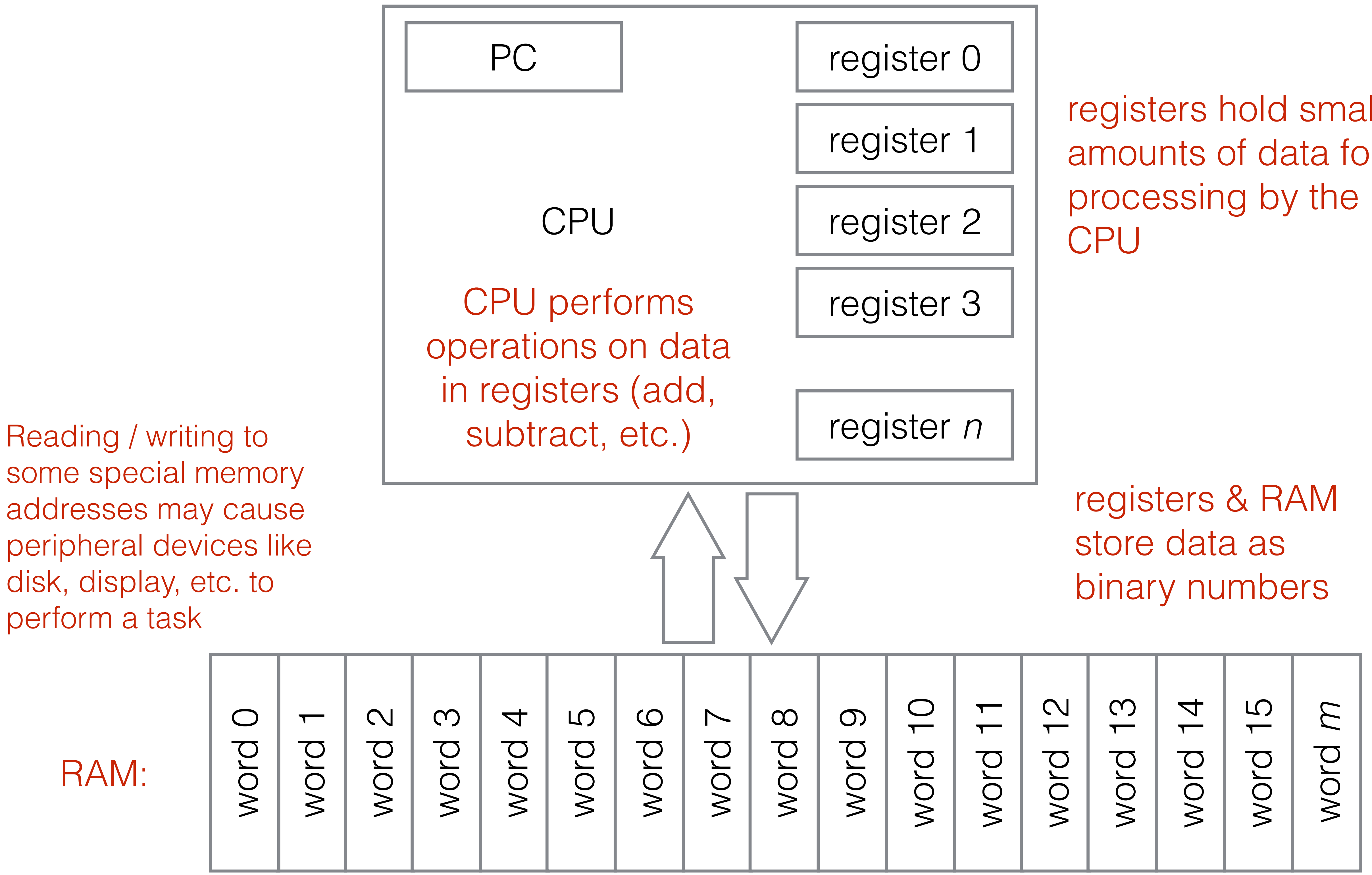
Computer Architecture

02-201 / 02-601

The Conceptual Architecture of a Computer



Binary Representation

Base 10 (decimal) notation:

$$4\ 2\ 5\ 6$$

$$\begin{array}{rl} & 6\text{x}10^0 \\ & 5\text{x}10^1 \\ & 2\text{x}10^2 \\ + & 4\text{x}10^3 \\ \hline & 4\ 2\ 5\ 6 \end{array}$$

Computers store the numbers in binary because it has transistors that can encode 0 and 1 efficient

Each 0 and 1 is a *bit.*

Built-in number types each have a maximum number of bits.

Base 2 (binary) notation:

$$1\ 0\ 0\ 0\ 0\ 1\ 0\ 1\ 0\ 0\ 0\ 0\ 0$$

$$\begin{array}{rl} & 0\text{x}2^0 \\ & 0\text{x}2^1 \\ & 0\text{x}2^2 \\ & 0\text{x}2^3 \\ & 0\text{x}2^4 \\ & 1\text{x}2^5 \\ & 0\text{x}2^6 \\ & 1\text{x}2^7 \\ & 0\text{x}2^8 \\ & 0\text{x}2^9 \\ & 0\text{x}2^{10} \\ & 0\text{x}2^{11} \\ + & 1\text{x}2^{12} \\ \hline \end{array}$$

$$4\ 2\ 5\ 6 = \ 1\ 0\ 0\ 0\ 0\ 1\ 0\ 1\ 0\ 0\ 0\ 0\ 0$$

Hexadecimal Representation

Decimal isn't good for computers because they work with bits.
But writing everything in binary would be tedious.
Hence, we often use base 16, aka "hexadecimal":

Base 10 (decimal) notation:

$$
\begin{array}{rl}
 & 4\ 2\ 5\ 6 \\
 & 6 \times 10^0 \\
 & 5 \times 10^1 \\
 & 2 \times 10^2 \\
+ & 4 \times 10^3 \\
\hline
 & 4\ 2\ 5\ 6
\end{array}
$$

Base 16 (hexadecimal) notation:

$$
\begin{array}{rl}
 & 1\ 0\ A\ 0 \\
 & 0 \times 16^0 \\
 & A \times 16^1 \\
 & 0 \times 16^2 \\
+ & 1 \times 16^3 \\
\hline
\end{array}
$$

$$1 \times 4096 + 10 \times 16 = 4256$$

Need 16 different digits, so use 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

A=10, B=11, C=12, D=13, E=14, F=15

Add CPU instruction

`ADD Rd, Rs, Rt` — Set register Rd to Rs + Rt

An instruction is encoded as a sequence of bits:

Instruction (aka opcode)				Rd				Rs				Rt			
0	0	0	1	0	0	1	0	0	0	1	0	1	0	0	0
1 = ADD				2 = R2				2 = R2				8 = R8			

Written as a hexadecimal number: 1228_{16}

Subtract CPU instruction

`SUB Rd, Rs, Rt`

Set register Rd to Rs - Rt

The SUB instruction is the same format as ADD, but with a different opcode:

Instruction (aka opcode)				Rd				Rs				Rt			
0	0	1	0	0	0	1	0	0	0	1	0	1	0	0	0
`2 = SUB`				`2 = R2`				`2 = R2`				`8 = R8`			

Written as a hexadecimal number: 2228_{16}

Load "Immediate" Instruction

`LIMM Rd, value`

Set register Rd to value

For LIMM, the last 8 bits give the value to copy into Rd:

Instruction (aka opcode)				Rd				immediate							
1	0	0	0	0	1	1	0	1	1	0	0	1	0	0	0
7 = LIMM				6 = R6				200 = value = $C8_{16}$							

Written as a hexadecimal number: $76C8_{16}$

Load Instruction

`LOAD Rd, addr`   Set register Rd to ram[addr]

For LOAD, the last 8 bits give the address of the memory cell to copy into Rd:

Instruction (aka opcode)				Rd				address							
1	0	0	0	0	1	0	0	1	0	0	0	1	0	0	0

`8 = LOAD`   `4 = R4`   `136 = addr`

$= 88_{16}$

Written as a hexadecimal number: 8488_{16}

Store Instruction

`STORE Rd, addr`   Set register ram[addr] to Rd

For LOAD, the last 8 bits give the address of the memory cell to copy into Rd:

Instruction (aka opcode)				Rd				address							
1	0	0	1	0	1	0	0	1	0	0	0	1	0	0	0

`9 = STORE`   `4 = R4`   `136 = addr`

$= 88_{16}$

Written as a hexadecimal number: 9488_{16}

An Example Program

```
LIMM R1, 64          // R1 = 200
LIMM R2, 1E          // R2 = 30
ADD R2, R1, R2       // R2 = R1 + R2
STORE R2, 46         // ram[70] = R2
```

What does this program do?

An Example Program

```
LIMM R1, 64         // R1 = 200
LIMM R2, 1E         // R2 = 30
ADD R2, R1, R2      // R2 = R1 + R2
STORE R2, 46        // ram[70] = R2
```

What does this program do?

Stores 200 + 30 into memory location 70

Similar to the following Go program:

```go
var r1 int = 200
var r2 int = 30
r2 = r1 + r2
var ram70 int = r2
```

Go manages the registers and memory locations for you.

It may keep a variable in a register, memory, or both.

Where is the program stored?

The program is just a sequence of integers that encode for instructions.

