Chapter 4

The Processor

Introduction

- CPU performance factors
  - Instruction count
    - Determined by ISA and compiler
  - CPI and Cycle time
    - Determined by CPU hardware

CPU Time=Instruction Count ×CPI ×Clock Cycle Time

- 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 inst. subset, but shows most aspects
  - Memory reference: `lw`, `sw`
  - Arithmetic/logical: `add`, `sub`, `and`, `or`, `slt`
  - Control transfer: `beq`, `j`

Review: MIPS Instruction Set Architecture (ISA)

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

Registers

Registers
R0 - R31
PC
HI
LO

3 Instruction Formats: all 32 bits wide

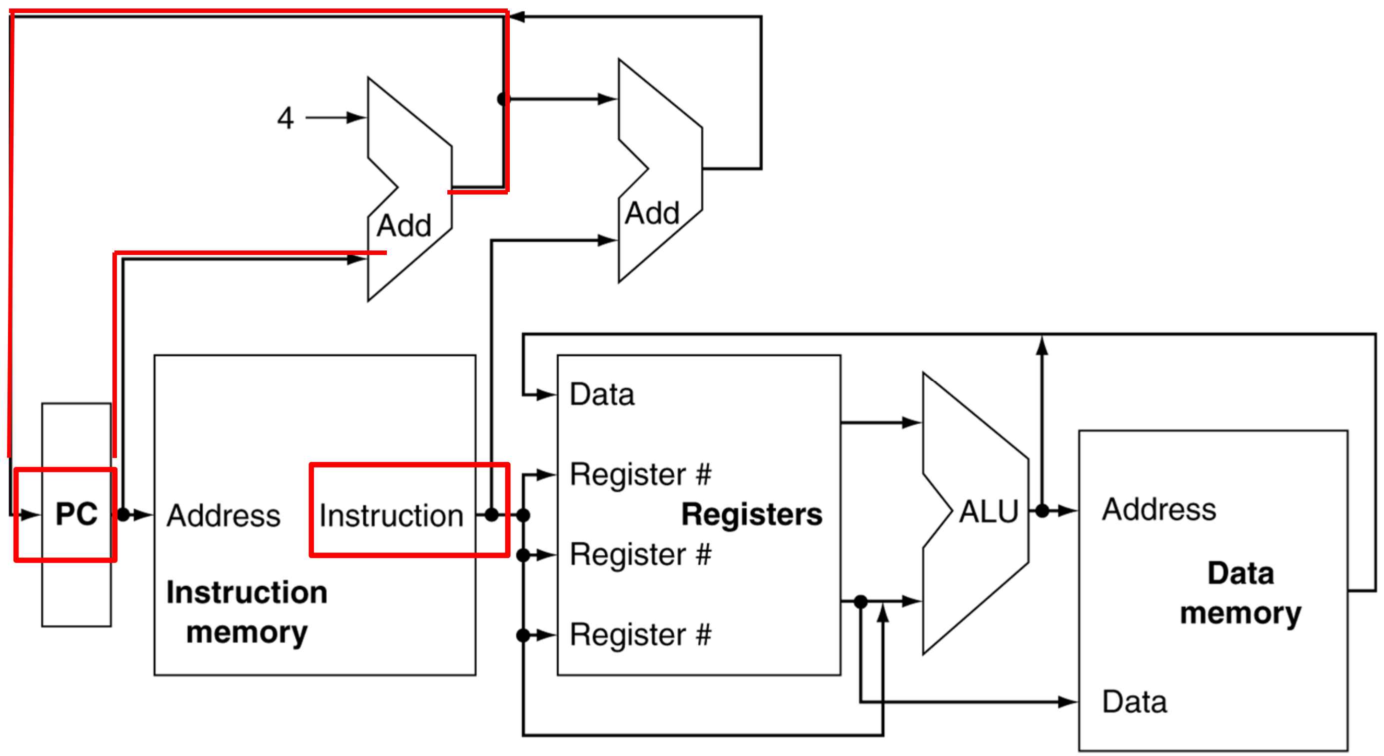
OP	**rs**	**rt**	**rd**	**sa**	**funct**	R format
OP	**rs**	**rt**	**immediate**			I format
OP	**jump target**					J format

Review: MIPS Register Convention

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

Instruction Execution

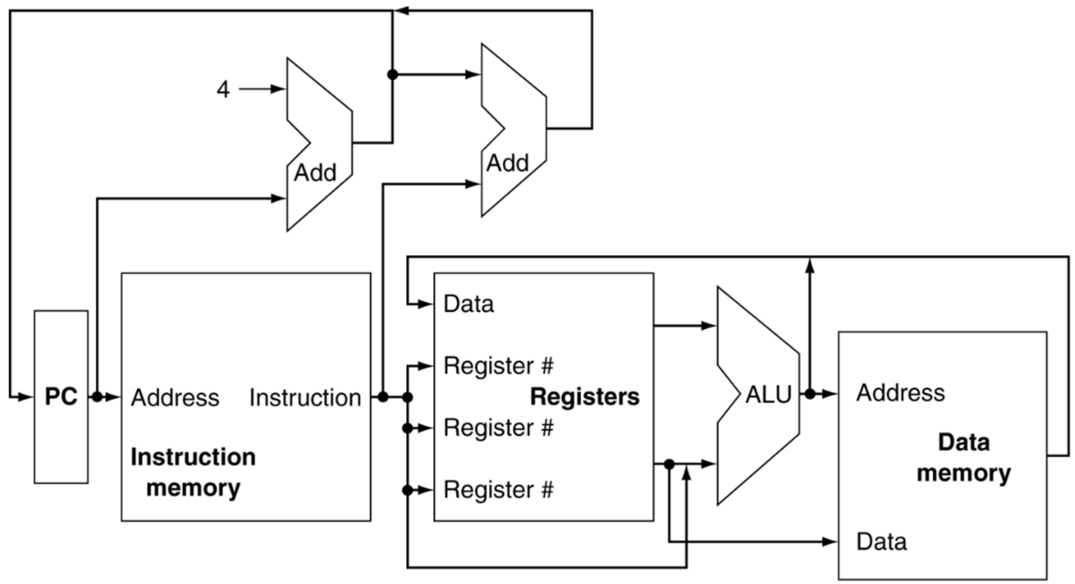
- PC (Program counter) is used to fetch instruction in the instruction memory)
- After instruction is obtained, register numbers in instructions is used to read registers in register files.
- PC ← PC +4 for sequentially execution



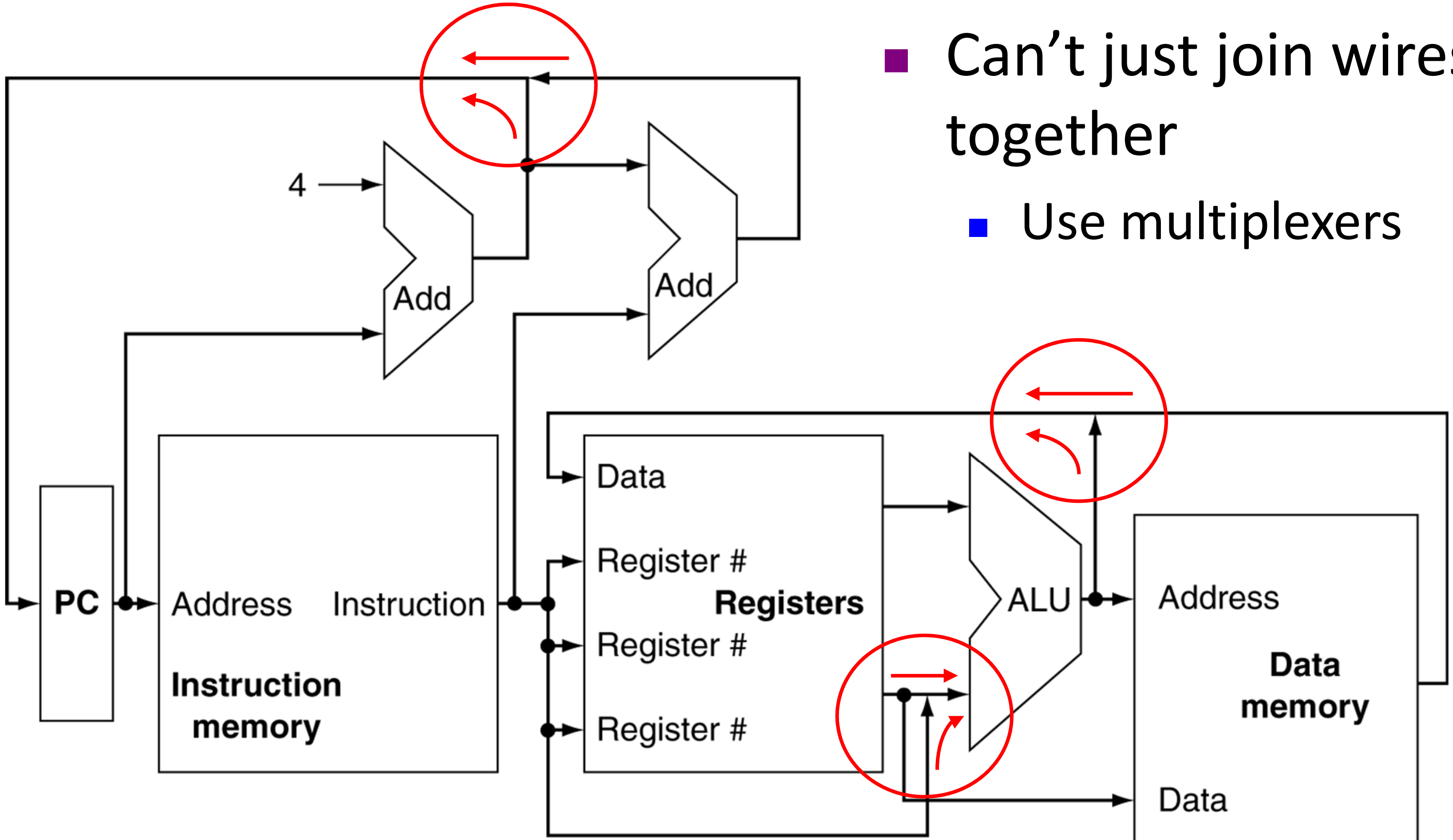
Different instructions have different actions

- Use ALU to calculate
  - Arithmetic result
  - Memory address for load/store
  - Branch target address
- Access data memory for load/store
- PC ← target address

```
add $t0, $s1, $s2
lw $s1, 20($s2)
bne  $t0, $s5, Exit
j     Loop
```

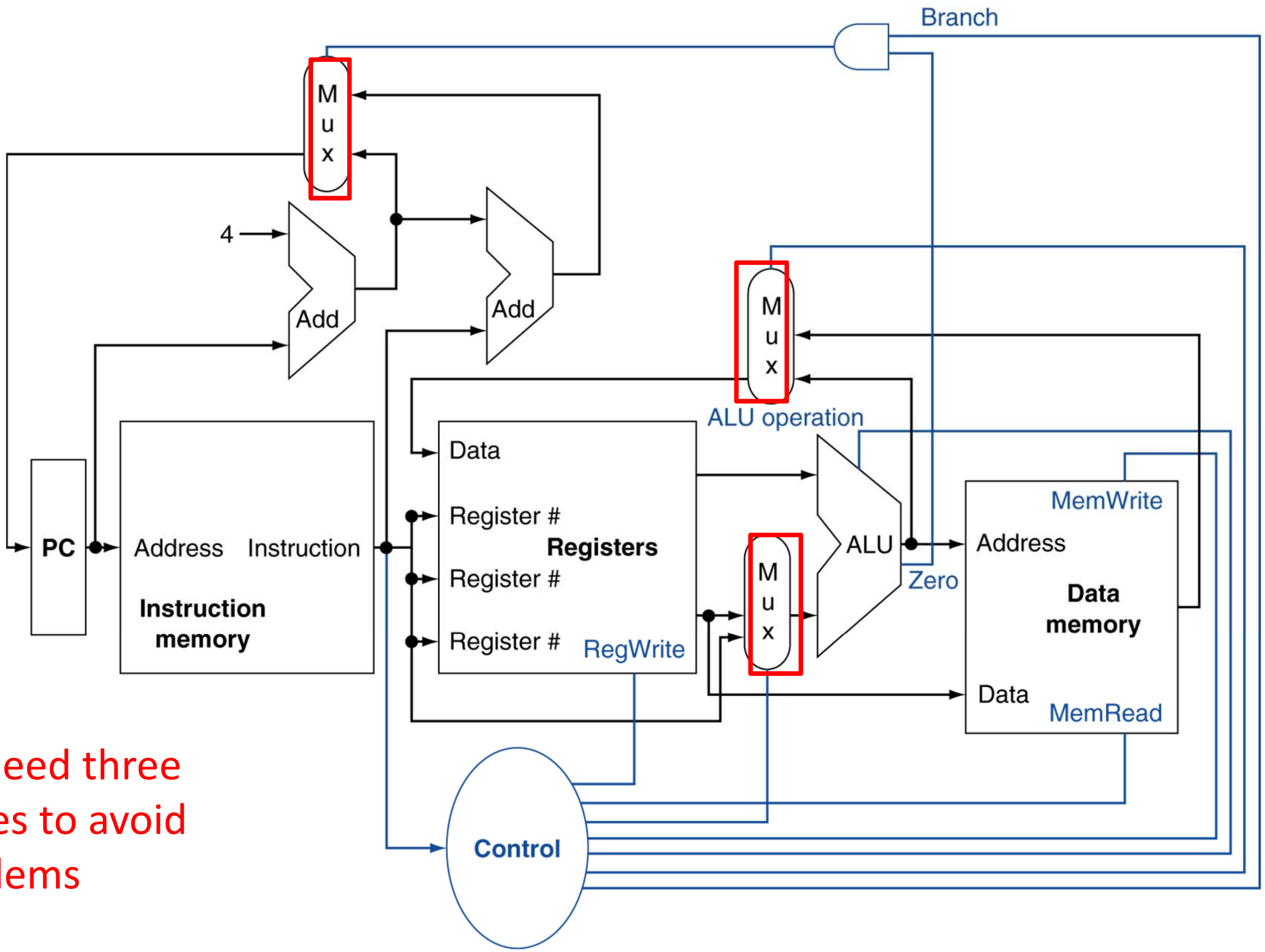


Need Multiplexers to fix problems



- Can't just join wires together
  - Use multiplexers

Modified CPU– An overview

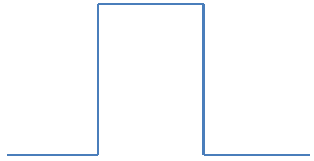


We need three Muxes to avoid problems

Details of each Mux and Control will be introduced later

Logic Design Basics

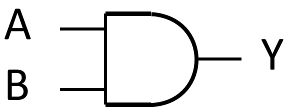
- Information encoded in binary
  - Low voltage = 0, High voltage = 1
  - One wire per bit
  - Multi-bit data encoded on multi-wire buses



