Assembly Language

CS245 Spring '19 — Kris Micinski

Logistics

- Out of town rest of week
- Lab 2 available on Wed/Thur
- Due the Tuesday after that
- **This week's lab**
  - Small bits of assembly code, but no function calls, etc..
- **Next week's project**
  - Covers function calls / calling conventions, etc.. also includes stack smashing

Assembly is that language spoken by the **processor**

Just as there are many processors, there are many different types of assembly

In this class, we will learn x86-64

(Also ARM, MIPS, SPARC, etc… but "x86" is most common, and x86-64 is 64-bit variant of x86)

Reading

- Required:
  - http://ian.seyler.me/easy_x86-64/
  - https://www.cs.cmu.edu/~fp/courses/15213-s07/misc/asm64-handout.pdf
  - http://nickdesaulniers.github.io/blog/2014/04/18/lets-write-some-x86-64/
- Optional (but strongly encouraged)
  - https://cs.brown.edu/courses/cs033/docs/guides/x64_cheatsheet.pdf
  - https://www3.nd.edu/~dthain/courses/cse40243/fall2015/intel-intro.html

Basics

- Assembly code consists of **instruction sequences**
  - Grouped into "functions" (procedures)
- Each instruction does a very simple task (add, multiply, jump)
- There are a limited number of variables (registers)
  - x86-64 has 16 of these! 2 hold pointers to stack (rsp/rbp)
- If you need more memory (e.g., for storing an array), must store in stack / heap / etc…

```
.data
_hello:
  .asciz "Hello, world!\n"

.text
.globl _main
_main:
  subq $8, %rsp

  movb $0, %rax
  leaq _hello(%rip), %rdi
  call _printf

  movq $0, %rdi
  call _exit
```

This is not **code**, you are telling the processor to put some **data** somewhere and name it `_hello`

```
.data
_hello:
  .asciz "Hello, world!\n"

.text
.globl _main
_main:
  subq $8, %rsp

  movb $0, %rax
  leaq _hello(%rip), %rdi
  call _printf

  movq $0, %rdi
  call _exit
```

```
.data
_hello:
  .asciz "Hello, world!\n"

.text
.globl _main
_main:
  subq $8, %rsp

  movb $0, %rax
  leaq _hello(%rip), %rdi
  call _printf

  movq $0, %rdi
  call _exit
```

Commands starting with dots (.) are directives that tell the assembler how to lay out your program

```
.data
_hello:
  .asciz "Hello, world!\n"

.text
.globl _main
_main:
  subq $8, %rsp

  movb $0, %rax
  leaq _hello(%rip), %rdi
  call _printf

  movq $0, %rdi
  call _exit
```

.text says “put this in the text segment”

```
.data
_hello:
  .asciz "Hello, world!\n"

.text
.globl _main
_main:
  subq $8, %rsp

  movb $0, %rax
  leaq _hello(%rip), %rdi
  call _printf

  movq $0, %rdi
  call _exit
```

.globl means "make this global"

```
.data
_hello:
  .asciz "Hello, world!\n"

.text
.globl _main
_main:
  subq $8, %rsp

  movb $0, %rax
  leaq _hello(%rip), %rdi
  call _printf

  movq $0, %rdi
  call _exit
```

OS **assumes** you will have a function named _main

```
.data
_hello:
  .asciz "Hello, world!\n"

.text
.globl _main
_main:
  subq $8, %rsp

  movb $0, %rax
  leaq _hello(%rip), %rdi
  call _printf

  movq $0, %rdi
  call _exit
```

The %rsp variable (register) is a 64-bit pointer to the stack.

Remember, the stack grows **down**

First command **subtracts 8** from %rsp

This “allocates” 8 bytes on the stack, so that our program can store data there.

This is complicated! More on it later

```
.data
_hello:
  .asciz "Hello, world!\n"

.text
.globl _main
_main:
  subq $8, %rsp

  movb $0, %rax
  leaq _hello(%rip), %rdi
  call _printf

  movq $0, %rdi
  call _exit
```

Moves 0 into %rax (general purpose 8-bit register)

Note:
opcode, source, dest

(This is the convention in AT&T syntax)

```
.data
_hello:
  .asciz "Hello, world!\n"

.text
.globl _main
_main:
  subq $8, %rsp

  movb $0, %rax
  leaq _hello(%rip), %rdi
  call _printf

  movq $0, %rdi
  call _exit
```

Loads address of `_hello` into rdi

`printf` is a special “variadic” function, so # extra arguments has to be put into rax

```
.data
_hello:
  .asciz "Hello, world!\n"

.text
.globl _main
_main:
  subq $8, %rsp

  movb $0, %rax
  leaq _hello(%rip), %rdi
  call _printf

  movq $0, %rdi
  call _exit
```

Actually performs the function call!

```
.data
_hello:
  .asciz "Hello, world!\n"

.text
.globl _main
_main:
  subq $8, %rsp

  movb $0, %rax
  leaq _hello(%rip), %rdi
  call _printf

  movq $0, %rdi
  call _exit
```

Returns here!

Since we want to call exit(0), need to put 0 in %rdi

Registers

Originally, 8-bit registers: al, bl, cl, dl

Traditionally, x86 architectures only had **four** 16-bit general purpose registers: ax, bx, cx, dx

Also other registers: bp, sp, di, si

Base pointer

(Start of frame)

Stack pointer

(Top of stack)

Originally, 8-bit registers: al, bl, cl, dl

Traditionally, x86 architectures only had **four** 16-bit general purpose registers: ax, bx, cx, dx

Also other registers: bp, sp, di, si

Base pointer

(Start of frame)

Stack pointer

(Top of stack)

IP: instruction pointer

Points at current instruction, incremented after each instruction

FLAGS: holds flags

Set on subtraction, comparison, etc..

Traditionally, x86 architectures only had **four** 16-bit general purpose registers: ax, bx, cx, dx

Also other registers: bp, sp, di, si

As time progressed, also added 32-bit registers: eax, ebx, ecx, edx

In past decade(s), 64-bit registers: rax, rbx, rcx, rdx

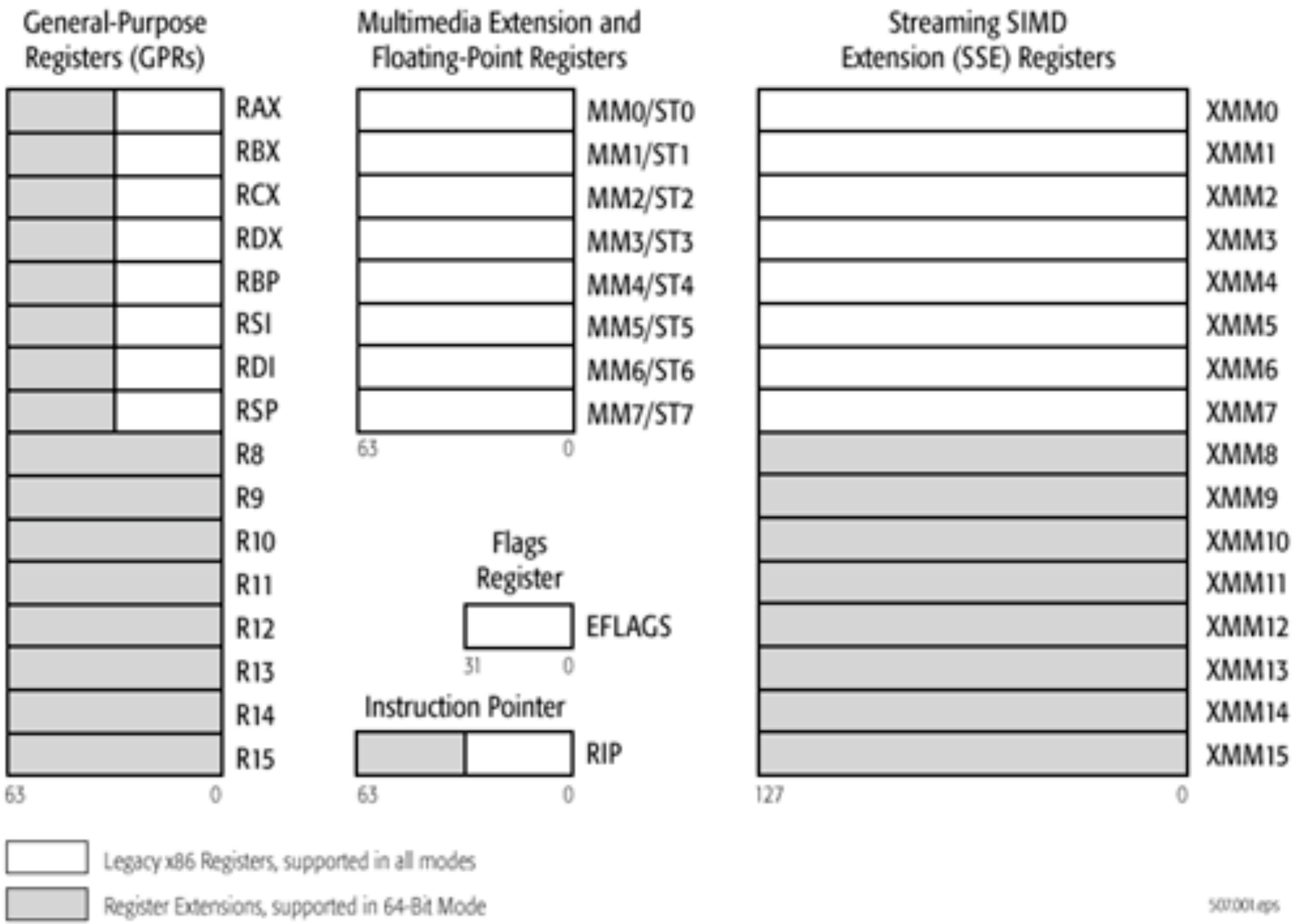
(Also 64-bit versions: rip, etc..)

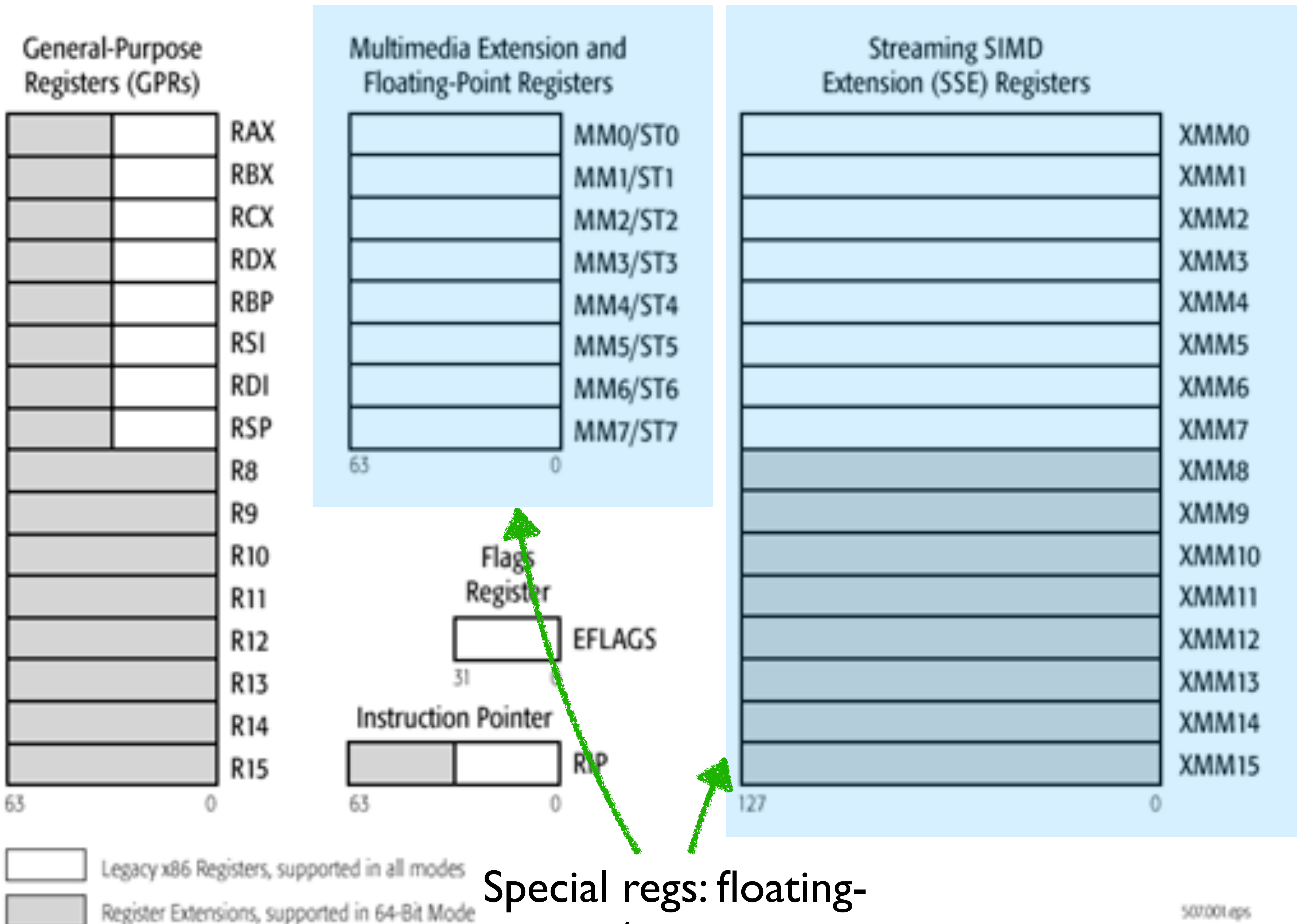
We'll pretty much exclusively use 64-bit registers!

Note RAX is an **extension** of EAX

If you change EAX, you change lower 32 bits of RAX

- Today we mostly have 64-bit ISAs
- 32-bit is still used (e.g., int in C++ on my machine)
- 16-bit basically gone, 8-bit totally gone
    - Some code out there that still works with the 32/16/8 bit registers, though
- Most assembly instructions have **suffixes** that denote which bit-width they're working on
    - E.g., movq says "move into a **quadword** (8-bytes)"
    - movl says "move into a **long** (4-bytes)"
    - These names are largely vestigial and don't mean much


General-Purpose Registers (GPRs)
RAX
RBX
RCX
RDX
RBP
RSI
RDI
RSP
R8
R9
R10
R11
R12
R13
R14
R15
63
0
Multimedia Extension and Floating-Point Registers
MM0/ST0
MM1/ST1
MM2/ST2
MM3/ST3
MM4/ST4
MM5/ST5
MM6/ST6
MM7/ST7
63
0
Flags Register
EFLAGS
31
0
Instruction Pointer
RIP
63
0
Streaming SIMD Extension (SSE) Registers
XMM0
XMM1
XMM2
XMM3
XMM4
XMM5
XMM6
XMM7
XMM8
XMM9
XMM10
XMM11
XMM12
XMM13
XMM14
XMM15
127
0
Legacy x86 Registers, supported in all modes
Register Extensions, supported in 64-Bit Mode
507001.eps



Special regs: floating-point / matrix ops

To represent `0x1234567890abcdef`

12	34	56	78	90	ab	cd	ef

Most Significant Byte — Least Significant Byte x86 is a **little-endian** architecture

If an n-byte value is stored at addresses a to a+(n-1) in memory, byte a will hold the **least significant byte**

`0x1234567890abcdef`

Exercise with partner

Instructions

Binary code is made up of giant sequences of "instructions"

Modern Intel / AMD chip has hundreds of them, some very complex

Moving memory around

Arithmetic

Branch / If

Matrix operations

Atomic-Instructions

Transactional memory instructions

Encoded as binary (as you may have seen from hardware-design course)

We (humans) write in a format named "assembly"

Confusingly: two types of assembly

AT&T	Intel
`mov 5, %rax`	`mov rax, 5`

I will basically always use AT&T

(Since that's what's used in GNU toolchain)

Several **addressing modes**

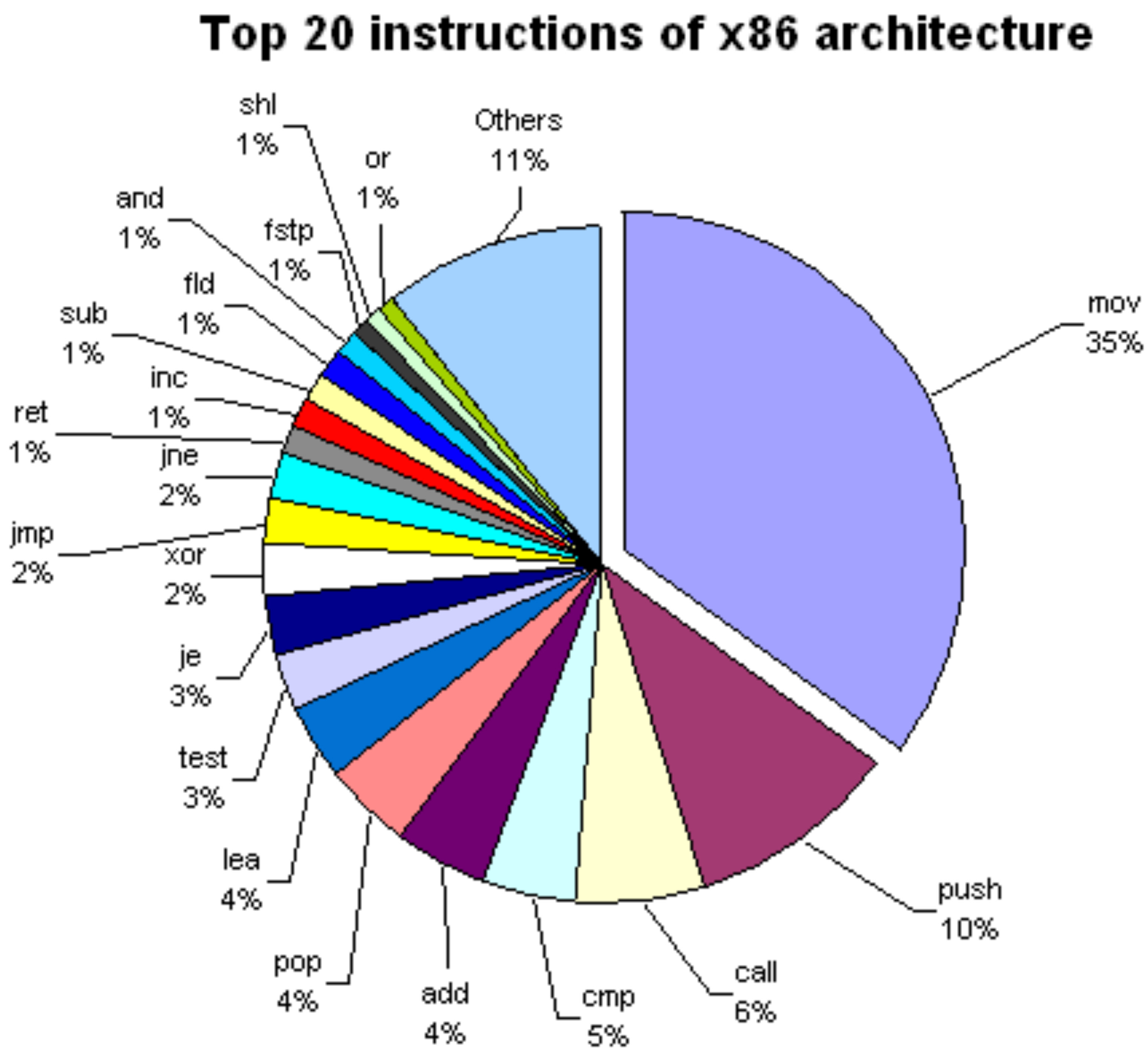
“Move the value from register rax into the register rbx”

Opcode name

Destination

```
mov  %rax, %rbx
```

Source



Plurality of instructions are **mov**s

Then **push**

Then **call**

Example: average two integers

How would you find avg of rax & rbx?

```
.text
.globl _main
_main:
  pushq %rbp
  movq $28, %rbx
  movq $23, %rax
  addq %rbx, %rax
  cqto
  movq $2, %rbx
  idivq %rbx
  movq %rax, %rdi
  call _exit
```

Example: average two integers

How would you find avg of rax & rbx?

```
.text
.globl _main
_main:
  pushq %rbp
  movq $28, %rbx
  movq $23, %rax
  addq %rbx, %rax
  cqto
  movq $2, %rbx
  idivq %rbx
  movq %rax, %rdi
  call _exit
```

Save %rbp onto the stack (need to do this for alignment)

Example: average two integers

How would you find avg of rax & rbx?

```
.text
.globl _main
_main:
  pushq %rbp
  movq $28, %rbx
  movq $23, %rax
  addq %rbx, %rax
  cqto
  movq $2, %rbx
  idivq %rbx
  movq %rax, %rdi
  call _exit
```

Move 28 into rbx

Example: average two integers

How would you find avg of rax & rbx?

```
.text
.globl _main
_main:
  pushq %rbp
  movq $28, %rbx
  movq $23, %rax
  addq %rbx, %rax
  cqto
  movq $2, %rbx
  idivq %rbx
  movq %rax, %rdi
  call _exit
```

Move 23 into rax

Example: average two integers

How would you find avg of rax & rbx?

```
.text
.globl _main
_main:
  pushq %rbp
  movq $28, %rbx
  movq $23, %rax
  addq %rbx, %rax
  cqto
  movq $2, %rbx
  idivq %rbx
  movq %rax, %rdi
  call _exit
```

Add rax to rbx, store result in rax

Example: average two integers

How would you find avg of rax & rbx?

```
.text
.globl _main
_main:
  pushq %rbp
  movq $28, %rbx
  movq $23, %rax
  addq %rbx, %rax
  cqto
  movq $2, %rbx
  idivq %rbx
  movq %rax, %rdi
  call _exit
```

“Sign extend” %rax into %rdx:%rax

(The idivq instructions expects its arguments to be in **both** rdx and rax! So must sign extend!)

