COMP 122/L Lecture 6

Mahdi Ebrahimi

Slides adapted from Dr. Kyle Dewey

Assembly

What's in a Processor?

Simple Language

- We have variables, integers, addition, and assignment
- Restrictions:
  - Can only assign integers directly to variables
  - Can only add variables, always two at a time

Want to say:

```
z = 5 + 7;
```

Translation

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

Implementation

- What do we need to implement this?

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

Core Components

- Some place to hold the statements as we operate on them
- Some place to hold which statement is next
- Some place to hold variables
- Some way to add numbers

Back to Processors

- Amazingly, these are all the core components of a processor
  - Why is this significant?

Back to Processors

- Amazingly, these are all the core components of a processor
  - Why is this significant?
- Processors just reads a series of statements (instructions) forever. No magic.

Core Components

- Some place to hold the statements as we operate on them
- Some place to hold which statement is next
- Some place to hold variables
- Some way to add numbers

Core Components

- Some place to hold the statements as we operate on them **- memory**
- Some place to hold which statement is next **- program counter**
- Some place to hold variables **- registers**
  - Behave just like variables with fixed names
- Some way to add numbers – **arithmetic logic unit (ALU)**
- Some place to hold which statement is currently being executed – **instruction register (IR)**

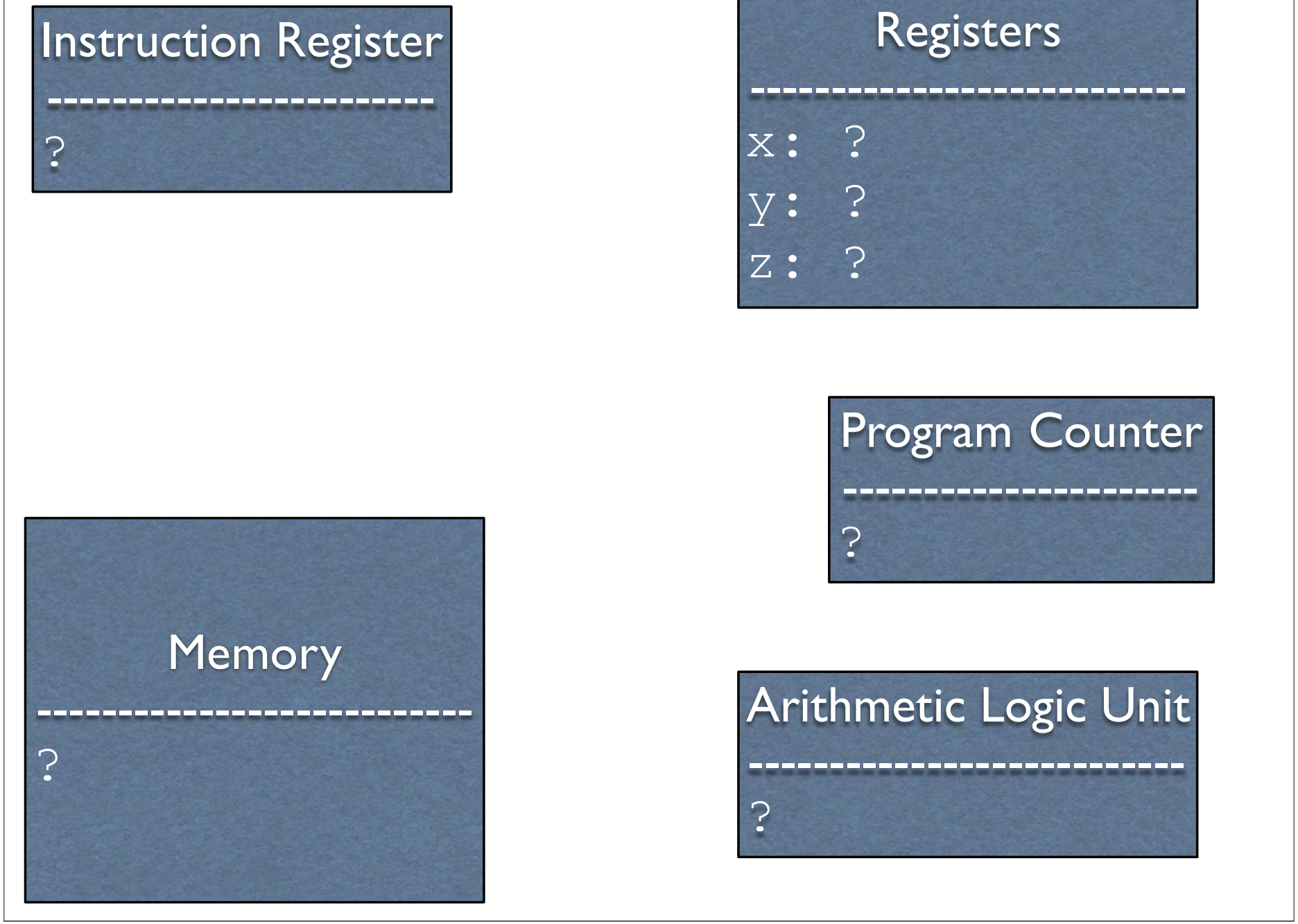
Basic Interaction

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

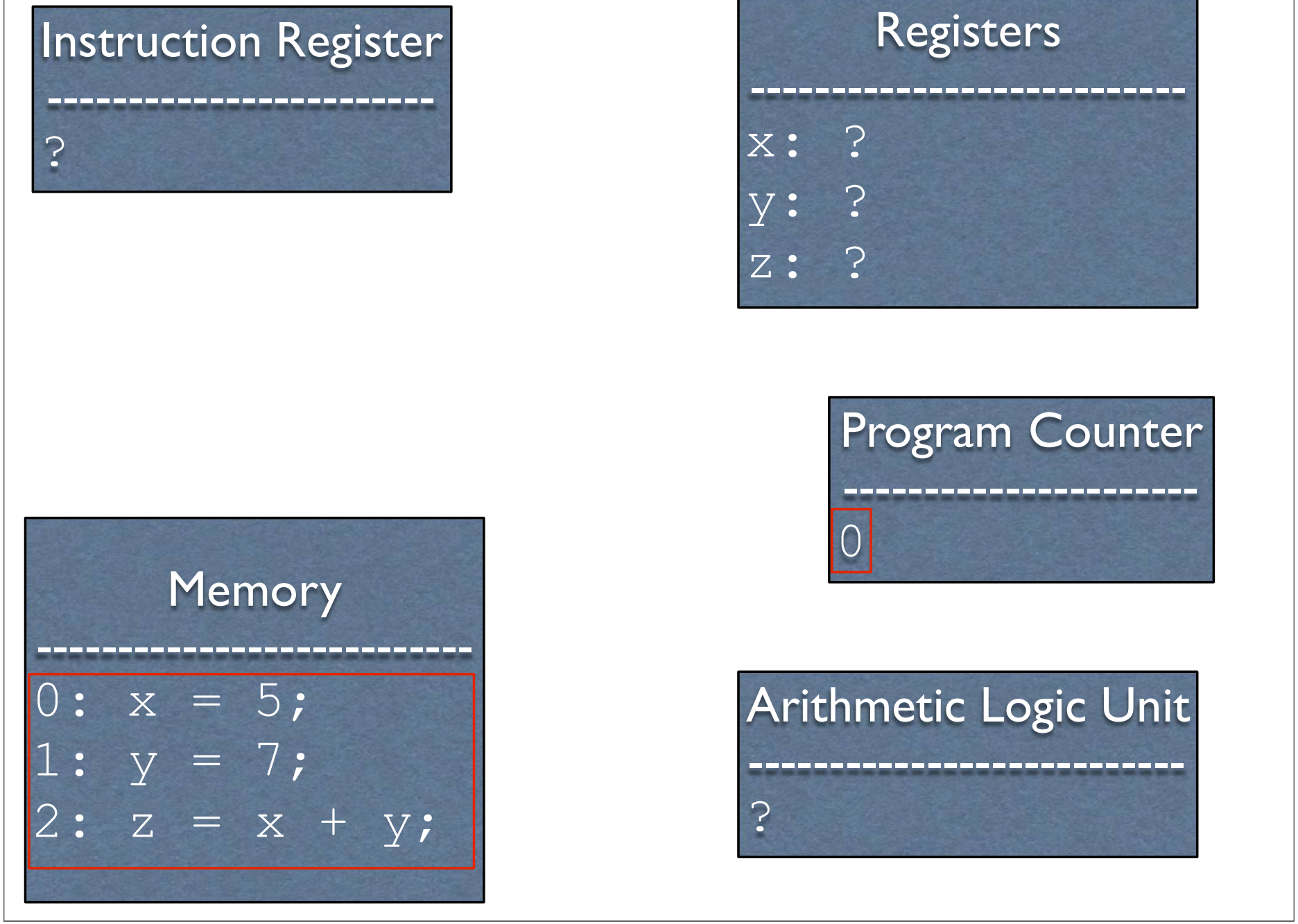
Basic Interaction

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

-initialize() will load in the initial state, and put instructions in memory
-execute(instruction_register) will read the instruction and do what it says, potentially looking at registers, assigning things to registers, and using the arithmetic logic unit
-Have this handy while going through next animation

