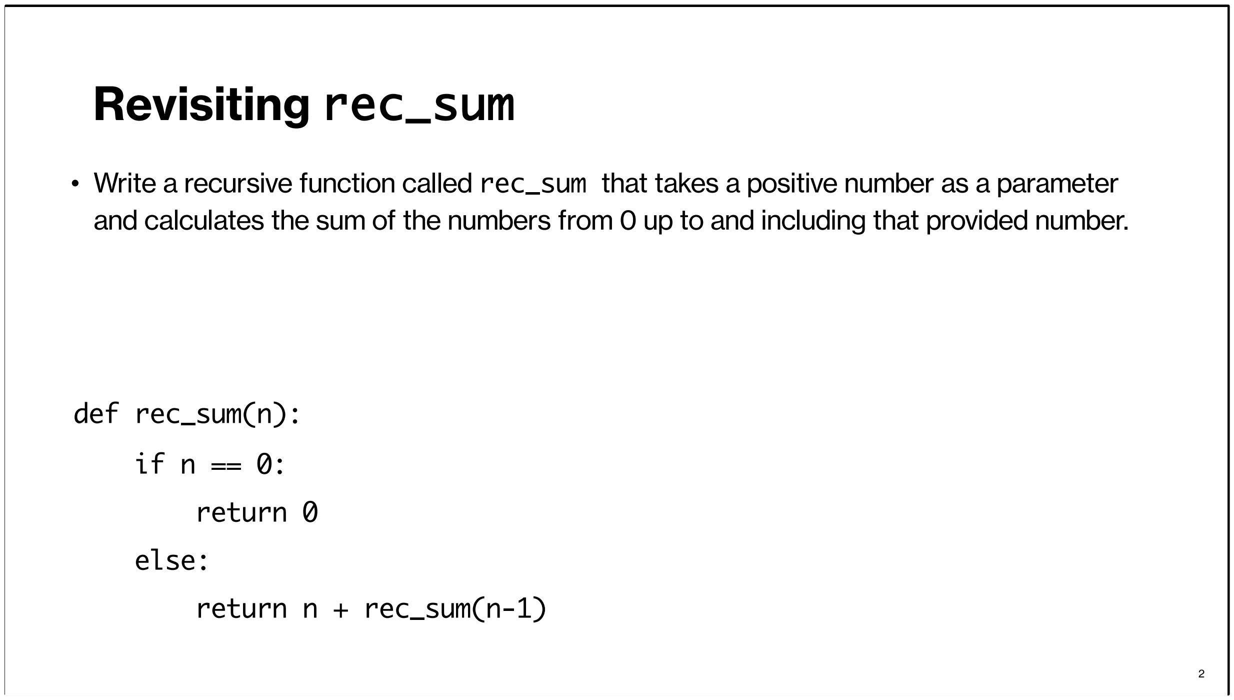
Mathematical Foundations

More Recursion

CS51 – Spring 2026

https://commons.wikimedia.org/w/index.php?curid=2747463

Today we will revisit recursion using the cs51 machine. We will also introduce the turtle module which you will use to draw (recursive!) shapes in your next assignment.

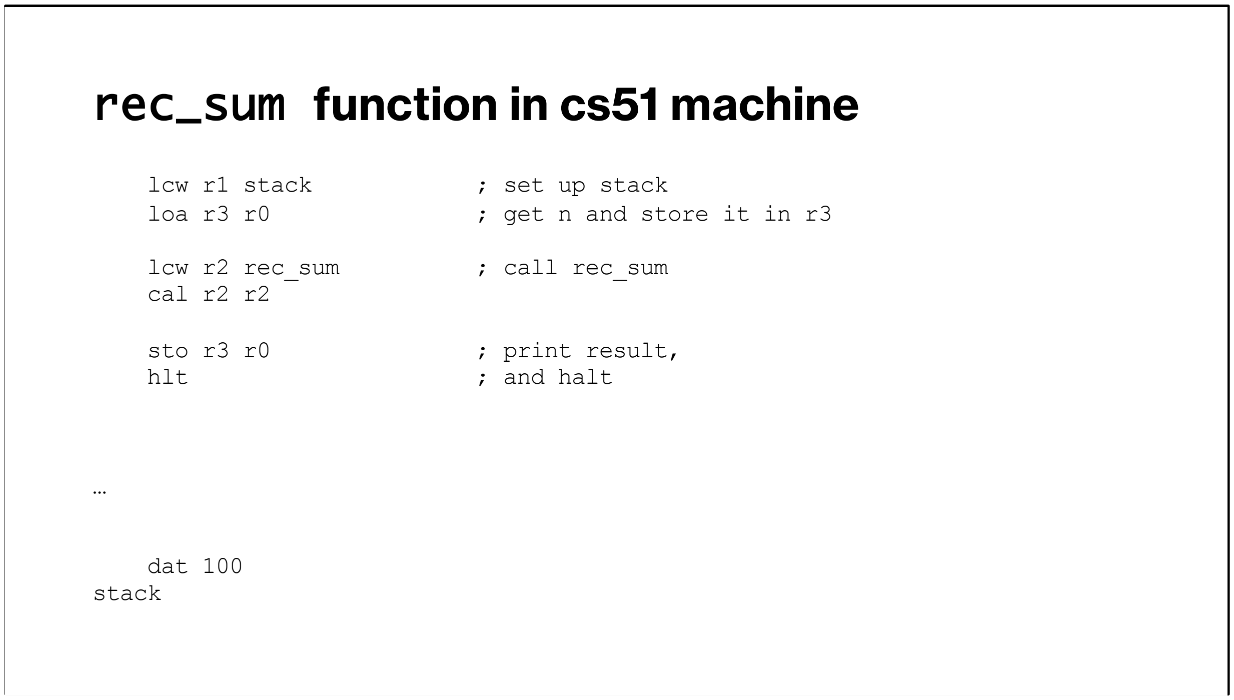


Revisiting `rec_sum`

- Write a recursive function called `rec_sum` that takes a positive number as a parameter and calculates the sum of the numbers from 0 up to and including that provided number.

```
def rec_sum(n):
    if n == 0:
        return 0
    else:
        return n + rec_sum(n-1)
```

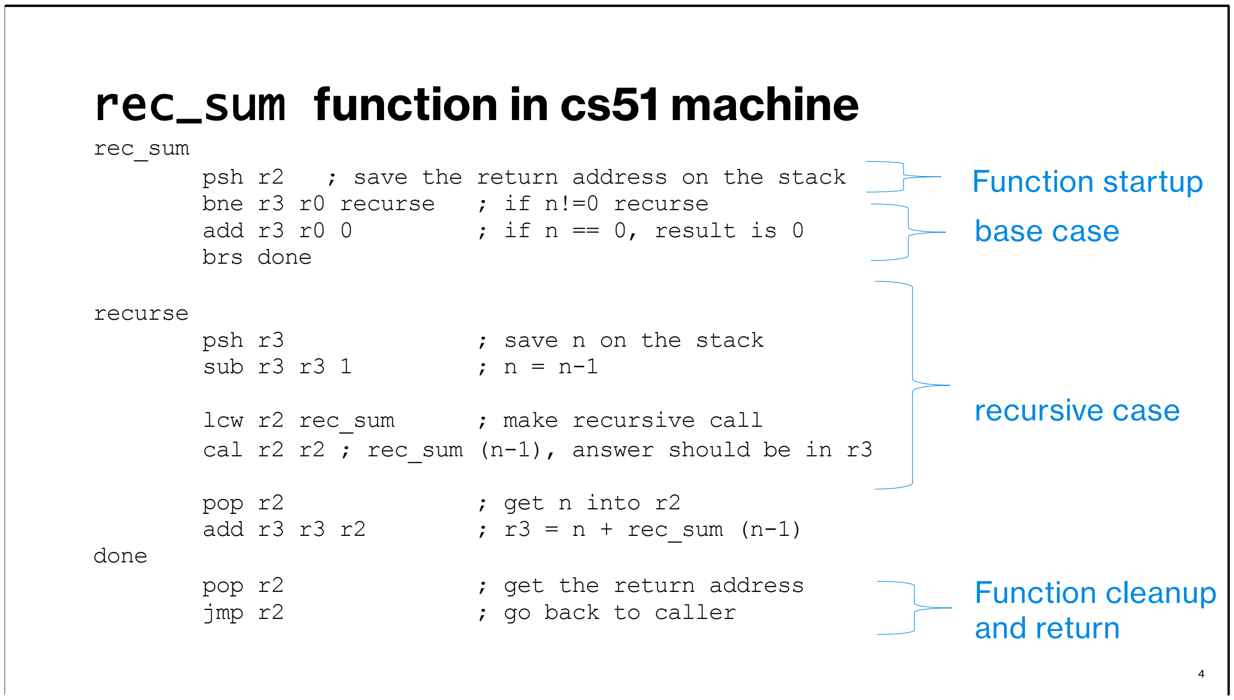
If you remember from our last lesson, we wrote together a recursive function that takes a positive integer as a parameter and calculates the sum of the numbers from 0 up to and including that provided number. We walked through the Python code which captures this idea.



rec_sum function in cs51 machine

```
    lcw r1 stack            ; set up stack
    loa r3 r0               ; get n and store it in r3

    lcw r2 rec_sum          ; call rec_sum
    cal r2 r2

    sto r3 r0               ; print result,
    hlt                     ; and halt



…


    dat 100
stack
```

Today we will work toward understanding how a recursive function would look like in assembly. For that, we will use our trusted cs51 machine. Here's a bit of code that will help us get things set up. We start by setting up the stack. We will prompt the user for the number n they want to calculate the sum from 0 to n. We will then call a rec_sum function which we'll see in detail. Eventually this rec_sum function should give us the answer in r3 which we will print and halt our program.



rec_sum function in cs51 machine

```
rec_sum
        psh r2   ; save the return address on the stack
        bne r3 r0 recurse   ; if n!=0 recurse
        add r3 r0 0         ; if n == 0, result is 0
        brs done

recurse
        psh r3              ; save n on the stack
        sub r3 r3 1         ; n = n-1

        lcw r2 rec_sum      ; make recursive call
        cal r2 r2 ; rec_sum (n-1), answer should be in r3

        pop r2              ; get n into r2
        add r3 r3 r2        ; r3 = n + rec_sum (n-1)
done
        pop r2              ; get the return address
        jmp r2              ; go back to caller
```

Function startup

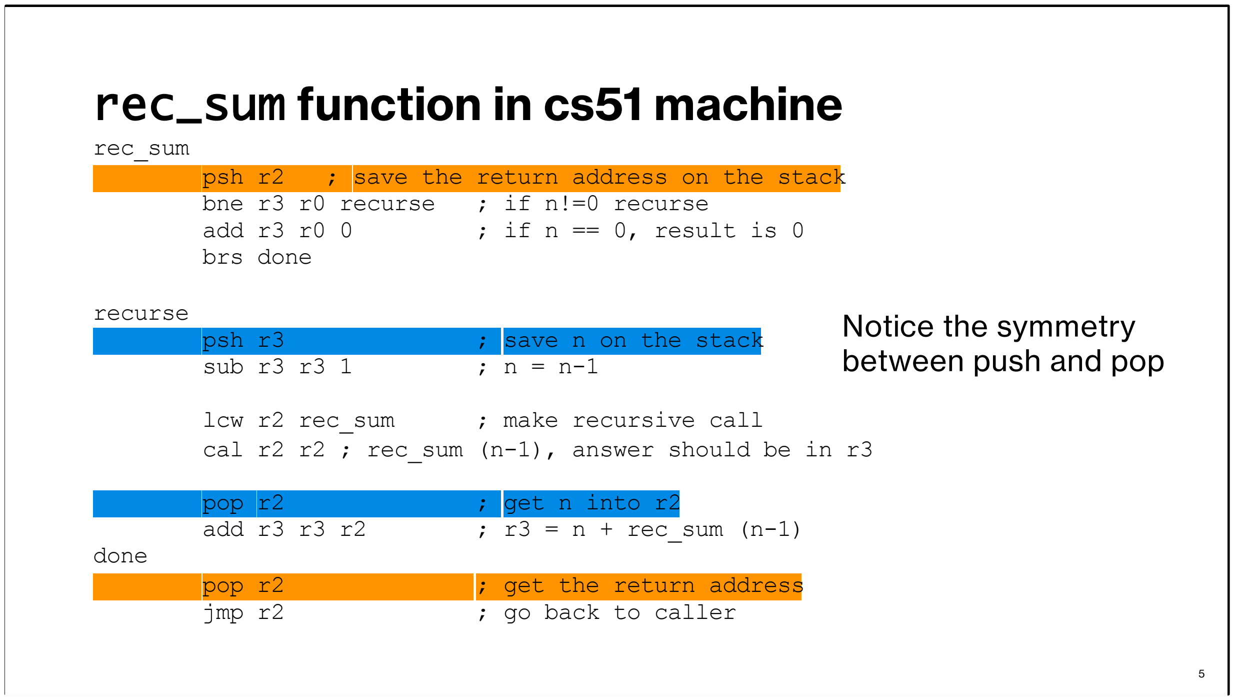
base case

recursive case

Function cleanup and return

4

This is the rec_sum function implemented in assembly. As always, the first thing we do in a function is to push the return address on the stack. Symmetrically, the last thing we will do is to pop and jump to that return address. I have annotated the base case and recursive case. Our base case is satisifed when n==0 where we just return 0. Otherwise, we would need to recurse, which we do in the recurse block of code which we will soon see in detail.



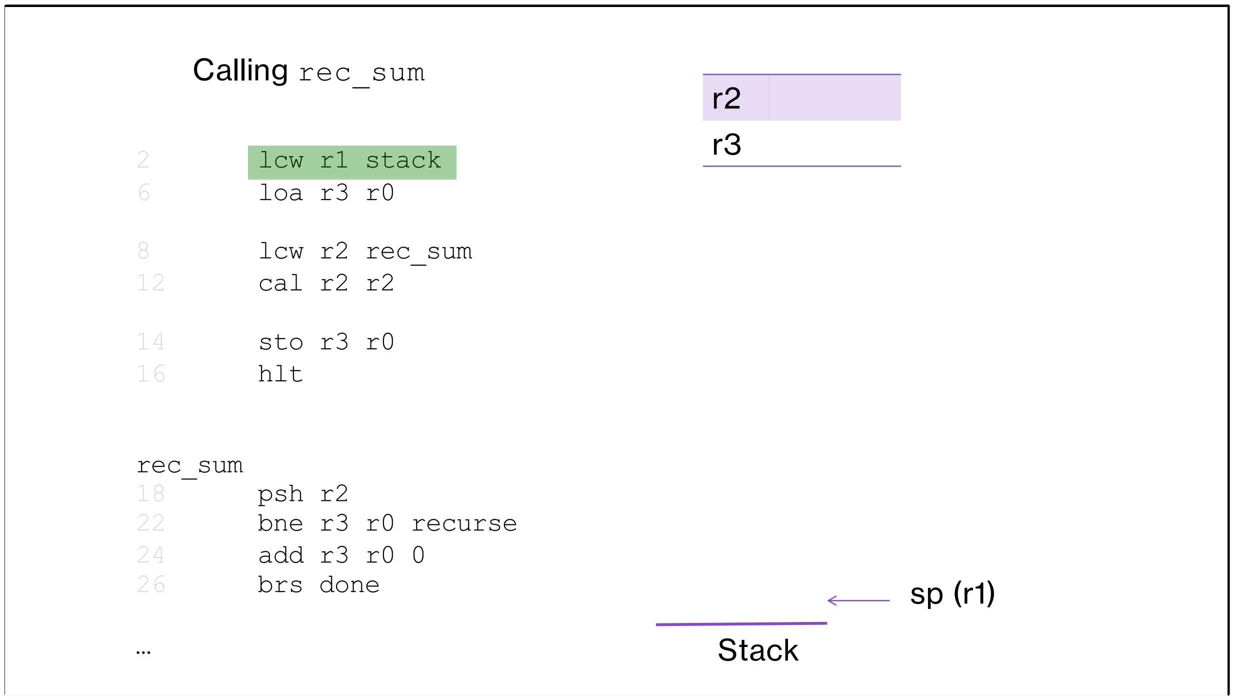
`rec_sum` function in cs51 machine

```
rec_sum
        psh r2    ; save the return address on the stack
        bne r3 r0 recurse   ; if n!=0 recurse
        add r3 r0 0         ; if n == 0, result is 0
        brs done

recurse
        psh r3              ; save n on the stack
        sub r3 r3 1         ; n = n-1

        lcw r2 rec_sum      ; make recursive call
        cal r2 r2 ; rec_sum (n-1), answer should be in r3

        pop r2              ; get n into r2
        add r3 r3 r2        ; r3 = n + rec_sum (n-1)
done
        pop r2              ; get the return address
        jmp r2              ; go back to caller
```

Notice the symmetry between push and pop

5

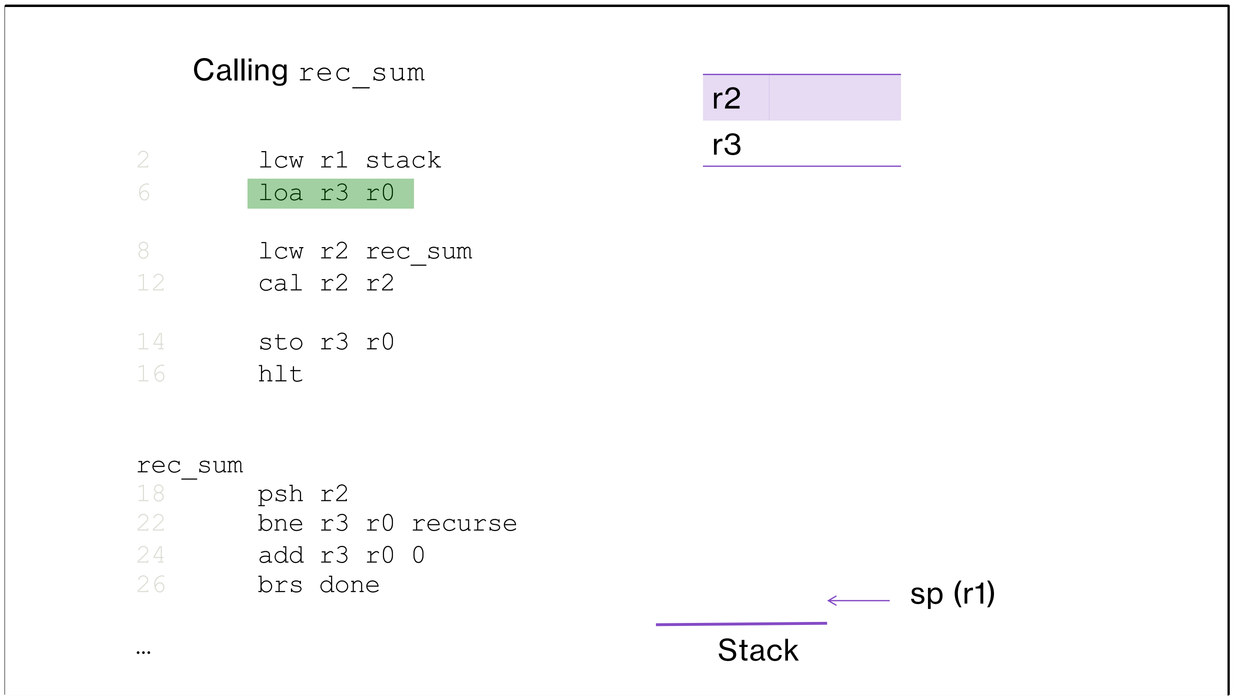
Notice how the push and pop are symmetrically. Specifically, I have color-coded the two pairs so you understand which one maps to which.



Calling `rec_sum`

```
2       lcw r1 stack
6       loa r3 r0

8       lcw r2 rec_sum
12      cal r2 r2

14      sto r3 r0
16      hlt


rec_sum
18      psh r2
22      bne r3 r0 recurse
24      add r3 r0 0
26      brs done

...
```

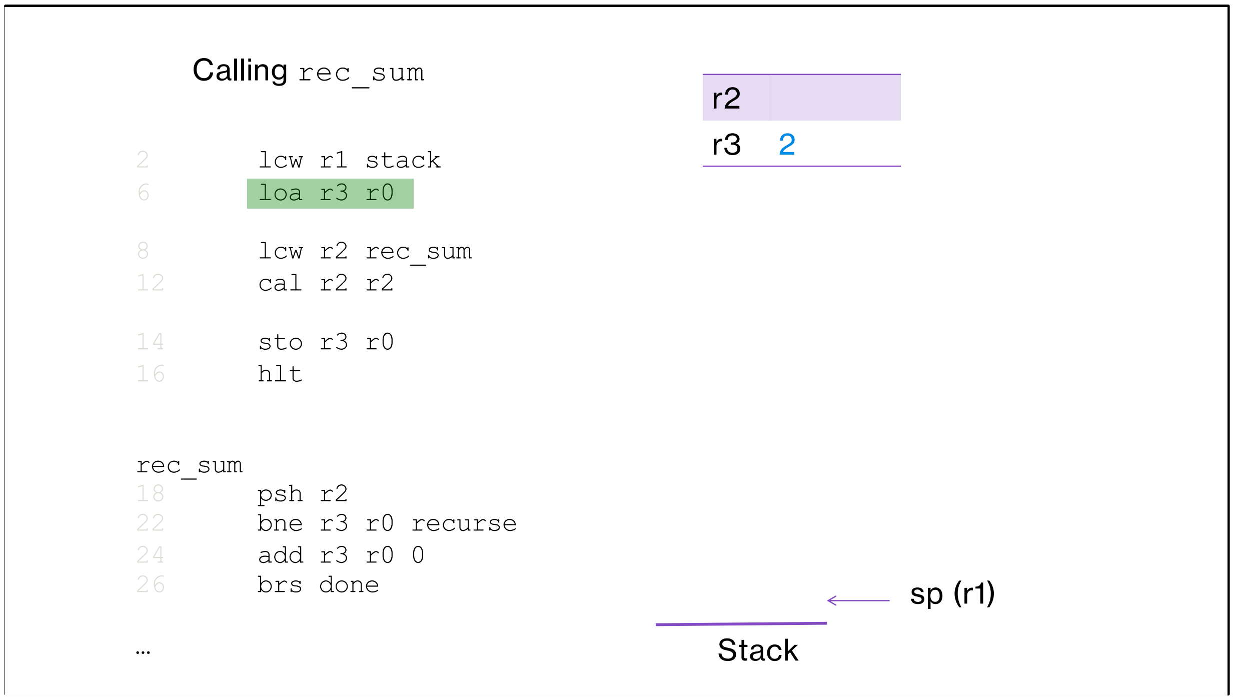
Let's walk through an example of how rec_sum would work. I have annotated on the left the numbers of the addresses. Remember, the word size is 2 bytes in cs51 and certain instructions like lcw, psh, and pop end up being decoded in two instructions. That's why we jump from 2 to 6 instead of 2 to 4. You see our stack has been set up and r1 is the stack pointer to its top.



Calling `rec_sum`

```
2       lcw r1 stack
6       loa r3 r0

8       lcw r2 rec_sum
12      cal r2 r2

14      sto r3 r0
16      hlt


rec_sum
18      psh r2
22      bne r3 r0 recurse
24      add r3 r0 0
26      brs done

…
```

next, we will prompt the user for their input n.

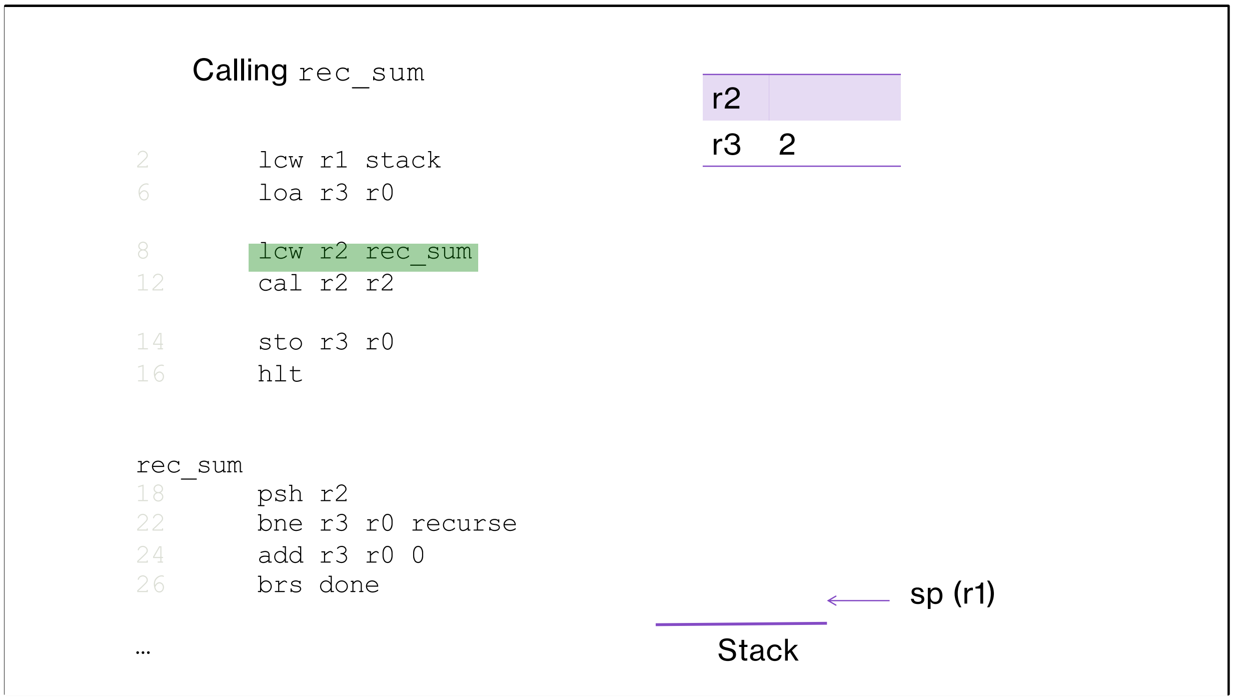


Calling `rec_sum`

```
2       lcw r1 stack
6       loa r3 r0

8       lcw r2 rec_sum
12      cal r2 r2

14      sto r3 r0
16      hlt


rec_sum
18      psh r2
22      bne r3 r0 recurse
24      add r3 r0 0
26      brs done

…
```

r2	
r3	2

Let's say they type 2 which we will store in r3

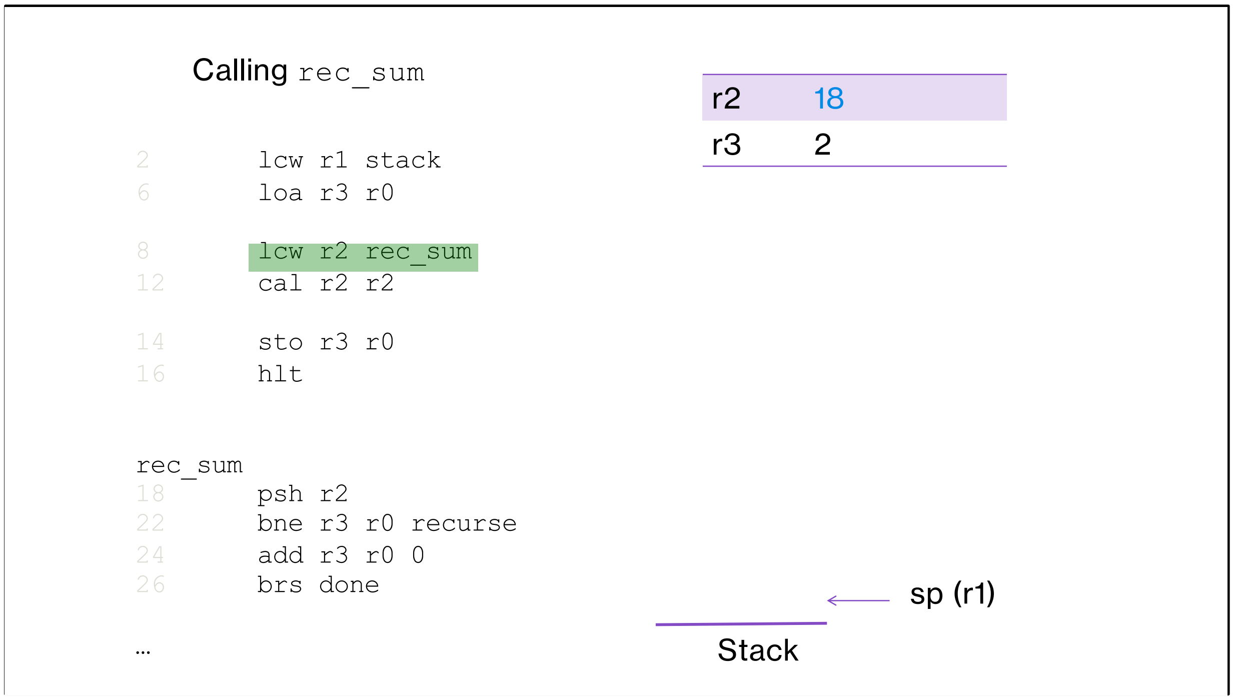


Calling `rec_sum`

```
2       lcw r1 stack
6       loa r3 r0

8       lcw r2 rec_sum
12      cal r2 r2

14      sto r3 r0
16      hlt


rec_sum
18      psh r2
22      bne r3 r0 recurse
24      add r3 r0 0
26      brs done

…
```

r2	
r3	2

Next, we have lcw r2 rec_sum



Calling `rec_sum`

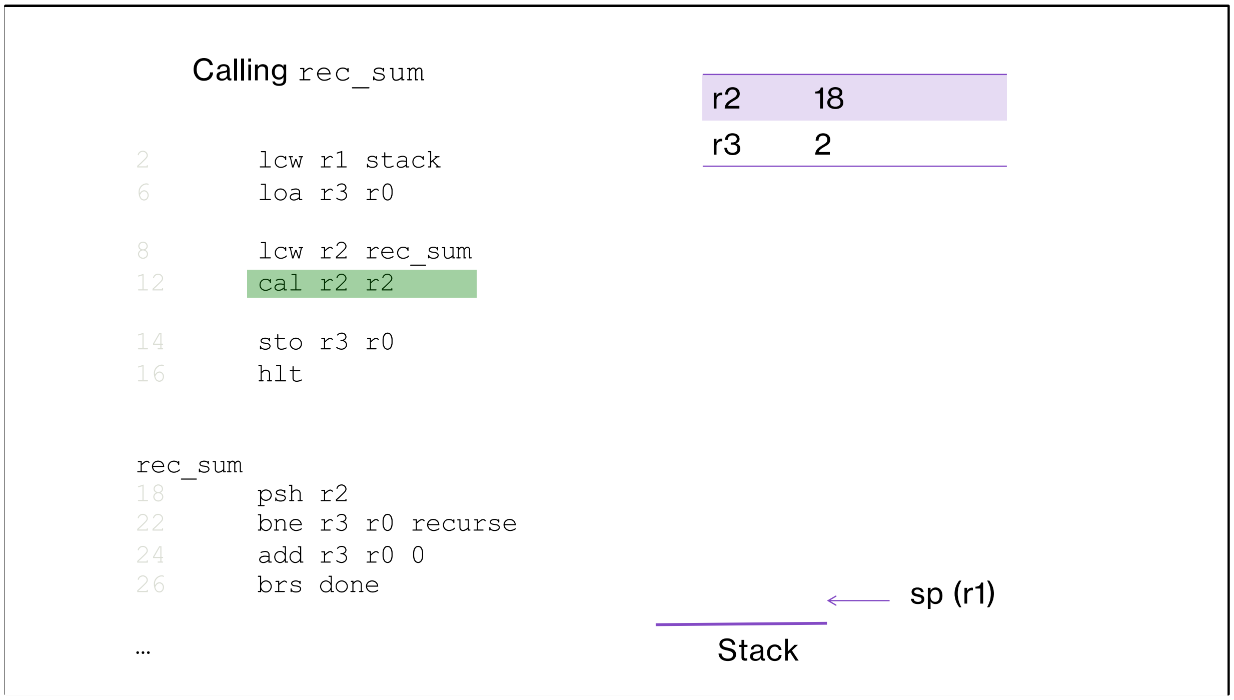
```
2        lcw r1 stack
6        loa r3 r0

8        lcw r2 rec_sum
12       cal r2 r2

14       sto r3 r0
16       hlt


rec_sum
18       psh r2
22       bne r3 r0 recurse
24       add r3 r0 0
26       brs done

…
```

r2	18
r3	2

sp (r1)

