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)**

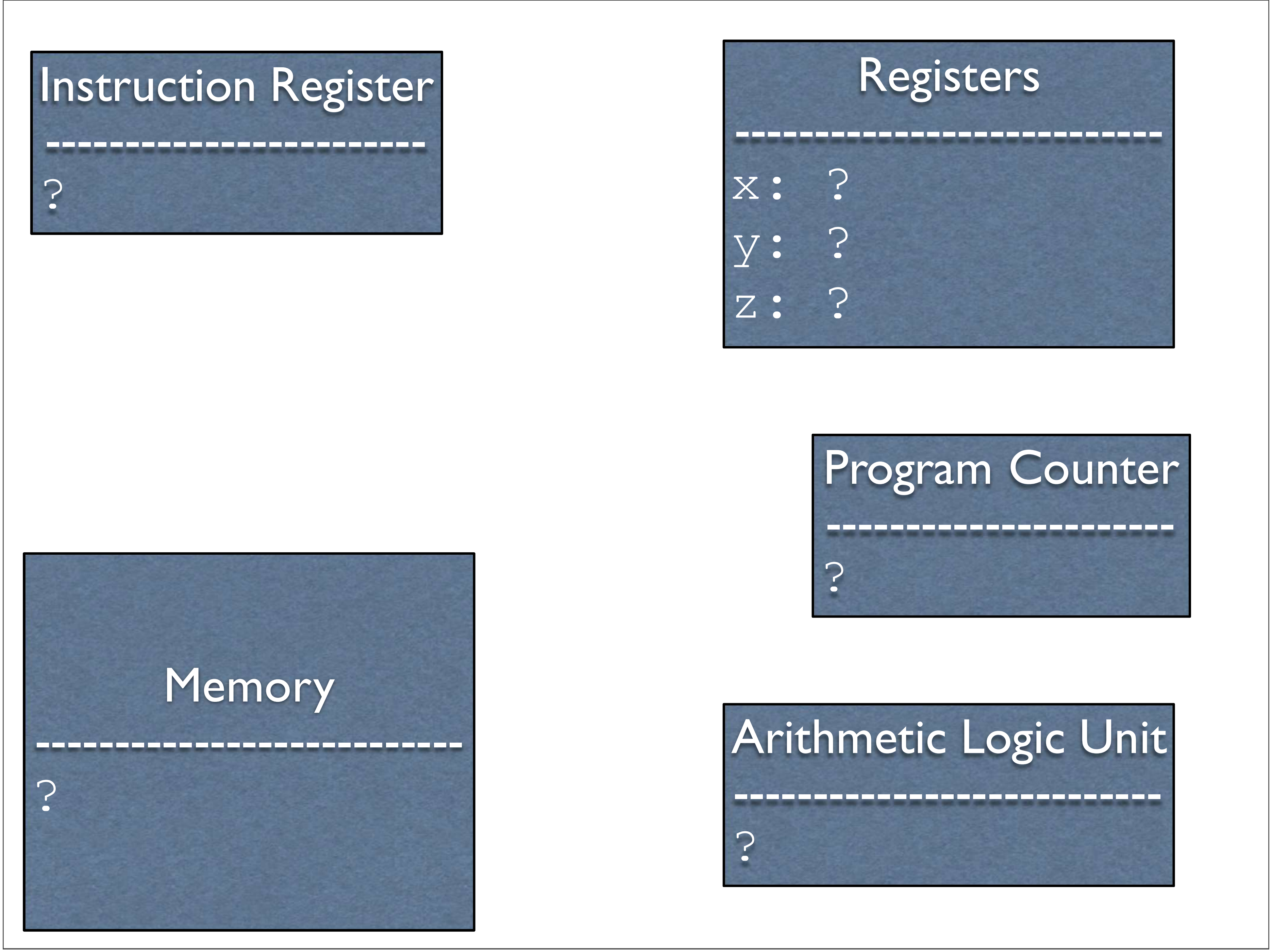
Basic Interaction

- 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

Instruction Register

?