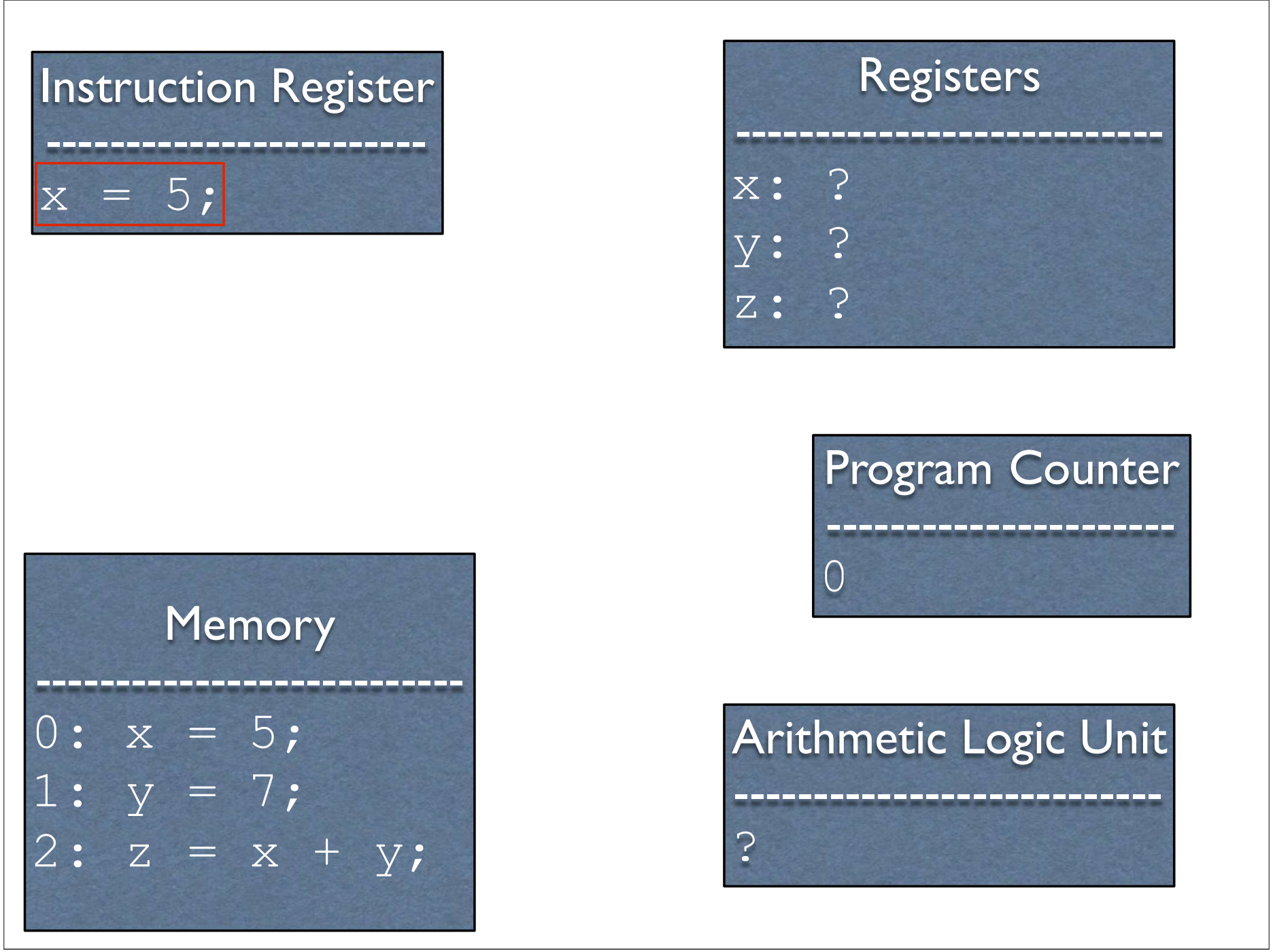


–All the hardware, before initialization

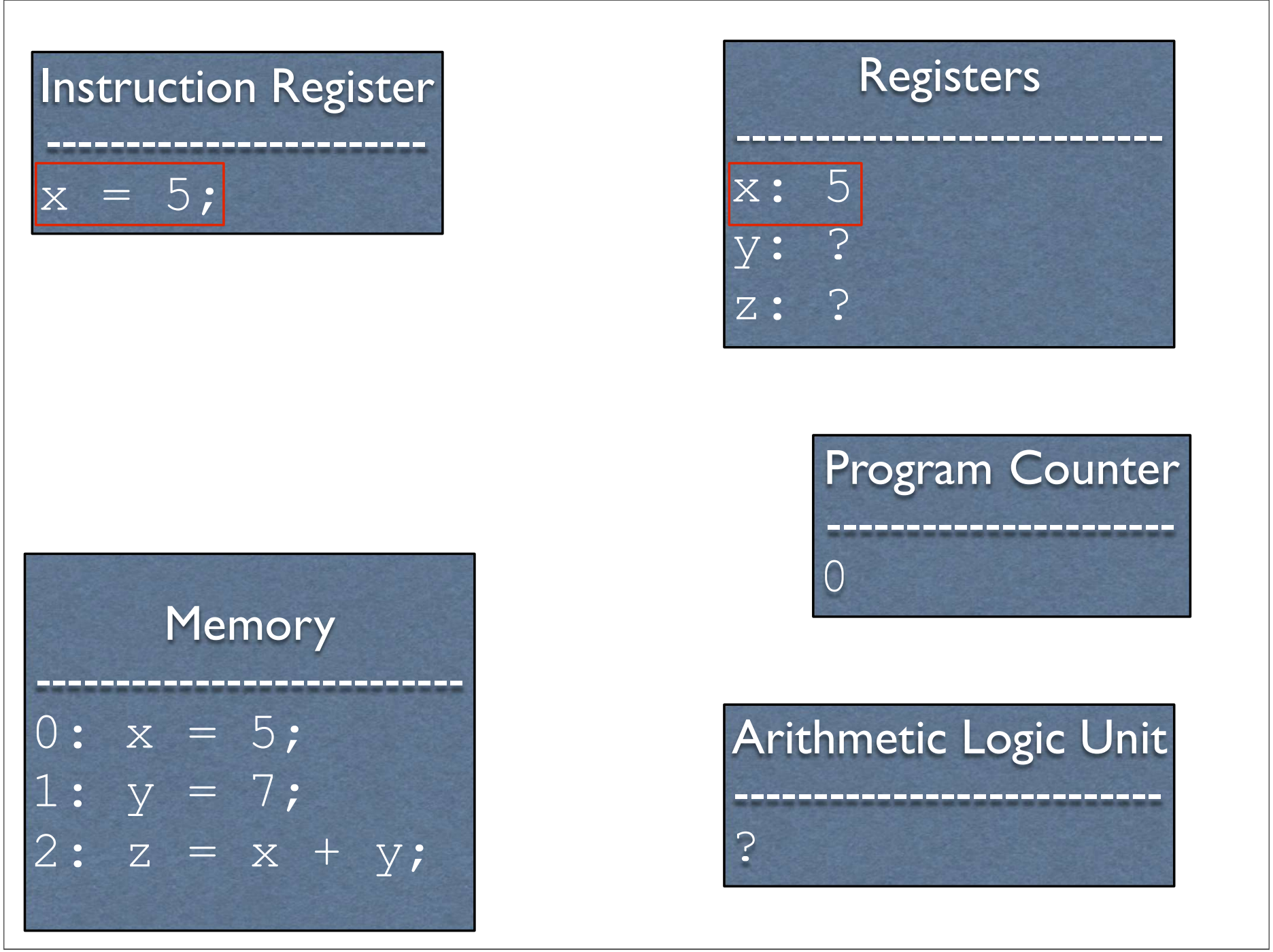


-Initialization occurs. Instructions are in memory, and the program counter is set to 0.

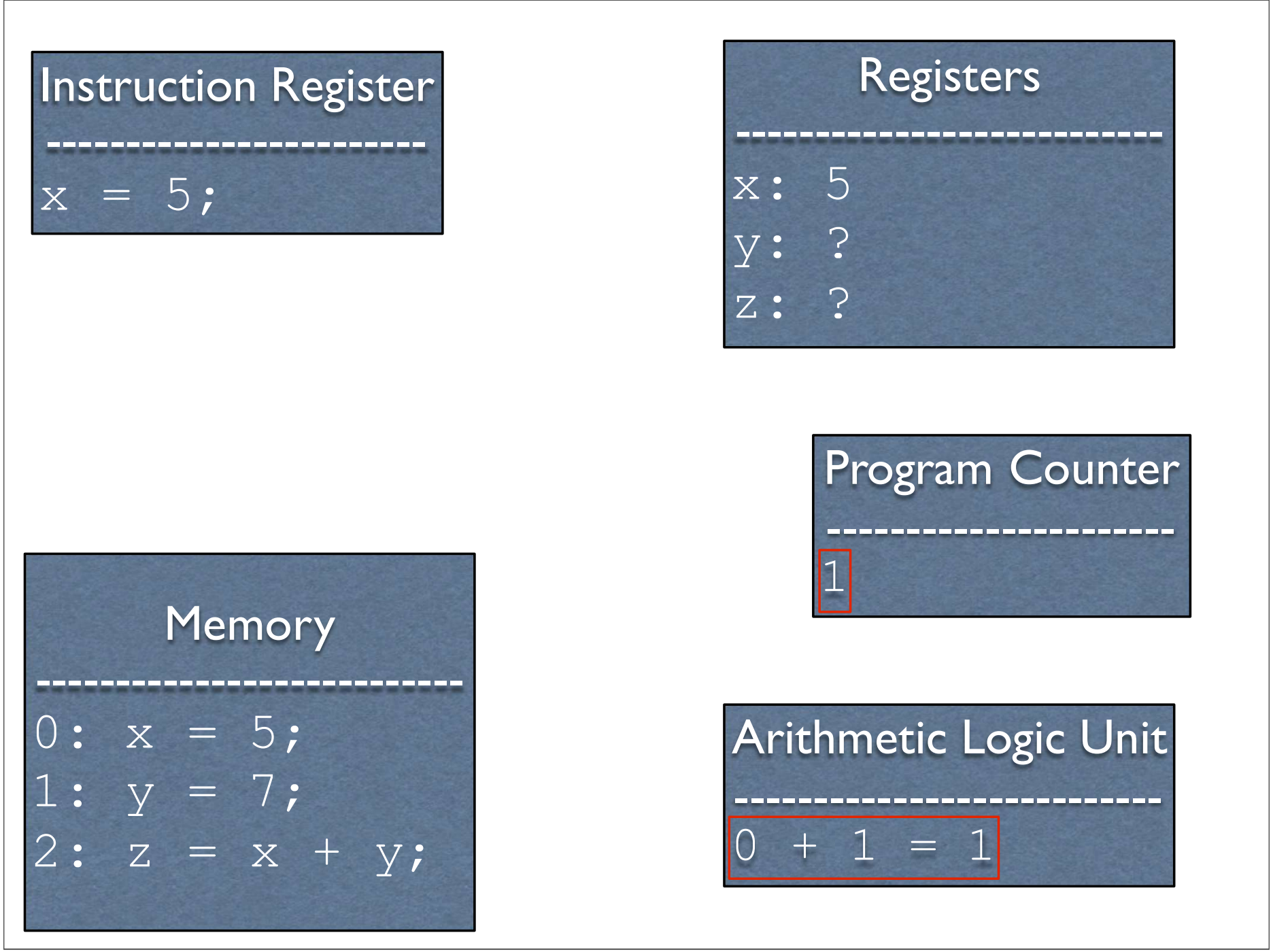
-In a real processor, there is some very basic initialization when it boots up, at which point the BIOS (and subsequently the OS) take over. From then on, its the responsibility of whatever is loaded in to set the contents of memory, the registers, and the program counter correctly. The operating systems class covers this stuff.



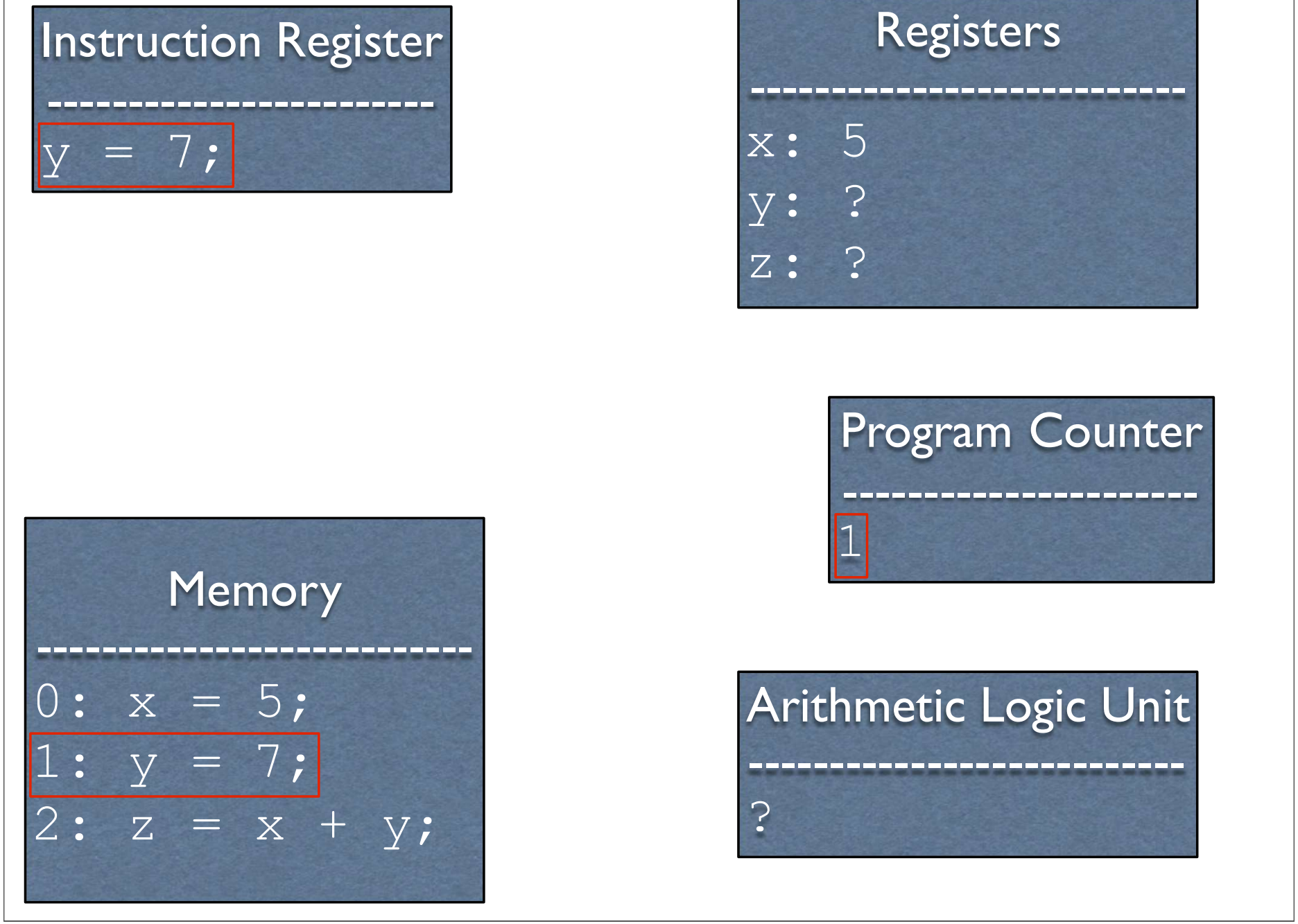
-We load instruction 0 into the instruction register



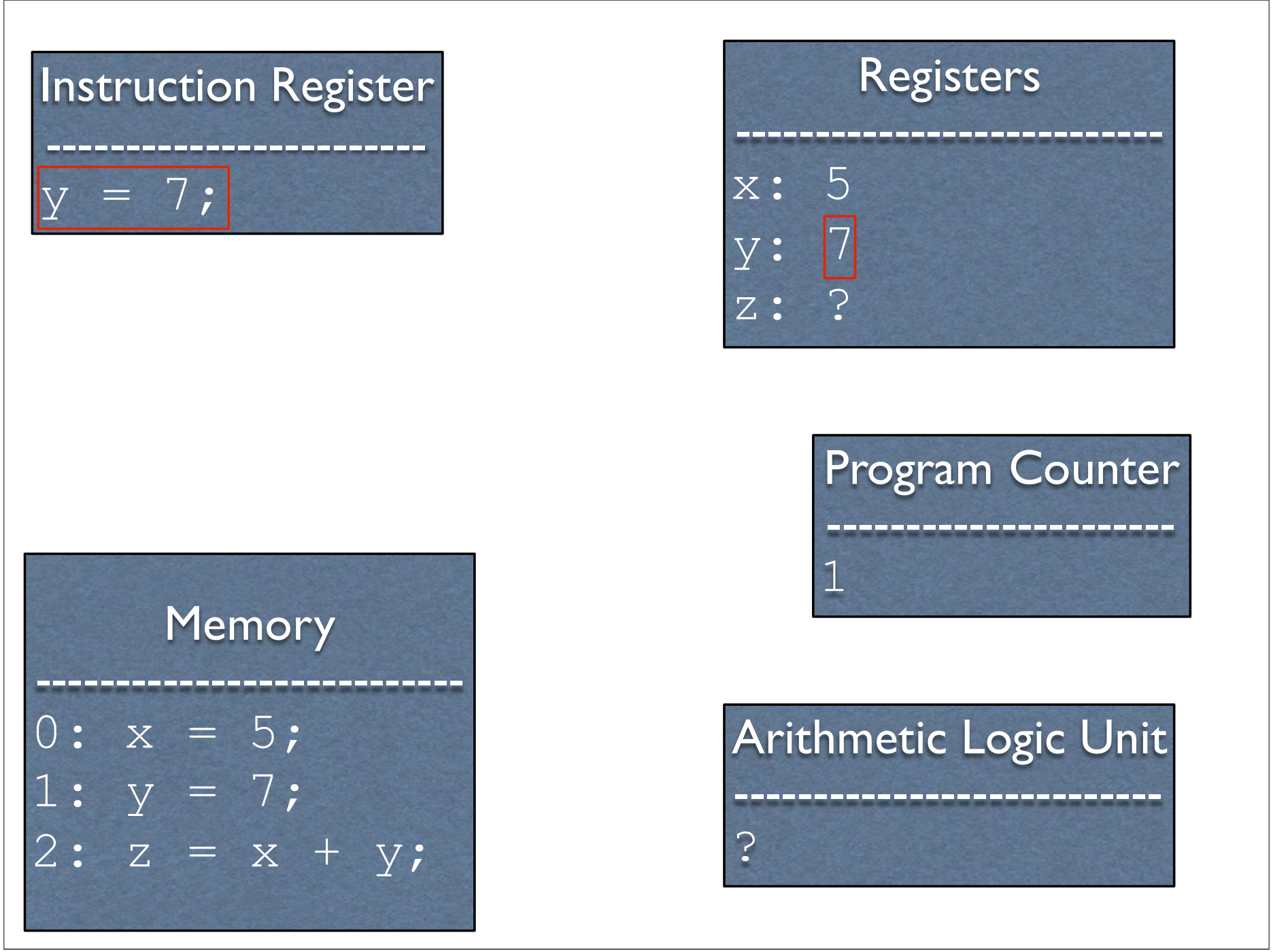
-We execute the instruction, setting register x to 5



