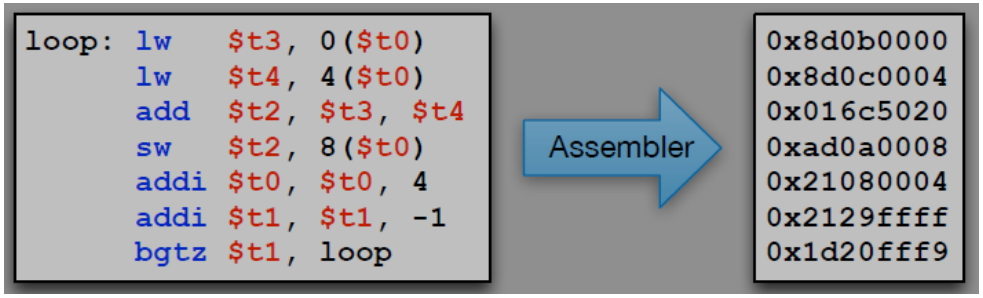


Intro to MIPS Assembly Language

CS 64: Computer Organization and Design Logic

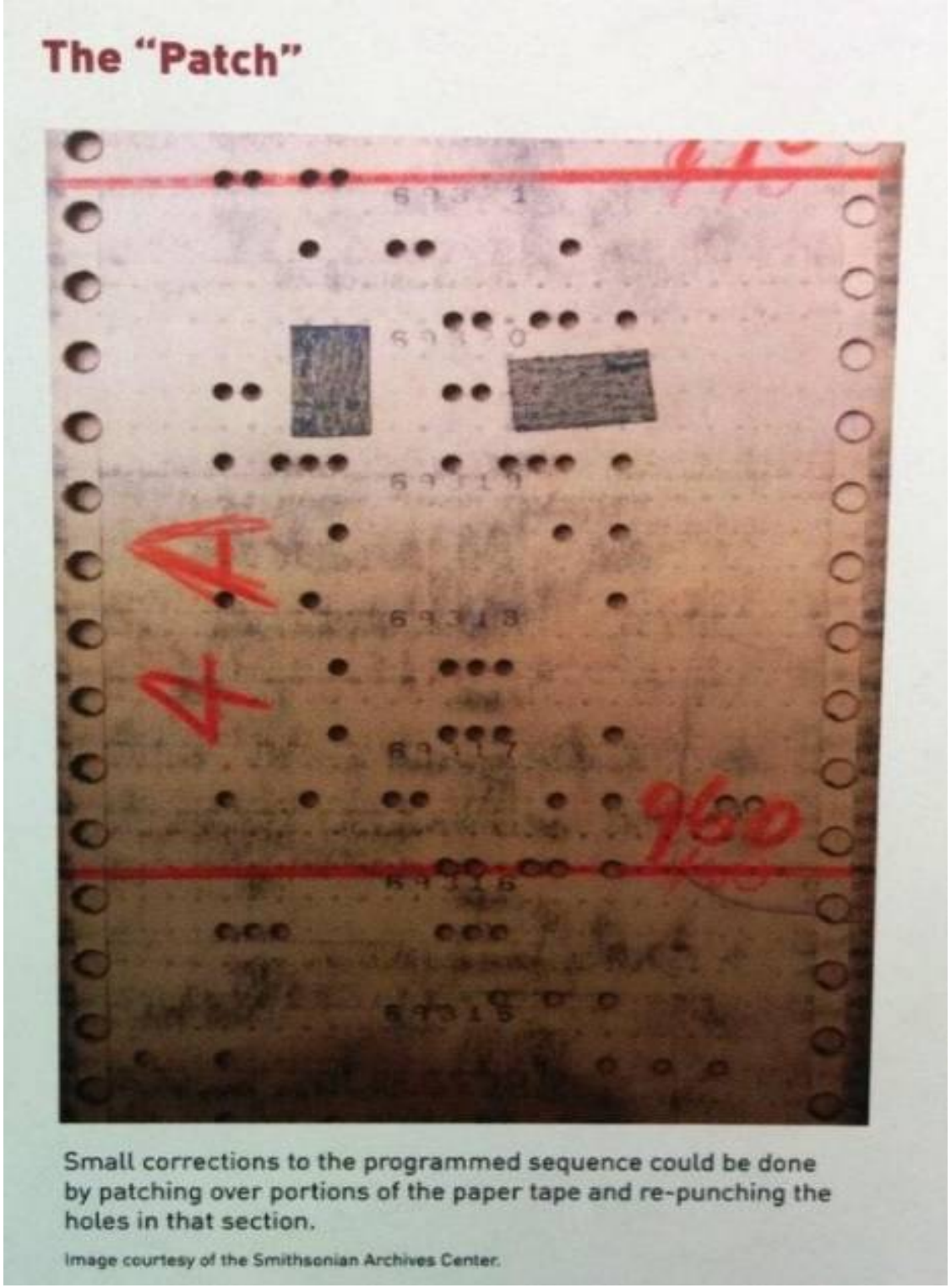
Lecture #4

Fall 2019

Ziad Matni, Ph.D.

Dept. of Computer Science, UCSB

This Week on "Didja Know Dat?!"



Administrative

- Pop Quiz #1

- Syllabus schedule will be slightly updated
- No change to exam dates…

