Assembly Language: Loops Revisited; Functions

Computer Science 111
Boston University

Vahid Azadeh-Ranjbar, Ph.D.

based in part on notes from the CS-for-All curriculum developed at Harvey Mudd College

Computing Factorial in Assembly

- Given an input of x, compute the factorial of x (x!)
- We could do it recursively in Assembly
- But we'll hold off on that! Instead, we'll use a loop.

Computing Factorial in Assembly (cont.)

- Begin by planning out how the registers will be used:
  - `r1` will hold the input, and gradually count down to 0
  - `r2` will gradually *accumulate* the result
- For example:

r1	r2
5	5
4	20
3	60
2	120
1	120
0	done!

- How does `r1` change each time?
  What about `r2`?

Computing Factorial in Assembly (cont.)

- Begin by planning out how the registers will be used:
  - `r1` will hold the input, and gradually count down to 0
  - `r2` will gradually *accumulate* the result
- For example:

r1	r2
5	5
4	20
3	60
2	120
1	120
0	done!

- How does `r1` change each time? `r1 = r1 - 1`
  What about `r2`? `r2 = r2 * r1`
- What should `r2`'s initial value be?

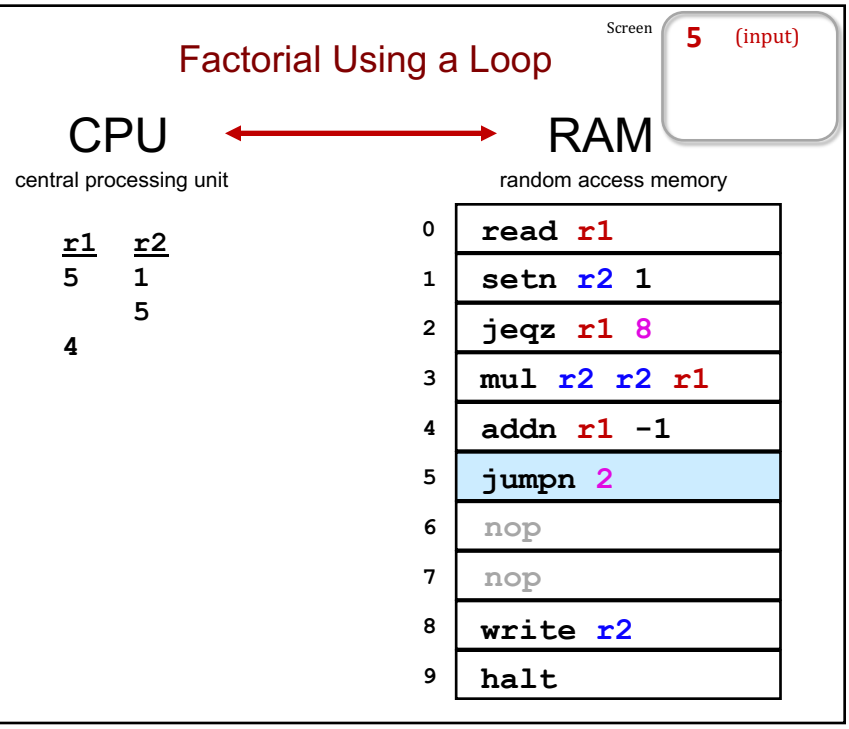
Computing Factorial in Assembly (cont.)

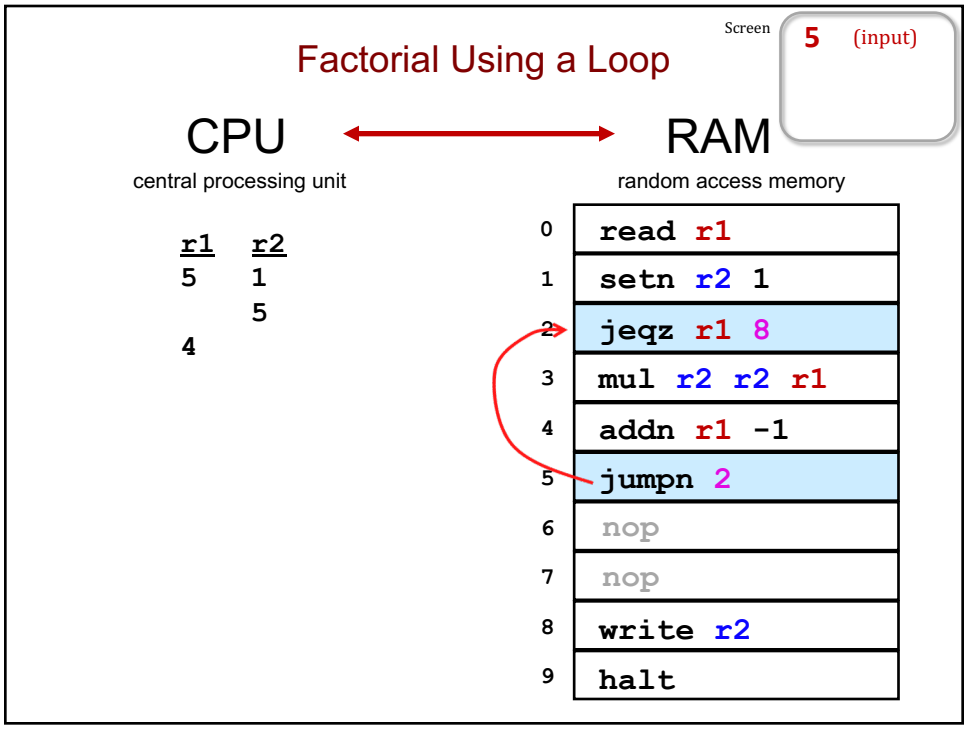
- Begin by planning out how the registers will be used:
  - `r1` will hold the input, and gradually count down to 0
  - `r2` will gradually *accumulate* the result
- For example:

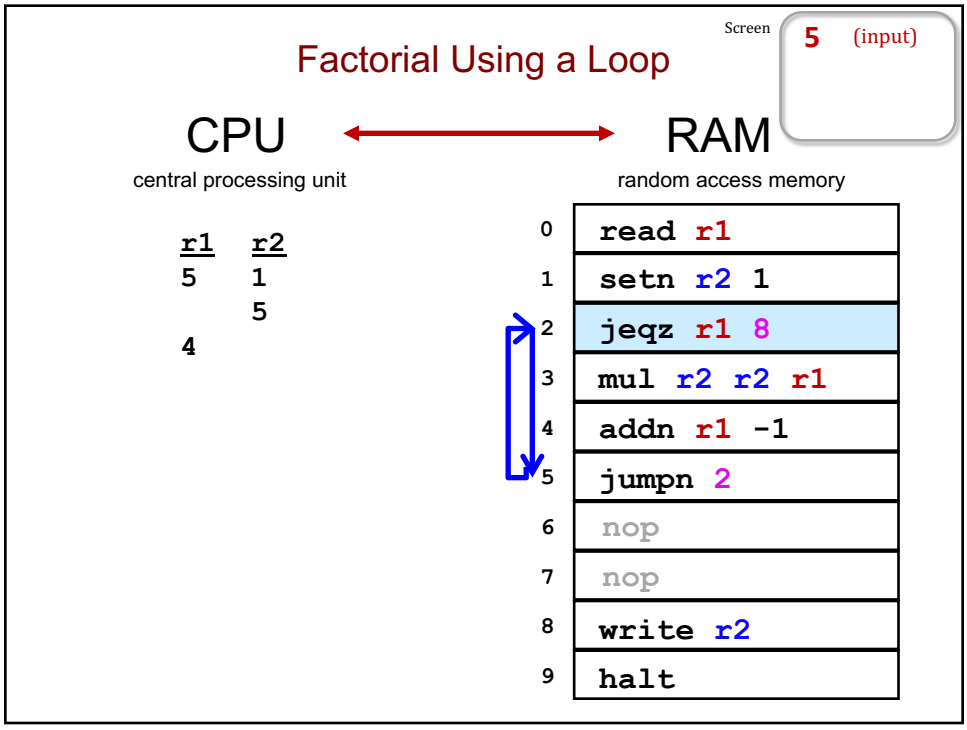
r1	r2
	1
5	5
4	20
3	60
2	120
1	120
0	done!

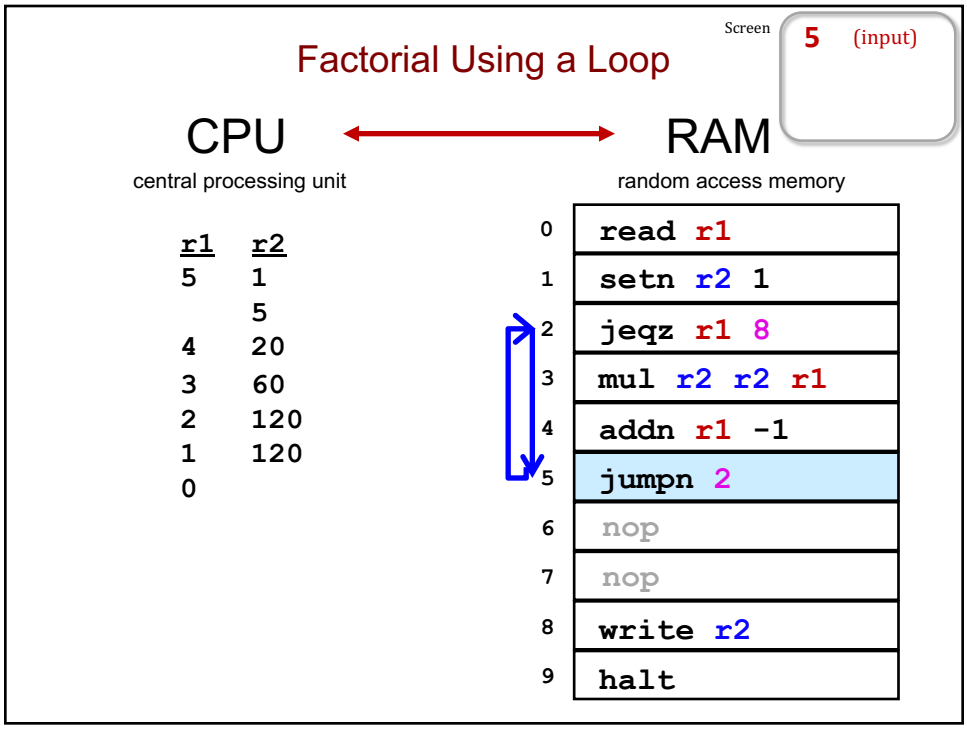
- How does `r1` change each time? `r1 = r1 - 1`
  What about `r2`? `r2 = r2 * r1`
- What should `r2`'s initial value be? `1`

