CS 3650 Computer Systems – Spring 2023

Memory, stack, and recursion

Week 3

Northeastern University

* Acknowledgements: created based on Christo Wilson, Ferdinand Vesely, and Alden Jackson’s lecture slides for the same course.

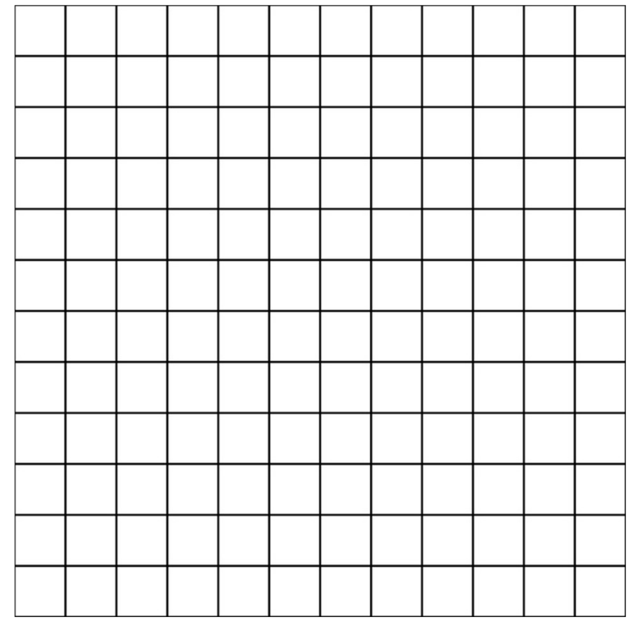
Memory on our machines

- The memory in our machines stores data so we can recall it later
- This occurs at several different levels
  - Networked drive (or cloud storage)
  - Hard drive
  - Dynamic memory
  - Cache
- For now, we can think of memory as a giant linear array.

Linear array of memory

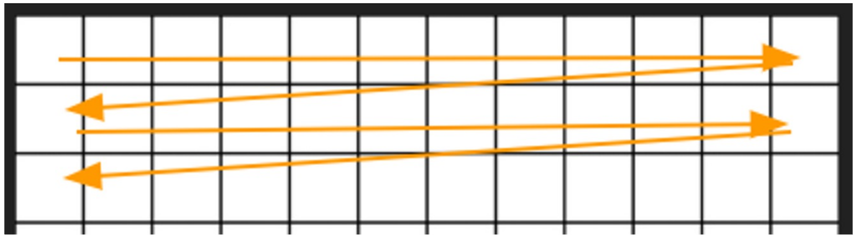
- Each ‘box’ here we will say is 1 byte of memory
  - (1 byte = 8 bits on most systems)

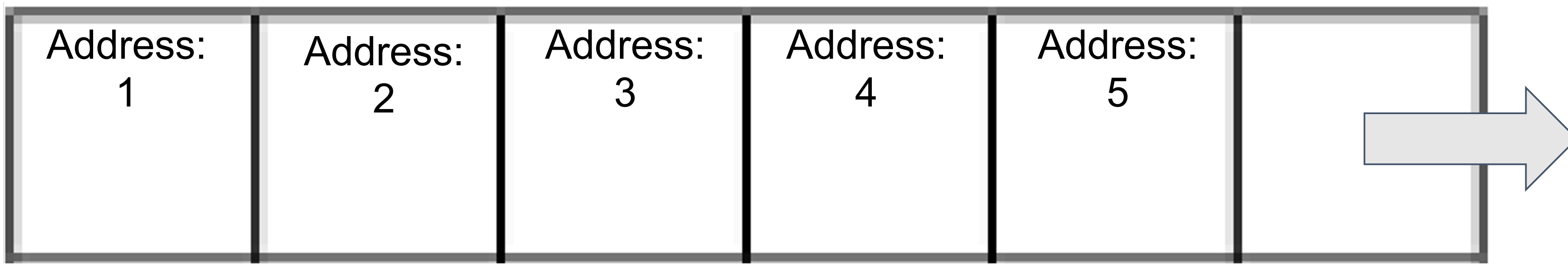
- Depending on the data we store,
  we will need 1 byte, 2 bytes, 4 bytes, etc.
  of memory



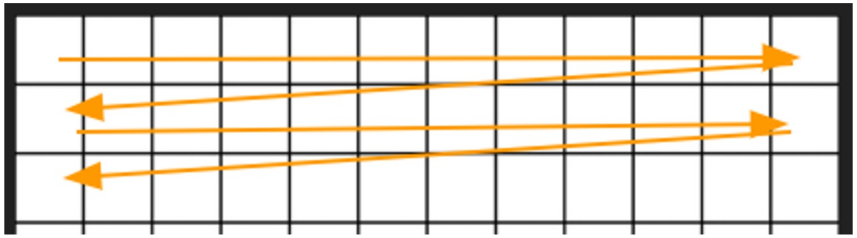
Linear array of memory

- Visually I have organized memory in a grid, but memory is really a linear array as depicted below.
  - There is one address after the other

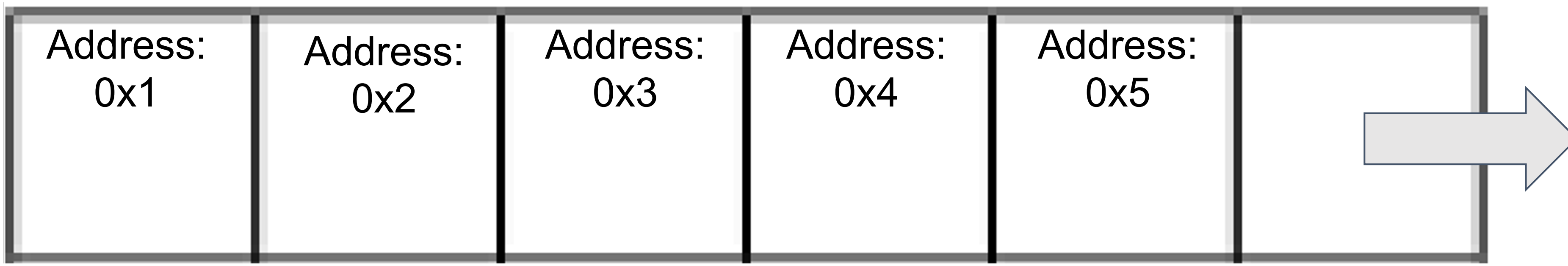




Linear array of memory



- Visually I have organized memory in a grid, but memory is really a linear array as depicted below.
  - There is one address after the other
  - Because these addresses grow large, typically we represent them in hexadecimal (16-base number system: a digit can be 0-9 and A-F)
    - (https://www.rapidtables.com/convert/number/hex-to-decimal.html)



Remember: “Everything is a number”

Data Type	Suffix	Bytes	Range (unsigned)
`char`	**b**	1	0 to 255 (=2^8)
`short int`	**w**	2	0 to 65,535 (=2^16)
`int`	**l**	4	0 to 4,294,967,295 (=2^32)
`long int`	**q**	8	0 to 18,446,744,073,709,551,615 (=2^64)

Addressing memory

- Address granularity: **bytes**
- Suppose we are looking at a chunk of memory
- First address we see: 0x41F00 (in hexadecimal)
- This diagram: each row shows 8 bytes (aka one quadword = 64 bits)

...

0x41F00	00	01	02	03	04	05	06	07
0x41F08	08	09	0A	0B	0C	0D	0E	0F
0x41F10	10	11	12	13	14	15	16	17
0x41F18	18	19	1A	1B	1C	1D	1E	1F
0x41F20	20	21	22	23	24	25	26	27
0x41F28	28	29	2A	2B	2C	2D	2E	2F
0x41F30	30	31	32	33	34	35	36	37
0x41F38	38	39	3A	3B	3C	3D	3E	3F

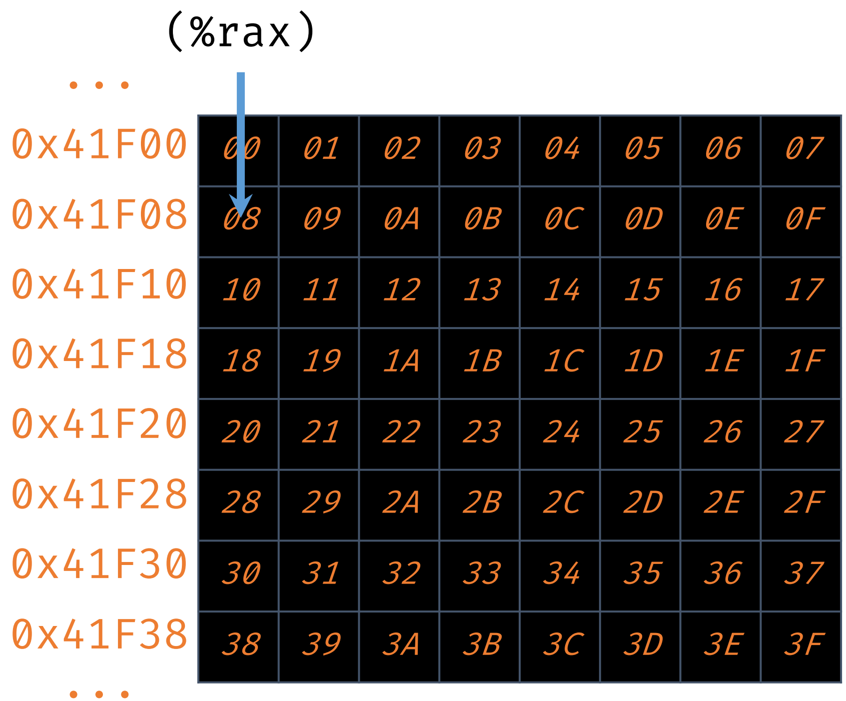
...

Addressing memory

```
mov $0x41F08, %rax
```

We move the address 0x41F08 into rax

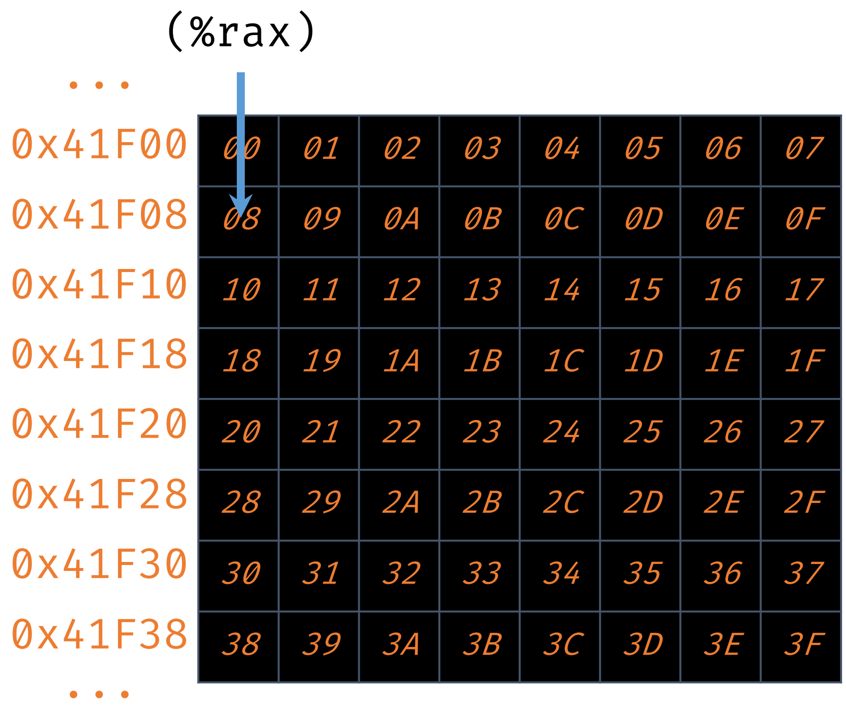
(%rax) now points to the contents of the corresponding chunk of memory



Addressing memory

Offset addressing:

