# CS107, Lecture 10

## Introduction to Assembly

Reading: B&O 3.1-3.4

# What is Assembly Code?

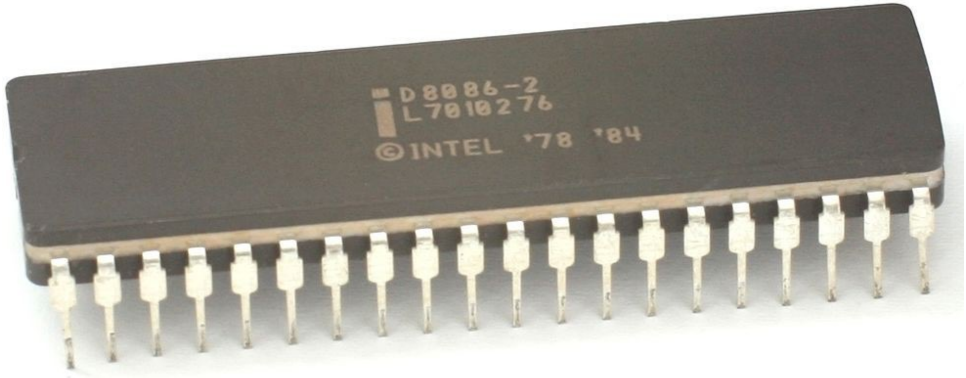
- Computers execute "machine code," which is a sequence of bytes that encode low-level operations for manipulating data, managing memory, read and write from storage, and communicate with networks.

- The "assembly code" for a computer is a textual representation of the machine code giving the individual instructions to the underlying machine.

# What is Assembly Code?

- `gcc` generates assembly code from C code
- Assembly is raw — there is no type checking, and the instructions are simple. It is unique to the type of processor (e.g., the assembly for your computer cannot run on your phone)
- Humans can write assembly (and, in fact, in the early days of computing they had to write assembly), but it is more productive to be able to read and understand what the compiler produces, than to write it by hand.
- `gcc` is almost always going to produce better optimized code than a human could, and understanding what the compiler produces is important.

# x86 Assembly



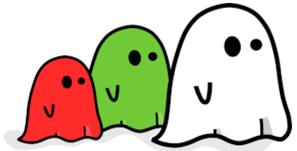
- The Intel-based computers we use are direct descendants of Intel's 16-bit, 1978 processor with the name 8086.
- Intel has taken a strict backwards-compatibility approach to new processors, and their 32- and 64-bit processors have built upon the original 8086 Assembly code.
- These days, when we learn x86 assembly code, we have to keep this history in mind. Naming of "registers," for example, has historical roots, so bear with it.

# Machine-Level Code

- Before we look at some assembly code, let's talk about some things that have been hidden from us when writing C code.
- Machine code is based on the "instruction set architecture" (ISA), which defines the behavior and layout of the system. Behavior is defined as if instructions are run one after the other, and memory appears as a very large byte array.

# Machine-Level Code

- New things that have been hidden:



  - The *program counter* (PC), called "`%rip`" indicates the address of the next instruction ("r"egister "i"nstruction "p"ointer". We cannot modify this directly.
  - The "register file" contains 16 named locations that store 64-bit values.

    Registers are the fastest memory on your computer. They *are not* in main memory, and *do not* have addresses. You cannot pass a pointer to a register, but a pointer may *hold* a register as its value.
    - Registers can hold addresses, or integer data. Some registers are used to keep track of your program's state, and others hold temporary data.
    - Registers are used for arithmetic, local variables, and return values for functions.
  - The condition code registers hold status information about the most recently executed arithmetic or logical instruction. These are used to control program flow — e.g., if the result of an addition is negative, exit a loop.
  - There are vector registers, which hold integer or floating point values.

# Machine-Level Code

- Unlike C, there is no model of different data types, and memory is simply a large, byte-addressable array.

- There is no distinction between signed and unsigned integers, between different types of pointers, or even between pointers and integers.

- A single machine instruction performs only a very elementary operation. For example:
  - there is an instruction to add two numbers in registers. That's all the instruction does.
  - there is an instruction that transfers data between a register and memory.
  - there is an instruction that conditionally branches to a new instruction address.

- Often, one C statement generates multiple assembly code instructions.

# 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

```
cp -r /afs/ir/class/cs107/lecture-code/lect10 .
```

# Lecture Plan

```
cp -r /afs/ir/class/cs107/lecture-code/lect10 .
```

# Bits all the way down

**Data representation so far**

- Integer (`unsigned int`, 2's complement signed `int`)
- `char` (ASCII)
- Address (unsigned long)
- Aggregates (arrays, `struct`s)

**The code itself is binary too!**

- Instructions (machine encoding)

# 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.

# Lecture Plan

```
cp -r /afs/ir/class/cs107/lecture-code/lect10 .
```

# 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

```
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    m
  401140:   39 f0             c
  401142:   7d 0b             j
  401144:   48 63 c8          m
  401147:   03 14 8f          a
  40114a:   83 c0 01          a
  40114d:   eb f1             j
  40114f:   89 d0             mov    %edx,%eax
  401151:   c3                retq
```

This is the machine code: 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. eax here).