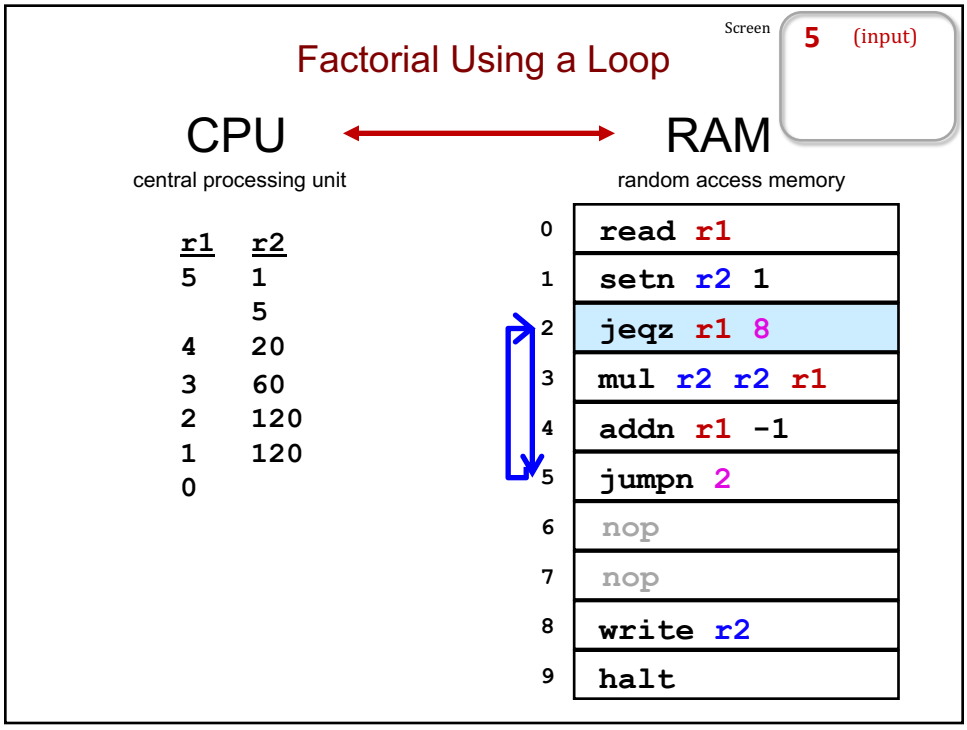
Factorial Using a Loop



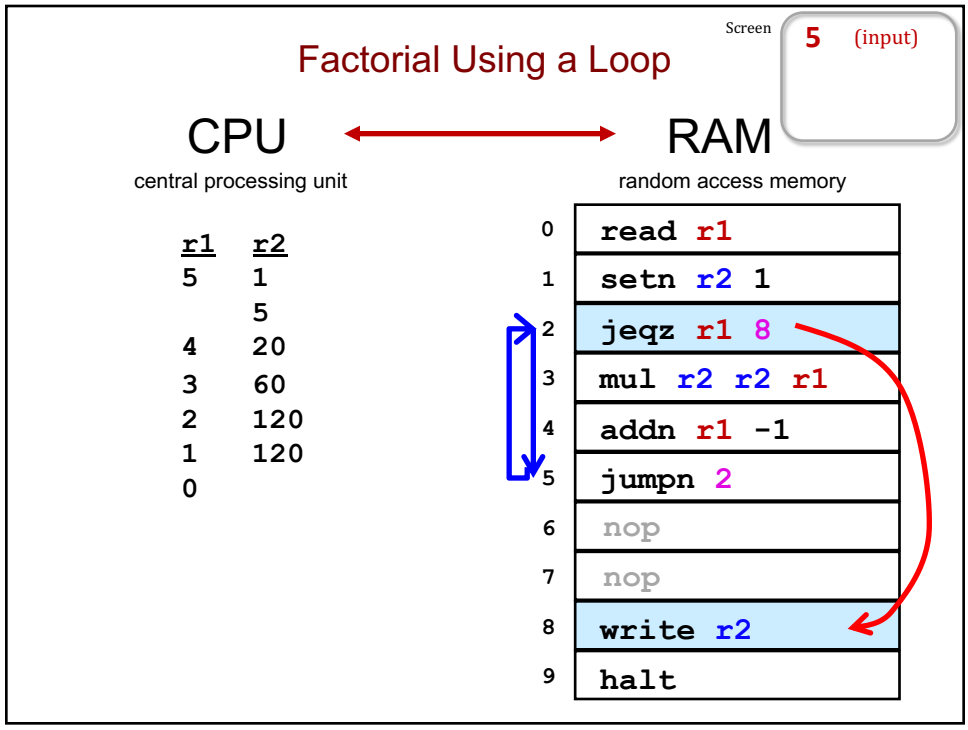

Factorial Using a Loop
Screen
5 (input)
CPU
central processing unit
RAM
random access memory
r1 r2
5 1
5
4
0 read r1
1 setn r2 1
2 jeqz r1 8
3 mul r2 r2 r1
4 addn r1 -1
5 jumpn 2
6 nop
7 nop
8 write r2
9 halt


Factorial Using a Loop
Screen
5 (input)
CPU
central processing unit
RAM
random access memory
r1 r2
5 1
5
4
0 read r1
1 setn r2 1
2 jeqz r1 8
3 mul r2 r2 r1
4 addn r1 -1
5 jumpn 2
6 nop
7 nop
8 write r2
9 halt


Factorial Using a Loop
Screen
5 (input)
CPU
central processing unit
RAM
random access memory
r1 r2
5 1
5
4 20
3 60
2 120
1 120
0
0 read r1
1 setn r2 1
2 jeqz r1 8
3 mul r2 r2 r1
4 addn r1 -1
5 jumpn 2
6 nop
7 nop
8 write r2
9 halt


Factorial Using a Loop
Screen
5 (input)
CPU
central processing unit
RAM
random access memory
r1 r2
5 1
5
4 20
3 60
2 120
1 120
0
0 read r1
1 setn r2 1
2 jeqz r1 8
3 mul r2 r2 r1
4 addn r1 -1
5 jumpn 2
6 nop
7 nop
8 write r2
9 halt

Factorial Using a Loop



From Python...

Consider this Python program:

```
def foo(x):
    y = x*(x-1)
    return y

x = int(input())
y = foo(x)
print(y)
```

- `foo(x)` takes a number x and returns `x*(x - 1)`

```
>>> foo(5)
20
>>> foo(7)
42
```

- the program:
  - gets a number from the user
  - calls `foo()`, passing in that number
  - prints the return value

From Python to Assembly

Consider this Python program:

```
def foo(x):
    y = x*(x-1)
    return y

x = int(input())
y = foo(x)
print(y)
```

Here's the assembly version:

```
0  read r1
1  call r14 4
2  write r13
3  halt
4  copy r13 r1
5  addn r1 -1
6  mul r13 r13 r1
7  jumpr r14
```

- Lines 0-3 correspond to the Python code from the global scope.
- Lines 4-7 correspond to the function `foo`.

