



CS107, Lecture 14

Introduction to Assembly

Reading: B&O 3.1-3.4

[Ed Discussion](#)

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?**

CS107 Topic 5

How does a computer interpret and execute C programs?

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 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.

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!

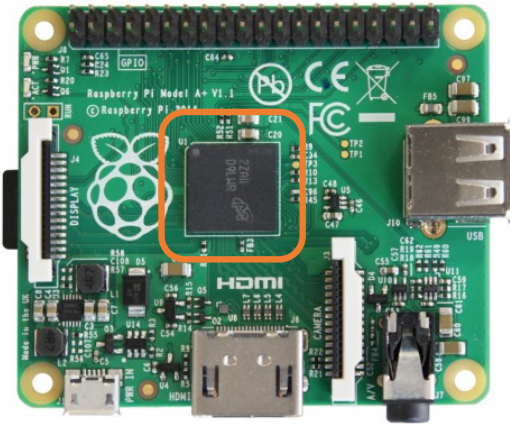
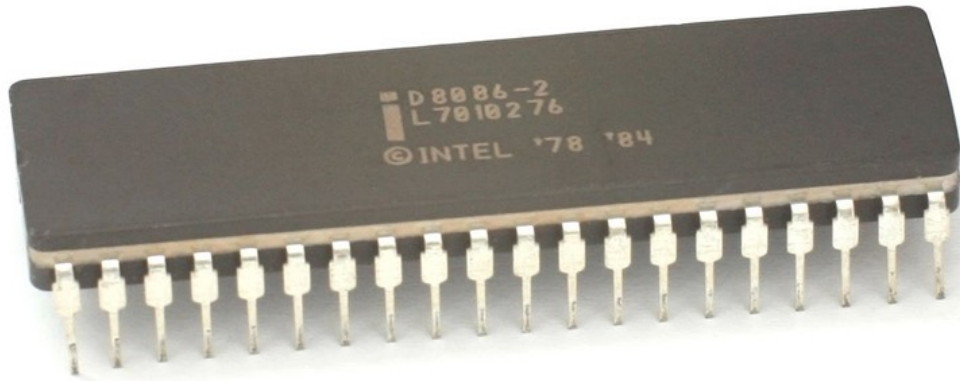
- Instructions (machine code)

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.

Central Processing Units (CPUs)

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

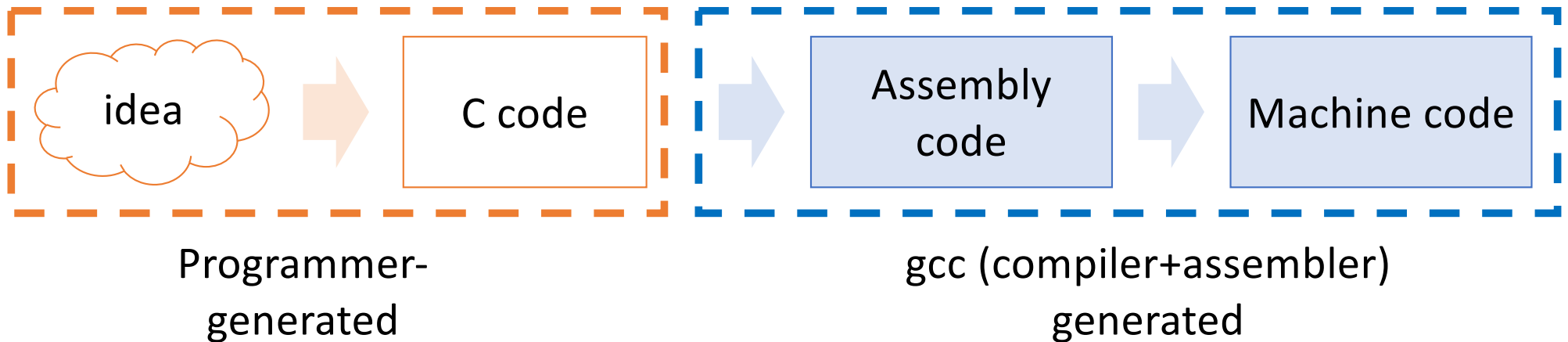


Raspberry Pi BCM2836 32-bit **ARM** microprocessor (\$35 for everything, 2015)



Intel Core i9-9900K 64-bit 8-core multi-core processor (\$449, 2018)

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 cleaner, more efficient code.