Registers

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

Program Counter

0

Memory

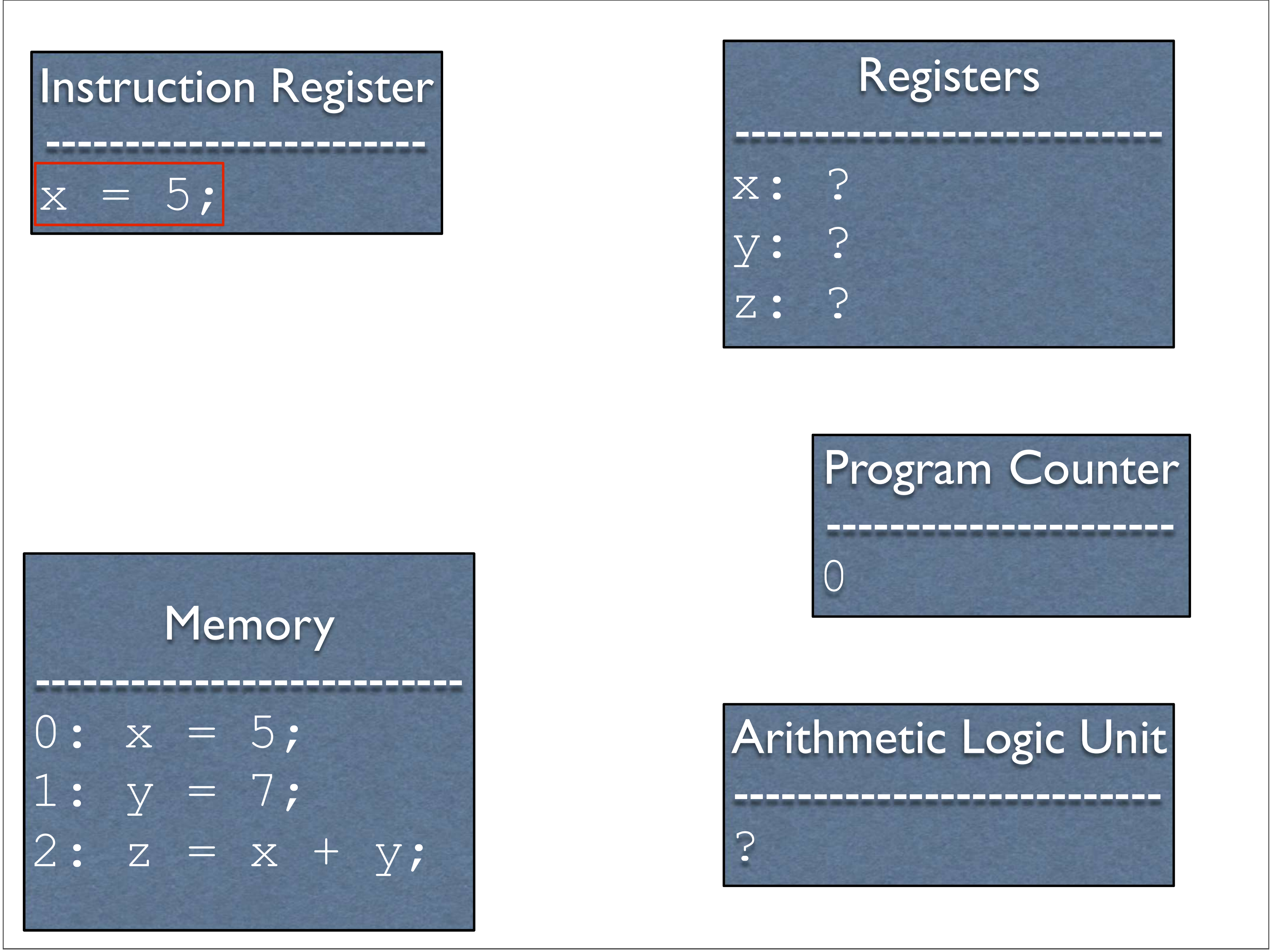
```
0: x = 5;
1: y = 7;
2: z = x + y;
```

Arithmetic Logic Unit

?

-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

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

```
?
```

-We execute the instruction, setting register x to 5

Instruction Register

```
x = 5;
```

Registers

```
x: 5
y: ?
z: ?
```

Program Counter

```
1
```

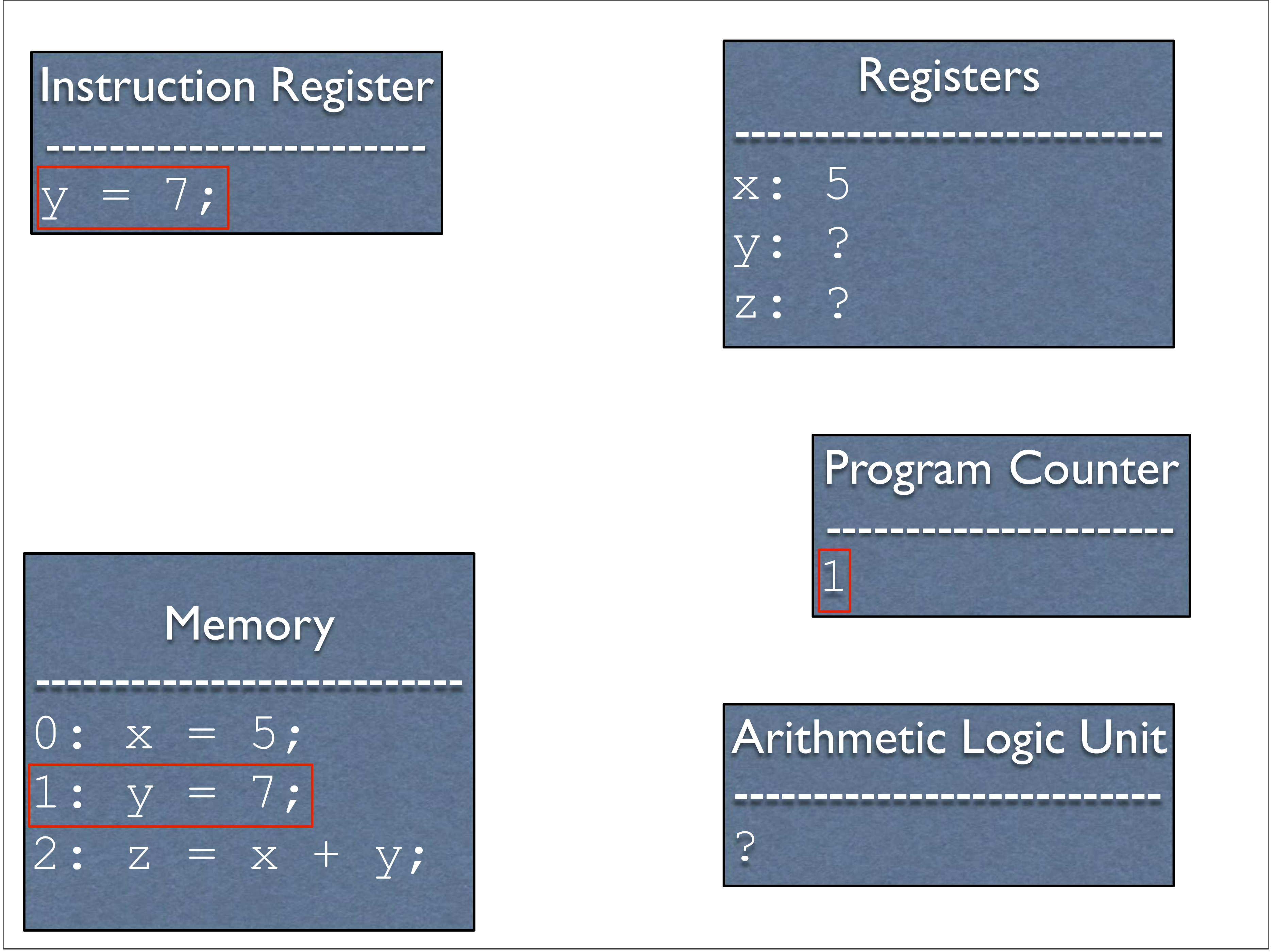
Memory

```
0: x = 5;
1: y = 7;
2: z = x + y;
```