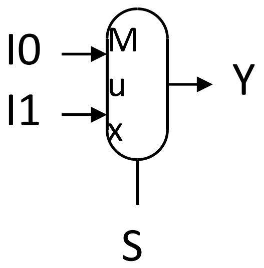
- Combinational element (See next slide)
  - Operate on data
  - Output is a function of input
- State (sequential) elements
  - Output is a function of input and current states
  - Store information

Review: Combinational Elements

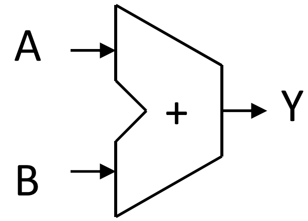
- AND-gate
  - Y = A & B



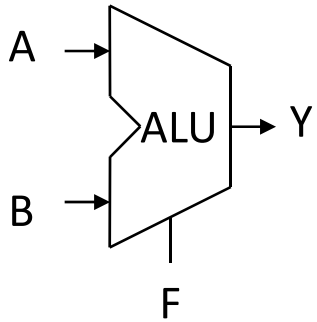
- Multiplexer
  - Y = S ? I1 : I0



- Adder
  - Y = A + B

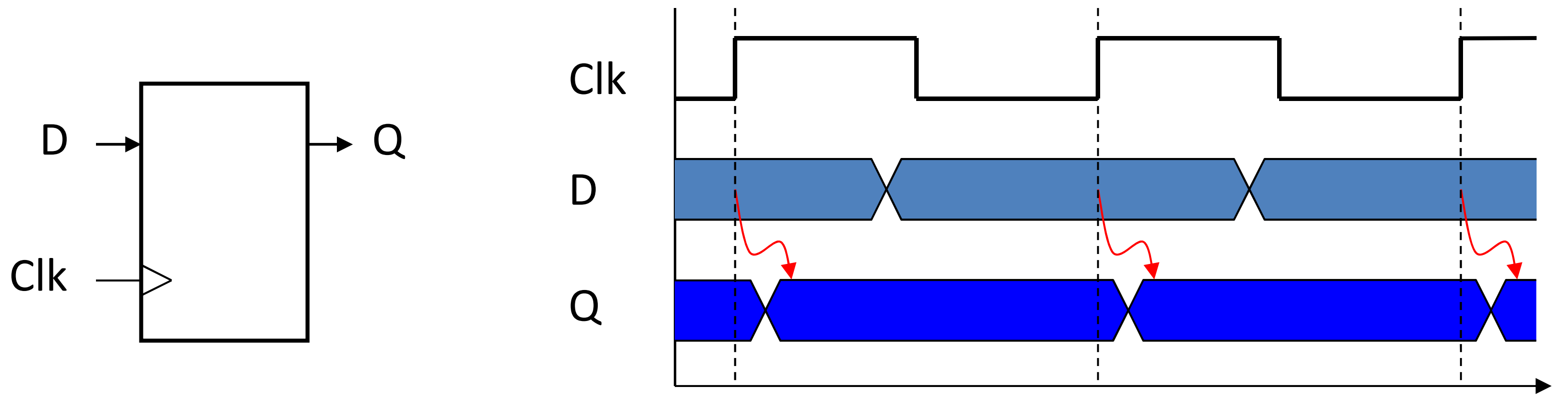


- Arithmetic/Logic Unit
  - Y = F(A, B)



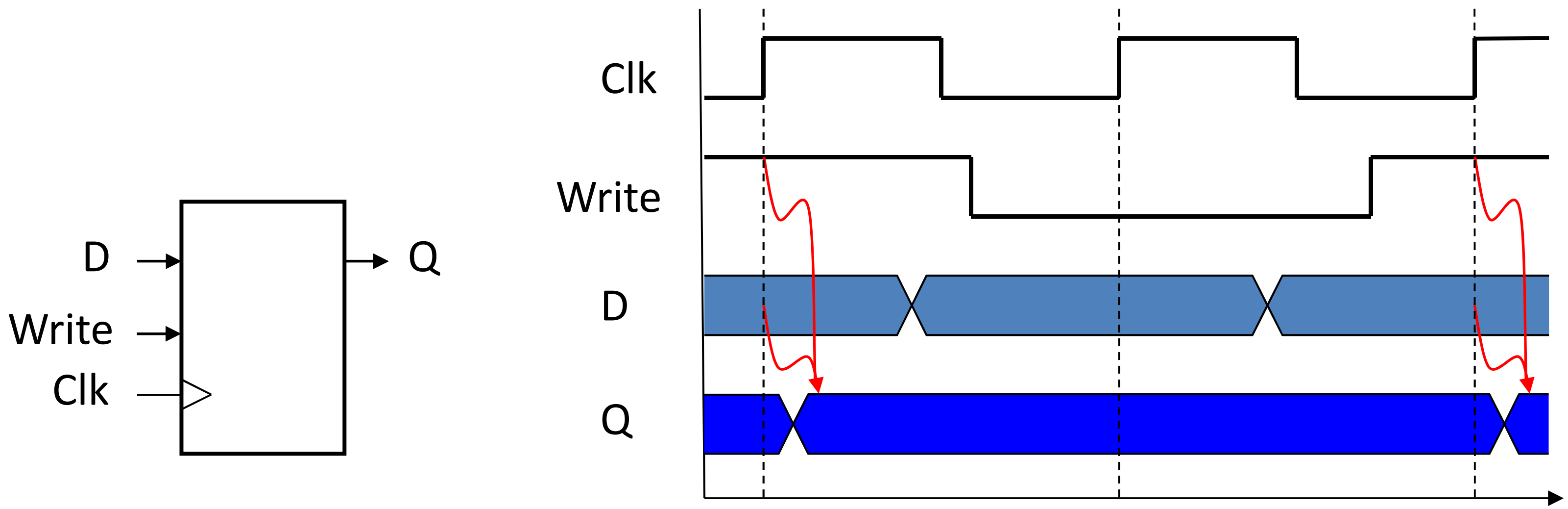
Review: Sequential Elements

- 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

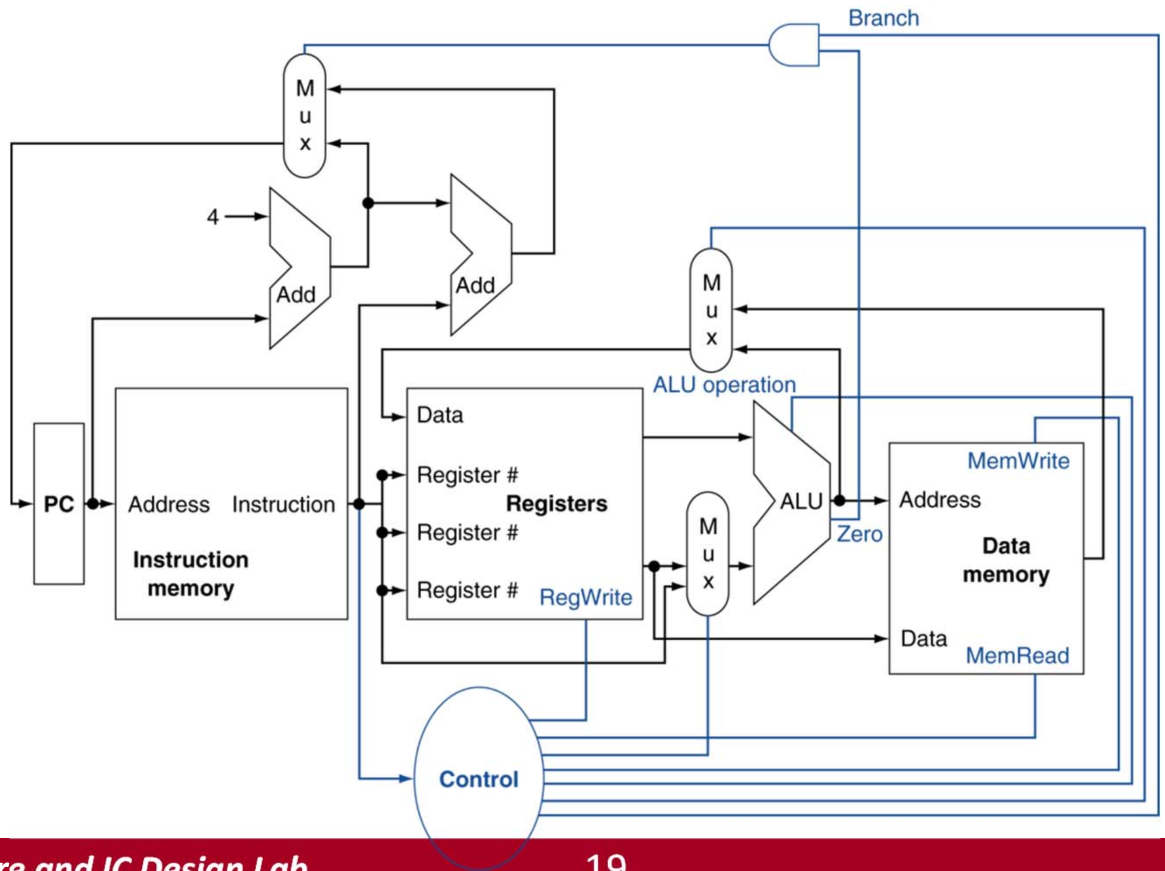


Q is not changed when Write=0

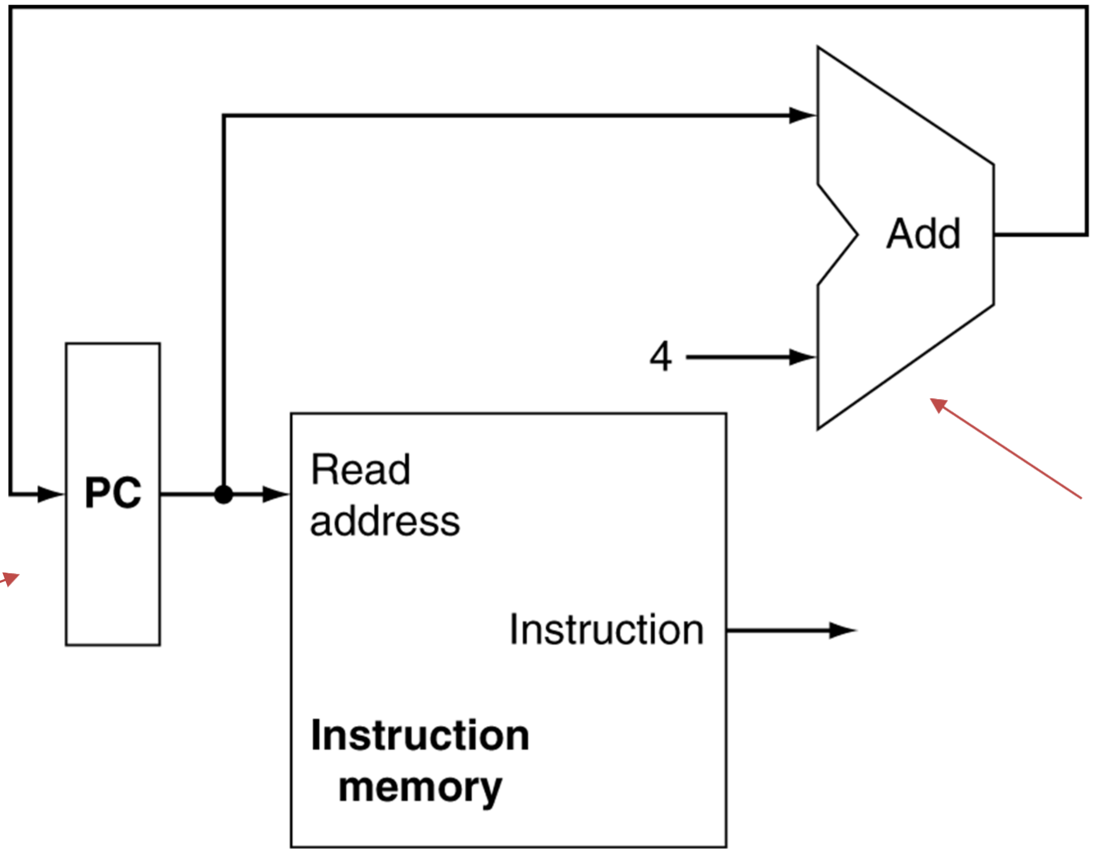
Building a Datapath

- 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

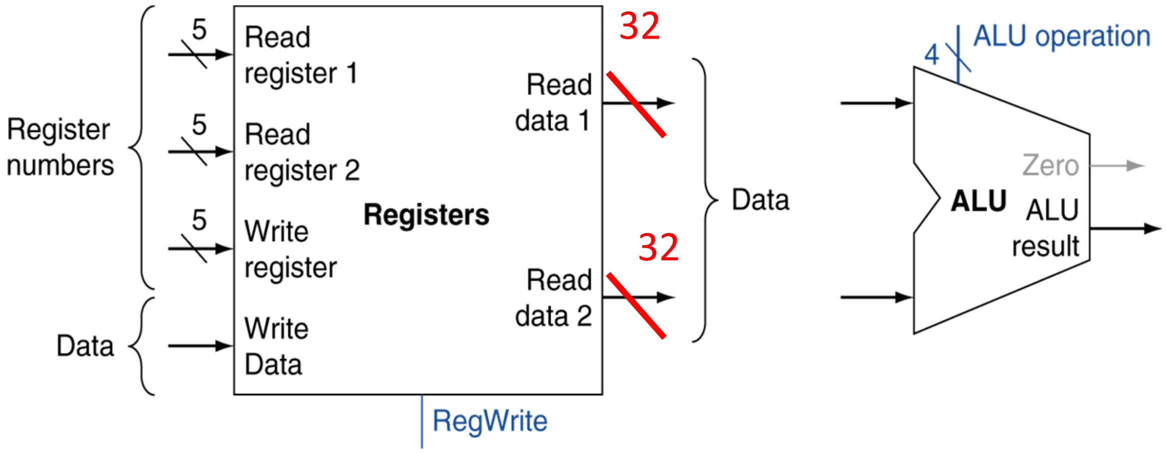


32-bit register

Increment by 4 for next instruction

R-Format Instructions

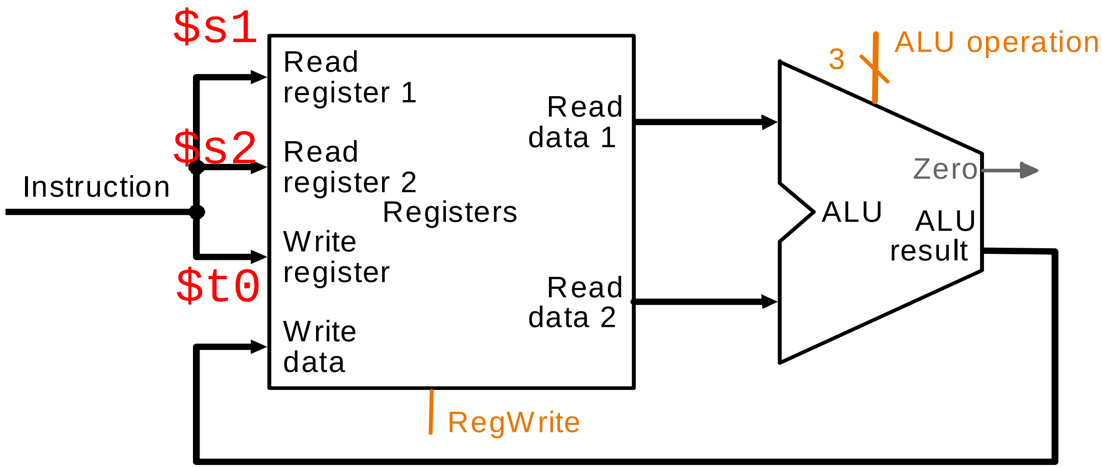
- Read two register operands
- Perform arithmetic/logical operation
- Write results into destination registers