Stack

Remember, this instruction will store in r2 the location of the first line of the rec_sum function. That would be address 18

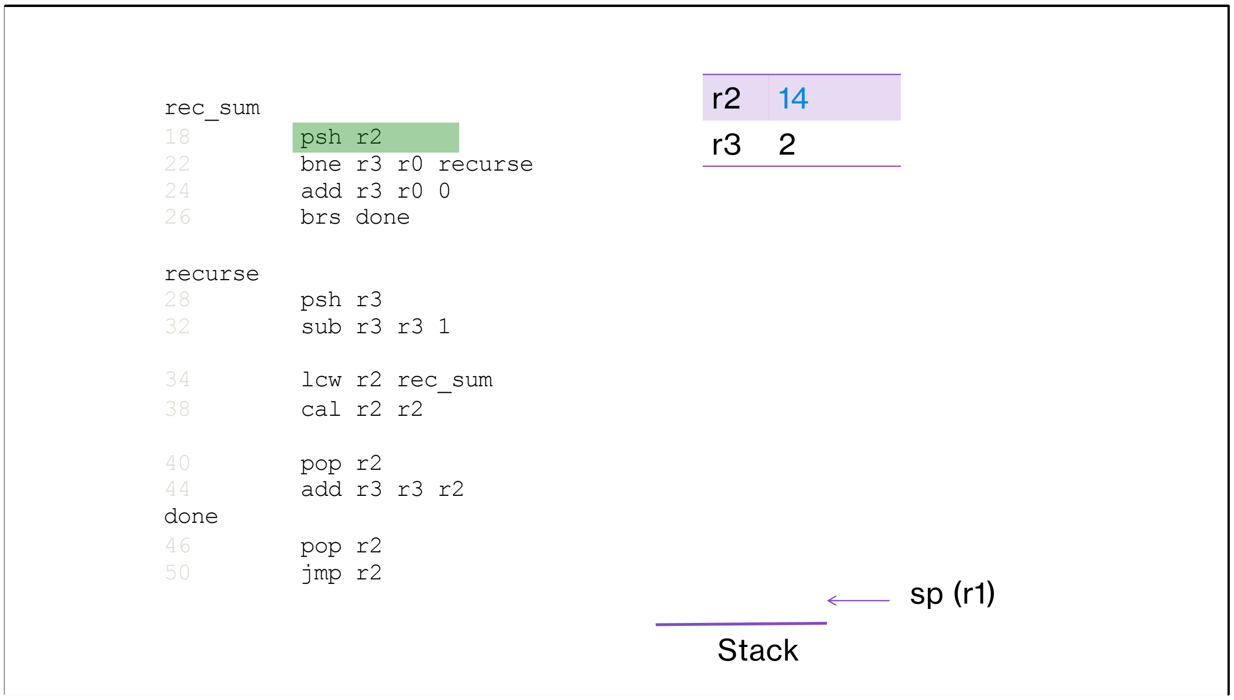


Calling `rec_sum`

```
2       lcw r1 stack
6       loa r3 r0

8       lcw r2 rec_sum
12      cal r2 r2

14      sto r3 r0
16      hlt


rec_sum
18      psh r2
22      bne r3 r0 recurse
24      add r3 r0 0
26      brs done

…
```

r2	18
r3	2

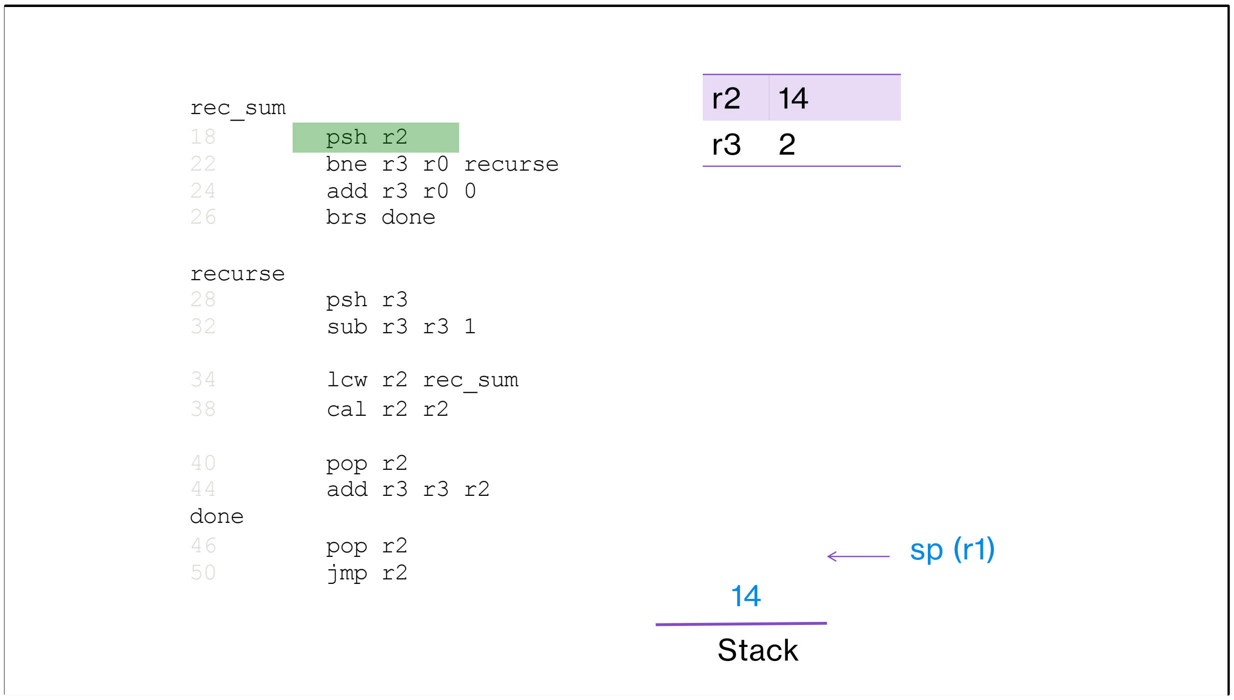
cal r2 r2 will do two things. First, it will jump to the address that r2 has stored (i.e. address 18 which is the location of the first line of the rec_sum function) and second it will update r2 to the address of the next instruction after cal, that is 14.



```
rec_sum
18        psh r2
22        bne r3 r0 recurse
24        add r3 r0 0
26        brs done

recurse
28        psh r3
32        sub r3 r3 1

34        lcw r2 rec_sum
38        cal r2 r2

40        pop r2
44        add r3 r3 r2
done
46        pop r2
50        jmp r2
```

r2	14
r3	2

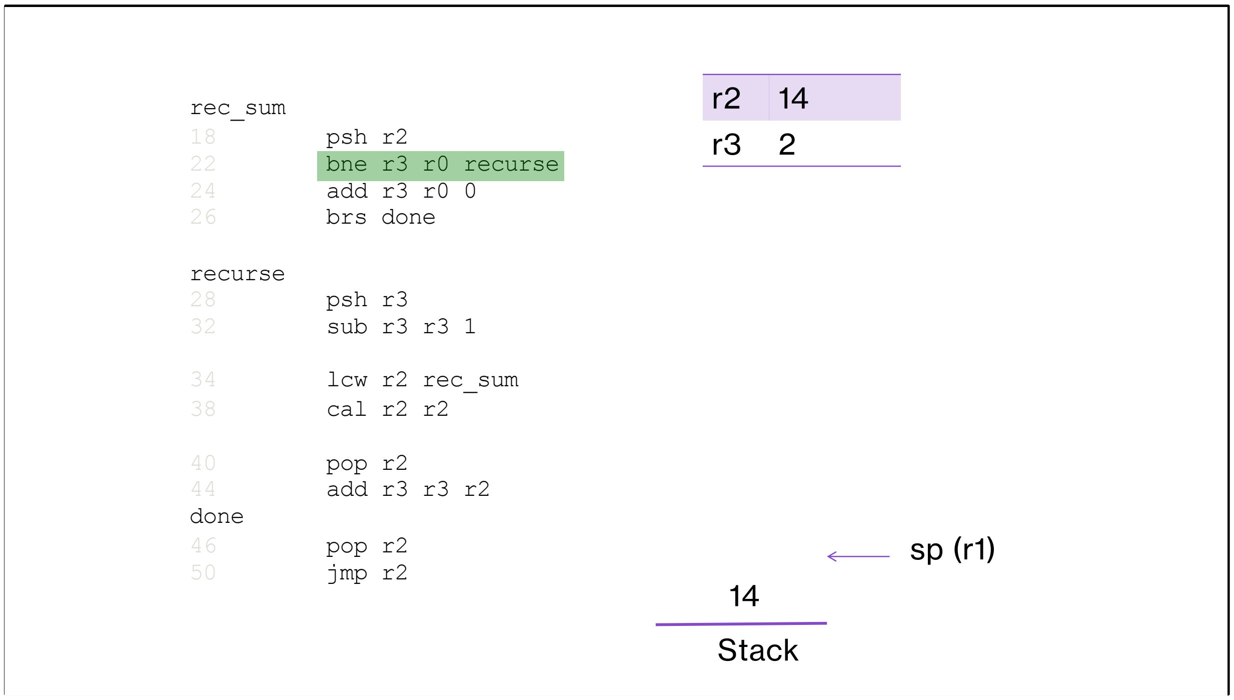
This results in moving the execution to line 18 and updating r2 with the address 14.



```
rec_sum
18          psh r2
22          bne r3 r0 recurse
24          add r3 r0 0
26          brs done

recurse
28          psh r3
32          sub r3 r3 1

34          lcw r2 rec_sum
38          cal r2 r2

40          pop r2
44          add r3 r3 r2
done
46          pop r2
50          jmp r2
```

r2	14
r3	2

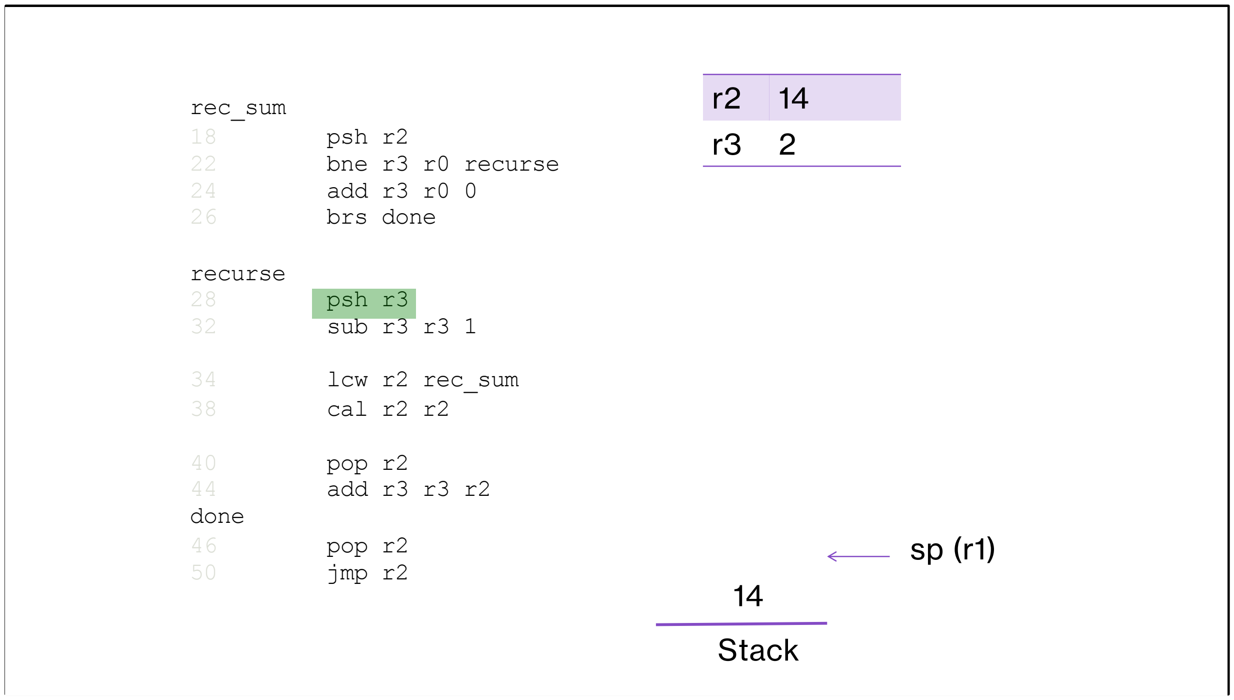
OK, we are now in the first line of rec_sum and we see need to push the contents of the r2 register into the stack. Remember, when you push something to the stack, the stack pointer (i.e., register r1) has to go up (i.e. in a lower memory address).



```
rec_sum
18        psh r2
22        bne r3 r0 recurse
24        add r3 r0 0
26        brs done

recurse
28        psh r3
32        sub r3 r3 1

34        lcw r2 rec_sum
38        cal r2 r2

40        pop r2
44        add r3 r3 r2
done
46        pop r2
50        jmp r2
```

r2	14
r3	2

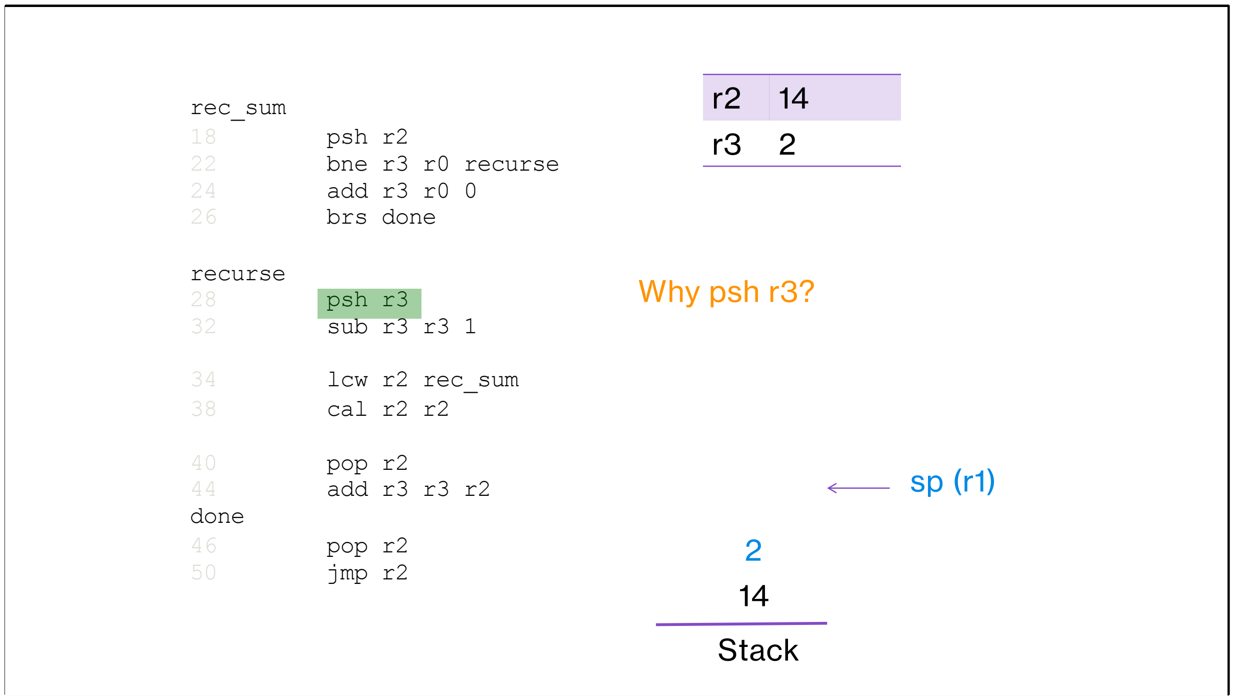
Next, we check the base case by asking whether r3!=0. Since it is, we have to branch to the label recurse.



```
rec_sum
18        psh r2
22        bne r3 r0 recurse
24        add r3 r0 0
26        brs done

recurse
28        psh r3
32        sub r3 r3 1

34        lcw r2 rec_sum
38        cal r2 r2

40        pop r2
44        add r3 r3 r2
done
46        pop r2
50        jmp r2
```

r2	14
r3	2

We see that we will need to push the contents of the register r3 into the stack.

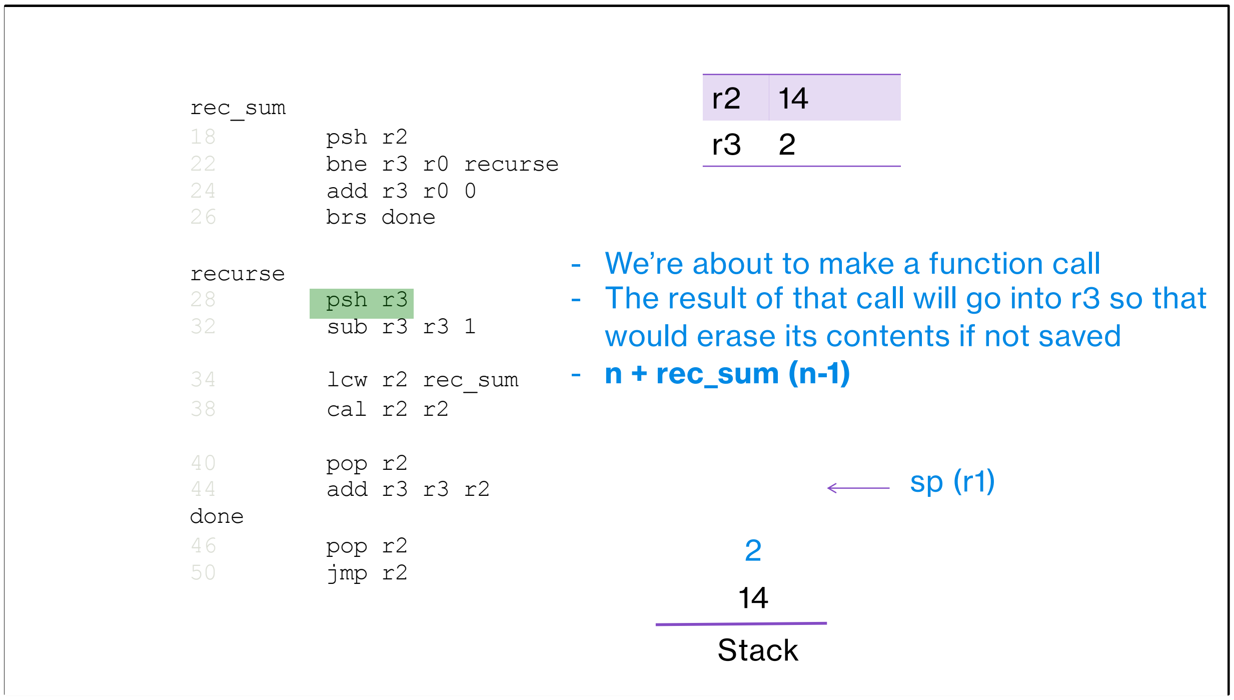


```
rec_sum
18          psh r2
22          bne r3 r0 recurse
24          add r3 r0 0
26          brs done

recurse
28          psh r3
32          sub r3 r3 1

34          lcw r2 rec_sum
38          cal r2 r2

40          pop r2
44          add r3 r3 r2
done
46          pop r2
50          jmp r2
```

r2	14
r3	2

Why psh r3?

This will result in 2 being pushed into the stack and the stack pointer r1 moving one word up. Why do you think we do this?



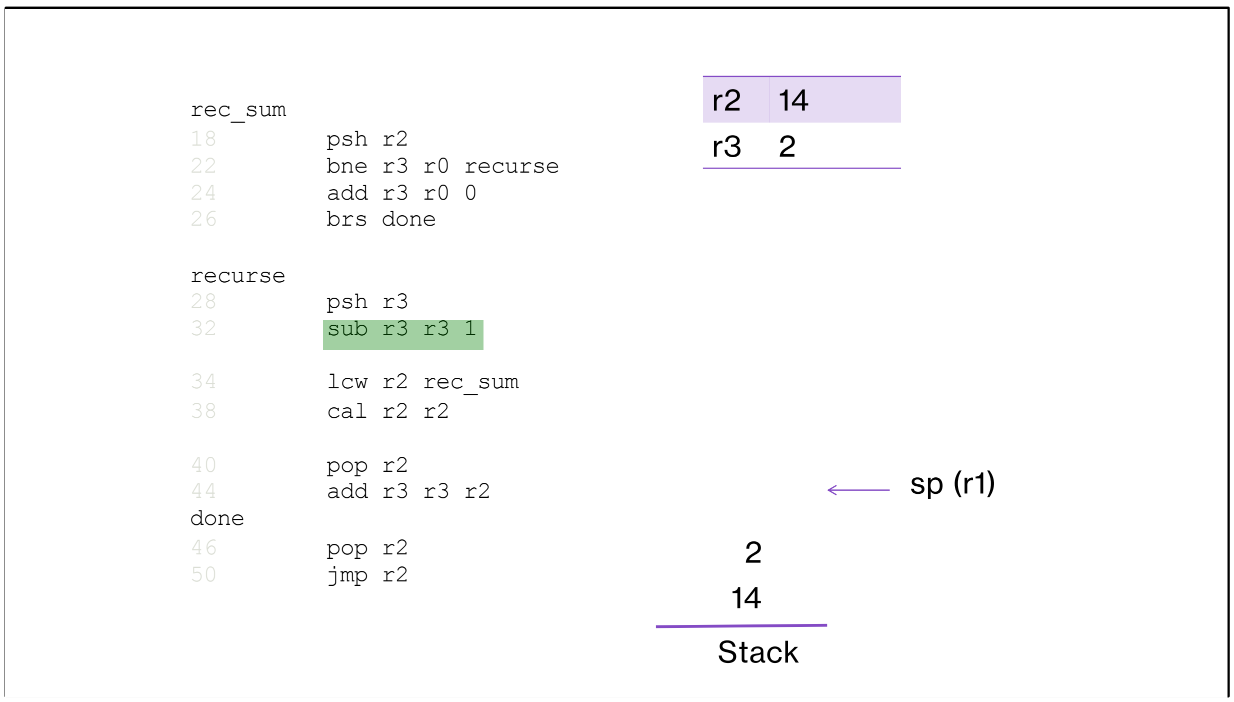
```
rec_sum
18          psh r2
22          bne r3 r0 recurse
24          add r3 r0 0
26          brs done

recurse
28          psh r3
32          sub r3 r3 1

34          lcw r2 rec_sum
38          cal r2 r2

40          pop r2
44          add r3 r3 r2
done
46          pop r2
50          jmp r2
```

r2	14
r3	2

- We're about to make a function call
- The result of that call will go into r3 so that would erase its contents if not saved
- **n + rec_sum (n-1)**

sp (r1)
2
14
Stack

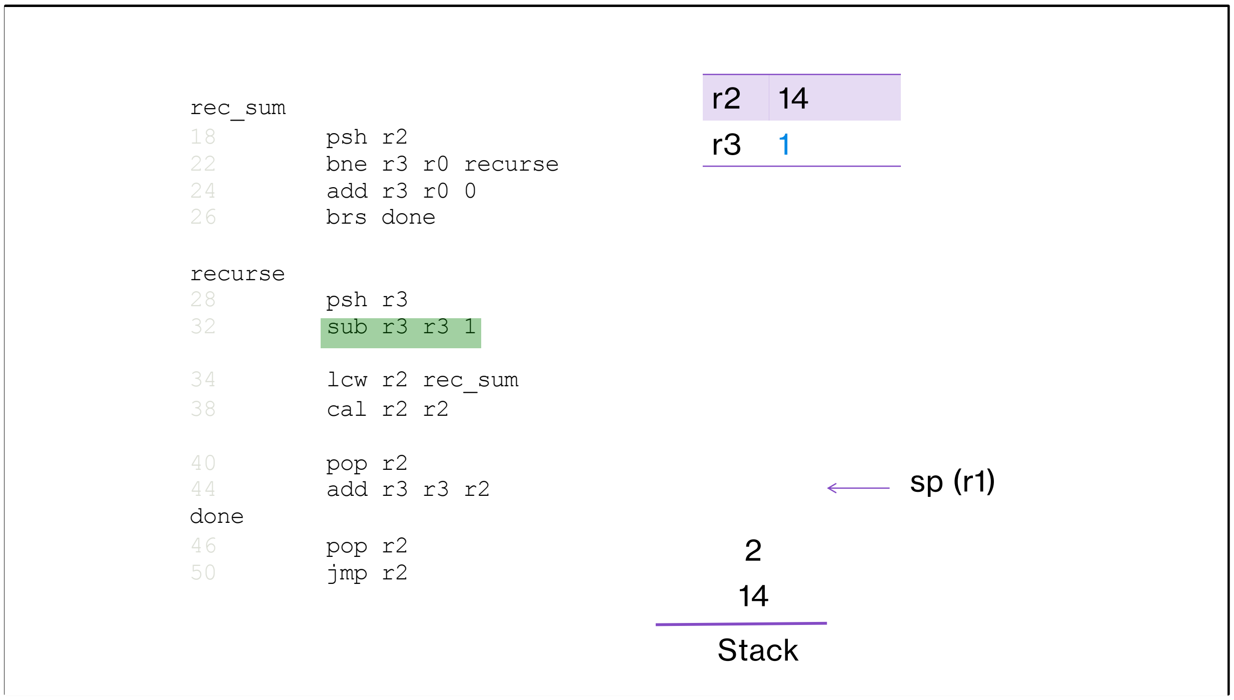
Remember, we are about to recurse, that is make a function call. The result of that call will go into r3 so that would erase the contents if we don't save them. So by pushing 2 in the stack, we can reuse it down the line. That ensures that when we have the expression n + rec_sum(n-1), our n has been stored safely in the stack.



```
rec_sum
18        psh r2
22        bne r3 r0 recurse
24        add r3 r0 0
26        brs done

recurse
28        psh r3
32        sub r3 r3 1

34        lcw r2 rec_sum
38        cal r2 r2

40        pop r2
44        add r3 r3 r2
done
46        pop r2
50        jmp r2
```

