CS107, Lecture 10
Introduction to Assembly

Reading: B&O 3.1-3.4
1. **Bits and Bytes** - How can a computer represent integer numbers?
2. **Chars and C-Strings** - How can a computer represent and manipulate more complex data like text?
3. **Pointers, Stack and Heap** – How can we effectively manage all types of memory in our programs?
4. **Generics** - How can we use our knowledge of memory and data representation to write code that works with any data type?
5. **Assembly** - How does a computer interpret and execute C programs?
6. **Heap Allocators** - How do core memory-allocation operations like `malloc` and `free` work?
CS107 Topic 5: How does a computer interpret and execute C programs?
How does a computer interpret and execute C programs?

Why is answering this question important?

- Learning how our code is really translated and executed helps us write better code
- We can learn how to reverse engineer and exploit programs at the assembly level

**assign5:** find and exploit vulnerabilities in an ATM program, reverse engineer a program without seeing its code, and de-anonymize users given a data leak.
Learning Assembly

This Lecture

Moving data around

Lecture 11

Arithmetic and logical operations

Lecture 12

Control flow

Lecture 13

Function calls
Learning Goals

• Learn what assembly language is and why it is important
• Become familiar with the format of human-readable assembly and x86
• Learn the `mov` instruction and how data moves around at the assembly level
Lecture Plan

- **Overview**: GCC and Assembly
- **Demo**: Looking at an executable
- Registers and The Assembly Level of Abstraction
- The `mov` Instruction

```bash
cp -r /afs/ir/class/cs107/lecture-code/lect10 .
```
Lecture Plan

• Overview: GCC and Assembly
• Demo: Looking at an executable
• Registers and The Assembly Level of Abstraction
• The mov Instruction

```bash
cp -r /afs/ir/class/cs107/lecture-code/lect10 .
```
Bits all the way down

Data representation so far
• Integer (unsigned int, 2’s complement signed int)
• char (ASCII)
• Address (unsigned long)
• Aggregates (arrays, structs)

The code itself is binary too!
• Instructions (machine encoding)
GCC

• **GCC** is the compiler that converts your human-readable code into machine-readable instructions.

• C, and other languages, are high-level abstractions we use to write code efficiently. But computers don’t really understand things like data structures, variable types, etc. Compilers are the translator!

• Pure machine code is 1s and 0s – everything is bits, even your programs! But we can read it in a human-readable form called **assembly**. (Engineers used to write code in assembly before C).

• There may be multiple assembly instructions needed to encode a single C instruction.

• We’re going to go behind the curtain to see what the assembly code for our programs looks like.
Central Processing Units (CPUs)

Intel 8086, 16-bit microprocessor ($86.65, 1978)

Raspberry Pi BCM2836 32-bit ARM microprocessor ($35 for everything, 2015)

Intel Core i9-9900K 64-bit 8-core multi-core processor ($449, 2018)
Why are we reading assembly?

Main goal: Information retrieval

- We will not be writing assembly! (that’s the compiler’s job)
- Rather, we want to translate the assembly back into our C code.
- Knowing how our C code is converted into machine instructions gives us insight into how to write more efficient, cleaner code.
Lecture Plan

• **Overview:** GCC and Assembly
• **Demo:** Looking at an executable
• Registers and The Assembly Level of Abstraction
• The **mov** Instruction

```bash
cp -r /afs/ir/class/cs107/lecture-code/lect10 .
```
Demo: Looking at an Executable (objdump -d)
Our First Assembly

int sum_array(int arr[], int nelems) {
    int sum = 0;
    for (int i = 0; i < nelems; i++) {
        sum += arr[i];
    }
    return sum;
}

What does this look like in assembly?
int sum_array(int arr[], int nelems) {
    int sum = 0;
    for (int i = 0; i < nelems; i++) {
        sum += arr[i];
    }
    return sum;
}

