CS107, Lecture 10
Introduction to Assembly

Reading: B&O 3.1-3.4
Attendance

https://forms.gle/jvUDKCbY9FHBfD5n9
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?

- \texttt{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.
- \texttt{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.
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
- Live Session
Lecture Plan

• **Overview:** GCC and Assembly
• **Demo:** Looking at an executable
• Registers and The Assembly Level of Abstraction
• The `mov` Instruction
• Live Session
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.
Lecture Plan

• **Overview**: GCC and Assembly
• **Demo**: Looking at an executable
• Registers and The Assembly Level of Abstraction
• The `mov` Instruction
• Live Session
Examining the Assembly

objdump -d
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

make objdump -d sum

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

This is the name of the function (same as C) and the memory address where the code for this function starts.

```
00000000000401136 <sum_array>:
    b8 00 00 00 00 00 00 00
        mov $0x0,%eax
    ba 00 00 00 00 00 00 00
        mov $0x0,%edx
    p     %esi,%eax
    vslq %eax,%rcx
    39 f0 cm
    7d 0b jg
    48 63 c8 mo
    03 14 8f ad
    (%rdi,%rcx,4),%edx
    83 c0 01
    eb f1
    89 d0
    c3
40114b:
    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.
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+0xda>
mov %edx,%eax
retq
```
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

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

```assembly
000000000000401136  <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 03 14 8f  c8  movslq  %eax,%rcx
  401147:  83 c0 01            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

00000000000401136 <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

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

000000000000401136 <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   401149 <sum_array+0xa>
  40114f: 89 d0                    mov   %edx,%eax

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

000000000000401136 <sum_array>:

000000000000401136:  b8 00 00 00 00 00
00000000000040113b:  ba 00 00 00 00 00
000000000000401140:  39 f0
000000000000401142:  7d 0b
000000000000401144:  48 63 c8
000000000000401147:  03 14 8f
00000000000040114a:  83 c0 01
00000000000040114d:  eb f1
00000000000040114f:  89 d0
000000000000401151:  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

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

%[name] means a register, a storage location on the CPU (e.g. edx here).
Lecture Plan

• **Overview:** GCC and Assembly
• **Demo:** Looking at an executable
• **Registers and The Assembly Level of Abstraction**
• The **mov** instruction
• 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
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 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>
<tbody>
<tr>
<td><code>int sum = x + y;</code></td>
<td>1) Copy x into register 1</td>
</tr>
<tr>
<td></td>
<td>2) Copy y into register 2</td>
</tr>
<tr>
<td></td>
<td>3) Add register 2 to register 1</td>
</tr>
<tr>
<td></td>
<td>4) Write register 1 to memory for sum</td>
</tr>
</tbody>
</table>
• 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**
- Live Session
The \texttt{mov} instruction \textit{copies} bytes from one place to another; it is similar to the assignment operator (\texttt{=}) in C.

\begin{equation*}
\texttt{mov \ src,\ dst}
\end{equation*}

The \texttt{src} and \texttt{dst} can each be one of:

\begin{itemize}
\item Immediate (constant value, like a number) \textit{(only src)}
\item Register
\item Memory Location \textit{(at most one of src, dst)}
\end{itemize}
Operand Forms: Immediate

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

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.
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.

\[ \text{mov} \ (\%rbx), \_\_\_\_ \]

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

\[ \_\_\_\_, (\%rbx) \]
Operand Forms: Base + Displacement

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

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

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

\[
\text{mov} \quad \text{__________}, 0x10(\%rax)
\]
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. \texttt{mov} $0x42, (%rax)$
2. \texttt{mov} 4(%rax), %rcx
3. \texttt{mov} 9(%rax, %rdx), %rcx

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

- \textbf{Displacement}: positive or negative constant (if missing, = 0)
- \textbf{Base}: register (if missing, = 0)
- \textbf{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.

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

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).
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.

```
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 (\texttt{0x4 plus} the value in register \texttt{\%rax} plus 2 times the value in register \texttt{\%rdx}) into some destination.

