Binary Arithmetic 2
Intro to Assembly

CS 64: Computer Organization and Design Logic
Lecture #3

Ziad Matni
Dept. of Computer Science, UCSB

Adding this Class

- The class has about **3** spots open
- I will make the announcement tomorrow afternoon via email and on the class website
- Once these students are added, the class will then be **CLOSED** (sorry).
  - I am teaching this class next quarter as well!!!

Lecture Outline

- Two's complement
- Addition and subtraction in binary
- Multiplication in binary

- How Assembly instructions work in the CPU

Any Questions From Last Lecture?

5-Minute Pop Quiz!!!

YOU MUST SHOW YOUR WORK!!!

1. Calculate and give your answer in hexadecimal:
   a) 0x52 & 0xFE
   b) ~(0x1E | 0xCC)

2. Convert from binary to decimal AND hexadecimal. Use any technique you like:
   a) 1001001
   b) 10010010

Answers…

1. Calculate and give your answer in hexadecimal:
   a) 0x52 & 0xFE **= 0x52**
   b) ~(0x1E | 0xCC) **= ~(0xDE) = 0x21**

2. Convert from binary to decimal AND hexadecimal. Use any technique you like:
   a) 1001001 **= 100 1001 = 0x49**
   = 1 + 8 + 64 = 73
   b) 10010010 **= 1001 0010 = 0x92**
   I see that it's (1001001) x 2 = 146

Twos Complement Method

- This is how Twos Complement fixes this.
- Let's write out **-6**$_{(10)}$ in 2s-Complement binary in **4 bits**:

First take the unsigned (abs) value (i.e. 6) and convert to binary:	**0110**
Then negate it (i.e. do a "NOT" function on it):	**1001**
Now add 1:	**1010**

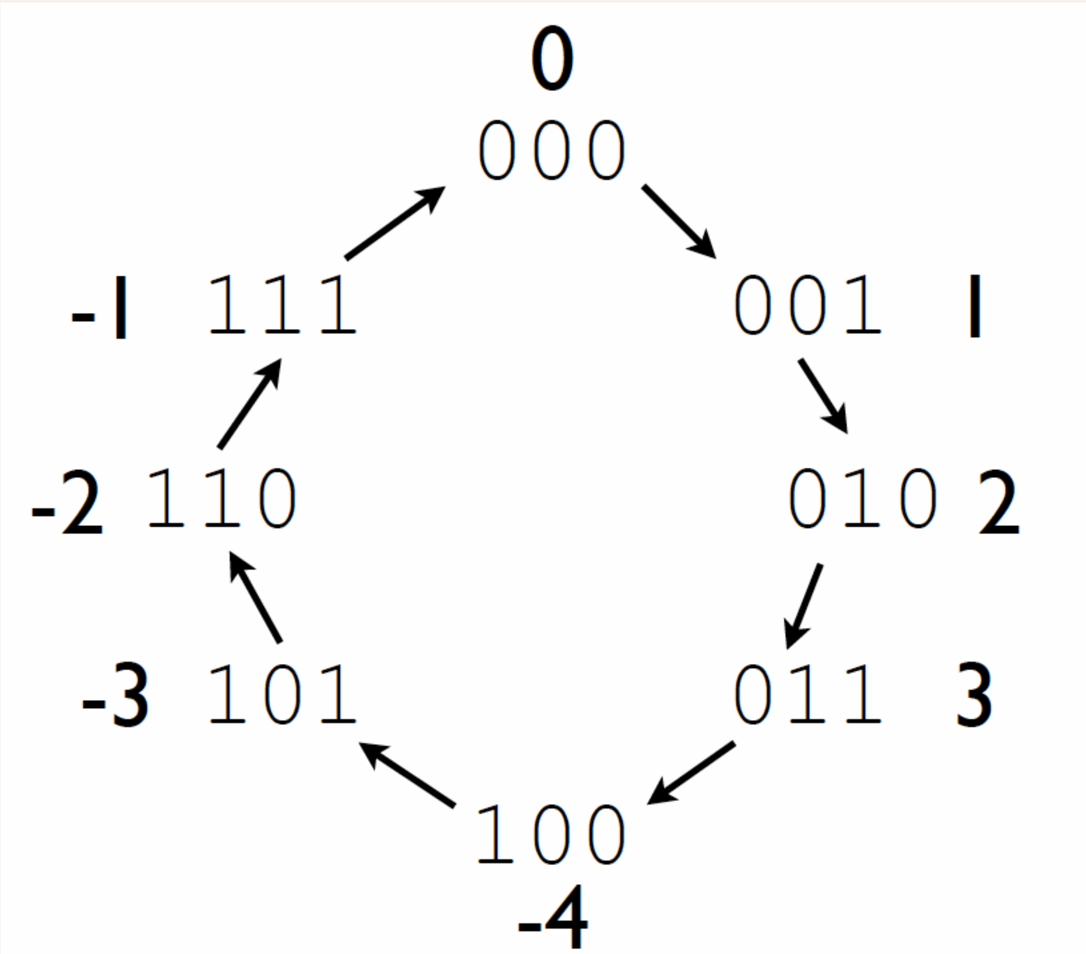
So, $-6_{(10)} = 1010_{(2)}$

Let's do it Backwards… By doing it THE SAME EXACT WAY!

