

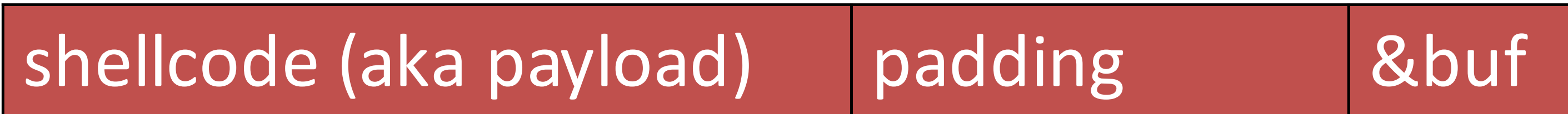
Lecture 12 – Code reuse attacks

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 $W \wedge X$ 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



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
 - Google Chrome 61.0.3163.91 links to 99 dynamic libraries!
 - 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

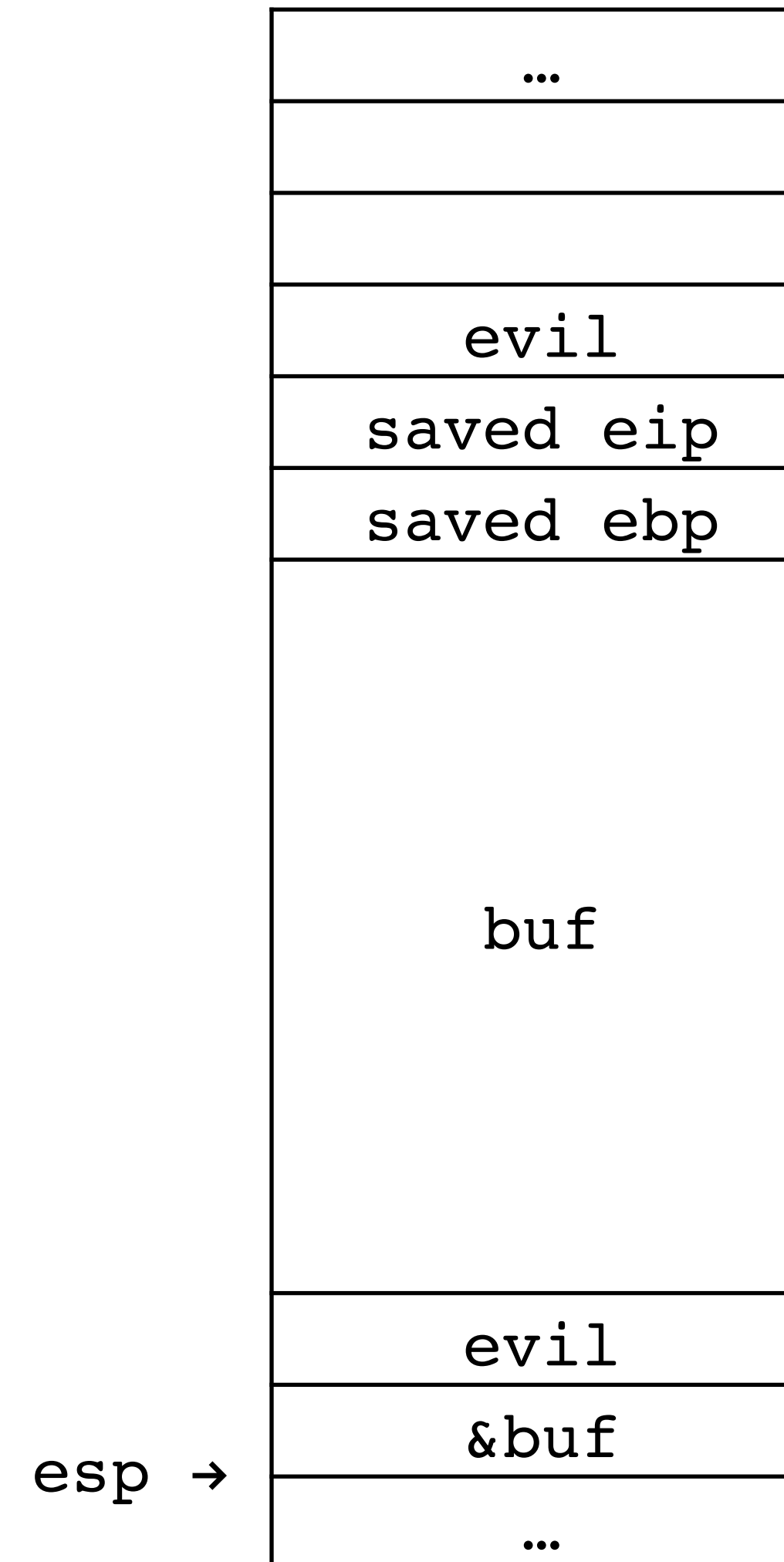
```
int system(const char *command);
```

Simple example

- Consider

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

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

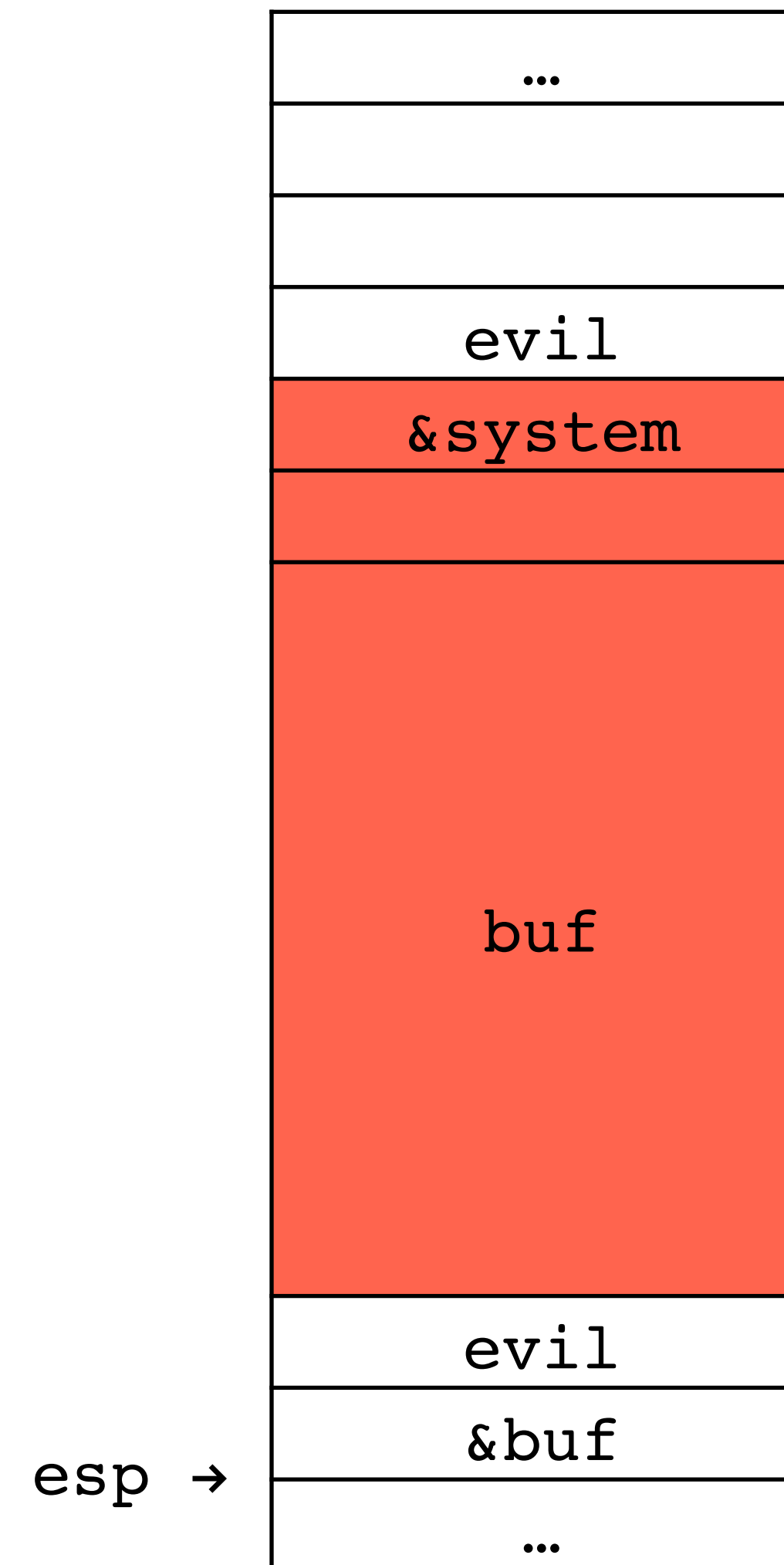


Simple example

- Consider

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

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

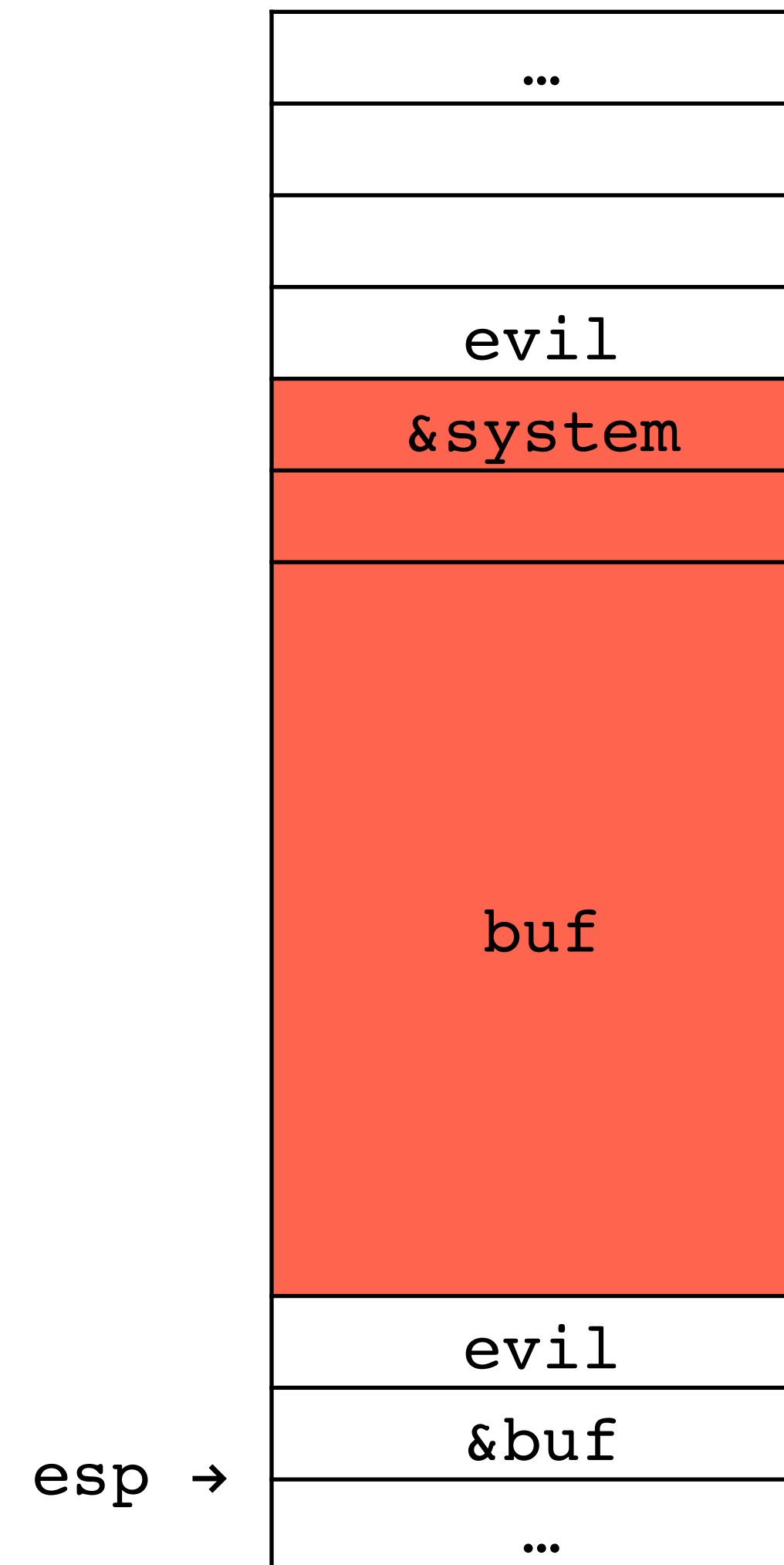


Simple example

- Consider

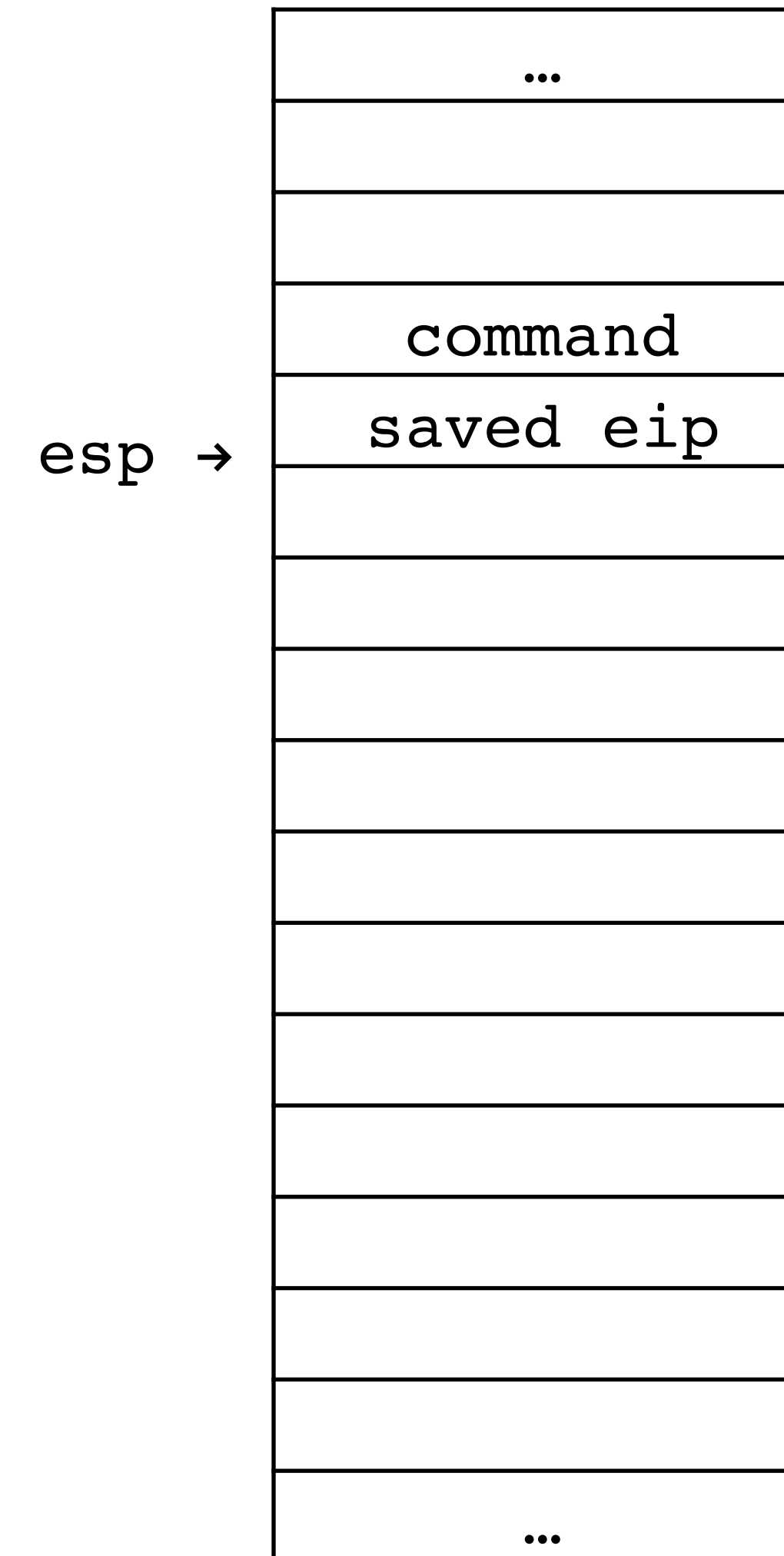
```
void foo(char *evil) {  
    char buf[32];  
    strcpy(buf, evil);  
}
```

- 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?



