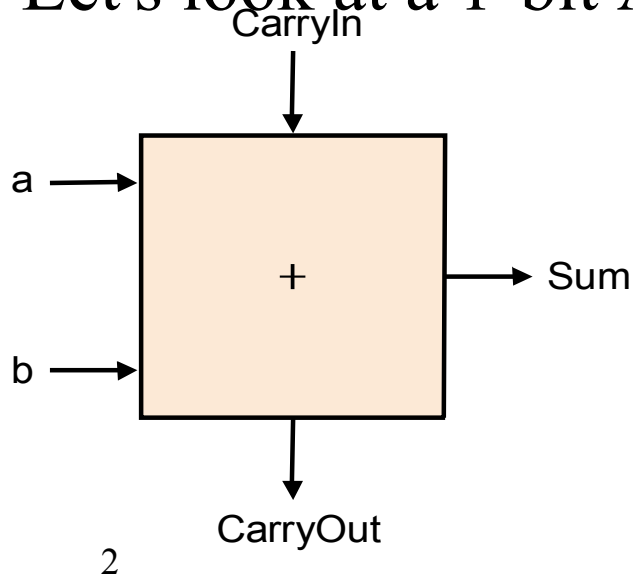


Building a CPU

Different Implementations

- Not easy to decide the “best” way to build something
 - Don't want too many inputs to a single gate
 - Don't want to have to go through too many gates
 - For our purposes, ease of comprehension is important

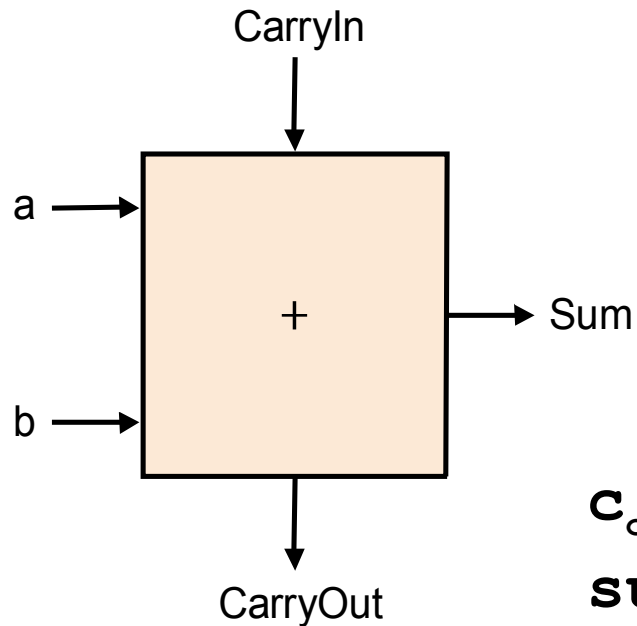
- Let's look at a 1-bit ALU for addition:



<u>ABCi</u>	<u>Co</u>	<u>Sum</u>
000	0	0
001	0	1
010	0	1
011	1	0
100	0	1
101	1	0
110	1	0
111	1	1

Different Implementations

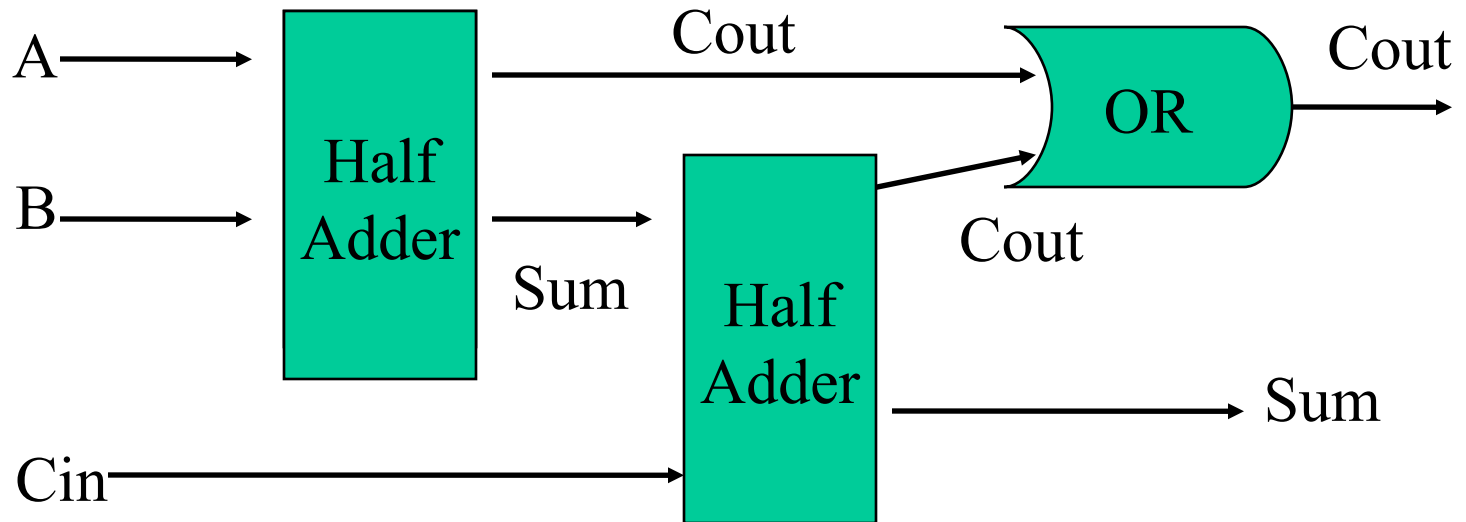
- Remember that we've developed a simple 1 bit adder from basic gates (and, or, xor)



$$\mathbf{c_{out} = a \cdot b + a \cdot c_{in} + b \cdot c_{in}}$$
$$\mathbf{sum = a \oplus b \oplus c_{in}}$$

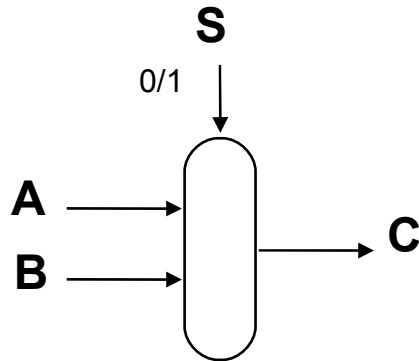
Review : Logic

- Combining half adders to make a full adder



Review: The Multiplexor

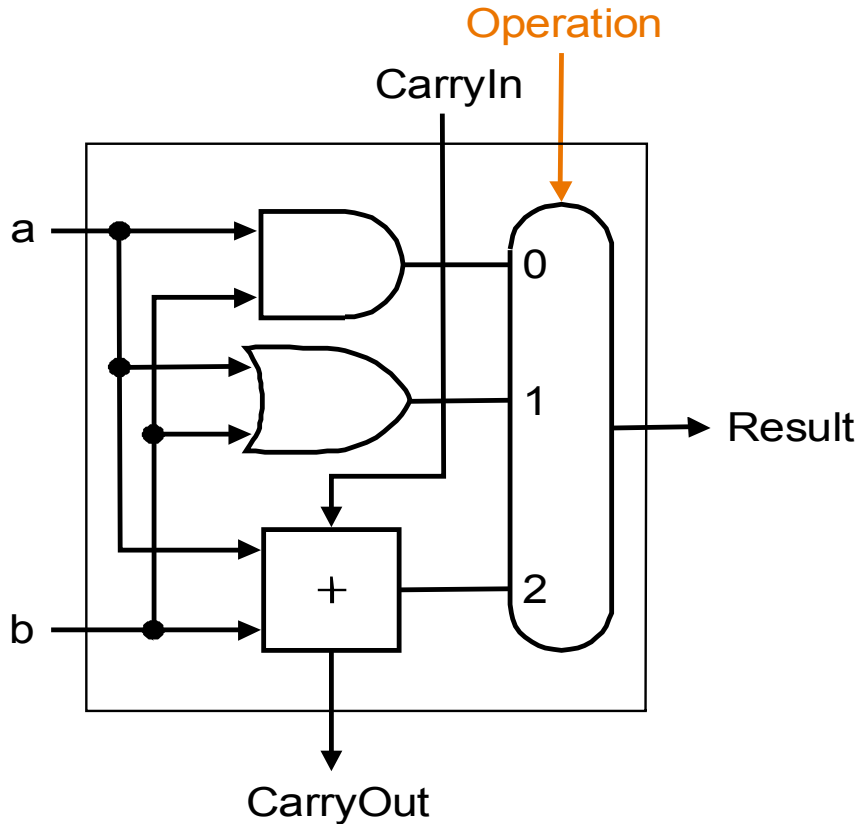
- Selects one of the inputs to be the output, based on a control input



*note: we call this a 2-input mux
even though it has 3 inputs!*

- Lets build our ALU using a MUX:

Building a 1 bit ALU



What is Result? if

$a = 1$

$b = 1$

$\text{CarryIn} = 0$

$\text{op} = 10$

What is Carry Out?

What is Result? if

$a = 1$

$b = 1$

$\text{CarryIn} = 0$

$\text{op} = 01$

What is Carry Out?

Building a 32 bit ALU

What is Result? if

$a = 0111$

$b = 0011$

CarryIn = 0

op = 00

What is Carry Out?

What is Result? if

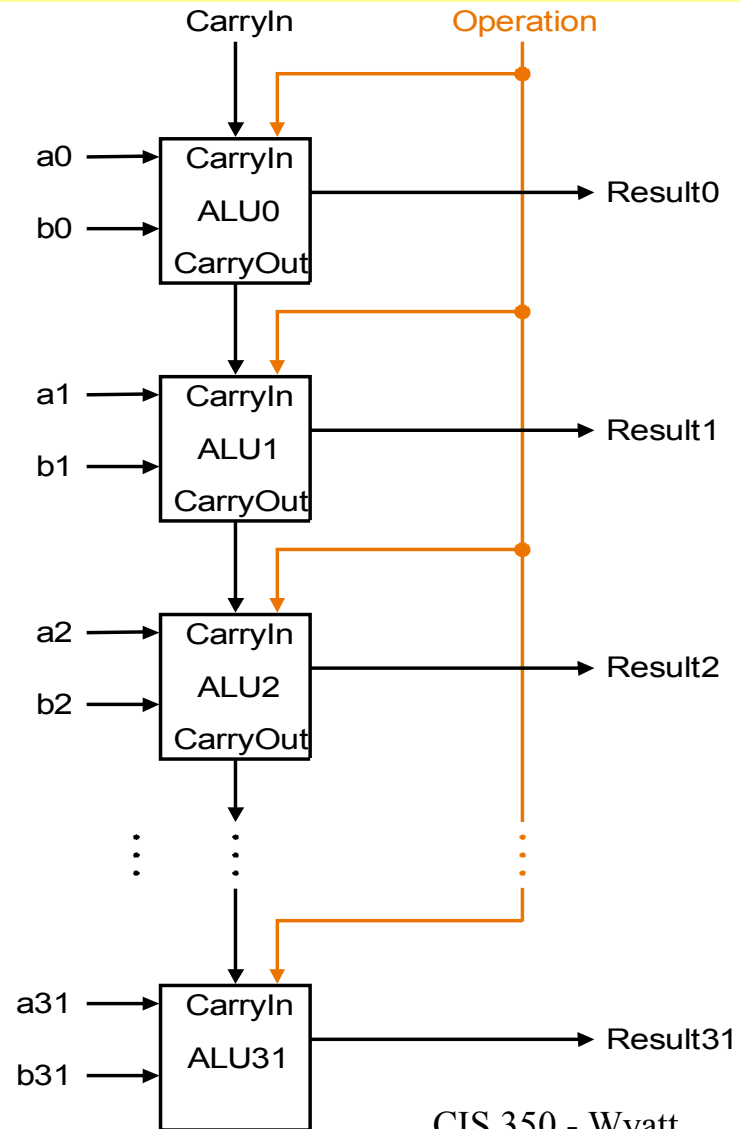
$a = 0111$

$b = 0011$

CarryIn = 0

op = 01

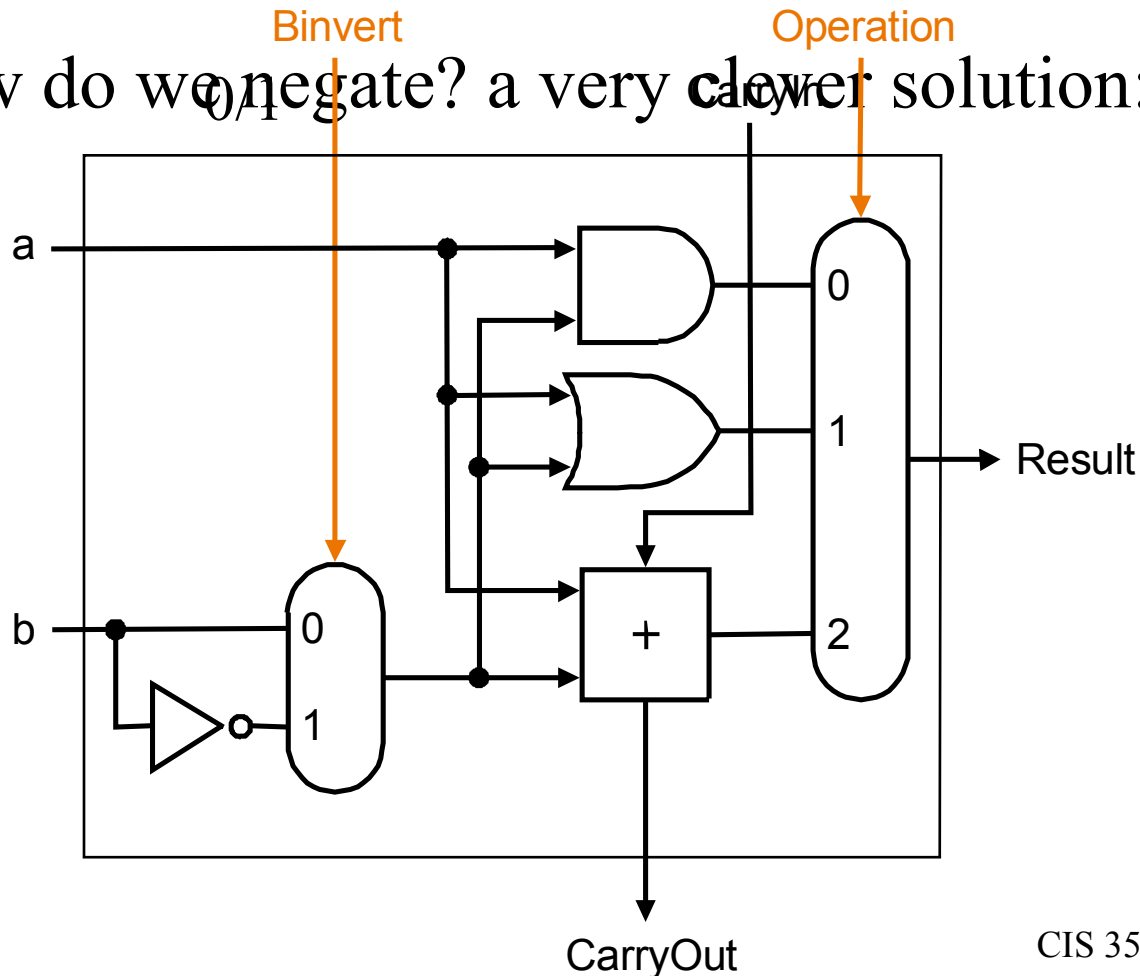
What is Carry Out?



What about subtraction $(a - b)$?

