Introduction to Assembly Language

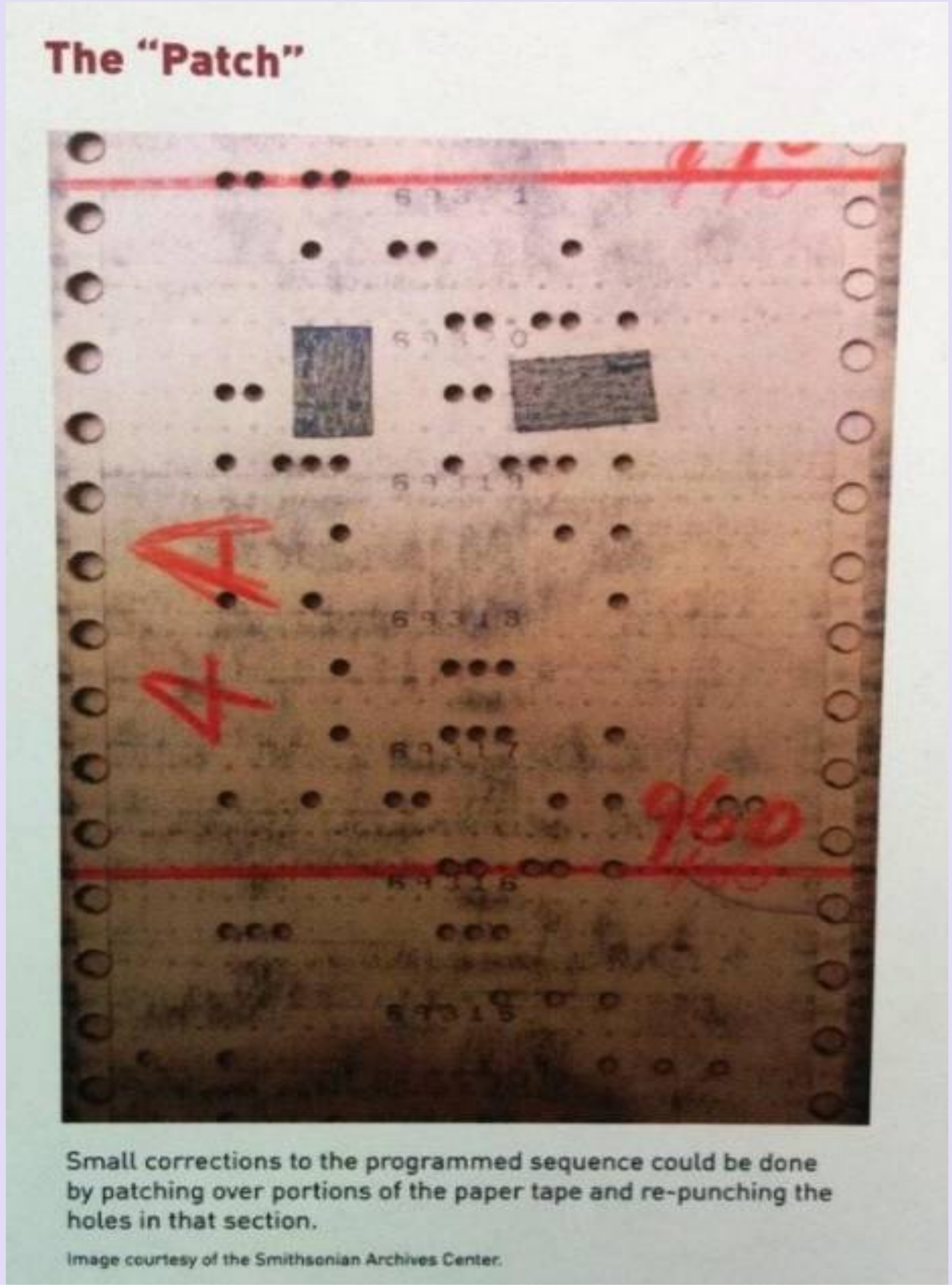
CS 64: Computer Organization and Design Logic
Lecture #4
Winter 2019

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

This Week

on

"Didja Know Dat?!"