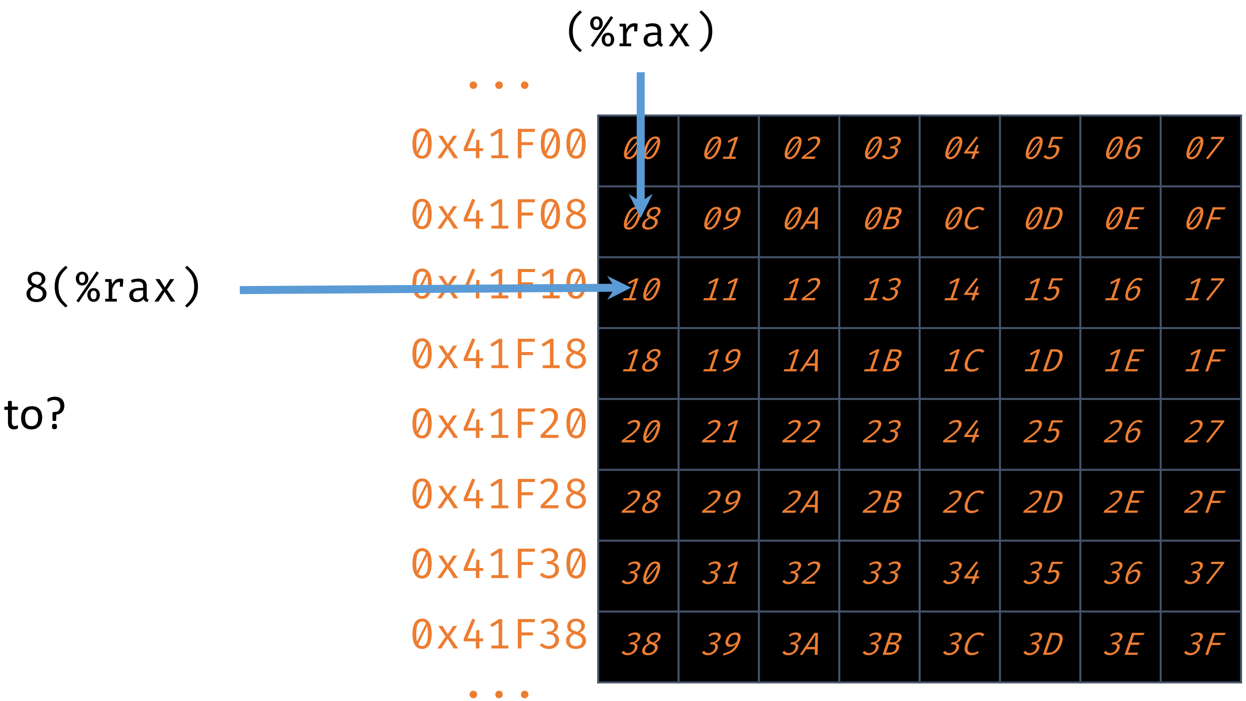
- We can point to addresses by adjusting the pointer register by an offset



Addressing memory

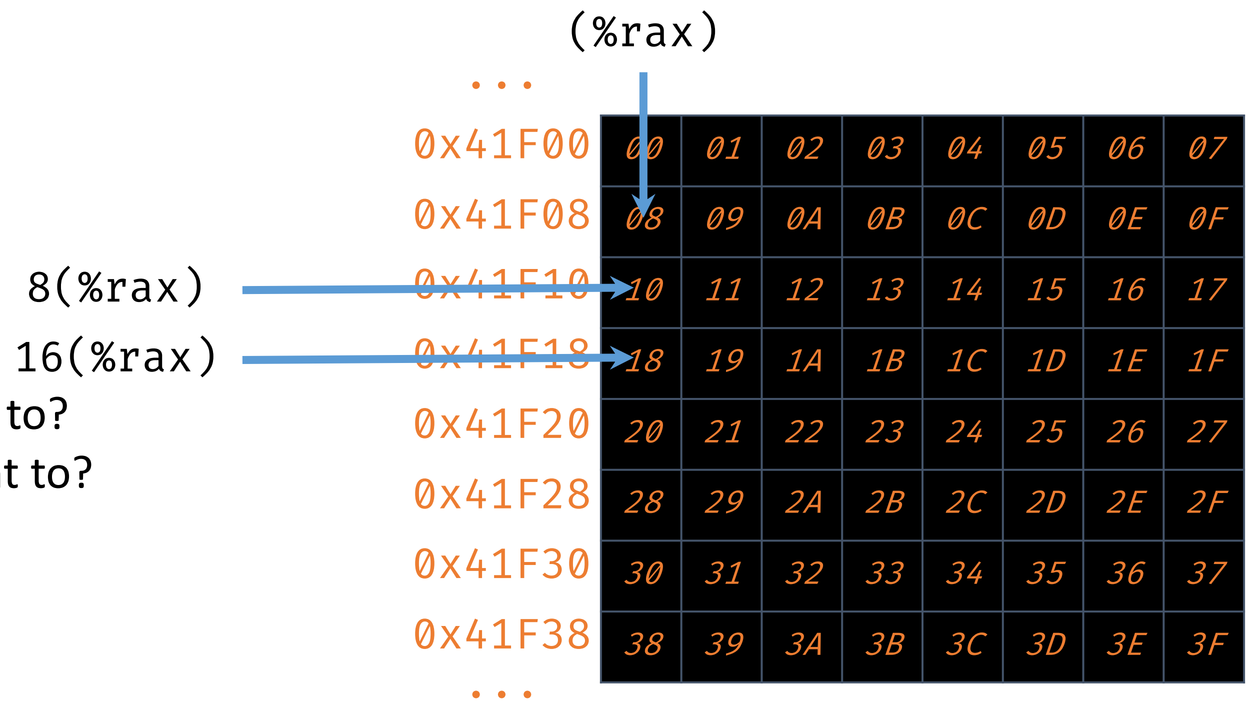
Offset addressing

Where does 8(%rax) point to?



Addressing memory

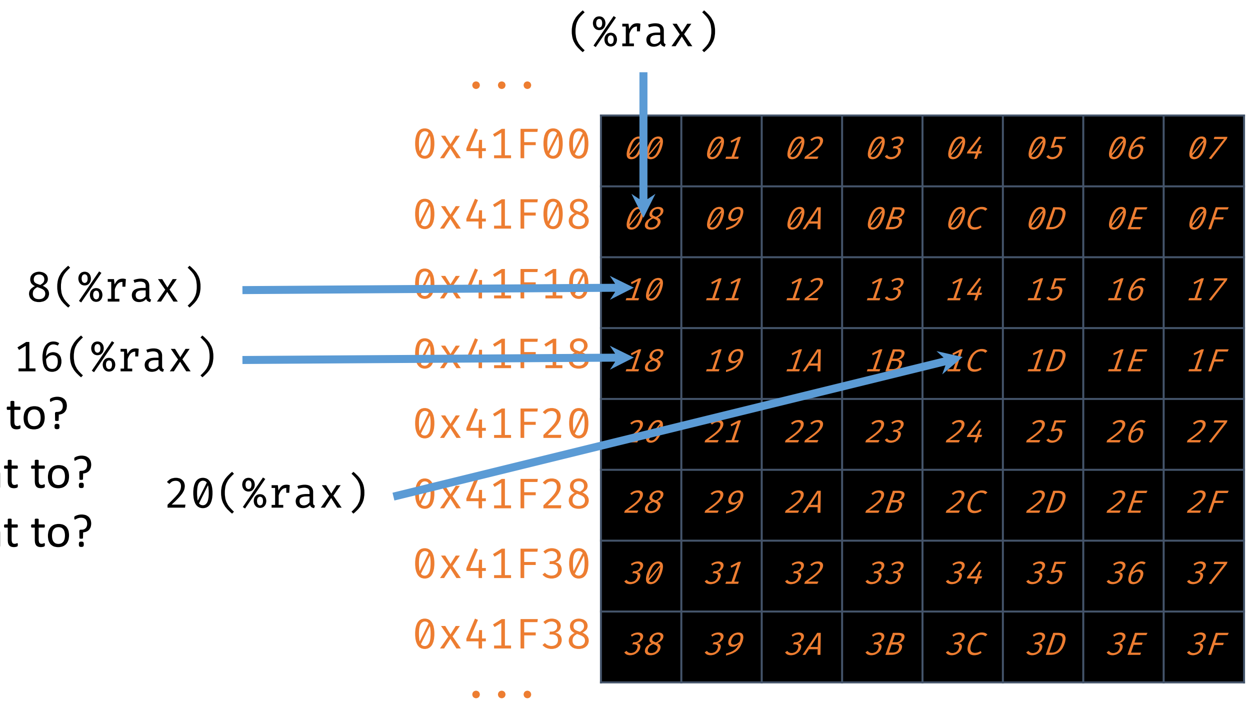
Offset addressing



Where does 8(%rax) point to?
Where does 16(%rax) point to?

Addressing memory

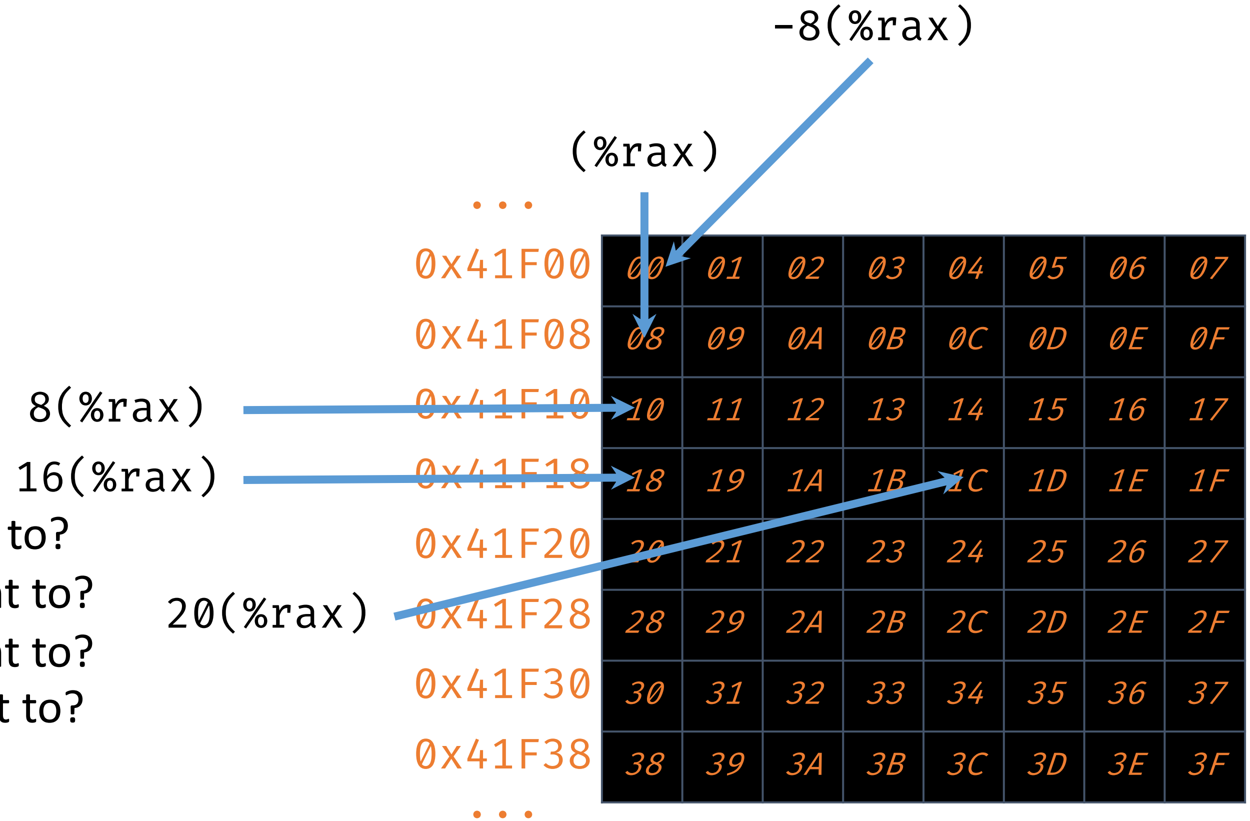
Offset addressing



Where does 8(%rax) point to?
Where does 16(%rax) point to?
Where does 20(%rax) point to?

