Lecture 14 – Return-oriented programming

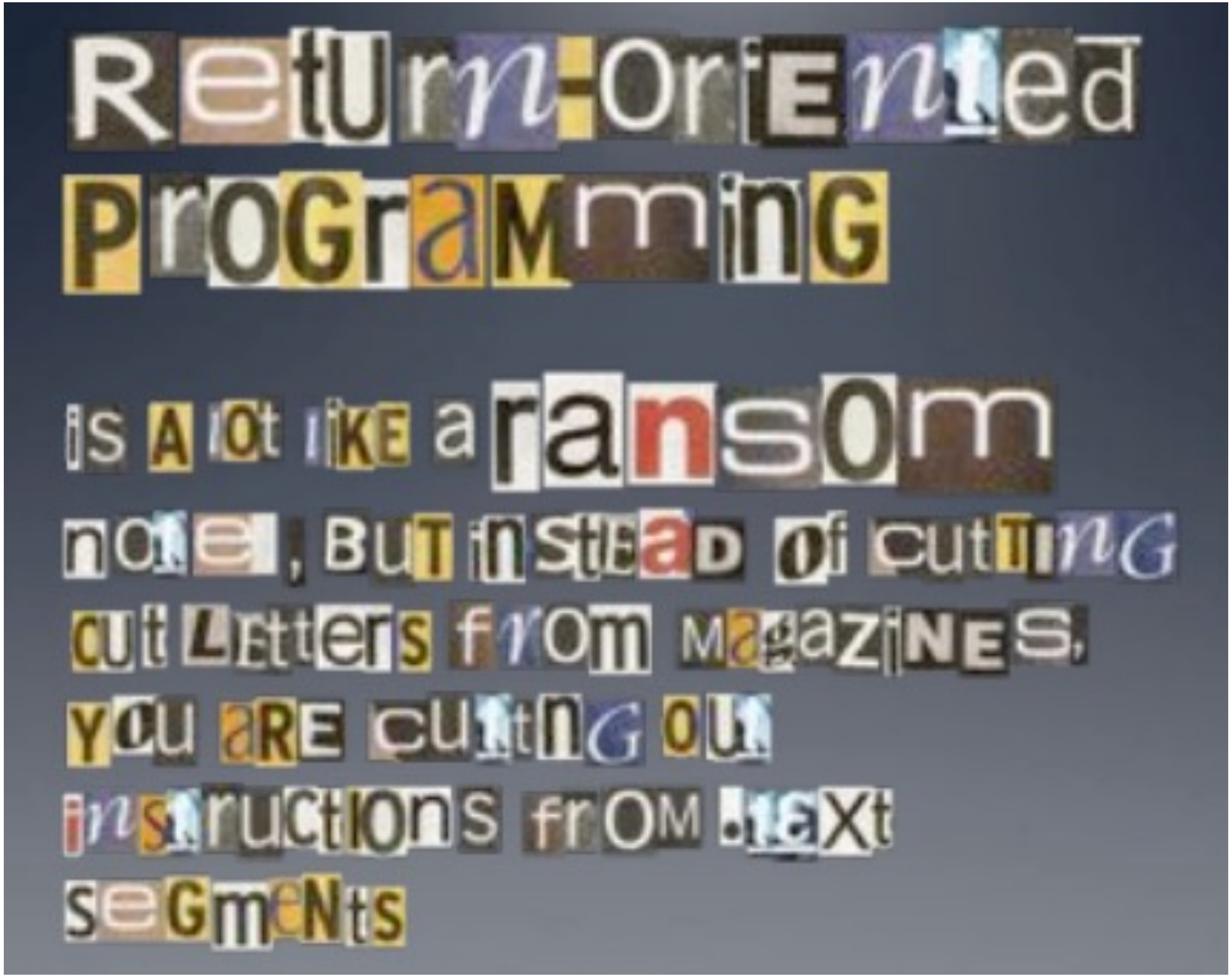
Stephen Checkoway

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 cut 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

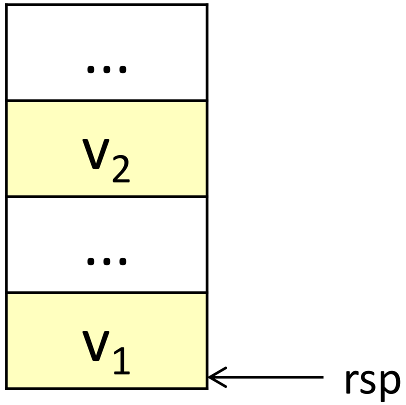
- When ROP was invented in 2007
  - Sequences of code ending in ret were the basic building blocks
  - Multiple sequences and data are assembled into reusable gadgets
- Subsequently
  - A gadget came to refer to any sequence of code ending in a ret
- In 2010
  - ROP without returns (e.g., code sequences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

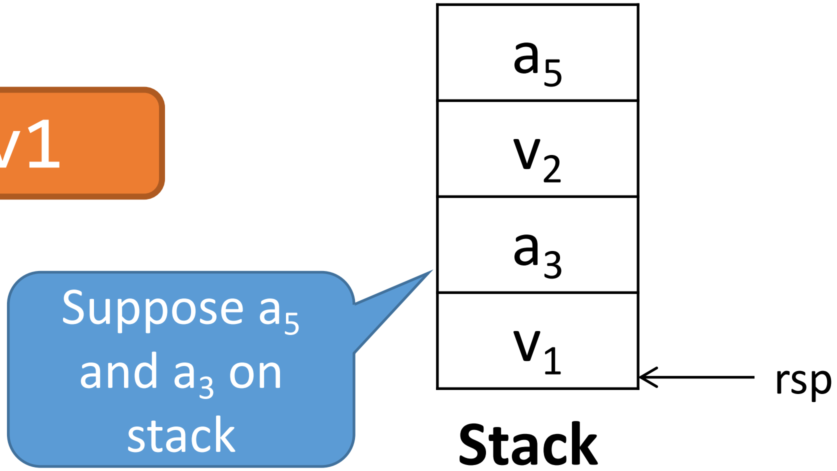
```
mov rax, [rsp]
mov rbx, [rsp+16]
mov [rbx], rax
```

Implementation 1

Constant store gadget

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

Desired Logic



Suppose a_5 and a_3 on stack

Stack

rax	v_1
rbx	
rip	a_1

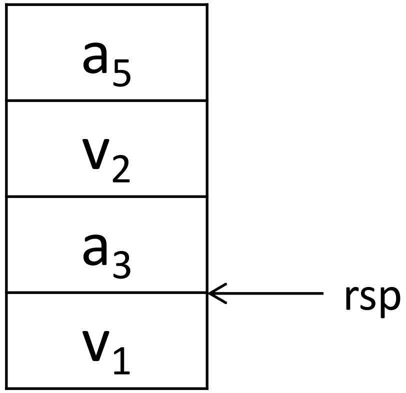
```
a1: pop rax;
a2: ret
a3: pop rbx;
a4: ret
a5: mov [rbx], rax
```

Implementation 2

Constant store gadget

Mem[v2] = v1

Desired Logic



Stack

rax	v_1
rbx	
rip	a_3

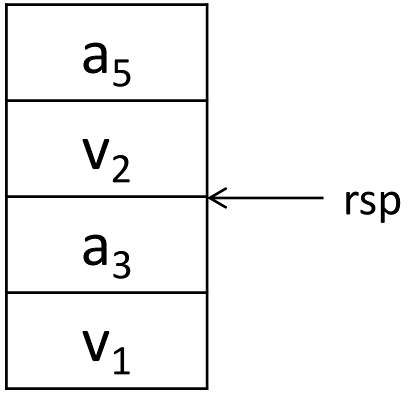
a_1: pop rax;
a_2: ret
a_3: pop rbx;
a_4: ret
a_5: mov [rbx], rax

Implementation 2

Constant store gadget

`Mem[v2] = v1`

Desired Logic



a_5

v_2

rsp

a_3

v_1

Stack

rax	v_1
rbx	v_2
rip	a_3

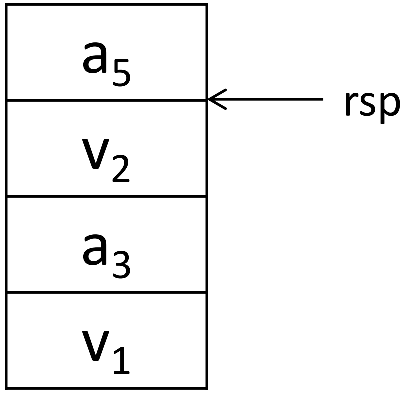
```
a1: pop rax;
a2: ret
a3: pop rbx;
a4: ret
a5: mov [rbx], rax
```

Implementation 2

Constant store gadget

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

Desired Logic



a_5

v_2

a_3

v_1

Stack

rax	v_1
rbx	v_2
rip	a_5

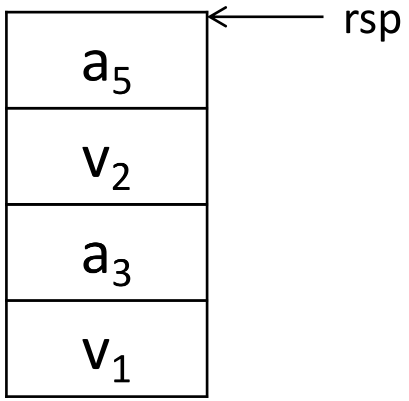
```
a₁: pop rax;
a₂: ret
a₃: pop rbx;
a₄: ret
a₅: mov [rbx], rax
```

Implementation 2

Constant store gadget

`Mem[v2] = v1`

Desired Logic



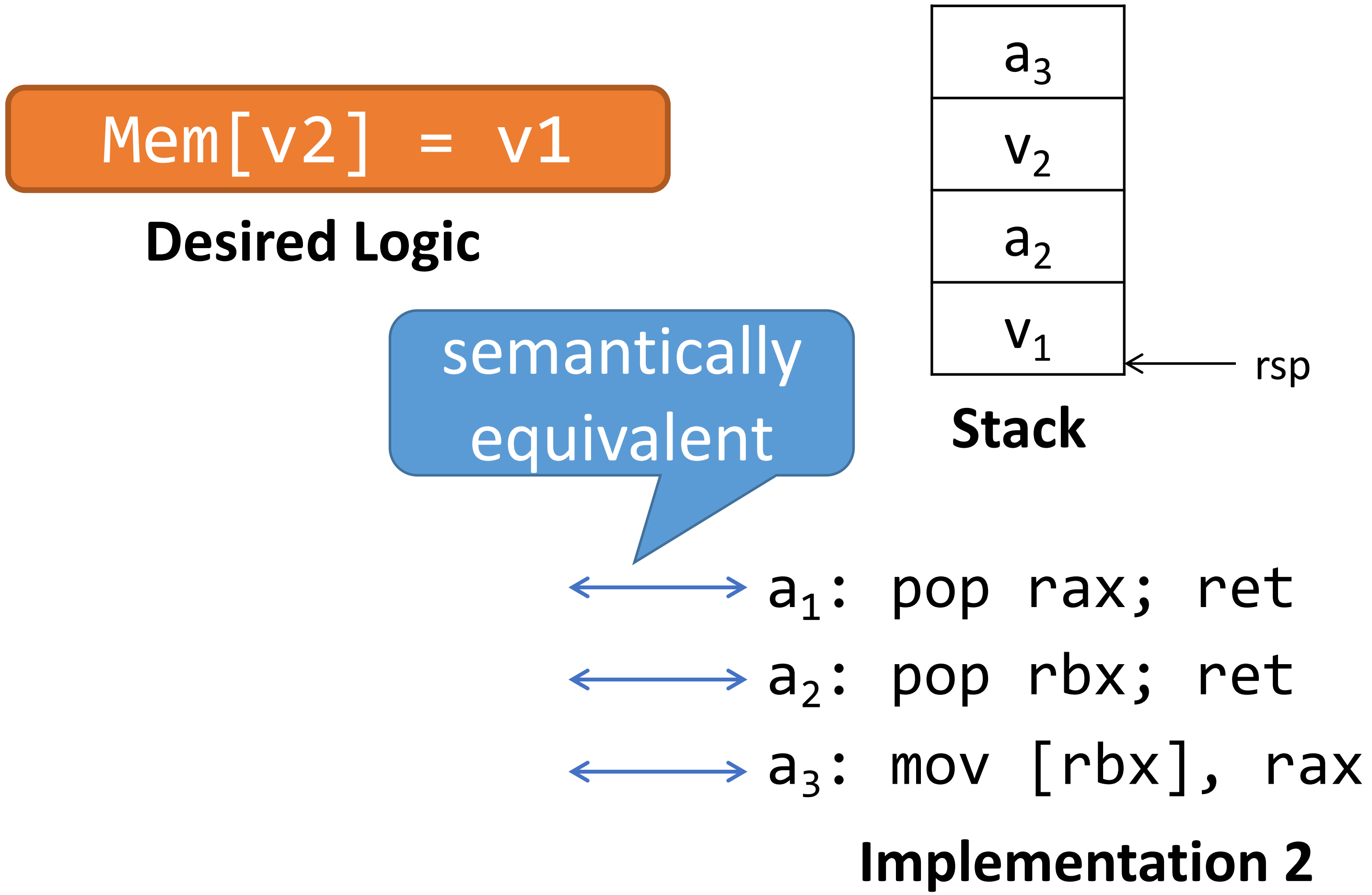
Stack

rax	v_1
rbx	v_2
rip	a_5

a_1: pop rax;
a_2: ret
a_3: pop rbx;
a_4: ret
a_5: mov [rbx], rax

Implementation 2

Equivalence



Mem[v2] = v1

Desired Logic

semantically equivalent

a_3

v_2

a_2

v_1

Stack

a_1: pop rax; ret

a_2: pop rbx; ret

a_3: mov [rbx], rax

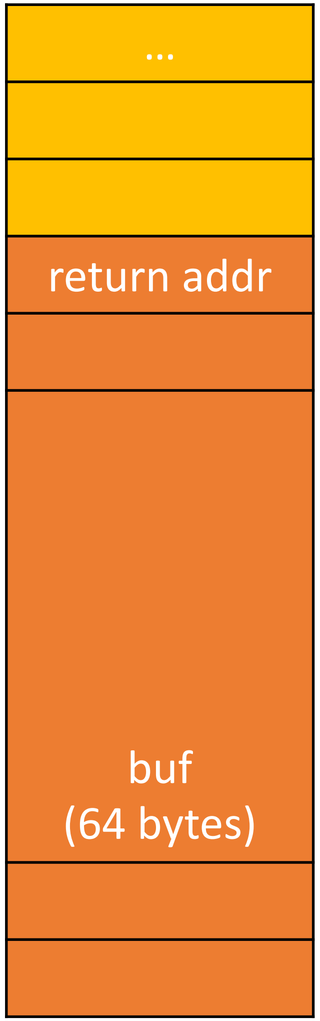
Implementation 2

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

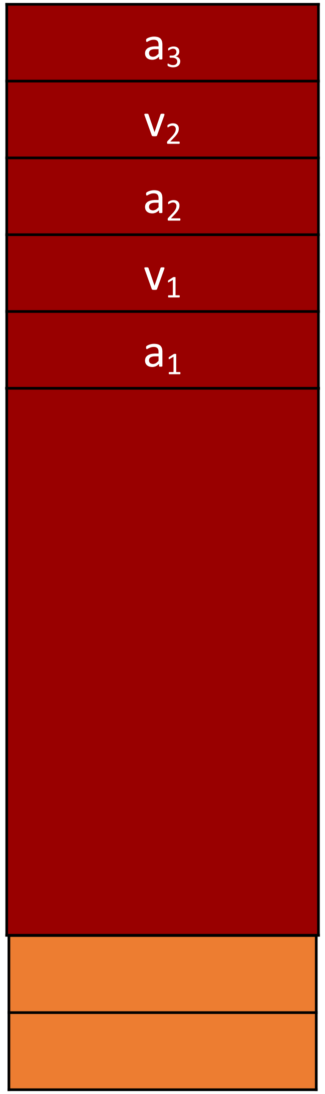
Desired *Shellcode*

a_1: pop rax; ret

a_2: pop rbx; ret

a_3: mov [rbx], rax

Desired store executed!



Arithmetic/logical operations: c = x op y

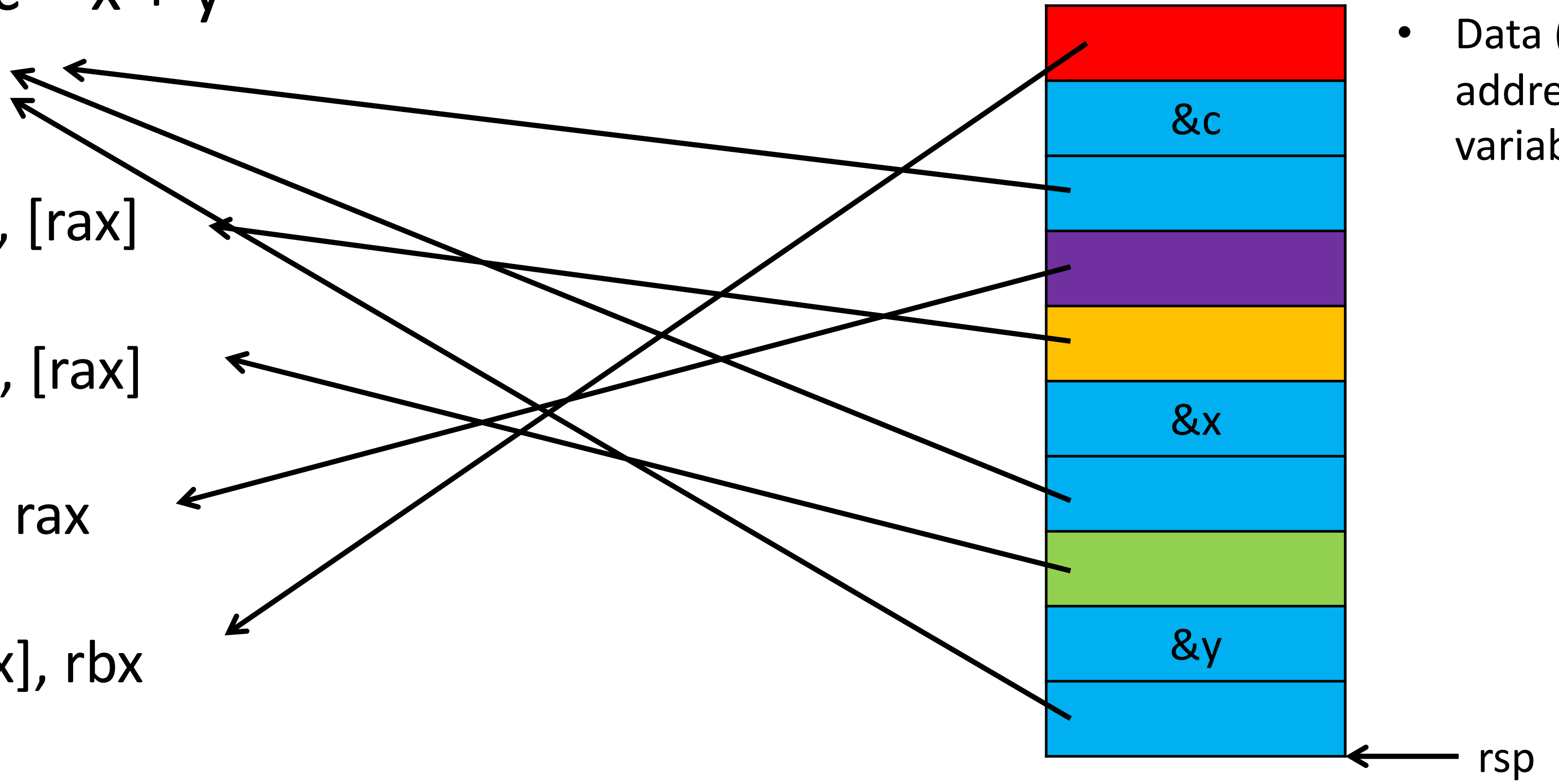
Basic strategy

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
  - pop rax
    ret
  - mov rax, [rax]
    ret
  - mov rbx, [rax]
    ret
  - add rbx, rax
    ret
  - mov [rax], rbx
    ret



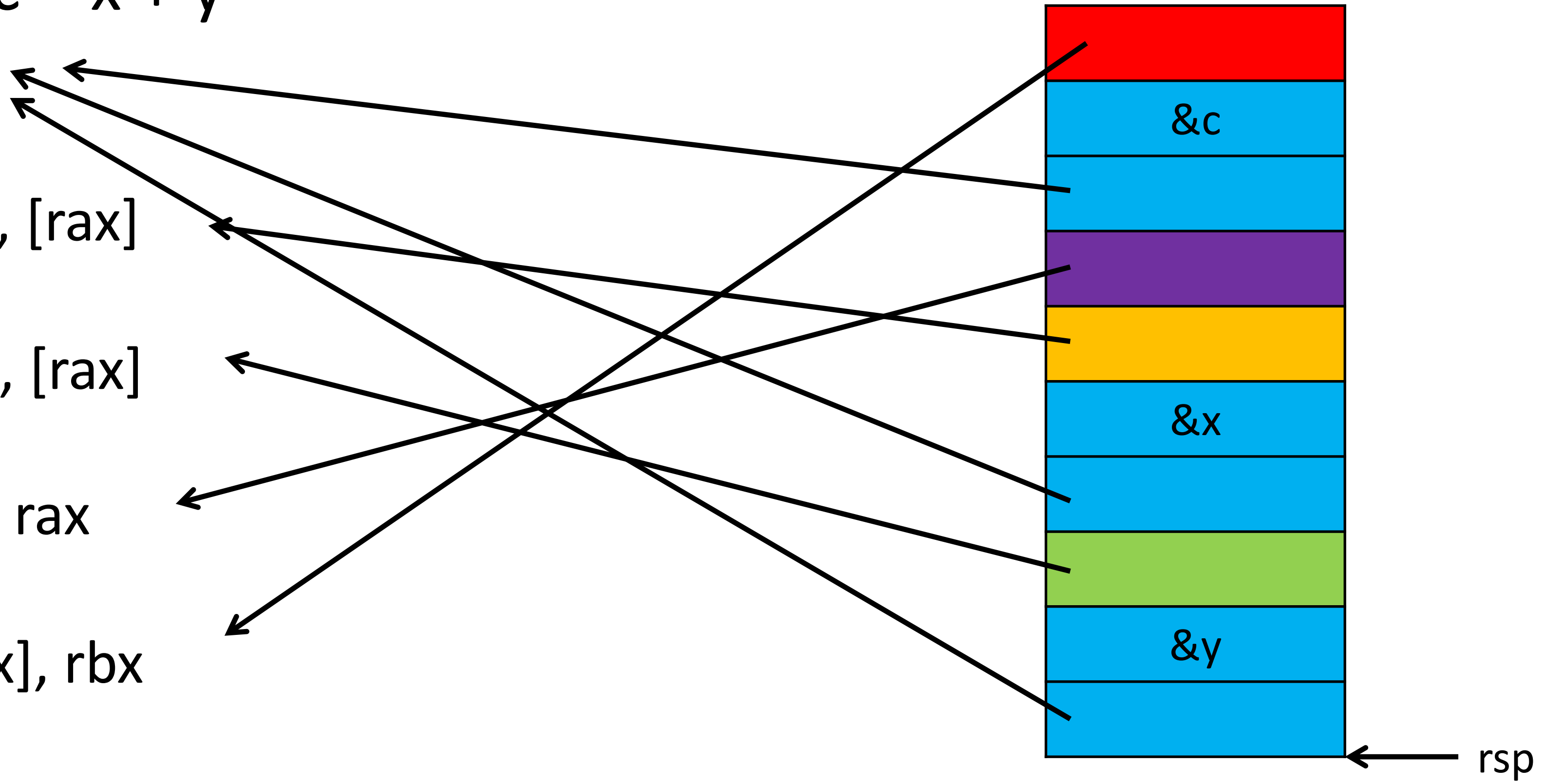
Stack contains

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

Arithmetic

Register	Value
rax	105
rbx	3852

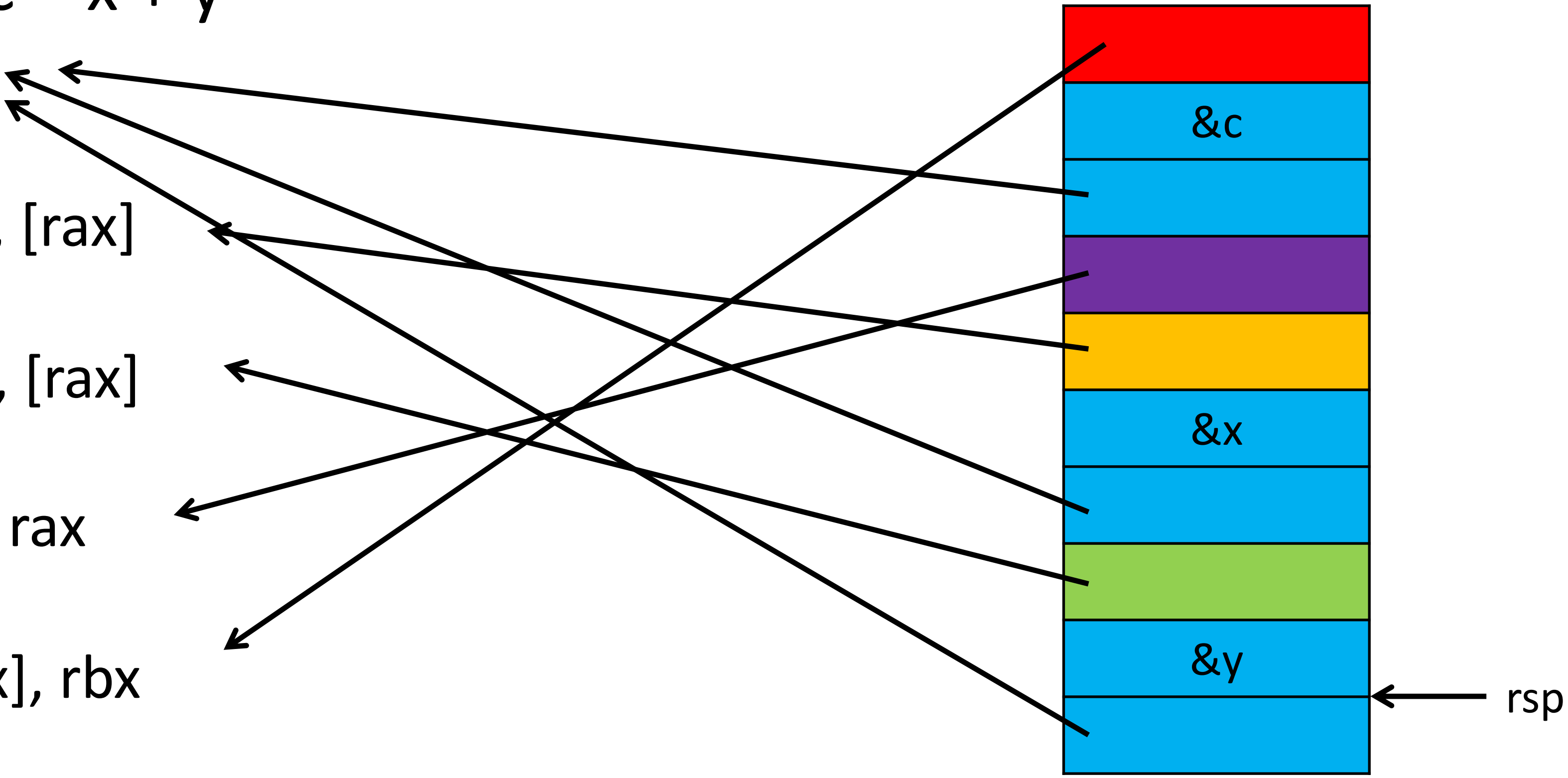
- Addition: c = x + y
  - pop rax
    ret
  - mov rax, [rax]
    ret
  - mov rbx, [rax]
    ret
  - add rbx, rax
    ret
  - mov [rax], rbx
    ret