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`

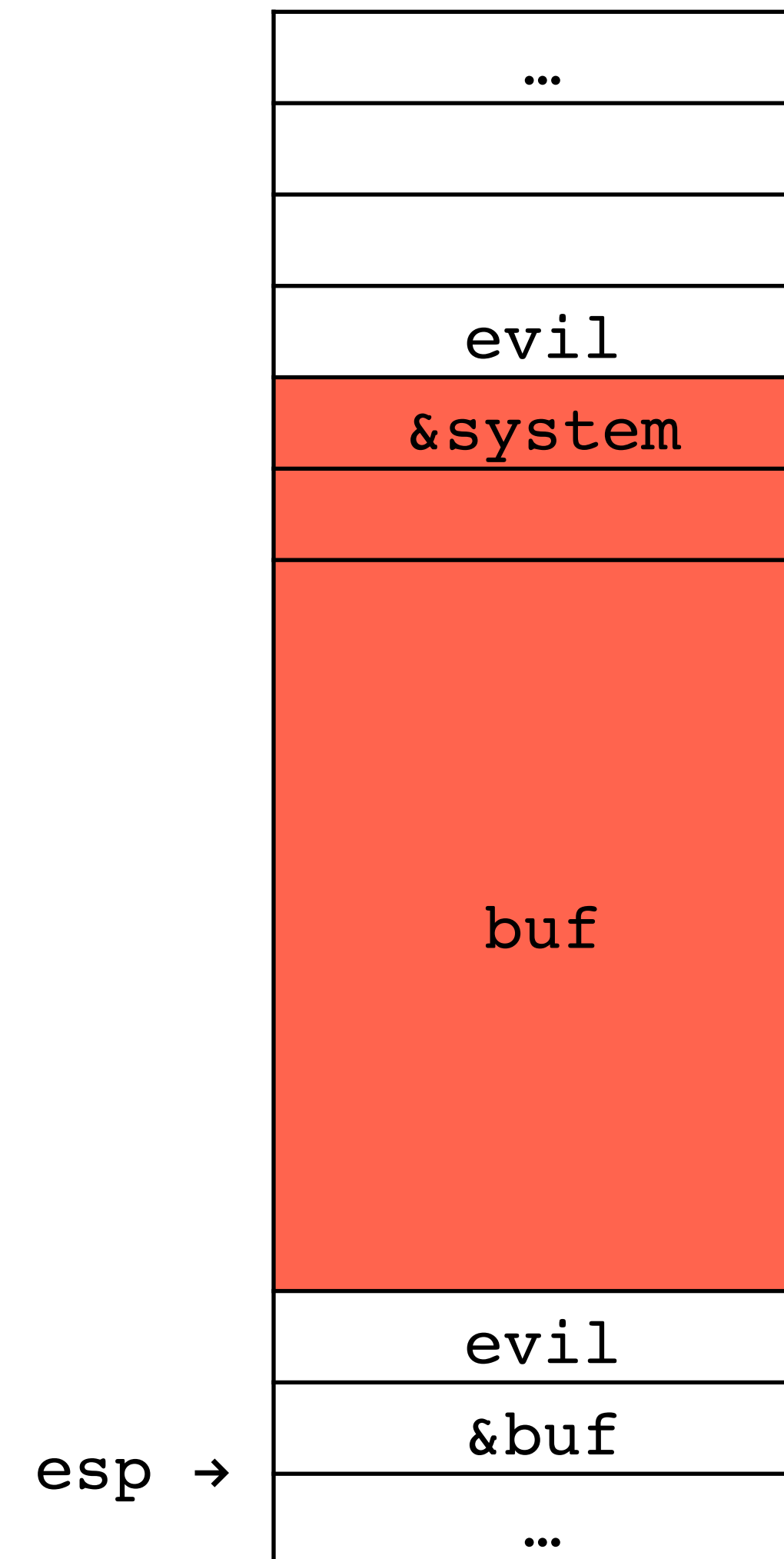


Simple example

- Consider

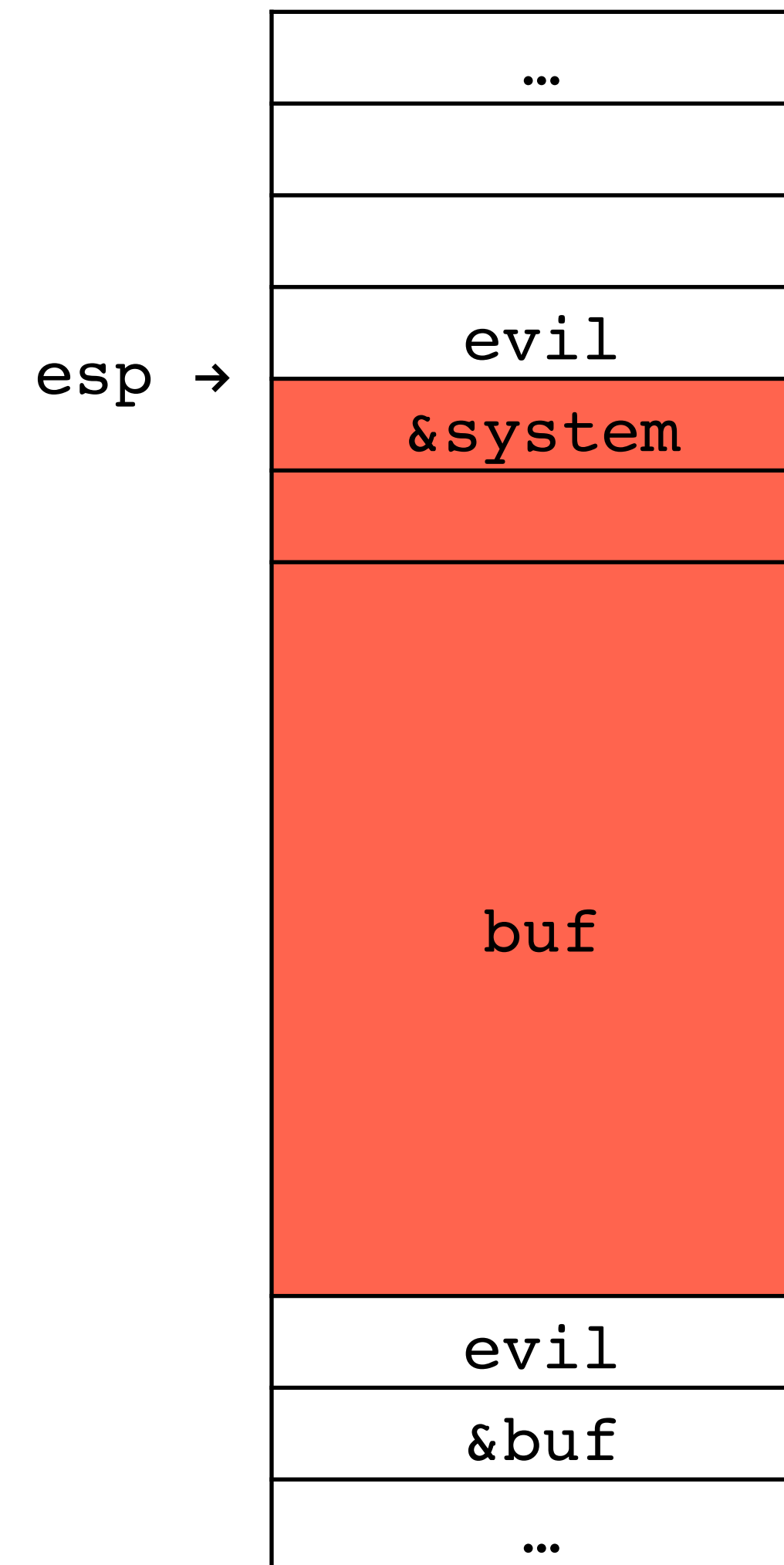
```
void foo(char *evil) {  
    char buf[32];  
    strcpy(buf, evil);  
}
```

- 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*



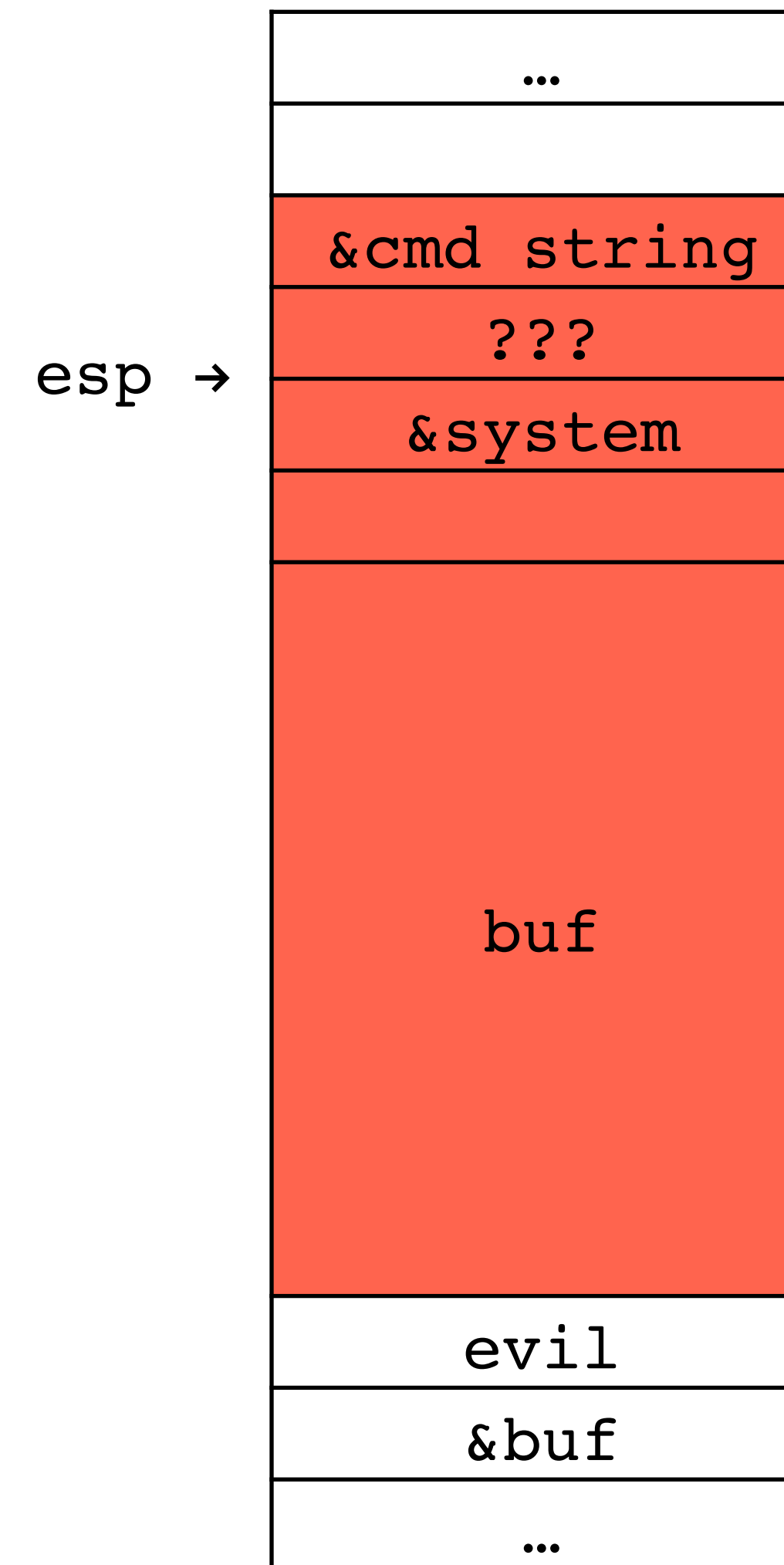
Simple example

- `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



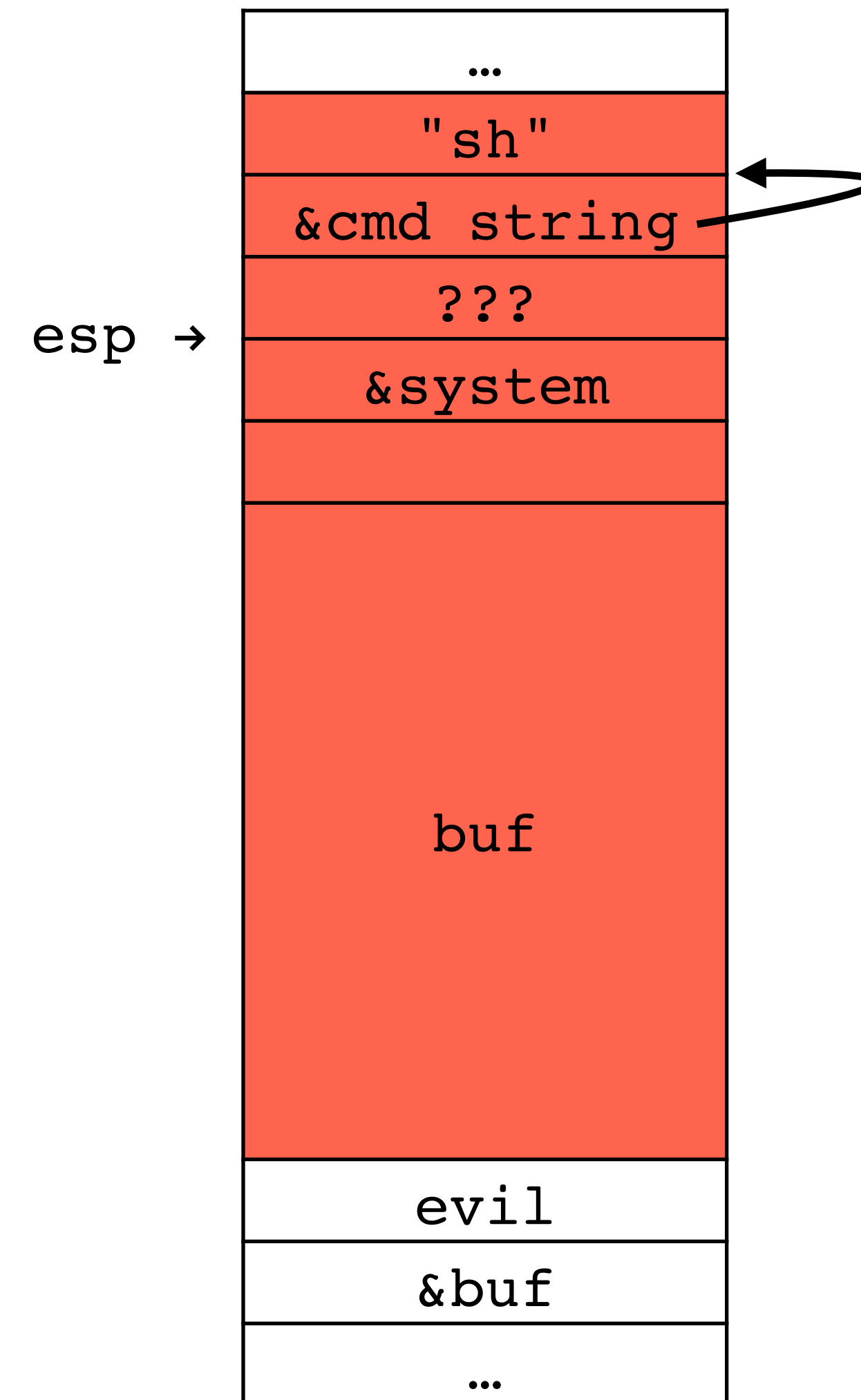
Simple example

- `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?



