Lec 05: x86_64

CSED415: Computer Security

Spring 2025

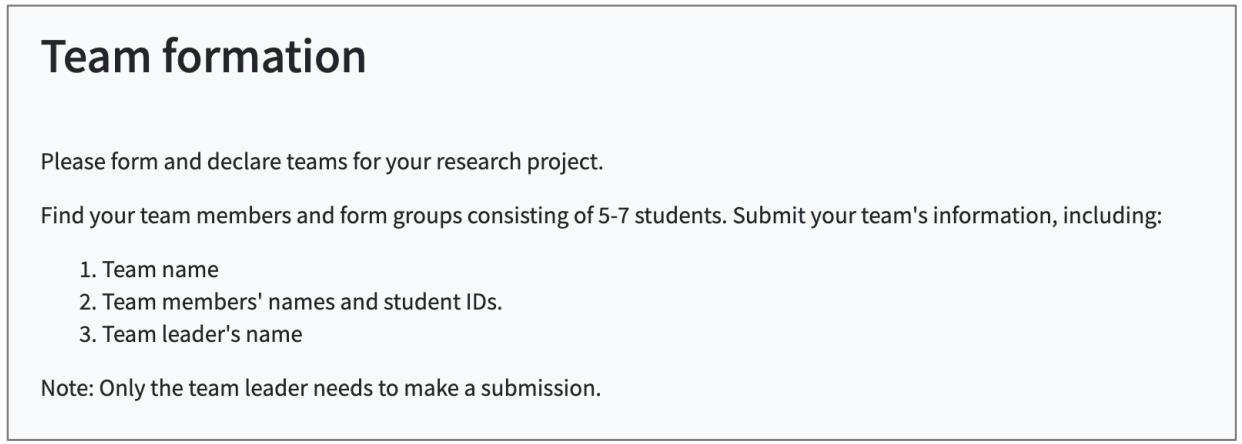
Seulbae Kim

POSTECH

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Administrivia

- Lab 02 has been released (due on March 21)
  - Start early!
  - Attend office hours if you need help!
- Team formation is due this Friday
  - Make a submission on PLMS



Team formation

Please form and declare teams for your research project.

Find your team members and form groups consisting of 5-7 students. Submit your team's information, including:

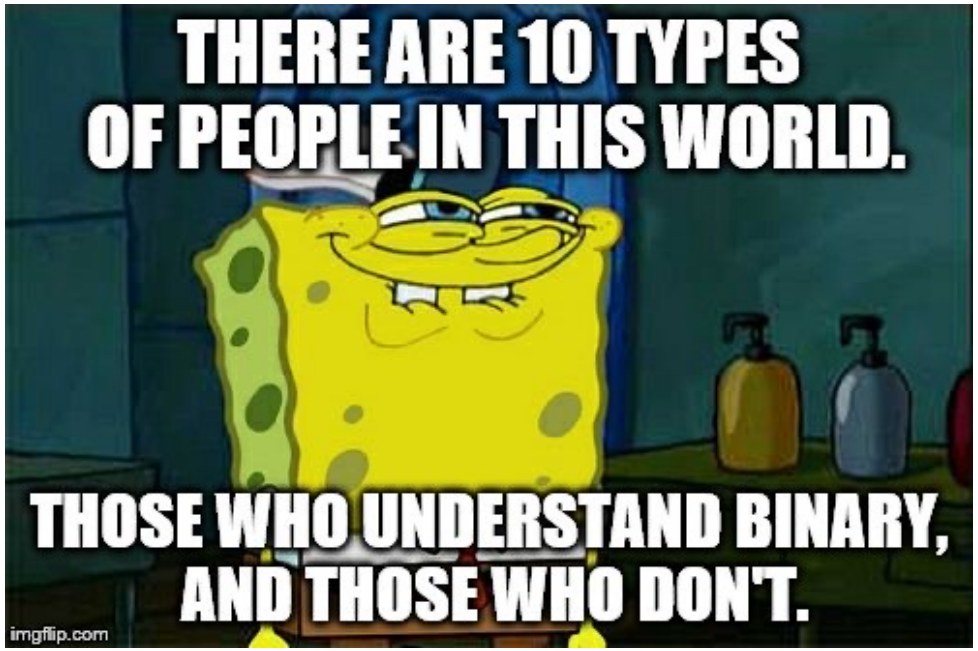
1. Team name
2. Team members' names and student IDs.
3. Team leader's name

Note: Only the team leader needs to make a submission.

Recap

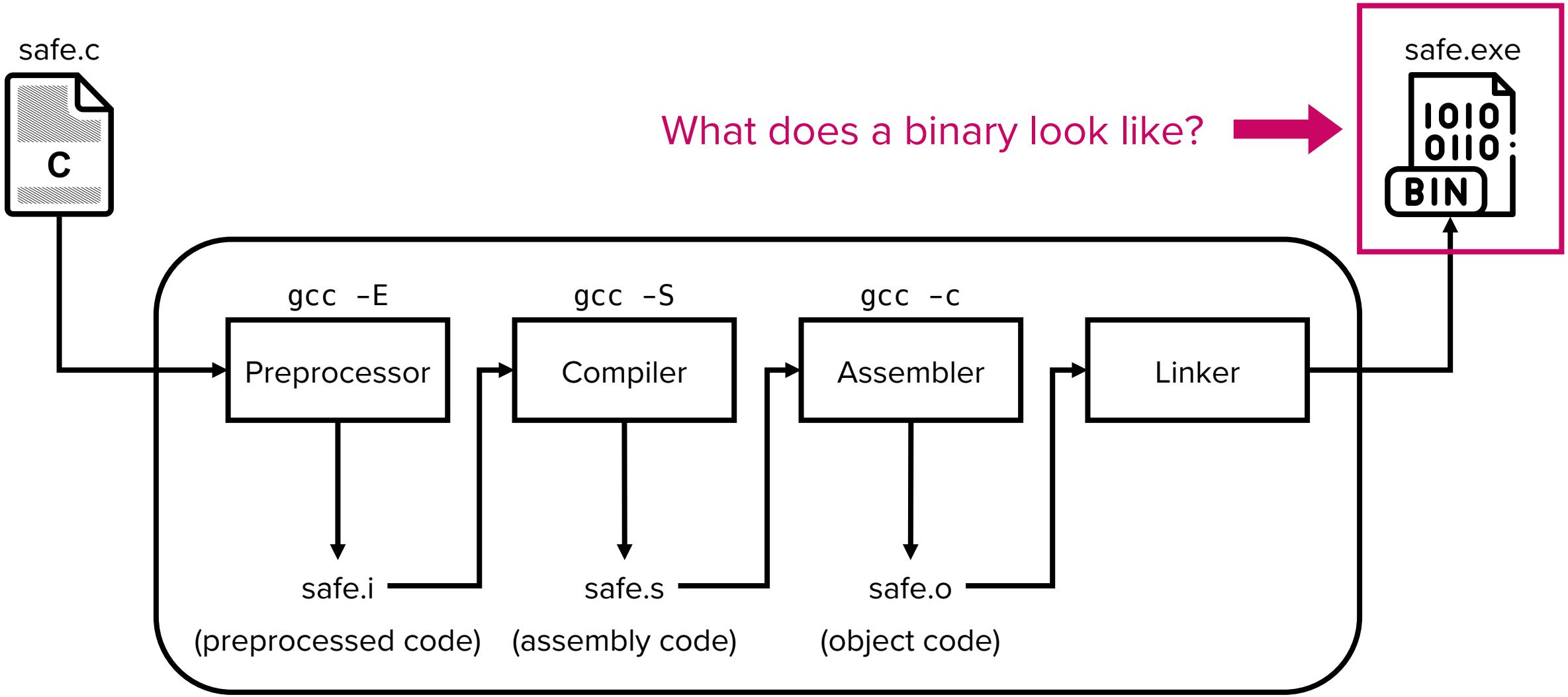
- Secure coding is necessary for building secure systems
- However, code-level mitigation may not be enough
  - We cannot trust anything that we did not create ourselves
    - e.g., The Thompson Compiler produces malicious binaries
- Binary analysis, like it or not, is required for us to analyze exactly what a CPU executes
  - Needed to detect malicious behavior

BIN


THERE ARE 10 TYPES
OF PEOPLE IN THIS WORLD.
THOSE WHO UNDERSTAND BINARY,
AND THOSE WHO DON'T.
imgflip.com

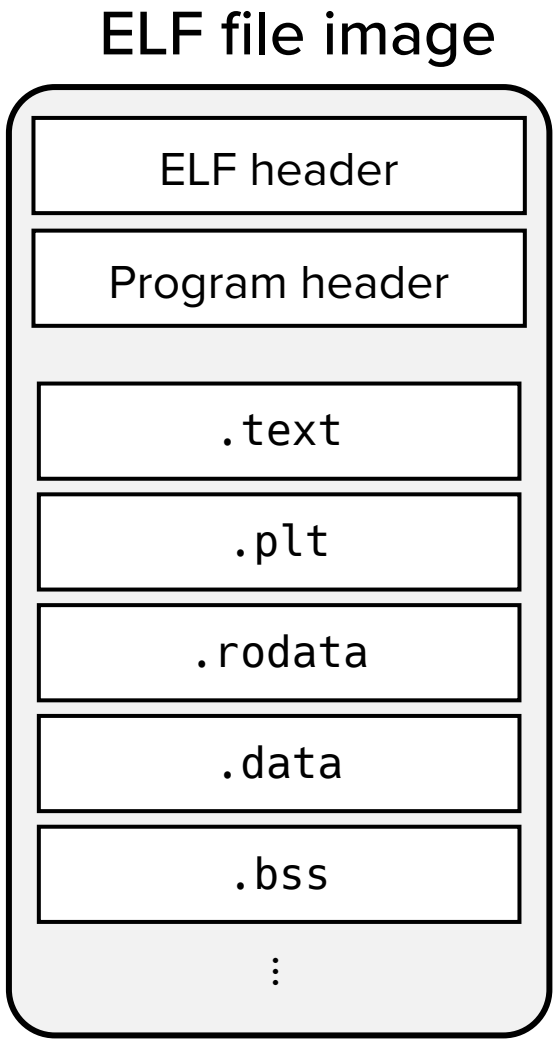
ARY

Recap: Compiler 101



Executable binary (ELF-formatted file)

- ELF (Executable and Linkable Format)
  - Most commonly used binary file format on Linux
    - Executables, object files (.o), and shared libraries (.so)
  - Key components
    - ELF header
    - Program header
    - Segments and sections

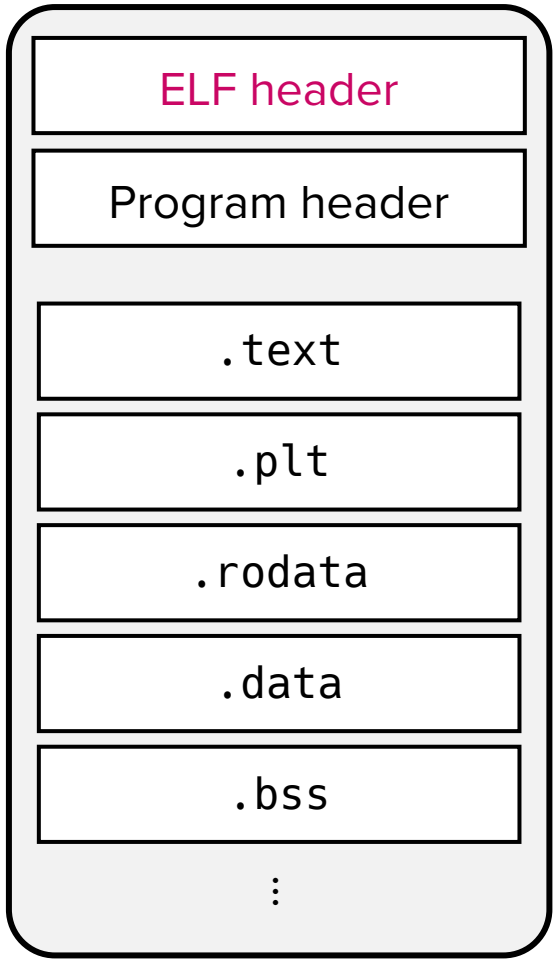


Executable binary

- ELF header
  - Located at the very beginning of an ELF file
  - Contains crucial information about the binary
    - Arch, endianness, file type, entry point address, header size, etc.

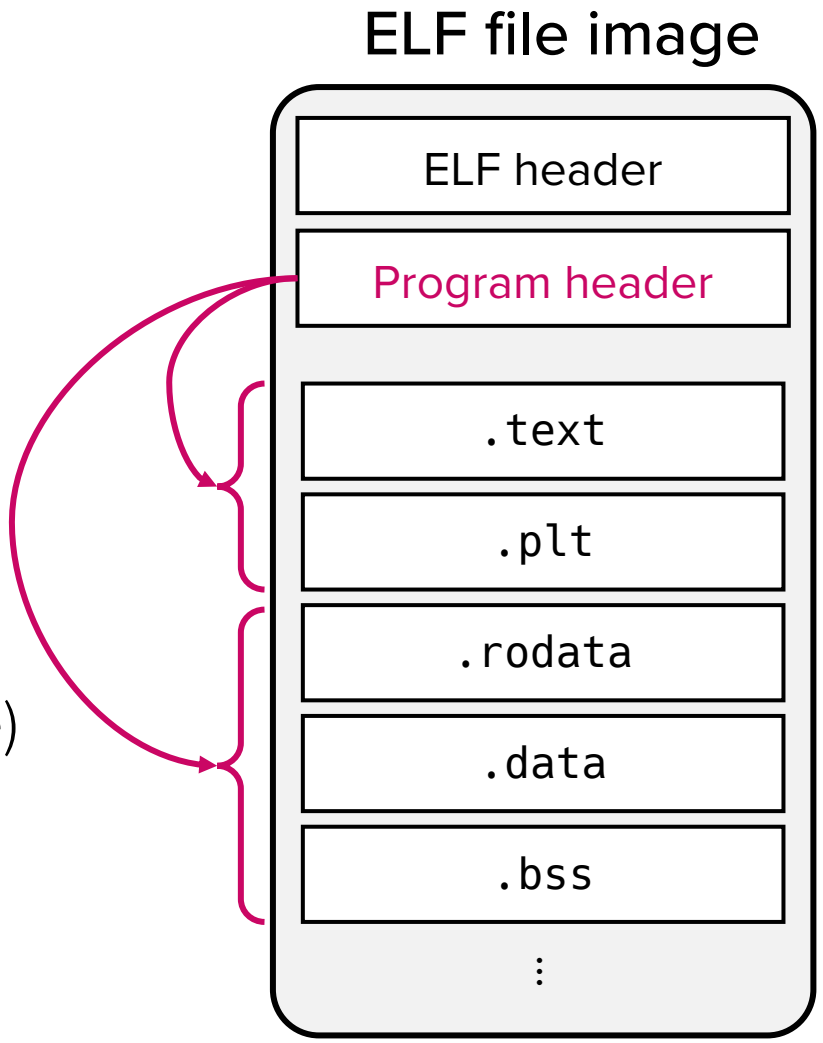
```
csed415-lab01@csed415:~$ readelf -h ./target
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x401210
  Start of program headers:          64 (bytes into file)
  Start of section headers:          15096 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         13
  Size of section headers:           64 (bytes)
  Number of section headers:         31
  Section header string table index: 30
```

ELF file image



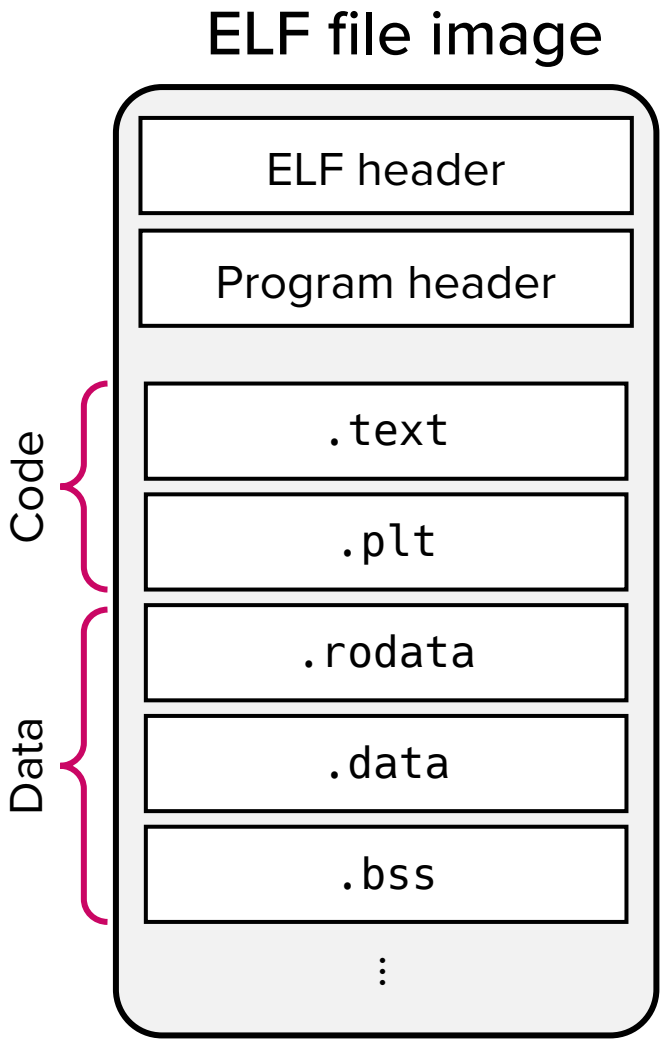
Executable binary

- Program header
  - Describes how segments should be mapped into memory at load time
  - Contains segment information
    - Offset and size of each segment
    - Virtual memory address on which each segment should be loaded
    - Permission of each segment (Read/Write/Execute)



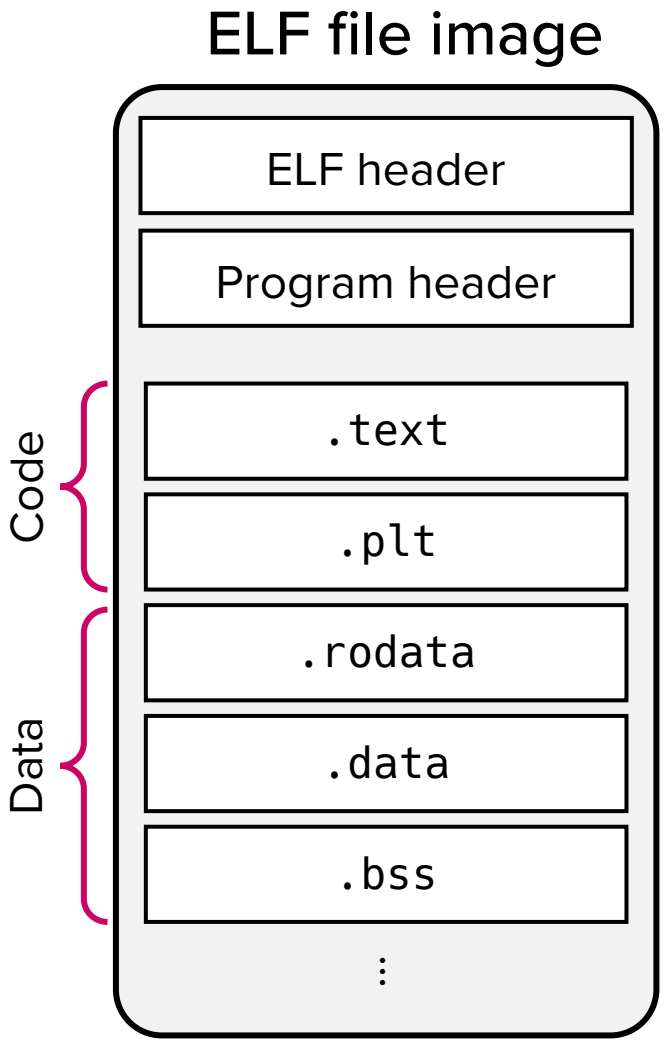
Executable binary

- Segments
  - Logical groups of sections
  - Each segment is loaded onto one or more virtual memory pages at runtime
  - Code-related segment
    - Contains executable instructions (`.text`, `.plt`, ...)
  - Data-related segment
    - Contains non-executable data (`.data`, `.rodata`, ...)



