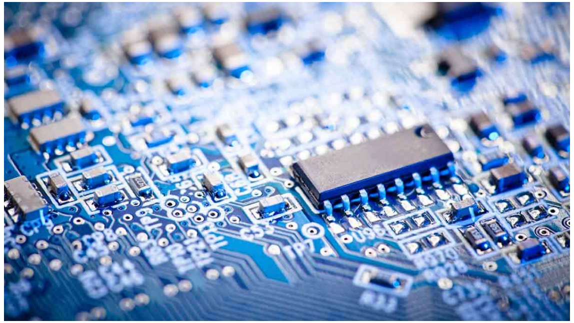
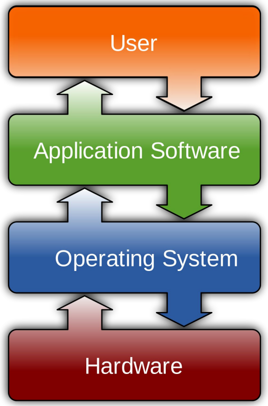
Please do not redistribute these slides without prior written permission



CS 3650
Computer Systems

Alden Jackson | Ferdinand Vesely



Intro	Virtualization		Concurrency	Persistence	Appendices
Preface	3 *Dialogue*	12 *Dialogue*	25 *Dialogue*	35 *Dialogue*	*Dialogue*
TOC	4 Processes	13 Address Spaces	26 Concurrency and Threads code	36 I/O Devices	Virtual Machines
1 *Dialogue*	5 Process API code	14 Memory API	27 Thread API	37 Hard Disk Drives	*Dialogue*
2 Introduction code	6 Direct Execution	15 Address Translation	28 Locks	38 Redundant Disk Arrays (RAID)	Monitors
	7 CPU Scheduling	16 Segmentation	29 Locked Data Structures	39 Files and Directories	*Dialogue*
	8 Multi-level Feedback	17 Free Space Management	30 Condition Variables	40 File System Implementation	Lab Tutorial
	9 Lottery Scheduling code	18 Introduction to Paging	31 Semaphores	41 Fast File System (FFS)	Systems Labs
	10 Multi-CPU Scheduling	19 Translation Lookaside Buffers	32 Concurrency Bugs	42 FSCK and Journaling	xv6 Labs
	11 *Summary*	20 Advanced Page Tables	33 Event-based Concurrency	43 Log-structured File System (LFS)	
		21 Swapping: Mechanisms	34 *Summary*	44 Flash-based SSDs	
		22 Swapping: Policies		45 Data Integrity and Protection	
		23 Case Study: VAX/VMS		46 *Summary*	
		24 *Summary*		47 *Dialogue*	
				48 Distributed Systems	
				49 Network File System (NFS)	
				50 Andrew File System (AFS)	
				51 *Summary*	

Lecture 3 - Assembly, contd.

Alden Jackson

Procedures/Functions

Procedure Mechanisms

Several things happen when calling a procedure (i.e. function or method)

1. Pass control
   - Start executing from start of procedure
   - Return back to where we called from
2. Pass data
   - Procedure arguments and return value are passed
3. Memory management
   - Memory allocated in the procedure, and then deallocated on return
4. x86-64 uses the minimum subset required

x86-64 Memory Space

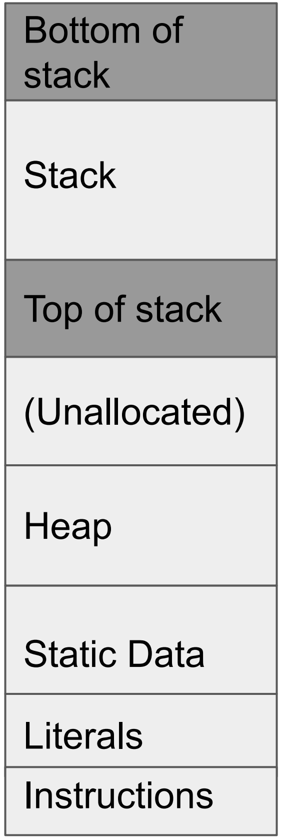
- Our view of a program is a giant byte array
- However, it is segmented into different regions
  - This separation is determined by the Application Binary Interface (ABI)
    - This is something typically chosen by the OS.

- We traverse our byte array as a stack

x86-64 stack

Our Program Memory Space is divided into several segments. Some parts of it are for long lived data (the heap), and the other is for short-lived data (the stack) typically used for functions and local variables.

Program Memory

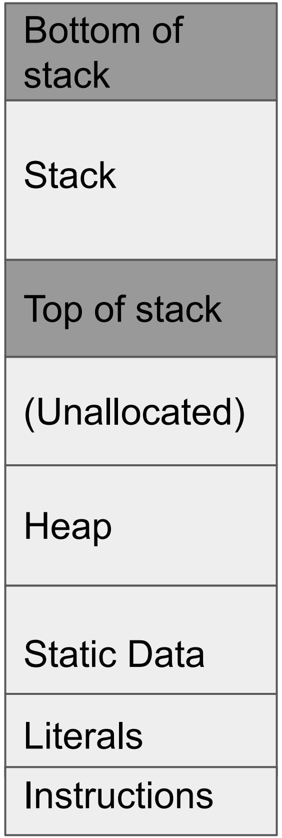


x86-64 stack

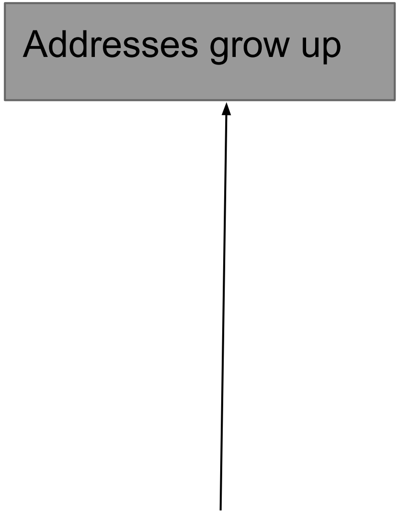
With a Stack data structure, we can perform two main operations

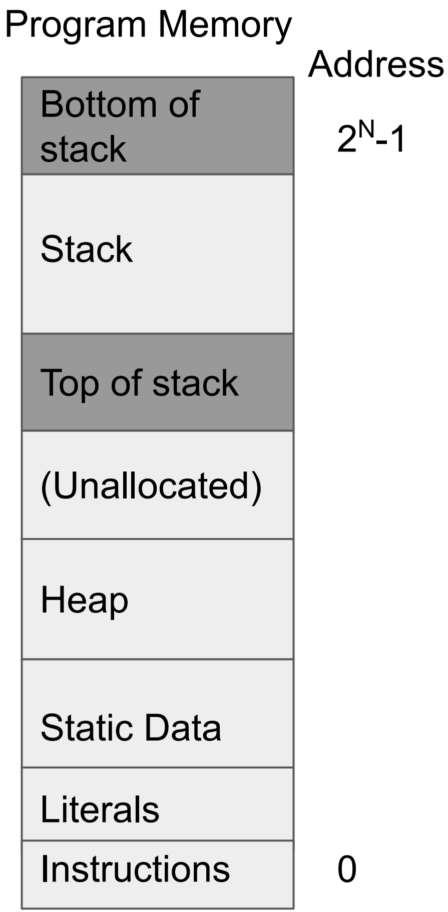
1. push data onto the stack (add information)
   a. Our stack grows
2. pop data off of the stack (remove information)
   a. Our stack shrinks

Program Memory



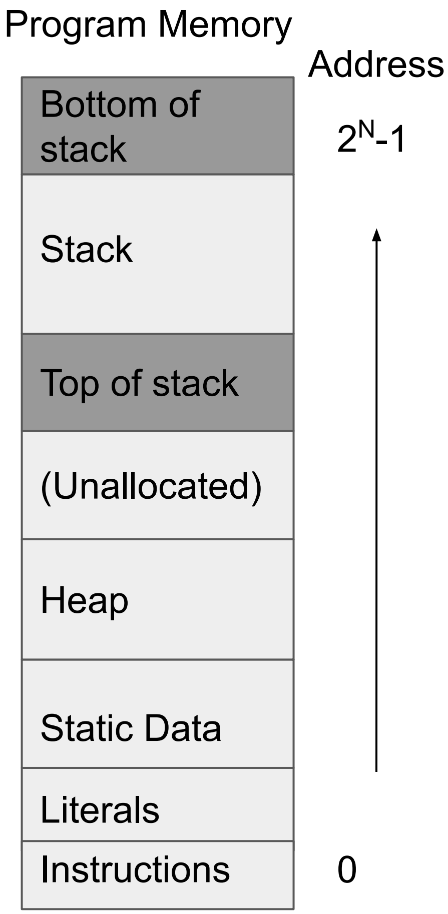
x86-64 stack


Addresses grow up


Program Memory
Address
Bottom of stack
2^N-1
Stack
Top of stack
(Unallocated)
Heap
Static Data
Literals
Instructions
0

