Data Structure Layout

In HERA/Assembly

Today, we're going to build some data structures in HERA

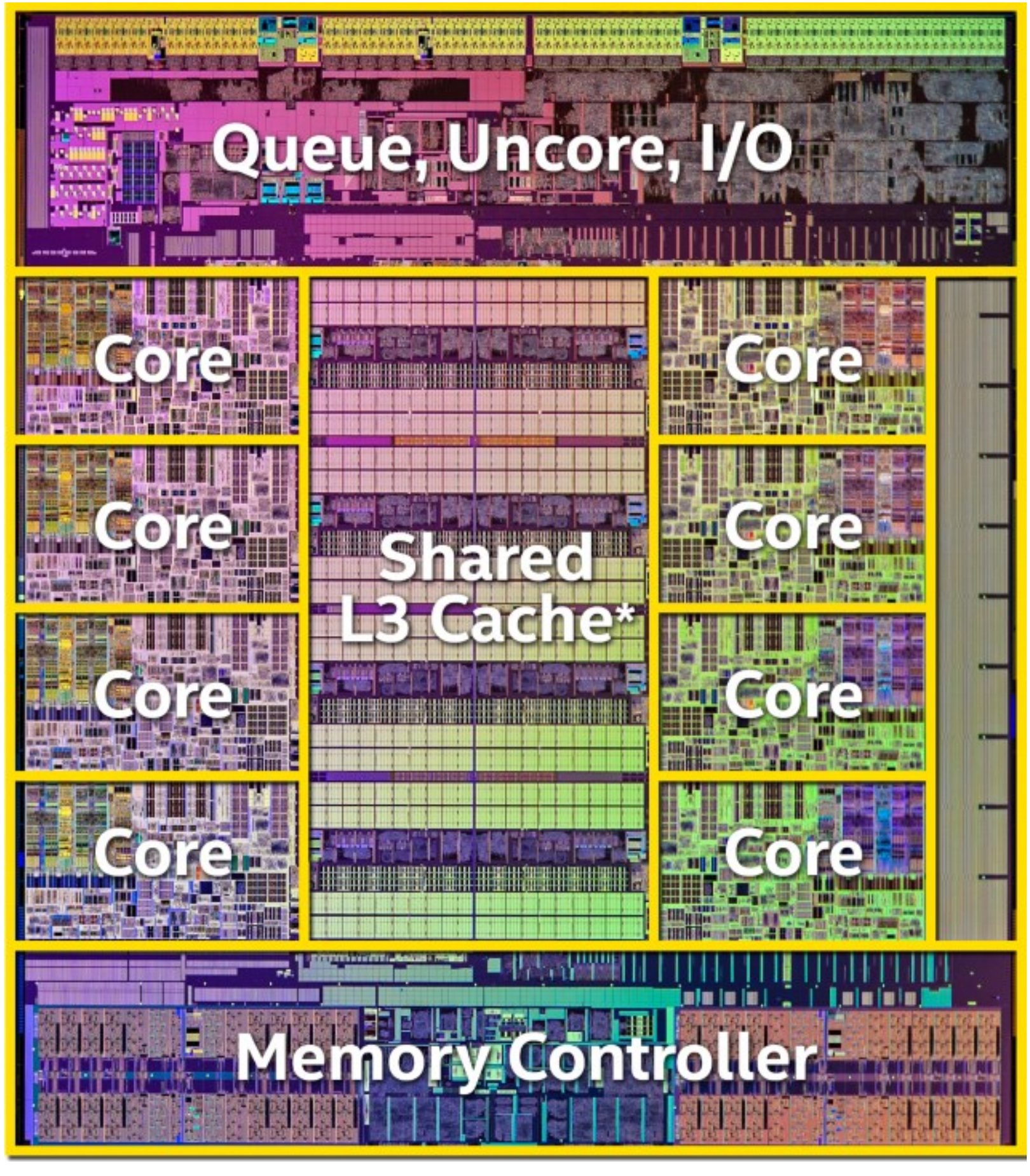
First, a note on memory

Registers are very fast

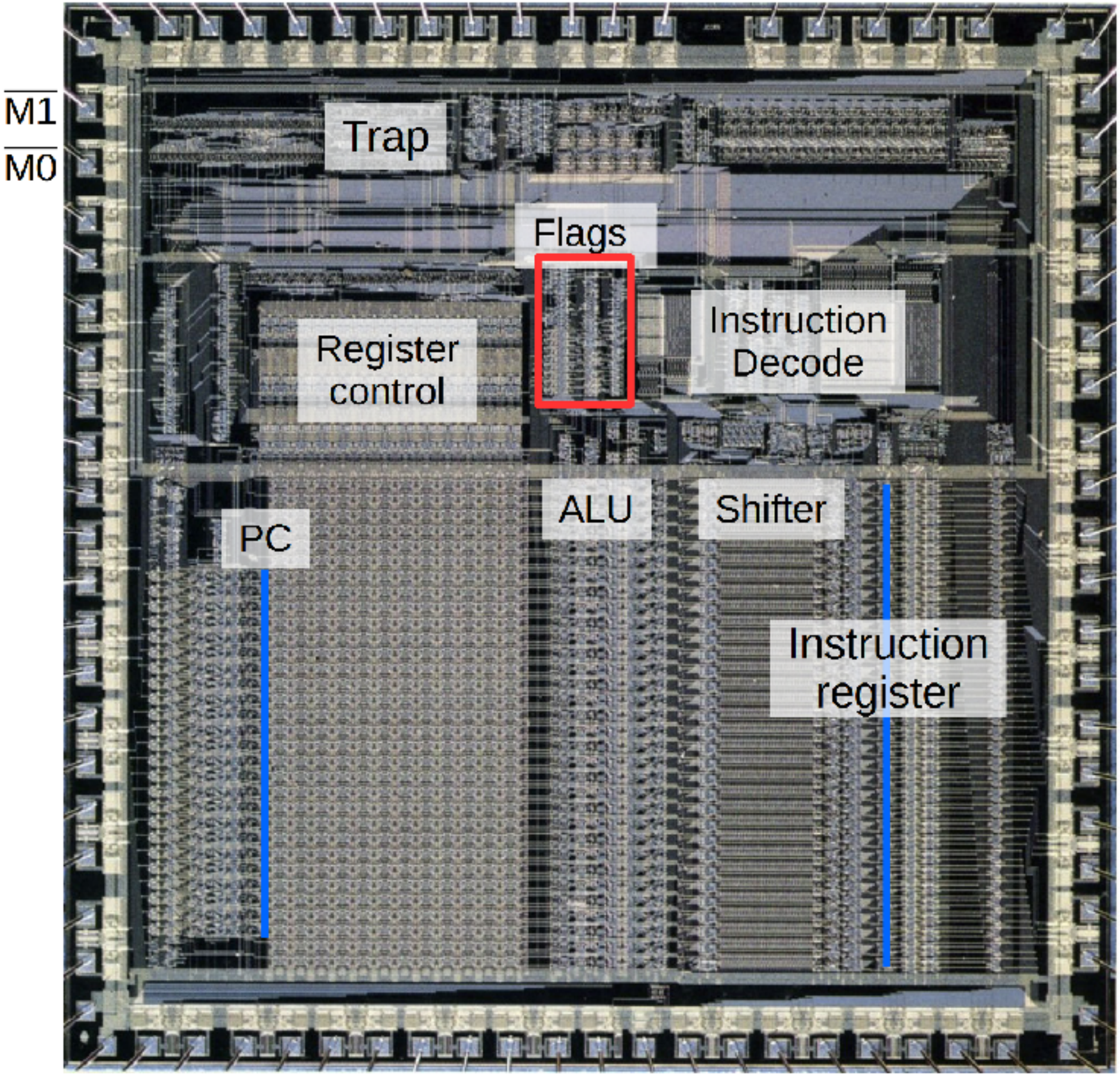
RAM is relatively slow

We use a cache to sit between them

8-core
Haswell chip




32K L1
I-cache
32K L1
D-cache
256K L2
Cache
8 MB Last Level Cache


M1
M0
Trap
Flags
Register control
Instruction Decode
ALU
Shifter
PC
Instruction register

Why can’t you just do everything in registers?

Here's the moral of today's lecture…

You could, but it would be pretty slow.

Smart programming is about knowing when to use one vs. the other

One nice thing about C/C++ is that the compiler is **very good** at making these kinds of decisions for you

One mistake I made from yesterday

HERA organizes memory into **words**,
two-byte sequences

So I said you would STORE(R_d,0,R_b) and R_b = 2

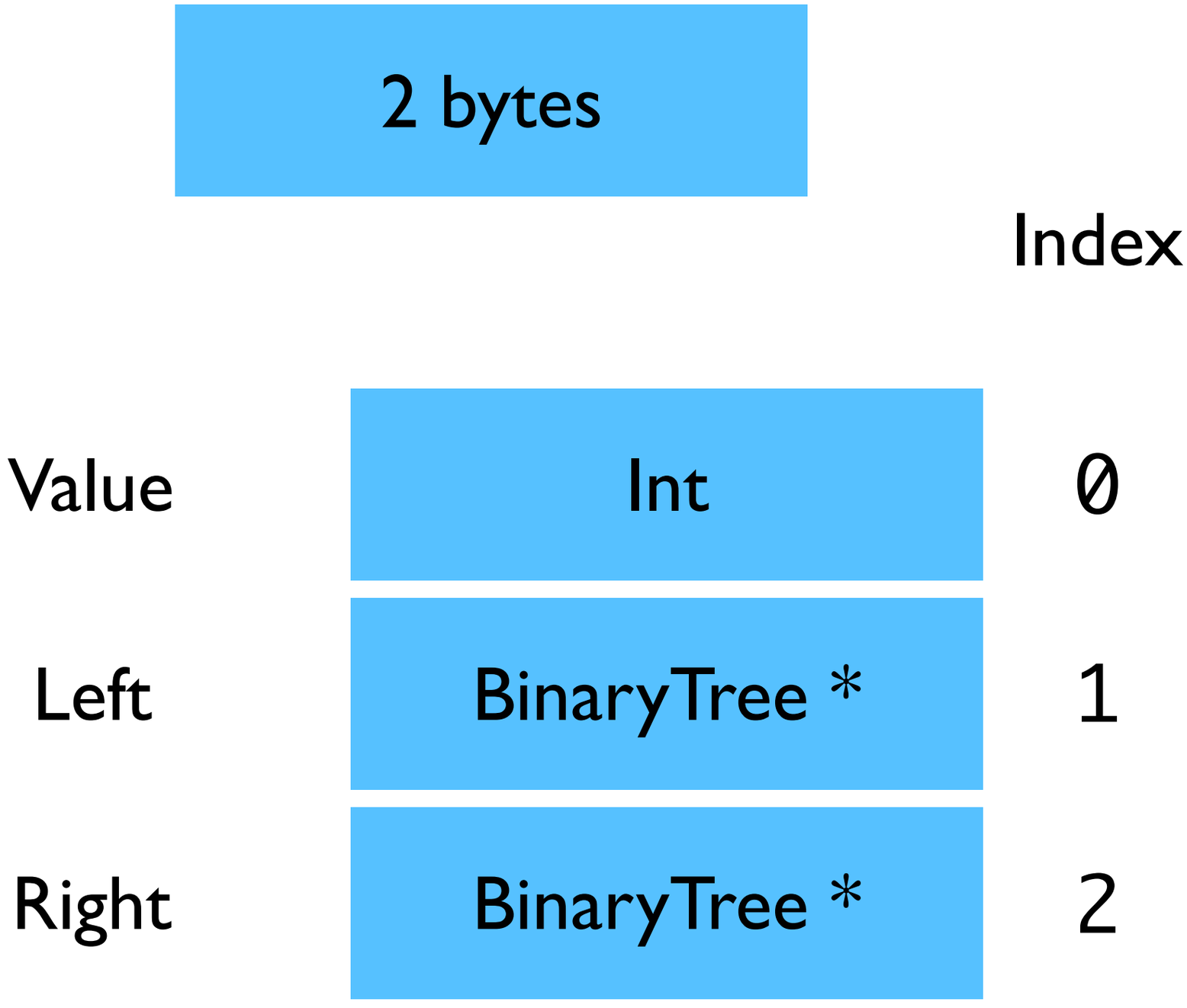
But I was wrong, it turns out you would use 1

HERA automatically multiplies all of your indices by 2

Let’s start with a binary tree

In C++

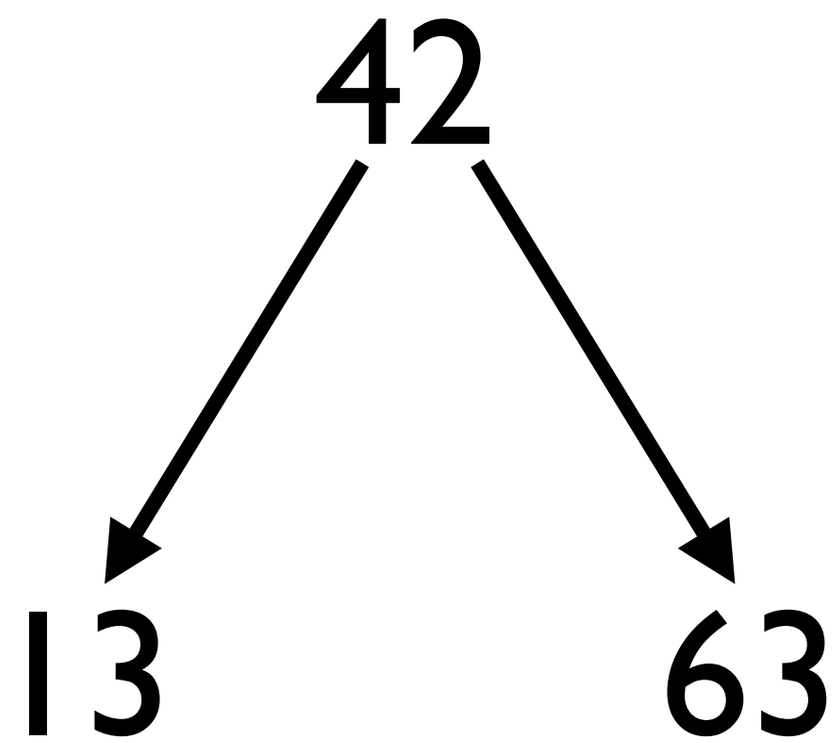
```
struct BinaryTree {
  int value;
  BinaryTree *left;
  BinaryTree *right;
}
```


2 bytes
Index
Value
Int
0
Left
BinaryTree *
1
Right
BinaryTree *
2

Use NULL (0) to indicate empty pointer

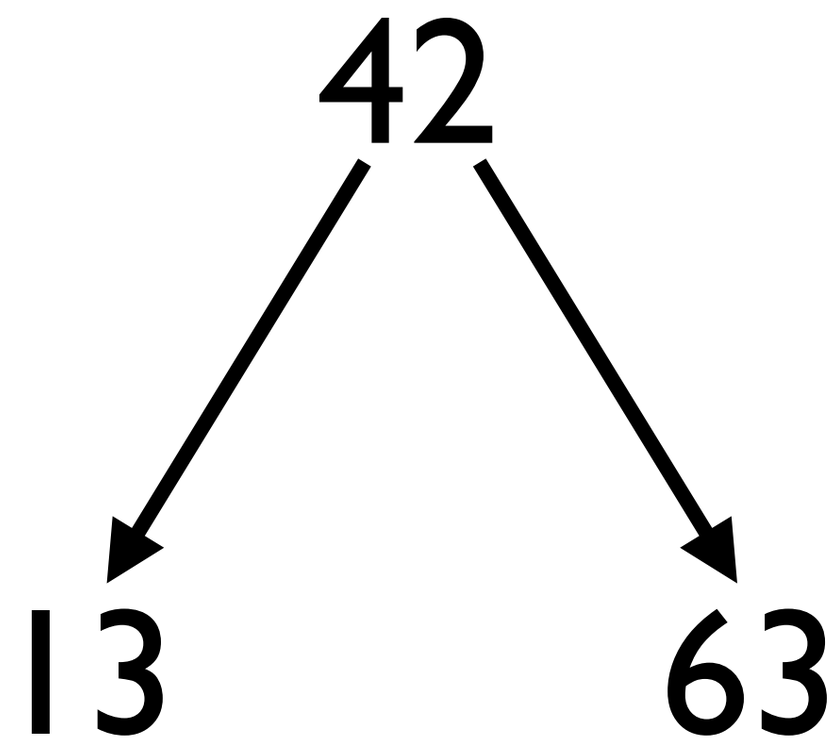
Your job, use INTEGER to create a binary tree containing 42. Label the tree `root`

Now consider this binary tree


42
13
63

Build this binary tree

Use INTEGER and labels



Functions

Implemented a variety of ways. The simplest way is to pass/return args/result in registers

For now, don’t worry about returning from the function

```
int getValue(BinaryTree *t) { return t->value; }
```

```
// Assume R1 contains the tree
// Result placed in R2
LABEL(getValue)
  LOAD(R2, 0, R1)
```

Note: the programmer using this has to know the input / outputs

Figure out how to “call” that function

Now, how do we **return** from our function?

Answer: pass a pointer to the **next** instruction when you call it

(Use a HERA label)

"I want you to get the value, and then **next**
I want you to go to label
`afterGetValue`"

```
// Assume R1 contains the tree
// Assume R2 contains next instr
// Result placed in R3
LABEL(getValue)
  LOAD(R3, 0, R1)
  BR(R2)
```

Your assignment

Call this function, pass in a label to "next" instruction, then add one to the result

Alternatively: instead of passing in label to return to, pass in offset (1+current instruction)

(We'll see how to use this later..)

Alternatively: instead of passing in label to return to, pass in offset (1+current instruction)

HERA convention: use **temporary** register, R_t/ R_{13}

setLeft sets the left child to some value

```
setLeft(t,node)
```

Assignment: write setLeft

Functions I call can corrupt **my** registers

Consider the following code…

```
// Assume R1 contains the tree
// Assume R2 contains next instr
// Result placed in R3
LABEL(getValue)
  LOAD(R3, 0, R1)
  SET(R4, 0)
  BR(R2)
```

This overwrites R4

So if caller using R4, this fails

A non-contrived example…

```
int sumTree(BinaryTree *t) {
  if (t == NULL) {
    return 0;
  } else {
    return sumTree(t->left)
         + sumTree(t->right)
         + sumTree(t->value);
  }
}
```

Attempt 1

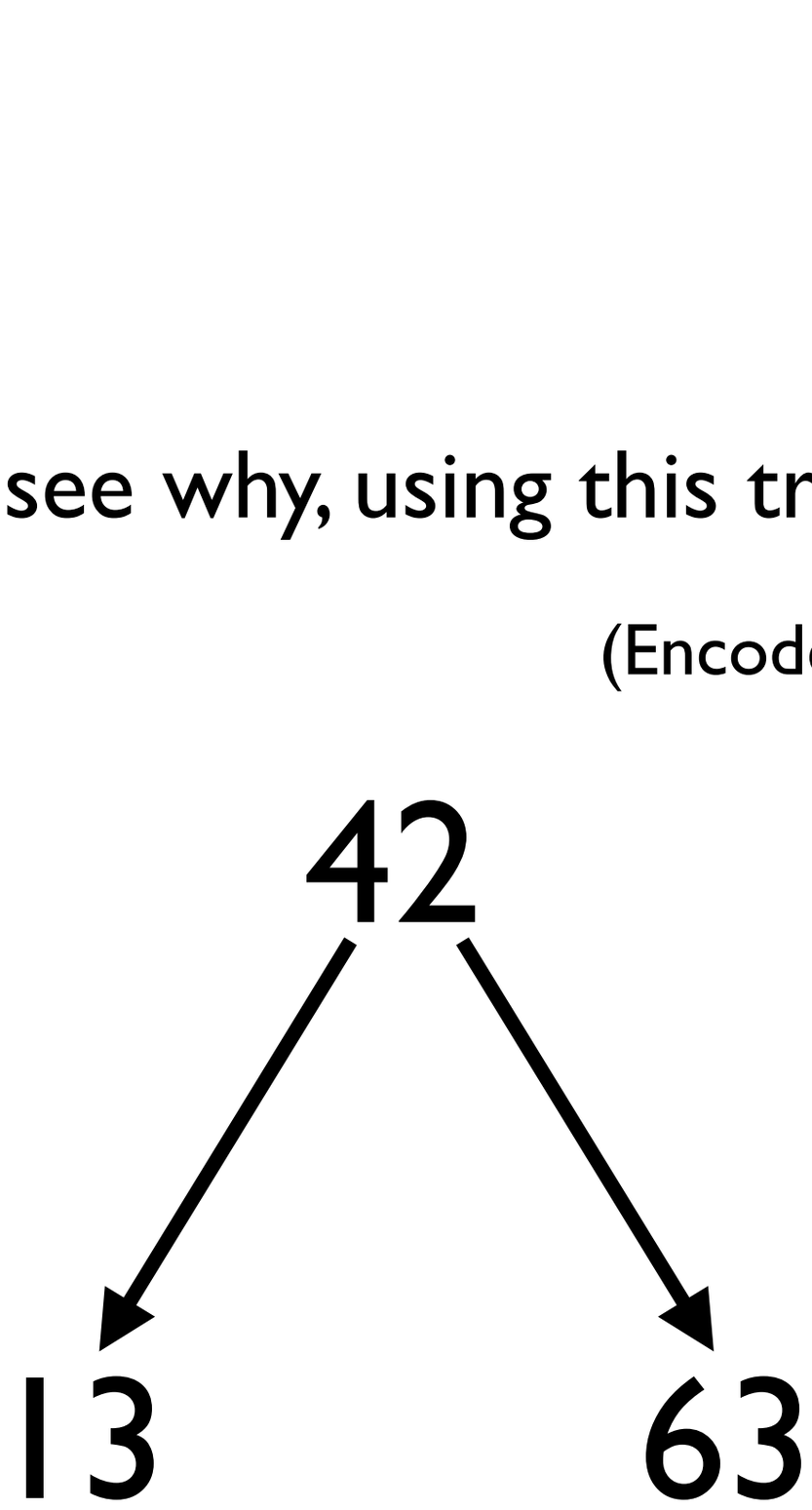
```
int sumTree(BinaryTree *t) {
  if (t == NULL) {
    return 0;
  } else {
    return sumTree(t->left)
         + sumTree(t->right)
         + sumTree(t->value);
  }
}
```

```
// Assume t is in R1
// Assume return addr in R2
// Return result in R3
LABEL(sumTree)
  SUB(R0,R0,R1)
  BNZ(not_null)
    SET(R3, 0)
    BR(R2)
  LABEL(not_null)
    LOAD(R4, 0, R1)
    LOAD(R1, 1, R1)
    SET(R2, after1)
    BR(sumTree)
LABEL(after1)
    ADD(R4,R3,R4)
    LOAD(R1, 2, R1)
    SET(R2, after2)
    BR(sumTree)
LABEL(after2)
    ADD(R4,R3,R4)
    MOVE(R3,R4)
    BR(R2)
```

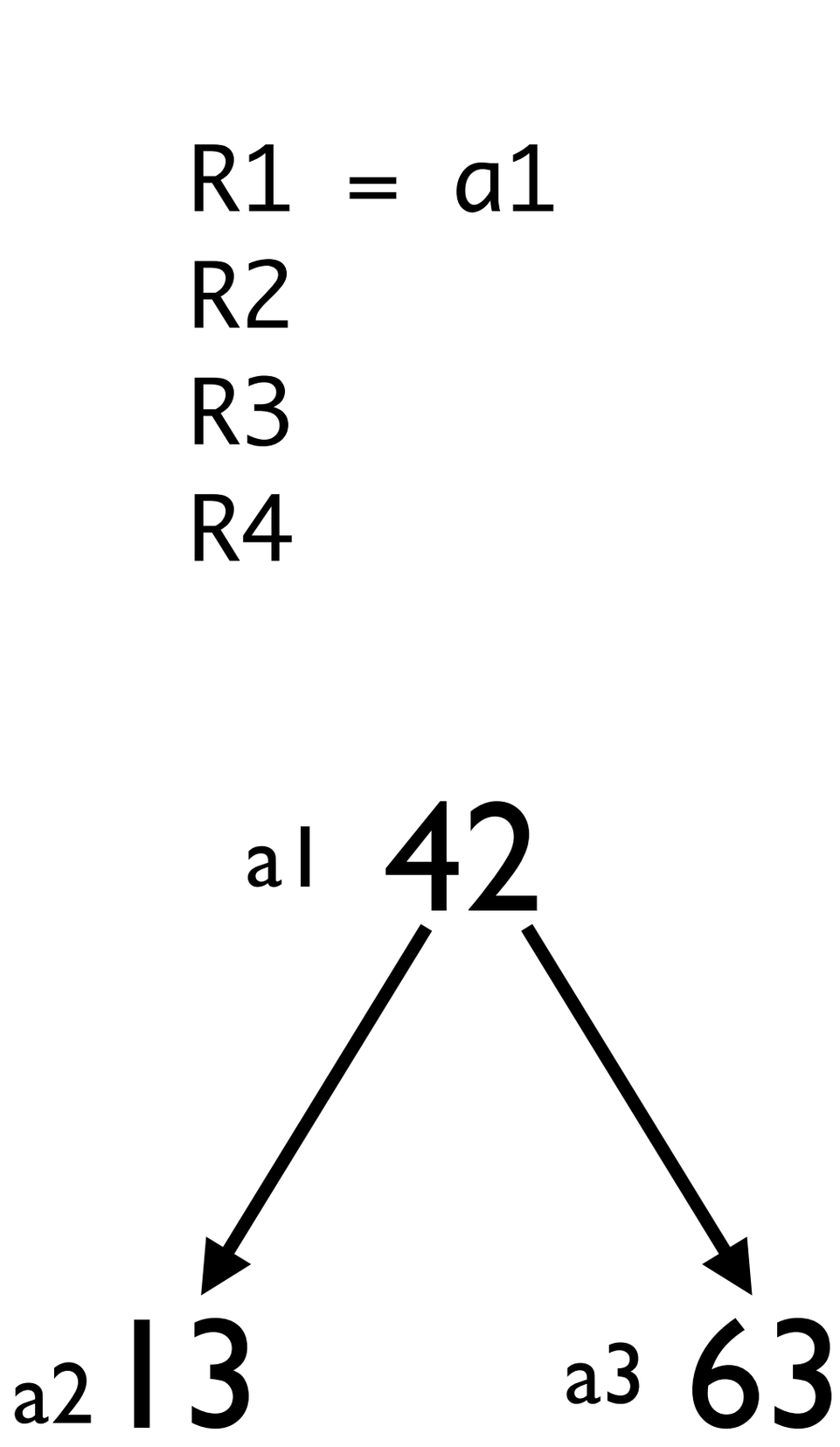
Does **not** work!