Arithmetic Logic Unit

```
0 + 1 = 1
```

–We update the program counter



-Load in the next instruction

Instruction Register

```
y = 7;
```

Registers

```
x: 5
y: 7
z: ?
```

Program Counter

```
1
```

Memory

```
0: x = 5;
1: y = 7;
2: z = x + y;
```

Arithmetic Logic Unit

```
?
```

-We execute the instruction, setting register y to 7

Instruction Register

```
y = 7;
```

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
```

–We update the program counter

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

`?`

-Load in the next instruction

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

`5 + 7 = 12`

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

Instruction Register

$\boxed{z = x + y;}$

Registers

---

x: 5
y: 7
z: $\boxed{12}$

Program Counter

2

Memory

```
0: x = 5;
1: y = 7;
2: z = x + y;
```

Arithmetic Logic Unit

---

5 + 7 = $\boxed{12}$

–The ALU sets the result

ARM

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

Why ARM?

- Incredibly popular in embedded devices
- Much simpler than Intel processors

-Embedded devices include things like phones and microwaves.
-Your computer may have an ARM processor in it even though the “main” processor is from Intel

Code on ARM

Original

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

ARM

```
mov r0, #5
mov r1, #7
add r2, r0, r1
```

Code on ARM

Original

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

ARM

```
mov r0, #5
mov r1, #7
add r2, r0, r1
```

move: put the given value into a register

r0: **r**egister **0**

Code on ARM

Original

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

ARM

```
mov r0, #5
mov r1, #7
add r2, r0, r1
```

move: put the given value into a register

r1: **r**egister **1**

Code on ARM

Original	ARM

`x = 5;`
`y = 7;`
`z = x + y;`
	`mov r0, #5`
	`mov r1, #7`
	`add r2, r0, r1`

	add: add the rightmost registers, putting the result in the first register
	r2: **r**egister **2**

Available Registers

- 17 registers in all
  - 16 "general-purpose"
  - 1 "special-purpose"
- For the moment, we will only consider registers `r0` - `r12`

-General purpose: can put values in them and take values out as I please
-Special purpose: holding certain flags. Can manipulate this, but not in the same way as a general-purpose register

Assembly

- The code that you see below is *ARM 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*)

```
mov r0, #5
mov r1, #7
add r2, r0, r1
```

-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