Executable binary

- Sections: Smallest unit of an object
  - `.text` (executable instructions)
  - `.data` (initialized data)
  - `.bss` (uninitialized data)
  - `.rodata` (read-only data)
  - `.plt`, `.got` (dynamic linking stubs/tables)
  - etc.



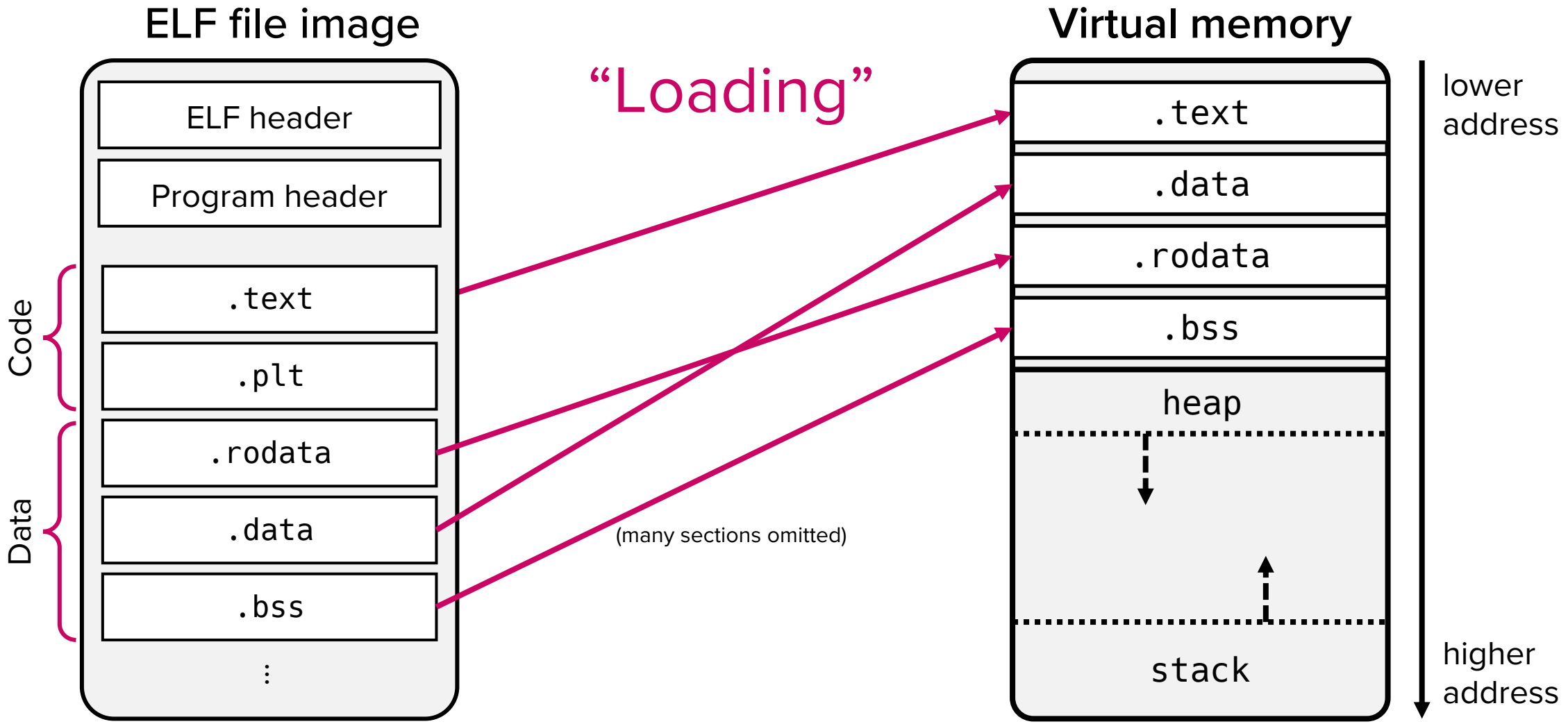
Executable binary

- Visualizing ELF via readelf

```
csed415-lab01@csed415:~$ readelf -a ./target
...
Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000400040 0x0000000000400040
                 0x00000000000002d8 0x00000000000002d8  R      0x8
  INTERP         0x0000000000000318 0x0000000000400318 0x0000000000400318
                 0x000000000000001c 0x000000000000001c  R      0x1
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x0000000000000918 0x0000000000000918  R      0x1000
  LOAD           0x0000000000001000 0x0000000000401000 0x0000000000401000
                 0x0000000000000765 0x0000000000000765  R E    0x1000
...
  Segment Sections...
   00
   01     .interp
   02     .interp .note.gnu.property .note.gnu.build-id .note.ABI-tag .gnu.hash .dynsym .dynstr
.gnu.version .gnu.version_r .rela.dyn .rela.plt
   03     .init .plt .plt.sec .text .fini  ← Sections in segment #3 (Note: .text is where your code lives)
```

"Segment #3 is readable and executable. It should be loaded at 0x401000"

Loader turns a binary image into a process



ELF loader workflow

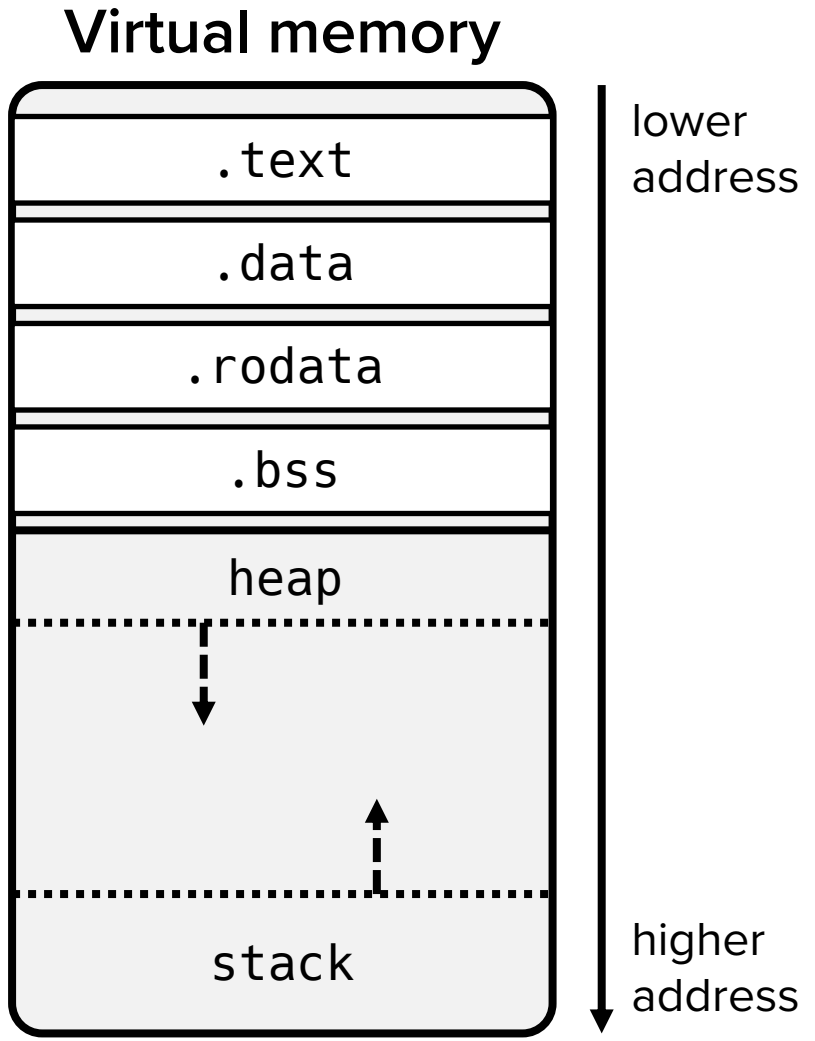
1. Read the ELF Header, which is always located at the very beginning of an ELF file
2. Read the program headers. These specify where in the file the program segments are located, and where they need to be loaded into memory
3. Parse the program headers to determine the number of program segments that must be loaded

ELF loader workflow

4. Load each of the loadable segments
   a. Allocate virtual memory for each segment, at the address specified by the p_vaddr member in the program header
   b. Copy the segment data from the ELF file offset specified by p_offset to the virtual memory address specified by p_vaddr
5. Read the executable's entry point from the ELF header
6. Jump to the executable's entry point in the newly loaded memory

A process in memory

- ELF sections + heap + stack
  - Heap
    - Dynamically allocated memory (e.g., using malloc)
    - Grows towards higher (larger) addresses
  - Stack
    - Contains runtime call stack
      - More on this later!
    - Grows towards lower (smaller) addresses



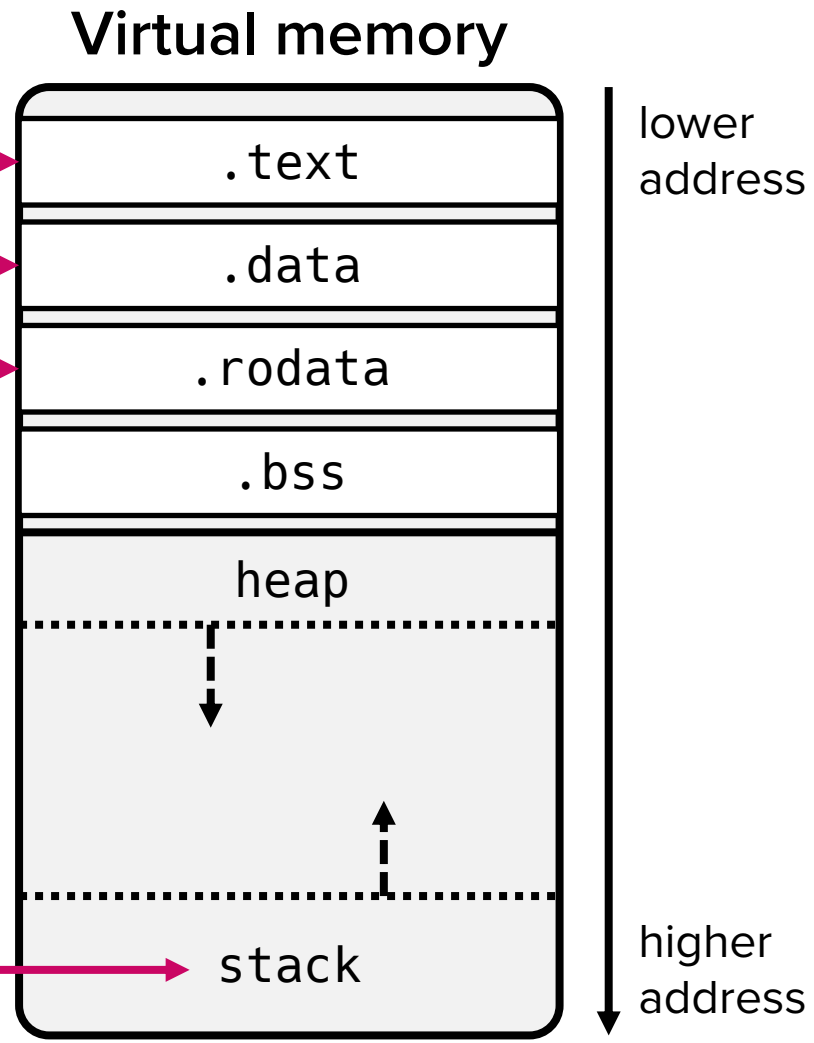
Example: Virtual memory of Lab 01's target

POSTECH

```
$ ssh csed415-lab01@141.223.181.16 -p 7022
$ gdb ./target
pwndbg> break *main
pwndbg> r
pwndbg> vmmap
```

VM addresses match the program header

```
LEGEND: STACK | HEAP | CODE | DATA | WX | RODATA
              Start                End Perm     Size Offset File
            0x400000           0x401000 r--p     1000      0 /home/csed415-lab01/target
            0x401000           0x402000 r-xp     1000   1000 /home/csed415-lab01/target
            0x402000           0x403000 r--p     1000   2000 /home/csed415-lab01/target
            0x403000           0x404000 r--p     1000   2000 /home/csed415-lab01/target
            0x404000           0x405000 rw-p     1000   3000 /home/csed415-lab01/target
      0x7f166e29c000     0x7f166e29f000 rw-p     3000      0 [anon_7f166e29c]
      0x7f166e29f000     0x7f166e2c7000 r--p    28000      0 /lib/x86_64-linux-gnu/libc.so.6
      0x7f166e2c7000     0x7f166e45c000 r-xp   195000  28000 /lib/x86_64-linux-gnu/libc.so.6
      0x7f166e45c000     0x7f166e4b4000 r--p    58000 1bd000 /lib/x86_64-linux-gnu/libc.so.6
      0x7f166e4b4000     0x7f166e4b5000 ---p     1000 215000 /lib/x86_64-linux-gnu/libc.so.6
      0x7f166e4b5000     0x7f166e4b9000 r--p     4000 215000 /lib/x86_64-linux-gnu/libc.so.6
      0x7f166e4b9000     0x7f166e4bb000 rw-p     2000 219000 /lib/x86_64-linux-gnu/libc.so.6
      0x7f166e4bb000     0x7f166e4c8000 rw-p     d000      0 [anon_7f166e4bb]
      0x7f166e4d3000     0x7f166e4d5000 rw-p     2000      0 [anon_7f166e4d3]
      0x7f166e4d5000     0x7f166e4d7000 r--p     2000      0 /lib/x86_64-linux-gnu/ld-linux-x
      0x7f166e4d7000     0x7f166e501000 r-xp    2a000   2000 /lib/x86_64-linux-gnu/ld-linux-x
      0x7f166e501000     0x7f166e50c000 r--p     b000  2c000 /lib/x86_64-linux-gnu/ld-linux-x
      0x7f166e50d000     0x7f166e50f000 r--p     2000  37000 /lib/x86_64-linux-gnu/ld-linux-x
      0x7f166e50f000     0x7f166e511000 rw-p     2000  39000 /lib/x86_64-linux-gnu/ld-linux-x
      0x7ffd99555000     0x7ffd99576000 rw-p    21000      0 [stack]
      0x7ffd995bb000     0x7ffd995bf000 r--p     4000      0 [vvar]
      0x7ffd995bf000     0x7ffd995c1000 r-xp     2000      0 [vdso]
0xffffffffff600000 0xffffffffff601000 --xp     1000      0 [vsyscall]
```

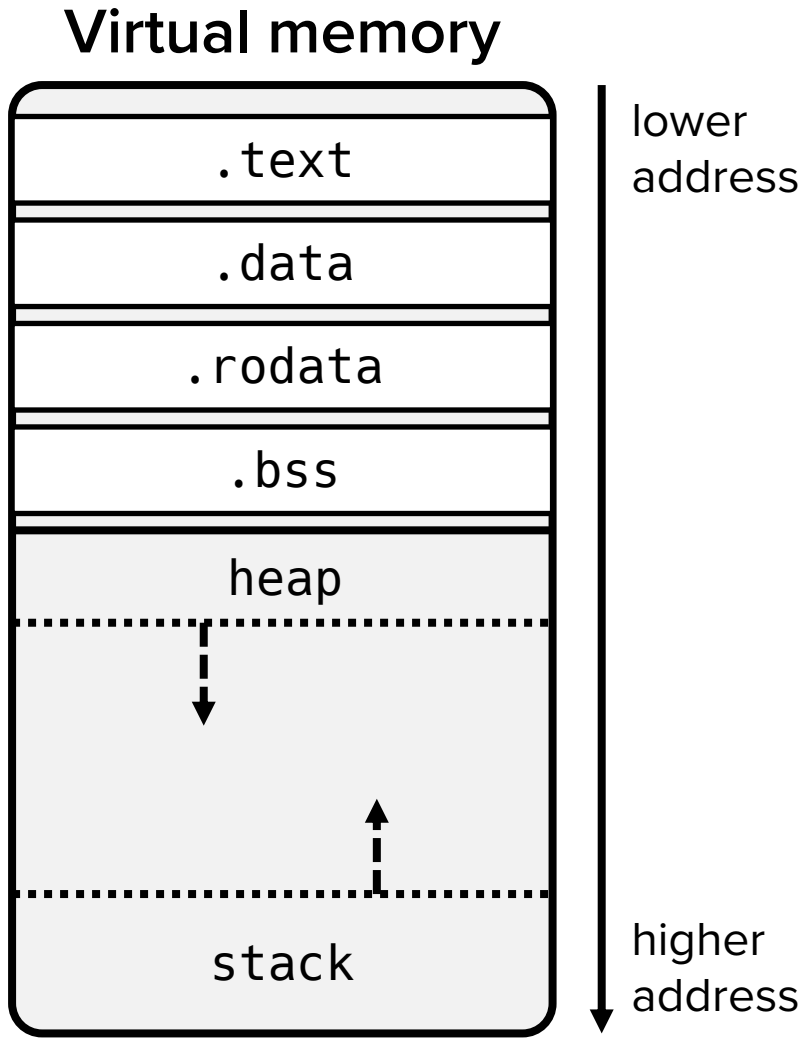


Example: Virtual memory of Lab 01's target

POSTECH

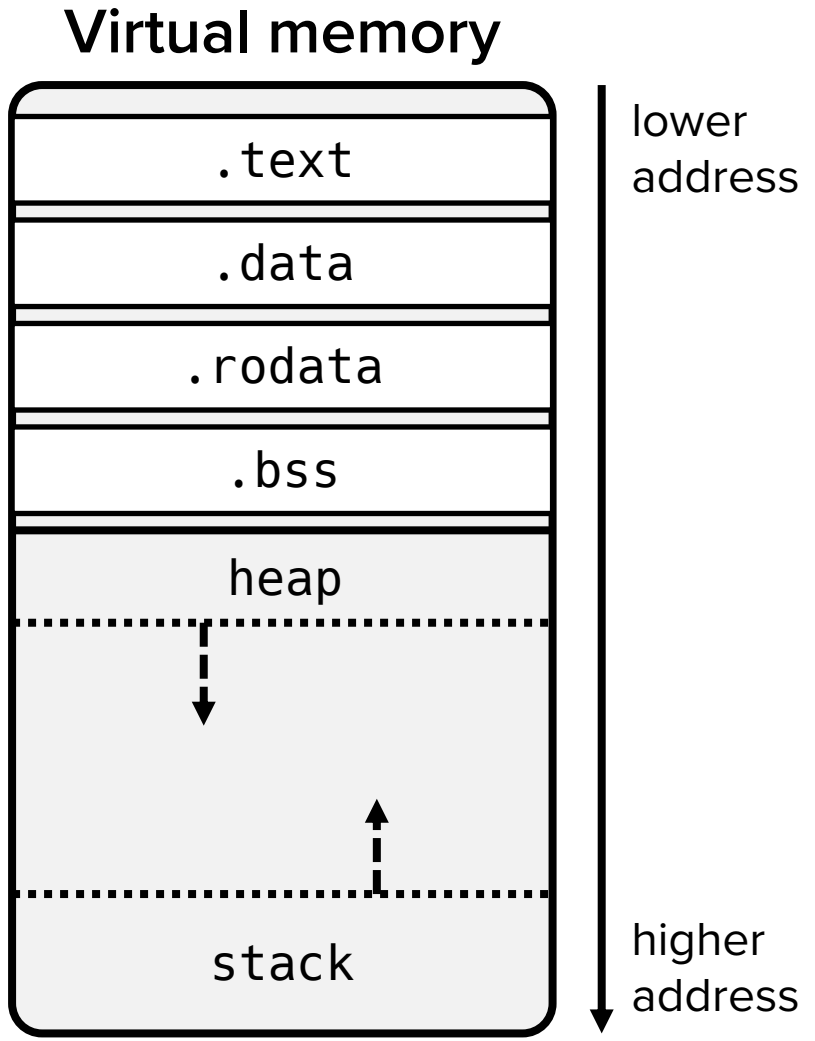
```
pwndbg> elfsections // check section info
pwndbg> x/20i 0x401000 // examine .text instructions
pwndbg> x/10s 0x402000 // examine .rodata strings
pwndbg> x/100wx 0x404000 // examine .data data
```

```
LEGEND: STACK | HEAP | CODE | DATA | WX | RODATA
              Start                End Perm     Size Offset File
            0x400000           0x401000 r--p     1000      0 /home/csed415-lab01/target
            0x401000           0x402000 r-xp     1000   1000 /home/csed415-lab01/target
            0x402000           0x403000 r--p     1000   2000 /home/csed415-lab01/target
            0x403000           0x404000 r--p     1000   2000 /home/csed415-lab01/target
            0x404000           0x405000 rw-p     1000   3000 /home/csed415-lab01/target
      0x7f166e29c000     0x7f166e29f000 rw-p     3000      0 [anon_7f166e29c]
      0x7f166e29f000     0x7f166e2c7000 r--p    28000      0 /lib/x86_64-linux-gnu/libc.so.6
      0x7f166e2c7000     0x7f166e45c000 r-xp   195000  28000 /lib/x86_64-linux-gnu/libc.so.6
      0x7f166e45c000     0x7f166e4b4000 r--p    58000 1bd000 /lib/x86_64-linux-gnu/libc.so.6
      0x7f166e4b4000     0x7f166e4b5000 ---p     1000 215000 /lib/x86_64-linux-gnu/libc.so.6
      0x7f166e4b5000     0x7f166e4b9000 r--p     4000 215000 /lib/x86_64-linux-gnu/libc.so.6
      0x7f166e4b9000     0x7f166e4bb000 rw-p     2000 219000 /lib/x86_64-linux-gnu/libc.so.6
      0x7f166e4bb000     0x7f166e4c8000 rw-p     d000      0 [anon_7f166e4bb]
      0x7f166e4d3000     0x7f166e4d5000 rw-p     2000      0 [anon_7f166e4d3]
      0x7f166e4d5000     0x7f166e4d7000 r--p     2000      0 /lib/x86_64-linux-gnu/ld-linux-x
      0x7f166e4d7000     0x7f166e501000 r-xp    2a000   2000 /lib/x86_64-linux-gnu/ld-linux-x
      0x7f166e501000     0x7f166e50c000 r--p     b000  2c000 /lib/x86_64-linux-gnu/ld-linux-x
      0x7f166e50d000     0x7f166e50f000 r--p     2000  37000 /lib/x86_64-linux-gnu/ld-linux-x
      0x7f166e50f000     0x7f166e511000 rw-p     2000  39000 /lib/x86_64-linux-gnu/ld-linux-x
      0x7ffd99555000     0x7ffd99576000 rw-p    21000      0 [stack]
      0x7ffd995bb000     0x7ffd995bf000 r--p     4000      0 [vvar]
      0x7ffd995bf000     0x7ffd995c1000 r-xp     2000      0 [vdso]
0xffffffffff600000 0xffffffffff601000 --xp     1000      0 [vsyscall]
```