Let's see why, using this tree…

(Encoded in HERA…)

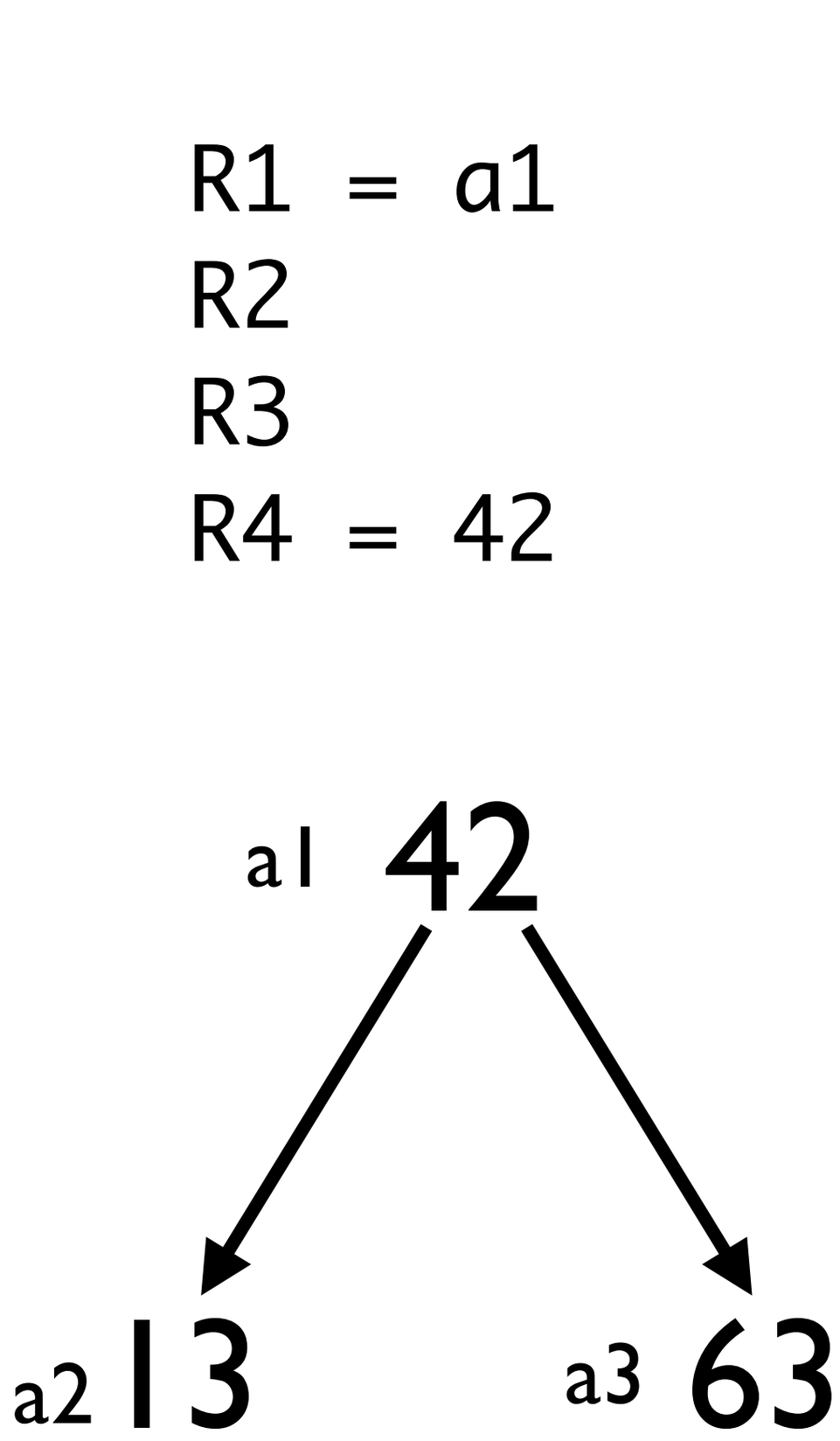


R1 = a1
R2
R3
R4



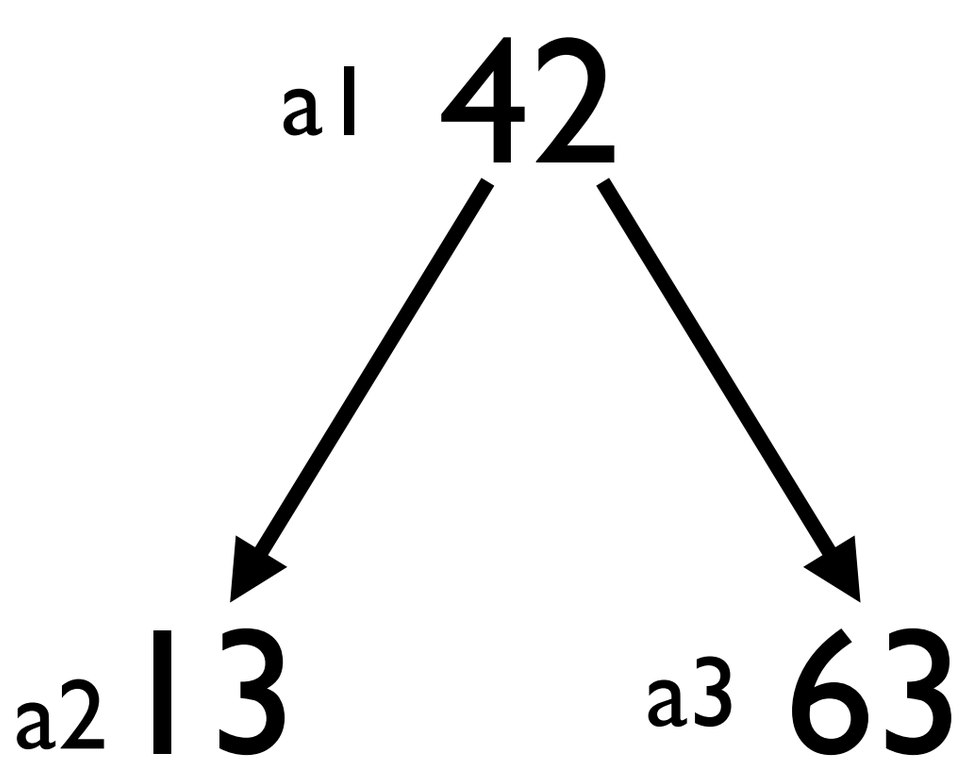

```
// Assume t is in R1
// Assume return addr in R2
// Return result in R3
LABEL(sumTree)
  SUB(R0,R0,R1)
  BNZ(not_null)
    SET(R3, 0)
    BR(R2)
  LABEL(not_null)
    LOAD(R4, 0, R1)
    LOAD(R1, 1, R1)
    SET(R2, after1)
    BR(sumTree)
LABEL(after1)
  ADD(R4,R3,R4)
  LOAD(R1, 2, R1)
  SET(R2, after2)
  BR(sumTree)
LABEL(after2)
  ADD(R4,R3,R4)
  MOVE(R3,R4)
  BR(R2)
```

R1 = a1
R2
R3
R4 = 42



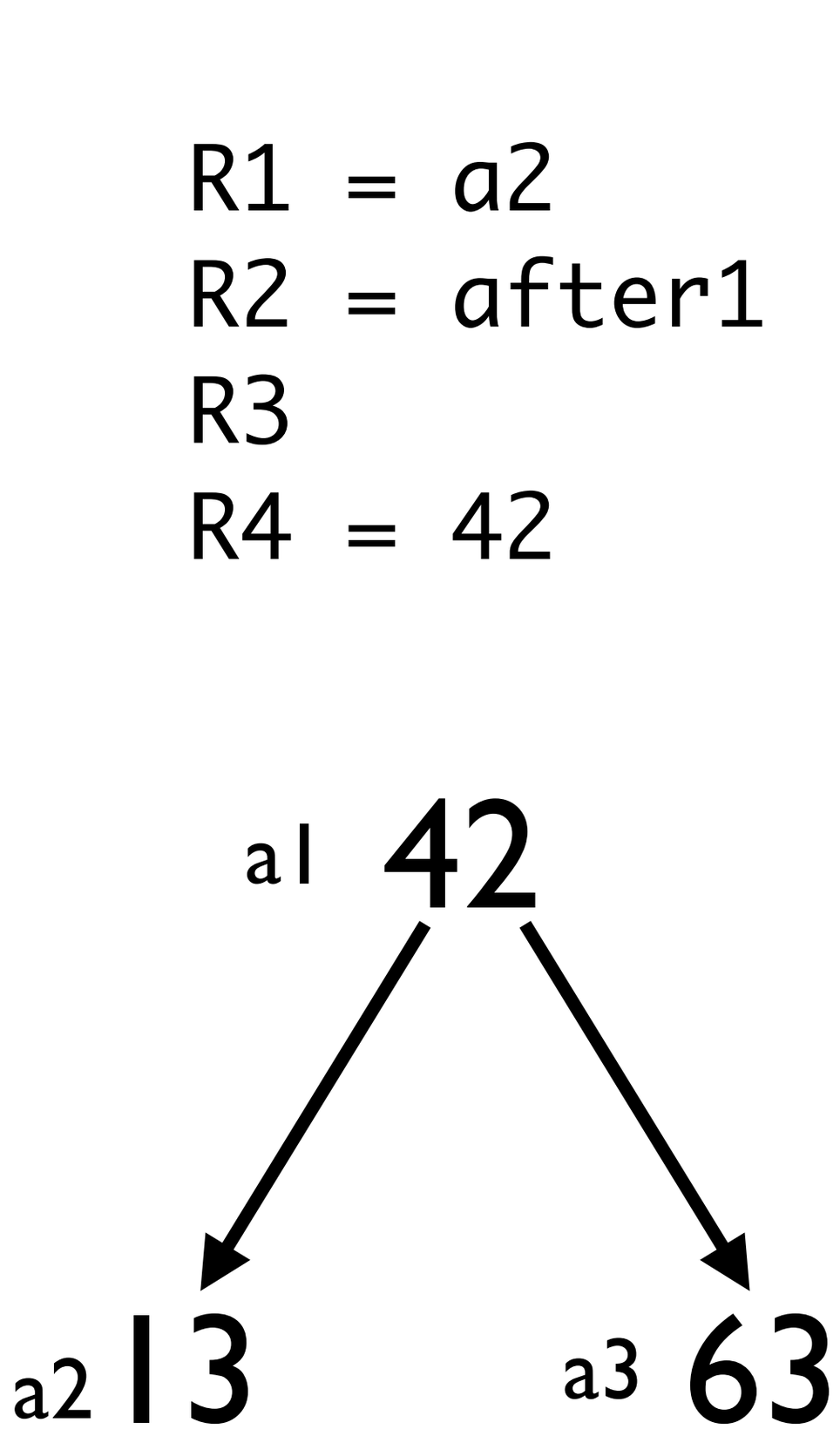

```
// Assume t is in R1
// Assume return addr in R2
// Return result in R3
LABEL(sumTree)
  SUB(R0,R0,R1)
  BNZ(not_null)
    SET(R3, 0)
    BR(R2)
  LABEL(not_null)
    LOAD(R4, 0, R1)
    LOAD(R1, 1, R1)
    SET(R2, after1)
    BR(sumTree)
LABEL(after1)
  ADD(R4,R3,R4)
  LOAD(R1, 2, R1)
  SET(R2, after2)
  BR(sumTree)
LABEL(after2)
  ADD(R4,R3,R4)
  MOVE(R3,R4)
  BR(R2)
```

R1 = a2
R2
R3
R4 = 42



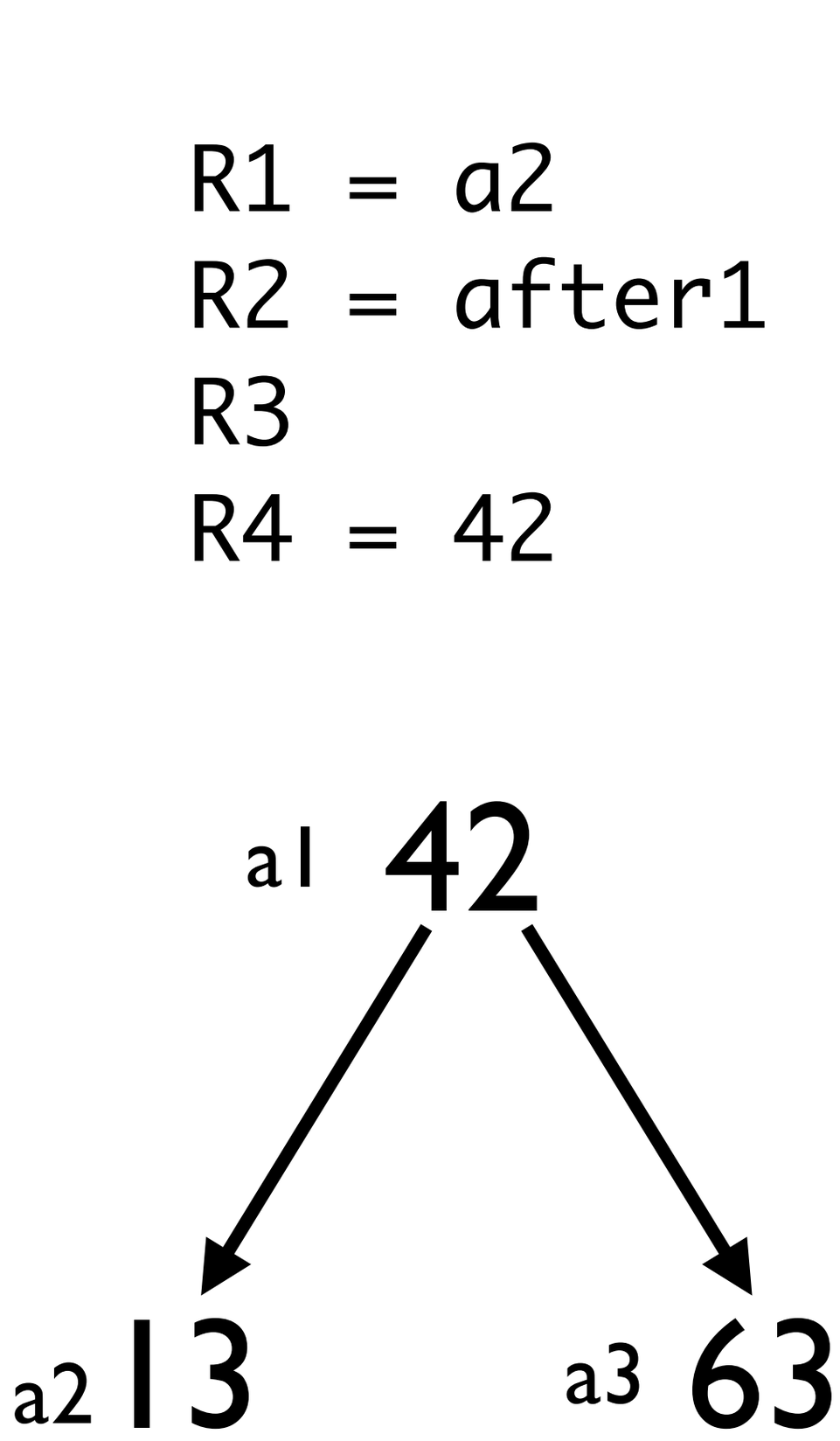

```
// Assume t is in R1
// Assume return addr in R2
// Return result in R3
LABEL(sumTree)
  SUB(R0,R0,R1)
  BNZ(not_null)
    SET(R3, 0)
    BR(R2)
  LABEL(not_null)
    LOAD(R4, 0, R1)
    LOAD(R1, 1, R1)
    SET(R2, after1)
    BR(sumTree)
LABEL(after1)
    ADD(R4,R3,R4)
    LOAD(R1, 2, R1)
    SET(R2, after2)
    BR(sumTree)
LABEL(after2)
    ADD(R4,R3,R4)
    MOVE(R3,R4)
    BR(R2)
```

R1 = a2
R2 = after1
R3
R4 = 42



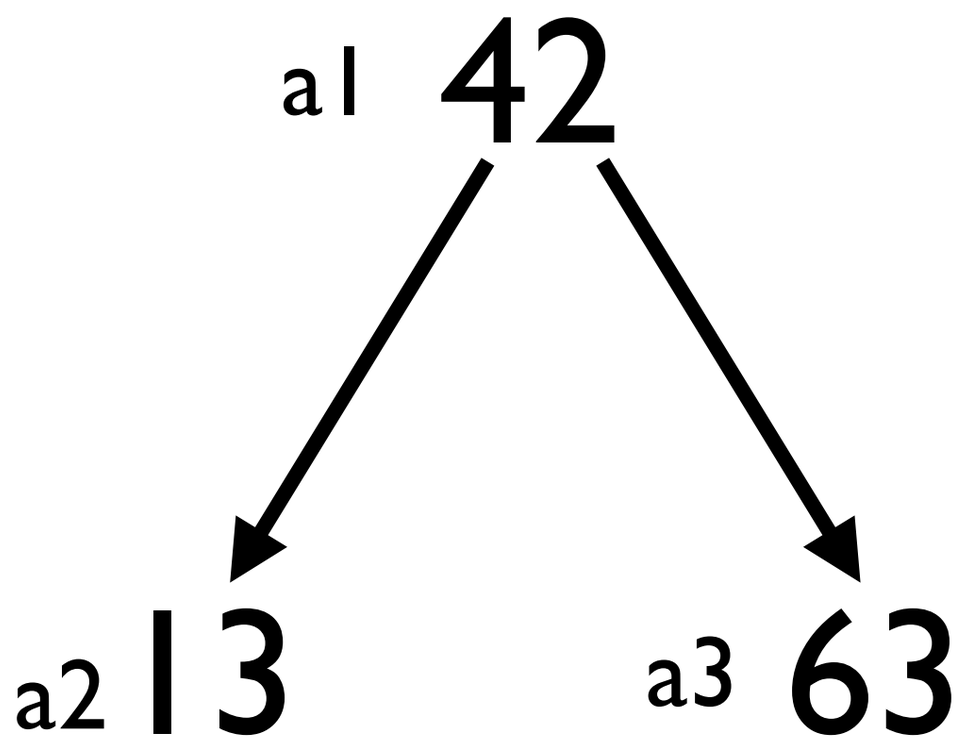

```
// Assume t is in R1
// Assume return addr in R2
// Return result in R3
LABEL(sumTree)
  SUB(R0,R0,R1)
  BNZ(not_null)
    SET(R3, 0)
    BR(R2)
  LABEL(not_null)
    LOAD(R4, 0, R1)
    LOAD(R1, 1, R1)
    SET(R2, after1)
    BR(sumTree)
LABEL(after1)
  ADD(R4,R3,R4)
  LOAD(R1, 2, R1)
  SET(R2, after2)
  BR(sumTree)
LABEL(after2)
  ADD(R4,R3,R4)
  MOVE(R3,R4)
  BR(R2)
```

R1 = a2
R2 = after1
R3
R4 = 42



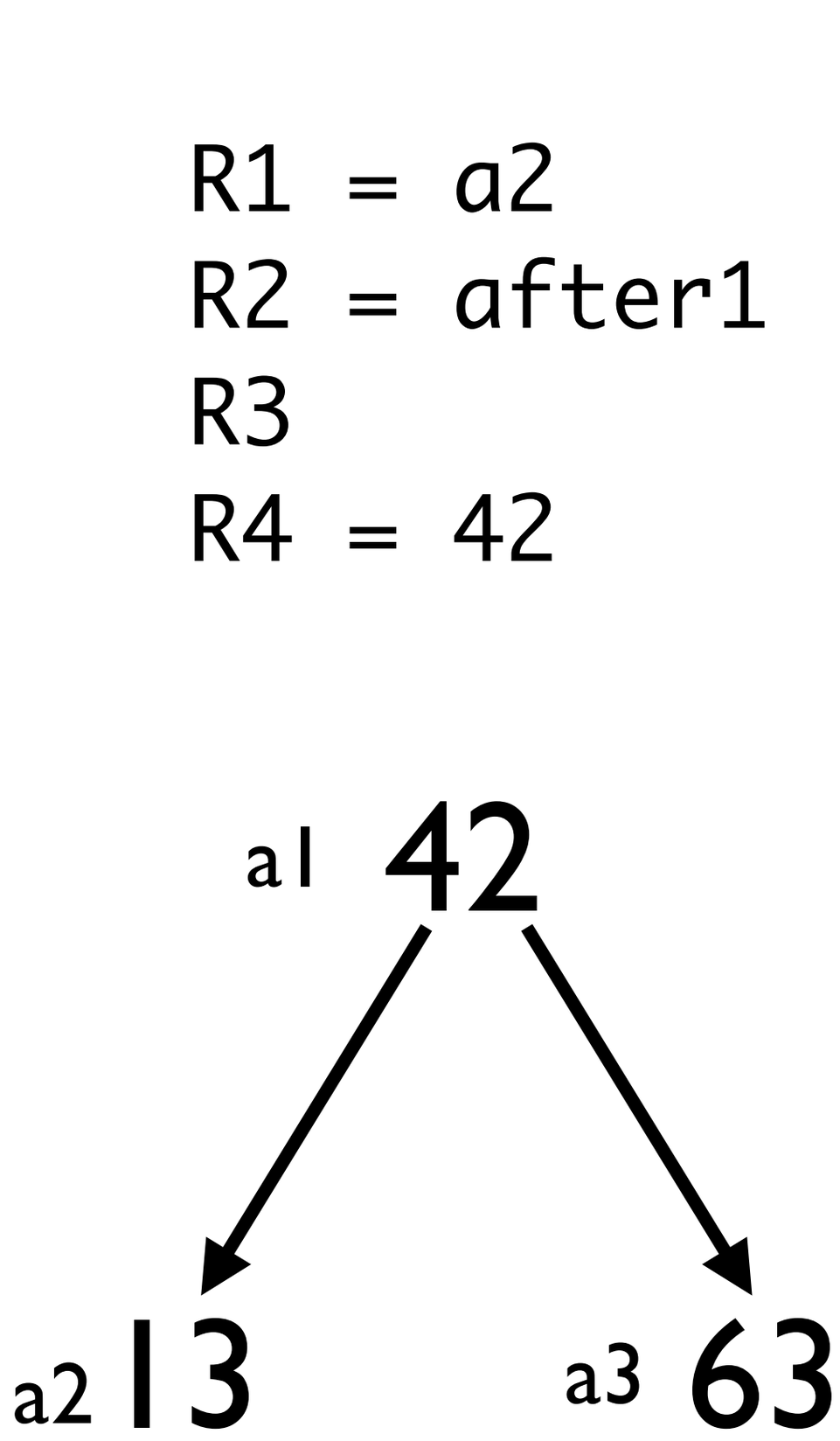

```
// Assume t is in R1
// Assume return addr in R2
// Return result in R3
LABEL(sumTree)
  SUB(R0,R0,R1)
  BNZ(not_null)
    SET(R3, 0)
    BR(R2)
  LABEL(not_null)
    LOAD(R4, 0, R1)
    LOAD(R1, 1, R1)
    SET(R2, after1)
    BR(sumTree)
LABEL(after1)
  ADD(R4,R3,R4)
  LOAD(R1, 2, R1)
  SET(R2, after2)
  BR(sumTree)
LABEL(after2)
  ADD(R4,R3,R4)
  MOVE(R3,R4)
  BR(R2)
```

