

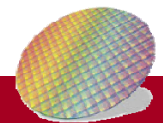


成功大學

National Cheng Kung University

# Chapter 4

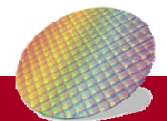
The Processor



# Introduction

- CPU performance factors
  - Instruction count
    - Determined by ISA and compiler
  - CPI and Cycle time
    - Determined by CPU hardware
- We will examine two MIPS implementations
  - A simplified version (Single-cycle implementation)
  - A more realistic pipelined version
  - Multi-cycle version is removed in this version)
- Implement simple subset, but shows most aspects
  - Memory reference: lw, sw
  - Arithmetic/logical: add, sub, and, or, slt
  - Control transfer: beq, j

$$\text{CPU Time} = \text{Instruction Count} \times \text{CPI} \times \text{Clock Cycle Time}$$



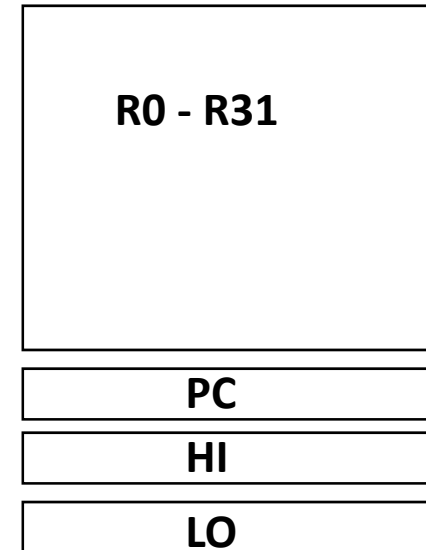
# Review: MIPS Instruction Set Architecture (ISA)



- Instruction Categories

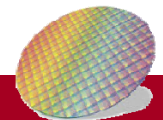
- Arithmetic
- Load/Store
- Jump and Branch
- Floating Point
  - coprocessor
- Memory Management
- Special

## Registers



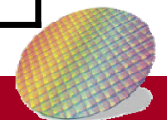
## 3 Instruction Formats: all 32 bits wide

OP	rs	rt	rd	sa	funct	R format
OP	rs	rt	immediate			I format
OP	jump target					J format



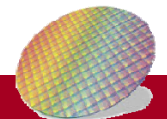
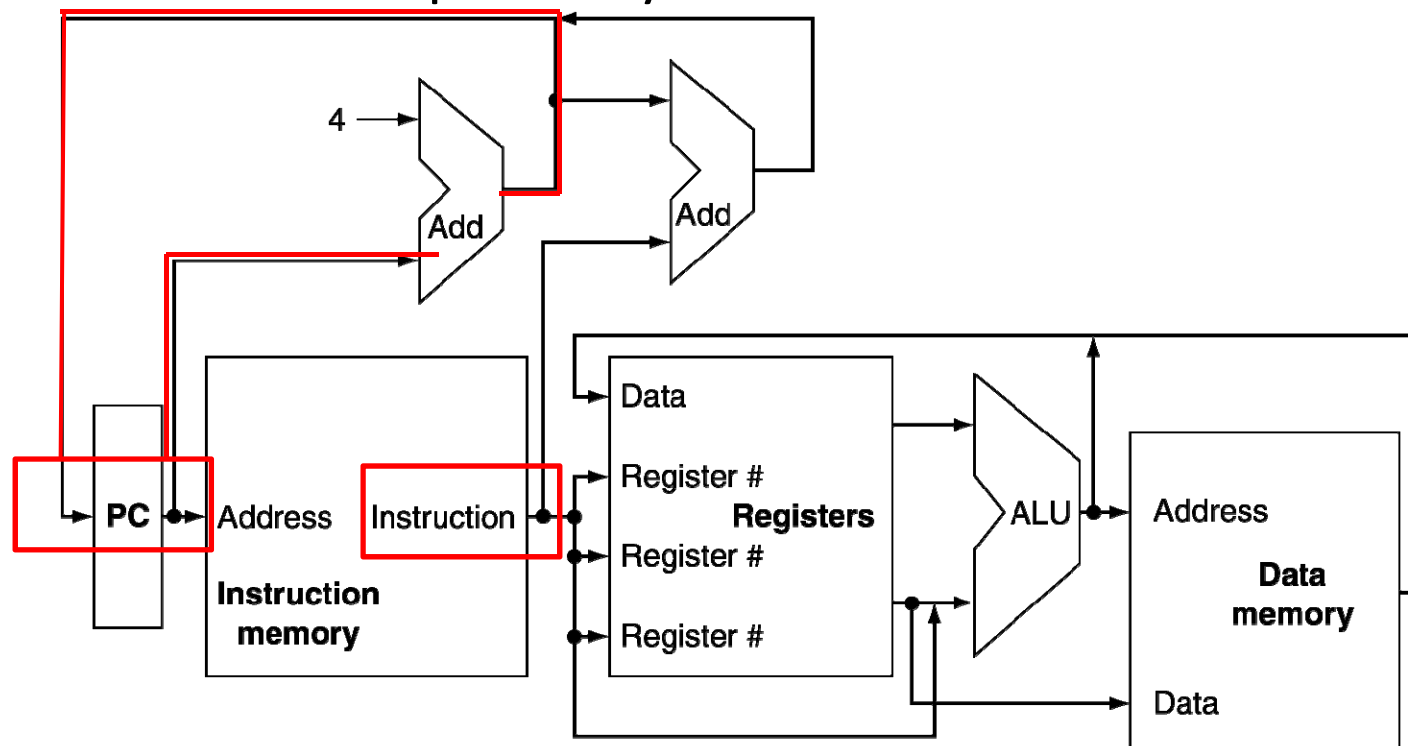
## Review: MIPS Register Convention

Name	Register Number	Usage	Preserve on call?
\$zero	0	constant 0 ( <b>hardware</b> )	n.a.
\$at	1	<b>reserved</b> for assembler	n.a.
\$v0 - \$v1	2-3	returned values	no
\$a0 - \$a3	4-7	arguments	<b>yes</b>
\$t0 - \$t7	8-15	temporaries	no
\$s0 - \$s7	16-23	saved values	<b>yes</b>
\$t8 - \$t9	24-25	temporaries	no
\$gp	28	global pointer	<b>yes</b>
\$sp	29	stack pointer	<b>yes</b>
\$fp	30	frame pointer	<b>yes</b>
\$ra	31	return addr ( <b>hardware</b> )	<b>yes</b>



# Instruction Execution

- PC (Program counter) is used to fetch instruction in the **instruction** memory)
- After **instruction** is obtained, **register numbers** in instructions is used to read registers in register files.
- $PC \leftarrow PC + 4$  for sequentially execution



# Different actions for different instruction classes

- Use **ALU** to calculate
  - Arithmetic result
  - Memory address for load/store
  - Branch target address
- Access data memory for load/store
- $PC \leftarrow$  target address

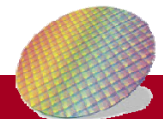
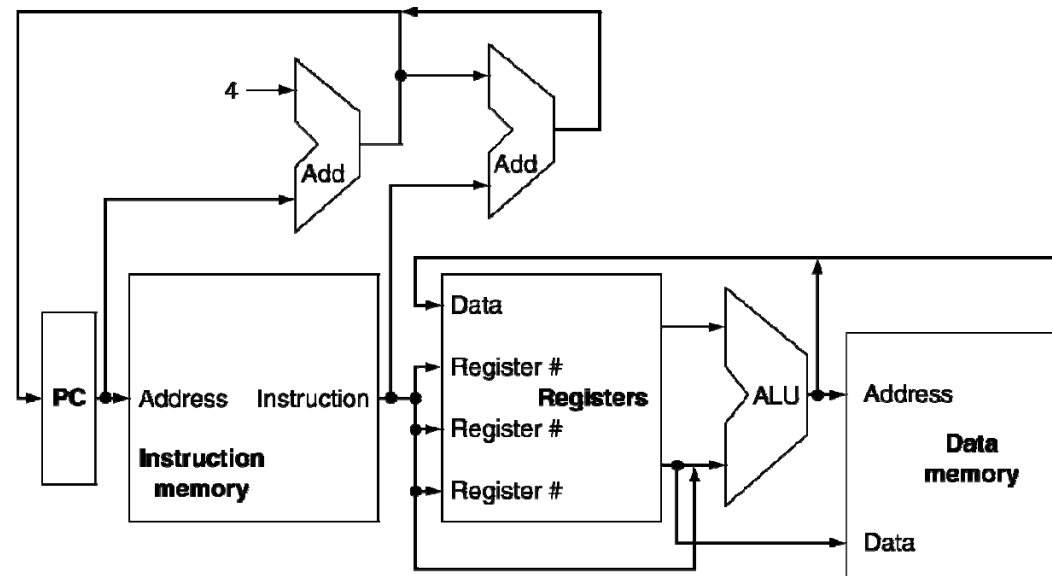
add \$t0, \$s1, \$s2

lw \$s1, 20(\$s2)

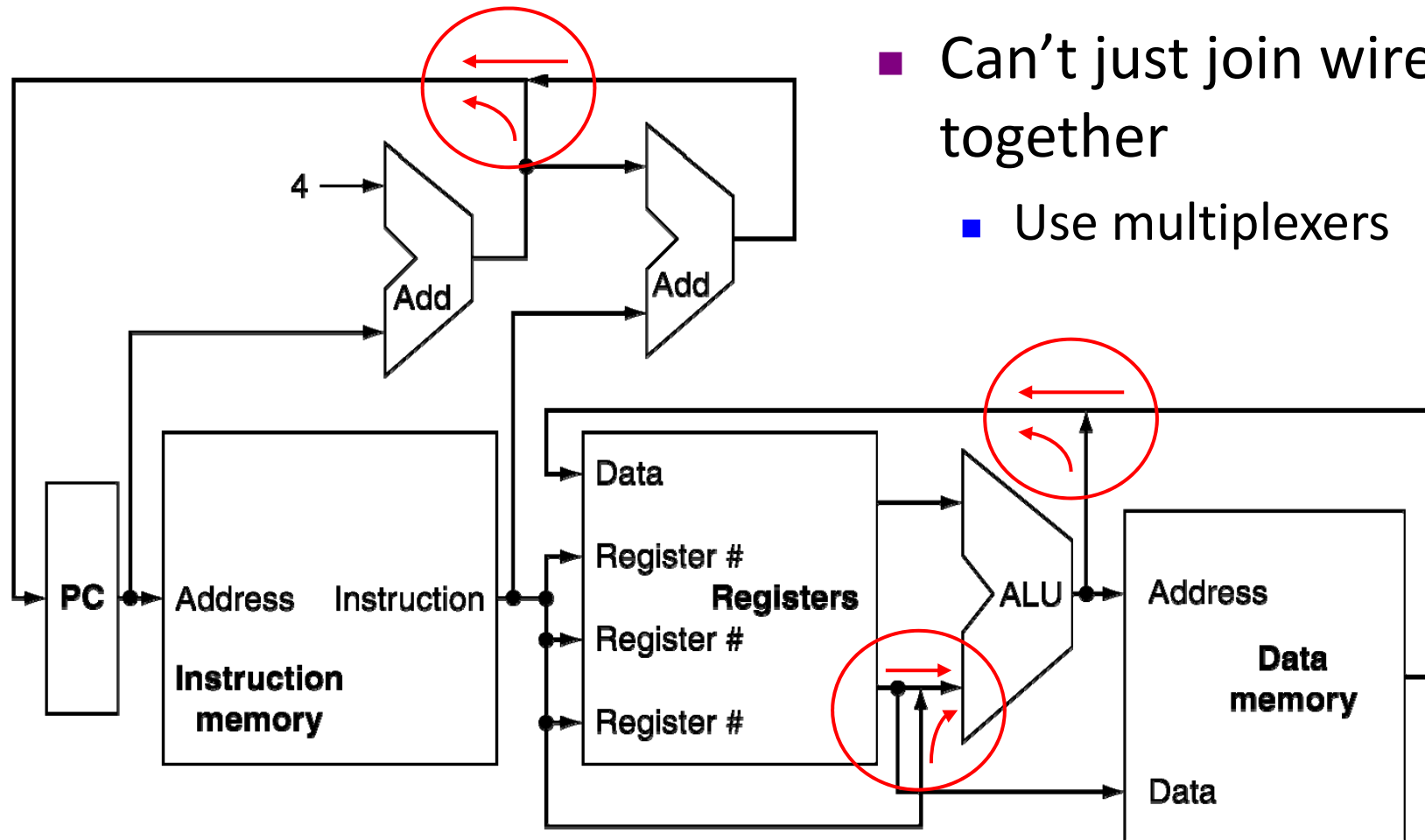
bne \$t0, \$s5, Exit

lw \$s1, 20(\$s2)