Lecture Outline

- Review of Carry Out vs. Overflow bits
- MIPS core processing blocks
- Basic programming in assembly
- Arithmetic programs

Administrative Stuff

- How did Lab# 2 go?
  - Too easy? Too hard? Just right?
  - Remember: it's due via *turnin* on Monday!

- We will be providing assignment (lab) feedback on GauchoSpace!
  - Follow up with me/TAs during office hours

- Remember, our office hours! ☺
  - Prof. Matni — Th. 1 – 2:30 PM — *SSMS 4409*
  - TA Bay-Yuan — Fr. 11 AM – 1 PM — *Trailer 936*
  - TA Shiyu — Fr. 3 – 5 PM — *Trailer 936*

Any Questions From Last Lecture?

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, so what's wrong here?
  - The answer 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 that's
  beyond the range of the given bits**.**

How Do We Determine if Overflow Has Occurred?

- When adding 2 *signed* numbers: ***x + y = s***

if **x, y > 0** AND **s < 0**

OR if **x, y < 0** AND **s > 0**

Then, overflow has occurred

Example 1

Add: -39 and 92 in *signed* 8-bit binary

> ***Side-note:***
> What is the range of signed numbers w/ 8 bits?
>
> -2^7 to $(2^7 – 1)$, or
> -128 to 127

Cin_signed_bit → *1*

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

Cout → 1

That's 53 in signed 8-bits! Looks ok!

There's a carry-out (we don't care)
But there is no overflow (V)
Note that V = 0, while Cout = 1 and Cin_signed_bit = 1

Example 2

V = Cout ⊕ Cin_signed_bit

Add: 104 and 45 in *signed* 8-bit binary

Cin_signed_bit

```
                1
 104        0110 1000
  45        0010 1101
 ---        ---------
 149        1001 0101
```

Cout = 0

That's NOT 149 in signed 8-bits!

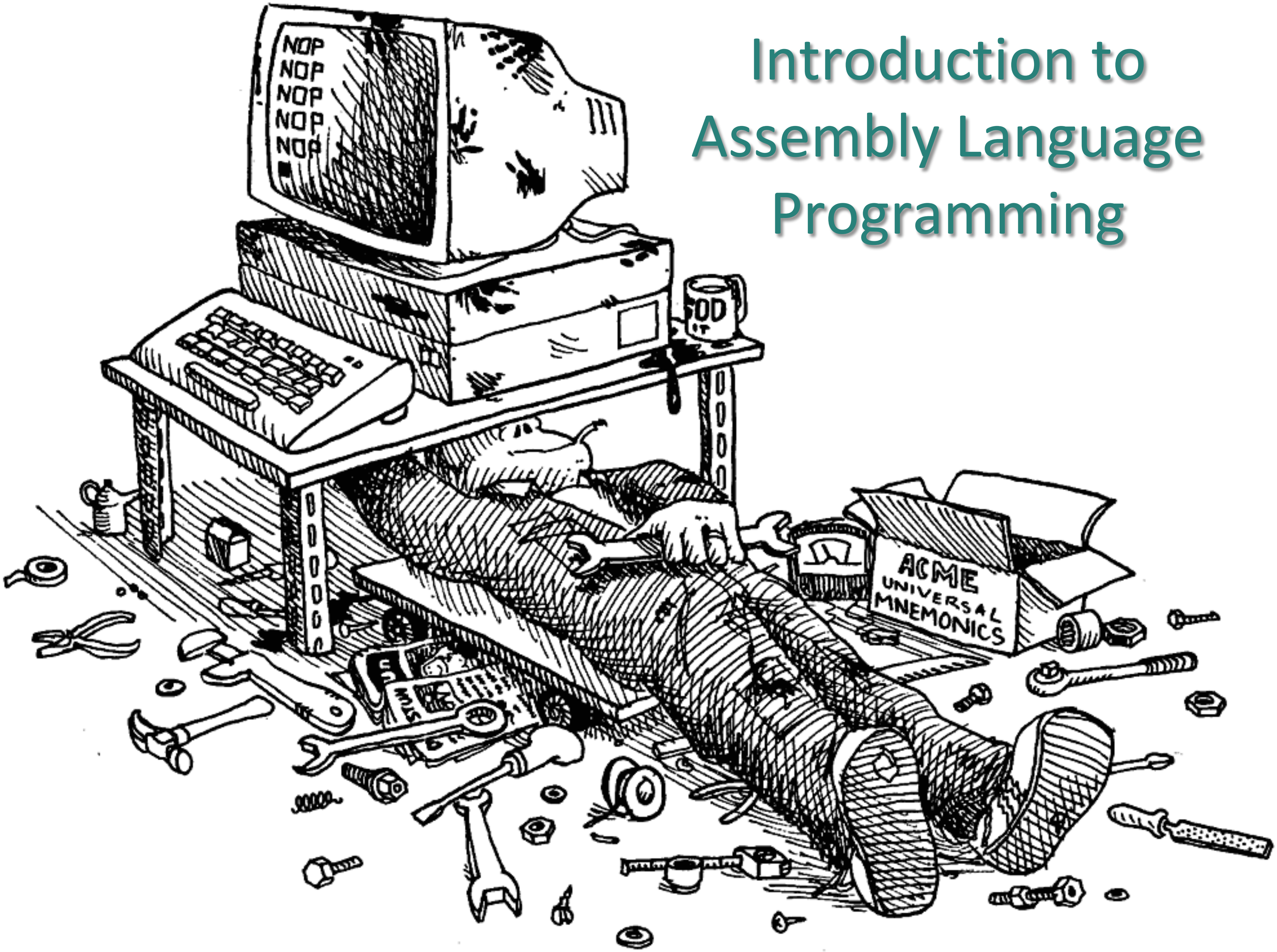
There's no carry-out (again, we don't care)

