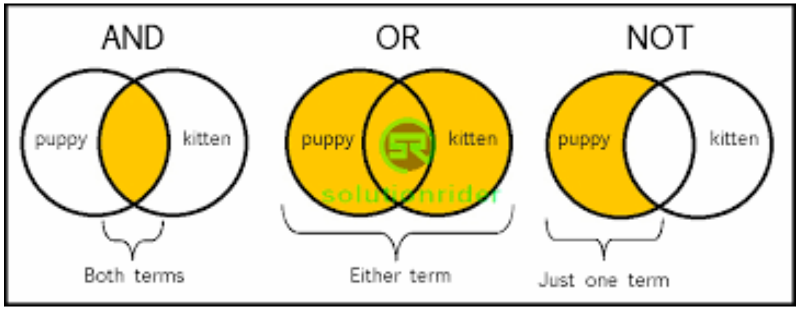


Logic Operations on Binaries
Intro to MIPS

CS 64: Computer Organization and Design Logic

Lecture #3

Fall 2020

Ziad Matni, Ph.D.
Dept. of Computer Science, UCSB

Why do CPU programmers celebrate Christmas and Halloween on the same day?

Because Oct-31 = Dec-25 !!!

Administrative Stuff

- Assignment 1 is due on Tuesday on Gradescope
  - How was lab on Thursday?

- Assignment 2 will be issued soon

- Reminder: No class next week Monday (Uni. Holiday)

Any Questions From Last Lecture?

Practice on Binary Addition, etc…

See board…

- Addition
- Subtraction
- Carry Out (C)
- Overflow (V)

Binary Logic Refresher
NOT, AND, OR

X	NOT X $\overline{X}$
0	1
1	0

X	Y	X AND Y X && Y X.Y
0	0	0
0	1	0
1	0	0
1	1	1

X	Y	X OR Y X || Y X + Y
0	0	0
0	1	1
1	0	1
1	1	1

Binary Logic Refresher
Exclusive-OR (XOR)

The output is "1" only if the inputs are opposite

X	Y	X XOR Y $X \oplus Y$
0	0	0
0	1	1
1	0	1
1	1	0

Bitwise NOT

- Similar to logical NOT (!), except it works on a bit-by-bit manner
- In C/C++, it's denoted by a tilde: ~

```
~(1001) = 0110
```

Exercises

- Remember: hexadecimal numbers are often written in the **0xhh** notation, so for example:

The hex 3B would be written as **0x3B**

- What is ~(0x04)?
  - Ans: 0xFB

- What is ~(0xE7)?
  - Ans: 0x18

Bitwise AND

- Similar to logical AND (&&), except it works on a bit-by-bit manner

- In C/C++, it's denoted by a single ampersand: &

```
(1001 & 0101) = 1 0 0 1
              & 0 1 0 1

              = 0 0 0 1
```

Exercises

- What is (0xFF) & (0x56)?
  - Ans: 0x56
- What is (0x0F) & (0x56)?
  - Ans: 0x06
- What is (0x11) & (0x56)?
  - Ans: 0x10
- Note how & can be used as a “masking” function
  - Masking??! What’s being “masked”???

Bitwise OR

- Similar to logical OR (||), except it works on a bit-by-bit manner

- In C/C++, it's denoted by a single pipe: |

```
(1001 | 0101)    = 1 0 0 1
                 | 0 1 0 1

                 = 1 1 0 1
```

Exercises

- What is (0xFF) | (0x92)?
  - Ans: 0xFF
- What is (0xAA) | (0x55)?
  - Ans: 0xFF
- What is (0xA5) | (0x92)?
  - Ans: 0xB7

Bitwise XOR

- Works on a bit-by-bit manner
- In C/C++, it's denoted by a single carat: ^

```
(1001 ^ 0101) =  1 0 0 1
              ^  0 1 0 1

              =  1 1 0 0
```

Exercises

- What is (0xA1) ^ (0x13)?
  - Ans: 0xB2

- What is (0xFF) ^ (0x13)?
  - Ans: 0xEC

- Note how (**1**^b) is always the inverse of b (**~b**) and how (**0**^b) is always just **b**

Bit Shift *Left*

- Move all the bits N positions to the left
- What do you do the positions now empty?
  - You put in N number of 0s

- Example: Shift “1001” 2 positions to the left

1001 << 2 = **100100**

- Why is this useful as a form of *multiplication*?