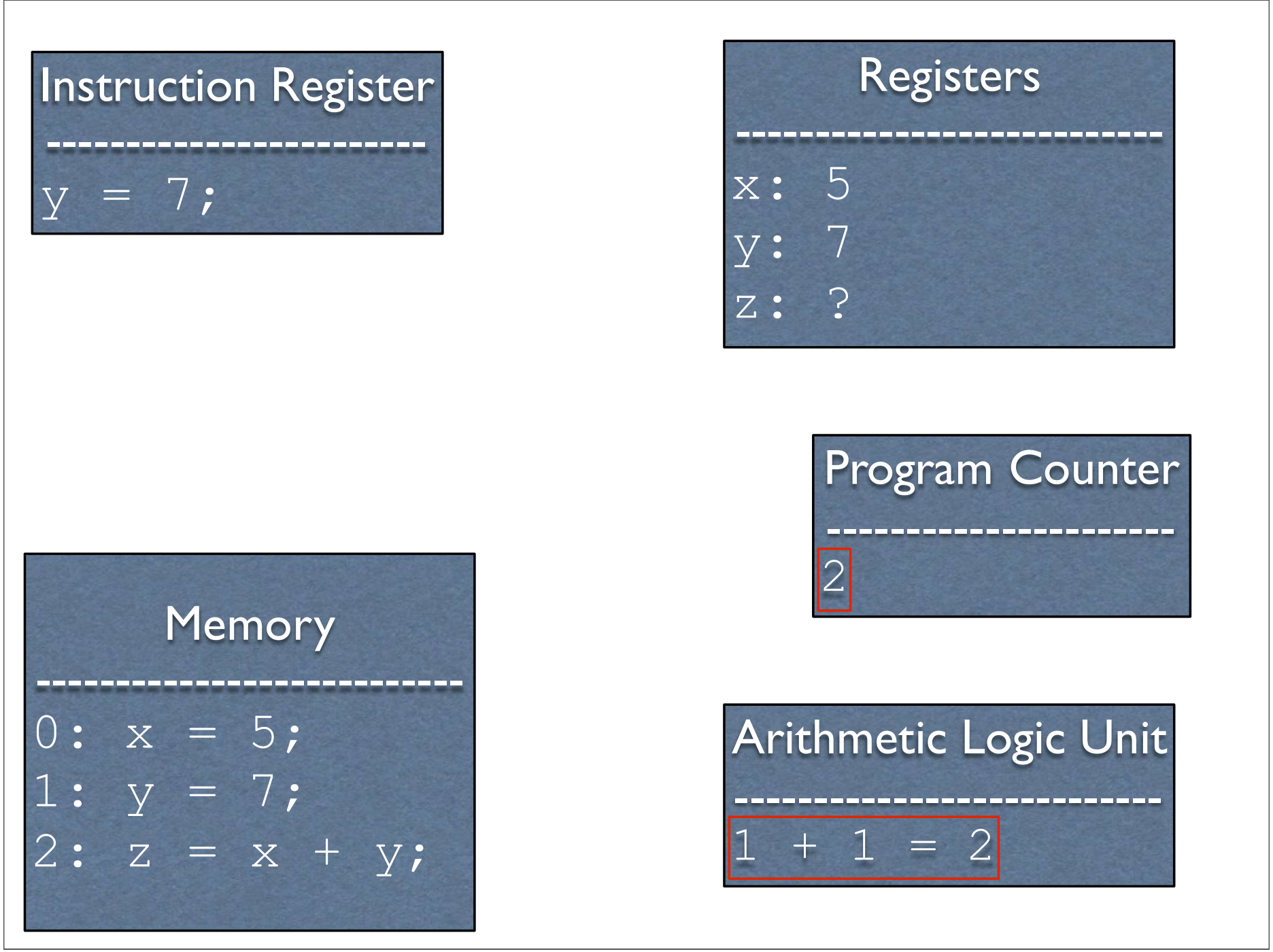
-We update the program counter



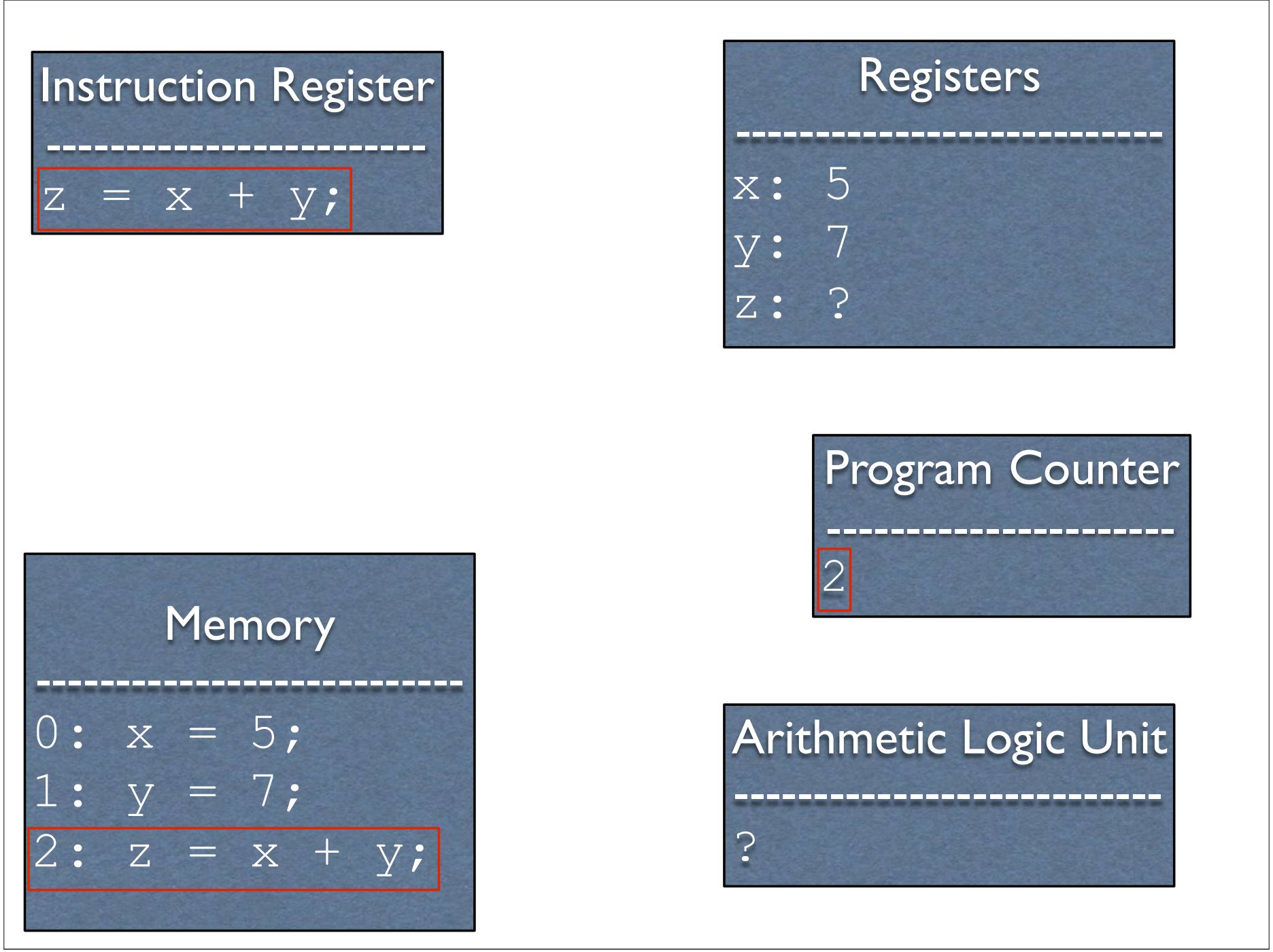
-Load in the next instruction



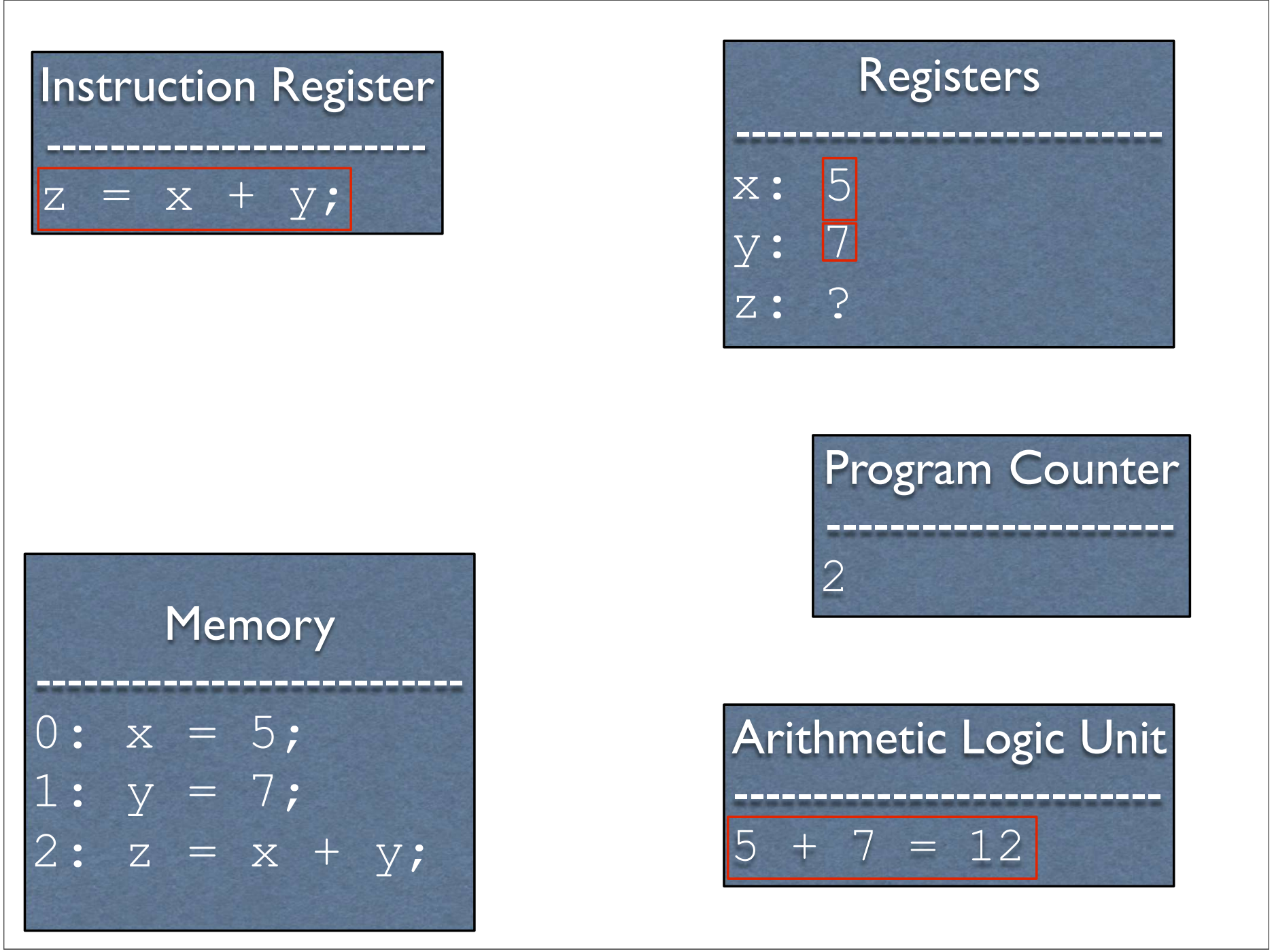
-We execute the instruction, setting register y to 7



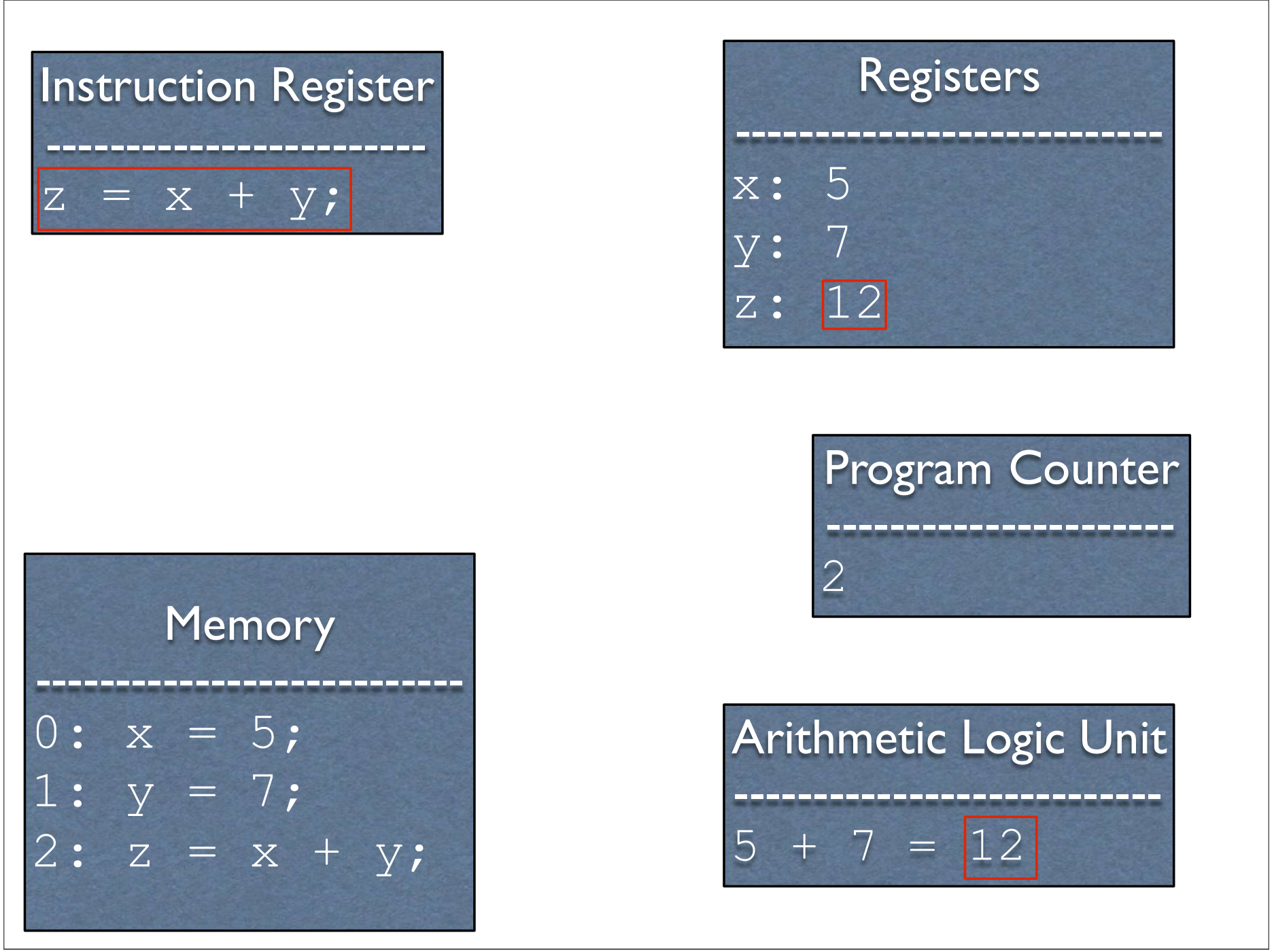
-We update the program counter



-Load in the next instruction



-Execute it, consulting the registers to get the values of x and y
-This consults the ALU



-The ALU sets the result

Microprocessor without Interlocked Pipelined Stages (MIPS)

https://en.wikipedia.org/wiki/MIPS_architecture

Why MIPS?

