CS107, Lecture 11
Introduction to Assembly

Reading: B&O 3.1-3.4
Learning Goals

• Learn what assembly language is and why it is important
• Be familiar with the format of human-readable assembly
• Understand the x86 Instruction Set and how it moves data around
Plan For Today

• **Overview:** GCC and Assembly
• **Demo:** Looking at an executable
• Registers and The Assembly Level of Abstraction
• A Brief History
• Our First Assembly
• **Break:** Announcements
• The `mov` instruction
Plan For Today

• **Overview:** GCC and Assembly
• **Demo:** Looking at an executable
• Registers and The Assembly Level of Abstraction
• A Brief History
• Our First Assembly
• **Break:** Announcements
• The `mov` instruction
• **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, variables, 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.
Demo: Looking At An Executable (objdump -d)
Plan For Today

- **Overview**: GCC and Assembly
- **Demo**: Looking at an executable
- **Registers and The Assembly Level of Abstraction**
- A Brief History
- Our First Assembly
- **Break**: Announcements
- The `mov` instruction
Assembly Abstraction

• C abstracts away the low level details of machine code. It lets us work using functions, variables, variable types, etc.
• 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 very machine-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
• 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!
Machine-Level Code

• 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
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>
Plan For Today

• **Overview**: GCC and Assembly
• **Demo**: Looking at an executable
• Registers and The Assembly Level of Abstraction
• **A Brief History**
• Our First Assembly
• **Break**: Announcements
• The **mov** instruction
• 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.

• Intel originally designed their instruction set back in 1978. It has evolved significantly since then, but has aggressively preserved backwards compatibility.

• Originally 16 bit processor -> then 32 -> now 64 bit. This dictated the register sizes (and even register names).
Plan For Today

- **Overview:** GCC and Assembly
- **Demo:** Looking at an executable
- Registers and The Assembly Level of Abstraction
- A Brief History
- **Our First Assembly**
- **Break:** Announcements
- The `mov` instruction
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?
Our First Assembly

000000000004005b6 <sum_array>:

4005b6:   ba 00 00 00 00 00  mov $0x0,%edx
4005bb:   b8 00 00 00 00 00  mov $0x0,%eax
4005c0:   eb 09  jmp 4005cb <sum_array+0x15>
4005c2:   48 63 ca  movslq %edx,%rcx
4005c5:   03 04 8f  add (%rdi,%rcx,4),%eax
4005c8:   83 c2 01  add $0x1,%edx
4005cb:   39 f2  cmp %esi,%edx
4005cd:   7c f3  jl 4005c2 <sum_array+0xc>
4005cf:   f3 c3  repz 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.
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.

```
4005b6:   ba 00 00 00 00 00
4005bb:   b8 00 00 00 00 00
4005c0:   eb 09              jmp 4005cb <sum_array+0x15>
4005c2:   48 63 ca          movslq %edx,%rcx
        03 04 8f          add (%rdi,%rcx,4),%eax
        83 c2 01          add $0x1,%edx
        39 f2          cmp %esi,%edx
        7c f3          jl 4005c2 <sum_array+0xc>
        f3 c3          repz retq
```

```
000000000004005b6 <sum_array>:
  mov $0x0,%edx
  mov $0x0,%eax
  jmp 4005cb <sum_array+0x15>
  movslq %edx,%rcx
  add (%rdi,%rcx,4),%eax
  add $0x1,%edx
  cmp %esi,%edx
  jl 4005c2 <sum_array+0xc>
  repz 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.
Our First Assembly

00000000004005b6 <sum_array>:

4005b6:  ba 00 00 00 00 00  mov $0x0,%edx
4005bb:  b8 00 00 00 00 00  mov $0x0,%eax
4005c0:  eb 09  jmp 4005cb <sum_array+0x15>
4005c2:  48 63 ca  movslq %edx,%rcx
4005c5:  03 04 8f  add (%rdi,%rcx,4),%eax
4005c8:  83 c2 01  add $0x1,%edx
4005cb:  39 f2  cmp %esi,%edx
4005cd:  7c f3  jl 4005c2 <sum_array+0xc>
4005cf:  f3 c3  repz retq
Our First Assembly

00000000004005b6 <sum_array>:
4005b6:   ba 00 00 00 00 00   mov $0x0,%edx
4005bb:   b8 00 00 00 00 00   mov $0x0,%eax
4005c0:   eb 09               jmp 4005cb <sum_array+0x15>
4005c2:   48 63 ca            movslq %edx,%rcx
4005c5:   03 04 8f            add (%rdi,%rcx,4),%eax
4005c8:   83 c2 01            add $0x1,%edx
4005cb:   39 f2               cmp %esi,%edx
4005cd:   7c f3               jl 4005c2 <sum_array+0xc>
4005cf:   f3 c3               repz retq

Each instruction has an operation name ("opcode").
Each instruction can also have arguments (“operands”).
Our First Assembly

00000000004005b6 <sum_array>:
4005b6:    ba 00 00 00 00 00     mov $0x0,%edx
4005bb:    b8 00 00 00 00 00     mov $0x0,%eax
4005c0:    eb 09                 jmp 4005cb <sum_array+0x15>
4005c2:    48 63 ca              movslq %edx,%rcx
4005c5:    03 04 8f              add (%rdi,%rcx,4),%eax
4005c8:    83 c2 01              add $0x1,%edx
4005cb:    39 f2                 cmp %esi,%edx
4005cd:    7c f3                 jl 4005c2 <sum_array+0xbc>
4005cf:    f3 c3                 repz retq

$[number] means a constant value (e.g. 1 here).
Our First Assembly