- 2s-Complement to Decimal method **is the same!**

- Take **1010** from our previous example
- Negate it and it becomes **0101**
- Now add 1 to it & it becomes **0110**, which is $6_{(10)}$

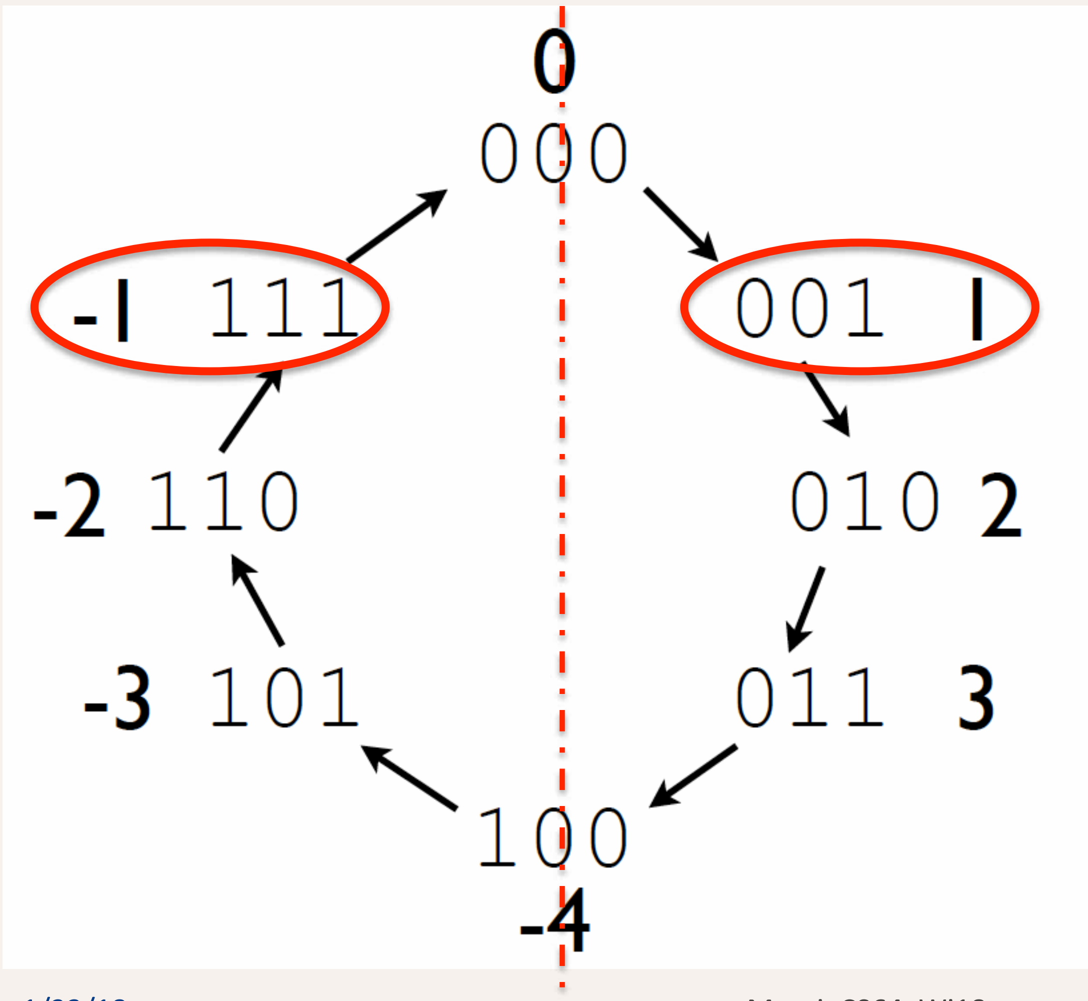
Another View of 2s Complement



NOTE:

In Two's Complement, if the number's MSB is "1", then that means it's a negative number and if it's "0" then the number is positive.

