

CS107, Lecture 10

Introduction to Assembly

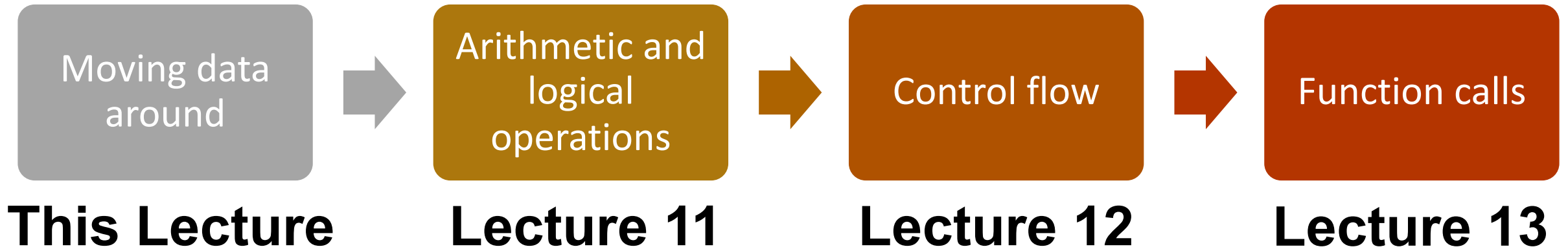
Reading: B&O 3.1-3.4

Course Overview

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

CS107 Topic 5: How does a computer interpret and execute C programs?

Learning Assembly



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 7
- **Demo:** Looking at an executable 11
- Registers and The Assembly Level of Abstraction 24
- The **mov** Instruction 35
- Live Session 57

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

Lecture Plan

- **Overview: GCC and Assembly** 7
- **Demo:** Looking at an executable 11
- Registers and The Assembly Level of Abstraction 24
- The **mov** Instruction 35
- Live Session 57

```
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, structs)

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

- **Overview:** GCC and Assembly 7
- **Demo: Looking at an executable** 11
- Registers and The Assembly Level of Abstraction 24
- The **mov** Instruction 35
- Live Session 57

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

Demo: Looking at an Executable (objdump -d)



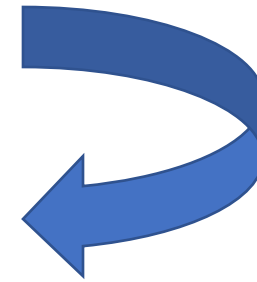
Our First Assembly

```
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

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	j1	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
```

Our First Assembly

0000000004005b6 <sum_array>:

4005b6: b2 00 00 00 00

4005bb: b8 00 00 00 00

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

4005cb: 39 f2

4005cd: 7c f3

4005cf: f3 c3

mov \$0x0,%edx

mov \$0x0,%eax

mp 4005cb <sum_array+0x15>

ovslq %edx,%rcx

dd (%rdi,%rcx,4),%eax

dd \$0x1,%edx

cmp %esi,%edx

jl 4005c2 <sum_array+0xc>

repz retq

Our First Assembly

000000004005b6 <sum_array>:

```
4005b6: ba 00 00 00 00      mov     $0x0,%edx
4005bb: b8 00 00 00 00      mov     $0x0,%eax
4005c0: eb 00              jmp     4005c2 <sum_array+0x15>
4005c2: 4005c2: 00 00 00 00      mov     $0x0,%eax
4005c5: 00 00 00 00      mov     $0x0,%eax
4005c8: 80 00              jmp     4005cb <sum_array+0x18>
4005cb: 31 00              jmp     4005cd <sum_array+0x1b>
4005cd: 7c f3              j1     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


4005bb: b8 00 00 00 00

4005c0: cb 00

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

4005cd: 7c f3

4005cf: f3 c3




```
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
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
```



mov \$0x0,%edx

This is the machine code: raw hexadecimal instructions, representing binary as read by the computer. Different instructions may be different byte lengths.

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
```

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          jle    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	<u>%esi,%edx</u>
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	%edi,%edx
4005cd:	7c f3	jl	4005c2 <sum_array+0xc>
4005cf:	f3 c3	repz	retq



\$[number] means a constant value, or “immediate” (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, a storage location on the CPU (e.g. edx here).

Lecture Plan

- **Overview:** GCC and Assembly 7
- **Demo:** Looking at an executable 11
- **Registers and The Assembly Level of Abstraction** 24
- The **mov** instruction 35

```
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



%rsi



%r8



%r12



%rbx



%rdi



%r9



%r13



%rcx



%rbp



%r10



%r14



%rdx



%rsp



%r11



%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!

