



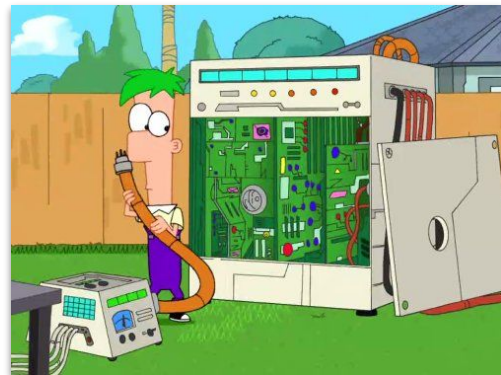
CS107 Lab 5 – Welcome Back!



Assembly, Machine-Level Code

Ben Yan, Winter 2026 ❄️

Slides Link: stanford.edu/~bbyan/cs107 :)



🎵 My Song of the Week: Whiplash by Aespa

Stanford | ENGINEERING
Computer Science

Welcome Back to Lab!



Welcome Back to Lab!

*Hope that the midterm went well for you!
In any case, we still have much ahead 😊*



Winter



Week6!



Spring



We have office hours each week in CoDa :) Feel free to stop by, get coffee, or get started on Assign4 in a shared study space!

Getting Setup for Lab 5



- ❑ To get started, log into the **myth machines**, and clone the lab starter code using the command:

```
git clone /afs/ir/class/cs107/repos/lab5/shared lab5
```

- ❑ Next, also open the **online lab check-off** linked below (or in the lab 1 handout), so you can write as you go.

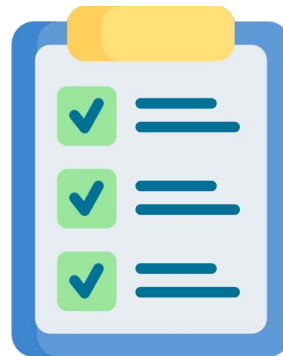


[Lab Handout](#)



[Lab Check-Off Form](#)

Don't worry at all if you fall behind on the write up! I'll aim to go over everything, though just see me after lab if anything is still left



Lecture Recap: Demystifying Assembly!



~15 minutes

Welcome to the C multiverse

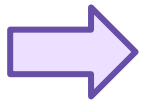




Assembly and Registers

The **compiler** converts our **C code** into low-level **assembly instructions**, which can then be converted into machine code (0s and 1s) run by your computer.

```
int gold = 10;  
int treasure = 5;  
gold += treasure;
```



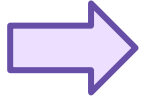
```
mov $10, %rax  
mov $5, %rbx  
addl %rbx, %rax
```



Assembly and Registers

The **compiler** converts our **C code** into low-level **assembly instructions**, which can then be converted into machine code (0s and 1s) run by your computer.

```
int gold = 10;  
int treasure = 5;  
gold += treasure;
```



```
mov $10, %rax  
mov $5, %rbx  
addl %rbx, %rax
```



Think in terms of **registers**: variable boxes used as temporary storage / **scratch paper** for programs.



A register is an **very fast read/write slot** right on the CPU (not in memory), which can hold variable values

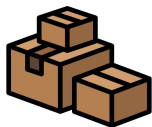


Move data in & out of registers to perform operations on it with the CPU, e.g., arithmetic operations, parameters into functions, return values



0x107 314

RAM/Memory

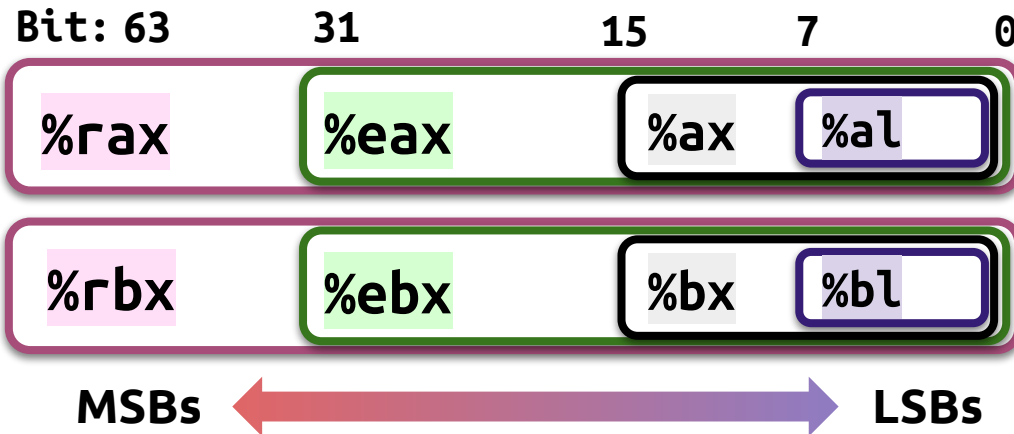


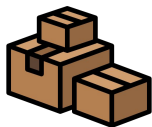
Register Sizes and Sub-registers

Registers can be **partially or fully referred to**. For instance, we can access just 32 bits of a register (e.g., `$edi`) instead of the full 64 bits (e.g., `$rdi`).

Specifically, a reference to the **e-named register** accesses the **lower 32-bit subregister**

`%edi` is subregister of `%rdi`
`%eax` is subregister of `%rax`



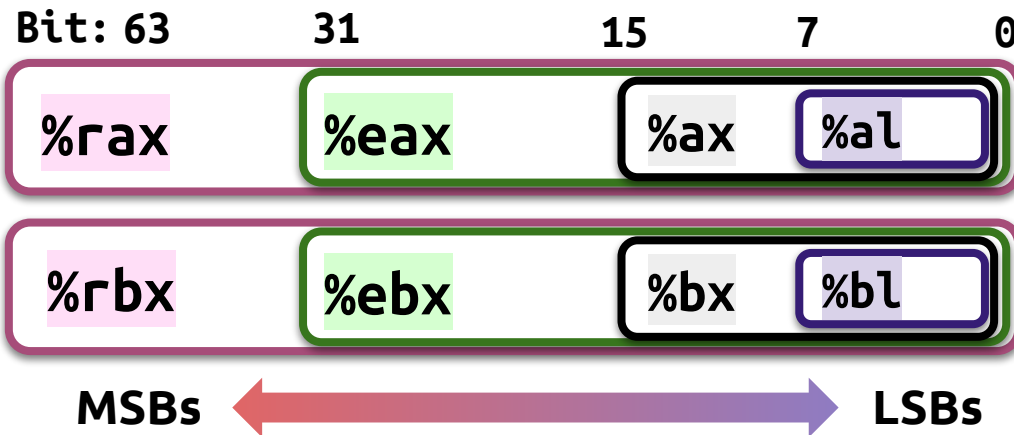


Register Sizes and Sub-registers

Registers can be **partially or fully referred to**. For instance, we can access just 32 bits of a register (e.g., `$edi`) instead of the full 64 bits (e.g., `$rdi`).

Specifically, a reference to the **e-named register** accesses the **lower 32-bit subregister**

`%edi` is subregister of `%rdi`
`%eax` is subregister of `%rax`



When do you want 32 bits (4 bytes) versus 64 bits (8 bytes)?

Tl;dr if you don't need the extra bits. For instance, operations on **pointers or longs** (64 bits) typically use the **full registers**, while operations on **ints** use the **e-named registers**.



Registers: With Great Power Comes Great Responsibility

Many registers are given designated roles or responsibilities, e.g., for function calls:

- ❑ The first 4 arguments passed in are **\$rdi**, **\$rsi**, **\$rdx**, **\$rcx**, respectively
- ❑ The return value of a function (if any) is written to **\$rax** before it ends

Register	Conventional use	Low 32-bits	Low 16-bits	Low 8-bits
%rax	Return value, callee-owned	%eax	%ax	%al
%rdi	1st argument, callee-owned	%edi	%di	%dil
%rsi	2nd argument, callee-owned	%esi	%si	%sil
%rdx	3rd argument, callee-owned	%edx	%dx	%dl
%rcx	4th argument, callee-owned	%ecx	%cx	%cl
%r8	5th argument, callee-owned	%r8d	%r8w	%r8b
%r9	6th argument, callee-owned	%r9d	%r9w	%r9b
%r10	Scratch/temporary, callee-owned	%r10d	%r10w	%r10b
%r11	Scratch/temporary, callee-owned	%r11d	%r11w	%r11b
%rsp	Stack pointer, caller-owned	%esp	%sp	%spl



*Full list can be found on
course's x86-64 guide!*



Assembly: Addressing Modes

Immediate: `mov $0x5, dst`

A **dollar sign (\$)** indicates source is **constant value** to be moved into dst

Direct: `mov 0x106b, dst`

Fixed, static memory address

This reads from **Mem[0x106b]**, the memory at address 0x106b.

Indirect with Displacement:
`mov 8($rax), dst`

Same as indirect, but has a **number prefix as bytes plus the address**, like pointer arithmetic!