Example: average two integers

How would you find avg of rax & rbx?

```
.text
.globl _main
_main:
  pushq %rbp
  movq $28, %rbx
  movq $23, %rax
  addq %rbx, %rax
  cqto
  movq $2, %rbx
  idivq %rbx
  movq %rax, %rdi
  call _exit
```

Move 2 into rbx

Example: average two integers

How would you find avg of rax & rbx?

```
.text
.globl _main
_main:
  pushq %rbp
  movq $28, %rbx
  movq $23, %rax
  addq %rbx, %rax
  cqto
  movq $2, %rbx
  idivq %rbx
  movq %rax, %rdi
  call _exit
```

Divide %rdx:%rax by %rbx

(Since %rdx will be 0 here, this basically means: %rax/%rbx, store result in %rax, remainder stored in %rdx)

Example: average two integers

How would you find avg of rax & rbx?

```
.text
.globl _main
_main:
  pushq %rbp
  movq $28, %rbx
  movq $23, %rax
  addq %rbx, %rax
  cqto
  movq $2, %rbx
  idivq %rbx
  movq %rax, %rdi
  call _exit
```

Move result into %rdi in preparation to call `exit`

Quiz: What would equiv C++ code look like?

```
.text
.globl _main
_main:
  pushq %rbp
  movq $28, %rbx
  movq $23, %rax
  addq %rbx, %rax
  cqto
  movq $2, %rbx
  idivq %rbx
  movq %rax, %rdi
  call _exit
```

Many other solns possible!

```
int x = 28;
int y = 23;
x += y;
x /= 2;
exit(x);
```

Example: finding max of two ints

```
.text
.globl _main
_main:
  pushq %rbp
  movq $8, %rax
  movq $7, %rbx
  cmp %rax, %rbx
  jg _mov_needed
  jmp _no_mov_needed
_mov_needed:
  movq %rbx, %rax
_no_mov_needed:
  movq %rax, %rdi
  call _exit
```

Push %rbp

Move 8 into rax

Move 7 into rbx

Compare rax and rbx

If %rax is greater, go to _mov_needed

Unconditional jump to _no_mov_needed

Label for _mov_needed

Join point for computation

Move rax into rdi to call _exit

Quiz: what would C++ code look like?

```
.text
.globl _main
_main:
  pushq %rbp
  movq $8, %rax
  movq $7, %rbx
  cmp %rax, %rbx
  jg _mov_needed
  jmp _no_mov_needed
_mov_needed:
  movq %rbx, %rax
_no_mov_needed:
  movq %rax, %rdi
  call _exit
```

Quiz: what would C++ code look like?

```
.text
.globl _main
_main:
  pushq %rbp
  movq $8, %rax
  movq $7, %rbx
  cmp %rax, %rbx
  jg _mov_needed
  jmp _no_mov_needed
_mov_needed:
  movq %rbx, %rax
_no_mov_needed:
  movq %rax, %rdi
  call _exit
```

```
int x = 8; // rax
int y = 7; // rbx
if (y > x)
  x = y;
exit(x);
```

Example: using a loop

```
.text
.globl _main
_main:
  pushq %rbp
  movq $1, %rax
  movq $6, %rbx
  movq $0, %rcx
_loop_begin:
  cmp %rcx, %rbx
  je _loop_end
  addq %rax, %rax
  addq $1, %rcx
  jmp _loop_begin
_loop_end:
  movq %rax, %rdi
  call _exit
```

%rax will accumulate a value

%rbx is going to track when to exit

%rcx will count up by one until it hits %rbx

_ starts a **label**

Compare %rcx and %rbx

If previous comparison was =, jump to …

Unconditional jump to _loop_begin

Target of jump after cmp

Move %rax into %rdi to call _exit

```
.text
.globl _main
_main:
  pushq %rbp
  movq $1, %rax
  movq $6, %rbx
  movq $0, %rcx
_loop_begin:
  cmp %rcx, %rbx
  je _loop_end
  addq %rax, %rax
  addq $1, %rcx
  jmp _loop_begin
_loop_end:
  movq %rax, %rdi
  call _exit
```

Quiz: what does this program compute?

```
.text
.globl _main
_main:
  pushq %rbp
  movq $1, %rax
  movq $6, %rbx
  movq $0, %rcx
_loop_begin:
  cmp %rcx, %rbx
  je _loop_end
  addq %rax, %rax
  addq $1, %rcx
  jmp _loop_begin
_loop_end:
  movq %rax, %rdi
  call _exit
```

Quiz: write corresponding C++ code for this

```
.text
.globl _main
_main:
  pushq %rbp
  movq $1, %rax
  movq $6, %rbx
  movq $0, %rcx
_loop_begin:
  cmp %rcx, %rbx
  je _loop_end
  addq %rax, %rax
  addq $1, %rcx
  jmp _loop_begin
_loop_end:
  movq %rax, %rdi
  call _exit
```

```
int x = 0; // rax
int y = 6; // rbx
int z = 0; // rcx
while (y != z) {
  x += x;
  ++z;
}
exit(x);
```

Memory: a **giant chunk of bytes**

You can read from it and write to it in 1/2/4/8/16-byte increments

```
mov  (%rax), %rbx
```

"Move the value **at address** %rax into register %rbx"

Opcode name

Destination

```
mov  (%rax), %rbx
```

Source

%rax	0xffffffff00000000	0xffffffff00000008	0xaf23c8a223356ac
		0xffffffff00000000	0xdeadbeefdeadbeef
%rbx	0x1234123412341234		

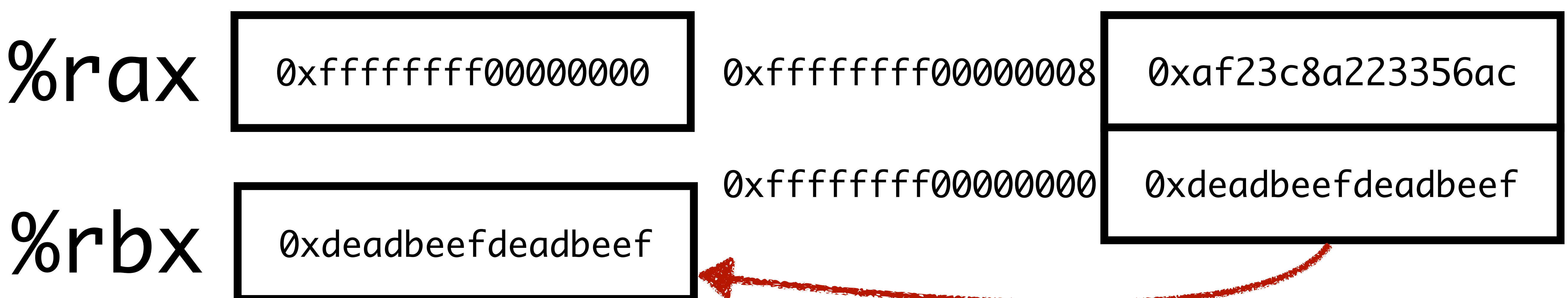
"Move the value **at address** %rax into register %rbx"

Opcode name

Destination

`mov (%rax), %rbx`

Source



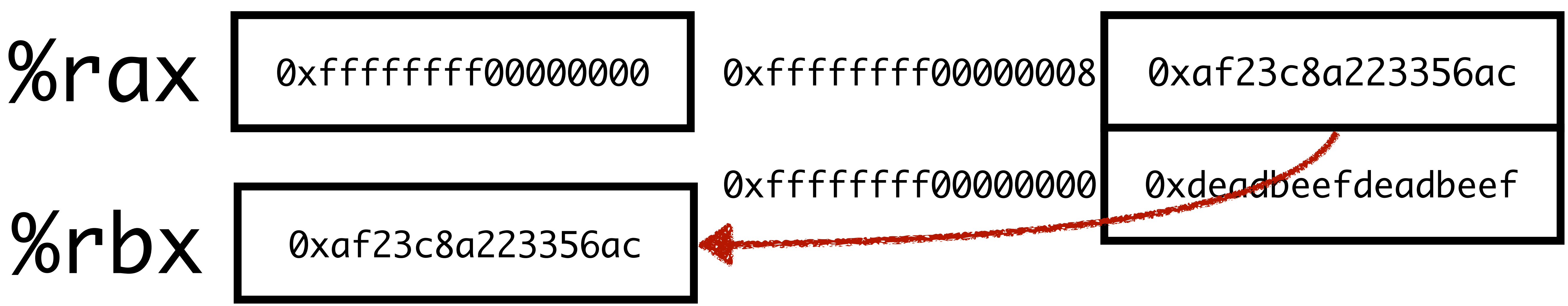
“Move the value **at address** %rax+8 into register %rbx”

Opcode name

Destination

```
mov  8(%rax), %rbx
```

Source



A few other more complicated ones that
allow you to add registers, offsets, etc…

Different instructions allow different addressing-modes

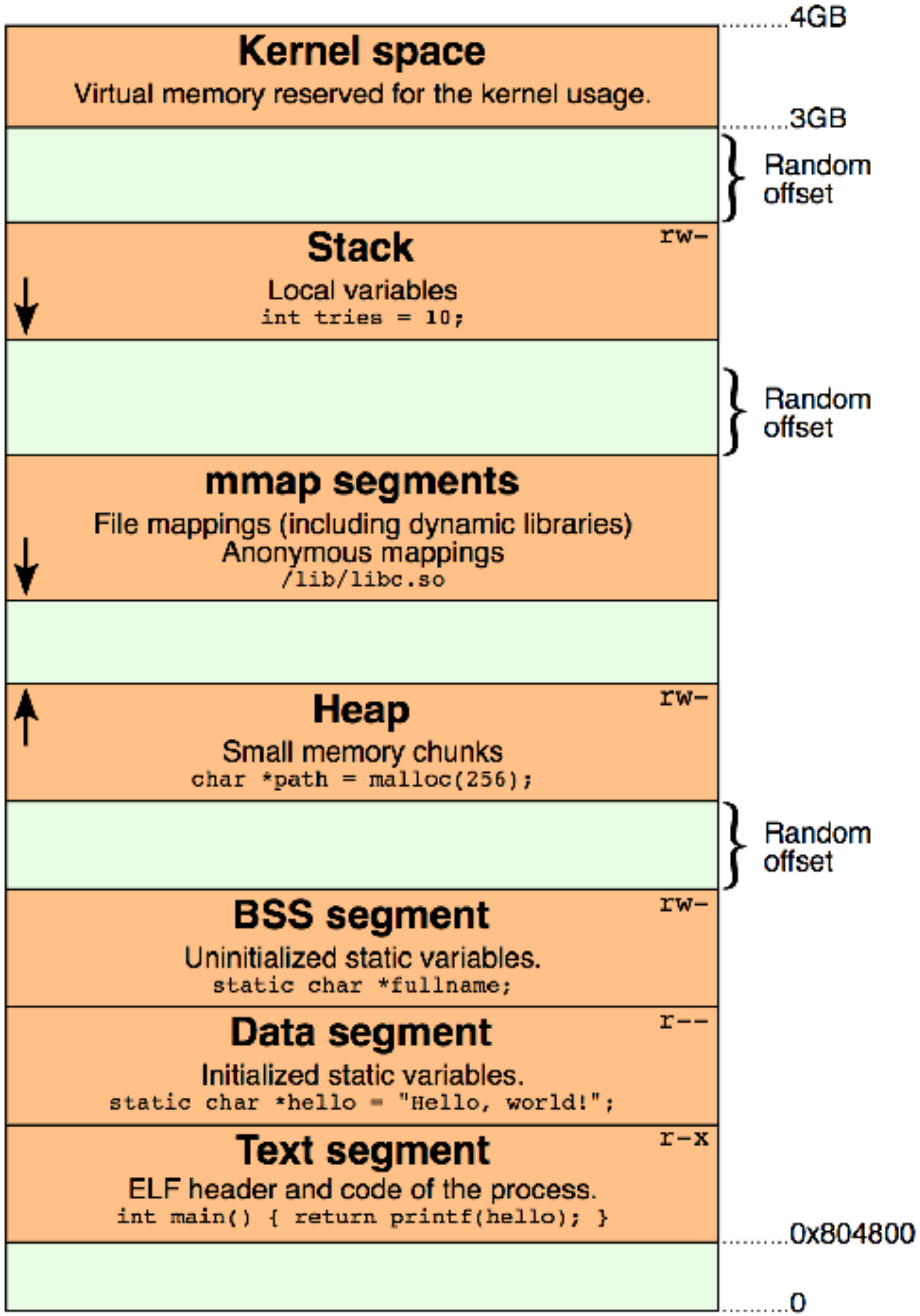
```
# Full example: load *(ebp + (edx * 4) - 8) into eax
movq    -8(%ebp, %edx, 4), %eax
# Typical example: load a stack variable into eax
movq    -4(%ebp), %eax
# No index: copy the target of a pointer into a register
movq    (%ecx), %edx
# Arithmetic: multiply eax by 4 and add 8
leaq    8(,%eax,4), %eax
# Arithmetic: multiply eax by 2 and add edx
leaq    (%edx,%eax,2), %eax
```

```
# Full example: load *(ebp + (edx * 4) - 8) into eax
movq    -8(%ebp, %edx, 4), %eax
# Typical example: load a stack variable into eax
movq    -4(%ebp), %eax
# No index: copy the target of a pointer into a register
movq    (%ecx), %edx
# Arithmetic: multiply eax by 4 and add 8
leaq    8(,%eax,4), %eax
# Arithmetic: multiply eax by 2 and add edx
leaq    (%edx,%eax,2), %eax
```

`leaq` **is “load effective address (quad).” You can think of it as the assembly analogue of C++’s & (address of) operator**

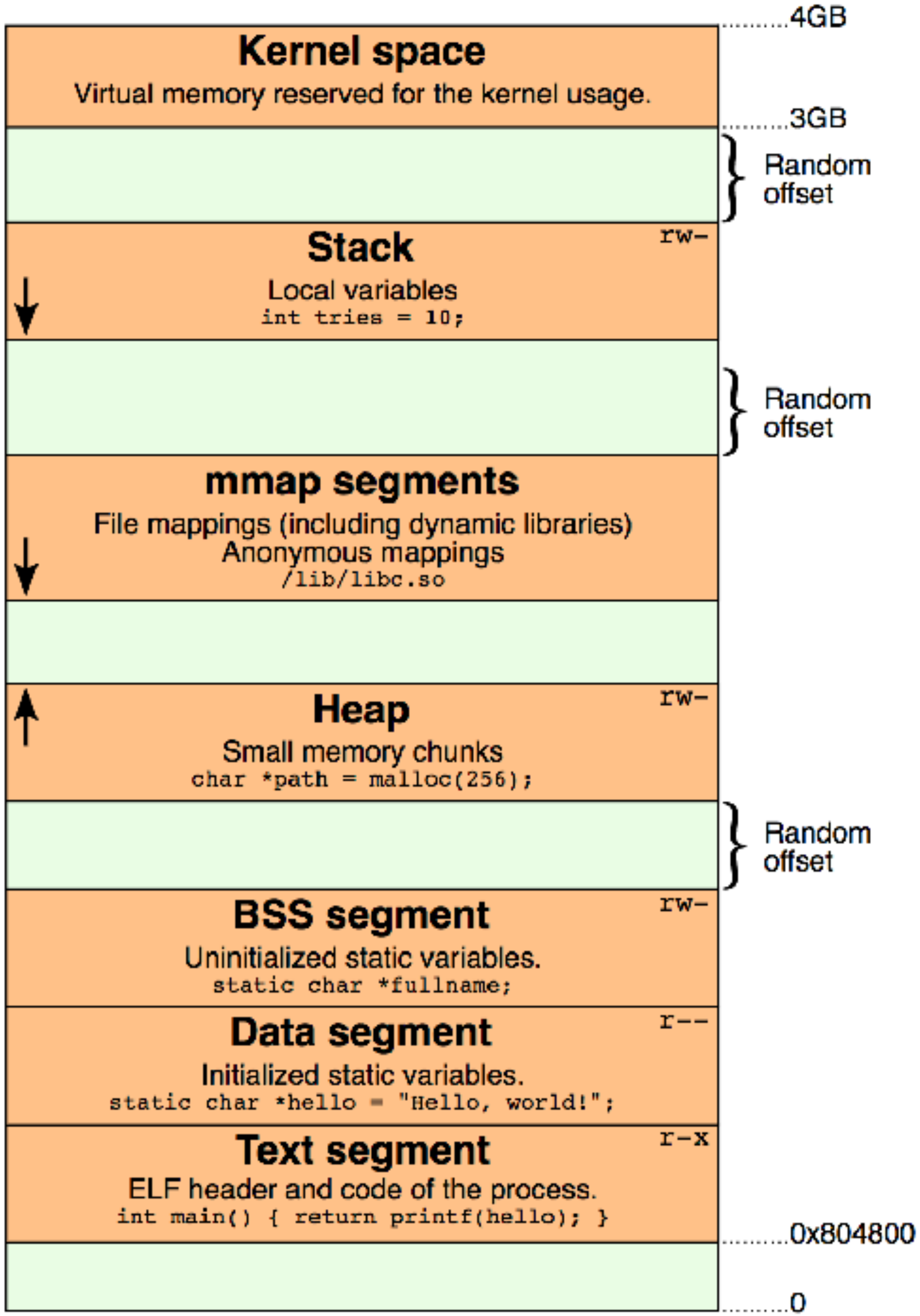
Memory is divided into different regions

Name a few?