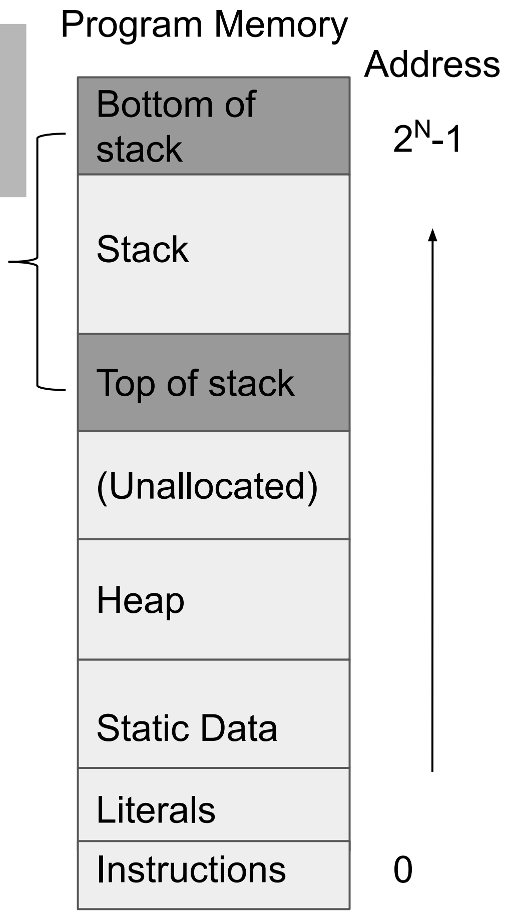
x86-64 stack

Addresses grow up



x86-64 stack

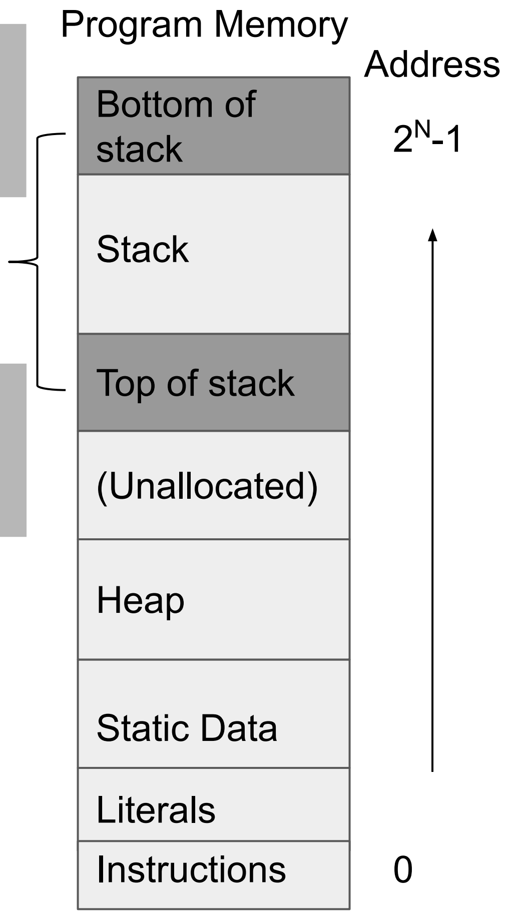
Stack grows down
(But hopefully not into the heap--otherwise error!



x86-64 stack

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

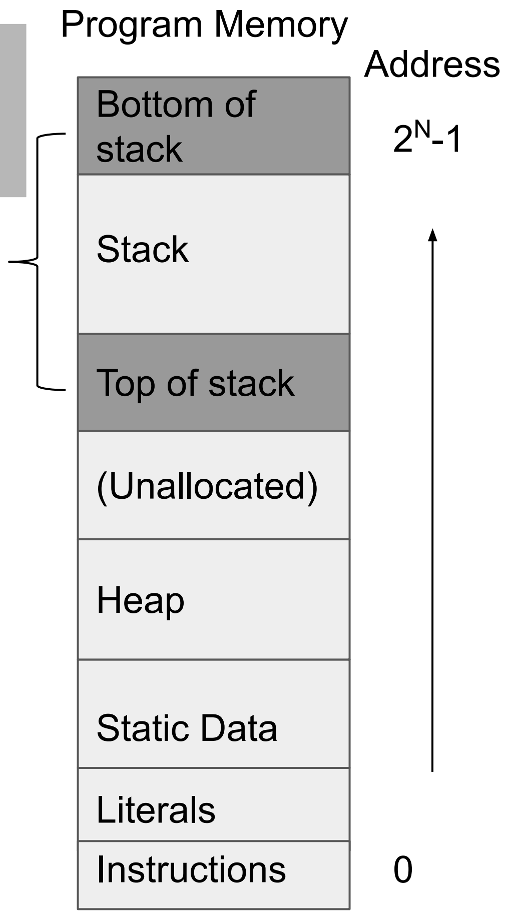
That means the top of our stack is approaching address 0



x86-64 stack

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

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

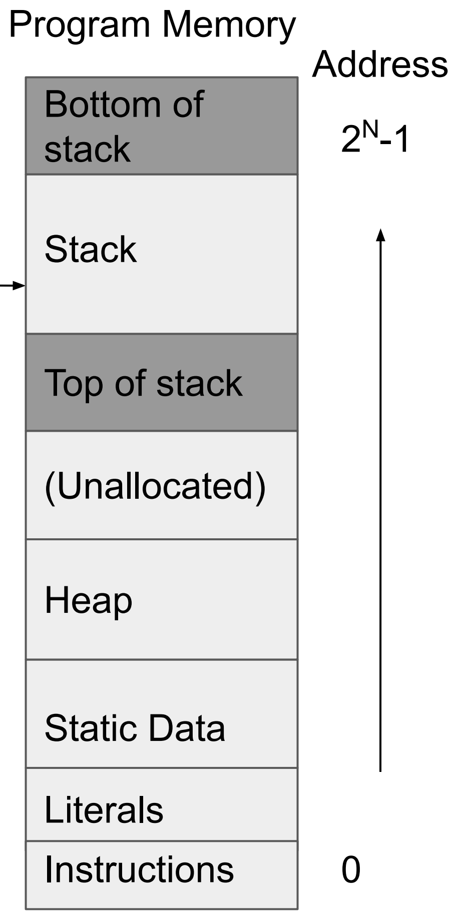


x86-64 stack

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

Stack Pointer: %rsp
Always contains lowest address

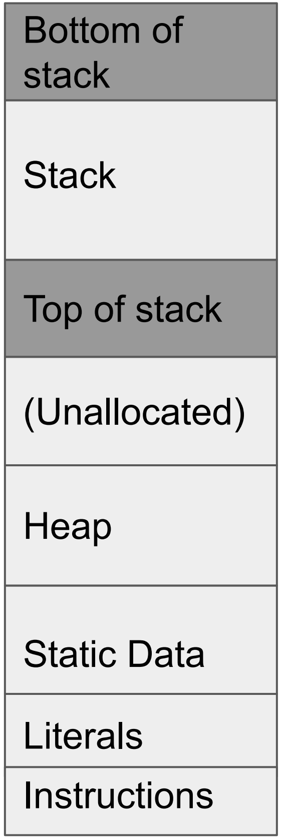
This is the "top" of the stack



Remember these registers?

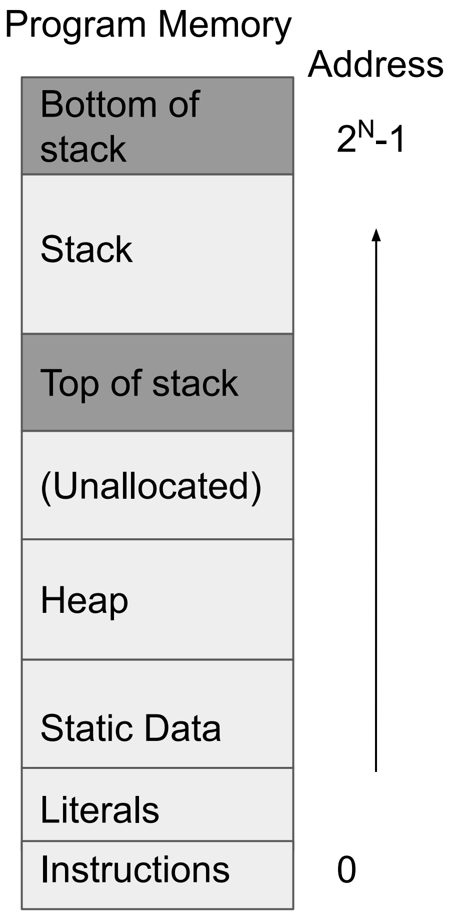
- This can be dependent on the instruction being used
- **%rsp** - keeps track of where the stack is for example
- %rdi - the first argument in a function
- %rsi - The second argument in a function
- %rdx - the third argument of a function
- %rip - the Program Counter
- %r8-%r15 **-** These are the general purpose registers