From Python to Assembly

Consider this Python program:

```
def foo(x):
    y = x*(x-1)
    return y

x = int(input())
y = foo(x)
print(y)
```

Here's the assembly version:

```
0  read r1
1  call r14 4
2  write r13
3  halt
4  copy r13 r1
5  addn r1 -1
6  mul r13 r13 r1
7  jumpr r14
```


- Use registers for the variables:
  - `r1` for x (the user's input), which is also used by the function
  - `r13` for y (the function's return value)

Making a Function Call in Assembly

Consider this Python program:

```
def foo(x):
    y = x*(x-1)
    return y

x = int(input())
y = foo(x)
print(y)
```

Here's the assembly version:

```
0  read r1
1  call r14 4
2  write r13
3  halt
4  copy r13 r1
5  addn r1 -1
6  mul r13 r13 r1
7  jumpr r14
```

- In assembly, function calls do ***not*** explicitly pass in any inputs.
- Rather, we put the inputs into registers ***before*** the call.
  - there's only one set of registers, so the function can get the inputs from the registers

Making a Function Call in Assembly (cont.)

- To call a function in Hmmm, we use a `call` instruction:

```
call rX L
```

the line number where the function begins

any register from r1-r15; used to store the line number of the instruction *after* the call

```
0  read r1
1  call r14 4
2  write r13
3  halt
4  copy r13 r1
5  addn r1 -1
6  mul r13 r13 r1
7  jumpr r14
```

- The register used by `call` is referred to as the *return address* register.
  - stores the *return address* – the memory address that we will return to after the function completes

Returning From a Function in Assembly

Consider this Python program:

```
def foo(x):
    y = x*(x-1)
    return y

x = int(input())
y = foo(x)
print(y)
```

Here's the assembly version:

```
0  read r1
1  call r14 4
2  write r13
3  halt
4  copy r13 r1
5  addn r1 -1
6  mul r13 r13 r1
7  jumpr r14
```

- In assembly, we do ***not*** explicitly return a value.
- Rather, we put the return value into a register.
  - there's only one set of registers, so the return value can be obtained from the register

Returning From a Function in Assembly (cont.)

- To return from a function in Hmmm, we use a **`call`** instruction:

```
jumpr rX
```

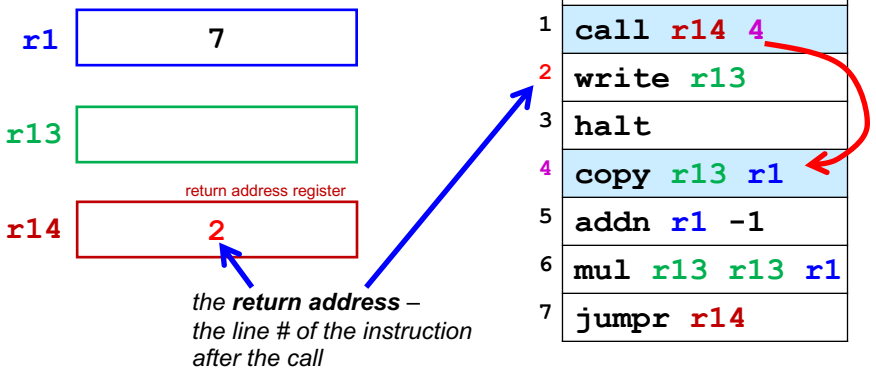
any register from r1-r15;
the return address register --
the same one used by **`call`**!

```
0  read r1
1  call r14 4
2  write r13
3  halt
4  copy r13 r1
5  addn r1 -1
6  mul r13 r13 r1
7  jumpr r14
```

- **`jumpr`** performs an *indirect* jump.
  - it jumps to the line number stored in the specified register – the return address that was stored there by **`call`**

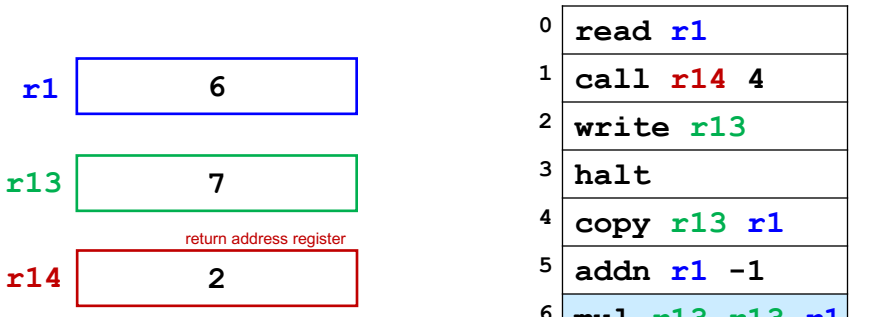
Calling and Returning: `call` and `jumpr`

Screen: 7 (input)



Calling and Returning: `call` and `jumpr`

Screen: 7 (input)



equivalent to:

```
r13 = r13 * r1
      7   * 6
      42
```

Calling and Returning: `call` and `jumpr`

Screen: **7** (input)

r1	6
r13	42
r14 (return address register)	2

equivalent to:

```
r13 = r13 * r1
      7   * 6
      42
```

```
0  read r1
1  call r14 4
2  write r13
3  halt
4  copy r13 r1
5  addn r1 -1
6  mul r13 r13 r1
7  jumpr r14
```

Calling and Returning: `call` and `jumpr`

Screen: **7** (input)

r1	6
r13	42
r14 (return address register)	2

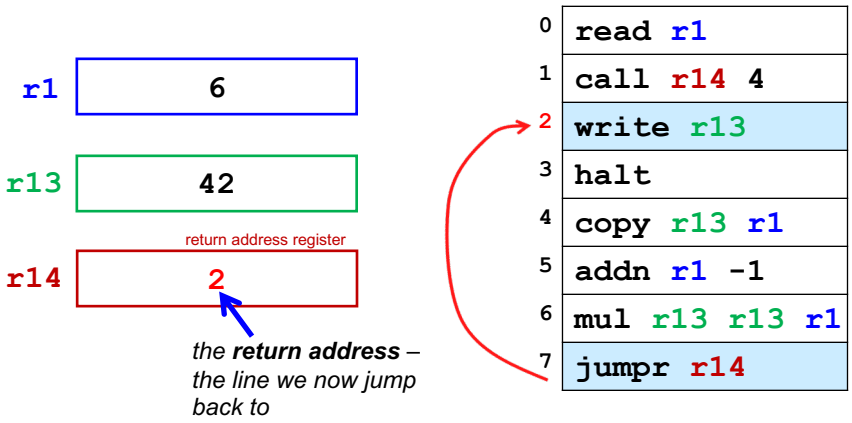
*the **return address** – the line we now jump back to*

```
0  read r1
1  call r14 4
2  write r13
3  halt
4  copy r13 r1
5  addn r1 -1
6  mul r13 r13 r1
7  jumpr r14
```