r2	14
r3	2

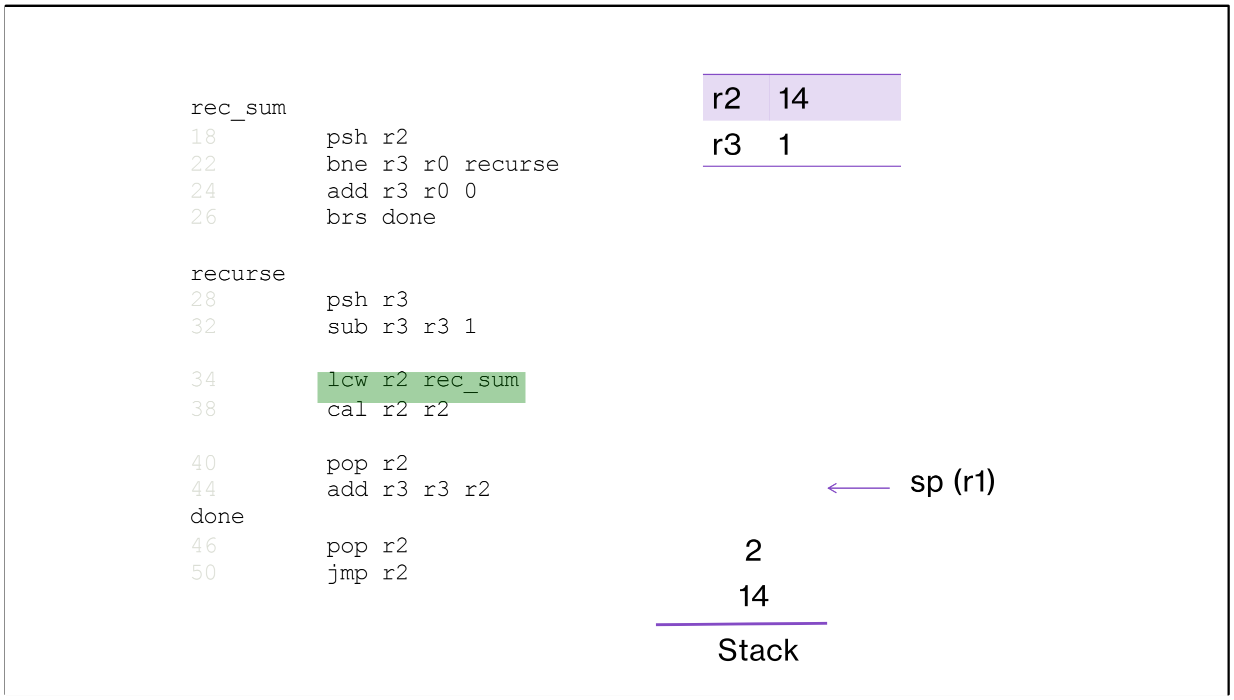
Now that we settled that, we move to the next instruction, sub r3 r3 1.



```
rec_sum
18        psh r2
22        bne r3 r0 recurse
24        add r3 r0 0
26        brs done

recurse
28        psh r3
32        sub r3 r3 1

34        lcw r2 rec_sum
38        cal r2 r2

40        pop r2
44        add r3 r3 r2
done
46        pop r2
50        jmp r2
```

r2	14
r3	1

This will decrement r3 by 1, which will update r3 to 1. (see our stack is still safely storing 2!).

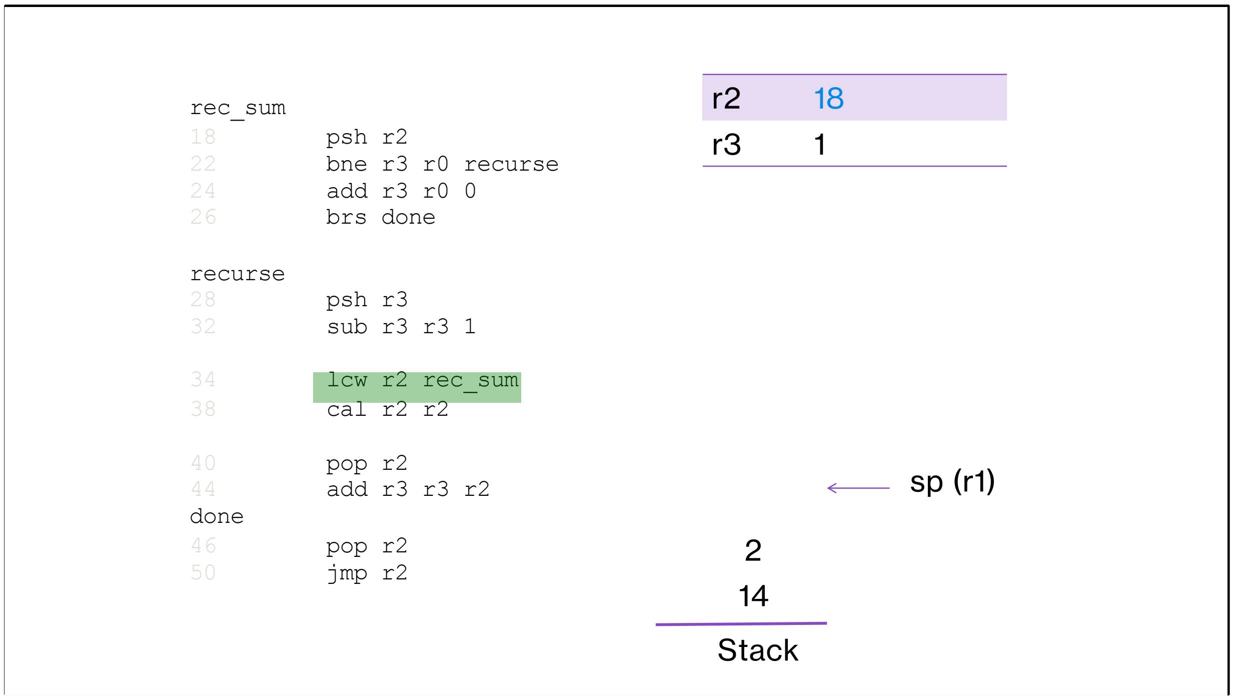


```
rec_sum
18        psh r2
22        bne r3 r0 recurse
24        add r3 r0 0
26        brs done

recurse
28        psh r3
32        sub r3 r3 1

34        lcw r2 rec_sum
38        cal r2 r2

40        pop r2
44        add r3 r3 r2
done
46        pop r2
50        jmp r2
```

r2	14
r3	1

sp (r1)
2
14
Stack

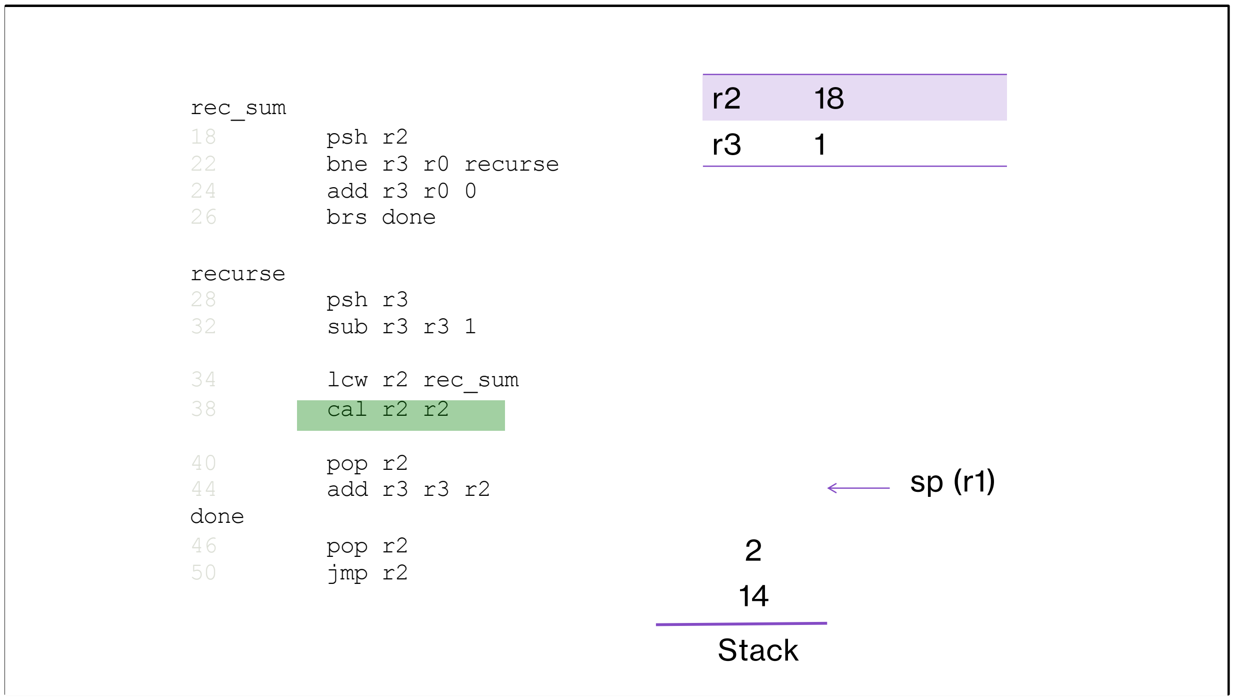
Next we have the recursive call to rec_sum.



```
rec_sum
18        psh r2
22        bne r3 r0 recurse
24        add r3 r0 0
26        brs done

recurse
28        psh r3
32        sub r3 r3 1

34        lcw r2 rec_sum
38        cal r2 r2

40        pop r2
44        add r3 r3 r2
done
46        pop r2
50        jmp r2
```

r2	18
r3	1

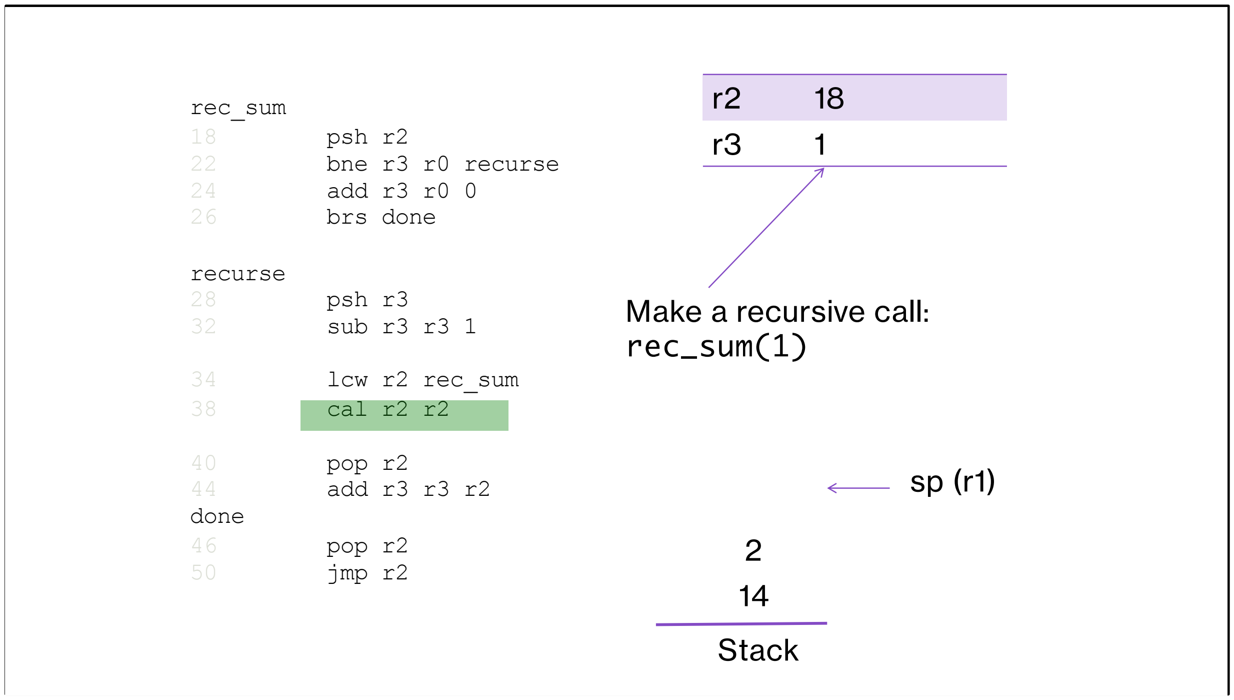
lcw r2 rec_sum will result in r2 storing the address of the first address of rec_sum function, i.e. 18



```
rec_sum
18        psh r2
22        bne r3 r0 recurse
24        add r3 r0 0
26        brs done

recurse
28        psh r3
32        sub r3 r3 1

34        lcw r2 rec_sum
38        cal r2 r2

40        pop r2
44        add r3 r3 r2
done
46        pop r2
50        jmp r2
```

r2	18
r3	1

Next, we have cal r2 r2.

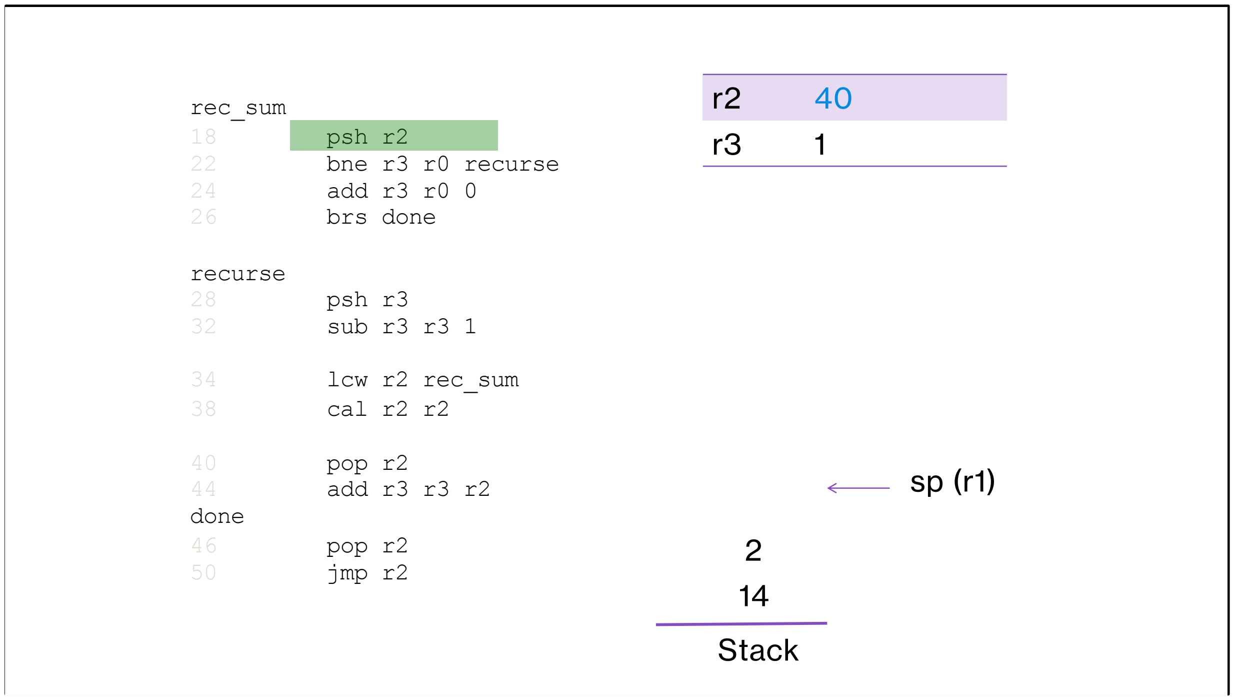


```
rec_sum
18        psh r2
22        bne r3 r0 recurse
24        add r3 r0 0
26        brs done

recurse
28        psh r3
32        sub r3 r3 1

34        lcw r2 rec_sum
38        cal r2 r2

40        pop r2
44        add r3 r3 r2
done
46        pop r2
50        jmp r2
```

r2	18
r3	1

Make a recursive call:
`rec_sum(1)`

Remember that cal does two things. We will jump to the address stored by r2, that is to the first line of rec_sum which is psh r2 in address 18 and update r2 to the address of the next instruction after cal which is pop r2 at address 40.



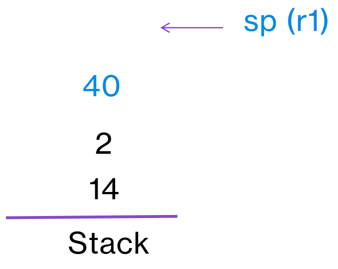
```
rec_sum
18          psh r2
22          bne r3 r0 recurse
24          add r3 r0 0
26          brs done

recurse
28          psh r3
32          sub r3 r3 1

34          lcw r2 rec_sum
38          cal r2 r2

40          pop r2
44          add r3 r3 r2
done
46          pop r2
50          jmp r2
```

r2	40
r3	1

we will also update r2 to hold the address of the next line of where we came, that is the next line after cal r2 r2. Since this is our second time doing this, I will note that as loc:cal 1.

```
rec_sum
18          psh r2
22          bne r3 r0 recurse
24          add r3 r0 0
26          brs done

recurse
28          psh r3
32          sub r3 r3 1

34          lcw r2 rec_sum
38          cal r2 r2

40          pop r2
44          add r3 r3 r2
done
46          pop r2
50          jmp r2
```

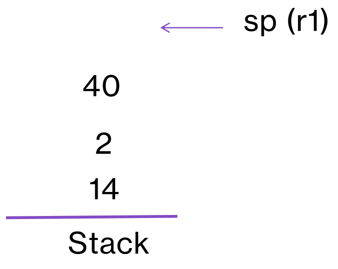
r2	40
r3	1



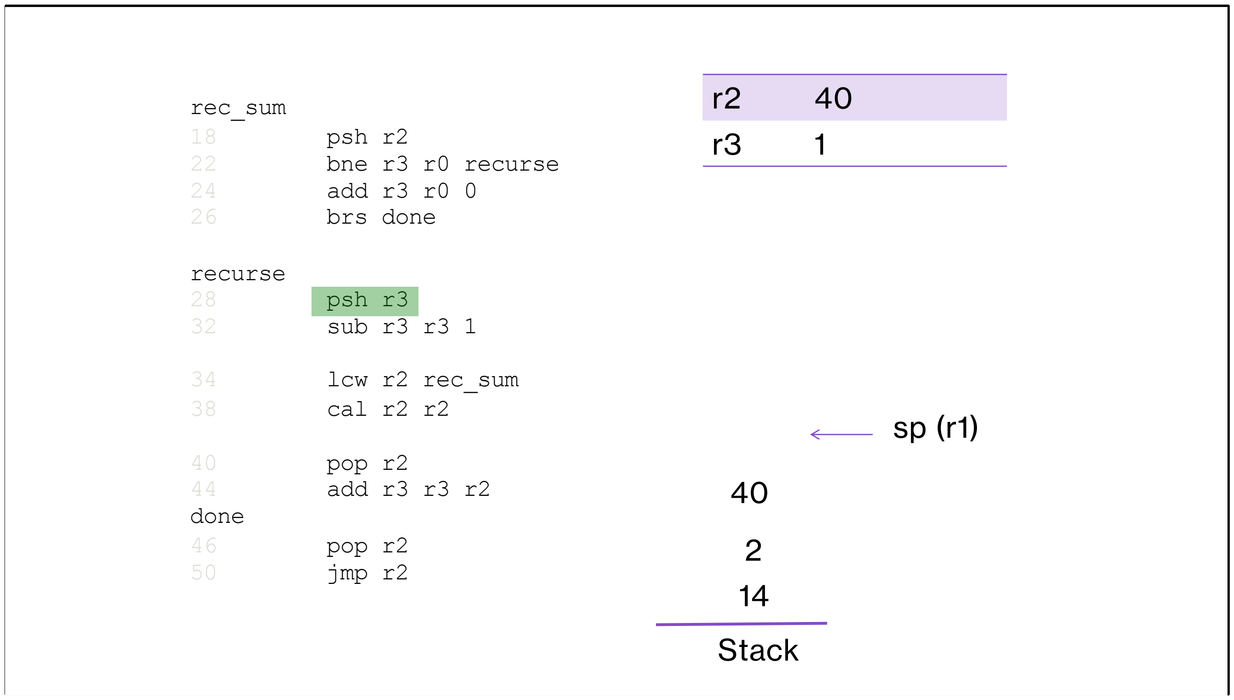
We are again in rec_sum and the first line asks us to push the contents of r2 in the stack, that is push 40 and move our stack pointer in a lower memory address (up).

```
rec_sum
18        psh r2
22        bne r3 r0 recurse
24        add r3 r0 0
26        brs done

recurse
28        psh r3
32        sub r3 r3 1

34        lcw r2 rec_sum
38        cal r2 r2

40        pop r2
44        add r3 r3 r2
done
46        pop r2
50        jmp r2
```

r2	40
r3	1



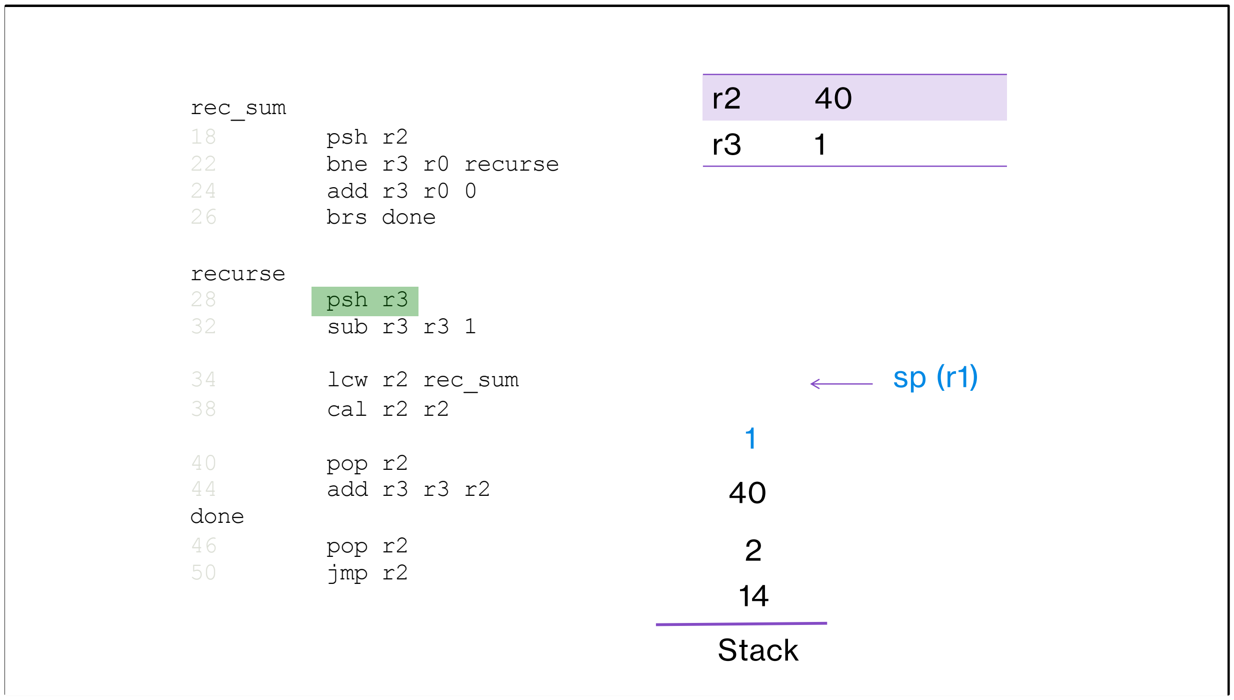
Next instruction checks the base case and since r3!=0 we will branch to the label recurse.



```
rec_sum
18        psh r2
22        bne r3 r0 recurse
24        add r3 r0 0
26        brs done

recurse
28        psh r3
32        sub r3 r3 1

34        lcw r2 rec_sum
38        cal r2 r2

40        pop r2
44        add r3 r3 r2
done
46        pop r2
50        jmp r2
```

r2	40
r3	1

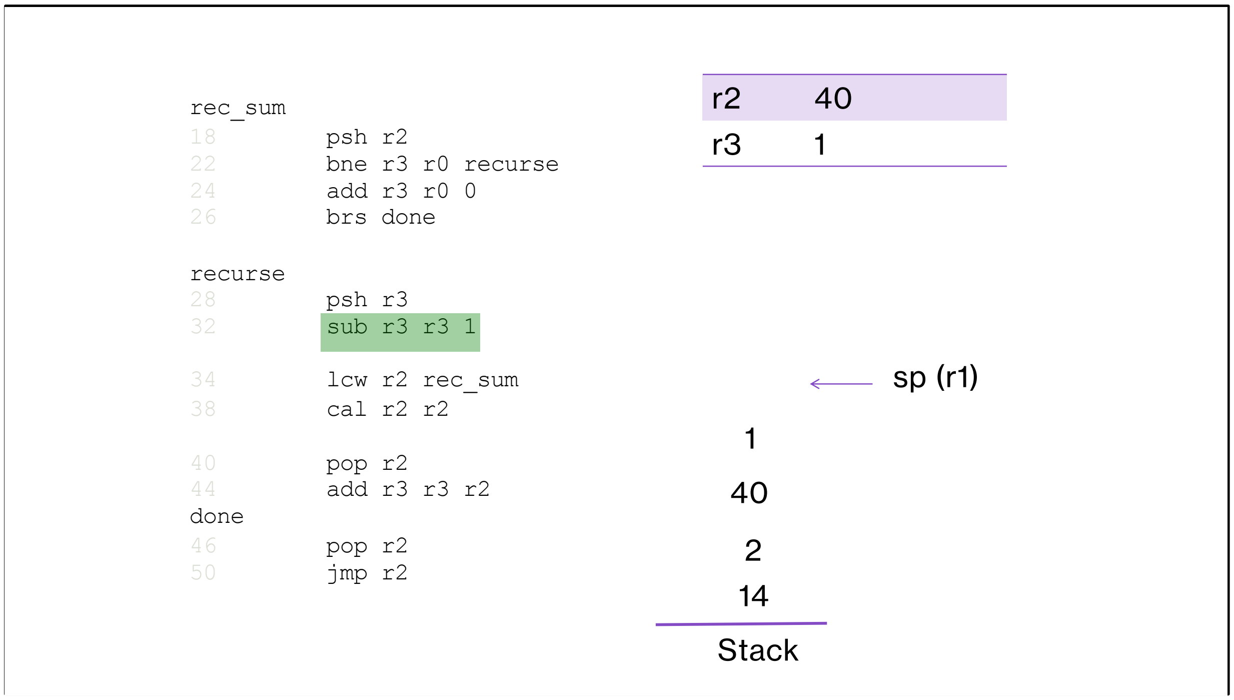
First thing we do in recurse is to push the contents of r3 to 1. Remember, we do that so we don't lose the n from n+rec_sum(n-1).



```
rec_sum
18        psh r2
22        bne r3 r0 recurse
24        add r3 r0 0
26        brs done

recurse
28        psh r3
32        sub r3 r3 1

34        lcw r2 rec_sum
38        cal r2 r2

40        pop r2
44        add r3 r3 r2
done
46        pop r2
50        jmp r2
```

r2	40
r3	1

That means that 1 is safely in our stack for later use and we move the stack pointer in a lower memory address.