The above reads from **Mem[\$rax + 8]**

Register: `mov $rax, dst`

Moves **value held in register** to dst.

Indirect: `mov ($rax), dst`

This reads from **Mem[\$rax]**, where the register \$rax holds an address, e.g., 0x106b

Note **parentheses** to contrast with above.

Indirect with Displacement + Scaled Index: `mov 8($rsp, $rcx, 4), dst`

More complex displacement: **offset + scaling factor** (e.g., 1,2,4,8), often to index into arrays.

Reads from **Mem[\$rsp + 8 + (4 * \$rcx)]**.



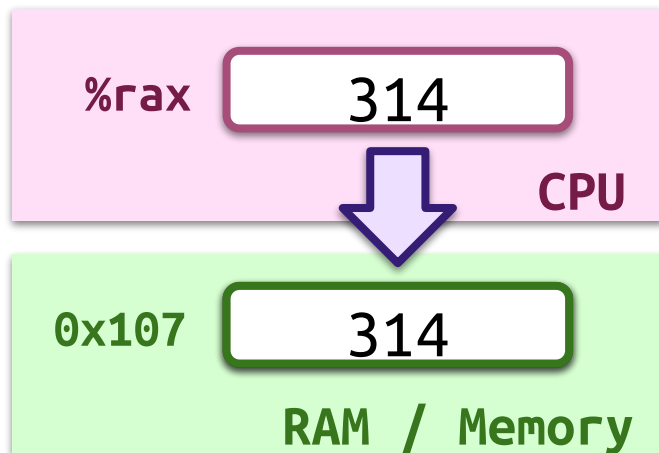
mov instruction

“i like to mov it mov it” – *King Julien*

Key Instruction: mov src, dest

- ❑ Responsible for **copying data** around!
- ❑ **src** can be an immediate (constant value), register, or memory location
- ❑ **dst** can be a register or memory location
- ❑ **Exception: src and dst cannot both be memory locations!** (must first temporarily copy the data into a register)

e.g., `mov %rax, 0x107`



Instructions like `mov` can **specify via a suffix the # of bytes** being moved, etc.

- ❑ **movb** for 1 byte being copied
- ❑ **movl** for “double word” (4 bytes)
- ❑ **movw** for “word” (2 bytes)
- ❑ **movq** for “quad word” (8 bytes)



Just the cursed technique I use! Feel free to use what works best for you :)

Ben's 'Waterfall Method' of Reading Assembly

General Guidelines:

(may have to adapt to specific assembly task!)

- ❑ 🏹 Write **each register value** (or corresponding variable) **in a separate, labeled box**.
- ❑ 🍂 Assembly is poetry, flowing top to bottom – **for each line, write a new box lower than the previous one and indented to the right**, with the updated value (don't cross out old boxes!).
- ❑ 📖 Treat function calls as a stanza break – **write each function call explicitly, and draw arrows from the registers being passed in**. Write a box for return `$rax` after, and reset indentation.



Ben's 'Waterfall Method' of Reading Assembly

General Guidelines:

(may have to adapt to specific assembly task!)

- ❑ Write **each register value** (or corresponding variable) **in a separate, labeled box**.
- ❑ Assembly is poetry, flowing top to bottom – **for each line, write a new box lower than the previous one and indented to the right**, with the updated value (don't cross out old boxes!).
- ❑ Treat function calls as a stanza break – **write each function call explicitly, and draw arrows from the registers being passed in**. Write a box for return **\$rax** after, and reset indentation.

```
  is_not_poohbear(user):  
<+0> mov $rdi, $rsi  
<+4> mov $0x106a106b, $rdi  
<+8> mov $0x5, $rdx  
<+12> call strncmp  
<+16> ret
```

Small toy example! Assume `$0x106a106b` corresponds to the string "Jerry Cain".



Ben's 'Waterfall Method' of Reading Assembly

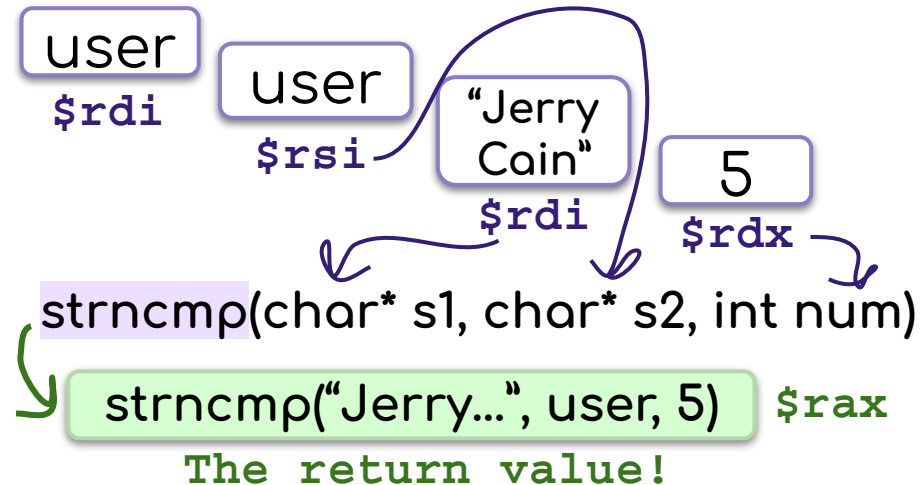
General Guidelines:

(may have to adapt to specific assembly task!)

- ❑ 🚩 Write **each register value** (or corresponding variable) in a **separate, labeled box**.
- ❑ 🍂 Assembly is poetry, flowing top to bottom – **for each line, write a new box lower than the previous one and indented to the right**, with the updated value (don't cross out old boxes!).
- ❑ 📖 Treat function calls as a stanza break – **write each function call explicitly, and draw arrows from the registers being passed in**. Write a box for return **\$rax** after, and reset indentation.

```
🐻🚀 is_not_poohbear(user):  
<+0> mov $rdi, $rsi  
<+4> mov $0x106a106b, $rdi  
<+8> mov $0x5, $rdx  
<+12> call strncmp  
<+16> ret
```

Small toy example! Assume `$0x106a106b` corresponds to the string "Jerry Cain".





Ben's 'Waterfall Method' of Reading Assembly

General Guidelines:

- ❑ 🚀 Write **each register value** (or corresponding variable) in a **separate, labeled box**.
- ❑ Assembly is poetry, flowing top to bottom – **for each line, write a new box lower than the previous one and indented to the right**, with the updated value (don't cross out old boxes!).
- ❑ Treat function calls as a stanza break – **write each function call explicitly, and draw arrows from the registers being passed in**. Write a box for return `$rax` after, and reset indentation.



`is_not_poohbear(user):`

```
<+0> mov $rdi, $rsi
<+4> mov $0x106a106b, $rdi
<+8> mov $0x5, $rdx
<+12> call strncmp
<+16> ret
```

user
\$rdi

Small toy example! Assume `$0x106a106b` corresponds to the string "Jerry Cain".



Ben's 'Waterfall Method' of Reading Assembly

General Guidelines:

- ❑ 🚀 Write **each register value** (or corresponding variable) in a **separate, labeled box**.
- ❑ 🍂 Assembly is poetry, flowing top to bottom – **for each line, write a new box lower than the previous one and indented to the right**, with the updated value (don't cross out old boxes!).
- ❑ Treat function calls as a stanza break – **write each function call explicitly, and draw arrows from the registers being passed in**. Write a box for return `$rax` after, and reset indentation.

```
🐻🚀 is_not_poohbear(user):  
<+0> mov $rdi, $rsi  
<+4> mov $0x106a106b, $rdi  
<+8> mov $0x5, $rdx  
<+12> call strncmp  
<+16> ret
```

user
\$rdi

user
\$rsi

Small toy example! Assume `$0x106a106b` corresponds to the string "Jerry Cain".



Ben's 'Waterfall Method' of Reading Assembly

General Guidelines:

- ❑ 🏹 Write **each register value** (or corresponding variable) in a **separate, labeled box**.
- ❑ 🍂 Assembly is poetry, flowing top to bottom – **for each line, write a new box lower than the previous one and indented to the right**, with the updated value (don't cross out old boxes!).
- ❑ Treat function calls as a stanza break – **write each function call explicitly, and draw arrows from the registers being passed in**. Write a box for return `$rax` after, and reset indentation.

```
🐻🚀 is_not_poohbear(user):  
  <+0> mov $rdi, $rsi  
  <+4> mov $0x106a106b, $rdi  
  <+8> mov $0x5, $rdx  
  <+12> call strncmp  
  <+16> ret
```

user
\$rdi

user
\$rsi

"Jerry
Cain"
\$rdi

Small toy example! Assume `$0x106a106b` corresponds to the string "Jerry Cain".