Multiplication by Bit Left Shifting

- Veeeery useful in CPU (ALU) design
  - Why?

- Because you don't have to design a "multiplier" function

- You just have to design a way for the bits to shift (which is a relatively easier design)

Bit Shift *Right*

- Move all the bits N positions to the ***right***, subbing-in either N number of 0s or N 1s on the left
- Takes on two different forms

- Example: Shift "1001" 2 positions to the right

1001 >> 2 = either **0010** or **1110**

- The information carried in the last 2 bits is *lost*.
- If Shift Left does *multiplication*, what does Shift Right do?
  - It divides, **but** it truncates the result

Two Forms of Shift Right

- Subbing-in 0s makes sense (esp. if the number is unsigned)

- BUT! When should we sub-in the leftmost bits with 1s?
  - ANS: When the number is signed and negative

- So what if it’s a signed number that’s positive?
  - ANS: You should sub-in the leftmost bits with 0s!

- This is called ***“arithmetic”*** shift right:

1100 (arithmetic) >> 1 = 1110

0101 (arithmetic) >> 1 = 0010

Two Forms of Shift Right

- If the number is unsigned (and thus always positive), we can use "**logical**" shift right
  - Never use this type of shift right on signed numbers…

- **Arithmetic** shift preserves sign bit

- **Logical** shift cannot/does not preserve sign bit

Exercise Using Logic Ops

- Given an argument that's a 32-bit integer number **i**, write a function in C++ that can isolate the bit in **position 5** of that integer and print it.