\texttt{mov 0x4(\%rax,\%rdx,2),_____}

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

\texttt{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

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

**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>$Imm(r^*)$</td>
<td>$M[Imm + R[r^*]]$</td>
<td>Base + displacement</td>
</tr>
<tr>
<td>Memory</td>
<td>$(r^*, r_#)$</td>
<td>$M[R[r^*] + R[r_#]]$</td>
<td>Indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>$Imm(r^*, r_#)$</td>
<td>$M[Imm + R[r^*] + R[r_#]]$</td>
<td>Indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>$(r^*, r_#, s)$</td>
<td>$M[R[r_#] \cdot s]$</td>
<td>Scaled indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>$Imm(r^*, r_#, s)$</td>
<td>$M[Imm + R[r_#] \cdot s]$</td>
<td>Scaled indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>$(r^*, r_#, s)$</td>
<td>$M[R[r^*] + R[r_#] \cdot s]$</td>
<td>Scaled indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>$Imm(r^*, r_#, s)$</td>
<td>$M[Imm + R[r^*] + R[r_#] \cdot s]$</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. \texttt{mov} $0x42,0x\text{fc}(,\%rcx,4)$

2. \texttt{mov} (%rax,%rdx,4),%rbx

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

<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` \textit{copies} value into `dst`
- `sizeof(int)` is 4
- Instructions executed sequentially

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

• 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
Why are we reading assembly?

- 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.

**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.
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 -> 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];
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(...);
...
___???___ = _???_.
```

```assembly
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(...);
...
```

```assembly
___???___ = _???_; *ptr = x;
```

```assembly
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 = ____???____;
```

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

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

<table>
<thead>
<tr>
<th>&lt;val of num&gt;</th>
<th>3</th>
<th>&lt;val of arr&gt;</th>
<th>%rax</th>
<th>%rcx</th>
<th>%rdi</th>
</tr>
</thead>
</table>

"???

68
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 = ???;
```

```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>
%rax %rdi
%rcx
3
```
Fill in the blank to complete the C code that generates this assembly

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

1. generates this assembly
2. has this register layout

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

[val of str] [2] [val of %rdx]
3. Extra Practice

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';
```

<val of str>  2
%rcx %rdx
The below code is the objdump of a C function, foo.

- foo keeps its 1st and 2nd parameters are in registers %rdi and %rsi, respectively.

```
0x4005b6 <foo>    mov  (%rdi),%rax
0x4005b9 <foo+3>  mov  (%rsi),%rdx
0x4005bc <foo+6>  mov  %rdx,(%rdi)
0x4005bf <foo+9>  mov  %rax,(%rsi)
```

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.

```c
void foo(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

```plaintext
0x4005b6 <foo>   mov   (%rdi),%rax
0x4005b9 <foo+3> mov   (%rsi),%rdx
0x4005bc <foo+6> mov   %rdx,(%rdi)
0x4005bf <foo+9> mov   %rax,(%rsi)
```
Lecture Plan

• Recap: mov so far
• Data and Register Sizes
• The lea Instruction
• Logical and Arithmetic Operations
• Practice: Reverse Engineering

Reference Sheet:
https://web.stanford.edu/class/archive/cs/cs107/cs107.1248/guide/x86-64.html
See more guides on Resources page of course website!
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)*

- Register

- Memory Location *(at most one of src, dst)*
## Memory Location Syntax

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x104</td>
<td>Address 0x104 (no $)</td>
</tr>
<tr>
<td>(%rax)</td>
<td>What’s in %rax</td>
</tr>
<tr>
<td>4(%rax)</td>
<td>What’s in %rax, plus 4</td>
</tr>
<tr>
<td>(%rax, %rdx)</td>
<td>Sum of what’s in %rax and %rdx</td>
</tr>
<tr>
<td>4(%rax, %rdx)</td>
<td>Sum of values in %rax and %rdx, plus 4</td>
</tr>
<tr>
<td>(, %rcx, 4)</td>
<td>What’s in %rcx, times 4 (multiplier can be 1, 2, 4, 8)</td>
</tr>
<tr>
<td>(%rax, %rcx, 2)</td>
<td>What’s in %rax, plus 2 times what’s in %rcx</td>
</tr>
<tr>
<td>8(%rax, %rcx, 2)</td>
<td>What’s in %rax, plus 2 times what’s in %rcx, plus 8</td>
</tr>
</tbody>
</table>
# 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>$r_i$</td>
<td>$R[r_i]$</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>$(r_i)$</td>
<td>$M[R[r_i]]$</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>$Imm(r^n, r#)$</td>
<td>$M[Imm + R[r^n] + R[r#]]$</td>
<td>Indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>$(r^n, r#, s)$</td>
<td>$M[R[r#] \cdot s]$</td>
<td>Scaled indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>$Imm(r^n, r#, s)$</td>
<td>$M[Imm + R[r#] \cdot s]$</td>
<td>Scaled indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>$(r^n, r#, s)$</td>
<td>$M[R[r^n] + R[r#] \cdot s]$</td>
<td>Scaled indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>$Imm(r^n, r#, s)$</td>
<td>$M[Imm + R[r^n] + R[r#] \cdot s]$</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.”
Lecture Plan

