Lec 05:
Basics of Binary Analysis

CSED415: Computer Security

Spring 2026

Seulbae Kim

POSTECH
POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY

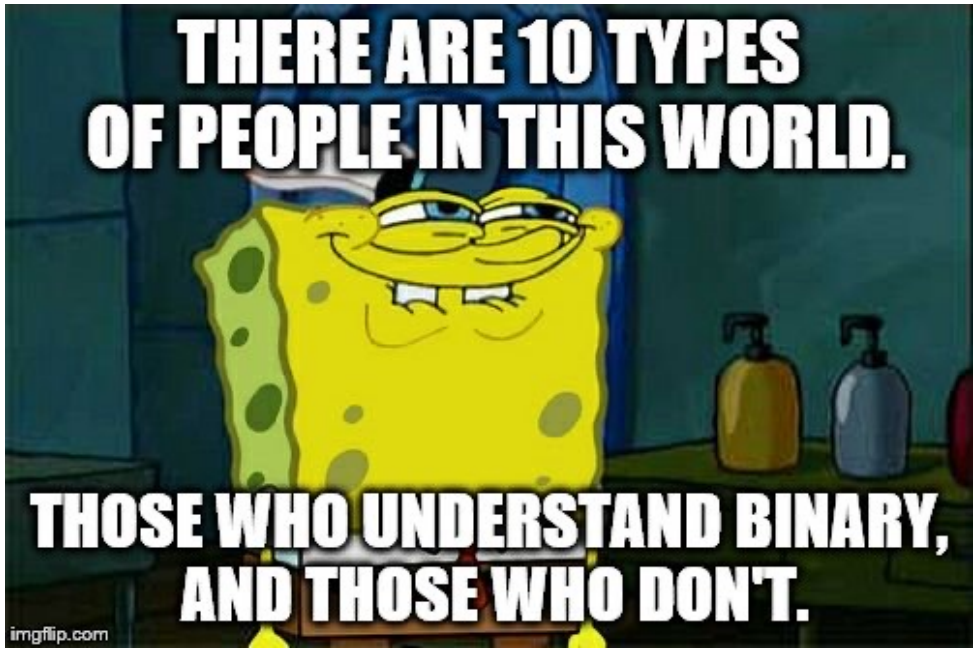
Announcements

- Lab 01 deadline has passed
  - Grace period until March 16 (submissions will receive 50% credit)
- Important: No in-class lecture on Wed, March 11
  - I will be traveling for a project proposal presentation (with a few other CSE faculty)
  - Lecture recording will be uploaded on Wednesday
    - Please watch it before next week's class!

Review of Week 2

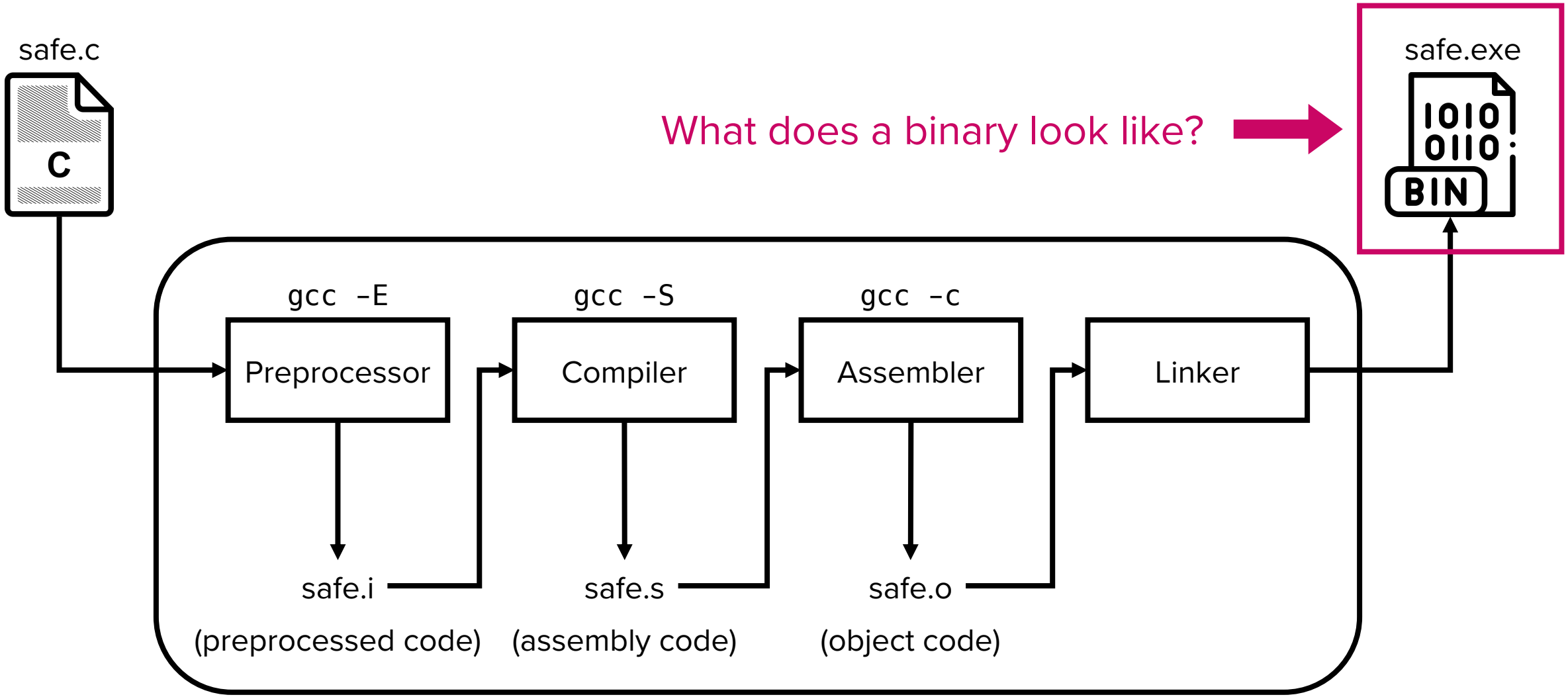
- Secure coding is necessary for building secure systems
- However, code-level mitigation may not be enough
  - We cannot trust anything that we did not create ourselves
    - e.g., The Thompson Compiler produces malicious binaries
- Binary analysis is required for us to analyze exactly what a CPU executes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

BIN


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

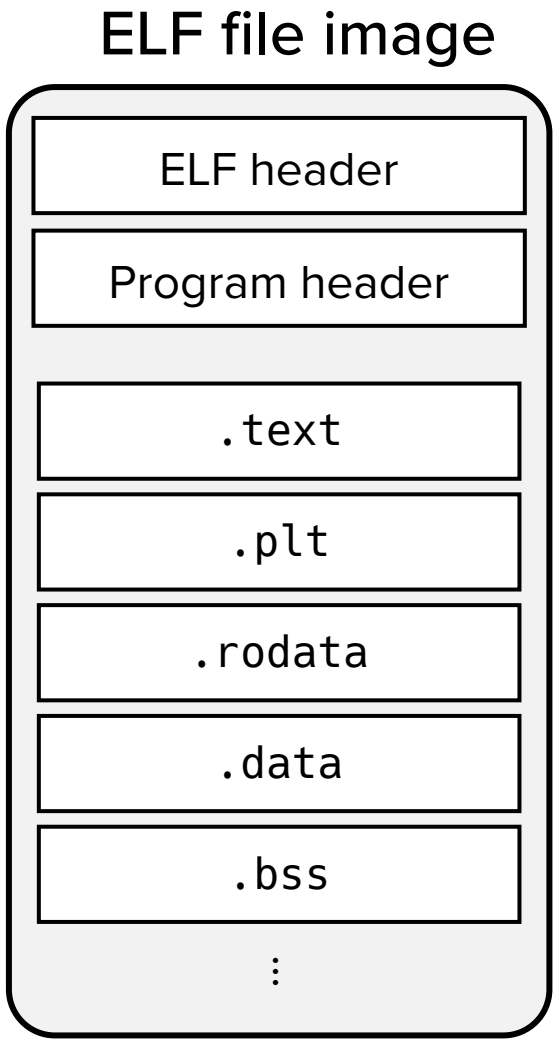
ARY

Recap: Compiler 101



Executable binary (on Linux)

- 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

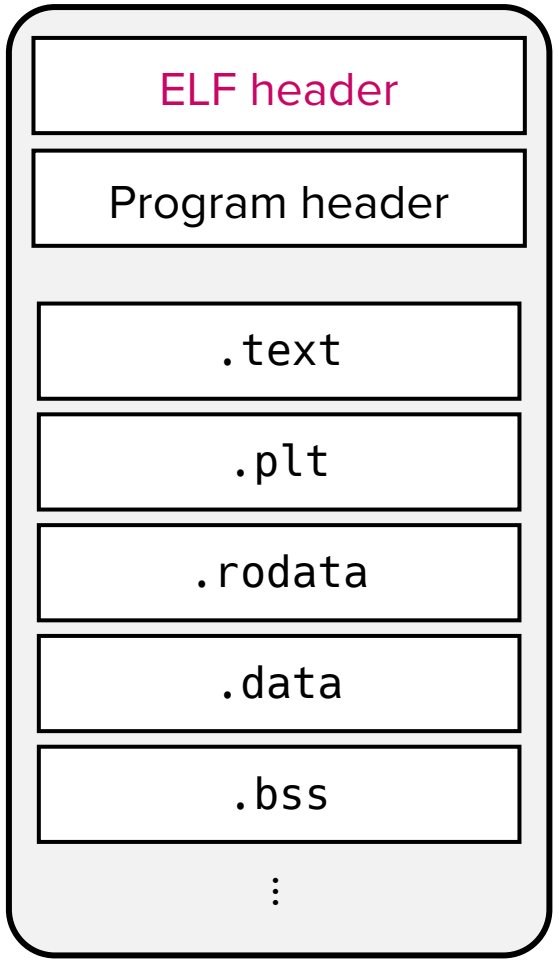


Executable binary (on Linux)

- 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.

