Lecture 14 – Return-oriented programming

Stephen Checkoway

Oberlin College

Based on slides by Bailey, Brumley, and Miller

ROP Overview

- Idea: We forge shellcode out of existing application logic gadgets
- Requirements:
  vulnerability + gadgets + some unrandomized code
- History:
  - No code randomized: Code injection
  - DEP enabled by default: ROP attacks using libc gadgets published 2007
  - ROP assemblers, compilers, shellcode generators
  - ASLR library load points: ROP attacks use .text segment gadgets
  - Today: all major OSes/compilers support position-independent executables

Return-Oriented
Programming

is a lot like a ransom
note, but instead of cutting
out letters from magazines,
you are cutting out
instructions from .text
segments

Image by Dino Dai Zovi

ROP Programming

1. Disassemble code (library or program)
2. Identify *useful* code sequences (usually ending in ret)
3. Assemble the useful sequences into reusable *gadgets**
4. Assemble gadgets into desired shellcode

* Forming gadgets is mostly useful when constructing complicated return-oriented shellcode by hand

A note on terminology

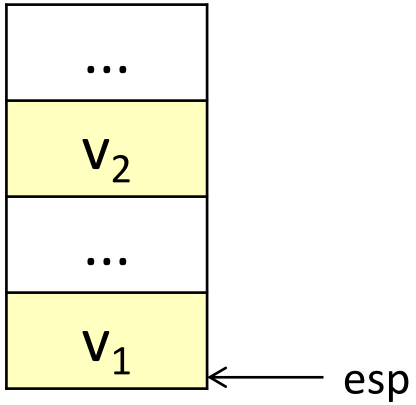
- When ROP was invented in 2007
  - Sequences of code ending in ret were the basic building blocks
  - Multiple sequences and data are assembled into reusable gadgets
- Subsequently
  - A gadget came to refer to any sequence of code ending in a ret
- In 2010
  - ROP without returns (e.g., code sequences ending in call or jmp)

There are many
semantically equivalent
ways to achieve the same
net shellcode effect

Equivalence

`Mem[v2] = v1`

Desired Logic



Stack

a_1: `mov eax, [esp]`

a_2: `mov ebx, [esp+8]`

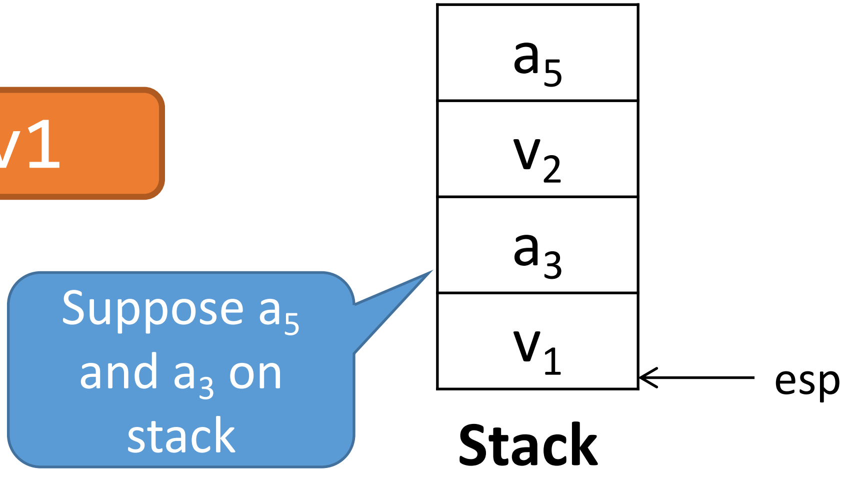
a_3: `mov [ebx], eax`

Implementation 1

Constant store gadget

`Mem[v2] = v1`

Desired Logic



Suppose a_5 and a_3 on stack

a_5

v_2

a_3

v_1

esp

Stack

eax	v_1
ebx	
eip	a_1

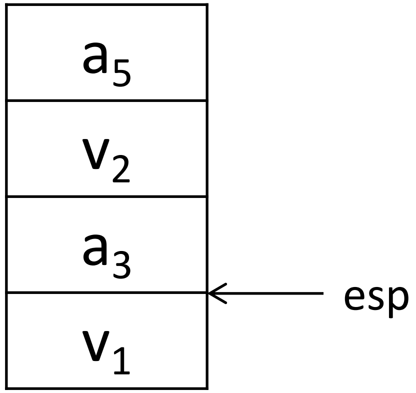
```
a1: pop eax;
a2: ret
a3: pop ebx;
a4: ret
a5: mov [ebx], eax
```

Implementation 2

Constant store gadget

Mem[v2] = v1

Desired Logic



a_5

v_2

a_3

esp

v_1

Stack

eax	v_1
ebx	
eip	a_3

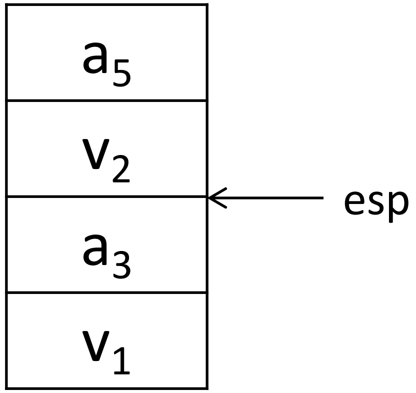
a_1: pop eax;
a_2: ret
a_3: pop ebx;
a_4: ret
a_5: mov [ebx], eax

Implementation 2

Constant store gadget

`Mem[v2] = v1`

Desired Logic



a_5

v_2

a_3

v_1

esp

Stack

eax	v_1
ebx	v_2
eip	a_3

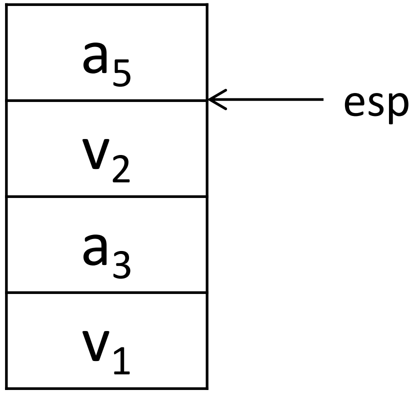
a_1: `pop eax;`
a_2: `ret`
a_3: `pop ebx;`
a_4: `ret`
a_5: `mov [ebx], eax`

Implementation 2

Constant store gadget

Mem[v2] = v1

Desired Logic



Stack

eax	v_1
ebx	v_2
eip	a_5

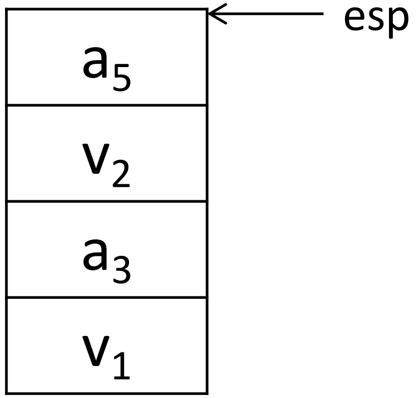
```
a₁: pop eax;
a₂: ret
a₃: pop ebx;
a₄: ret
a₅: mov [ebx], eax
```

Implementation 2

Constant store gadget

`Mem[v2] = v1`

Desired Logic


esp

a_5

v_2

a_3

v_1

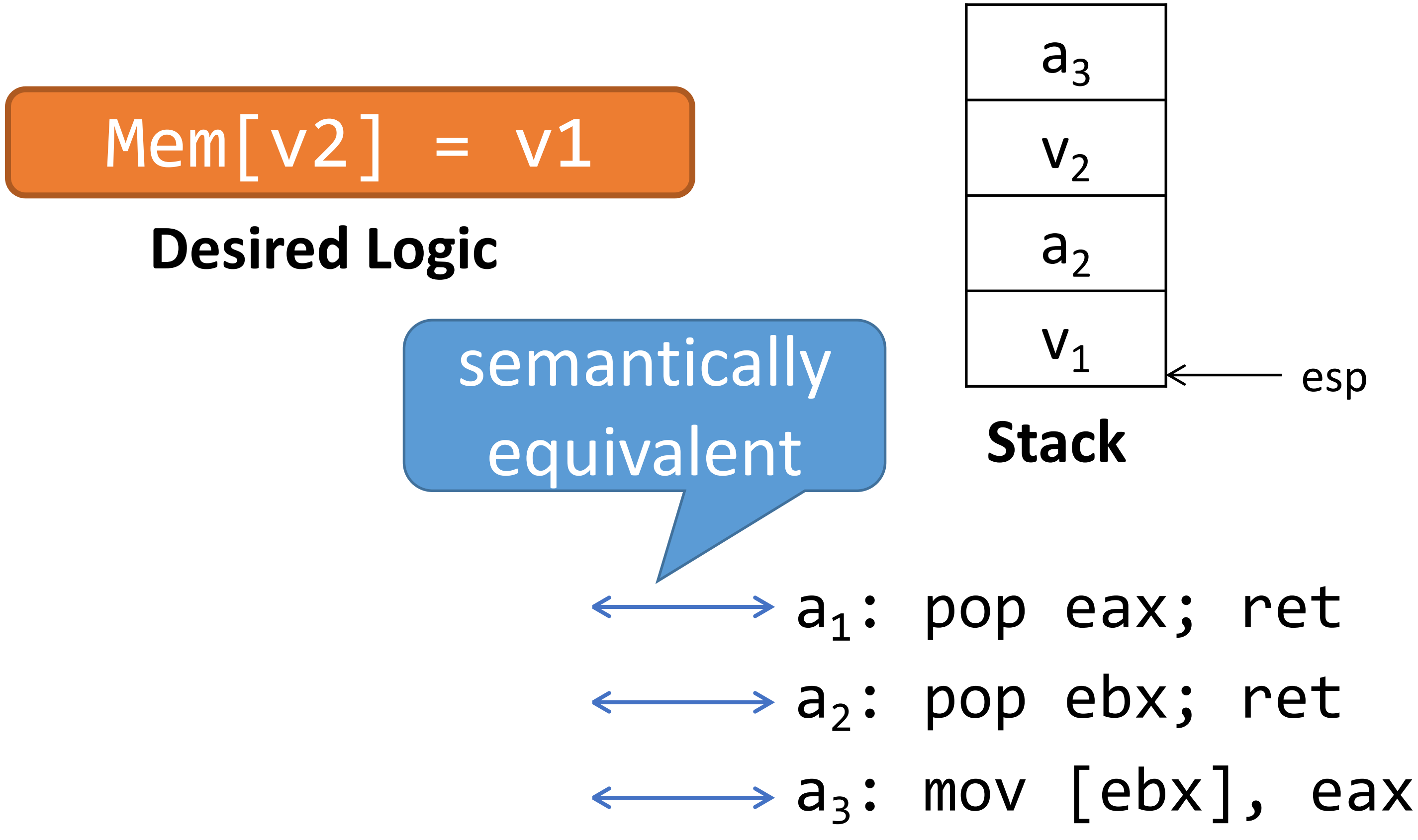
Stack

eax	v_1
ebx	v_2
eip	a_5

a_1: `pop eax;`
a_2: `ret`
a_3: `pop ebx;`
a_4: `ret`
a_5: `mov [ebx], eax`

Implementation 2

Equivalence



`Mem[v2] = v1`

Desired Logic

semantically equivalent

Stack
a_3
v_2
a_2
v_1 ← esp

Stack

a_1: `pop eax; ret`

a_2: `pop ebx; ret`

a_3: `mov [ebx], eax`

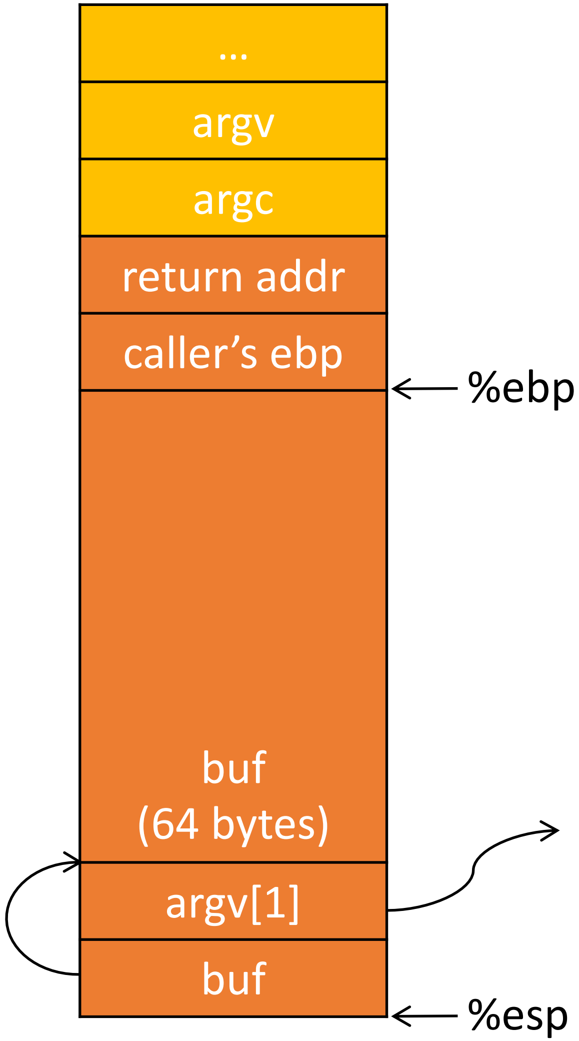
Implementation 2

Return-Oriented Programming

```
Mem[v2] = v1
```

Desired *Shellcode*

- Find needed instruction gadgets at addresses a_1, a_2, and a_3 in *existing* code
- Overwrite stack to execute a_1, a_2, and then a_3



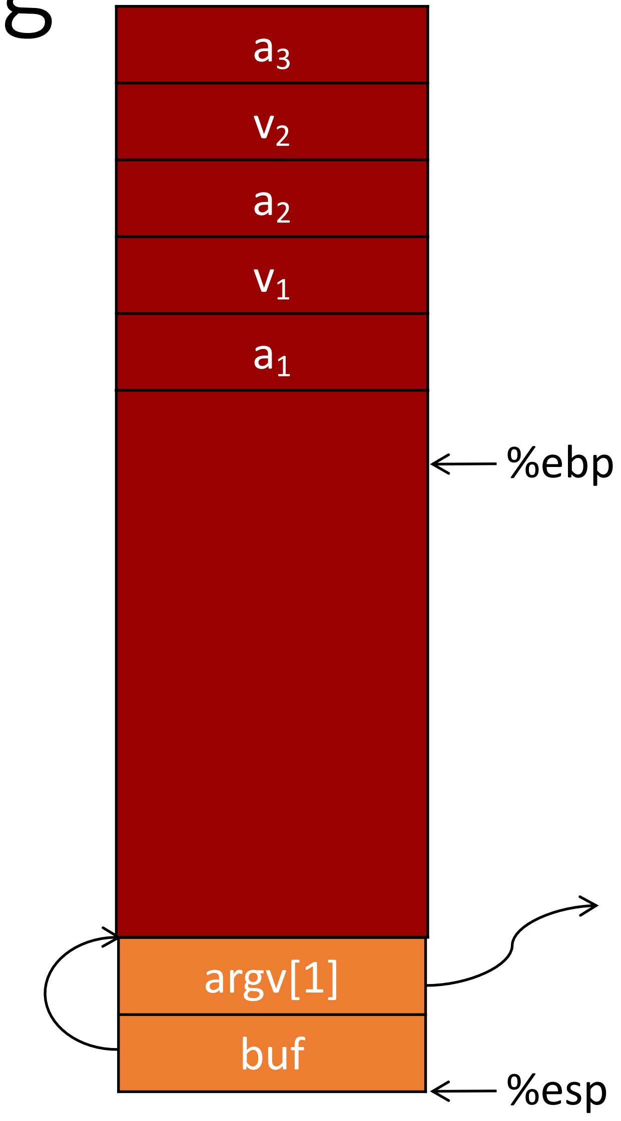
Return-Oriented Programming

`Mem[v2] = v1`

Desired *Shellcode*

```
a₁: pop eax; ret
a₂: pop ebx; ret
a₃: mov [ebx], eax
```

Desired store executed!



Arithmetic/logical operations: c = x op y

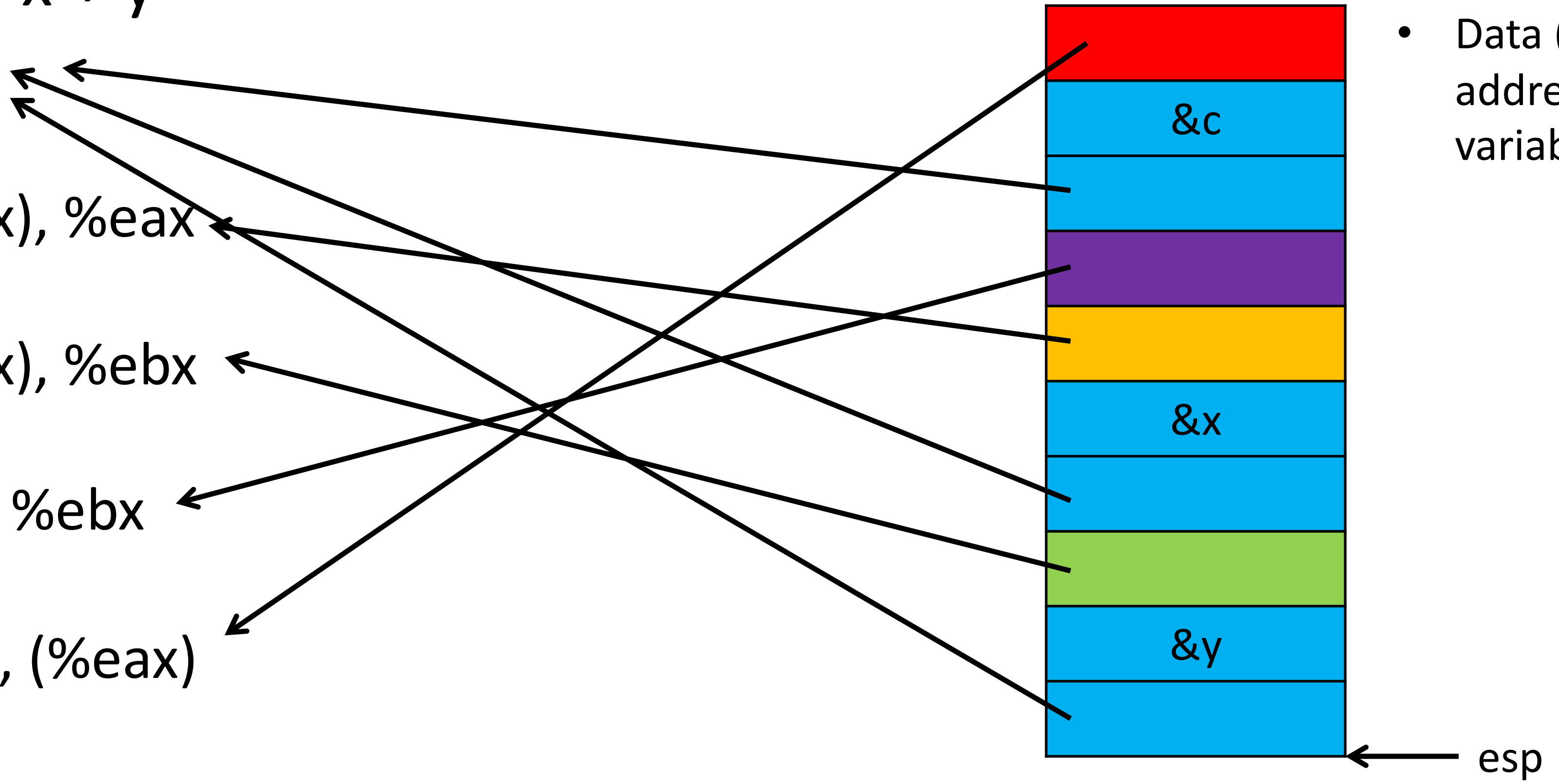
Basic strategy

1. Pop the address of one variable into a register
2. Load the value of the variable into a register
3. Pop the address of another variable into a register
4. Load the value of the variable into a register
5. Perform the operation
6. Pop the address of the destination variable into a register
7. Store the result of the operation at that address

Must be mindful of register interactions

Arithmetic

- Addition: c = x + y
  - popl %eax
    ret
  - movl (%eax), %eax
    ret
  - movl (%eax), %ebx
    ret
  - addl %eax, %ebx
    ret
  - movl %ebx, (%eax)
    ret



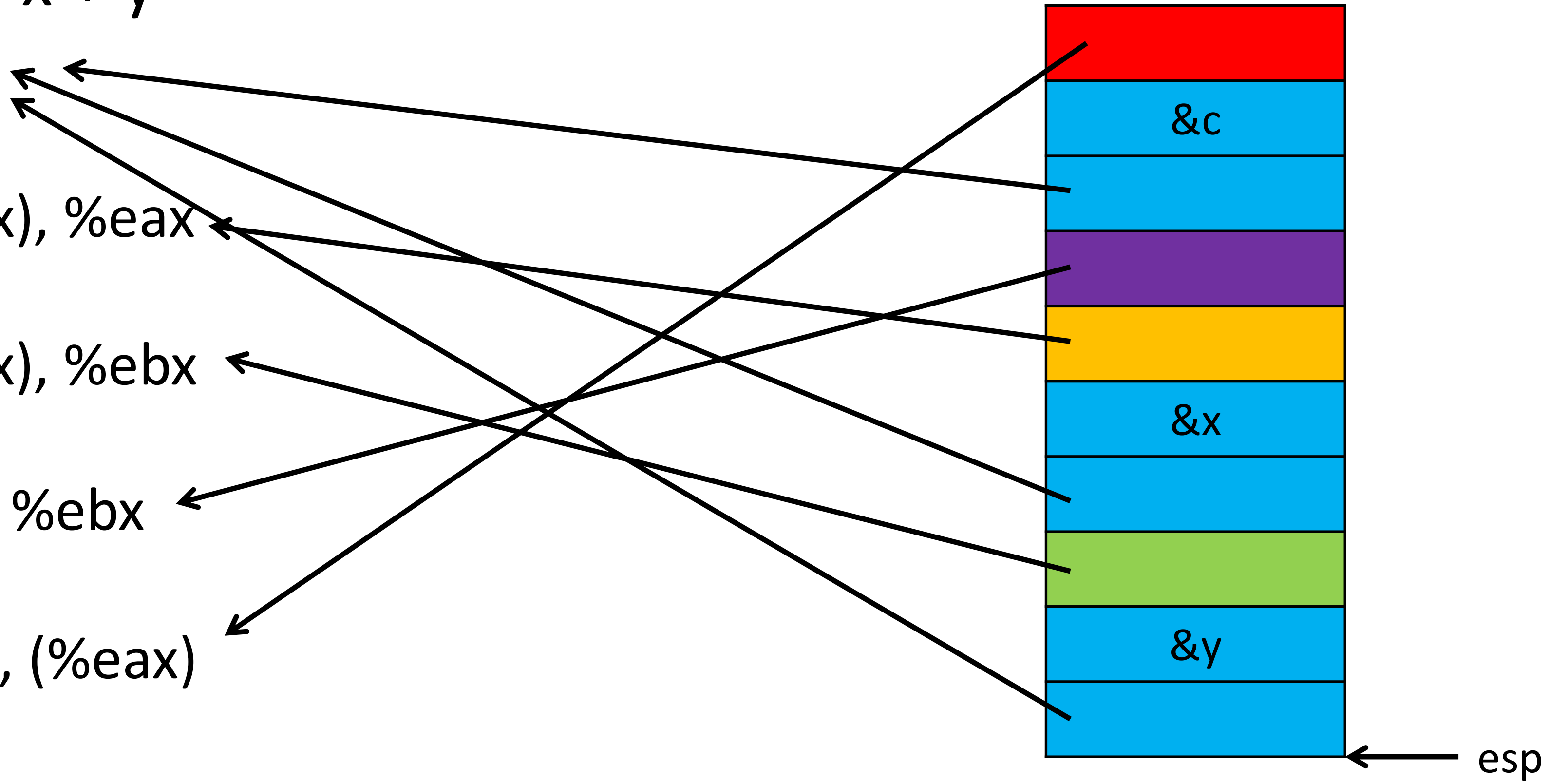
Stack contains

- Addresses of code snippets ending in ret
- Data (here, the addresses of our variables)

Arithmetic

Register	Value
eax	105
ebx	3852

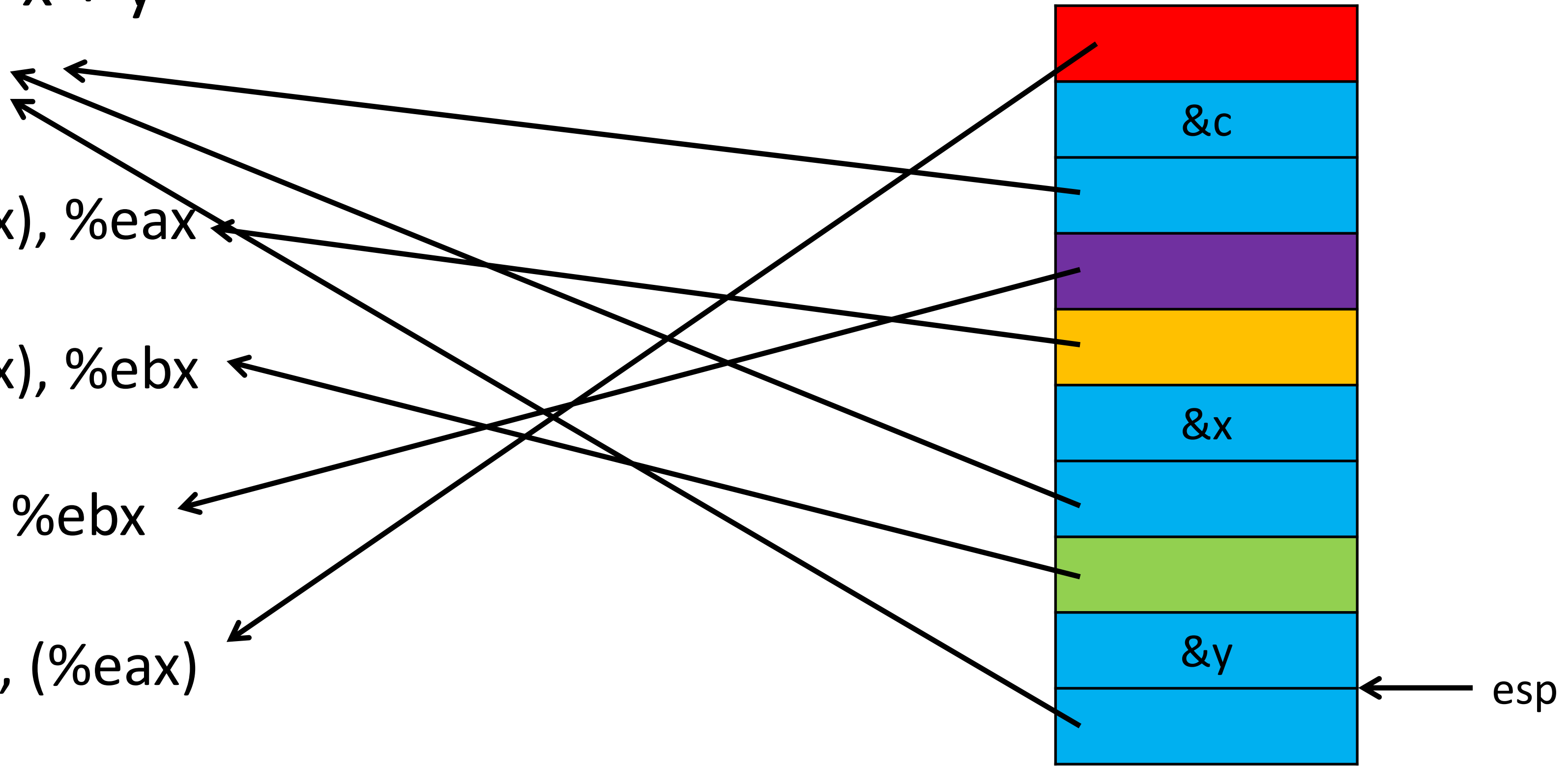
- Addition: c = x + y
  - popl %eax
    ret
  - movl (%eax), %eax
    ret
  - movl (%eax), %ebx
    ret
  - addl %eax, %ebx
    ret
  - movl %ebx, (%eax)
    ret