A process contains instructions and data

- Q1) Which component of a computer executes instructions? → CPU
- Q2) How does that component keep track of which instruction to execute next?
  → Using instruction pointer (special register)
- Q3) How does that component keep track of program execution states?
  → Using general purpose registers



x86_64

x86 and x86_64 architectures

- Our focus: x86 and x86_64 architecture
  - The most widely used architecture
    - Intel – Core i3, i5, i7, i9 families, Xeon server processors
    - AMD – Ryzen processors, EPYC server processors
  - Held 90% of the PC market and 85% of the server market (2022, Mercury Research)

x86 and x86_64 architectures

- x86 (Developed by Intel, 1985)
  - 32-bit architecture – handles 2^32 = 4 GB of memory addresses
    - Registers are 32 bits (except for segment registers)
- x86_64 (Developed by AMD, 2003)
  - Extends x86 to 64-bit registers
    - Also called x64, AMD64, or Intel 64
  - 64-bit architecture – handles 2^64 of memory addresses
    - Registers are 64 bits (except for segment registers)
  - Backward compatible: Can run both 32-bit and 64-bit programs

History of Intel/AMD processors

1978 — 8086, 80286
16-bit processor, used IP, SP, BP, ... registers

1985 — 80386, 80486, ...
x86
32-bit processor, used EIP, ESP, EBP, ... registers
(Extended)

2003 — Opteron, Prescott
2006 — Core 2, ...
x64
64-bit processor, used RIP, RSP, RBP, ... registers
(Register)

Central Processing Unit (CPU)

- CPU's core capability: Carry out a fixed number of operations
- Terminology
  - Instruction set: The set of operations that a CPU is designed to do
    - Instruction Set Architecture (ISA): Name for the set of operations
    - We will focus on x86 and x64 in this course
  - Instruction: One operation that a CPU can do
    - e.g., add two numeric values
  - Assembly language: A human-readable representation of instructions
    - e.g., `add rsp,4` → Add 4 to the current value in `rsp` and store the result in `rsp`

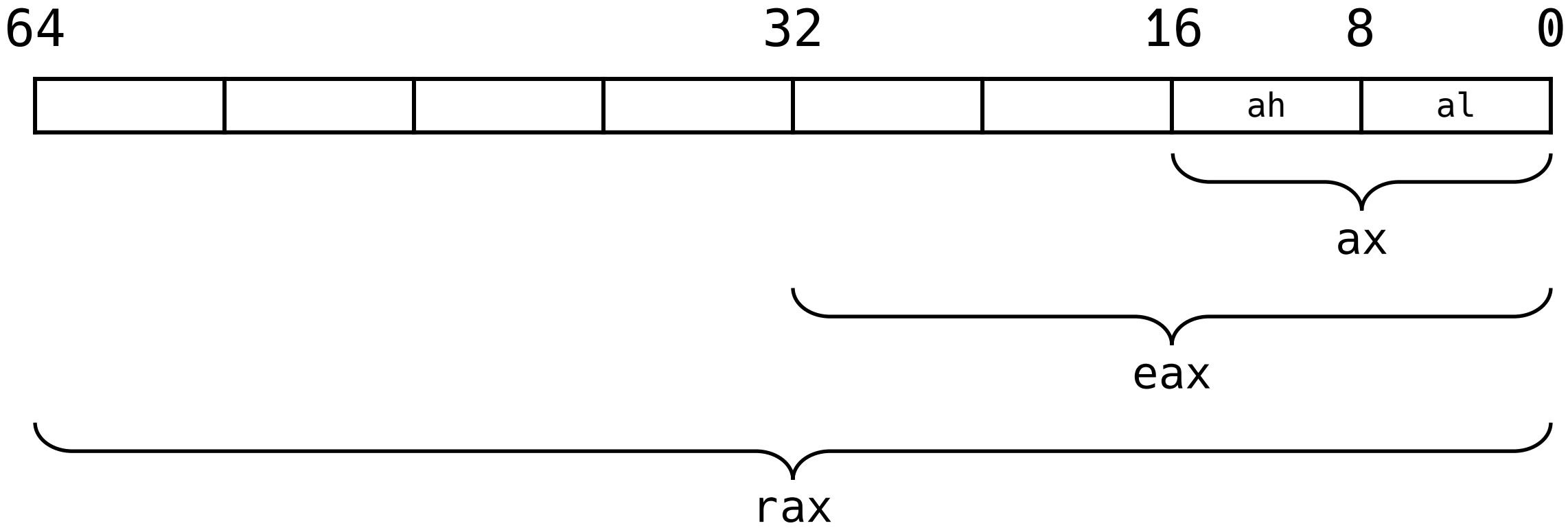
Types of x64 registers

- General purpose registers (GPRs) and their purposes
  - `rax`: Accumulator; often used for return values of function calls
  - `rbx`: Base register
  - `rcx`: Counter register (for loops and shifts)
  - `rdx`: Data register
  - `rsi`: Source index (for string/array operations)
  - `rdi`: Destination index (for string/array operations)
  - `rbp`: Stack base pointer
  - `rsp`: Stack top pointer
  - Additional: `r8, r9, ..., r15`

Types of x64 registers

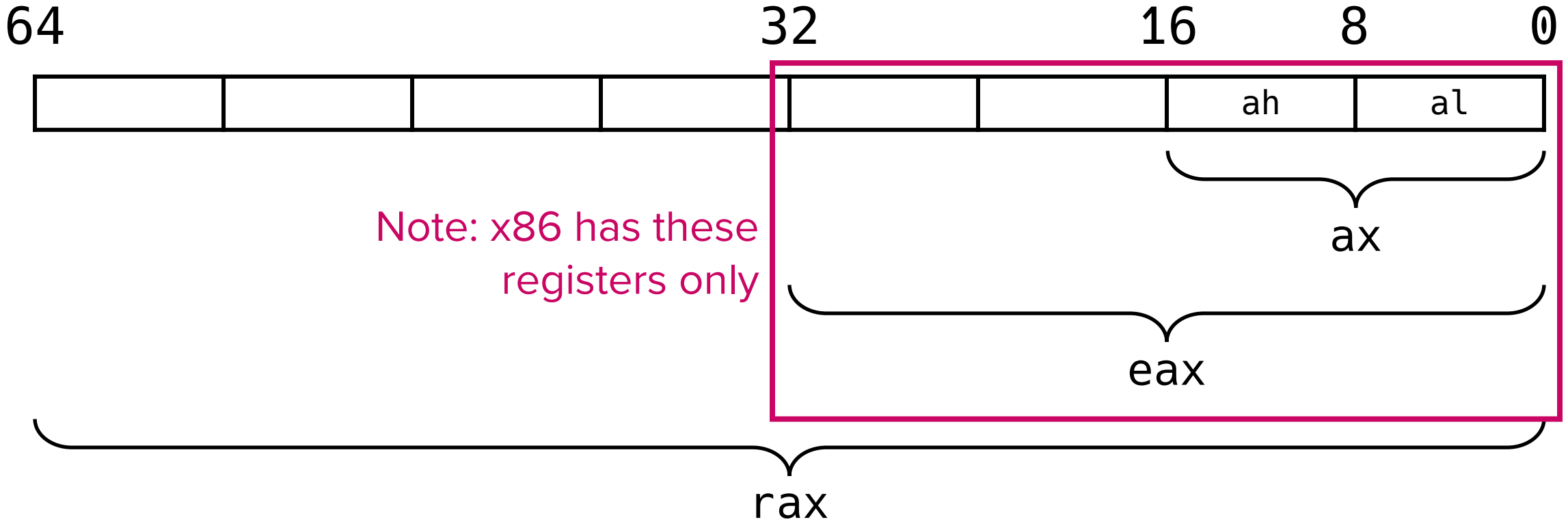
- Instruction pointer
  - `rip`: Holds the memory address of the next instruction to execute
    - Usually points to the `.text` section of a process where code exists
- Segment registers (Legacy)
  - `cs, ss, ds, es, fs, gs`
  - Used for memory addressing (legacy feature, not used in x64)
- Flag register
  - `rflags`: Stores various flags (carry flag, sign flag, ...)

Naming convention for x64 (and x86) registers



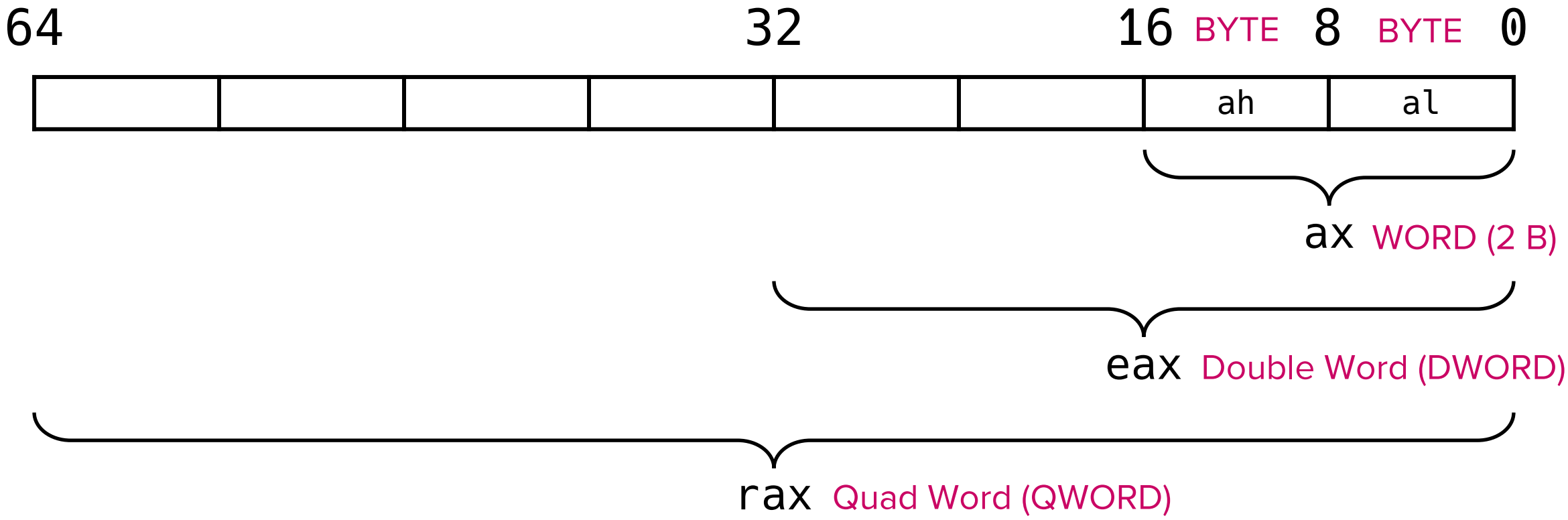
(Same applies to other general purpose registers and `rip`)

Naming convention for x64 (and x86) registers



(Same applies to other general purpose registers and `rip`)

Size convention: BYTE, WORD, DWORD, QWORD



Intel x86_64 Assembly

Assembly language basics

- Assembly: A human-readable representation of instructions
- Basic format:
  - An instruction consists of an opcode and operands

Instruction: `sub rsp, 16`

opcode operands

  - **Opcode** specifies the operation to be performed
  - **Operands** specify the data for the operation

Instruction operands

- Each instruction can have a specific number of operands
  - Intel Instructions can have 0, 1, or 2 operands

No operand:

```
ret
```

1 operand:

```
inc  rax
```

operand 1

2 operands:

```
mov  rax, rbx
```

operand 1

operand 2

Basic semantics

- Operand 1 (usually) stores the result

```
mov   rax, rbx     // rax = rbx

sub   rsp, 0xc     // rsp = rsp - 0xc

inc   rax          // rax = rax + 1
```

Operand types

- Operand can be a register, memory, or an immediate value

```
mov   rax, [rbx]
```

register memory pointed to by rbx

```
sub   rsp, 0xc
```

immediate value

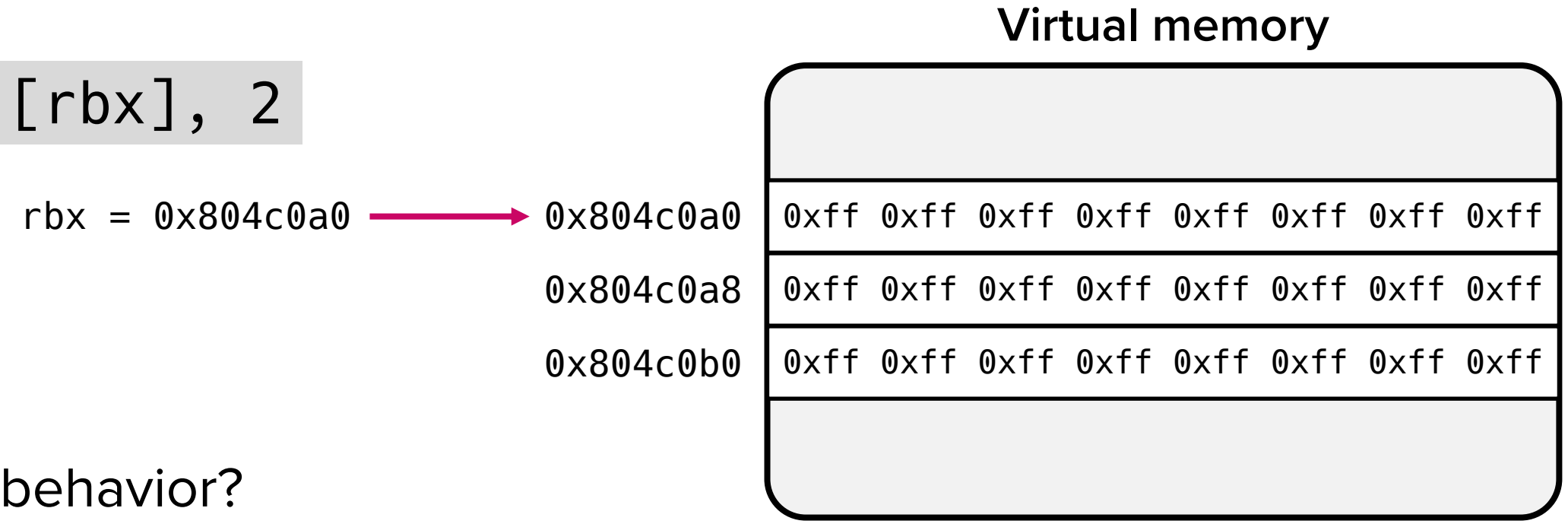
```
mov   al, BYTE ptr [rcx]
```

??? memory pointed to by rcx

Size directives – motivation

POSTECH

```
mov  [rbx], 2
```



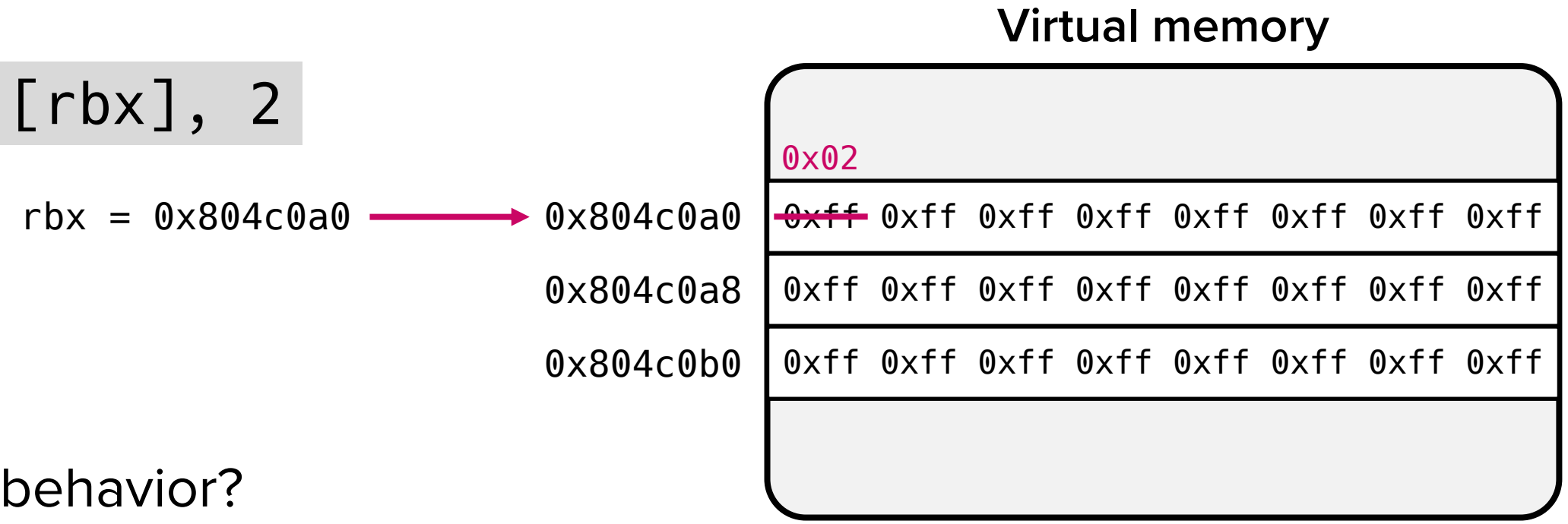
Q) Correct behavior?