```
mov r0, #5
mov r1, #7
add r2, r0, r1
```

Assembler
(analogous to a compiler)

Machine Code

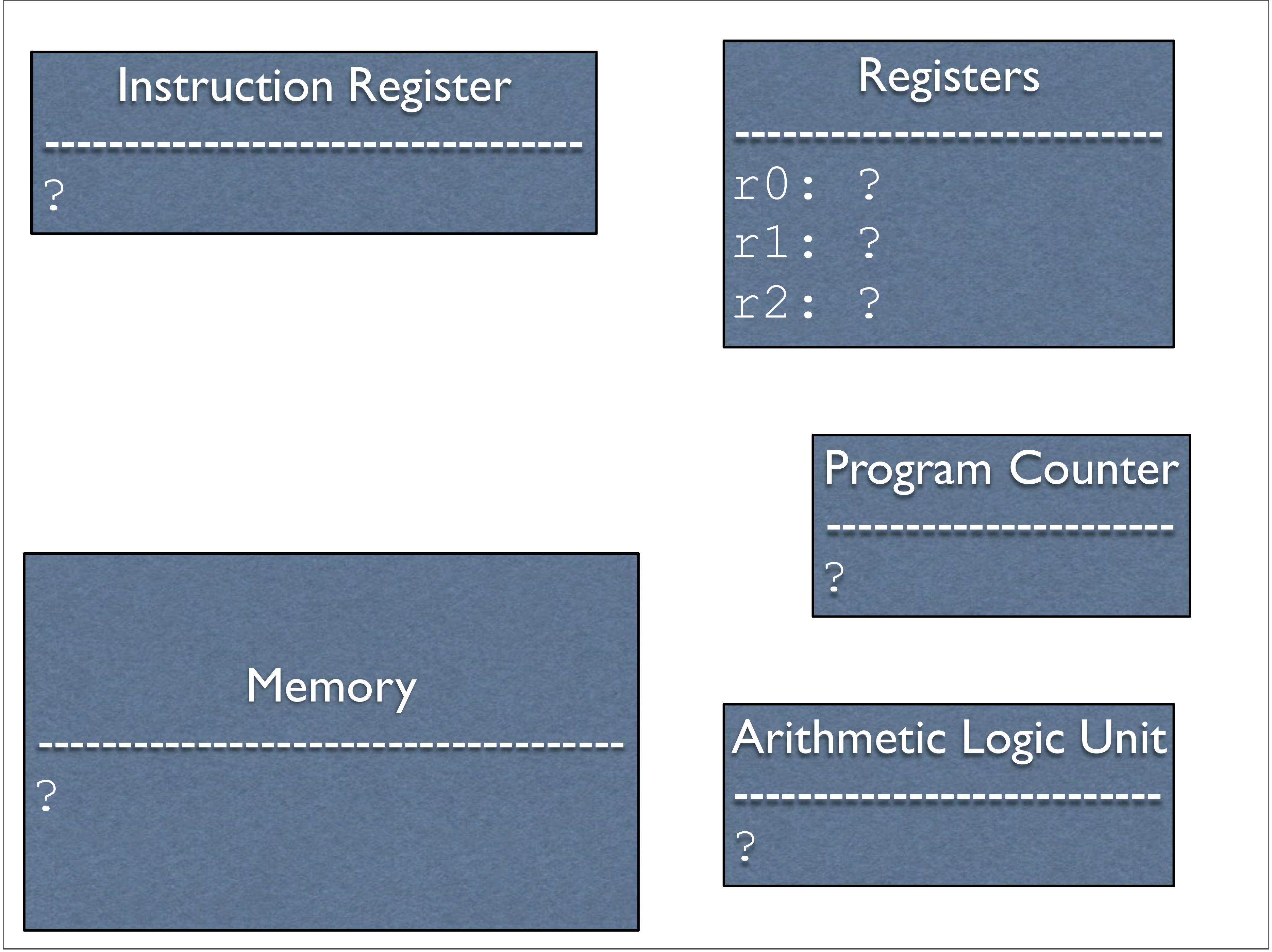
```
001101....
```

Machine Code

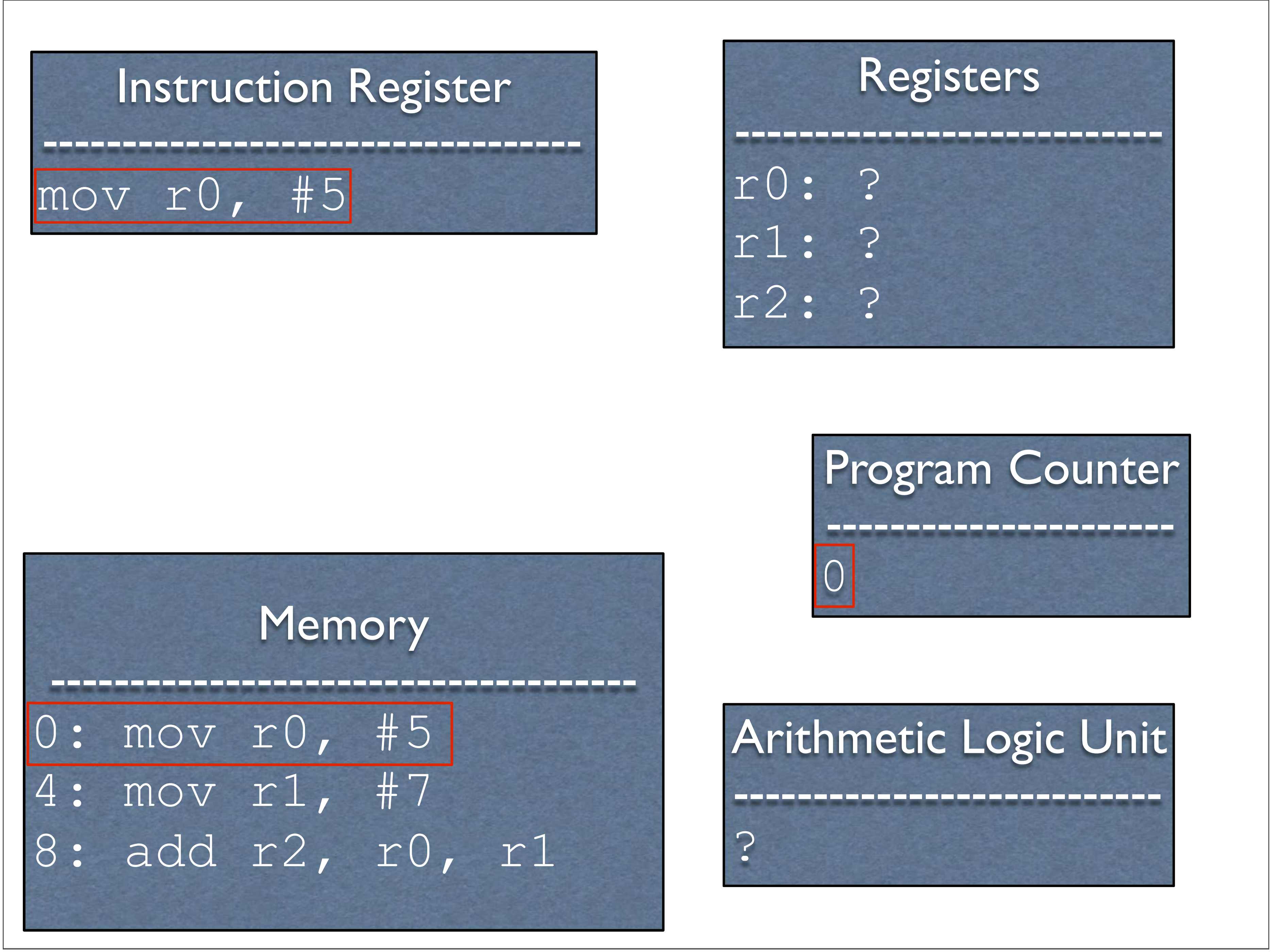
- This is what the process actually executes and accepts as input
- Each instruction is represented with 32 bits

