Hmmm—part 2

Write a Hmmm program that:

1. reads a number n from the user
2. doubles n
3. writes the result ($2n$) to the screen

Firstname Lastname **Th. 9 / 27**

(Your response)

Take-home midterm #1

Available: this Sunday night (9/30)

Must return by: next Sunday (10/7) at 5pm

Time-limit: one sitting (with small breaks)

Covers: everything up to and including this week
automata, circuits, and assembly

Resources: one, 8½ x 11 sheet of notes (double-sided)

Honor code: don’t discuss exam questions

There will be **assignments**, too.

Today’s goal:

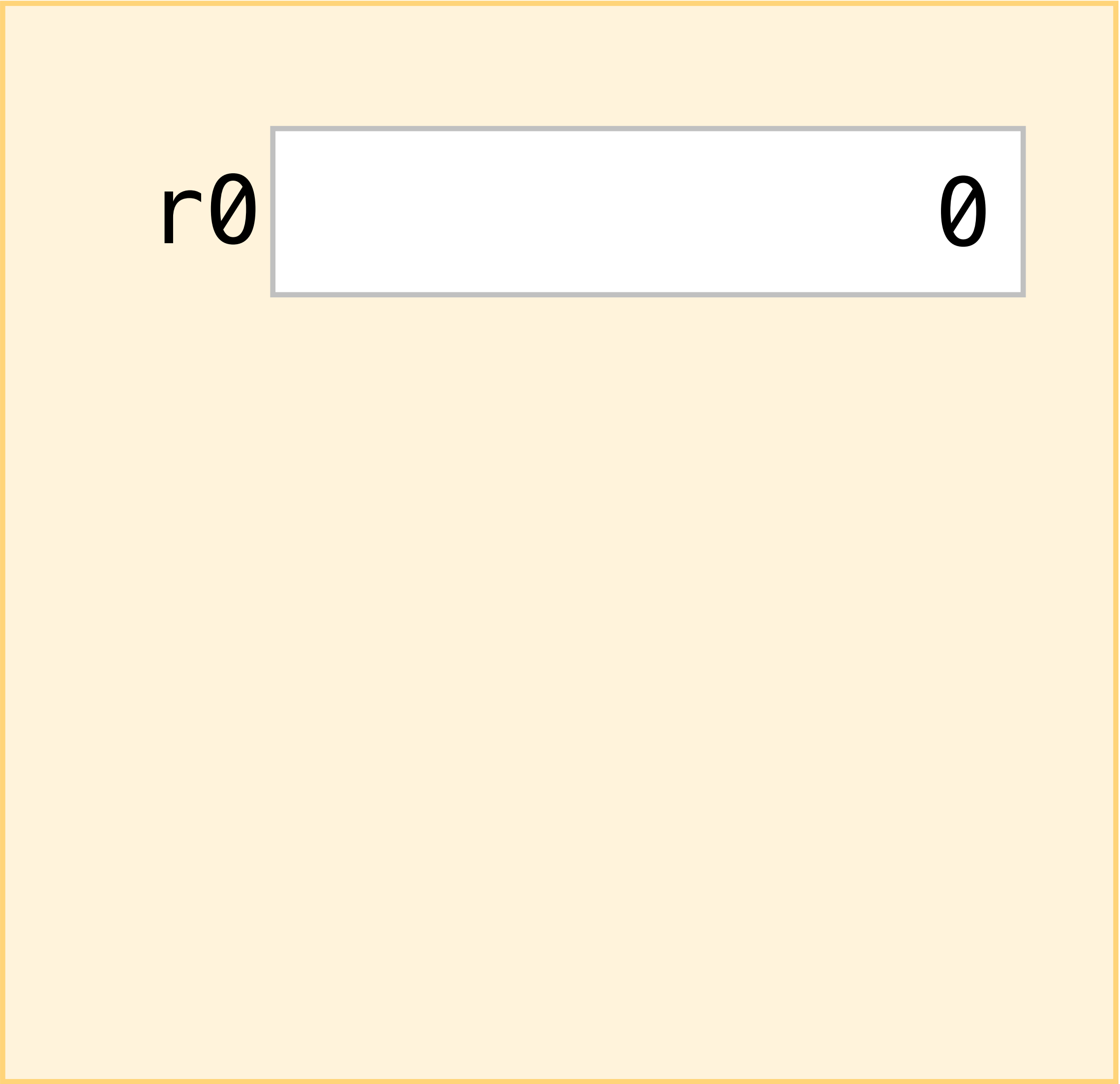
Understand how functions work

(in a stored-program machine)

Hmmm conventions

Human programming practices that help us write correct *and* readable programs

`r0` always contains the value `0`



Write lots of comments / documentation!

Clearer is better than shorter (but shorter *can be* clearer).

A program that computes 2n

```
00 read r1
01 add r1 r1 r1  # n = 2 * n
02 write r1
03 halt
```

A program that computes 2n

This program uses a “function” to compute 2n

```
00 read r1
01 jumpn 04    # double(n)
02 write r1
03 halt

# double(n)
04 add r1 r1 r1  # n = 2 * n
05 jumpn 02
```

A program that computes 4n

Uh, oh. What if we need to call the function twice?
We’ll need a place to save the “return location”...