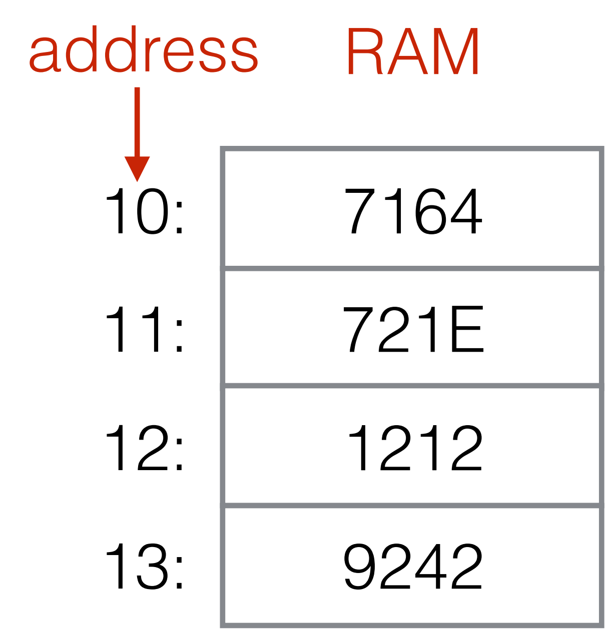
```
LIMM R1, 64       // R1 = 200         ; 7164
LIMM R2, 1E       // R2 = 30          ; 721E
ADD R2, R1, R2    // R2 = R1 + R2     ; 1212
STORE R2, 46      // ram[70] = R2     ; 9246
```

Where is the program stored?

The program is just a sequence of integers that encode for instructions.

```
LIMM R1, 64      // R1 = 200        ; 7164
LIMM R2, 1E      // R2 = 30         ; 721E
ADD R2, R1, R2   // R2 = R1 + R2    ; 1212
STORE R2, 46     // ram[70] = R2    ; 9246
```

These integers are stored in the same RAM used for variables:



Where is the program stored?

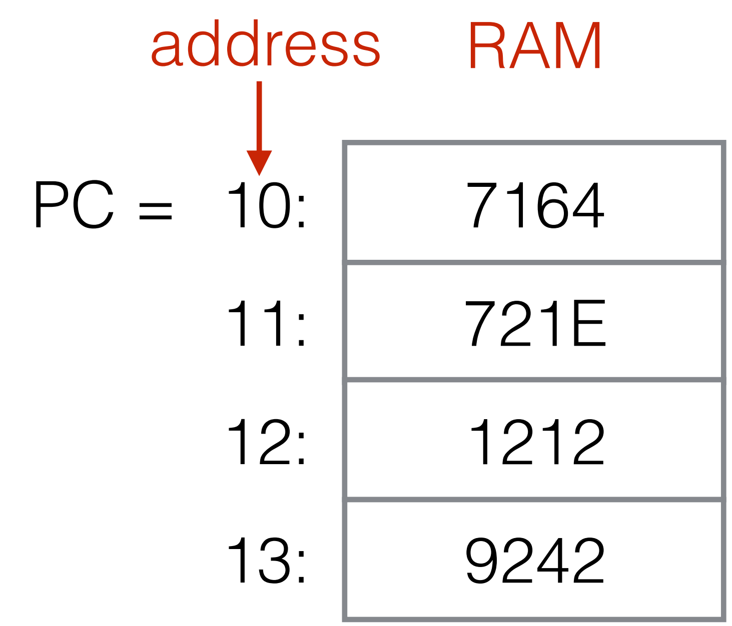
The program is just a sequence of integers that encode for instructions.

```
LIMM R1, 64       // R1 = 200          ; 7164
LIMM R2, 1E       // R2 = 30           ; 721E
ADD R2, R1, R2    // R2 = R1 + R2      ; 1212
STORE R2, 46      // ram[70] = R2      ; 9246
```

These integers are stored in the same RAM used for variables:

The CPU has a special register called PC ("program counter") that contains the address of the current instruction.

After each instruction, the PC is incremented by 1.



Where is the program stored?

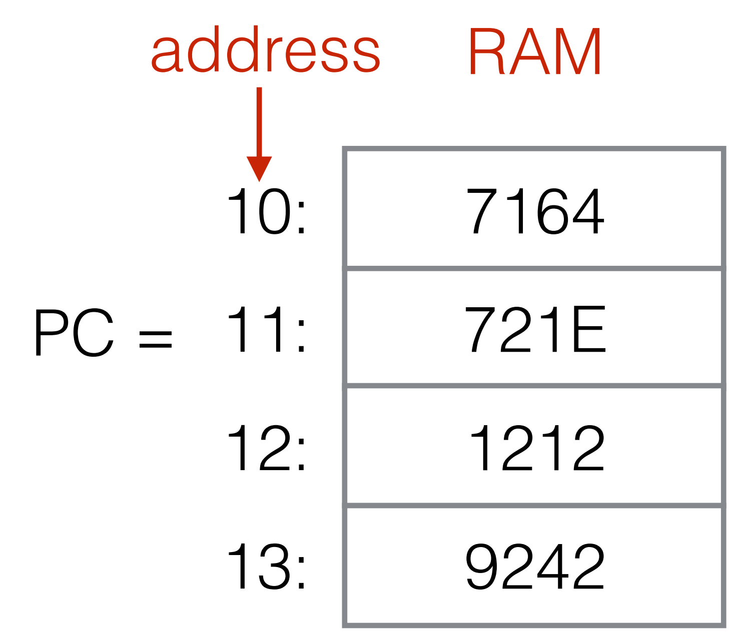
The program is just a sequence of integers that encode for instructions.

```
LIMM R1, 64       // R1 = 200         ; 7164
LIMM R2, 1E       // R2 = 30          ; 721E
ADD R2, R1, R2    // R2 = R1 + R2     ; 1212
STORE R2, 46      // ram[70] = R2     ; 9246
```

These integers are stored in the same RAM used for variables:

The CPU has a special register called PC ("program counter") that contains the address of the current instruction.

After each instruction, the PC is incremented by 1.



Where is the program stored?

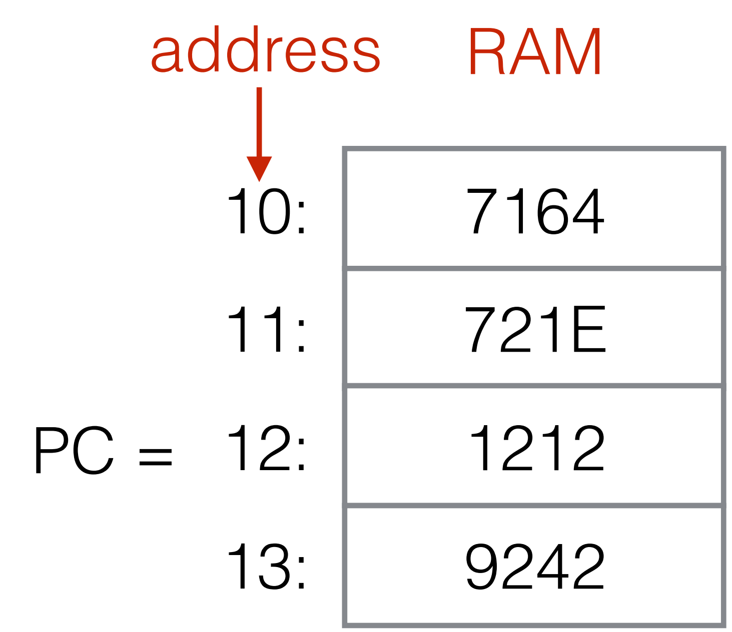
The program is just a sequence of integers that encode for instructions.

```
LIMM R1, 64        // R1 = 200          ; 7164
LIMM R2, 1E        // R2 = 30           ; 721E
ADD R2, R1, R2     // R2 = R1 + R2      ; 1212
STORE R2, 46       // ram[70] = R2      ; 9246
```

These integers are stored in the same RAM used for variables:

The CPU has a special register called PC ("program counter") that contains the address of the current instruction.

After each instruction, the PC is incremented by 1.



Where is the program stored?

The program is just a sequence of integers that encode for instructions.

```
LIMM R1, 64       // R1 = 200          ; 7164
LIMM R2, 1E       // R2 = 30           ; 721E
ADD R2, R1, R2    // R2 = R1 + R2      ; 1212
STORE R2, 46      // ram[70] = R2      ; 9246
```

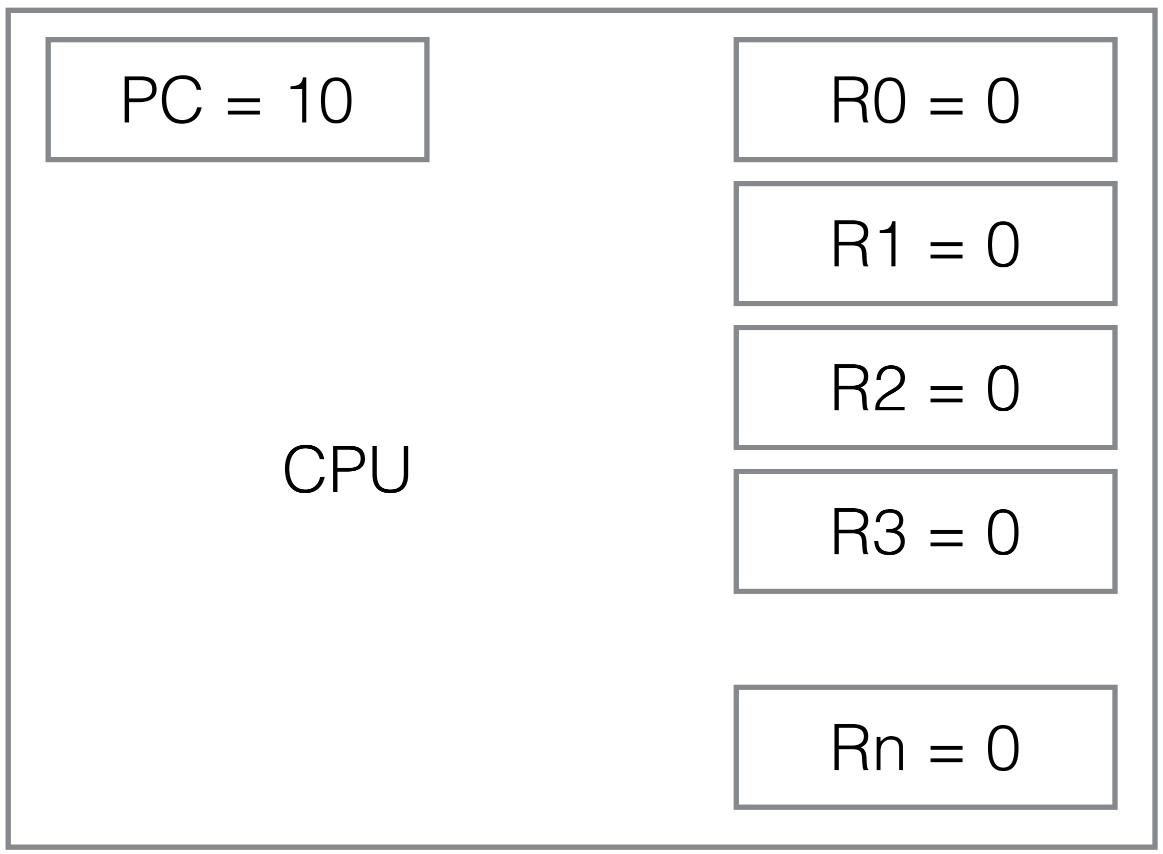
These integers are stored in the same RAM used for variables:

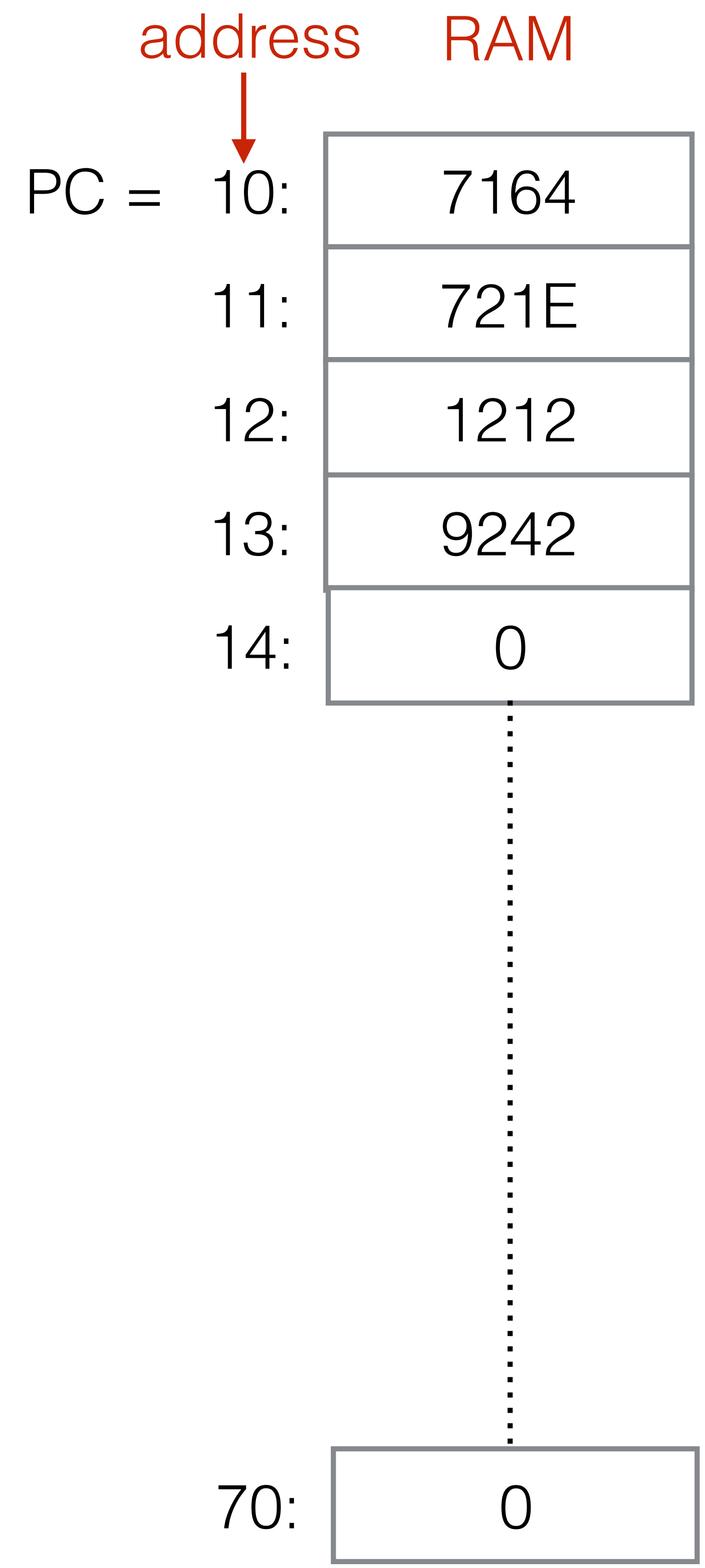
The CPU has a special register called PC ("program counter") that contains the address of the current instruction.

After each instruction, the PC is incremented by 1.

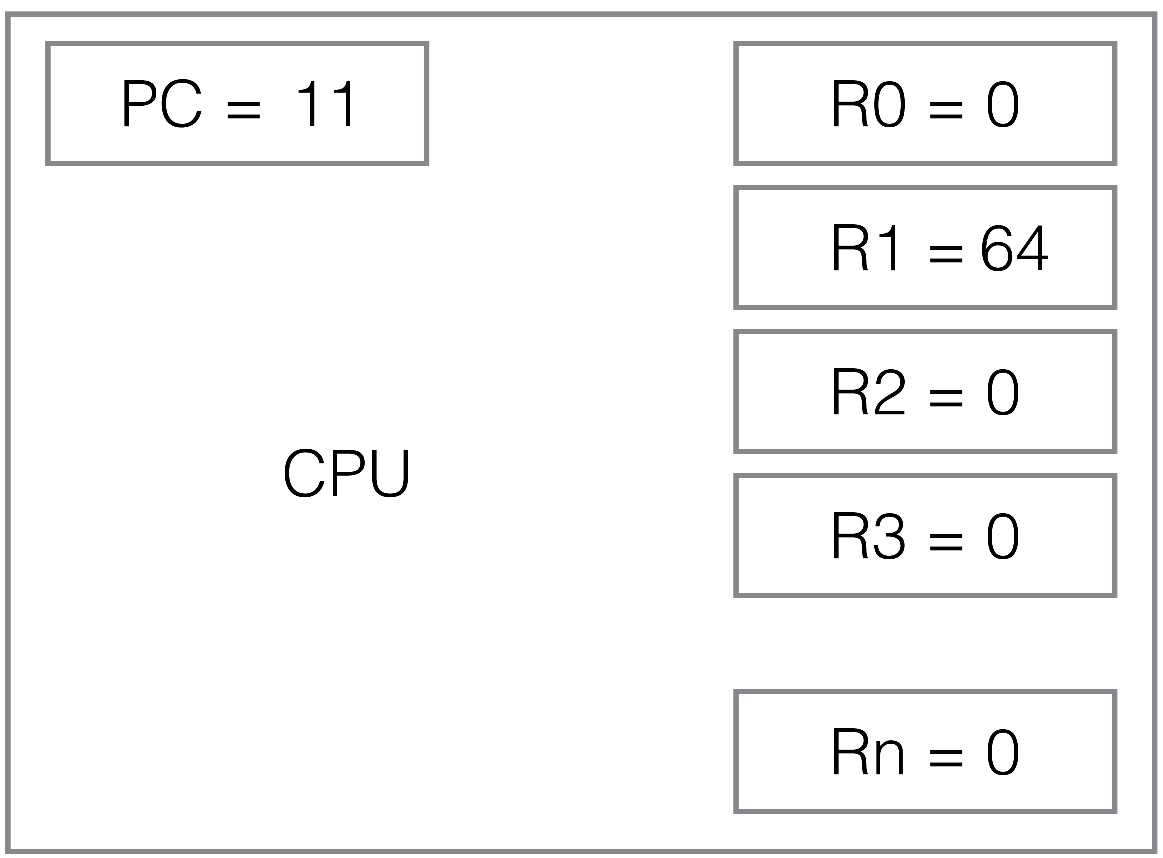
	address	RAM
	10:	7164
	11:	721E
	12:	1212
PC =	13:	9242

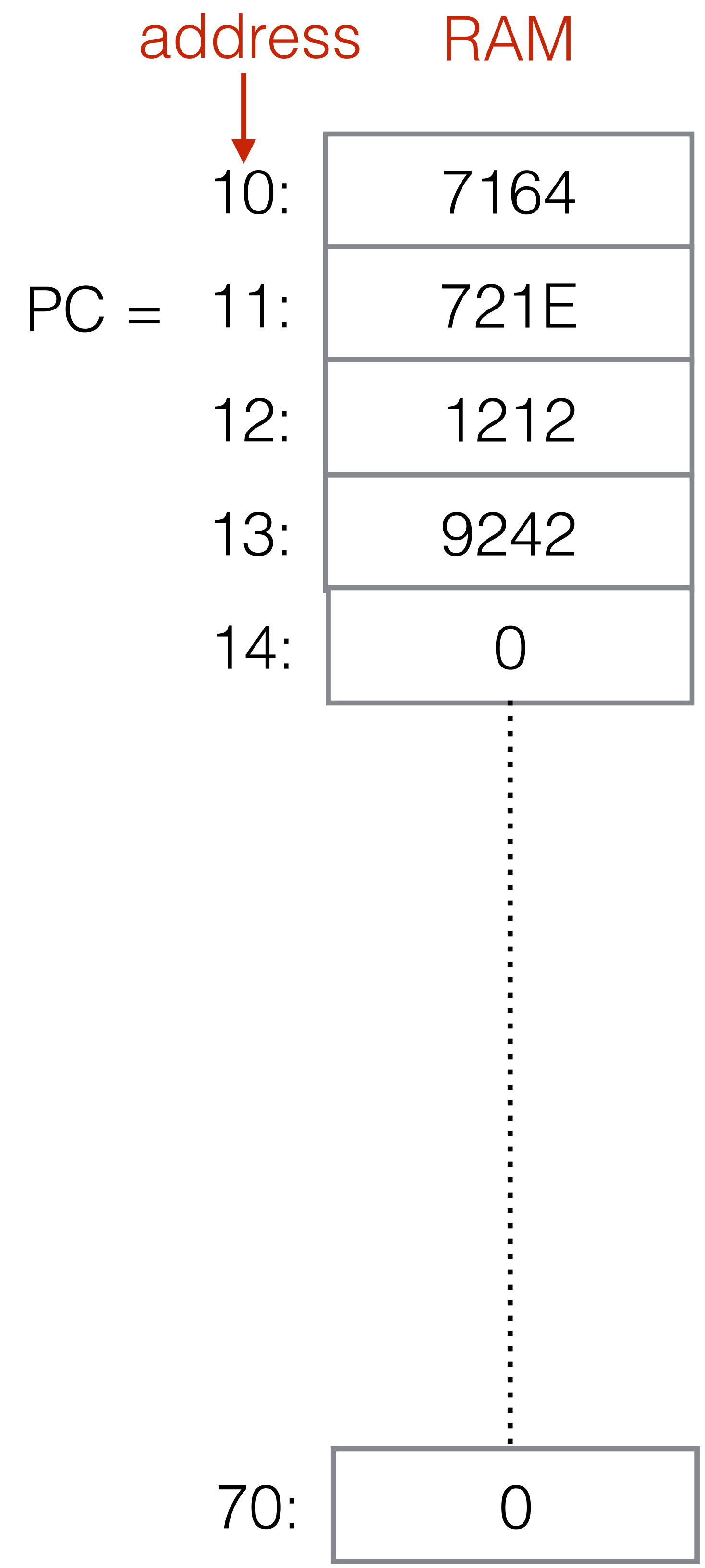
```
LIMM R1, 64        // R1 = 200        ; 7164
LIMM R2, 1E        // R2 = 30         ; 721E
ADD R2, R1, R2     // R2 = R1 + R2    ; 1212
STORE R2, 46       // ram[70] = R2    ; 9246
```



