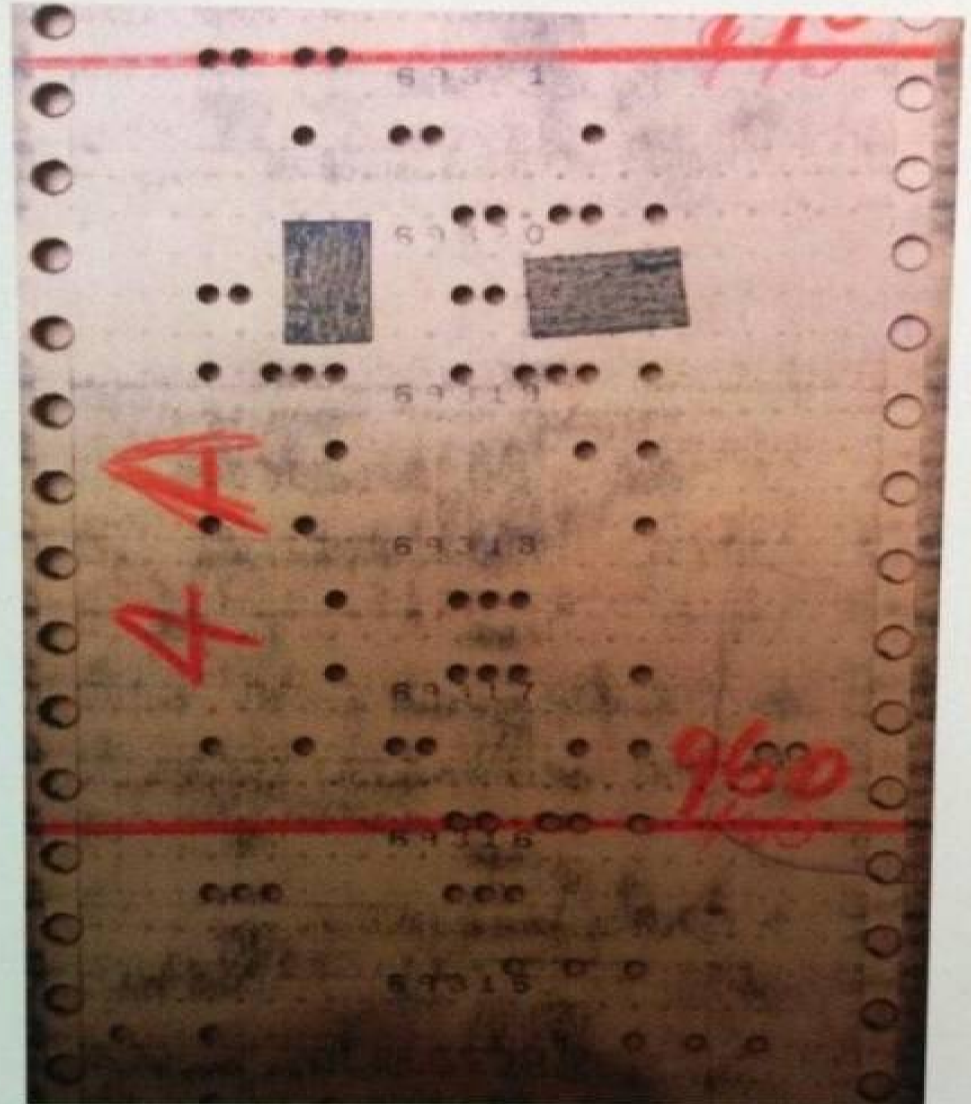
Introduction to Assembly Language

CS 64: Computer Organization and Design Logic
Lecture #4
Fall 2018

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

This Week on "Didja Know Dat?!"

The "Patch"



Small corrections to the programmed sequence could be done by patching over portions of the paper tape and re-punching the holes in that section.

Image courtesy of the Smithsonian Archives Center.