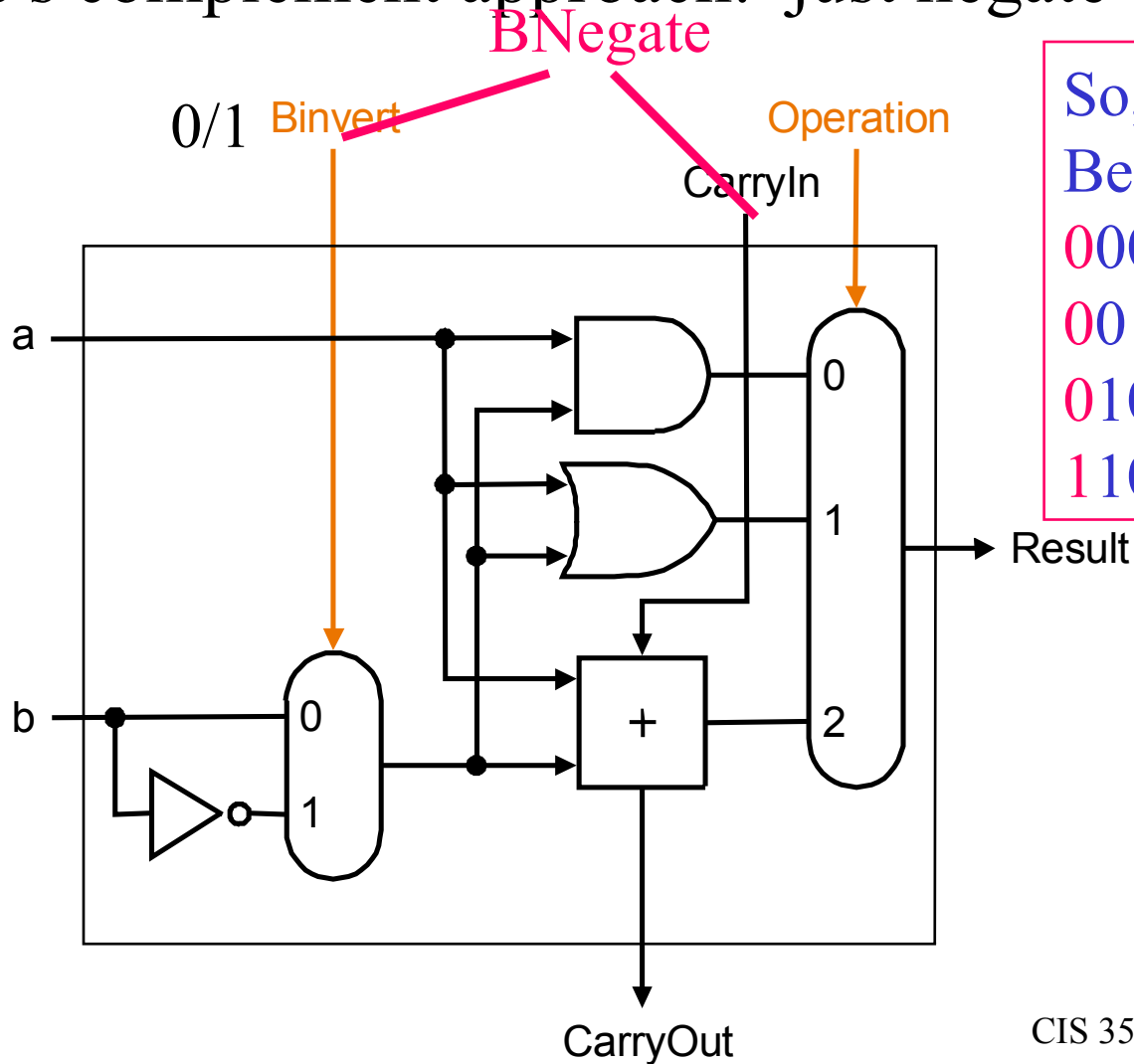
- Two's complement approach: just negate b and add.

- How do we negate? a very clever solution:



What about subtraction (a - b) ?

- Two's complement approach: just negate b and add.



So, OpCode
Becomes 3 bits
000 - and
001 - or
010 - add
110 - sub

Building a 32 bit ALU

What is Result? if

a = 0101

b = 0011

CarryIn = 0

op = 010

What is Carry Out?

What is Result? if

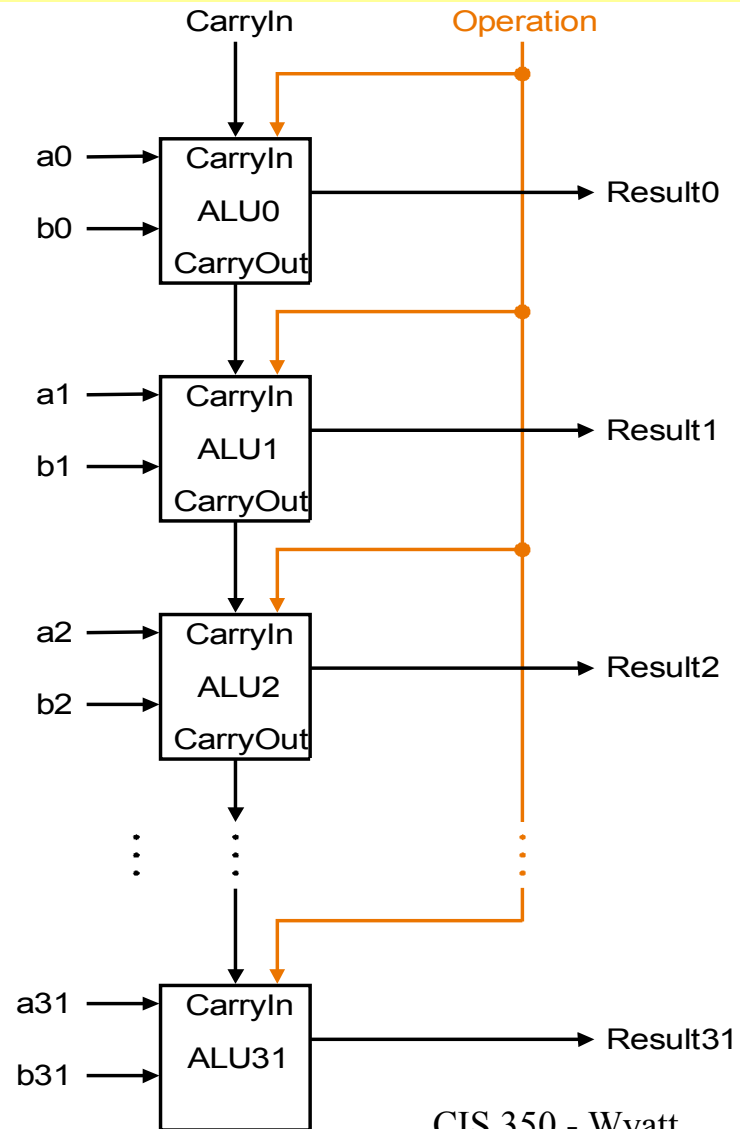
a = 0111

b = 0011

CarryIn = 1

op = 110

What is Carry Out?



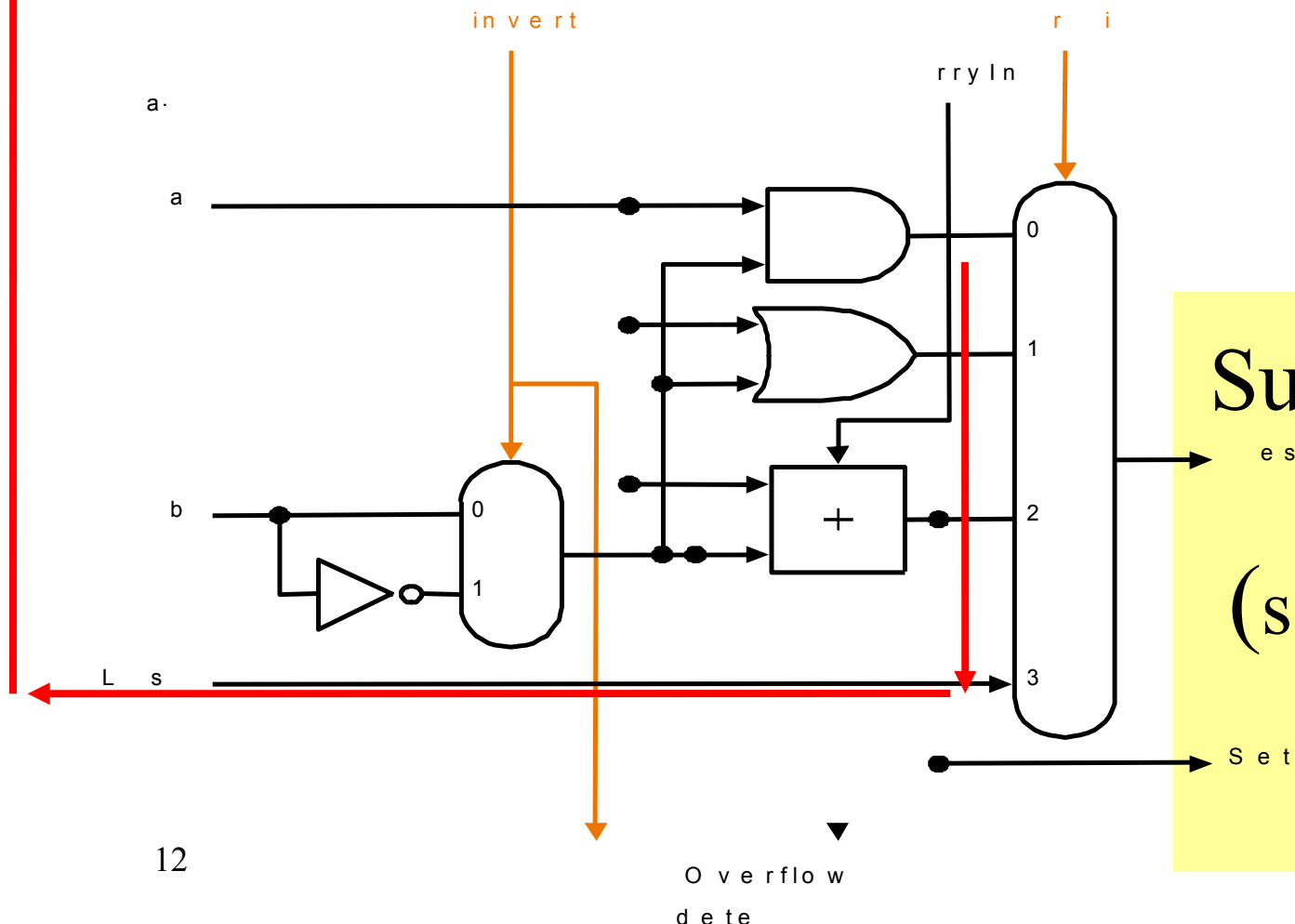
Tailoring the ALU to the MIPS

- Need to support the set-on-less-than instruction (slt)
 - remember: slt is an arithmetic instruction
 - produces a 1 if $rs < rt$ and 0 otherwise
 - use subtraction: $(a-b) < 0$ implies $a < b$
- Need to support test for equality (beq \$t5, \$t6, \$t7)
 - use subtraction: $(a-b) = 0$ implies $a = b$

To "less" on bit zero

Carry Out

High bit

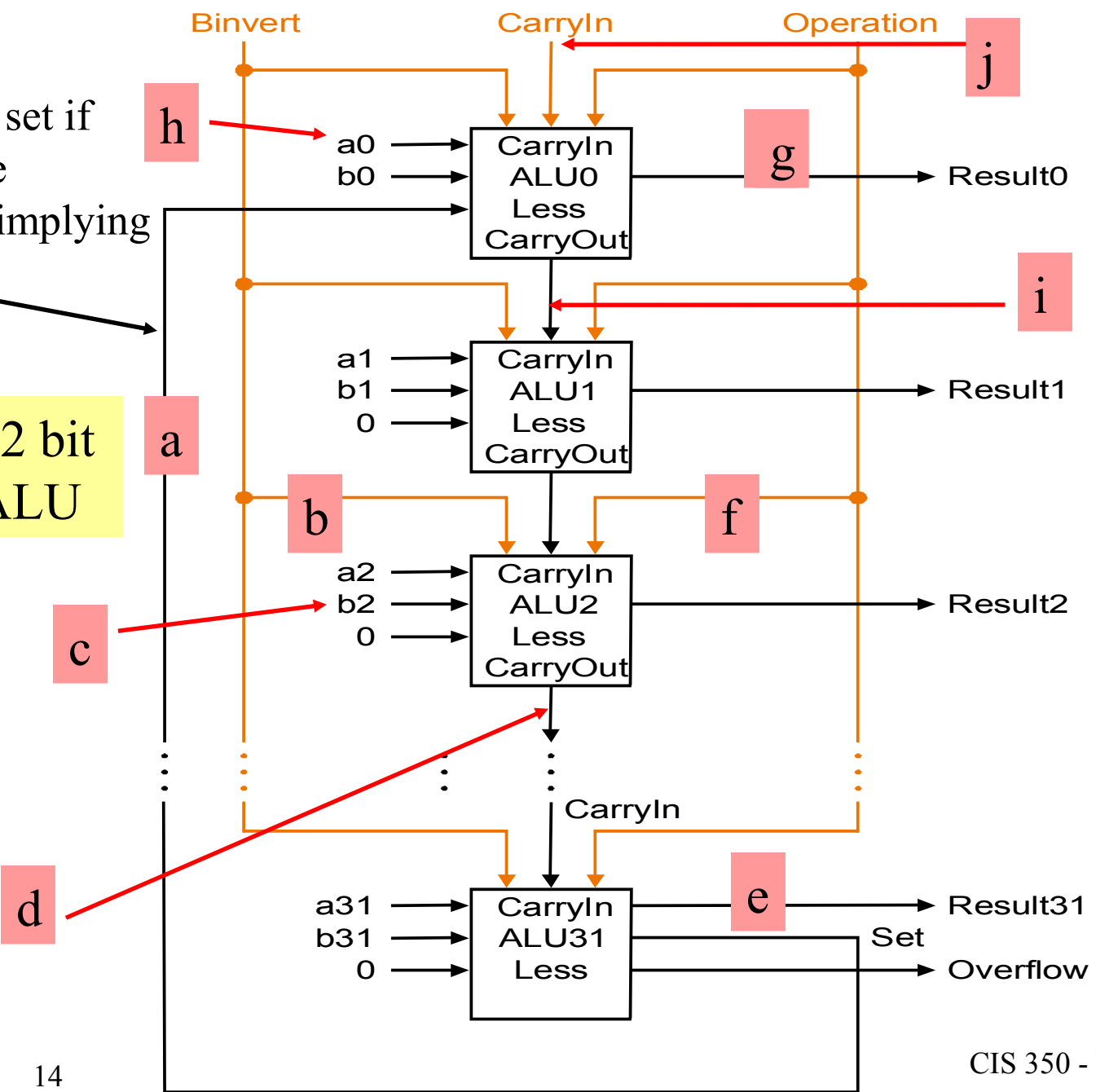


Supporting
slt
(see p. 238)
opcode
111

high bit set if
negative
answer implying
 $a < b$

32 bit
ALU

OPCODES
and 000
or 001
add 010
sub 110
slt 111

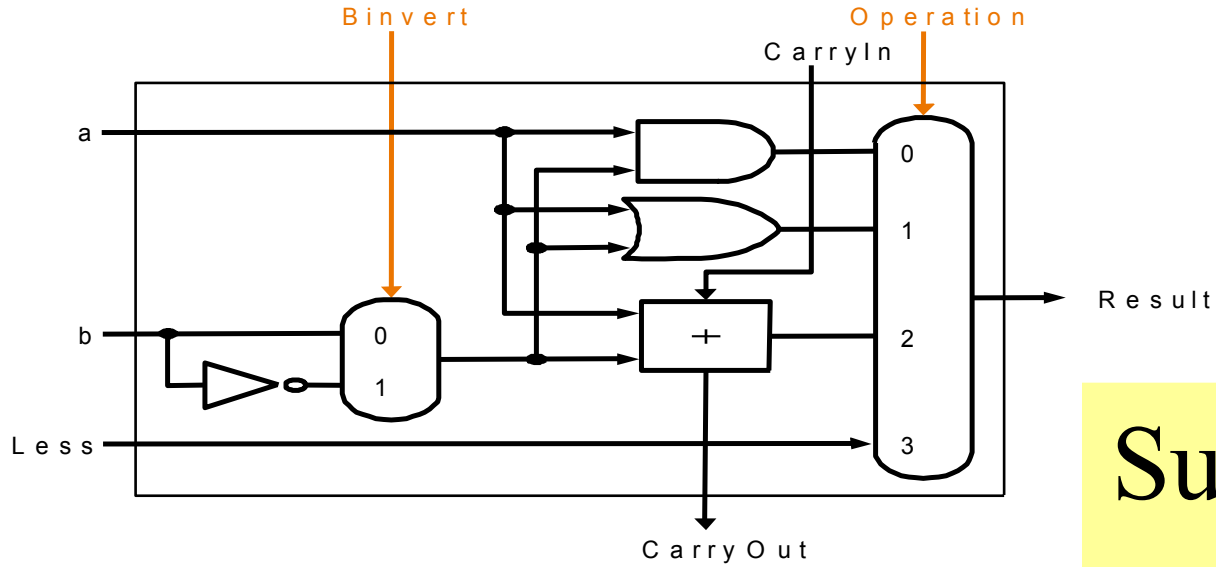


slt Example

- 3 bits
- $a = 001$, $b = 010$, $a < b$, so output a 1
- $\text{binvert} = 1$, $\text{carry in} = 1$, $\text{op} = 11$ (slt)
- result of addition = 111, carry out = 0, so high bit is set, actual result = 001
- this high bit is tied into LESS on bit 0, so it will be output (selected by the 111 on the multiplexor) indicating that $a < b$, our desired

