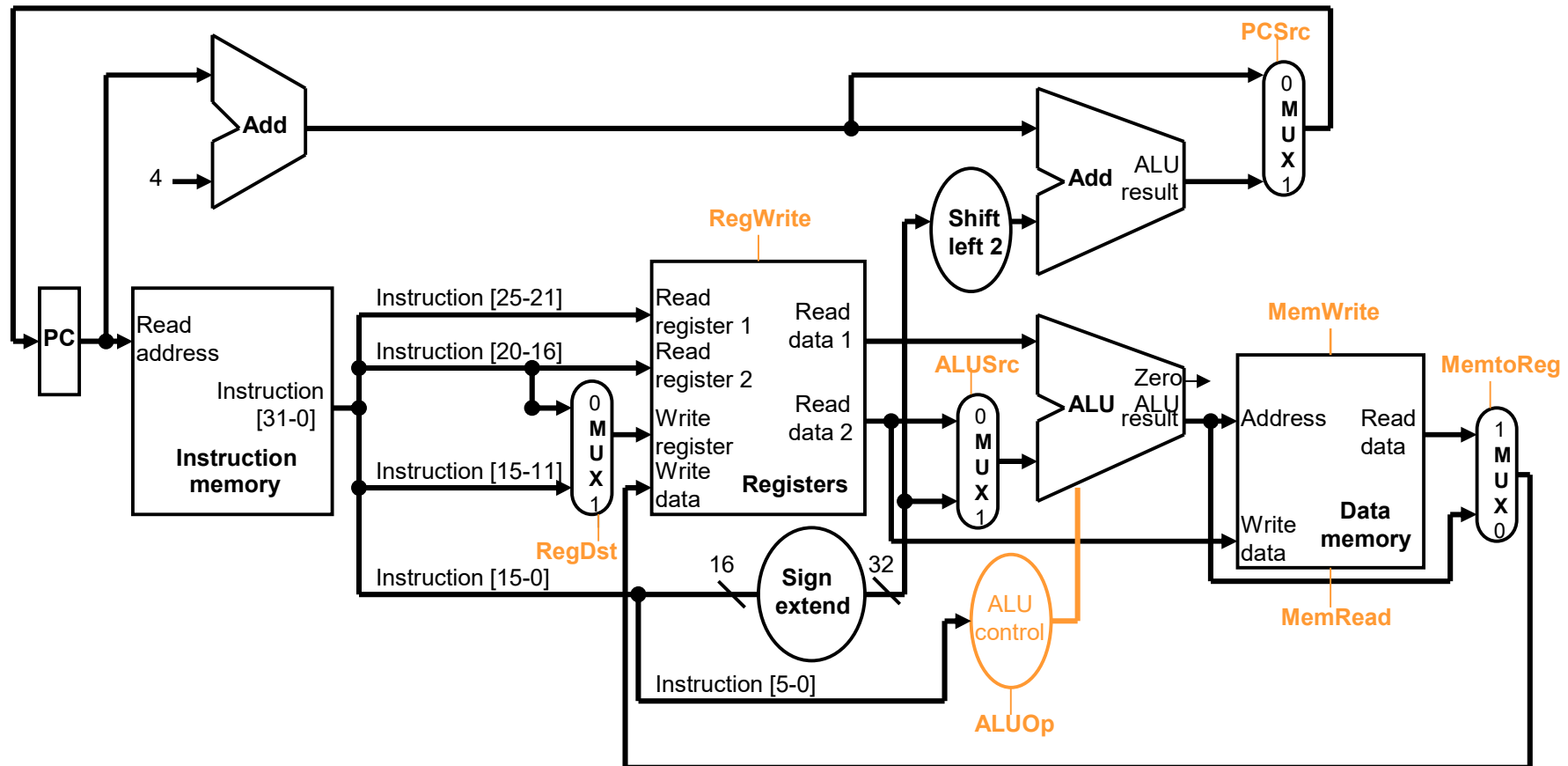
# Creating a Single-cycle Datapath



---

- ❑ The simplest datapath
  - Execute all instructions in one clock cycle
    - No datapath resource can be used more than once per instruction.
    - Any element needed more than once must be duplicated.
    - Separate instruction and data memories
  - Sharing a datapath between two different instruction types
    - Use multiplexor

# Single-Cycle Datapath



□ This datapath supports the following instructions:

- add, sub, and, or, slt, lw, sw, beq

# Single-Cycle Control



---

RegDst	Select destination register
RegWrite	Specify if the destination register is written
ALUSrc	Select whether source is register or immediate
ALUOp	Specify operation for ALU
MemWrite	Specify whether memory is to be written
MemRead	Specify whether memory is to be read
MemtoReg	Select whether memory or ALU output is used
PCSrc	Select whether next PC or computed address is used