Arithmetic

Register	Value
eax	105
ebx	3852

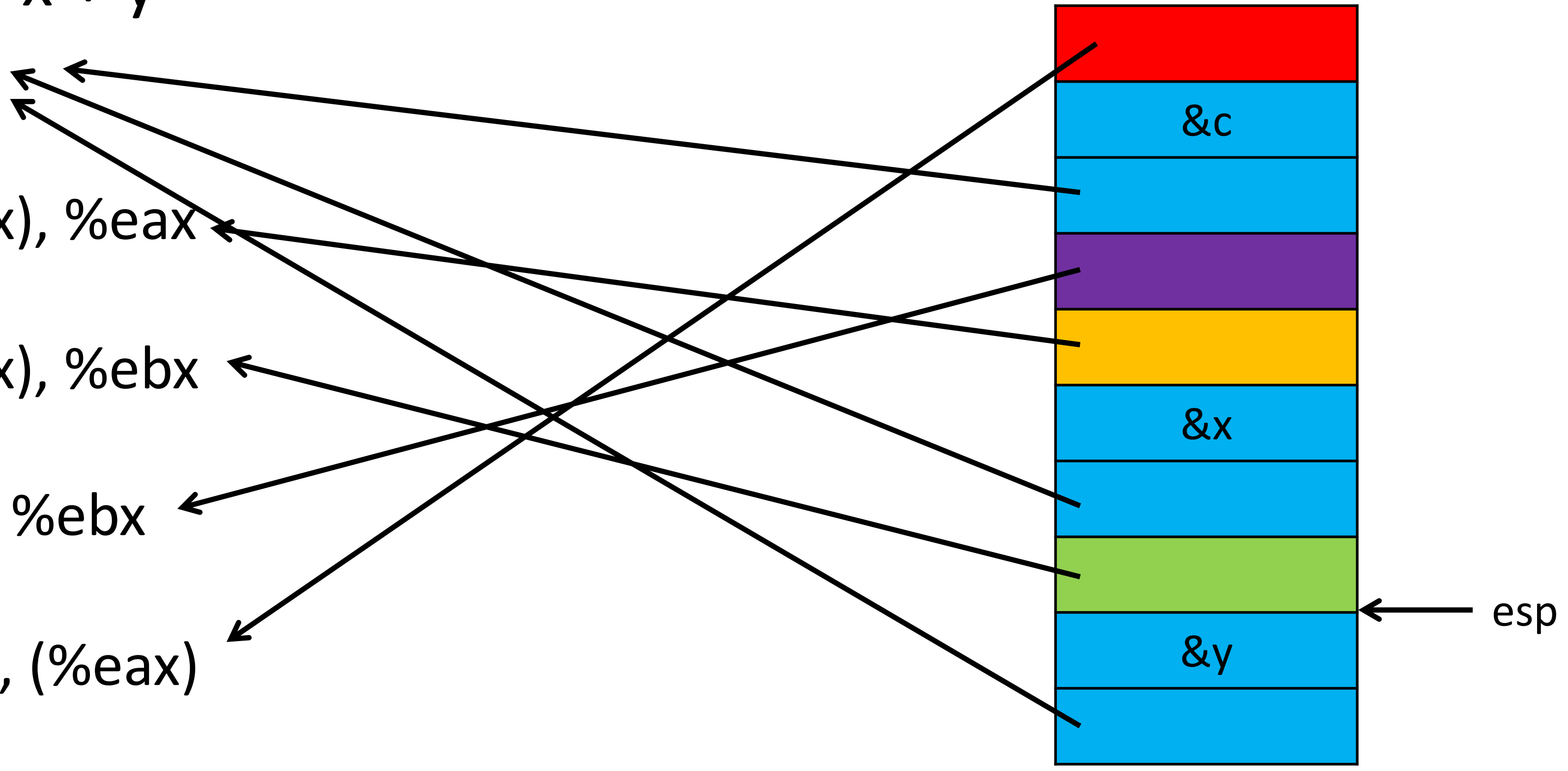
- Addition: c = x + y
  - popl %eax
    ret
  - movl (%eax), %eax
    ret
  - movl (%eax), %ebx
    ret
  - addl %eax, %ebx
    ret
  - movl %ebx, (%eax)
    ret



Arithmetic

Register	Value
eax	&y
ebx	3852

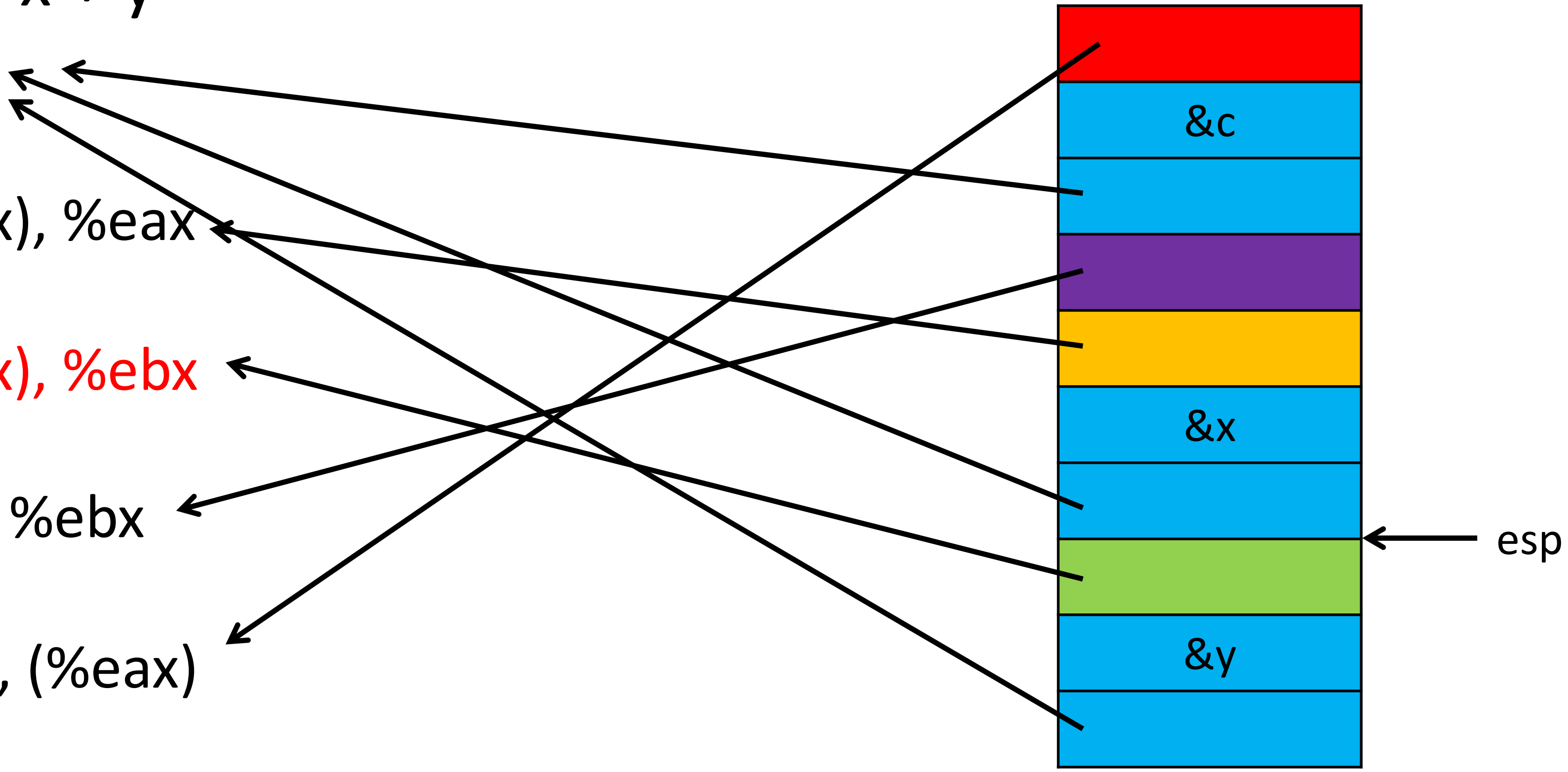
- Addition: c = x + y
  - popl %eax
    ret
  - movl (%eax), %eax
    ret
  - movl (%eax), %ebx
    ret
  - addl %eax, %ebx
    ret
  - movl %ebx, (%eax)
    ret



Arithmetic

Register	Value
eax	&y
ebx	3852

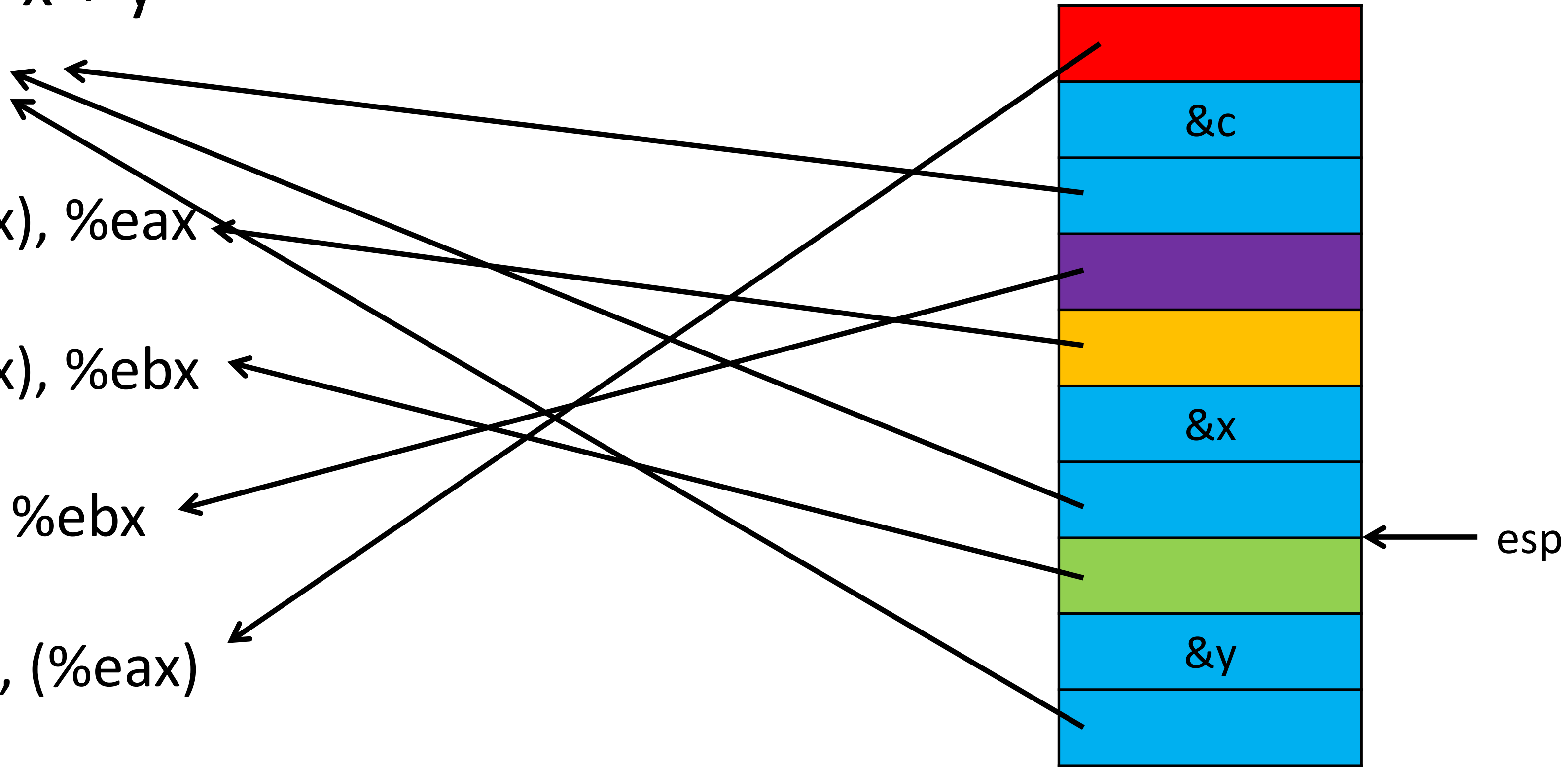
- Addition: c = x + y
  - popl %eax
    ret
  - movl (%eax), %eax
    ret
  - movl (%eax), %ebx
    ret
  - addl %eax, %ebx
    ret
  - movl %ebx, (%eax)
    ret



Arithmetic

Register	Value
eax	&y
ebx	y

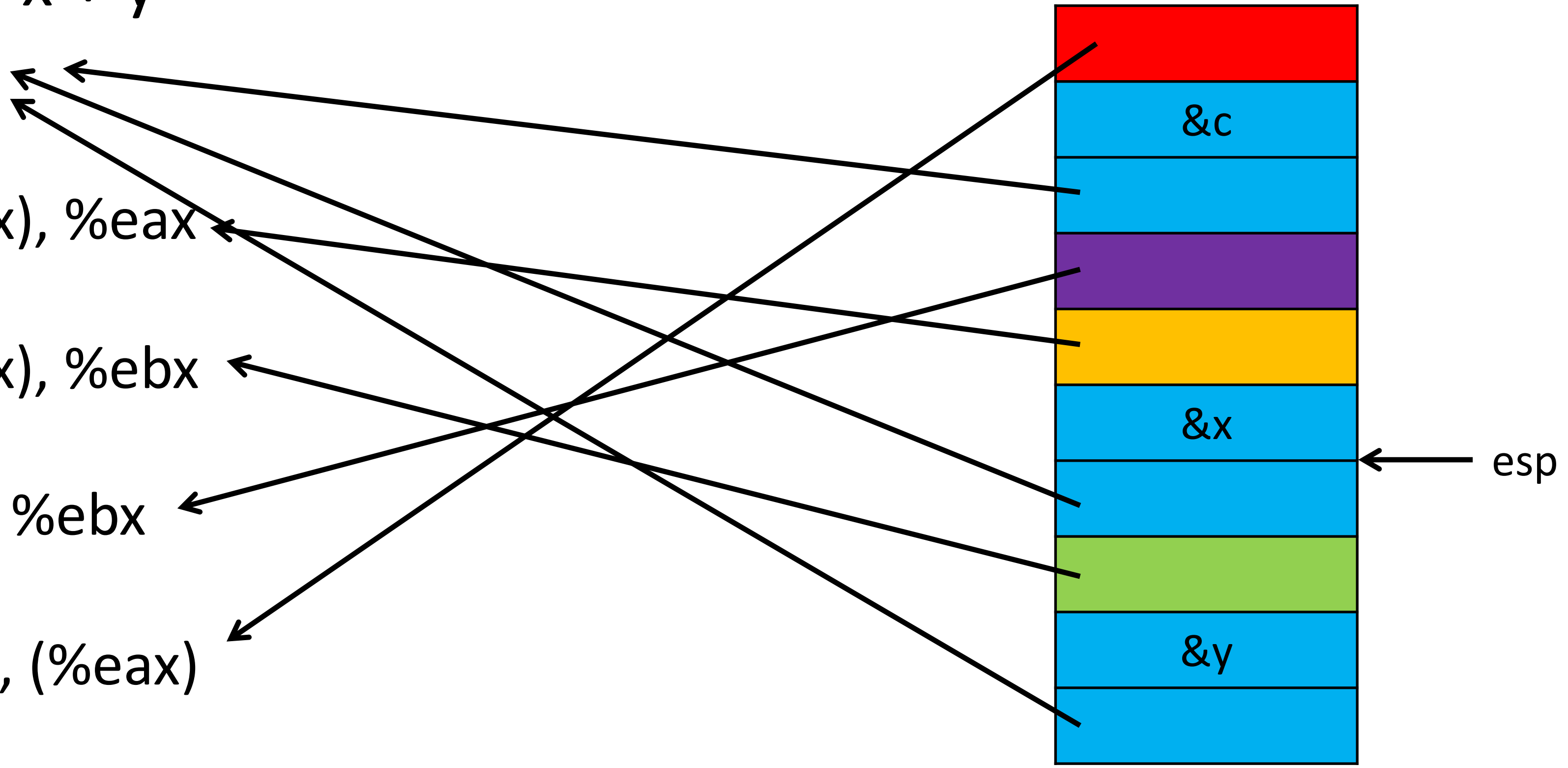
- Addition: c = x + y
  - popl %eax
    ret
  - movl (%eax), %eax
    ret
  - movl (%eax), %ebx
    ret
  - addl %eax, %ebx
    ret
  - movl %ebx, (%eax)
    ret



Arithmetic

Register	Value
eax	&y
ebx	y

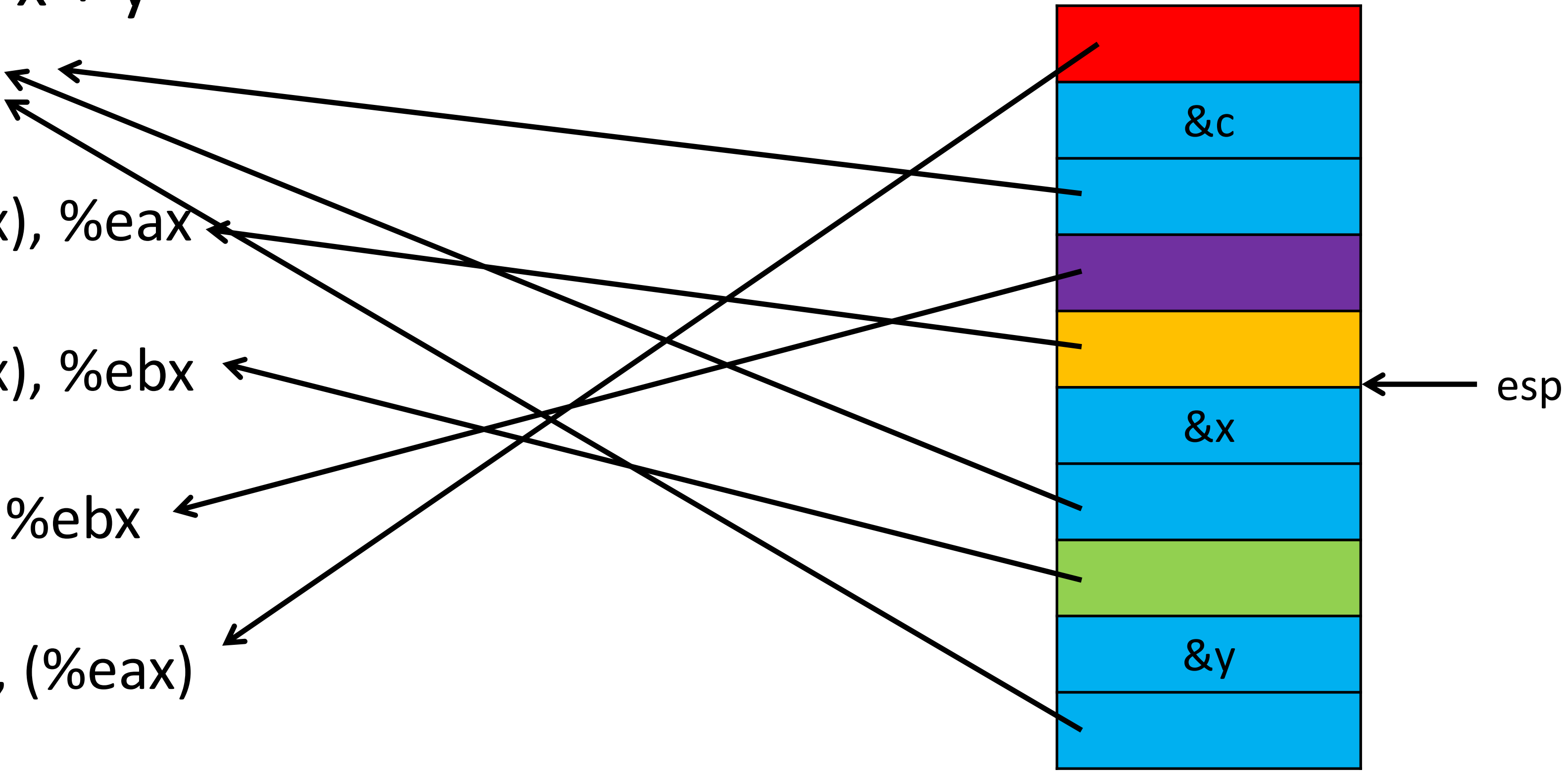
- Addition: c = x + y
  - popl %eax
    ret
  - movl (%eax), %eax
    ret
  - movl (%eax), %ebx
    ret
  - addl %eax, %ebx
    ret
  - movl %ebx, (%eax)
    ret



Arithmetic

Register	Value
eax	&x
ebx	y

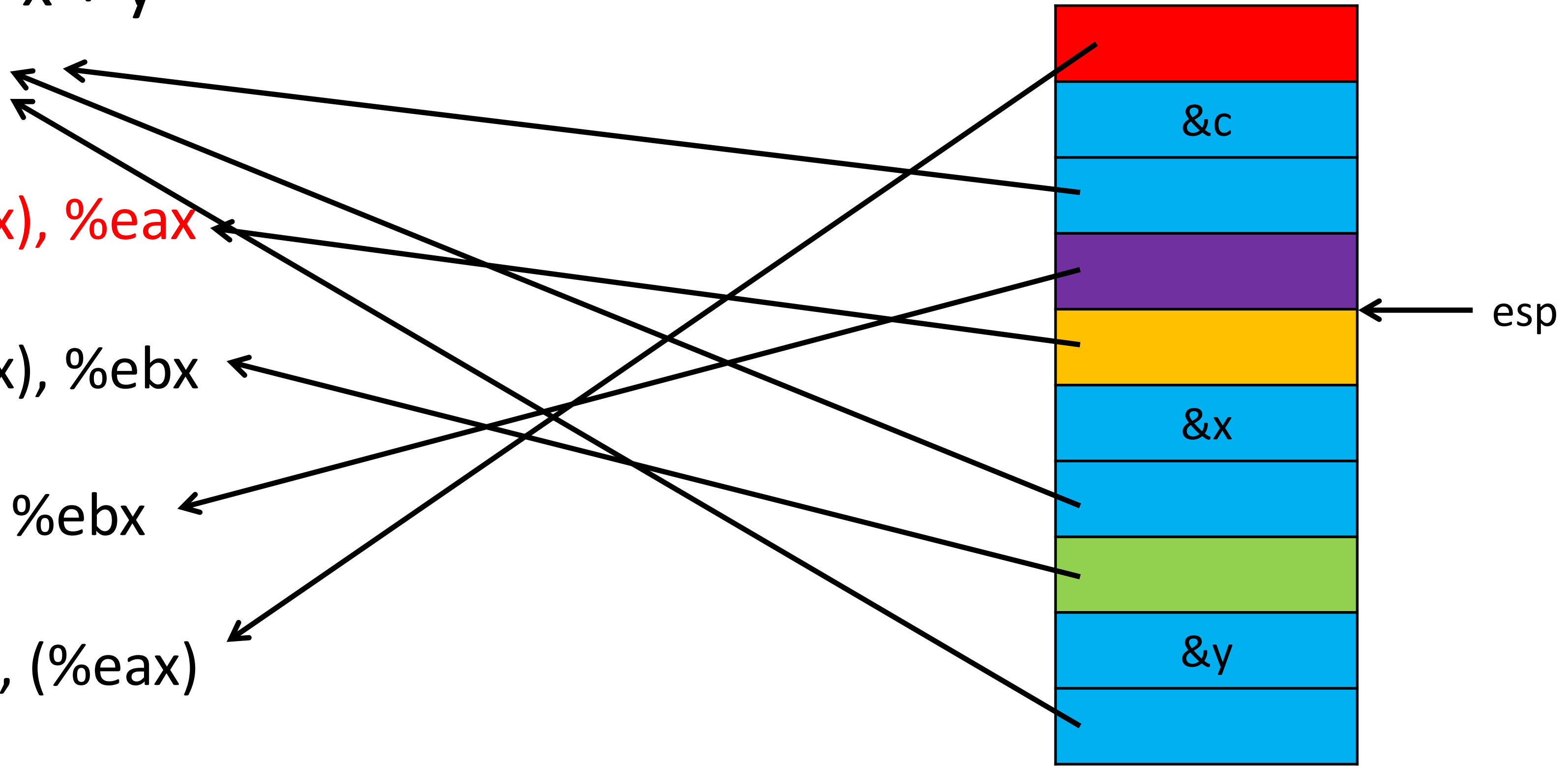
- Addition: c = x + y
  - popl %eax
    ret
  - movl (%eax), %eax
    ret
  - movl (%eax), %ebx
    ret
  - addl %eax, %ebx
    ret
  - movl %ebx, (%eax)
    ret



Arithmetic

Register	Value
eax	&x
ebx	y

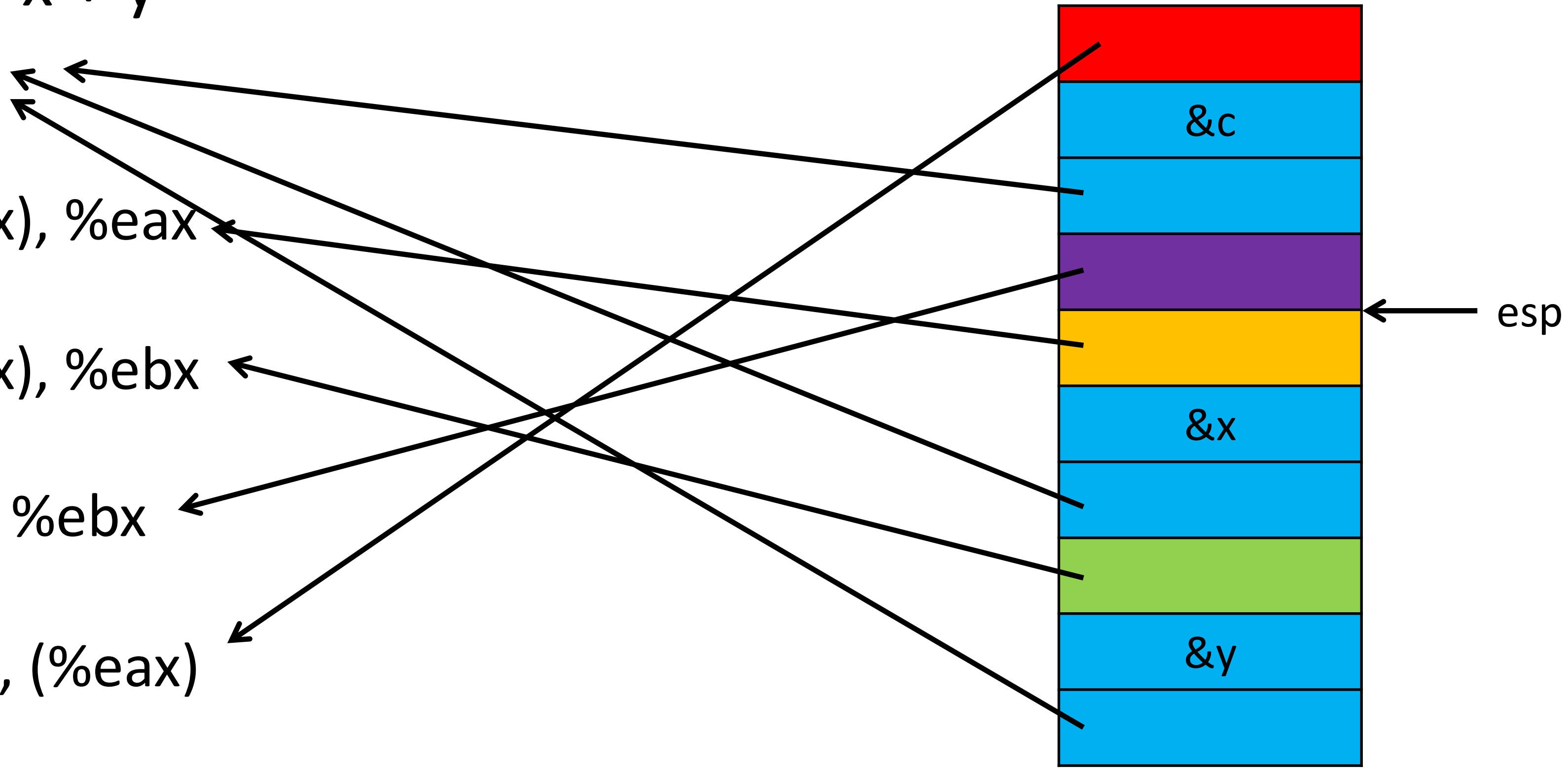
- Addition: c = x + y
  - popl %eax
    ret
  - movl (%eax), %eax
    ret
  - movl (%eax), %ebx
    ret
  - addl %eax, %ebx
    ret
  - movl %ebx, (%eax)
    ret