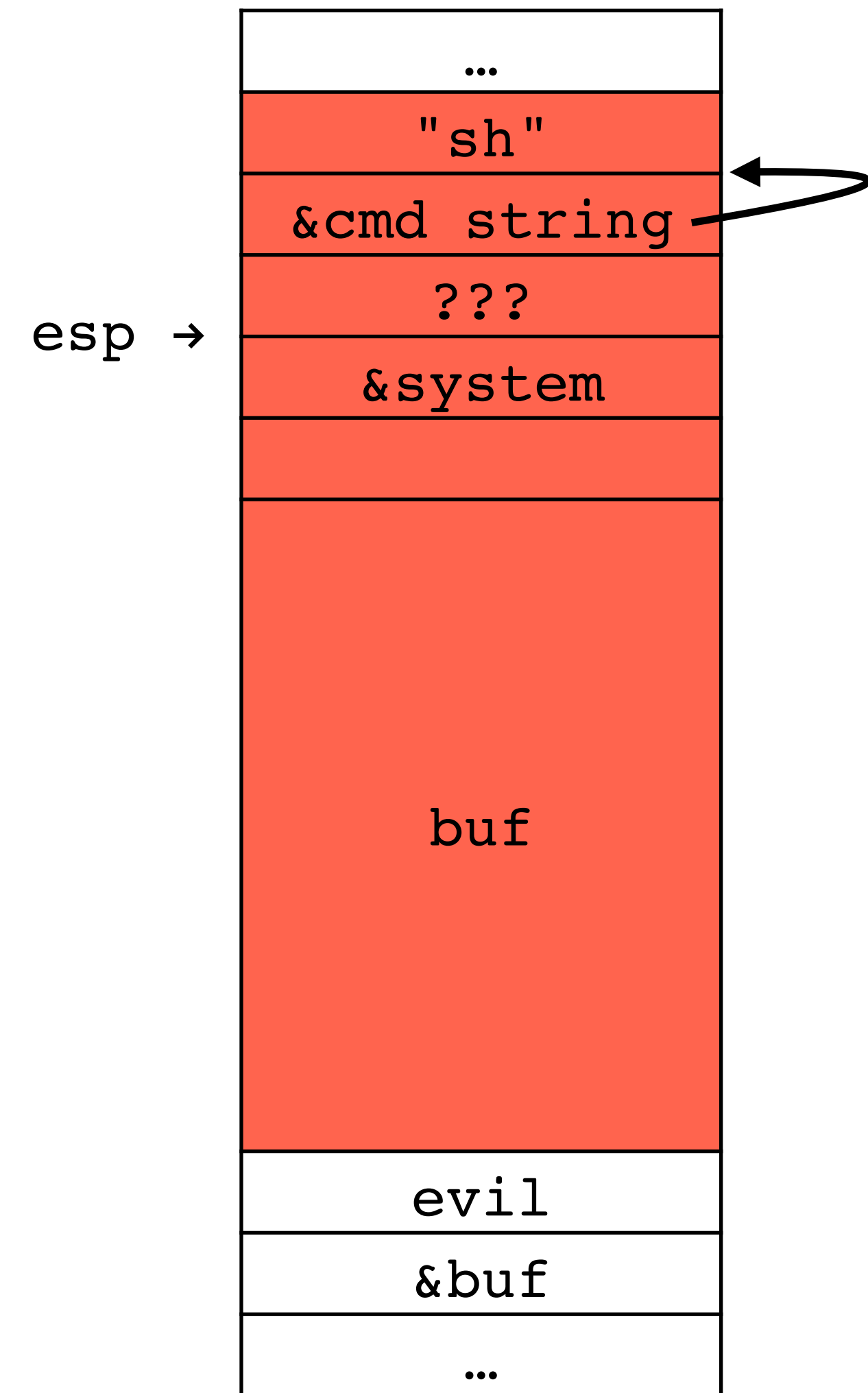
Simple example

- `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?



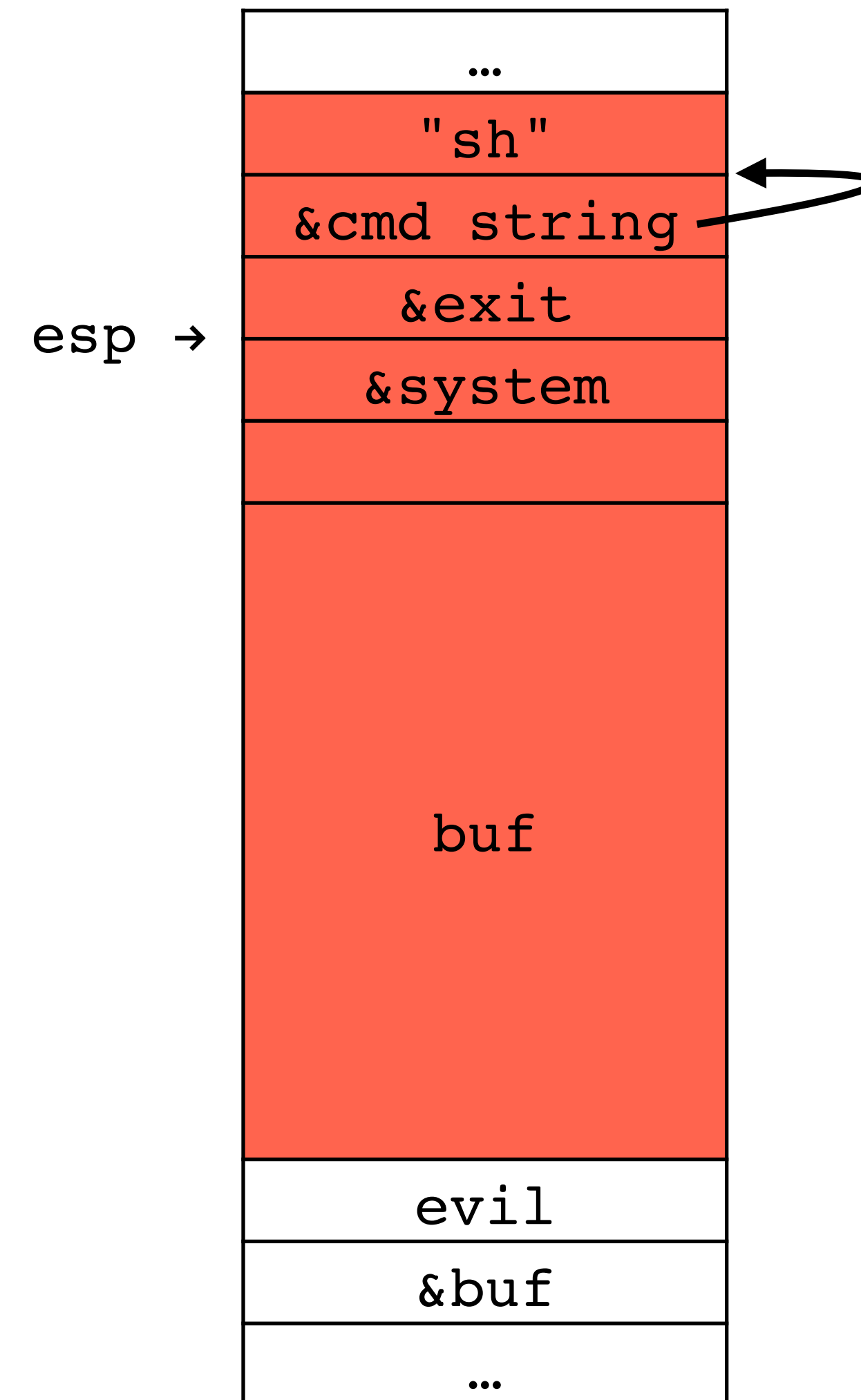
Simple example

- 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

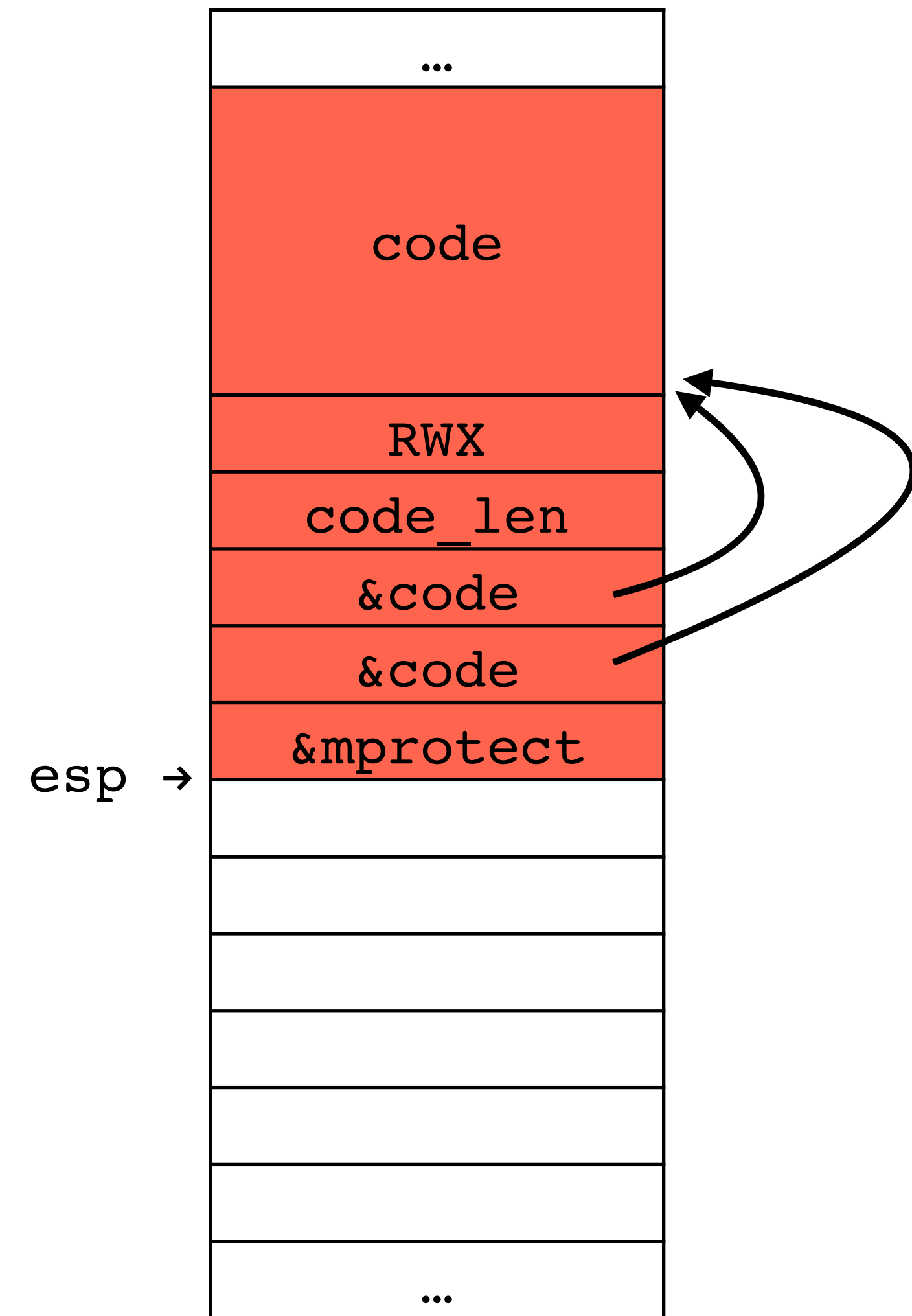


Injecting code

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

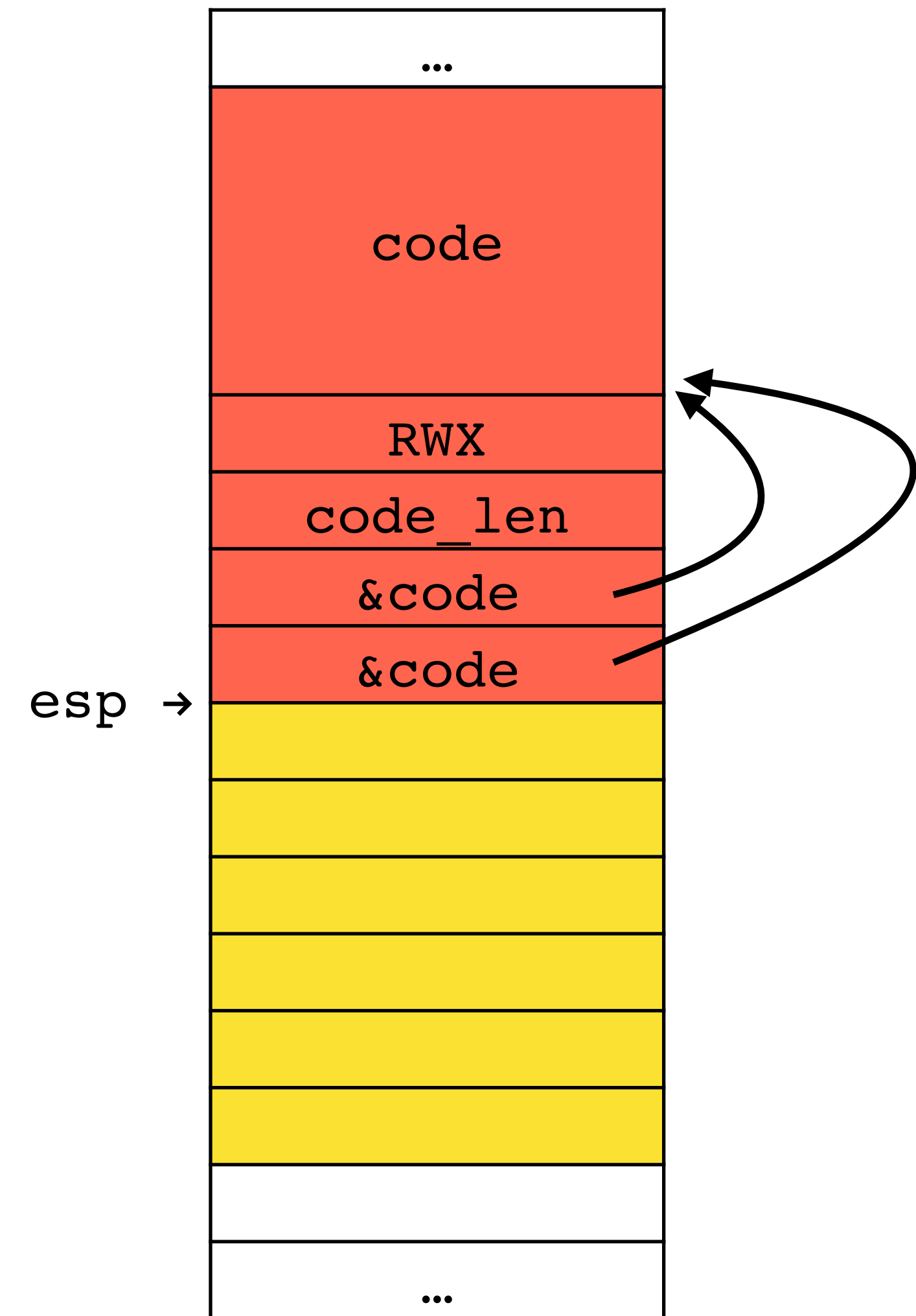
```
int mprotect(void *addr,  
            size_t len,  
            int prot);
```