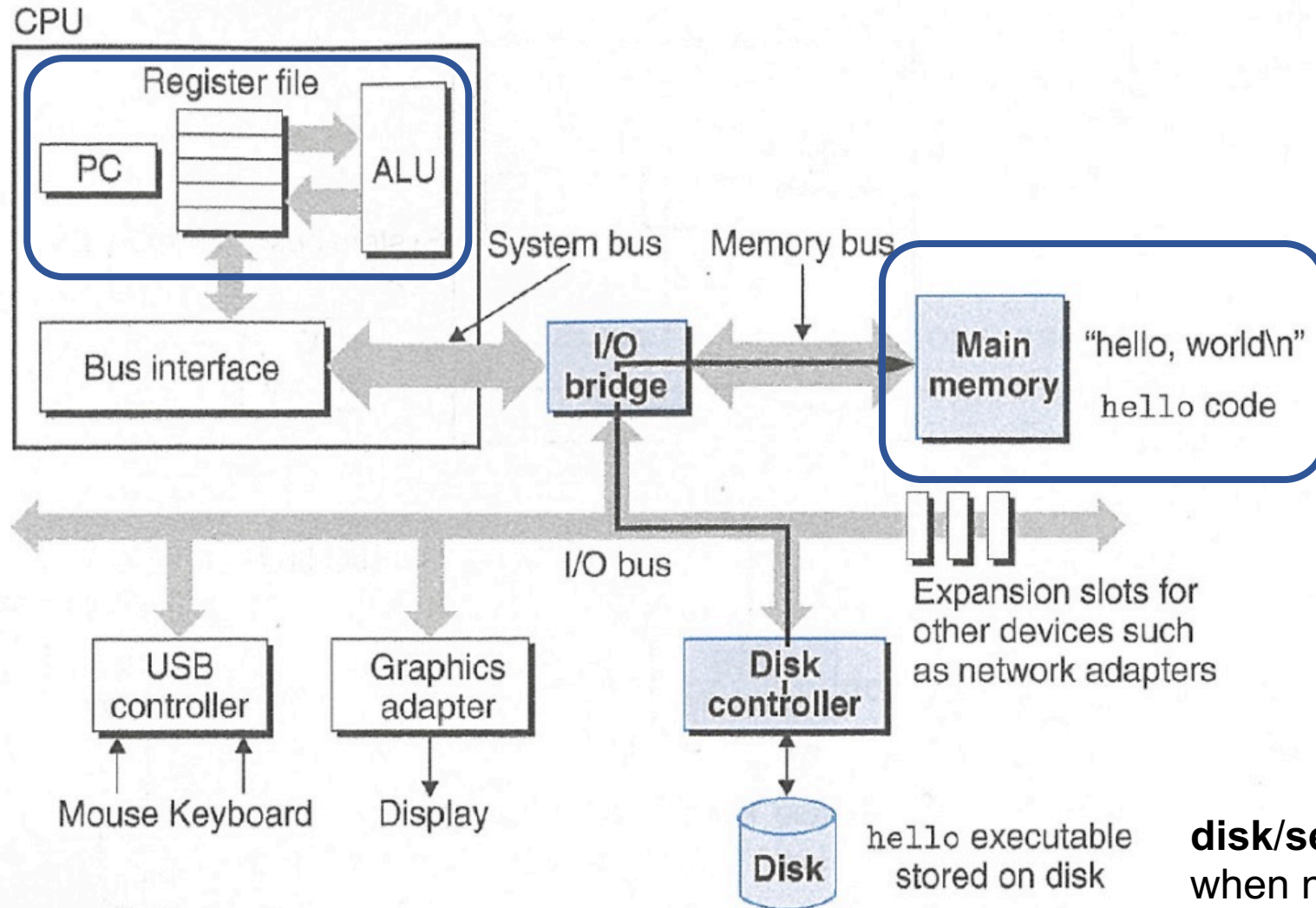
Machine-Level Code

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

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

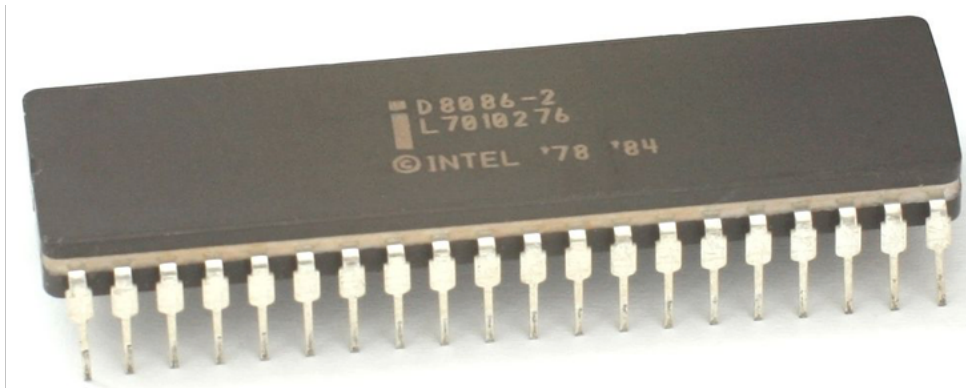
GCC And Assembly

- 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;	<ol style="list-style-type: none">1) Copy x into register 12) Copy y into register 23) Add register 2 to register 14) 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.



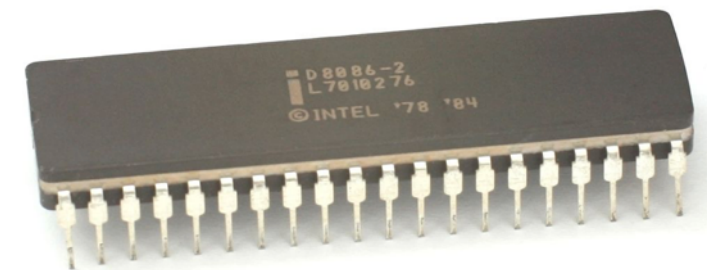
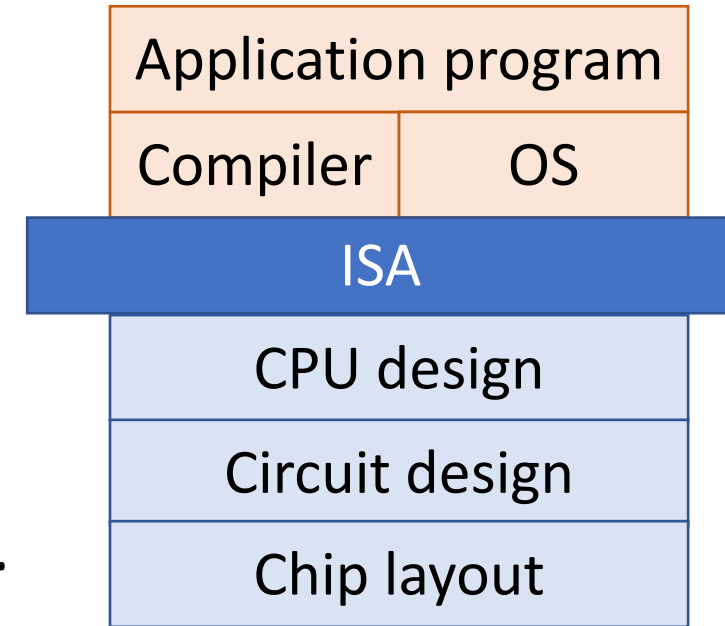
Instruction set architecture (ISA)

A contract between program/compiler and hardware:

- Defines operations that the processor (CPU) can execute
- Data read/write/transfer operations
- Control mechanisms

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

- **Overview:** GCC and Assembly 7
- **Demo:** Looking at an executable 11
- Registers and The Assembly Level of Abstraction 24
- **The mov Instruction** 35
- Live Session 57

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

\$0x104

%rbx

Direct address **0x6005c0**

Operand Forms: Immediate

mov **\$0x104, _____**



*Copy the value
0x104 into some
destination.*

Operand Forms: Registers

mov

%rbx, _____

Copy the value in register %rbx into some destination.

mov

_____, %rbx

Copy the value from some source into register %rbx.