- Relevant in the embedded systems domain
- All processors share the same core concepts as MIPS, just with extra stuff
- ...but most importantly...

-Embedded devices include things like phones and microwaves.

It's Simpler

- RISC (Reduced Instruction Set Computer)
  - Dozens of instructions as opposed to hundreds
  - Lack of redundant instructions or special cases
- 5 stage pipeline vs. 24 stages

Code on MIPS

Original

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

MIPS

```
li   $t0,  5
li   $t1,  7
add  $t2,  $t0,  $t1
```

Code on MIPS

Original

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

MIPS

```
li   $t0, 5
li   $t1, 7
add  $t2, $t0, $t1
```

load **i**mmediate: put the given value into a register

$t0: **t**emporary register **0**

Code on MIPS

Original

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

MIPS

```
li   $t0, 5
li   $t1, 7
add  $t2, $t0, $t1
```

load **i**mmediate: put the given value into a register

$t1: **t**emporary register **1**

Code on MIPS

Original

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

MIPS

```
li   $t0,  5
li   $t1,  7
add  $t2,  $t0,  $t1
```

add: add the rightmost registers, putting the result in the first register

$t2: temporary **r**egister **2**

Available Registers

- 32 registers in all
- For the moment, we will only consider registers $t0 - $t9

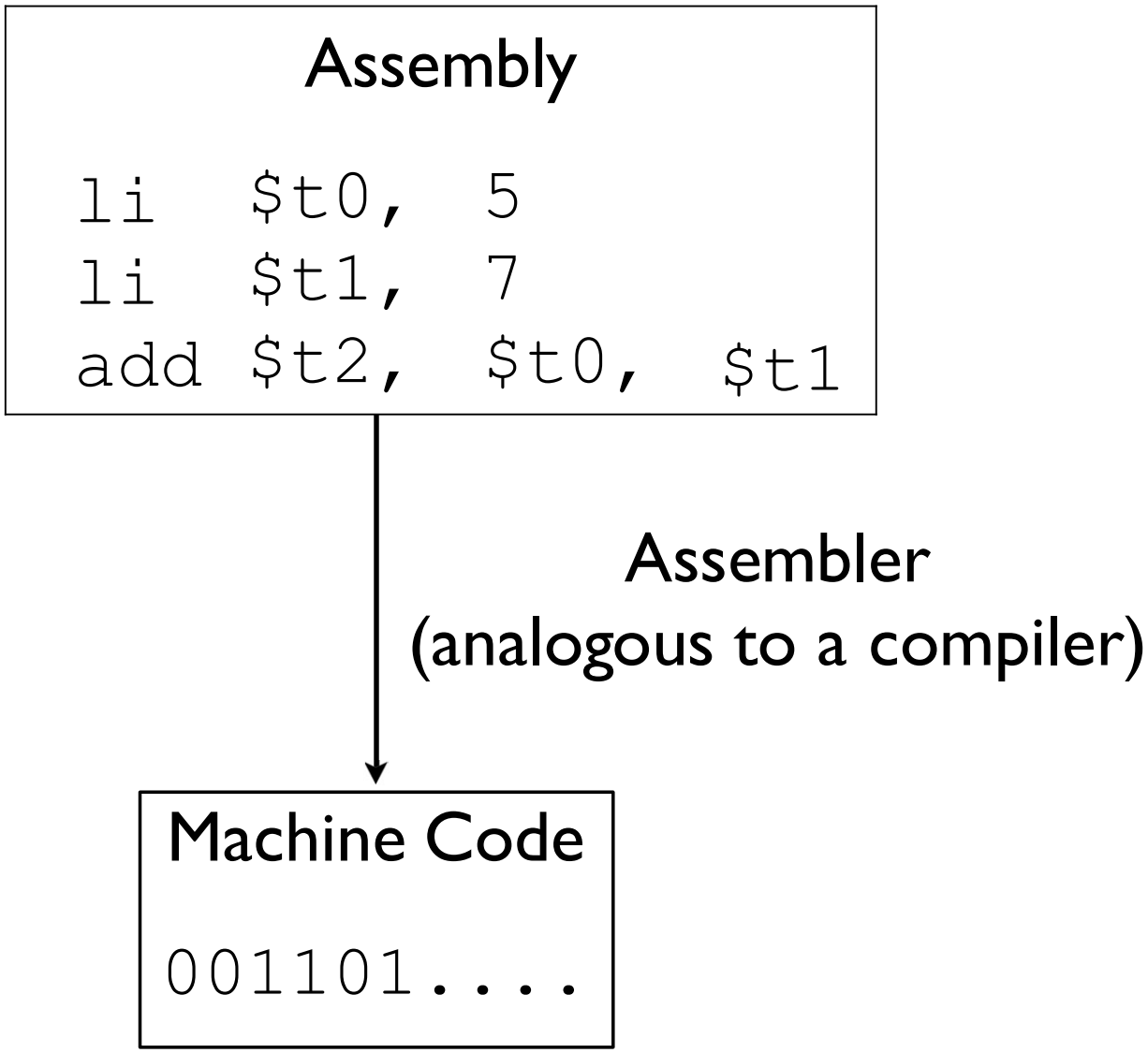
Assembly

- The code that you see below is *MIPS assembly*
- Assembly is *almost* what the machine sees. For the most part, it is a direct translation to binary from here (known as *machine code)*

```
li   $t0,  5
li   $t1,  7
add  $t2,  $t0,  $t1
```

-More on why I said "the most part" later. Psuedo instructions are translated to other instructions. Branches also need calculation to occur (for labels), and there are caveats about the instruction immediately after a branch

Workflow

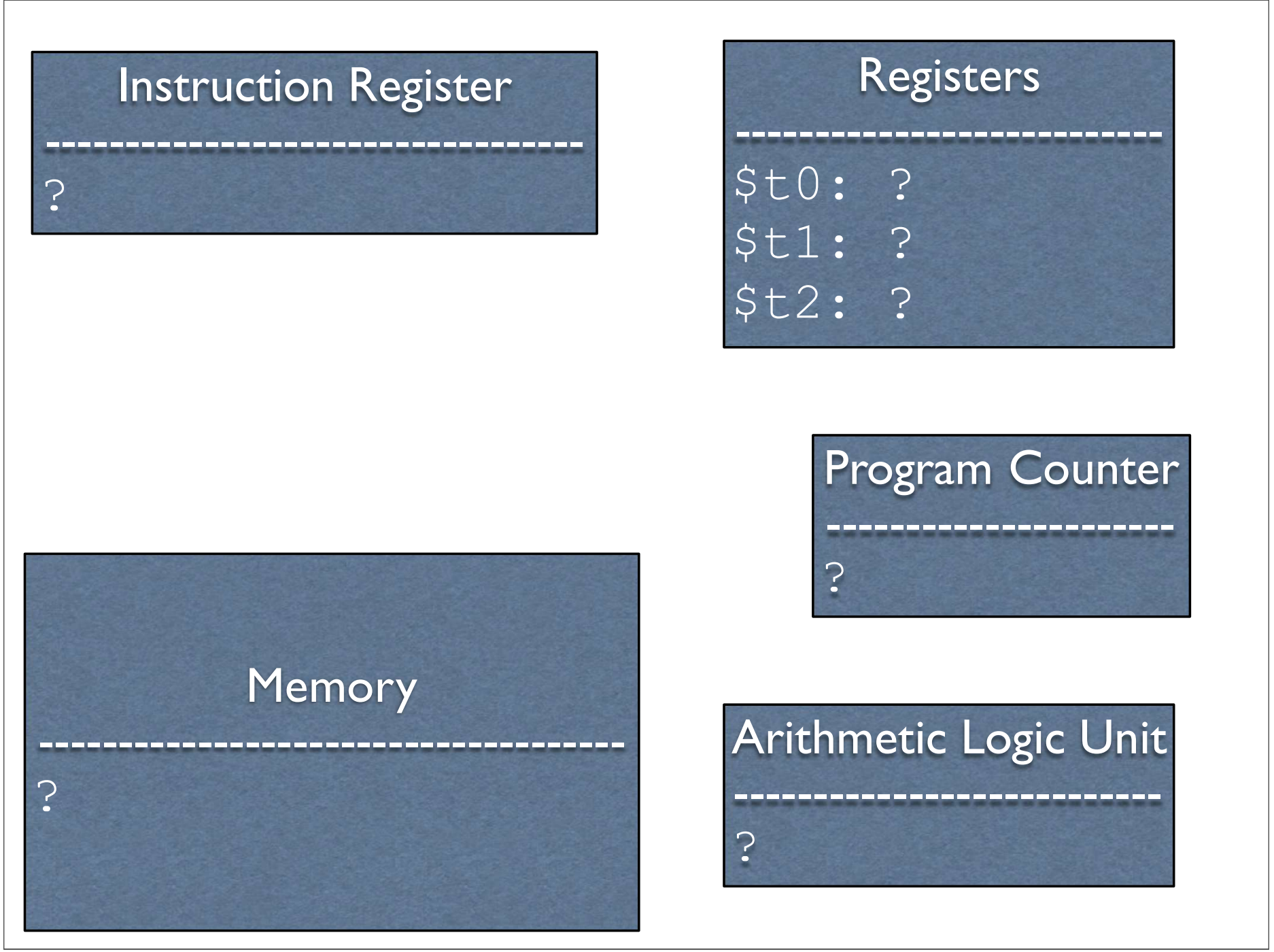

Assembly
li $t0, 5
li $t1, 7
add $t2, $t0, $t1
Assembler
(analogous to a compiler)
Machine Code
001101....

Machine Code

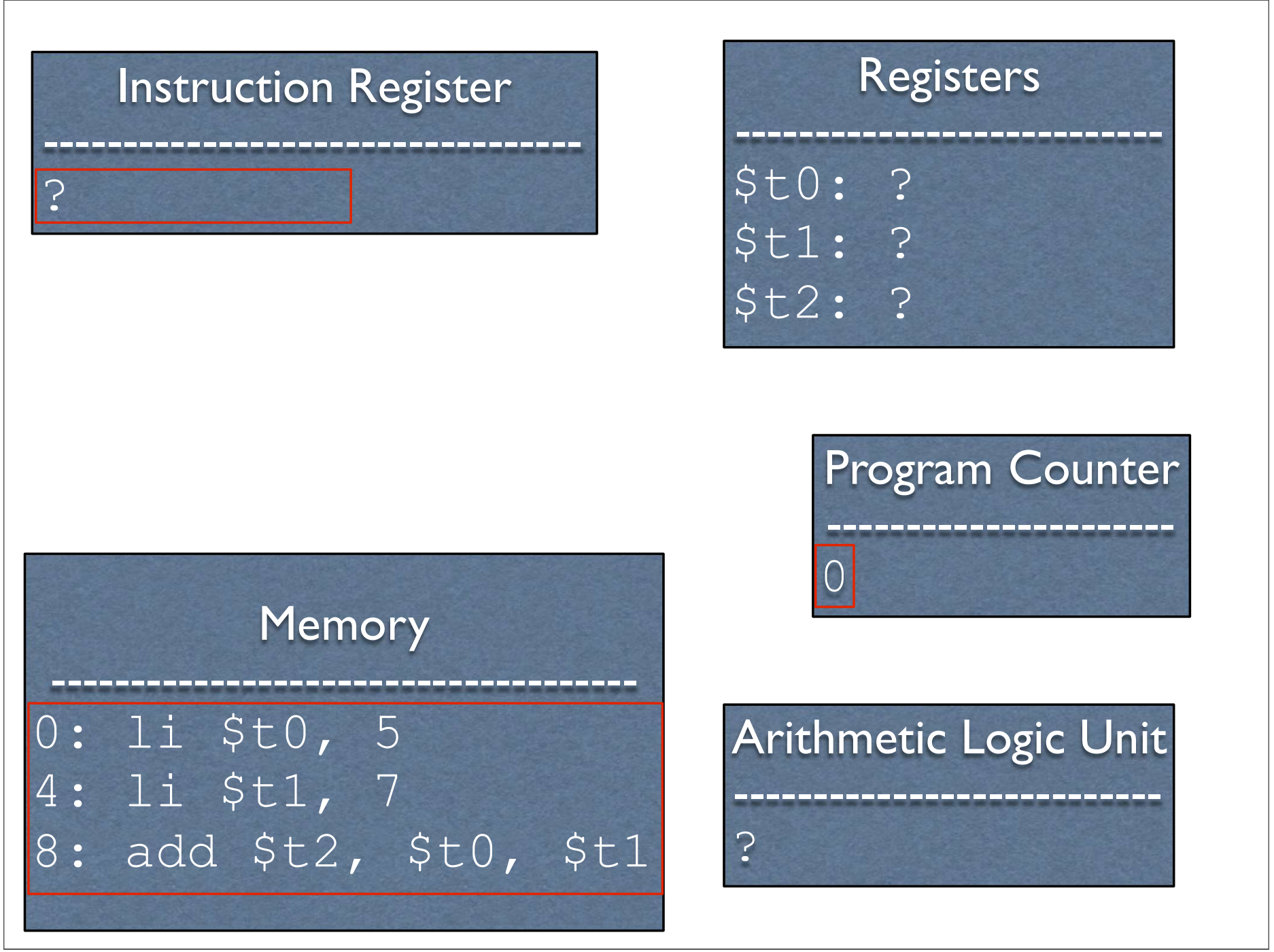
- This is what the process actually executes and accepts as input
- Each instruction is represented with 32 bits
- Three different instruction formats; for the moment, we'll only look at the R format

```
add $t2, $t0, $t1
```