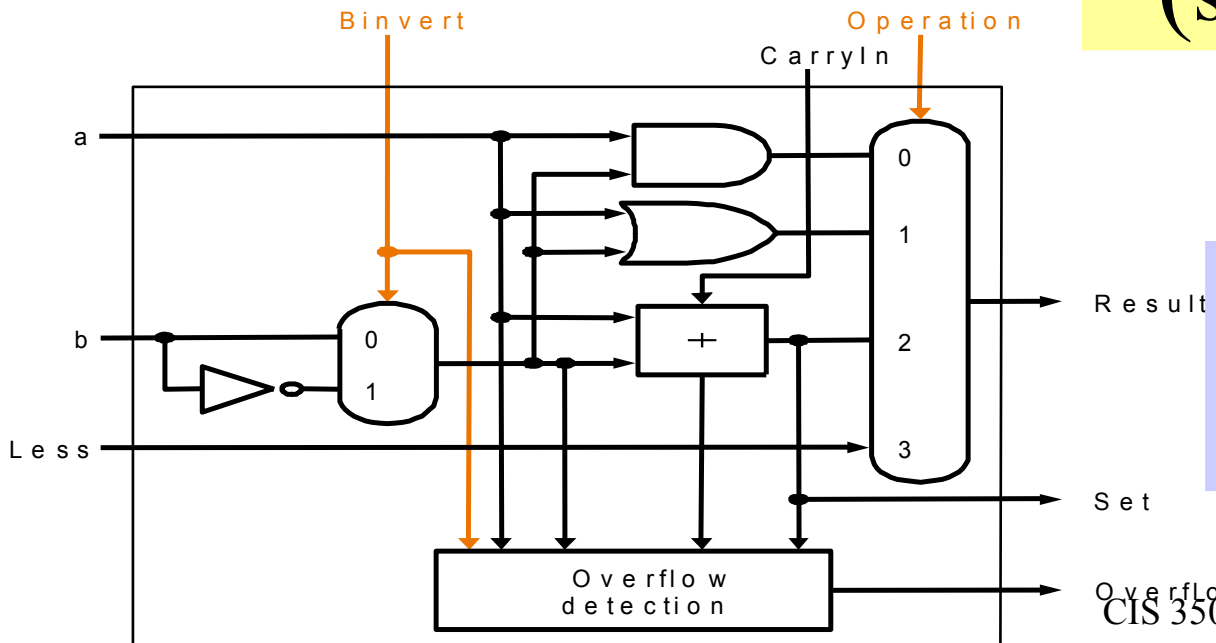
slt Example

- 3 bits
- $a = 011$, $b = 010$, $a \text{ NOT} < b$, so output a 0
- $\text{binvert} = 1$, $\text{carry in} = 1$, $\text{op} = 11$ (slt)
- result of addition = 001, $\text{carry out} = 0$, so high bit is clear, actual result is 000
- this high bit is tied into LESS on bit 0, so it will be output (selected by the 111 on the multiplexor) indicating that a is not $< b$, our desired response



a.

Supporting
slt
(see p. 238)



b.

Note overflow.
see p.238
see p.329

Overflow

- A simple test for overflow in 2s complement
 - look at carryout & carryin of high bit
 - 1. $0001 + 0001 = 0010$ $!CI$ and $!CO = !OV$
 - 2. $1111 + 1111 = 1110$ CI and $CO = !OV$
 - 3. $1000 + 1000 = 0000$ $!CI$ and $CO = OV$
 - 4. $0111 + 0111 = 1110$ CI and $!CO = OV$
- NOTE: for unsigned, you can simply look at the carryout. If carryout, then overflow

Overflow

- CI CO OV

0 0 0

0 1 1

1 0 1

1 1 0

- For 2's complement overflow detection, run the carry in on the high bit and the carry out of the high bit into an XOR gate = overflow detection

Test for equality- zero detector

- Note control lines:

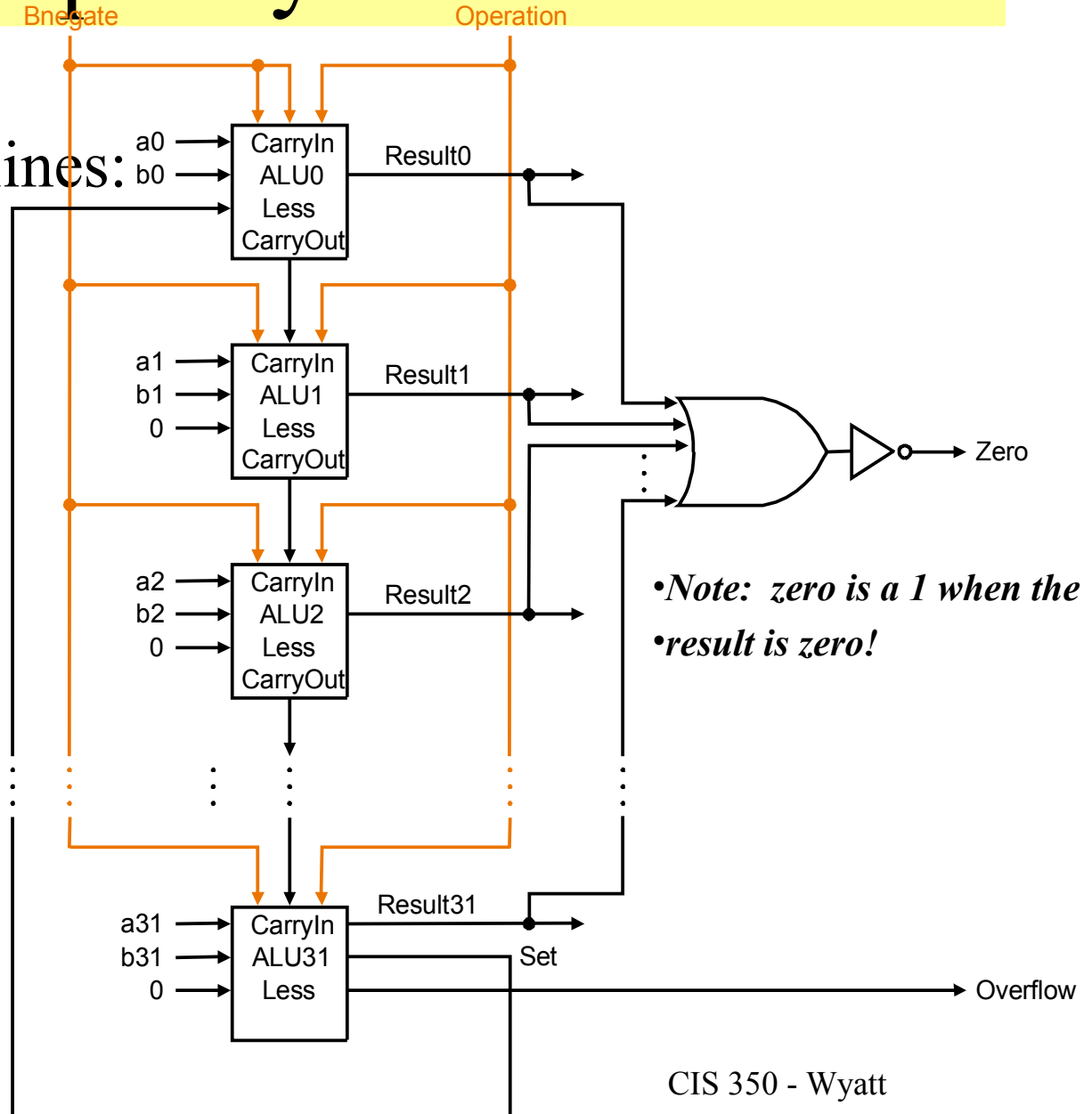
000 = and

001 = or

010 = add

110 = sub

111 = slt



Conclusion

- We can build an ALU to support the MIPS instruction set
 - key idea: use multiplexor to select the output we want
 - we can efficiently perform subtraction using two's complement
 - we can replicate a 1-bit ALU to produce a 32-bit ALU
- Important points about hardware
 - all of the gates are always working
 - the speed of a gate is affected by the number of inputs to the gate
 - the speed of a circuit is affected by the number of gates in series (on the “critical path” or the “deepest level of logic”)
- Our primary focus: comprehension, however,
 - Clever changes to organization can improve performance (similar to using better algorithms in software)
- we'll look at two examples for addition and multiplication

Problem: ripple carry adder is slow

- Is a 32-bit ALU as fast as a 1-bit ALU?
- Is there more than one way to do addition?
 - two extremes: ripple carry and sum-of-products
- Can you see the ripple? How could you get rid of it?

$$c_1 = b_0c_0 + a_0c_0 + a_0b_0$$

$$c_2 = b_1c_1 + a_1c_1 + a_1b_1$$

$$c_3 = b_2c_2 + a_2c_2 + a_2b_2$$

$$c_4 = b_3c_3 + a_3c_3 + a_3b_3$$

Carry-lookahead adder

- Motivation:

- If we didn't know the value of carry-in, what could we do?

- When would we always generate a carry?

$$g_i = a_i \cdot b_i$$

- generates a carryout independent of a carryin

- When would we propagate the carry?

$$p_i = a_i + b_i$$

- propagates a carryin to a carryout

- Did we get rid of the ripple?

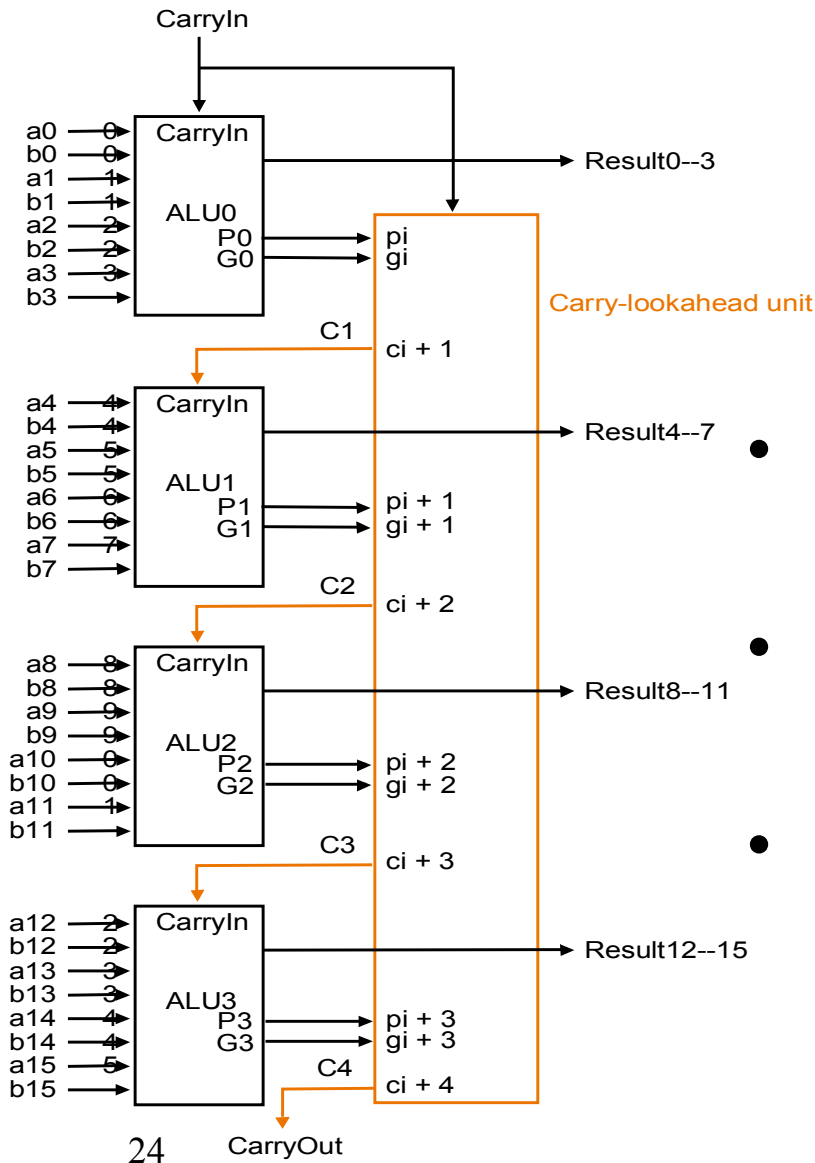
$$c_1 = g_0 + p_0 c_0$$

$$c_2 = g_1 + p_1 c_1 \quad c_2 =$$

$$c_3 = g_2 + p_2 c_2 \quad c_3 =$$

$$c_4 = g_3 + p_3 c_3 \quad c_4 =$$

Use principle to build bigger adders



- Can't build a 16 bit adder this way... (too big)
- Could use ripple carry of 4-bit CLA adders
- Better: use the CLA principle again!