Addressing memory

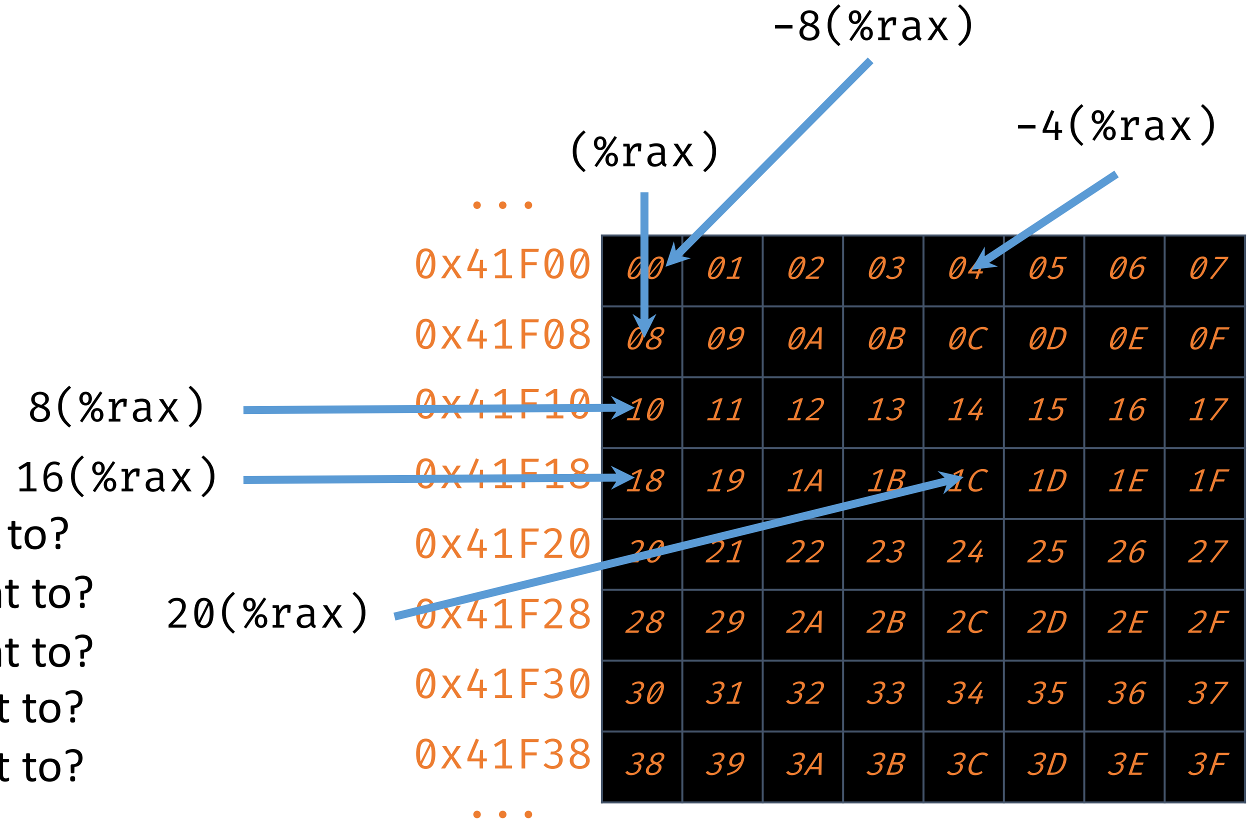
Offset addressing



Where does 8(%rax) point to?
Where does 16(%rax) point to?
Where does 20(%rax) point to?
Where does -8(%rax) point to?

Addressing memory

Offset addressing

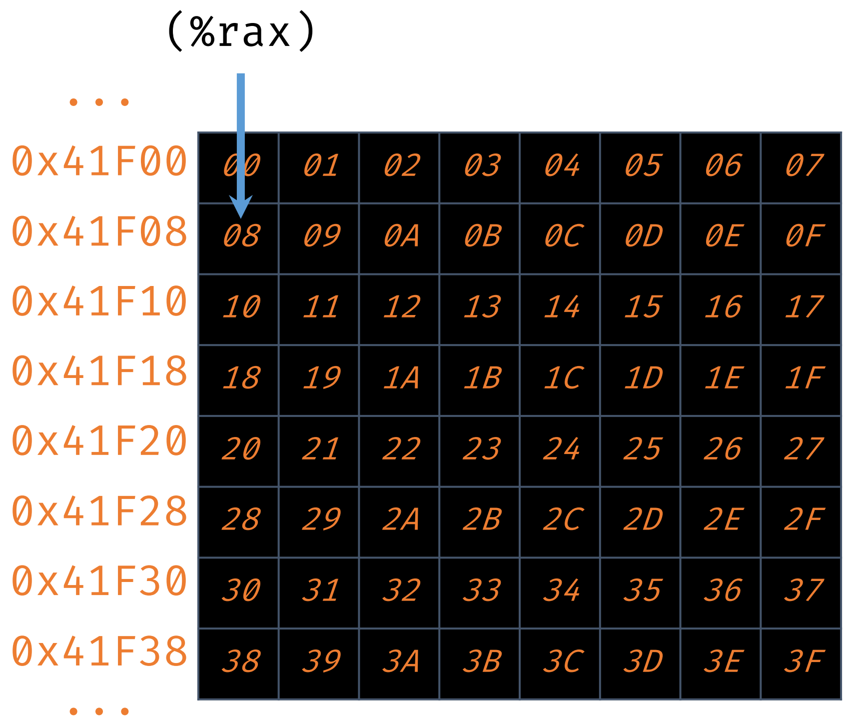


Where does 8(%rax) point to?
Where does 16(%rax) point to?
Where does 20(%rax) point to?
Where does -8(%rax) point to?
Where does -4(%rax) point to?

Addressing memory

`mov $0x1020304050607080, (%rax)`

What does this look like in memory?



Addressing memory

`mov $0x1020304050607080, (%rax)`

What does this look like in memory?

Like this?

(%rax)

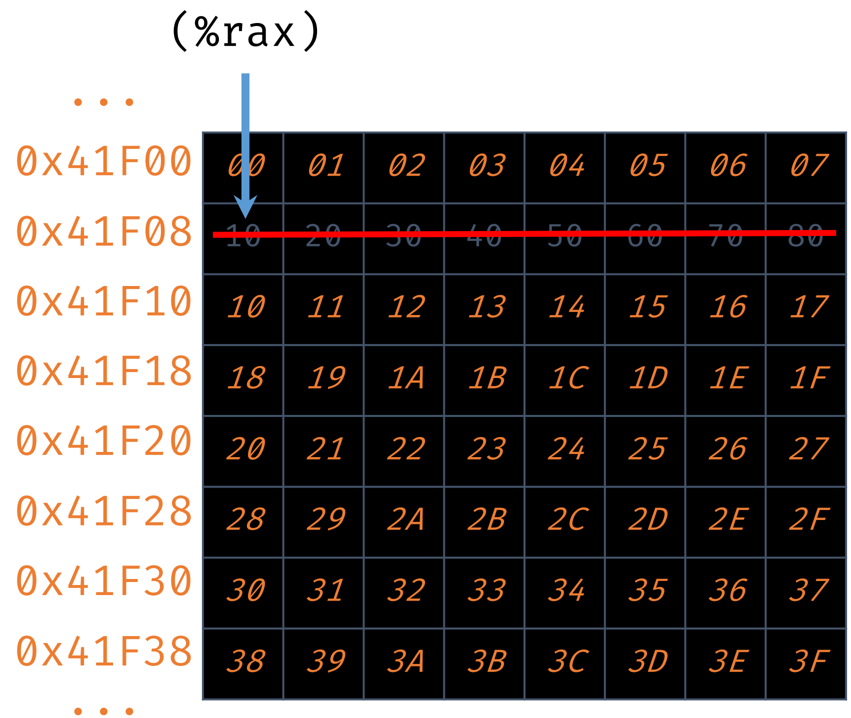
...								
0x41F00	00	01	02	03	04	05	06	07
0x41F08	10	20	30	40	50	60	70	80
0x41F10	10	11	12	13	14	15	16	17
0x41F18	18	19	1A	1B	1C	1D	1E	1F
0x41F20	20	21	22	23	24	25	26	27
0x41F28	28	29	2A	2B	2C	2D	2E	2F
0x41F30	30	31	32	33	34	35	36	37
0x41F38	38	39	3A	3B	3C	3D	3E	3F
...								

Addressing memory

```
mov $0x1020304050607080, (%rax)
```

What does this look like in memory?

Like this? **NO**



Addressing memory

`mov $0x1020304050607080, (%rax)`

What does this look like in memory?

Like this? **NO**

→ x86 is *little-endian*: the less significant bytes are stored at lesser addresses

(end byte of the number, 0x80, is little)

(%rax)

...

0x41F00	00	01	02	03	04	05	06	07
0x41F08	~~10~~	~~20~~	~~30~~	~~40~~	~~50~~	~~60~~	~~70~~	~~80~~
0x41F10	10	11	12	13	14	15	16	17
0x41F18	18	19	1A	1B	1C	1D	1E	1F
0x41F20	20	21	22	23	24	25	26	27
0x41F28	28	29	2A	2B	2C	2D	2E	2F
0x41F30	30	31	32	33	34	35	36	37
0x41F38	38	39	3A	3B	3C	3D	3E	3F

...

Addressing memory

```
mov $0x1020304050607080, (%rax)
```

What does this look like in memory?

Like this.

(%rax)

...

0x41F00	*00*	*01*	*02*	*03*	*04*	*05*	*06*	*07*
0x41F08	80	70	60	50	40	30	20	10
0x41F10	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*
0x41F18	*18*	*19*	*1A*	*1B*	*1C*	*1D*	*1E*	*1F*
0x41F20	*20*	*21*	*22*	*23*	*24*	*25*	*26*	*27*
0x41F28	*28*	*29*	*2A*	*2B*	*2C*	*2D*	*2E*	*2F*
0x41F30	*30*	*31*	*32*	*33*	*34*	*35*	*36*	*37*
0x41F38	*38*	*39*	*3A*	*3B*	*3C*	*3D*	*3E*	*3F*

...

Addressing memory

`movq (%rax), %r10`

Copies the contents of the address pointed to by (%rax) to %r10

`movq %rax, %r11`

Copies the contents of %rax to %r11. Now (%rax) and (%r11) point to the same location.

(%rax)

...

0x41F00	00	01	02	03	04	05	06	07
0x41F08	80	70	60	50	40	30	20	10
0x41F10	10	11	12	13	14	15	16	17
0x41F18	18	19	1A	1B	1C	1D	1E	1F
0x41F20	20	21	22	23	24	25	26	27
0x41F28	28	29	2A	2B	2C	2D	2E	2F
0x41F30	30	31	32	33	34	35	36	37
0x41F38	38	39	3A	3B	3C	3D	3E	3F

...

Addressing memory

```
movl (%rax), %ebx
```

What's in %ebx?

How much we move is determined by operand sizes / suffixes

(%rax) → 0x41F08

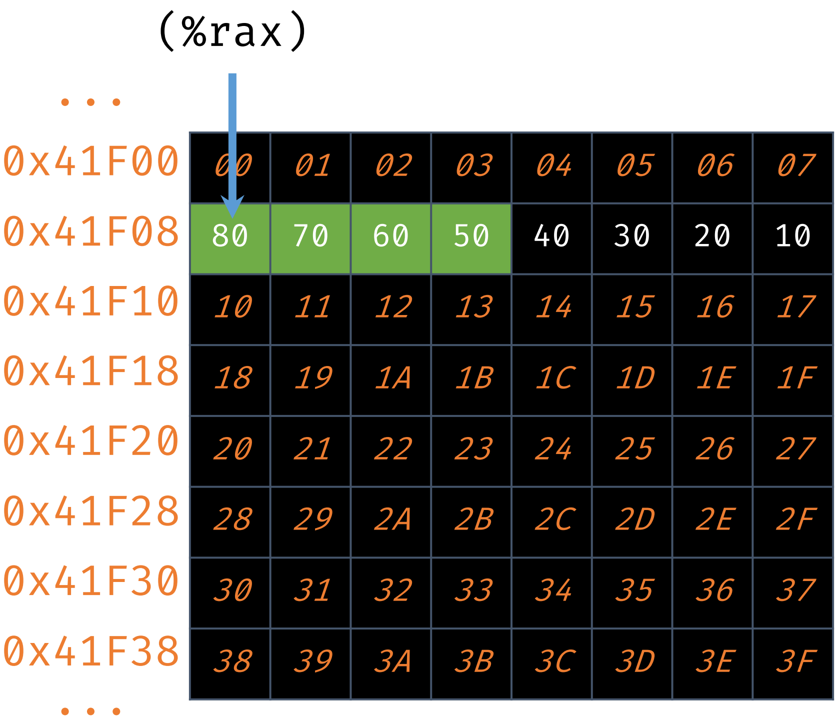
...								
0x41F00	00	01	02	03	04	05	06	07
0x41F08	80	70	60	50	40	30	20	10
0x41F10	10	11	12	13	14	15	16	17
0x41F18	18	19	1A	1B	1C	1D	1E	1F
0x41F20	20	21	22	23	24	25	26	27
0x41F28	28	29	2A	2B	2C	2D	2E	2F
0x41F30	30	31	32	33	34	35	36	37
0x41F38	38	39	3A	3B	3C	3D	3E	3F
...								