Calling and Returning: `call` and `jumpr`

Screen

7 (input)

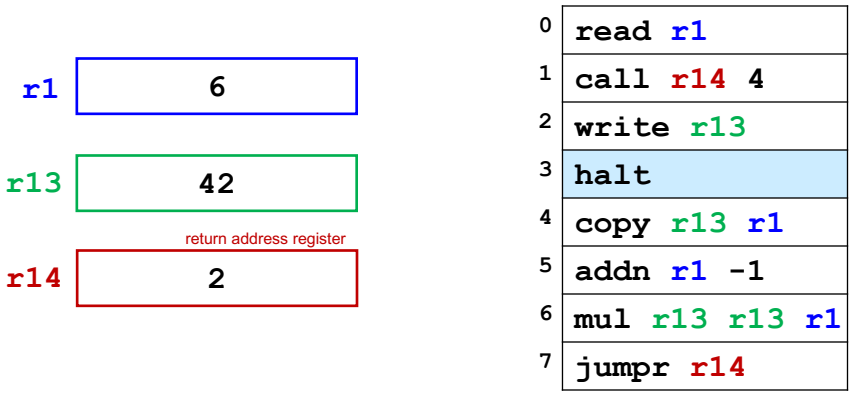


Calling and Returning: `call` and `jumpr`

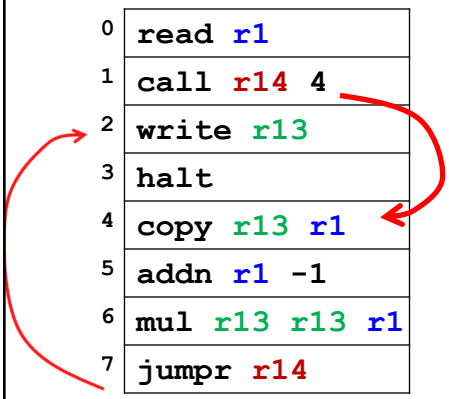
Screen

7 (input)

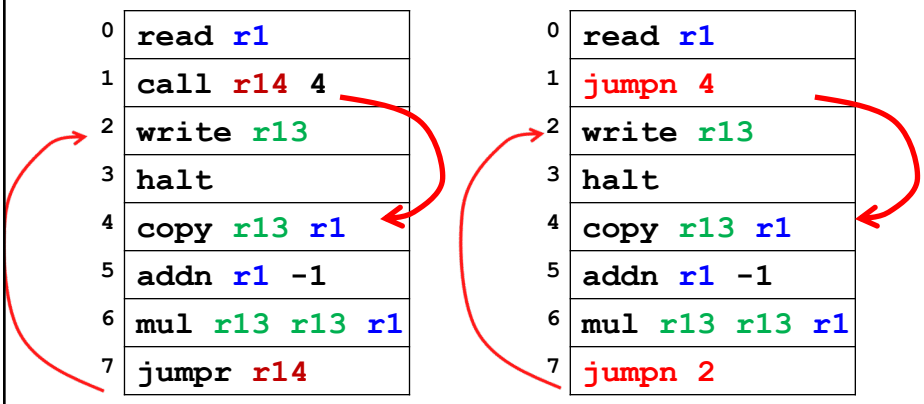
42 (output)



Why Do We Need `call` and `jumpr`?



Why Do We Need `call` and `jumpr`?



Unconditional jumps (**`jumpn`**) could be used in place of **`call/jumpr`** here, because we only call the function once.

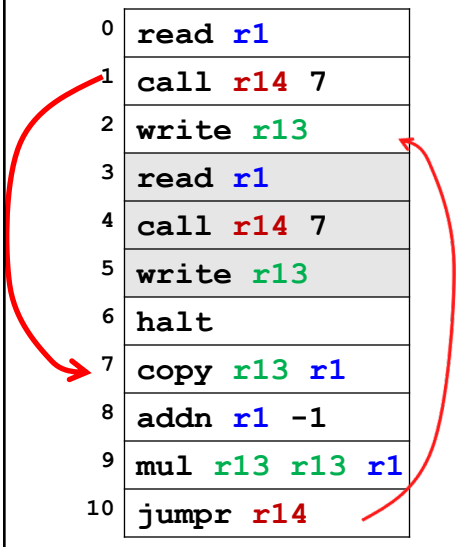
Why Do We Need `call` and `jumpr`?

```
0  read r1
1  call r14 7
2  write r13
3  read r1
4  call r14 7
5  write r13
6  halt
7  copy r13 r1
8  addn r1 -1
9  mul r13 r13 r1
10 jumpr r14
```

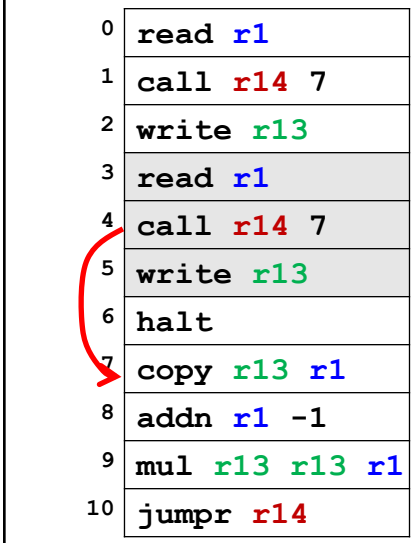
Why Do We Need `call` and `jumpr`?

```
0  read r1
1  call r14 7
2  write r13
3  read r1
4  call r14 7
5  write r13
6  halt
7  copy r13 r1
8  addn r1 -1
9  mul r13 r13 r1
10 jumpr r14
```

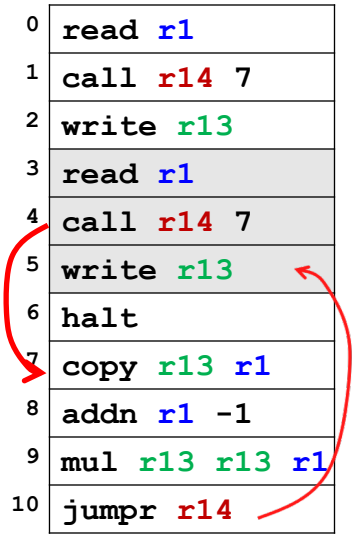
Why Do We Need `call` and `jumpr`?



