Chapter 4
The Processor

Da-Wei Chang, OSES Lab.
CSIE Dept., NCKU

Introduction

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

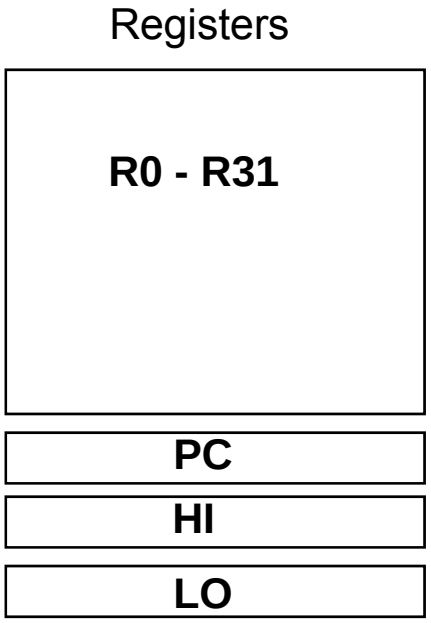
CPU Time=Instruction Count ×CPI ×Clock Cycle Time

- We will examine two MIPS implementations
  - A simplified version (single-cycle implementation)
  - A more realistic pipelined version
- Implement simple subset
  - Memory reference: `lw`, `sw`
  - Arithmetic/logical: `add`, `sub`, `and`, `or`, `slt`
  - Control transfer: `beq`, `j`

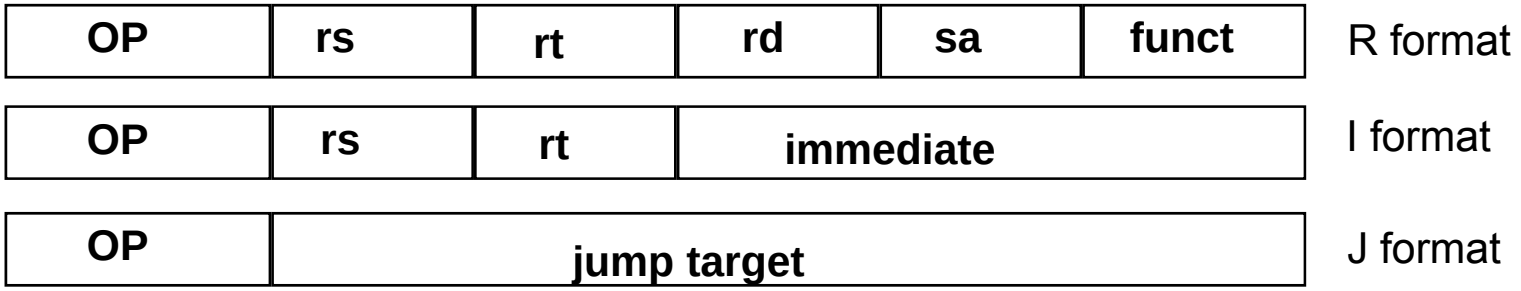
*illustrates key principles in creating a datapath and designing the control

Review: MIPS Instruction Set Architecture (ISA)

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



3 Instruction Formats: all 32 bits wide

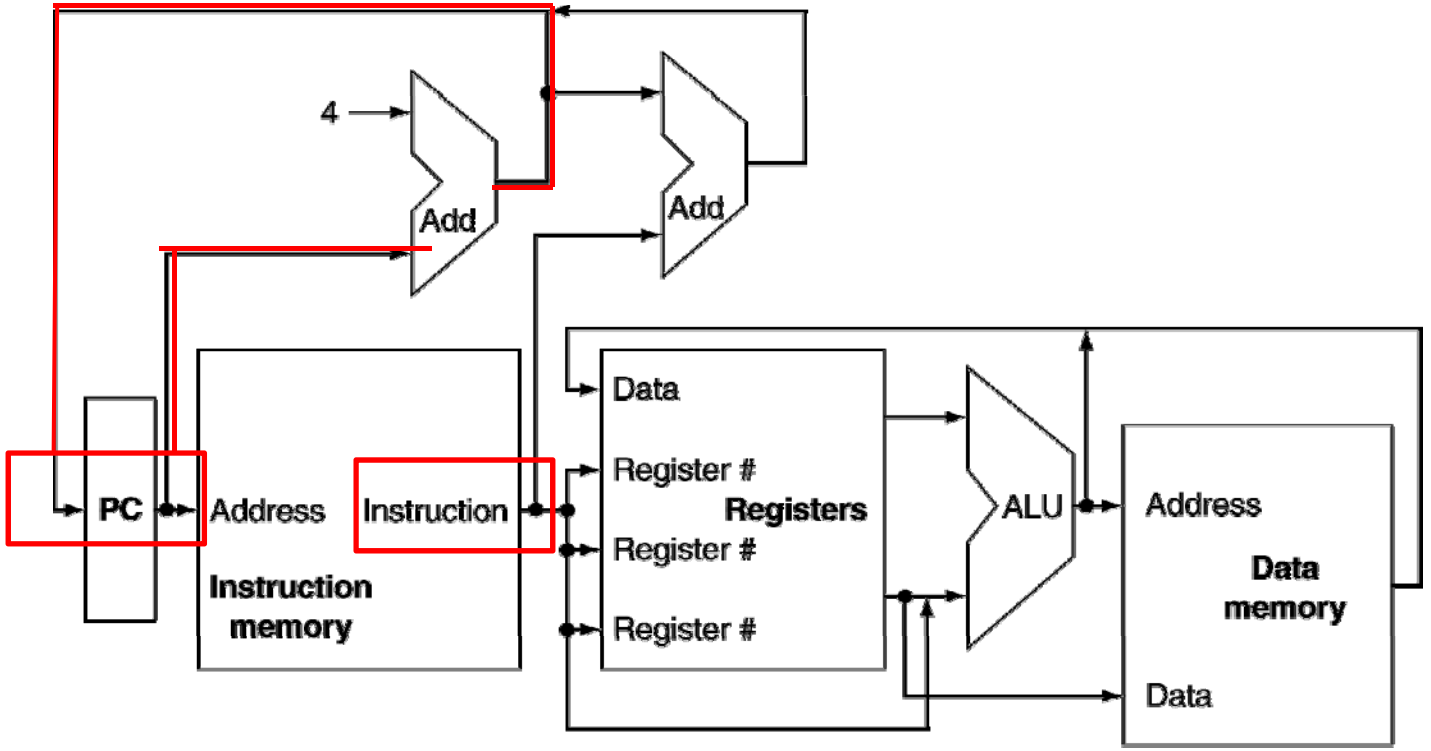


Review: MIPS Register Convention

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 (hardware)	yes

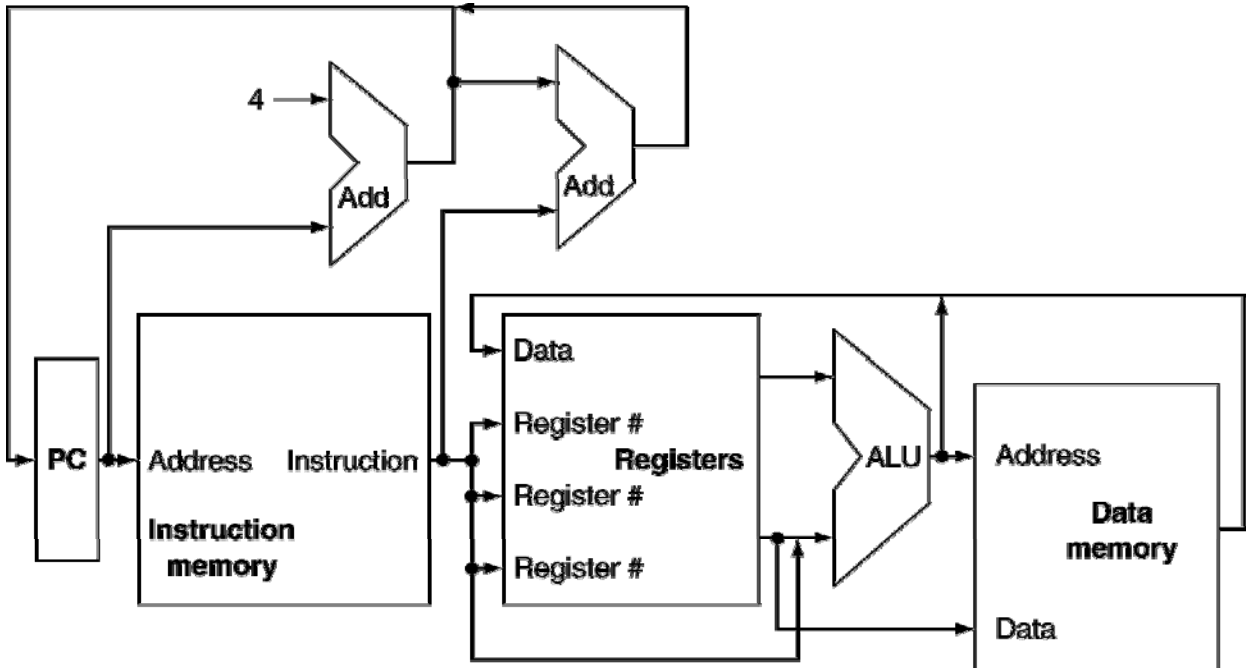
Instruction Execution

- 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

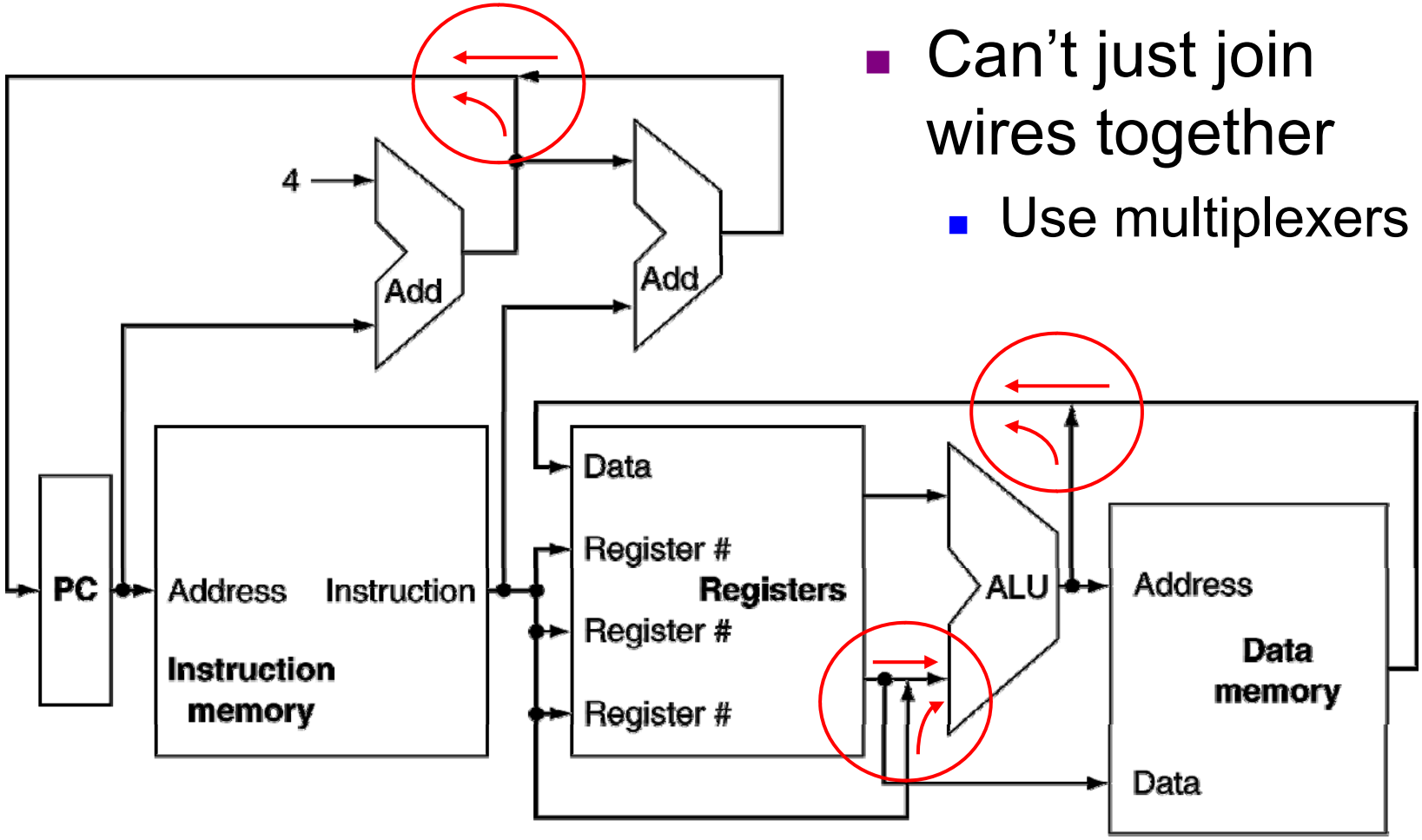


- Depending on instruction class, different actions are performed
  - Use ALU to calculate
    - Arithmetic result
    - Memory address for load/store
    - Branch target address
  - Access data memory for load/store
  - PC ← target address

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

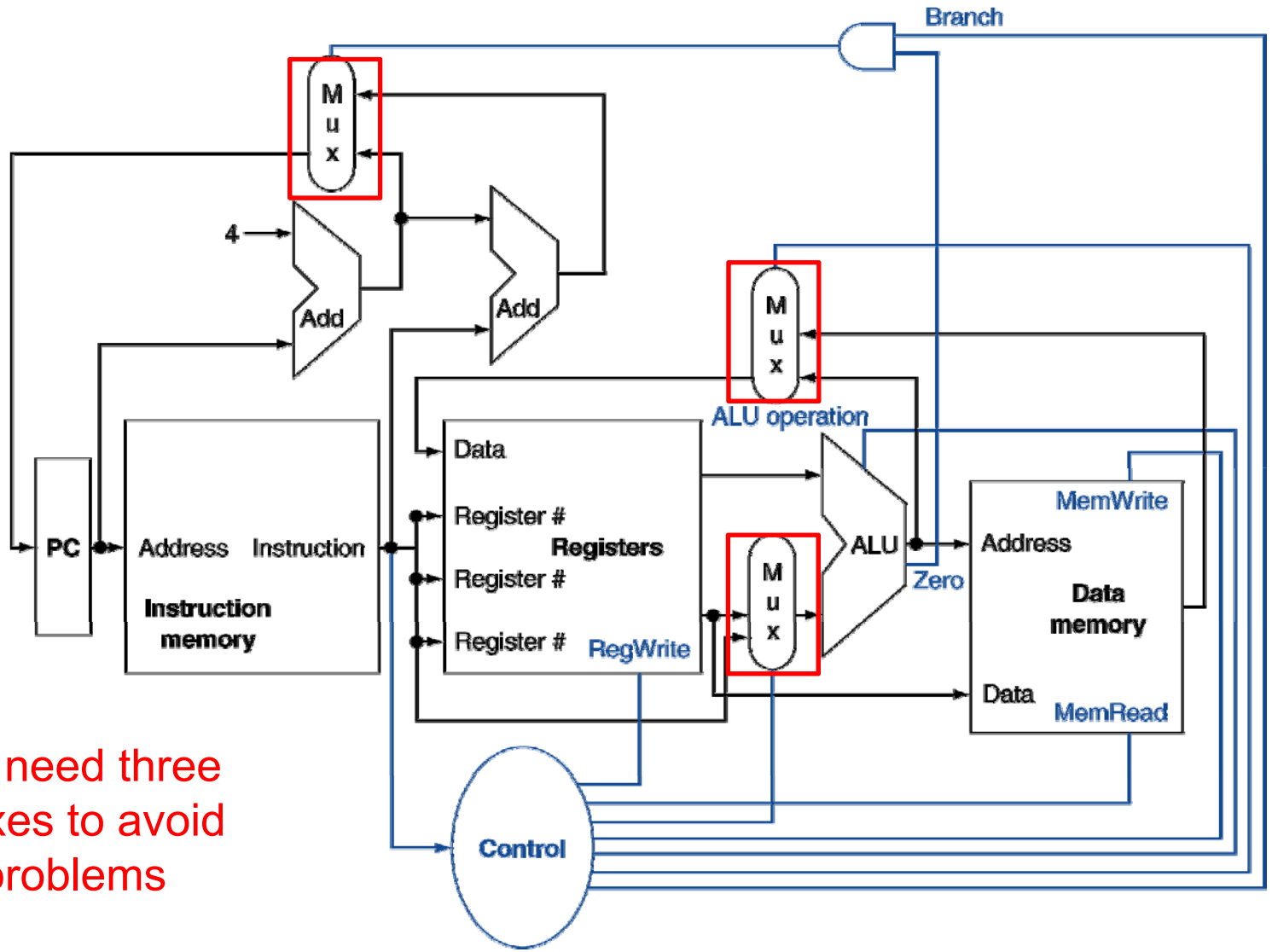


Multiplexers



- Can't just join wires together
  - Use multiplexers

CPU Overview

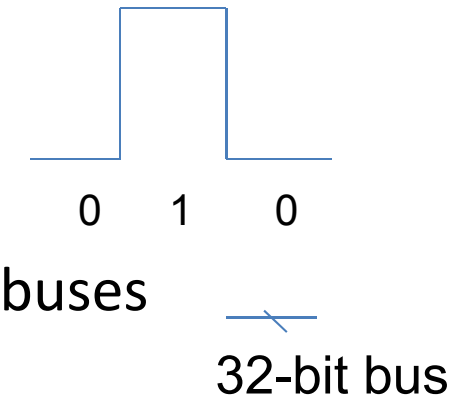


We need three Muxes to avoid problems

Details of each Mux and Control will be introduced later

Logic Design Basics

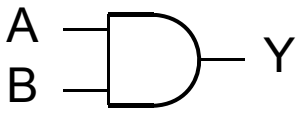
- Information encoded in binary
  - Low voltage = 0, High voltage = 1
  - One wire per bit
  - Multi-bit data encoded on multi-wire buses



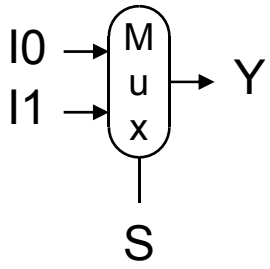
- Combinational element (See next slide)
  - Operate on data
  - Output is a function of input
- State (sequential) elements
  - Output is a function of input and current states
  - Store information

Review: Combinational Elements

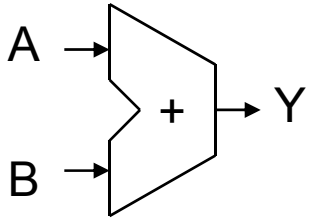
- AND-gate
  - Y = A & B



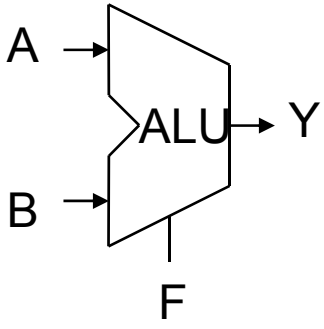
- Multiplexer
  - Y = S ? I1 : I0



- Adder
  - Y = A + B

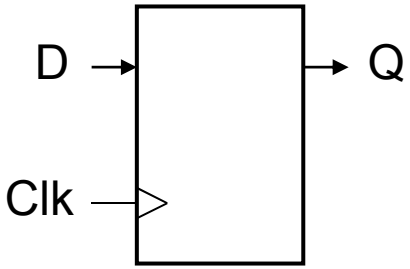


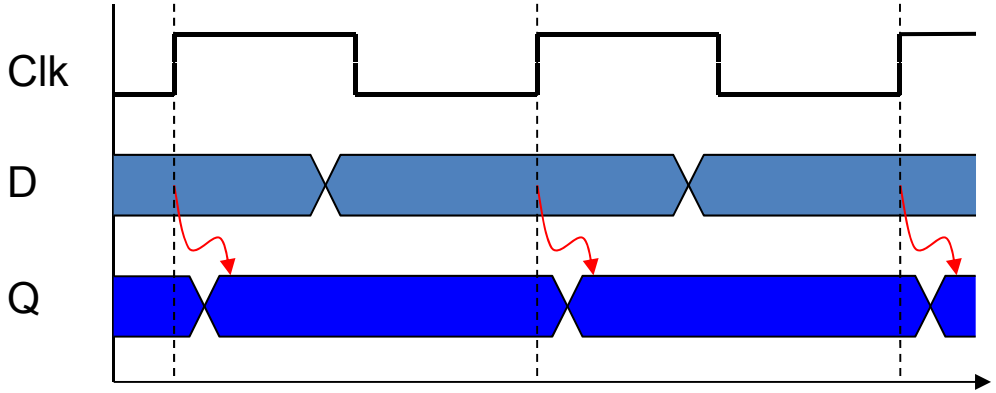
- Arithmetic/Logic Unit
  - Y = F(A, B)