```plaintext
00000000004005b6 <sum_array>:
4005b6:  ba 00 00 00 00  mov  $0x0,%edx
4005bb:  b8 00 00 00 00  mov  $0x0,%eax
4005c0:  eb 09  jmp  4005cb <sum_array+0x15>
4005c2:  48 63 ca  movslq %edx,%rcx
4005c5:  03 04 8f  add  (%rdi,%rcx,4),%eax
4005c8:  83 c2 01  add  $0x1,%edx
4005cb:  39 f2  cmp  %esi,%edx
4005cd:  7c f3  jl   4005c2 <sum_array+0xc>
4005cf:  f3 c3  repz retq
```

[%[name]] means a register (e.g. edx here).
Plan For Today

- **Overview**: GCC and Assembly
- **Demo**: Looking at an executable
- Registers and The Assembly Level of Abstraction
- A Brief History
- Our First Assembly

- **Break**: Announcements
- The mov instruction
Announcements

• TreeHacks hackathon this weekend – register online if you’d like to attend!
Plan For Today

- **Overview**: GCC and Assembly
- **Demo**: Looking at an executable
- Registers and The Assembly Level of Abstraction
- A Brief History
- Our First Assembly
- **Break**: Announcements
- **The mov instruction**
The **mov** instruction **copies** bytes from one place to another.

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

The **src** and **dst** can each be one of:
- Immediate (constant value, like a number)
- Register
- Memory Location (*at most one of src, dst*)
# 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  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.
<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>Address in %rax</td>
</tr>
<tr>
<td>4(%rax)</td>
<td>Address in %rax, plus 4</td>
</tr>
<tr>
<td>(%rax, %rdx)</td>
<td>Sum of values 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>Address in %rcx, times 4 (multiplier can be 1, 2, 4, 8)</td>
</tr>
<tr>
<td>2(%rax)</td>
<td>Value in %rax, plus 2 times address in %rcx</td>
</tr>
<tr>
<td>8(%rax, %rcx, 2)</td>
<td>Value in %rax, plus 2 times address in %rcx, plus 8</td>
</tr>
<tr>
<td>Syntax</td>
<td>Meaning</td>
</tr>
<tr>
<td>----------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td>0x104</td>
<td>Address 0x104 (no $)</td>
</tr>
<tr>
<td>Syntax</td>
<td>Meaning</td>
</tr>
<tr>
<td>---------</td>
<td>--------------------------------</td>
</tr>
<tr>
<td>0x104</td>
<td>Address 0x104 (no $)</td>
</tr>
<tr>
<td>(%rax)</td>
<td>Address in %rax</td>
</tr>
</tbody>
</table>
## 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>Address in %rax</td>
</tr>
<tr>
<td>4(%rax)</td>
<td>Address in %rax, plus 4</td>
</tr>
</tbody>
</table>
# 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>Address in %rax</td>
</tr>
<tr>
<td>4(%rax)</td>
<td>Address in %rax, plus 4</td>
</tr>
<tr>
<td>(%rax, %rdx)</td>
<td>Sum of values in %rax and %rdx</td>
</tr>
</tbody>
</table>
## 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>Address in %rax</td>
</tr>
<tr>
<td>4(%rax)</td>
<td>Address in %rax, plus 4</td>
</tr>
<tr>
<td>(%rax, %rdx)</td>
<td>Sum of values in %rax and %rdx</td>
</tr>
<tr>
<td>4(%rax, %rdx)</td>
<td>Sum of values in %rax and %rdx, plus 4</td>
</tr>
</tbody>
</table>
## 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>Address in %rax</td>
</tr>