But there **is** overflow!

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

Note that V = 1, while Cout = 0 and Cin_signed_bit = 1

Introduction to Assembly Language Programming



The Simple Language of a CPU

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

- Restrictions:
  - Can only assign **integers** directly to variables
  - Can only 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 all 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 all 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

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

Instruction Register

?

Registers

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

Program Counter

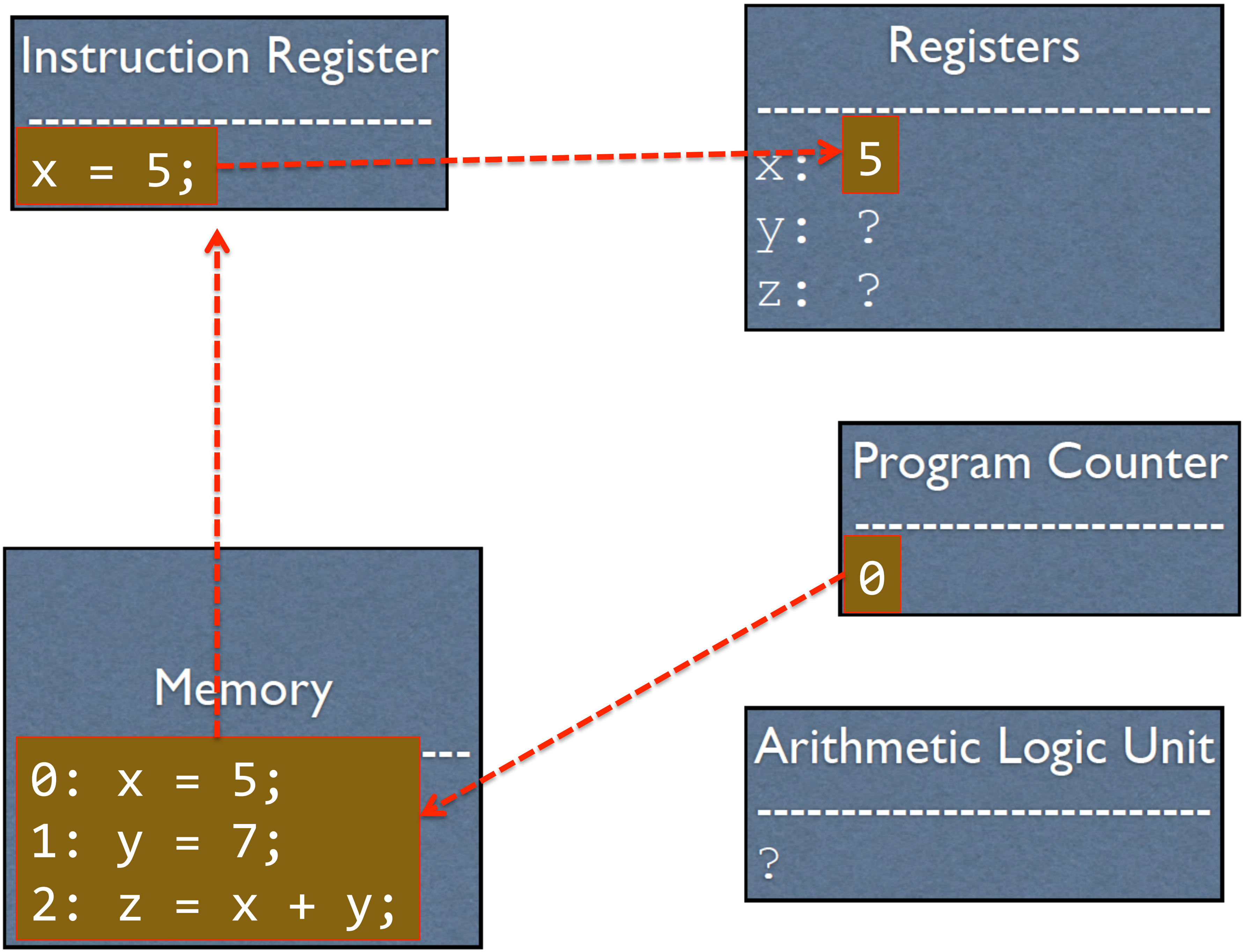
?

