

# **NEU CS 3650 Computer Systems**

Instructor: Dr. Ziming Zhao

\* Acknowledgements: created based on Christo Wilson, Ferdinand Vesely, Alden Jackson, Ben Weintraub, Gene Cooperman, Peter Desnoyers' lecture slides for the same course.

31 c0  
b8 00 00 00 00

xor eax, eax  
mov eax, 0x0

## CS3650 Computer Systems

[General](#)[Syllabus](#)[Schedule](#)[Office Hours](#)[Assignments](#)[Resources](#)

## Week 2

### ASSEMBLY

- [Readings](#)
- [Slides & Notes](#)
- [Code](#)
- [Additional Resources](#)

### Readings

- [Dive into Systems, Chapters 6 and 7](#)
- [Assembly Guide](#)

### Slides & Notes

- [Ferd's notes](#)
- [Ziming's slides](#)

### Code

*No files have been uploaded yet.*

### Additional Resources

Please look over these curated links.

- [Wikibooks x86 Assembly](#)
- [Wikibooks X86 Assembly/GAS Syntax](#)
- [x86-64 SysV ABI \(local PDF\)](#)
- [AMD Programmer's Manual, Volume 3](#)
- [AMD64 Linux Syscalls](#)
- [What is the 'ld' tool](#)
- [Writing a Function in Assembly: Intel x86 ATT Assembly Stack Part 1](#)
- [Writing a Function in Assembly: Intel x86 ATT Assembly Stack Part 2](#)
- [Writing a Function in Assembly: Intel x86 ATT Assembly Stack Part 3](#)

# Agenda

1. Memory (storage)
2. Stack - the data structure. Implementation in main memory
3. Recursion

# What we want for memory?

A tradeoff among Speed, Cost  
and Capacity

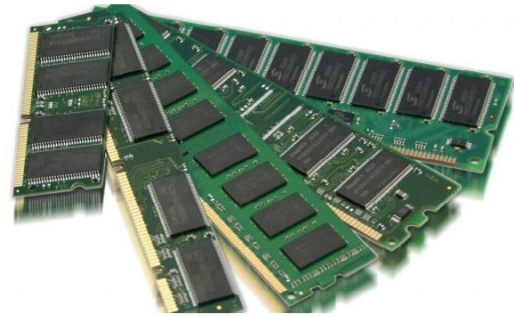
*Ideally one would desire an indefinitely large memory capacity such that any particular ... word would be immediately available. ... We are ... forced to recognize the possibility of constructing a hierarchy of memories, each of which has greater capacity than the preceding but which is less quickly accessible.*

**A. W. Burks, H. H. Goldstine, and  
J. von Neumann**

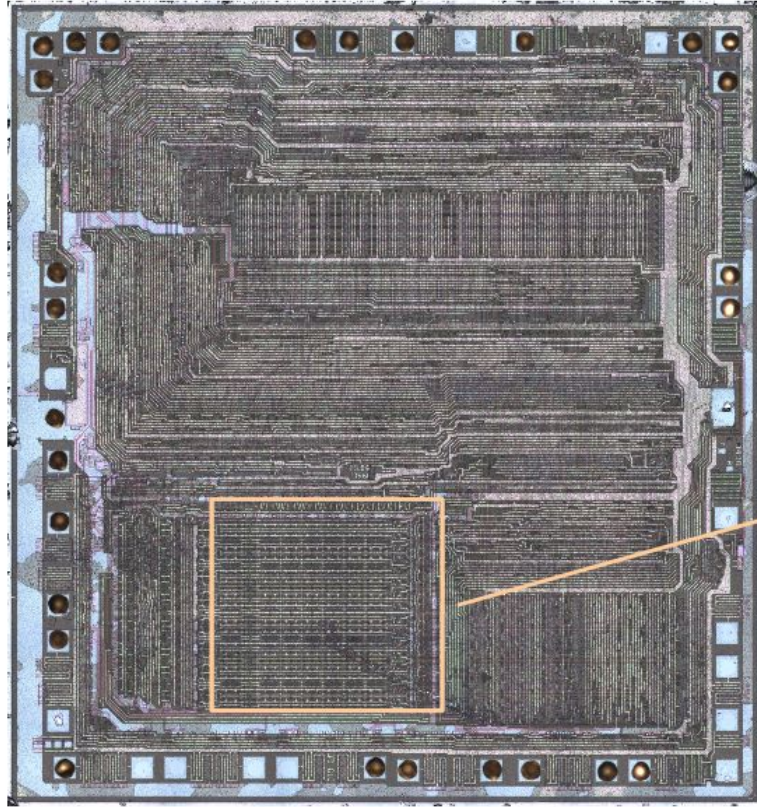
*Preliminary Discussion of the Logical Design of an  
Electronic Computing Instrument, 1946*

# 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

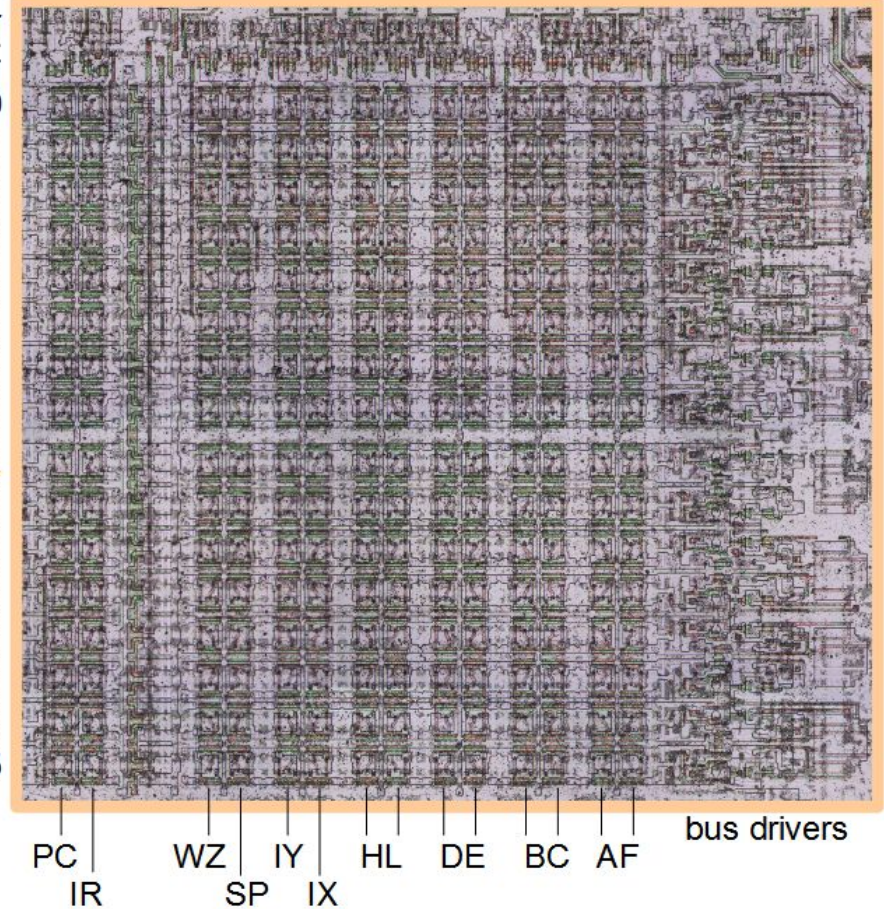


# CPU Registers (Z80, 16-bit register, 1976). We don't usually refer to registers as memory

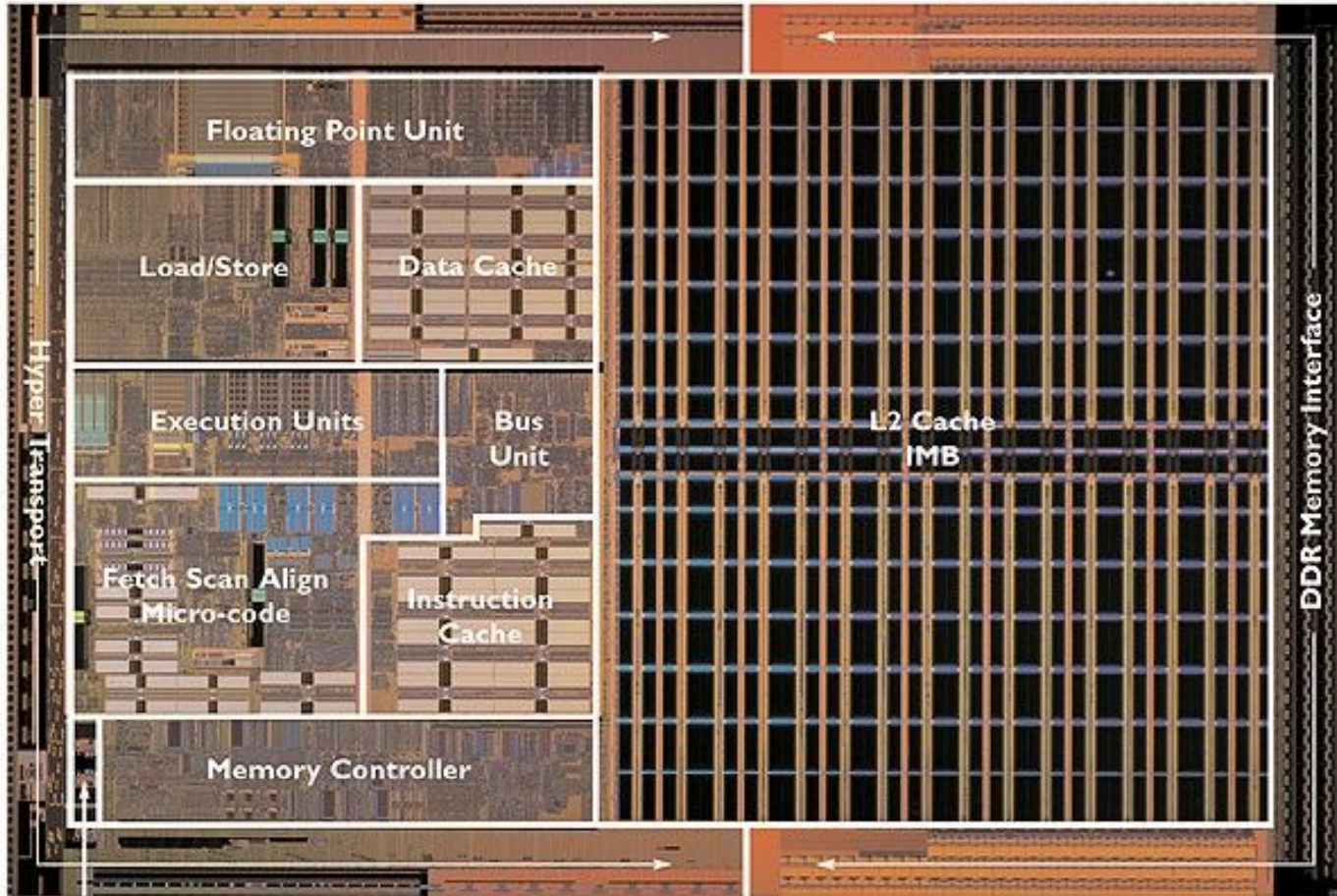


register  
select  
bit 0

bit 15



# Cache. AMD K8 (Athlon 64/Opteron) processor, from around the mid-2000s era.



# Cache. Intel Pentium 4.

## Intel Pentium 4 Northwood

### Buffer Allocation & Register Rename

Instruction Queue (for less critical fields of the uOps)

General Instruction Address Queue & Memory Instruction Address Queue (queues register entries and latency fields of the uOps for scheduling)

Floating Point, MMX, SSE2 Renamed Register File 128 entries of 128 bit.