Kernel memory

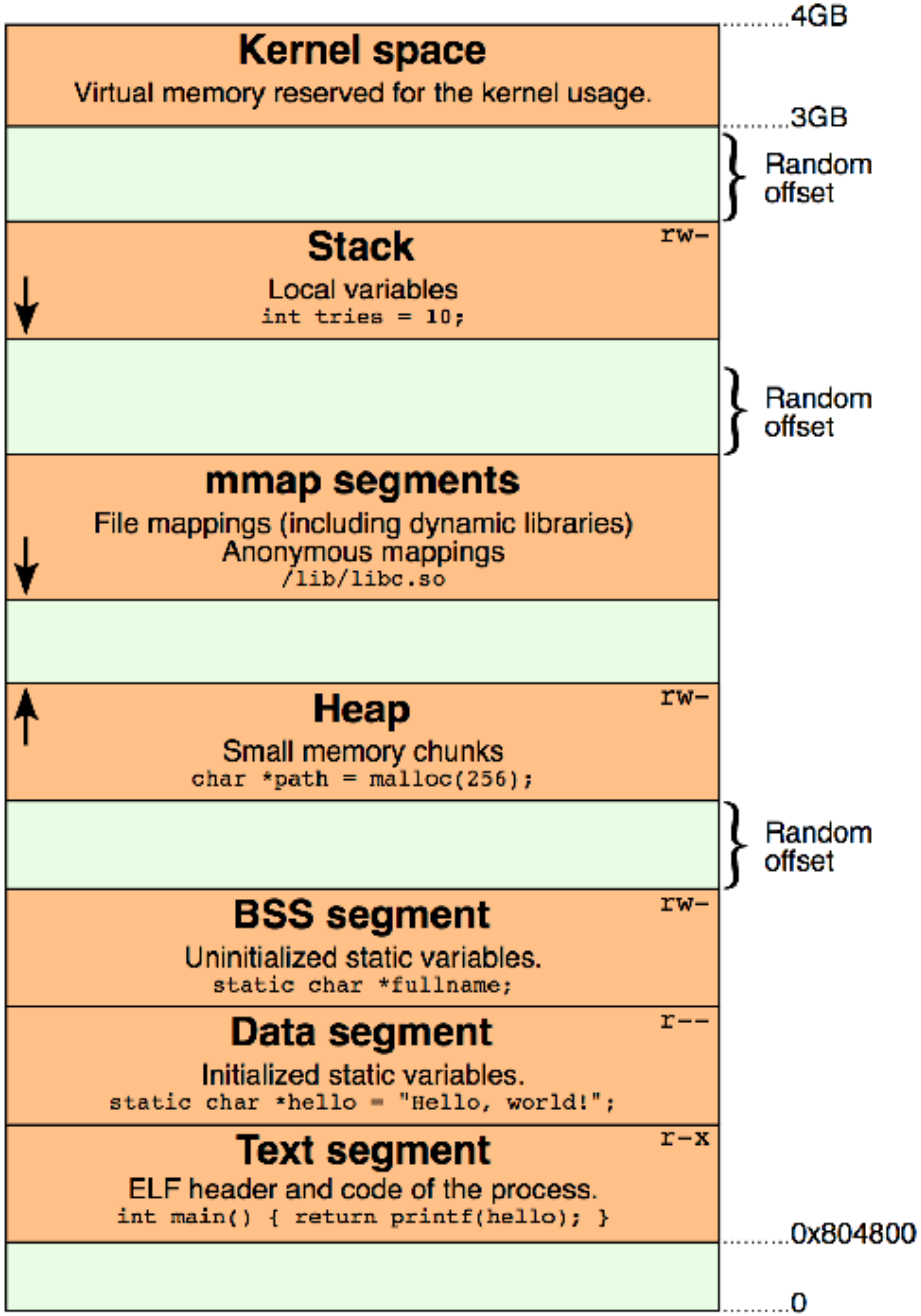
Your OS uses it



Stack: push / pop

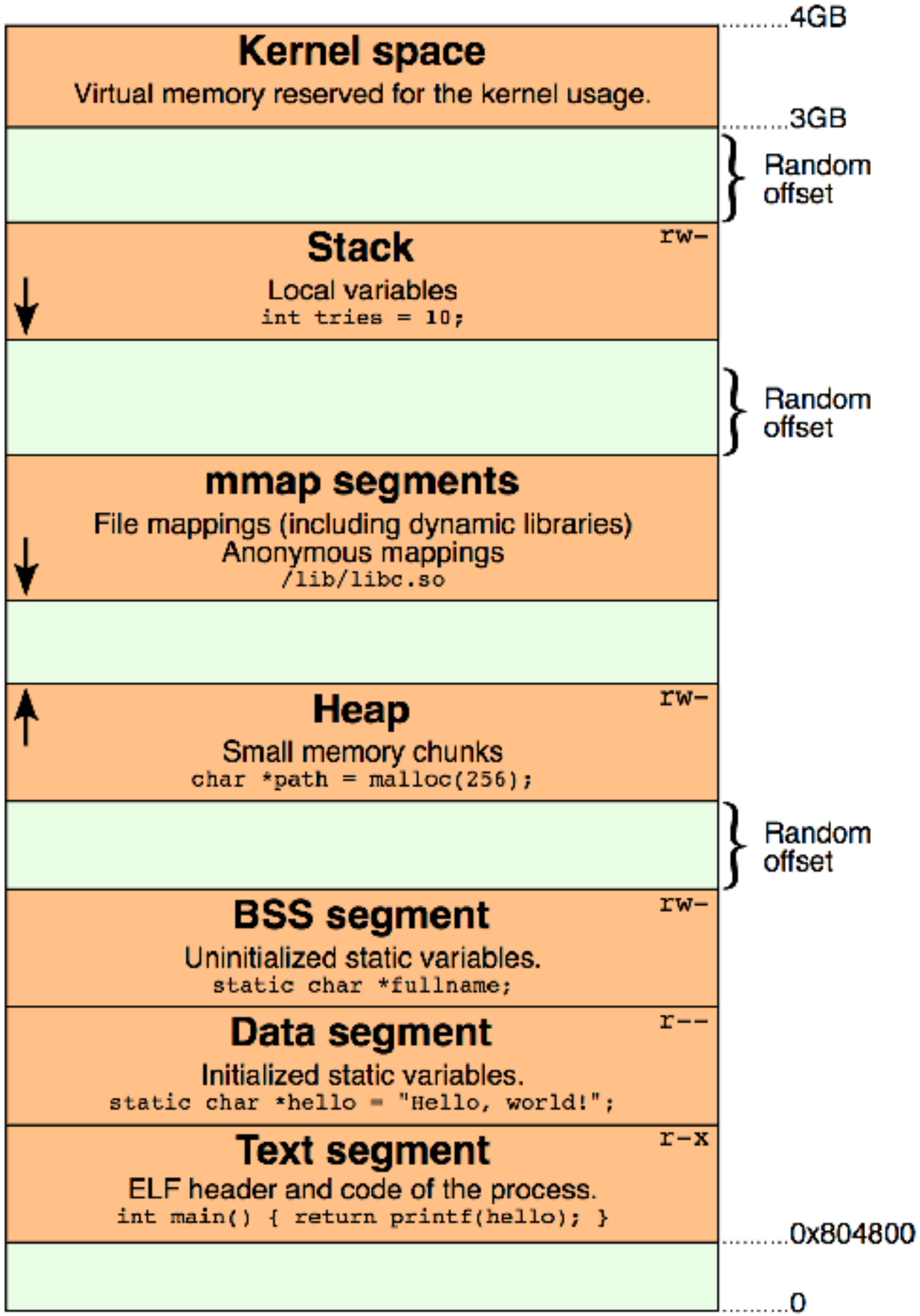
Very important:

The stack grows **down**



mmap segments

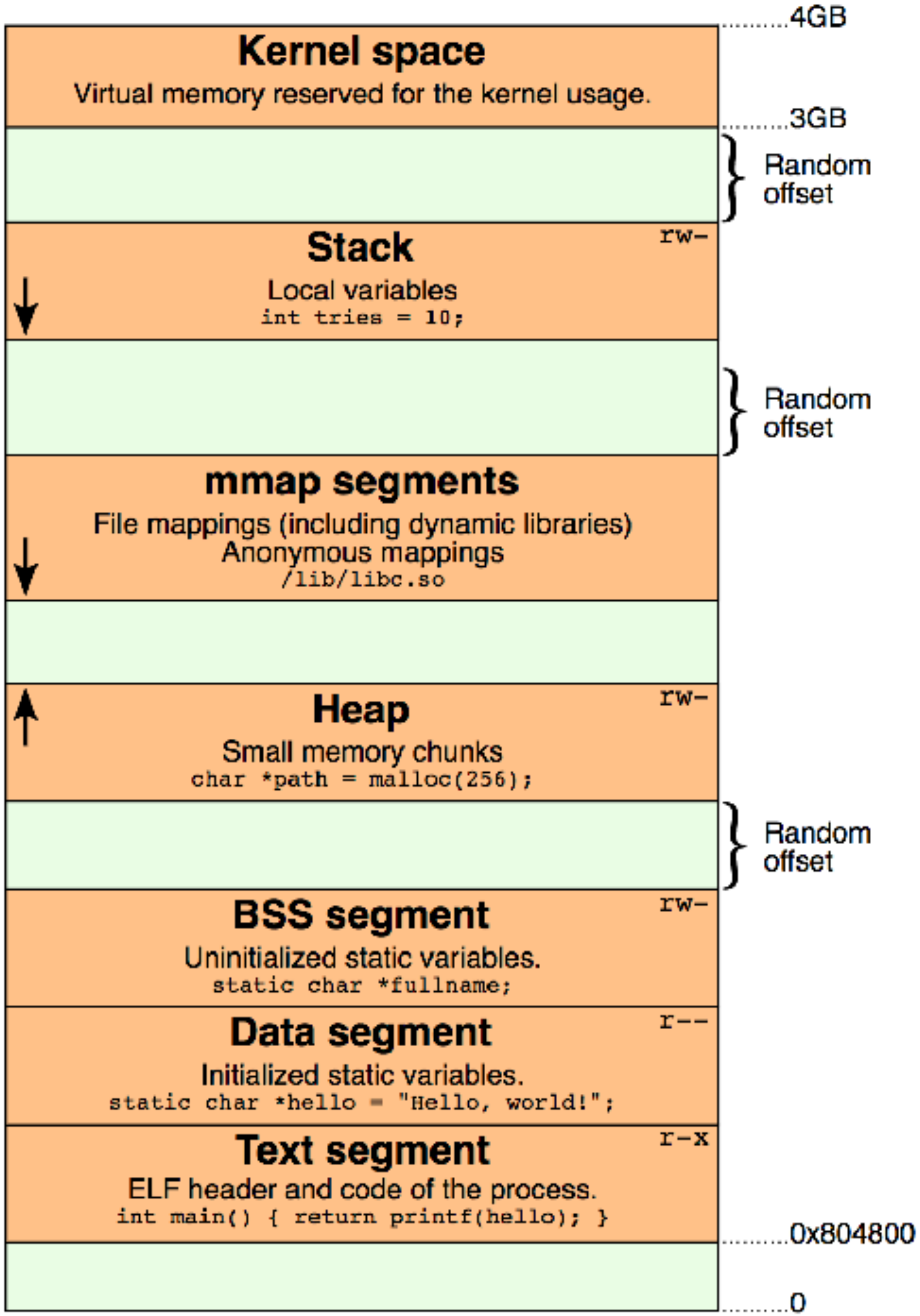
Allows you to **map** a file to memory



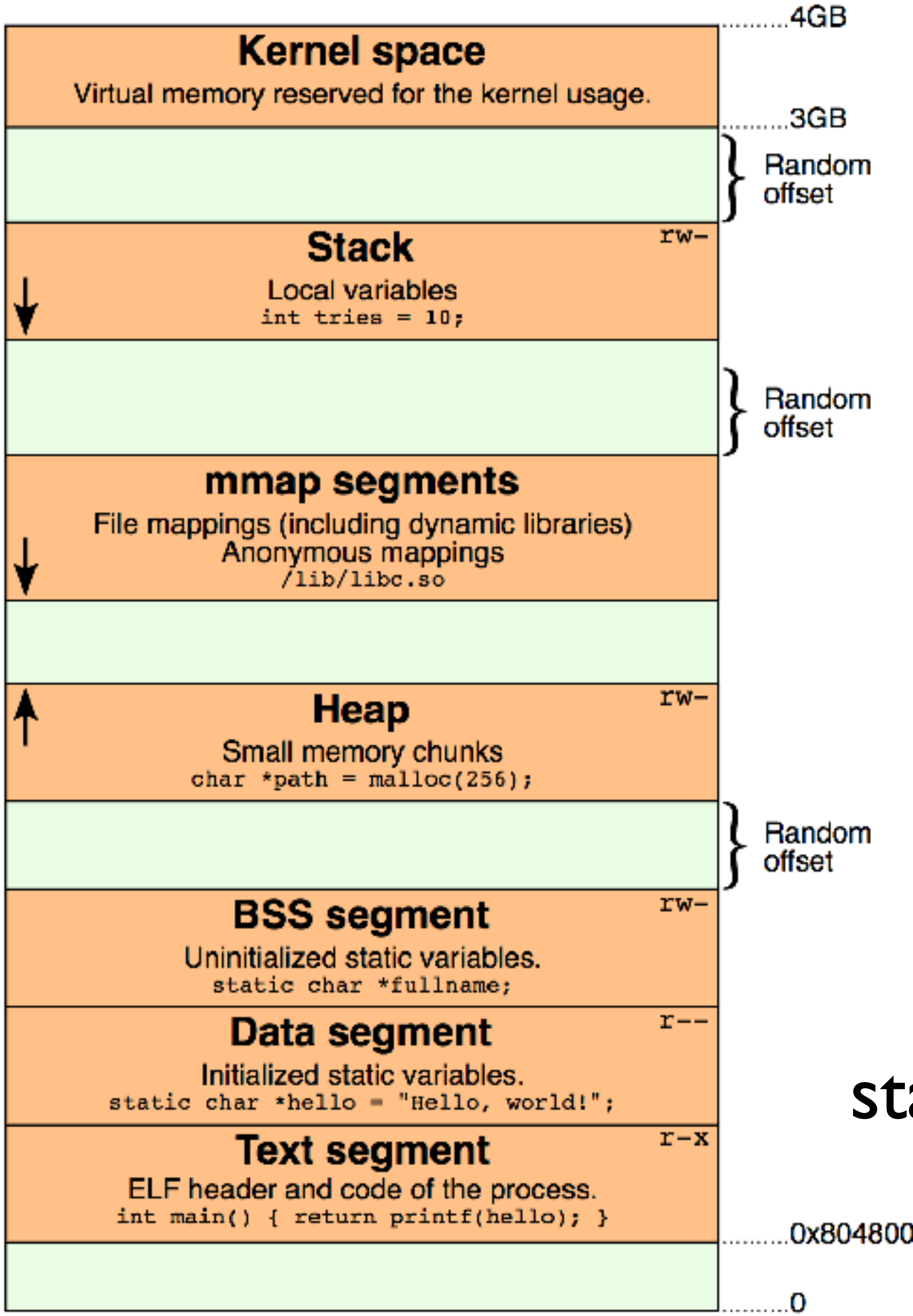
Heap: dynamic allocation

C++: New / delete

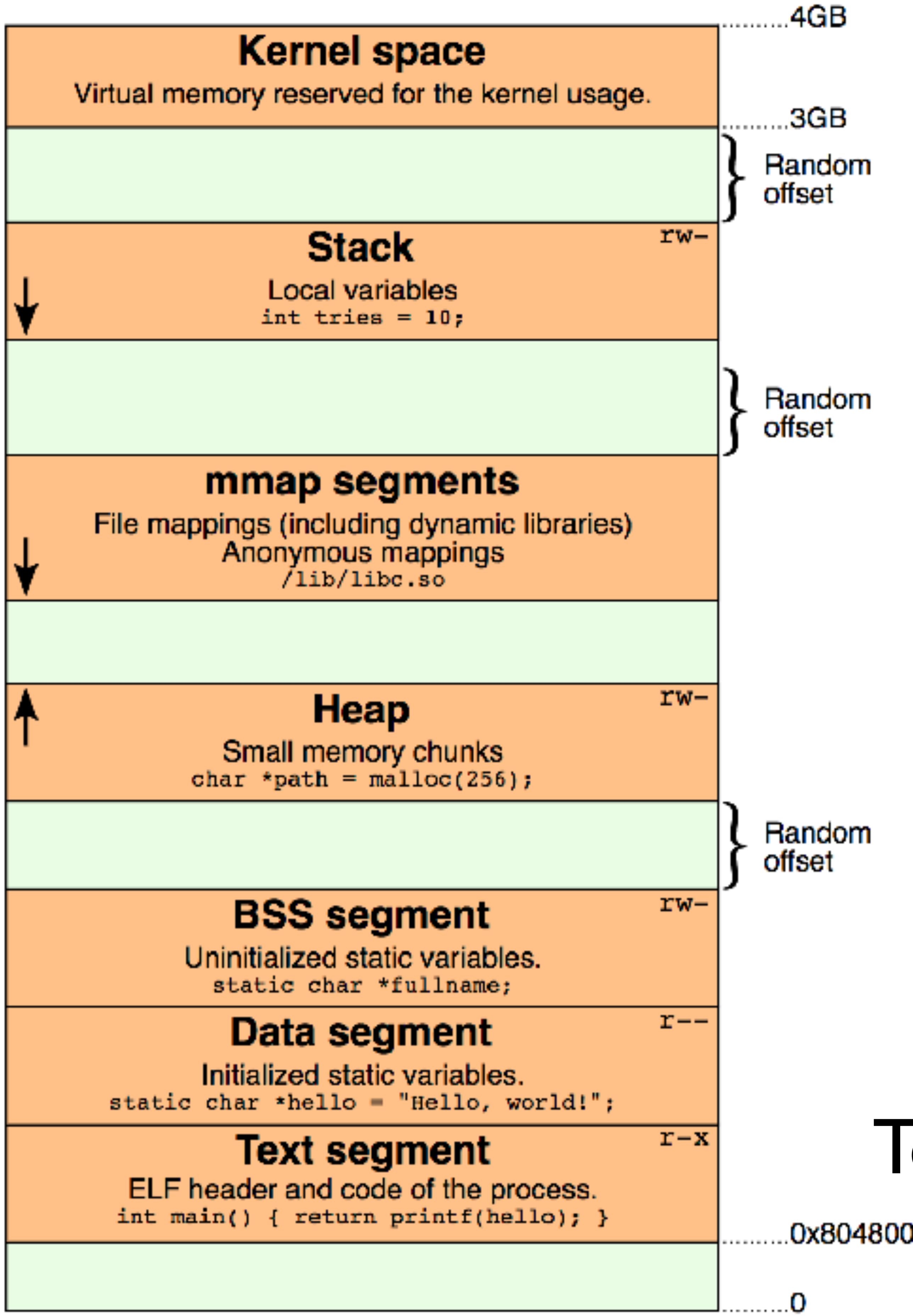
C: Malloc / free



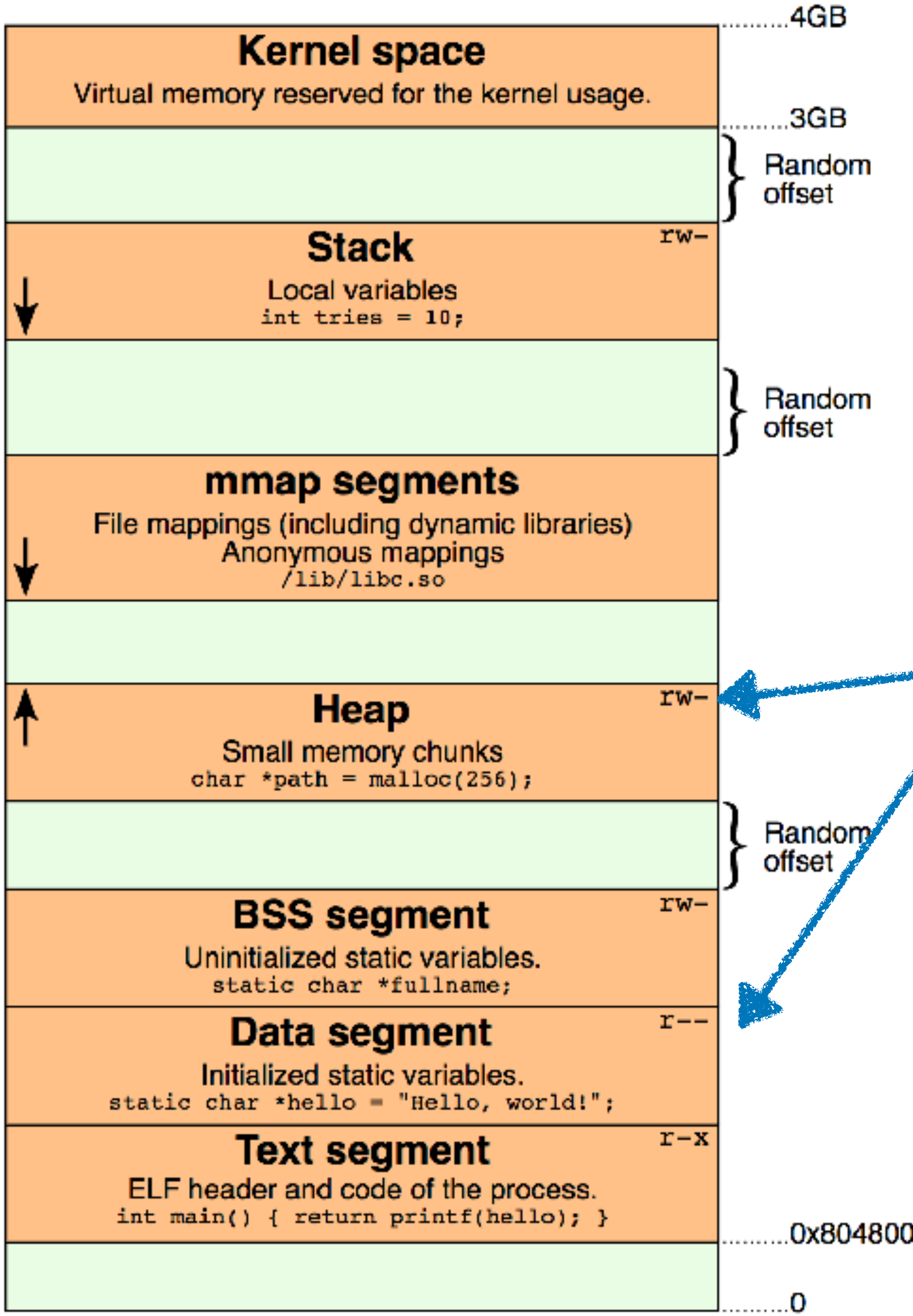
BSS: Uninitialized static vars (globals)