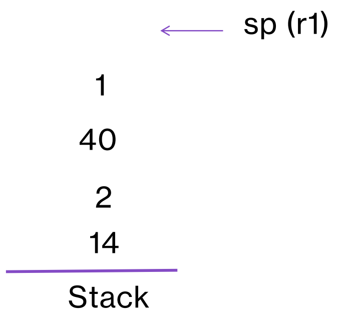
```
rec_sum
18        psh r2
22        bne r3 r0 recurse
24        add r3 r0 0
26        brs done

recurse
28        psh r3
32        sub r3 r3 1

34        lcw r2 rec_sum
38        cal r2 r2

40        pop r2
44        add r3 r3 r2
done
46        pop r2
50        jmp r2
```

r2	40
r3	1

Next, we decrement r3 by 1.

```
rec_sum
18        psh r2
22        bne r3 r0 recurse
24        add r3 r0 0
26        brs done

recurse
28        psh r3
32        sub r3 r3 1

34        lcw r2 rec_sum
38        cal r2 r2

40        pop r2
44        add r3 r3 r2
done
46        pop r2
50        jmp r2
```

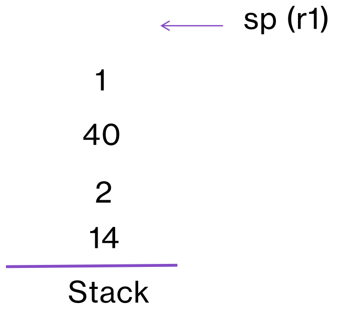
r2	40
r3	0



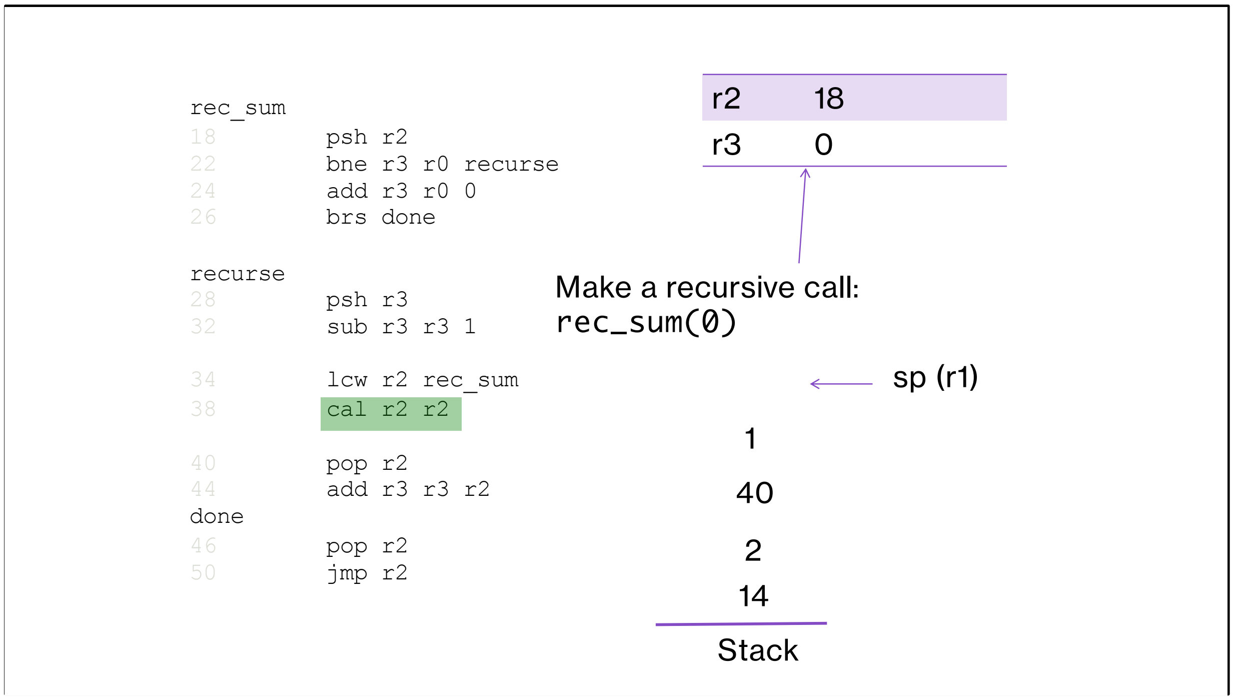
Which results in r3 being equal to 0.

```
rec_sum
18        psh r2
22        bne r3 r0 recurse
24        add r3 r0 0
26        brs done

recurse
28        psh r3
32        sub r3 r3 1

34        lcw r2 rec_sum
38        cal r2 r2

40        pop r2
44        add r3 r3 r2
done
46        pop r2
50        jmp r2
```

r2	18
r3	0



Again, we update r2 to hold the address of the first line of rec_sum, i.e. 18

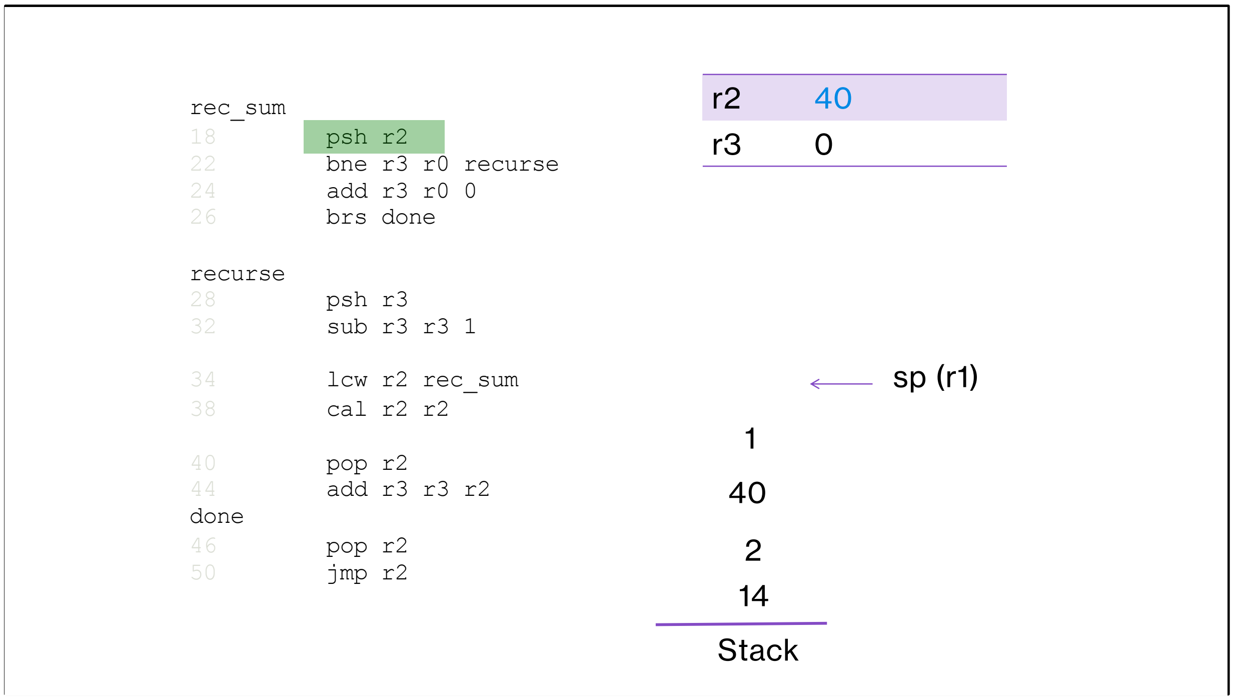


```
rec_sum
18        psh r2
22        bne r3 r0 recurse
24        add r3 r0 0
26        brs done

recurse
28        psh r3
32        sub r3 r3 1

34        lcw r2 rec_sum
38        cal r2 r2

40        pop r2
44        add r3 r3 r2
done
46        pop r2
50        jmp r2
```

r2	18
r3	0

Make a recursive call:
`rec_sum(0)`

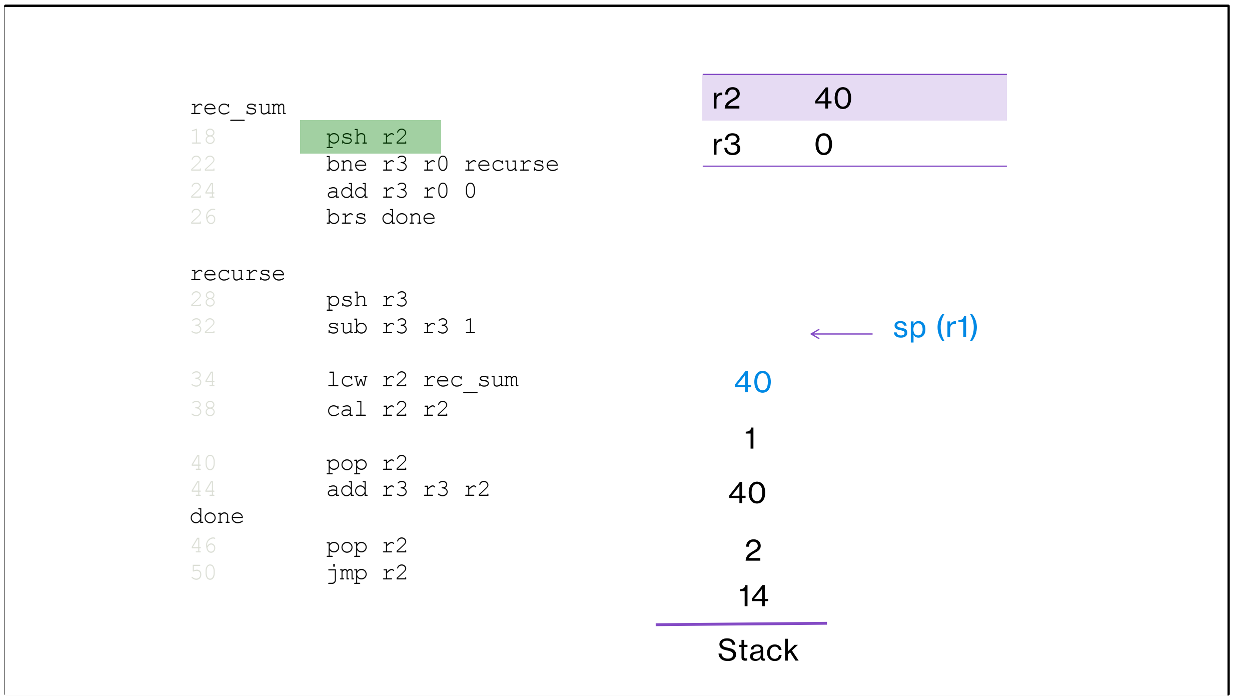
cal r2 r2 will do two things, it will jump the execution to the location stated by r2 (i.e. address 18) and update r2 to the address of the next instruction after cal, that is the address of pop r2 which is 40.



```
rec_sum
18          psh r2
22          bne r3 r0 recurse
24          add r3 r0 0
26          brs done

recurse
28          psh r3
32          sub r3 r3 1

34          lcw r2 rec_sum
38          cal r2 r2

40          pop r2
44          add r3 r3 r2
done
46          pop r2
50          jmp r2
```

r2	40
r3	0

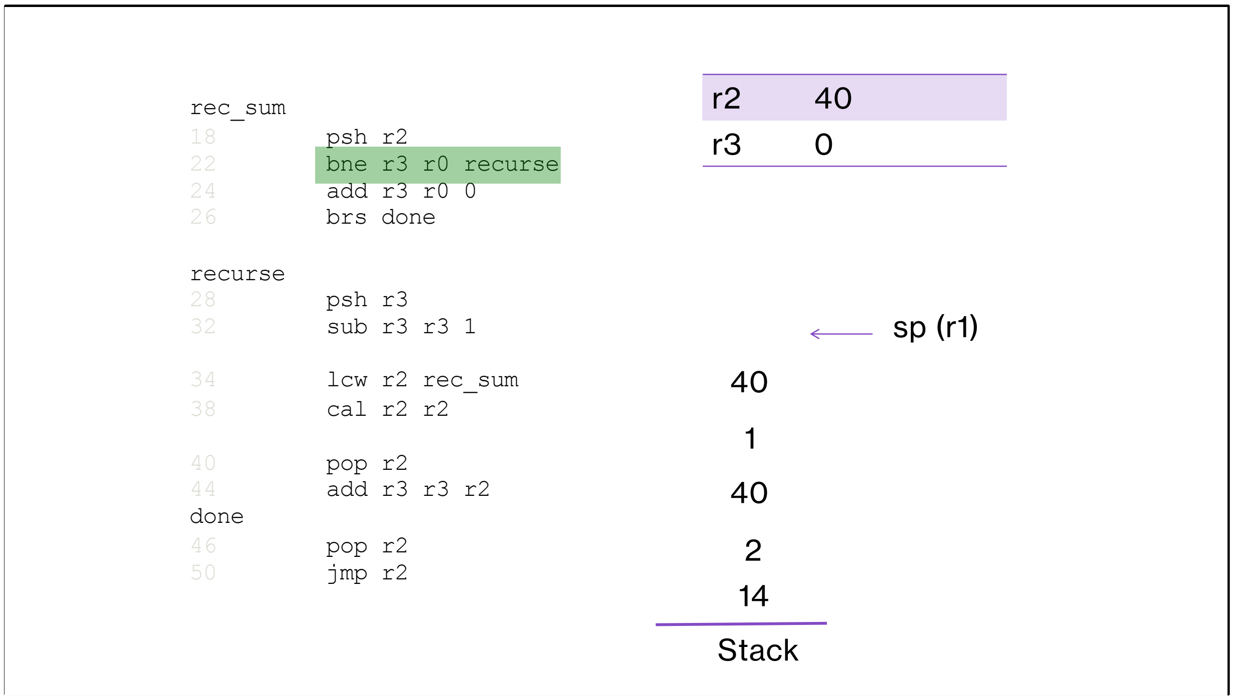
You see that now the execution is at address 18 and r2 has been updated to 40



```
rec_sum
18          psh r2
22          bne r3 r0 recurse
24          add r3 r0 0
26          brs done

recurse
28          psh r3
32          sub r3 r3 1

34          lcw r2 rec_sum
38          cal r2 r2

40          pop r2
44          add r3 r3 r2
done
46          pop r2
50          jmp r2
```

r2	40
r3	0

We will next push that address to the stack and move the stack pointer.



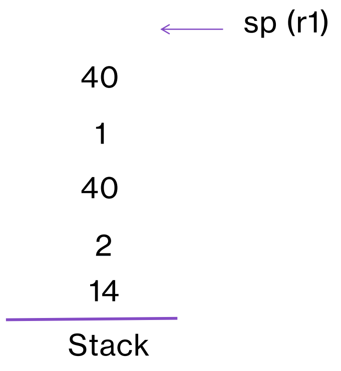
```
rec_sum
18        psh r2
22        bne r3 r0 recurse
24        add r3 r0 0
26        brs done

recurse
28        psh r3
32        sub r3 r3 1

34        lcw r2 rec_sum
38        cal r2 r2

40        pop r2
44        add r3 r3 r2
done
46        pop r2
50        jmp r2
```

r2	40
r3	0

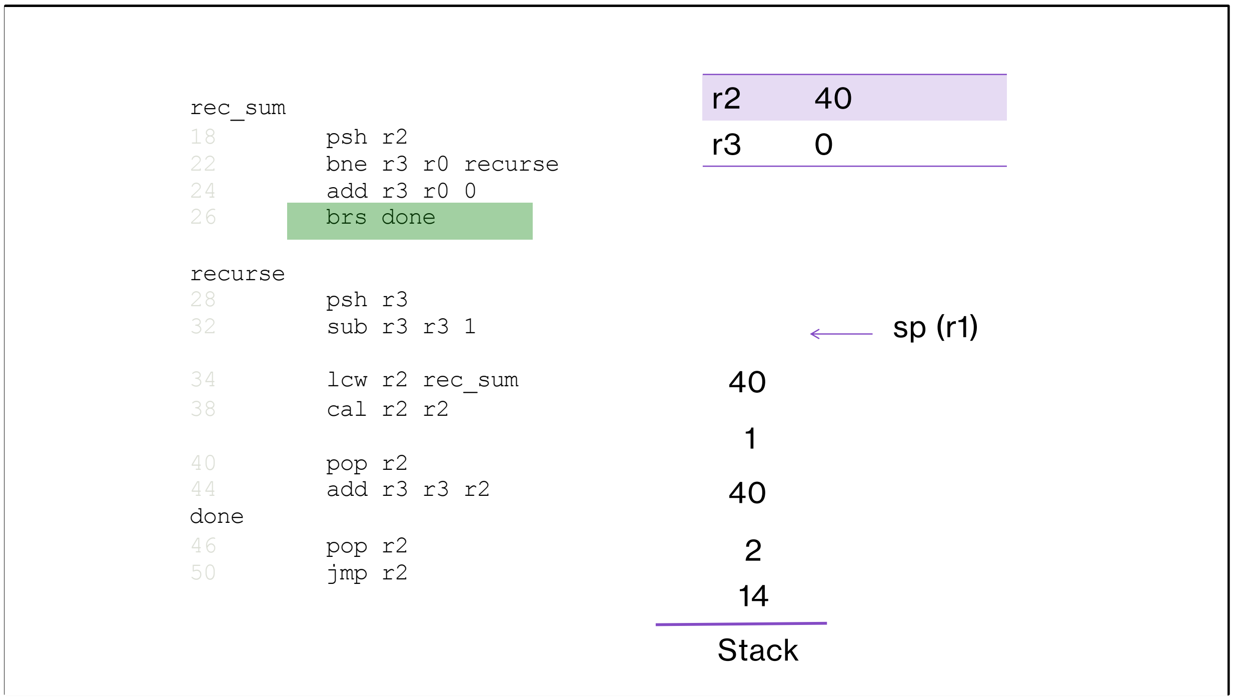
We check our base case and for the first time, r3 is equal to 0. So instead of branching to the recurse label, we need to continue sequentially to our next instruction.

```
rec_sum
18        psh r2
22        bne r3 r0 recurse
24        add r3 r0 0
26        brs done

recurse
28        psh r3
32        sub r3 r3 1

34        lcw r2 rec_sum
38        cal r2 r2

40        pop r2
44        add r3 r3 r2
done
46        pop r2
50        jmp r2
```

r2	40
r3	0



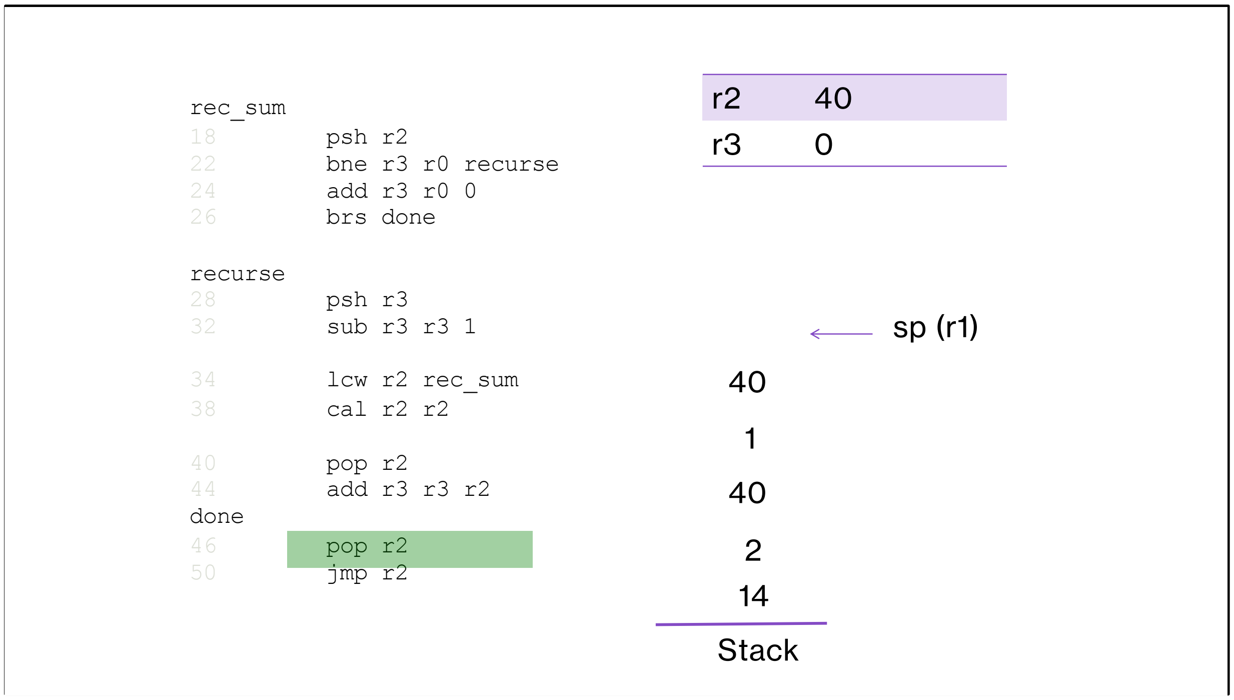
This will set r3 to 0 which is equivalent to returning 0.



```
rec_sum
18        psh r2
22        bne r3 r0 recurse
24        add r3 r0 0
26        brs done

recurse
28        psh r3
32        sub r3 r3 1

34        lcw r2 rec_sum
38        cal r2 r2

40        pop r2
44        add r3 r3 r2
done
46        pop r2
50        jmp r2
```

r2	40
r3	0

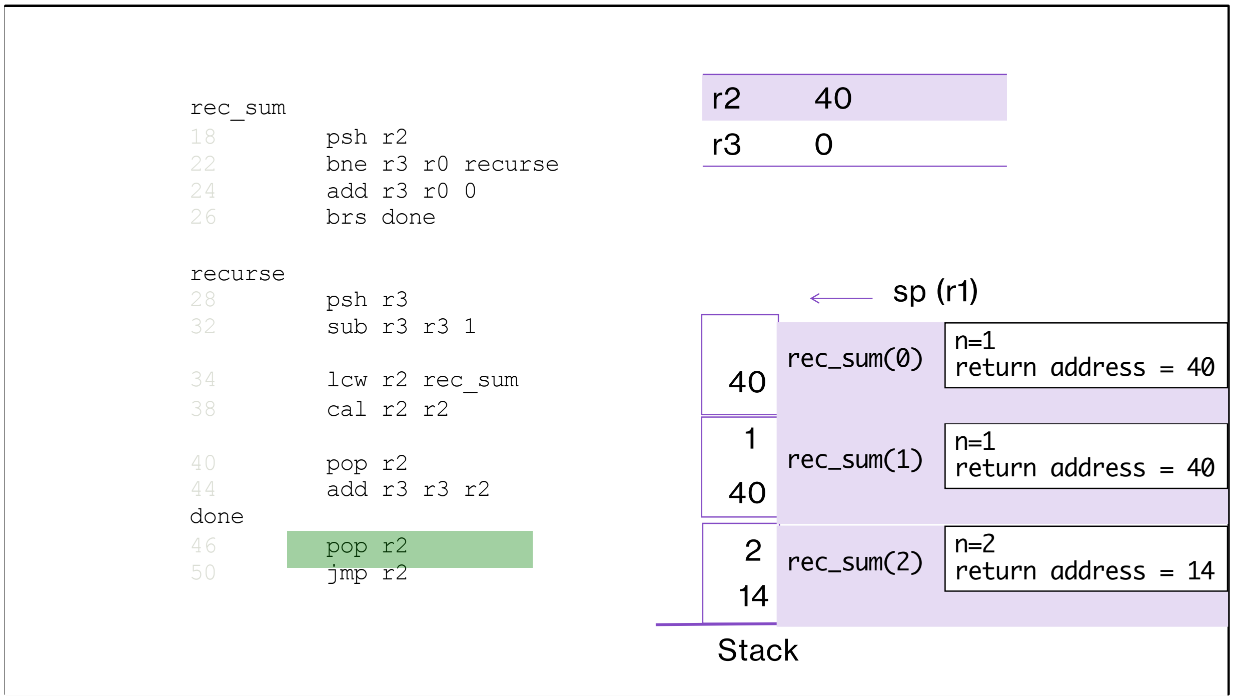
We are ready now to branch to the label done.



```
rec_sum
18        psh r2
22        bne r3 r0 recurse
24        add r3 r0 0
26        brs done

recurse
28        psh r3
32        sub r3 r3 1

34        lcw r2 rec_sum
38        cal r2 r2

40        pop r2
44        add r3 r3 r2
done
46        pop r2
50        jmp r2
```

r2	40
r3	0

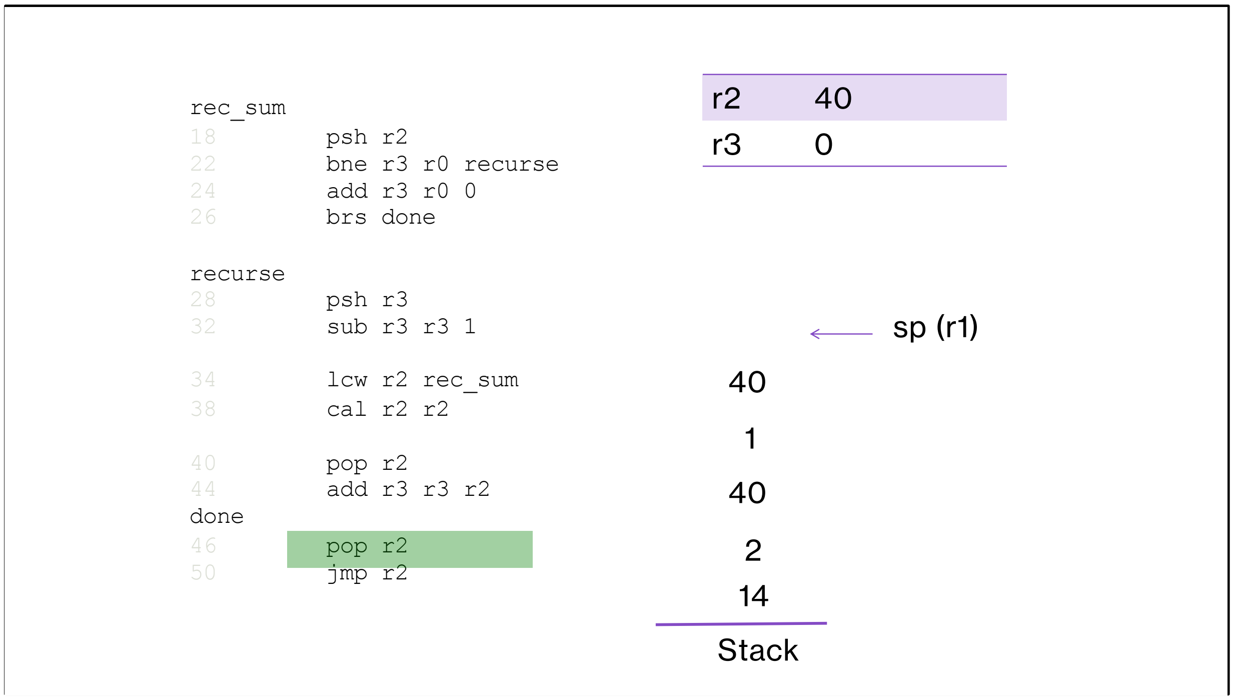
This brings us to the instruction pop r2



```
rec_sum
18        psh r2
22        bne r3 r0 recurse
24        add r3 r0 0
26        brs done

recurse
28        psh r3
32        sub r3 r3 1

34        lcw r2 rec_sum
38        cal r2 r2

40        pop r2
44        add r3 r3 r2
done
46        pop r2
50        jmp r2
```