### uOp Schedulers

FP Move Scheduler: (8x8 dependency matrix)

Parallel (Matrix) Scheduler for the two double pumped ALU's

General Floating Point and Slow Integer Scheduler: (8x8 dependency matrix)

Load / Store uOp Scheduler: (8x8 dependency matrix)

Load / Store Linear Address Collision History Table

### Integer Execution Core

- (1) uOp Dispatch unit & Replay Buffer Dispatches up to 6 uOps / cycle
- (2) Integer Renamed Register File 128 entries of 32 bit + 6 status flags 12 read ports and six write ports
- (3) Databus switch & Bypasses to and from the Integer Register File.
- (4) Flags, Write Back
- (5) Double Pumped ALU 0
- (6) Double Pumped ALU 1
- (7) Load Address Generator Unit
- (8) Store Address Generator Unit
- (9) Load Buffer ( 48 entries )
- (10) Store Buffer ( 24 entries )

### Execution Pipeline Start

Register Alias History Tables (2x126)  
Register Alias Tables uOp Queue

Micro code Sequencer  
Micro code ROM & Flash

### Instruction Trace Cache

Trace Cache Fill Buffers  
Distributed Tag comparators 24 bit virtual Tags

### Trace Cache Access, next Address Predict

Trace Cache Branch Prediction Table (BTB), 512 entries.  
Return Stacks (2x16 entries)  
Trace Cache next IP's (2x)  
Miscellaneous Tag Data

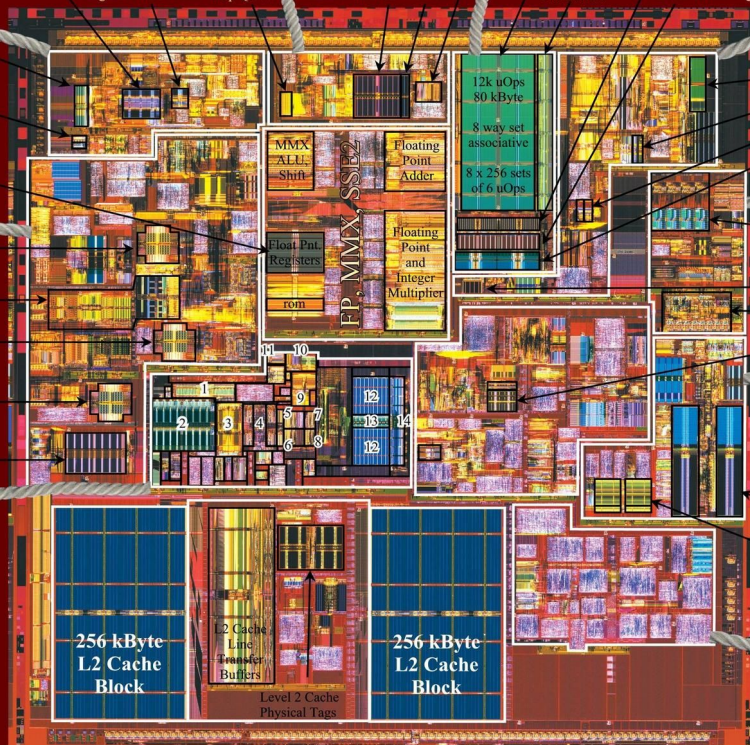
### Instruction Decoder

Up to 4 decoded uOps/cycle out. (from max. one x86 instr/cycle)  
Instructions with more than four are handled by Micro Sequencer  
Trace Cache LRU bits  
Raw Instruction Bytes in Data TLB, 64 entry fully associative, between threads dual ported (for loads and stores)

### Instruction Fetch from L2 cache and Branch Prediction

Front End Branch Prediction Tables (BTB), shared, 4096 entries in total  
Instruction TLB's 2x64 entry, fully associative for 4k and 4M pages. In: Virtual address [31:12] Out: Physical address [35:12] + 2 page level bits

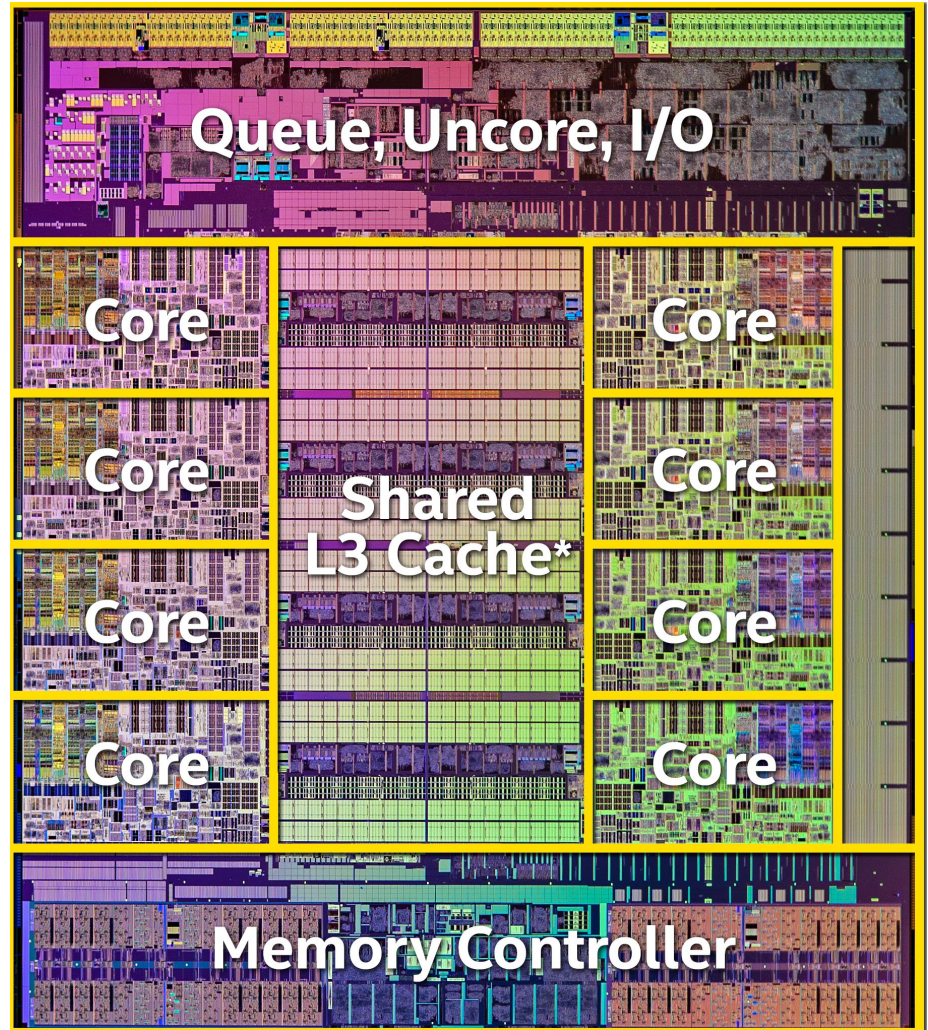
### Front Side Bus Interface, 400..800 MHz



- (11) ROB Reorder Buffer 3x42 entries
- (12) 8 kByte Level 1 Data cache four way set associative. 1R/1W

- (13) Summed Address Index decode and Way Predict
- (14) Cache Line Read / Write Transferbuffers and 256 bit wide bus to and from L2 cache

**Cache. Intel Xeon  
E5/E7 Sandy Bridge-EP  
around 2011–2013,  
showing a 10-core  
server-class die layout.**

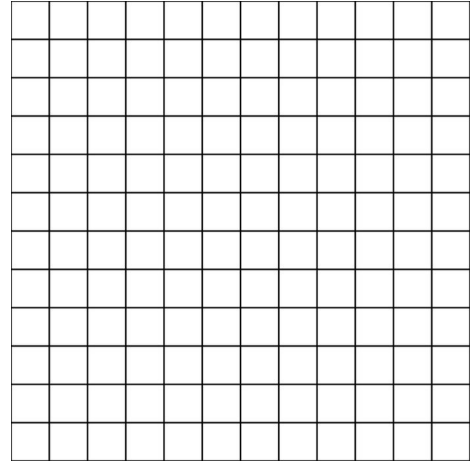


Level	Latency (Typical)	Bandwidth (Typical)	Typical Size (per core / chip)	Approx. Cost per GB (USD)	Notes
CPU Registers	~0.2–0.5 ns	~5–10 TB/s (internal)	Tens to hundreds of registers	on-die only	Fastest storage, directly wired into execution units
L1 Cache (SRAM)	~0.5 ns	~1–2 TB/s	32–64 KB per core	on-die only	Separate instruction & data caches
L2 Cache (SRAM)	~2–5 ns	~500 GB/s	256 KB–1 MB per core	on-die only	Private per core
L3 Cache (SRAM)	~10–15 ns	~100–300 GB/s	8–64 MB shared per chip	on-die only	Shared among all cores
Main Memory (DRAM)	~60–100 ns	~50–100 GB/s	8–128 GB	\$3–\$5	Volatile, off-chip, DDR4/DDR5
NVMe SSD (Flash)	~50–100 µs	~2–7 GB/s	0.5–4 TB	\$0.05–\$0.10	Non-volatile, PCIe interface
SATA SSD (Flash)	~100–200 µs	~500 MB/s	0.5–2 TB	~\$0.05	Slower interface than NVMe
HDD (Magnetic disk)	~5–10 ms	~100–200 MB/s	1–20 TB	~\$0.02	Mechanical, highest latency

For now, we discuss the  
**Main Memory (DRAM)**  
and we can think of it as a  
giant linear array.

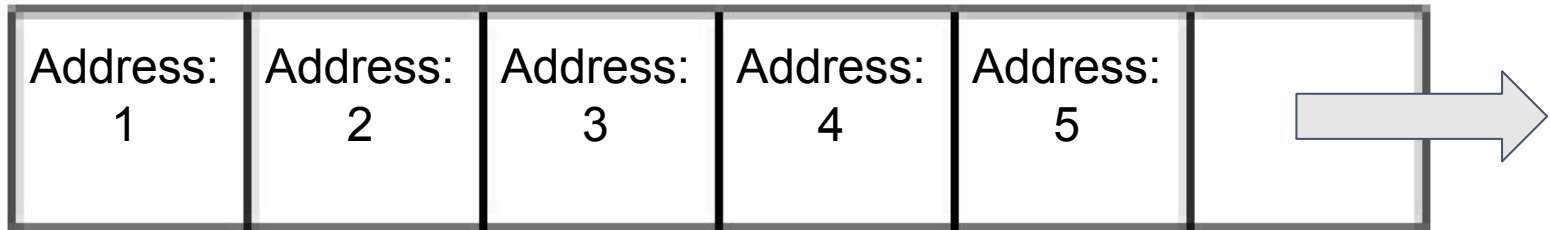
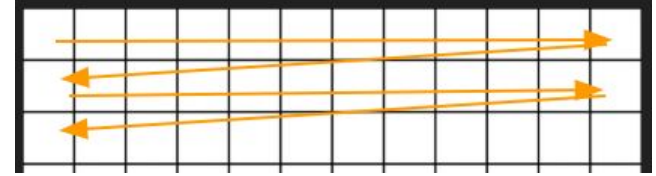
# Linear array of memory

- 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

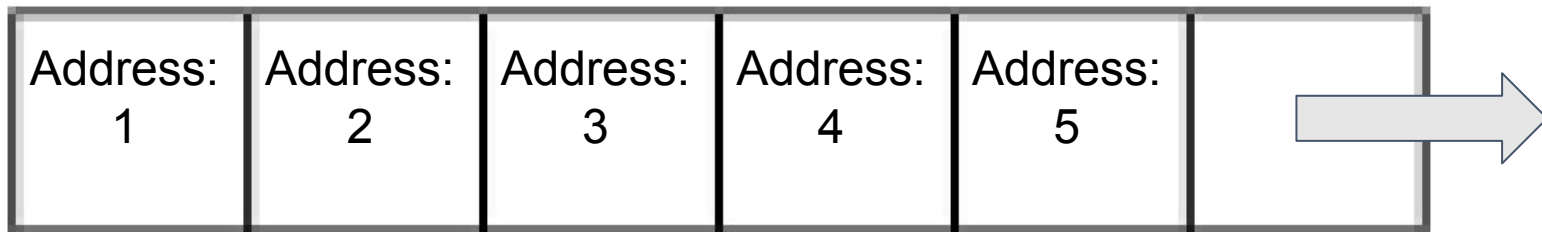


# Linear array of memory

- 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>b</b>	1	0 to 255 ( $=2^8$ )
short int	<b>w</b>	2	0 to 65,535 ( $=2^{16}$ )
int	<b>l</b>	4	0 to 4,294,967,295 ( $=2^{32}$ )
long int	<b>q</b>	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

...