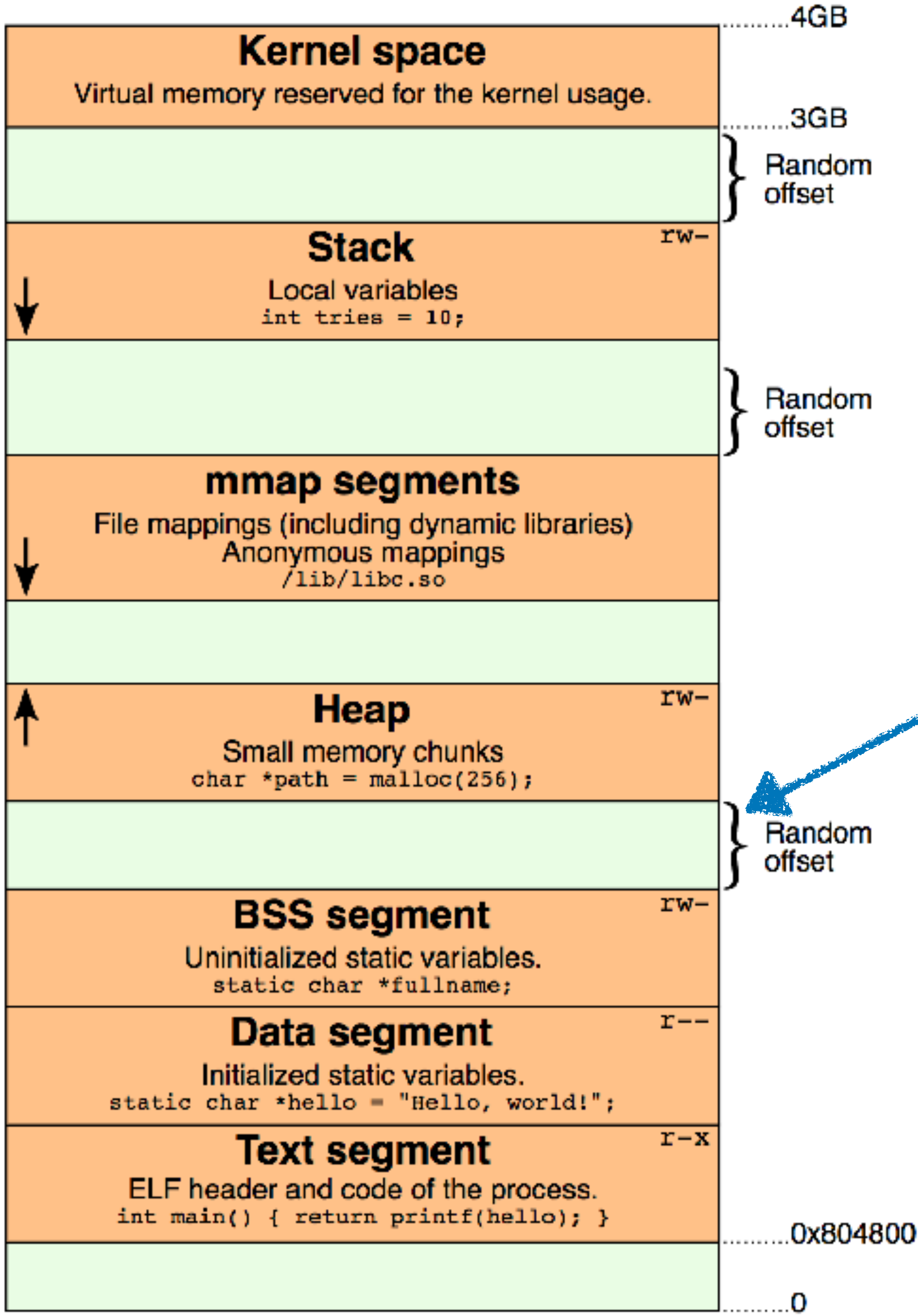
Data segment: initialized statics—e.g., constant strings



Text segment: program code



Note the **permissions**



This **random offset** really security feature

Example: Summing an array

```
.text
.globl _main
_main:
  pushq %rbp
  leaq data(%rip), %rax    # rax -- Pointer to count
  movq $5, %rbx            # rbx -- Size of count array
  movq $0, %rcx            # rcx -- Index var for loop
  movq $0, %rdx            # rdx -- Sum total of array
_loop:
  cmp %rcx, %rbx
  je _end_of_loop
  mov (%rax, %rcx, 8), %r8 # Loads *(rax + %rcx * 8) -> %r8
  addq %r8, %rdx
  addq $1, %rcx
  jmp _loop
_end_of_loop:
  movq %rdx, %rdi
  call _exit

.data
data:
  .quad   4, 2, -3, 1, 8  # Declares an array of 8-byte values
```

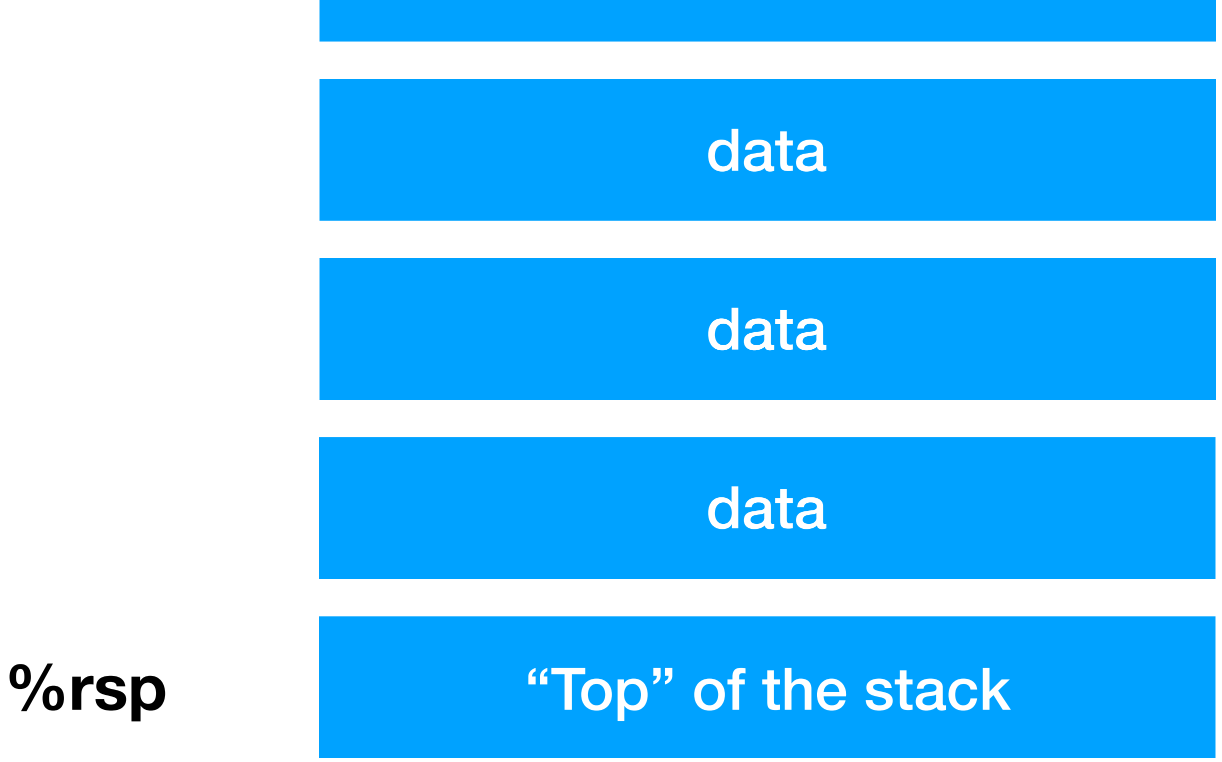
What do you do when you run out of registers..?

(There are only a limited number, so you **will** run out!)

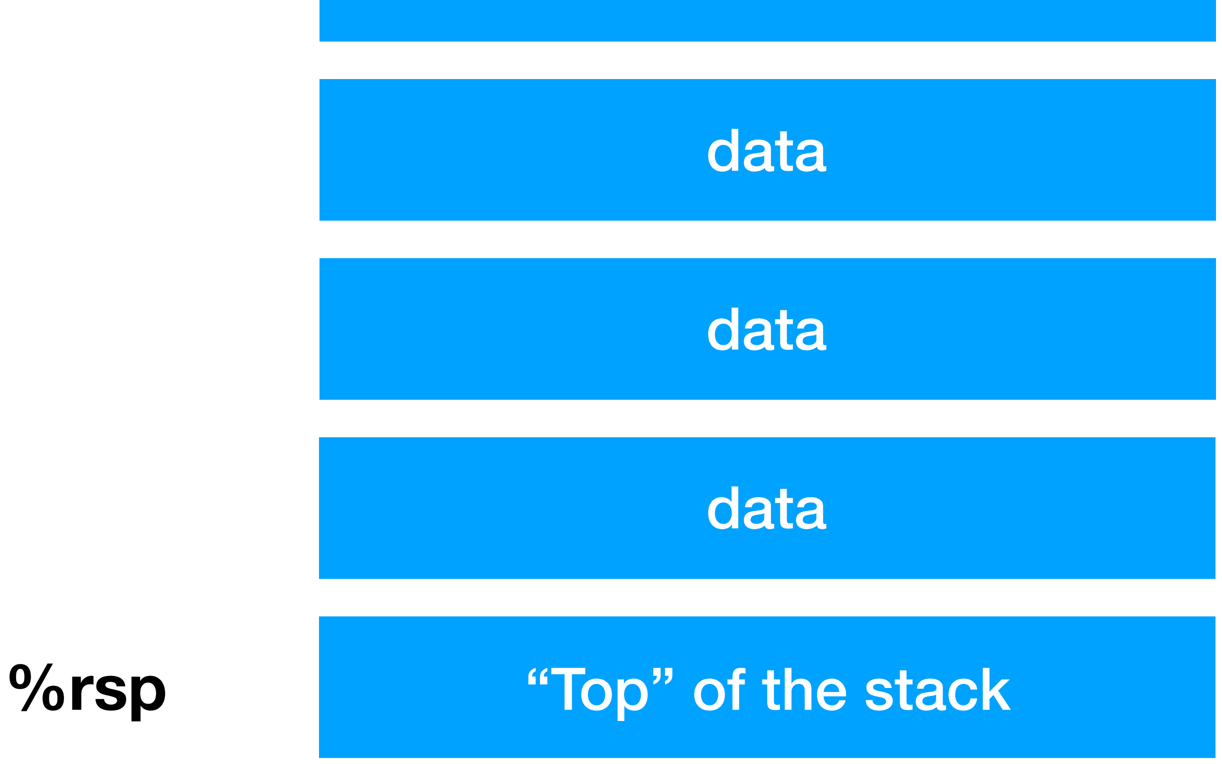
What do you do when you run out of registers..?

(There are only a limited number, so you **will** run out!)

Observation: can also use the **stack** to store data!

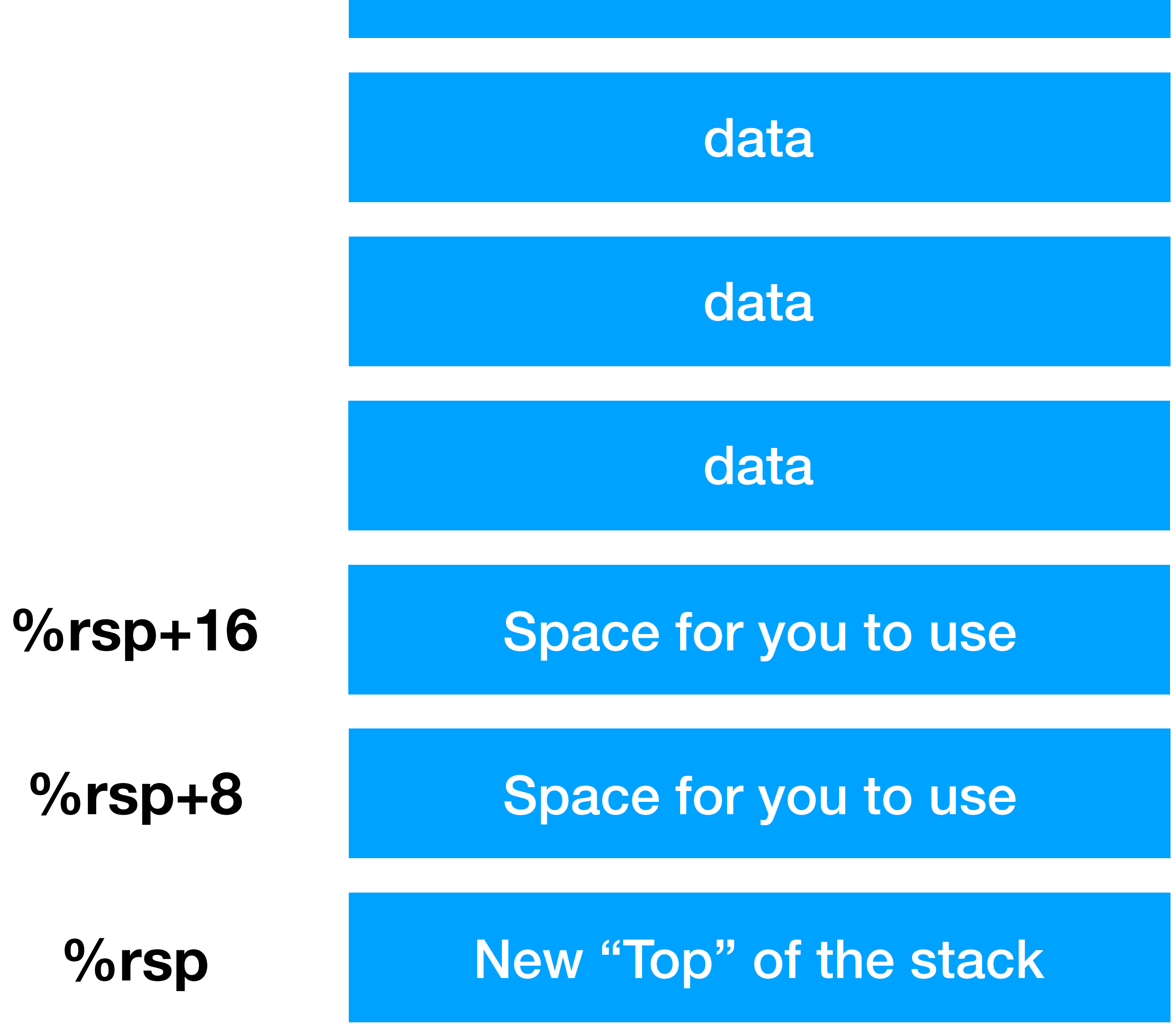


The **stack pointer** %rsp points at the top of the stack



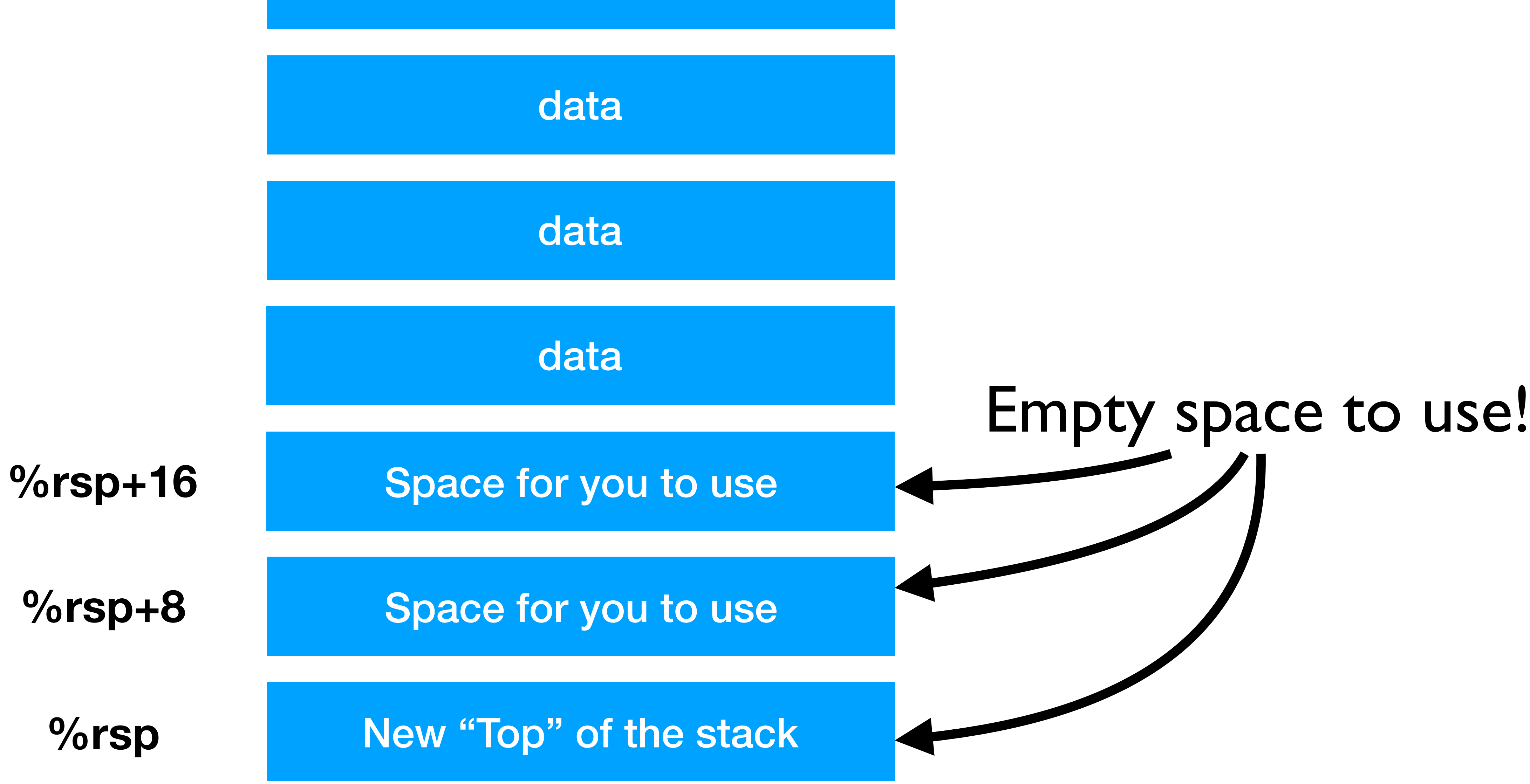
The **stack pointer** %rsp points at the top of the stack

If you want to store data on the stack, just subtract from %rsp and store there!