Operand Forms: Absolute Addresses

mov **0x104**, _____

Copy the value at address 0x104 into some destination.

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**

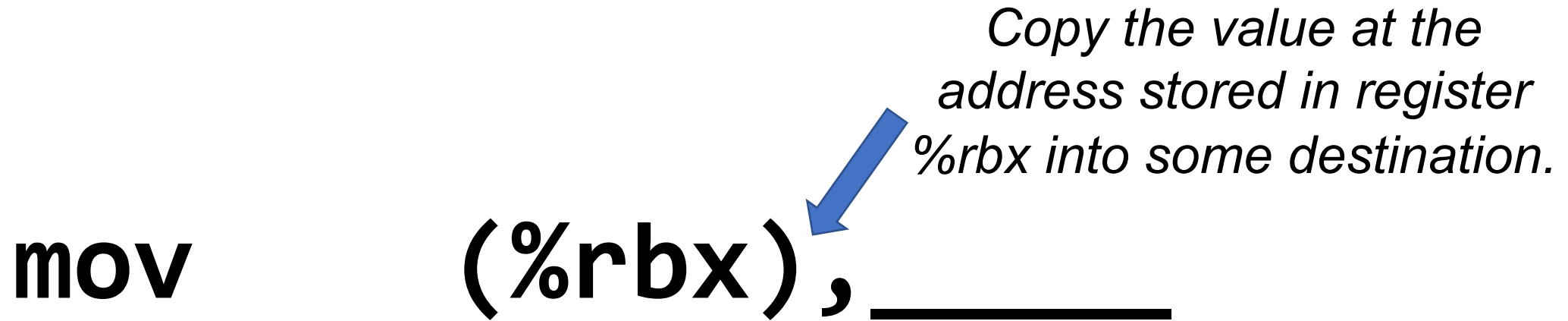
2. **mov 0x42,%rax**

3. **mov %rbx,0x55**

Operand Forms: Indirect

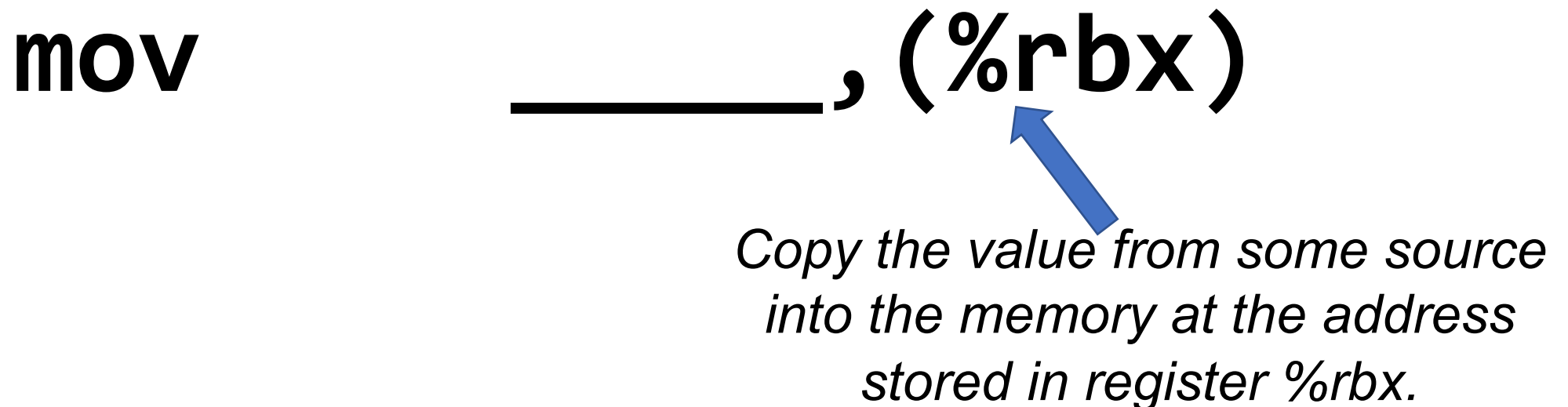
mov **(%rbx), _____**

Copy the value at the address stored in register %rbx into some destination.



mov **_____, (%rbx)**

Copy the value from some source into the memory at the address stored in register %rbx.