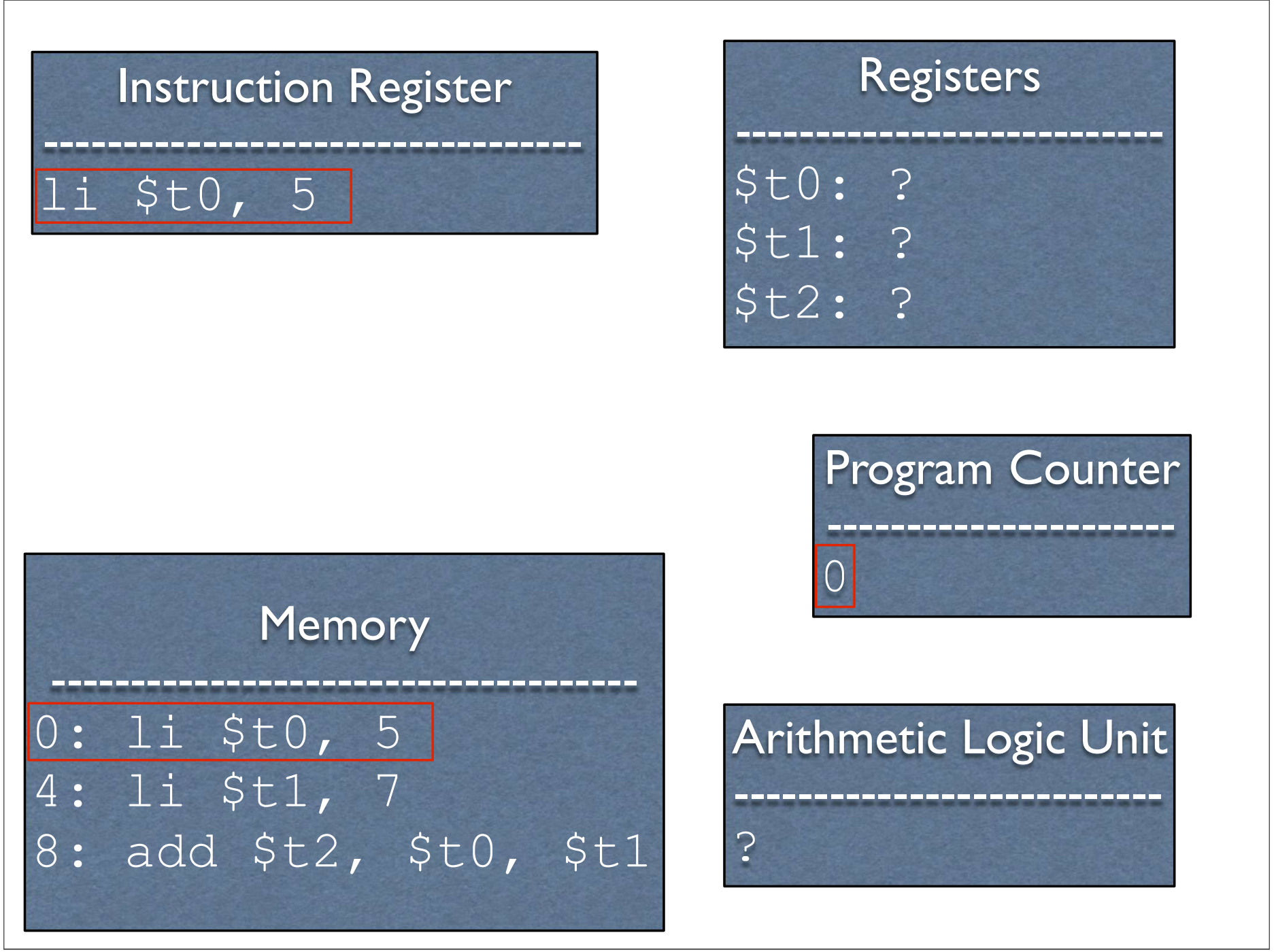
-Let's start to decipher the MIPS green sheet
-Converting to machine code is mostly one-to-one: just put the right bits in the right places



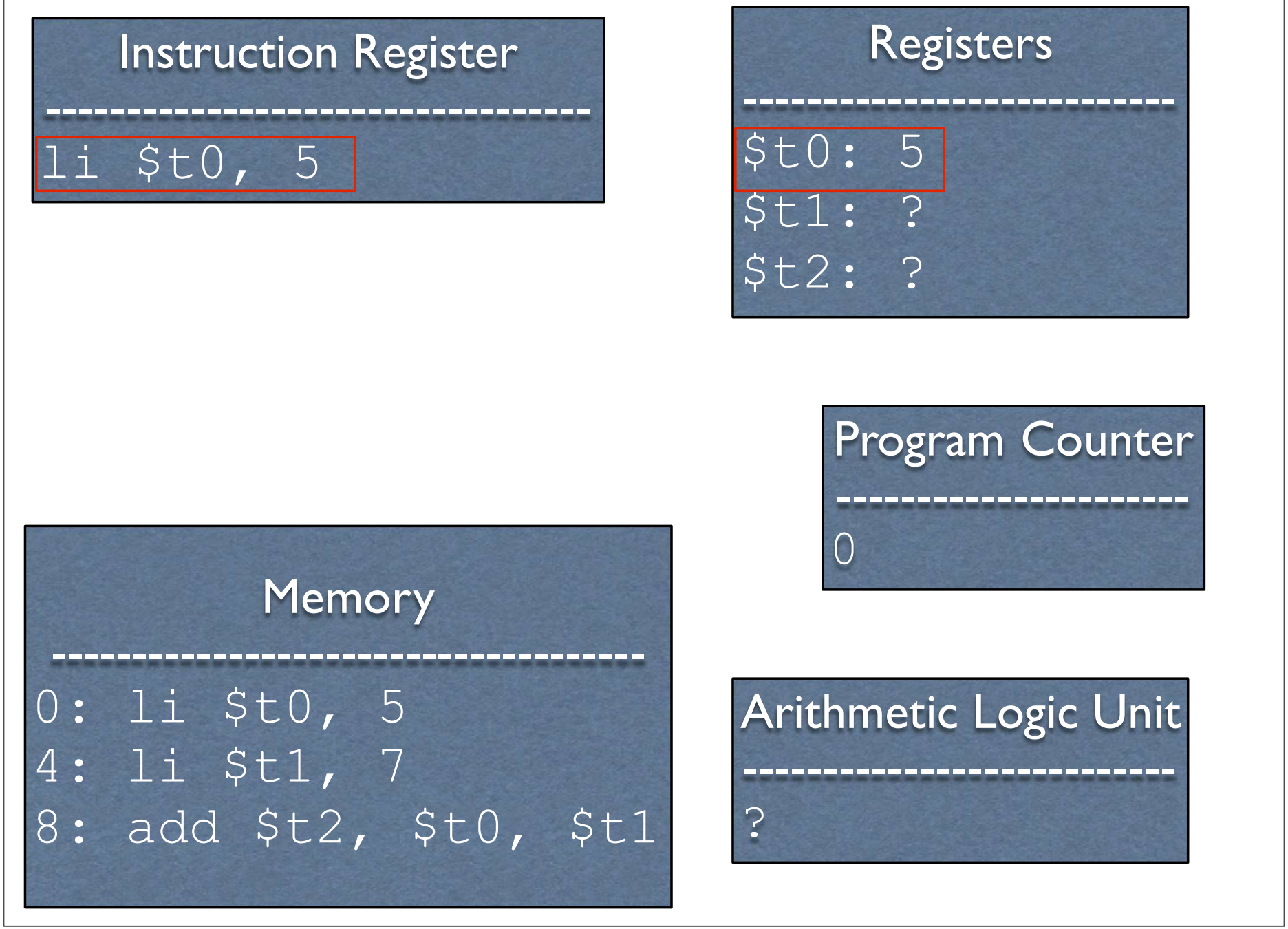
-All the hardware, before initialization



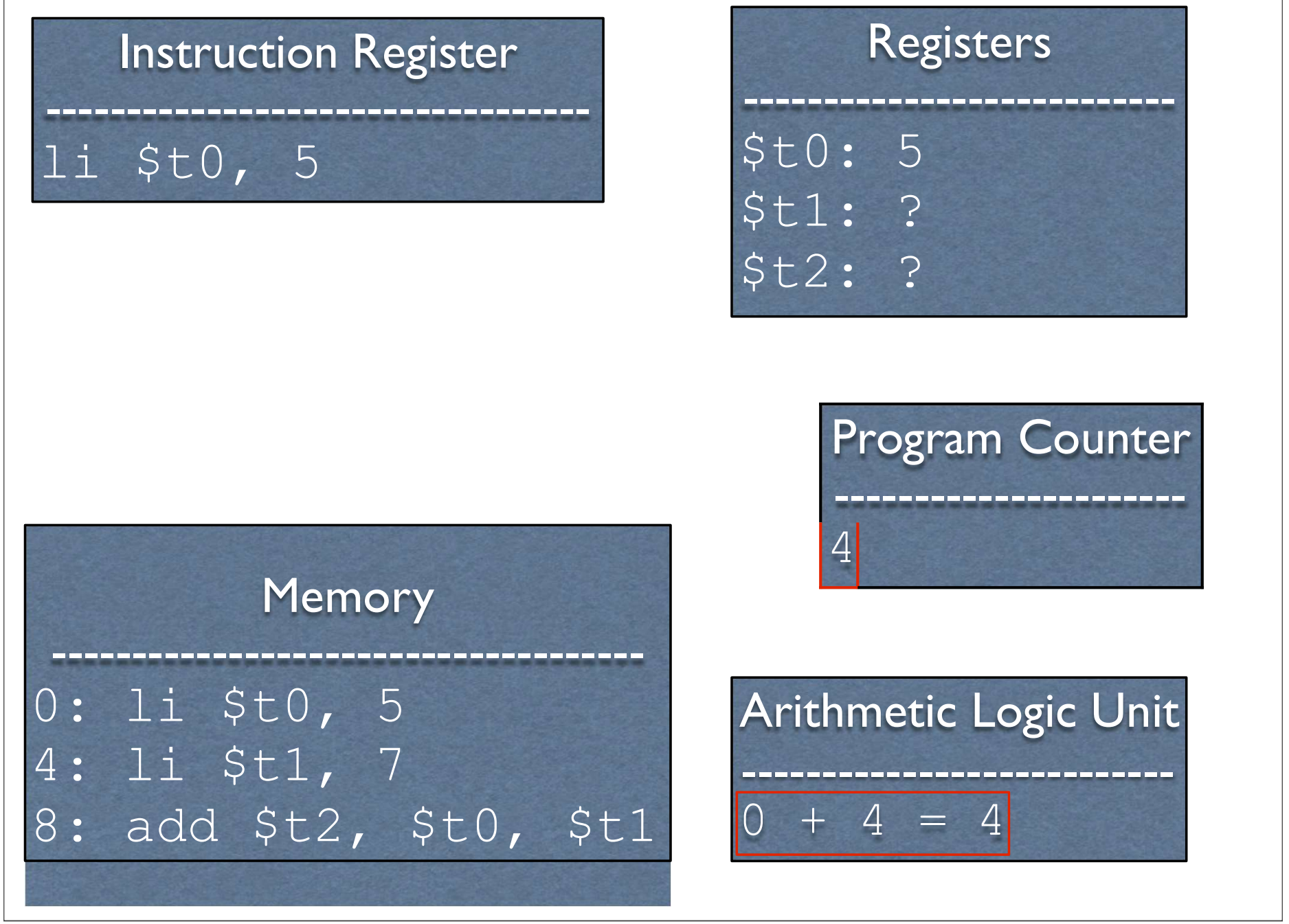
-Initialization occurs. Instructions are in memory, and the program counter is set to 0.
-Note that we address by byte. Given that addresses are 32 bits wide (4 bytes long), each address is aligned to four bytes