Memory

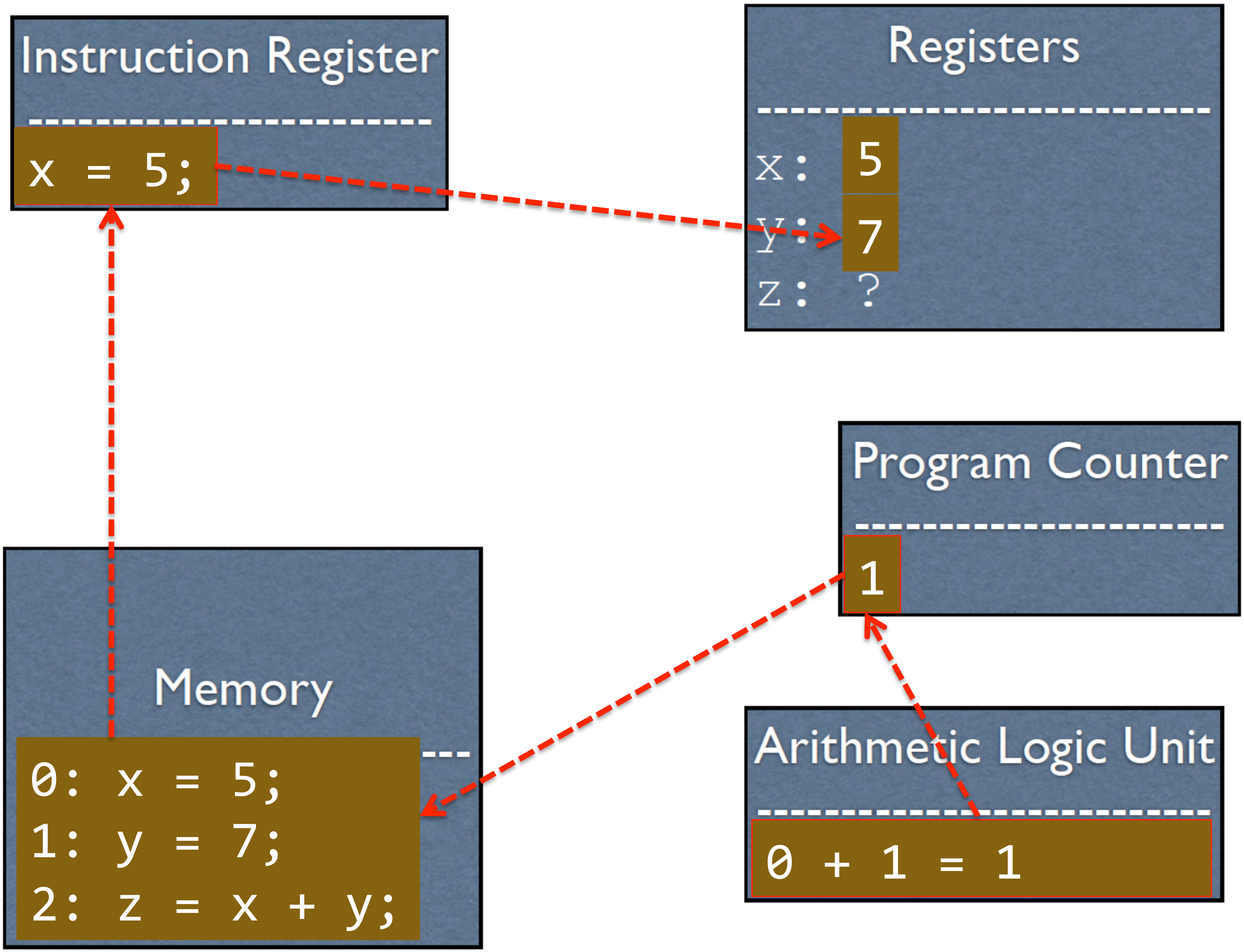
?