Overflow

- A simple test for overflow in 2s complement
 - look at carryout & carryin of high bit
 - 1. $0001 + 0001 = 0010$ $!CI$ and $!CO = !OV$
 - 2. $1111 + 1111 = 1110$ CI and $CO = !OV$
 - 3. $1000 + 1000 = 0000$ $!CI$ and $CO = OV$
 - 4. $0111 + 0111 = 1110$ CI and $!CO = OV$
- NOTE: for unsigned, you can simply look at the carryout. If carryout, then overflow

Test for equality- zero detector

- Note control lines:

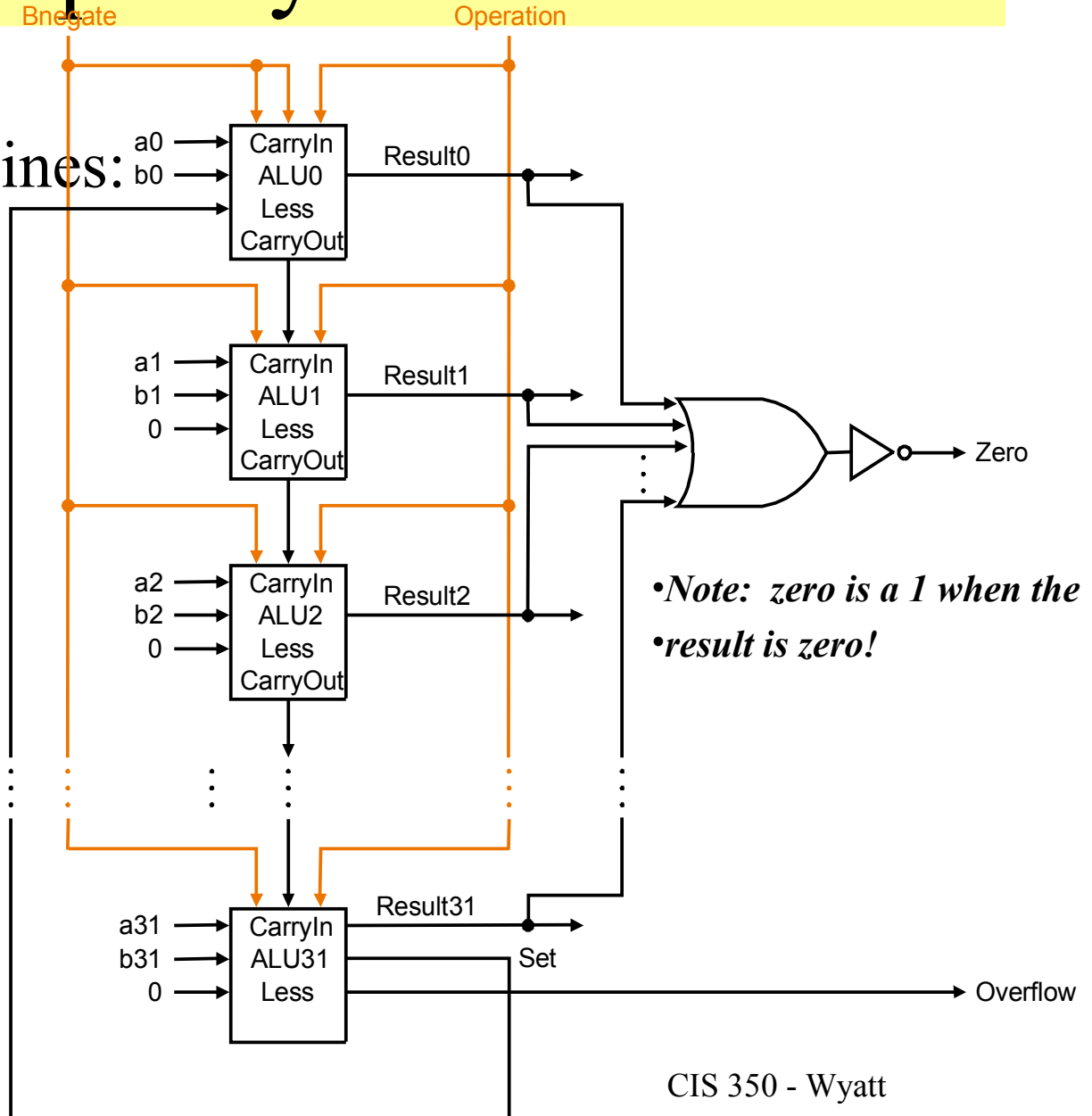
000 = and

001 = or

010 = add

110 = sub

111 = slt

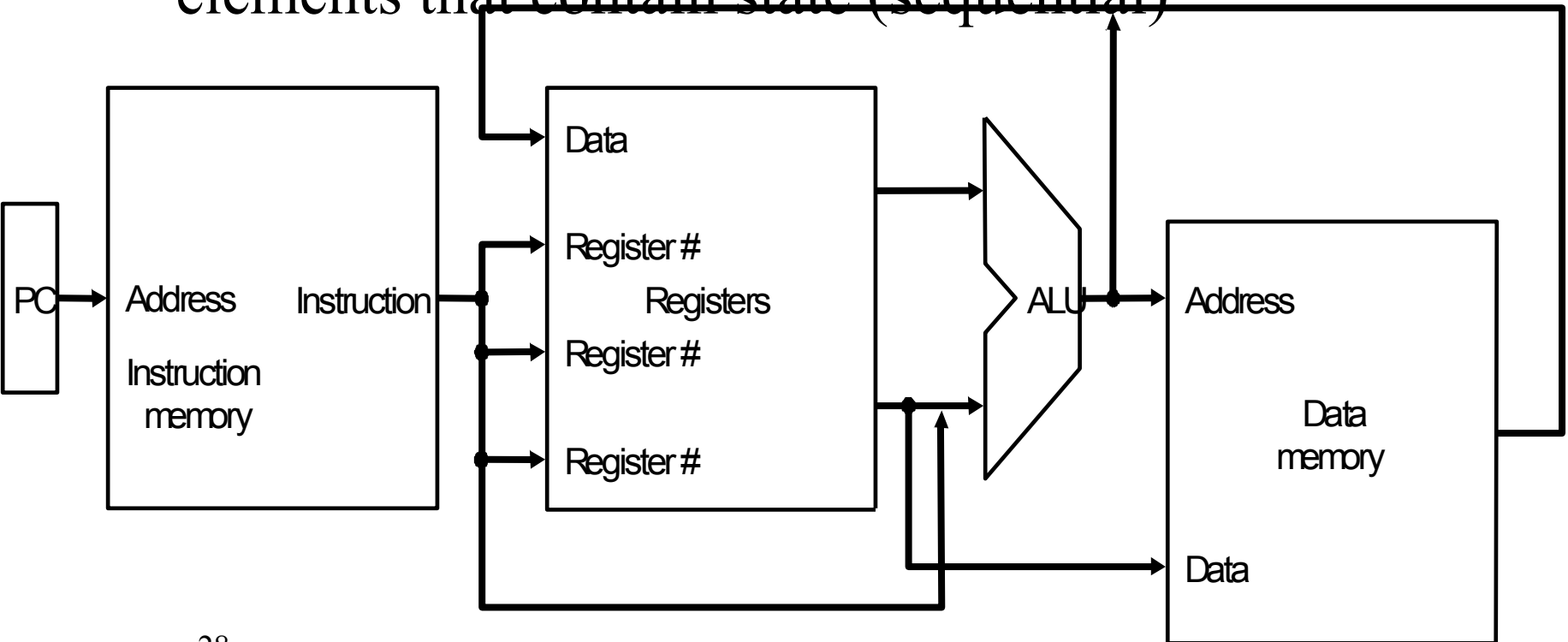


The Processor: Datapath & Control

- We're ready to look at implementation of MIPS
- Simplified to contain only:
 - memory-reference instructions: `lw`, `sw`
 - arithmetic-logical instructions: `add`, `sub`, `and`, `or`, `slt`
 - control flow instructions: `beq`, `j`
- Generic Implementation:
 - use the program counter (PC) to supply instruction address
 - get the instruction from memory
 - read registers
 - use the instruction to decide exactly what to do

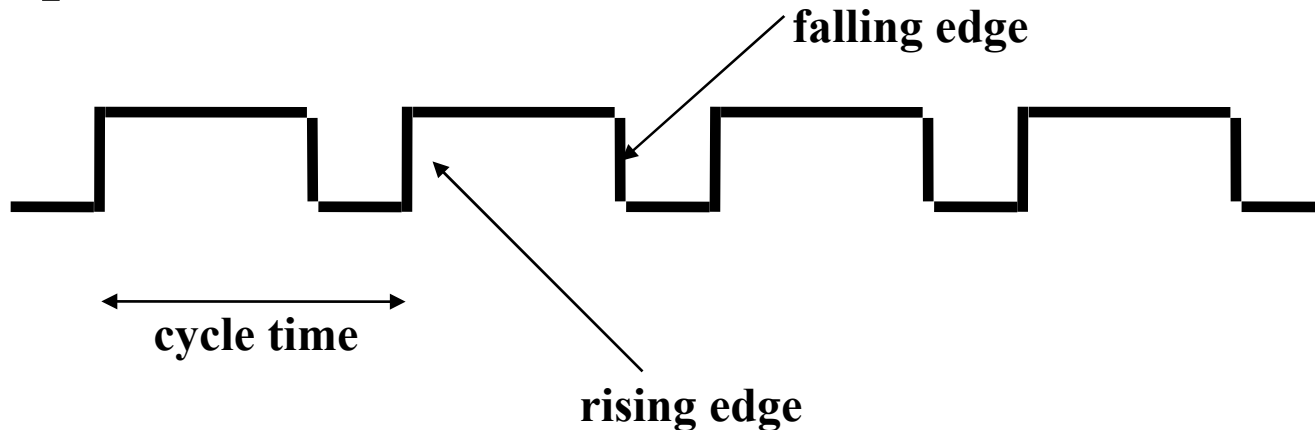
Implementation Details

- Abstract / Simplified View:
Two types of functional units:
 - elements that operate on data values (combinational)
 - elements that contain state (sequential)



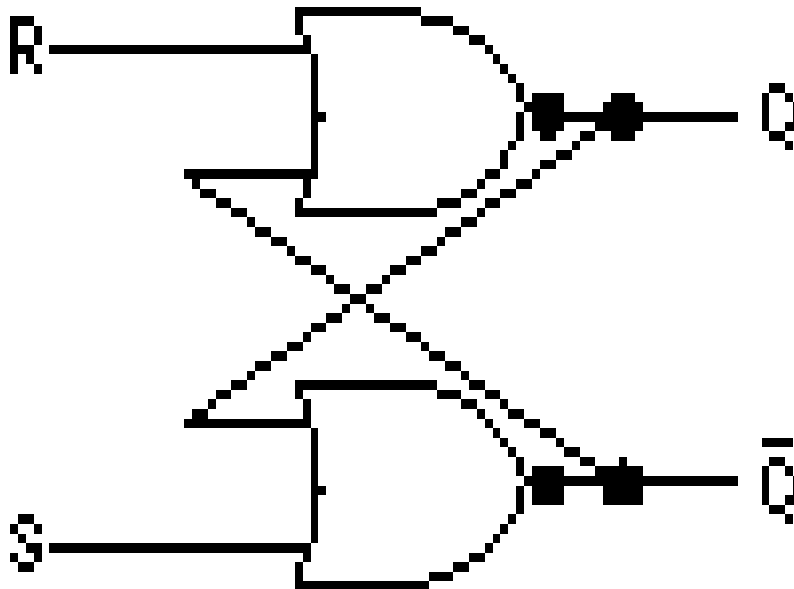
State Elements

- Clocks used in synchronous logic
- when should an element that contains state be updated?



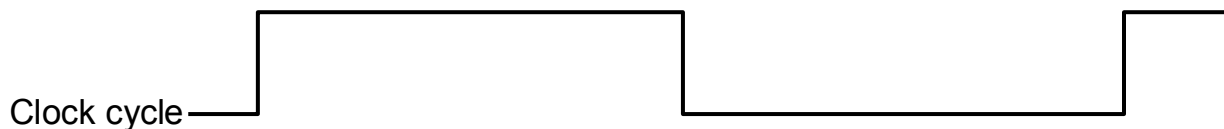
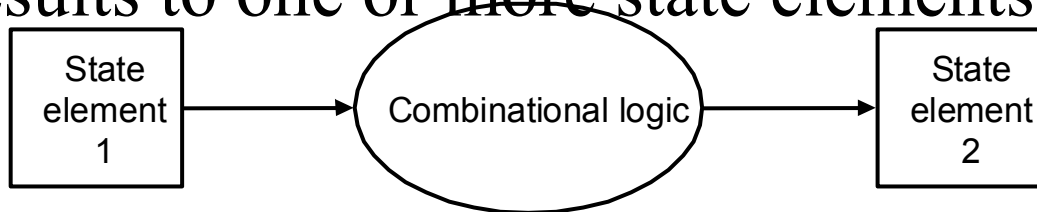
An unclocked state element

- The set-reset latch
- output depends on present inputs and also on past inputs



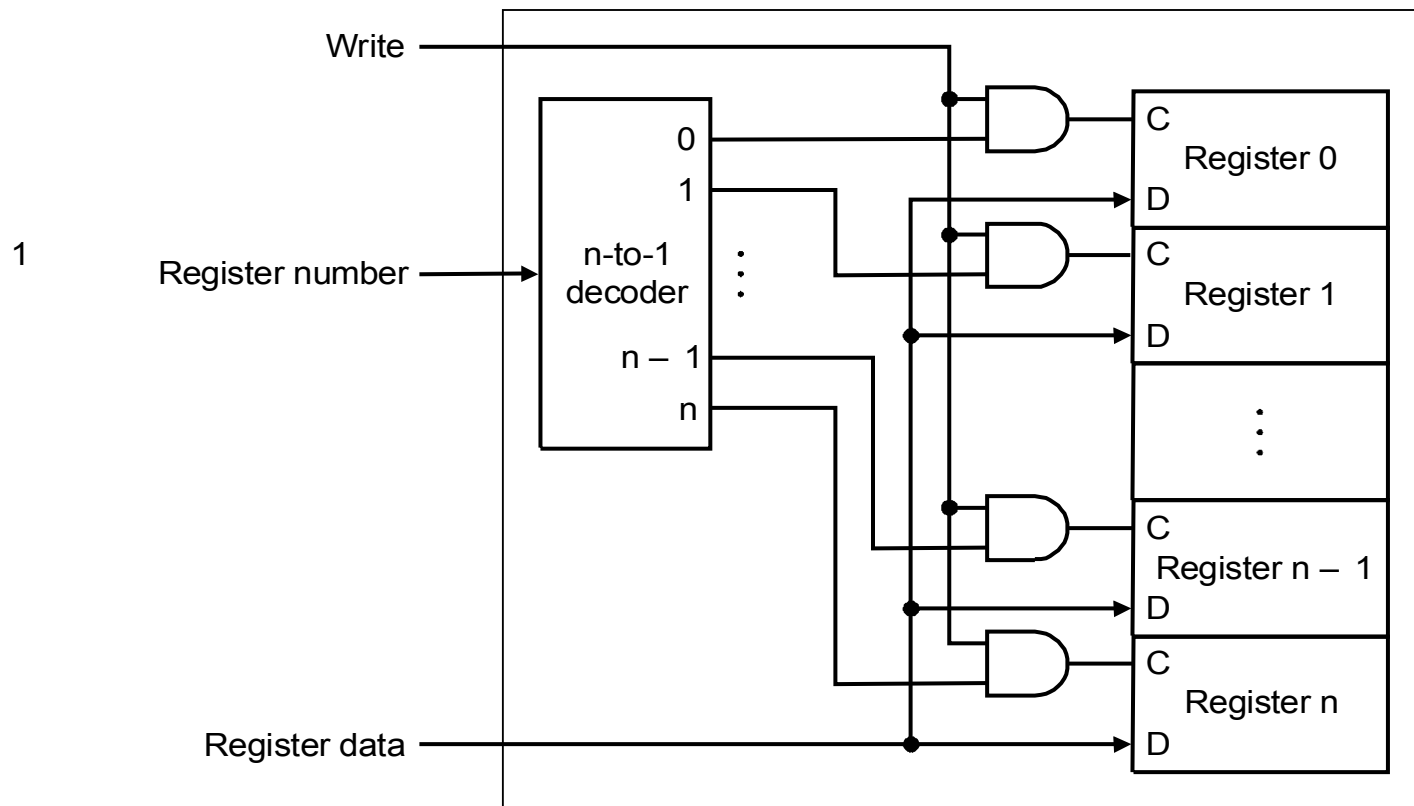
Our Implementation

- An edge triggered methodology
- Typical execution:
 - read contents of some state elements,
 - send values through some combinational logic
 - write results to one or more state elements