Program Memory



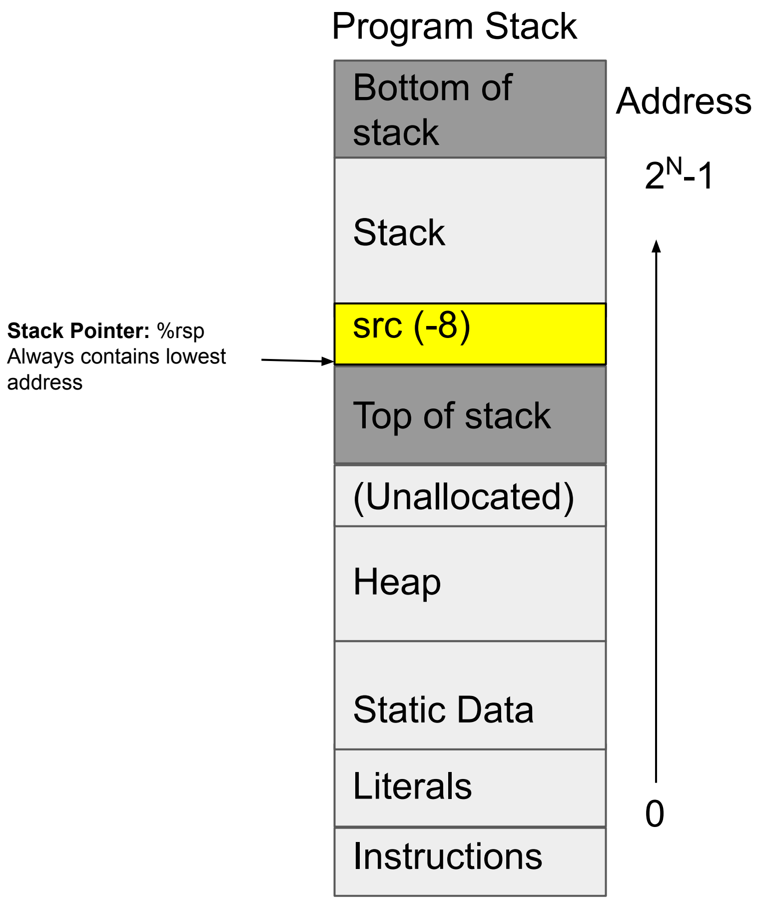
x86-64 stack | PUSHQ Example

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



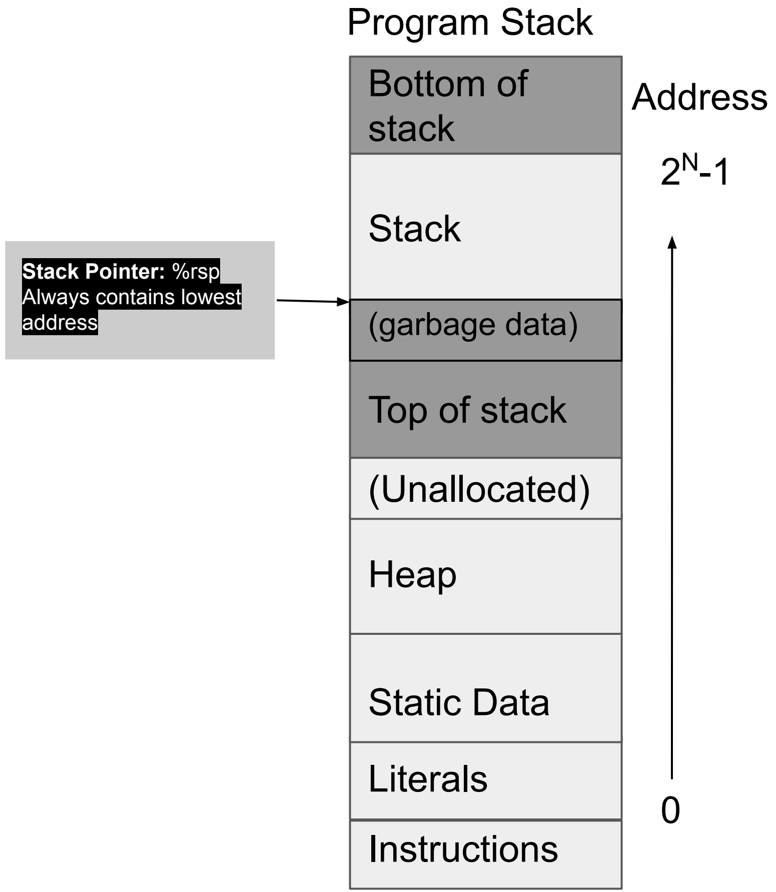
x86-64 stack | PUSHQ Example

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



x86-64 stack | POPQ Example

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



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

(Originally by Nat Tuck)

1. Signature

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

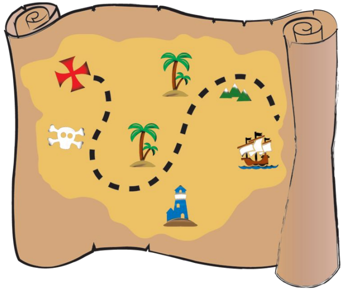
```
# long min(long a, long b)
gcd:
    ...

# 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) {
  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?

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

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 $16, $0  # Allocate / align stack
    # Body:
                  # Just say "TODO"
    # Epilogue:
    leave         # Clean up stack frame.
    pop %r12      # Restore saved regs.
    pop %r13
    ret           # Return to call site
```

5. Complete the Body

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

Translating Pseudocode

- Relatively straightforward
- Each line of C corresponds to one or a few instructions
- When you get stuck, use https://godbolt.org/ for inspiration

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 %r11
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, %rbx
imulq $2, %rbx
add $1, %rbx
mov %rbx, %rdx
```

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) jump to else1
  jge else1:

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

else1:
  ...
```

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 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

```
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)

Memory

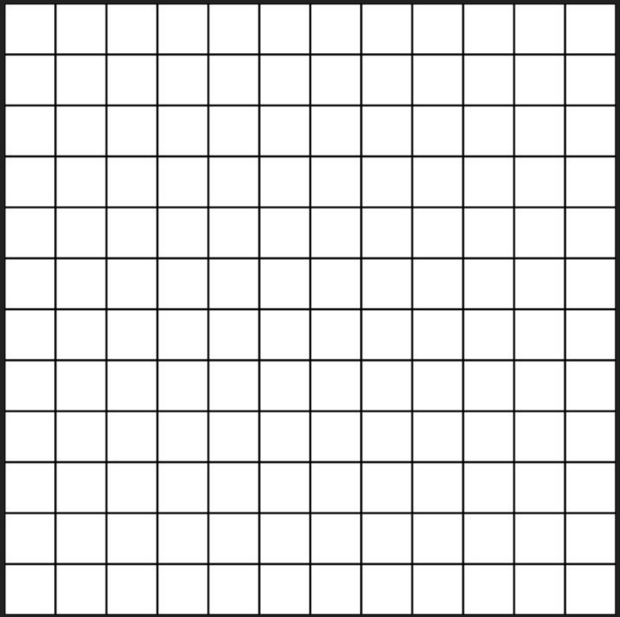
- So far, we’ve been mostly using the processor’s registers to store data
- In lab, we asked you to retrieve a command line argument in assembly
- Today we’ll talk more about addressing and accessing memory

Memory on our machines

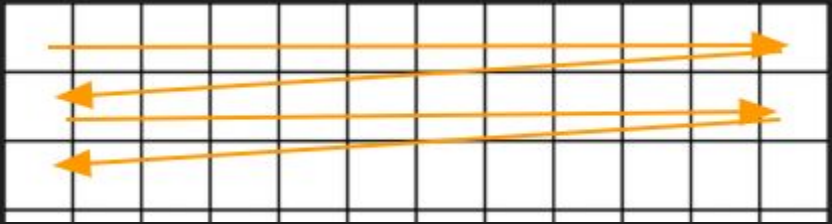
- The memory in our machines stores data so we can recall it later
- This occurs at several different levels
  - Networked drive (or cloud storage)
  - Hard drive
  - Dynamic memory
  - Cache
- For now, we can think of memory as a giant linear array.

Linear array of memory

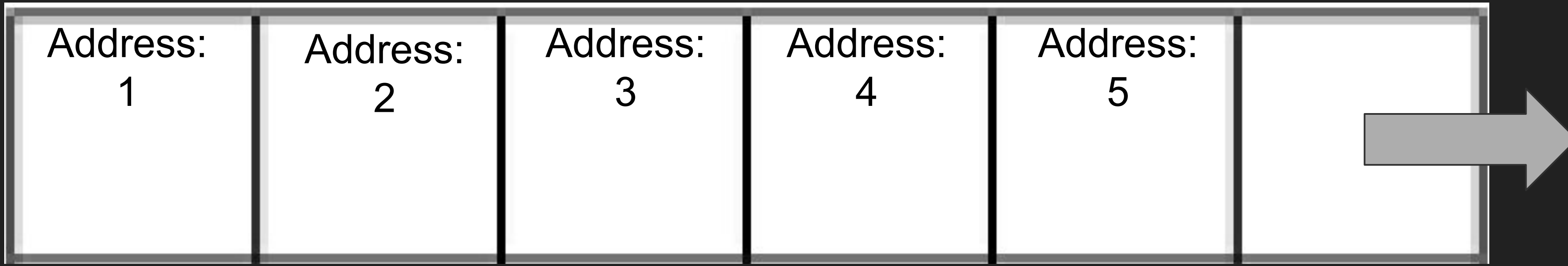
- Each ‘box’ here we will say is 1 byte of memory
  - (1 byte = 8 bits on most systems)
- Depending on the data we store, we will need 1 byte, 2 bytes, 4 bytes, etc. of memory



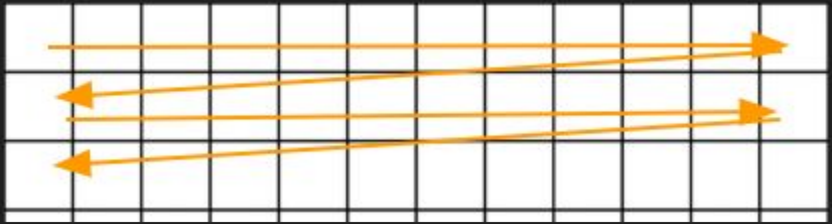
Linear array of memory



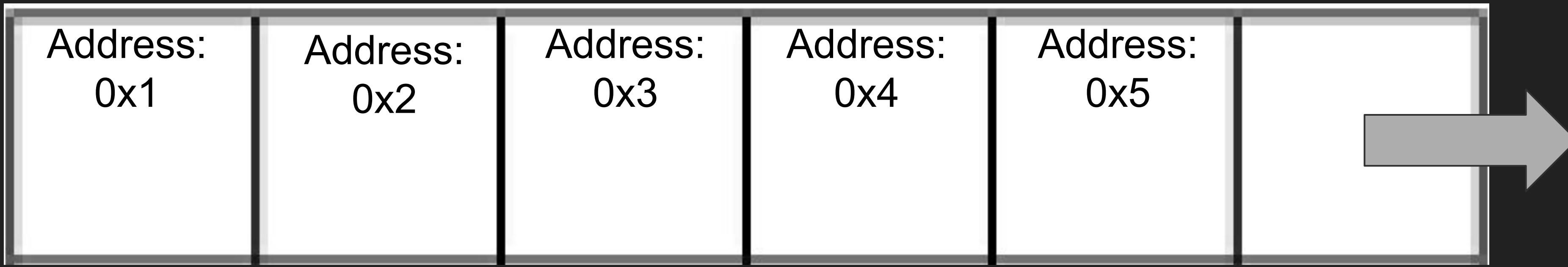
- Visually I have organized memory in a grid, but memory is really a linear array as depicted below.
  - There is 1 address after the other



Linear array of memory



- Visually I have organized memory in a grid, but memory is really a linear array as depicted below.
    - There is 1 address after the other
    - Because these addresses grow large, typically we represent them in hexadecimal (16-base number system)
        - (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
`short int`	**w**	2	0 to 65,535
`int`	**l**	4	0 to 4,294,967,295
`long int`	**q**	8	0 to 18,446,744,073,709,551,615

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)

…

0×41F00	*00*	*01*	*02*	*03*	*04*	*05*	*06*	*07*
0×41F08	*08*	*09*	*0A*	*0B*	*0C*	*0D*	*0E*	*0F*
0×41F10	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*
0×41F18	*18*	*19*	*1A*	*1B*	*1C*	*1D*	*1E*	*1F*
0×41F20	*20*	*21*	*22*	*23*	*24*	*25*	*26*	*27*
0×41F28	*28*	*29*	*2A*	*2B*	*2C*	*2D*	*2E*	*2F*
0×41F30	*30*	*31*	*32*	*33*	*34*	*35*	*36*	*37*
0×41F38	*38*	*39*	*3A*	*3B*	*3C*	*3D*	*3E*	*3F*

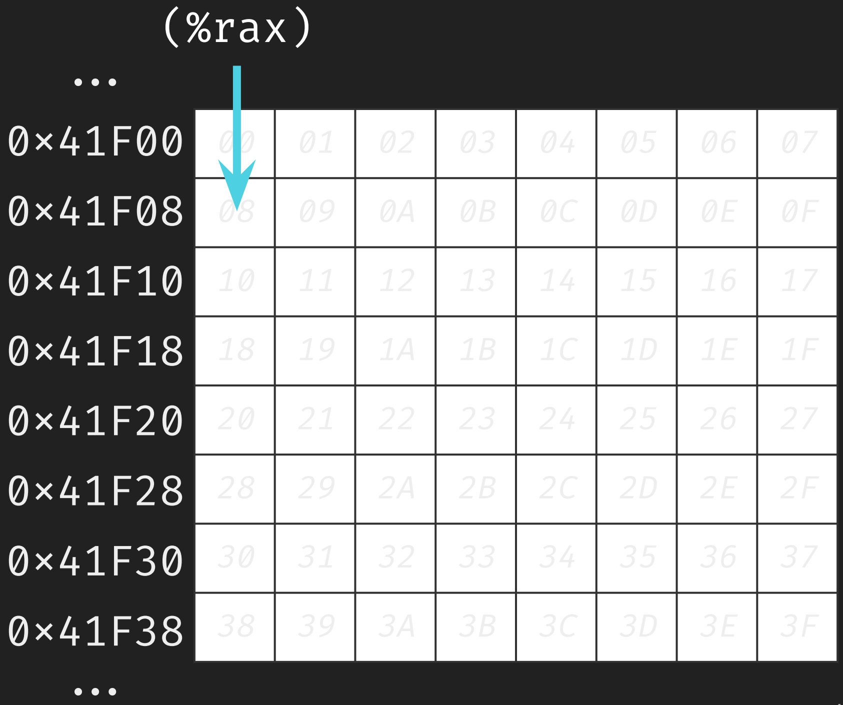
…

Addressing memory