Ben's 'Waterfall Method' of Reading Assembly

General Guidelines:

- ❑ 🚩 Write **each register value** (or corresponding variable) in a **separate, labeled box**.
- ❑ 🍂 Assembly is poetry, flowing top to bottom – **for each line, write a new box lower than the previous one and indented to the right**, with the updated value (don't cross out old boxes!).
- ❑ 📄 Treat function calls as a stanza break – **write each function call explicitly, and draw arrows from the registers being passed in**. Write a box for return `$rax` after, and reset indentation.

```
🐻🚀 is_not_poohbear(user):  
  <+0> mov $rdi, $rsi  
  <+4> mov $0x106a106b, $rdi  
  <+8> mov $0x5, $rdx  
  <+12> call strncmp  
  <+16> ret
```

user
\$rdi

user
\$rsi

"Jerry
Cain"
\$rdi

5
\$rdx

Small toy example! Assume `$0x106a106b` corresponds to the string "Jerry Cain".

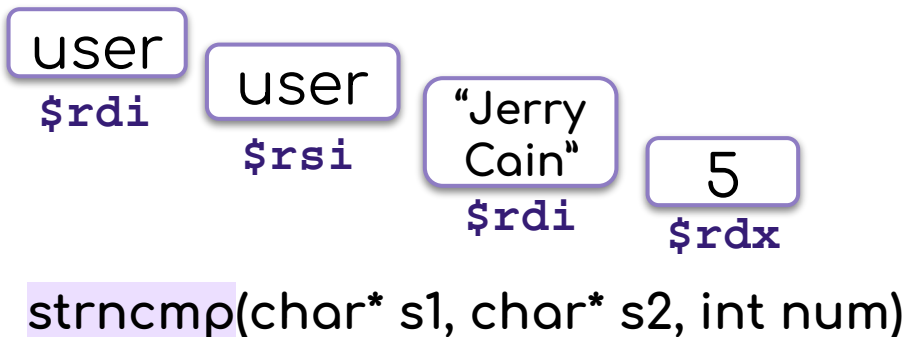


Ben's 'Waterfall Method' of Reading Assembly

General Guidelines:

- ❏ 🚀 Write **each register value** (or corresponding variable) in a **separate, labeled box**.
- ❏ Assembly is poetry, flowing top to bottom – **for each line, write a new box lower than the previous one and indented to the right**, with the updated value (don't cross out old boxes!).
- ❏ 📖 Treat function calls as a stanza break – **write each function call explicitly, and draw arrows from the registers being passed in**. Write a box for return **\$rax** after, and reset indentation.

```
🐻🚀 is_not_poohbear(user) :  
  <+0> mov $rdi, $rsi  
  <+4> mov $0x106a106b, $rdi  
  <+8> mov $0x5, $rdx  
  <+12> call strncmp  
  <+16> ret
```



Small toy example! Assume **\$0x106a106b** corresponds to the string **"Jerry Cain"**.



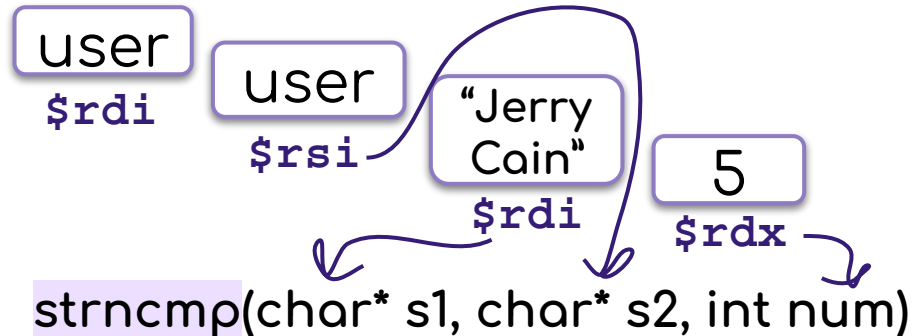
Ben's 'Waterfall Method' of Reading Assembly

General Guidelines:

- ❏ 🚀 Write **each register value** (or corresponding variable) in a **separate, labeled box**.
- ❏ Assembly is poetry, flowing top to bottom – **for each line, write a new box lower than the previous one and indented to the right**, with the updated value (don't cross out old boxes!).
- ❏ 📖 Treat function calls as a stanza break – **write each function call explicitly, and draw arrows from the registers being passed in**. Write a box for return **\$rax** after, and reset indentation.

```
🐻🚀 is_not_poohbear(user):  
  <+0> mov $rdi, $rsi  
  <+4> mov $0x106a106b, $rdi  
  <+8> mov $0x5, $rdx  
  <+12> call strncmp  
  <+16> ret
```

Small toy example! Assume `$0x106a106b` corresponds to the string "Jerry Cain".



First argument is register `$rdi`, then `$rsi`, `$rdx`, `$rcx`, `$r8`, `$r9`



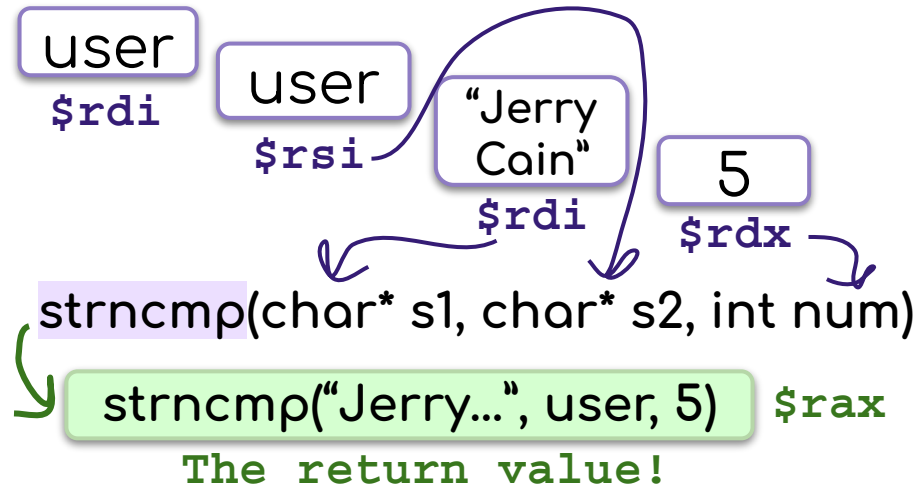
Ben's 'Waterfall Method' of Reading Assembly

General Guidelines:

- ❏ 🚀 Write **each register value** (or corresponding variable) in a **separate, labeled box**.
- ❏ Assembly is poetry, flowing top to bottom – **for each line, write a new box lower than the previous one and indented to the right**, with the updated value (don't cross out old boxes!).
- ❏ 📖 Treat function calls as a stanza break – **write each function call explicitly, and draw arrows from the registers being passed in**. Write a box for return **\$rax** after, and reset indentation.

```
🐻🚀 is_not_poohbear(user):  
  <+0> mov $rdi, $rsi  
  <+4> mov $0x106a106b, $rdi  
  <+8> mov $0x5, $rdx  
  <+12> call strncmp  
  <+16> ret
```

Small toy example! Assume `$0x106a106b` corresponds to the string "Jerry Cain".





(Includes Jerry's own pointers toward eventual task of C reverse-engineering!)

General Assembly Reading Tips! YOU CAN DO IT

Try to understand what the assembly is doing certainly, but be more concerned with mapping your understanding to **equivalent C code**, and not necessarily the original.



General Assembly Reading Tips! YOU CAN DO IT

Try to understand what the assembly is doing certainly, but be more concerned with mapping your understanding to **equivalent C code**, and not necessarily the original.

Look for registers that alternate between being updated and being read (**think very active, lively registers**), because they typically correspond to key **local variables**.



General Assembly Reading Tips! YOU CAN DO IT

Try to understand what the assembly is doing certainly, but be more concerned with mapping your understanding to **equivalent C code**, and not necessarily the original.

Look for registers that alternate between being updated and being read (**think very active, lively registers**), because they typically correspond to key **local variables**.

“ If it looks like a **char***, swims like a **char***, and quacks like a **char***, it’s probably a **char*** ”



General Assembly Reading Tips! YOU CAN DO IT

Try to understand what the assembly is doing certainly, but be more concerned with mapping your understanding to **equivalent C code**, and not necessarily the original.

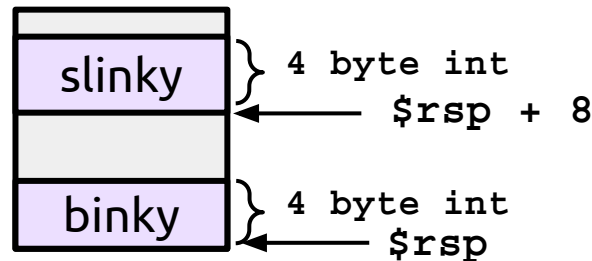
Look for registers that alternate between being updated and being read (**think very active, lively registers**), because they typically correspond to key **local variables**.