Lecture Outline

- MIPS core processing blocks
- Basic programming in assembly
- Intro to SPIM use

Any Questions From Last Lecture?

Recall: 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

Recall: 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.

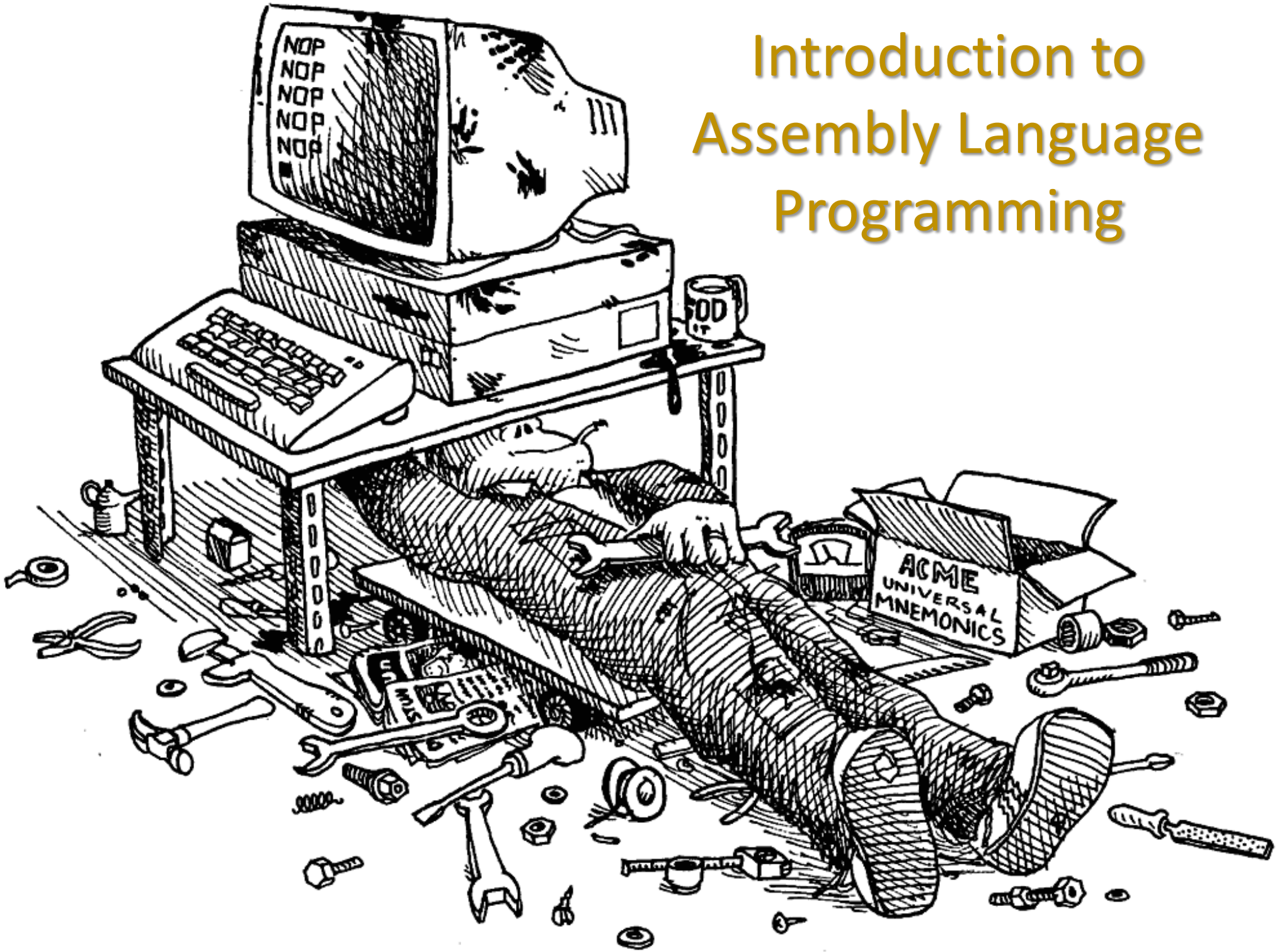
- Example: **i** = 1266
- In 32-bits of binary, that's:

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

- Restrictions:
  - 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:

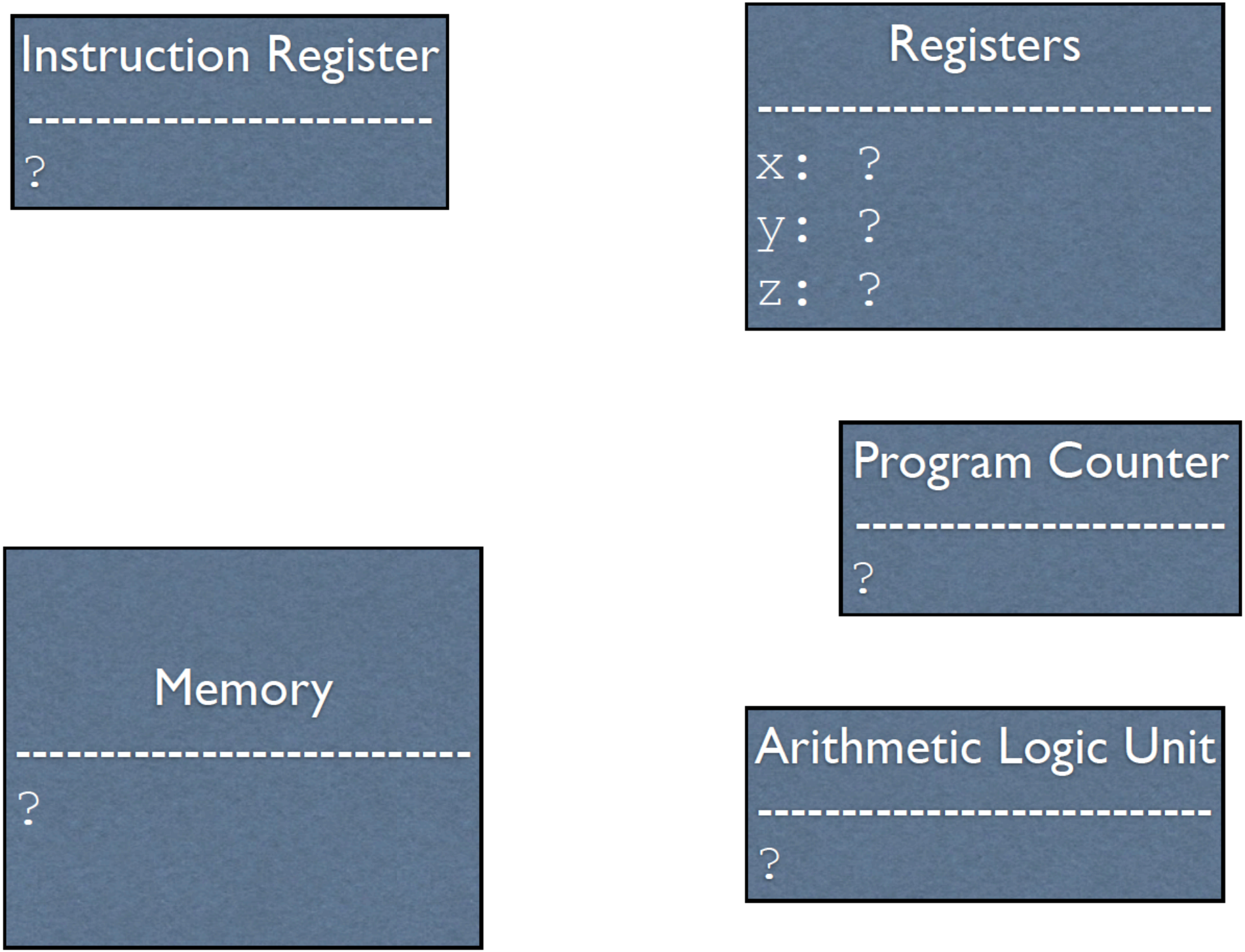
- Some place to tell us which statement is **currently** being executed → INSTRUCTION REGISTER (IR)