Addressing memory

`movl (%rax), %ebx`

What’s in %ebx?

0x50607080



Addressing memory

```
movw 4(%rax), %bx
```

What's in %bx?

(%rax)

...								
0x41F00	00	01	02	03	04	05	06	07
0x41F08	80	70	60	50	40	30	20	10
0x41F10	10	11	12	13	14	15	16	17
0x41F18	18	19	1A	1B	1C	1D	1E	1F
0x41F20	20	21	22	23	24	25	26	27
0x41F28	28	29	2A	2B	2C	2D	2E	2F
0x41F30	30	31	32	33	34	35	36	37
0x41F38	38	39	3A	3B	3C	3D	3E	3F
...								

Addressing memory

```
movw 4(%rax), %bx
```

What's in %bx?

0x3040

(%rax)

...

0x41F00	00	01	02	03	04	05	06	07
0x41F08	80	70	60	50	40	30	20	10
0x41F10	10	11	12	13	14	15	16	17
0x41F18	18	19	1A	1B	1C	1D	1E	1F
0x41F20	20	21	22	23	24	25	26	27
0x41F28	28	29	2A	2B	2C	2D	2E	2F
0x41F30	30	31	32	33	34	35	36	37
0x41F38	38	39	3A	3B	3C	3D	3E	3F

...

Addressing memory

```
movb 6(%rax), %bl
```

What's in %bl?

(%rax)

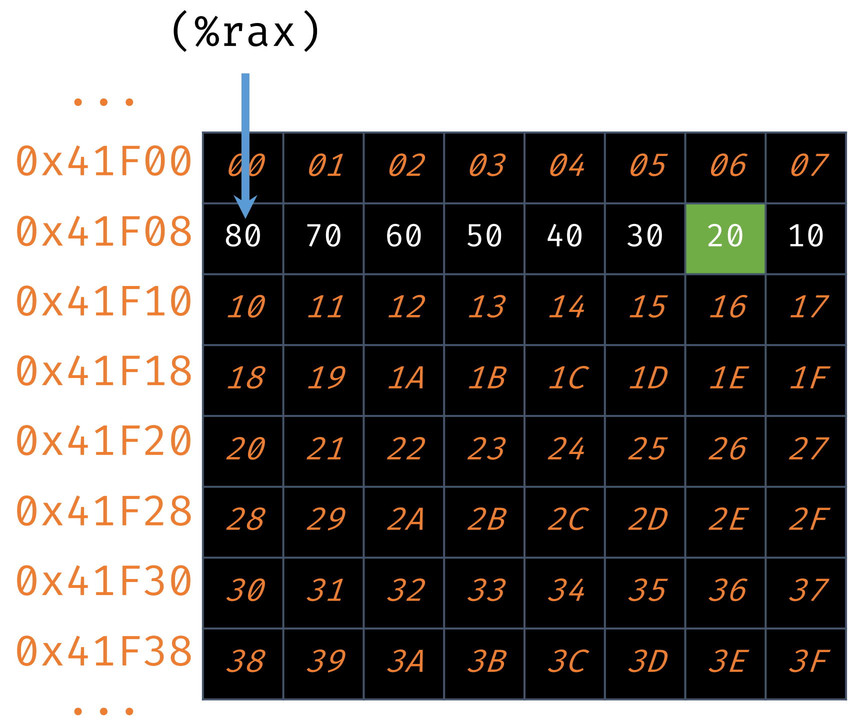
...								
0x41F00	00	01	02	03	04	05	06	07
0x41F08	80	70	60	50	40	30	20	10
0x41F10	10	11	12	13	14	15	16	17
0x41F18	18	19	1A	1B	1C	1D	1E	1F
0x41F20	20	21	22	23	24	25	26	27
0x41F28	28	29	2A	2B	2C	2D	2E	2F
0x41F30	30	31	32	33	34	35	36	37
0x41F38	38	39	3A	3B	3C	3D	3E	3F
...								

Addressing memory

`movb 6(%rax), %bl`

What's in %bl?

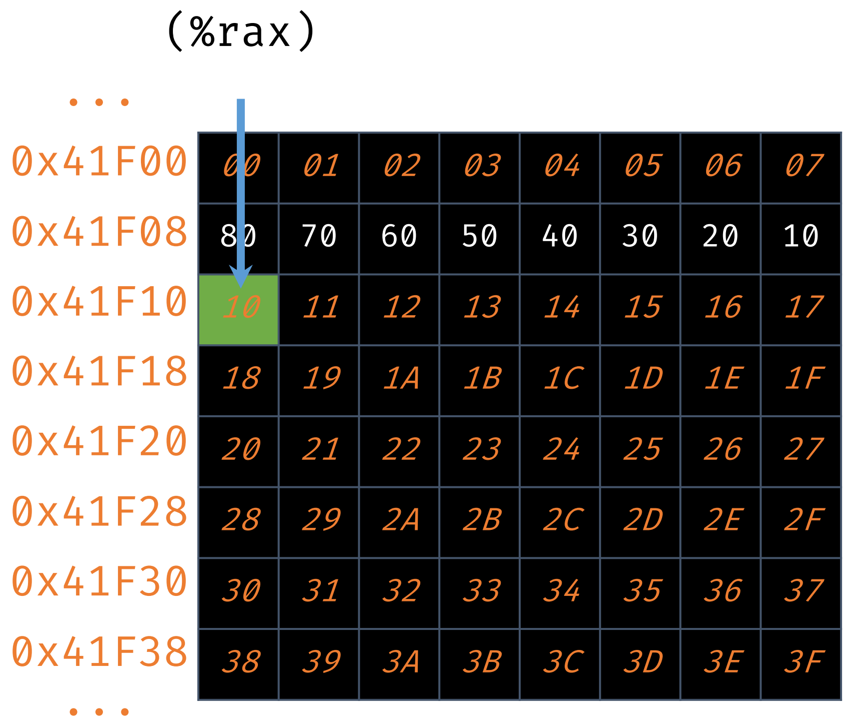
0x20



Northeastern University

Addressing memory

```
add $8, %rax
```

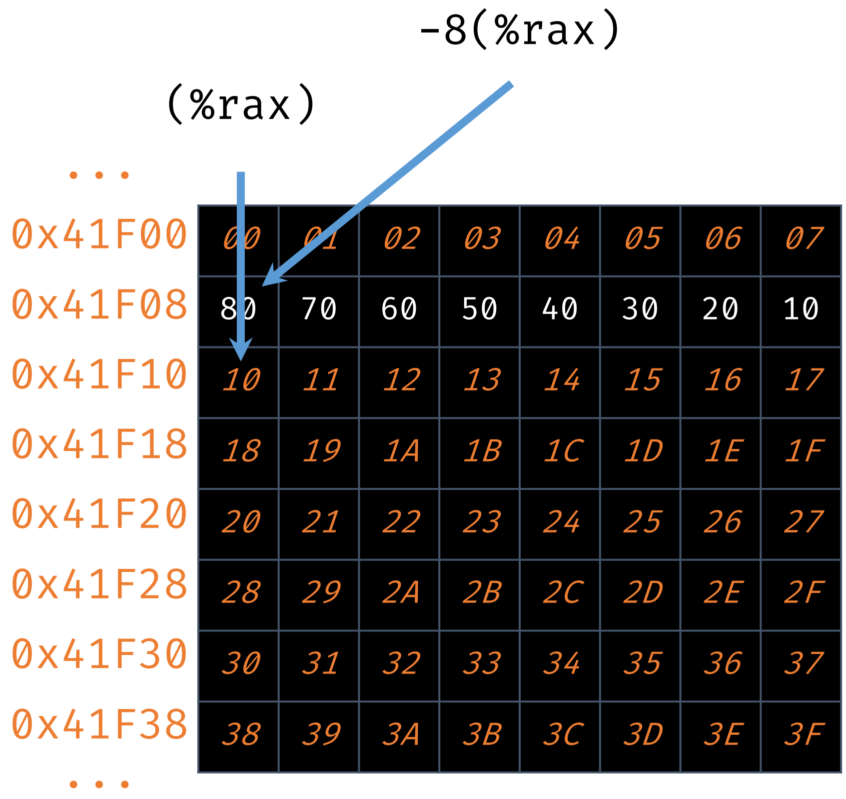


Modifying %rax changes where it points

Addressing memory

```
add $8, %rax
```

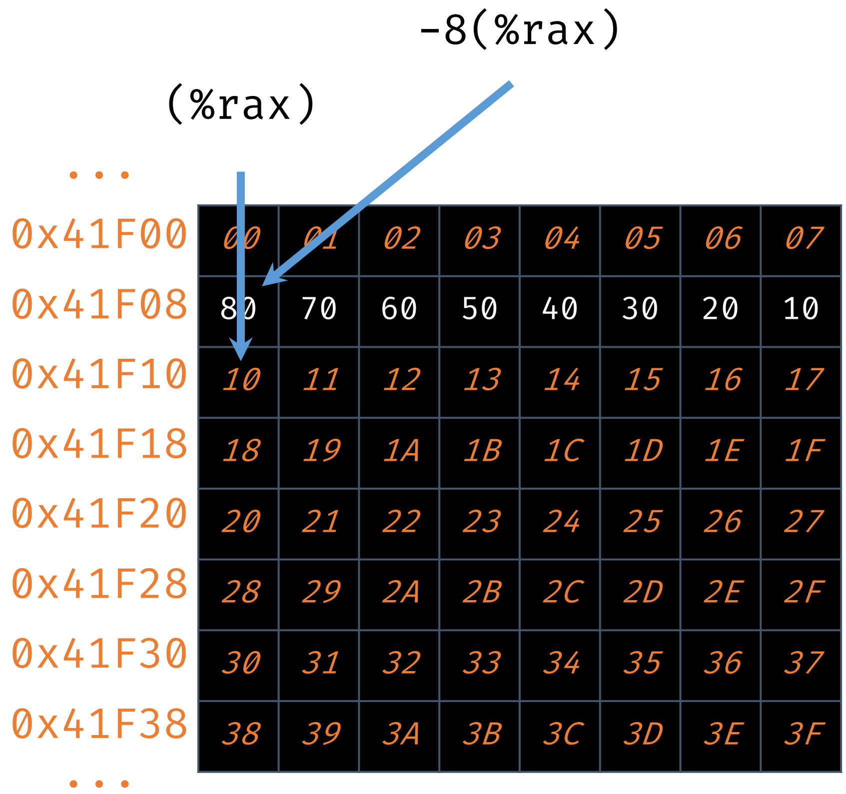
Modifying %rax changes where it points



Addressing memory

```
add $8, %rax
movq $0x42, (%rax)
```

How does movq change the memory state?



Addressing memory

```
add $8, %rax
movq $0x42, (%rax)
```

(%rax)

...								
0x41F00	*00*	*01*	*02*	*03*	*04*	*05*	*06*	*07*
0x41F08	80	70	60	50	40	30	20	10
0x41F10	42	00	00	00	00	00	00	00
0x41F18	*18*	*19*	*1A*	*1B*	*1C*	*1D*	*1E*	*1F*
0x41F20	*20*	*21*	*22*	*23*	*24*	*25*	*26*	*27*
0x41F28	*28*	*29*	*2A*	*2B*	*2C*	*2D*	*2E*	*2F*
0x41F30	*30*	*31*	*32*	*33*	*34*	*35*	*36*	*37*
0x41F38	*38*	*39*	*3A*	*3B*	*3C*	*3D*	*3E*	*3F*
...								