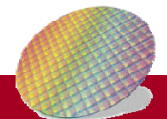
j Loop



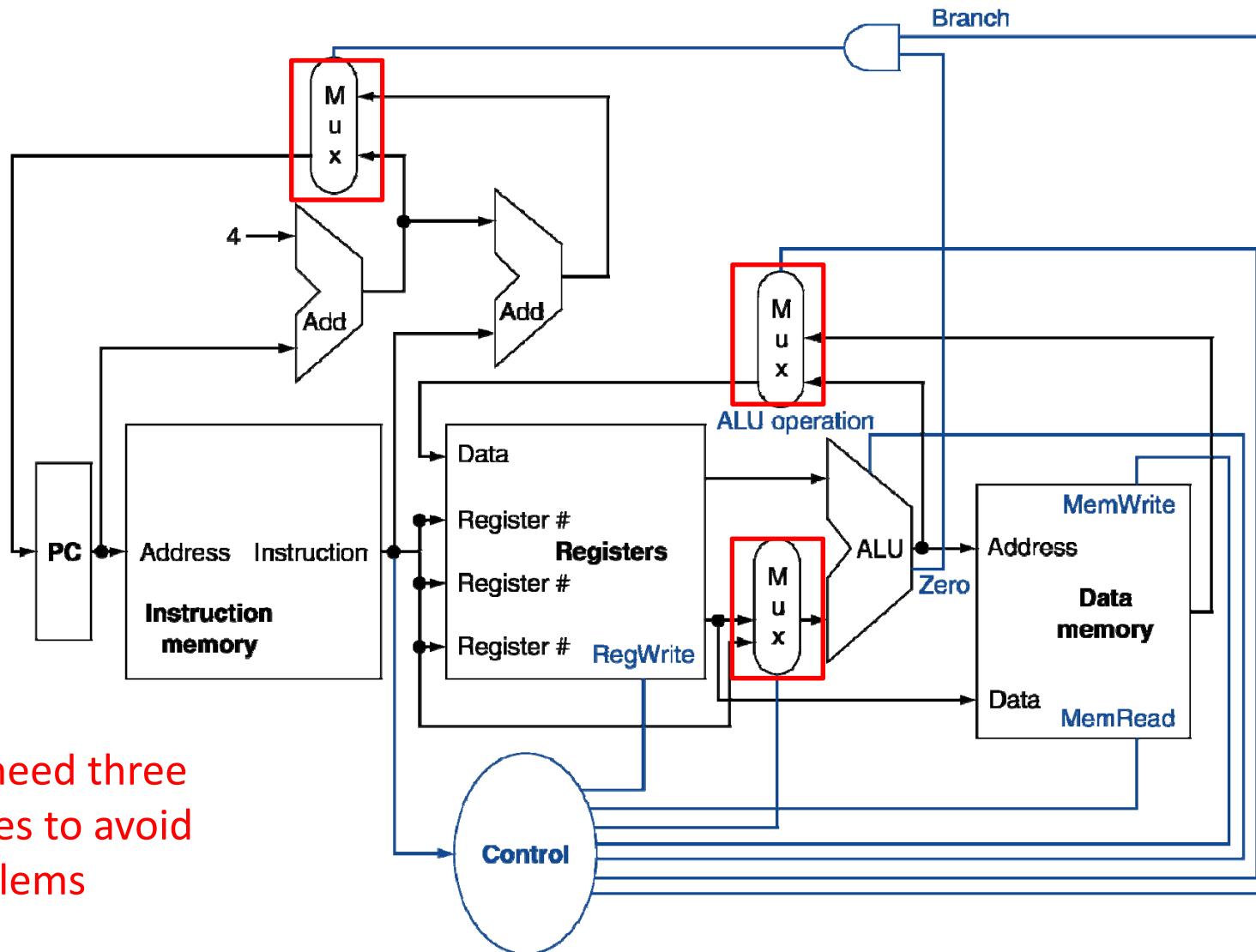
# Multiplexers



- Can't just join wires together
  - Use multiplexers

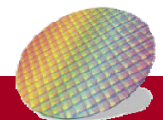


# CPU Overview



We need three  
Muxes to avoid  
problems

Details of each Mux and Control will be introduced later





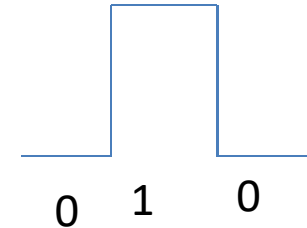
# Logic Design Basics

- Information encoded in binary

- Low voltage = 0, High voltage = 1

- One wire per bit

- Multi-bit data encoded on multi-wire buses



- Combinational element (See <sup>32-bit bus</sup> next slide)

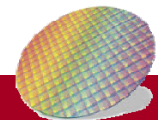
- Operate on data

- Output is a function of input

- State (sequential) elements

- Output is a function of input and current states

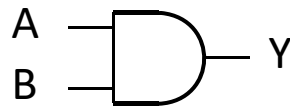
- Store information



# Review: Combinational Elements

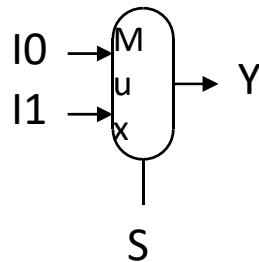
- AND-gate

- $Y = A \& B$



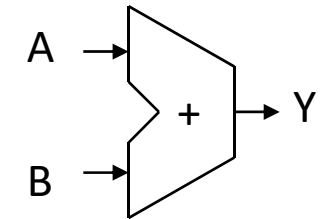
- Multiplexer

- $Y = S ? I1 : I0$



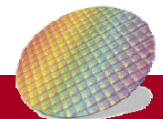
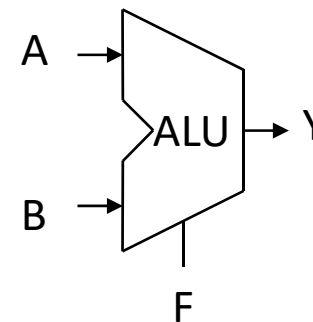
- Adder

- $Y = A + B$



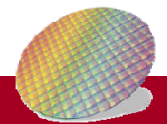
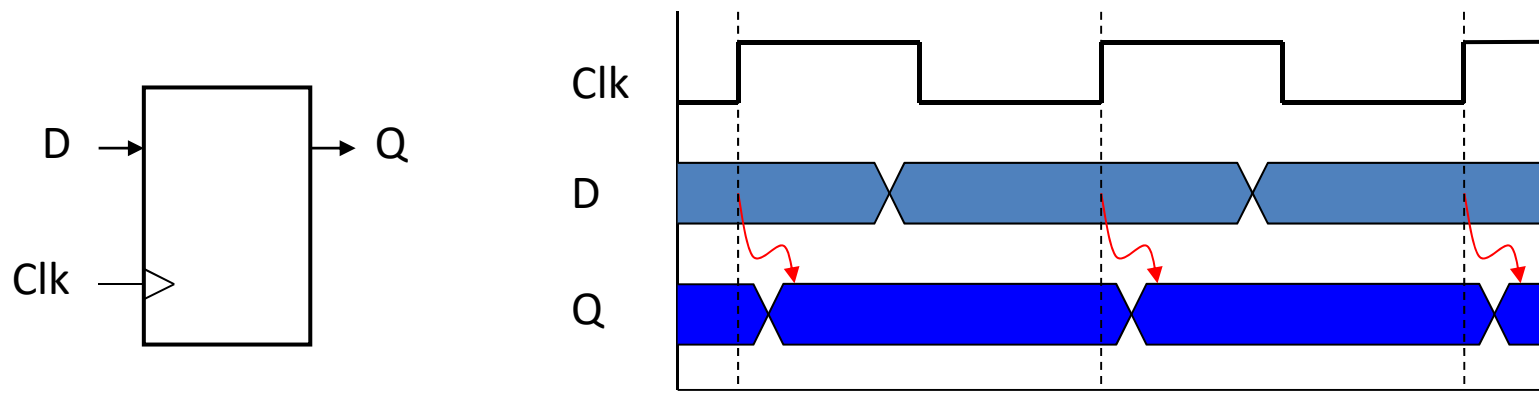
- Arithmetic/Logic Unit

- $Y = F(A, B)$



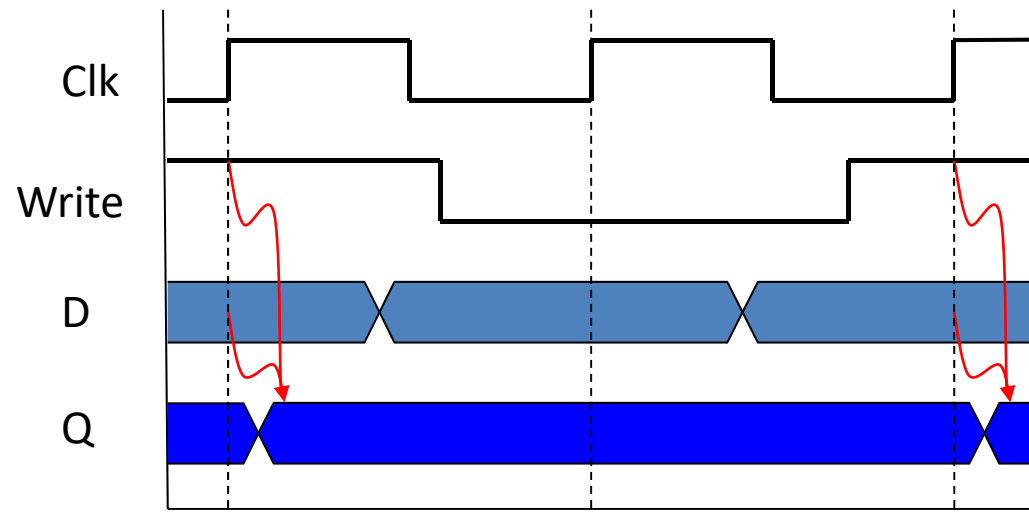
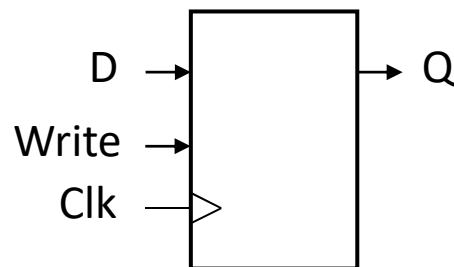
# Review: Sequential Elements

- Register: stores data in a circuit
  - Uses a **clock** signal to determine when to update the stored value
  - **Edge-triggered**: update when **Clk** changes (0-> 1 or 1-> 0)
  - The following figure is **positive edge-triggered**: update when Clk changes from **0 to 1**

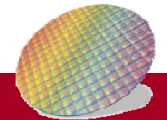


# Review: Sequential Elements (with write enable)

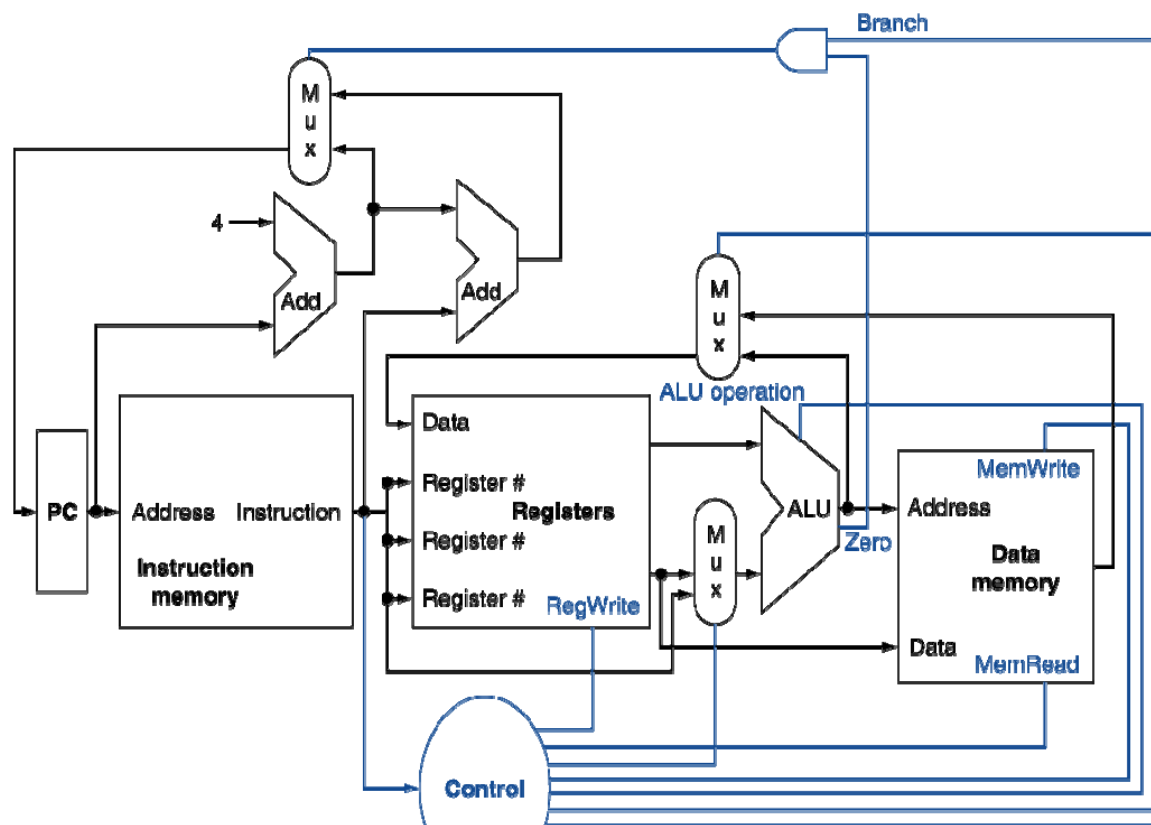
- Register with write control
  - Only updates on clock edge when **write control** input is 1
  - Used when stored value is required later



Q is not changed when **Write=0**

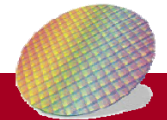
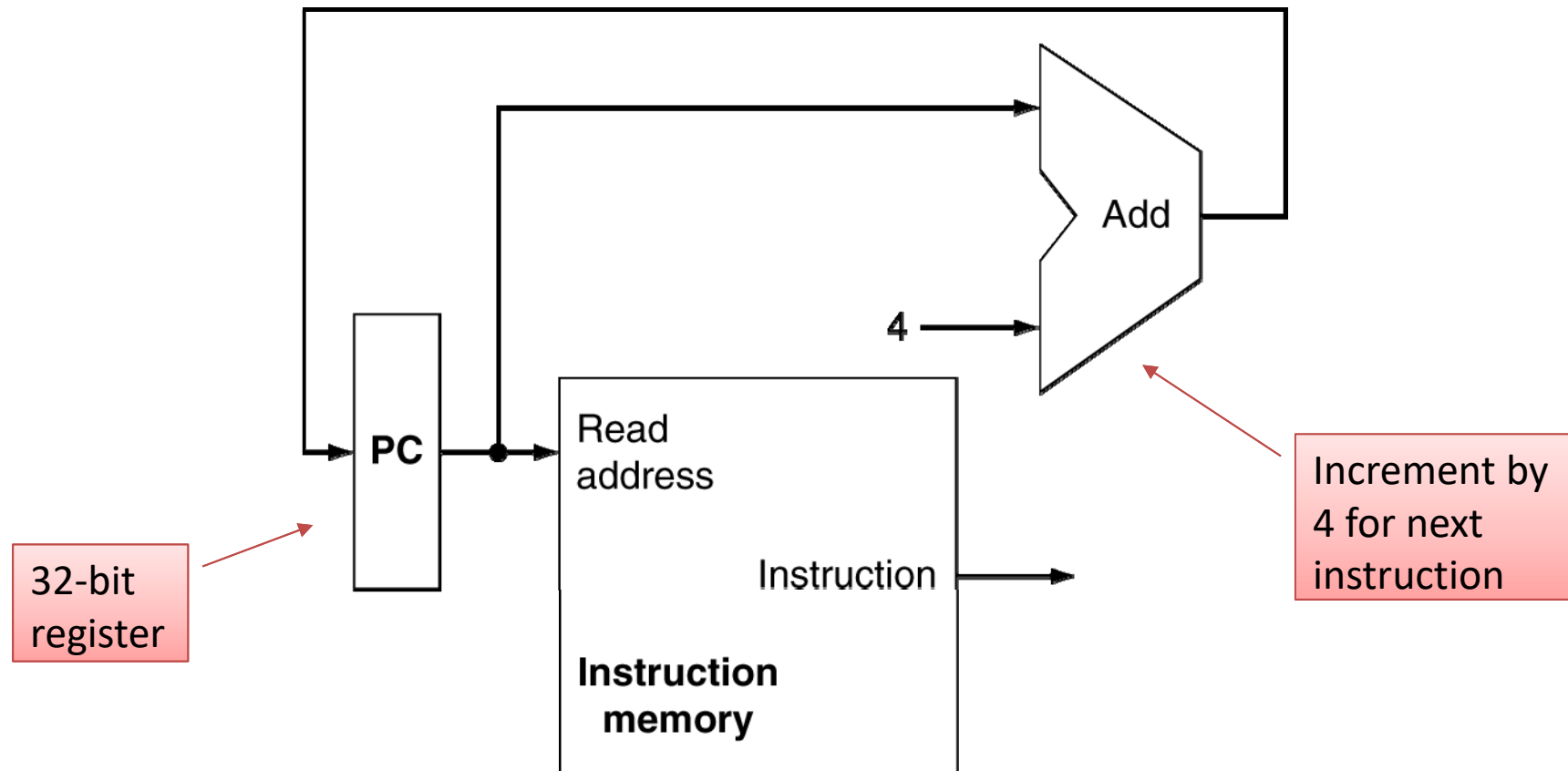


- Datapath : Elements that process data and addresses in the CPU
  - Registers, ALUs, mux's, memories, ...