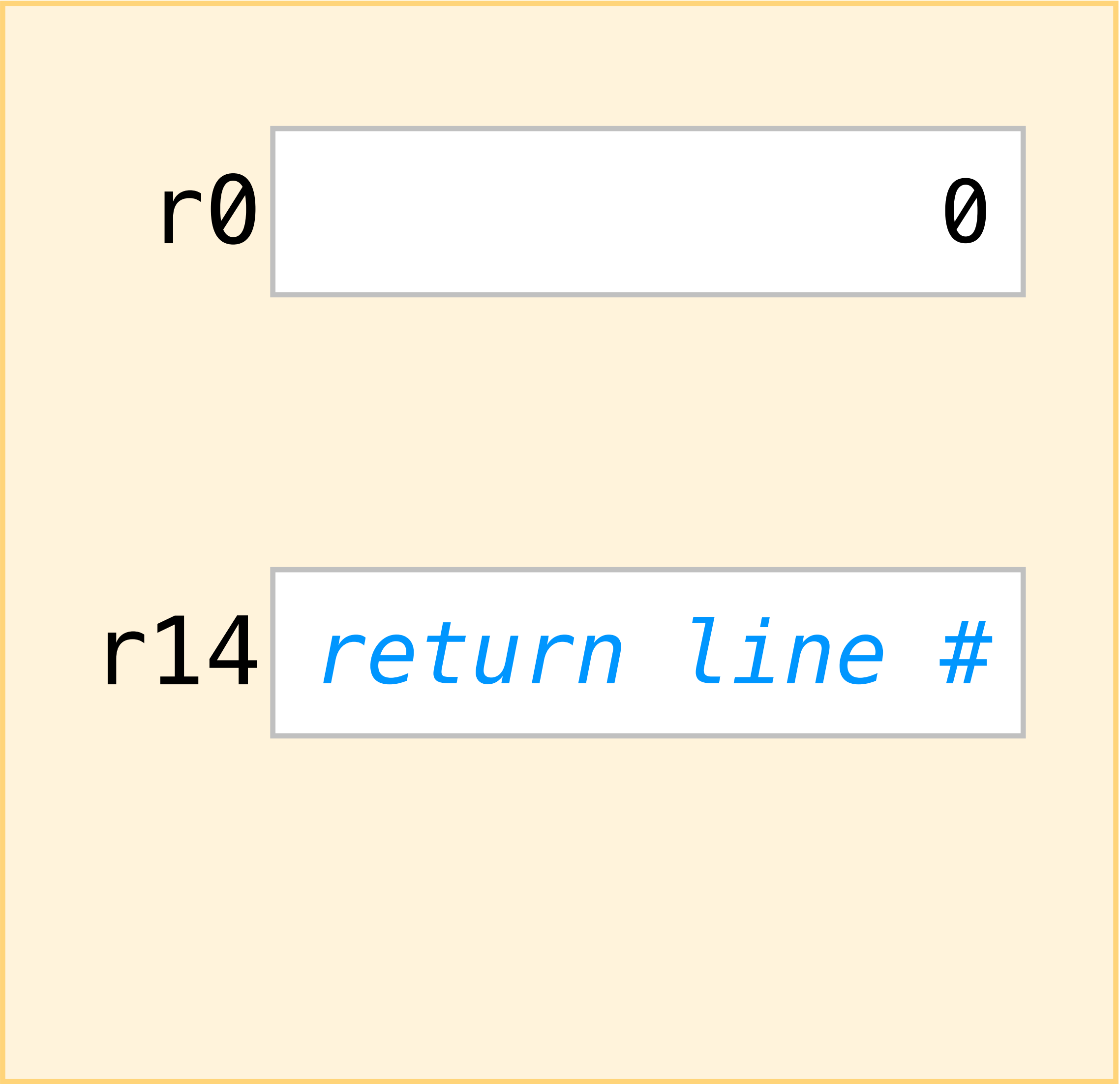
```
00 read r1
01 jumpn 05    # double(n)
02 jumpn 05    # double(n)
03 write r1
04 halt
# double(n)
05 add r1 r1 r1  # n = 2 * n
06 jumpn 02
```

Hmmm conventions

Human programming practices that help us write correct *and* readable programs

`r0` always contains the value `0`

`r14` is for the return location



Write lots of comments / documentation!
Clearer is better than shorter (but shorter *can be* clearer).

Calling a function

Set `r14` to be the next line, then jump to the start of the function

```
00 read r1
01 calln r14 05
02 ...

      ...

05 add r1 r1 r1
06 ...
```

pc 01
r14 0

pc 05
r14 02

“return location” function start

calln r14 05

Returning from a function

`jumpr r14`

```
00 read r1
01 calln r14 05
02 ...

     ...

05 add r1 r1 r1
06 jumpr r14
```

pc 06
r14 02

pc 02
r14 02

“return location”

`jumpr r14`

A program that computes 4n

This program saves the return location.

```
00 read r1
01 calln r14 05    # double(n)
02 calln r14 05    # double(n)
03 write r1
04 halt
# double(n)
05 add r1 r1 r1  # n = 2 * n
06 jumpr r14
```