R1 = a2
R2 = after1
R3
R4 = 42



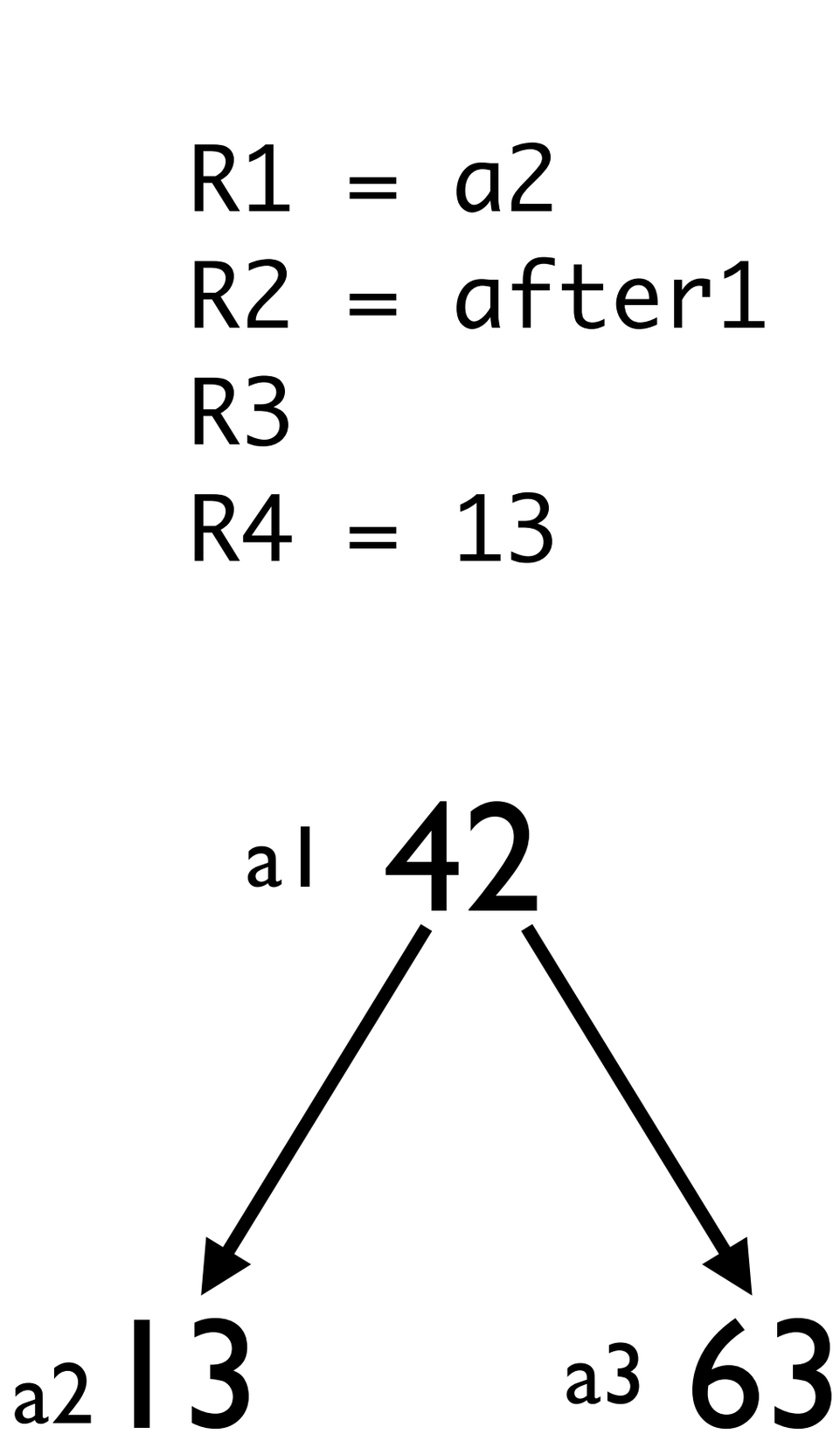

```
// Assume t is in R1
// Assume return addr in R2
// Return result in R3
LABEL(sumTree)
  SUB(R0,R0,R1)
  BNZ(not_null)
    SET(R3, 0)
    BR(R2)
  LABEL(not_null)
    LOAD(R4, 0, R1)
    LOAD(R1, 1, R1)
    SET(R2, after1)
    BR(sumTree)
LABEL(after1)
  ADD(R4,R3,R4)
  LOAD(R1, 2, R1)
  SET(R2, after2)
  BR(sumTree)
LABEL(after2)
  ADD(R4,R3,R4)
  MOVE(R3,R4)
  BR(R2)
```

R1 = a2
R2 = after1
R3
R4 = 42



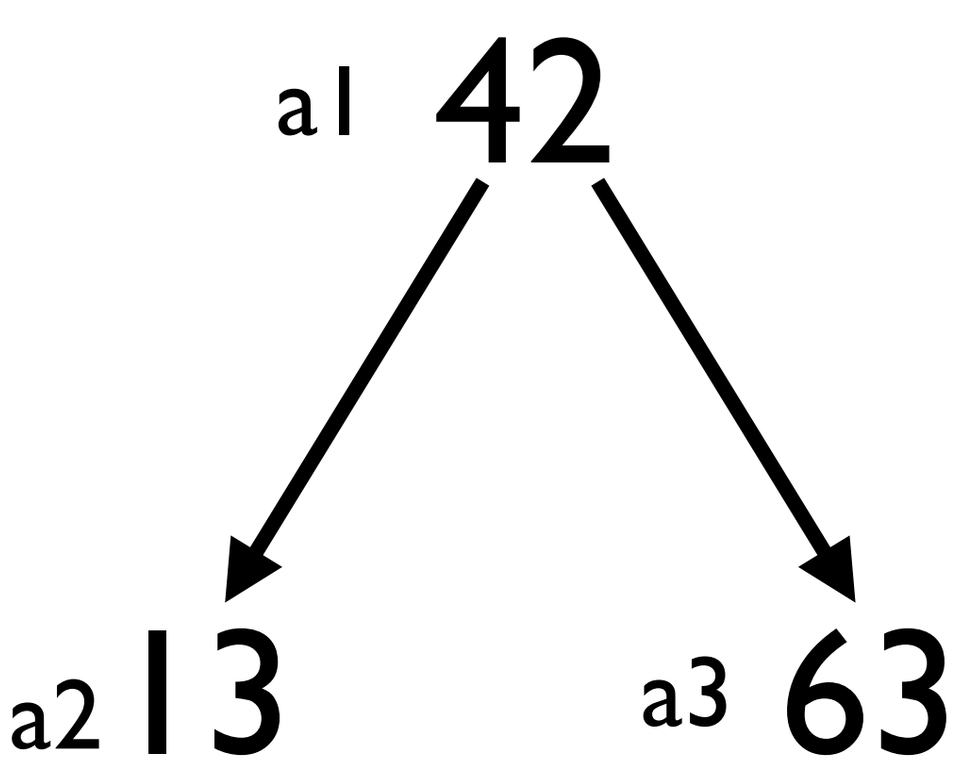

```
// Assume t is in R1
// Assume return addr in R2
// Return result in R3
LABEL(sumTree)
  SUB(R0,R0,R1)
  BNZ(not_null)
    SET(R3, 0)
    BR(R2)
  LABEL(not_null)
    LOAD(R4, 0, R1)
    LOAD(R1, 1, R1)
    SET(R2, after1)
    BR(sumTree)
LABEL(after1)
  ADD(R4,R3,R4)
  LOAD(R1, 2, R1)
  SET(R2, after2)
  BR(sumTree)
LABEL(after2)
  ADD(R4,R3,R4)
  MOVE(R3,R4)
  BR(R2)
```

R1 = a2
R2 = after1
R3
R4 = 13



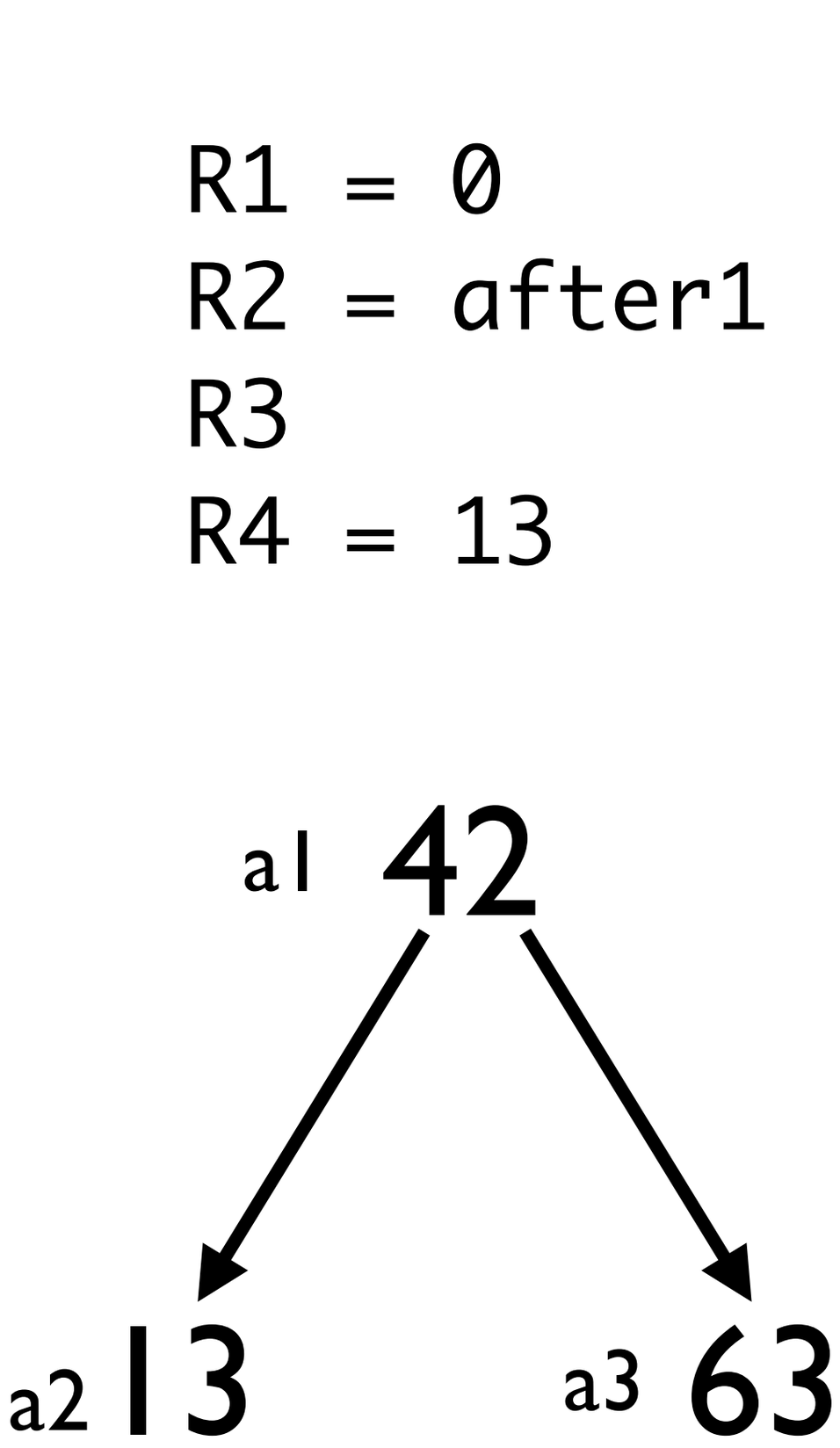

```
// Assume t is in R1
// Assume return addr in R2
// Return result in R3
LABEL(sumTree)
  SUB(R0,R0,R1)
  BNZ(not_null)
    SET(R3, 0)
    BR(R2)
  LABEL(not_null)
    LOAD(R4, 0, R1)
    LOAD(R1, 1, R1)
    SET(R2, after1)
    BR(sumTree)
LABEL(after1)
  ADD(R4,R3,R4)
  LOAD(R1, 2, R1)
  SET(R2, after2)
  BR(sumTree)
LABEL(after2)
  ADD(R4,R3,R4)
  MOVE(R3,R4)
  BR(R2)
```

R1 = 0
R2 = after1
R3
R4 = 13



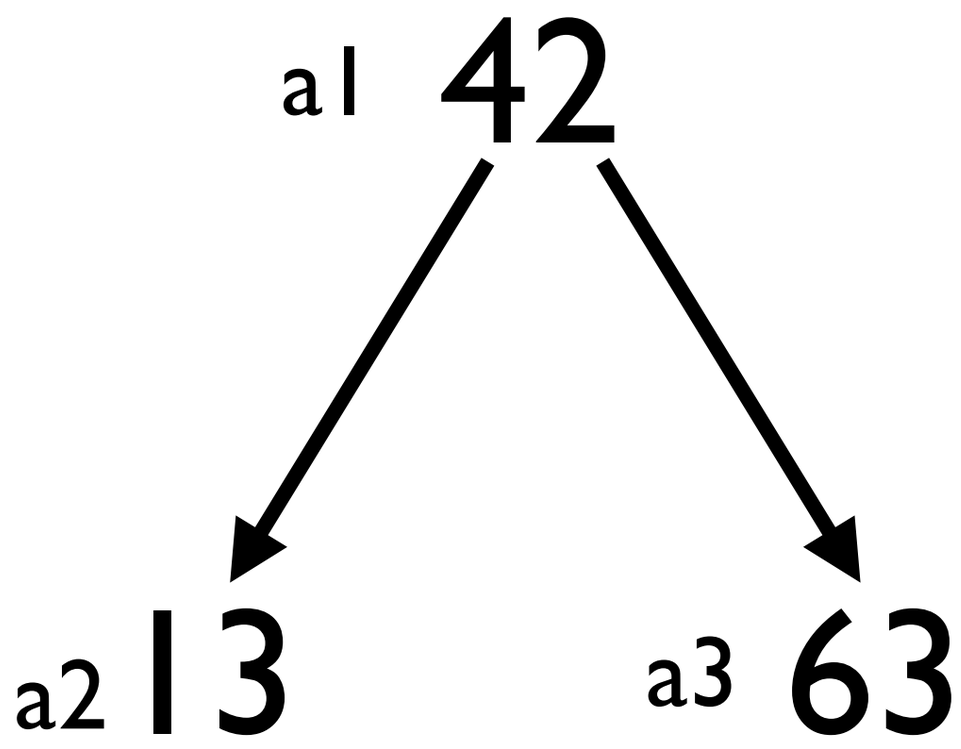

```
// Assume t is in R1
// Assume return addr in R2
// Return result in R3
LABEL(sumTree)
  SUB(R0,R0,R1)
  BNZ(not_null)
    SET(R3, 0)
    BR(R2)
  LABEL(not_null)
    LOAD(R4, 0, R1)
    LOAD(R1, 1, R1)
    SET(R2, after1)
    BR(sumTree)
LABEL(after1)
  ADD(R4,R3,R4)
  LOAD(R1, 2, R1)
  SET(R2, after2)
  BR(sumTree)
LABEL(after2)
  ADD(R4,R3,R4)
  MOVE(R3,R4)
  BR(R2)
```

R1 = 0
R2 = after1
R3
R4 = 13



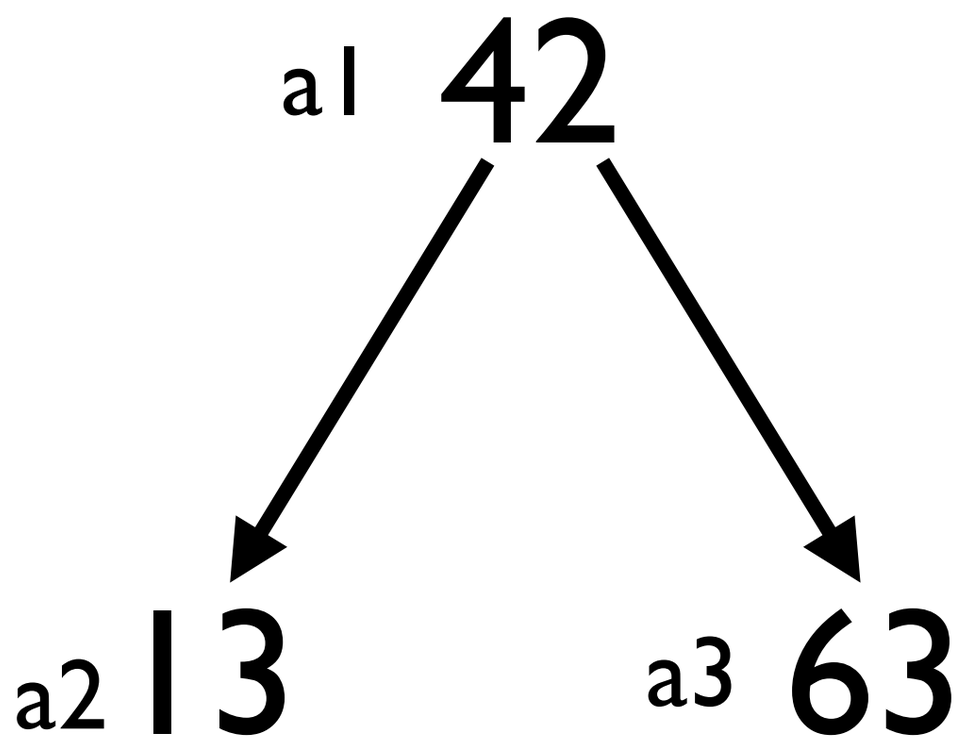

```
// Assume t is in R1
// Assume return addr in R2
// Return result in R3
LABEL(sumTree)
  SUB(R0,R0,R1)
  BNZ(not_null)
    SET(R3, 0)
    BR(R2)
  LABEL(not_null)
    LOAD(R4, 0, R1)
    LOAD(R1, 1, R1)
    SET(R2, after1)
    BR(sumTree)
LABEL(after1)
  ADD(R4,R3,R4)
  LOAD(R1, 2, R1)
  SET(R2, after2)
  BR(sumTree)
LABEL(after2)
  ADD(R4,R3,R4)
  MOVE(R3,R4)
  BR(R2)
```

R1 = 0
R2 = after1
R3
R4 = 13



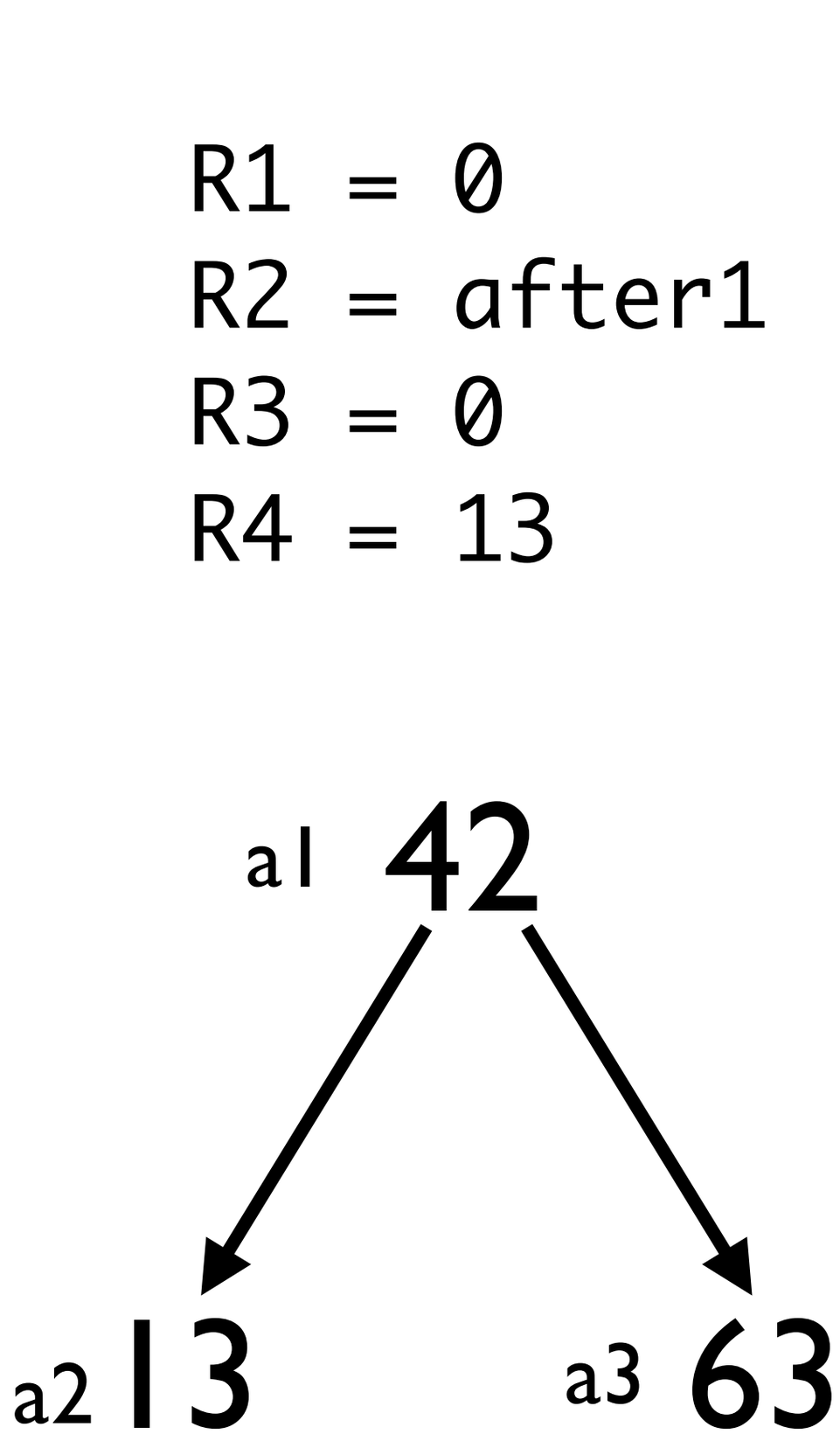

```
// Assume t is in R1
// Assume return addr in R2
// Return result in R3
LABEL(sumTree)
  SUB(R0,R0,R1)
  BNZ(not_null)
    SET(R3, 0)
    BR(R2)
  LABEL(not_null)
    LOAD(R4, 0, R1)
    LOAD(R1, 1, R1)
    SET(R2, after1)
    BR(sumTree)
LABEL(after1)
  ADD(R4,R3,R4)
  LOAD(R1, 2, R1)
  SET(R2, after2)
  BR(sumTree)
LABEL(after2)
  ADD(R4,R3,R4)
  MOVE(R3,R4)
  BR(R2)
```

```
R1 = 0
R2 = after1
R3 = 0
R4 = 13
```



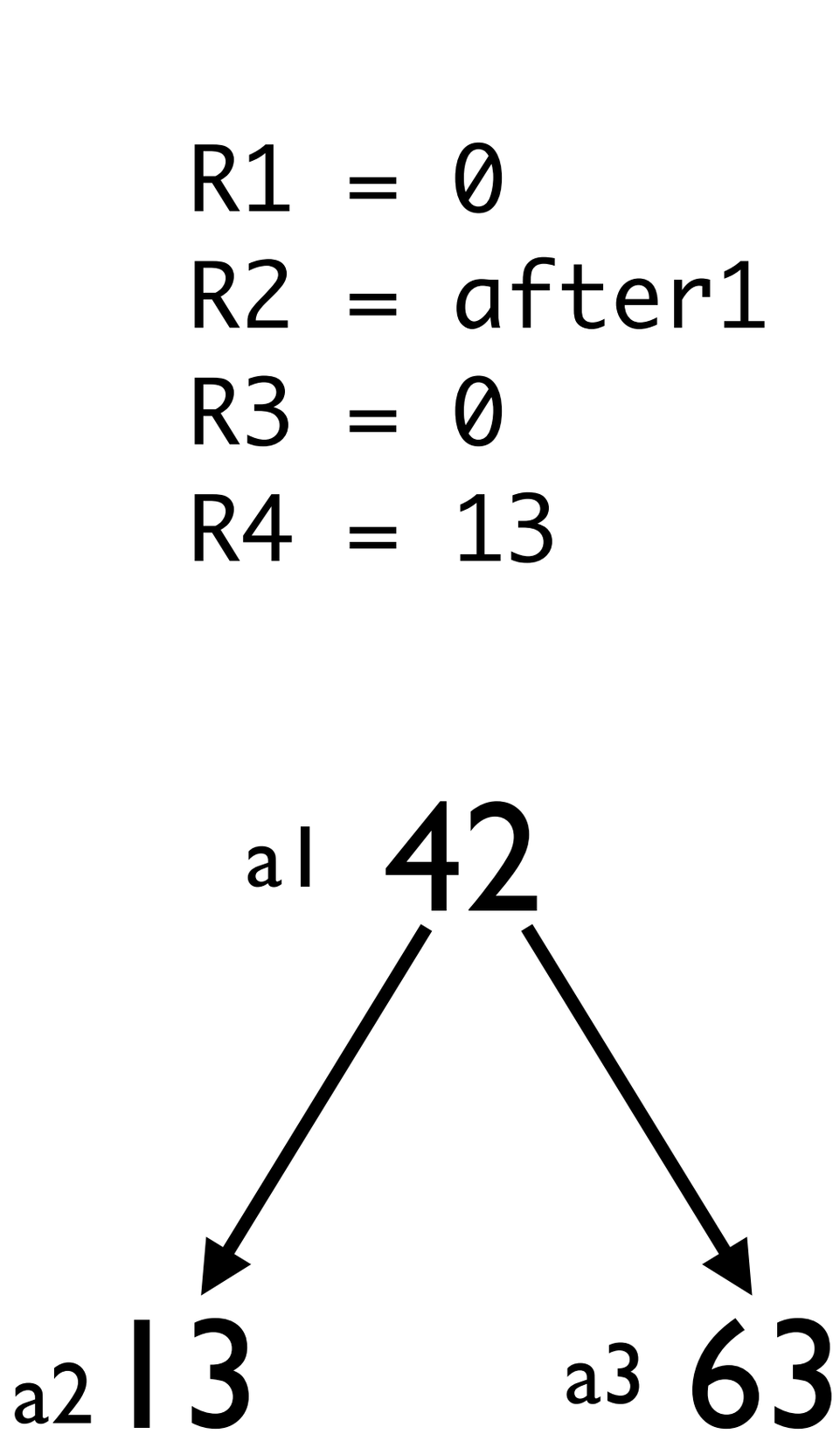

```
// Assume t is in R1
// Assume return addr in R2
// Return result in R3
LABEL(sumTree)
  SUB(R0,R0,R1)
  BNZ(not_null)
    SET(R3, 0)
    BR(R2)
  LABEL(not_null)
    LOAD(R4, 0, R1)
    LOAD(R1, 1, R1)
    SET(R2, after1)
    BR(sumTree)
LABEL(after1)
  ADD(R4,R3,R4)
  LOAD(R1, 2, R1)
  SET(R2, after2)
  BR(sumTree)
LABEL(after2)
  ADD(R4,R3,R4)
  MOVE(R3,R4)
  BR(R2)
```