0000000000401136 <sum_array>:
401136:   b8 00 00 00 00
40113b:   ba 00 00 00 00
401140:   39 f0
401142:   7d 0b
401144:   48 63 c8
401147:   03 14 8f
40114a:   83 c0 01
40114d:   eb f1
40114f:   89 d0
401151:   c3

mov $0x0,%eax
mov $0x0,%edx
cmp %esi,%eax
jge 40114f <sum_array+0x19>
movslq %eax,%rcx
add (%rdi,%rcx,4),%edx
add $0x1,%eax
jmp 401140 <sum_array+0xa>
mov %edx,%eax
retq

make objdump -d sum
Our First Assembly

0000000000401136 <sum_array>:

401136:  b8  00  00  00  00  mov  $0x0,%eax
        40113b:  ba  00  00  00  00  mov  $0x0,%edx
        401140:  39  f0  cmp   %esi,%eax
        401142:  7d  0b  jge  40114f <sum_array+0x19>
        401144:  48  63  c8  movslq  %eax,%rcx
        401147:  03  14  8f  add   (%rdi,%rcx,4),%edx
        40114a:  83  c0  01  add   $0x1,%eax
        40114d:  eb  f1  jmp  401140 <sum_array+0xa>
        40114f:  89  d0  mov   %edx,%eax
        401151:  c3  retq
This is the name of the function (same as C) and the memory address where the code for this function starts.
Our First Assembly

These are the memory addresses where each of the instructions live. Sequential instructions are sequential in memory.
Our First Assembly

This is the assembly code: “human-readable” versions of each machine code instruction.

```
0000000000401136 <sum_array>:
  401136:   b8 00 00 00 00 00
  40113b:   ba 00 00 00 00 00
  401140:   39 f0
  401142:   7d 0b
  401144:   48 63 c8
  401147:   03 14 8f
  40114a:   83 c0 01
  40114d:   eb f1
  40114f:   89 d0
  401151:   c3

mov  $0x0,%eax
mov  $0x0,%edx
cmp  %esi,%eax
jge  40114f <sum_array+0x19>
movslq  %eax,%rcx
add  (%rdi,%rcx,4),%edx
add  $0x1,%eax
jmp  401140 <sum_array+0xa>
mov  %edx,%eax
retq
```
Our First Assembly

This is the machine code: raw hexadecimal instructions, representing binary as read by the computer. Different instructions may be different byte lengths.

```
00000000000401136 <sum_array>:
401136:   b8 00 00 00 00
40113b:   ba 00 00 00 00
401140:   39 f0
401142:   7d 0b
401144:   48 63 c8
401147:   03 14 8f
40114a:   83 c0 01
40114d:   eb f1
40114f:   89 d0
401151:   c3
```

```cpp
mov    $0x0,%eax
mov    $0x0,%edx
cmp    %esi,%eax
jge    40114f <sum_array+0x19>
movslq  %eax,%rcx
add    (%rdi,%rcx,4),%edx
add    $0x1,%eax
jmp    401140 <sum_array+0xa>
mov    %edx,%eax
retq
```
Our First Assembly

00000000000401136 <sum_array>:
401136:   b8 00 00 00 00 00                     mov $0x0,%eax
40113b:   ba 00 00 00 00 00                     mov $0x0,%edx
401140:   39 f0                                cmp %esi,%eax
401142:   7d 0b                                jge 40114f <sum_array+0x19>
401144:   48 63 c8                             movslq %eax,%rcx
401147:   03 14 8f                             add (%rdi,%rcx,4),%edx
40114a:   83 c0 01                             add $0x1,%eax
40114d:   eb f1                               jmp 401140 <sum_array+0xa>
40114f:   89 d0                               mov %edx,%eax
401151:   c3                                 retq
Our First Assembly

0000000000401136 <sum_array>:

```
401136:   b8 00 00 00 00       mov    $0x0,%eax
40113b:   ba 00 00 00 00       mov    $0x0,%edx
401140:   39 f0              cmp    %esi,%eax
401142:   7d 0b              jge    40114f <sum_array+0x19>
401144:   48 63 c8           movslq  %eax,%rcx
401147:   03 14 8f           add    (%rdi,%rcx,4),%edx
40114a:   83 c0 01           add    $0x1,%eax
40114d:   eb f1              jmp    401140 <sum_array+0xa>
40114f:   89 d0              mov    %edx,%eax
401151:   c3                retq
```