<tr>
<td>4(%rax)</td>
<td>Address in %rax, plus 4</td>
</tr>
<tr>
<td>(%rax, %rdx)</td>
<td>Sum of values 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>Address in %rcx, times 4 (multiplier can be 1, 2, 4, 8)</td>
</tr>
</tbody>
</table>
# 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>Address in %rax</td>
</tr>
<tr>
<td>4(%rax)</td>
<td>Address in %rax, plus 4</td>
</tr>
<tr>
<td>(%rax, %rdx)</td>
<td>Sum of values 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>Address in %rcx, times 4 (multiplier can be 1, 2, 4, 8)</td>
</tr>
<tr>
<td>(%rax, %rcx, 2)</td>
<td>Value in %rax, plus 2 times address in %rcx</td>
</tr>
</tbody>
</table>
# 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>Address in %rax</td>
</tr>
<tr>
<td>4(%rax)</td>
<td>Address in %rax, plus 4</td>
</tr>
<tr>
<td>(%rax, %rdx)</td>
<td>Sum of values 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>Address in %rcx, times 4 (multiplier can be 1, 2, 4, 8)</td>
</tr>
<tr>
<td>(%rax, %rcx, 2)</td>
<td>Value in %rax, plus 2 times address in %rcx</td>
</tr>
<tr>
<td>8(%rax, %rcx, 2)</td>
<td>Value in %rax, plus 2 times address in %rcx, plus 8</td>
</tr>
</tbody>
</table>
Practice With Operand Forms

Assume the following values are stored at the indicated memory addresses and registers:

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>0xFF</td>
<td>%rax</td>
<td>0x100</td>
</tr>
<tr>
<td>0x104</td>
<td>0xAB</td>
<td>%rcx</td>
<td>0x1</td>
</tr>
<tr>
<td>0x108</td>
<td>0x13</td>
<td>%rdx</td>
<td>0x3</td>
</tr>
<tr>
<td>0x10C</td>
<td>0x11</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fill in the table to the right showing the values for the indicated operands.

Reminder:

**Most general form:** \( \text{Imm}(r_b, r_i, s) \)

\( \text{Imm} + R[r_b] + R[r_i] \times s \)

Also: 260d = 0x104
Assume the following values are stored at the indicated memory addresses and registers:

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>0xFF</td>
<td>%rax</td>
<td>0x100</td>
</tr>
<tr>
<td>0x104</td>
<td>0xAB</td>
<td>%rcx</td>
<td>0x1</td>
</tr>
<tr>
<td>0x108</td>
<td>0x13</td>
<td>%rdx</td>
<td>0x3</td>
</tr>
<tr>
<td>0x10C</td>
<td>0x11</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fill in the table to the right showing the values for the indicated operands.

**Operand** | **Value** | **Comment**
---|---|---
%rax | 0x100 | Register
0x104 | 0xAB | Absolute address
$0x108 | 0x108 | Immediate
(%rax) | 0xFF | Address 0x100
4(%rax) | 0xAB | Address 0x104
9(%rax,%rdx) | 0x11 | Address 0x10C
260(%rcx,%rdx) | 0x13 | Address 0x108
0xFC(,%rcx,4) | 0xFF | Address 0x100
(%rax,%rdx,4) | 0x11 | Address 0x10C

**Most general form:** \( \text{Imm}(r_b, r_i, s) \)

\[ \text{Imm} + R[r_b] + R[r_i] * s \]

Also: \(260d = 0x104\)
Recap

- **Overview:** GCC and Assembly
- **Demo:** Looking at an executable
- Registers and The Assembly Level of Abstraction
- A Brief History
- Our First Assembly
- **Break:** Announcements
- The `mov` instruction

**Next time:** diving deeper into assembly