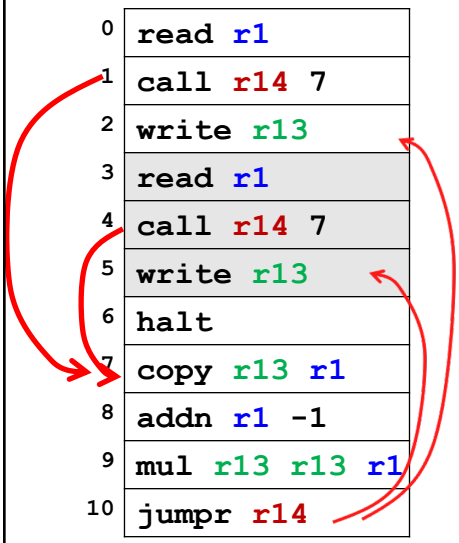
Why Do We Need `call` and `jumpr`?



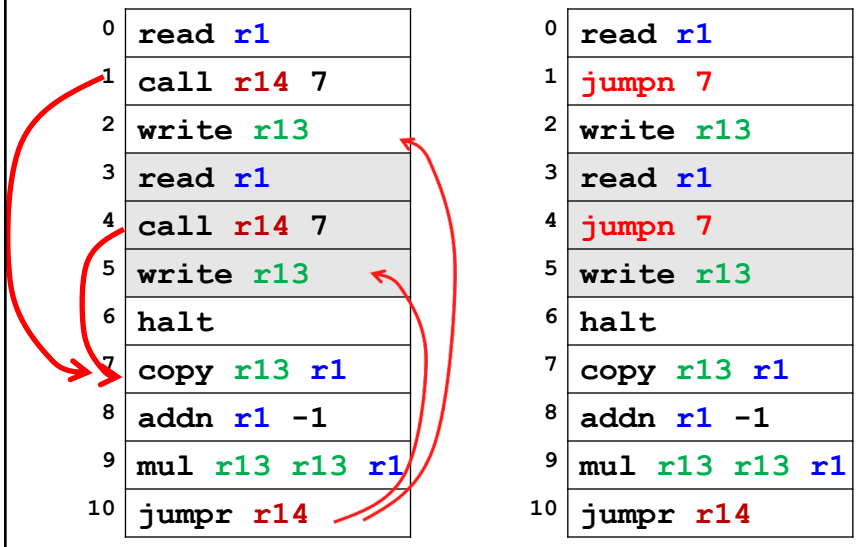
Why Do We Need `call` and `jumpr`?



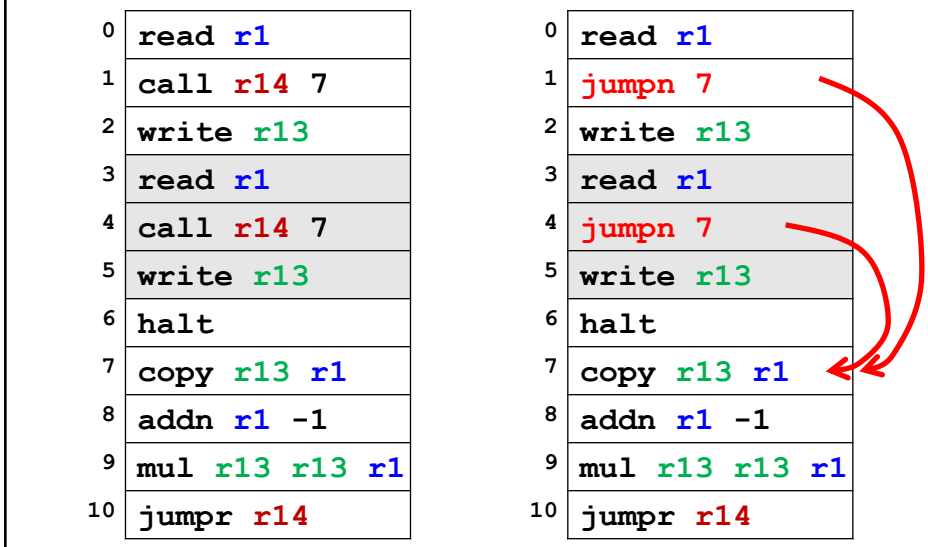
Why Do We Need `call` and `jumpr`?



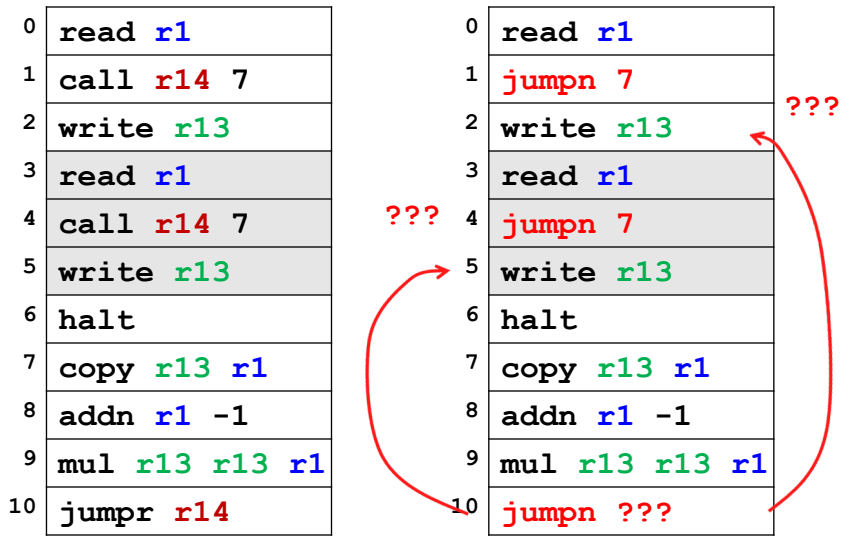
Why Do We Need `call` and `jumpr`?