```
mov $0×41F08, %rax
```

We move the address 0x41F08 into rax

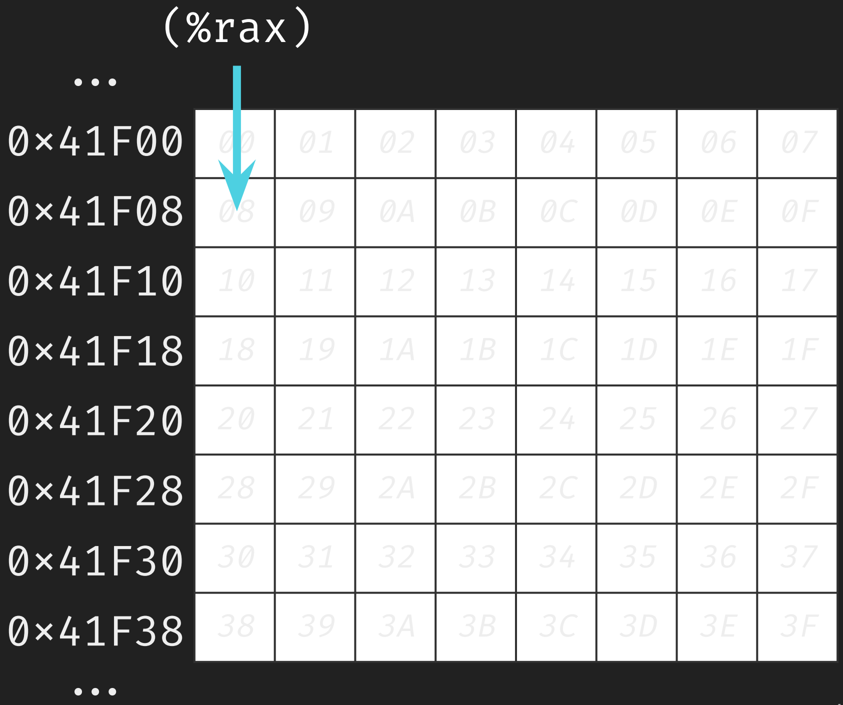
(%rax) now points to the contents of the corresponding chunk of memory



Addressing memory

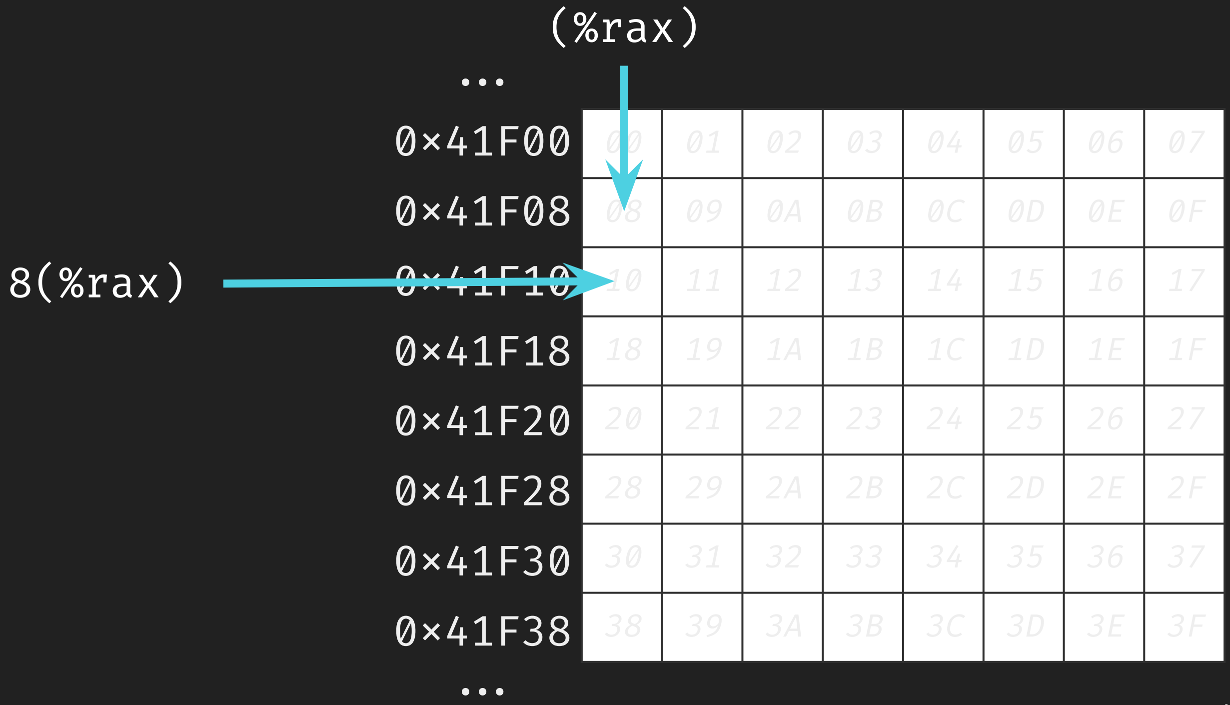
Offset addressing:

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



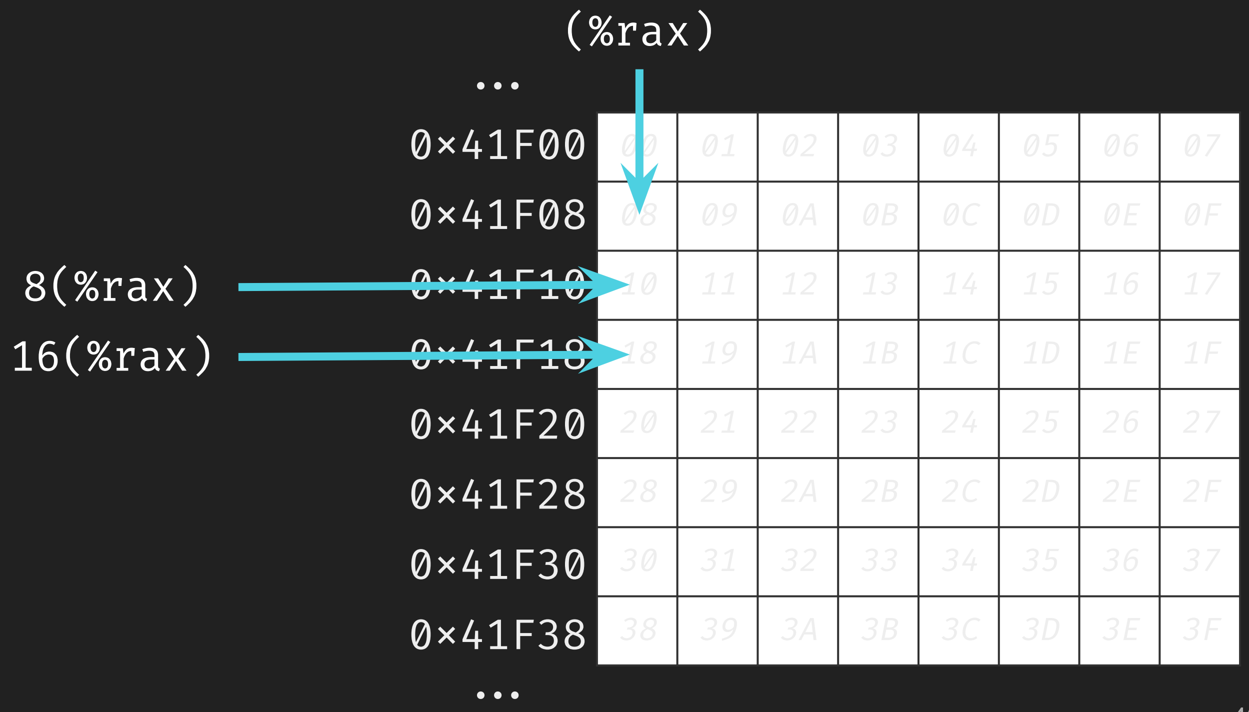
Addressing memory

