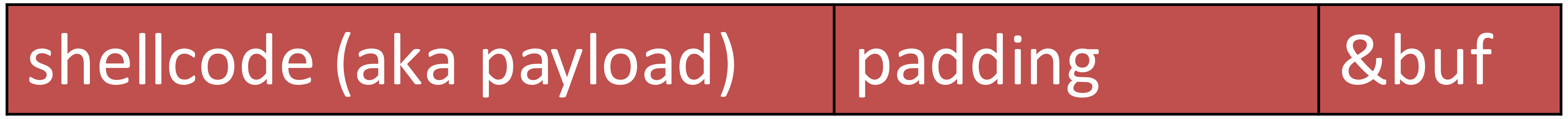
Lecture 12 – Code reuse attacks

Stephen Checkoway
Oberlin College

Last time

- No good reason for stack/heap/static data to be executable
- No good reason for code to be writable
  - An exception to this would be a JIT
- Data Execution Prevention (DEP) or W ^ X gives us exactly that
  - A page of memory can be writable
  - A page of memory can be executable
  - No page can ever be both
  - (Pages can be neither writable nor executable, of course)

Think like an attacker



shellcode (aka payload) | padding | &buf

computation + ***control***

- We (as attackers) are now prevented from executing any injected code
- We still want to perform our computation
- We talked about how to bypass stack canaries last time, so let's ignore them for now and focus on bypassing DEP
- If we can't execute injected code, what code should we execute?

Existing code in binaries

- Program code itself
- Dynamic libraries
  - Chromium 145.0.7632.116 links to 112 dynamic libraries on Debian!
  - libc is linked into (almost) every program
- libc contains useful functions
  - `system` — Run a shell command
  - `mprotect` — Change the memory protection on a region of code

Return to libc (ret2libc)

- Rather than returning to our shellcode, let’s return to a standard library function like `system`
- We need to set the stack up precisely how `system` expects

```
int system(const char *command);
```

Again, we'll start with x86

For 32-bit x86, arguments are passed on the stack

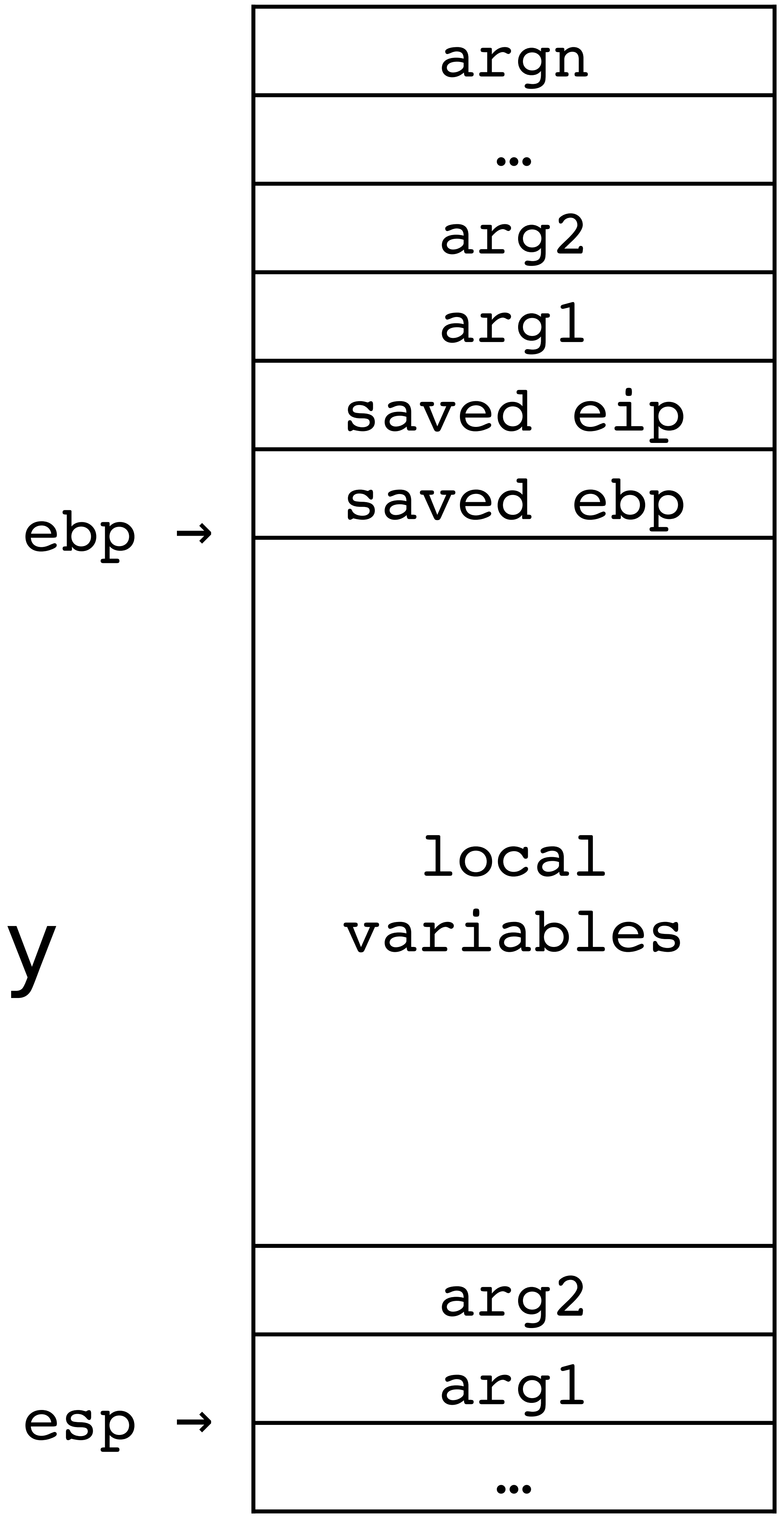
Compilers for x86 used to like to use the frame pointer `ebp`

- This gives consistent offsets from `ebp` to the arguments and local variables, regardless of `esp`:

```
mov eax, [ebp + 8]  # arg1
mov ebx, [ebp + 12] # arg2
mov ecx, [ebp - 4]  # local variable
lea edx, [ebp - 20] # address of local array
```

- Thus the following slides will show the frame pointer

Implication: smashing the stack enables us to put the address of a function in the return address *and* the arguments to the function

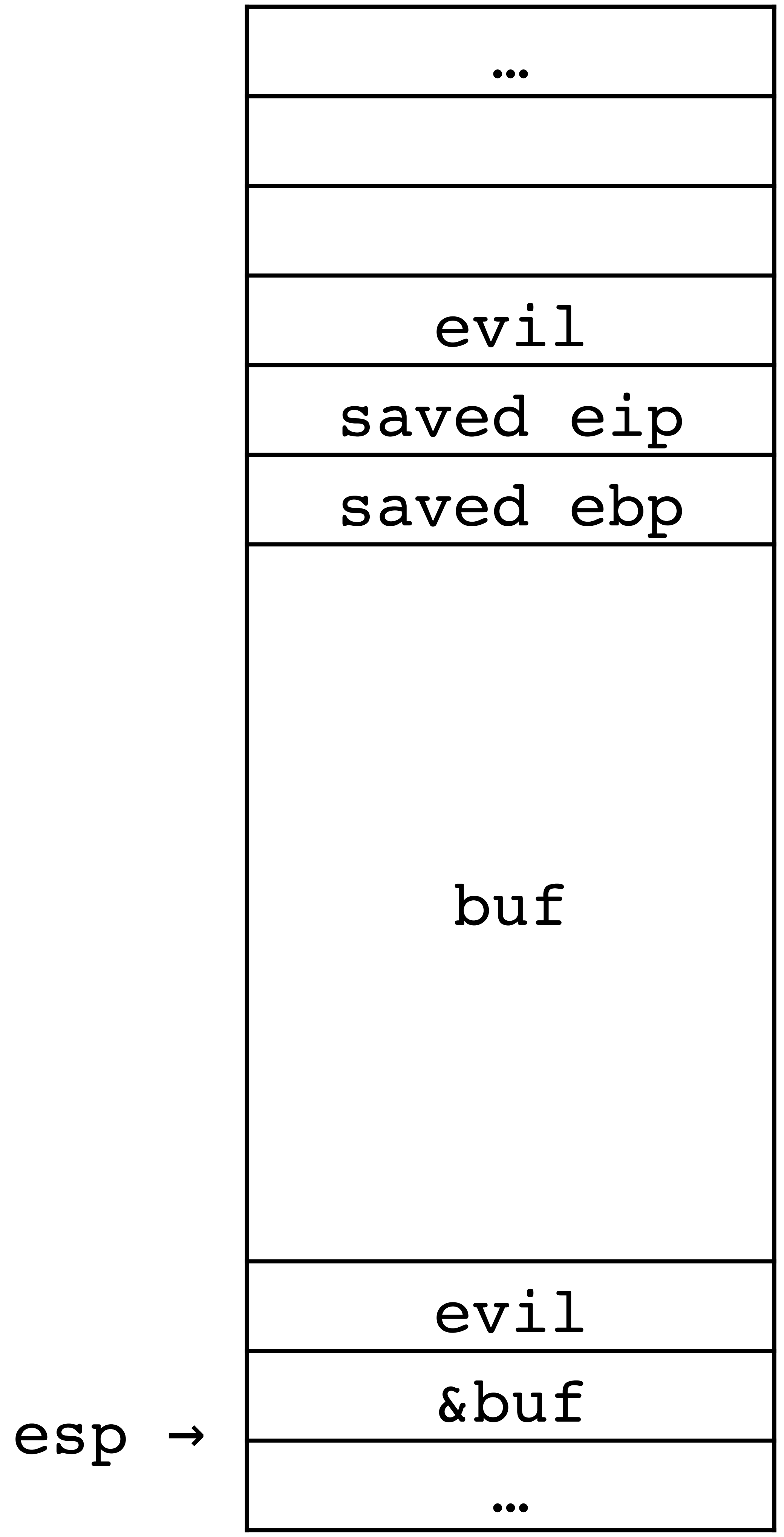


Simple example

- Consider

```
void foo(char *evil) {
    char buf[32];
    strcpy(buf, evil);
}
```

- Let’s overwrite the saved eip with the address of `system`



…

evil

saved eip

saved ebp

buf

evil

esp → &buf

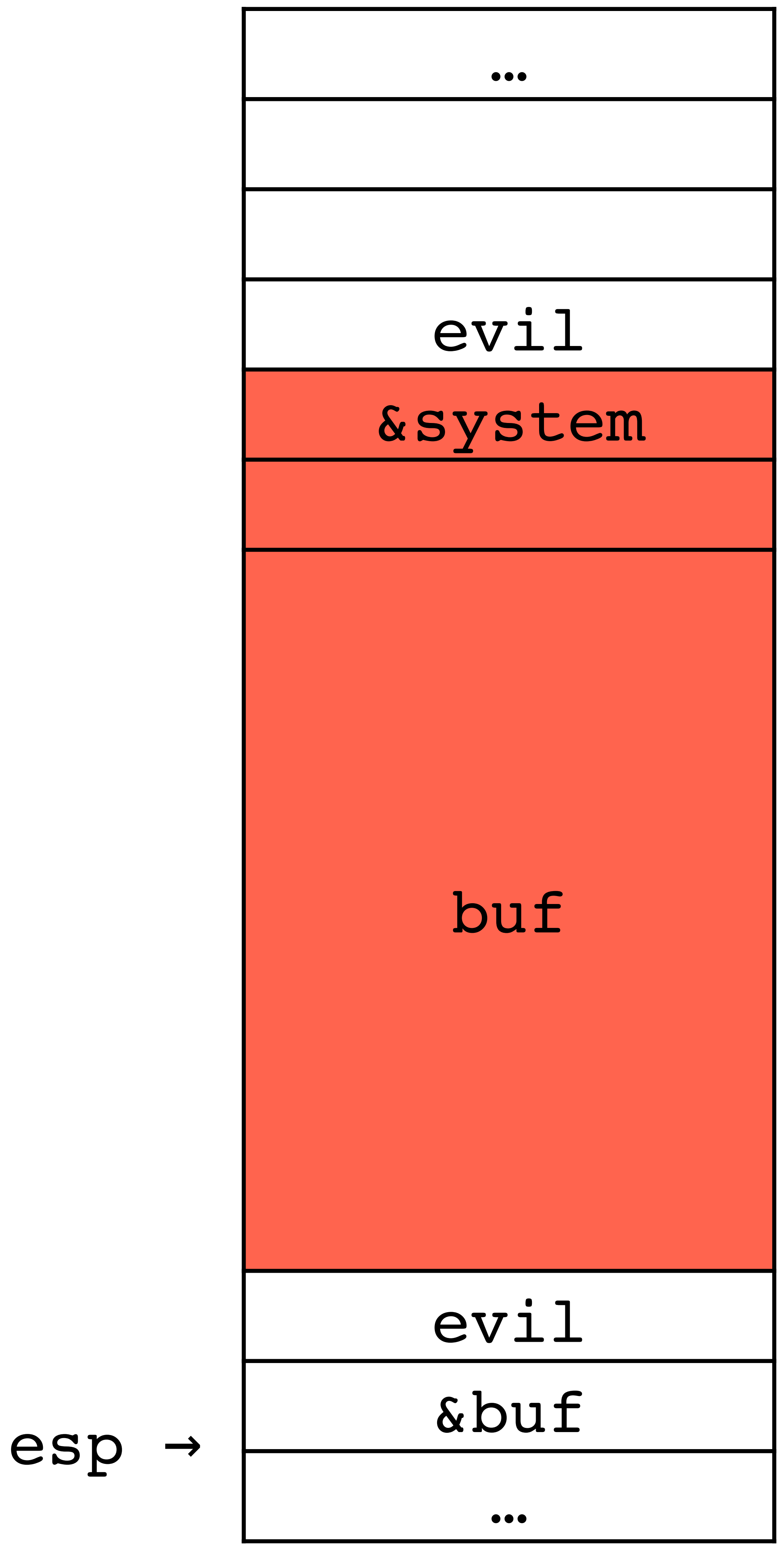
…

Simple example

- Consider

```
void foo(char *evil) {
    char buf[32];
    strcpy(buf, evil);
}
```

- Let's overwrite the saved eip with the address of `system`

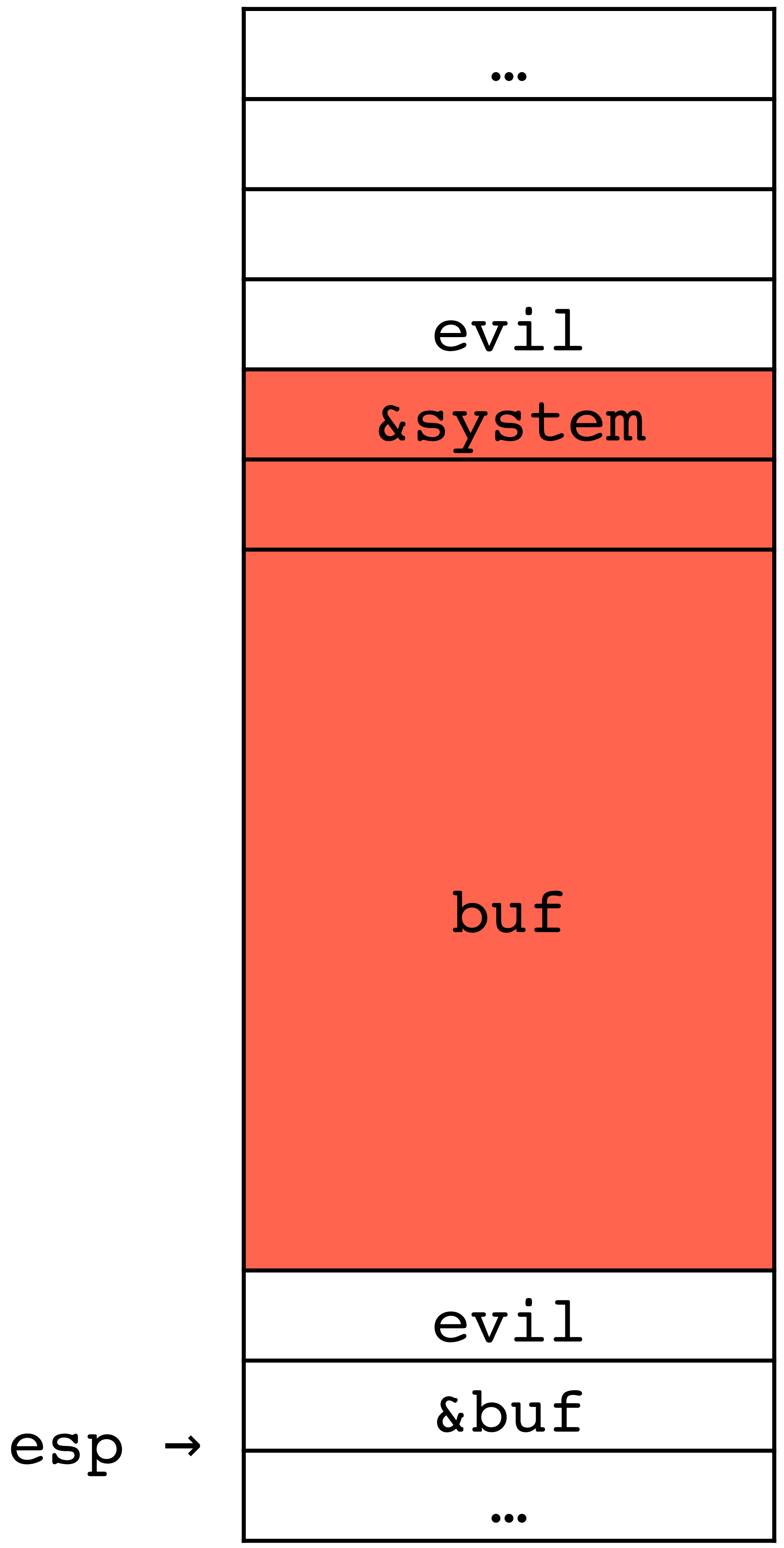


Simple example

- Consider

```
void foo(char *evil) {
    char buf[32];
    strcpy(buf, evil);
}
```

- Let’s overwrite the saved eip with the address of `system`
- `system` takes one argument, a pointer to the command string; where does it go?