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

Because Oct-31 = Dec-25

Lecture Outline

- MIPS core processing blocks
- Basic programming in assembly
- Arithmetic programs

Administrative Stuff

- TA Office Hours (in Trailer 936)
  - Bay Yuan Hsu, bhsu@ucsb.edu,
    Thursdays 12:30 – 2:30 PM
  - Harmeet Singh, harmeetsingh@ucsb.edu,
    Thursdays 9:30 – 11:30 AM

- How did Lab# 2 go?
  - Too easy? Too hard? Just right?

Any Questions From Last Lecture?

The Simple Language of a CPU

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

- <u>Restrictions:</u>
  - 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

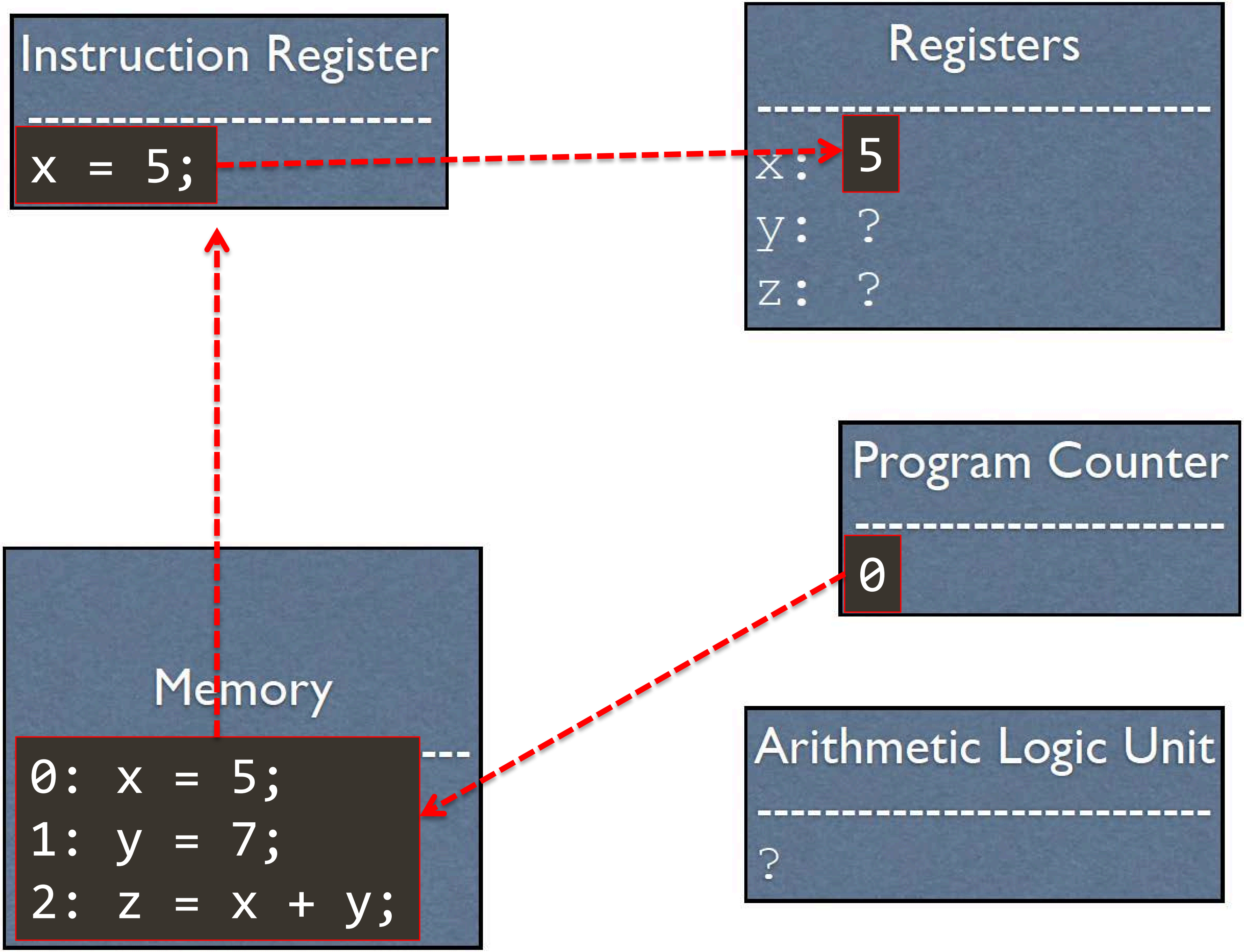
?

