Lec 05: x86 Assembly

CSED415: Computer Security
Spring 2024

Seulbae Kim

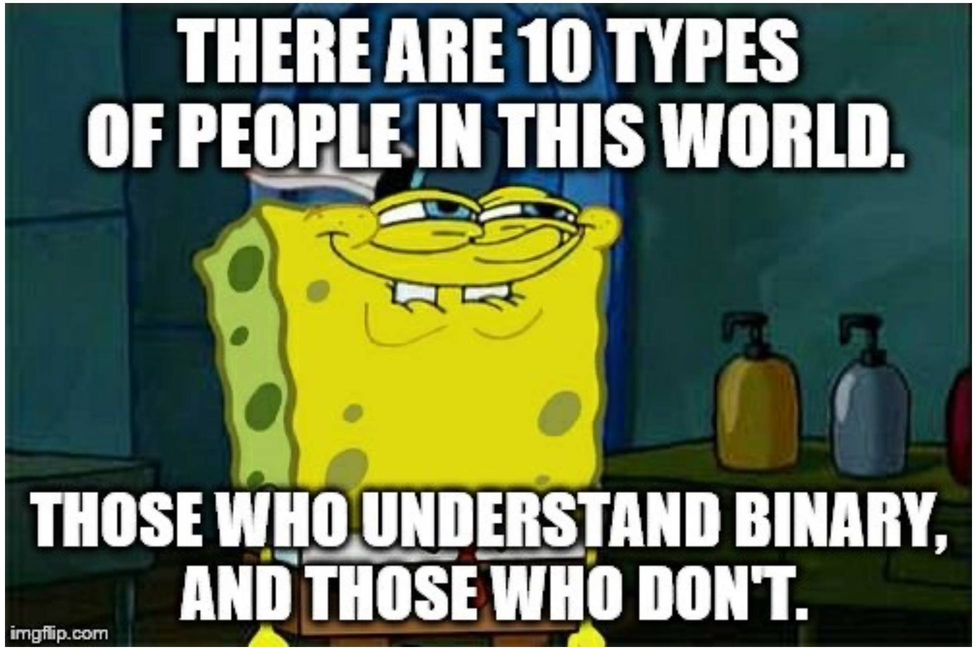
POSTECH
POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Administrivia

- Lab 01 is due this Sunday
  - Focus on today's lecture if you are struggling!
- Team forming is also due this Sunday
  - 2 students per team, six teams
  - Check "Find your Teammates" board on PLMS
- Jeongjin (TA) is taking a leave of absence
  - Please do not send him emails
  - Use PLMS Q&A board for questions

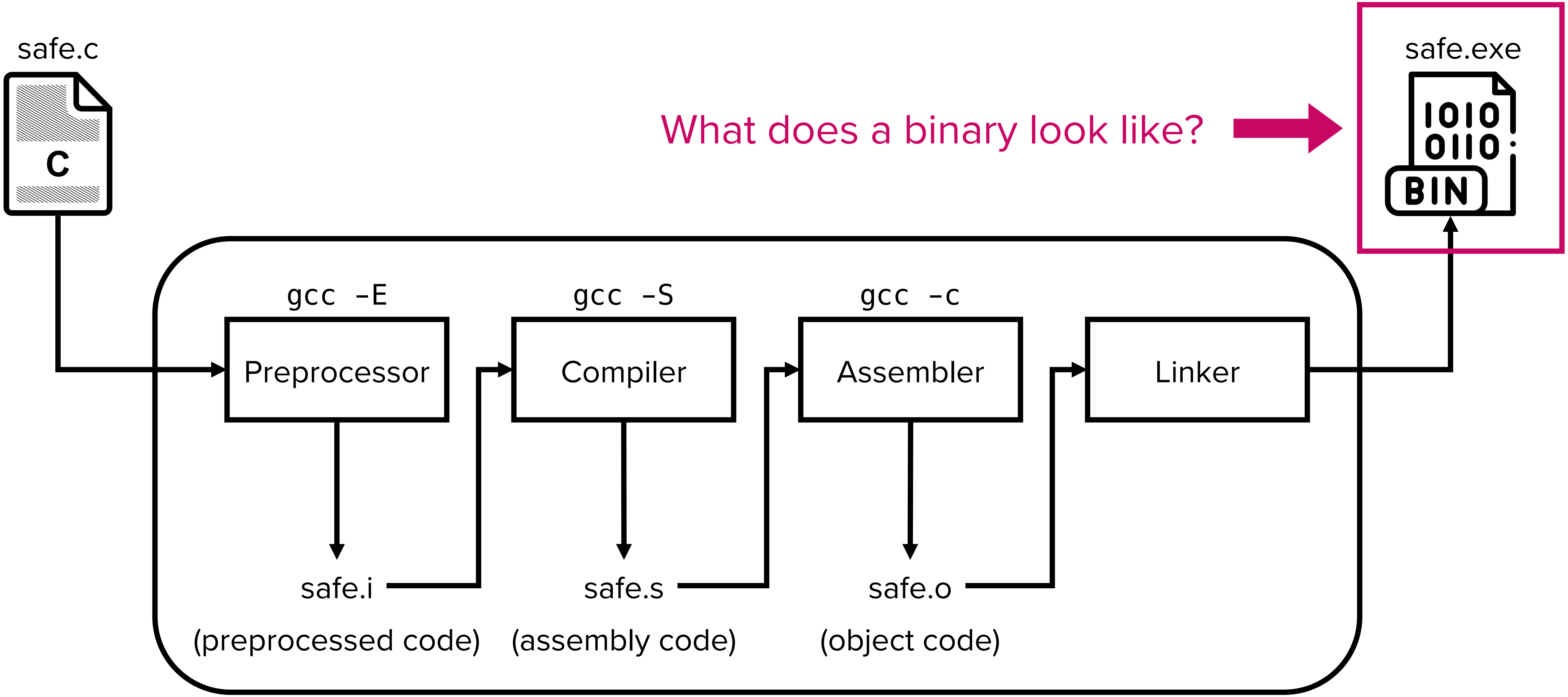
Recap

- Secure coding is necessary to build secure computer systems
- However, code-level mitigation may not be enough
  - We cannot trust anything that we did not create ourselves
  - e.g., Thompson compiler
- Binary analysis, like it or not, is required
  - We want to find out whether a system is secure or not
  - Analyze exactly what gets executed on the CPU

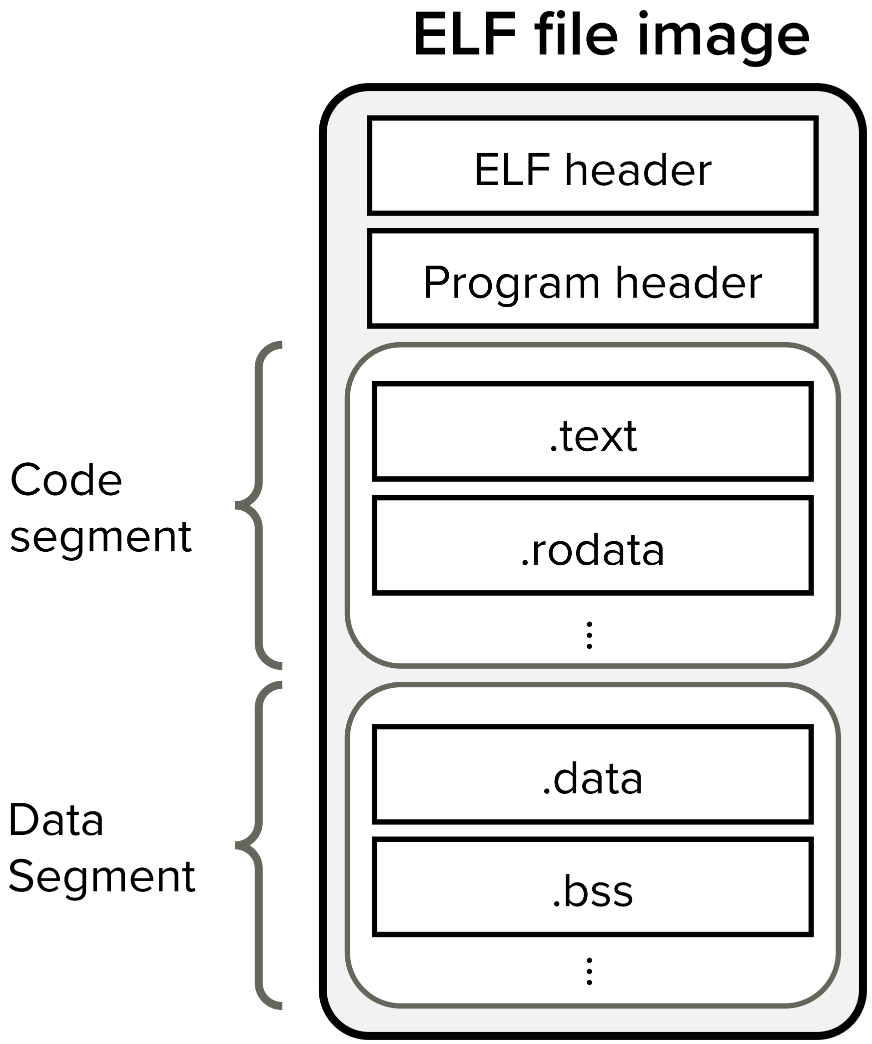
BIN  ARY

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

Recap: Compiler 101

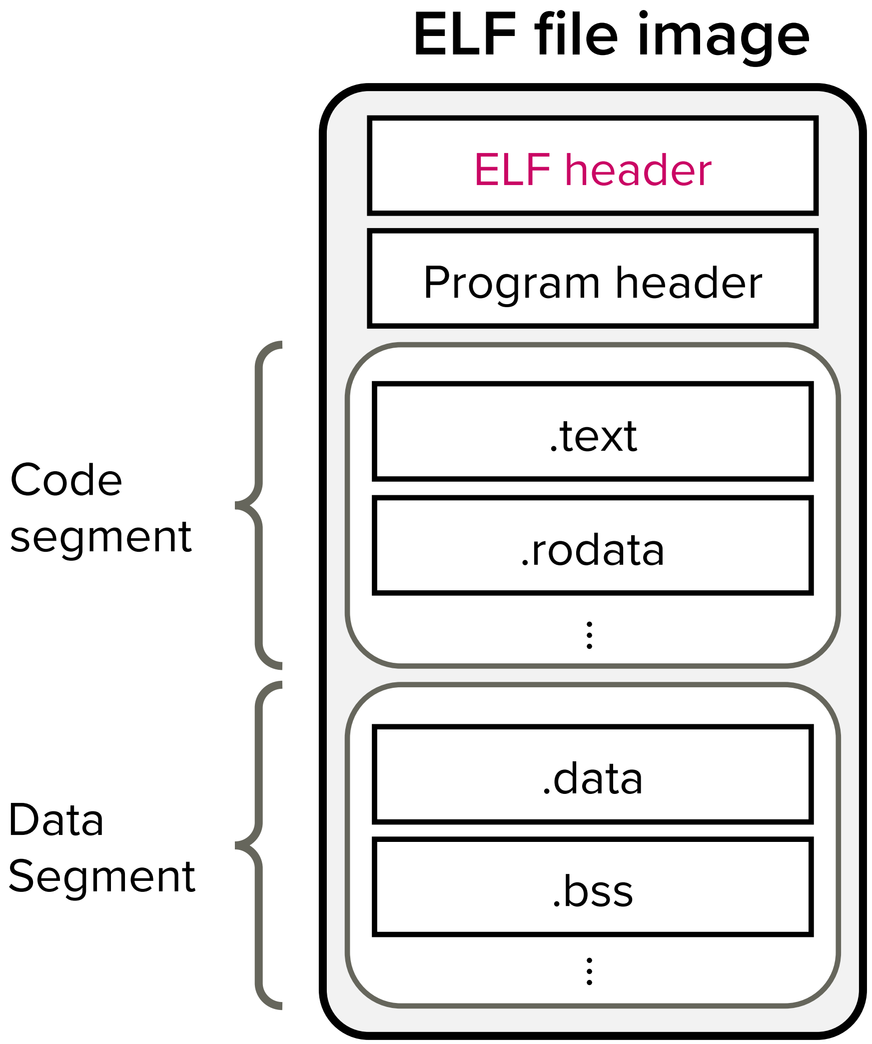


Executable binary (ELF-formatted file)



- ELF (Executable and Linkable Format)
  - ELF header
  - Program header
  - Segments
    - Contains Sections

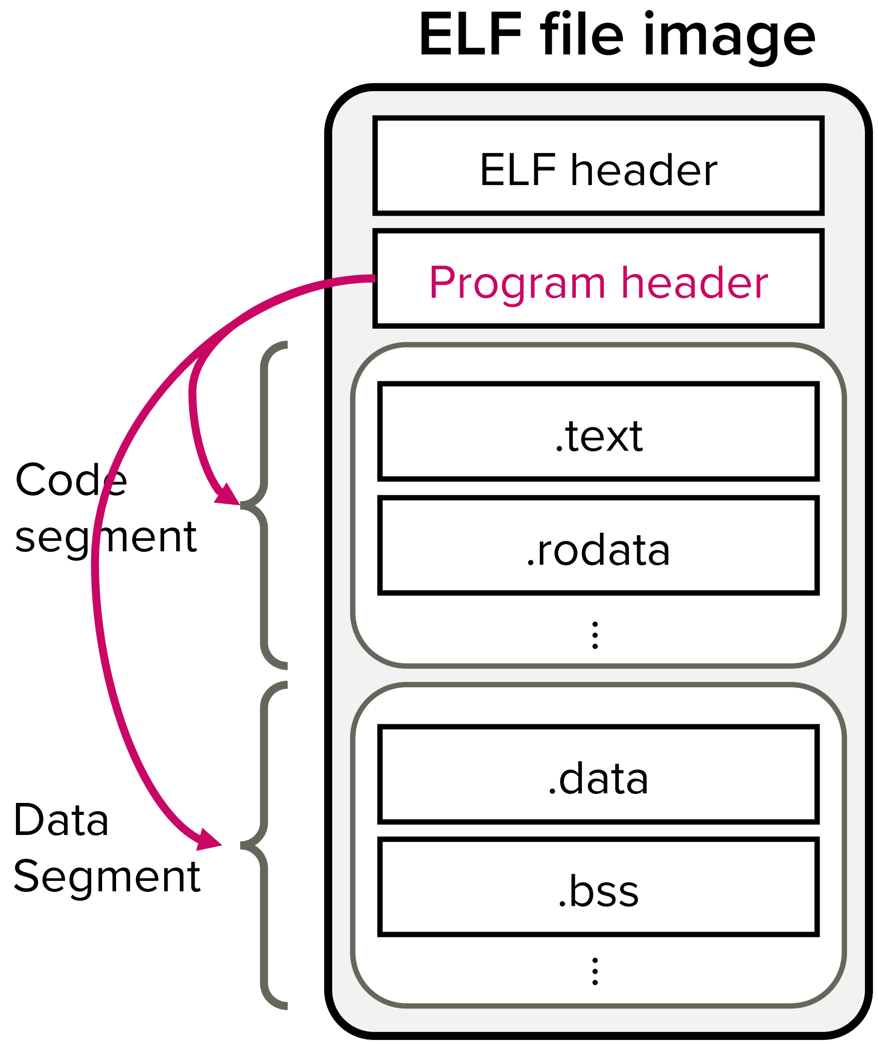
Executable binary



- ELF header
  - Always at the beginning of an ELF file
  - Contains magic number, architecture info, entry point address, …

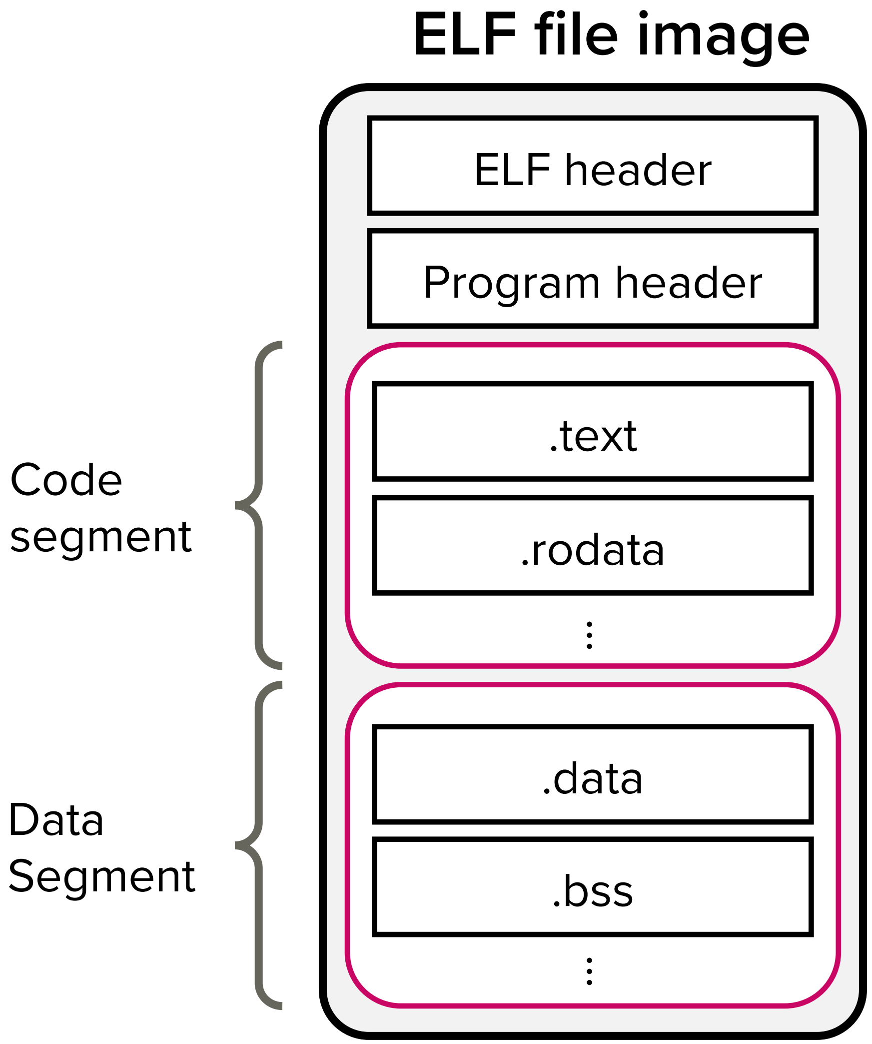
```
lab01@csed415:~$ readelf -h ./target
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x8049110
  Start of program headers:          52 (bytes into file)
  Start of section headers:          14328 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         11
  Size of section headers:           40 (bytes)
  Number of section headers:         29
  Section header string table index: 28
```

