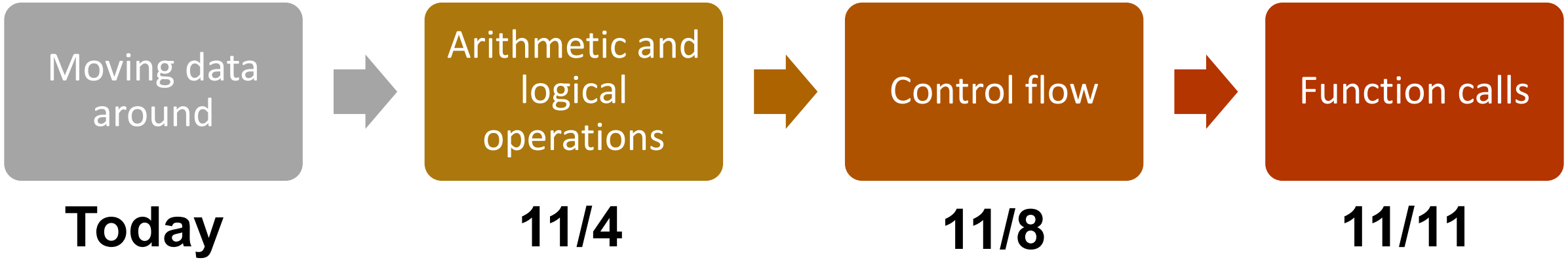
CS107, Lecture 11
Introduction to Assembly

Reading: B&O 3.1-3.4

CS107 Topic 6: How does a computer interpret and execute C programs?

Learning Assembly



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
- 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

- **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.