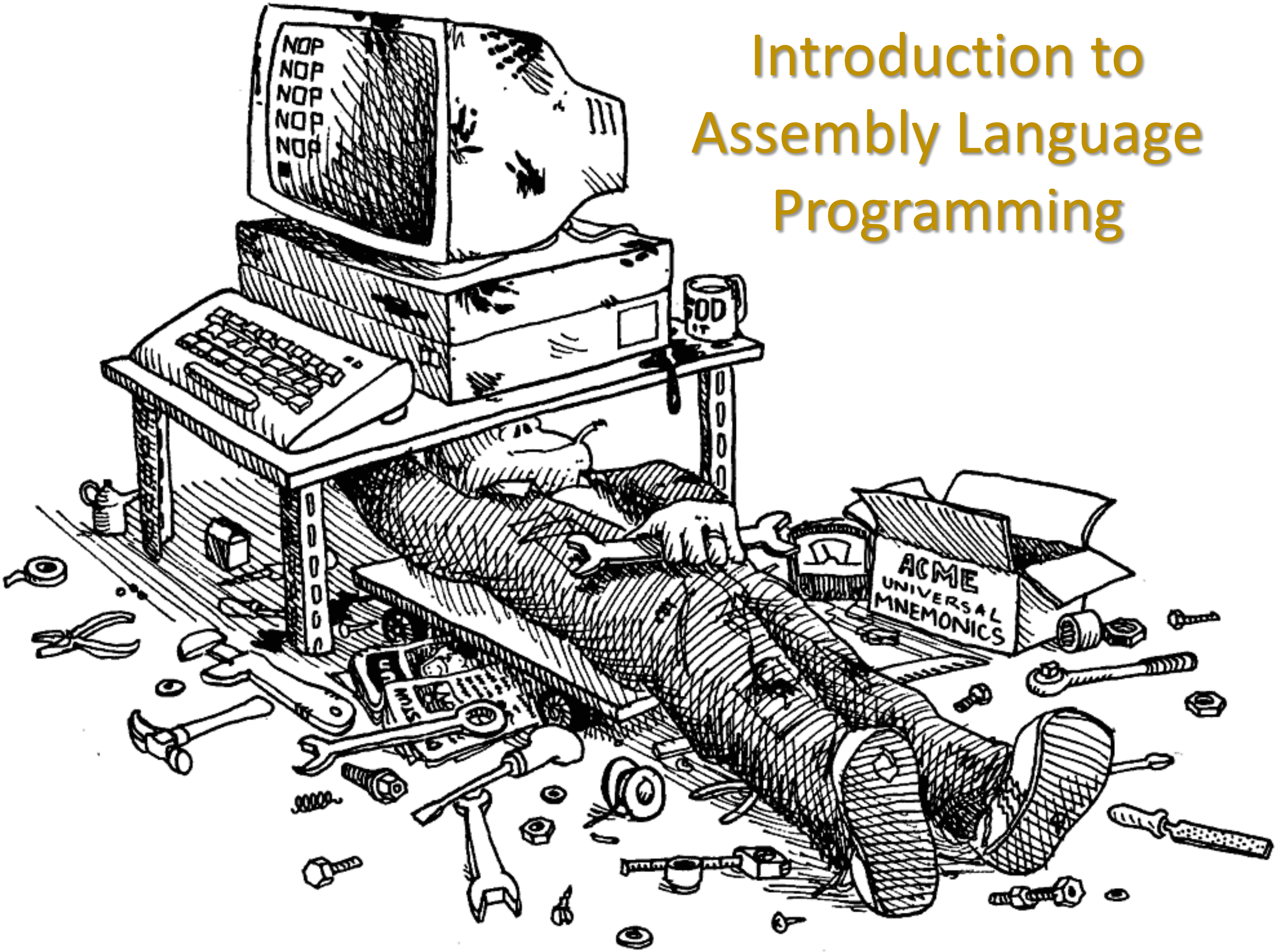
- Example: **i** = 1266
- In 32-bits of binary, that's:

  0000 0000 0000 0000 0000 0100 1111 0010

- So, the bit in position 5 is the highlighted one (it's 1)
- So your code should print out "**1**"
- Answer:

```
void print5(int i):
{
    i >> 5;
    i = i & 1;
    cout << i;
}
```

Introduction to Assembly Language Programming



The Simple Language of a CPU

- We have: variables, integers, floating points, arithmetic ops, and assignment ops

- <u>Restrictions:</u>
  - Can only assign **integers** directly to variables
  - Can only do arithmetic on (e.g. add) variables, always **two at a time** (no more)

EXAMPLE:

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

```
x = 5;
y = 7;
z = x + y;
```

What func is 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 the *variables*
- Some *way* to 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 the variables** → **REGISTERS**
- Some *way* to **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

pseudocode

```
Initialize();
while (true) {
    instruc_reg = GetFromMem[prog_countr];
    executeInstruc(instruc_reg);
    prog_countr++;
}
```

Instruction Register

?

Registers

x: ?

y: ?

z: ?