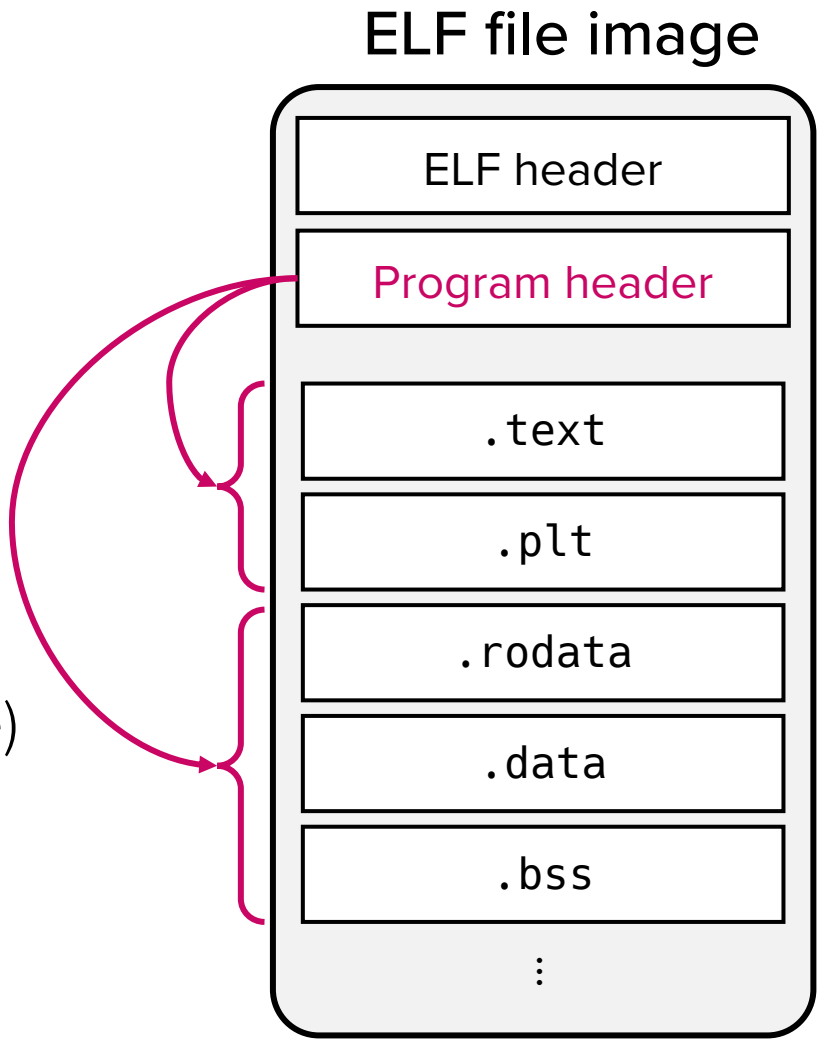
```
csed415-lab01@csed415:~$ readelf -h ./target
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x4011f0
  Start of program headers:          64 (bytes into file)
  Start of section headers:          277072 (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



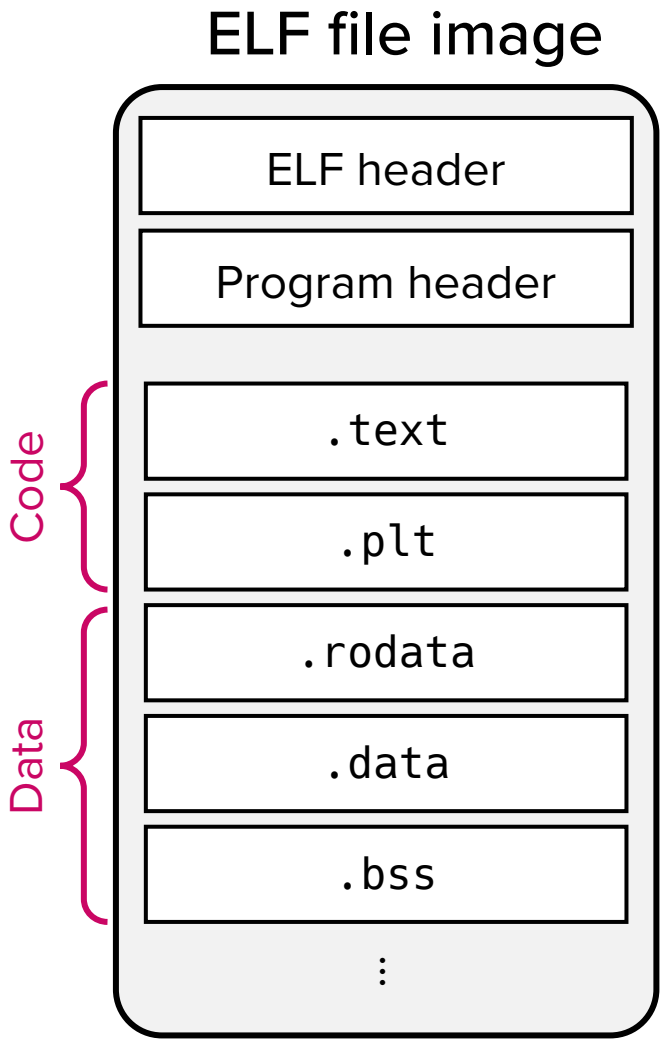
Executable binary (on Linux)

- 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)



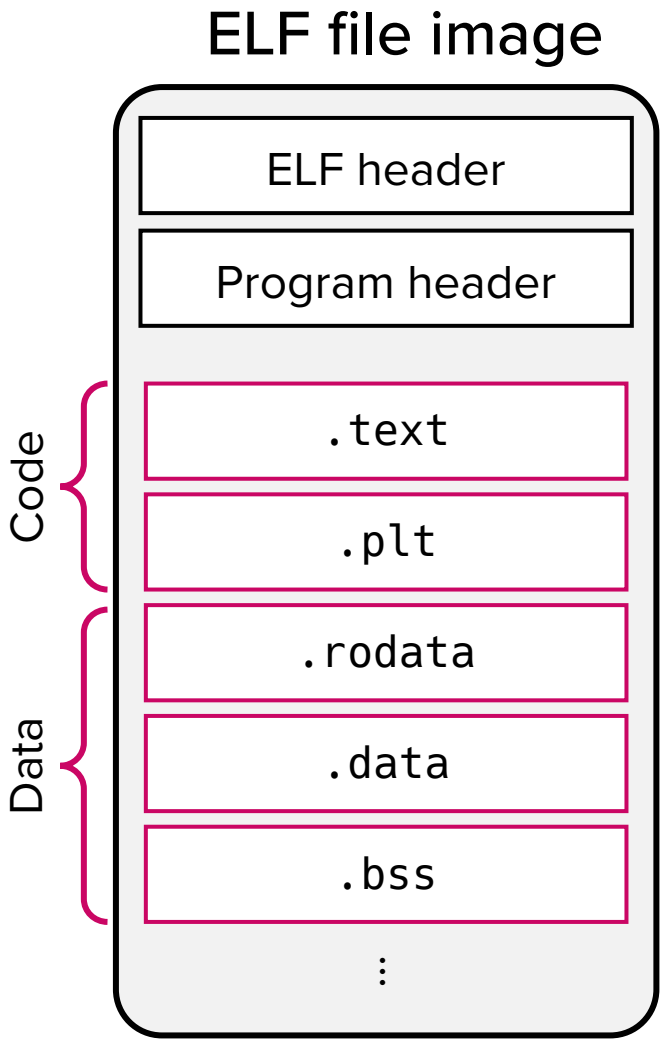
Executable binary (on Linux)

- 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`, …)

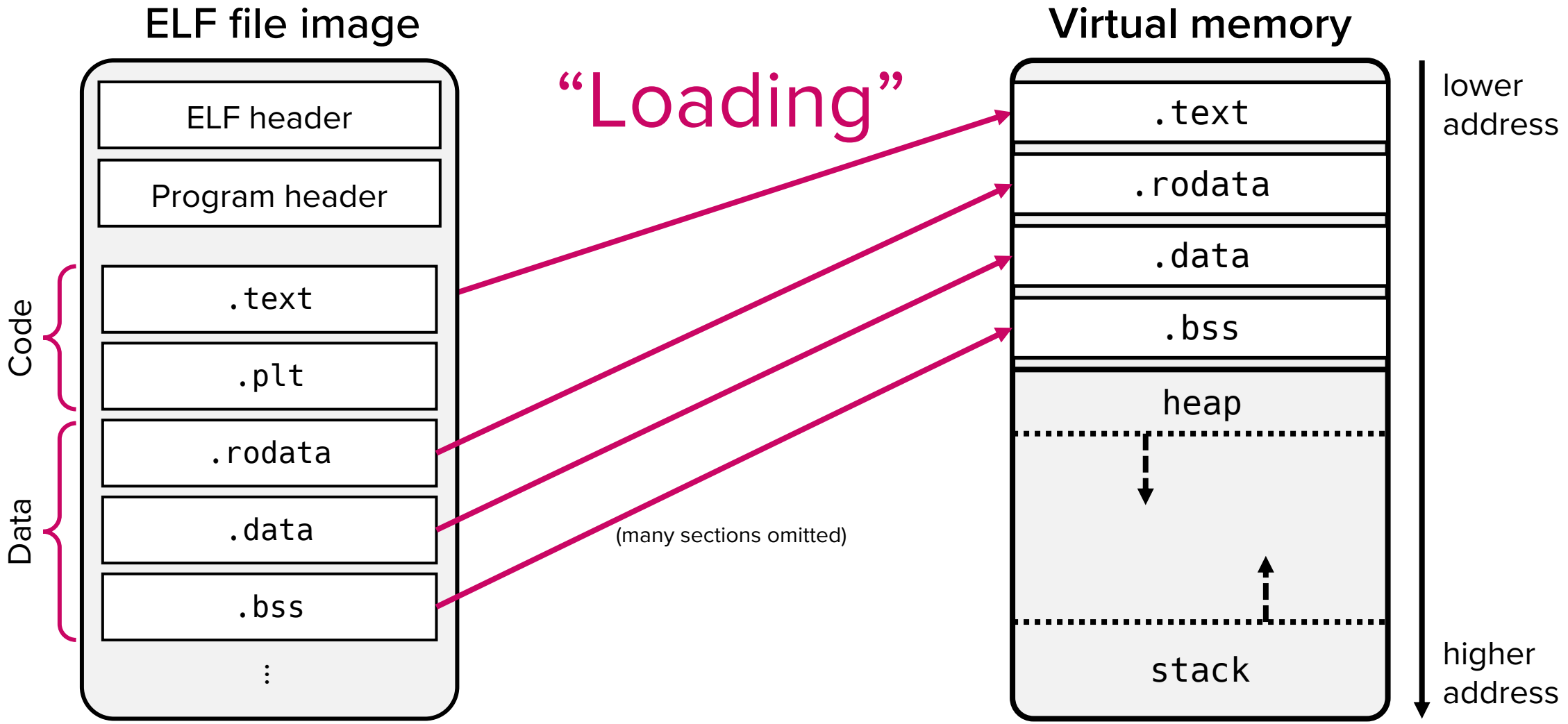


Executable binary (on Linux)

- Sections: Smallest unit of an object
  - `.text`: executable instructions
    - e.g., `add rsp, 8`
  - `.data`: initialized data
    - e.g., `uint32_t password = 0xdeadbeef;`
  - `.bss`: uninitialized data
    - e.g., `char global_array[32];`
  - `.rodata`: read-only data
    - e.g., `char *err_s = "Failed to open file";`

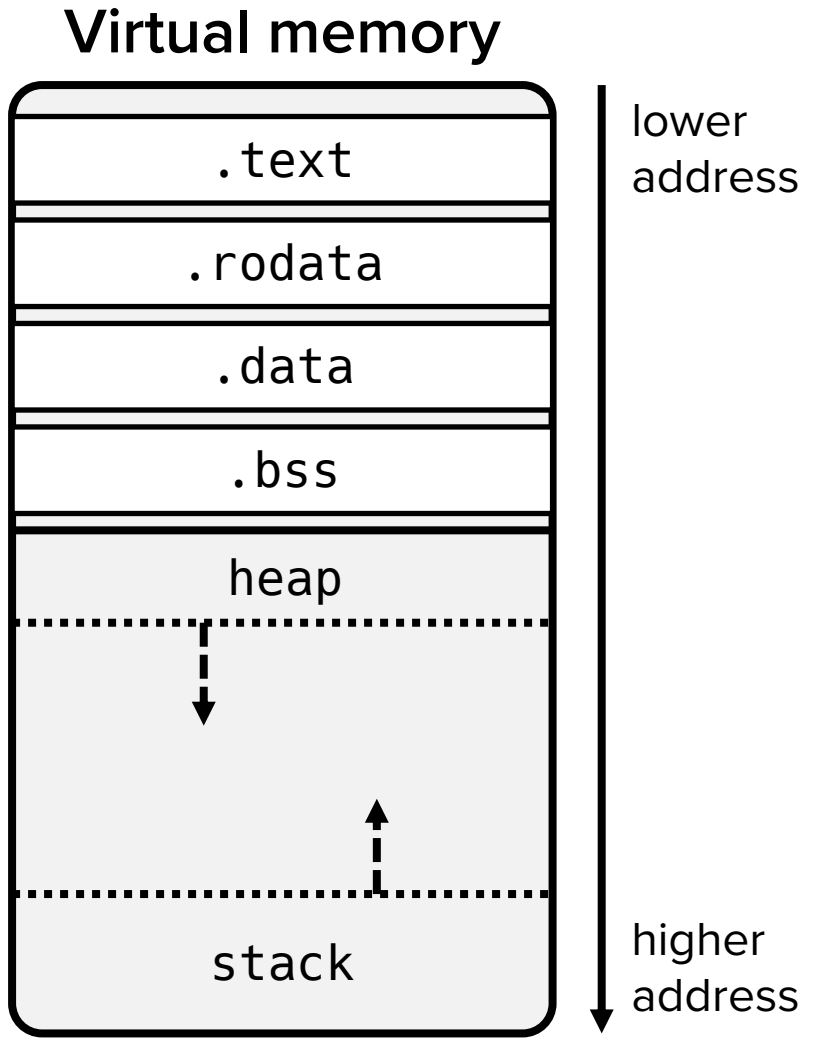


When you run a ELF executable, loader turns the binary image into a process



A process in memory

- Process = 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

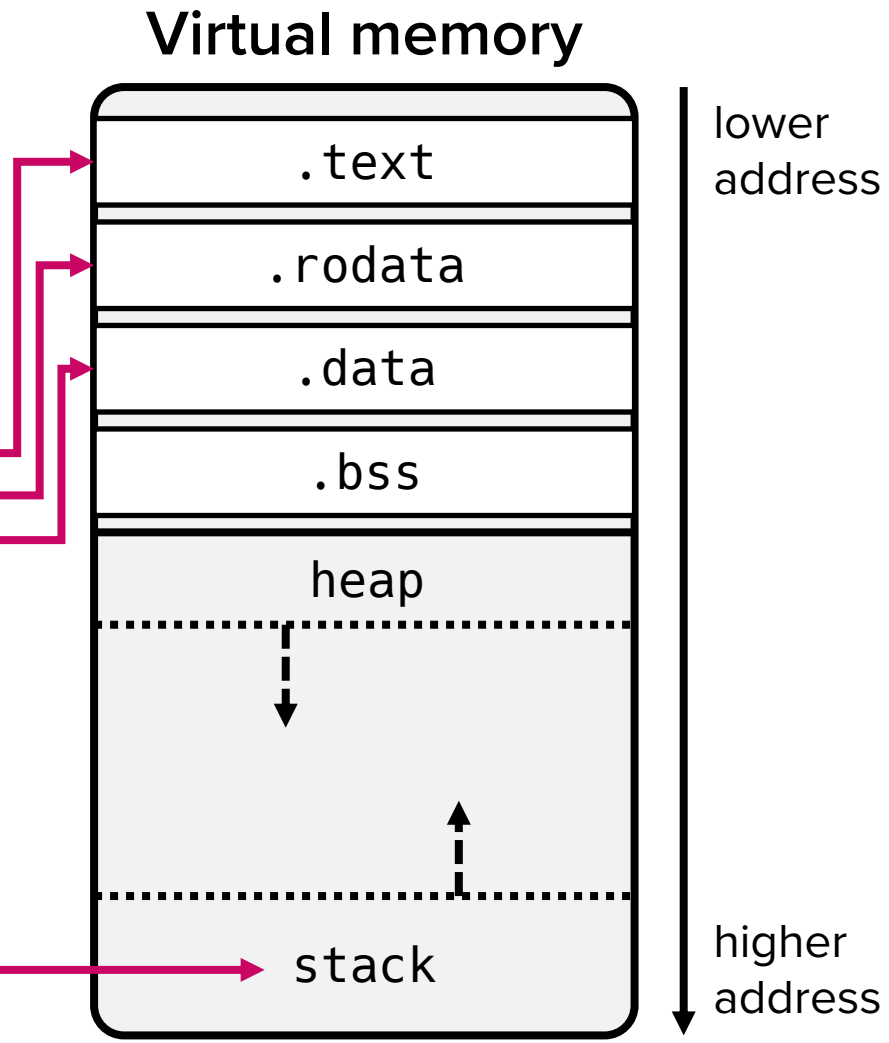


Example: Virtual memory of Lab 01's target

POSTECH

```
$ ssh csed415-lab01@141.223.124.55
$ gdb ./target
pwndbg> break *main
pwndbg> r
pwndbg> vmmap
```

```
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]
```

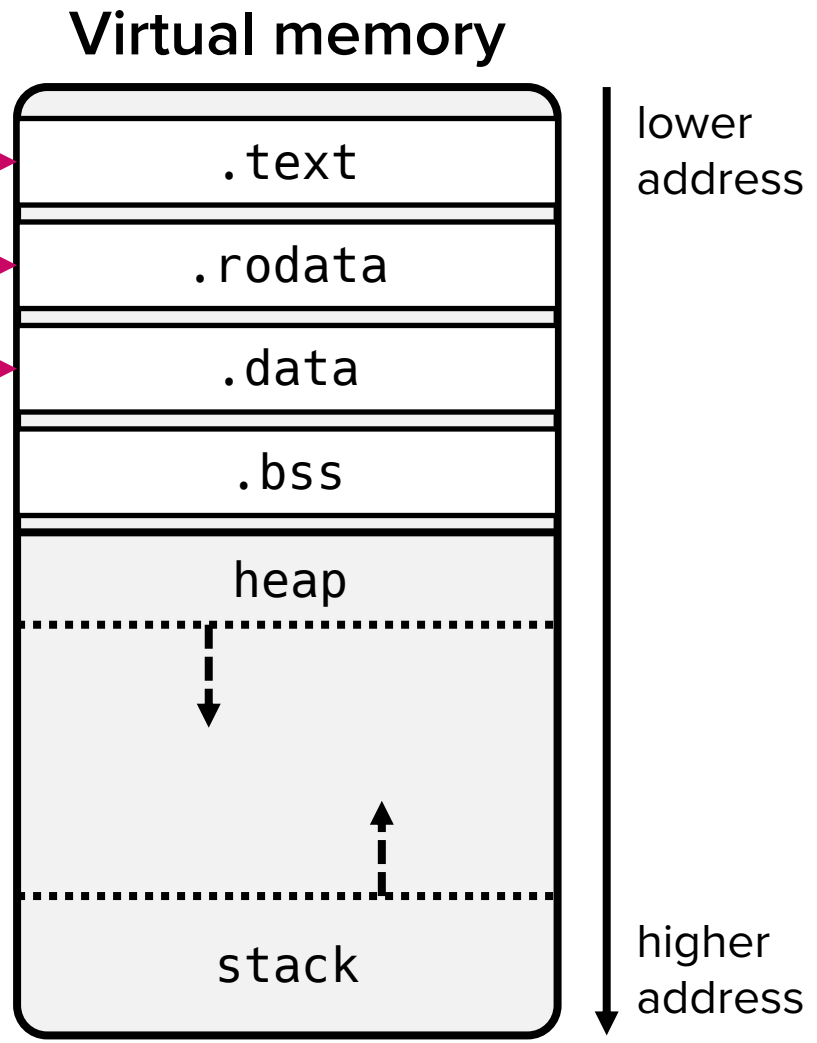


Example: Virtual memory of Lab 01's target

POSTECH

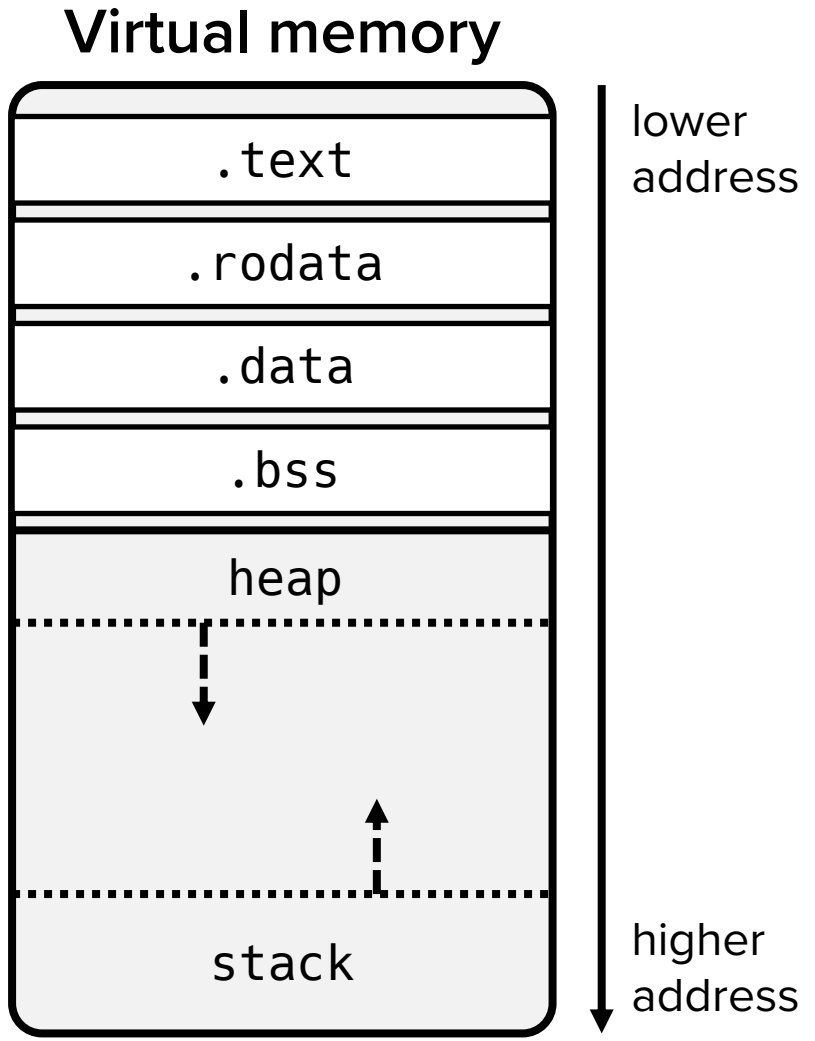
```
pwndbg> elfsections // check section info
pwndbg> x/20i 0x401000 // examine instructions in .text
pwndbg> x/40s 0x402000 // examine strings in .rodata
pwndbg> x/100wx 0x404000 // examine data in .data
```