1. Overwrite the byte at **`0x804c0a0`** as 2 because the value is byte-sized
2. Overwrite the 8 bytes (64 bits) starting from **`0x804c0a0`** as 2 because the register (**`rbx`**) is qword-sized

Size directives – motivation

POSTECH

```
mov  [rbx], 2
```



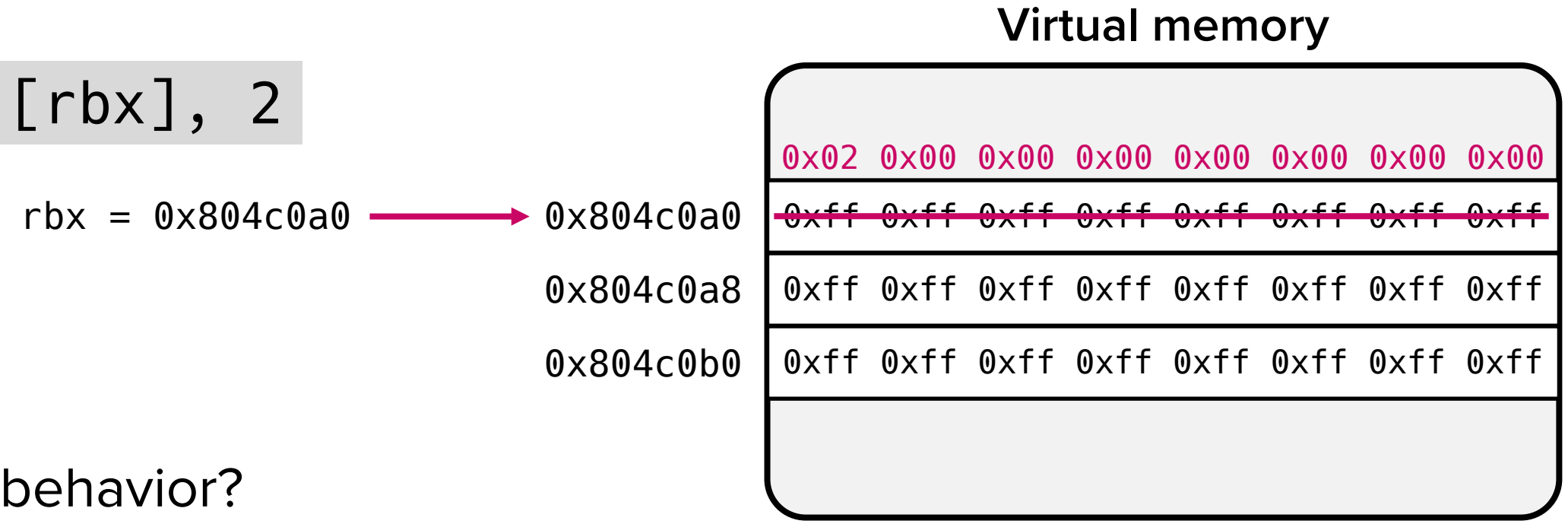
Q) Correct behavior?

1. Overwrite the byte at **`0x804c0a0`** as 2 because the value is byte-sized
2. Overwrite the 8 bytes (64 bits) starting from **`0x804c0a0`** as 2 because the register (**`rbx`**) is qword-sized

Size directives – motivation

POSTECH

```
mov  [rbx], 2
```



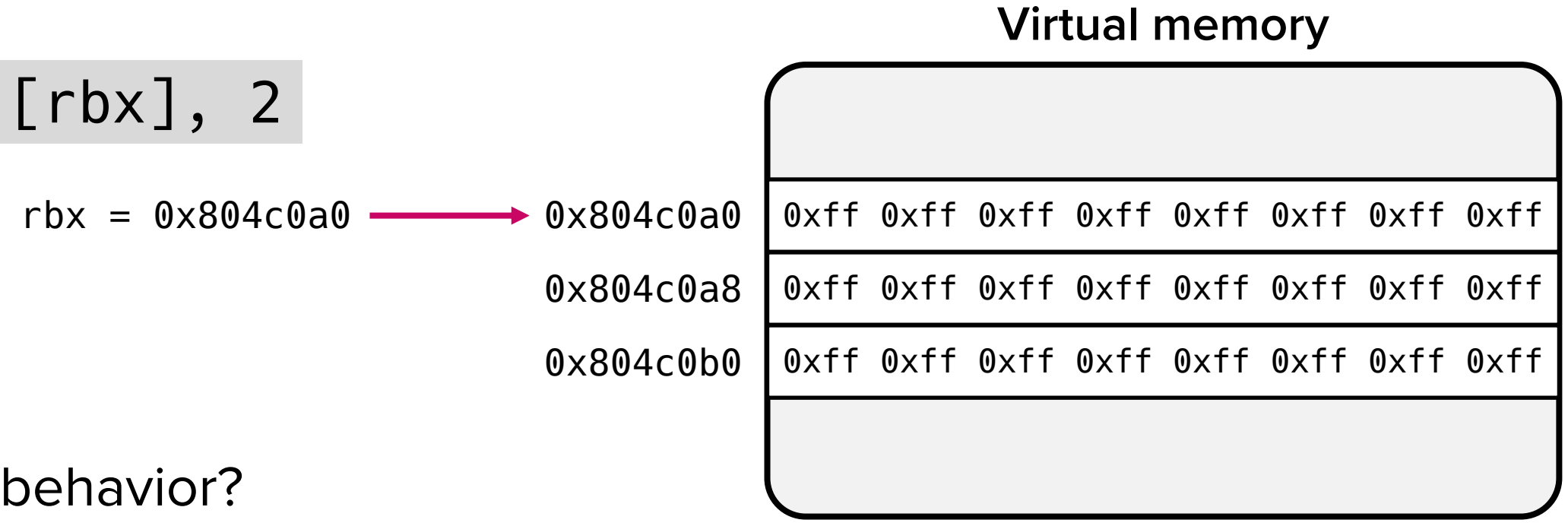
Q) Correct behavior?

1. Overwrite the byte at **`0x804c0a0`** as 2 because the value is byte-sized
2. Overwrite the 8 bytes (64 bits) starting from **`0x804c0a0`** as 2 because the register (**`rbx`**) is qword-sized

Size directives – motivation

POSTECH

```
mov  [rbx], 2
```



Q) Correct behavior?

1. Overwrite the byte at **`0x804c0a0`** as 2 because the value is byte-sized
2. Overwrite the 8 bytes (64 bits) starting from **`0x804c0a0`** as 2 because the register (**`rbx`**) is qword-sized

A) Both are incorrect. `mov [rbx], 2` triggers an error due to ambiguity.

Size directives: SIZE PTR

POSTECH

```
mov   [rbx], 2
```
ERROR

```
mov   BYTE PTR [rbx], 2
```

or

```
mov   WORD PTR [rbx], 2
```

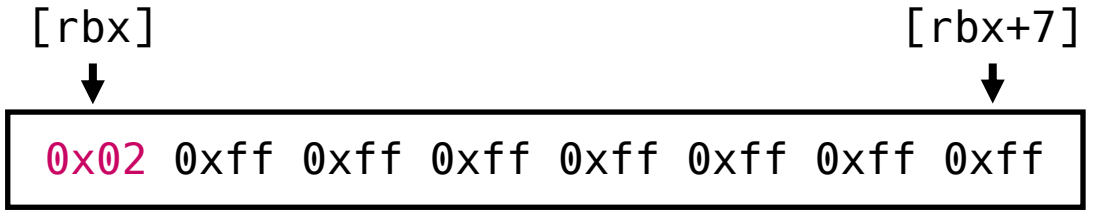
or

```
mov   DWORD PTR [rbx], 2
```

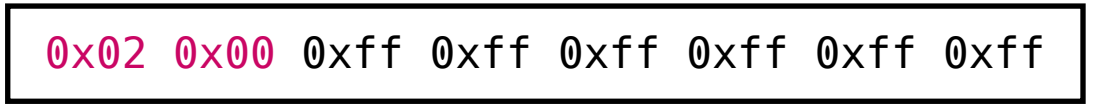
or

```
mov   QWORD PTR [rbx], 2
```

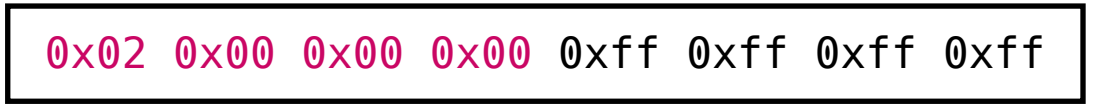
Byte at the addr rbx holds



Word at the addr rbx holds



Double word at the addr rbx holds

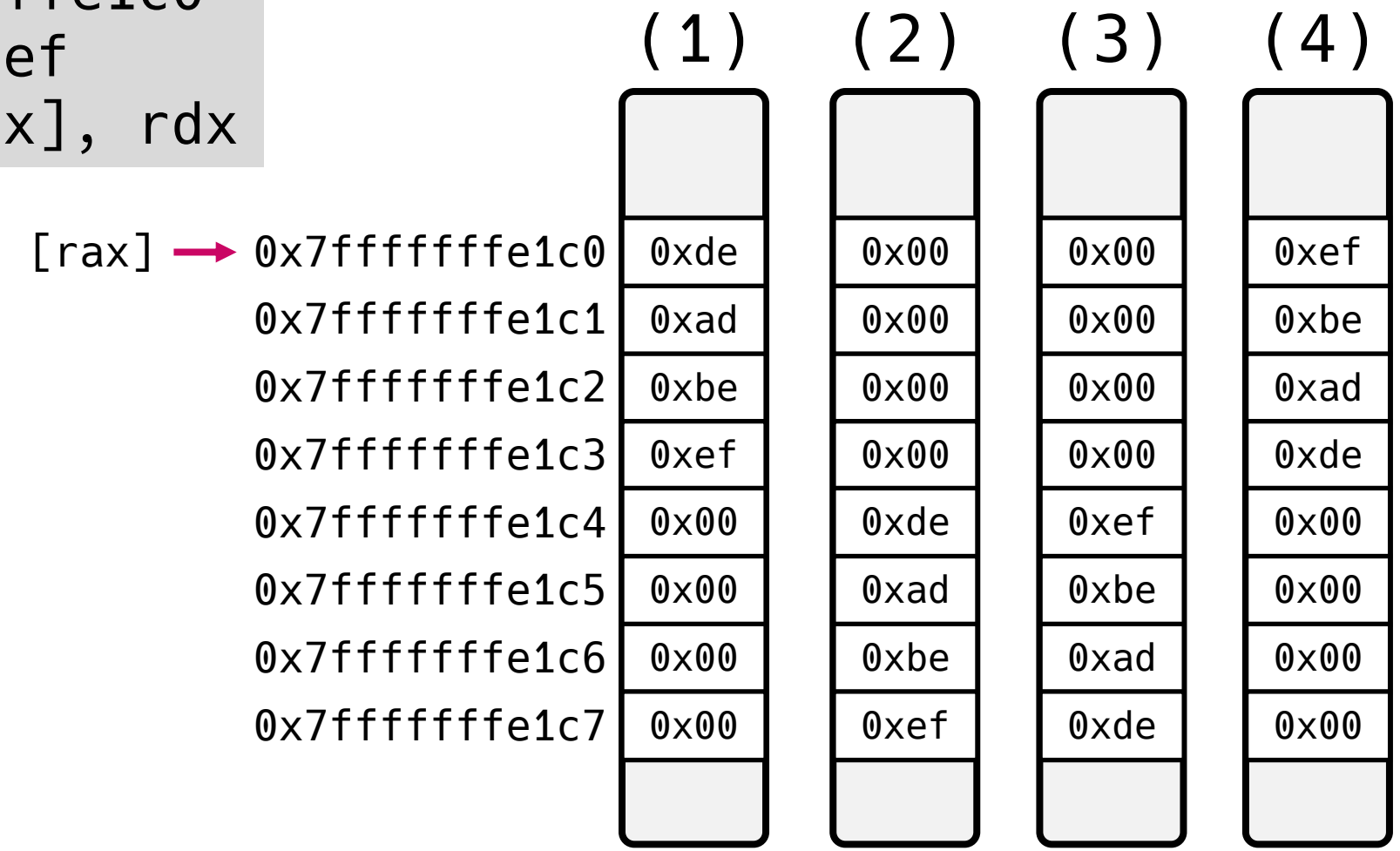


Quad word at the addr rbx holds

Note: Endianness (Order of bytes stored in mem)

```
mov   rax, 0x7ffffffe1c0
mov   rdx, 0xdeadbeef
mov   QWORD PTR [rax], rdx
```

Q) Which layout is correct?



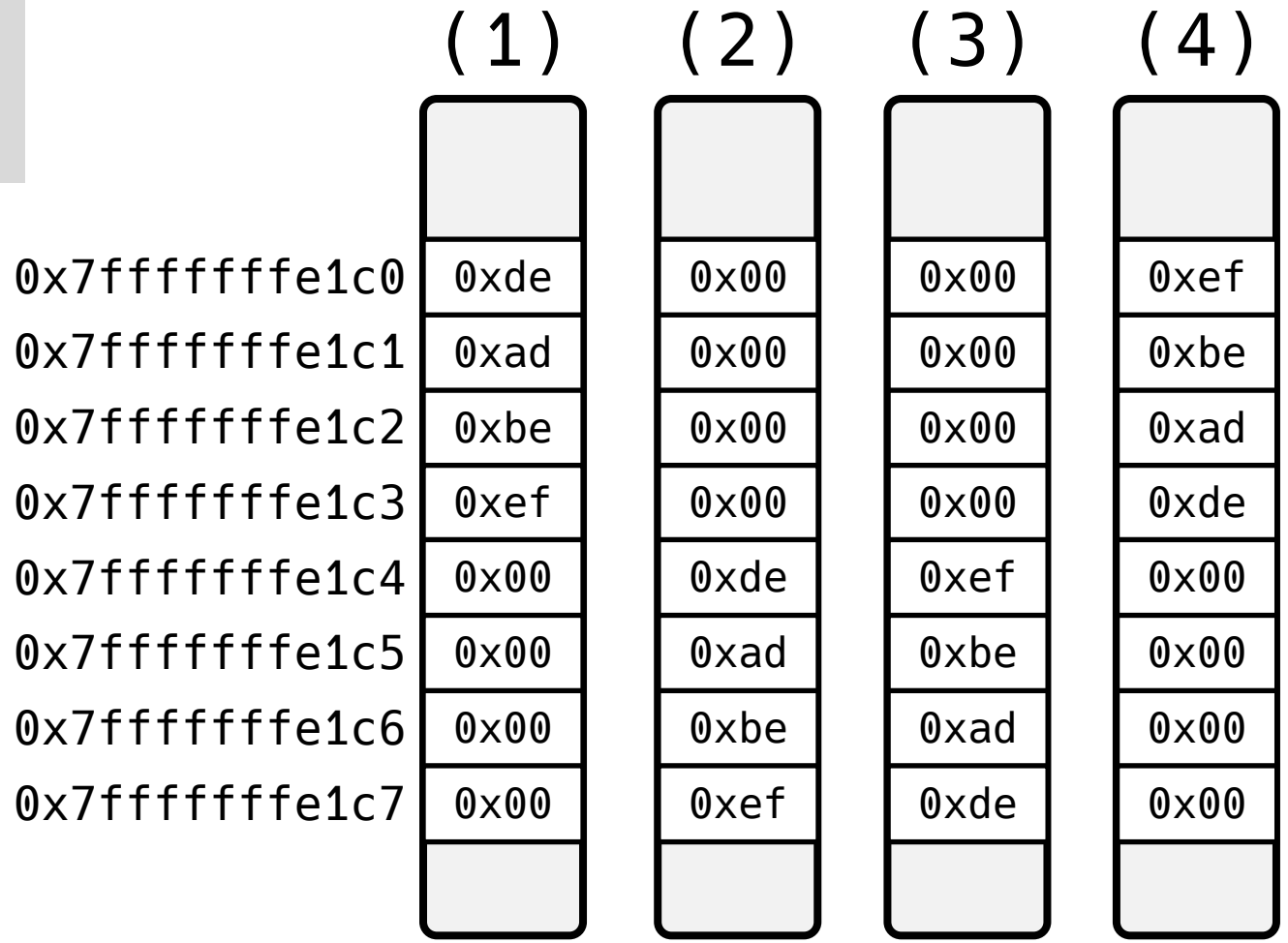
Note: Endianness (Order of bytes stored in mem)


```
mov   rax, 0x7ffffffe1c0
mov   rdx, 0xdeadbeef
mov   QWORD PTR [rax], rdx
```

Q) What are the
- Most significant bit (MSB) and
- Least significant bit (LSB) of `rdx`?

A) Since `rdx` (x64 register) is 64 bits long,
→ `rdx = 0x00000000deadbeef`
MSB: 0 LSB: 1

Q) Which layout is correct?



Note: Endianness (Order of bytes stored in mem)

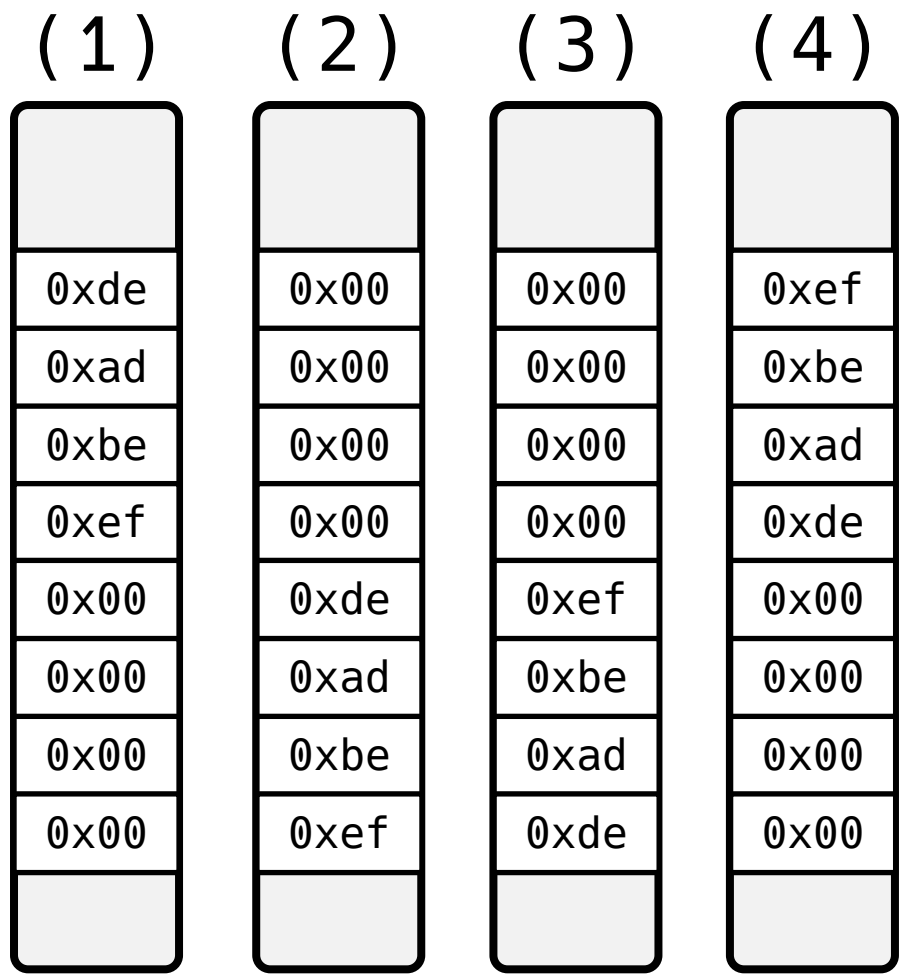

```
mov   rax, 0x7ffffffe1c0
mov   rdx, 0xdeadbeef
mov   QWORD PTR [rax], rdx
```

Q) What are the
- Most significant bit (MSB) and
- Least significant bit (LSB) of `rdx`?