Arithmetic

Register	Value
eax	x
ebx	y

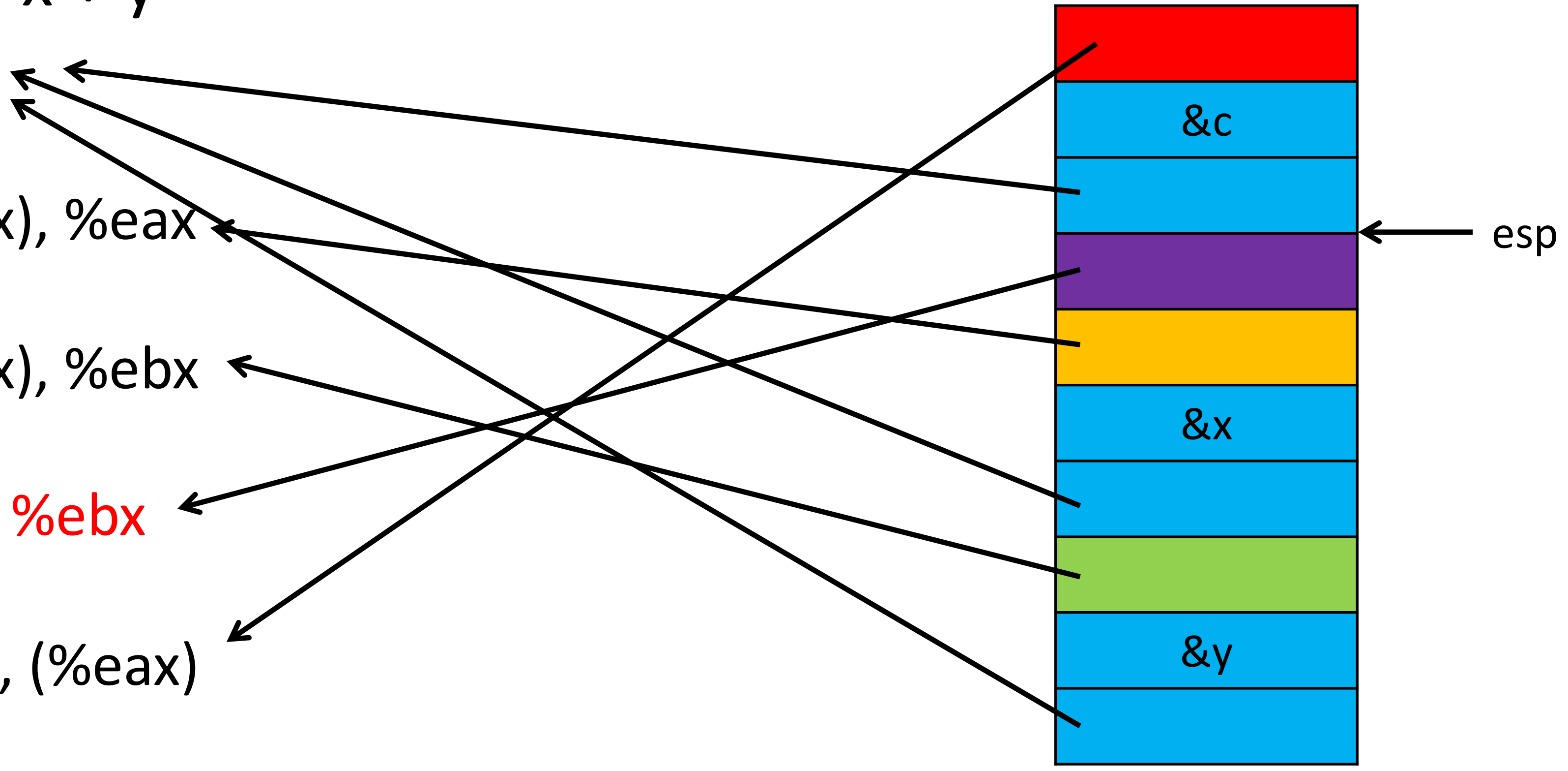
- Addition: c = x + y
    - popl %eax
ret
    - movl (%eax), %eax
ret
    - movl (%eax), %ebx
ret
    - addl %eax, %ebx
ret
    - movl %ebx, (%eax)
ret



Arithmetic

Register	Value
eax	x
ebx	y

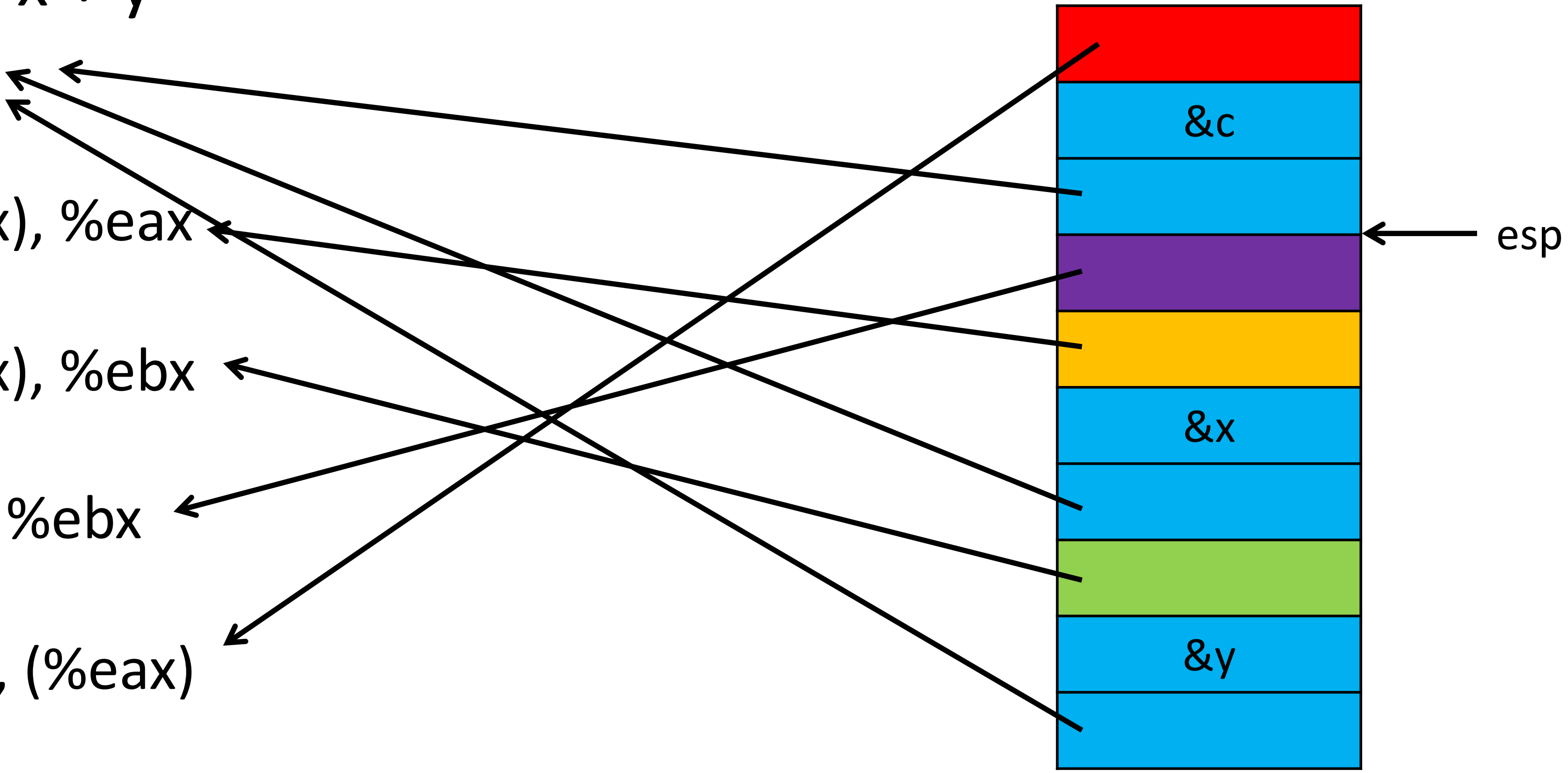
- Addition: c = x + y
  - popl %eax
ret
  - movl (%eax), %eax
ret
  - movl (%eax), %ebx
ret
  - addl %eax, %ebx
ret
  - movl %ebx, (%eax)
ret



Arithmetic

Register	Value
eax	x
ebx	y + x

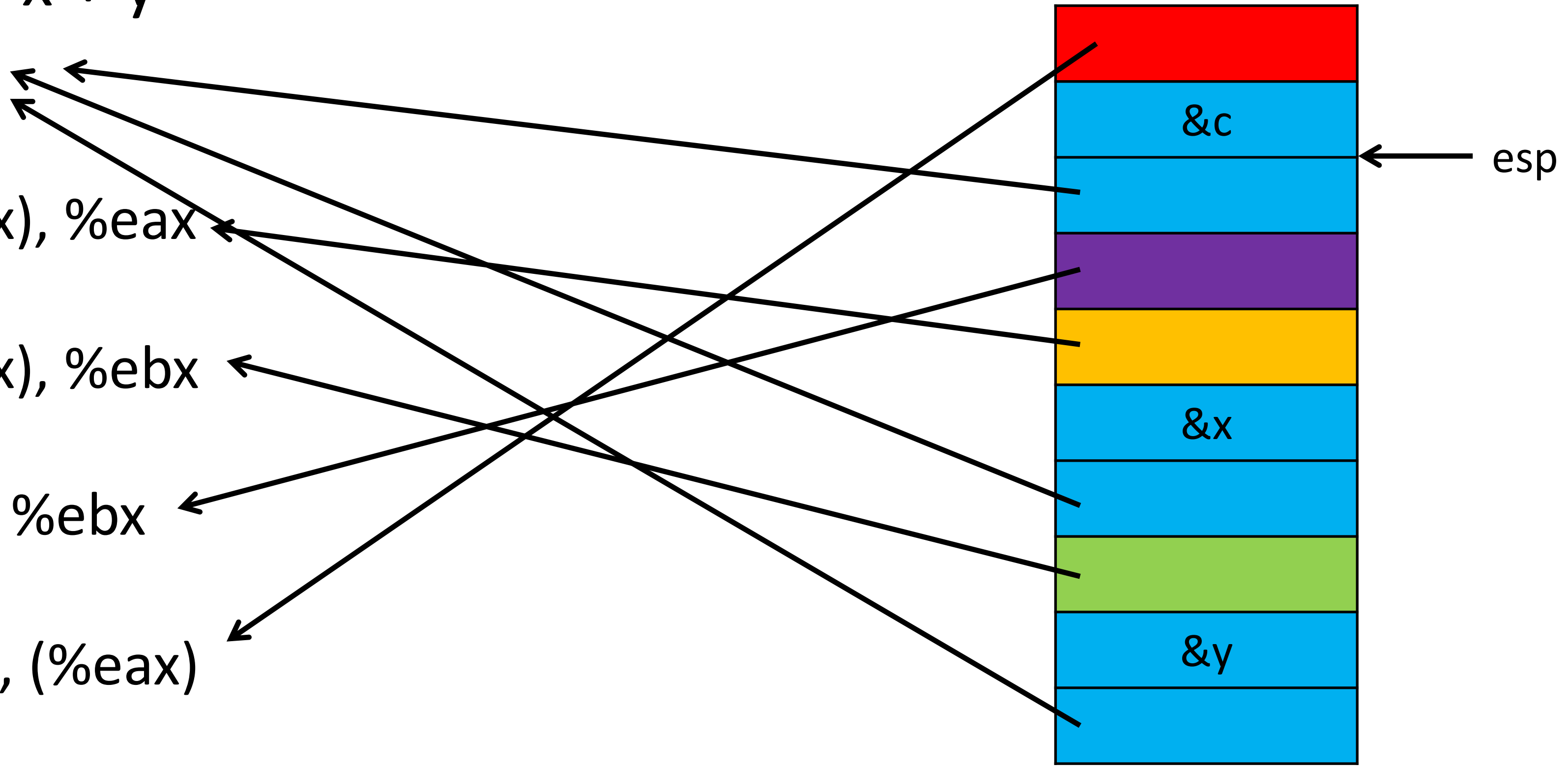
- Addition: c = x + y
  - popl %eax
    ret
  - movl (%eax), %eax
    ret
  - movl (%eax), %ebx
    ret
  - addl %eax, %ebx
    ret
  - movl %ebx, (%eax)
    ret



Arithmetic

Register	Value
eax	x
ebx	y + x

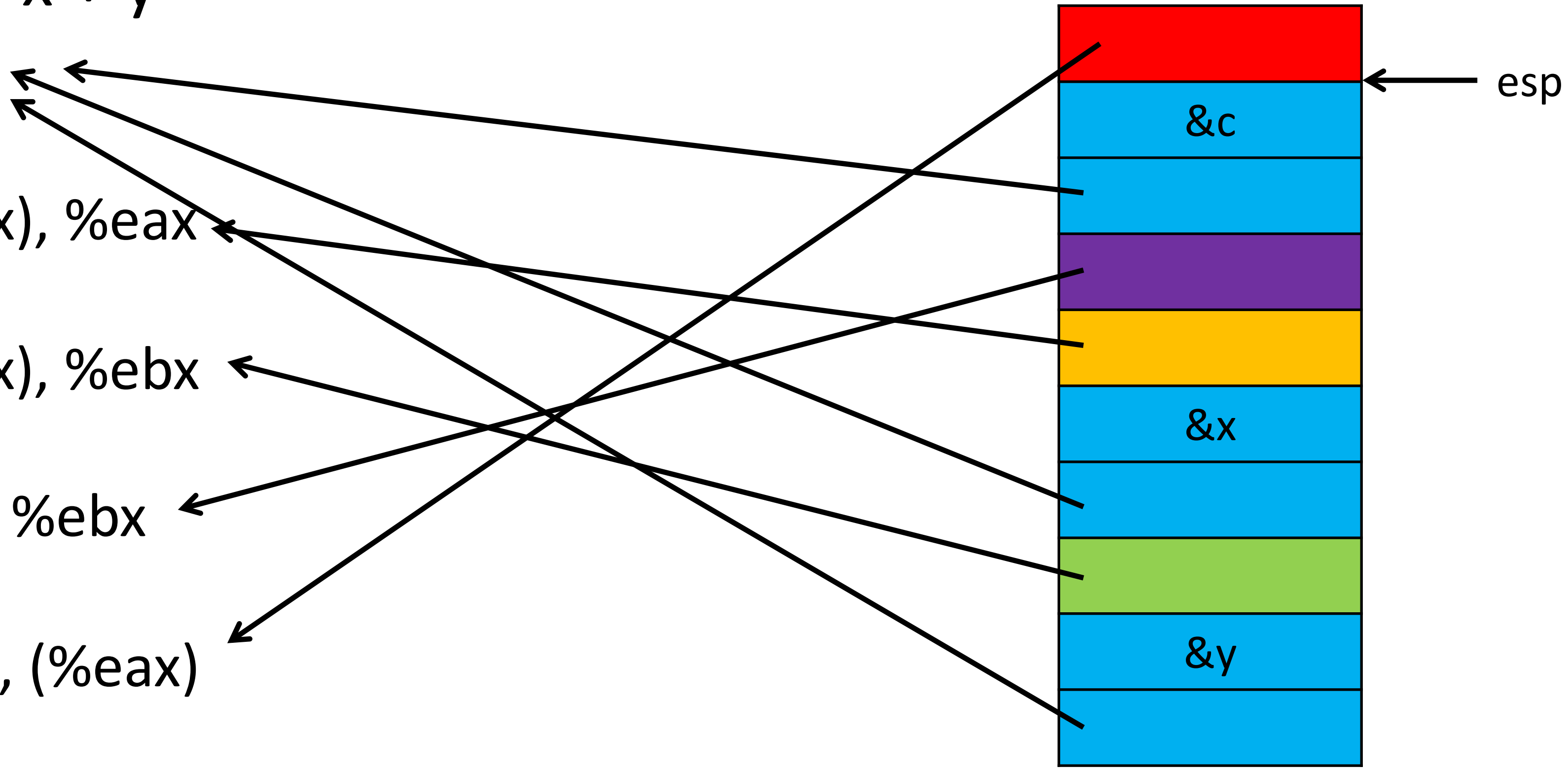
- Addition: c = x + y
  - popl %eax
    ret
  - movl (%eax), %eax
    ret
  - movl (%eax), %ebx
    ret
  - addl %eax, %ebx
    ret
  - movl %ebx, (%eax)
    ret



Arithmetic

Register	Value
eax	&c
ebx	y + x

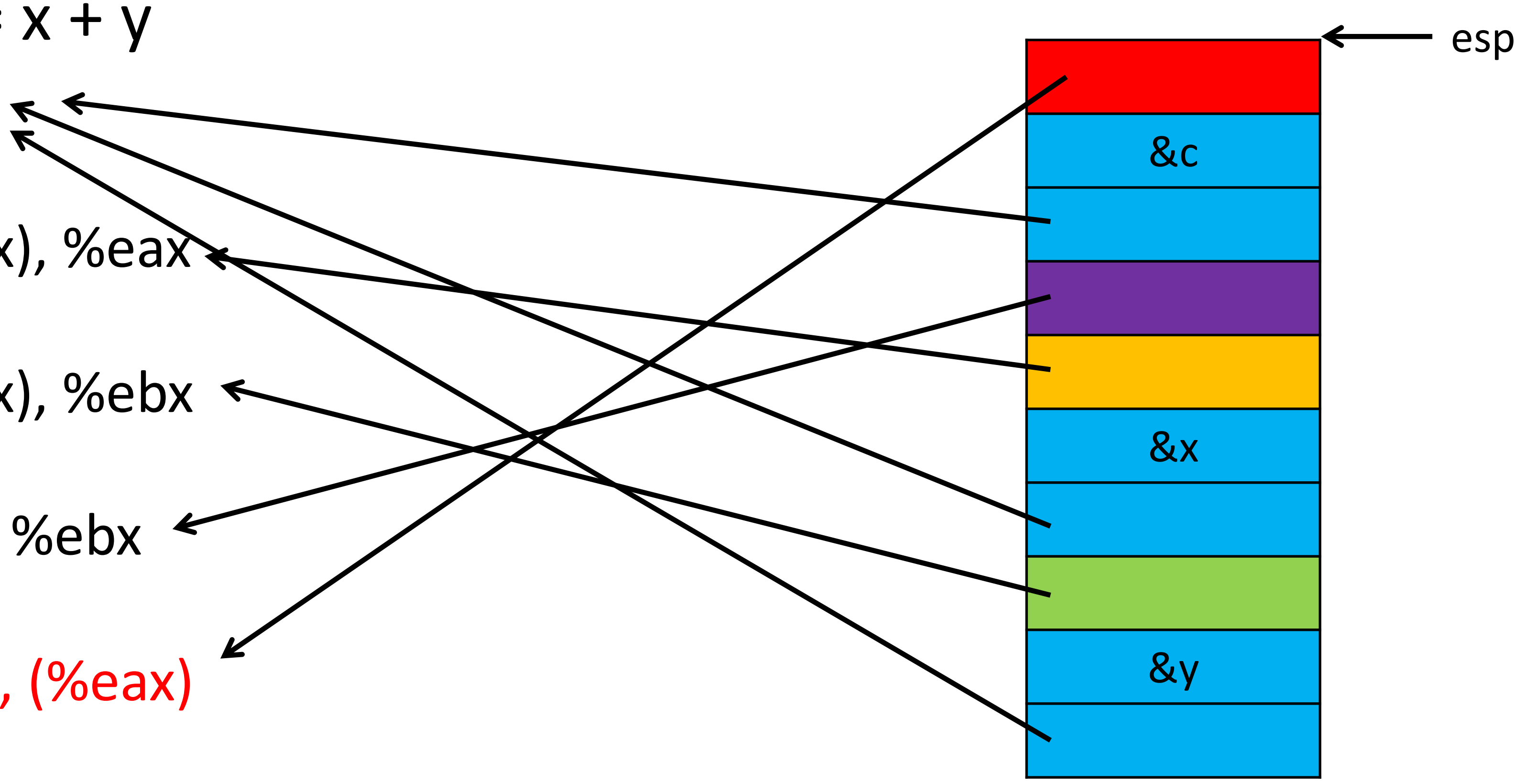
- Addition: c = x + y
  - popl %eax
    ret
  - movl (%eax), %eax
    ret
  - movl (%eax), %ebx
    ret
  - addl %eax, %ebx
    ret
  - movl %ebx, (%eax)
    ret



Arithmetic

Register	Value
eax	&c
ebx	y + x

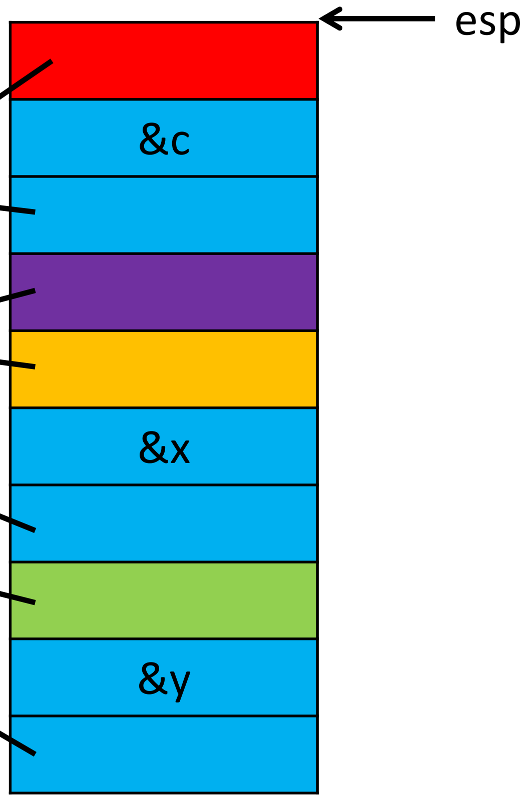
- Addition: c = x + y
  - popl %eax
    ret
  - movl (%eax), %eax
    ret
  - movl (%eax), %ebx
    ret
  - addl %eax, %ebx
    ret
  - movl %ebx, (%eax)
    ret



Arithmetic

Register	Value
eax	&c
ebx	y + x

- Addition: c = x + y
  - popl %eax
    ret
  - movl (%eax), %eax
    ret
  - movl (%eax), %ebx
    ret
  - addl %eax, %ebx
    ret
  - movl %ebx, (%eax)
    ret



What else can we do?

- Depends on the code we have access to
- Usually: Arbitrary Turing-complete behavior
  - Arithmetic
  - Logic
  - Conditionals and loops
  - Subroutines
  - Calling existing functions
  - System calls
- Sometimes: More limited behavior
  - Often enough for straight-line code and system calls

Comparing ROP to normal programming

	Normal programming	ROP
Instruction pointer	eip	esp
No-op	nop	ret
Unconditional jump	jmp address	set esp to address of gadget
Conditional jump	jnz address	set esp to address of gadget if some condition is met
Variables	memory and registers	mostly memory
Inter-instruction (inter-gadget) register and memory interaction	minimal, mostly explicit; e.g., adding two registers only affects the destination register	can be complex; e.g., adding two registers may involve modifying many registers which impacts other gadgets

Return-oriented conditionals

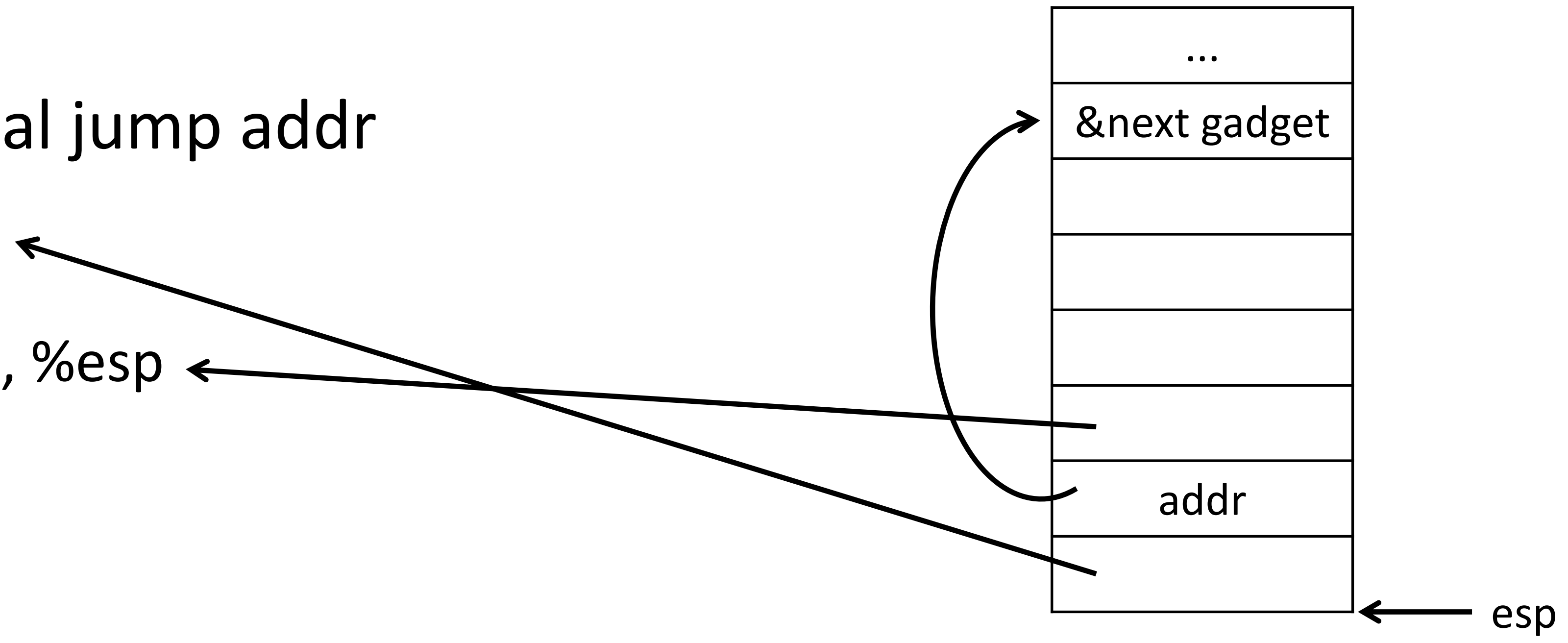
- Processors support instructions that conditionally change the PC
  - On x86
    - Jcc family: jz, jnz, jl, jle, etc. 33 in total
    - loop, loope, loopne
    - Based on condition codes mostly; and on ecx for some
  - On MIPS
    - beq, bne, blez, etc.
    - Based on comparison of registers
- Processors generally don't support for conditionally changing the stack pointer (with some exceptions)