Operand Forms: Base + Displacement

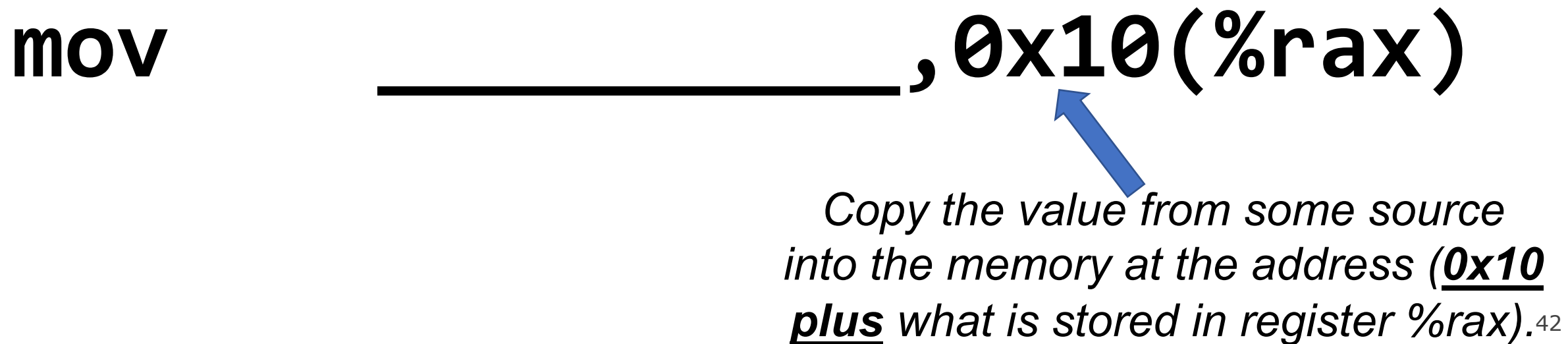
mov **0x10(%rax), _____**

Copy the value at the address (0x10 plus what is stored in register %rax) into some destination.



mov **_____, 0x10(%rax)**

Copy the value from some source into the memory at the address (0x10 plus what is stored in register %rax).⁴²



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**

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

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

Base: register (if missing, = 0)

Index: register (if missing, = 0)

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

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

is equivalent to...

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

Most General Operand Form

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

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_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 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`

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



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

Live Session Slides

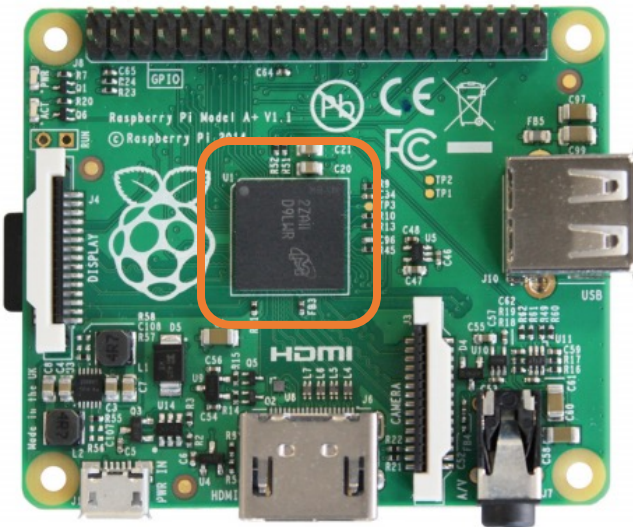
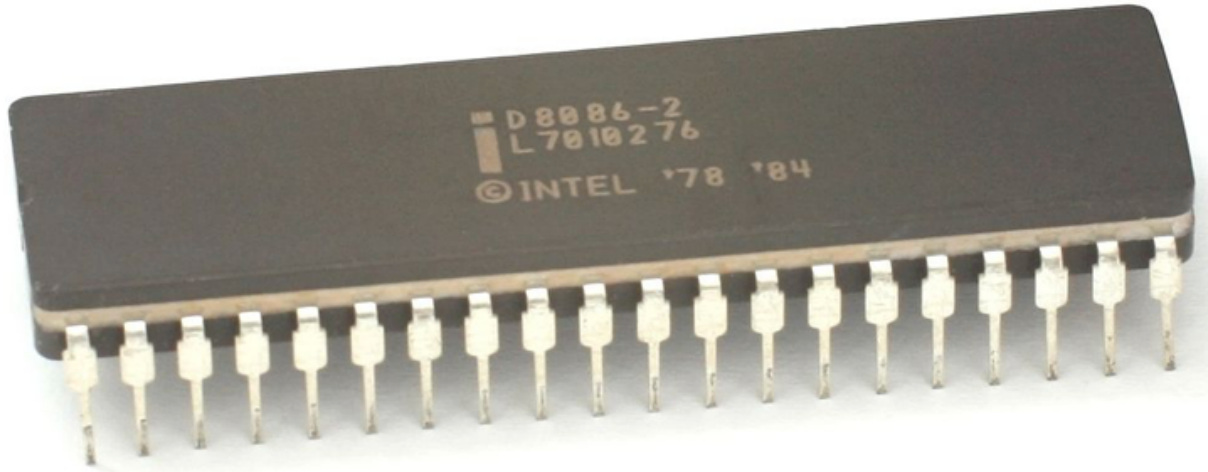
Plan For Today