# Lecture Plan

```
cp -r /afs/ir/class/cs107/lecture-code/lect10 .
```

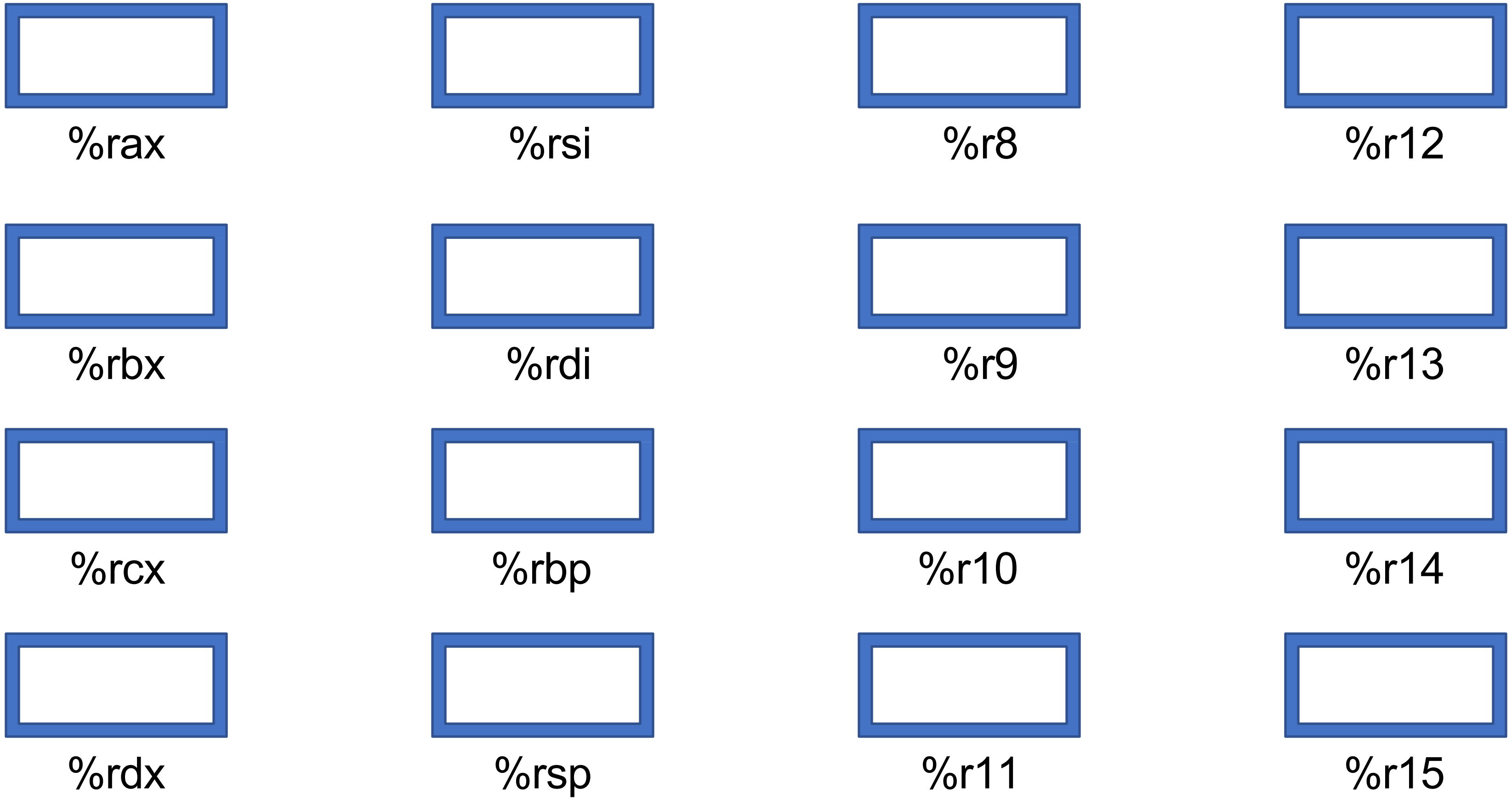
# Assembly Abstraction

- 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



%rax
%rbx
%rcx
%rdx

%rsi
%rdi
%rbp
%rsp

%r8
%r9
%r10
%r11

%r12
%r13
%r14
%r15

# 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 **not** located in 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!