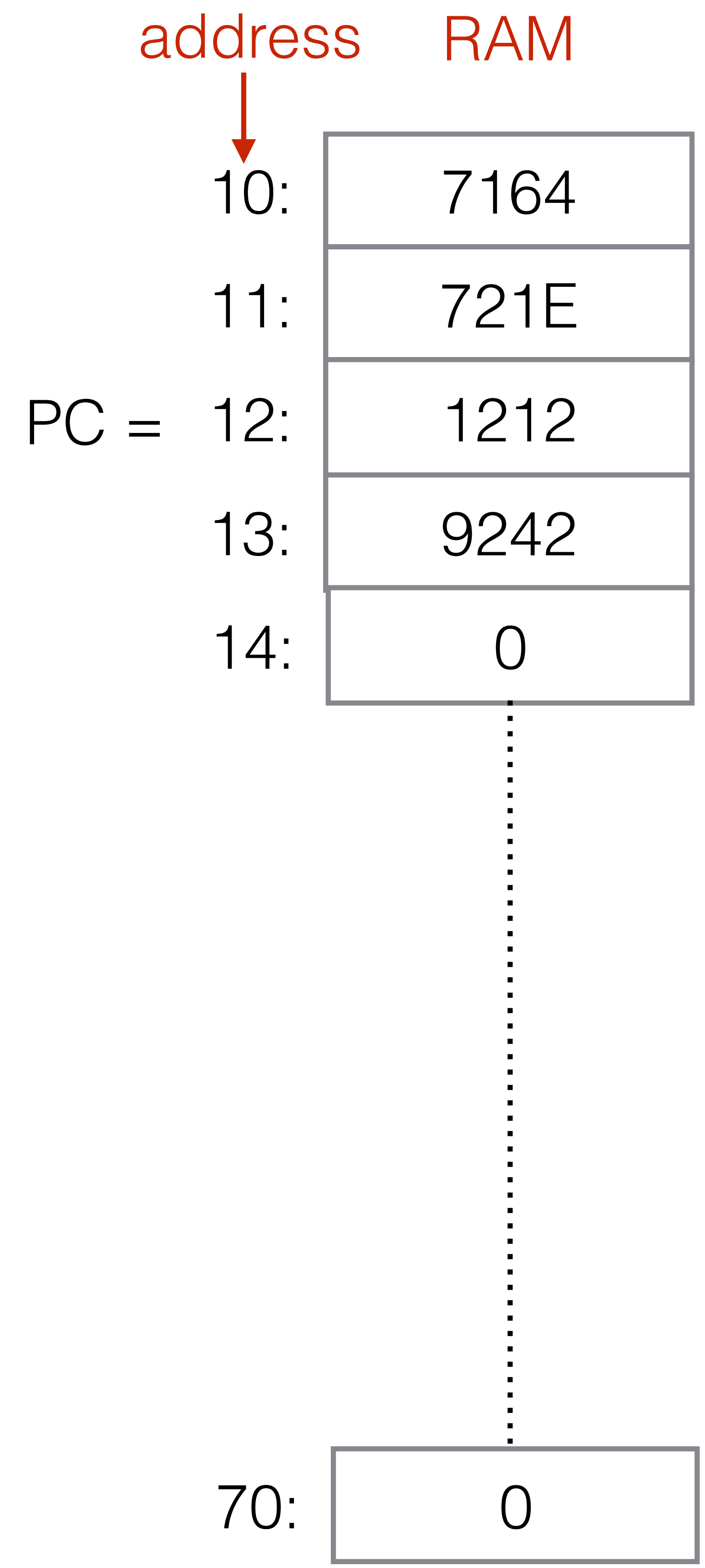


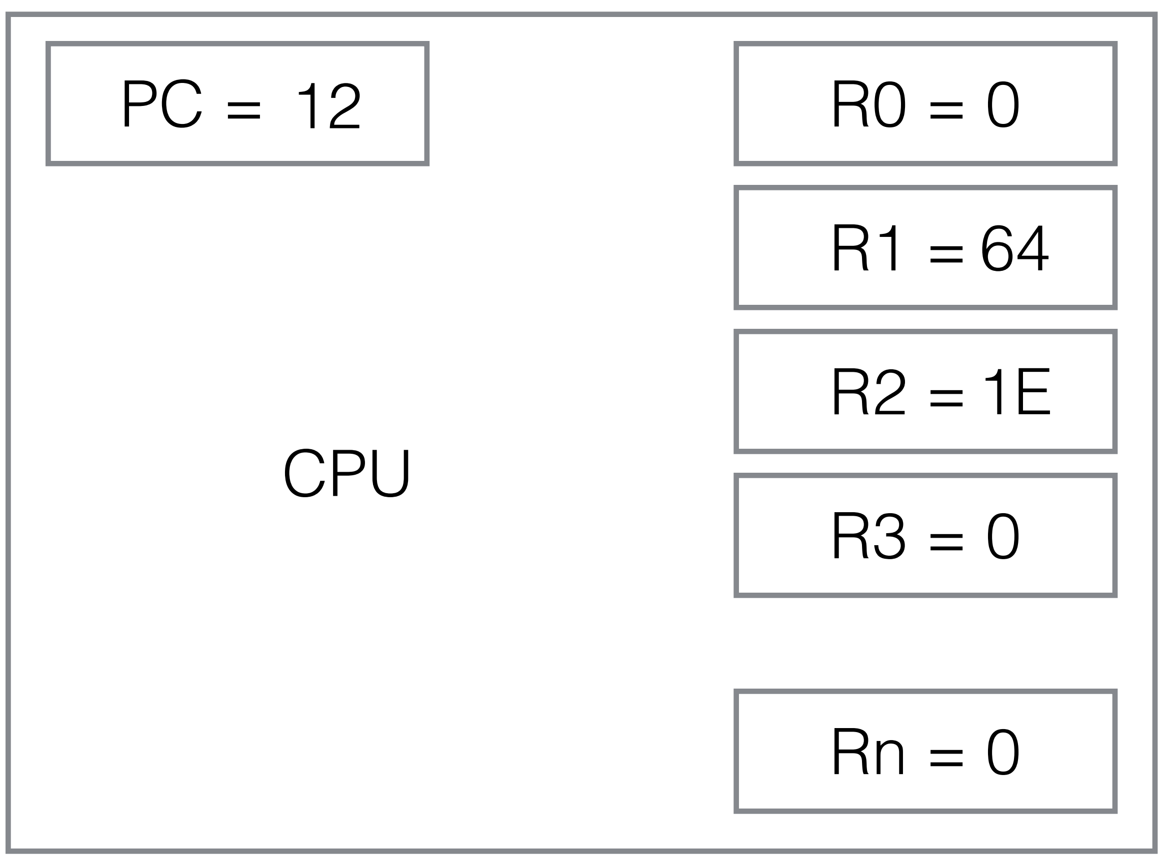
```
LIMM R1, 64        // R1 = 200        ; 7164
LIMM R2, 1E        // R2 = 30         ; 721E
ADD R2, R1, R2     // R2 = R1 + R2    ; 1212
STORE R2, 46       // ram[70] = R2    ; 9246
```



