Binary Arithmetic
Intro to Assembly Language

CS 64: Computer Organization and Design Logic
Lecture #3

Ziad Matni
Dept. of Computer Science, UCSB

Adding this Class

- The class is full – I will not be adding more ppl ☹
  - Even if others drop

Lecture Outline

- Addition and subtraction in binary
- Multiplication in binary

- Introduction to MIPS Assembly Language

Any Questions From Last Lecture?

5-Minute Pop Quiz!!!

YOU MUST SHOW YOUR WORK!!!

1. Calculate and give your answer in hexadecimal:
   a) 0xA2 & 0xFE
   b) ~(0x3E | 0xFC)

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

Answers…

1. Calculate and give your answer in hexadecimal:
    a) 0xA2 & 0xFE **= 0xA2**
    b) ~(0x3E | 0xFC) **= ~(0xFE) = 0x01**

2. Convert from binary to decimal AND hexadecimal. Use any technique you like:
    a) 1001001 **= 0100 1001 = 0x49**
    = 1 + 8 + 64 = 73
    b) 10010010 **= 1001 0010 = 0x92**
    I see that it's (1001001) x 2 = 146

Twos Complement Method

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

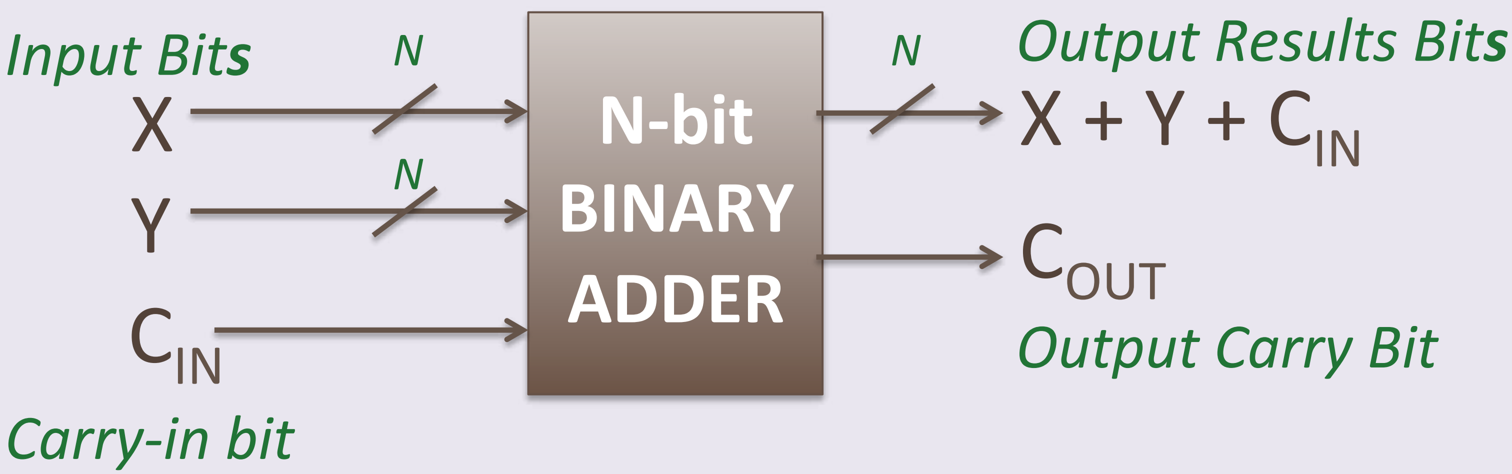
Exercises

RECALL THESE 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 a 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 **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

Cin_signed_bit

```
                 1
 -39        1101 1001
  92        0101 1100
 ---        ---------
  53       10011 0101
```

Cout

That's 53 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 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

$V = Cout \oplus Cin\_signed\_bit$

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

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

Cout = 0

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

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

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

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

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 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