- **10 minutes:** general review
- **5 minutes:** post questions or comments on Ed for what we should discuss
- **15 minutes:** open Q&A
- **25 minutes:** extra practice

Lecture 10 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.

Central Processing Units (CPUs)

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

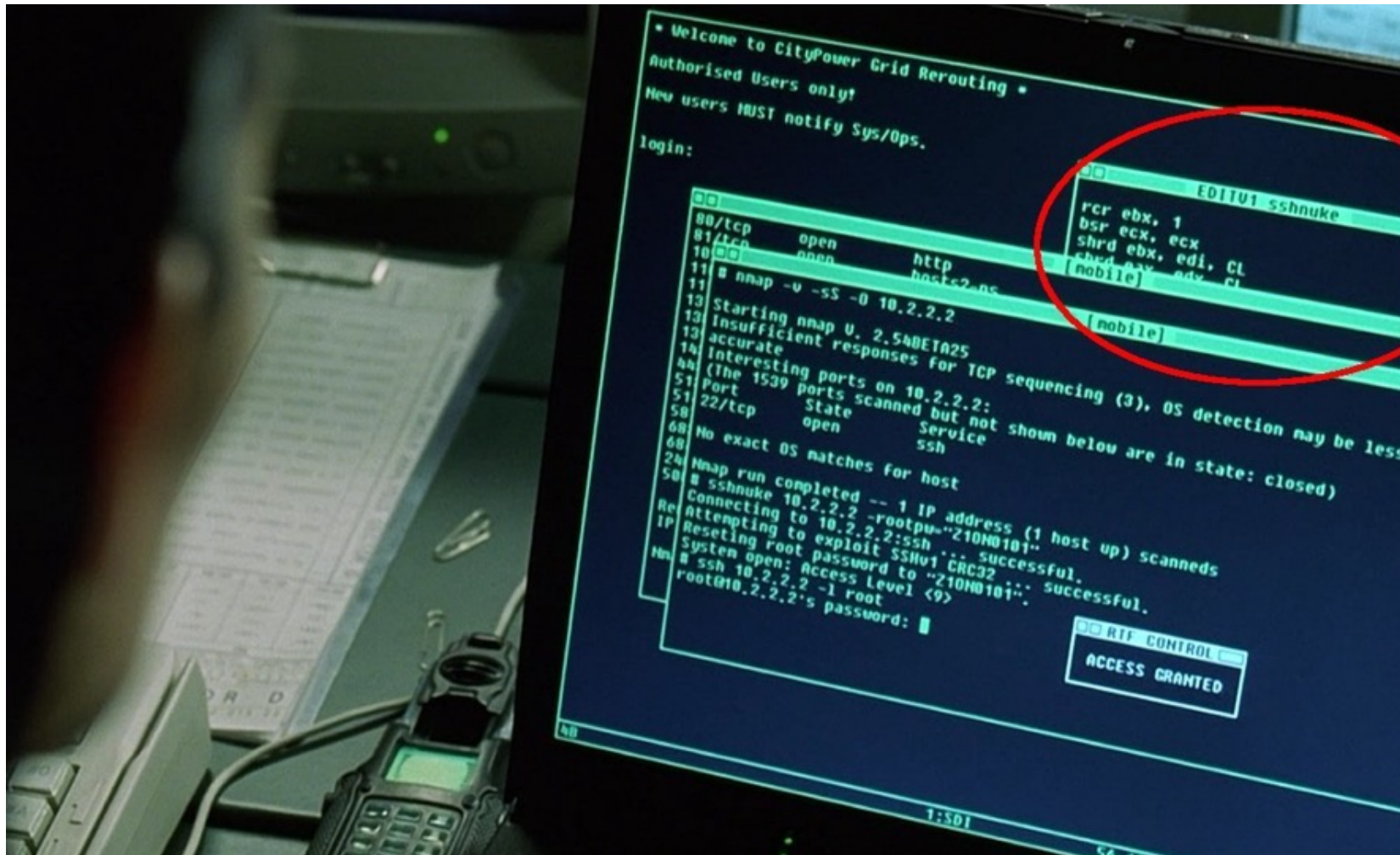


Raspberry Pi BCM2836 32-bit **ARM** microprocessor (\$35 for everything, 2015)



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

★ 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:
 - 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

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	(r_b, r_c)	$M[Imm + R[r_c]]$	Base + displacement
Memory	(r_b, r_c)	$M[R[r_a] + R[r_c]]$	Indexed
Memory	$Imm(r_b, r_c)$	$M[Imm + R[r_a] + R[r_c]]$	Indexed
Memory	$(, r_1, s)$	$M[R[r_1] \cdot s]$	Scaled indexed
Memory	$Imm(, r_1, s)$	$M[Imm + R[r_1] \cdot s]$	Scaled indexed
Memory	(r_b, r_1, s)	$M[R[r_a] + R[r_1] \cdot s]$	Scaled indexed
Memory	$Imm(r_b, r_1, s)$	$M[Imm + R[r_a] + R[r_1] \cdot s]$	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.