Modifying %rax changes where it points

Addressing memory: full syntax

```
displacement(base, index, scale)
```

```
ADDRESS = base + (index * scale) + displacement
```

Mostly used for addressing arrays:

displacement: (immediate) offset / adjustment (e.g., -8, 8, 4, …)
base: (register) base pointer (%rax in previous examples)
index: (register) index of element
scale: (immediate) size of an element

Addressing memory: full syntax

`displacement(base, index, scale)`

`ADDRESS = base + (index * scale) + displacement`

Mostly used for addressing arrays:

displacement: (immediate) offset / adjustment (e.g., -8, 8, 4, …)
base: (register) base pointer (%rax in previous examples)
index: (register) index of element
scale: (immediate) size of an element

> Note:
> `8(%rax)` is equivalent to `8(%rax, 0, 0)`

Addressing memory: full syntax

```
    mov $0x41F00, %rax

    mov $0, %rcx
    mov $0, %r10

loop:
    cmp $8, %rcx
    jge loop_end

    add (%rax, %rcx, 8), %r10
    inc %rcx
    jmp loop
```

What's in %r10 after loop_end?

```
loop_end:
```

Address	Value
...	
0x41F00	01
0x41F08	02
0x41F10	03
0x41F18	04
0x41F20	05
0x41F28	06
0x41F30	07
0x41F38	08
...	

Procedures/Functions

Procedure Mechanisms

- Several things happen when calling a procedure (i.e., function or method)
- Pass control
  - Start executing from start of procedure
  - Return back to where we called from
- Pass data
  - Procedure arguments and the return value are passed
- Memory management
  - Memory allocated in the procedure, and then deallocated on return
- x86-64 uses the minimum subset required

x86-64 Memory Space

- Our view of a program is a giant byte array

- However, it is segmented into different regions
  - This separation is determined by the Application Binary Interface (ABI)

  - This is something typically chosen by the OS.

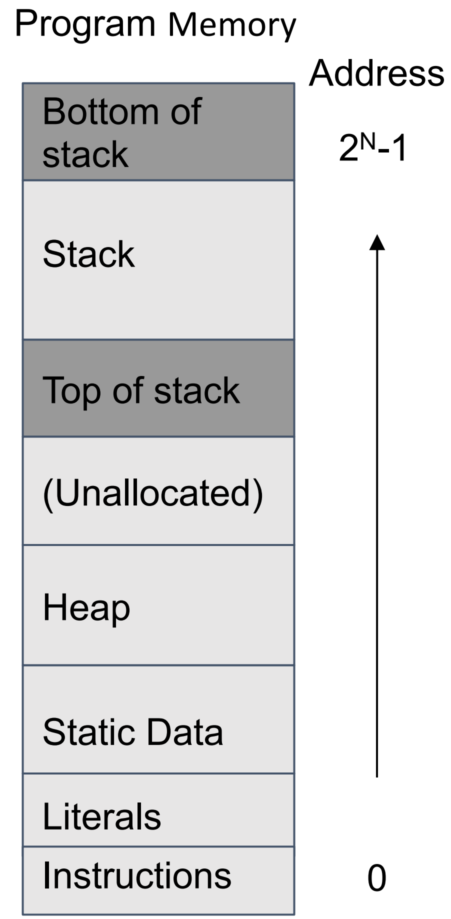
- We traverse our byte array as a stack

x86-64 Memory Space

Addresses grow up

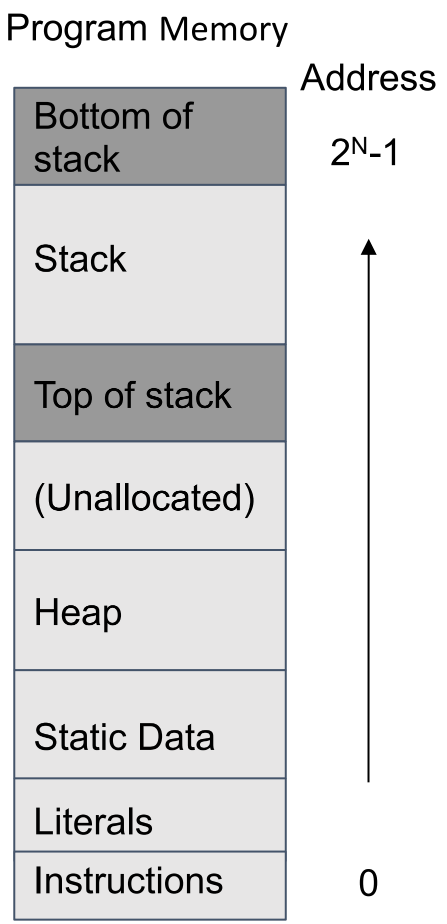
Our Program Memory Space is divided into several segments.

- Some parts of it are for long lived data (the heap)
- The other is for short-lived data (the stack) typically used for functions and local variables.

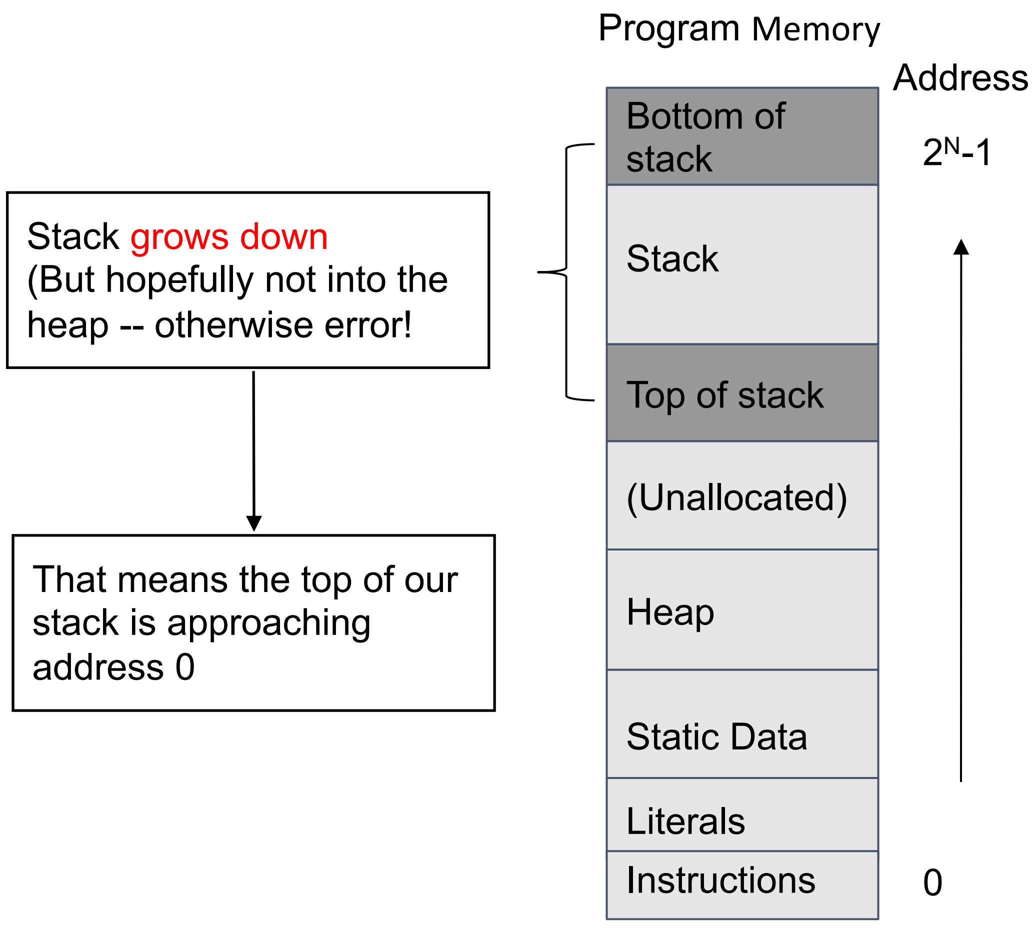


x86-64 stack

- There is a stack at the top of the memory
  - Yes, the stack that you learned in data structures course
  - You can push and pop data



x86-64 stack

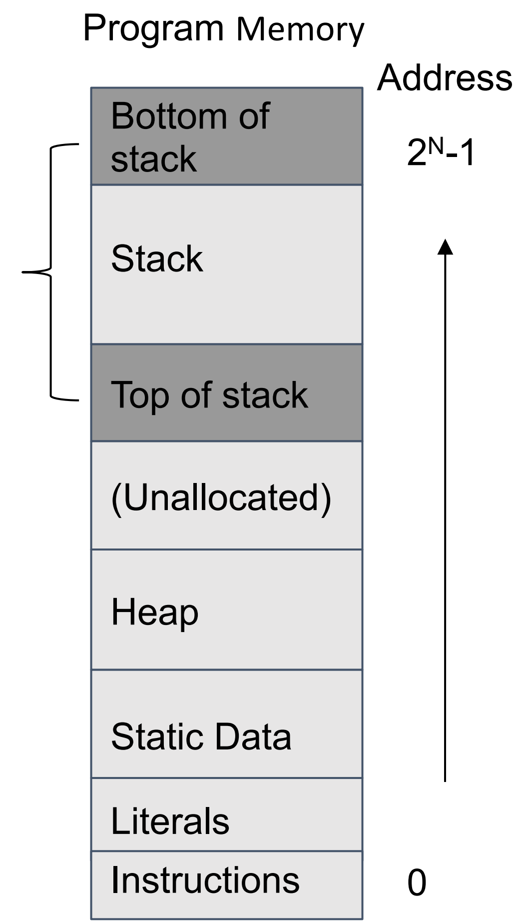

Stack grows down
(But hopefully not into the heap -- otherwise error!
That means the top of our stack is approaching address 0
Program Memory
Address
Bottom of stack
2^N-1
Stack
Top of stack
(Unallocated)
Heap
Static Data
Literals
Instructions
0

x86-64 stack

You'll observe things like `-8(%rsp)` in your assemble to remind you that things are *growing down* in the stack

Stack grows down
(But hopefully not into the heap -- otherwise error!

That means the top of our stack is approaching address 0

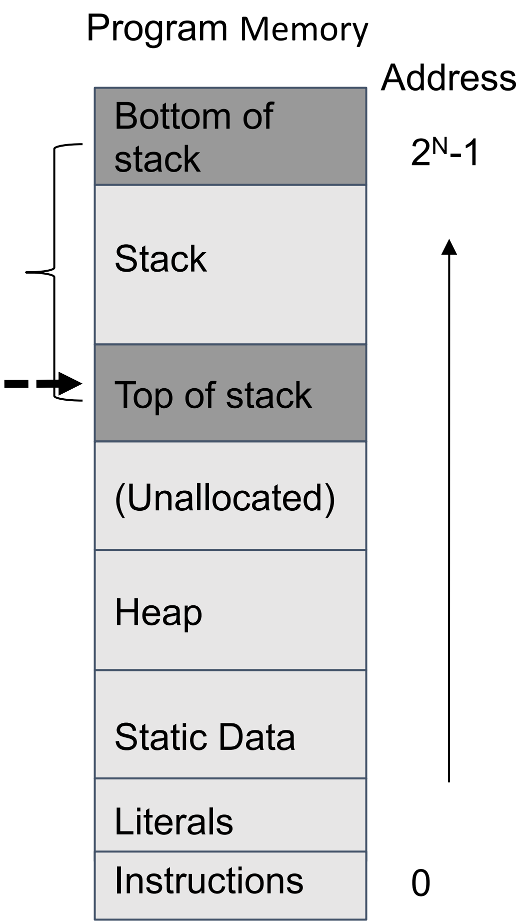


x86-64 stack

Stack Pointer: %rsp
Always contains lowest address

This is the “top” of the stack

You’ll observe things like `-8(%rsp)` in your assemble to remind you that things are *growing down* in the stack

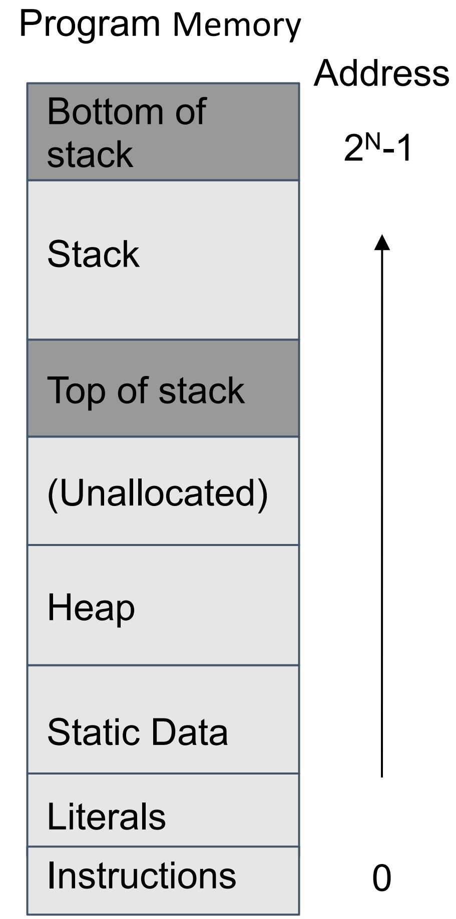


x86-64 stack

With a Stack data structure, we can perform two main operations

1. push data onto the stack (add information)
   a. Our stack grows
      a. Pushes data to top of the stack
      b. Moves the stack pointer downward