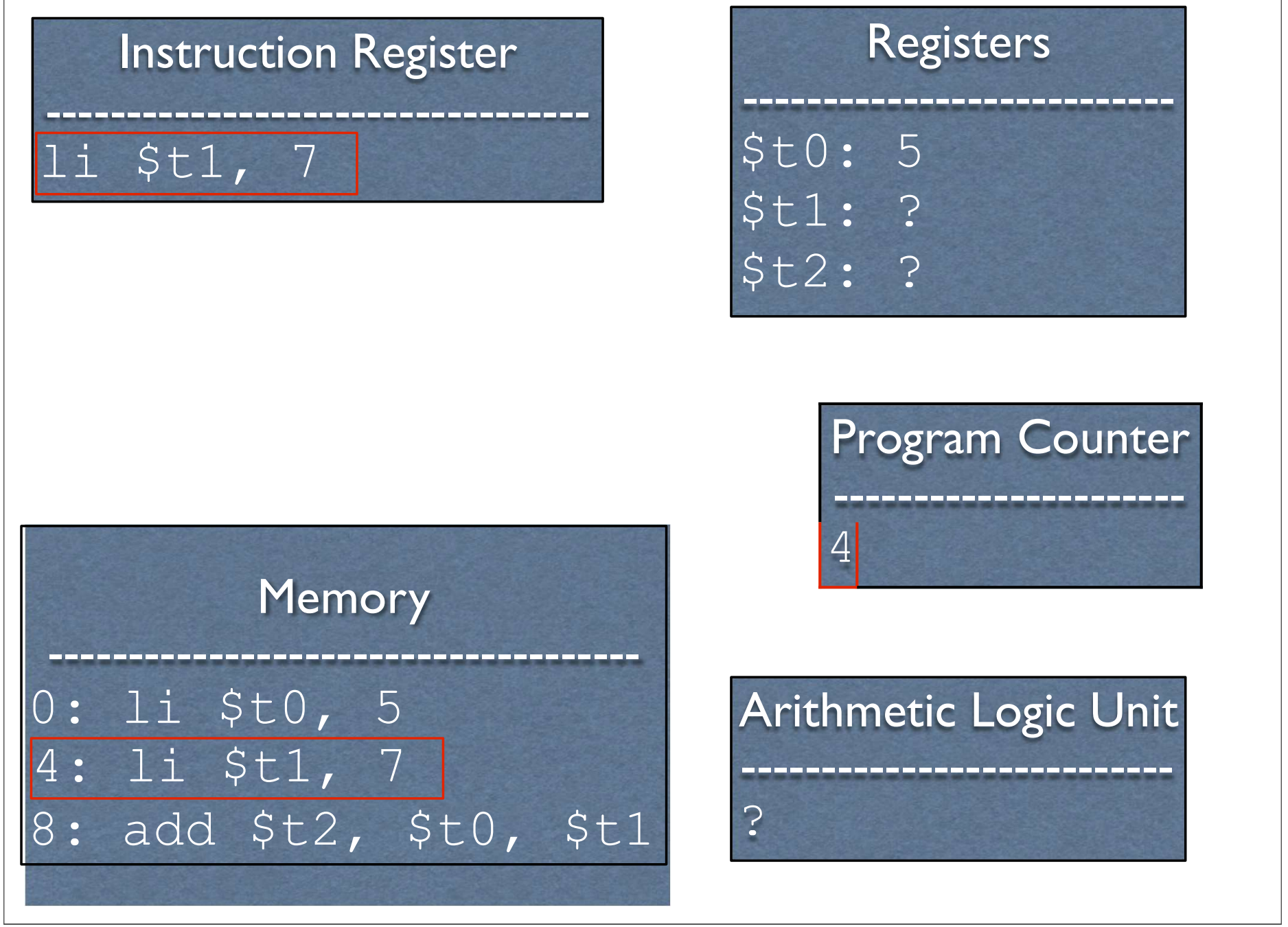
-We load instruction 0 into the instruction register



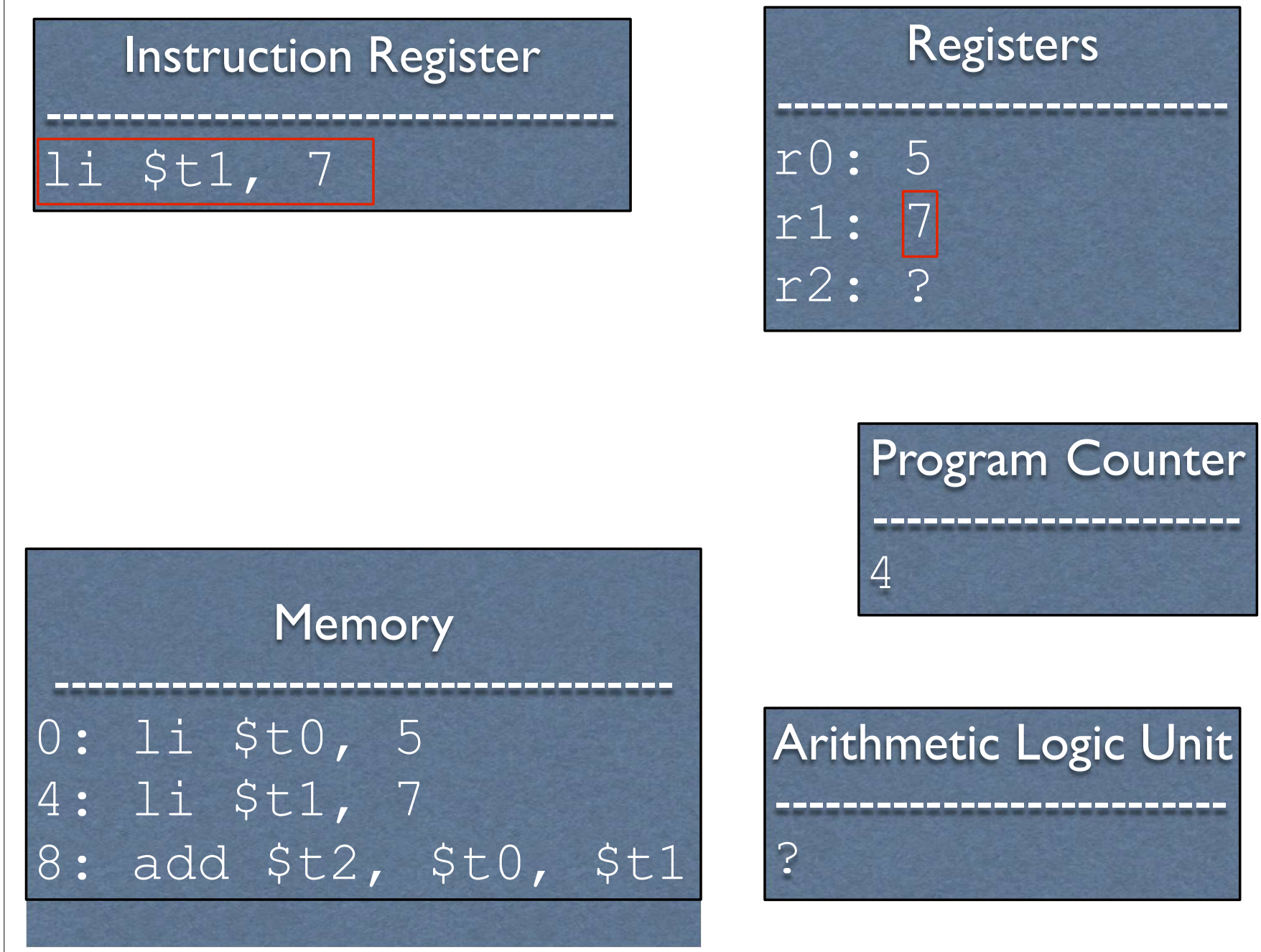
-We execute the instruction, setting register $t0 to 5



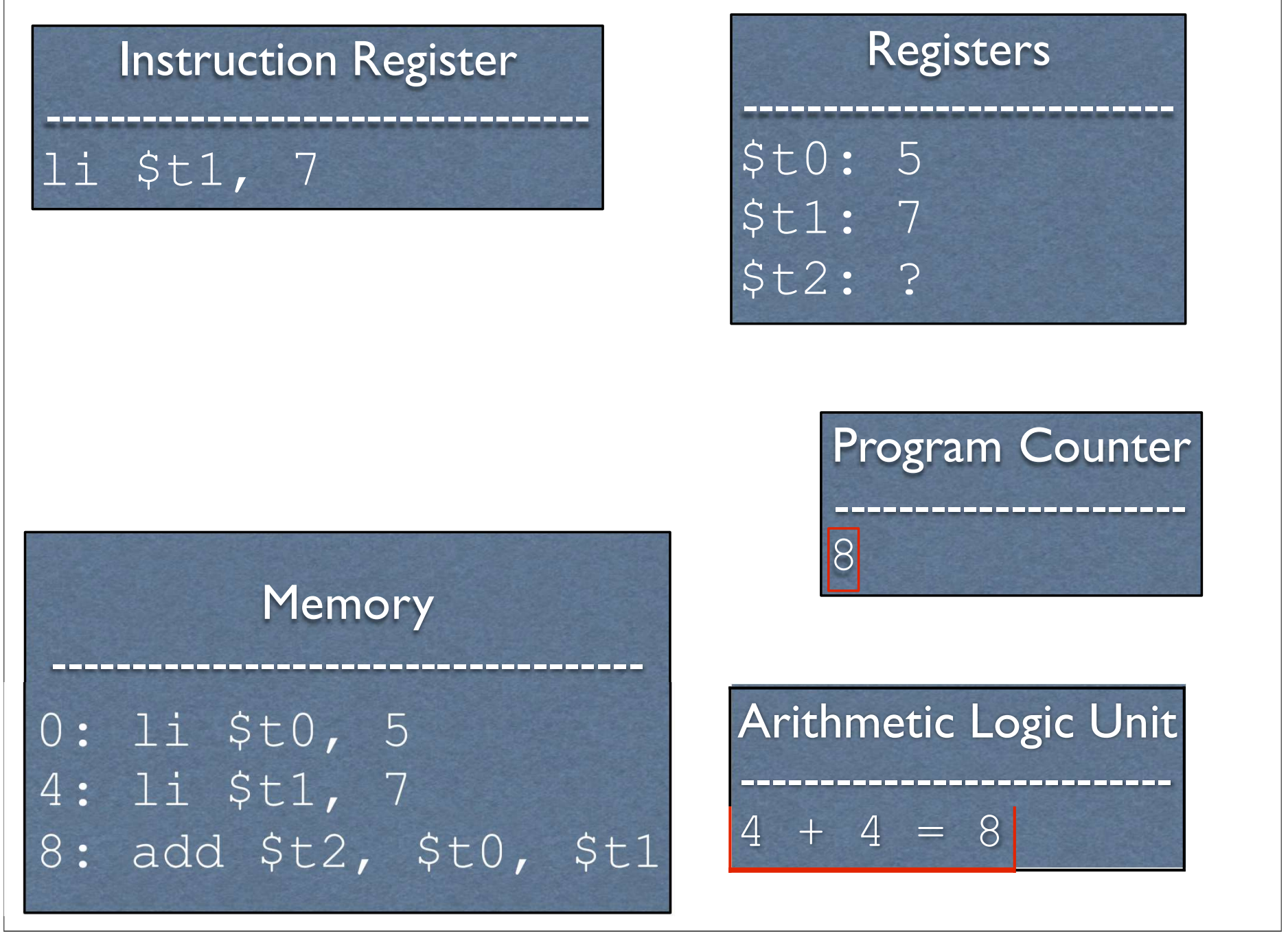
-We update the program counter
-Note that we add 4 instead of one, as instructions are four bytes long