(%rax)

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

Offset addressing:

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

...

(%rax)

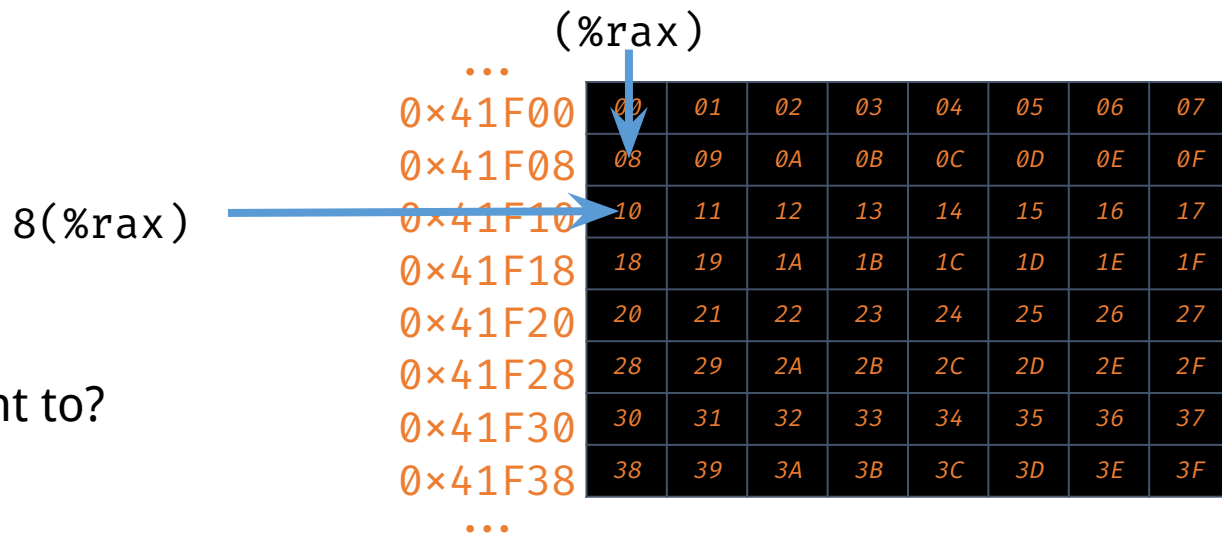


0x41F00	00	01	02	03	04	05	06	07
0x41F08	08	09	0A	0B	0C	0D	0E	0F
0x41F10	10	11	12	13	14	15	16	17
0x41F18	18	19	1A	1B	1C	1D	1E	1F
0x41F20	20	21	22	23	24	25	26	27
0x41F28	28	29	2A	2B	2C	2D	2E	2F
0x41F30	30	31	32	33	34	35	36	37
0x41F38	38	39	3A	3B	3C	3D	3E	3F

...

# Addressing memory

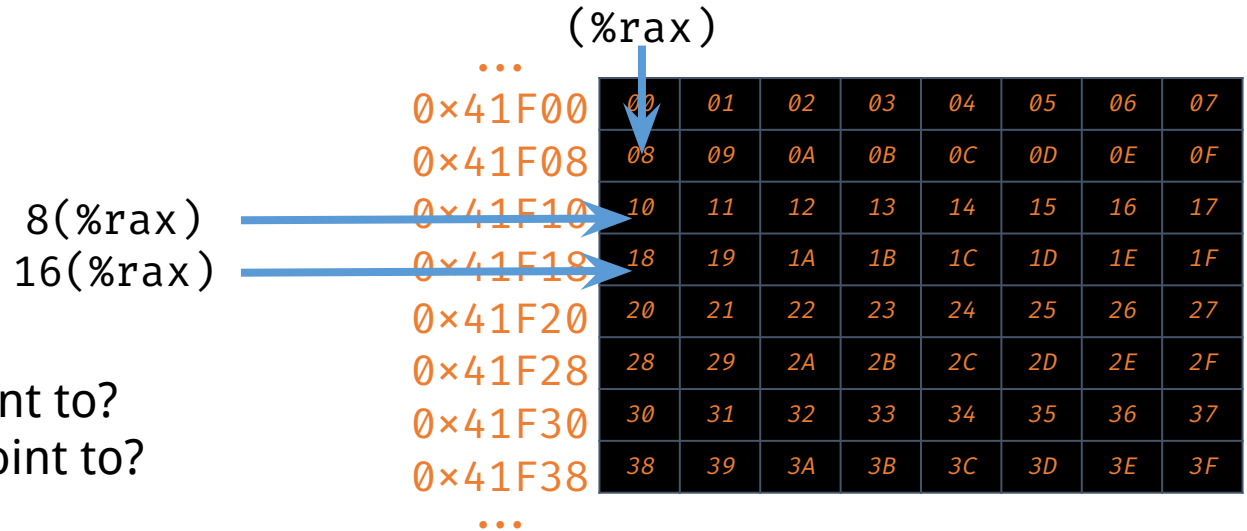
Offset addressing



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

# Addressing memory

Offset addressing

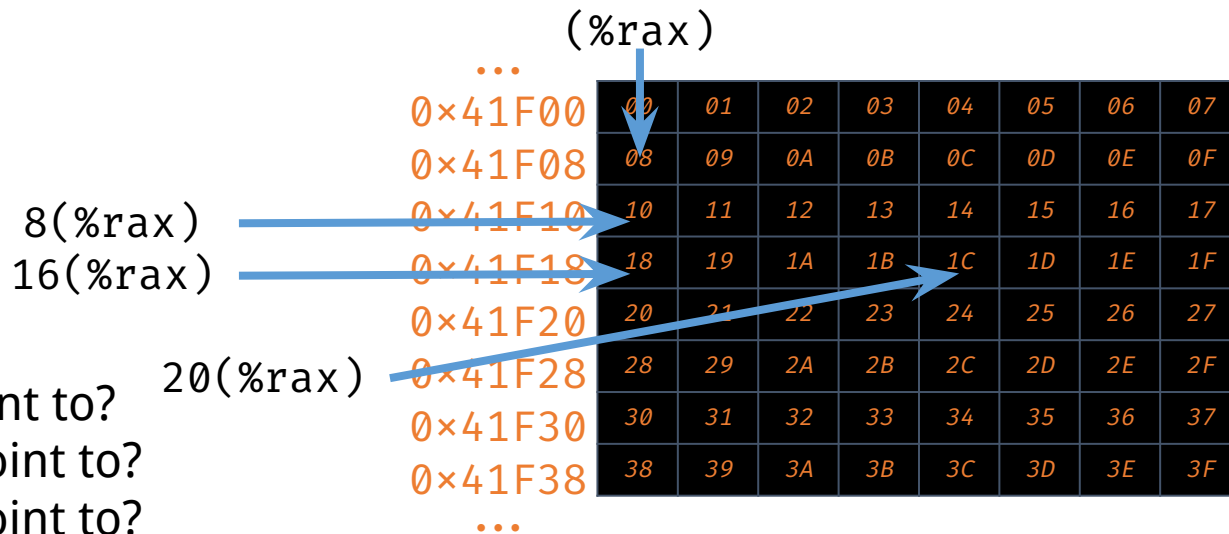


Where does 8(%rax) point to?

Where does 16(%rax) point to?

# Addressing memory

## Offset addressing



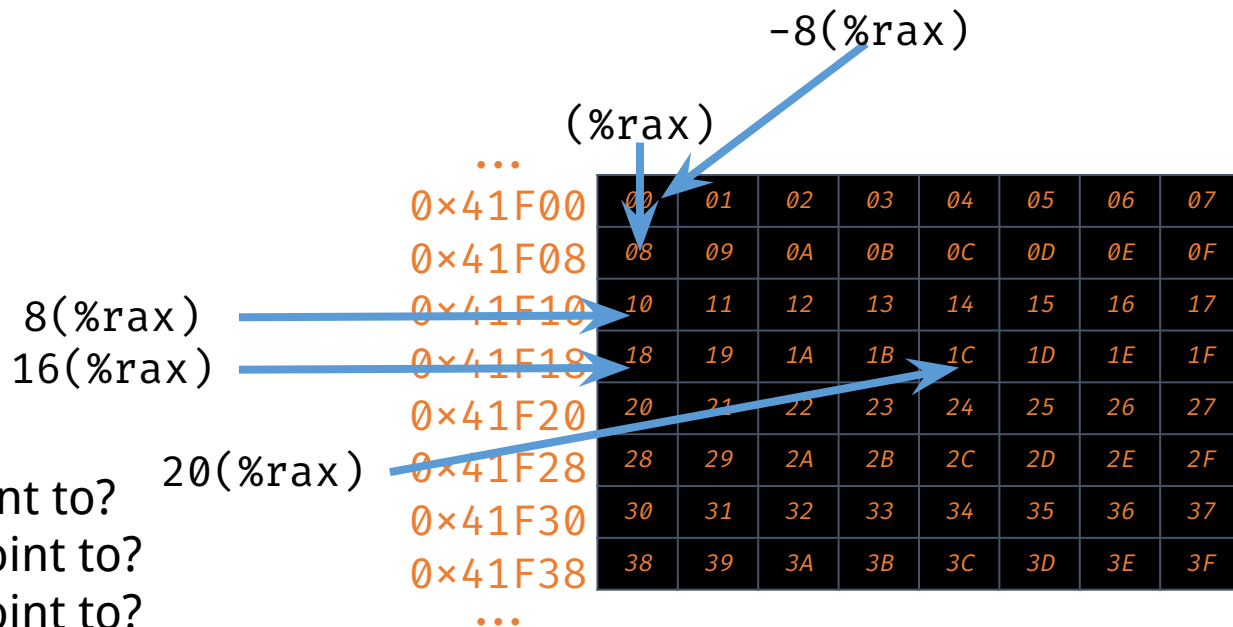
## Where does 8(%rax) point to?

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

## Where does 20(%rax) point to?

# Addressing memory

Offset addressing



Where does  $8(\%rax)$  point to?

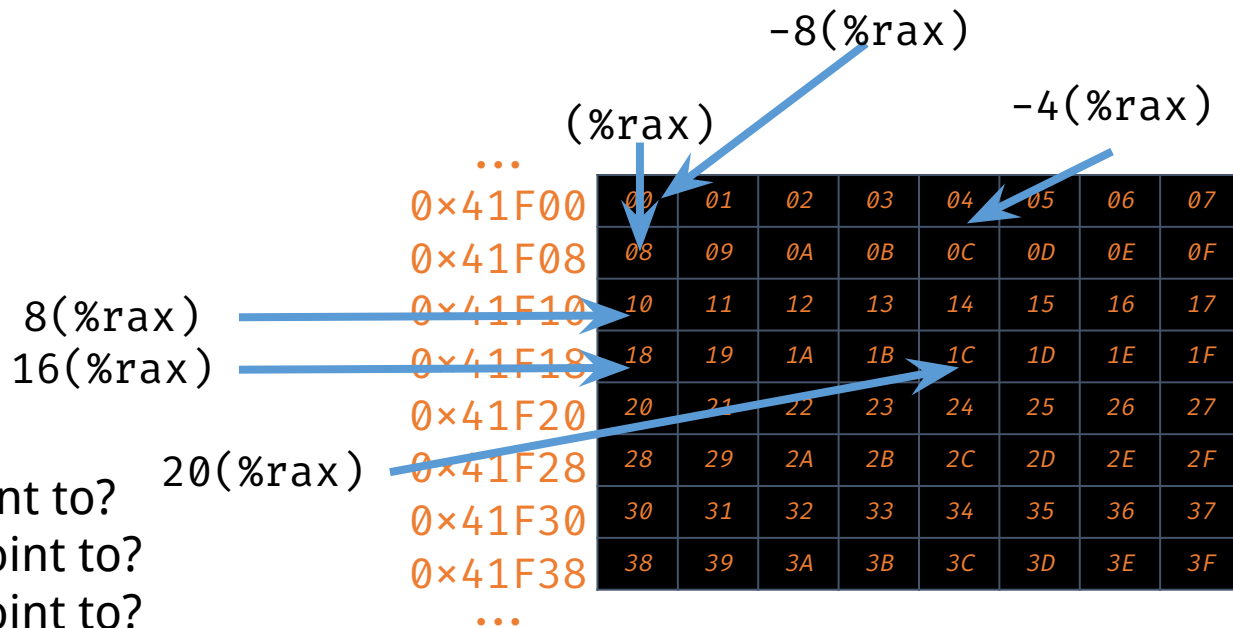
Where does  $16(\%rax)$  point to?