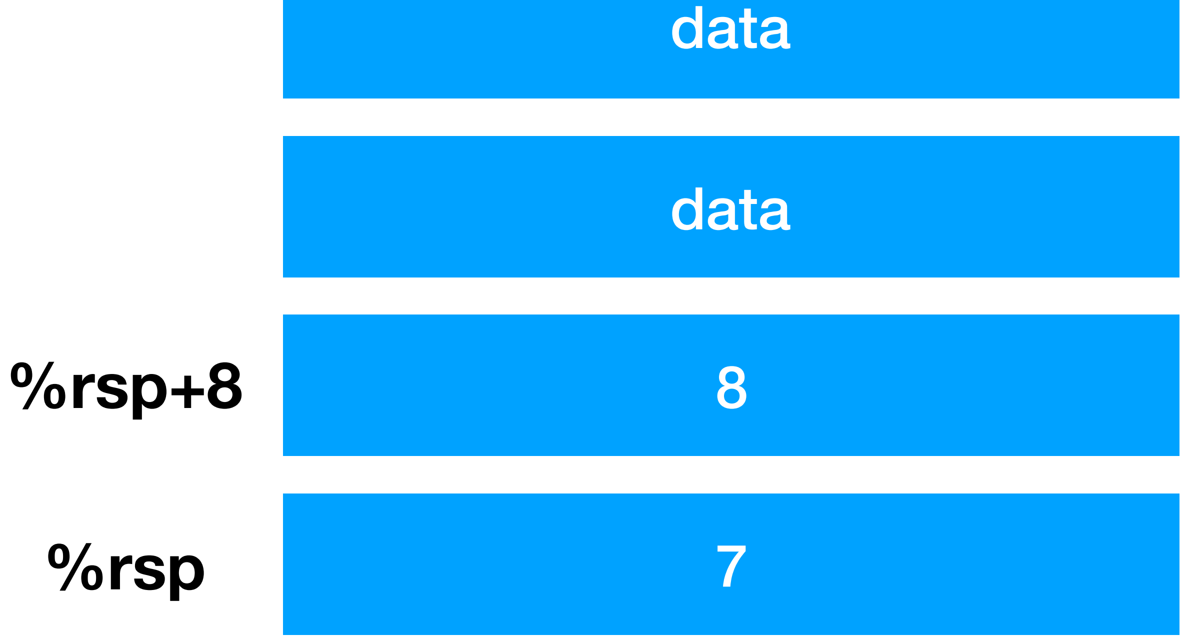
The **stack pointer** %rsp points at the top of the stack

If you want to store data on the stack, just subtract from %rsp and store there!



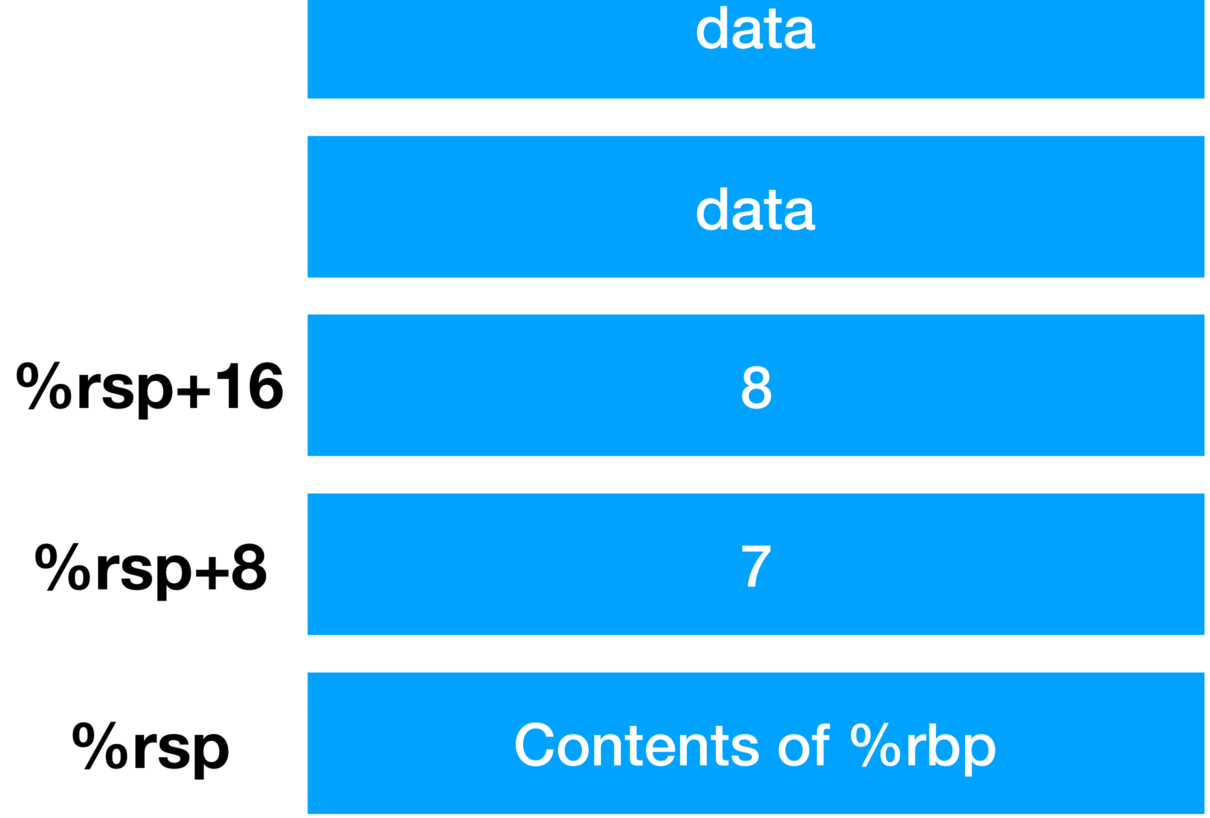
The **stack pointer** %rsp points at the top of the stack

If you want to store data on the stack, just subtract from %rsp and store there!



The "push" opcode decrements the stack and puts new data onto it

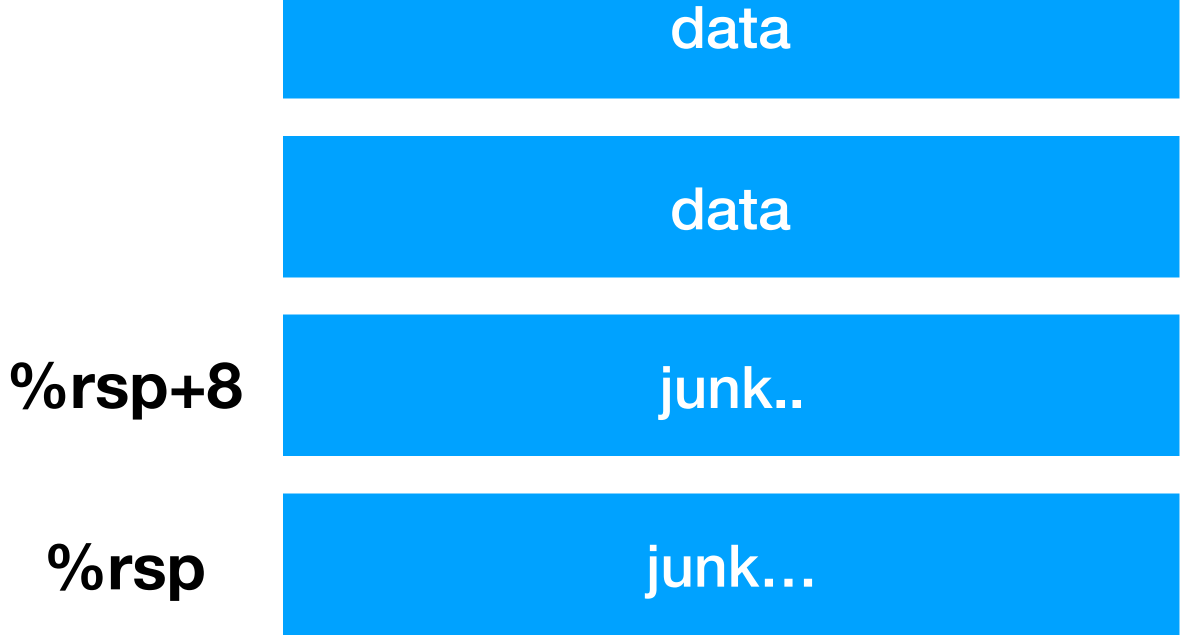
pushq %rbp



The "push" opcode decrements the stack and puts new data onto it

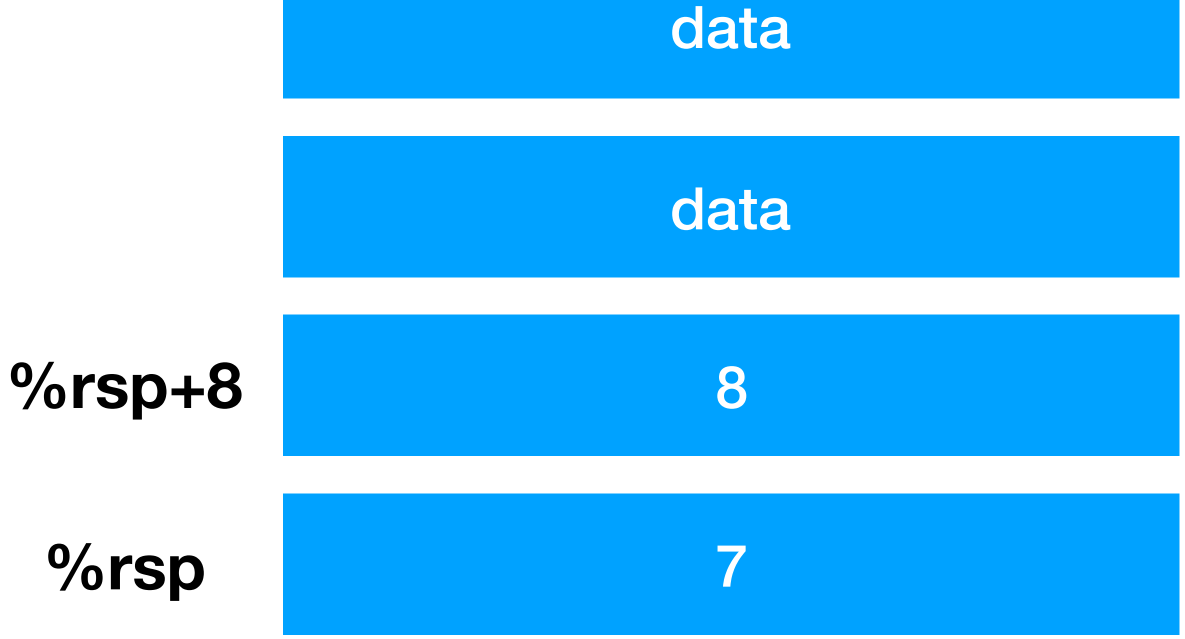
pushq %rbp


```
.text
.globl _main
_main:
  pushq %rbp
  subq $16, %rsp     # Reserve 16 bytes on the stack
  movq $7, (%rsp)      # Move 7 onto the top of the stack
  movq $3, 8(%rsp)   # Move 3 onto the next qword on the stack
  movq (%rsp), %rax  # Move *rsp into %rax
  movq 8(%rsp), %rbx # Move *(rsp+8) into %rbx
  addq %rax, %rbx
  movq %rbx, %rdi
  call _exit
```




```
.text
.globl _main
_main:
  pushq %rbp
  subq $16, %rsp      # Reserve 16 bytes on the stack
  movq $7, (%rsp)       # Move 7 onto the top of the stack
  movq $3, 8(%rsp)    # Move 3 onto the next qword on the stack
  movq (%rsp), %rax   # Move *rsp into %rax
  movq 8(%rsp), %rbx # Move *(rsp+8) into %rbx
  addq %rax, %rbx
  movq %rbx, %rdi
  call _exit
```


```
.text
.globl _main
_main:
  pushq %rbp
  subq $16, %rsp     # Reserve 16 bytes on the stack
  movq $7, (%rsp)      # Move 7 onto the top of the stack
  movq $3, 8(%rsp)   # Move 3 onto the next qword on the stack
  movq (%rsp), %rax  # Move *rsp into %rax
  movq 8(%rsp), %rbx # Move *(rsp+8) into %rbx
  addq %rax, %rbx
  movq %rbx, %rdi
  call _exit
```




```
.text
.globl _main
_main:
  pushq %rbp
  subq $16, %rsp     # Reserve 16 bytes on the stack
  movq $7, (%rsp)      # Move 7 onto the top of the stack
  movq $3, 8(%rsp)   # Move 3 onto the next qword on the stack
  movq (%rsp), %rax  # Move *rsp into %rax
  movq 8(%rsp), %rbx # Move *(rsp+8) into %rbx
  addq %rax, %rbx
  movq %rbx, %rdi
  call _exit
```

In many compilers (especially nonoptimizing ones), local variables are stored on the stack

(Even when they could be in registers!)

```
.text
.globl _main
_main:
  pushq %rbp
  subq $16, %rsp     # Reserve 16 bytes on the stack
  movq $7, (%rsp)      # Move 7 onto the top of the stack
  movq $3, 8(%rsp)   # Move 3 onto the next qword on the stack
  movq (%rsp), %rax  # Move *rsp into %rax
  movq 8(%rsp), %rbx # Move *(rsp+8) into %rbx
  addq %rax, %rbx
  movq %rbx, %rdi
  call _exit
```

```
int main() {
  int x = 7;
  int y = 3;
  exit(x+y);
}
```

```
.text
.globl _main
_main:
  pushq %rbp
  subq $16, %rsp      # Reserve 16 bytes on the stack
  movq %rsp, %rbp     # Move %rsp into the base pointer %rbp
  movq $7, (%rbp)     # Move 7 onto the top of the stack
  movq $3, 8(%rbp)    # Move 3 onto the next qword on the stack
  movq (%rbp), %rax  # Move *rsp into %rax
  movq 8(%rbp), %rbx # Move *(rsp+8) into %rbx
  addq %rax, %rbx
  movq %rbx, %rdi
  call _exit
```

Because the stack often grows up and down, programmers sometimes use %rbp

(“**base pointer:**” points at **base** of local variables)

(Dereferences can use %rbp even when %rsp changes)

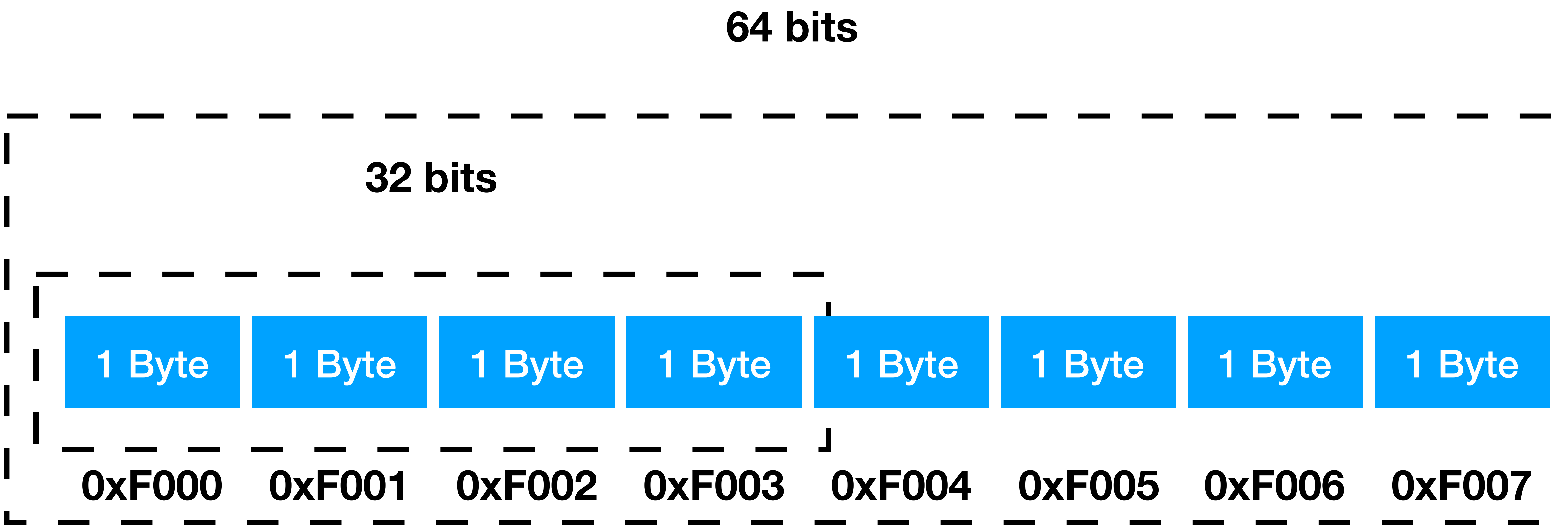
Because functions often store their local variables on the stack, a common "recipe" for writing a function is:

- Push %rbp onto the stack (save the caller's %rbp)
- Subtract **x** bytes from the stack
  - Where x is the number of bytes taken by local variables
  - Often padded to the nearest 16-byte value for alignment
- Move %rsp into %rbp
- Each local variable is now at (%rbp), 8(%rbp), …

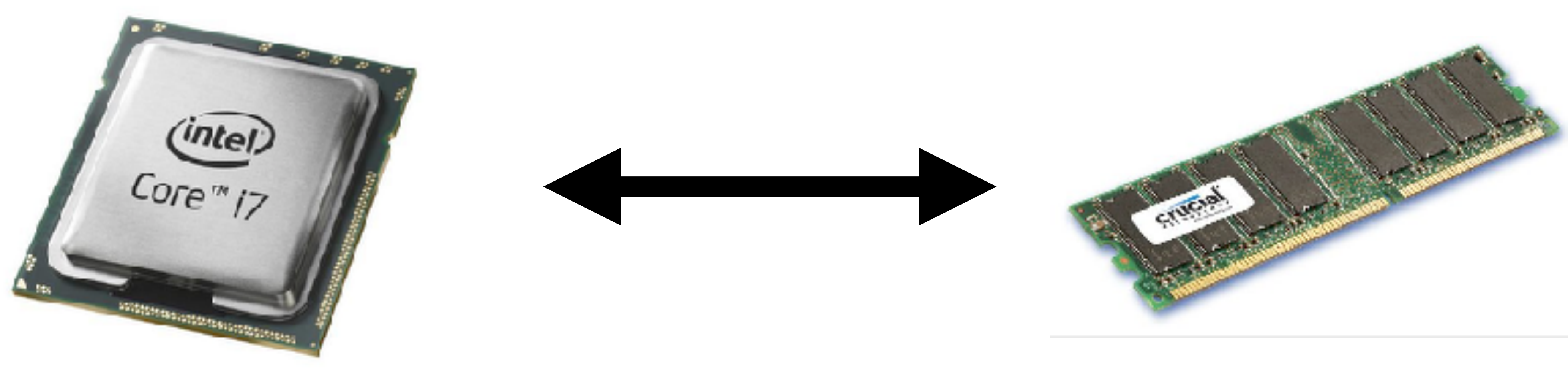
```
int foo() {
  int x;          // 4 bytes
  int y;          // 4 bytes
  char foo[16];   // 16 bytes
  double z;       // 8 bytes
}
```