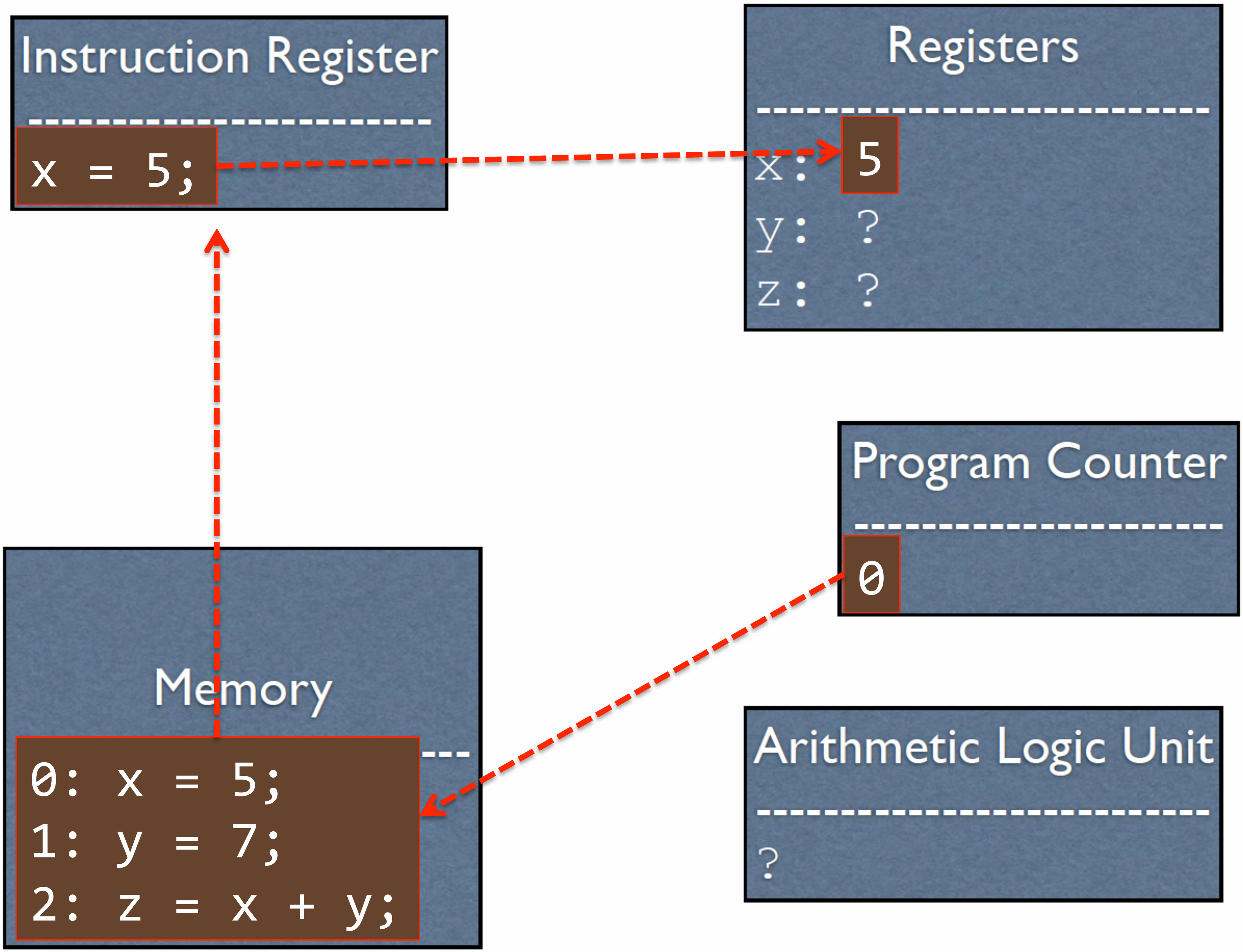
?