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?

(%rax)

...

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 $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**

...

0x41F00

0x41F08

0x41F10

0x41F18

0x41F20

0x41F28

0x41F30

0x41F38

...

(%rax)

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

# 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*)

...

0x41F00

0x41F08

0x41F10

0x41F18

0x41F20

0x41F28

0x41F30

0x41F38

...

(%rax)

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

...

0x41F00

0x41F08

0x41F10

0x41F18

0x41F20

0x41F28

0x41F30

0x41F38

...

(%rax)

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

...

0x41F00

0x41F08

0x41F10

0x41F18

0x41F20


0x41F28

0x41F30

0x41F38

...

(%rax)



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

# Addressing memory

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

What's in %ebx?

0x50607080

...

(%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?

(%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

...

0x41F00

0x41F08

0x41F10

0x41F18

0x41F20


0x41F28

0x41F30

0x41F38

...

(%rax)



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

# Addressing memory

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

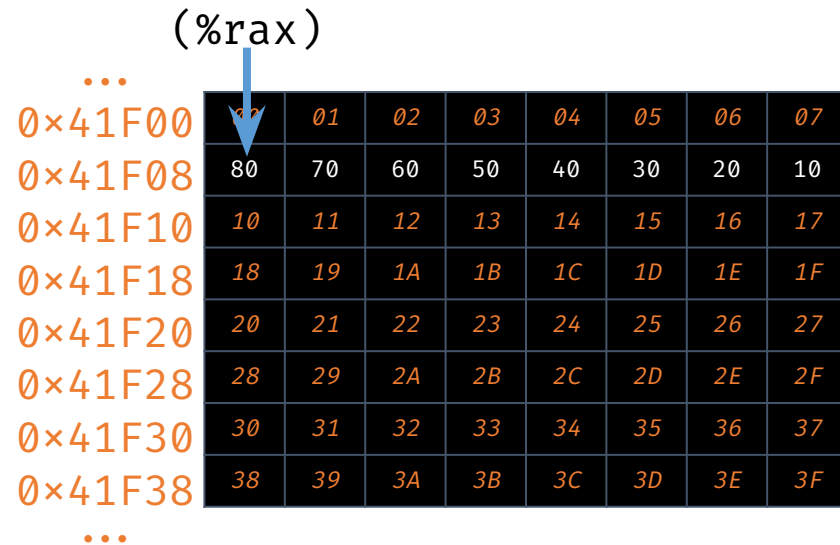
What's in %bl?

(%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

...

0x41F00

0x41F08

0x41F10

0x41F18

0x41F20

0x41F28

0x41F30

0x41F38

...

(%rax)

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

# Addressing memory

add \$8, %rax

Modifying %rax changes where it points

(%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

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

## Modifying %rax changes where it points

Diagram illustrating the stack frame layout for a function call. The stack grows downwards (increasing memory address).

Memory addresses (left):

- 0x41F00
- 0x41F08
- 0x41F10
- 0x41F18
- 0x41F20
- 0x41F28
- 0x41F30
- 0x41F38

Stack contents (hexadecimal values):

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

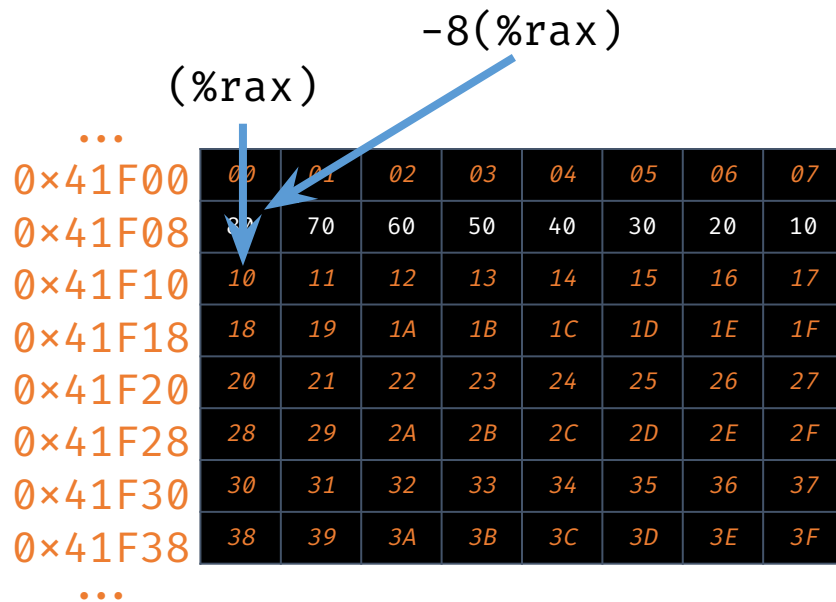
Annotations:

- Blue arrow pointing to address 0x41F00: `(%rax)`
- Blue arrow pointing to address 0x41F08: `-8(%rax)`

# Addressing memory

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

How does movq change the memory state?



# Addressing memory

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

Modifying %rax changes where it points

(%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

...

# 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:
```

...

0x41F00

0x41F08

0x41F10

0x41F18

0x41F20

0x41F28

0x41F30

0x41F38

...

01
02
03
04
05
06
07
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 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.
- In functions, we traverse our byte array as a stack

**API** = *how you call it in source code*

e.g. you `#include <stdio.h>` and call `printf()`

**ABI** = *how that call actually looks at the binary level*

e.g. the compiled code puts `format` in `%rdi`, sets up the stack frame, aligns the stack, and expects the return value in `%rax`.

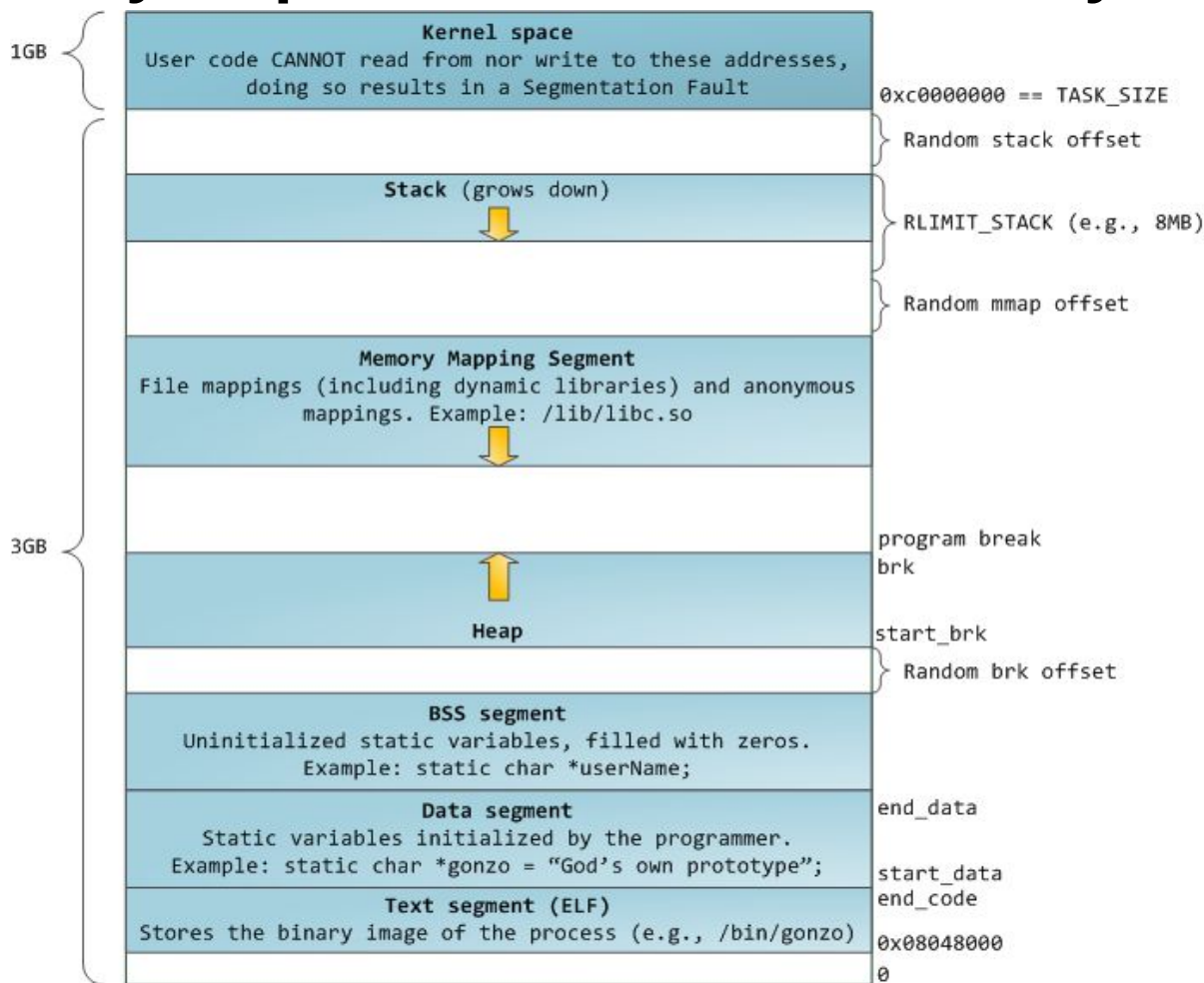
# Memory Map of Linux Process (32 bit)

Each process in a multi-tasking OS runs in its own memory sandbox.

This sandbox is the **virtual address space**, which in 32-bit mode is **always a 4GB block of memory addresses**.

These virtual addresses are mapped to physical memory by **page tables**, which are maintained by the operating system kernel and consulted by the processor.

# Memory Map of Linux Process (32 bit system)



<https://manybutfinite.com/pos-anatomy-of-a-program-in-memory/>

# /proc/pid\_of\_process/maps

## Example processmap.c

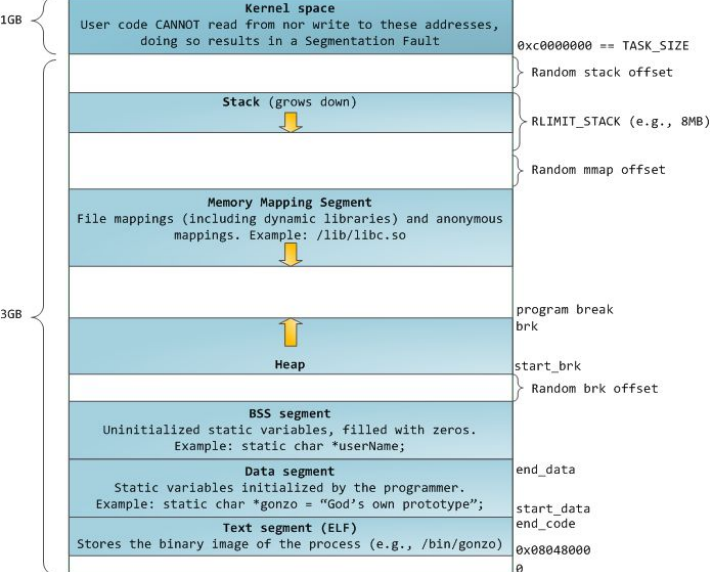
```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    getchar();
    return 0;
}
```

cat /proc/pid/maps

pmap -X pid

pmap -X `pidof pm`



