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

- **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)

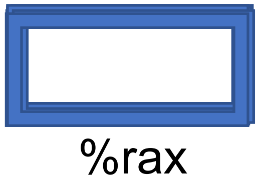
Plan For Today

- **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

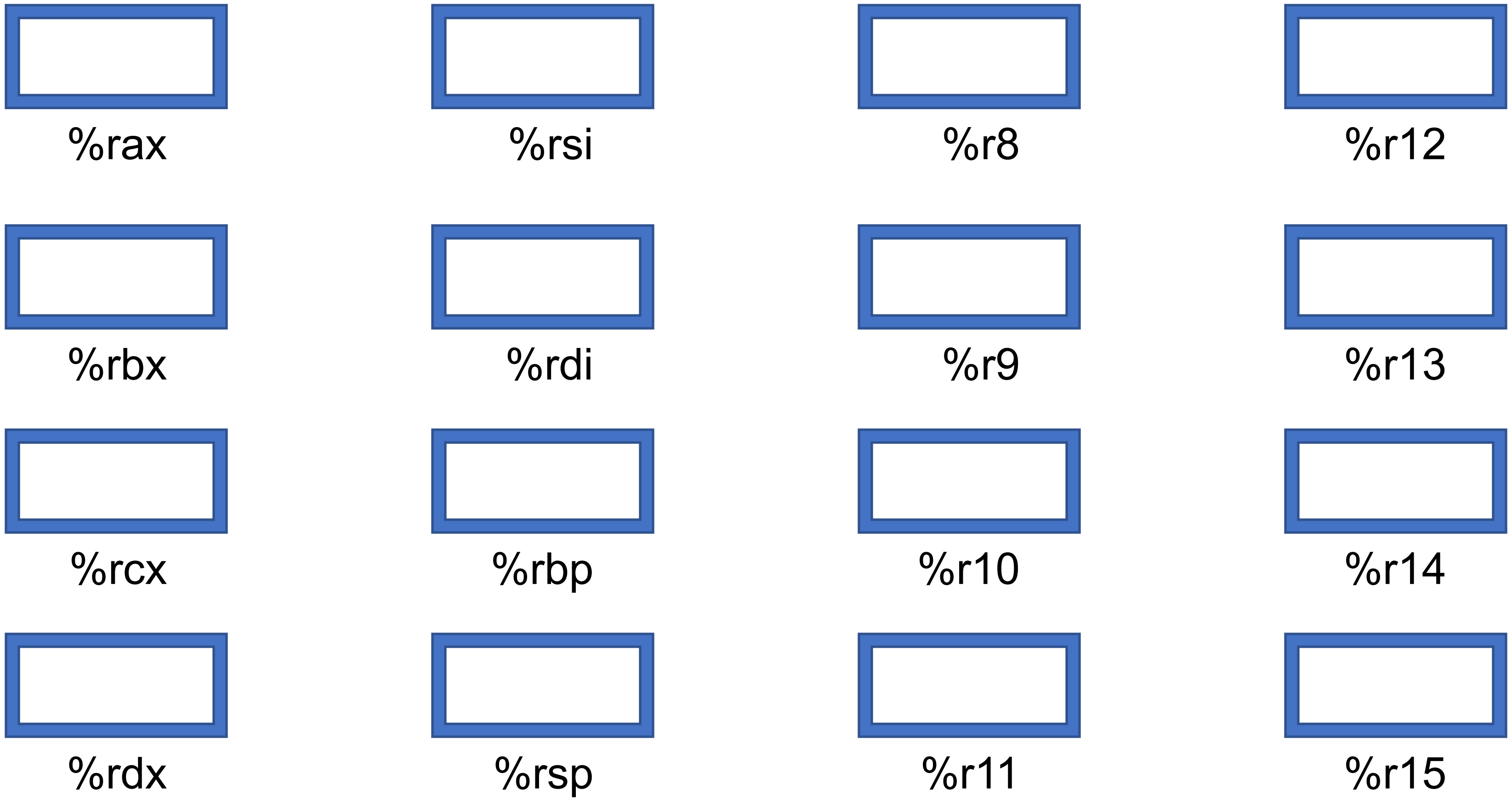
Assembly Abstraction

- 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



Registers



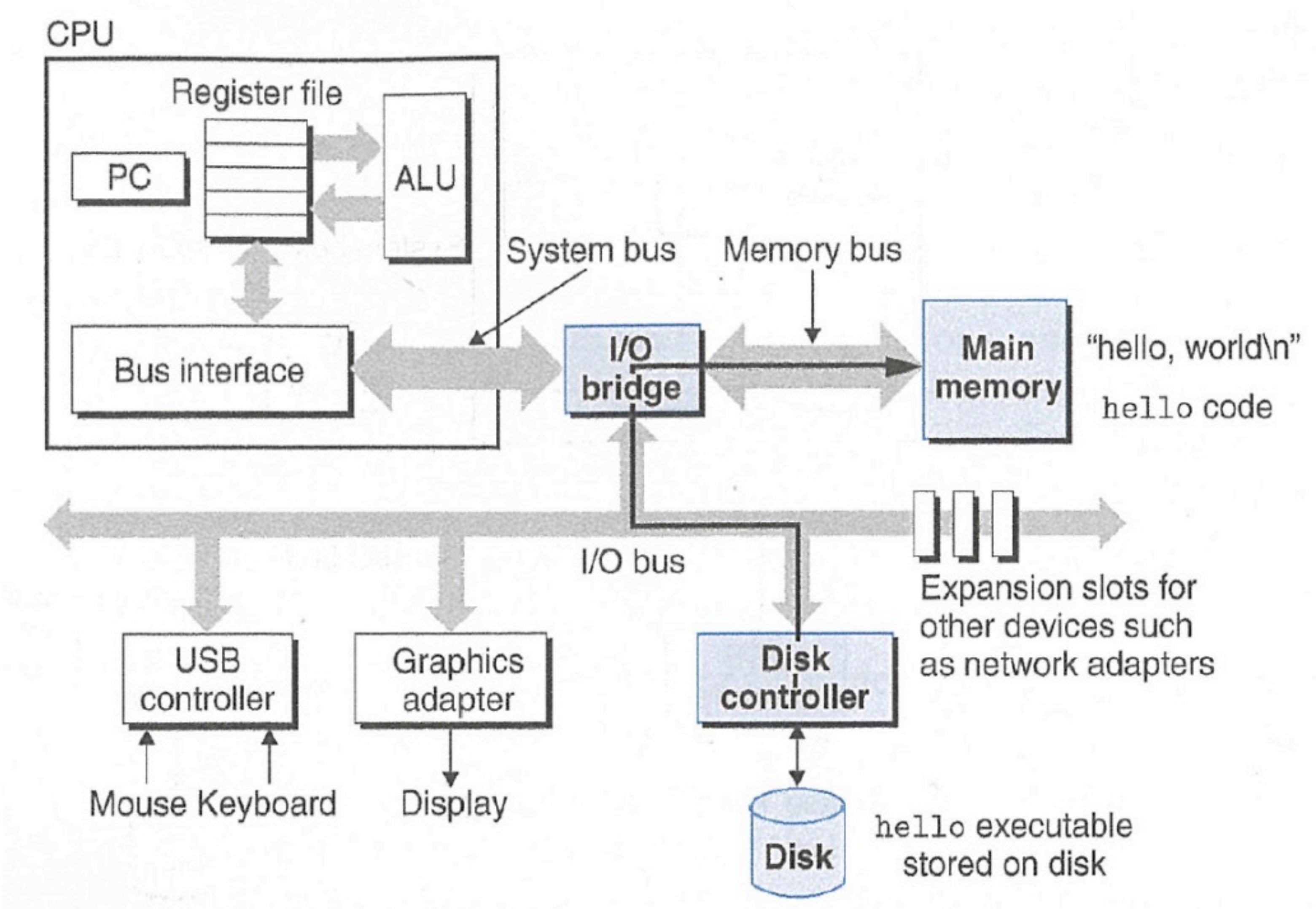
Registers

- 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:

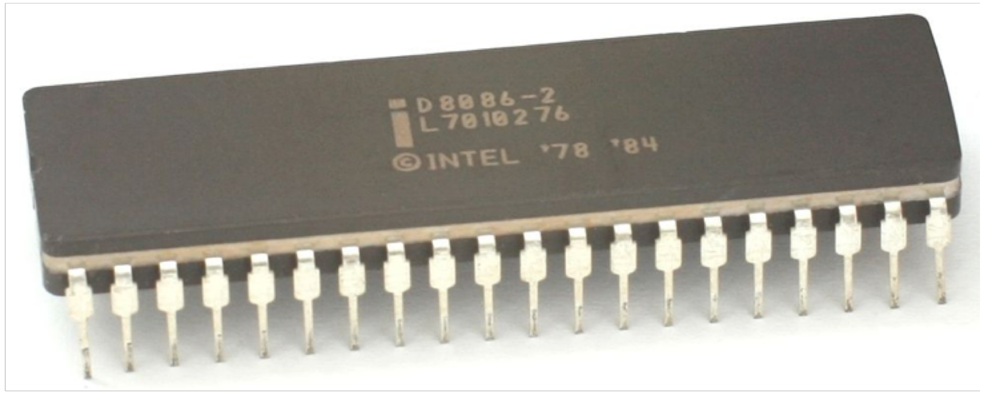
C	Assembly Abstraction
`int sum = x + y;`	1) Copy x into register 1 2) Copy y into register 2 3) Add register 2 to register 1 4) Write register 1 to memory for sum

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

```
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
```

Our First Assembly

```
00000000004005b6 <sum_array>:
  4005b6:     ba 00 00 00 00          mov    $0x0,%edx
  4005bb:     b8 00 00 00 00          mov    $0x0,%eax
                                       mp    4005cb <sum_array+0x15>
                                      ovslq  %edx,%rcx
                                       dd    (%rdi,%rcx,4),%eax
                                       dd    $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

```
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:   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

```
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>
                                   movslq %edx,%rcx
                                   add    (%rdi,%rcx,4),%eax
                                   add    $0x1,%edx
                                   cmp    %esi,%edx
  4005cd:    7c f3                 jl     4005c2 <sum_array+0xc>
  4005cf:    f3 c3                 repz retq
```

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

Our First Assembly

```
00000000004005b6 <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

```
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
```

Our First Assembly

```
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
```

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

Our First Assembly

```
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
```

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

Our First Assembly

```
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
```

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

Our First Assembly