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;  
}
```

0000000000401136 <sum_array>:

```
401136:  b8 00 00 00 00  
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  
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
```



make
objdump -d sum

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>:

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

```
401136: b8 00 00 00 00    mov     $0x0,%eax
40113b: ba 00 00 00 00    mov     $0x0,%edx
401140: 4d 00 00 00 00    mov     %esi,%eax
40114f: <sum_array+0x19>
vs1q %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
```

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 19               jle    <sum_array+0x19>
401144: 48                 inc    %eax
401147: 03 03               add    (%eax),%edx
40114a: 83 03               inc    (%eax)
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: 70 50
```

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
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
401151: c3
```

```
mov     $0x0,%eax
```

This is the machine code: raw hexadecimal instructions, representing binary as read by the computer. Different instructions require a varying number of bytes.

```
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    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] identifies a register, a storage location on the CPU (e.g., eax here).

★ 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 learning how to read—though not speak—a new language!

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

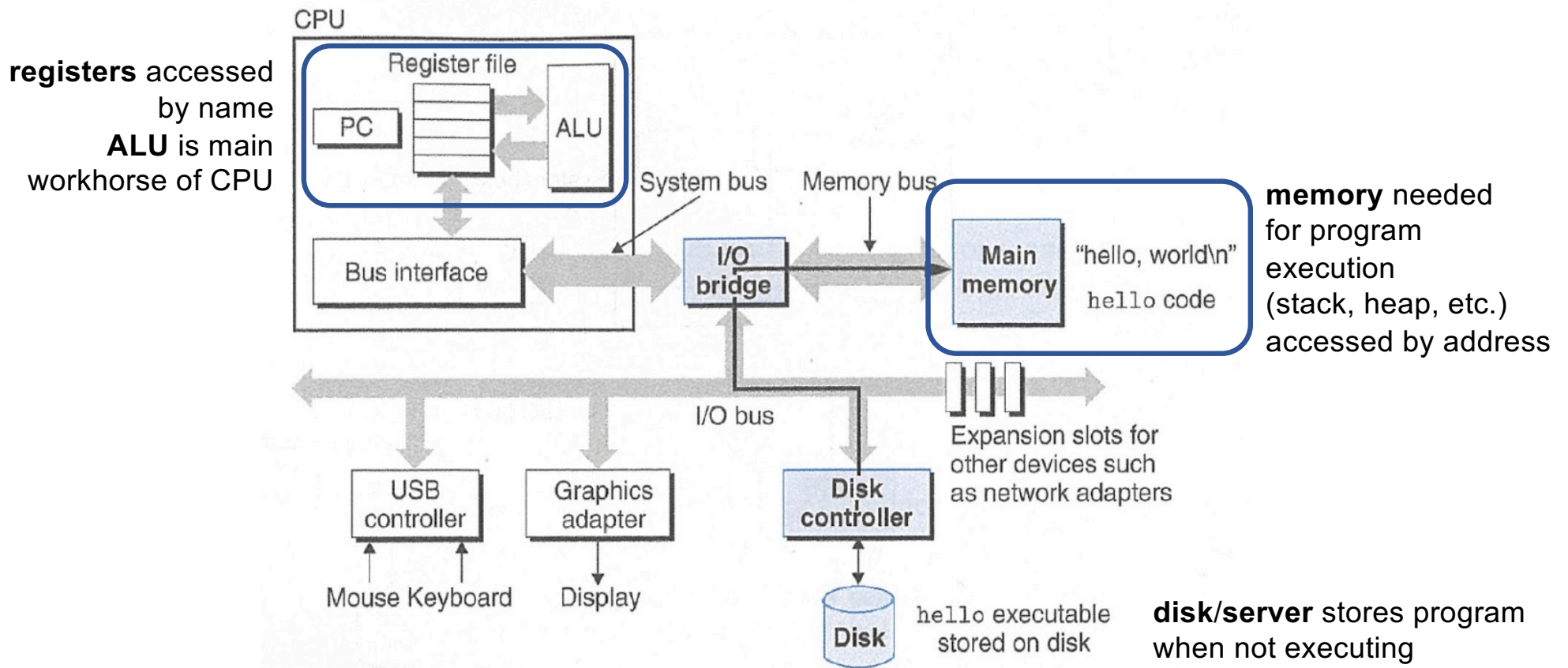
- A **register** is a 64-bit space inside the processor.
- There are 16 registers, each with a unique name.
- Registers are like "scratch paper" for the processor. Data being accessed or manipulated are first moved into registers. Most ALU operations—that is, arithmetic-logic unit operations—can only act on values stored in 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 operations 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