a. Registers

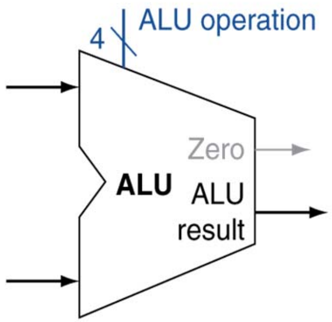
b. ALU

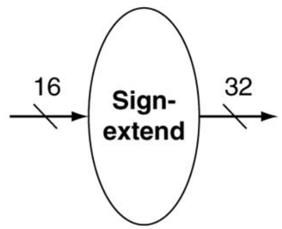
```
add   $t0, $s1, $s2
```

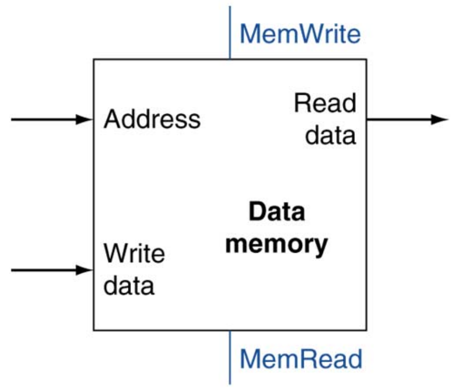


Load/Store Instructions (need 4 components)

- 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



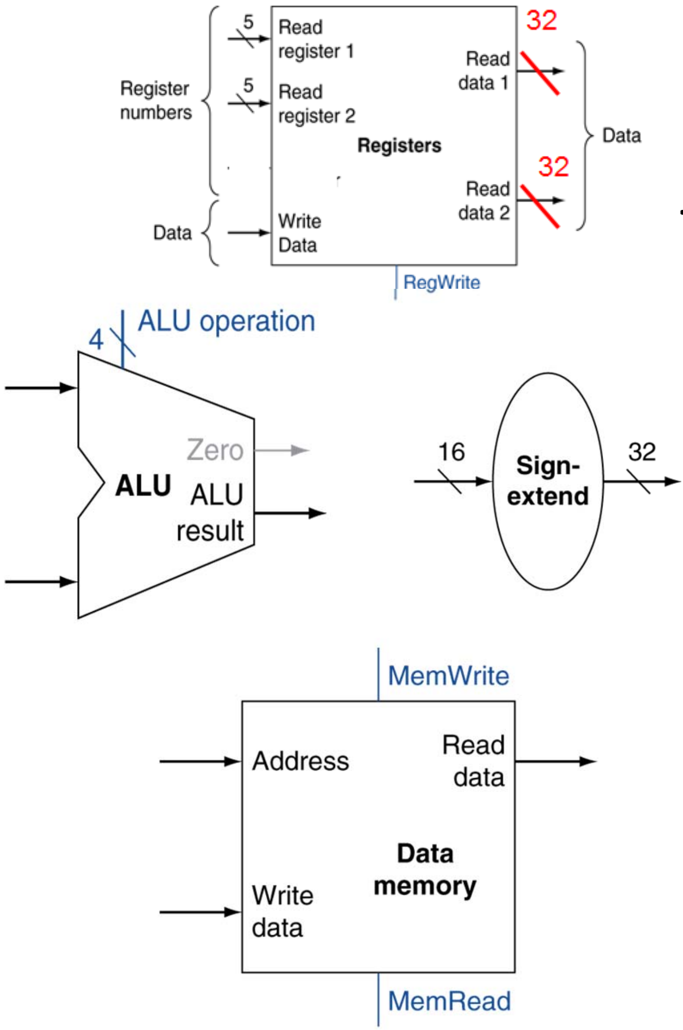


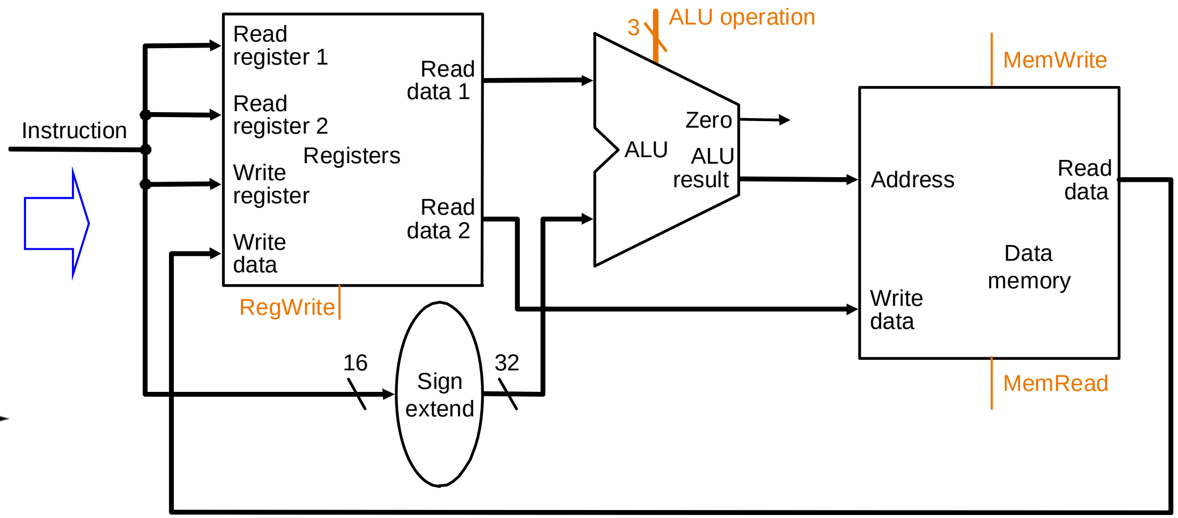


lw $t0, 4($s3) #load word from memory
sw $t0, 8($s3) #store word to memory

Datapath: Load/Store Instruction

- Load/store





Datapath

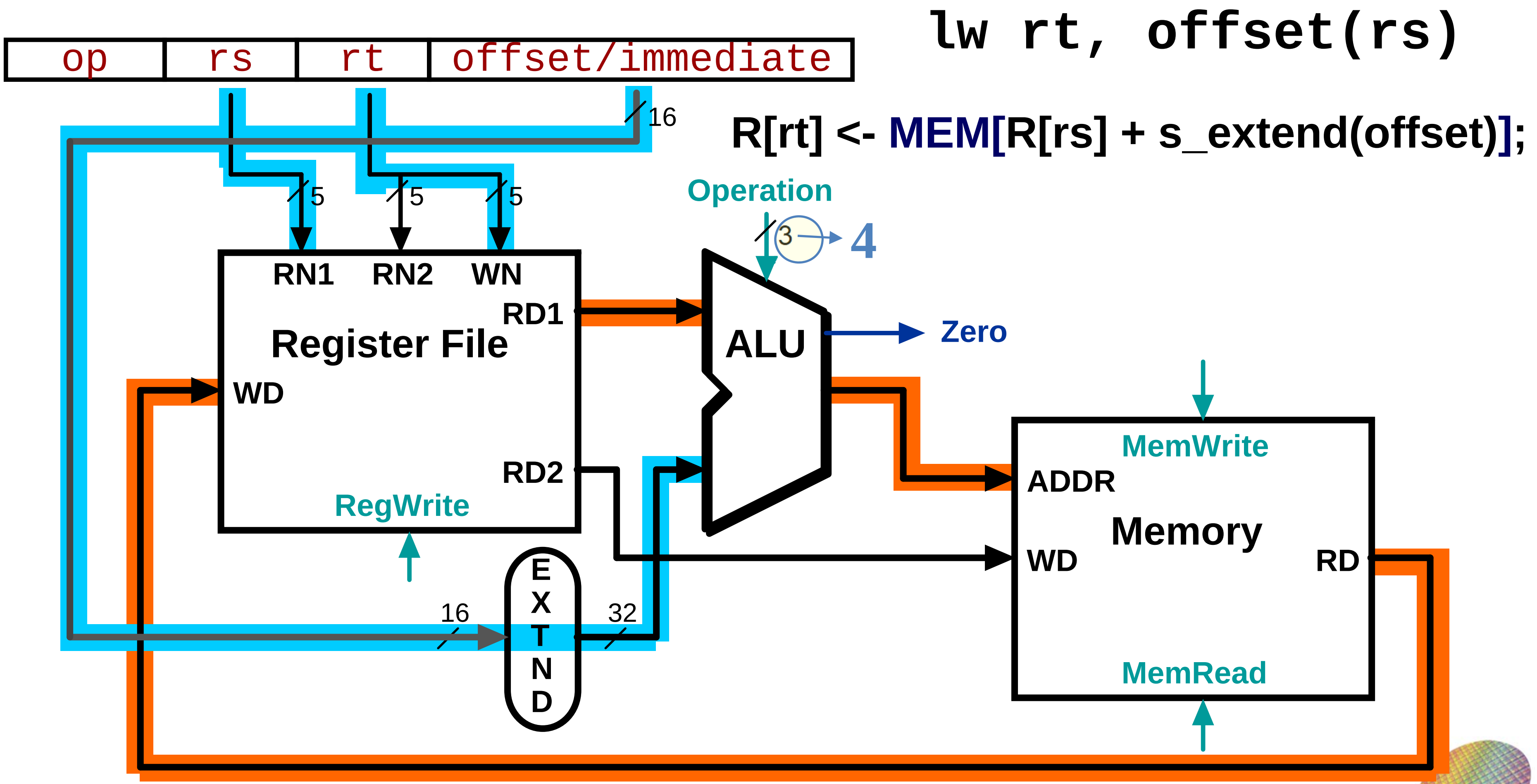
Two additional elements (sign extension and data memory) used To implement load/stores

Animating the Datapath- load

- Load

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

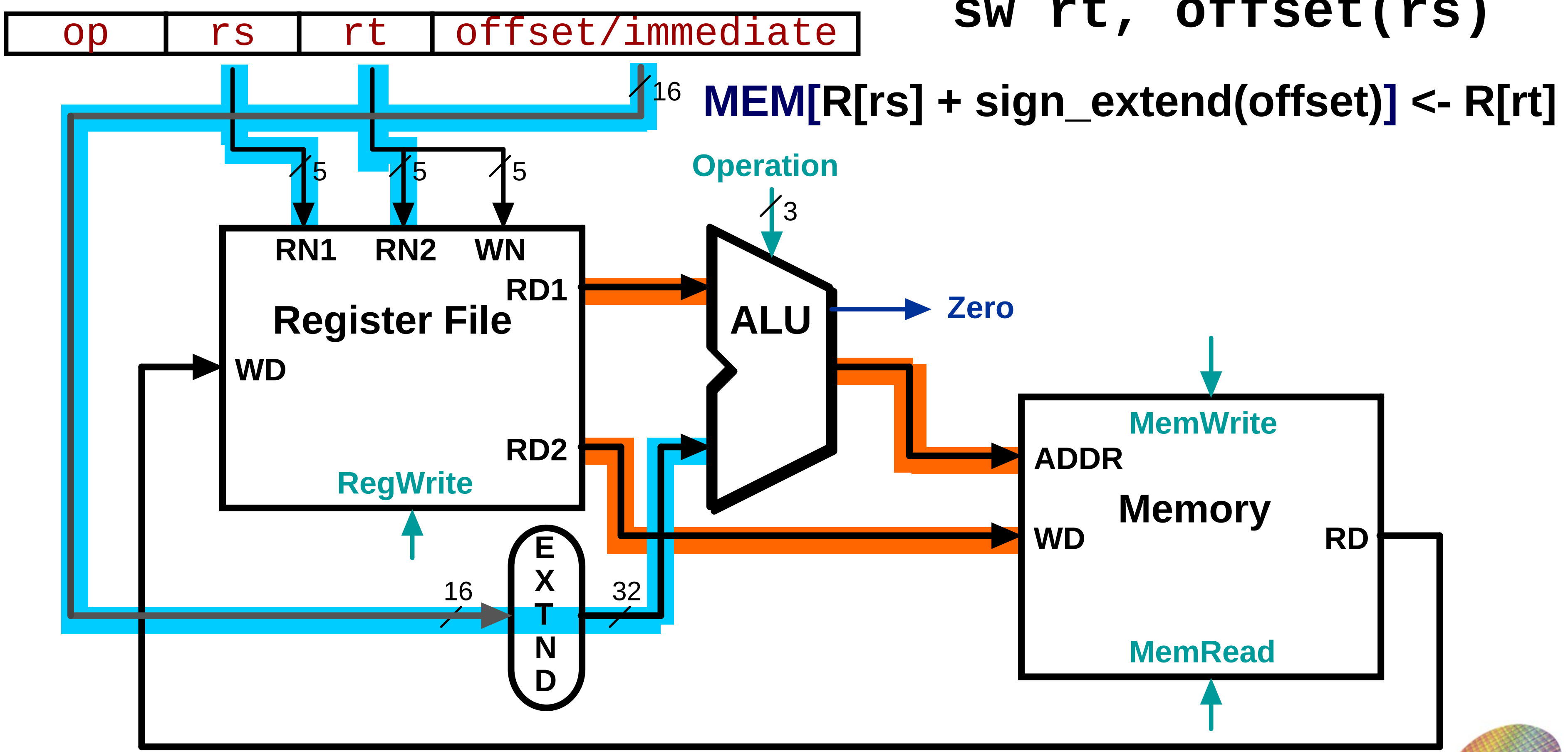
- RN1: register number 1
- RN2: register number 2
- WN: register number that will be written
- WD: write data