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



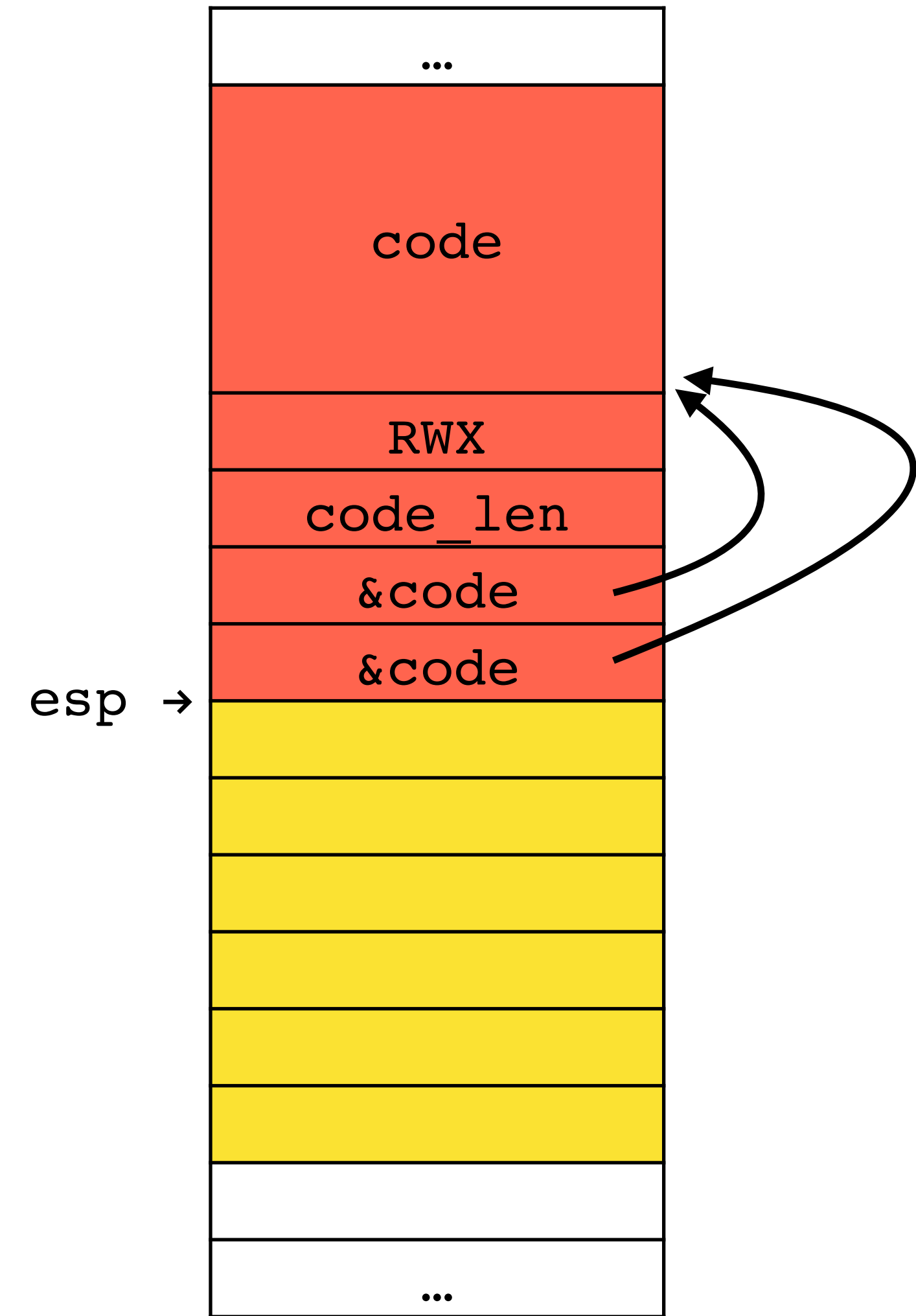
Injecting code

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



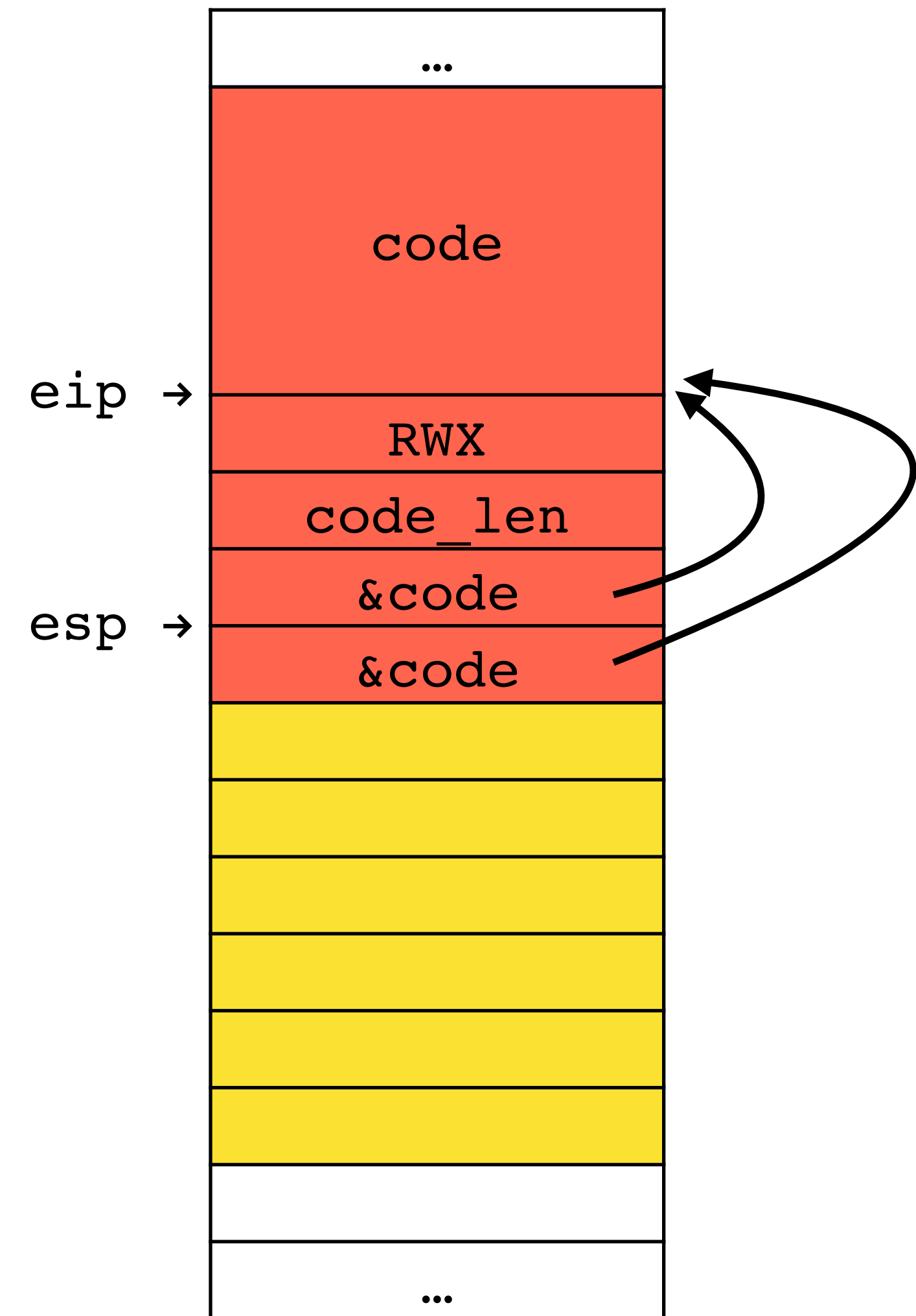
Injecting code

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



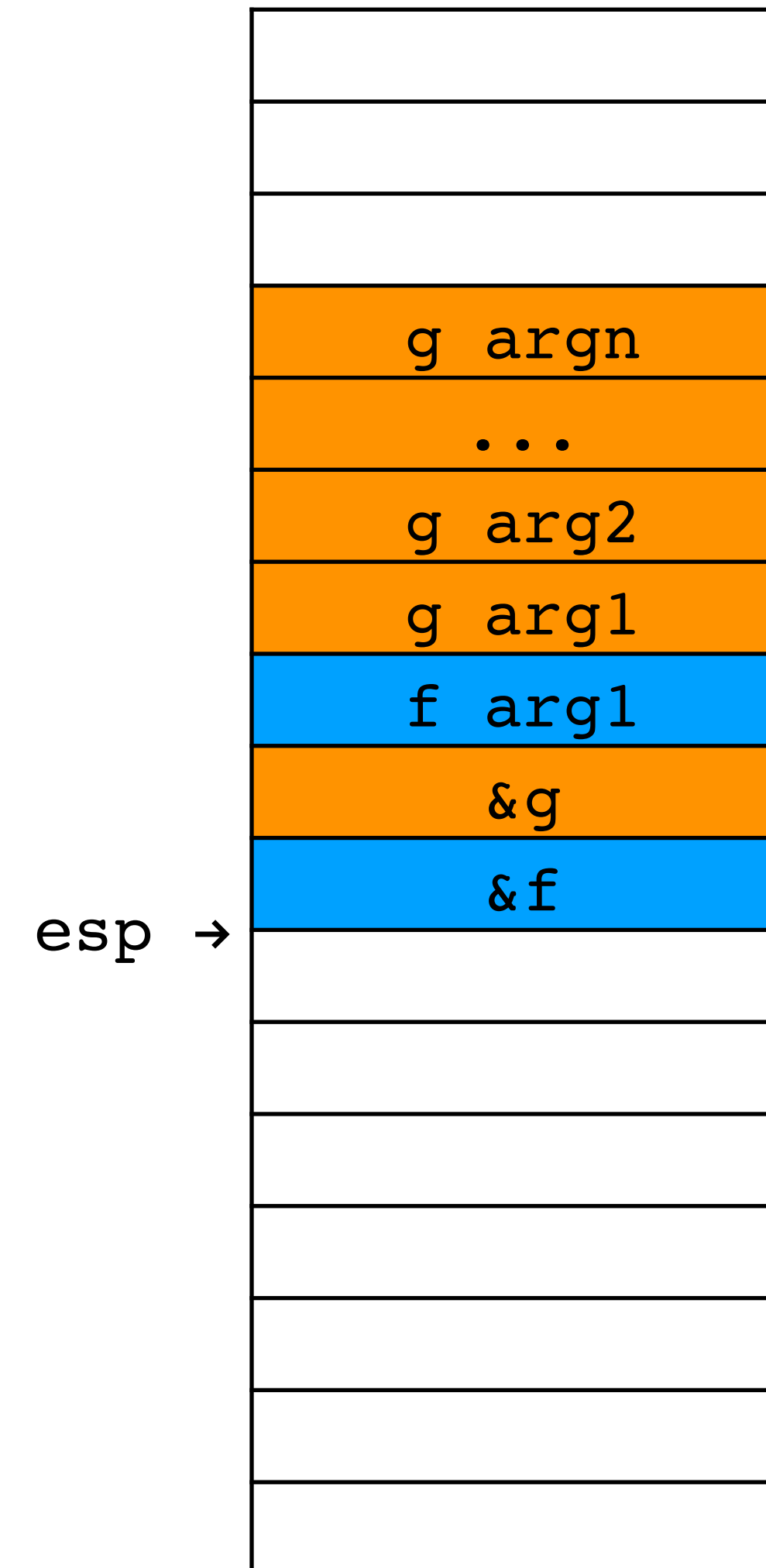
Injecting code

- 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



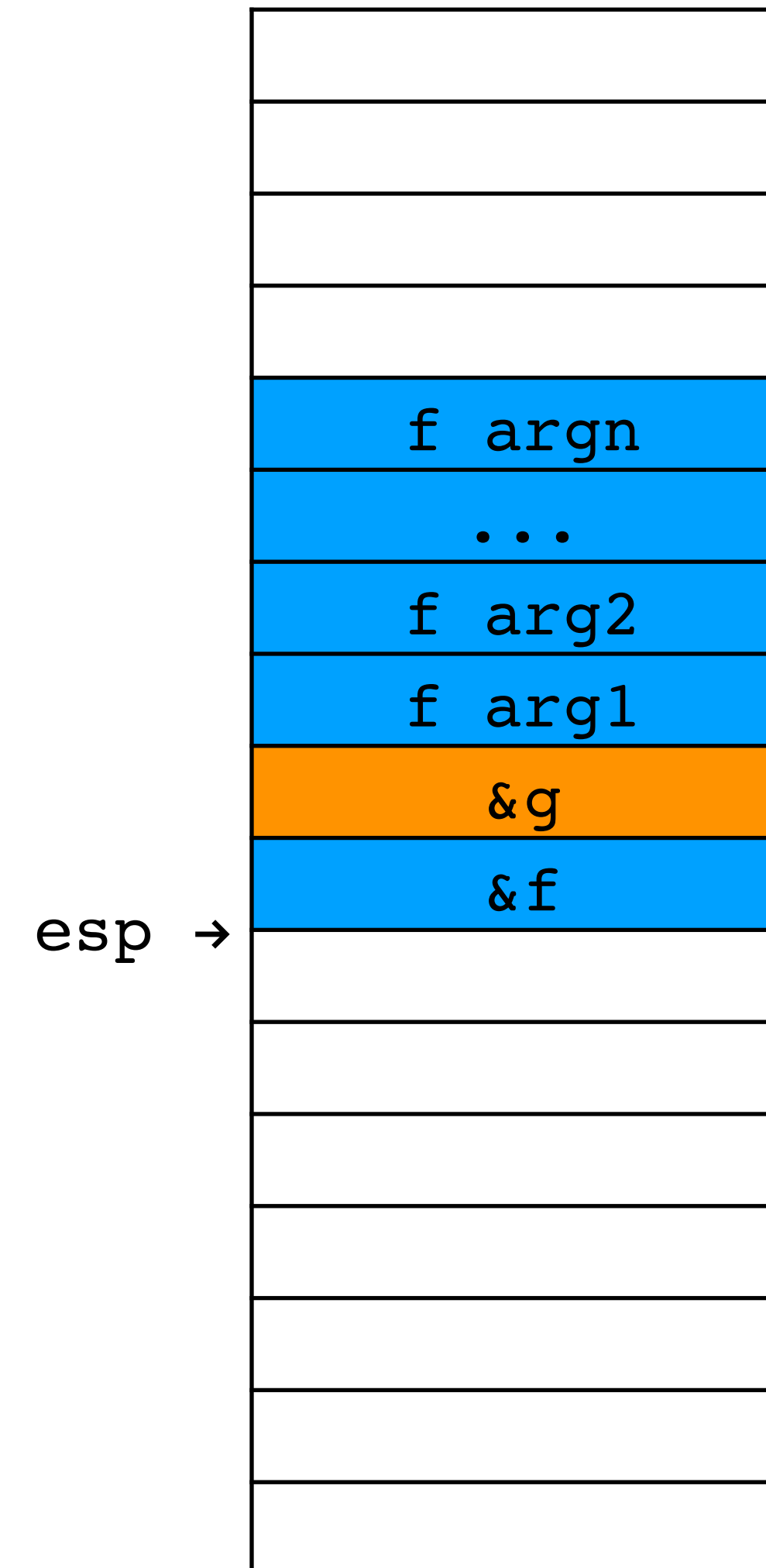
Chaining functions

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



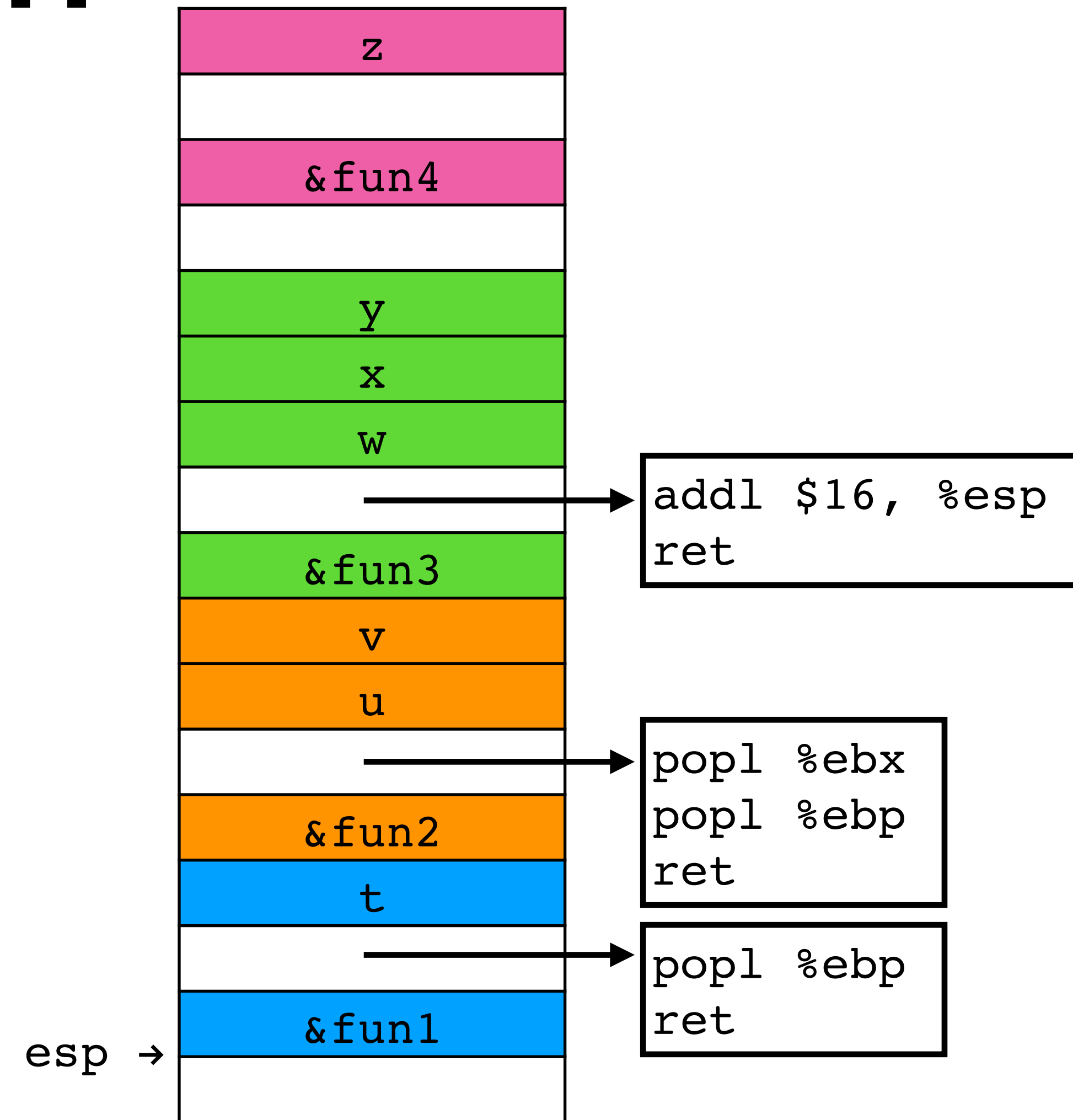
Chaining functions

- 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



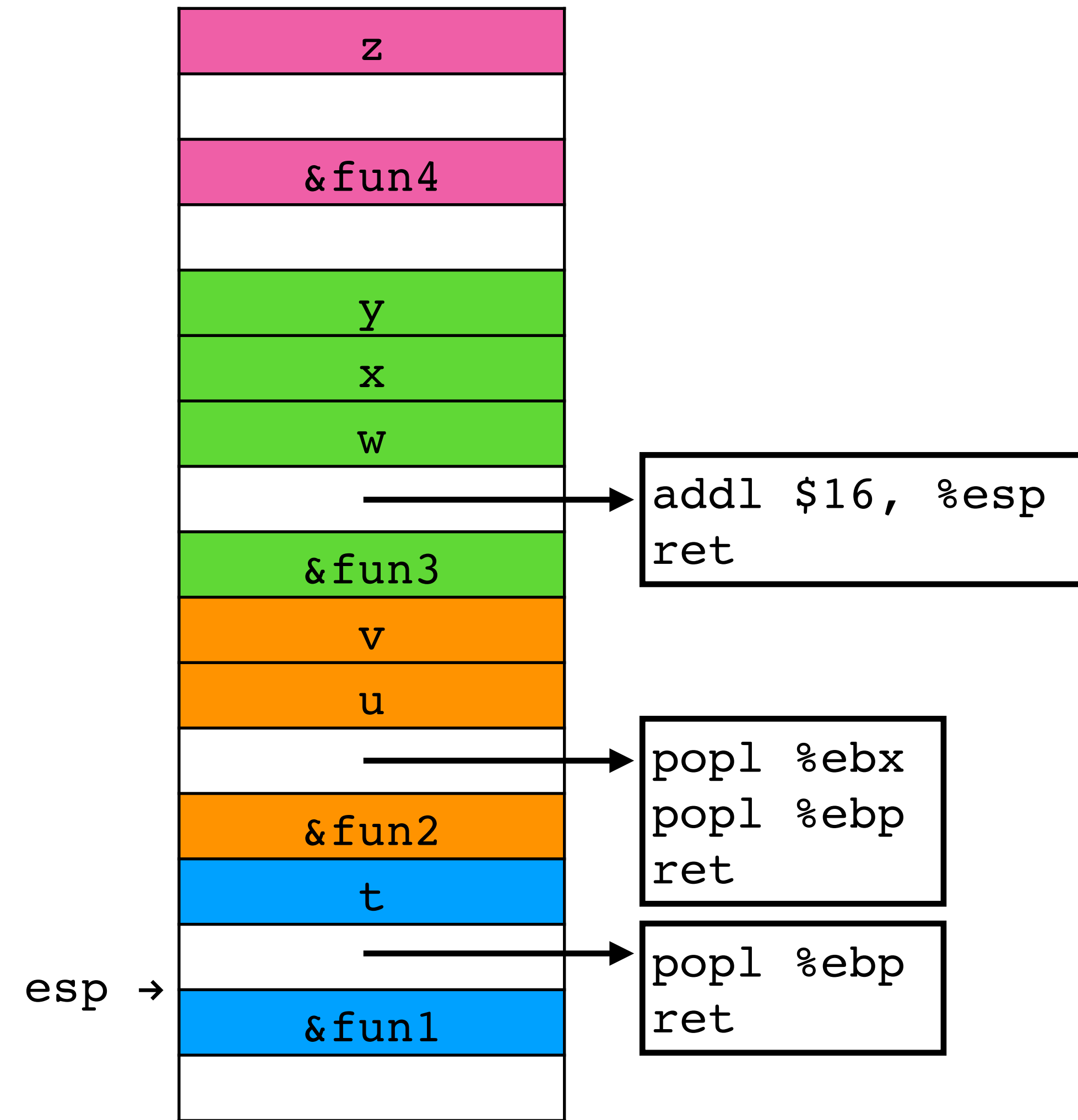
Cleaning up between

- 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 clean up the stack and return to those between function calls
- Examples:
 - `popl %ebp; ret`
 - `popl %ebx; popl %ebp; ret`
 - `addl $16, %esp; ret`