• Recap: mov so far
• Data and Register Sizes
• The lea Instruction
• Logical and Arithmetic Operations
• Practice: Reverse Engineering

Reference Sheet:
https://web.stanford.edu/class/archive/cs/cs107/cs107.1248/guide/x86-64.html
See more guides on Resources page of course website!
Data Sizes

Data sizes in assembly have slightly different terminology to get used to:

• A **byte** is 1 byte.
• A **word** is 2 bytes.
• A **double word** is 4 bytes.
• A **quad word** is 8 bytes.

Assembly instructions can have suffixes to refer to these sizes:

• b means **byte**
• w means **word**
• l means **double word**
• q means **quad word**
## Register Sizes

<table>
<thead>
<tr>
<th>Bit:</th>
<th>63</th>
<th>31</th>
<th>15</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>%eax</td>
<td>%ax</td>
<td>%al</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rbx</td>
<td>%ebx</td>
<td>%bx</td>
<td>%bl</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rcx</td>
<td>%ecx</td>
<td>%cx</td>
<td>%cl</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rdx</td>
<td>%edx</td>
<td>%dx</td>
<td>%dl</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rsi</td>
<td>%esi</td>
<td>%si</td>
<td>%sil</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rdi</td>
<td>%edi</td>
<td>%di</td>
<td>%dil</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Register Sizes

<table>
<thead>
<tr>
<th>Bit:</th>
<th>63</th>
<th>31</th>
<th>15</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>%r8</td>
<td>%r8d</td>
<td>%r8w</td>
<td>%r8b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%r9</td>
<td>%r9d</td>
<td>%r9w</td>
<td>%r9b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%r10</td>
<td>%r10d</td>
<td>%r10w</td>
<td>%r10b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%r11</td>
<td>%r11d</td>
<td>%r11w</td>
<td>%r11b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rsp</td>
<td>%esp</td>
<td>%sp</td>
<td>%spl</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rbp</td>
<td>%ebp</td>
<td>%bp</td>
<td>%bp1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Register Sizes

<table>
<thead>
<tr>
<th>Bit</th>
<th>63</th>
<th>31</th>
<th>15</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>%r12</td>
<td>%r12d</td>
<td>%r12w</td>
<td>%r12b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%r13</td>
<td>%r13d</td>
<td>%r13w</td>
<td>%r13b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%r14</td>
<td>%r14d</td>
<td>%r14w</td>
<td>%r14b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%r15</td>
<td>%r15d</td>
<td>%r15w</td>
<td>%r15b</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Some registers take on special responsibilities during program execution.

- `%rax` stores the return value
- `%rdi` stores the first parameter to a function
- `%rsi` stores the second parameter to a function
- `%rdx` stores the third parameter to a function
- `%rip` stores the address of the next instruction to execute
- `%rsp` stores the address of the current top of the stack

Reference Sheet:
https://web.stanford.edu/class/archive/cs/cs107/cs107.1248/guide/x86-64.html
See more guides on Resources page of course website!
mov Variants

• **mov** can take an optional suffix (b,w,l,q) that specifies the size of data to move: movb, movw, movl, movq

• **mov** only updates the specific register bytes or memory locations indicated.
  • **Exception:** **movl** writing to a register will also set high order 4 bytes to 0.
Practice: mov And Data Sizes

For each of the following mov instructions, determine the appropriate suffix based on the operands (e.g. movb, movw, movl or movq).

1. mov___ %eax, (%rsp)
2. mov___ (%rax), %dx
3. mov___ $0xff, %bl
4. mov___ (%rsp,%rdx,4),%dl
5. mov___ (%rdx), %rax
6. mov___ %dx, (%rax)
Practice: mov And Data Sizes

For each of the following mov instructions, determine the appropriate suffix based on the operands (e.g. **movb**, **movw**, **movl** or **movq**).