Animating the Datapath- store

- store

  sw $t0, 8($s3)



Review: Specifying Branch Destinations

- MIPS conditional branch instructions:

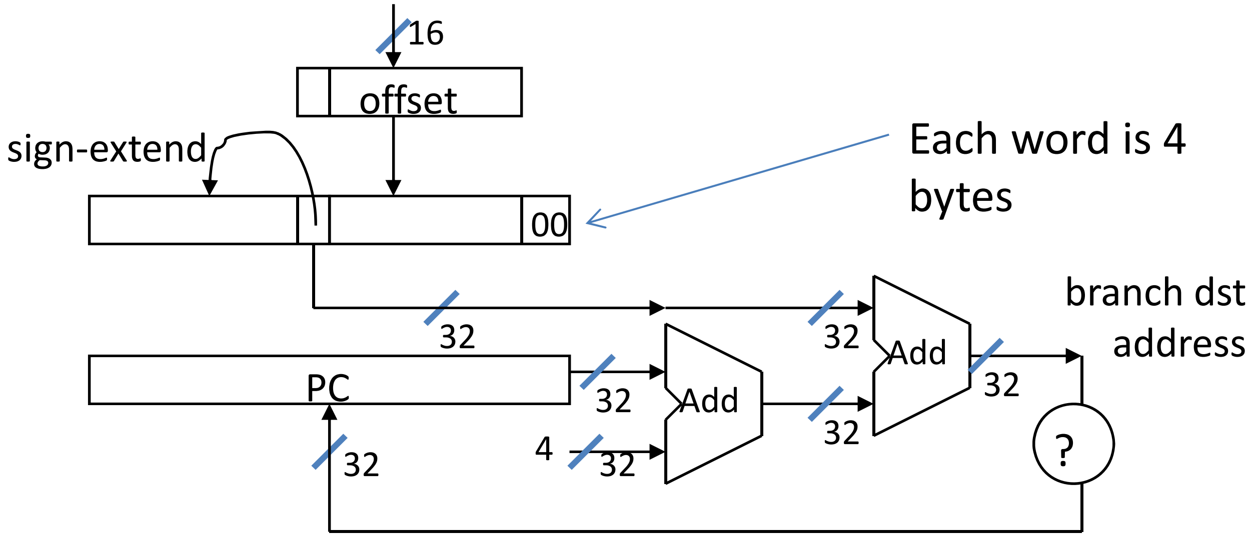
op	rs	rt	offset
6 bits	5 bits	5 bits	16 bits

2000 beq \$s0 \$t1 2
2004 ....
2008 ...
200C

- PC-relative addressing
  - Target address = PC + offset $\times$ 4
  - PC already incremented by 4 by this time

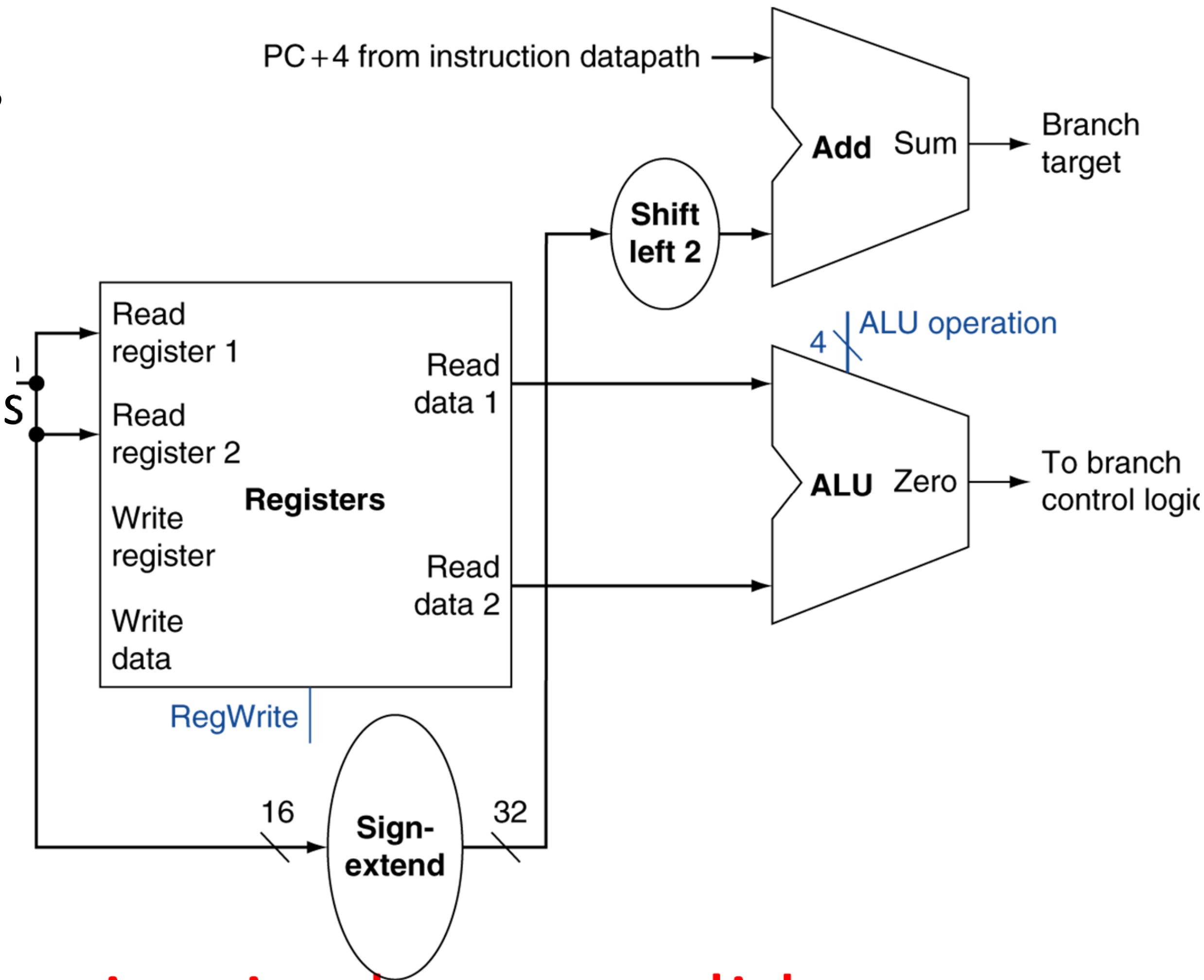
Target Address (address of next instruction) =?
200C

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)

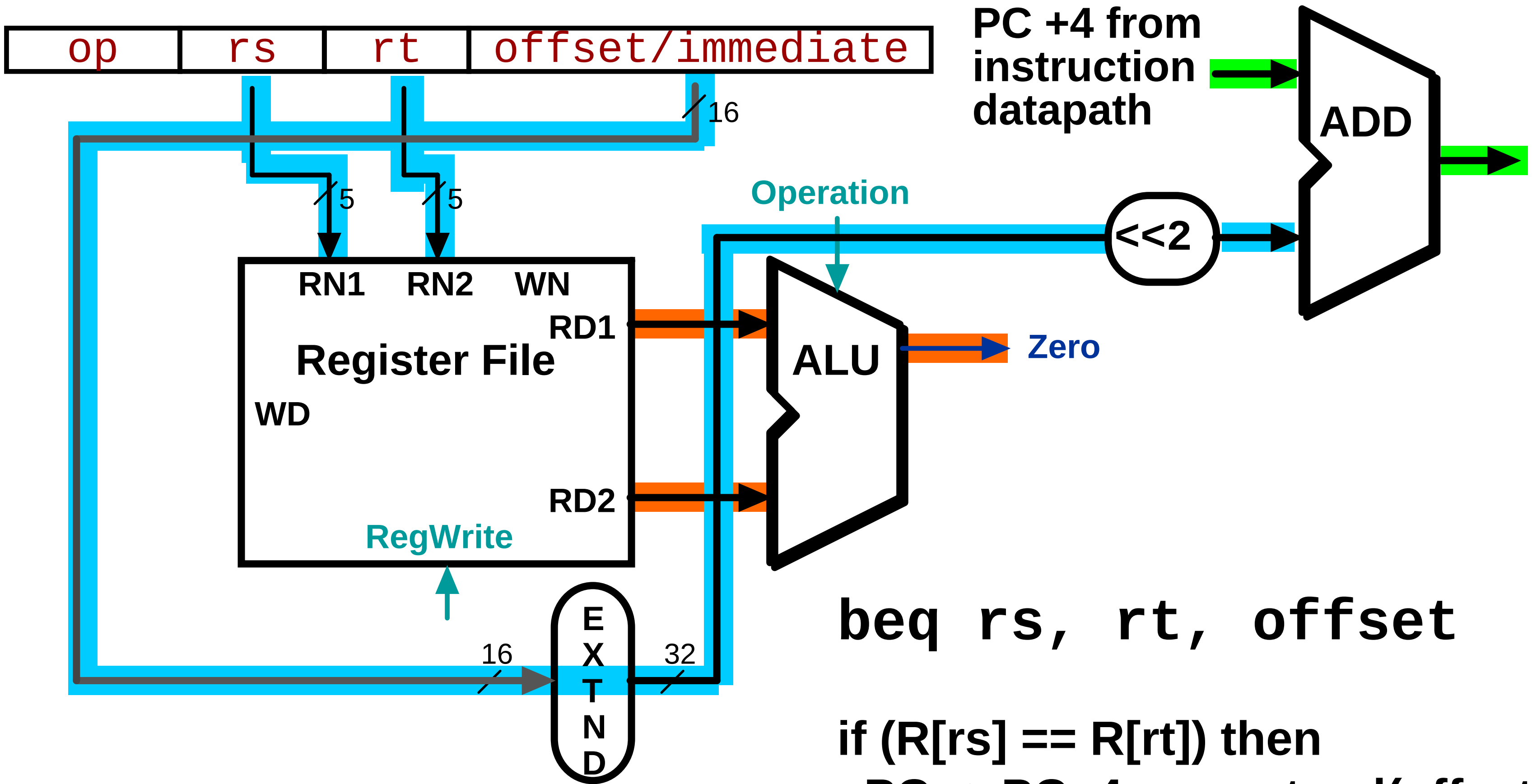


See animation in the next slide

Animating the Datapath (beq)

- Beq rs, rt, offset

e.g. beq $s0 $t1 2

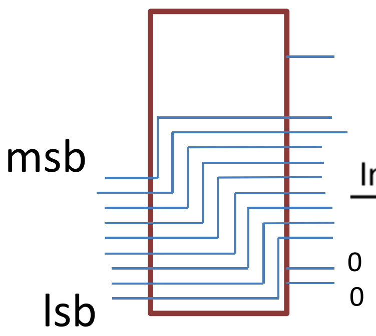


beq rs, rt, offset

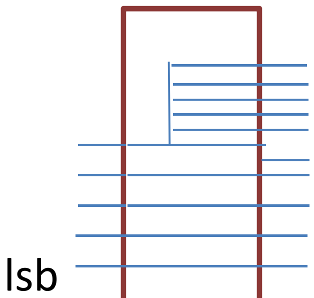
if (R[rs] == R[rt]) then
PC <- PC+4 + s_extend(offset<<2)

Sign-extension and shift left by 2 hardware

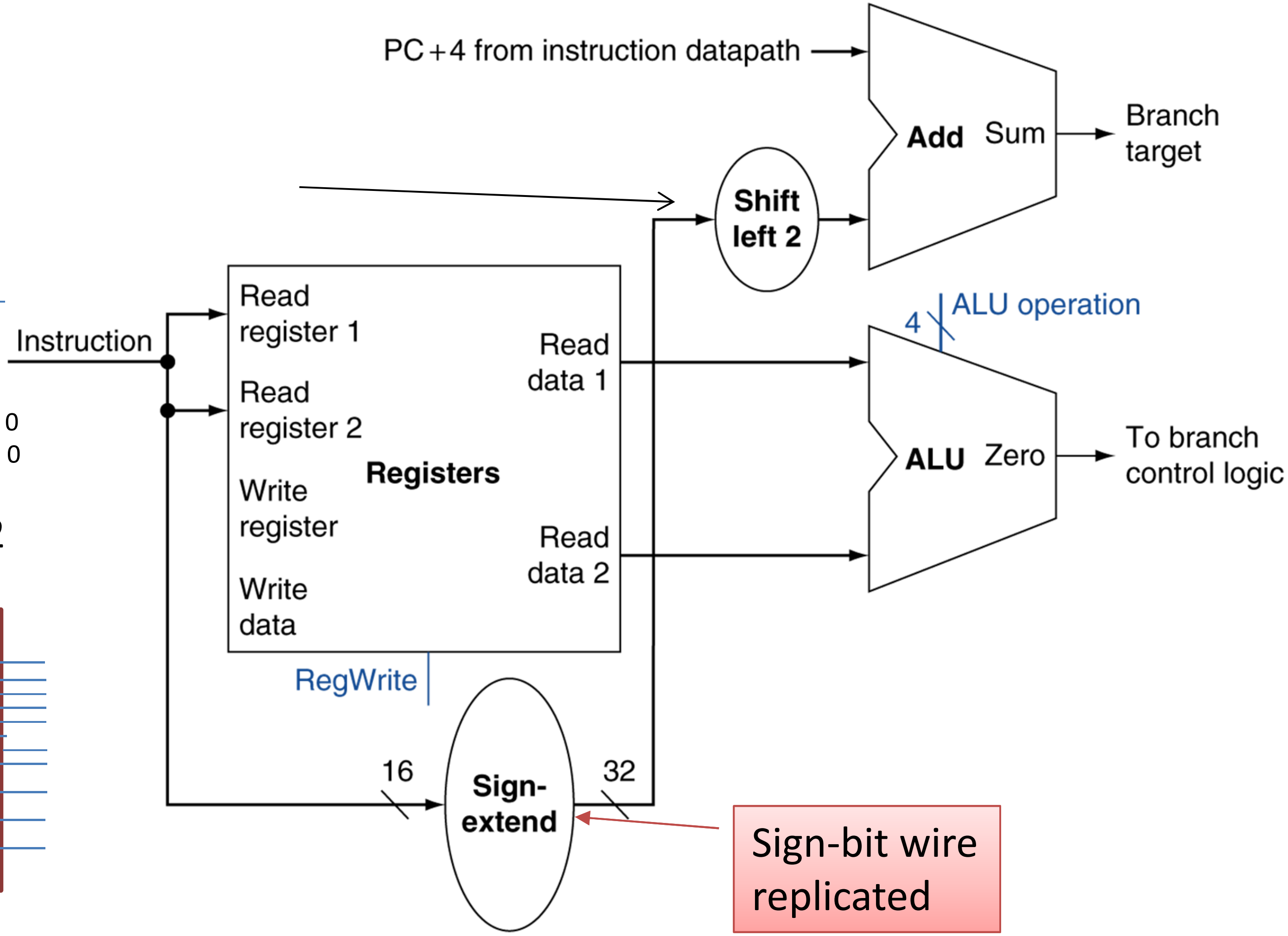
- Simple hardware is used for sign extension and shift left by 2