```
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
```

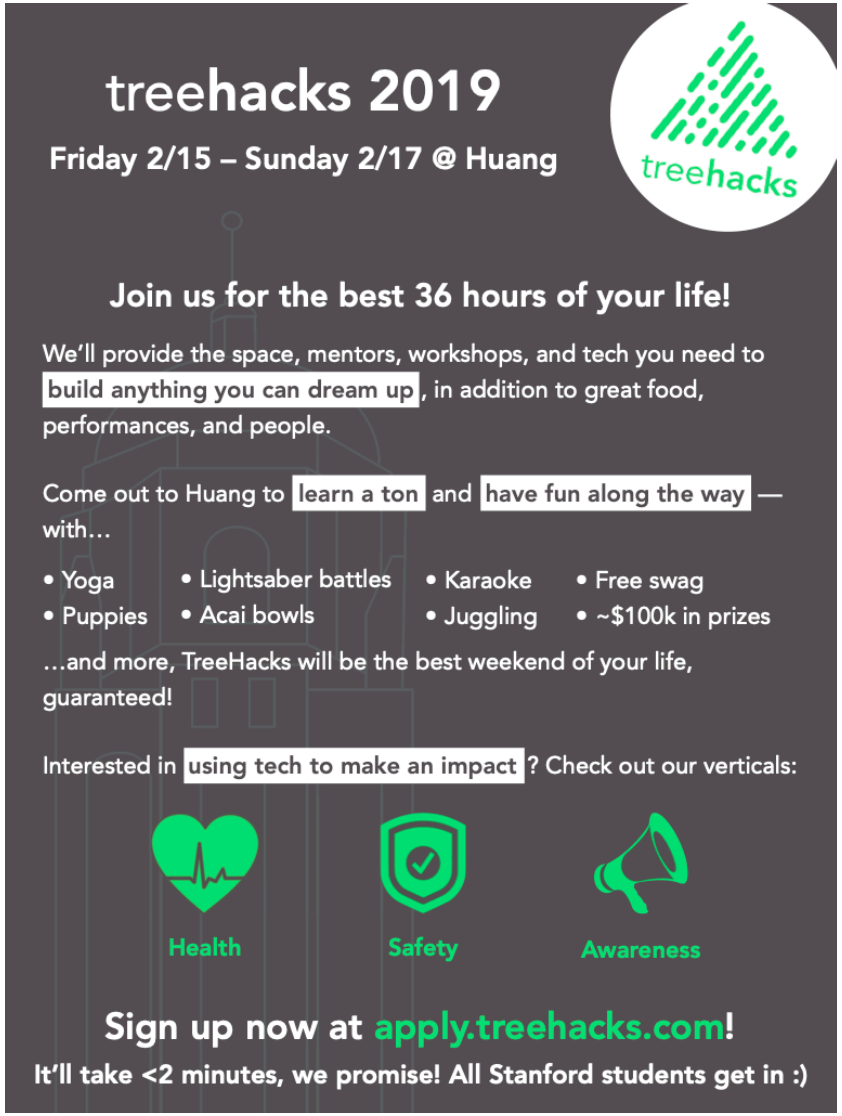
%[name] means a register (e.g. edx here).

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

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

Operand Forms

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

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.

Memory Location Syntax

Syntax	Meaning

Memory Location Syntax

Syntax	Meaning
0x104	Address 0x104 (no $)

Memory Location Syntax

Syntax	Meaning
0x104	Address 0x104 (no $)
(%rax)	Address in %rax

Memory Location Syntax

Syntax	Meaning
0x104	Address 0x104 (no $)
(%rax)	Address in %rax
4(%rax)	Address in %rax, plus 4

Memory Location Syntax

Syntax	Meaning
0x104	Address 0x104 (no $)
(%rax)	Address in %rax
4(%rax)	Address in %rax, plus 4
(%rax, %rdx)	Sum of values in %rax and %rdx

Memory Location Syntax

Syntax	Meaning
0x104	Address 0x104 (no $)
(%rax)	Address in %rax
4(%rax)	Address in %rax, plus 4
(%rax, %rdx)	Sum of values in %rax and %rdx
4(%rax, %rdx)	Sum of values in %rax and %rdx, plus 4

Memory Location Syntax

Syntax	Meaning
0x104	Address 0x104 (no $)
(%rax)	Address in %rax
4(%rax)	Address in %rax, plus 4
(%rax, %rdx)	Sum of values in %rax and %rdx
4(%rax, %rdx)	Sum of values in %rax and %rdx, plus 4
(, %rcx, 4)	Address in %rcx, times 4 (multiplier can be 1, 2, 4, 8)

Memory Location Syntax

Syntax	Meaning
0x104	Address 0x104 (no $)
(%rax)	Address in %rax
4(%rax)	Address in %rax, plus 4
(%rax, %rdx)	Sum of values in %rax and %rdx
4(%rax, %rdx)	Sum of values in %rax and %rdx, plus 4
(, %rcx, 4)	Address in %rcx, times 4 (multiplier can be 1, 2, 4, 8)
(%rax, %rcx, 2)	Value in %rax, plus 2 times address in %rcx

Memory Location Syntax

Syntax	Meaning
0x104	Address 0x104 (no $)
(%rax)	Address in %rax
4(%rax)	Address in %rax, plus 4
(%rax, %rdx)	Sum of values in %rax and %rdx
4(%rax, %rdx)	Sum of values in %rax and %rdx, plus 4
(, %rcx, 4)	Address in %rcx, times 4 (multiplier can be 1, 2, 4, 8)
(%rax, %rcx, 2)	Value in %rax, plus 2 times address in %rcx
8(%rax, %rcx, 2)	Value in %rax, plus 2 times address in %rcx, plus 8

Practice With Operand Forms

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

Address	Value	Register	Value
0x100	0xFF	%rax	0x100
0x104	0xAB	%rcx	0x1
0x108	0x13	%rdx	0x3
0x10C	0x11		

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

Operand	Value
%rax	______
0x104	______
$0x108	______
(%rax)	______
4(%rax)	______
9(%rax,%rdx)	______
260(%rcx,%rdx)	______
0xFC(,%rcx,4)	______
(%rax,%rdx,4)	______

Reminder:

Most general form: *Imm*(r_b,r_i,*s*)

$$Imm + R[r_b] + R[r_i] * s$$

Also: 260d = 0x104

Practice With Operand Forms

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

Address	Value	Register	Value
`0x100`	`0xFF`	`%rax`	`0x100`
`0x104`	`0xAB`	`%rcx`	`0x1`
`0x108`	`0x13`	`%rdx`	`0x3`
`0x10C`	`0x11`		

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

Reminder:

Most general form: $Imm(r_b,r_i,s)$

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

Also: 260d = 0x104

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`

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