1. movl %eax, (%rsp)
2. movw (%rax), %dx
3. movb $0xff, %bl
4. movb (%rsp,%rdx,4),%dl
5. movq (%rdx), %rax
6. movw %dx, (%rax)
mov

• The **movabsq** instruction is used to write a 64-bit Immediate (constant) value.
• The regular **movq** instruction can only take 32-bit immediates.
• 64-bit immediate as source, only register as destination.

```assembly
movabsq $0x0011223344556677, %rax
```
• There are two mov instructions that can be used to copy a smaller source to a larger destination: movz and movs.

• movz fills the remaining bytes with zeros

• movs fills the remaining bytes by sign-extending the most significant bit in the source.

• The source must be from memory or a register, and the destination is a register.
## movz and movs

MOVZ $S, $R  \quad R \leftarrow \text{ZeroExtend}(S)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>movzbw</td>
<td>Move zero-extended byte to word</td>
</tr>
<tr>
<td>movzbl</td>
<td>Move zero-extended byte to double word</td>
</tr>
<tr>
<td>movzwl</td>
<td>Move zero-extended word to double word</td>
</tr>
<tr>
<td>movzbq</td>
<td>Move zero-extended byte to quad word</td>
</tr>
<tr>
<td>movzwq</td>
<td>Move zero-extended word to quad word</td>
</tr>
</tbody>
</table>
### movz and movs

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>movsbw</td>
<td>Move sign-extended byte to word</td>
</tr>
<tr>
<td>movsbl</td>
<td>Move sign-extended byte to double word</td>
</tr>
<tr>
<td>movswl</td>
<td>Move sign-extended word to double word</td>
</tr>
<tr>
<td>movsbq</td>
<td>Move sign-extended byte to quad word</td>
</tr>
<tr>
<td>movswq</td>
<td>Move sign-extended word to quad word</td>
</tr>
<tr>
<td>movslq</td>
<td>Move sign-extended double word to quad word</td>
</tr>
</tbody>
</table>
| cltq        | Sign-extend %eax to %rax  
%rax <- SignExtend(%eax) |
Lecture Plan

• Recap: mov so far
• Data and Register Sizes
• The lea Instruction
• Logical and Arithmetic Operations
• Practice: Reverse Engineering

Reference Sheet:
https://web.stanford.edu/class/archive/cs/cs107/cs107.1248/guide/x86-64.html
See more guides on Resources page of course website!
The `lea` instruction copies an “effective address” from one place to another.

```
lea src, dst
```

Unlike `mov`, which copies data at the address `src` to the destination, `lea` copies the value of `src itself` to the destination.

The syntax for the destinations is the same as `mov`. The difference is how it handles the `src`. 
### lea vs. mov

<table>
<thead>
<tr>
<th>Operands</th>
<th>mov Interpretation</th>
<th>lea Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(6(%rax), %rdx)</td>
<td>Go to the address ((6 + \text{what’s in }%rax)), and copy data there into %rdx</td>
<td>Copy (6 + \text{what’s in }%rax) into %rdx</td>
</tr>
</tbody>
</table>
# lea vs. mov

<table>
<thead>
<tr>
<th>Operands</th>
<th>mov Interpretation</th>
<th>lea Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>6(%rax), %rdx</td>
<td>Go to the address (6 + what’s in %rax), and copy data there into %rdx</td>
<td>Copy 6 + what’s in %rax into %rdx.</td>
</tr>
<tr>
<td>(%rax, %rcx), %rdx</td>
<td>Go to the address (what’s in %rax + what’s in %rcx) and copy data there into %rdx</td>
<td>Copy (what’s in %rax + what’s in %rcx) into %rdx.</td>
</tr>
</tbody>
</table>
## lea vs. mov

<table>
<thead>
<tr>
<th>Operands</th>
<th>mov Interpretation</th>
<th>lea Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>6(%rax), %rdx</td>
<td>Go to the address (6 + what’s in %rax), and copy data there into %rdx</td>
<td>Copy 6 + what’s in %rax into %rdx.</td>
</tr>
<tr>
<td>(%rax, %rcx), %rdx</td>
<td>Go to the address (what’s in %rax + what’s in %rcx) and copy data there into %rdx</td>
<td>Copy (what’s in %rax + what’s in %rcx) into %rdx.</td>
</tr>
<tr>
<td>(%rax, %rcx, 4), %rdx</td>
<td>Go to the address (%rax + 4 * %rcx) and copy data there into %rdx.</td>
<td>Copy (%rax + 4 * %rcx) into %rdx.</td>
</tr>
</tbody>
</table>
# lea vs. mov