We will show how to build MIPS datapath

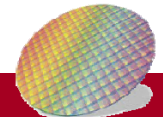
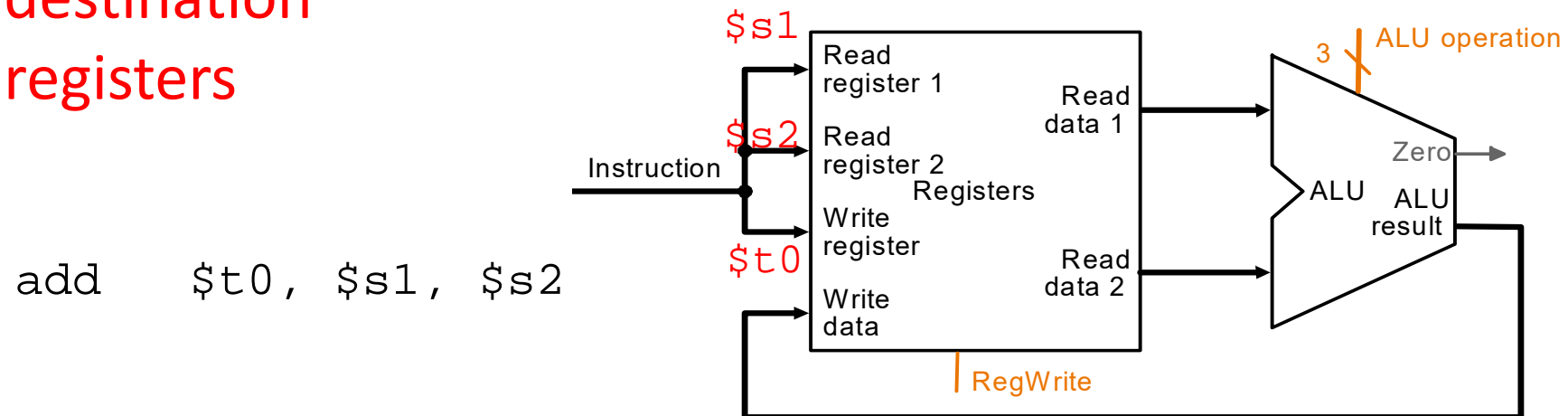
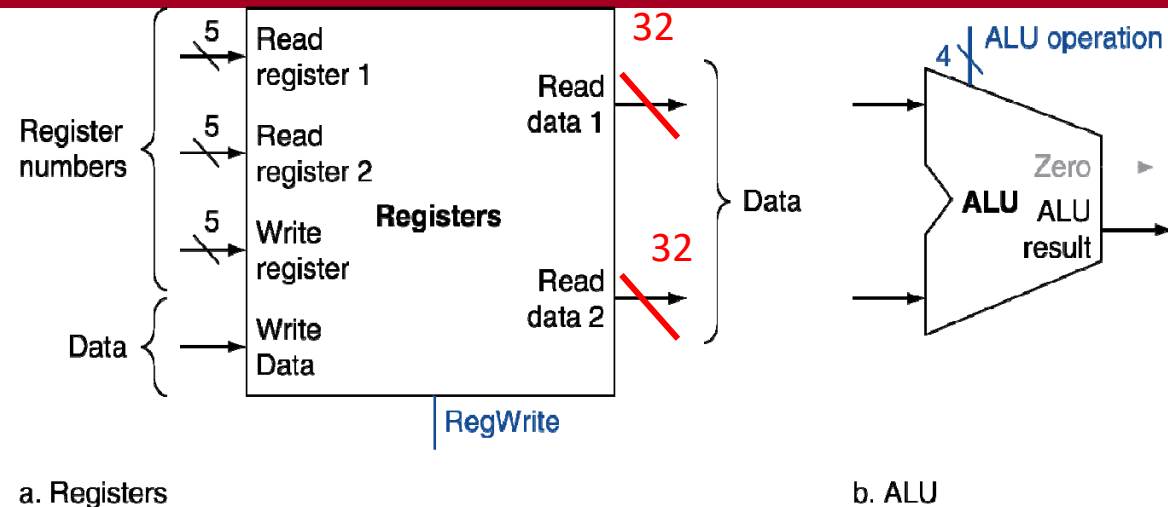
# Instruction Fetch





# R-Format Instructions

- Read two register operands
- Perform arithmetic/logical operation
- Write results into destination registers

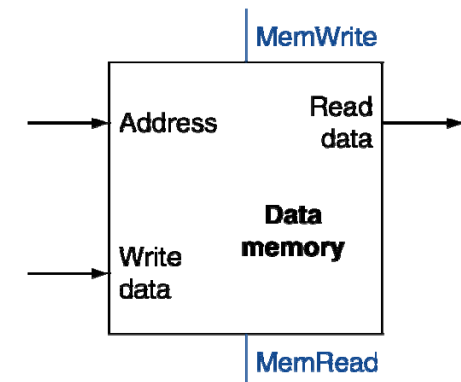
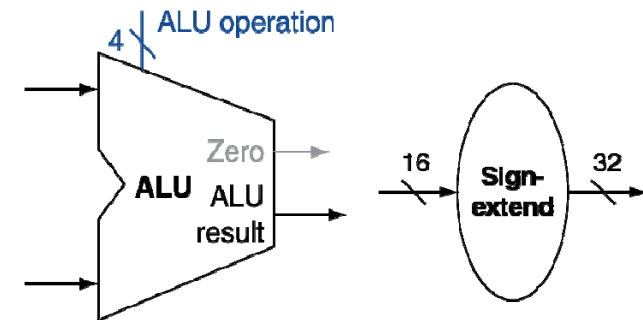
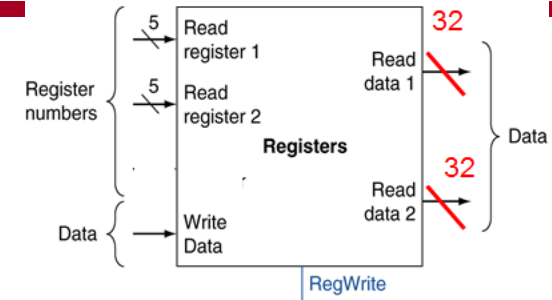




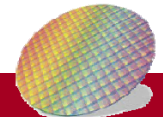
成功大學

# Load/Store Instructions (need 4 components)

- Read register operands => register files
- Calculate address using 16-bit offset
  - Use **ALU**, but **sign-extend offset**
- Load/store: **read** memory and update register, and **write** register value to memory
  - Need data memory



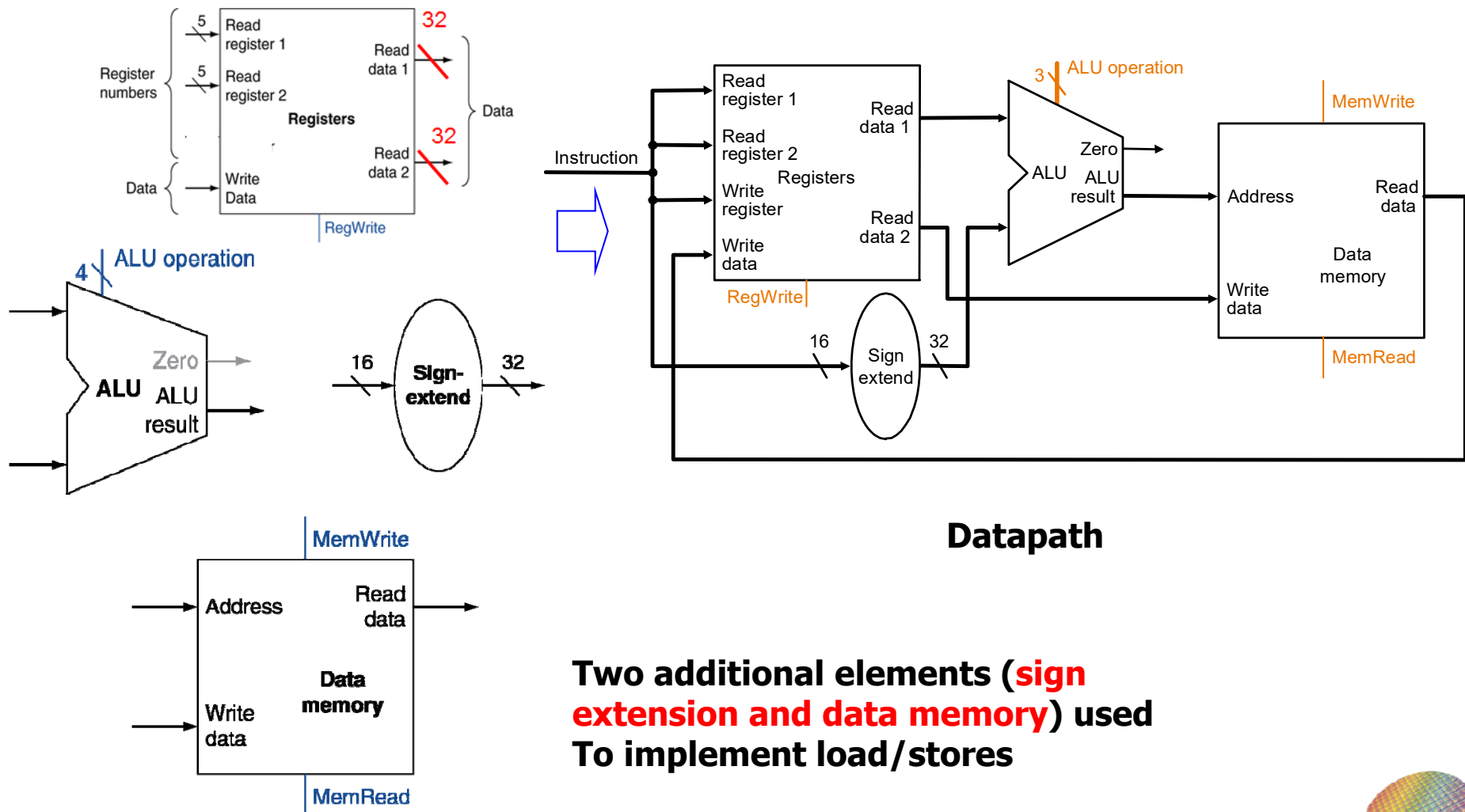
**lw** \$t0, 4(\$s3) #load word from memory  
**sw** \$t0, 8(\$s3) #store word to memory





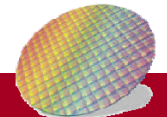
# Datapath: Load/Store Instruction

- Load/store



Datapath

Two additional elements (**sign extension** and **data memory**) used To implement load/stores





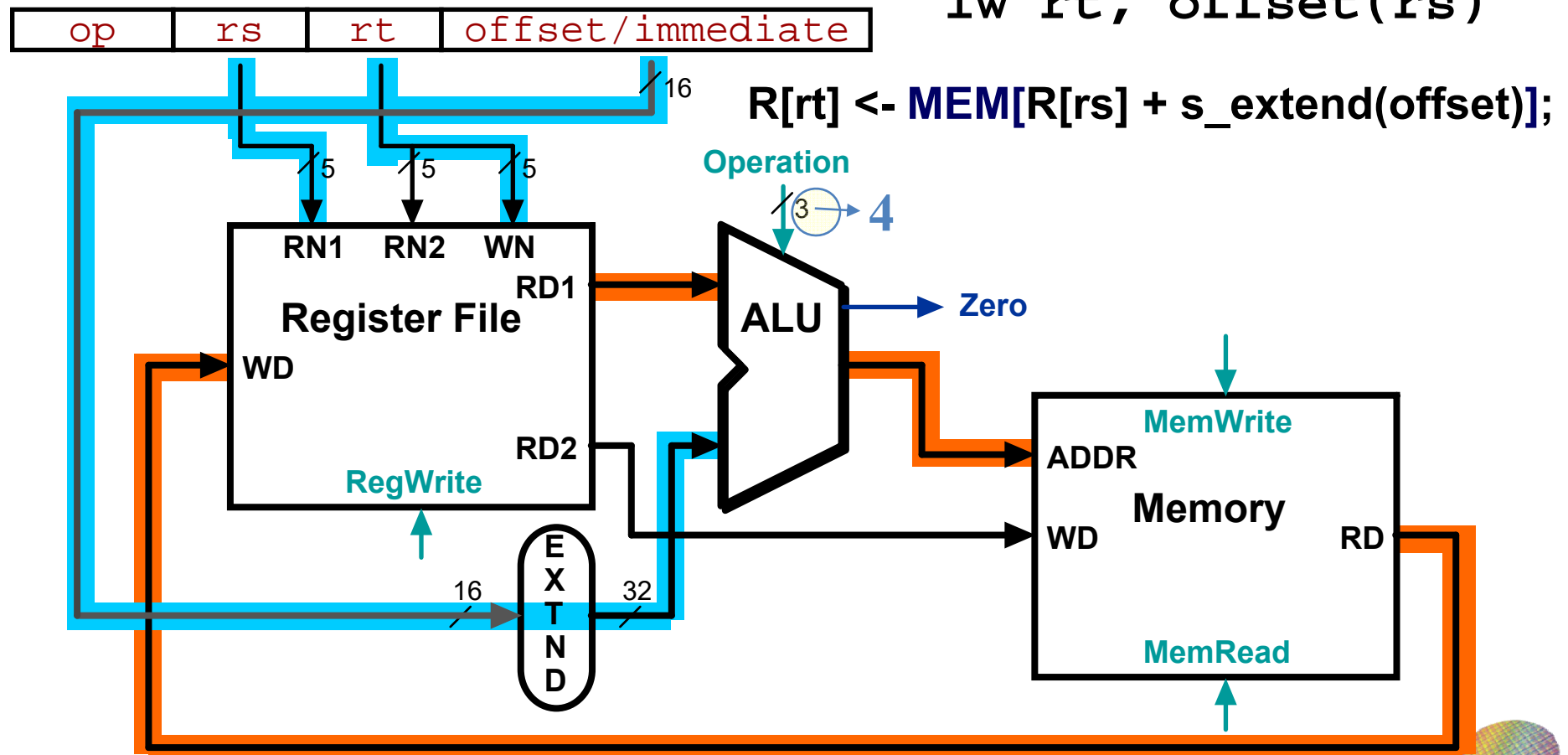
# Animating the Datapath- load

- Load

e.g. `lw $t0, 4($s3)`

- RN1: register number 1
- RN2: register number 2
- WN: register number that will be written
- WD: write data