R1 = 0
R2 = after1
R3 = 0
R4 = 13



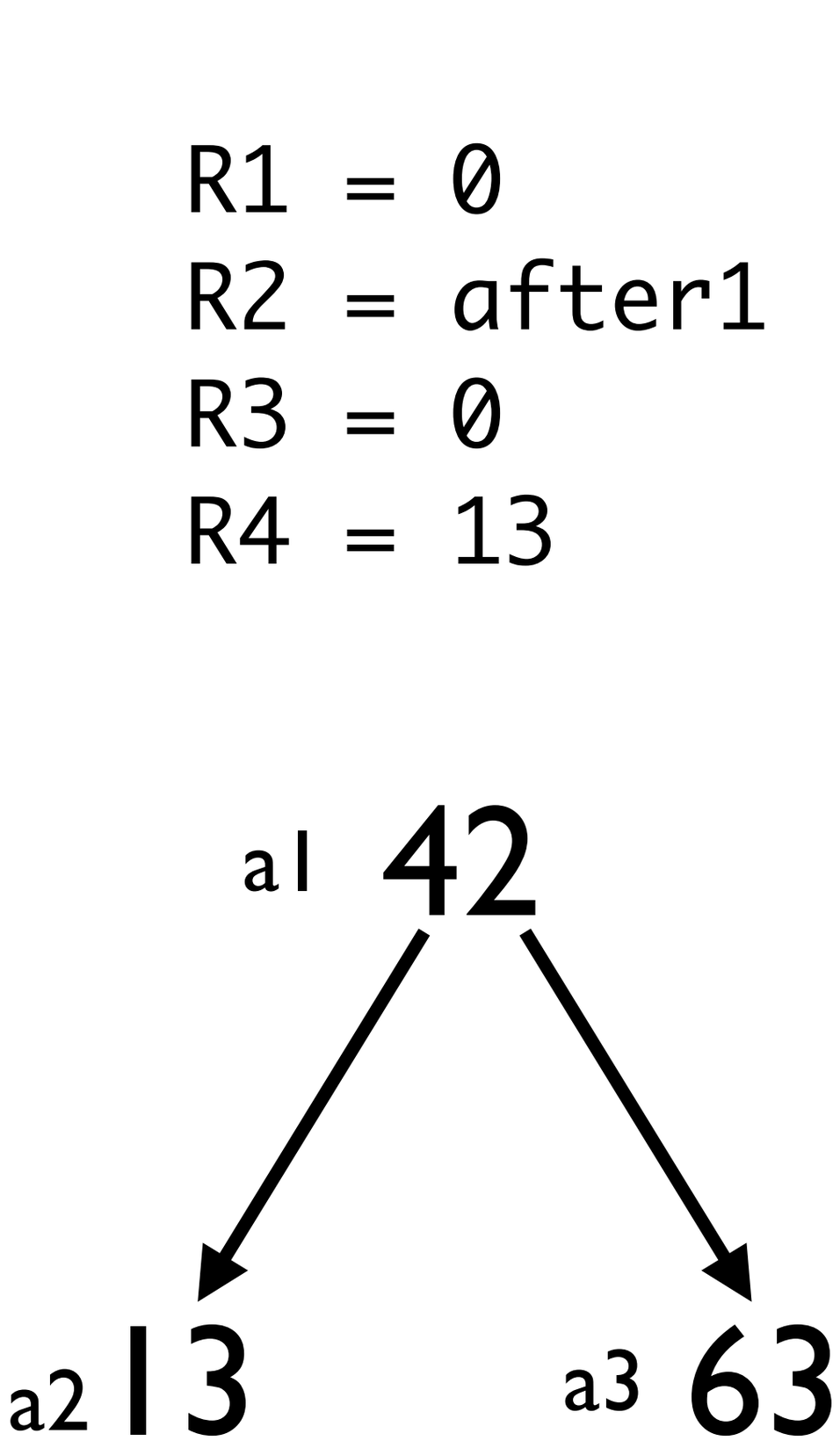

```
// Assume t is in R1
// Assume return addr in R2
// Return result in R3
LABEL(sumTree)
  SUB(R0,R0,R1)
  BNZ(not_null)
    SET(R3, 0)
    BR(R2)
  LABEL(not_null)
    LOAD(R4, 0, R1)
    LOAD(R1, 1, R1)
    SET(R2, after1)
    BR(sumTree)
LABEL(after1)
  ADD(R4,R3,R4)
  LOAD(R1, 2, R1)
  SET(R2, after2)
  BR(sumTree)
LABEL(after2)
  ADD(R4,R3,R4)
  MOVE(R3,R4)
  BR(R2)
```

R1 = 0
R2 = after1
R3 = 0
R4 = 13



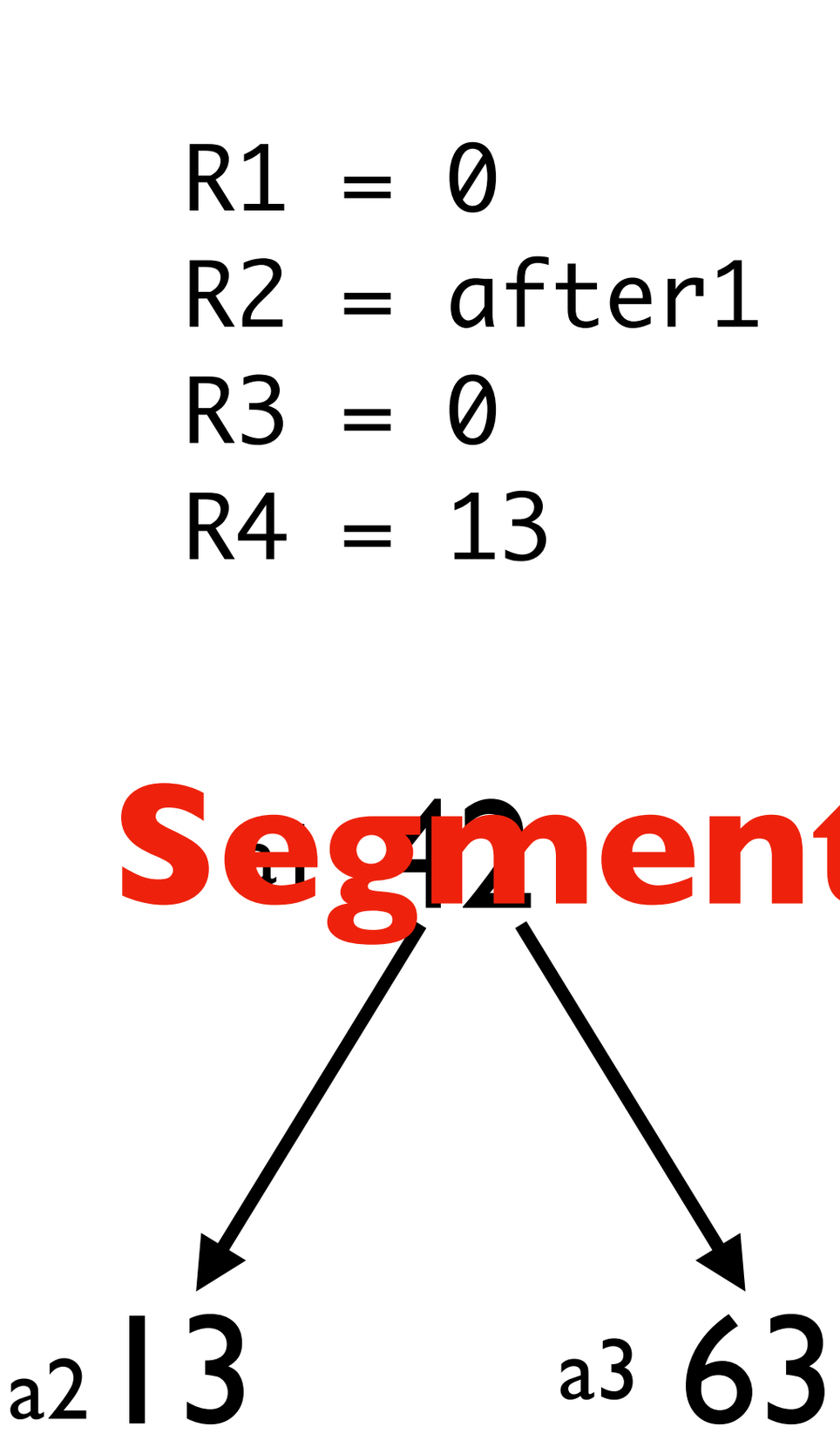

```
// Assume t is in R1
// Assume return addr in R2
// Return result in R3
LABEL(sumTree)
  SUB(R0,R0,R1)
  BNZ(not_null)
    SET(R3, 0)
    BR(R2)
  LABEL(not_null)
    LOAD(R4, 0, R1)
    LOAD(R1, 1, R1)
    SET(R2, after1)
    BR(sumTree)
LABEL(after1)
  ADD(R4,R3,R4)
  LOAD(R1, 2, R1)
  SET(R2, after2)
  BR(sumTree)
LABEL(after2)
  ADD(R4,R3,R4)
  MOVE(R3,R4)
  BR(R2)
```

R1 = 0
R2 = after1
R3 = 0
R4 = 13




```
// Assume t is in R1
// Assume return addr in R2
// Return result in R3
LABEL(sumTree)
  SUB(R0,R0,R1)
  BNZ(not_null)
    SET(R3, 0)
    BR(R2)
  LABEL(not_null)
    LOAD(R4, 0, R1)
    LOAD(R1, 1, R1)
    SET(R2, after1)
    BR(sumTree)
LABEL(after1)
  ADD(R4,R3,R4)
  LOAD(R1, 2, R1)
  SET(R2, after2)
  BR(sumTree)
LABEL(after2)
  ADD(R4,R3,R4)
  MOVE(R3,R4)
  BR(R2)
```

R1 = 0
R2 = after1
R3 = 0
R4 = 13



Segmentation fault

```
// Assume t is in R1
// Assume return addr in R2
// Return result in R3
LABEL(sumTree)
  SUB(R0,R0,R1)
  BNZ(not_null)
    SET(R3, 0)
    BR(R2)
  LABEL(not_null)
    LOAD(R4, 0, R1)
    LOAD(R1, 1, R1)
    SET(R2, after1)
    BR(sumTree)
LABEL(after1)
    ADD(R4,R3,R4)
    LOAD(R1, 2, R1)
    SET(R2, after2)
    BR(sumTree)
LABEL(after2)
    ADD(R4,R3,R4)
    MOVE(R3,R4)
    BR(R2)
```

So… What went wrong?

I assumed my locals were my own

Which registers does `sumTree` overwrite?

Which registers does `sumTree` overwrite?

R1

R2

R3

R4

So, how do we fix this?

Use the **stack**

I need to **save** my registers when I call functions

Stack

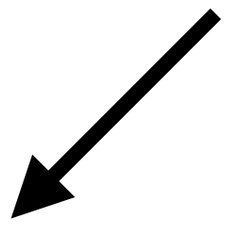
Note: in HERA, unlike some other processors, the stack grows **up**

(This is different than my i7)

…

…

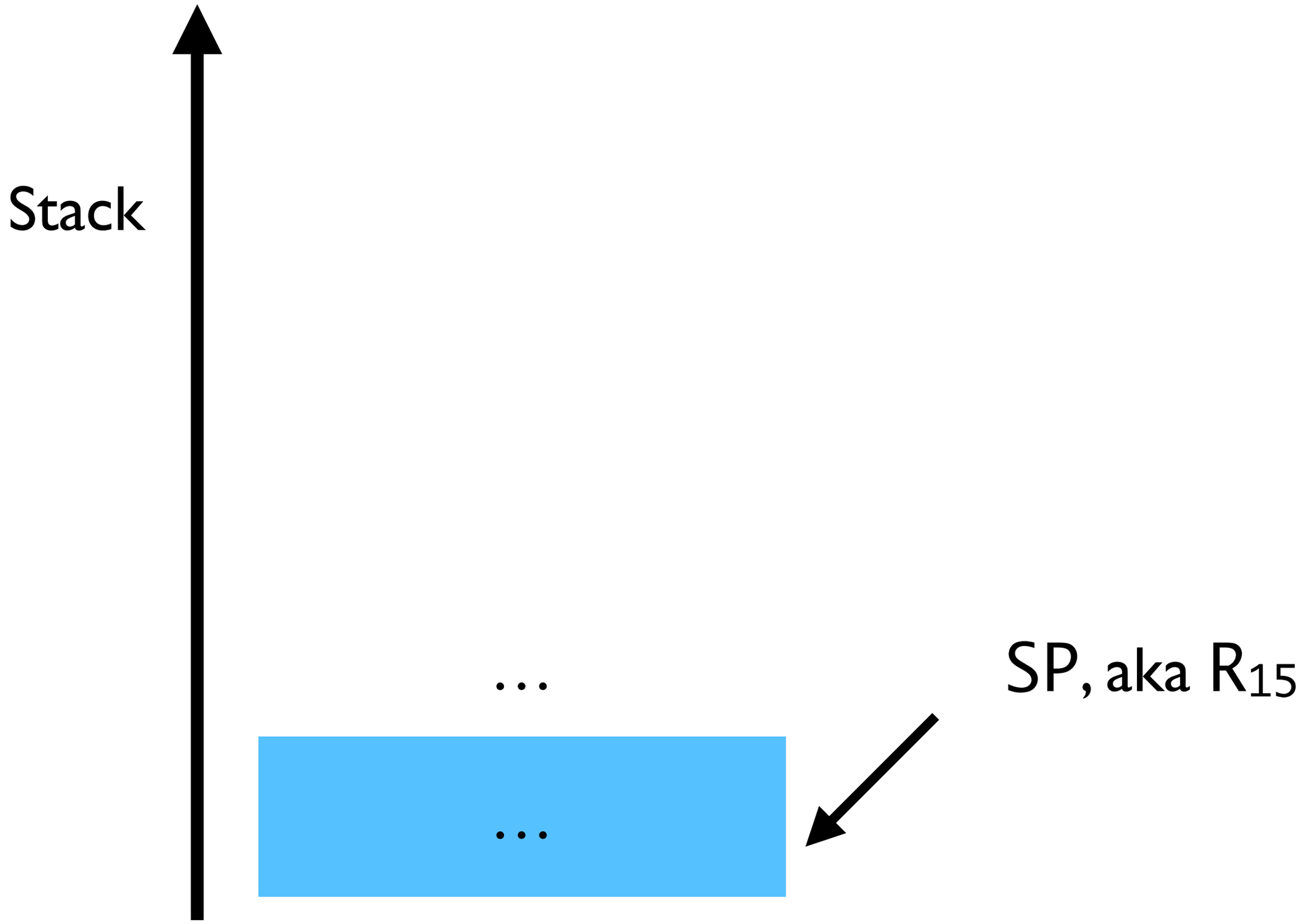
“top” of the stack



We can use a pointer to hold a reference to the top of the stack

This is typically called the **stack pointer**

The stack pointer points at the **next available word on the stack**


Stack
...
...
SP, aka R15

Stack

The stack pointer by convention always points at the top of the stack

You can change it, nothing stops you

It's just a regular register

…

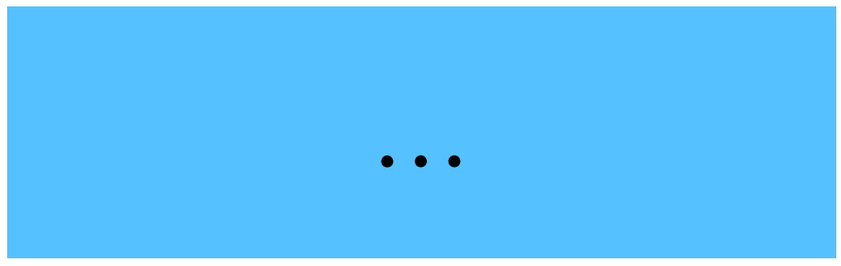
SP, aka R_{15}

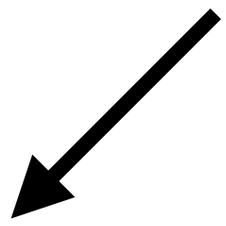
…

Stack

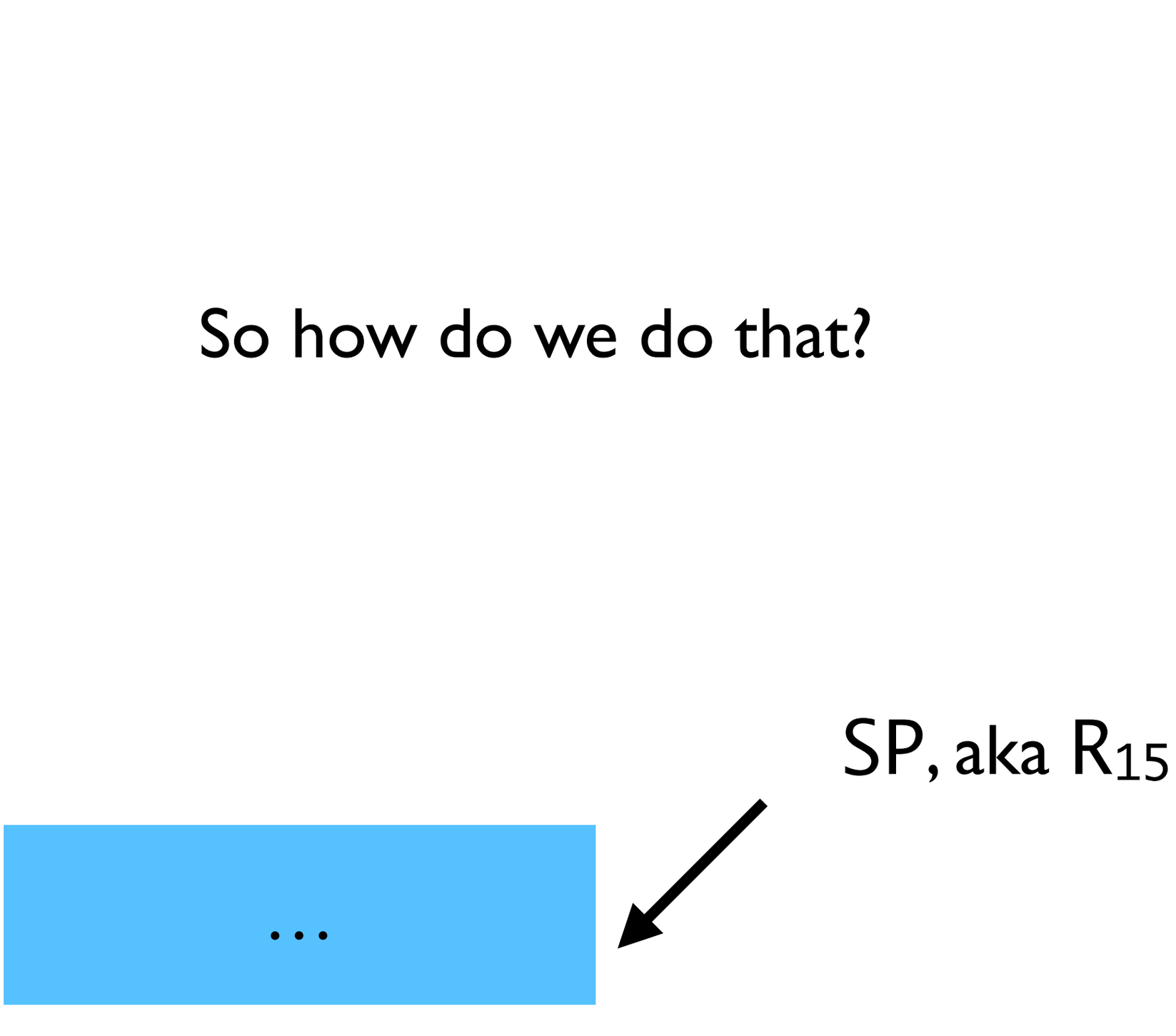
I want to make myself some space
for some local variables

SP, aka R_{15}





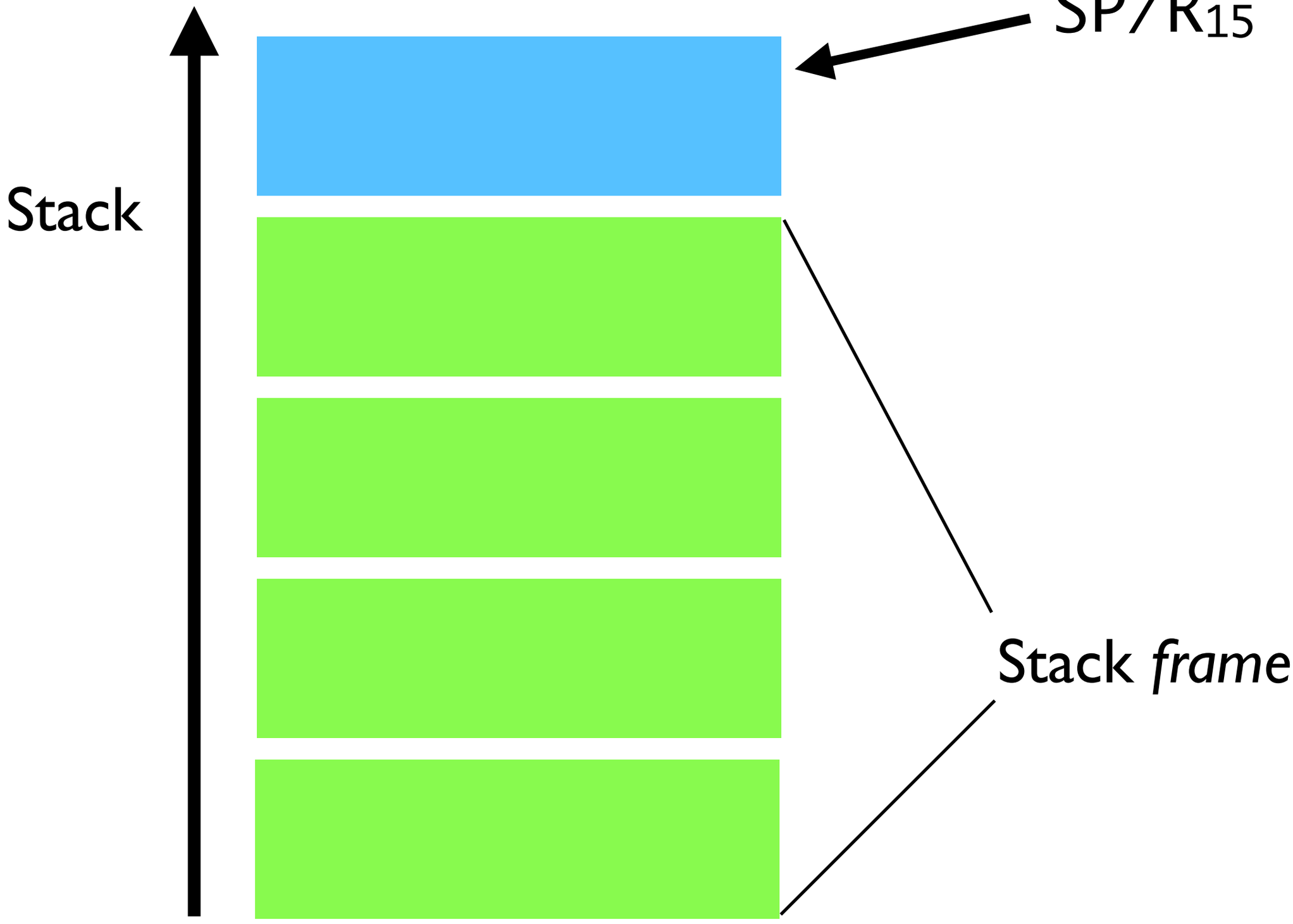
Stack



So how do we do that?