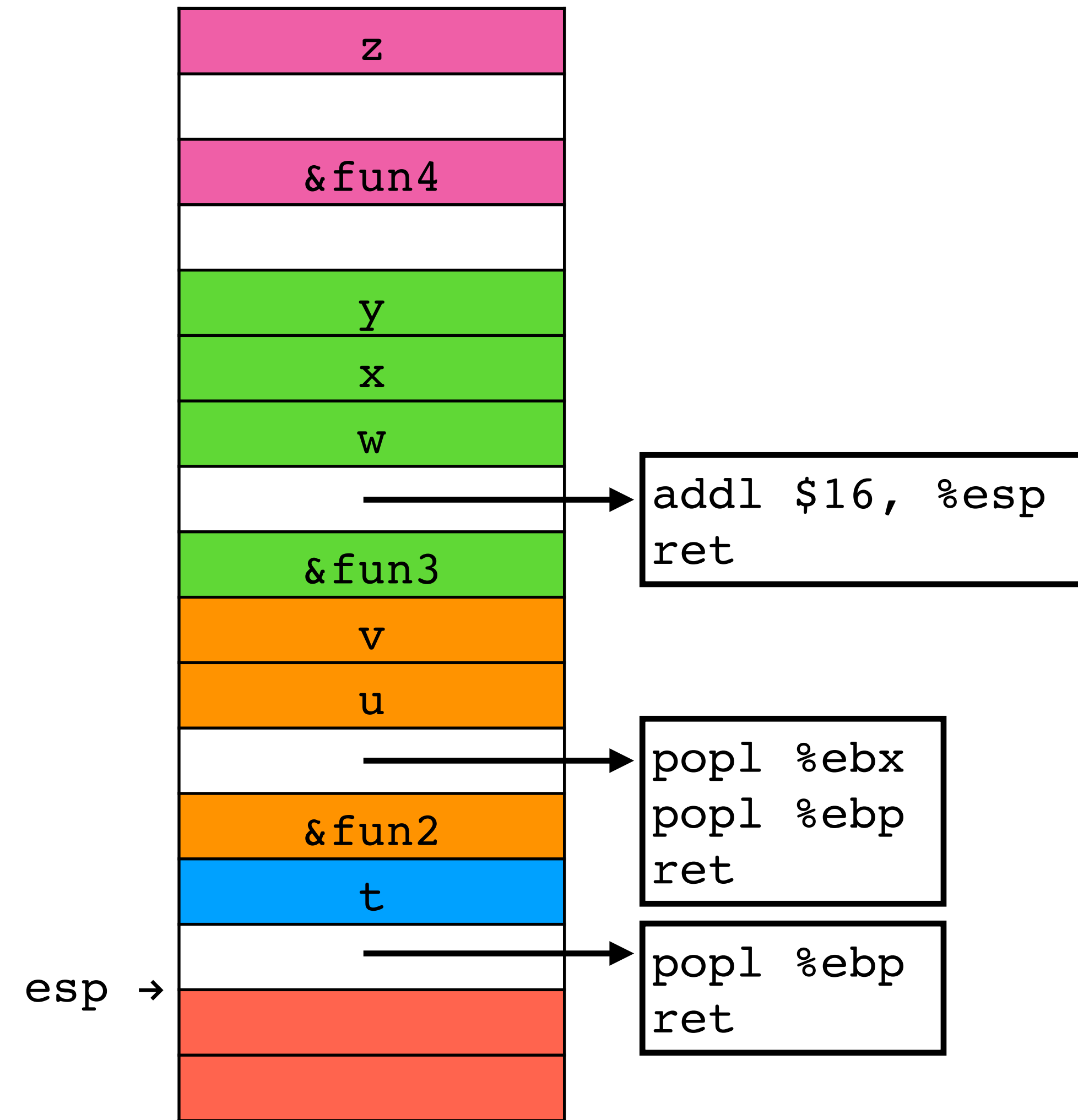
Running

1. Return to fun1



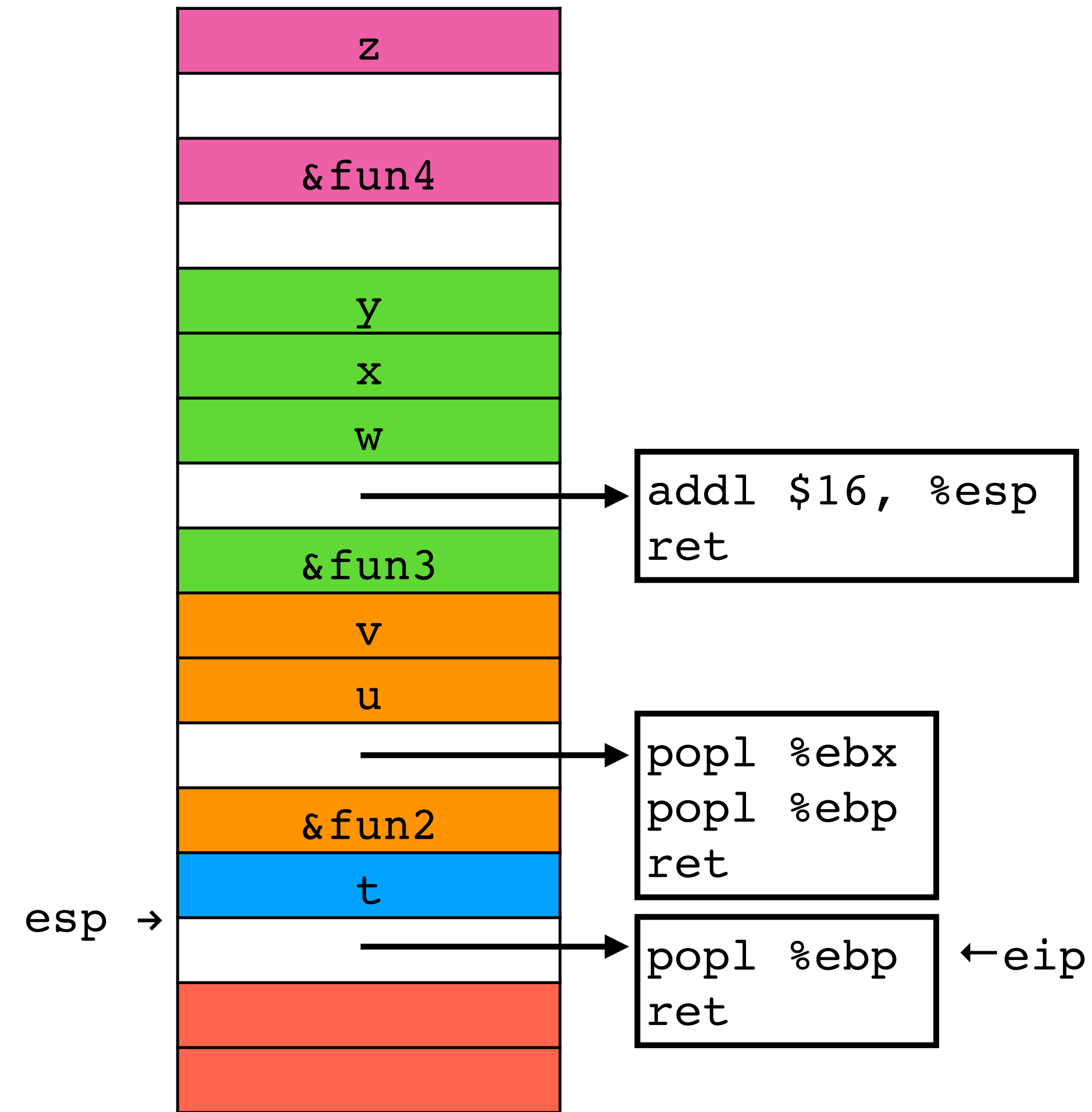
Running

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



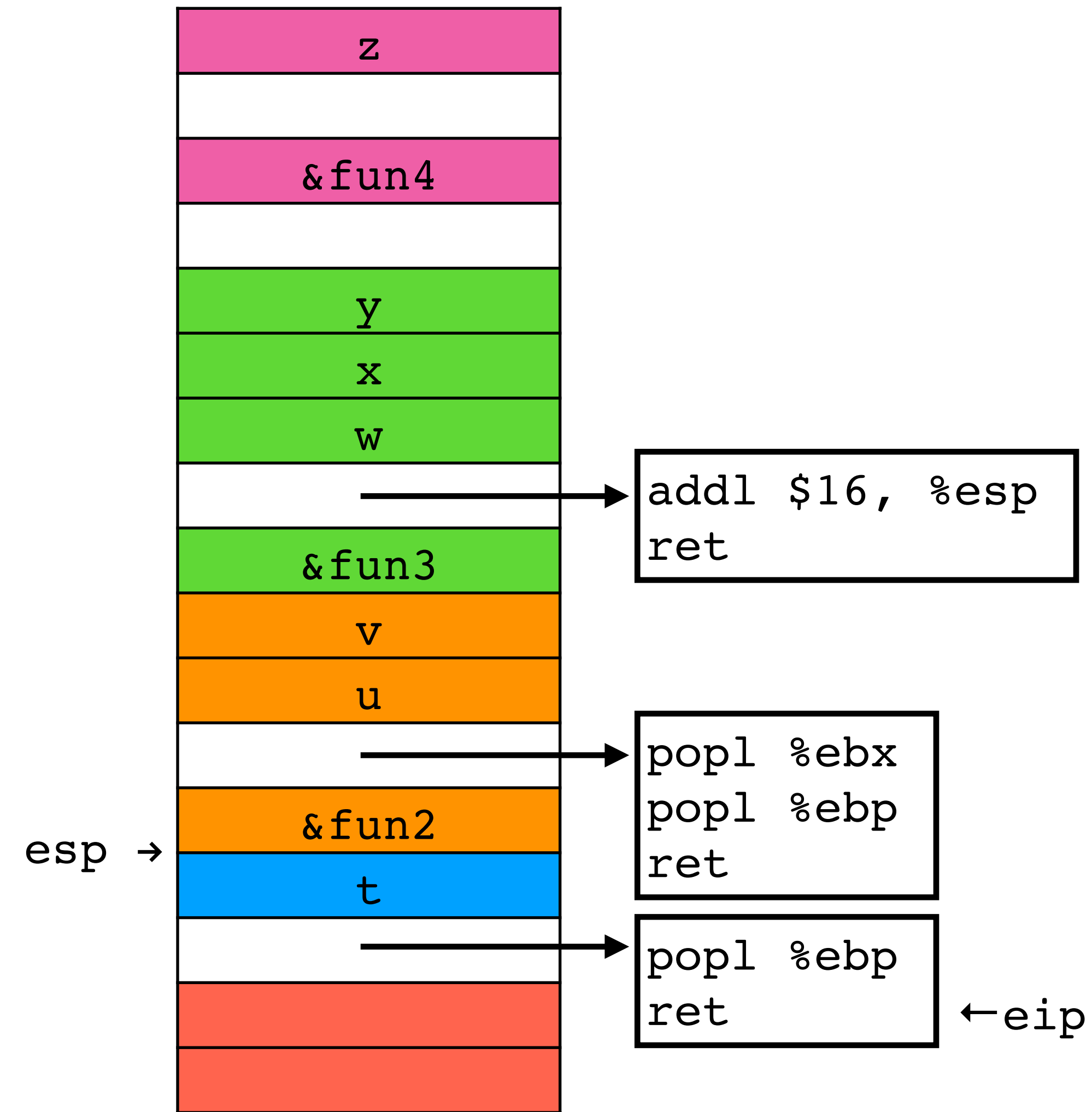
Running

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



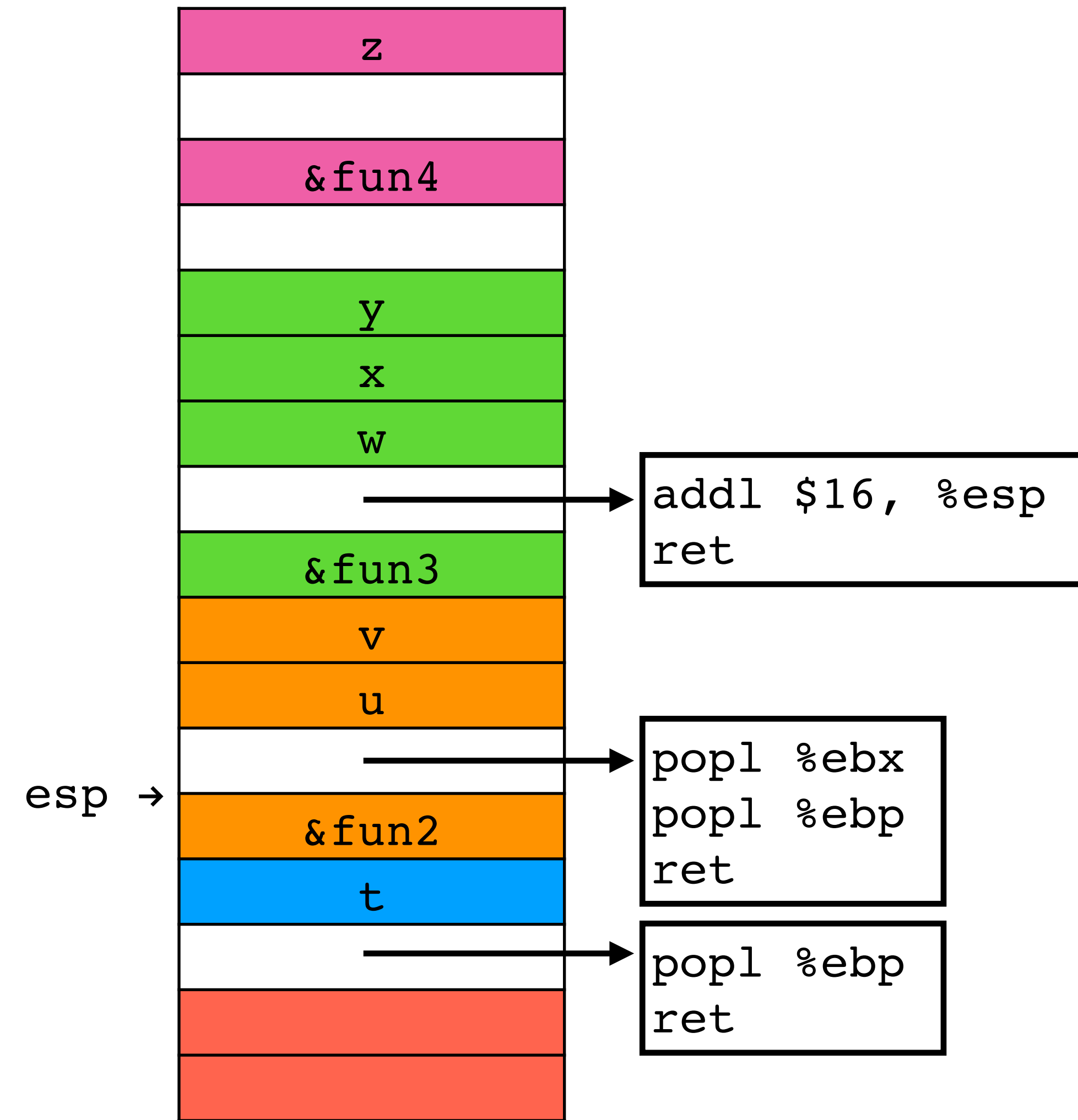
Running

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



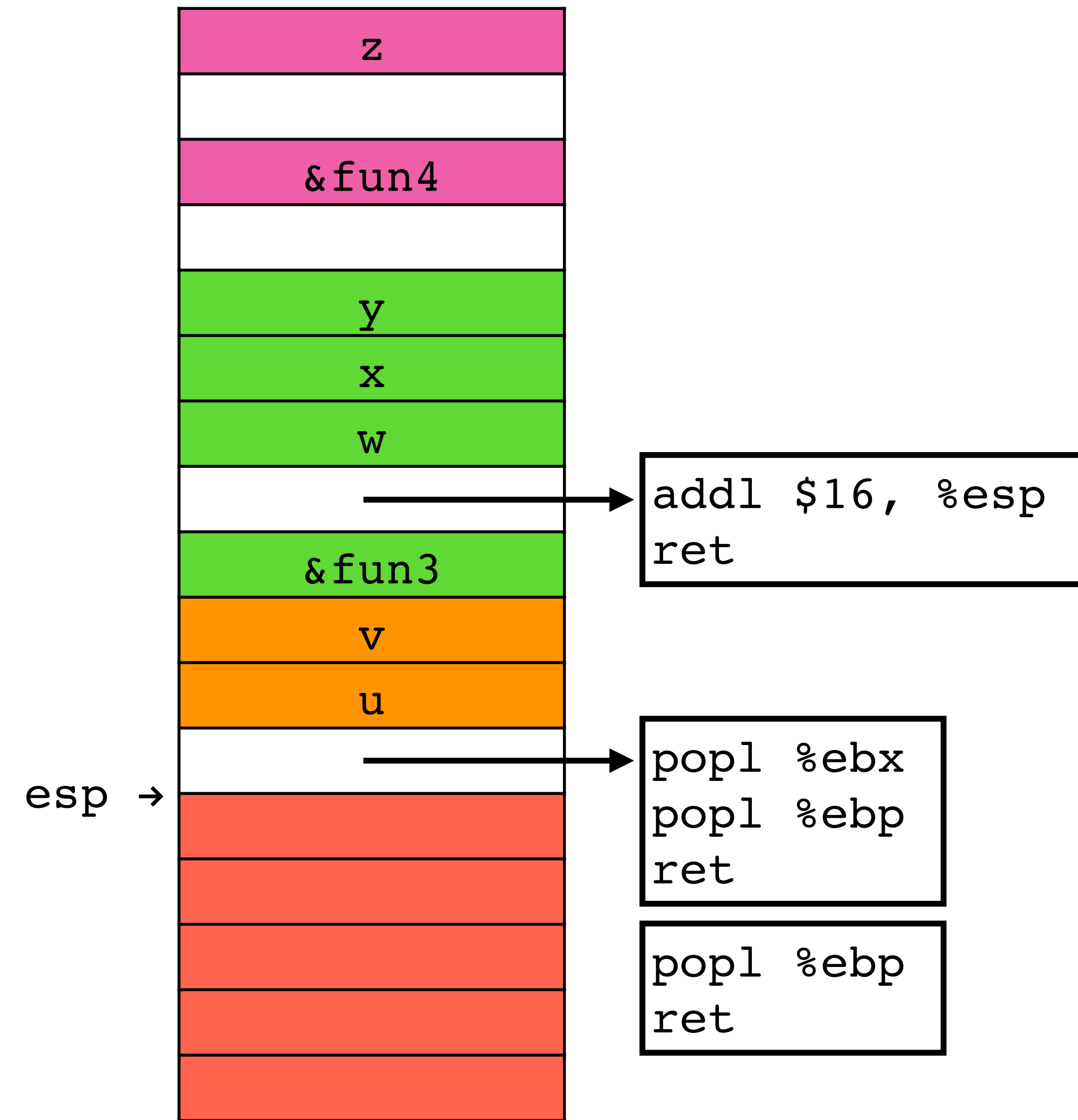
Running

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