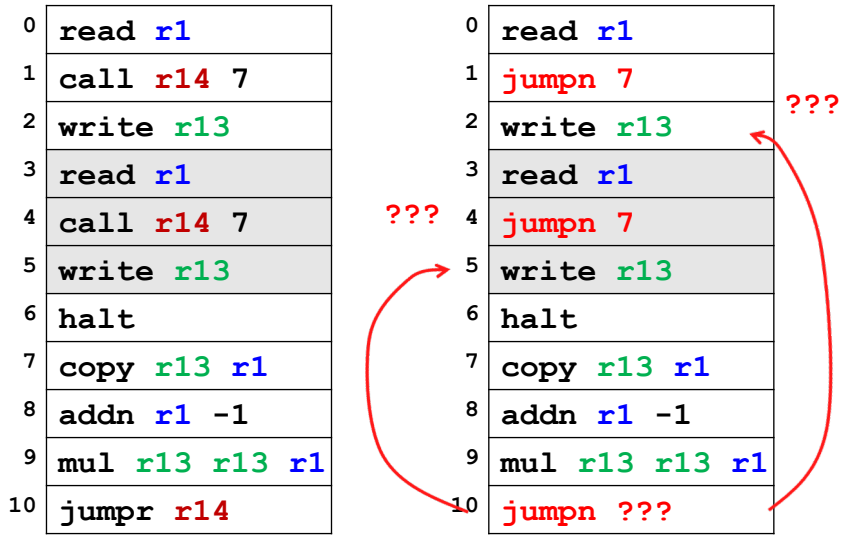
Why Do We Need `call` and `jumpr`?



Why Do We Need `call` and `jumpr`?



Why Do We Need `call` and `jumpr`?



Here, we call the function twice, so we need call/jumpr so that line 10 will know whether to go back to line 2 or 5.

For the inputs at right, what are the final values of r13 and r14?

Screen
4 (1st input)
8 (2nd input)

	r13	r14
A.	8	2
B.	4	3
C.	8	3
D.	4	5
E.	8	5

- Which lines make up the function?
- What Python function is this?

```
0  read r1
1  read r2
2  call r14 5
3  write r13
4  halt
5  sub r3 r1 r2
6  jltz r3 9
7  copy r13 r1
8  jumpr r14
9  copy r13 r2
10 jumpr r14
```

For the inputs at right, what are the final values of r13 and r14?

Screen
4 (1st input)
8 (2nd input)

	r13	r14
A.	8	2
B.	4	3
C.	8	3
D.	4	5
E.	8	5

return address: where to go after the function is done

- Which lines make up the function?
- What Python function is this?

```
0  read r1
1  read r2
2  call r14 5
3  write r13
4  halt
5  sub r3 r1 r2
6  jltz r3 9
7  copy r13 r1
8  jumpr r14
9  copy r13 r2
10 jumpr r14
```

For the inputs at right, what are the final values of r13 and r14?

Screen
4 (1st input)
8 (2nd input)

	r13	r14
A.	8	2
B.	4	3
C.	8	3
D.	4	5
E.	8	5

return value

return address: where to go after the function is done

```
0  read r1
1  read r2
2  call r14 5
3  write r13
4  halt
5  sub r3 r1 r2
6  jltz r3 9
7  copy r13 r1
8  jumpr r14
9  copy r13 r2
10 jumpr r14
```

- Which lines make up the function?
- What Python function is this?

For the inputs at right, what are the final values of r13 and r14?

Screen
4 (1st input)
8 (2nd input)

	r13	r14
A.	8	2
B.	4	3
C.	8	3
D.	4	5
E.	8	5

return value

return address: where to go after the function is done

```
0  read r1
1  read r2
2  call r14 5
3  write r13
4  halt
5  sub r3 r1 r2
6  jltz r3 9
7  copy r13 r1
8  jumpr r14
9  copy r13 r2
10 jumpr r14
```

- Which lines make up the function?
- What Python function is this?

For the inputs at right, what are the final values of r13 and r14?

Screen
4 (1st input)
8 (2nd input)

	r13	r14
A.	8	2
B.	4	3
C.	8	3
D.	4	5
E.	8	5

return value — *return address:* where to go after the function is done

```
0  read r1
1  read r2
2  call r14 5
3  write r13
4  halt
5  sub r3 r1 r2
6  jltz r3 9
7  copy r13 r1
8  jumpr r14
9  copy r13 r2
10 jumpr r14
```

- Which lines make up the function?
  lines 5-10
- What Python function is this?

For the inputs at right, what are the final values of r13 and r14?

Screen
4 (1st input)
8 (2nd input)

	r13	r14
A.	8	2
B.	4	3
C.	8	3
D.	4	5
E.	8	5

return value — *return address:* where to go after the function is done

```
0  read r1
1  read r2
2  call r14 5
3  write r13
4  halt
5  sub r3 r1 r2
6  jltz r3 9
7  copy r13 r1
8  jumpr r14
9  copy r13 r2
10 jumpr r14
```

- Which lines make up the function?
  lines 5-10

- What Python function is this?
  max()

The Need for the Stack

```
def foo(x):
    y = x*(x-1)
    return y

x = int(input())
y = foo(x)
print(y)
```

```
0  read r1
1  call r14 4
2  write r13
3  halt
4  copy r13 r1
5  addn r1 -1
6  mul r13 r13 r1
7  jumpr r14
```

The Need for the Stack

```
def foo(x):
    y = x*(x-1)
    return y

x = int(input())
y = foo(x)
y = y + x
print(y)
```

```
0  read r1
1  call r14 4
2  write r13
3  halt
4  copy r13 r1
5  addn r1 -1
```

What if we add this line?

We want to add the user's input **x** to the function's return value **y** before printing **y**.