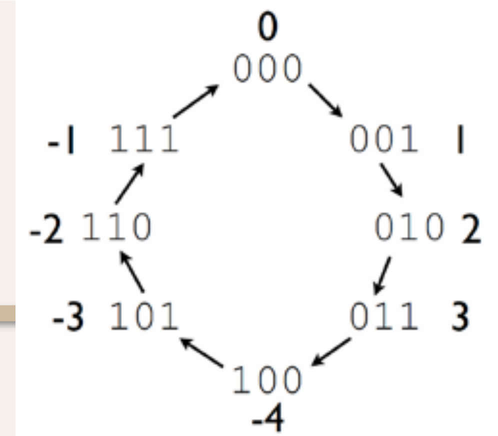
Another View of 2s Complement



NOTE:
Opposite numbers show up as symmetrically opposite each other in the circle.

NOTE AGAIN:
When we talk of 2s complement, we must also mention the number of bits involved

Ranges



- The *range* represented by number of bits differs between positive and negative binary numbers

- Given **N** bits, the range represented is:

  0 to **$+2^{N} – 1$** *for positive numbers*

and **$–2^{N-1}$** to **$+2^{N-1} – 1$**

for 2's Complement negative numbers

Addition

- We have an elementary notion of adding single digits, along with an idea of carrying digits
  - Example: when adding 3 to 9, we put forward 2 and carry the 1 (i.e. to mean 12)

- We can build on this notion to add numbers together that are more than one digit long

- Example:

$$
\begin{array}{rccc}
 & 1 & 1 & \\
 & 1 & 2 & 3 \\
+ & 3 & 8 & 9 \\
\hline
 & 5 & 1 & 2
\end{array}
$$

Addition in Binary

- Same mathematical principal applies

```
carry carry carry carry
  1     1     1     1
        0     0     1     1
  +     1     1     0     1
---------------------------
  1     0     0     0     0
```

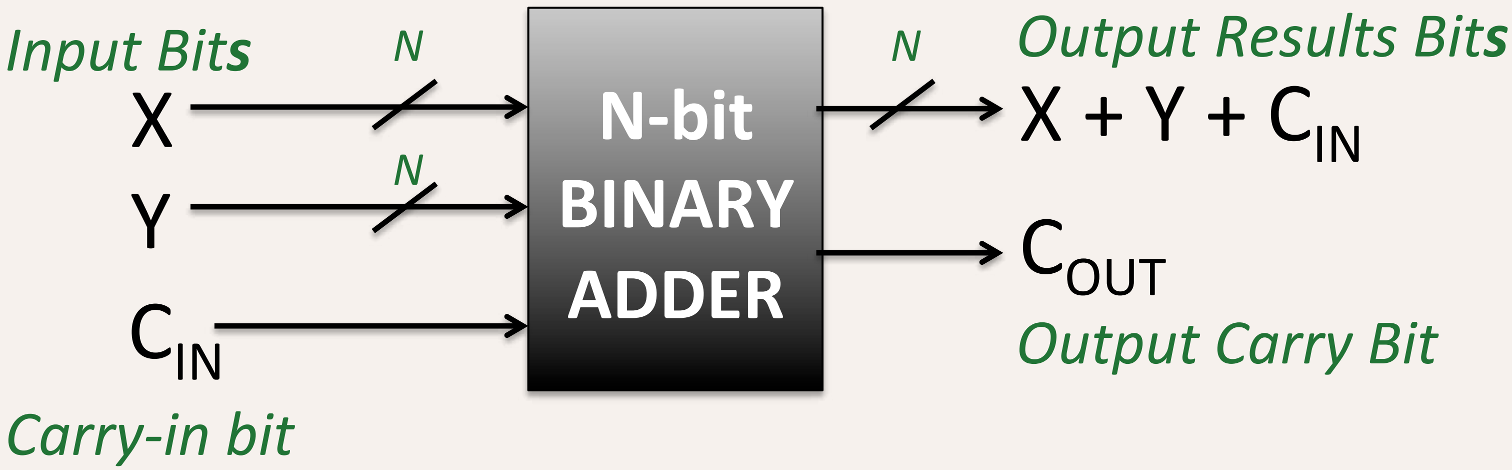
$$\begin{array}{r} 3 \\ +\ 13 \\ \hline 16 \end{array}$$

Exercises

Implementing an 8-bit adder:

- What is (0x52) + (0x4B) ?
  - Ans: 0x9D, output carry bit = 0
- What is (0xCA) + (0x67)?
  - Ans: 0x31, output carry bit = 1

Black Box Perspective of ANY N-Bit Binary Adder



This is an extremely useful perspective for either writing an N-bit adder function in code, or for designing the actual digital circuit that does this!