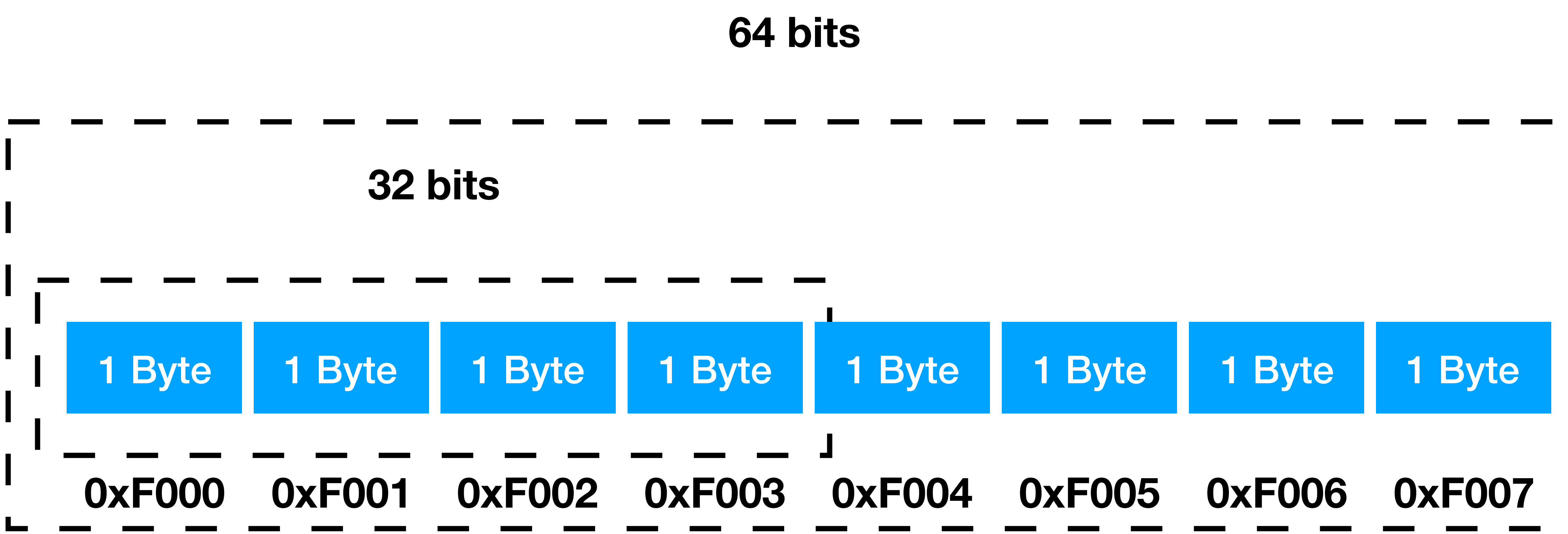
→

```
_foo:
  pushq %rbp
  subq  $16, %rsp
  movq  %rsp, %rbp
  # z is (%rbp)
  # foo is 8(%rbp)
  # y is 24(%rbp)
  # z is 28(%rbp)
```



Your processor talks to RAM via a bus

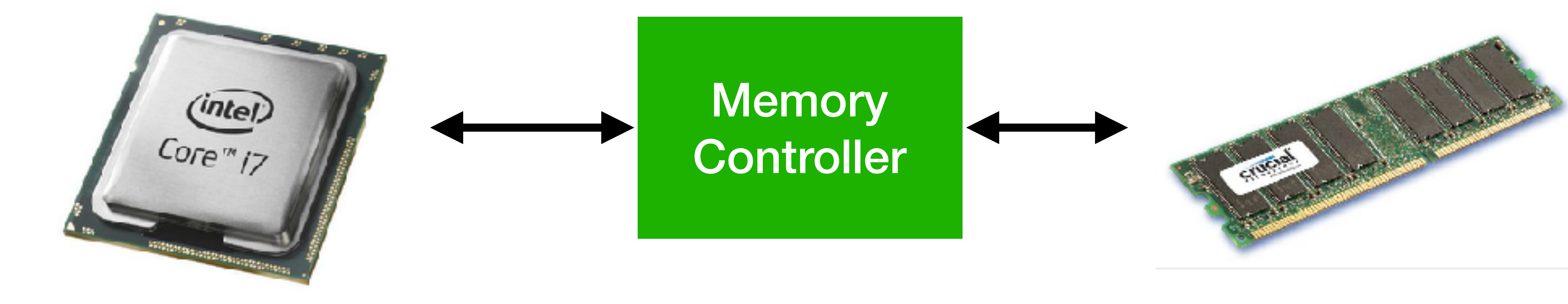


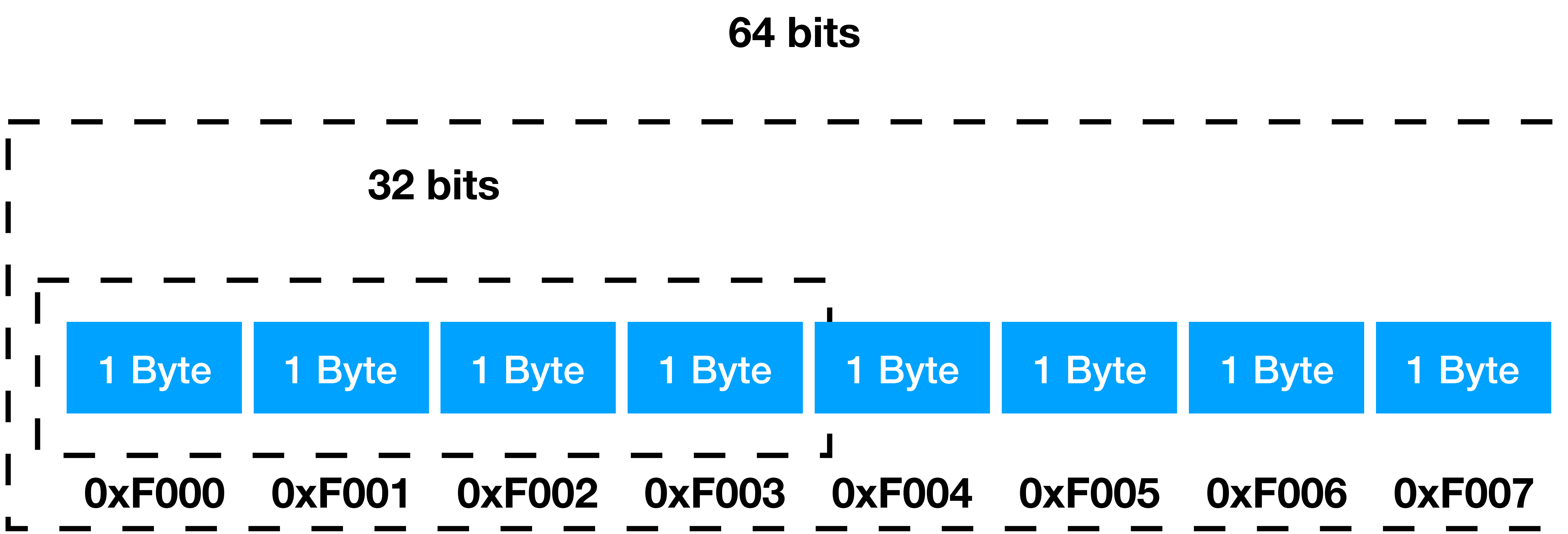


Your processor talks to RAM via a bus

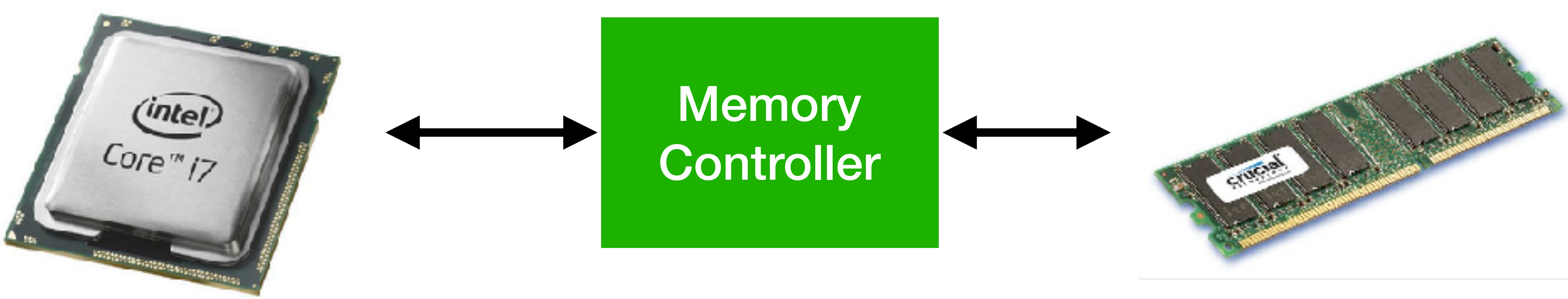
The memory controller interfaces the RAM banks to the CPU

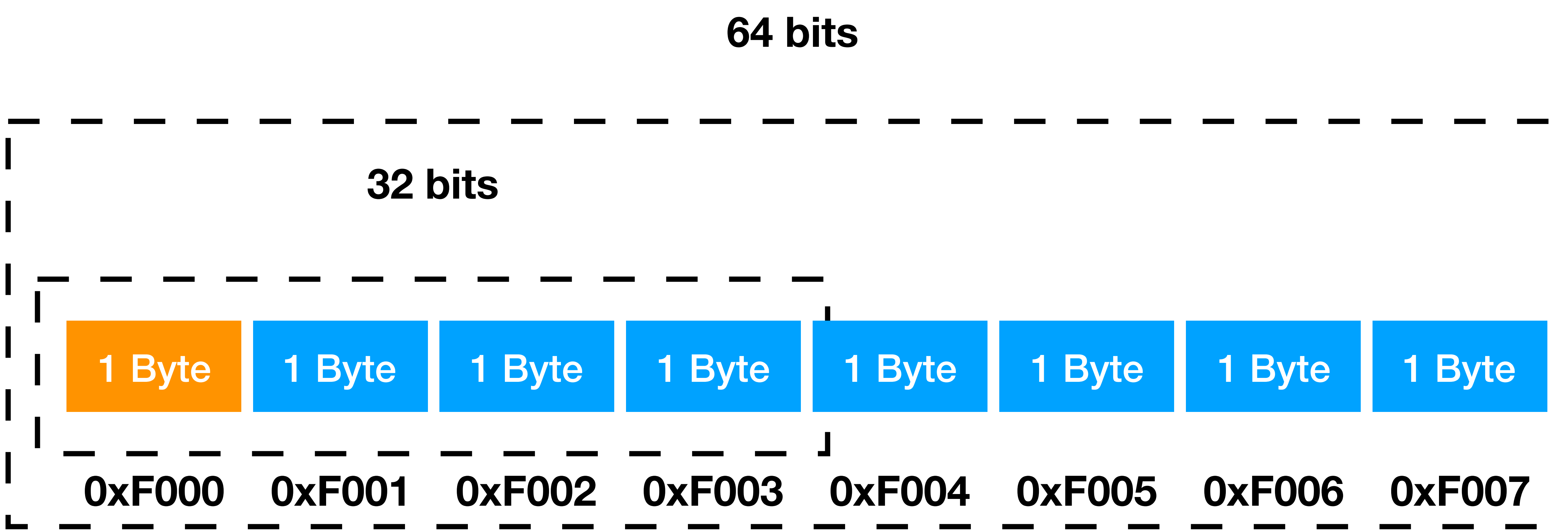
(E.g., what if multiple CPUs access same RAM at once)



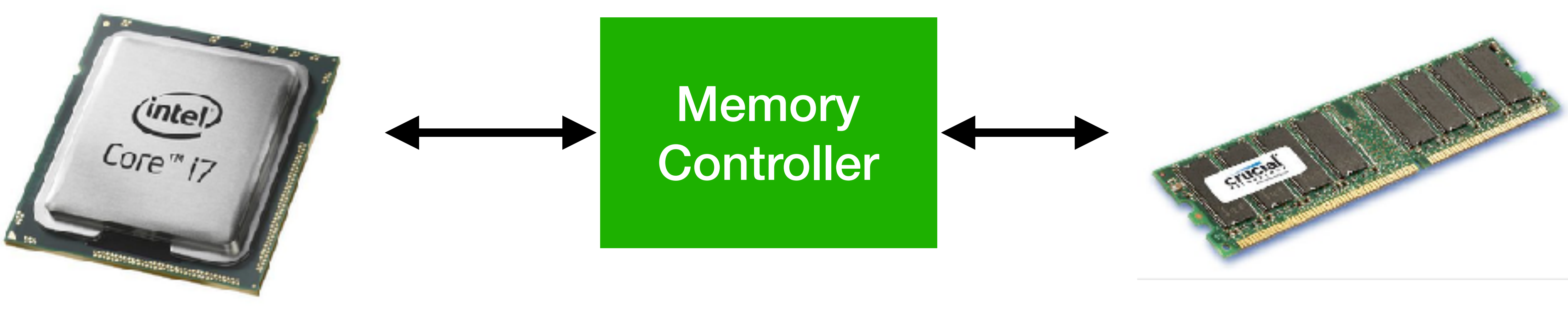


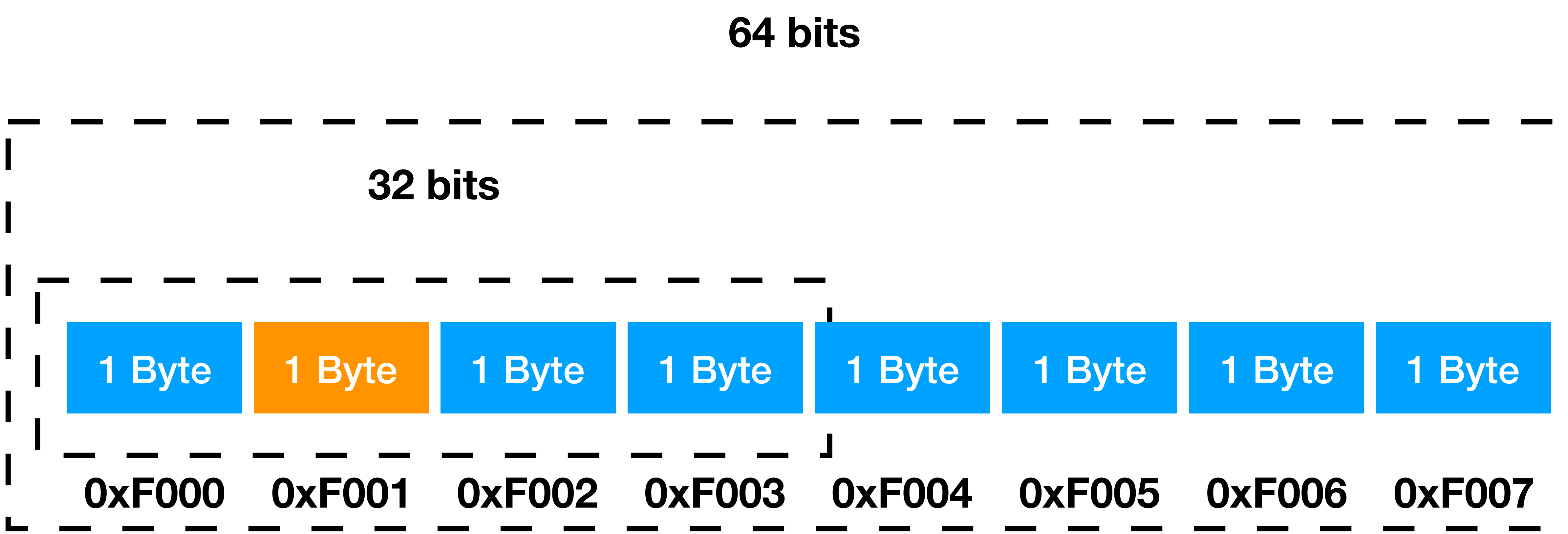
It makes the memory controller circuitry simpler when it only allows accessing memory at an address which is a multiple of 8 (etc..)



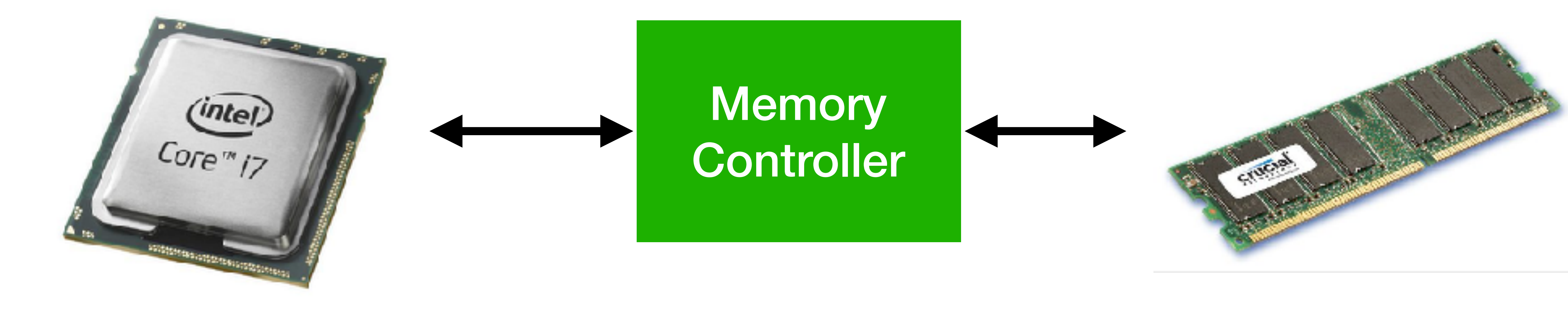


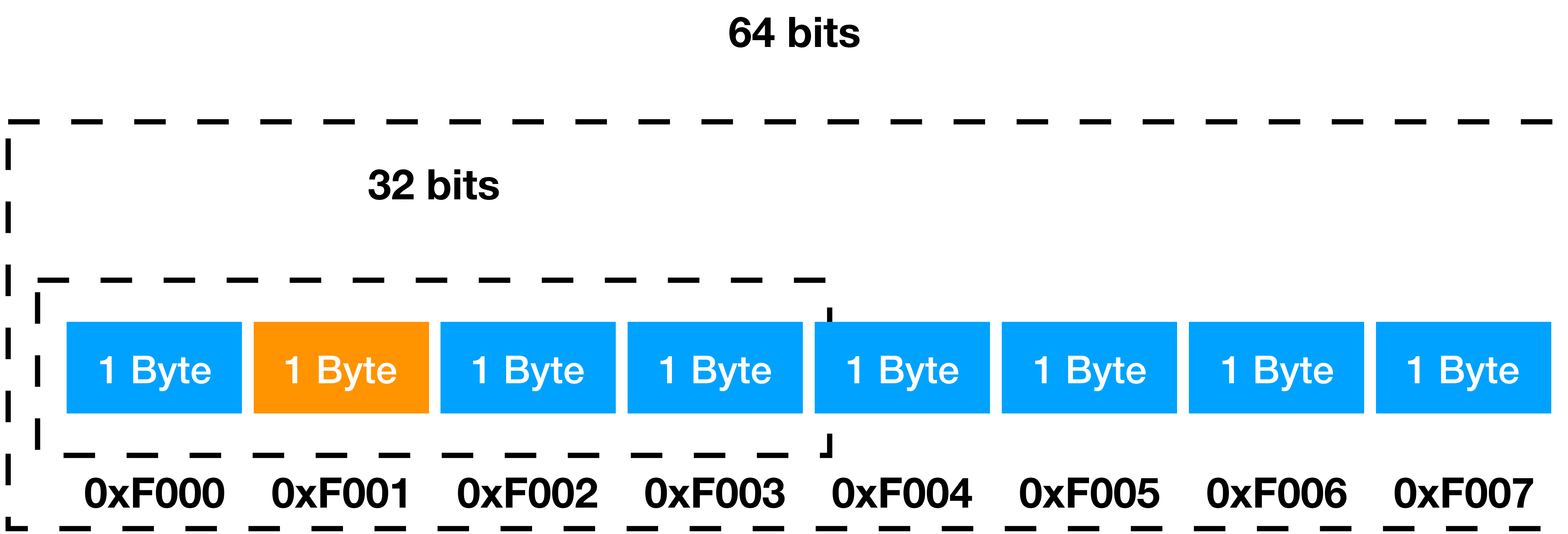
Valid start of an 8-byte datatype





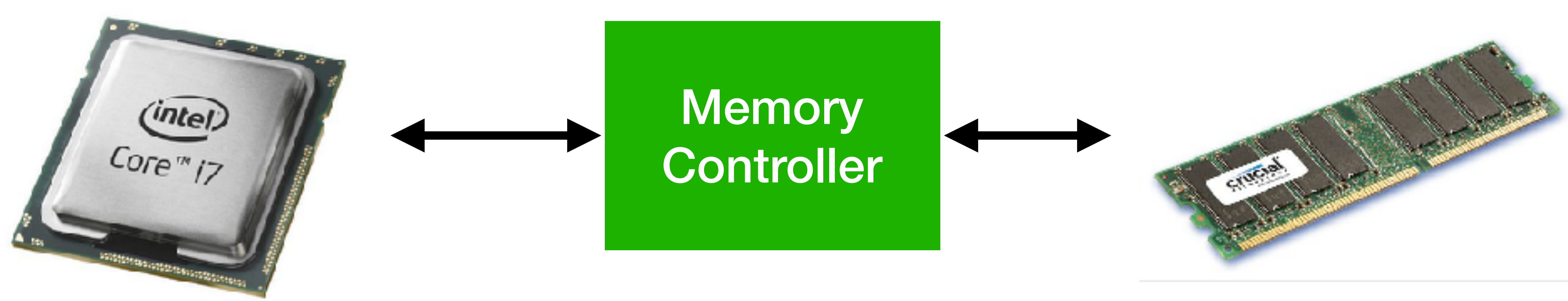
Invalid: address (0xF001) is not multiple of 8