The Need for the Stack

```
def foo(x):
    y = x*(x-1)
    return y

x = int(input())
y = foo(x)
y = y + x
print(y)
```

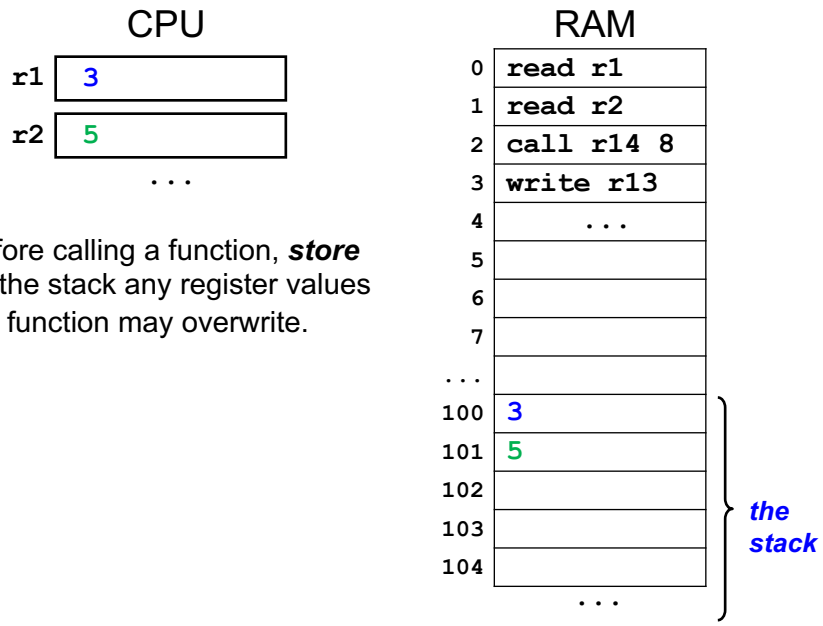
```
0   read r1
1   call r14 4
1.5 add r13 r13 r1
2   write r13
3   halt
4   copy r13 r1
5   addn r1 -1
6   mul r13 r13 r1
7   jumpr r14
```

doesn't work!

the user's input gets changed

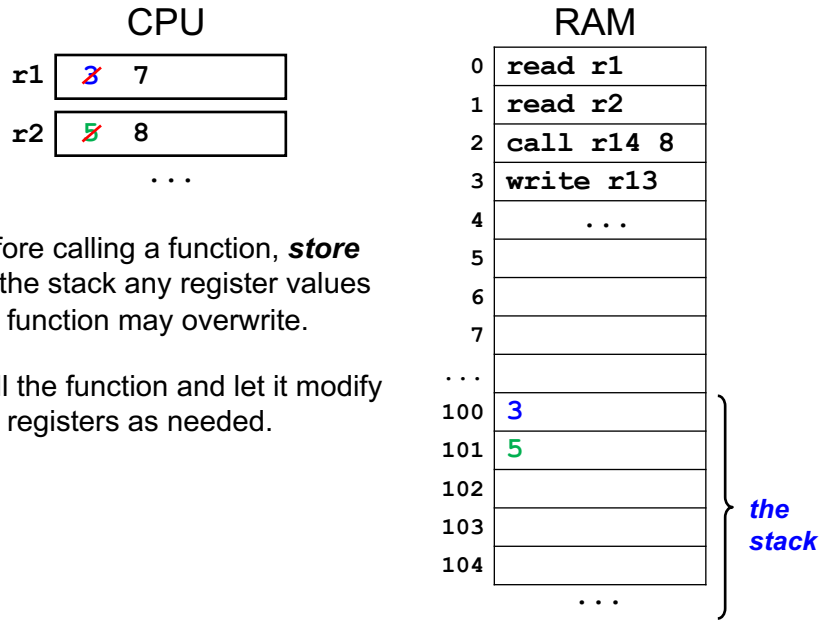
- There's only one set of registers – shared by all lines of code!
  - in line 5, the function changes the value of **r1**
  - line 1.5 gets the changed value

Solution: Store Things on the Stack



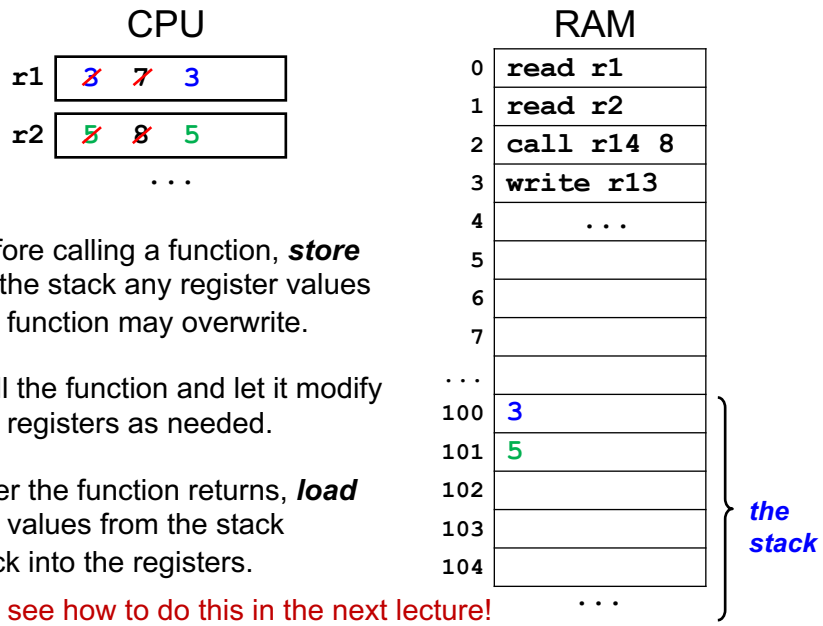
- Before calling a function, ***store*** on the stack any register values the function may overwrite.

Solution: Store Things on the Stack



- Before calling a function, ***store*** on the stack any register values the function may overwrite.
- Call the function and let it modify the registers as needed.

Solution: Store Things on the Stack



- Before calling a function, ***store*** on the stack any register values the function may overwrite.
- Call the function and let it modify the registers as needed.
- After the function returns, ***load*** the values from the stack back into the registers.
- We'll see how to do this in the next lecture!