We want conditional jumps

- Unconditional jump addr
  - popl %eax
    ret
  - movl %eax, %esp
    ret

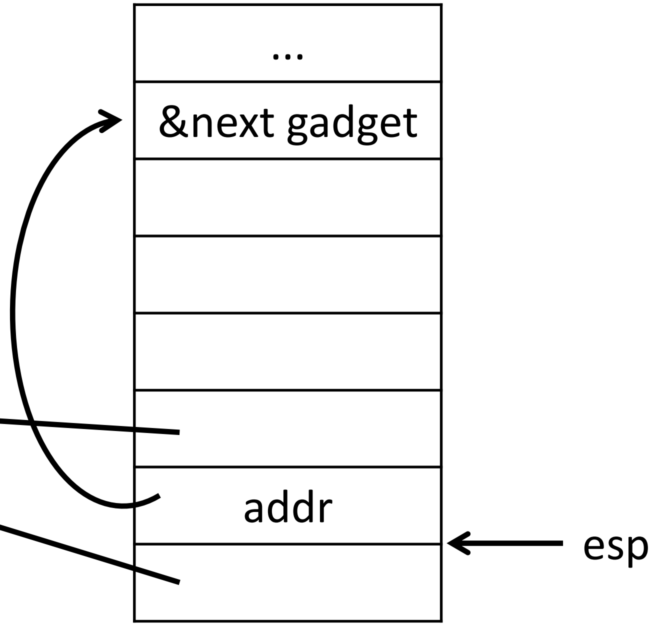
We want conditional jumps

- Unconditional jump addr
  - popl %eax
    ret
  - movl %eax, %esp
    ret



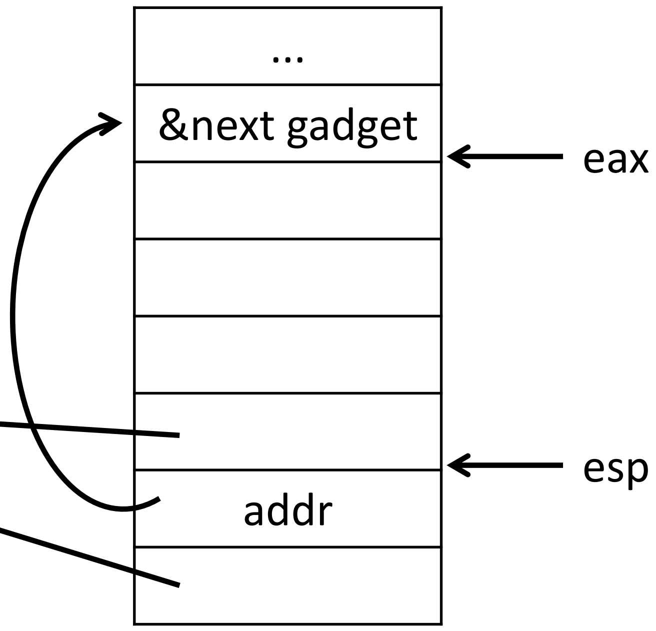
We want conditional jump

- Unconditional jump addr
  - popl %eax
    ret
  - movl %eax, %esp
    ret



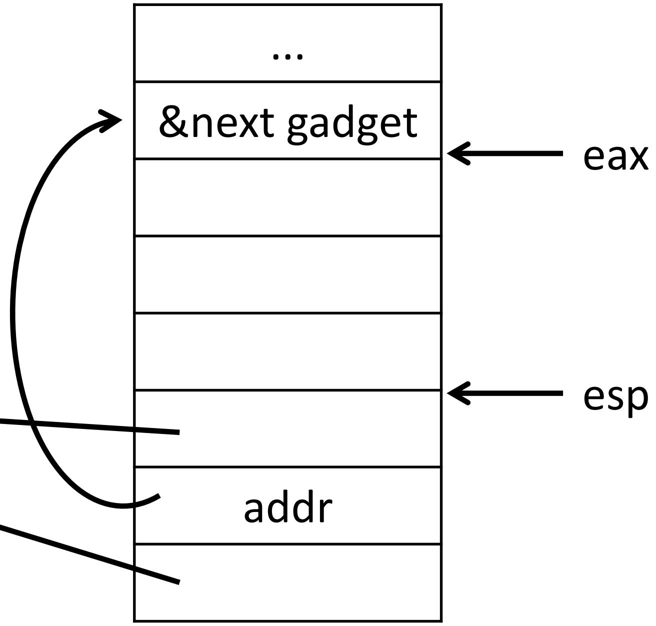
We want conditional jump

- Unconditional jump addr
  - popl %eax
    ret
  - movl %eax, %esp
    ret



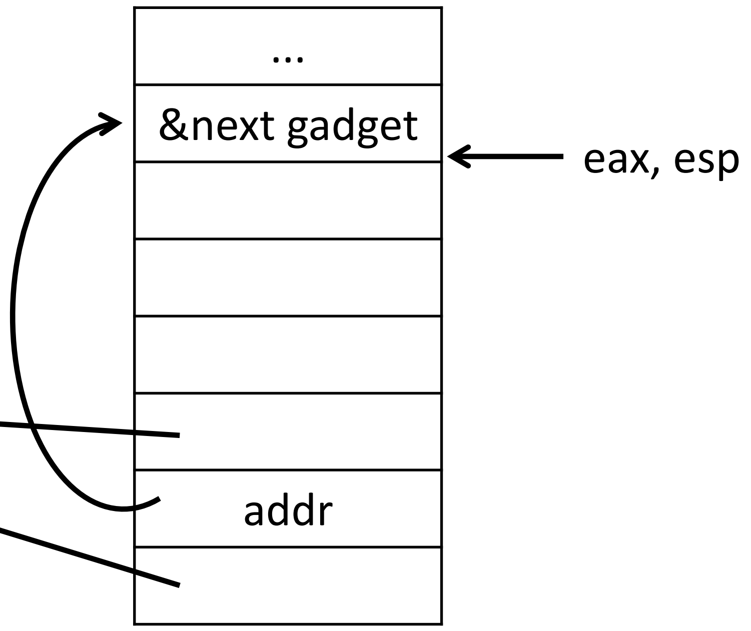
We want conditional jumps

- Unconditional jump addr
  - popl %eax
    ret
  - movl %eax, %esp
    ret



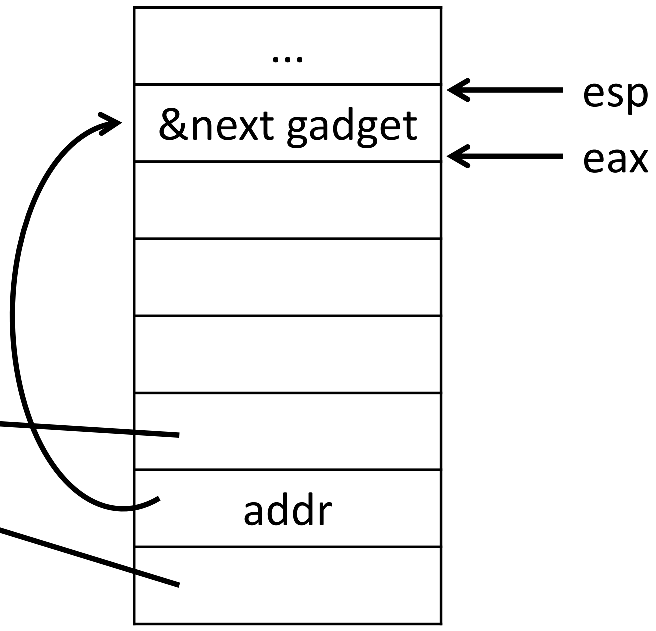
We want conditional jumps

- Unconditional jump addr
  - popl %eax
    ret
  - movl %eax, %esp
    ret



We want conditional jumps

- Unconditional jump addr
  - popl %eax
    ret
  - movl %eax, %esp
    ret



We want conditional jumps

- Unconditional jump addr
  - popl %eax
    ret
  - movl %eax, %esp
    ret
- Conditional jump addr, one way
  - Conditionally set a register to 0 or 0xffffffff
  - Perform a logical AND with the register and an offset
  - Add the result to esp

Conditionally set a register to 0 or 0xffffffff

- Compare registers eax and ebx and set ecx to
  - 0xffffffff if eax < ebx
  - 0 if eax >= ebx
- Ideally we would find a sequence like

```
        cmpl %ebx, %eax             set carry flag cf according to eax - ebx
        sbbl %ecx, %ecx             ecx ← ecx - ecx - cf; or ecx ← -cf
        ret
```

- Unlikely to find this; instead look for cmp; ret and sbb; ret sequences

Performing a logical AND with a constant

- Pop the constant into a register using pop; ret
- Use an and; ret sequence

Updating the stack pointer

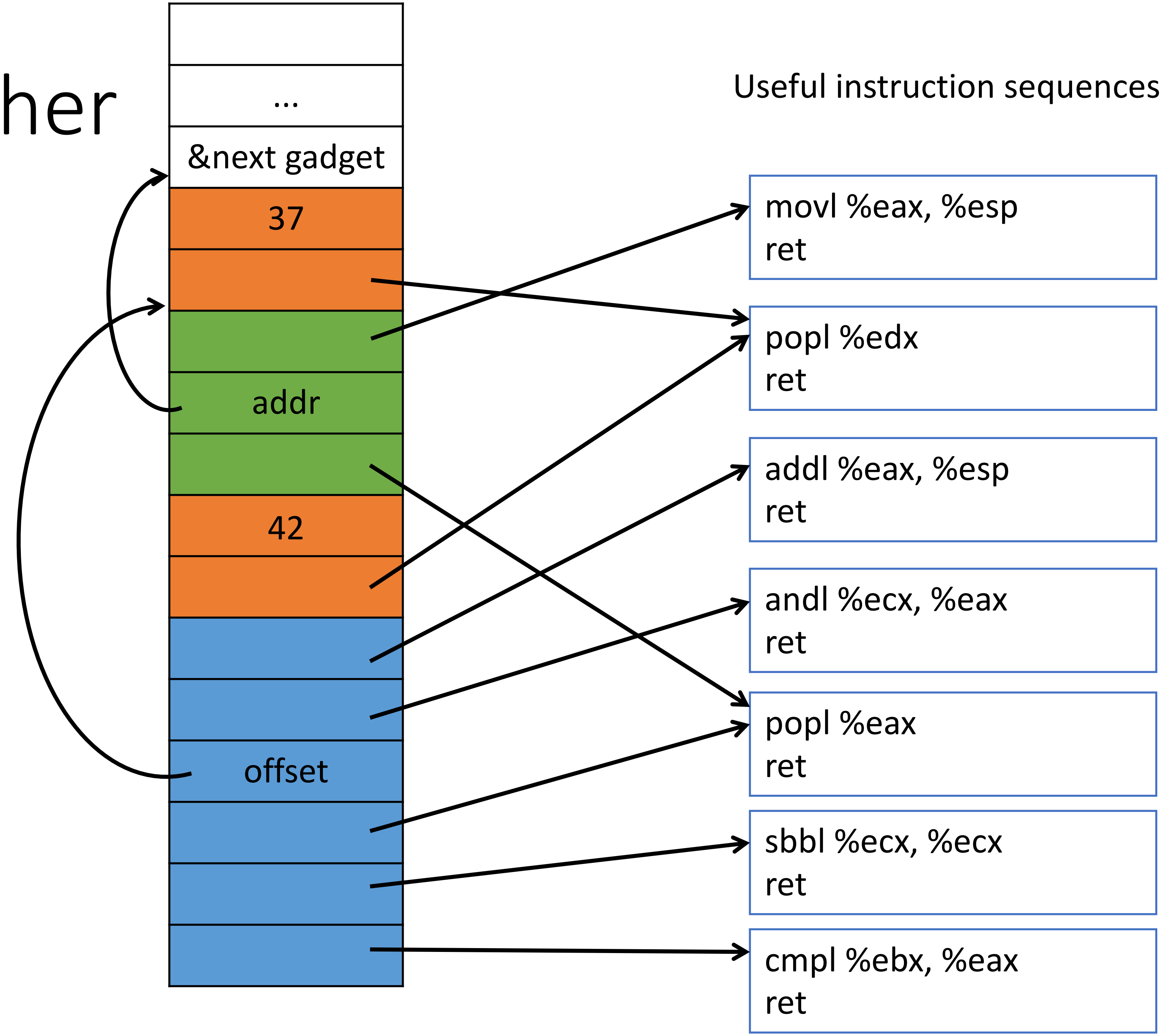
- Use an add esp; ret sequence

Putting it together

Conditional jump

Load constant in edx

Unconditional jump



Useful instruction sequences

```
movl %eax, %esp
ret
```

```
popl %edx
ret
```

```
addl %eax, %esp
ret
```

```
andl %ecx, %eax
ret
```

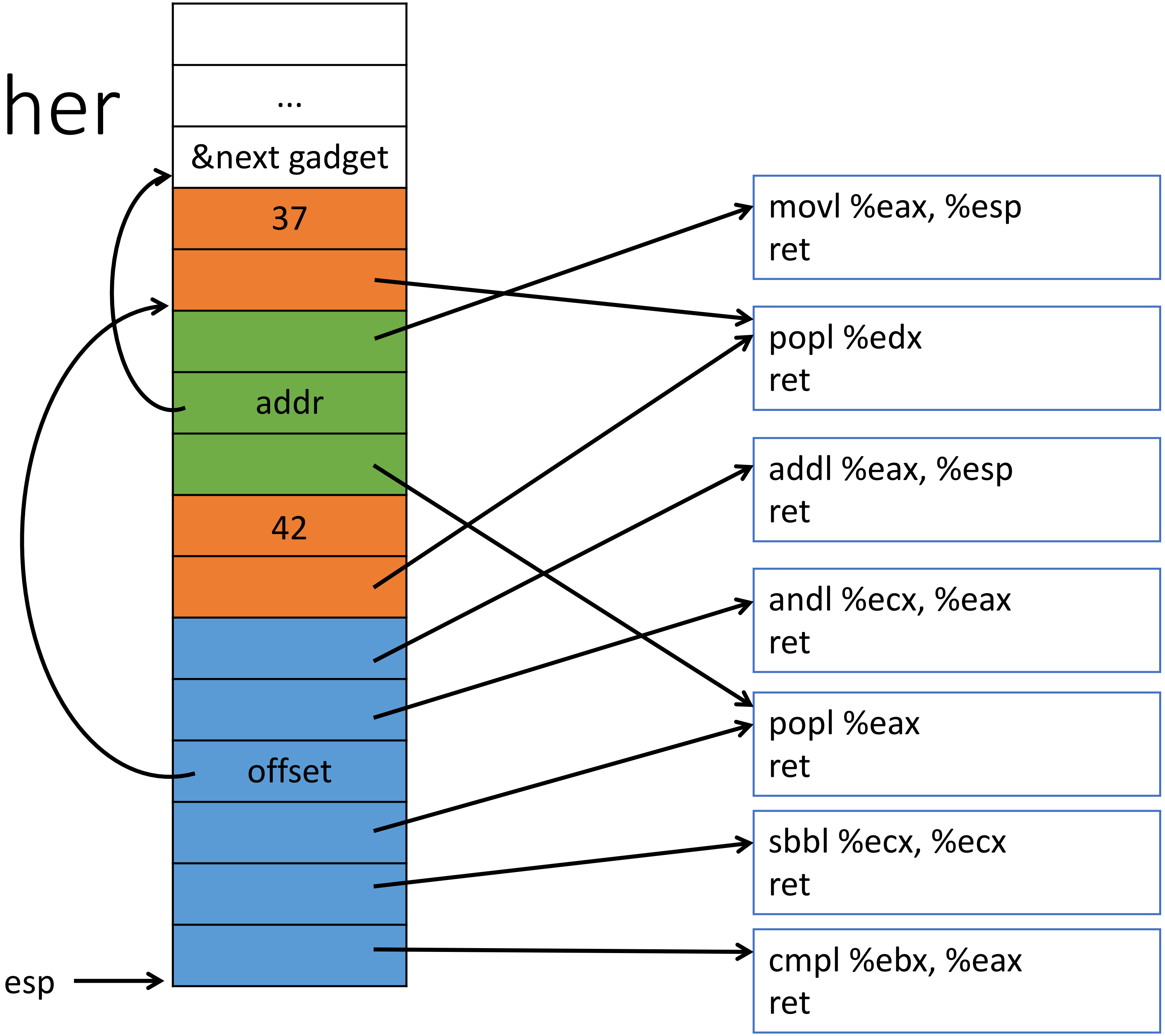
```
popl %eax
ret
```

```
sbbl %ecx, %ecx
ret
```

```
cmpl %ebx, %eax
ret
```

Putting it together

Register	Value
eax	10
ebx	20
ecx	108
edx	17



movl %eax, %esp
ret

popl %edx
ret

addl %eax, %esp
ret

andl %ecx, %eax
ret

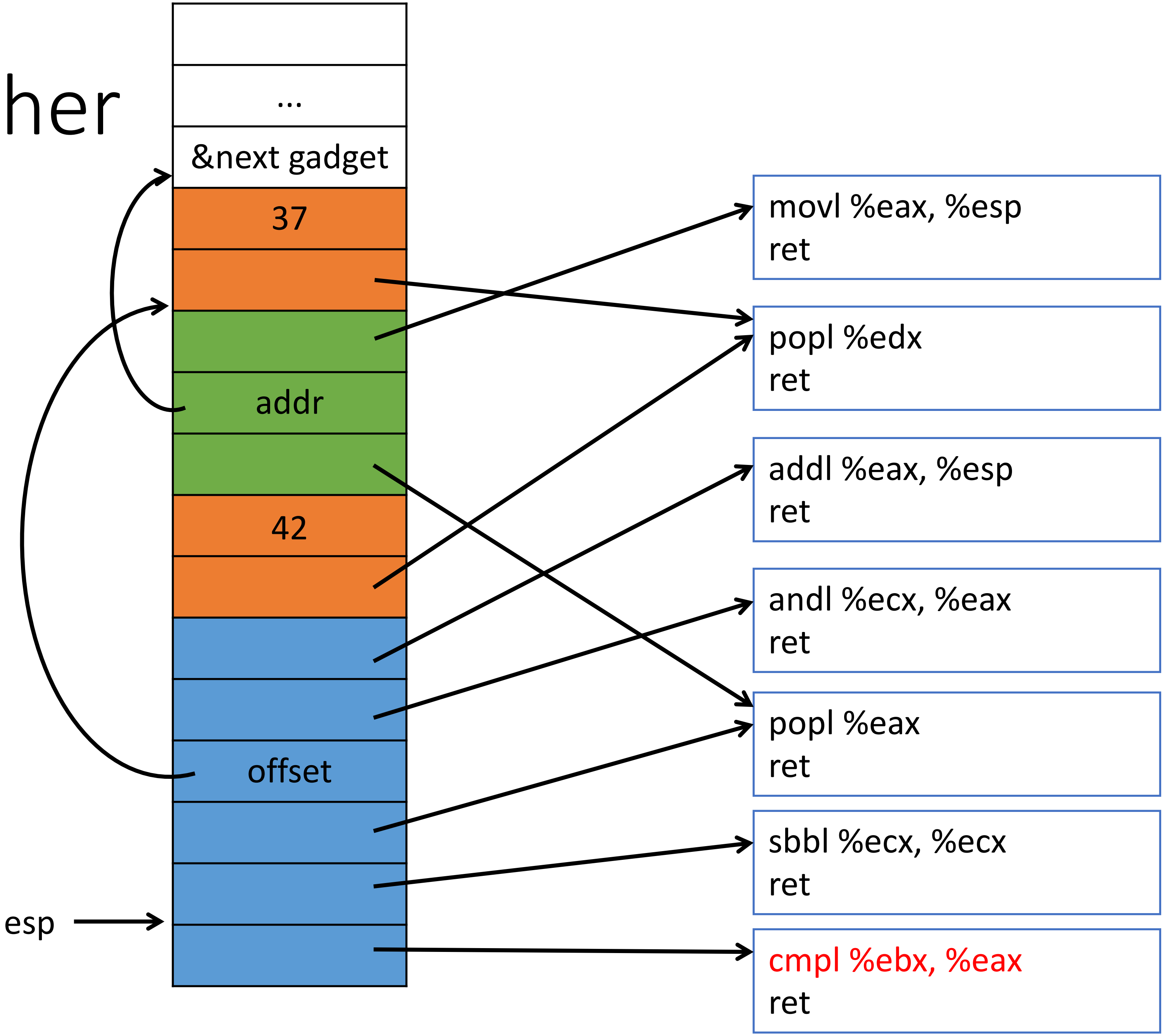
popl %eax
ret

sbbl %ecx, %ecx
ret

cmpl %ebx, %eax
ret

Putting it together

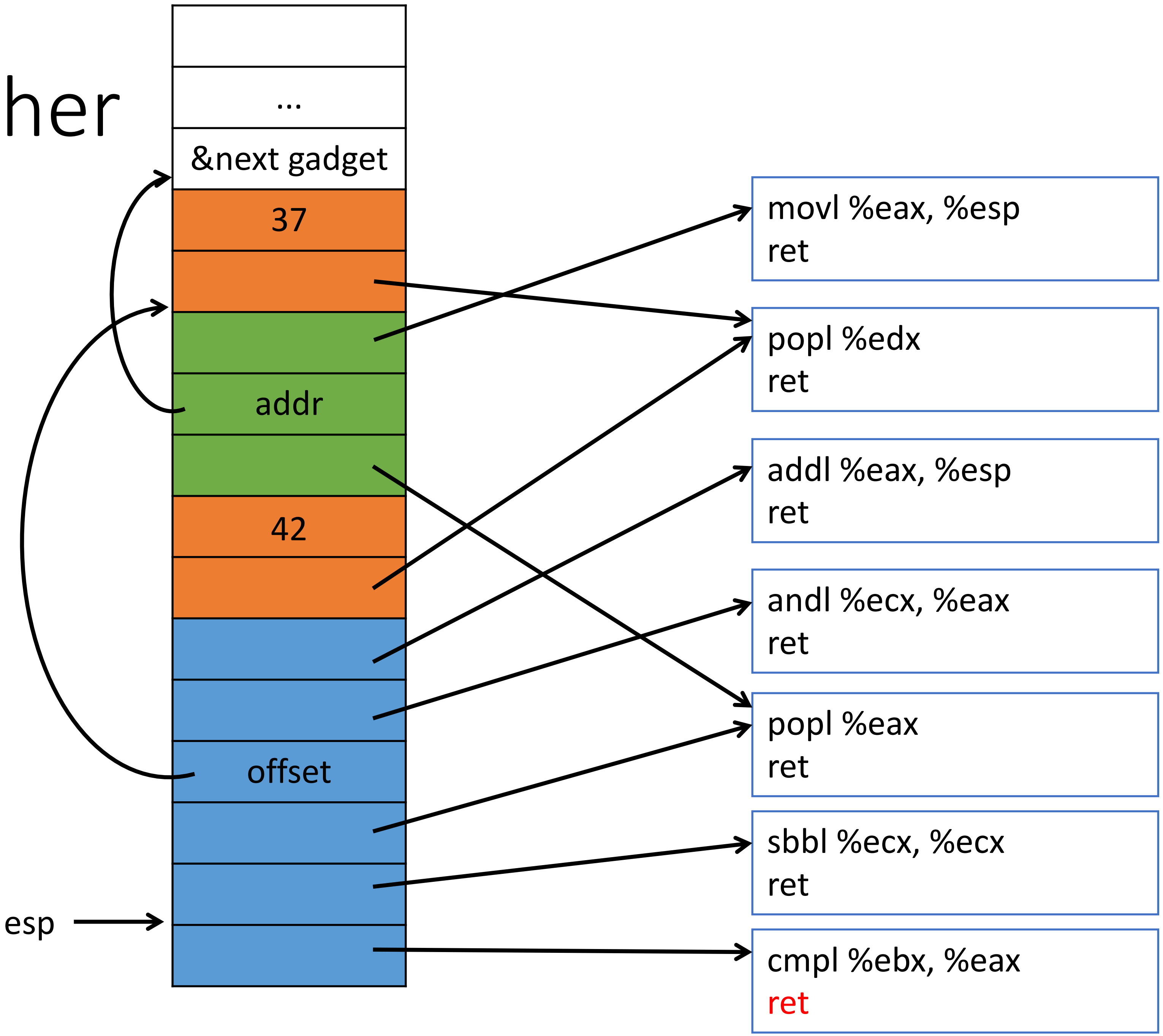
Register	Value
eax	10
ebx	20
ecx	108
edx	17



Putting it together

Register	Value
eax	10
ebx	20
ecx	108
edx	17

cf = 1



```
movl %eax, %esp
ret
```

```
popl %edx
ret
```

```
addl %eax, %esp
ret
```

```
andl %ecx, %eax
ret
```

```
popl %eax
ret
```

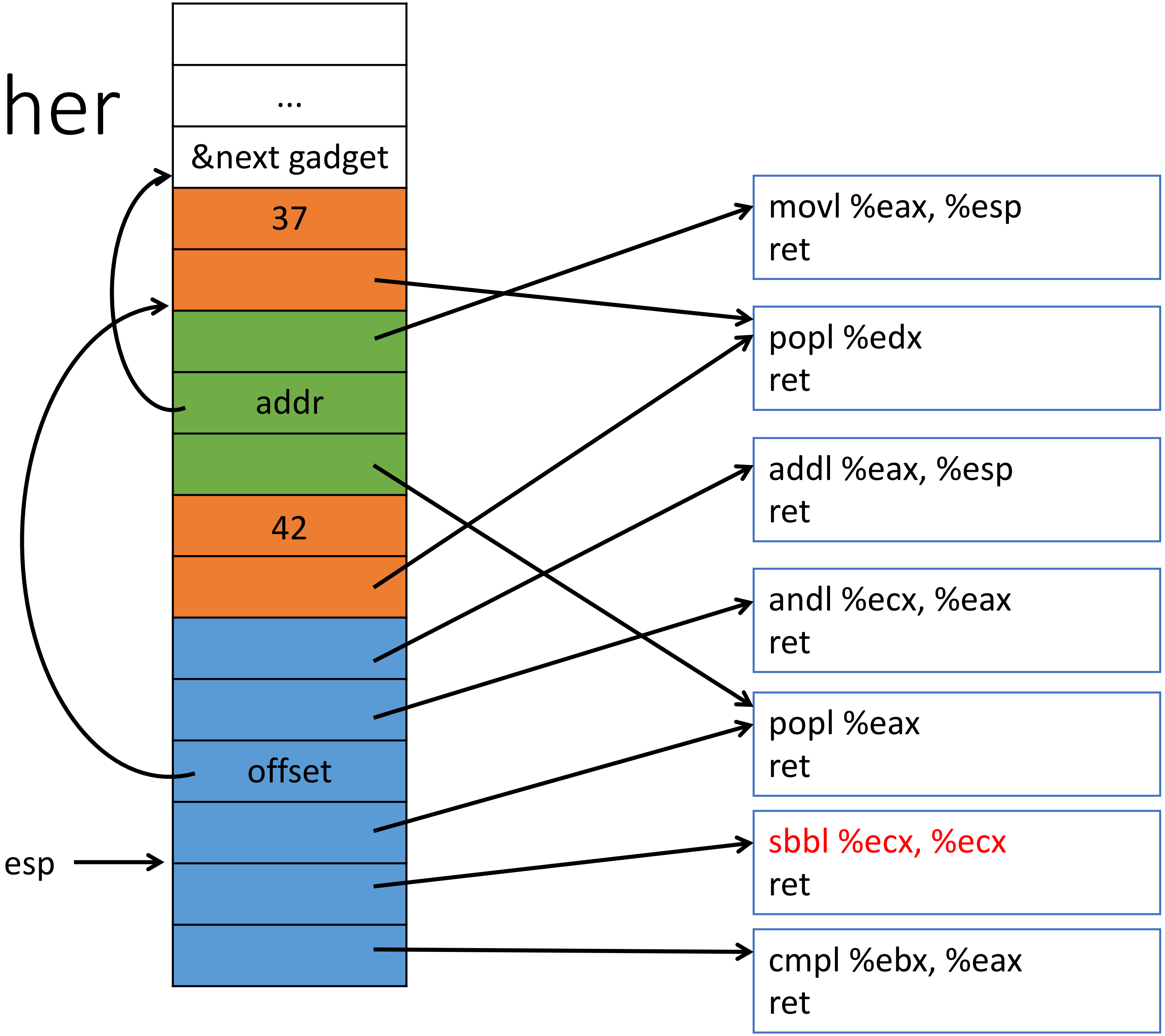
```
sbbl %ecx, %ecx
ret
```

```
cmpl %ebx, %eax
ret
```