Program Counter

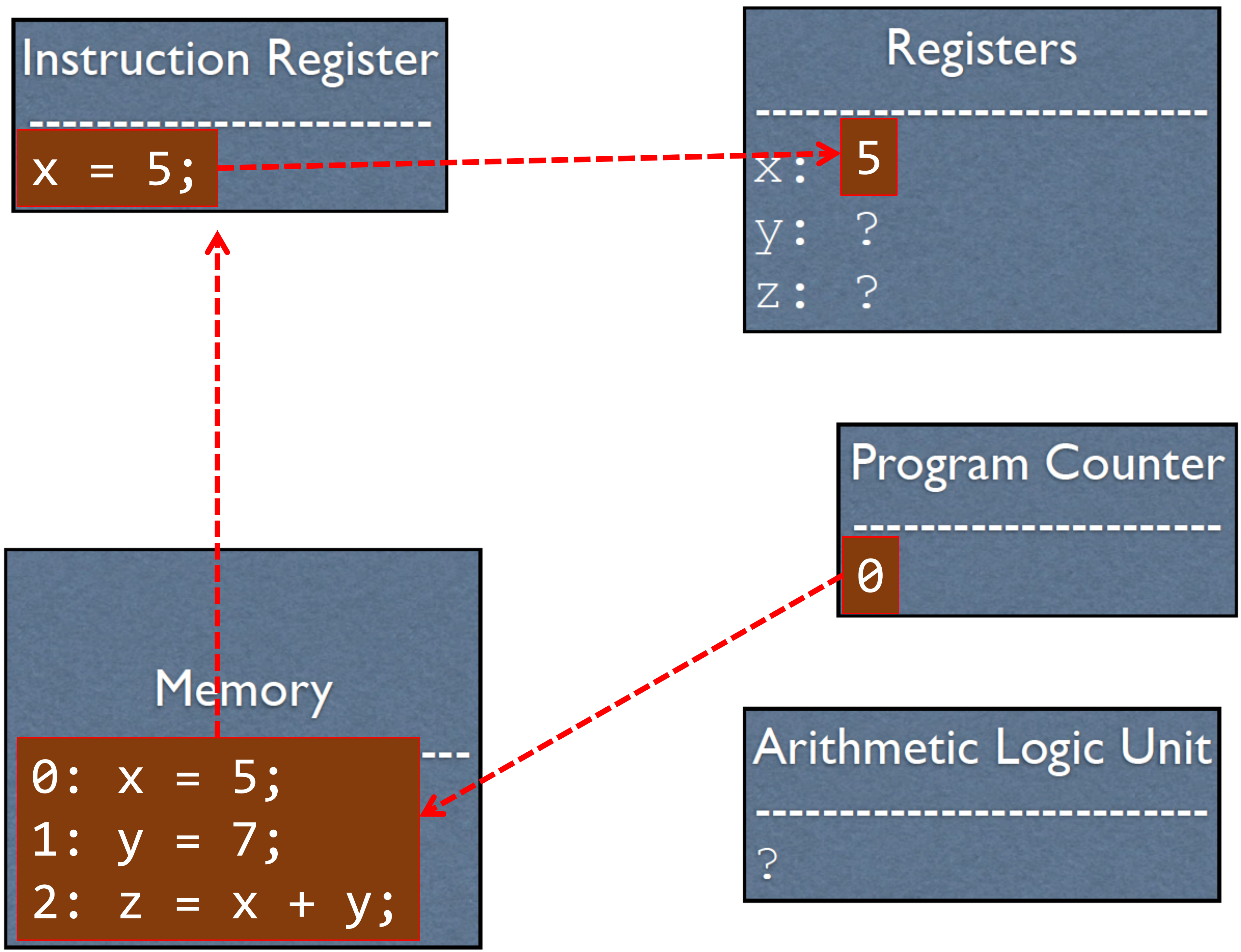
?