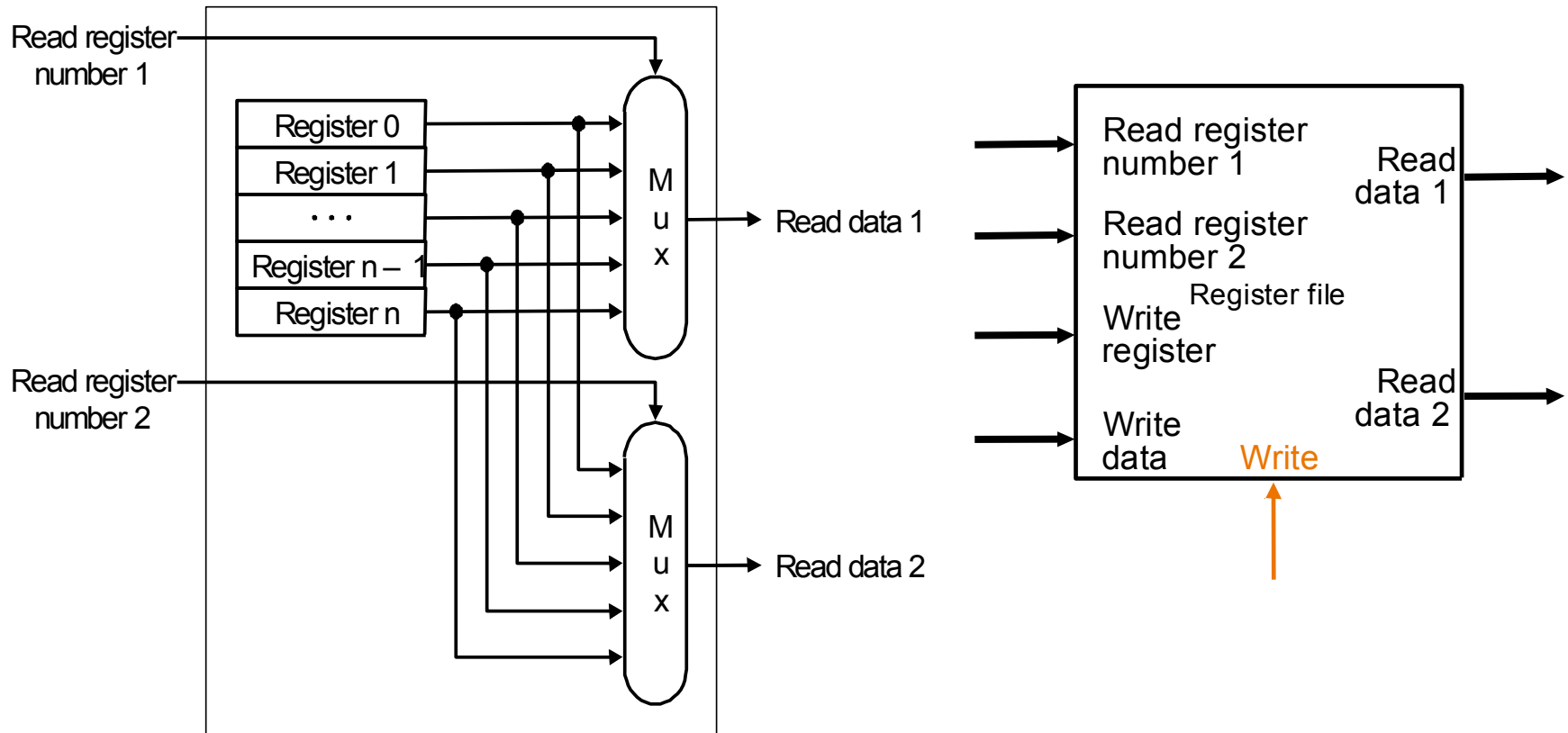


Register File

- Note: we still use the real clock to determine when to write

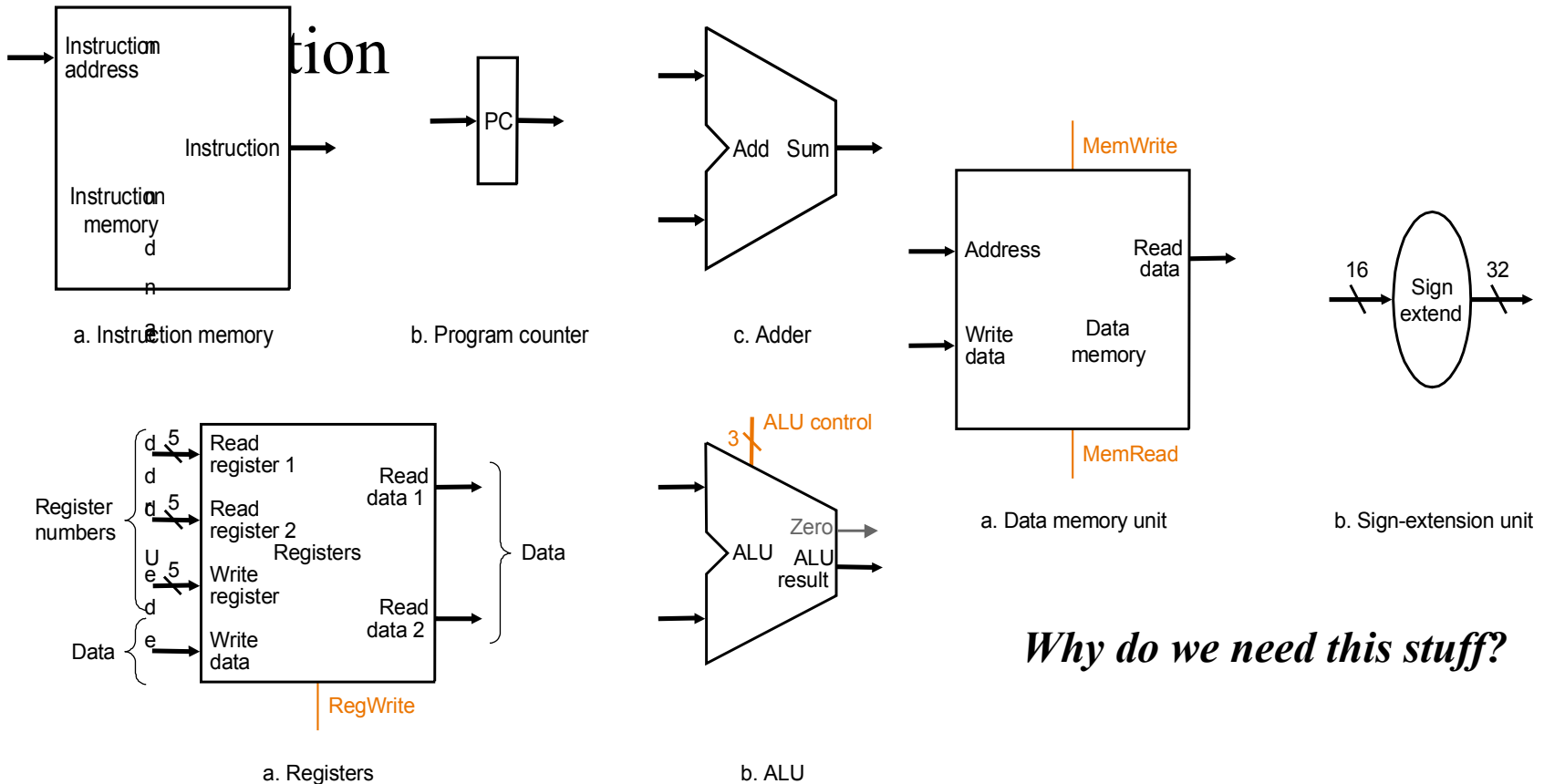


Register File



Simple Implementation

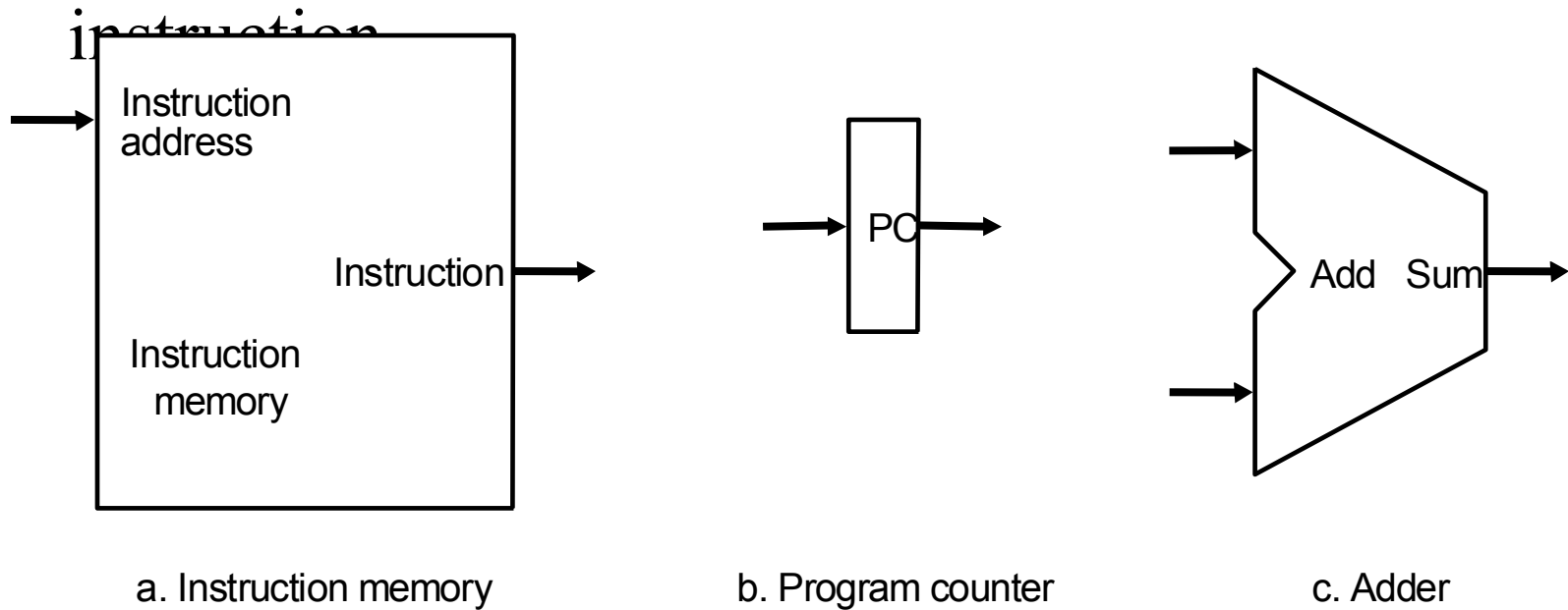
- Include the functional units we need for each



Why do we need this stuff?