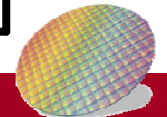
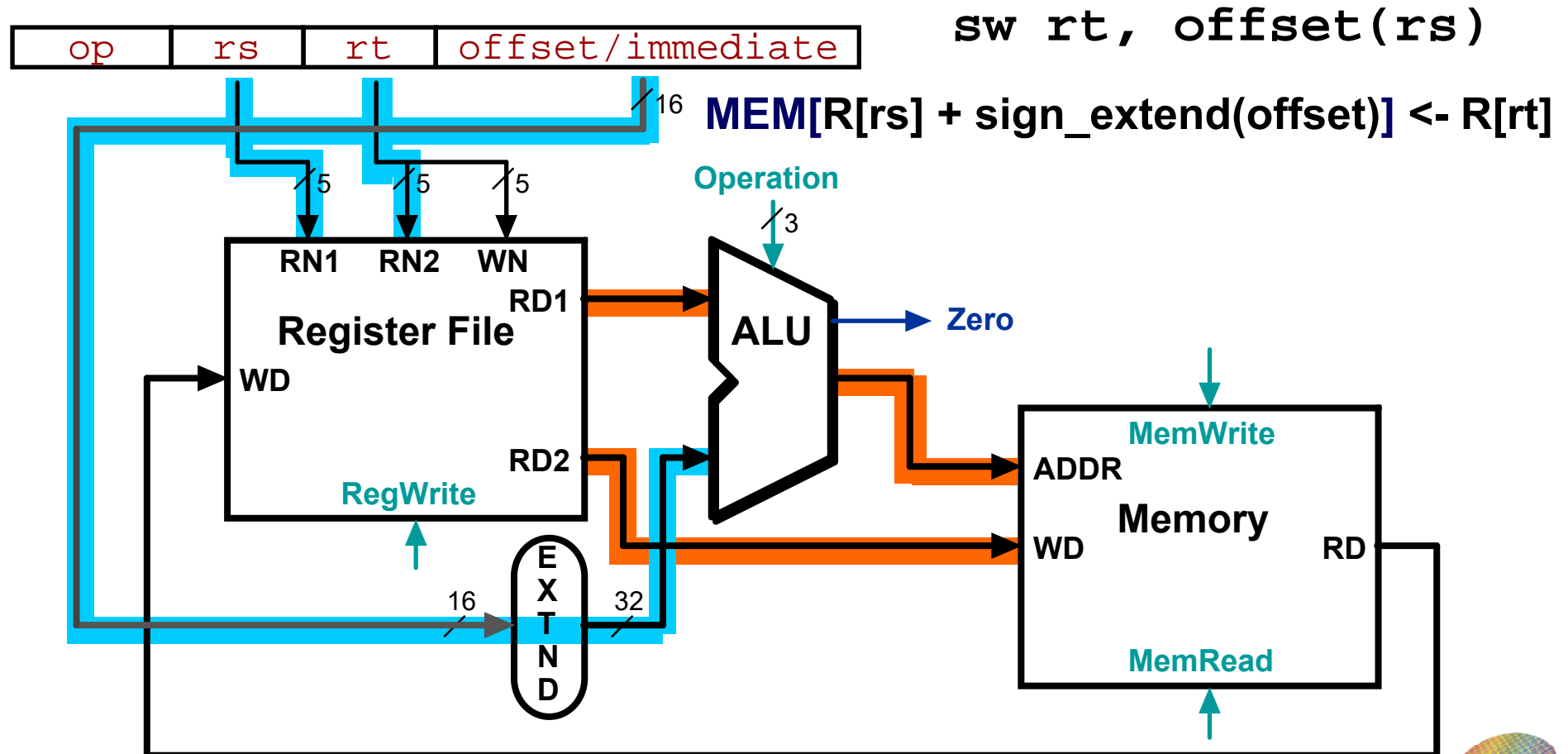
`lw rt, offset(rs)`



# Animating the Datapath- store

- store

sw \$t0, 8(\$s3)



# Review: Specifying Branch Destinations

- MIPS conditional branch instructions:

op	rs	rt	offset
6 bits	5 bits	5 bits	16 bits

2000 beq \$s0 \$t1 2

2004 ....

2008 ...

200C

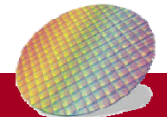
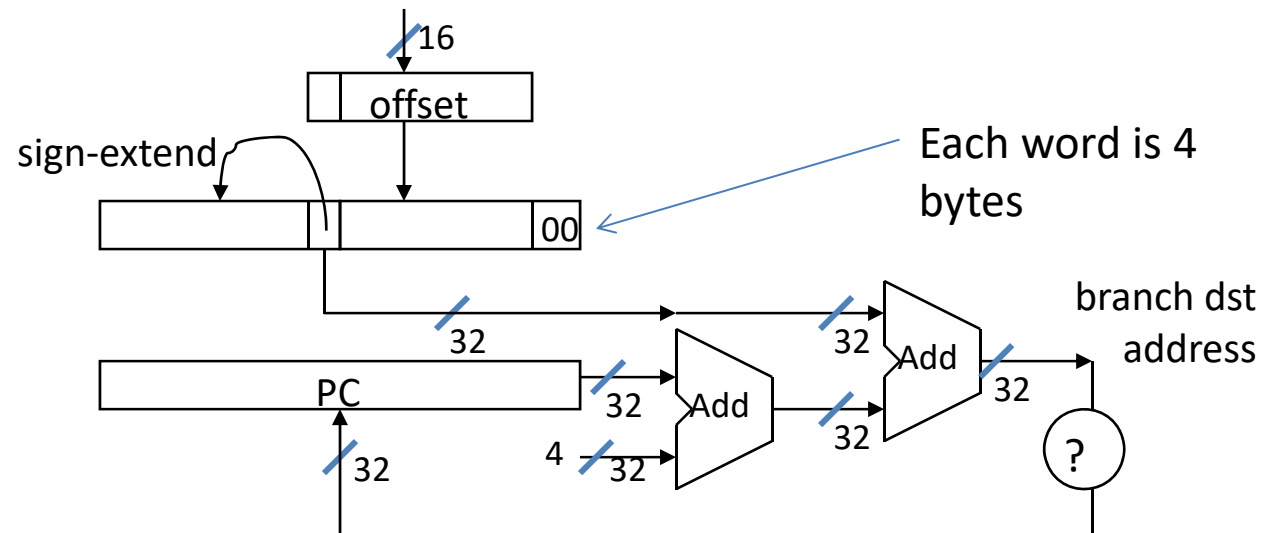
Target Address (address of next instruction) = ?

200C

- PC-relative addressing

– Target address =  $PC + \text{offset} \times 4$

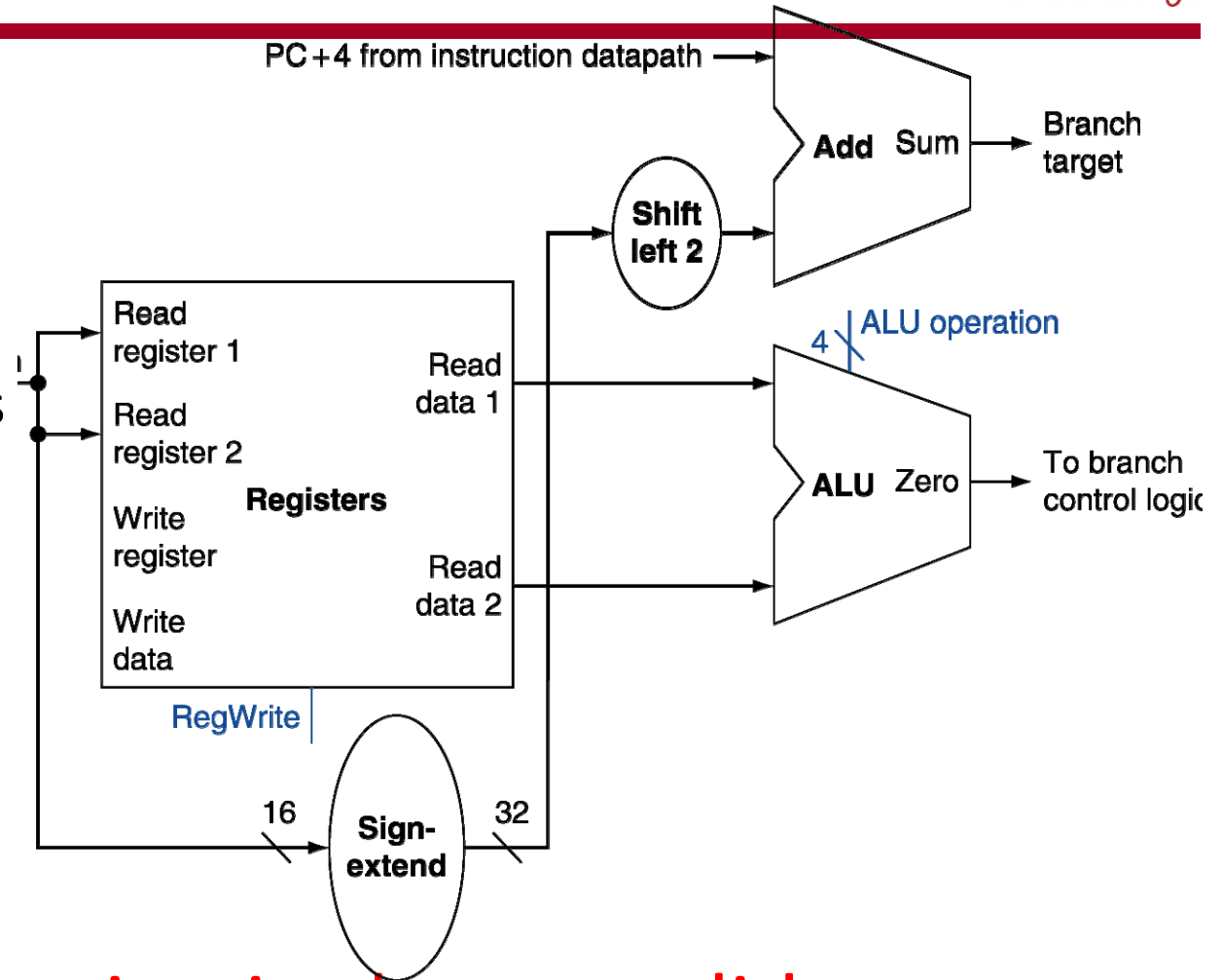
– PC already incremented by 4 by this time  
from the low order 16 bits of the branch instruction



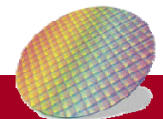


# Datapath: Branch Instructions

- Read register operands
- Compare operands
  - Use **ALU**, subtract and check Zero output
- Calculate target address
  - **Sign-extend** offset
  - Shift left **2 bits** (word displacement)
  - Add to **PC + 4** (already calculated by instruction fetch)



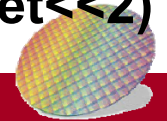
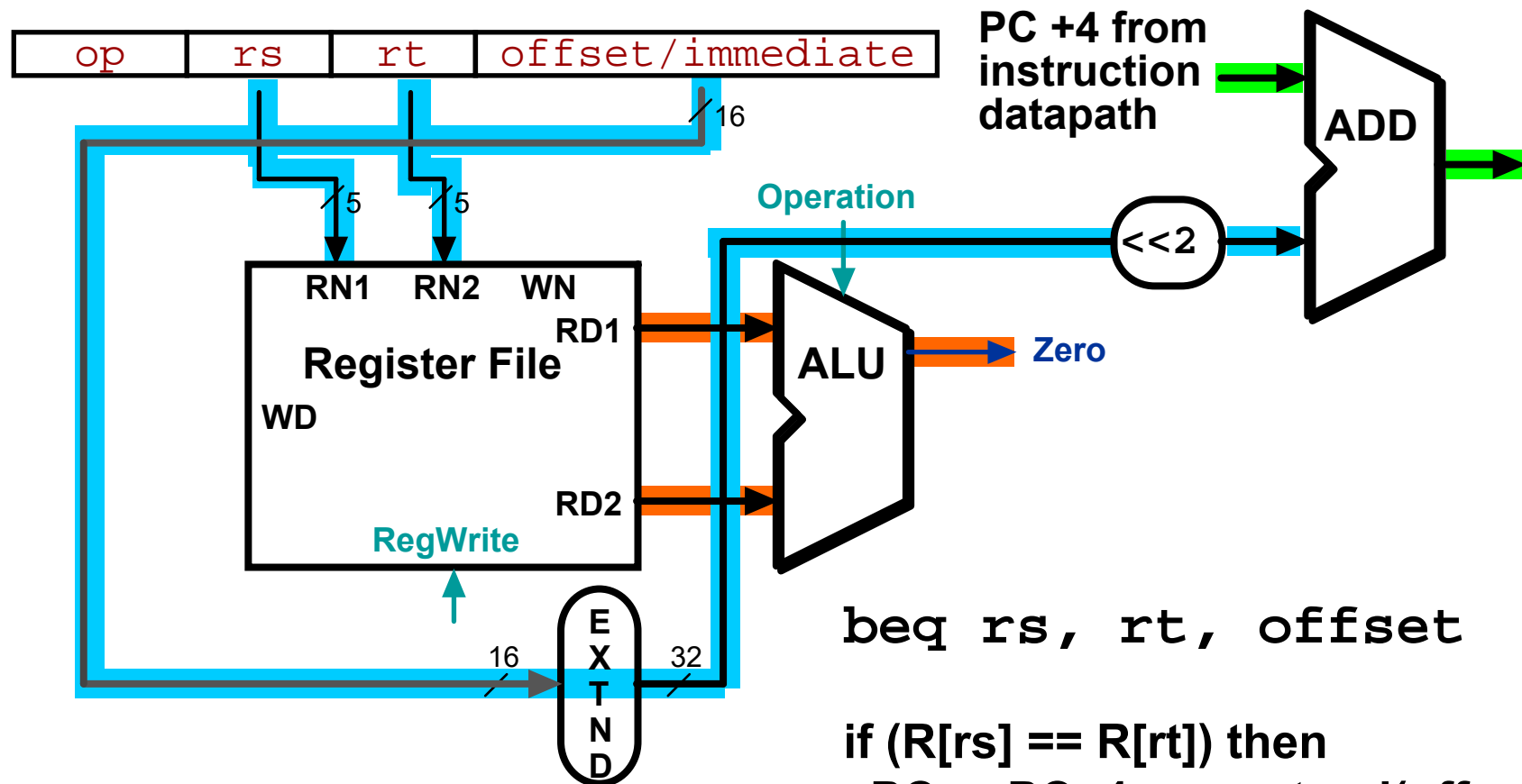
See animation in the next slide



