CS107, Lecture 15

Introduction to Assembly

Reading: B&O 3.1-3.4

😷 masks recommended

CS107 Topic 5

<u>How does a computer interpret and execute C programs?</u>

Why is answering this question important?

- Learning how our code is really translated and executed helps us write better code
- We can learn how to reverse engineer and exploit programs at the assembly level

assign5: find and exploit vulnerabilities in an ATM program, reverse engineer a program without seeing its code, and de-anonymize users given a data leak.

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

```
cp -r /afs/ir/class/cs107/lecture-code/lect15 .
```

Lecture Plan

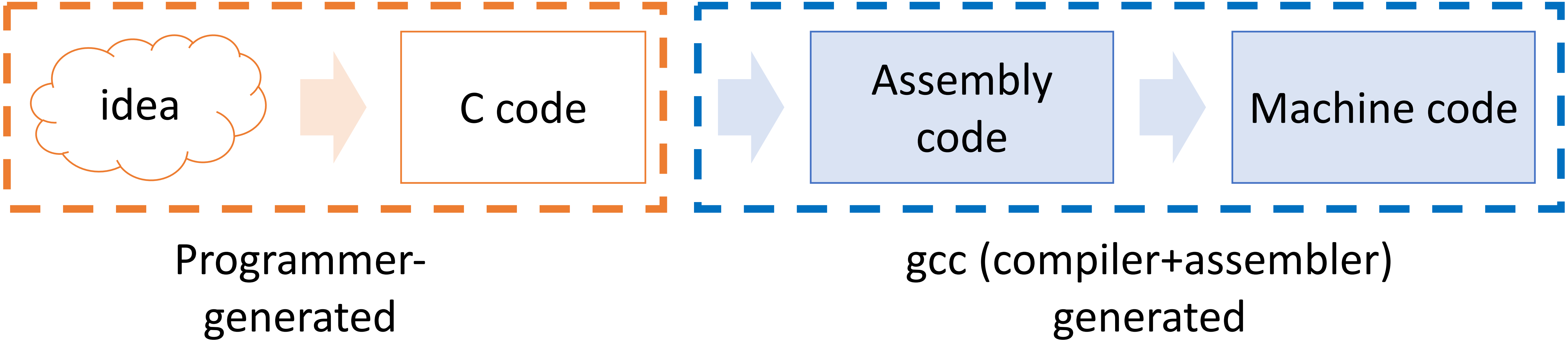
- **Overview: GCC and Assembly**
- **Demo:** Looking at an executable
- Registers and The Assembly Level of Abstraction
- The **mov** Instruction

```
cp -r /afs/ir/class/cs107/lecture-code/lect15 .
```

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.

Why are we reading assembly?



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.

Lecture Plan

- **Overview:** GCC and Assembly
- **Demo: Looking at an executable**
- Registers and The Assembly Level of Abstraction
- The **mov** Instruction

```
cp -r /afs/ir/class/cs107/lecture-code/lect15 .
```

Demo: Looking at an Executable (objdump -d)

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

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

```
make
objdump -d sum
```

```
0000000000401136 <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    401140 <sum_array+0xa>
  40114f:    89 d0                   mov    %edx,%eax
  401151:    c3                      retq
```

Our First Assembly

```
0000000000401136 <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    401140 <sum_array+0xa>
  40114f:  89 d0                 mov    %edx,%eax
  401151:  c3                    retq
```

Our First Assembly

```
0000000000401136 <sum_array>:
  401136:   b8 00 00 00 00        mov    $0x0,%eax
  40113b:   ba 00 00 00 00        mov    $0x0,%edx
                                  p      %esi,%eax
                                  e      40114f <sum_array+0x19>
                                  vslq   %eax,%rcx
                                  d      (%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
```

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

Our First Assembly

```
0000000000401136 <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                           array+0x19>
  401144: 48
  401147: 03                           4),%edx
  40114a: 83
  40114d: eb f1                 jmp    401140 <sum_array+0xa>
  40114f: 89 d0                 mov    %edx,%eax
  401151: c3                    retq
```

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

Our First Assembly

```
0000000000401136 <sum_array>:
  401136:   b8 00 00 00 00
  40113b:   ba 00 00 00 00
  401140:   39 f0
```

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

```
  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
```

Our First Assembly

```
0000000000401136 <sum_array>:
  401136:   b8 00 00 00 00          mov    $0x0,%eax
  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                   mov    %edx,%eax
  401151:   c3                      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