# Review on Instruction Format



Field	0	rs	rt	rd	shamt	funct
Bit positions	31:26	25:21	20:16	15:11	10:6	5:0

a. R-type instruction

Field	35 or 43	rs	rt	address
Bit positions	31:26	25:21	20:16	15:0

b. Load or store instruction

Field	4	rs	rt	address
Bit positions	31:26	25:21	20:16	15:0

c. Branch instruction

# Execution of an R-format Instruction

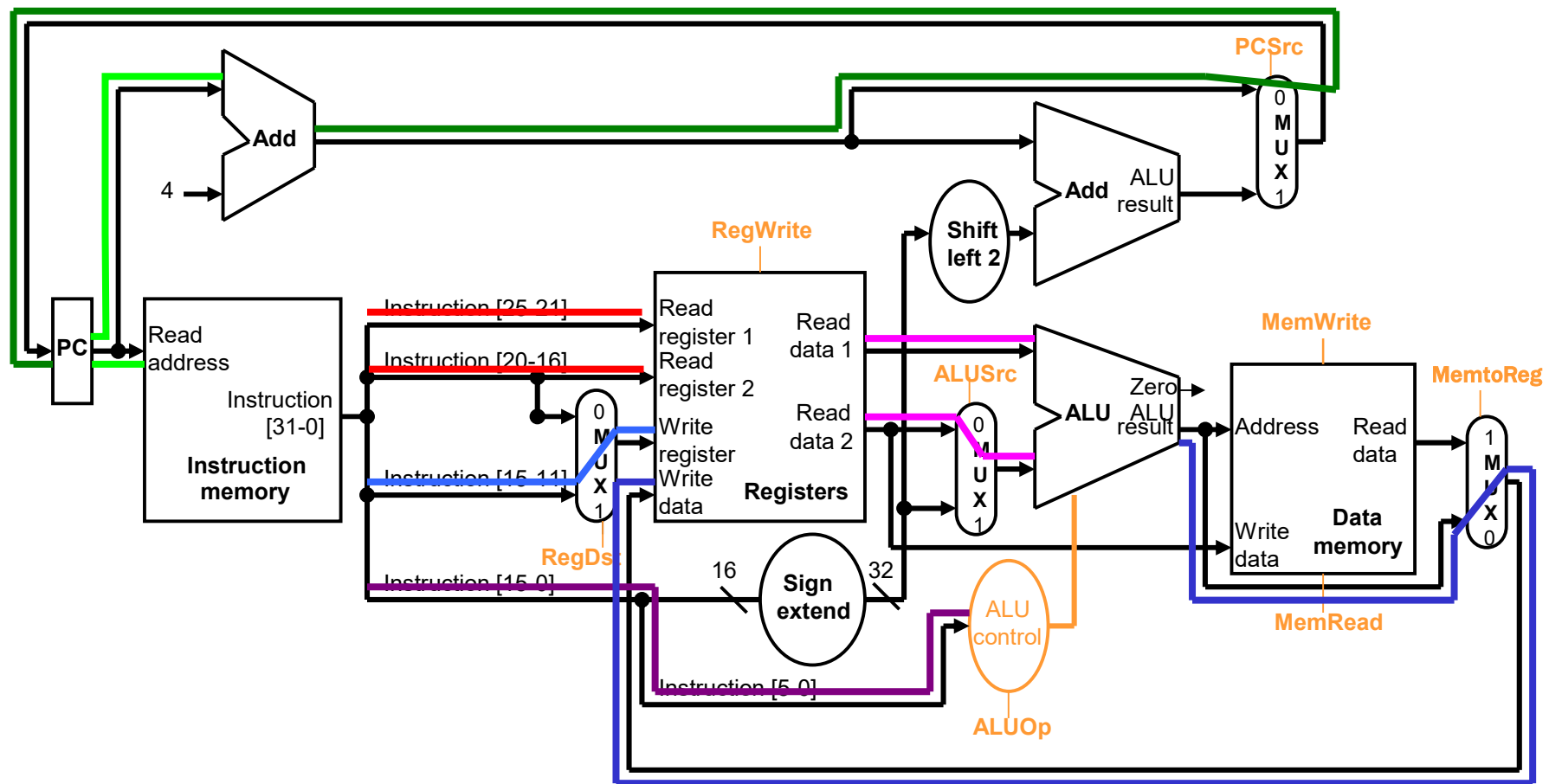


□ e.g. `add $t1, $t2, $t3`

1. Fetch instruction and increment PC.
2. Read two registers (`$t2` and `$t3`) and generate control signals in the main control unit.
3. ALU operates on the data, using the function code.
4. ALU result is written into `$t1`.

# R-format Instruction Dataflow

- For add, sub, and, or, slt instructions



# R-format Instruction Control

## □ Control signal summary

RegDst	1 to select Rd
RegWrite	1 to enable writing Rd
ALUSrc	0 to select Rt value from register file
ALUOp	OP (→ <i>Dependent on operation</i> ) (see below)
MemWrite	0 to disable writing memory
MemRead	0 to disable reading memory
MemtoReg	0 to select ALU output to register
PCSrc	0 to select next PC

## □ ALUOp

instruction	add	sub	slt	and	or
ALUOp	OP	OP	OP	OP	OP
<i>ALUControl</i>	<i>add</i>	<i>sub</i>	<i>slt</i>	<i>and</i>	<i>or</i>