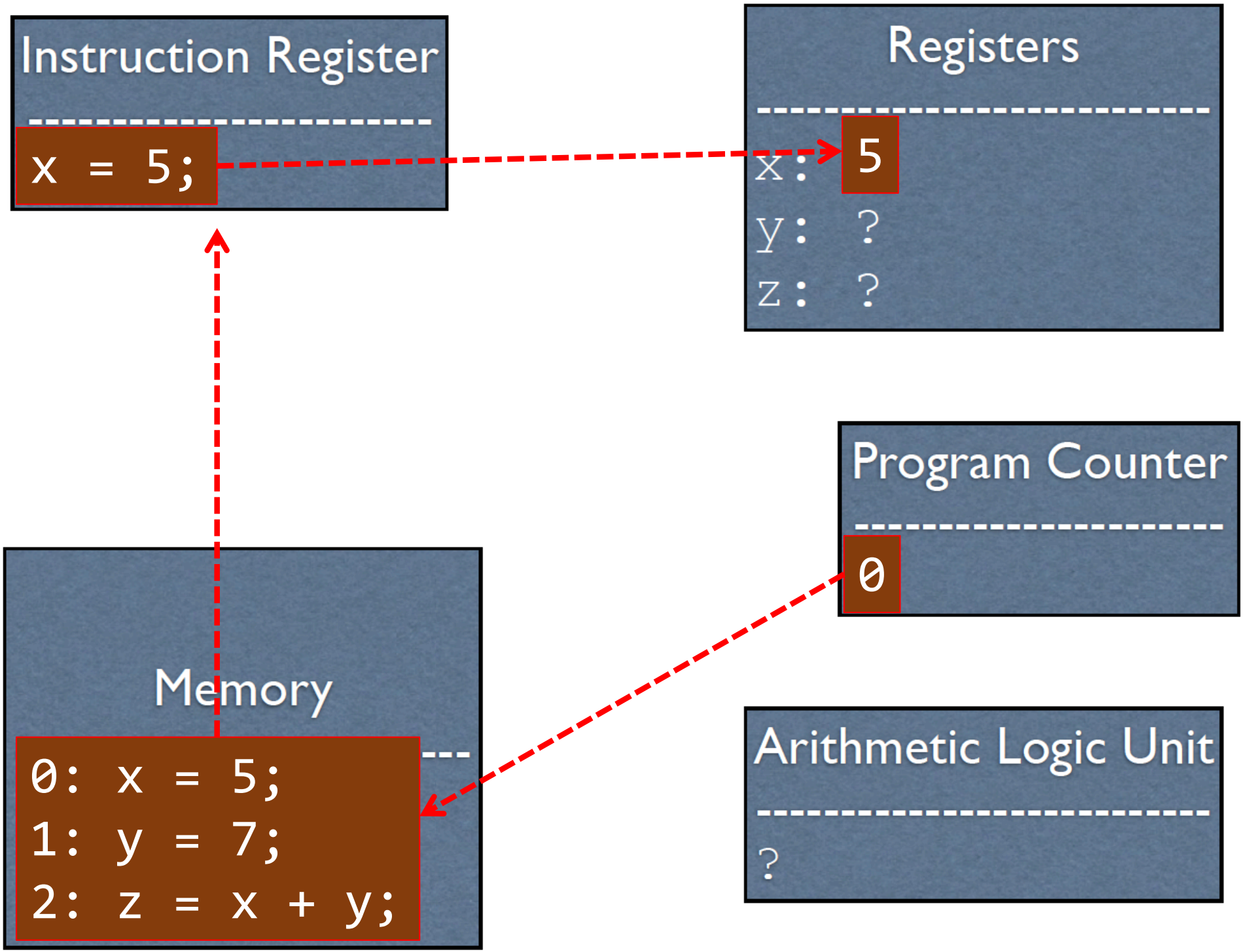
Basic Interaction

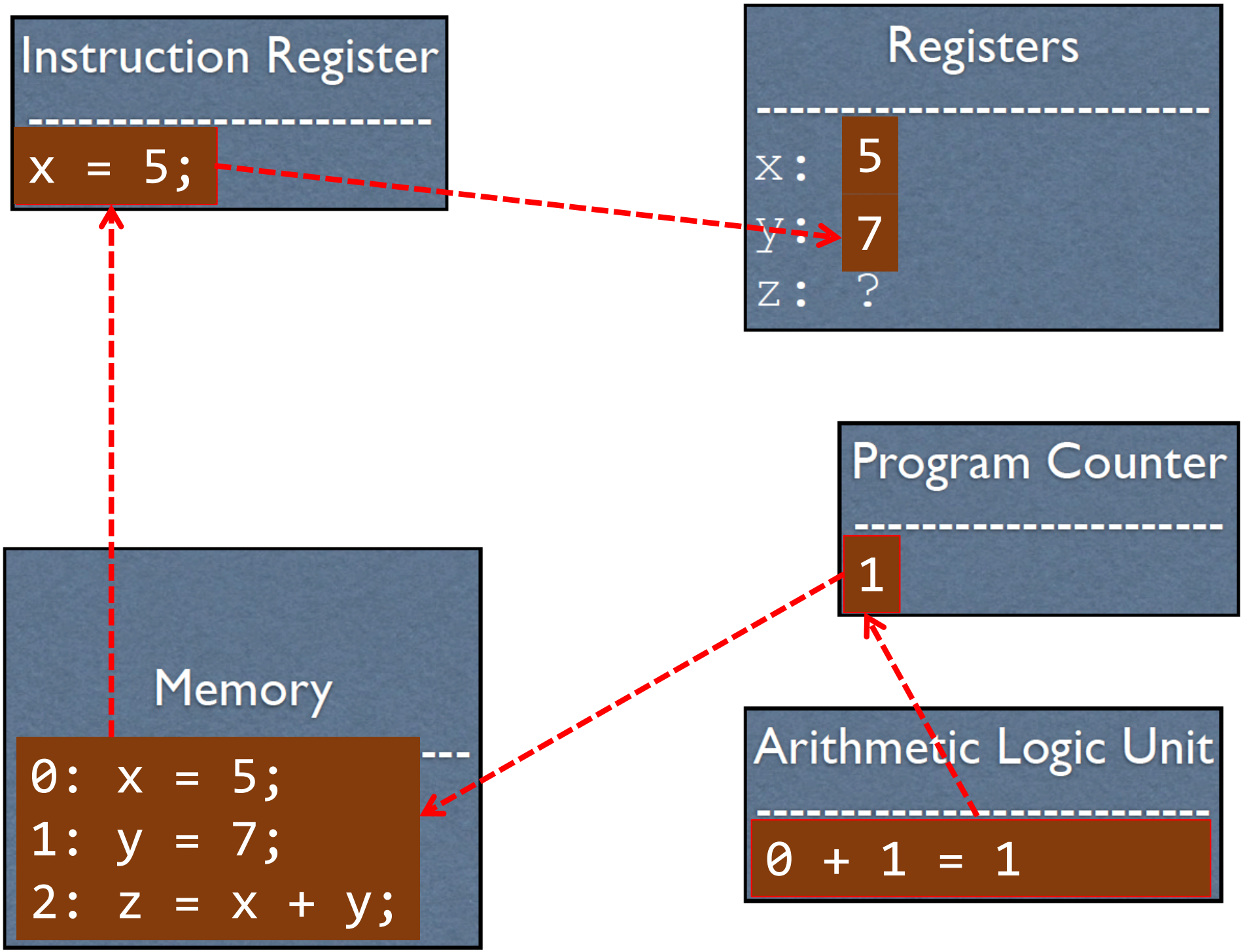
- Copy instruction from **memory** at wherever the **program counter (PC)** says into the **instruction register (IR)**
- Execute it, possibly involving registers and the **arithmetic logic unit (ALU)**
- Update the **PC** to point to the next instruction
- Repeat