Each instruction has an operation name ("opcode").
Our First Assembly

```
0000000000401136 <sum_array>:
401136:  b8 00 00 00 00 00
        mov $0x0,%eax
40113b:  ba 00 00 00 00
        mov $0x0,%edx
401140:  39 f0
        cmp %esi,%eax
401142:  7d 0b
        jge 40114f <sum_array+0x19>
401144:  48 63 c8
        movslq %eax,%rcx
401147:  03 14 8f
        add (%rdi,%rcx,4),%edx
40114a:  83 c0 01
        add $0x1,%eax
40114d:  eb f1
        jmp 401140 <sum_array+0xa>
40114f:  89 d0
        mov %edx,%eax
```

Each instruction can also have arguments (“operands”).
Our First Assembly

00000000000401136 <sum_array>:
401136:   b8 00 00 00 00 00 00 00 00 00 00 00 00 00 00
40113b:   ba 00 00 00 00 00 00 00 00 00 00 00 00 00 00
401140:   39 f0
401142:   7d 0b
401144:   48 63 c8
401147:   03 14 8f
40114a:   83 c0 01
40114d:   eb f1
40114f:   89 d0
401151:   c3

mov $0x0,%eax
mov $0x0,%edx
cmp %esi,%eax
jge 40114f <sum_array+0x19>
movslq %eax,%rcx
add (%rdi,%rcx,4),%edx
add $0x1,%eax
jmp 401140 <sum_array+0xa>
mov %edx,%eax
retq

$[number] means a constant value, or “immediate” (e.g. 1 here).
Our First Assembly

0000000000401136 <sum_array>:
401136:  b8 00 00 00 00 00
40113b:  ba 00 00 00 00 00
401140:  39 f0
401142:  7d 0b
401144:  48 63 c8
401147:  03 14 8f
40114a:  83 c0 01
40114d:  eb f1
40114f:  89 d0
401151:  c3

mov $0x0,%eax
mov $0x0,%edx
cmp %esi,%eax
jge 40114f <sum_array+0x19>
movslq %eax,%rcx
add (%rdi,%rcx,4),%edx
add $0x1,%eax
jmp 401140 <sum_array+0xa>
mov %edx,%eax
retq

[%[name]] means a register, a storage location on the CPU (e.g. edx here).
• **https://web.stanford.edu/class/cs107/resources/x86-64-reference.pdf**

• **B&O book:**
  - Canvas -> Files
    -> Bryant_OHallaron_ch3.1-3.8.pdf

• It’s like learning how to read (not speak) a new language! (again!)
Lecture Plan

• **Overview**: GCC and Assembly
• **Demo**: Looking at an executable
• **Registers and The Assembly Level of Abstraction**
• The **mov** instruction
Assembly Abstraction

• C abstracts away the low-level details of machine code. It lets us work using variables, variable types, and other higher-level abstractions.
• C and other languages let us write code that works on most machines.
• Assembly code is just bytes! No variable types, no type checking, etc.
• Assembly/machine code is processor-specific.
• What is the level of abstraction for assembly code?
Registers

%rax
### Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>![rax]</td>
</tr>
<tr>
<td>%rbx</td>
<td>![rbx]</td>
</tr>
<tr>
<td>%rcx</td>
<td>![rcx]</td>
</tr>
<tr>
<td>%rdx</td>
<td>![rdx]</td>
</tr>
<tr>
<td>%rsi</td>
<td>![rsi]</td>
</tr>
<tr>
<td>%rdi</td>
<td>![rdi]</td>
</tr>
<tr>
<td>%rbp</td>
<td>![rbp]</td>
</tr>
<tr>
<td>%rsp</td>
<td>![rsp]</td>
</tr>
<tr>
<td>%r8</td>
<td>![r8]</td>
</tr>
<tr>
<td>%r9</td>
<td>![r9]</td>
</tr>
<tr>
<td>%r10</td>
<td>![r10]</td>
</tr>
<tr>
<td>%r11</td>
<td>![r11]</td>
</tr>
<tr>
<td>%r12</td>
<td>![r12]</td>
</tr>
<tr>
<td>%r13</td>
<td>![r13]</td>
</tr>
<tr>
<td>%r14</td>
<td>![r14]</td>
</tr>
<tr>
<td>%r15</td>
<td>![r15]</td>
</tr>
</tbody>
</table>
What is a register?