Executable binary



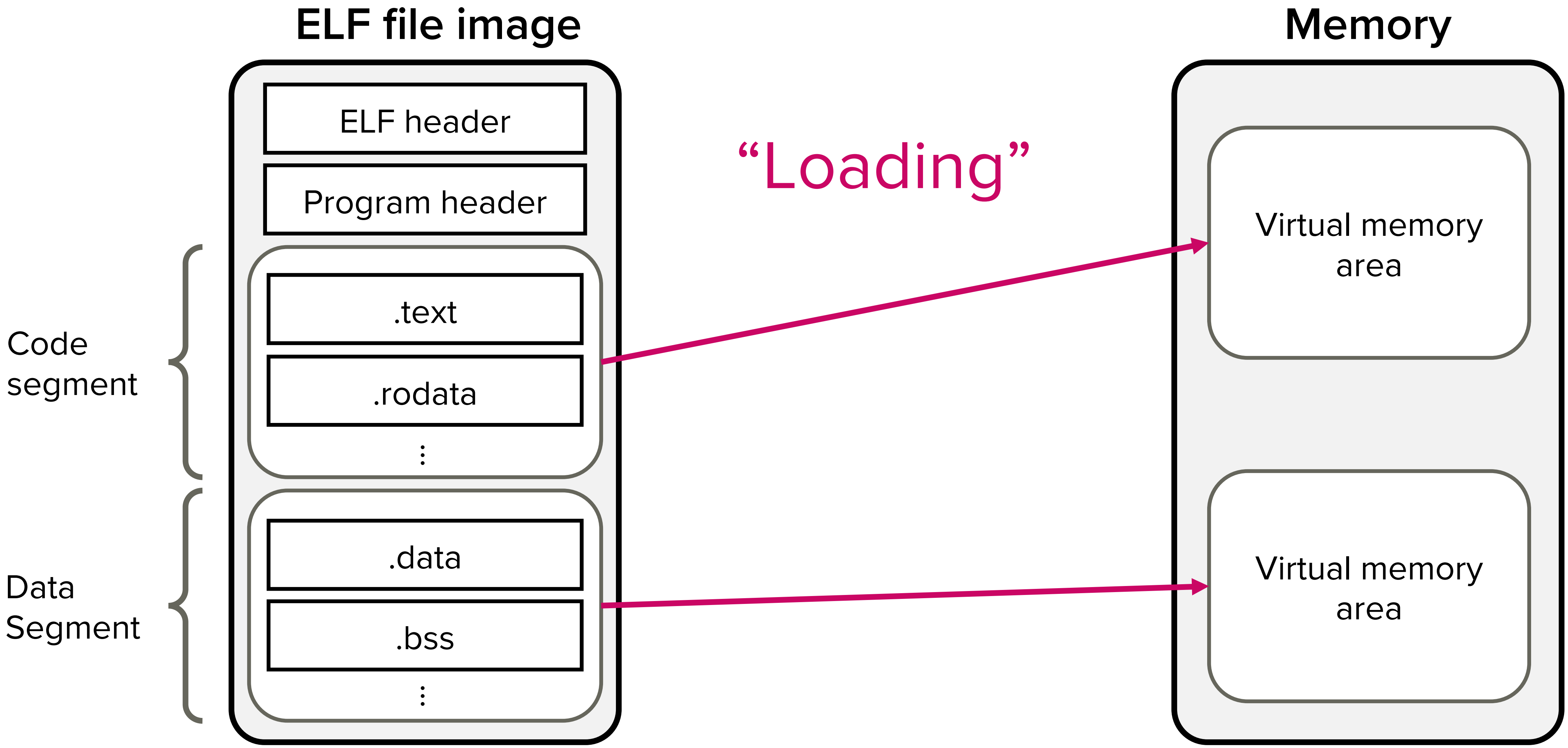
- Program header
  - Contains segment information
    - Offset and size of each segment (code segment is size bytes from offset)
    - Virtual memory address each segment should be loaded
    - Permission of each segment

Executable binary



- Segments
  - Logical groups of sections
  - Each segment is loaded onto one or more virtual memory areas at runtime
  - Code segment (text segment)
    - Includes program instructions and read only data (e.g., string literals)
  - Data segment
    - Includes static variables, uninitialized variables, …

Loader turns a binary image into a process



ELF loader workflow

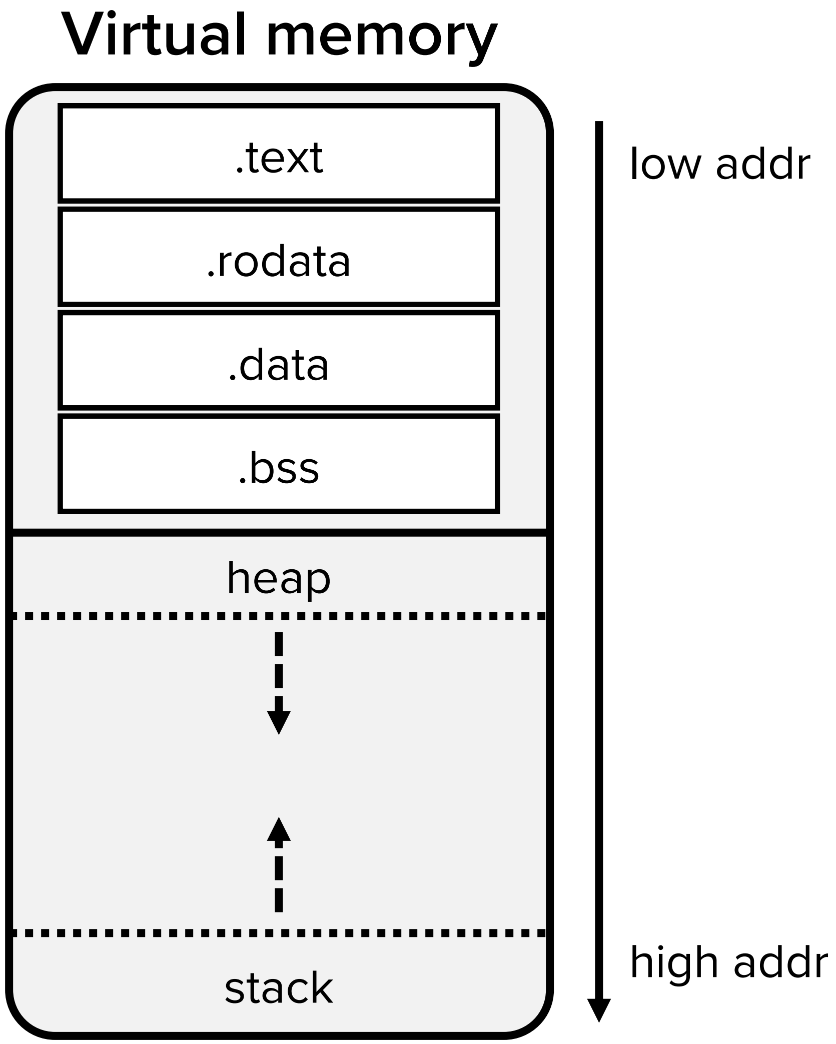
1. Read the ELF Header, which is always located at the very beginning of an ELF file
2. Read the program headers. These specify where in the file the program segments are located, and where they need to be loaded into memory
3. Parse the program headers to determine the number of program segments that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

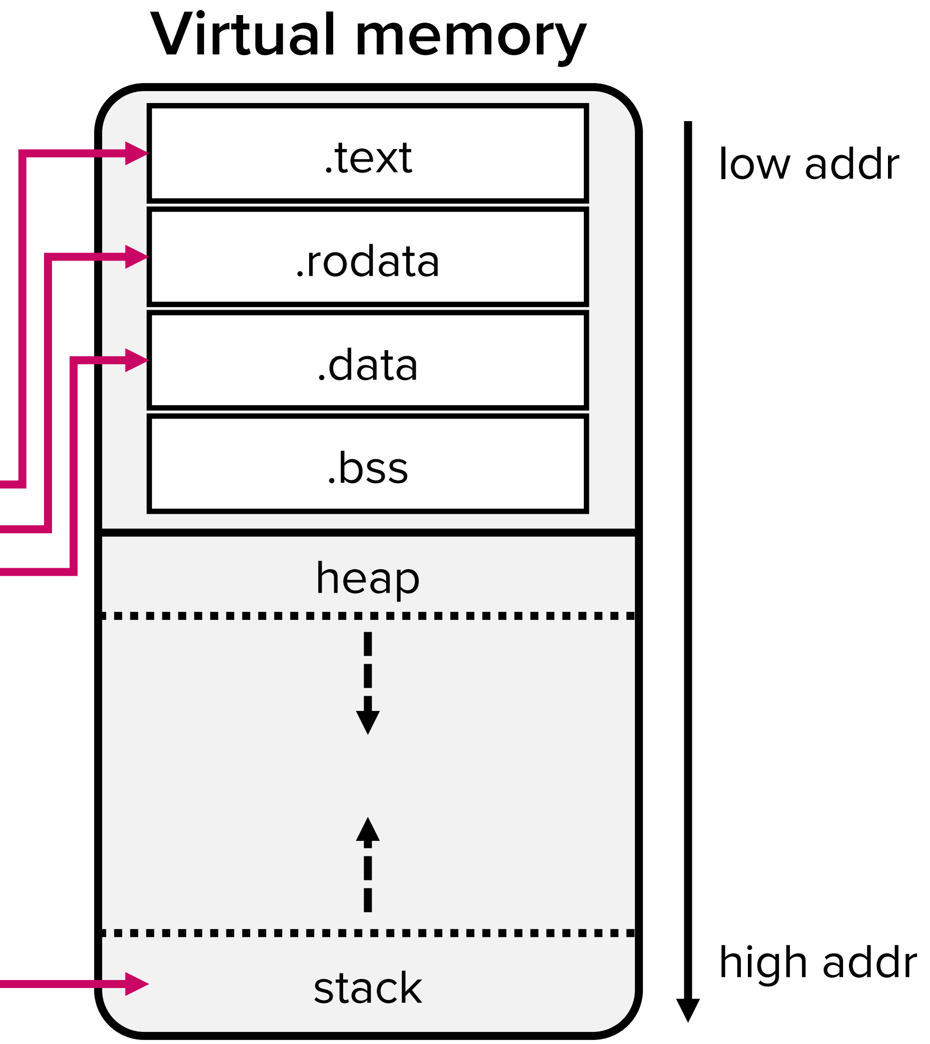
- ELF segments + heap + stack
  - Heap
    - Dynamically allocated memory (e.g., using malloc)
    - Grows towards higher (larger) addresses
  - Stack
    - Contains runtime call stack
      - More on this later!
    - Grows towards lower (smaller) addresses



Exercise: Examine Lab 01's target in GDB

```
$ ssh lab01@141.223.181.22
$ gdb ./target
pwndbg> break main
pwndbg> r
pwndbg> vmmap
```

```
pwndbg> vmmap
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
     Start        End Perm     Size Offset File
 0x8048000  0x8049000 r--p     1000      0 /home/lab01/target
 0x8049000  0x804a000 r-xp     1000   1000 /home/lab01/target
 0x804a000  0x804b000 r--p     1000   2000 /home/lab01/target
 0x804b000  0x804c000 r--p     1000   2000 /home/lab01/target
 0x804c000  0x804d000 rw-p     1000   3000 /home/lab01/target
0xf7cc1000 0xf7ce1000 r--p    20000      0 /lib/i386-linux-gnu/libc.so.6
0xf7ce1000 0xf7e63000 r-xp   182000  20000 /lib/i386-linux-gnu/libc.so.6
0xf7e63000 0xf7ee8000 r--p    85000 1a2000 /lib/i386-linux-gnu/libc.so.6
0xf7ee8000 0xf7ee9000 ---p     1000 227000 /lib/i386-linux-gnu/libc.so.6
0xf7ee9000 0xf7eeb000 r--p     2000 227000 /lib/i386-linux-gnu/libc.so.6
0xf7eeb000 0xf7eec000 rw-p     1000 229000 /lib/i386-linux-gnu/libc.so.6
0xf7eec000 0xf7ef6000 rw-p     a000      0 [anon_f7eec]
0xf7eff000 0xf7f01000 rw-p     2000      0 [anon_f7eff]
0xf7f01000 0xf7f05000 r--p     4000      0 [vvar]
0xf7f05000 0xf7f07000 r-xp     2000      0 [vdso]
0xf7f07000 0xf7f08000 r--p     1000      0 /lib/i386-linux-gnu/ld-linux.so.2
0xf7f08000 0xf7f2d000 r-xp    25000   1000 /lib/i386-linux-gnu/ld-linux.so.2
0xf7f2d000 0xf7f3c000 r--p     f000  26000 /lib/i386-linux-gnu/ld-linux.so.2
0xf7f3c000 0xf7f3e000 r--p     2000  34000 /lib/i386-linux-gnu/ld-linux.so.2
0xf7f3e000 0xf7f3f000 rw-p     1000  36000 /lib/i386-linux-gnu/ld-linux.so.2
0xffeca000 0xffeeb000 rw-p    21000      0 [stack]
```

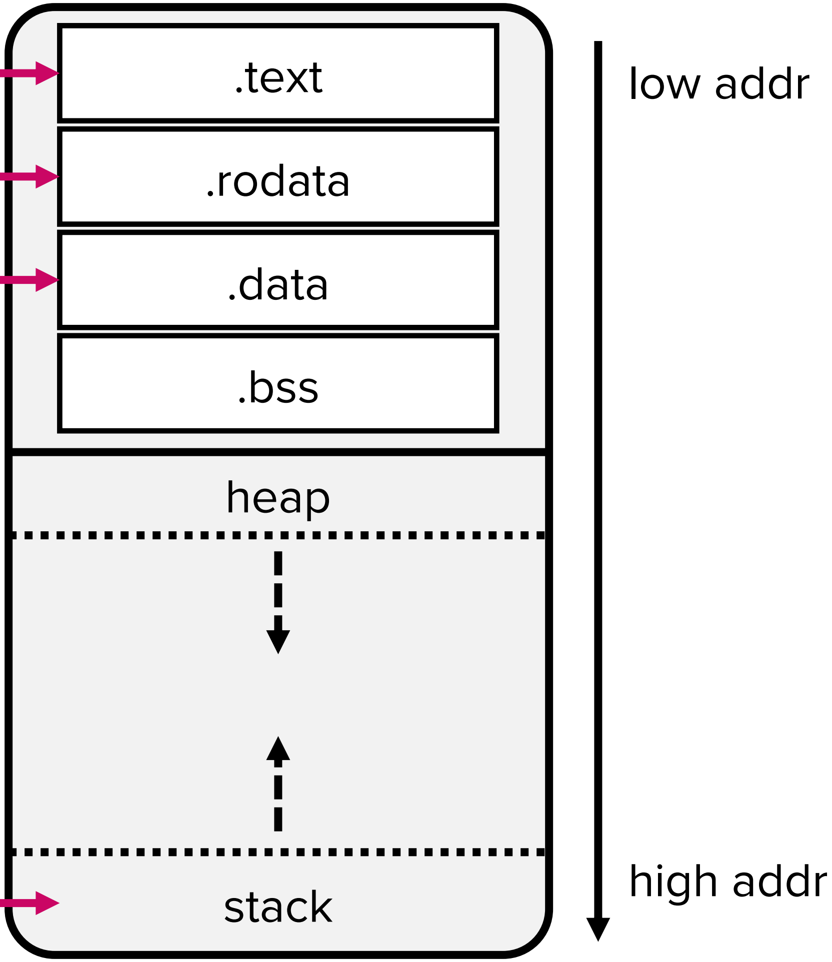


Exercise: Examine Lab 01's target in GDB

```
pwndbg> elfsections // check section info
pwndbg> x/20i 0x8049110 // exam text sec. instructions
pwndbg> backtrace // backtrace from main (start_)
pwndbg> x/10s 0x804a000 // exam rodata sec. strings
pwndbg> x/100wx 0x804c000 // exam data/bss sec.
```