“ If it looks like a `char*`, swims like a `char*`, and quacks like a `char*`, it’s probably a `char*` ”



Keep a record / sketch of the stack frame.

And when portions of it are written to or updated (e.g., a 4-byte sliver at address `$rsp` to store an integer there), **update your sketch accordingly!**





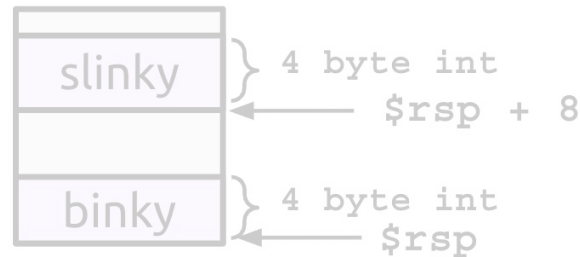
General Assembly Reading Tips! YOU CAN DO IT

Try to understand what the assembly is doing certainly, but be more concerned with mapping your understanding to **equivalent C code**, and not necessarily the original.

Look for registers that alternate between being updated and being read (**think very active, lively registers**), because they typically correspond to key **local variables**.

“ If it looks like a `char*`, swims like a `char*`, and quacks like a `char*`, it’s probably a `char*` ”

Keep a record / sketch of the stack frame. And when portions of it are written to or updated (e.g., a 4-byte sliver at address `$rsp` to store an integer there), **update your sketch accordingly!**



 **Decomposition!** If you encounter a long af stream of assembly, use function calls to **segment the instructions into smaller, more manageable sections**. And when you see a loop 🌀, first aim to divide it into 3 or so parts: a **pre-loop, within-loop, and post-loop**.



(includes Jerry's own pointers toward eventual task of C reverse-engineering!)

General Assembly Reading Tips! YOU CAN DO IT

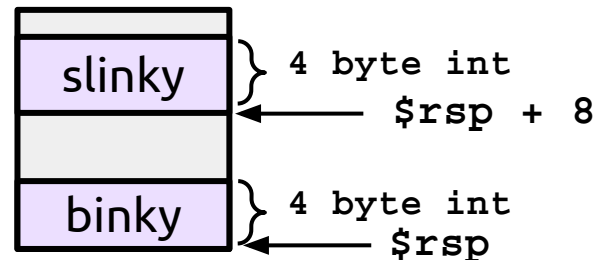
Try to understand what the assembly is doing certainly, but be more concerned with mapping your understanding to **equivalent C code**, and not necessarily the original.

Look for registers that alternate between being updated and being read (**think very active, lively registers**), because they typically correspond to key **local variables**.

“ If it looks like a **char***, swims like a **char***, and quacks like a **char***, it's probably a **char*** ”

👉 Assembly operates directly on raw data/bits, but **context can help in determining type!**

📖 **Keep a record / sketch of the stack frame.** And when portions of it are written to or updated (e.g., a 4-byte sliver at address `$rsp` to store an integer variable there), **update your sketch accordingly!**



🧩 **Decomposition!** If you encounter a long af stream of assembly, use function calls to **segment the instructions into smaller, more manageable sections**. And when you see a loop 🌀, first aim to divide it into 3 or so parts: a **pre-loop, within-loop, and post-loop**.



(Includes Jerry's own pointers toward eventual task of C reverse-engineering!)

Additional Assembly Reading Tips

You can usually ignore all push and pop operations when reading, as they're almost always boilerplate in place to save & ultimately restore caller-owner register values.

When you suspect there's an **if test** or an **if/else test**:

- Code branches off, so look for where/how it branches, i.e. any  **conditional jumps** and **unconditional jumps** to higher addresses
- Use that to demarcate what is part of the **if test**, the **if body**, and **potentially the body of a corresponding else branch** .

check condition

Jump to [IfBody] if condition true

[Else]:

<If false statements>

Jump to [EndIf]

[IfBody]:

<If true statements>

[EndIf]



More on for/while loops: When you see one, **decompose it!**

- Look for the  **conditional jump forward** and the **unconditional jump backwards** (can be the reverse! i.e. conditional jump back)
- Above case: **backwards jump keeps us in loop**, **forward jump exits**
- Use this to demarcate which instructions are **within the body of the loop**, **which ones precede it**, and **which ones come after it**.

[Initialize] (e.g., int i = 0)

[Test]:

Check **OPPOSITE** of loop condition

Jump to [LoopEnd] if true

[LoopBody]:

<statements>

<Update> (e.g., i++)

Jump to [Test]

[LoopEnd]:

everything else

Happy to discuss any questions!

Reading and interpreting assembly, relevant resources, etc.





New GDB Commands for Assembly!



disassemble: prints the assembly for the currently executing function (can write *(gdb) disassemble func_name* to disassemble another function)



b *address: Places a breakpoint on a specific assembly instruction, via its address



stepi, nexti: Assembly versions of `step`, `next`, going one line of assembly at a time



info reg: Prints out the values in all integer registers



print \$register: Prints out a specific register's contents, e.g., `print $rax`

Note: Can print using a typecast, e.g., `(gdb) print (char*) ptr`, or in hex, via `print /x`

```
(gdb) p (char *)$rax
$3 = 0x400720 "Hello, world!\n"
```

```
(gdb) p *(long *)$rax
(gdb) p *(long *)((char *)$rsp + 8)
```



set \$register = value: manually set a register's contents, useful for testing behavior

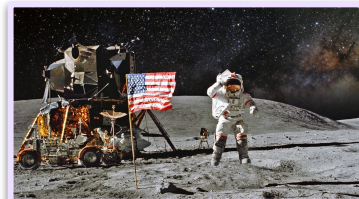
In Terminal / myth, can run **objdump -d executable_name** to list all assembly instructions

Check-Off Question

What's the difference between the gdb **step** and **stepi** commands?

- ☐ **step** executes one line of C code
- ☐ **stepi** executes one line of assembly **instruction**.
- ☐ A single line of C code can have multiple assembly instructions!

“One small step for C, one giant leap for assembly”
– Neil Armstrong, I think



Comment: Note that, by comparison, **nexti** will step over function calls, while **stepi** will jump into the assembly instructions of the function being called



Some Nifty References For Assembly

CS107 x86-64 Reference Sheet

Common instructions		
mov src, dst	dst = src	push src add to top of stack
movsbl src, dst	byte to int, sign-extend	Mem[−%rsp] = src
movzbl src, dst	byte to int, zero-fill	pop dst remove top from stack
cmov src, reg	reg = src when condition holds, using same condition suffixes as jmp	dst = Mem[%rsp++]
		call fn push %rip, jmp to fn
		ret pop %rip
lea addr, dst	dst = addr	
add src, dst	dst += src	Condition codes/flags
sub src, dst	dst -= src	ZF Zero flag
imul src, dst	dst *= src	SF Sign flag
neg dst	dst = -dst (arith inverse)	CF Carry flag
		OF Overflow flag
imulq S	signed full multiply R[%rdx]:R[%rax] <- S * R[%rax]	Addressing modes
mulq S	unsigned full multiply same effect as imulq	Example source operands to mov
divq S	signed divide R[%rdx] <- R[%rdx]:R[%rax] mod S	Immediate
		mov \$0x5 , dst
		\$val source is constant value



x86-64 reference sheet [\[Link\]](#)

Guide to x86-64

A CS107 joint staff effort (Erik, Julie, Nate)

Registers
Addressing modes
Common instructions
Assembly and gdb

x86-64 (also known as just x64 and/or AMD64) is the 64-bit version of the x86/IA32 instruction set. Below is our overview of its features that are relevant to CS107. There is more extensive coverage on these topics in Chapter 3 of the B&O textbook. See also our [x86-64 sheet](#) for a compact reference.

Registers

The table below lists the commonly used registers (sixteen general-purpose plus two special). Each register is 64 bits wide; the lower 32-, 16- and 8-bit portions are selectable by a pseudo-register name. Some registers are designated for a certain purpose, such as %rsp being used as the stack pointer or %rax for the return value from a function. Other registers are all-purpose, but have a conventional use depending on whether caller-owned or callee-owned. If the function `binky` calls `winky`, we refer to `binky` as the caller and `winky` as the callee. For example, the registers used for the first 6 arguments and return value are all callee-owned. The callee can freely use those registers, overwriting existing values without taking any precautions. If %rax holds a value the caller wants to retain, the caller must copy the value to a "safe" location before making a call. The callee-owned registers are ideal for scratch/temporary use by the callee. In contrast, if the callee intends to use a caller-owned register, it must first preserve its value and restore it before exiting the call. The caller-owned registers are used for local state of the caller that needs to be preserved across further function calls.

Register	Conventional use	Low 32-bits	Low 16-bits	Low 8-bits
%rax	Return value, callee-owned	%eax	%ax	%al
%rdi	1st argument, callee-owned	%edi	%di	%dil



longer assembly guide [\[Link\]](#)



Exercise 1: Assembly and GDB

In your **lab4** directory, look through the **code.c** file, compile it and load into GDB.