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

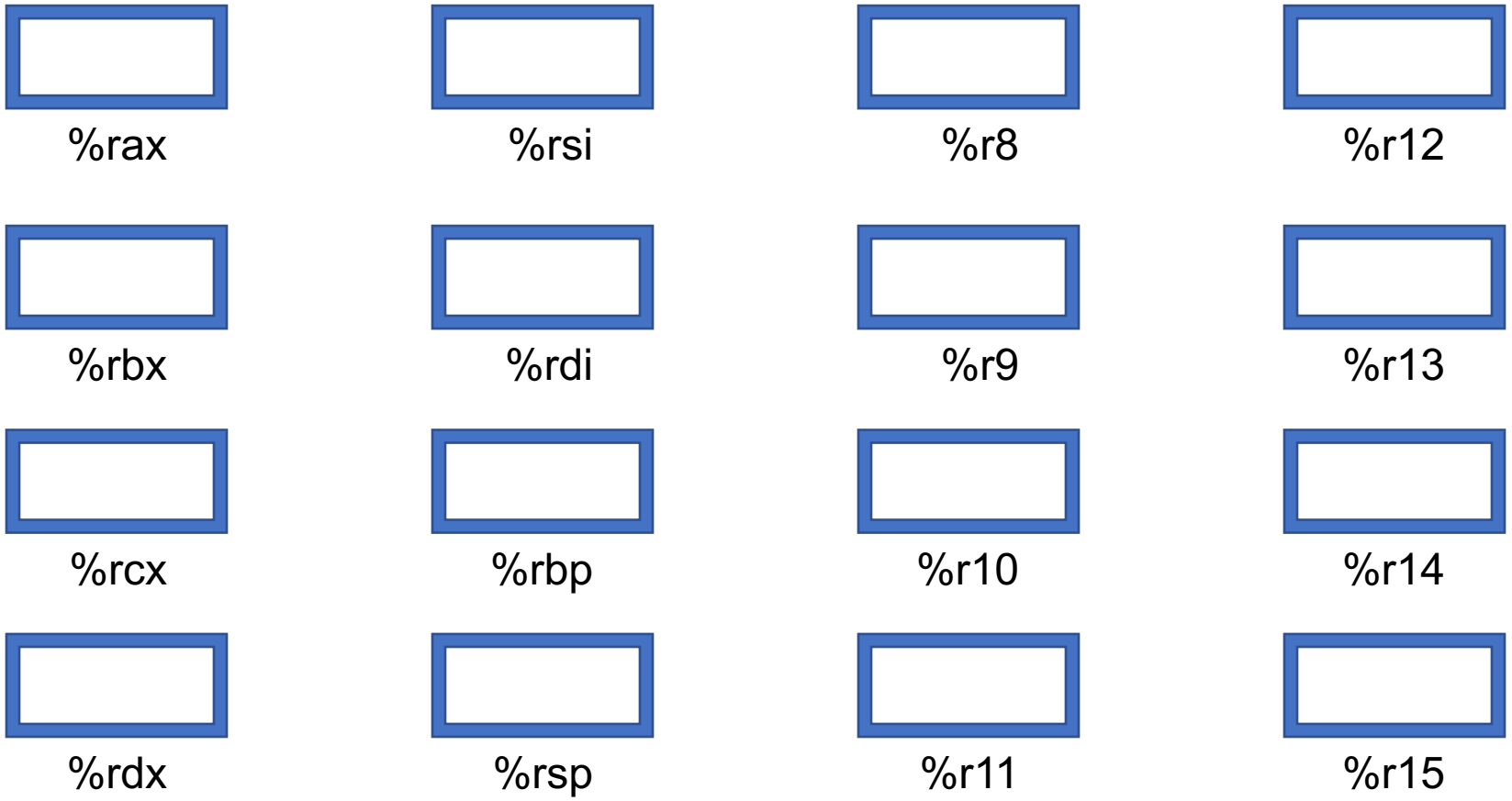
Assembly Abstraction

- C abstracts away the low level details of machine code. It lets us work using 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



Registers

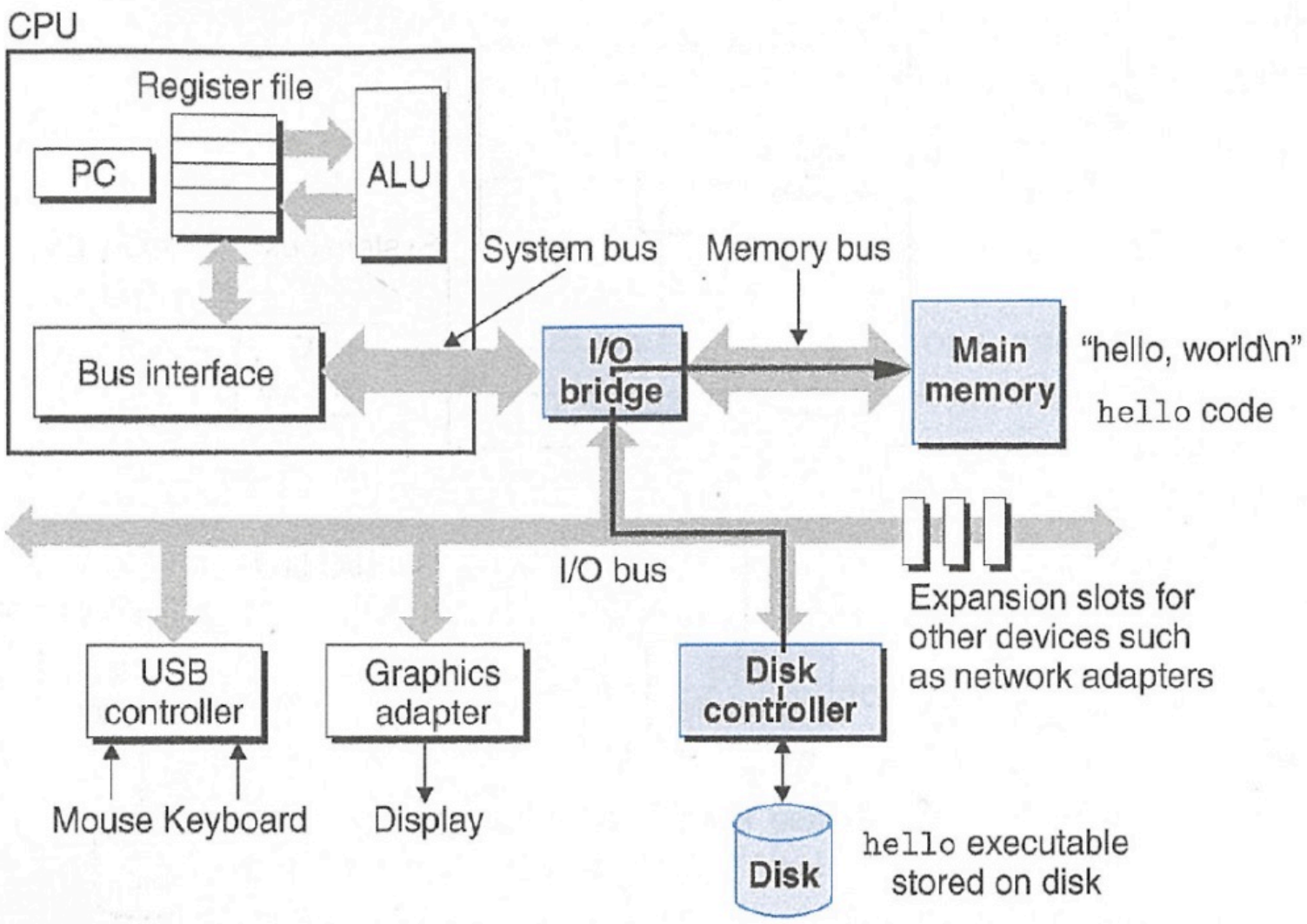
- A **register** is a 64-bit space inside the processor.
- There are 16 registers available, each with a unique name.
- Registers are like “scratch paper” for the processor. Data being calculated or manipulated is moved to registers first. Operations are performed on registers.
- Registers also hold 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:

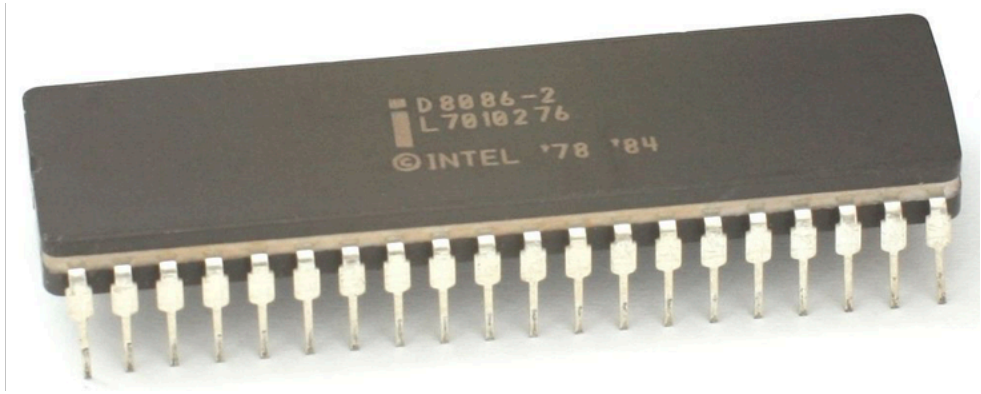
C	Assembly Abstraction
`int sum = x + y;`	1) Copy x into register 1 2) Copy y into register 2 3) Add register 2 to register 1 4) Write register 1 to memory for sum

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

- 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

Our First Assembly

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

What does this look like in assembly?