```
0000000000401136 <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    401140 <sum_array+0xa>
  40114f:   89 d0                 mov    %edx,%eax
  401151:   c3                    retq
```

Our First Assembly

```
0000000000401136 <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    401140 <sum_array+0xa>
  40114f:   89 d0                   mov    %edx,%eax
  401151:   c3                      retq
```

Each instruction has an operation name ("opcode").

Our First Assembly

```
0000000000401136 <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    401140 <sum_array+0xa>
  40114f:   89 d0                   mov    %edx,%eax
  401151:   c3
```

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

Our First Assembly

```
0000000000401136 <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    401140 <sum_array+0xa>
  40114f:   89 d0                   mov    %edx,%eax
  401151:   c3                      retq
```

$[number] means a constant value, or “immediate” (e.g. 1 here).

Our First Assembly

```
0000000000401136 <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    401140 <sum_array+0xa>
  40114f:   89 d0                   mov    %edx,%eax
  401151:   c3                      retq
```

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

🌟 Keep a resource guide handy 🌟

- https://web.stanford.edu/class/cs107/resources/x86-64-reference.pdf
- CS107 x86 Guide: https://cs107.stanford.edu/guide/x86-64.html
- B&O book:
  - Canvas -> Files -> Bryant_OHallaron_ch3.1-3.8.pdf
- It’s like learning how to read (not speak) a new language! (again!)

Lecture Plan

- **Overview:** GCC and Assembly
- **Demo:** Looking at an executable
- **Registers and The Assembly Level of Abstraction**
- The **mov** instruction