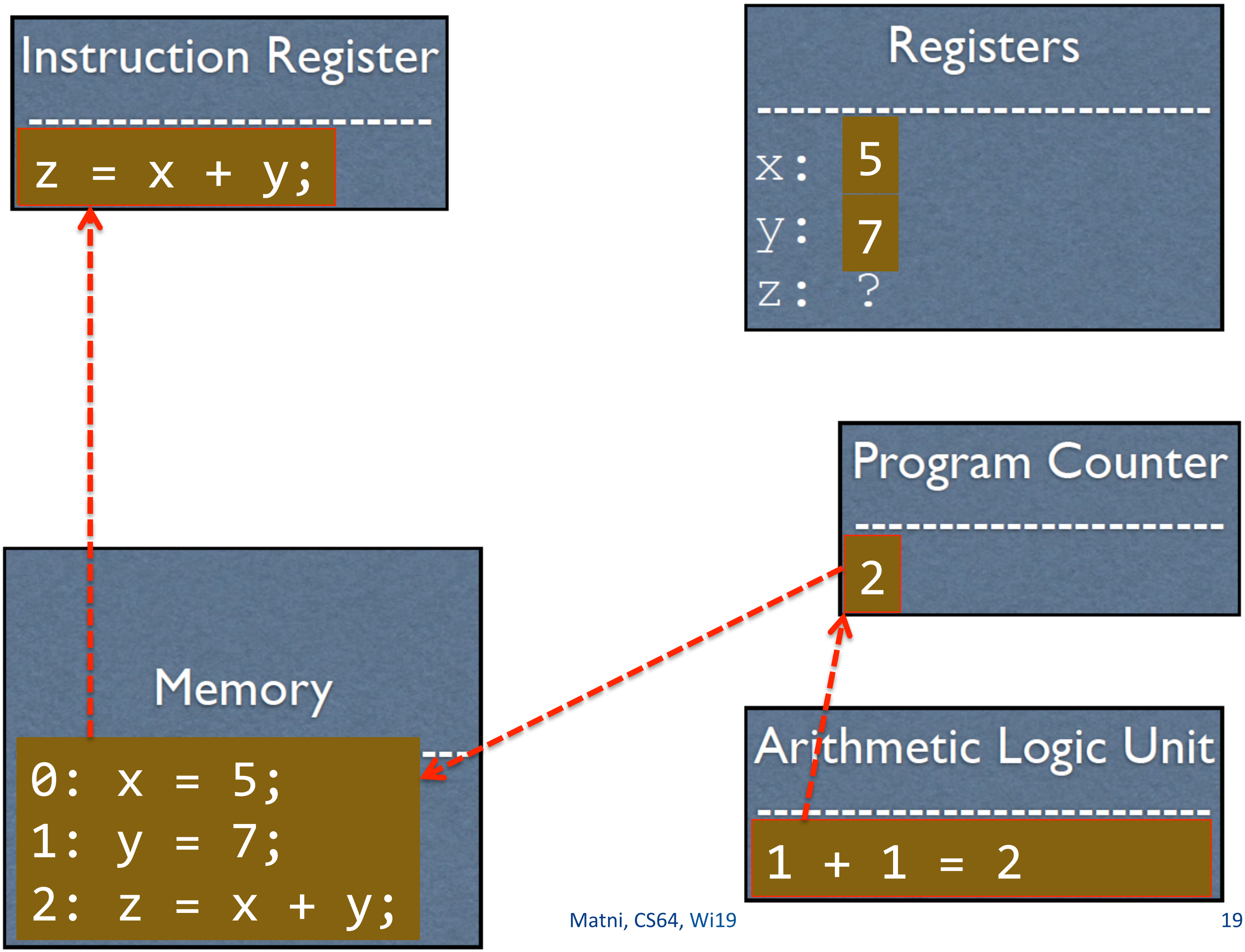
Arithmetic Logic Unit

?