<table>
<thead>
<tr>
<th>Operands</th>
<th>mov Interpretation</th>
<th>lea Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>6(%rax), %rdx</td>
<td>Go to the address (6 + what’s in %rax), and copy data there into %rdx</td>
<td>Copy 6 + what’s in %rax into %rdx.</td>
</tr>
<tr>
<td>(%rax, %rcx), %rdx</td>
<td>Go to the address (what’s in %rax + what’s in %rcx) and copy data there into %rdx</td>
<td>Copy (what’s in %rax + what’s in %rcx) into %rdx.</td>
</tr>
<tr>
<td>(%rax, %rcx, 4), %rdx</td>
<td>Go to the address (%rax + 4 * %rcx) and copy data there into %rdx.</td>
<td>Copy (%rax + 4 * %rcx) into %rdx.</td>
</tr>
<tr>
<td>7(%rax, %rax, 8), %rdx</td>
<td>Go to the address (7 + %rax + 8 * %rax) and copy data there into %rdx.</td>
<td>Copy (7 + %rax + 8 * %rax) into %rdx.</td>
</tr>
</tbody>
</table>

Unlike `mov`, which copies data at the address `src` to the destination, `lea` copies the value of `src` itself to the destination.
Lecture Plan

• Recap: mov so far
• Data and Register Sizes
• The lea Instruction
• Logical and Arithmetic Operations
• Practice: Reverse Engineering

Reference Sheet:
https://web.stanford.edu/class/archive/cs/cs107/cs107.1248/guide/x86-64.html
See more guides on Resources page of course website!
Unary Instructions

The following instructions operate on a single operand (register or memory):

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inc D</td>
<td>D ← D + 1</td>
<td>Increment</td>
</tr>
<tr>
<td>dec D</td>
<td>D ← D - 1</td>
<td>Decrement</td>
</tr>
<tr>
<td>neg D</td>
<td>D ← -D</td>
<td>Negate</td>
</tr>
<tr>
<td>not D</td>
<td>D ← ~D</td>
<td>Complement</td>
</tr>
</tbody>
</table>

Examples:

- incq 16(%rax)
- dec %rdx
- not %rcx
Binary Instructions

The following instructions operate on two operands (both can be register or memory, source can also be immediate). Both cannot be memory locations. Read it as, e.g. “Subtract S from D”:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add S, D</td>
<td>D ← D + S</td>
<td>Add</td>
</tr>
<tr>
<td>sub S, D</td>
<td>D ← D - S</td>
<td>Subtract</td>
</tr>
<tr>
<td>imul S, D</td>
<td>D ← D * S</td>
<td>Multiply</td>
</tr>
<tr>
<td>xor S, D</td>
<td>D ← D ^ S</td>
<td>Exclusive-or</td>
</tr>
<tr>
<td>or S, D</td>
<td>D ← D</td>
<td>S</td>
</tr>
<tr>
<td>and S, D</td>
<td>D ← D &amp; S</td>
<td>And</td>
</tr>
</tbody>
</table>

Examples:

- addq %rcx, (%rax)
- xorq $16, (%rax, %rdx, 8)
- subq %rdx, 8(%rax)
Large Multiplication

• Multiplying 64-bit numbers can produce a 128-bit result. How does x86-64 support this with only 64-bit registers?

• If you specify two operands to **imul**, it multiplies them together and truncates until it fits in a 64-bit register.

  \[
  \text{imul } S, D \quad D \leftarrow D \times S
  \]

• If you specify one operand, it multiplies that by `%rax`, and splits the product across 2 registers. It puts the high-order 64 bits in `%rdx` and the low-order 64 bits in `%rax`.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>imulq S</td>
<td>R[%rdx]:R[%rax] ← S x R[%rax]</td>
<td>Signed full multiply</td>
</tr>
<tr>
<td>mulq S</td>
<td>R[%rdx]:R[%rax] ← S x R[%rax]</td>
<td>Unsigned full multiply</td>
</tr>
</tbody>
</table>
• **Terminology:** *dividend / divisor = quotient + remainder*

• **x86-64** supports dividing up to a 128-bit value by a 64-bit value.

• The high-order 64 bits of the dividend are in `%rdx`, and the low-order 64 bits are in `%rax`. The divisor is the operand to the instruction.