```
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]
```



Binary analysis?

- 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



Binary analysis?

POSTECH

- Q1) Which component of a computer executes instructions? → CPU
- Q2) How does that component keep track of which instruction
- Q3) How does that component keep track of program execution states?

  → Using general purpose registers

If we can observe CPU register states and associated sections, we can understand the binary's runtime behavior!

x86-64

x86-64 architecture

- Our focus: x86-64 architecture
  - Also known as x64 architecture
  - The most widely used architecture for PCs and servers
    - Intel – Core i3/i5/i7/i9 families, Xeon server processors
    - AMD – Ryzen processors, EPYC server processors
  - In 2022, x86-64 architecture held 90% of the PC market and 85% of the server market (Mercury Research)

→ Most exploits are developed targeting x86-64 systems

Central Processing Unit (CPU)

- CPU's core capability: Carry out a fixed number of operations
- Terminology
  - Instruction: One operation that a CPU can do
    - e.g., add two numeric values
  - Instruction set: The set of operations that a CPU is designed to do
    - Instruction Set Architecture (ISA): Name for the set of operations
    - x64 processors understand and run x64 instructions!
  - Assembly language: A human-readable representation of instructions
    - e.g., `add rsp,8` → Add `8` to the current value in `rsp` and store the result in `rsp`

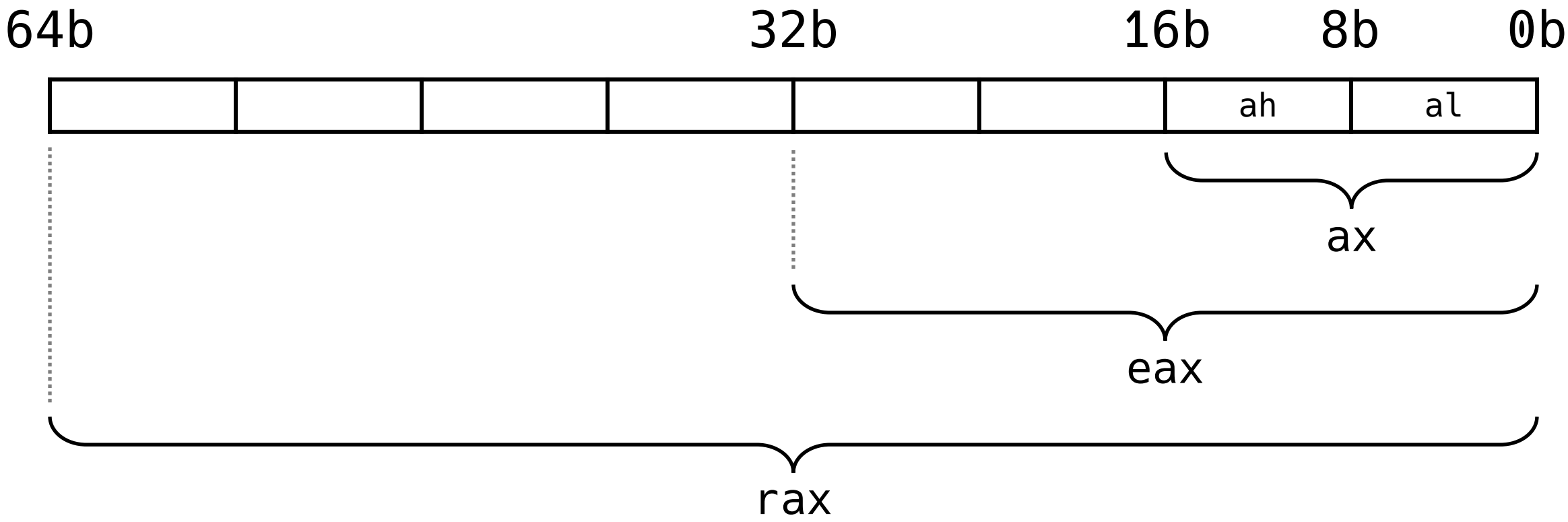
x64 registers

- General purpose registers (GPRs)
  - `rax`, `rbx`, `rcx`, `rdx`, `rsi`, `rdi`
  - Additional: `r8`, `r9` , `...`, `r15`
  - Stack-related (more on these later):
    - `rbp`: Base pointer
    - `rsp`: Stack pointer

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 (instructions) exists
- Flag register
  - `rflags`: Stores status flags (carry flag, sign flag, …)
- Segment registers (Legacy)
  - `cs, ss, ds, es, fs, gs`
  - Used for memory addressing (legacy feature, not used in x64)

Naming convention for x64 registers



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

x86-64 Assembly Basics

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 is the destination register
  - Stores the results
- Operand 2 is the source register

```
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: SIZE PTR

POSTECH

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

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

or

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

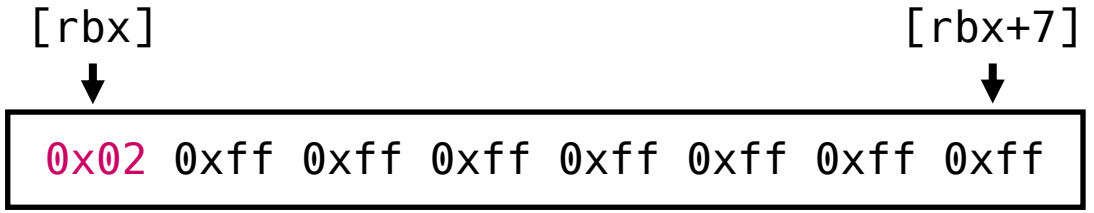
or

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

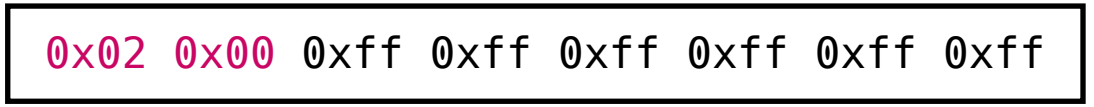
or

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

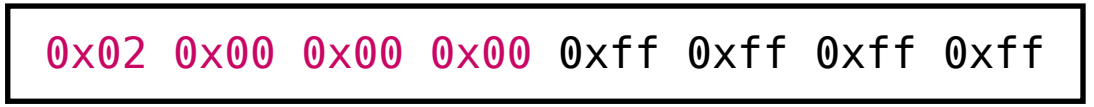
Byte at the addr rbx holds



Word at the addr rbx holds



Double word at the addr rbx holds



Quad word at the addr rbx holds

Note: Endianness (Order of numbers in mem)


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

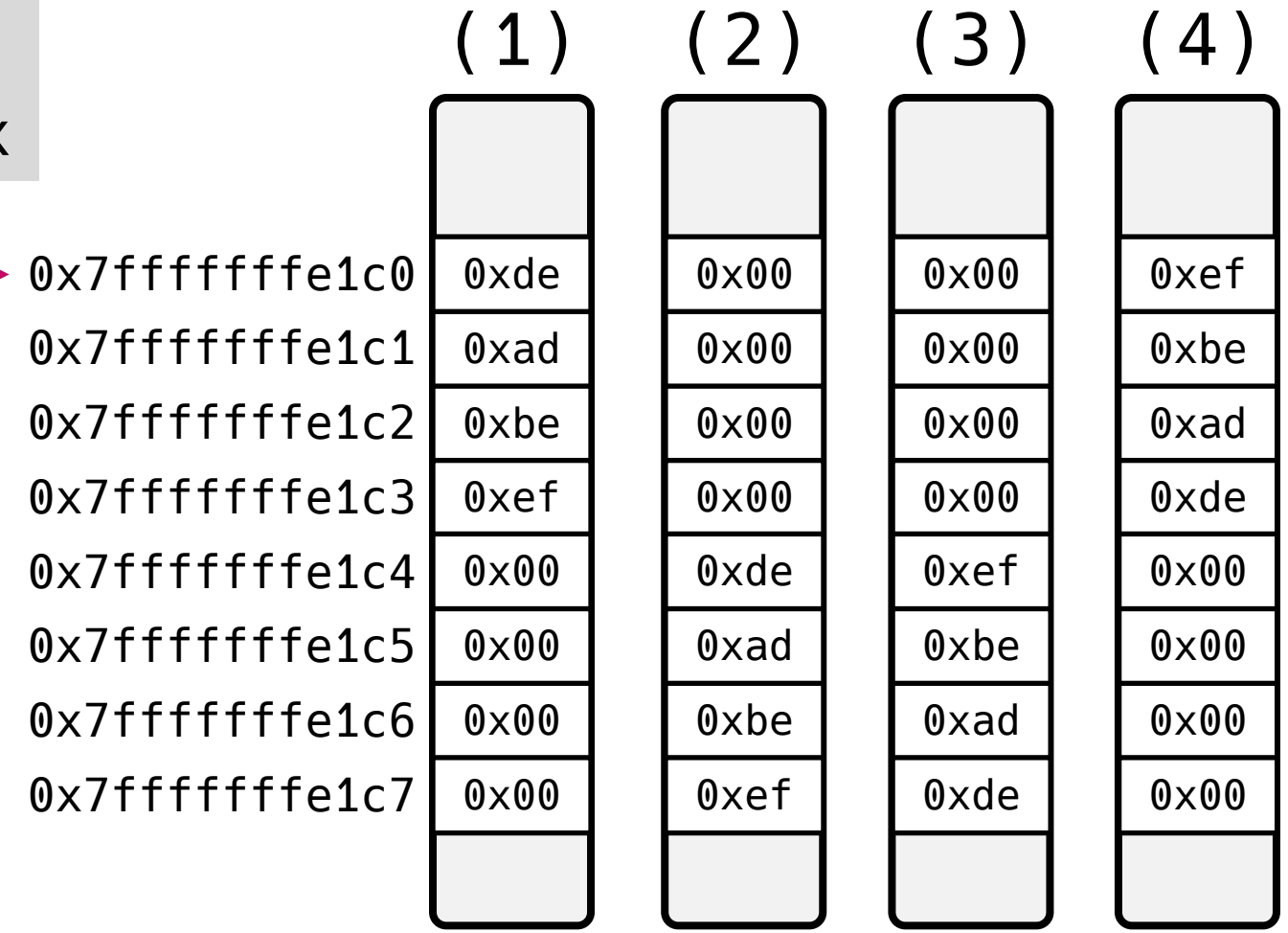
- Little-endian of x64:
- The least-significant byte (LSB) of multi-byte numbers is placed at the lowest memory address

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

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

MSB: `0x00` LSB: `0xef`

Q) Which layout is correct?