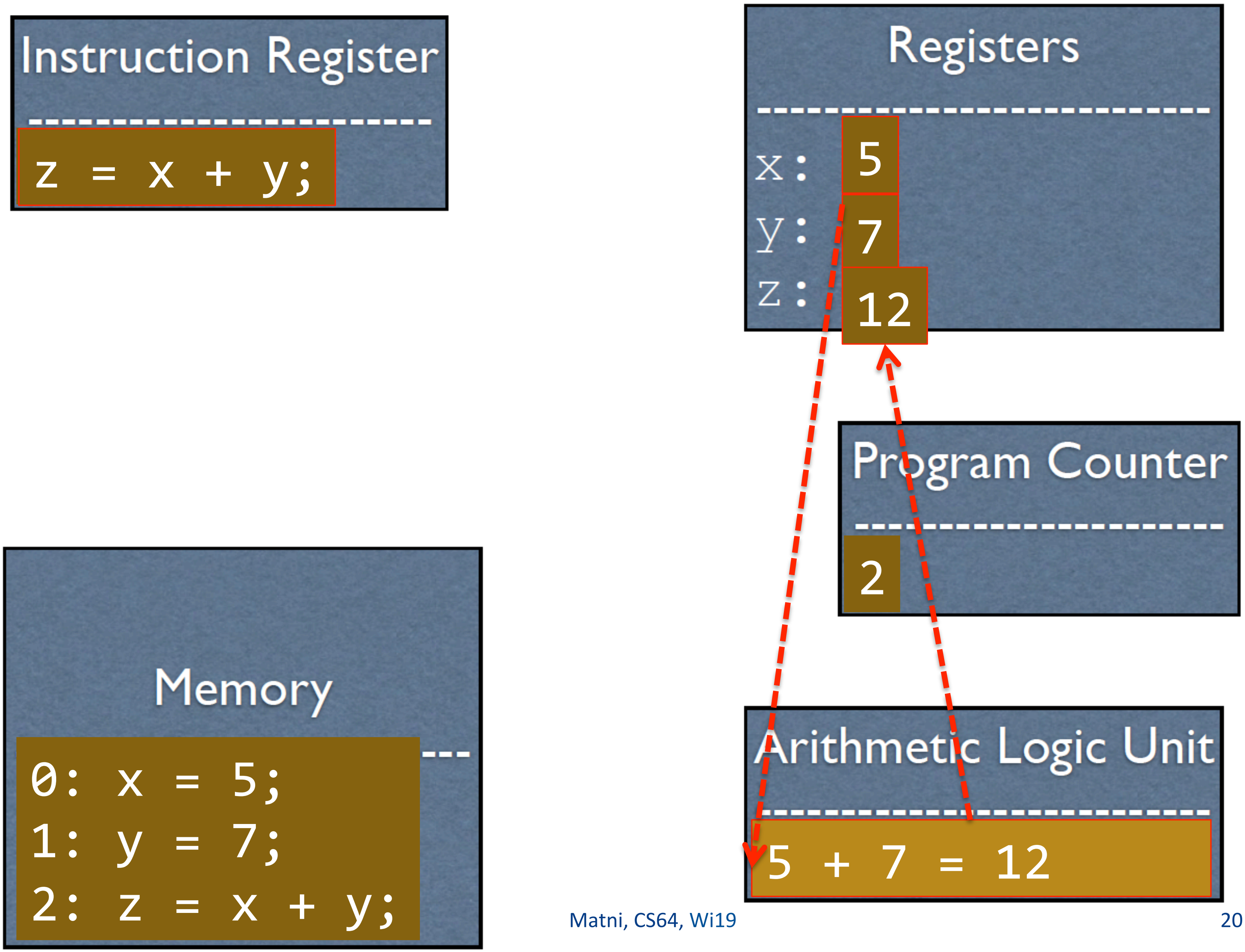


Instruction Register
x = 5;
Registers
x: 5
y: ?
z: ?
Program Counter
0
Memory
0: x = 5;
1: y = 7;
2: z = x + y;
Arithmetic Logic Unit
?


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

Why MIPS?

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

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

- All modern commercial processors share the same core concepts as MIPS, just with extra stuff

- ...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 24 stages

Note: Pipelining in CPUs

- Pipelining is a fundamental design in CPUs
- Allows multiple instructions to go on at once
  - a.k.a instruction-level parallelism

Basic five-stage pipeline

Instr. No. \ Clock cycle	1	2	3	4	5	6	7
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX

(IF = Instruction Fetch, ID = Instruction Decode, EX = Execute, MEM = Memory access, WB = Register write back).

Code on MIPS

Original	MIPS

`x = 5;`
`y = 7;`
`z = x + y;`
	`li $t0, 5`
	`li $t1, 7`
	`add $t3, $t0, $t1`

Code on MIPS

Original	MIPS

`x = 5;`
`y = 7;`
`z = x + y;`
	`li $t0, 5`
	`li $t1, 7`
	`add $t3, $t0, $t1`
	
	load immediate: put the given value into a register
	
	$t0: temporary register **0**

Code on MIPS

Original	MIPS

`x = 5;`
`y = 7;`
`z = x + y;`
	`li $t0, 5`
	`li $t1, 7`
	`add $t3, $t0, $t1`

	load immediate: put the given value into a register **$t1**: temporary register **1**

Code on MIPS

Original	MIPS

`x = 5;`
`y = 7;`
`z = x + y;`
	`li $t0, 5`
	`li $t1, 7`
	`add $t3, $t0, $t1`

	add: add the rightmost registers, putting the result in the first register **$t3**: temporary register **3**

Available Registers in MIPS

- 32 registers in all
  - Refer to your MIPS Reference Card