Here, you'll practice assembly-level navigation of a program, analyzing how it's compiled into fine-grained instructions.

Please ask me for any questions! And do work with the people around you as well!



~ 15 min + 10 min all-lab discussion

```
benjaminian — bbyan@myth61: ~/CS107_TA/assignments/li
code.c
7          int myfn(long p
B+>8          int arr[] =
9          int count =

B+ 0x401106 <myfn>          mov...
> 0x40110e <myfn+8>        mov...
0x401116 <myfn+16>        mov...

process 30936 In: myfn
(gdb) stepi
(gdb) layout split
```

- 👉 Try **(gdb) layout split** to view assembly, code, GDB all at once!
- 👉 Or **(gdb) asm**, then **(gdb) regs** for just assembly + registers



Let's disassemble: code.c

```
int myfn(long param) {  
    int arr[] = {1, 2, 3, param, 0x4567, INT_MIN,  
                 strlen("Stanford University")};  
    int count = sizeof(arr) / sizeof(arr[0]);  
    int total = arr[param];  
    int squared = total * total;  
    return squared + count;  
}
```

```
<+0>:    movl    $0x1,-0x20(%rsp)  
<+8>:    movl    $0x2,-0x1c(%rsp)  
<+16>:   movl    $0x3,-0x18(%rsp)  
<+24>:   mov     %edi,-0x14(%rsp)  
<+28>:   movl    $0x4567,-0x10(%rsp)  
<+36>:   movl    $0x80000000,-0xc(%rsp)  
<+44>:   movl    $0x13,-0x8(%rsp)
```

mov src, dest

%edi

param

%rsp

stack pointer

%rsp = an address! CPU

%rsp



myfn Stack Memory



Let's disassemble: code.c

```
int myfn(long param) {  
    int arr[] = {1, 2, 3, param, 0x4567, INT_MIN,  
                 strlen("Stanford University")};  
    int count = sizeof(arr) / sizeof(arr[0]);  
    int total = arr[param];  
    int squared = total * total;  
    return squared + count;  
}
```

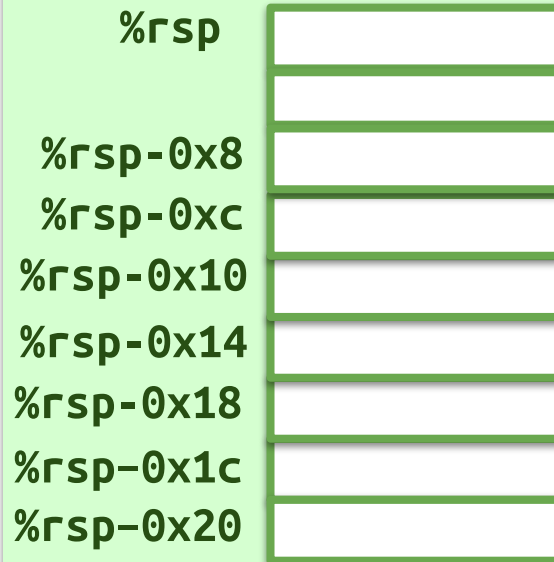
```
<+0>:    movl    $0x1, -0x20(%rsp)  
<+8>:    movl    $0x2, -0x1c(%rsp)  
<+16>:   movl    $0x3, -0x18(%rsp)  
<+24>:   mov     %edi, -0x14(%rsp)  
<+28>:   movl    $0x4567, -0x10(%rsp)  
<+36>:   movl    $0x80000000, -0xc(%rsp)  
<+44>:   movl    $0x13, -0x8(%rsp)
```

mov SRC, dest

%edi param

%rsp stack pointer

%rsp = an address! CPU



myfn Stack Memory



Let's disassemble: code.c

```
int myfn(long param) {  
    int arr[] = {1, 2, 3, param, 0x4567, INT_MIN,  
                 strlen("Stanford University")};  
    int count = sizeof(arr) / sizeof(arr[0]);  
    int total = arr[param];  
    int squared = total * total;  
    return squared + count;  
}
```

```
<+0>:    movl    $0x1, -0x20(%rsp)  
<+8>:    movl    $0x2, -0x1c(%rsp)  
<+16>:   movl    $0x3, -0x18(%rsp)  
<+24>:   mov     %edi, -0x14(%rsp)  
<+28>:   movl    $0x4567, -0x10(%rsp)  
<+36>:   movl    $0x80000000, -0xc(%rsp)  
<+44>:   movl    $0x13, -0x8(%rsp)
```

mov src, dest

%edi param

%rsp stack pointer

%rsp = an address! CPU

%rsp	
%rsp-0x8	13
%rsp-0xc	INT_MIN
%rsp-0x10	0x4567
%rsp-0x14	param
%rsp-0x18	3
%rsp-0x1c	2
%rsp-0x20	1

myfn Stack Memory



Q1: Arrays in Assembly

What expression represents the location of the **first element** in arr? The **last**?

```
<+0>:    movl    $0x1,-0x20(%rsp)
<+8>:    movl    $0x2,-0x1c(%rsp)
<+16>:   movl    $0x3,-0x18(%rsp)
<+24>:   mov     %edi,-0x14(%rsp)
<+28>:   movl    $0x4567,-0x10(%rsp)
<+36>:   movl    $0x80000000,-0xc(%rsp)
<+44>:   movl    $0x13,-0x8(%rsp)
```

📌 **1st element:**

📌 **Last element:**

%edi param

%rsp stack pointer

%rsp = an address! CPU

%rsp	
%rsp-0x8	13
%rsp-0xc	INT_MIN
%rsp-0x10	0x4567
%rsp-0x14	param
%rsp-0x18	3
%rsp-0x1c	2
%rsp-0x20	1

myfn Stack Memory



Q1: Arrays in Assembly

What expression represents the location of the **first element** in arr? The **last**?

```
<+0>:    movl    $0x1,-0x20(%rsp)
<+8>:    movl    $0x2,-0x1c(%rsp)
<+16>:   movl    $0x3,-0x18(%rsp)
<+24>:   mov     %edi,-0x14(%rsp)
<+28>:   movl    $0x4567,-0x10(%rsp)
<+36>:   movl    $0x80000000,-0xc(%rsp)
<+44>:   movl    $0x13,-0x8(%rsp)
```

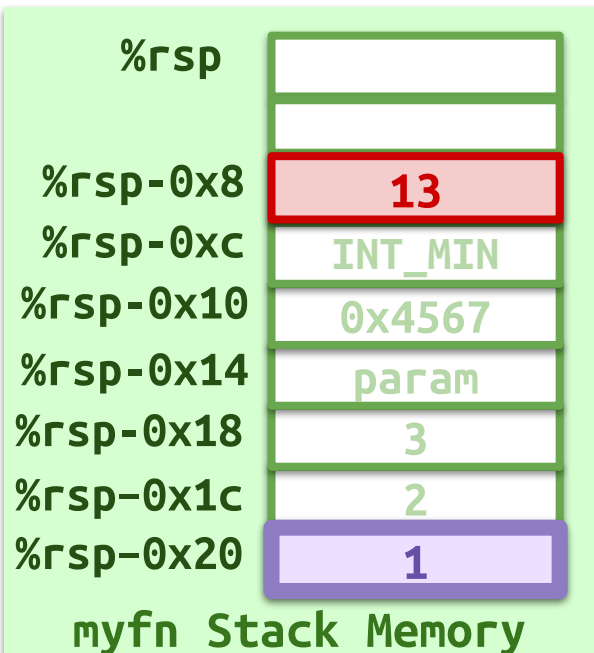
1st element: 1, at address **%rsp - 0x20**

Last element: 13, at address **%rsp - 0x8**

%edi param

%rsp stack pointer

%rsp = an address! CPU





Check-Off: GDB Walkthrough in code

```
code.c
7          int myfn(long param) {
B+ 8          int arr[] = {1, 2, 3, param, 0x4567, 0x1234, 0x9876, 0x5432};
9          int count = sizeof(arr) / sizeof(arr[0]);
10
>11         int total = arr[param];
```

```
B+ 0x401106 <myfn>      movl    $0x1,-0x20(%rsp)
0x40110e <myfn+8>      movl    $0x2,-0x1c(%rsp)
0x401116 <myfn+16>     movl    $0x3,-0x18(%rsp)
0x40111e <myfn+24>     mov     %edi,-0x14(%rsp)
0x401122 <myfn+28>     movl    $0x4567,-0x10(%rsp)
```

```
native process 141161 In: myfn
```

```
(gdb) break myfn
```

```
Breakpoint 1 at 0x401106: file code.c, line 8.
```

```
(gdb) run
```

```
Starting program: /afs/.ir.stanford.edu/users/b/b/bbyar
```

```
Breakpoint 1, myfn (param=param@entry=3) at code.c:8
```

```
(gdb) next
```

```
(gdb) print ((int *)(-0x20 + $rsp))[0]@7
```

```
$1 = {1, 2, 3, 3, 17767, -2147483648, 19}
```

```
(gdb) |
```

Our goal is to print out the array in **myfn** using just register info!