Review: State/Sequential Elements

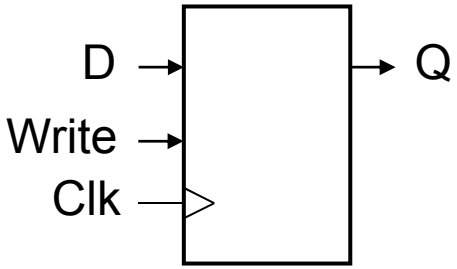
- Register: stores data in a circuit
  - Uses a clock signal to determine when to update the stored value
  - Edge-triggered: update when Clk changes (0-> 1 or 1-> 0)
  - The following figure is positive edge-triggered: update when Clk changes from 0 to 1

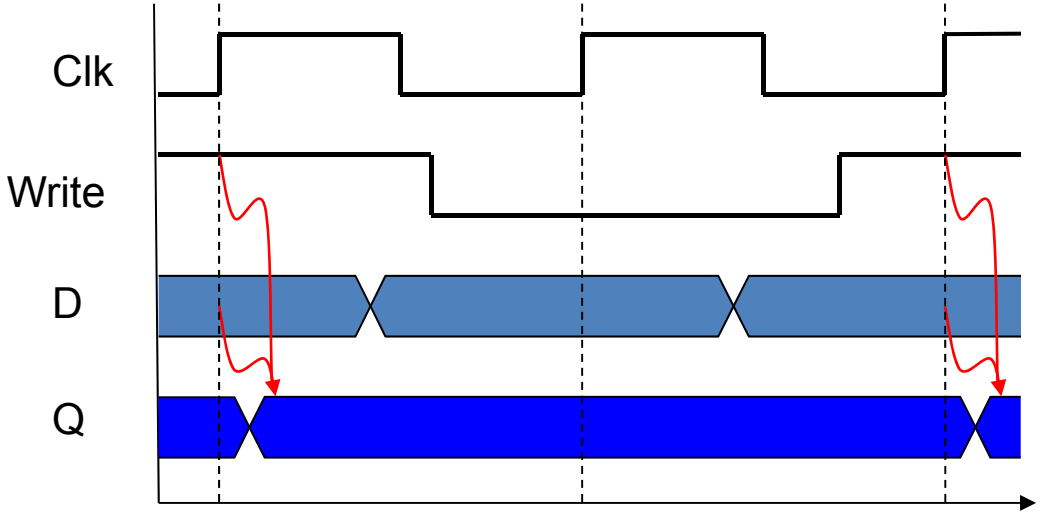




Review: State Elements (with write enable)

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

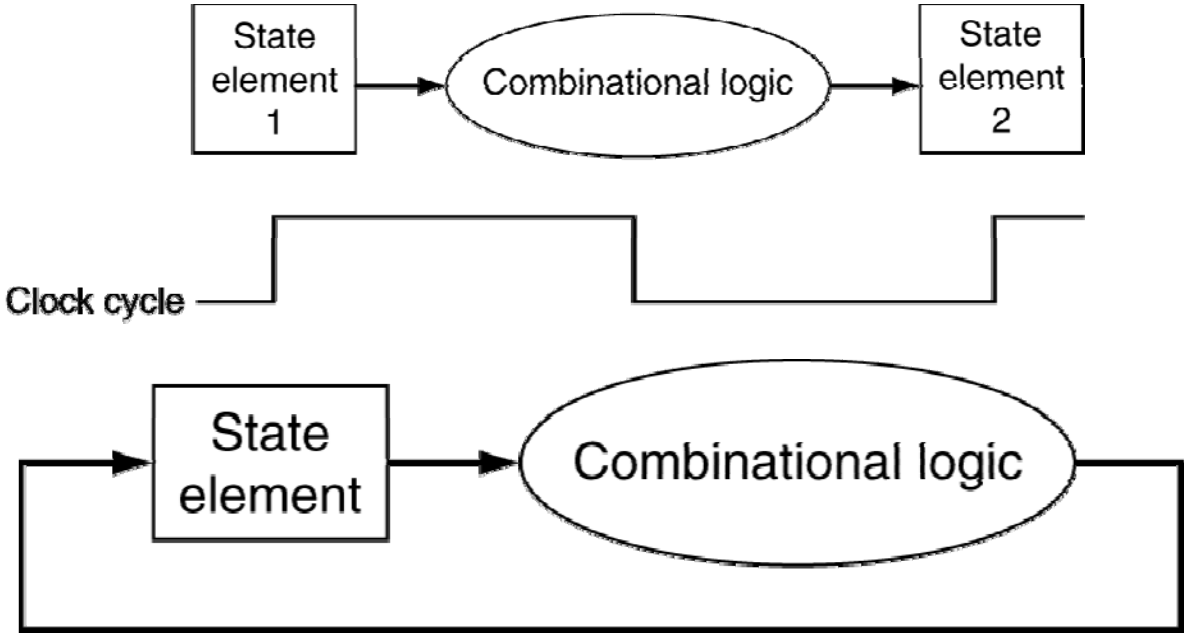




Q is not changed when Write=0

Clocking Methodology

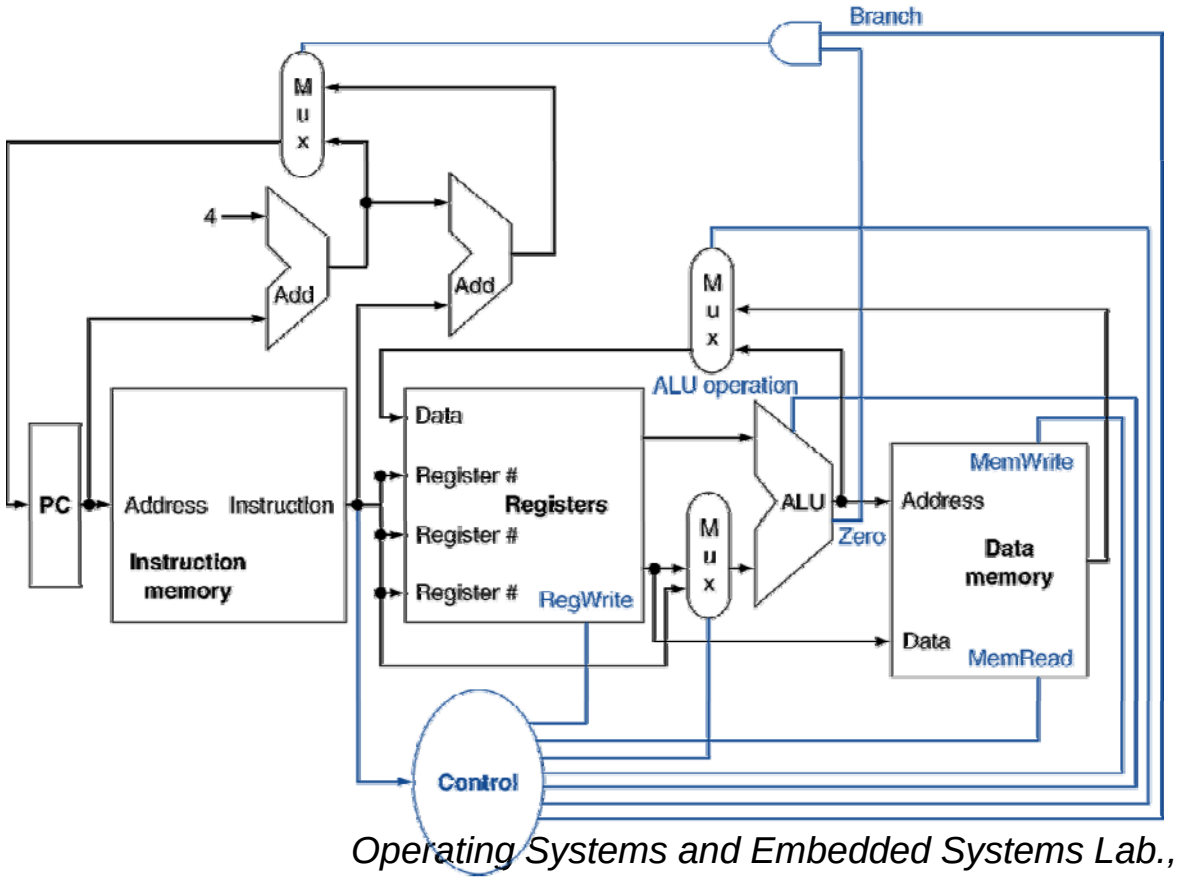
- Combinational logic transforms data from state elements to state elements during clock cycles
  - Between clock edges
  - Longest delay determines clock period



State element read and written in the same clock cycle

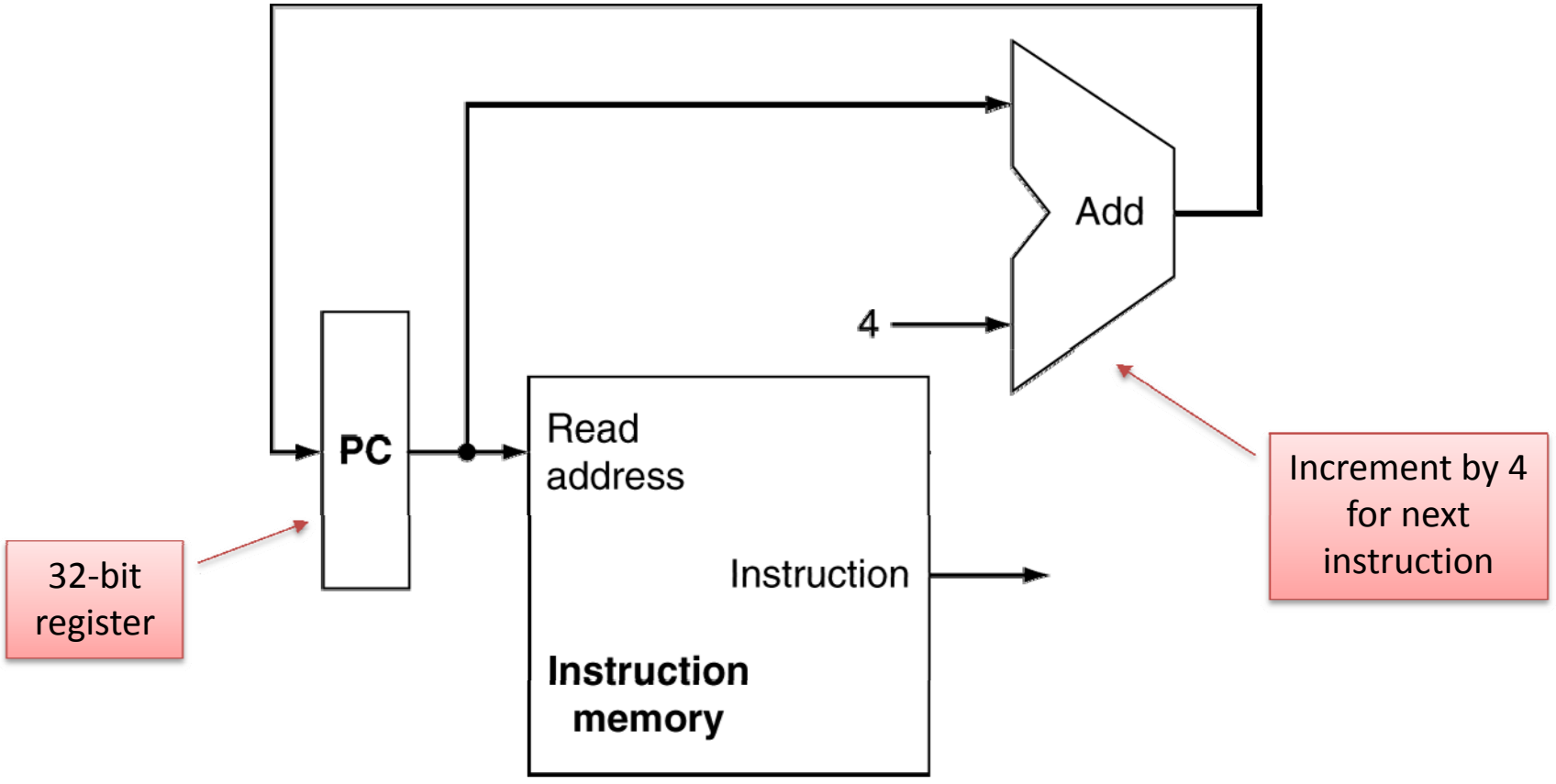
Building a Datapath

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



We will show how to build the MIPS datapath

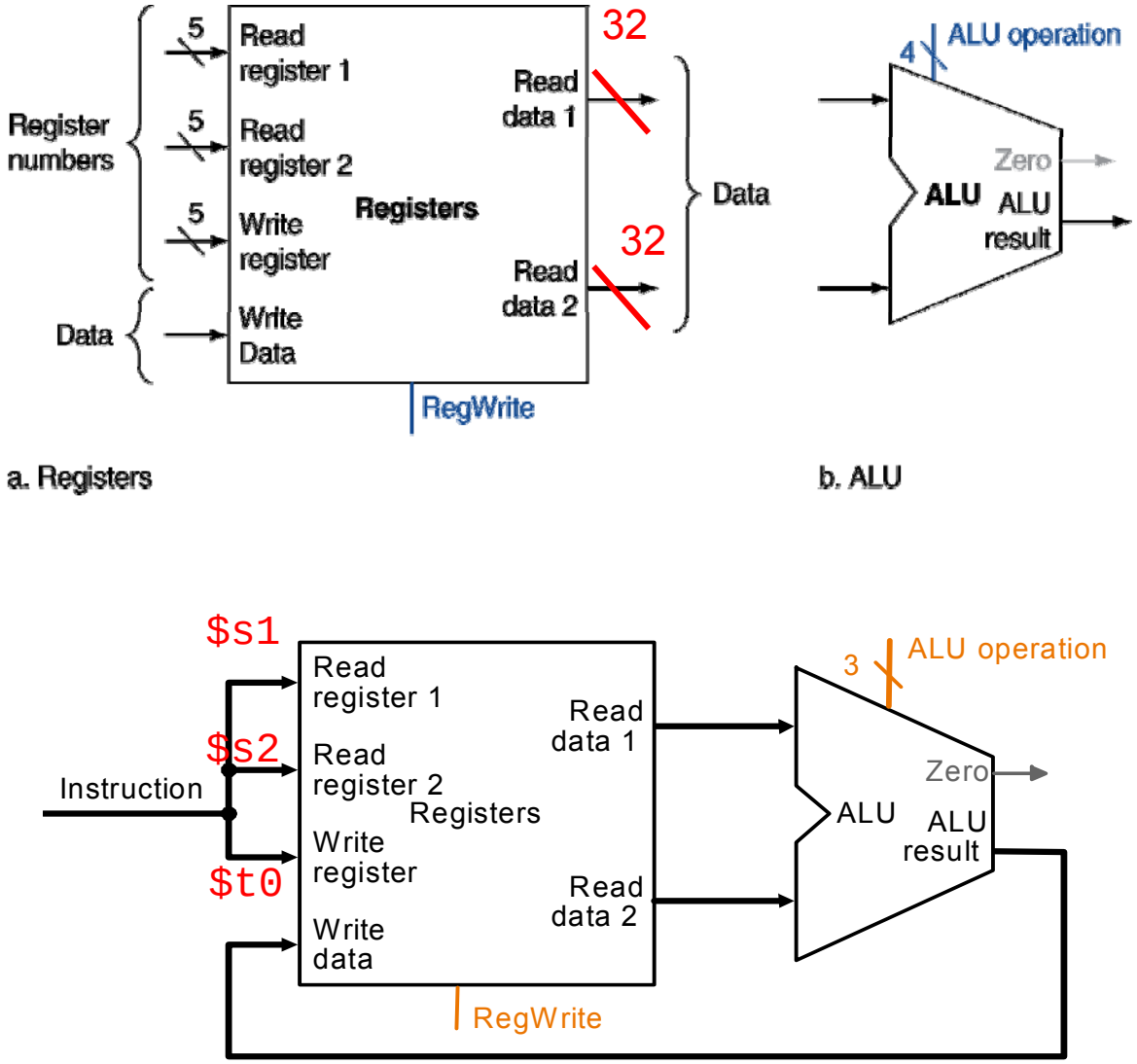
Instruction Fetch



R-Format Instructions

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

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



Review: Load/Store Instructions (I-format)

- MIPS has two basic data transfer instructions for accessing memory

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

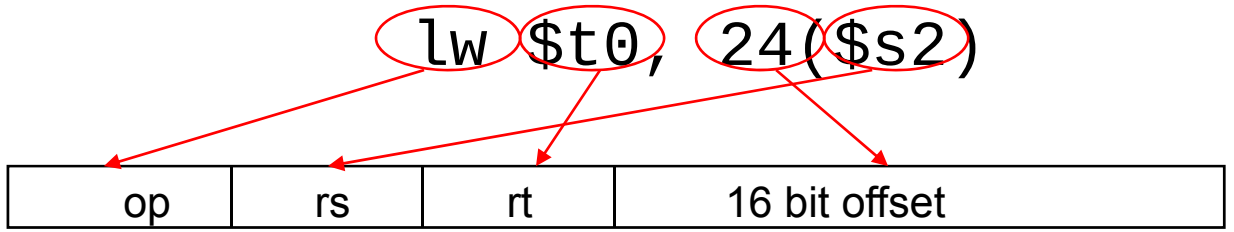
- The data is loaded into (lw) or stored from (sw) a register in the register file – a 5-bit address
- The memory address – a 32-bit address – is formed by adding the contents of the base address register to the offset value
  - A 16-bit field meaning access is limited to memory locations within a region of $\pm 2^{13}$ or 8,192 words ($\pm 2^{15}$ or 32,768 bytes) of the address in the base register
  - Note that the offset can be positive or negative

Ex.

op	rs	rt	16 bit offset
lw	$s3	$t0	4

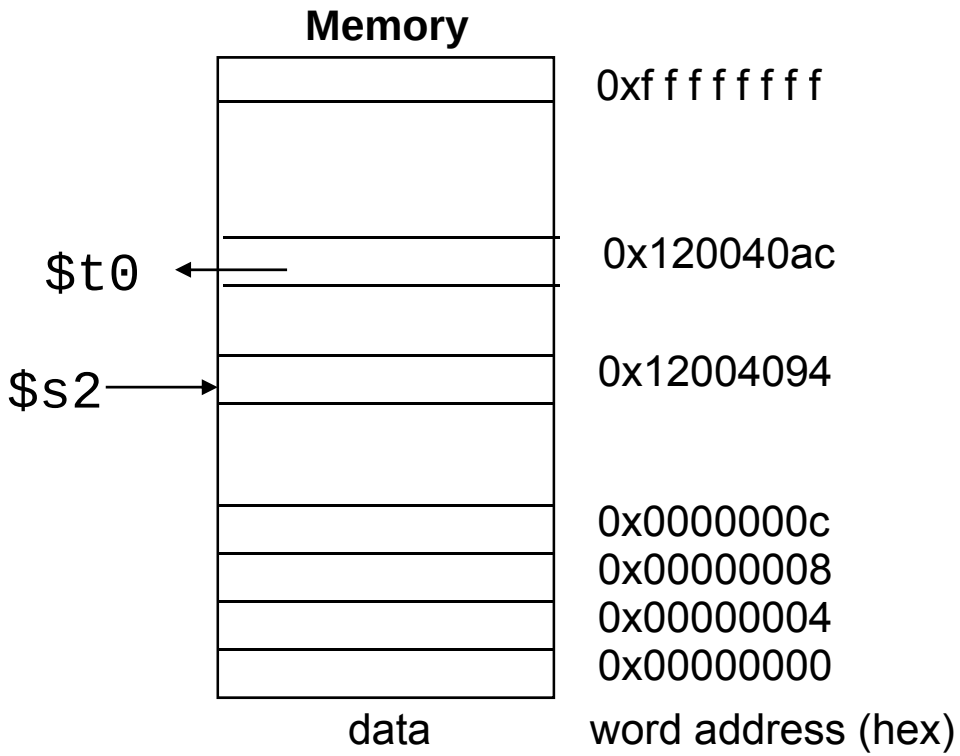
Review - Load Instruction (I format)