# Execution of a load Instruction



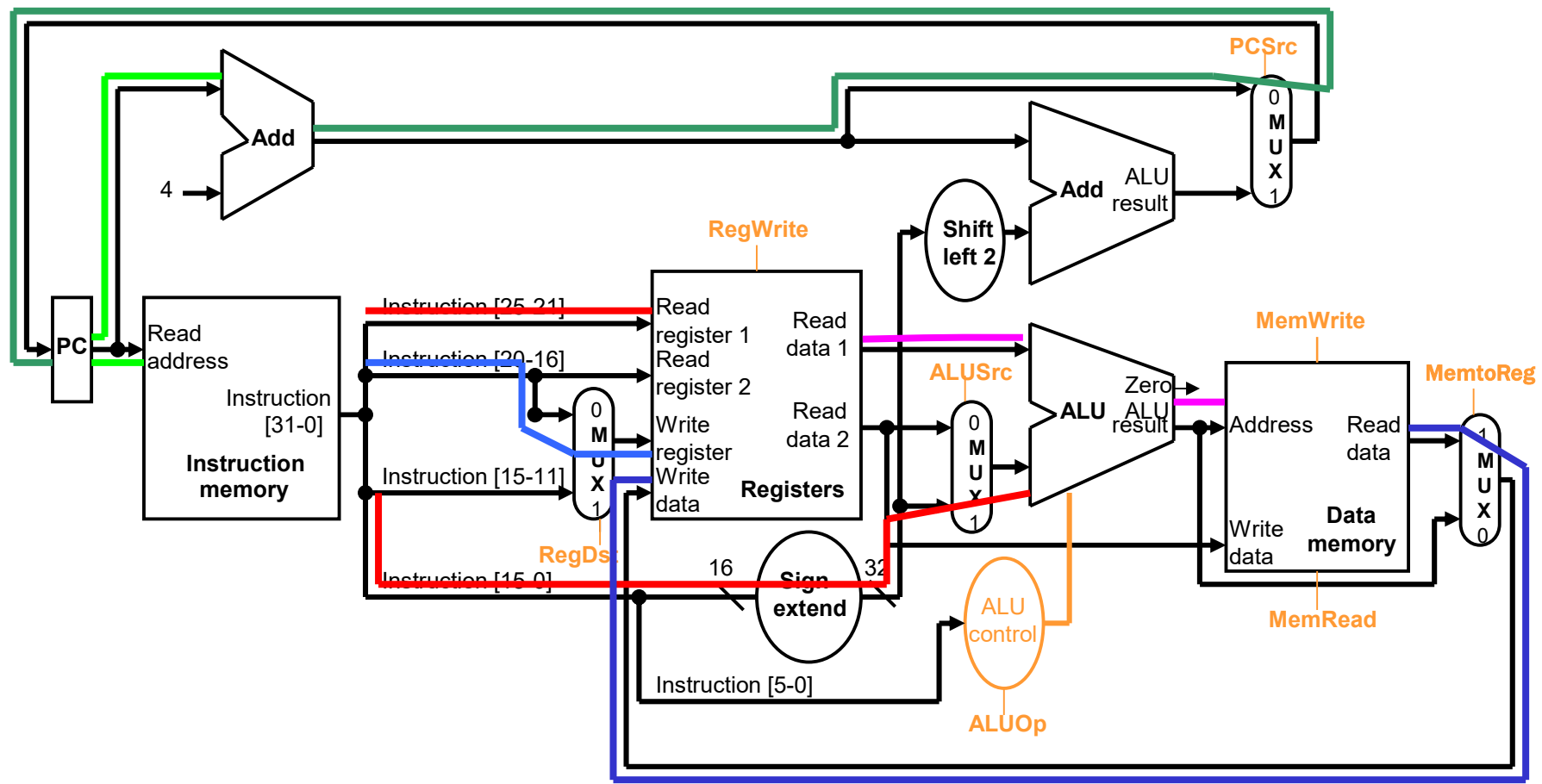
---

□ e.g. `lw $t1, offset($t2)`


1. Fetch instruction and increment PC.
2. Read a register (`$t2`) and generate control signals in the main control unit.
3. ALU computes the effective address by adding `$t2` and sign-extended offset.
4. Use the ALU output as the address for the data memory.
5. Write the data from the memory into `$t1`.

# Load Instruction (I-format) Dataflow

- For lw instruction



# Load Instruction (I-format) Control

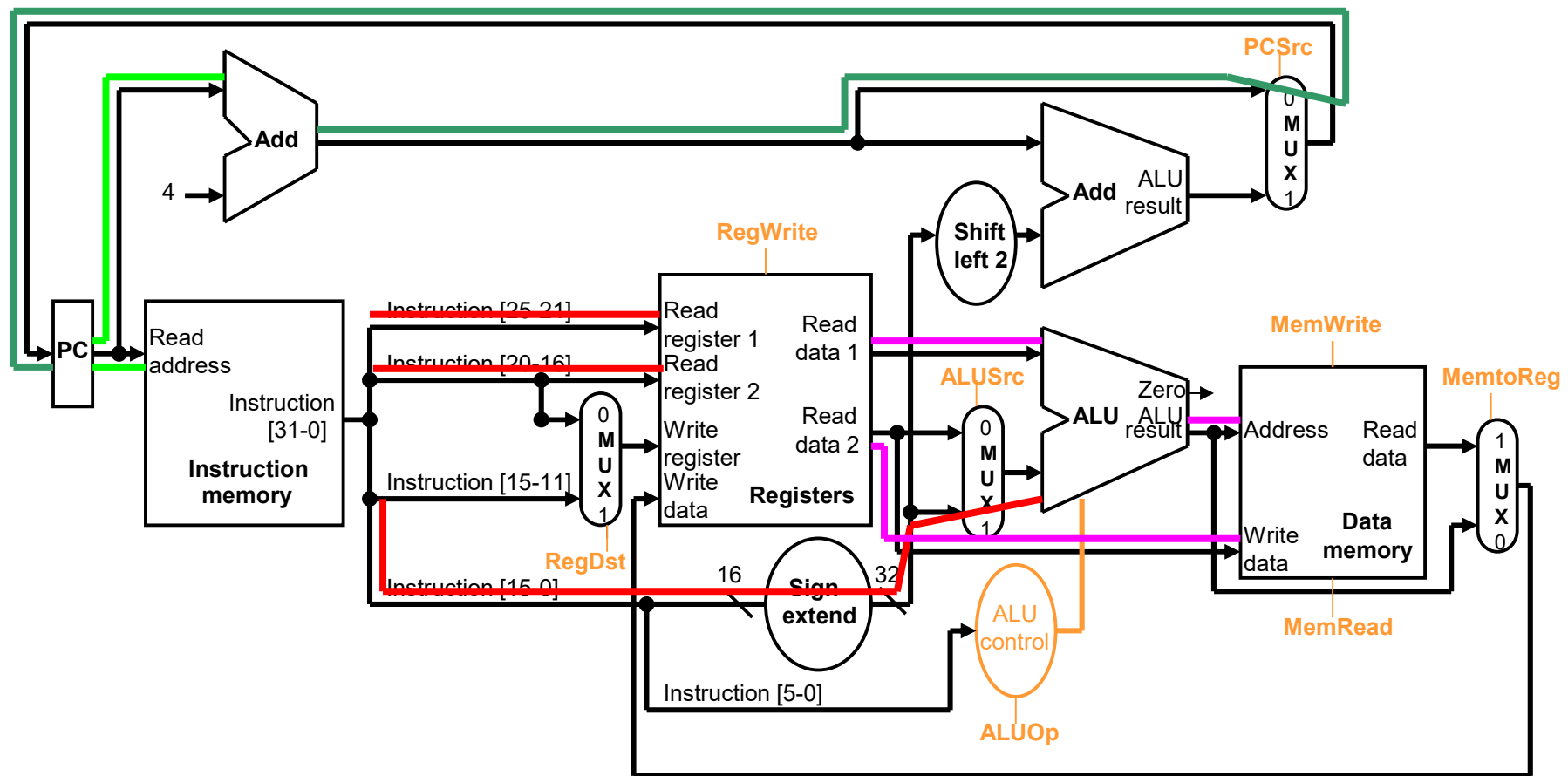


## □ Control signal summary

RegDst	0 to select Rt
RegWrite	1 to enable writing Rt
ALUSrc	1 to select immediate field value from instruction
ALUOp	add
MemWrite	0 to disable writing memory
MemRead	1 to enable reading memory
MemtoReg	1 to select memory output to register
PCSrc	0 to select next PC


# Store Instruction (I-format) Dataflow

- For sw instruction





# Store Instruction (I-format) Control



## □ Control signal summary

RegDst	x (don't care)
RegWrite	0 to disable writing a register
ALUSrc	1 to select Rt value from register file
ALUOp	add
MemWrite	1 to enable writing memory
MemRead	0 to disable reading memory
MemtoReg	x (don't care)
PCSrc	0 to select next PC