Note: Endianness (Order of numbers in mem)

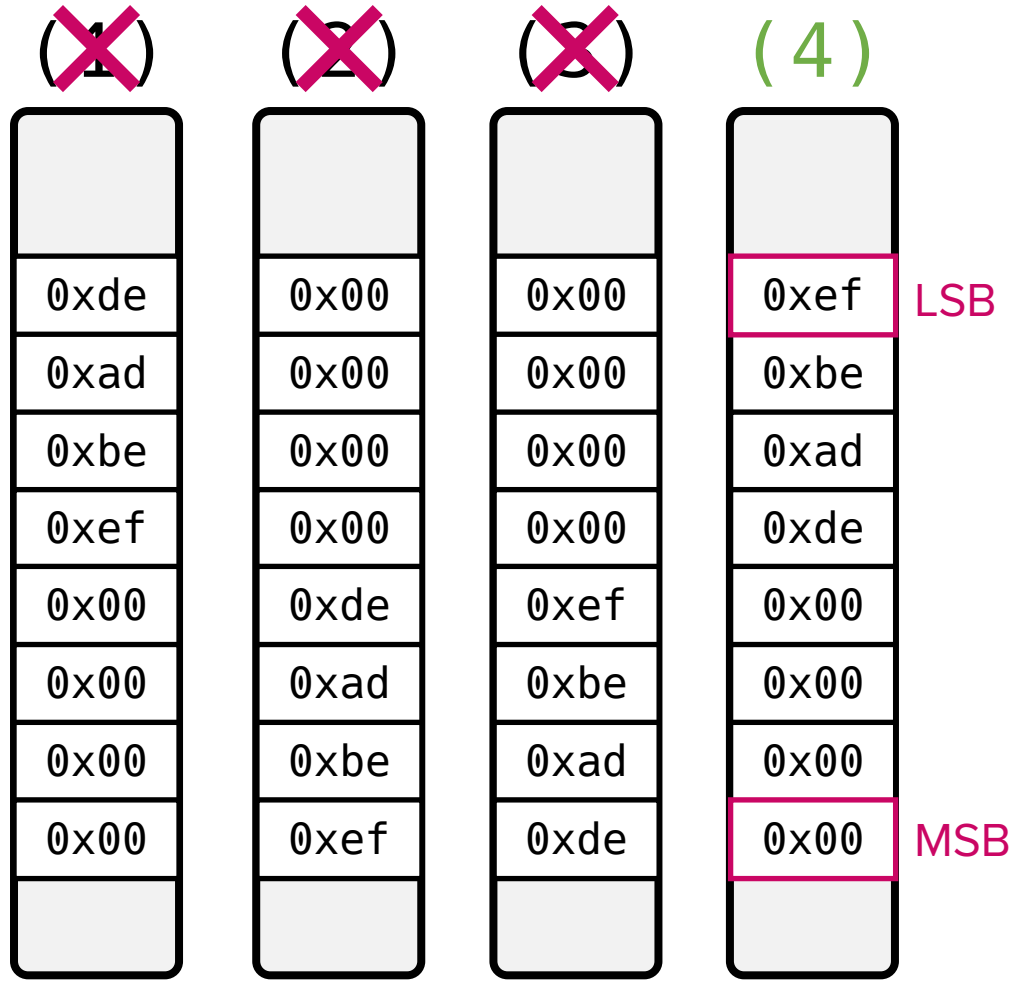

```
mov   rax, 0x7fffffffe1c0
mov   rdx, 0xdeadbeef
mov   QWORD PTR [rax], rdx
```

- Little-endian of x64:
- The least-significant byte (LSB) of multi-byte numbers is placed at the lowest memory address

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

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

MSB: `0x00` LSB: `0xef`



`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]`

Q) `rax = ?`

A) `rax = 0x555555555ff8`

→ The address (not the value) is loaded

mov vs. lea

POSTECH

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
```

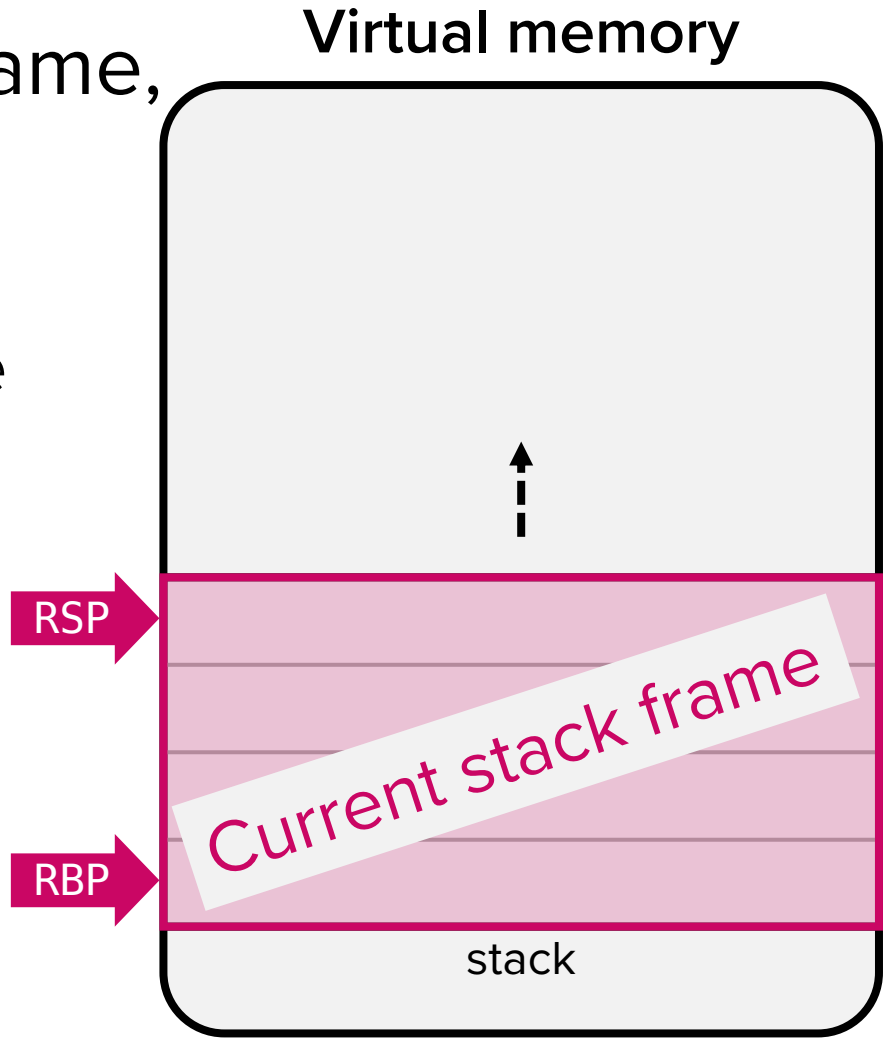
Stack Layout

Stack frames

- When your code calls a function, space is reserved on the stack for local variables
  - This space is known as the **stack frame** for the function
  - The stack frame is removed (forgotten) once the function returns
- The stack starts at higher addresses
  - Every time your code calls a function, the stack reserves extra space by growing up (towards the lower addresses)

Stack frames

- To keep track of the current stack frame, we store two pointers in registers
  - The `rbp` (base pointer) register points to the bottom of the current stack frame
  - The `rsp` (stack pointer) register points to the top of the current stack frame



Stack operations: push

- push enlarges the stack

```
push   rax
```

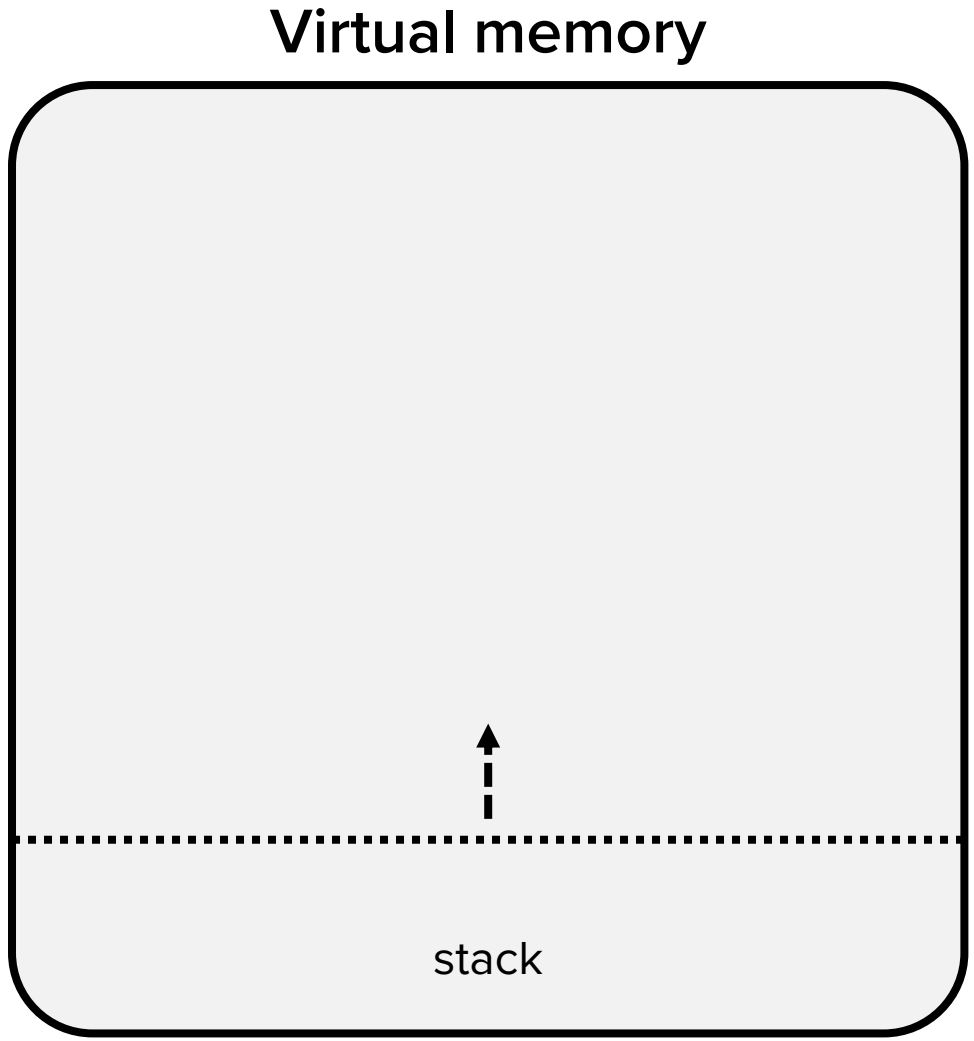
Push the value rax holds

```
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
```

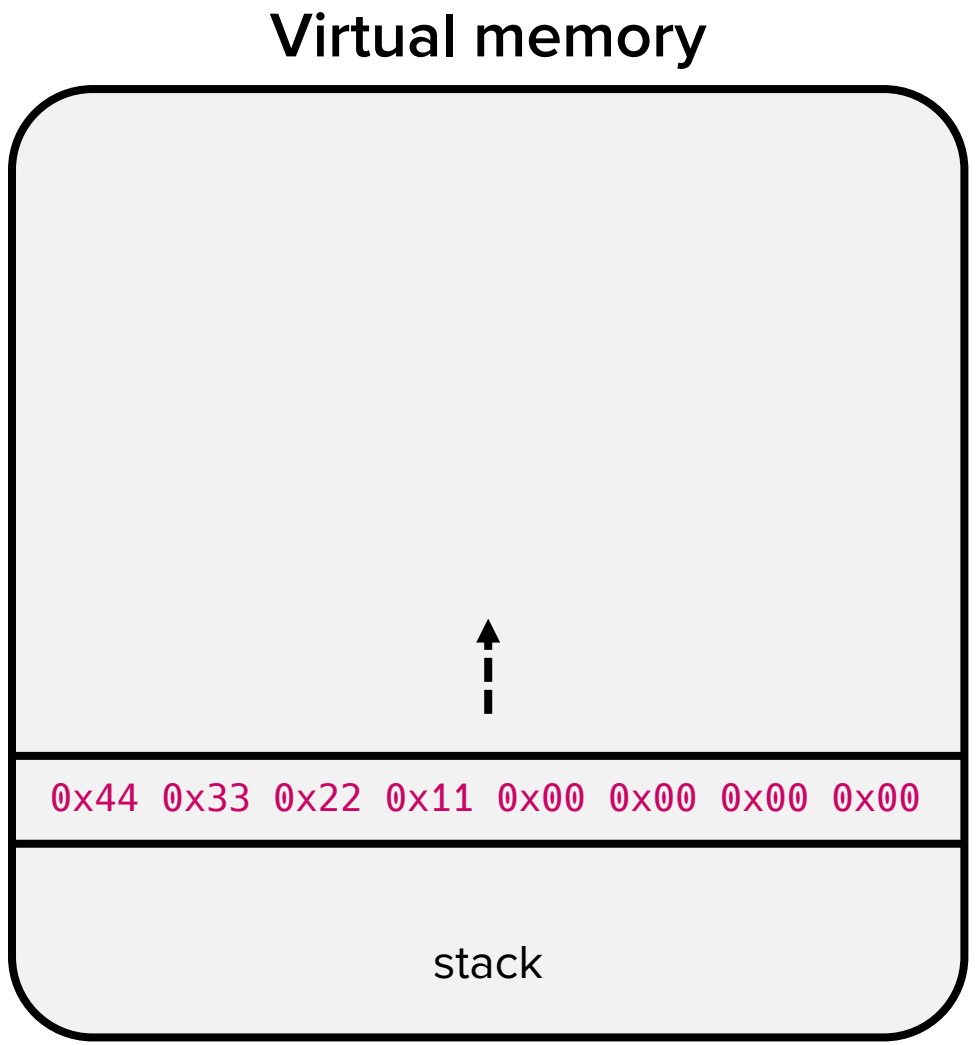
Push the value rax holds

```
push   [rax]
```

Push the value at the address rax points to

```
push   0x11223344
```

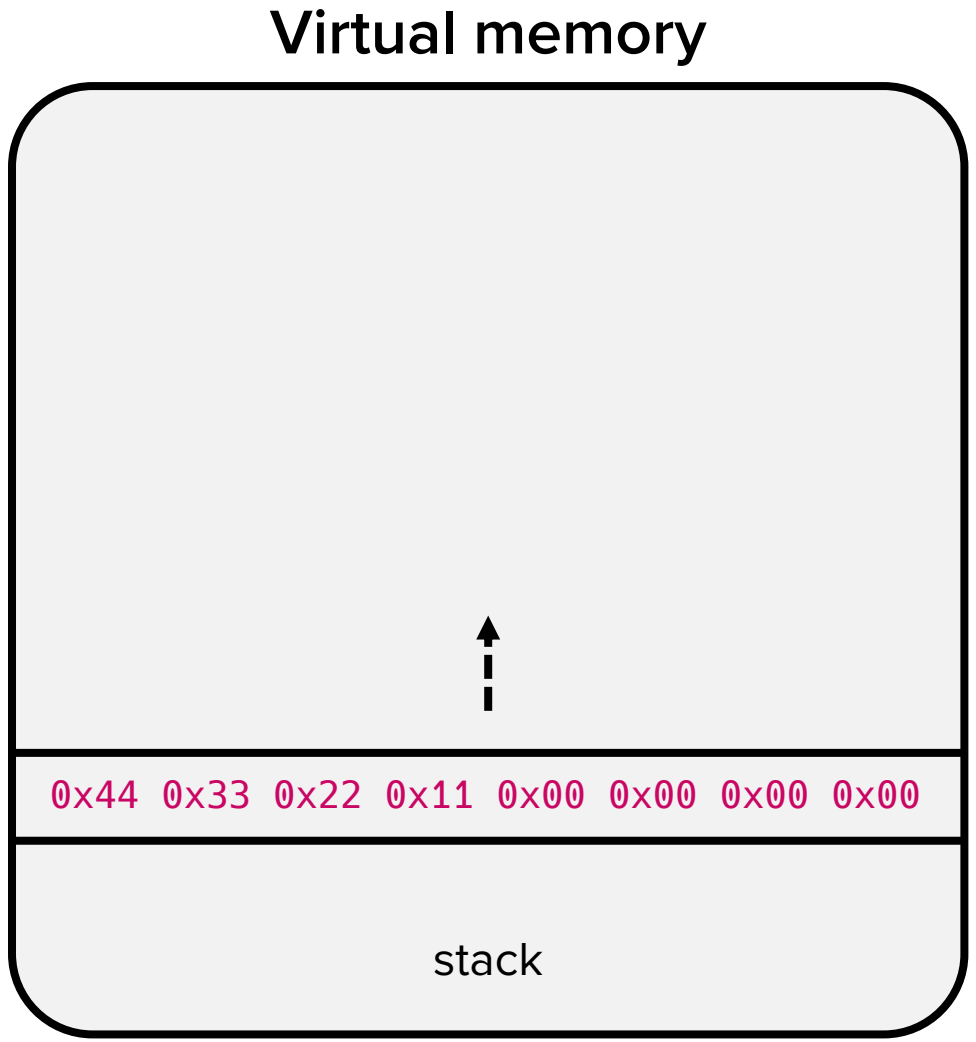
Push constant value



Stack operations: push

- push equivalence

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



Stack operations: pop

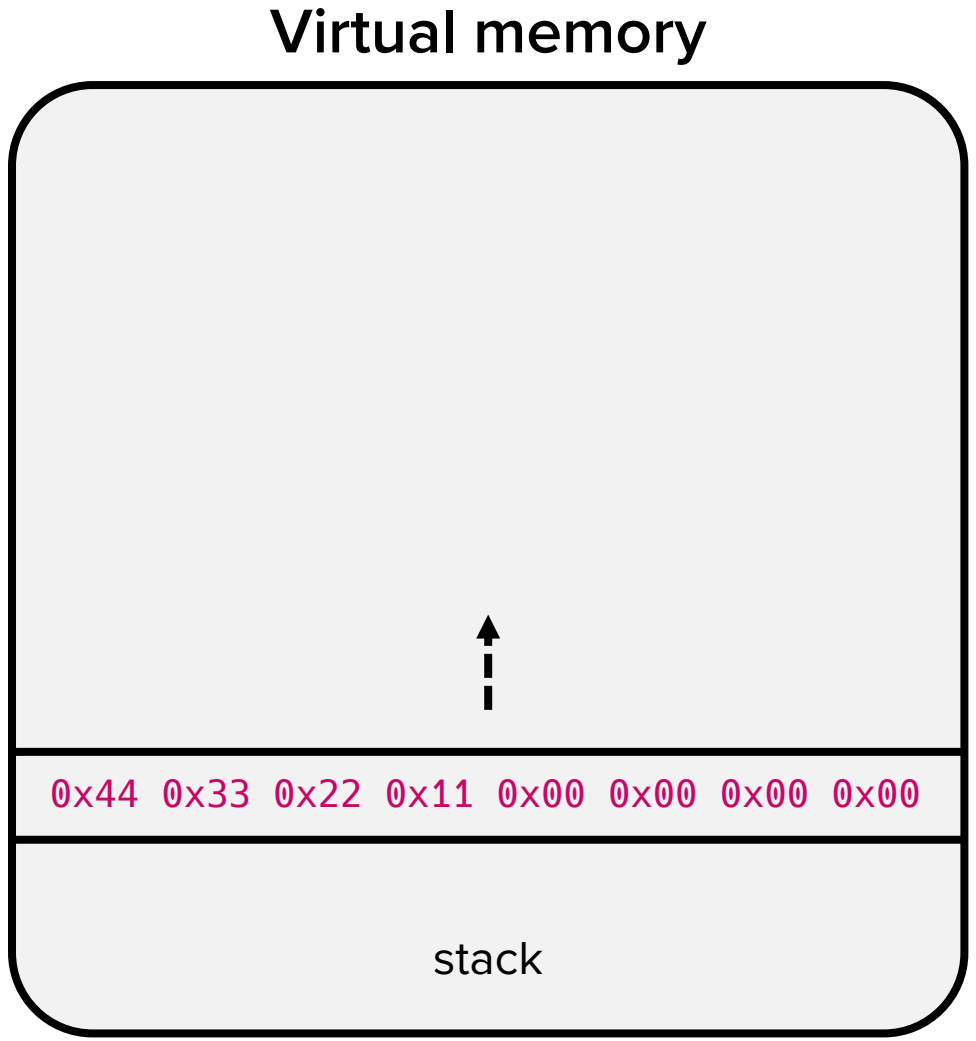
- pop shrinks the stack