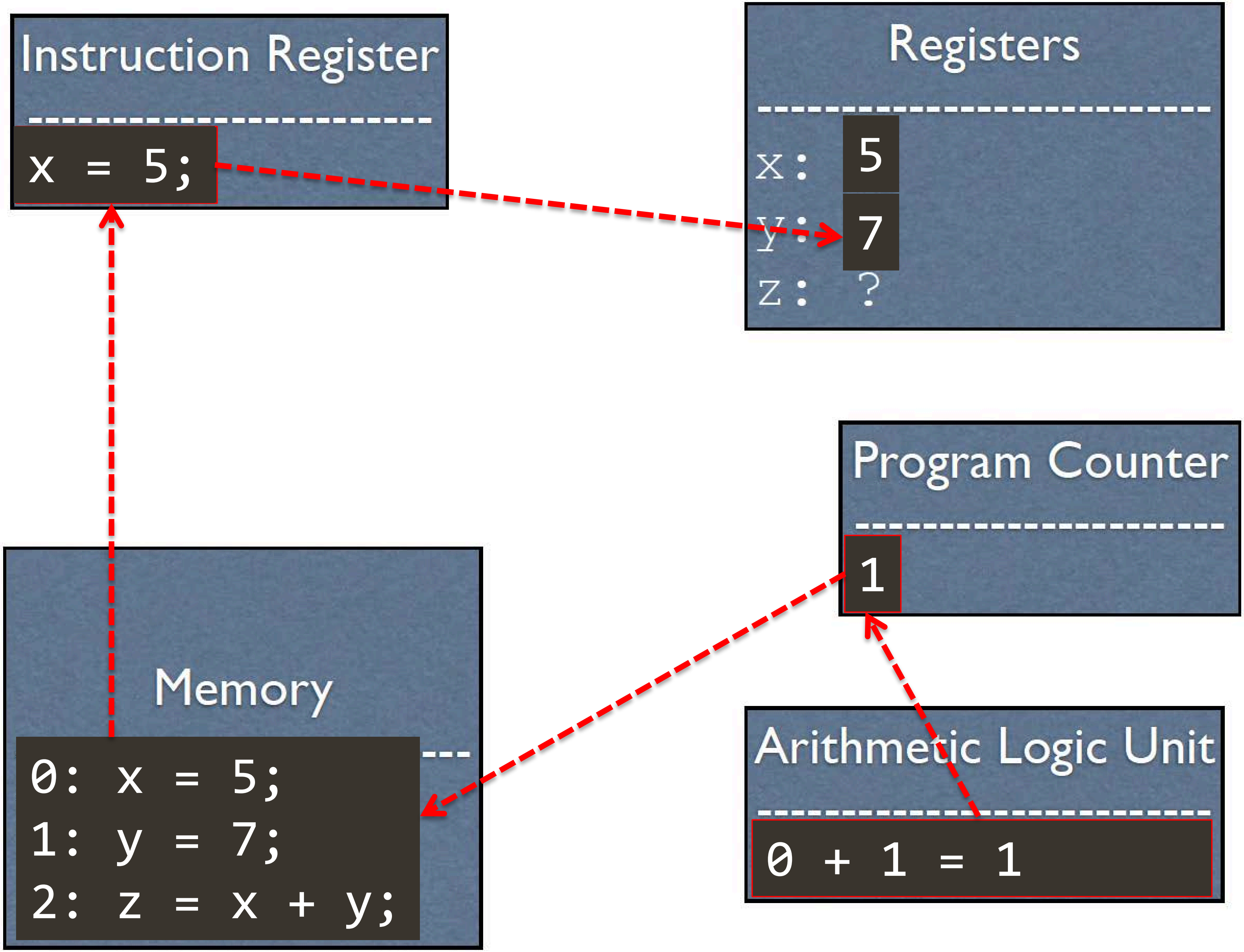
Memory