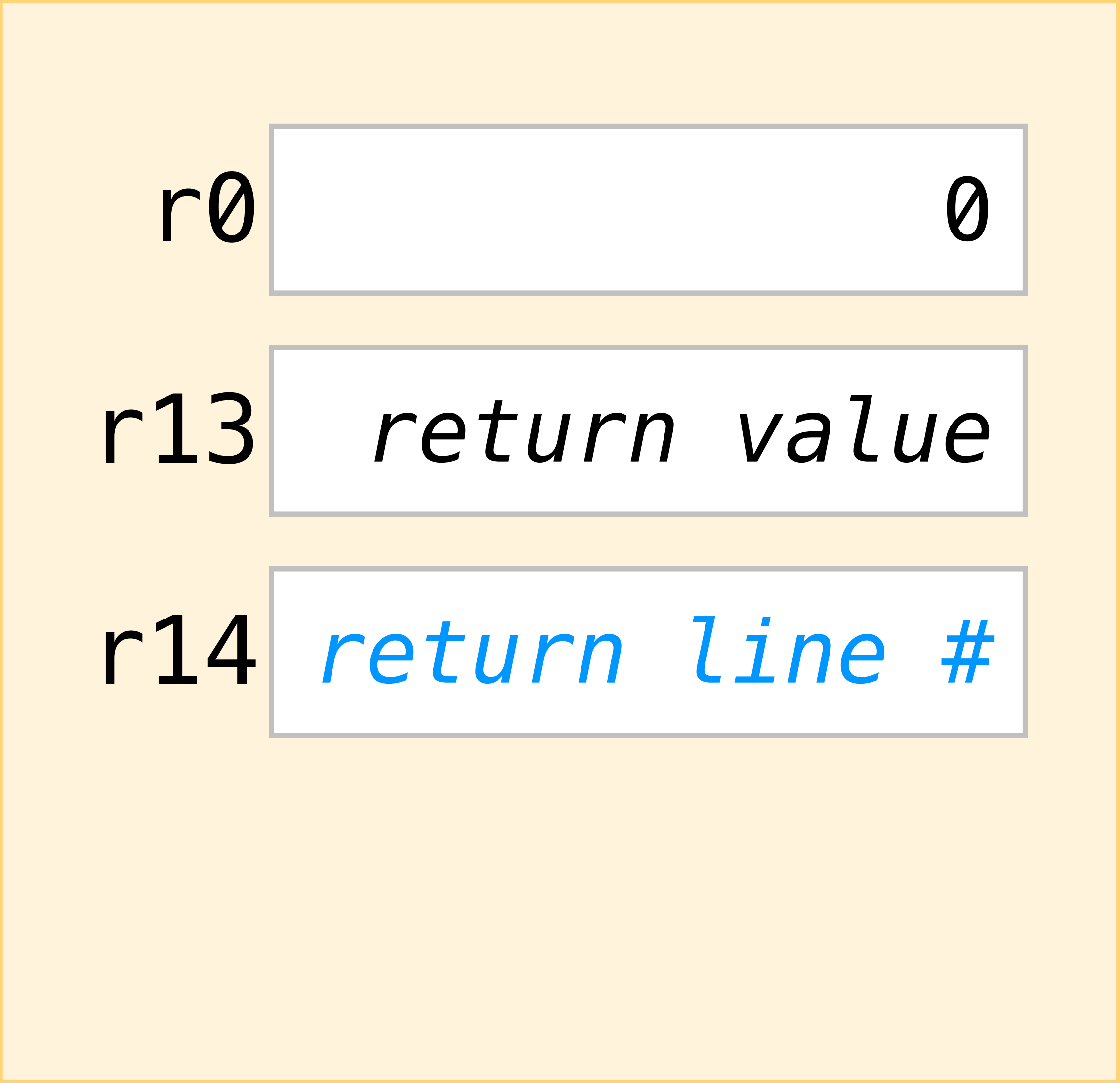
Hmmm conventions

Human programming practices that help us write correct *and* readable programs

`r0` always contains the value `0`

`r13` is for the return value

`r14` is for the return location



r0 0

r13 *return value*

r14 *return line #*

Write lots of comments / documentation!

Clearer is better than shorter (but shorter *can be* clearer).

A program that computes 4n

This program saves the return value and return location.

```
00 read r1
01 calln r14 06
02 copy r1 r13   # prepare arguments
03 calln r14 06
04 write r13
05 halt
# double(n)
06 add r13 r1 r1  # return value = 2 * n
07 jumpr r14
```

A program that computes 4n

Uh, oh. What if the caller needs the register values after the call?
We’ll need a place to save the caller’s state…

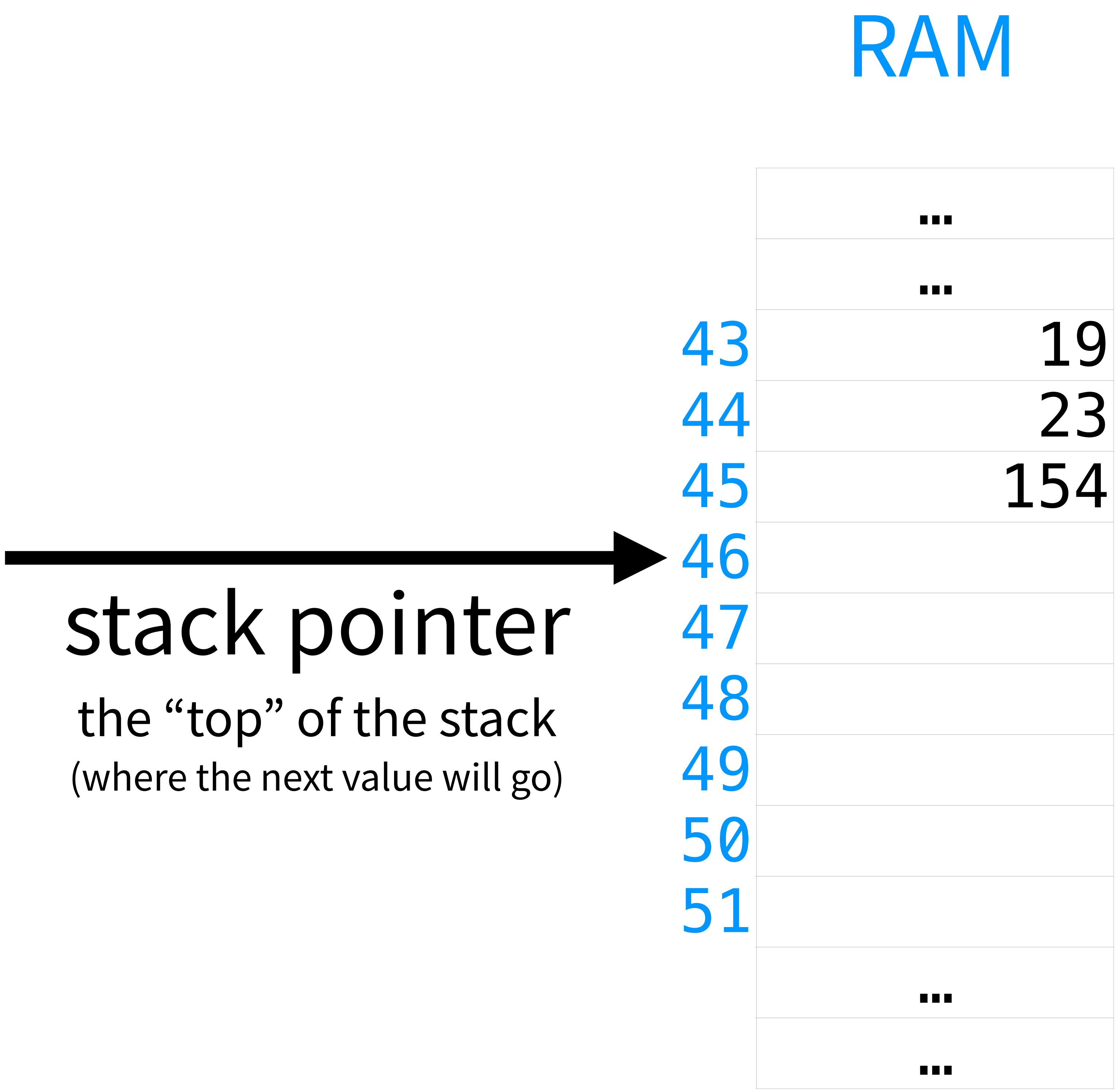
```
00 read r1
01 calln r14 06   # quadruple(n)
02 write r13
03 halt

# double(n)
04 add r13 r1 r1  # return value = 2 * n
05 jumpr r14

# quadruple(n)
06 calln r14 04   # double(n)
07 copy r1 r13    # prepare arguments
08 calln r14 04   # double(n)
09 jumpr r14
```