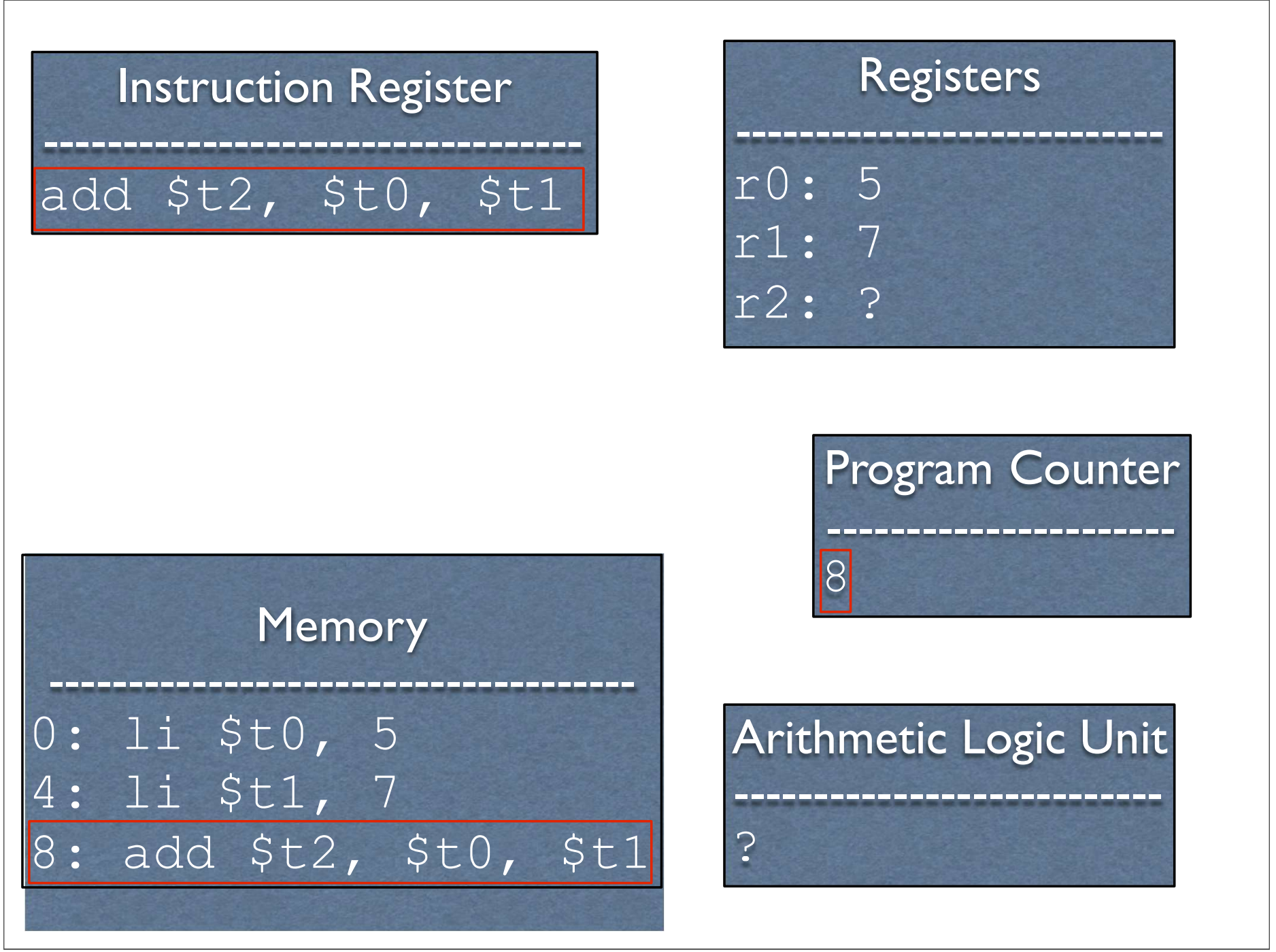
-Load in the next instruction



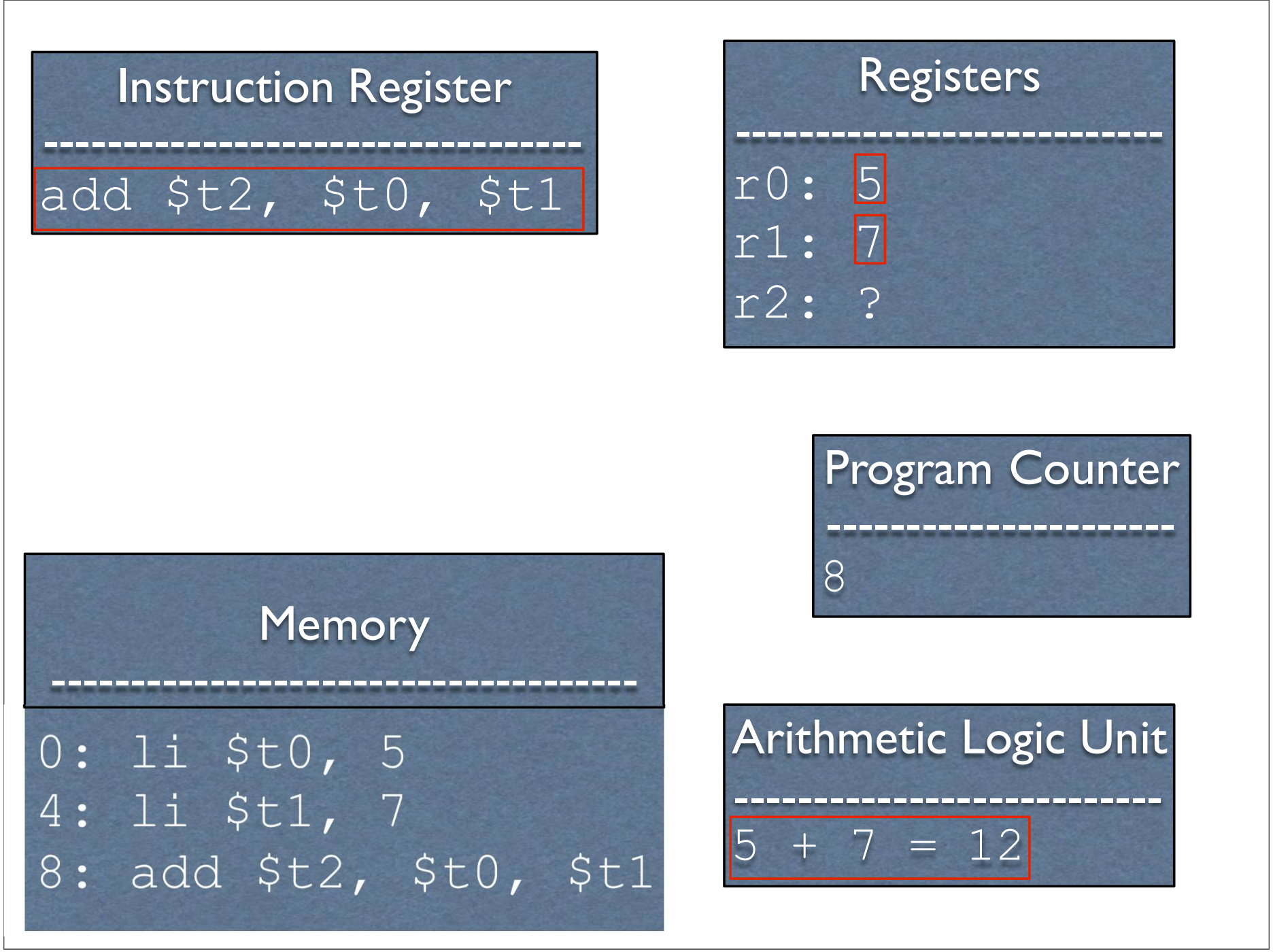
-We execute the instruction, setting register $t1 to 7



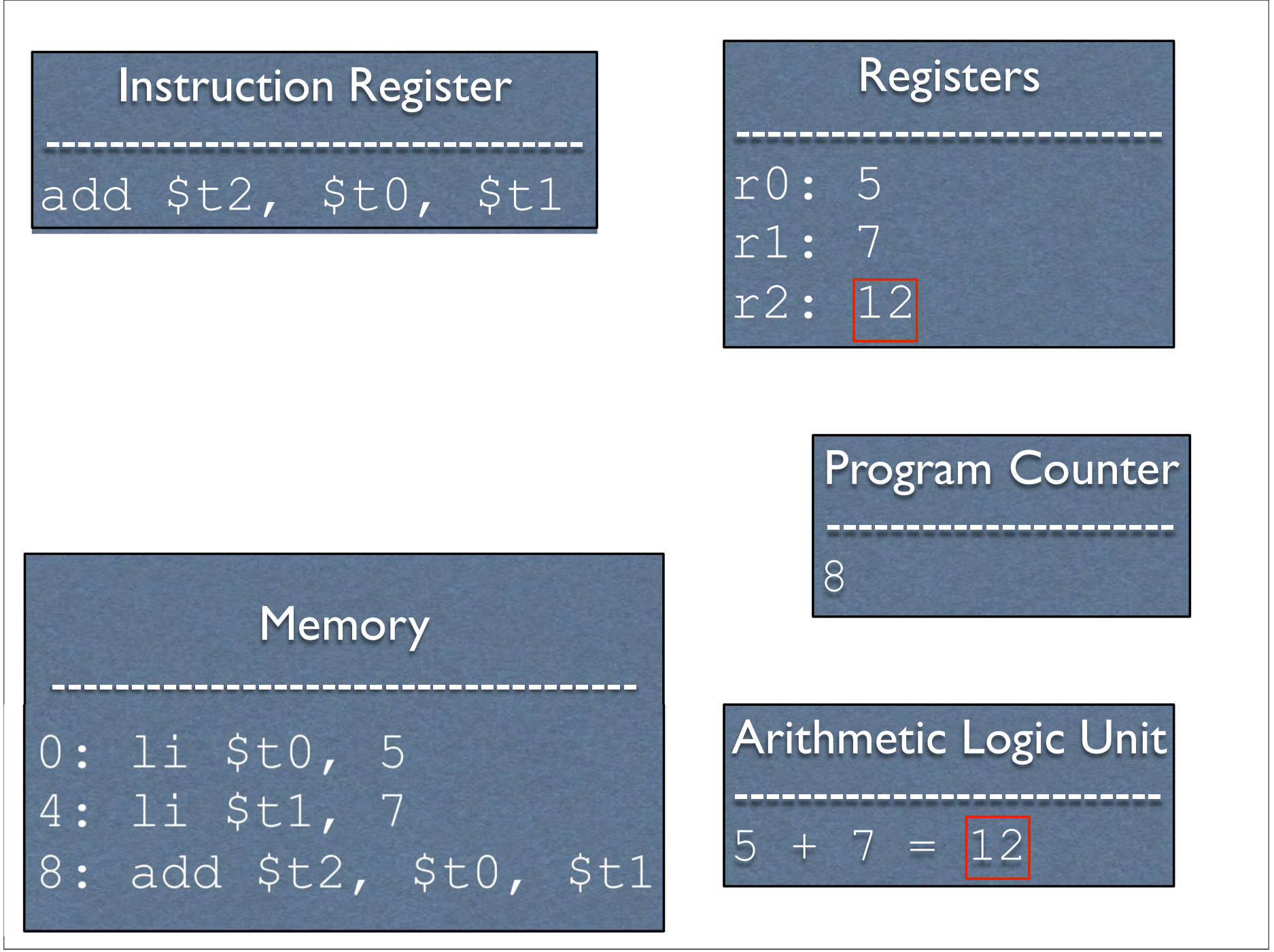
-We update the program counter



-Load in the next instruction



-Execute it, consulting the registers to get the values of $t0 and $t1
-This consults the ALU



-The ALU sets the result

Adding More Functionality

- We need a way to display the result
- What does this entail?

-Actually quite the tall order

Adding More Functionality

- We need a way to display the result
- What does this entail?
  - Input/output. This entails talking to devices, which the operating system handles
  - We need a way to tell the operating system to kick in

–Actually quite the tall order

Talking to the OS

- We are going to be running on a MIPS emulator, SPIM
- We cannot directly access system libraries (they aren't even in the same machine language)
- How might we print something?

SPIM Routines

- MIPS features a `syscall` instruction, which triggers a *software interrupt or exception*
- Outside of an emulator, these pause the program and tell the OS to check something
- Inside the emulator, it tells the **emulator** to check something

`syscall`

- So we have the OS/emulator's attention. But how does it know what we want?

syscall

- So we have the OS/emulator's attention. But how does it know what we want?
  - It has access to the registers
  - Put special values in the registers to indicate what you want

(Finally) Printing an Integer

- For SPIM, if register `$v0` contains 1, then it will print whatever integer is stored in register `$a0`
- Note that `$v0` and `$a0` are distinct from `$t0` – `$t9`

-Other SPIM utilities available via syscall: https://www.doc.ic.ac.uk/lab/secondyear/spim/node8.html

Augmenting with Printing

```
li        $t0, 5
li        $t1, 7
add       $t2, $t0,  $t1

li        $v0, 1
move      $a0, $t2
syscall
```

Exiting

- If you are using SPIM, then you need to say when you are done as well
- How might this be done?

Exiting

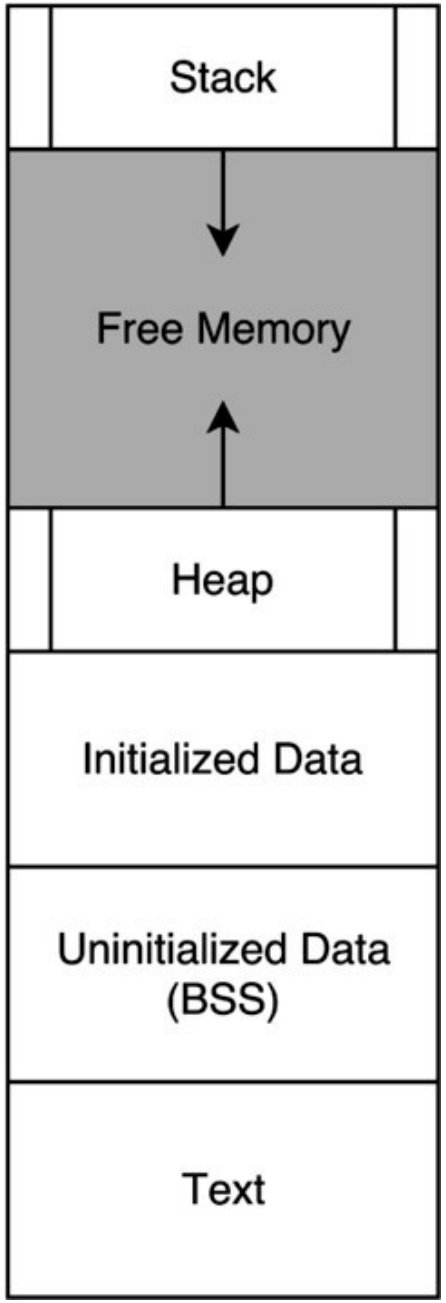
- If you are using MIPS, then you need to say when you are done as well
- How might this be done?
  - `syscall` with a special value in `$v0` (specifically, 10 decimal)

Augmenting with Exiting

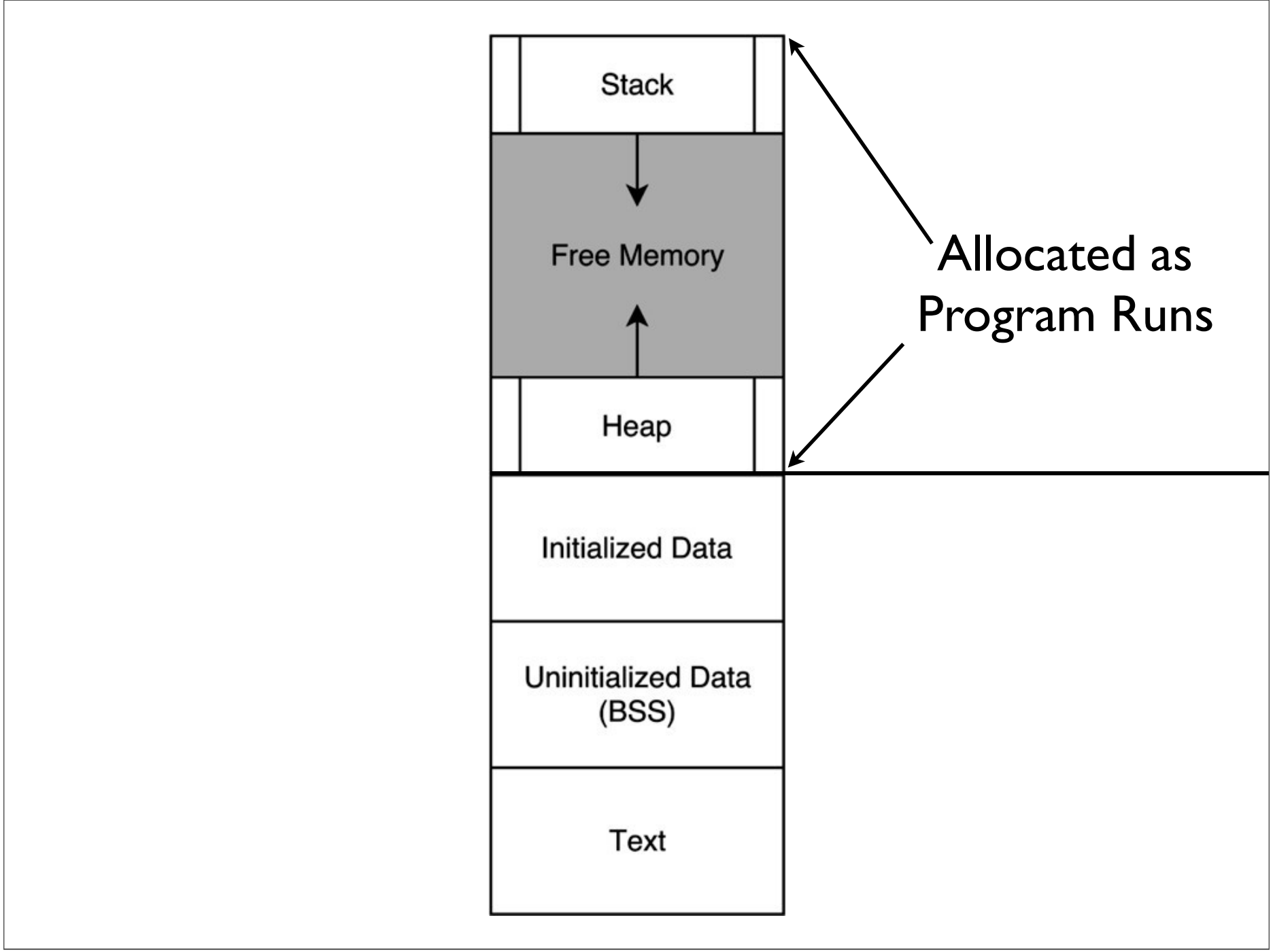
```
// load immediate - load
specific value in a register
li $t0, 5 // %t0 = 5
li $t1, 7
add $t2, $t0, $t1

li $v0, 1 // print int
// move destination, source
move $a0, $t2
syscall

li $v0, 10
syscall
```