- For the moment, let's only consider registers **$t0 thru $t9**

NAME	NUMBER	USE
$zero	0	The Constant Value 0
$at	1	Assembler Temporary
$v0-$v1	2-3	Values for Function Results and Expression Evaluation
$a0-$a3	4-7	Arguments
$t0-$t7	8-15	Temporaries
$s0-$s7	16-23	Saved Temporaries
$t8-$t9	24-25	Temporaries
$k0-$k1	26-27	Reserved for OS Kernel
$gp	28	Global Pointer
$sp	29	Stack Pointer
$fp	30	Frame Pointer
$ra	31	Return Address

Assembly

- The code that you see is MIPS assembly

```
li $t0, 5
li $t1, 7
add $t3, $t0, $t1
```

- Assembly is *almost* what the machine sees. For the most part, it is a **direct** translation to binary from here (known as **machine language/code**)

- An **assembler** takes assembly code and changes it into the actual 1's and 0's for machine code
  - Analogous to a compiler for HL code

Machine Code/Language

- What a CPU actually accepts as input
- What actually gets executed

- Each instruction is represented with **32 bits**
  - No more, no less

- There are **three** different ***instruction formats***: **R**, **I**, and **J**
  - These allow for instructions to take on different roles
  - R-Format is used when it's all about **registers**
  - I-Format is used when you involve **(immediate) numbers**
  - J-Format is used when you do code "**jumping**" (i.e. branching)

Instruction Register

?

Registers

`$t0: ?`
`$t1: ?`
`$t2: ?`

Since all instructions are 32-bits, then they each occupy 4 Bytes of memory.
Memory is addressed in Bytes
(more on this later).

Program Counter

?

Memory

?

Arithmetic Logic Unit

?

Instruction Register

?

Registers

```
$t0: ?
$t1: ?
$t2: ?
```

Since all instructions are 32-bits, then they each occupy 4 Bytes of memory.