The stack

A place in RAM where we can save values for later



RAM

	...
	...
43	19
44	23
45	154
46	
47	
48	
49	
50	
51	
	...
	...

stack pointer

the “top” of the stack

(where the next value will go)

Hmmm conventions

Human programming practices that help us write correct *and* readable programs

`r0` always contains the value `0`

`r13` is for the return value

`r14` is for the return location

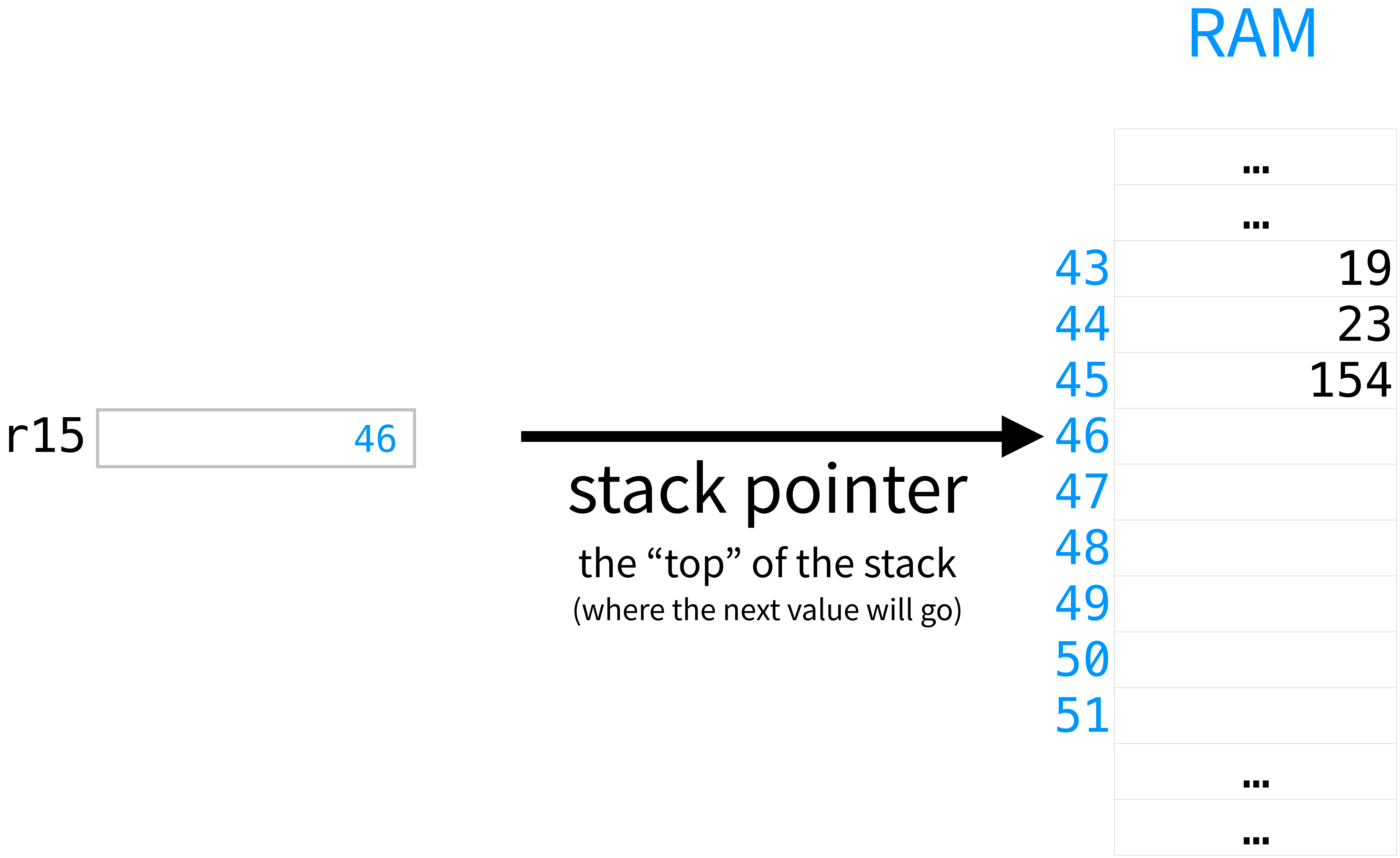
`r15` is for the stack pointer

`r0`	`0`
`r13`	`return value`
`r14`	`return line #`
`r15`	`stack pointer`

Write lots of comments / documentation!
Clearer is better than shorter (but shorter *can be* clearer).