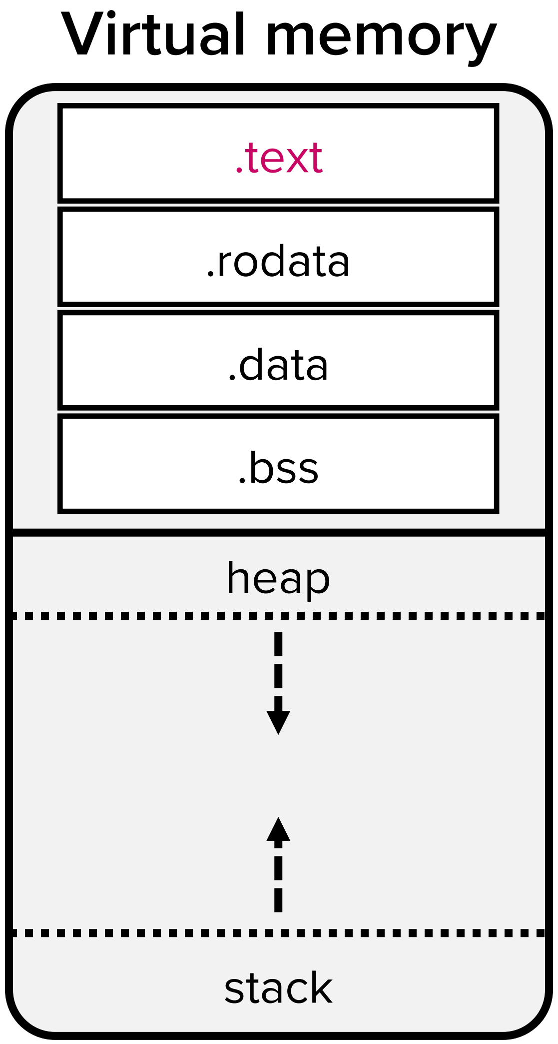
```
pwndbg> vmmap
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
     Start        End Perm     Size Offset File
 0x8048000  0x8049000 r--p     1000      0 /home/lab01/target
 0x8049000  0x804a000 r-xp     1000   1000 /home/lab01/target
 0x804a000  0x804b000 r--p     1000   2000 /home/lab01/target
 0x804b000  0x804c000 r--p     1000   2000 /home/lab01/target
 0x804c000  0x804d000 rw-p     1000   3000 /home/lab01/target
0xf7cc1000 0xf7ce1000 r--p    20000      0 /lib/i386-linux-gnu/libc.so.6
0xf7ce1000 0xf7e63000 r-xp   182000  20000 /lib/i386-linux-gnu/libc.so.6
0xf7e63000 0xf7ee8000 r--p    85000 1a2000 /lib/i386-linux-gnu/libc.so.6
0xf7ee8000 0xf7ee9000 ---p     1000 227000 /lib/i386-linux-gnu/libc.so.6
0xf7ee9000 0xf7eeb000 r--p     2000 227000 /lib/i386-linux-gnu/libc.so.6
0xf7eeb000 0xf7eec000 rw-p     1000 229000 /lib/i386-linux-gnu/libc.so.6
0xf7eec000 0xf7ef6000 rw-p     a000      0 [anon_f7eec]
0xf7eff000 0xf7f01000 rw-p     2000      0 [anon_f7eff]
0xf7f01000 0xf7f05000 r--p     4000      0 [vvar]
0xf7f05000 0xf7f07000 r-xp     2000      0 [vdso]
0xf7f07000 0xf7f08000 r--p     1000      0 /lib/i386-linux-gnu/ld-linux.so.2
0xf7f08000 0xf7f2d000 r-xp    25000   1000 /lib/i386-linux-gnu/ld-linux.so.2
0xf7f2d000 0xf7f3c000 r--p     f000  26000 /lib/i386-linux-gnu/ld-linux.so.2
0xf7f3c000 0xf7f3e000 r--p     2000  34000 /lib/i386-linux-gnu/ld-linux.so.2
0xf7f3e000 0xf7f3f000 rw-p     1000  36000 /lib/i386-linux-gnu/ld-linux.so.2
0xffeca000 0xffeeb000 rw-p    21000      0 [stack]
```

Virtual memory



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

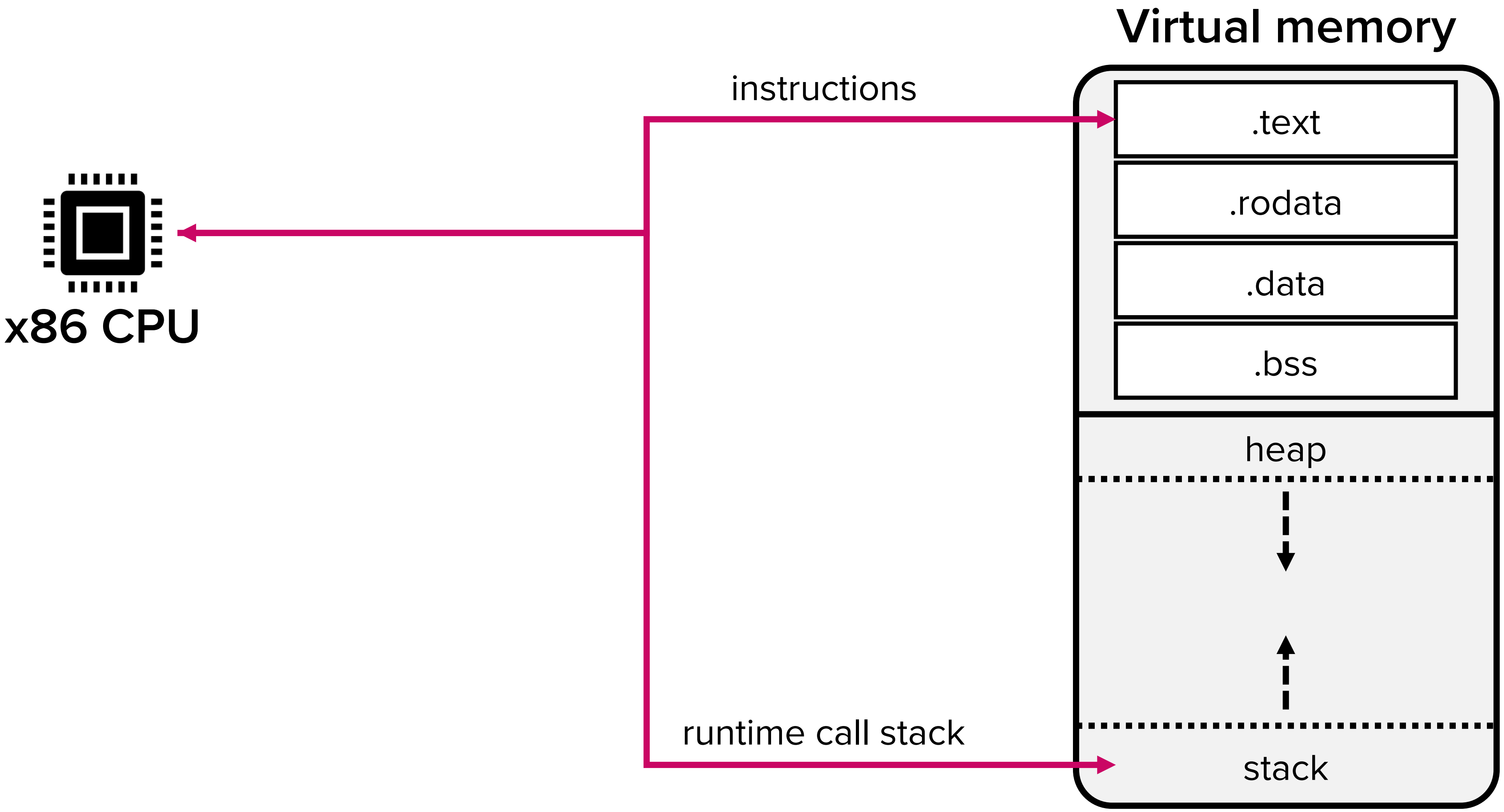


x86 Registers

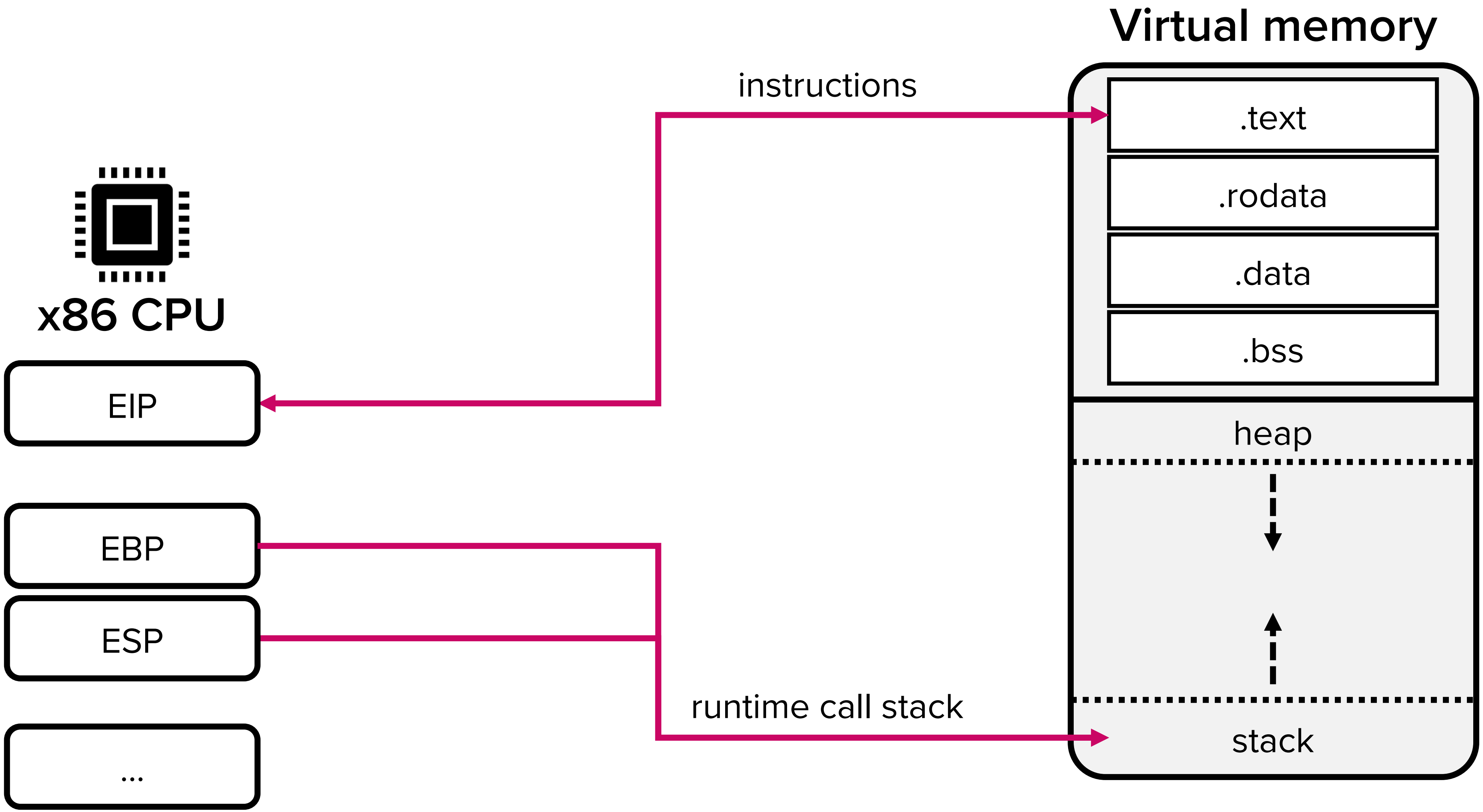
x86 architecture

- Developed by Intel in 1985
- 32-bit address space
  - Theoretically 2^32 = 4GB of virtual memory can be used
- One of the most common architectures

CPU executes a program



CPU executes a program using registers



x86 registers

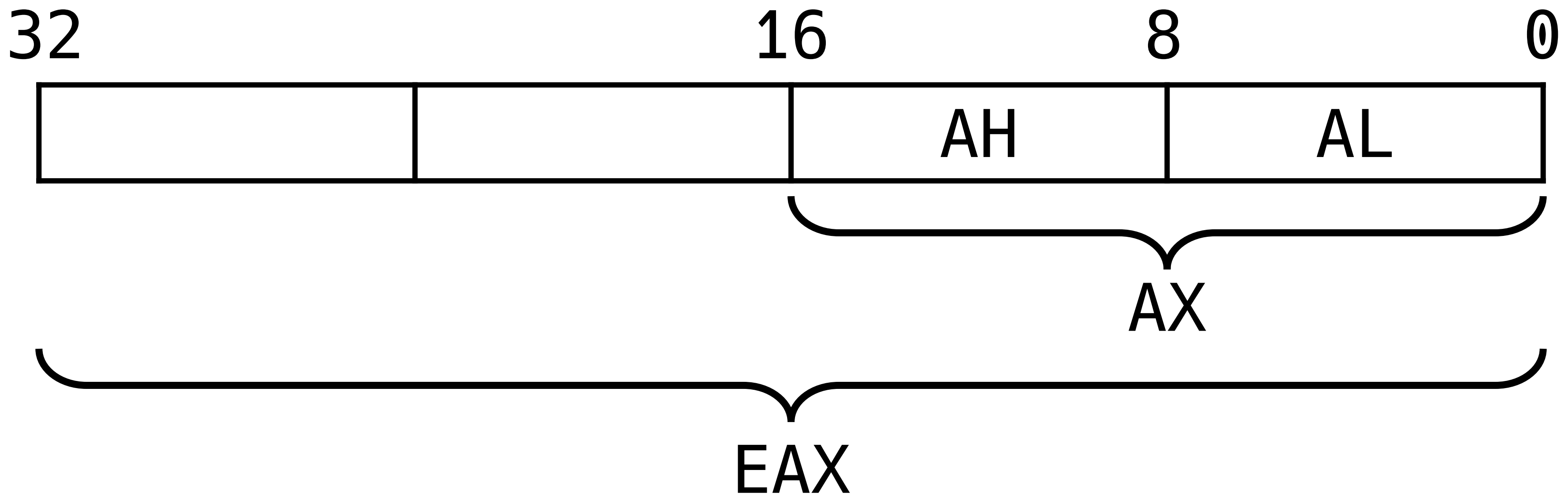
- General purpose registers (GPR)
  - EAX, EBX, ECX, EDX
- Pointers
  - ESI, EDI
- Stack pointers
  - ESP, EBP
- Special purpose registers
  - EIP, EFLAGS

History of Intel/AMD processors

1978	8086 80286	16-bit processor, used IP, SP, BP, … registers
1985	80386 80486 …	32-bit processor, used EIP, ESP, EBP, … registers (Extended)
2003 2006	Opteron Prescott Core 2 …	64-bit processor, used RIP, RSP, RBP, … registers (Register)

x86 registers are extended from 16-bit registers

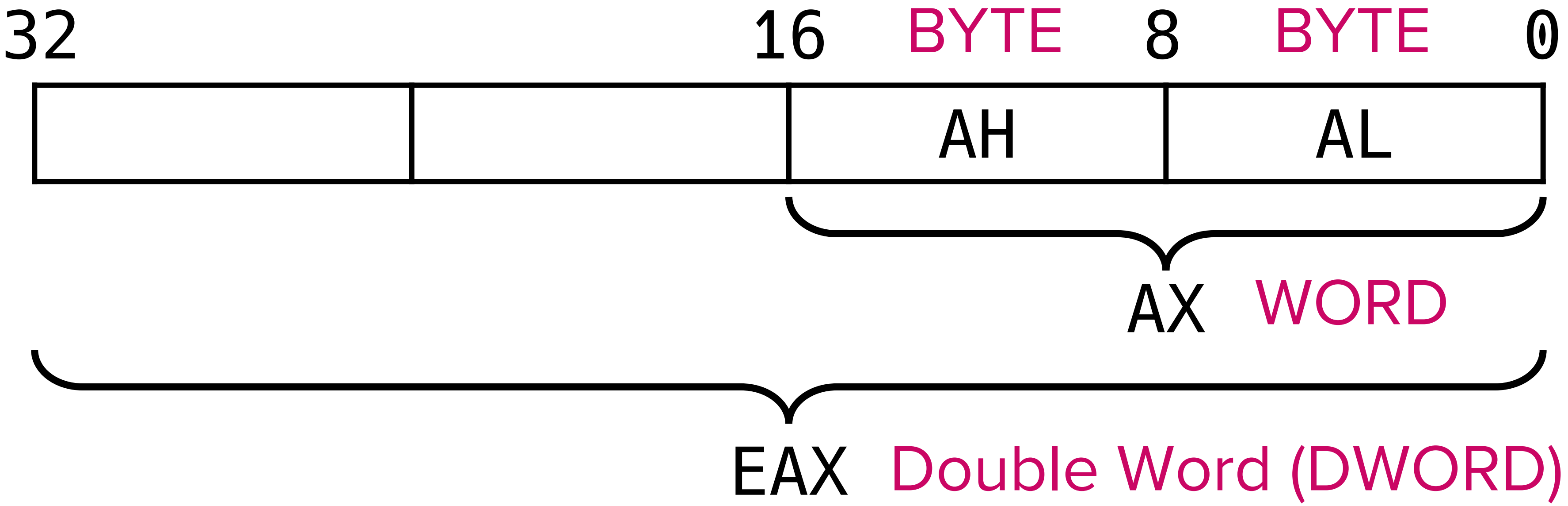
- AL, AH, AX, and EAX are accessiable



(Same applies to other general purpose registers)

Q) Can you draw RAX for x86_64?

x86 size convention: Word is 16 bits



RAX: Quad Word (QWORD)

x86 Assembly

Basic format of x86 instructions

- x86 instructions have 0, 1, or 2 operands
  - 0 operand: `ret`
  - 1 operand: `inc   eax`

    operand 1
  - 2 operands: `mov   eax, ebx`

    operand 1

    operand 2

Basic semantics

- Operand 1 (usually) stores the result

```
mov  eax, ebx
```
// eax = ebx

```
sub  esp, 0xc
```
// esp = esp – 0x8

```
inc  eax
```
// eax = eax + 1

