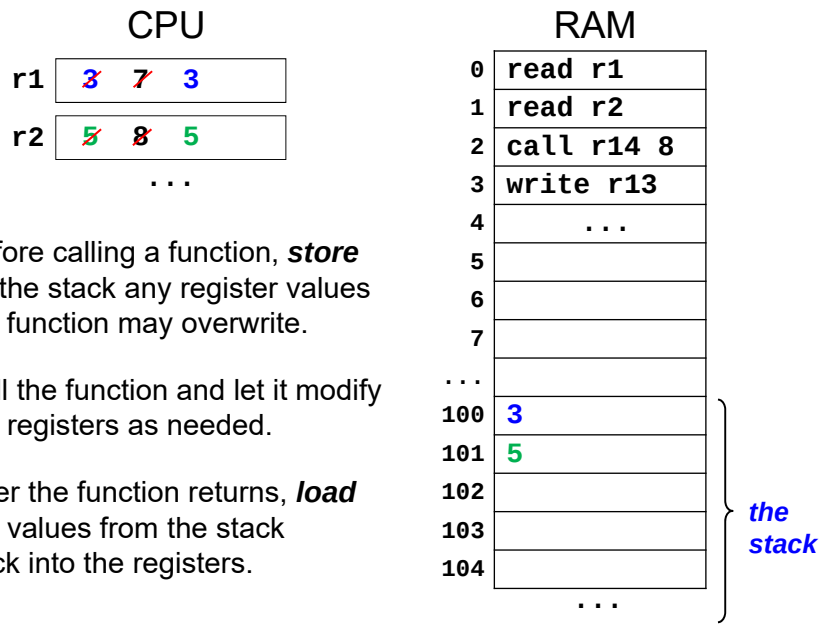
Optional: Recursion in Assembly

Computer Science 111
Boston University

David G. Sullivan, Ph.D.

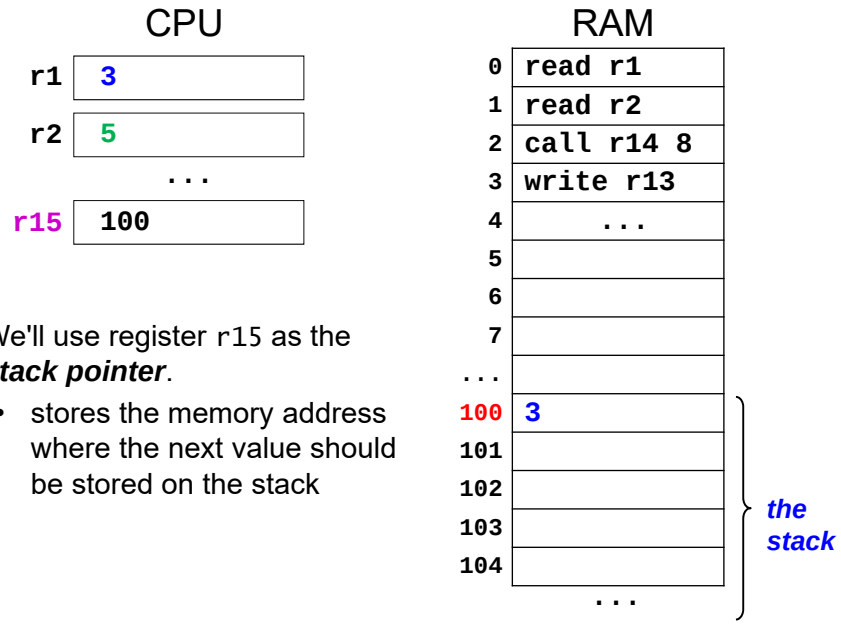
based in part on notes from the CS-for-All curriculum developed at Harvey Mudd College

Recall: Storing 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.

Recall: The Stack Pointer



- We'll use register `r15` as the ***stack pointer***.
  - stores the memory address where the next value should be stored on the stack

Using the Stack: **`storer`** and **`loadr`**

```
0   read r1
1   setn r15 100
2   storer r1 r15
3   call r14 8
4   loadr r1 r15
5   add r13 r13 r1
6   write r13
7   halt
8   copy r13 r1
9   addn r1 -1
10  mul r13 r13 r1
11  jumpr r14
```

```
# initialize the stack pointer
# store r1's value on the stack
  (in the memory address found
   in r15 – not in r15 itself)
# load back r1's value from the
  stack (from the memory address
  found in r15 – not from r15 itself)
```

Computing Factorial Recursively

Python

```
def fac(x):
   if x == 0:
      return 1
   else:
      fac_rest = fac(x - 1)
      return x * fac_rest

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

Assembly

```
00 read r1
01 setn r15 100
02 call r14 06
03 write r13
04 halt
05 nop
06 jnez r1 09
07 setn r13 1
08 jumpr r14
09 addn r15 1
10 storer r1 r15
11 addn r15 1
12 storer r14 r15
13 addn r1 -1
14 call r14 06
15 loadr r14 r15
16 addn r15 -1
17 loadr r1 r15
18 addn r15 -1
19 mul r13 r13 r1
20 jumpr r14
```

Computing Factorial Recursively

Python

```
def fac(x):
   if x == 0:
      return 1
   else:
      fac_rest = fac(x - 1)
      return x * fac_rest

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

Assembly

```
00 read r1
01 setn r15 100
02 call r14 06
03 write r13
04 halt
05 nop
06 jnez r1 09
07 setn r13 1
08 jumpr r14
09 addn r15 1
10 storer r1 r15
11 addn r15 1
12 storer r14 r15
13 addn r1 -1
14 call r14 06
15 loadr r14 r15
16 addn r15 -1
17 loadr r1 r15
18 addn r15 -1
19 mul r13 r13 r1
20 jumpr r14
```