# Execution of a branch Instruction

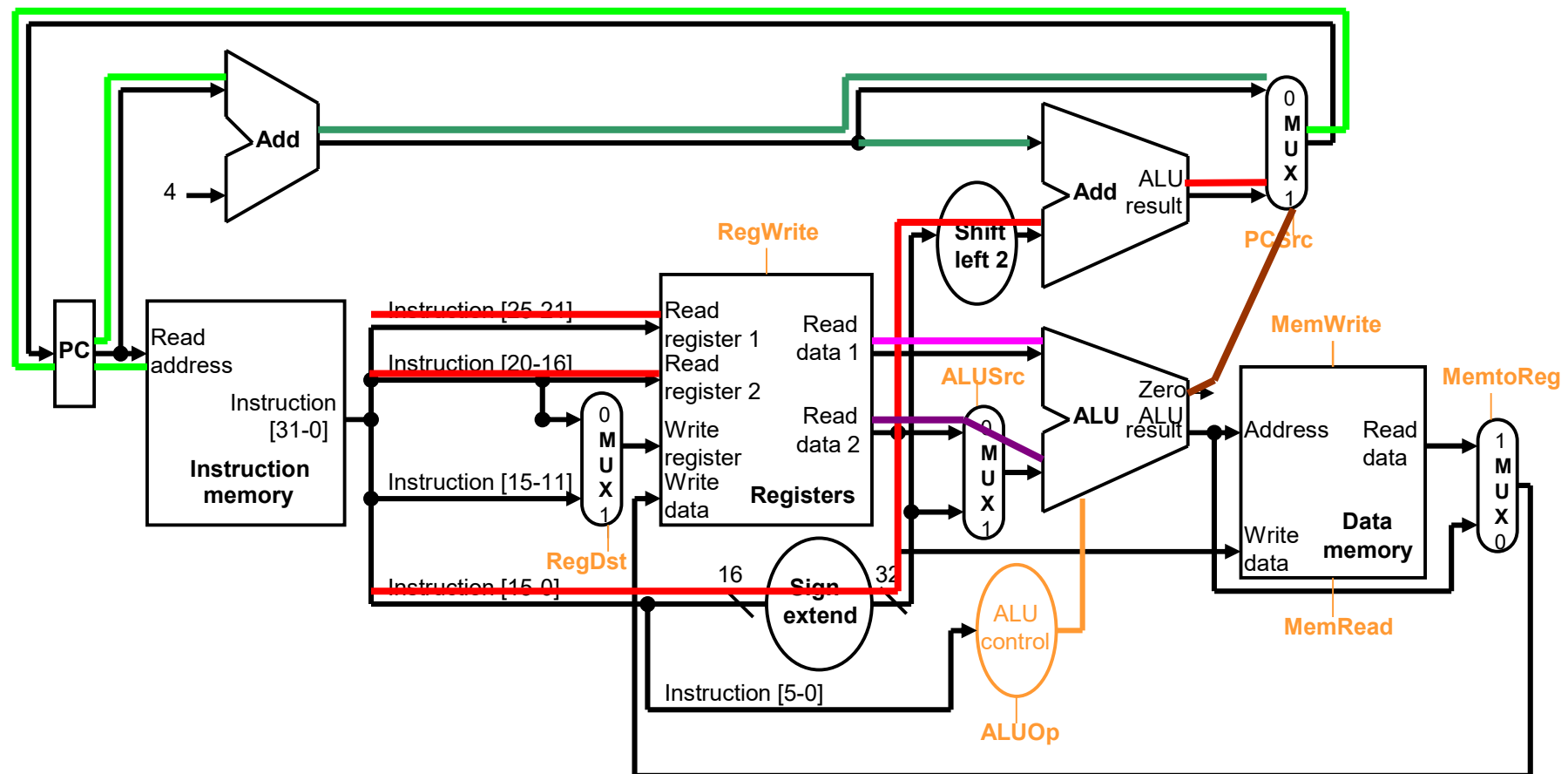


□ e.g. `beq $t1, $t2, offset`

1. Fetch instruction and increment PC.
2. Read two registers (`$t2` and `$t3`) and generate control signals in the main control unit.
3. Subtract the two data from registers. And compute branch target address by adding  $(PC+4)$  and sign-extended and shifted-left `offset`.
4. Decide which adder result to store into the PC using the Zero result from ALU.

# Branch Instruction (I-format) Dataflow

- For beq instruction



# Branch Instruction (I-format) Control



## □ Control signal summary

RegDst	x (don't care)
RegWrite	0 to disable writing a register
ALUSrc	0 to select Rt value from register file
ALUOp	sub
MemWrite	0 to disable writing memory
MemRead	0 to disable reading memory
MemtoReg	x (don't care)
PCSrc	zero

# Single-Cycle Control Signals Summary



<u>Signal</u>	<u>R-format</u>	<u>I-format (lw)</u>	<u>I-format (sw)</u>	<u>I-format (beq)</u>
RegDst	1	0	x	x
RegWrite	1	1	0	0
ALUSrc	0	1	1	0
ALUOp	OP	add	add	sub
MemWrite	0	0	1	0
MemRead	0	1	0	0
MemtoReg	0	1	x	x
PCSrc	0	0	0	zero

\* **OP** : *dependent on operation*

# ALU control Design

---



ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set on less than
1100	NOR

# Opcode and ALU control Input

## ❑ Controlling the ALU uses of multiple decoding levels

main control unit generates the **ALUOp bits**

ALU control unit generates **ALUcontrol bits**

\* 명령어에 따른 **ALUOp** 정의

R-format : **10** (OP), lw / sw: **00** (add), beq: **01**(sub)

Instruction opcode	<b>ALUOp</b>	Instruction operation	Function field	Desired ALU <b>action</b>	<b>ALU control</b>
lw	<b>00</b>	load word	xxxxxx	add	0010
sw	<b>00</b>	store word	xxxxxx	add	0010
beq	<b>01</b>	branch on equal	xxxxxx	subtract	0110
R-format	<b>10</b>	add	10 <b>0000</b>	add	0010
R-format	<b>10</b>	subtract	10 <b>0010</b>	subtract	0110
R-format	<b>10</b>	AND	10 <b>0100</b>	and	0000
R-format	<b>10</b>	OR	10 <b>0101</b>	or	0001
R-format	<b>10</b>	set on less than	10 <b>1010</b>	set on less than	0111

# ALU Control Truth Table

F5	F4	F3	F2	F1	F0	ALU Op <sub>1</sub>	ALU Op <sub>0</sub>	ALU control <sub>3</sub>	ALU control <sub>2</sub>	ALU control <sub>1</sub>	ALU control <sub>0</sub>
X	X	X	X	X	X	0	0	0	1	1	0
X	X	X	X	X	X	0	1	1	1	1	0
X	X	0	0	0	0	1	0	0	1	1	0
X	X	0	0	1	0	1	0	1	1	1	0
X	X	0	1	0	0	1	0	0	0	0	0
X	X	0	1	0	1	1	0	0	0	0	1
X	X	0	1	1	0	1	0	0	0	1	0
X	X	0	1	1	1	1	0	0	0	1	1
X	X	1	0	1	0	1	0	1	1	1	1