A register is a fast read/write memory slot right on the CPU that can hold variable values.

Registers are **not** located in memory.
• A **register** is a 64-bit space inside the processor.

• There are 16 registers available, each with a unique name.

• Registers are like “scratch paper” for the processor. Data being calculated or manipulated is moved to registers first. Operations are performed on registers.

• Registers also hold parameters and return values for functions.

• Registers are extremely *fast* memory!

• Processor instructions consist mostly of moving data into/out of registers and performing arithmetic on them. This is the level of logic your program must be in to execute!
Assembly instructions manipulate these registers. For example:

• One instruction adds two numbers in registers
• One instruction transfers data from a register to memory
• One instruction transfers data from memory to a register
Computer architecture

registers accessed by name
ALU is main workhorse of CPU

memory needed for program execution (stack, heap, etc.) accessed by address

disk/server stores program when not executing
• GCC compiles your program – it lays out memory on the stack and heap and generates assembly instructions to access and do calculations on those memory locations.

• Here’s what the “assembly-level abstraction” of C code might look like:

<table>
<thead>
<tr>
<th>C</th>
<th>Assembly Abstraction</th>
</tr>
</thead>
</table>
| int sum = x + y; | 1) Copy x into register 1  
                          2) Copy y into register 2  
                          3) Add register 2 to register 1  
                          4) Write register 1 to memory for sum |
Assembly

- We are going to learn the x86-64 instruction set architecture. This instruction set is used by Intel and AMD processors.
- There are many other instruction sets: ARM, MIPS, etc.
Instruction set architecture (ISA)

A contract between program/compiler and hardware:
• Defines operations that the processor (CPU) can execute
• Data read/write/transfer operations
• Control mechanisms

Intel originally designed their instruction set back in 1978.
• Legacy support is a huge issue for x86-64
• Originally 16-bit processor, then 32 bit, now 64 bit. These design choices dictated the register sizes (and even register/instruction names).
Lecture Plan

• **Overview:** GCC and Assembly
• **Demo:** Looking at an executable
• Registers and The Assembly Level of Abstraction
• **The mov Instruction**

```bash
cp -r /afs/ir/class/cs107/lecture-code/lect10 .
```
The `mov` instruction copies bytes from one place to another; it is like the assignment operator (=) in C.

\[
\text{mov} \quad \text{src}, \text{dst}
\]

The `src` and `dst` can each be one of:

- Immediate (constant value, like a number) \((\text{only src})\)
  
  \[\$0x104\]
  
  \[%rbx\]

- Register
  
  \[\text{Direct address} \quad 0x6005c0\]

- Memory Location
  
  \((\text{at most one of src, dst})\)
Operand Forms: Immediate

```
mov $0x104,___
```

Copy the value 0x104 into some destination.
Operand Forms: Registers

Copy the value in register %rbx into some destination.

\texttt{mov} %rbx,____

Copy the value from some source into register %rbx.

\texttt{mov} ____,%rbx
Operand Forms: Absolute Addresses

Copy the value at address 0x104 into some destination.

\[
\text{mov} \quad 0x104, \\
\text{mov} \quad \\
\]

Copy the value from some source into the memory at address 0x104.
Practice #1: Operand Forms

What are the results of the following move instructions (executed separately)? For this problem, assume the value 5 is stored at address 0x42, and the value 8 is stored in %rbx.