```
add r2, r0, r1
```

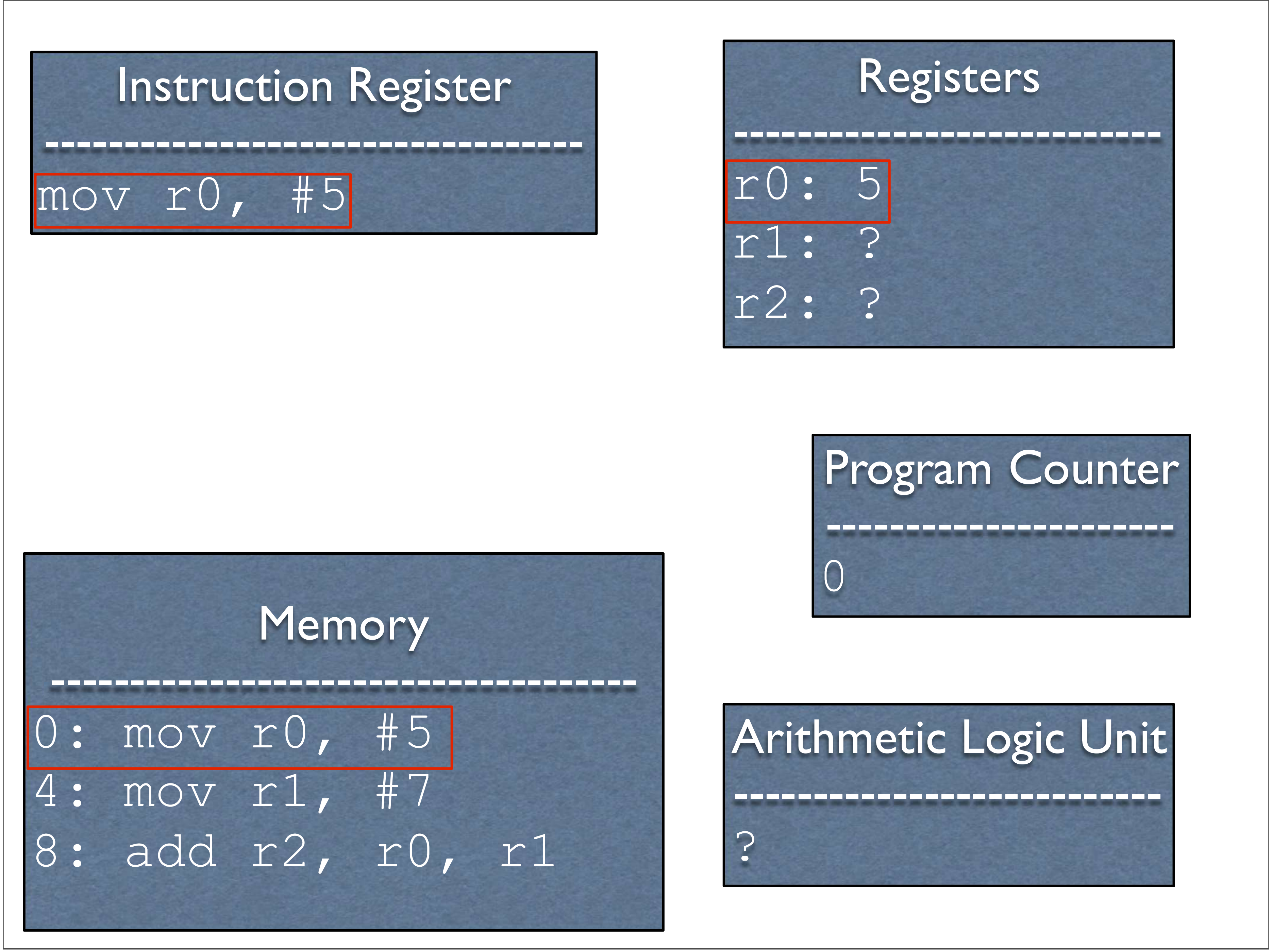
-Converting to machine code is mostly one-to-one: just put the right bits in the right places
-There are some exceptions where we have to be a bit smarter, but not much smarter. Assemblers are nowhere near as complex as compilers.



–All the hardware, before initialization



–We load instruction 0 into the instruction register



–We execute the instruction, setting register r0 to 5

Instruction Register

```
mov r0, #5
```

Registers

```
r0: 5
r1: ?
r2: ?
```

Program Counter

```
4
```

Memory

```
0: mov r0, #5
4: mov r1, #7
8: add r2, r0, r1
```

Arithmetic Logic Unit

```
0 + 4 = 4
```

-We update the program counter

-Note that we add 4 instead of one, as instructions are four bytes long

Instruction Register

```
mov r1, #7
```

Registers

```
r0: 5
r1: ?
r2: ?
```

Program Counter

```
4
```

Memory

```
0: mov r0, #5
4: mov r1, #7
8: add r2, r0, r1
```

Arithmetic Logic Unit

```
?
```

–Load in the next instruction

Instruction Register

```
mov r1, #7
```

Registers

```
r0: 5
r1: 7
r2: ?
```

Program Counter

```
4
```

Memory

```
0: mov r0, #5
4: mov r1, #7
8: add r2, r0, r1
```

Arithmetic Logic Unit

```
?
```

–We execute the instruction, setting register r1 to 7

Instruction Register