- Four, 6-input truth tables



# ALU Control Truth Table

Our ALU m control input

F5	F4	F3	F2	F1	F0	ALU Op <sub>1</sub>	ALU Op <sub>0</sub>	ALU control <sub>3</sub>	ALU control <sub>2</sub>	ALU control <sub>1</sub>	ALU control <sub>0</sub>
X	X	X	X	X	X	0	0	0	1	1	0
X	X	X	X	X	X	0	1	1	1	1	0
X	X	0	0	0	0	1	0	0	1	1	0
X	X	0	0	1	0	1	0	1	1	1	0
X	X	0	1	0	0	1	0	0	0	0	0
X	X	0	1	0	1	1	0	0	0	0	1
X	X	0	1	1	0	1	0	0	0	1	0
X	X	0	1	1	1	1	0	0	0	1	1
X	X	1	0	1	0	1	0	1	1	1	1

□ Four, 6-input truth tables

Add/subt

Mux control

# Truth Table for the ALU control



ALUOp		Function field						Operation (ALU control)
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	x	x	x	x	x	x	0010
x	1	x	x	x	x	x	x	0110
1	x	x	x	0	0	0	0	0010
1	x	x	x	0	0	1	0	0110
1	x	x	x	0	1	0	0	0000
1	x	x	x	0	1	0	1	0001
1	x	x	x	1	0	1	0	0111