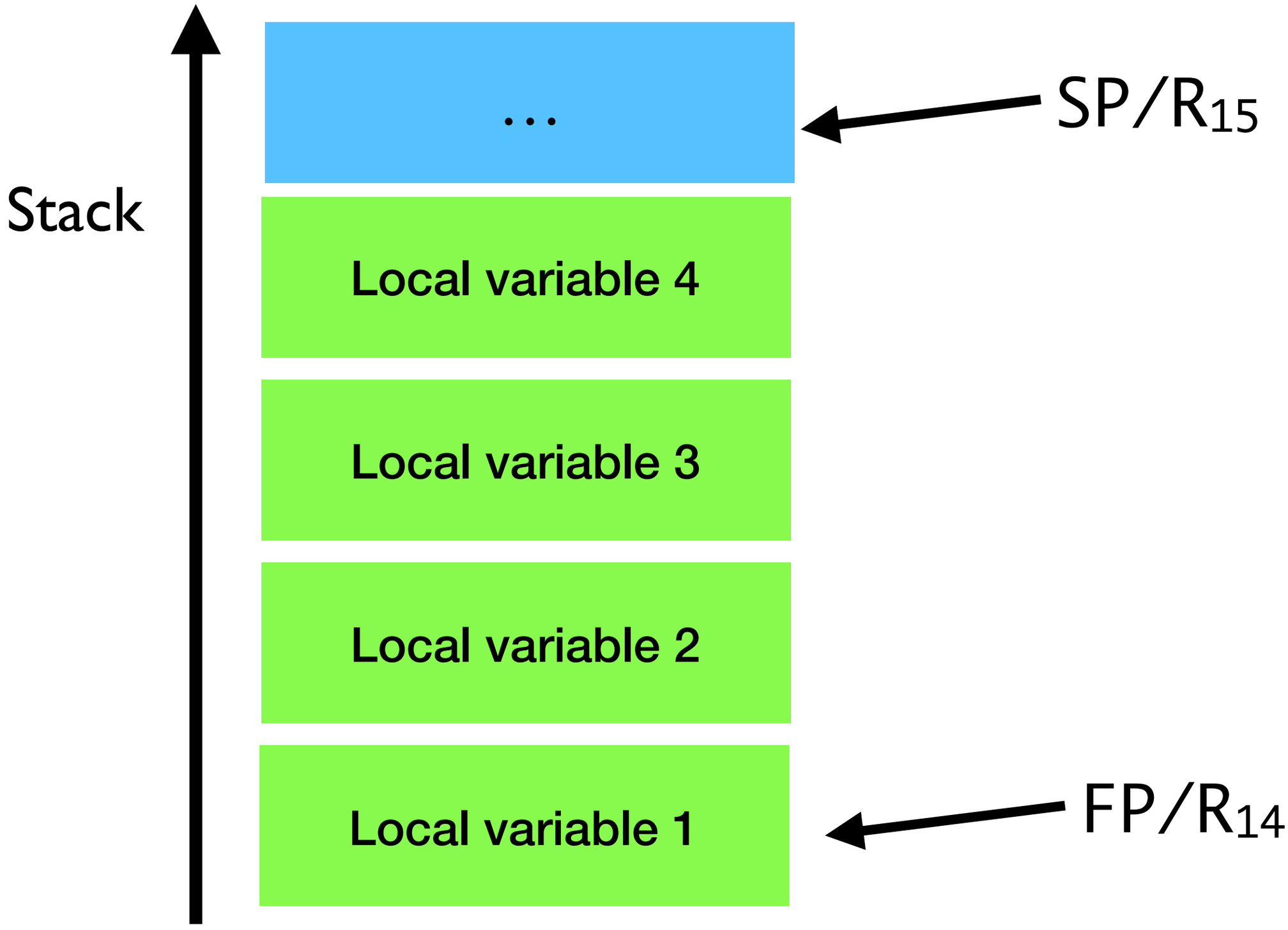
We *increment* the stack pointer

ADD(SP, 4, SP)



Stack Frame: portion of stack that holds local variables for single invocation of some function

By convention we use a *frame pointer* to refer to the base of the frame


Stack
...
SP/R15
Local variable 4
Local variable 3
Local variable 2
Local variable 1
FP/R14

Frame pointer is handy because I can now use

LOAD(R_d, o, FP)

To load from local variable o

STORE(R_b, o, FP)

To store into local variable o

This generalizes to **caller-save** convention

To call a function…

This generalizes to **caller-save** convention

To call a function…

Caller passes arguments on the stack

This generalizes to **caller-save** convention

To call a function…

Caller passes arguments on the stack

Saves registers before call

(Also save old SP, FP, and ret. addr)

This generalizes to **caller-save** convention

To call a function…

Caller passes arguments on the stack

Saves registers before call

(Also save old SP, FP, and ret. addr)

Result returned on the stack

The Big Rule

If I'm going to **change** a register, I had better **save** it first

I have two possibilities:

Either the **caller** saves the registers

Args passed in registers

("Caller save")

Or the **callee** saves the registers

Args passed on stack

("Callee save")

Caller save

Here's what I do

Let's say I'm currently using R4 and R5

And `foo` *expects* R4/R5 as arguments

R4 23 R5 89

R4 23 R5 89

And my stack looks like this…

R4 23 R5 89

Stack

And my stack looks like this…

SP, aka R_{15}

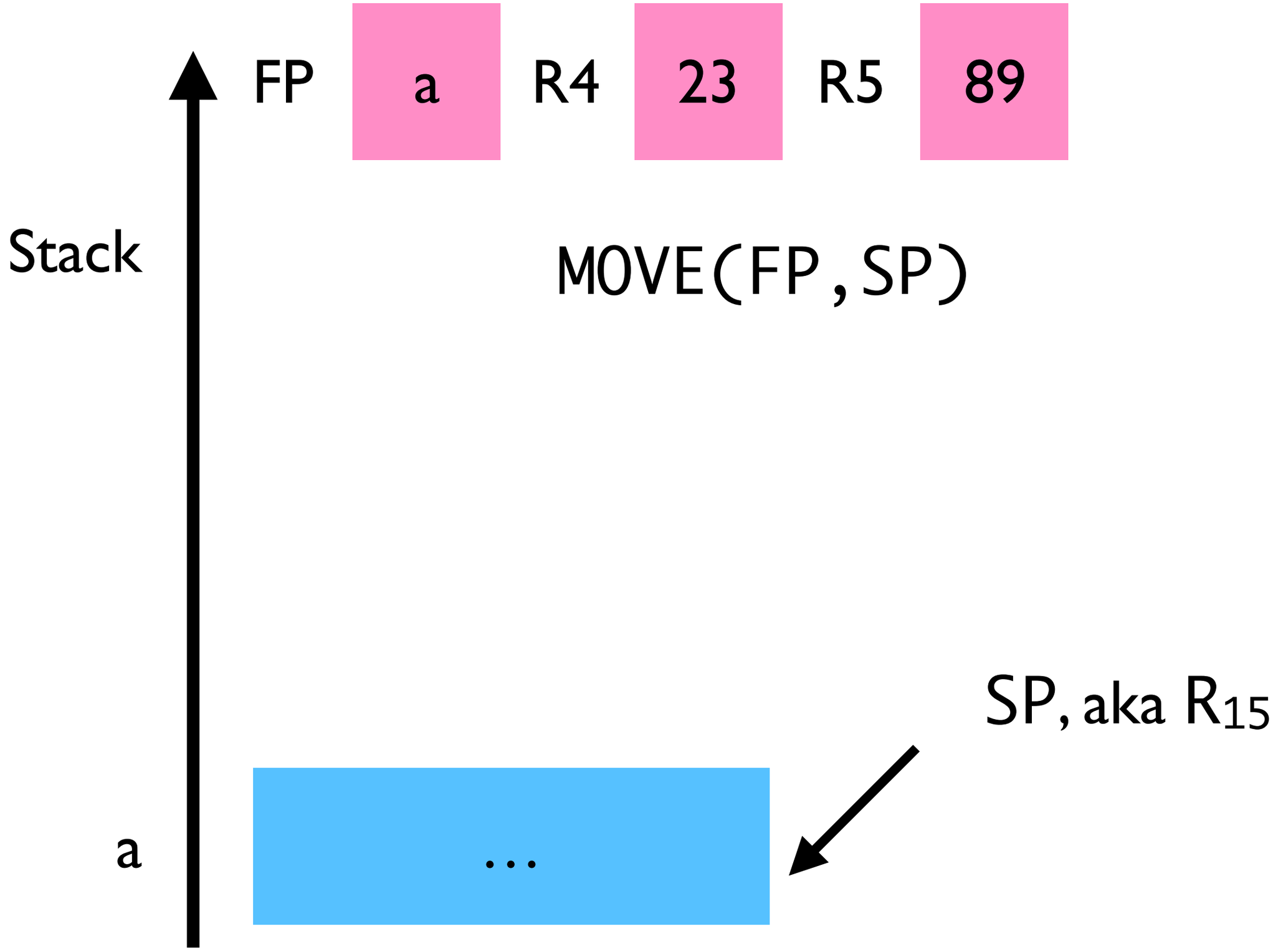
…

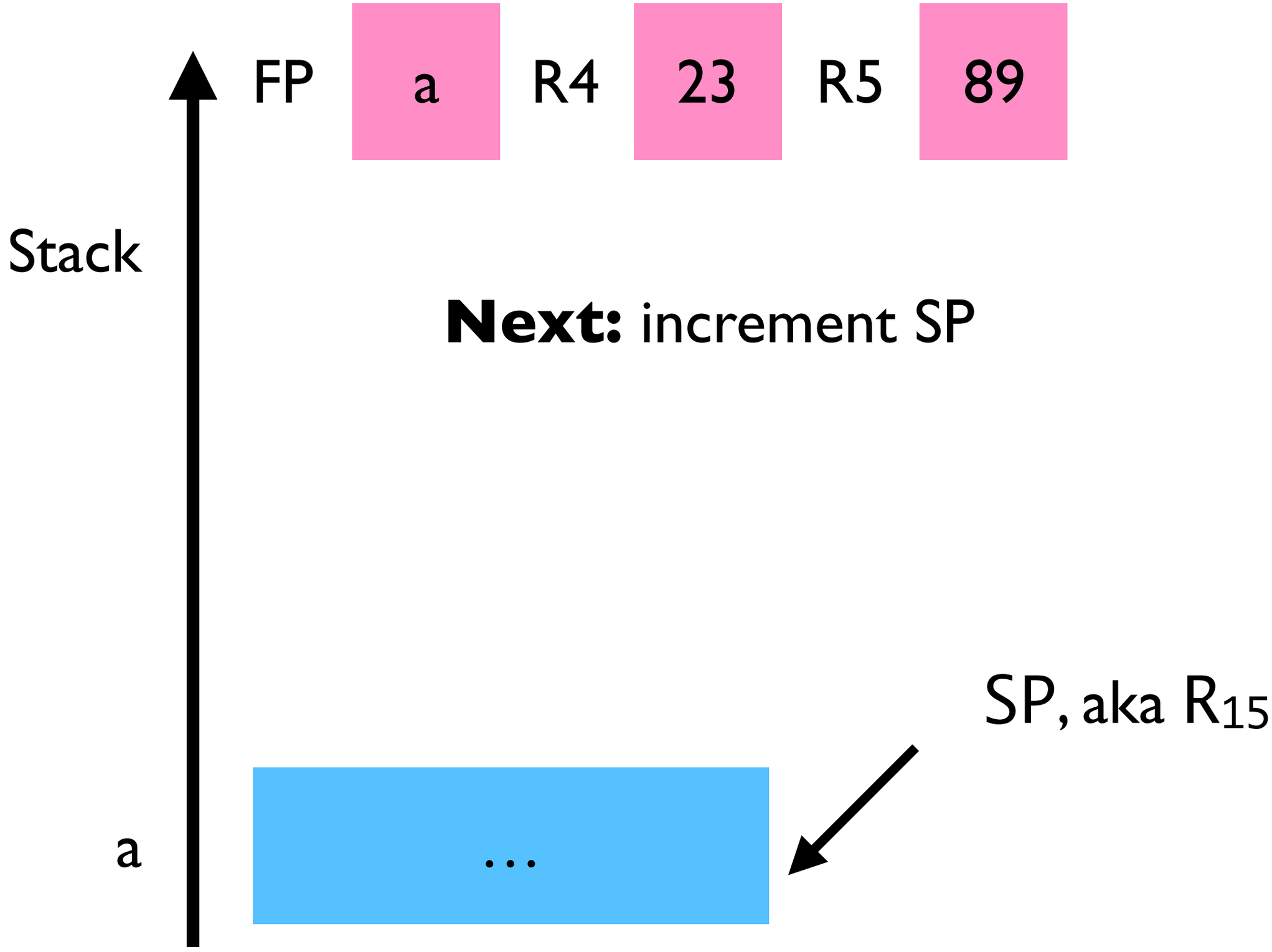
R4 23 R5 89

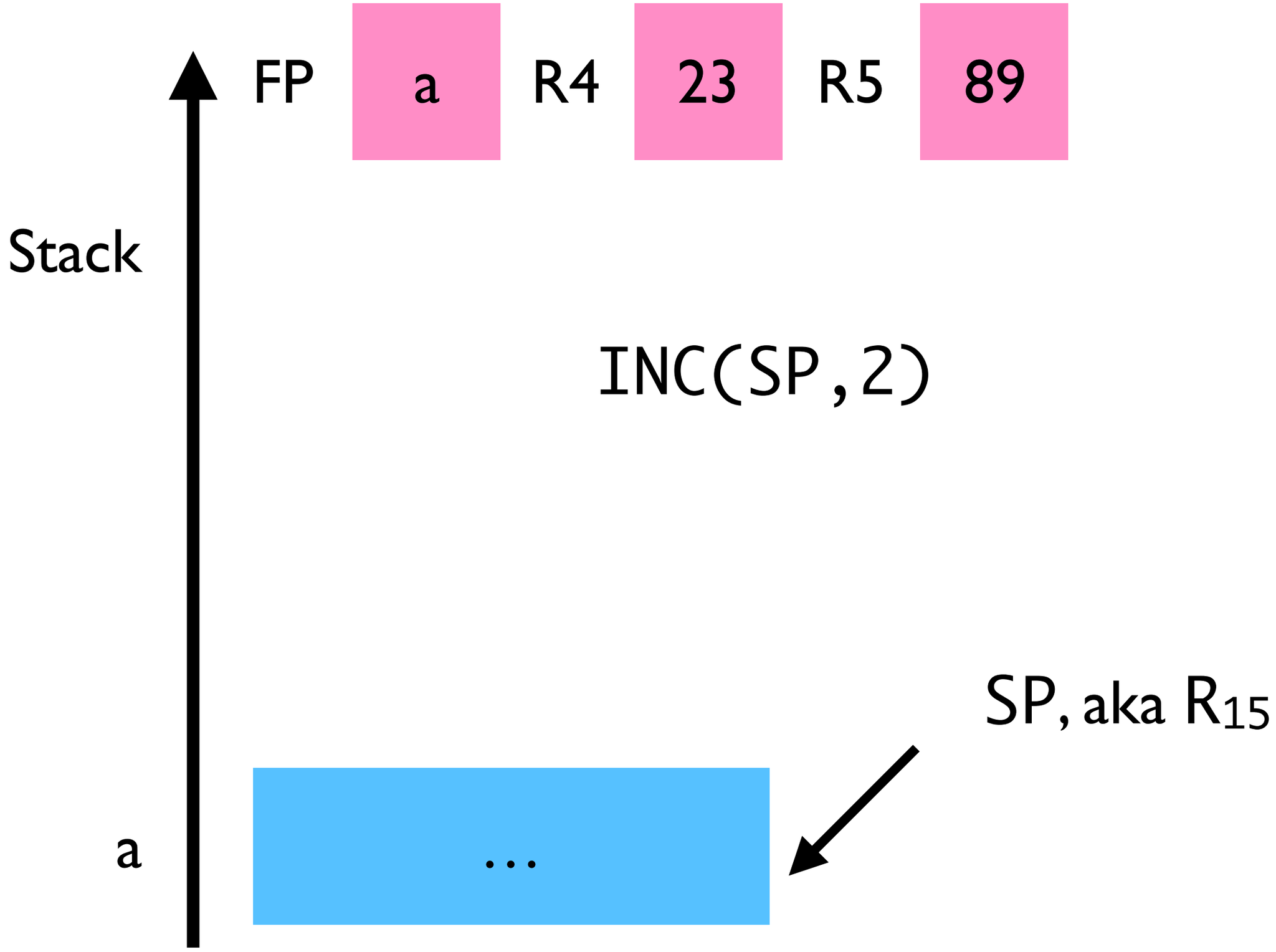
First: save stack pointer in frame pointer

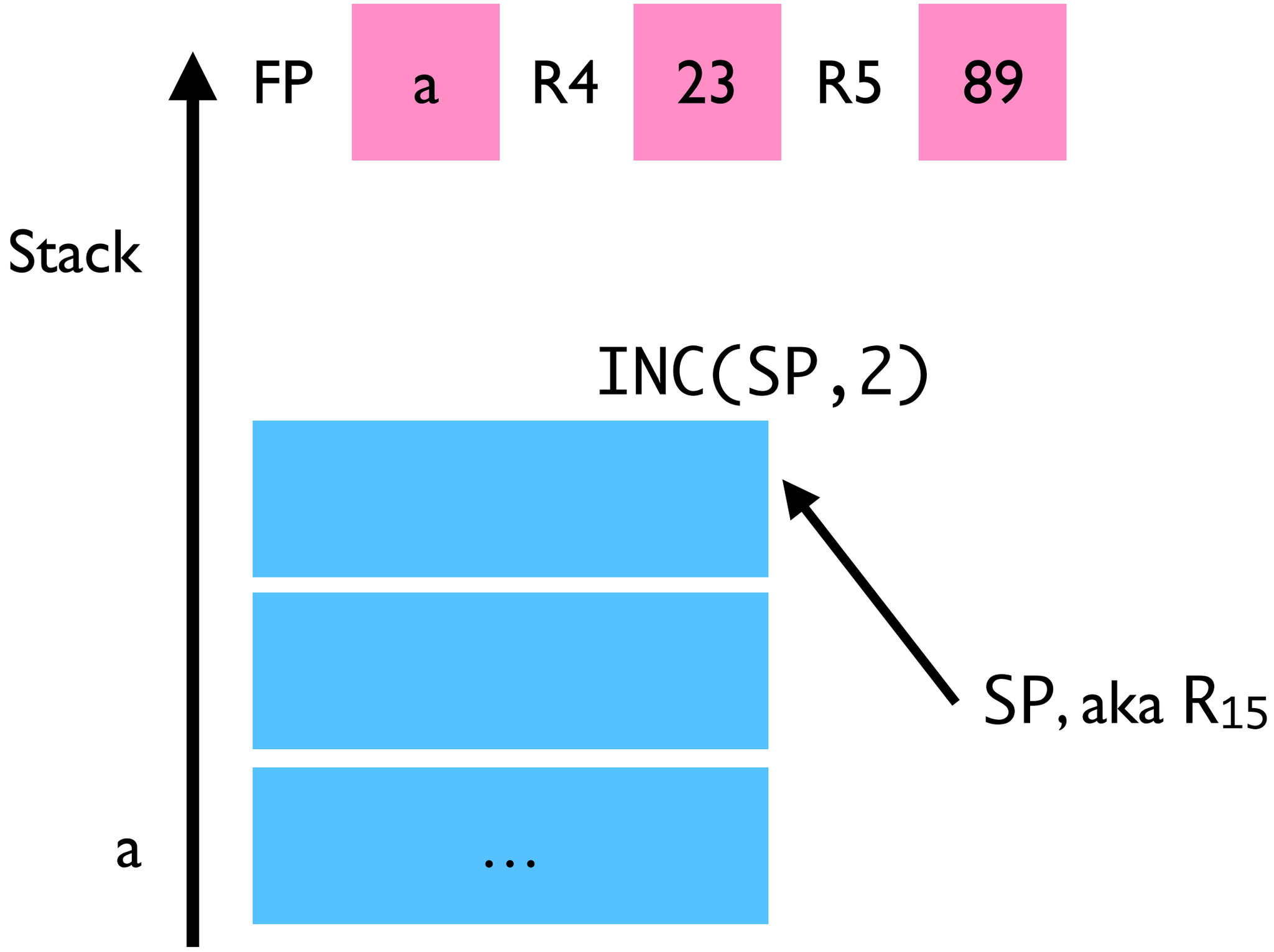
Stack

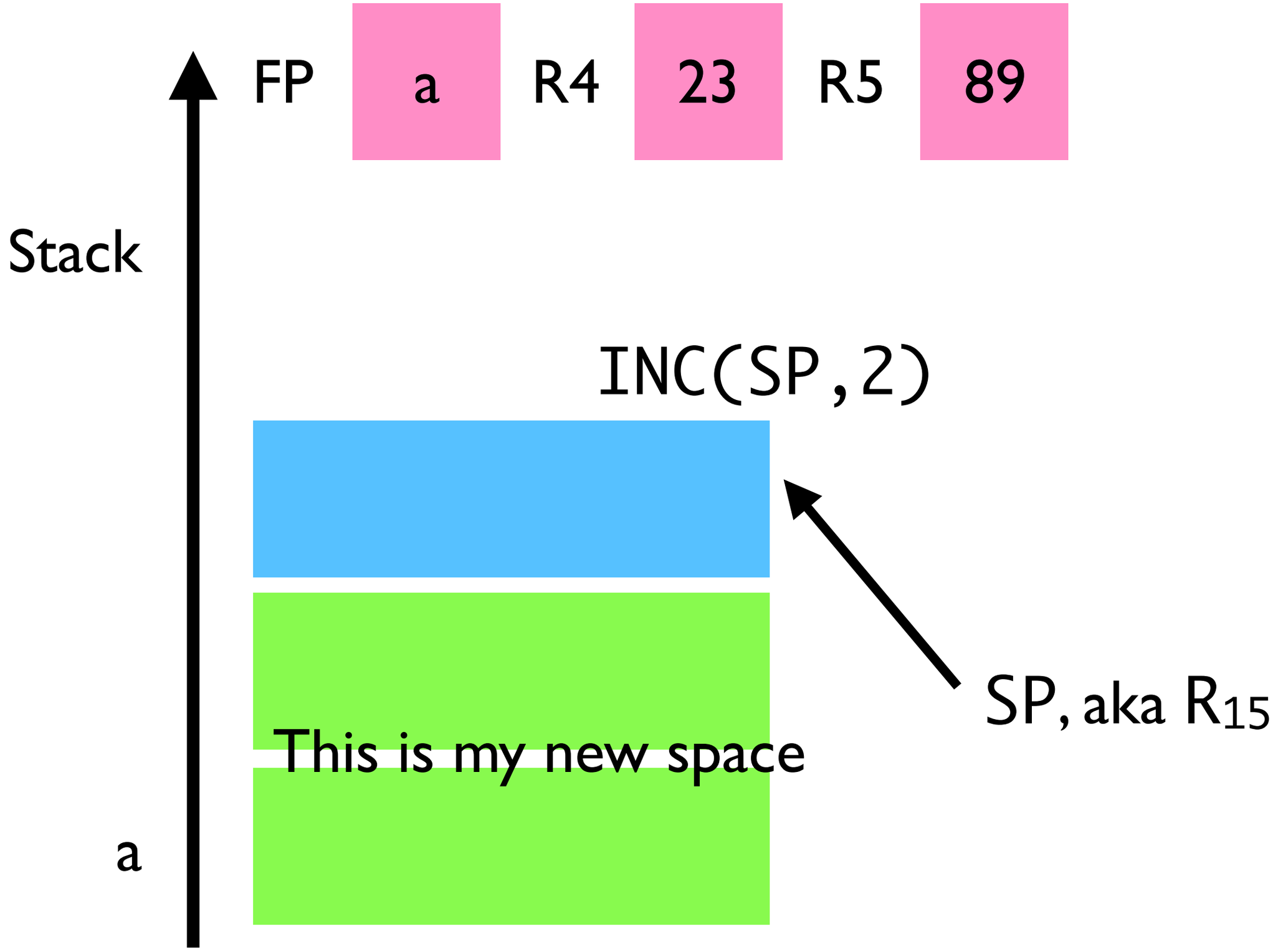
SP, aka R_{15}


FP
a
R4
23
R5
89
Stack
MOVE(FP,SP)
a
...
SP, aka R15


FP
a
R4
23
R5
89
Stack
Next: increment SP
SP, aka R15
a
...


FP
a
R4
23
R5
89
Stack
INC(SP,2)
SP, aka R15
a
...


FP
a
R4
23
R5
89
Stack
INC(SP,2)
SP, aka R15
a
...