The stack

A place in RAM where we can save values for later



RAM

	...
	...
43	19
44	23
45	154
46	
47	
48	
49	
50	
51	
	...
	...

r15 46

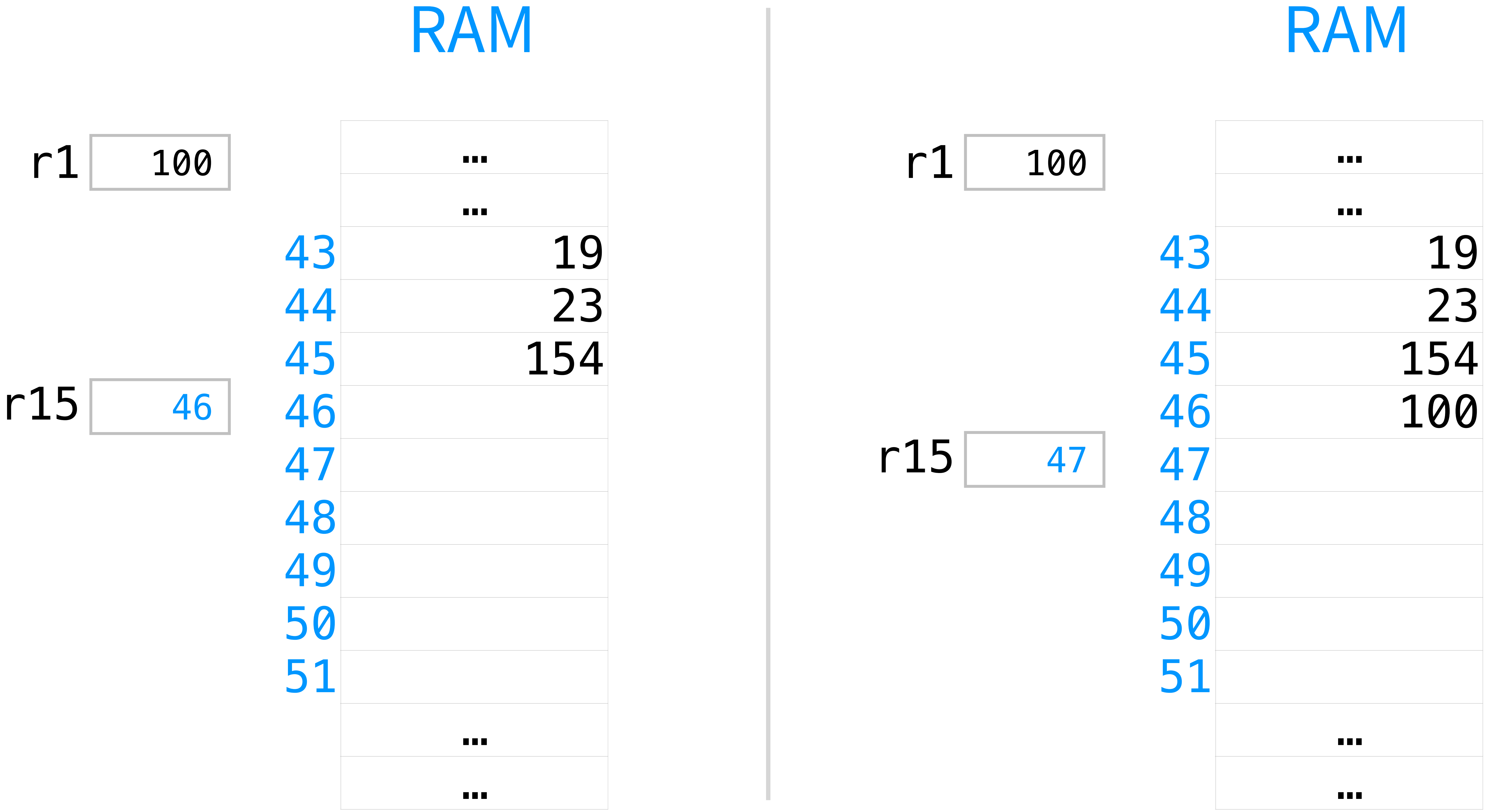
stack pointer

the “top” of the stack

(where the next value will go)