Operand types

- Operand can be register, memory, or constant

```
mov   eax, [ebx]
```

register   memory pointed to by ebx

```
sub   esp, 0xc
```

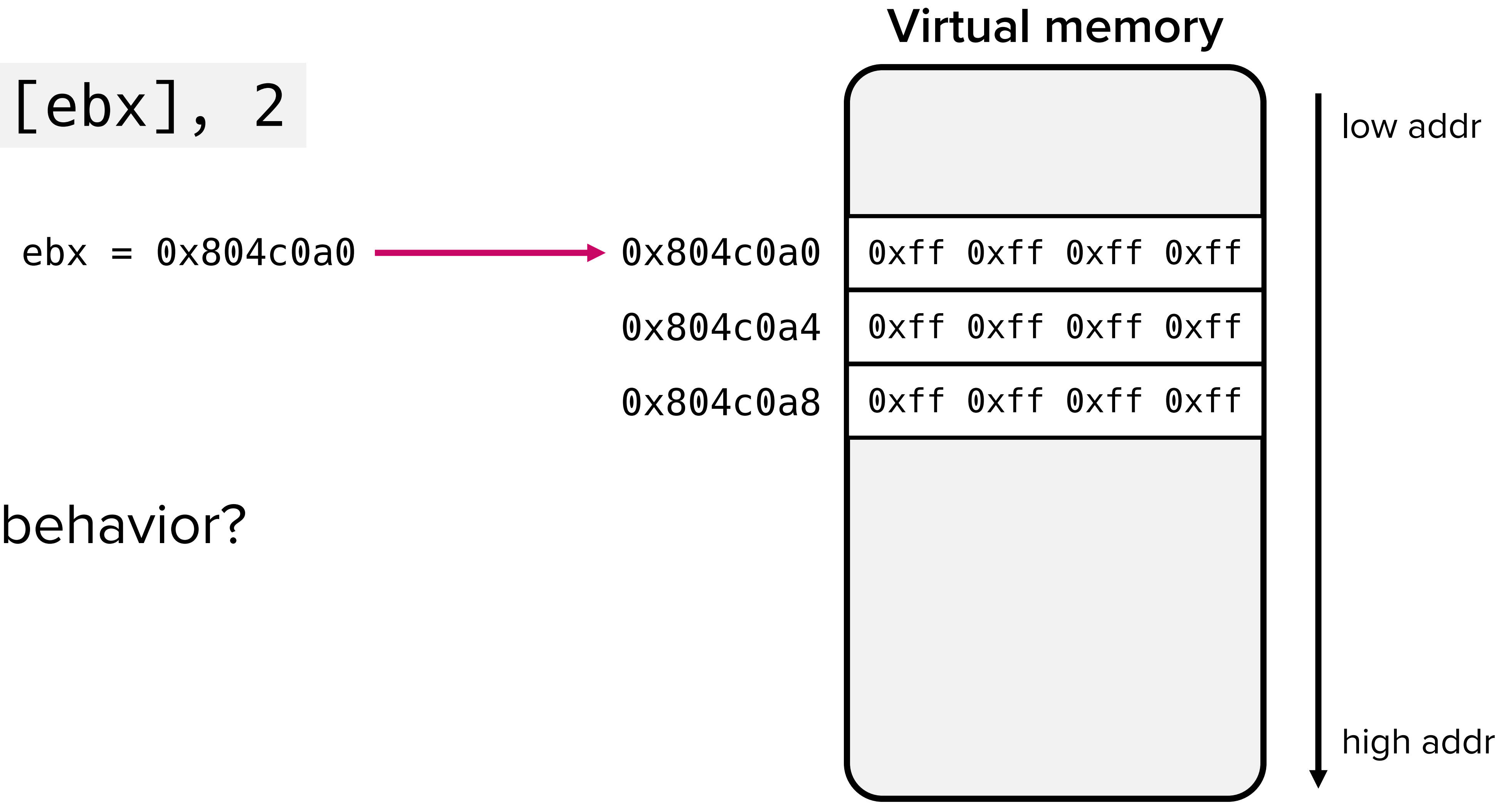
constant

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

???   memory pointed to by ecx

Size directives – motivation

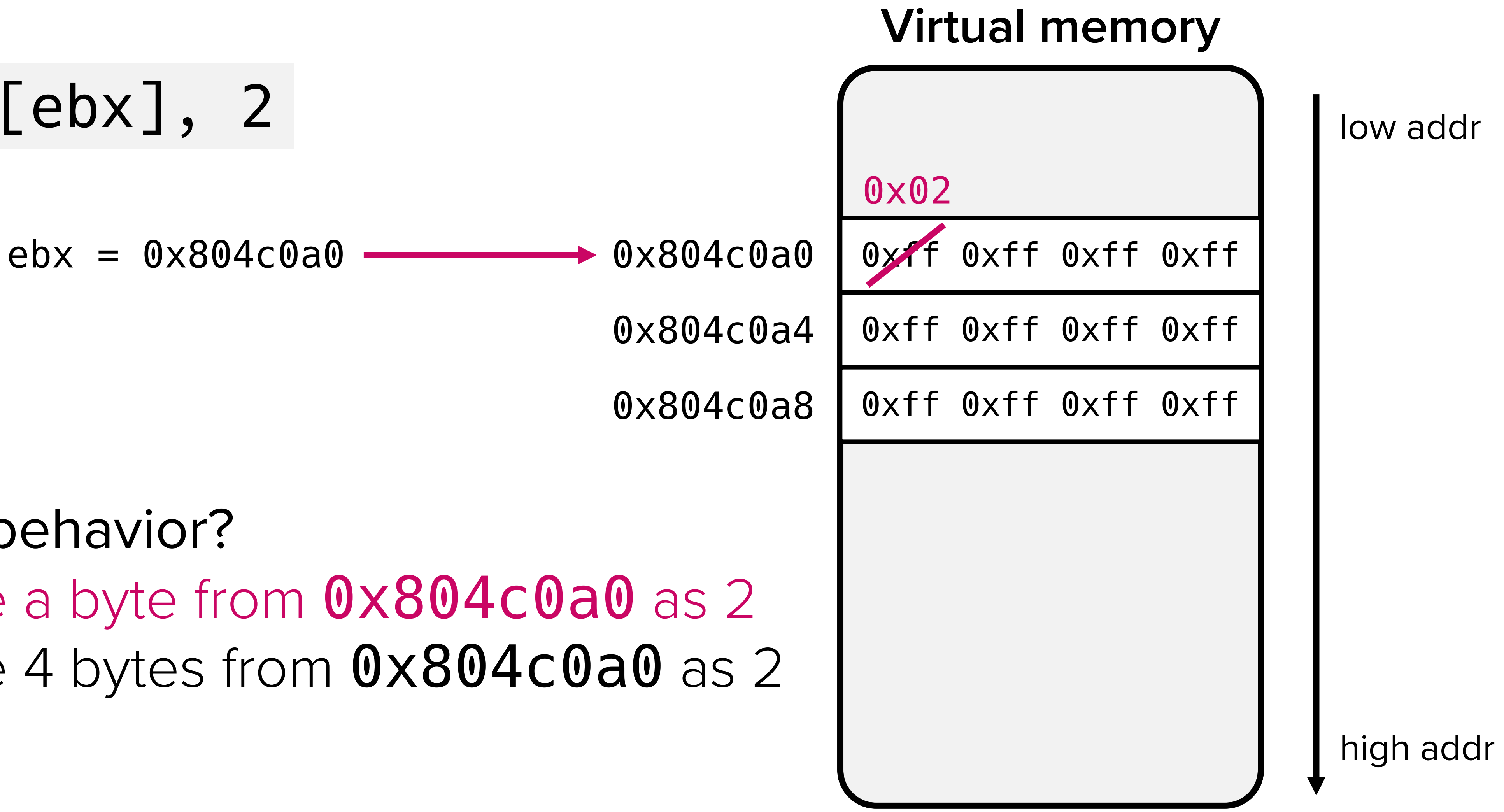
```
mov  [ebx], 2
```



Q) Correct behavior?

Size directives – motivation

```
mov  [ebx], 2
```



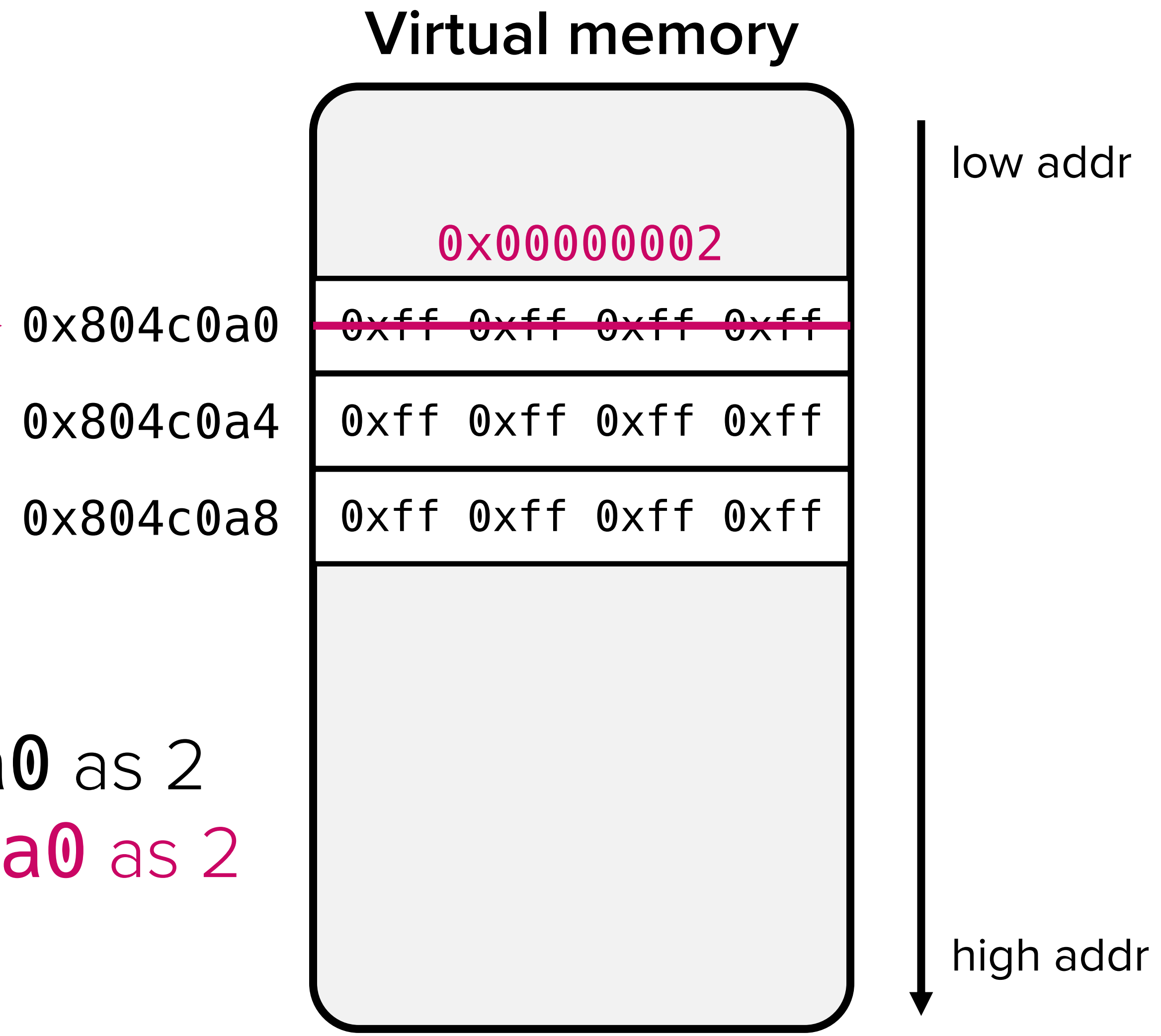
Q) Correct behavior?

1. Overwrite a byte from `0x804c0a0` as 2
2. Overwrite 4 bytes from `0x804c0a0` as 2

Size directives – motivation

```
mov  [ebx], 2
```

`ebx = 0x804c0a0` →

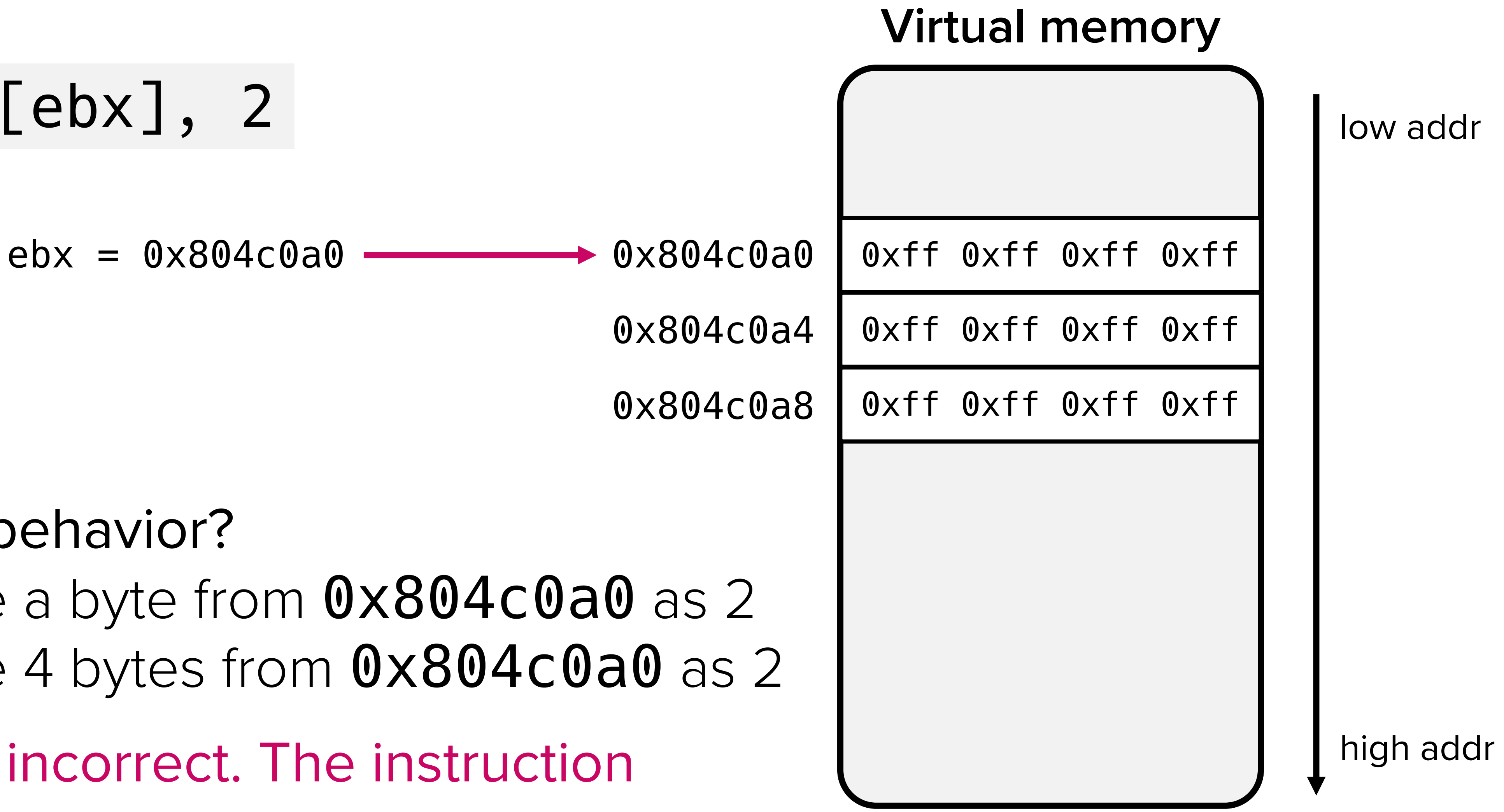


Q) Correct behavior?

1. Overwrite a byte from **`0x804c0a0`** as 2
2. Overwrite 4 bytes from **`0x804c0a0`** as 2

Size directives – motivation

```
mov  [ebx], 2
```



Q) Correct behavior?

1. Overwrite a byte from **`0x804c0a0`** as 2
2. Overwrite 4 bytes from **`0x804c0a0`** as 2

A) Both are incorrect. The instruction leads to an error due to ambiguity.