FP
a
R4
23
R5
89
Stack
INC(SP,2)
This is my new space
SP, aka R15
a

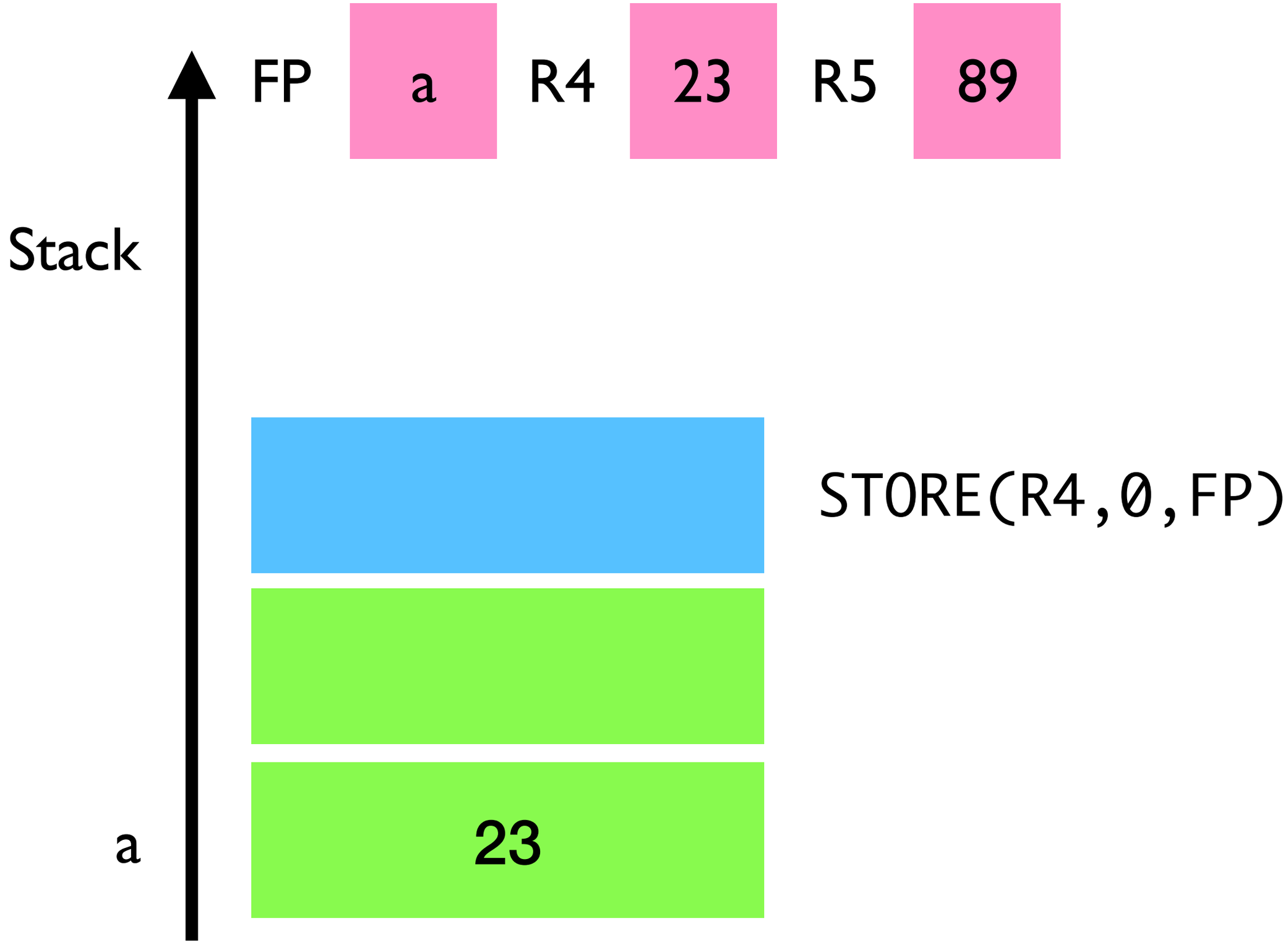
Next: save R4 and R5 there

FP a R4 23 R5 89

Stack

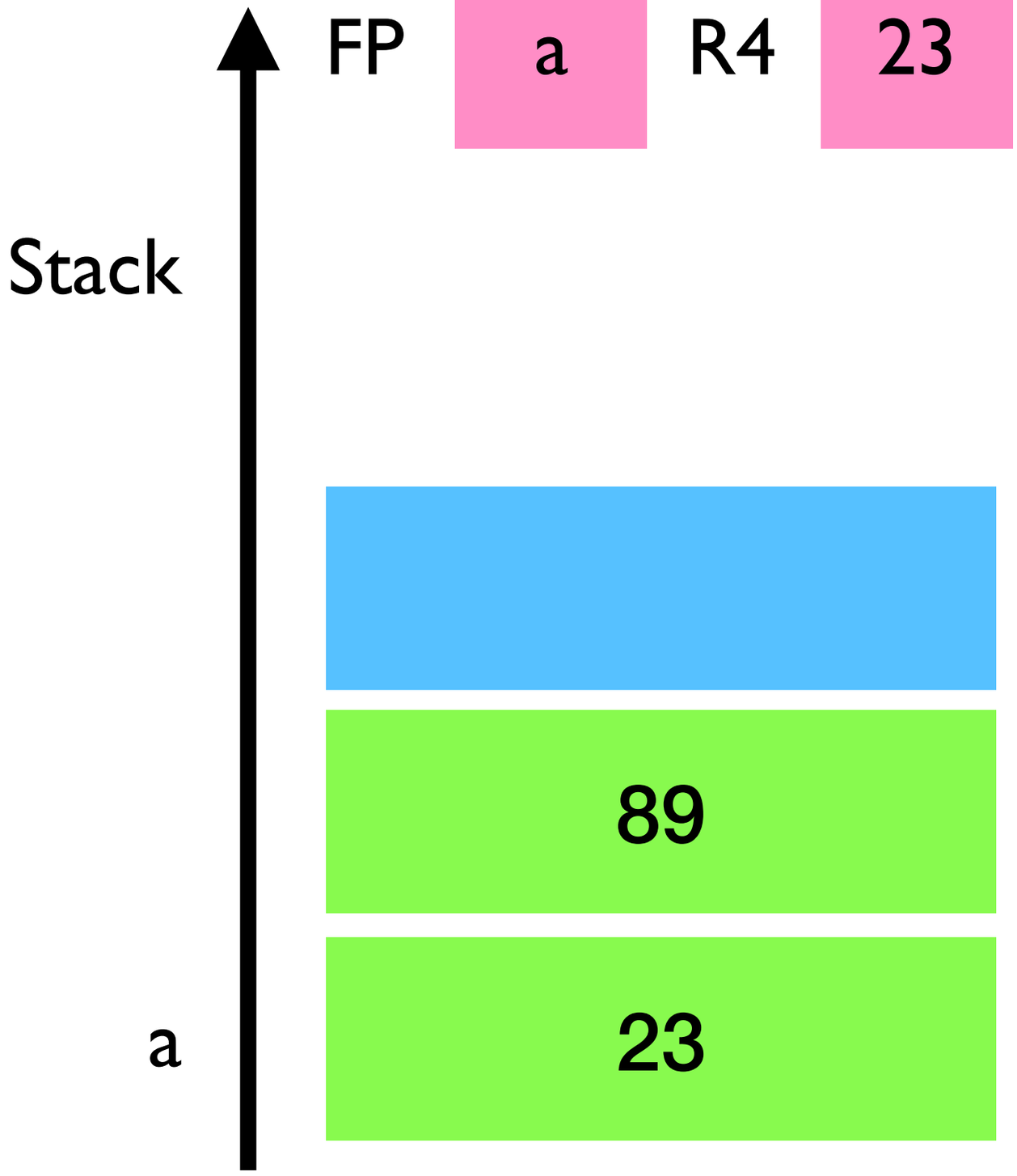
STORE(R4,0,FP)

a


FP
a
R4
23
R5
89
Stack
STORE(R4,0,FP)
23
a

FP a R4 23 R5 89

Stack

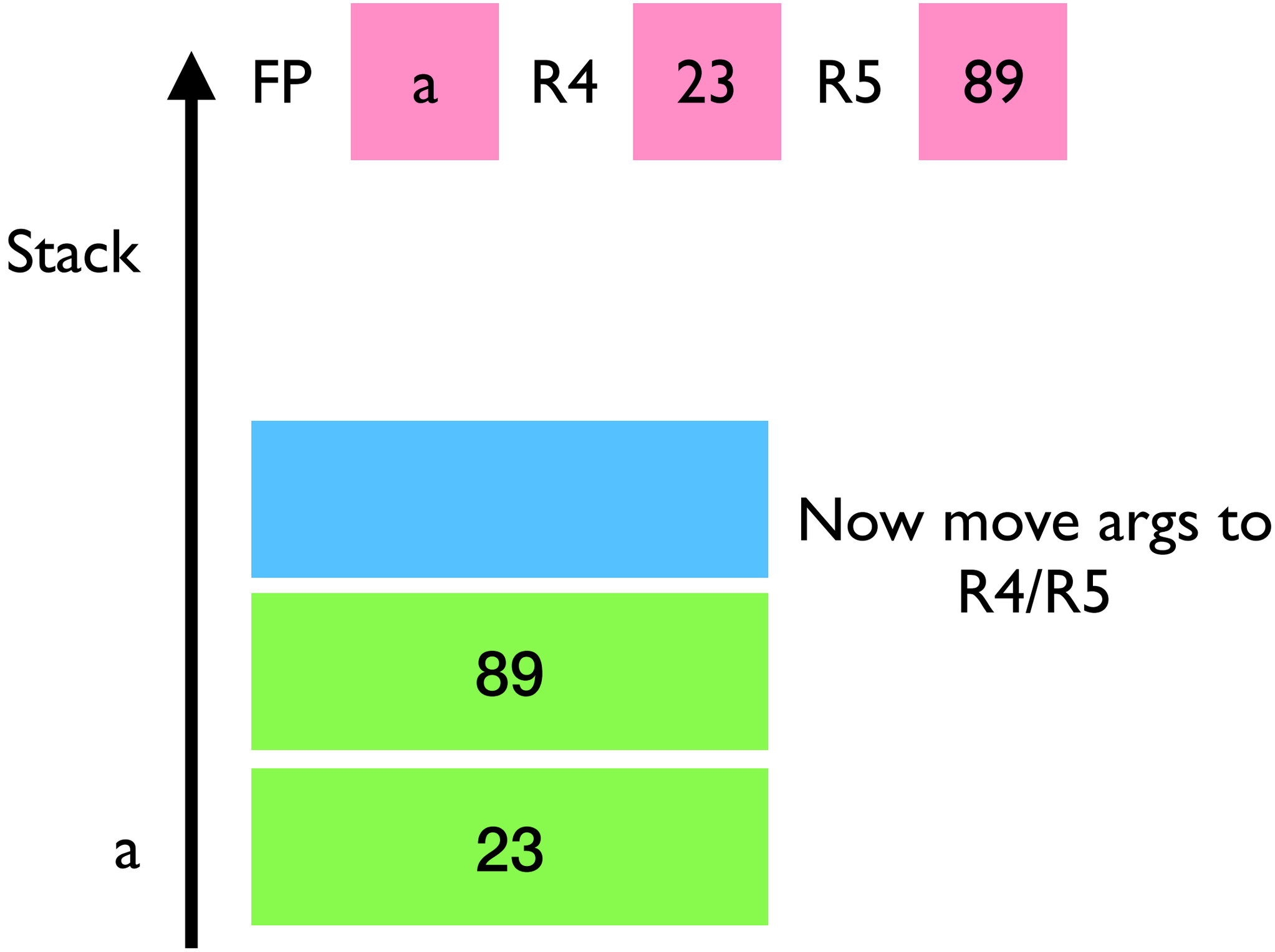


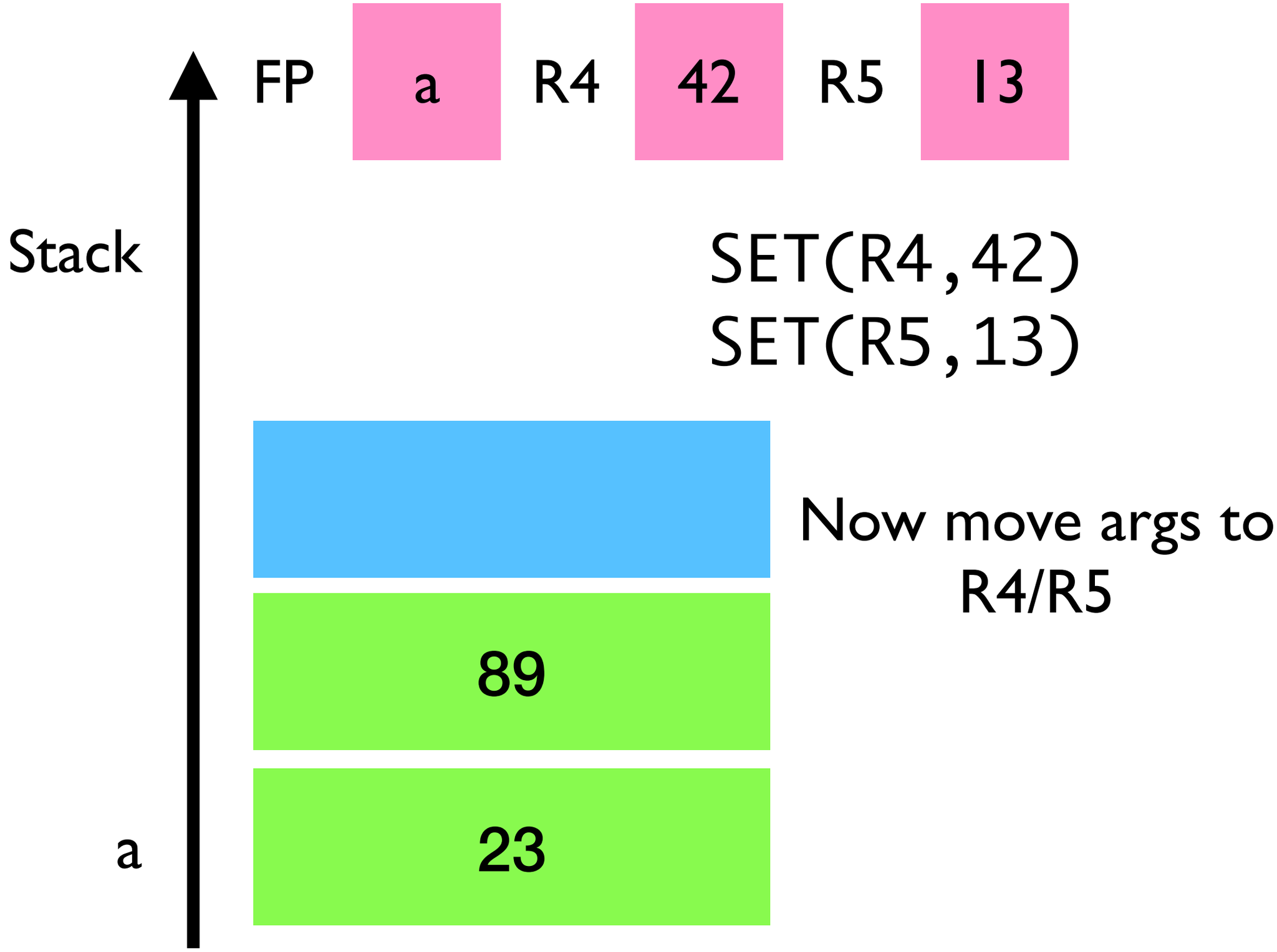
STORE(R5,1,FP)

89

23

a


FP
a
R4
23
R5
89
Stack
a
89
23
Now move args to R4/R5


FP
a
R4
42
R5
13
Stack
SET(R4,42)
SET(R5,13)
89
23
a
Now move args to R4/R5

FP

a

R4

42

R5

13

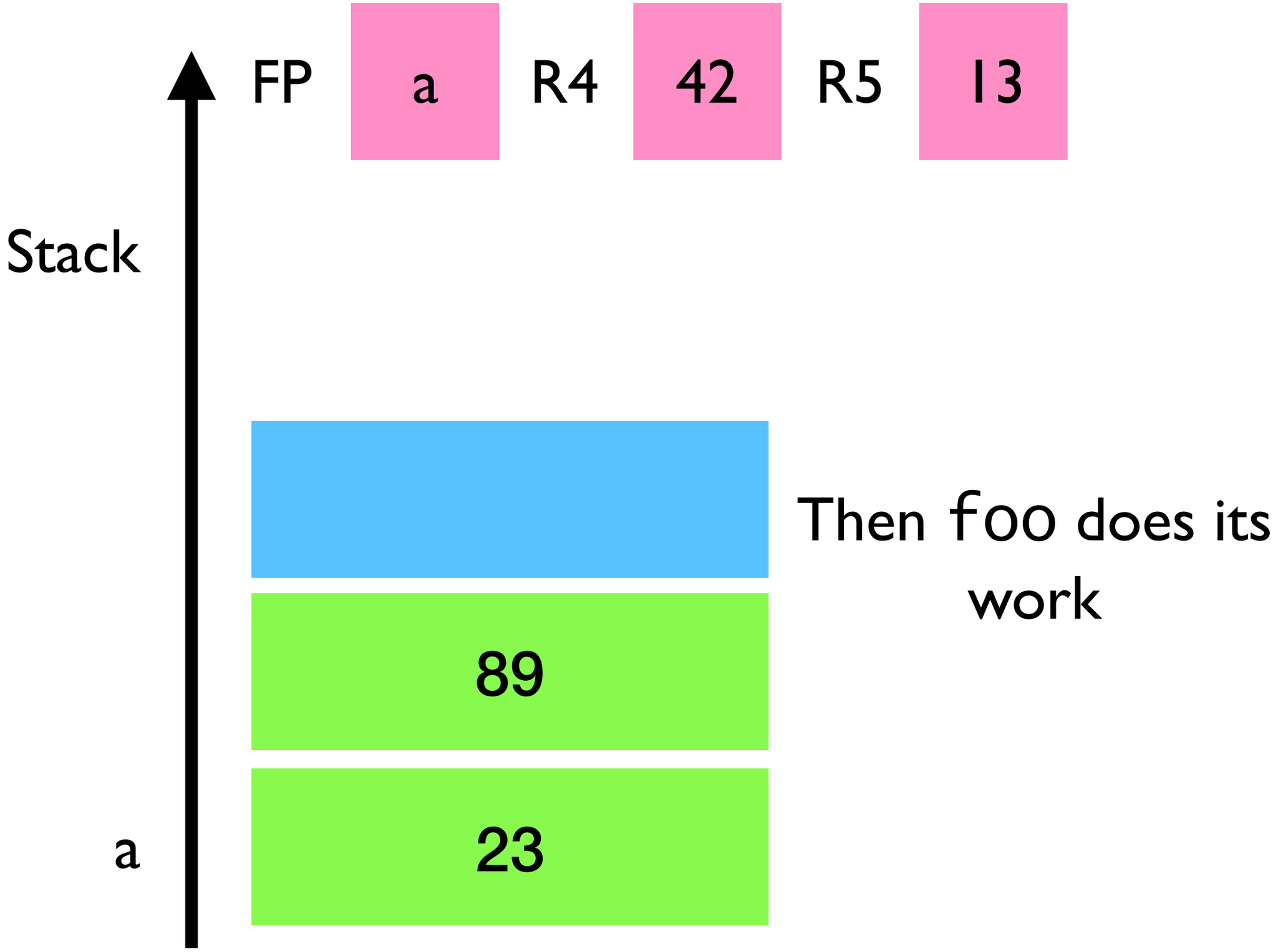
Stack

a

89

23

Now I'm ready to call foo!



Then foo does its work

FP a R4 **10** R5 13

Stack

Assume it returns result in R4

a

89

23

FP a R4 **10** R5 13

Stack

a

89

23

Now I need to **decrement** SP

FP a R4 **10** R5 13

Stack

Now I need to
decrement SP

SUB(SP,2,SP)

SP

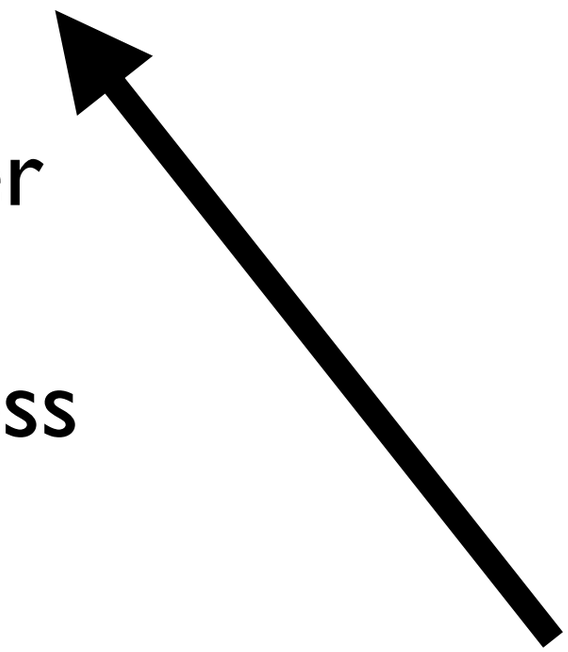
a

Note: when we call a function, we **may** also need to save three other registers…

- Stack pointer
- Frame pointer
- Return address

Note: when we call a function, we **may** also need to save three other registers…

- Stack pointer
- Frame pointer
- Return address



Remember, I'm going to increment the stack pointer

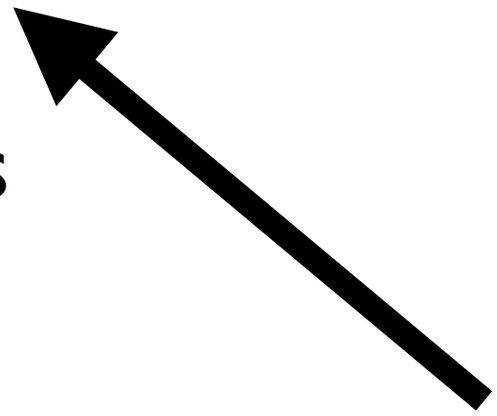
Note: when we call a function, we **may** also need to save three other registers…

- Stack pointer
  - If I didn't, I'd forget where the stack was
- Frame pointer

- Return address

Note: when we call a function, we **may** also need to save three other registers…

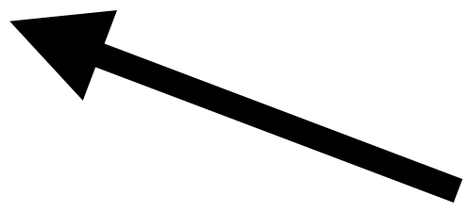
- Stack pointer
  - If I didn't, I'd forget where the stack was
- Frame pointer
- Return address



I'm going to add a frame, too…

Note: when we call a function, we **may** also need to save three other registers…

- Stack pointer
  - If I didn't, I'd forget where the stack was
- Frame pointer
  - If I didn't, I'd forget where the frame was
- Return address



I'm going to update the return address…

Note: when we call a function, we **may** also need to save three other registers…

- Stack pointer
  - If I didn't, I'd forget where the stack was
- Frame pointer
  - If I didn't, I'd forget where the frame was
- Return address
  - If I didn't, I'd forget where to go next

CALL and RETURN instructions

CALL(R_a, R_b)

Calls function at address R_b with new stack frame starting at R_a

Typically you will use the current SP for R_a

CALL(R_a, R_b)

$PC \leftarrow R_b$

$R_b \leftarrow PC + 1$

$FP \leftarrow R_a$

$R_a \leftarrow FP$

So CALL handles all of the common things to save FP/SP/RT and then set the frame pointer appropriately…

Caller save

Code using foo

```
MOVE(FP,SP)
INC(SP,1)
STORE(R4,0,FP)
SET(R2,foo)
CALL(SP,R2)
LOAD(R4,0,FP)
SUB(SP,1,SP)
```

Definition of foo

```
LABEL(getValue)
  LOAD(R3, 0, R1)
  SET(R4, 0)
  BR(R2)
  RETURN()
```

Callee save

In **callee** save, arguments are *passed* on stack,
and result *returned* on stack

```
int two_x_plus_y(int x, int y) {
  return x+x+y;
}
```

This is how I'm going to call `two_x_plus_y`

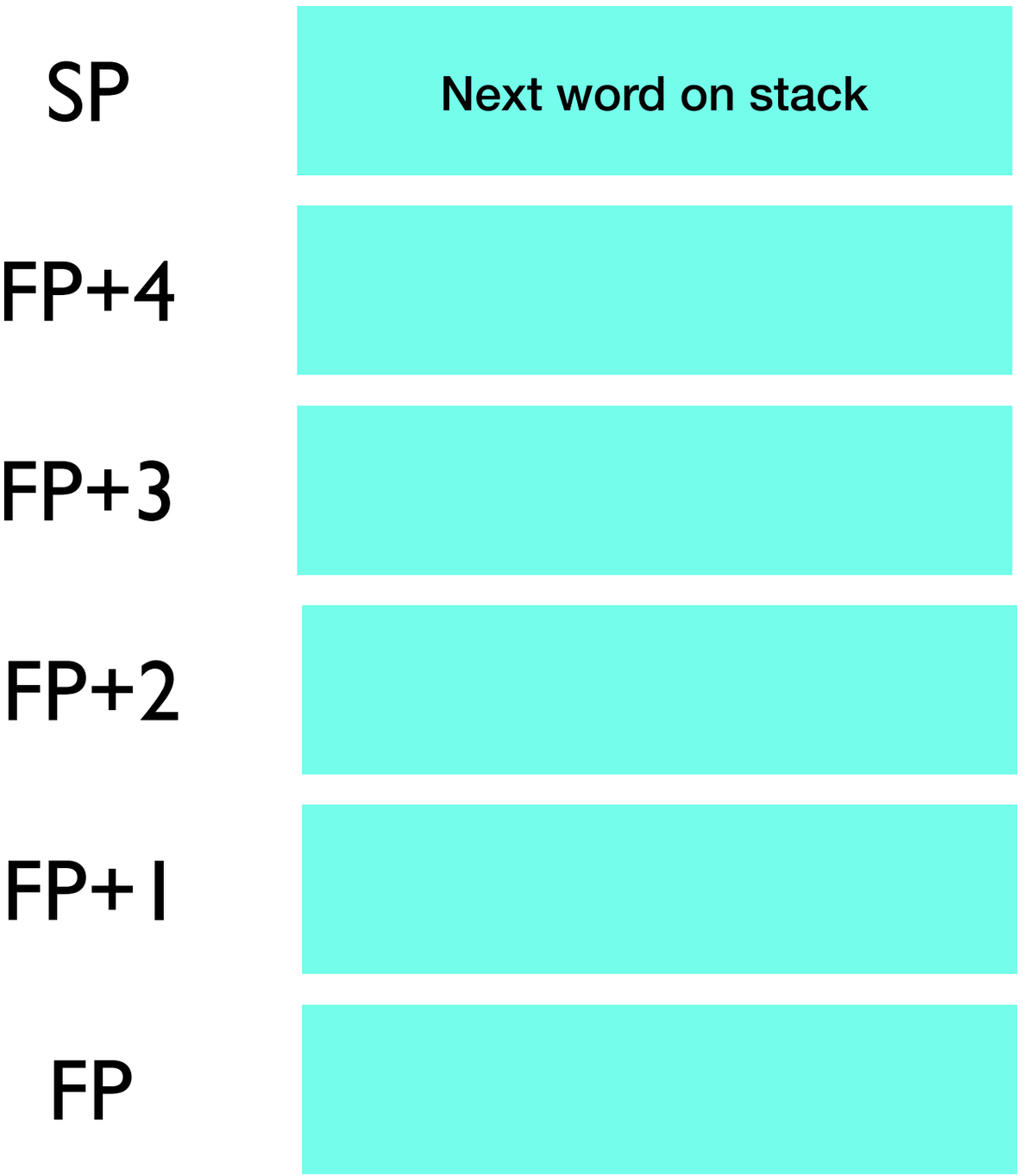
```
SETCB()
MOVE(R12, SP)
INC(SP, 5)
SET(r7,2)
STORE(r7, 3,R12)          // parameter 1 = 2
SET(r7,10)
STORE(r7, 4,R12)          // parameter 2 = 10
CALL(R12, foo)
LOAD(r1,  3,R12)          // retrieve result
DEC(SP, 5)
```

```
SETCB()
MOVE(R12, SP)
INC(SP, 5)
SET(r7,2)
STORE(r7, 3,R12)
SET(r7,10)
STORE(r7, 4,R12)
CALL(R12, foo)
LOAD(r1,  3,R12)
DEC(SP, 5)
```