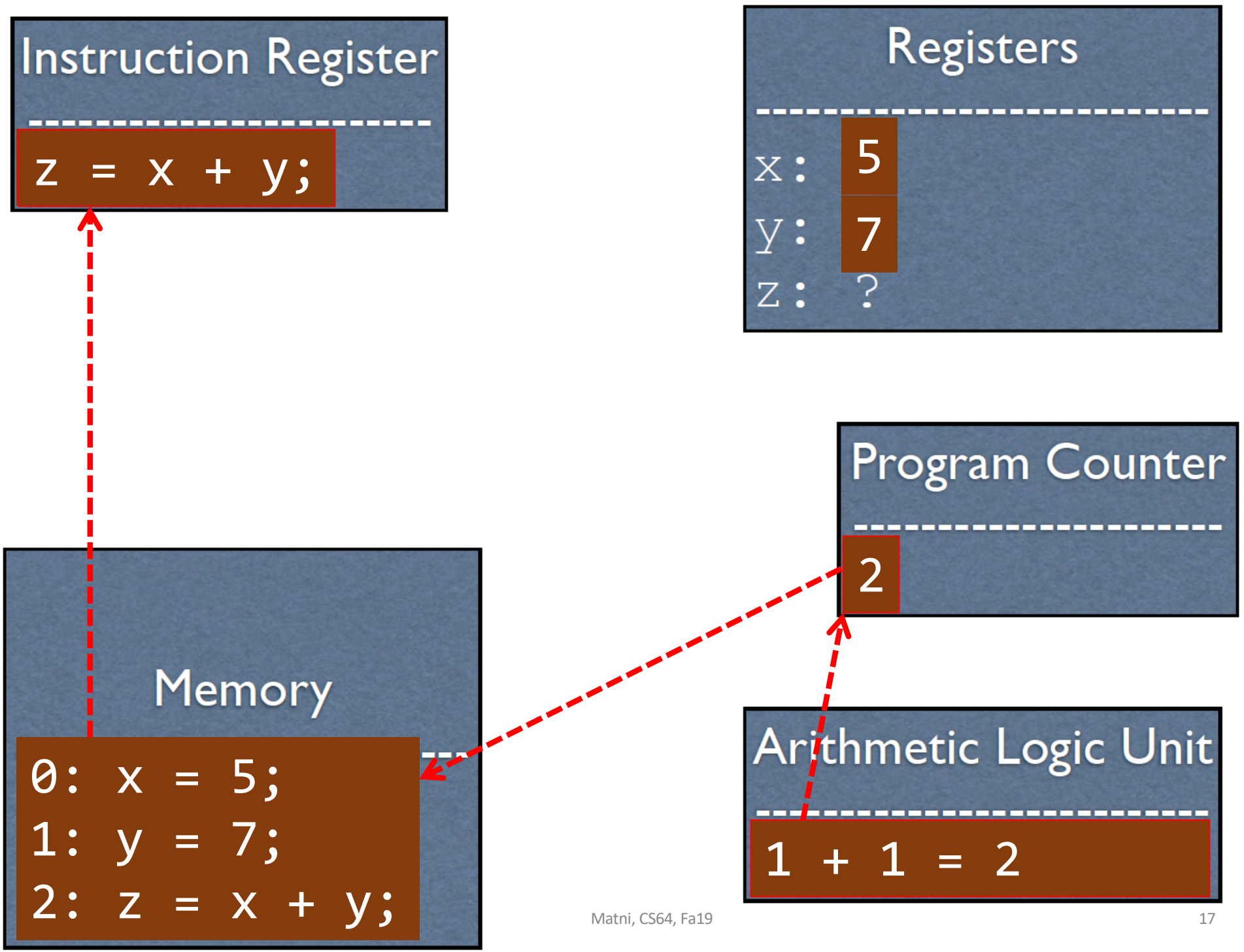
pseudocode

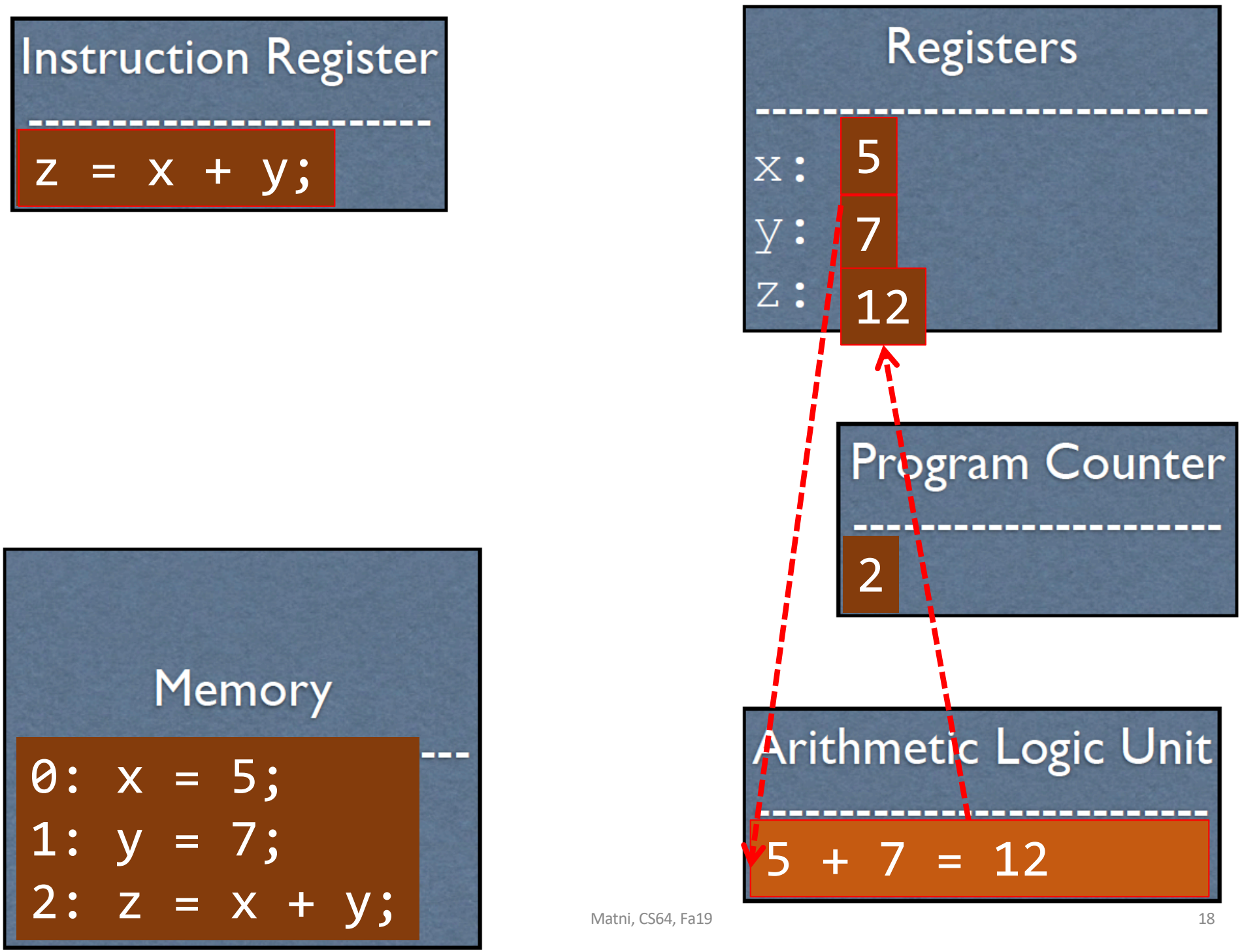
```
Initialize();
while (true) {
    instruc_reg = GetFromMem[prog_countr];
    executeInstruc(instruc_reg);
    prog_countr++;
}
```


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
- ...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)
- Modern CPUs include Intel, ARM, AMD

Code on MIPS

Original

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

MIPS

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

Code on MIPS

Original

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

MIPS

```
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

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

MIPS