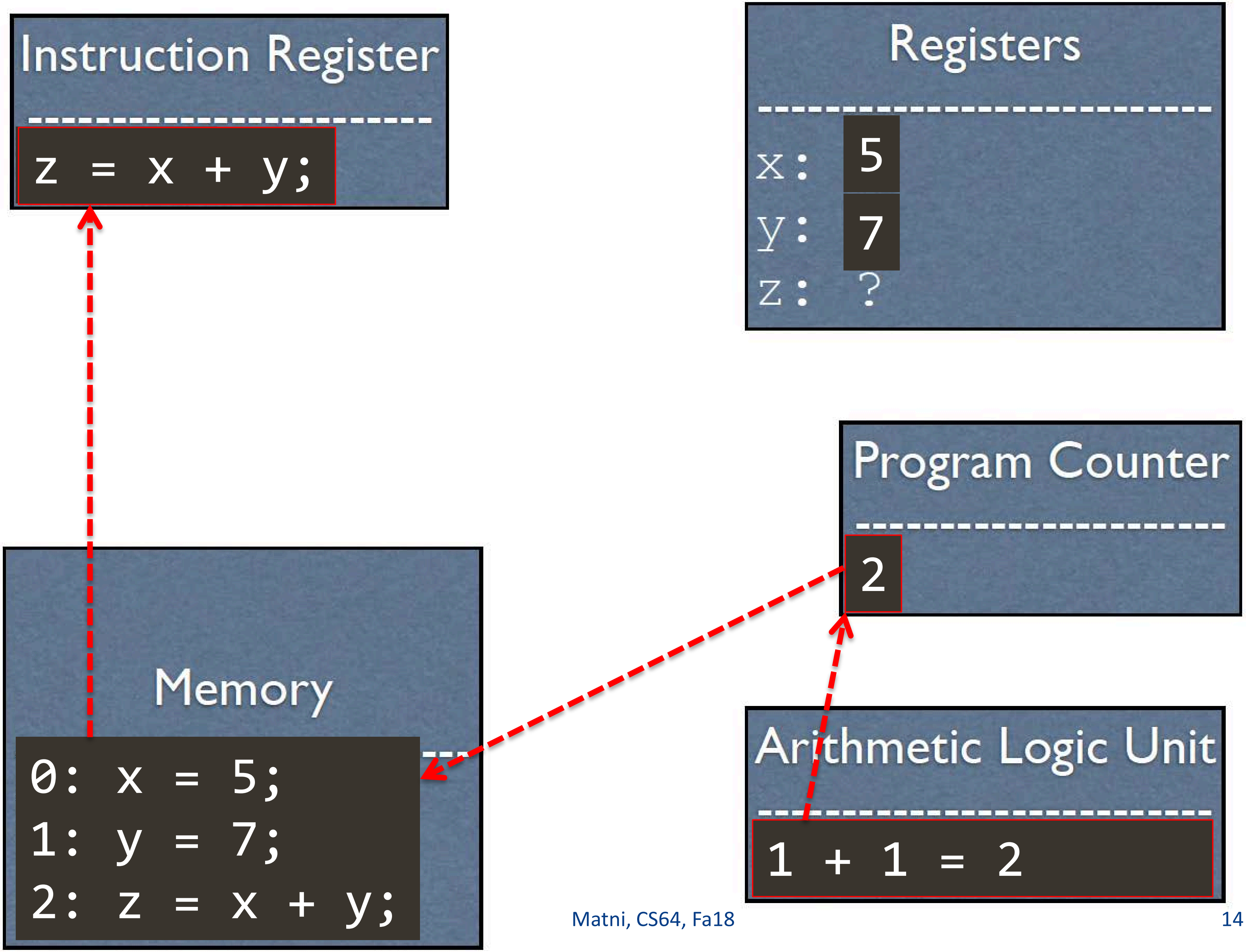
?