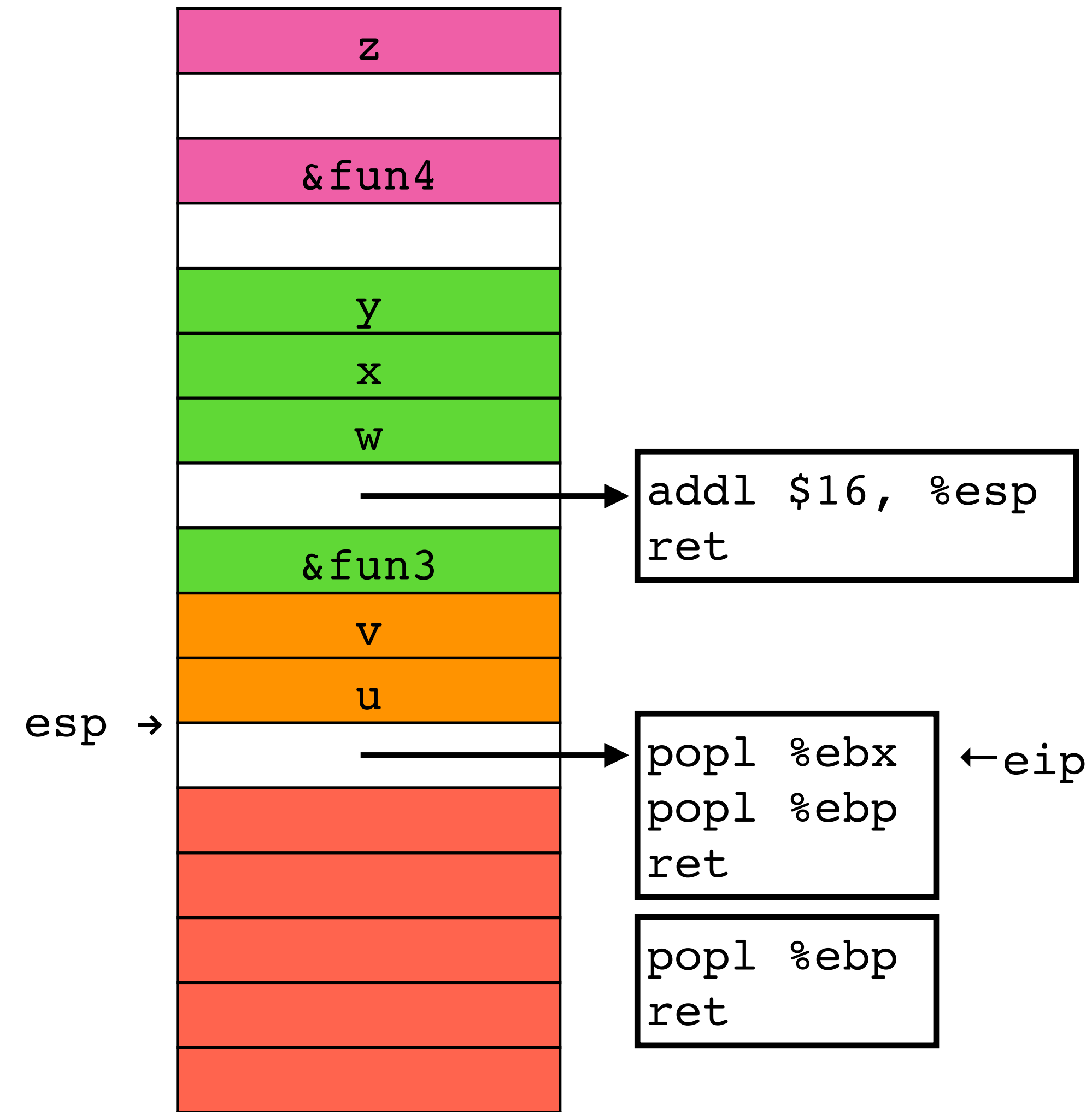
Running

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



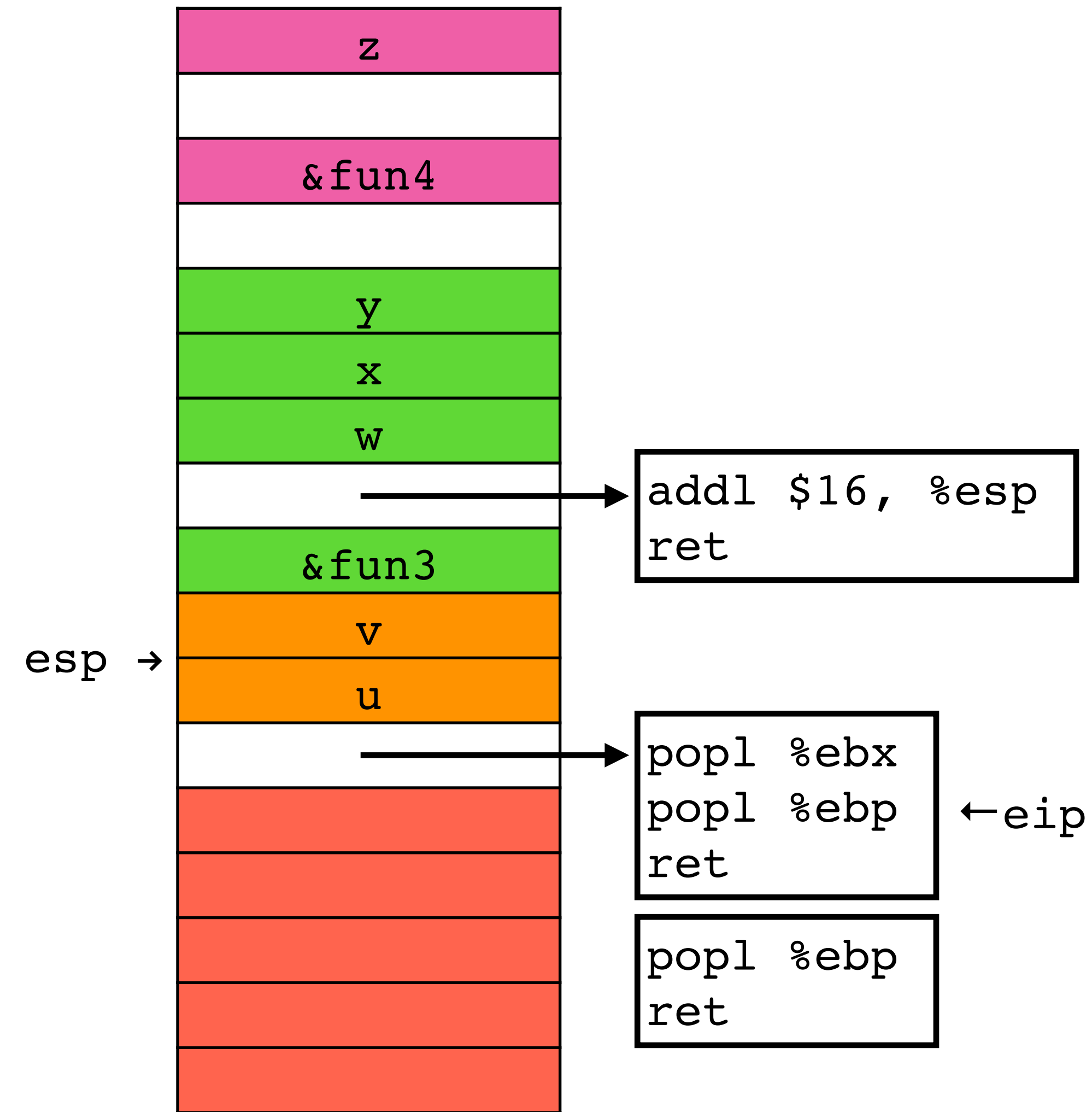
Running

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`



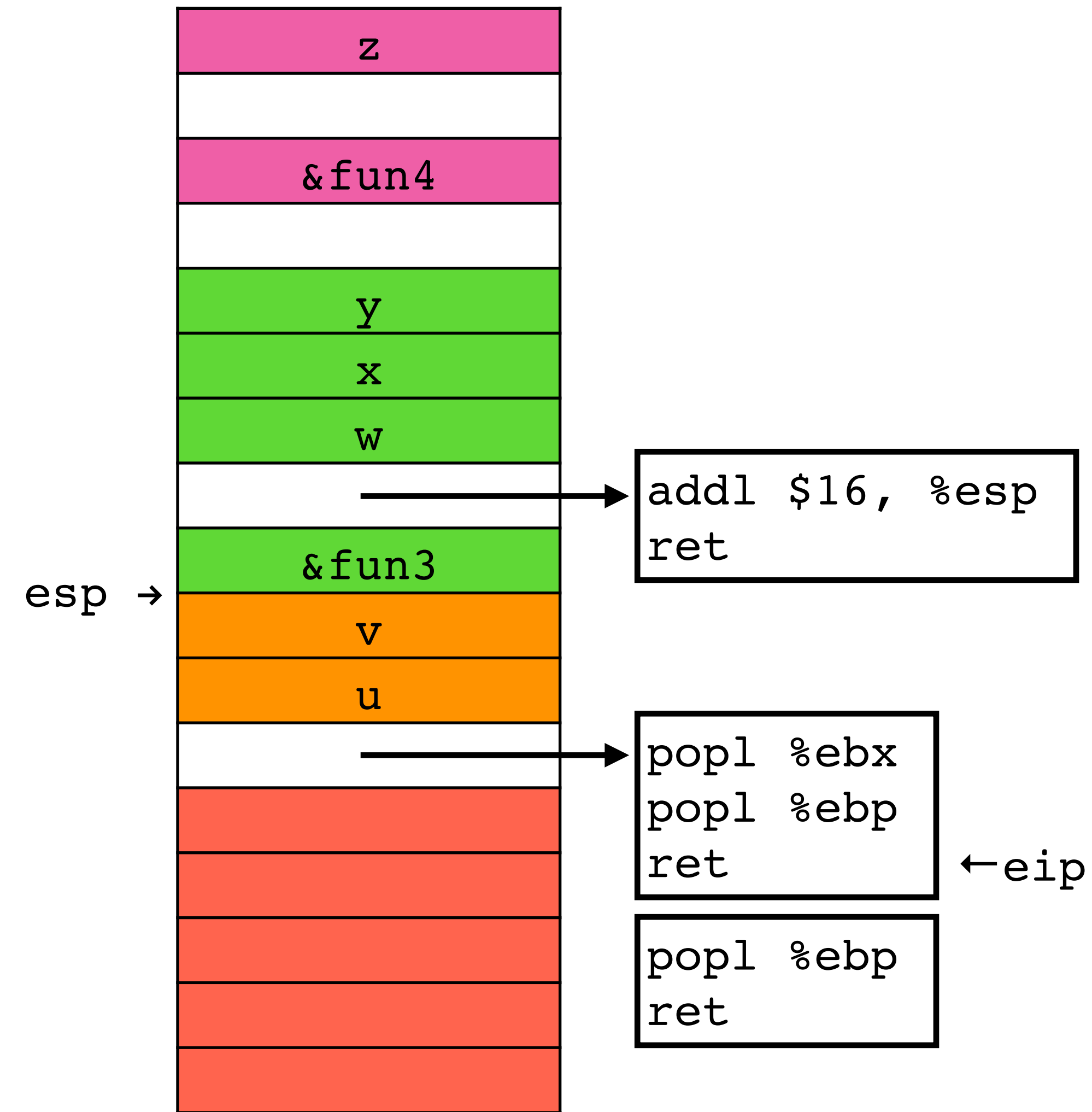
Running

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`



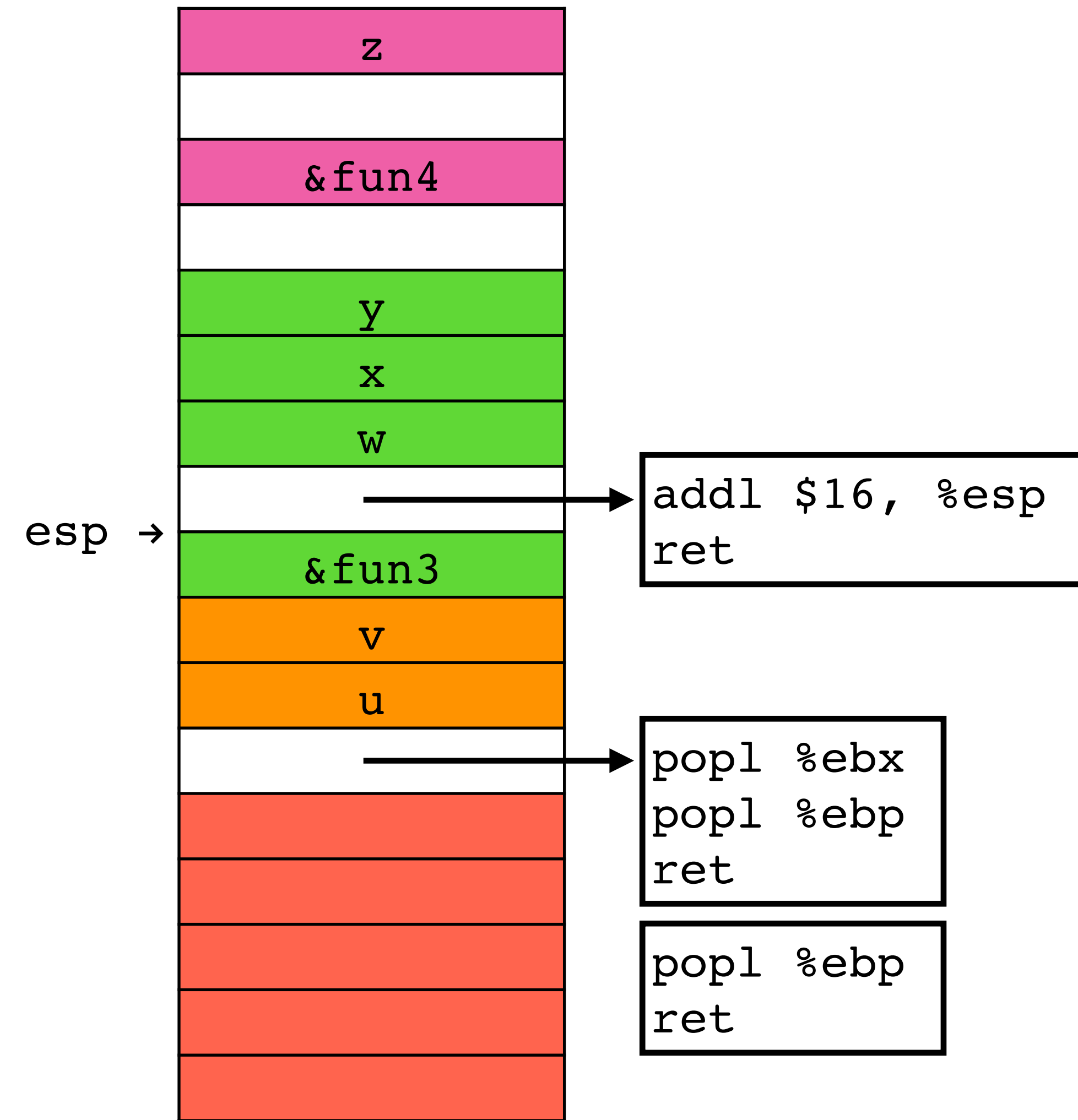
Running

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`