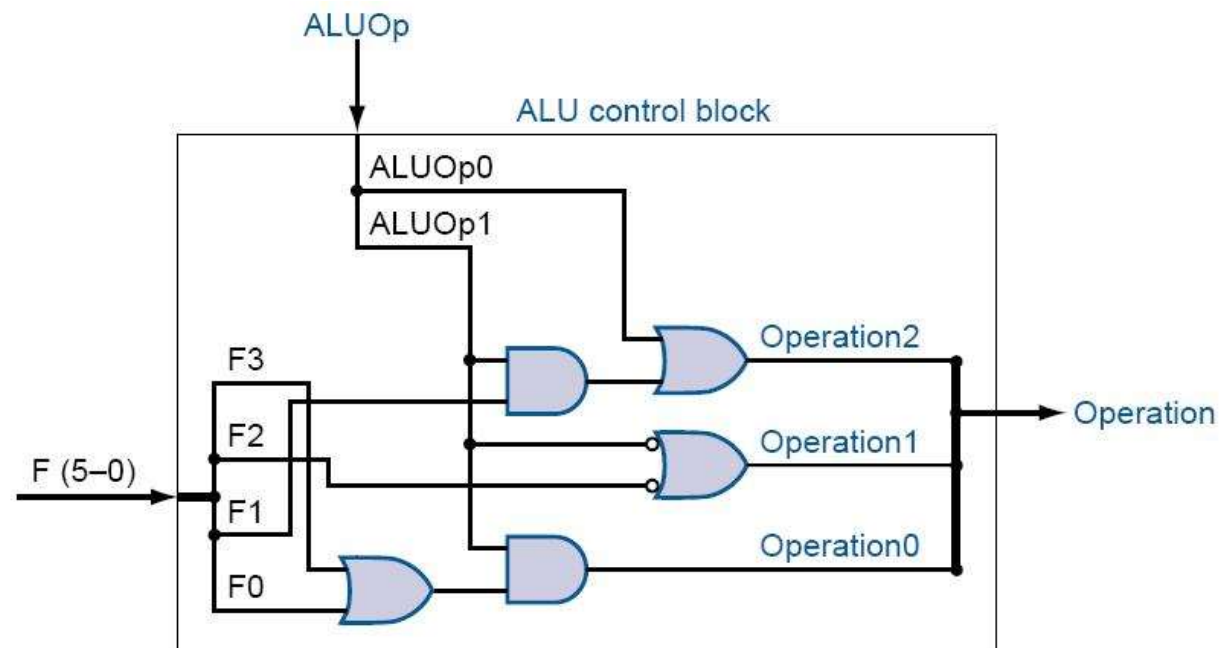
# Implementation of ALU control

Operation3 = 0

Operation2 =  $\text{ALUOp0} + \text{ALUOp1} \cdot F1$

Operation1 =  $\text{ALUOp1}' + F2'$

Operation0 =  $\text{ALUOp1} \cdot (F0 + F3)$



# Control Unit Outputs

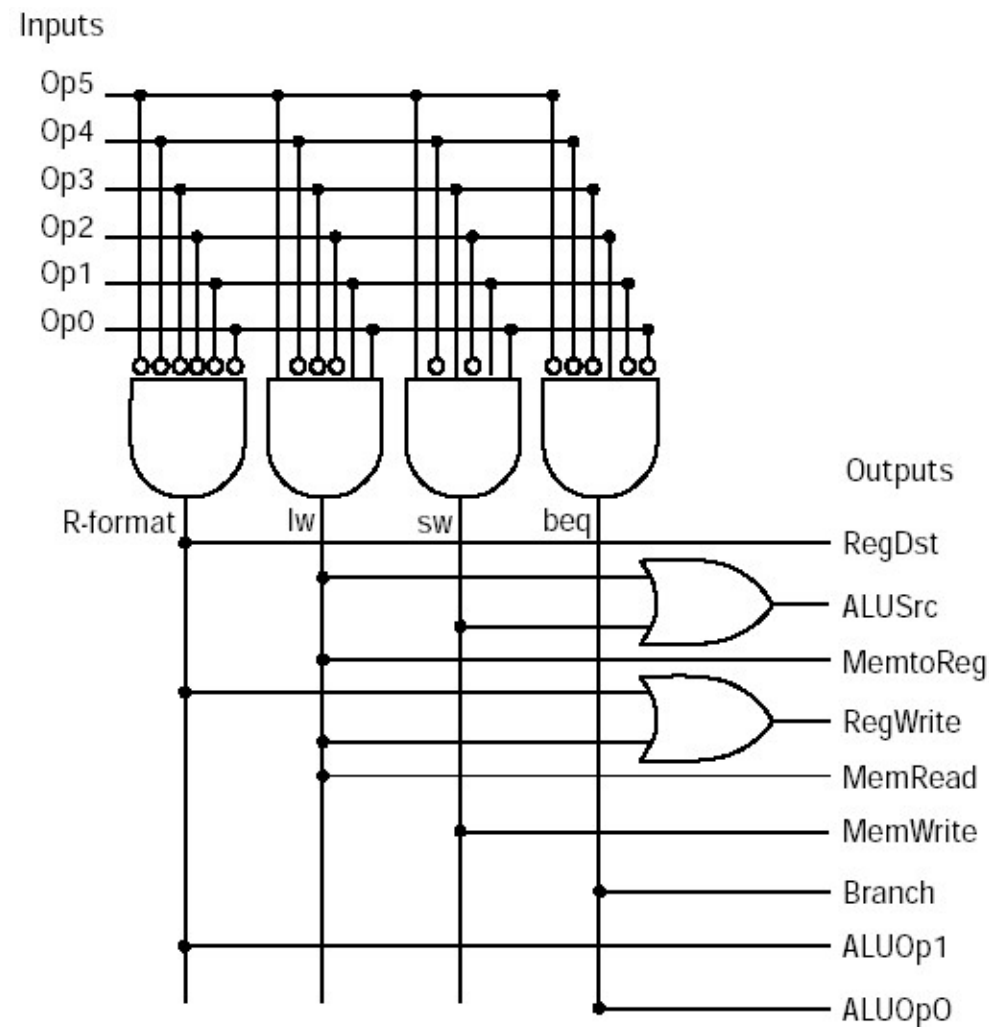


Instruction	Reg Dst	ALU Src	MEM toReg	Reg Write	Mem Read	Mem Write	Branch	ALU-Op1	ALU-Op0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	x	1	x	0	0	1	0	0	0
beq	x	0	x	0	0	0	1	0	1

# Control Outputs (Final)

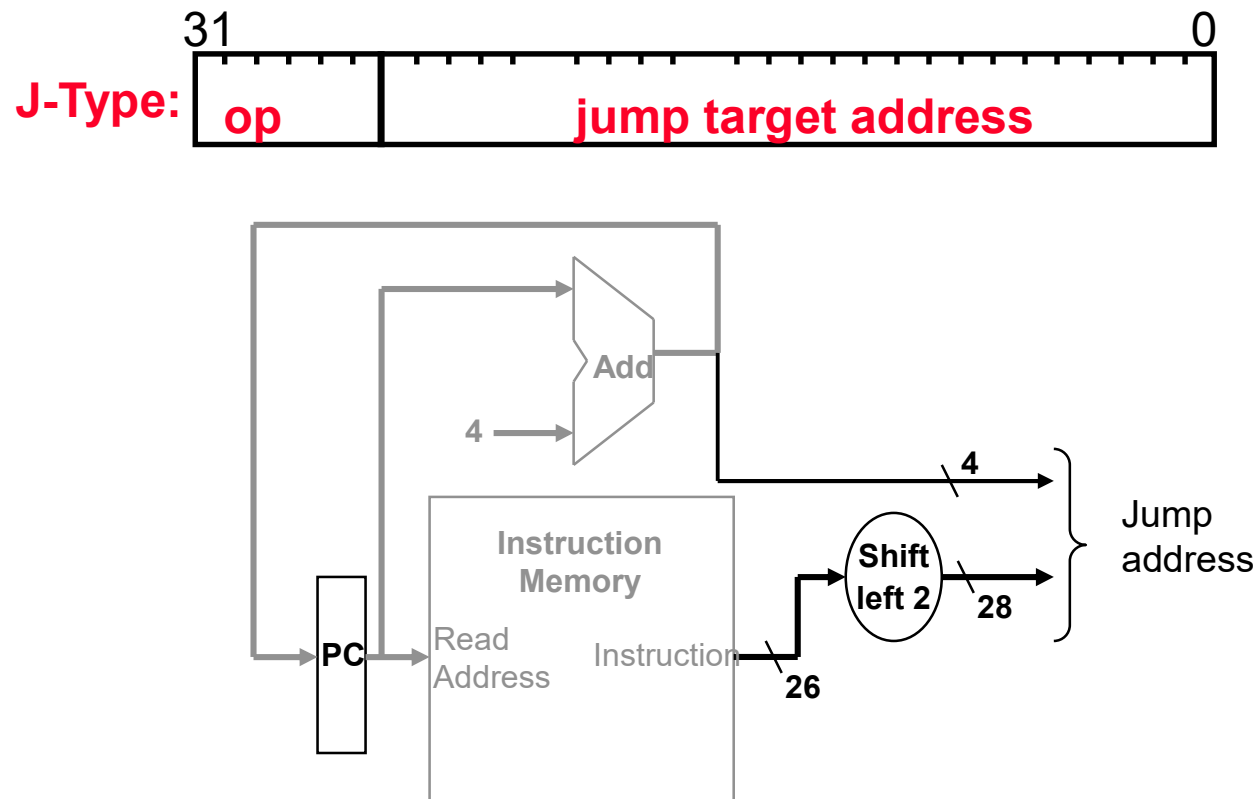
Control	Signal name	R-format	lw	sw	beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