```
mov r1, #7
```

Registers

```
r0: 5
r1: 7
r2: ?
```

Program Counter

```
8
```

Memory

```
0: mov r0, #5
4: mov r1, #7
8: add r2, r0, r1
```

Arithmetic Logic Unit

```
4 + 4 = 8
```

–We update the program counter

Instruction Register

```
add r2, r0, r1
```

Registers

```
r0: 5
r1: 7
r2: ?
```

Program Counter

```
8
```

Memory

```
0: mov r0, #5
4: mov r1, #7
8: add r2, r0, r1
```

Arithmetic Logic Unit

```
?
```

–Load in the next instruction

Instruction Register

```
add r2, r0, r1
```

Registers

```
r0: 5
r1: 7
r2: ?
```

Program Counter

```
8
```

Memory

```
0: mov r0, #5
4: mov r1, #7
8: add r2, r0, r1
```

Arithmetic Logic Unit

```
5 + 7 = 12
```

-Execute it, consulting the registers to get the values of r0 and r1
-This consults the ALU

Instruction Register

`add r2, r0, r1`

Registers

```
r0: 5
r1: 7
r2: 12
```

Program Counter

`8`

Memory

```
0: mov r0, #5
4: mov r1, #7
8: add r2, r0, r1
```

Arithmetic Logic Unit

`5 + 7 = 12`

–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 an ARM simulator, ARMSim#
- We cannot directly access system libraries (they aren't even in the same machine language)
- How might we print something?

ARMSim# Routines

- ARM features a `swi` instruction, which triggers a *software interrupt*
- Outside of a simulator, these pause the program and tell the OS to check something
- Inside the simulator, it tells the **simulator** to check something

`swi`

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

`swi`

- So we have the OS/simulator's attention. But how does it know what we want?
  - `swi` operand: integer saying what to do
  - The OS/simulator can also read the registers to get extra information as needed

–"Integer saying what to do": e.g., we agree that "5" means "print something"
–With reading the registers, these could include exactly what to print

(Finally) Printing an Integer

- For ARMSim#, the integer that says "print an integer" is `0x6B`
- Register `r1` holds the integer to print
- Register `r0` holds where to print it; `1` means "print to standard output (screen)"

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

Augmenting with Printing