Putting it together

Register	Value
eax	10
ebx	20
ecx	108
edx	17

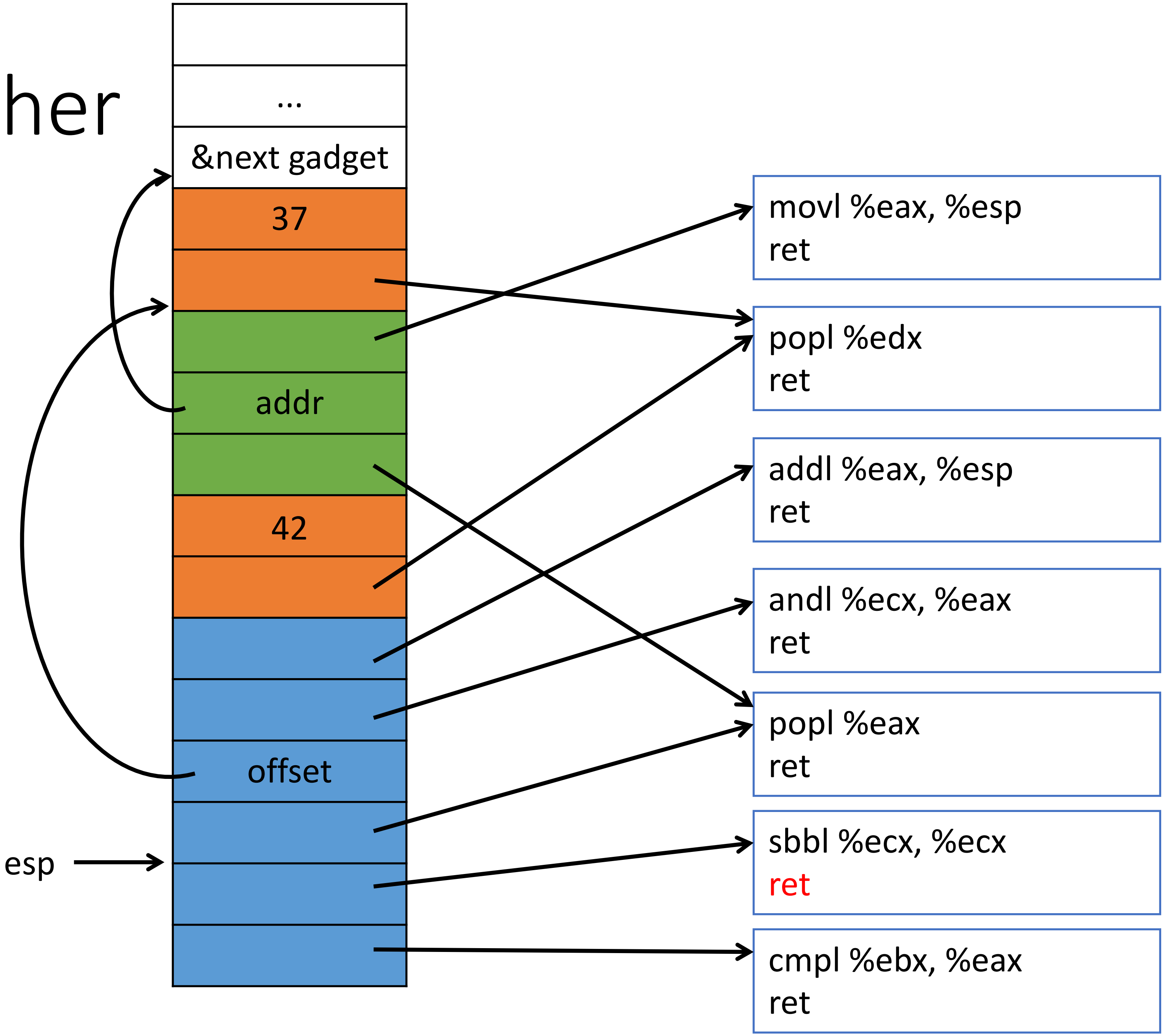
cf = 1



Putting it together

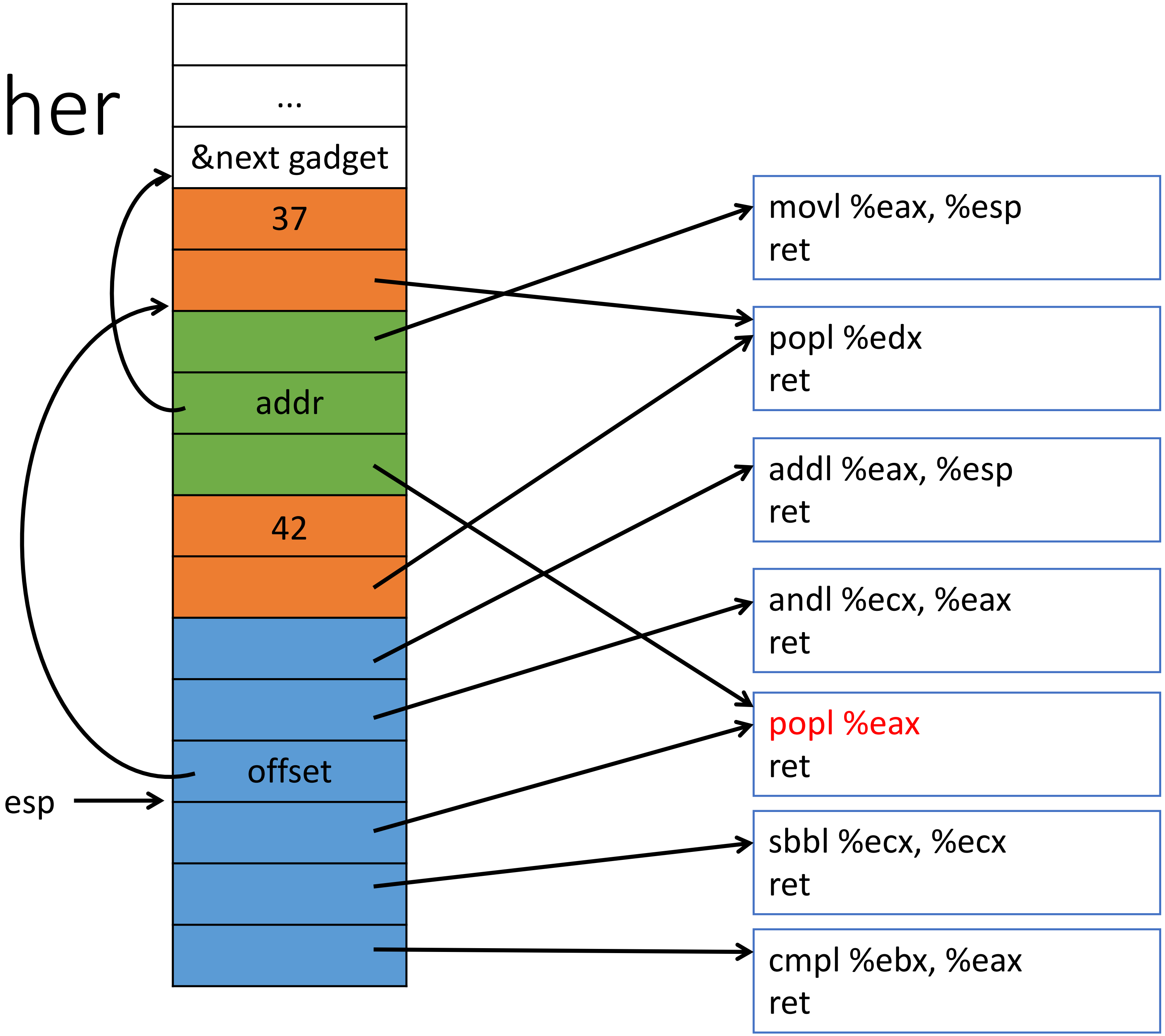
Register	Value
eax	10
ebx	20
ecx	0xffffffff
edx	17

cf = 1



Putting it together

Register	Value
eax	10
ebx	20
ecx	0xffffffff
edx	17



movl %eax, %esp
ret

popl %edx
ret

addl %eax, %esp
ret

andl %ecx, %eax
ret

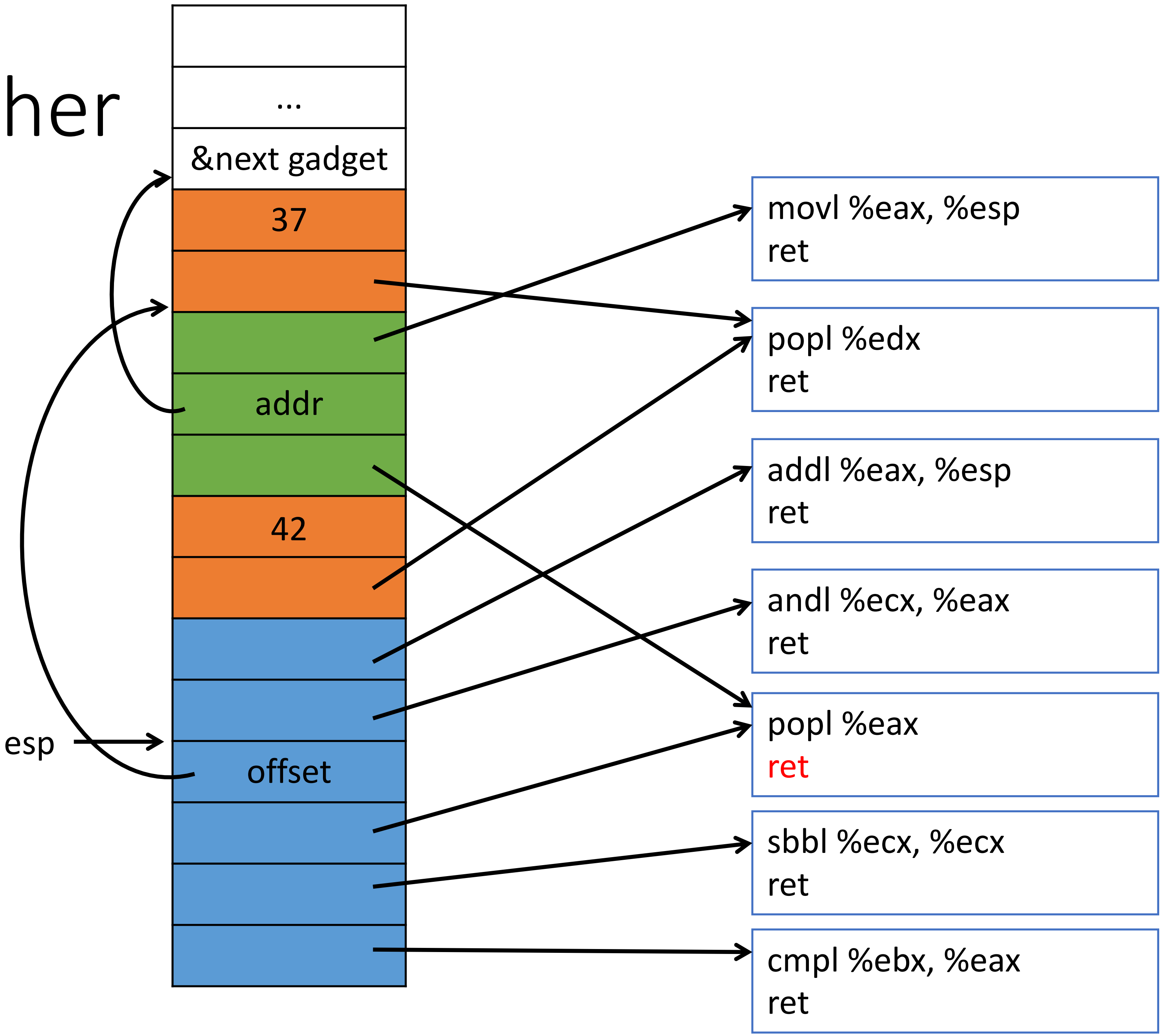
popl %eax
ret

sbbl %ecx, %ecx
ret

cmpl %ebx, %eax
ret

Putting it together

Register	Value
eax	20 = offset
ebx	20
ecx	0xffffffff
edx	17



```
movl %eax, %esp
ret
```

```
popl %edx
ret
```

```
addl %eax, %esp
ret
```

```
andl %ecx, %eax
ret
```

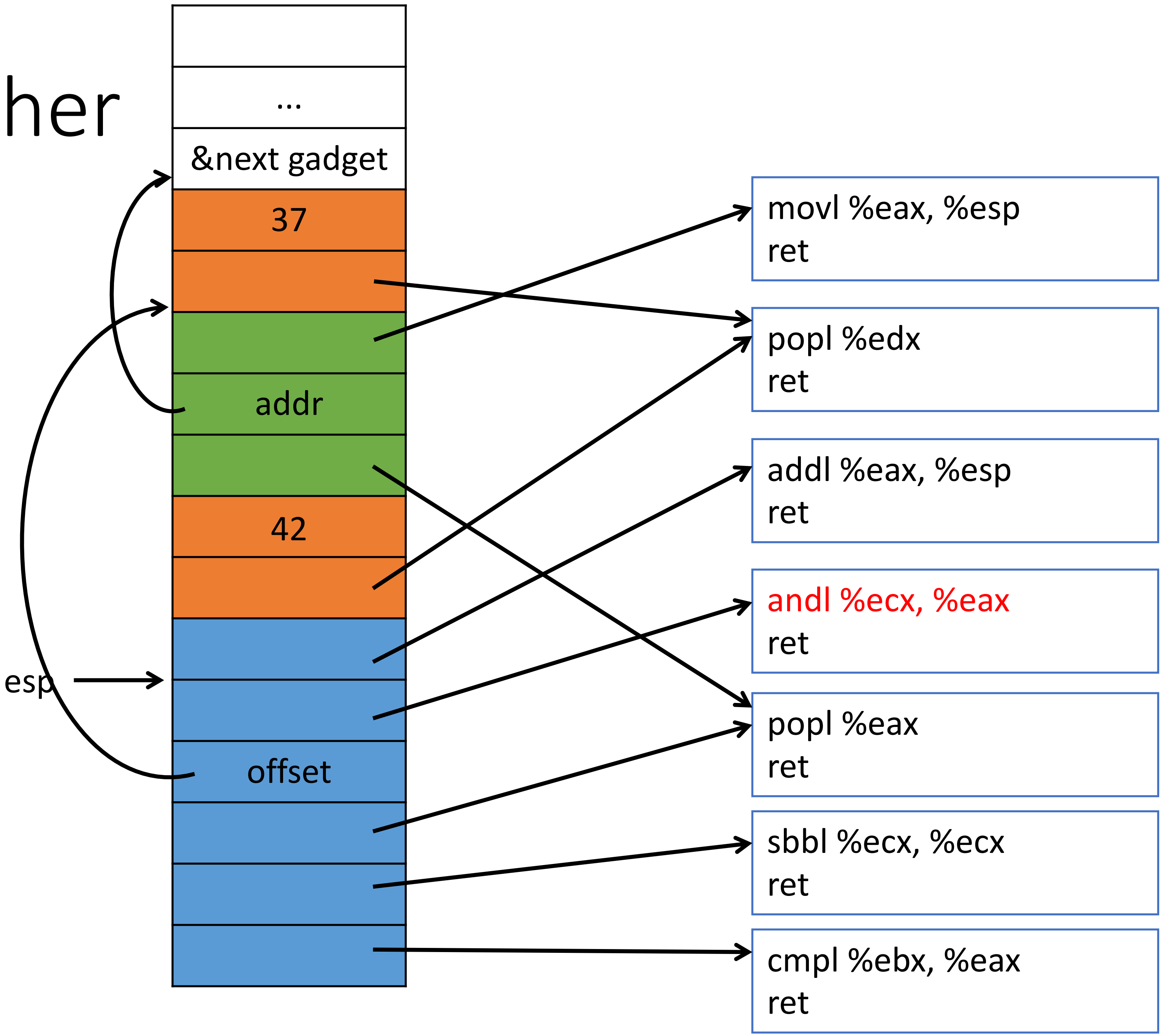
```
popl %eax
ret
```

```
sbbl %ecx, %ecx
ret
```

```
cmpl %ebx, %eax
ret
```

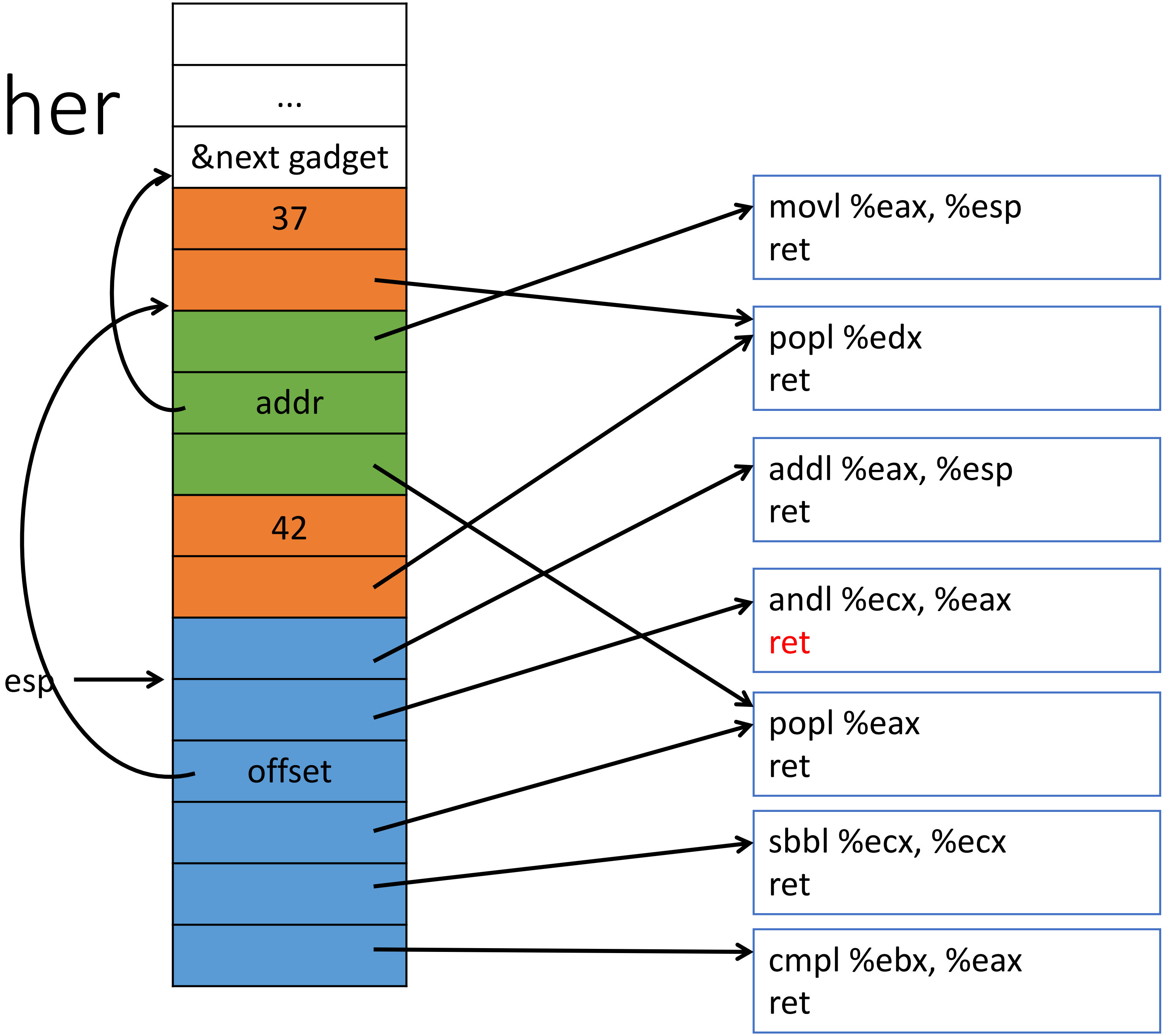
Putting it together

Register	Value
eax	20 = offset
ebx	20
ecx	0xffffffff
edx	17



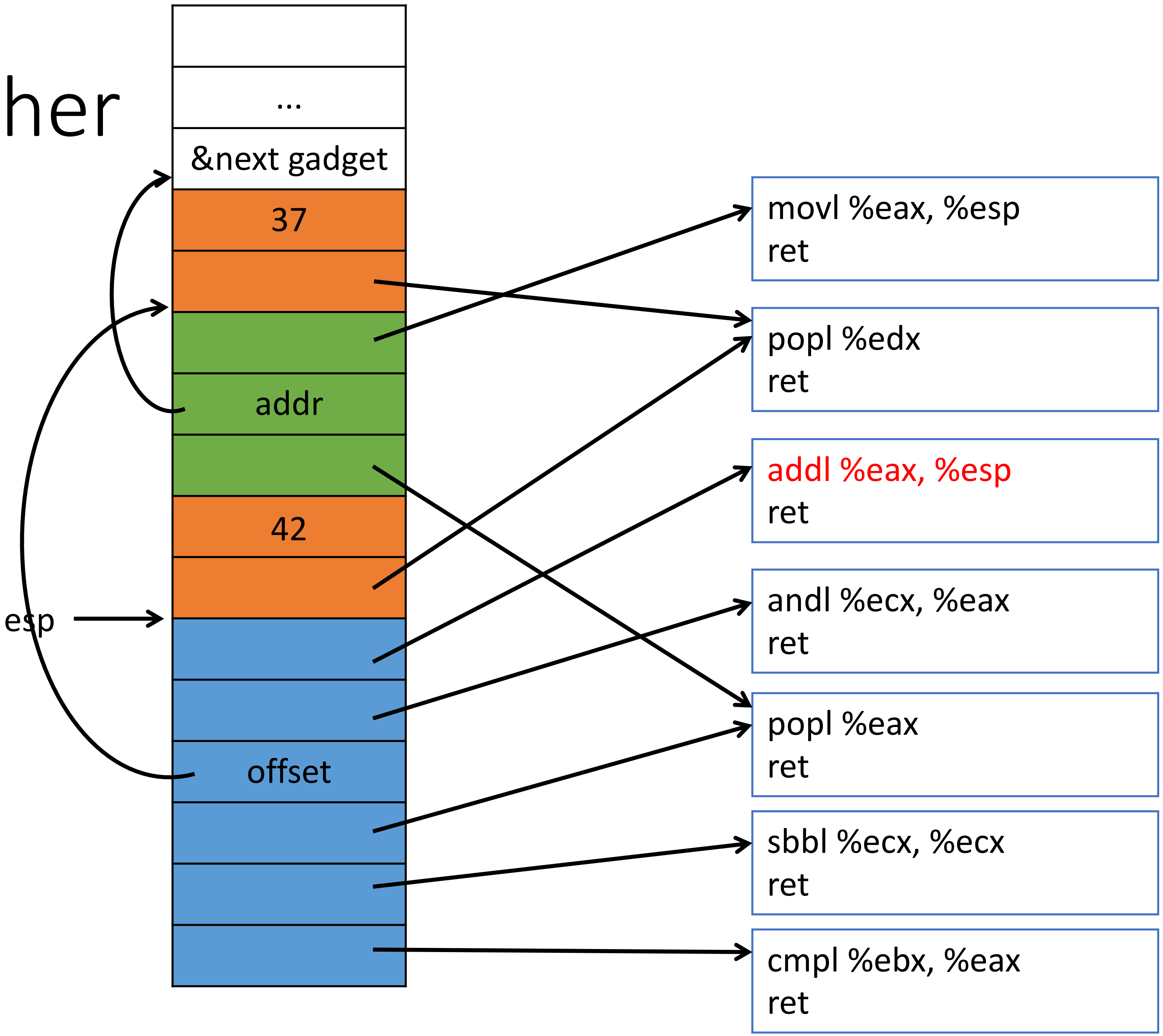
Putting it together

Register	Value
eax	20 = offset
ebx	20
ecx	0xffffffff
edx	17



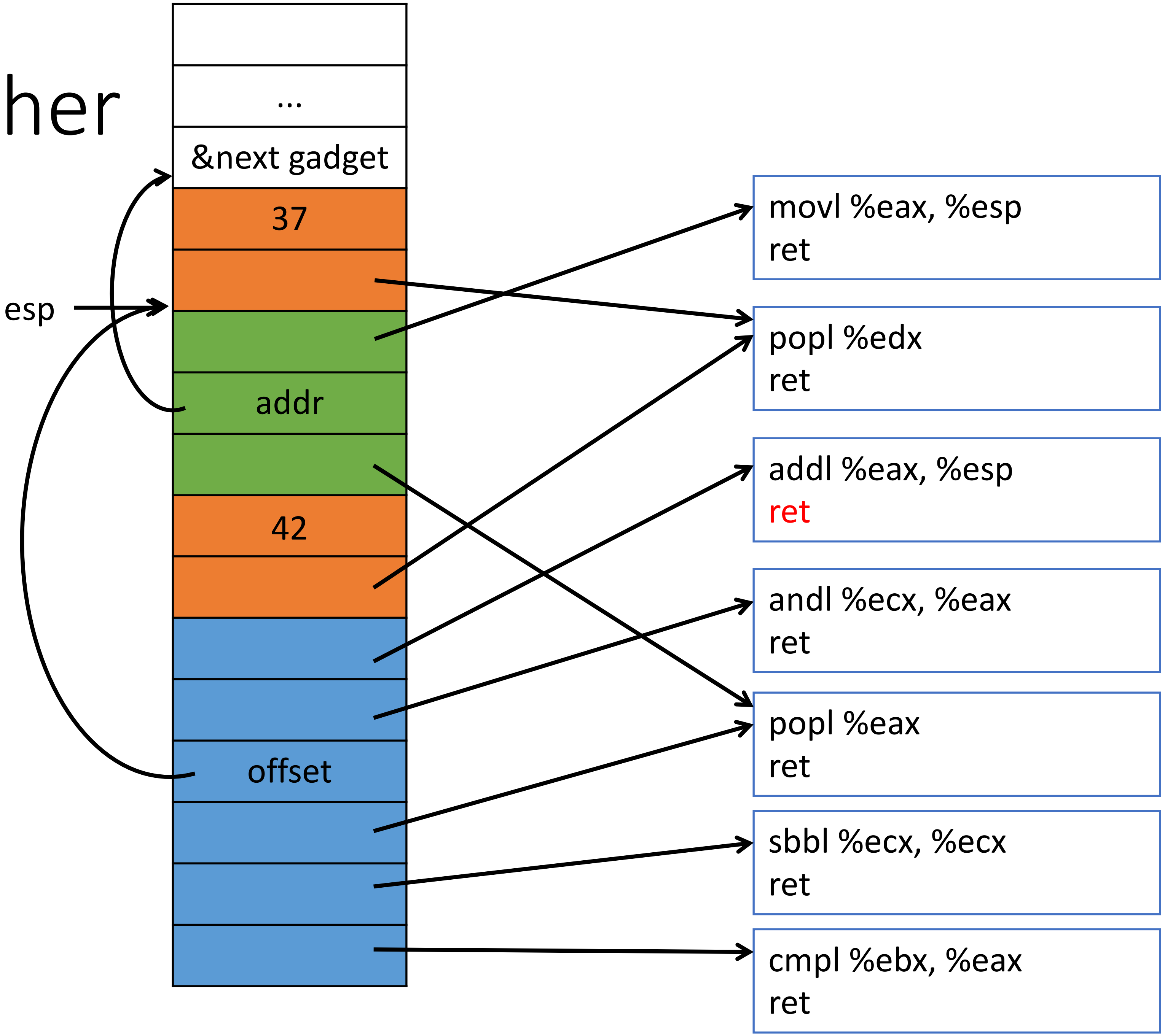
Putting it together

Register	Value
eax	20 = offset
ebx	20
ecx	0xffffffff
edx	17



Putting it together

Register	Value
eax	20 = offset
ebx	20
ecx	0xffffffff
edx	17



```
movl %eax, %esp
ret
```

```
popl %edx
ret
```

```
addl %eax, %esp
ret
```