Our First Assembly

```
0000000004005b6 <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
```

Our First Assembly

```
0000000004005b6 <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 d2              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
```

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

Our First Assembly

```
0000000004005b6 <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:   4
  4005c5:   0                   ,4),%eax
  4005c8:   8
  4005cb:   3
  4005cd:   7c f3               jl     4005c2 <sum_array+0xc>
  4005cf:   f3 c3               repz retq
```

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

Our First Assembly

```
0000000000004005b6 <sum_array>:
  4005b6:    ba 00 00 00 00
  4005bb:    b8 00 00 00 00
  4005c0:    eb 09
  4005cd:    7c f3
  4005cf:    f3 c3
```

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

```
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

```
0000000004005b6 <sum_array>:
  4005b6:     ba 00 00 00 00     mov    $0x0,%edx
  4005bb:     b8 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              repz retq
```

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

```
0000000004005b6 <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
```

Our First Assembly

```
0000000004005b6 <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
```

Each instruction has an operation name (“opcode”).

Our First Assembly

```
0000000004005b6 <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
```

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

Our First Assembly

```
0000000004005b6 <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
```

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

Our First Assembly

```
0000000004005b6 <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).

Announcements

- The midterm exam is **Fri. 11/1 11:30AM-1:20PM in Nvidia Aud. and Cubberley Aud.**
  - **Last names A-N: Nvidia Auditorium**
  - **Last Names O-Z: Cubberley Auditorium**
- We will confirm via email accommodations for all students who have requested midterm accommodations by the end of the day today. If you need accommodations but did not contact us, please email the course staff immediately.
- Assignment 4 on time deadline is tonight, assignment 5 goes out then and is due **Fri. 11/8**. We recommend starting to work on it *after* the midterm exam.

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

mov

The **mov** instruction copies bytes from one place to another.

```
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***)

Operand Forms: Immediate

```
mov     $0x104,_____
```

Copy the value 0x104 into some destination.

Operand Forms: Registers

Copy the value in register %rbx into some destination.

```
mov     %rbx,_____
```

```
mov     _____,%rbx
```

Copy the value from some source into register %rbx.

Operand Forms: Absolute Addresses

Copy the value at address 0x104 into some destination.

```
mov     0x104,_______
```

```
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.

```
mov     (%rbx),_______
```

```
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.*

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

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

*Copy the value from some source into the memory at the address (**0x10 plus** what is stored in register %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),_____
```

```
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 **0x10 plus** the values in registers %rax and %rdx) into some destination.*

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

```
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).*

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
```

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.

```
mov    _______,(,%rdx,4)
```

*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),_______
```

```
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 (**<u>the value in register %rax</u>** plus 2 times the value in register %rdx) into some destination.*

```
mov     (%rax,%rdx,2),__________
```

```
mov     __________,(%rax,%rdx,2)
```

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

Operand Forms: Scaled Indexed

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

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

```
mov     _______,0x4(%rax,%rdx,2)
```

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

Most General Operand Form

$\texttt{Imm}(r_b, r_i, s)$

is equivalent to…

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

Operand Forms

Type	Form	Operand Value	Name
Immediate	$\$Imm$	Imm	Immediate
Register	r_a	$\mathrm{R}[r_a]$	Register
Memory	Imm	$\mathrm{M}[Imm]$	Absolute
Memory	(r_a)	$\mathrm{M}[\mathrm{R}[r_a]]$	Indirect
Memory	$Imm(r_b)$	$\mathrm{M}[Imm + \mathrm{R}[r_b]]$	Base + displacement
Memory	(r_b, r_i)	$\mathrm{M}[\mathrm{R}[r_b] + \mathrm{R}[r_i]]$	Indexed
Memory	$Imm(r_b, r_i)$	$\mathrm{M}[Imm + \mathrm{R}[r_b] + \mathrm{R}[r_i]]$	Indexed
Memory	$(, r_i, s)$	$\mathrm{M}[\mathrm{R}[r_i] \cdot s]$	Scaled indexed
Memory	$Imm(, r_i, s)$	$\mathrm{M}[Imm + \mathrm{R}[r_i] \cdot s]$	Scaled indexed
Memory	(r_b, r_i, s)	$\mathrm{M}[\mathrm{R}[r_b] + \mathrm{R}[r_i] \cdot s]$	Scaled indexed
Memory	$Imm(r_b, r_i, s)$	$\mathrm{M}[Imm + \mathrm{R}[r_b] + \mathrm{R}[r_i] \cdot s]$	Scaled indexed

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
```

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