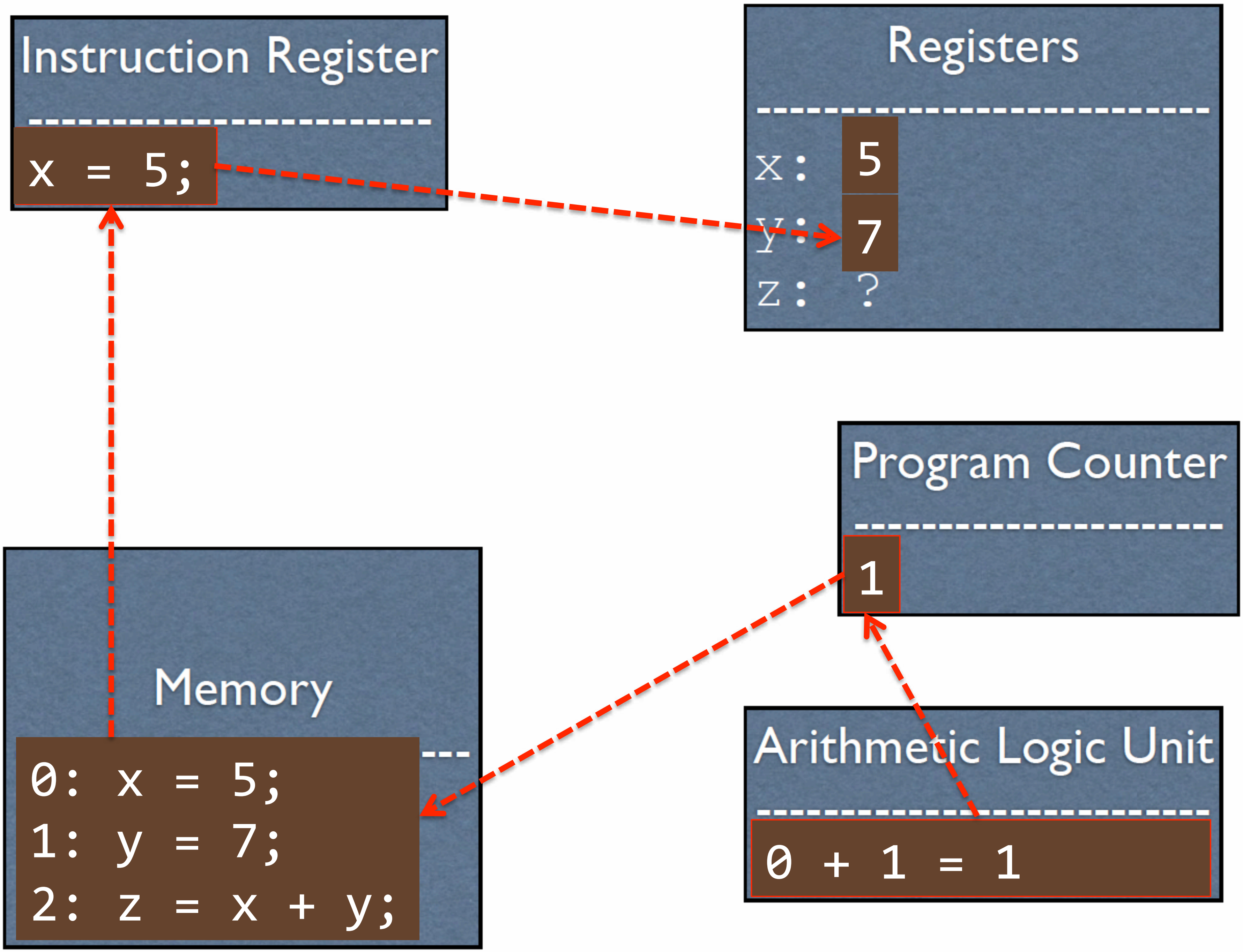
Memory