Computing Factorial Recursively

Python

```
def fac(x):
    if x == 0:
        return 1
    else:
        fac_rest = fac(x - 1)
        return x * fac_rest

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

Assembly

```
00 read r1
01 setn r15 100
02 call r14 06
03 write r13
04 halt
05 nop
06 jnez r1 09
07 setn r13 1
08 jumpr r14
09 addn r15 1
10 storer r1 r15
11 addn r15 1
12 storer r14 r15
13 addn r1 -1
14 call r14 06
15 loadr r14 r15
16 addn r15 -1
17 loadr r1 r15
18 addn r15 -1
19 mul r13 r13 r1
20 jumpr r14
```

factorial function

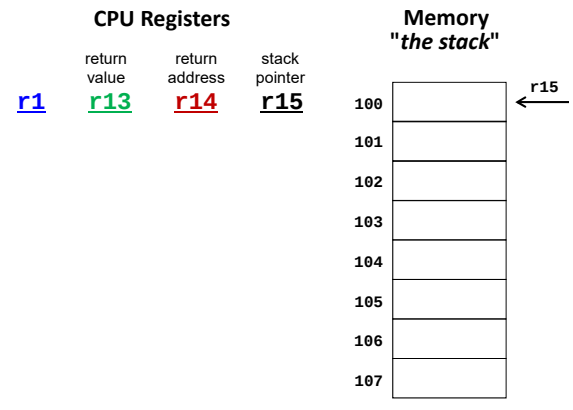
store

recursive call

load

How many lines execute for an input of 0?

```
00 read r1
01 setn r15 100
02 call r14 06
03 write r13
04 halt
05 nop
06 jnez r1 09
07 setn r13 1
08 jumpr r14
09 addn r15 1
10 storer r1 r15
11 addn r15 1
12 storer r14 r15
13 addn r1 -1
14 call r14 06
15 loadr r14 r15
16 addn r15 -1
17 loadr r1 r15
18 addn r15 -1
19 mul r13 r13 r1
20 jumpr r14
```



A. 5

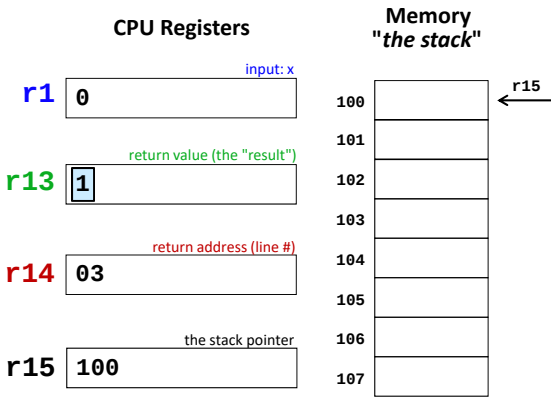
B. 8

C. 10

D. more than 10

How many lines execute for an input of 0?

```
00 read r1
01 setn r15 100
02 call r14 06
03 write r13
04 halt
05 nop
06 jnez r1 09
07 setn r13 1
08 jumpr r14
09 addn r15 1
10 storer r1 r15
11 addn r15 1
12 storer r14 r15
13 addn r1 -1
14 call r14 06
15 loadr r14 r15
16 addn r15 -1
17 loadr r1 r15
18 addn r15 -1
19 mul r13 r13 r1
20 jumpr r14
```

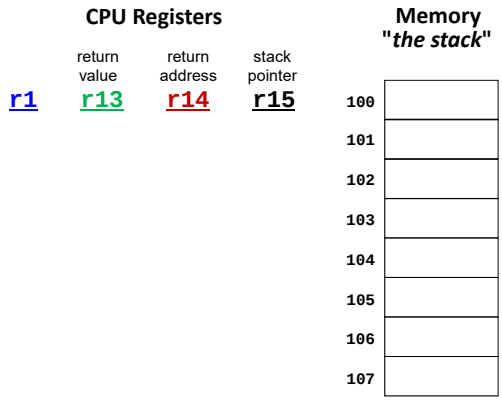


A. 5

B. **8** -- lines 0, 1, 2, 6, 7, 8, 3, 4

Trace what happens for an input of 3...

```
00 read r1
01 setn r15 100
02 call r14 06
03 write r13
04 halt
05 nop
06 jnez r1 09
07 setn r13 1
08 jumpr r14
09 addn r15 1
10 storer r1 r15
11 addn r15 1
12 storer r14 r15
13 addn r1 -1
14 call r14 06
15 loadr r14 r15
16 addn r15 -1
17 loadr r1 r15
18 addn r15 -1
19 mul r13 r13 r1
20 jumpr r14
```



Trace what happens for an input of 3...

```
00 read r1
01 setn r15 100
02 call r14 06
03 write r13
04 halt
05 nop
06 jnez r1 09
07 setn r13 1
08 jumpr r14
09 addn r15 1
10 storer r1 r15
11 addn r15 1
12 storer r14 r15
13 addn r1 -1
14 call r14 06
15 loadr r14 r15
16 addn r15 -1
17 loadr r1 r15
18 addn r15 -1
19 mul r13 r13 r1
20 jumpr r14
```

CPU Registers

	r1	return value r13	return address r14	stack pointer r15
	3			100
			03	101
				102
	2		15	103
				104
	1		15	105
				106
	0		15	
		1	15	105
	1			104
		1	15	103
	2			102
		2	03	101
	3			100
		6		

Memory "*the stack*"