…

evil

&system

buf

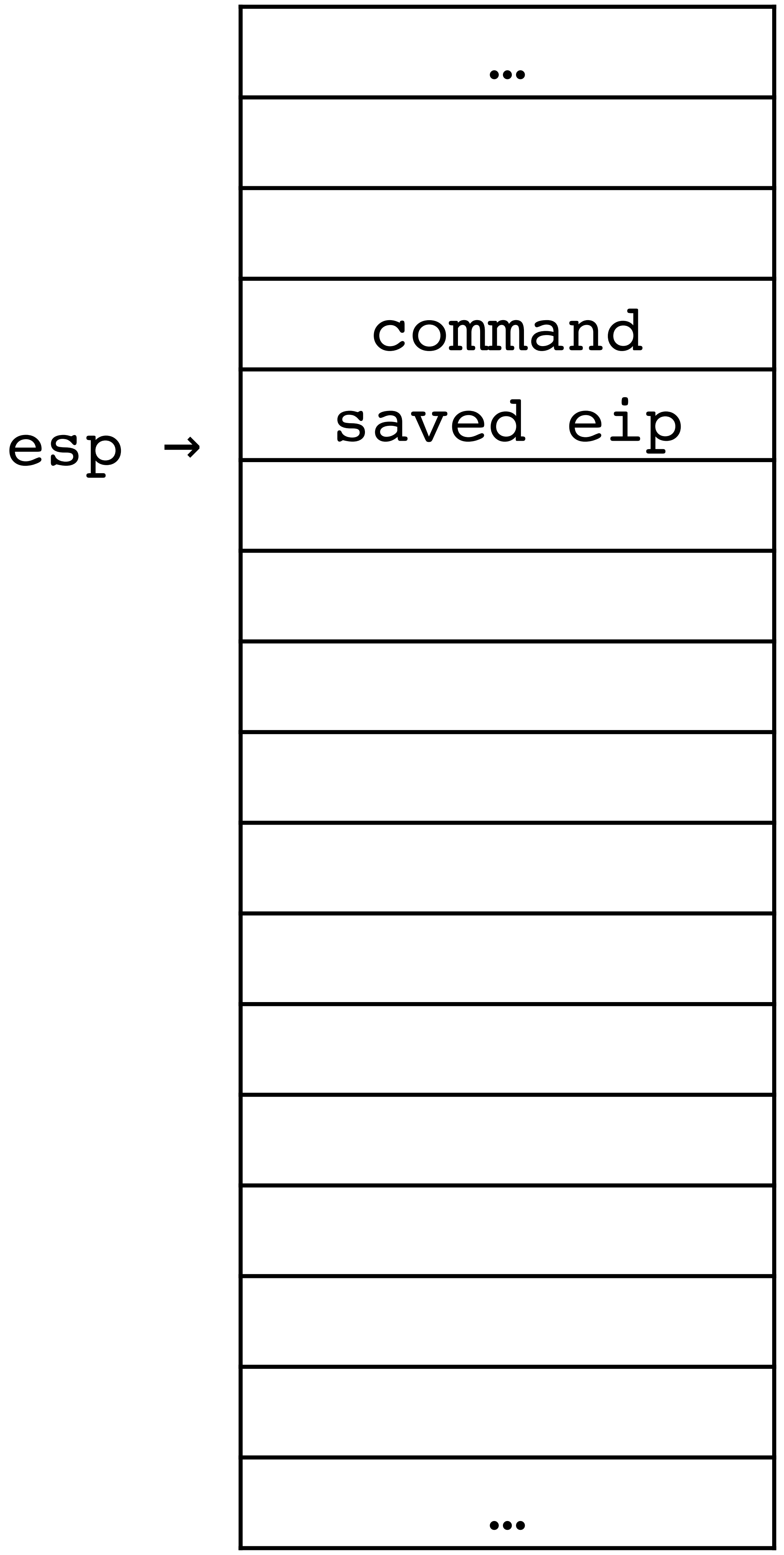
evil

esp → &buf

…

Back to basics

- Imagine we called `system` directly via `system(command);`
- Look at the stack layout before the first instruction in `system`
- As usual, the first argument is at `esp + 4`



…

command

esp → saved eip

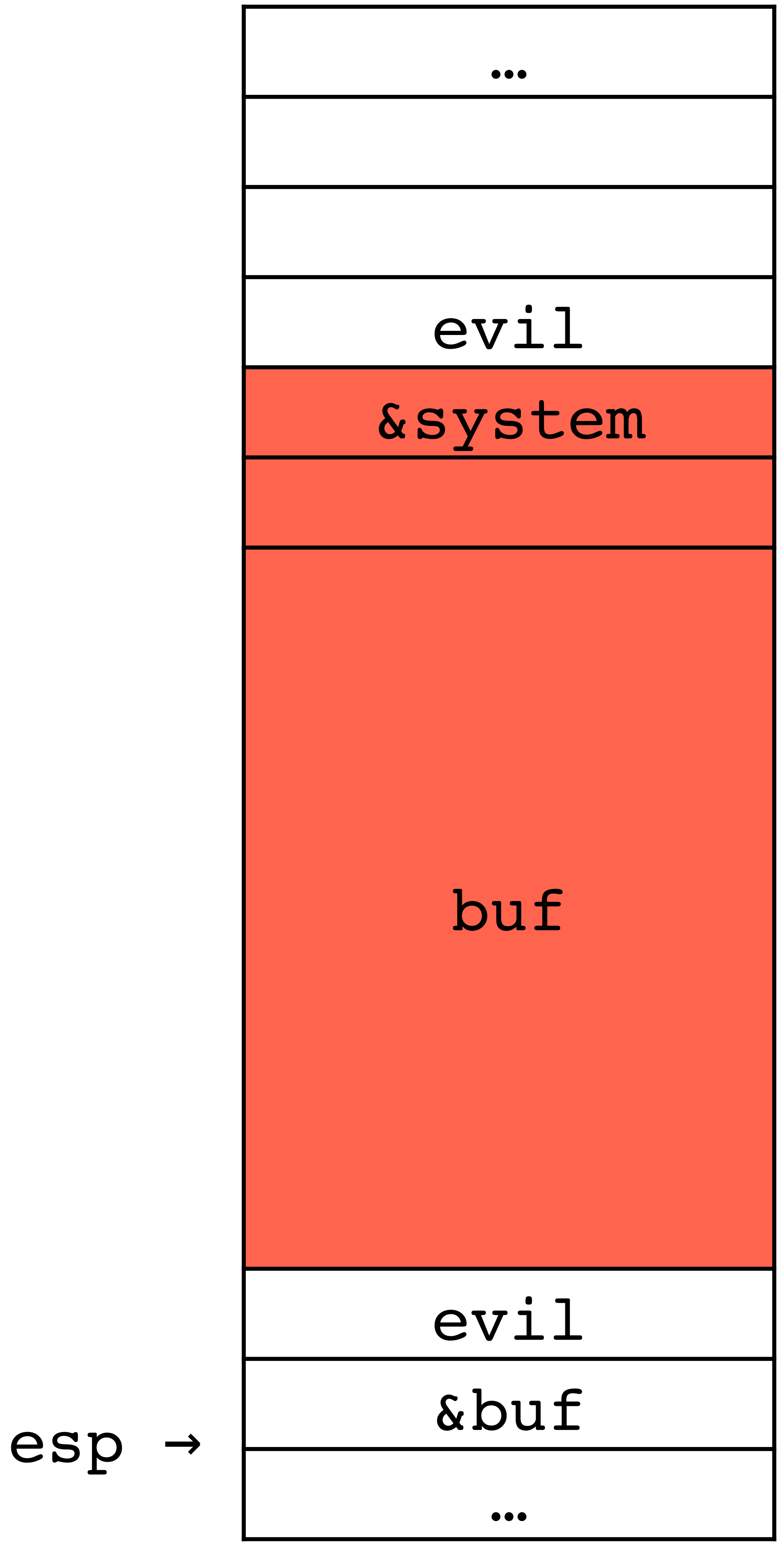
…

Simple example

- Consider

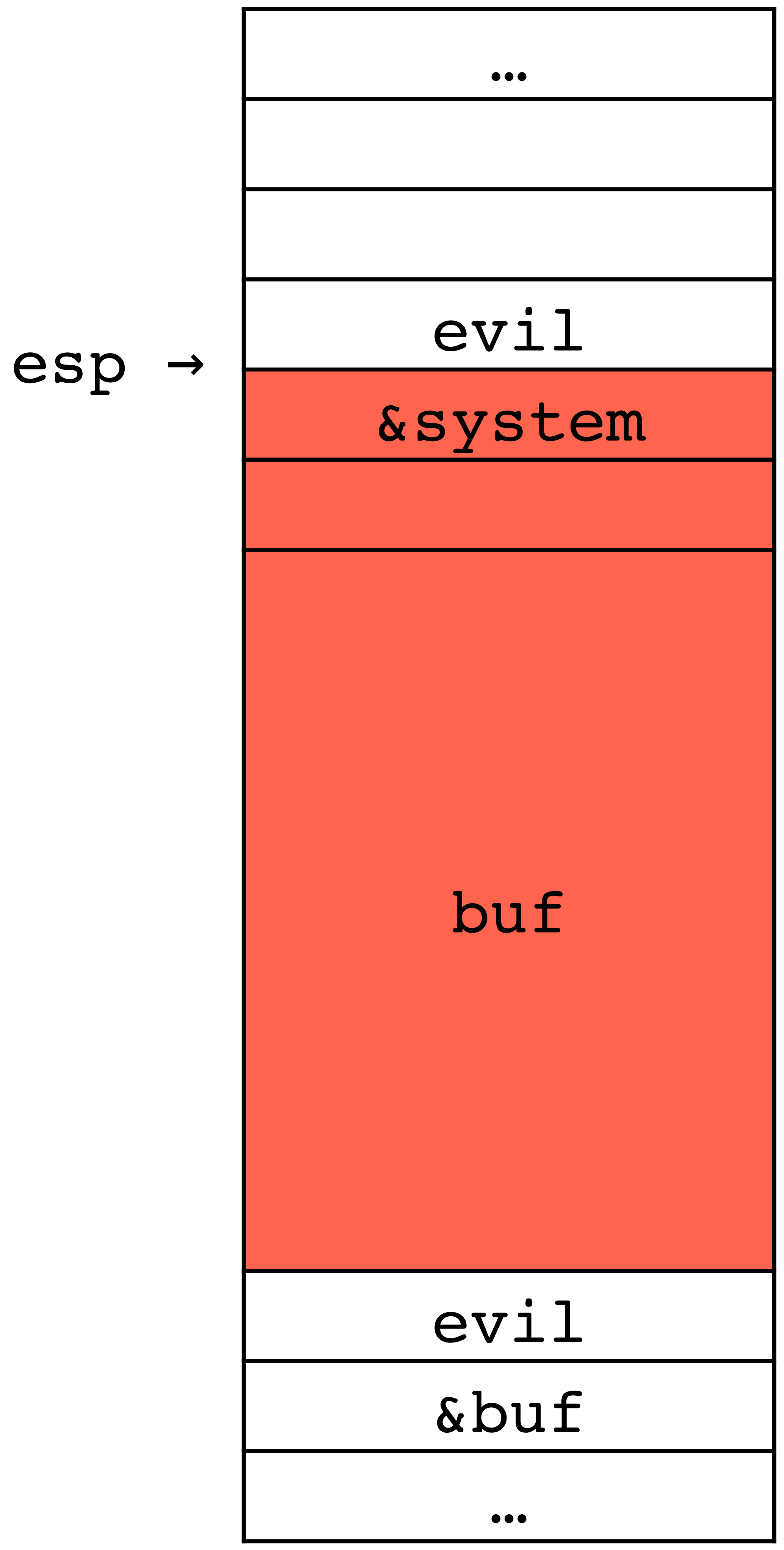
```
void foo(char *evil) {
    char buf[32];
    strcpy(buf, evil);
}
```

- Let's overwrite the saved eip with the address of `system`
- `system` takes one argument, a pointer to the command string; where does it go? `esp + 4` *after the* `ret`



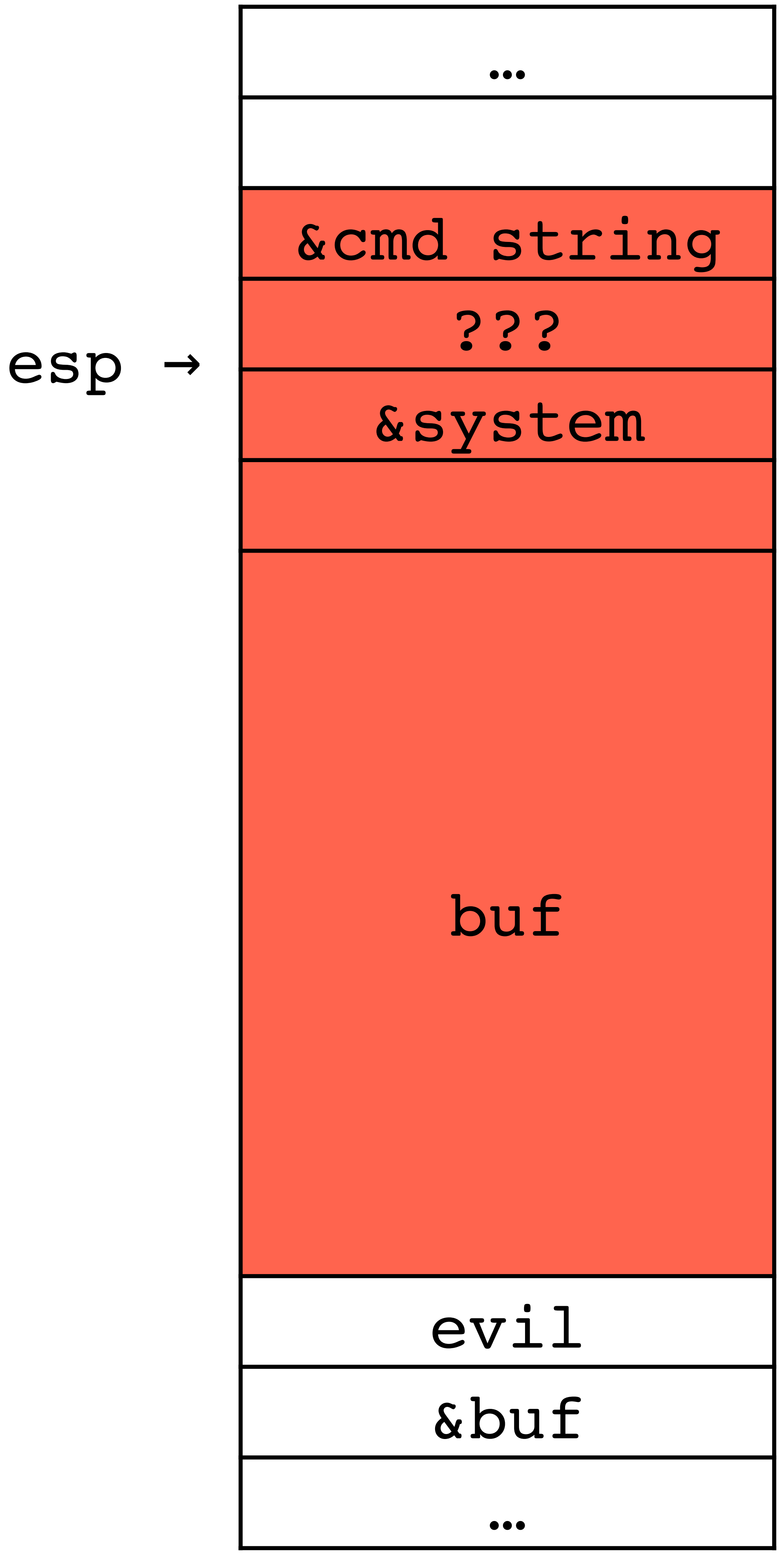
Simple example

- `ret` pops the address of `system` off the stack and into `eip` leaving the stack pointer pointing at the first `evil`
- 4 bytes above that should be our pointer to the command string



Simple example

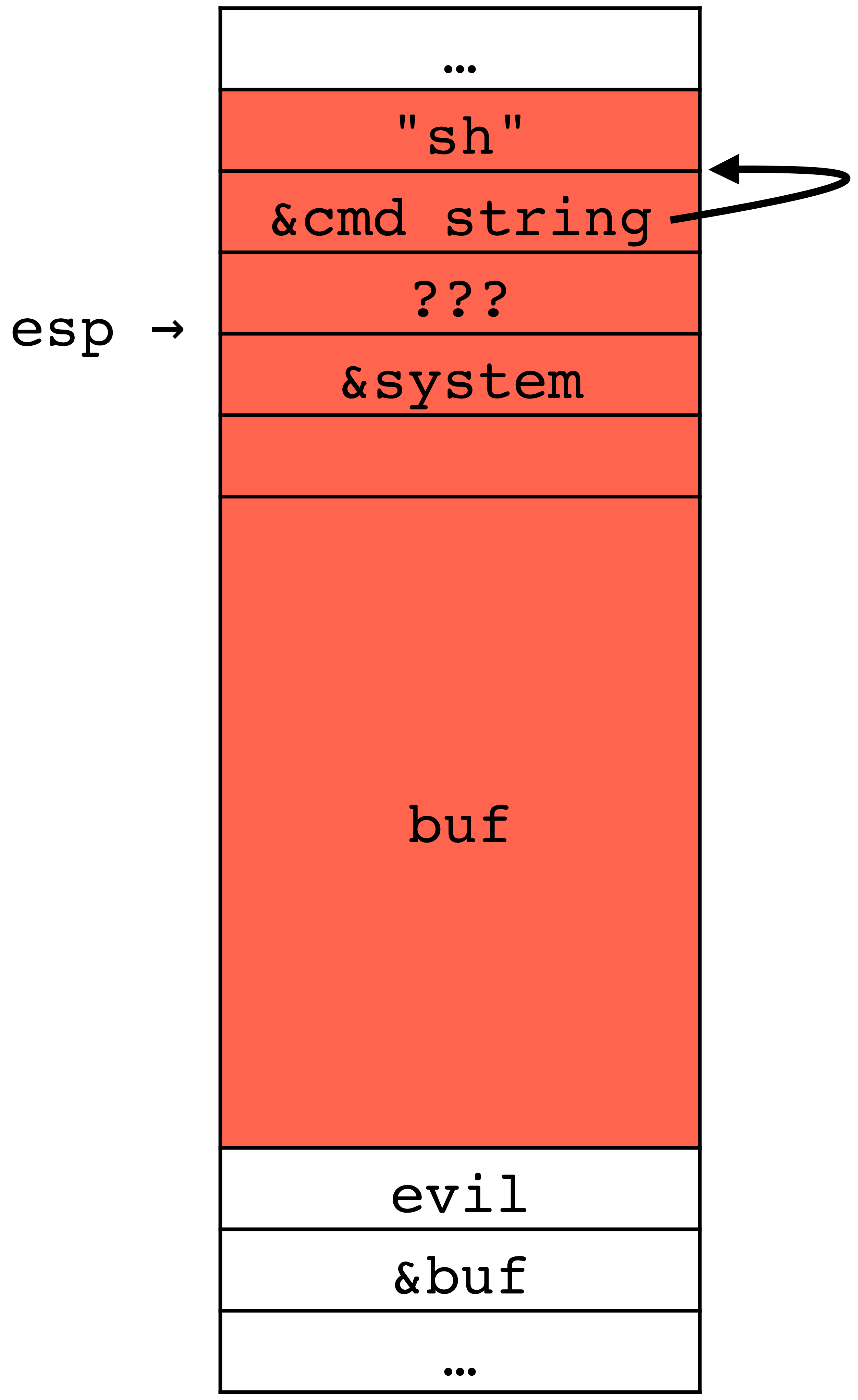
- `ret` pops the address of `system` off the stack and into `eip` leaving the stack pointer pointing at the first `evil`
- 4 bytes above that should be our pointer to the command string
- Where should we put the command string `"sh"` itself?
  - In buf?
  - Above the pointer to the command string?