Arithmetic

Register	Value
rax	105
rbx	3852

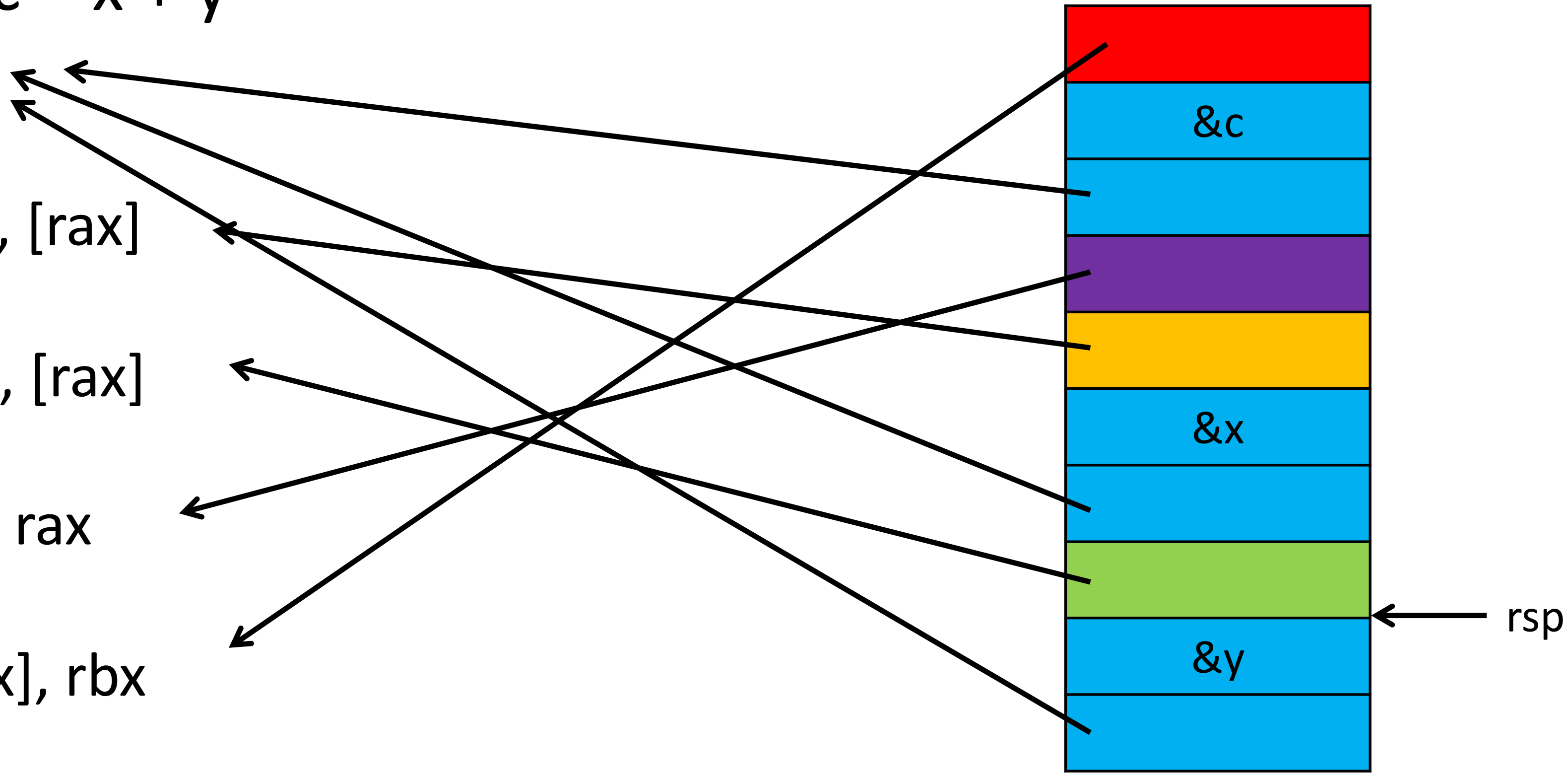
- Addition: c = x + y
  - pop rax
    ret
  - mov rax, [rax]
    ret
  - mov rbx, [rax]
    ret
  - add rbx, rax
    ret
  - mov [rax], rbx
    ret



Arithmetic

Register	Value
rax	&y
rbx	3852

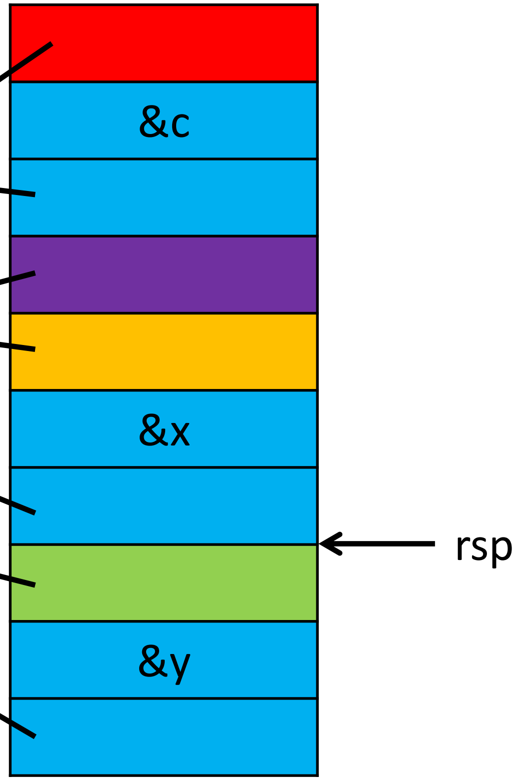
- Addition: c = x + y
  - pop rax
    ret
  - mov rax, [rax]
    ret
  - mov rbx, [rax]
    ret
  - add rbx, rax
    ret
  - mov [rax], rbx
    ret



Arithmetic

Register	Value
rax	&y
rbx	3852

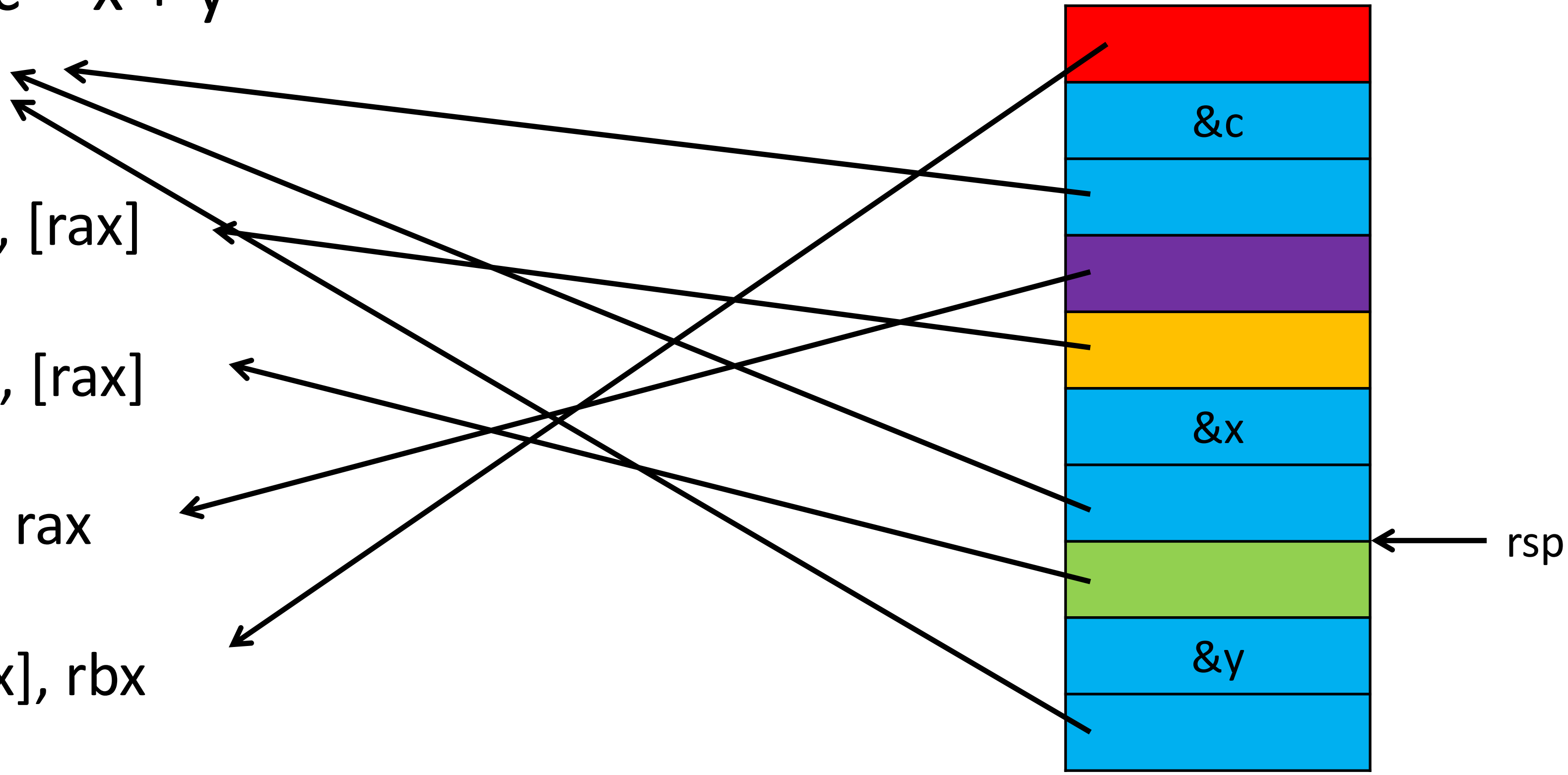
- Addition: c = x + y
  - pop rax
    ret
  - mov rax, [rax]
    ret
  - mov rbx, [rax]
    ret
  - add rbx, rax
    ret
  - mov [rax], rbx
    ret



Arithmetic

Register	Value
rax	&y
rbx	y

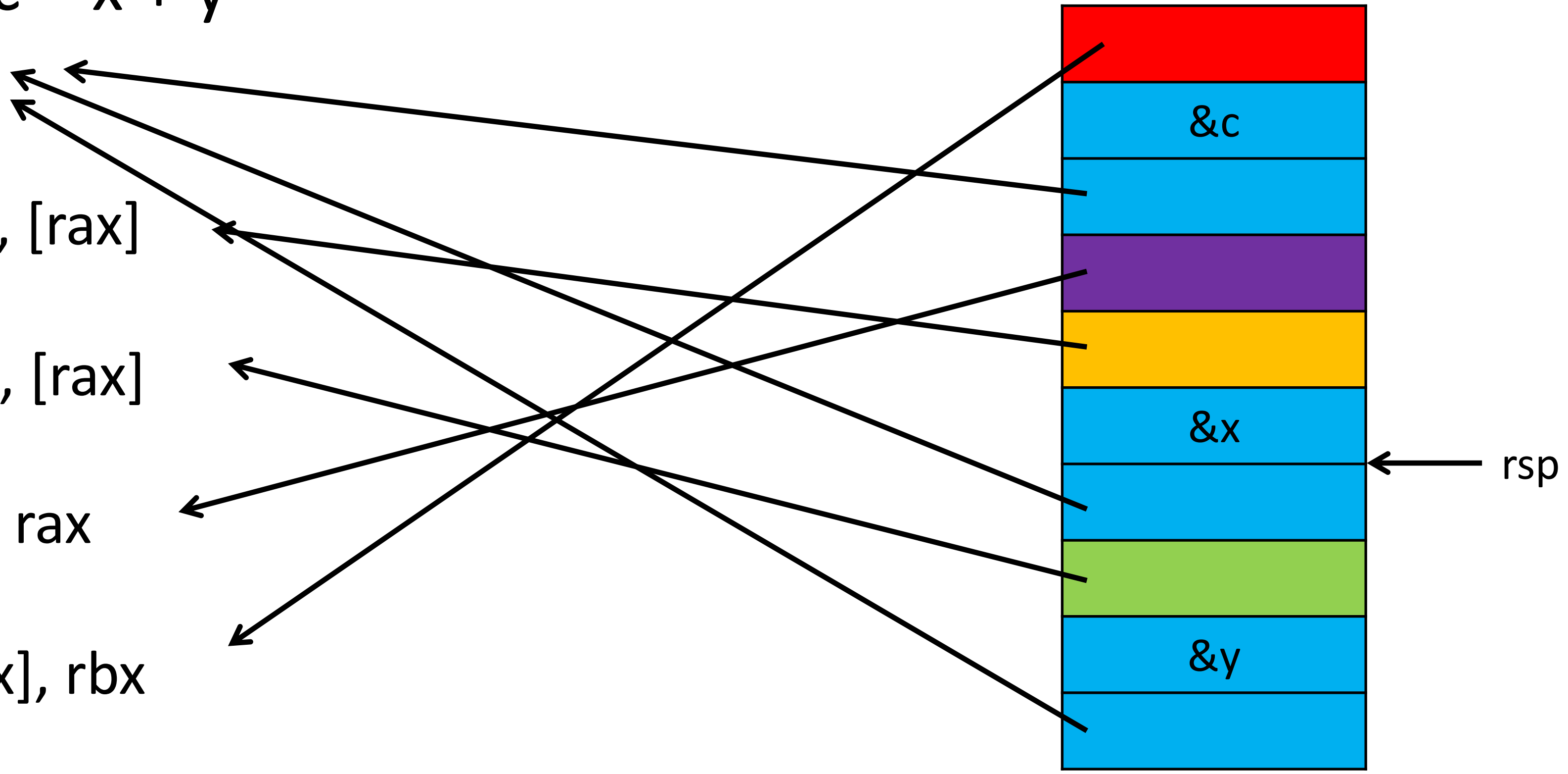
- Addition: c = x + y
  - pop rax
    ret
  - mov rax, [rax]
    ret
  - mov rbx, [rax]
    ret
  - add rbx, rax
    ret
  - mov [rax], rbx
    ret



Arithmetic

Register	Value
rax	&y
rbx	y

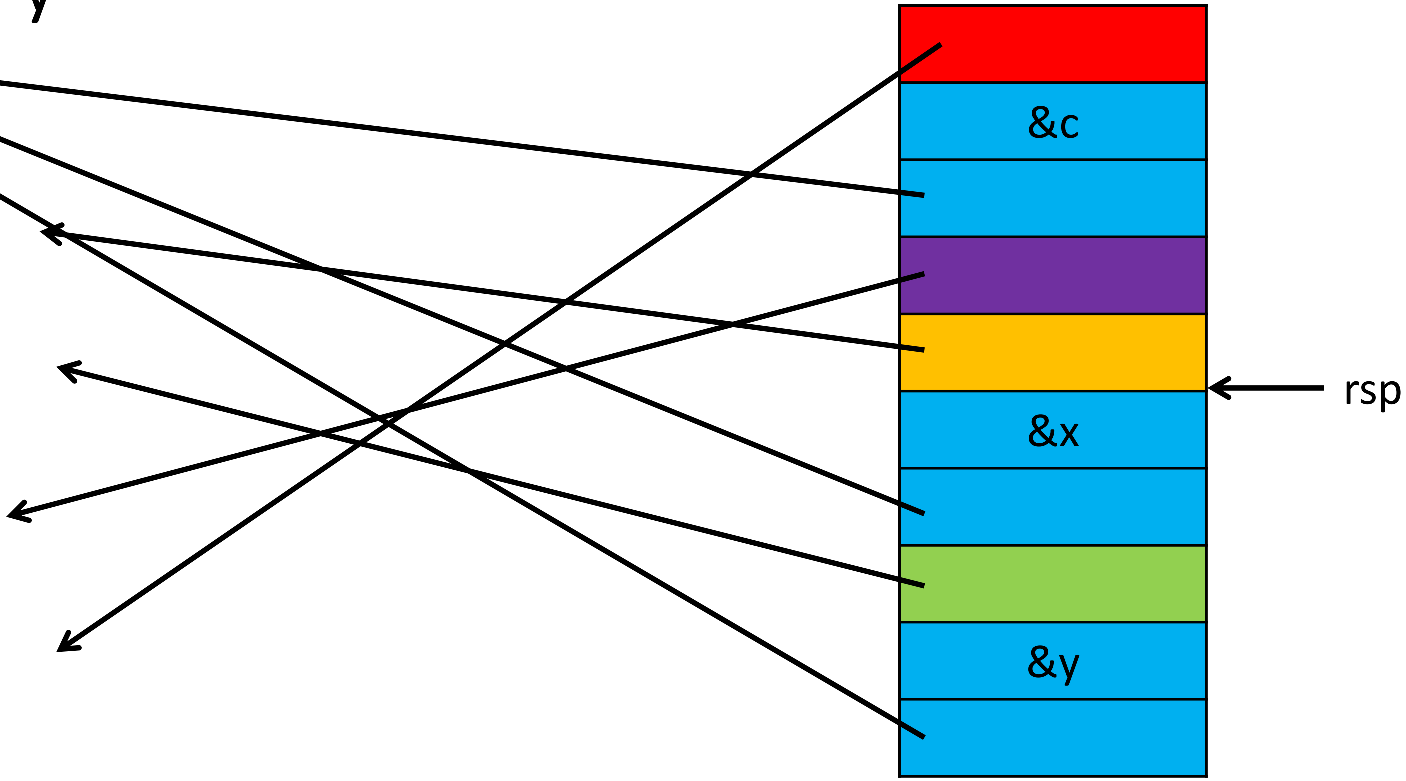
- Addition: c = x + y
  - pop rax
    ret
  - mov rax, [rax]
    ret
  - mov rbx, [rax]
    ret
  - add rbx, rax
    ret
  - mov [rax], rbx
    ret



Arithmetic

Register	Value
rax	&x
rbx	y

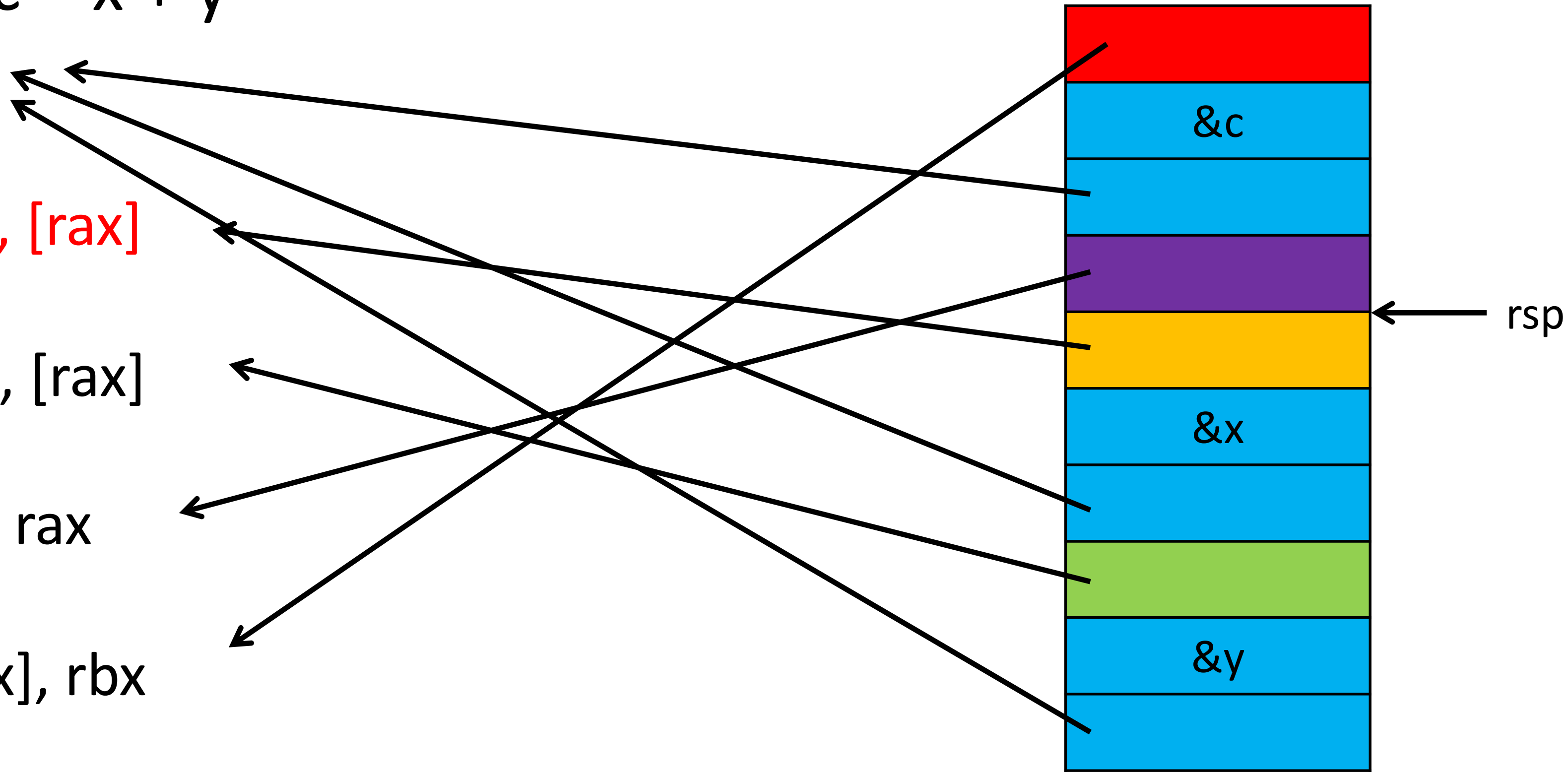
- Addition: c = x + y
  - pop rax
    ret
  - mov rax, [rax]
    ret
  - mov rbx, [rax]
    ret
  - add rbx, rax
    ret
  - mov [rax], rbx
    ret



Arithmetic

Register	Value
rax	&x
rbx	y

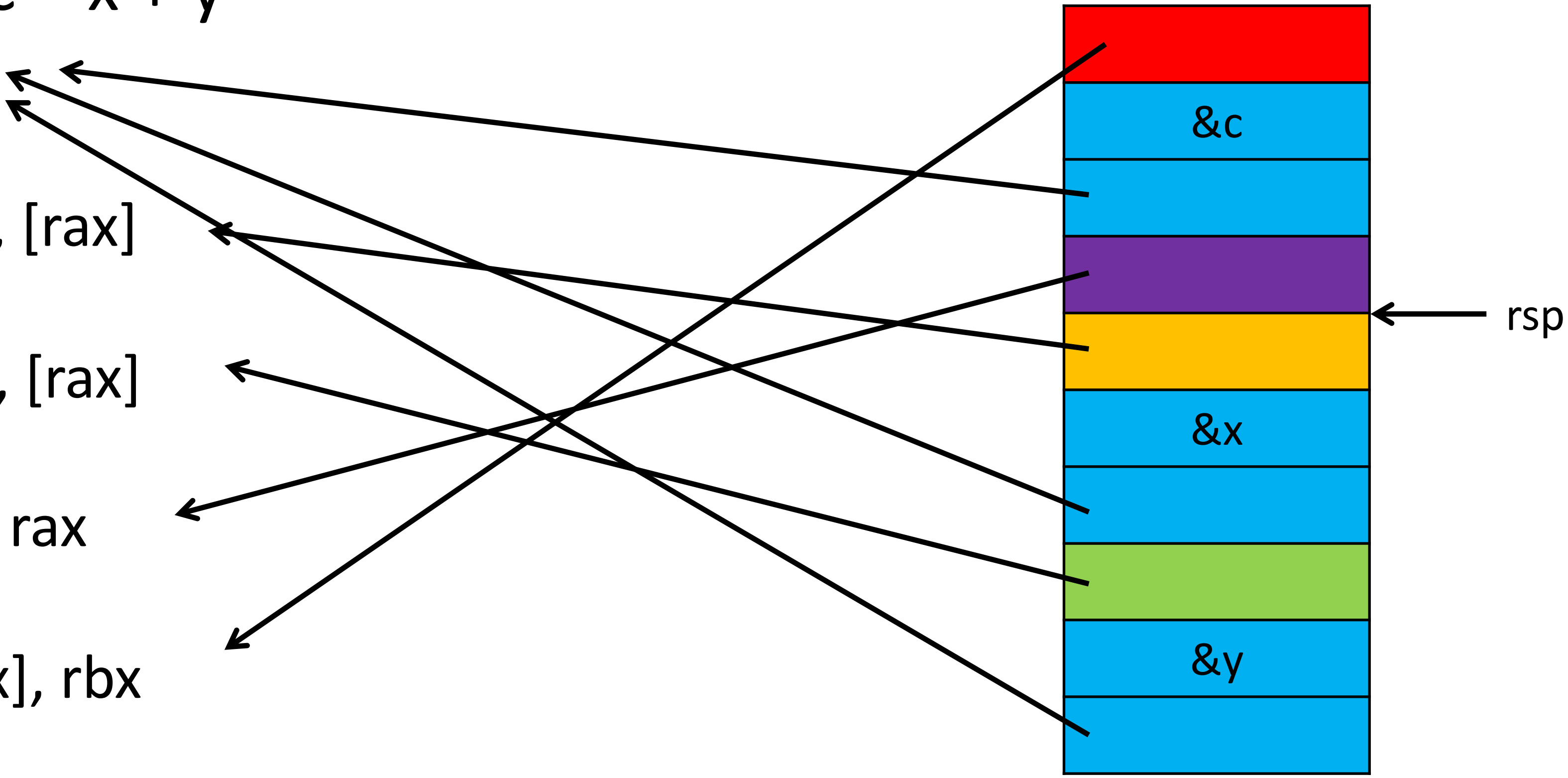
- Addition: c = x + y
  - pop rax
    ret
  - mov rax, [rax]
    ret
  - mov rbx, [rax]
    ret
  - add rbx, rax
    ret
  - mov [rax], rbx
    ret



Arithmetic

Register	Value
rax	x
rbx	y

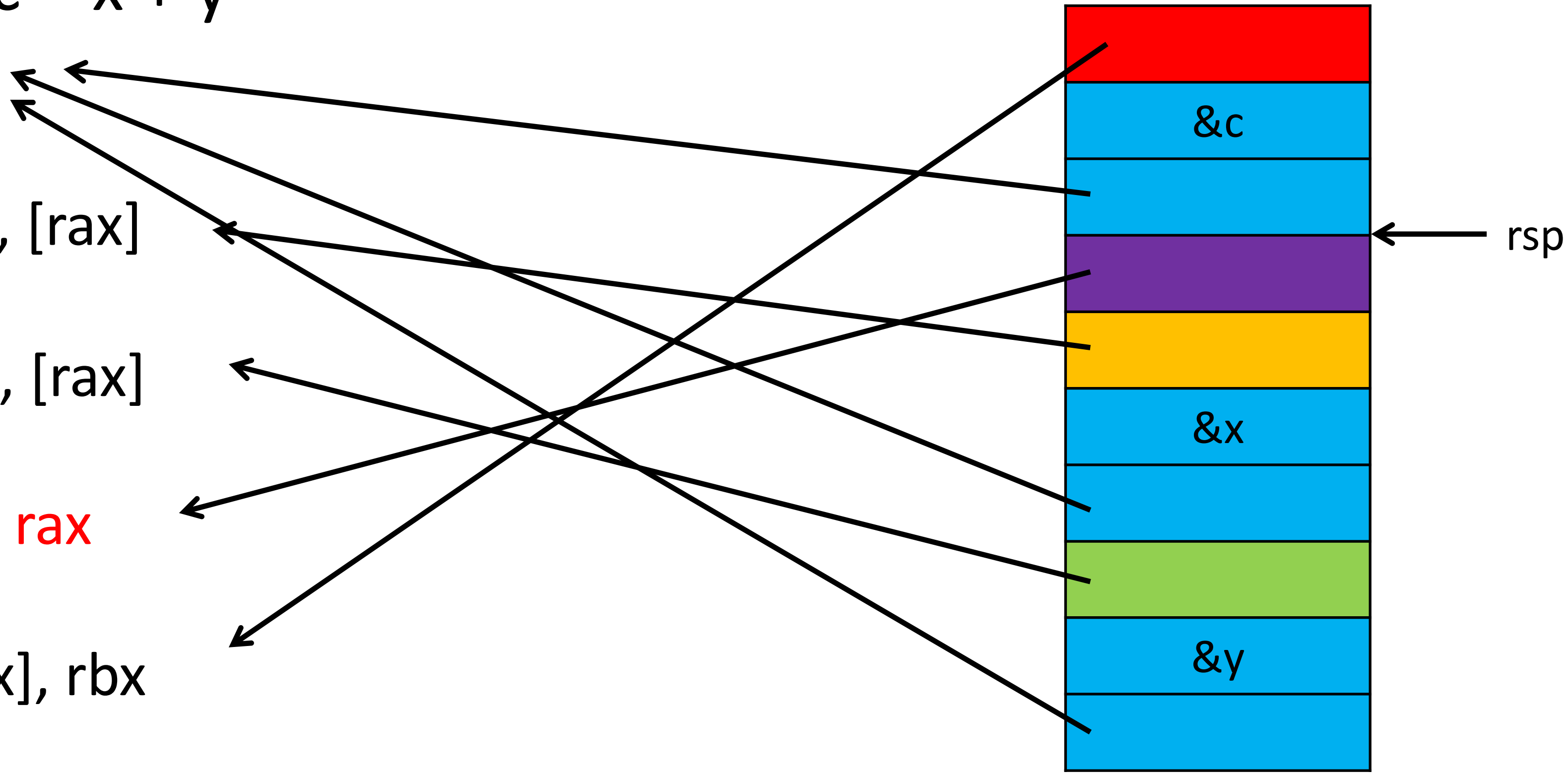
- Addition: c = x + y
  - pop rax
    ret
  - mov rax, [rax]
    ret
  - mov rbx, [rax]
    ret
  - add rbx, rax
    ret
  - mov [rax], rbx
    ret