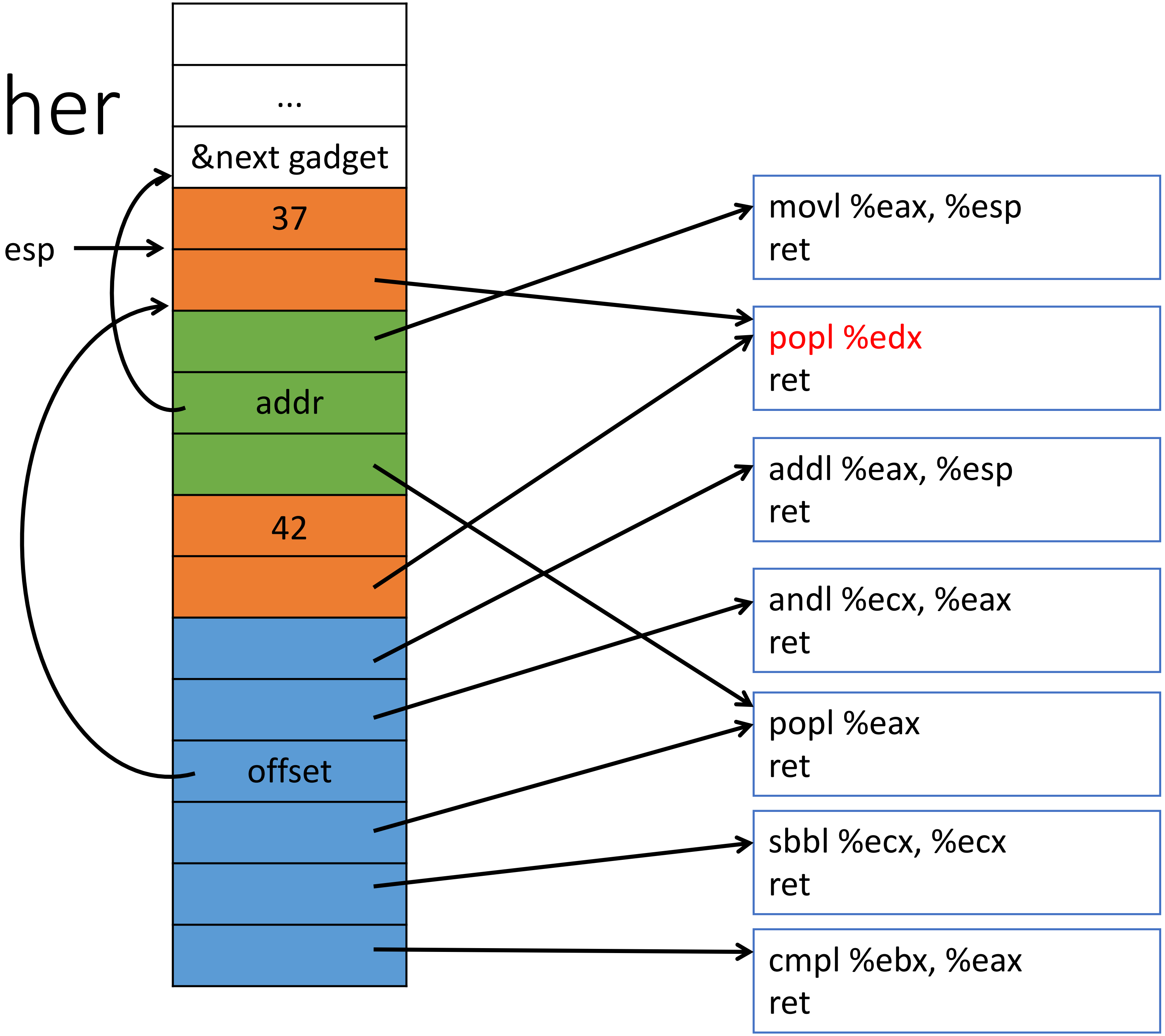
```
andl %ecx, %eax
ret
```

```
popl %eax
ret
```

```
sbbl %ecx, %ecx
ret
```

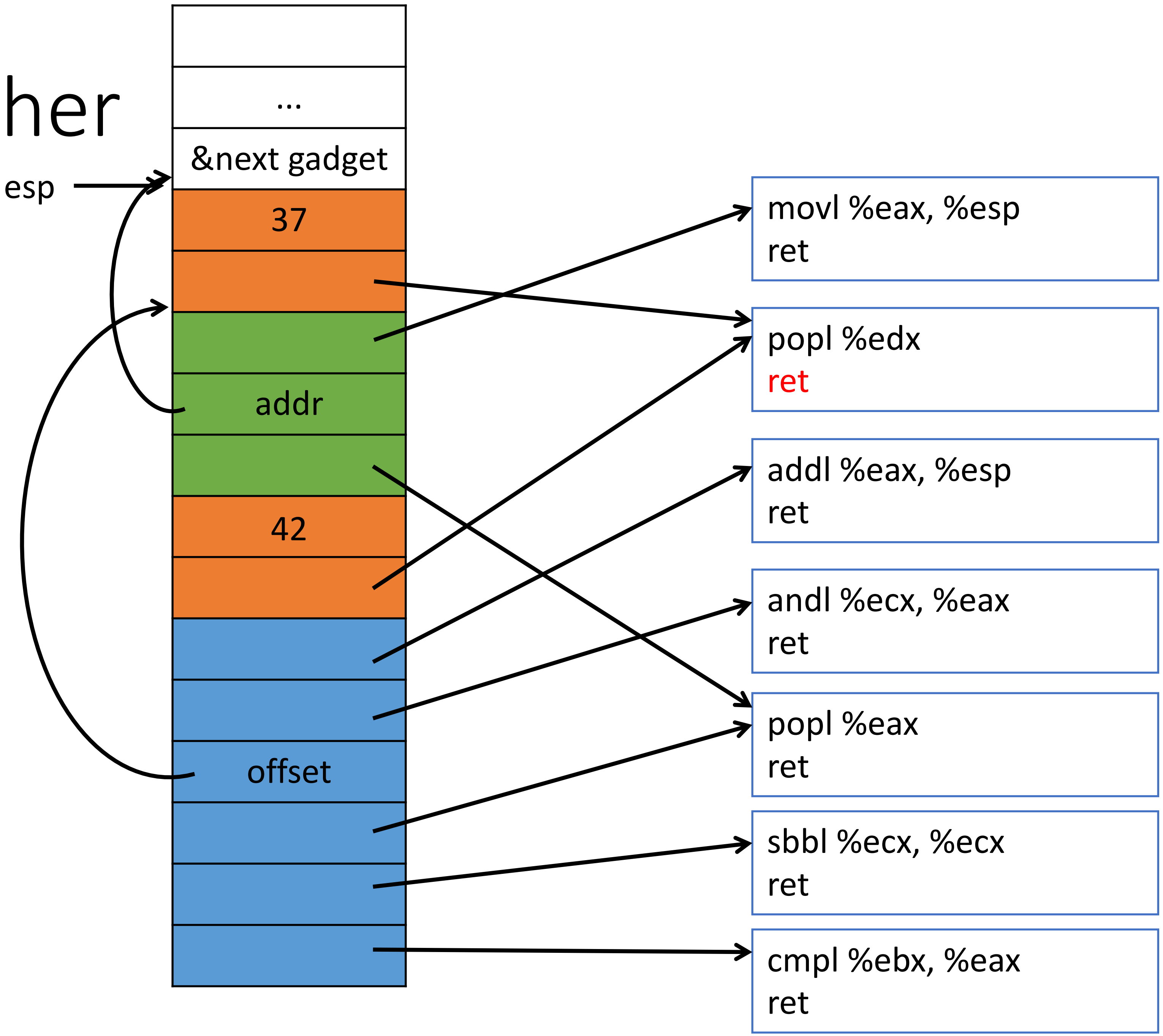
```
cmpl %ebx, %eax
ret
```

Putting it together



Register	Value
eax	20 = offset
ebx	20
ecx	0xffffffff
edx	17

Putting it together



```
movl %eax, %esp
ret
```

```
popl %edx
ret
```

```
addl %eax, %esp
ret
```

```
andl %ecx, %eax
ret
```

```
popl %eax
ret
```

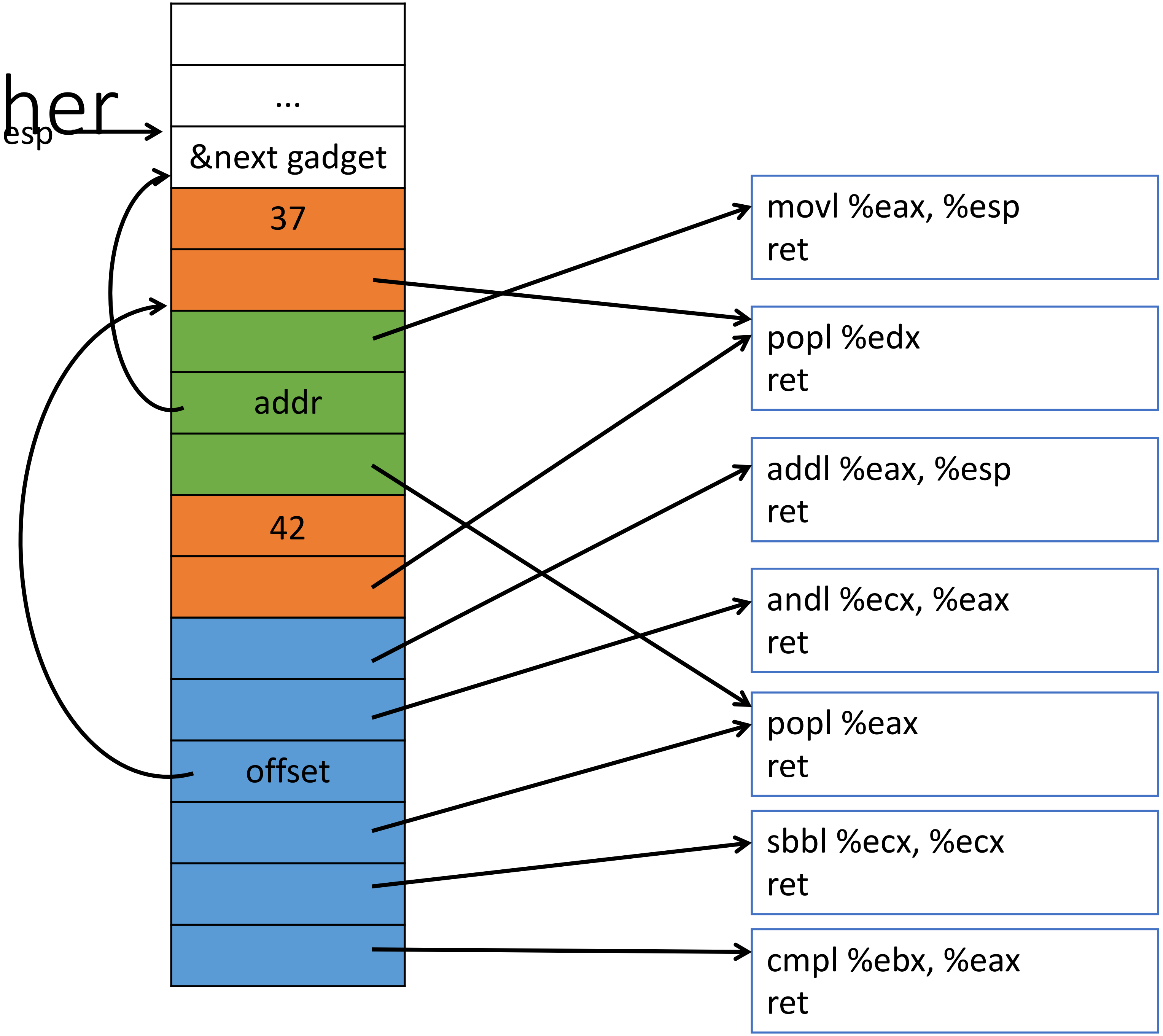
```
sbbl %ecx, %ecx
ret
```

```
cmpl %ebx, %eax
ret
```

Register	Value
eax	20 = offset
ebx	20
ecx	0xffffffff
edx	37

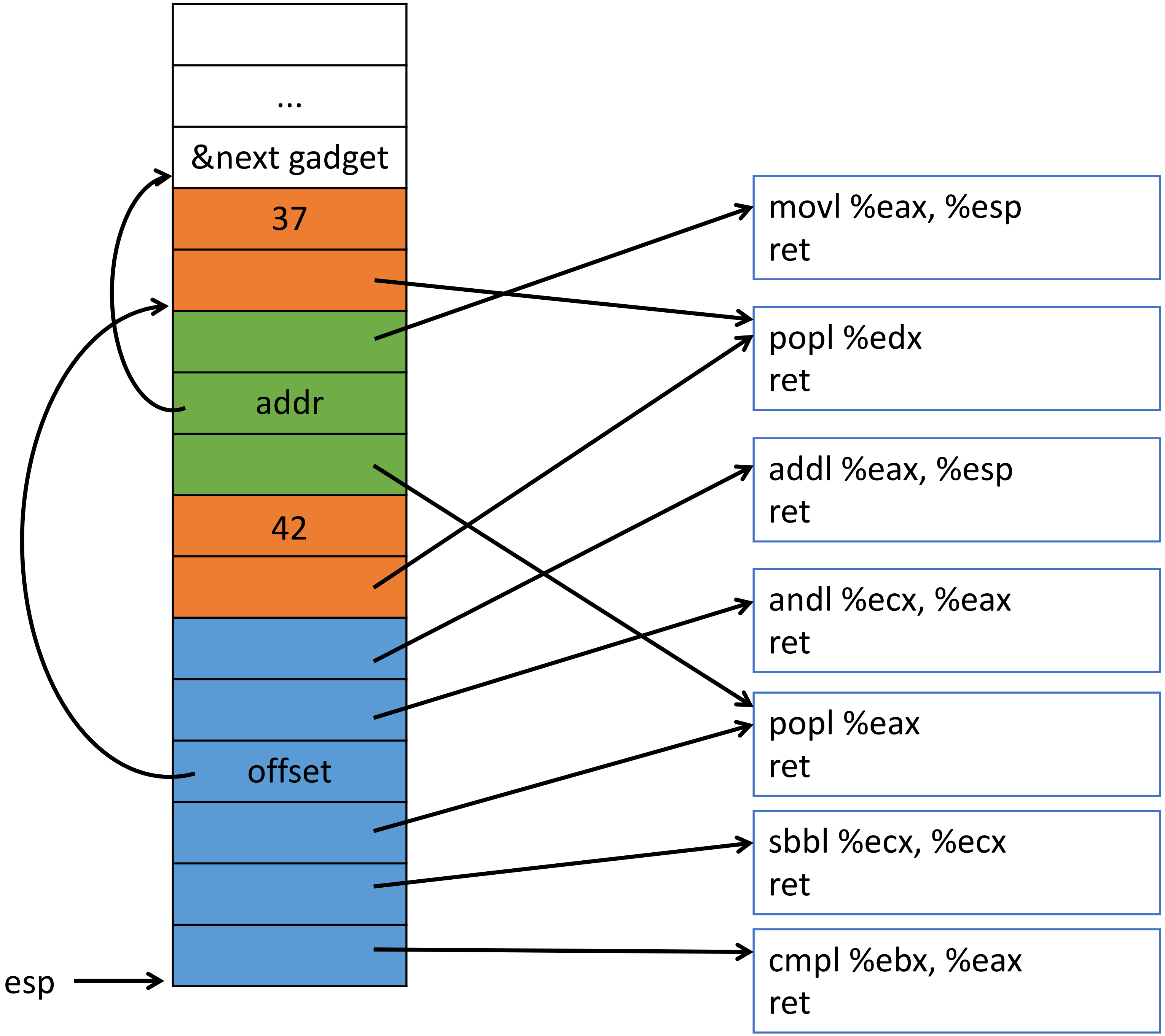
Putting it together

Register	Value
eax	20 = offset
ebx	20
ecx	0xffffffff
edx	37



And again!

Register	Value
eax	500
ebx	20
ecx	108
edx	17



```
movl %eax, %esp
ret
```

```
popl %edx
ret
```

```
addl %eax, %esp
ret
```

```
andl %ecx, %eax
ret
```

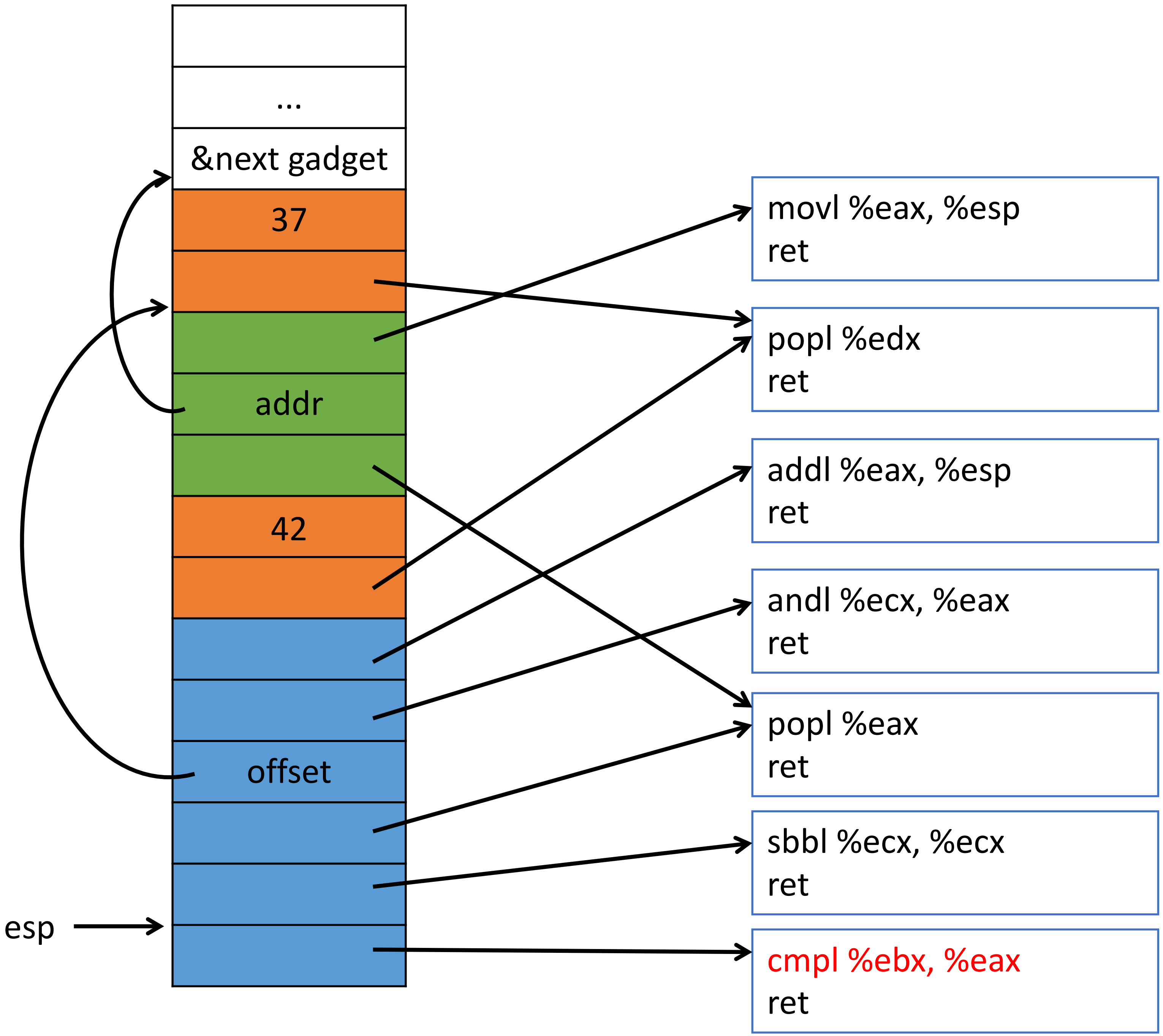
```
popl %eax
ret
```

```
sbbl %ecx, %ecx
ret
```

```
cmpl %ebx, %eax
ret
```

And again!

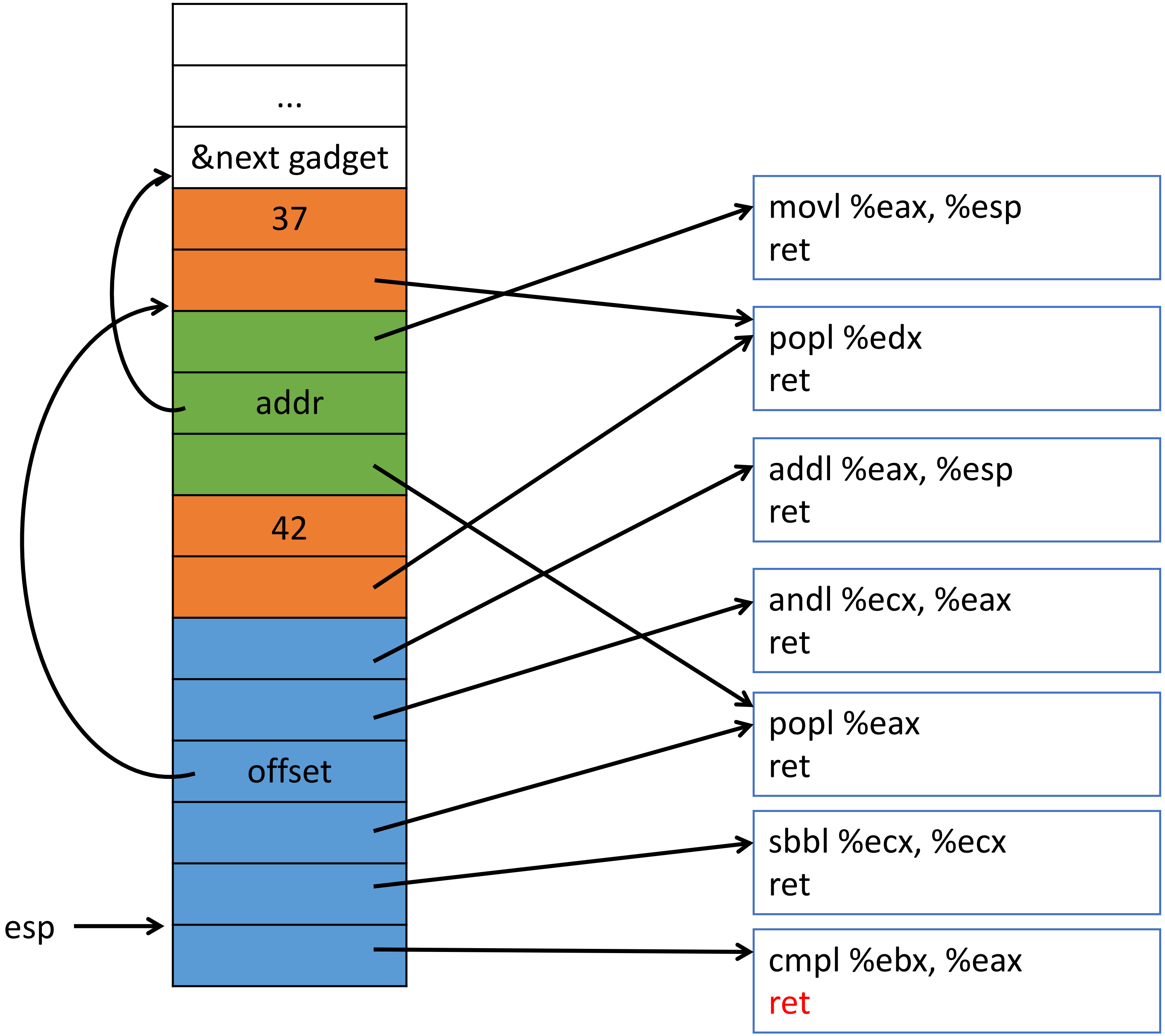
Register	Value
eax	500
ebx	20
ecx	108
edx	17



And again!

Register	Value
eax	500
ebx	20
ecx	108
edx	17

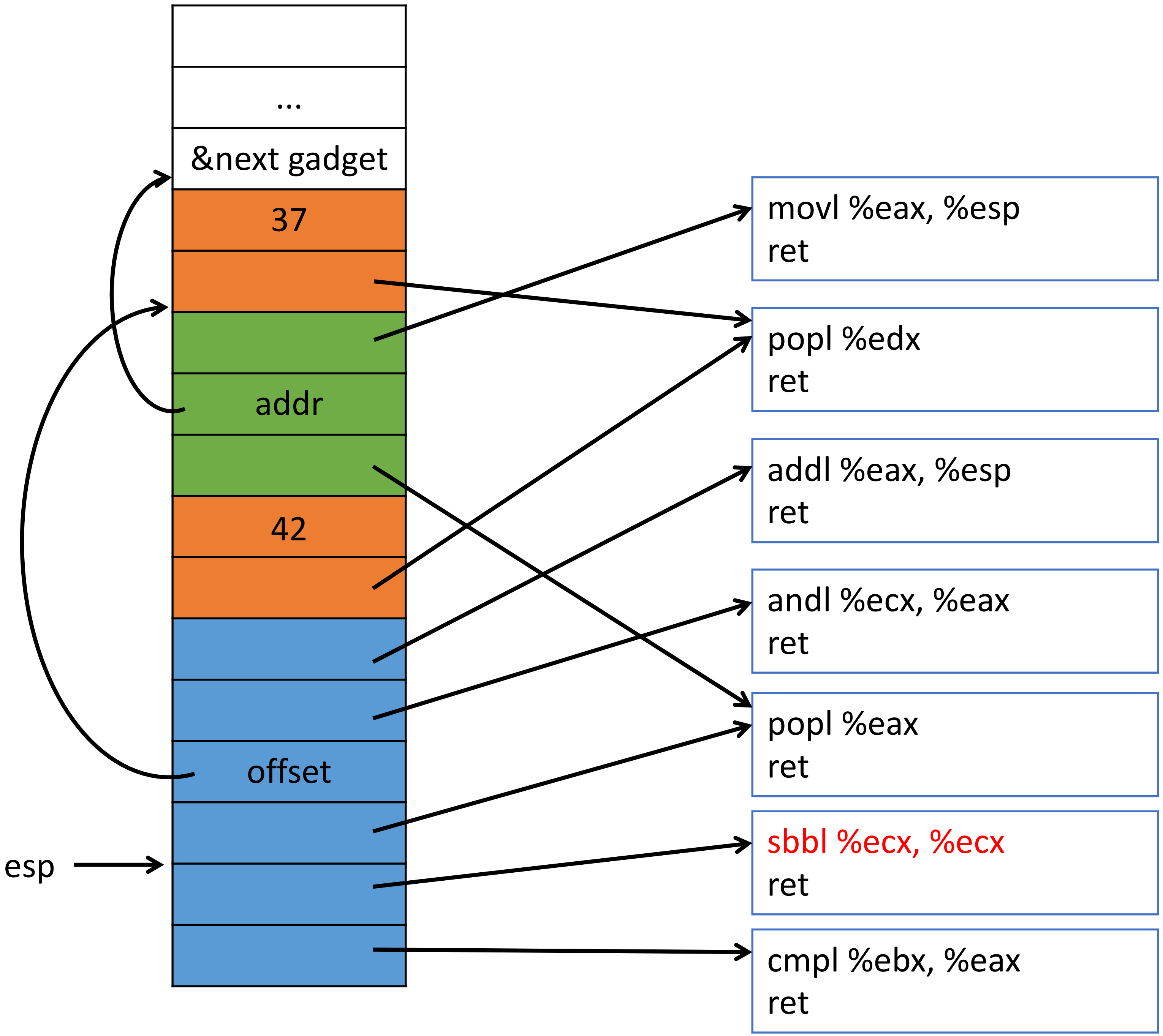
cf = 0



And again!