Size directives: BYTE/WORD/DWORD PTR

```
mov  [ebx], 2
```

ERROR

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

or

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

or

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

[ebx] [ebx+3]

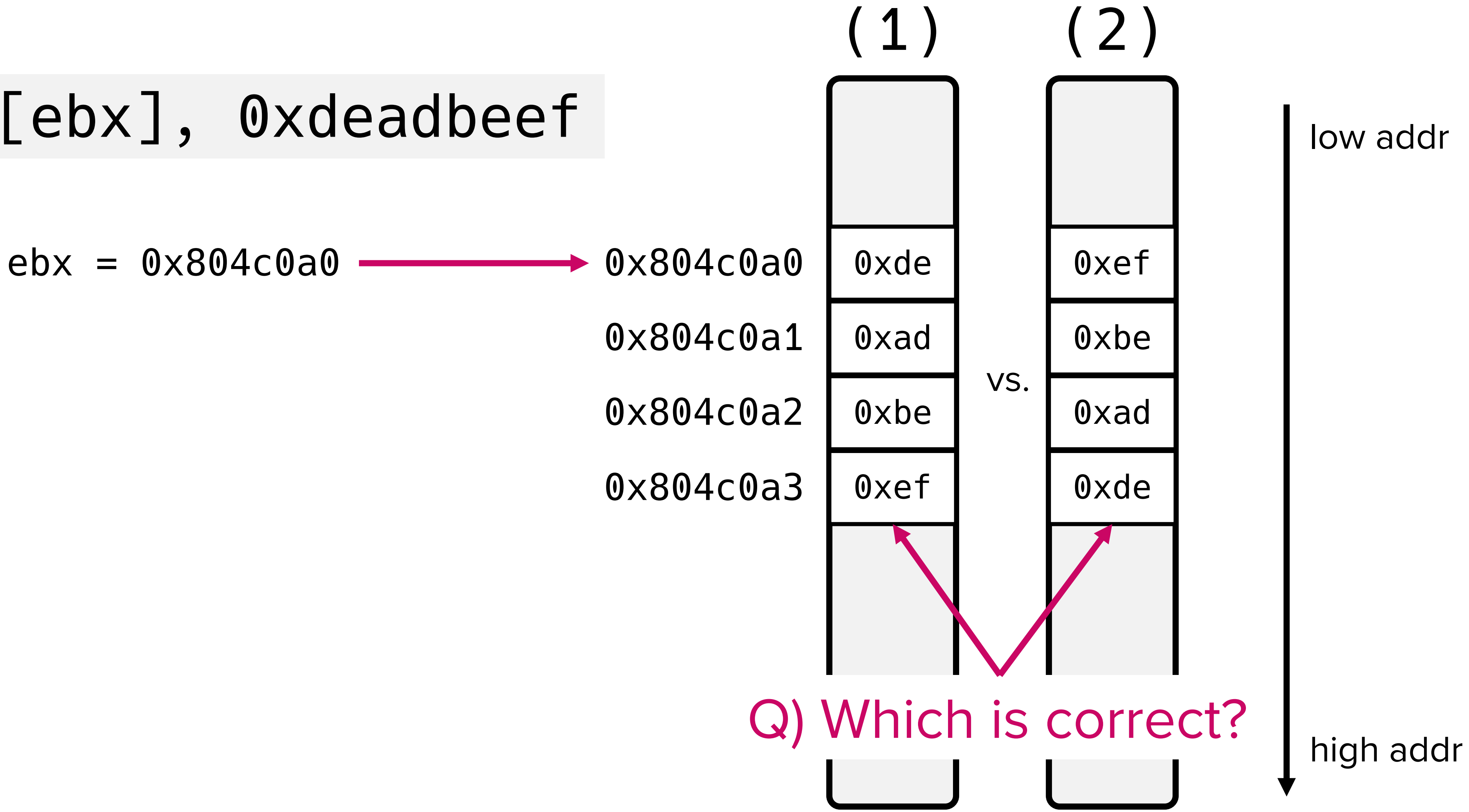
0x02	0xff	0xff	0xff

0x02	0x00	0xff	0xff

0x02	0x00	0x00	0x00

Endianness

```
mov  [ebx], 0xdeadbeef
```



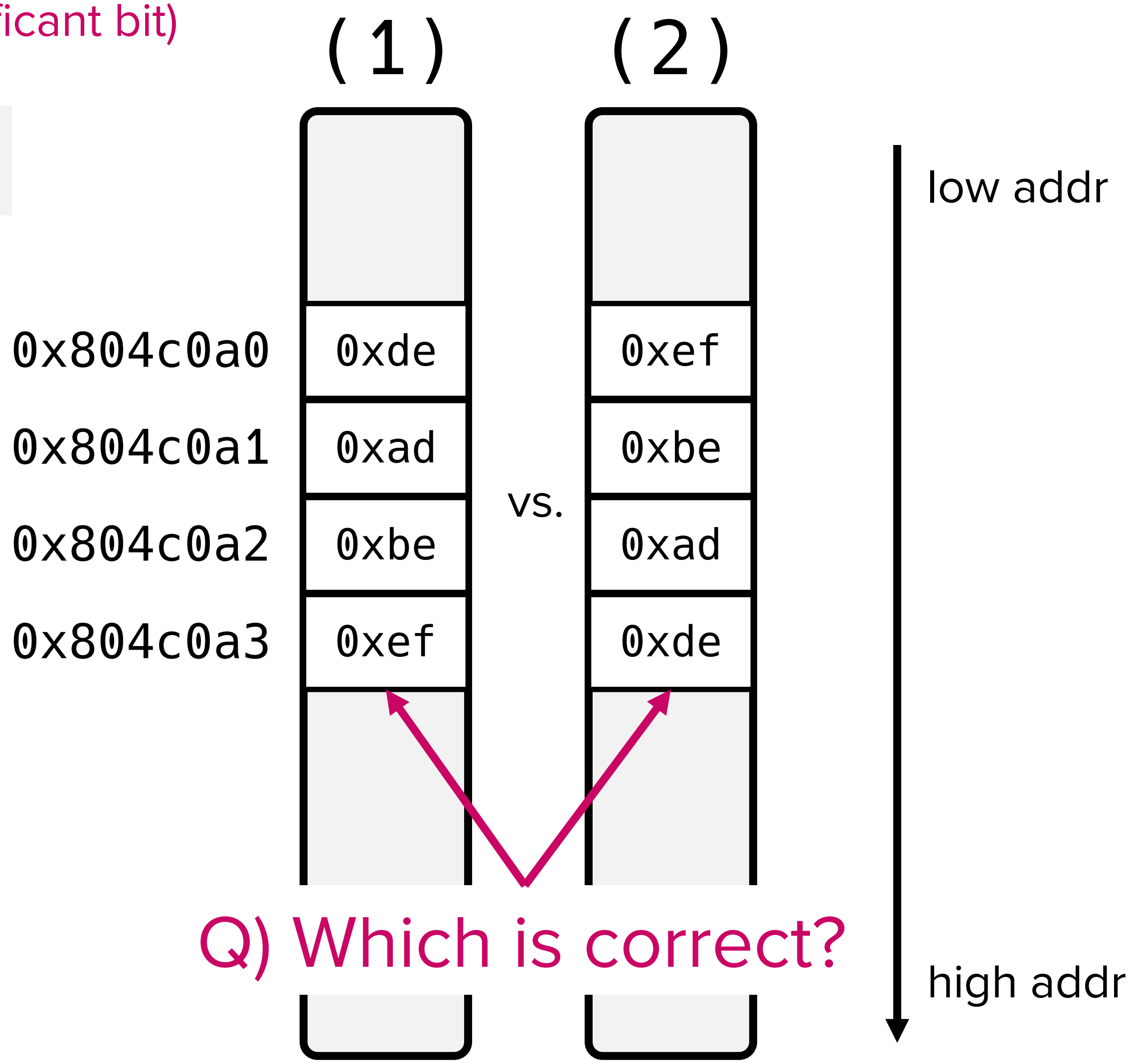
Endianness: How a sequence of bytes is stored in memory

MSB (most significant bit)

```
mov  [ebx], 0xdeadbeef
```

LSB (least SB)

`ebx = 0x804c0a0`



(1) is correct if Big Endian
- The MSB goes to the lowest address

(2) is correct if Little Endian
- The LSB goes to the lowest address

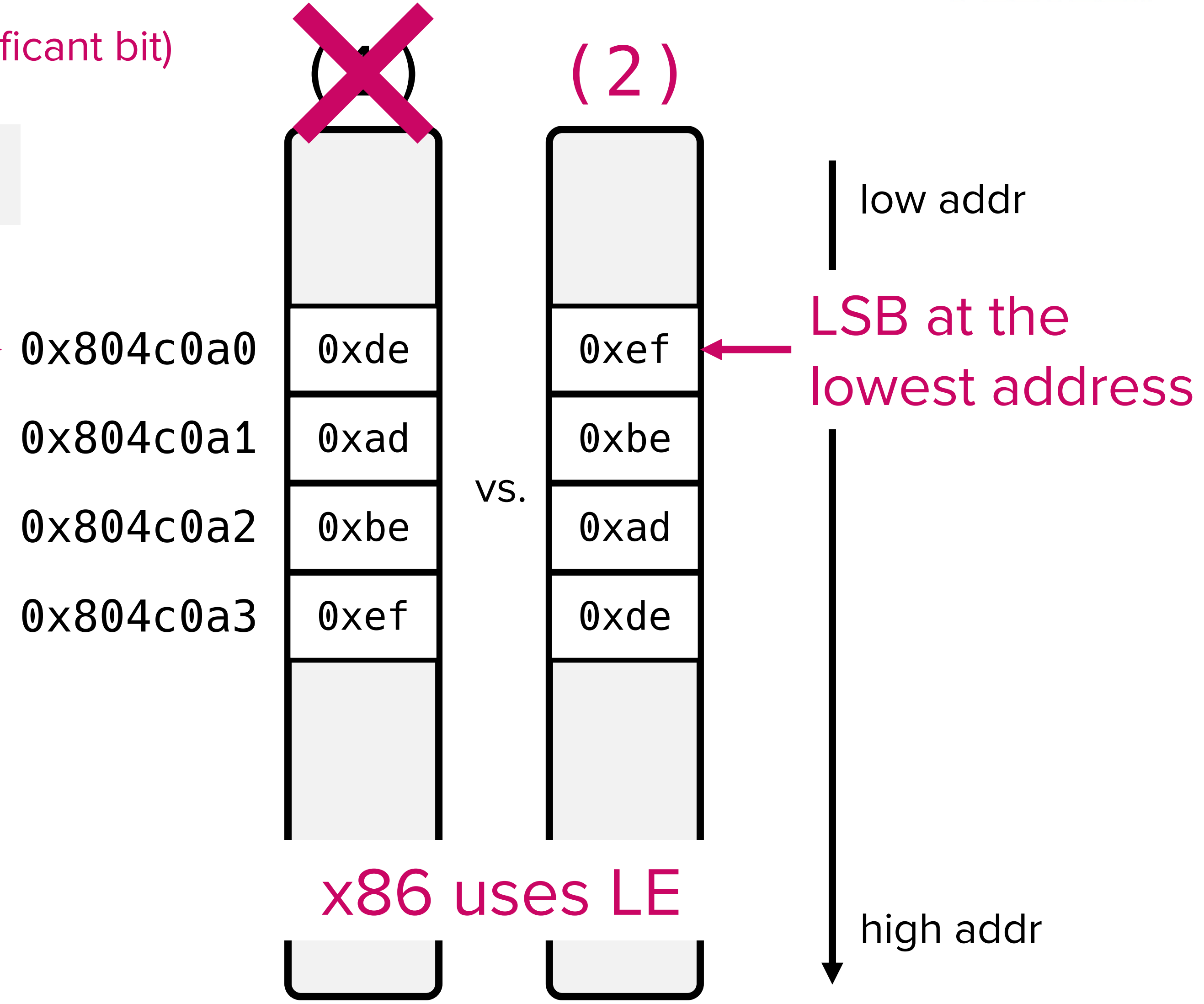
Endianness: x86 uses Little Endian (LE)

MSB (most significant bit)

```
mov  [ebx], 0xdeadbeef
```

LSB (least SB)

`ebx = 0x804c0a0`



(1) is correct if Big Endian
- The MSB goes to the lowest address

(2) is correct if Little Endian
- The LSB goes to the lowest address

`mov` instruction syntax w/ exercise

- Legal
  1. `mov  reg, reg`
  2. `mov  reg, mem`
  3. `mov  mem, reg`
  4. `mov  reg, const`
  5. `mov  mem, const`

- Illegal
  - `mov  mem, mem`
  - `mov  const, const`

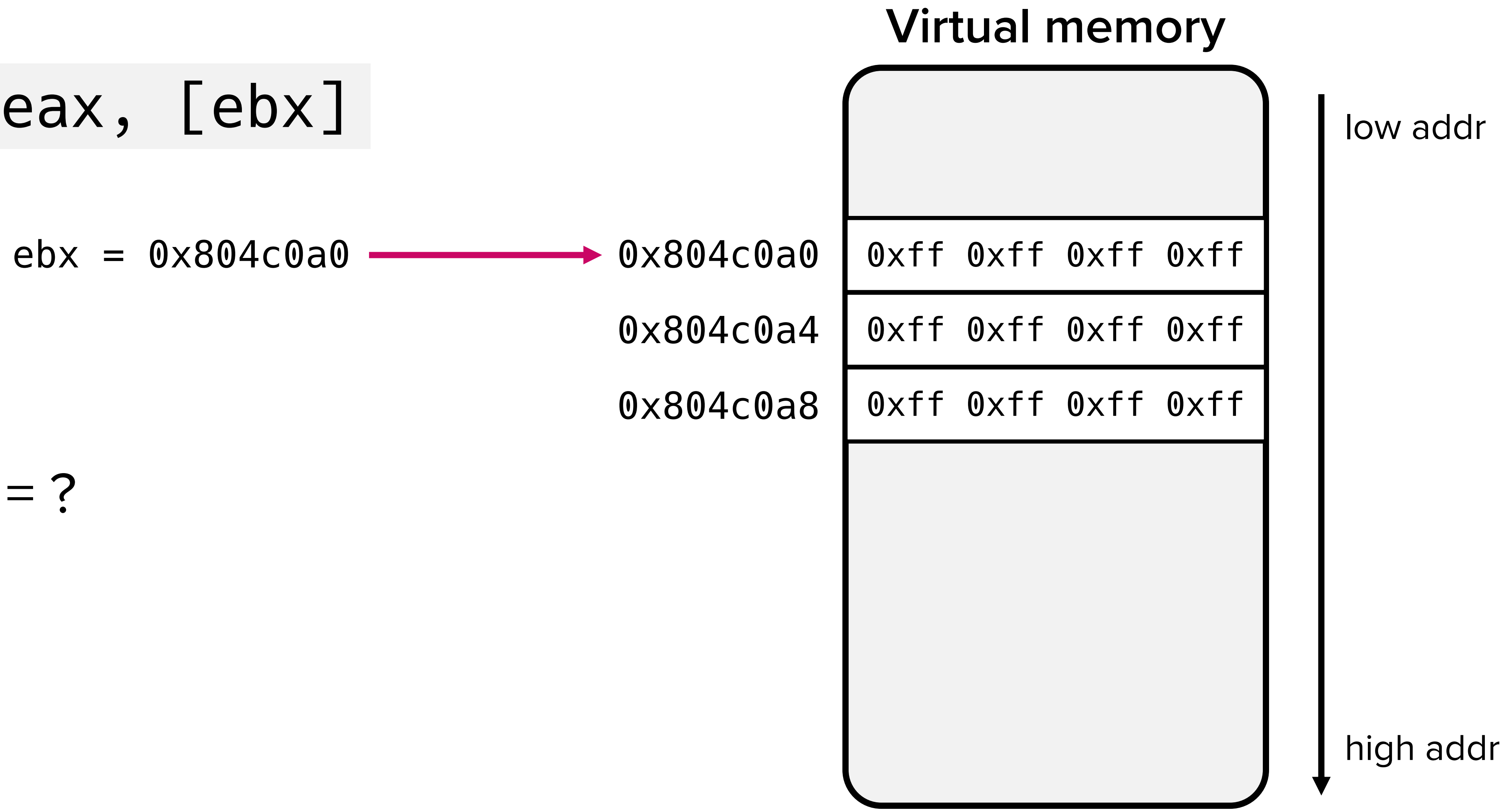
mov instruction syntax w/ exercise

- Legal
  1. `mov reg, reg`
  2. `mov reg, mem`
  3. `mov mem, reg`
  4. `mov reg, const`
  5. `mov mem, const`
- Illegal
  - `mov mem, mem`
  - `mov const, const`

`mov eax, ebx`	Legal (R to R)
`mov al, bl`	Legal (R to R)
`mov [eax], ebx`	Legal (R to M)
`mov eax, [ebx]`	Legal (M to R)
`mov eax, [ebx + edx * 4]`	Legal (M to R)
`mov al, BYTE PTR [esi]`	Legal (M to R)
`mov eax, 16`	Legal (C to R)
`mov [ebx], 16`	Legal (C to M)
`mov [eax], [ebx]`	ILLEGAL (M to M)

`lea`: Load effective address

```
lea  eax, [ebx]
```

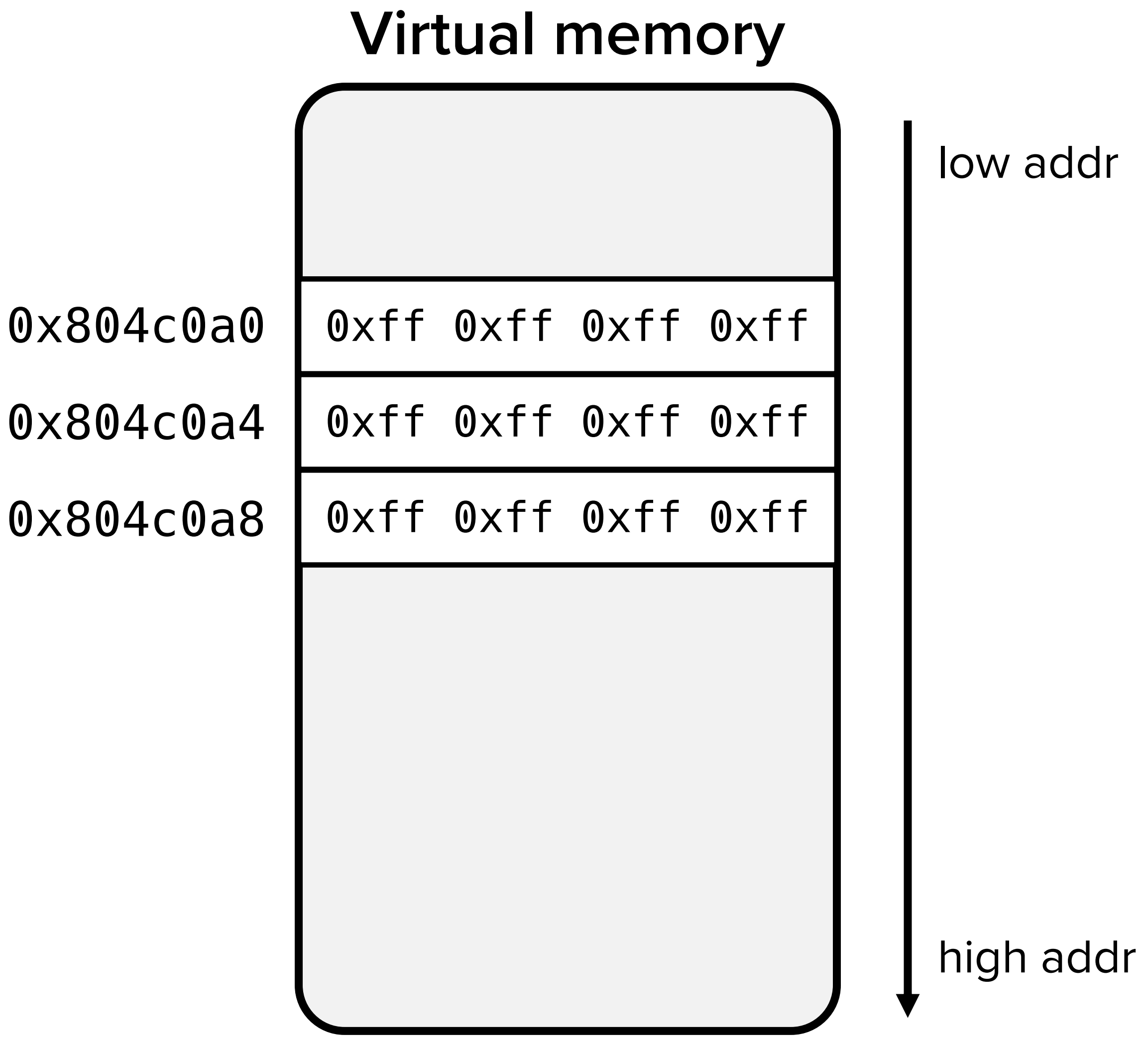


Q) eax = ?