2. pop data off of the stack (remove information)
   a. Our stack shrinks
      a. Pops data from the top of the stack
      b. Moves the stack pointer upward

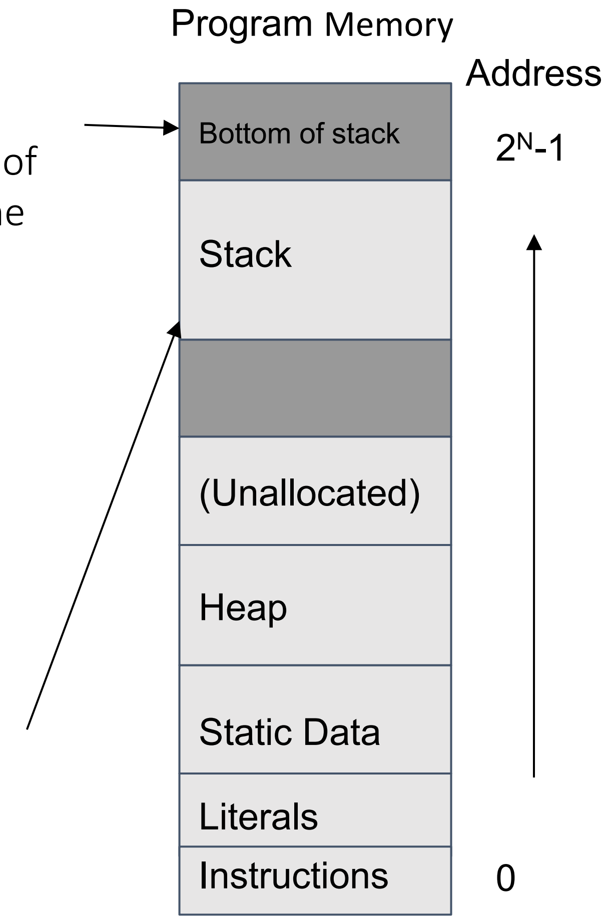


x86-64 stack | PUSHQ Example

- PUSHQ Src
  - Fetch operand at src
  - decrement %rsp by 8 (Q bytes)
  - Write operand at address given by %rsp

Base Pointer: %rbp
Always contains address of top of current stack frame

Stack Pointer: %rsp
Always contains lowest address in current stack frame

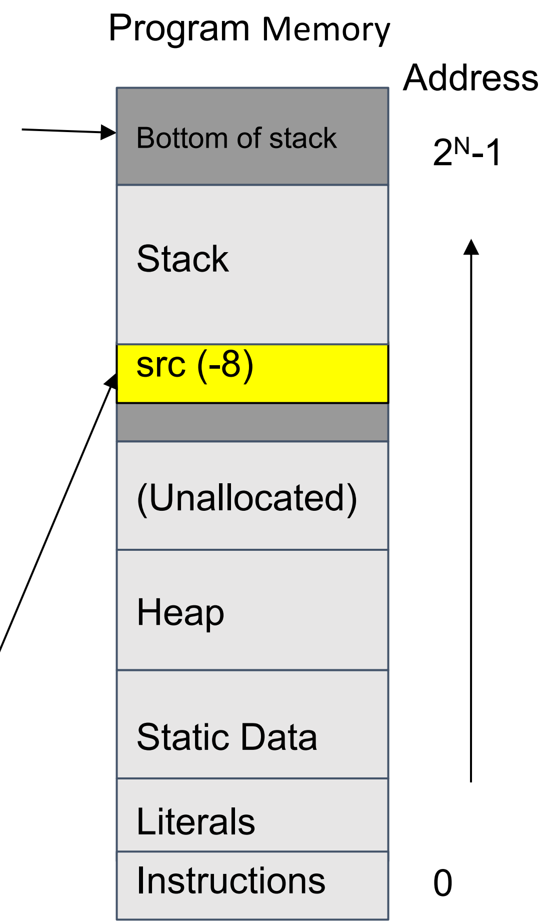


x86-64 stack | PUSHQ Example

- PUSHQ Src
  - Fetch operand at src
  - decrement %rsp by 8 (Q bytes)
  - Write operand at address given by %rsp
  - %rbp is unchanged

Base Pointer: %rbp
Always contains address of top of current stack frame

Stack Pointer: %rsp
Always contains lowest address in current stack frame

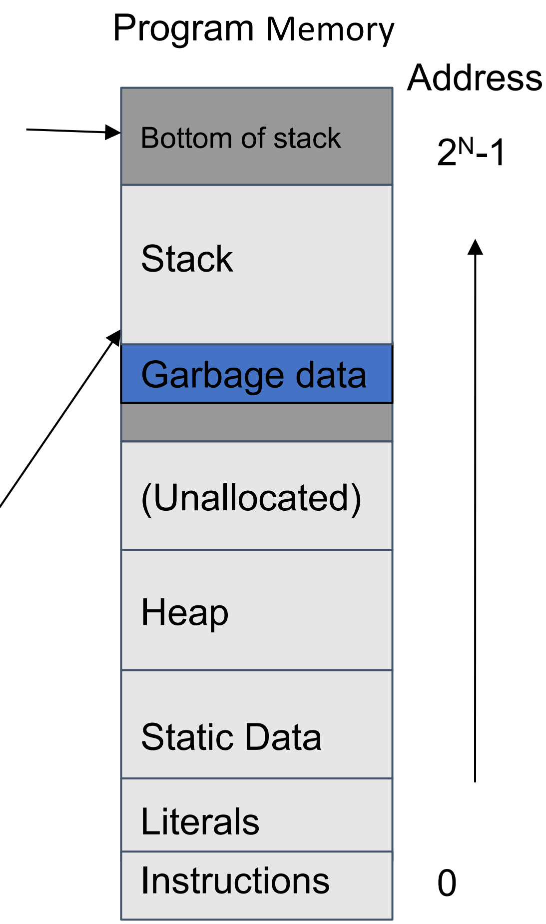


x86-64 stack | POPQ Example

- POPQ Dest
    - Read value at address given by %rsp
    - Increment %rsp by 8 (Q bytes)
    - Store value at Dest
    - %rbp unchanged

Base Pointer: %rbp
Always contains address of top of current stack frame

Stack Pointer: %rsp
Always contains lowest address



The Process Stack

- Each process has a stack in memory that stores:
  - Local variables
  - Arguments to functions
  - Return addresses from functions

- On x86:
  - The stack grows downwards

  - RSP (Stack Pointer) points to the bottom of the stack (= newest data)
  - RBP (Base Pointer) points to the base of the current frame

  - Instructions like push, pop, call, ret, int, and iret all modify the stack

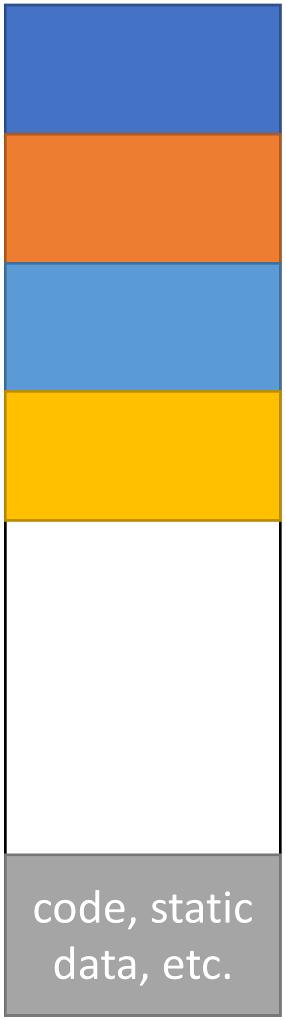
Creating and deleting stack frames for a function

```
void main(void) {
        …
        foo(x);
        baz(y);
}
```

```
void foo(int a) {
        …
        bar(z);
}
```

```
void bar(int b) {
        …
        baz(n);
}
```

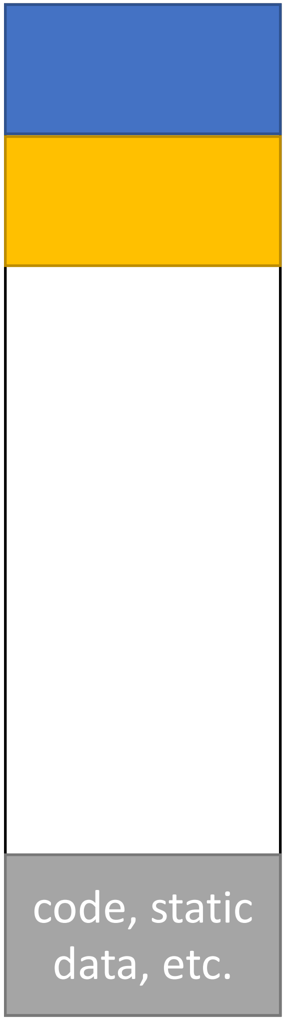
```
void baz(int c) {
        …
}
```



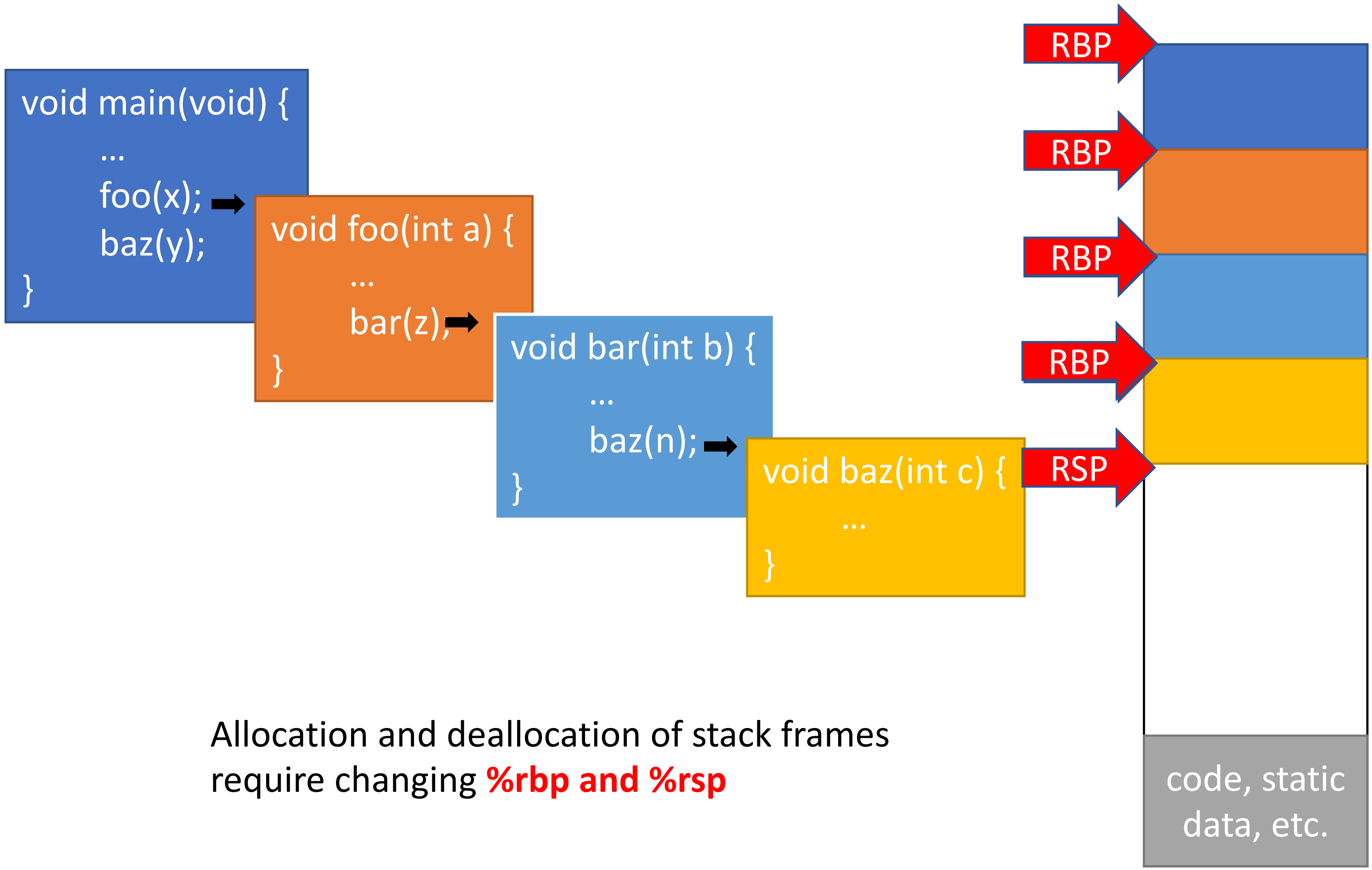
Creating and deleting stack frames for a function

```
void main(void) {
        …
        foo(x);
        baz(y);
}
```

```
void baz(int c) {
        …
}
```