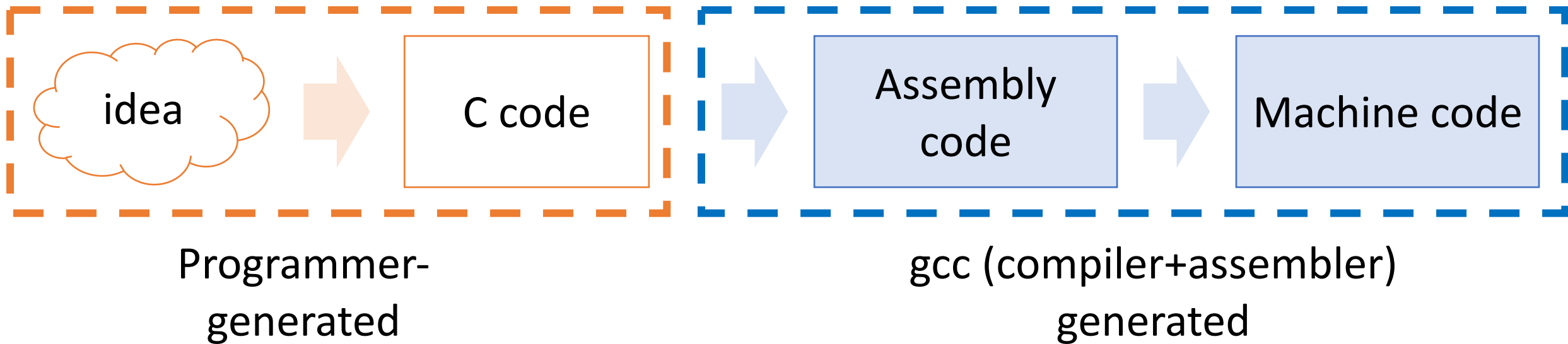
result. x86-64 supports a number of operand forms (see Figure 3.3). Source values can be given as constants or read from registers or memory. Results can be stored in either registers or memory. Thus, the different operand possibilities can be classified into three types. The first type, *immediate*, 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 $Imm(r_b, r_c, s)$. Such a reference has four components: an immediate offset Imm , a base register r_b , an index register r_c , 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 $Imm + R[r_b] + R[r_c] \cdot s$. 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?



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.

Plan For Today

- **10 minutes:** general review
- **5 minutes:** post questions or comments on Ed for what we should discuss
- **15 minutes:** open Q&A
- **25 minutes:** extra practice

Lecture 10 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.

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):

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];`

Indirect addressing
is like pointer
arithmetic/deref!