Output Carry Bit Significance

- For unsigned (i.e. positive) numbers, C_{OUT} indicates if the result **did not fit all the way into the number of bits allotted**

- Could be used as an error condition for software
  - For example, **you've designed a 16-bit adder** and during some calculation of positive numbers, your carry bit/flag goes to "1". Conclusion?
  - Your result is ***outside the maximum range allowed by 16 bits.***

Carry vs. Overflow

- The **carry** bit/flag works for – and is looked at – only for ***unsigned (positive)*** numbers

- A similar bit/flag works is looked at for if ***signed*** (two's complement) numbers are used in the addition: the **overflow** bit

Overflow: for Negative Number Addition

- What about if I'm adding two ***negative*** numbers? Like: 1001 + 1011?
  - Then, I get: 0100 with the extra bit set at 1
  - Sanity Check:
    That's adding (-7) + (-5), so I expected -12, which is beyond the capability of 4 bits in 2's complement!

- The extra bit in this case is called **overflow** and it indicates that the addition of negative numbers has resulted in a number ***beyond the range of the given bits*.**

How Do We Determine if Overflow Has Occurred?

- When adding 2 *signed* numbers: $\boldsymbol{x + y = s}$

if $\mathbf{x, y > 0}$ AND $\mathbf{s < 0}$

OR if $\mathbf{x, y < 0}$ AND $\mathbf{s > 0}$

Then, overflow has occurred

Example 1

Add: -39 and 92 in 8-bit binary

```
-39     1101 1001
 92     0101 1100
---     ---------
 53    10011 0101
```

$\underbrace{0011\ 0101}$ *That's 53 in signed 8-bits!*

There's a carry-out (we don't care)

But there is no overflow

Side-note:
What is the range of signed numbers w/ 8 bits?

-2^7 to $2^7 - 1$, or
-128 to 127

Example 2

Add: 104 and 45 in 8-bit binary

```
104      0110 1000
 45      0010 1101
---      ---------
149      1001 0101
```

$\underbrace{\texttt{1001 0101}}_{\textit{That's NOT 149 in signed 8-bits!}}$

Side-note:
What is the range of signed numbers w/ 8 bits?

-2^7 to $2^7 - 1$, or
-128 to 127

There's no carry-out (again, we don't care)

But there **is** overflow! Given that this binary result is not 149, but actually **–107** !

Multiplication

- More complicated than addition…
  - Unless it's just "multiply by a power of 2"!!

- We'll only assume positive numbers
- Look at just one of many algorithms that can do this…

Central Idea

- Accumulate a partial product: the result of the multiplication as we go on
- Computed via a series of additions
- When we are finished, the partial product becomes the final product (the result)
- Build off of addition and multiplication of a single digit (much like with addition)

Decimal Algorithm

- Let **P** be the partial *product*, **M** be the *multiplicand*, and **N** be the *multiplier*
  - *i.e. P eventually will be M * N*

- Initially, P is 0
- If N is 0, then P = the result
- If not, then P += (the rightmost digit of N) times M
- Shift N <u>right</u> once, and M <u>left</u> once
- Repeat

Example with Decimals

803 * 151 *(which we expect to be 121,253)*

P	M	N
0	803	151

1. N is not 0
2. P **+=** (rightmost digit of $N_{[1]}$) * $M_{[803]}$
   Shift N right once, M left once
   N is not 0
3. P **+=** (rightmost digit of $N_{[5]}$) * $M_{[8030]}$
   Shift N right once, M left once
   N is not 0
4. P **+=** (rightmost digit of $N_{[1]}$) * $M_{[80300]}$
   Shift N right once, M left once
   N IS 0 ; END

Example with Decimals

803 * 151 *(which we expect to be 121,253)*

P	M	N
0	803	151
803	8030	15
40953	80300	1
121253	803000	0

1. N is not 0

2. P **+=** (rightmost digit of N[1]) * M[803]
   Shift N right once, M left once
   N is not 0
3. P **+=** (rightmost digit of N[5]) * M[8030]
   Shift N right once, M left once
   N is not 0
4. P **+=** (rightmost digit of N[1]) * M[80300]
   Shift N right once, M left once
   N IS 0 ; END

Simplified Binary Version of the Multiplication Algorithm

- In binary, it's easier to implement

- Initially, P is 0
- If N is 0, then P = the result
- If not, then P += (the rightmost digit of N) times M
- Shift N <u>right</u> once, and M <u>left</u> once
- Repeat

Simplified Binary Version of the Multiplication Algorithm

- In binary, it's easier to implement
- Initially, P is 0
- If N is 0, then P = the result
- If not, then P += (the rightmost digit of N) times M
  - "rightmost digit of N" is going to be either:
    0 (so, P doesn't increment anything), or **1** (P increment by M)
- Shift N <u>right</u> once, and M <u>left</u> once
- Repeat

Simplified Binary Version of the Multiplication Algorithm

- In binary, it's easier to implement

- Initially, P is 0
- If N is 0, then P = the result
- If the rightmost digit if N is 1, then P += M
- Shift N right once, and M left once
- Repeat

ASSEMBLY

The Simple Language of a CPU

- We have: variables, integers, addition, and assignment

- Restrictions:
  - Can only assign integers directly to variables (not indep.)
  - Can only add variables, always two at a time

EXAMPLE:

z = 5 + 7; has to be simplified to:

x = 5;

y = 7;

z = x + y;

What's needed to implement this?