# More Details on Control Signal Generation

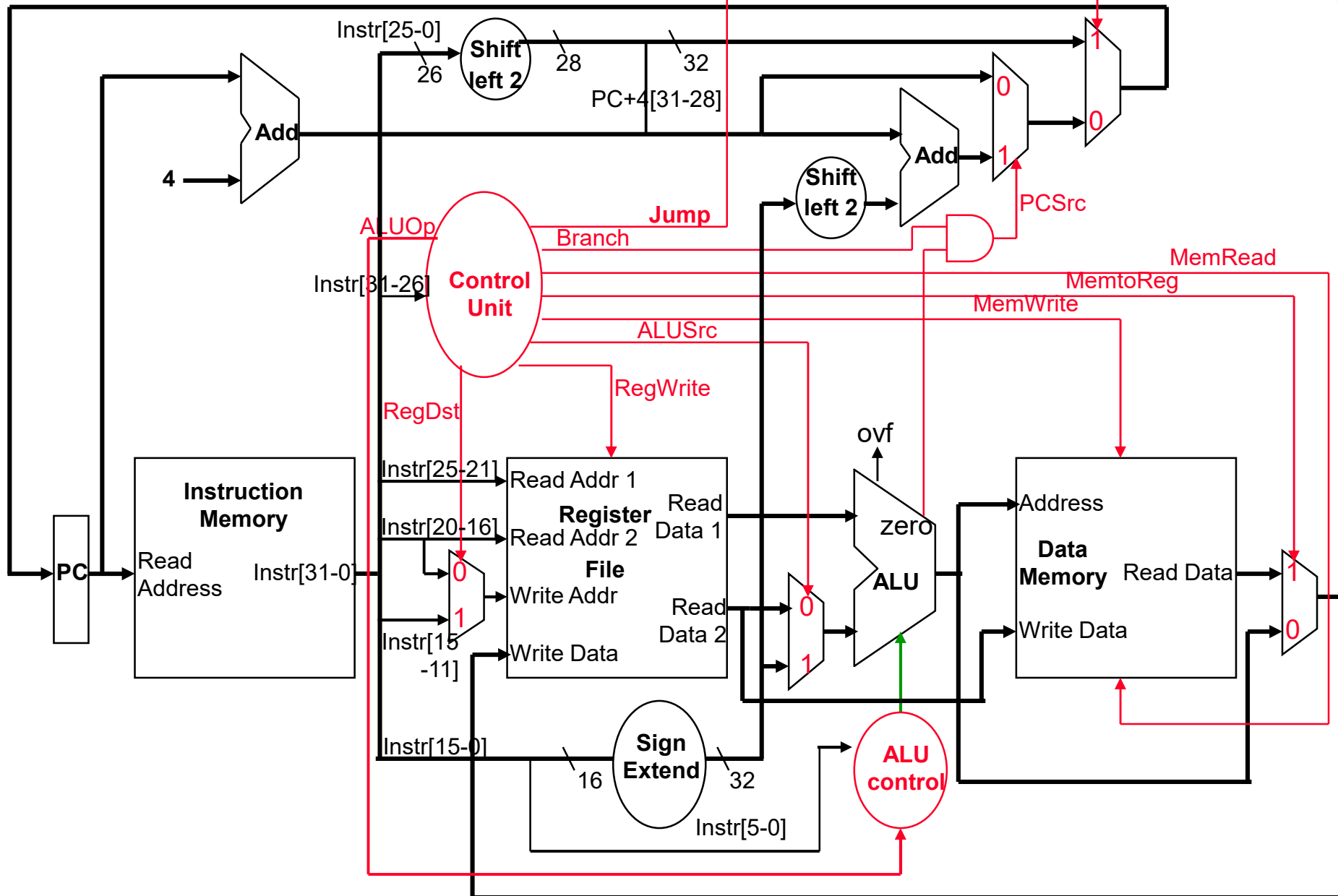


# Review: Handling Jump Operations

- Jump operation have to
  - replace the lower 28 bits of the PC with the lower 26 bits of the fetched instruction shifted left by 2 bits

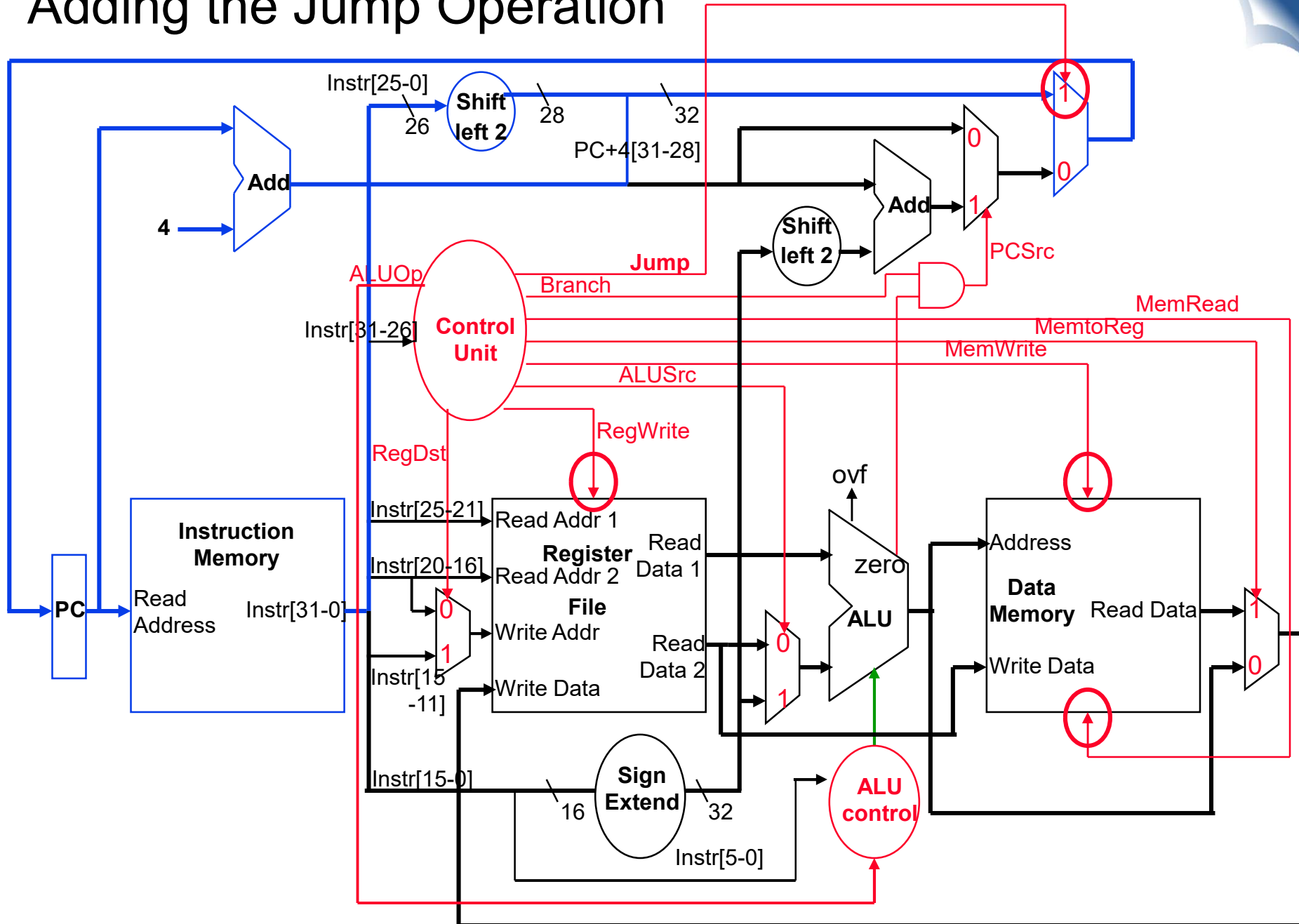


# Adding the Jump Operation





# Adding the Jump Operation



# Main Control Unit



Instr	RegDst	ALUSrc	MemReg	RegWr	MemRd	MemWr	Branch	ALUOp	Jump
<b>R-type</b> 000000									
<b>lw</b> 100011									
<b>sw</b> 101011									
<b>beq</b> 000100									
<b>j</b> 000010									