Arithmetic

Register	Value
rax	x
rbx	y

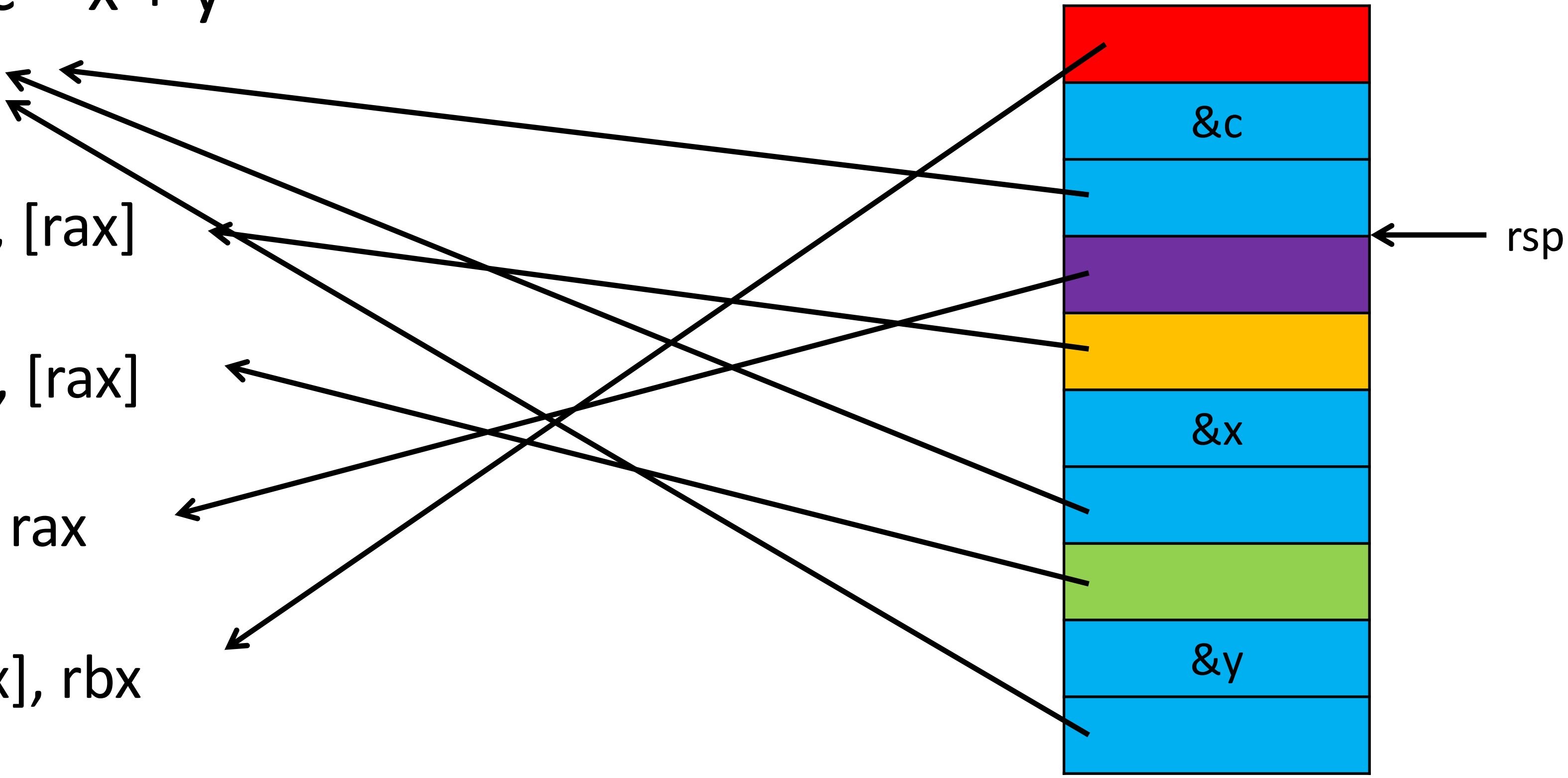
- Addition: c = x + y
  - pop rax
    ret
  - mov rax, [rax]
    ret
  - mov rbx, [rax]
    ret
  - add rbx, rax
    ret
  - mov [rax], rbx
    ret



Arithmetic

Register	Value
rax	x
rbx	y + x

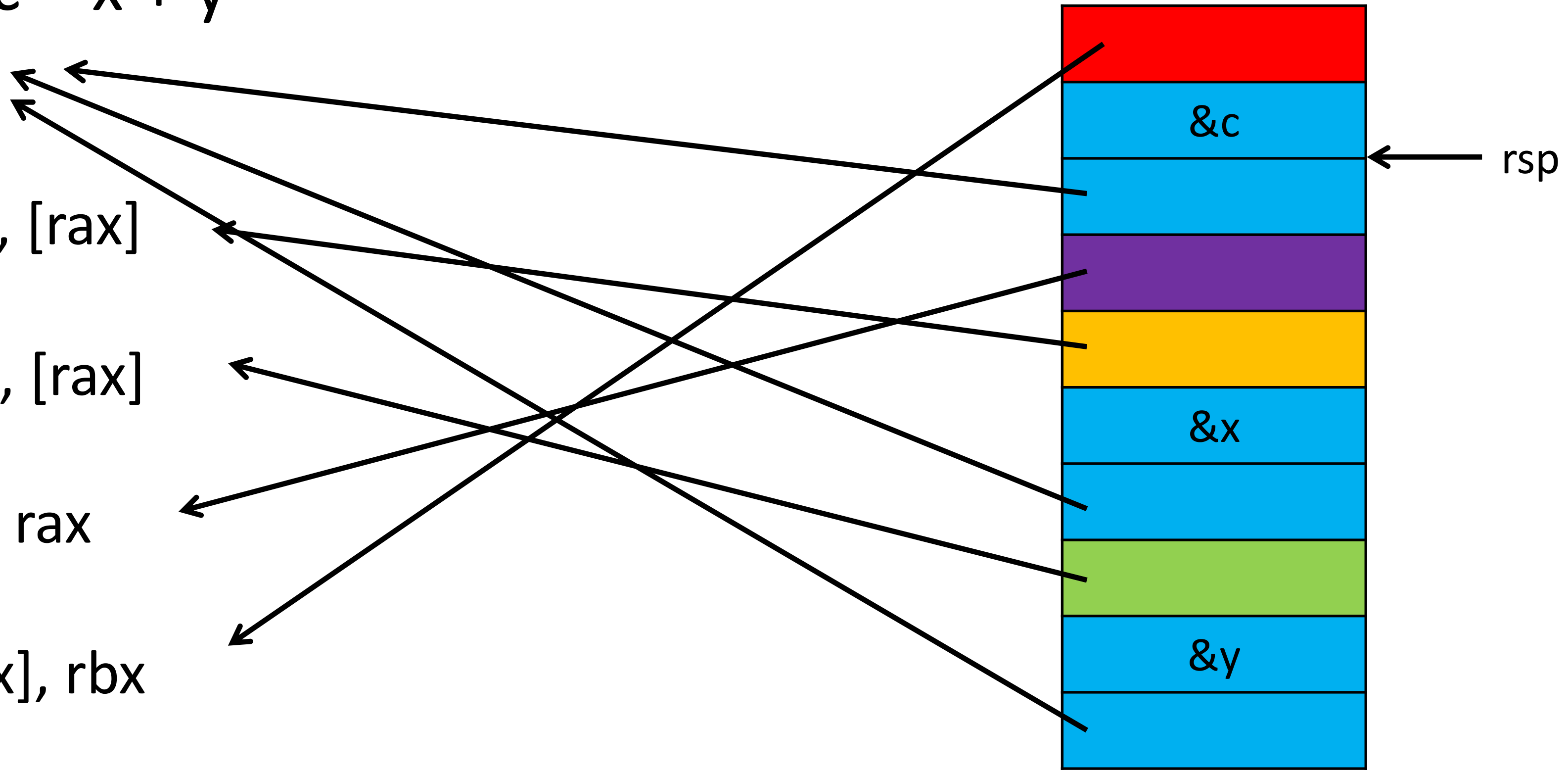
- Addition: c = x + y
  - pop rax
    ret
  - mov rax, [rax]
    ret
  - mov rbx, [rax]
    ret
  - add rbx, rax
    ret
  - mov [rax], rbx
    ret



Arithmetic

Register	Value
rax	x
rbx	y + x

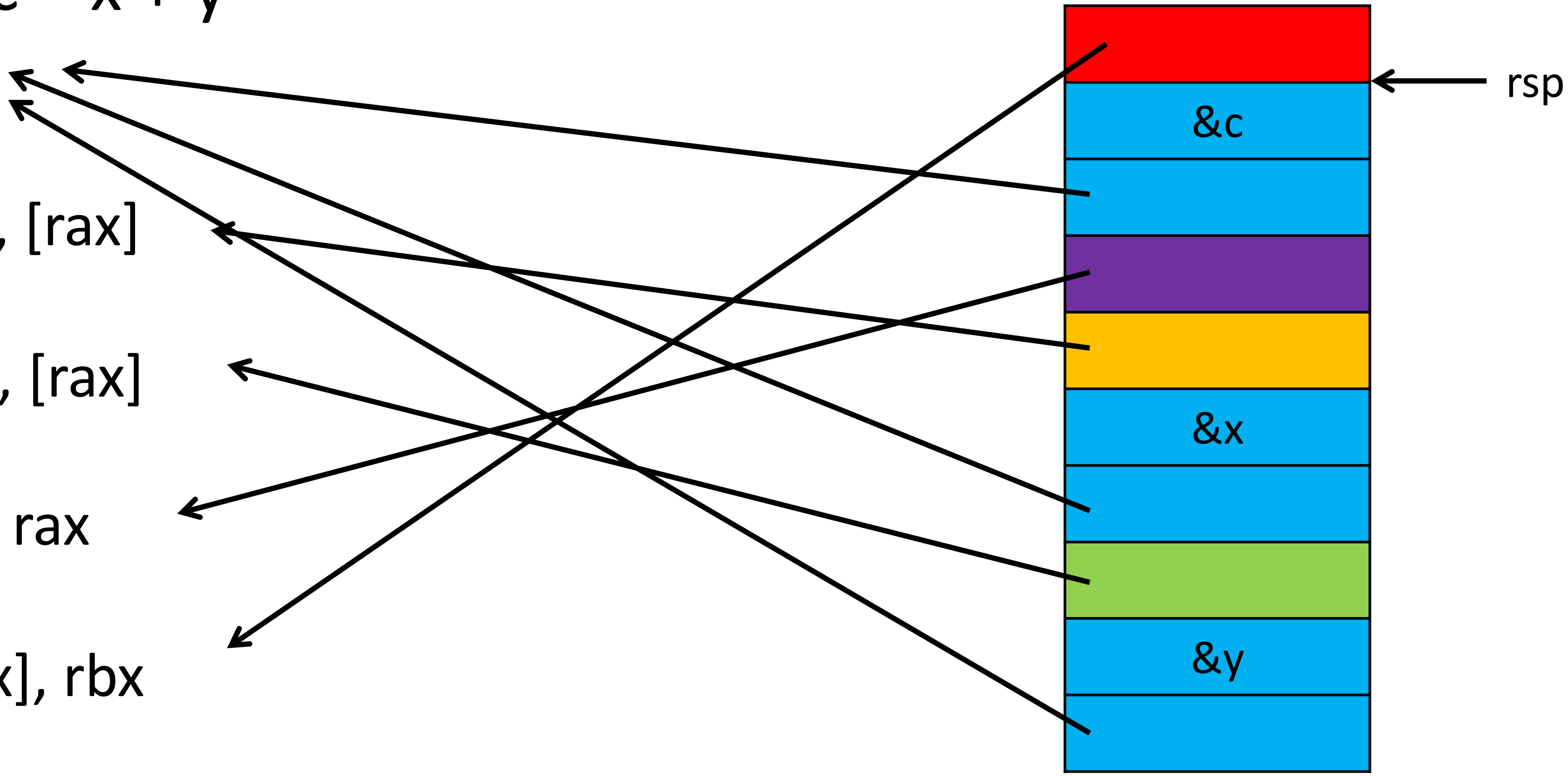
- Addition: c = x + y
  - pop rax
    ret
  - mov rax, [rax]
    ret
  - mov rbx, [rax]
    ret
  - add rbx, rax
    ret
  - mov [rax], rbx
    ret



Arithmetic

Register	Value
rax	&c
rbx	y + x

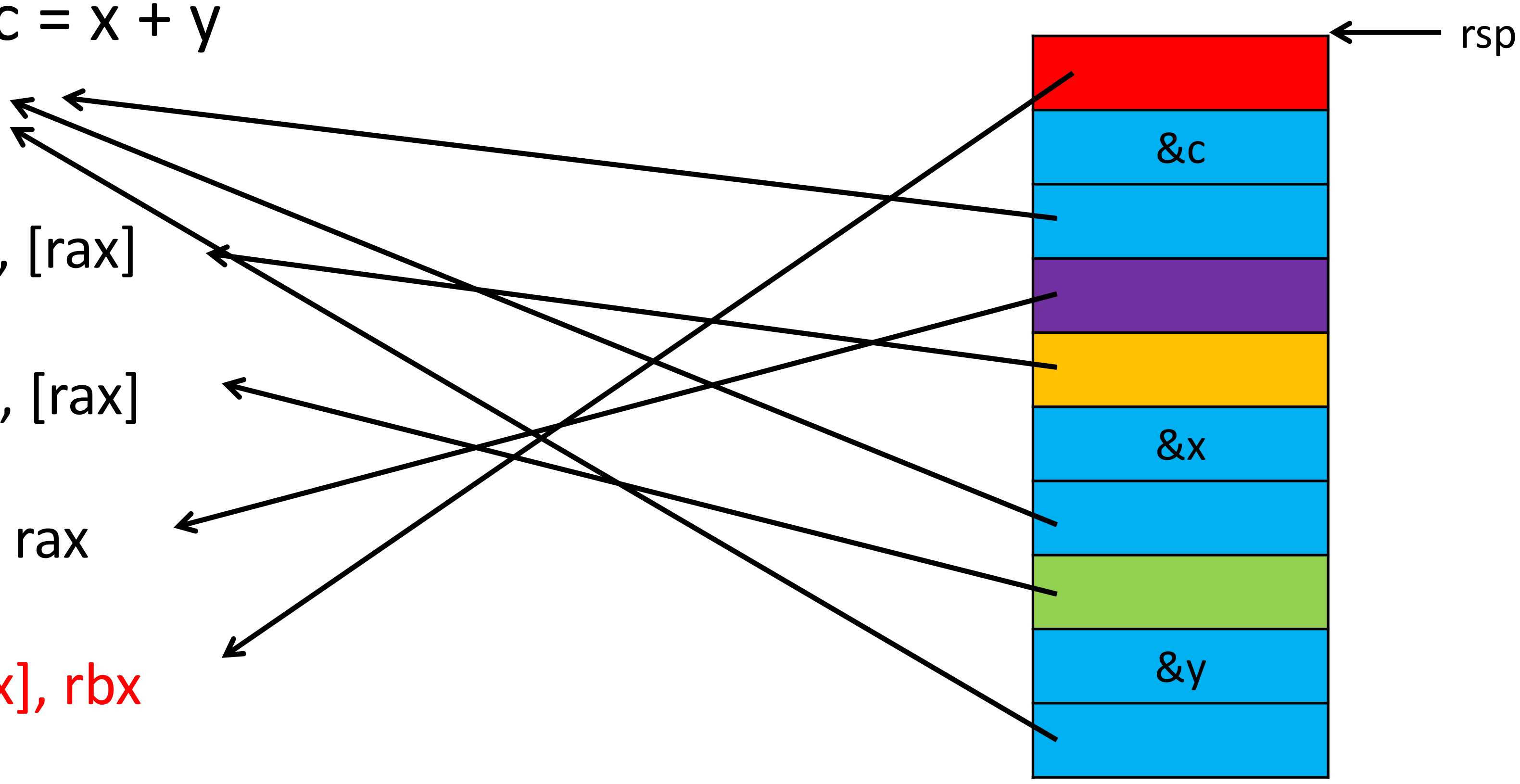
- Addition: c = x + y
  - pop rax
    ret
  - mov rax, [rax]
    ret
  - mov rbx, [rax]
    ret
  - add rbx, rax
    ret
  - mov [rax], rbx
    ret



Arithmetic

Register	Value
rax	&c
rbx	y + x

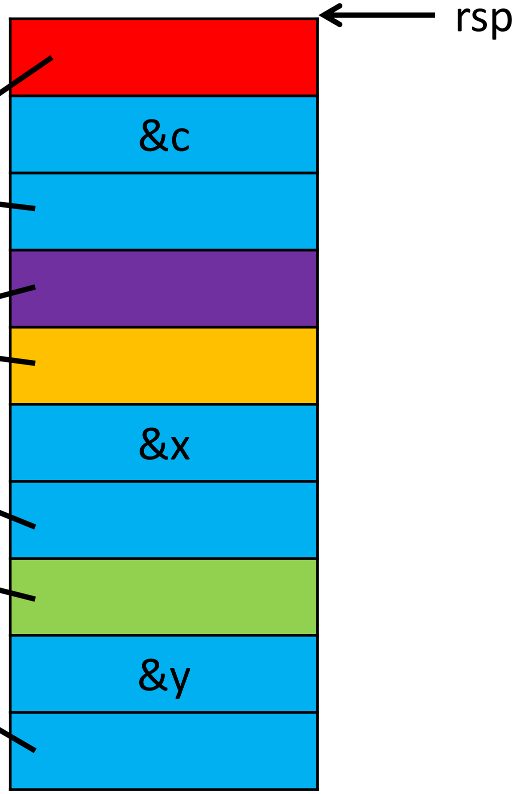
- Addition: c = x + y
  - pop rax
    ret
  - mov rax, [rax]
    ret
  - mov rbx, [rax]
    ret
  - add rbx, rax
    ret
  - mov [rax], rbx
    ret



Arithmetic

Register	Value
rax	&c
rbx	y + x

- Addition: c = x + y
  - pop rax
    ret
  - mov rax, [rax]
    ret
  - mov rbx, [rax]
    ret
  - add rbx, rax
    ret
  - mov [rax], rbx
    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	rip	rsp
No-op	nop	ret
Unconditional jump	jmp address	set rsp to address of gadget
Conditional jump	jnz address	set rsp 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

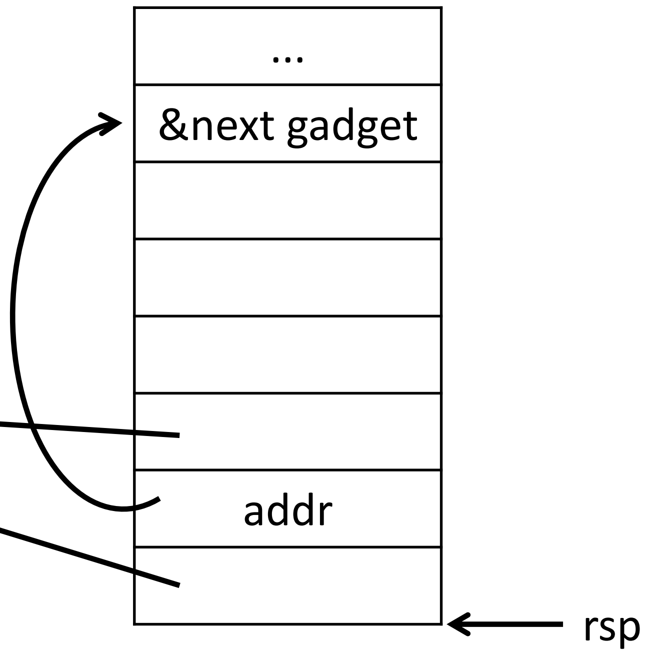
- Processors support instructions that conditionally change the PC
  - On x86
    - Jcc family: jz, jnz, jl, jle, etc. 33 in total
    - loop, loope, loopne
    - Based on condition codes mostly; and on rcx for some
  - On MIPS
    - beq and bne
    - Based on comparison of registers
- Processors generally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