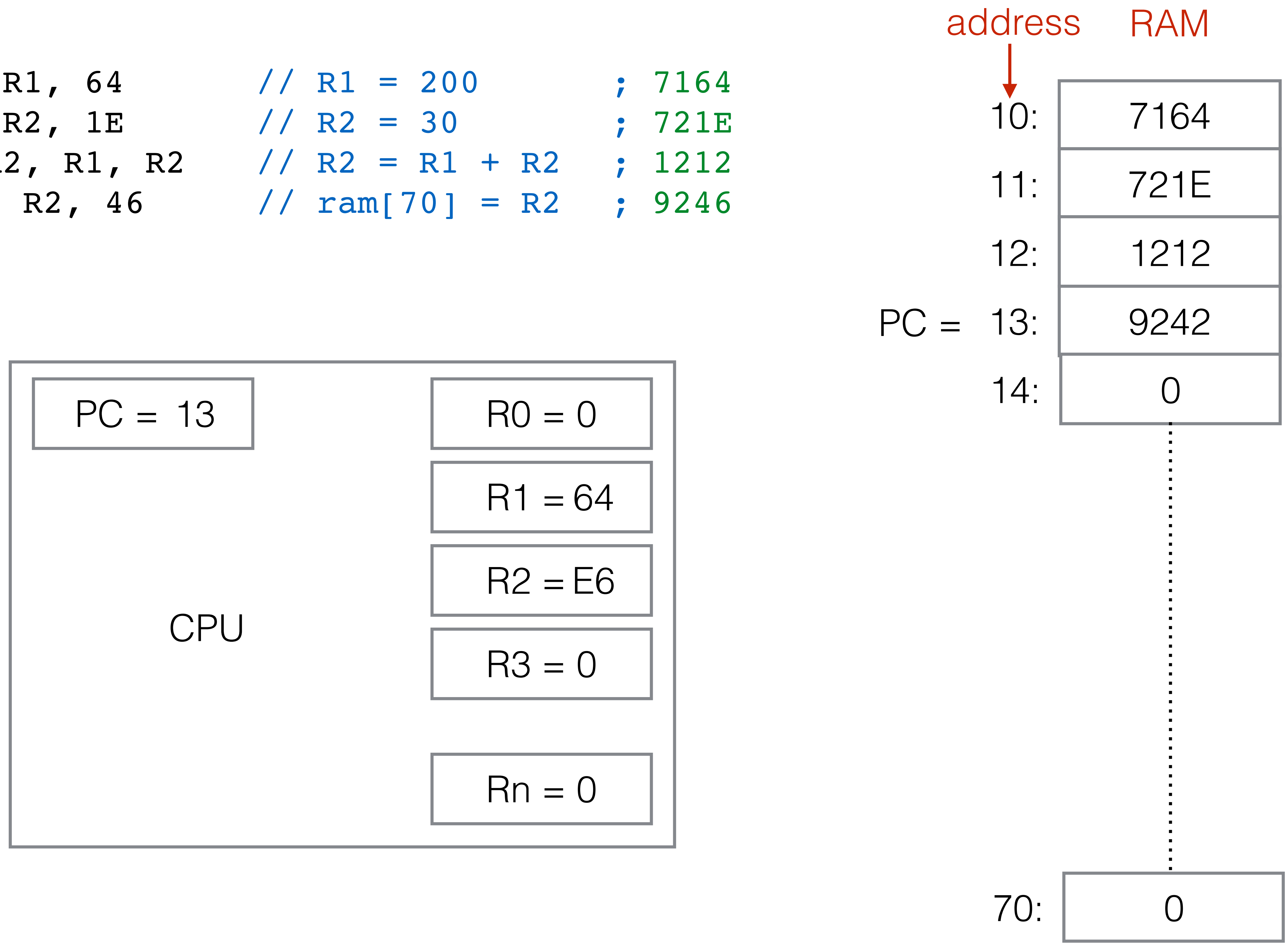


```
LIMM R1, 64        // R1 = 200          ; 7164
LIMM R2, 1E        // R2 = 30           ; 721E
ADD R2, R1, R2     // R2 = R1 + R2      ; 1212
STORE R2, 46       // ram[70] = R2      ; 9246
```

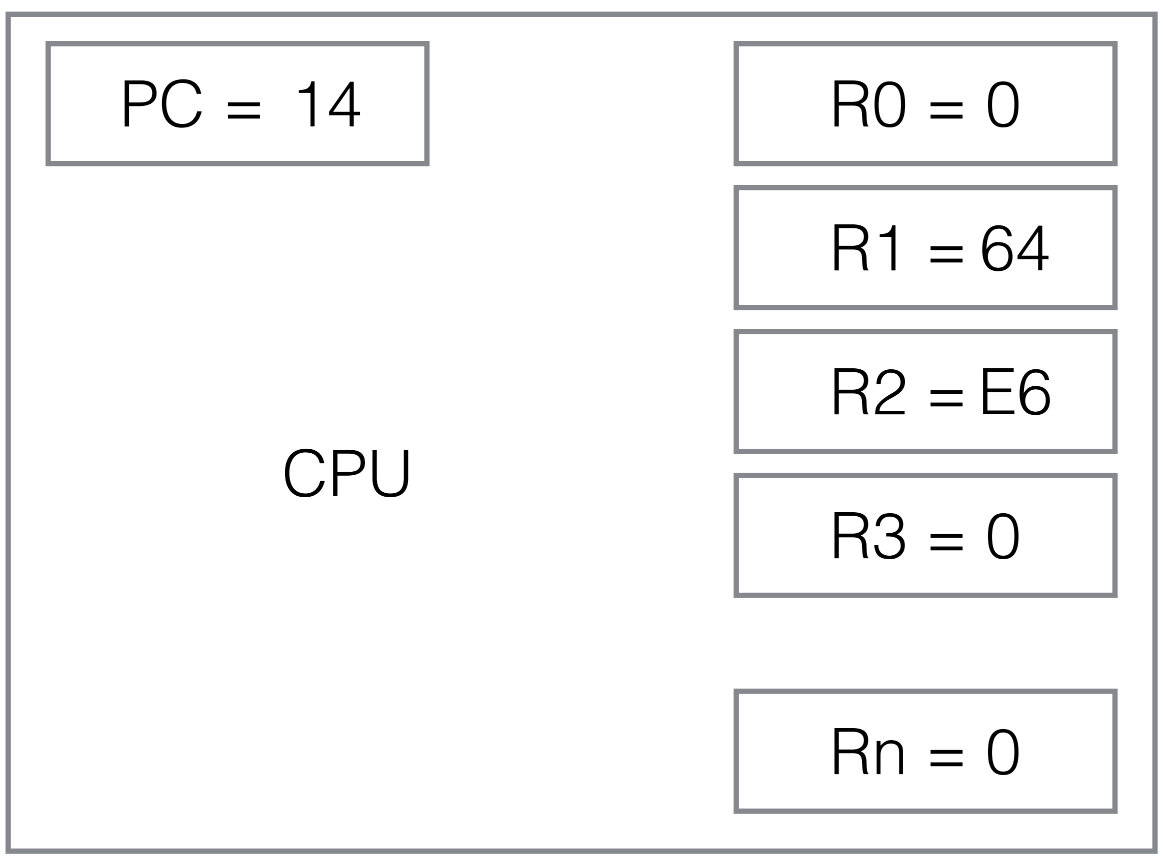


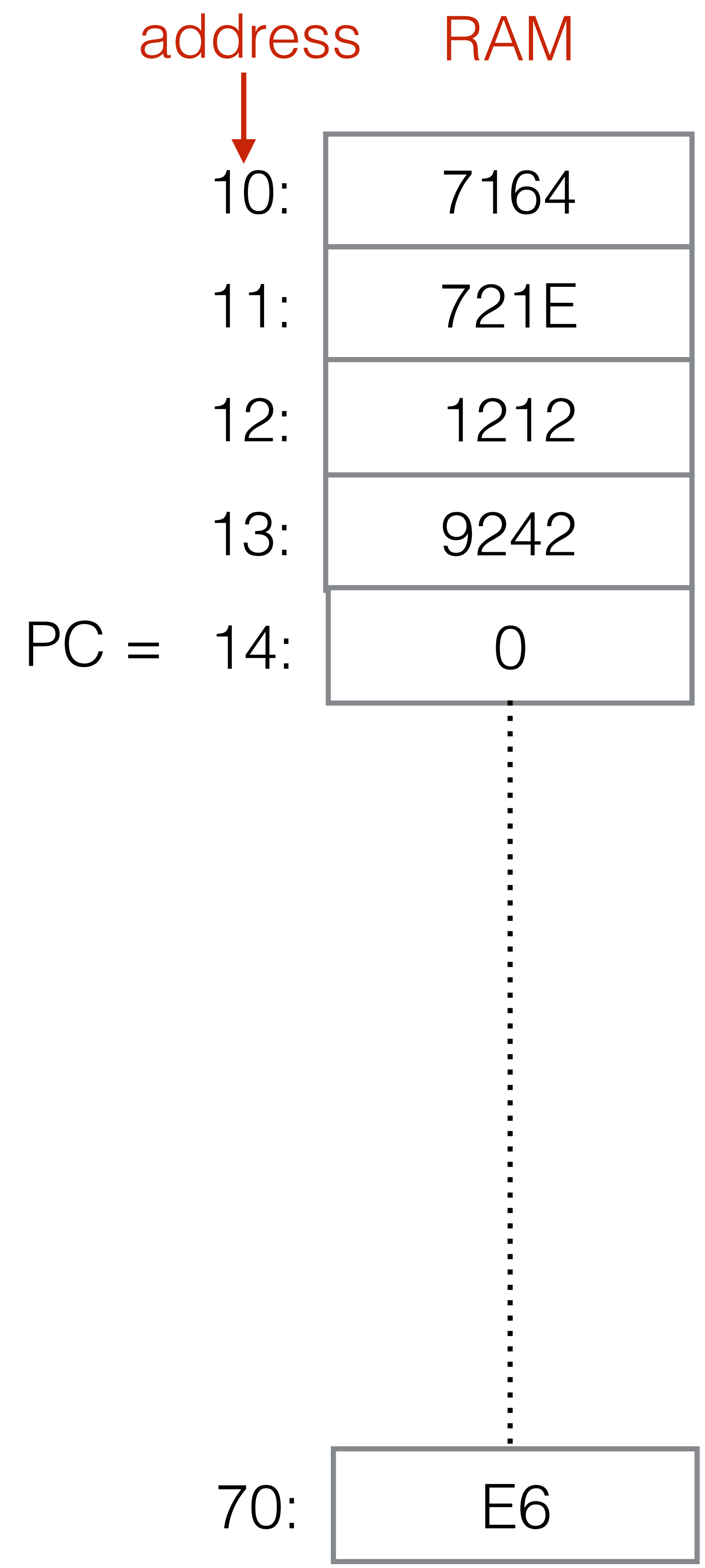


```
LIMM R1, 64        // R1 = 200          ; 7164
LIMM R2, 1E        // R2 = 30           ; 721E
ADD R2, R1, R2     // R2 = R1 + R2      ; 1212
STORE R2, 46       // ram[70] = R2      ; 9246
```



```
LIMM R1, 64         // R1 = 200          ; 7164
LIMM R2, 1E         // R2 = 30           ; 721E
ADD R2, R1, R2      // R2 = R1 + R2      ; 1212
STORE R2, 46        // ram[70] = R2      ; 9246
```





Ending the program

The computer will keep grabbing an integer, interpreting it as an instruction, and then incrementing PC indefinitely.