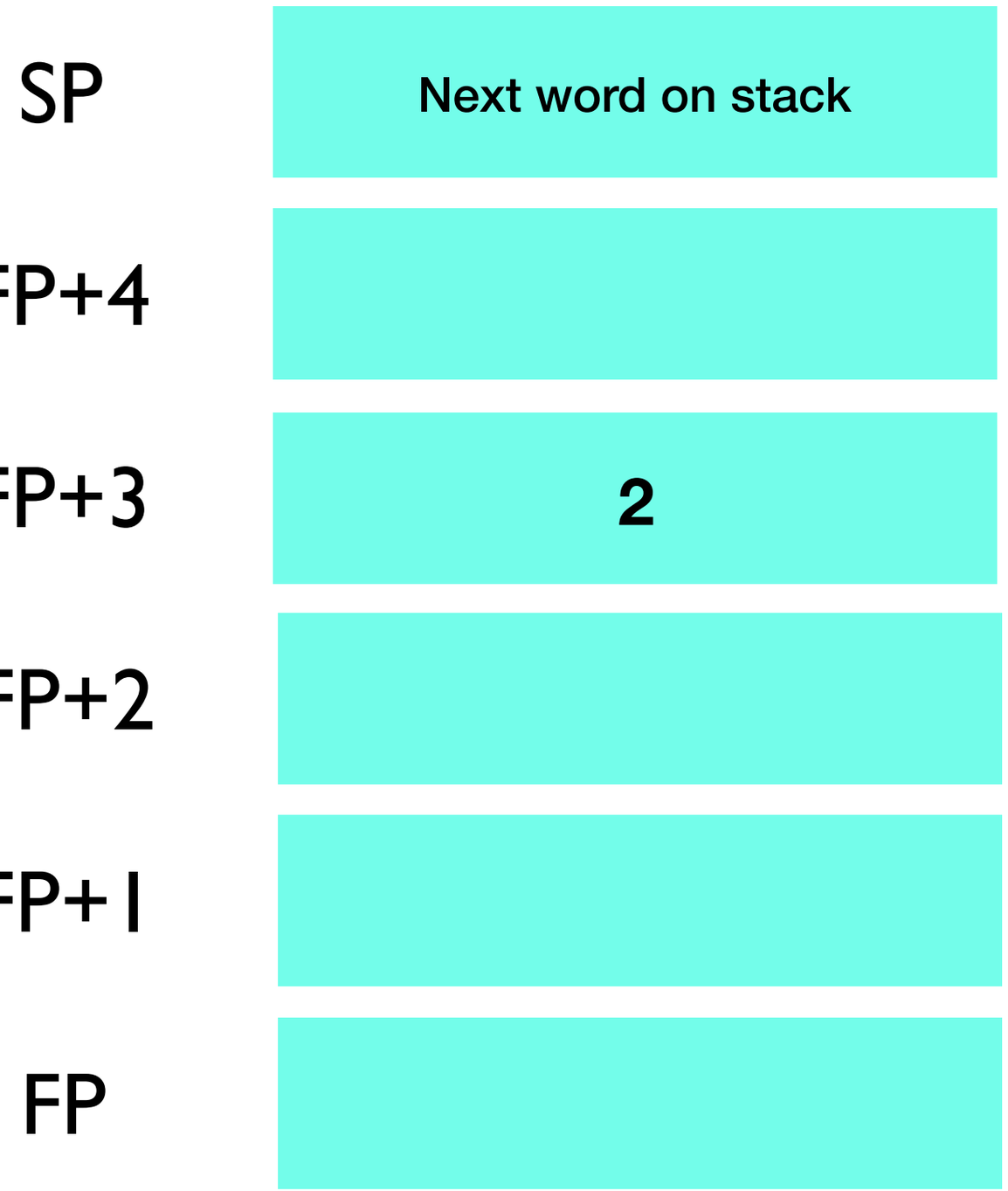
SP

Next word on stack

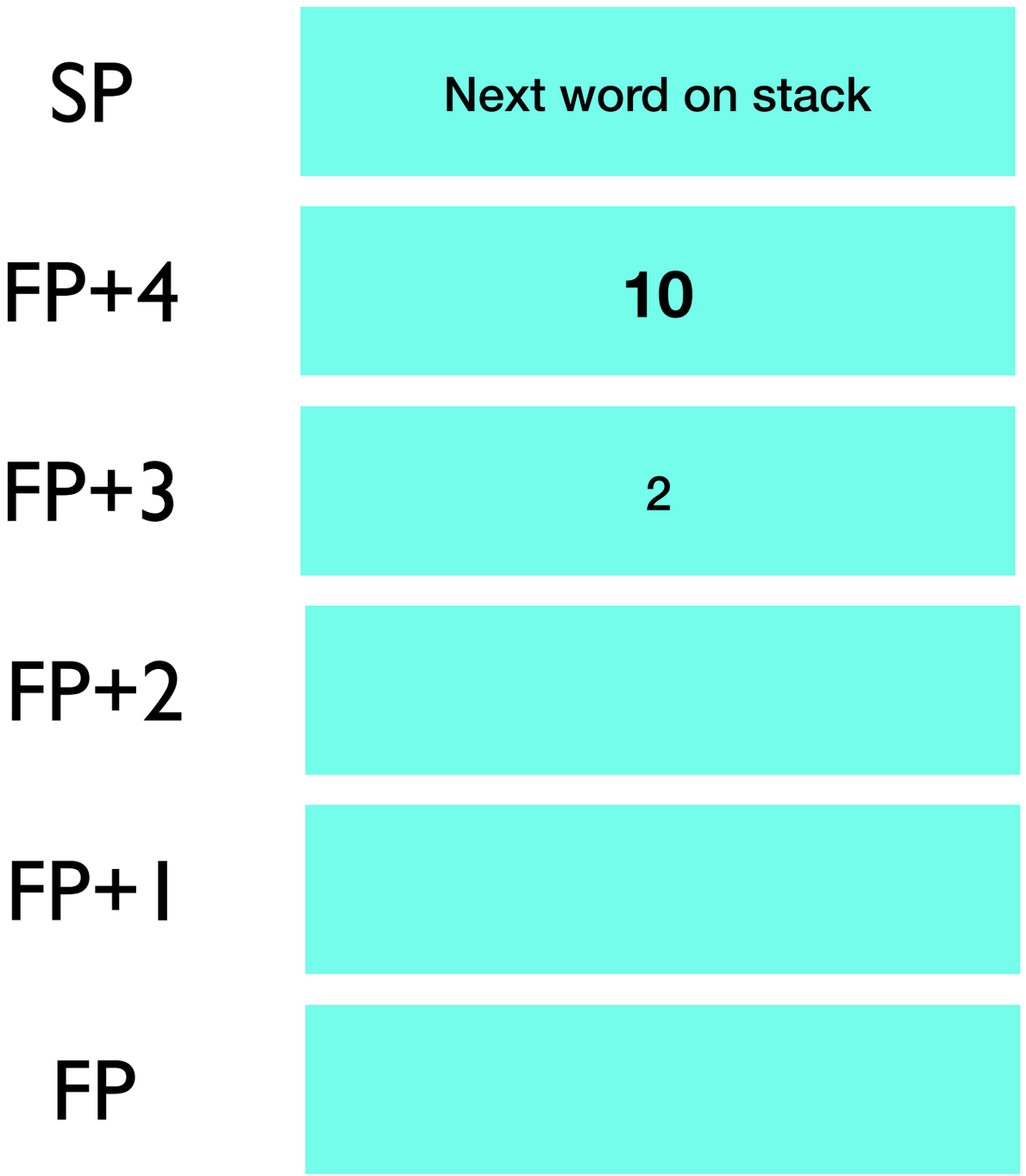
```
SETCB()
MOVE(R12, SP)
INC(SP, 5)
SET(r7,2)
STORE(r7, 3,R12)
SET(r7,10)
STORE(r7, 4,R12)
CALL(R12, foo)
LOAD(r1,  3,R12)
DEC(SP, 5)
```



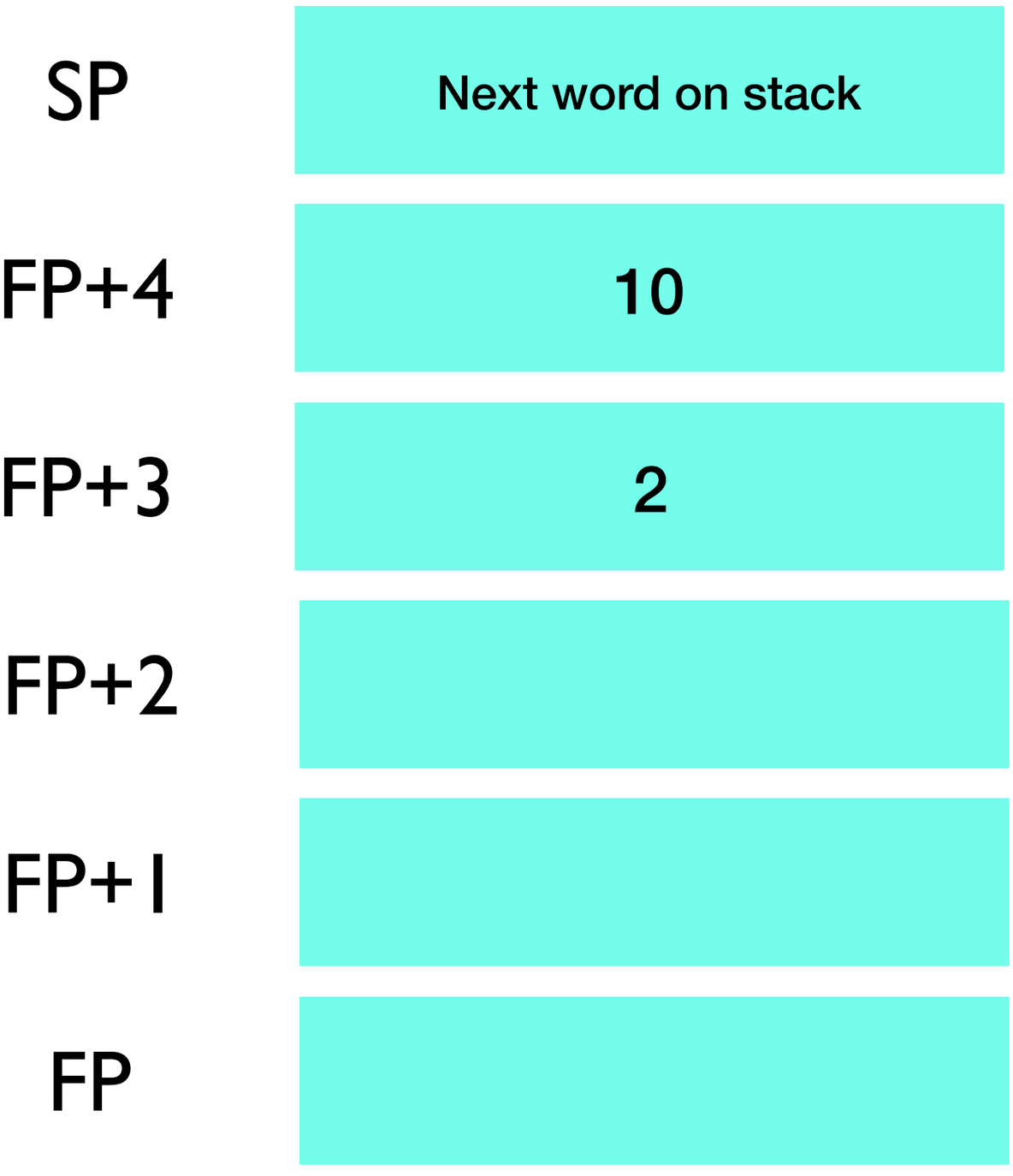
```
SETCB()
MOVE(R12, SP)
INC(SP, 5)
SET(r7,2)
STORE(r7, 3,R12)
SET(r7,10)
STORE(r7, 4,R12)
CALL(R12, foo)
LOAD(r1,  3,R12)
DEC(SP, 5)
```



```
SETCB()
MOVE(R12, SP)
INC(SP, 5)
SET(r7,2)
STORE(r7, 3,R12)
SET(r7,10)
STORE(r7, 4,R12)
CALL(R12, foo)
LOAD(r1,  3,R12)
DEC(SP, 5)
```



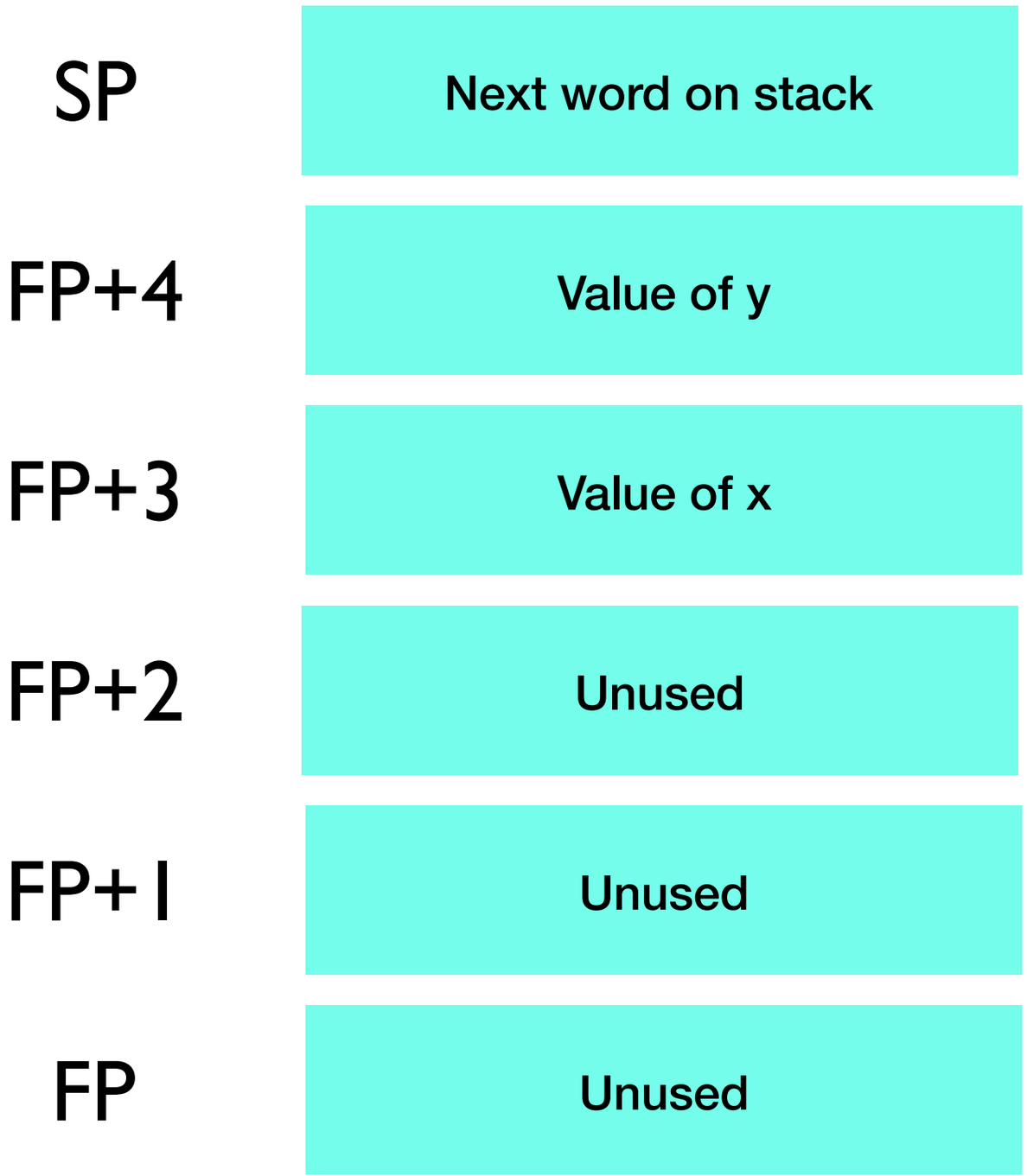
`two_x_plus_y` receives this



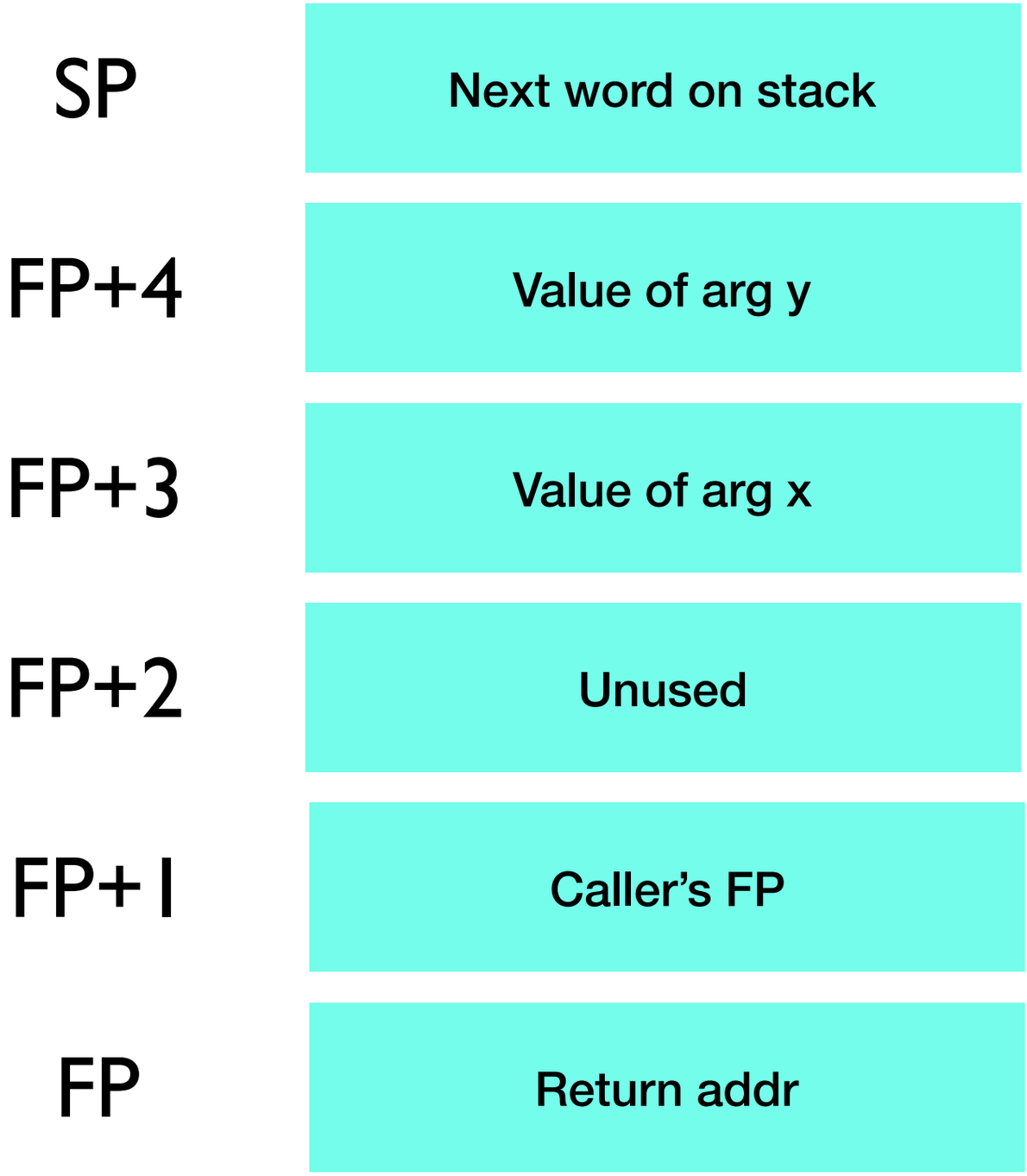
```
LABEL(two_x_plus_y)
// FIRST, make space to save r1 and r2 and then save Rt, R12, and them
    INC(SP, 2)
    STORE(Rt, 0,FP)     // Return address, i.e. PC+1 from before CALL
    STORE(R12, 1,FP)    // Control Link, i.e. FP from before the CALL
    STORE(r1, 5,FP)     // skip FP+3 and FP+4, where a and b will be...
    STORE(r2, 6,FP)     //   ... and save r1 and r2 in FP+5/FP+6

// Load "x" and "y" from stack frame (calling func. put them there)
    LOAD(r1, 3,FP)      // r1 = x
    LOAD(r2, 4,FP)      // r2 = y

// Compute the result
    ADD(r1, r1,r1)      // r1 = x+x
    ADD(r1, r1,r2)      // r1 = x+x+y

// Store the result where the caller of two_x_plus_y will find it
    STORE(r1, 3,FP)

// FINALLY, restore registers (including Rt and R12) and return
    LOAD(r2, 6,FP)      // Restore r2
    LOAD(r1, 5,FP)      // Restore r1
    LOAD(Rt, 0,FP)      // Restore Rt to provide R.A. for return
    LOAD(R12, 1,FP)     // Restore R12 (C.L.) to provide old FP for return
    DEC(SP, 2)
    RETURN(R12, Rt)
```

Let’s dive into this code…

What we've got now..

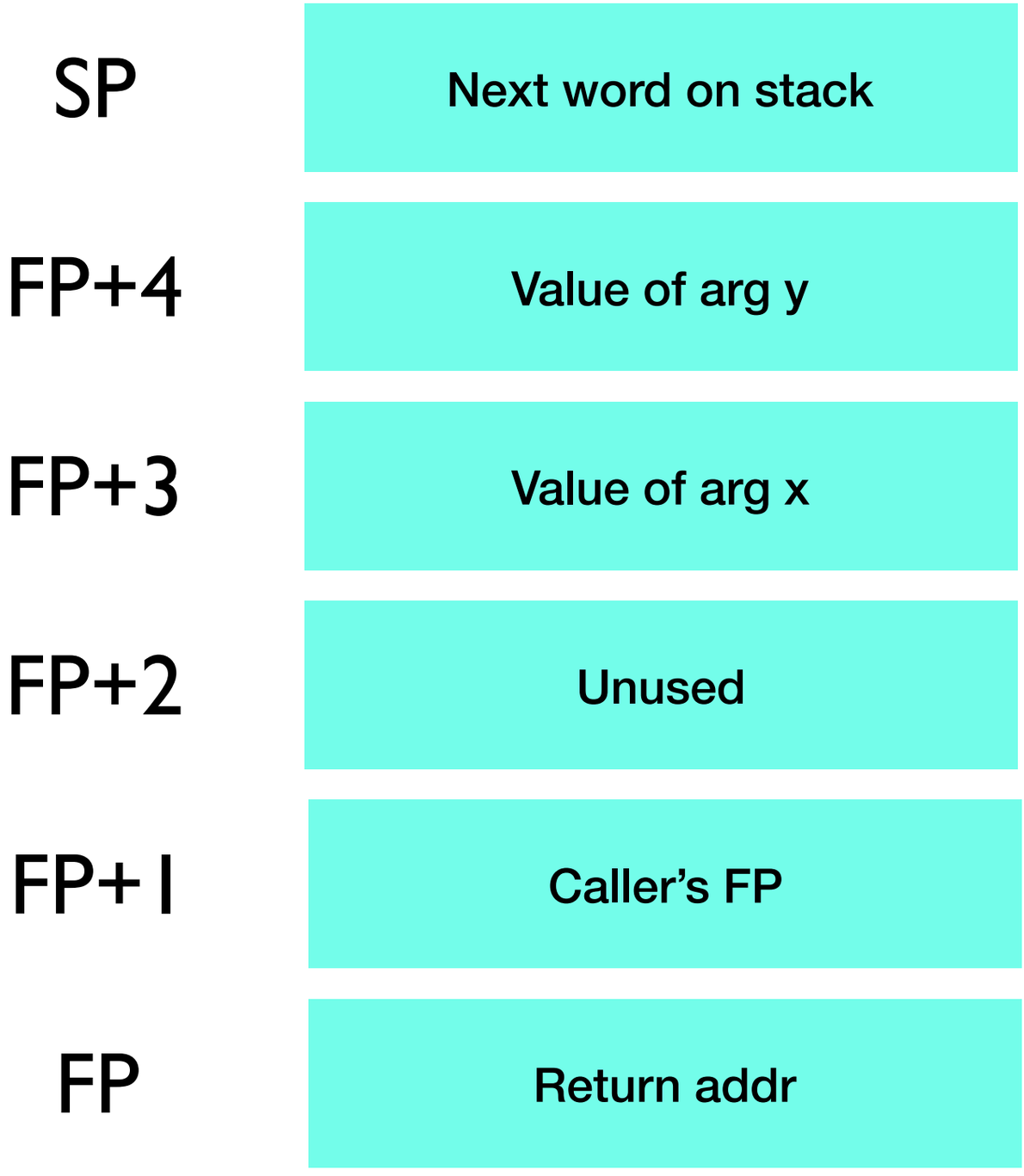


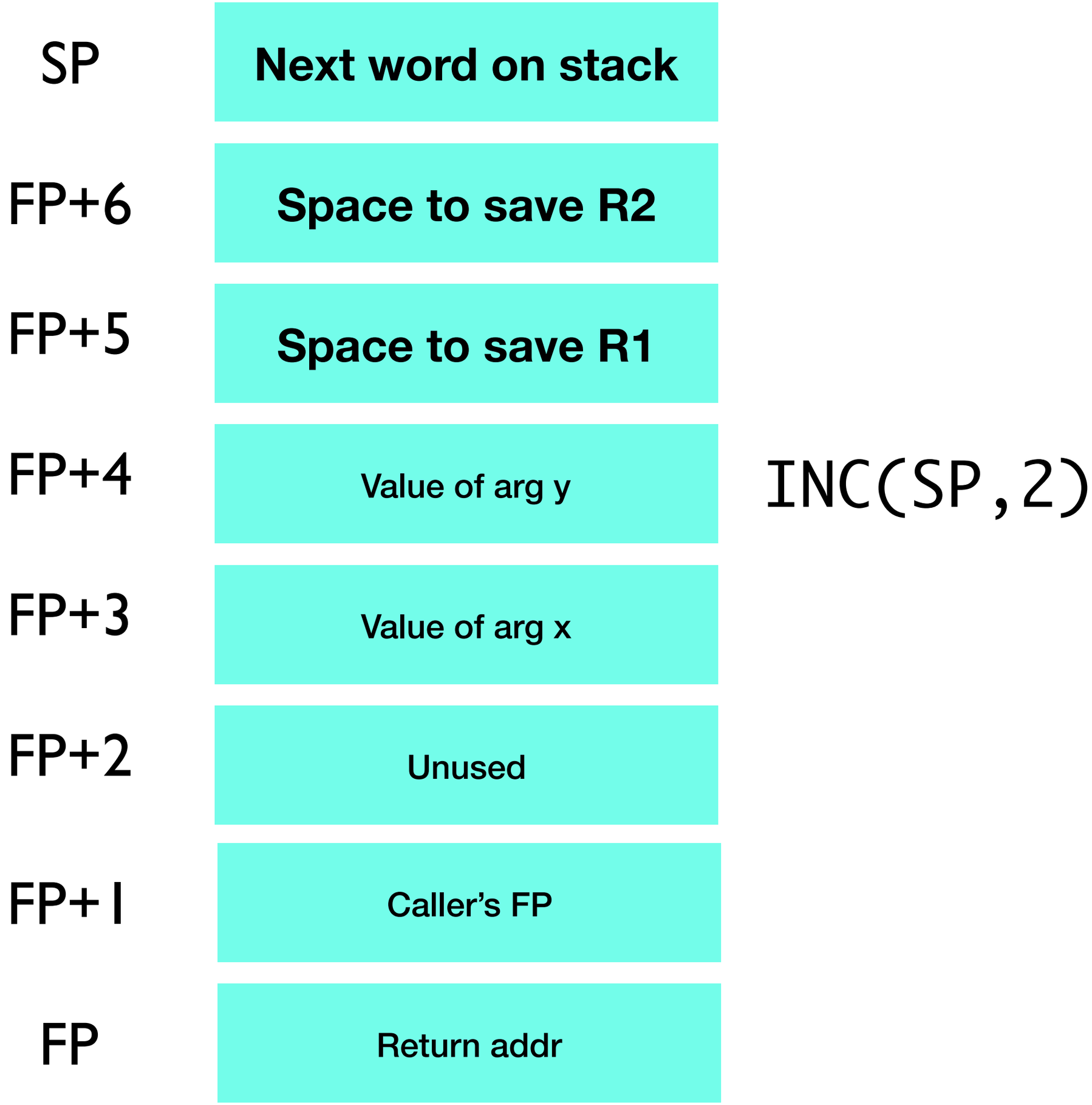
```
LABEL(two_x_plus_y)
// FIRST, make space to save r1 and r2 and then save Rt, R12, and them
    INC(SP, 2)
    STORE(Rt, 0,FP)      // Return address, i.e. PC+1 from before CALL
    STORE(R12, 1,FP)     // Control Link, i.e. FP from before the CALL
    STORE(r1, 5,FP)      // skip FP+3 and FP+4, where a and b will be...
    STORE(r2, 6,FP)      //    ... and save r1 and r2 in FP+5/FP+6

// Load "x" and "y" from stack frame (calling func. put them there)
    LOAD(r1, 3,FP)       // r1 = x
    LOAD(r2, 4,FP)       // r2 = y

// Compute the result
    ADD(r1, r1,r1)       // r1 = x+x
    ADD(r1, r1,r2)       // r1 = x+x+y

// Store the result where the caller of two_x_plus_y will find it
    STORE(r1, 3,FP)

// FINALLY, restore registers (including Rt and R12) and return
    LOAD(r2, 6,FP)       // Restore r2
    LOAD(r1, 5,FP)       // Restore r1
    LOAD(Rt, 0,FP)       // Restore Rt to provide R.A. for return
    LOAD(R12, 1,FP)      // Restore R12 (C.L.) to provide old FP for return
    DEC(SP, 2)
    RETURN(R12, Rt)
```