# Animating the Datapath (beq)

- beq

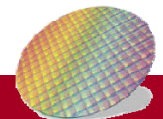
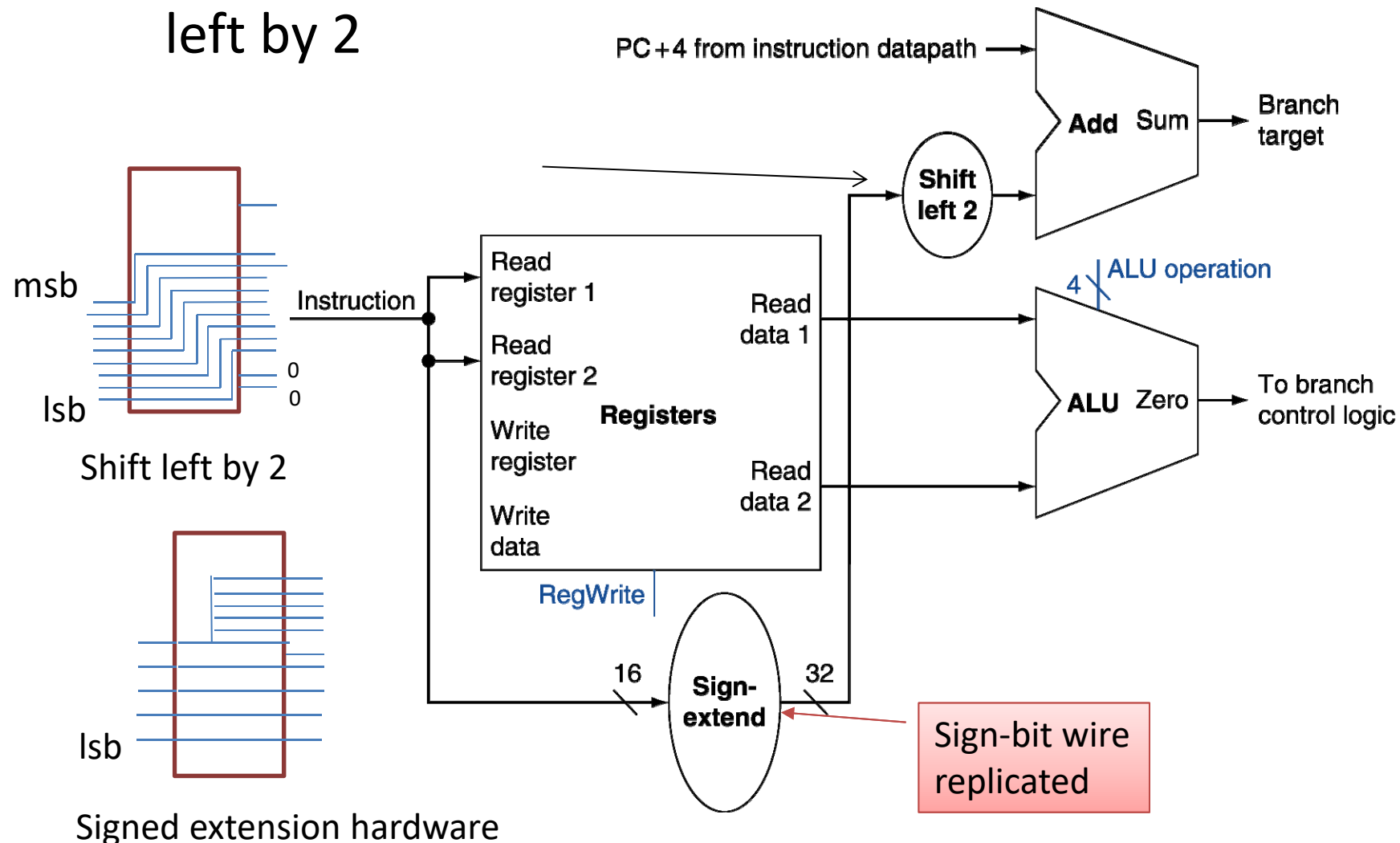
e.g. beq \$s0 \$t1 2





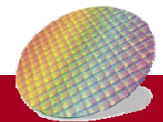
# Sign-extension and shift left by 2 hardware

- Simple hardware is used for sign extension and shift left by 2



## Composing the Elements

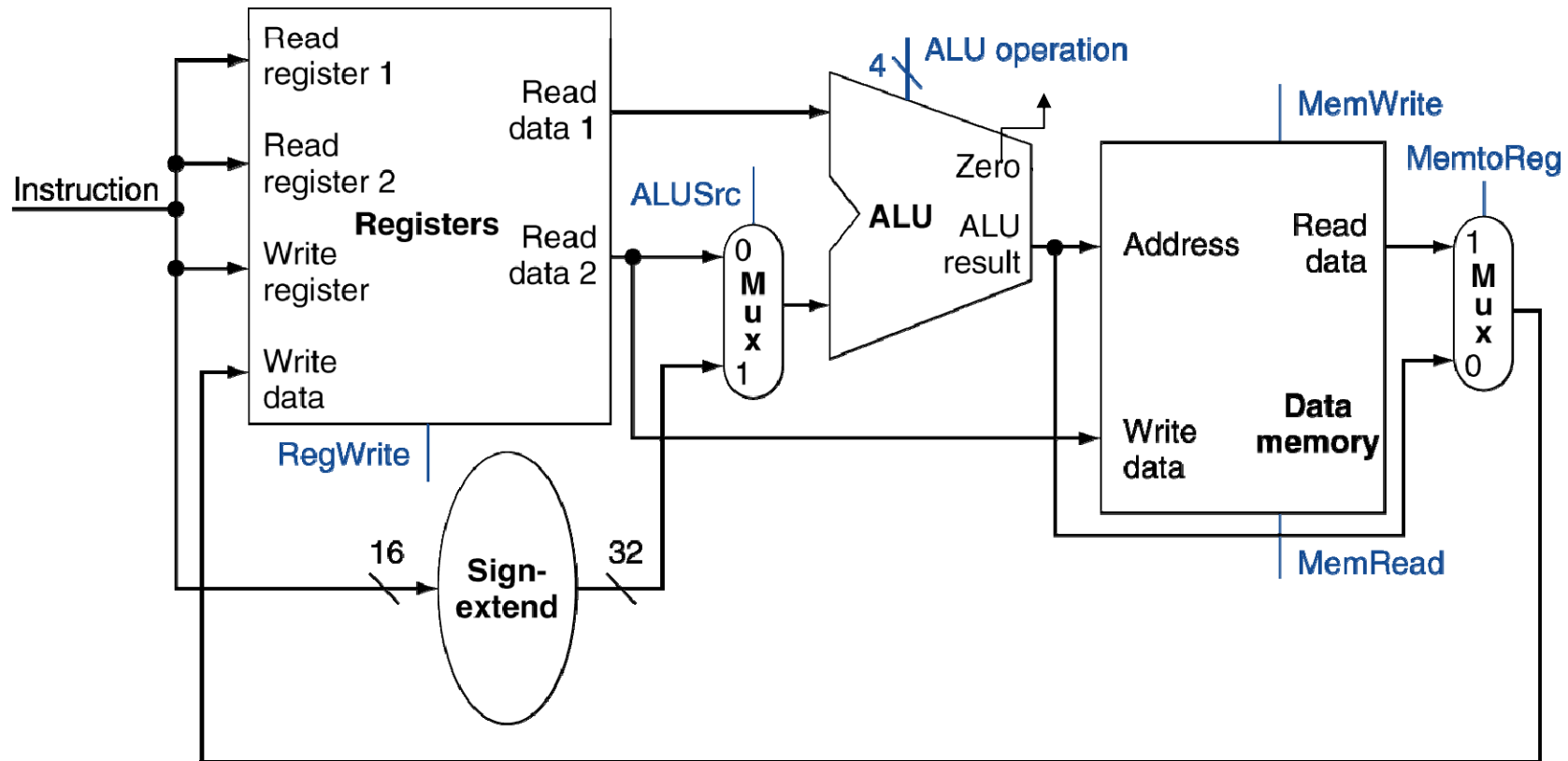
- Make Data path do an instruction in **one clock cycle**
  - Each datapath **element** can only do **one** function at a time
  - Hence, we need **separate instruction** and **data** memories
- Use **multiplexers** where alternate data sources are used for different instructions



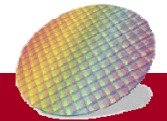


# R-Type/Load/Store Datapath

## A Single Cycle Datapath

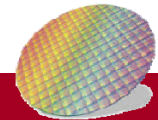
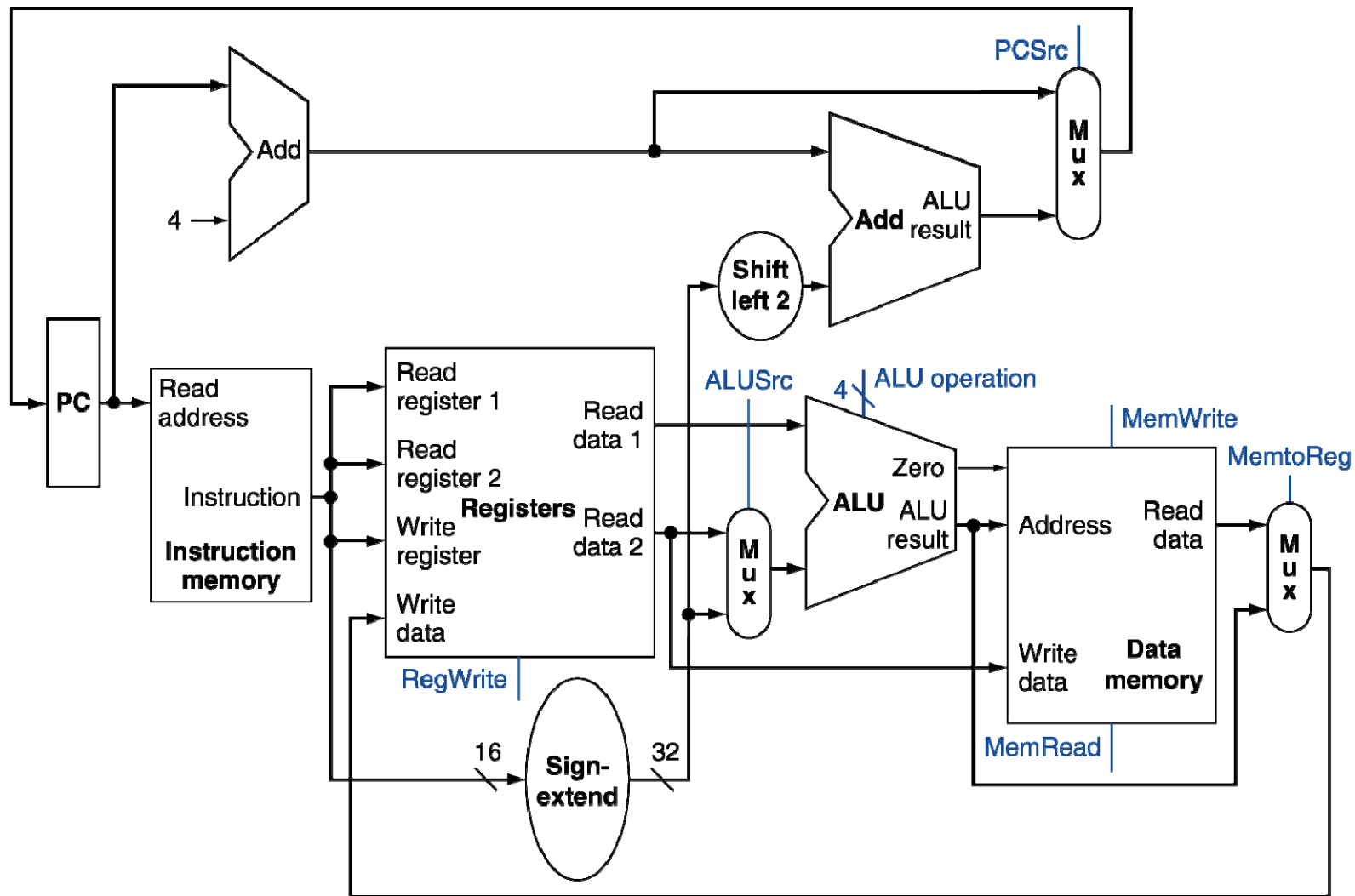


Correct Control signal (RegWrite, ALUSrc, ALU operation, MemWrite, MemtoReg, MemRead) are needed to make sure correct operation is done





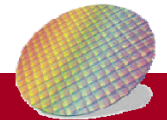
# Full Datapath (Single Cycle Datapath)