Pushing onto the stack

Add a new value to the “top” of the stack



RAM

r1 100

r15 46

	…
	…
43	19
44	23
45	154
46	
47	
48	
49	
50	
51	
	…
	…

RAM

r1 100

r15 47

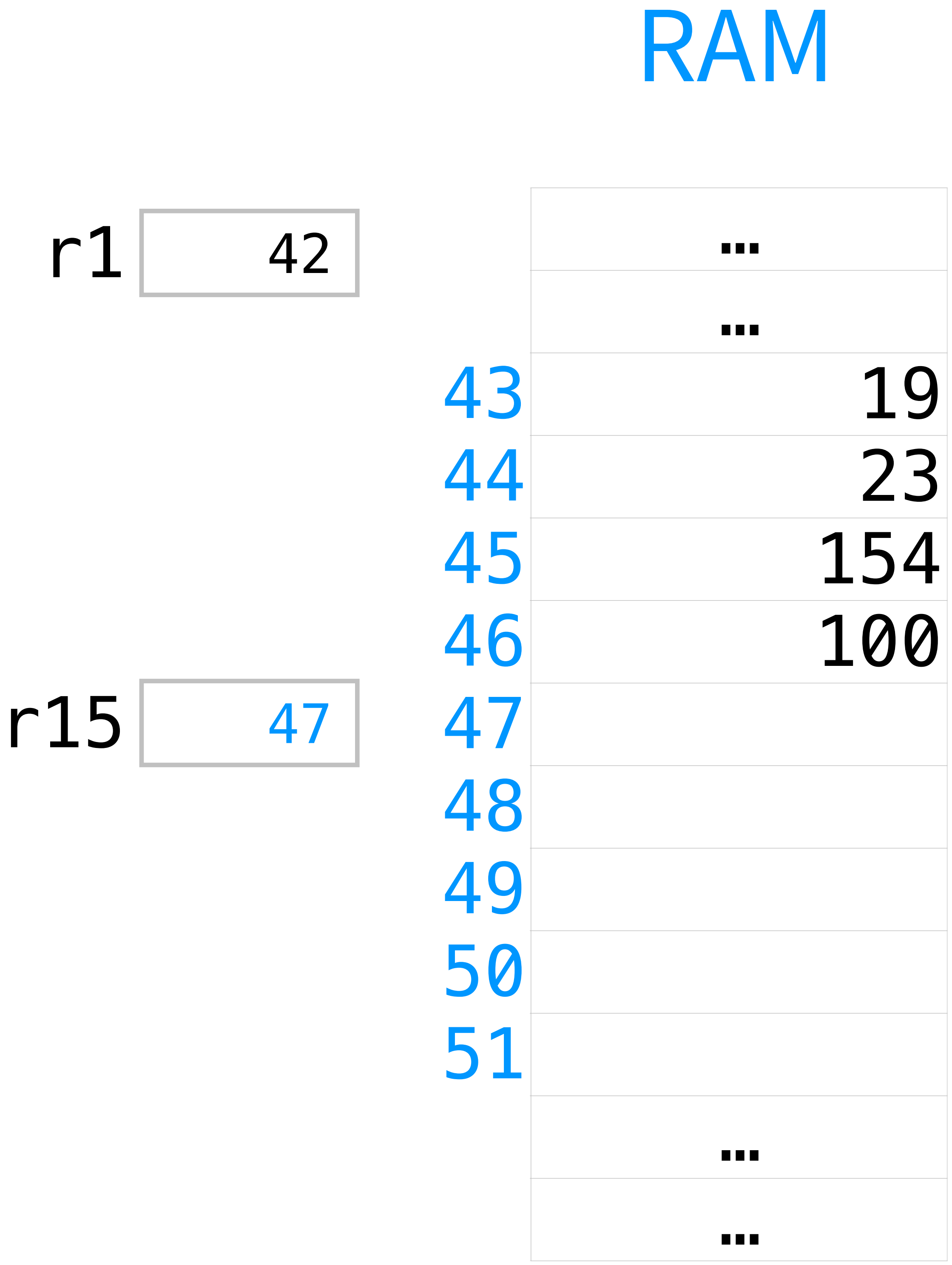
	…
	…
43	19
44	23
45	154
46	100
47	
48	
49	
50	
51	
	…
	…