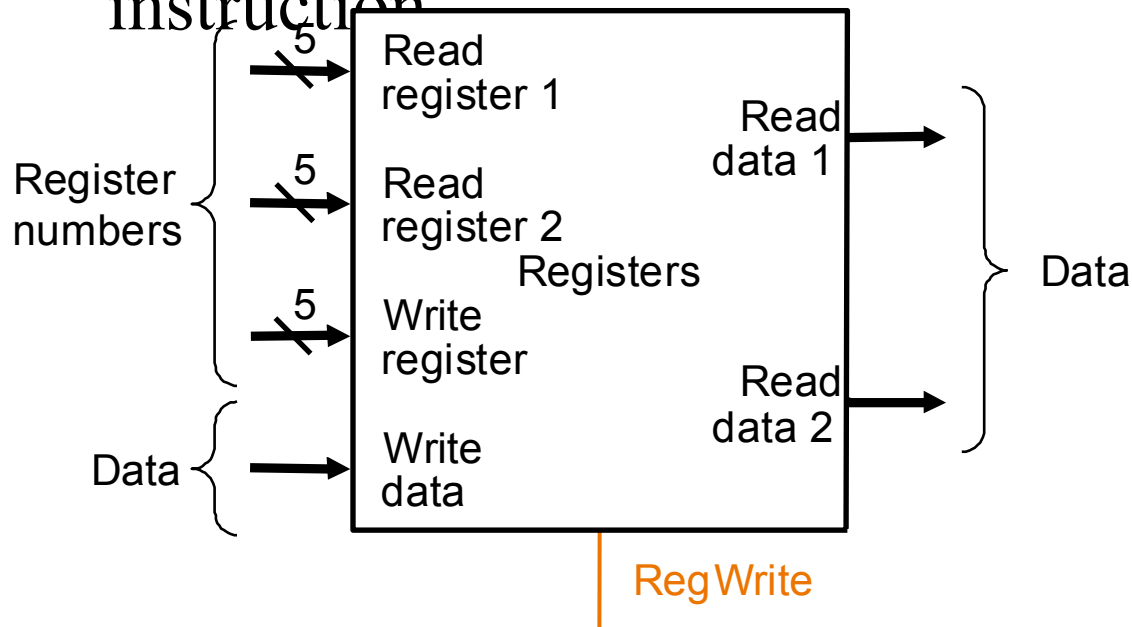
Simple Implementation

- Include the functional units we need for each

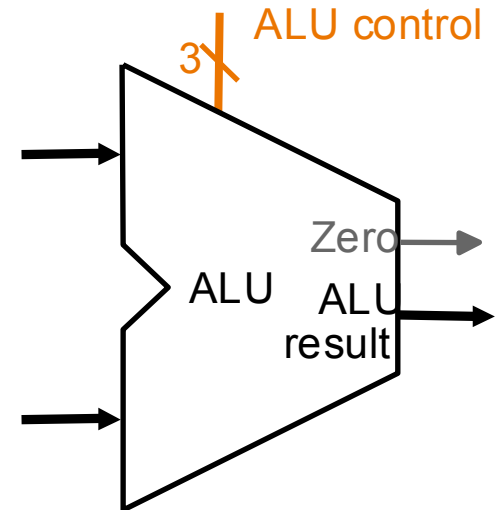


Simple Implementation

- Include the functional units we need for each instruction



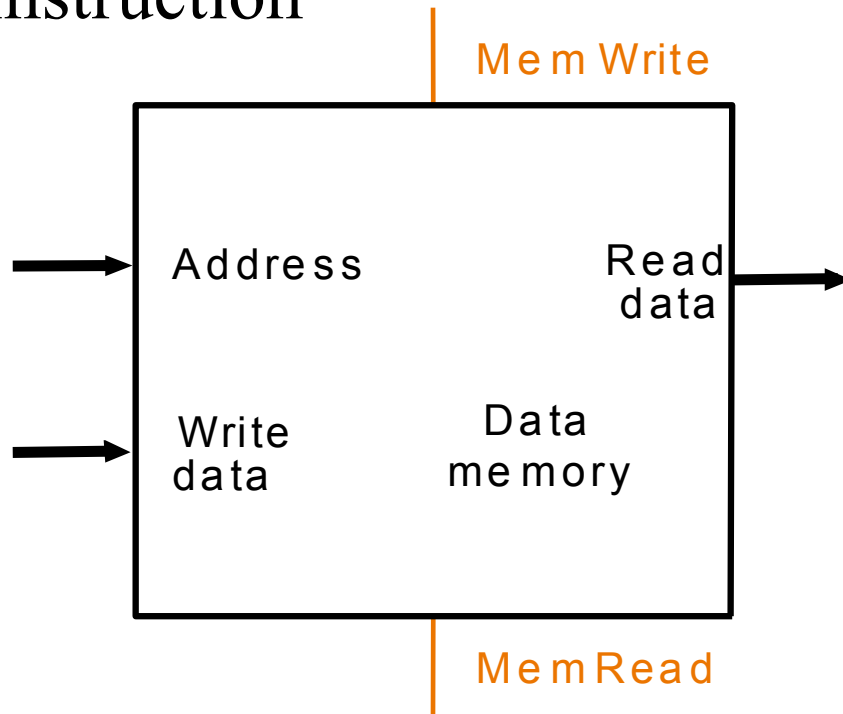
a. Registers



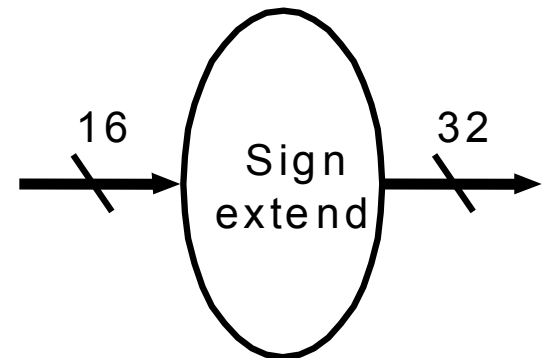
b. ALU

Simple Implementation

- Include the functional units we need for each instruction



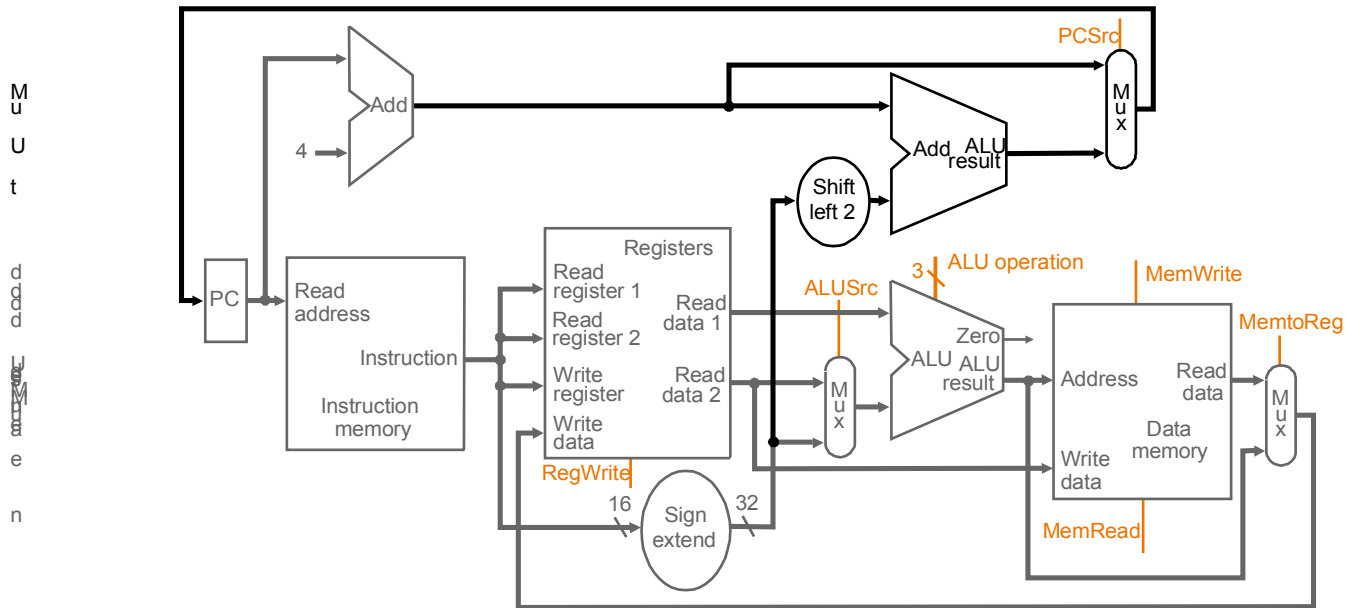
a. Data memory unit

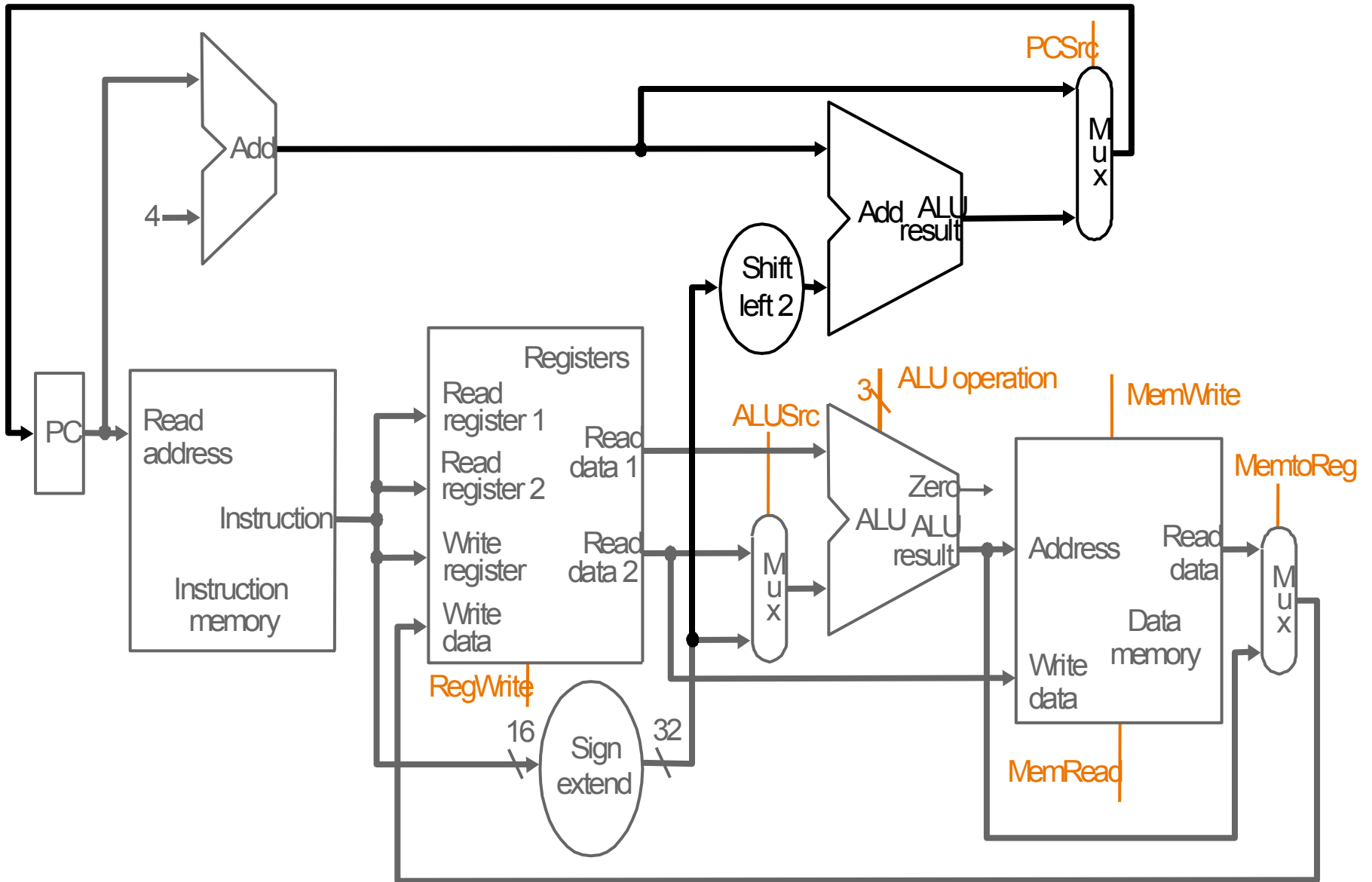


b. Sign-extension unit

Building the Datapath

- Use multiplexors to stitch them together



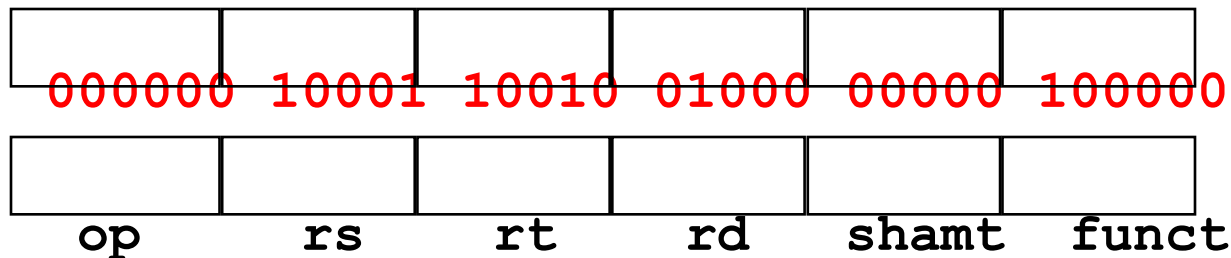


Control

- Selects the operations to perform (ALU, read/write, etc.)
- Controls the flow of data (multiplexor inputs)
- Information comes from the 32 bits of the instruction
- **Example:**

add \$8, \$17, \$18

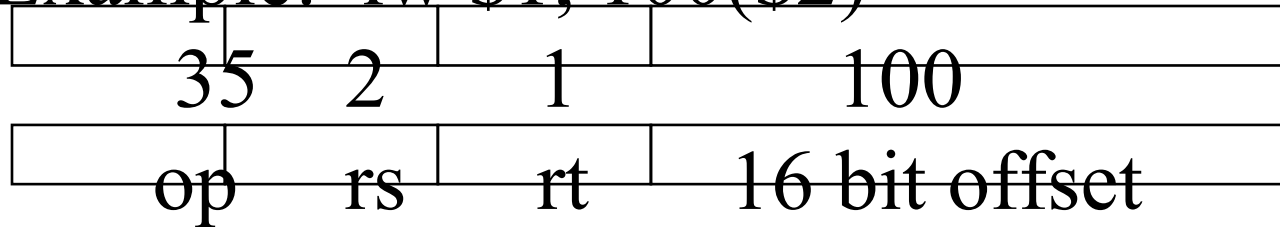
Instruction Format:



ALU's operation based on instruction type and function code

Control

- Example: lw \$1, 100(\$2)



- | | | | |
|--------|-------|-------|------------------|
| 100011 | 00010 | 00001 | 0000000001100100 |
|--------|-------|-------|------------------|

Control

- ALU control input

000 AND

001 OR

010 add

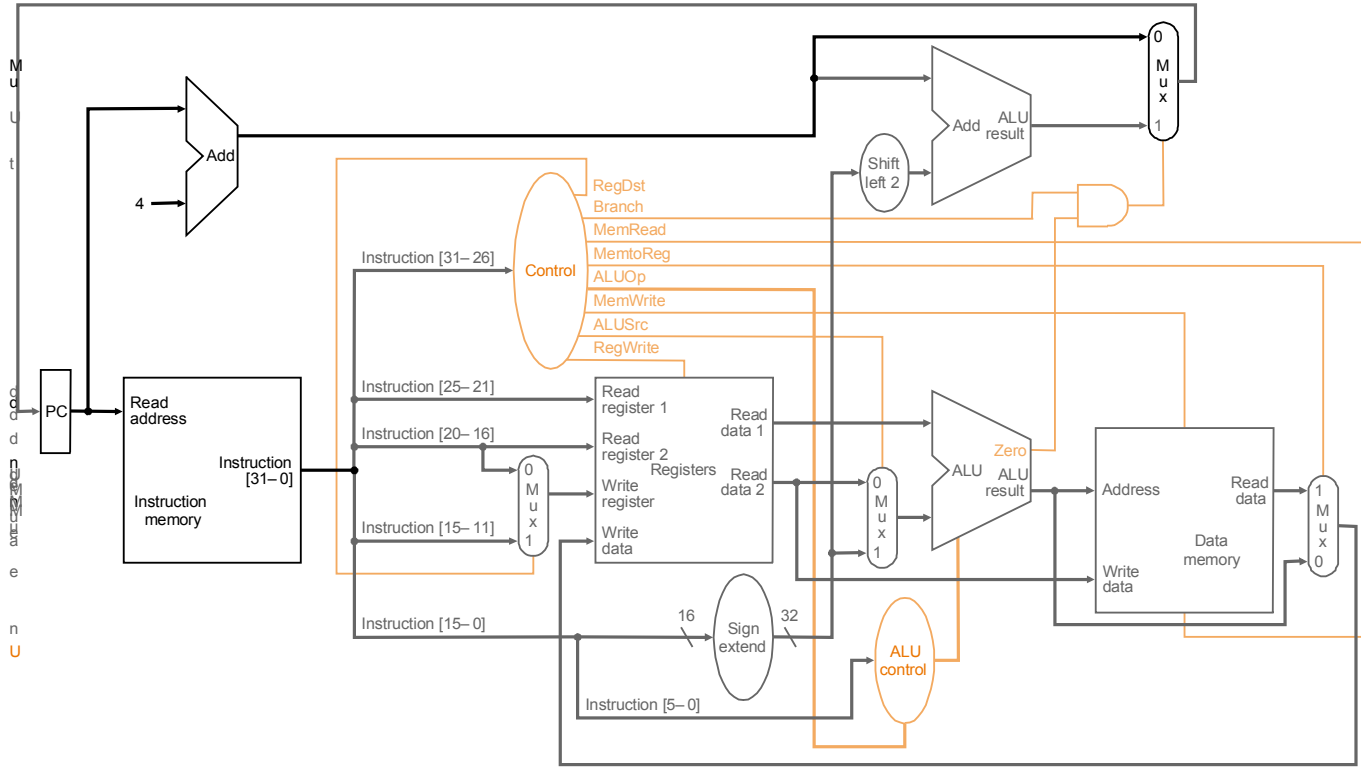
110 subtract

111 set-on-less-than

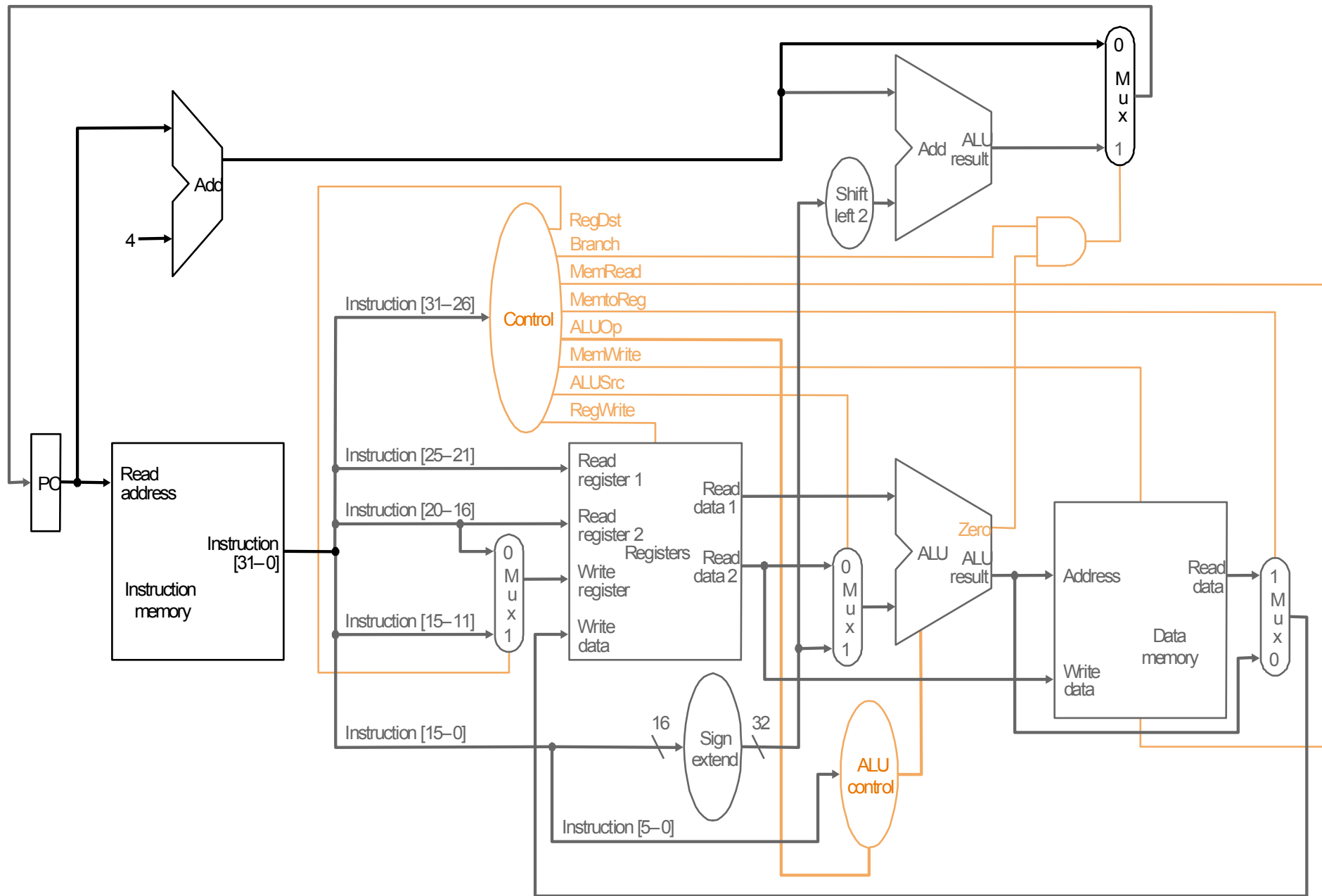
Why is the code for subtract 110 and not 011?