Invalid: address (0xF001) is not multiple of 8

(In this case, the processor actually does **two** fetches.
One from 0xF000 to get 0xF001-0xF007, One to get 0xF008-…)



Alignment

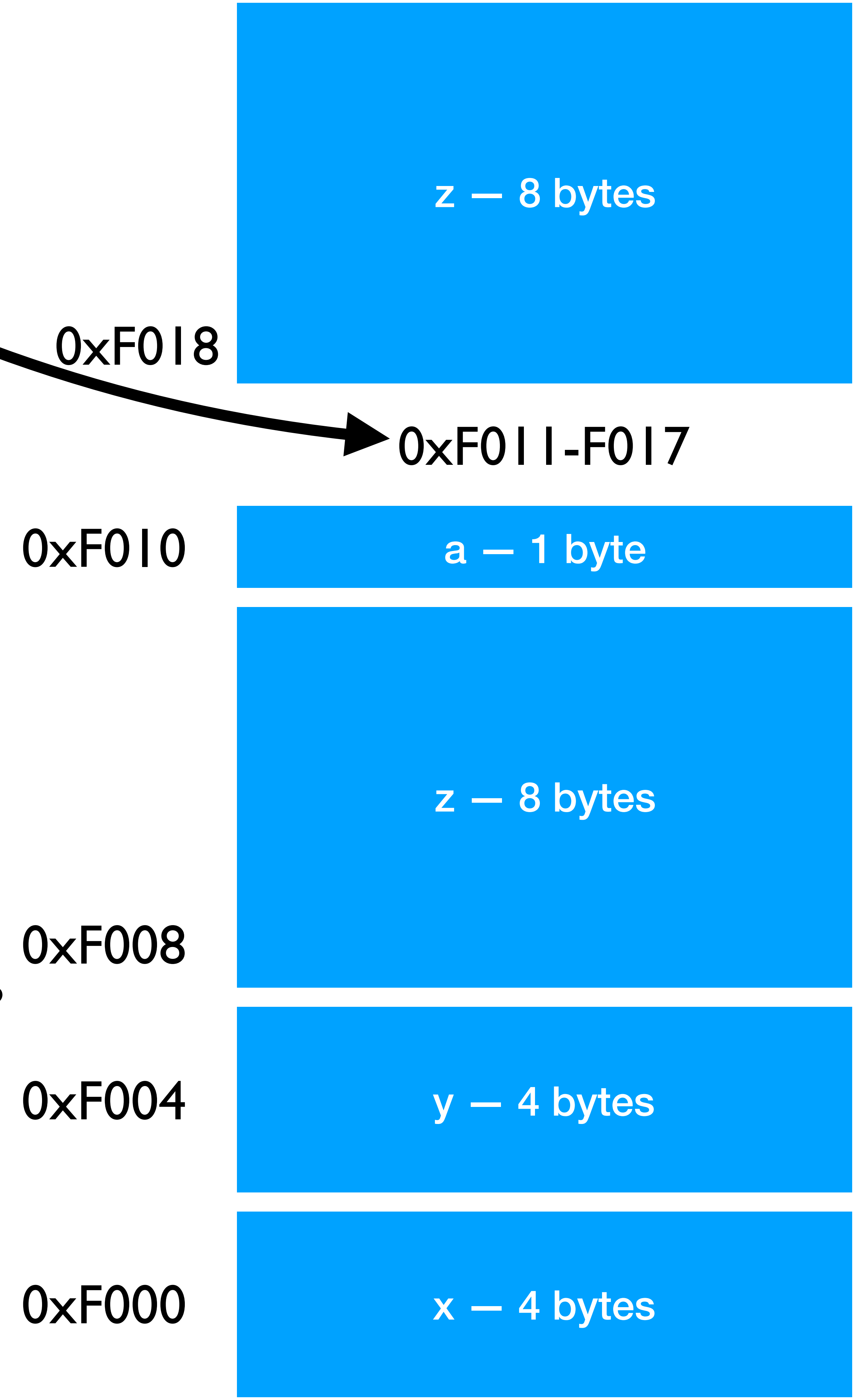
Concept of laying out data in memory to respect constraints of the ISA's memory access conventions

Typically, an n-byte datatype will be aligned on an n-byte boundary (where n is 1,2,4,8,16,…)

E.g., a **double** in C++ is 8-bytes in size, meaning it must sit at a memory address which is divisible by 8 (0x00, 0x08, 0x10, …)

Empty space for alignment

```
struct Foo {
  int x;
  int y;
  char *z;
  char a;
  int *num;
}
```



Structs laid out sequentially in memory, but alignment must be maintained!

Quiz: Which takes less space?

```
struct Foo {
  char y;
  int *num;
  int  z;
}
```

```
struct Foo {
  int *num;
  int  z;
  char y;
}
```

Calling conventions

Touch-tone phones, send an acoustic wave over the wire

If Alice wants to call Bob, her phone needs to send the right sounds over the wire in the right order

Calling conventions

When function A wants to call function B, it has to do the same

- Where do arguments go?
- How to store return address?
- Who saves registers?
- Where is result stored?

Calling conventions

Modern computers use a few **different** calling conventions

De-facto standard (Linux / MacOS / etc..) : **x86-64 System V ABI**

- Where do arguments go?
- How to store return address?
- Who saves registers?
- Where is result stored?

Note: this is **new** for the 64 bit API. You might see stuff online for the 32-bit API that is **different**

Calling conventions: x86-64 System V ABI

- Where do arguments go?
  - First six: rdi,rsi,rdx,rcx,r8,r9
- How to store return address?
  - `call` instruction puts on top of stack
- Who saves registers?
  - Caller saves caller-save registers
    - R10,R11, any ones used for args
- Where is result stored?
  - Result stored in %rax

x86-64 Integer Registers: Usage Conventions

`%rax`	Return value	`%r8`	Argument #5
`%rbx`	Callee saved	`%r9`	Argument #6
`%rcx`	Argument #4	`%r10`	Caller saved
`%rdx`	Argument #3	`%r11`	Caller Saved
`%rsi`	Argument #2	`%r12`	Callee saved
`%rdi`	Argument #1	`%r13`	Callee saved
`%rsp`	Stack pointer	`%r14`	Callee saved
`%rbp`	Callee saved	`%r15`	Callee saved

http://slideplayer.com/slide/9679824/

x86-64 System V ABI

Rules for **caller**:

- Save caller-save registers
- First six args in registers, after that put on stack
- Execute `call`—pushes ret addr

Afterwards:

- Pop saved registers
- Result now in %rax

x86-64 System V ABI

Rules for **callee**:

- First six args available in registers
- Push %rbp—caller's base pointer
- Move %rsp to %rbp—Setup new frame
- Subtract necessary stack space
- Push callee-save registers
- Before exit: restore rbp/callee-saved regs
  - `leave` instruction restores rbp
- When function done, put result in %rax
- Use `ret` instruction to pop return rip

These rules are cumbersome: I frequently look them up, they change depending on the kind of function you're calling, etc…

Upshot: don't feel you have to memorize, just get the gist / know how to recognize them

Small examples: interactive demo of x86-64 ABI

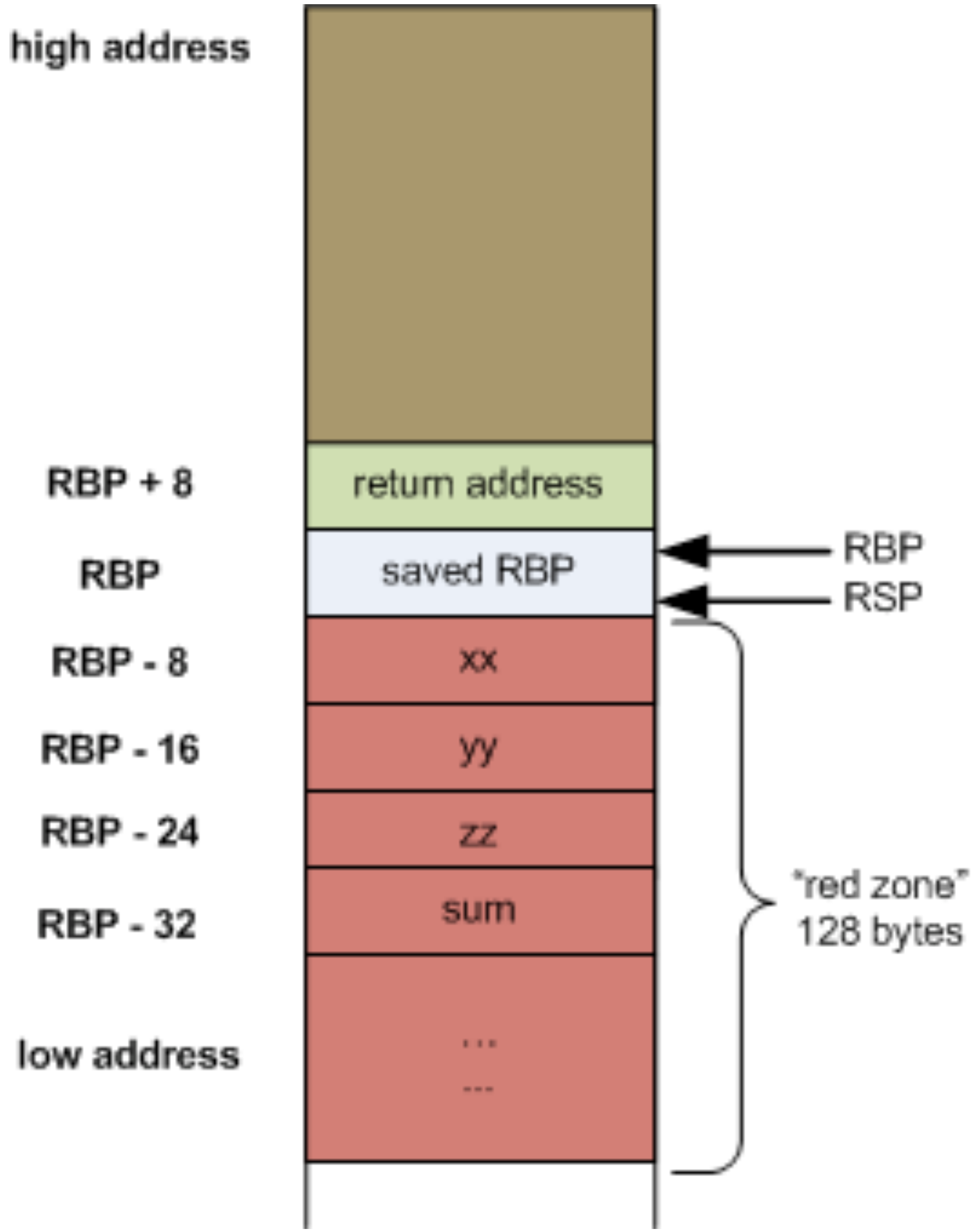
Trivia: the red zone

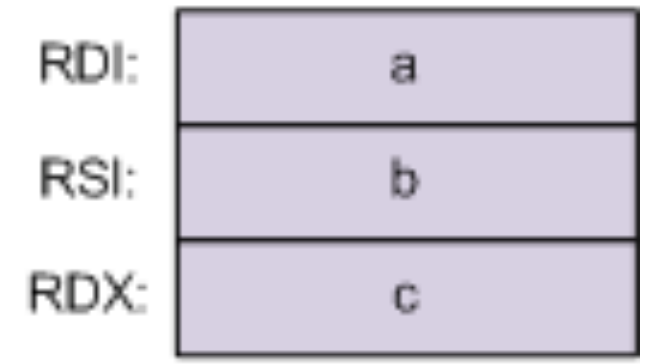
```
int bar(int a, int b) {
  return a + b;
}
```

Weird! This code using -4(%rbp) before decrementing the stack pointer!!

Turns out: x86-64 **guarantees** there are always 128 bytes below %rsp

```
bar:
    pushq   %rbp
    movq    %rsp, %rbp
    movl    %edi, -4(%rbp)
    movl    %esi, -8(%rbp)
    movl    -4(%rbp), %edx
    movl    -8(%rbp), %eax
    addl    %edx, %eax
    popq    %rbp
    ret
```





Upshot: if a function uses at most 128 bytes below RSP, doesn't have to subtract anything from RSP

This is an optimization for "small" functions: so they never have to subtract from RSP

Question: why does GCC generate such stupid code?

Answer: code unoptimized, add -O(1/2/3) to optimize it

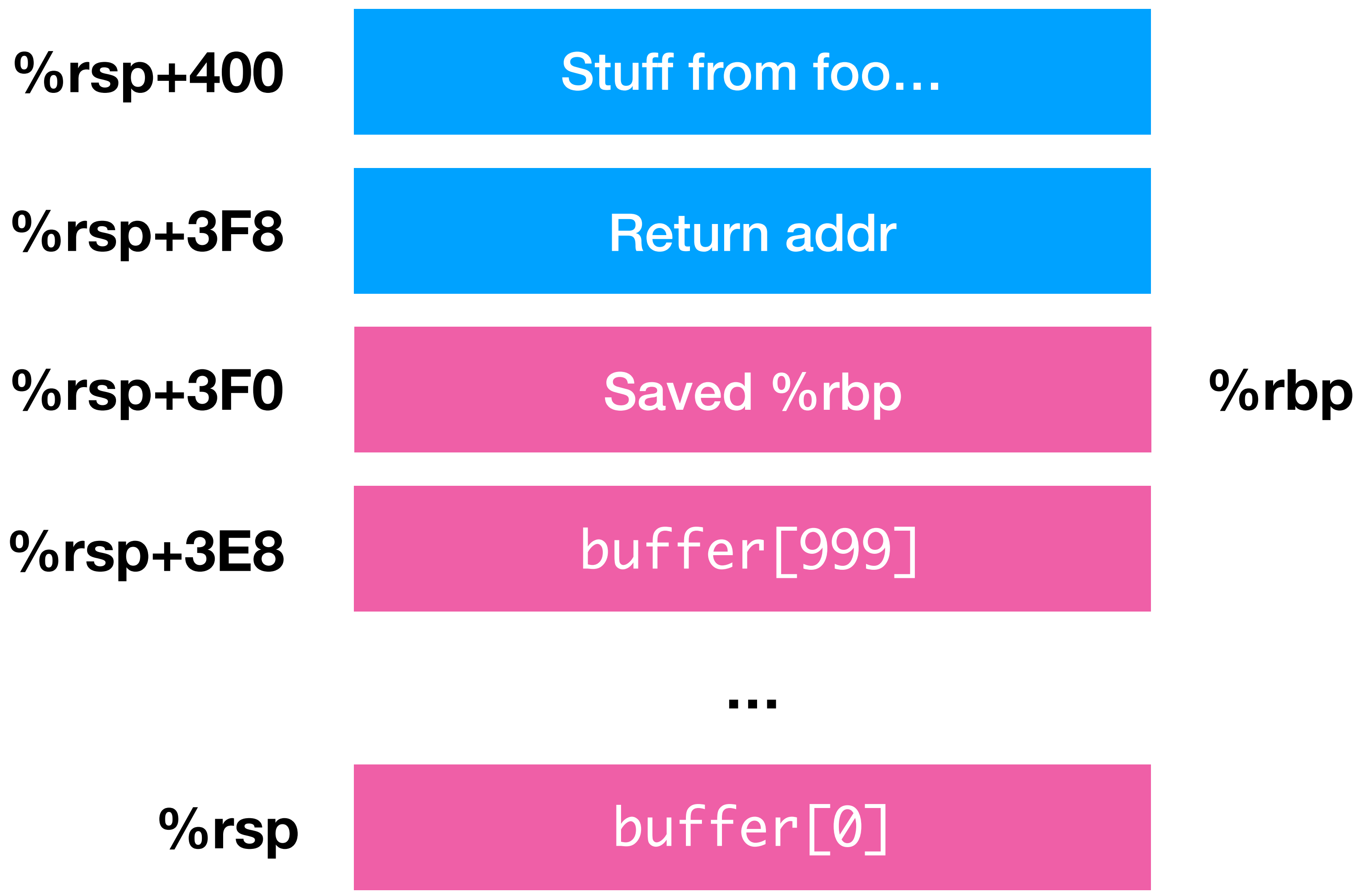
-O0 generates code that is predictable and easy to read

First attack: Stack Smashing

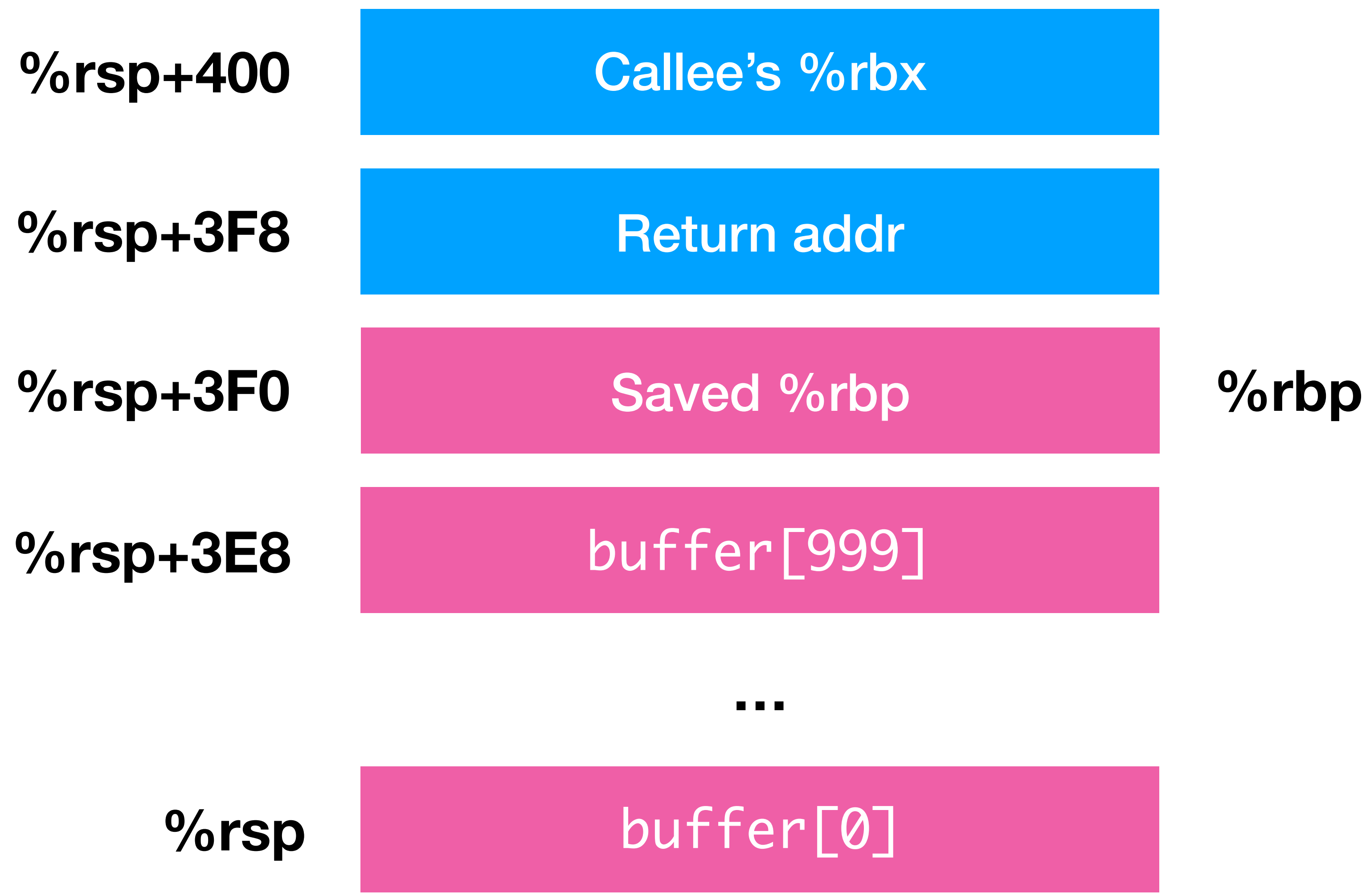
This code is bad because it doesn't check the length of the string in `ptr`…

```
void foo(char *ptr) {
    char buffer[1000];
    strcpy(buffer, ptr);
    printf("length: %d\n", strlen(buffer));
}
```