Simple example

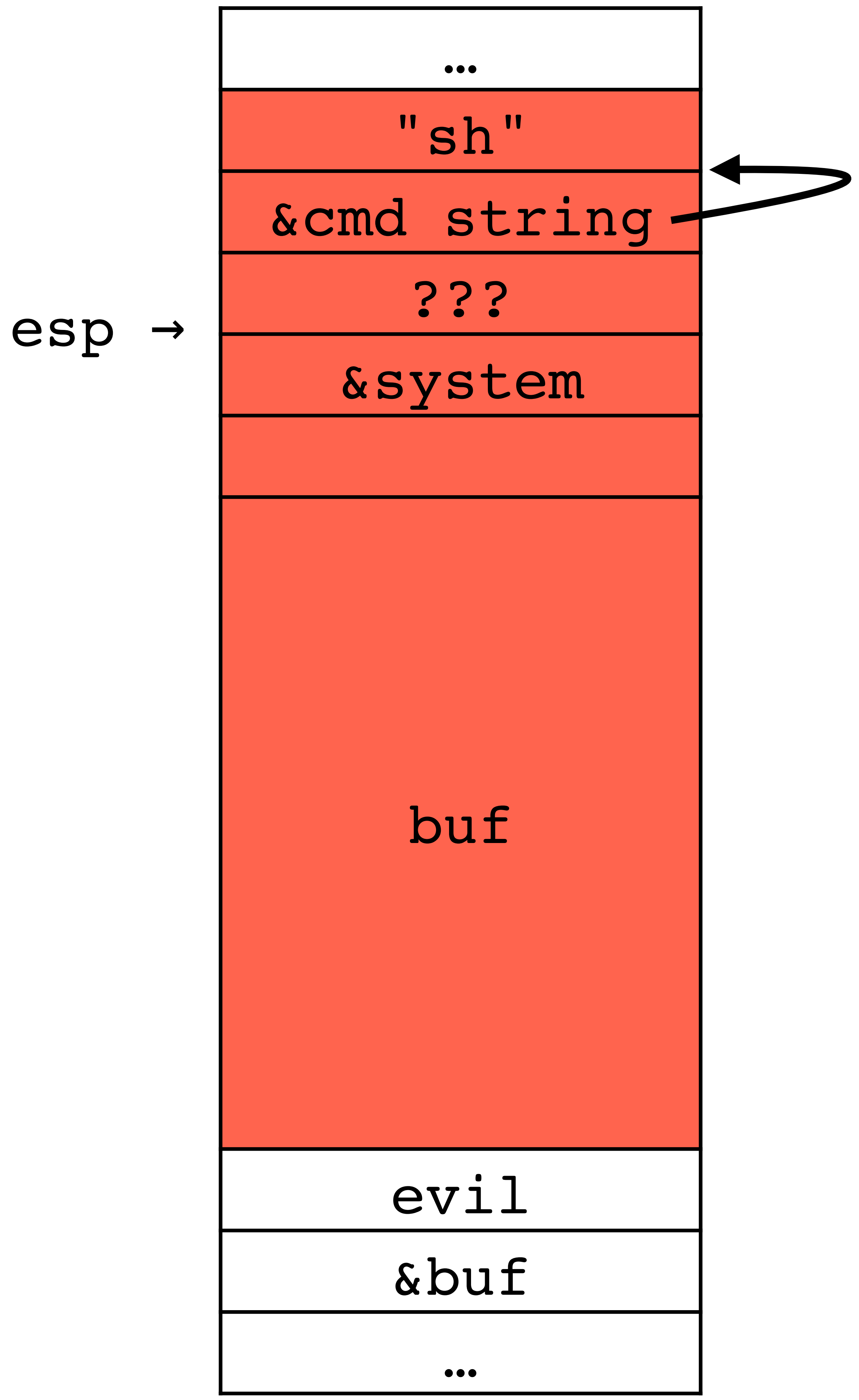
- `ret` pops the address of `system` off the stack and into `eip` leaving the stack pointer pointing at the first `evil`
- 4 bytes above that should be our pointer to the command string
- Where should we put the command string `"sh"` itself?
  - ~~In buf?~~
  - Above the pointer to the command string?

[ret2libc DEMO]



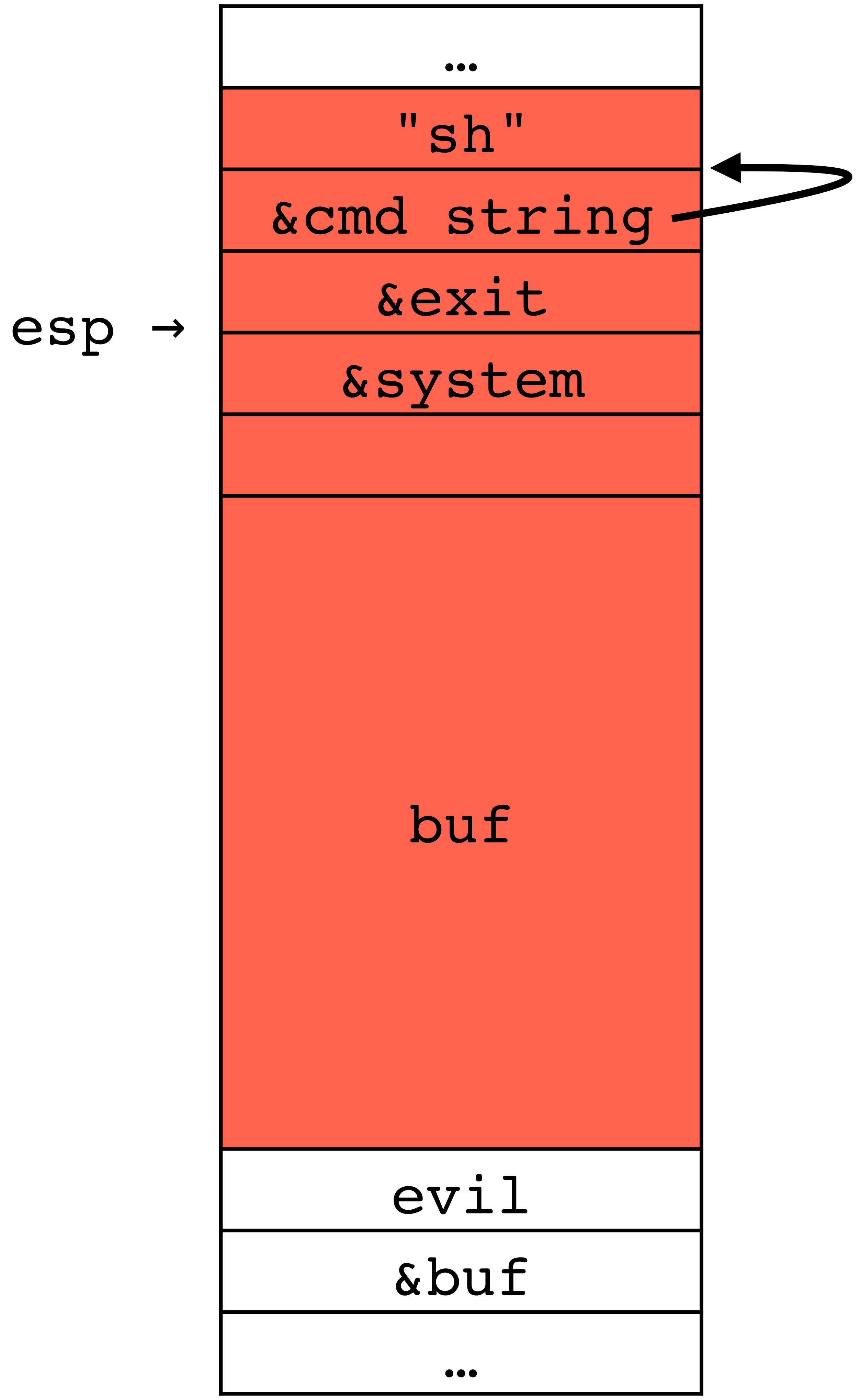
Simple example

- When `system` returns, it'll return to the address on the stack at `esp` (the ???)
- This will likely crash unless we pick a good value to put there



Simple example

- When `system` returns, it'll return to the address on the stack at `esp` (the ???)
- This will likely crash unless we pick a good value to put there
- The address of `exit` is a good choice
- Now when `system` returns, the program will exit without crashing

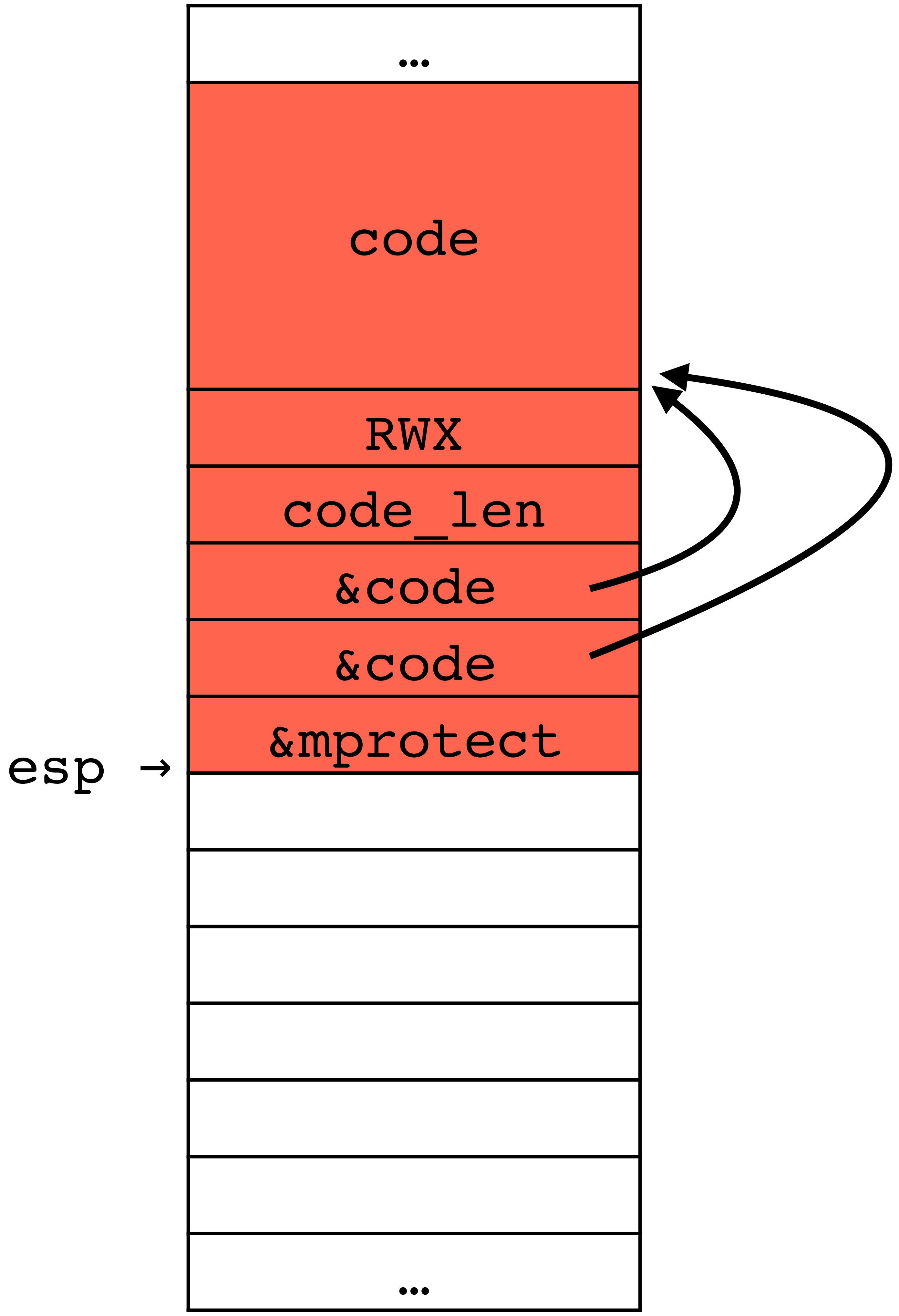


Injecting code

- We cannot run injected code directly, but we can first make it executable by calling `mprotect`

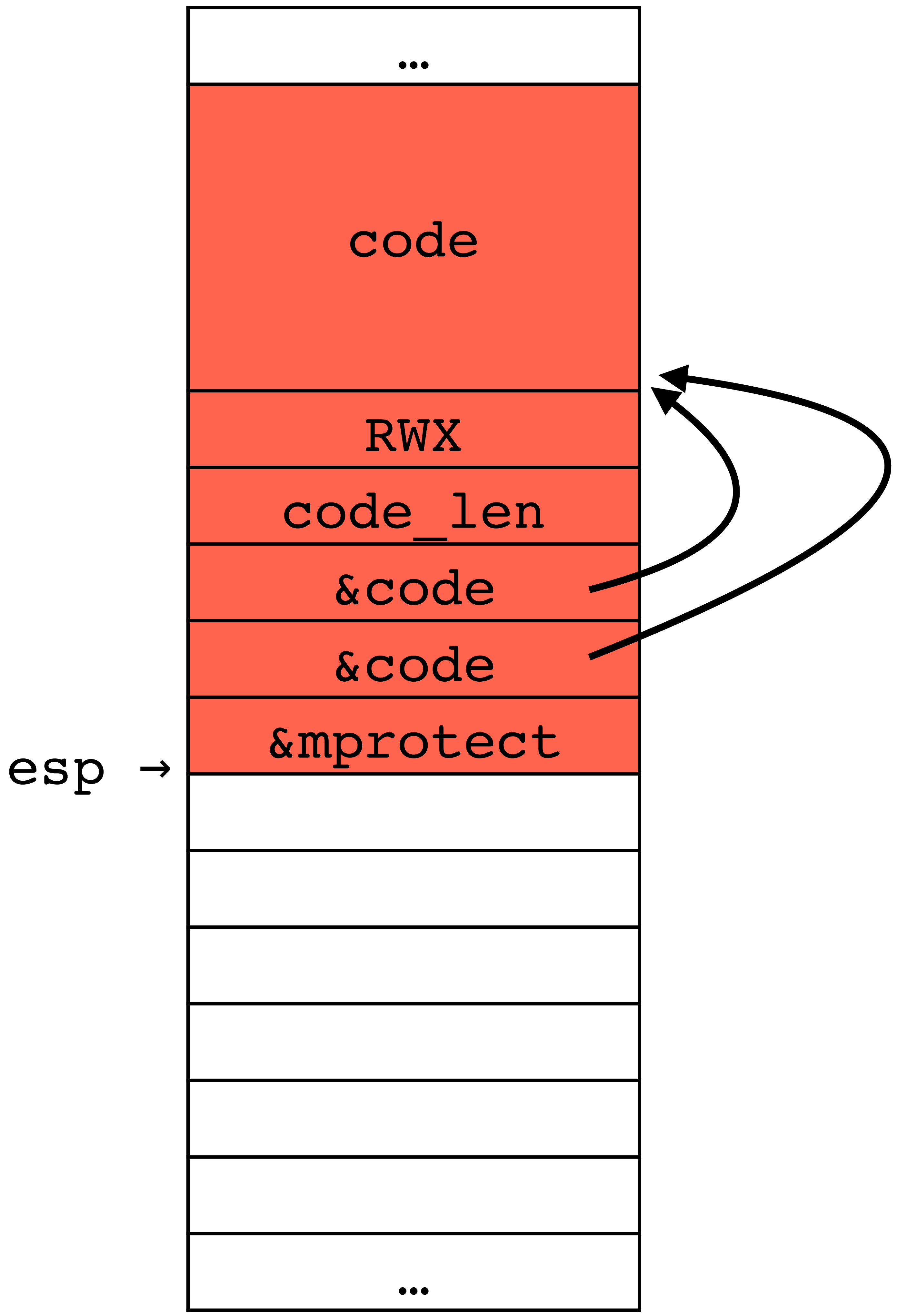
```
int mprotect(void *addr,
             size_t len,
             int prot);
```

- This can be tricky since there are likely to be zero bytes
  - Use memcpy instead of strcpy
  - Use return-oriented programming (next class)