1. mov $0x42,%rax

2. mov 0x42,%rax

3. mov %rbx,0x55
Operand Forms: Indirect

Copy the value at the address stored in register %rbx into some destination.

```
mov (%rbx),_____  
```

Copy the value from some source into the memory at the address stored in register %rbx.

```
mov _____,(%rbx)
```
Operand Forms: Base + Displacement

```
mov 0x10(%rax),_________
```

Copy the value at the address (\texttt{0x10} plus what is stored in register \texttt{%rax}) into some destination.

```
mov __________,0x10(%rax)
```

Copy the value from some source into the memory at the address (\texttt{0x10 plus} what is stored in register \texttt{%rax}).
Operand Forms: Indexed

Copy the value at the address which is (the sum of the values in registers %rax and %rdx) into some destination.

```
mov (%rax,%rdx),__________
```

Copy the value from some source into the memory at the address which is (the sum of the values in registers %rax and %rdx).

```
mov _____________,(%rax,%rdx)
```
Operand Forms: Indexed

Copy the value at the address which is (the sum of 0x10 plus the values in registers %rax and %rdx) into some destination.

```
mov 0x10(%rax,%rdx),_____
```

Copy the value from some source into the memory at the address which is (the sum of 0x10 plus the values in registers %rax and %rdx).

```
mov ________,0x10(%rax,%rdx)
```
Practice #2: Operand Forms

What are the results of the following move instructions (executed separately)? For this problem, assume the value 0x11 is stored at address 0x10C, 0xAB is stored at address 0x104, 0x100 is stored in register %rax and 0x3 is stored in %rdx.

1. `mov $0x42,(%rax)`
2. `mov 4(%rax),%rcx`
3. `mov 9(%rax,%rdx),%rcx`

Imm\( (r_b, r_i) \) is equivalent to address Imm + R\[r_b\] + R\[r_i\]
Operand Forms: Scaled Indexed

Copy the value at the address which
is \(4\ \text{times}\) the value in register
\(\%rdx\) into some destination.

\[
\text{mov} \ (,\%rdx,4),\underline{}\]

Copy the value from some source into the
memory at the address which is \(4\ \text{times}\)
the value in register \(\%rdx\).

\[
\underline{} \ (,\%rdx,4)\]

The scaling factor
(e.g. 4 here) must
be hardcoded to
be either 1, 2, 4
or 8.
Operand Forms: Scaled Indexed

Copy the value at the address which is (4 times the value in register %rdx, plus 0x4), into some destination.

```
mov 0x4(%rdx,4),______
```

Copy the value from some source into the memory at the address which is (4 times the value in register %rdx, plus 0x4).

```
mov ______,0x4(%rdx,4)
```
Operand Forms: Scaled Indexed

Copy the value at the address which is (the value in register %rax plus 2 times the value in register %rdx) into some destination.

\[ \text{mov} \quad (%rax, %rdx, 2), \_\_\_\_\_\_\_\_ \]

Copy the value from some source into the memory at the address which is (the value in register %rax plus 2 times the value in register %rdx).

\[ \text{mov} \quad \_\_\_\_\_\_\_\_\_\_\_, (%rax, %rdx, 2) \]
Operand Forms: Scaled Indexed

Copy the value at the address which is \(0x4\) plus the value in register \(%rax\) plus 2 times the value in register \(%rdx\) into some destination.

```
mov \[0x4(%rax,%rdx,2)],_____
```

Copy the value from some source into the memory at the address which is \(0x4\) plus the value in register \(%rax\) plus 2 times the value in register \(%rdx\).

```
mov ______,\[0x4(%rax,%rdx,2)
```
Most General Operand Form

\[ \text{Imm}(r_b, r_i, s) \]

is equivalent to...

\[ \text{Imm} + R[r_b] + R[r_i]*s \]
**Most General Operand Form**

\[ \text{Imm}(r_b, r_i, s) \text{ is equivalent to address } \text{Imm} + R[r_b] + R[r_i]*s \]

- **Displacement:** pos/neg constant (if missing, = 0)
- **Index:** register (if missing, = 0)
- **Base:** register (if missing, = 0)
- **Scale** must be 1, 2, 4, or 8 (if missing, = 1)
### Operand Forms