First, start up GDB with
myth: gdb code

Place a breakpoint on **myfn**,
then **run** the program!

Array base is at address **-0x20 + \$rsp** (from previous slide). Next, use the GDB array printing syntax!



(gdb) print ((int *)(-0x20 + \$rsp))[0]@7



Q2: Register Reuse

What register stores the values of **total** and **squared**?

```
int total = arr[param];  
int squared = total * total;  
return squared + count;
```

```
0x40113a <+52>: mov  -0x20(%rsp,%rdi,4),%eax  
0x40113e <+56>: imul %eax,%eax  
0x401141 <+59>: add  $0x7,%eax  
0x401144 <+62>: retq
```

CPU

%edi

param

%rsp

stack pointer

%rsp = an address!

%rsp

%rsp-0x8

13

%rsp-0xc

INT_MIN

%rsp-0x10

0x4567

%rsp-0x14

param

%rsp-0x18

3

%rsp-0x1c

2

%rsp-0x20

1

myfn Stack Memory



Q2: Register Reuse

What register stores the values of **total** and **squared**?

```
int total = arr[param];  
int squared = total * total;  
return squared + count;
```

```
0x40113a <+52>: mov  -0x20(%rsp,%rdi,4),%eax  
0x40113e <+56>: imul %eax,%eax  
0x401141 <+59>: add  $0x7,%eax  
0x401144 <+62>: retq
```

total = (%rsp - 0x20) + 4 * %rdi;

Address of starting element of the array!

Each int is 4 bytes, so multiply by index (param / \$rdi) to get the # of bytes to the desired element!



The value of **total** is then stored in **\$eax**

CPU

%edi

param

%rsp

stack pointer

%rsp = an address!

%eax

total

%rsp

%rsp-0x8

13

%rsp-0xc

INT_MIN

%rsp-0x10

0x4567

%rsp-0x14

param

%rsp-0x18

3

%rsp-0x1c

2

%rsp-0x20

1

myfn Stack Memory



Q2: Register Reuse

What register stores the values of **total** and **squared**?

```
int total = arr[param];  
int squared = total * total;  
return squared + count;
```

```
0x40113a <+52>: mov  -0x20(%rsp,%rdi,4),%eax
```

```
0x40113e <+56>: imul  %eax,%eax
```

```
0x401141 <+59>: add  $0x7,%eax
```

```
0x401144 <+62>: retq
```

CPU

%edi

param

%rsp

stack pointer

%rsp = an address!

%eax

total



Q2: Register Reuse

What register stores the values of **total** and **squared**?

```
int total = arr[param];  
int squared = total * total;  
return squared + count;
```

```
0x40113a <+52>: mov -0x20(%rsp,%rdi,4),%eax
```

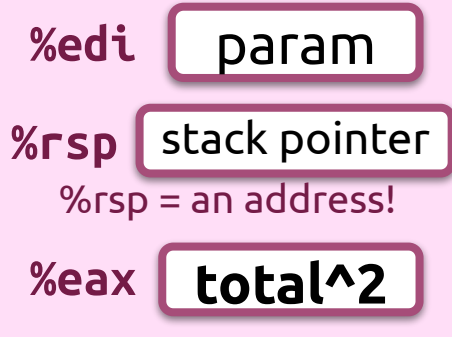
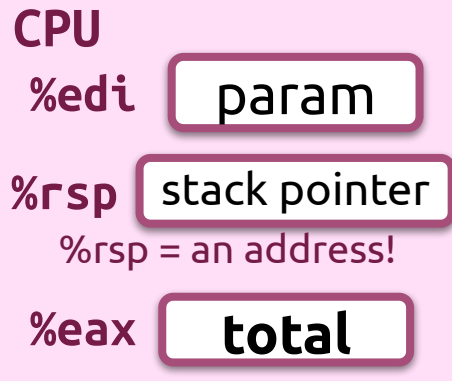
```
0x40113e <+56>: imul %eax,%eax
```

```
0x401141 <+59>: add $0x7,%eax
```

```
0x401144 <+62>: retq
```

imul src, dest is equivalent to **dest *= src;**

This does **\$eax *= \$eax**, squaring the value currently in register \$eax, or total. Thus, **\$eax** is now updated with value **total * total**, or the variable **squared**.





Exercise 2: C to Assembly Translation

Using a **compiler explorer**, get more experience with connecting assembly back to the C source.

Our eventual intent is **reverse-engineering** (uncovering a program's behavior purely from its assembly—no source code!).

Start with Q3 and Q4!



~ 40 min + 10 min all-lab discussion!



Q3: How do these functions differ in assembly?

```
void deref_one(char *ptr, long index) {  
    *ptr='\0';  
    ptr[index]='\0';  
}
```

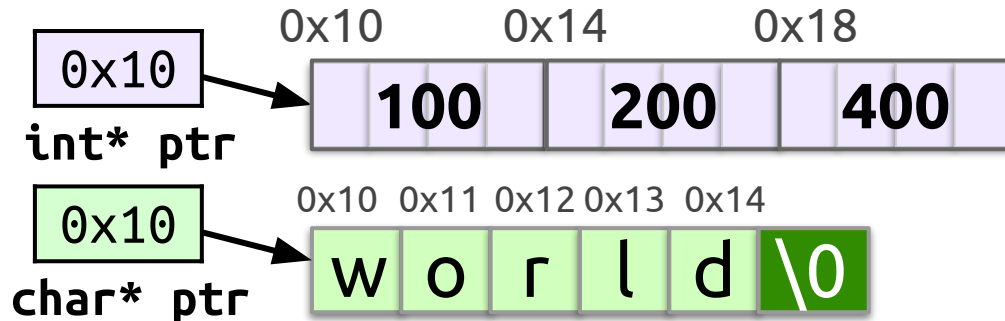
```
void deref_two(int *ptr, long index) {  
    *ptr = 0;  
    ptr[index] = 0;  
}
```

deref_one:

```
mov $0, ($rdi)  
movb $0, ($rdi, $rsi)  
ret
```

deref_two:

```
mov $0, ($rdi)  
movl $0, ($rdi, $rsi, 4)  
ret
```



Q3: How do these functions differ in assembly?

```
void deref_one(char *ptr, long index) {  
    *ptr='\0';  
    ptr[index]='\0';  
}
```

Each **char** element is
1 byte, so use **movb**

```
void deref_two(int *ptr, long index) {  
    *ptr = 0;  
    ptr[index] = 0;  
}
```

Each **int** element is **4 bytes**, so use **movl**

deref_one:

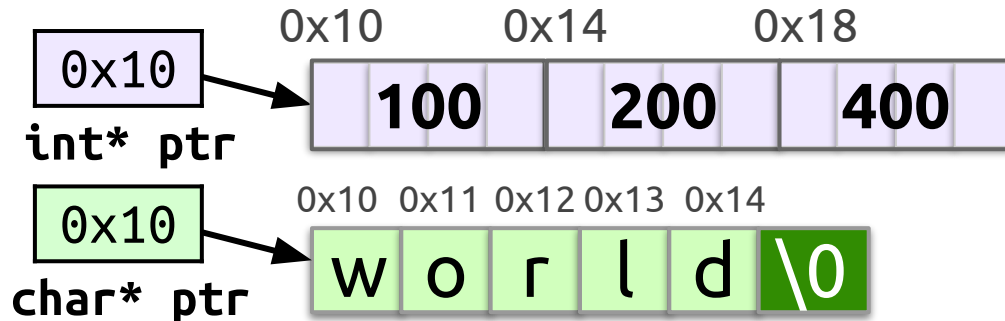
```
mov $0, ($rdi)  
movb $0, ($rdi, $rsi)  
ret
```

This is $\$rdi + \$rsi$

deref_two:

```
mov $0, ($rdi)  
movl $0, ($rdi, $rsi, 4)  
ret
```

This is $\$rdi + 4 * \$rsi$



Pointer arithmetic also changes!

- ❑ With an **int*** pointer, we move **4 bytes for each index**.
- ❑ With a **char*** pointer, we move **just 1 byte per index**.

Q4: Two's complement ~~so very~~ ingenious math

```
int signed_arithmetic(int a, int b) {  
    return (a - b) * a;  
}
```

```
unsigned int unsigned_arithmetic  
    (unsigned int a, unsigned int b) {  
    return (a - b) * a;  
}
```



```
mov $edi, $eax  
subl $esi, $eax  
imul $edi, $eax  
ret
```

Check-Off Q : What benefit does two's complement have at the assembly level?

