CS107 Lecture 11
Introduction to Assembly

reading:
B&O 3.1-3.4
Lisa was unable to deposit $1 into her bank account with $16,777,216. Why?

32-bit float for $V = 16,777,216 = 1.0 \times 2^{24}$

<table>
<thead>
<tr>
<th>s</th>
<th>exponent (8 bits)</th>
<th>fraction (23 bits):</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1011 0111</td>
<td>0000 0000 0000 0000 0000 0000 000</td>
</tr>
</tbody>
</table>

$V = (-1)^s \times M \times 2^E$

$E = \text{exponent} - 127$

$M = 1. [\text{fraction bits}]$

A. Overflow in float when trying to store 16,777,217
B. float does not have enough precision to store 16,777,217
C. float does not have enough range to store 16,777,217
D. Bug in the code
E. Use double (64-bit floating point)
F. Other
Lisa was unable to deposit $1 into her bank account with $16,777,216. Why?

32-bit float for $V = 16,777,216 = 1.0 \times 2^{24}$

<table>
<thead>
<tr>
<th>s</th>
<th>exponent (8 bits)</th>
<th>fraction (23 bits):</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1011 0111</td>
<td>0000 0000 0000 0000 0000 000</td>
</tr>
</tbody>
</table>

$V = (-1)^s \times M \times 2^E$

$E = \text{exponent} - 127$

$M = 1. [\text{fraction bits}]$

A. Overflow in float when trying to store 16,777,217

B. float does not have enough precision to store 16,777,217

C. float does not have enough range to store 16,777,217

D. Bug in the code

E. Use double (64-bit floating point)

F. Other

- **Overflow for floats means INF; underflow means 0**
- Precision ≈ # bits in fraction
- Range ≈ # bits in exponent
- Maybe 😊
- Different account balances could still have limitations in precision
Thank you for your honest opinions about CS107!

We’re happy that CS107 is working well for some of you, but we also realize that CS107 is a *huge* step up in **difficulty, time, and independent learning** compared to the CS106 series.

**Learning goals for this course:**

- Improve programming skills
- Learn C and computer system design/layout
- Develop ability to glean important information from dense resources (C manual, website, lecture, lab/assignment specs, **textbook**)

(We also realize that Hewlett 200 is a *gigantic* lecture hall!)
1. **Bits and Bytes** - How can a computer represent integer numbers?

2. **Chars and C-Strings** - How can a computer represent and manipulate more complex data like text?

3. **Pointers, Stack and Heap** – How can we effectively manage all types of memory in our programs?

4. **Generics** - How can we use our knowledge of memory and data representation to write code that works with any data type?

5. **Floats** - How can a computer represent floating point numbers in addition to integer numbers?

6. **Assembly** - How does a computer interpret and execute C programs?

7. **Heap Allocators** - How do core memory-allocation operations like malloc and free work?
CS107 Topic 6: How does a computer interpret and execute C programs?
Learning Assembly

Moving data around

Today

2/17

Arithmetic and logical operations

Midterm

Control flow

2/21

Function calls

2/24

(get excited: learn how the stack is actually managed)
Today’s 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
Plan For Today

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

```bash
cp -r /afs/ir/class/cs107/samples/lectures/lect11 .
```
Plan For Today

- **Overview**: GCC and Assembly
- **Demo**: Looking at an executable
- Registers and The Assembly Level of Abstraction
- **Break**: Announcements
- The `mov` instruction
It’s bits all the way down...

Data representation so far
• Integer (unsigned int, 2’s complement signed int)
• char (ASCII)
• Address (unsigned long)
• float/double (IEEE floating point)
• Aggregates (arrays, structs)

The code itself is binary too!
• Instructions (machine encoding)
gcc (compiler) compiles human-readable code into machine-readable instructions (bits and bytes).

Assembly/machine code is processor-dependent (C code isn’t).

Assembly code is a shorthand/legible version of machine code.

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

```assembly
mov     $0x0,%edx
mov     $0x0,%eax
jmp     4005cb <sum_array+0x15>
movslq  %edx,%rcx
```

Machine code:

```
ba 00 00 00 00 00
b8 00 00 00 00 00
eb 09
48 63 ca
```
Plan For Today

- **Overview:** GCC and Assembly
- **Demo:** Looking at an executable
- Registers and The Assembly Level of Abstraction
- **Break:** Announcements
- The `mov` instruction
Demo: Looking at an Executable (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?
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;
}
```