A) Since `rdx` (x64 register) is 64 bits long,
→ `rdx = 0x00000000deadbeef`
MSB: 0 LSB: 1

- Endianness
  - Big Endian: Store MSB in the lowest addr
  - Little Endian: Store LSB in the lowest addr

Q) Which layout is correct?



x64 (x86) uses Little Endian ordering

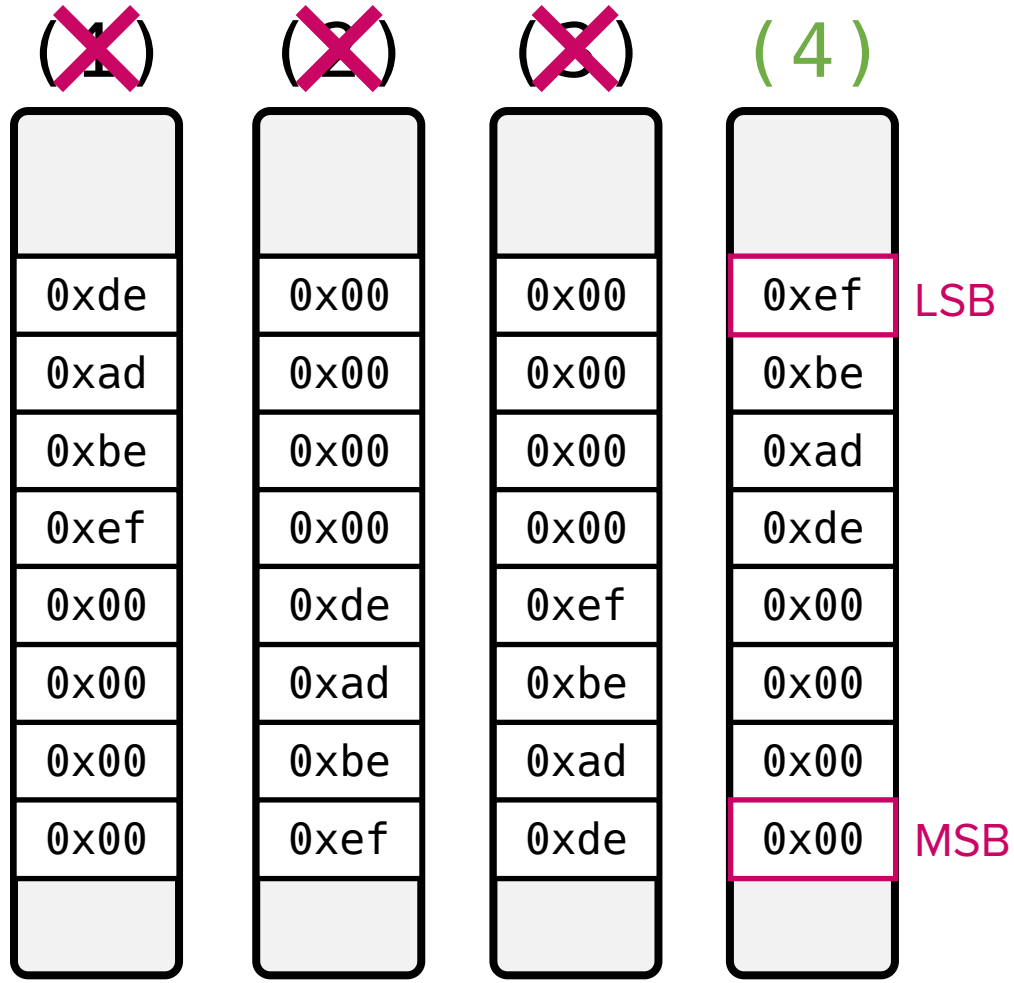
POSTECH

```
mov   rax, 0x7ffffffe1c0
mov   rdx, 0xdeadbeef
mov   QWORD PTR [rax], rdx
```

Q) What are the
- Most significant bit (MSB) and
- Least significant bit (LSB) of `rdx`?

A) Since `rdx` (x64 register) is 64 bits long,
→ `rdx = 0x00000000deadbeef`
MSB: 0 LSB: 1

- **Endianness**
  - Big Endian: Store MSB in the lowest addr
  - Little Endian: Store LSB in the lowest addr



`mov` instruction

- `mov` copies data from one place to another
  - Despite the name “move”, it does not remove data from the source
  - e.g., `mov rax, rbx`
    - Copies the value in `rbx` (source) into `rax` (destination)

General forms of mov

- Register to register `mov rax, rbx`
  - Copies the value in `rbx` into `rax`
- Immediate value to register `mov rax, 0xdeadbeef`
  - Places the immediate value `0xdeadbeef` into `rax`
- Memory to register `mov rax, [rbx]`
  - Takes the value at memory address stored in `rbx` and places it in `rax`
- Register to memory `mov [rax], rbx`
  - Copies the value in `rbx` into the memory address pointed to by `rax`
- Immediate value to memory `mov [rax], 0xdeadbeef`
  - Places the immediate value `0xdeadbeef` at the memory pointed to by `rax`

`lea`: Load effective address

- `lea` computes the address of operand 2 and places the result in operand 1
  - e.g., `lea  rax, [rbx + 8]`

`rbx = 0x555555555ff0` → `0x555555555ff0`

`0x555555555ff8`

`0x555555556000`

Q) `rax = ?`

A) `rax = 0x555555555ff8`

→ The address (not the value) is loaded

mov vs. lea

Given: `rbp = 0x7fffffffe338`

```
mov  rsp, [rbp-0x8]
```

`rsp` = value at address `0x7fffffffe330`
i.e., `rsp = *(rbp - 0x8);`

```
lea  rsp, [rbp-0x8]
```

`rsp` = `0x7fffffffe330`
i.e., `rsp = (rbp - 0x8);`

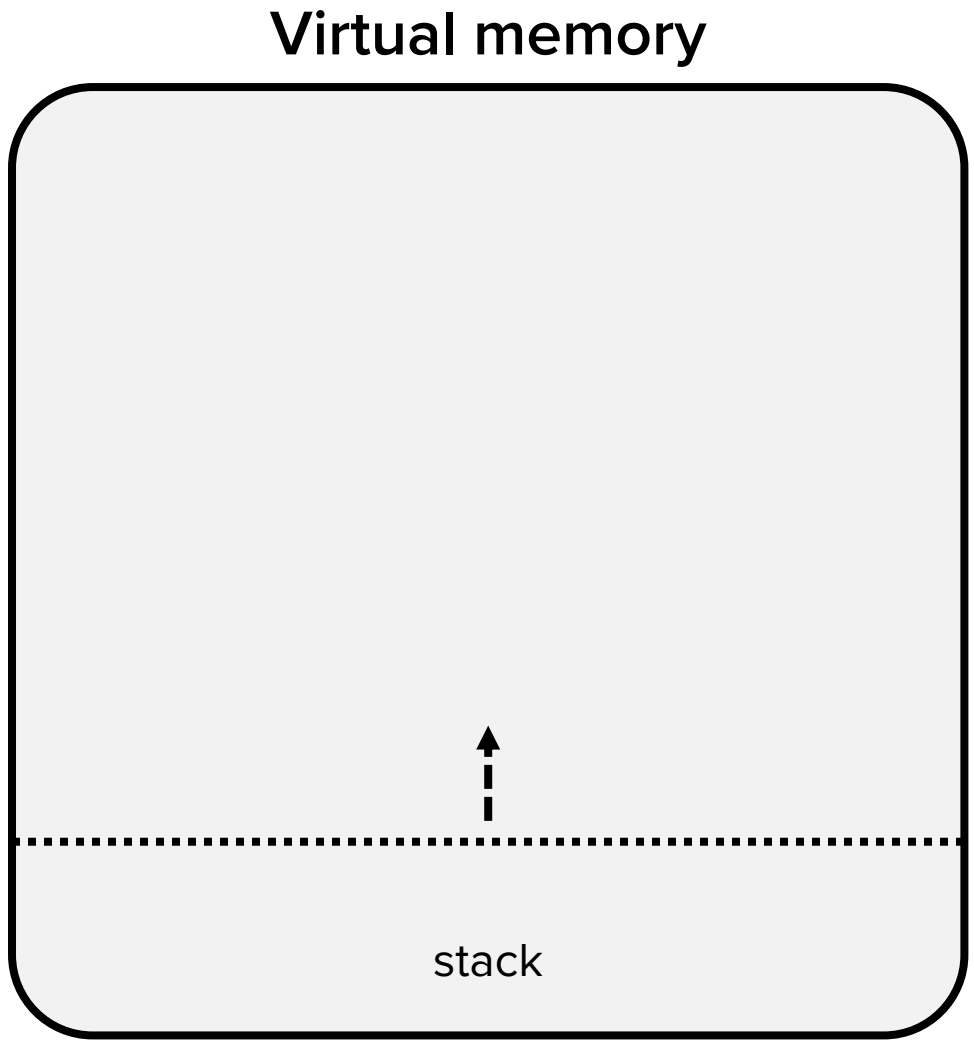
Arithmetic operations

- Perform addition, subtraction, multiplication, and more

```
add rax, rbx == rax = rax + rbx

sub rax, rcx == rax = rax - rcx

mul rbx      == rax = rbx * rax (result in rax, overflow in rdx)

div rbx      == rax = rbx / rax (quotient in rax, remainder in rdx)

inc rax      == rax = rax + 1

dec rax      == rax = rax - 1
```

Stack operations

- `rsp` points to the top of the stack



`rsp = 0x7fffffffe330` → `0x7fffffffe330`

Stack operations: push

- push enlarges the stack

```
push   rax
```

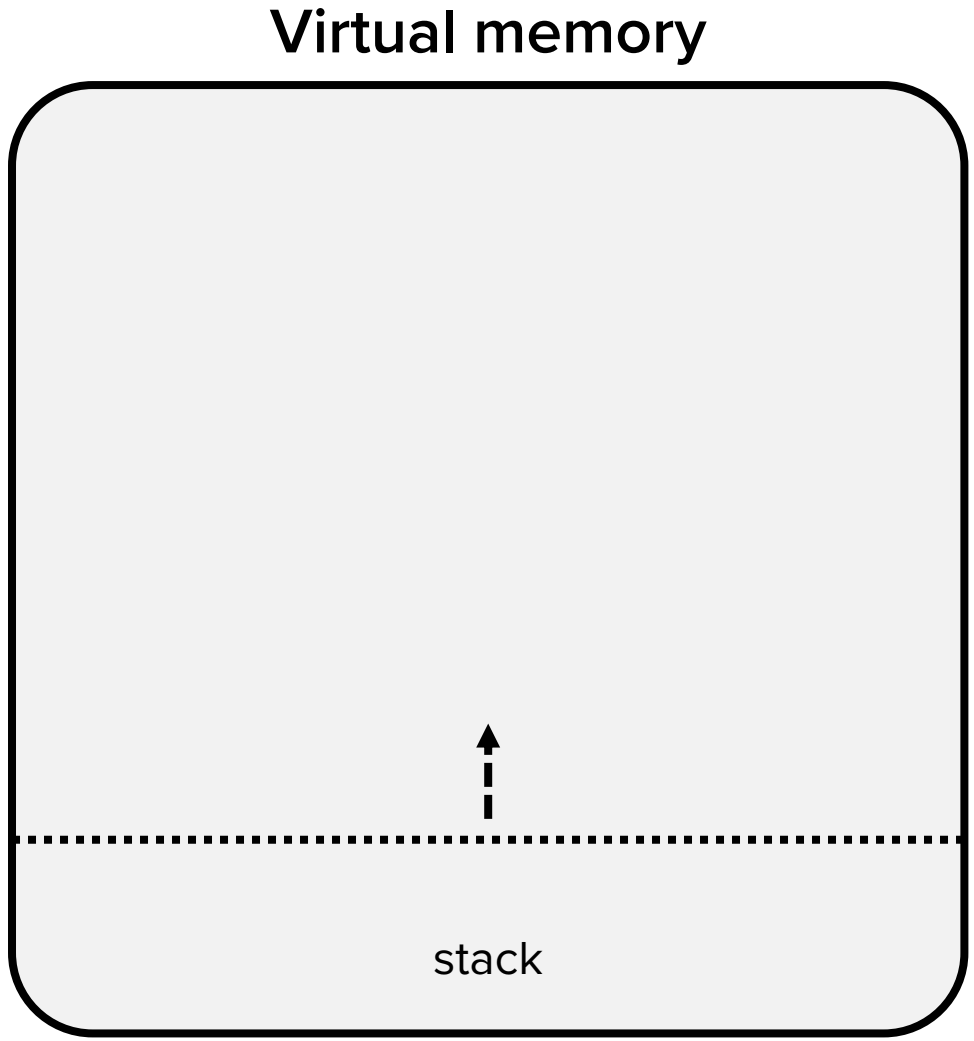
Push the value rax holds

```
push   [rax]
```

Push the value at the address rax points to

```
push   0x11223344
```

Push immediate value



`rsp = 0x7fffffffe330` → `0x7fffffffe330`

Stack operations: push

- push enlarges the stack

```
push   rax
```

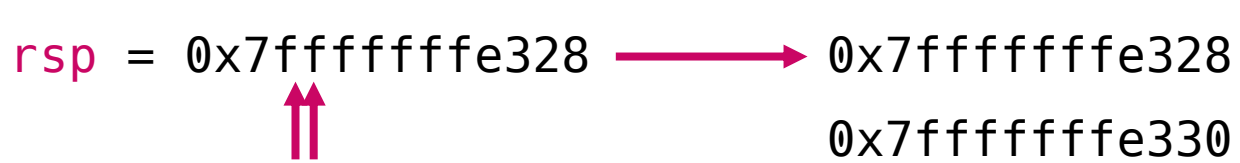
Push the value rax holds

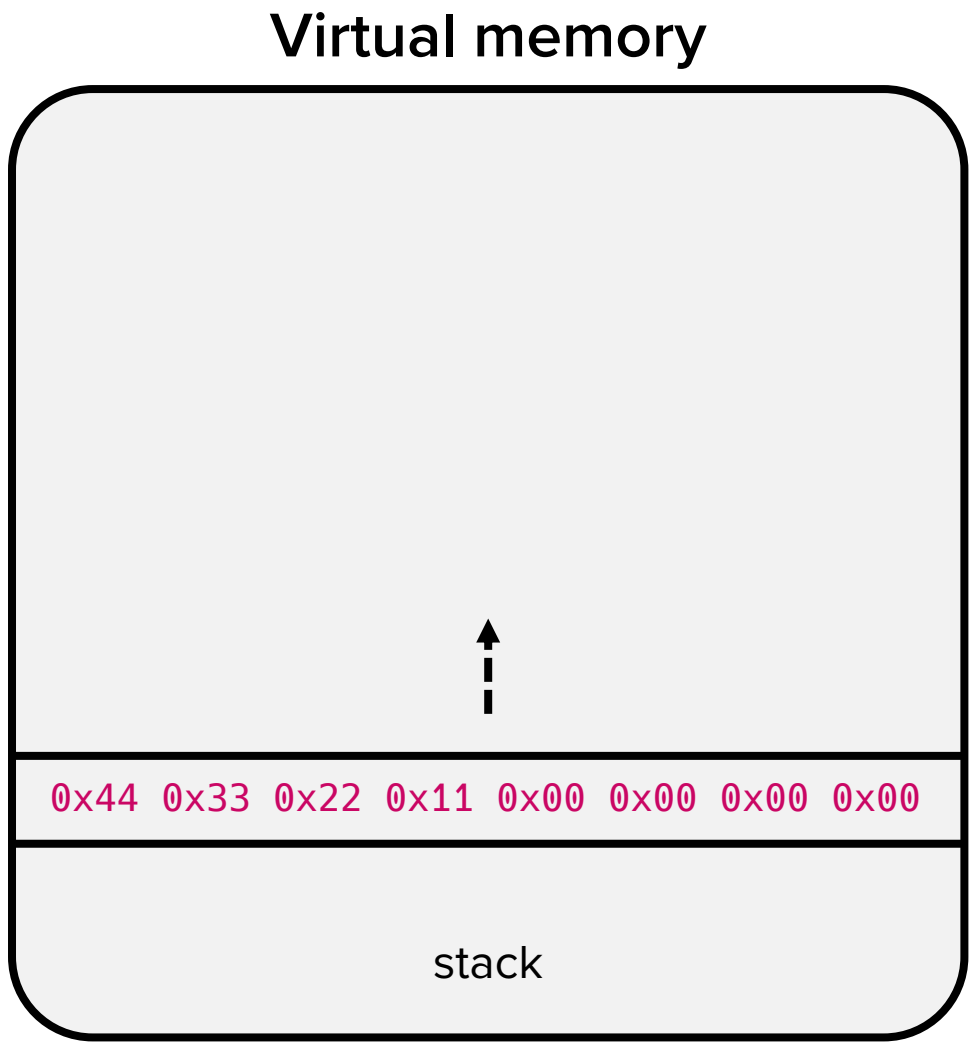
```
push   [rax]
```

Push the value at the address rax points to

```
push   0x11223344
```

Push constant value

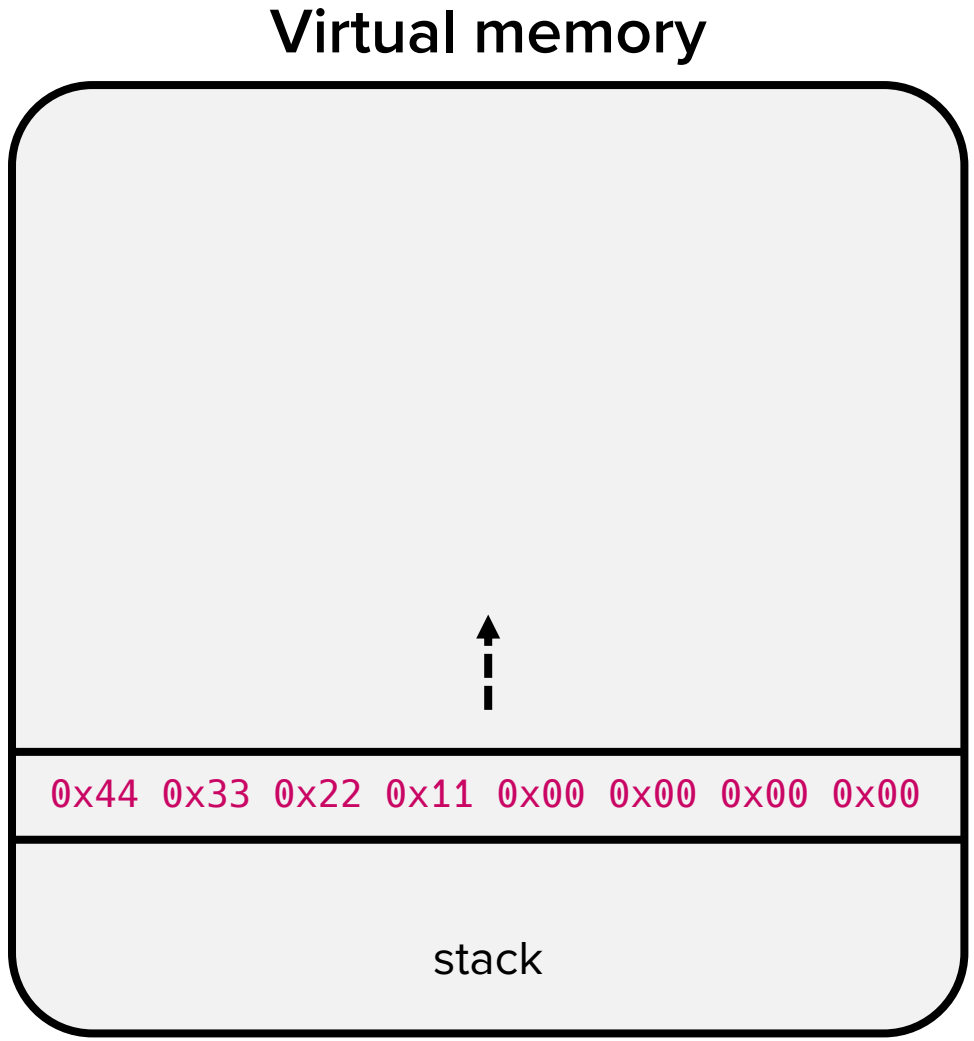




Stack operations: push

- push equivalence

```
push   rax   ==   sub   rsp, 8
                  mov   [rsp], rax
```