Control

Look at drawing in book



Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
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

- Try at home this:
- add \$3, \$5, \$17
- $PC = 100$
- $\$3 = ??$
- $\$5 = 68$
- $\$17 = 32$

Control

- Try this:
- $1w \$3, (16) \15
- $PC = 100$
- $\$ 3 = ??$
- $\$ 15 = 400$
- $\text{memory } @ (16) \$15 = 257$

MAIN Control

Instruction	RegDst	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	ALUOp1	ALUOp2
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

ALU Control

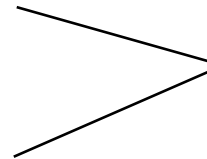
- Describe hardware to compute 3-bit ALU control input

- given instruction type

00 = lw, sw

01 = beq,

11 = arithmetic



ALUOp

computed from instruction type

- function code for arithmetic

Describe

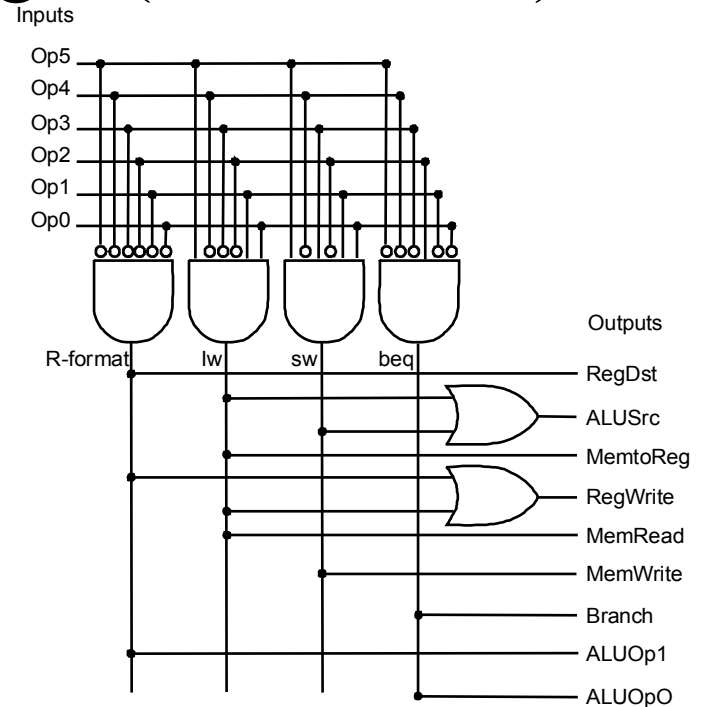
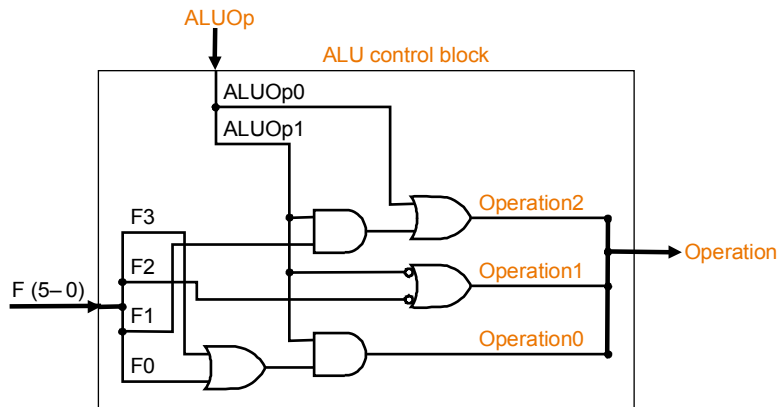
ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	010
X	1	X	X	X	X	X	X	110
1	X	X	X	0	0	0	0	010
1	X	X	X	0	0	1	0	110
1	X	X	X	0	1	0	0	000
1	X	X	X	0	1	0	1	001
1	X	X	X	1	0	1	0	111

):

Look at drawing in book

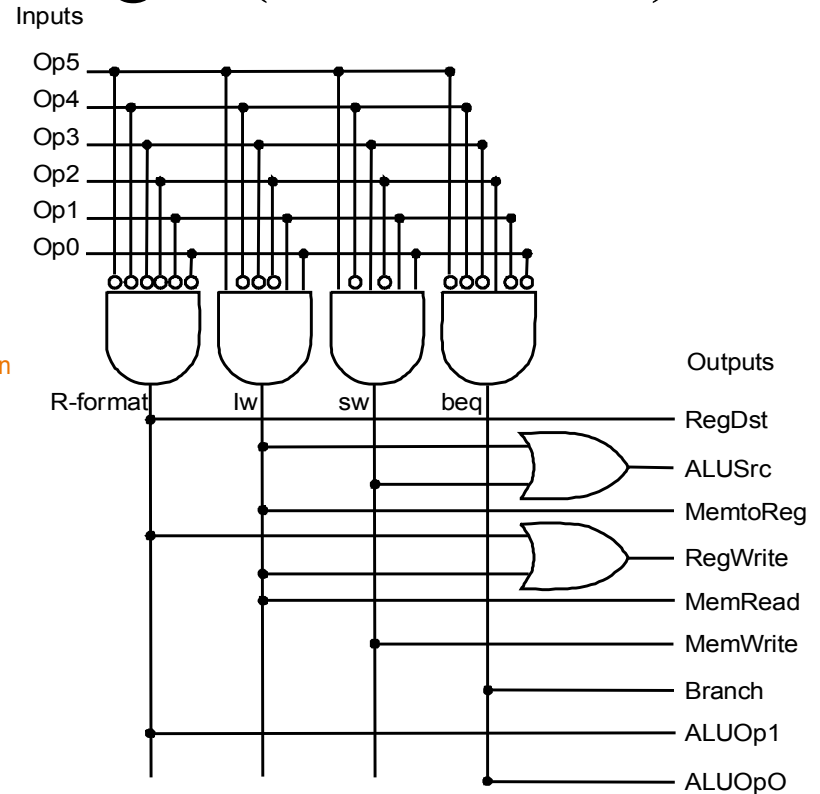
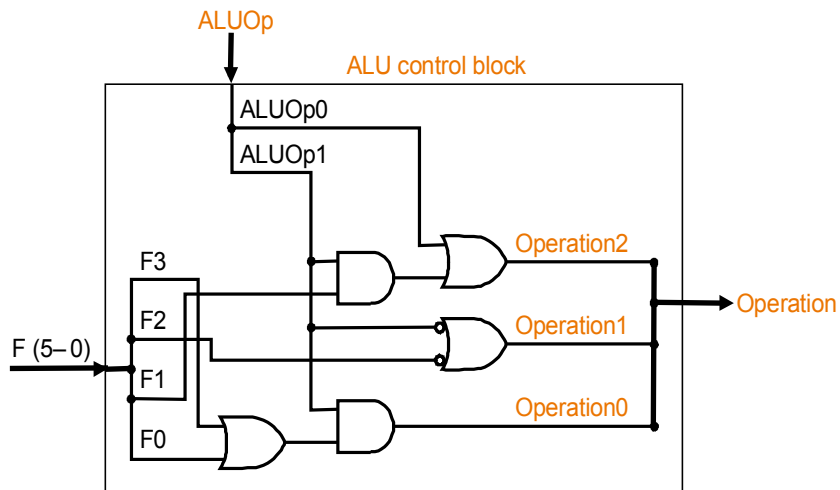
Control

- Simple combinational logic (truth tables)



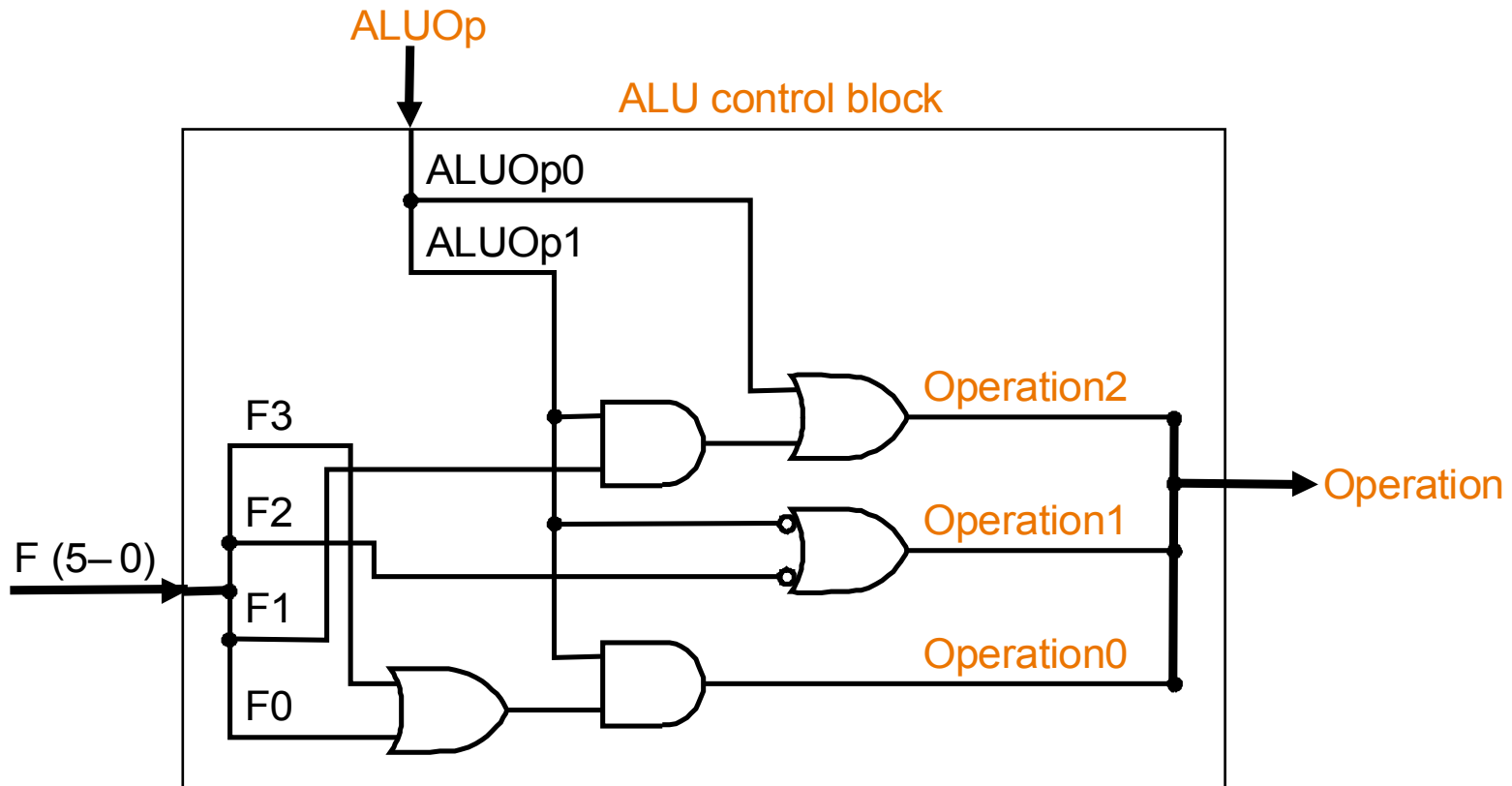
ALU & MAIN Control

- Simple combinational logic (truth tables)



Control

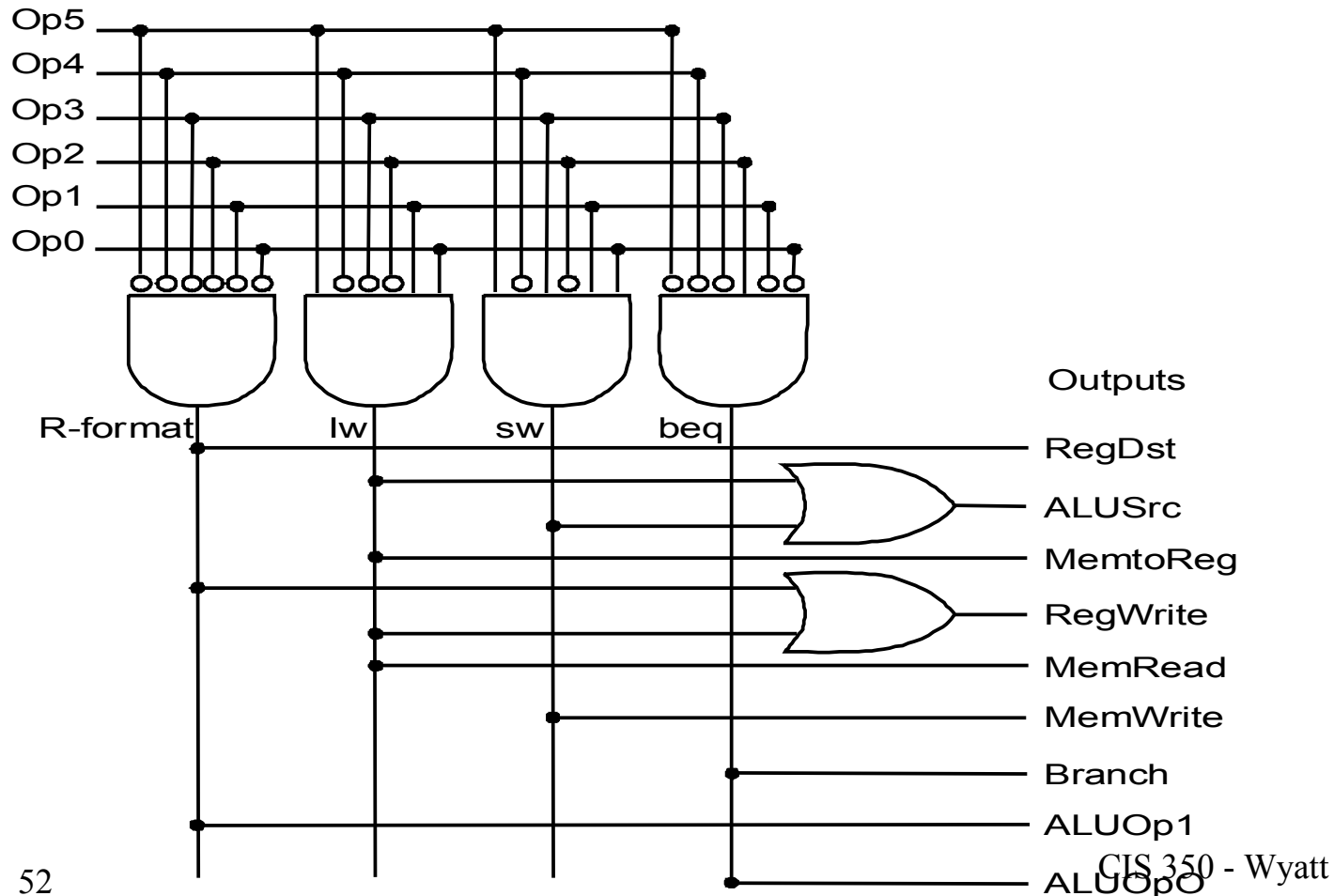
- Simple combinational logic (truth tables)