We want conditional jumps

- Unconditional jump addr
  - pop rax
    ret
  - mov rsp, rax
    ret

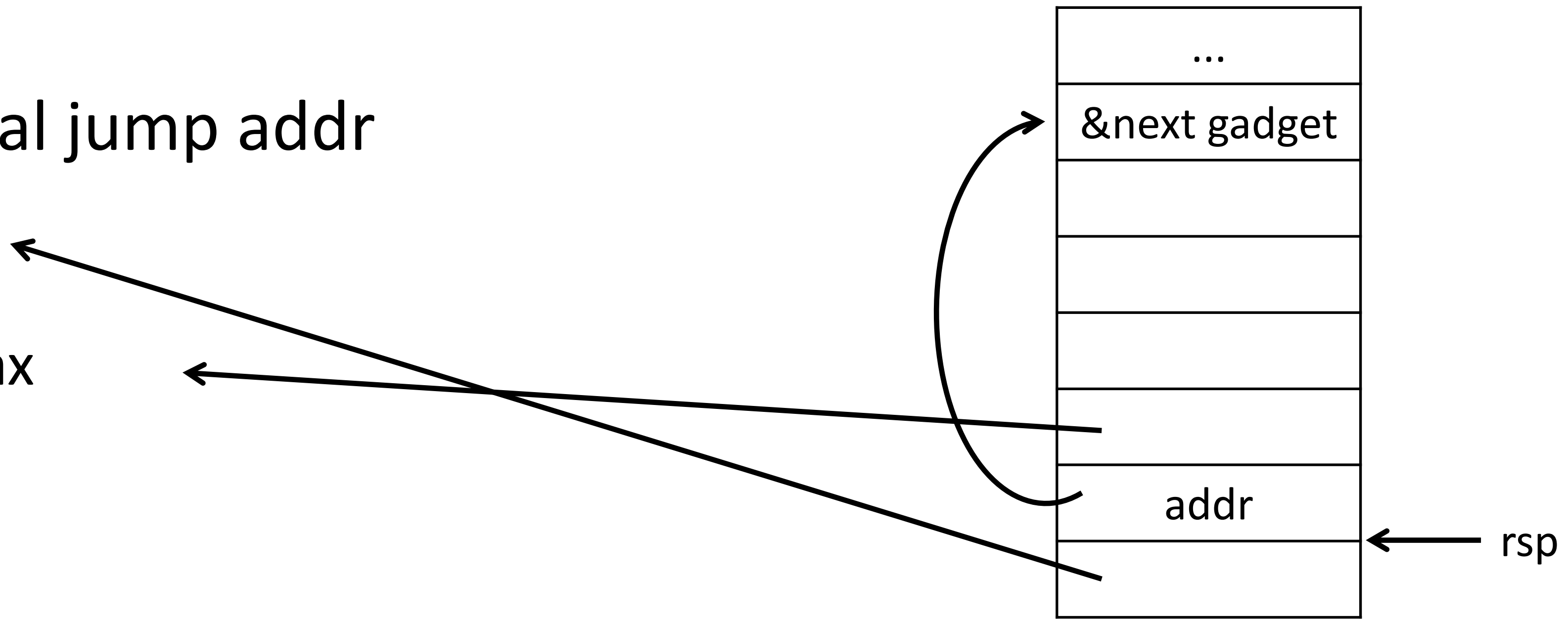
We want conditional jumps

- Unconditional jump addr
    - pop rax
      ret
    - mov rsp, rax
      ret



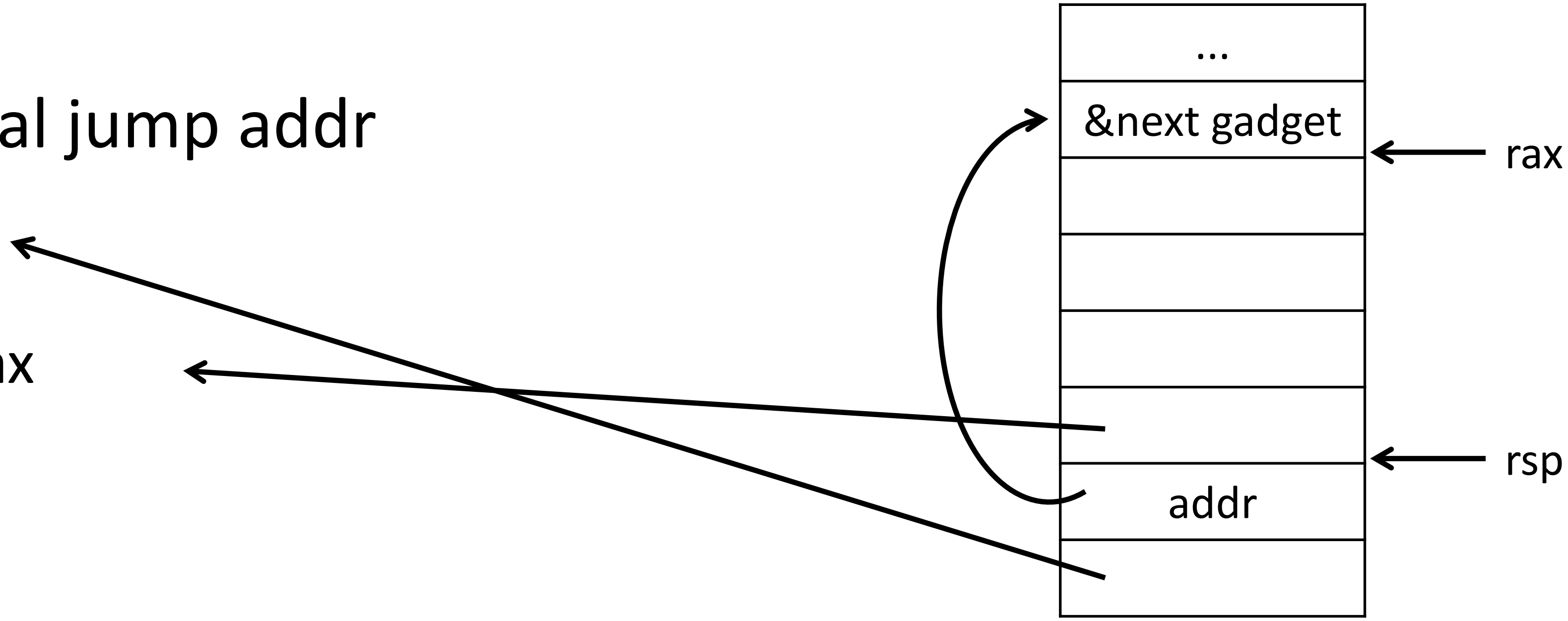
We want conditional jump

- Unconditional jump addr
  - pop rax
    ret
  - mov rsp, rax
    ret



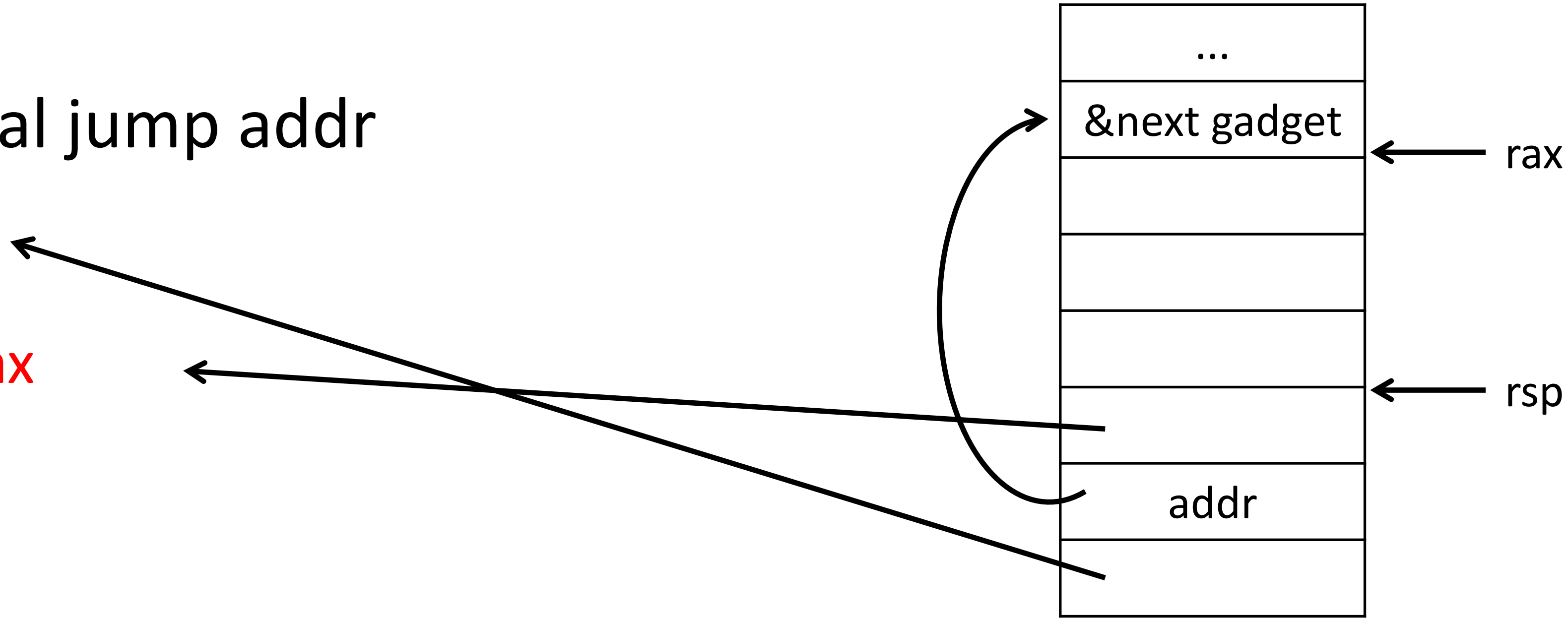
We want conditional jump

- Unconditional jump addr
  - pop rax
    ret
  - mov rsp, rax
    ret



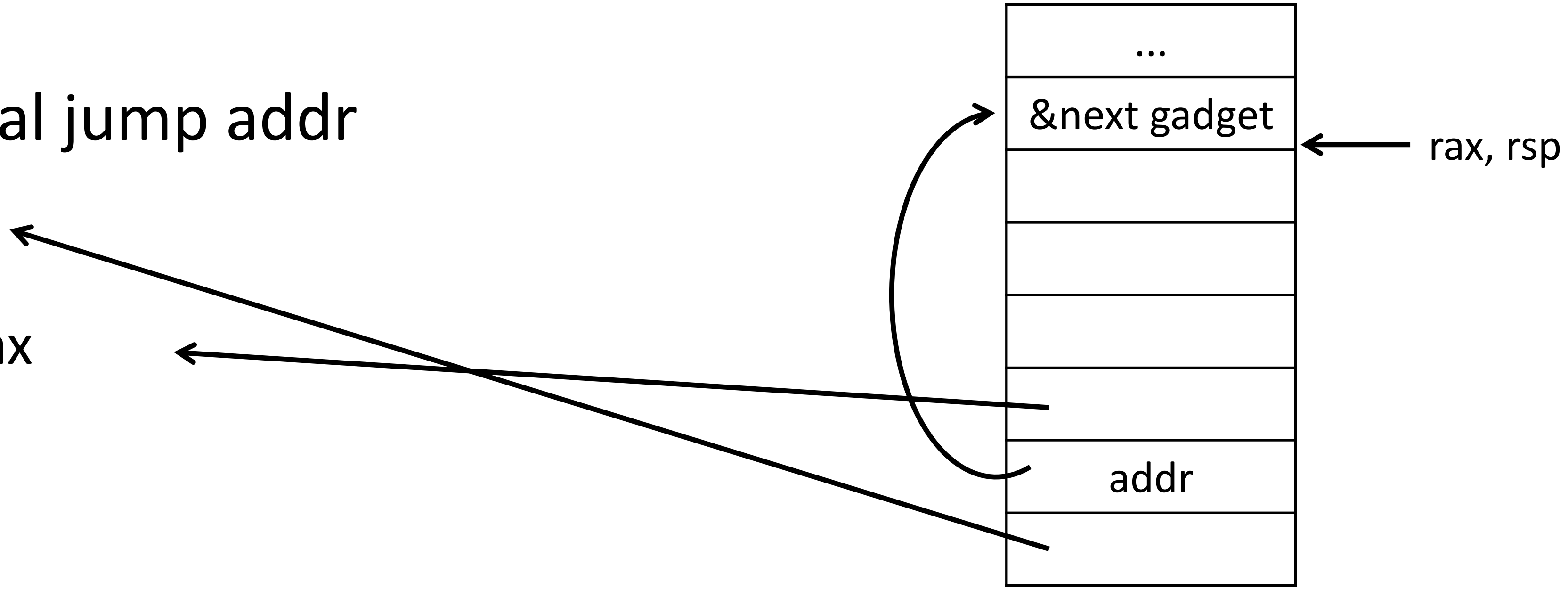
We want conditional jumps

- Unconditional jump addr
  - pop rax
    ret
  - mov rsp, rax
    ret



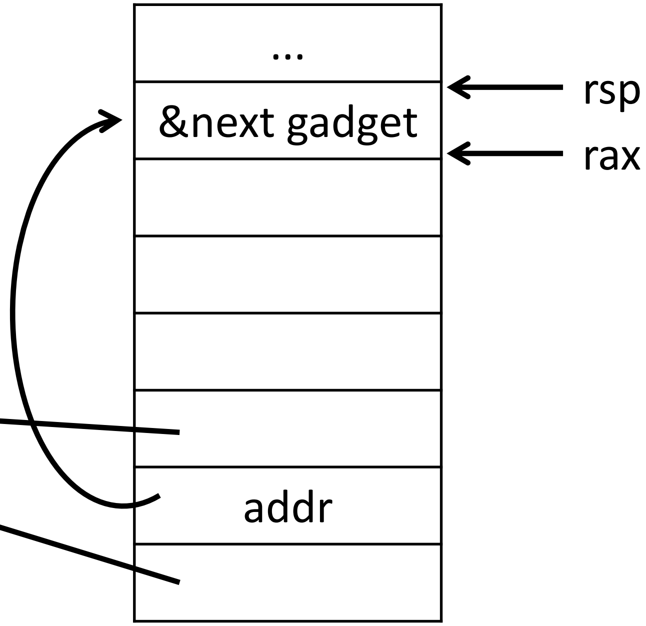
We want conditional jumps

- Unconditional jump addr
  - pop rax
    ret
  - mov rsp, rax
    ret



We want conditional jumps

- Unconditional jump addr
  - pop rax
    ret
  - mov rsp, rax
    ret



We want conditional jumps

- Unconditional jump addr
  - pop rax
    ret
  - mov rsp, rax
    ret
- Conditional jump addr, one way
  - Conditionally set a register to 0 or 0xffffffffffffffff = -1
  - Perform a logical AND with the register and an offset
  - Add the result to rsp
- Another approach: use conditional move instructions cmovz, cmovnz after setting the zero flag via cmp or test

Conditionally set a register to 0 or -1

- Compare registers rax and rbx and set rcx to
  - -1 if rax < rbx
  - 0 if rax >= rbx
- Ideally we would find a sequence like

```
cmp rax, rbx        set carry flag cf according to rax - rbx
sbb rcx, rcx        rcx ← rcx - rcx - cf; or rcx ← -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 rsp, reg; ret sequence

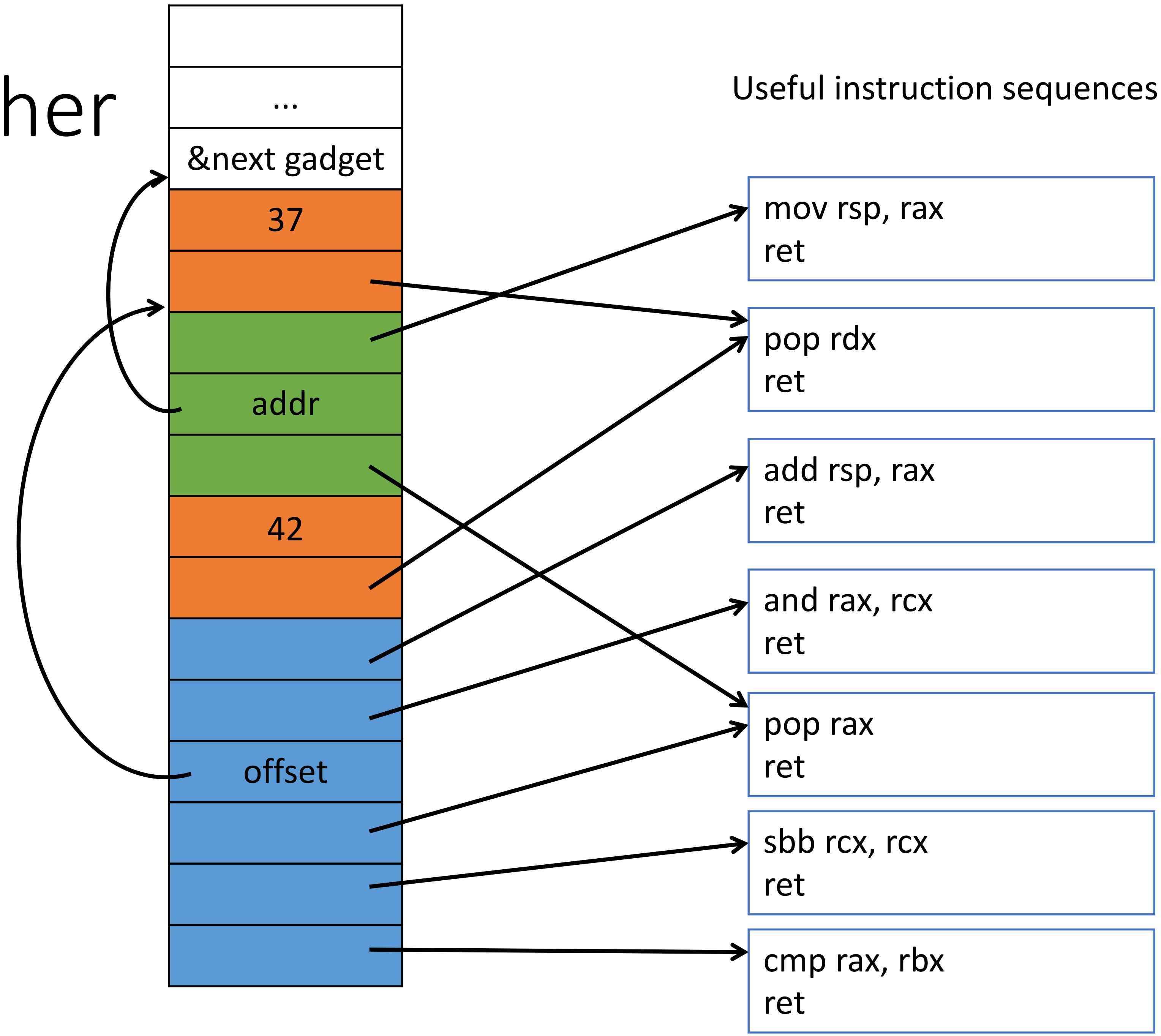
- Other options available, e.g., if rsp + offset is in rax
    - push rax ; pop rsp ; ret (has the downside that it modified the stack which is an issue for loops)
    - xchg rax, rsp

Putting it together

Conditional jump

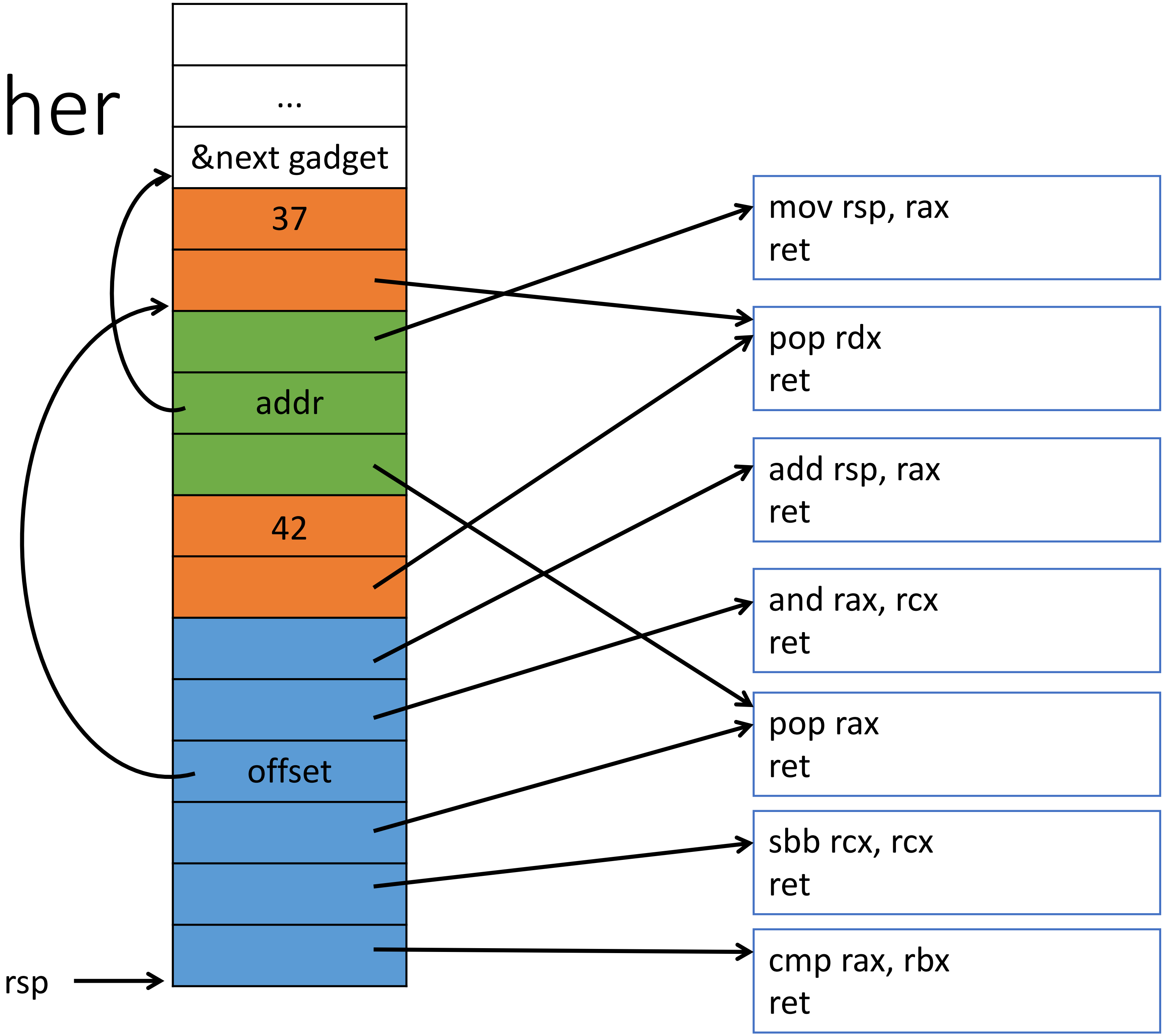
Load constant in rdx

Unconditional jump



Putting it together

Register	Value
rax	10
rbx	20
rcx	108
rdx	17



…

&next gadget

37

addr

42

offset

rsp

```
mov rsp, rax
ret
```

```
pop rdx
ret
```

```
add rsp, rax
ret
```

```
and rax, rcx
ret
```

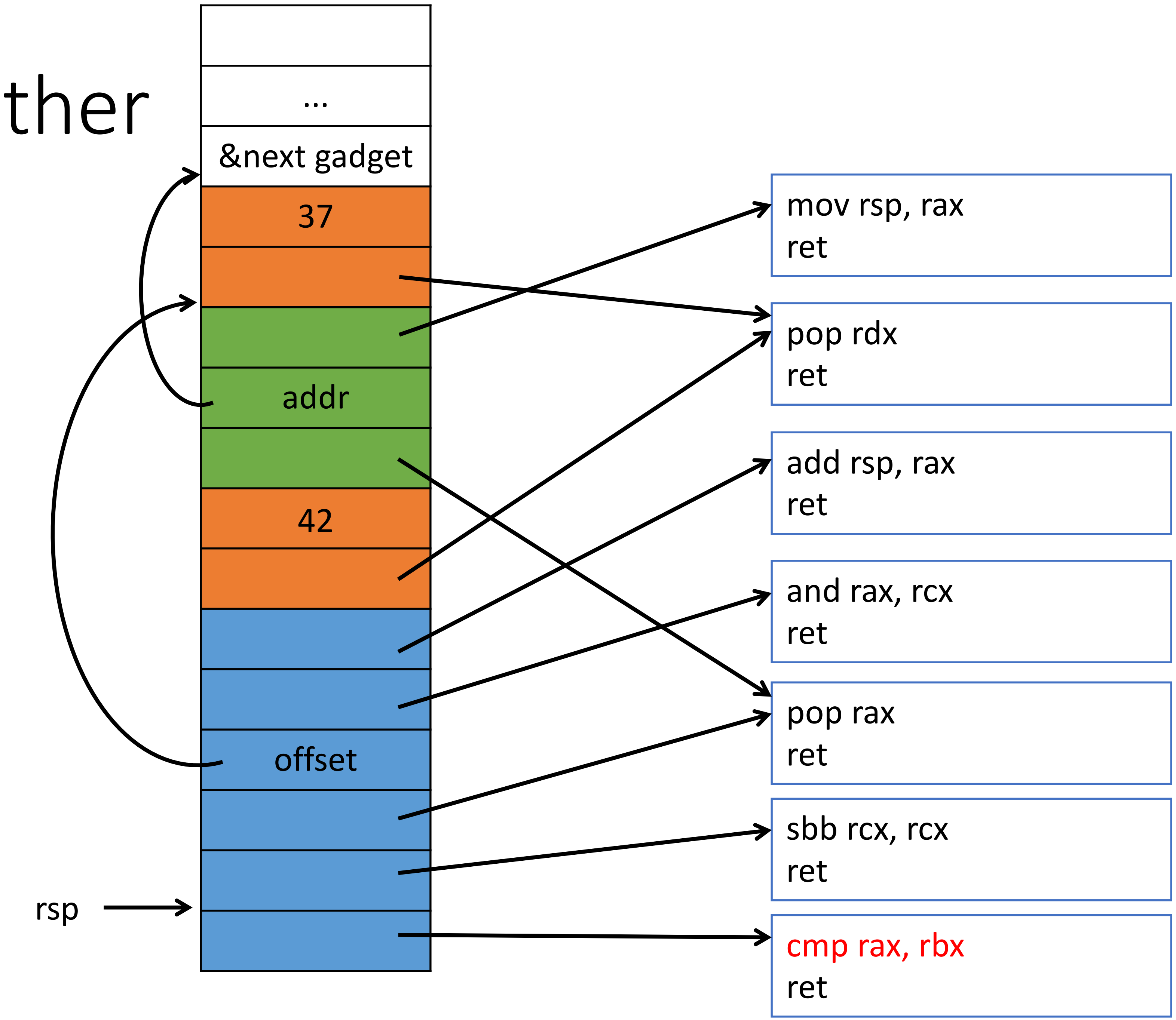
```
pop rax
ret
```

```
sbb rcx, rcx
ret
```

```
cmp rax, rbx
ret
```

Putting it together

Register	Value
rax	10
rbx	20
rcx	108
rdx	17



…

&next gadget

37

addr

42

offset

rsp

mov rsp, rax
ret

pop rdx
ret

add rsp, rax
ret

and rax, rcx
ret

pop rax
ret

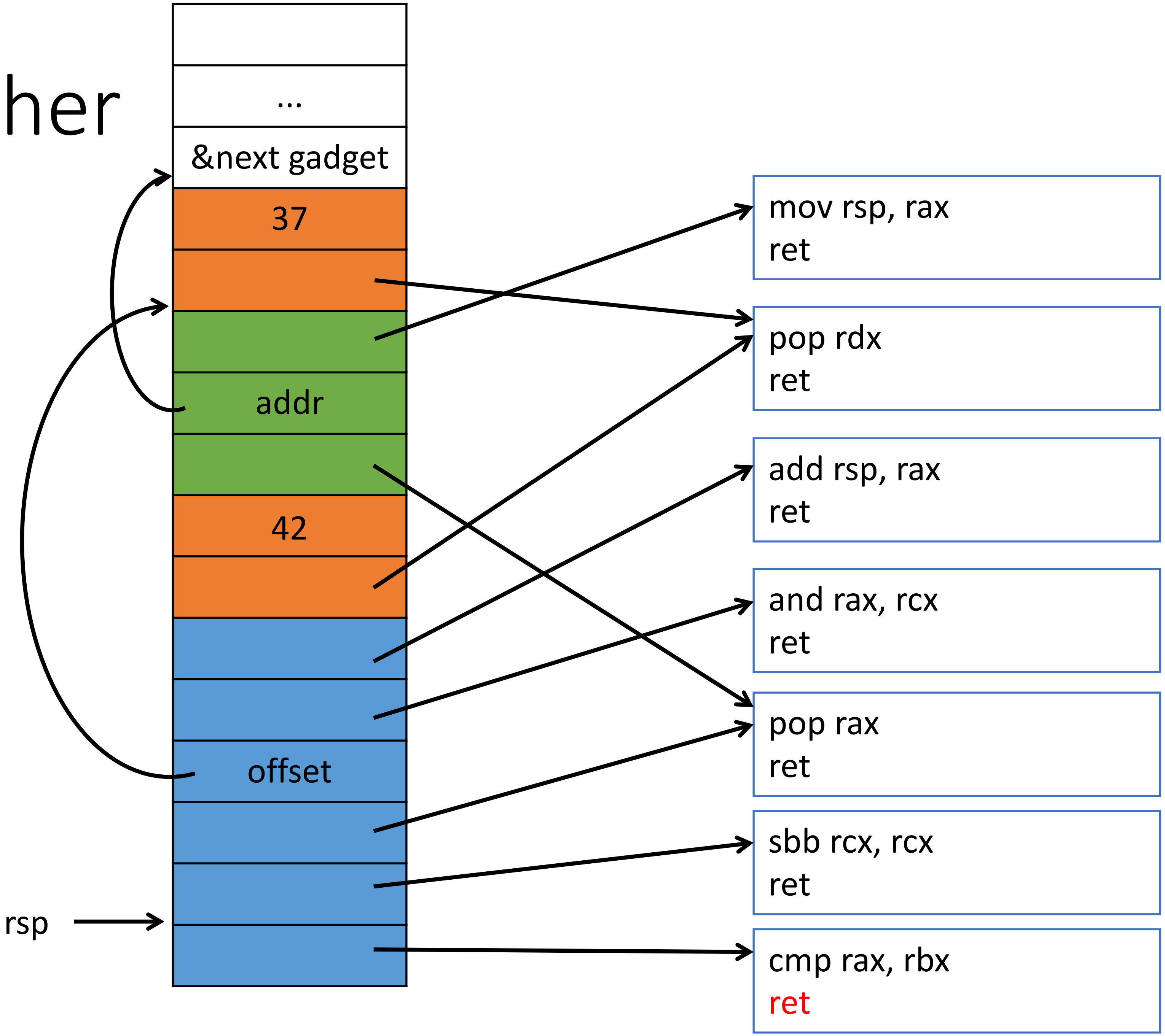
sbb rcx, rcx
ret

cmp rax, rbx
ret

Putting it together

Register	Value
rax	10
rbx	20
rcx	108
rdx	17

cf = 1



```
mov rsp, rax
ret
```

```
pop rdx
ret
```

```
add rsp, rax
ret
```

```
and rax, rcx
ret
```

```
pop rax
ret
```

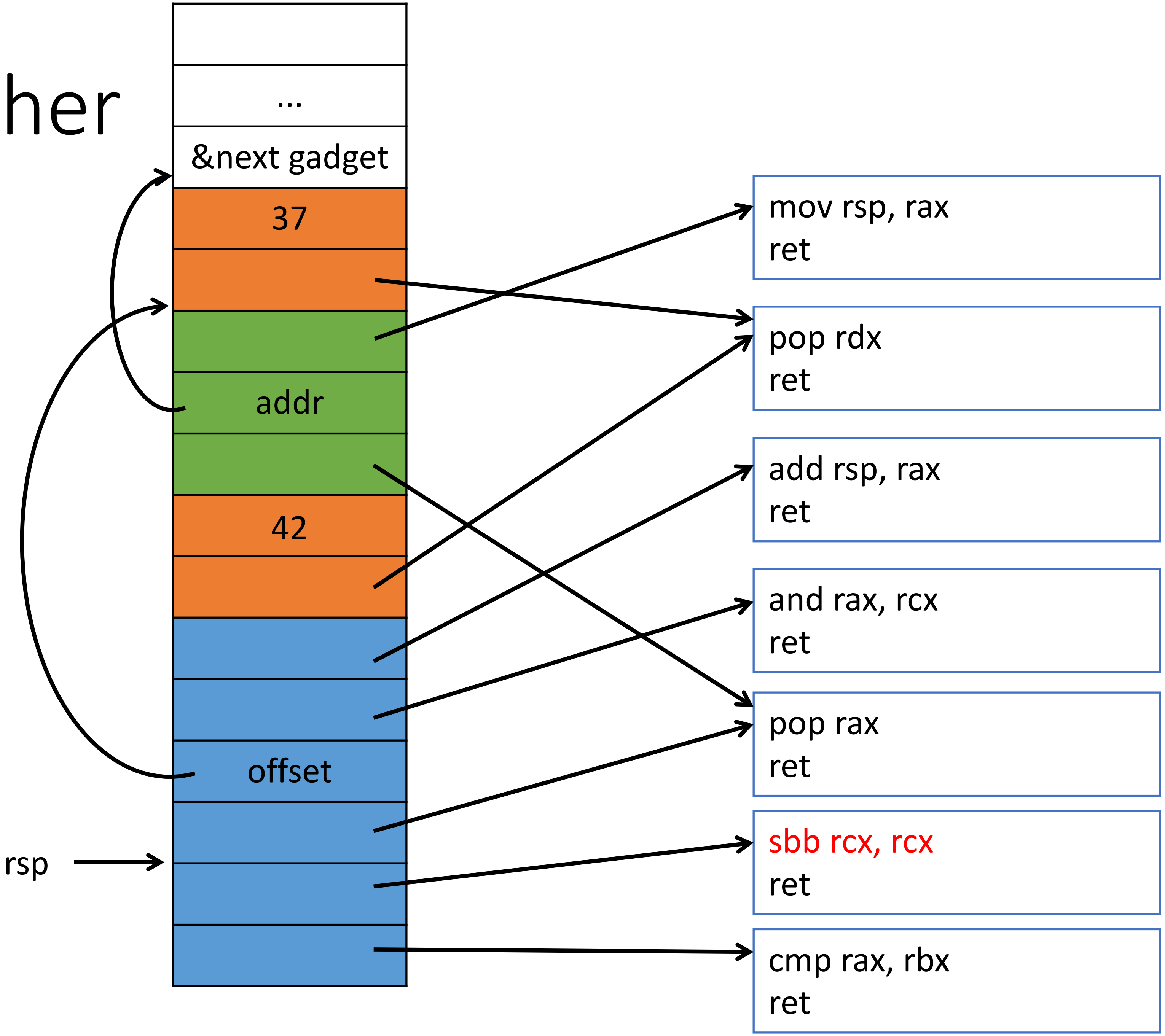
```
sbb rcx, rcx
ret
```

```
cmp rax, rbx
ret
```

Putting it together

Register	Value
rax	10
rbx	20
rcx	108
rdx	17

cf = 1