←←←

An adder: but how many bits?

Core Components

What we need in a CPU is:

- Some place to hold the statements (instructions to the CPU) as we operate on them
- Some *place* to tell us *which statement* is next
- Some *place* to hold all the *variables*
- Some *way* to add (do arithmetic on) *numbers*

That's ALL that Processors Do!!

Processors just read a series of statements (instructions) forever. No magic!

Core Components

What we need in a CPU is:

- Some place to hold the statements (instructions to the CPU) as we operate on them → MEMORY
- Some *place* to tell us *which statement* is next → PROGRAM COUNTER (PC)
- Some *place* to hold all the *variables* → REGISTERS
- Some *way* to add (do arithmetic on) *numbers* → ARITHMETIC LOGIC UNIT (ALU)

…And one more thing:

- Some place to tell us which statement is **currently** being executed → INSTRUCTION REGISTER (IR)

Basic Interaction

- Copy instruction from **memory** at wherever the **program counter (PC)** says into the **instruction register (IR)**

- Execute it, possibly involving registers and the **arithmetic logic unit (ALU)**

- Update the **PC** to point to the next instruction

- Repeat

```
initialize();
while (true) {
    instruction_register =
        memory[program_counter];
    execute(instruction_register);
    program_counter++;
}
```

Instruction Register

?

Registers

```
x:  ?
y:  ?
z:  ?
```

Program Counter

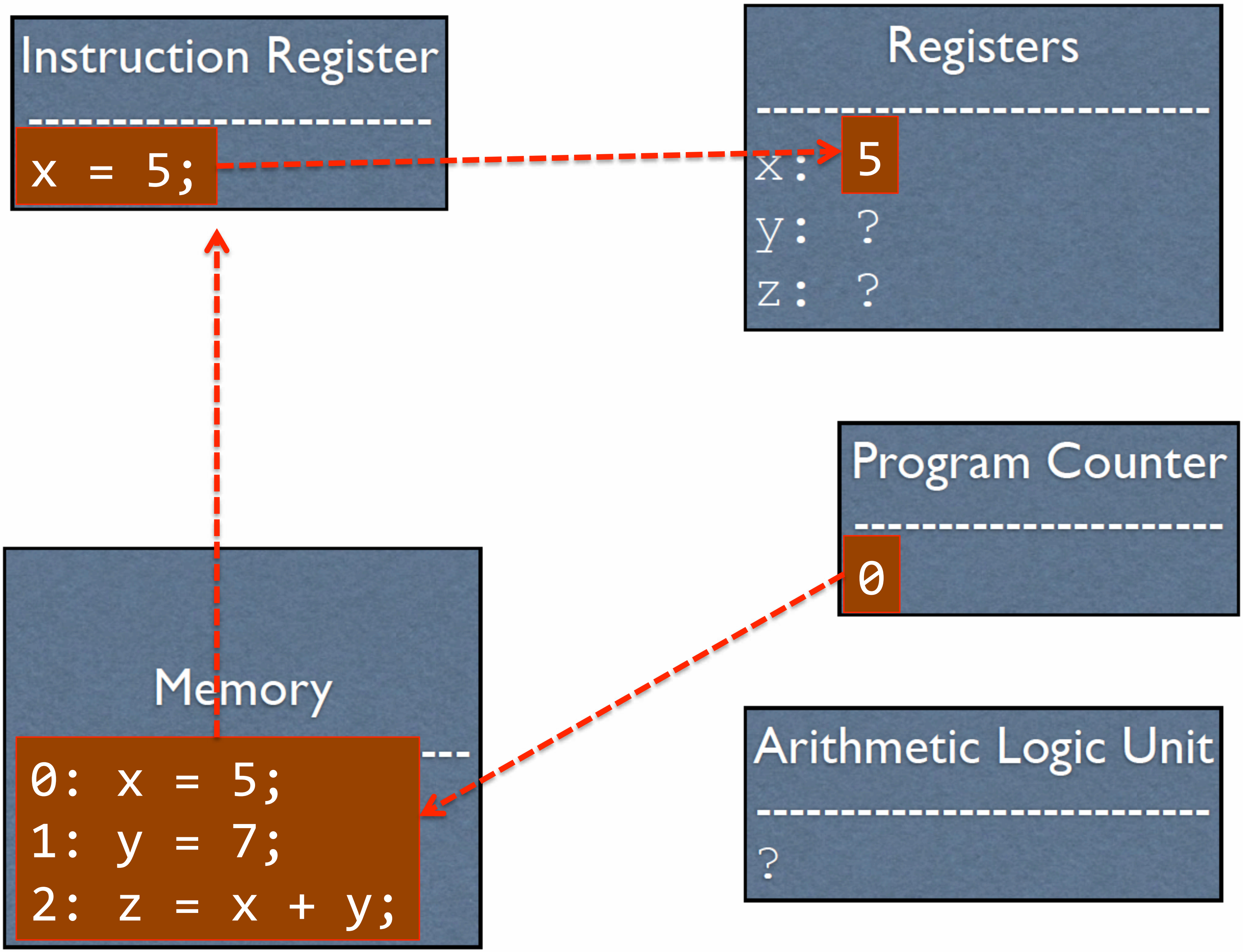
?