To stop this process, you have to add a HALT instruction:

Instruction (aka opcode)				ignored											
0	0	0	0	0	1	0	0	1	0	0	0	1	0	0	0

```
0 = HALT
```

```
LIMM R1, 64        // R1 = 200         ; 7164
LIMM R2, 1E        // R2 = 30          ; 721E
ADD R2, R1, R2     // R2 = R1 + R2     ; 1212
STORE R2, 46       // ram[70] = R2     ; 9246
HALT               // stop program     ; 0000
```

Input, Output, and 0

Register 0 always has value 0

Stores to memory location FF writes the value to the output

Reads from memory location FF read a value from the input

`STORE R3, FF`	Write the value R3 to the output
`LOAD R4, FF`	Read the next input value into R3

Example:

```
LIMM R1, 64        // R1 = 200              ; 7164
LOAD R2, FF        // R2 = input            ; 72FF
ADD R2, R1, R2     // R2 = R1 + R2          ; 1212
STORE R2, FF       // write R2 to output    ; 9246
HALT               // stop program          ; 0000
```

Other Arithmetic Operations: AND

`AND Rd, Rs, Rt`

Set register Rd to Rs AND Rt

AND takes two binary numbers a and b and creates a binary c number where the ith bit of c is 1 if and only if the ith bits of both a and b are 1:

a:	0	0	1	0	0	0	1	0	0	0	1	0	1	1	0	0
b:	0	0	1	0	1	0	0	0	0	1	1	0	1	0	0	0
c:	0	0	1	0	0	0	0	0	0	0	1	0	1	0	0	0

1 because both a and b have 1 in this position.

0 because both a has 0 in this position.

Exclusive Or: XOR

`XOR Rd, Rs, Rt`

Set register Rd to Rs XOR Rt

XOR takes two binary numbers a and b and creates a binary c number where the ith bit of c is 1 if either a or b but not both have 1 in their ith bit:

a:	0	0	1	0	0	0	1	0	0	0	1	0	1	1	0	0
b:	0	0	1	0	1	0	0	0	0	1	1	0	1	0	0	0
c:	0	0	0	0	1	0	1	0	0	1	0	0	0	1	0	0

0 because both a and b have 1 in this position.

1 because exactly one of a, b have 1 at this position.

AND and XOR in Go

Go has bitwise operators:

& = AND

^ = XOR

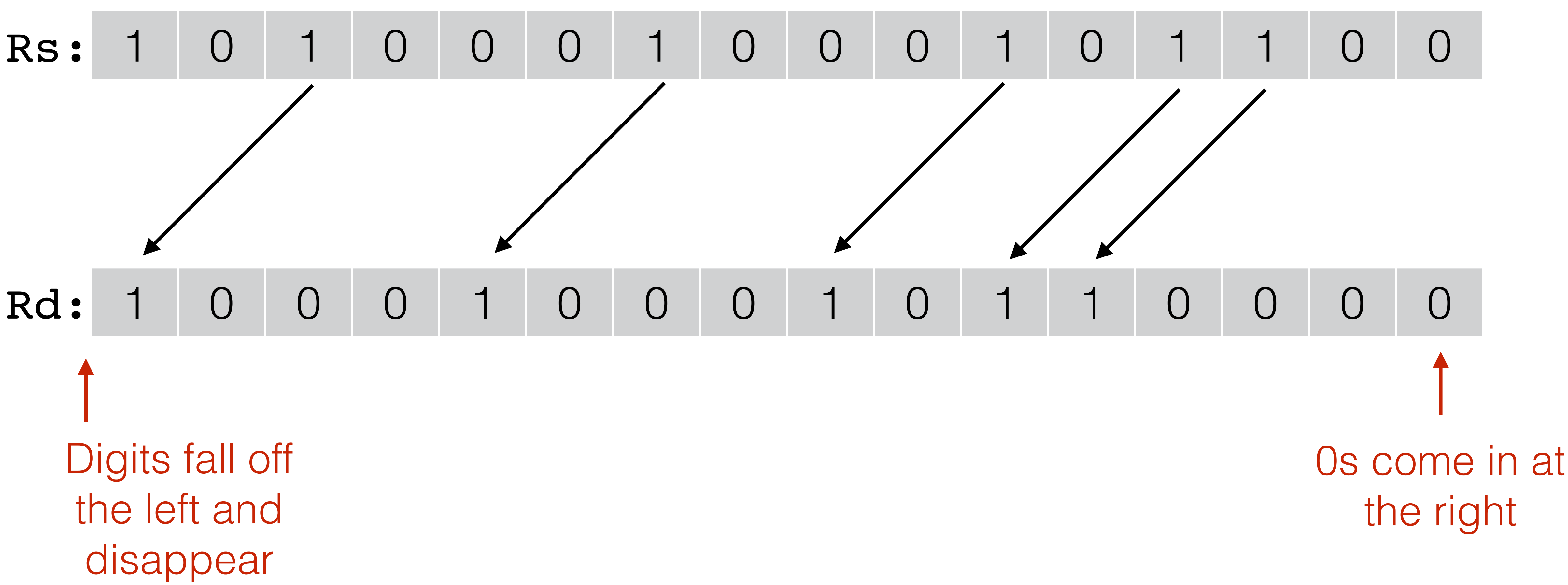
```
var r1 int = 200
var r2 int = 30
r2 = r1 & r2
var r3 int
r3 = r2 ^ r1
```

```
r1 = 0000000011001000
r2 = 0000000000011110
r2 = 0000000000001000

r3 = 0000000011000000
```

Left and Right Shift

`LSHIFT Rd, Rs, Rt`

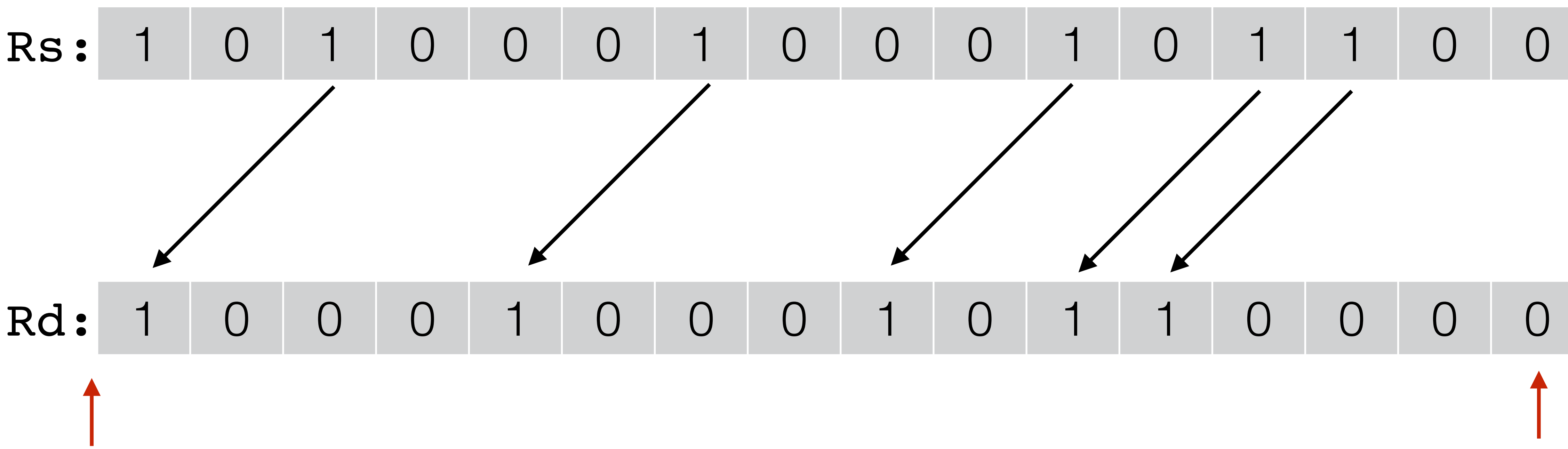
Set register Rd to Rs shifted to the left by Rt digits



Left and Right Shift

```
LSHIFT Rd, Rs, Rt
```

Set register Rd to Rs shifted to the left by Rt digits



Digits fall off the left and disappear

0s come in at the right

RSHIFT is the same except it shifts to the right:

```
RSHIFT Rd, Rs, Rt
```

Set register Rd to Rs shifted to the **right** by Rt digits

AND, XOR, LSHIFT, RSHIFT

`AND Rd, Rs, Rt`	Set register Rd to Rs AND Rt
`XOR Rd, Rs, Rt`	Set register Rd to Rs XOR Rt
`LSHIFT Rd, Rs, Rt`	Set register Rd to Rs << Rt
`RSHIFT Rd, Rs, Rt`	Set register Rd to Rs >> Rt

The instruction format similar to ADD, SUB:

Instruction (aka opcode)				Rd				Rs				Rt			
0	0	1	1	0	0	1	0	0	0	1	0	1	0	0	0

3 = AND	2 = R2	2 = R2	8 = R8	3228_{16}
4 = XOR	2 = R2	2 = R2	8 = R8	4228_{16}
5 = LSHIFT	2 = R2	2 = R2	8 = R8	5228_{16}
6 = RSHIFT	2 = R2	2 = R2	8 = R8	6228_{16}

Load Indirect

Use the value in a register as an address into RAM to read from:

`LOAD.I Rd, Rt`   Set register Rd to ram[Rt]

Instruction (aka opcode)				Rd				ignored				Rt			
1	0	1	0	0	0	1	0	0	0	0	0	1	0	0	0

`10 = LOAD.I`   `2 = R2`   `8 = R8` $\mathtt{A228}_{16}$

What's the analog in Go of this operation?