# 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

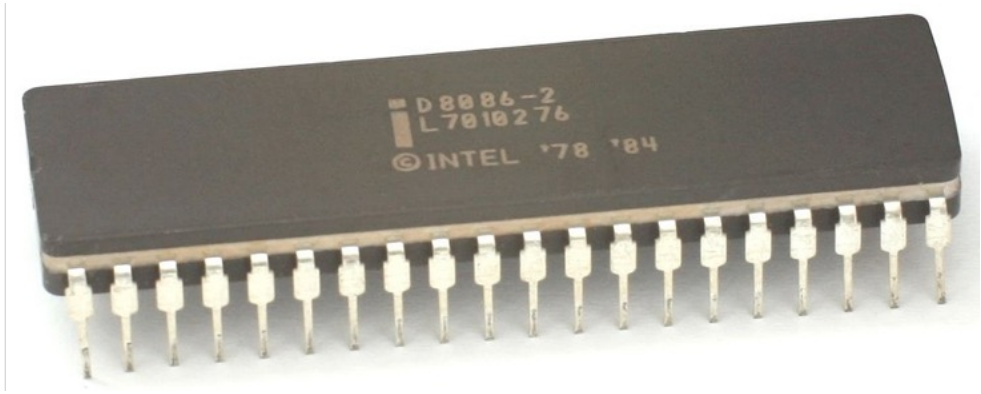
# GCC And Assembly

- 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<br>2) Copy y into register 2<br>3) Add register 2 to register 1<br>4) Write register 1 to memory for sum |

# Assembly

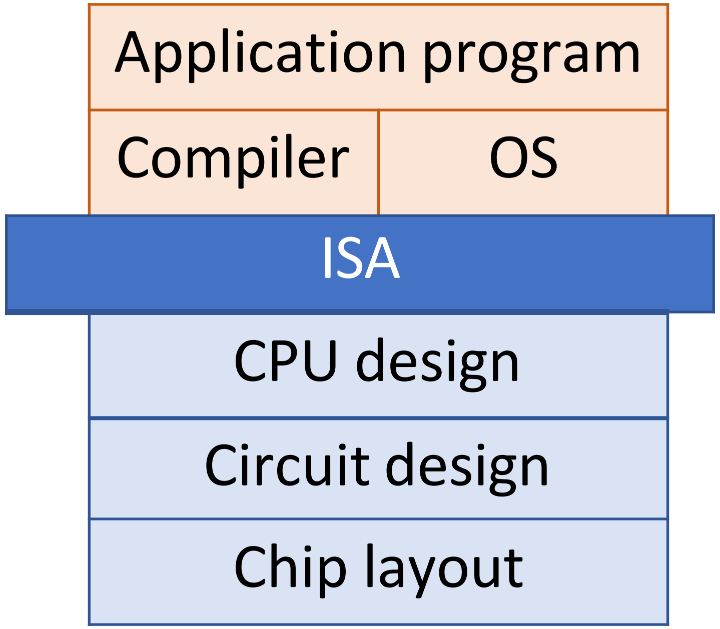
- We are going to learn the **x86-64** instruction set architecture. This instruction set is used by Intel and AMD processors.
- There are many other instruction sets: ARM, MIPS, etc.



# 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



Application program

Compiler | OS

ISA

CPU design

Circuit design

Chip layout

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).

# Lecture Plan

```
cp -r /afs/ir/class/cs107/lecture-code/lect10 .
```

# mov

The **mov** instruction copies bytes from one place to another;
it is similar to 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 (**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
```

$\texttt{Imm}(\texttt{r}_\texttt{b},\ \texttt{r}_\texttt{i})$ is equivalent to address $\texttt{Imm} + \texttt{R}[\texttt{r}_\texttt{b}] + \texttt{R}[\texttt{r}_\texttt{i}]$

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

**Base**: register (if missing, = 0)

**Index**: register (if missing, = 0)

# 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 (**the value in register %rax** 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 (**the value in register %rax** 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…*

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

# Most General Operand Form

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

**Displacement**: pos/neg constant (if missing, = 0)

**Base**: register (if missing, = 0)

**Index**: register (if missing, = 0)

**Scale** must be 1,2,4, or 8 (if missing, = 1)

# Operand Forms

| Type | Form | Operand Value | Name |
|---|---|---|---|
| Immediate | $\$Imm$ | $Imm$ | Immediate |
| Register | $r_{!}$ | $\mathrm{R}[r_{!}]$ | Register |
| Memory | $Imm$ | $\mathrm{M}[Imm]$ | Absolute |
| Memory | $(r_{!})$ | $\mathrm{M}[\mathrm{R}[r_{!}]]$ | Indirect |
| Memory | $Imm(r_{"})$ | $\mathrm{M}[Imm + \mathrm{R}[r_{"}]]$ | Base + displacement |
| Memory | $(r_{"}, r_{\#})$ | $\mathrm{M}[\mathrm{R}[r_{"}] + \mathrm{R}[r_{\#}]]$ | Indexed |
| Memory | $Imm(r_{"}, r_{\#})$ | $\mathrm{M}[Imm + \mathrm{R}[r_{"}] + \mathrm{R}[r_{\#}]]$ | Indexed |
| Memory | $(, r_{\#}, s)$ | $\mathrm{M}[\mathrm{R}[r_{\#}] \;.\; s]$ | Scaled indexed |
| Memory | $Imm(, r_{\#}, s)$ | $\mathrm{M}[Imm + \mathrm{R}[r_{\#}] \;.\; s]$ | Scaled indexed |
| Memory | $(r_{"}, r_{\#}, s)$ | $\mathrm{M}[\mathrm{R}[r_{"}] + \mathrm{R}[r_{\#}] \;.\; s]$ | Scaled indexed |
| Memory | $Imm(r_{"}, r_{\#}, s)$ | $\mathrm{M}[Imm + \mathrm{R}[r_{"}] + \mathrm{R}[r_{\#}] \;.\; s]$ | 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
```

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

Displacement Base Index Scale

(1,2,4,8)

# Goals of indirect addressing: C

Why are there so many forms of indirect addressing?