```
pop   rax
```

Pop the stack top into rax

```
pop   [rax]
```

Pop the stack top into the memory pointed to by rax



Stack operations: pop

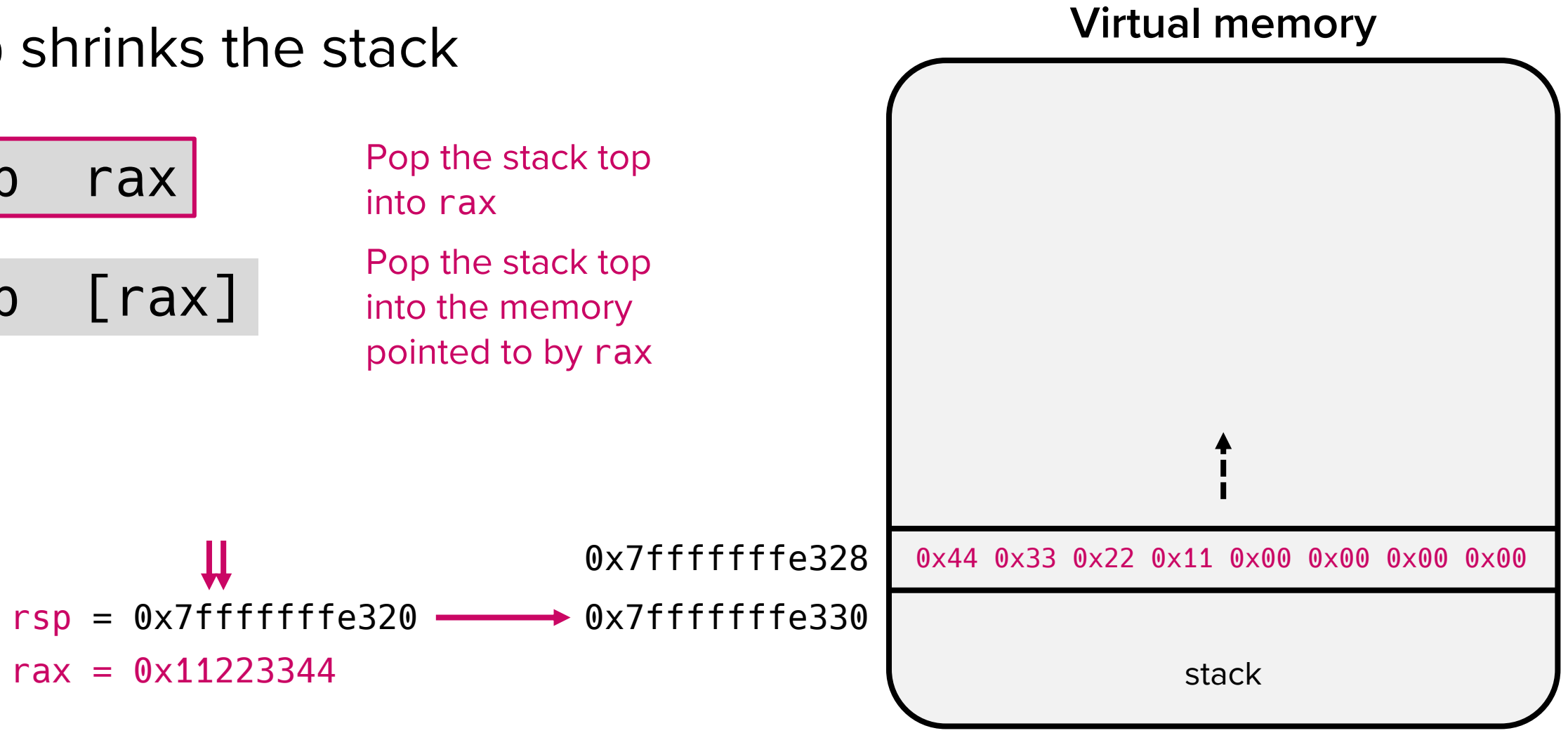
- pop shrinks the stack

```
pop   rax
```

Pop the stack top into rax

```
pop   [rax]
```

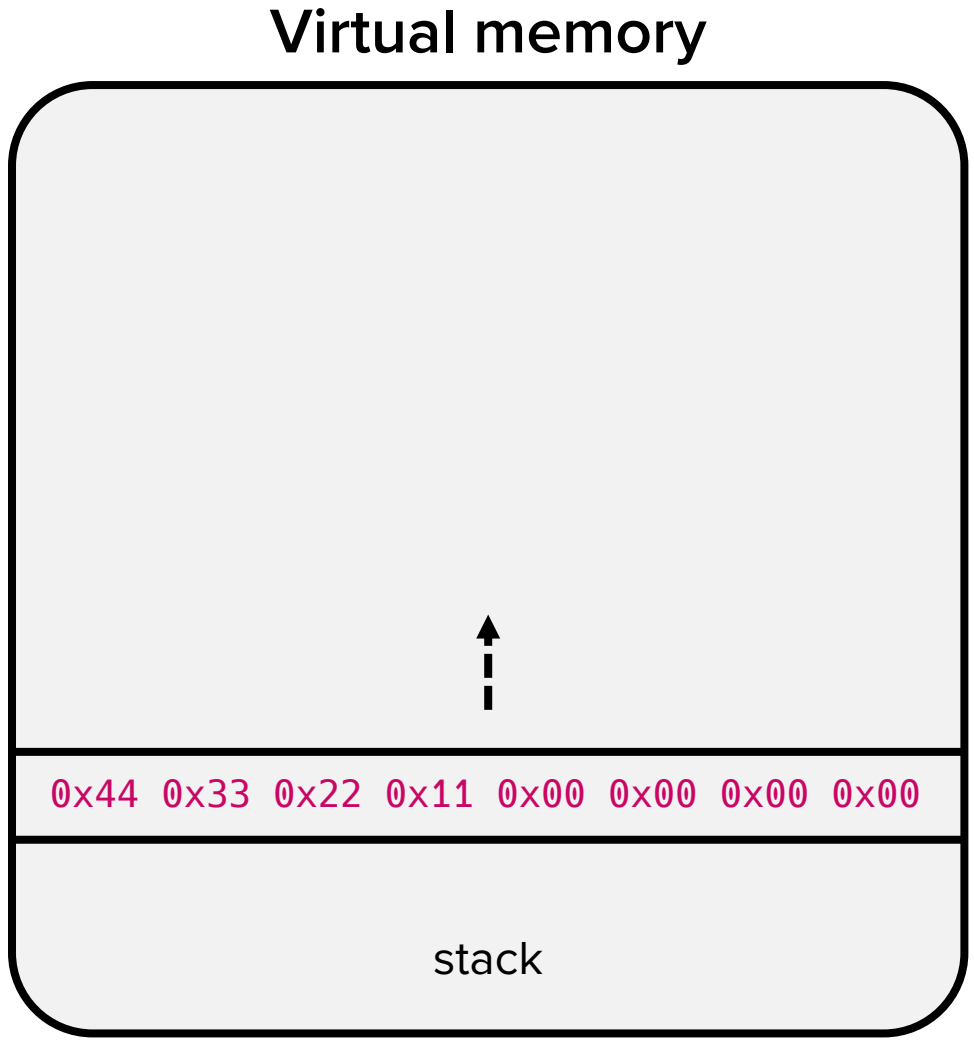
Pop the stack top into the memory pointed to by rax



Stack operations: pop

- pop equivalence

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



Function Calling Convention

Function calls

- Function call
  - The caller function (**`main`**) calls the callee function (**`foo`**)
  - The callee function executes and then returns control to the caller

Before function call

```
int main() {
  int a = 1;
  foo();
  return 0;
}
```

Caller

During function call