Offset addressing



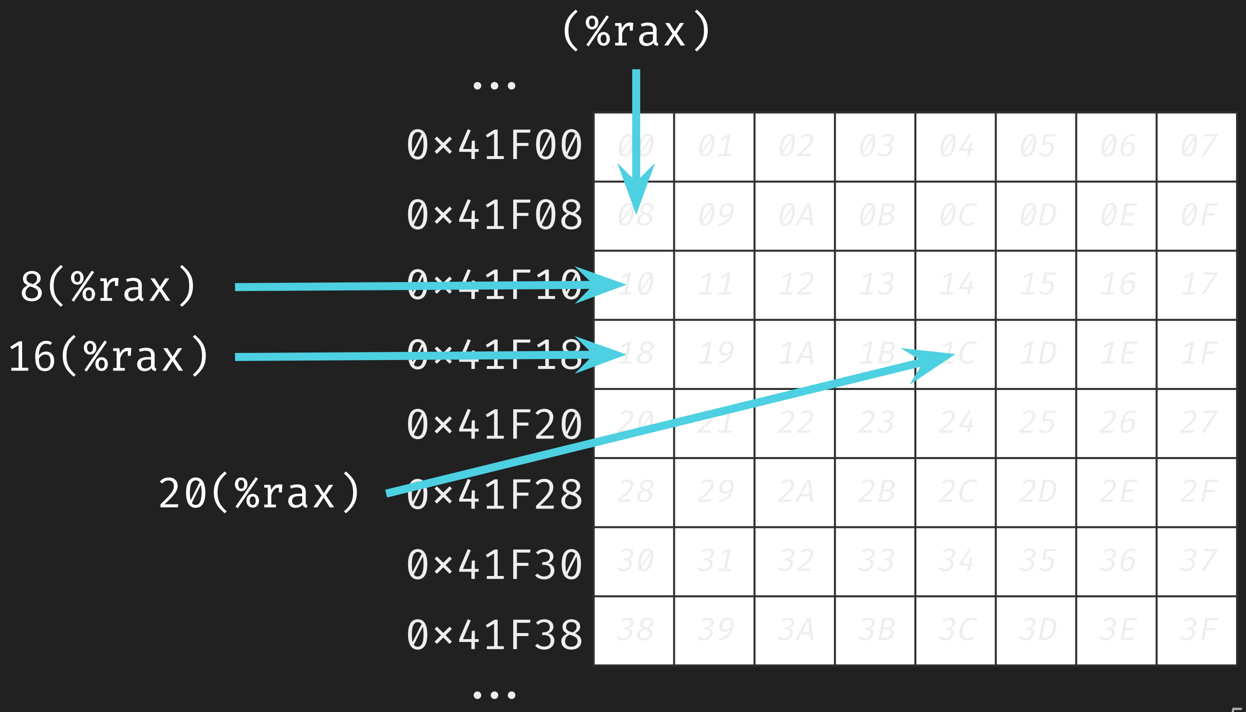
Addressing memory

Offset addressing



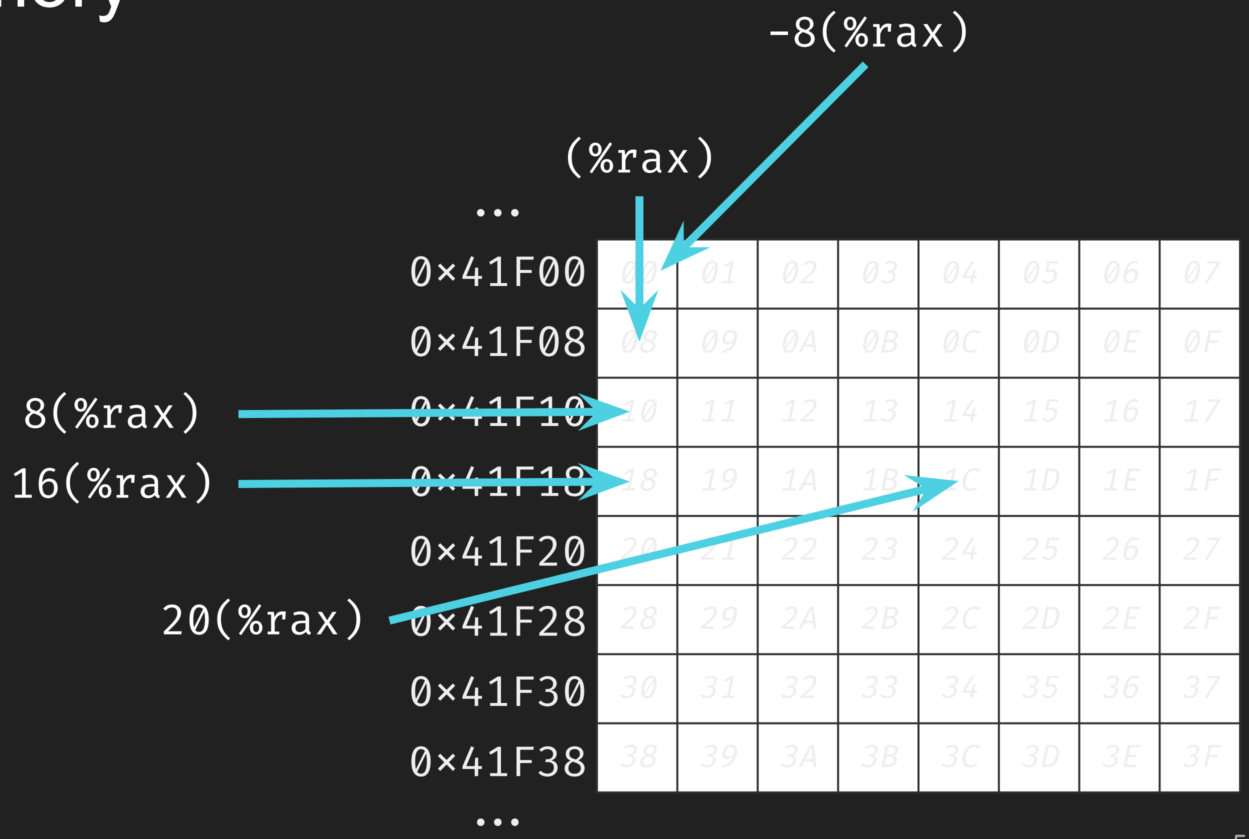
Addressing memory

Offset addressing



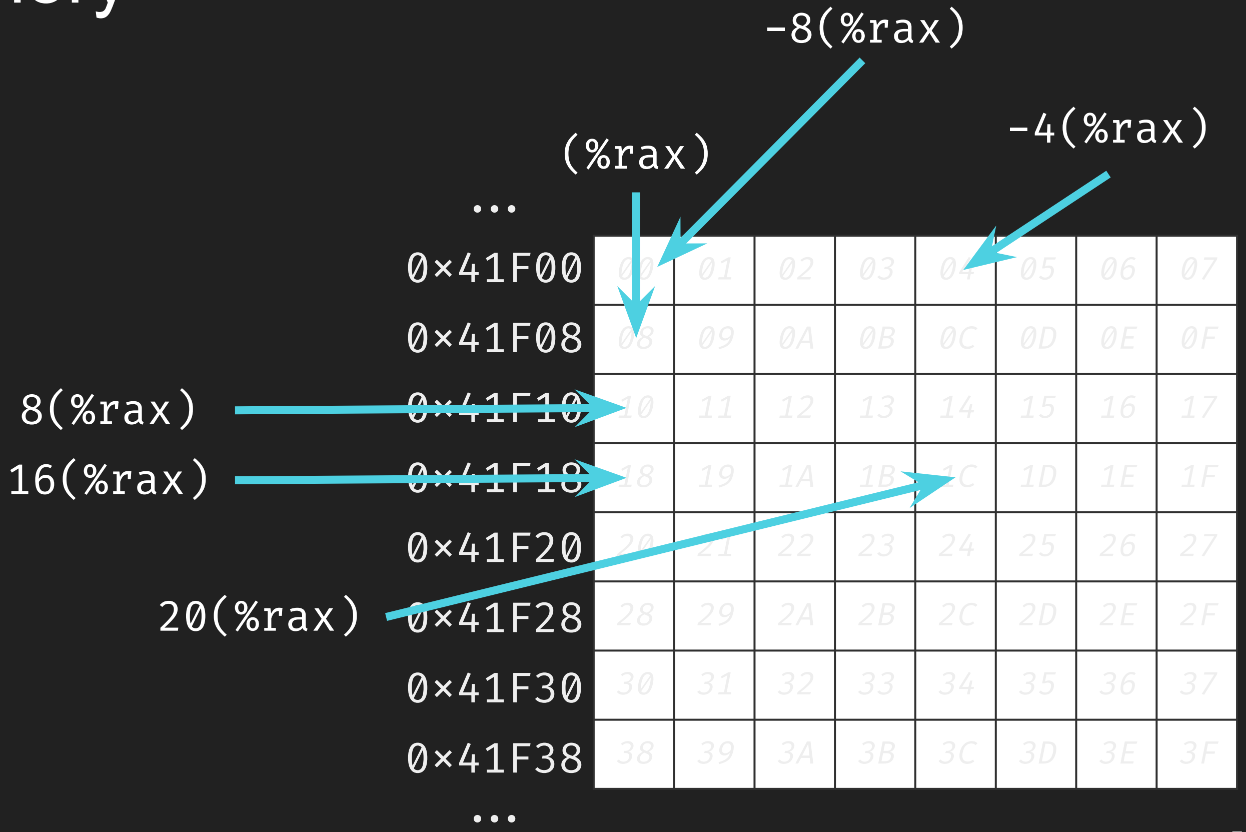
Addressing memory

Offset addressing



Addressing memory

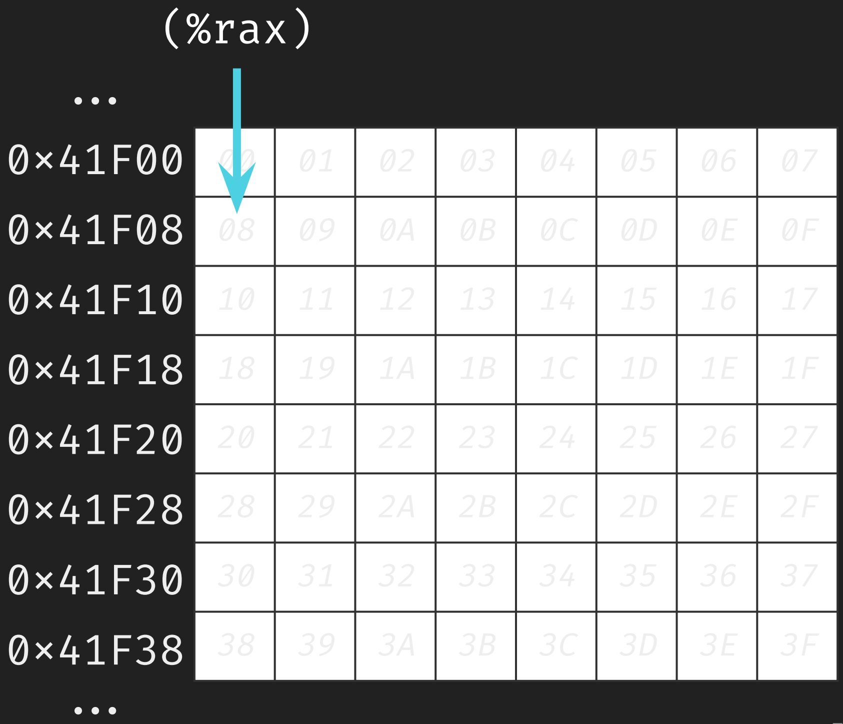
Offset addressing



Addressing memory