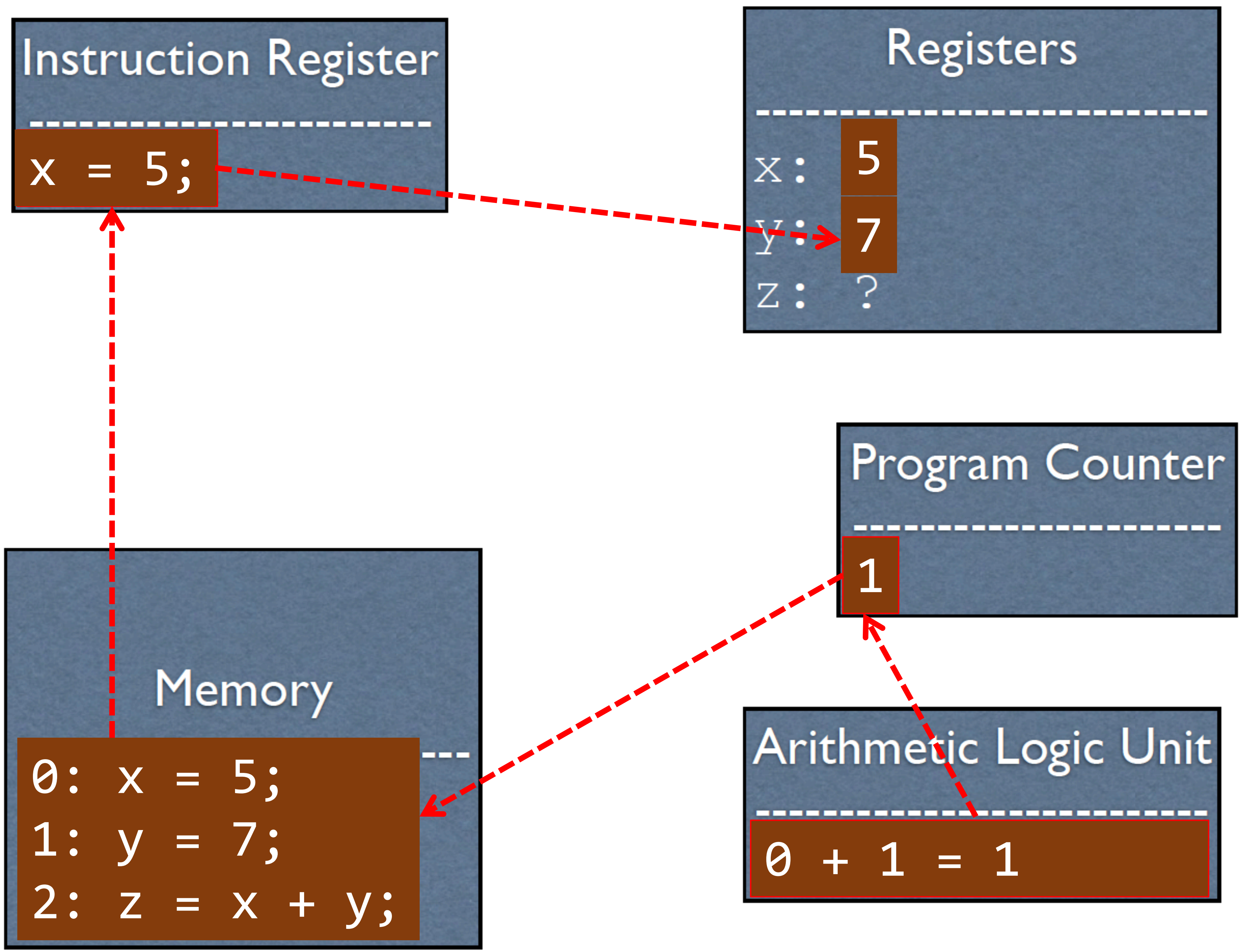
Memory