Q4: Two's complement ~~so very~~ ingenious math

```
int signed_arithmetic(int a, int b) {  
    return (a - b) * a;  
}
```

```
unsigned int unsigned_arithmetic  
    (unsigned int a, unsigned int b) {  
    return (a - b) * a;  
}
```

```
mov $edi, $eax  
subl $esi, $eax  
imul $edi, $eax  
ret
```

Same assembly code for both signed and unsigned arithmetic!

Check-Off Q ☒: What benefit does two's complement have at the assembly level?

4	0 ... 1 0 0	-2	1 ... 1 1 0
+ 3	0 ... 0 1 1	+ 3	0 ... 0 1 1
7	0 ... 1 1 1	1	0 ... 0 0 1

Under **two's complement**, the bit representations are created so that we can just **add / subtract the bits** without worrying about signed or unsigned.

Thus, **in assembly**, as we're working with bit representations, **we can just perform the same addition of bits in both cases.**



Q5: Right-Shift in Assembly: Arithmetic vs Logical

```
int signed_arithmetic(int a, int b) {  
    a >>= b;  
    return (a - b) * a;  
}
```

```
unsigned int unsigned_arithmetic  
(unsigned int a, unsigned int b) {  
    a >>= b;  
    return (a - b) * a;  
}
```

```
signed_arithmetic:  
    movl $esi, $ecx  
    sarl %cl, $edi  
    ...
```

```
unsigned_arithmetic:  
    movl $esi, $ecx  
    shrl %cl, $edi  
    ...
```



Q5: Right-Shift in Assembly: Arithmetic vs Logical

```
int signed_arithmetic(int a, int b) {  
    a >>= b;  
    return (a - b) * a;  
}  
  
unsigned int unsigned_arithmetic  
    (unsigned int a, unsigned int b) {  
    a >>= b;  
    return (a - b) * a;  
}
```

Why are they different?

```
signed_arithmetic:  
    movl $esi, $ecx  
    sarl %cl, $edi  
    ...
```

```
unsigned_arithmetic:  
    movl $esi, $ecx  
    shrl %cl, $edi  
    ...
```

▶ Q5: Right-Shift in Assembly: Arithmetic vs Logical

```
int signed_arithmetic(int a, int b) {  
    a >>= b;  
    return (a - b) * a;  
}  
  
unsigned int unsigned_arithmetic  
(unsigned int a, unsigned int b) {  
    a >>= b;  
    return (a - b) * a;  
}
```

signed_arithmetic:

```
movl $esi, $ecx
```

```
sarl %cl, $edi
```

... Uses **sarl** (arithmetic
right-shift, fill w/ sign bit)

unsigned_arithmetic:

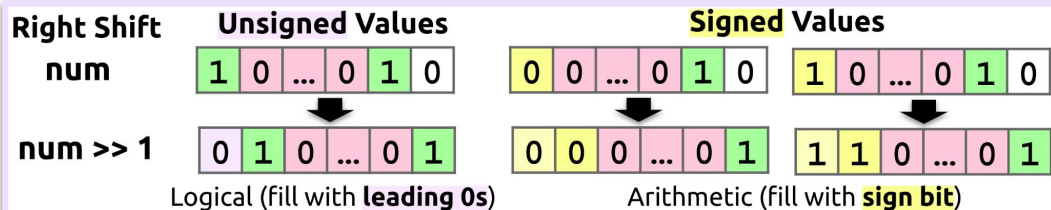
```
movl $esi, $ecx
```

```
shrl %cl, $edi
```

... Uses **shrl** (logical
right-shift, fill w/ 0 bits)

Why are they different?

Recall from Lecture (and Lab 1!),
that **right-shifting works
differently** between unsigned
and signed integers.



Continue with Q7, Q8,
Q9 of *C to Assembly!*





lea instruction: load effective address

Key Instruction: lea src, dest

- Similar to mov, **except that it doesn't dereference / look at memory**, i.e. it copies the value of src itself to a register, not the data at the address src.
- It's really just pointer arithmetic! 🤔



lea instruction: load effective address

Key Instruction: lea src, dest

- Similar to mov, **except that it doesn't dereference / look at memory**, i.e. it copies the value of src itself to a register, not the data at the address src.
- It's really just pointer arithmetic! 🤔

%rax

0x100

%rcx

0x10

%rdx

CPU

2002

0x100

2025

0x140

RAM/Memory

leaq (%rax, %rcx, 4), %rdx

%rdx

movq (%rax, %rcx, 4), %rdx

%rdx



lea instruction: load effective address

Key Instruction: lea src, dest

- Similar to mov, **except that it doesn't dereference / look at memory**, i.e. it copies of the value of src itself to a register, not the data at the address src.
- It's really just pointer arithmetic! 🤔

%rax

0x100

%rcx

0x10

%rdx

CPU

2002

0x100

2025

0x140

RAM/Memory

leaq (%rax, %rcx, 4), %rdx

%rdx

0x140

← holds %rax + 4 * %rcx

lea doesn't dereference this address/look at memory!

movq (%rax, %rcx, 4), %rdx

%rdx

2025

← holds Mem[%rax + 4 * %rcx]

Meanwhile, **mov** dereferences the address to get 2025!



Q7: ~~lua-magic~~ lea magic

By storing the parameters `x` and `y` in registers, we can effectively **do addition and scaling just using `lea`** due to its syntax quirks!

```
int combine(int x, int y) {  
    return x + y;  
}
```

combine:

```
    leal ($rdi, $rsi), $eax  
    ret
```

```
int combine2(int x, int y) {  
    return x + 2 * y;  
}
```

combine2:

```
    leal ($rdi, $rsi, 2), $eax  
    ret
```



Q7: ~~lua-magic~~ lea magic

By storing the parameters `x` and `y` in registers, we can effectively **do addition and scaling just using lea** due to its syntax quirks!

```
                %rdi    %rsi
int combine(int x, int y) {
    return x + y;
}
```

```
                %rdi    %rsi
int combine2(int x, int y) {
    return x + 2 * y;
}
```

combine:

```
    leal ($rdi, $rsi), %eax
    ret
```

combine2:

```
    leal ($rdi, $rsi, 2), %eax
    ret
```

%rdi

X

%rsi

Y

%rax

CPU



Q7: ~~luka-magic~~ lea magic

By storing the parameters `x` and `y` in registers, we can effectively **do addition and scaling just using lea** due to its syntax quirks!

```
int combine(%rdiint x, %rsiint y) {  
    return x + y;  
}
```

```
int combine2(%rdiint x, %rsiint y) {  
    return x + 2 * y;  
}
```

combine:

```
leal ($rdi, $rsi), %eax  
ret
```

This is $\$rdi + \$rsi$, or $x + y$.

combine2:

```
leal ($rdi, $rsi, 2), %eax  
ret
```

This is $\$rdi + 2 * \$rsi$, or $x + 2 * y$.

%rdi %rsi %rax

CPU



Q8: left-shifting magic to avoid multiplying

```
int combine3(int x, int y) {  
    return x * 17;  
}
```

combine3:

```
    movl $edi, $eax  
    sall $4, $eax  
    addl $edi, $eax  
    ret
```

%rdi

x

%rsi

y

%rax

CPU



Q8: left-shifting magic to avoid multiplying

```
int combine3(int x, int y) {  
    return x * 17;  
}
```

combine3:

`movl $edi, %eax`

`sall $4, %eax`

`addl $edi, %eax`

`ret`

`%rdi`

`x`

`%rsi`

`y`

`%rax`

CPU

`%eax`

`x`

Moves `$edi (x)` into `%eax`

`%eax`

`16*x`

`%eax << 4`, or `%eax * 24`

`%eax`

`17*x`

`%eax += $edi`, or add `x`



Q8: left-shifting magic to avoid multiplying

```
int combine3(int x, int y) {  
    return x * 17;  
}
```

combine3:

```
movl $edi, %eax  
sall $4, %eax  
addl $edi, %eax  
ret
```

%rdi %rsi %rax
CPU

%eax Moves %edi (x) into %eax
%eax %eax << 4, or %eax * 2^4
%eax %eax += %edi, or add x

Multiplication is hard. I'm not a huge fan—neither is the compiler.

Instead of directly multiplying by 17, it **(1) “multiplies” by 16**=2⁴, by a left shift of 4, via `sall $4, %eax`, then **(2) adds itself**. This is all because:

$$17 * x = 16 * x + x = (2^4 * x) + x. \text{ Lol } \checkmark$$



Division in Assembly

Example: **div %ebx**

DIVIDEND: **%rdx** (higher 64 bits)
and **%rax** (lower 64 bits)

DIVISOR: argument that you pass in
(e.g., **%ebx**)

Note: Dividend must be stored across
all 128 bits of **\$rdx** and **\$rax**!