r2	40
r3	0

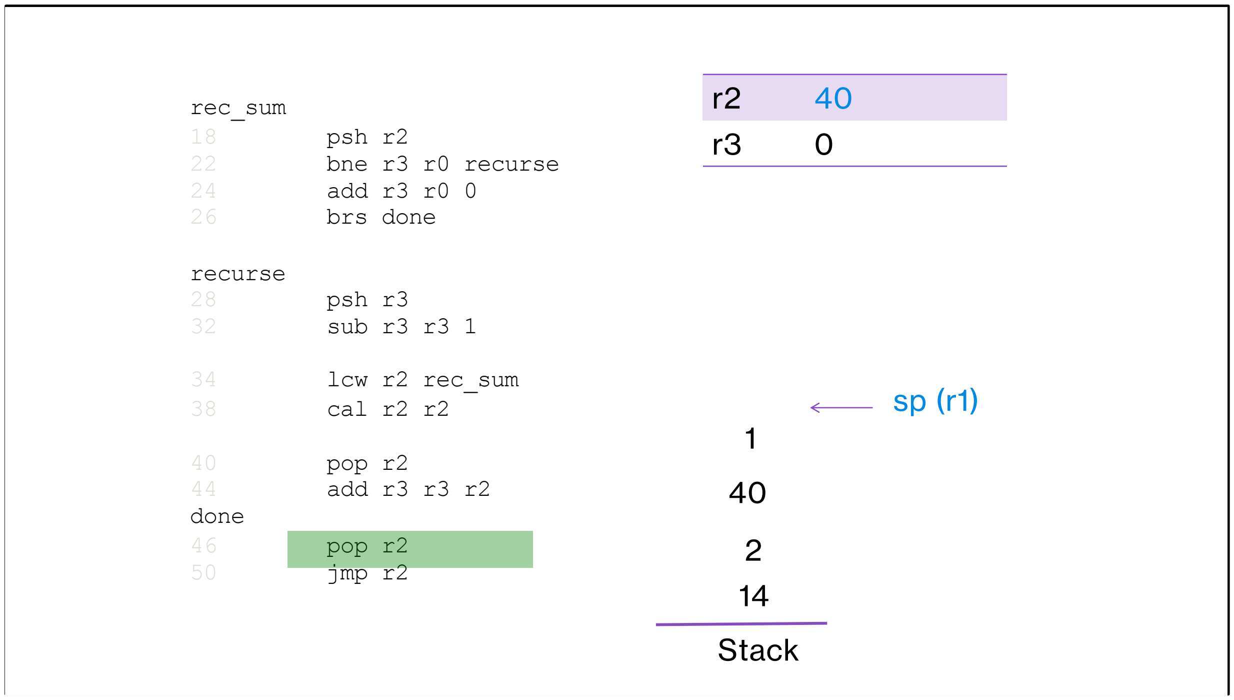
If you think about it, what we have been doing is directly linked with stack frames!



```
rec_sum
18        psh r2
22        bne r3 r0 recurse
24        add r3 r0 0
26        brs done

recurse
28        psh r3
32        sub r3 r3 1

34        lcw r2 rec_sum
38        cal r2 r2

40        pop r2
44        add r3 r3 r2
done
46        pop r2
50        jmp r2
```

r2	40
r3	0

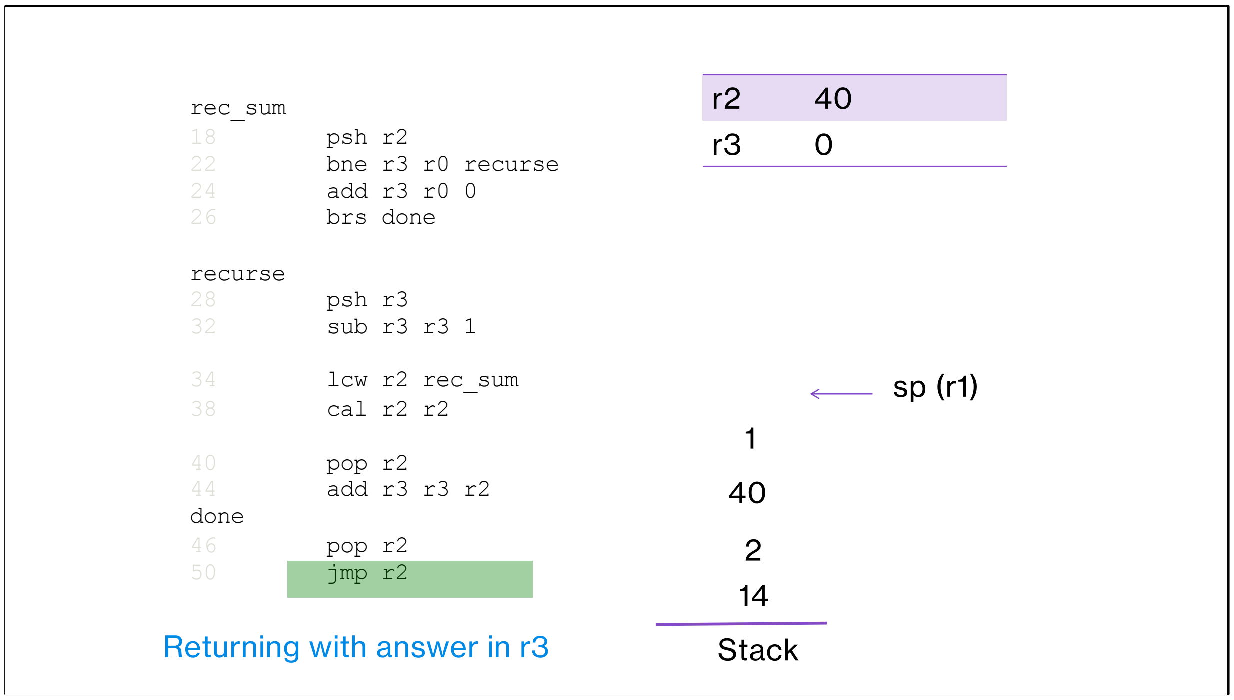
OK let’s go back to seeing how our code execution will unfold. We were about to pop the top of the stack and update r2.



```
rec_sum
18          psh r2
22          bne r3 r0 recurse
24          add r3 r0 0
26          brs done

recurse
28          psh r3
32          sub r3 r3 1

34          lcw r2 rec_sum
38          cal r2 r2

40          pop r2
44          add r3 r3 r2
done
46          pop r2
50          jmp r2
```

r2	40
r3	0

This will result in 40 being popped and stored in r2 and the stack pointer moving to higher memory addresses.

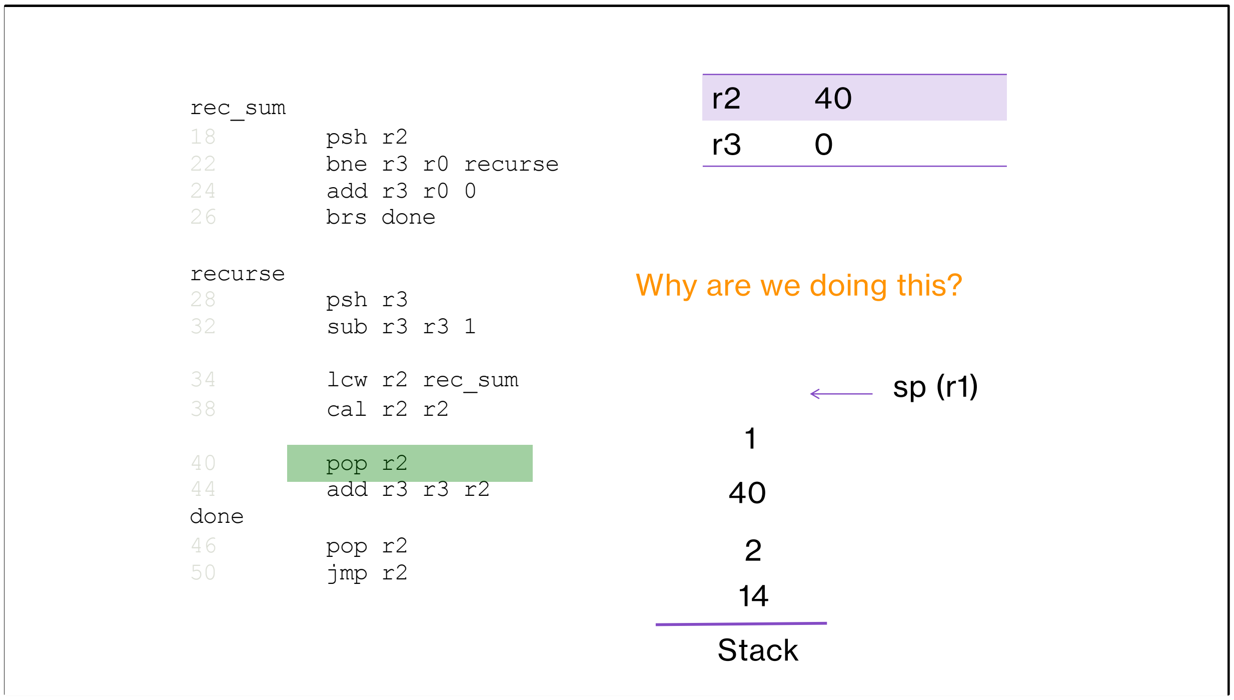


```
rec_sum
18          psh r2
22          bne r3 r0 recurse
24          add r3 r0 0
26          brs done

recurse
28          psh r3
32          sub r3 r3 1

34          lcw r2 rec_sum
38          cal r2 r2

40          pop r2
44          add r3 r3 r2
done
46          pop r2
50          jmp r2
```

Returning with answer in r3

r2	40
r3	0

Next, we execute jmp r2 which moves the execution of code to the address stored in r2, that is address 40. This will also coincide with returning with the answer (0) stored in r3.



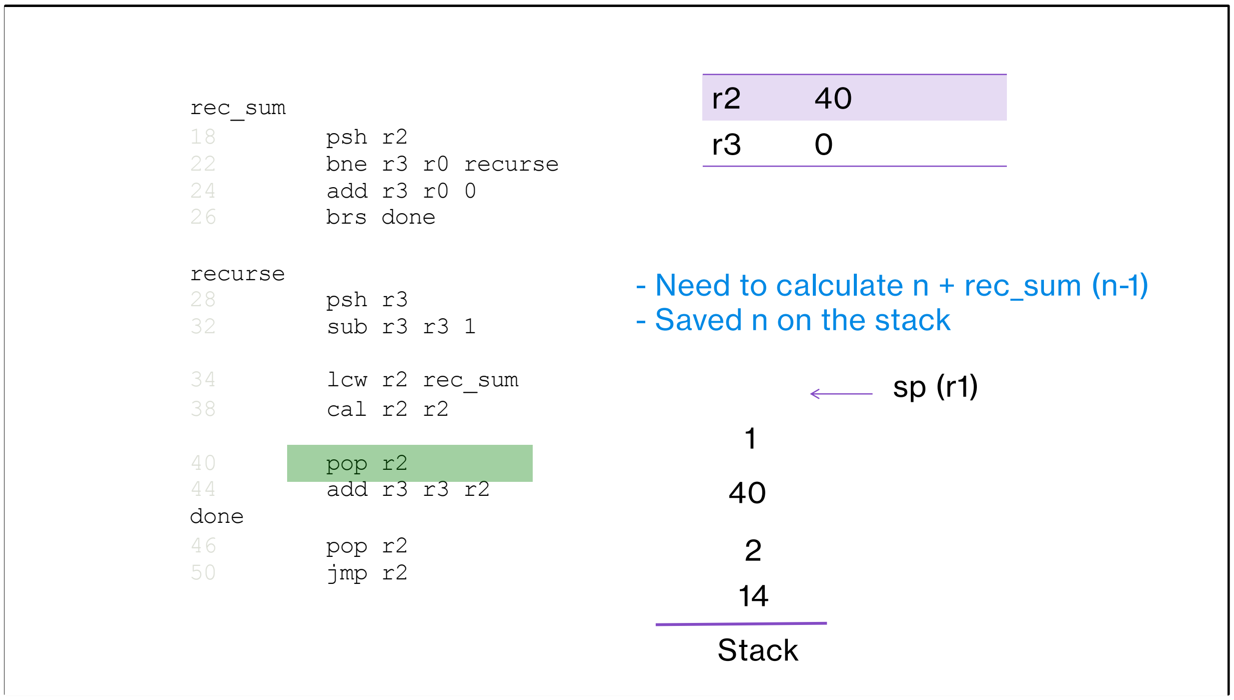
```
rec_sum
18          psh r2
22          bne r3 r0 recurse
24          add r3 r0 0
26          brs done

recurse
28          psh r3
32          sub r3 r3 1

34          lcw r2 rec_sum
38          cal r2 r2

40          pop r2
44          add r3 r3 r2
done
46          pop r2
50          jmp r2
```

r2	40
r3	0

Why are we doing this?

sp (r1)
1
40
2
14
Stack

Why do we have to pop next?



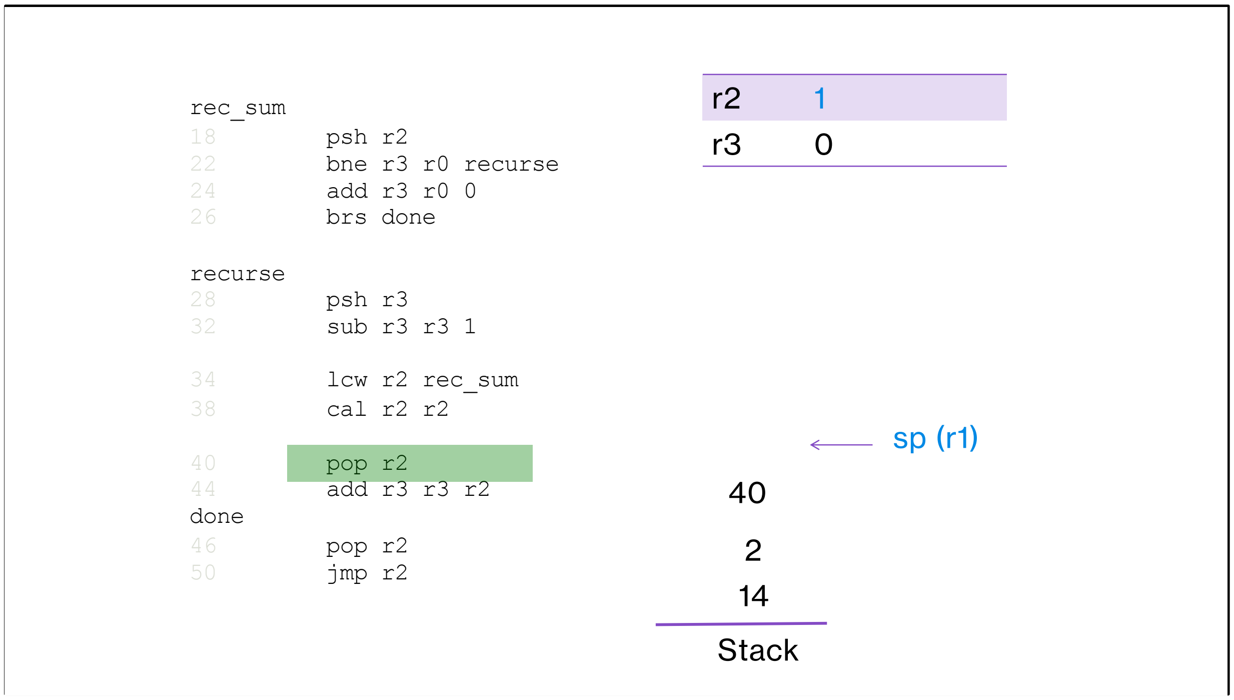
```
rec_sum
18        psh r2
22        bne r3 r0 recurse
24        add r3 r0 0
26        brs done

recurse
28        psh r3
32        sub r3 r3 1

34        lcw r2 rec_sum
38        cal r2 r2

40        pop r2
44        add r3 r3 r2
done
46        pop r2
50        jmp r2
```

r2	40
r3	0

- Need to calculate n + rec_sum (n-1)
- Saved n on the stack

sp (r1)
1
40
2
14
Stack

Remember, we had stashed away our n (i.e. 1) into the stack. Now it's time to use it by popping it.



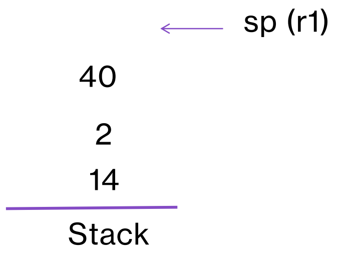
```
rec_sum
18          psh r2
22          bne r3 r0 recurse
24          add r3 r0 0
26          brs done

recurse
28          psh r3
32          sub r3 r3 1

34          lcw r2 rec_sum
38          cal r2 r2

40          pop r2
44          add r3 r3 r2
done
46          pop r2
50          jmp r2
```

r2	1
r3	0

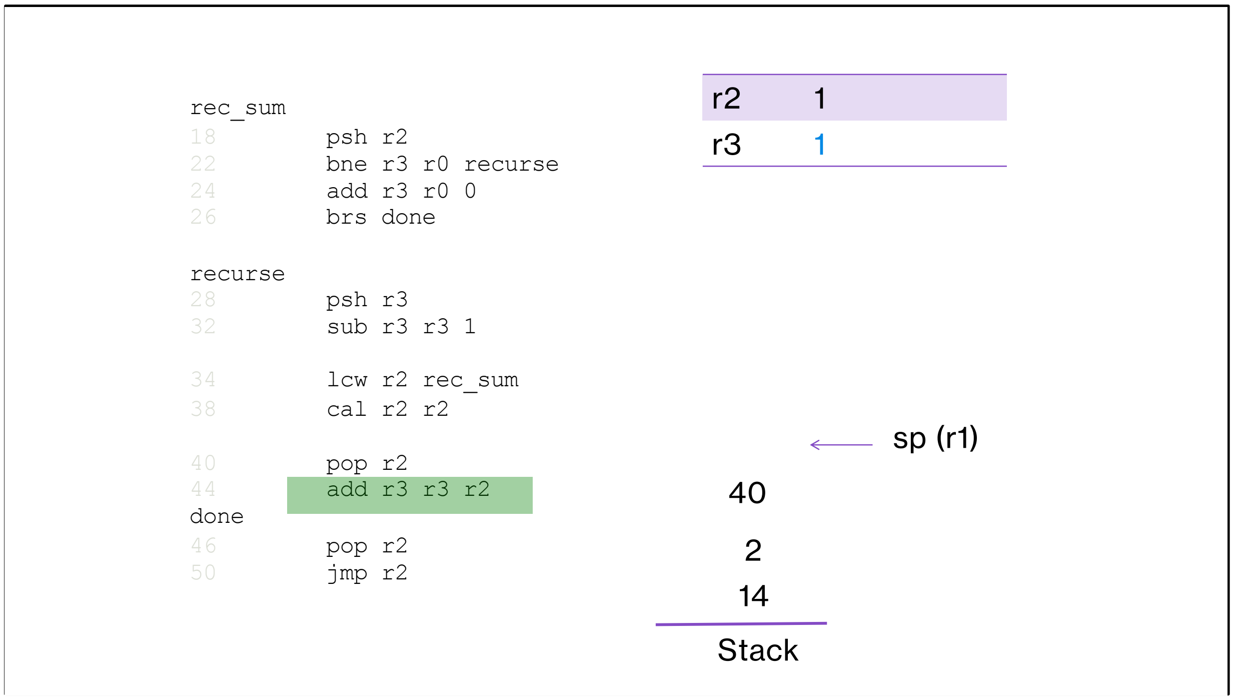
Ok we pop 1 from the top of the stack, move the stack pointer in higher memory address, and store 1 in r2.

```
rec_sum
18        psh r2
22        bne r3 r0 recurse
24        add r3 r0 0
26        brs done

recurse
28        psh r3
32        sub r3 r3 1

34        lcw r2 rec_sum
38        cal r2 r2

40        pop r2
44        add r3 r3 r2
done
46        pop r2
50        jmp r2
```

r2	1
r3	0



Next, we will update r3 by adding to it r2 (this is where we do the n+rec_sum(n-1), where n=1).



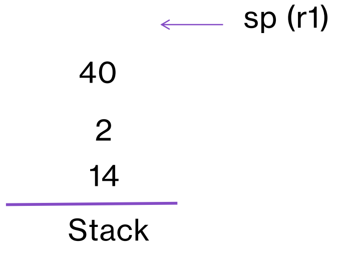
```
rec_sum
18        psh r2
22        bne r3 r0 recurse
24        add r3 r0 0
26        brs done

recurse
28        psh r3
32        sub r3 r3 1

34        lcw r2 rec_sum
38        cal r2 r2

40        pop r2
44        add r3 r3 r2
done
46        pop r2
50        jmp r2
```

r2	1
r3	1

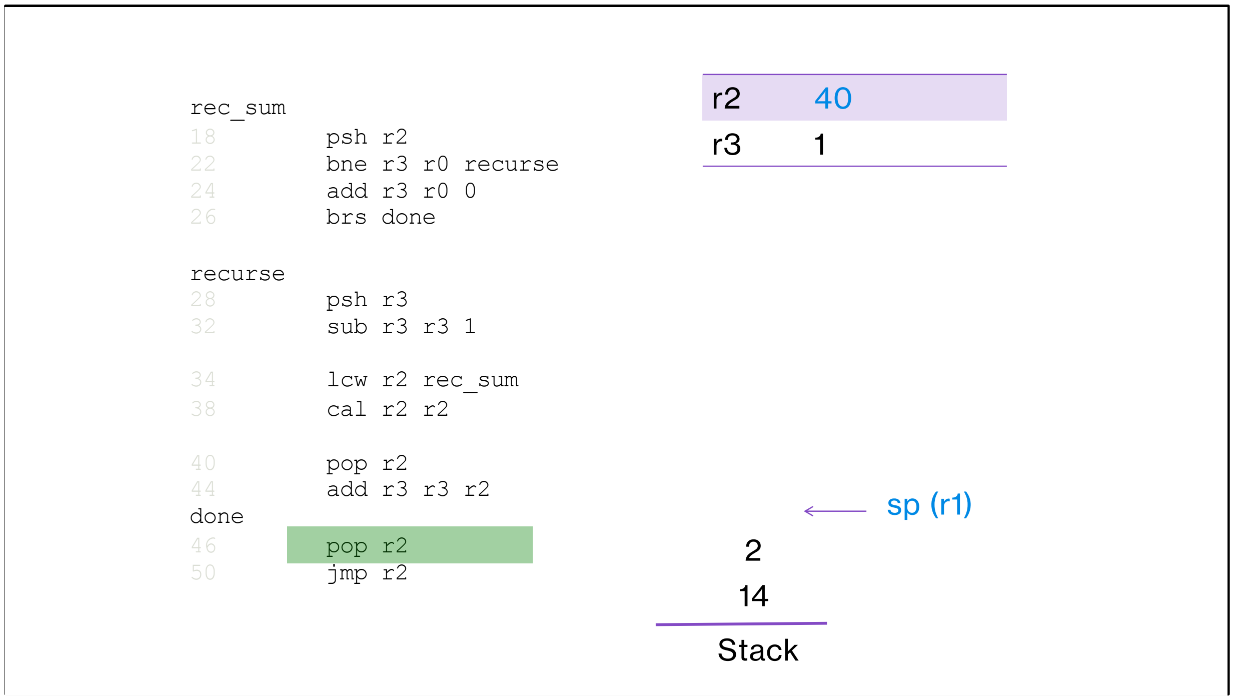
This will result in r3 being updated to 1.

```
rec_sum
18        psh r2
22        bne r3 r0 recurse
24        add r3 r0 0
26        brs done

recurse
28        psh r3
32        sub r3 r3 1

34        lcw r2 rec_sum
38        cal r2 r2

40        pop r2
44        add r3 r3 r2
done
46        pop r2
50        jmp r2
```

r2	1
r3	1



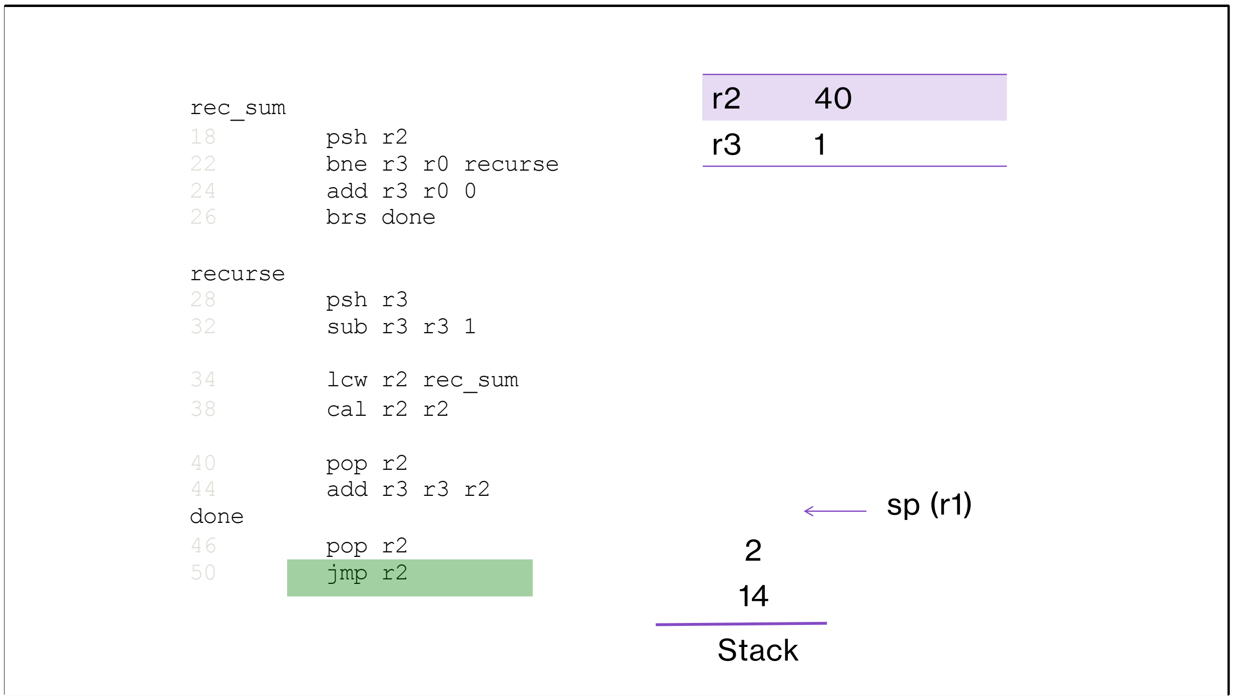
We continue sequentially the execution of our code which will take us to pop r2.



```
rec_sum
18          psh r2
22          bne r3 r0 recurse
24          add r3 r0 0
26          brs done

recurse
28          psh r3
32          sub r3 r3 1

34          lcw r2 rec_sum
38          cal r2 r2

40          pop r2
44          add r3 r3 r2
done
46          pop r2
50          jmp r2
```

r2	40
r3	1

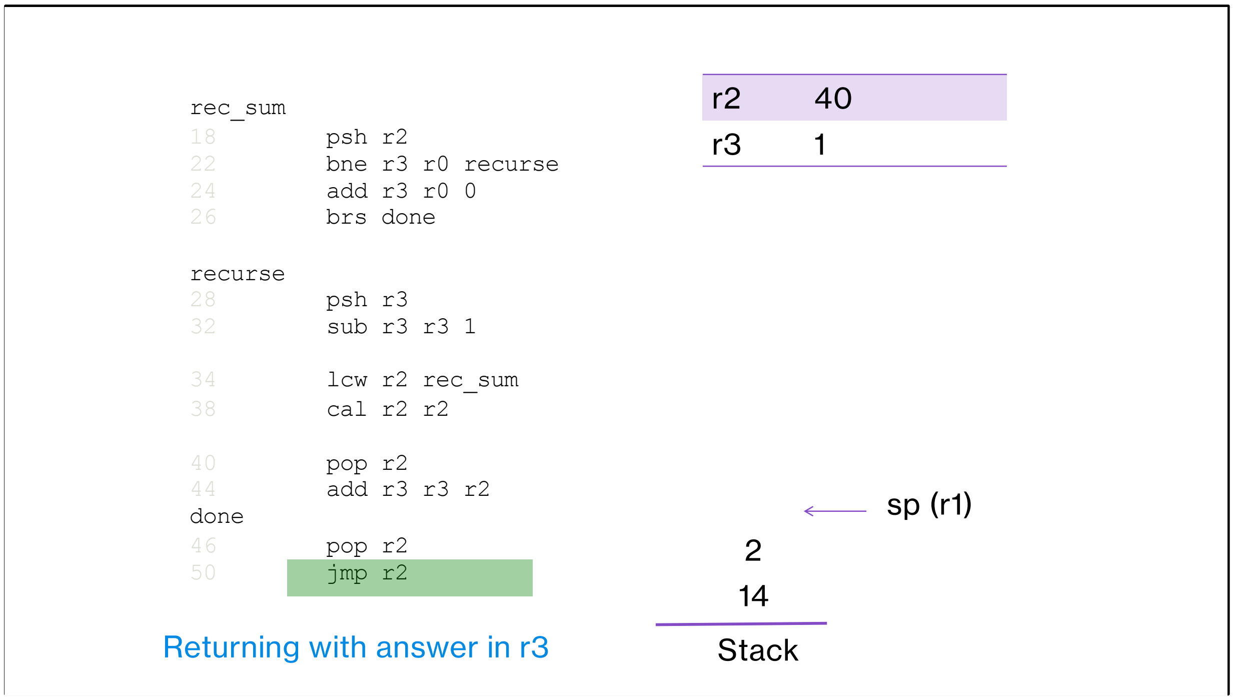
This will result in the top of the stack (40) being popped and stored in r2 and the stack pointer r1 moving.



```
rec_sum
18        psh r2
22        bne r3 r0 recurse
24        add r3 r0 0
26        brs done

recurse
28        psh r3
32        sub r3 r3 1

34        lcw r2 rec_sum
38        cal r2 r2

40        pop r2
44        add r3 r3 r2
done
46        pop r2
50        jmp r2
```

r2	40
r3	1

Next, we are in address 50 which says we should jump to the address stored in r2, that is address 40.



```
rec_sum
18         psh r2
22         bne r3 r0 recurse
24         add r3 r0 0
26         brs done

recurse
28         psh r3
32         sub r3 r3 1

34         lcw r2 rec_sum
38         cal r2 r2

40         pop r2
44         add r3 r3 r2
done
46         pop r2
50         jmp r2
```