Shift left by 2



Signed extension hardware

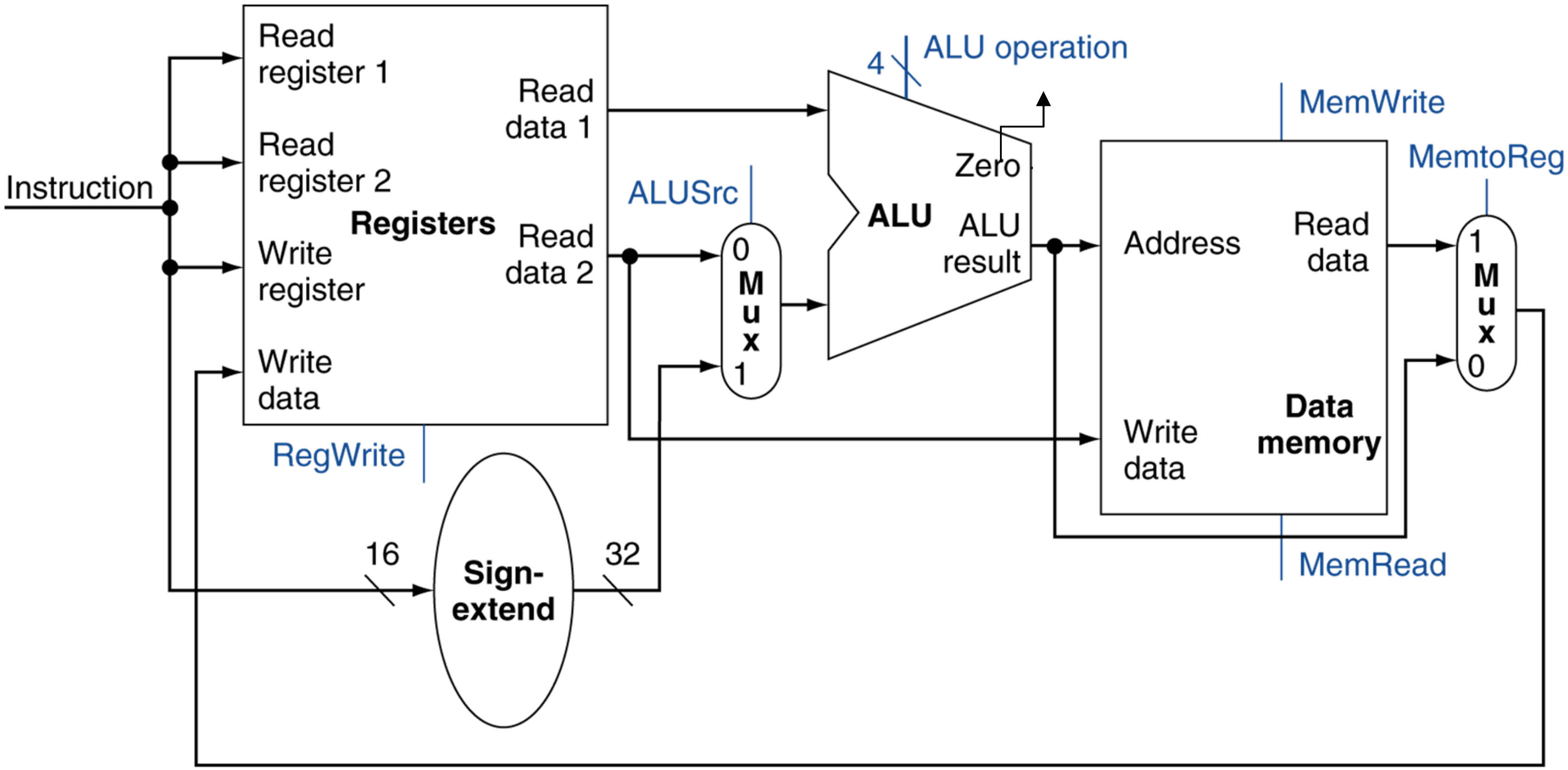


Composing the Elements

- Make datapath 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

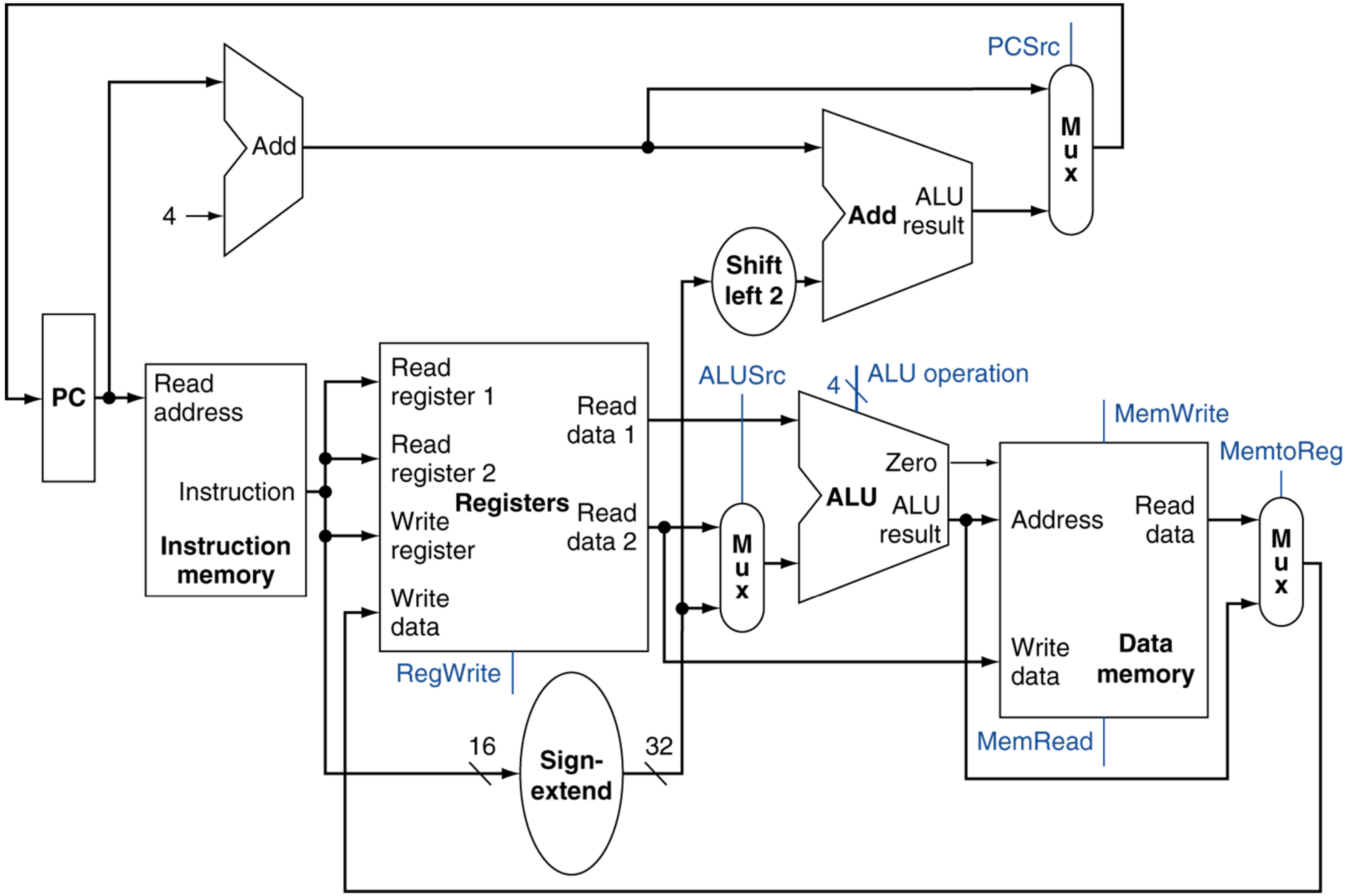
R-Type/Load/Store Datapath

A Single Cycle Datapath



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

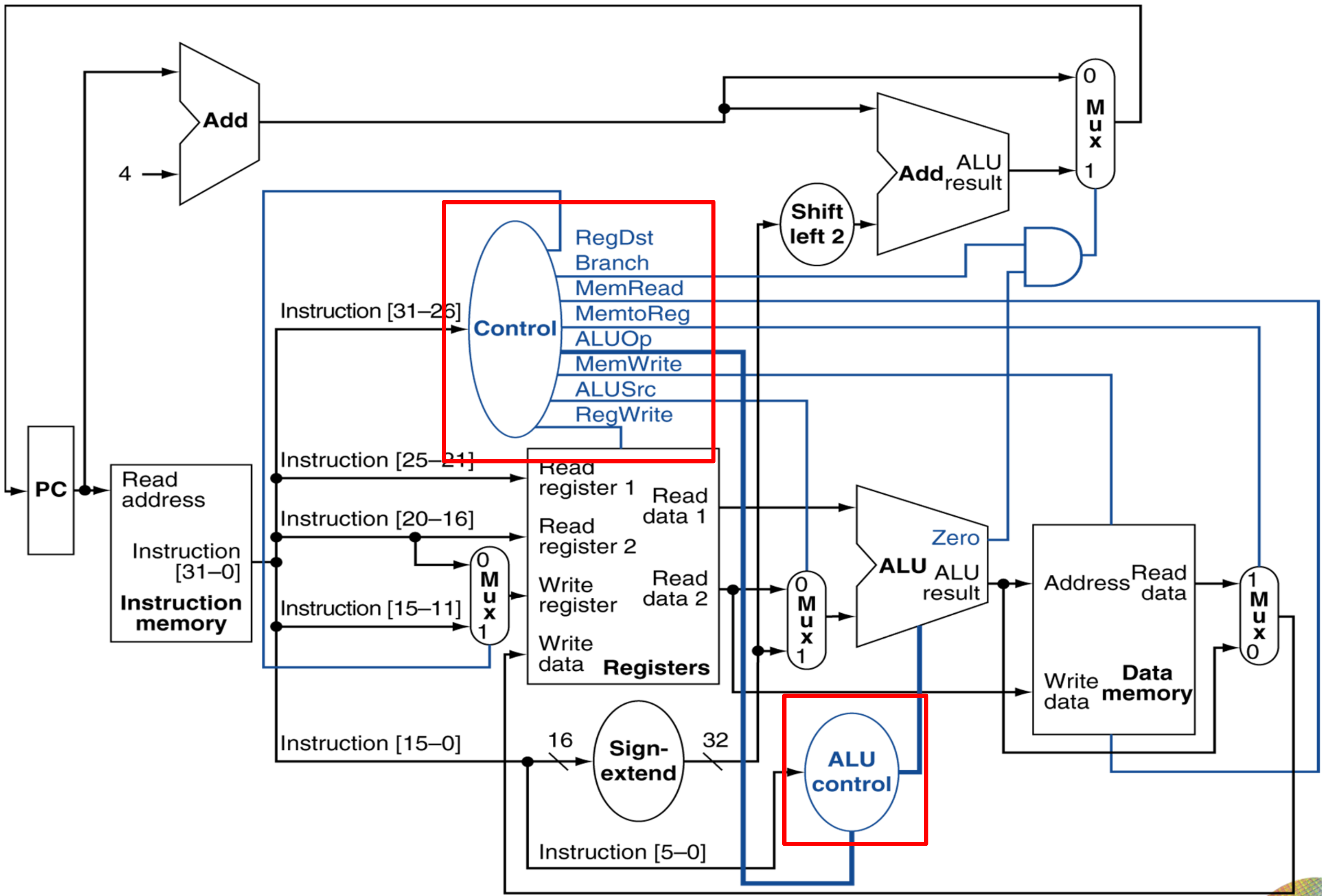
Full Datapath (Single Cycle Datapath)



Control for the single-cycle CPU

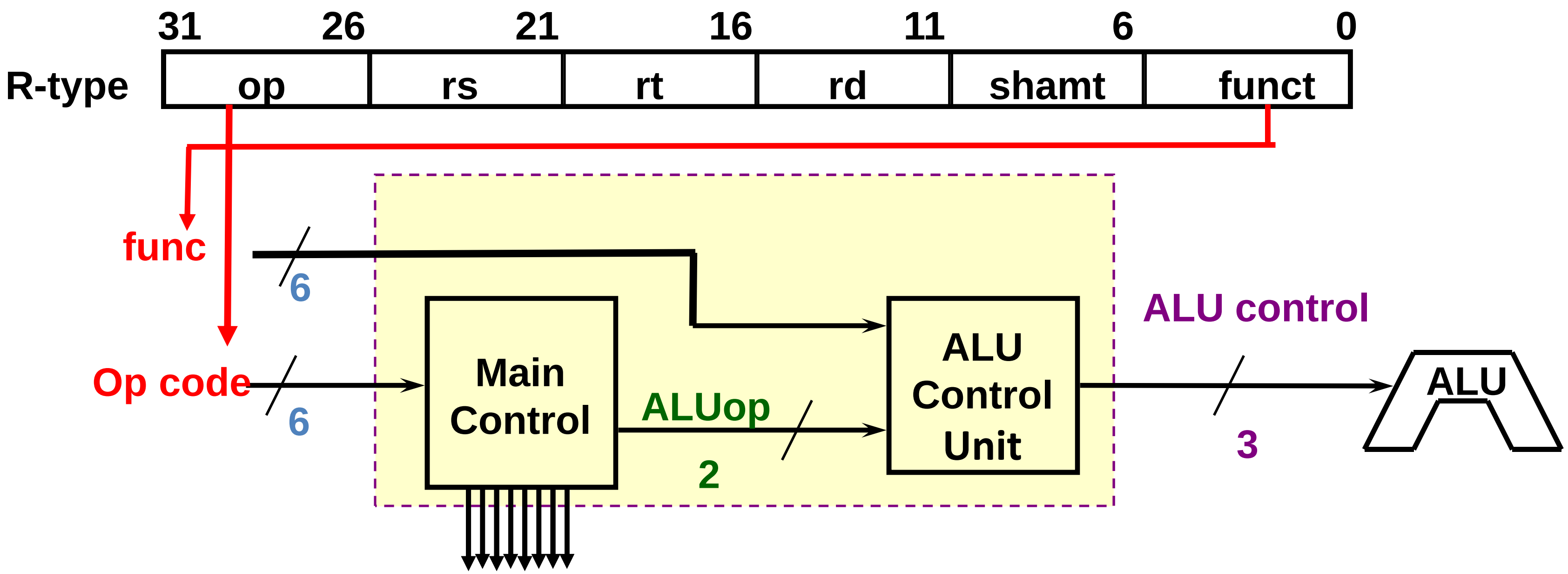
- A single-cycle implementation (the datapath)
  - R-type instruction (Arithmetic-logic instructions)
  - Memory instruction (load/store)
  - Branch
  - J-type instruction (j)
- Determine control for the single-cycle CPU to ensure instructions can be executed correctly
  - Main controller
  - ALU controller