MAIN Control

- Simple combinational logic (truth tables)

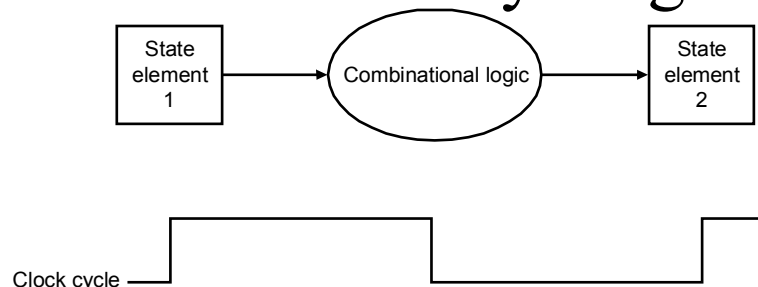
Inputs



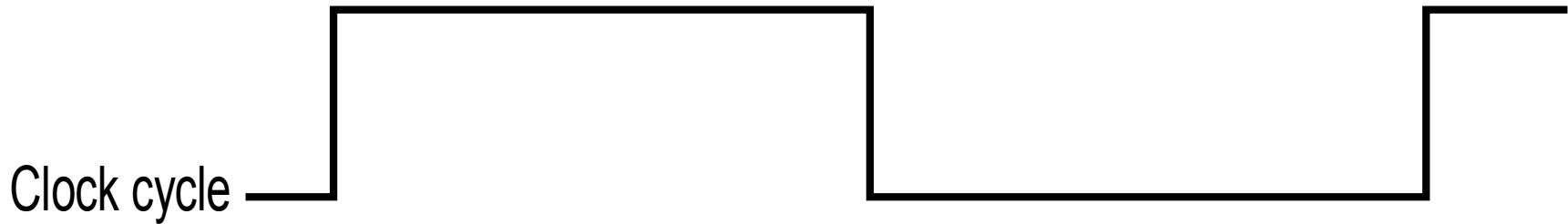
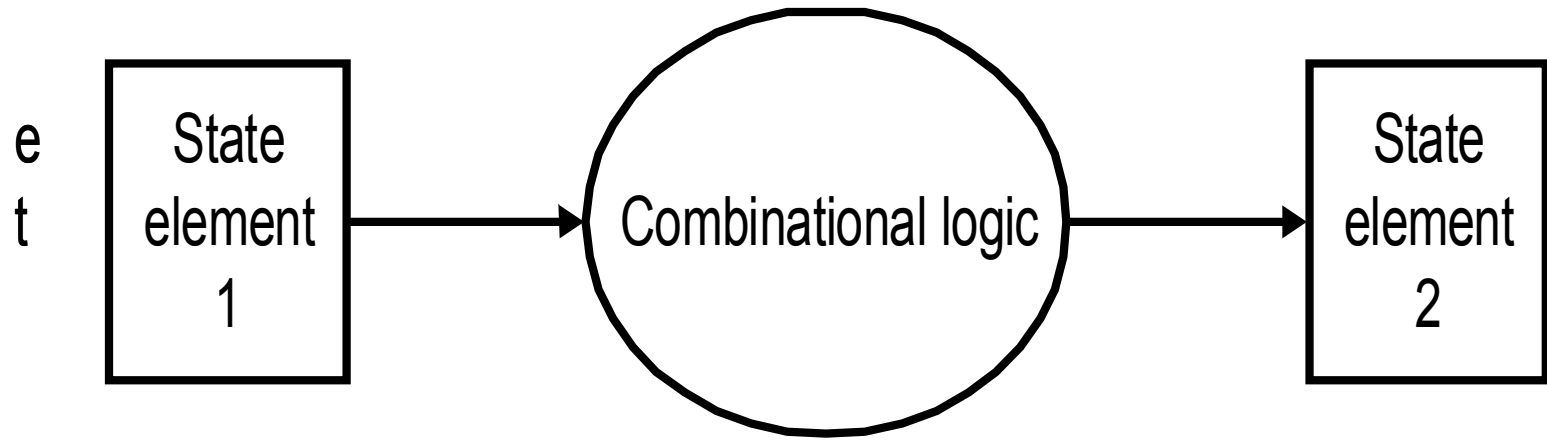
Our Simple Control Structure

- All of the logic is combinational
- We wait for everything to settle down, and the right thing to be done
- ALU might not produce “right answer” right away
- we use write signals along with clock to determine when to write
- Cycle time determined by length of the longest path

e
t

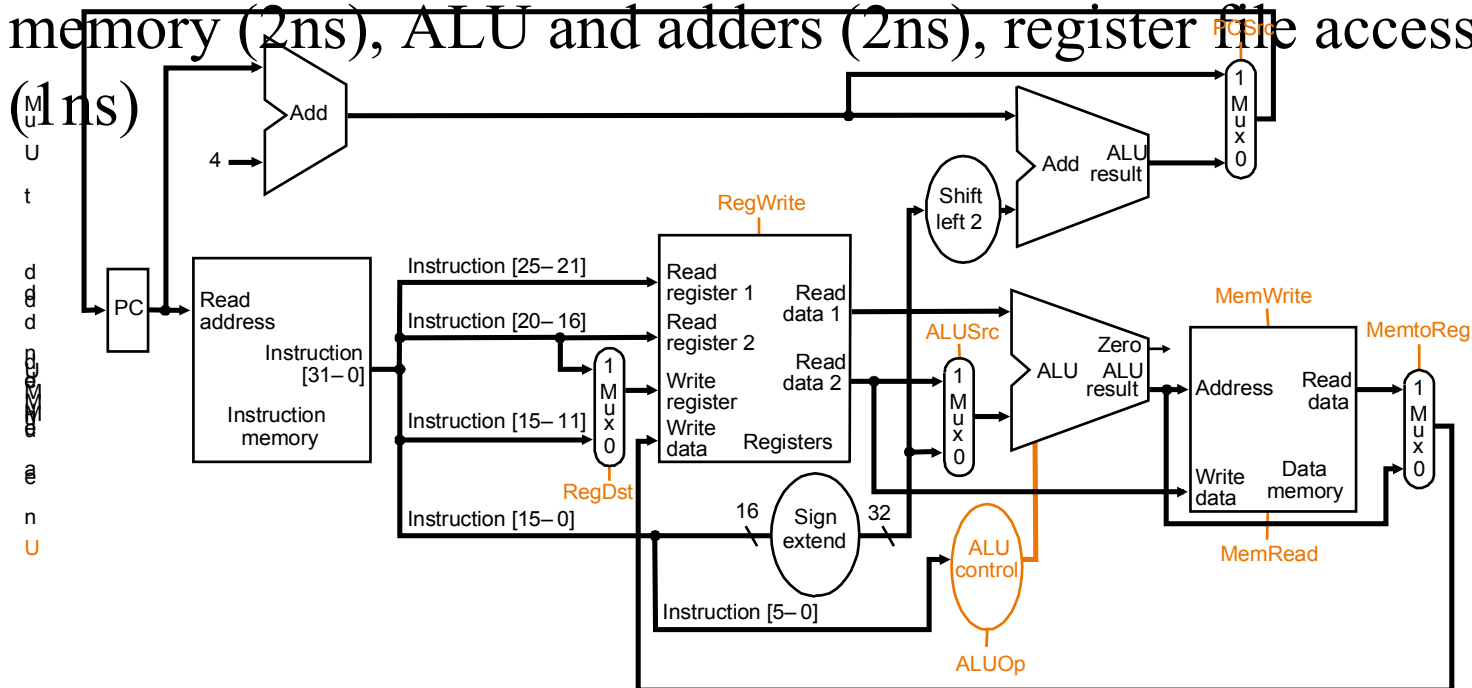


Our Simple Control Structure



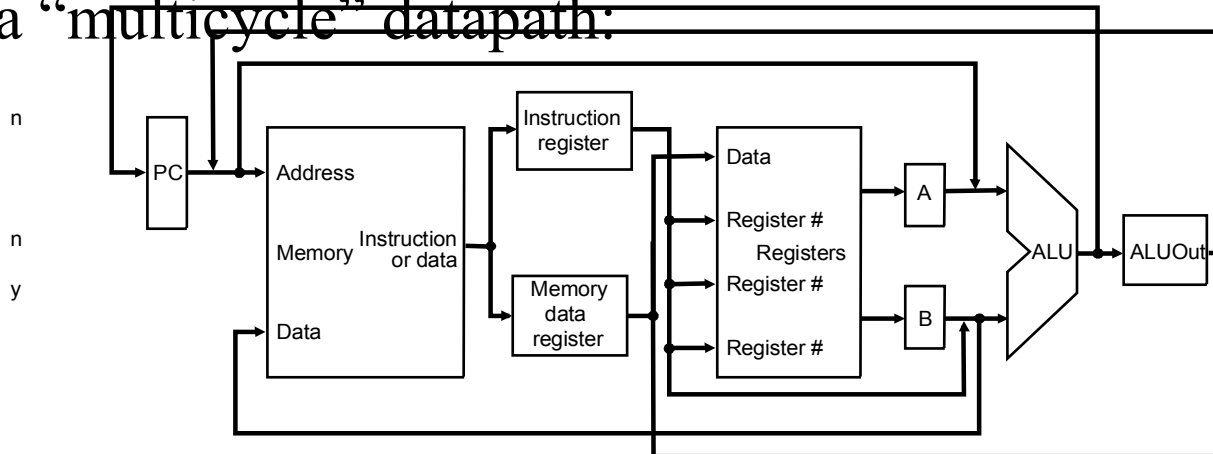
Single Cycle Implementation

- Calculate cycle time assuming negligible delays except:
- memory (2ns), ALU and adders (2ns), register file access

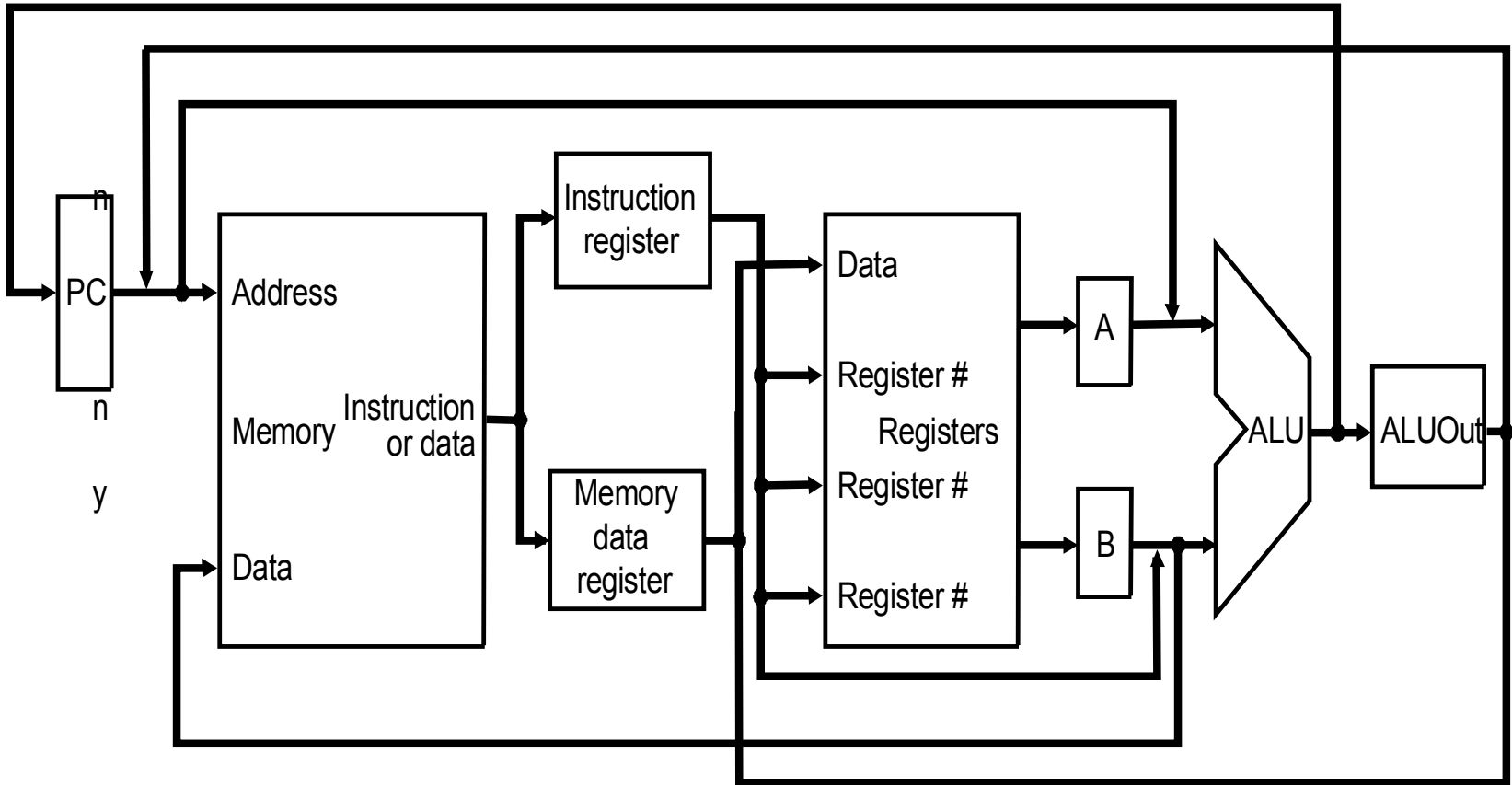


Where we are headed

- Single Cycle Problems:
 - what if we had more complicated instruction like floating point?
 - wasteful of area
- One Solution:
 - use a “smaller” cycle time
 - have different instructions take different numbers of cycles
 - a “multicycle” datapath:



Where we are headed



Multicycle Approach

- We will be reusing functional units
 - ALU used to compute address and to increment PC
 - Memory used for instruction and data
- Our control signals will not be determined solely by instruction
 - e.g., what should the ALU do for a “subtract” instruction?
- We’ll use a finite state machine for control