Stack operations: pop

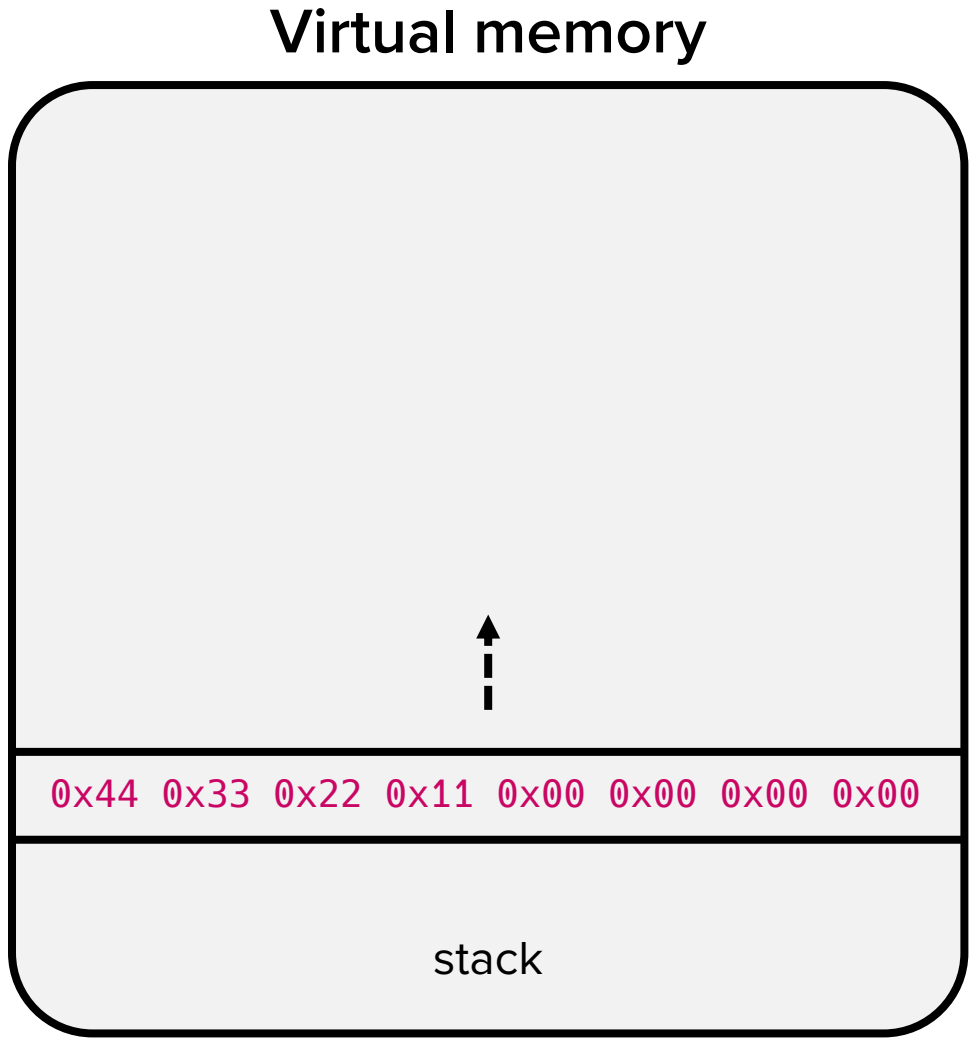
- pop shrinks the stack

```
pop   rax
```

Pop the stack top into rax

```
pop   [rax]
```

Pop the stack top into the memory pointed to by rax



Stack operations: pop

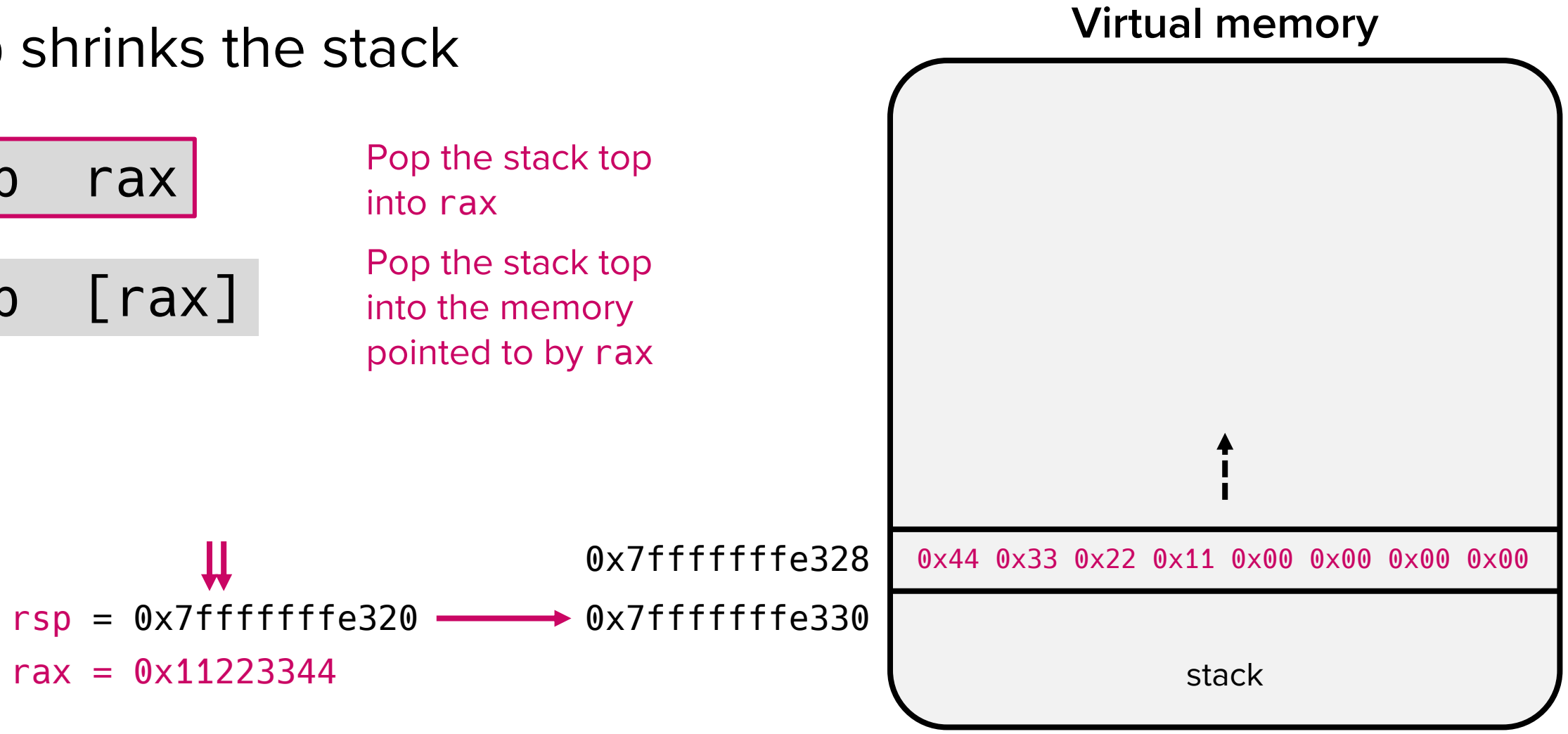
- pop shrinks the stack

```
pop   rax
```

Pop the stack top into rax

```
pop   [rax]
```

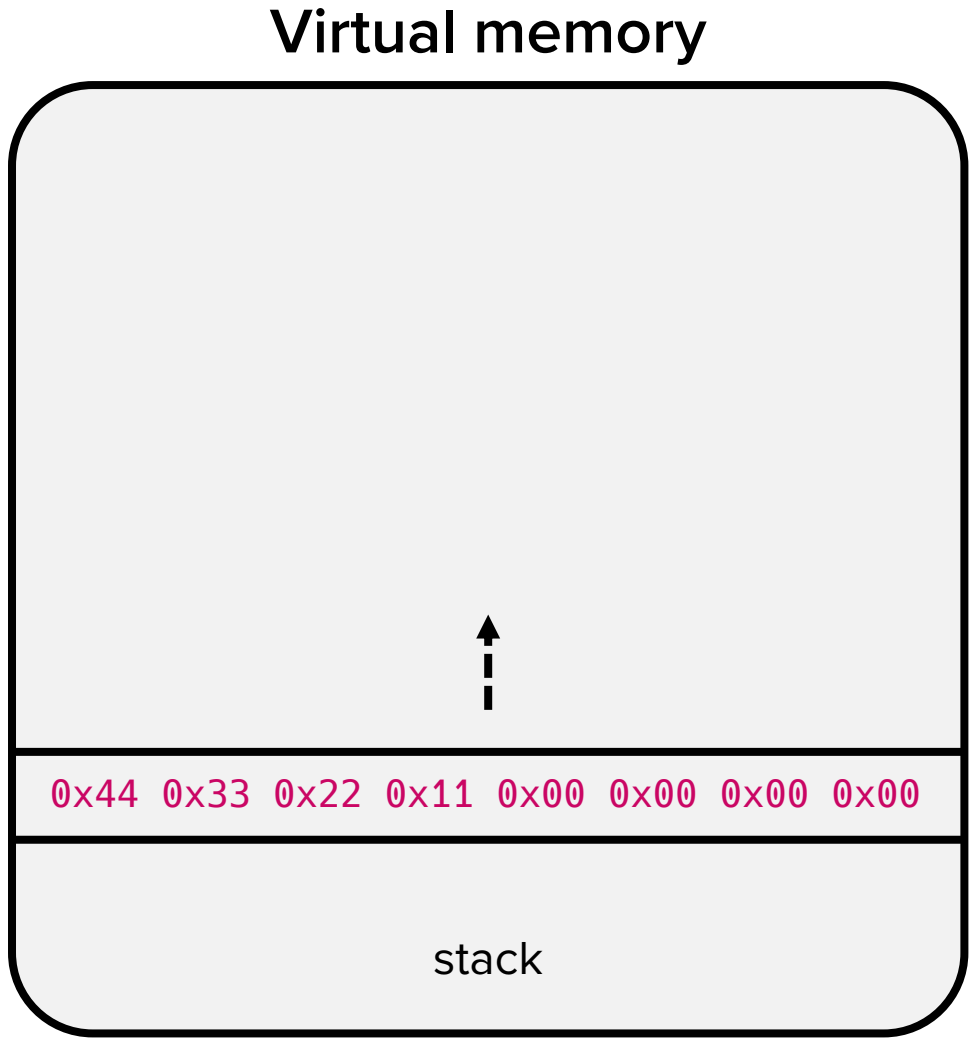
Pop the stack top into the memory pointed to by rax



Stack operations: pop

- pop equivalence

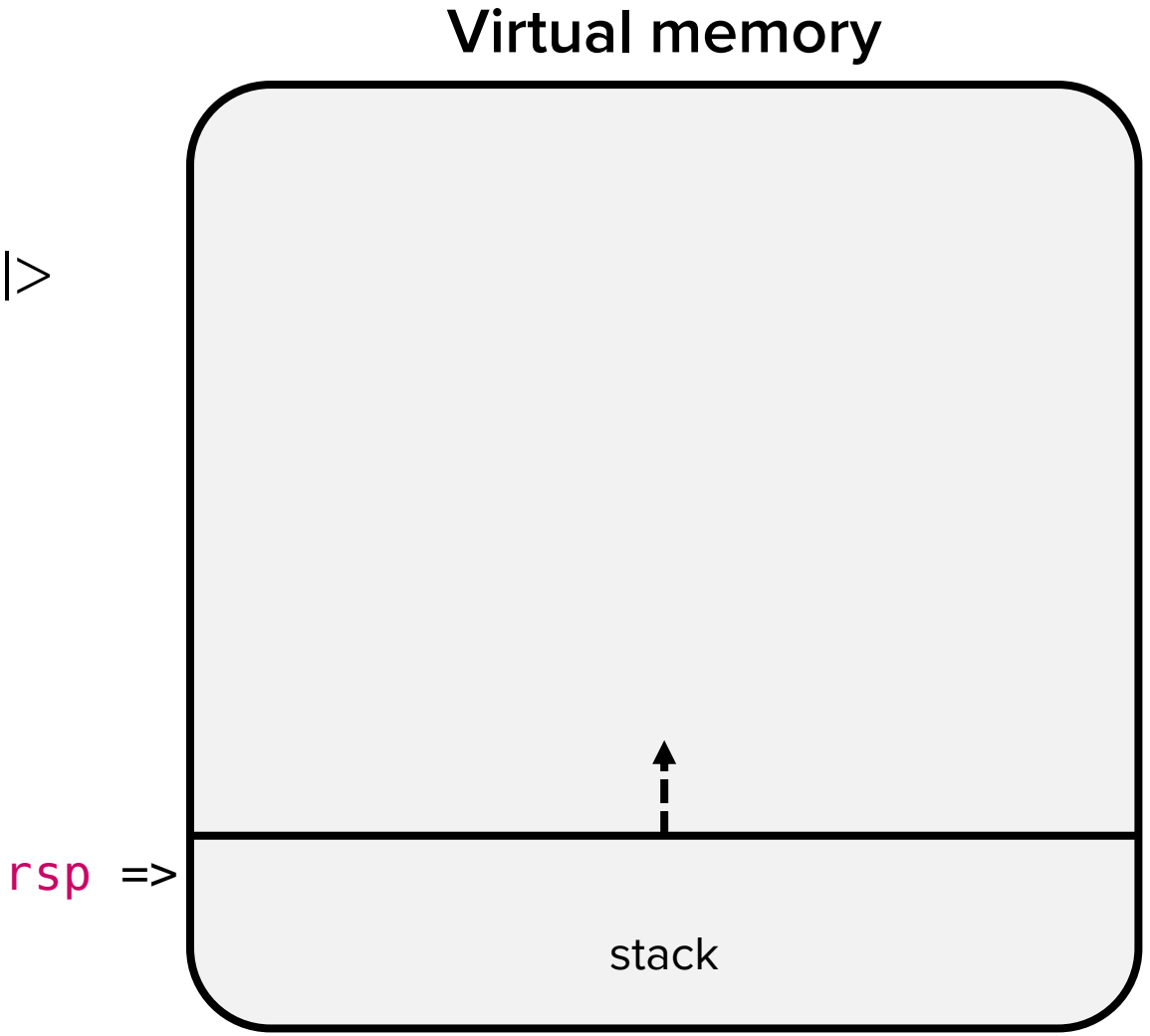
```
pop   rax   ==   mov   rax, [rsp]
                 add   rsp, 8
```



Subroutine instructions: `call`

- `call` <label>
  - **`push`** the address to return to
  - perform unconditional **`jmp`** to <label>
  - e.g.,

```
rip => 0x55555555518d: call <some_func>
       0x555555555192: mov rax, 0
       ...

       <some_func>
       0x55555555514d: push rbp
       0x55555555514e: mov rbp, rsp
       ...
       0x555555555184: ret
```



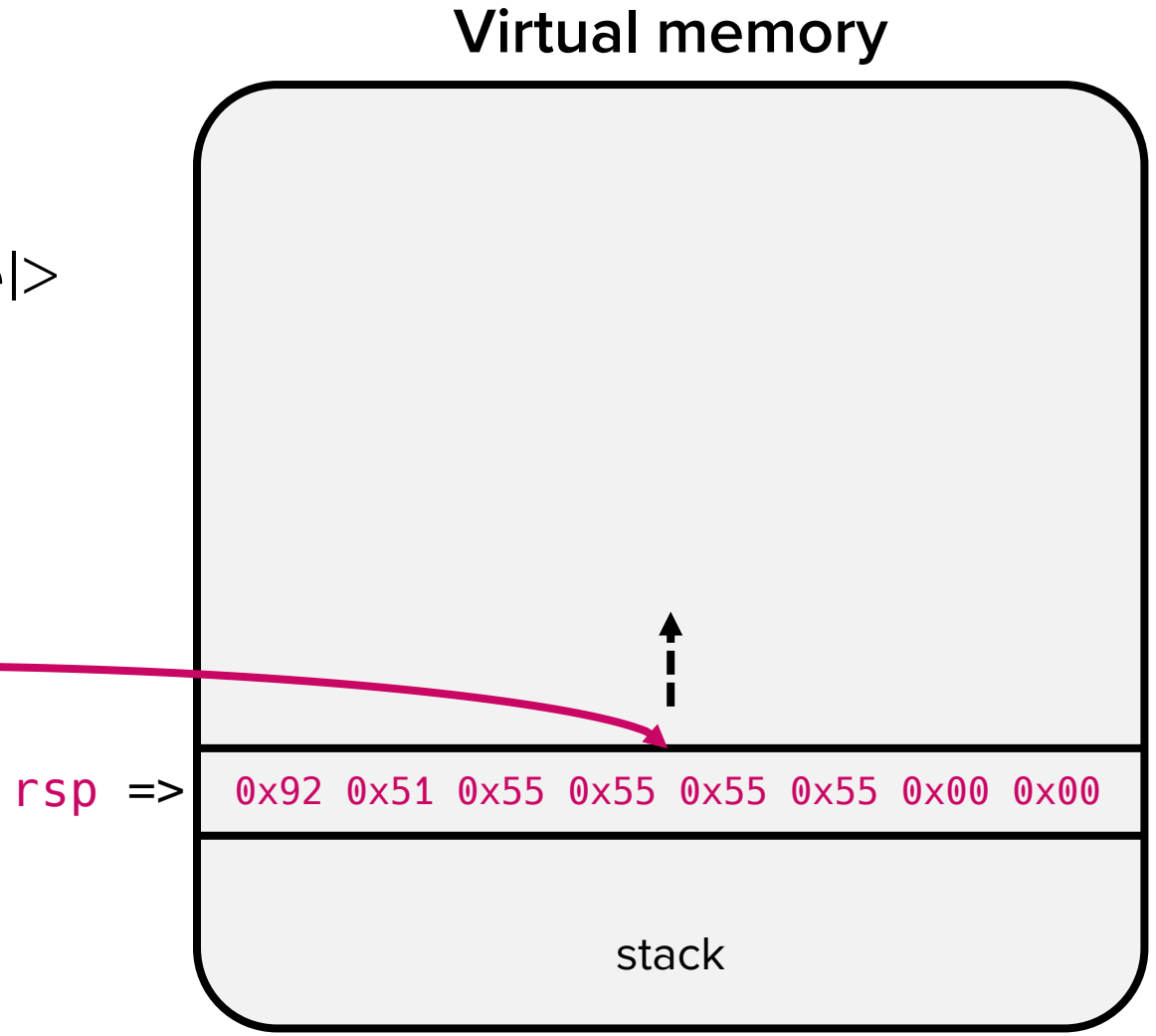
Subroutine instructions: `call`

POSTECH

- `call` <label>
  - `push` the address to return to
  - perform unconditional `jmp` to <label>
  - e.g.,

```
rip => 0x55555555518d: call <some_func>
       0x555555555192: mov rax, 0
       ...
```

```
<some_func>
0x55555555514d: push rbp
0x55555555514e: mov rbp, rsp
...
0x555555555184: ret
```