Load Indirect

Use the value in a register as an address into RAM to read from:

`LOAD.I Rd, Rt` Set register Rd to ram[Rt]

Instruction (aka opcode)				Rd				ignored				Rt			
1	0	1	0	0	0	1	0	0	0	0	0	1	0	0	0

`10 = LOAD.I` `2 = R2` `8 = R8` $A228_{16}$

What's the analog in Go of this operation?

Pointer dereferencing with *:

```
var r8 *int = 70 // not legal in Go
var r2 int
r2 = *r8      // LOAD.I R2 R8
```

Store Indirect

Use the value in a register as an address into RAM to write to:

`STORE.I Rd, Rt`

Set register ram[Rt] to Rd

Instruction (aka opcode)				Rd				ignored				Rt			
1	0	1	1	0	0	1	0	0	0	0	0	1	0	0	0

`11 = LOAD.I`

`2 = R2`

`8 = R8`

$\texttt{B228}_{16}$

What's the analog in Go of this operation?

Store Indirect

Use the value in a register as an address into RAM to write to:

`STORE.I Rd, Rt` Set register ram[Rt] to Rd

Instruction (aka opcode)				Rd				ignored				Rt			
1	0	1	1	0	0	1	0	0	0	0	0	1	0	0	0

`11 = LOAD.I` `2 = R2` `8 = R8` $\mathtt{B228}_{16}$

What's the analog in Go of this operation?

Pointer dereferencing with * in an assignment:

```
var r8 *int = 70 // not legal in Go
var r2 int
*r8 = r2      // STORE.I R2 R8
```

Summary So Far

- Instructions are encoded as integers stored in memory.
- PC incremented after each instruction.
- Can read / write to memory using either an explicit address (immediate), or the contents of another register as the address (indirect)
- Can perform arithmetic operations on registers.
- Input / Output done via reads/writes to special memory locations.

How would we write an **for** loop?

Jumps: Manipulating the PC

`JUMP Rd`	Set PC to Rd
`JMP0 Rd, addr`	If Rd == 0, set PC to addr

```
15: LIMM R1, 1        // 7101

16: LIMM R5, 18       // 7518
17: ADD R6, R6, R0    // 1600
18: JMP0 R3, 1C       // C31C
19: ADD R6, R6, R2    // 1662
1A: SUB R3, R3, R1    // 2441
1B: JUMP R5           // E500
1C: STORE R2, FF      // 92FF
```

```
func mul(r2, r3 int) {
  r1 := 1
  r4 := r3 + 0


  for r4 != 0 {
    r2 = r2 + r3
    r4 = r4 - r1
  }
  fmt.Print(r2)
}
```

If Statements

`JMPP Rd, addr` If Rd > 0, set PC to addr

```
if r3 >= r4 {
   fmt.Println(r3)
} else {
   fmt.Println(r4)
}
```

```
10: LIMM r5, 16
11: SUB r2, r4, r3
12: JMPP r2, 15
13: STORE r3, FF
14: JUMP r5
15: STORE r4, FF
16:
```

How would we write a condition a == b?

If Statements

`JMPP Rd, addr` — If Rd > 0, set PC to addr

```
if r3 >= r4 {
    fmt.Println(r3)
} else {
    fmt.Println(r4)
}
```

	`10: LIMM r5, 16`
condition	`11: SUB r2, r4, r3`
	`12: JMPP r2, 15`
then part	`13: STORE r3, FF`
	`14: JUMP r5`
else part	`15: STORE r4, FF`
	`16:`

How would we write a condition a == b?

If Statements

`JMPP Rd, addr`

If Rd > 0, set PC to addr

```
if r3 >= r4 {
    fmt.Println(r3)
} else {
    fmt.Println(r4)
}
```

condition

then part

else part

```
10: LIMM r5, 16
11: SUB r2, r4, r3
12: JMPP r2, 15
13: STORE r3, FF
14: JUMP r5
15: STORE r4, FF
16:
```

if condition was false
(r4-r3 > 0)

How would we write a condition a == b?

If Statements

`JMPP Rd, addr` If Rd > 0, set PC to addr

```
if r3 >= r4 {
    fmt.Println(r3)
} else {
    fmt.Println(r4)
}
```

```
           10: LIMM r5, 16
condition  11: SUB r2, r4, r3
           12: JMPP r2, 15
then part  13: STORE r3, FF
           14: JUMP r5
else part  15: STORE r4, FF
           16:
```

if condition was false
(r4-r3 > 0)

Skip the else part if
we did the then part

How would we write a condition a == b?

Example If Statement #2

```
if r3 == r4 {
   fmt.Println(r3)
} else {
   fmt.Println(r4)
}
```

```
10: LIMM r5, 16
11: SUB r2, r4, r3
12: JMP0 r2, 15
13: STORE r4, FF
14: JUMP r5
15: STORE r3, FF
16:
```

Example If Statement #2

```
if r3 == r4 {
    fmt.Println(r3)
} else {
    fmt.Println(r4)
}
```

```
           10: LIMM r5, 16
condition  11: SUB r2, r4, r3
           12: JMP0 r2, 15
else part  13: STORE r4, FF
           14: JUMP r5
then part  15: STORE r3, FF
           16:
```

Example If Statement #2

```
if r3 == r4 {
   fmt.Println(r3)
} else {
   fmt.Println(r4)
}
```

```
           10: LIMM r5, 16
condition  11: SUB r2, r4, r3
           12: JMP0 r2, 15
else part  13: STORE r4, FF
           14: JUMP r5
then part  15: STORE r3, FF
           16:
```

if condition was true
(r4-r3 == 0)

Example If Statement #2

```
if r3 == r4 {
    fmt.Println(r3)
} else {
    fmt.Println(r4)
}
```

```
10: LIMM r5, 16
11: SUB r2, r4, r3
12: JMP0 r2, 15
13: STORE r4, FF
14: JUMP r5
15: STORE r3, FF
16:
```

condition (lines 11–12)

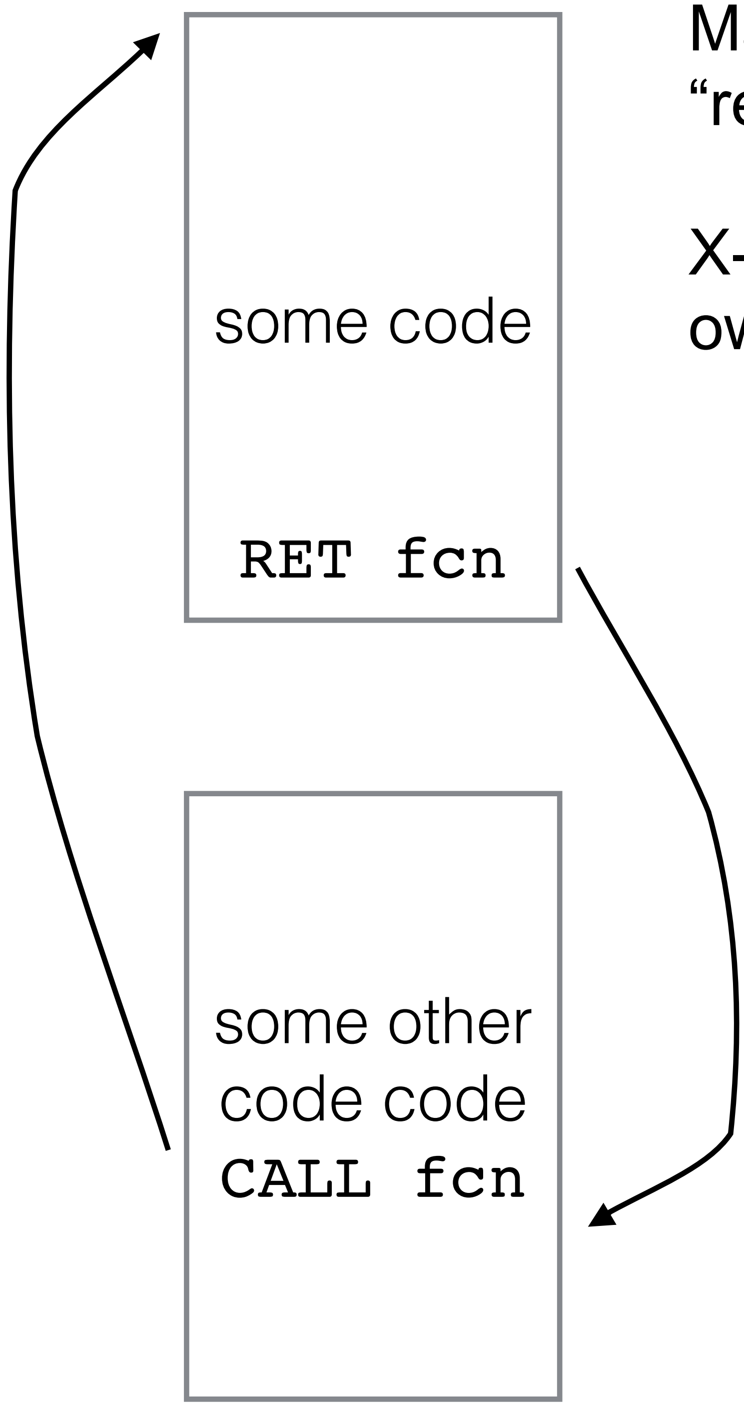
else part (line 13)

then part (line 15)

if condition was true (r4-r3 == 0)

Skip the then part if we did the else part

Function Calls



Many processors (including Intel) have explicit “call” and “return” instructions.

X-TOY doesn’t: it has an instruction that lets you write your own RET (RETURN) and CALL functions:

`JAL Rd, addr`

Set Rd to PC+1
Set PC to addr

This is “jump and link”: it jumps to an address, and saves where you were in a register.

```
CALL addr      JAL R15, addr

RETURN         JUMP R15
```

Function Call Parameters

Note: a function is just a block of instructions that we plan to jump into from elsewhere in the program.