```
mov rsp, rax
ret
```

```
pop rdx
ret
```

```
add rsp, rax
ret
```

```
and rax, rcx
ret
```

```
pop rax
ret
```

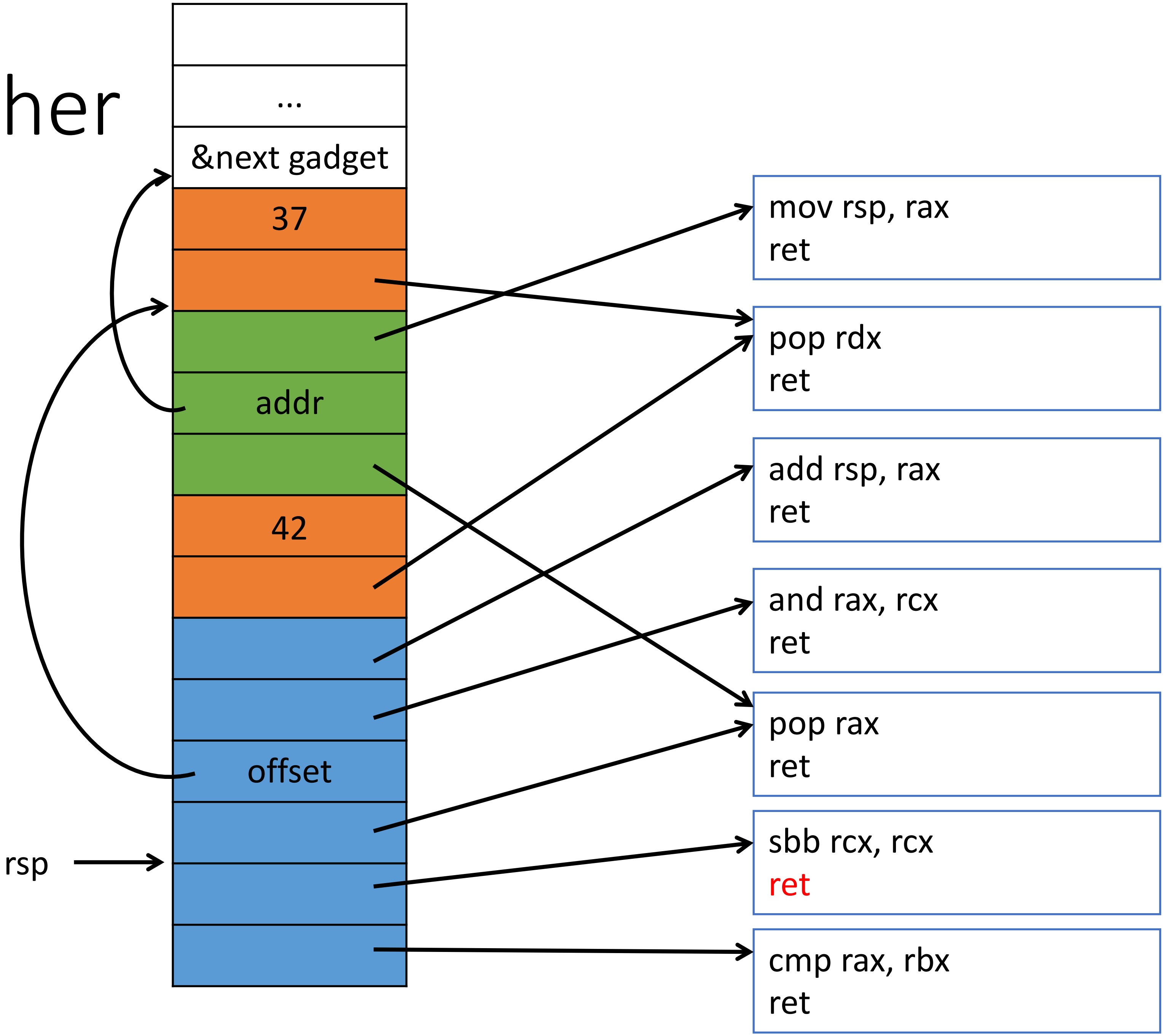
```
sbb rcx, rcx
ret
```