```
int foo() {
  int b = 0;
  return;
}
```

Callee

After function call

```
int main() {
  int a = 1;
  foo();
  return 0;
}
```

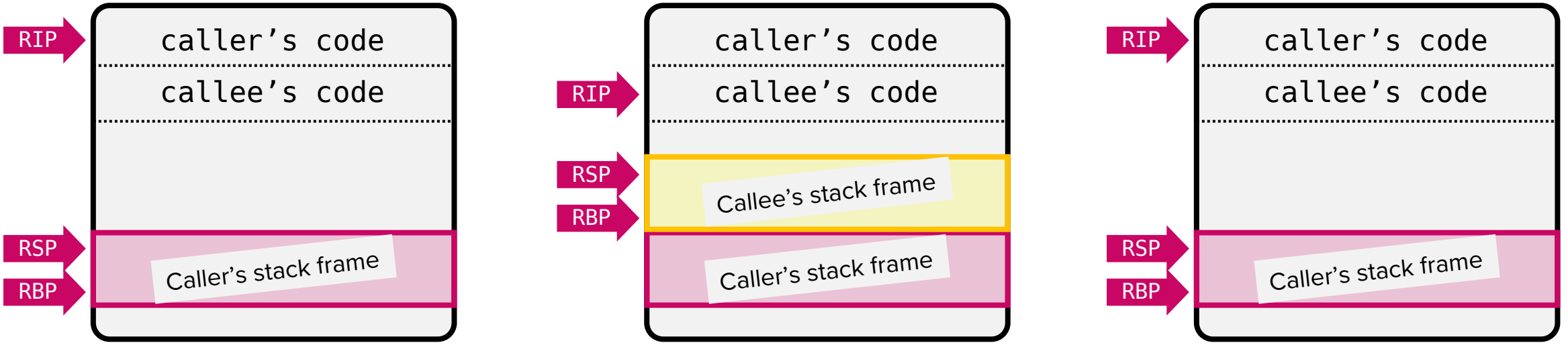
Caller

x86-64 calling convention

- Calling convention:
  A contract that defines how functions call each other
  - Passing arguments
    - Caller moves first six arguments into `rdi`, `rsi`, `rdx`, `rcx`, `r8`, and `r9`
    - Caller pushes additional arguments (if any) onto the stack
  - Receiving return values
    - Callee stores its return value in `rax` register

Function calling and stack frames

- When calling a function, `rsp` and `rbp` need to shift to create a new stack frame, and `rip` must move to the callee's code
- When returning to the caller function, `rsp`, `rbp`, and `rip` must be restored to their old values



Steps of a function call

1. Move arguments #1-#6 to registers
2. Push arguments #7- onto the stack
3. Push old `rip` onto the stack
4. Update `rip`

Caller (`main`)

Updating `rip` transfers control from `main` to `foo`

5. Push old `rbp` onto the stack
6. Update `rbp`
7. Update `rsp`
8. Execute function code
9. Update `rsp`
10. Restore old `rbp`
11. Restore old `rip`

Callee (`foo`)

Restoring old `rip` transfers control back to `main`

12. Remove arguments #7- from the stack

Caller (`main`)

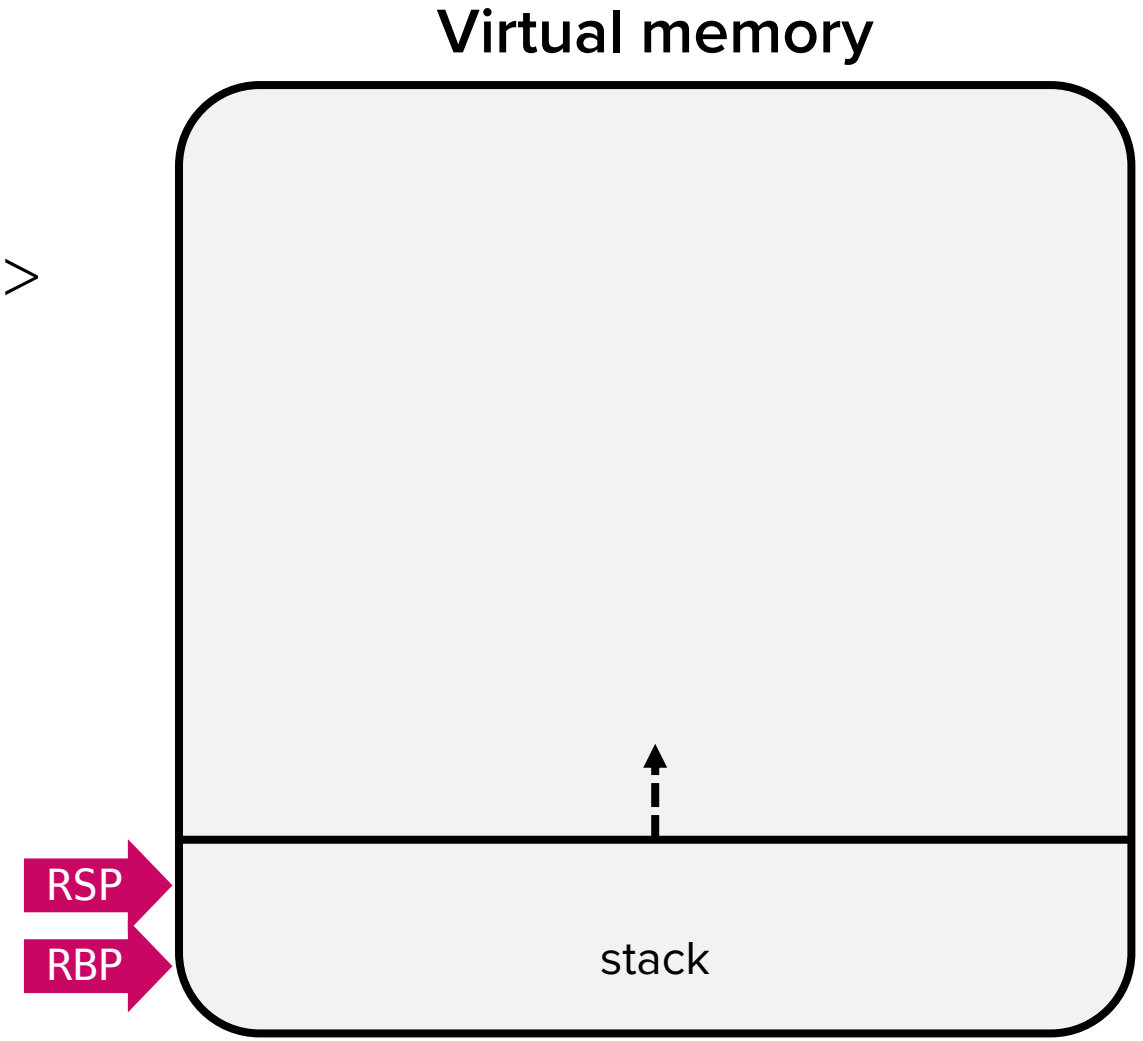
Subroutine instructions: `call`

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

RIP

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

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



Subroutine instructions: `call`

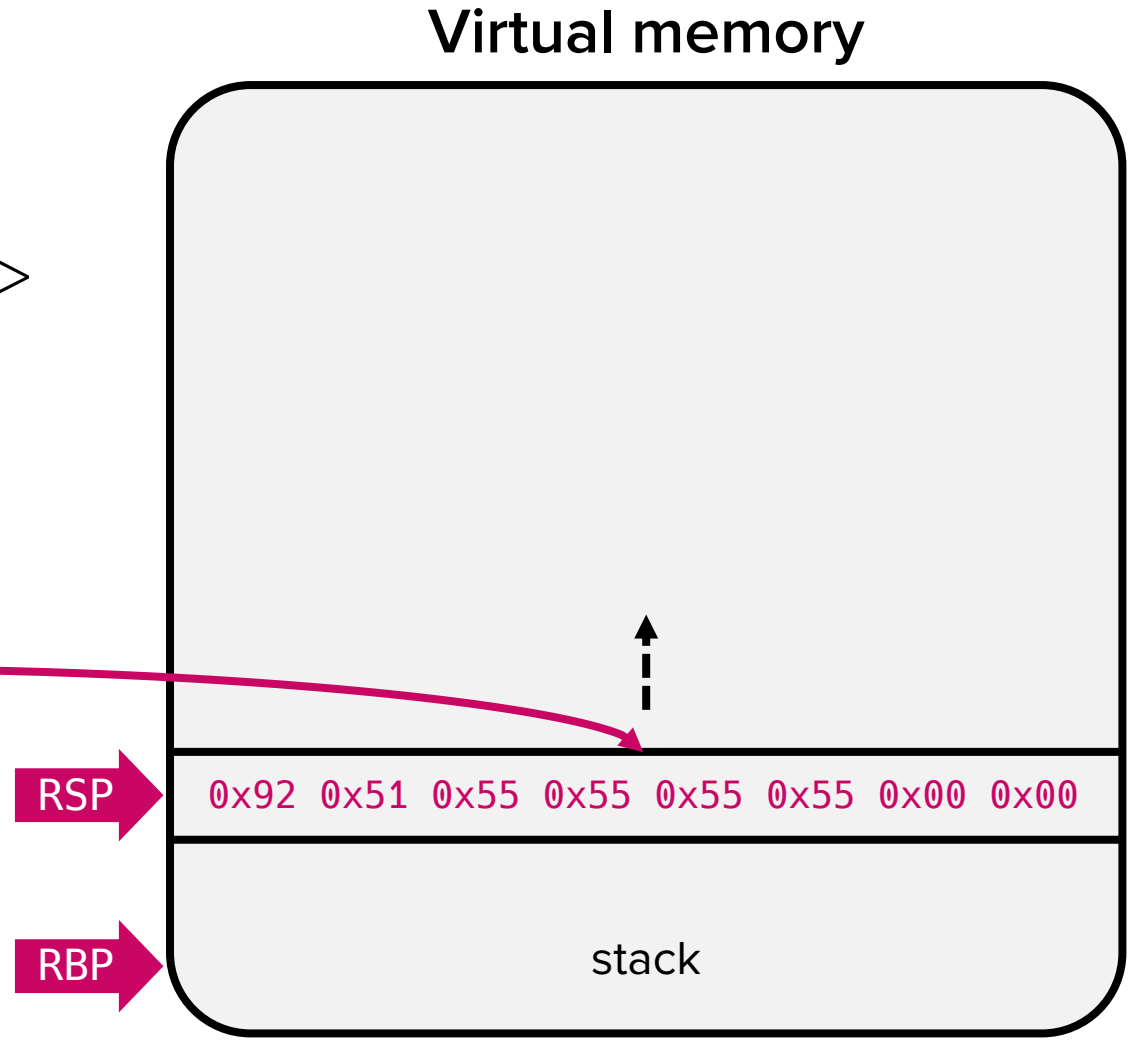
POSTECH

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