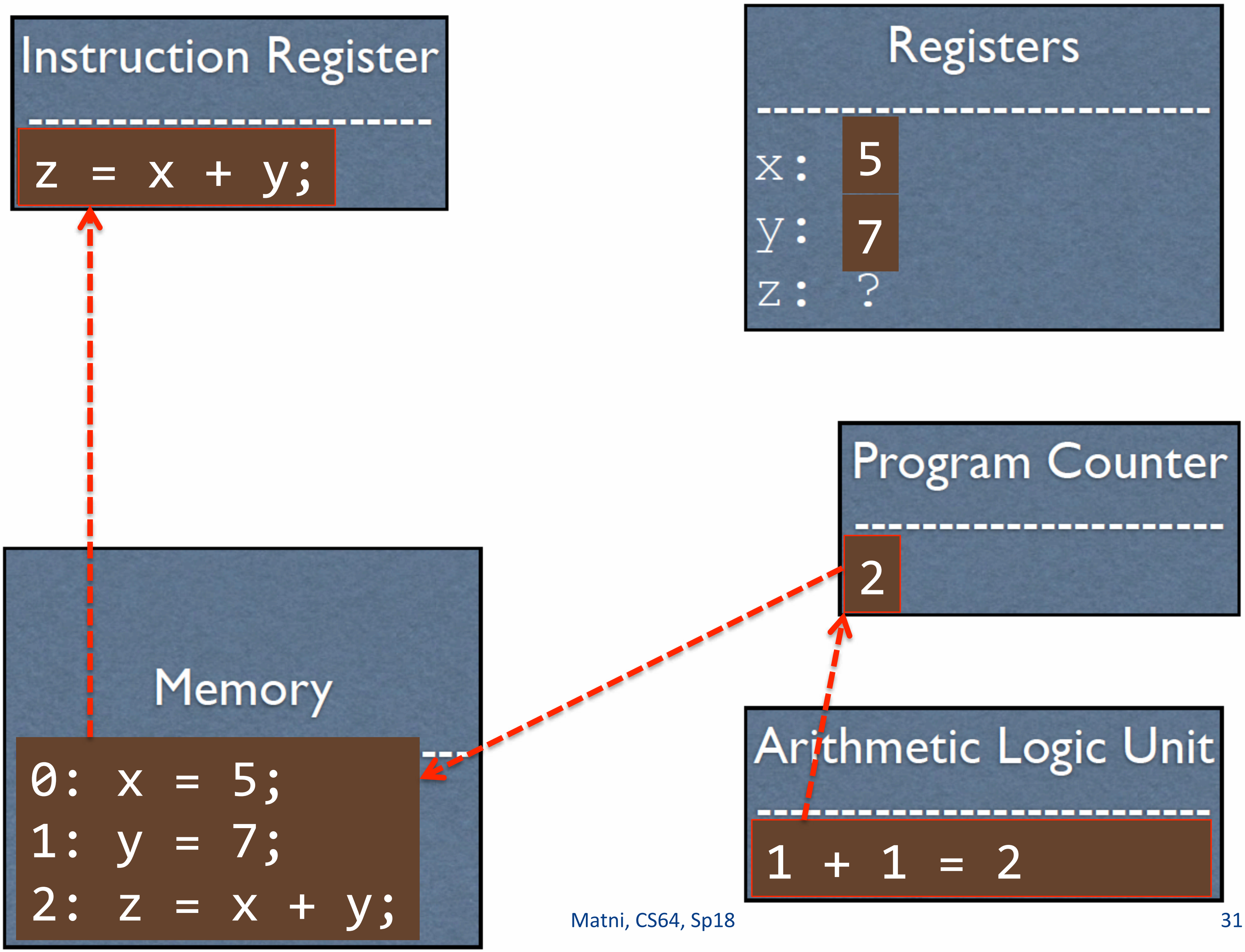
?