`lea`: Load effective address

```
lea  eax, [ebx]
```

ebx = 0x804c0a0



A) eax = `0x0804c0a0`
The address is loaded!

`mov` vs. `lea`

```
mov   esp, [ebp-0x8]
```

// esp = value that address 0xffffd1450 points to
i.e., `esp = *(ebp - 0x8);`

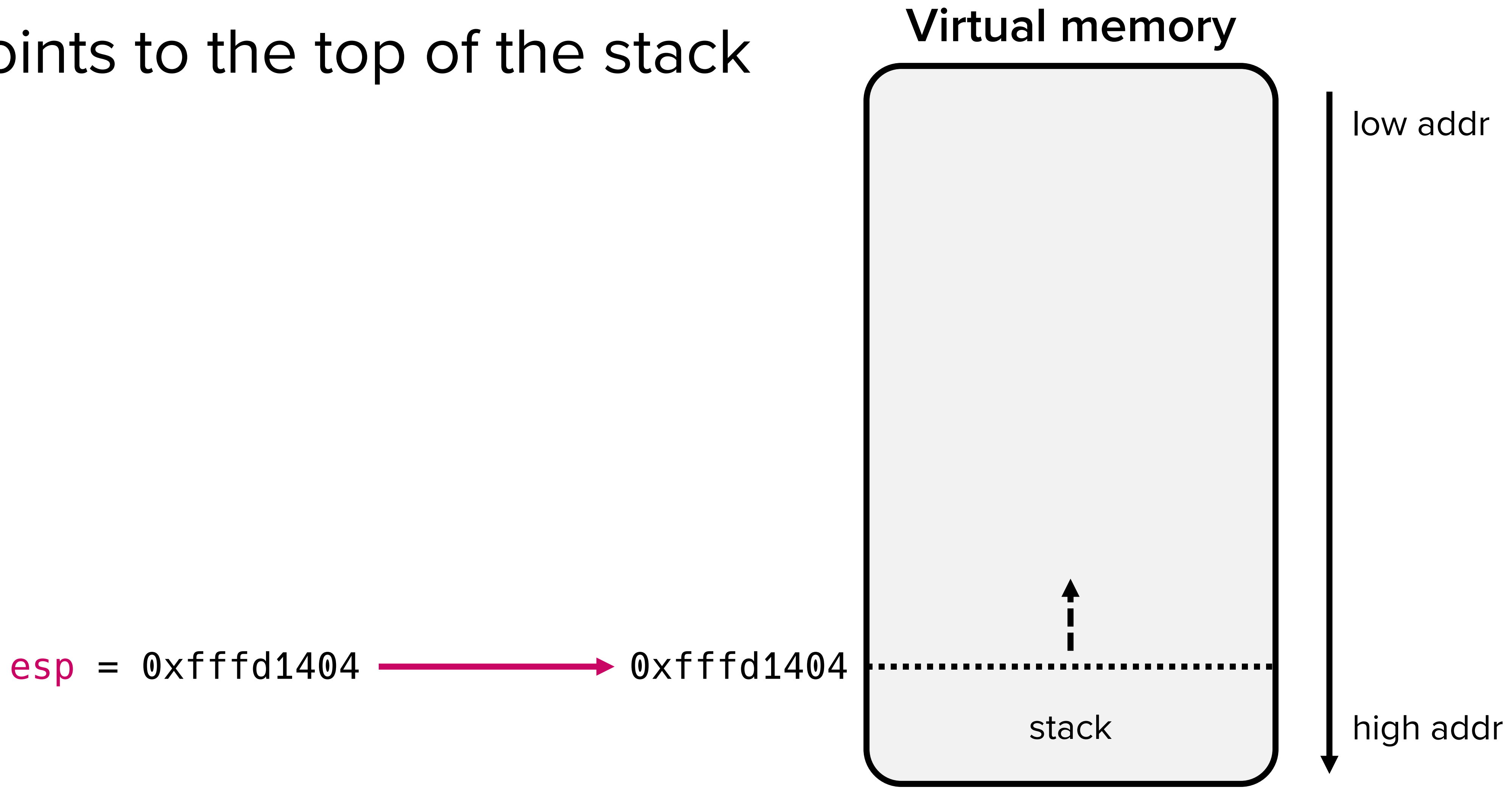
```
lea   esp, [ebp-0x8]
```

// esp = 0xffffd1450
i.e., `esp = (ebp - 0x8);`

```
ebp = 0xfffd1458
```

Stack operations

- `esp` points to the top of the stack



Stack operations: push

- `push` enlarges the stack

```
push  eax
```

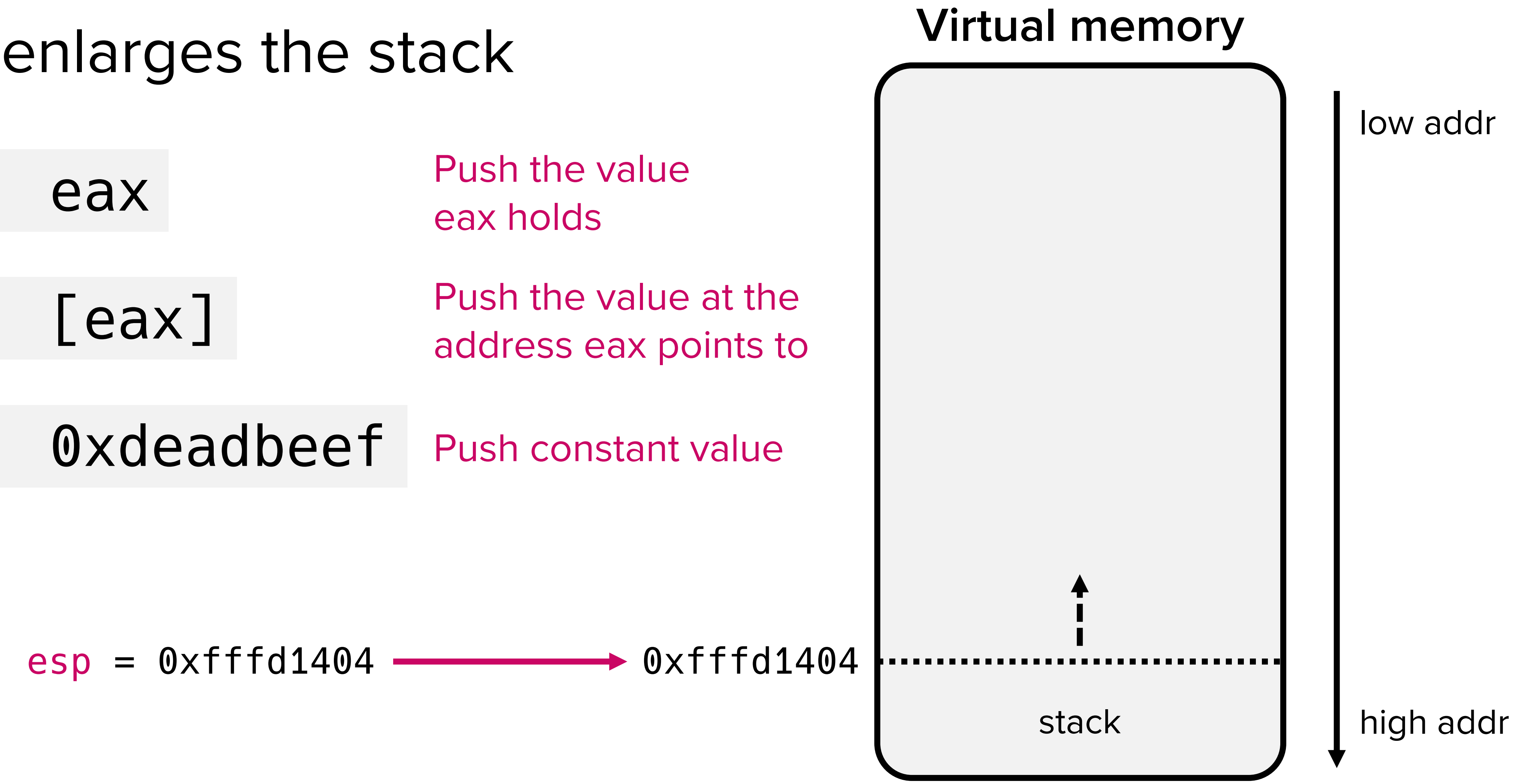
Push the value eax holds

```
push  [eax]
```

Push the value at the address eax points to

```
push  0xdeadbeef
```

Push constant value



Stack operations: push

- `push` enlarges the stack

```
push   eax
```

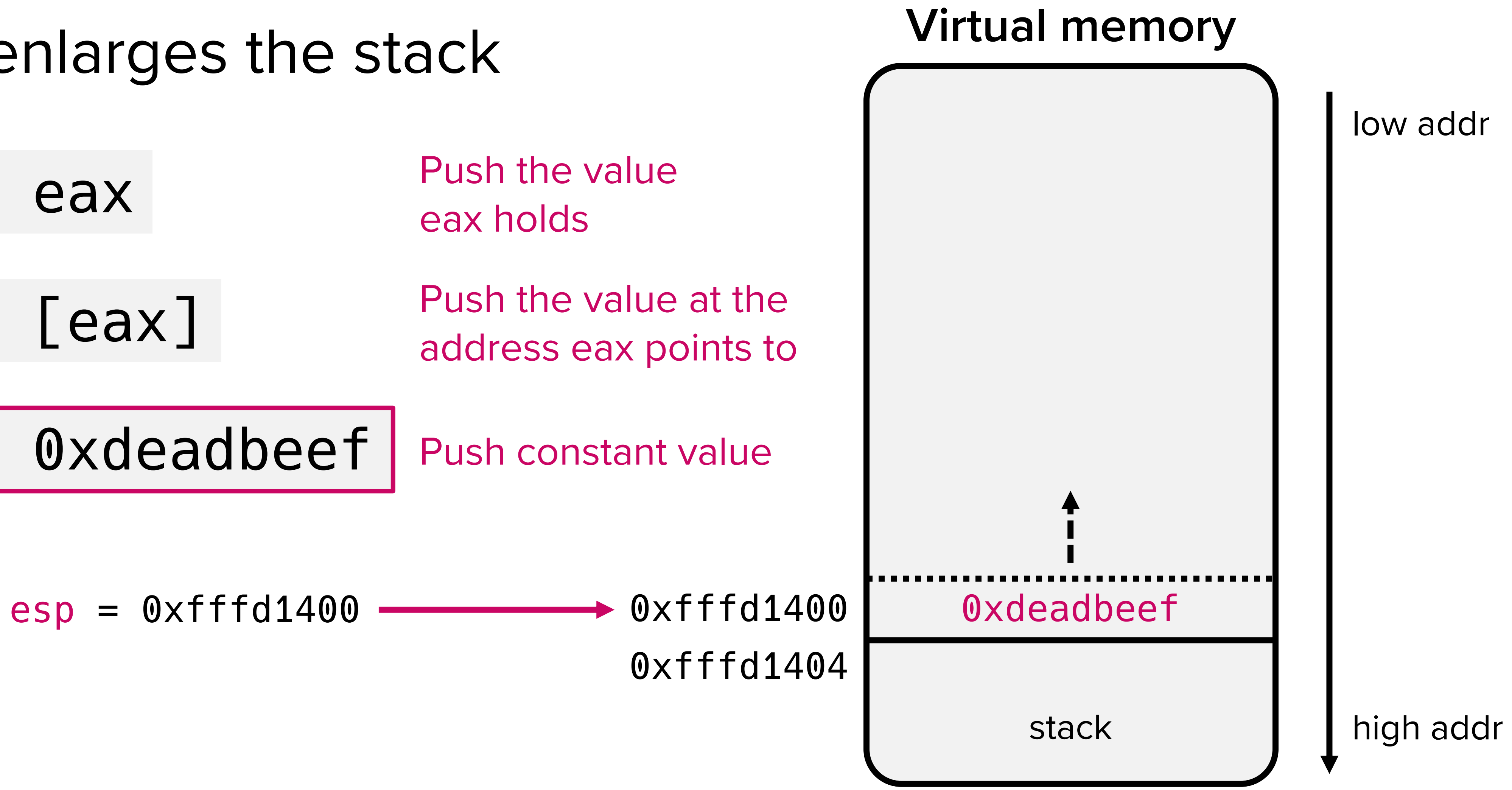
Push the value eax holds

```
push   [eax]
```