Creating and deleting stack frames for a function



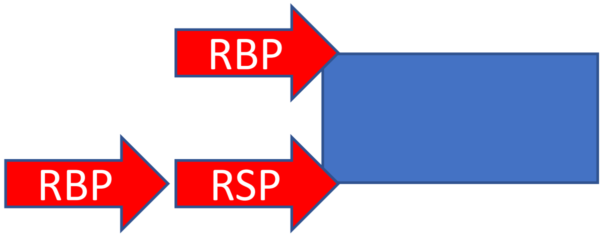
Allocation and deallocation of stack frames require changing **%rbp and %rsp**

Creating a new stack frame for a function and exiting

Create (enter) the new stack frame

```
push %rbp           # push location of base pointer to stack
mov  %rsp, %rbp     # copies the value of the stack pointer
                    # %rsp to the base pointer %rbp->%rsb and %rsp
                    # now both point to the top of the stack
```

Do function here…



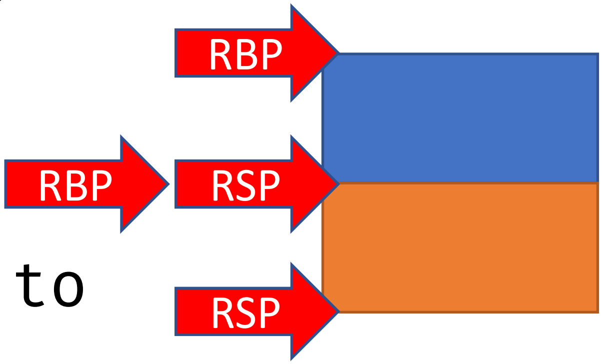
When function is done, remove (leave) stack frame

```
mov %rbp, %rsp      # sets %rsp to %rbp
pop %rbp            # pops the top of the stack into %rbp,
                    # where we stored the previous value
                    # from the push
```

enter and leave

`# enter` **creates a stack frame**

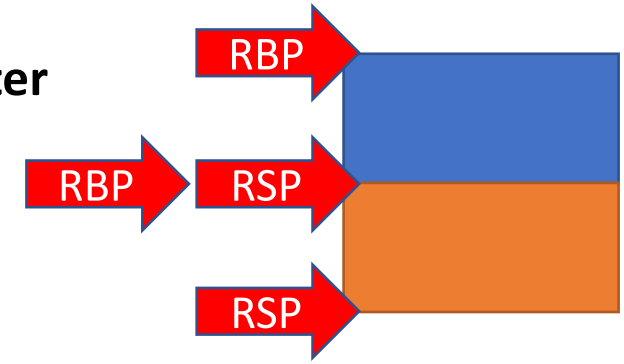
```
enter $0, $0    # is equivalent to
                # push %rbp
                # mov %rsp, %rbp
```



 # and can allocate space in the stack

```
enter $24, $0   # is equivalent to
                # push %rbp
                # mov %rsp, %rbp
                # sub $24, %rsp

# the second arg indicates nesting level
```

enter and **leave**



`# leave` **exits a stack frame: does the inverse of enter**

```
leave               # is equivalent to
                    # mov %rbp, %rsp
                    # pop %rbp
```

Recall,

```
mov %rbp, %rsp # sets %rsp to %rbp

pop %rbp            # pops the top of the stack to %rbp,
                    # where we stored the previous
                    # value from enter
```

stack_exam.c example

```
int bar(int a, int b) {
  int r = rand();
  return a + b - r;
}

int foo(int a) {
  int x, y;
  x = a * 2;
  y = a - 7;
  return bar(x, y);
}

int main(void) {
  …
  foo(12);
  …
}
```

Note that generated assembly code can vary depending on the compiler

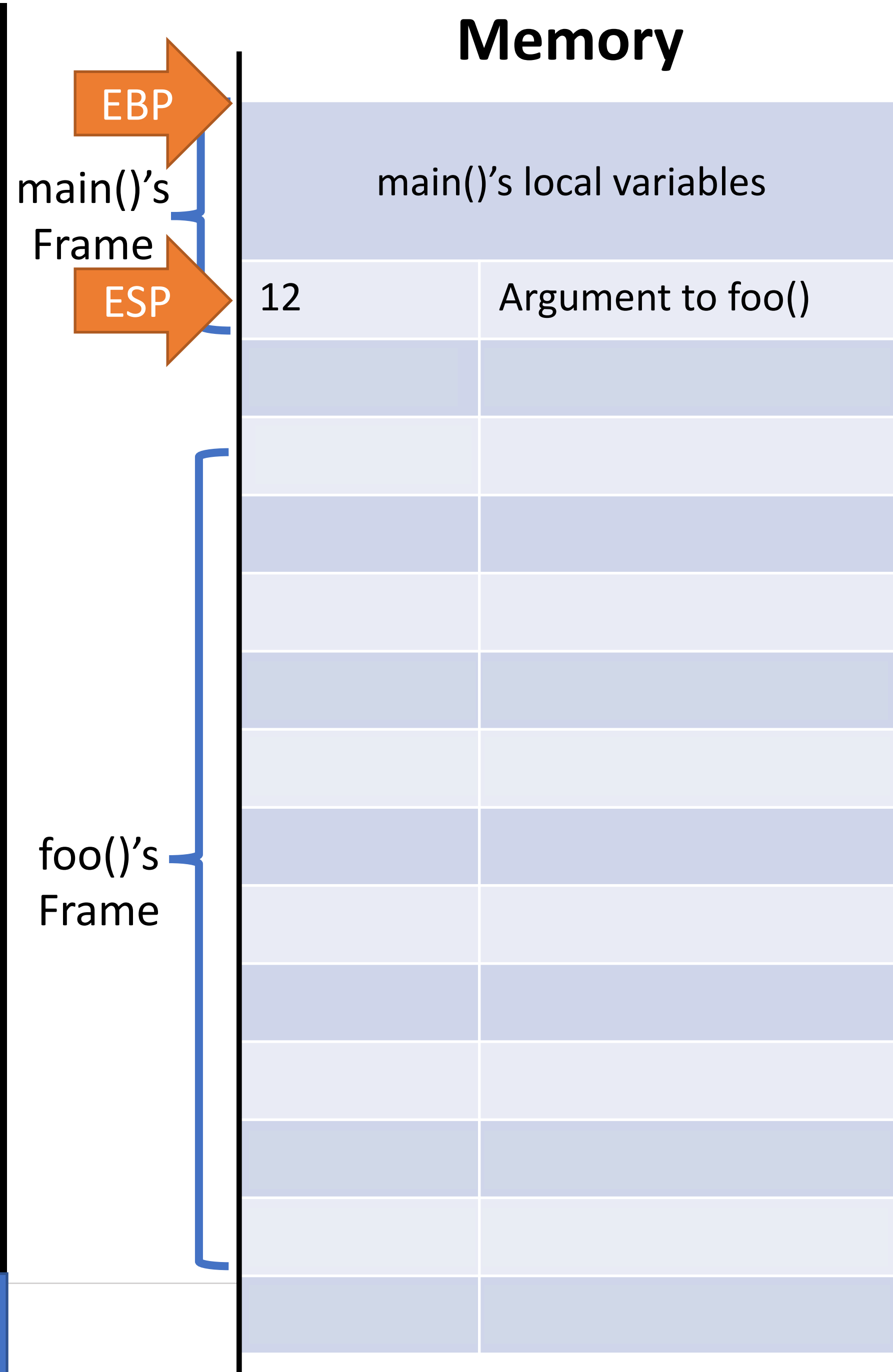
The example in the following slides

- are based on 32-bit architecture,
- use push and mov to create a stack frame, (One can use "enter" instead)
- pass function arguments only through the stack (One may use %rdi, %rsi, %rdx, %rcx, … instead)

The stack is usually used to pass the function arguments when you run out of registers or write recursive functions

```
$ gcc -g -fno-stack-protector -m32 -o stack_exam stack_exam.c
$ objdump --disassemble ./stack_exam
…
804842a:   e8 c0 ff ff ff       call   80483ef <foo>
804842f:   b8 00 00 00 00       mov 0x0, eax
…
080483ef <foo>:
80483ef:   55                   push   ebp
80483f0:   89 e5                mov    esp, ebp
80483f2:   83 ec 28             sub    0x28, esp
80483f5:   8b 45 08             mov    [ebp+0x8], eax
80483f8:   01 c0                add    eax, eax
80483fa:   89 45 f4             mov    eax, [ebp-0xc]
80483fd:   8b 45 08             mov    [ebp+0x8], eax
8048400:   83 e8 07             sub    0x7, eax
8048403:   89 45 f0             mov    eax, [ebp-0x10]
8048406:   8b 45 f0             mov    [ebp-0x10], eax
8048409:   89 44 24 04          mov    eax, [esp+0x4]
804840d:   8b 45 f4             mov    [ebp-0xc], eax
8048410:   89 04 24             mov    eax, [esp]
8048413:   e8 bc ff ff ff       call   80483d4 <bar>
8048418:   c9                   leave
8048419:   c3                   ret
```

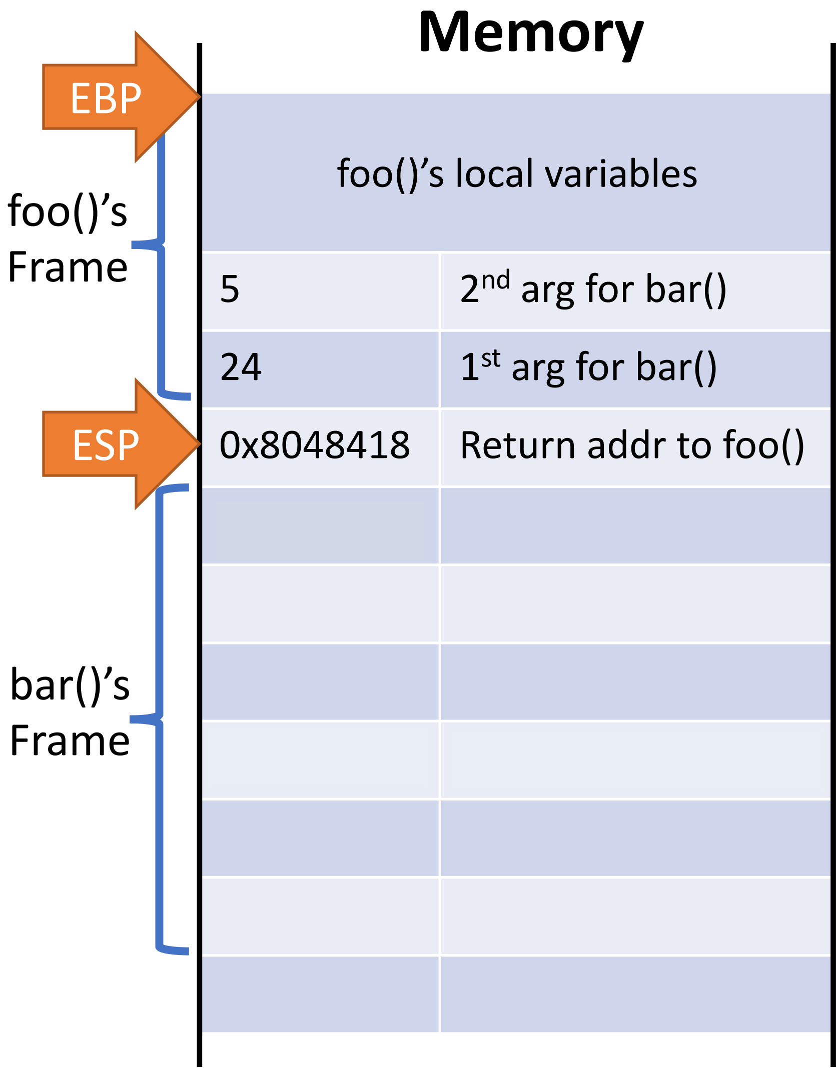
EIP



Note that this is a different assembly syntax from what we use

```
...
080483d4 <bar>:
80483d4:   55              push   ebp
80483d5:   89 e5           mov    esp, ebp
80483d7:   83 ec 18        sub    0x18, esp
80483da:   e8 31 ff ff ff  call   8048310 <rand@plt>
80483df:   89 45 f4        mov    eax, [ebp-0xc]
80483e2:   8b 45 0c        mov    [ebp+0xc], eax,
80483e5:   8b 55 08        mov    [ebp+0x8], edx
80483e8:   01 d0           add    edx, eax
80483ea:   2b 45 f4        sub    [ebp-0xc], eax
80483ed:   c9              leave
80483ee:   c3              ret
...
```