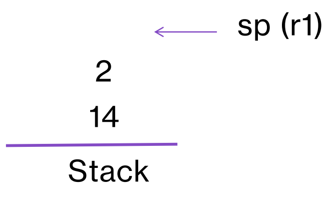
r2	40
r3	1

Returning with answer in r3

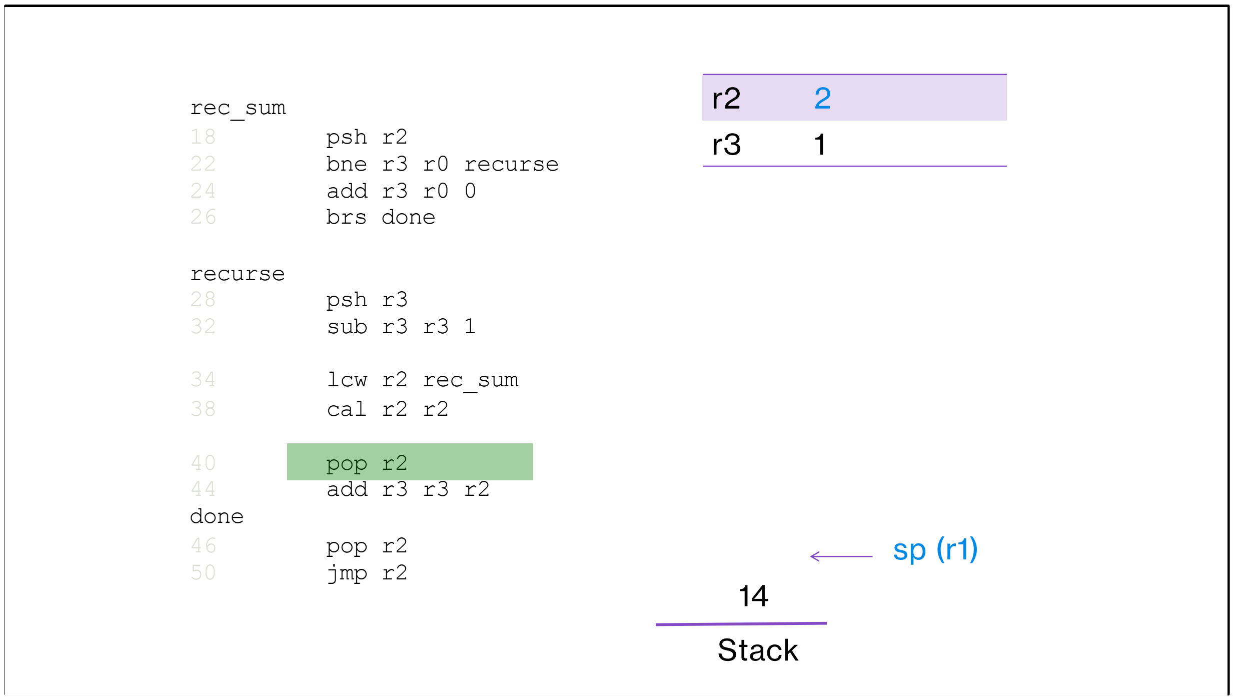
Again, our answer from the call rec_sum(1) is stored in r3 and is equal to 1.

```
rec_sum
18        psh r2
22        bne r3 r0 recurse
24        add r3 r0 0
26        brs done

recurse
28        psh r3
32        sub r3 r3 1

34        lcw r2 rec_sum
38        cal r2 r2

40        pop r2
44        add r3 r3 r2
done
46        pop r2
50        jmp r2
```

r2	40
r3	1



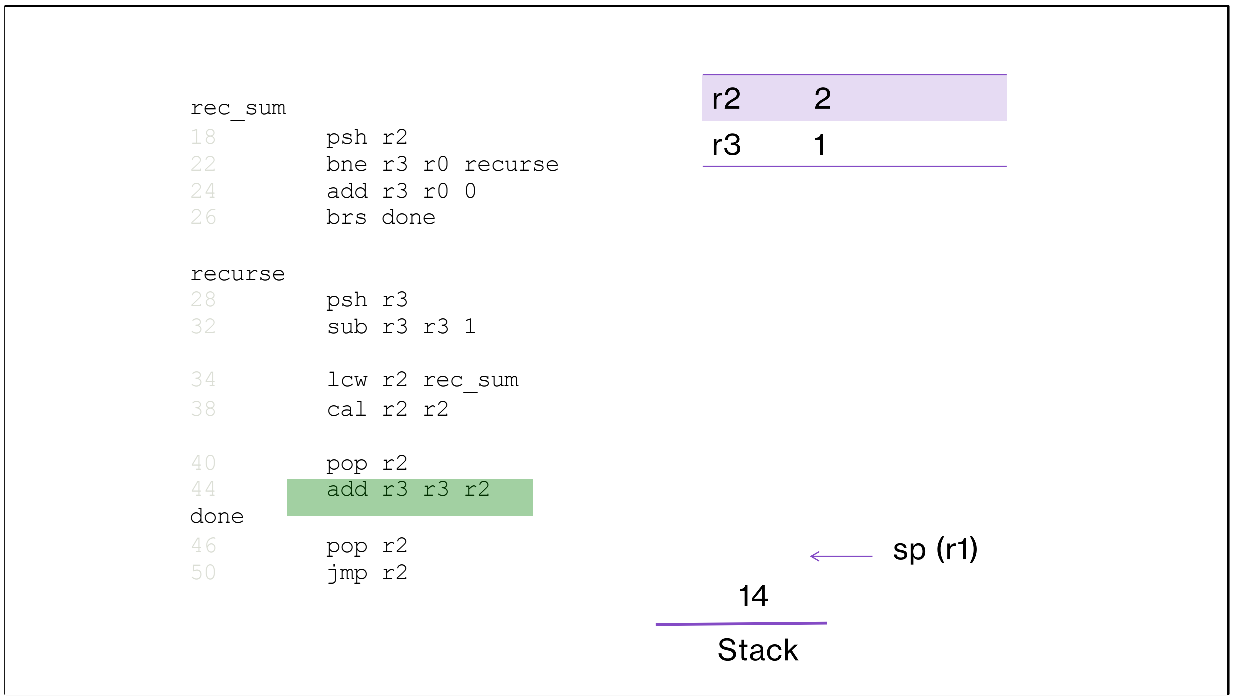
We are next asked to pop the top of the stack and store it in r2.



```
rec_sum
18        psh r2
22        bne r3 r0 recurse
24        add r3 r0 0
26        brs done

recurse
28        psh r3
32        sub r3 r3 1

34        lcw r2 rec_sum
38        cal r2 r2

40        pop r2
44        add r3 r3 r2
done
46        pop r2
50        jmp r2
```

r2	2
r3	1

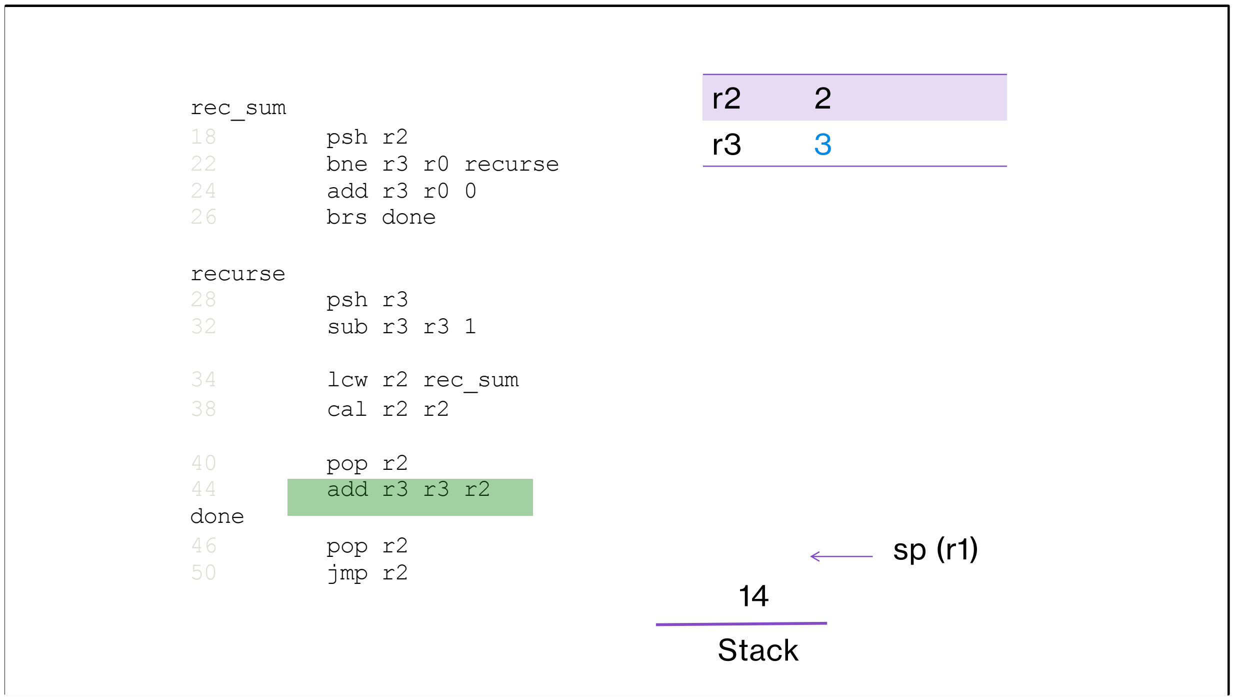
This will result in address 2 stored in r2 and the stack pointer moving.



```
rec_sum
18        psh r2
22        bne r3 r0 recurse
24        add r3 r0 0
26        brs done

recurse
28        psh r3
32        sub r3 r3 1

34        lcw r2 rec_sum
38        cal r2 r2

40        pop r2
44        add r3 r3 r2
done
46        pop r2
50        jmp r2
```

r2	2
r3	1

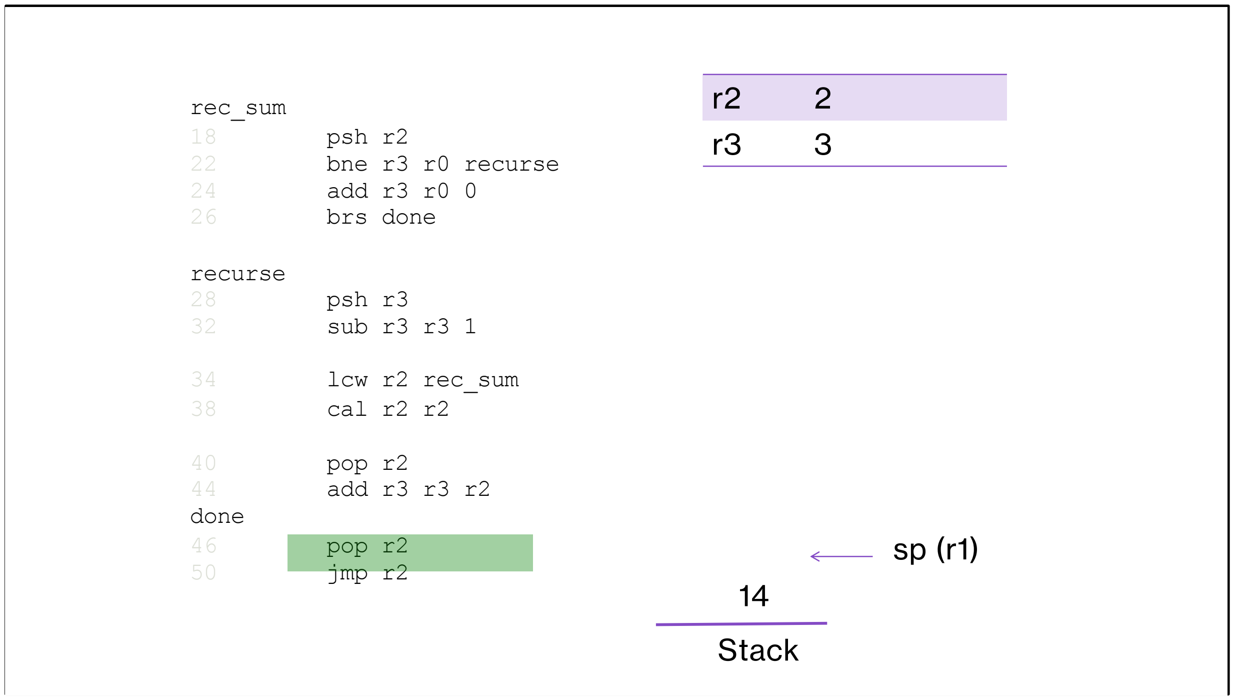
Next, we have to add to r3 r2.



```
rec_sum
18        psh r2
22        bne r3 r0 recurse
24        add r3 r0 0
26        brs done

recurse
28        psh r3
32        sub r3 r3 1

34        lcw r2 rec_sum
38        cal r2 r2

40        pop r2
44        add r3 r3 r2
done
46        pop r2
50        jmp r2
```

r2	2
r3	3

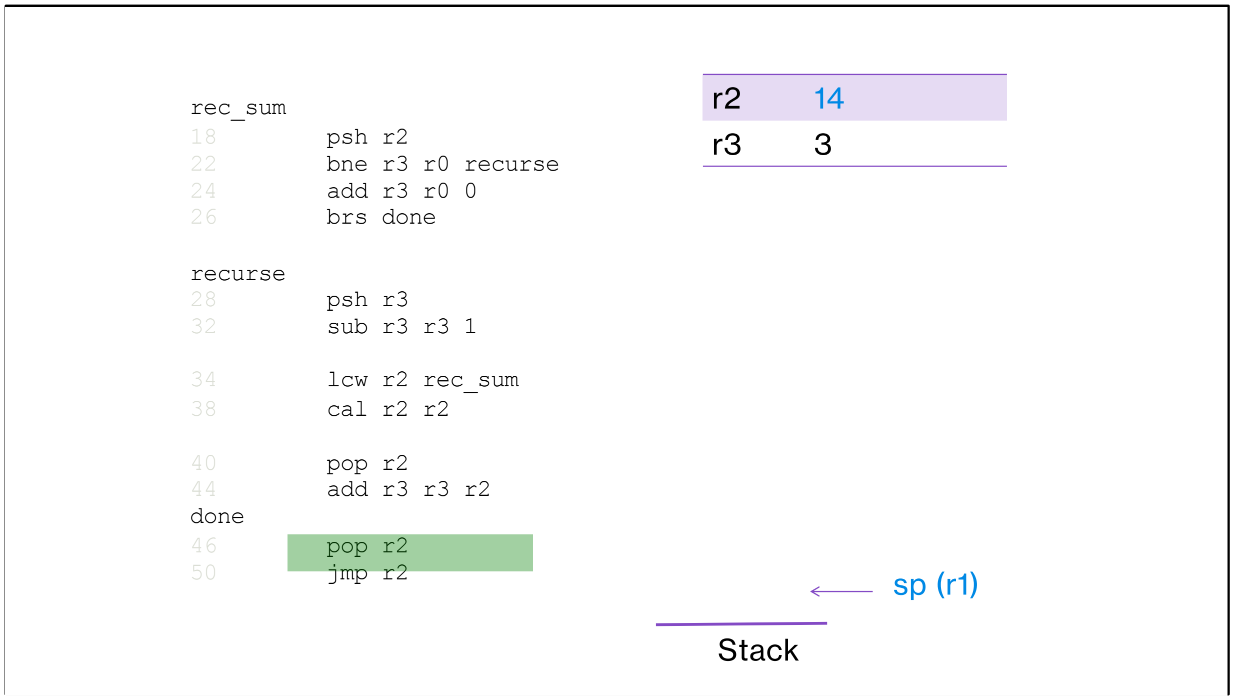
This will result in r3 having the result 3



```
rec_sum
18        psh r2
22        bne r3 r0 recurse
24        add r3 r0 0
26        brs done

recurse
28        psh r3
32        sub r3 r3 1

34        lcw r2 rec_sum
38        cal r2 r2

40        pop r2
44        add r3 r3 r2
done
46        pop r2
50        jmp r2
```

r2	2
r3	3

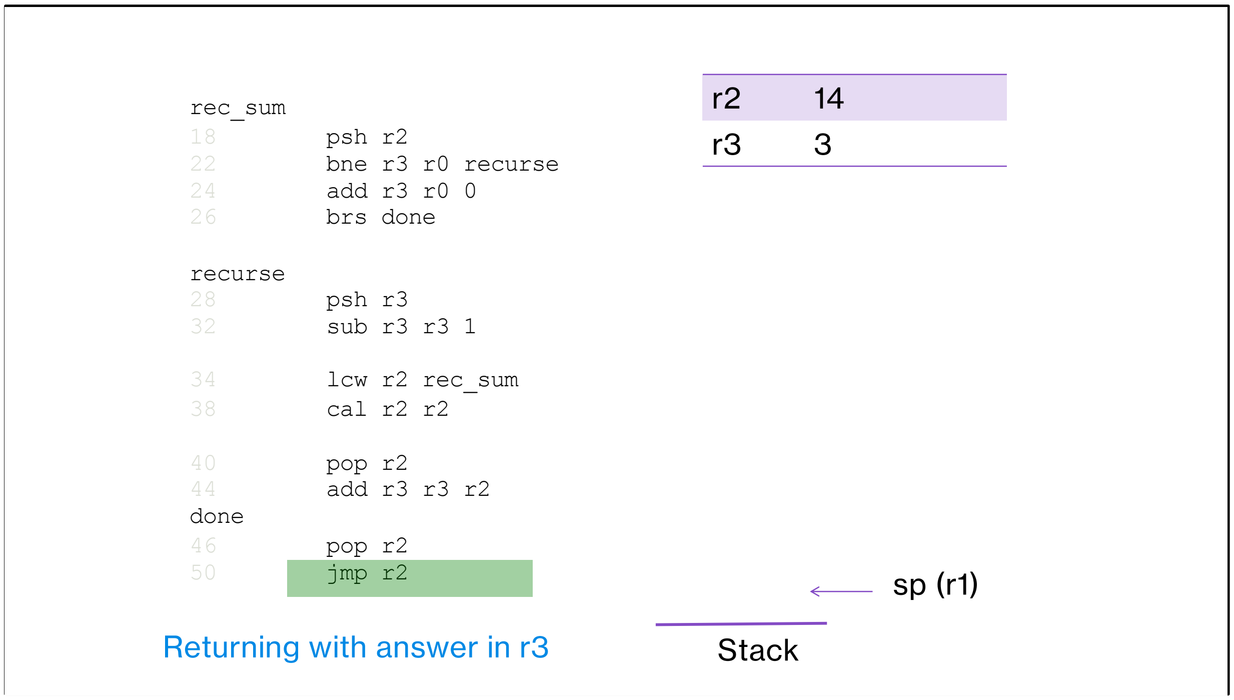
We continue to the next instruction, i.e. pop r2.



```
rec_sum
18        psh r2
22        bne r3 r0 recurse
24        add r3 r0 0
26        brs done

recurse
28        psh r3
32        sub r3 r3 1

34        lcw r2 rec_sum
38        cal r2 r2

40        pop r2
44        add r3 r3 r2
done
46        pop r2
50        jmp r2
```

r2	14
r3	3

This will result in 14 being popped and stored in r2 and r1 moving.

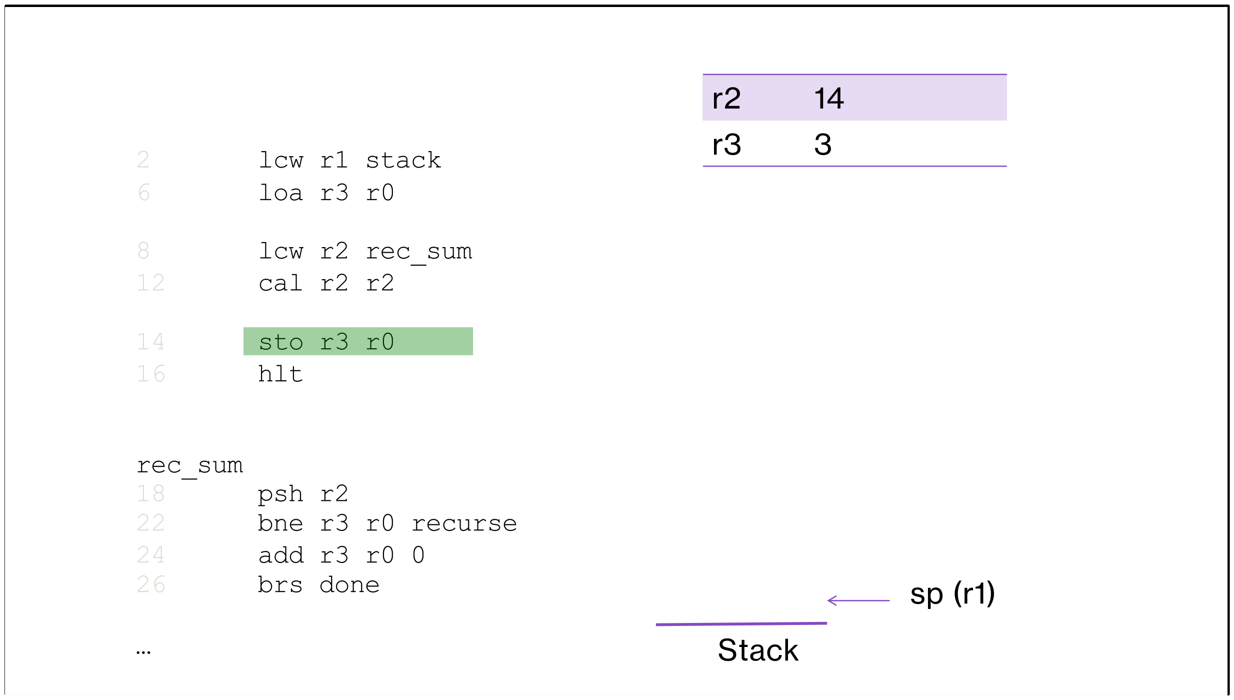


```
rec_sum
18          psh r2
22          bne r3 r0 recurse
24          add r3 r0 0
26          brs done

recurse
28          psh r3
32          sub r3 r3 1

34          lcw r2 rec_sum
38          cal r2 r2

40          pop r2
44          add r3 r3 r2
done
46          pop r2
50          jmp r2
```

r2	14
r3	3

Returning with answer in r3

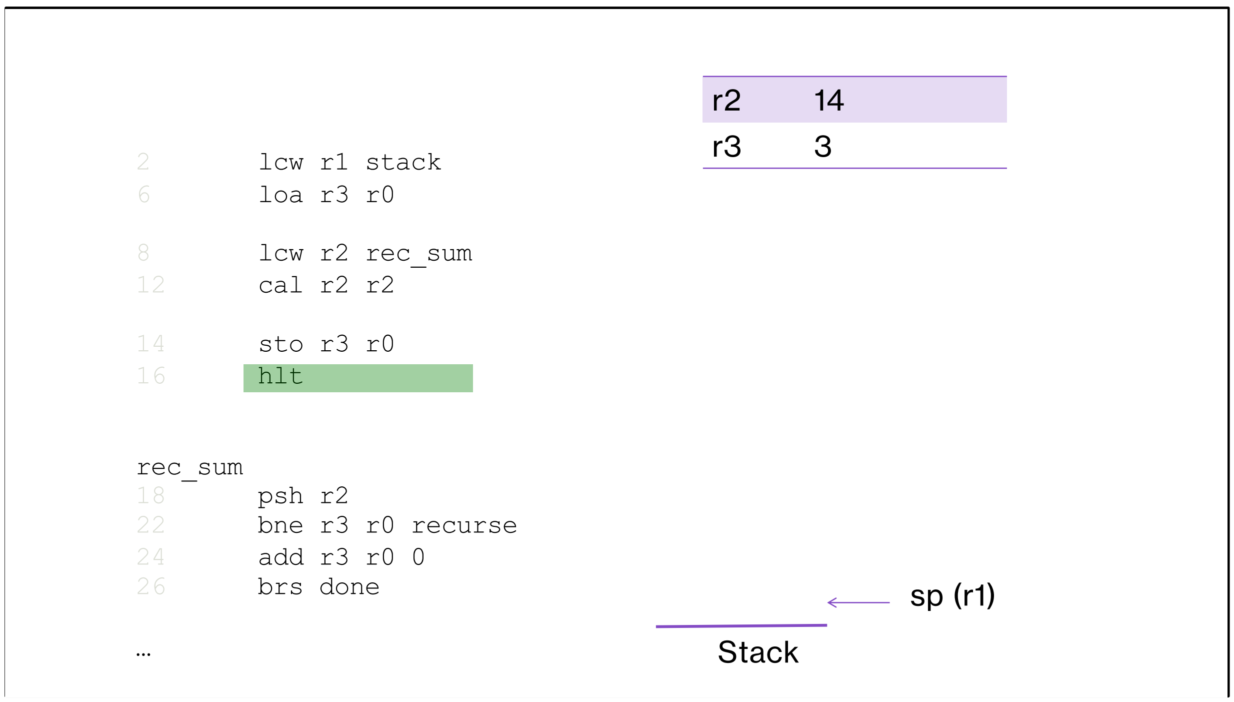
We then are asked to jump to the address stored in register r2, that is to address 14. Along the way, we have our ultimate answer in r3!



```
2       lcw r1 stack
6       loa r3 r0

8       lcw r2 rec_sum
12      cal r2 r2

14      sto r3 r0
16      hlt


rec_sum
18      psh r2
22      bne r3 r0 recurse
24      add r3 r0 0
26      brs done

…
```

r2	14
r3	3

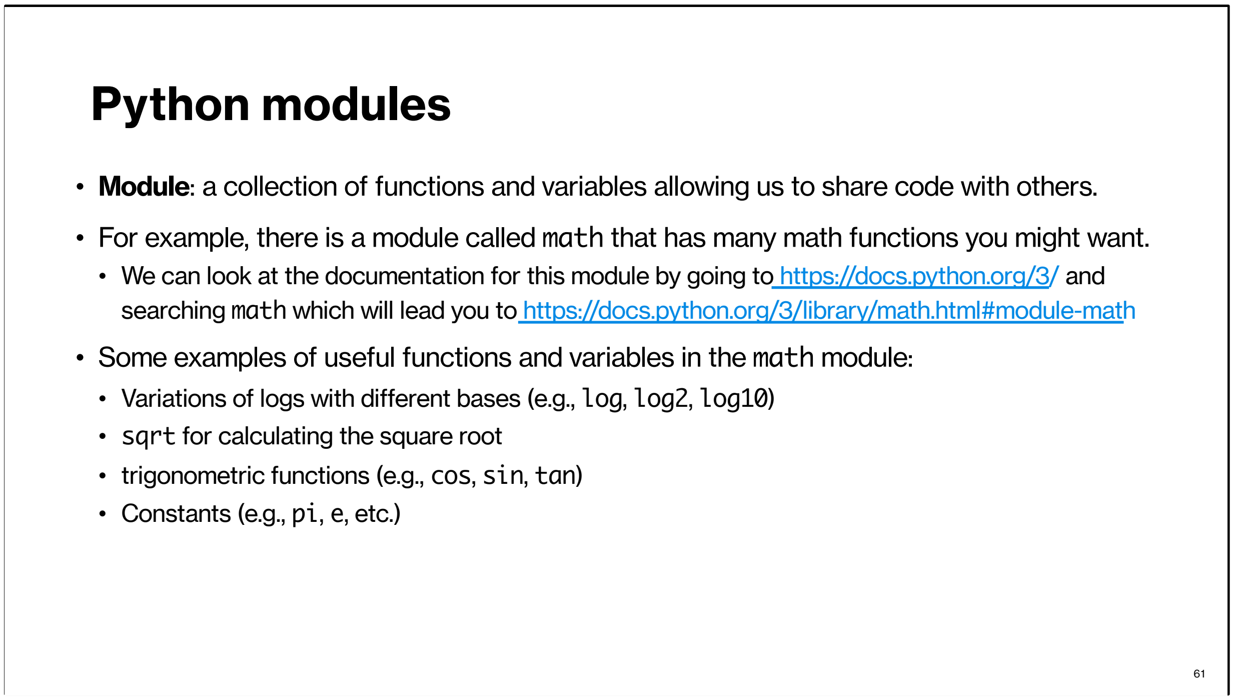
We are now ready to print the answer (3) to the output.



r2	14
r3	3

```
2       lcw r1 stack
6       loa r3 r0

8       lcw r2 rec_sum
12      cal r2 r2

14      sto r3 r0
16      hlt


rec_sum
18      psh r2
22      bne r3 r0 recurse
24      add r3 r0 0
26      brs done

…
```

And proceed with halting our program. TADA!!!