Memory is addressed in Bytes (more on this later).

Program Counter

0

Memory

```
0: li $t0, 5
4: li $t1, 7
8: add $t3, $t0, $t1
```

Arithmetic Logic Unit

?

Instruction Register

```
li $t0, 5
```

Registers

```
$t0: ?
$t1: ?
$t2: ?
```

Since all instructions are 32-bits, then they each occupy 4 Bytes of memory.

Memory is addressed in Bytes (more on this later).

Program Counter

```
0
```

Memory

```
0: li $t0, 5
4: li $t1, 7
8: add $t3, $t0, $t1
```

Arithmetic Logic Unit

```
?
```

Instruction Register

```
li $t0, 5
```

Registers

```
$t0: 5
$t1: ?
$t2: ?
```

Since all instructions are 32-bits, then they each occupy 4 Bytes of memory.

Memory is addressed in Bytes
(more on this later).

Program Counter

```
0
```

Memory

```
0: li $t0, 5
4: li $t1, 7
8: add $t3, $t0, $t1
```

Arithmetic Logic Unit

```
?
```

Instruction Register

```
li $t0, 5
```

Registers

```
$t0: 5
$t1: ?
$t2: ?
```

Since all instructions are 32-bits, then they each occupy 4 Bytes of memory.

Memory is addressed in Bytes (more on this later).

Program Counter

```
4
```

Memory

```
0: li $t0, 5
4: li $t1, 7
8: add $t3, $t0, $t1
```

Arithmetic Logic Unit

```
0 + 4 = 4
```

Instruction Register

```
li $t1, 7
```

Registers

```
$t0: 5
$t1: ?
$t2: ?
```

Since all instructions are 32-bits, then they each occupy 4 Bytes of memory.

Memory is addressed in Bytes (more on this later).

Program Counter

```
4
```

Memory

```
0: li $t0, 5
4: li $t1, 7
8: add $t3, $t0, $t1
```

Arithmetic Logic Unit

```
?
```

Instruction Register

```
li $t1, 7
```

Registers

```
$t0: 5
$t1: 7
$t2: ?
```

Since all instructions are 32-bits, then they each occupy 4 Bytes of memory.

Memory is addressed in Bytes
(more on this later).

Program Counter

```
4
```

Memory

```
0: li $t0, 5
4: li $t1, 7
8: add $t3, $t0, $t1
```

Arithmetic Logic Unit

```
?
```

Instruction Register

```
li $t1, 7
```

Registers

```
$t0: 5
$t1: 7
$t2: ?
```

Since all instructions are 32-bits, then they each occupy 4 Bytes of memory.

Memory is addressed in Bytes (more on this later).

Program Counter

```
8
```

Memory

```
0: li $t0, 5
4: li $t1, 7
8: add $t3, $t0, $t1
```

Arithmetic Logic Unit

```
4 + 4 = 8
```

Instruction Register

```
add $t3, $t0, $t1
```

Registers

```
$t0: 5
$t1: 7
$t2: ?
```

Since all instructions are 32-bits, then they each occupy 4 Bytes of memory.

Memory is addressed in Bytes (more on this later).

Program Counter

```
8
```

Memory

```
0: li $t0, 5
4: li $t1, 7
8: add $t3, $t0, $t1
```

Arithmetic Logic Unit

```
?
```

Instruction Register

`add $t3, $t0, $t1`

Registers

```
$t0: 5
$t1: 7
$t2: ?
```

Since all instructions are 32-bits, then they each occupy 4 Bytes of memory.

Memory is addressed in Bytes (more on this later).

Program Counter

8

Memory

```
0: li $t0, 5
4: li $t1, 7
8: add $t3, $t0, $t1
```

Arithmetic Logic Unit

---

5 + 7 = 12

Instruction Register

```
add $t3, $t0, $t1
```

Registers

```
$t0: 5
$t1: 7
$t2: 12
```

Since all instructions are 32-bits, then they each occupy 4 Bytes of memory.

Memory is addressed in Bytes
(more on this later).

Program Counter

```
8
```

Memory

```
0: li $t0, 5
4: li $t1, 7
8: add $t3, $t0, $t1
```

Arithmetic Logic Unit

```
5 + 7 = 12
```

</LECTURE>