Push the value at the address eax points to

```
push   0xdeadbeef
```

Push constant value



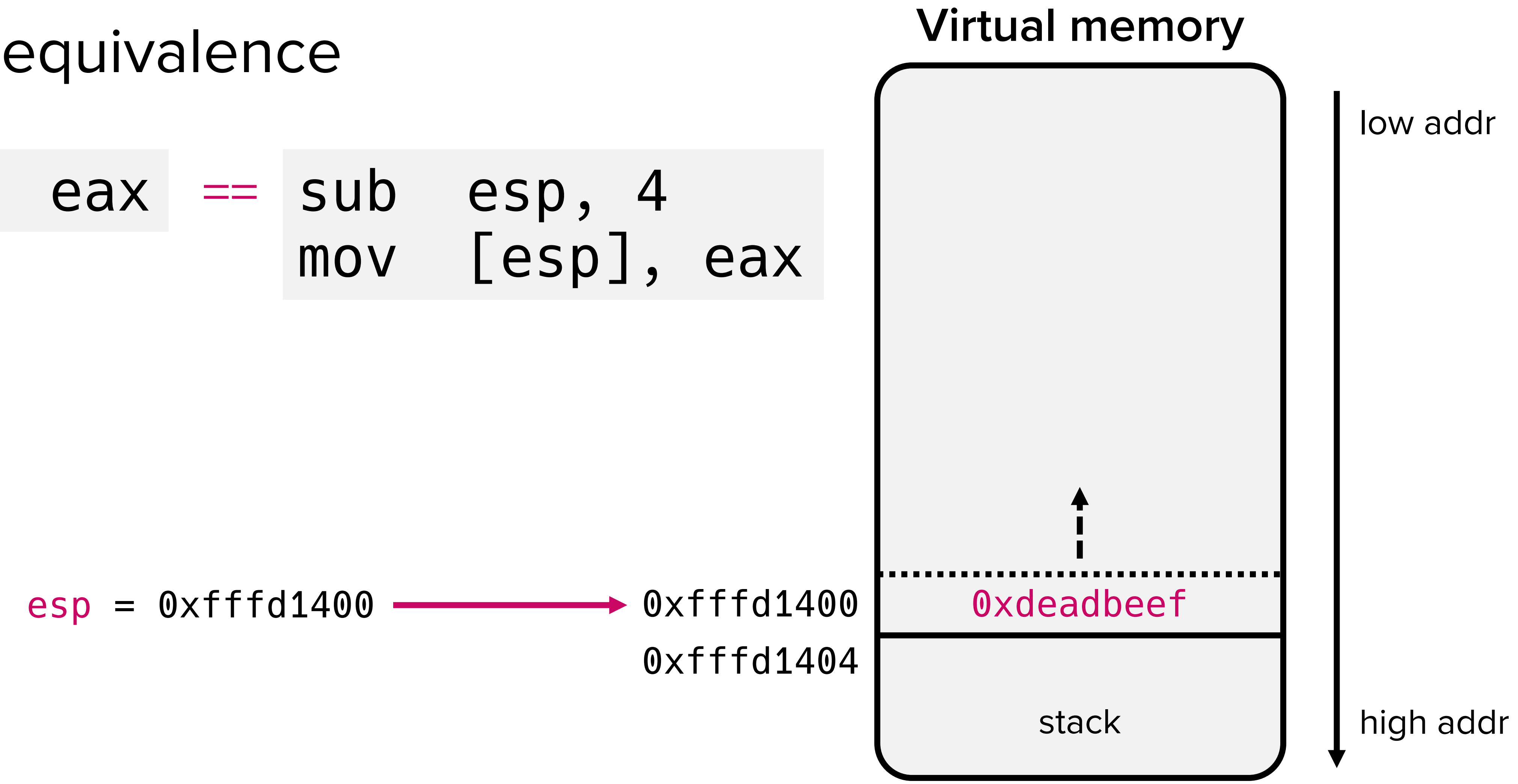
Stack operations: push

- `push` equivalence

```
push  eax
```

==

```
sub  esp, 4
mov  [esp], eax
```



Stack operations: pop

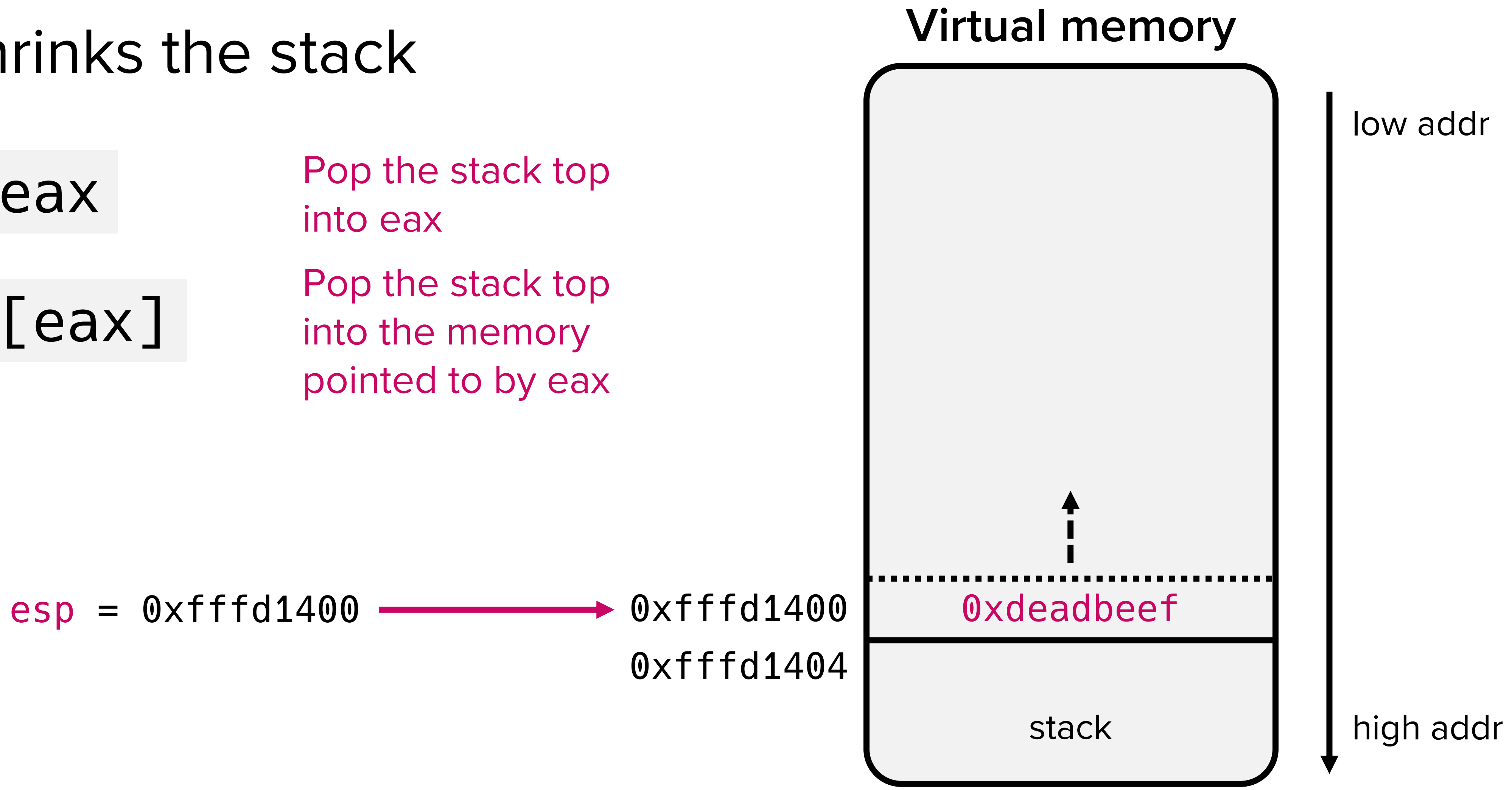
- pop shrinks the stack

```
pop   eax
```

Pop the stack top into eax

```
pop   [eax]
```

Pop the stack top into the memory pointed to by eax



Stack operations: pop

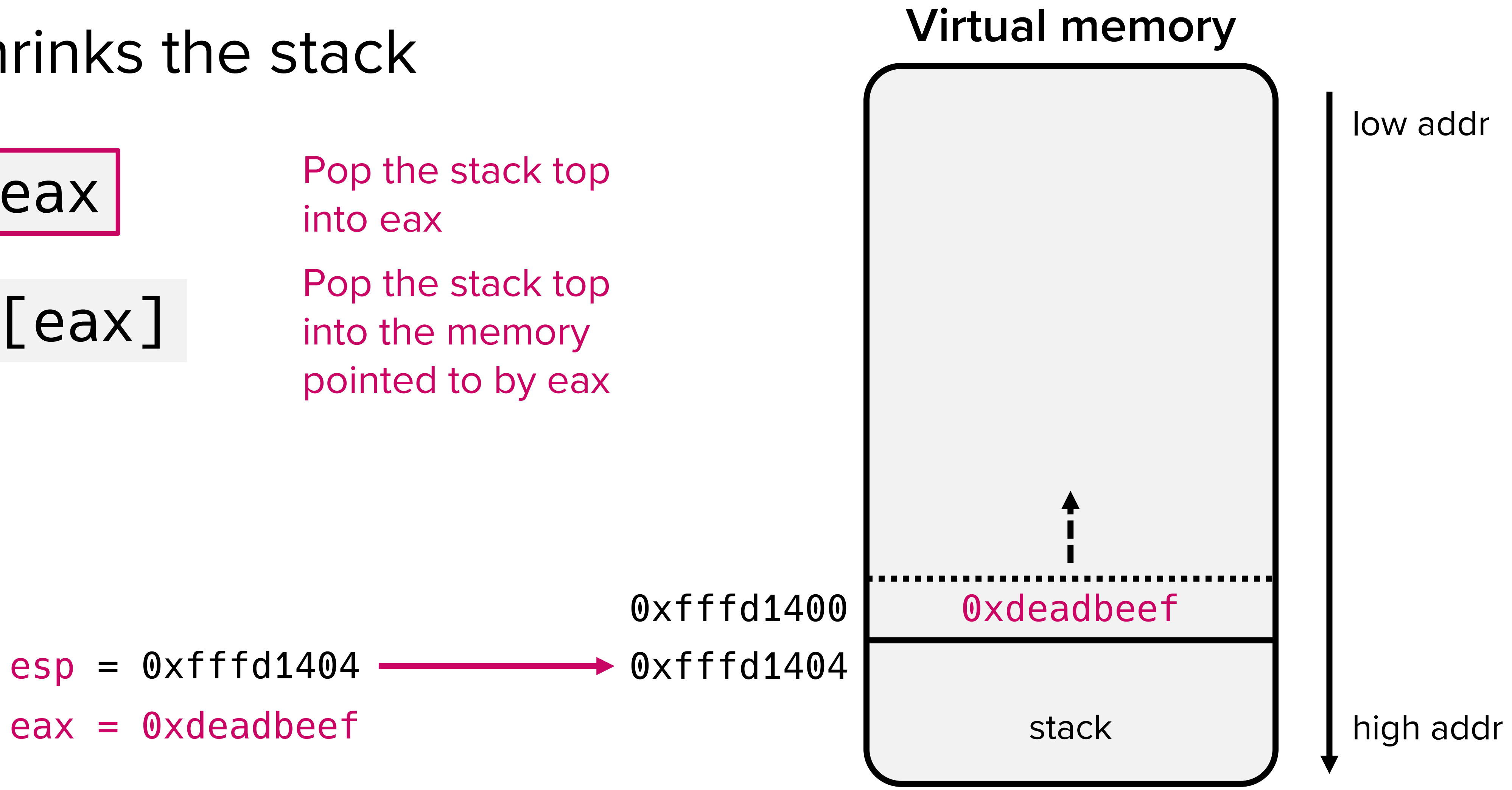
- **pop** shrinks the stack

```
pop   eax
```

Pop the stack top into eax

```
pop   [eax]
```

Pop the stack top into the memory pointed to by eax



Stack operations: pop

- pop equivalence

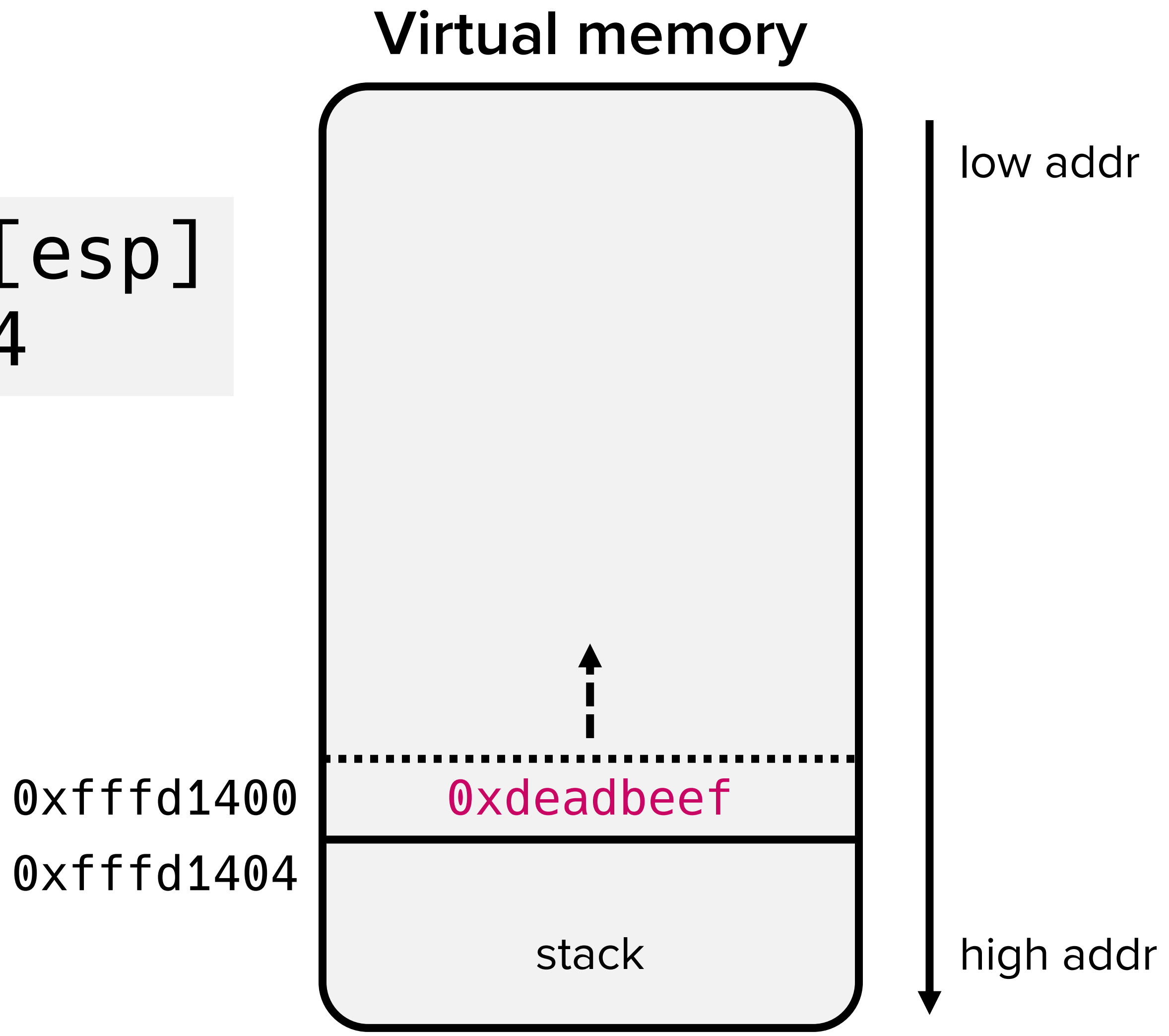
```
pop  eax
```

==

```
mov  eax, [esp]
add  esp, 4
```

`esp = 0xfffd1404`

`eax = 0xdeadbeef`

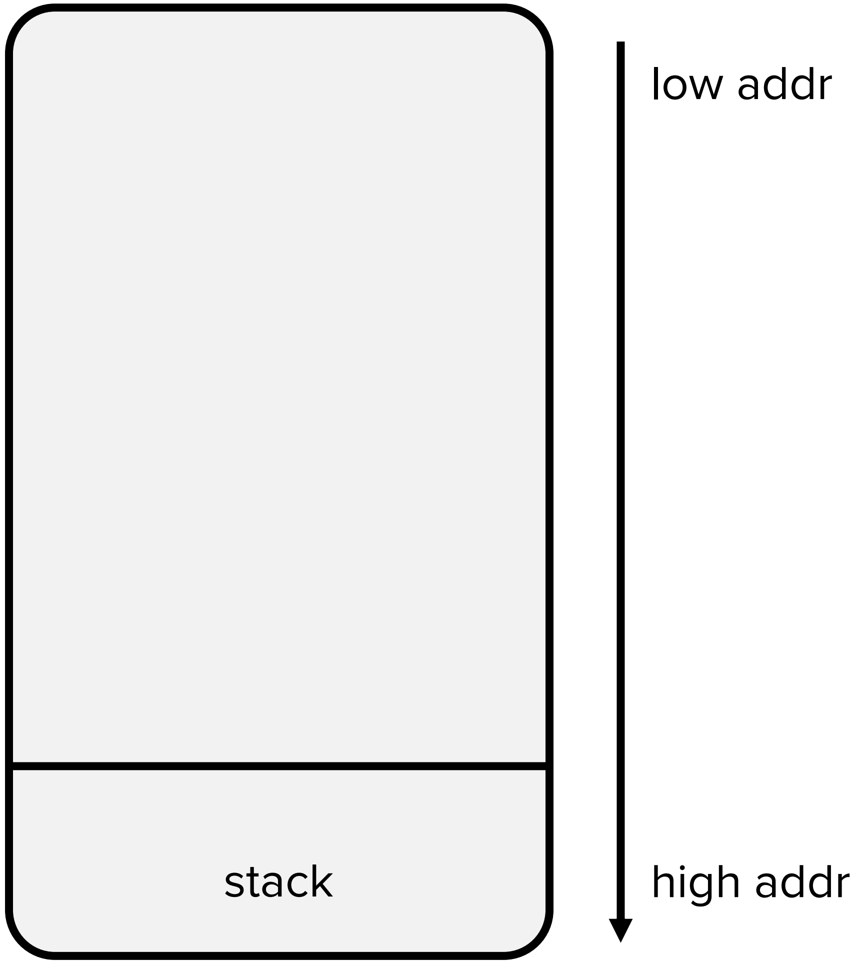


Subroutine instructions: `call`

- `call` <label>
  - `push` the address to return to
  - perform unconditional `jmp` to <label>
  - e.g., in main() of lab01's target

```
eip =>  0x080494d3: call 0x8049385 <phase_one>
        0x080494d8: call 0x8049352 <phase_two>
```

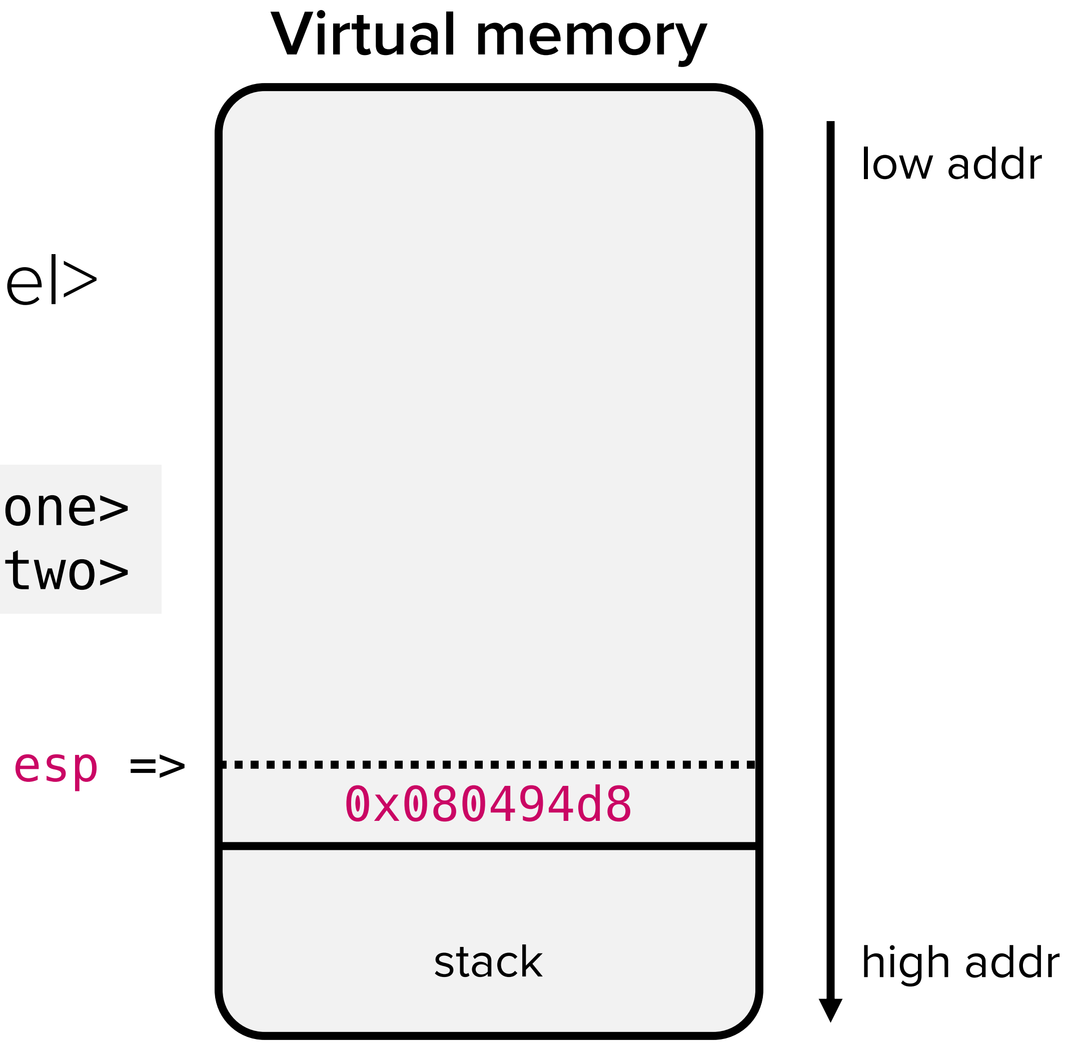
Virtual memory



Subroutine instructions: `call`

- `call` <label>
  - `push` the address to return to
  - perform unconditional `jmp` to <label>
  - e.g., in main() of lab01's target

```
eip => 0x080494d3: call 0x8049385 <phase_one>
       0x080494d8: call 0x8049352 <phase_two>
```