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

RIP

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



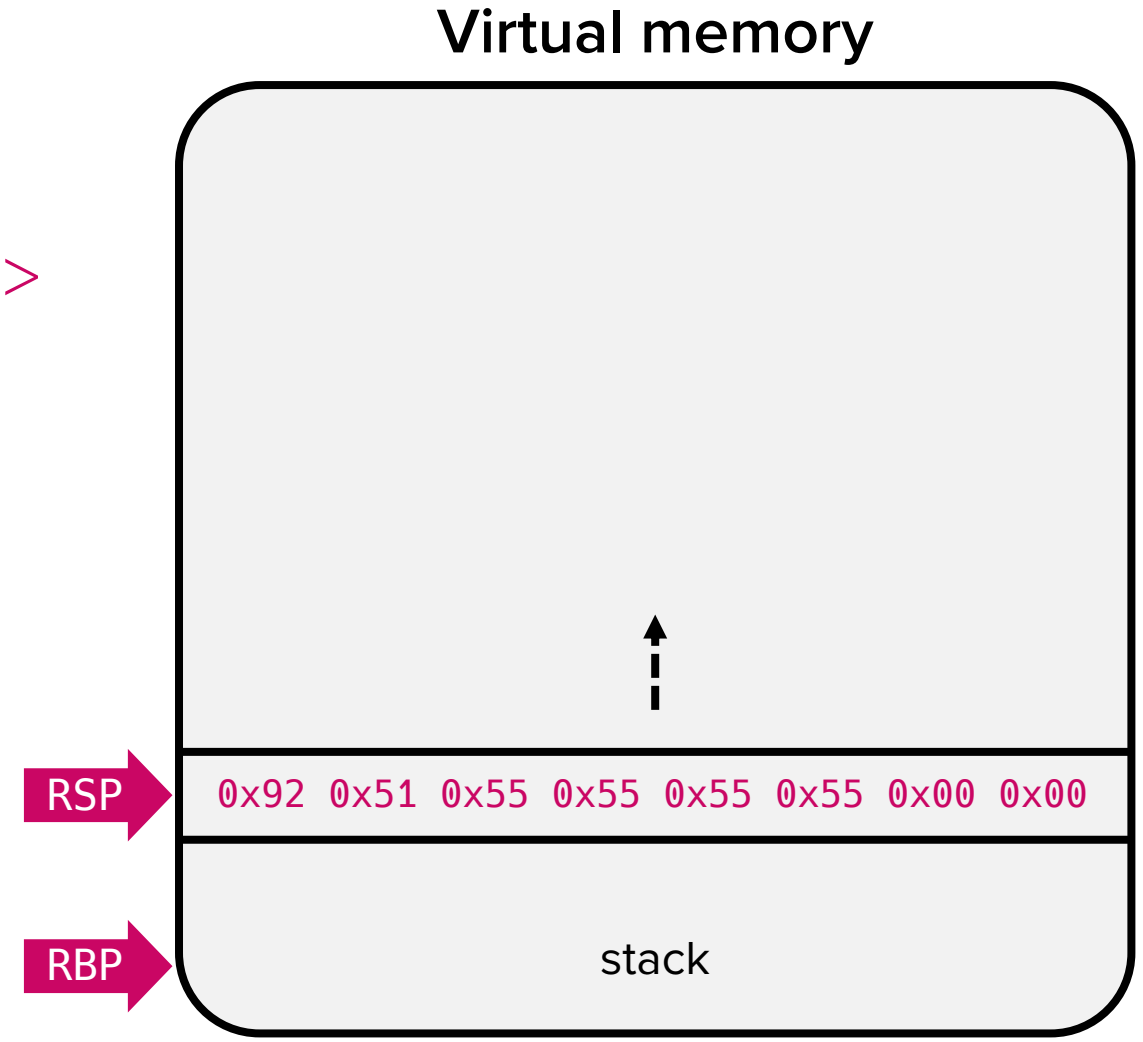
Subroutine instructions: `call`

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

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

RIP

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



Subroutine instructions: `call`

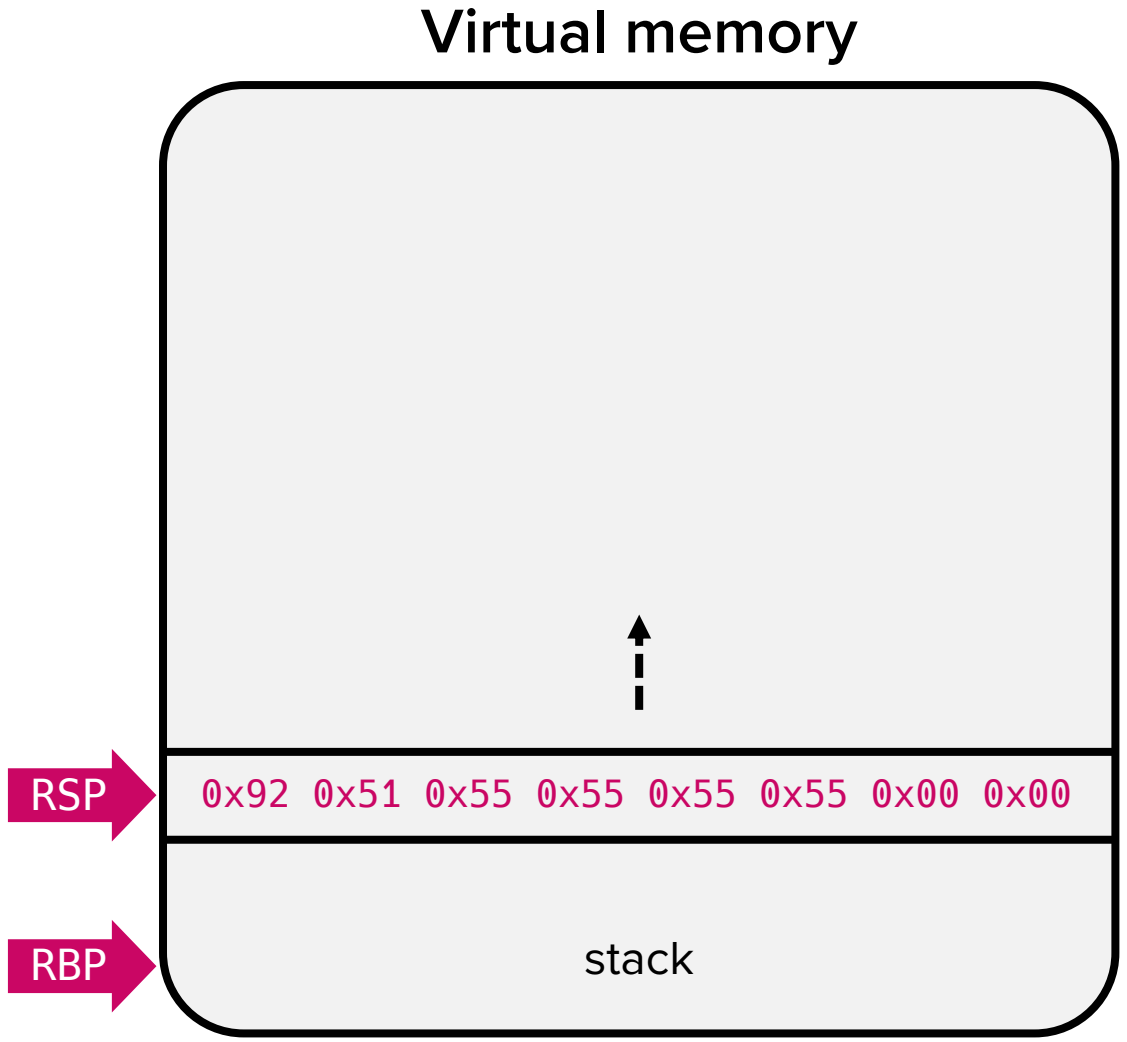
POSTECH

- `call` <label>
  - Callee function's prologue usually looks like: `push rbp; mov rbp, rsp`
  - e.g.,

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

RIP

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



Subroutine instructions: `call`

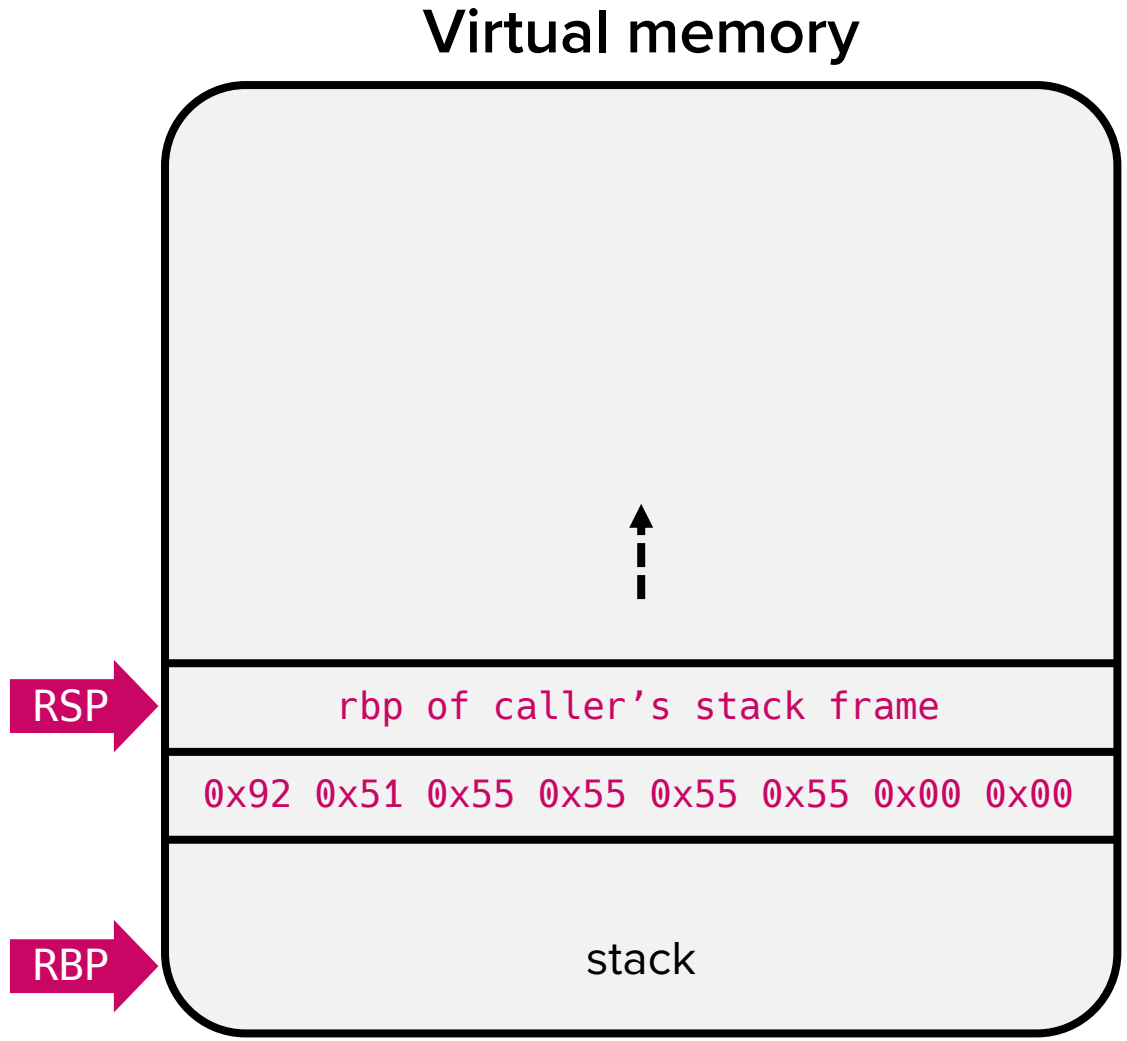
POSTECH

- `call` <label>
  - Callee function's prologue usually looks like: `push rbp; mov rbp, rsp`
  - e.g.,

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

RIP

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



Subroutine instructions: `call`

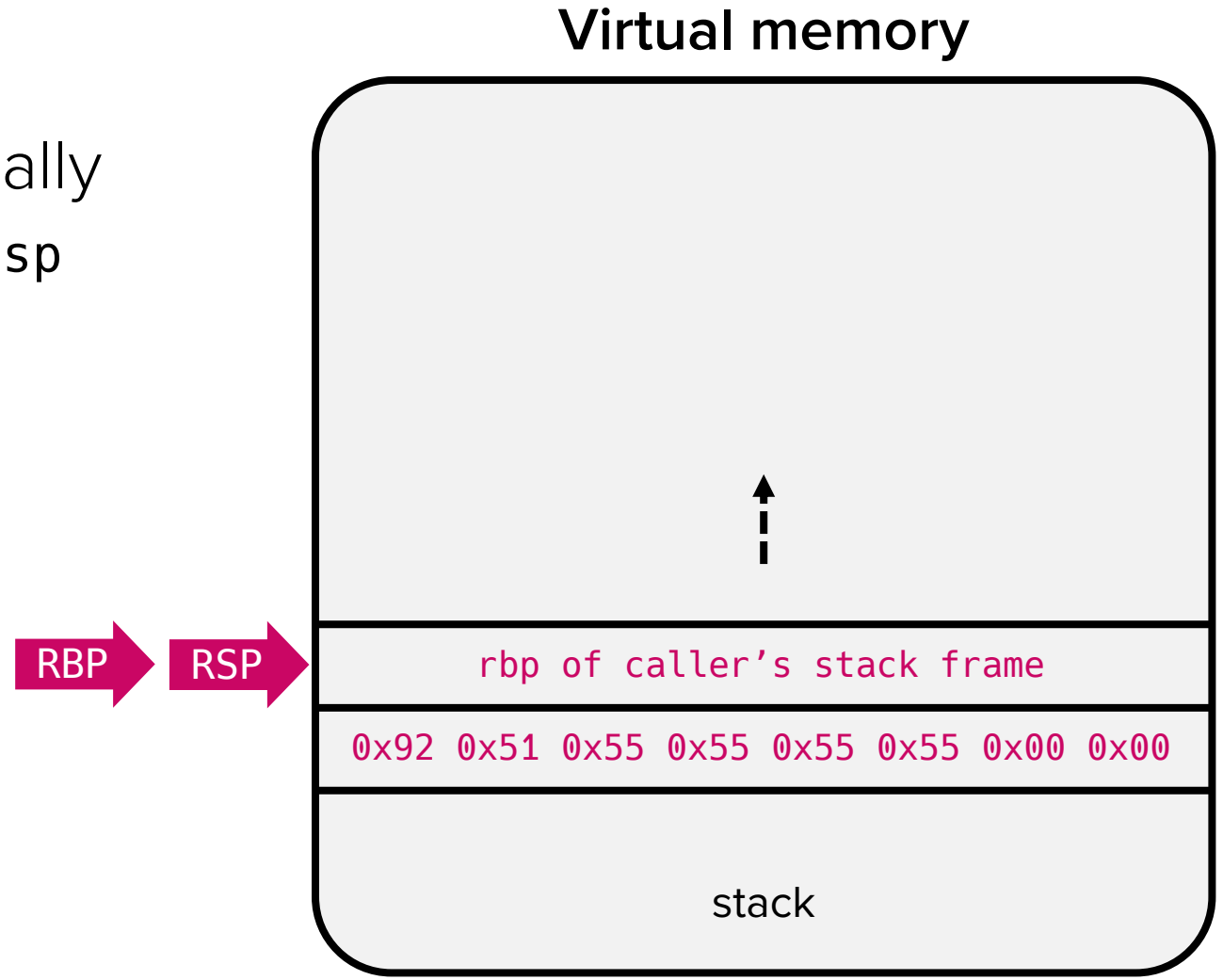
POSTECH

- `call` <label>
  - Callee function's prologue usually looks like: `push rbp; mov rbp, rsp`
  - e.g.,

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

RIP

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



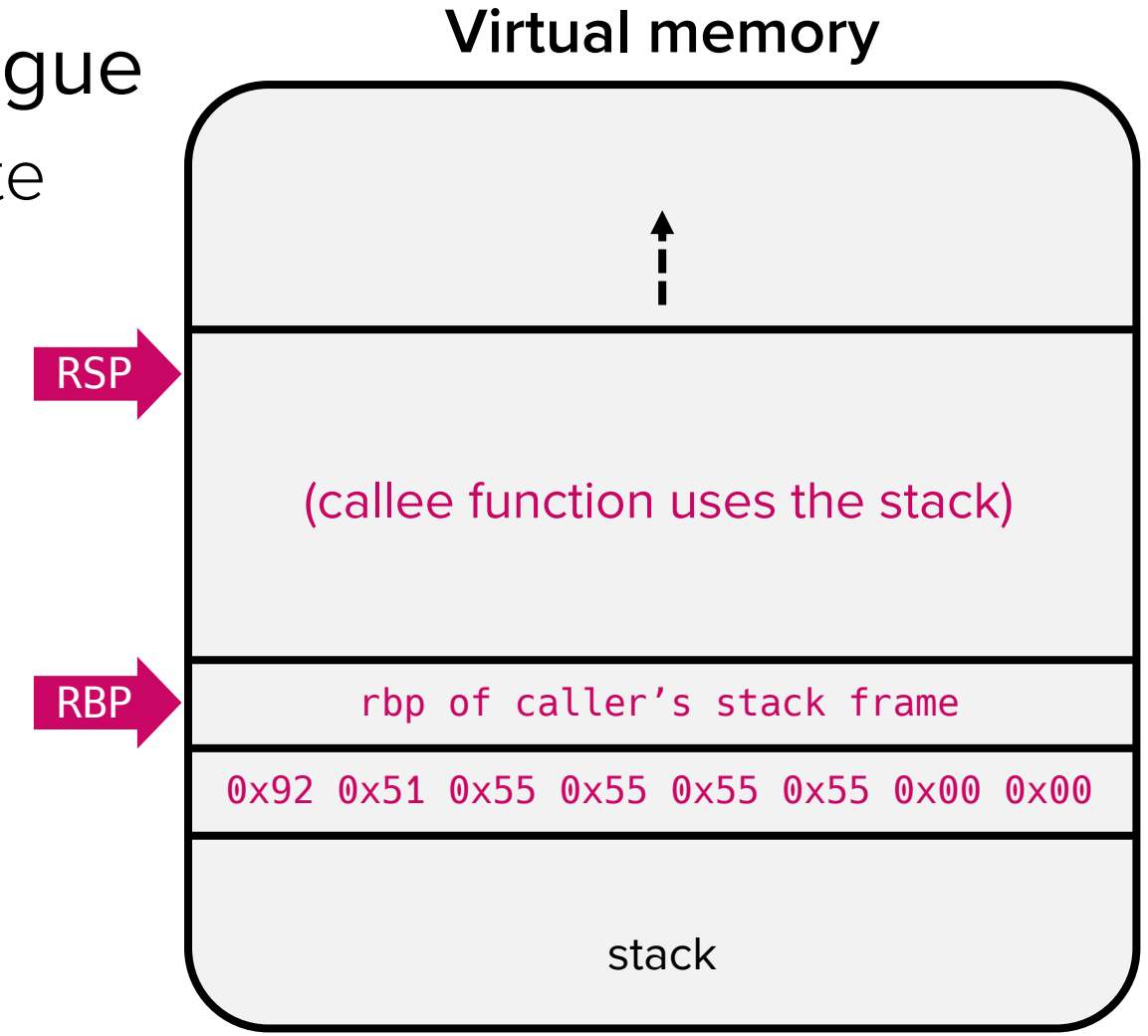
Subroutine instructions: `leave`

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

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

RIP

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



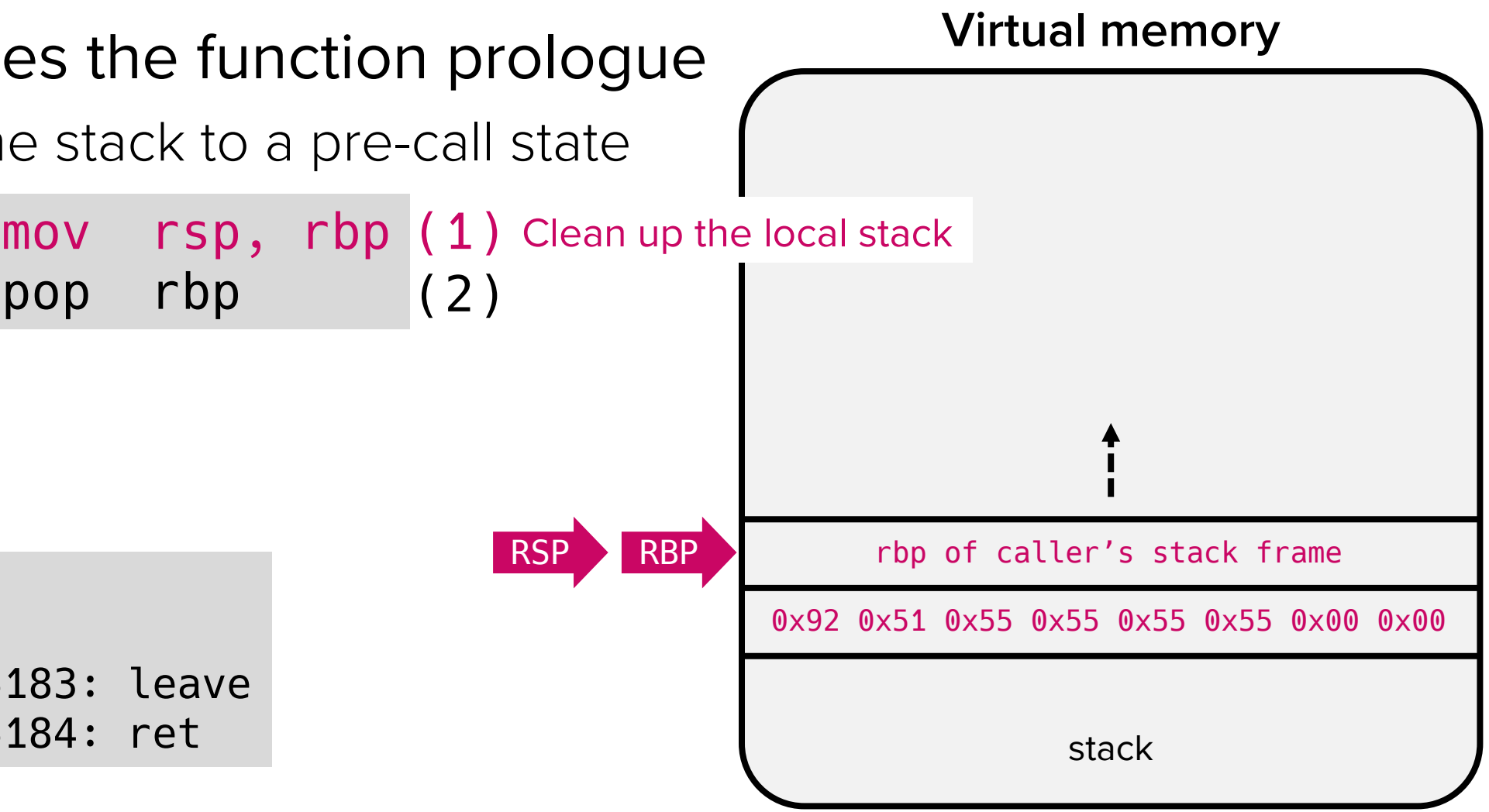
Subroutine instructions: `leave`

- `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)
```