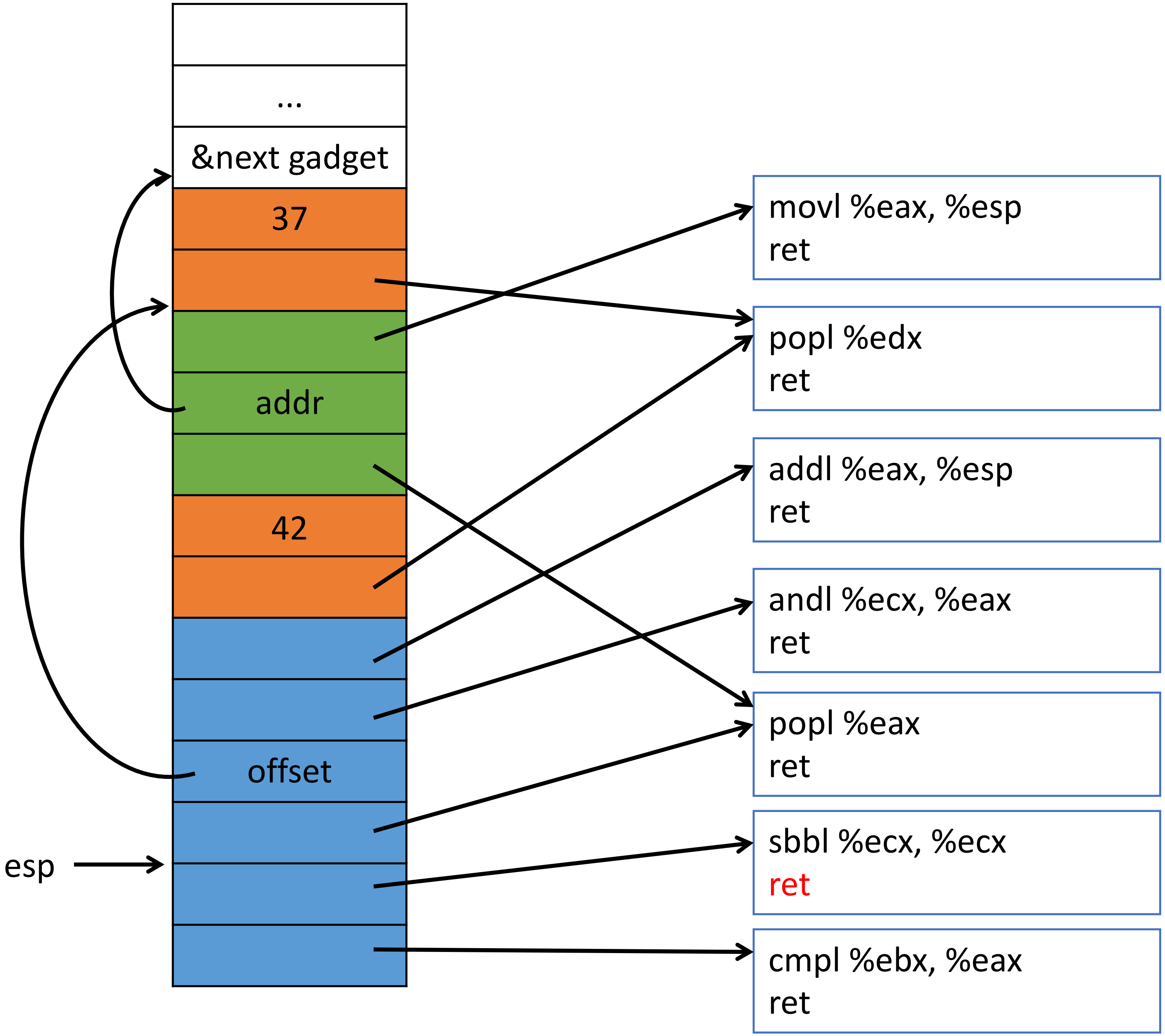
Register	Value
eax	500
ebx	20
ecx	108
edx	17

cf = 0



And again!

Register	Value
eax	500
ebx	20
ecx	0
edx	17



```
movl %eax, %esp
ret
```

```
popl %edx
ret
```

```
addl %eax, %esp
ret
```

```
andl %ecx, %eax
ret
```

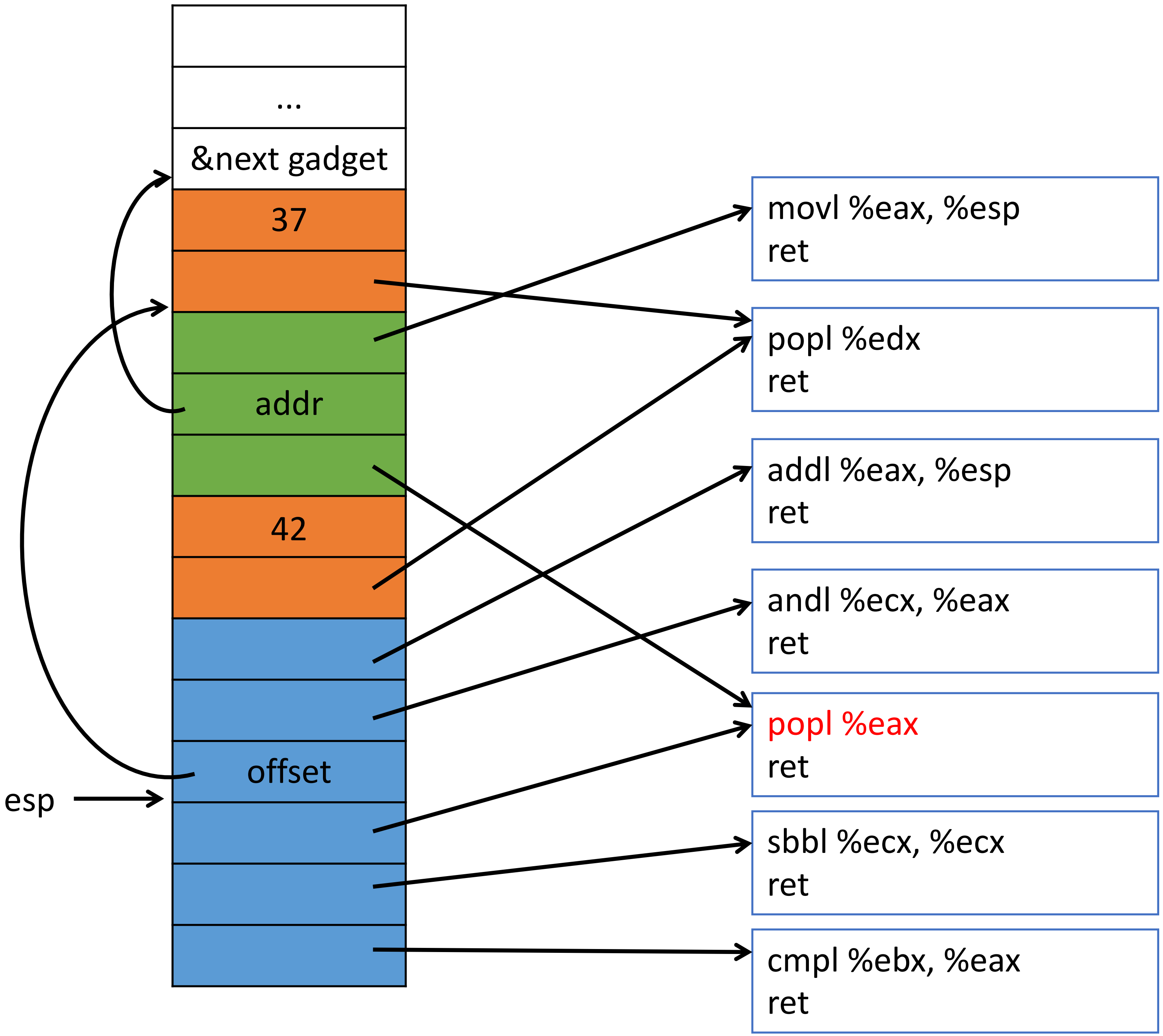
```
popl %eax
ret
```

```
sbbl %ecx, %ecx
ret
```

```
cmpl %ebx, %eax
ret
```

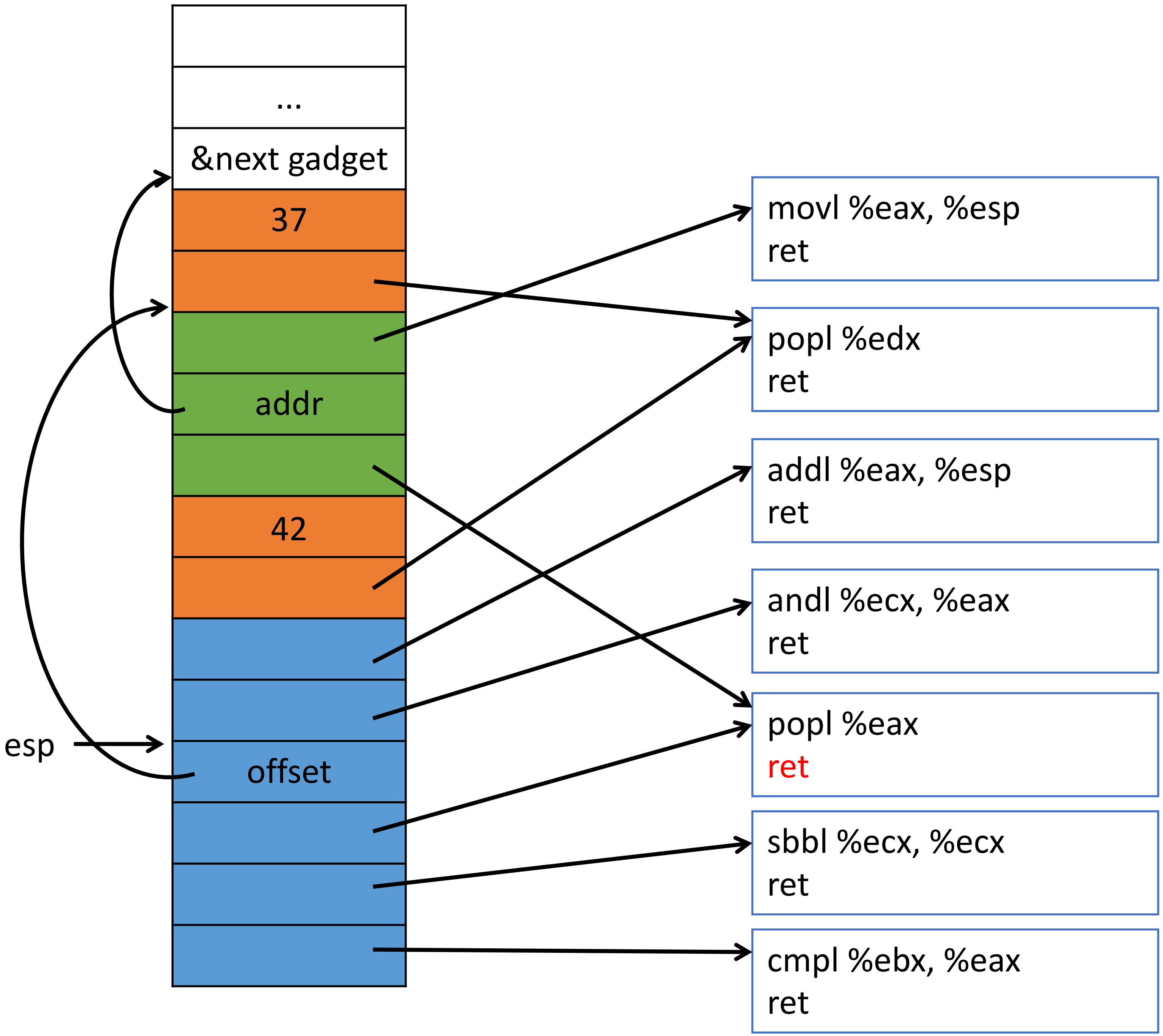
And again!

Register	Value
eax	500
ebx	20
ecx	0
edx	17



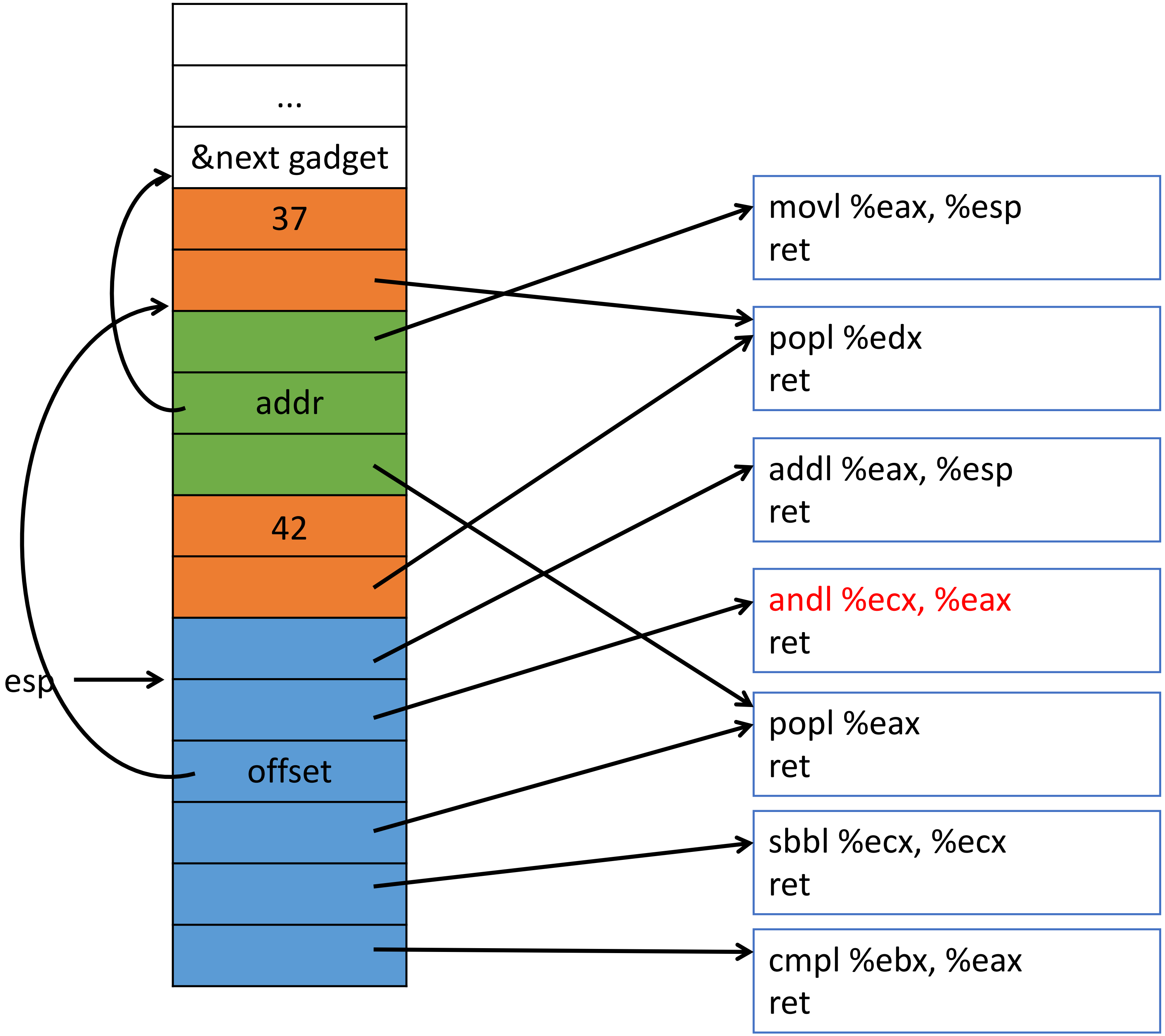
And again!

Register	Value
eax	20 = offset
ebx	20
ecx	0
edx	17



And again!

Register	Value
eax	20 = offset
ebx	20
ecx	0
edx	17



```
movl %eax, %esp
ret
```

```
popl %edx
ret
```

```
addl %eax, %esp
ret
```

```
andl %ecx, %eax
ret
```

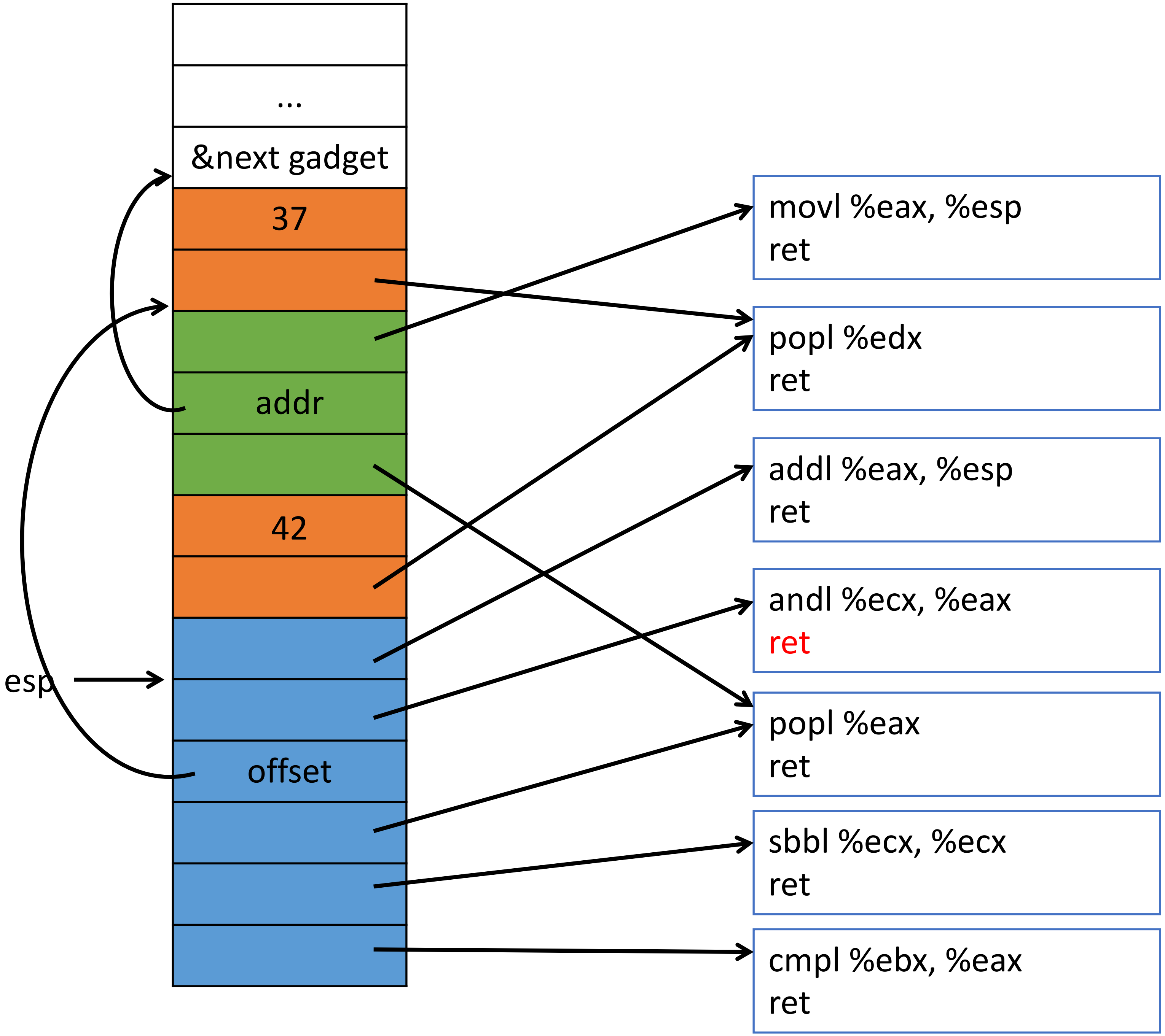
```
popl %eax
ret
```

```
sbbl %ecx, %ecx
ret
```

```
cmpl %ebx, %eax
ret
```

And again!

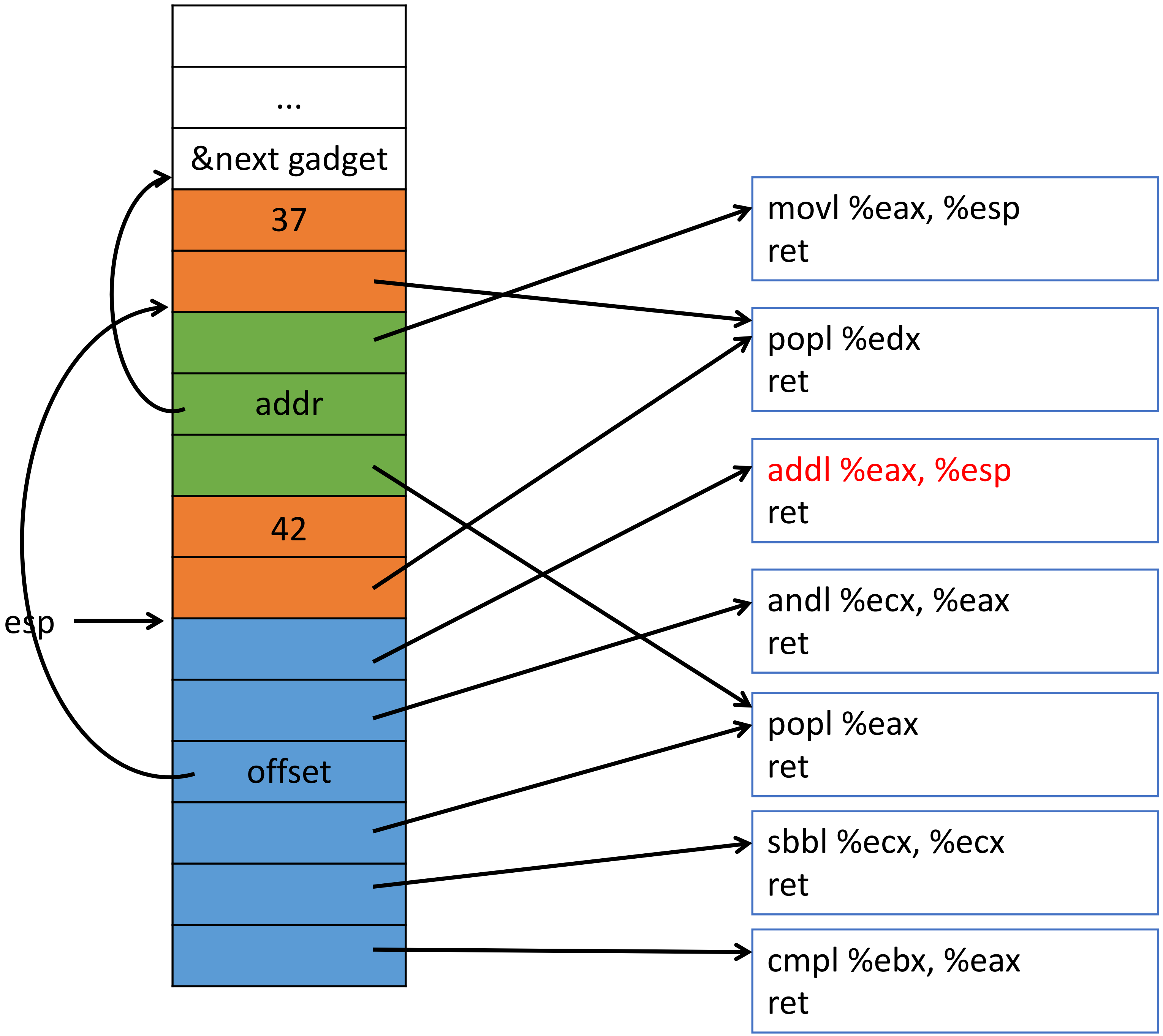
Register	Value
eax	0
ebx	20
ecx	0
edx	17



```
movl %eax, %esp
ret

popl %edx
ret

addl %eax, %esp
ret

andl %ecx, %eax
ret

popl %eax
ret

sbbl %ecx, %ecx
ret

cmpl %ebx, %eax
ret
```

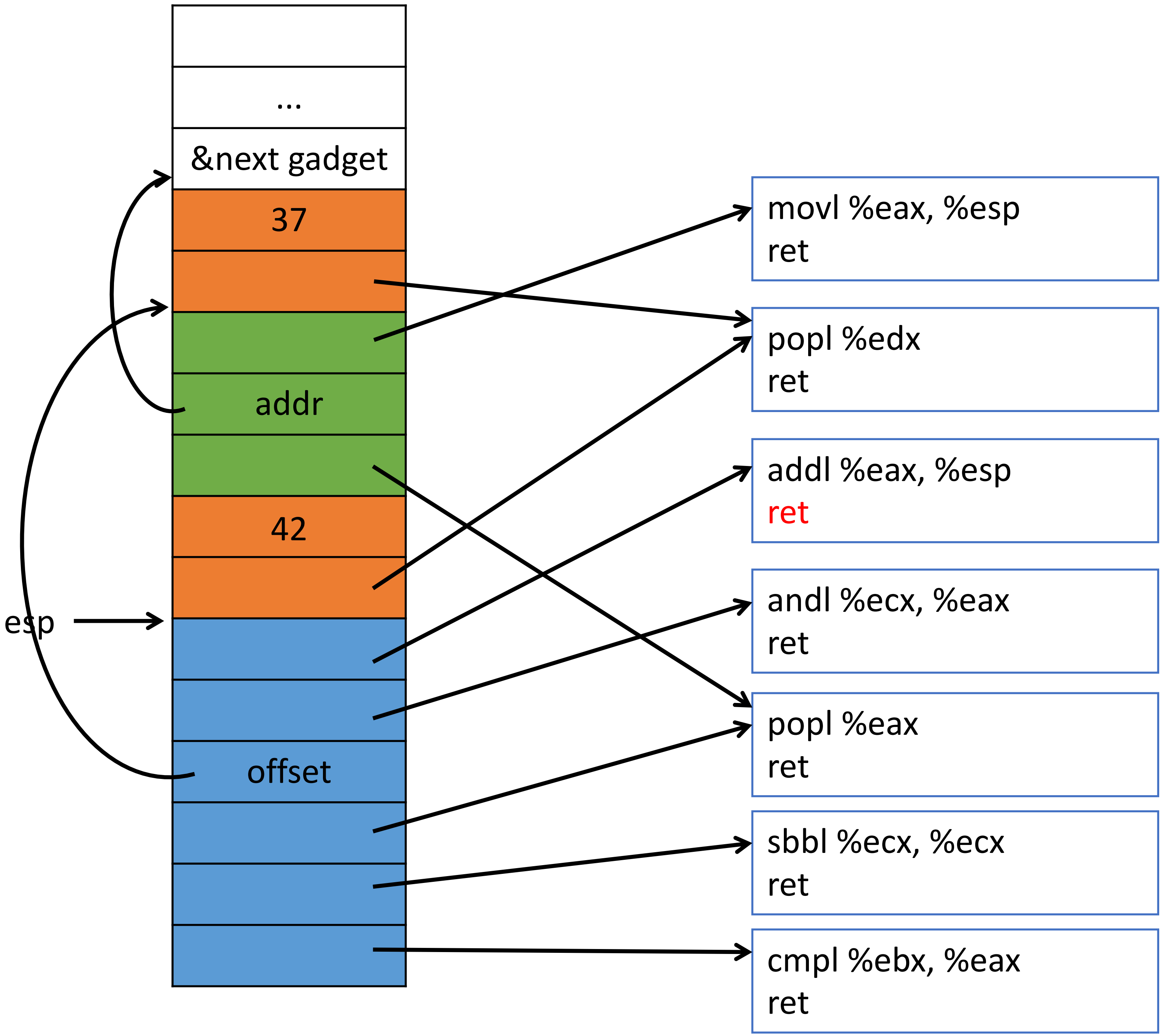
And again!