<table>
<thead>
<tr>
<th>Type</th>
<th>Form</th>
<th>Operand Value</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate</td>
<td>$Imm$</td>
<td>Imm</td>
<td>Immediate</td>
</tr>
<tr>
<td>Register</td>
<td>$r_a$</td>
<td>R[$r_a$]</td>
<td>Register</td>
</tr>
<tr>
<td>Memory</td>
<td>Imm</td>
<td>M[Imm]</td>
<td>Absolute</td>
</tr>
<tr>
<td>Memory</td>
<td>($r_a$)</td>
<td>M[R[$r_a$]]</td>
<td>Indirect</td>
</tr>
<tr>
<td>Memory</td>
<td>Imm($r_b$)</td>
<td>M[Imm + R[$r_b$]]</td>
<td>Base + displacement</td>
</tr>
<tr>
<td>Memory</td>
<td>($r_b$, $r_i$)</td>
<td>M[R[$r_b$] + R[$r_i$]]</td>
<td>Indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>Imm($r_b$, $r_i$)</td>
<td>M[Imm + R[$r_b$] + R[$r_i$]]</td>
<td>Indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>($r_i$, $s$)</td>
<td>M[R[$r_i$] · $s$]</td>
<td>Scaled indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>Imm($r_i$, $s$)</td>
<td>M[Imm + R[$r_i$] · $s$]</td>
<td>Scaled indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>($r_b$, $r_i$, $s$)</td>
<td>M[R[$r_b$] + R[$r_i$] · $s$]</td>
<td>Scaled indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>Imm($r_b$, $r_i$, $s$)</td>
<td>M[Imm + R[$r_b$] + R[$r_i$] · $s$]</td>
<td>Scaled indexed</td>
</tr>
</tbody>
</table>

**Figure 3.3 from the book:** “Operand forms. Operands can denote immediate (constant) values, register values, or values from memory. The scaling factor $s$ must be either 1, 2, 4, or 8.”
Practice #3: Operand Forms

What are the results of the following move instructions (executed separately)? For this problem, assume the value 0x1 is stored in register %rcx, the value 0x100 is stored in register %rax, the value 0x3 is stored in register %rdx, and value 0x11 is stored at address 0x10C.

1. mov $0x42,0x{fc(,%rcx,4)}

2. mov (%rax,%rdx,4),%rbx

\[
\text{Imm}(r_b, r_i, s) \text{ is equivalent to address } \text{Imm} + R[r_b] + R[r_i]*s
\]

Displacement Base Index Scale
(1,2,4,8)
Goals of indirect addressing: C

Why are there so many forms of indirect addressing?

We see these indirect addressing paradigms in C as well!
int \texttt{sum\_array}(\texttt{int} \ arr[\texttt{]}, \ \texttt{int} \ nelems) \{ 
  \texttt{int} \ \texttt{sum} = 0; 
  \texttt{for} \ (\texttt{int} \ i = 0; \ i < \nelems; \ i++) \{ 
    \texttt{sum} += \texttt{arr}[i]; 
  \} 
  \texttt{return} \ \texttt{sum}; 
\}

\texttt{We’re 1/4}^{\text{th}} \text{ of the way to understanding assembly!} 
\textbf{What looks understandable right now?} 

Some notes: 
\begin{itemize}
  \item Registers store addresses and values
  \item \texttt{mov} \texttt{src}, \texttt{dst} \textit{copies} value into \texttt{dst}
  \item \texttt{sizeof} (\texttt{int}) is 4
  \item Instructions executed sequentially
\end{itemize}

\texttt{00000000004005b6 <sum\_array>}: 
\begin{verbatim}
4005b6:   ba 00 00 00 00
4005bb:   b8 00 00 00 00
4005c0:   eb 09
4005c2:   48 63 ca
4005c5:   03 04 8f
4005c8:   83 c2 01
\end{verbatim}

\texttt{mov} \texttt{$0x0,%edx} 
\texttt{mov} \texttt{$0x0,%eax} 
\texttt{jmp} \texttt{4005cb <sum\_array+0x15>} 
\texttt{movslq} \%edx,\%rcx 
\texttt{add} \texttt{(%rdi,\%rcx,4),%eax} 
\texttt{add} \texttt{$0x1,%edx} 
\texttt{cmp} \texttt{%esi,\%edx} 
\texttt{jl} \texttt{4005c2 <sum\_array+0xc>} 
\texttt{repz retq}

\texttt{We’ll come back to this example in future lectures!}
From Assembly to C

Spend a few minutes thinking about the main paradigms of the mov instruction.