```
mov r0, #5
mov r1, #7
add r2, r0, r1

mov r1, r2  ; r1: integer to print
mov r0, #1  ; r0: where to print it
swi 0x6B    ; 0x6B: print integer
```

Exiting

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

Exiting

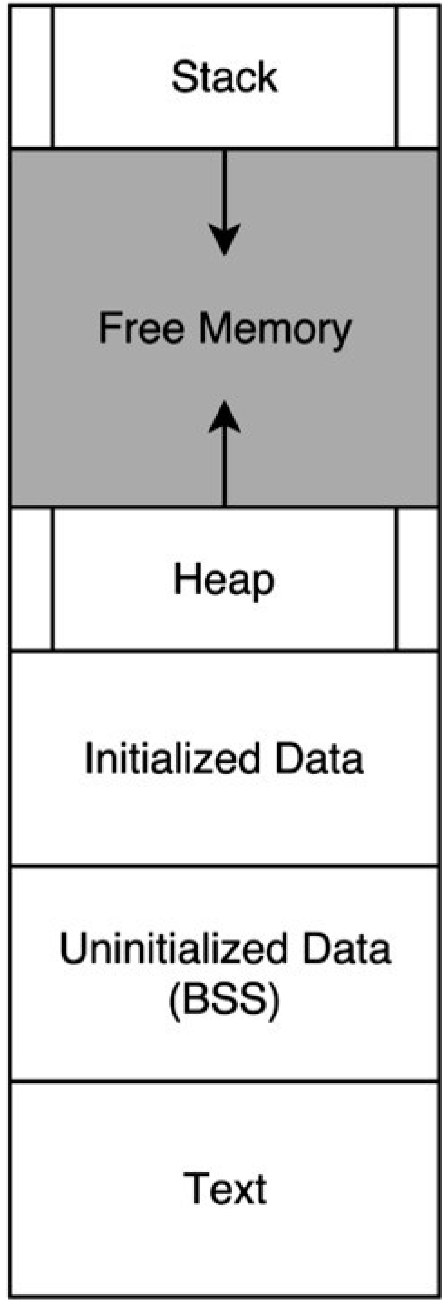
- If you are using ARMSim#, then you need to say when you are done as well
- How might this be done?
    - `swi` with a particular operand (specifically `0x11`)

Augmenting with Exiting

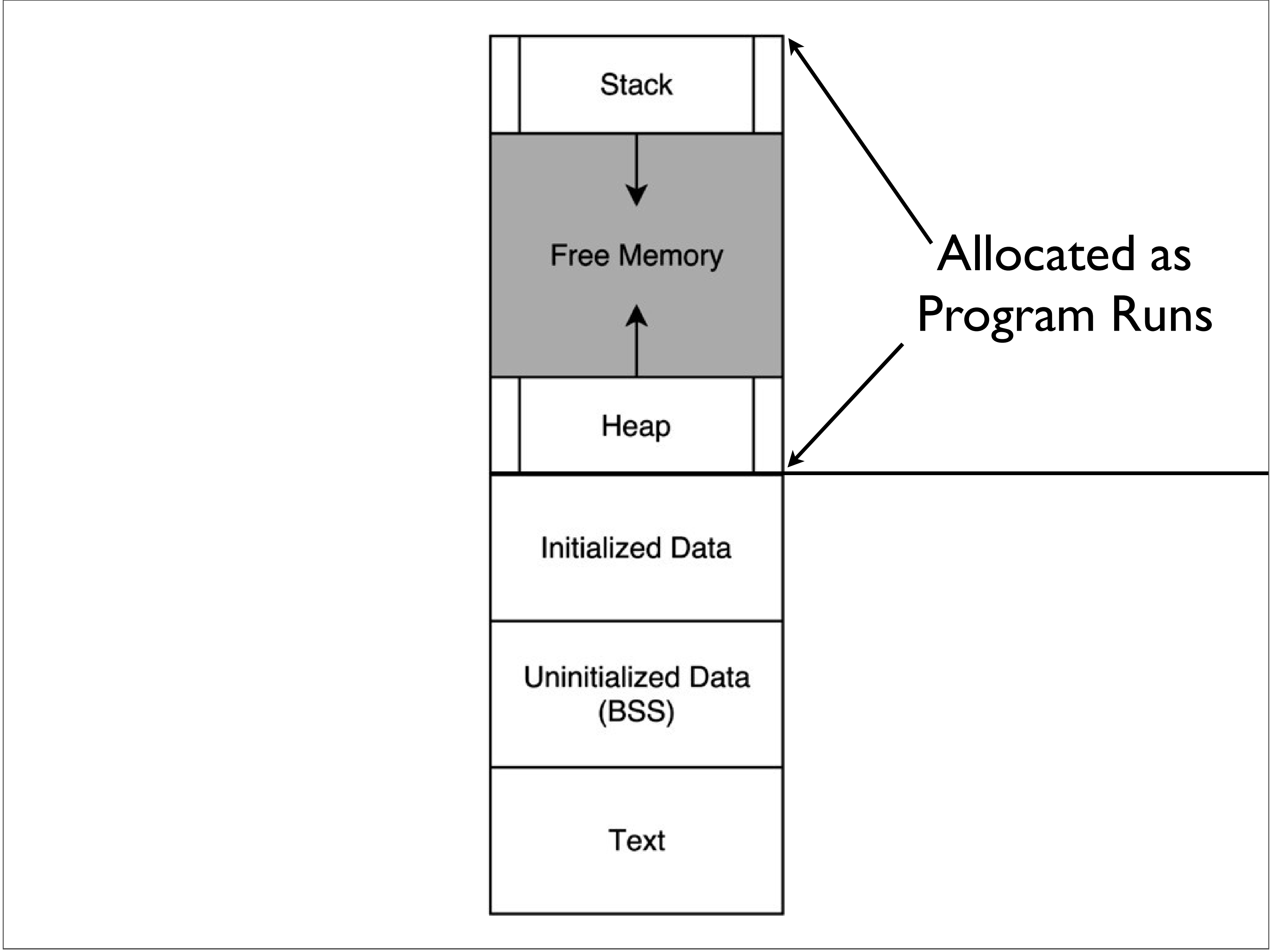
```
mov  r0,  #5
mov  r1,  #7
add  r2,  r0,  r1

mov  r1,  r2  ;  r1: integer to print
mov  r0,  #1  ;  r0: where to print it
swi  0x6B     ;  0x6B: print integer