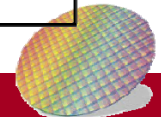
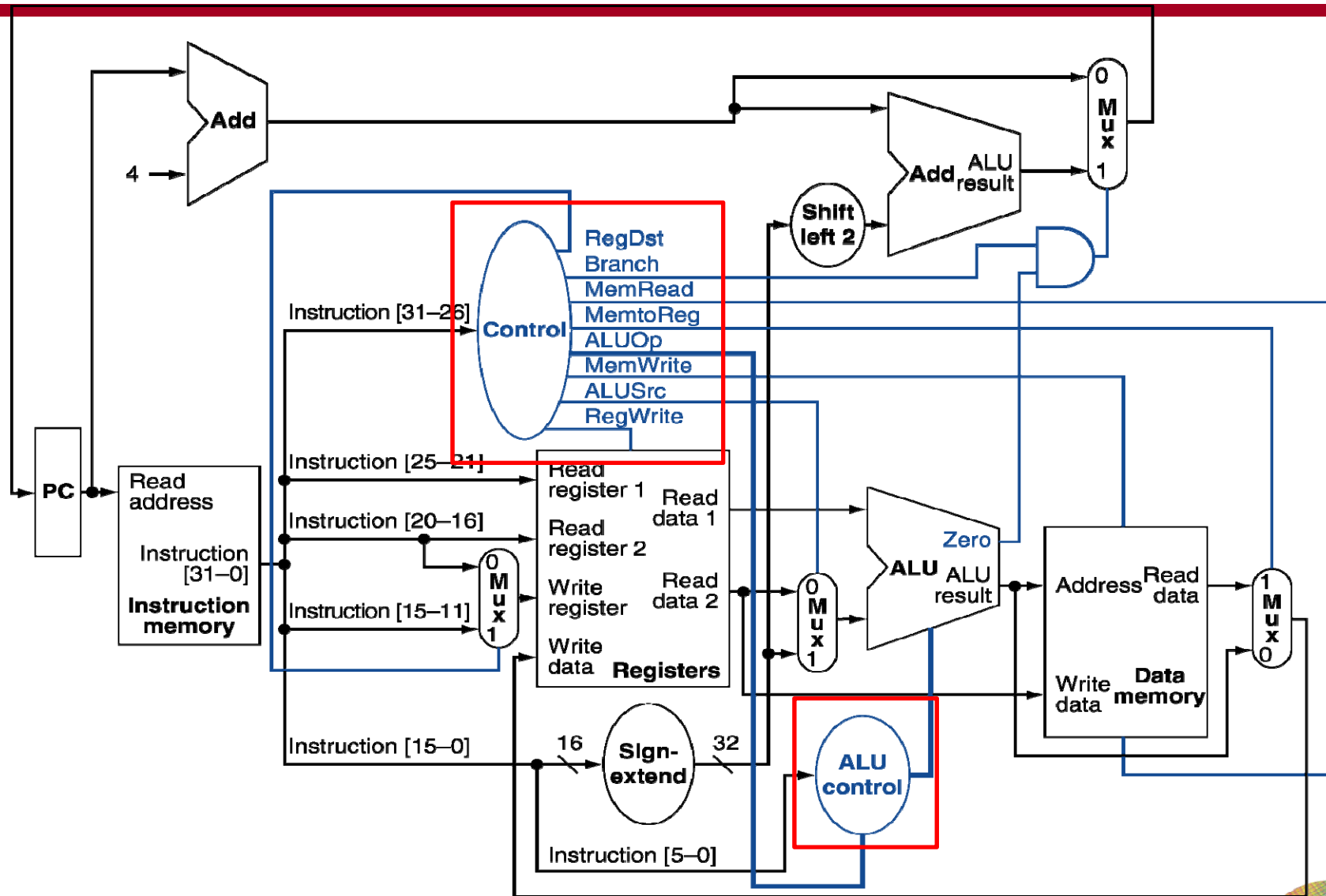
## Control for the single-cycle CPU

- Designing a processor
- A single-cycle implementation (the datapath)
- **Control for the single-cycle CPU**
  - Control of CPU operations
  - ALU controller
  - Main controller





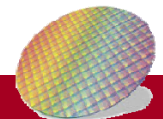
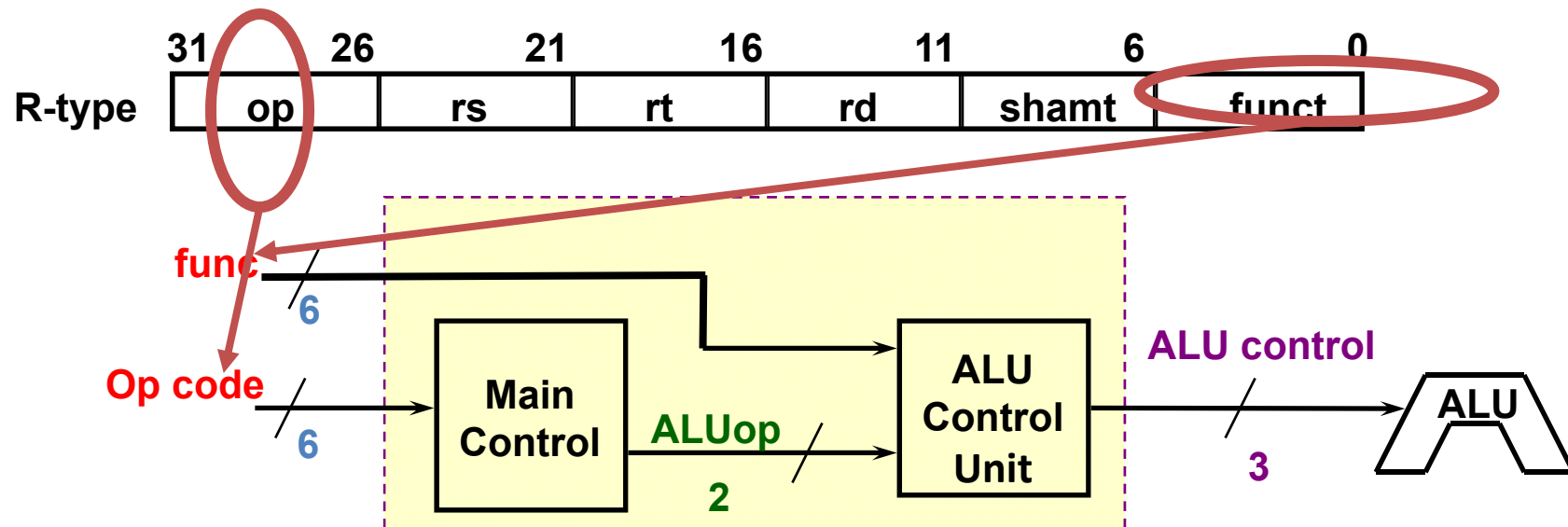
## Next: Building Datapath With Control





# Main Control and ALU Control

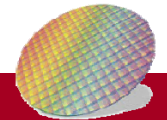
- **Main Control**: Based on opcode: generate RegDst, Branch, MemRead MemtoReg, ALUOp MemWrite, ALUSrc, RegWrite
- **ALU Control**: Based on **2-bit ALUOp** and the **6-bit func field** of instruction, the ALU control unit generates the 3-bit ALU control field



# Deciding ALU Control

- Assume 2-bit ALUOp derived from opcode
  - Combinational logic derives ALU control

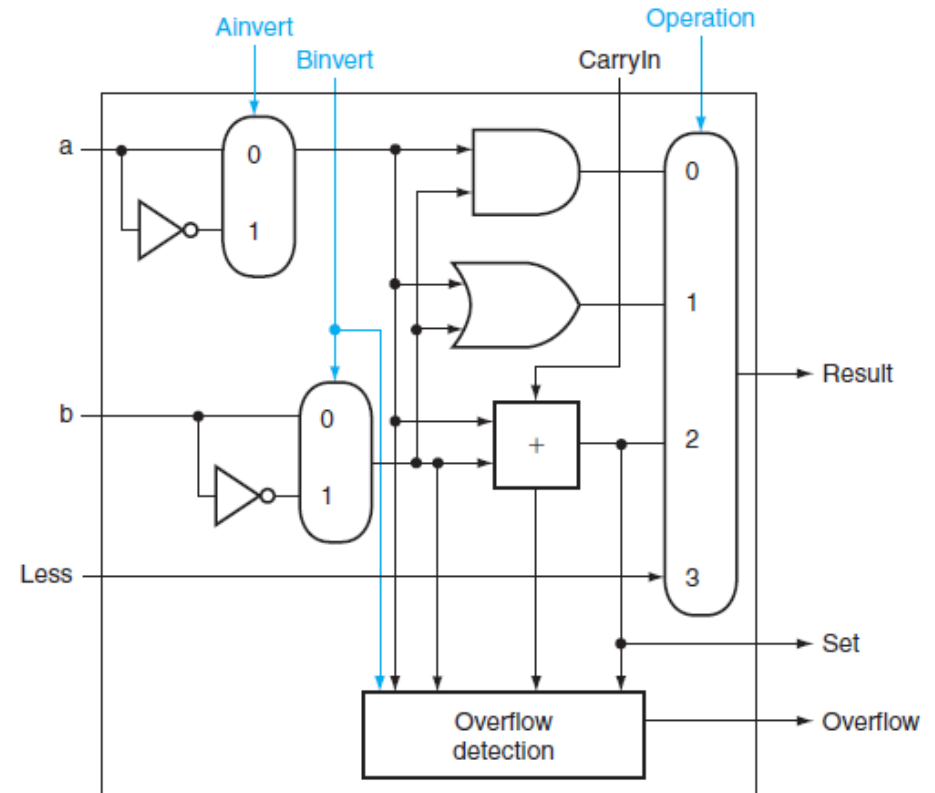
opcode	ALUOp	Operation	funct	ALU function	ALU control
lw	00	load word	XXXXXX	add	?
sw	00	store word	XXXXXX	add	?
beq	01	branch equal	XXXXXX	subtract	?
R-type	10	add	100000	add	?
		subtract	100010	subtract	?
		AND	100100	AND	?
		OR	100101	OR	?
		set-on-less-than	101010	set-on-less-than	?



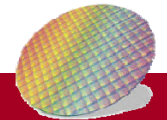
# ALU Control

- ALU Control has 4 four bits:  
**Ainvert**, **Binvert**, and  
**Operation (2 bits)**

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



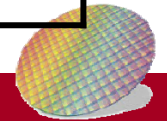
If **NOR** is not supported, we just need **three** bits (MSB bit is always 0)



# ALU Control

- ALU used for
  - Load/Store: function = add
  - Branch: function = subtract
  - R-type: function depends on **funct** field
- Assume 2-bit ALUOp derived from opcode
  - Combinational logic derives ALU control

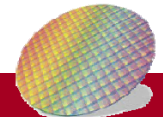
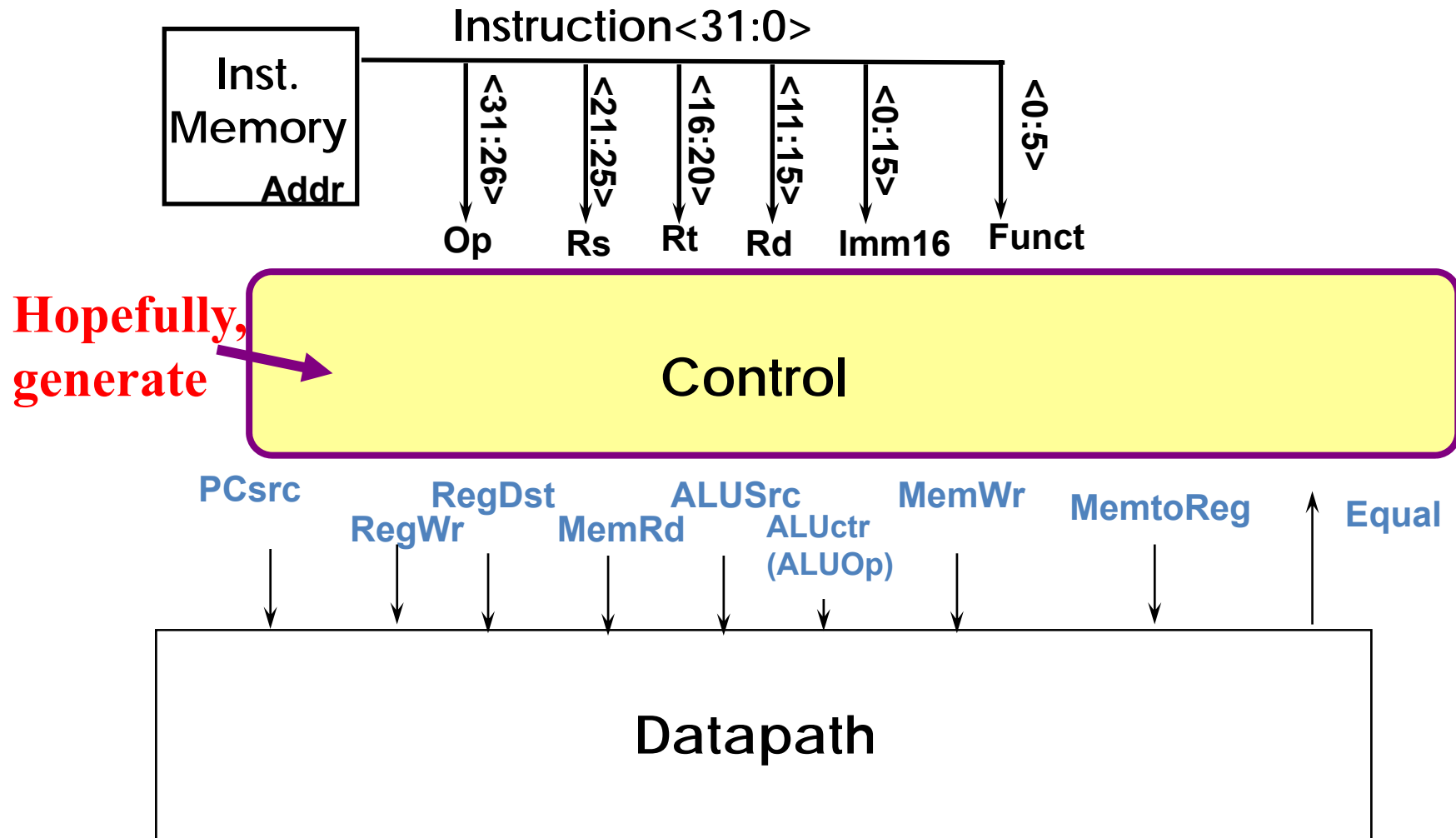
opcode	ALUOp	Operation	funct	ALU function	ALU control
lw	00	load word	XXXXXX	add	010
sw	00	store word	XXXXXX	add	010
beq	01	branch equal	XXXXXX	subtract	110
R-type	10	add	100000	add	010
		subtract	100010	subtract	110
		AND	100100	AND	000
		OR	100101	OR	001
		set-on-less-than	101010	set-on-less-than	111