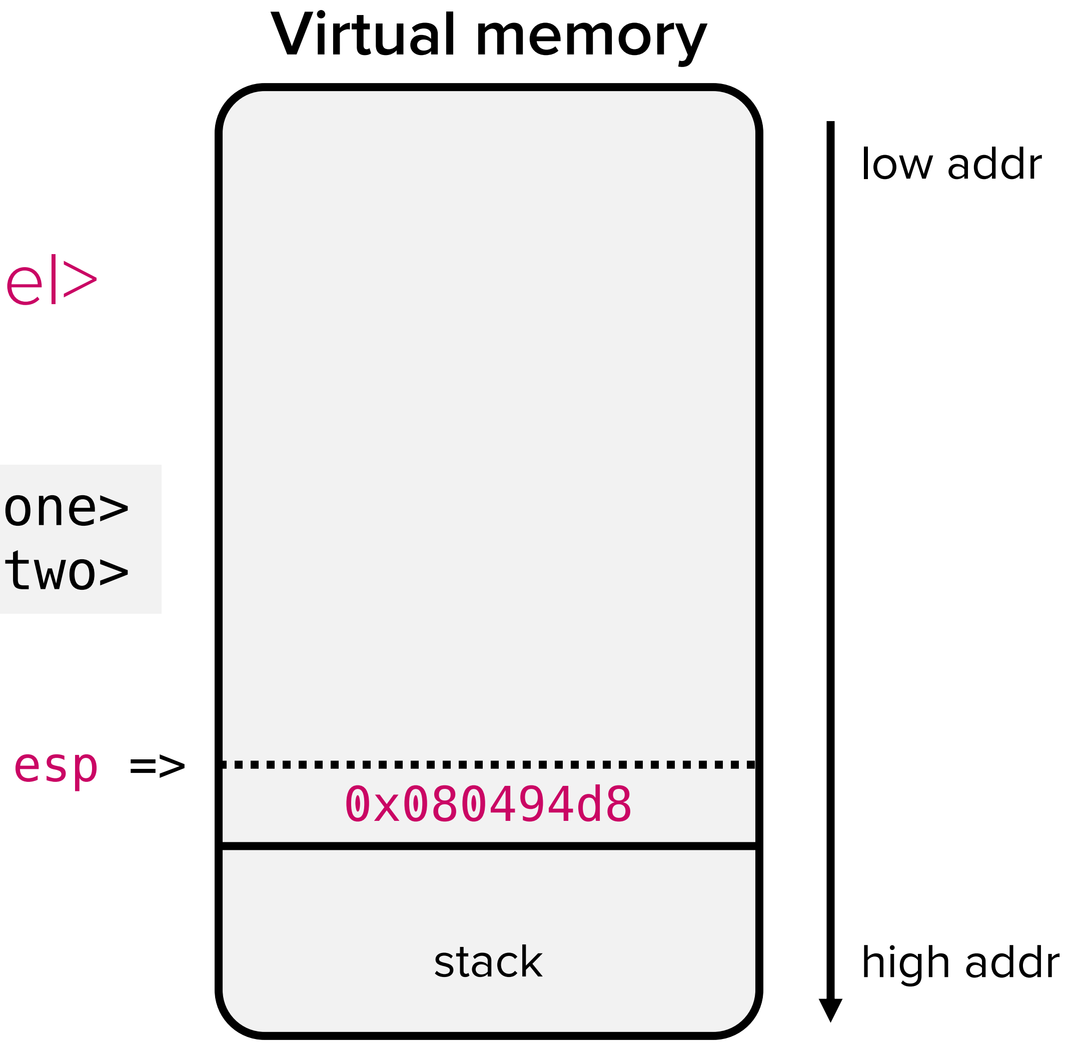


Subroutine instructions: `call`

- `call` <label>
  - `push` the address to return to
  - perform unconditional `jmp` to <label>
  - e.g., in main() of lab01's target

```
0x080494d3: call 0x8049385 <phase_one>
0x080494d8: call 0x8049352 <phase_two>
```

```
       <phase_one>
eip => 0x08049385: push ebp
       0x08049386: mov  ebp, esp
       ...
```



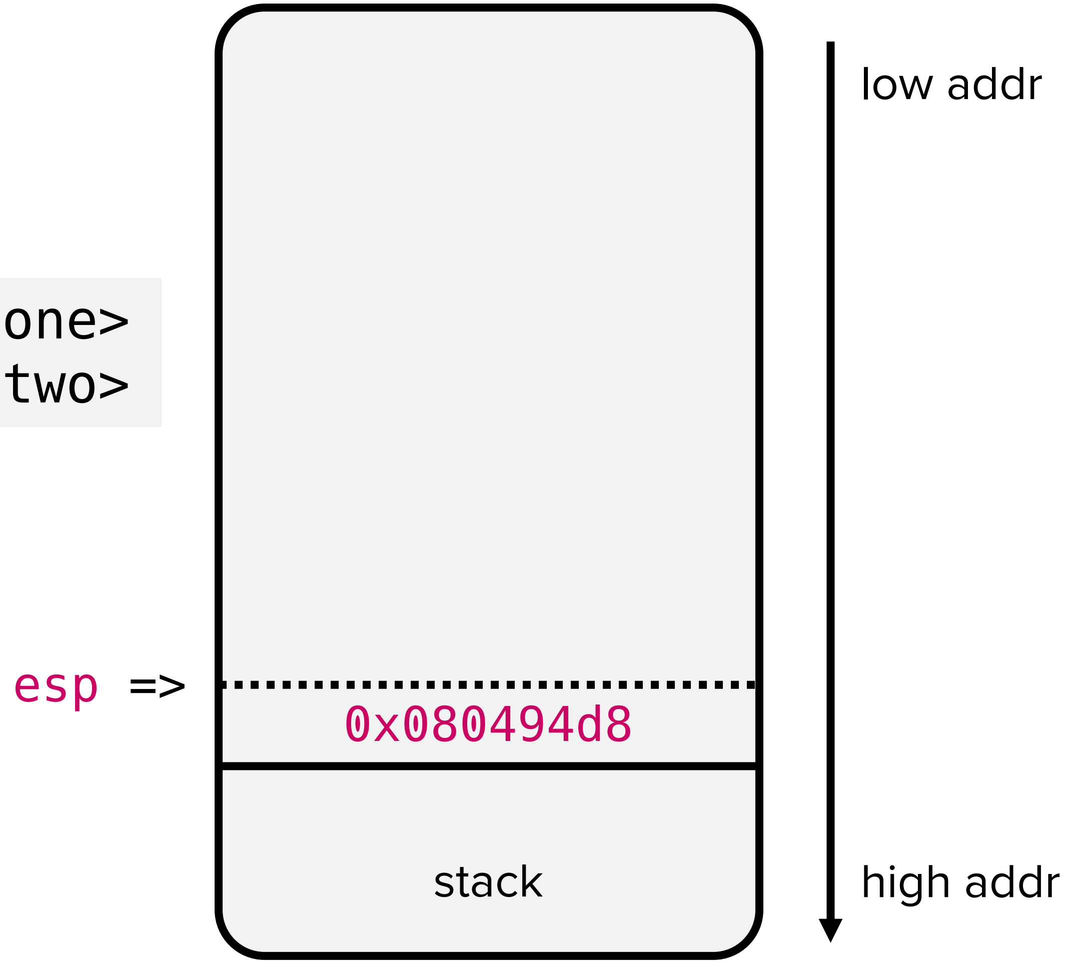
Subroutine instructions: `ret`

- `ret`
  - `pop` stack top into eip
  - e.g., in main() of lab01's target

```
0x080494d3: call 0x8049385 <phase_one>
0x080494d8: call 0x8049352 <phase_two>
```

```
       <phase_one>
       0x08049385: push ebp
       0x08049386: mov  ebp, esp
       ...
eip => 0x08049482: ret
```

Virtual memory

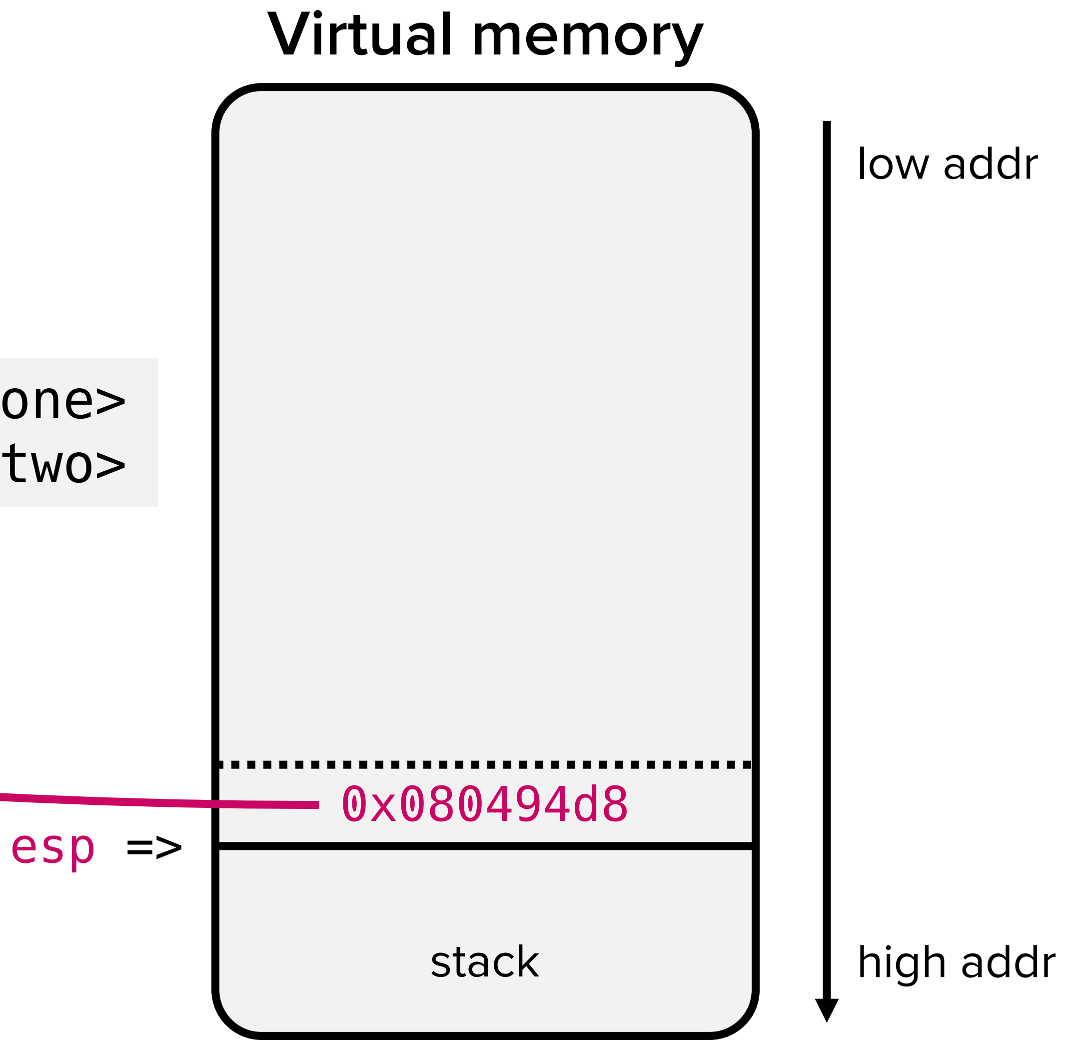


Subroutine instructions: ret

- ret
  - **pop** stack top into eip
  - e.g., in main() of lab01's target

```
         0x080494d3: call 0x8049385 <phase_one>
eip =>   0x080494d8: call 0x8049352 <phase_two>
```

```
<phase_one>
0x08049385: push ebp
0x08049386: mov  ebp, esp
...
0x08049482: ret
```



Branches in assembly

- Implemented using `cmp` + conditional jump

```
cmp   eax, 0xdeadbeef
je    addr
```

```
if (eax == 0xdeadbeef) {
  goto addr;
}
```

```
cmp   eax, 0xdeadbeef
jne   addr
```

```
if (eax != 0xdeadbeef) {
  goto addr;
}
```

```
cmp   eax, 0xdeadbeef
jge   addr
```

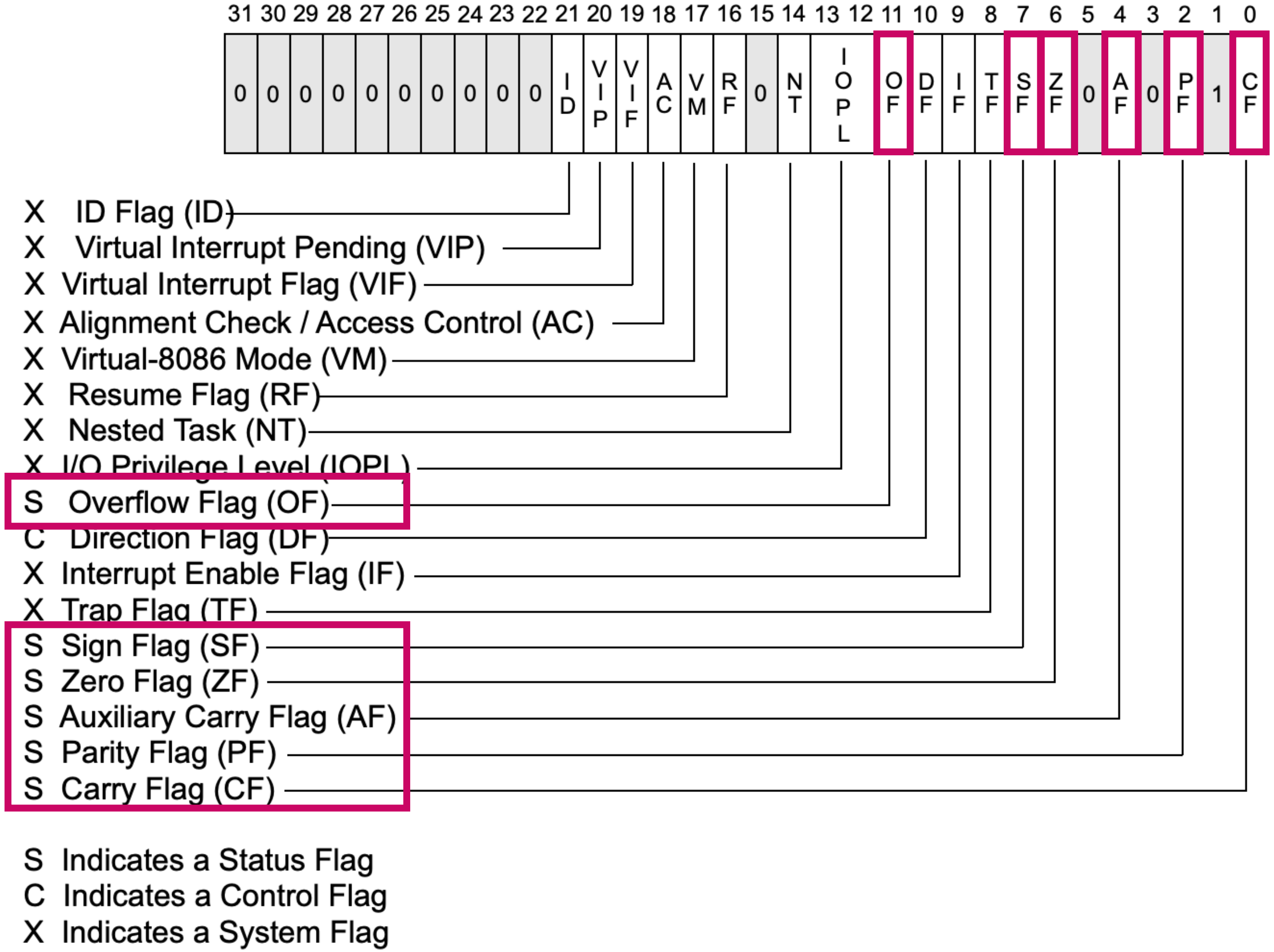
```
if (eax >= 0xdeadbeef) {
  goto addr;
}
```

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

x86 registers

- General purpose registers (GPR)
  - EAX, EBX, ECX, EDX
- Pointers
  - ESI, EDI
- Stack pointers
  - ESP, EBP
- Special purpose registers
  - EIP, EFLAGS ← This guy!

EFLAGS register stores status / control / system flags



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

Conditional jumps and their conditions

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

Summary: With assembly, we can

- Move data around
  - Load/store data from/to data and registers
  - Compute pointer addresses
  - Use stack as a buffer (push, pop)
- Perform arithmetic operations
  - add, sub, and, xor, …
- Call and return from functions
  - Push return address on call, pop that on ret
- Control execution flow
  - cmp followed by conditional jump family

Turing complete!

Given enough time and memory, can solve any computational problem

Exercise: reading lab01's target

- Focus:
  - mov operation
  - push and pop
  - Function call and return
  - cmp and jumps

Summary

- Do not get intimidated by assembly code!
  - They ARE human-readable

Coming up next

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

Questions?