```
cmp rax, rbx
ret
```

Putting it together

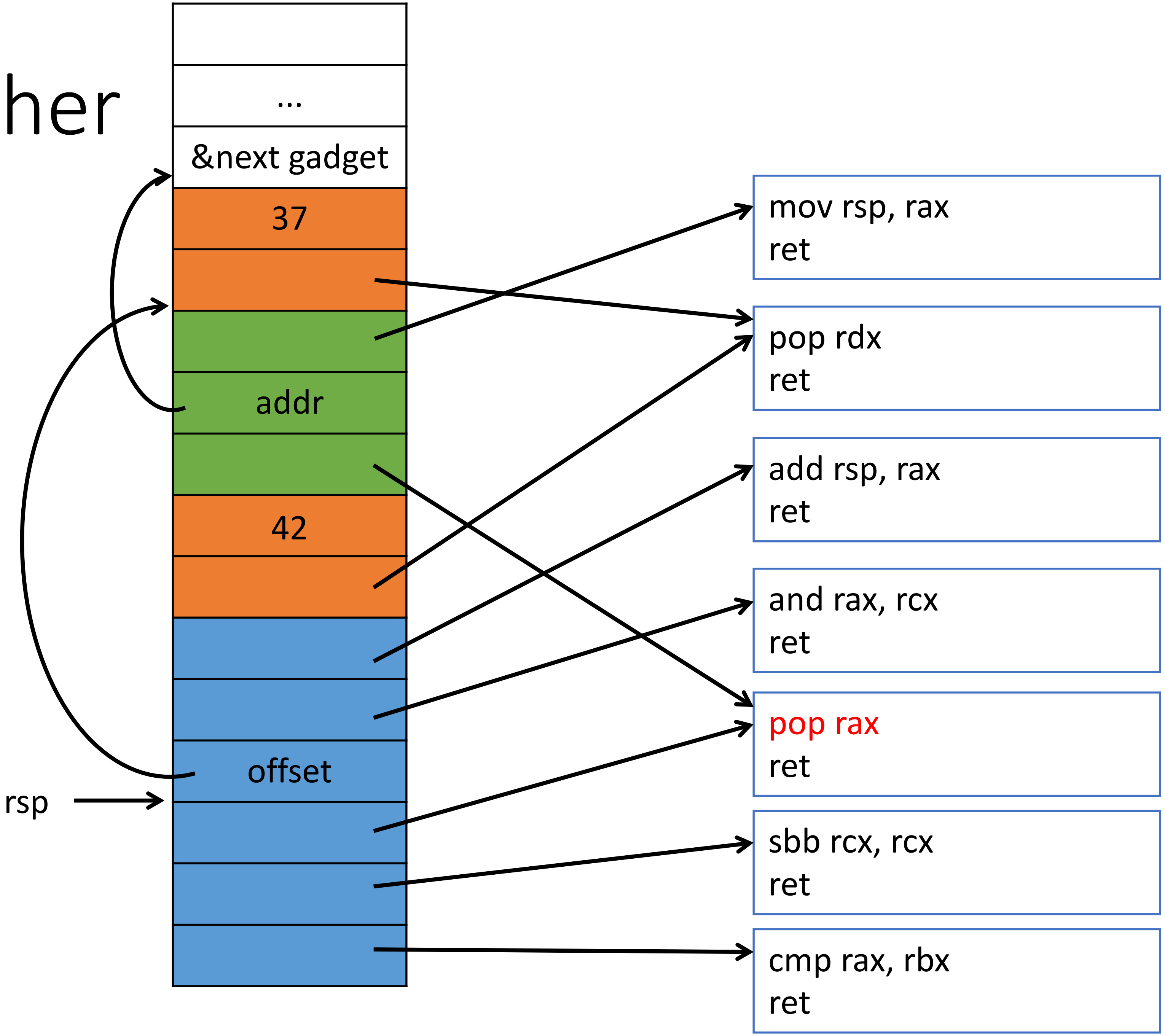
Register	Value
rax	10
rbx	20
rcx	-1
rdx	17

cf = 1



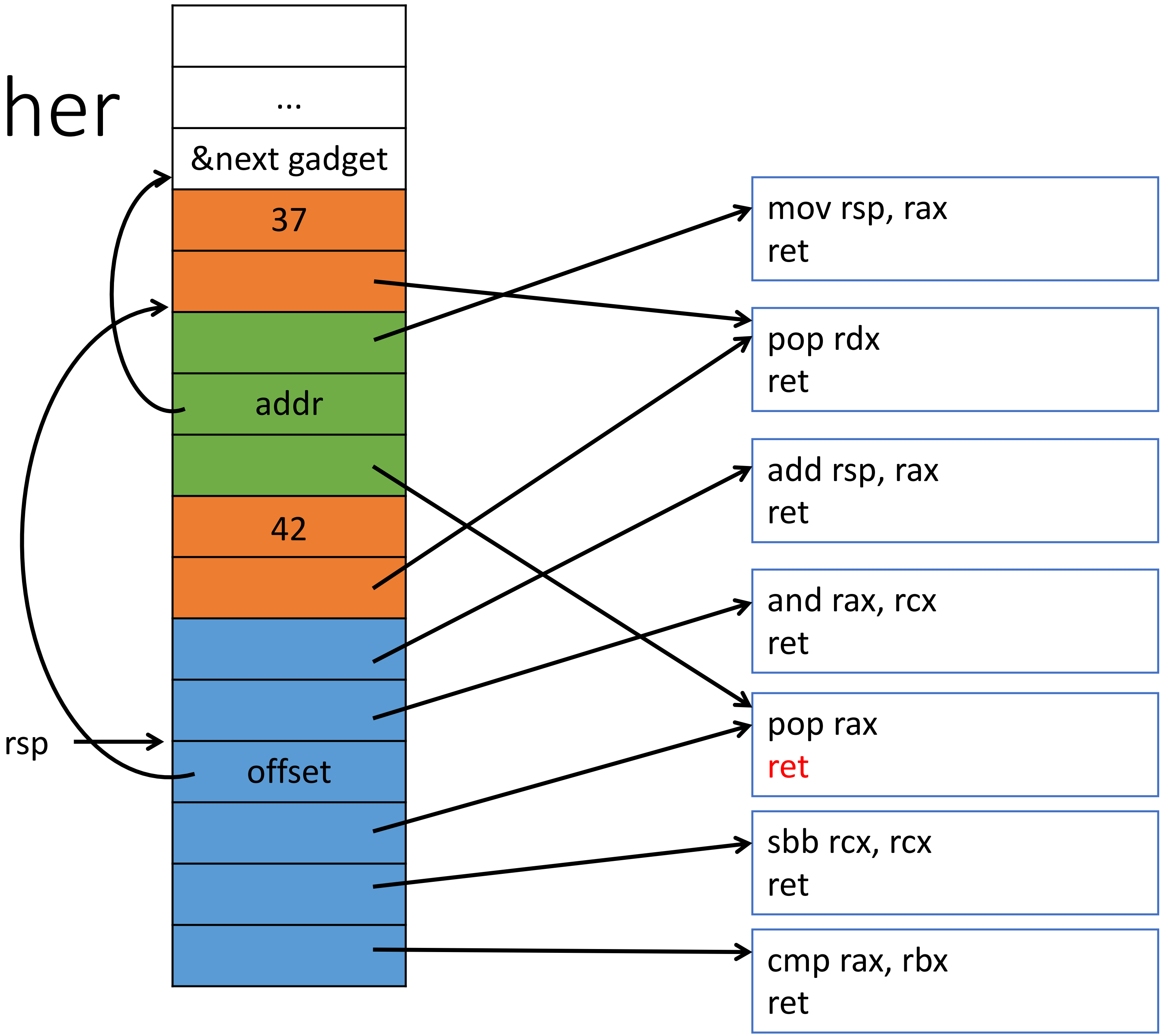
Putting it together

Register	Value
rax	10
rbx	20
rcx	-1
rdx	17



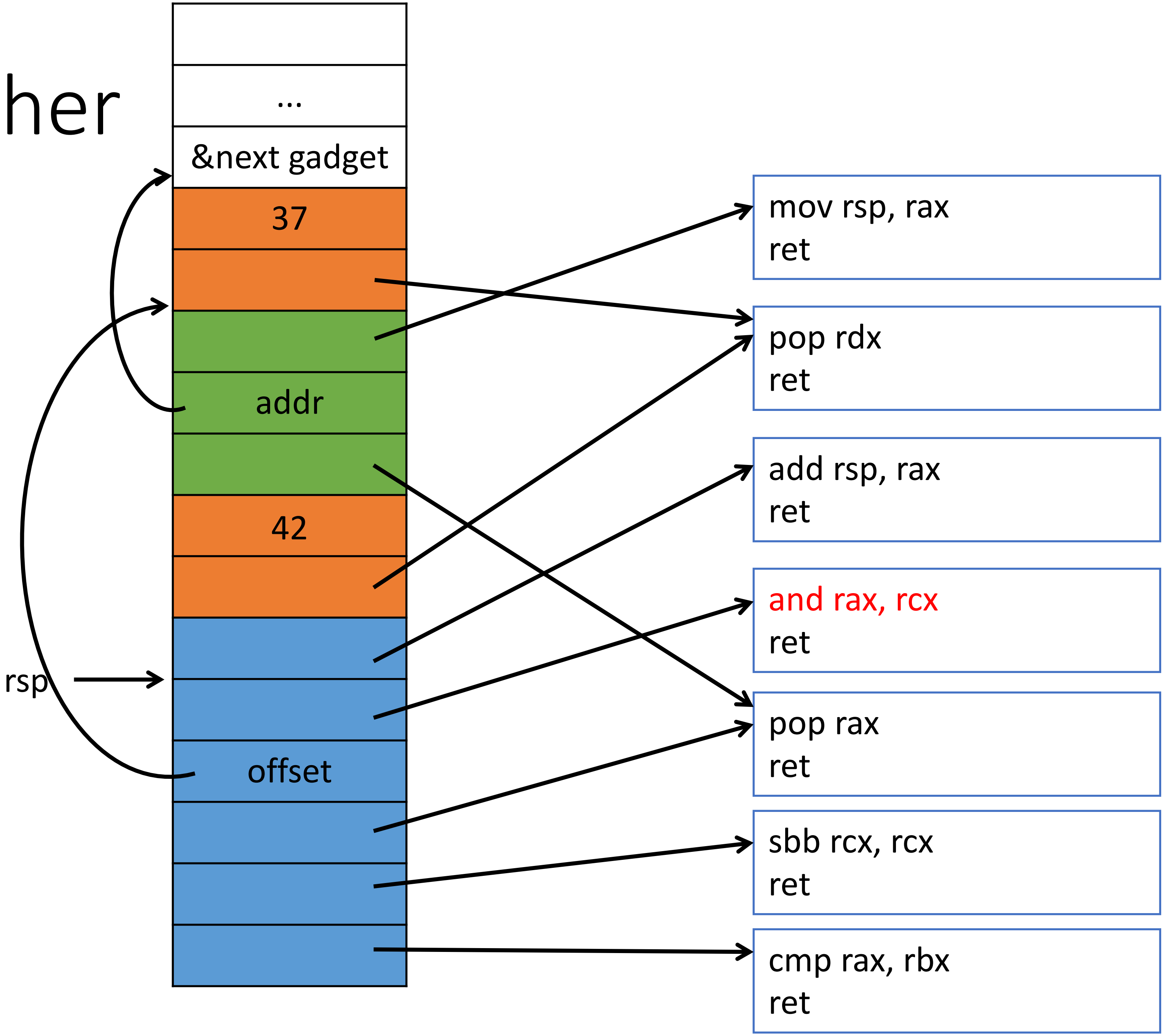
Putting it together

Register	Value
rax	40 = offset
rbx	20
rcx	-1
rdx	17



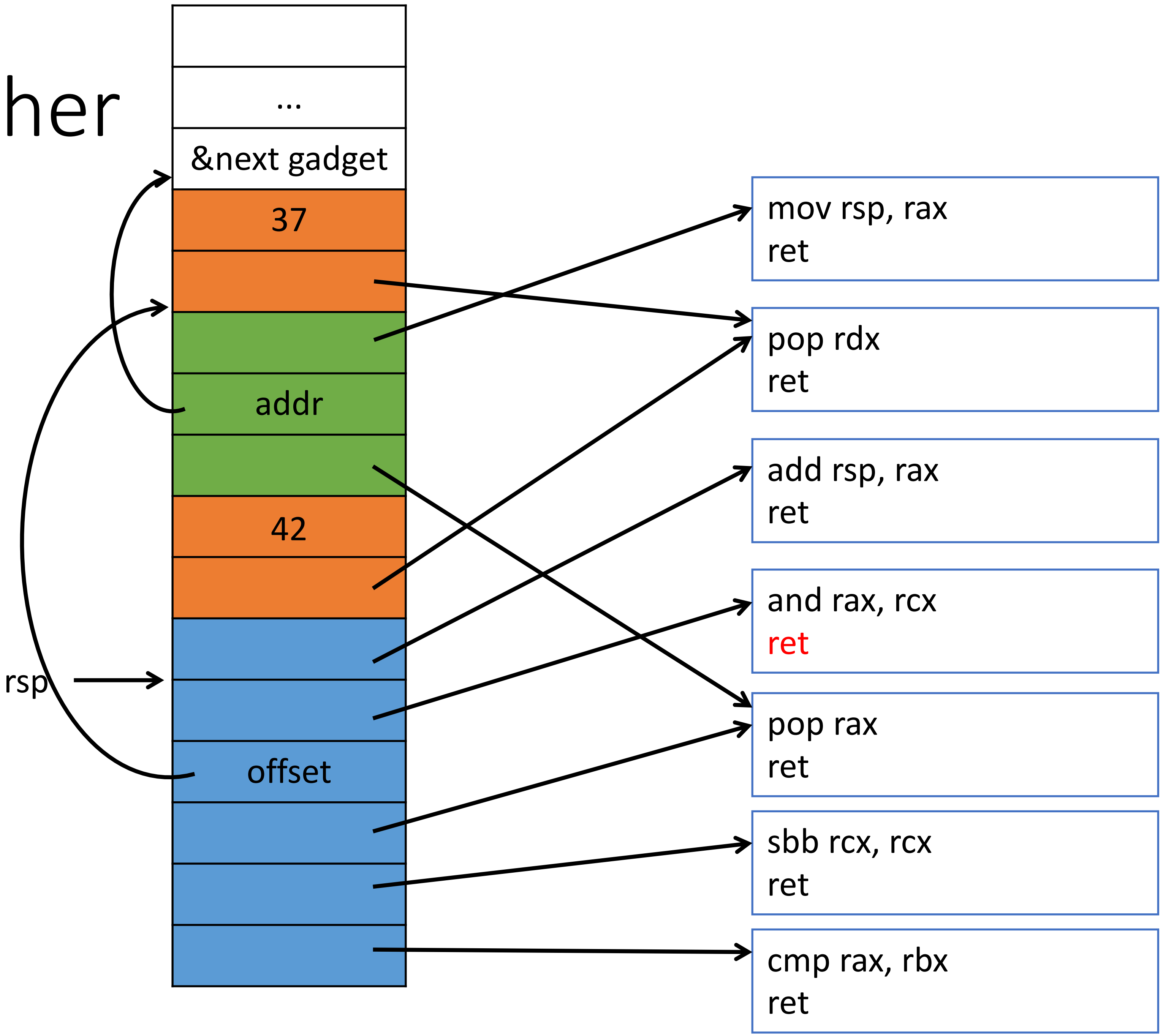
Putting it together

Register	Value
rax	40 = offset
rbx	20
rcx	-1
rdx	17



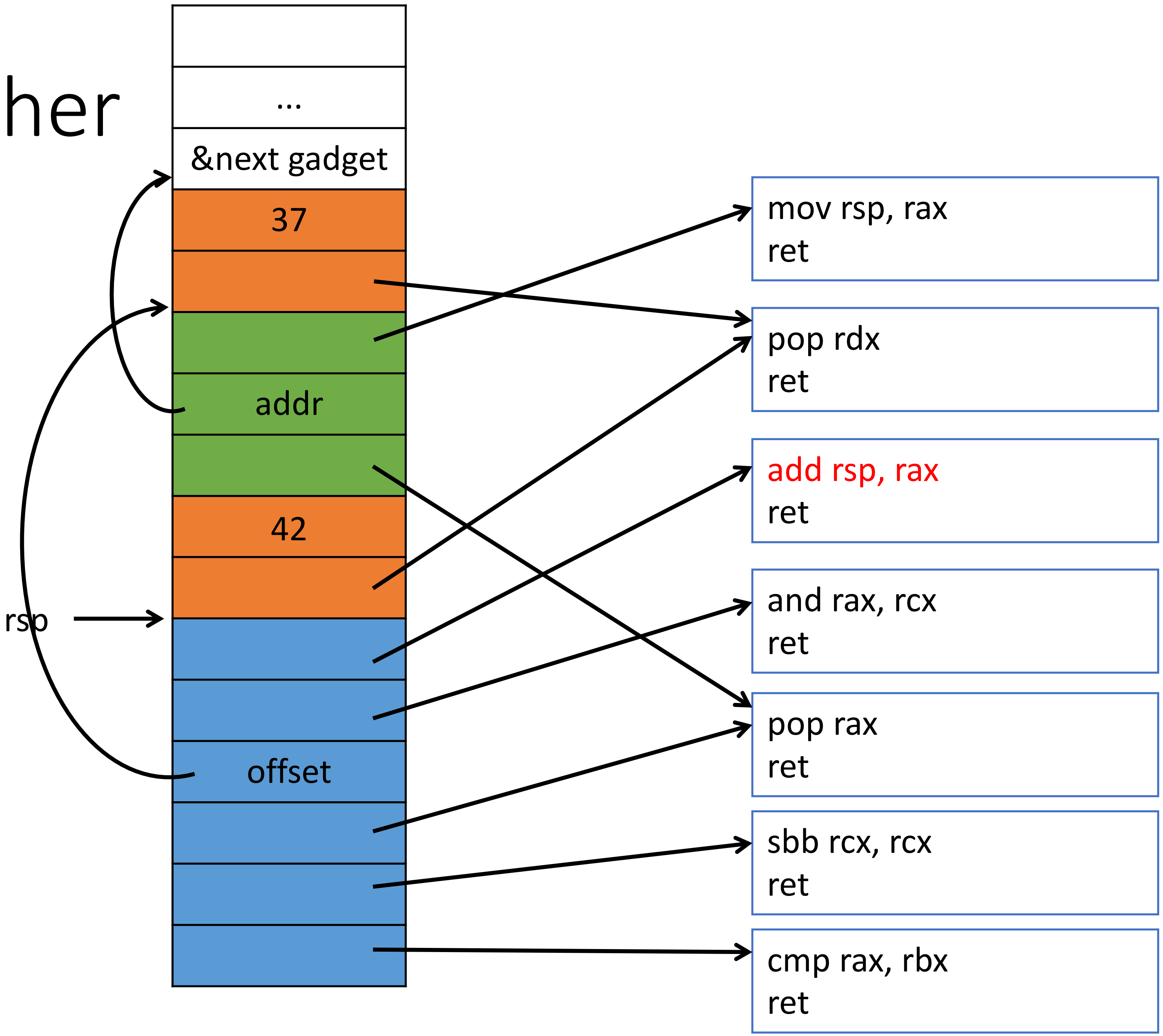
Putting it together

Register	Value
rax	40 = offset
rbx	20
rcx	-1
rdx	17

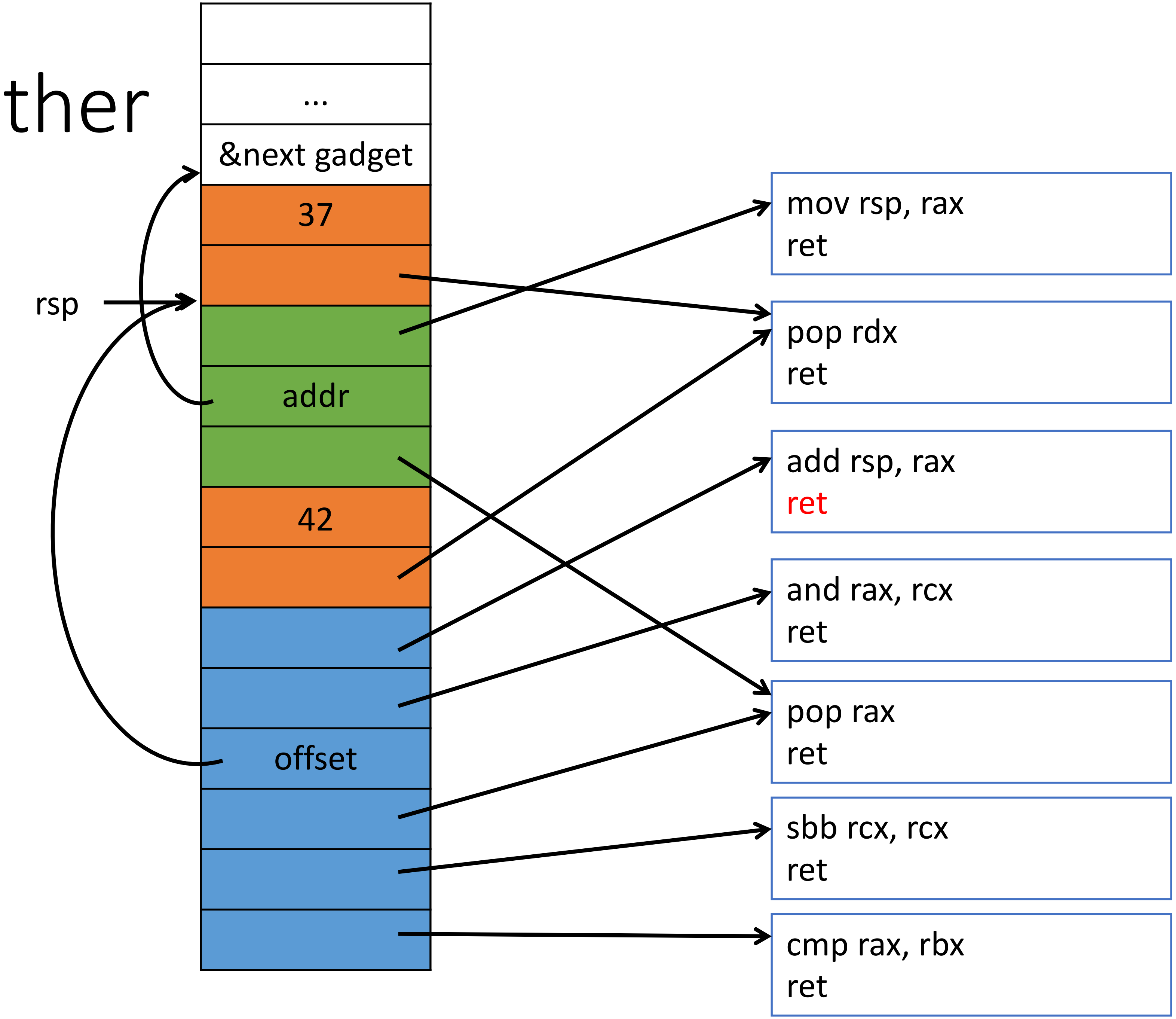


Putting it together

Register	Value
rax	40 = offset
rbx	20
rcx	-1
rdx	17



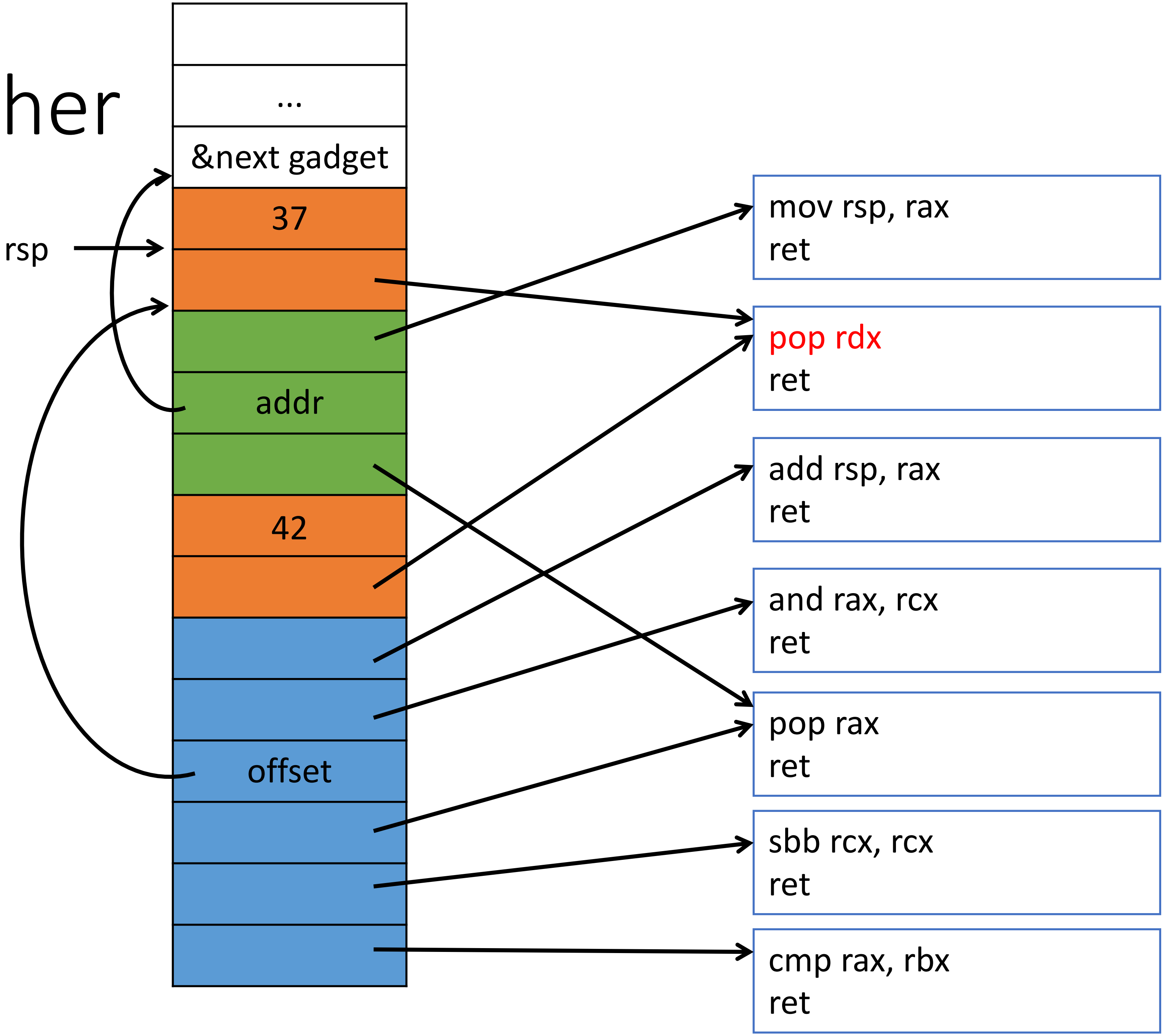
Putting it together



Register	Value
rax	40 = offset
rbx	20
rcx	-1
rdx	17

Putting it together

Register	Value
rax	40 = offset
rbx	20
rcx	-1
rdx	17



mov rsp, rax
ret

pop rdx
ret

add rsp, rax
ret

and rax, rcx
ret

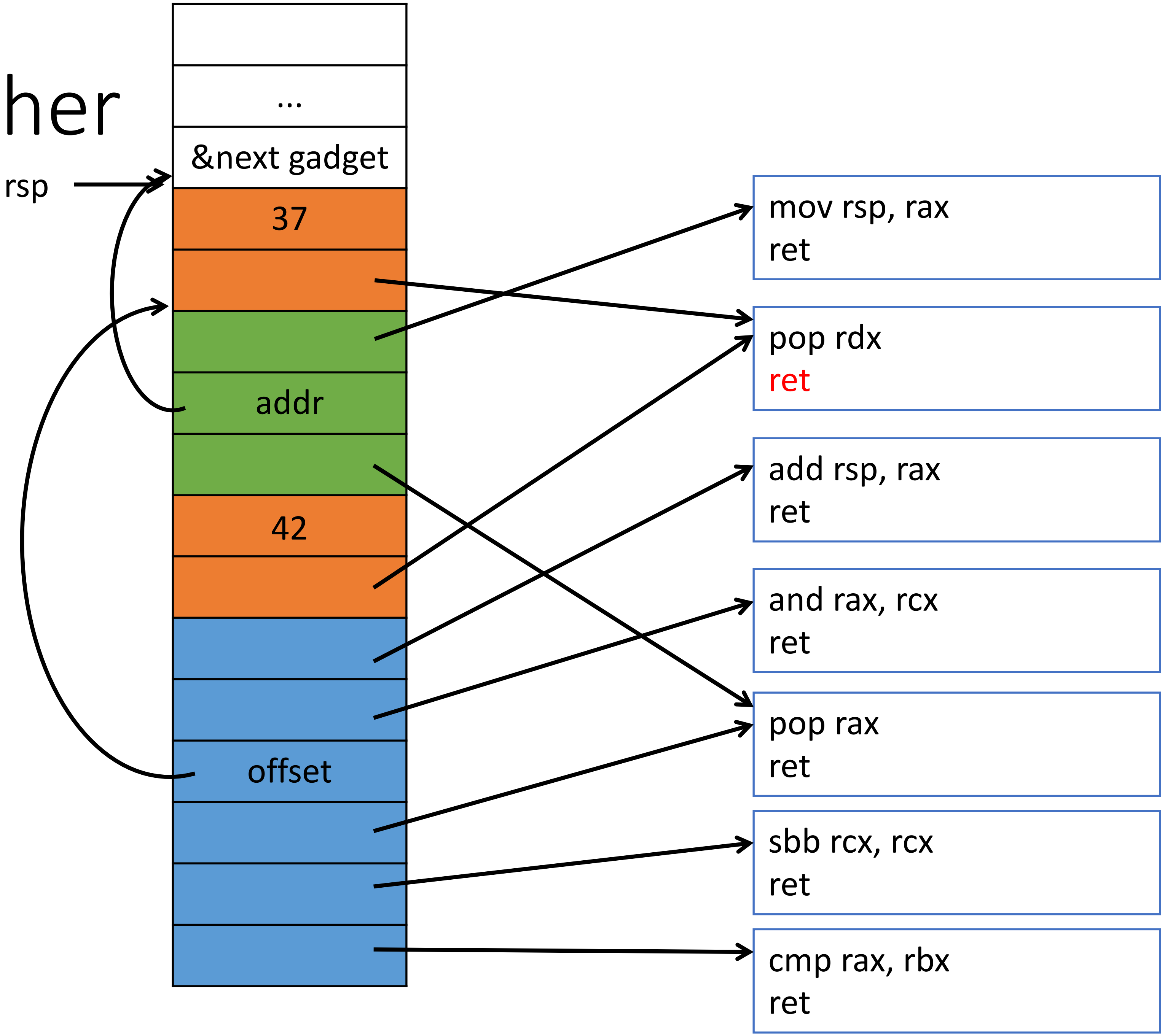
pop rax
ret

sbb rcx, rcx
ret

cmp rax, rbx
ret

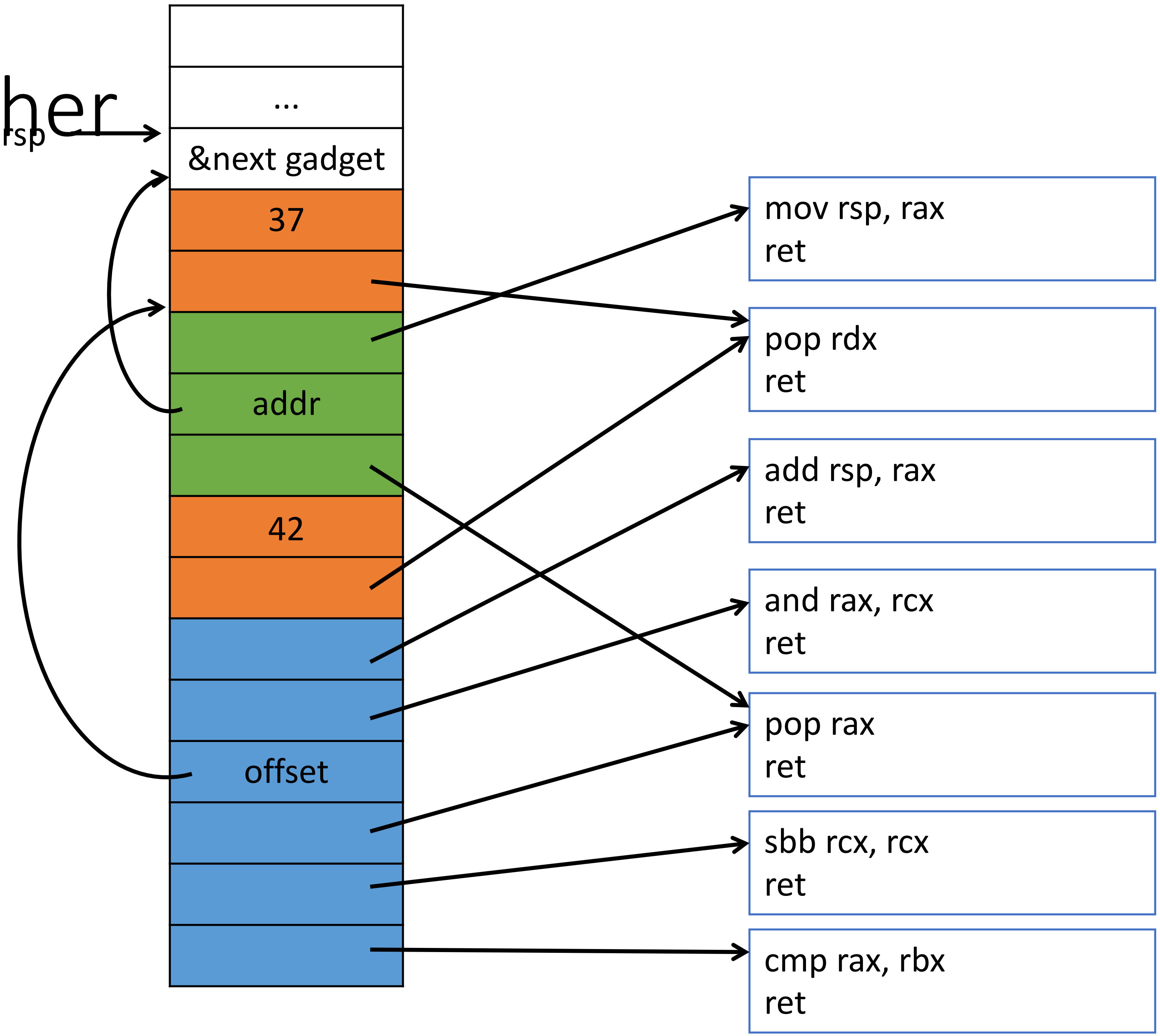
Putting it together

Register	Value
rax	40 = offset
rbx	20
rcx	-1
rdx	37



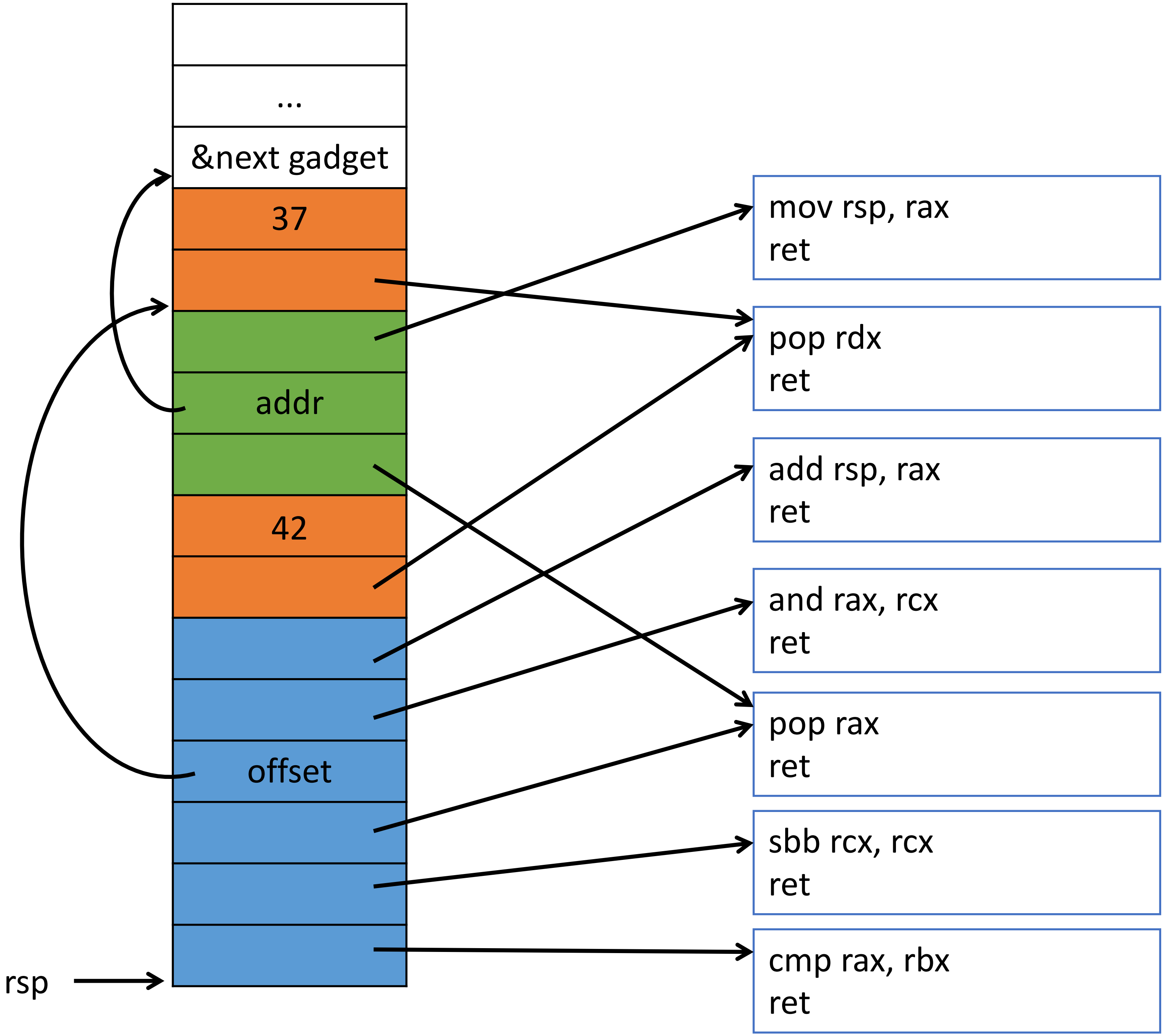
Putting it together

Register	Value
rax	40 = offset
rbx	20
rcx	-1
rdx	37



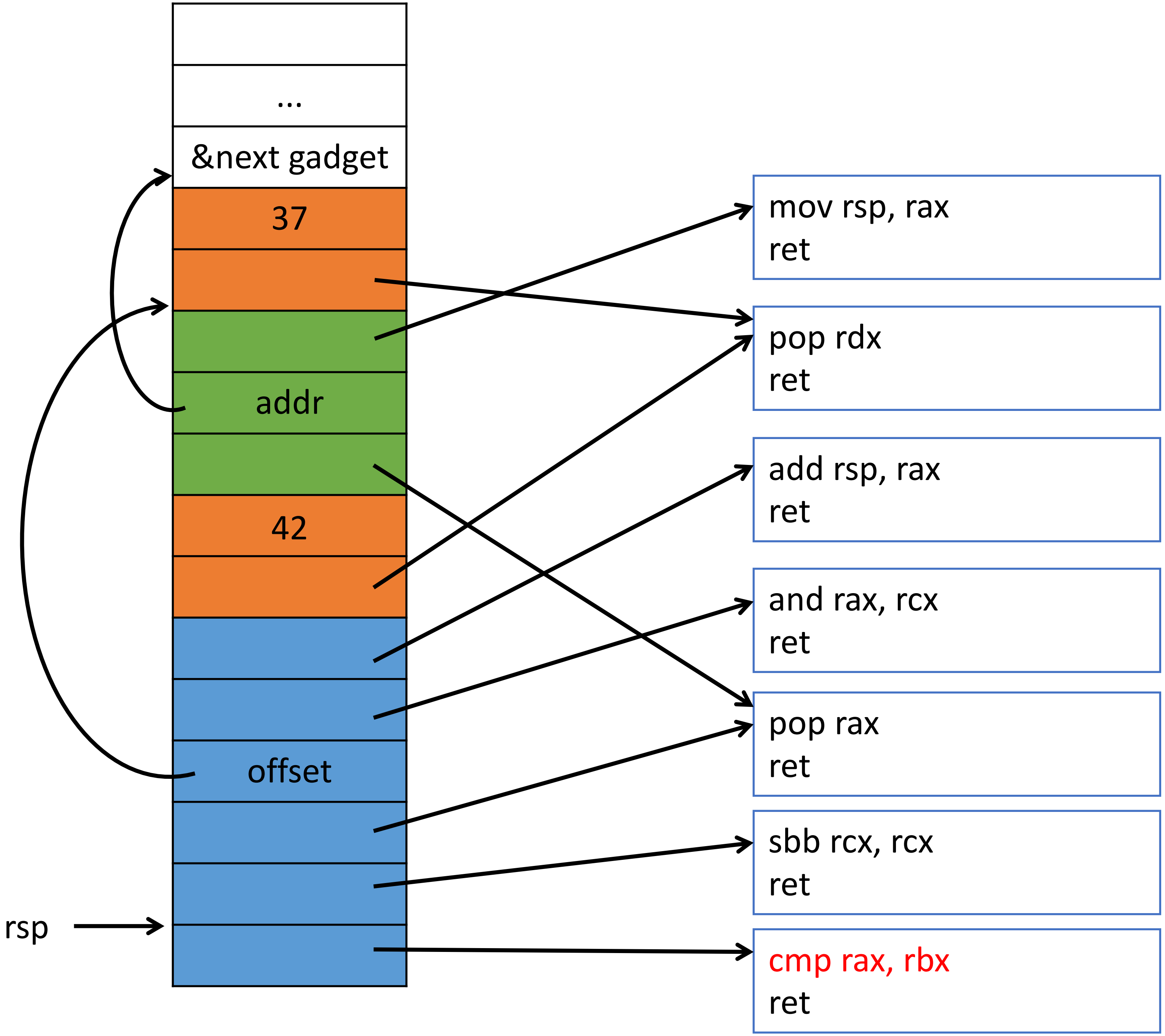
And again!

Register	Value
rax	500
rbx	20
rcx	108
rdx	17



And again!

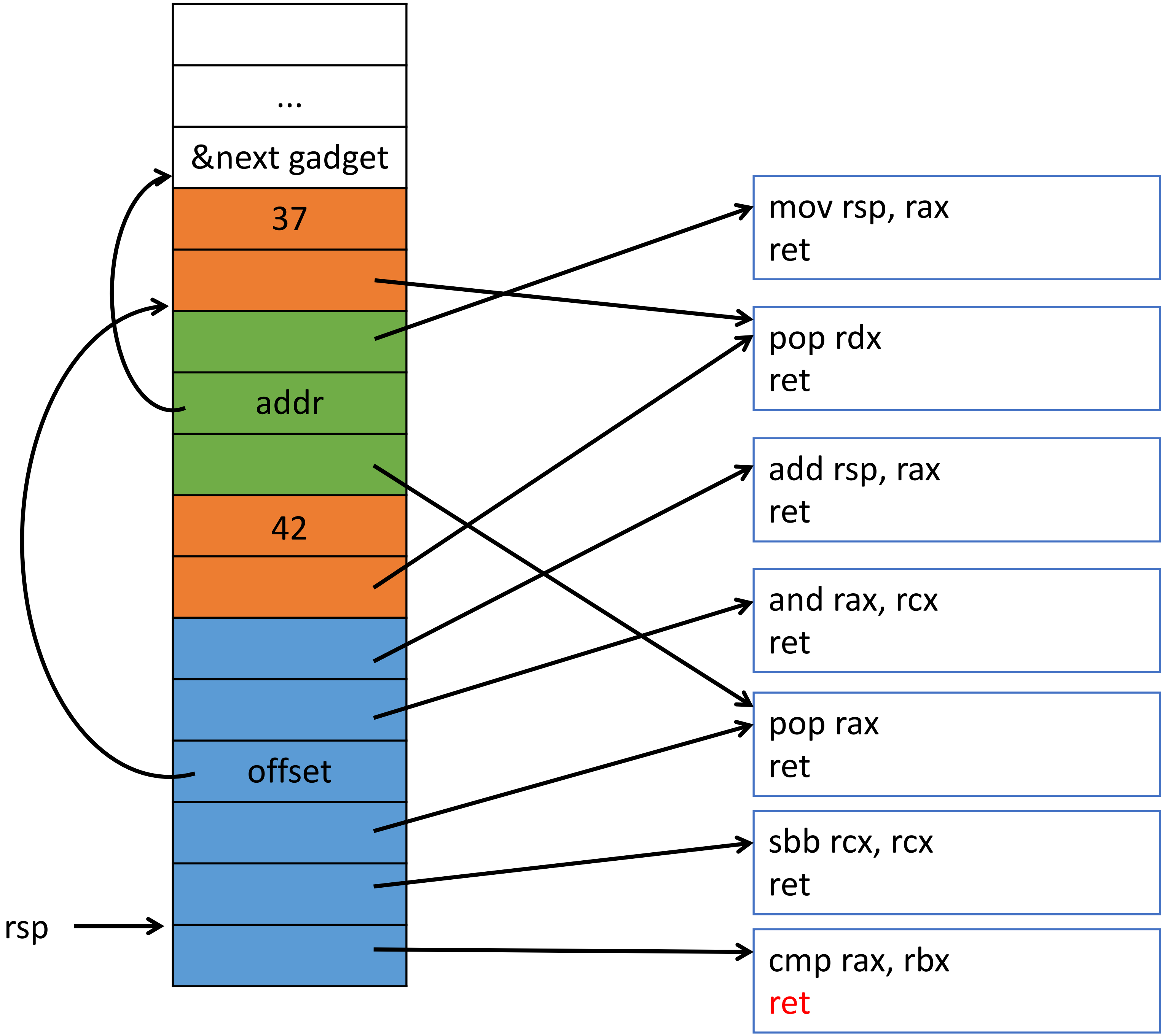
Register	Value
rax	500
rbx	20
rcx	108
rdx	17



And again!

Register	Value
rax	500
rbx	20
rcx	108
rdx	17

cf = 0



...

&next gadget

37

addr

42

offset

rsp

mov rsp, rax
ret

pop rdx
ret

add rsp, rax
ret

and rax, rcx
ret

pop rax
ret

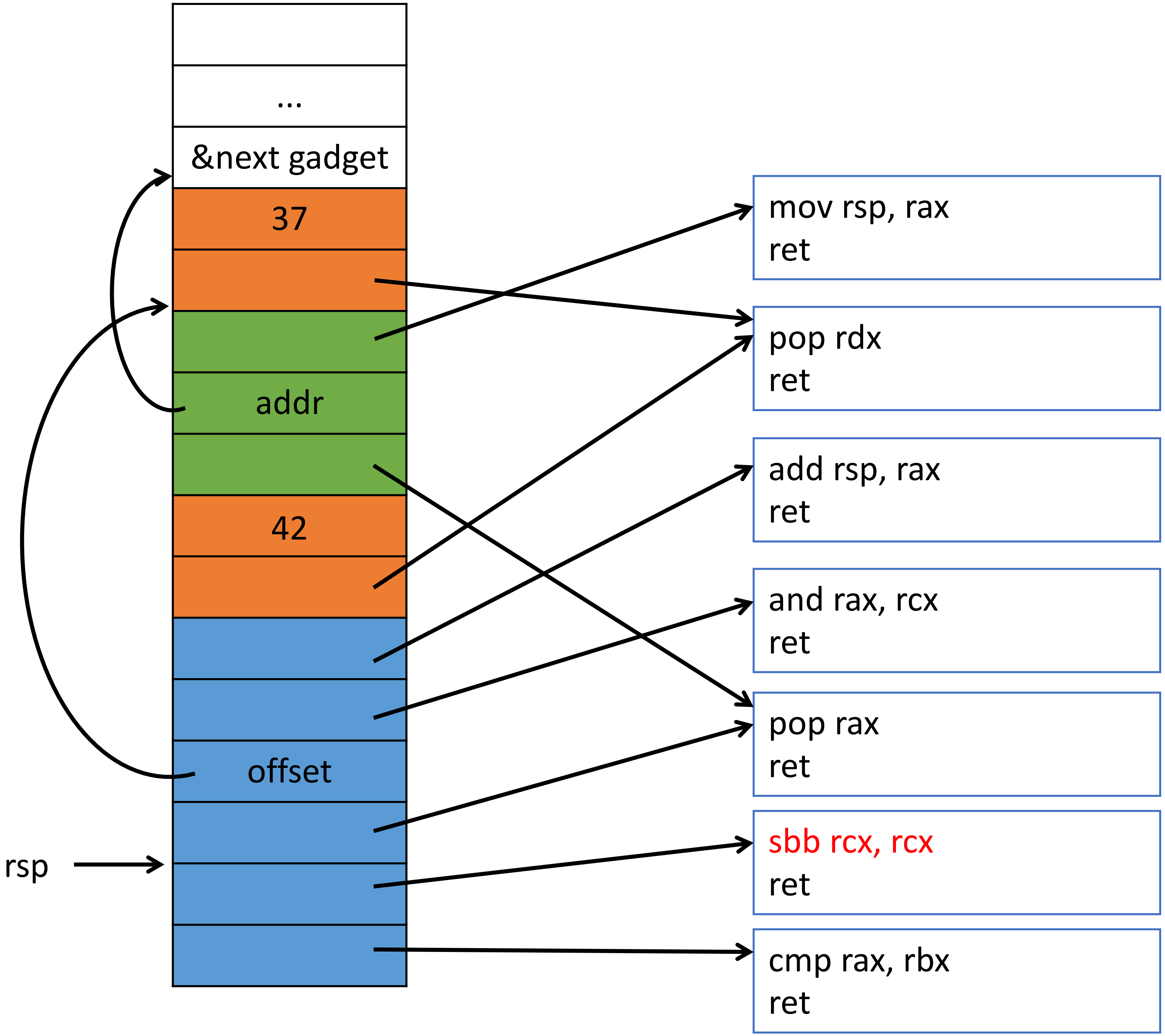
sbb rcx, rcx
ret

cmp rax, rbx
ret

And again!

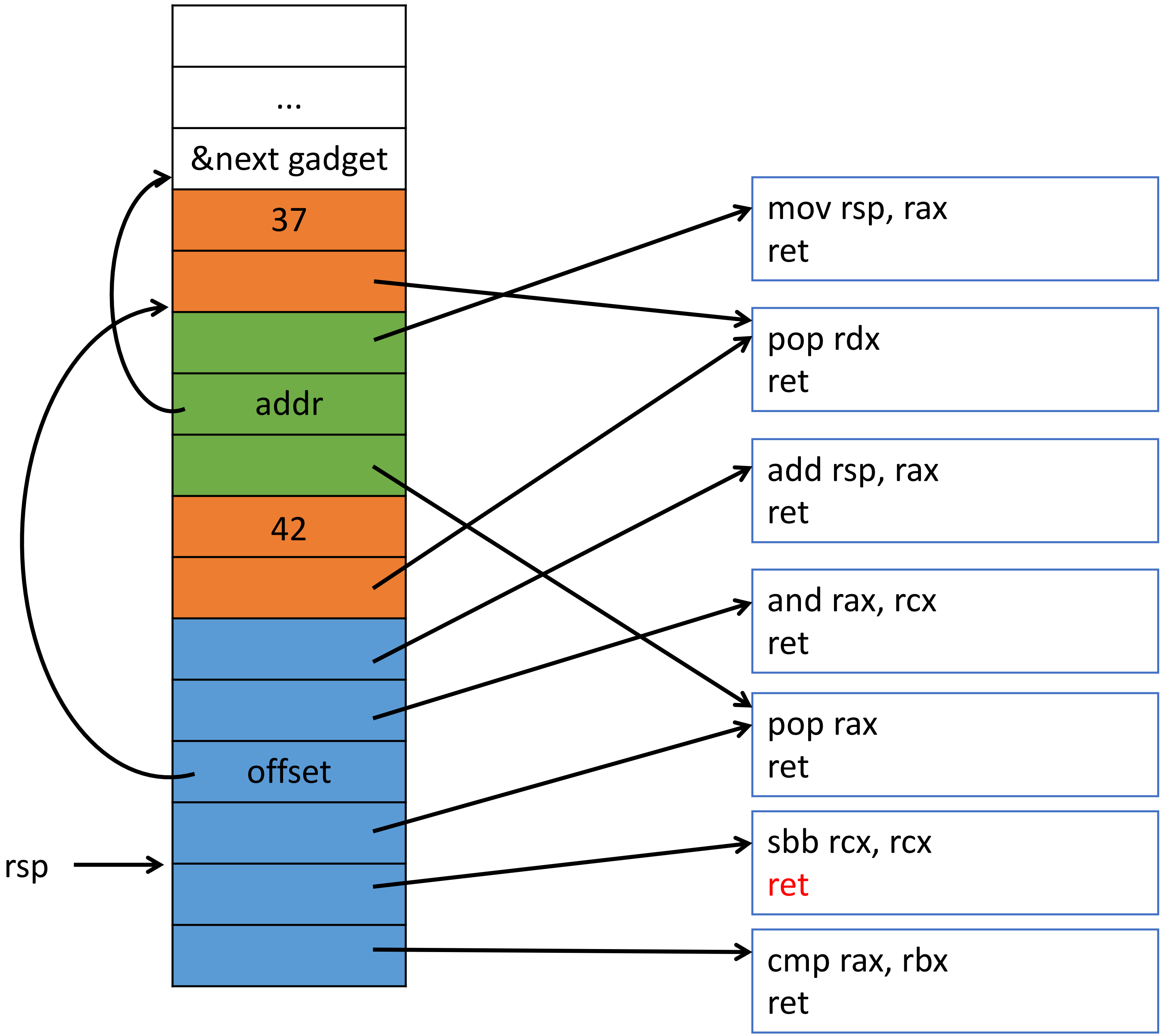
Register	Value
rax	500
rbx	20
rcx	108
rdx	17

cf = 0



And again!

Register	Value
rax	500
rbx	20
rcx	0
rdx	17



…

&next gadget

37

addr

42

offset

rsp