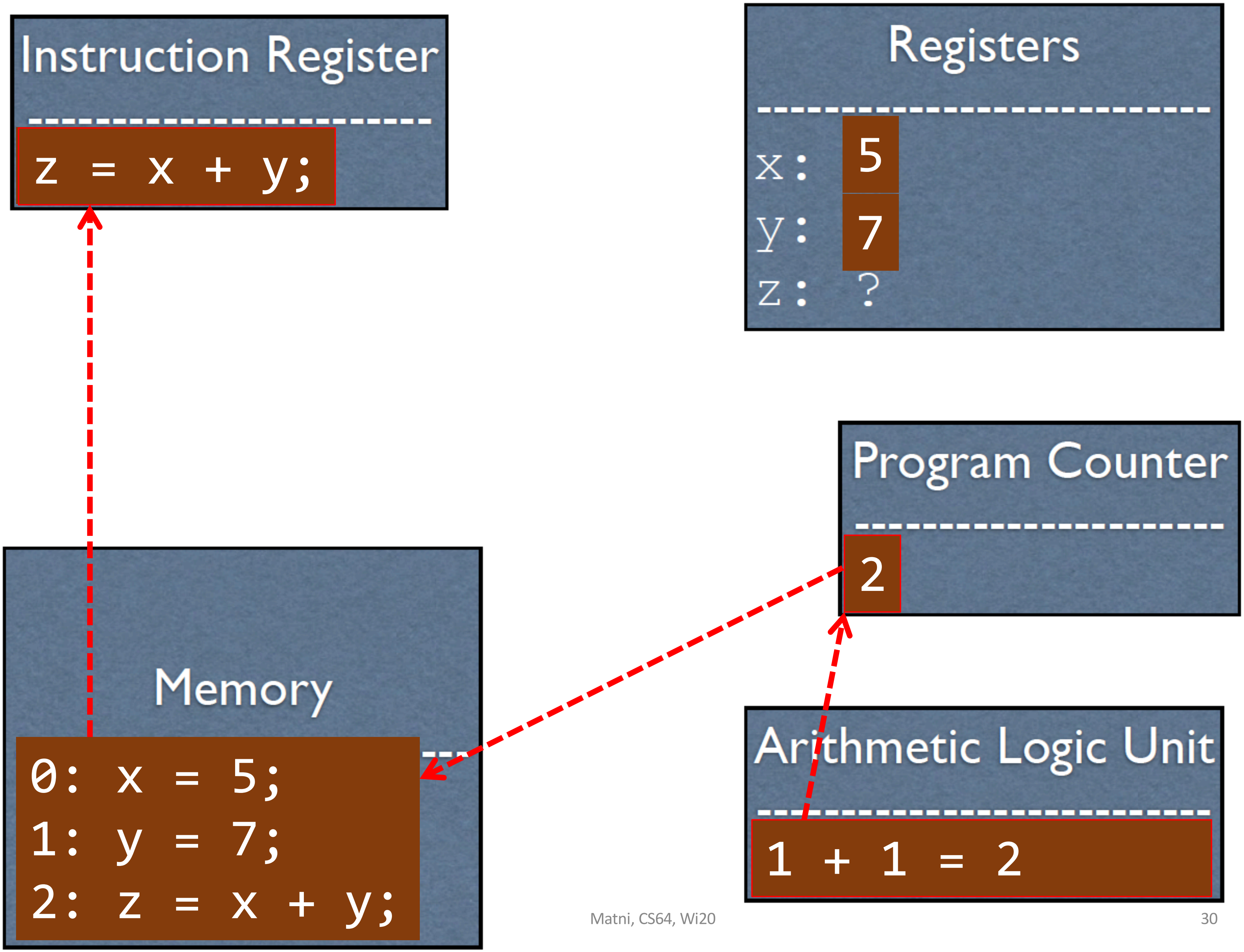
?