```
mov $0×1020304050607080, (%rax)
```

What does this look like in memory?

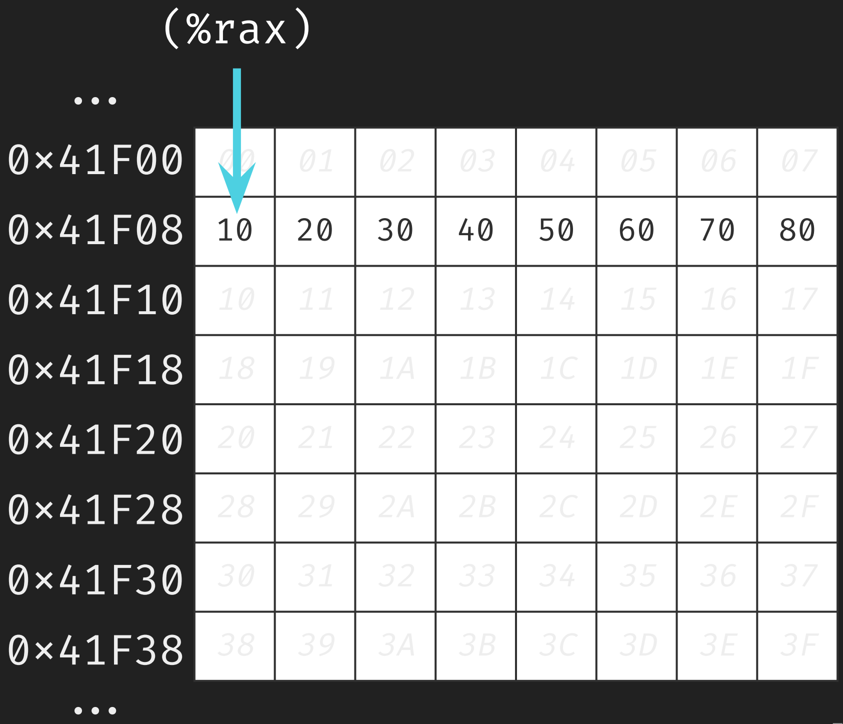


Addressing memory

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

What does this look like in memory?

Like this?

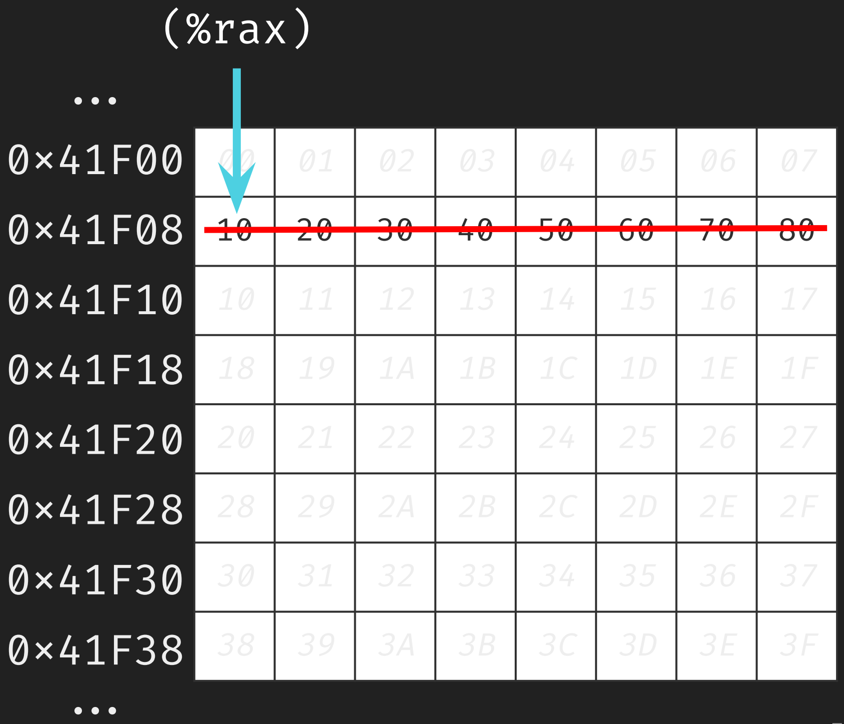


Addressing memory

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

What does this look like in memory?

Like this? **NO**



Addressing memory

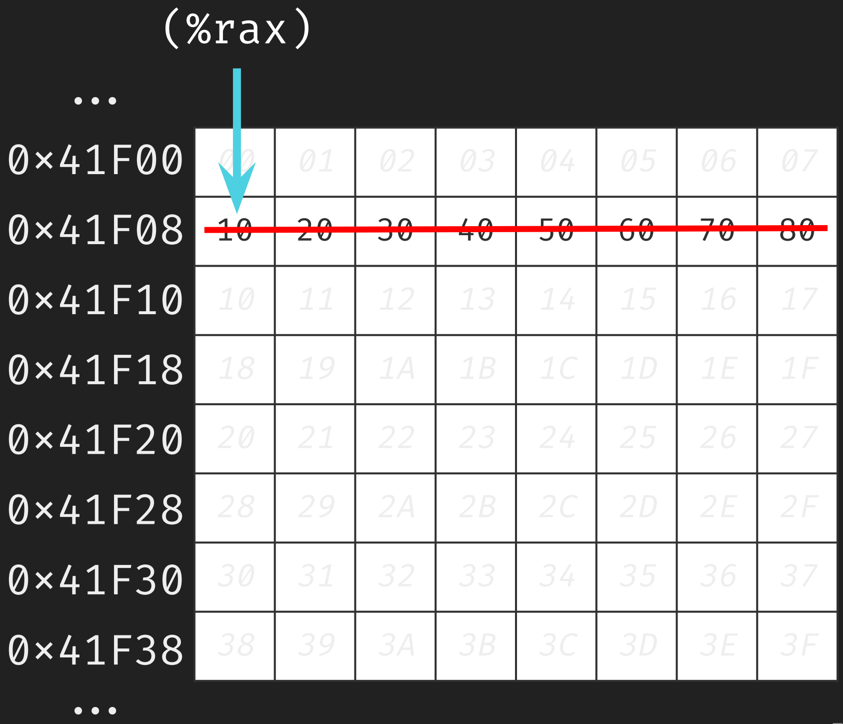
`mov $0×1020304050607080, (%rax)`

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)

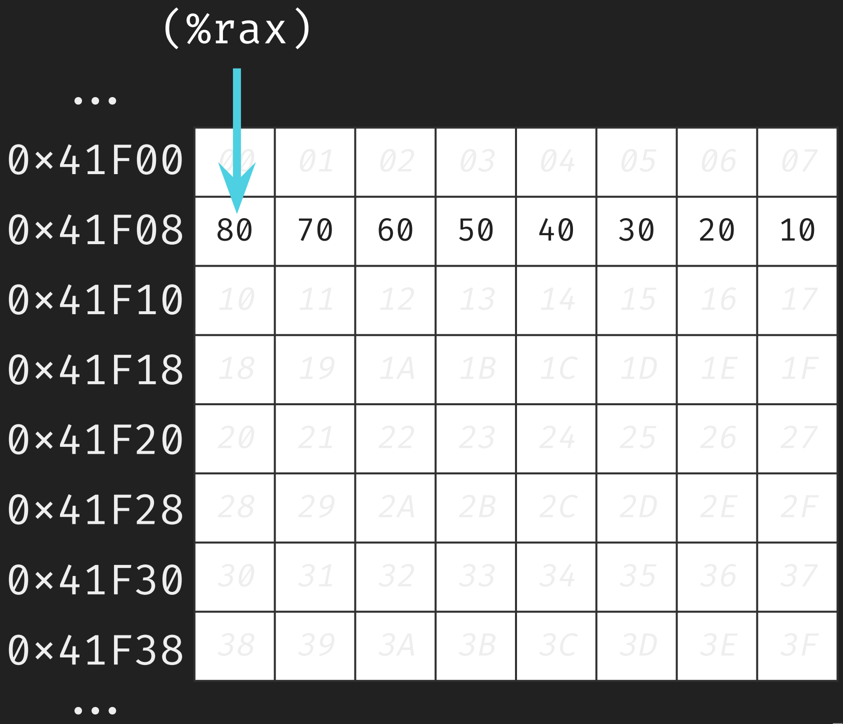


Addressing memory