```
mov rsp, rax
ret
```

```
pop rdx
ret
```

```
add rsp, rax
ret
```

```
and rax, rcx
ret
```

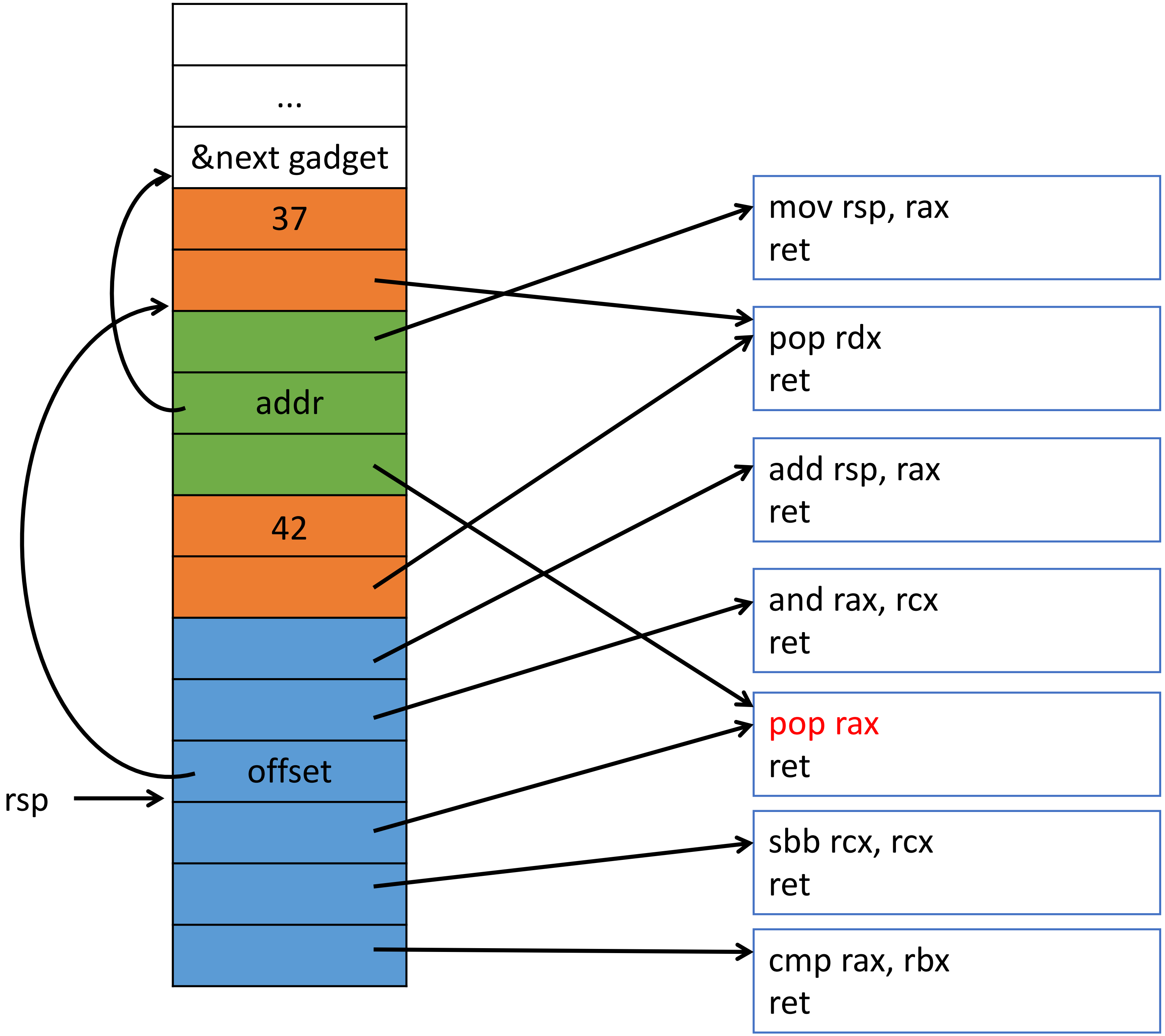
```
pop rax
ret
```

```
sbb rcx, rcx
ret
```

```
cmp rax, rbx
ret
```

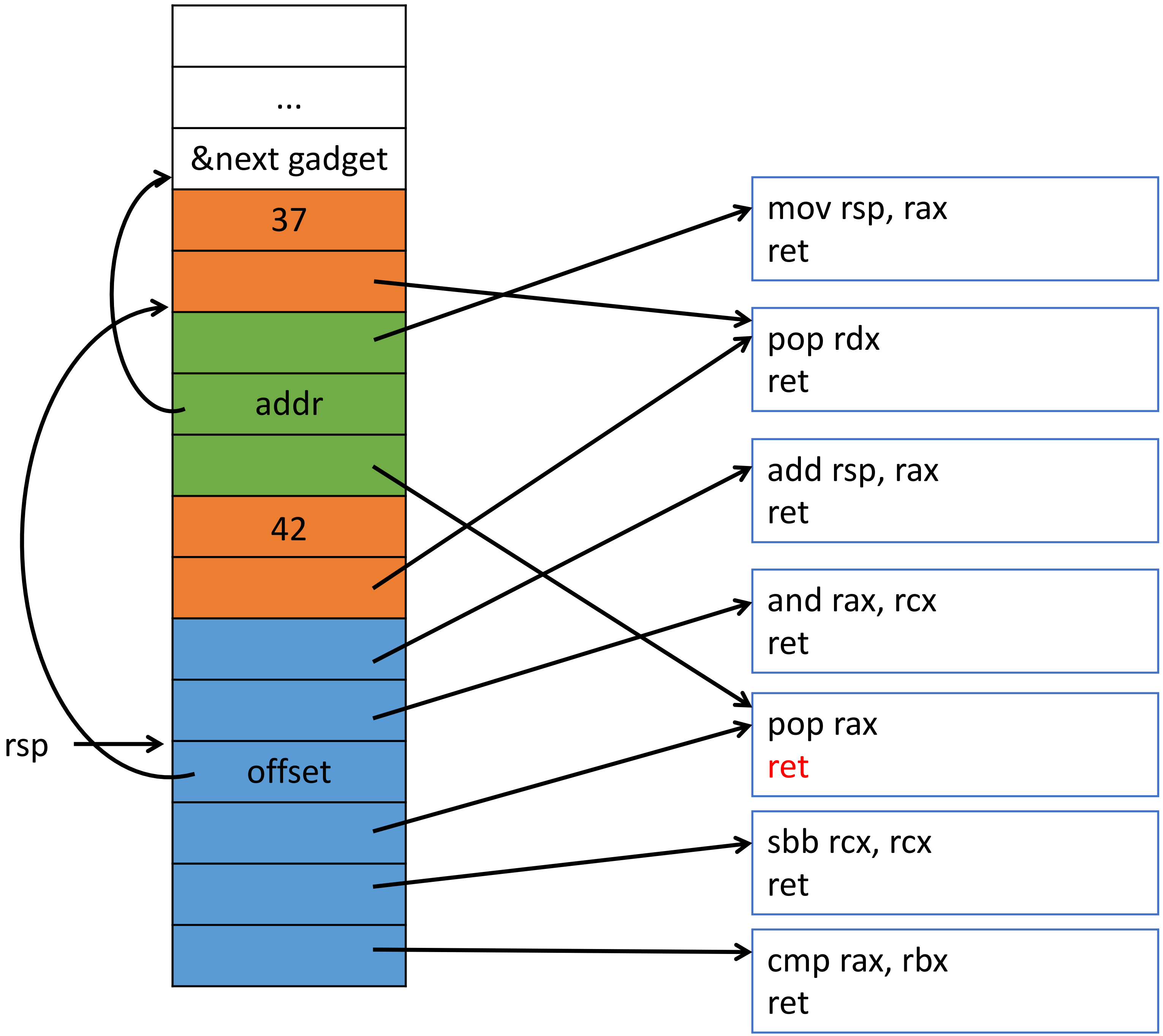
And again!

Register	Value
rax	500
rbx	20
rcx	0
rdx	17



And again!

Register	Value
rax	40 = offset
rbx	20
rcx	0
rdx	17



…

&next gadget

37

addr

42

rsp

offset

mov rsp, rax
ret

pop rdx
ret

add rsp, rax
ret

and rax, rcx
ret

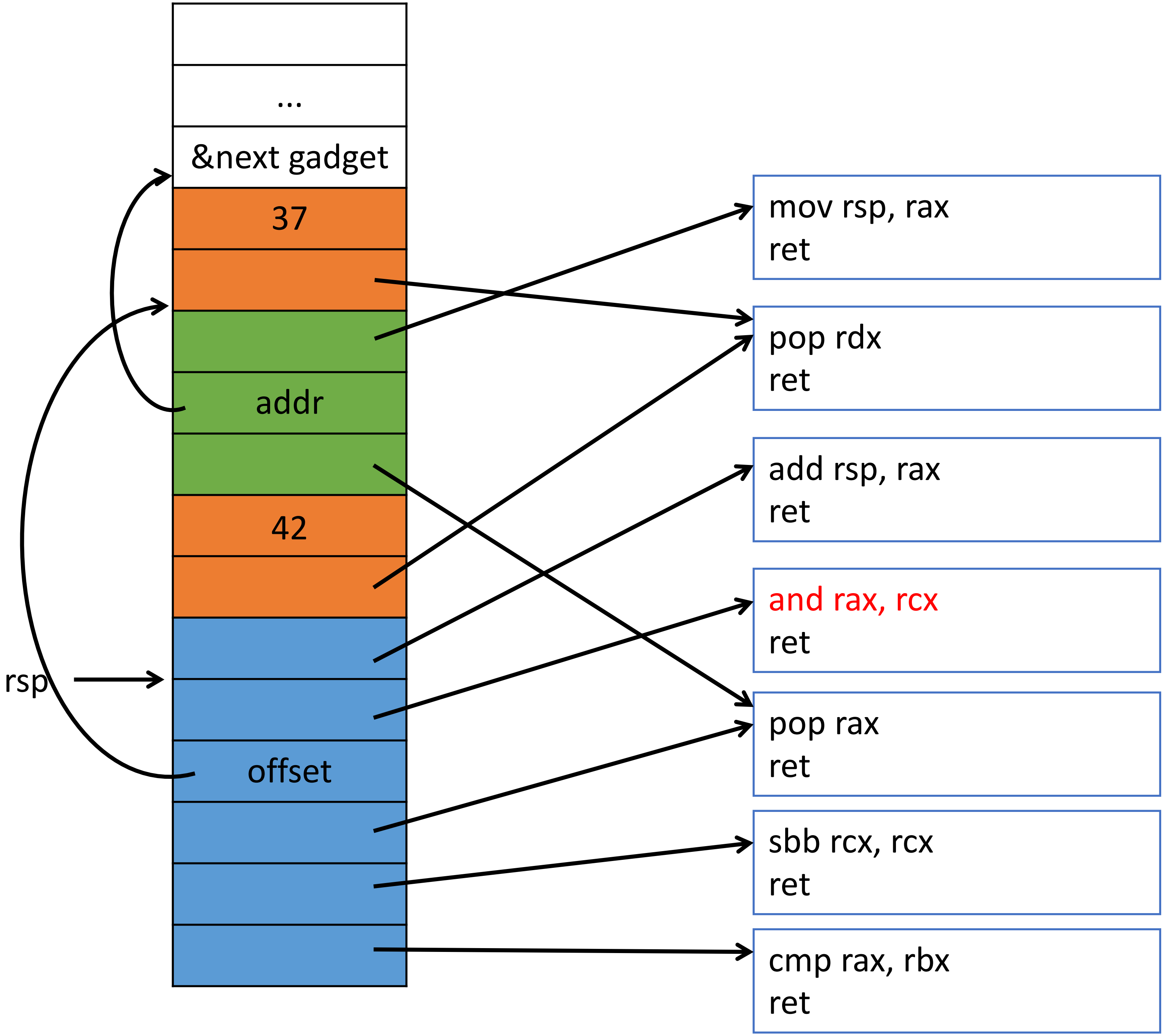
pop rax
ret

sbb rcx, rcx
ret

cmp rax, rbx
ret

And again!

Register	Value
rax	40 = offset
rbx	20
rcx	0
rdx	17



```
mov rsp, rax
ret
```

```
pop rdx
ret
```

```
add rsp, rax
ret
```

```
and rax, rcx
ret
```

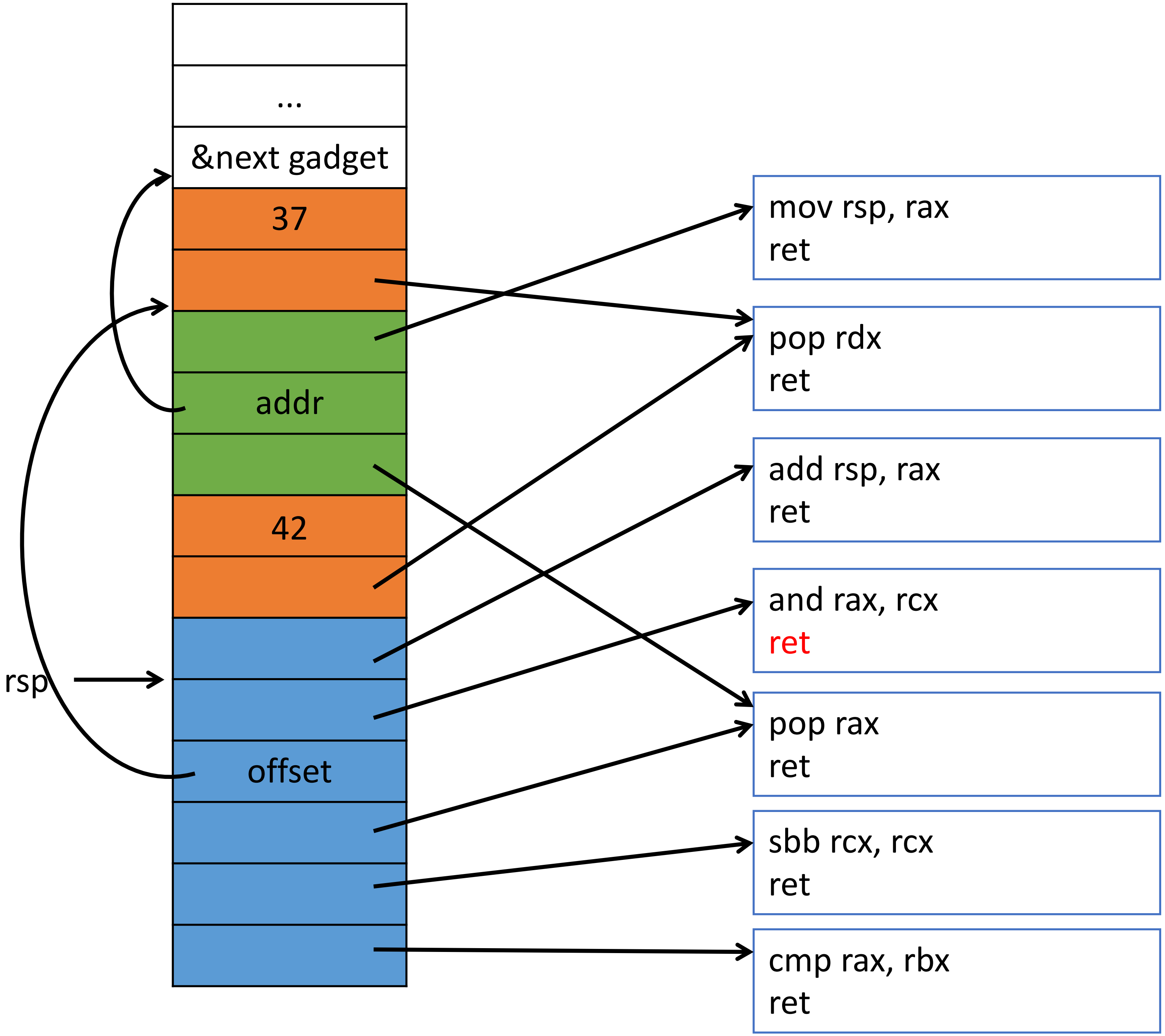
```
pop rax
ret
```