Register	Value
eax	0
ebx	20
ecx	0
edx	17



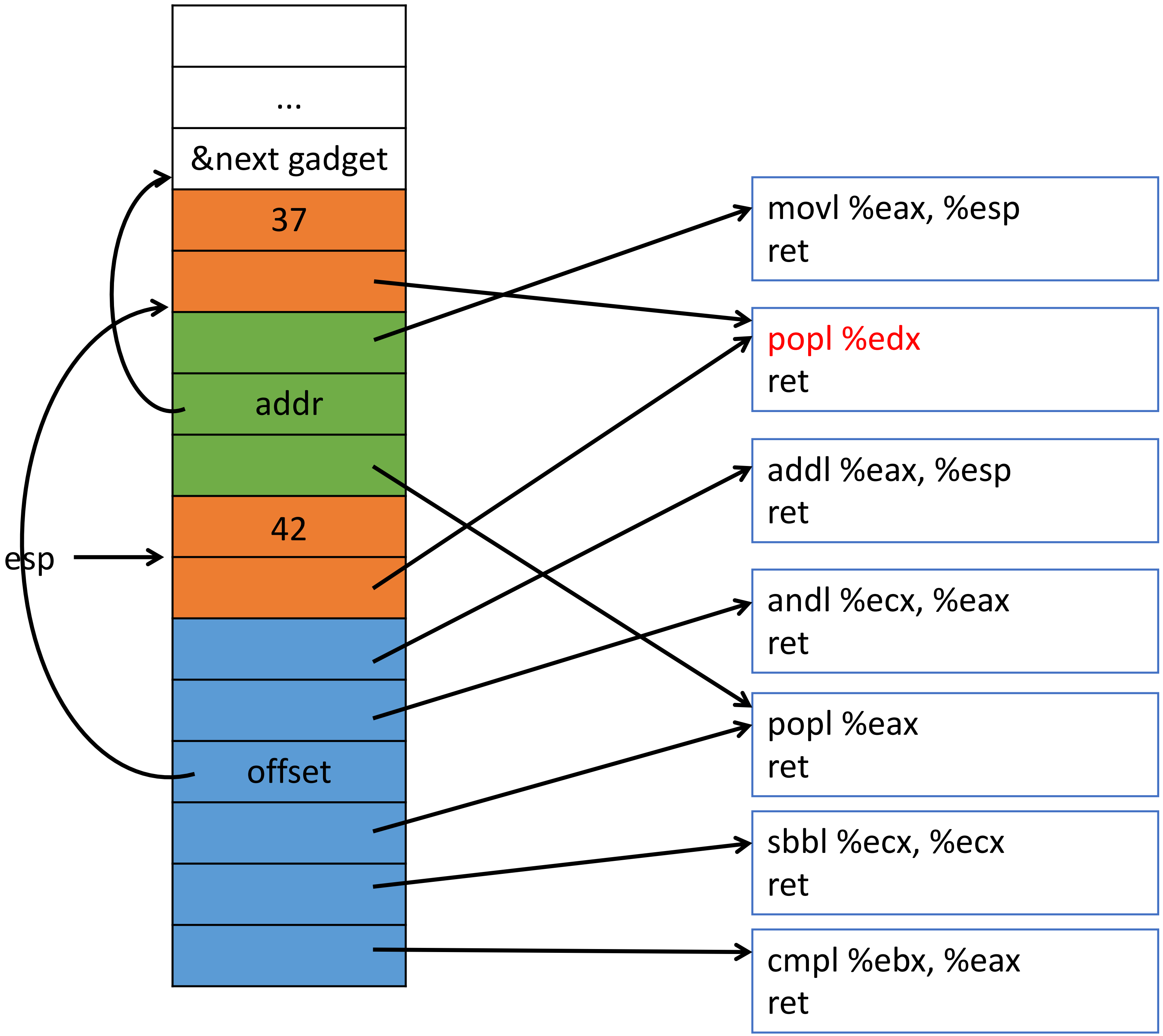
And again!

Register	Value
eax	0
ebx	20
ecx	0
edx	17



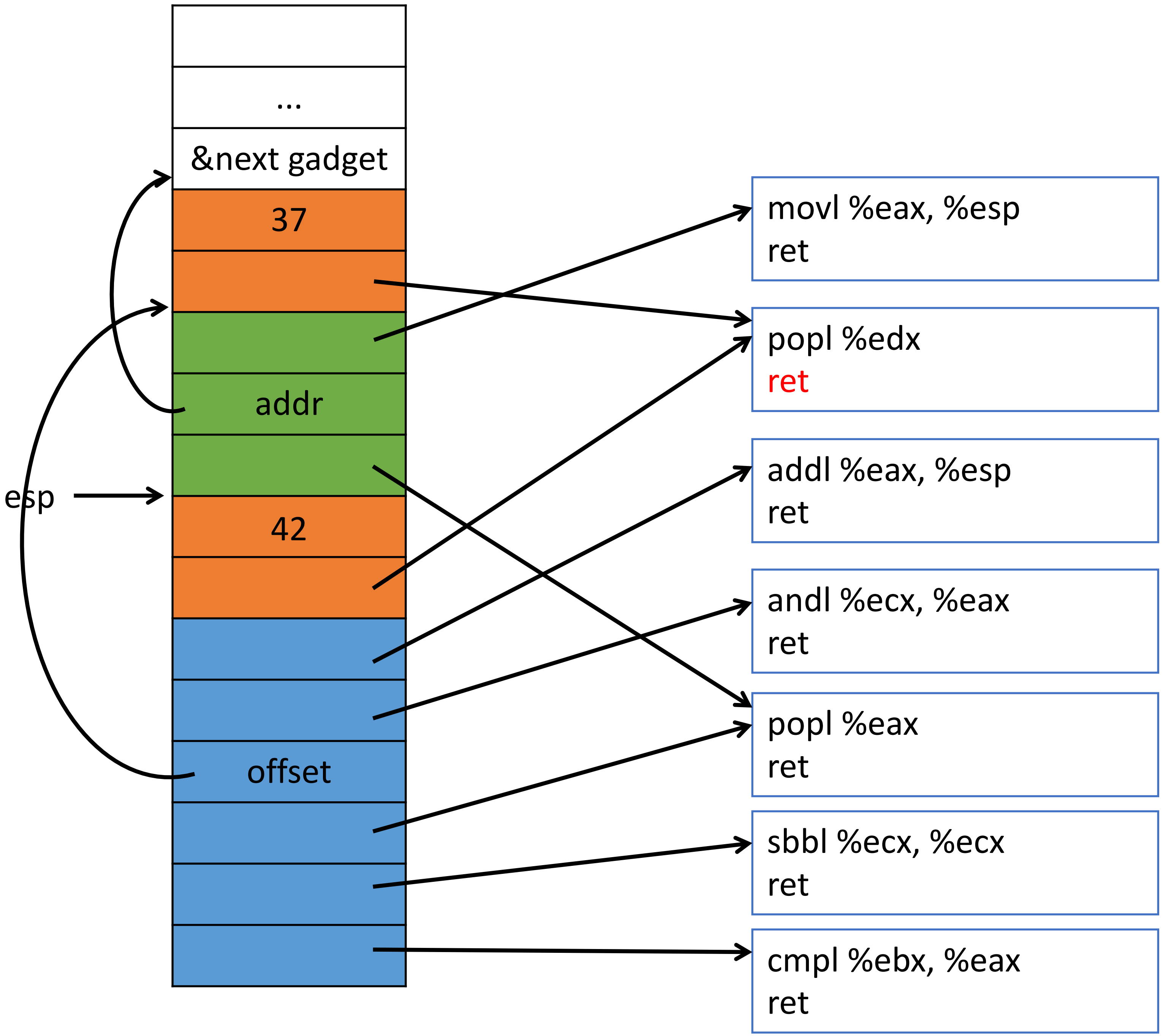
And again!

Register	Value
eax	0
ebx	20
ecx	0
edx	17



And again!

Register	Value
eax	0
ebx	20
ecx	0
edx	42



```
movl %eax, %esp
ret
```

```
popl %edx
ret
```

```
addl %eax, %esp
ret
```

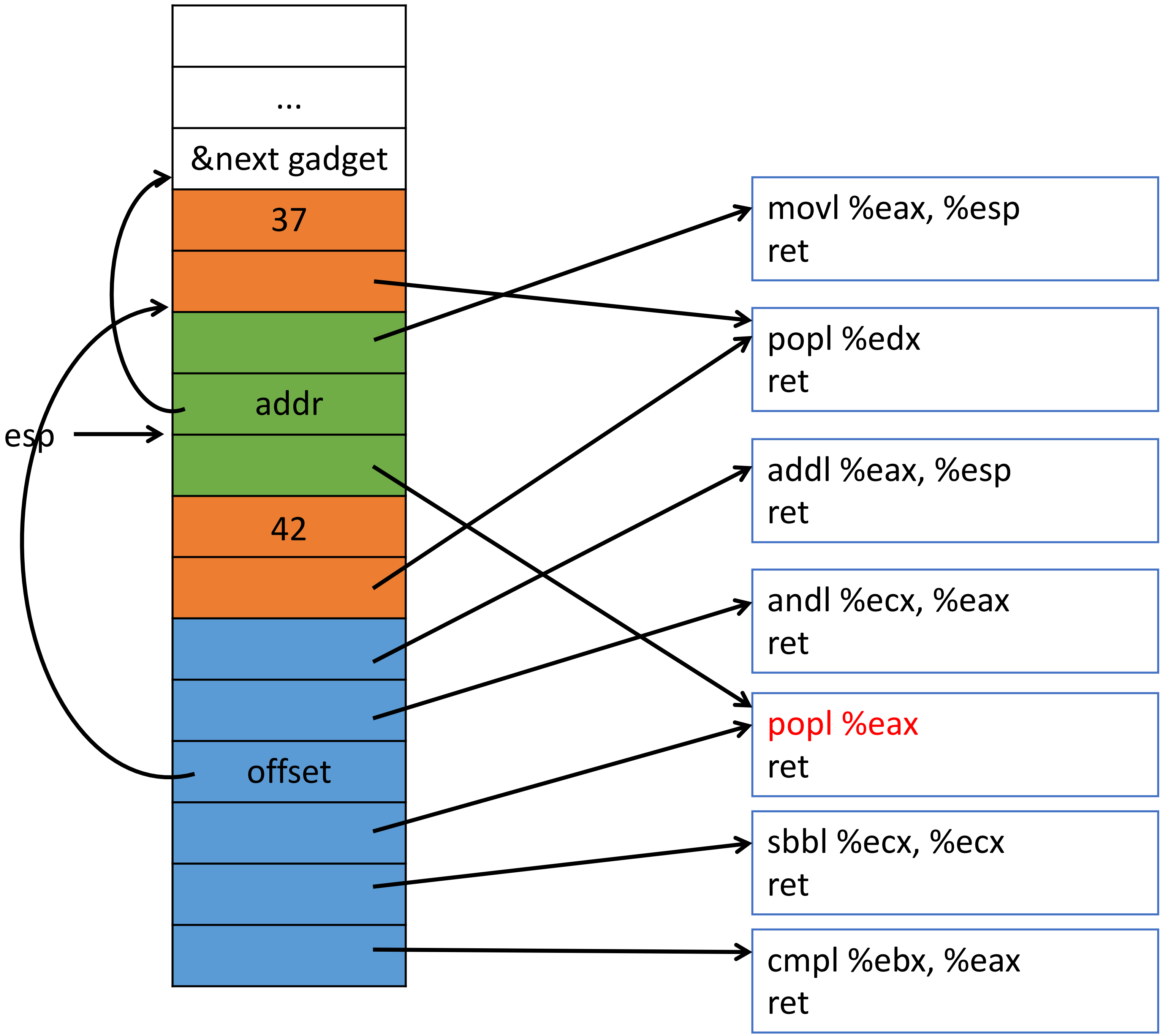
```
andl %ecx, %eax
ret
```

```
popl %eax
ret
```

```
sbbl %ecx, %ecx
ret
```

```
cmpl %ebx, %eax
ret
```

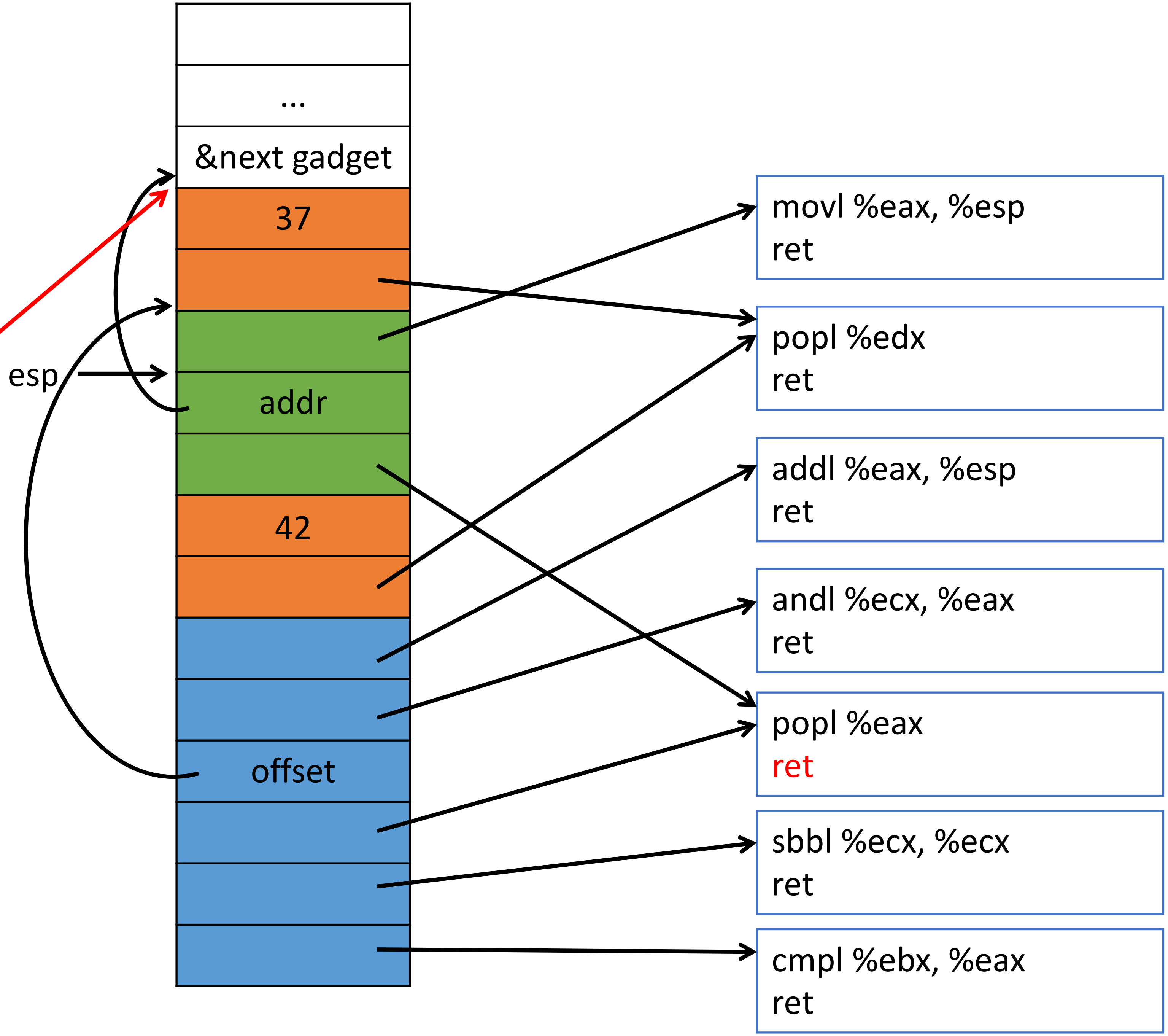
And again!



Register	Value
eax	0
ebx	20
ecx	0
edx	42

And again!

Register	Value
eax	addr
ebx	20
ecx	0
edx	42



```
movl %eax, %esp
ret
```

```
popl %edx
ret
```

```
addl %eax, %esp
ret
```

```
andl %ecx, %eax
ret
```

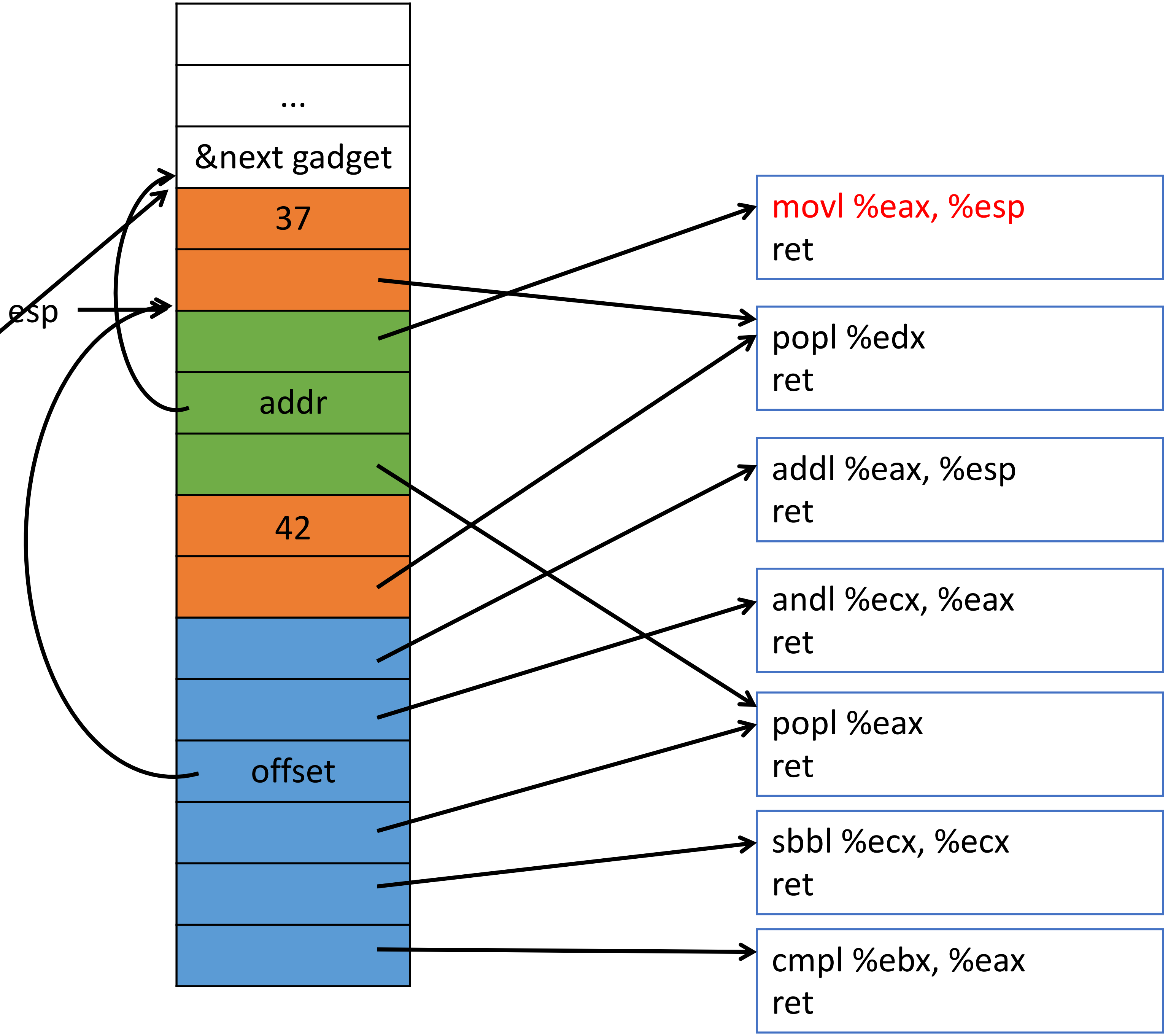
```
popl %eax
ret
```

```
sbbl %ecx, %ecx
ret
```

```
cmpl %ebx, %eax
ret
```

And again!

Register	Value
eax	addr
ebx	20
ecx	0
edx	42



esp

…

&next gadget

37

addr

42

offset

```
movl %eax, %esp
ret
```

```
popl %edx
ret
```

```
addl %eax, %esp
ret
```

```
andl %ecx, %eax
ret
```

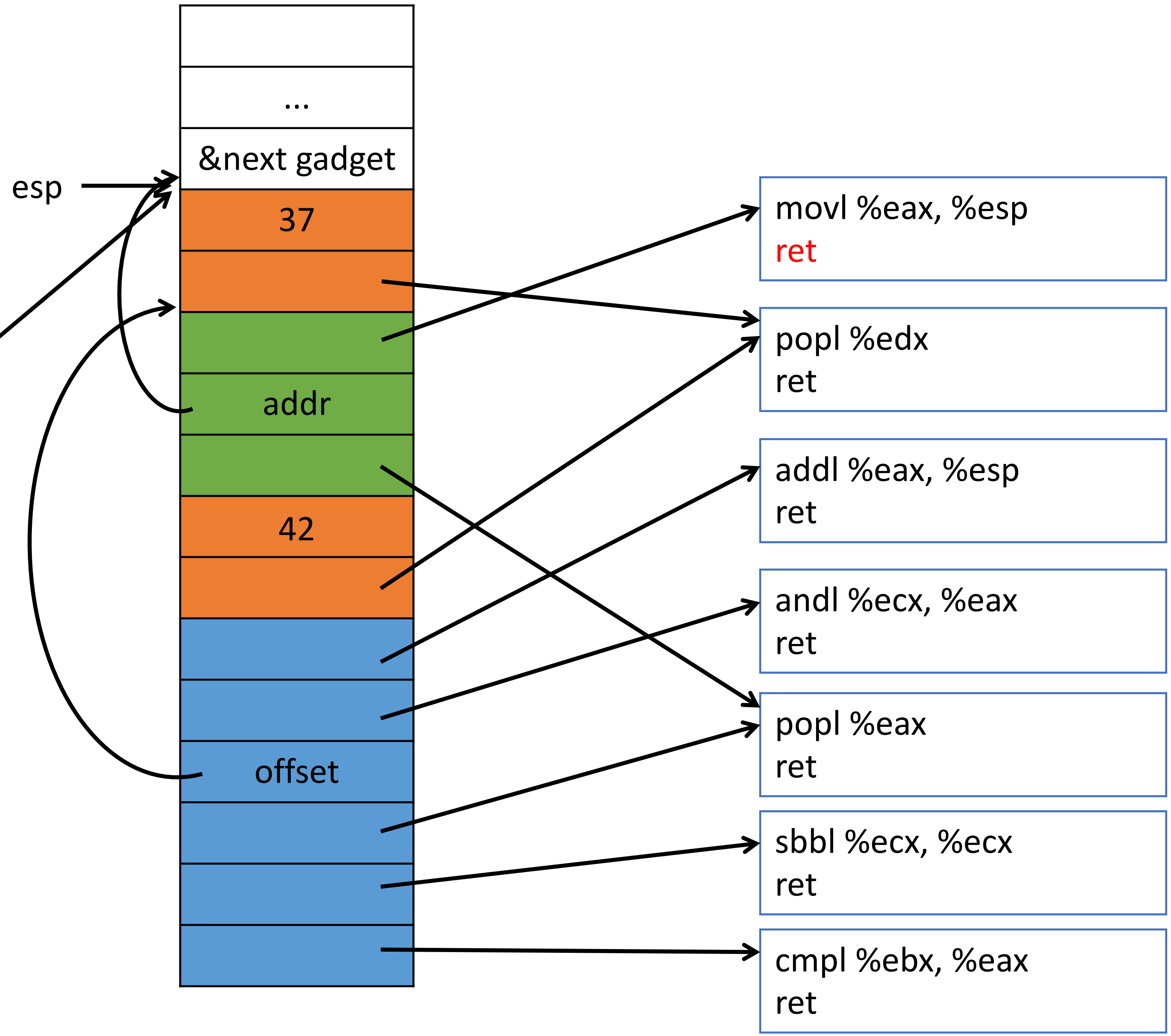
```
popl %eax
ret
```

```
sbbl %ecx, %ecx
ret
```

```
cmpl %ebx, %eax
ret
```

And again!

Register	Value
eax	addr
ebx	20
ecx	0
edx	42



```
movl %eax, %esp
ret
```

```
popl %edx
ret
```

```
addl %eax, %esp
ret
```

```
andl %ecx, %eax
ret
```

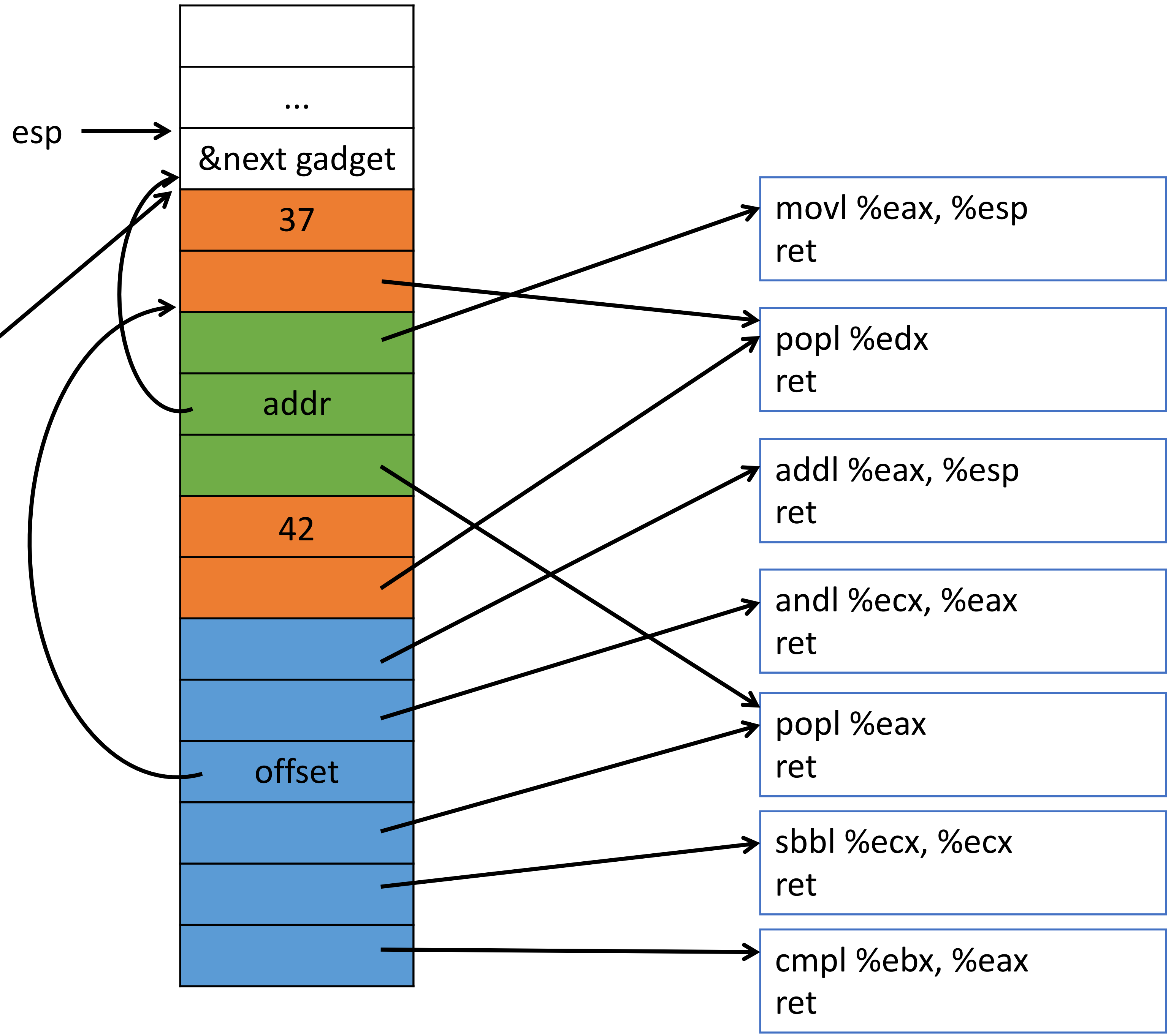
```
popl %eax
ret
```

```
sbbl %ecx, %ecx
ret
```

```
cmpl %ebx, %eax
ret
```

And again!

Register	Value
eax	addr
ebx	20
ecx	0
edx	42



Compare

Register	Value
eax	10
ebx	20
ecx	108
edx	17

Register	Value
eax	20
ebx	20
ecx	0xffffffff
edx	37

```
if (eax < ebx)
        edx = 37;
else
        edx = 42;
```

Register	Value
eax	500
ebx	20
ecx	108
edx	17

Register	Value
eax	addr
ebx	20
ecx	0
edx	42