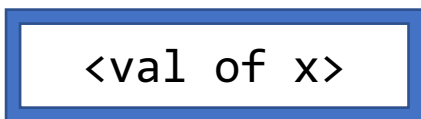
1. Extra Practice

Fill in the blank to complete the C code that

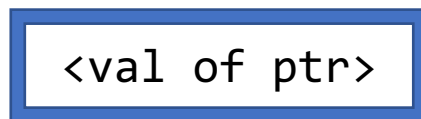
1. generates this assembly
2. has this register layout

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

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



%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>

%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>

%rax

3

%rcx

<val of arr>

%rdi



2. Extra Practice

Fill in the blank to complete the C code that

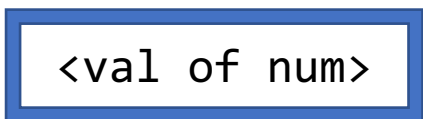
1. generates this assembly
2. **results in** this register layout

```
long arr[5];  
...  
long num = _____???
```

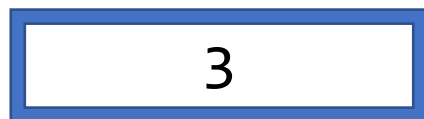
```
long num = arr[3];  
long num = *(arr + 3);  
long num = *(arr + y);
```

(assume long y = 3;
declared earlier)

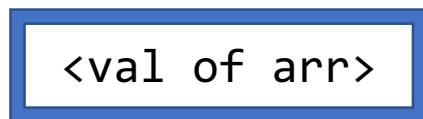
```
mov (%rdi, %rcx, 8),%rax
```



%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)
```

<val of str>

%rcx

2

%rdx



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)
```

<val of str>

%rcx

2

%rdx

Bonus: Sneak peek into Lecture 11

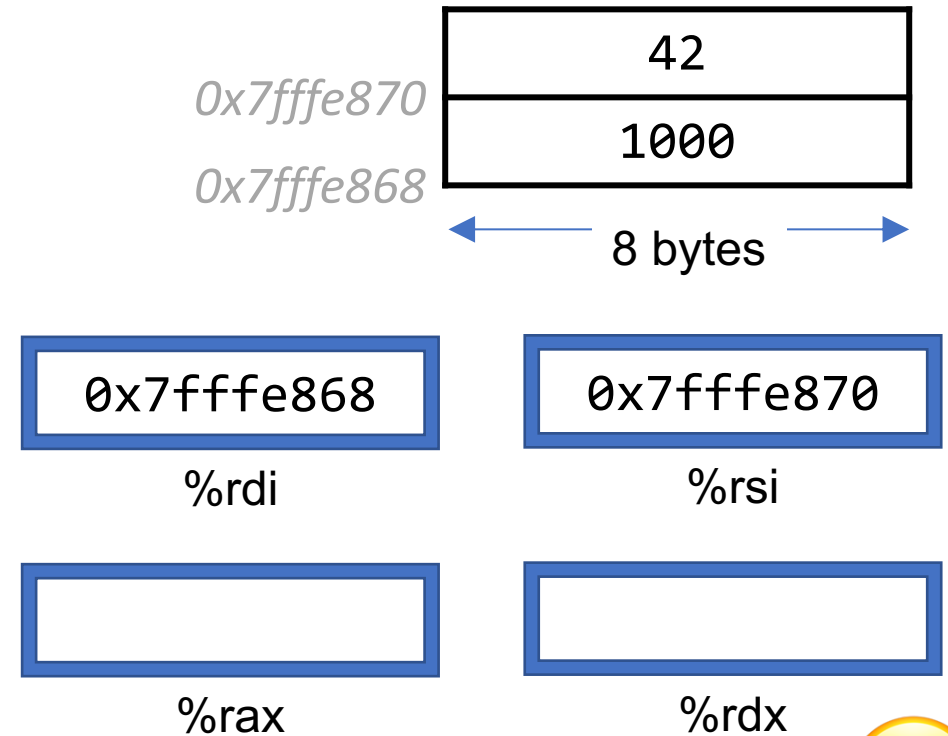
- The below code is the objdump of a C function, foo.
 - foo keeps its 1st and 2nd parameters in registers %rdi and %rsi, respectively.

```
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)



Bonus: Sneak peek into Lecture 11

- 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)
```