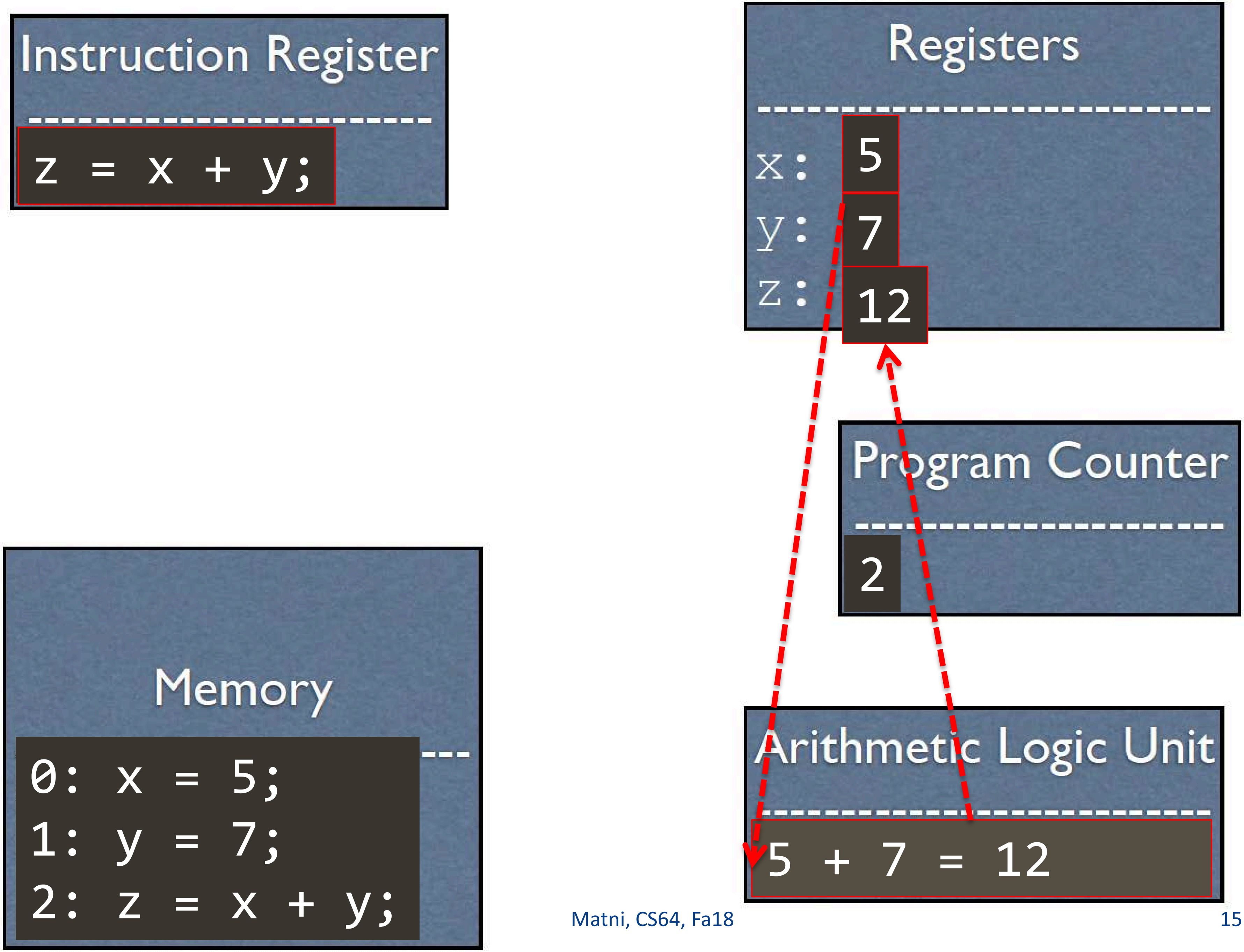
Arithmetic Logic Unit

?


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 Registers

?