```
cp -r /afs/ir/class/cs107/lecture-code/lect15 .
```

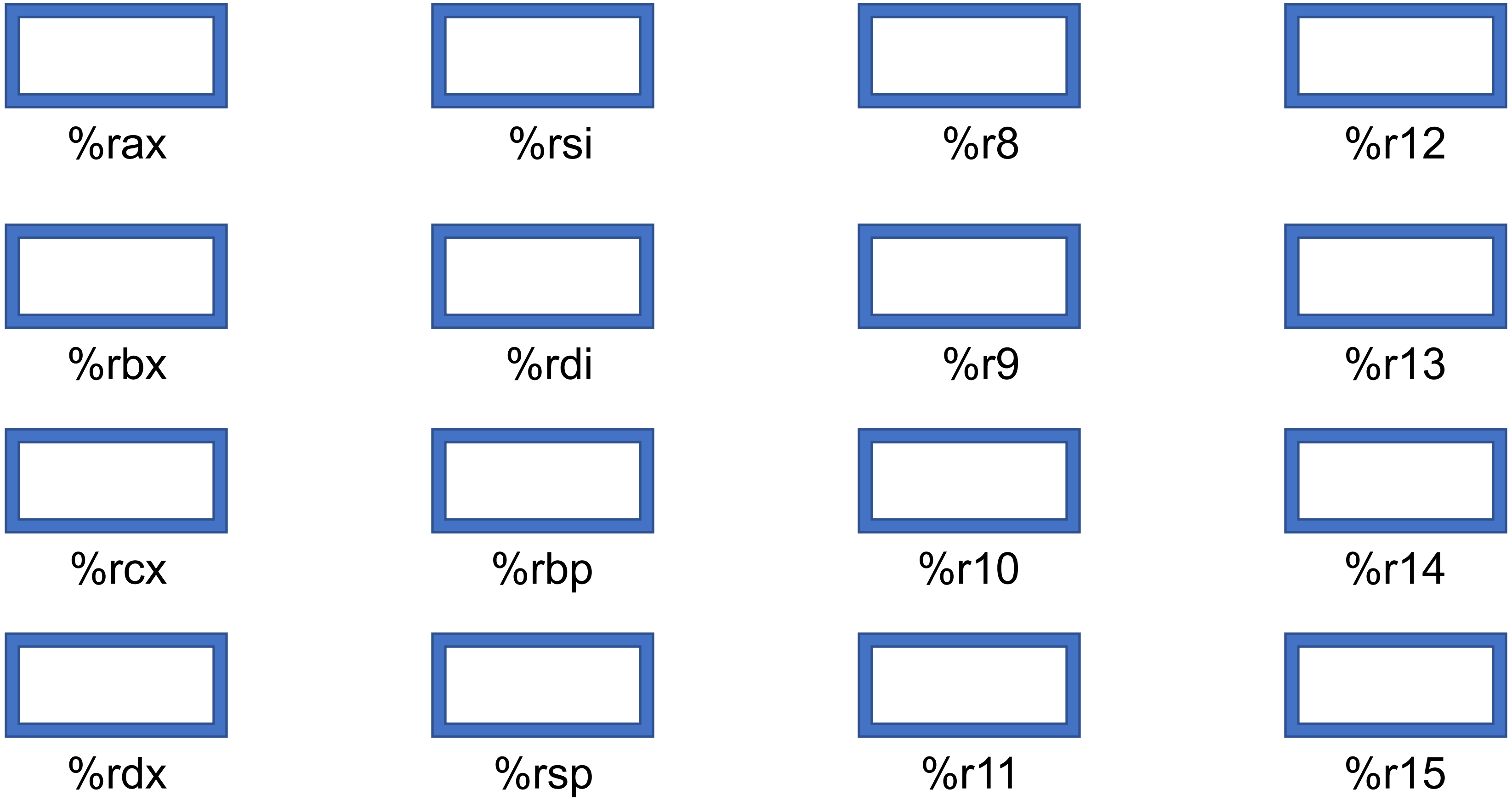
Assembly Abstraction

- 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

What is a register?

A register is a fast read/write memory slot right on the CPU that can hold variable values.

Registers are located in the CPU; they are separate from main memory.

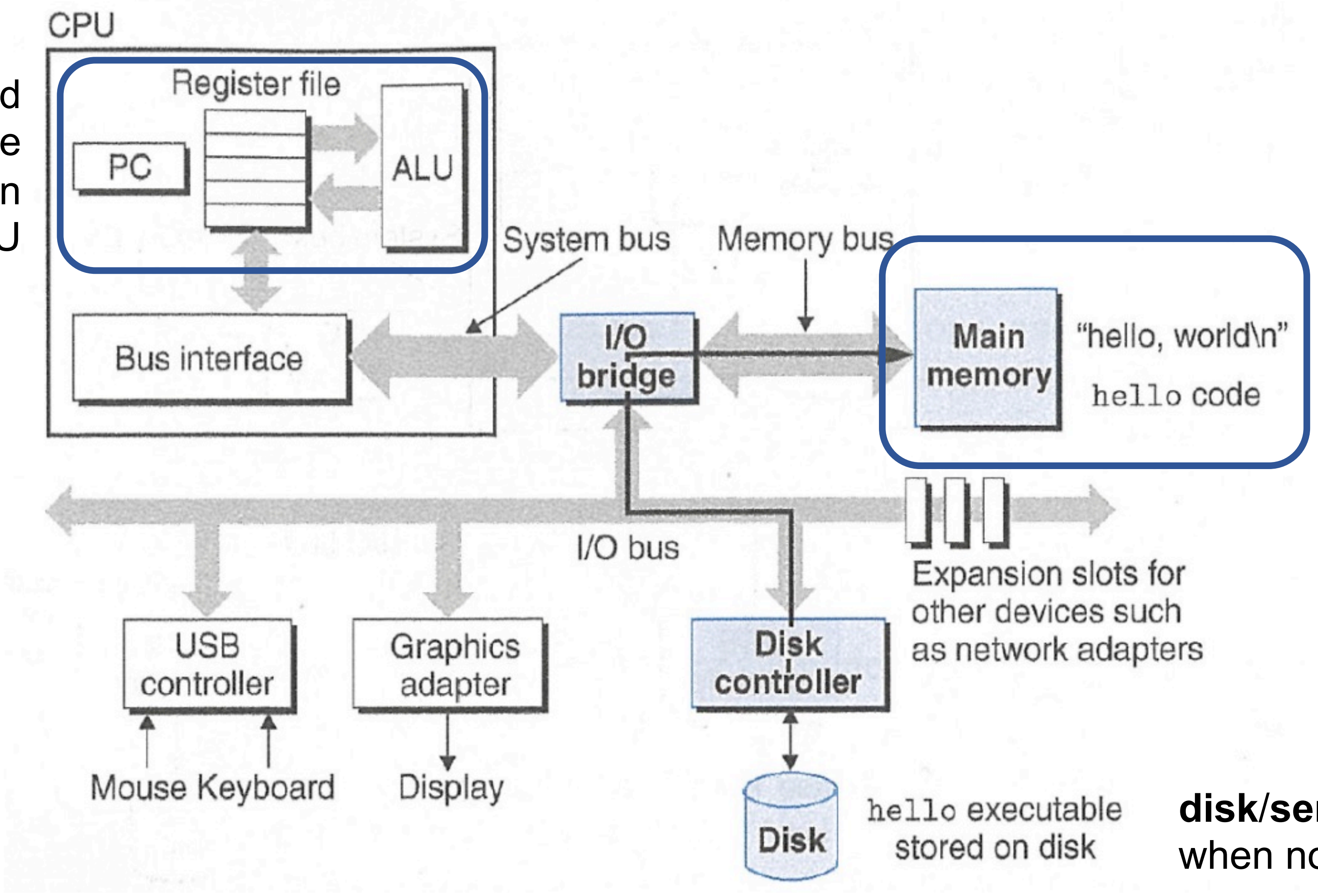
Registers

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!

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

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

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

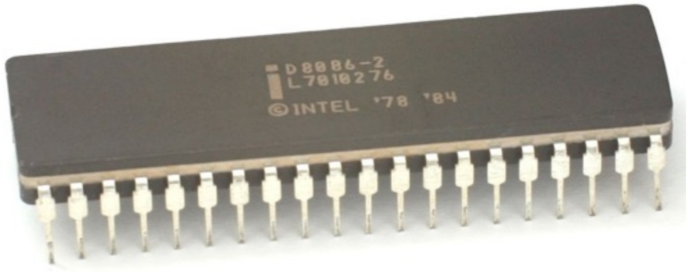
Instruction set architecture (ISA)

We are going to learn the **x86-64** instruction set architecture (used by Intel and AMD processors).

- There are many other instruction sets: ARM, MIPS, etc.

- An ISA is 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 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).

Application program	
Compiler	OS
ISA	
CPU design	
Circuit design	
Chip layout	



Aside: 32-to-64-bit Transition

- **Early 2000s:** most computers were **32-bit.** This means that pointers were **4 bytes (32 bits).**
- 32-bit pointers store a memory address from 0 to 2^{32}-1, equaling **2^{32} bytes of addressable memory**. This equals **4 Gigabytes**, meaning that 32-bit computers could have at most **4GB** of memory (RAM)!
- Because of this, computers transitioned to **64-bit.** This means that datatypes were enlarged; pointers in programs were now **64 bits.**
- 64-bit pointers store a memory address from 0 to 2^{64}-1, equaling **2^{64} bytes of addressable memory.** This equals **16 Exabytes**, meaning that 64-bit computers could have at most **1024*1024*1024*16 GB** of memory (RAM)!

Lecture Plan

- **Overview:** GCC and Assembly
- **Demo:** Looking at an executable
- Registers and The Assembly Level of Abstraction
- **The mov Instruction**