CPU

0	100	3
%rdx	%rax	%ebx



Division in Assembly

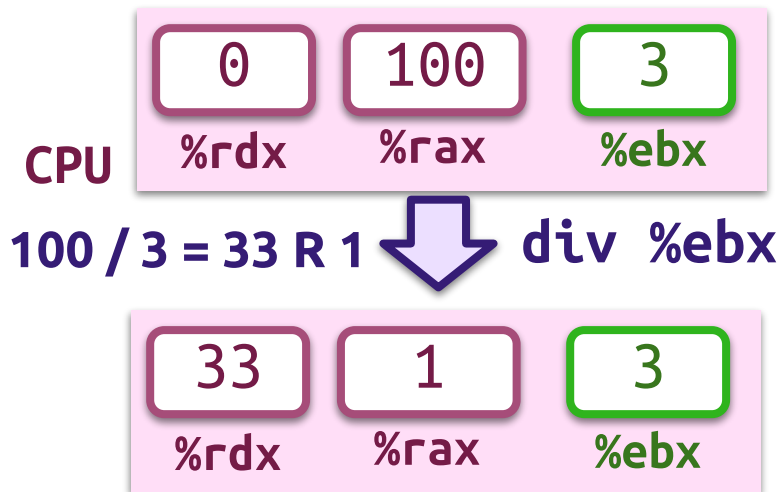
Example: **div %ebx**

DIVIDEND: **%rdx** (higher 64 bits)
and **%rax** (lower 64 bits)

DIVISOR: argument that you pass in
(e.g., **%ebx**)

Afterward, QUOTIENT is stored in **\$rdx**,
and REMAINDER in **\$rax**

Note: Dividend must be stored across
all 128 bits of \$rdx and \$rax!





Division in Assembly

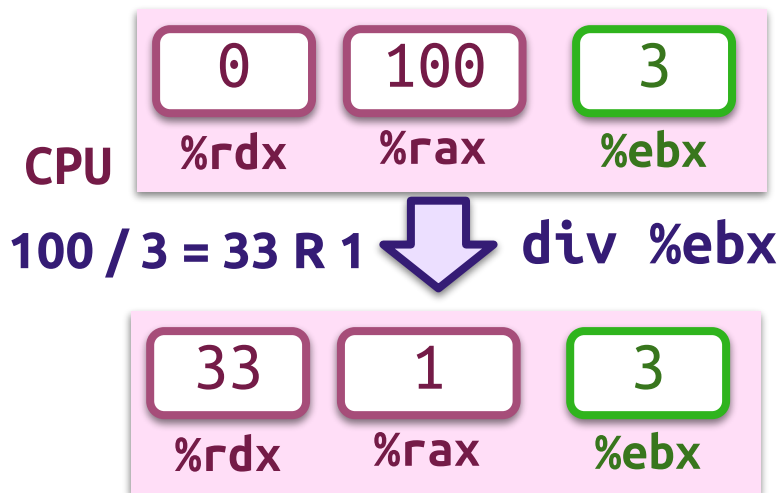
Example: **div %ebx**

DIVIDEND: **%rdx** (higher 64 bits)
and **%rax** (lower 64 bits)

DIVISOR: argument that you pass in
(e.g., **%ebx**)

Afterward, QUOTIENT is stored in **\$rdx**,
and REMAINDER in **\$rax**

Note: Dividend must be stored across
all 128 bits of \$rdx and \$rax!



Like multiplication, division is tedious for CPUs (remember how scary long division felt back then! 🤪), so it **avoids using div when possible**, e.g.,

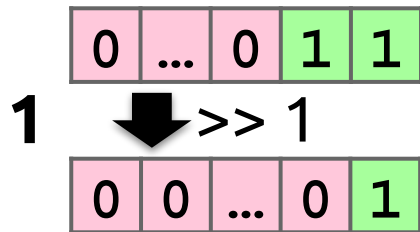
- ❑ Dividing by 2 is the same as **>> 1**, **right-shifting** by 1 position
- ❑ Dividing by 4 is the same as **>> 2**, **right-shifting** by 2 positions, and so forth



Integer Division: Rounding via Right-Shift

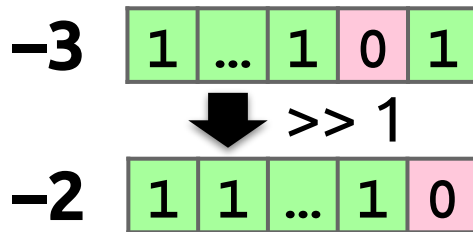
In C, when we divide integers, we **round the result** to an int, and **towards zero**. As an example, say we're dividing by 2, for which we can use right-shift ($\gg$).

3 Positive Case



$3 / 2 = 1$ ✓ Rounding down

Negative Case



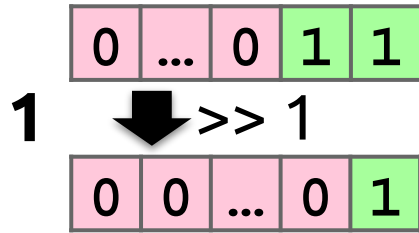
$-3 / 2 \neq -2$ ✗ Rounding away from 0



Integer Division: Rounding via Right-Shift

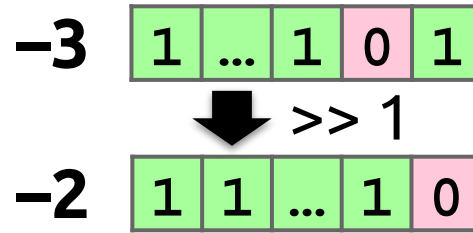
In C, when we divide integers, we **round the result** to an int, and **towards zero**. As an example, say we're dividing by 2, for which we can use right-shift ($\gg$).

3 Positive Case



$3 / 2 = 1$ ✓ Rounding down

Negative Case



We want $-3 / 2 = -1$ instead, rounding up or towards 0!

$-3 / 2 \neq -2$ ✗ Rounding away from 0

For positive integers, we can just right-shift; but for **negative integers**, this will round down / away from 0 — so **we need to “fix-up” the value** beforehand!

Example: For $-3 / 2$, calculate $-2 / 2$ instead, by adding a **fix-up of +1** to -3 .



Now can use a simple right shift ($\gg$) to get correct answer rounded up!

Check-Off Question

What tricks does GCC (the compiler) use to handle division for positive and negative numbers? Why are these necessary?

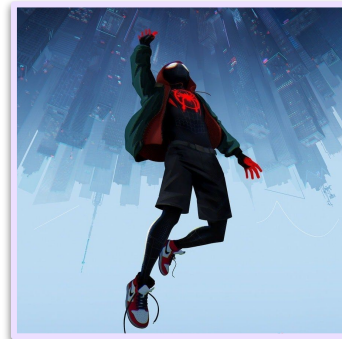
Check-Off Question

What tricks does GCC (the compiler) use to handle division for positive and negative numbers? Why are these necessary?

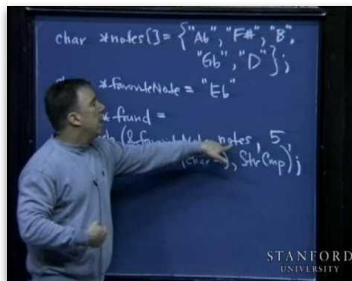
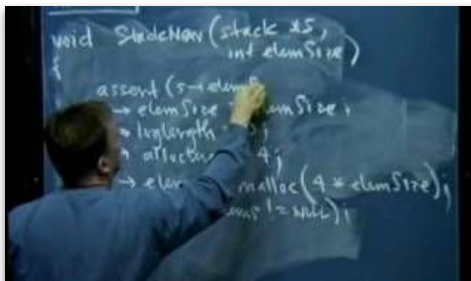
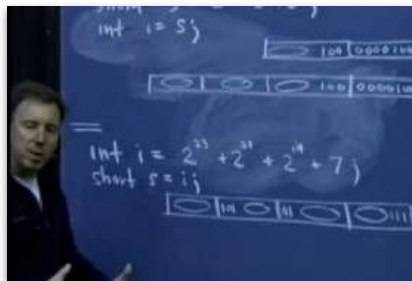
Answer: The GCC compiler has to treat division by positive and negative numbers as special cases, and adds a fix-up* (e.g., adding the sign bit) to negative numbers to ensure **they're rounded toward zero**.

This is needed because naturally, shifting a negative number will round down (e.g., **away from zero, or more negative**), whereas for a positive number it will round toward zero (**less positive**).

Note: Fix-up value depends on the dividend, e.g., +1 for dividing by 2, +3 for dividing by 4, etc., (**think round_up from Lab 1!**)



Thank you and have a beautiful weekend!



- Please reach out to me if you have any questions (lab, course material, etc), or topics you want covered!
-  Contact: bbyan@stanford.edu, email or over Slack
- **Assembly is an art**, and I promise you'll get fantastic at it the more you practice, diagram, and rehearse :)



“Avengers Assembly!” — Captain America

“Binary Bomb was the time of my life” — not me lol, but may be you!! 