```
mov $0×1020304050607080, (%rax)
```

What does this look like in memory?

Like this.



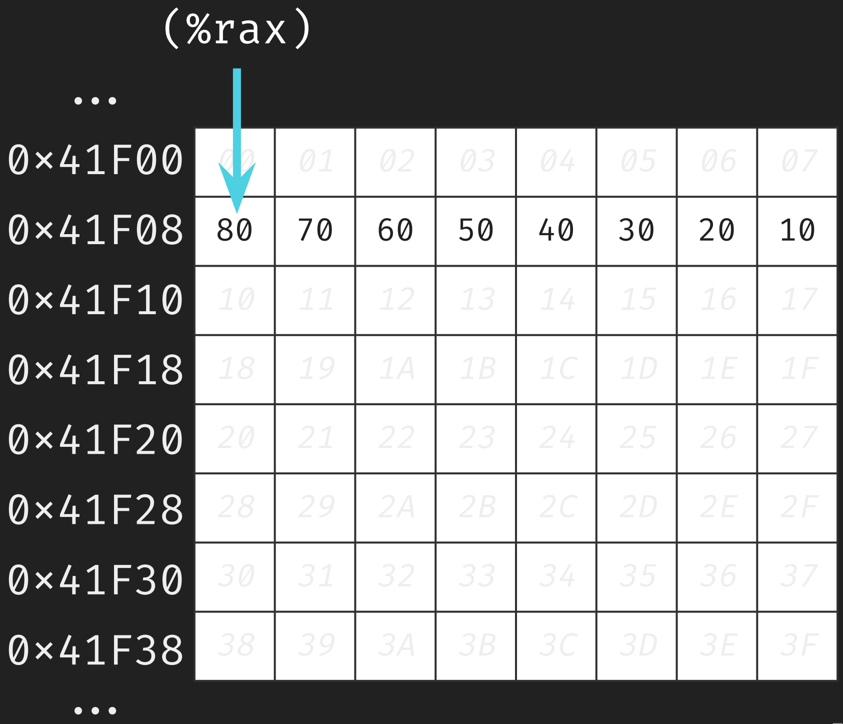
Addressing memory