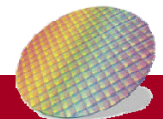
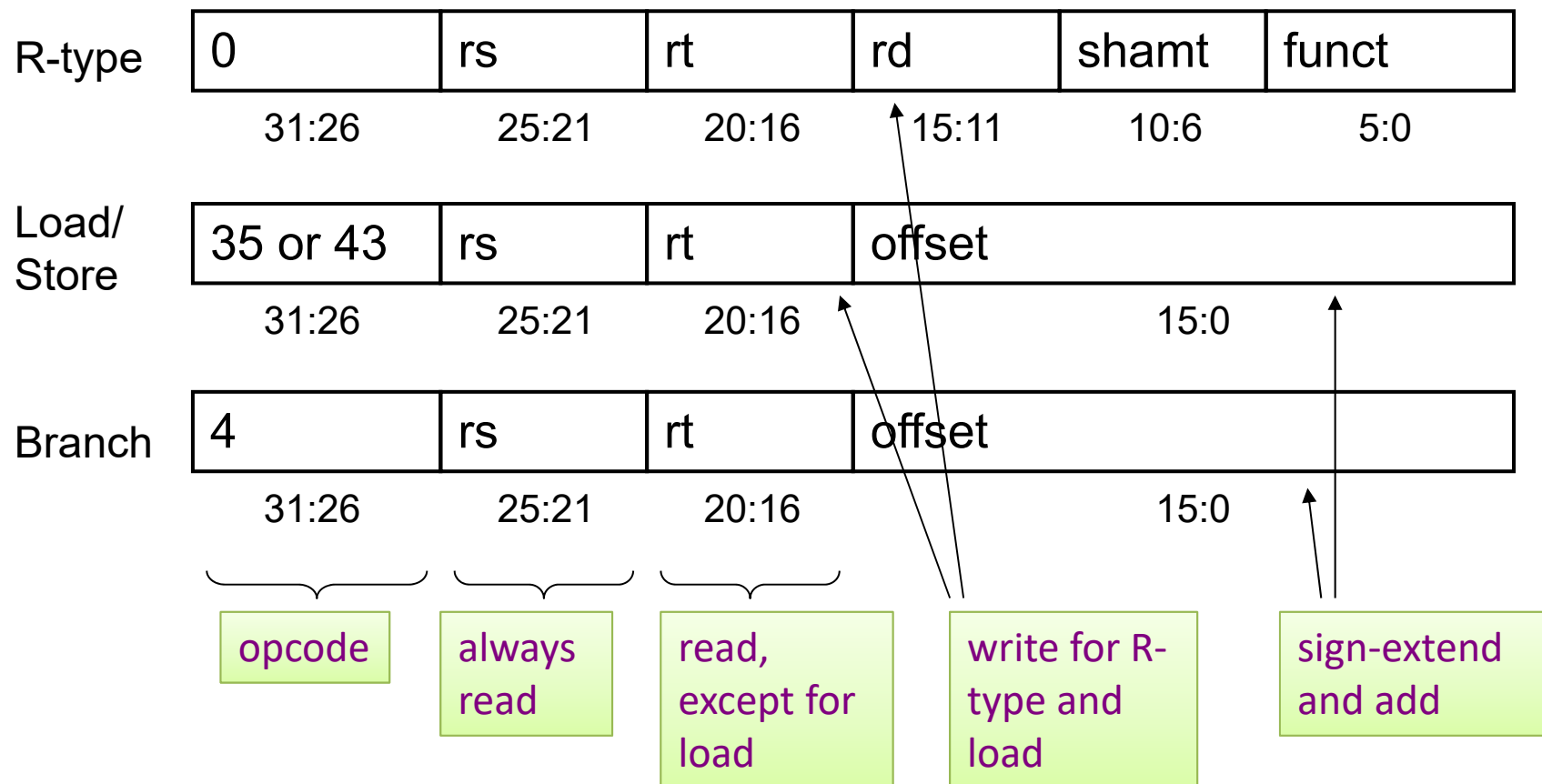
# Deciding Main Control Signals

- Control I signal



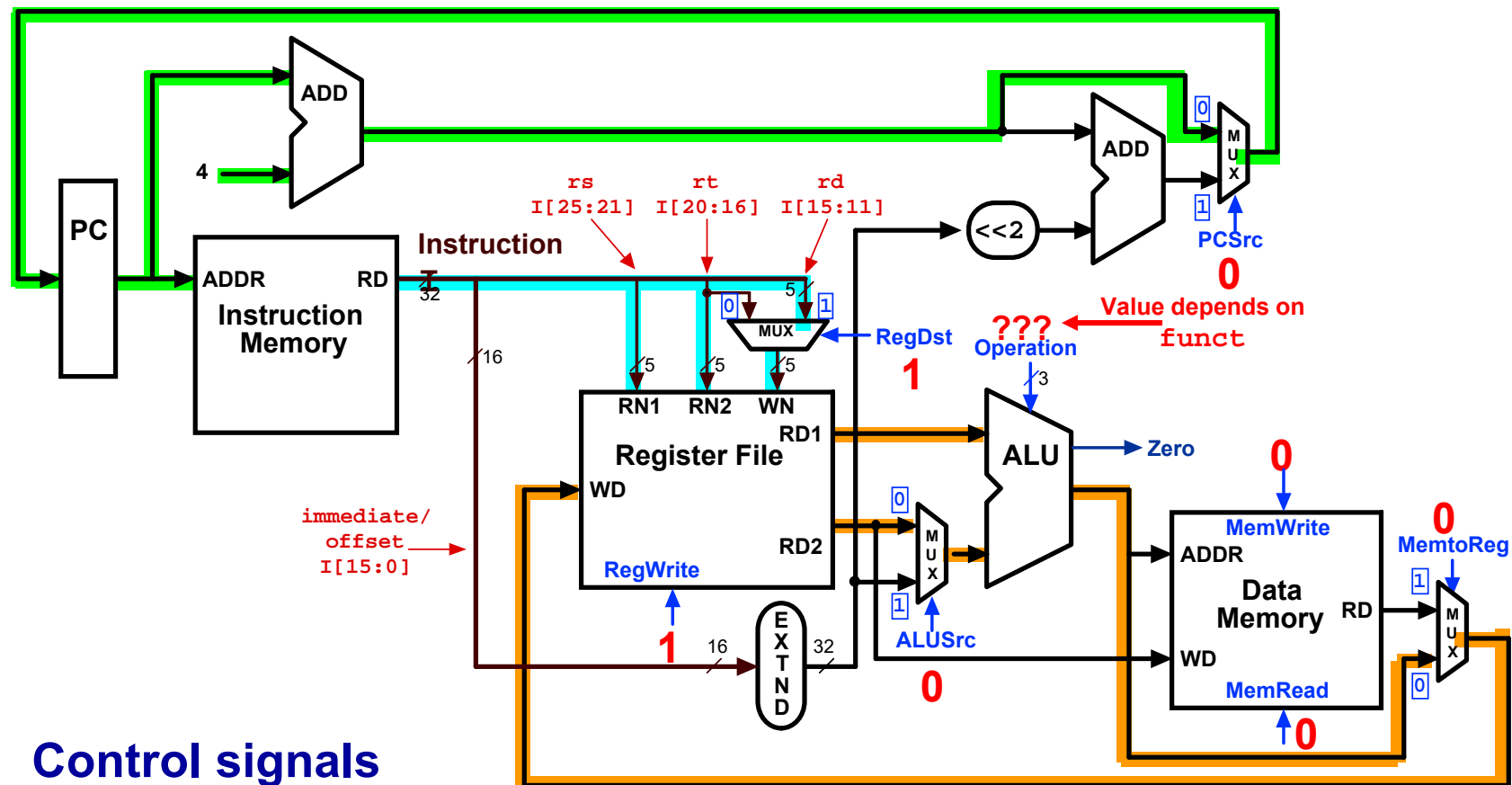
# Review: The Main Control Unit

- Control signals derived from instruction

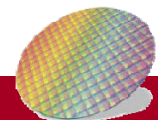




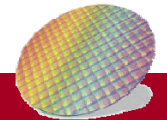
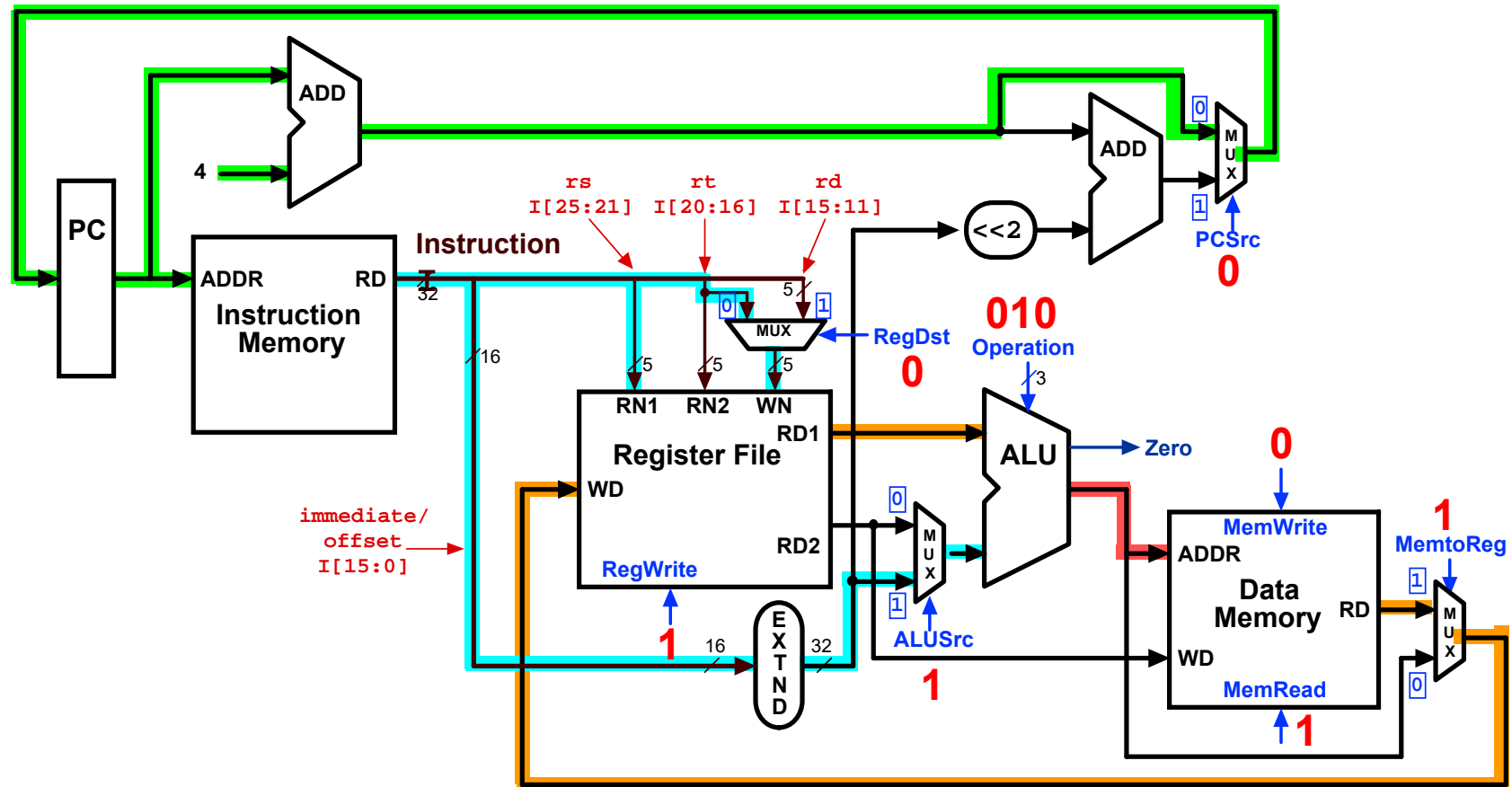
# Control Signals for R-Type Instruction



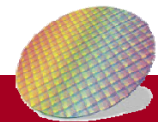
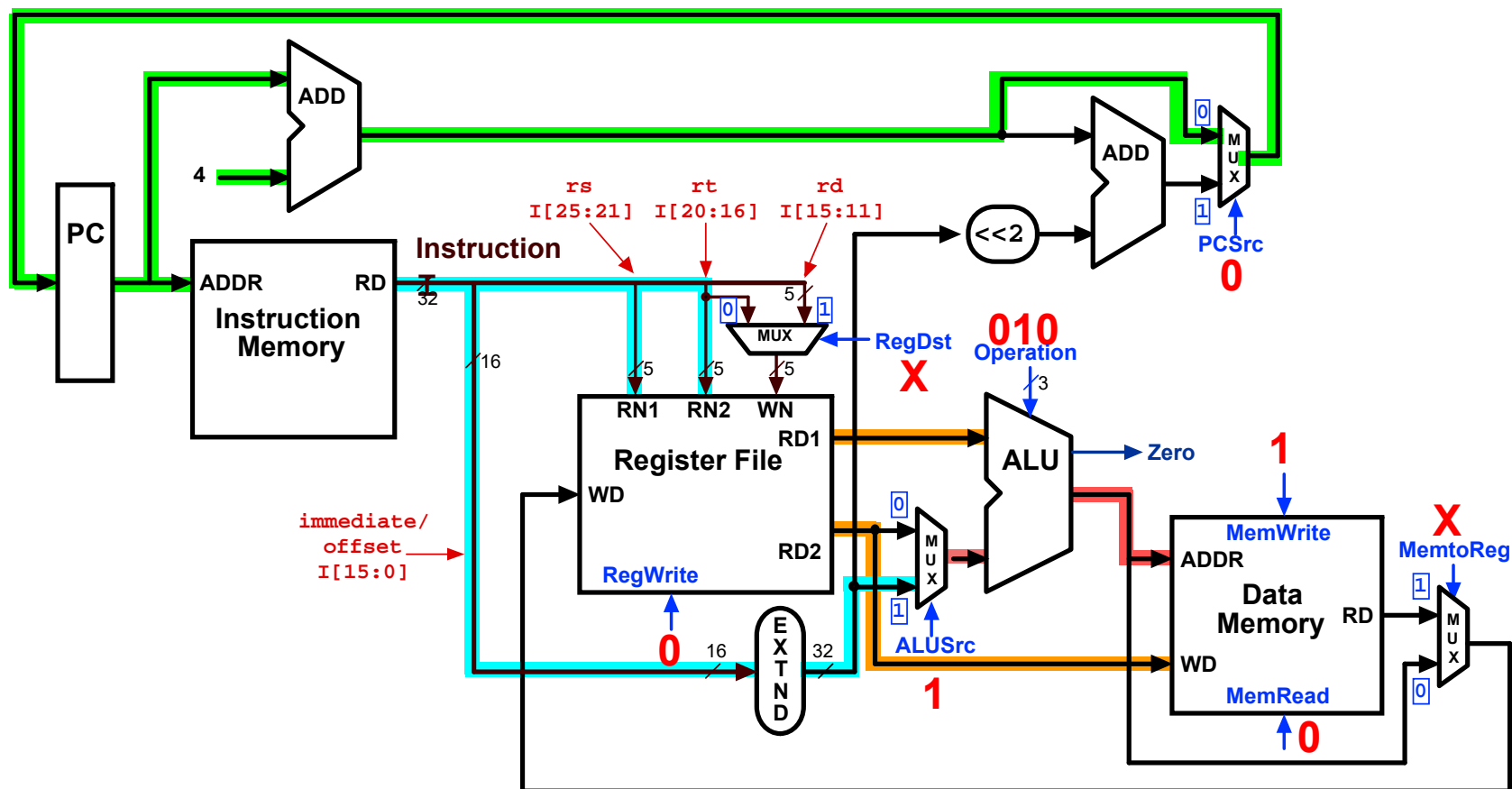
Control signals  
shown in blue