- ❑ Setting of the MemRd signal (for R-type, sw, beq) depends on the memory design

# Main Control Unit



Instr	RegDst	ALUSrc	MemReg	RegWr	MemRd	MemWr	Branch	ALUOp	Jump
<b>R-type</b> 000000	1	0	0	1	0	0	0	10	0
<b>lw</b> 100011	0	1	1	1	1	0	0	00	0
<b>sw</b> 101011	X	1	X	0	0	1	0	00	0
<b>beq</b> 000100	X	0	X	0	0	0	1	01	0
<b>j</b> 000010	X	X	X	0	0	0	X	XX	1

- ❑ Setting of the MemRd signal (for R-type, sw, beq) depends on the memory design

# Instruction Critical Paths

- ❑ Calculate cycle time assuming negligible delays (for muxes, control unit, sign extend, PC access, shift left 2, wires, setup and hold times) except:

Instruction and Data Memory (4 ns)

ALU and adders (2 ns)

Register File access (reads or writes) (1 ns)

Instr.	I Mem	Reg Rd	ALU Op	D Mem	Reg Wr	Total
R-type	4	1	2		1	8
load	4	1	2	4	1	12
store	4	1	2	4		11
beq	4	1	2			7
jump	4					4

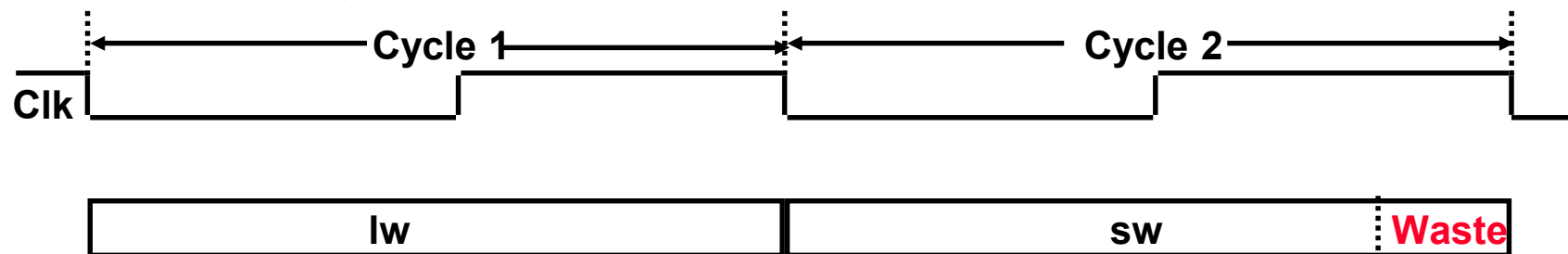
# Why Single-cycle Implementation Is Not Used



- ❑  $CPI = 1$ , but
  - Clock cycle is determined by the longest possible path
- ❑ **Load** instruction
  - instruction memory
  - register file
  - ALU
  - data memory
  - register file

# Single Cycle Disadvantages & Advantages

- ❑ Uses the clock cycle inefficiently – the clock cycle must be timed to accommodate the **slowest** instr  
especially problematic for more complex instructions like floating point multiply



- ❑ May be wasteful of area since some functional units (e.g., adders) must be duplicated since they can not be shared during a clock cycle

but

- ❑ It is simple and easy to understand

# Performance Evaluation



- ❑ Operation times for the major functional units
  - Memory units : 200 ps
  - ALU and adders : 100 ps
  - Register file : 50 ps
  - Multiplexors, control unit, PC, sign-extension unit, wires: no delay
- ❑ Instruction mix
  - 25% lw, 10% sw, 45% R-format, 15% beq, 5% j
- ❑ Which would be faster and by how much ?
  - (1) One clock cycle of a fixed length
  - (2) One clock cycle for every instruction but variable-length clock

## [Answer-1]



---

- ❑ CPU execution time
  - = instruction count x CPI x clock cycle time
  - = instruction count x clock cycle time ( $\because$  CPI = 1 )
- ❑ Compare the clock cycle time for both implementation.
- ❑ We can find critical path and instruction execution time for each instruction at the next page.



## [Answer-2]

Instruction class	Functional units used by the instruction class				
R-type	Instruction fetch	Register access	ALU	Register access	
Load word	Instruction fetch	Register access	ALU	Memory access	Register access
Store word	Instruction fetch	Register access	ALU	Memory access	
Branch	Instruction fetch	Register access	ALU		
Jump	Instruction fetch				

Instruction class	Instruction memory	Register read	ALU operation	Data memory	Register write	Total
R-type	200	50	100	0	50	400 ps
Load word	200	50	100	200	50	600 ps
Store word	200	50	100	200		550 ps
Branch	200	50	100	0		350 ps
Jump	200					200 ps

## [Answer-3]

- For the fixed-length clock implementation,

Clock cycle = 600 ps.

- For the variable clock implementation,

CPU clock cycle

$$= 600 \times 25\% + 550 \times 10\% + 400 \times 45\% + 350 \times 15\% + 200 \times 5\% = 447.5 \text{ ps}$$

$$\begin{aligned} \frac{\text{CPU Performance}_{\text{variable clock}}}{\text{CPU Performance}_{\text{single clock}}} &= \frac{\text{CPU execution time}_{\text{single clock}}}{\text{CPU execution time}_{\text{variable clock}}} \\ &= \frac{\text{IC} \times \text{CPU clock cycle}_{\text{single clock}}}{\text{IC} \times \text{CPU clock cycle}_{\text{variable clock}}} \\ &= \frac{\text{CPU clock cycle}_{\text{single clock}}}{\text{CPU clock cycle}_{\text{variable clock}}} = \frac{600}{447.5} = 1.34 \end{aligned}$$

- The variable clock implementation would be 1.34 times faster.
- But, it is hard to implement and its overhead is large.

# Problems with a Single-Cycle Implementation

- ❑ Too long a clock cycle with floating-point unit or complex instruction set
- ❑ Implementation techniques that reduce common case delay but do not reduce worst-case cycle time cannot be used.
- ❑ Some functional units must be duplicated.



**Inefficient both in performance and in hardware cost.**



- **Multicycle datapath**
  - ❖ Shorter clock cycle
  - ❖ Multiple clock cycles for each instruction