How can we pass parameters into a “function”?

Function Call Parameters

Note: a function is just a block of instructions that we plan to jump into from elsewhere in the program.

How can we pass parameters into a “function”?

Option 1: The caller and the function just agree about which registers to store the parameters in:

```
// R2 and R3 should contain the
// numbers to multiply; R15 should
// contain the address to return to
15: LIMM R1, 1          // 7101
16: LIMM R5, 18         // 7518
17: ADD R6, R6, R0      // 1600
18: JMP0 R3, 1C         // C31C
19: ADD R6, R6, R2      // 1662
1A: SUB R3, R3, R1      // 2441
1B: JUMP R5             // E500
1C: STORE R2, FF        // 92FF
1D: JUMP RF             // EF00
```

```
func mul(r2, r3 int) {
  r1 := 1
  r4 := r3 + 0

  for r4 != 0 {
    r2 = r2 + r3
    r4 = r4 - r1
  }
  fmt.Print(r2)
}
```

Example Call in X-TOY

```
program Mul
// Input: None
// Output: 8 * 2 = 16 = 0x10
//
10: 7208   R[2] <- 0008
11: 7302   R[3] <- 0002
12: FF15   R[F] <- pc; goto 15
13: 0000   halt

function mul
// Input: R2 and R3
// Return address: R15
// Output: to screen
// Temporary variables: R5, R6
15: 7101   R[1] <- 0001
16: 7518   R[5] <- 0018
17: 1600   no-op
18: C31C   if (R[3] == 0) goto 1C
19: 1662   R[6] <- R[6] + R[2]
1A: 2331   R[3] <- R[3] - R[1]
1B: E500   goto R[5]
1C: 96FF   write R[6]
1D: EF00   goto R[F]
```

Notes:

Program starts at address 0x10

You must say the address of every line of code by prefixing it with addr:

(X-TOY Bugs? 1600 is not a no-op and FF15 sets RF to **pc+1**)

Option 2: Push Parameters onto the Stack

Agree that the stack grows from memory address FE downward towards 0

Agree that R14 always holds a pointer to the top of the stack

"PUSH R7"

```
LIMM R1, 1
ADD RE, RE, R1
STORE.I R7 RE
```

"POP R9"

```
LOAD.I R9 RE
LIMM R1, 1
SUB RE, RE, R1
```

Option 2: Push Parameters onto the Stack

```
// The top of the stack should contain the
// two numbers to multiply; R15 should
// contain the address to return to

10: LOAD.I R2, RE        // A20E     Grab the number at the top of the stack
11: LIMM R1, 1           // 7101     "pop": move the top of the stack
12: SUB RE, RE, R1       // 2EE1     down by 1
13: LOAD.I R3, RE        // A30E     Grab the number at the top of the stack
14: SUB RE, RE, R1       // 2EE1     move the top of the stack down by 1

15: LIMM R1, 1           // 7101
16: LIMM R5, 18          // 7518
17: ADD R6, R6, R0       // 1600
18: JMP0 R3, 1C          // C31C
19: ADD R6, R6, R2       // 1662
1A: SUB R3, R3, R1       // 2441
1B: JUMP R5              // E500
1C: STORE R6, FF         // 96FF
1D: JUMP RF              // EF00
```

How many registers are there?

16 in X-TOY
This is a typical number (8-32)

What if you “run out”?

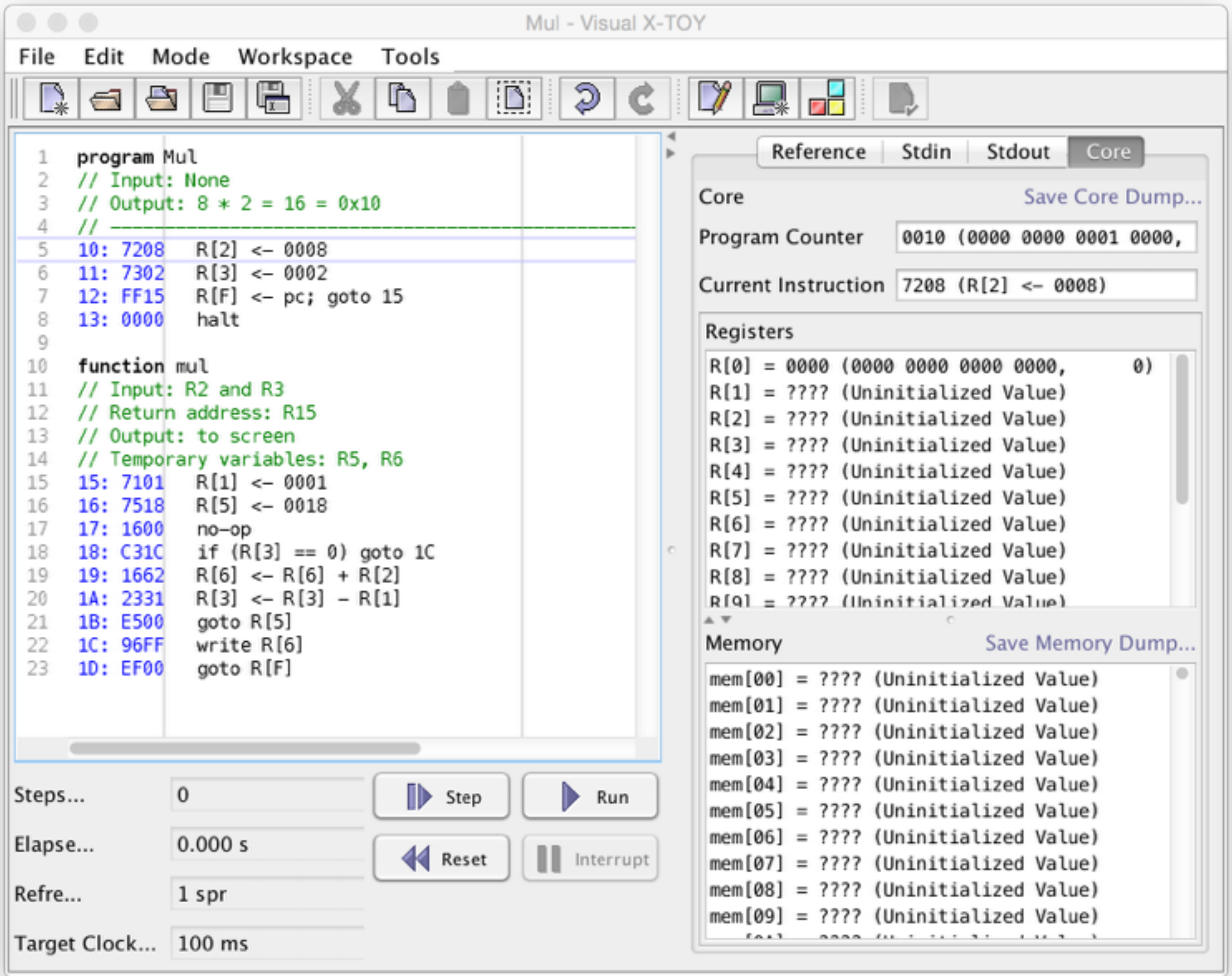
Yep, that’s a problem: you may have to shuffle variables between RAM and registers if you need to use the registers for something.

Summary of X-TOY Computer

```
INSTRUCTION FORMATS

              | .... | .... | .... | .... |
  Format 1:   |  op  |  d   |  s   |  t   |
  Format 2:   |  op  |  d   |     imm     |


ARITHMETIC and LOGICAL operations
    1: add              R[d] <- R[s] + R[t]
    2: subtract         R[d] <- R[s] - R[t]
    3: and              R[d] <- R[s] & R[t]
    4: xor              R[d] <- R[s] ^ R[t]
    5: shift left       R[d] <- R[s] << R[t]
    6: shift right      R[d] <- R[s] >> R[t]

TRANSFER between registers and memory
    7: load immediate R[d] <- imm
    8: load           R[d] <- mem[imm]
    9: store          mem[imm] <- R[d]
    A: load indirect  R[d] <- mem[R[t]]
    B: store indirect mem[R[t]] <- R[d]

CONTROL
    0: halt           halt
    C: branch zero    if (R[d] == 0) pc <- imm
    D: branch pos.    if (R[d] > 0) pc <- imm
    E: jump register  pc <- R[d]
    F: jump and link  R[d] <- pc; pc <- imm


R[0] always reads 0.
Loads from mem[FF] come from stdin.
Stores to mem[FF] go to stdout.
```

From the X-TOY instructions

X-TOY Environment



Intel 8088 Instruction Set

ADD	Add
SUB	Subtraction

AND	Logical AND
XOR	Exclusive OR
SHL	Shift left (unsigned shift left)
SHR	Shift right (unsigned shift right)

JMP	Jump
JCXZ	Jump if CX is zero
JNS	Jump if not negative

INC	Increment by 1
DEC	Decrement by 1

Another motivation for the ++ and - - statements in Go (and C, c++, Java..): They correspond directly to a machine instruction.

PUSH	Push data onto stack
POP	Pop data from stack

Has several instructions to push and pop data onto THE stack.

and about 80 others…

http://en.wikipedia.org/wiki/X86_instruction_listings#Original_8086.2F8088_instructions

MacPaint

http://www.computerhistory.org/atchm/macpaint-and-quickdraw-source-code/

Summary

- Trees are an incredibly common way to organize data:
  - folders on your hard drives
  - URLs: http://www.cs.cmu.edu/~ckingsf/software/sailfish
  - BST, Splay trees, AVL trees, B-trees, Quad-trees, kd-trees, red-black trees, M-trees, … probably thousands of variants that are good for different data and different queries.
- Binary trees in particular are nice because each node partitions the data into 2 subsets and because there are nice relationships between # of nodes and # of leaves, etc.
- Typically, trees are represented using nodes & pointers, though this does not have to be the case.