“local variable” Stack pointer

```
pushr r1 r15
```

Popping from the stack

Take an existing value from the "top" of the stack

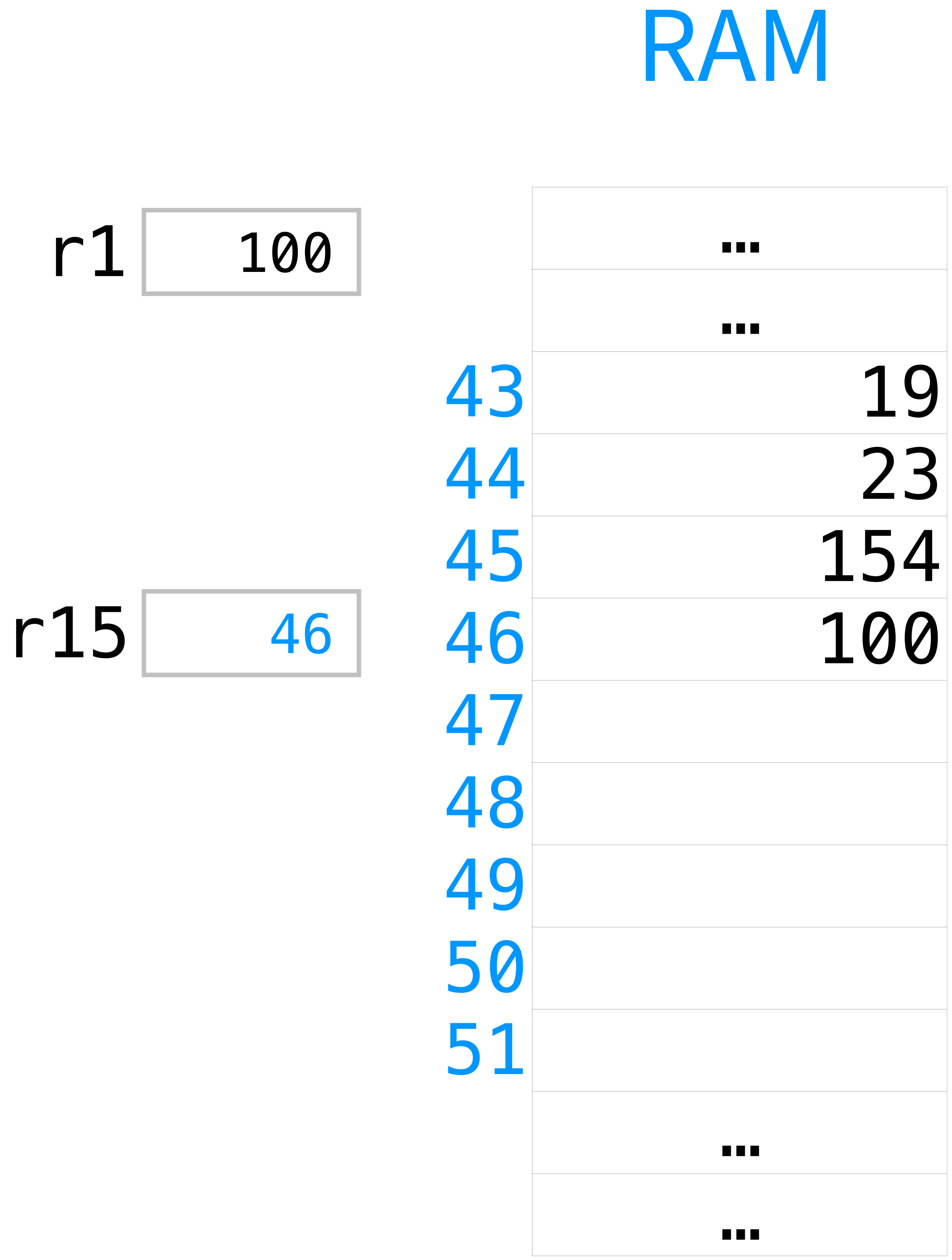


RAM

r1 42

r15 47

	…
	…
43	19
44	23
45	154
46	100
47	
48	
49	
50	
51	
	…
	…



RAM

r1 100

r15 46

	…
	…
43	19
44	23
45	154
46	100
47	
48	
49	
50	
51	
	…
	…

"local variable" Stack pointer

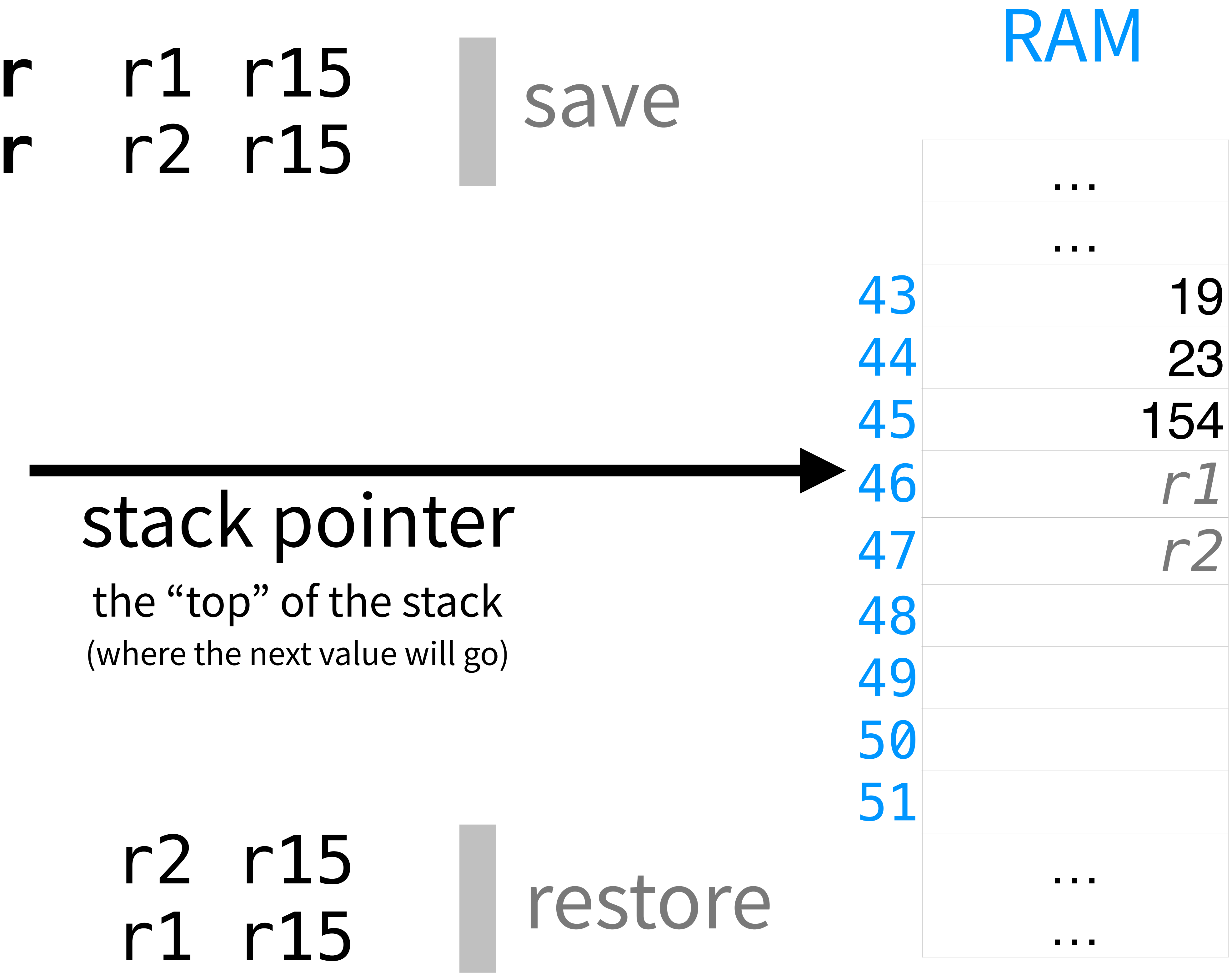
```
popr    r1  r15
```

The stack

“Last-in, first out” (LIFO): pop values in reverse order

```
pushr   r1 r15
pushr   r2 r15
```

save



RAM

	...
	...
43	19
44	23
45	154
46	*r1*
47	*r2*
48	
49	
50	
51	
	...
	...

stack pointer

the “top” of the stack

(where the next value will go)

```
popr    r2 r15
popr    r1 r15
```

restore

A program that computes 4n

This program uses the stack to save and restore the caller’s state.

```
00 setn r15 100    # stack starts at address 100 & grows UP
01 read r1
02 calln r14 07    # quadruple(n)
03 write r13
04 halt

# double(n)
05 add r13 r1 r1   # return value = 2 * n
06 jumpr r14

# quadruple(n)
07 pushr r14 r15
08 calln r14 05    # double(n)
09 popr  r14 r15
10 copy r1 r13     # prepare arguments
11 pushr r14 r15
12 calln r14 05    # double(n)
13 popr  r14 r15
14 jumpr r14
```

Function calls in Hmmm

treat register values
like local variables!

Caller (outside the function): assume the function writes to every register

Callee (inside the function): assume every register is yours

```
# initialize stack pointer
setn   r15 S
```

```
# save any register value that I'll need later      caller
pushr rN r15


# prepare the arguments by assigning values to registers

calln r14 N      # call the function


# restore all the register values that I saved (LIFO!)
popr  rN r15
```

```
N # function start                                    callee
# write to registers with gleeful abandon
# if the function should return a value, save it in r13
M jumpr  r14           # return
```

https://www.youtube.com/watch?v=QNoQvjlmGdk

Let’s practice!

Write a Hmmm program that reads a positive integer value *n* into `r1`, then writes the value n! + n to the screen.