```

ziming@ziming-ThinkPad:~/Dropbox/myTeaching/System Security - Attack and Defense for Binaries UB 2020/code/processmap$ pmap -X 21732
21732:  ./pm
Address Perm  Offset Device      Inode  Size  Rss  Pss  Referenced  Anonymous  LazyFree  ShmemPmdMapped  Shared_Hugetlb  Private_Hugetlb  Swap  SwapPss  Locked  Mapping
56569000 r-xp 00000000 103:02 28575310 4 4 4 4 0 0 0 0 0 0 0 0 pm
5656a000 r--p 00000000 103:02 28575310 4 4 4 4 4 0 0 0 0 0 0 0 pm
5656b000 rw-p 00001000 103:02 28575310 4 4 4 4 4 0 0 0 0 0 0 0 pm
57cf2000 rw-p 00000000 00:00 0 136 4 4 4 4 0 0 0 0 0 0 0 [heap]
f7d73000 r-xp 00000000 103:02 2883591 1876 772 772 772 0 0 0 0 0 0 0 0 libc-2.27.so
f7f48000 ---p 001d5000 103:02 2883591 4 0 0 0 0 0 0 0 0 0 0 0 libc-2.27.so
f7f49000 r--p 001d5000 103:02 2883591 8 8 8 8 8 0 0 0 0 0 0 0 0 libc-2.27.so
f7f4b000 rw-p 001d7000 103:02 2883591 4 4 4 4 4 0 0 0 0 0 0 0 0 libc-2.27.so
f7f4c000 rw-p 00000000 00:00 0 12 8 8 8 8 0 0 0 0 0 0 0 0
f7f75000 rw-p 00000000 00:00 0 8 8 8 8 8 0 0 0 0 0 0 0 0
f7f77000 r--p 00000000 00:00 0 12 0 0 0 0 0 0 0 0 0 0 0 [vvar]
f7f7a000 r-xp 00000000 00:00 0 8 8 8 8 8 0 0 0 0 0 0 0 [vdso]
f7f7c000 r-xp 00000000 103:02 2883587 152 144 144 144 0 0 0 0 0 0 0 0 ld-2.27.so
f7fa2000 r--p 00025000 103:02 2883587 4 4 4 4 4 0 0 0 0 0 0 0 0 ld-2.27.so
f7fa3000 rw-p 00026000 103:02 2883587 4 4 4 4 4 0 0 0 0 0 0 0 0 ld-2.27.so
ffef3000 rw-p 00000000 00:00 0 132 12 12 12 12 12 0 0 0 0 0 0 0 [stack]
=====
2372 988 988 988 60 0 0 0 0 0 0 0 0 0 0 0 KB

```

# Memory Map of Linux Process (64 bit system)

```
ziming@ziming-ThinkPad:~/Dropbox/myTeaching/System Security - Attack and Defense for Binaries UB 2020/code/processmap$ pmap -X 22891
```

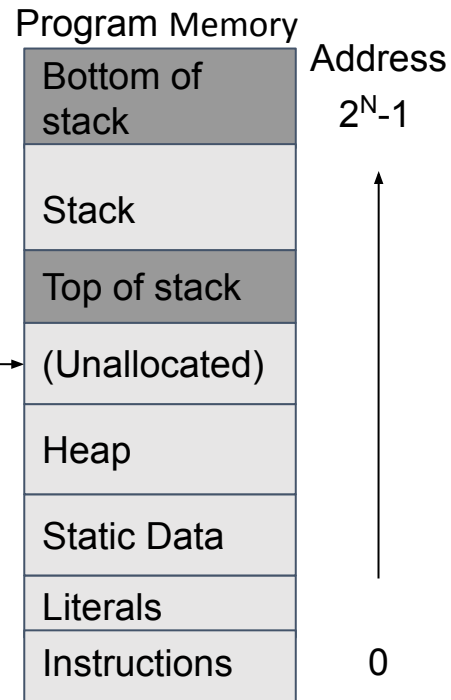
22891:	./pm64																	
	Address	Perm	Offset	Device	Inode	Size	Rss	Pss	Referenced	Anonymous	LazyFree	ShmemPmdMapped	Shared_Hugetlb	Private_Hugetlb	Swap	SwapPss	Locked	Mapping
	55bf7ae37000	r-xp	00000000	103:02	28577490	4	4	4	4	0	0	0	0	0	0	0	0	pm64
	55bf7b037000	r--p	00000000	103:02	28577490	4	4	4	4	4	0	0	0	0	0	0	0	pm64
	55bf7b038000	rw-p	00001000	103:02	28577490	4	4	4	4	4	0	0	0	0	0	0	0	pm64
	55bf7cc0c000	rw-p	00000000	00:00	0	132	4	4	4	4	0	0	0	0	0	0	0	[heap]
	7fc7ebdb6000	r-xp	00000000	103:02	660090	1948	992	5	992	0	0	0	0	0	0	0	0	libc-2.27.so
	7fc7ebf9d000	---p	001e7000	103:02	660090	2048	0	0	0	0	0	0	0	0	0	0	0	libc-2.27.so
	7fc7ec19d000	r--p	001e7000	103:02	660090	16	16	16	16	16	0	0	0	0	0	0	0	libc-2.27.so
	7fc7ec1a1000	rw-p	001eb000	103:02	660090	8	8	8	8	8	0	0	0	0	0	0	0	libc-2.27.so
	7fc7ec1a3000	rw-p	00000000	00:00	0	16	12	12	12	12	0	0	0	0	0	0	0	0
	7fc7ec1a7000	r-xp	00000000	103:02	660062	156	156	0	156	0	0	0	0	0	0	0	0	ld-2.27.so
	7fc7ec3a6000	rw-p	00000000	00:00	0	8	8	8	8	8	0	0	0	0	0	0	0	0
	7fc7ec3ce000	r--p	00027000	103:02	660062	4	4	4	4	4	0	0	0	0	0	0	0	ld-2.27.so
	7fc7ec3cf000	rw-p	00028000	103:02	660062	4	4	4	4	4	0	0	0	0	0	0	0	ld-2.27.so
	7fc7ec3d0000	rw-p	00000000	00:00	0	4	4	4	4	4	0	0	0	0	0	0	0	0
	7ffe05803000	rw-p	00000000	00:00	0	132	12	12	12	12	0	0	0	0	0	0	0	[stack]
	7ffe058b9000	r--p	00000000	00:00	0	12	0	0	0	0	0	0	0	0	0	0	0	[vvar]
	7ffe058bc000	r-xp	00000000	00:00	0	8	4	0	4	0	0	0	0	0	0	0	0	[vdso]
	fffffffff600000	r-xp	00000000	00:00	0	4	0	0	0	0	0	0	0	0	0	0	0	[syscall]
=====						=====	=====	=====	=====	=====	=====	=====	=====	=====	=====	=====	=====	=====
						4512	1236	89	1236	80	0	0	0	0	0	0	0	0 KB

# 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.



# Stack

Stack is essentially scratch memory for functions

- Used in MIPS, ARM, x86, and x86-64 processors

Starts at high memory addresses, and grows down

Functions are free to push registers or values onto the stack, or pop values from the stack into registers

The assembly language supports this on x86

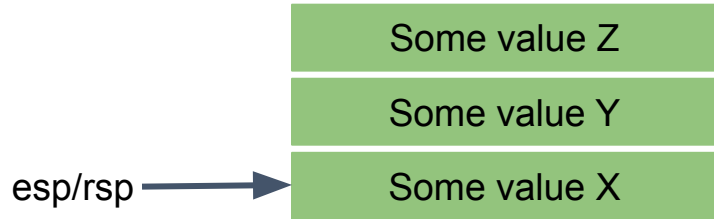
- **esp/rsp** holds the address of the top of the stack
- push eax/rax 1) decrements the stack pointer (esp/rsp) then 2) stores the value in eax/rax to the location pointed to by the stack pointer
- pop eax/rax 1) stores the value at the location pointed to by the stack pointer into eax/rax, then 2) increments the stack pointer (esp/rsp)

# **x86/64 Instructions that affect Stack**

push, pop, call, ret, enter, leave

# x86/64 Instructions that affect Stack

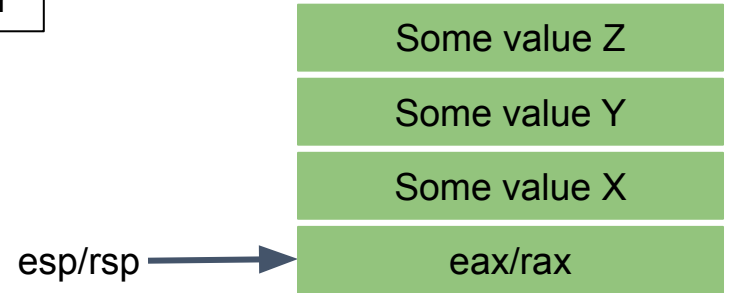
Before:



**push `eax/rax`**

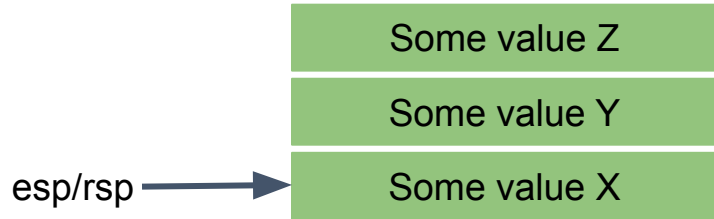


After



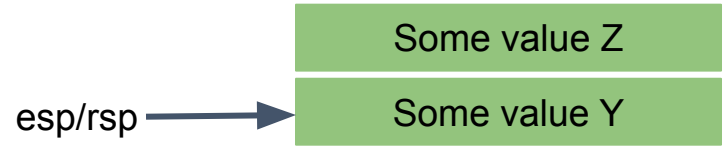
# x86/64 Instructions that affect Stack

Before:

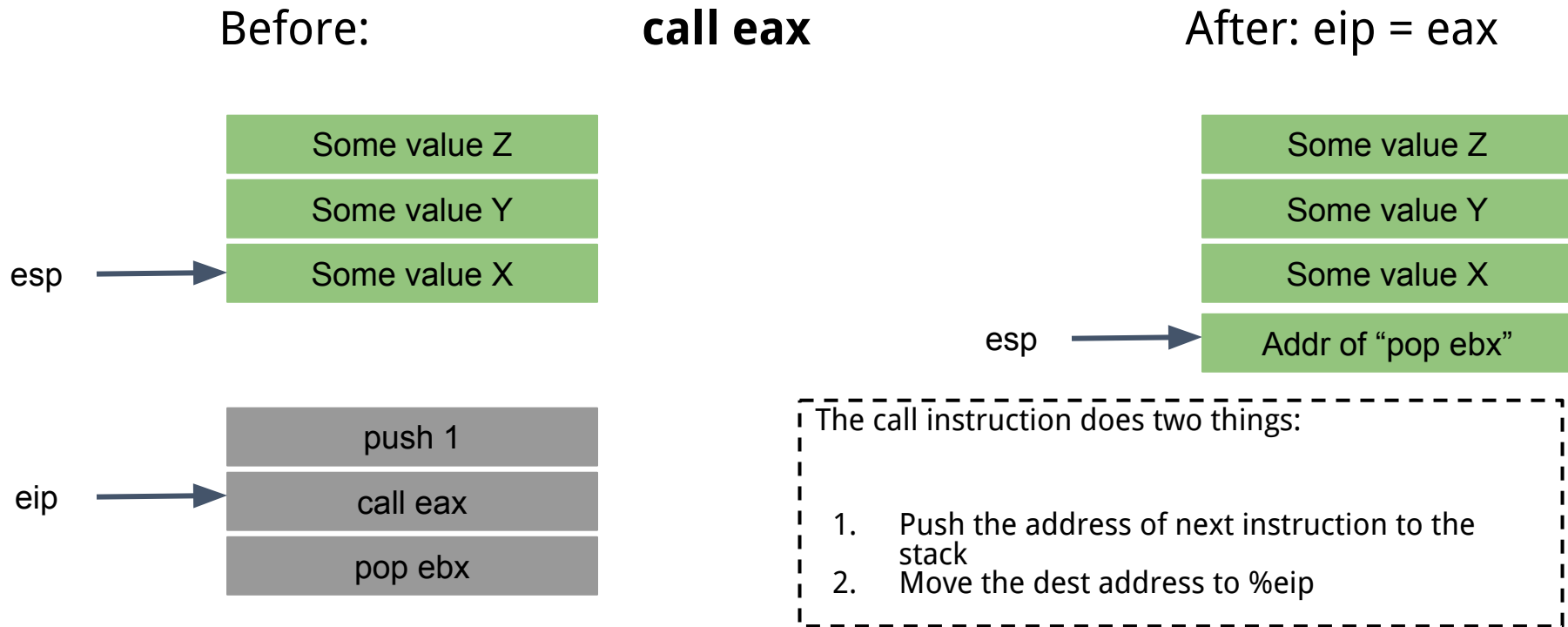


**pop eax/rax**

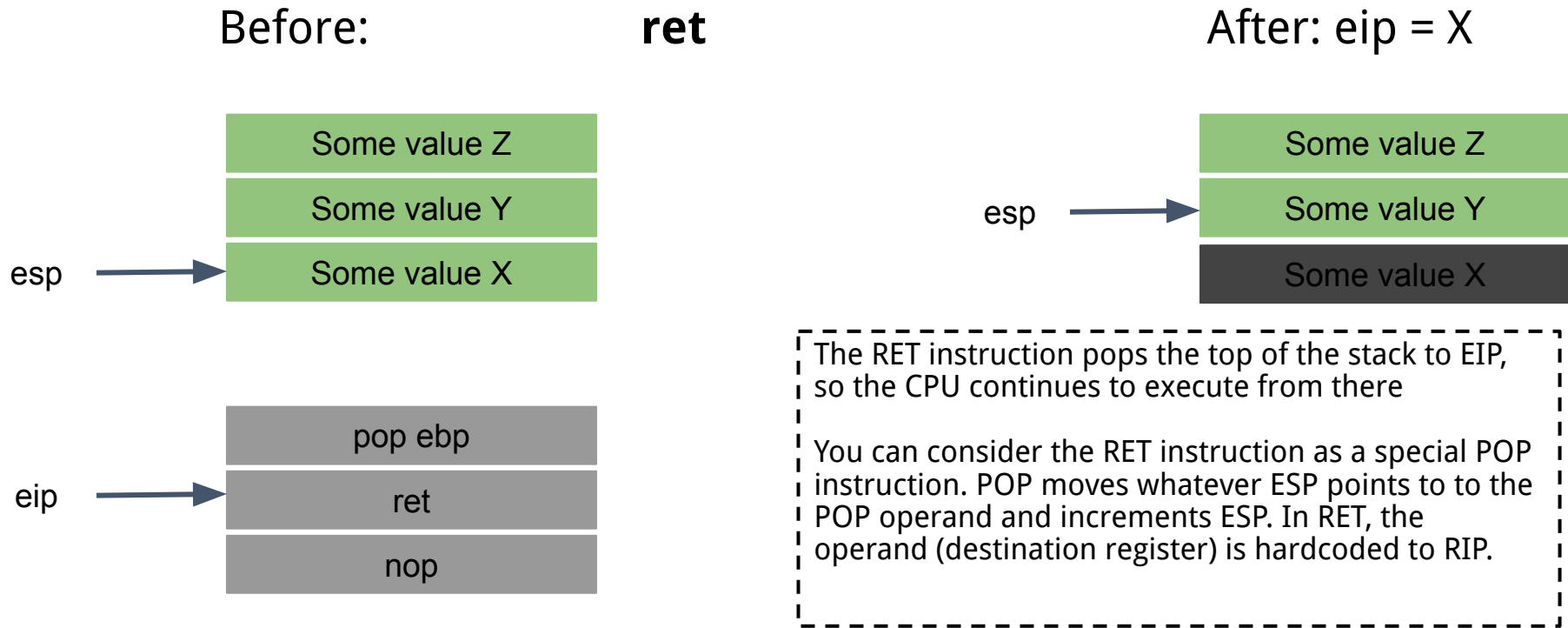
After:  $\text{eax/rax} = X$



# x86/64 Instructions that affect Stack



# x86/64 Instructions that affect Stack

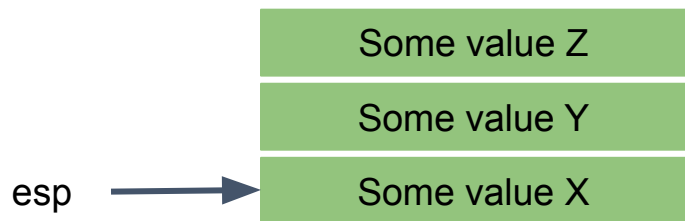


# x86/64 Instructions that affect Stack

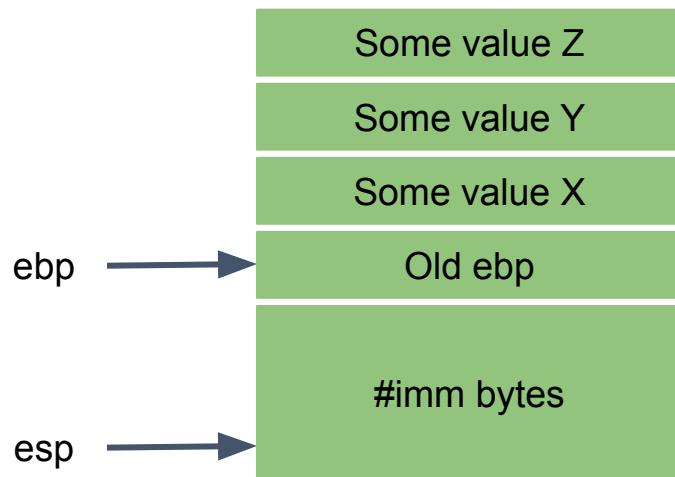
```
push ebp  
mov ebp, esp  
sub esp, #imm
```

**enter**

Before:



After:



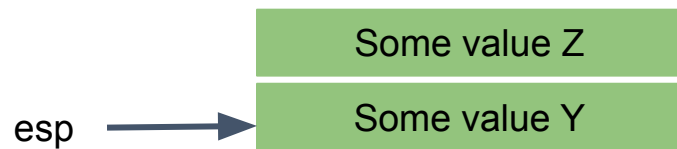
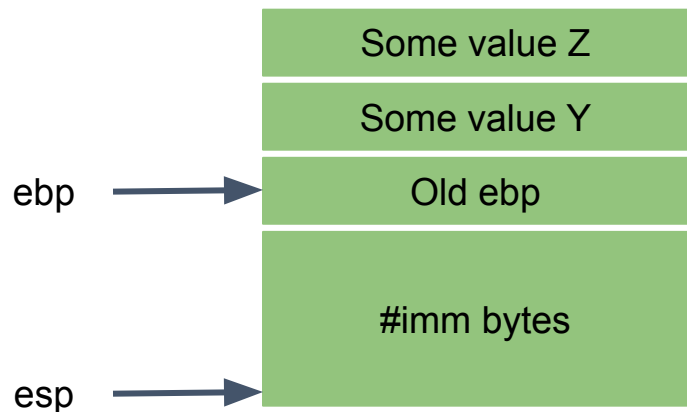
# x86/64 Instructions that affect Stack