We see these indirect addressing paradigms in C as well!

# 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;
}
```

We’re 1/4th of the way to understanding assembly!
**What looks understandable right now?**

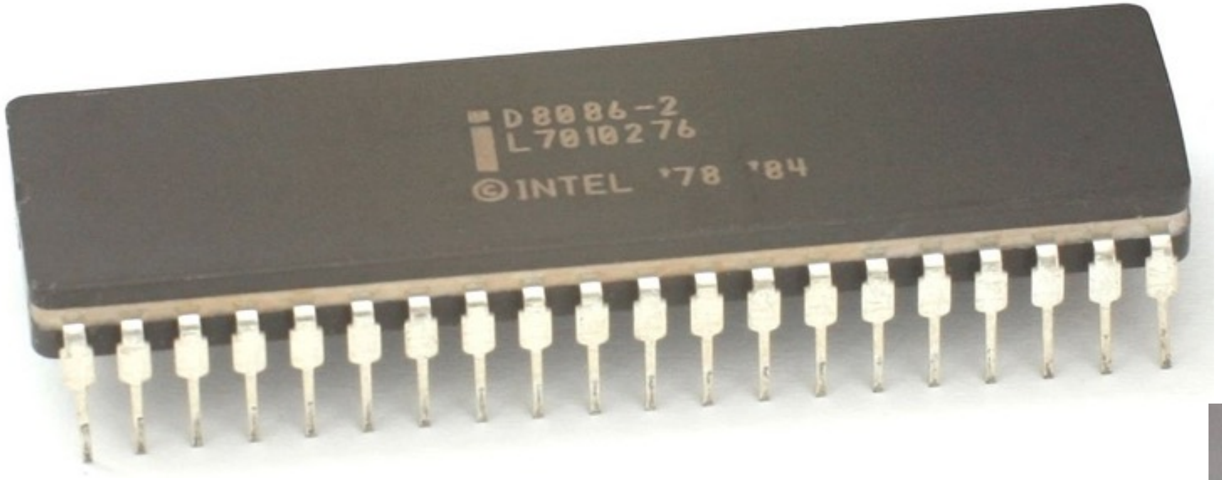
Some notes:

- Registers store addresses and values
- `mov src, dst` ***copies*** value into dst
- `sizeof(int)` is 4
- Instructions executed sequentially

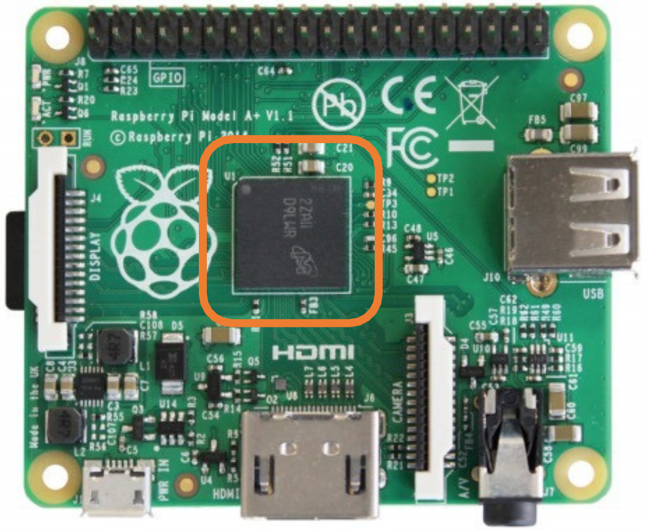
```
00000000004005b6 <sum_array>:
  4005b6:    ba 00 00 00 00       mov    $0x0,%edx
  4005bb:    b8 00 00 00 00       mov    $0x0,%eax
  4005c0:    eb 09                jmp    4005cb <sum_array+0x15>
  4005c2:    48 63 ca             movslq %edx,%rcx
  4005c5:    03 04 8f             add    (%rdi,%rcx,4),%eax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

We’ll come back to this example in future lectures!