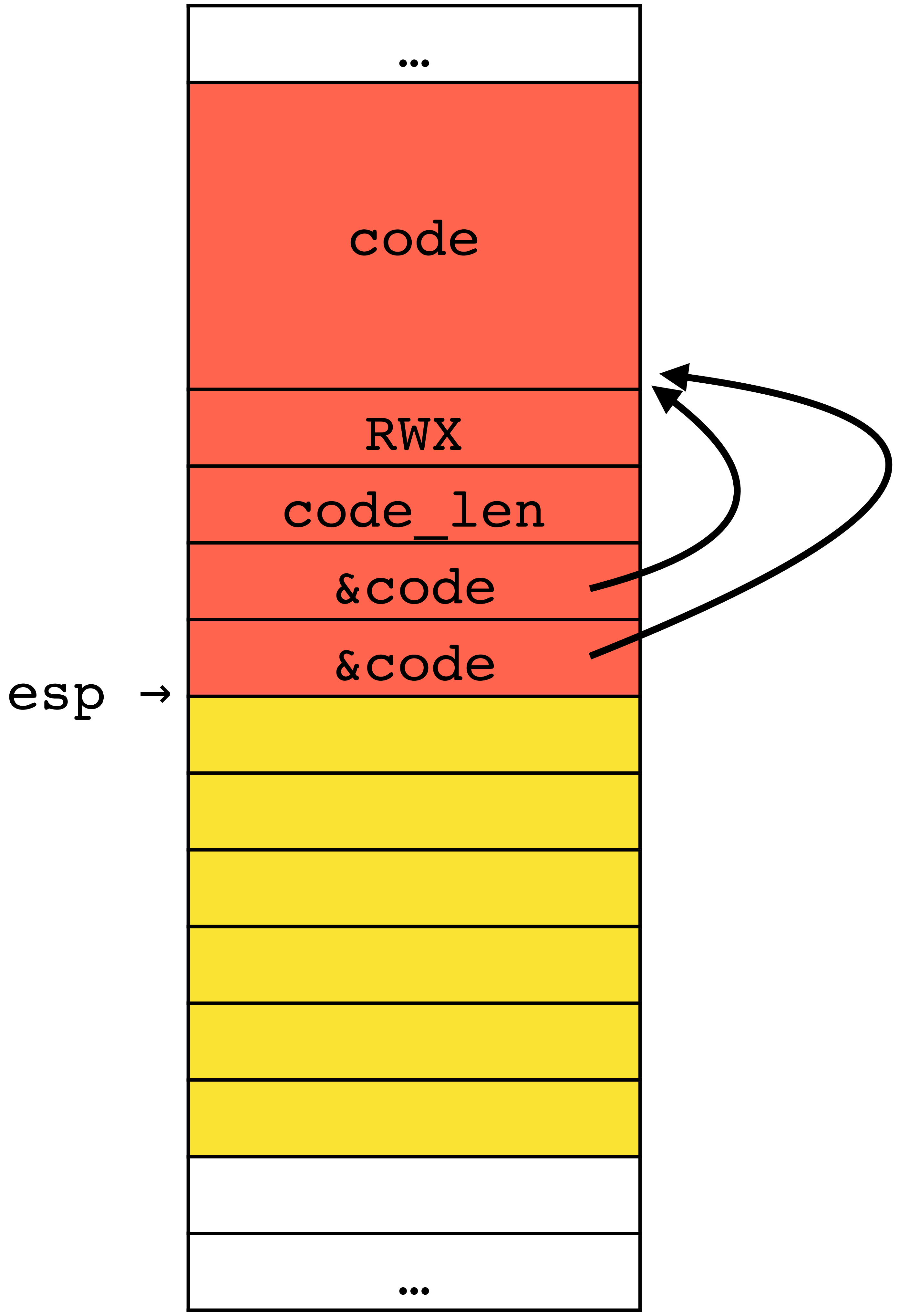
Injecting code

- Return to `mprotect`



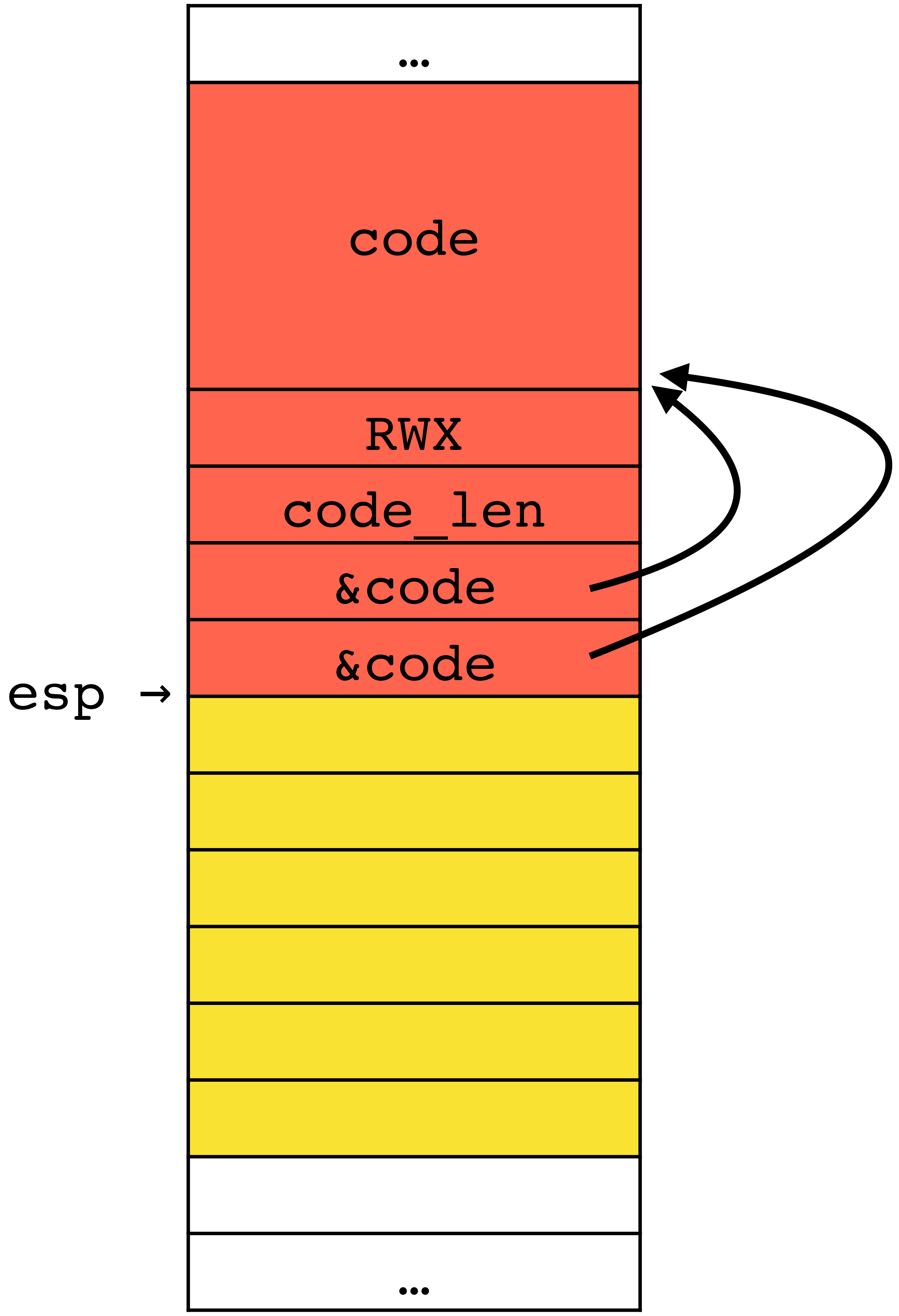
Injecting code

- Return to `mprotect`
  - Increments `esp` by 4
  - Runs `mprotect` making the injected code executable
  - Modifies the stack below `esp`



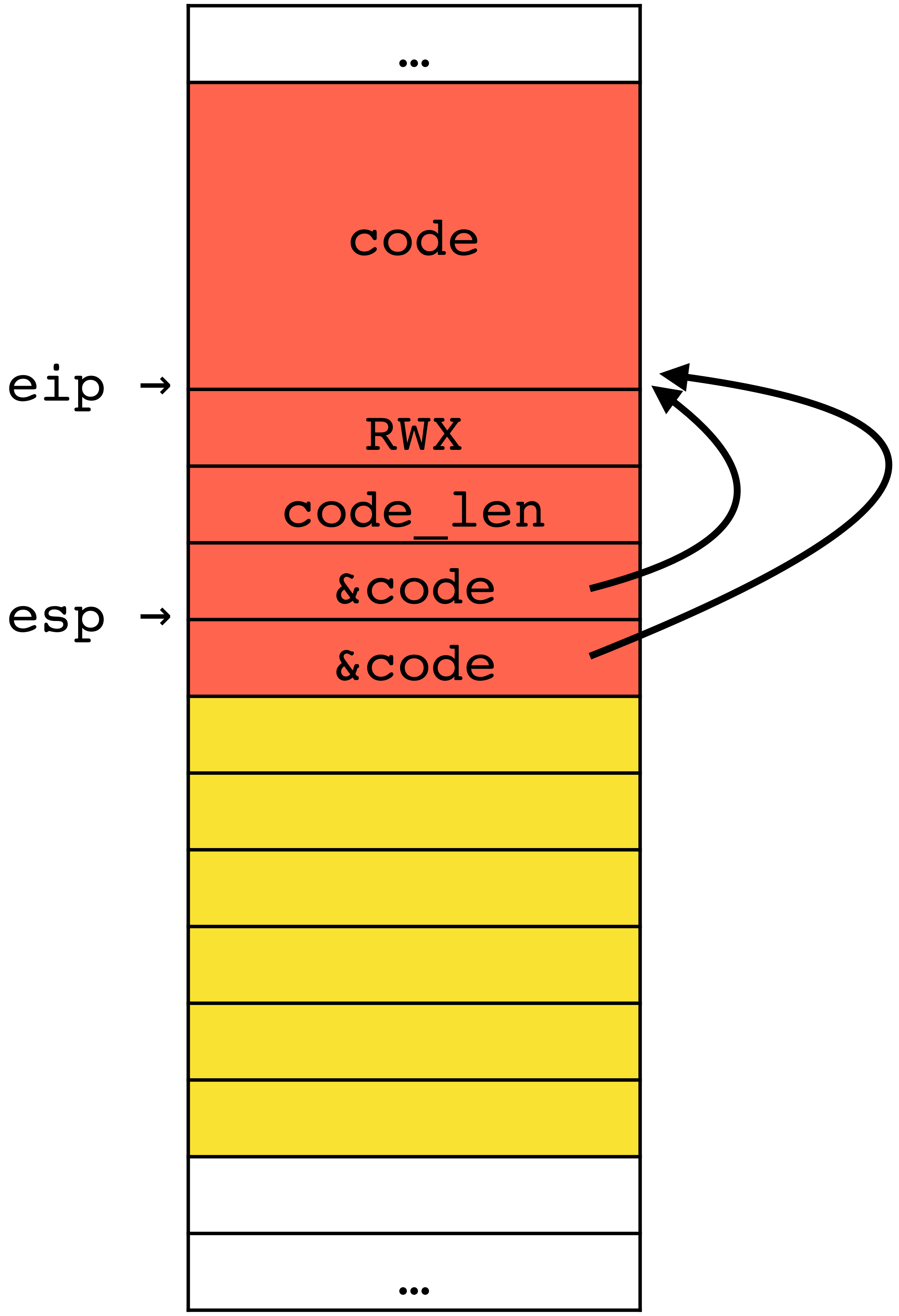
Injecting code

- Return to `mprotect`
  - Increments `esp` by 4
  - Runs `mprotect` making the injected code executable
  - Modifies the stack below `esp`
- Return from `mprotect` to `code`



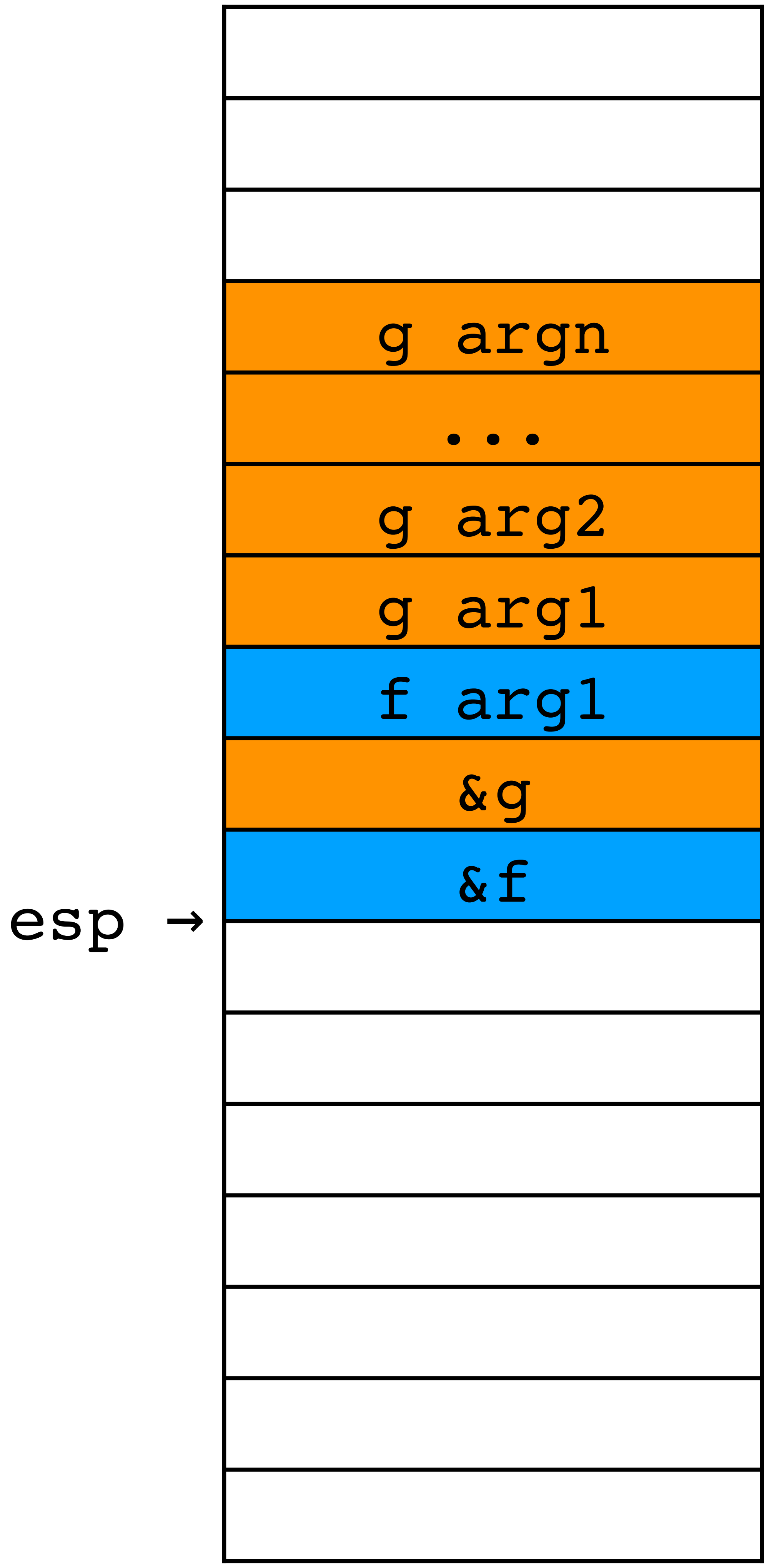
Injecting code

- Return to `mprotect`
  - Increments `esp` by 4
  - Runs `mprotect` making the injected code executable
  - Modifies the stack below `esp`
- Return from `mprotect` to `code`
  - Increments `esp` by 4
  - Runs `code`



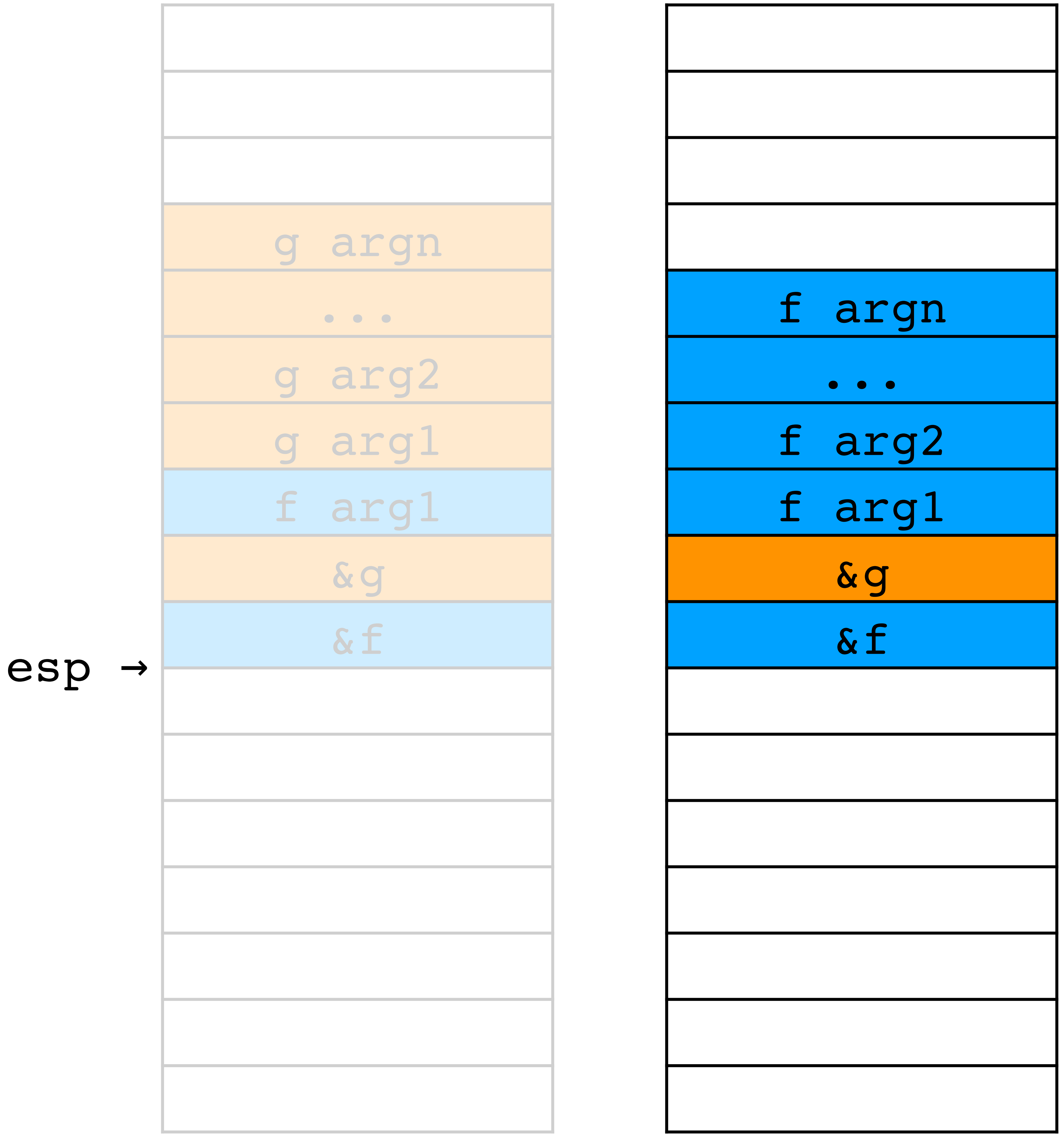
Chaining functions

- We can chain two functions together if
  - the first has one argument and the second any number of arguments



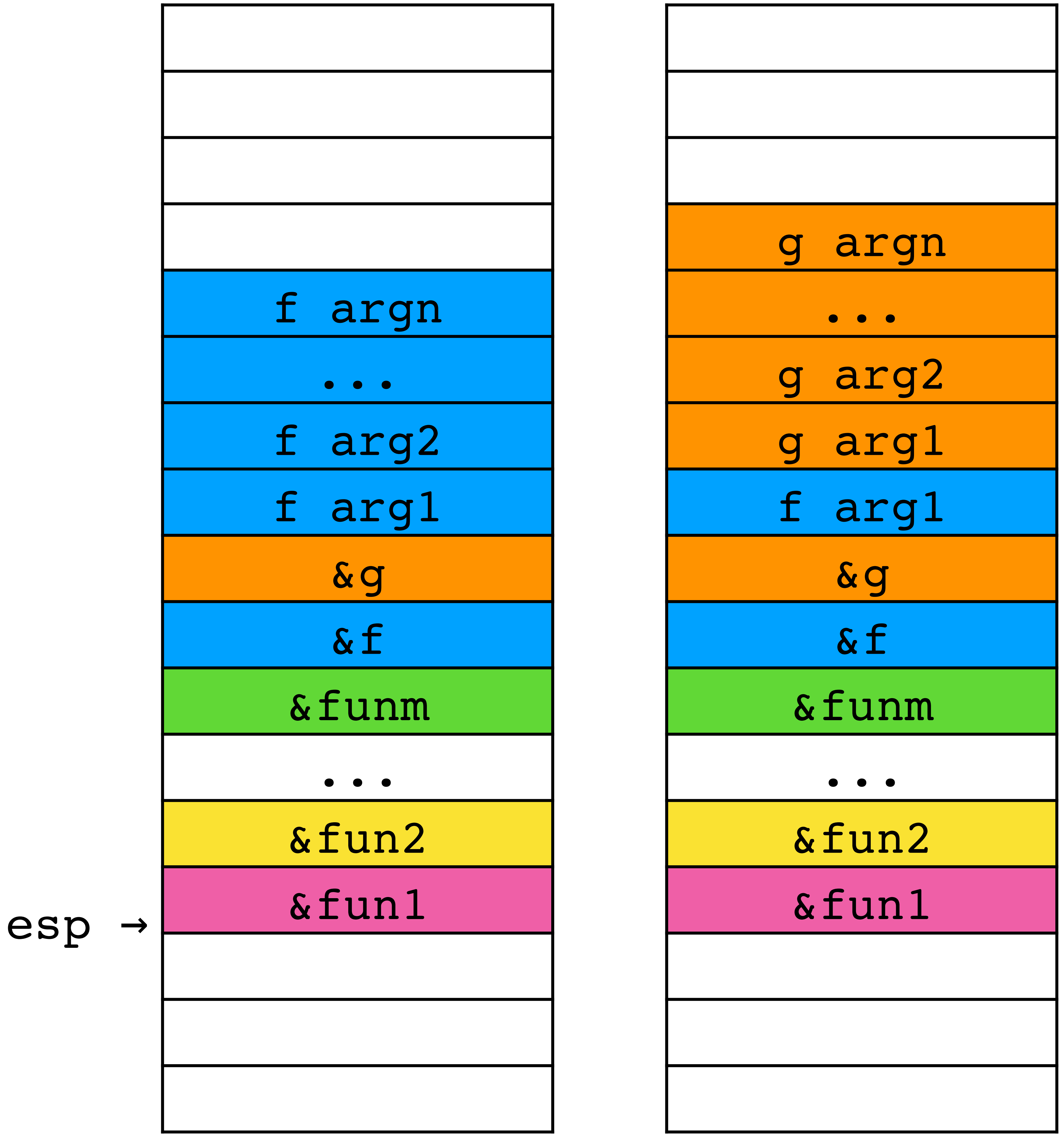
Chaining functions

- We can chain two functions together if
  - the first has one argument and the second any number of arguments; or
  - the first has any number of arguments and the second has none