Memory

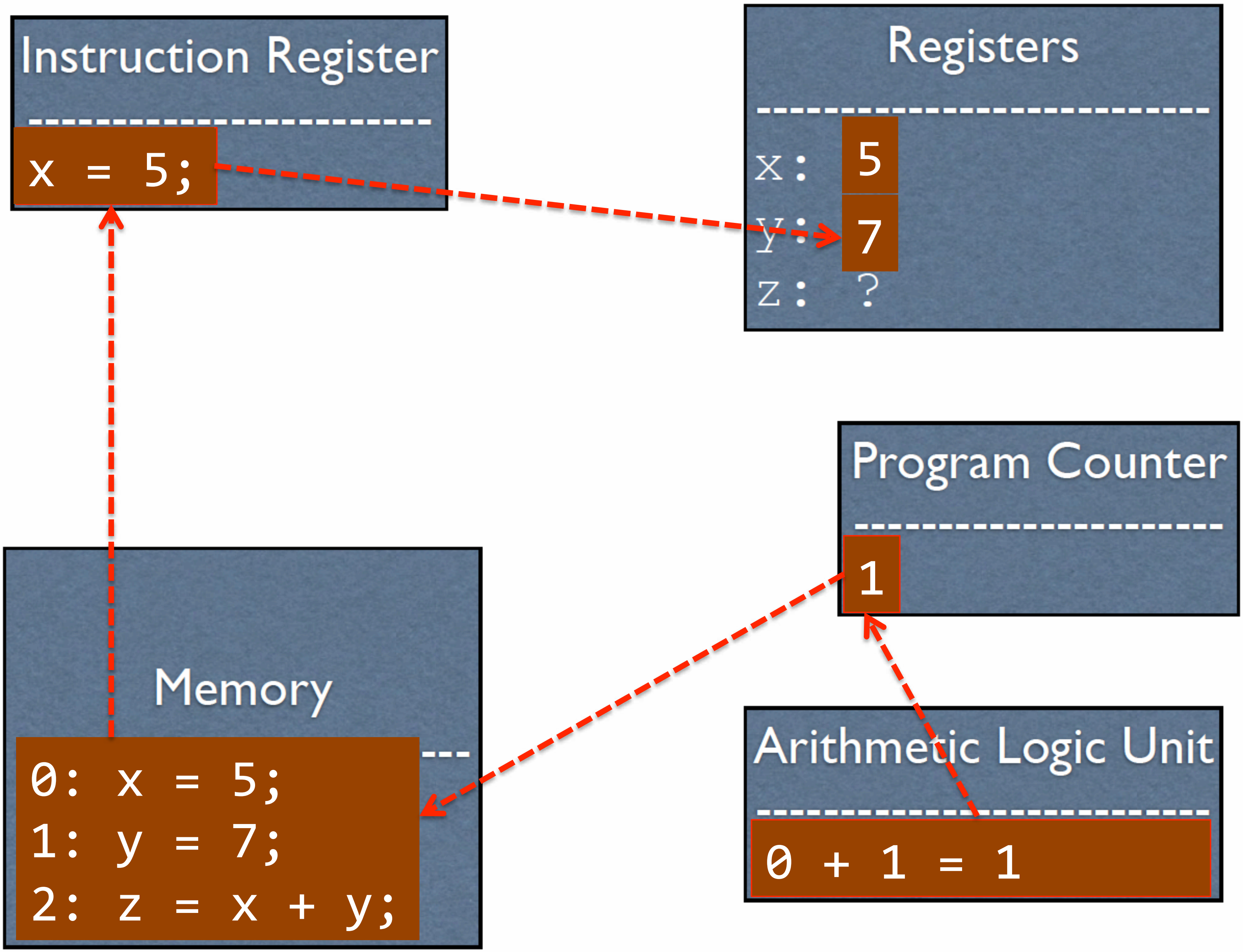
?