```
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

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

MIPS

```
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**)

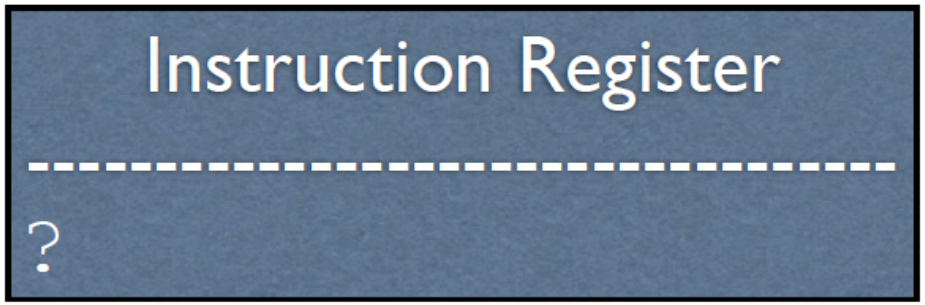
- An **assembler** takes assembly code and changes it into the actual 1's and 0's for machine code
  - Analogous to a compiler for HL code

Machine Code/Language

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

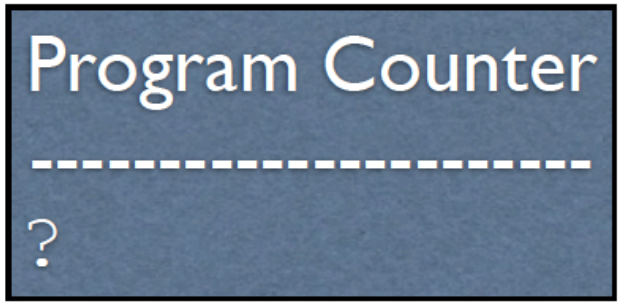
- Each instruction is represented with **32 bits**
  - No more, no less

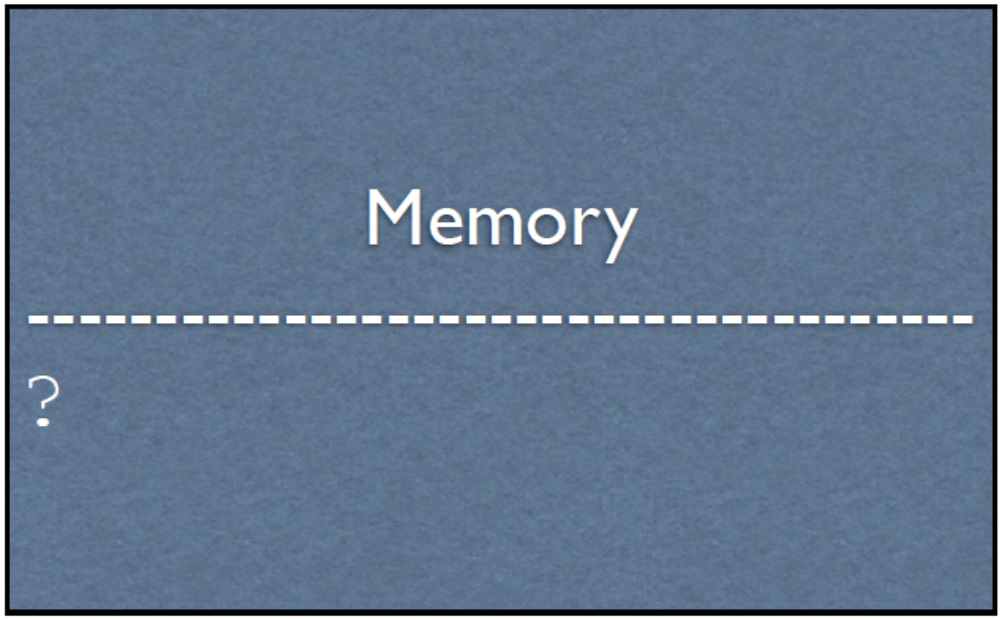
- There are **three** different ***instruction formats***: **R**, **I**, and **J**
  - These allow for instructions to take on different roles
  - R-Format is used when it's all about **registers**
  - I-Format is used when you involve **(immediate) numbers**
  - J-Format is used when you do code "**jumping**"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

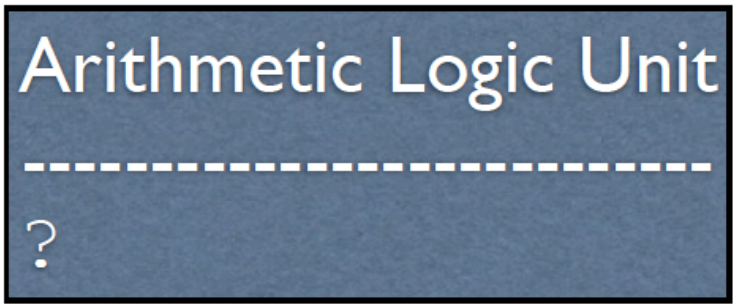


Since all instructions are 32-bits, then they each occupy 4 Bytes of memory.

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







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
```

Ok. Where's My MIPS Computer???

- You're not getting one.
- Who needs hardware when "cutting edge" software can do the job?!?!?!?!
- We will be SIMULATING a MIPS processor using software on our Macs/Windows/Linux machines.
- Hence… ***SPIM***… **The MIPS Emulator**!
  - Something funny about that name…

Adding More Functionality

- Ok, so I know how to add 2 numbers in MIPS.
  - Wow
- 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**

YOUR TO-DOs

- Do readings!
  - Check syllabus for details!
- Submit Assignment #1 TODAY! ☺
- Get to Assignment #2
  - You have to submit it into ***Gradescope as 2 parts***
    - PDF with answers to questions + Program (in C/C++)
  - Due on **Wednesday, 10/16, by 11:59:59 PM**


</LECTURE>