```
cp -r /afs/ir/class/cs107/lecture-code/lect15 .
```

mov

The **mov** instruction copies bytes from one place to another;
it is like 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***) `$0x104`
- Register `%rbx`
- Memory Location (*at most one of* ***src, dst***) Direct address `0x6005c0`

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 (**<u>0x10 plus</u>** 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 (**<u>0x10 plus</u>** 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 **<u>0x10 plus</u>** 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 **<u>0x10 plus</u>** 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.

For #3, respond with your thoughts on PollEv: pollev.com/cs107 or text CS107 to 22333 once to join.

```
1. mov    $0x42,(%rax)
2. mov    4(%rax),%rcx
3. mov    9(%rax,%rdx),%rcx
```

`Imm(`r_b`, `r_i`)` is equivalent to address `Imm + R[`r_b`] + R[`r_i`]`

Displacement: positive or negative constant (if missing, = 0)

Base: register (if missing, = 0)

Index: register (if missing, = 0)

Recap

- **Overview:** GCC and Assembly
- **Demo:** Looking at an executable
- Registers and The Assembly Level of Abstraction
- The **mov** instruction

Next time: diving deeper into assembly

Lecture 15 takeaway: Assembly is the human-readable version of the form our programs are ultimately executed in by the processor. The compiler translates source code to machine code. The most common assembly instruction is *mov* to move data around.

Extra Practice

1. Extra Practice

Fill in the blank to complete the C code that

1. generates this assembly
2. has this register layout

```
int x = ...
int *ptr = malloc(…);
...
___???___ = _???_;
```

```
mov %ecx,(%rax)
```

<val of x>	<val of ptr>
%ecx	%rax

(Pedantic: You should sub in <x> and <ptr> with actual values, like 4 and 0x7fff80) 🤔

1. Extra Practice

Fill in the blank to complete the C code that

1. generates this assembly
2. has this register layout

```
int x = ...
int *ptr = malloc(…);
...
___???___ = _???_;    *ptr = x;
```

```
mov %ecx,(%rax)
```

<val of x>	<val of ptr>
%ecx	%rax