```
mov esp, ebp  
pop ebp
```

Before:

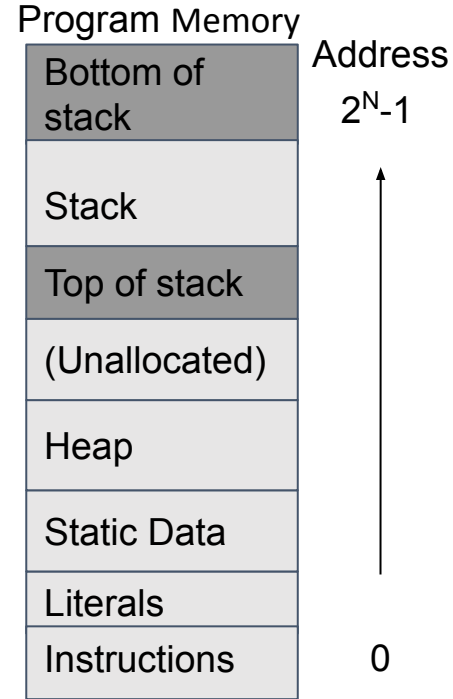
**leave**

After: ebp = old ebp



# x86-64 Stack

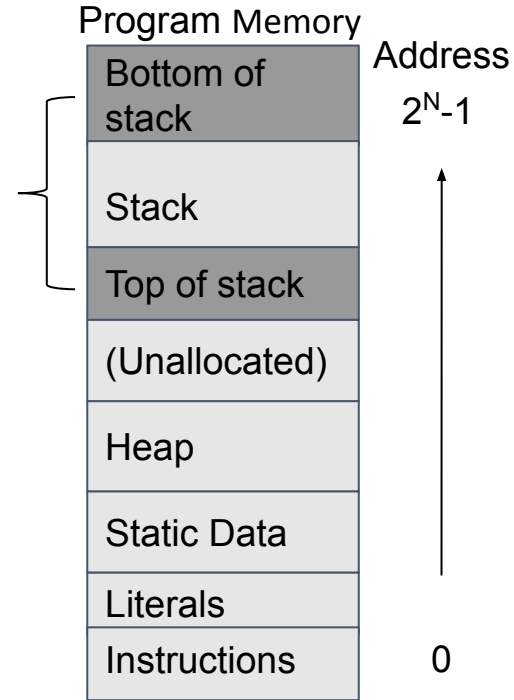
- There is a stack at the top of the memory
  - Yes, the stack that you learned in data structures course
  - You can push and pop data



# 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

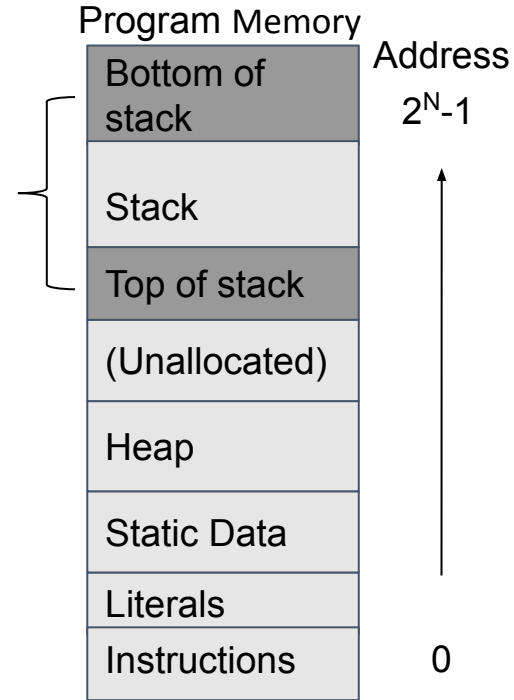


# x86-64 Stack

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 assembly to remind you that things are *growing down* in the stack

Program Memory

Bottom of stack

Stack

Top of stack

(Unallocated)

Heap

Static Data

Literals

Instructions

Address

$2^N - 1$

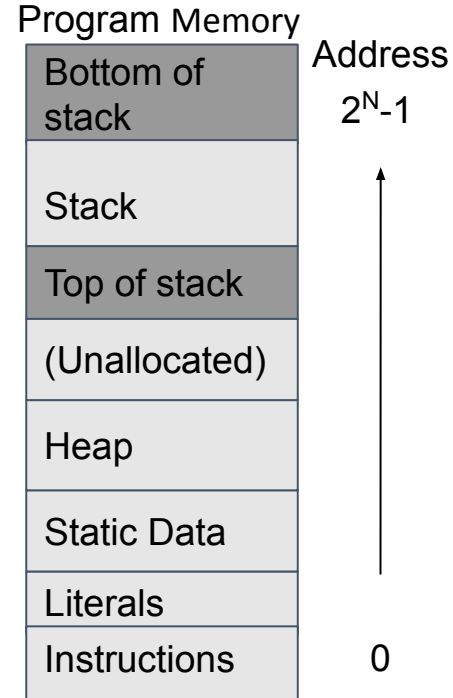


0

# 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

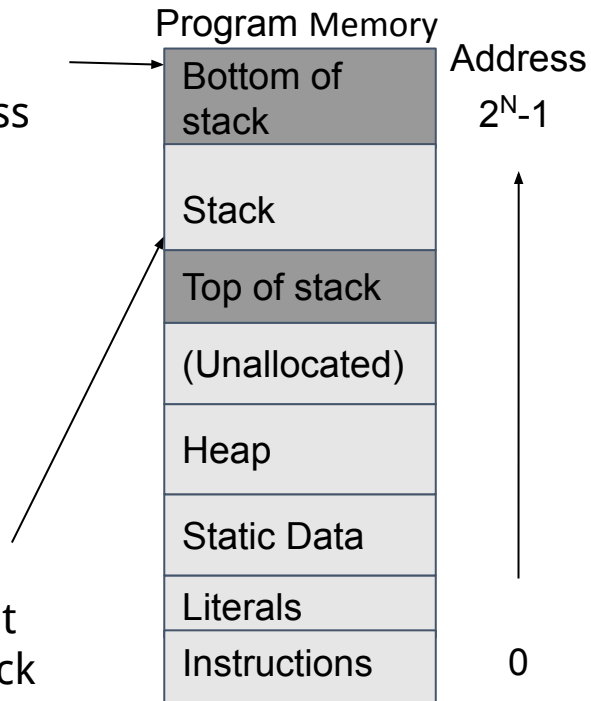


# x86-64 stack | PUSHQ Example

- 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

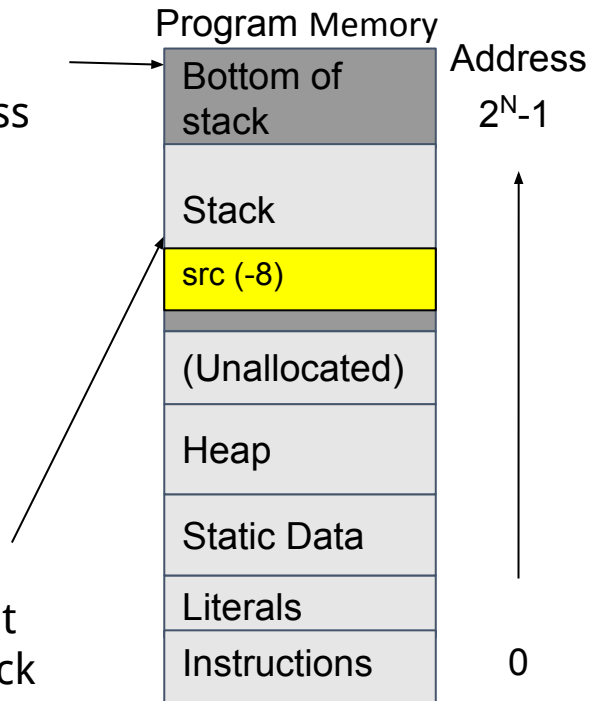


# x86-64 stack | PUSHQ Example

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

**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

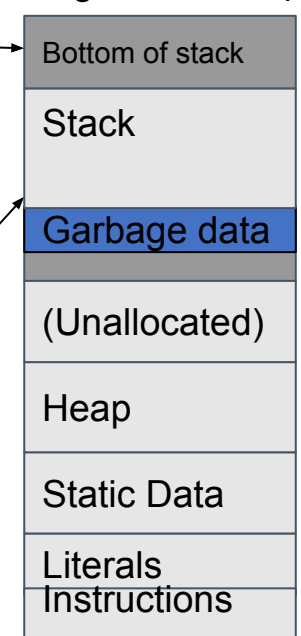
**Base Pointer: %rbp**

Always contains address  
of top of current stack  
frame

**Stack Pointer: %rsp**

Always contains lowest  
address

Program Memory



Address

$2^N - 1$

0

# C/C++ Function in x86

What information do we need to call a function at runtime? Where are they stored?

- Code
- Parameters
- Return value
- Global variables
- Local variables
- Temporary variables
- Return address
- *Function frame pointer*
- *Previous function Frame pointer*

# Global and Local Variables in C/C++

Variables that are declared inside a function or block are called **local variables**. They can be used only by statements that are inside that function or block of code. Local variables are not known to functions outside their own.

**Global variables** are defined outside a function. Global variables hold their values throughout the lifetime of your program and they can be accessed inside any of the functions defined for the program.

In the definition of function parameters which are called **formal parameters**. Formal parameters are similar to local variables.

# Global and Local Variables (misc/globallocalv)