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
```

Adding More Functionality

- What about: display results???? *Yes, that's kinda important…*
- What would this entail?
  - Engaging with Input / Output part of the computer
  - i.e. talking to devices
    - **Q: What usually handles this?** **A: the operating system**

- So we need a way to tell
  the operating system to kick in

Talking to the OS

- We are going to be running on MIPS ***emulator*** called **SPIM**
  - Optionally, through a program called **QtSPIM** (GUI based)
  - *What is an emulator?*

- We're not actually running our commands on an actual MIPS (hardware) processor!!

  …we're letting software pretend it's hardware…

  …so, in other words… we're "faking it"

- Ok, so how might we print something onto std.out?

SPIM Routines

- MIPS features a `syscall` instruction, which triggers a ***software interrupt***, or ***exception***

- Outside of an emulator (i.e. in the real world), these instructions **pause the program** and tell the OS to go do something with I/O

- Inside the emulator, it tells the emulator to go *emulate* something with I/O

`syscall`

- So we have the OS/emulator's attention, but how does it know what we want?
- The OS/emulator has access to the CPU registers
- We put special values (codes) in the registers to indicate what we want
  - These are codes that can't be used for anything else, so they're understood to be just for `syscall`
  - So… is there a "code book"????

Yes! All CPUs come with manuals.
For us, we have the **MIPS Ref. Card**

(Finally) Printing an Integer

- For SPIM, if register **`$v0`** contains **1** and then we issue a **syscall**, then SPIM will *print whatever **integer** is stored in register **`$a0`*** ***← this is a specific rule using a specific code***
  - Note: $v0 is used for other stuff as well – more on that later…
  - When $v0=1, syscall is *expecting* an integer!

- Other values put into **$v0** indicate other types of I/O calls to **syscall**

  Examples:
  - $v0 = 3 means **double (or the mem address of one) in $a0**
  - $v0 = 4 means **string (or the mem address of one) in $a0**
  - We'll explore some of these later, but check **MIPS ref card** for all of them

(Finally) Printing an Integer

- Remember, the usual syntax to load immediate a value into a register is:

```
li <register>, <value>
```

Example: **li $v0, 1** # PUTS THE NUMBER 1 INTO REG. $v0

- You can move the value of one register into another too!
- E.g. To make sure that the register **$a0** has the value of what you want to print out (let's say it's in another register), use the **move** command:

```
move <to register>, <from register>
```

Example: **move $a0, $t0** # PUTS THE VALUE IN REG. $t0 INTO REG. $a0

Ok… So About Those Registers
MIPS has 32 registers, each is 32 bits

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

Used for data

Program Files for MIPS Assembly

- The files have to be text
- Typical file extension type is **.asm**
- To leave comments,
  use **#** at the start of the line

Augmenting with Printing

```
# Main program
li $t0, 5
li $t1, 7
add $t3, $t0, $t1

# Print an integer to std.output
li $v0, 1
move $a0, $t3
syscall
```

We’re Not Quite Done Yet! Exiting an Assembly Program in SPIM

- If you are using SPIM, then you need to say when you are done as well
  - Most HLL programs do this for you automatically
- How is this done?
  - Issue a `syscall` with a special value in **$v0 = 10** (decimal)

Augmenting with Exiting

```
.text     # We always have to have this starting line
# Main program
li $t0, 5
li $t1, 7
add $t3, $t0, $t1

# Print to std.output
li $v0, 1
move $a0, $t3
syscall
# End program
li $v0, 10
syscall
```

MIPS Peculiarity:
NOR used a NOT

- How to make a NOT function using ***NOR*** instead
- Recall: NOR = NOT OR
- Truth-Table:

A	B	A NOR B
0	0	1
0	1	0
1	0	0
1	1	0

Note that:

0 NOR ***x*** = NOT ***x***

- So, in the absence of a NOT function,
  use a NOR with a 0 as one of the inputs!

Let’s Run This Program Already!

Using SPIM

- We’ll call it **simpleadd.asm**
- Run it on CSIL as: `$ spim –f simpleadd.asm`

DEMO !!!

- We’ll also run other arithmetic programs and explain them as we go along
  – TAKE NOTES!

YOUR TO-DOs

- Review ALL the demo code
  - Available via the class website

- Assignment #3
  - Lab tomorrow!
  - Due Friday

</LECTURE>