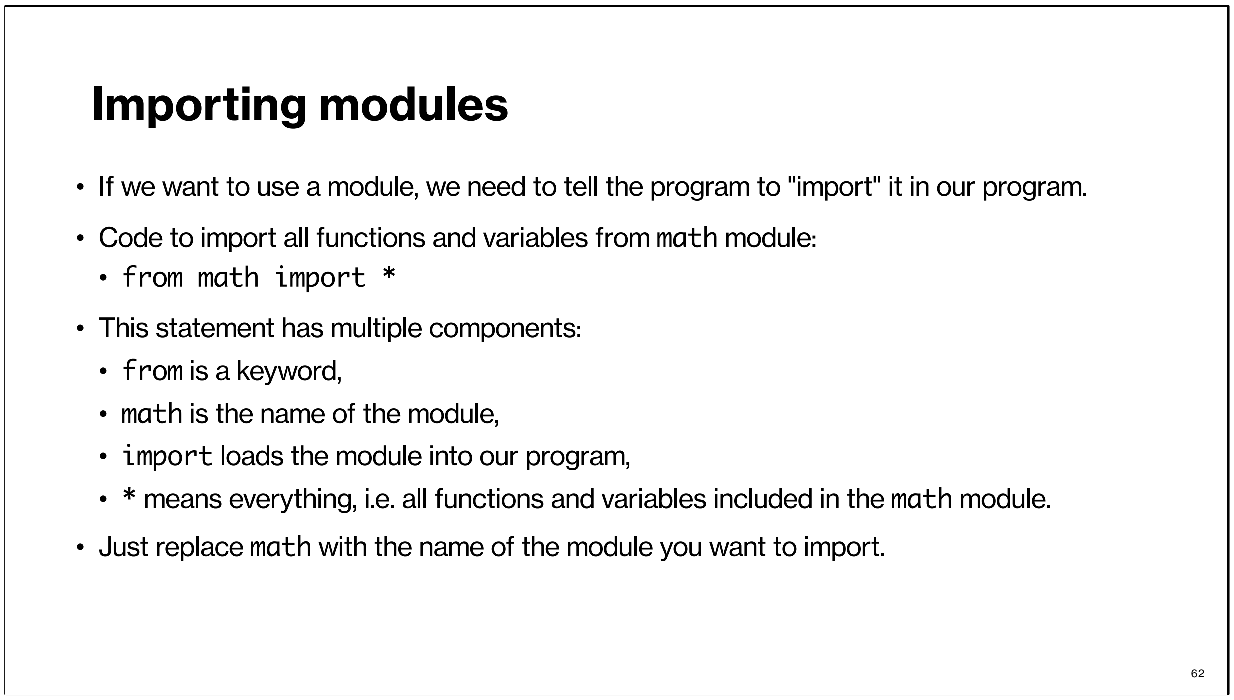


Python modules

- **Module**: a collection of functions and variables allowing us to share code with others.
- For example, there is a module called `math` that has many math functions you might want.
  - We can look at the documentation for this module by going to https://docs.python.org/3/ and searching `math` which will lead you to https://docs.python.org/3/library/math.html#module-math
- Some examples of useful functions and variables in the `math` module:
  - Variations of logs with different bases (e.g., `log`, `log2`, `log10`)
  - `sqrt` for calculating the square root
  - trigonometric functions (e.g., `cos`, `sin`, `tan`)
  - Constants (e.g., `pi`, e, etc.)

61

In Python, a module is a collection of functions and variables. Modules allows us to use code that other people have written and enable us to share our own code. For example, there is a module called math that has many of the math functions you might want. If you look up the official documentation of Python, you will discover that the math module offers you access to all sorts of functions, such as logarithmic ones, square root, trigonometric functions, and constants such as PI, e, etc.



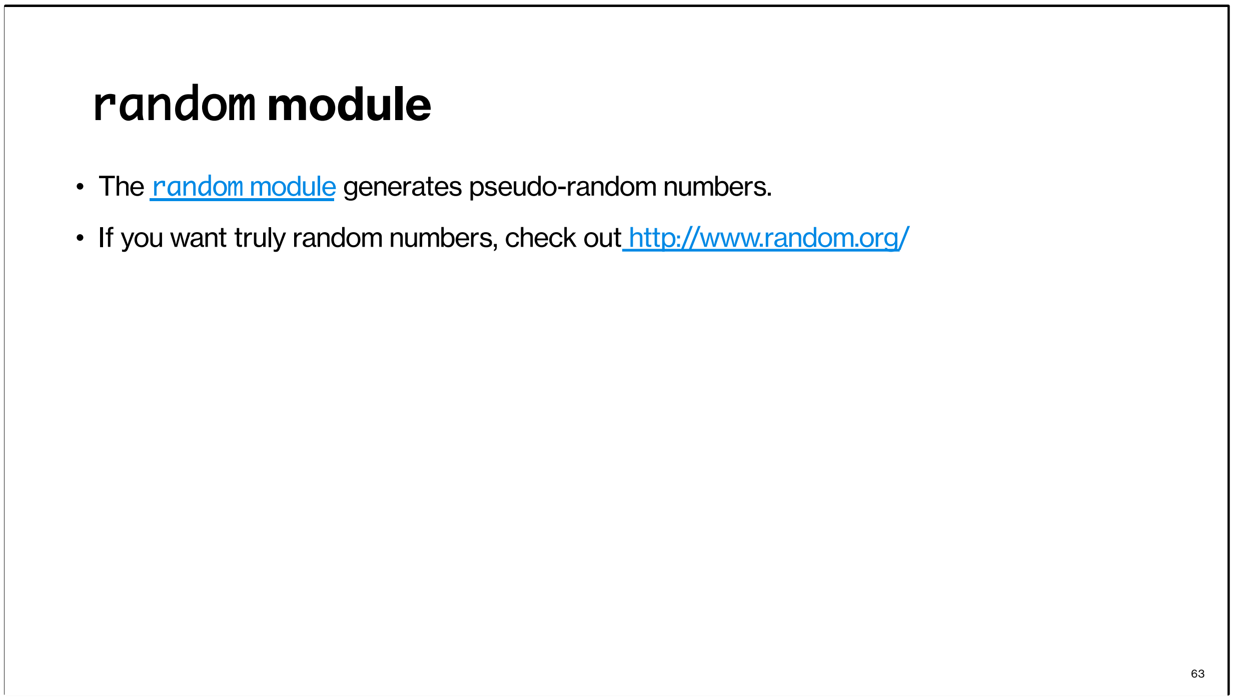
Importing modules

- If we want to use a module, we need to tell the program to "import" it in our program.
- Code to import all functions and variables from `math` module:
  - `from math import *`
- This statement has multiple components:
  - `from` is a keyword,
  - `math` is the name of the module,
  - `import` loads the module into our program,
  - `*` means everything, i.e. all functions and variables included in the `math` module.
- Just replace `math` with the name of the module you want to import.

62

If we want to use a module, we need to tell the program to include it with our program. We would type from math import *.
This statement has multiple components:
from is a keyword,
math is the name of the module,
import loads the module into our program,

- means everything, i.e. load everything included in the math module.
- Just replace `math` with the name of the module you want to import.



`random` module

- The `random` module generates pseudo-random numbers.
- If you want truly random numbers, check out http://www.random.org/

63

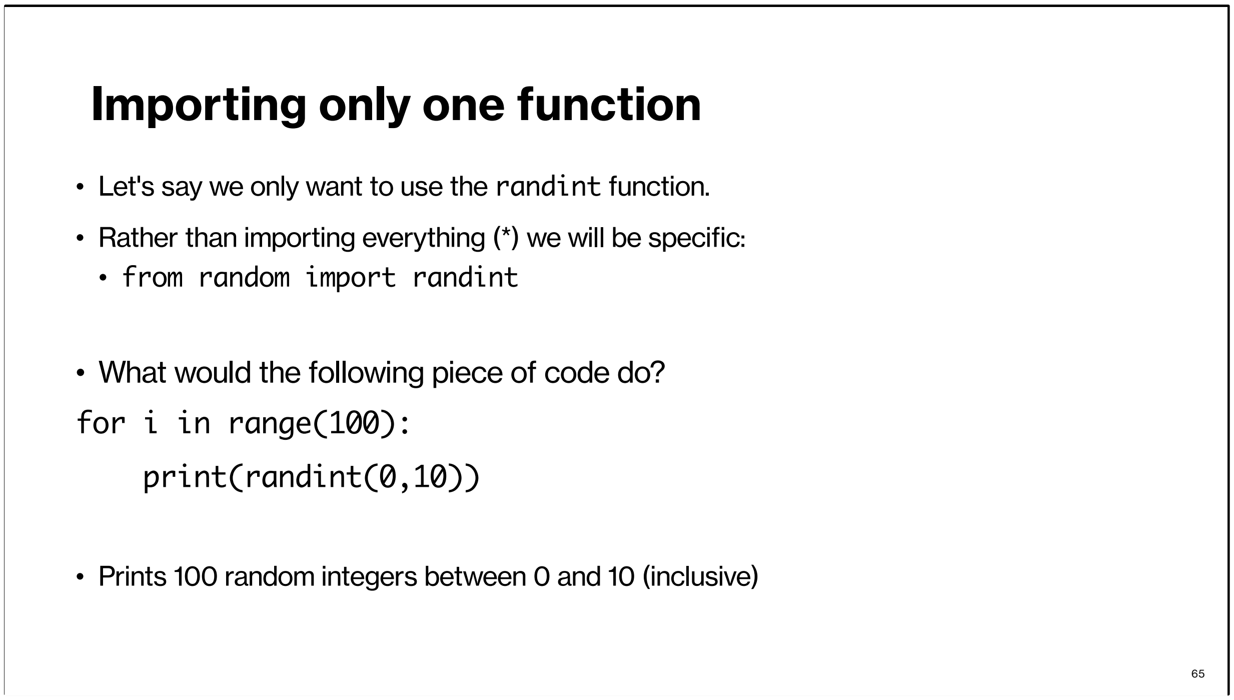
Another module that will be handy is the the random module. The random module contains functions that generate pseudo-random numbers. Why pseudo (==not genuine)? The numbers are not technically random, they are generated by an algorithm but for most purposes, including ours, they are random enough. If you want truly random numbers check out the website http://www.random.org/

Useful `random` functions

- `random()` - returns a random float between 0 and 1.
- `uniform(a, b)` - returns a random float between a and b.
- `randint(a, b)` - returns a random integer between a and b.

64

Some functions that you might find useful are random (returns a number between [0,1), uniform (returns a float between [a,b] or [a,b) depending on rounding), and randint (returns an int between [a,b]), as well as samples from well-known distributions.



Importing only one function

- Let's say we only want to use the `randint` function.
- Rather than importing everything (*) we will be specific:
  - `from random import randint`

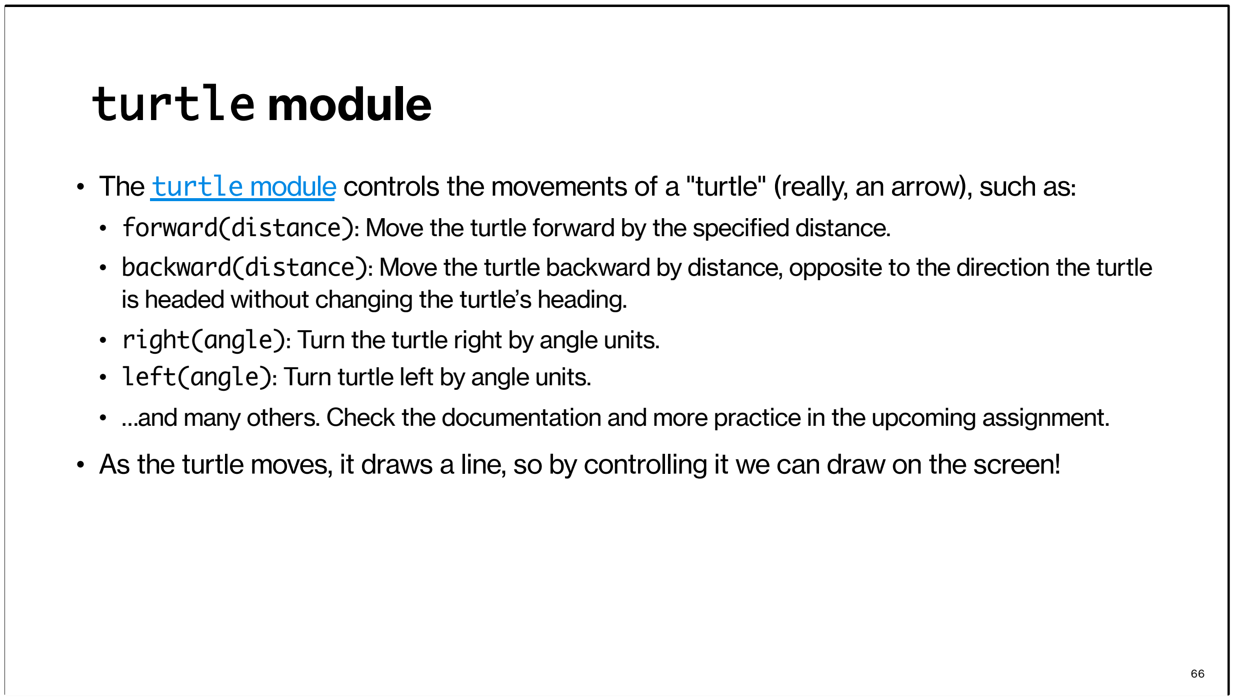
- What would the following piece of code do?

```
for i in range(100):
    print(randint(0,10))
```

- Prints 100 random integers between 0 and 10 (inclusive)

65

Let's say we only want to use the `randint` function.
. Instead of loading all functions, we can be specific: from random import randint.



`turtle` module

- The `turtle` module controls the movements of a "turtle" (really, an arrow), such as:
  - `forward(distance)`: Move the turtle forward by the specified distance.
  - `backward(distance)`: Move the turtle backward by distance, opposite to the direction the turtle is headed without changing the turtle's heading.
  - `right(angle)`: Turn the turtle right by angle units.
  - `left(angle)`: Turn turtle left by angle units.
  - …and many others. Check the documentation and more practice in the upcoming assignment.
- As the turtle moves, it draws a line, so by controlling it we can draw on the screen!

66

Time to have fun with a cool module called turtle…The turtle module implements a set of commands similar to the Logo programming language
The basic idea is that you control the movements of a turtle (in our case, it will be an arrow) through basic commands such as:
forward
backward
right
left
…and many others
As the turtle moves, it draws a line behind it, so by giving it different commands, we can draw things on the screen!
Check the documentation for the turtle class online
You'll be getting more comfortable with this documentation in the upcoming assignment.

Open VS Code

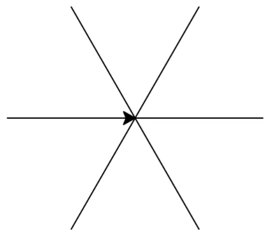
- Go to the Python shell at the bottom
- `python3`
- `>>> from turtle import *`

OK let's give it a try together. If you have your laptop, open VS Code and type python3.
Then from turtle import *

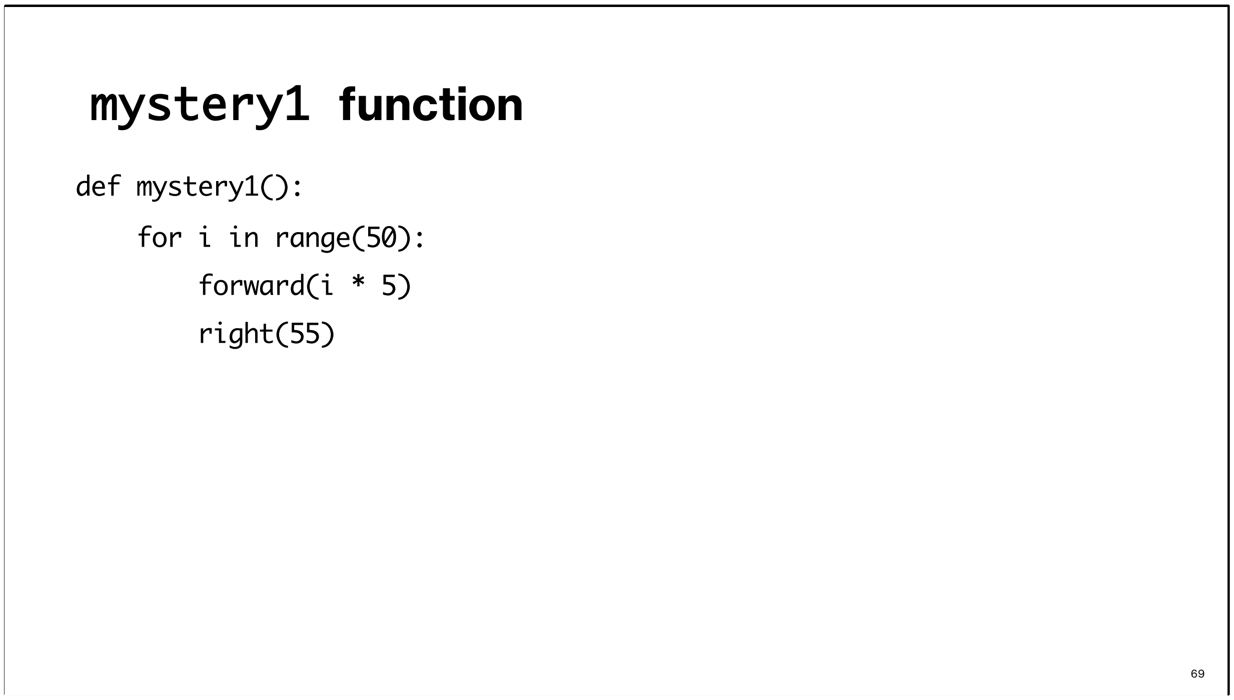
Note that if you try to import the turtle module on macOS and receive the following error "import _tkinter # If this fails your Python may not be configured for Tk", you will need to go https://www.python.org/downloads/release/python-3143/ download again python, install it, and then open a Terminal and write
PATH="/Library/Frameworks/Python.framework/Versions/3.14/bin:${PATH}"
export PATH

A star with a different number of spokes

```
def asterisk_star(length, spokes):
    angle = 360 / spokes
    for i in range(spokes):
        forward(length)
        backward(length)
        right(angle)
```



Here's an example of a function that uses the turtle module. It creates a star/asterisk with a different number of spokes. We first figure out how we have to space the spokes. We do a for loop over the number of spokes. At each iteration we draw a spoke, go backwards for the next spoke, rotate right based on the angle we calculated. Here’s the result of asterisk_star(100, 6)



`mystery1` function

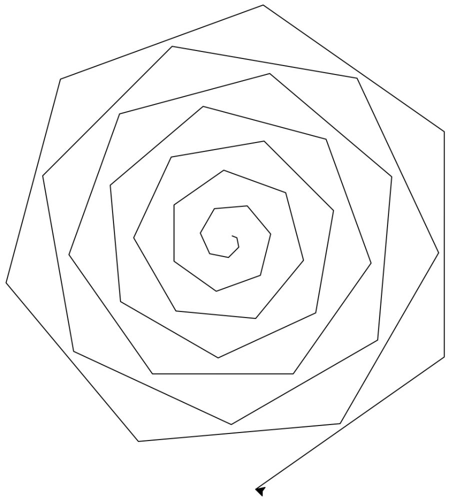
```
def mystery1():
    for i in range(50):
        forward(i * 5)
        right(55)
```

69

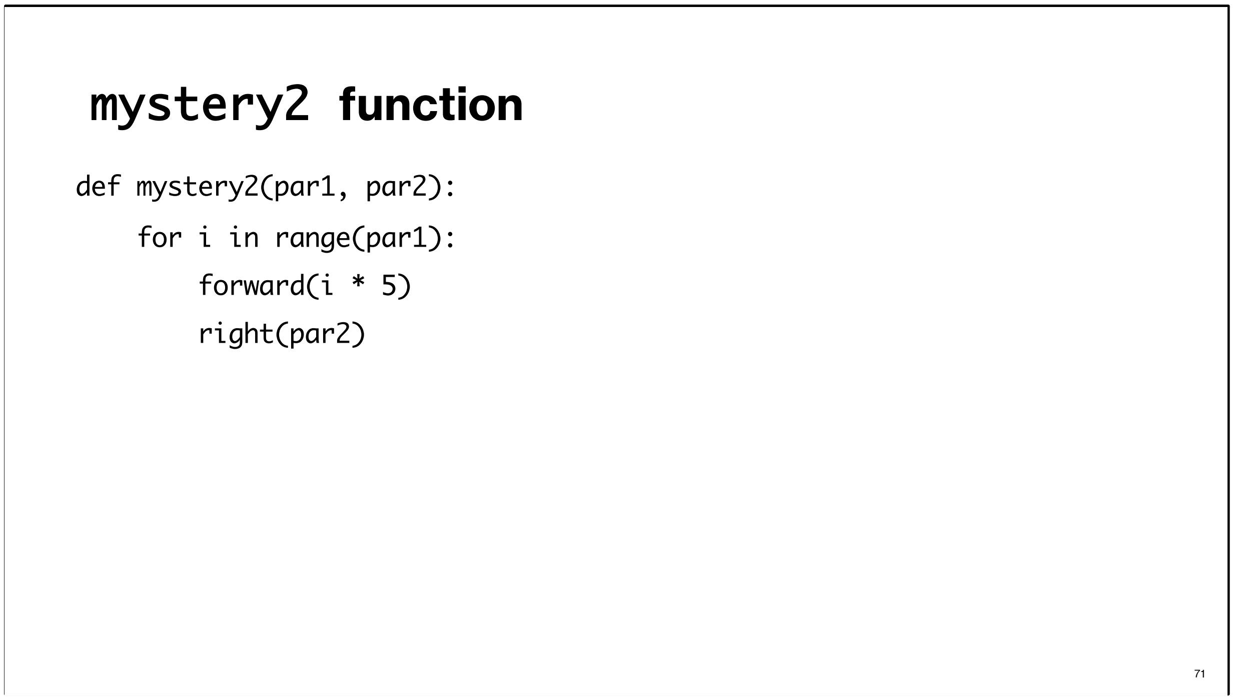
What do you think this mystery function does?

`simple_spiral` function

```
def simple_spiral():
    for i in range(50):
        forward(i * 5)
        right(55)
```



It draws a simple spiral! Look at the code: The variable i will keep count of how many times we need to get in the loop. For small loops (i.e. just a handful of statements), it's common to just name the counter variable as i which stands for index. Each time we get in the loop, the length of the edge drawn will be longer by 5: 0, 5, 10, 15, 20, ... and it will be at a 50 degree angle. If it is less than 90 degrees, it will spiral out, and above it, it will spiral in.