# Central Processing Units (CPUs)



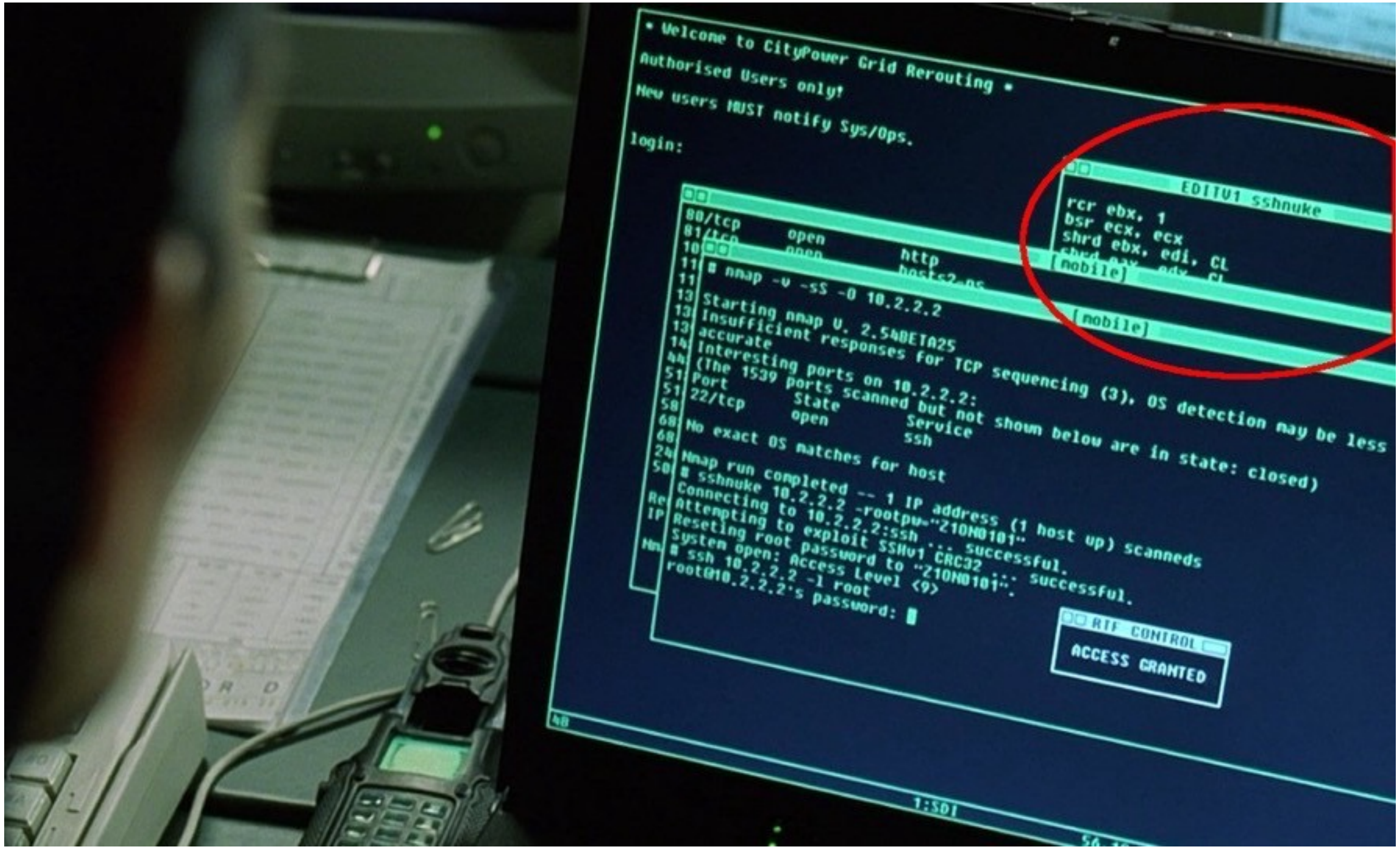
Intel 8086, 16-bit
microprocessor
($86.65, 1978)



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



```
* Welcome to CityPower Grid Rerouting *
Authorised Users only!
New users MUST notify Sys/Ops.

login:
```

```
rcr ebx, 1
bsr ecx, ecx
shrd ebx, edi, CL
```

```
80/tcp
81/tcp      open        http
```

```
# nmap -v -sS -O 10.2.2.2
Starting nmap V. 2.54BETA25
Insufficient responses for TCP sequencing (3), OS detection may be less
accurate
Interesting ports on 10.2.2.2:
(The 1539 ports scanned but not shown below are in state: closed)
Port       State       Service
22/tcp     open        ssh
No exact OS matches for host
Nmap run completed -- 1 IP address (1 host up) scanneds
# sshnuke 10.2.2.2 -rootpw="Z10N0101"
Connecting to 10.2.2.2:ssh ... successful.
Attempting to exploit SSHv1 CRC32 ... successful.
Reseting root password to "Z10N0101".
System open: Access Level <9>
# ssh 10.2.2.2 -l root
root@10.2.2.2's password:
```