- Load/Store Instruction Format (I format):



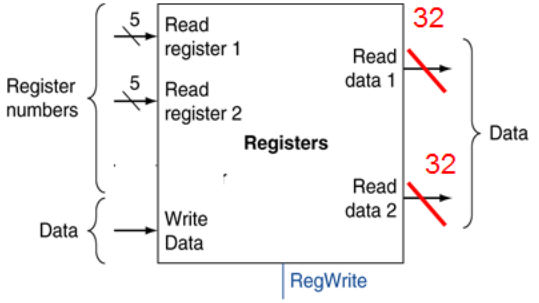
24_{10} + $s2 =

. . . 0001 1000
+. . . 1001 0100
. . . 1010 1100 =
0x120040ac

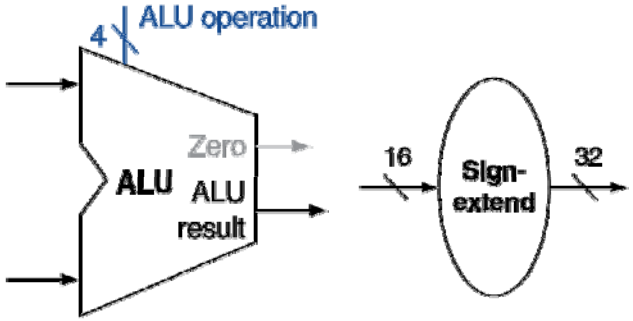


Load/Store Instructions (need 4 components)

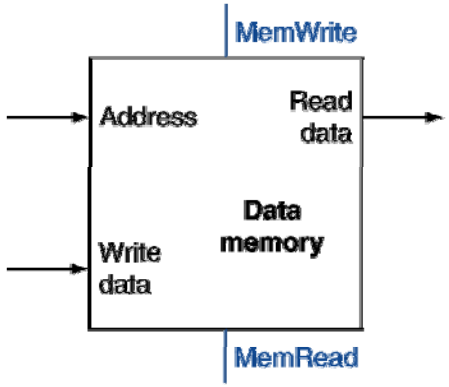
- Read register operands =>register files



- Calculate address using 16-bit offset
  - Use ALU, but sign-extend offset



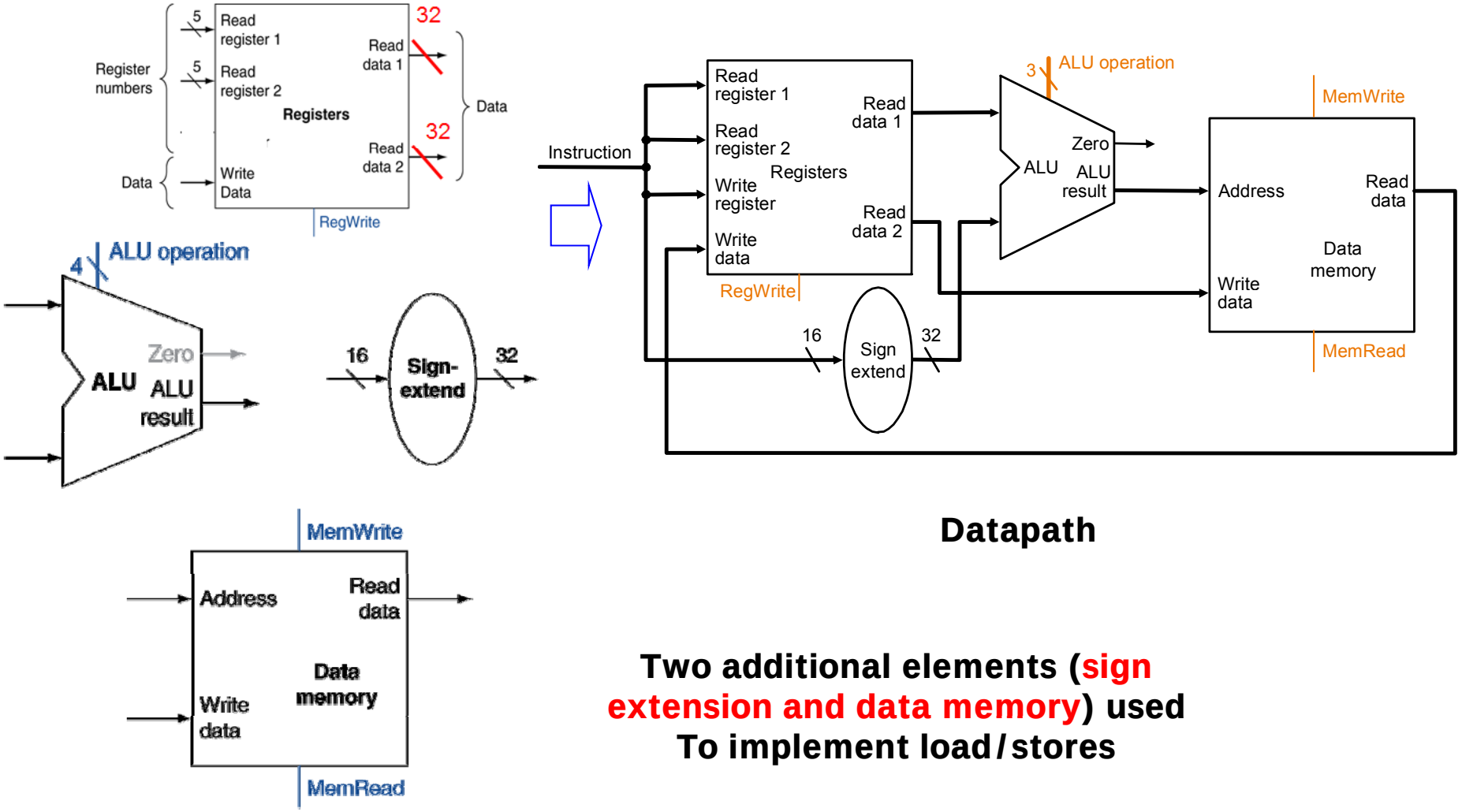
- Load/store: read memory and update register, and write register value to memory
  - Need data memory




```
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

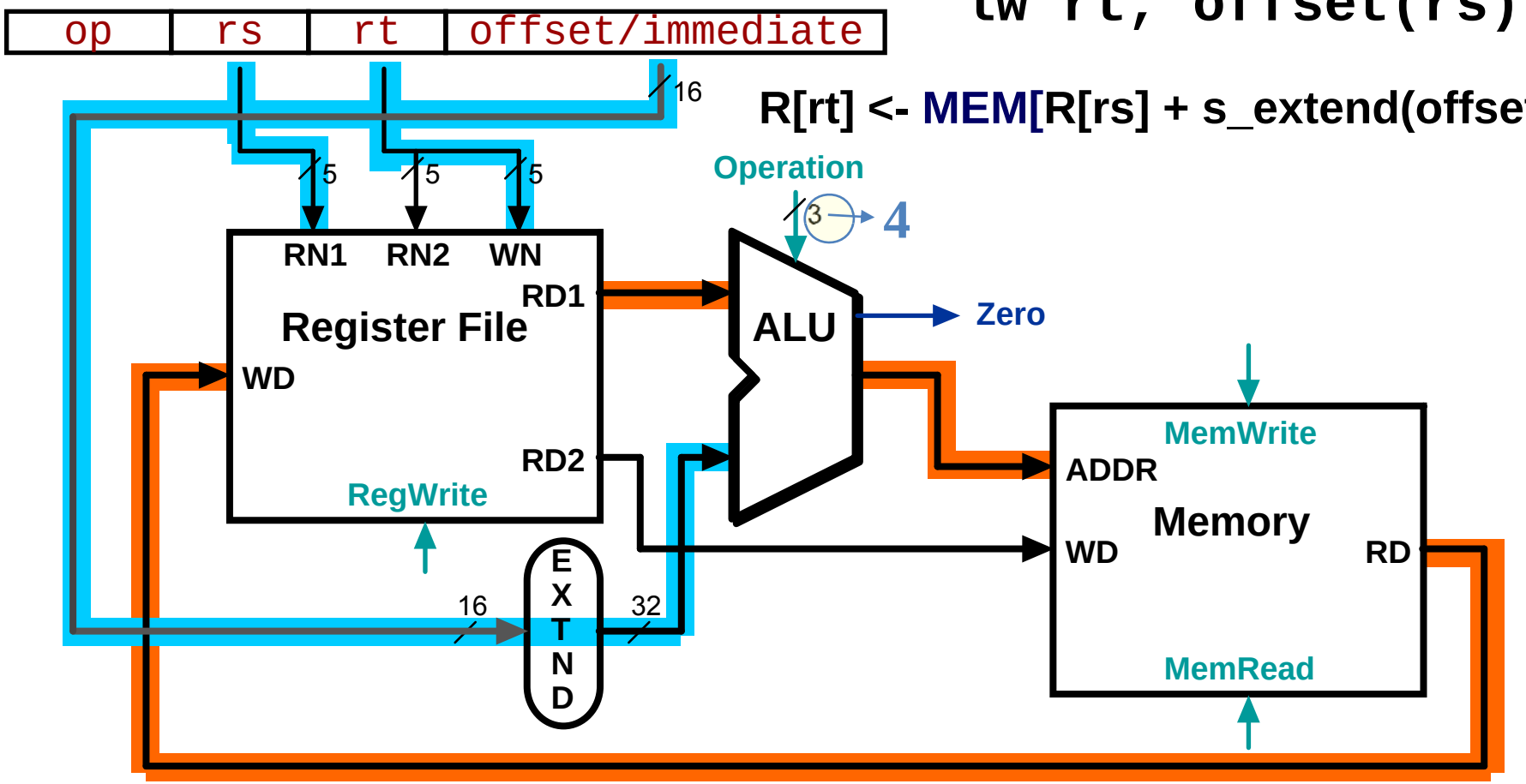
- Load

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)
```

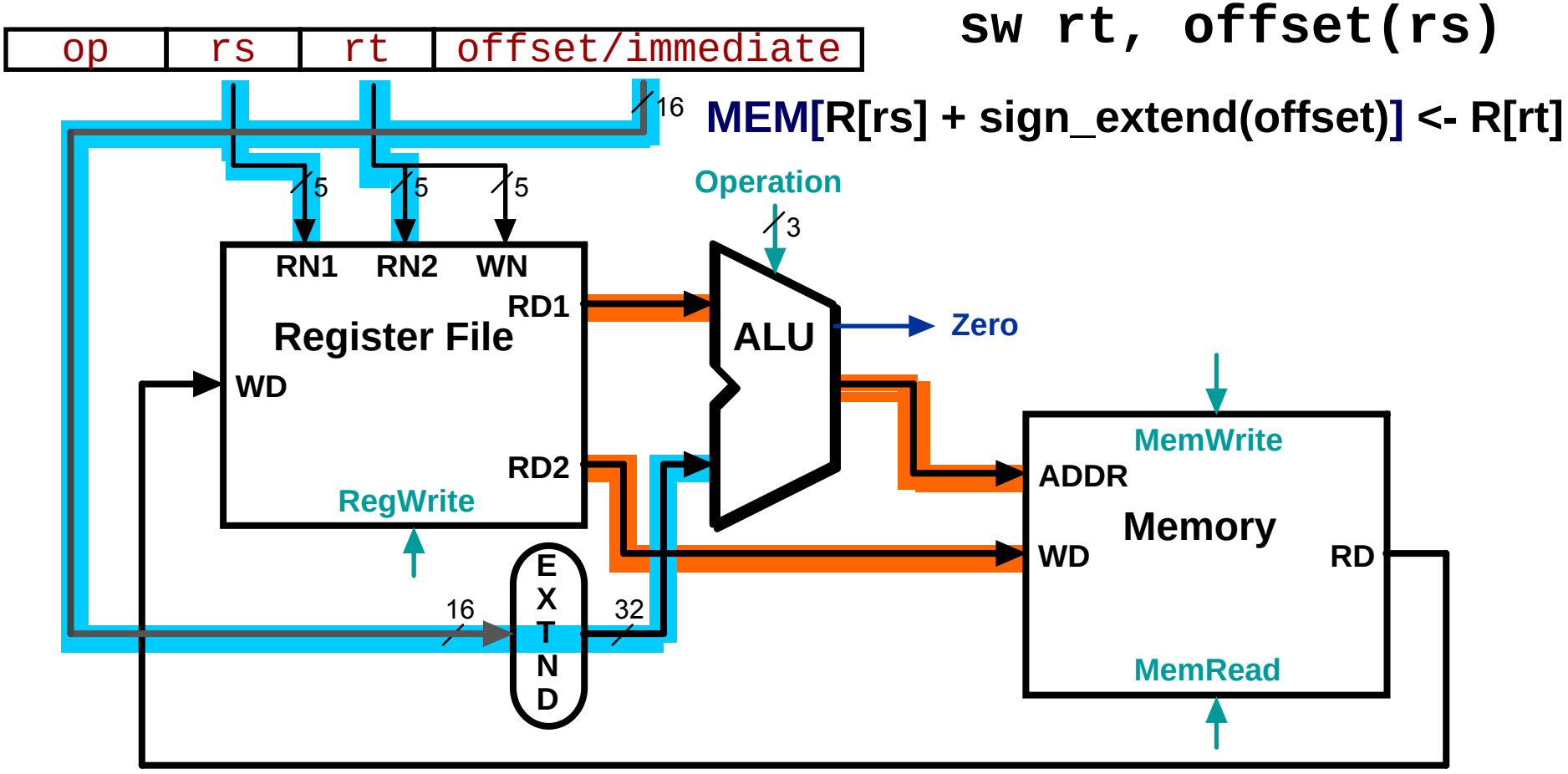
R[rt] <- MEM[R[rs] + s_extend(offset)];



Animating the Datapath- store

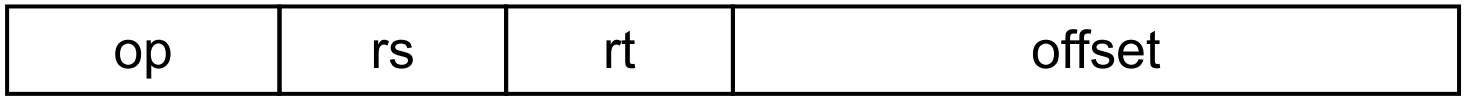
- store

sw $t0, 8($s3)



Specifying Branch Destinations

- MIPS conditional branch instructions:

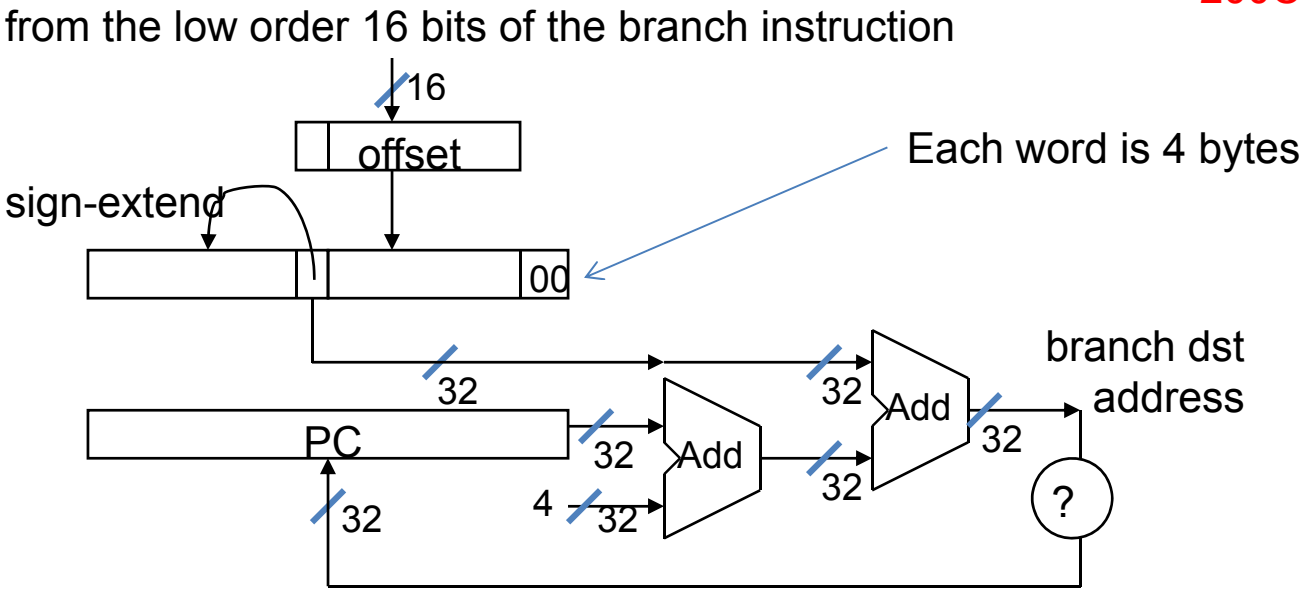


- PC-relative addressing
  - Target address = PC + offset × 4
  - PC already incremented by 4 by this time

```
2000    beq $s0 $t1 2
2004    ….
2008    …
200C
```

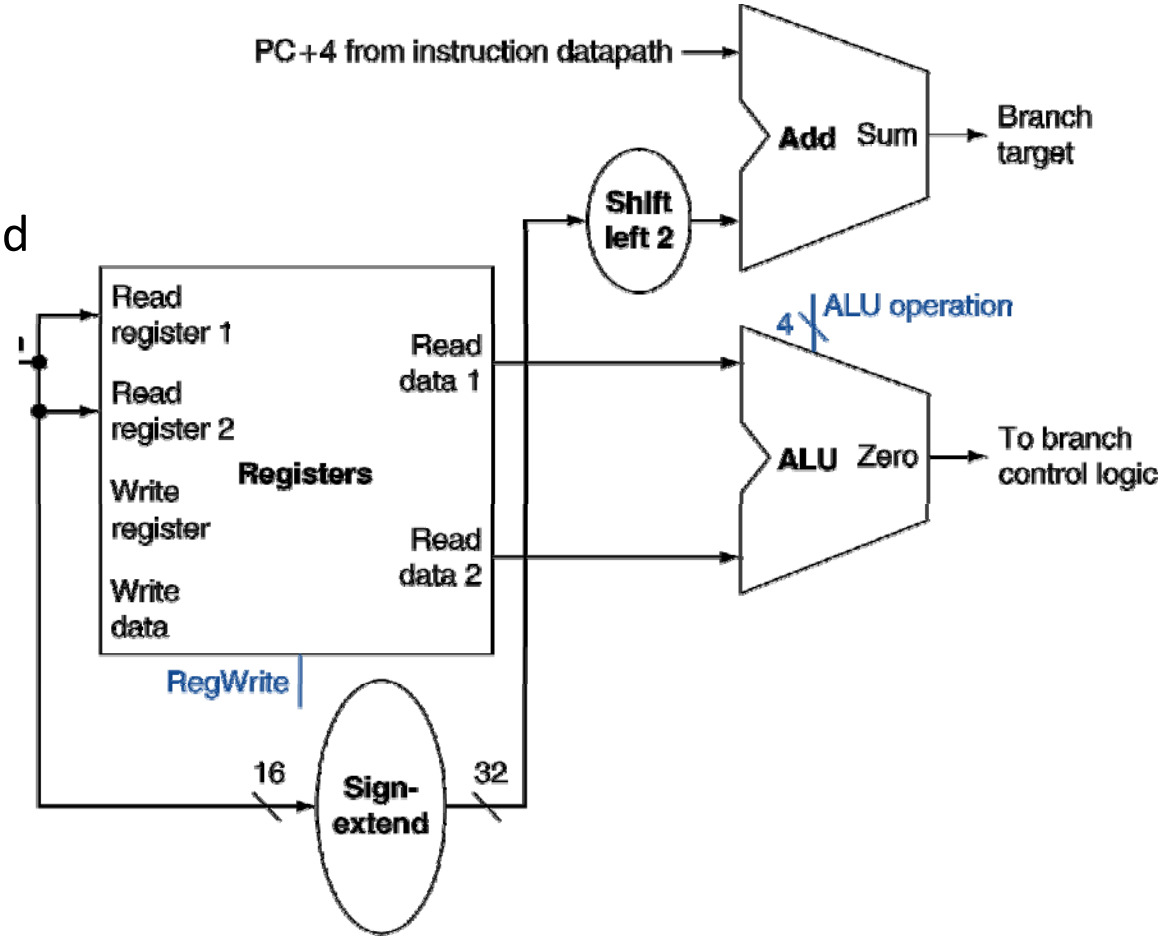
Target Address (address of next instruction) =?