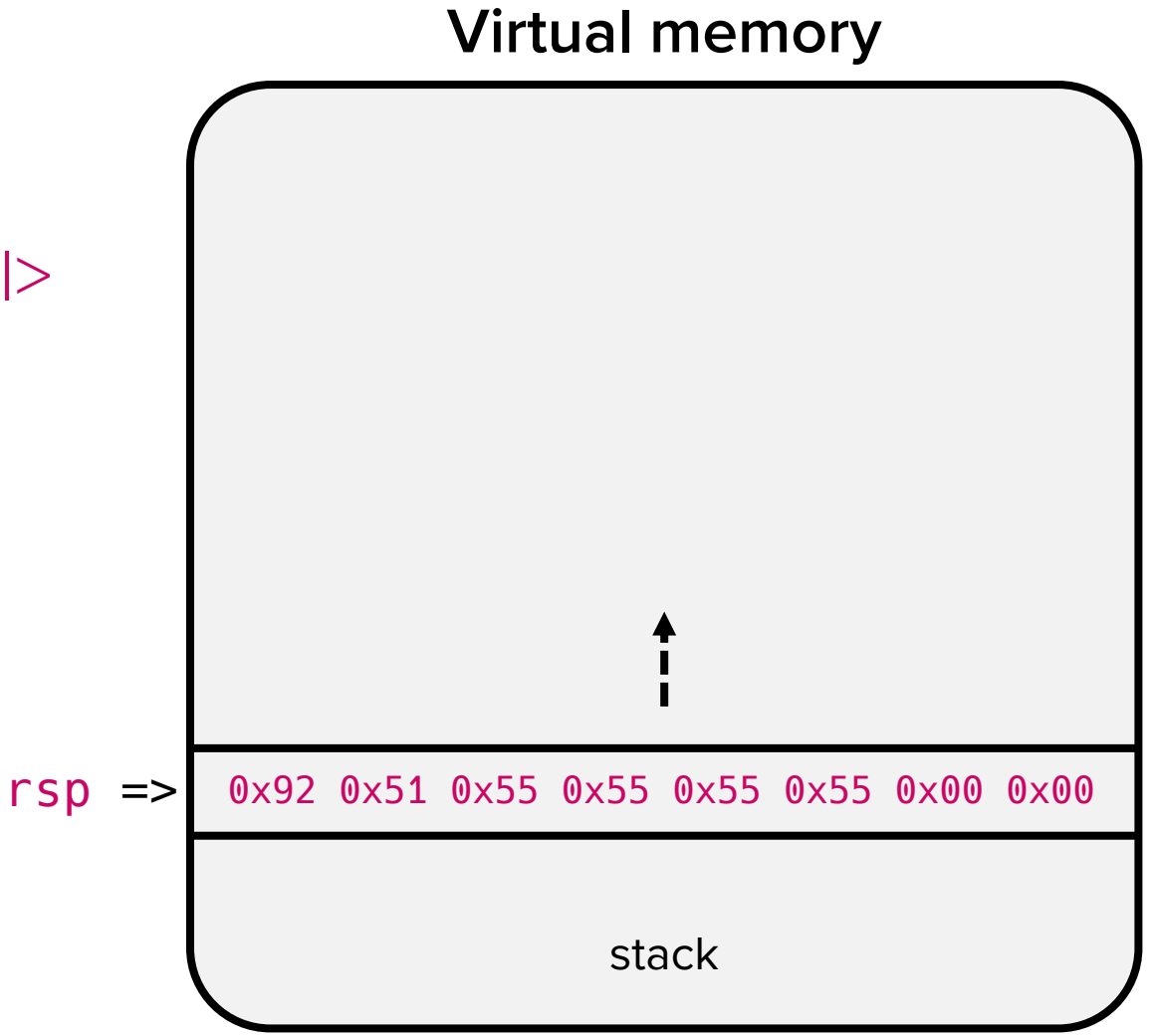


Subroutine instructions: `call`

- `call` <label>
  - `push` the address to return to
  - perform unconditional `jmp` to <label>
  - e.g.,

```
0x55555555518d: call <some_func>
0x555555555192: mov rax, 0
...
```

```
rip =>  <some_func>
        0x55555555514d: push rbp
        0x55555555514e: mov rbp, rsp
        ...
        0x555555555184: ret
```

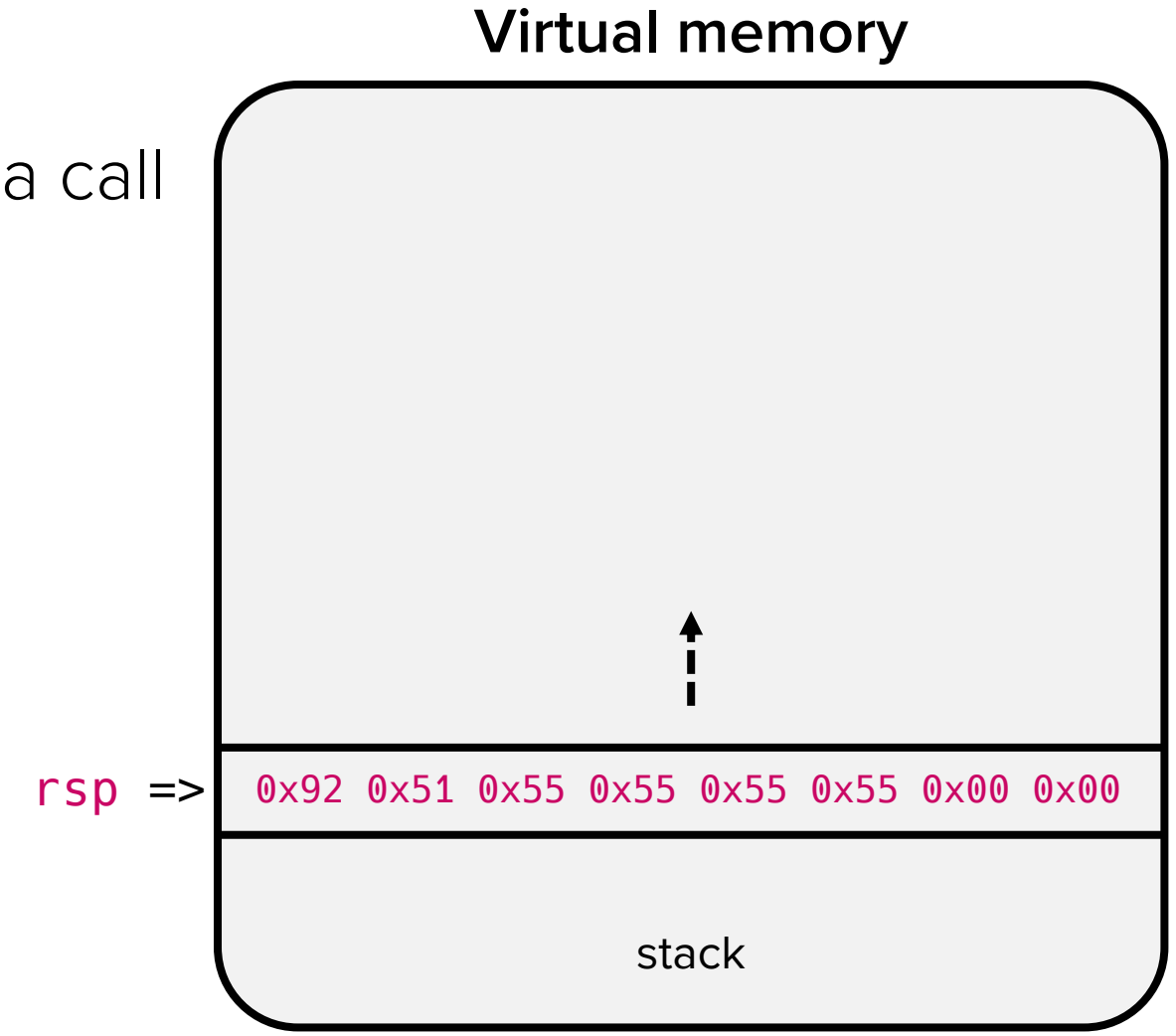


Subroutine instructions: `call`

- `call` <label>
  - Function prologue usually follows a call
    - `push rbp; mov rbp, rsp`
  - e.g.,

```
0x55555555518d: call <some_func>
0x555555555192: mov rax, 0
...
```

```
rip =>  <some_func>
        0x55555555514d: push rbp
        0x55555555514e: mov rbp, rsp
        ...
        0x555555555184: ret
```



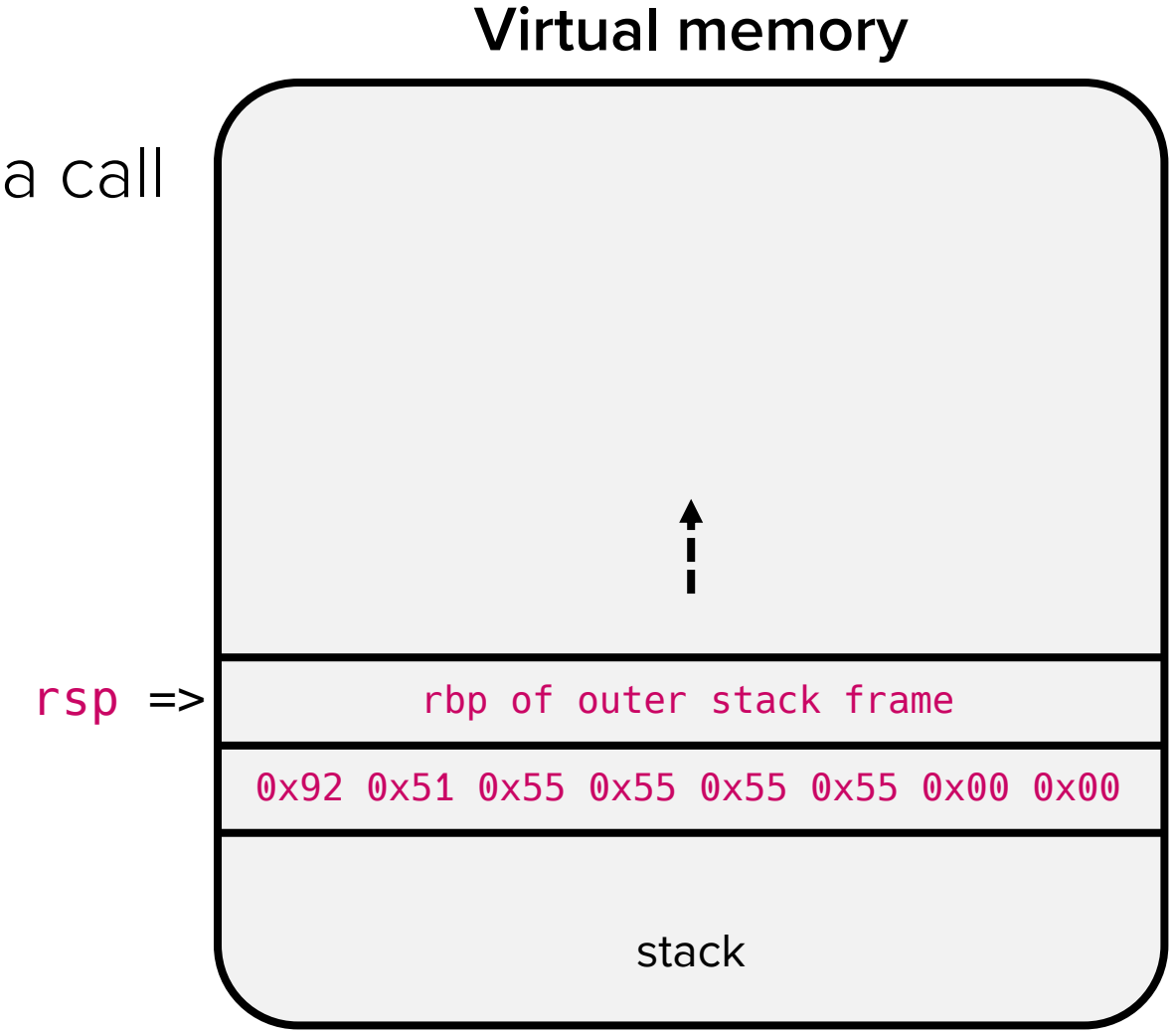
Subroutine instructions: `call`

- `call` <label>
  - Function prologue usually follows a call
    - `push rbp; mov rbp, rsp`
  - e.g.,

```
0x55555555518d: call <some_func>
0x555555555192: mov rax, 0
...
```

```
<some_func>
0x55555555514d: push rbp
0x55555555514e: mov rbp, rsp
...
0x555555555184: ret
```

rip =>



Subroutine instructions: `call`

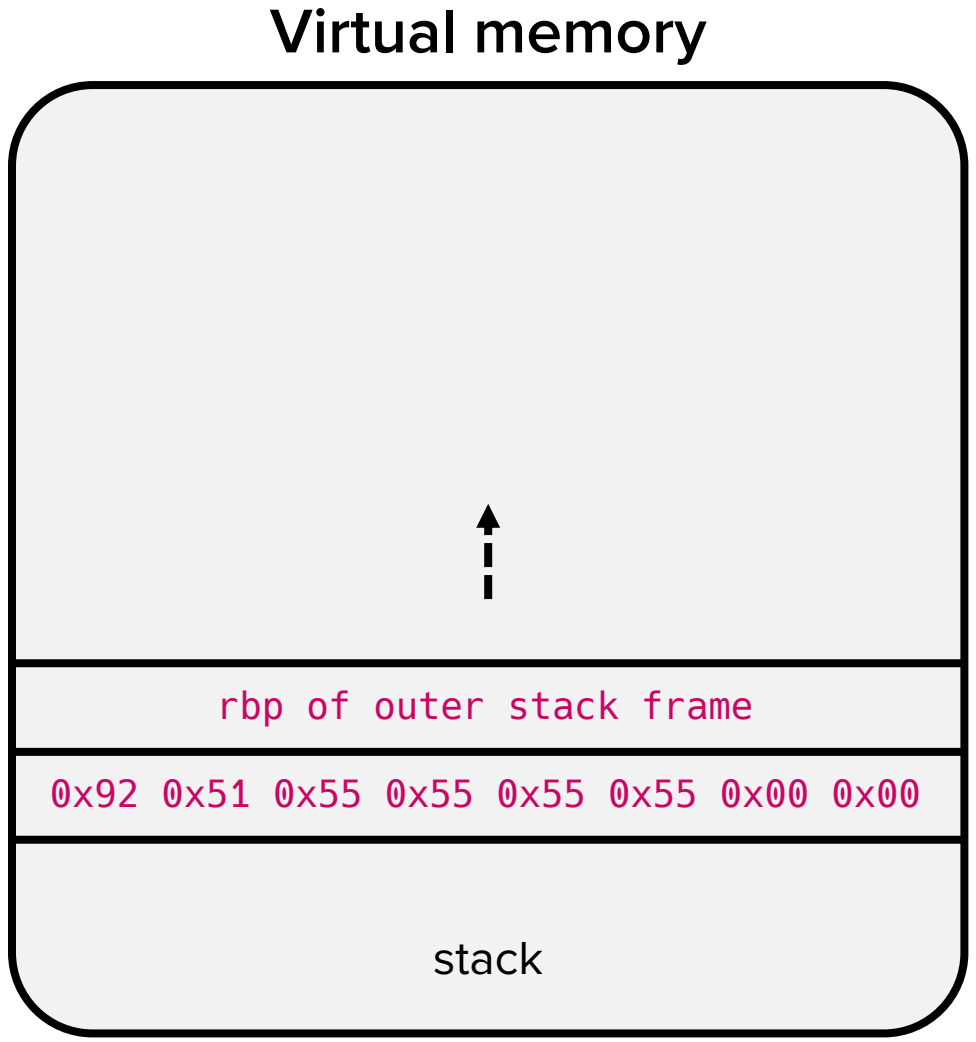
- `call` <label>
  - Function prologue usually follows a call
    - `push rbp; mov rbp, rsp`
  - e.g.,

```
0x55555555518d: call <some_func>
0x555555555192: mov rax, 0
...
```

```
<some_func>
0x55555555514d: push rbp
0x55555555514e: mov rbp, rsp
...
0x555555555184: ret
```

rip =>

rbp, rsp =>

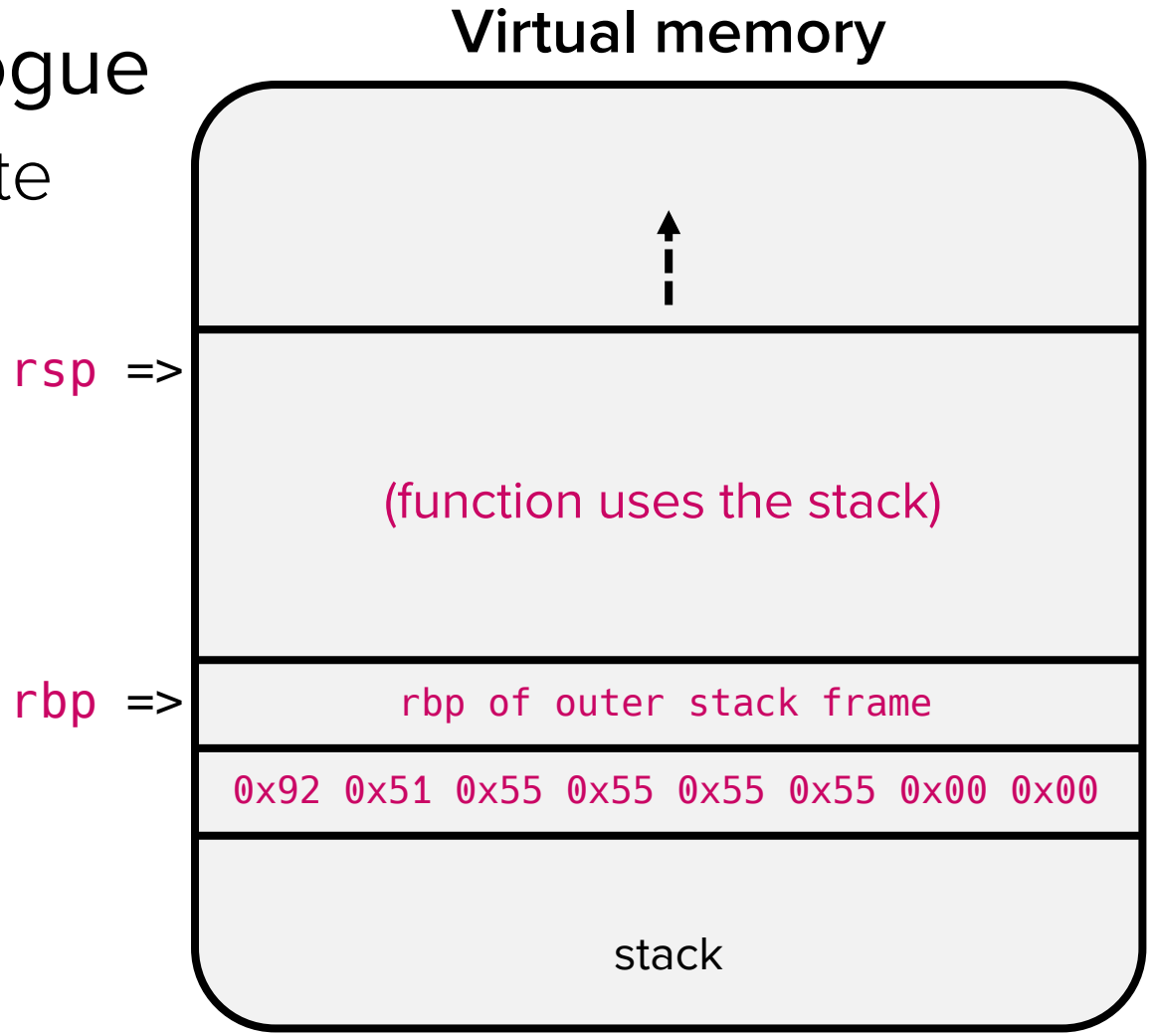


Subroutine instructions: `leave`

- `leave` undoes the function prologue
  - Restores the stack to a pre-call state

```
leave == mov  rsp, rbp  (1)
         pop  rbp       (2)
```

```
        <some_func>
        ...
rip =>  0x555555555183: leave
        0x555555555184: ret
```