```
char g_i[] = "I am an initialized global variable\n";
```

```
char* g_u;
```

```
int func(int p)
```

```
{
```

```
    int l_i = 10;
```

```
    int l_u;
```

```
    printf("l_i in func() is at %p\n", &l_i);
```

```
    printf("l_u in func() is at %p\n", &l_u);
```

```
    printf("p in func() is at %p\n", &p);
```

```
    return 0;
```

```
}
```

```
int main(int argc, char *argv[])
```

```
{
```

```
    int l_i = 10;
```

```
    int l_u;
```

```
    printf("g_i is at %p\n", &g_i);
```

```
    printf("g_u is at %p\n", &g_u);
```

```
    printf("l_i in main() is at %p\n", &l_i);
```

```
    printf("l_u in main() is at %p\n", &l_u);
```

```
    func(10);
```

```
}
```

Tools: readelf; nm

# Global and Local Variables (misc/globallocalv 32bit)

```
ziming@ziming-ThinkPad:~/Dropbox/my
g_i is at 0x56558020
g_u is at 0x5655804c
l_i in main() is at 0xffff7c6d4
l_u in main() is at 0xffff7c6d8
l_i in func() is at 0xffff7c6a4
l_u in func() is at 0xffff7c6a8
p in func() is at 0xffff7c6c0
```

# Global and Local Variables (misc/globallocalv 64bit)

```
→ globallocalv ./main64  
g_i is at 0x55c30d676020  
g_u is at 0x55c30d676050  
l_i in main() is at 0x7ffcd74866dc  
l_u in main() is at 0x7ffcd74866d8  
l_i in func() is at 0x7ffcd74866ac  
l_u in func() is at 0x7ffcd74866a8  
p in func() is at 0x7ffcd748669c
```

# C/C++ Function in x86/64

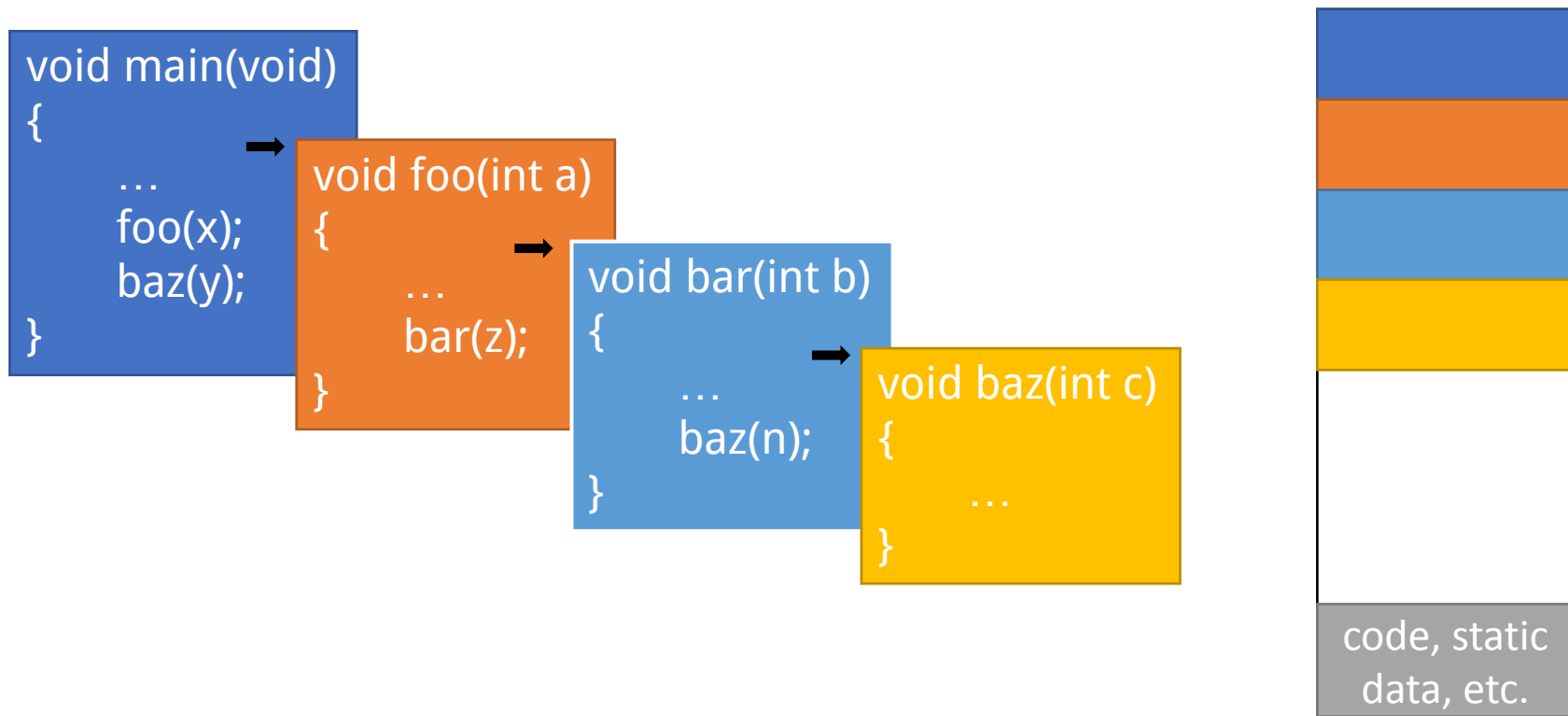
What information do we need to call a function at runtime? Where are they stored?

- Code [.text]
- Parameters [mainly stack (32bit); registers + stack (64bit)]
- Return value [eax, rax]
- Global variables [.bss, .data]
- Local variables [stack; registers]
- Temporary variables [stack; registers]
- Return address [stack]
- Function frame pointer [ebp, rbp]
- Previous function Frame pointer [stack]

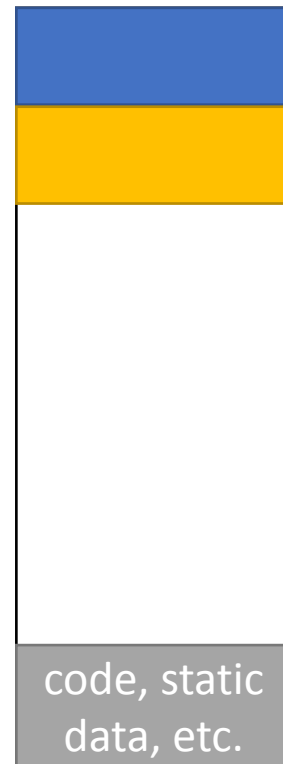
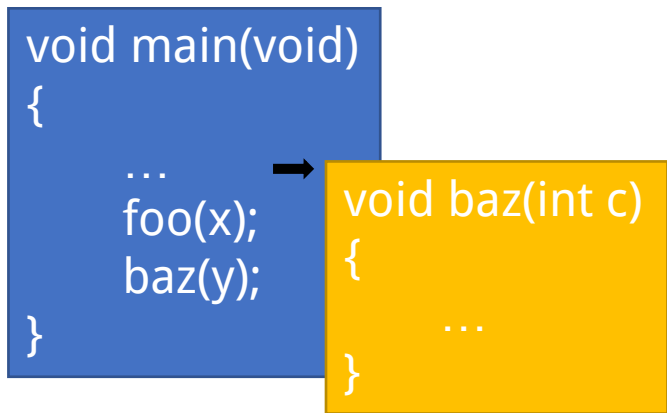
# The Process Stack

- 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

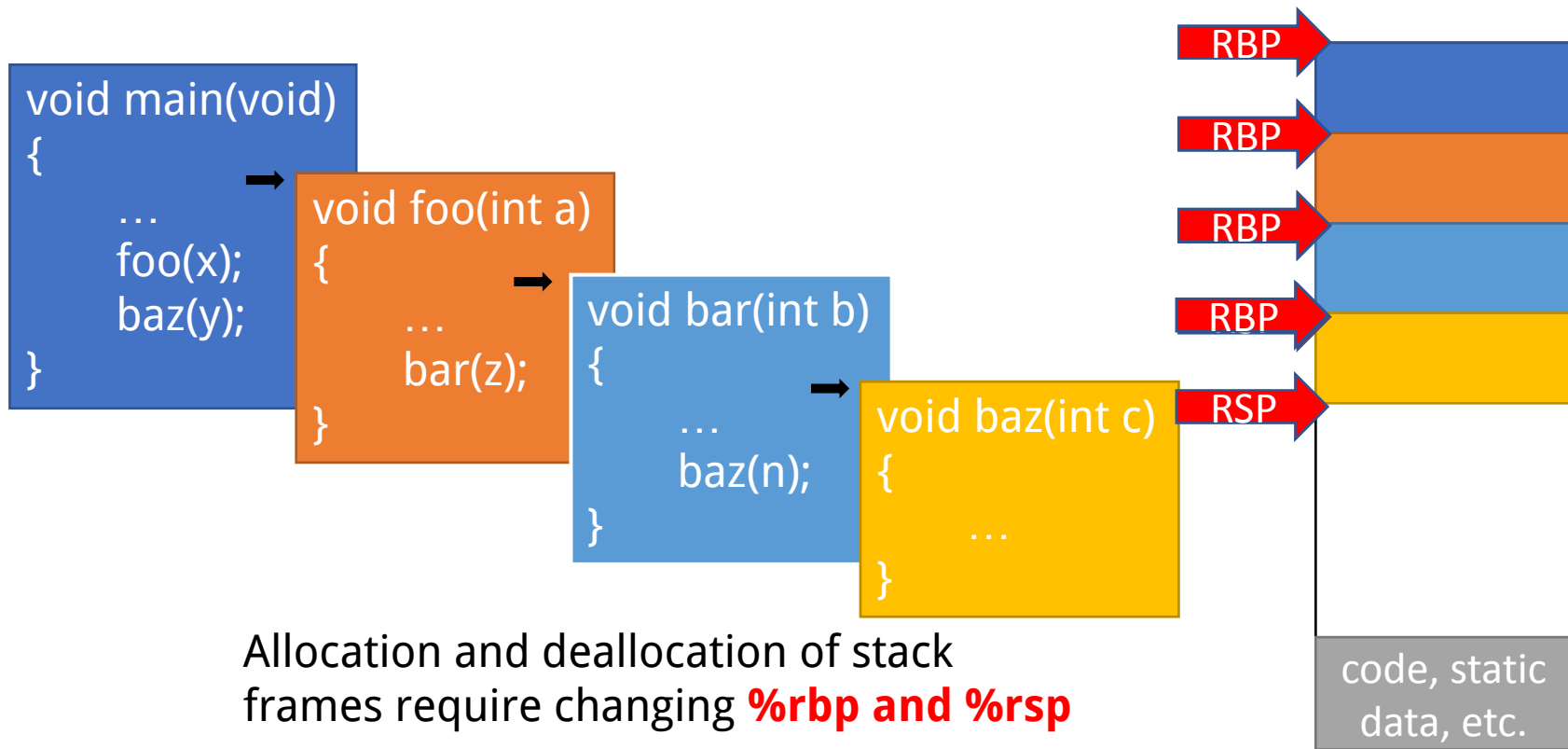
# Creating and deleting stack frames for a function



# Creating and deleting stack frames for a function



# Creating and deleting stack frames for a function

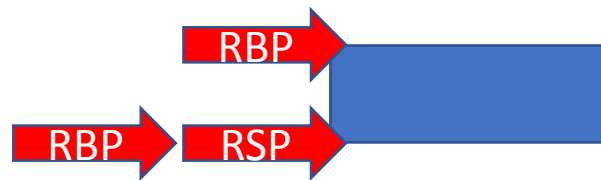


# Creating a new stack frame for a function and exiting

## 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

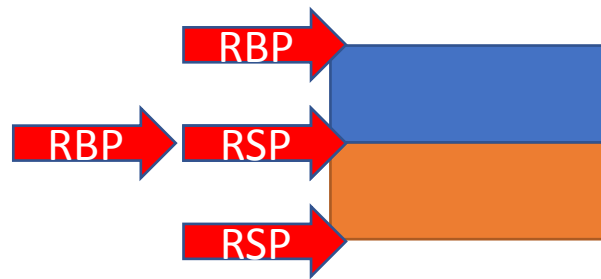
# enter **creates a stack frame**