200C



Branch Instructions

- Read register operands
- Compare operands
  - Use ALU, subtract/xor 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)

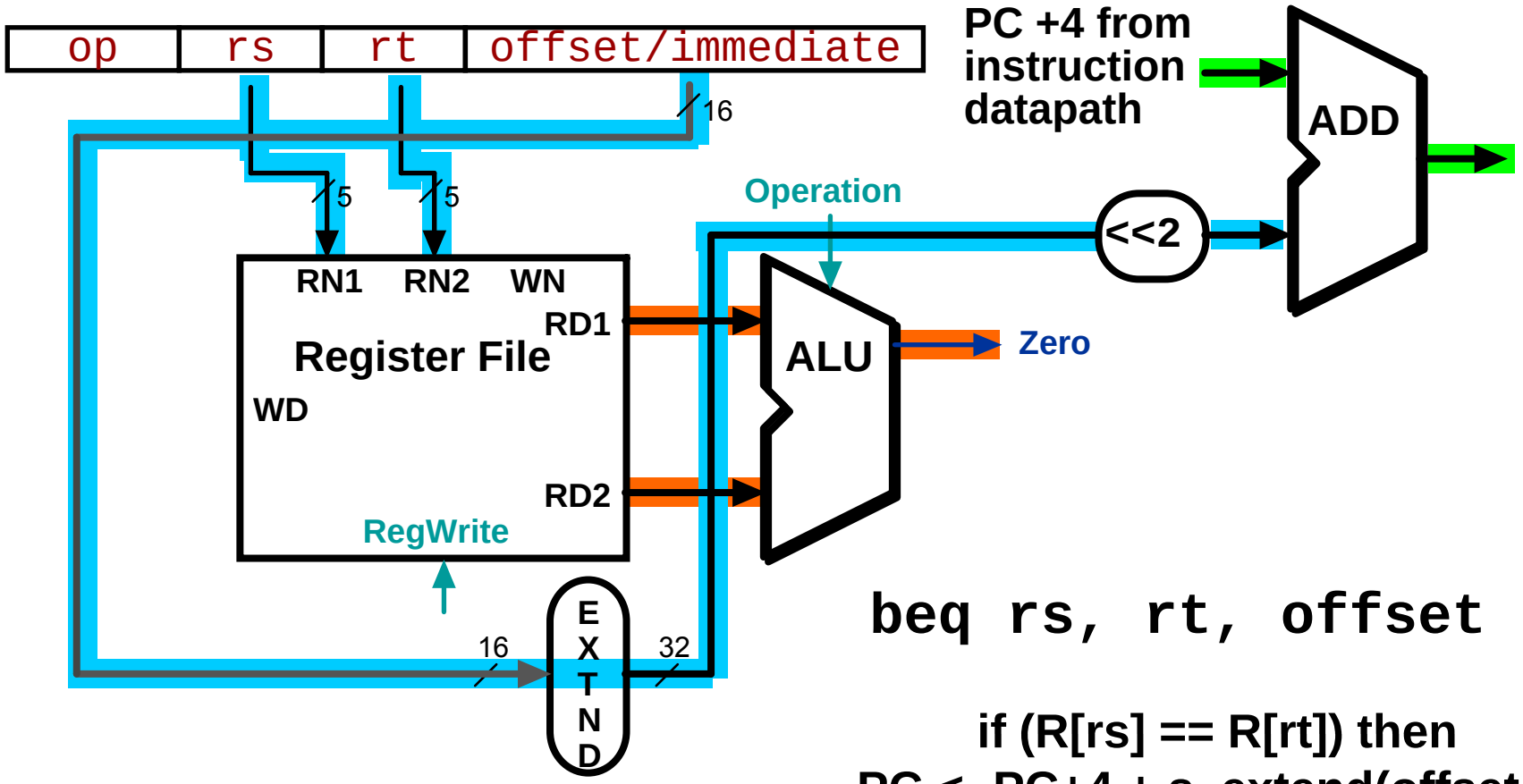


See animation in the next slide

Animating the Datapath (beq)

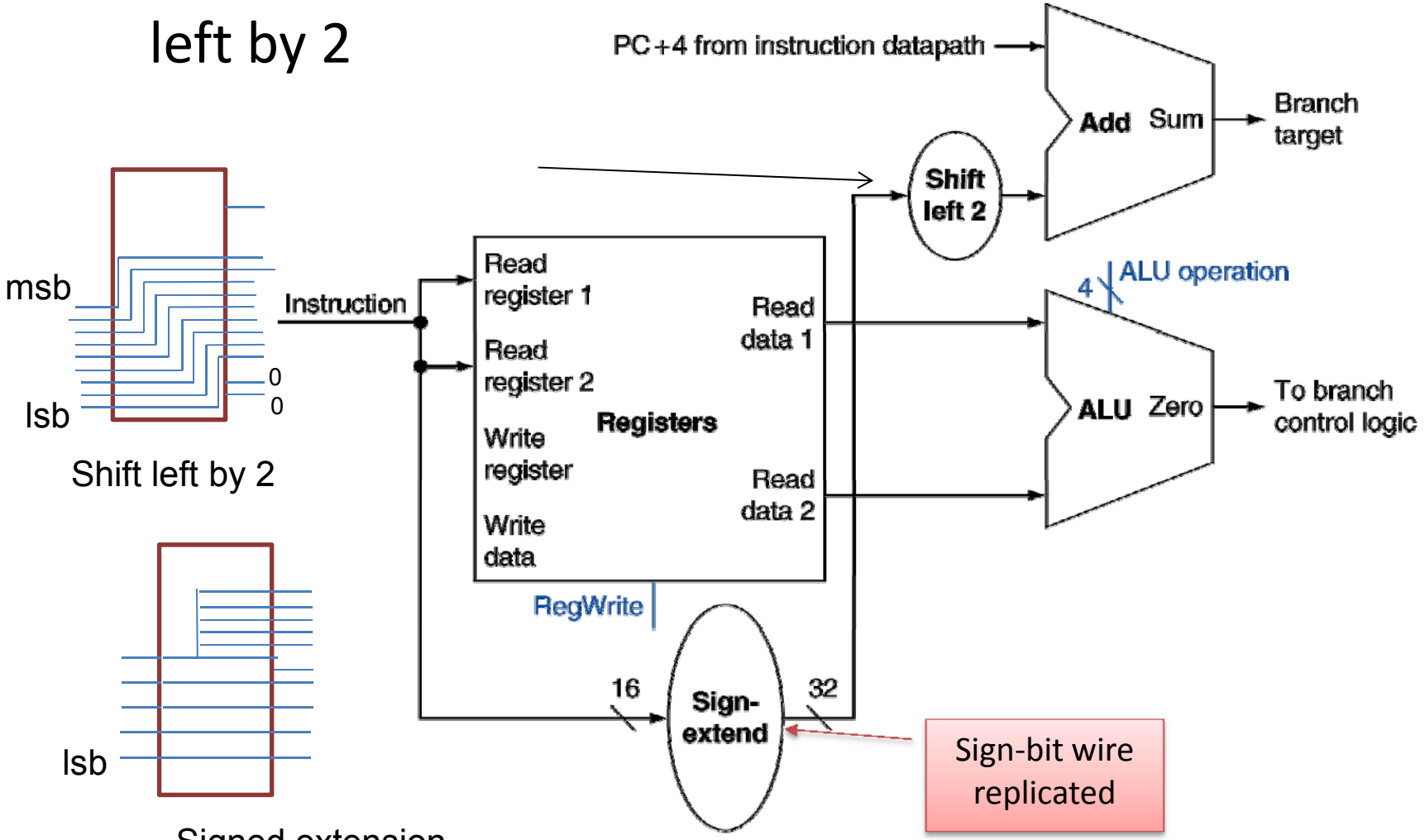
- beq

  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

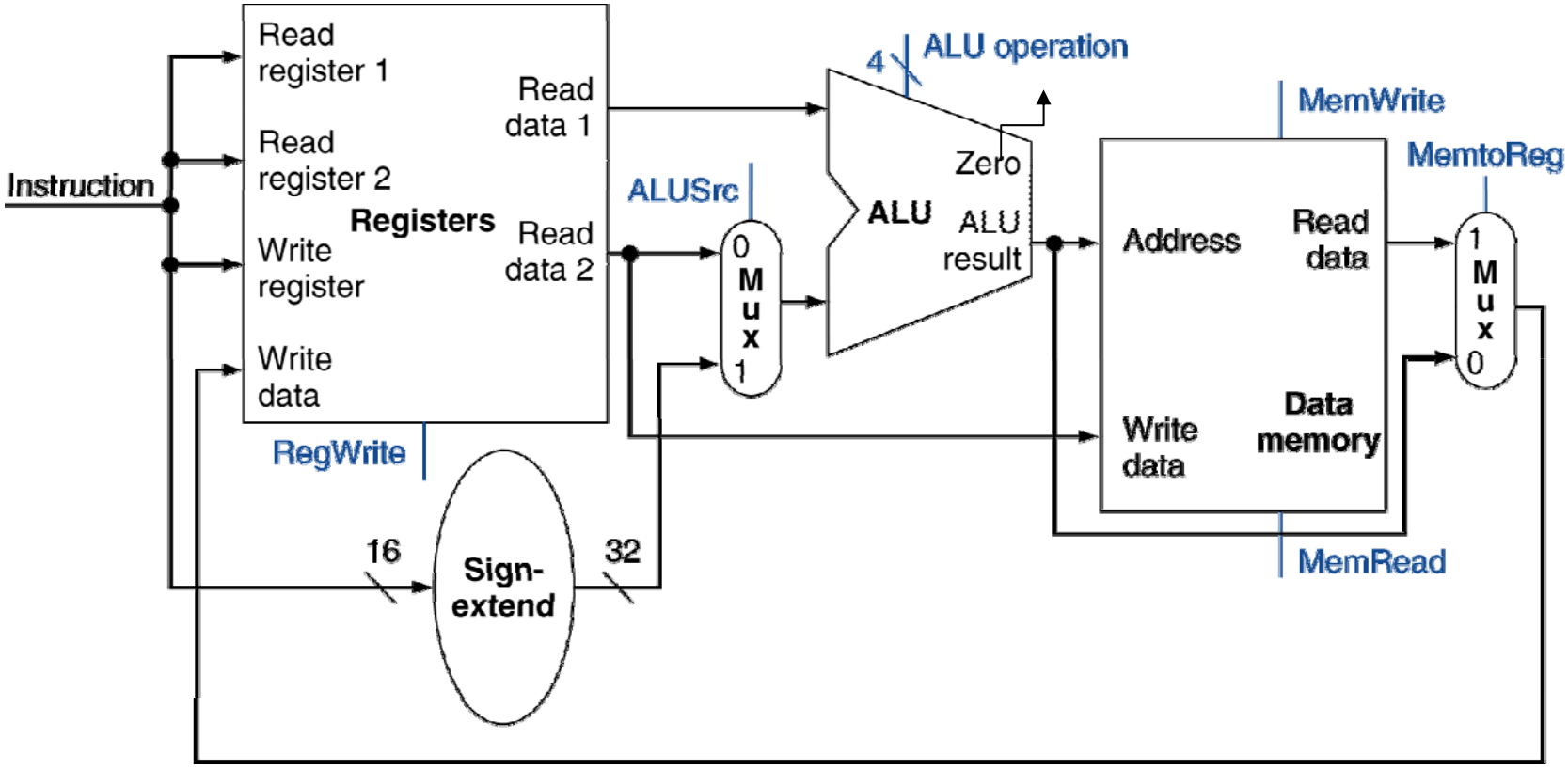


Composing the Elements

- 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

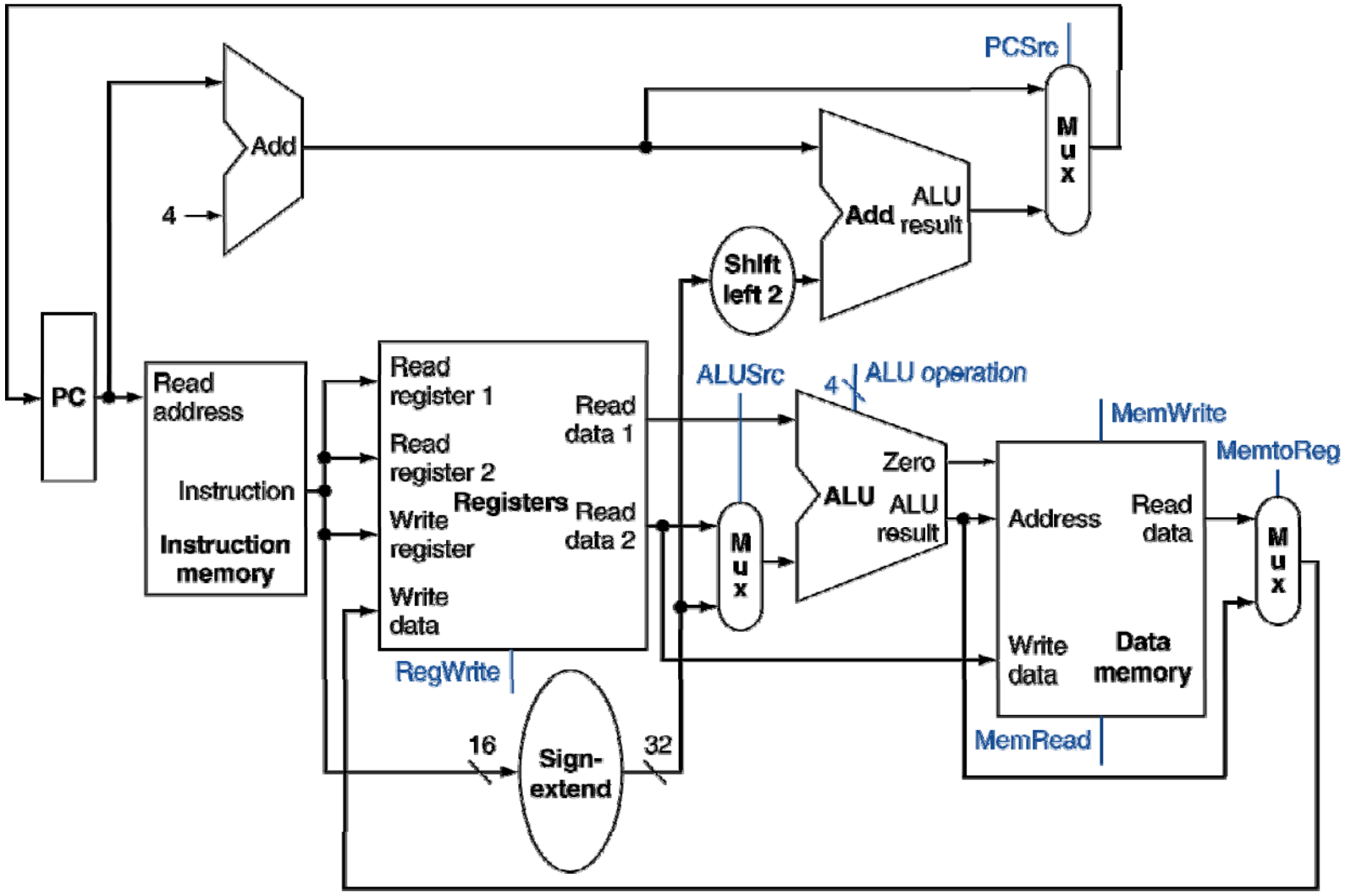
R-Type/Load/Store Datapath

A Single Cycle Datapath



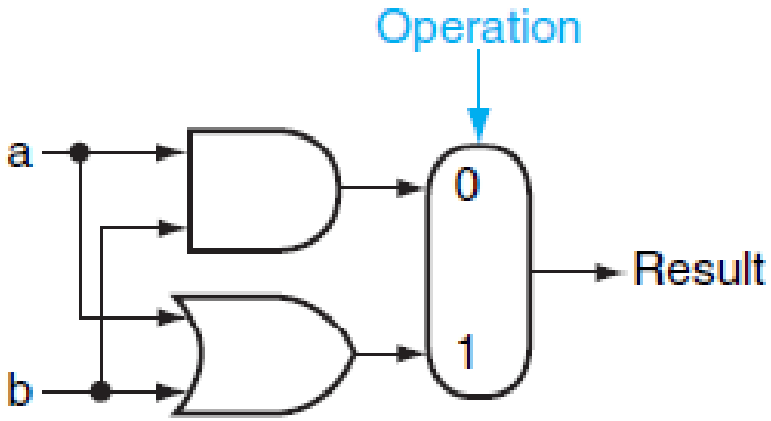
Correct Control signal (RegWrite, ALUSrc, ALU operation, MemWrite, MemtoReg, MemRead) are needed to make sure correct operation is done

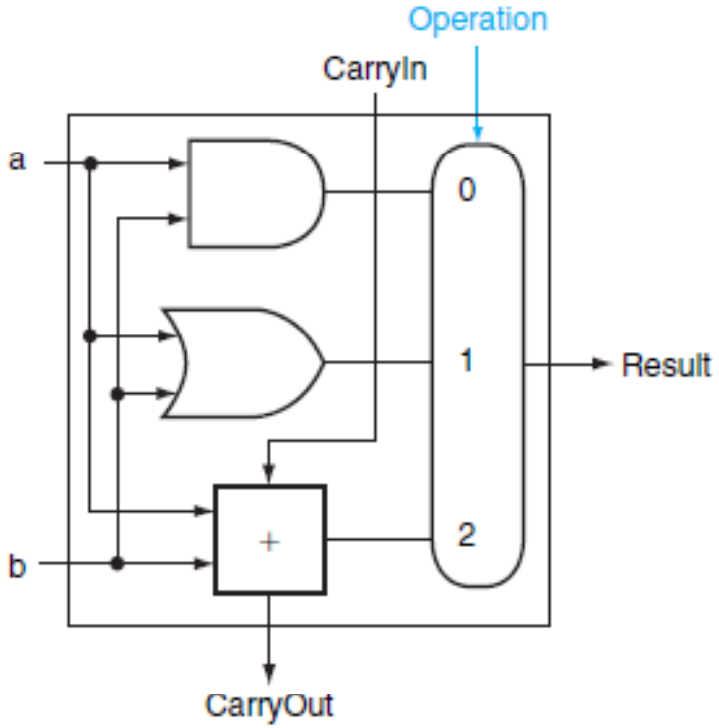
Full Datapath (Single Cycle Datapath)