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

RIP



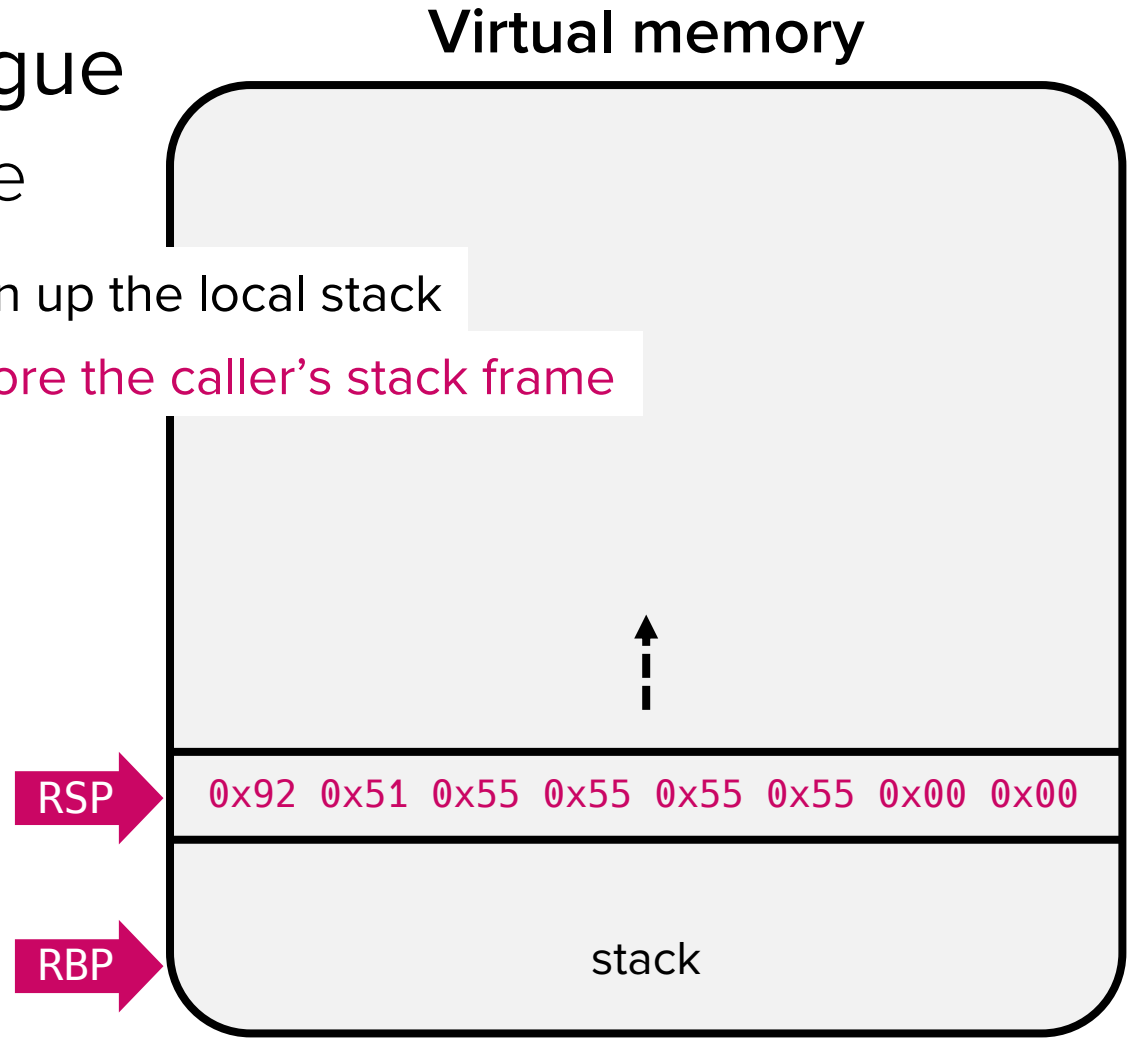
Subroutine instructions: `leave`

- `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
```

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

RIP



Subroutine instructions: `ret`

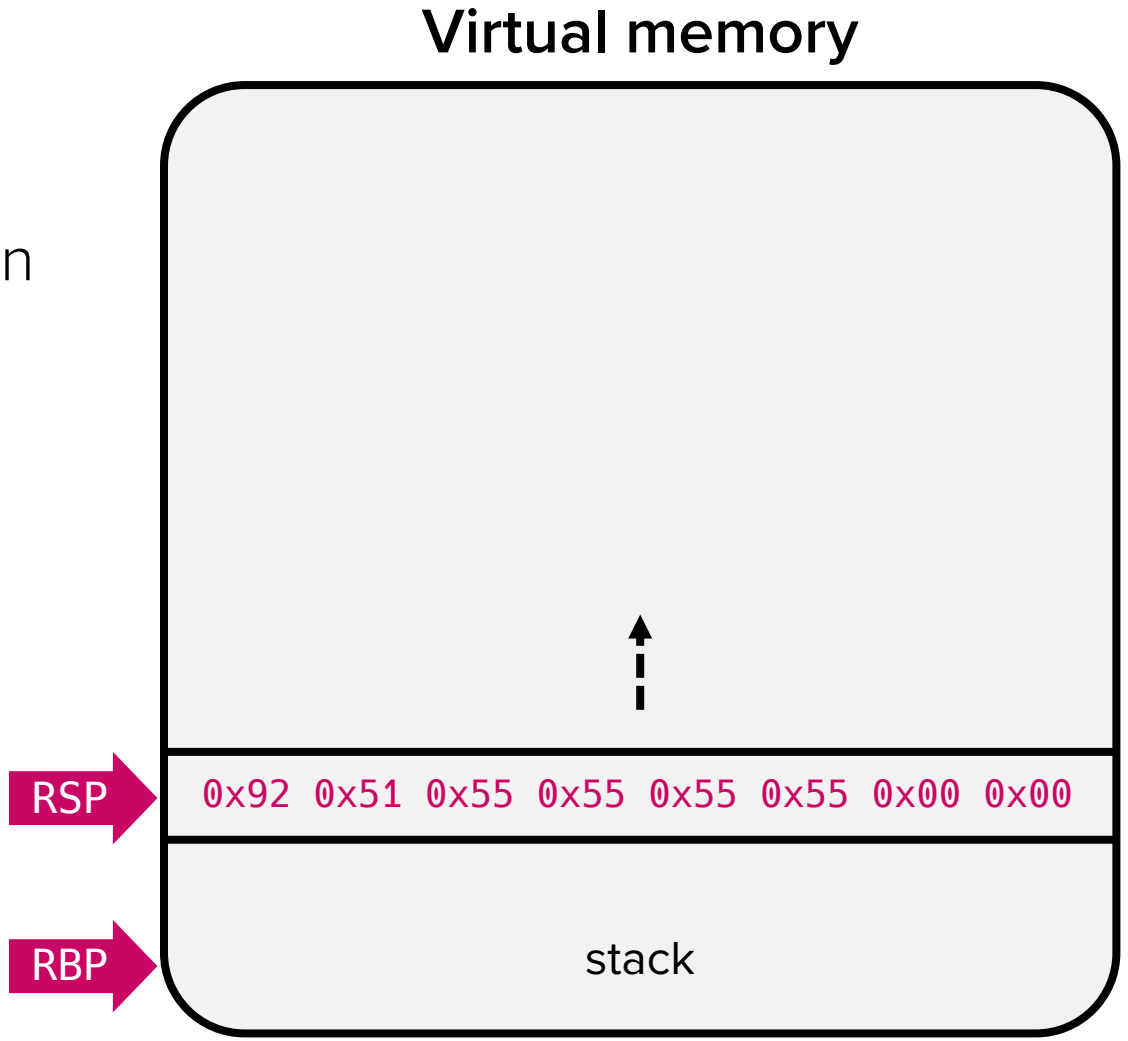
POSTECH

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

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

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

RIP



Subroutine instructions: `ret`

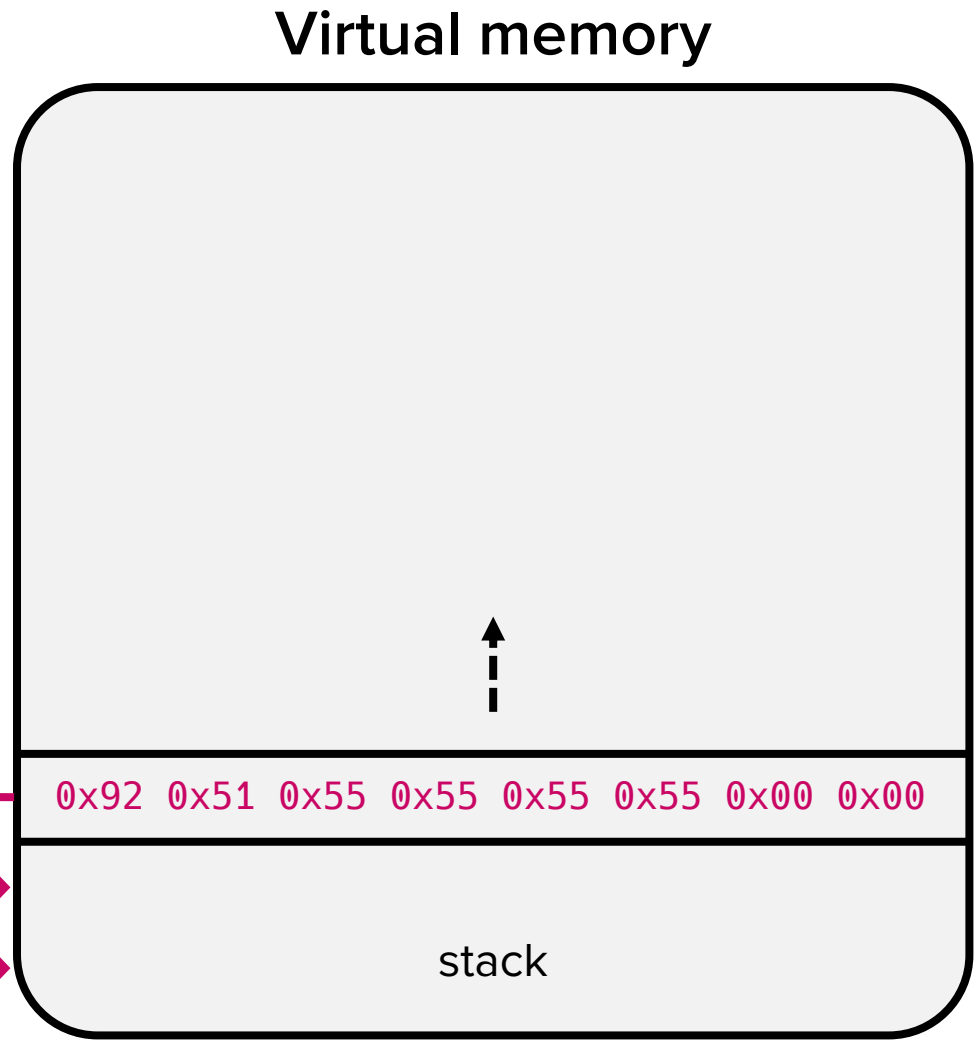
POSTECH

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

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

RIP

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



Arguments

- Arguments #1-#6 are register-passed
  - C

```c
int foo() {
  int result = bar(1, 2);
  return result * 2;
}
```

```c
int bar(int x, int y) {
  return x + y;
}
```

  - Assembly

```
<foo>
mov rdi, 1
mov rsi, 2
call <bar>
add rax, rax
ret
```

Caller sets arguments in registers

```
<bar>
mov rax, rdi
add rax, rsi
...
ret
```

Callee accesses the arguments from the registers

Return values

- Callee stores its return value in rax
  - C

```c
int foo() {
  int result = bar(1, 2);
  return result * 2;
}
```

```c
int bar(int x, int y) {
  return x + y;
}
```

  - Assembly

```
<foo>
mov rdi, 1
mov rsi, 2
call <bar>
add rax, rax
ret
```

```
<bar>
mov rax, rdi
add rax, rsi
...
ret
```

rax is the return value

foo's return value is also in rax

Branches in assembly

- Implemented using `cmp` + conditional jump

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

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

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

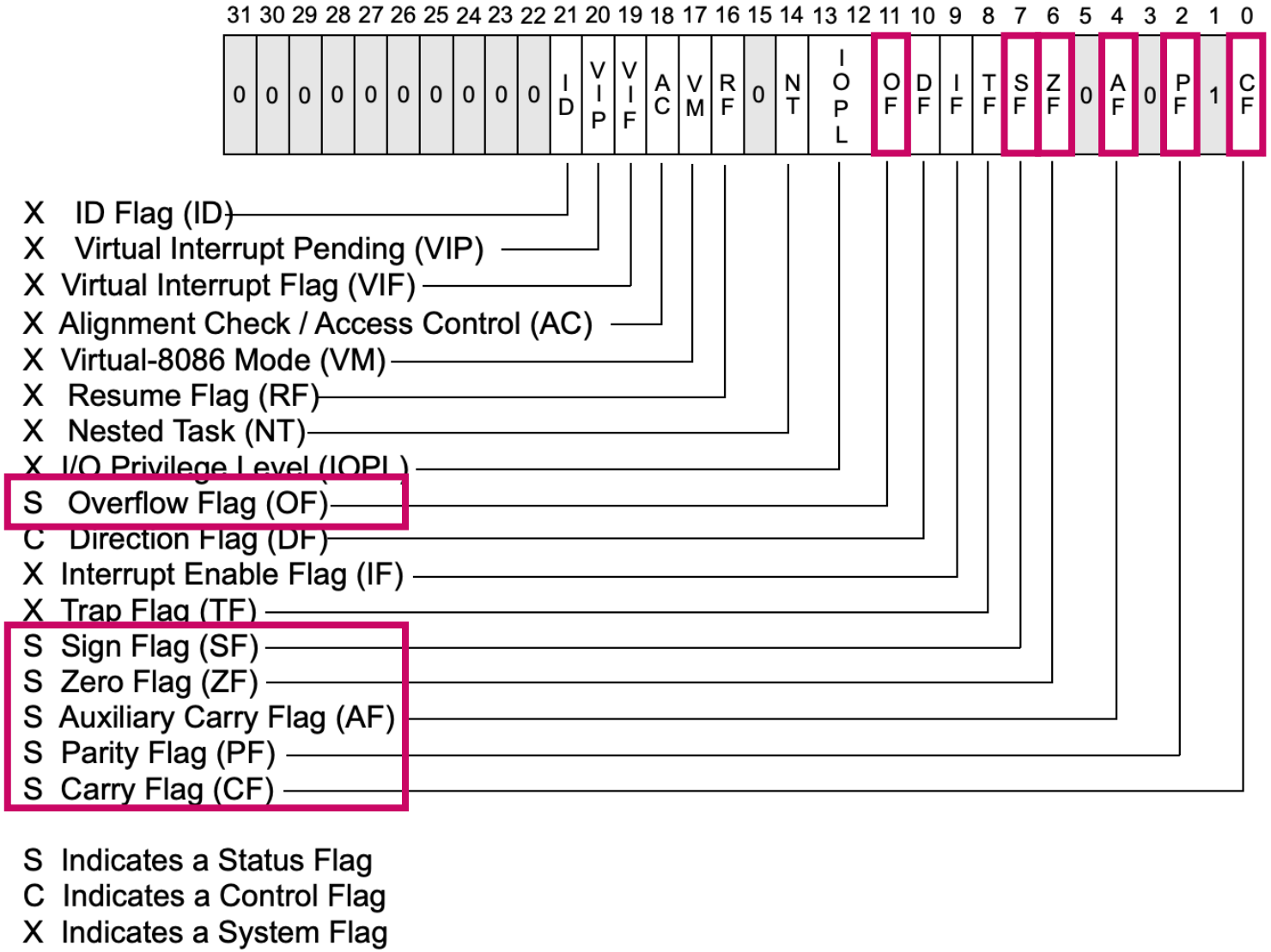
```
if (rax != 15) {
  goto addr;
}
```

```
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 01'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

- Attacking memory safety vulnerabilities
  - Exploiting buffer overflow to alter program's execution flow

Questions?