```
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

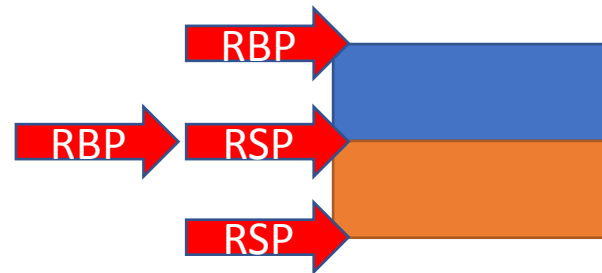
# leave **exits a stack frame: does the inverse of enter**

```
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
```



# x86 (32 bit) Linux Calling Convention (cdecl)

Caller (in this order)

- Pushes arguments onto the stack (in right to left order)
- Execute the **call** instruction (pushes address of instruction after call, then moves dest to **eip**)

Callee

- Pushes previous frame pointer onto stack (ebp)
- Setup new frame pointer (mov ebp, esp)
- Creates space on stack for local variables (sub esp, #imm)
- Ensures that stack is consistent on return
- Return value in **eax** register

# Callee Allocate a stack (Function prologue) 32-bit

Three instructions:

**push ebp;** (Pushes previous frame pointer onto stack)

**mov ebp, esp;** (change the base pointer to the stack)

**sub esp, 10;** (allocating a local stack space)

# **Callee Deallocate a stack (Function epilogue) 32-bit**

**mov esp, ebp**

**pop ebp**

**ret**

# Global and Local Variables (misc/globallocalv)

```
int func(int p)
{
    int l_i = 10;
    int l_u;

    printf("l_i in func() is at %p\n", &l_i);
    printf("l_u in func() is at %p\n", &l_u);
    printf("p in func() is at %p\n", &p);
    return 0;
}
```

Function main()

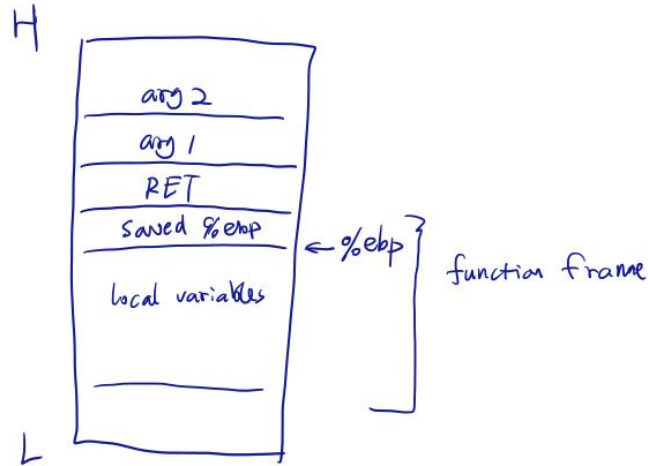
```
657: 83 ec 0c      sub    esp,0xc
65a: 6a 0a        push   0xa
65c: e8 3c ff ff   call   59d <func>
661: 83 c4 10      add    esp,0x10
```

Function func()

```
59d: 55           push   ebp
59e: 89 e5        mov    ebp,esp
5a0: 83 ec 18     sub    esp,0x18
5a3: c7 45 f4 0a 00 00 00 mov    DWORD PTR [ebp-0xc],0xa
5aa: 83 ec 08     sub    esp,0x8
5ad: 8d 45 f4     lea    eax,[ebp-0xc]
5b0: 50           push   eax
5b1: 68 00 07 00 00 push   0x700
5b6: e8 fc ff ff   call   5b7 <func+0x1a>
5bb: 83 c4 10     add    esp,0x10
5be: 83 ec 08     sub    esp,0x8
5c1: 8d 45 f0     lea    eax,[ebp-0x10]
5c4: 50           push   eax
5c5: 68 18 07 00 00 push   0x718
5ca: e8 fc ff ff   call   5cb <func+0x2e>
5cf: 83 c4 10     add    esp,0x10
5d2: 83 ec 08     sub    esp,0x8
5d5: 8d 45 08     lea    eax,[ebp+0x8]
5d8: 50           push   eax
5d9: 68 30 07 00 00 push   0x730
5de: e8 fc ff ff   call   5df <func+0x42>
5e3: 83 c4 10     add    esp,0x10
5e6: b8 00 00 00 00 mov    eax,0x0
5eb: c9           leave
5ec: c3           ret
```

# Draw the stack (x86 cdecl)

x86, cdecl in a function



(`%ebp`) : saved `%ebp`

4(`%ebp`) : RET

8(`%ebp`) : first argument

-8(`%ebp`) : maybe a local variable

# Stack example: misc/fiveParameters\_64

```
int func(int a, int b, int c, int d, int e)
{
    return a + b + c + d + e;
}

int main(int argc, char *argv[])
{
    func(1, 2, 3, 4, 5);
}
```

X86-64 disassembly

# 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:
    ...

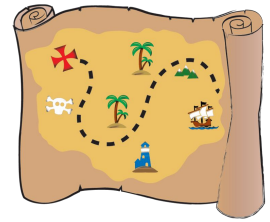
# long factorial(long x)
factorial:
    ...
```

## 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
```

## 4. Function Skeleton

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

### Prologue:

- `push` callee-saves
- `enter` - allocate stack space
  - stack alignment!

### Epilogue:

- `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

```
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

```
// 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);
```

Variables:

- x is -8(%rbp)

```
loop:  
    add $1, -8(%rbp)  
  
    cmp $10, -8(%rbp)  
    # reversed for cmp arg order  
  
    jl loop  
    # sense not reversed  
  
    # ...
```

# While loops

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

Variables:

- x is -8(%rbp)



# While loops

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

Variables:

- x is -8(%rbp)

```
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:  
    ...
```

# **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...

```
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)
```

```
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
```

```
long fact(long n) {  
    if (n ≤ 1) {  
        return 1;  
    }  
  
    return n * fact(n - 1);  
}
```

fact(3)

code, static  
data, etc.

# 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);
}
```

fact(3)

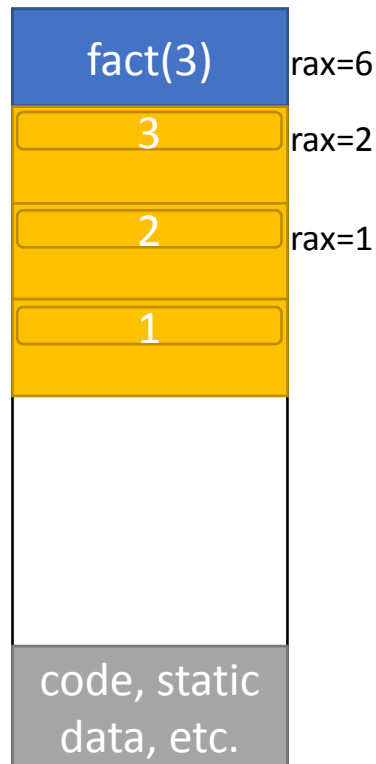
code, static  
data, etc.

# 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);
}
```



# Stack example: misc/factorial

```
int fact(int n)
{
    printf("---In fact(%d)\n", n);
    printf("&n is %p\n", &n);

    if (n <= 1)
        return 1;

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

```
int main(int argc, char *argv[])
{
    if (argc != 2)
    {
        printf("Usage: fact integer\n");
        return 0;
    }

    printf("The factorial of %d is %d\n.",
        atoi(argv[1]), fact(atoi(argv[1])));
}
```