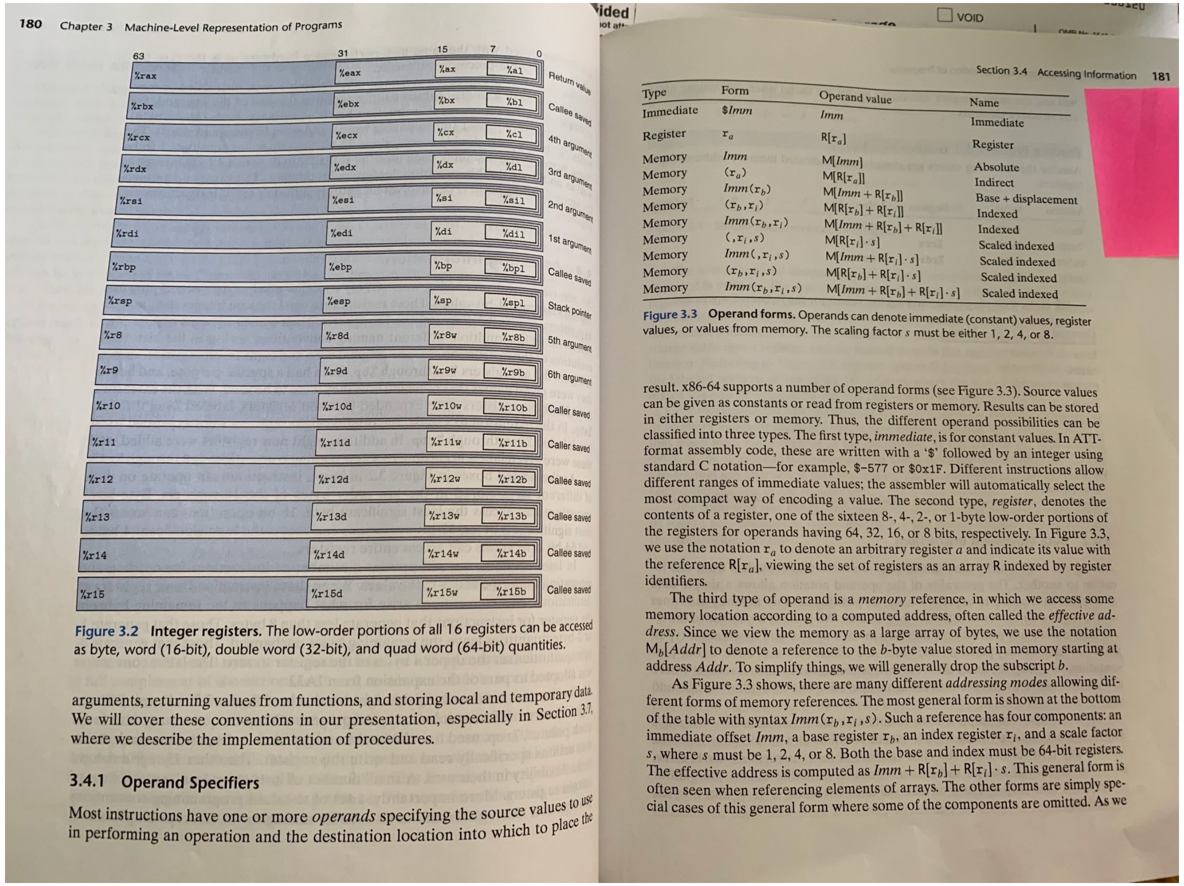
Trinity saving the world by hacking into the power grid using Nmap Network Scanning
*The Matrix Reloaded*, 2003

# 🌟 Keep a resource guide handy 🌟

- https://web.stanford.edu/class/cs107/resources/x86-64-reference.pdf
- B&O book:
  - Canvas -> Files
    -> Bryant_OHallaron_ch3.1-3.8.pdf
- It's like study abroad:

BING OVERSEAS
STUDIES PROGRAM
STANFORD UNIVERSITY

  - You took LANG 1A
  - Your tools give too much/too little information (a book reference, a rudimentary translator)
  - No one expects you to **speak** the language fluently…
  - …But the more you internalize,
    the better you can use tools to **read** the language



180 Chapter 3 Machine-Level Representation of Programs

| 63 | 31 | 15 | 7 … 0 | |
|---|---|---|---|---|
| %rax | %eax | %ax | %al | Return value |
| %rbx | %ebx | %bx | %bl | Callee saved |
| %rcx | %ecx | %cx | %cl | 4th argument |
| %rdx | %edx | %dx | %dl | 3rd argument |
| %rsi | %esi | %si | %sil | 2nd argument |
| %rdi | %edi | %di | %dil | 1st argument |
| %rbp | %ebp | %bp | %bpl | Callee saved |
| %rsp | %esp | %sp | %spl | Stack pointer |
| %r8 | %r8d | %r8w | %r8b | 5th argument |
| %r9 | %r9d | %r9w | %r9b | 6th argument |
| %r10 | %r10d | %r10w | %r10b | Caller saved |
| %r11 | %r11d | %r11w | %r11b | Caller saved |
| %r12 | %r12d | %r12w | %r12b | Callee saved |
| %r13 | %r13d | %r13w | %r13b | Callee saved |
| %r14 | %r14d | %r14w | %r14b | Callee saved |
| %r15 | %r15d | %r15w | %r15b | Callee saved |

**Figure 3.2** **Integer registers.** The low-order portions of all 16 registers can be accessed as byte, word (16-bit), double word (32-bit), and quad word (64-bit) quantities.

arguments, returning values from functions, and storing local and temporary data. We will cover these conventions in our presentation, especially in Section 3.7, where we describe the implementation of procedures.

### 3.4.1 Operand Specifiers

Most instructions have one or more *operands* specifying the source values to use in performing an operation and the destination location into which to place the

Section 3.4 Accessing Information 181

| 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.