Ask yourself: which register(s) do I need to save / restore?

tinyurl.com/hmc-hmmm

Here is a function that computes `r1!` and stores the result in `r13`:

`r13 = r1!`

```
10 setn r13 1
11 jeqzn r1 15
12 mul r13 r13 r1
13 addn r1 -1
14 jumpn 11
15 jumpr r14
```

Bonus: can you write a recursive factorial?

Let's practice!

A Hmmm program that reads a positive integer value n into `r1`, then writes the value n! + n to the screen.

```
# initialization
00 setn r15 100    # start the stack pointer at address 100

# read the input from the user
01 read r1

# save r1 so we can use it after the function call
02 pushr r1 r15

# call the factorial function: r13 = n!
03 calln r14 10

# restore r1, so we can add it to the result of r1!
04 popr r1 r15

# compute and print the result
05 add r13 r13 r1 # r13 += r1
06 write r13
07 halt
```

```
10 r13 = r1!
```

Hmmm conventions

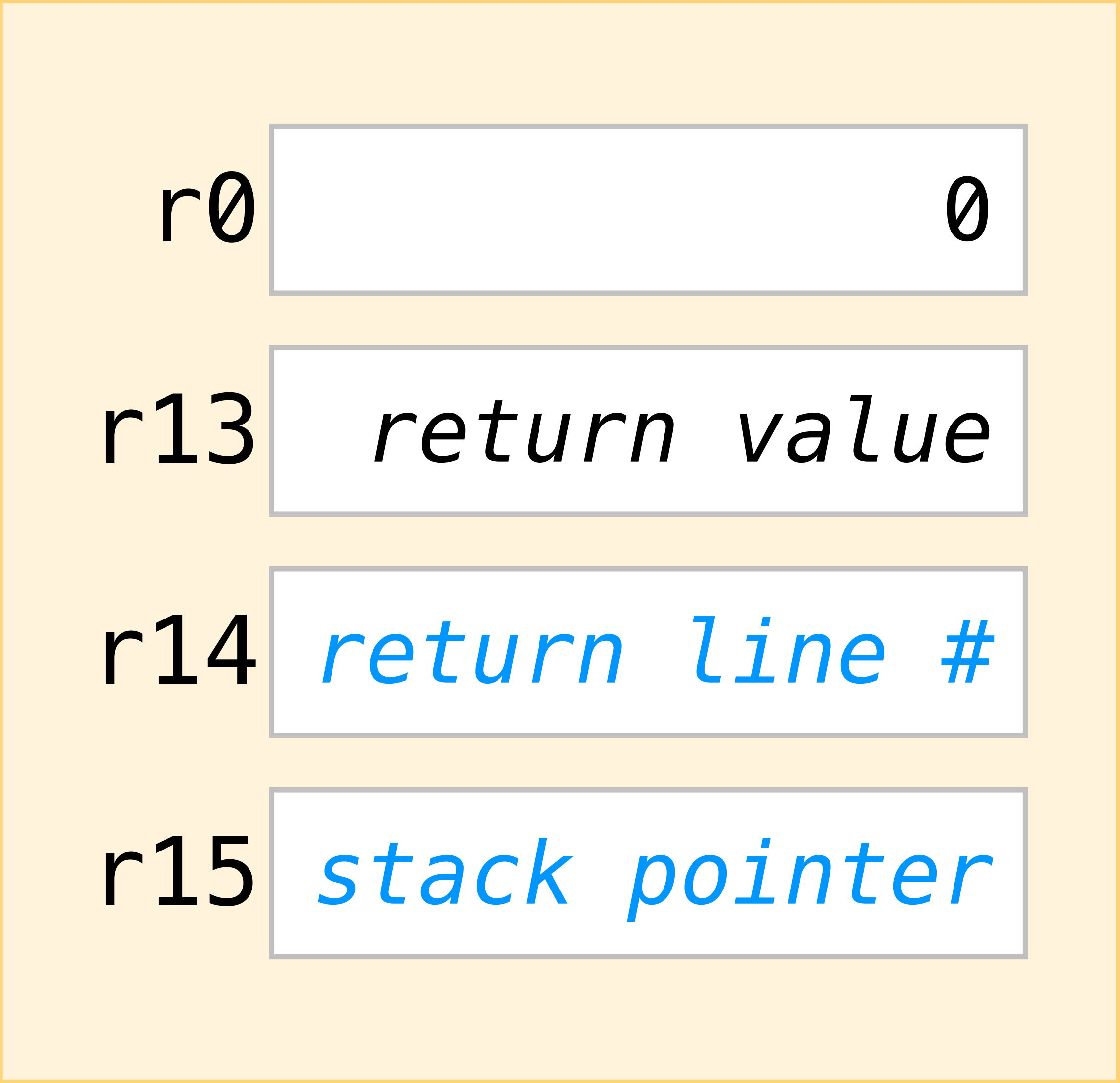
Human programming practices that help us write correct *and* readable programs

`r0` always contains the value `0`

`r13` is for the return value

`r14` is for the return location

`r15` is for the stack pointer



r0 `0`

r13 `return value`

r14 `return line #`

r15 `stack pointer`

Write lots of comments / documentation!
Clearer is better than shorter (but shorter *can be* clearer).

Function calls in Hmmm

treat register values
like local variables!

Caller (outside the function): assume the function writes to every register

Callee (inside the function): assume every register is yours

```
# initialize stack pointer
setn   r15 S
```

caller

```
# save any register value that I'll need later
pushr rN r15


# prepare the arguments by assigning values to registers

calln r14 N      # call the function


# restore all the register values that I saved (LIFO!)
popr  rN r15
```

callee

```
N # function start
# write to registers with gleeful abandon
# if the function should return a value, save it in r13
M jumpr  r14            # return
```

What counts as a problem?

Decision problems on finite, bitstring inputs.

What kinds of **problems** can **computers** solve?

Can sequential logic solve all the problems that a DFA can? How about a Turing Machine?

Yes!

No!

What counts as a computer?