Chaining functions

- We can chain two functions together if
  - the first has one argument and the second any number of arguments; or
  - the first has any number of arguments and the second has none
- We can start with any number of zero argument functions for either case

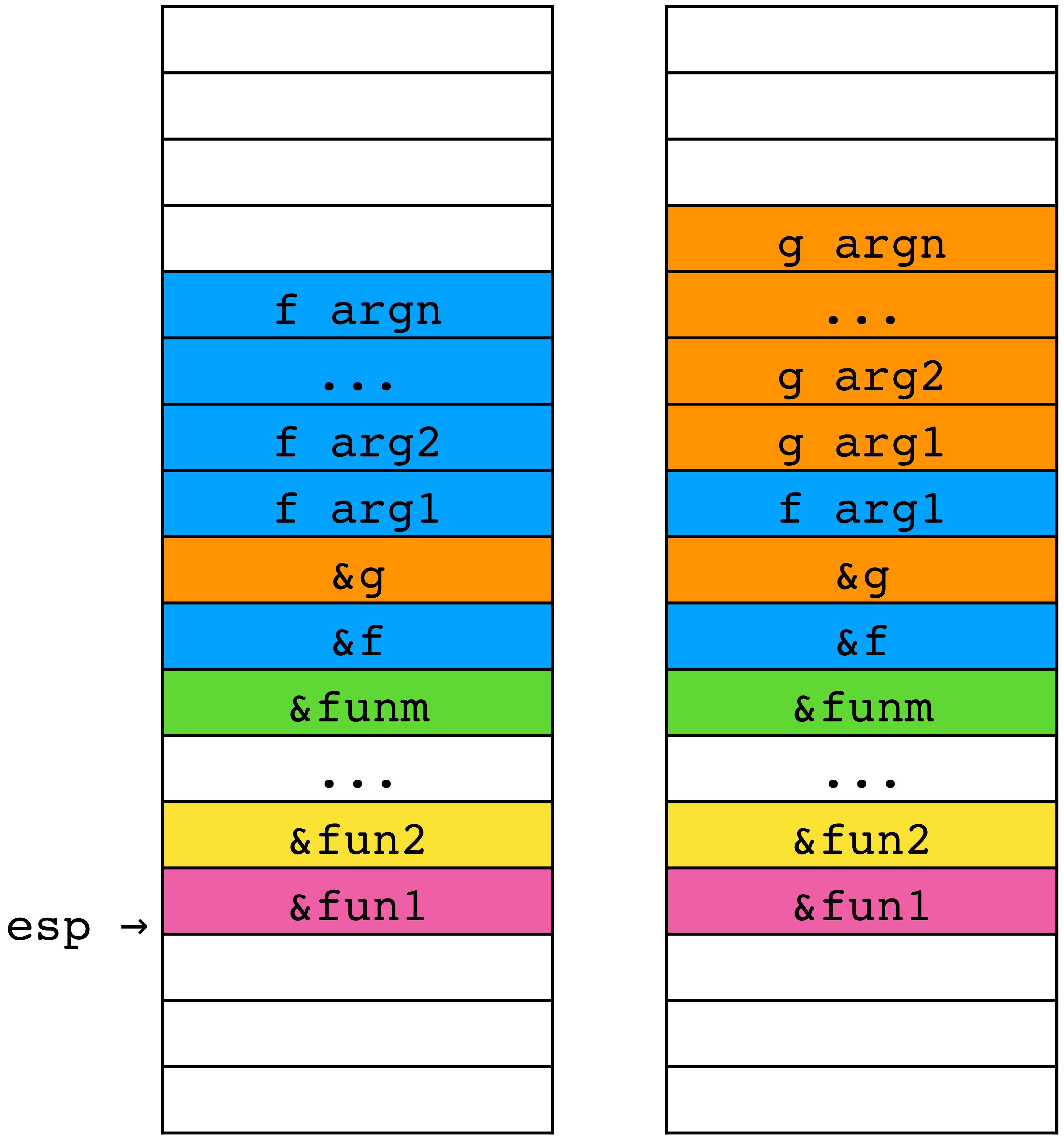


f argn
...
f arg2
f arg1
&g
&f
&funm
...
&fun2
&fun1
esp →

g argn
...
g arg2
g arg1
f arg1
&g
&f
&funm
...
&fun2
&fun1

Limitation

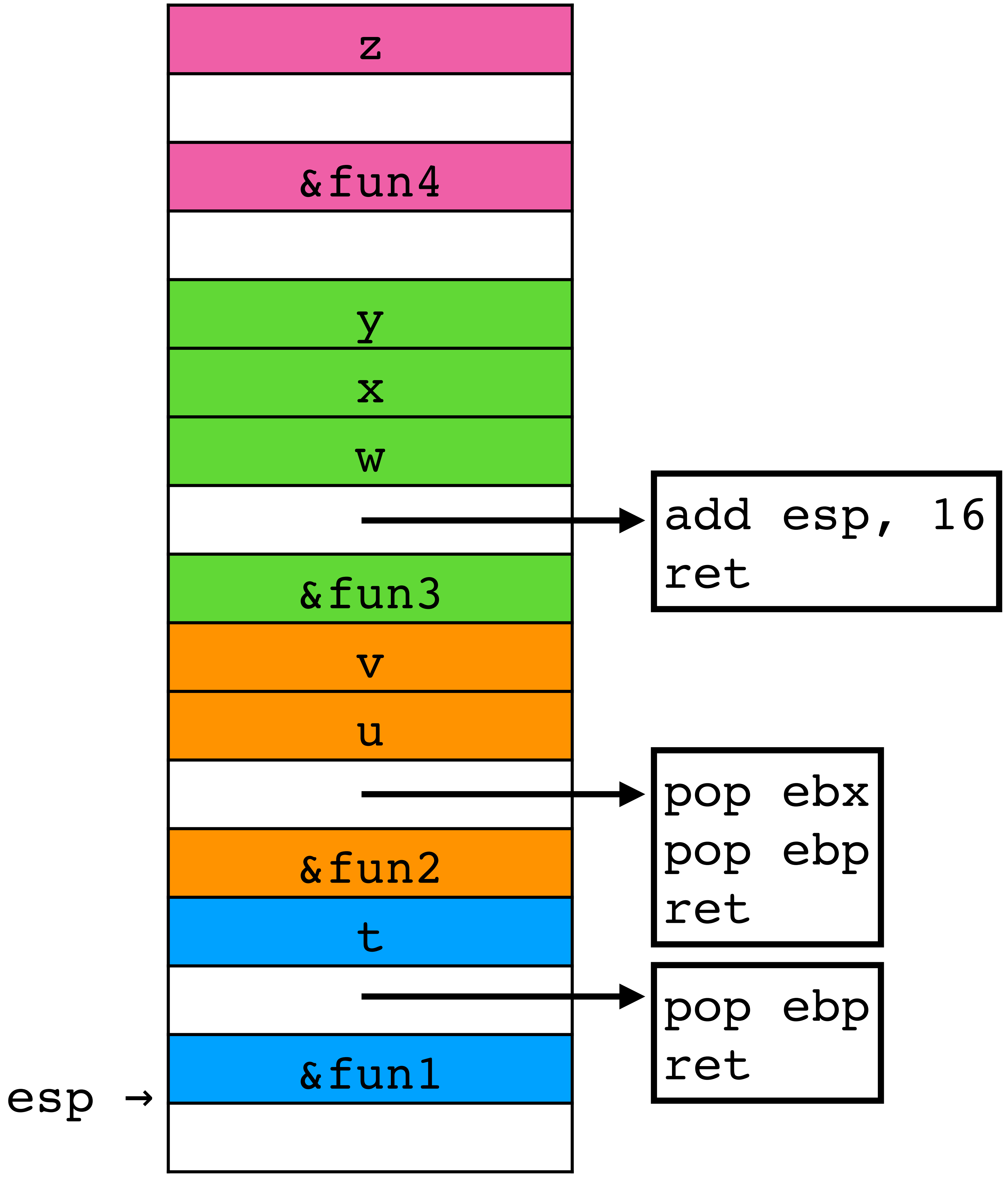
- We're limited to these cases because each function that gets called expects the top of the stack to contain the return address
- But the stack *also* has to contain the arguments
- If we want to call multiple functions, we need to advance the stack pointer over the arguments [How can we do that?]



		g argn
	f argn	...
	...	g arg2
	f arg2	g arg1
	f arg1	f arg1
	&g	&g
	&f	&f
	&funm	&funm
	...	...
	&fun2	&fun2
esp →	&fun1	&fun1

Advancing esp

- What if we want to chain the four function calls fun1(t), fun2(u,v), fun3(w,x,y), fun4(z)?
- **Identify pieces of code that advances the stack pointer beyond the arguments and return to those between function calls**
- Examples:
  - `pop ebp; ret`
  - `pop ebx; pop ebp; ret`
  - `add esp, 16; ret`



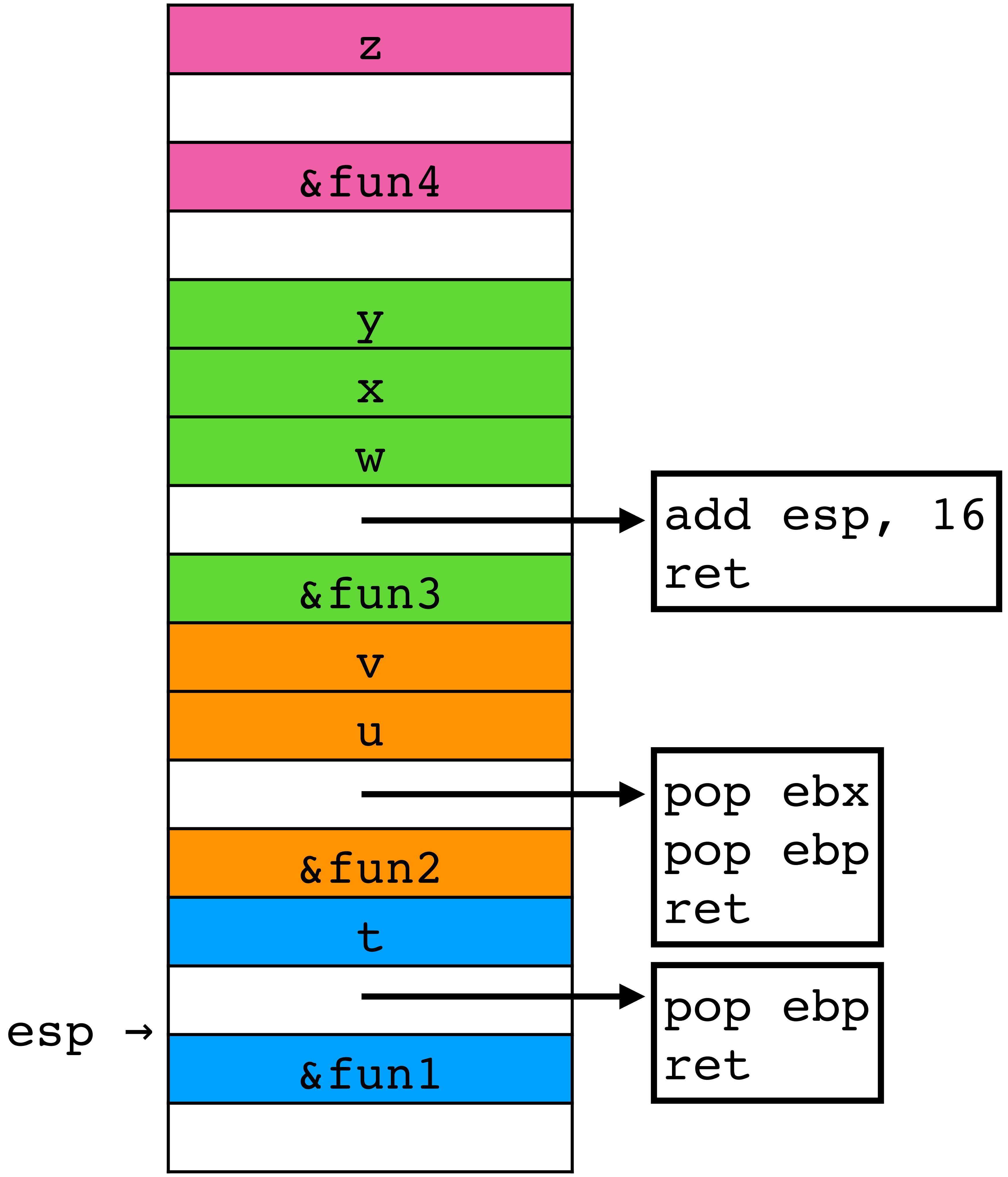
```
add esp, 16
ret
```

```
pop ebx
pop ebp
ret
```

```
pop ebp
ret
```

Running

1. Return to `fun1`



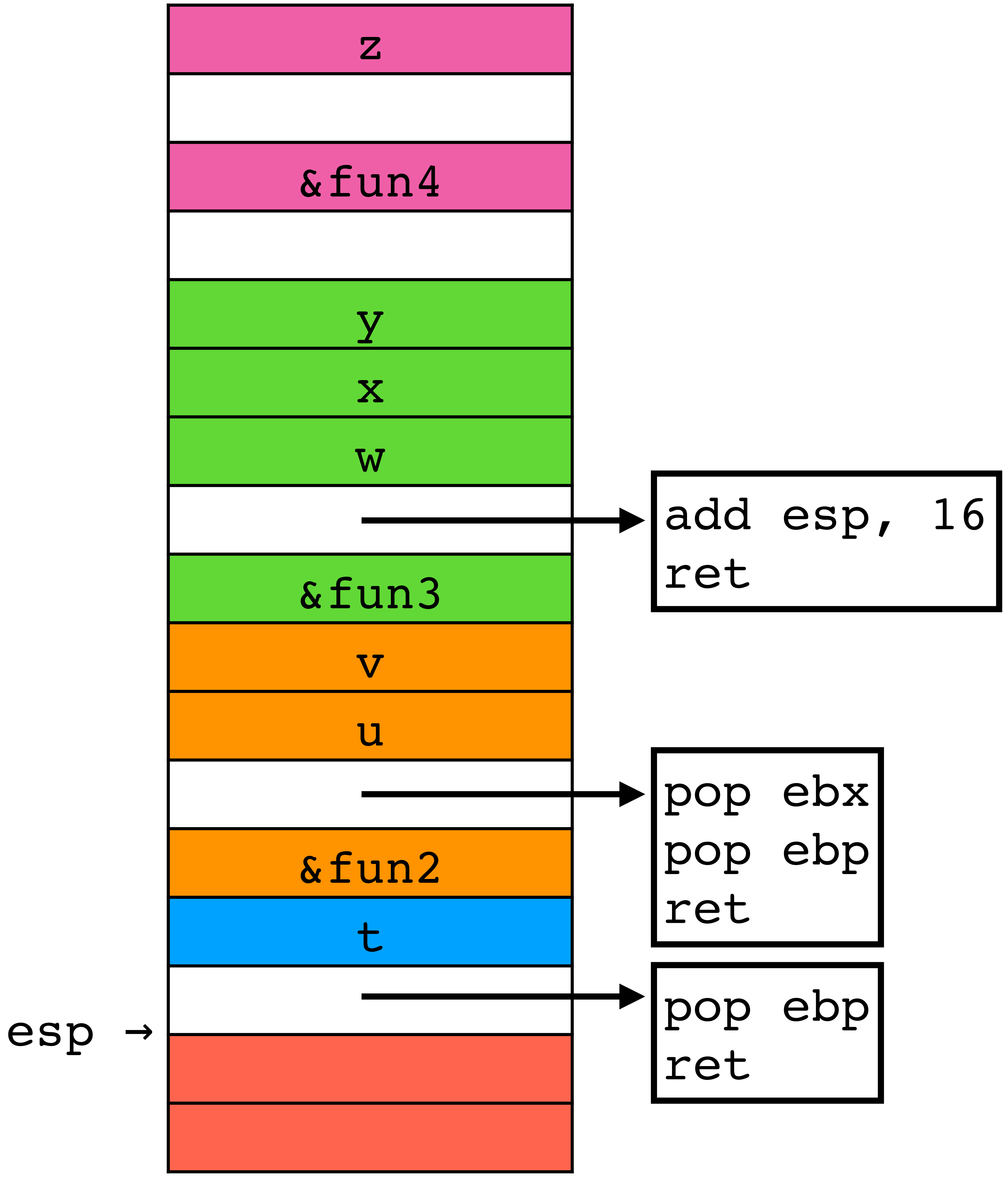
```
add esp, 16
ret
```

```
pop ebx
pop ebp
ret
```

```
pop ebp
ret
```

Running

1. Return to `fun1` which runs, modifies stack



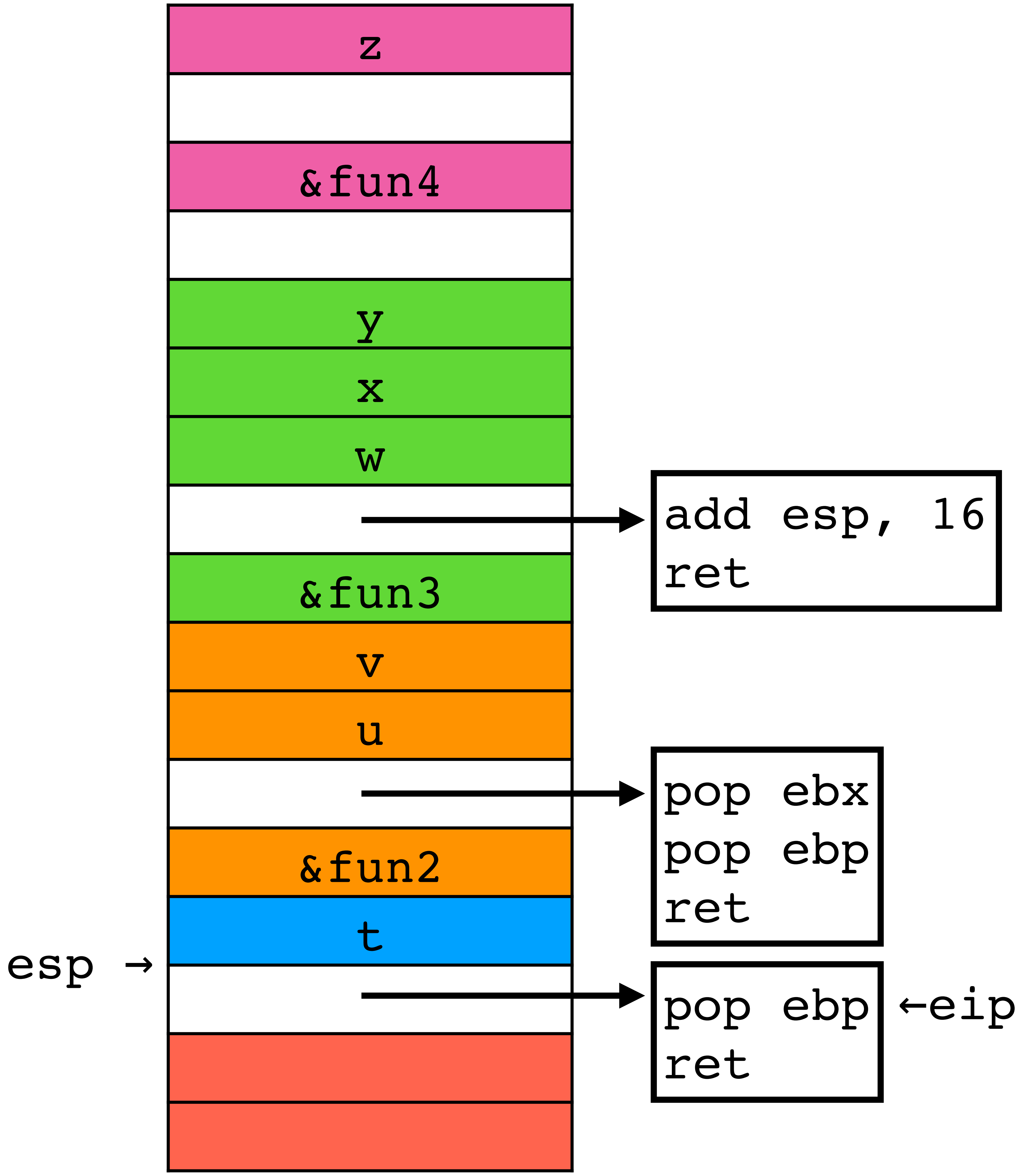
```
add esp, 16
ret
```

```
pop ebx
pop ebp
ret
```

```
pop ebp
ret
```