```
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>
```

```
make objdump -d sum
```
What’s in an object file?

**machine code**
- each instruction encoded in binary

**assembly code**
- each machine instruction decoded into human-readable assembly

**function pointer**: name of function, mem. address of code

- sequential instructions are at sequential addresses
What is an assembly instruction?

00000000004005b6 <sum_array>:  
4005b6:    ba 00 00 00 00 00  
4005bb:    b8 00 00 00 00 00  
4005c0:    eb 09  
4005c2:    48 63 ca  
4005c5:    03 04 8f  
4005c8:    83 c2 01  
4005cb:    39 f2  
4005cd:    7c f3  
4005cf:    f3 c3  

00000000004005bb:  
4005bb:    b8 00 00 00 00 00  
4005c0:    eb 09  
4005c2:    48 63 ca  
4005c5:    03 04 8f  
4005c8:    83 c2 01  
4005cb:    39 f2  
4005cd:    7c f3  
4005cf:    f3 c3  

What is an assembly instruction?

- **OpCode**: The instruction name (e.g., `mov`, `add`, `cmp`, `jmp`, `repz`, `retq`).
- **Operands**: Arguments to the instruction (e.g., constants, registers, memory addresses).

**Example**:  
```
4005b6:    ba 00 00 00 00 00  
          mov 钡 00 00 00 00 00  

4005bb:    b8 00 00 00 00 00  
          mov 0x0, %edx  

4005c0:    eb 09  
          jmp 0x0, %edx  

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  
```

- **%eax** is a register name (storage location on CPU).
- **$0x1** is a constant value ("immediate").
- **4005c2** is a direct address in memory.

---

18
Plan For Today

- **Overview**: GCC and Assembly
- **Demo**: Looking at an executable
- **Registers and The Assembly Level of Abstraction**
- **Break**: Announcements
- **The mov instruction**
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.
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
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!
Storage abstraction: C vs assembly

High-level programming language (C)

Variable
- Variable type (int, char, void*, etc.) determines # of bytes stored + valid ops
- Local to stack frame (current function call)

Assembly language (x86-64)

Register
- 64-bit space inside processor, simply holds bits
- Registers are shared across all function calls
Storage abstraction: C vs assembly

High-level programming language (C)

Variable
- Variable type (int, char, void*, etc.) determines # of bytes stored + valid ops
- Local to stack frame (current function call)

Assembly language (x86-64)

Register
- 64-bit space inside processor, simply holds bits
- Registers are shared across all function calls

Shared abstraction (C and x86-64)

Memory
- Byte-addressable: Each memory address refers to the start of one byte
- Stack managed automatically in C, manually in assembly

Shared abstraction (C and x86-64)

• Read/write to memory
• Assignment to variable/register
• Arithmetic on variables/registers
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).
Two major categories of ISAs

CISC (e.g., x86)
• C for “Complex”
• Large set of expressive, specialized instructions
• Used in most computers
• Developed by AMD, cross-licensed by Intel

RISC (e.g., ARM, MIPS)
• R for “Reduced”
• Small set of simple instructions, but more instructions in code
• Used in low-power, low-cost embedded systems
• Former Stanford President John Hennessy designed the MIPS processor

JOHN L HENNESSY
United States – 2017
CITATION
For pioneering a systematic, quantitative approach to the design and evaluation of computer architectures with enduring impact on the microprocessor industry.
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
Plan For Today

- **Overview**: GCC and Assembly
- **Demo**: Looking at an executable
- Registers and The Assembly Level of Abstraction
- **Break**: Announcements
- The `mov` instruction
Midterm Exam

The midterm exam is **Fri. 2/14 12:30PM-2:20PM in Hewlett 200.**

- Covers material through **lab4/assign4** (no floats or assembly language)
- Closed-book, 1 2-sided page of notes permitted, C reference sheet provided

Administered via BlueBook software (on your laptop)

- Practice materials and BlueBook download available on course website

Assignment 4 on time deadline is tonight, assignment 5 goes out today and is due **Fri. 2/21.** We recommend starting to work on it *after* the midterm exam.
Joke break
Plan For Today

- **Overview:** GCC and Assembly
- **Demo:** Looking at an executable
- **Registers and The Assembly Level of Abstraction**
- **Break:** Announcements
- **The mov instruction**
The **mov** instruction **copies** bytes from one place to another; it is similar to the assignment operator (=) in C.

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

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

- Immediate (constant value, like a number) (**only src**) $0x104$
- Register \%rbx
- Memory Location (at most one of **src**, **dst**)
  
  Direct address 0x6005c0
Operand forms: load/store

mov $0x0, 0x6040

Store the value 0 into memory at address 0x6040.

mov %rbx, %rax

Load the value in register %rbx, store into register %rax

mov 0x6040, %rbx

Load value from address 0x6040 into register %rbx

0xC0FF

%rax

0xFEFE

%rbx

0x6050

0x6048

0x6040

0x1234
Practice #1: Imm/reg/direct

What are the results of the following move instructions (executed separately)?

1. `mov $0x100,%rax`
   - `%rax` will be set to $0x100

2. `mov 0x100,%rax`
   - `%rax` will be set to $0x100

3. `mov %rbx,0x120`
   - `%rbx` will be set to $0xCD
   - The value at $0x120` will be $0x120

Diagram:

```
0x00 0x100 0x110 0x120 0x128
%rbx
0xCD
0x00 0x100 0x110 0x120 0x128
%rax
0x100
0xBEEF
```

8 bytes
The **mov** instruction **copies** bytes from one place to another; it is similar to the assignment operator (\(=\)) in C.

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

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

- Immediate (constant value, like a number) (**only src**)
- Register
- Memory Location **(at most one of src, dst)**

**Example:**
- Direct address: \(0x104\)
- Indirect address: \(0x6005c0\) **(%rbx)**
Operand forms: Indirect (1/2)

\[
\text{mov } (\%rax),\%rax \quad \text{Load value at address } \%rax \text{ and store into register } \%rax.
\]

\[
\text{mov } 0x0,16(\%rbx) \quad \text{Store the constant 0 at address } (16 \text{ plus } \%rbx)
\]

\[
\text{mov } 0x0,(\%rbx,\%rcx) \quad \text{Store the constant 0 at address } (\%rbx + \%rcx)
\]

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

- **Displacement**: positive or negative constant
- **Base**: register
- **Index**
Practice #2: Operand Forms

What are the results of the following move instructions (executed separately)?

1. **mov** $0x42,(%rax)

2. **mov** -8(%rax),%rbx

3. **mov** 9(%rax,%rcx),%rbx

---

**Imm(r_b, r_i)** is equivalent to address

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

Displacement  Base  Index

---

<table>
<thead>
<tr>
<th>Imm(r_b, r_i) is equivalent to address</th>
<th>Imm + R[r_b] + R[r_i]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Displacement</td>
<td>Base</td>
</tr>
<tr>
<td>Index</td>
<td></td>
</tr>
</tbody>
</table>
Operand forms: Indirect (2/2)

mov (,%rax,4),%rbx

mov $0x0,0x10(%rbx,%rax,2)

Load value at address \((4 \text{ times } \%rax)\) and store in register \%rbx.

Store the constant 0 at address \((0x10 \text{ plus } \%rbx + (2 \text{ times } \%rax))\)

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

- **Base**: register (if missing, = 0)
- **Displacement**: pos/neg constant (if missing, = 0)
- **Index**
- **Scale** must be 1,2,4, or 8 (if missing, = 1)
Practice #3: Operand Forms

What are the results of the following move instructions (executed separately)?

1. `mov $0x42, 0xfc(,%rbx,4)`
   - `0x108`
   - `%rax
   - `%rbx
   - `%rcx

2. `mov (%rax,%rcx,4),%rdx`
   - `0x1"
   - `%rax
   - `%rcx
   - `%rdx

Imm($rb, ri, s)$ is equivalent to address Imm + R[rb] + R[ri]*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>
Most General Operand Form

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

is equivalent to...

\[ \text{Imm} + R[r_b] + R[r_i] \times s \]
### 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>$M[R[r_i] \cdot s]$</td>
<td>Scaled indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>$Imm(r_i, s)$</td>
<td>$M[Imm + R[r_i] \cdot s]$</td>
<td>Scaled indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>($r_b, r_i, s$)</td>
<td>$M[R[r_b] + R[r_i] \cdot s]$</td>
<td>Scaled indexed</td>
</tr>
<tr>
<td>Memory</td>
<td>$Imm(r_b, r_i, s)$</td>
<td>$M[Imm + R[r_b] + R[r_i] \cdot s]$</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.”
Why are there so many forms of indirect addressing?

We see these indirect addressing paradigms in C as well!
Goals of indirect addressing: C

```c
long exchange(long *xp, long y) {
    long x = *xp;
    *xp = y;
    return y;
}

void last_element(long *arr, int nelems) {
    long z = arr[nelems - 1];
}
```

Try your intuition: How do you think each of the C assignments might map to mov instructions? (many right answers!)

<table>
<thead>
<tr>
<th>Imm(r_b, r_i, s)</th>
<th>is equivalent to address Imm + R[r_b] + R[r_i] * s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Displacement</td>
<td>Base</td>
</tr>
<tr>
<td></td>
<td>Index</td>
</tr>
<tr>
<td></td>
<td>Scale (1, 2, 4, 8)</td>
</tr>
</tbody>
</table>
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
• \texttt{mov \ src, \ dst} \textit{copies} value into \texttt{dst}
• \texttt{sizeof(int)} is 4
• Instructions executed sequentially

We’ll come back to this example in a week!