• What might be the equivalent C-like operation?

• Examples (note %r__ registers are 64-bit):

1. mov $0x0,%rdx
2. mov %rdx,%rcx
3. mov $0x42,(%rdi)
4. mov (%rax,%rcx,8),%rax
From Assembly to C

Spend a few minutes thinking about the main paradigms of the `mov` instruction.

• What might be the equivalent C-like operation?

• Examples (note `%r__` registers are 64-bit):

1. `mov $0x0,%rdx` -> maybe long `x = 0`
2. `mov %rdx,%rcx` -> maybe long `x = y;`
3. `mov $0x42,(%rdi)` -> maybe `*ptr = 0x42;`
4. `mov (%rax,%rcx,8),%rax` -> maybe long `x = arr[i];`

Indirect addressing is like pointer arithmetic/deref!
Recap

- **Overview:** GCC and Assembly
- **Demo:** Looking at an executable
- **Registers and The Assembly Level of Abstraction**
- **The mov instruction**

**Next time:** diving deeper into assembly

**Lecture 10 takeaway:**
Assembly is the human-readable version of the form our programs are ultimately executed in by the processor. The compiler translates source code to machine code. The most common assembly instruction is `mov` to move data around.
Extra Practice
1. Extra Practice

Fill in the blank to complete the C code that

1. generates this assembly
2. has this register layout

```c
int x = ...
int *ptr = malloc(...);
...
___???___ = _???_;

mov %ecx,(%rax)
```

(Pedantic: You should sub in `<x>` and `<ptr>` with actual values, like 4 and 0x7fff80)
Fill in the blank to complete the C code that 1. generates this assembly 2. has this register layout

```c
int x = ...;
int *ptr = malloc(...);
...
___???___ = _???_;  *ptr = x;

mov %ecx,(%rax)
```

<val of x> <val of ptr>

%ecx  %rax
2. Extra Practice

Fill in the blank to complete the C code that

1. generates this assembly
2. results in this register layout

```c
long arr[5];
...
long num = ____??__;  

mov (%rdi, %rcx, 8),%rax
```

<val of num> 3 <val of arr>

%rax  %rcx  %rdi
Fill in the blank to complete the C code that generates this assembly and results in this register layout:

```
long arr[5];
...
long num = ____???____;
```

```
long num = arr[3];
long num = *(arr + 3);
long num = *(arr + y);
```

(assume long y = 3; declared earlier)

```
mov (%rdi, %rcx, 8),%rax
```

<val of num> 3 <val of arr>

%rax %rcx %rdi
Fill in the blank to complete the C code that

1. generates this assembly
2. has this register layout

```c
char str[5];
...
___???___ = 'c';
```

```asm
mov $0x63,(%rcx,%rdx,1)
```

<table>
<thead>
<tr>
<th><code>&lt;val of str&gt;</code></th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rcx</td>
<td>%rdx</td>
</tr>
</tbody>
</table>
Fill in the blank to complete the C code that 1. generates this assembly 2. has this register layout

```c
char str[5];
...
___???___ = 'c';
```

```assembly
mov $0x63,(%rcx,%rdx,1)
```

```c
str[2] = 'c';
*(str + 2) = 'c';
```

---

3. Extra Practice

<val of str>

<table>
<thead>
<tr>
<th>&lt;val of str&gt;</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rcx</td>
<td></td>
</tr>
<tr>
<td>%rdx</td>
<td></td>
</tr>
</tbody>
</table>