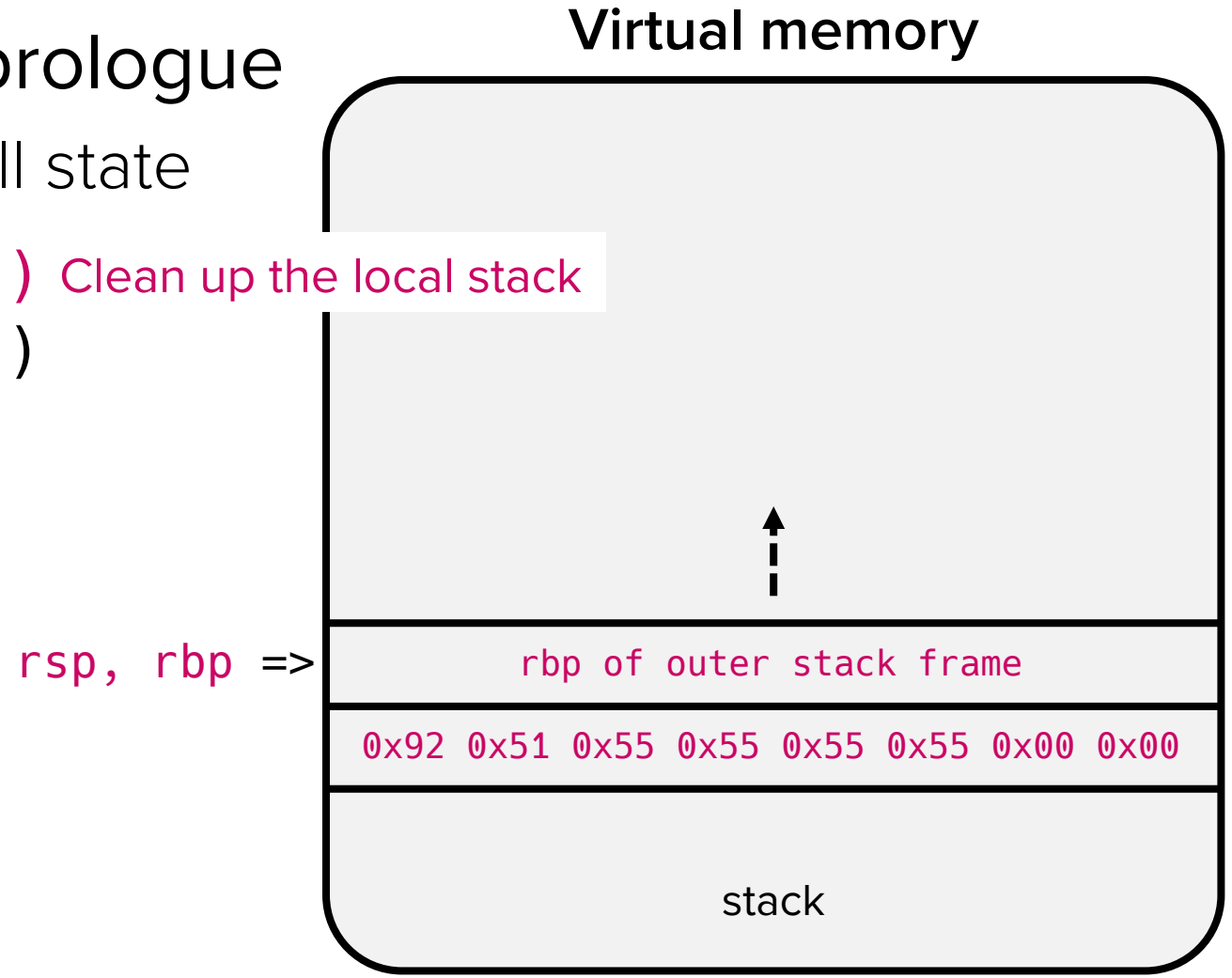


Subroutine instructions: `leave`

- `leave` undoes the function prologue
  - Restores the stack to a pre-call state

```
leave == mov  rsp, rbp   (1) Clean up the local stack
         pop  rbp        (2)
```

```
        <some_func>
        ...
rip =>  0x555555555183: leave
        0x555555555184: ret
```

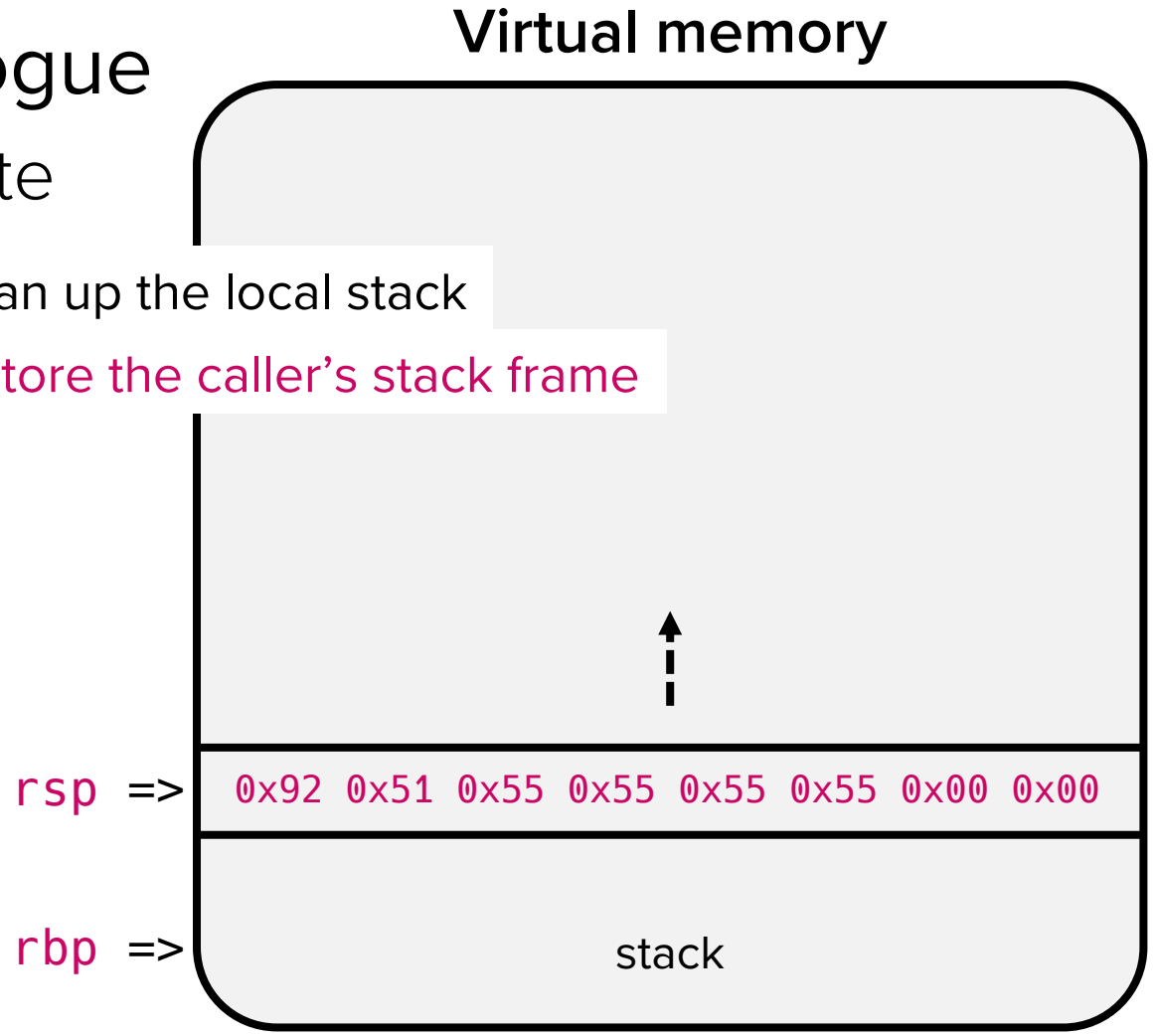


Subroutine instructions: `leave`

- `leave` undoes the function prologue
  - Restores the stack to a pre-call state

```
leave == mov  rsp, rbp   (1) Clean up the local stack
         pop  rbp        (2) Restore the caller's stack frame
```

```
        <some_func>
        ...
        0x555555555183: leave
rip =>  0x555555555184: ret
```



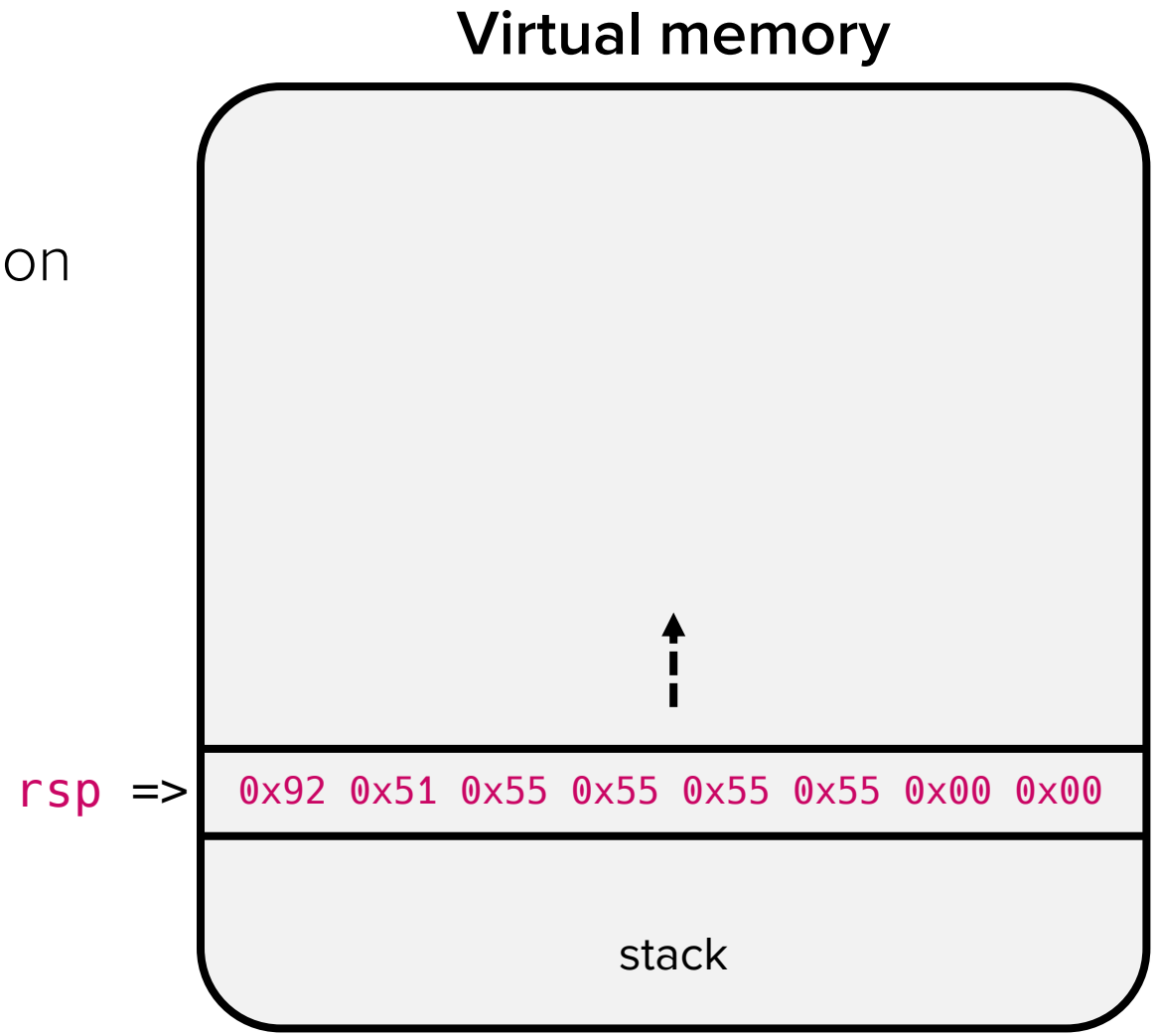
Subroutine instructions: `ret`

- `ret`
  - `pop` stack top into `rip`
    - Program re-executes the caller function
  - e.g.,

```
0x55555555518d: call <some_func>
0x555555555192: mov rax, 0
...
```

```
<some_func>
...
0x555555555183: leave
0x555555555184: ret
```

`rip =>`



Subroutine instructions: `ret`

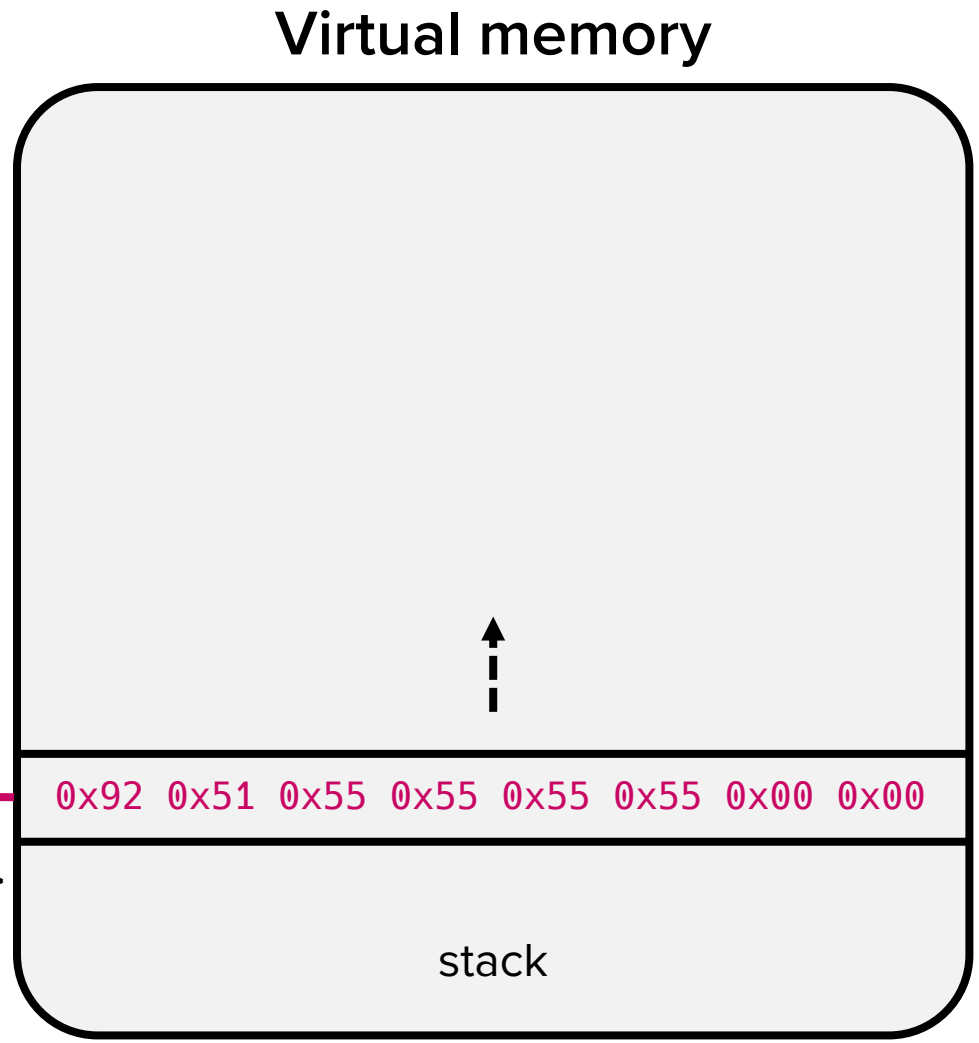
POSTECH

- `ret`
  - `pop` stack top into `rip`
    - Program re-executes the caller function
  - e.g.,

```
0x55555555518d: call <some_func>
0x555555555192: mov rax, 0
...
```

`rip =>`

```
<some_func>
...
0x555555555183: leave
0x555555555184: ret
```



Branches in assembly

- Implemented using `cmp` + conditional jump

```
cmp   rax, 0xf
je    addr
```

```
if (rax == 15) {
  goto addr;
}
```

```
cmp   rax, 0xf
jne   addr
```

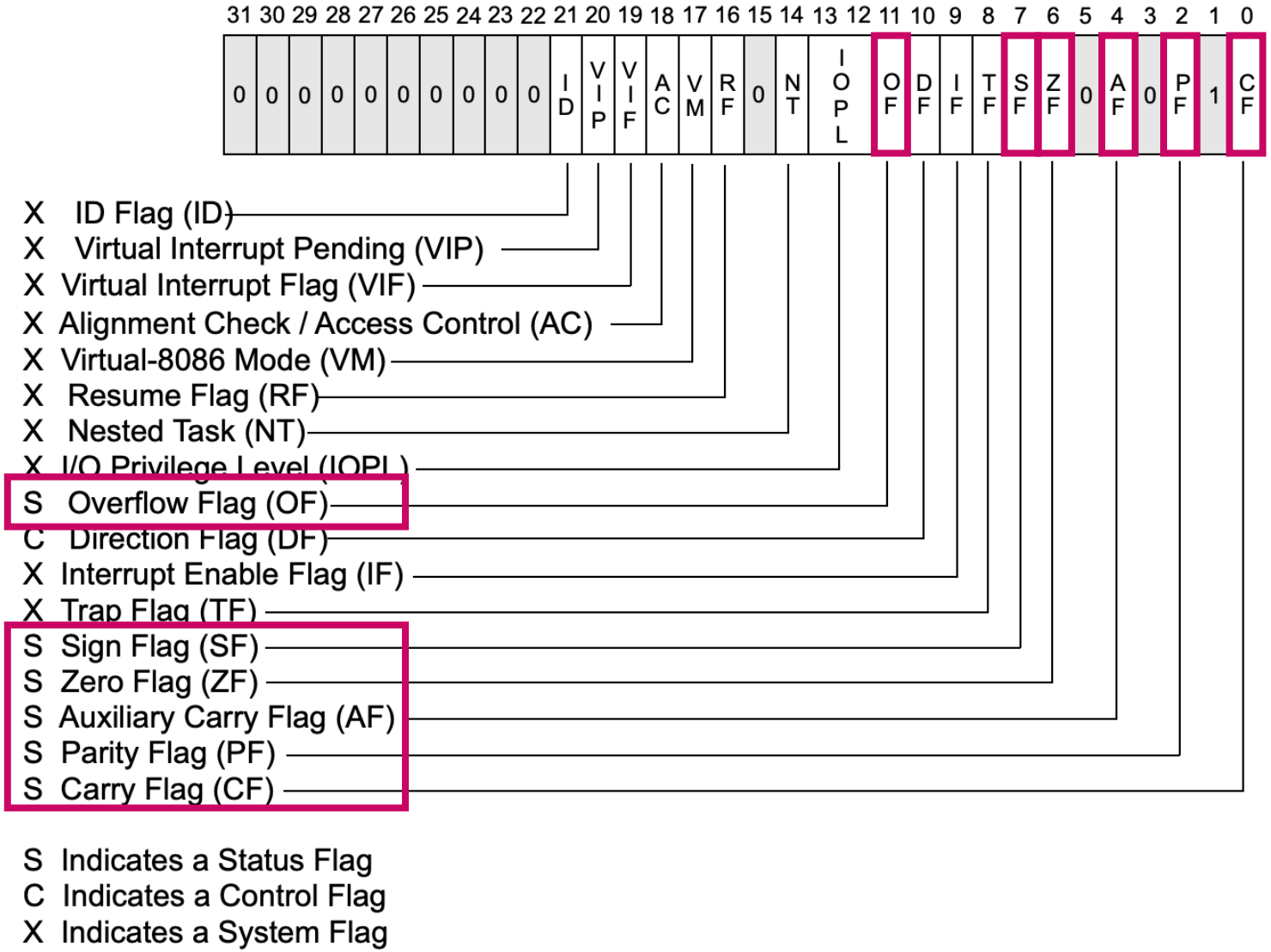
```
if (rax != 15) {
  goto addr;
}
```

```
cmp   rax, 0xf
jge   addr
```

```
if (rax >= 15) {
  goto addr;
}
```

Q) Where does `cmp` store the result??

rflags register stores status / control / system flags



img: Intel 64 and IA-32 Software Developer's Manual

Conditional jumps and their conditions

POSTECH

Instruction Mnemonic	Condition (Flag States)	Description
Unsigned Conditional Jumps		
JA/JNBE	(CF or ZF) = 0	Above/not below or equal
JAE/JNB	CF = 0	Above or equal/not below
JB/JNAE	CF = 1	Below/not above or equal
JBE/JNA	(CF or ZF) = 1	Below or equal/not above
JC	CF = 1	Carry
JE/JZ	ZF = 1	Equal/zero
JNC	CF = 0	Not carry
JNE/JNZ	ZF = 0	Not equal/not zero
JNP/JPO	PF = 0	Not parity/parity odd
JP/JPE	PF = 1	Parity/parity even
JCXZ	CX = 0	Register CX is zero
JECXZ	ECX = 0	Register ECX is zero
Signed Conditional Jumps		
JG/JNLE	((SF xor OF) or ZF) = 0	Greater/not less or equal
JGE/JNL	(SF xor OF) = 0	Greater or equal/not less
JL/JNGE	(SF xor OF) = 1	Less/not greater or equal
JLE/JNG	((SF xor OF) or ZF) = 1	Less or equal/not greater
JNO	OF = 0	Not overflow
JNS	SF = 0	Not sign (non-negative)
JO	OF = 1	Overflow
JS	SF = 1	Sign (negative)

img: Intel 64 and IA-32 Software Developer's Manual

Exercise: Reverse-engineering Lab 02's target binary

Summary

- ELF is a flexible, widely used format that organizes code and data into segments and sections
- Assembly can look intimidating, but it is just a direct mapping of CPU instructions
  - Move and load data (mov, lea)
  - Manipulate the stack (push, pop)
  - Perform arithmetic/logical operations (add, sub, and, xor, etc.)
  - Control flow (call, ret, jmp, cmp)

Coming up next

- Writing malicious assembly code
- Exploiting buffer overflow to alter program's execution flow

Questions?