CS107, Lecture 10
Introduction to Assembly

Reading: B&O 3.1-3.4
What is Assembly Code?

• Computers execute "machine code," which is a sequence of bytes that encode low-level operations for manipulating data, managing memory, read and write from storage, and communicate with networks.

• The "assembly code" for a computer is a textual representation of the machine code giving the individual instructions to the underlying machine.
What is Assembly Code?

- `gcc` generates assembly code from C code
- Assembly is raw — there is no type checking, and the instructions are simple. It is unique to the type of processor (e.g., the assembly for your computer cannot run on your phone)
- Humans can write assembly (and, in fact, in the early days of computing they had to write assembly), but it is more productive to be able to read and understand what the compiler produces, than to write it by hand.
- `gcc` is almost always going to produce better optimized code than a human could, and understanding what the compiler produces is important.
x86 Assembly

• The Intel-based computers we use are direct descendants of Intel's 16-bit, 1978 processor with the name 8086.
• Intel has taken a strict backwards-compatibility approach to new processors, and their 32- and 64-bit processors have built upon the original 8086 Assembly code.
• These days, when we learn x86 assembly code, we have to keep this history in mind. Naming of "registers," for example, has historical roots, so bear with it.
Before we look at some assembly code, let's talk about some things that have been hidden from us when writing C code.

Machine code is based on the "instruction set architecture" (ISA), which defines the behavior and layout of the system. Behavior is defined as if instructions are run one after the other, and memory appears as a very large byte array.
Machine-Level Code

- New things that have been hidden:
  - The *program counter* (PC), called "%rip" indicates the address of the next instruction ("r"egister "i"nstruction "p"ointer"). We cannot modify this directly.
  - The "register file" contains 16 named locations that store 64-bit values.

Registers are the fastest memory on your computer. They *are not* in main memory, and *do not* have addresses. You cannot pass a pointer to a register, but a pointer may *hold* a register as its value.

- Registers can hold addresses, or integer data. Some registers are used to keep track of your program's state, and others hold temporary data.
- Registers are used for arithmetic, local variables, and return values for functions.
- The condition code registers hold status information about the most recently executed arithmetic or logical instruction. These are used to control program flow — e.g., if the result of an addition is negative, exit a loop.
- There are vector registers, which hold integer or floating point values.
Machine-Level Code

- Unlike C, there is no model of different data types, and memory is simply a large, byte-addressable array.

- There is no distinction between signed and unsigned integers, between different types of pointers, or even between pointers and integers.

- A single machine instruction performs only a very elementary operation. For example:
  - there is an instruction to add two numbers in registers. That's all the instruction does.
  - there is an instruction that transfers data between a register and memory.
  - there is an instruction that conditionally branches to a new instruction address.

- Often, one C statement generates multiple assembly code instructions.
• 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
• Live Session

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

- Overview: GCC and Assembly 10
- Demo: Looking at an executable 13
- Registers and The Assembly Level of Abstraction 24
- The `mov` Instruction 35
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** 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.
• **Overview:** GCC and Assembly  
• **Demo:** Looking at an executable  
• Registers and The Assembly Level of Abstraction  
• The `mov` Instruction  

```
cp -r /afs/ir/class/cs107/lecture-code/lect10 .
```
Demo: Looking at an Executable (objdump -d)
What does this look like in assembly?
Our First Assembly

```c
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
Here is our first assembly program:

```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
              retq
```
Our First Assembly

0000000000401136 <sum_array>:

401136: b8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 mov $0x0,%eax
40113b: ba 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 mov $0x0,%edx
401140: 39 f0 cm
401142: 7d 0b jg
401144: 48 63 c8 mo
401147: 03 14 8f ad (%rdi,%rcx,4),%edx
40114f <sum_array+0x19> %eax,%rcx
401151: 83 c0 01 add $0x1,%eax
401154: 89 d0 mov %edx,%eax
401156: 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.

- 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:

- Hexadecimal instructions, representing binary as read by the computer. Different instructions may be different byte lengths.