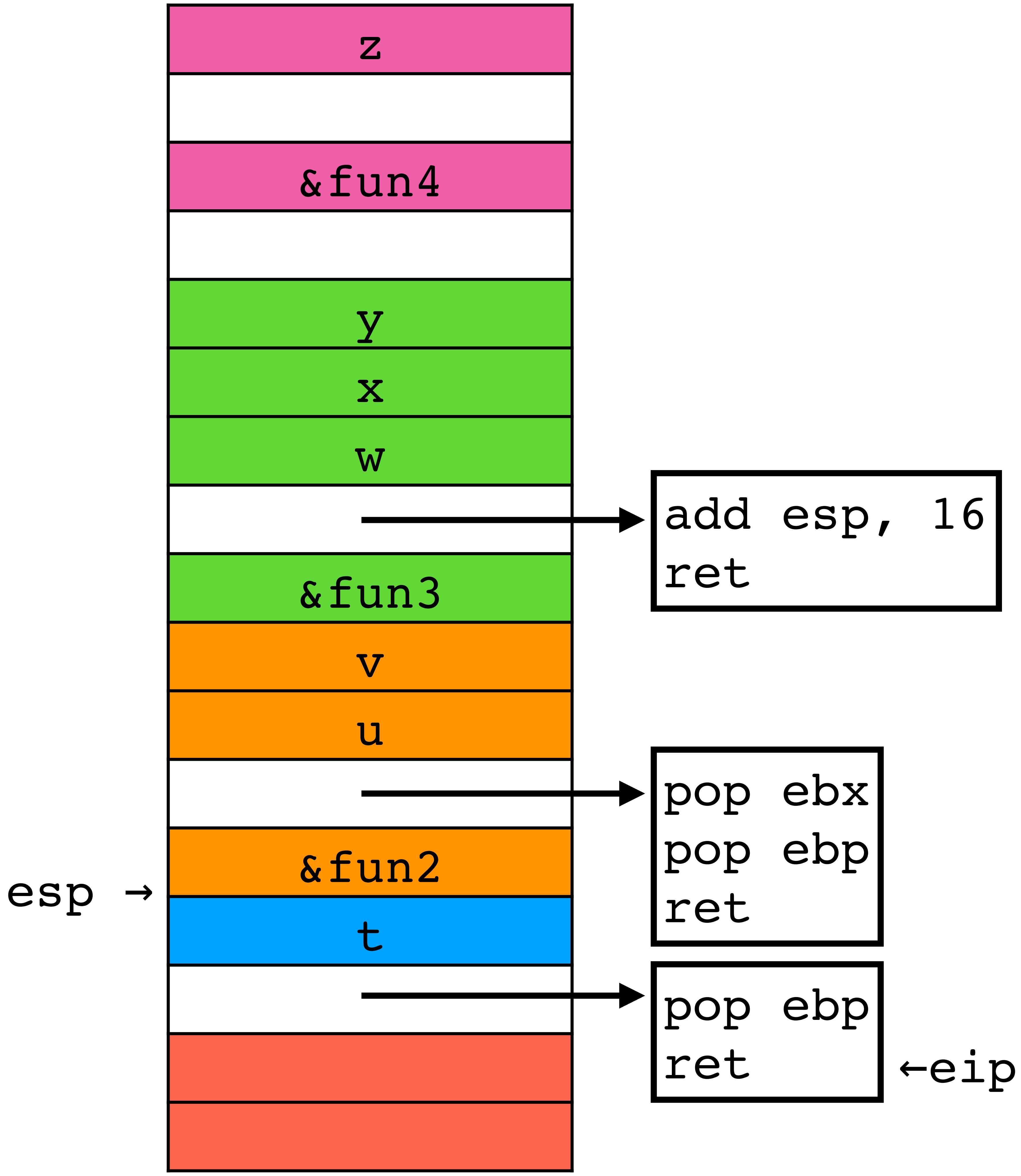
Running

1. Return to `fun1` which runs, modifies stack
2. Return to pop; ret



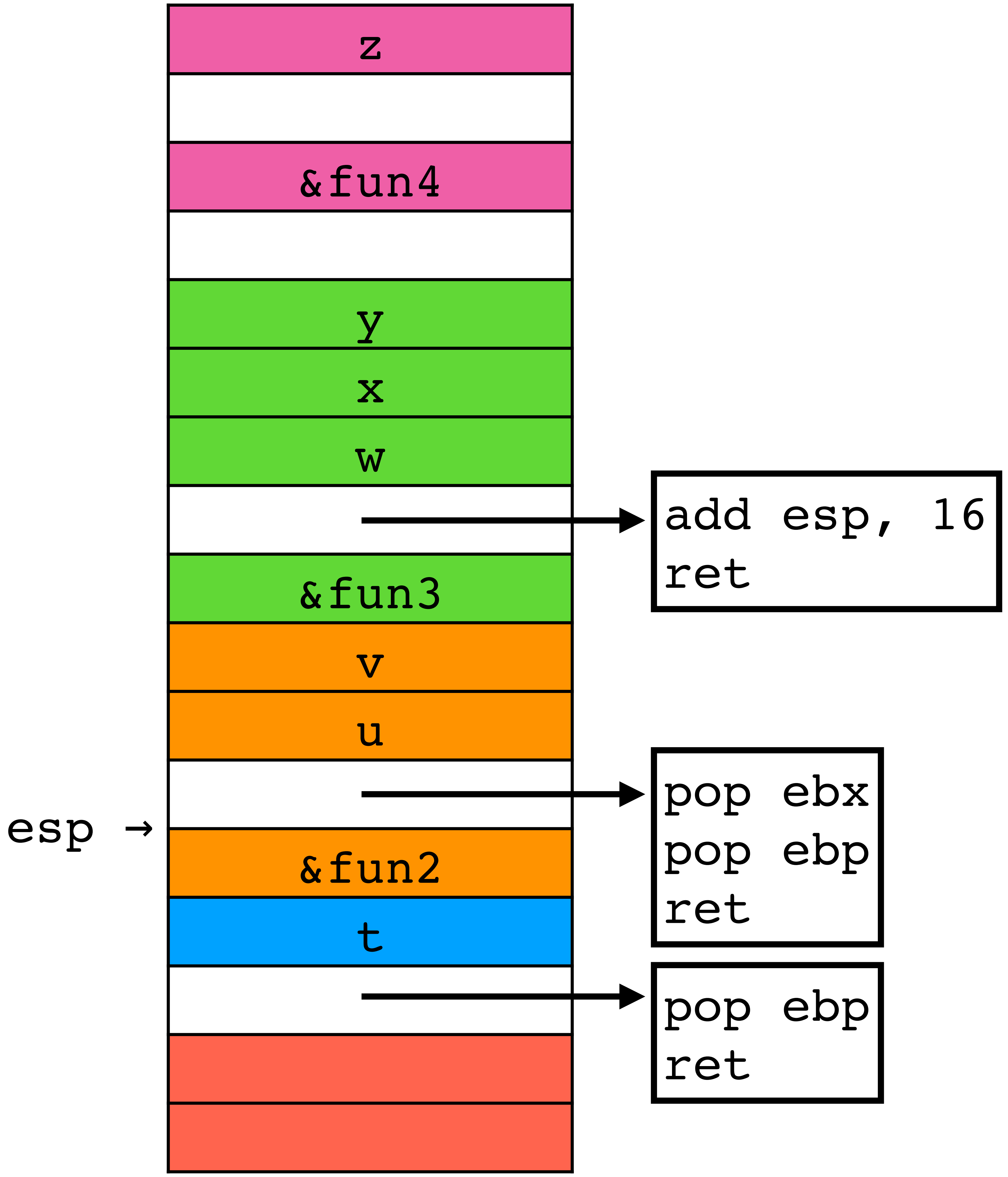
Running

1. Return to `fun1` which runs, modifies stack
2. Return to pop; ret



Running

1. Return to `fun1` which runs, modifies stack
2. Return to pop; ret
3. Return to `fun2`



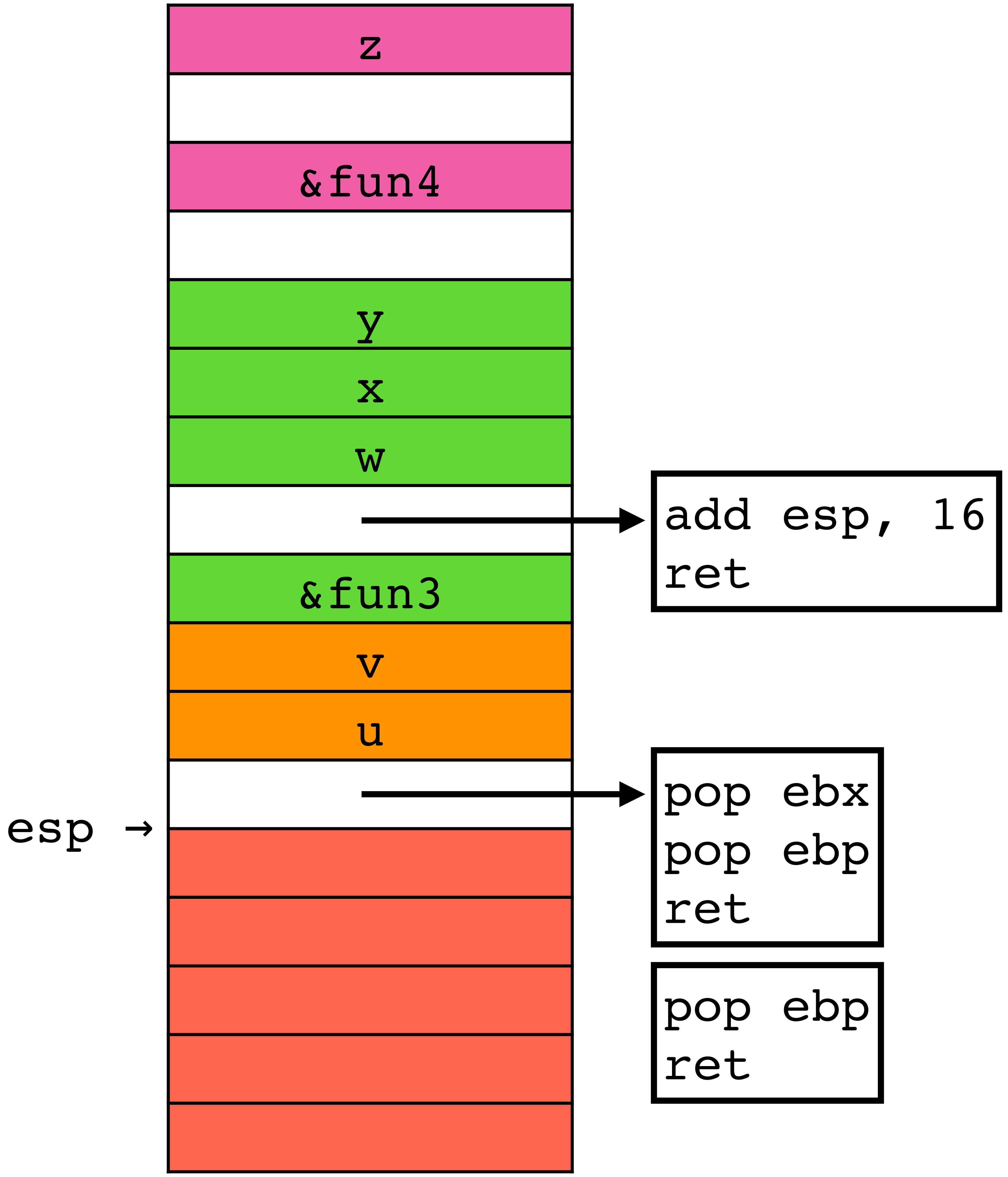
```
add esp, 16
ret
```

```
pop ebx
pop ebp
ret
```

```
pop ebp
ret
```

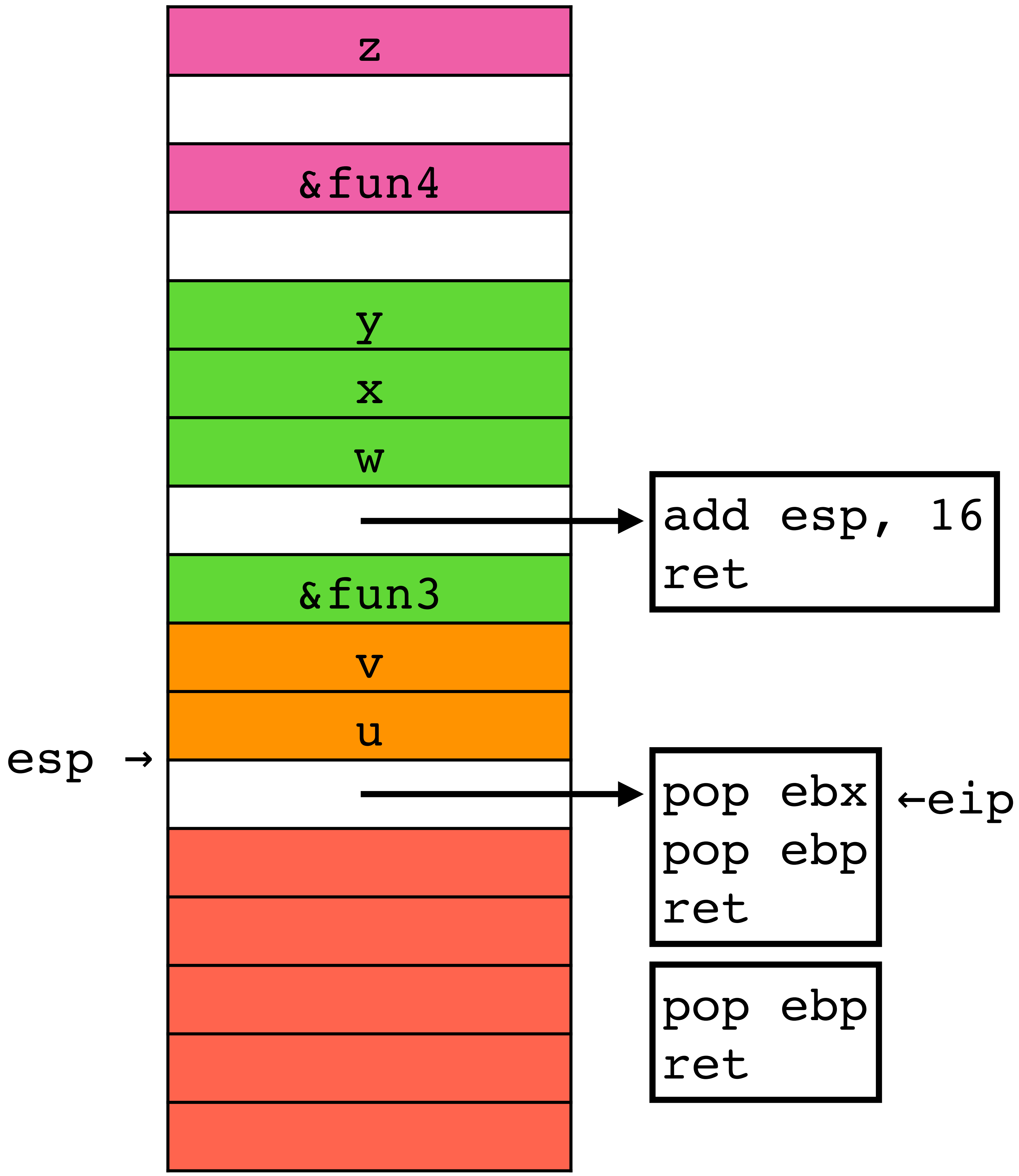
Running

1. Return to `fun1` which runs, modifies stack
2. Return to pop; ret
3. Return to `fun2` which runs, modifies stack



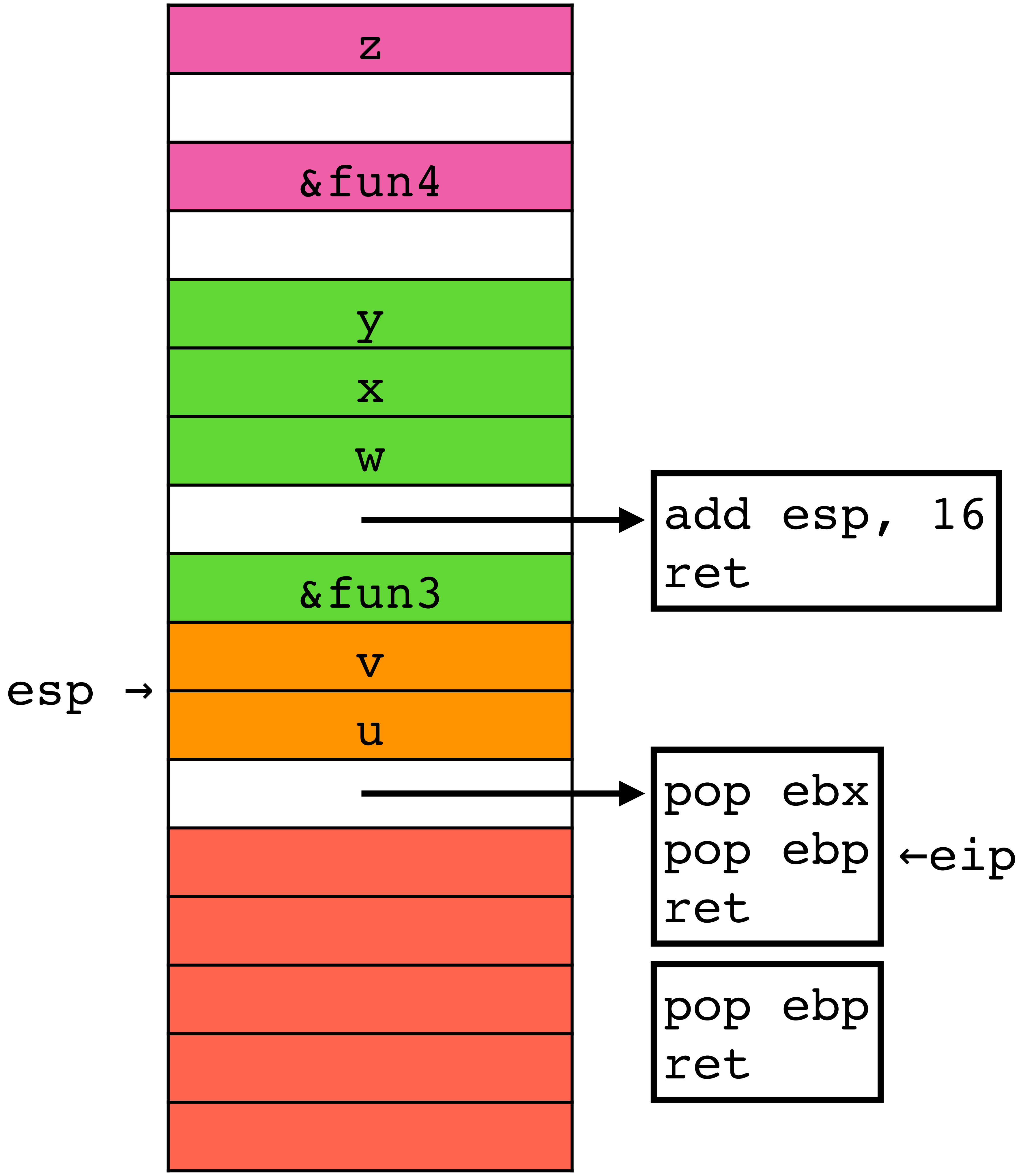
Running

1. Return to `fun1` which runs, modifies stack
2. Return to pop; ret
3. Return to `fun2` which runs, modifies stack
4. Return to pop; pop; ret