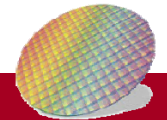
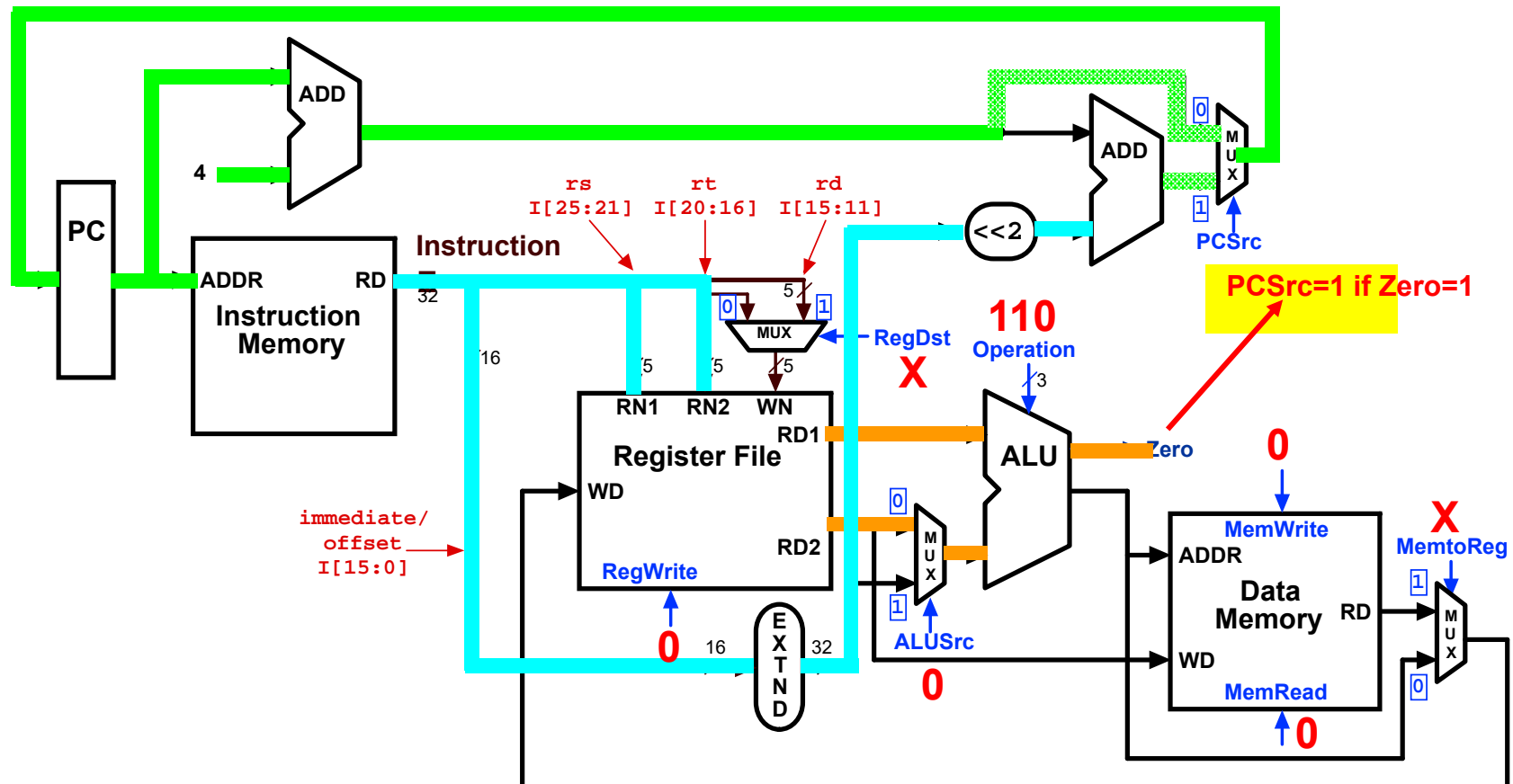
# Control Signals: lw Instruction



# Control Signals: sw Instruction

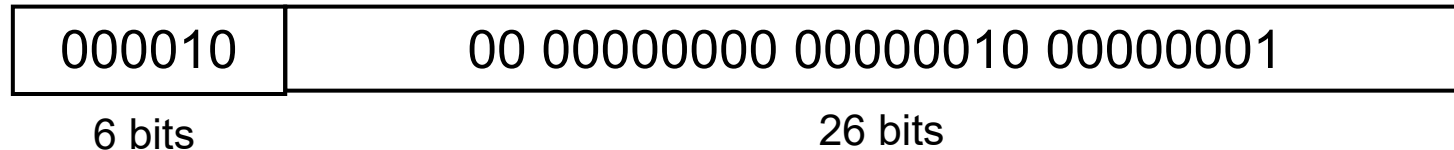


# Control Signals: beq Instruction



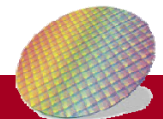
## Review: Target address of Jump

- Assume  $PC=40000000_{16}$ , what is the target address of the jump instruction?



Address in the instruction=  $0x0000201$

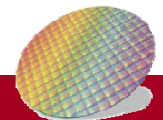
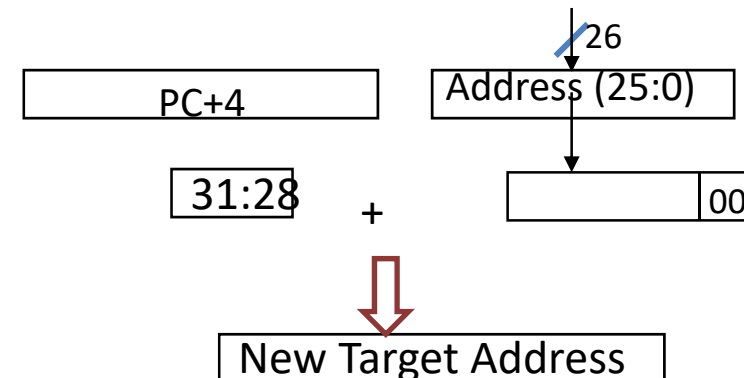
$$\text{Target Address} = PC[31:28] + 0021_{16} * 4 = 0x40000804$$



# Review: Implementing Jumps



- Jump uses **word** address
- Update **PC** with concatenation of
  - Top **4** bits of **old PC**
  - **26**-bit jump address
  - **00**
- Need an extra control signal decoded from opcode

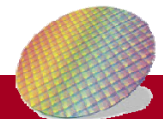
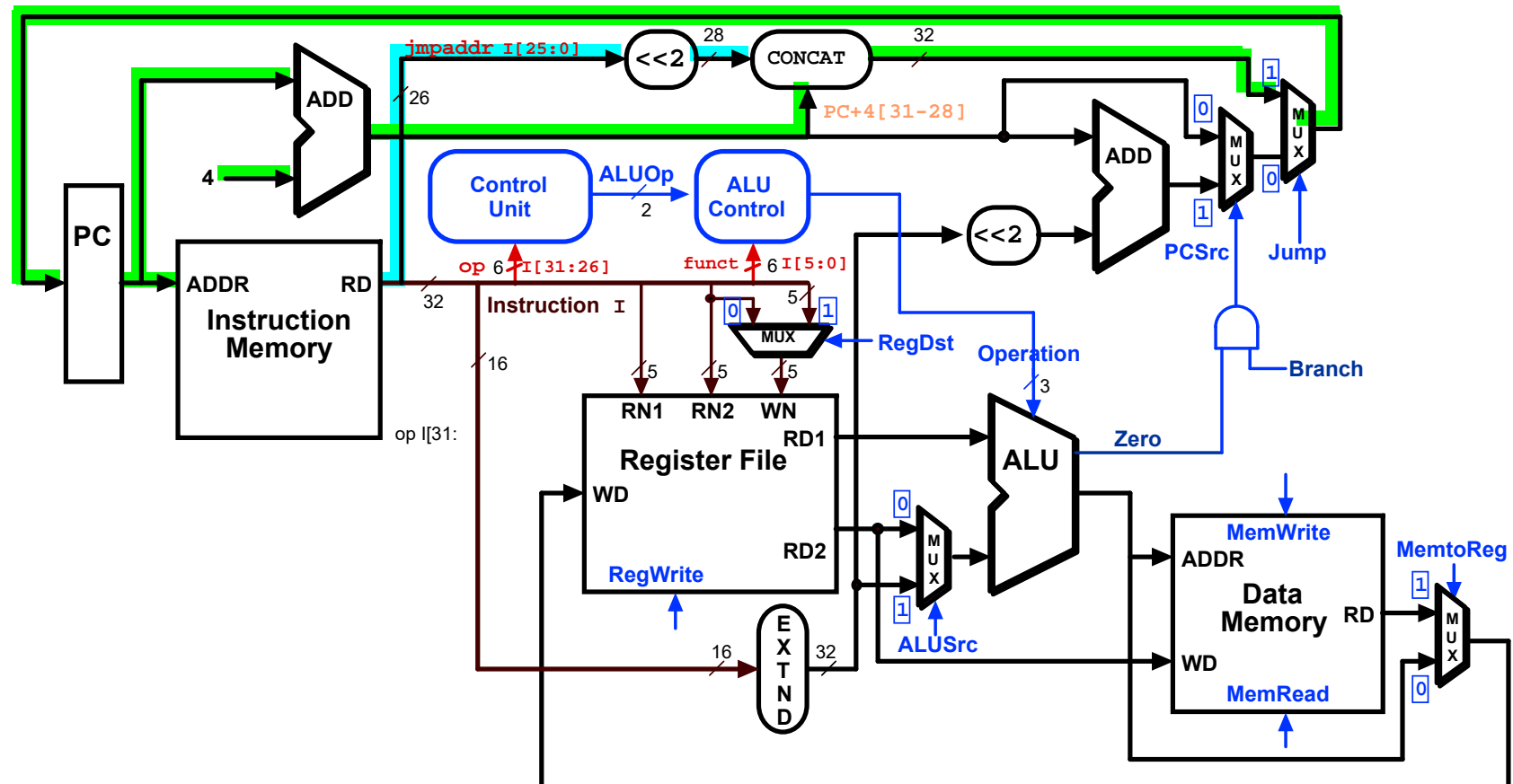






# Datapath Executing j

- j

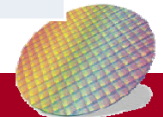


# Truth Table for Main Control Signals

- Current design of control is for
  - lw, sw, beq, and, or, add, sub, slt
- I-format: lw, sw, beq
- R-format: and, or, add, sub, slt
- Given 4 OP codes (each 6 bits) as “inputs”, the “outputs” are as follows  
=> a main control logic (the next slide)

See **appendix D**  
for details

inputs			outputs						
Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUOp0
R-format 000000	1	0	0	1	0	0	0	1	0
lw 100011	0	1	1	1	1	0	0	0	0
sw 101011	X	1	X	0	0	1	0	0	0
beq 000100	X	0	X	0	0	0	1	0	1





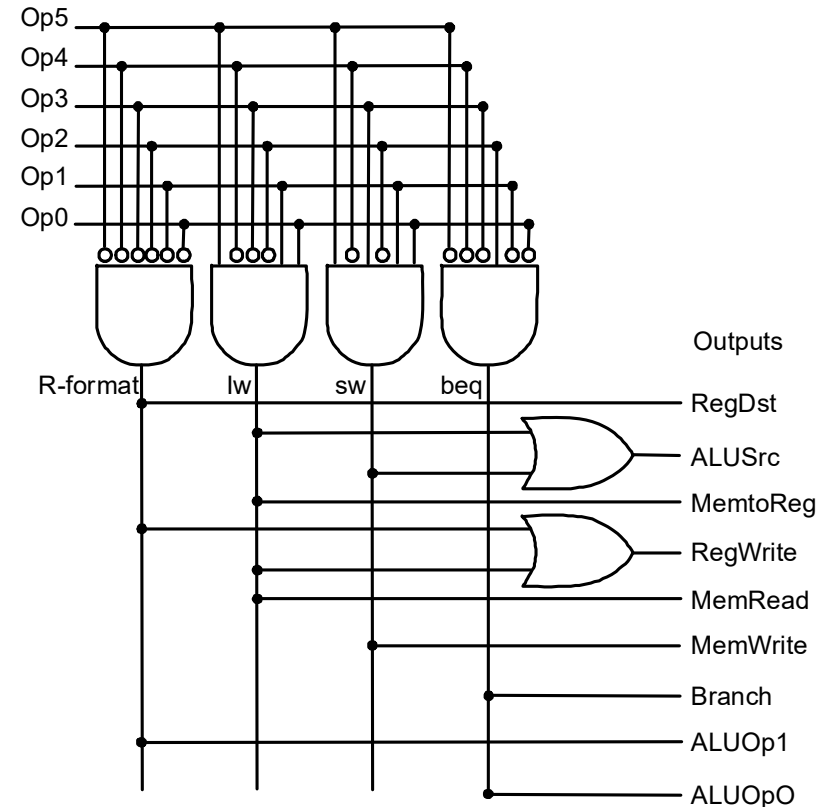
# Implementation of Main Control Block (Use PLA)

- Use PLA

	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

Truth table for main control signals

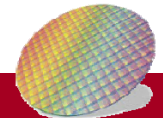
Inputs



**Main control PLA (programmable logic array)**

$$RegDst = \overline{Op5} \cdot \overline{Op4} \cdot \overline{Op3} \cdot \overline{Op2} \cdot \overline{Op1} \cdot \overline{Op0}$$

ALUSrc=?



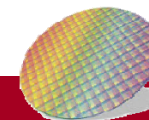
Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	AND	0000
R-type	10	OR	100101	OR	0001
R-type	10	set on less than	101010	set on less than	0111

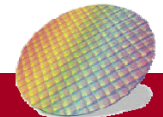
## Truth Table for ALU control signals

← inputs →      outputs

ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	$C_3C_2C_1C_0$
0	0	X	X	X	X	X	X	0010 add
0	1	X	X	X	X	X	X	0110 subtract
1	X	X	X	0	0	0	0	0010 add
1	X	X	X	0	0	1	0	0110 subtract
1	X	X	X	0	1	0	0	0000 and
1	X	X	X	0	1	0	1	0001 or
1	X	X	X	1	0	1	0	0111 slt

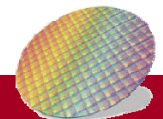
Merge LW & SW





# Performance Issues

- Longest delay determines clock period
  - Critical path: load instruction
    - Instruction memory → register file → ALU → data memory → register file
- Not feasible to vary period for different instructions
- Violates design principle
  - Making the common case fast
- We will improve performance by pipelining





成功大學

National Cheng Kung University

## Backup Slides