```
sbb rcx, rcx
ret
```

```
cmp rax, rbx
ret
```

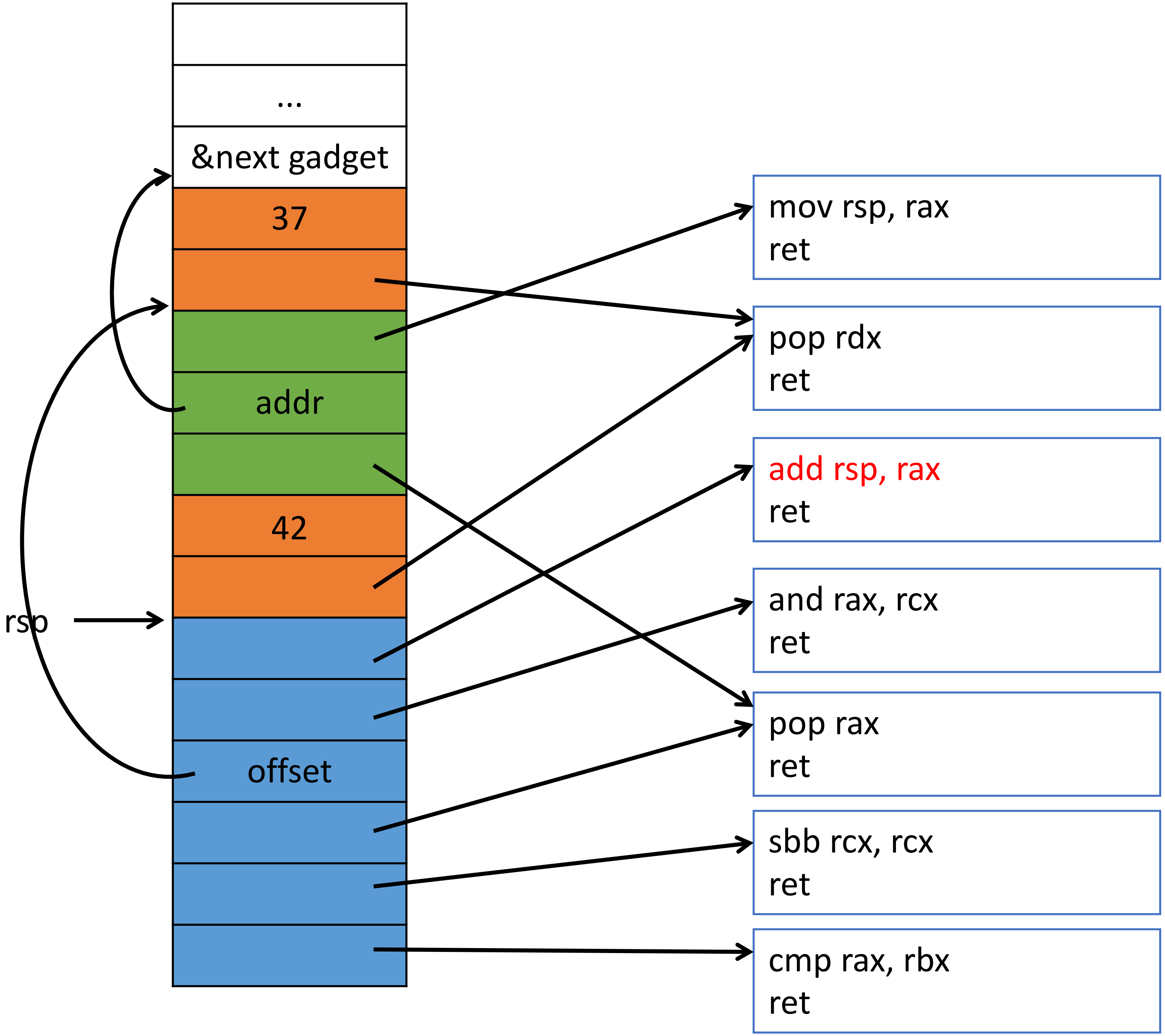
And again!

Register	Value
rax	0
rbx	20
rcx	0
rdx	17



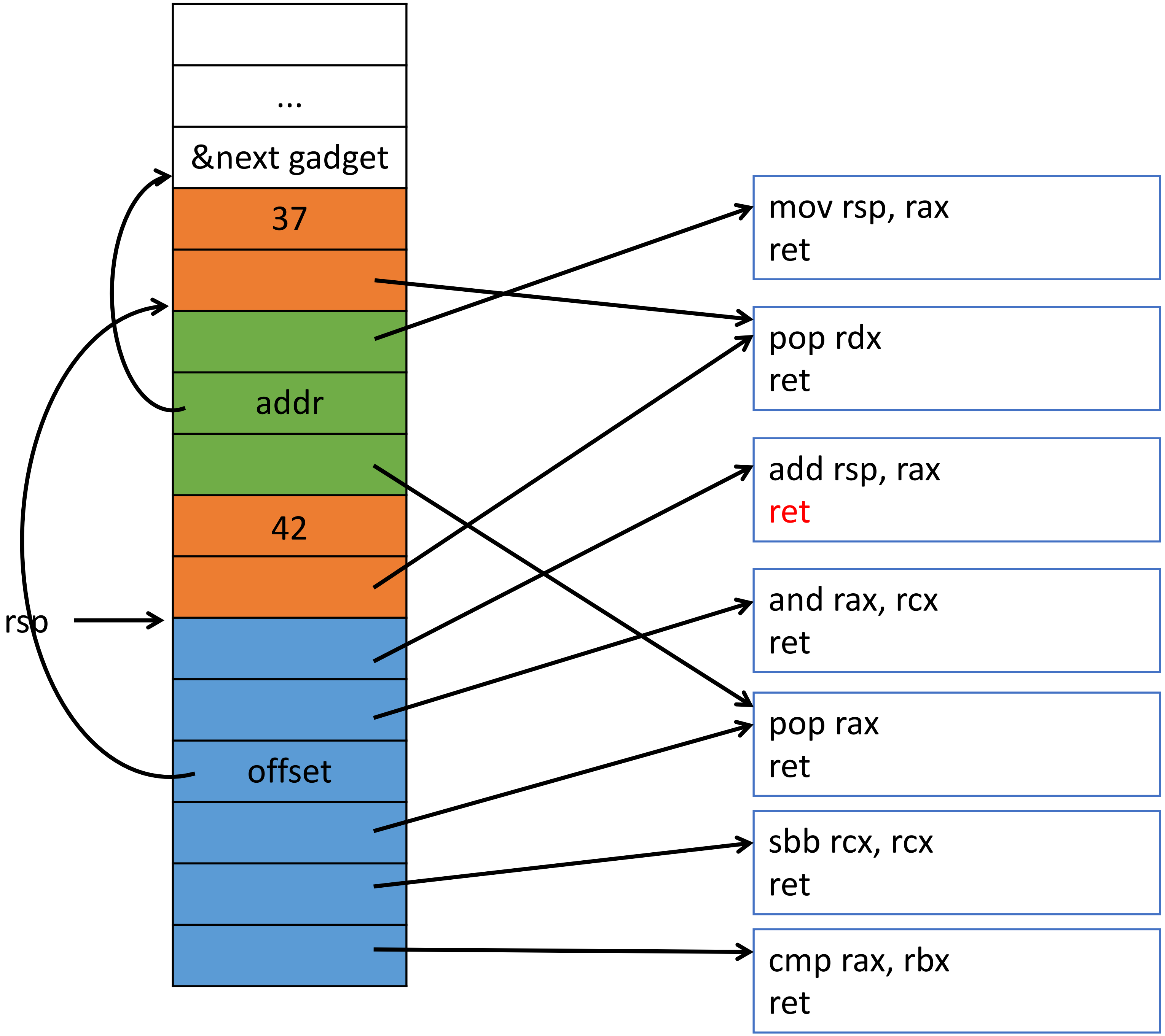
And again!

Register	Value
rax	0
rbx	20
rcx	0
rdx	17



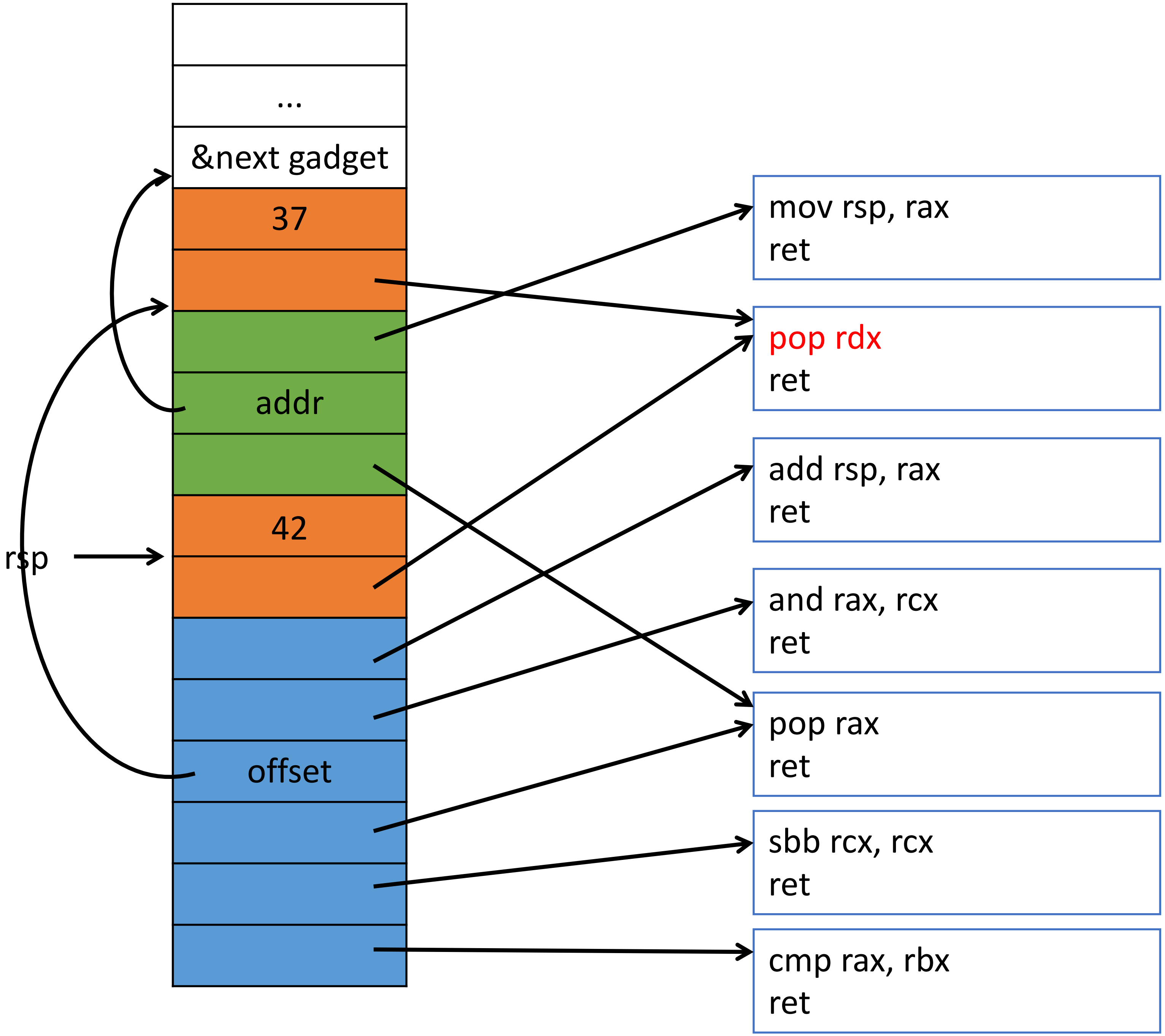
And again!

Register	Value
rax	0
rbx	20
rcx	0
rdx	17



And again!

Register	Value
rax	0
rbx	20
rcx	0
rdx	17



mov rsp, rax
ret

pop rdx
ret

add rsp, rax
ret

and rax, rcx
ret

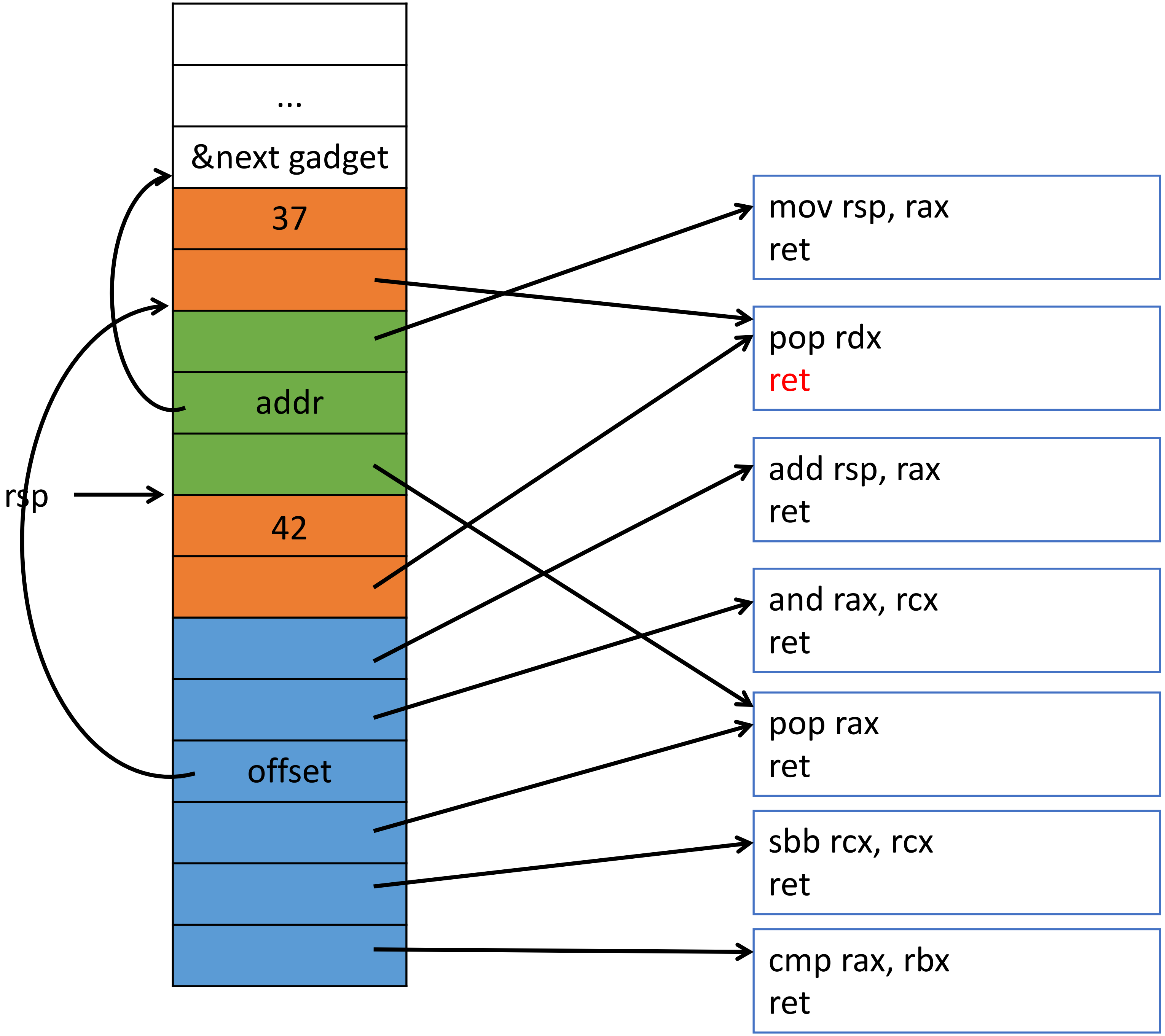
pop rax
ret

sbb rcx, rcx
ret

cmp rax, rbx
ret

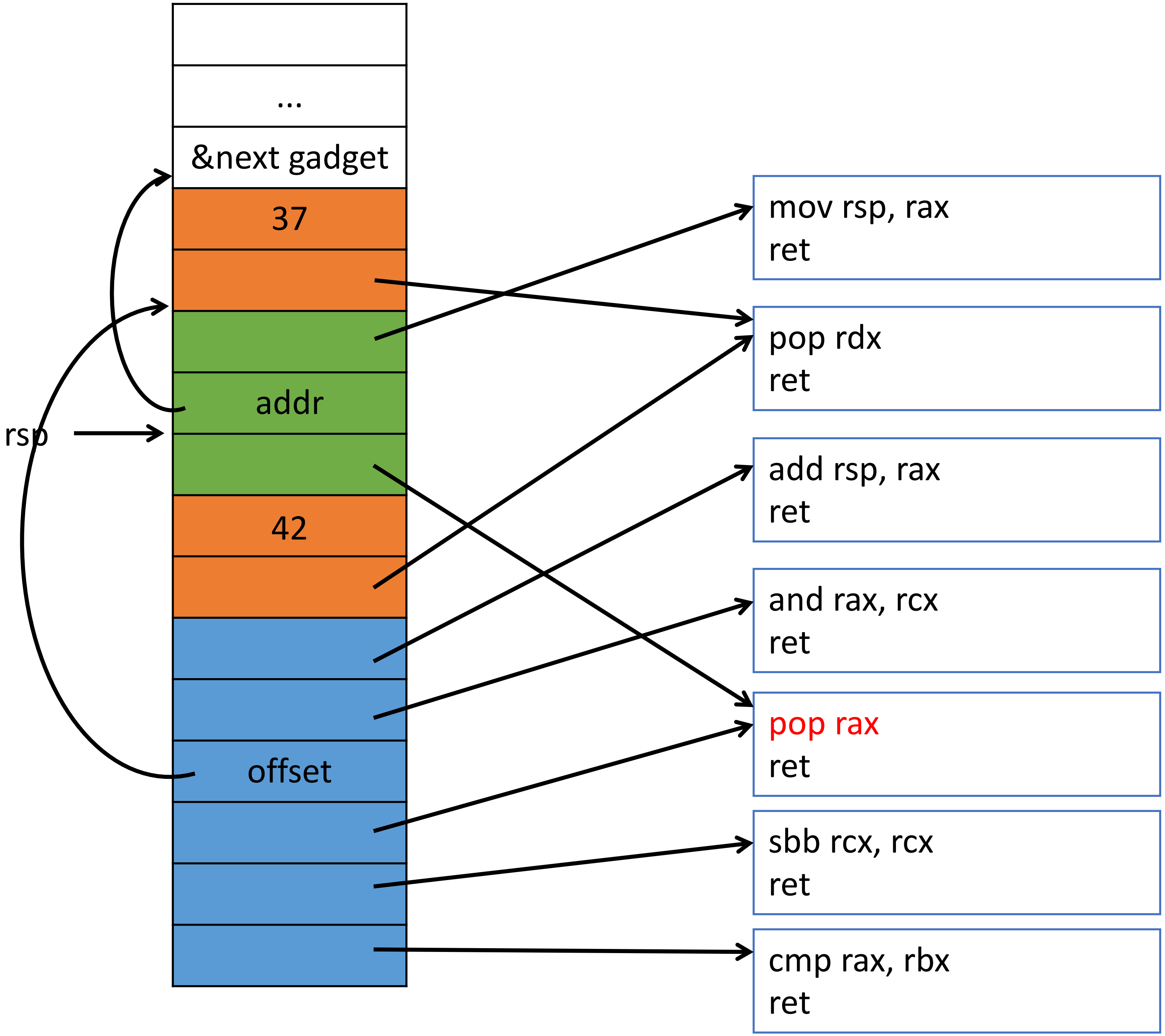
And again!

Register	Value
rax	0
rbx	20
rcx	0
rdx	42



And again!

Register	Value
rax	0
rbx	20
rcx	0
rdx	42



...

&next gadget

37

addr

rsp

42

offset

mov rsp, rax
ret

pop rdx
ret

add rsp, rax
ret

and rax, rcx
ret

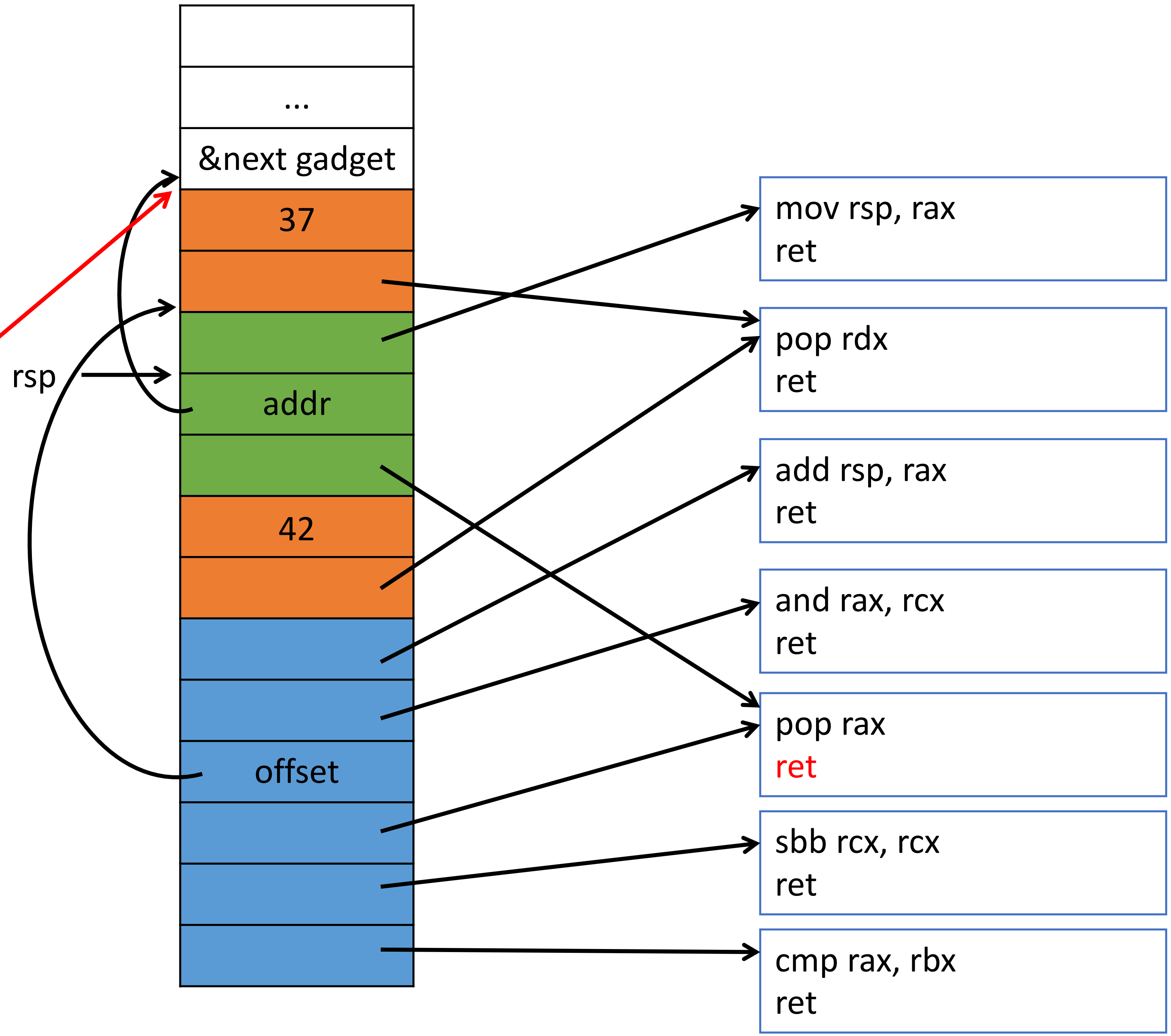
pop rax
ret

sbb rcx, rcx
ret

cmp rax, rbx
ret

And again!

Register	Value
rax	addr
rbx	20
rcx	0
rdx	42



…

&next gadget

37

rsp

addr

42

offset

mov rsp, rax
ret

pop rdx
ret

add rsp, rax
ret

and rax, rcx
ret

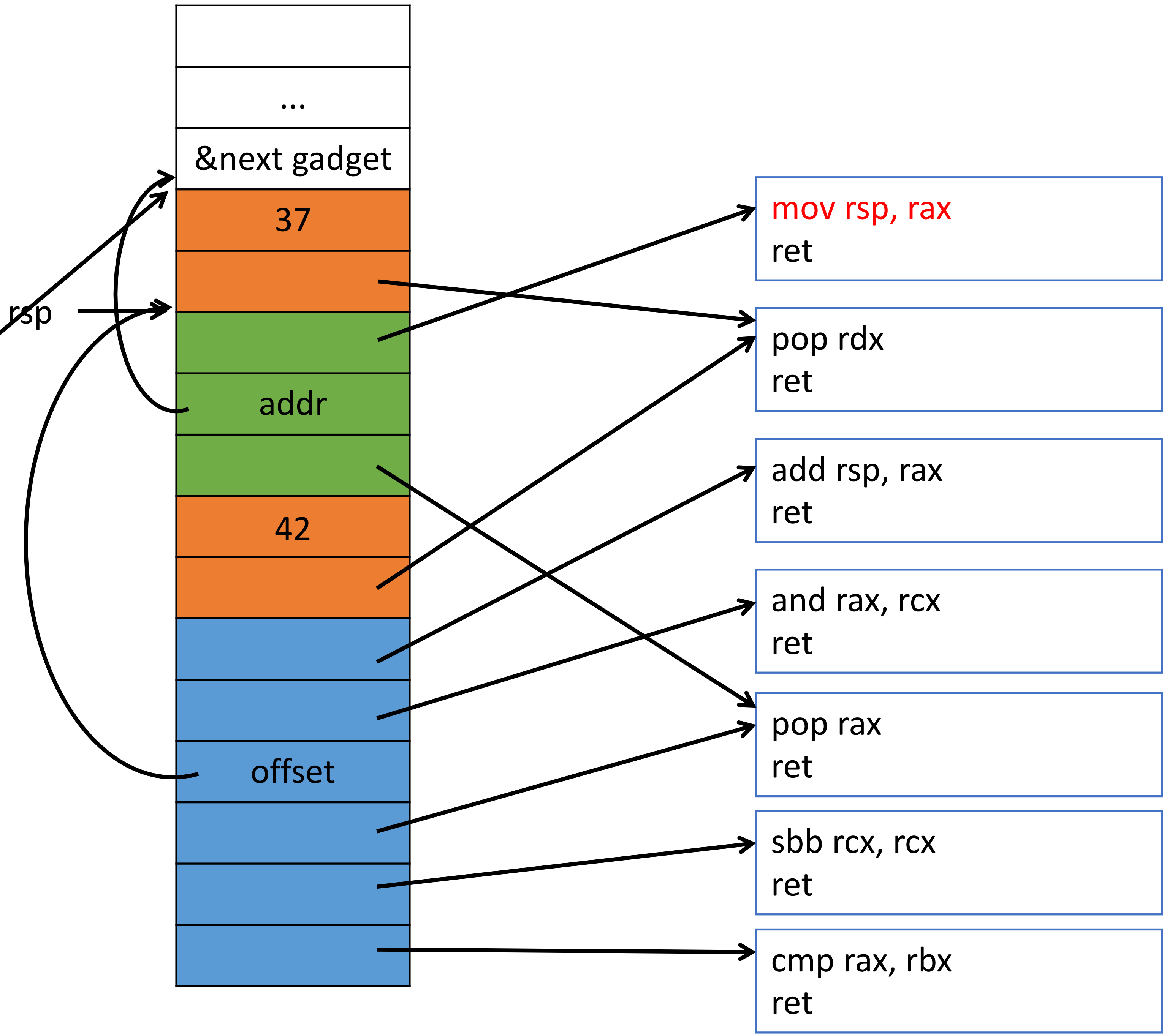
pop rax
ret

sbb rcx, rcx
ret

cmp rax, rbx
ret

And again!

Register	Value
rax	addr
rbx	20
rcx	0
rdx	42



…

&next gadget

37

rsp

addr

42

offset

mov rsp, rax
ret

pop rdx
ret

add rsp, rax
ret

and rax, rcx
ret

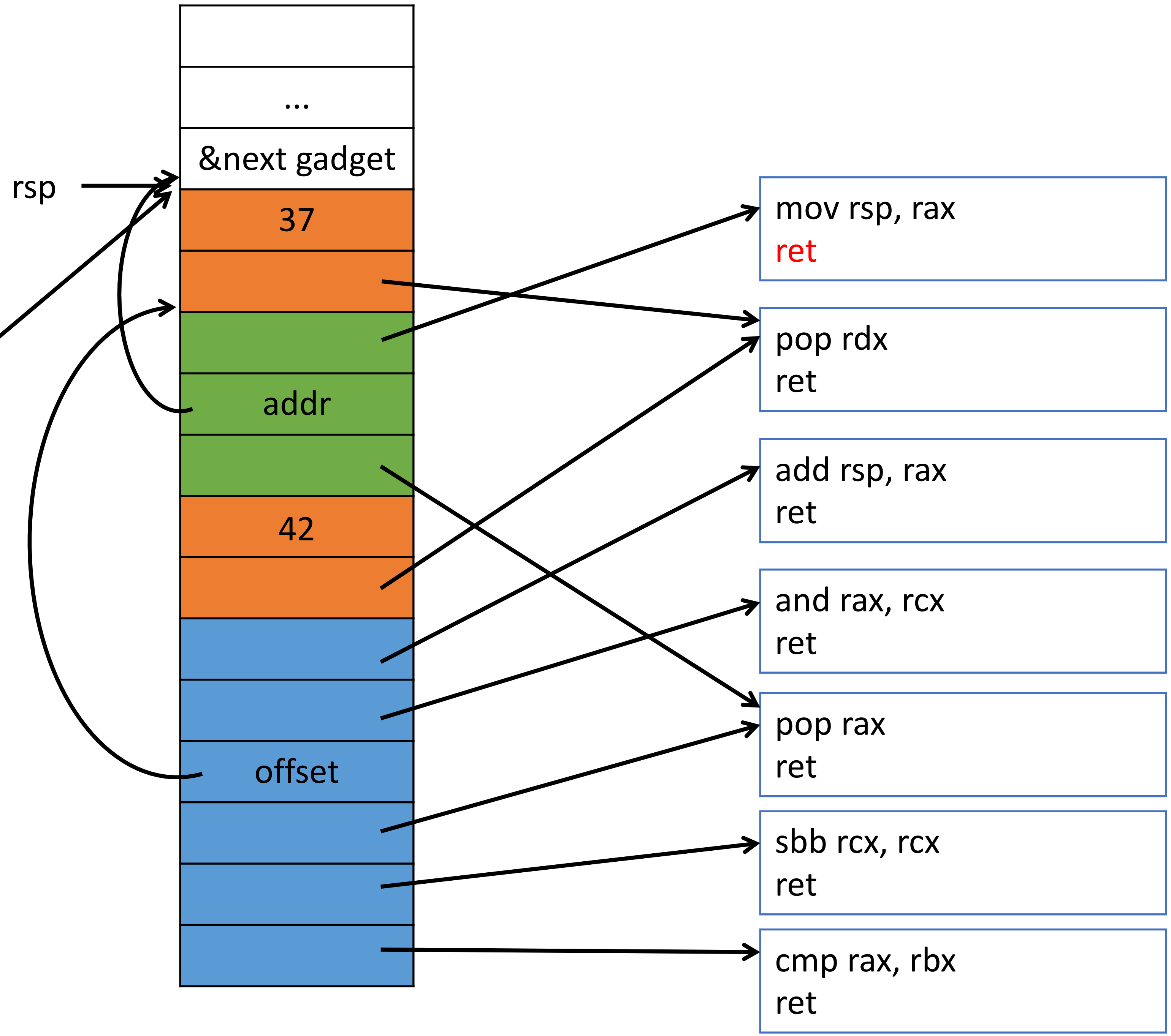
pop rax
ret

sbb rcx, rcx
ret

cmp rax, rbx
ret

And again!

Register	Value
rax	addr
rbx	20
rcx	0
rdx	42



```
mov rsp, rax
ret
```

```
pop rdx
ret
```

```
add rsp, rax
ret
```

```
and rax, rcx
ret
```

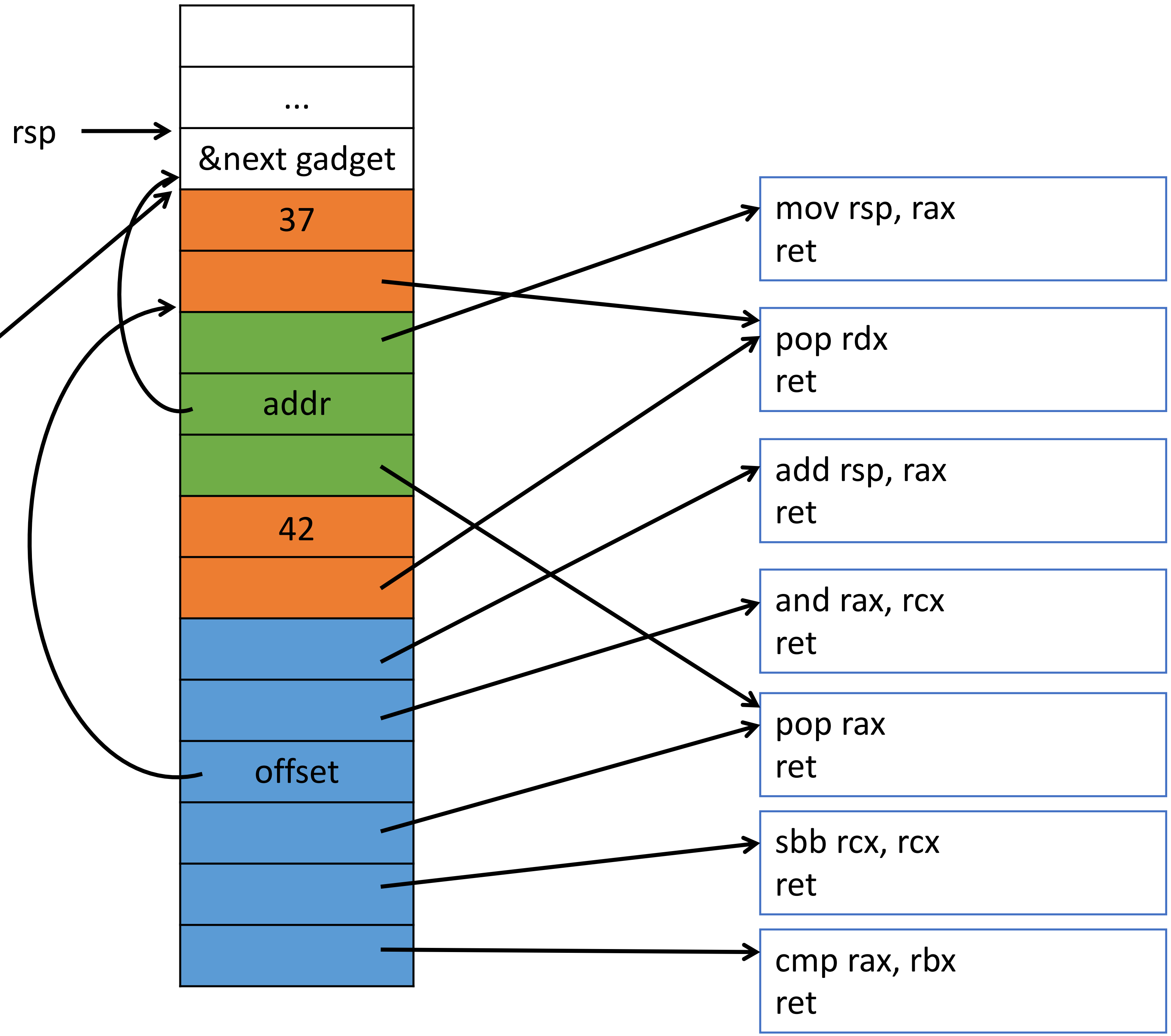
```
pop rax
ret
```

```
sbb rcx, rcx
ret
```

```
cmp rax, rbx
ret
```

And again!

Register	Value
rax	addr
rbx	20
rcx	0
rdx	42



Compare

Register	Value
rax	10
rbx	20
rcx	108
rdx	17

Register	Value
rax	40
rbx	20
rcx	-1
rdx	37

```
if (rax < rbx)
            rdx = 37;
else
            rdx = 42;
```

Register	Value
rax	500
rbx	20
rcx	108
rdx	17

Register	Value
rax	addr
rbx	20
rcx	0
rdx	42

Stack pivot: What if the ROP program is not on the stack?

- We can’t always overflow a stack buffer (e.g., canaries)
- Place our return-oriented program somewhere else like a heap-allocated buffer
- Perform a stack pivot: change rsp to that buffer
    - Requires overwriting the saved return address or a code pointer with the address of a pivot gadget

Some pivoting options

- xchg rax, rsp ; ret exchanges the values in rax and rsp

- push rax ; pop rsp ; ret

- Also useful if you don’t know stack addresses but do know the address of where the return-oriented program is written

More code = more ROP options

- The more code in programs and libraries, the easier it is to construct return-oriented programs
- **If you cannot find the gadget you need directly, you can often use multiple gadgets to construct the same behavior**
- Sometimes you'll need to deal with extra instructions between the one you care about and the return
- Sometimes those instructions have requirements
  - E.g., `mov [rax + 16], rdx ; div [rcx] ; ret` requires you put the address you want minus 16 in rax, rcx needs to be a valid address pointing to a nonzero value, and rax and rdx will both change as a result!

Other gadget endings

- We’ve looked at sequences of instructions ending in ret; not the only option
- `pop rdx ; ret 16` will pop a value into rdx, pop the next word into rip, and then increment the stack pointer by 16
- `mov rax, rbx ; jmp rcx` works just fine as a gadget so long as rcx holds the address of a ret instruction
- You can even construct a whole return-oriented program that never uses a ret, e.g., by finding gadgets ending in pop rax ; jmp rax
- Far return: retf pops rip and cs off the stack

Other architectures

- ROP works on basically every computer architecture
  - x86, x86-64, ARM32, ARM64, RISC-V, MIPS, Sparc, Z80
- Some architectures require less code or are easier than others (in my experience)
  - x86 and ARM32 are easy
  - Z80 required a tiny amount of code (every instruction sequence is valid)
  - x86-64 requires more code
  - Sparc has a weird register window calling convention
  - RISC-V is “annoying” I’ve been told

x86-64 is surprisingly tricky compared to x86

- Basic issue is most of the opcodes are the same for x86-64 and x86
  - `add eax, 0x12345678` is encoded as `05 78 56 34 12` for both
  - `add eax, ebx` is encoded as `01 d8` for both (actually there are two valid encodings of this!)
- Accessing the 64-bit registers requires a REX prefix 40 through 4F
  - `add rax, 0x12345678` is `48 05 78 56 34 12`
  - `add rax, rbx` is `48 01 d8`
- Consequence: it’s harder to find unintended instructions operating on 64-bit registers

Unintended instructions

- Variable-length instructions mean there are not well-defined instruction boundaries
- Consider `add rsi, -1009254072`
  encoding `48 81 c6 48 01 d8 c3`
- Starting 1 byte in gives `add esi, -1009254072`
- However, starting 4 bytes in gives

  ```
  48 01 d8     mov rax, rbx
  c3           ret
  ```