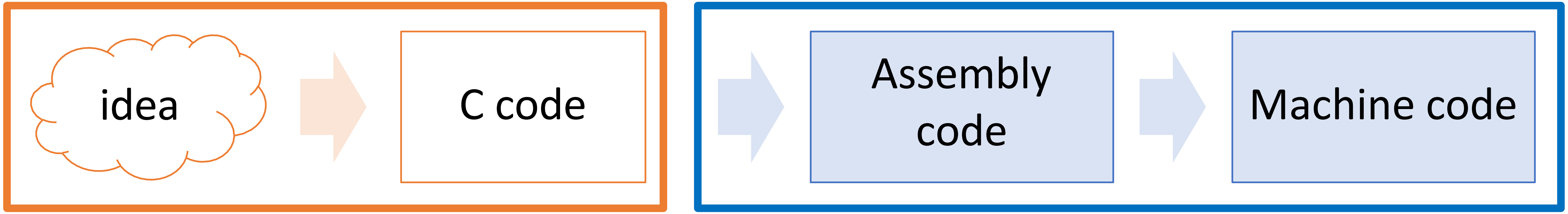
result. x86-64 supports a number of operand forms (see Figure 3.3). Source values can be given as constants or read from registers or memory. Results can be stored in either registers or memory. Thus, the different operand possibilities can be classified into three types. The first type, *immediate*, is for constant values. In ATT-format assembly code, these are written with a '\$' followed by an integer using standard C notation—for example, `$-577` or `$0x1F`. Different instructions allow different ranges of immediate values; the assembler will automatically select the most compact way of encoding a value. The second type, *register*, denotes the contents of a register, one of the sixteen 8-, 4-, 2-, or 1-byte low-order portions of the registers for operands having 64, 32, 16, or 8 bits, respectively. In Figure 3.3, we use the notation $r_a$ to denote an arbitrary register *a* and indicate its value with the reference $R[r_a]$, viewing the set of registers as an array R indexed by register identifiers.

The third type of operand is a *memory* reference, in which we access some memory location according to a computed address, often called the *effective address*. Since we view the memory as a large array of bytes, we use the notation $M_b[Addr]$ to denote a reference to the *b*-byte value stored in memory starting at address *Addr*. To simplify things, we will generally drop the subscript *b*.

As Figure 3.3 shows, there are many different *addressing modes* allowing different forms of memory references. The most general form is shown at the bottom of the table with syntax $Imm(r_b,r_i,s)$. Such a reference has four components: an immediate offset *Imm*, a base register $r_b$, an index register $r_i$, and a scale factor $s$, where $s$ must be 1, 2, 4, or 8. Both the base and index must be 64-bit registers. The effective address is computed as $Imm + R[r_b] + R[r_i] \cdot s$. This general form is often seen when referencing elements of arrays. The other forms are simply special cases of this general form where some of the components are omitted. As we

Chapter 3, Figures 3.2-3.3 (p. 180-181)

# Why are we reading assembly?



Programmer-generated

gcc (compiler+assembler) generated

**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.

# TADA!

And that's it for today!

# Extended warmup: Information Synthesis

Spend a few minutes thinking about the main paradigms of the `mov` instruction.

- What might be the equivalent C-like operation?
- Examples (note %r__ registers are 64-bit):

1. `mov     $0x0,%rdx`
2. `mov     %rdx,%rcx`
3. `mov     $0x42,(%rdi)`
4. `mov     (%rax,%rcx,8),%rax`

🤔

# Extended warmup: Information Synthesis

Spend a few minutes thinking about the main paradigms of the `mov` instruction.

- What might be the equivalent C-like operation?
- Examples (note %r__ registers are 64-bit):

Indirect addressing is like pointer arithmetic/deref!

1. `mov    $0x0,%rdx -> maybe long x = 0`
2. `mov    %rdx,%rcx -> maybe long x = y;`
3. `mov    $0x42,(%rdi) -> maybe *ptr = 0x42;`
4. `mov    (%rax,%rcx,8),%rax -> maybe long x = arr[i];`

# 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>`

%ecx

`<val of ptr>`

%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>`

%ecx

`<val of ptr>`

%rax

# 2. Extra Practice

Fill in the blank to complete the C code that

1. generates this assembly
2. **results in** this register layout

```
long arr[5];
...
long num = ____???___;
```

---

```
mov (%rdi, %rcx, 8),%rax
```

| <val of num> | 3 | <val of arr> |
|---|---|---|
| %rax | %rcx | %rdi |

# 2. Extra Practice

Fill in the blank to complete the C code that

1. generates this assembly
2. **results in** this register layout

```
long arr[5];
...
long num = ____???___;
```

```
long num = arr[3];
long num = *(arr + 3);
long num = *(arr + y);
```

(assume `long y = 3;` declared earlier)

```
mov (%rdi, %rcx, 8),%rax
```

| `<val of num>` | 3 | `<val of arr>` |
|---|---|---|
| %rax | %rcx | %rdi |

# 3. Extra Practice

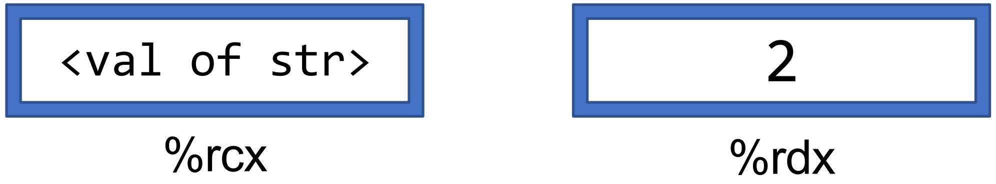
Fill in the blank to complete the C code that