Arithmetic Logic Unit

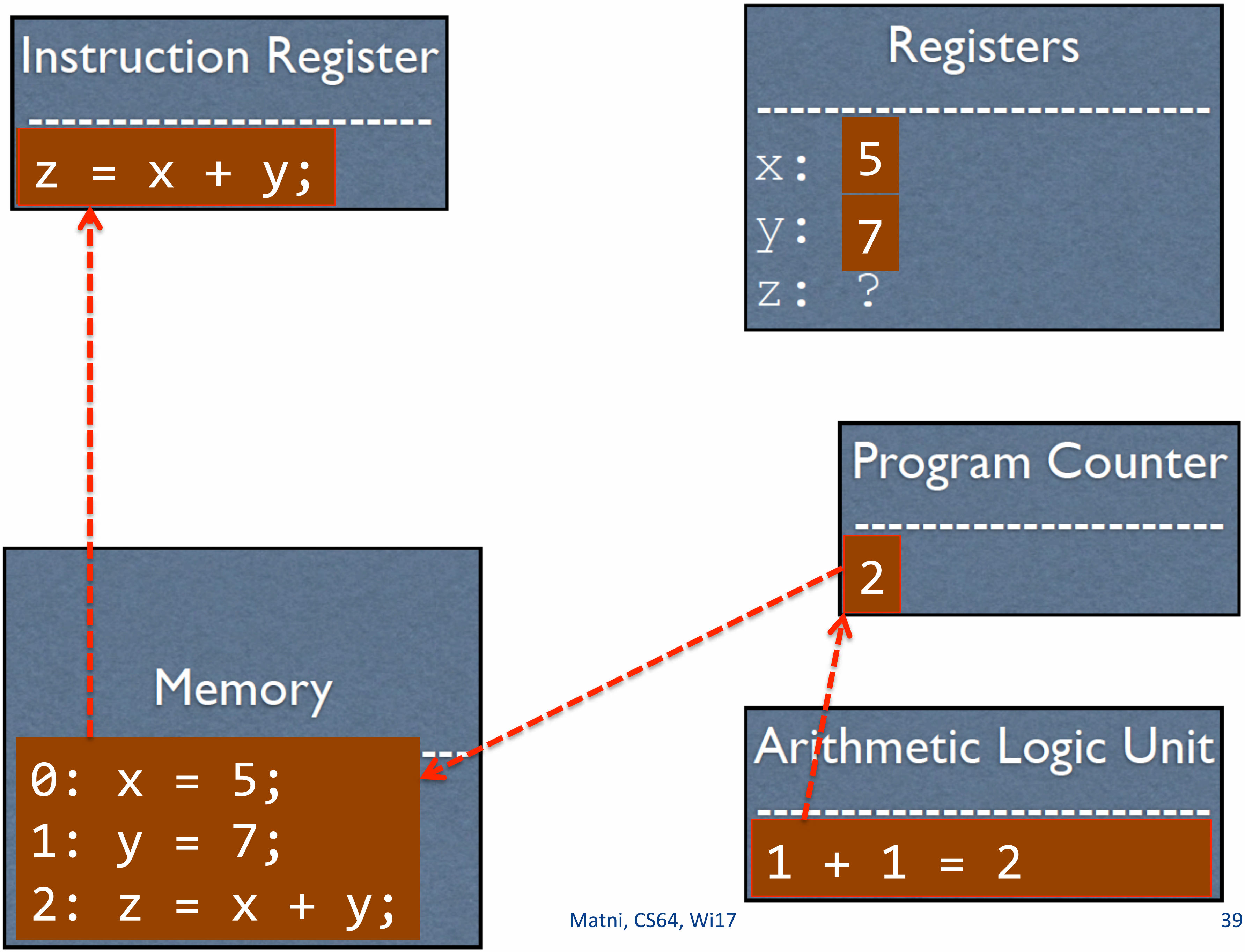
?