```
movq (%rax), %r10
```

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

```
movq %rax, %r11
```

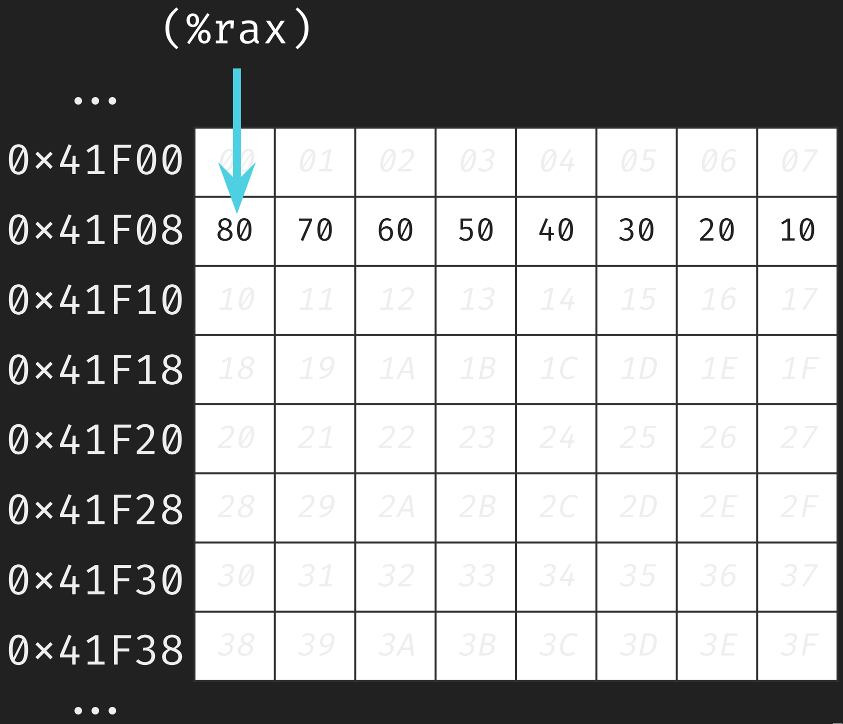
Copies the contents of %rax to %r11. Now (%rax) and (%r11) point to the same location.



Addressing memory

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

What's in %ebx?



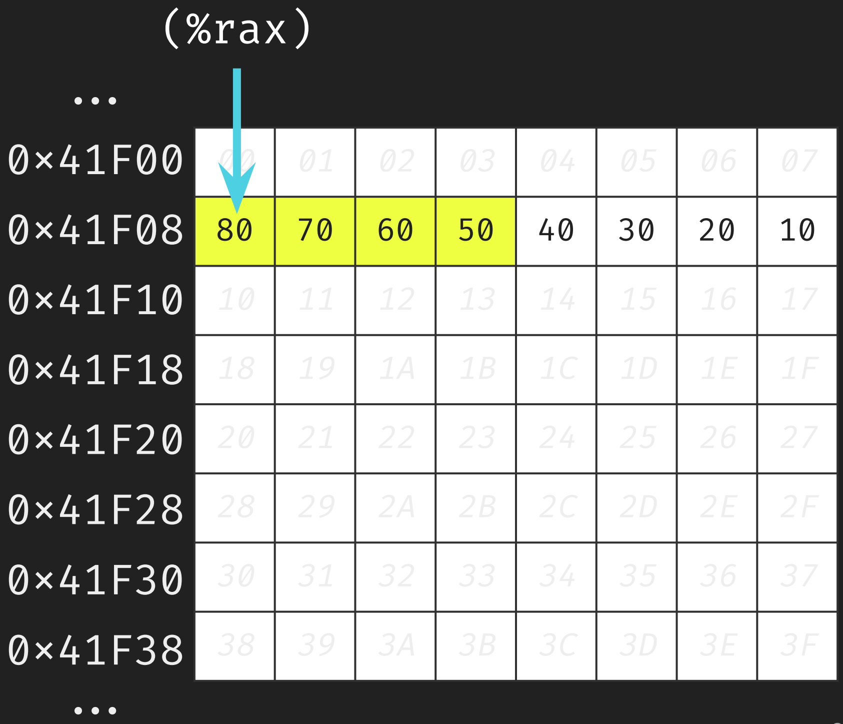
How much we move is determined by operand sizes / suffixes

Addressing memory

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

What's in %ebx?

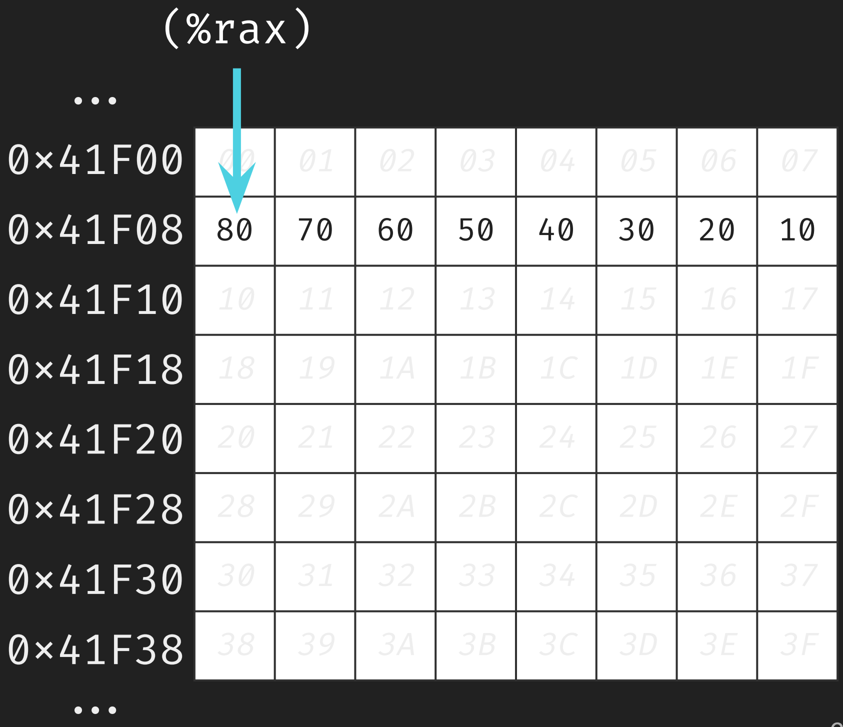
0x50607080



Addressing memory

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

What's in %bx?

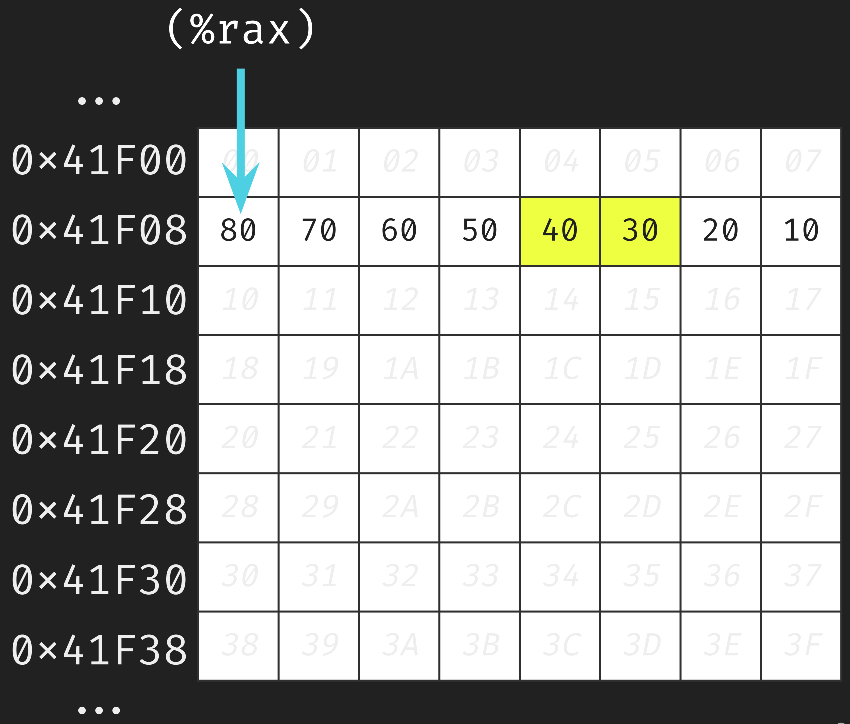


Addressing memory

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

What's in %bx?

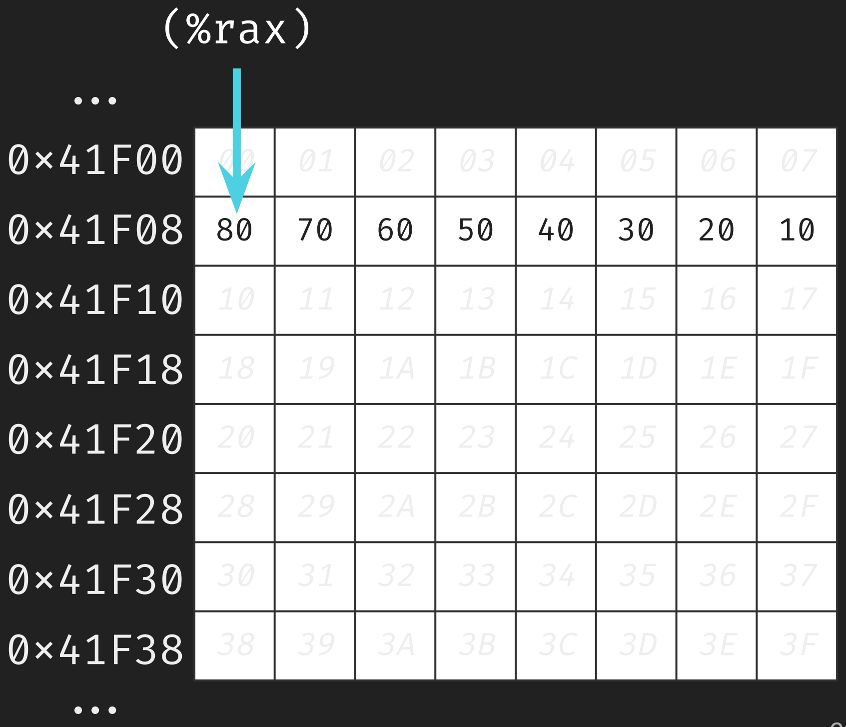
0x3040



Addressing memory

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

What's in %bx?

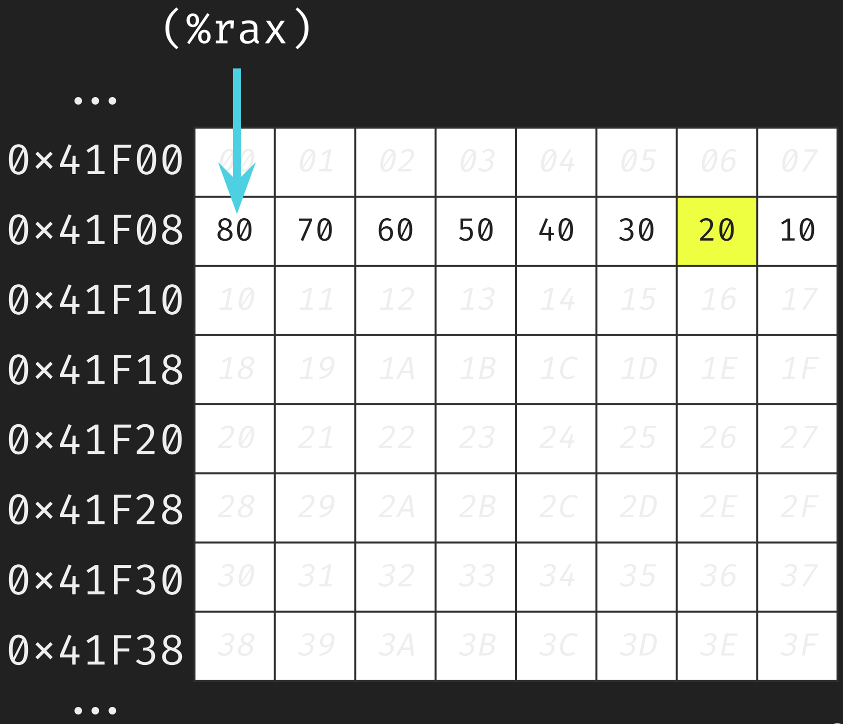


Addressing memory

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

What's in %bx?

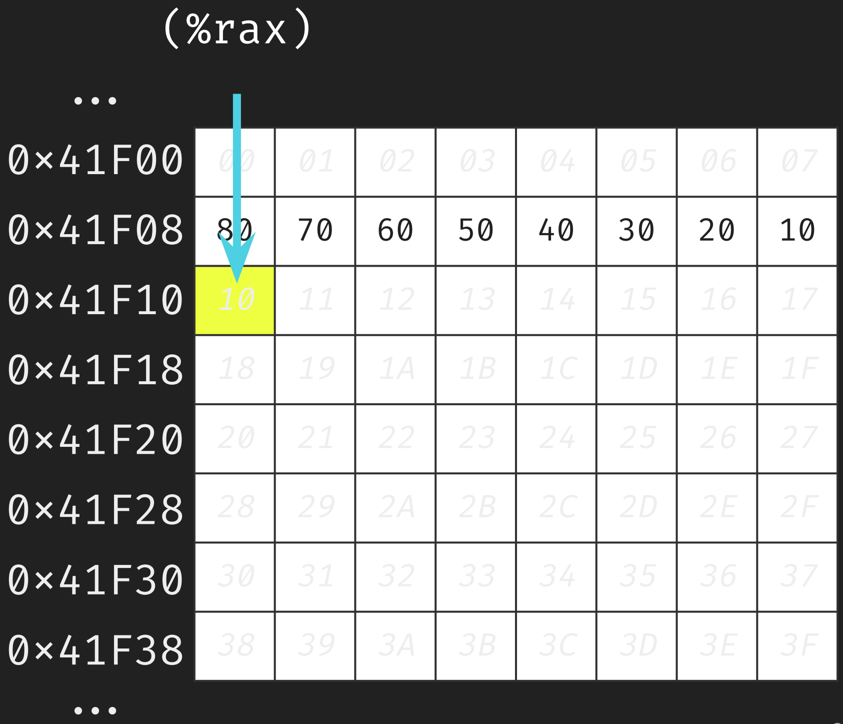
0x0020



Addressing memory

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

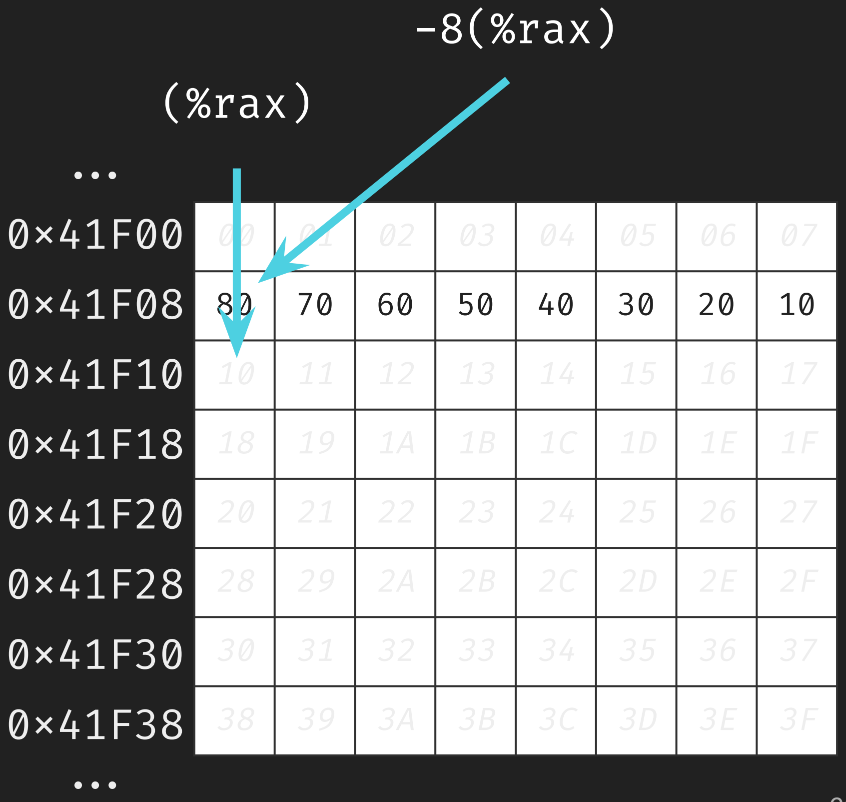
Modifying %rax changes where it points



Addressing memory

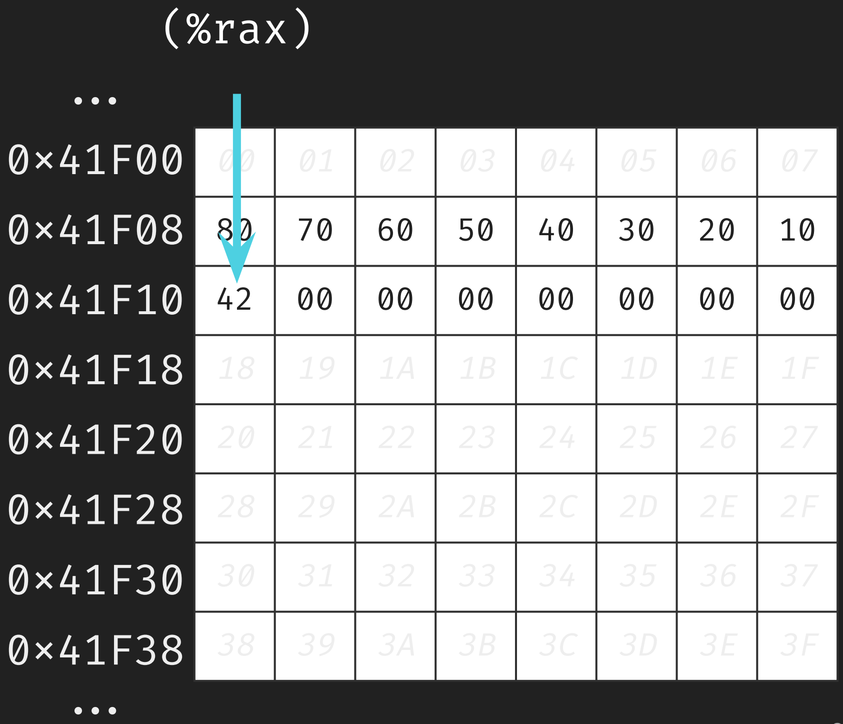
```
add $8, %rax
```

Modifying %rax changes where it points



Addressing memory

```
add $8, %rax
movq $42, (%rax)
```



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) element index
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

loop_end:
```

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

What's in %r10 after loop_end?

The Stack

How to 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

1. What are arguments?
2. What is returned?

```
#long fact(long n)
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

1. Storing temp variable on the stack
2. Returning result in %rax

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

Follow Design Recipe: Function Skeleton

```
#long fact(long n)
fact:
# n-1 → (-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-1 → (-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.
```

Foll

```
#long fact(long n)
fact:
# n-1 → (-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
```