SP
Next word on stack
FP+4
Value of arg y
FP+3
Value of arg x
FP+2
Unused
FP+1
Caller's FP
FP
Return addr

`two_x_plus_y` wants to use R1 and R2

So make space by incrementing SP




SP
Next word on stack
FP+6
Space to save R2
FP+5
Space to save R1
FP+4
Value of arg y
FP+3
Value of arg x
FP+2
Unused
FP+1
Caller's FP
FP
Return addr
INC(SP,2)

SP	**Next word on stack**
FP+6	**Previous R2**
FP+5	**Previous R1**
FP+4	Value of arg y
FP+3	Value of arg x
FP+2	Unused
FP+1	Caller’s FP
FP	Return addr

```
LABEL(two_x_plus_y)
// FIRST, make space to save r1 and r2 and then save Rt, R12, and them
    INC(SP, 2)
    STORE(Rt, 0,FP)      // Return address, i.e. PC+1 from before CALL
    STORE(R12, 1,FP)     // Control Link, i.e. FP from before the CALL
    STORE(r1, 5,FP)      // skip FP+3 and FP+4, where a and b will be...
    STORE(r2, 6,FP)      //   ... and save r1 and r2 in FP+5/FP+6

// Load "x" and "y" from stack frame (calling func. put them there)
    LOAD(r1, 3,FP)       // r1 = x
    LOAD(r2, 4,FP)       // r2 = y

// Compute the result
    ADD(r1, r1,r1)       // r1 = x+x
    ADD(r1, r1,r2)       // r1 = x+x+y

// Store the result where the caller of two_x_plus_y will find it
    STORE(r1, 3,FP)

// FINALLY, restore registers (including Rt and R12) and return
    LOAD(r2, 6,FP)       // Restore r2
    LOAD(r1, 5,FP)       // Restore r1
    LOAD(Rt, 0,FP)       // Restore Rt to provide R.A. for return
    LOAD(R12, 1,FP)      // Restore R12 (C.L.) to provide old FP for return
    DEC(SP, 2)
    RETURN(R12, Rt)
```

SP

Next word on stack

FP+6

Previous R2

R1

x

FP+5

Previous R1

FP+4

Value of arg y

R2

y

FP+3

Value of arg x

FP+2

Unused

FP+1

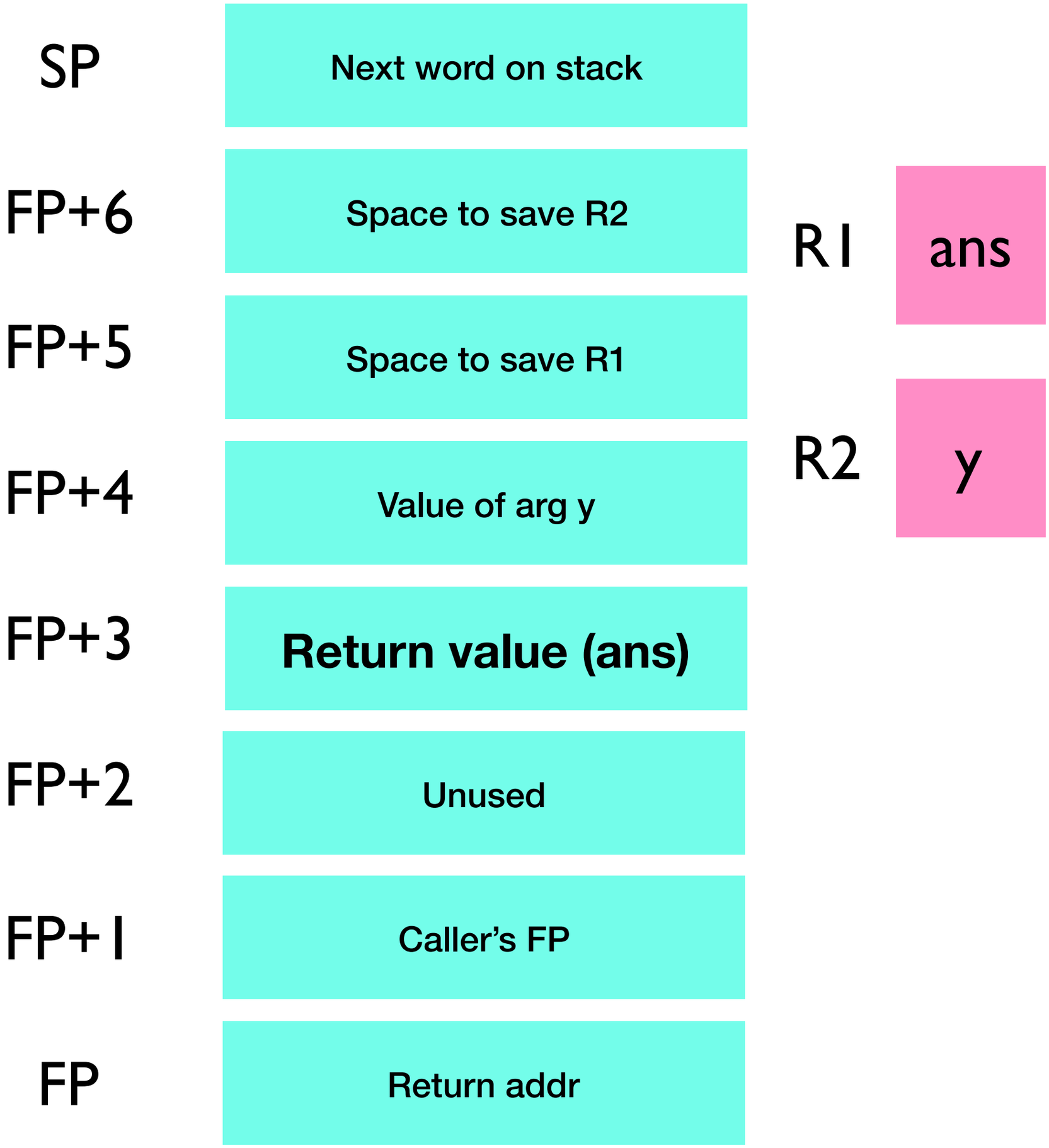
Caller's FP

FP

Return addr

```
LABEL(two_x_plus_y)
// FIRST, make space to save r1 and r2 and then save Rt, R12, and them
    INC(SP, 2)
    STORE(Rt, 0,FP)      // Return address, i.e. PC+1 from before CALL
    STORE(R12, 1,FP)     // Control Link, i.e. FP from before the CALL
    STORE(r1, 5,FP)      // skip FP+3 and FP+4, where a and b will be...
    STORE(r2, 6,FP)      //   ... and save r1 and r2 in FP+5/FP+6

// Load "x" and "y" from stack frame (calling func. put them there)
    LOAD(r1, 3,FP)       // r1 = x
    LOAD(r2, 4,FP)       // r2 = y

// Compute the result
    ADD(r1, r1,r1)       // r1 = x+x
    ADD(r1, r1,r2)       // r1 = x+x+y

// Store the result where the caller of two_x_plus_y will find it
    STORE(r1, 3,FP)

// FINALLY, restore registers (including Rt and R12) and return
    LOAD(r2, 6,FP)       // Restore r2
    LOAD(r1, 5,FP)       // Restore r1
    LOAD(Rt, 0,FP)       // Restore Rt to provide R.A. for return
    LOAD(R12, 1,FP)      // Restore R12 (C.L.) to provide old FP for return
    DEC(SP, 2)
    RETURN(R12, Rt)
```

```
LABEL(two_x_plus_y)
// FIRST, make space to save r1 and r2 and then save Rt, R12, and them
    INC(SP, 2)
    STORE(Rt, 0,FP)      // Return address, i.e. PC+1 from before CALL
    STORE(R12, 1,FP)     // Control Link, i.e. FP from before the CALL
    STORE(r1, 5,FP)      // skip FP+3 and FP+4, where a and b will be...
    STORE(r2, 6,FP)      //   ... and save r1 and r2 in FP+5/FP+6

// Load "x" and "y" from stack frame (calling func. put them there)
    LOAD(r1, 3,FP)       // r1 = x
    LOAD(r2, 4,FP)       // r2 = y

// Compute the result
    ADD(r1, r1,r1)       // r1 = x+x
    ADD(r1, r1,r2)       // r1 = x+x+y

// Store the result where the caller of two_x_plus_y will find it
    STORE(r1, 3,FP)

// FINALLY, restore registers (including Rt and R12) and return
    LOAD(r2, 6,FP)       // Restore r2
    LOAD(r1, 5,FP)       // Restore r1
    LOAD(Rt, 0,FP)       // Restore Rt to provide R.A. for return
    LOAD(R12, 1,FP)      // Restore R12 (C.L.) to provide old FP for return
    DEC(SP, 2)
    RETURN(R12, Rt)
```

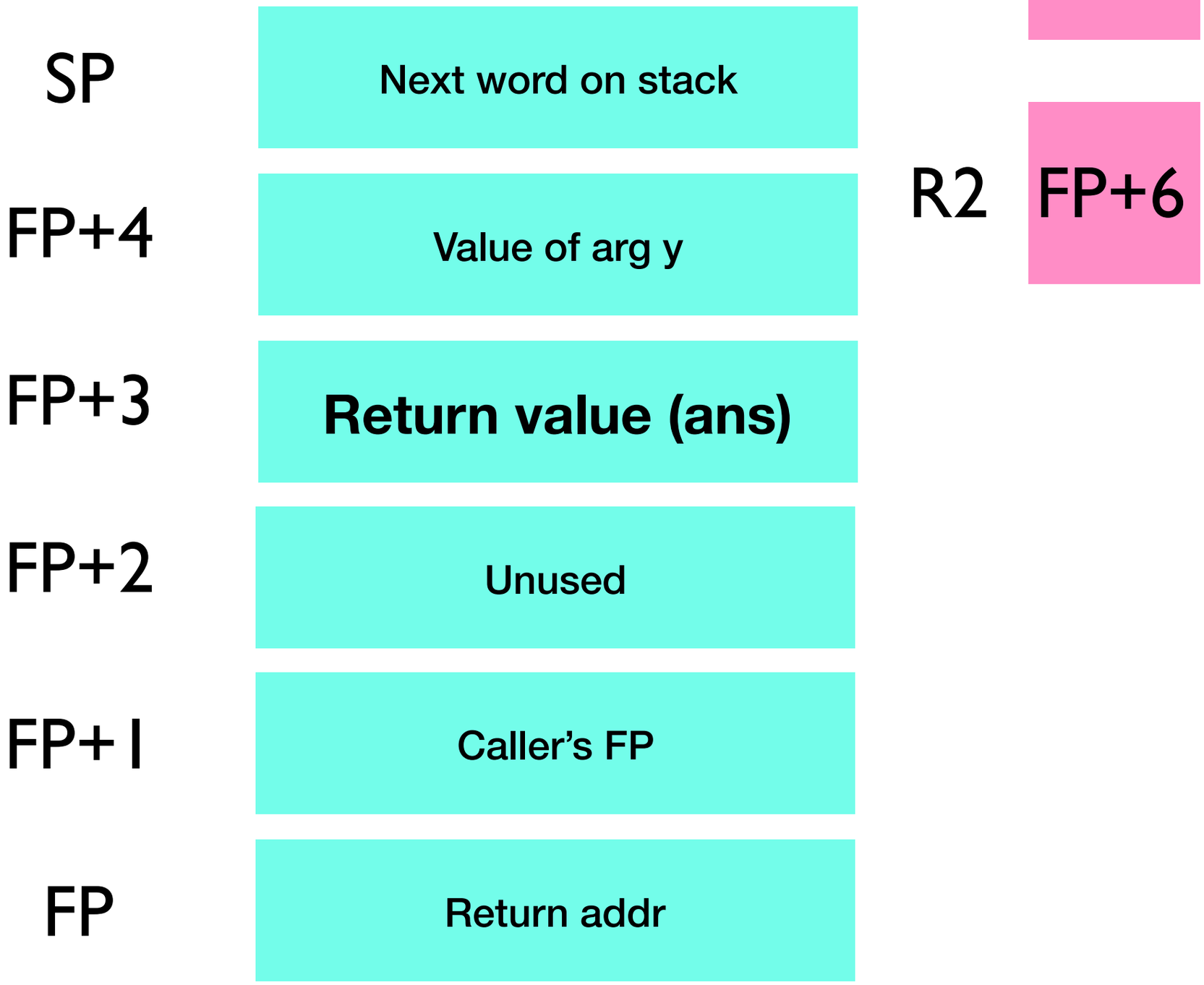

SP
Next word on stack
FP+6
Space to save R2
FP+5
Space to save R1
FP+4
Value of arg y
FP+3
Return value (ans)
FP+2
Unused
FP+1
Caller's FP
FP
Return addr
R1
ans
R2
y

```
LABEL(two_x_plus_y)
// FIRST, make space to save r1 and r2 and then save Rt, R12, and them
    INC(SP, 2)
    STORE(Rt, 0,FP)     // Return address, i.e. PC+1 from before CALL
    STORE(R12, 1,FP)    // Control Link, i.e. FP from before the CALL
    STORE(r1, 5,FP)     // skip FP+3 and FP+4, where a and b will be...
    STORE(r2, 6,FP)     //   ... and save r1 and r2 in FP+5/FP+6

// Load "x" and "y" from stack frame (calling func. put them there)
    LOAD(r1, 3,FP)      // r1 = x
    LOAD(r2, 4,FP)      // r2 = y

// Compute the result
    ADD(r1, r1,r1)      // r1 = x+x
    ADD(r1, r1,r2)      // r1 = x+x+y

// Store the result where the caller of two_x_plus_y will find it
    STORE(r1, 3,FP)

// FINALLY, restore registers (including Rt and R12) and return
    LOAD(r2, 6,FP)      // Restore r2
    LOAD(r1, 5,FP)      // Restore r1
    LOAD(Rt, 0,FP)      // Restore Rt to provide R.A. for return
    LOAD(R12, 1,FP)     // Restore R12 (C.L.) to provide old FP for return
    DEC(SP, 2)
    RETURN(R12, Rt)
```

SP	Next word on stack		
FP+6	Space to save R2	R1	FP+5
FP+5	Space to save R1		
FP+4	Value of arg y	R2	FP+6
FP+3	**Return value (ans)**		
FP+2	Unused		
FP+1	Caller's FP		
FP	Return addr		

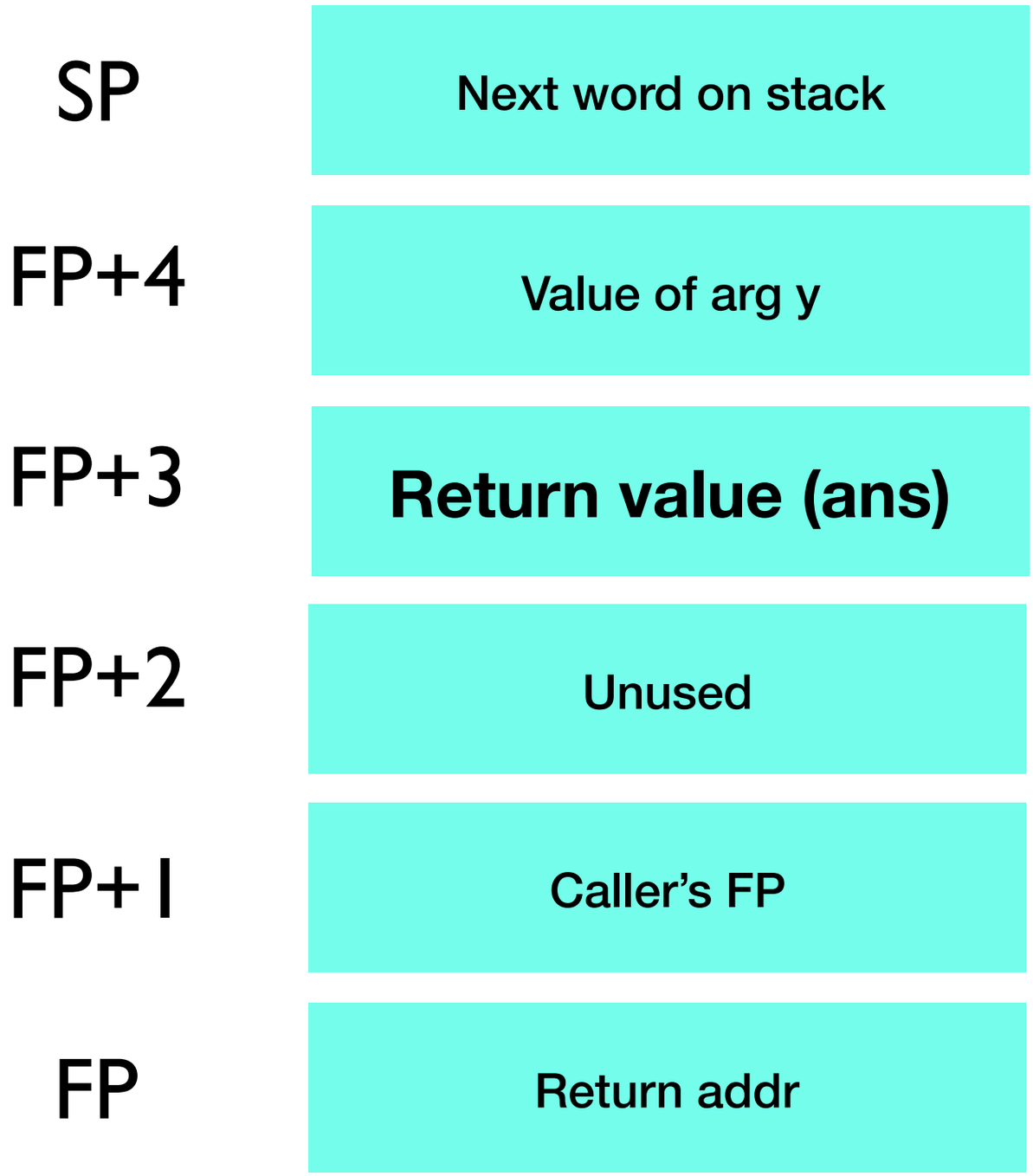
```
LABEL(two_x_plus_y)
// FIRST, make space to save r1 and r2 and then save Rt, R12, and them
    INC(SP, 2)
    STORE(Rt, 0,FP)     // Return address, i.e. PC+1 from before CALL
    STORE(R12, 1,FP)    // Control Link, i.e. FP from before the CALL
    STORE(r1, 5,FP)     // skip FP+3 and FP+4, where a and b will be...
    STORE(r2, 6,FP)     //   ... and save r1 and r2 in FP+5/FP+6

// Load "x" and "y" from stack frame (calling func. put them there)
    LOAD(r1, 3,FP)      // r1 = x
    LOAD(r2, 4,FP)      // r2 = y

// Compute the result
    ADD(r1, r1,r1)      // r1 = x+x
    ADD(r1, r1,r2)      // r1 = x+x+y

// Store the result where the caller of two_x_plus_y will find it
    STORE(r1, 3,FP)

// FINALLY, restore registers (including Rt and R12) and return
    LOAD(r2, 6,FP)      // Restore r2
    LOAD(r1, 5,FP)      // Restore r1
    LOAD(Rt, 0,FP)      // Restore Rt to provide R.A. for return
    LOAD(R12, 1,FP)     // Restore R12 (C.L.) to provide old FP for return
    DEC(SP, 2)
    RETURN(R12, Rt)
```

```
LABEL(two_x_plus_y)
// FIRST, make space to save r1 and r2 and then save Rt, R12, and them
    INC(SP, 2)
    STORE(Rt, 0,FP)     // Return address, i.e. PC+1 from before CALL
    STORE(R12, 1,FP)    // Control Link, i.e. FP from before the CALL
    STORE(r1, 5,FP)     // skip FP+3 and FP+4, where a and b will be...
    STORE(r2, 6,FP)     //   ... and save r1 and r2 in FP+5/FP+6

// Load "x" and "y" from stack frame (calling func. put them there)
    LOAD(r1, 3,FP)      // r1 = x
    LOAD(r2, 4,FP)      // r2 = y

// Compute the result
    ADD(r1, r1,r1)      // r1 = x+x
    ADD(r1, r1,r2)      // r1 = x+x+y

// Store the result where the caller of two_x_plus_y will find it
    STORE(r1, 3,FP)

// FINALLY, restore registers (including Rt and R12) and return
    LOAD(r2, 6,FP)      // Restore r2
    LOAD(r1, 5,FP)      // Restore r1
    LOAD(Rt, 0,FP)      // Restore Rt to provide R.A. for return
    LOAD(R12, 1,FP)     // Restore R12 (C.L.) to provide old FP for return
    DEC(SP, 2)
    RETURN(R12, Rt)
```


R1
FP+5
SP
Next word on stack
R2
FP+6
FP+4
Value of arg y
FP+3
Return value (ans)
FP+2
Unused
FP+1
Caller's FP
FP
Return addr

```
LABEL(two_x_plus_y)
// FIRST, make space to save r1 and r2 and then save Rt, R12, and them
    INC(SP, 2)
    STORE(Rt, 0,FP)     // Return address, i.e. PC+1 from before CALL
    STORE(R12, 1,FP)    // Control Link, i.e. FP from before the CALL
    STORE(r1, 5,FP)     // skip FP+3 and FP+4, where a and b will be...
    STORE(r2, 6,FP)     //   ... and save r1 and r2 in FP+5/FP+6

// Load "x" and "y" from stack frame (calling func. put them there)
    LOAD(r1, 3,FP)      // r1 = x
    LOAD(r2, 4,FP)      // r2 = y

// Compute the result
    ADD(r1, r1,r1)      // r1 = x+x
    ADD(r1, r1,r2)      // r1 = x+x+y

// Store the result where the caller of two_x_plus_y will find it
    STORE(r1, 3,FP)

// FINALLY, restore registers (including Rt and R12) and return
    LOAD(r2, 6,FP)      // Restore r2
    LOAD(r1, 5,FP)      // Restore r1
    LOAD(Rt, 0,FP)      // Restore Rt to provide R.A. for return
    LOAD(R12, 1,FP)     // Restore R12 (C.L.) to provide old FP for return
    DEC(SP, 2)
    RETURN(R12, Rt)
```

Now caller clears these by decrementing SP



So which one should *you* use?

You **could** mix and match

Much better to pick one and stick to it

One big reason: if other code wants to interact with your code, you need a common calling convention

The **ABI** defines your calling convention

Application Binary Interface

If functions use the same ABI, they can work even if they were written in **different languages**

For example, mixing C and C++

Or call assembly from C++

(This is useful for high-performance apps)