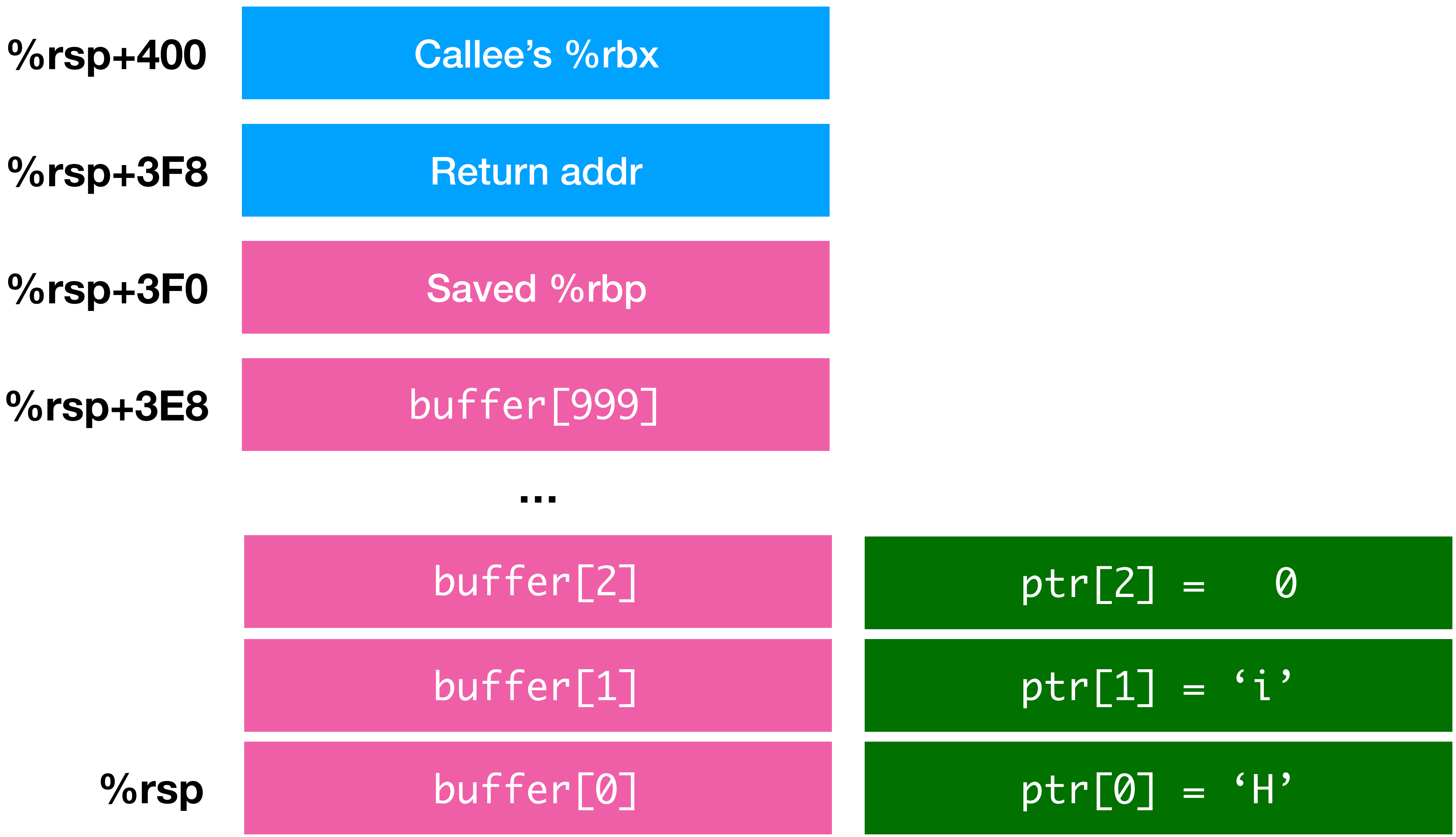
After `foo` starts



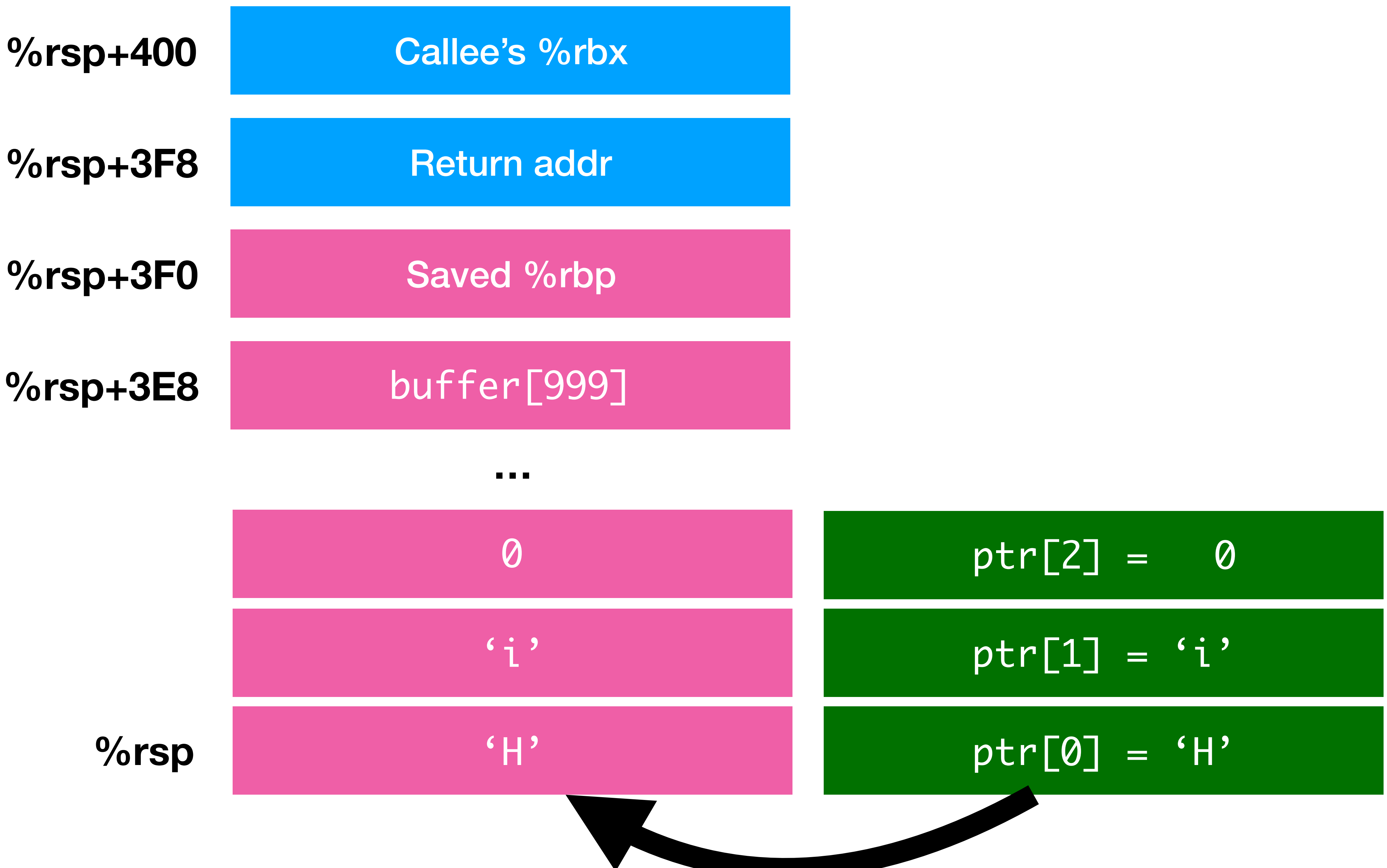
After `foo` starts



Key observation: the stack **grows down**



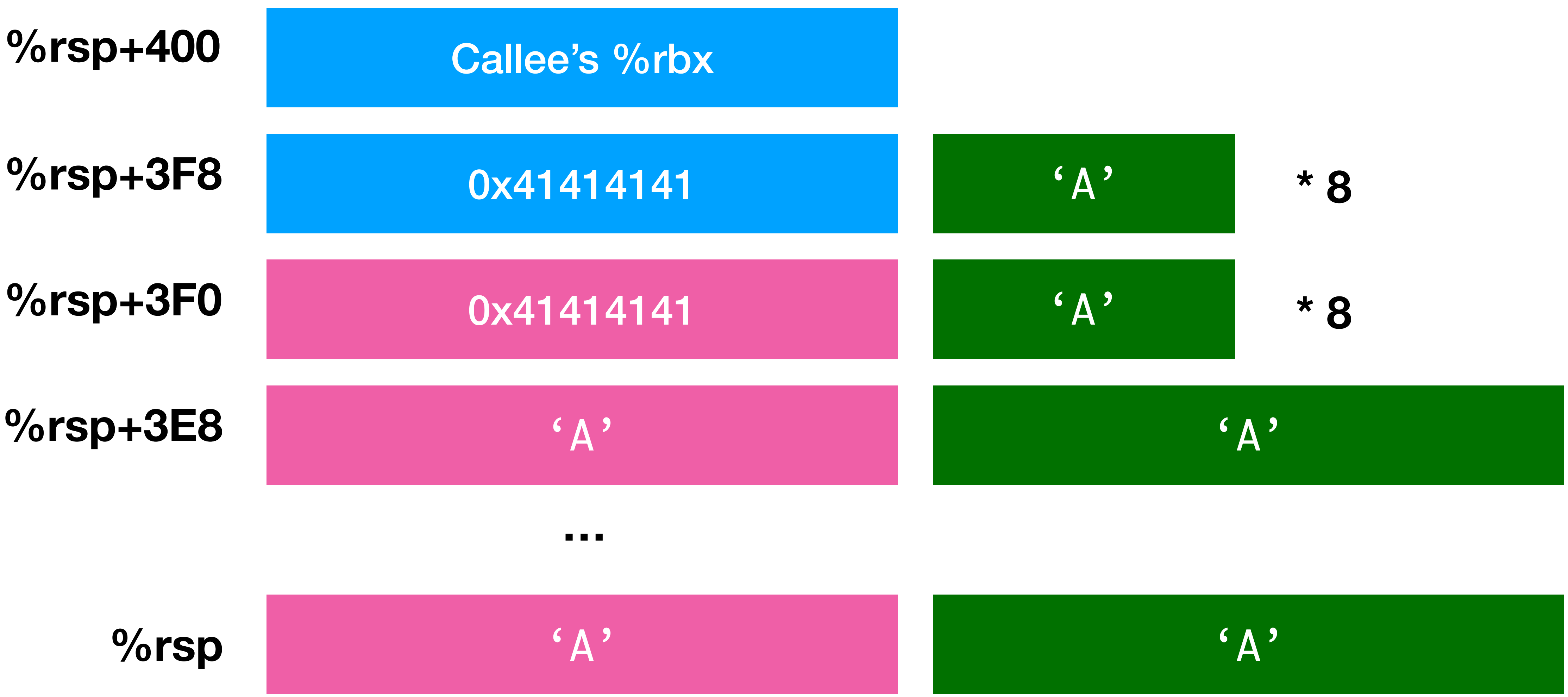
Consider what happens when `strcpy(buffer,ptr)`



Consider what happens when `strcpy(buffer,ptr)`

(This one is fine..)

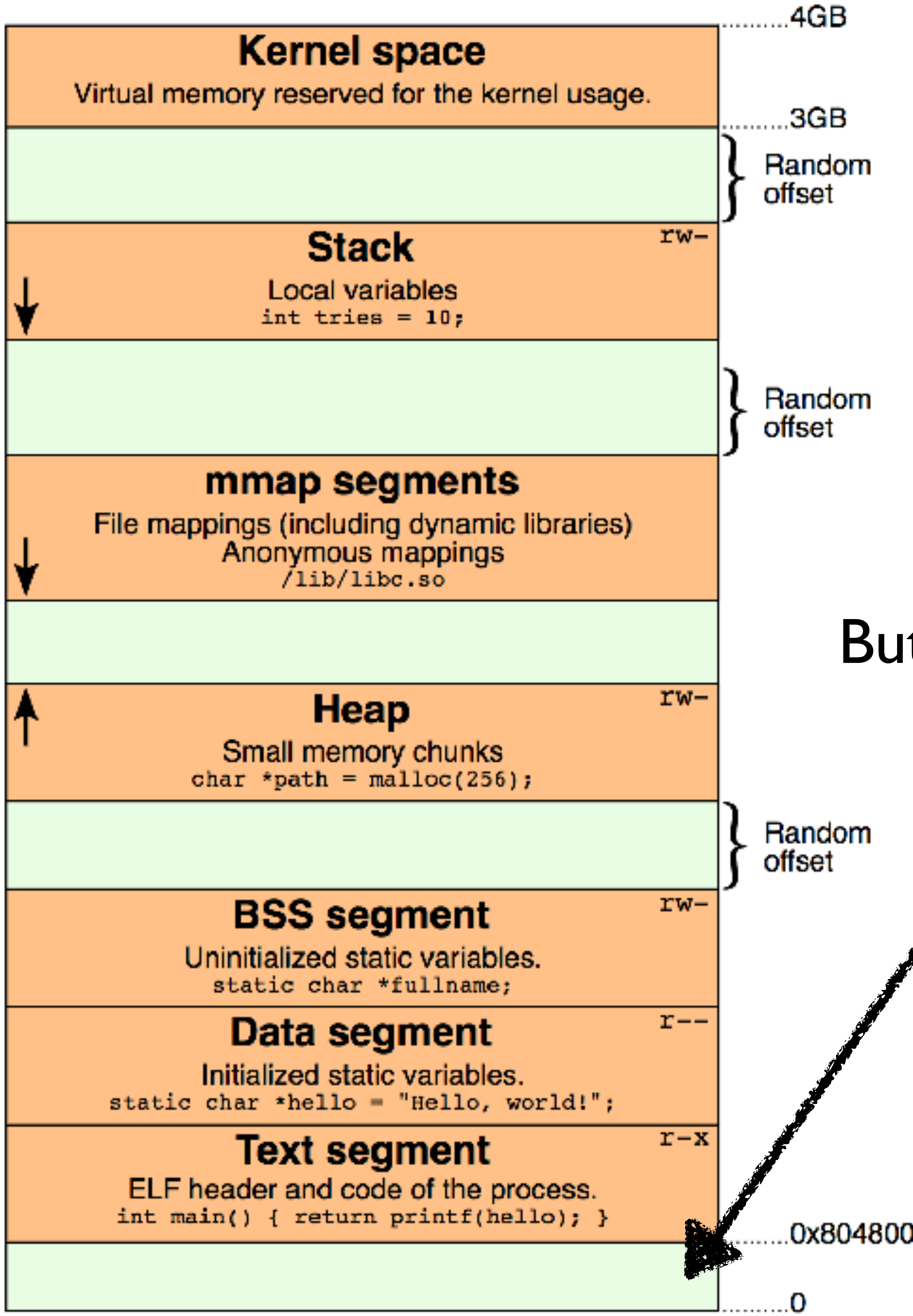
Now consider what happens when we provide input ‘A’ * 1008



Return addr becomes 0x41414141 (‘A’ four times)

Upon return, control goes to 0x41414141

If anything at this address, program will execute it



But falls in here, unmapped memory

Result: most common C crash

Segmentation Fault

The compiler translates binary code into machine code

execve("/bin/sh")

↓

Compiler

We'll cover this assembly later in class!

↓

```
"\x48\x31\xd2"                                  // xor    %rdx, %rdx
"\x48\xbb\x2f\x2f\x62\x69\x6e\x2f\x73\x68"    // mov $0x68732f6e69622f2f, %rbx
"\x48\xc1\xeb\x08"                              // shr    $0x8, %rbx
"\x53"                                          // push   %rbx
"\x48\x89\xe7"                                  // mov    %rsp, %rdi
"\x50"                                          // push   %rax
"\x57"                                          // push   %rdi
"\x48\x89\xe6"                                  // mov    %rsp, %rsi
"\xb0\x3b"                                      // mov    $0x3b, %al
"\x0f\x05";                                     // syscall
```

man execve

All that code is **loaded** by the kernel at a specific place in memory

Let’s assume for a second that the compiler loads that code at `0x41414141`

In the next few slides we’ll see what happens if it’s **not** there

Return pointer: 0x41414141

After returning, we *expect* the code to go back here

```
// foo's caller
foo(p);
x = x+1;

void foo(char *ptr) {
    char buffer[ptr];
    strcpy(buffer, ptr);
    printf("length: %d\n", strlen(buffer));
}
```

0x41414141

```
"\x48\x31\xd2"                                  // xor    %rdx, %rdx
"\x48\xbb\x2f\x2f\x62\x69\x6e\x2f\x73\x68"      // mov $0x68732f6e69622f2f, %rbx
"\x48\xc1\xeb\x08"                              // shr    $0x8, %rbx
"\x53"                                          // push   %rbx
"\x48\x89\xe7"                                  // mov    %rsp, %rdi
"\x50"                                          // push   %rax
"\x57"                                          // push   %rdi
"\x48\x89\xe6"                                  // mov    %rsp, %rsi
"\xb0\x3b"                                      // mov    $0x3b, %al
"\x0f\x05";                                     // syscall
```

Return pointer: 0x41414141

But the return address has been *overwritten* (stack has been smashed)

```
// foo's caller
foo(p);
x = x+1;

void foo(char *ptr) {
    char buffer[ptr];
    strcpy(buffer, ptr);
    printf("length: %d\n", strlen(buffer));
}
```

Instead, return goes here

0x41414141

```
"\x48\x31\xd2"                              // xor    %rdx, %rdx
"\x48\xbb\x2f\x2f\x62\x69\x6e\x2f\x73\x68"  // mov $0x68732f6e69622f2f, %rbx
"\x48\xc1\xeb\x08"                          // shr    $0x8, %rbx
"\x53"                                      // push   %rbx
"\x48\x89\xe7"                              // mov    %rsp, %rdi
"\x50"                                      // push   %rax
"\x57"                                      // push   %rdi
"\x48\x89\xe6"                              // mov    %rsp, %rsi
"\xb0\x3b"                                  // mov    $0x3b, %al
"\x0f\x05";                                 // syscall
```

Now, the computer executes a shell instead!!!

Might not be so bad if it's a local program

But bad if it's a connection to a remote server!

In your first project, you'll mount one of these attacks on a vulnerable file server

So my job **as an attacker** is to find a buffer overflow in the program and then craft an input that sends the code where I want

Question 1: How do I find a bug?

A: Dig through the source manually, if source is available

(If source unavailable, use a ***decompiler***)

A: Some automated testing tools

So my job **as an attacker** is to find a buffer overflow in the program and then craft an input that sends the code where I want

Question 2: What if program doesn't have bugs!?

A: You're hosed, can't perform this attack

But some other attacks we'll talk about on Thursday

The best way to prevent these attacks is to write in languages where these bugs can't occur!!

So my job **as an attacker** is to find a buffer overflow in the program and then craft an input that sends the code where I want

Question 3: How do I know what code to execute?

A: Find the code you want in the binary

A: We'll also learn how you can **inject your own** code

So my job **as an attacker** is to find a buffer overflow in the program and then craft an input that sends the code where I want

Question 4: How do I know **where the code is**

A: Use GDB to find it after booting up the binary

But there's a critical catch!