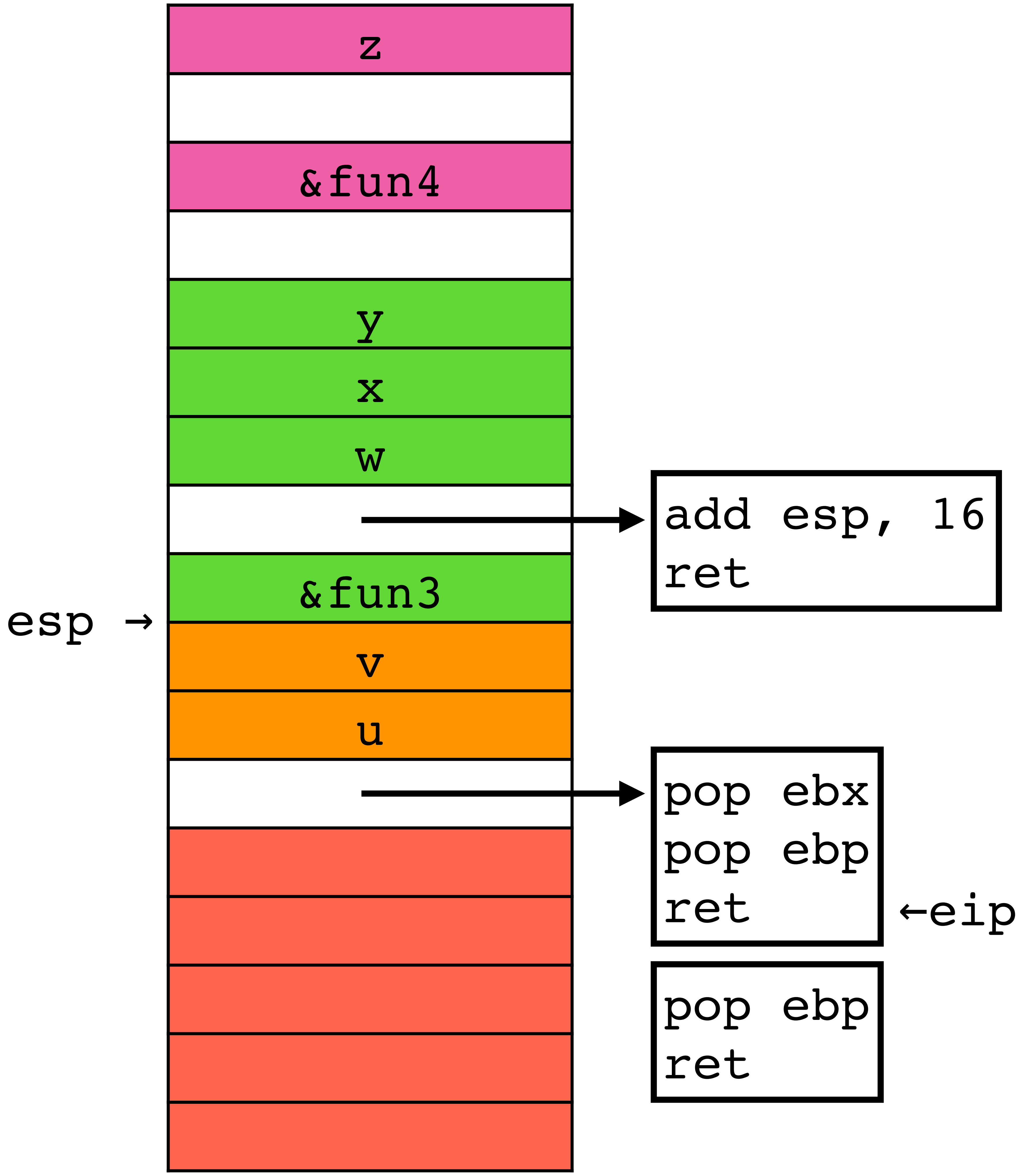
Running

1. Return to `fun1` which runs, modifies stack
2. Return to pop; ret
3. Return to `fun2` which runs, modifies stack
4. Return to pop; pop; ret



Running

1. Return to `fun1` which runs, modifies stack
2. Return to pop; ret
3. Return to `fun2` which runs, modifies stack
4. Return to pop; pop; ret



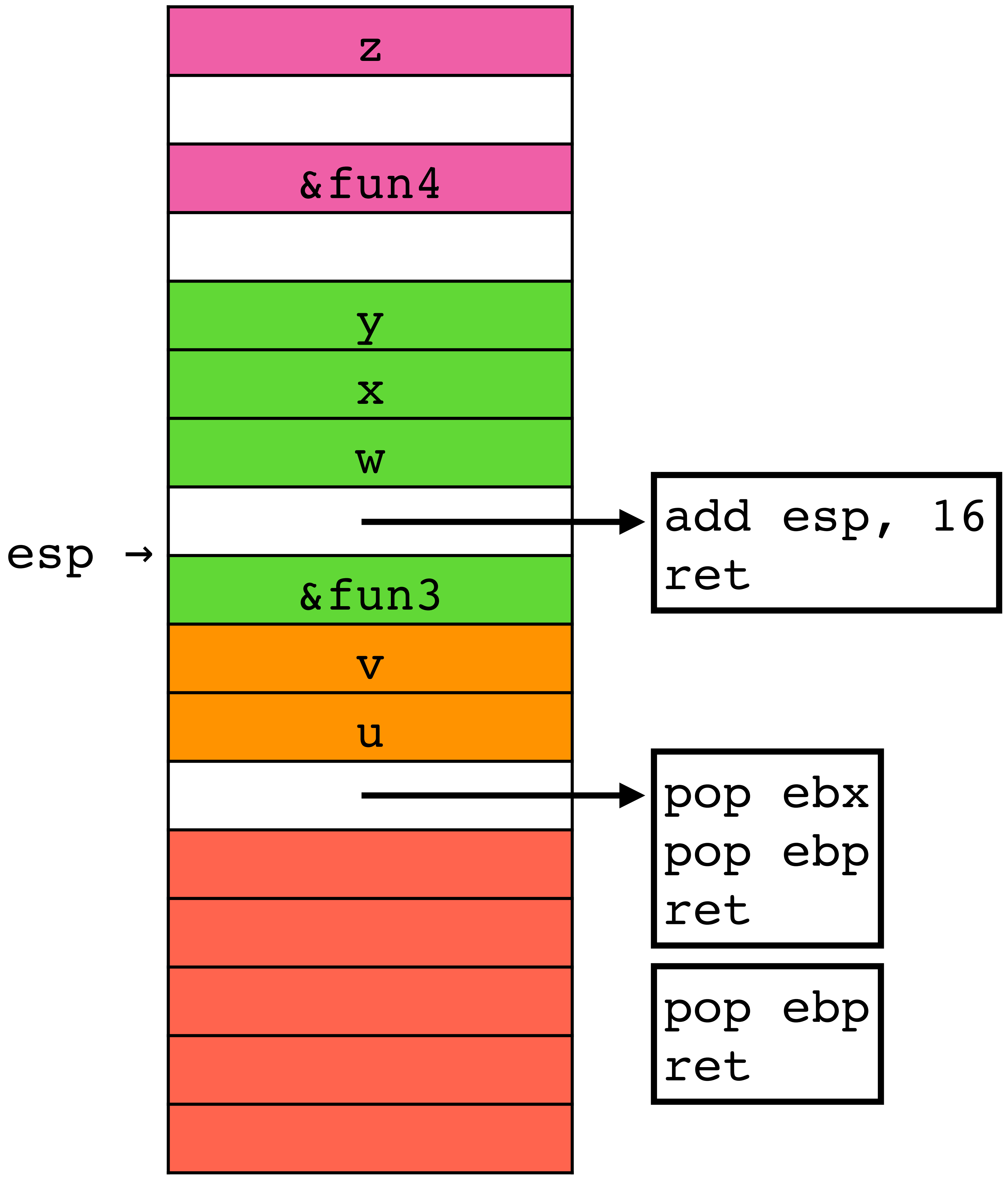
```
add esp, 16
ret
```

```
pop ebx
pop ebp
ret
```

```
pop ebp
ret
```

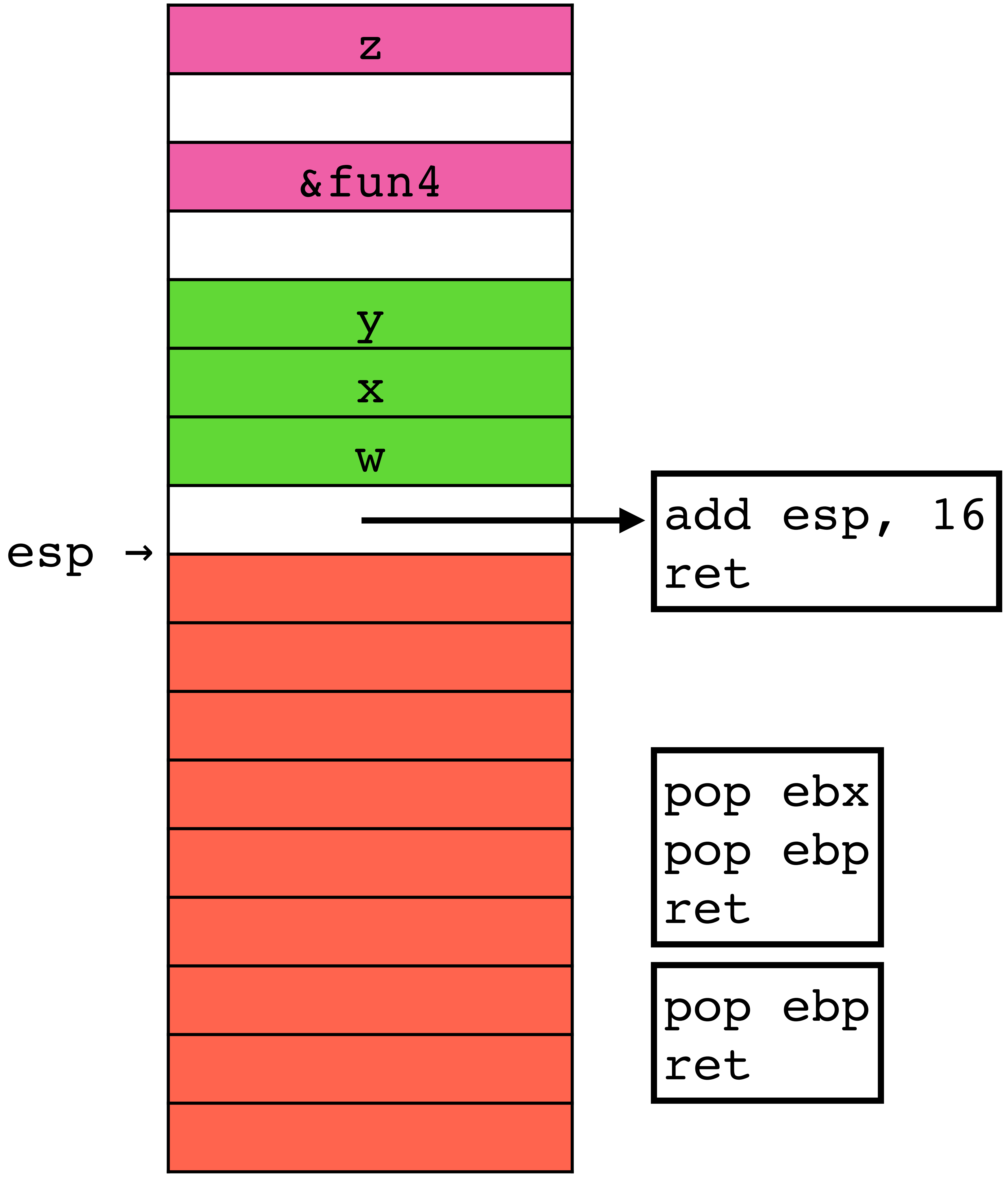
Running

1. Return to `fun1` which runs, modifies stack
2. Return to pop; ret
3. Return to `fun2` which runs, modifies stack
4. Return to pop; pop; ret
5. Return to `fun3`



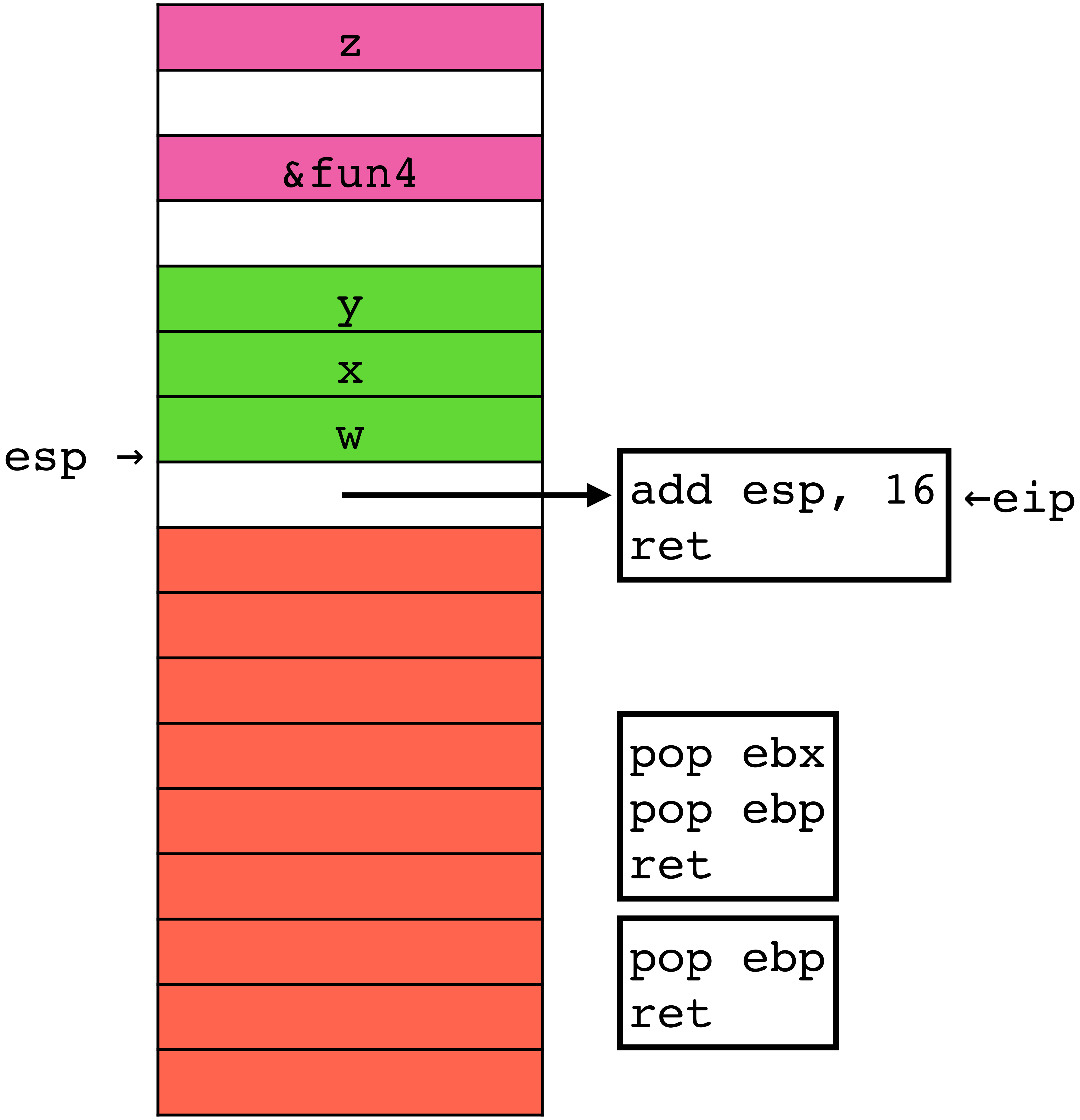
Running

1. Return to `fun1` which runs, modifies stack
2. Return to pop; ret
3. Return to `fun2` which runs, modifies stack
4. Return to pop; pop; ret
5. Return to `fun3` which runs, modifies stack



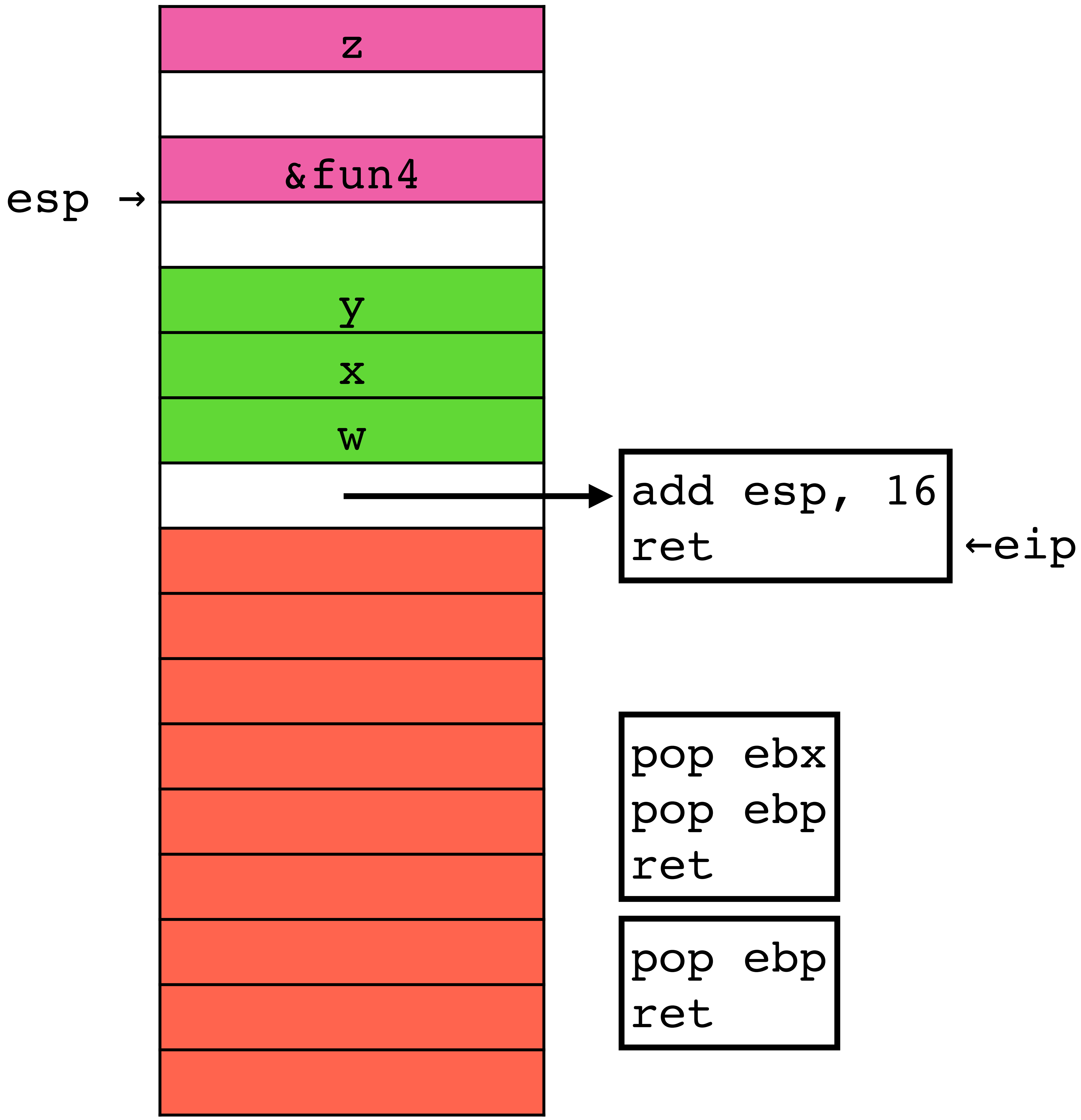
Running

1. Return to `fun1` which runs, modifies stack
2. Return to pop; ret
3. Return to `fun2` which runs, modifies stack
4. Return to pop; pop; ret
5. Return to `fun3` which runs, modifies stack
6. Return to add; ret