• The quotient is stored in `%rax`, and the remainder in `%rdx`. 

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
<th>Description</th>
</tr>
</thead>
</table>
| `idivq S` | \( R[\%rdx] \leftarrow R[\%rdx]:R[\%rax] \mod S; \)
\( R[\%rax] \leftarrow R[\%rdx]:R[\%rax] \div S \) | Signed divide |
| `divq S` | \( R[\%rdx] \leftarrow R[\%rdx]:R[\%rax] \mod S; \)
\( R[\%rax] \leftarrow R[\%rdx]:R[\%rax] \div S \) | Unsigned divide |
Division and Remainder

**Terminology:** dividend / divisor = quotient + remainder

- The high-order 64 bits of the dividend are in %rdx, and the low-order 64 bits are in %rax. The divisor is the operand to the instruction.

- Most division uses only 64-bit dividends. The **cqto** instruction sign-extends the 64-bit value in %rax into %rdx to fill both registers with the dividend, as the division instruction expects.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>idivq S</td>
<td>R[%rdx] ← R[%rdx]:R[%rax] mod S; R[%rax] ← R[%rdx]:R[%rax] ÷ S</td>
<td>Signed divide</td>
</tr>
<tr>
<td>divq S</td>
<td>R[%rdx] ← R[%rdx]:R[%rax] mod S; R[%rax] ← R[%rdx]:R[%rax] ÷ S</td>
<td>Unsigned divide</td>
</tr>
<tr>
<td>cqto</td>
<td>R[%rdx]:R[%rax] ← SignExtend(R[%rax])</td>
<td>Convert to oct word</td>
</tr>
</tbody>
</table>
Shift Instructions

The following instructions have two operands: the shift amount \( k \) and the destination to shift, \( D \). \( k \) can be either an immediate value, or the byte register \( %cl \) (and only that register!)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sal k, D</code></td>
<td>( D \leftarrow D \ll k )</td>
<td>Left shift</td>
</tr>
<tr>
<td><code>shl k, D</code></td>
<td>( D \leftarrow D \ll k )</td>
<td>Left shift (same as <code>sal</code>)</td>
</tr>
<tr>
<td><code>sar k, D</code></td>
<td>( D \leftarrow D \gg_A k )</td>
<td>Arithmetic right shift</td>
</tr>
<tr>
<td><code>shr k, D</code></td>
<td>( D \leftarrow D \gg_L k )</td>
<td>Logical right shift</td>
</tr>
</tbody>
</table>

Examples:

- `shll $3,(%rax)`
- `shr1 %cl,%(rax,%rdx,8)`
- `sar1 $4,8(%rax)`
• When using `%cl`, the width of what you are shifting determines what portion of `%cl` is used.

• For `w` bits of data, it looks at the low-order `log2(w)` bits of `%cl` to know how much to shift.
  • If `%cl = 0xff`, then: `shlb` shifts by 7 because it considers only the low-order `log2(8) = 3` bits, which represent 7. `shlw` shifts by 15 because it considers only the low-order `log2(16) = 4` bits, which represent 15.
Lecture Plan

• Recap: mov so far
• Data and Register Sizes
• The lea Instruction
• Logical and Arithmetic Operations
• Practice: Reverse Engineering

Reference Sheet:
https://web.stanford.edu/class/archive/cs/cs107/cs107.1248/guide/x86-64.html
See more guides on Resources page of course website!
Let’s pull these commands together and see how some C code might be translated to assembly.

Compiler Explorer is a handy website that lets you quickly write C code and see its assembly translation. Let’s check it out!

https://godbolt.org/z/WPzz6G4a9
/∗ Returns the sum of x and the first element in arr ∗
int add_to_first(int x, int arr[]) {
    int sum = x;
    sum += arr[0];
    return sum;
}

---------

add_to_first:
  movl %edi, %eax
  addl (%rsi), %eax
  ret
// Returns x/y, stores remainder in location stored in remainder_ptr
long full_divide(long x, long y, long *remainder_ptr) {
    long quotient = x / y;
    long remainder = x % y;
    *remainder_ptr = remainder;
    return quotient;
}

full_divide:
    movq %rdi, %rax
    movq %rdx, %rcx
cqto
    idivq %rsi
    movq %rdx, (%rcx)
    ret
Assembly Exercise 1

000000000040116e <sum_example1>:
  40116e: 8d 04 37           lea (%rdi,%rsi,1),%eax
  401171:  c3                 retq

Which of the following is most likely to have generated the above assembly?