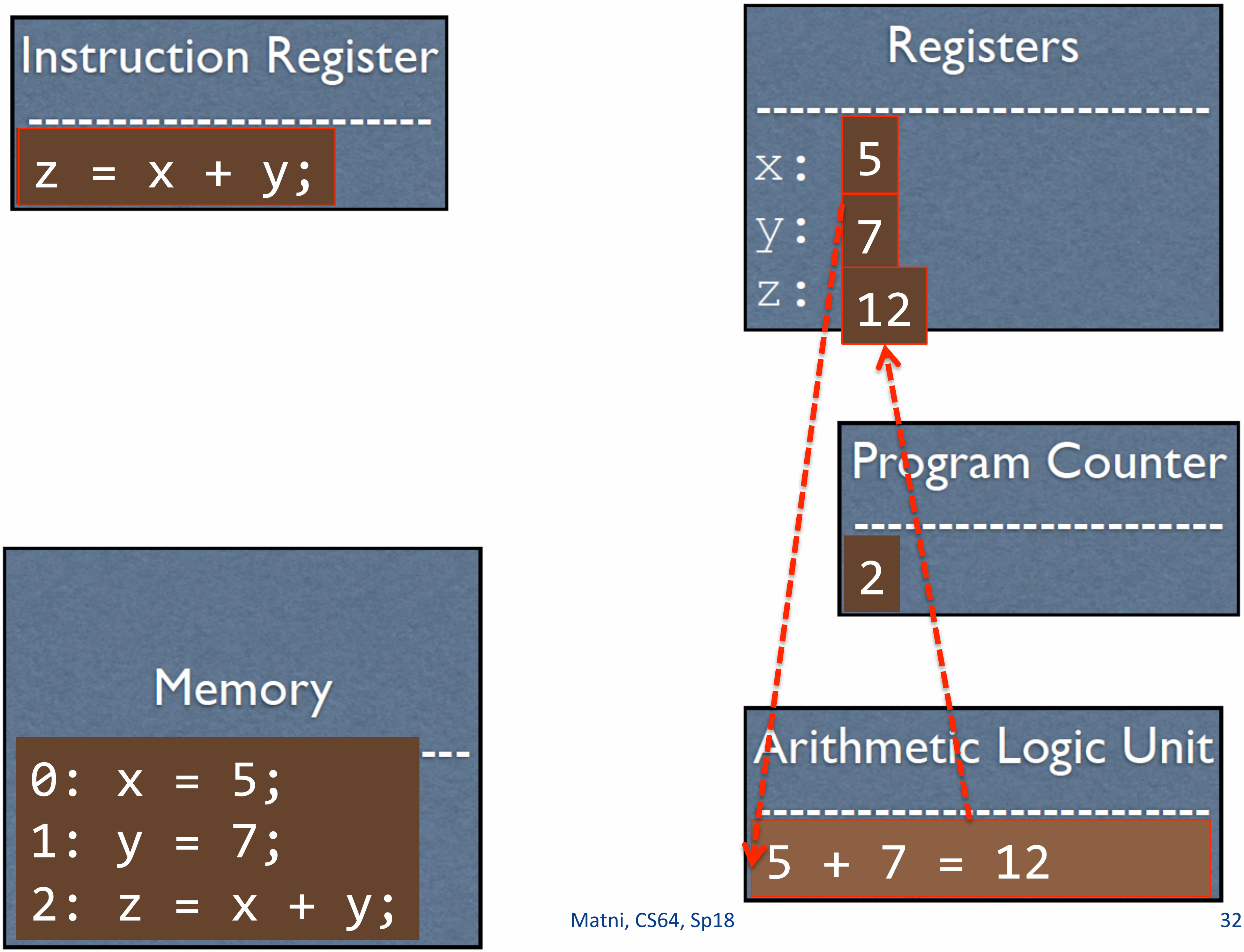
Arithmetic Logic Unit

?


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

Why MIPS?

- MIPS:
  - a reduced instruction set computer (RISC) architecture developed by MIPS Technologies (1981)
- Relevant in the embedded systems domain
- All modern commercial processors share the same core concepts as MIPS, just with extra stuff
- ...but most importantly...

It’s Simpler…

- … than other instruction sets for CPUs
  - So it’s a good learning tool

- Dozens of instructions as opposed to hundreds
- Lack of redundant instructions or special cases
- 5 stage pipeline versus 24 stages

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

```
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	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

```
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

- 32 registers in all
  - Each one has 32 bits
- For the moment, we will only consider registers $t0 - $t9
  - Your MIPS Reference Card shows you all of them

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 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

- This is what the process actually executes and accepts as input
- Each instruction is represented with **32 bits**
- Three different instruction formats; for the moment, we'll only look at the ***R format***

YOUR TO-DOs

- Assignment #1
  - Due Friday

</LECTURE>