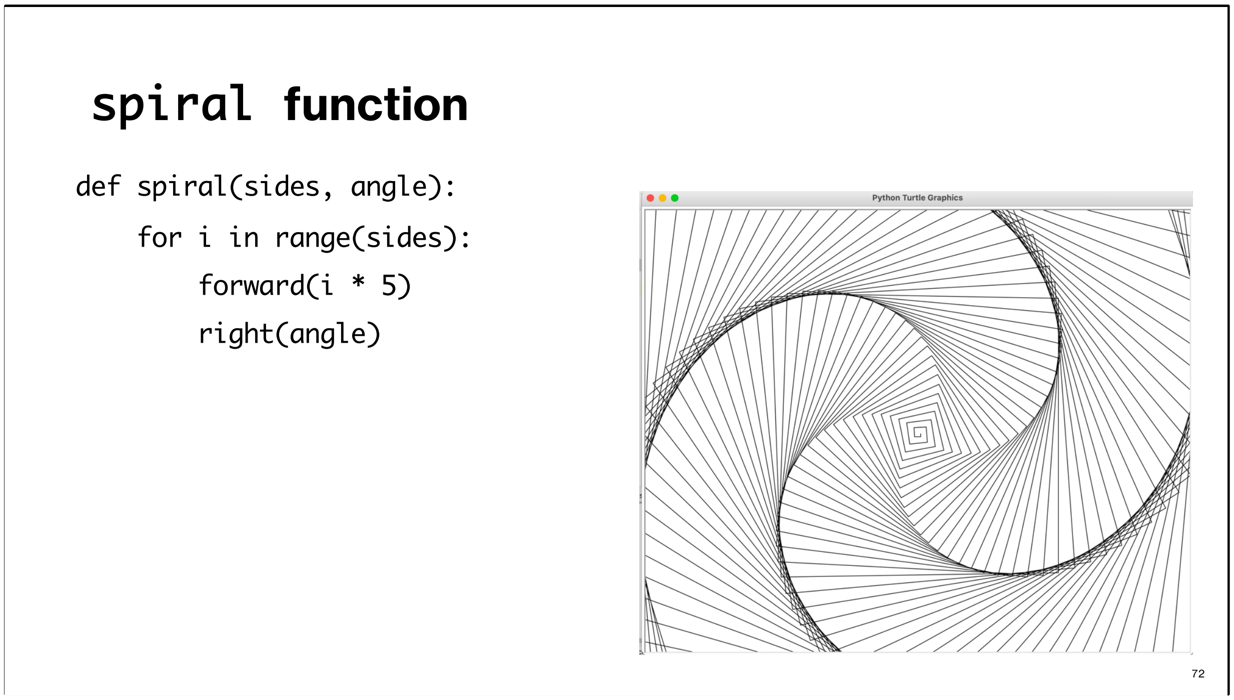


`mystery2` function

```
def mystery2(par1, par2):
    for i in range(par1):
        forward(i * 5)
        right(par2)
```

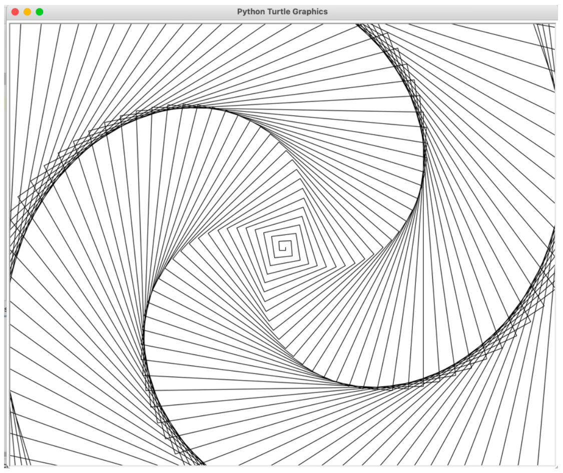
71

What about this mystery function

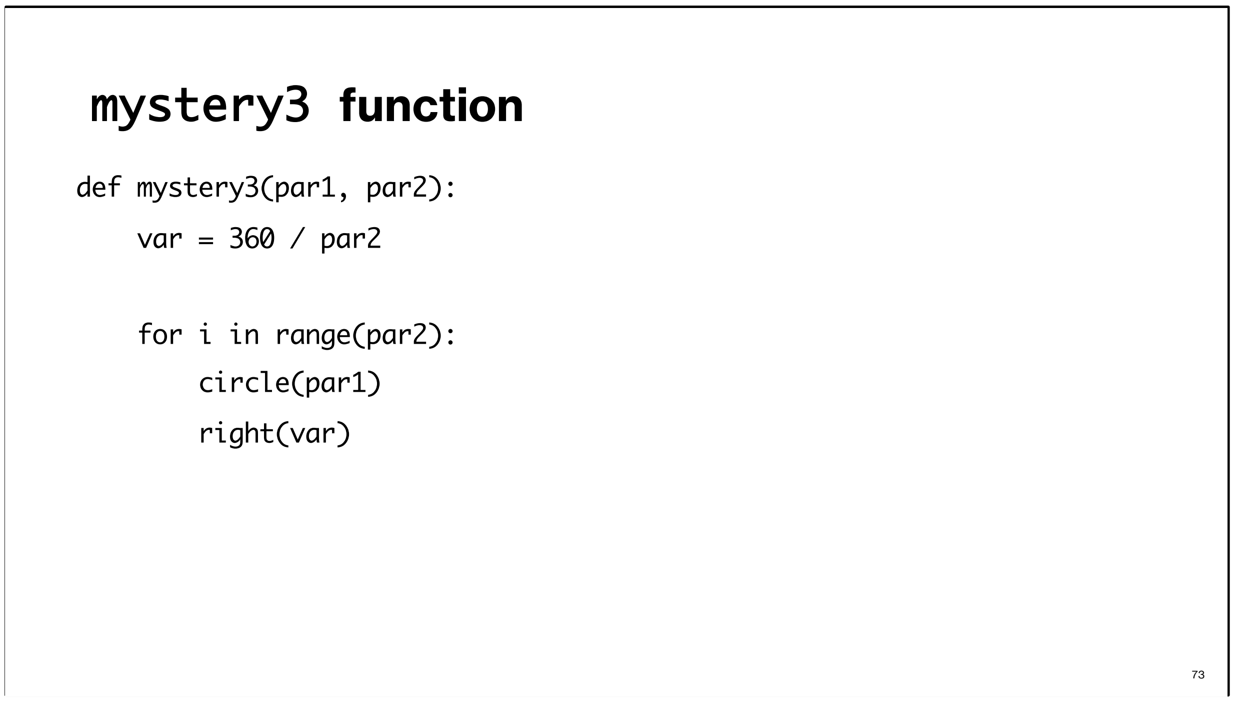


`spiral` function

```
def spiral(sides, angle):
    for i in range(sides):
        forward(i * 5)
        right(angle)
```



The spiral function is similar to simple_spiral, however now we've parameterized the length of the sides and the angle. A good example, is side = 200 and angle = 89

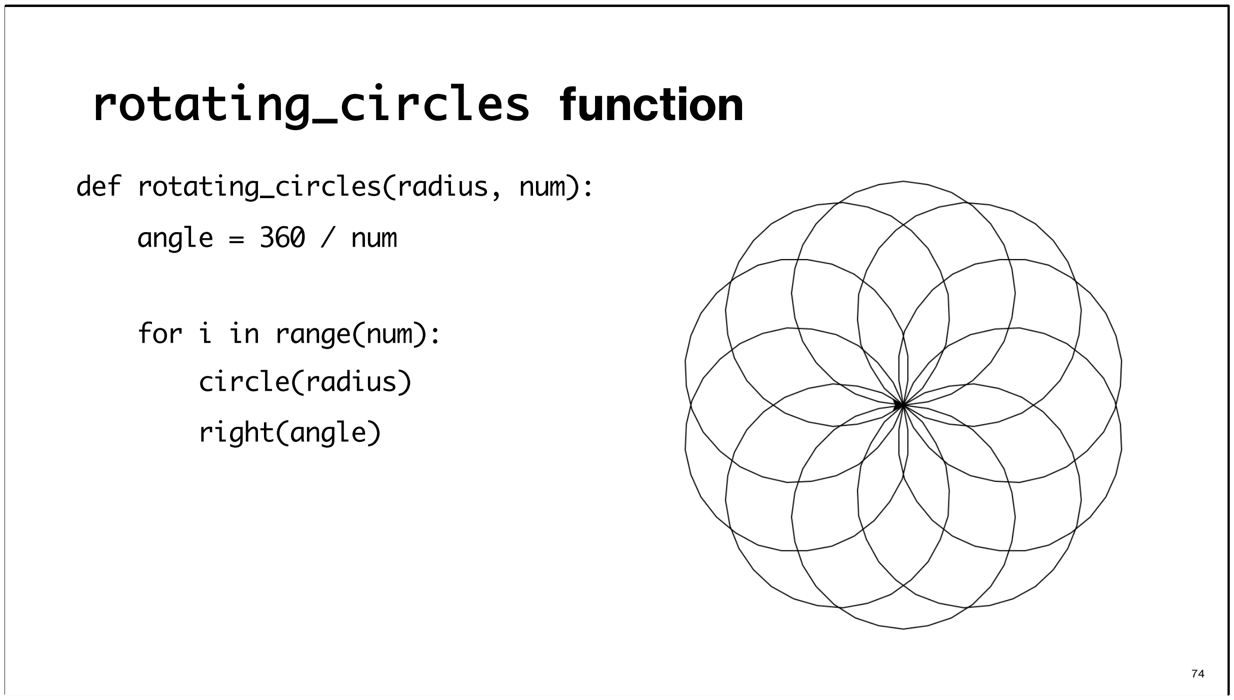


`mystery3` function

```
def mystery3(par1, par2):
    var = 360 / par2

    for i in range(par2):
        circle(par1)
        right(var)
```

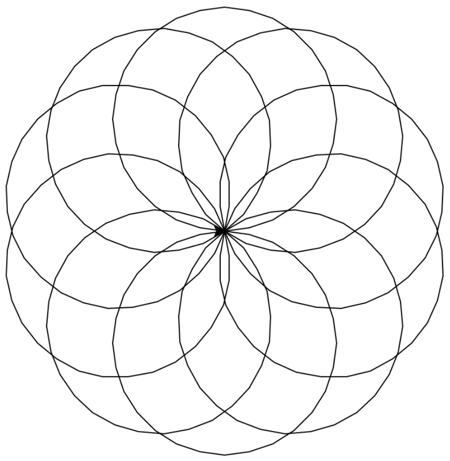
73

What about this function?



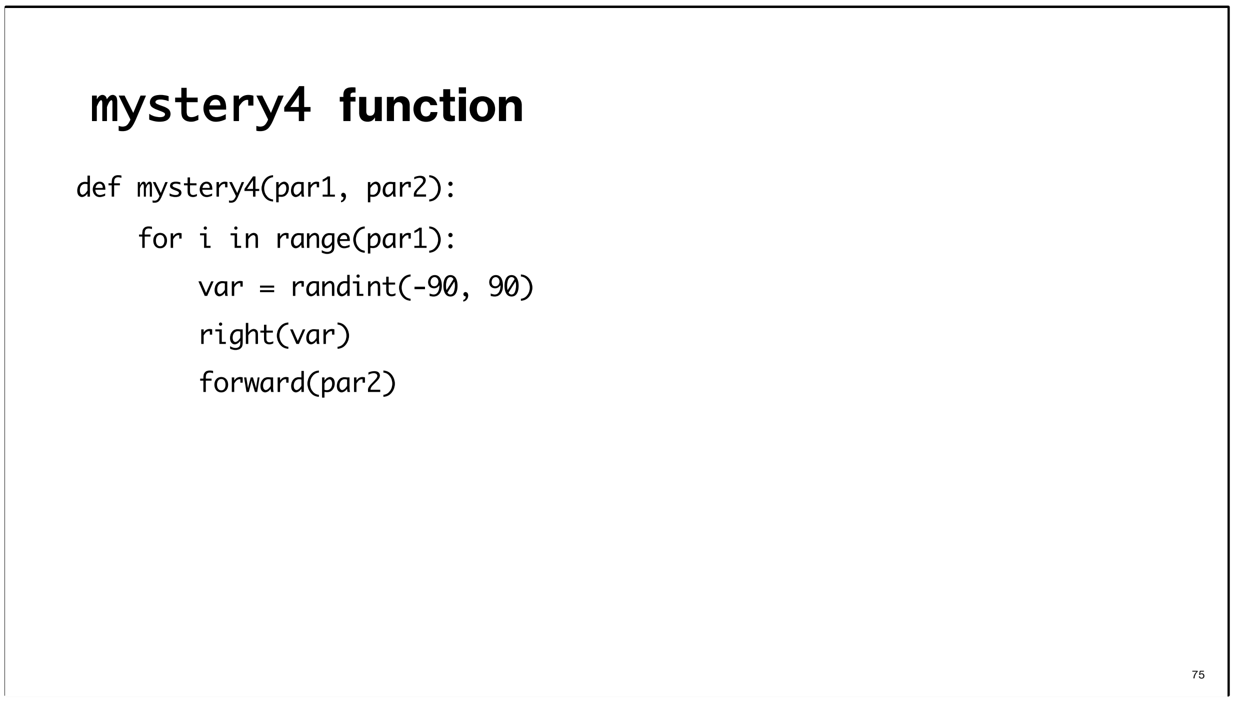
`rotating_circles` function

```
def rotating_circles(radius, num):
    angle = 360 / num

    for i in range(num):
        circle(radius)
        right(angle)
```



74

Finally, rotating_circles is another fun function that draws “num” circles, each one rotated "angle" degrees from the previous one. This is the result of the rotating_circles(100, 10) call.



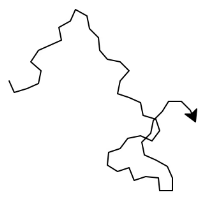
`mystery4` function

```
def mystery4(par1, par2):
    for i in range(par1):
        var = randint(-90, 90)
        right(var)
        forward(par2)
```

What about this function?

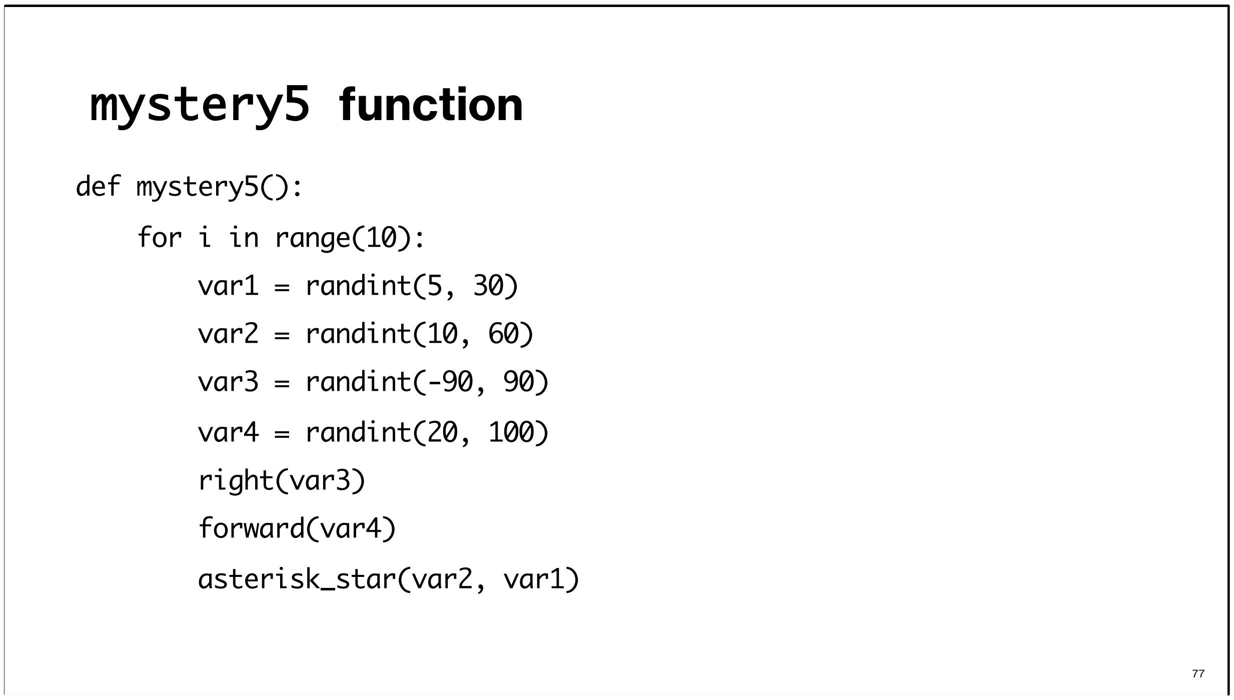
walk function

```
def walk(num_steps, step_size):
    for i in range(num_steps):
        angle = randint(-90, 90)
        right(angle)
        forward(step_size)
```



76

It draws a random walk: It turns a random angle between -90 and 90 and steps forward some step size for num_steps.

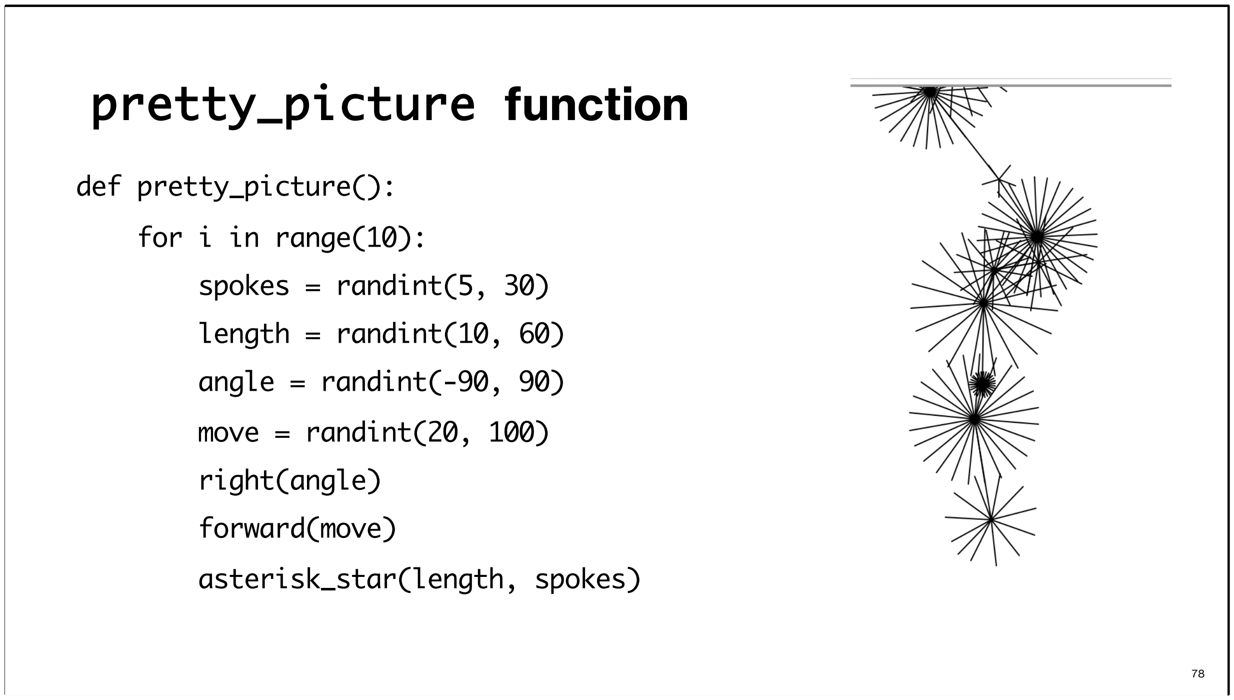


mystery5 function

```
def mystery5():
    for i in range(10):
        var1 = randint(5, 30)
        var2 = randint(10, 60)
        var3 = randint(-90, 90)
        var4 = randint(20, 100)
        right(var3)
        forward(var4)
        asterisk_star(var2, var1)
```

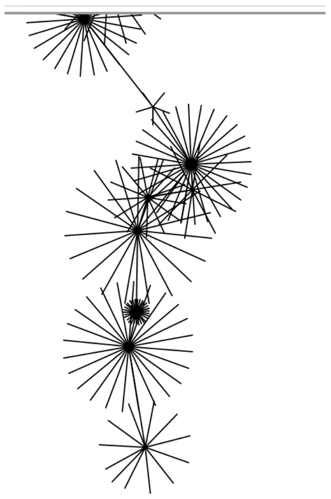
77

What about this mystery function?



pretty_picture function

```
def pretty_picture():
    for i in range(10):
        spokes = randint(5, 30)
        length = randint(10, 60)
        angle = randint(-90, 90)
        move = randint(20, 100)
        right(angle)
        forward(move)
        asterisk_star(length, spokes)
```



78

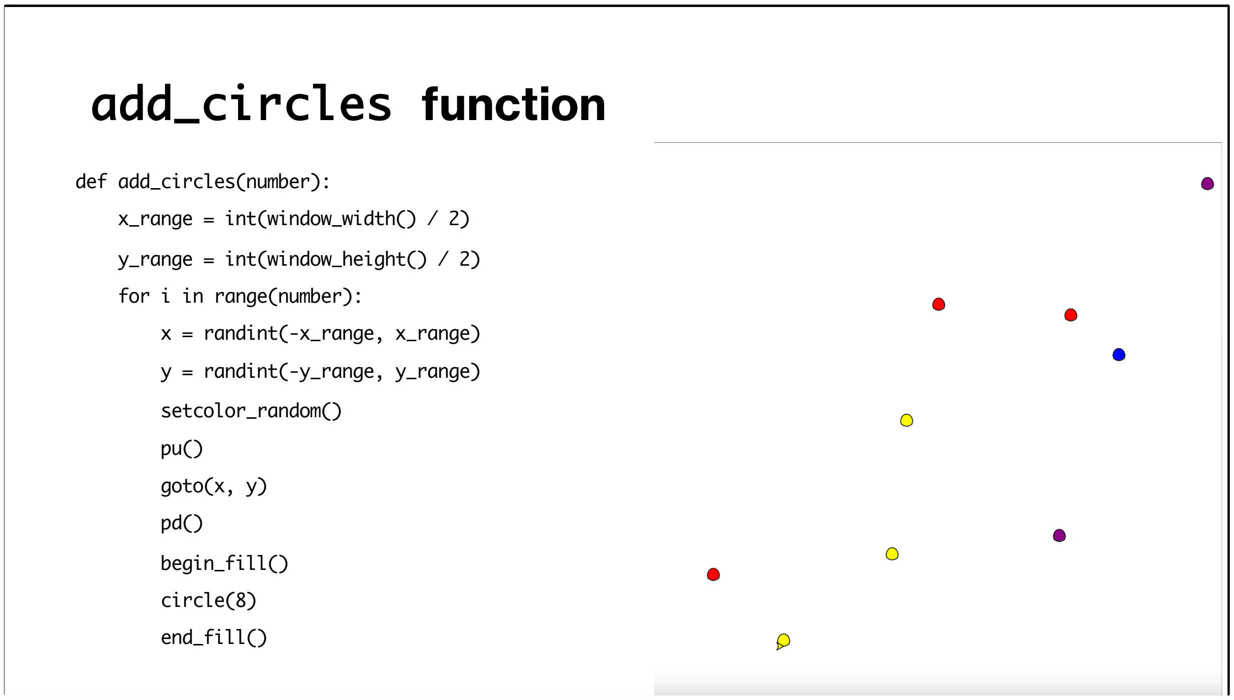
It draws a line of length between 10 and 60 at an angle between -90 and 90 then draws a star of length between 10 and 60 and with “spokes” spokes. It does that 10 times. Pretty, right?

`setcolor_random` function

```
def setcolor_random():
    color = randint(1, 4)
    if color == 1:
        fillcolor("blue")
    elif color == 2:
        fillcolor("purple")
    elif color == 3:
        fillcolor("red")
    else:  # color == 4
        fillcolor("yellow")
```

79

What about the setcolor_random function? It randomly picks between blue, purple, red and yellow (instead of based on x, y). Any ideas on how could we get this behavior? We will use random.randint to select a number between 1 and 4. We will save this number and use it in an if-elif-else statement. We MUST save this number to a variable and not try and do your if/else statement based on new calls to random.randint!!!



add_circles function

```
def add_circles(number):
    x_range = int(window_width() / 2)
    y_range = int(window_height() / 2)
    for i in range(number):
        x = randint(-x_range, x_range)
        y = randint(-y_range, y_range)
        setcolor_random()
        pu()
        goto(x, y)
        pd()
        begin_fill()
        circle(8)
        end_fill()
```

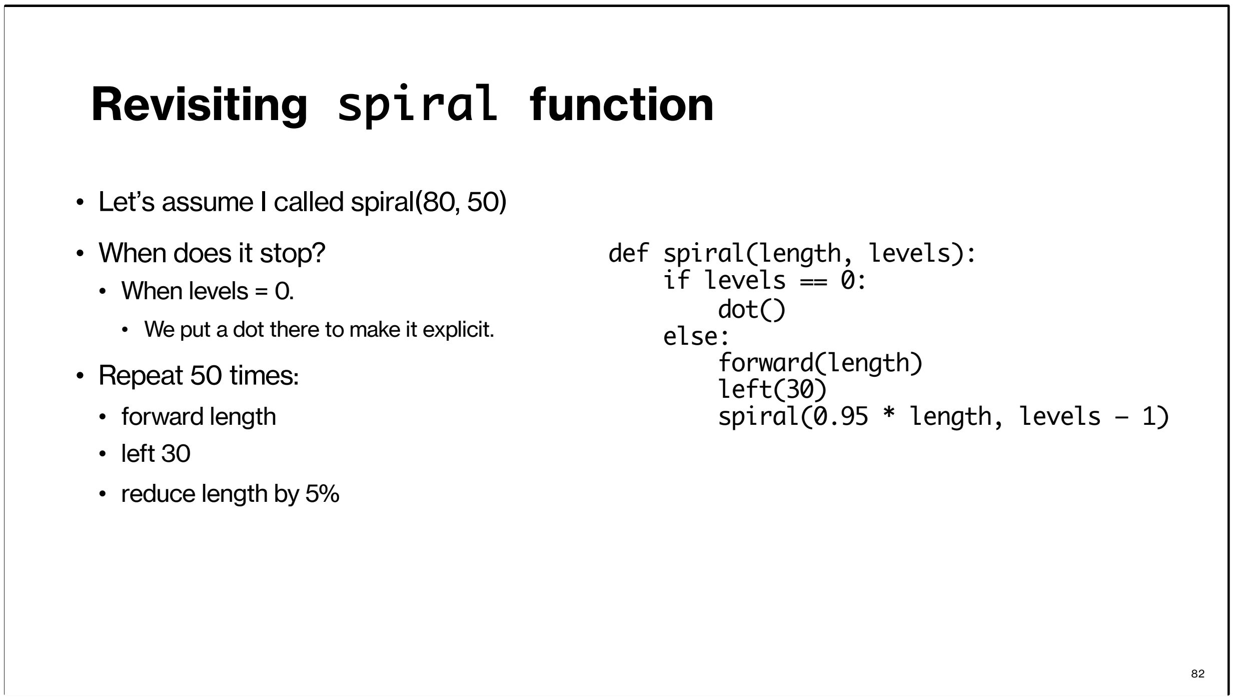
It picks random x and y coordinates to draw a circle. It uses the randint function which we have seen before. Now how are the colors chosen? Each quadrant of the xy-axes is a different color. Wait, how can we do this? We will have to ask a question about x and y.

Revisiting creating a spiral

```
def spiral(length, levels):
    if levels == 0:
        dot()
    else:
        forward(length)
        left(30)
        spiral(0.95 * length, levels - 1)
```

81

Let’s revisit the spiral function written now in a recursive manner

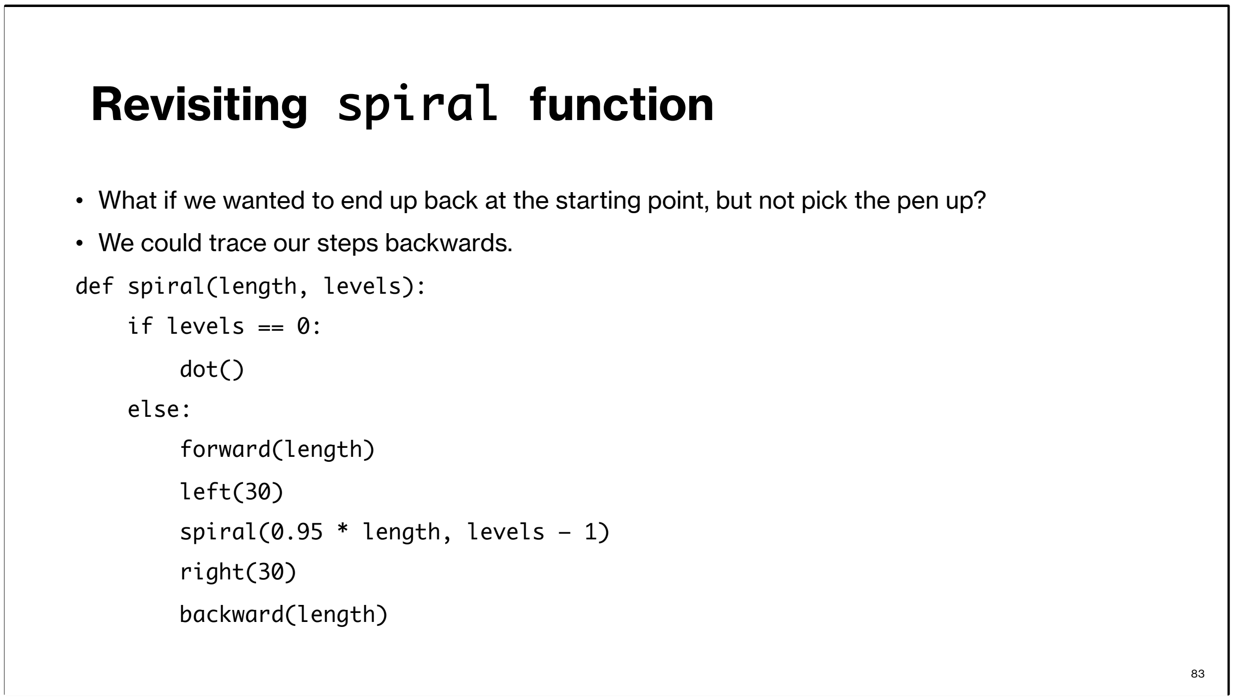


Revisiting `spiral` function

- Let’s assume I called spiral(80, 50)
- When does it stop?
  - When levels = 0.
    - We put a dot there to make it explicit.
- Repeat 50 times:
  - forward length
  - left 30
  - reduce length by 5%

```
def spiral(length, levels):
    if levels == 0:
        dot()
    else:
        forward(length)
        left(30)
        spiral(0.95 * length, levels - 1)
```

82



Revisiting `spiral` function

- What if we wanted to end up back at the starting point, but not pick the pen up?
- We could trace our steps backwards.

```
def spiral(length, levels):
    if levels == 0:
        dot()
    else:
        forward(length)
        left(30)
        spiral(0.95 * length, levels - 1)
        right(30)
        backward(length)
```

83

What if we wanted to end up back at the starting point, but not pick the pen up?
We could trace our steps backwards.

Assume that the recursive call returns back to its starting point. What would we need to do to make sure that our call returned back to the starting point?

Add the following after the recursive call:

```
right(30)
backward(length)
```

if we run it now, we draw the spiral all the way down, and then we retrace backwards.:

each call to `spiral` retraces its own part after the recursive call.
the call stack keeps track of each of the recursive calls.

`broccoli_demo` function

1. Define what the header function is:
   - `broccoli(x, y, length, angle)`
2. Define the recursive case:
   - broccoli is a line with three other broccolis at the end:
     - one directly straight out
     - one 20 degrees to the left
     - one 20 degrees to the right
   - the three other broccolis should be smaller/shorter than the current

84

`broccoli_demo` function

3. Define the base case:
   - in each case, the length of the broccoli to be drawn gets shorter.
   - We stop at `length < 10` and place a yellow dot
4. put it all together!
   - Check the base case first:
     - if `length < 10`
       - Draw a yellow dot.
   - Otherwise:
     - draw three smaller broccolis at different angles.

- `new_x` and `new_y` are the ending coordinates of the line being drawn. We save them because after the first recursive call to broccoli the turtle won't be in the same place.

85

Code

rec_sum.a51

turtle-examples.py

turtle-recursion.py