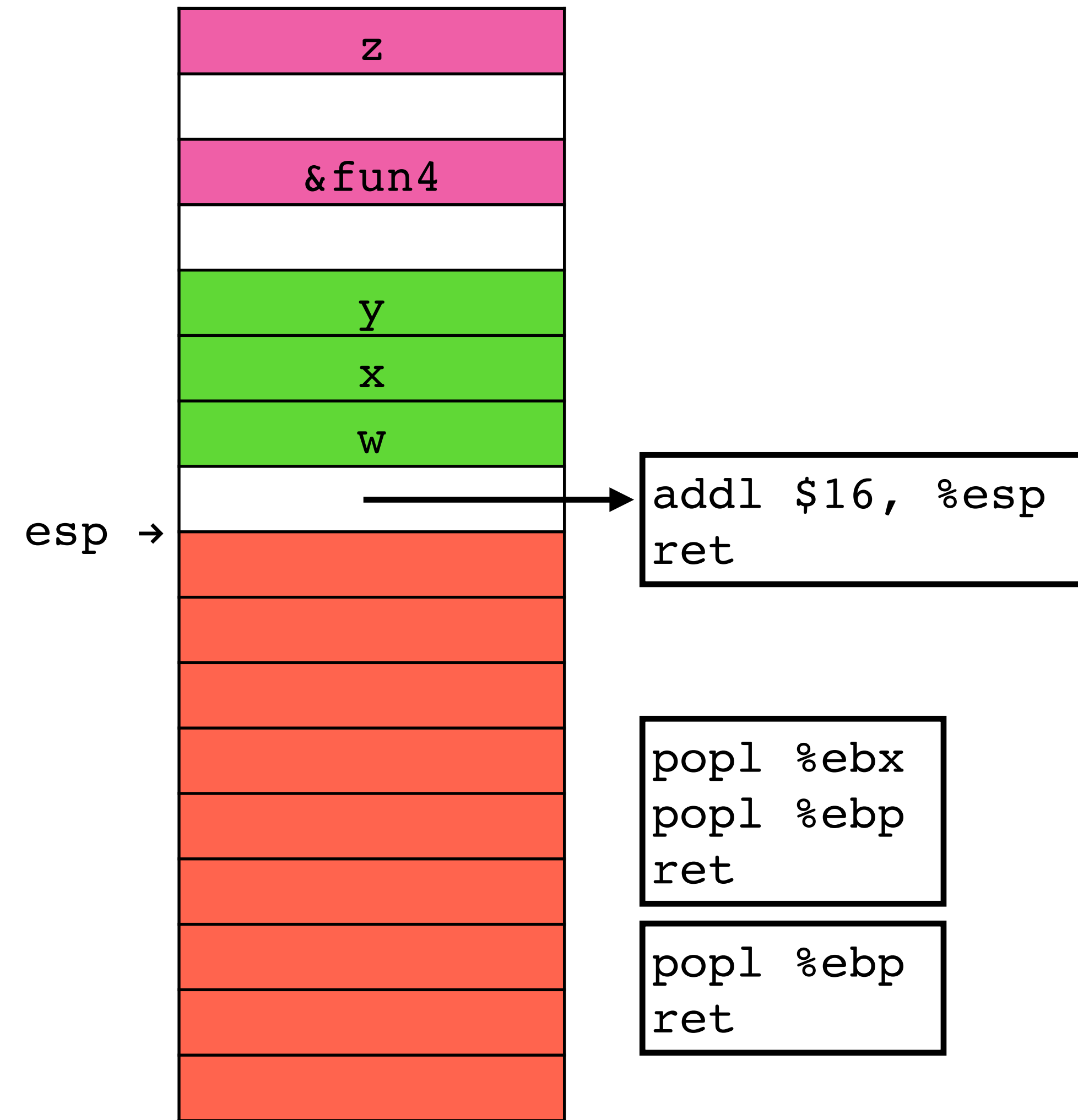
Running

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`



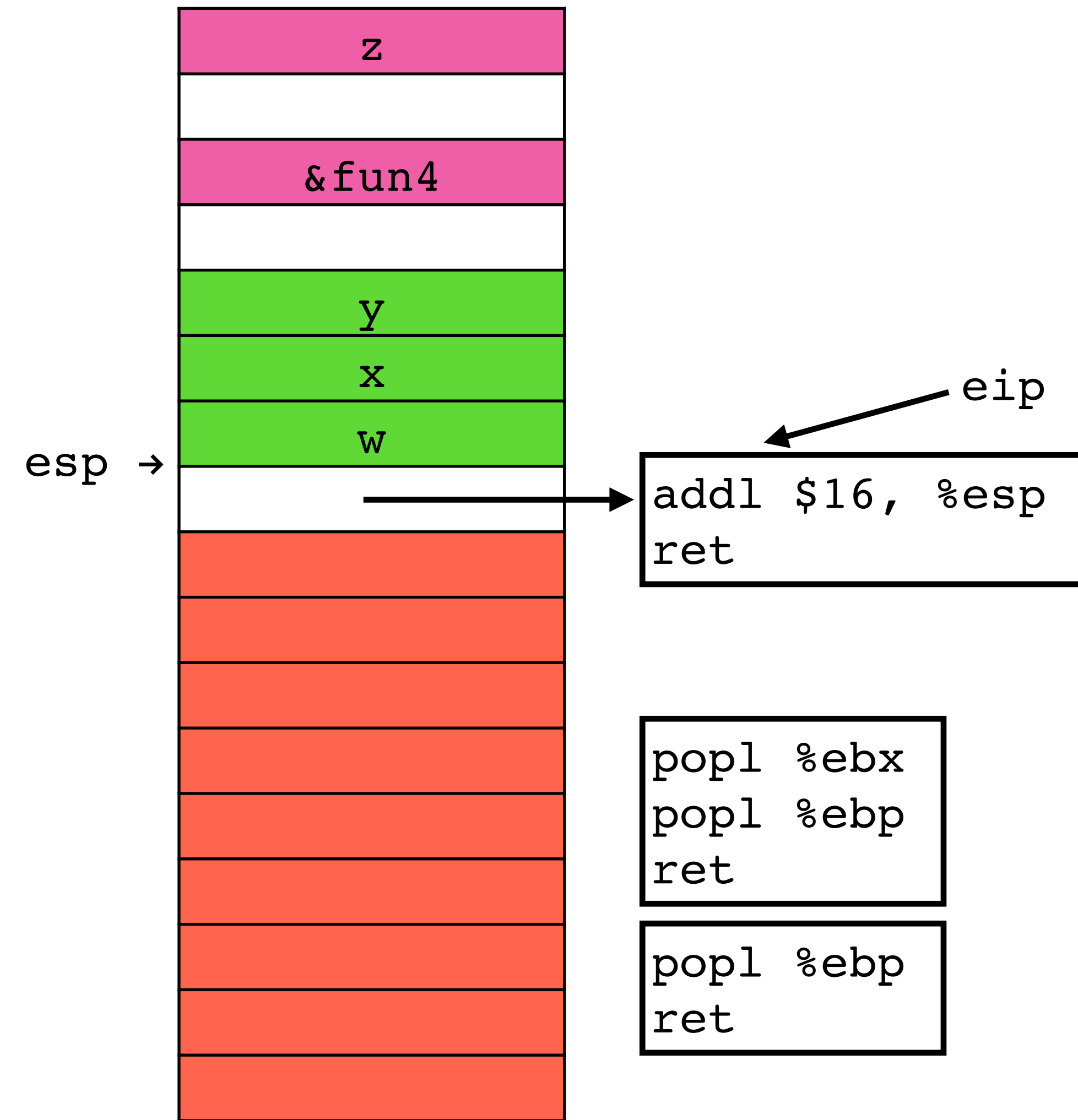
Running

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



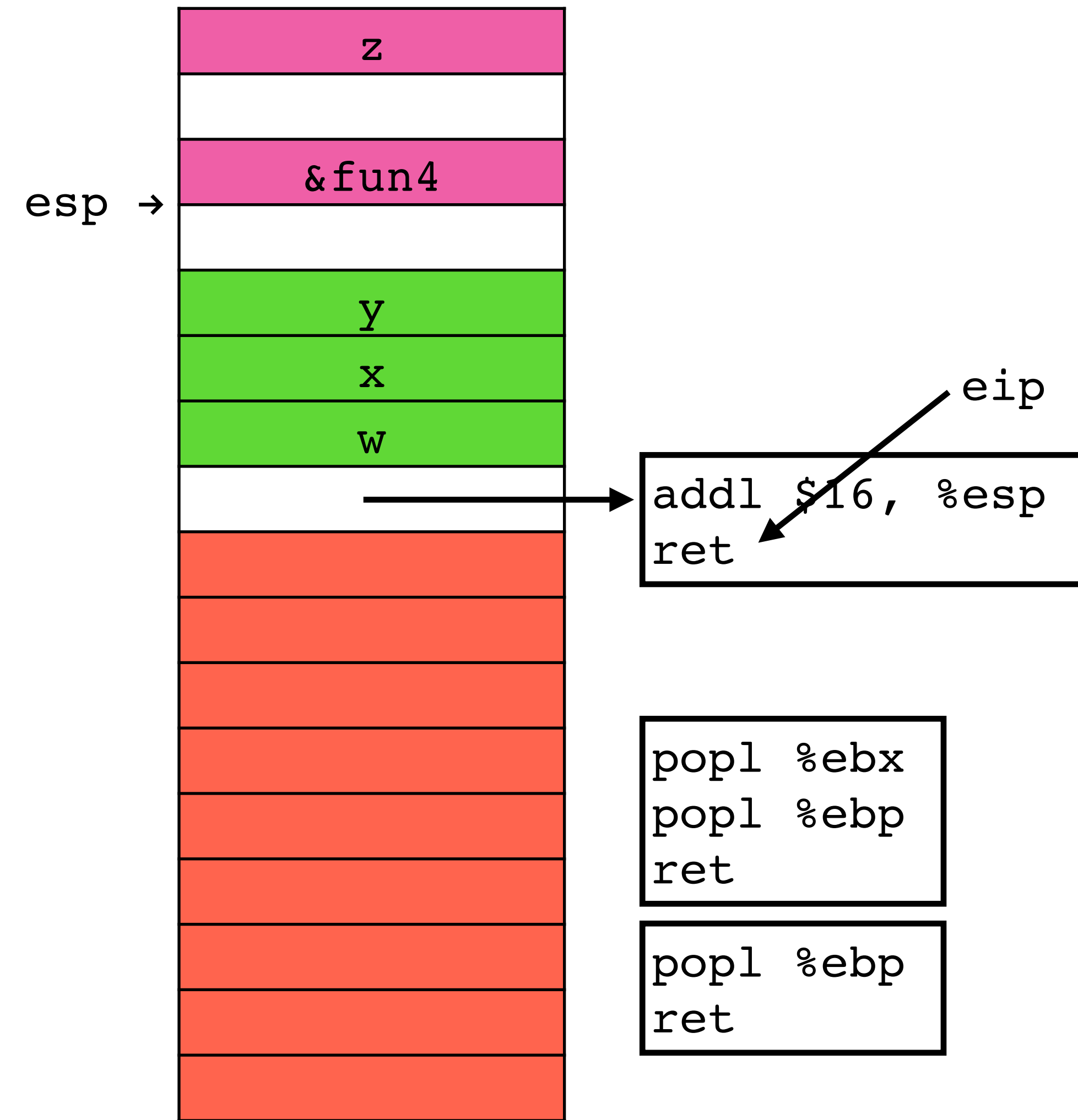
Running

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`



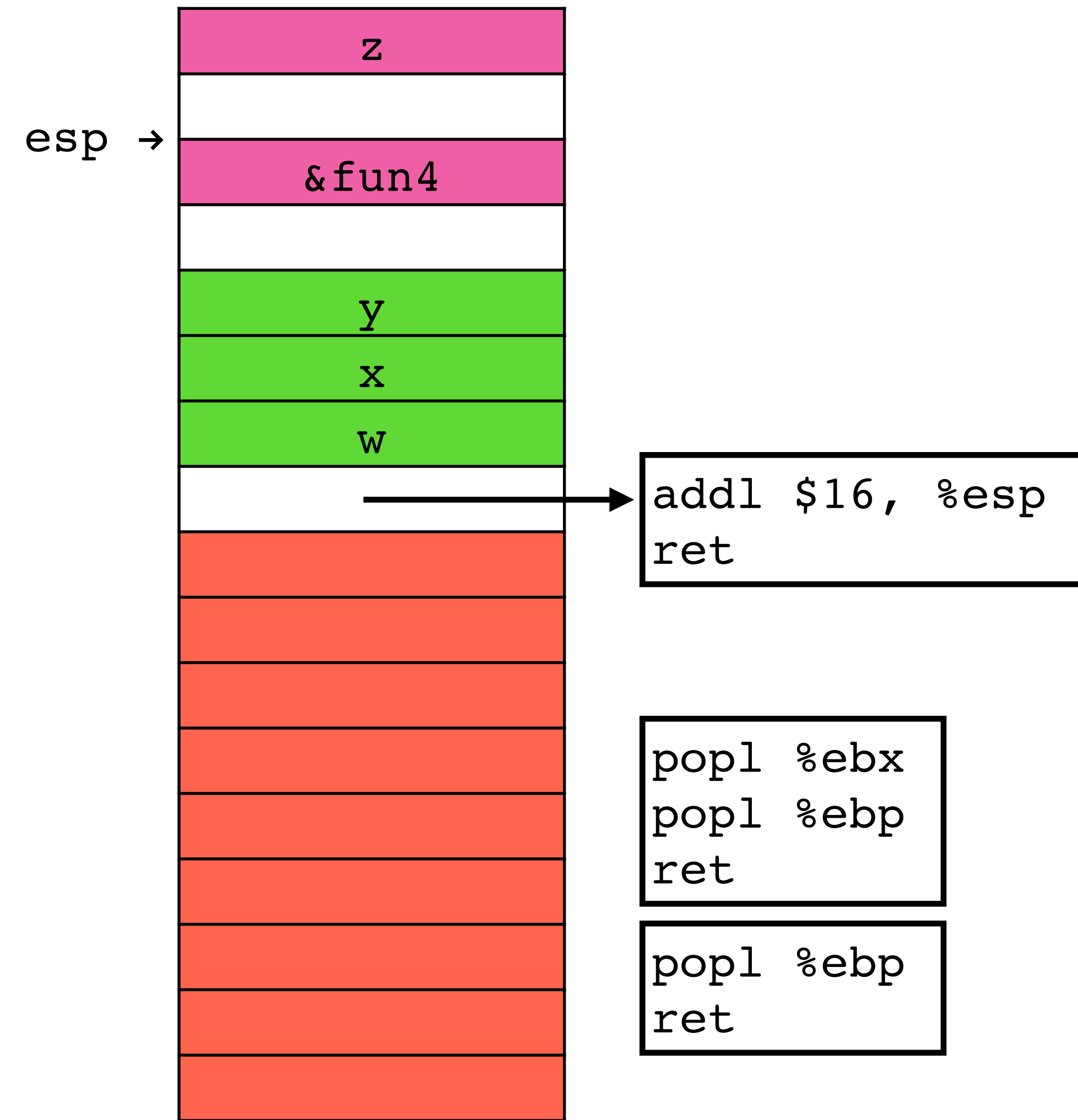
Running

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`



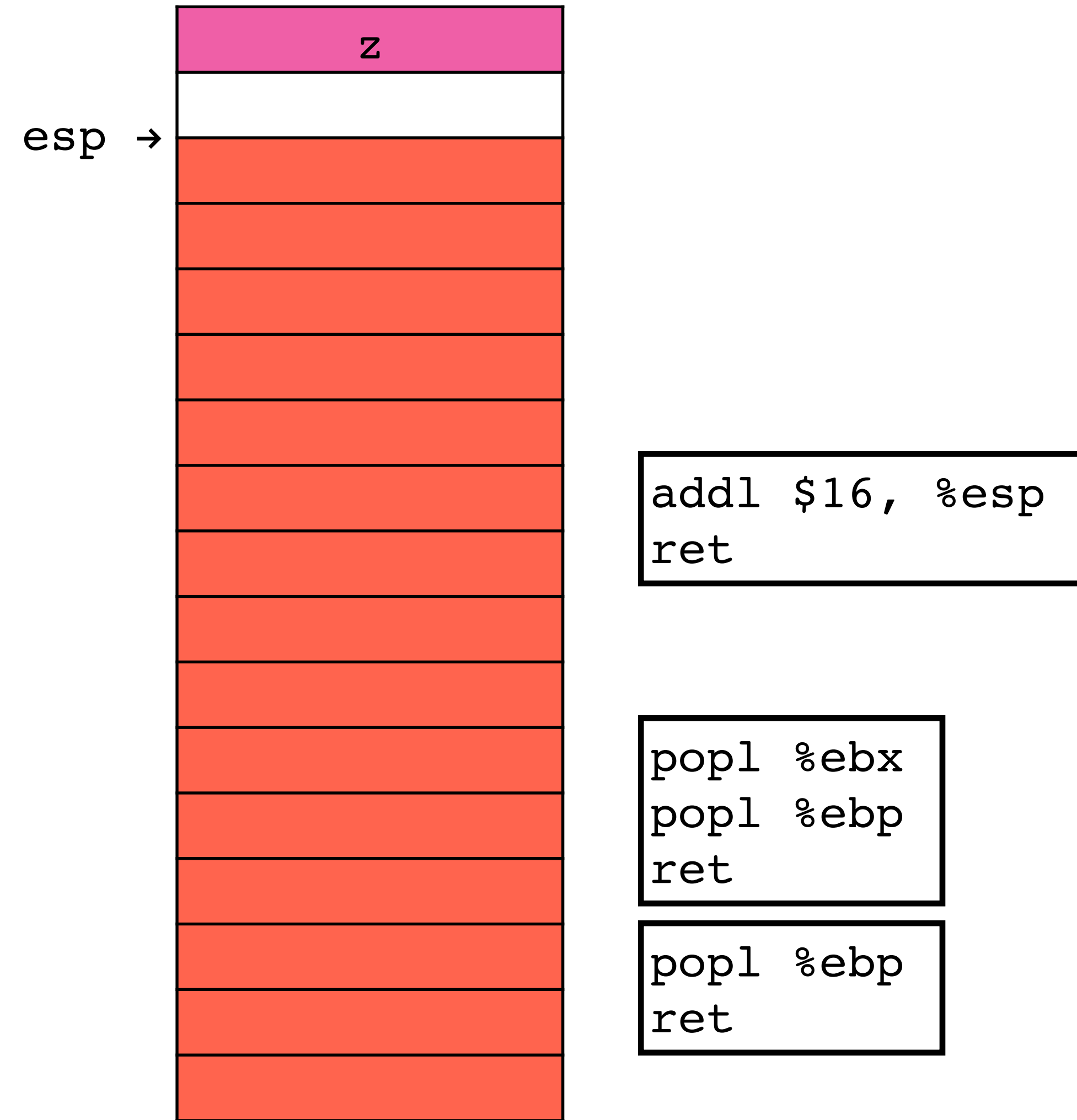
Running