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 signal



RegDst, Branch, MemRead MemtoReg, ALUOp
MemWrite, ALUSrc, RegWrite

Deciding ALU Control

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

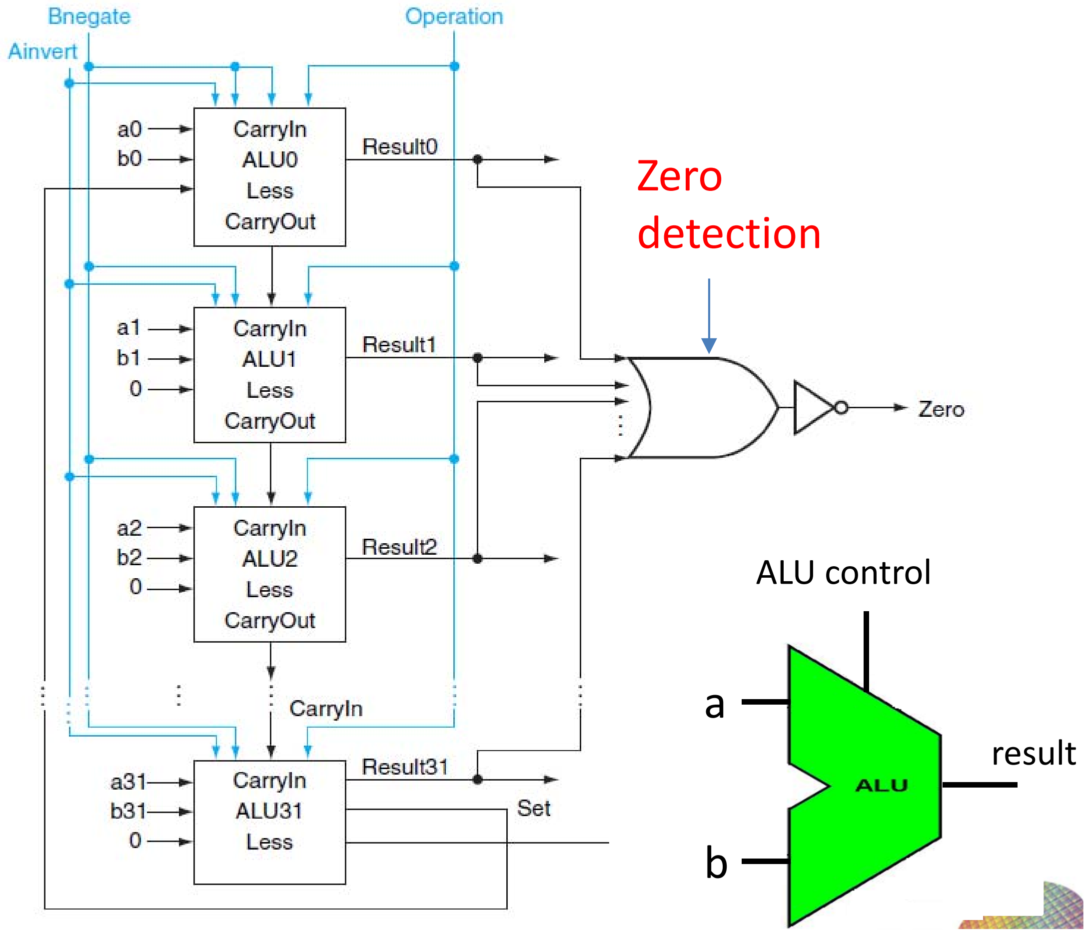
opcode	Operation	funct	ALU function	ALU control
lw	load word	XXXXXX	add	?
sw	store word	XXXXXX	add	?
beq	branch equal	XXXXXX	subtract	?
R-type	add	100000	add	?
	subtract	100010	subtract	?
	AND	100100	AND	?
	OR	100101	OR	?
	set-on-less-than	101010	set-on-less-than	?

Review: 32-bit ALU in Chapter 3

- Binvert is compatible to CarryIn => Connect Binvert to CarryIn => is renamed to Bnegate
- Add Zero detection circuit => If all bit is 0=> Zero=1

Ainvert	Binvert	CarryIn	Op.	Func.
0	0	X	0	a and b
0	0	X	1	a or b
0	0	0	2	a + b
0	1	1	2	a - b
0	1	1	3	slt

Bnegate	Op[1:0]	Func.
0	00	a and b
0	01	a or b
0	10	a + b
1	10	a - b
1	11	slt



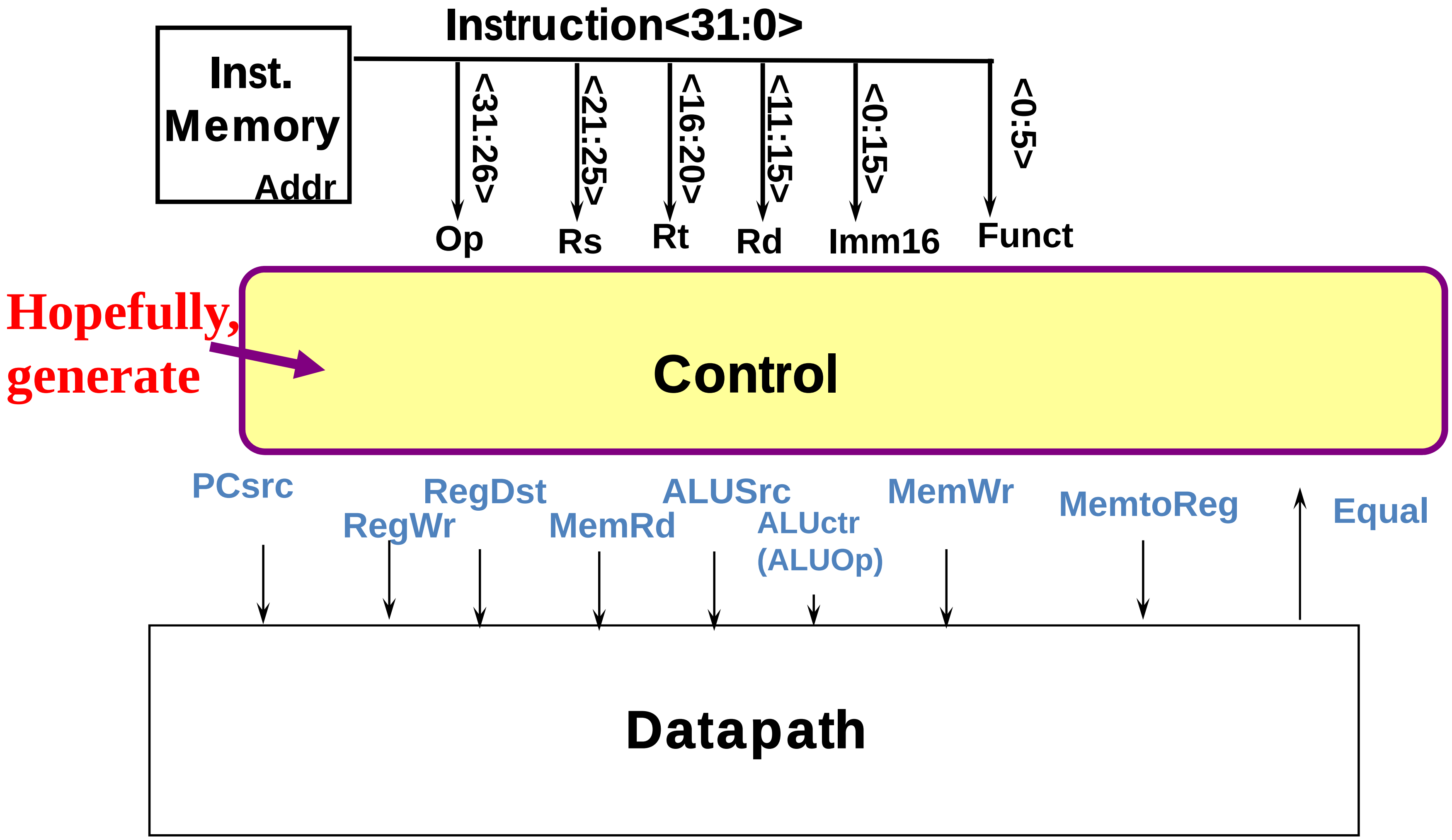
ALU Control

- ALU used for
  - Load/Store: function = add
  - Branch: function = subtract
  - R-type: function depends on funct field
- Assume 2-bit ALUOp derived from opcode (generated by main controller)
  - Use ALUOp and funct to generate "ALU control" (discuss later)