1. generates this assembly
2. has this register layout

```
char str[5];
...
___???___ = 'c';
```

---

```
mov $0x63,(%rcx,%rdx,1)
```



# 3. Extra Practice

Fill in the blank to complete the C code that

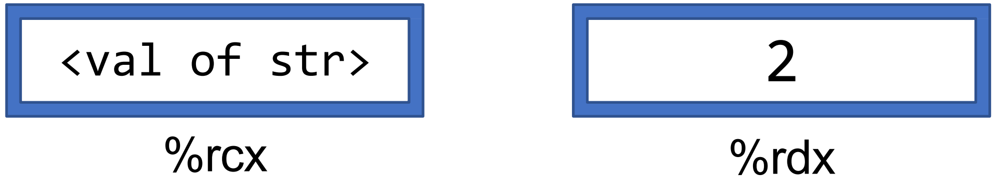
1. generates this assembly
2. has this register layout

```
char str[5];
...
___???___ = 'c';
```

```
str[2] = 'c';
*(str + 2) = 'c';
```

---

```
mov $0x63,(%rcx,%rdx,1)
```



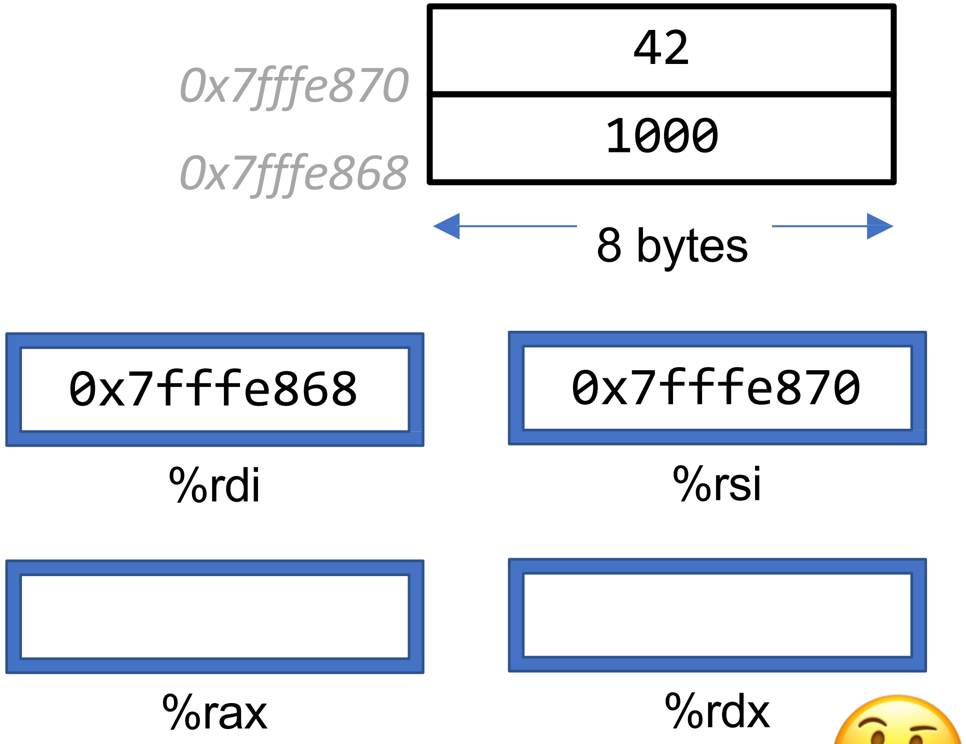
# Coming Up Soon To A Slide Near You

- The below code is the objdump of a C function, `foo`.
  - `foo` keeps its 1st and 2nd parameters are in registers %rdi and %rsi, respectively.

```
0x4005b6 <foo>      mov    (%rdi),%rax
0x4005b9 <foo+3>    mov    (%rsi),%rdx
0x4005bc <foo+6>    mov    %rdx,(%rdi)
0x4005bf <foo+9>    mov    %rax,(%rsi)
```

1. **What does this function do?**
2. **What C code could have generated this assembly?**

(Hints: make up C variable names as needed, assume all regs 64-bit)



*0x7fffe870* 42

*0x7fffe868* 1000

8 bytes

%rdi: 0x7fffe868

%rsi: 0x7fffe870

%rax

%rdx

# Coming Up Soon To A Slide Near You

- The below code is the objdump of a C function, `foo`.
    - `foo` keeps its 1st and 2nd parameters are in registers %rdi and %rsi, respectively.

```
0x4005b6 <foo>      mov     (%rdi),%rax
0x4005b9 <foo+3>    mov     (%rsi),%rdx
0x4005bc <foo+6>    mov     %rdx,(%rdi)
0x4005bf <foo+9>    mov     %rax,(%rsi)
```