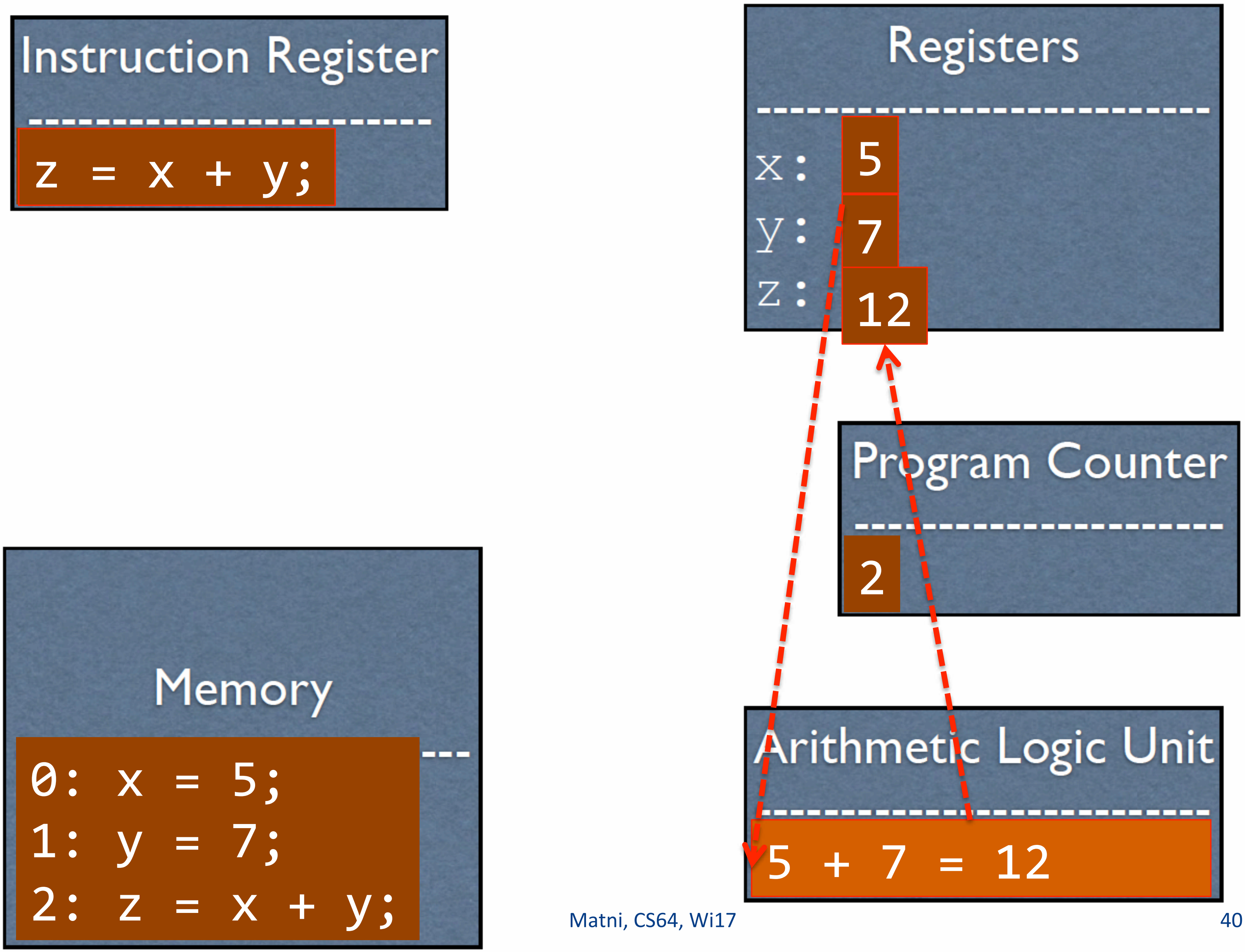


Instruction Register
x = 5;
Registers
x: 5
y: ?
z: ?
Program Counter
0
Memory
0: x = 5;
1: y = 7;
2: z = x + y;
Arithmetic Logic Unit
?



Instruction Register
x = 5;
Registers
x: 5
y: 7
z: ?
Program Counter
1
Memory
0: x = 5;
1: y = 7;
2: z = x + y;
Arithmetic Logic Unit
0 + 1 = 1


Instruction Register
z = x + y;
Registers
x: 5
y: 7
z: ?
Program Counter
2
Memory
0: x = 5;
1: y = 7;
2: z = x + y;
Arithmetic Logic Unit
1 + 1 = 2


Instruction Register
z = x + y;
Registers
x: 5
y: 7
z: 12
Program Counter
2
Memory
0: x = 5;
1: y = 7;
2: z = x + y;
Arithmetic Logic Unit
5 + 7 = 12

YOUR TO-DOs

- Assignment #2
  - Go to lab Thursday
    (look for the assignment online on Wednesday)
  - Due on Friday, 1/26, by 11:59 PM
  - Submit it via **turnin**

</LECTURE>