Basic Arithmetic Logic Unit

- Basic ALU





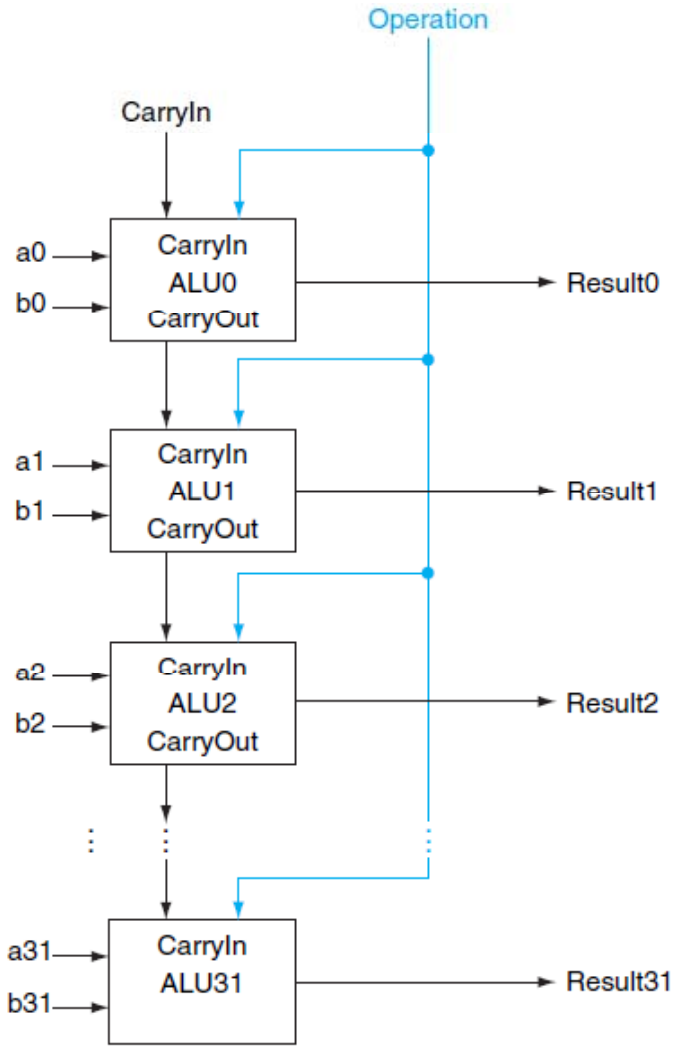
One-bit ALU that performs AND and OR

Operation(Op.)	Funct.
0	a AND b
1	a OR b

Operation(Op.)	Funct.
0	a AND b
1	a OR b
2	a + b

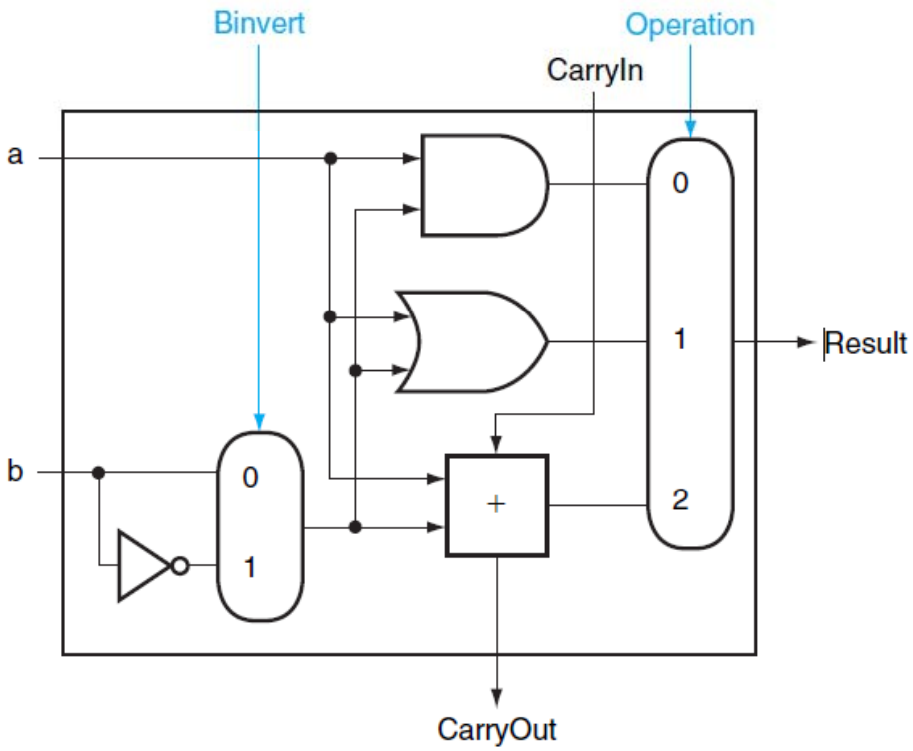
32-bit ALU

- Cascading 1-bit ALU to 32-bit ALU
- carry-out is the carry-in of the next bit



Enhanced Arithmetic Logic Unit

- ALU that performs (a AND b), (a OR b) and (a + b) and (a- b=a + $\bar{b}$+1)
- =>Binvert=1 and carryIn=1

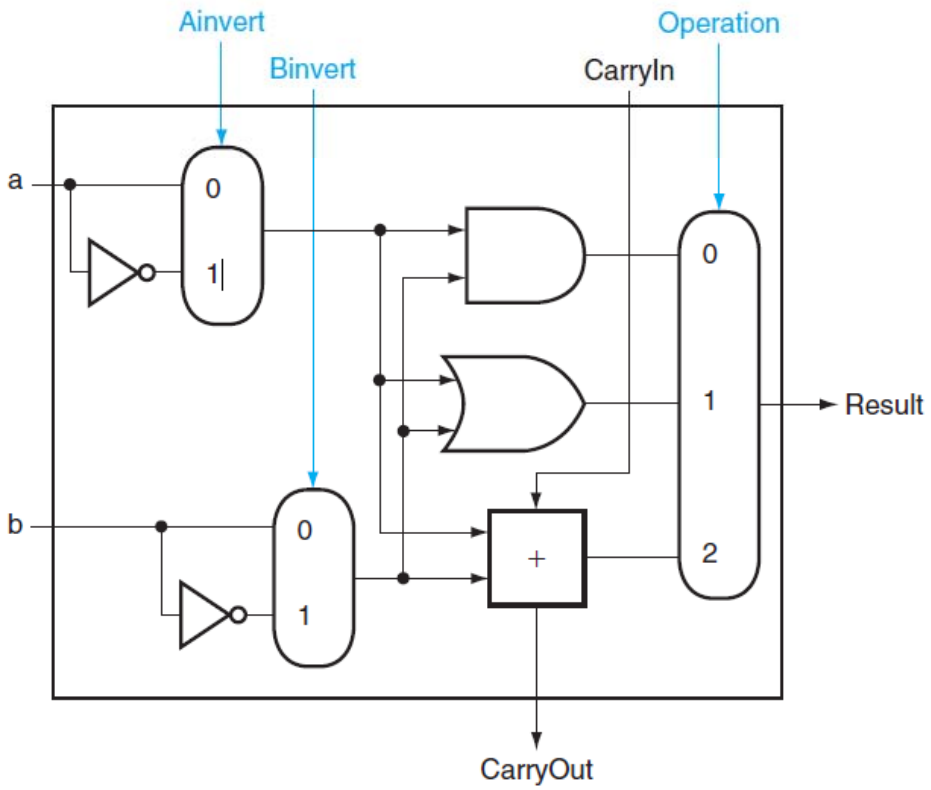


$a-b = a+\bar{b}+1$

Binvert	CarryIn	Op.	Function
0	X	0	a and b
0	X	1	a or b
0	0	2	a + b
1	1	2	a-b

Enhanced Arithmetic Logic Unit

- Enhanced with NOR and NAND



$$\bar{a}\,V\,\bar{b} = \overline{ab}$$

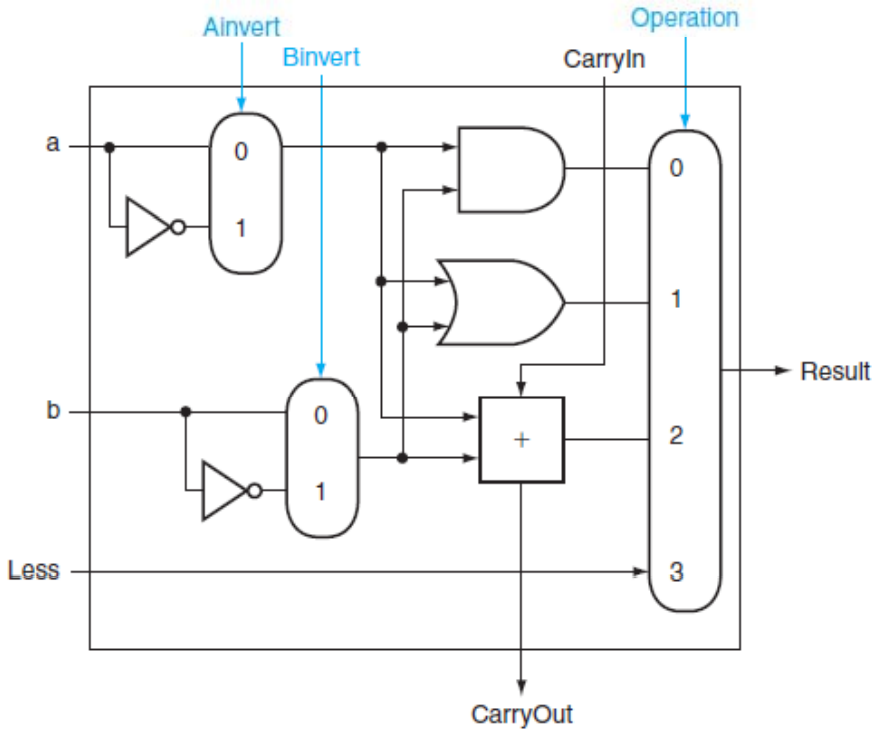
$$\bar{a}\bar{b} = \overline{a\,V\,b}$$

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
1	1	X	0	$\overline{a+b}$
1	1	X	1	$\overline{ab}$

ALUs with Set Less Than

Review: Set less than
slt $t0 $t1 $t2 =>When $t1 < $t2 , $t0 =1, otherwise $t0=0

- We use a-b to implement slt
  - When a-b < 0, signed bit =1
  - When a-b >=0, signed bit =0
- Less signal=>
  - Connect LSB to the signed bit of MSB (See next slide)
  - Other signals are assigned to 0

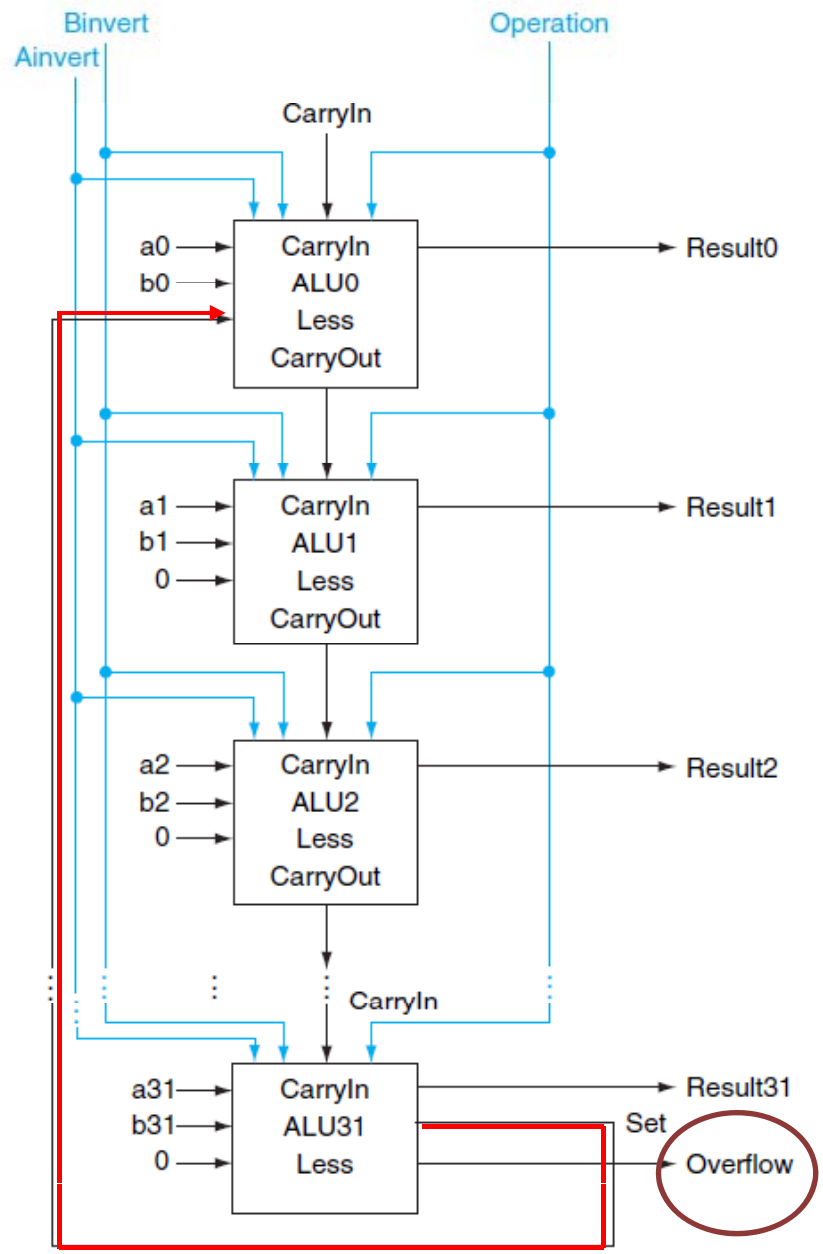


32-bit ALU with Set Less than

- Less signal=>
  - Connect LSB to the signed bit of MSB
  - Other signals are assigned to 0

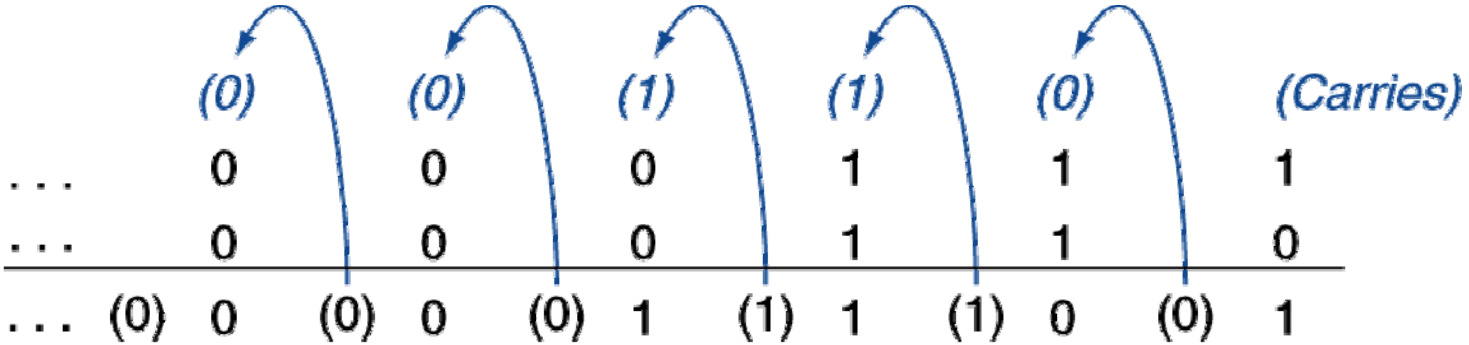
When a31…a0 < b31….b0 , result is 0…….1, otherwise 0……0

Note that MSB is different than other bits=> it has one additional signal (Overflow) which will be discussed later



Integer Addition and Subtraction

- Addition Example: 7 + 6



- Subtraction Example: 7-6 = 7+(-6)

```
  00000111
– 00000110
  00000001
```

```
+7: 0000 0000 … 0000 0111
+–6: 1111 1111 … 1111 1010
+1:  0000 0000 … 0000 0001
```

Situations when overflow occurs

- Situation that overflow occurs for signed integers

Operation	A	B	Result when Overflow
A+B	A>=0	B>=0	<0
A+B	<0	<0	>=0
A-B	A>=0	B<0	<0
A-B	A<0	B>=0	>=0

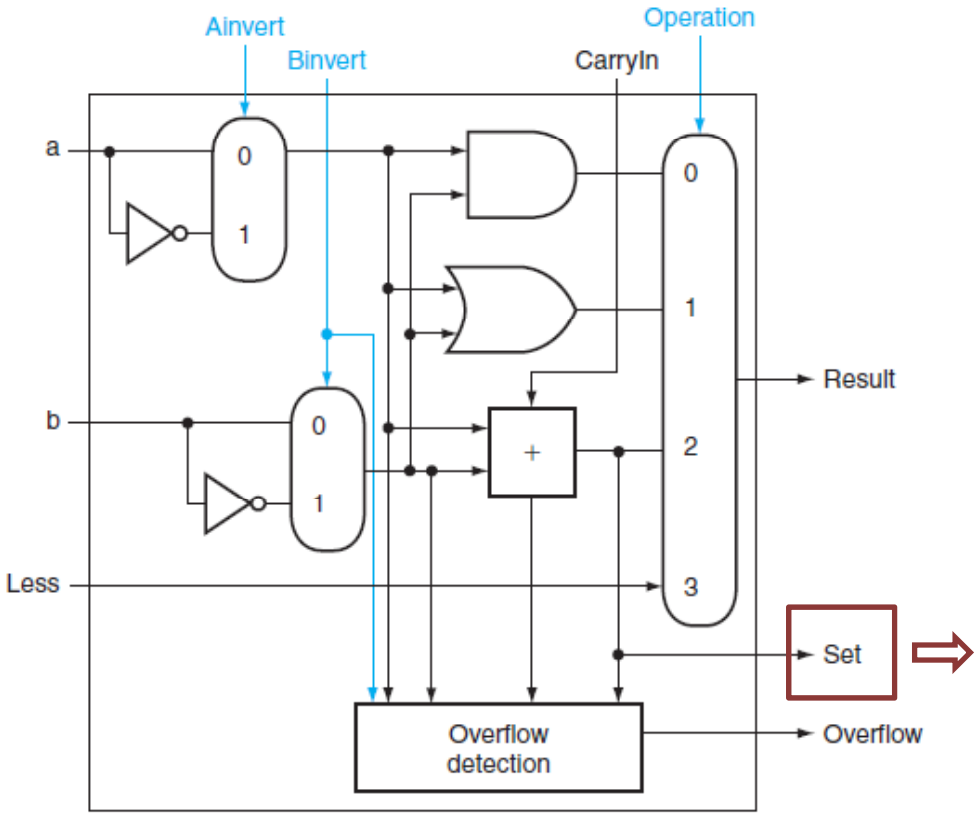
$$\begin{array}{r} 0111 \\ +0001 \\ \hline 1000 \end{array}$$