swi  0x11     ;  0x11:  exit  program
```

Making it a Full Program

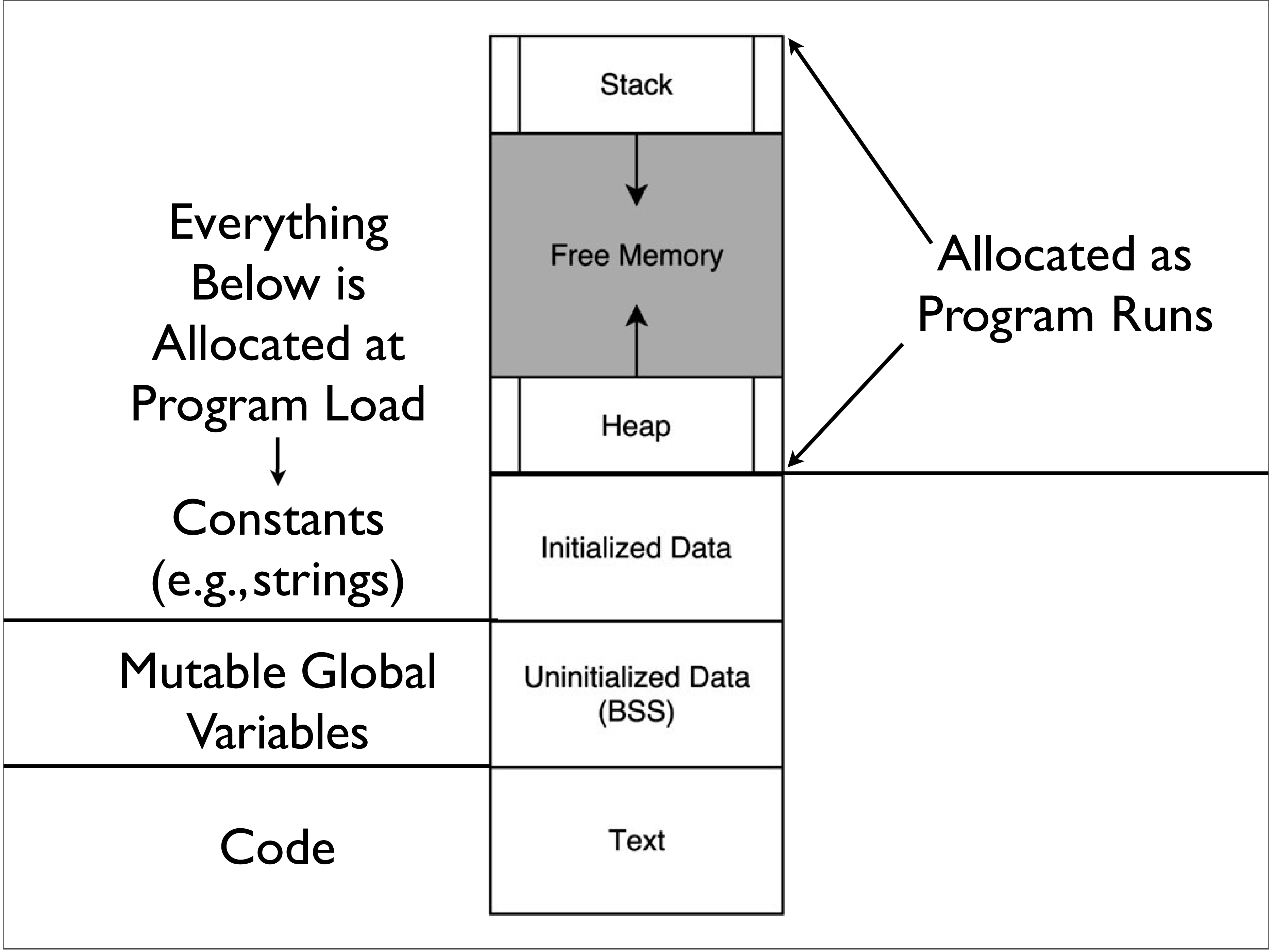
- Everything is just a bunch of bits
- We need to tell the assembler which bits should be placed where in memory



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



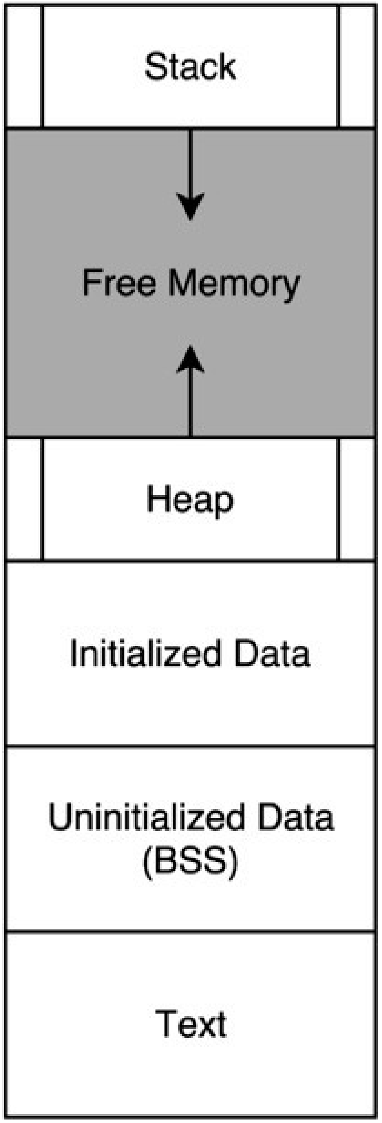
- Image source: https://en.wikipedia.org/wiki/Data_segment
- You've seen these two before
- What might the rest be?



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

Marking Code

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

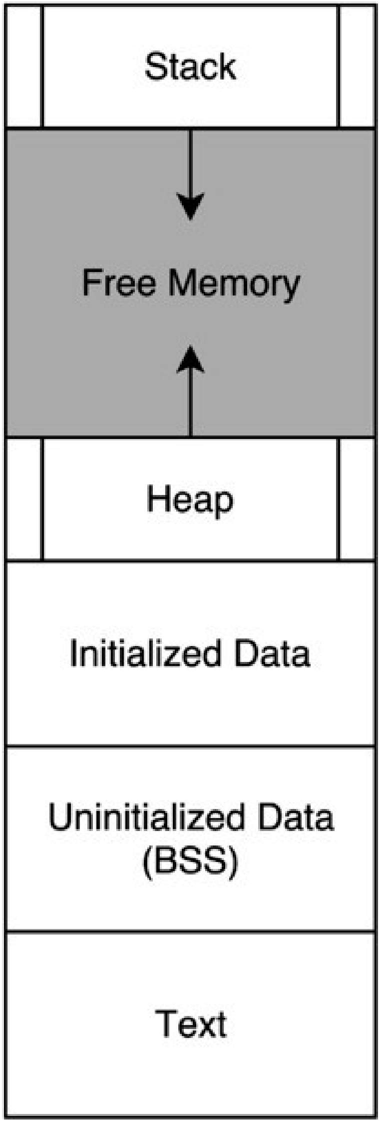



```
.text
mov r0, #5
mov r1, #7
add r2, r0, r1

mov r1, r2
mov r0, #1
swi 0x6B

swi 0x11
```

–Directives tell the assembler to do something

Marking Code

Use a `.data` *directive* to specify data section




```
     .data
string1:
     .asciz "hello"
string2:
     .asciz "bye"
```

–Directives tell the assembler to do something

ARMSim# Demo:
`hello.s`

ARMSim# Demo:
`arithmetic_ops.s`

ARMSim# Demo:

`read_and_print_int.s`