// A)
void sum_example1() {
    int x;
    int y;
    int sum = x + y;
}

// B)
int sum_example1(int x, int y) {
    return x + y;
}

// C)
void sum_example1(int x, int y) {
    int sum = x + y;
}
.int sum_example2(int arr[]) {
    int sum = 0;
    sum += arr[0];
    sum += arr[3];
    sum -= arr[6];
    return sum;
}

What location or value in the assembly above represents the C code’s sum variable?

%eax
int sum_example2(int arr[]) {
    int sum = 0;
    sum += arr[0];
    sum += arr[3];
    sum -= arr[6];
    return sum;
}

What location or value in the assembly code above represents the C code’s 6 (as in arr[6])?

0x18
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 1/2 of the way to understanding assembly!

What looks understandable right now?
A Note About Operand Forms

• Many instructions share the same address operand forms that mov uses.
  • Eg. 7(%rax, %rcx, 2).

• These forms work the same way for other instructions, e.g. sub:
  • sub 8(%rax,%rdx),%rcx -> Go to 8 + %rax + %rdx, subtract what’s there from %rcx

• The exception is lea:
  • It interprets this form as just the calculation, not the dereferencing
  • lea 8(%rax,%rdx),%rcx -> Calculate 8 + %rax + %rdx, put it in %rcx
Shift Amount

- When using `%cl`, the width of what you are shifting determines what portion of `%cl` is used.
- For `w` bits of data, it looks at the low-order $\log_2(w)$ bits of `%cl` to know how much to shift.
  - If `%cl` = 0xff, then: `shlb` shifts by 7 because it considers only the low-order $\log_2(8) = 3$ bits, which represent 7. `shlw` shifts by 15 because it considers only the low-order $\log_2(16) = 4$ bits, which represent 15.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sal</code> <code>k</code>, <code>D</code></td>
<td><code>D ← D &lt;&lt; k</code></td>
<td>Left shift</td>
</tr>
<tr>
<td><code>shl</code> <code>k</code>, <code>D</code></td>
<td><code>D ← D &lt;&lt; k</code></td>
<td>Left shift (same as <code>sal</code>)</td>
</tr>
<tr>
<td><code>sar</code> <code>k</code>, <code>D</code></td>
<td><code>D ← D &gt;&gt;_A k</code></td>
<td>Arithmetic right shift</td>
</tr>
<tr>
<td><code>shr</code> <code>k</code>, <code>D</code></td>
<td><code>D ← D &gt;&gt;_L k</code></td>
<td>Logical right shift</td>
</tr>
</tbody>
</table>
Division and Remainder

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>idivq S</td>
<td>R[%rdx] ← R[%rdx]:R[%rax] mod S; R[%rax] ← R[%rdx]:R[%rax] / S</td>
<td>Signed divide</td>
</tr>
<tr>
<td>divq S</td>
<td>R[%rdx] ← R[%rdx]:R[%rax] mod S; R[%rax] ← R[%rdx]:R[%rax] / S</td>
<td>Unsigned divide</td>
</tr>
</tbody>
</table>

• **Terminology**: dividend / divisor = quotient + remainder
• **x86-64** supports dividing up to a 128-bit value by a 64-bit value.
• The high-order 64 bits of the dividend are in `%rdx`, and the low-order 64 bits are in `%rax`. The divisor is the operand to the instruction.
• The quotient is stored in `%rax`, and the remainder in `%rdx`. 
Extra Practice

https://godbolt.org/z/hGKPWszq4
int add_to(int x, int arr[], int i)
{
    int sum = __?__;
    sum += arr[__?__];
    return __?__;
}

-------

add_to:
    movslq %edx, %rdx
    movl %edi, %eax
    addl (%rsi,%rdx,4), %eax
    ret
int add_to(int x, int arr[], int i)
{
    int sum = ?
    sum += arr[?]
    return ?
}

-------------
// x in %edi, arr in %rsi, i in %edx
%edx add_to:
movslq %edx, %rdx // sign-extend i into full register
movl %edi, %eax   // copy x into %eax
addl (%rsi,%rdx,4), %eax // add arr[i] to %eax
ret
```c
int add_to(int x, int arr[], int i)
{
    int sum = x;
    sum += arr[i];
    return sum;
}
```