7+1 ≠ -8

$$\begin{array}{r} 1111 \\ +1000 \\ \hline 0111 \end{array}$$

-1+(-8) ≠ 7

Detecting Overflow



- Overflow detection circuit requires
  - Binvert (to know a+b or a-b is executed)
  - Sign bits of a, b
    - to know if a or b is >= 0 or <0)
  - MSB of (a+b)
  - Carryout bit of (a+b)

for slt (used to set the LSB bit when a<b is true

$$\begin{array}{r} 1111 \\ +1000 \\ \hline 10111 \end{array}$$

Dealing with Overflow

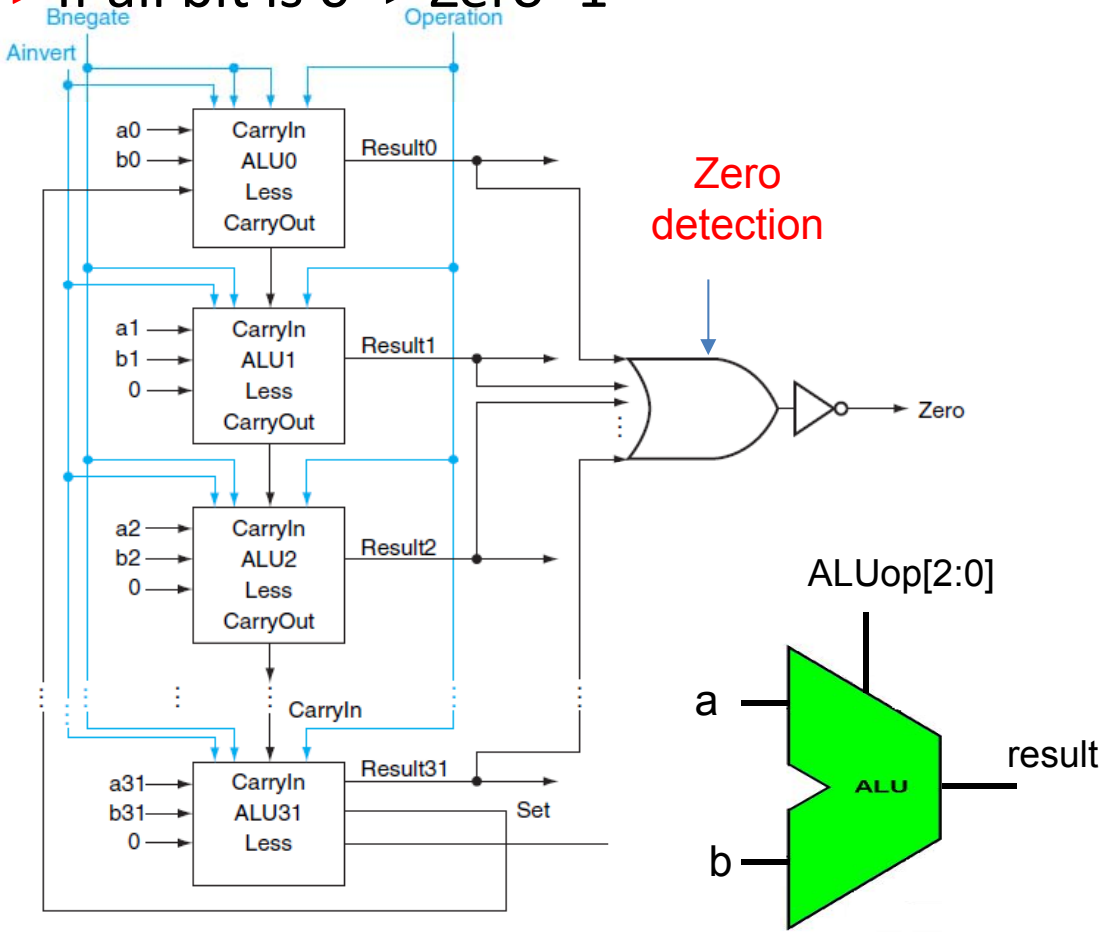
- Some languages (e.g., C) ignore overflow
- Other languages (e.g., Ada, Fortran) require raising an exception
- MIPS
  - addu, addui, subu instructions ignore overflow
  - add, addi, sub instructions may cause exceptions on overflow
  - So, C compiler on MIPS generates addu, addui, subu instructions
- MIPS exception handling on overflow
    - Save PC in exception program counter (EPC) register
    - Jump to predefined handler
    - `mfc0` (move from coprocessor reg) instruction can retrieve EPC value, to return after corrective action

Final 32-bit ALU

- Binvert is compatible to CarryIn according to the following table => 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

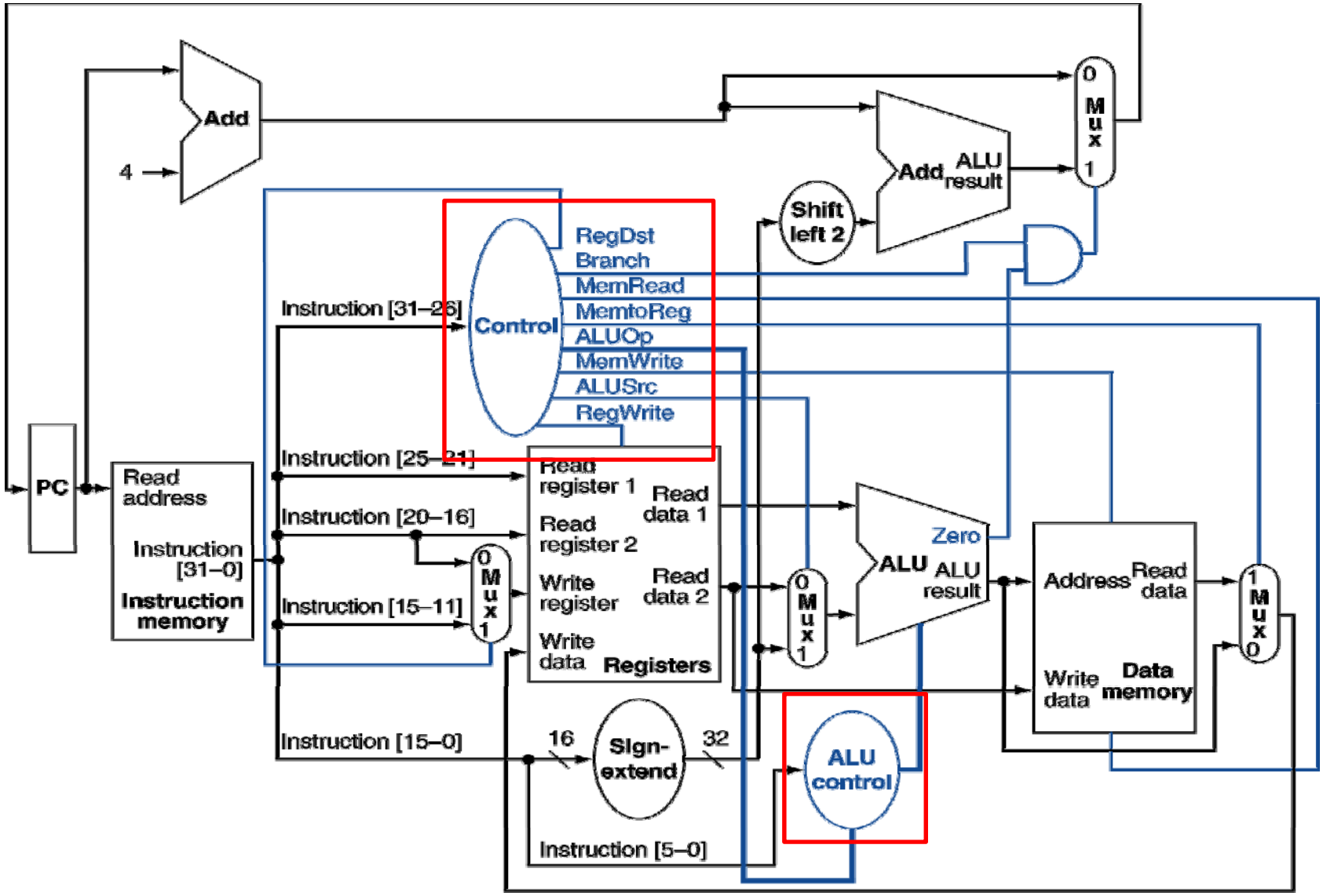
Bnegate	Op[1:0]	Func.
0	00	a and b
0	01	a or b
0	10	a + b
1	10	a - b
1	11	slt



Outline

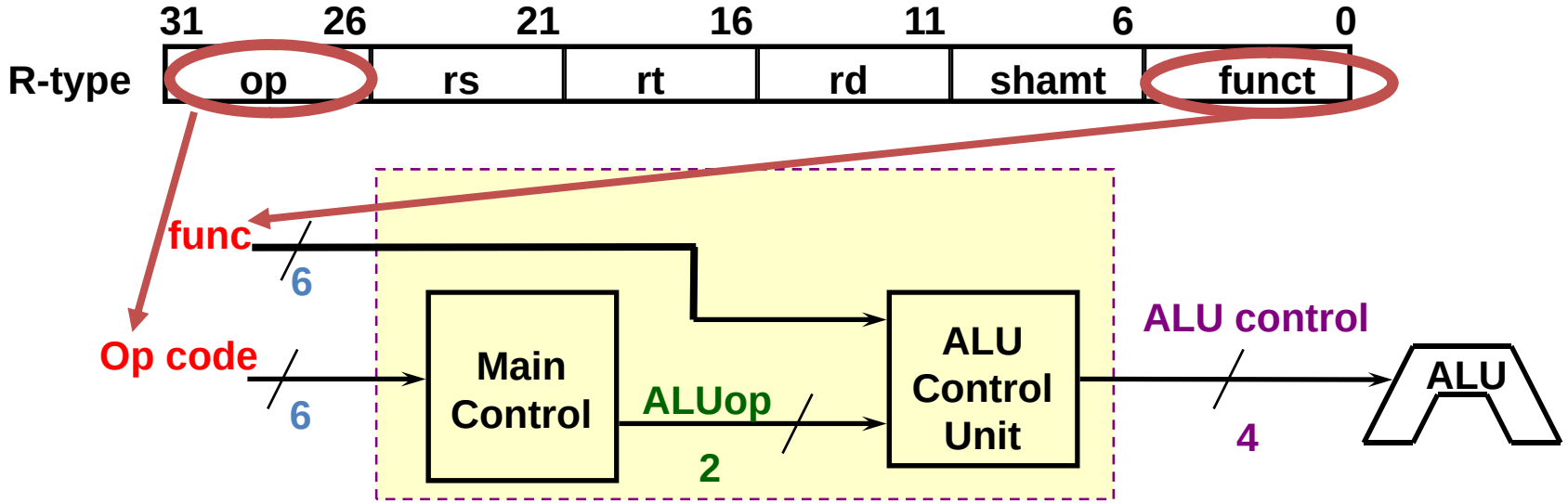
- Designing a processor
- A single-cycle implementation (the datapath)
- Control for the single-cycle CPU
  - Control of CPU operations
  - ALU control
  - Main control

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 funct field of instruction, the ALU control unit generates the 4-bit ALU control field



Deciding ALU Control

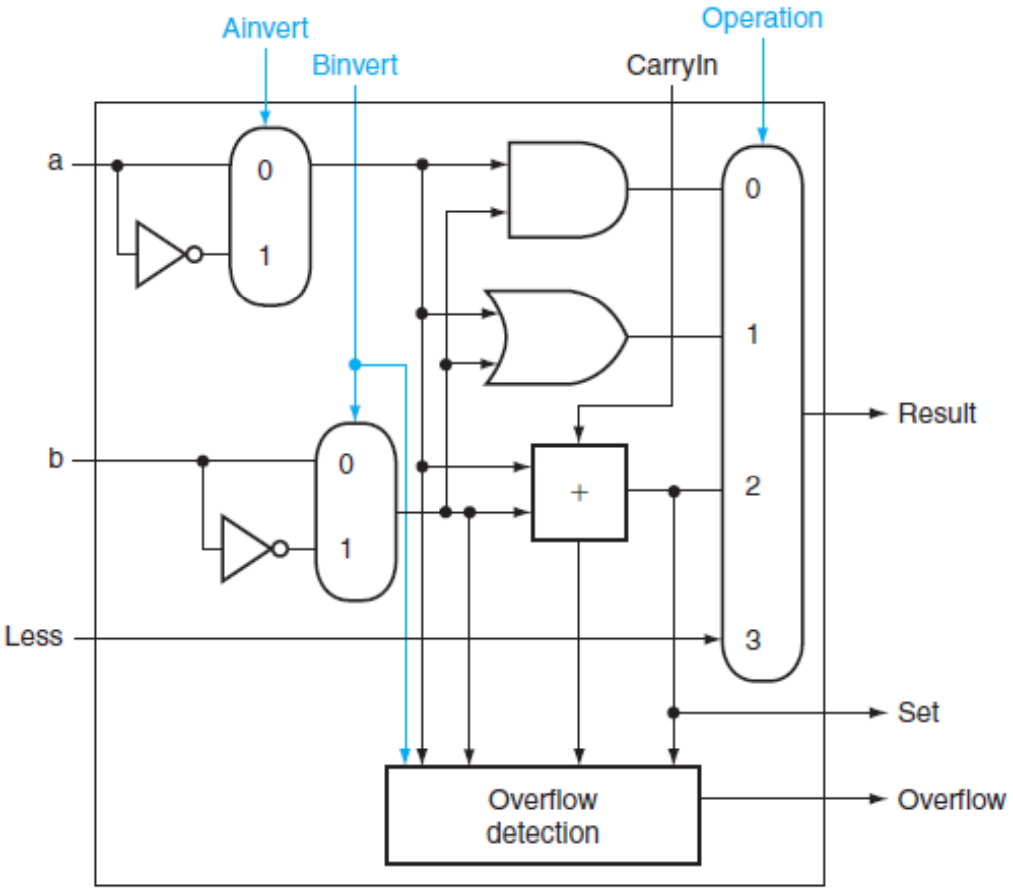
- Assume 2-bit ALUOp derived from opcode
  - Combinational logic derives ALU control

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



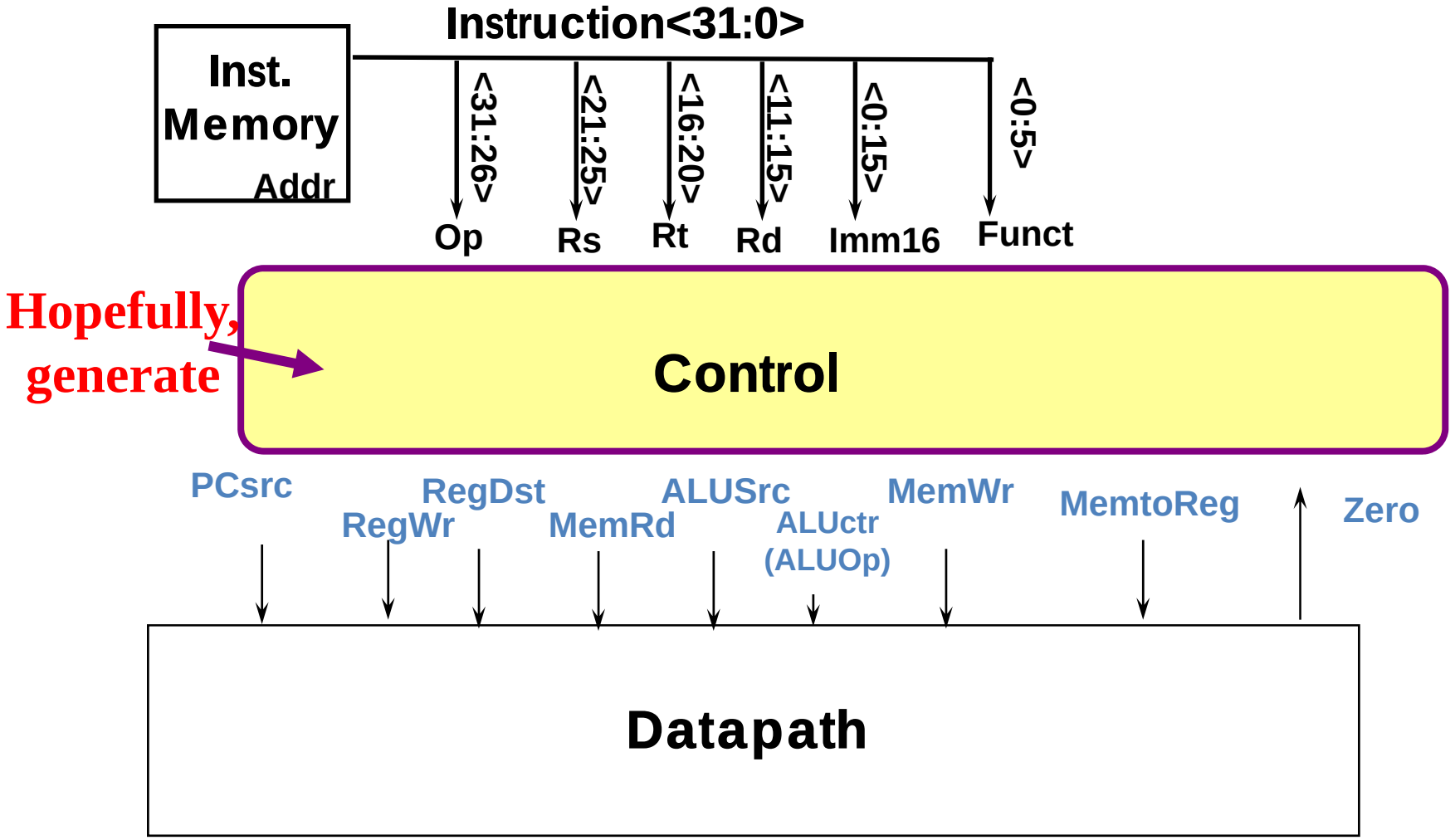
ALU Control

- 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	0010
sw	00	store word	XXXXXX	add	0010
beq	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
		subtract	100010	subtract	0110
		AND	100100	AND	0000
		OR	100101	OR	0001
		set-on-less-than	101010	set-on-less-than	0111