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

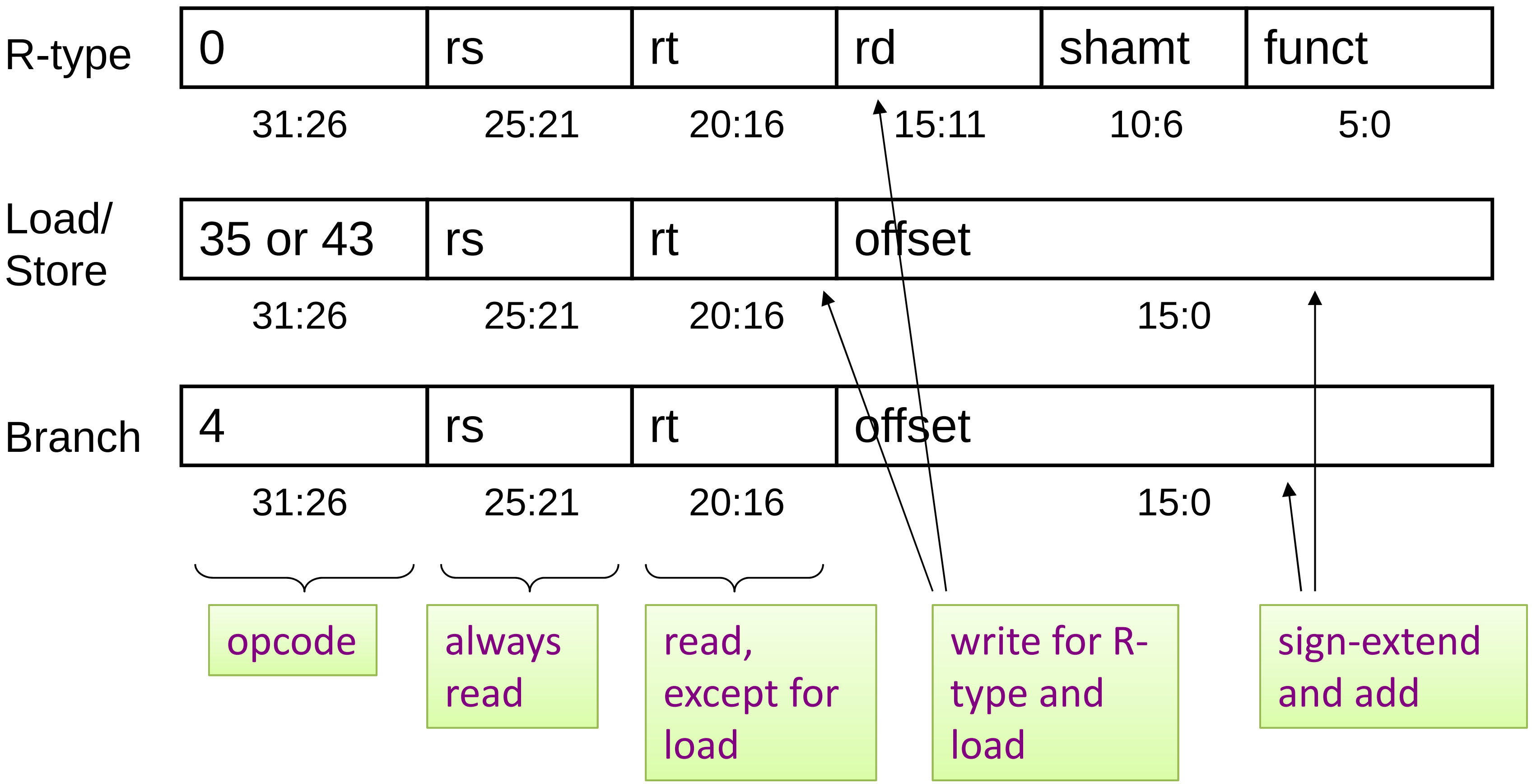
Determine Main Control Signals

- Control l signal

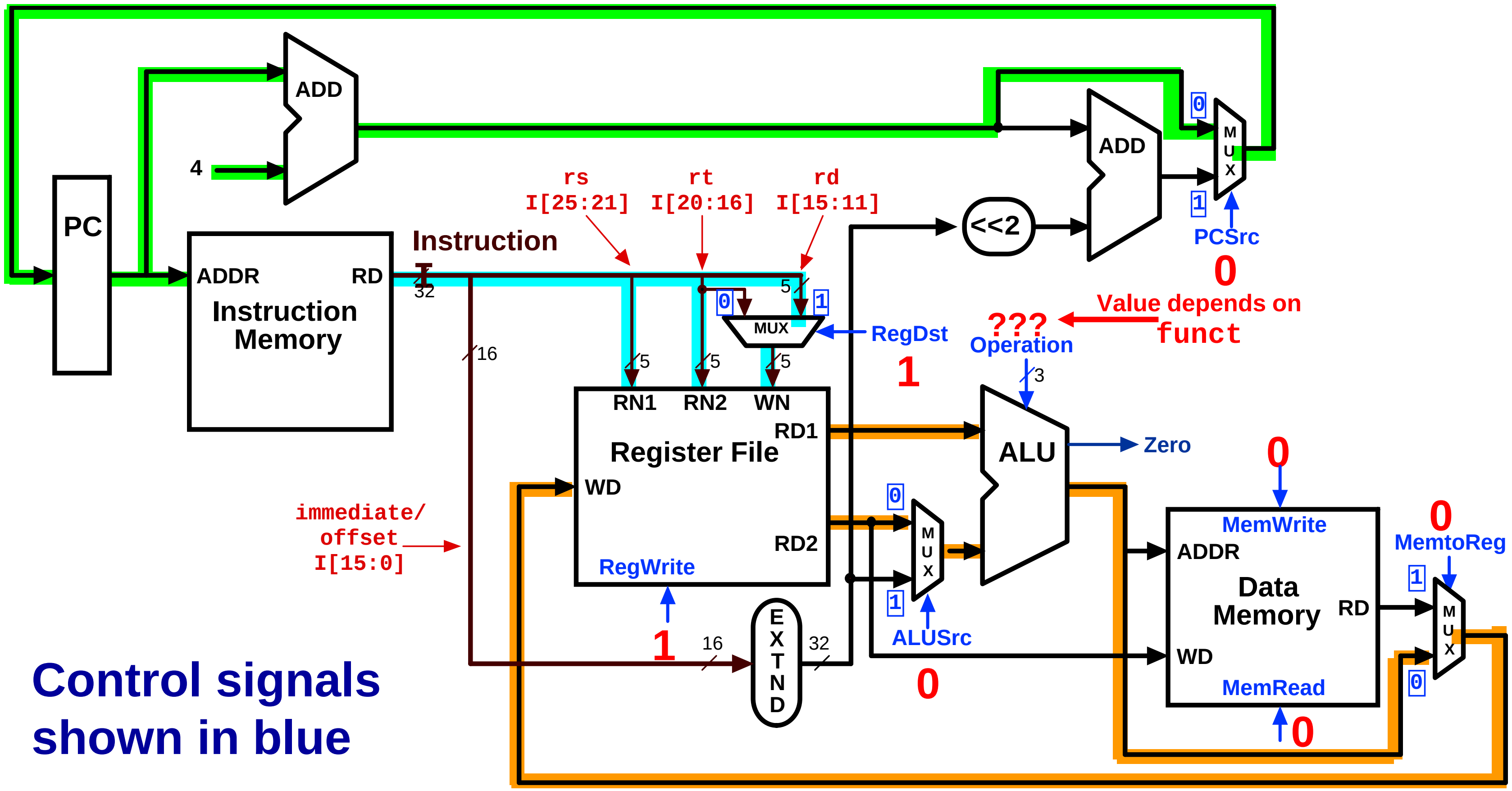


Review: The Main Control Unit

- Control signals derived from instruction



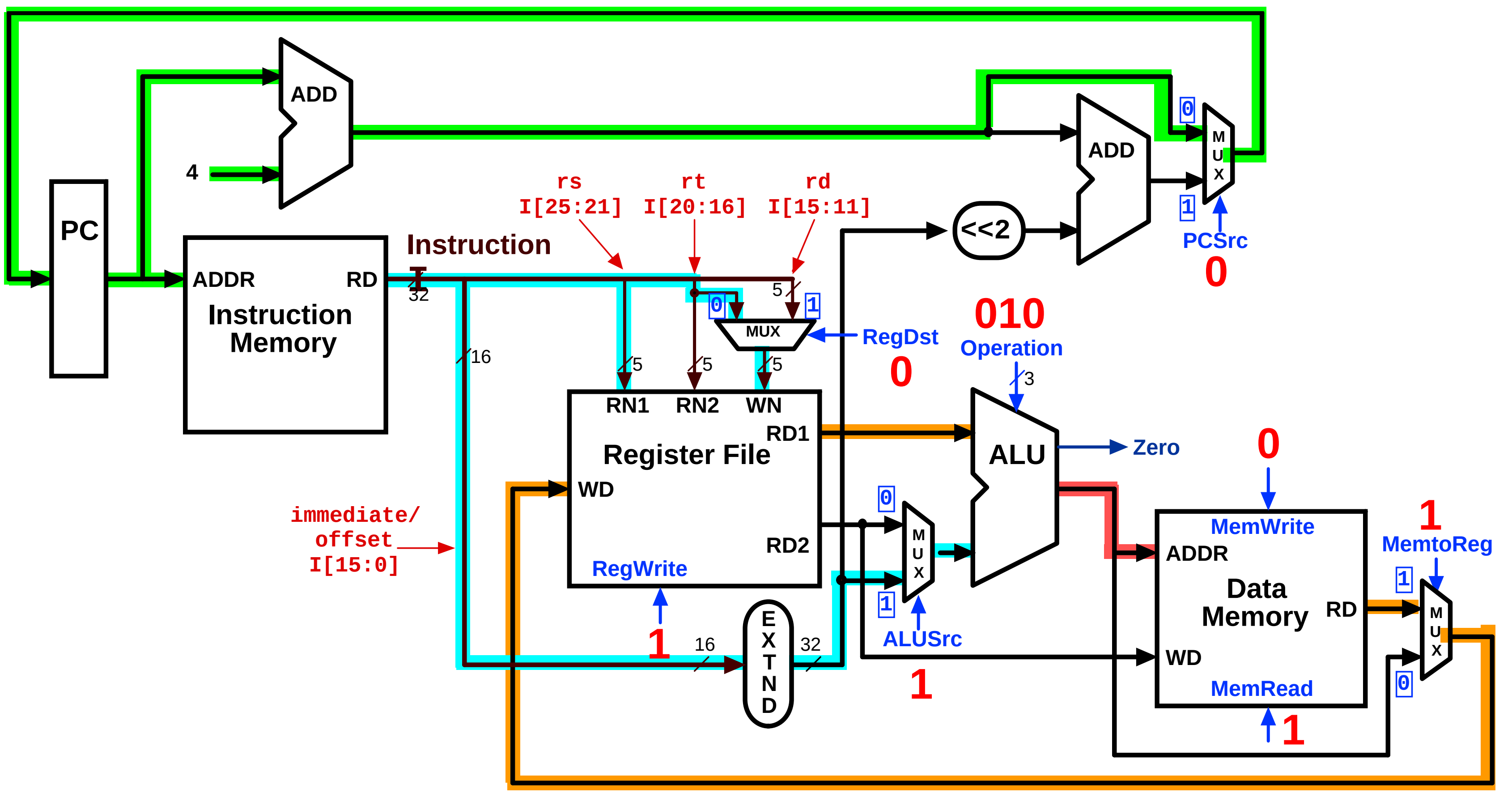
Control Signals for R-Type Instruction



Control signals shown in blue

R-type	0	rs	rt	rd	shamt	funct
	31:26	25:21	20:16	15:11	10:6	5:0

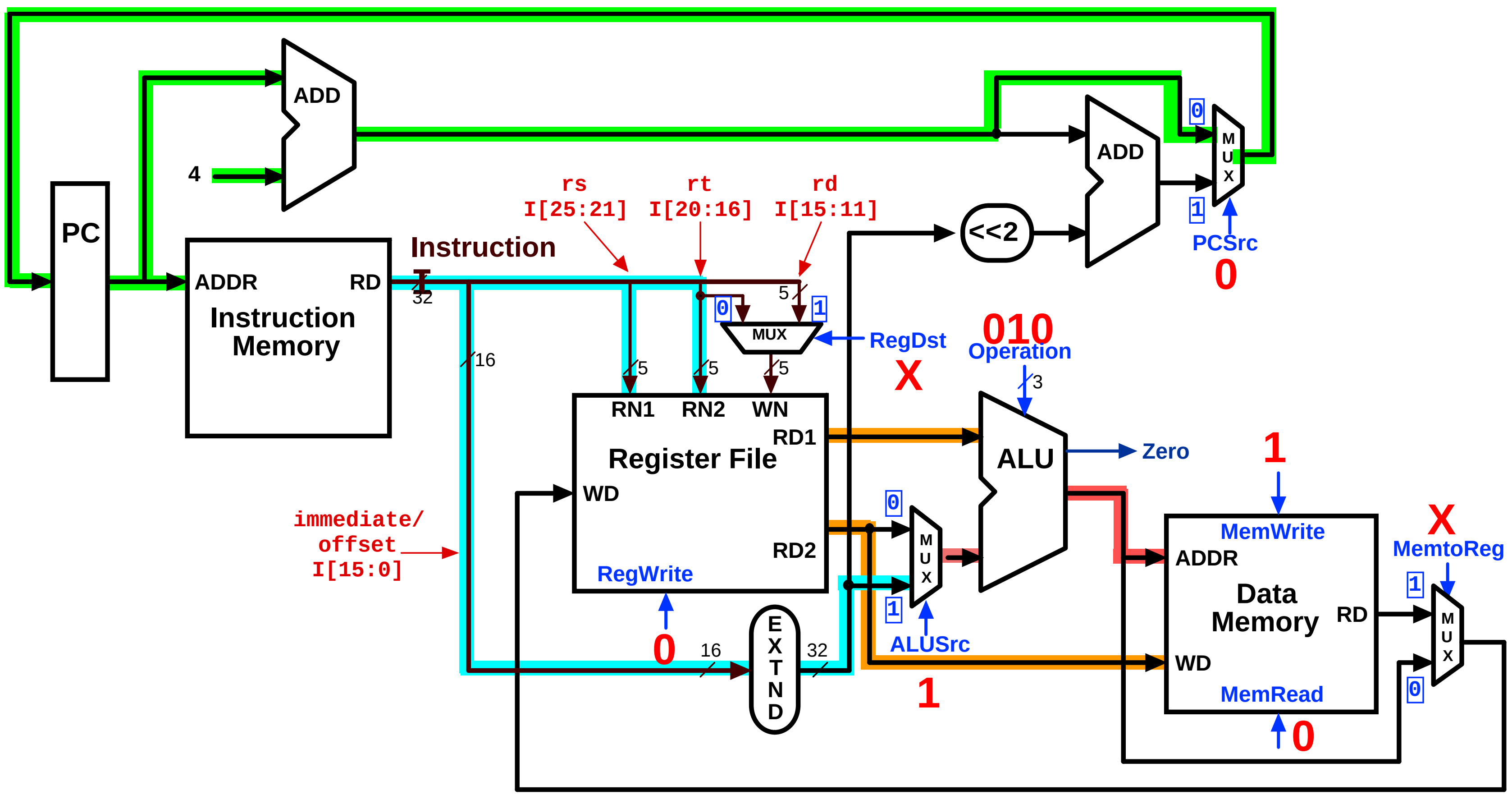
Control Signals: lw Instruction



Load	35 or 43	rs	rt	offset
	31:26	25:21	20:16	15:0

$rt register is written

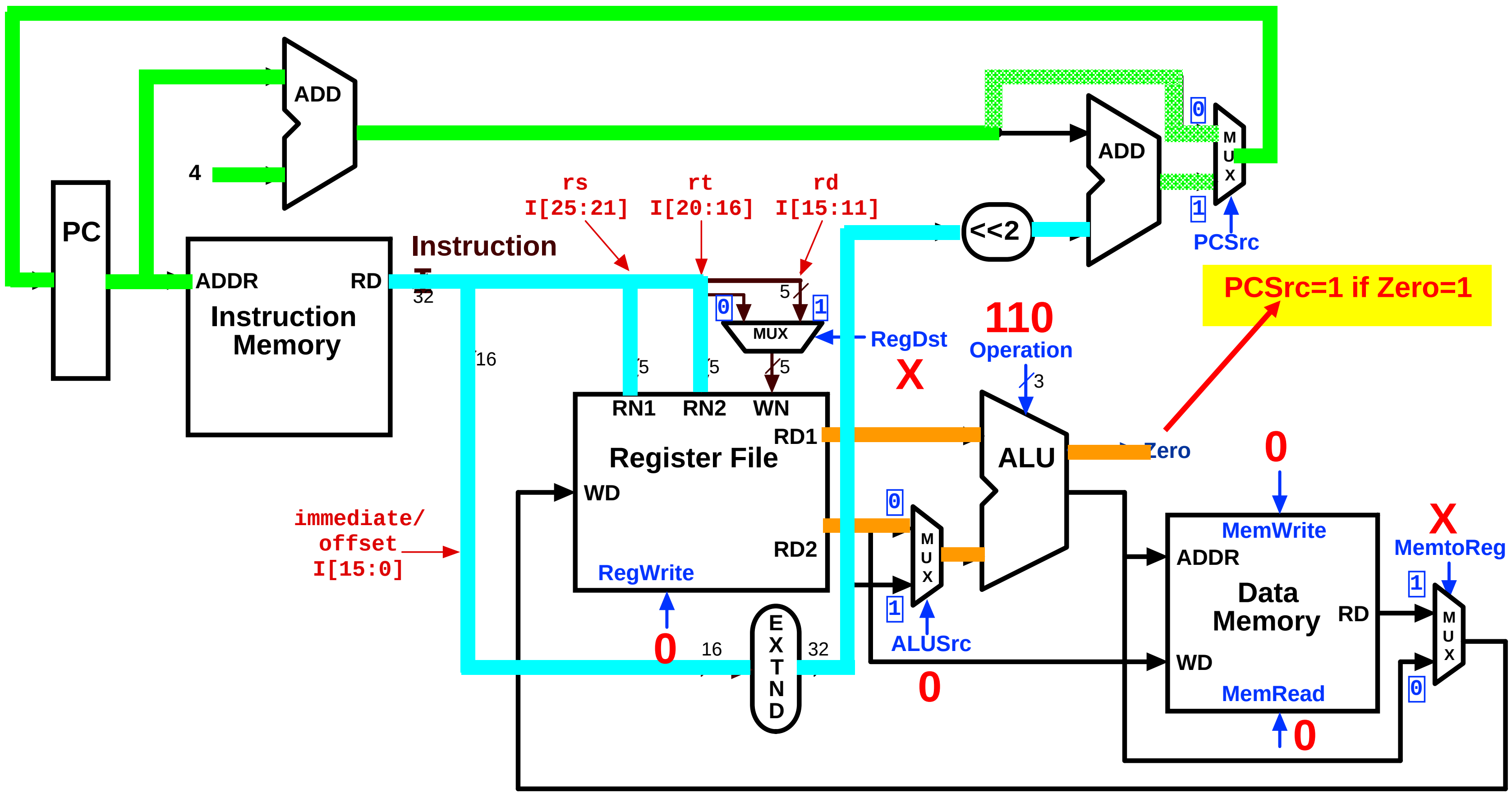
Control Signals: sw Instruction



sw	35 or 43	rs	rt	offset
	31:26	25:21	20:16	15:0

$rt register is read

Control Signals: beq Instruction

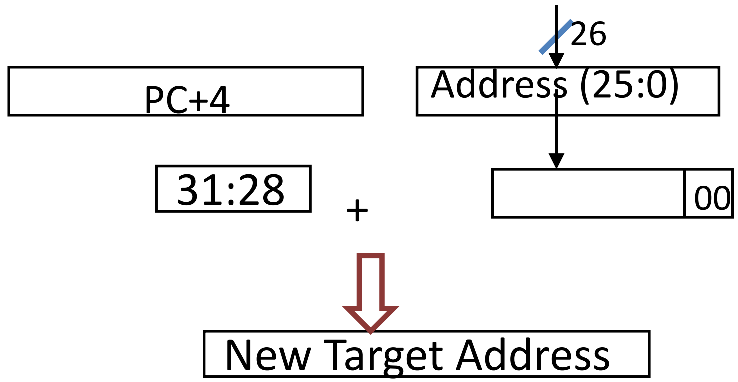


Branch	4	rs	rt	offset
	31:26	25:21	20:16	15:0

Review: Implementing Jumps

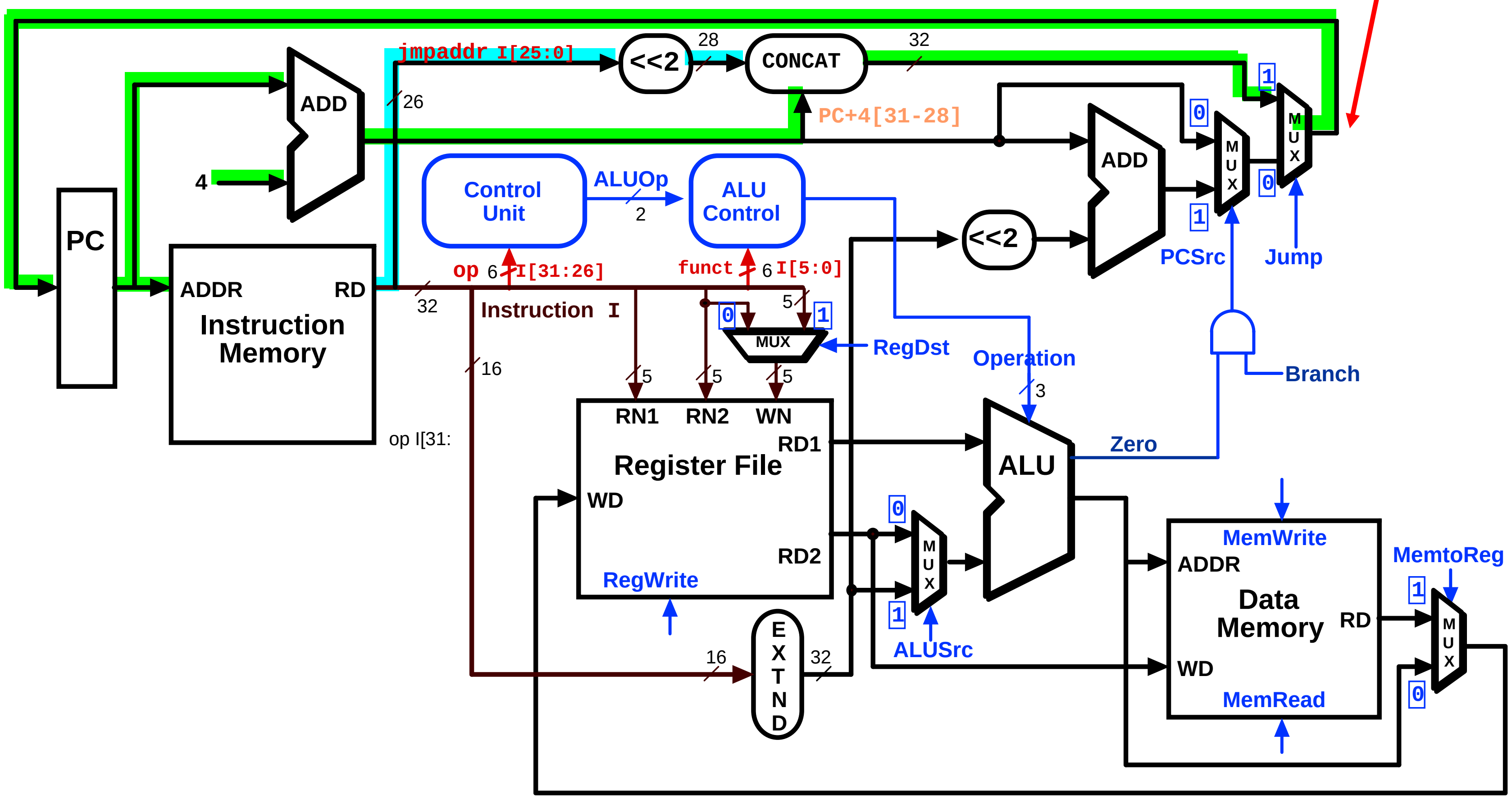
Jump	2	address
	31:26	25:0

- 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 instruction

Mux for Jump Inst
Jump=1 if it is a jump instruction



Jump	2	address
	31:26	25:0

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 has 6 bits) as "inputs", the "outputs" are as follows => a main control logic (the next slide)

See appendix D for details

inputs — outputs

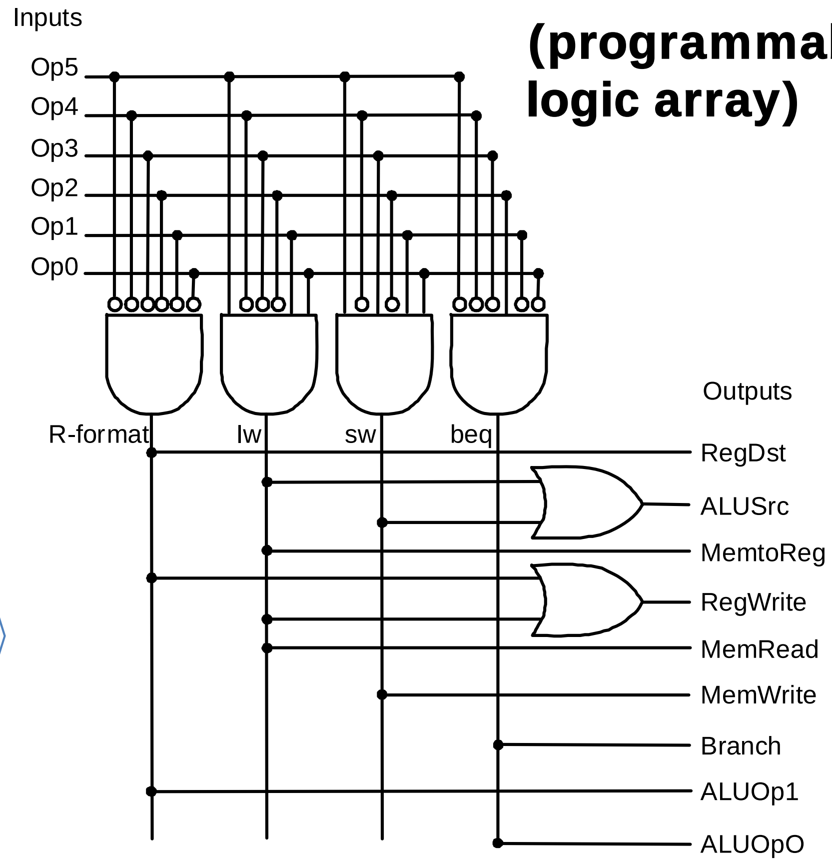
Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUOp0
R-format 000000	1	0	0	1	0	0	0	1	0
lw 100011	0	1	1	1	1	0	0	0	0
sw 101011	X	1	X	0	0	1	0	0	0
beq 000100	X	0	X	0	0	0	1	0	1

Implementation of Main Control Block (Use PLA)

	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

(programmable logic array)



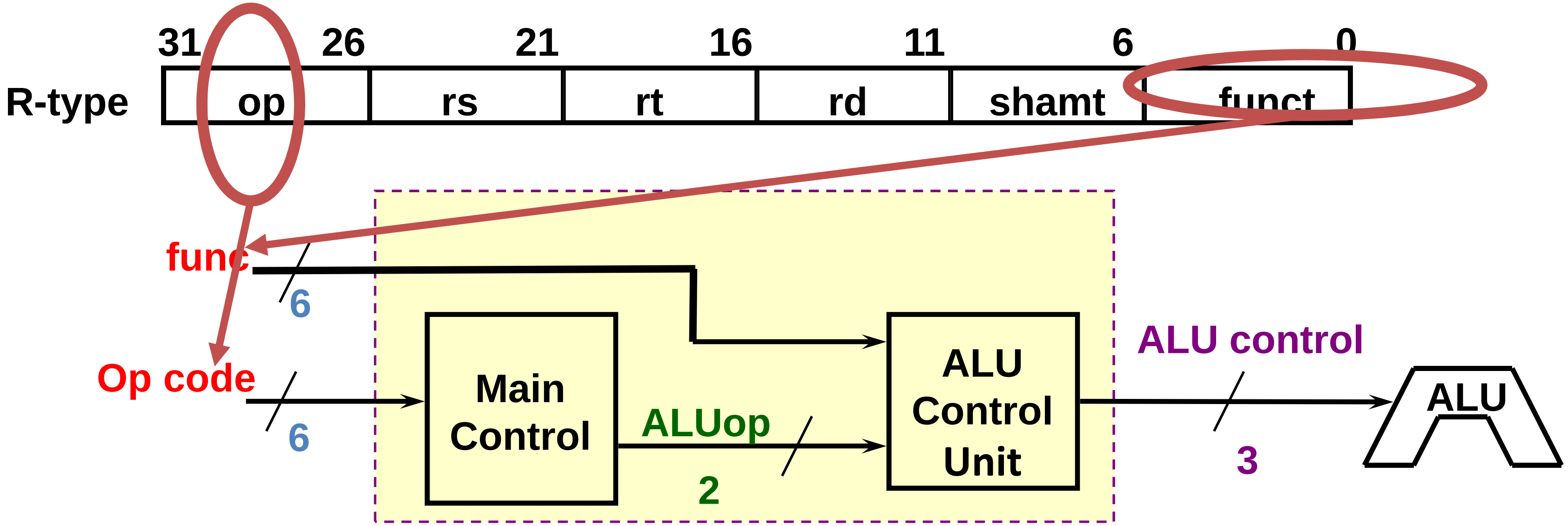
Main control PLA

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

$$ALUSrc = \begin{matrix} (Op5 \cdot \overline{Op4} \cdot \overline{Op3} \cdot \overline{Op2} \cdot Op1 \cdot Op0)\, or \\ (Op5 \cdot \overline{Op4} \cdot Op3 \cdot \overline{Op2} \cdot Op1 \cdot Op0) \end{matrix}$$

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



Input Input output

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