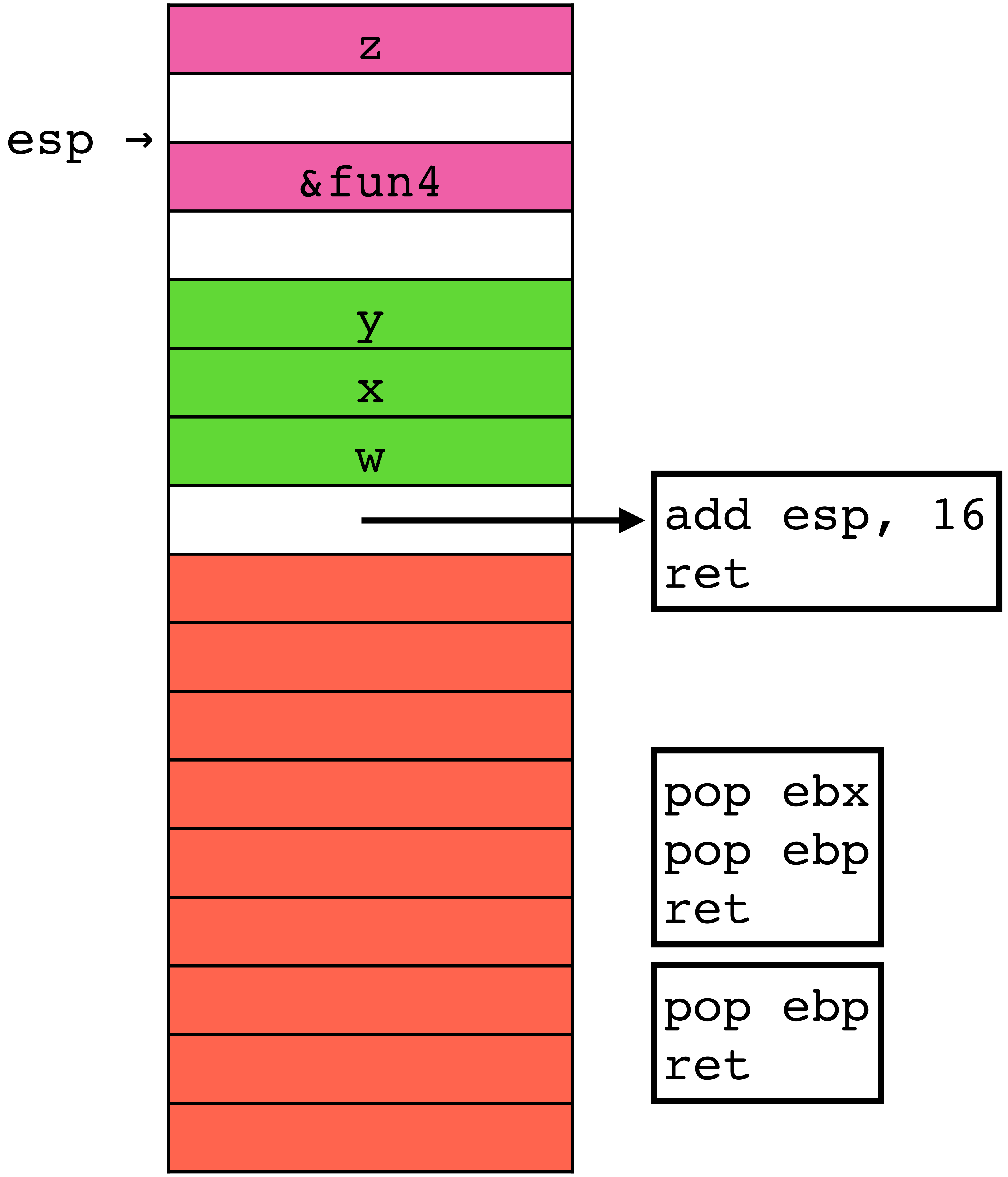
Running

1. Return to `fun1` which runs, modifies stack
2. Return to pop; ret
3. Return to `fun2` which runs, modifies stack
4. Return to pop; pop; ret
5. Return to `fun3` which runs, modifies stack
6. Return to add; ret



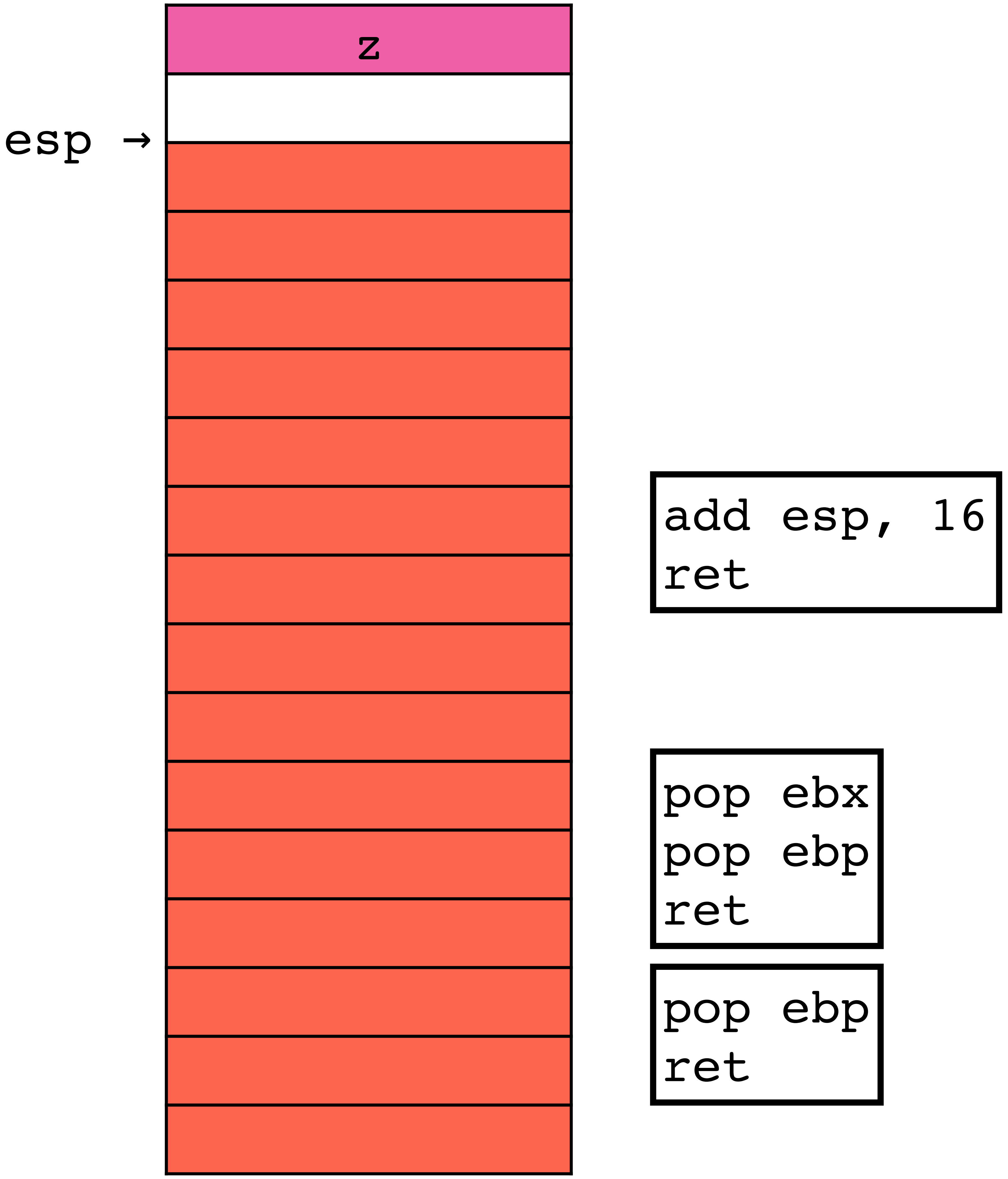
Running

1. Return to `fun1` which runs, modifies stack
2. Return to pop; ret
3. Return to `fun2` which runs, modifies stack
4. Return to pop; pop; ret
5. Return to `fun3` which runs, modifies stack
6. Return to add; ret
7. Return to `fun4`



Running

1. Return to `fun1` which runs, modifies stack
2. Return to pop; ret
3. Return to `fun2` which runs, modifies stack
4. Return to pop; pop; ret
5. Return to `fun3` which runs, modifies stack
6. Return to add; ret
7. Return to `fun4` which runs, modifies stack



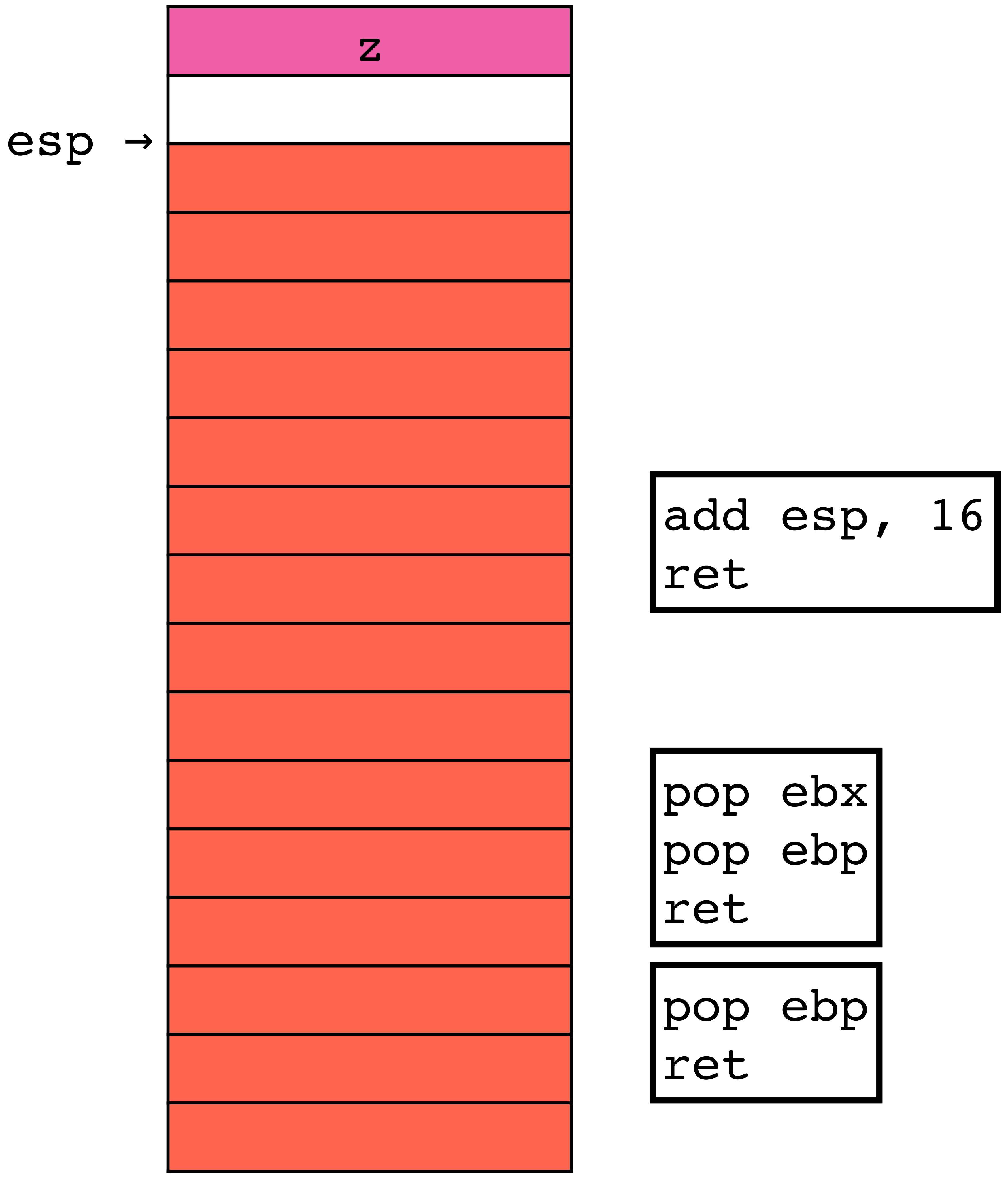
```
add esp, 16
ret
```

```
pop ebx
pop ebp
ret
```

```
pop ebp
ret
```

Running

1. Return to `fun1` which runs, modifies stack
2. Return to pop; ret
3. Return to `fun2` which runs, modifies stack
4. Return to pop; pop; ret
5. Return to `fun3` which runs, modifies stack
6. Return to add; ret
7. Return to `fun4` which runs, modifies stack
8. Et cetera



```
add esp, 16
ret
```

```
pop ebx
pop ebp
ret
```

```
pop ebp
ret
```

Stack advancing code

- Two key pieces
  - Stack modification (pop or add esp). Modifies the stack pointer to move over the arguments to the function
  - Return at the end. Returns to the next function whose address is on the stack
- Together, this lets us chain a more or less arbitrary number of function calls **with constant parameters**
  - Depends on how much stack space we have (but we can change the stack pointer via a sequence like `xchg eax, esp; ret`)
  - Depends on what advancing code we can find in the program/libraries (turns out there's a whole lot there)

ret2libc on x86-64

We can take the same approach as before: return to addresses of functions

Some limitations apply

- Addresses will contain 0 bytes
- Arguments are in rdi, rsi, rdx, etc. rather than on the stack

We’ve seen how to deal with 0 bytes (don’t use strcpy()/strcat())

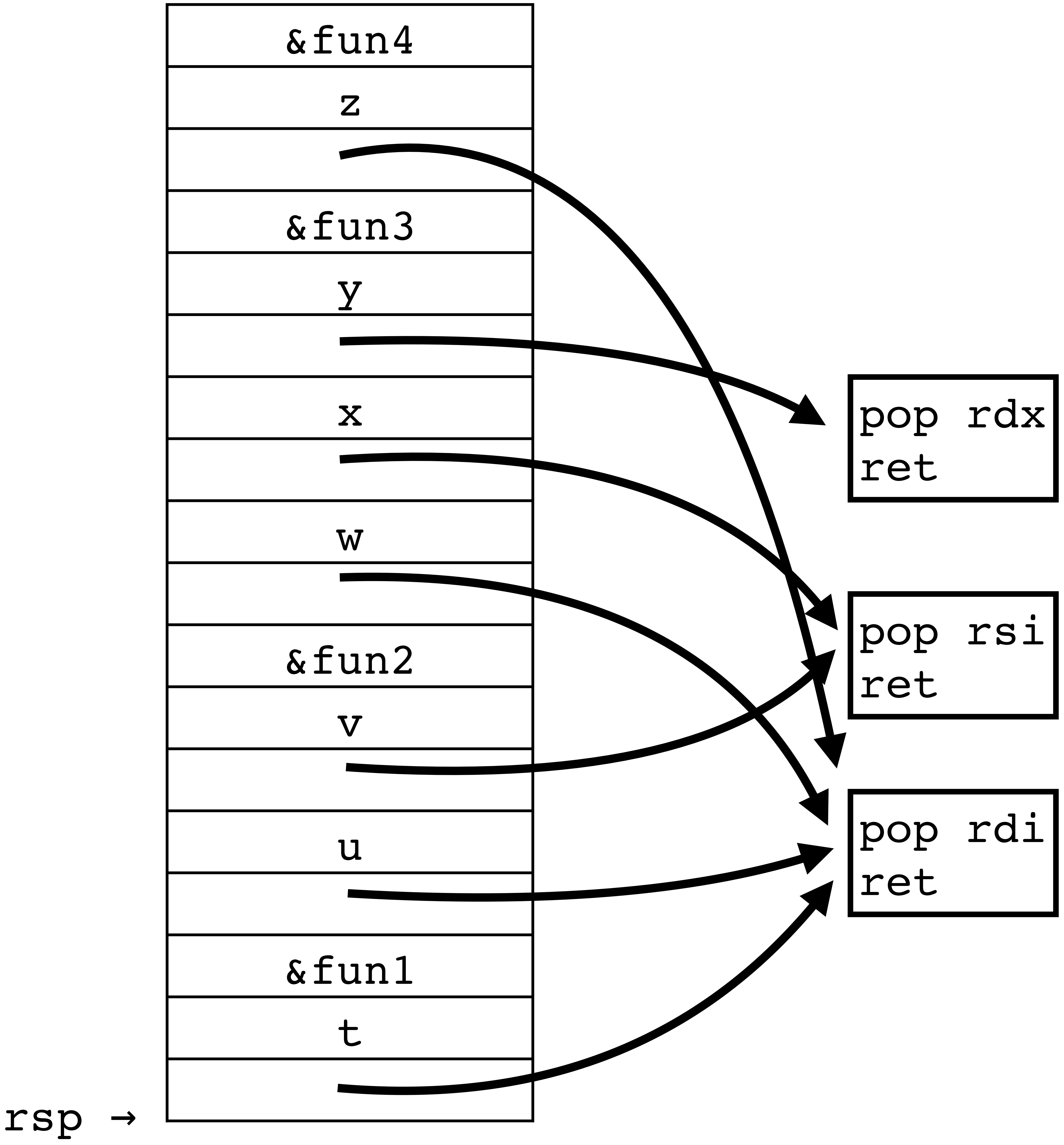
How can we get arguments from the stack into registers before we return to the function? Hint: Think about the stack advancing code!

ret2libc on x86-64

We can return to `pop rdi ; ret` or `pop rsi ; ret` sequences

This will pop a value from the stack and advance the stack pointer at the same time

```
fun1(t)
fun2(u, v)
fun3(w, x, y)
fun4(z)
```



Next time

- There's no need to limit ourselves to returning to functions and advancing code
- We can encode arbitrary computation (including conditionals and loops) by returning to sequences of code ending in `ret`