Note that this is a different assembly syntax from what we use



- leave → mov ebp, esp; pop ebp;
- Return value is placed in EAX

A "Design Recipe for Assembly"

1. Signature (C-ish)
2. Pseudocode (ditto)
3. Variable mappings (registers, stack offsets)
4. Skeleton
5. Fill in the blanks

I strongly recommend you to read

Nat Tuck's Assembly Design Recipe in the reading list

1. Signature

- What are our arguments?
- What will we return?

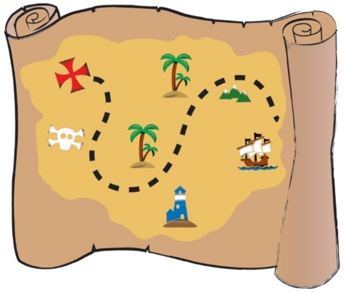
```
# long min(long a, long b)
min:
    ...

# long factorial(long x)
factorial:
    ...
```

2. Pseudocode

- How do we compute the function?
- Thinking in directly in assembly is *hard*
- Translating pseudocode, on the other hand, is quite straightforward
- C works pretty well

```
# long factorial(long x)
factorial:
    # long res = 1;
    # while (x > 1) {
    #   res = res * x;
    #   x--;
    # }
    # return res;
```

3. Variable Mappings



- Need to decide where we store temporary values
- Arguments are given: `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`, then the stack
- Do we keep variables in registers?
  - Callee-save? `%r12`, `%r13`, `%r14`, `%r15`, `%rbx`
  - Caller-save? `%r10`, `%r11` + argument registers
- Do we use the stack?

Callee must restore the original value before exiting

Callee can freely modify the register

```
# long factorial(long x)
factorial:
    # x -> %r12
    # res -> %rax
```

Northeastern University

4. Function Skeleton

```
label:
    # Prologue:
    #   Set up stack frame.
    # Body:
    #   Just say "TODO"
    # Epilogue:
    #   Clean up stack frame.
```

Prologue:	Epilogue:
● `push` callee-saves ● `enter` – allocate stack space ○ stack alignment!	● `leave` - deallocate stack space ● Restore (`pop`) any pushed registers ● `ret` - return to call site

4. Function Skeleton

```
min:
    # Prologue:
    push %r12       # Save callee-save regs.
    push %r13
    enter $24, $0   # Allocate / align stack
    # Body:
                    # Just say "TODO"
    # Epilogue:
    leave           # Clean up stack frame.
    pop %r13        # Restore saved regs.
    pop %r12
    ret             # Return to call site
```

5. Complete the Body

- Translate your pseudocode into assembly - line by line
- Apply variable mappings

Variables, Temporaries, Assignment

- Each C variable maps to a register or a stack location (by using `enter`)
- Temporary results go into registers
- Registers can be shared / reused - keep track carefully

```c
long x = 5;
long y = x * 2 + 1;
```

With:

x in %r10

y in %rbx

Temporary for x * 2 is %rdx

?

Variables, Temporaries, Assignment

- Each C variable maps to a register or a stack location (by using `enter`)
- Temporary results go into registers
- Registers can be shared / reused - keep track carefully

```
long x = 5;
long y = x * 2 + 1;
```

With:
x in %r10
y in %rbx
Temporary for x * 2 is %rdx

```
# long x = 5;
mov $5, %r10

# long y = x * 2 + 1;
mov %r10, %rdx
imulq $2, %rdx
add $1, %rdx
mov %rdx, %rbx
```

If statements 1

```
// Case 1
if (x < y) {
  y = 7;
}
```

Variables:

- x is `-8(%rbp)`
- y is `-16(%rbp)` or, temporarily, `%r10`

If statements 1

```
// Case 1
if (x < y) {
  y = 7;
}
```

Variables:

- x is `-8(%rbp)`
- y is `-16(%rbp)` or, temporarily, `%r10`

```
# if (x < y)
  # cmp can only take one indirect arg
  mov -16(%rbp), %r10
  cmp %r10, -8(%rbp)
  # cmp order backwards from C
  # condition reversed, skip block
  # _unless_ cond
  # jge -> if (-8(%rbp) >= %r10)
  # then jump to else1
  jge else1:

  # y = 7
  movq $7, -16(%rbp)
  # need suffix to set size of "7"
else1:
  ...
```

If statements 2

```c
// Case 2
if (x < y) {
  y = 7;
}
else {
  y = 9;
}
```

Variables:

- x is `-8(%rbp)`
- y is `-16(%rbp)` or, temporarily, `%r10`

If statements 2

```
// Case 2
if (x < y) {
  y = 7;
}
else {
  y = 9;
}
```

Variables:

- x is `-8(%rbp)`
- y is `-16(%rbp)` or, temporarily, `%r10`

```
# if (x < y)
  mov -16(%rbp), %r10
  cmp %r10, -8(%rbp)
  jge else1:
  # then {
  # y = 7
  movq $7, -16(%rbp)
  # need suffix to set size of "7"

  jmp done1           # skip else

  # } else {
else1:
  # y = 9
  movq $9, -16(%rbp)

  # }
done1:
  ...
```

Do-while loops

```
do {
  x = x + 1;
} while (x < 10);
```

Variables:

- x is `-8(%rbp)`

?

Do-while loops

```
do {
  x = x + 1;
} while (x < 10);
```

```
loop:
  add $1, -8(%rbp)

  cmp $10, -8(%rbp)
  # reversed for cmp arg order

  jl loop
  # sense not reversed

  # ...
```

Variables:

- x is `-8(%rbp)`

While loops

```
while (x < 10) {
  x = x + 1;
}
```

?

Variables:

- x is `-8(%rbp)`

While loops

```c
while (x < 10) {
  x = x + 1;
}
```

```
loop_test:
  cmp $10, -8(%rbp) # reversed for cmp
  jge loop_done # jump out if greater than

  add $1, -8(%rbp)
  jmp loop_test

loop_done:
  ...
```

Variables:

- x is `-8(%rbp)`

Recursive Functions and the Stack

How to Use Recursion?

- Let’s say we want to write a factorial function.

How to program Recursion?

- Let's say we want to write a recursive factorial function.

- ...something like:

```
long fact(long n) {
  if (n <= 1) {
    return 1;
  }

  return n * fact(n - 1);
}
```

Factorial

In general: we need to use the stack to hold on to data when doing recursive calls.

Follow Design Recipe: Signature

- What are arguments?
- What is returned?

```
#long fact(long )
fact:
 ...
```

Follow Design Recipe: Pseudocode

- The C looks good…

```c
long fact(long n) {
  if (n <= 1) {
    return 1;
  }

  return n * fact(n - 1);
}
```

Follow Design Recipe: Variable Mappings

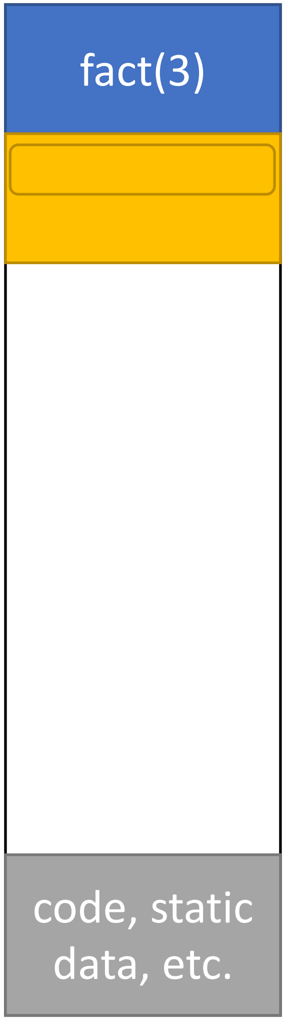
- Storing temp variable on the stack
- Returning result in %rax

```
#long fact(long n)
fact:
# n   -> (-8)%rbp
# res -> %rax
 ...
```

Follow Design Recipe: Function Skeleton

```
long fact(long n) {
  if (n <= 1) {
    return 1;
  }

  return n * fact(n - 1);
}
```

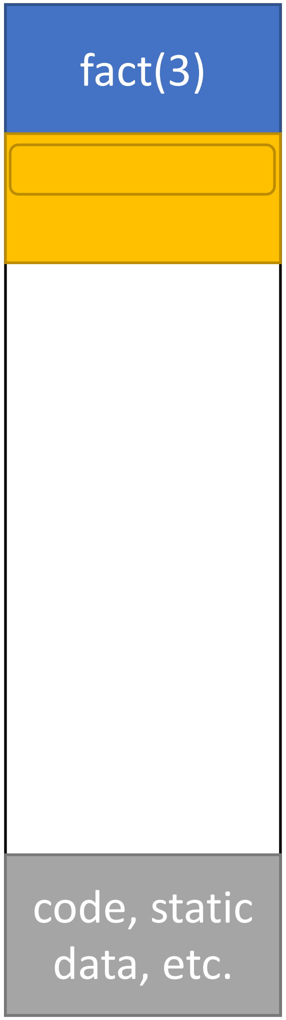
```
#long fact(long n)
fact:
# n   -> (-8)%rbp
# res -> %rax
    # Prologue:
    enter $16, $0  # Allocate / align stack
    # Body:
                   # Just say "TODO"
    # Epilogue:
    leave          # Clean up stack frame.
    ret            # Return to call site
```



Follow Design Recipe: Complete the Body

```
#long fact(long n)
fact:
# n   -> (-8)%rbp
# res -> %rax
    # Prologue:
    enter $16, $0  # Allocate / align stack
    # Body:
    movq    %rdi, -8(%rbp) # copy argument to stack
    cmpq    $1, -8(%rbp)   # if (n > 1)
    jg      .decrement     # goto fact(n-1)
    movq    $1, %rax       # else return 1
    jmp     .end
.decrement
   . . .
    # Epilogue:
.end
    leave          # Clean up stack frame.
    ret            # Return to call site
```

```
long fact(long n) {
  if (n <= 1) {
    return 1;
  }

  return n * fact(n - 1);
}
```



Follow Design Recipe: Complete the Body

```
#long fact(long n)
fact:
# n   -> (-8)%rsp
# res -> %rax
    # Prologue:
    enter $16, $0  # Allocate / align stack
    # Body:
    movq    %rdi, -8(%rbp) # copy 1st argument to stack
    cmpq    $1, -8(%rbp)   # if (n > 1)
    jg      .decrement     #   goto fact(n-1)
    movq    $1, %rax       # else return 1
    jmp     .end
.decrement
    movq    -8(%rbp), %rax # copy argument off stack to %rax
    subq    $1, %rax       # n-1
    movq    %rax, %rdi     # copy n-1 to 1st argument register %rdi
    call    fact           # call fact(n-1)
    imulq   -8(%rbp), %rax # n * fact(n-1)
    # Epilogue:
.end
    leave          # Clean up stack frame.
    ret            # Return to call site
```

```
long fact(long n) {
  if (n <= 1) {
    return 1;
  }

  return n * fact(n - 1);
}
```