Arithmetic Logic Unit

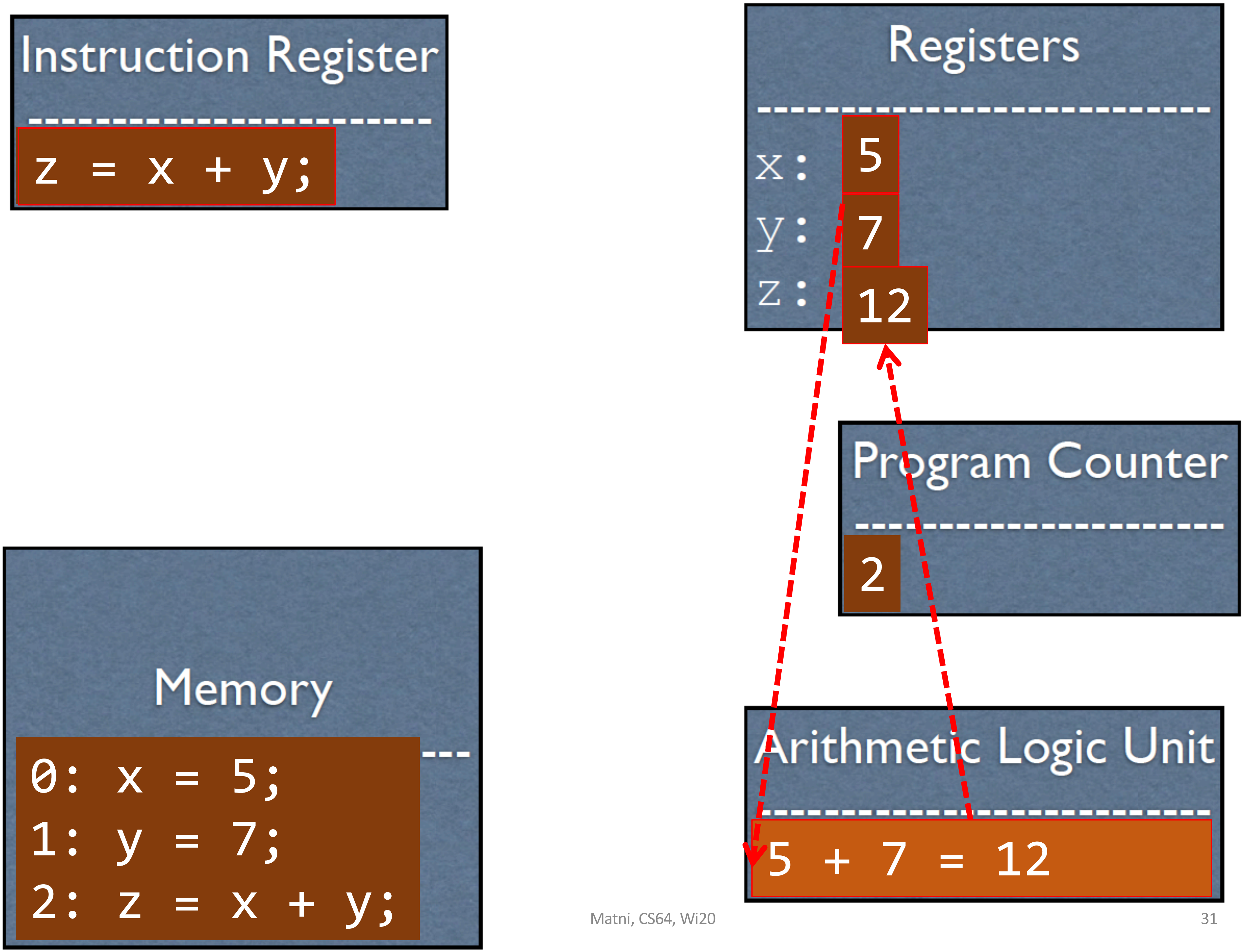
?


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

Why MIPS?

- MIPS:
  - a **r**educed **i**nstruction **s**et **c**omputer (RISC) architecture developed by a company called MIPS Technologies (1981)

- Relevant in *embedded systems*
  - An area of CS/CE

- All modern commercial processors share the same core concepts as MIPS, just with extra stuff
  - Some modern CPUs include Intel, ARM, AMD

- ...but most importantly...

MIPS is Simpler…

… than other instruction sets for CPUs

So it's a great learning tool!

- Dozens of instructions (as opposed to hundreds)
- Lack of redundant instructions or special cases
- 5 stage pipeline versus 12 stages (Intel i7 processors)

YOUR TO-DOs

- Readings! Do Them!
  - Consult syllabus…

- Finish Assignment #1
  - You have to submit it as a **PDF** using ***Gradescope***
  - Due on **Tuesday 1/14, by 11:59:59 PM**

</LECTURE>