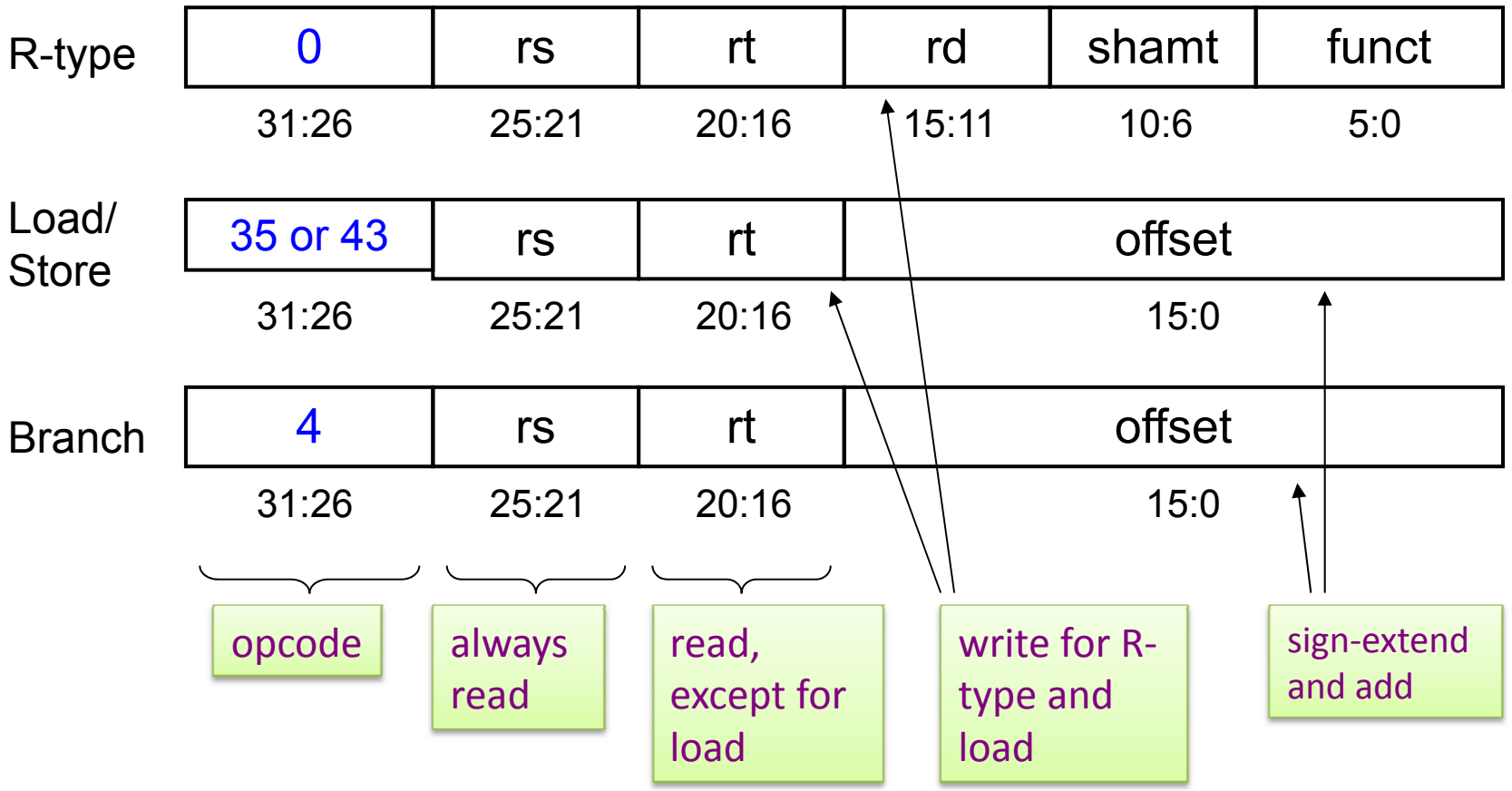
Deciding Main Control Signals

- Control l signal

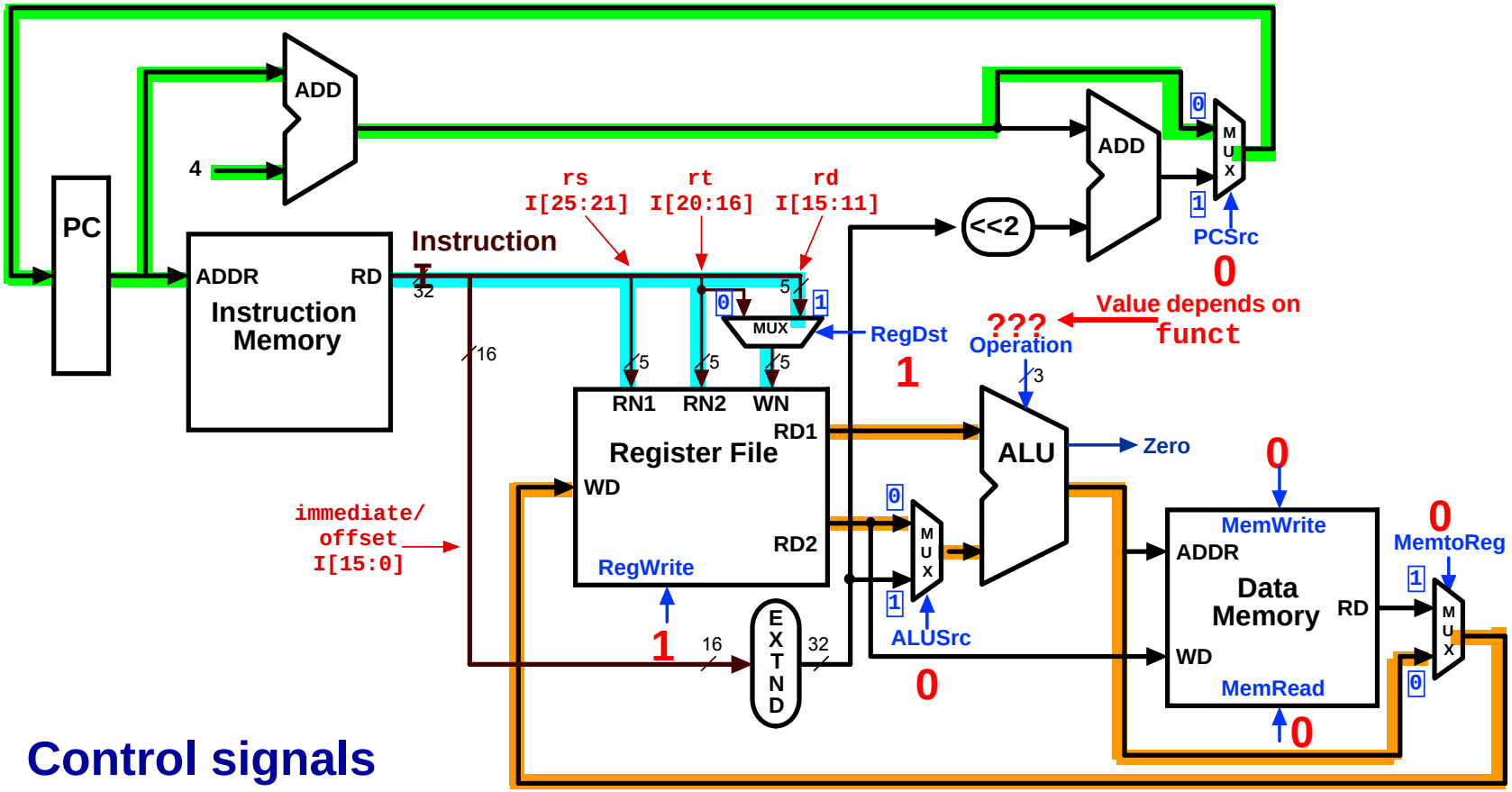


Review: The Main Control Unit

- Control signals derived from instruction

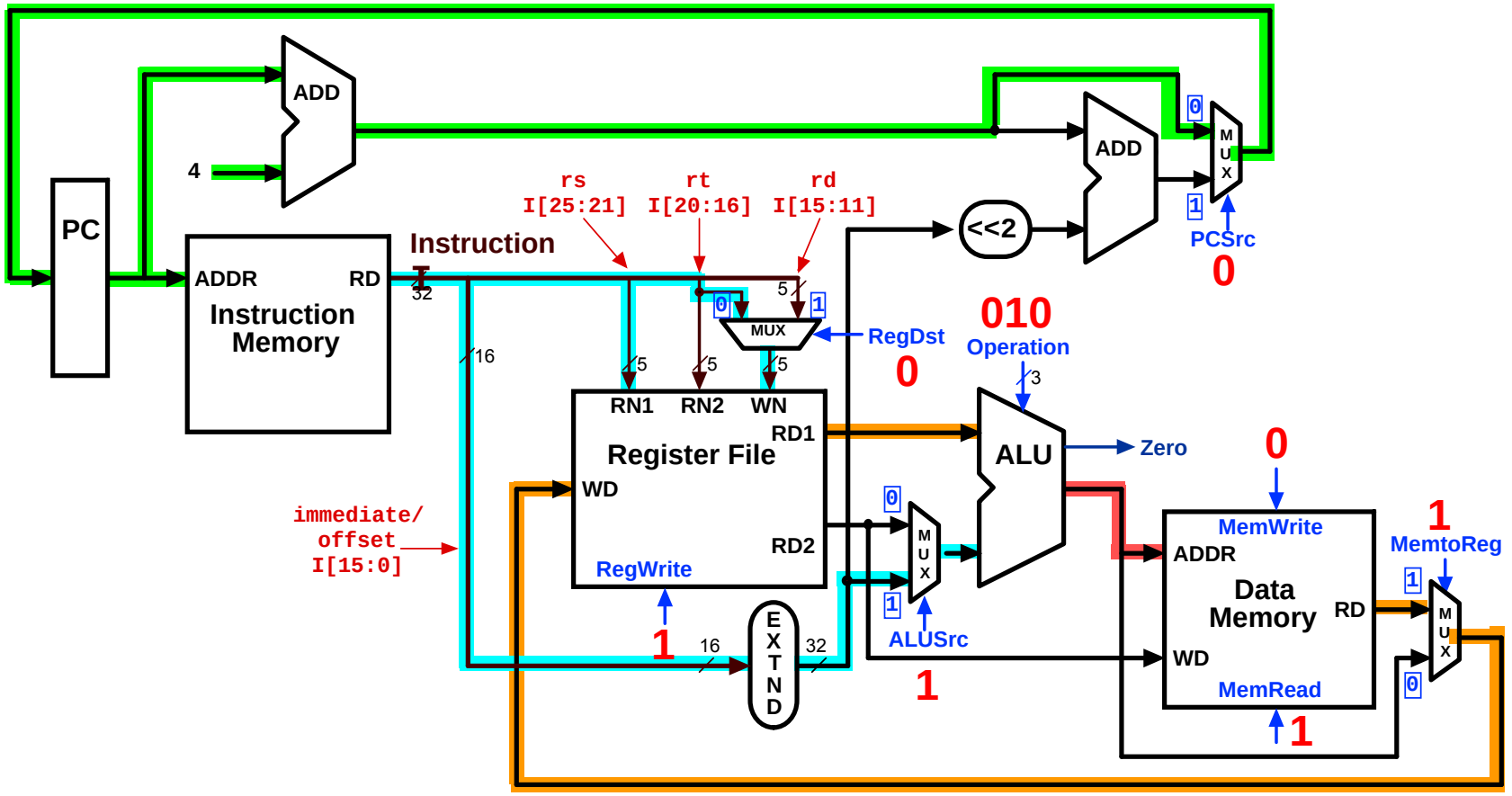


Control Signals for R-Type Instruction

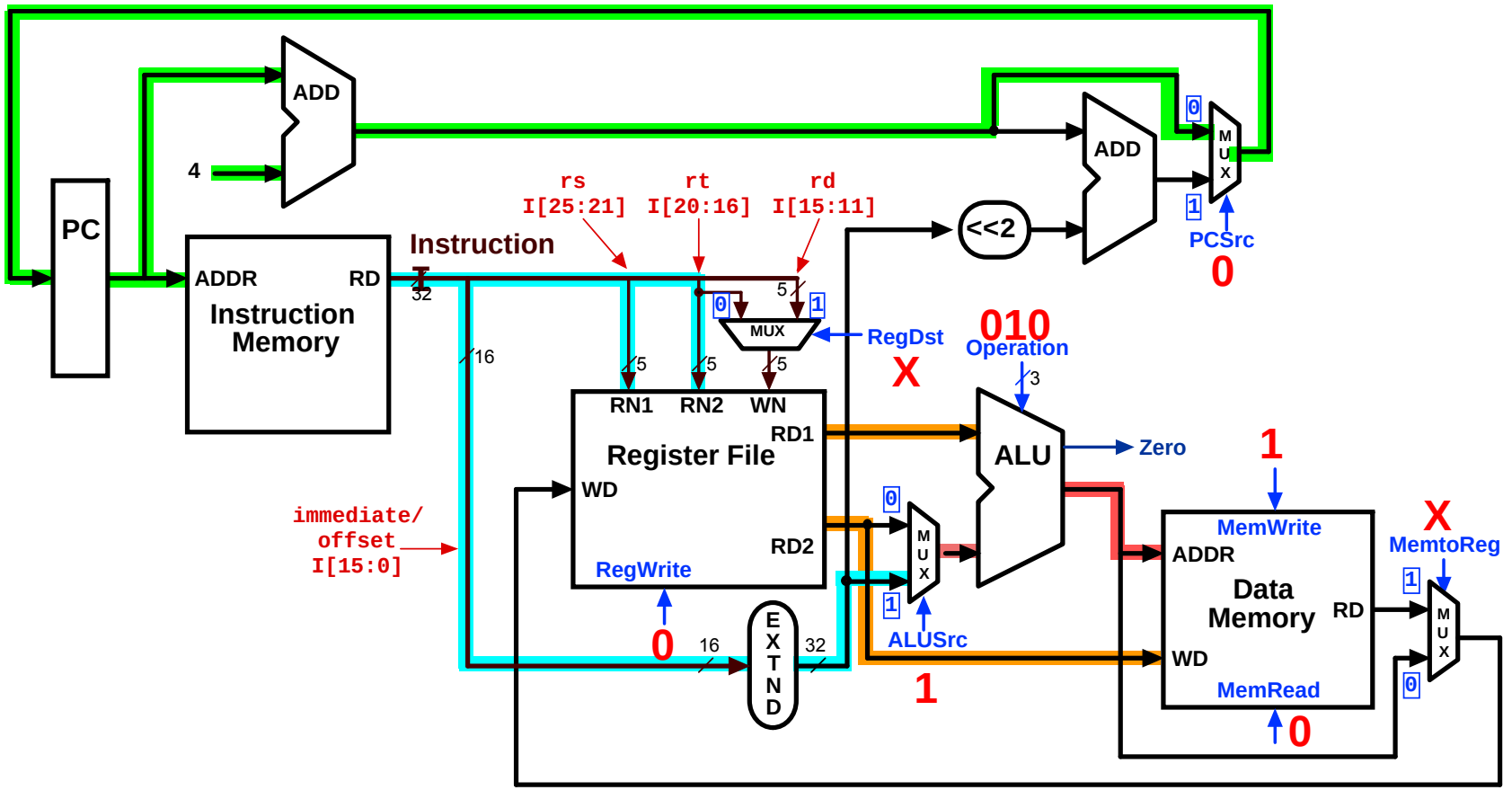


Control signals shown in blue

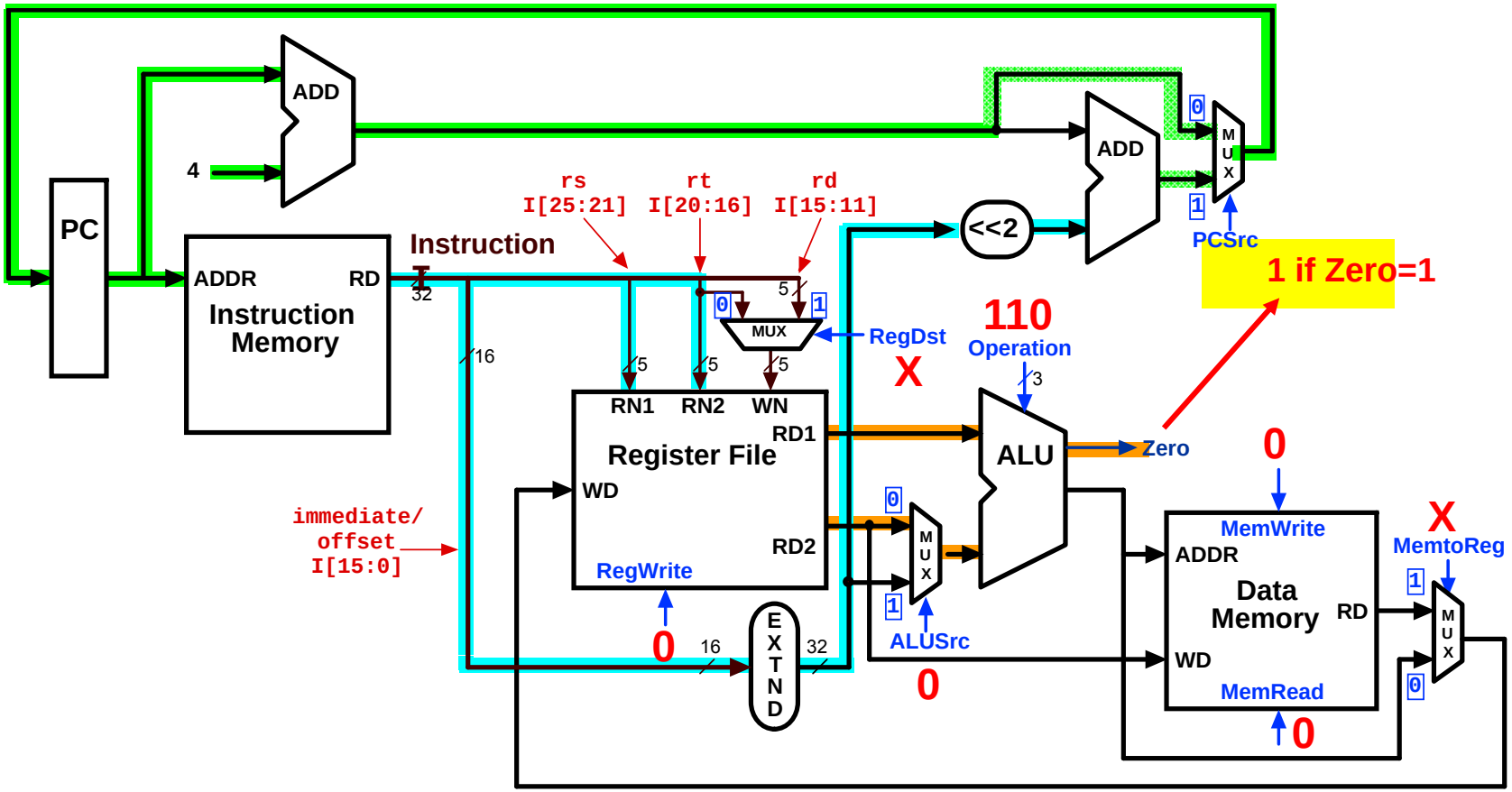
Control Signals: lw Instruction



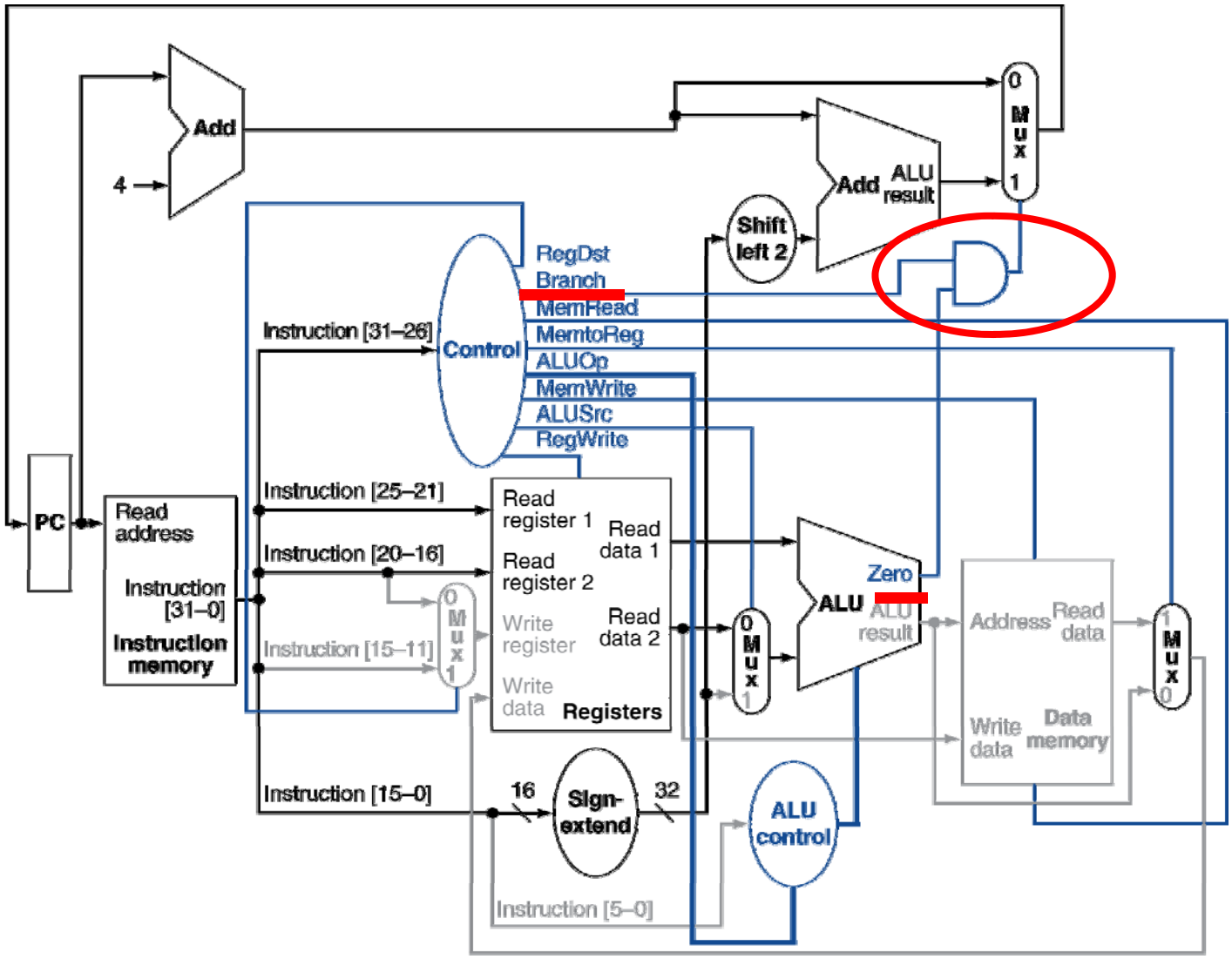
Control Signals: sw Instruction



Control Signals: beq Instruction

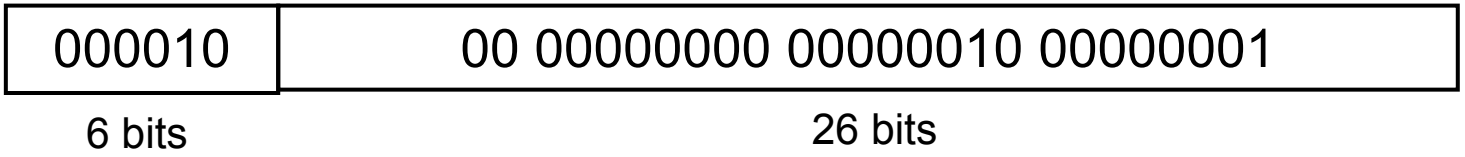


Branch-on-Equal Instruction



Review: Target address of Jump

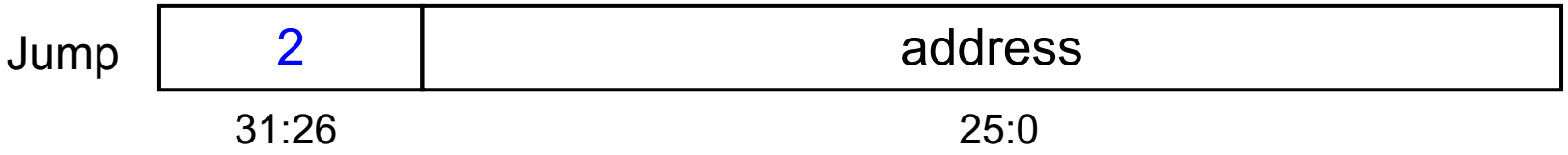
- Assume PC=40000000_{16}, what is the target address of the jump instruction?