```
  mov    $0x0, %eax
  mov    %esi, %eax
  jge    40114f
  movslq %eax, %rcx
  dd     (%rdi,%rcx,4),%edx
  dd     $0x1,%eax
  mov    40114a:
  83ebc001f1a
  be
  c8
  4863c8
  03148f
  83c001
  eb f1
  89d0
  c3
```
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
Our First Assembly

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>401136</td>
<td>b8 00 00 00 00</td>
<td>mov $0x0,%eax</td>
</tr>
<tr>
<td>40113b</td>
<td>ba 00 00 00 00</td>
<td>mov $0x0,%edx</td>
</tr>
<tr>
<td>401140</td>
<td>39 f0</td>
<td>cmp %esi,%eax</td>
</tr>
<tr>
<td>401142</td>
<td>7d 0b</td>
<td>jge 40114f &lt;sum_array+0x19&gt;</td>
</tr>
<tr>
<td>401144</td>
<td>48 63 c8</td>
<td>movslq %eax,%rcx</td>
</tr>
<tr>
<td>401147</td>
<td>03 14 8f</td>
<td>add (%rdi,%rcx,4),%edx</td>
</tr>
<tr>
<td>40114a</td>
<td>83 c0 01</td>
<td>add $0x1,%eax</td>
</tr>
<tr>
<td>40114d</td>
<td>eb f1</td>
<td>jmp 401140 &lt;sum_array+0xa&gt;</td>
</tr>
<tr>
<td>40114f</td>
<td>89 d0</td>
<td>mov %edx,%eax</td>
</tr>
<tr>
<td>401151</td>
<td>c3</td>
<td>retq</td>
</tr>
</tbody>
</table>

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

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

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 401149 <sum_array+0xa>
mov %edx,%eax

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

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

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
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

%[name] means a register, a storage location on the CPU (e.g. eax here).
# 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 .
```
• 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

%rax
%rbx
%rcx
%rdx
%rsi
%rdi
%rbp
%rsp
%r8
%r9
%r10
%r11
%r12
%r13
%r14
%r15
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
GCC And Assembly

• 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.
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 7
• **Demo:** Looking at an executable 11
• Registers and The Assembly Level of Abstraction 24
• **The mov Instruction** 35
• Live Session 57

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

```
mov src, dst
```

The `src` and `dst` can each be one of:
- Immediate (constant value, like a number) *(only `src`)*
  
  
  \(\text{Direct address } 0x6005c0\)
- Register
  
  \(\text{Direct address } %rbx\)
- Memory Location *(at most one of `src`, `dst`)*

\(0x104\)
Operand Forms: Immediate

`mov $0x104,_____`  

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

`mov %rbx,___`

Copy the value in register %rbx into some destination.

`mov ___,%rbx`

Copy the value from some source into register %rbx.
Operand Forms: Absolute Addresses

`mov 0x104, ____`

Copy the value at address 0x104 into some destination.

`mov ____ , 0x104`

Copy the value from some source into the memory at address 0x104.
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

**mov (%rbx), _____**

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.
Operand Forms: Base + Displacement

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

\[ \text{mov} \quad 0x10(\%rax), \_\_\_\_\_\_\_\_\_\_\_\_ \]

Copy the value from some source into the memory at the address (0x10 plus what is stored in register %rax).

\[ \text{mov} \quad \_\_\_\_\_\_\_\_\_\_, 0x10(\%rax) \]
Operand Forms: Indexed

mov (%rax,%rdx), ___________

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).
Operand Forms: Indexed

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

\begin{align*}
\text{mov} & \quad \texttt{0x10(\%rax,\%rdx)}, \quad \text{_____} \\
\text{mov} & \quad \text{_____}, \texttt{0x10(\%rax,\%rdx)}
\end{align*}

Copy the value from some source into the memory at the address which is (the sum of \texttt{0x10 plus} the values in registers \texttt{%rax and %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 $Imm + R[r_b] + R[r_i]$

- **Displacement**: positive or negative constant (if missing, = 0)
- **Base**: register (if missing, = 0)
- **Index**: register (if missing, = 0)
Operand Forms: Scaled Indexed

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

`mov (,%rdx,4),_____`  
The scaling factor (e.g. 4 here) must be hardcoded to be either 1, 2, 4 or 8.