// x in %edi, arr in %rsi, i in %edx
add_to:
   movslq %edx, %rdx       // sign-extend i into full register
   movl %edi, %eax        // copy x into %eax
   addl (%rsi,%rdx,4), %eax  // add arr[i] to %eax
   ret
int elem_arithmetic(int nums[], int y)
{ int z = nums[_?___] * ___?___;
  z -= ___?___;
  z >>= ___?___;
  return ___?___;
}

----------

elem_arithmetic:
  movl %esi, %eax
  imull (%rdi), %eax
  subl 4(%rdi), %eax
  sarl $2, %eax
  addl $2, %eax
  ret
int elem_arithmetic(int nums[], int y)
{
    int z = nums[_?____] * ___?___;
    z -= ___?___;
    z >>= ___?___;
    return ___?___;
}

// nums in %rdi, y in %esi
elem_arithmetic:
    movl %esi, %eax       // copy y into %eax
    imull (%rdi), %eax    // multiply %eax by nums[0]
    subl 4(%rdi), %eax    // subtract nums[1] from %eax
    sarl $2, %eax         // shift %eax right by 2
    addl $2, %eax         // add 2 to %eax
    ret
int elem_arithmetic(int nums[], int y)
{
    int z = nums[0] * y;
    z -= nums[1];
    z >>= 2;
    return z + 2;
}

// nums in %rdi, y in %esi
elem_arithmetic:
    movl %esi, %eax    // copy y into %eax
    imull (%rdi), %eax // multiply %eax by nums[0]
    subl 4(%rdi), %eax // subtract nums[1] from %eax
    sarl $2, %eax      // shift %eax right by 2
    addl $2, %eax      // add 2 to %eax
    ret
long func(long x, long *ptr) {
    *ptr = ___?___ + 1;
    long result = x % ___?___;
    return ___?___;
}

---------

func:
    movq %rdi, %rax
    leaq 1(%rdi), %rcx
    movq %rcx, (%rsi)
    cqto
    idivq %rcx
    movq %rdx, %rax
    ret
long func(long x, long *ptr) {
    *ptr = ___?___ + 1;
    long result = x % ___?___;
    return ___?___;
}

---------
// x in %rdi, ptr in %rsi
func:
    movq %rdi, %rax  // copy x into %rax
    leaq 1(%rdi), %rcx  // put x + 1 into %rcx
    movq %rcx, (%rsi)  // copy %rcx into *ptr
    cqto  // sign-extend x into %rdx
    idivq %rcx  // calculate x / (x + 1)
    movq %rdx, %rax  // copy the remainder into %rax
    ret
long func(long x, long *ptr) {
    *ptr = x + 1;
    long result = x % *ptr; // or x + 1
    return result;
}

// x in %rdi, ptr in %rsi
func:
    movq %rdi, %rax    // copy x into %rax
    leaq 1(%rdi), %rcx // put x + 1 into %rcx
    movq %rcx, (%rsi) // copy %rcx into *ptr
    cqto               // sign-extend x into %rdx
    idivq %rcx        // calculate x / (x + 1)
    movq %rdx, %rax   // copy the remainder into %rax
    ret
long func(long x, long *ptr) {
    *ptr = x + 1;
    long result = x % *ptr; // or x + 1
    return result;
}---------
// x in %rdi, ptr in %rsi
func:
    leaq 1(%rdi), %rcx // put x + 1 into %rcx
    movq %rcx, (%rsi) // copy %rcx into *ptr
    movq %rdi, %rax // copy x into %rax
    cqto // sign-extend x into %rdx
    idivq %rcx // calculate x / (x + 1)
    movq %rdx, %rax // copy the remainder into %rax
    ret
References and Advanced Reading

• References:
  • Stanford guide to x86-64: https://web.stanford.edu/class/cs107/guide/x86-64.html
  • CS107 one-page of x86-64: https://web.stanford.edu/class/cs107/resources/onepage_x86-64.pdf
  • gdbtui: https://beej.us/guide/bggdb/
  • More gdbtui: https://sourceware.org/gdb/onlinedocs/gdb/TUI.html
  • Compiler explorer: https://gcc.godbolt.org

• Advanced Reading:
  • history of x86 instructions: https://en.wikipedia.org/wiki/X86_instruction_listings
  • x86-64 Wikipedia: https://en.wikipedia.org/wiki/X86-64