inputs outputs

Merge LW & SW

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
0	1	X	X	X	X	X	X	0110
1	0	X	X	0	0	0	0	0010
1	0	X	X	0	0	1	0	0110
1	0	X	X	0	1	0	0	0000
1	0	X	X	0	1	0	1	0001
1	0	X	X	1	0	1	0	0111

(lw/sw) => add
(beq) => subtract
(add) =>add
(sub) => subtract
(and) => and
(or) => or
(slt) => slt

$C_3=0,\ C_2=?,\ C_1=?,\ C_0=?$

C_0=?

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
0	1	X	X	X	X	X	X	0110
1	0	X	X	0	0	0	0	0010
1	0	X	X	0	0	1	0	0110
1	0	X	X	0	1	0	0	0000
1	0	X	X	0	1	0	1	0001
1	0	X	X	1	0	1	0	0111

C_0=1 at row 6 & row 7, how to identify row 6 and row 7

$C0 = (\text{ALOP 1 and F0})$ or (ALUOP1 and F3)
= ALUOP1 and (F0 or F3)

C_1=?

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
0	1	X	X	X	X	X	X	0110
1	0	X	X	0	0	0	0	0010
1	0	X	X	0	0	1	0	0110
1	0	X	X	0	1	0	0	0000
1	0	X	X	0	1	0	1	0001
1	0	X	X	1	0	1	0	0111

C_1 is 1 at row 1, 2, 3, 4 & 7,
How to identify row 1, 2,3, 4 and 7

$$C1 = \overline{F2} \;\; or \;\; \overline{ALUOP1}$$

Row 3 or 4 or 7

Row 1 or 2

inputs

C_2=?

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
0	1	X	X	X	X	X	X	0110
1	0	X	X	0	0	0	0	0010
1	0	X	X	0	0	1	0	0110
1	0	X	X	0	1	0	0	0000
1	0	X	X	0	1	0	1	0001
1	0	X	X	1	0	1	0	0111

C2=1 at row 2, row 4 & row 7

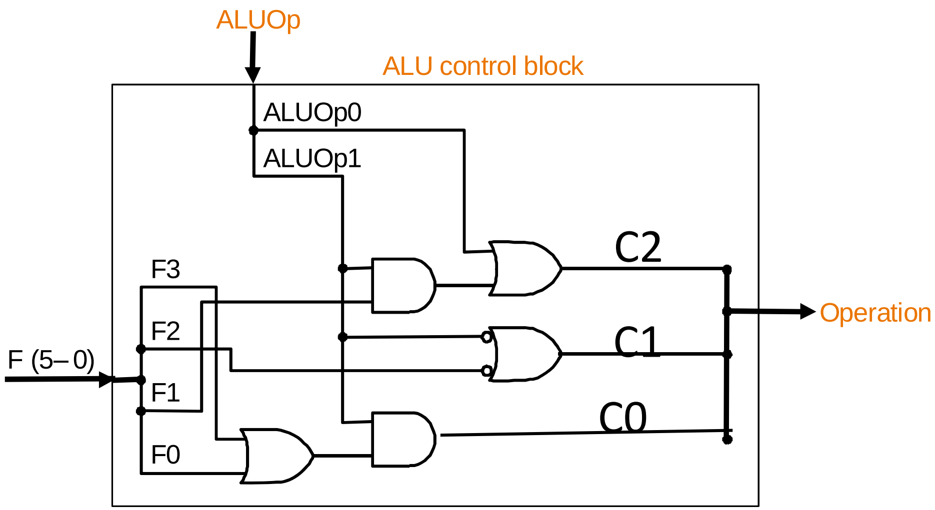
How to identify row 2, row 4, and row7

$$\mathrm{C2} = \mathrm{ALUOp0} \ \text{ or } (\mathrm{ALUOp1} \text{ and } \mathrm{F1})$$

Row 2　　Row 4 & Row 7

$$C1 = \overline{F2} \ \ or \ \ \overline{ALUOP1}$$

$$C0 = ALUOP1 \ and \ (F0 \ or \ F3)$$



Why a SC implementation is not used today

- In single-cycle design, each clock cycle must have the same length for every instruction
  - Longest delay determines clock period
- Critical path (longest delay): load instruction
  - Instruction memory $\rightarrow$ register file $\rightarrow$ ALU $\rightarrow$ data memory $\rightarrow$ register file
- Performance is poor because clock cycle is too long
  - Violates design principle: Making the common case fast
- We will improve performance by pipelining
  - Run multiple instructions simultaneously

Backup Slides