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

`mov _____,(,%rdx,4)`
Operand Forms: Scaled Indexed

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

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

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).
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.

```
mov (%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).*

```
mov __________,(%rax,%rdx,2)
```
Operand Forms: Scaled Indexed

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

\[
\text{mov} \quad 0x4(%rax,%rdx,2),_____
\]

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

\[
\text{mov} \quad ______,0x4(%rax,%rdx,2)
\]
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>$r_1$</td>
<td>$R[r_1]$</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>$(r_1)$</td>
<td>$M[R[r_1]]$</td>
<td>Indirect</td>
</tr>
<tr>
<td>Memory</td>
<td>$Imm(r^n)$</td>
<td>$M[Imm + R[r^n]]$</td>
<td>Base + displacement</td>
</tr>
<tr>
<td>Memory</td>
<td>$(r^n, r_#)$</td>
<td>$M[R[r^n] + R[r_#]]$</td>
<td>Indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>$(r^n, r_#, s)$</td>
<td>$M[R[r^n] + R[r_#] + R[r_#] . s]$</td>
<td>Indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>$(r^n, r_#, s)$</td>
<td>$M[R[r^n] . s]$</td>
<td>Scaled indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>$Imm(r^n, r_#, s)$</td>
<td>$M[Imm + R[r_#] . s]$</td>
<td>Scaled indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>$(r^n, r_#, s)$</td>
<td>$M[R[r^n] + R[r_#] . s]$</td>
<td>Scaled indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>$Imm(r^n, r_#, s)$</td>
<td>$M[Imm + R[r^n] + R[r_#] . s]$</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 $0x42,0xfc(%rcx,4)

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

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

<table>
<thead>
<tr>
<th>Displacement</th>
<th>Base</th>
<th>Index</th>
<th>Scale</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1,2,4,8)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Goals of indirect addressing: C

Why are there so many forms of indirect addressing?

We see these indirect addressing paradigms in C as well!
Our First Assembly

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

We’re 1/4\textsuperscript{th} of the way to understanding assembly!

**What looks understandable right now?**

Some notes:
- Registers store addresses and values
- `mov src, dst` **copies** value into dst
- `sizeof(int)` is 4
- Instructions executed sequentially

00000000004005b6 <sum_array>:

```
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
```

We’ll come back to this example in future lectures!
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)
Assembly code in movies

Trinity saving the world by hacking into the power grid using Nmap Network Scanning

*The Matrix Reloaded*, 2003
Keep a resource guide handy

- [https://web.stanford.edu/class/cs107/resources/x86-64-reference.pdf](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 study abroad:
  - You took LANG 1A
  - Your tools give too much/too little information (a book reference, a rudimentary translator)
  - No one expects you to **speak** the language fluently...
  - ...But the more you internalize, the better you can use tools to **read** the language

Chapter 3, Figures 3.2-3.3 (p. 180-181)
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.

Programmer-generated

idea

→

C code

→

Assembly code

→

Machine code

gcc (compiler+assembler) generated
And that’s it for today!
Extended warmup: Information Synthesis

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
Extended warmup: Information Synthesis

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

Fill in the blank to complete the C code that

1. generates this assembly
2. has this register layout

```
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 generates this assembly and has this register layout.

```c
int x = ...
int *ptr = malloc(...);
...
___???___ = _???_;  *ptr = x;
mov %ecx,(%rax)
```

<val of x> <val of ptr>
Fill in the blank to complete the C code that

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

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

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

| <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. results in this register layout

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

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

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

(assume long y = 3; declared earlier)

<val of num> <val of arr> 3
Fill in the blank to complete the C code that

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

1. generates this assembly
2. has this register layout

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

[val of str]  2

%rcx  %rdx
Fill in the blank to complete the C code that

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

1. generates this assembly
2. has this register layout

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

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

<val of str>  2
%rcx  %rdx
• The below code is the objdump of a C function, `foo`.
  • `foo` keeps its 1\textsuperscript{st} and 2\textsuperscript{nd} parameters are in registers `%rdi` and `%rsi`, respectively.

\begin{verbatim}
 0x4005b6 <foo>    mov (%rdi),%rax
 0x4005b9 <foo+3>  mov (%rsi),%rdx
 0x4005bc <foo+6>  mov %rdx,(%rdi)
 0x4005bf <foo+9>  mov %rax,(%rsi)
\end{verbatim}

1. **What does this function do?**
2. **What C code could have generated this assembly?**
   (Hints: make up C variable names as needed, assume all regs 64-bit)
• The below code is the objdump of a C function, foo.
  • foo keeps its 1\textsuperscript{st} and 2\textsuperscript{nd} parameters are in registers \%rdi and \%rsi, respectively.

\begin{verbatim}
0x4005b6 <foo>  mov   (%rdi),%rax
0x4005b9 <foo+3> mov   (%rsi),%rdx
0x4005bc <foo+6> mov   %rdx,(%rdi)
0x4005bf <foo+9> mov   %rax,(%rsi)
\end{verbatim}