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`



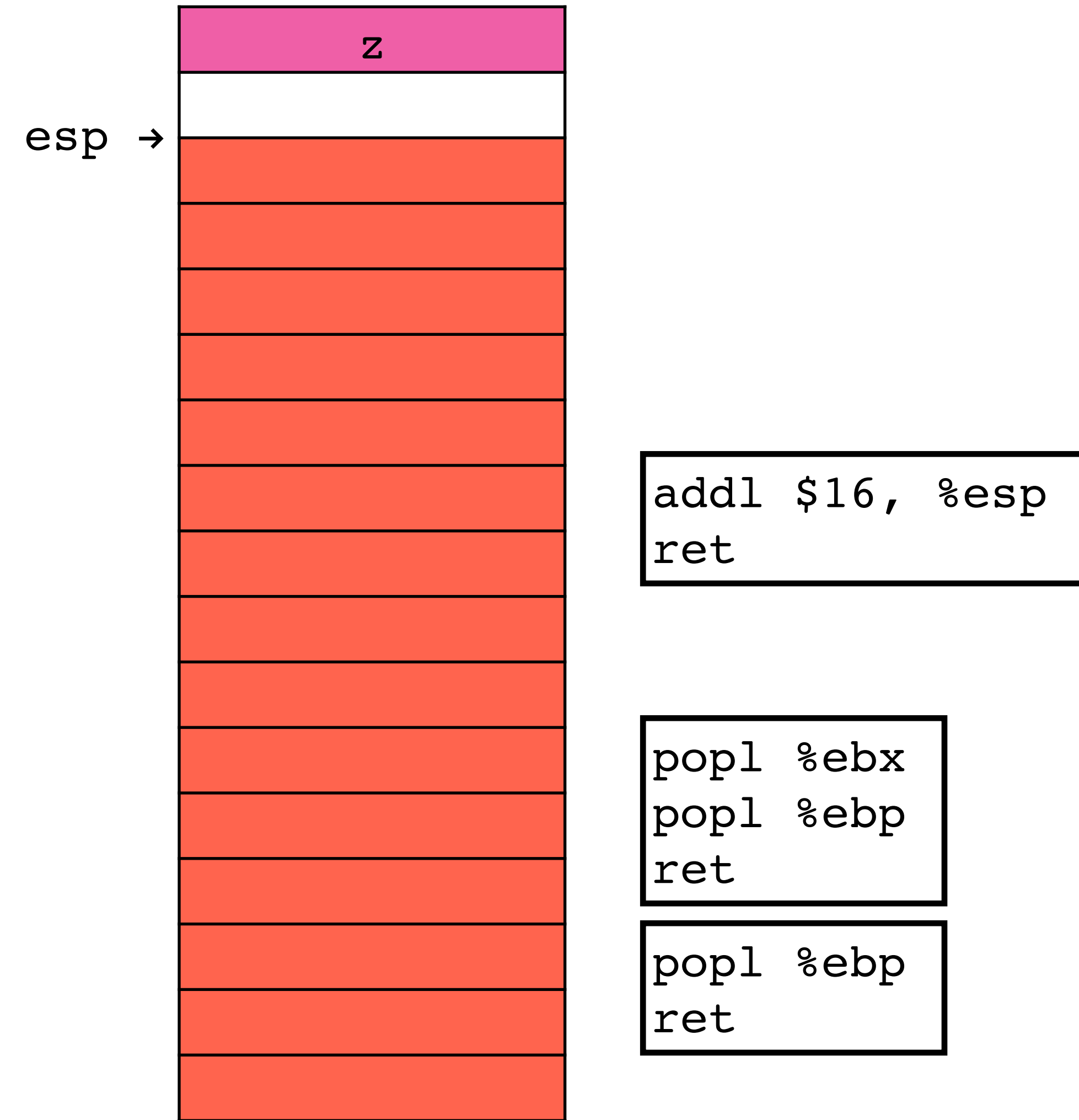
Running

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



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



Cleanup 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 `xchgl %eax, %esp; ret`)
 - Depends on what cleanup code we can find in the program/libraries (turns out there's a whole lot there)

Next time

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