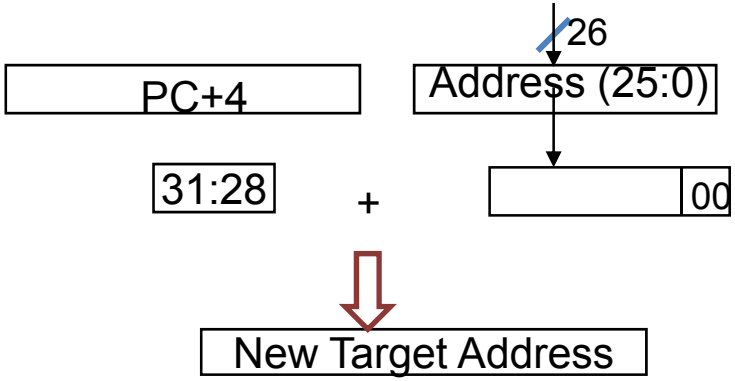
Address in the instruction= 0x0000201

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

Review: Implementing Jumps

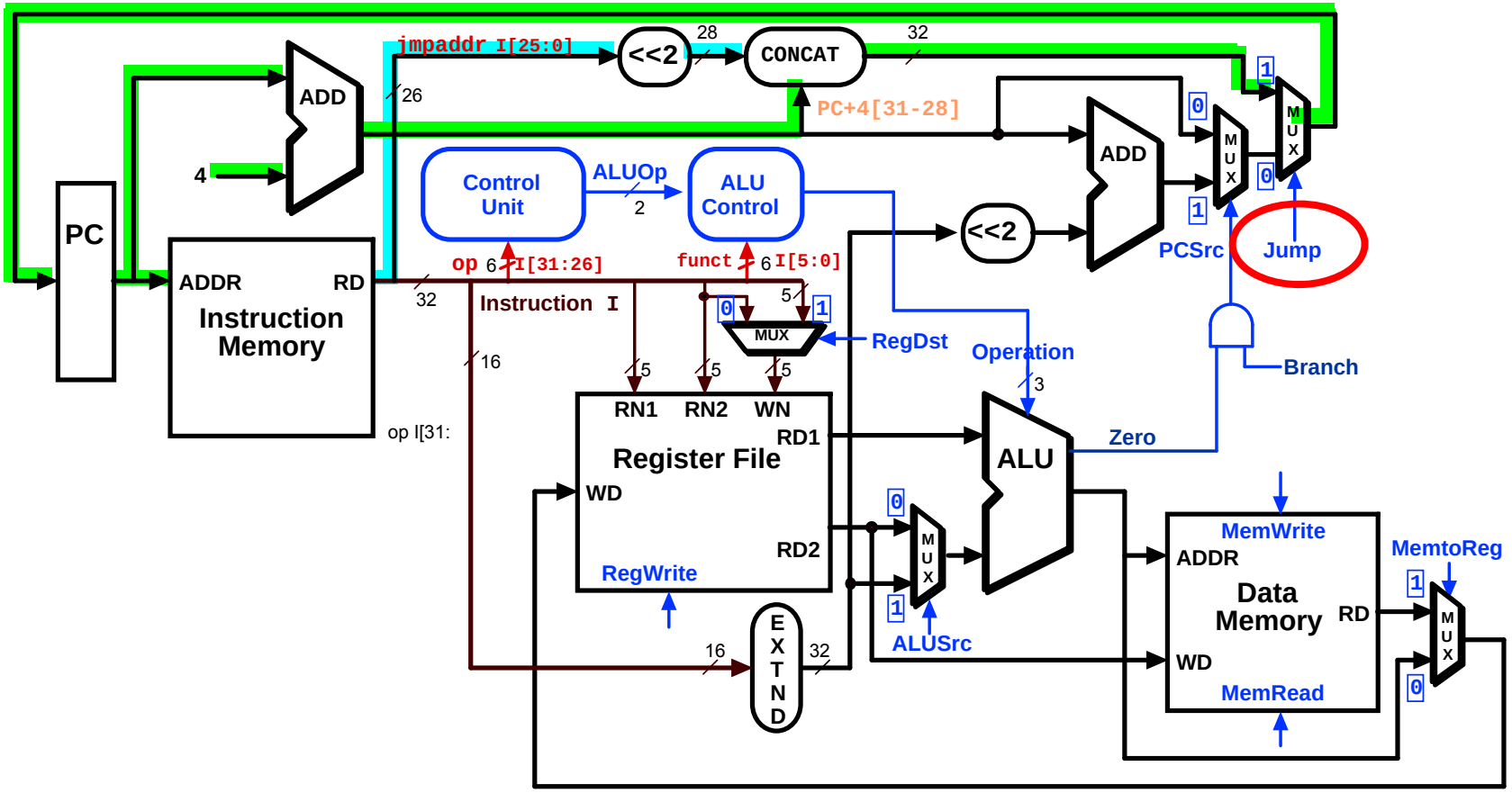


- Jump uses word address
- Update PC with concatenation of
  - Top 4 bits of old PC+4
  - 26-bit jump address
  - 00
- Need an extra control signal decoded from opcode

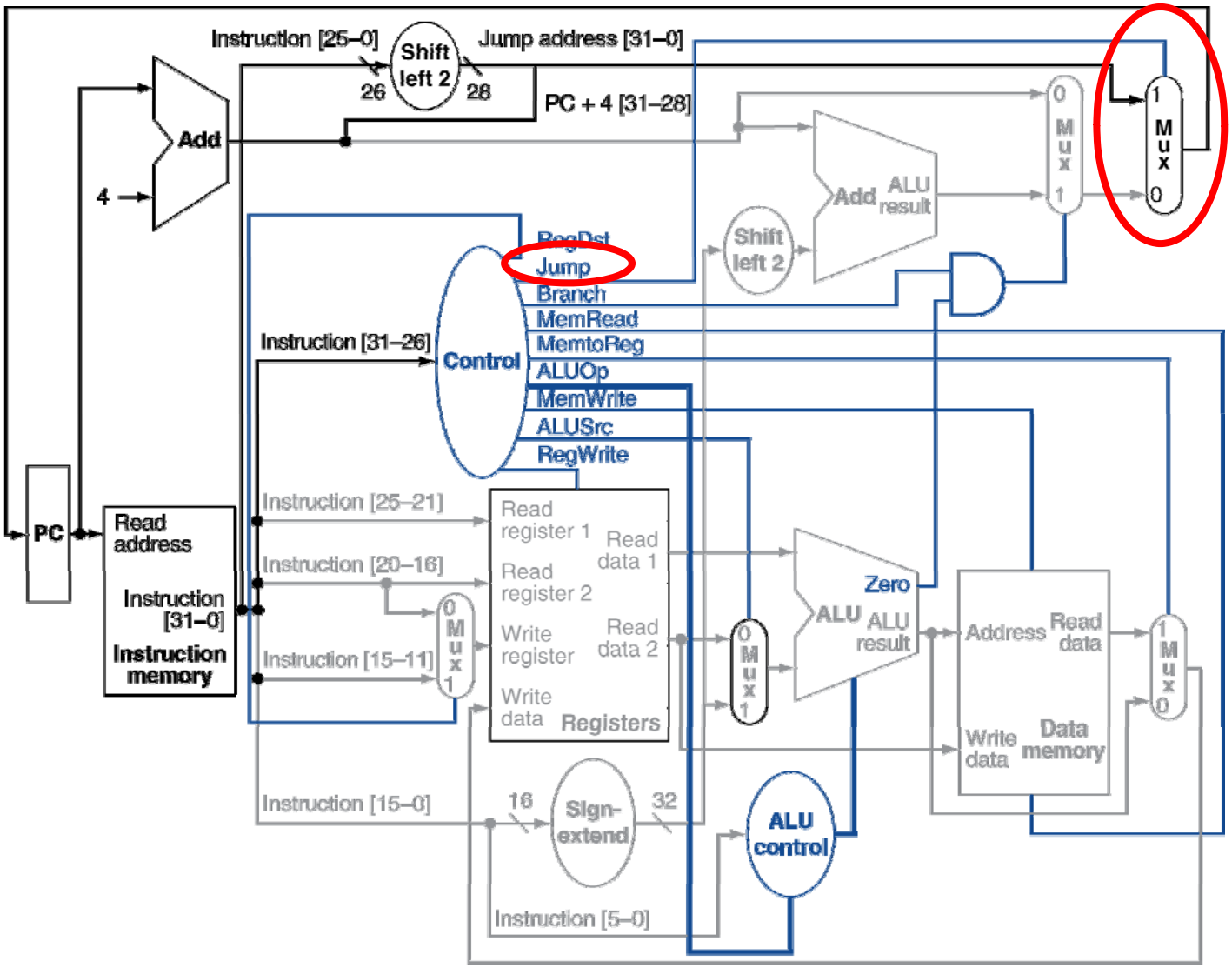


Datapath Executing j

- j



Datapath With Jumps Added



Truth Table for Main Control Signals

- Current design of control is for
  - lw, sw, beq, and, or, add, sub, slt, nor
- I-format: lw, sw, beq
- R-format: and, or, add, sub, slt, nor
- Given 4 OP codes (each 6 bits) as "inputs", the "outputs" are as follows => a main control logic (the next slide)

See appendix A for details

inputs ← outputs →

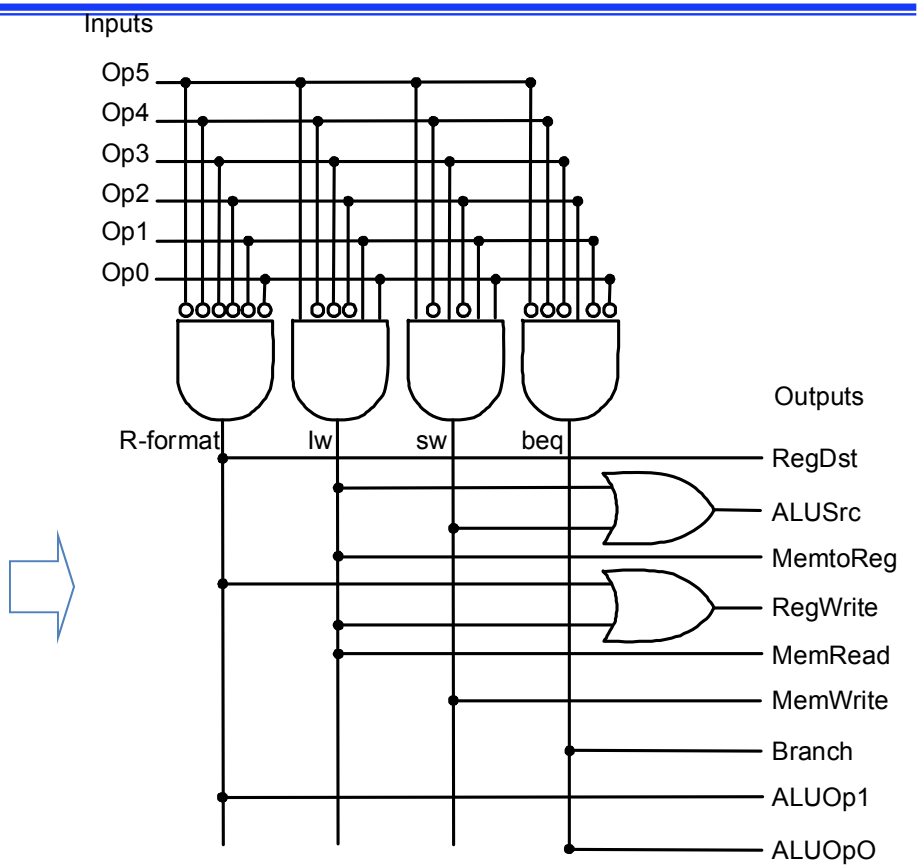
Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUOp0
R-format 000000	1	0	0	1	0	0	0	1	0
lw 100011	0	1	1	1	1	0	0	0	0
sw 101011	X	1	X	0	0	1	0	0	0
beq 000100	X	0	X	0	0	0	1	0	1

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



Main control PLA (programmable logic array)
principle underlying PLAs is that any logical expression can be written as a sum-of-products

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	$C_3C_2C_1C_0$
Merge LW & SW	0	0	X	X	X	X	X	X	0010
	0	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

add
subtract
add
subtract
and
or
slt

Implementation of ALU Control

- C3=0

C2

ALUOp		Function code fields					
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0
X	1	X	X	X	X	X	X
1	X	X	X	X	X	1	X

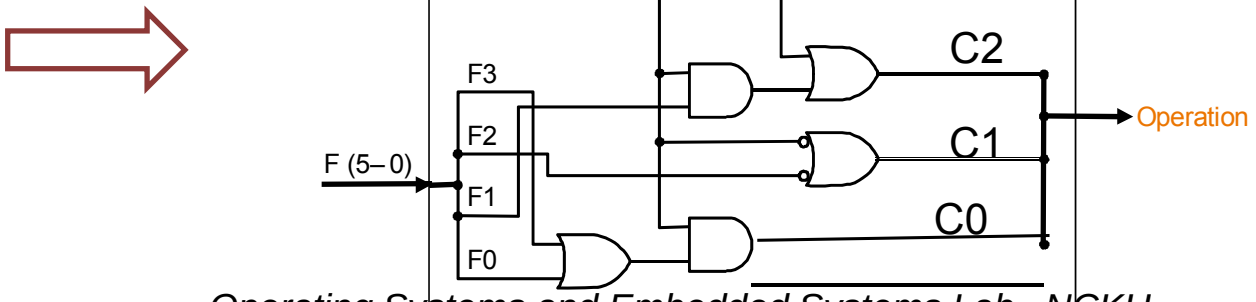
ALUOp0=1 is able to identify branch instruction

C1

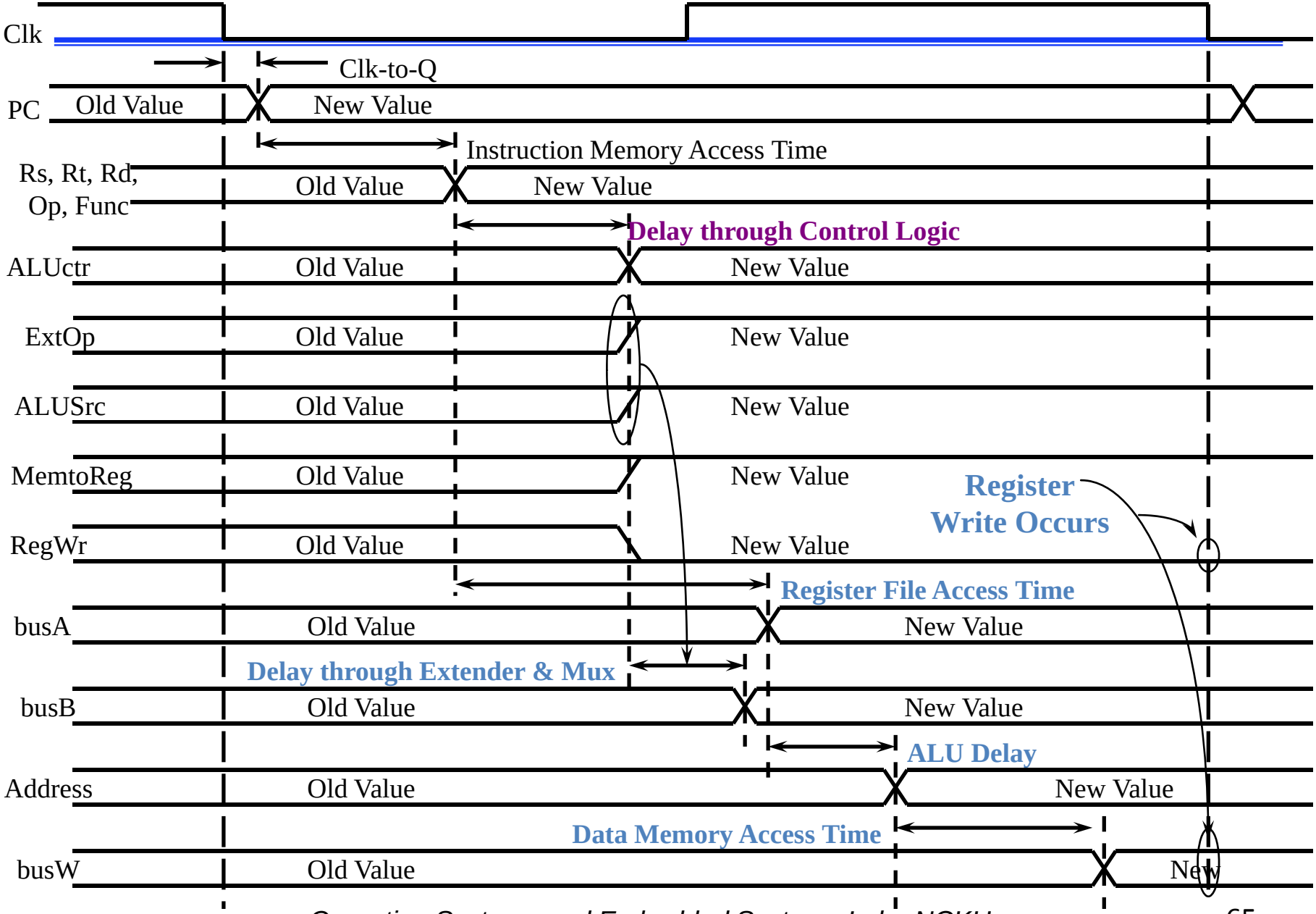
ALUOp		Function code fields					
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0
0	X	X	X	X	X	X	X
X	X	X	X	X	0	X	X

C0

ALUOp		Function code fields					
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0
1	X	X	X	X	X	X	1
1	X	X	X	1	X	X	X



Worst Case Timing (lw)



Drawback of Single-Cycle Design

- Long cycle time:
  - Cycle time must be long enough for the load instruction:

    PC's Clock -to-Q +

    Instruction Memory Access Time +

    Register File Read Time +

    ALU Delay (address calculation) +

    Data Memory Access Time +

    Register File Write Time

- Cycle time for load (lw) is much longer than needed for all other instructions

Summary

- Single cycle processor
  - CPI=1
  - Clock cycle time long
- 5 steps to design a processor:
  1. Analyze ISA => datapath requirements
  2. Select set of datapath components
  3. Assemble datapath meeting the requirements
  4. Analyze implementation of each instruction to determine setting of control points
  5. Assemble the control logic

Summary

- 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
- We will improve performance by pipelining