Making it a Full Program

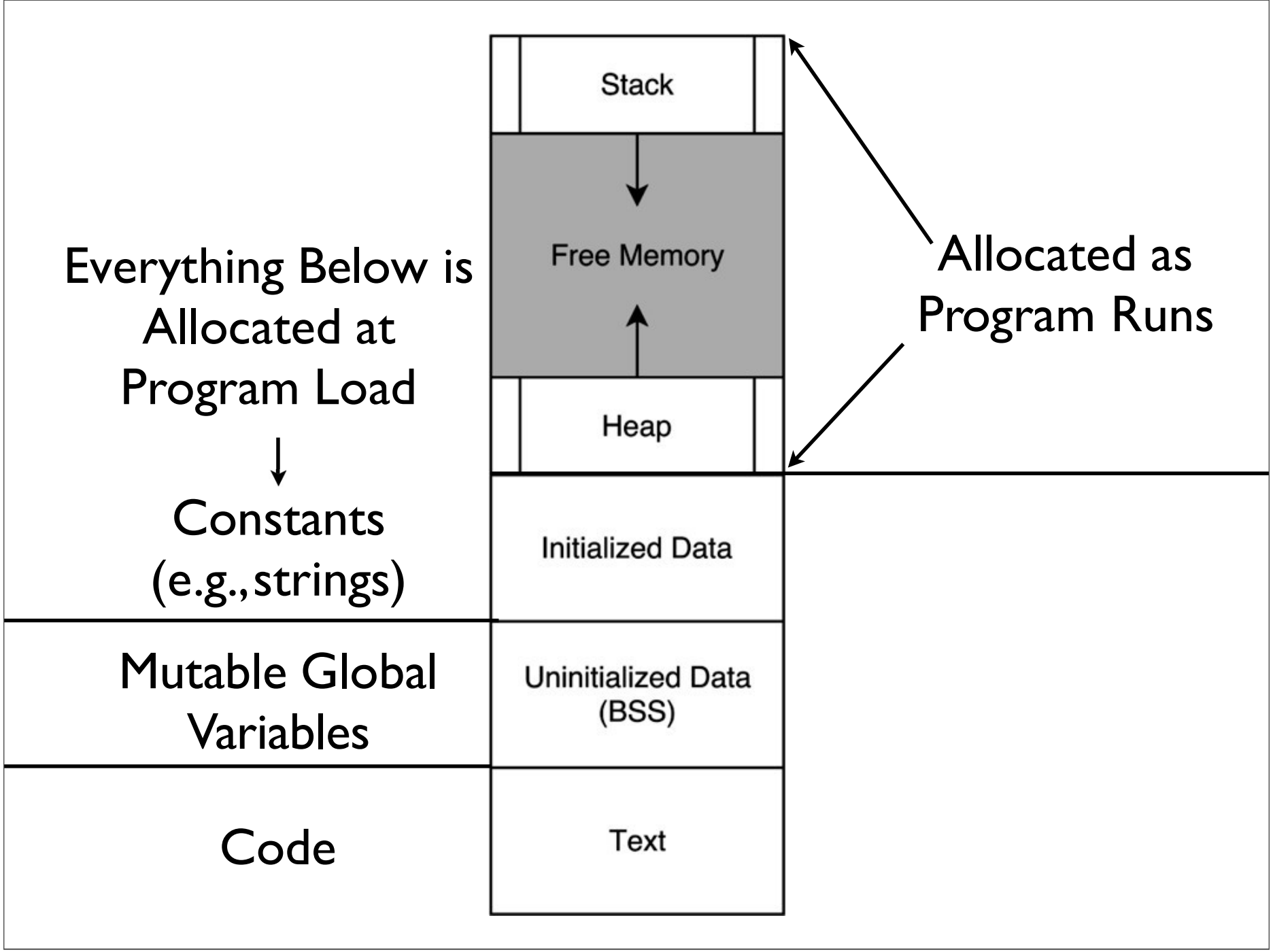
- Everything is just a bunch of bits
- We need to tell the assembler which bits should be placed where in memory



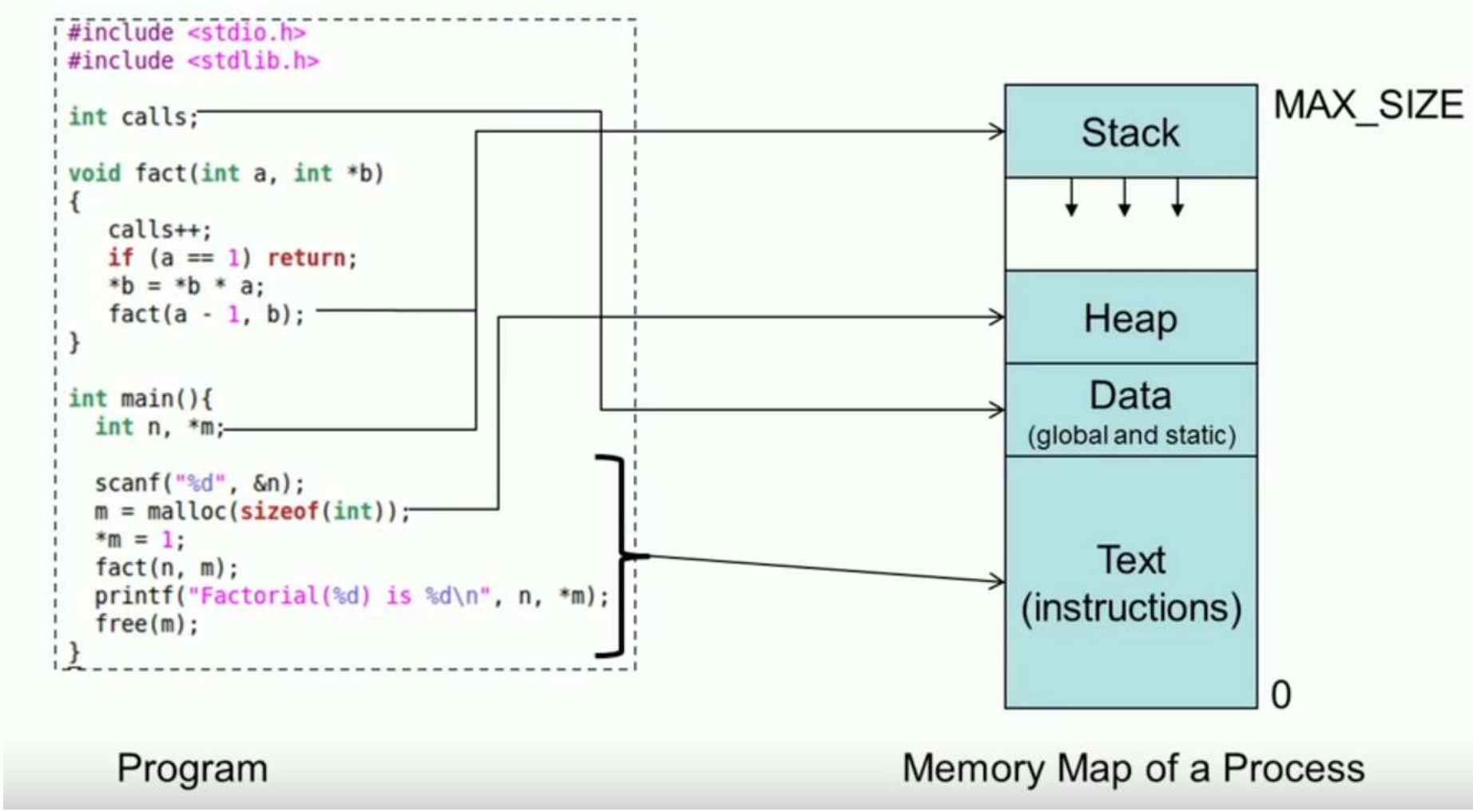
-Image source: https://en.wikipedia.org/wiki/Data_segment
-Representation of a program in memory
-What do you recognize?



-Image source: https://en.wikipedia.org/wiki/Data_segment
-You've seen these two before
-What might the rest be?

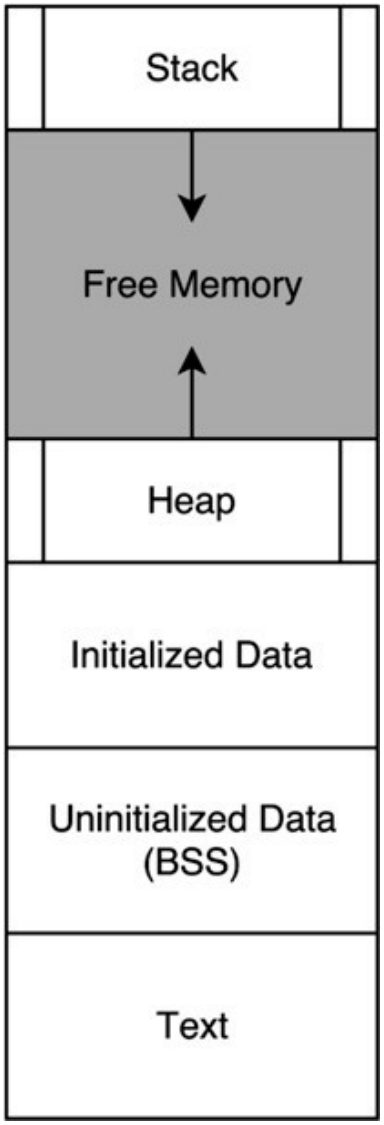


-Image source: https://en.wikipedia.org/wiki/Data_segment


#include <stdio.h>
#include <stdlib.h>
int calls;
void fact(int a, int *b)
{
calls++;
if (a == 1) return;
*b = *b * a;
fact(a - 1, b);
}
int main(){
int n, *m;
scanf("%d", &n);
m = malloc(sizeof(int));
*m = 1;
fact(n, m);
printf("Factorial(%d) is %d\n", n, *m);
free(m);
}
Stack
Heap
Data
(global and static)
Text
(instructions)
MAX_SIZE
0
Program
Memory Map of a Process

Marking Code

Use a `.text` *directive* to specify code section




```
.text

li $t0, 5
li $t1, 7
add $t2, $t0, $t1

li $v0 1
move $a0, $t2
Syscall

li $v0, 10
syscall
```

-Directives tell the assembler to do something

Running with SPIM:
(add2.s)

`move` Instruction

- The move instruction does not actually show up in SPIM
- It is a pseudoinstruction which is translated into an actual instruction

Original	Actual
`move $a0, $t2`	`addu $a0, $zero, $t2`

`$zero`

- Specified like a normal register, but does not behave like a normal register
  - Write to `$zero` are not saved
  - Reads from `$zero` always return 0

But Why?

- Why have move as a pseudoinstruction instead of as an actual instruction?

But Why?

- Why have move as a pseudoinstruction instead of as an actual instruction?
  - One less instruction to worry about
  - One design goal of RISC is to cut out redundancy

load intermediate

- The `li` instruction does not actually show up in SPIM
- It is a *pseudoinstruction* which is translated into actual instructions
- Why might `li` work this way?
  - Hint: instructions and registers are both 32 bits long

load intermediate

- The `li` instruction does not actually show up in SPIM
- It is a *pseudoinstruction* which is translated into actual instructions
- Why might `li` work this way?
  - Not enough room in one instruction to fit everything within 32 bits
  - I-type instructions only hold 16 bits

Assembly Coding Strategy

- Best to write it in C-like language, then translate down by hand
- This gets more complex when we get into control structures and memory

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

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

More Examples

- `swap.asm`
- `negate.asm`
- `mult80.asm`
- `div80.asm`
- `hello_world.asm`
- `read_and_print_int.asm`

Branches

Conditionals

Using all the instructions learned so far, how might we code up the following?

```
if (x == 0) {
  printf("x is zero");
}
```

Conditionals

Using all the instructions learned so far, how might we code up the following?

```
if (x == 0) {
  printf("x is zero");
}
```

Answer: We can't (realistically).

Conditionals

- What do we need to implement this?

```
if (x == 0) {
  printf("x is zero");
}
```

Conditionals

- What do we need to implement this?
  * A way to compare numbers
  * A way to conditionally execute code

```
if (x == 0) {
  printf("x is zero");
}
```

Relevant Instructions

- Comparing numbers: set-less-than (`slt`)
- Conditional execution: branch-on-equal (`beq`) and branch-on-not-equal (`bne`)
- Do we need anything else?

Relevant Instructions

- Comparing numbers: set-less-than (`slt`)
- Conditional execution: branch-on-equal (`beq`) and branch-on-not-equal (`bne`)
- Do we need anything else?
  - This is sufficient

```
if (x == 0) {
  printf("x is zero");
}
```

```
.data
x_is_zero:
  .asciiz "x is zero"

.text
  bne $t0, $zero, after_print
  li $v0, 4
  la $a0, x_is_zero
  syscall
after_print:
  li $v0, 10
  syscall
```

-Labels (the things ending with colons (:)) are symbolic addresses. The assembler will fill these in with whatever they actually point to.
-Note we inverted the condition, because we want to jump if we _don't_ meet it
-.asciiz indicates a a string which is null-terminated, like in C
-This code is in simple_branch.asm

Loops

- How might we translate the following to assembly?

```
sum = 0;
while (n != 0) {
   sum = sum + n;
   n--;
}
```

-Solution is in `add_0_to_n.asm`

Control Structure Examples

`max.asm`

`sort2.asm`

`add_0_to_n.asm`

Memory

Accessing Memory

- Two base instructions: load-word (`lw`) and store-word (`sw`)

- MIPS lacks instructions that do more with memory than access it (e.g., retrieve something from memory and add)
    - Mark of RISC architecture

Global Variable

- Typically, global variables are placed directly in memory, not registers

• Why might this be?

Global Variable

- Typically, global variables are placed directly in memory, not registers

- Why might this be?
    - Not enough registers