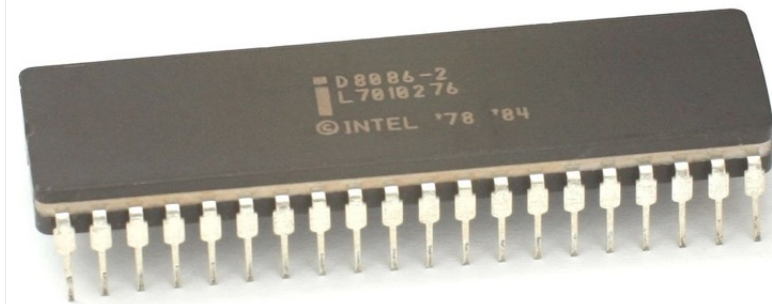
GCC And Assembly

- 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;	<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.



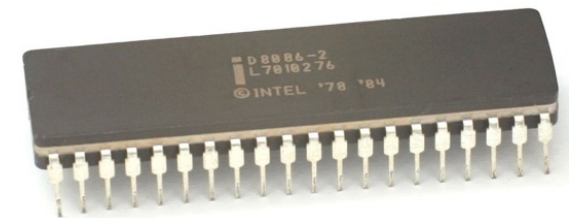
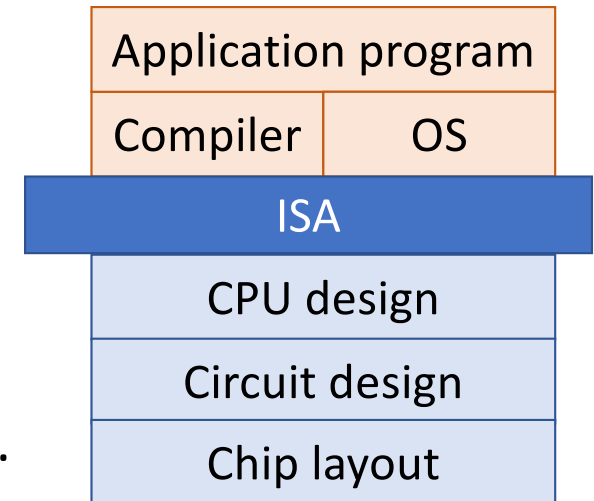
Instruction set architecture (ISA)

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).



mov

The **mov** instruction copies bytes from one place to another; it is like the assignment operator (=) in C, though the arguments are reversed.

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: 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`).³⁹

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: 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), _____

mov _____, (, %rdx, 4)

The scaling factor (e.g. 4 here) must be hardcoded to be either 1, 2, 4 or 8.

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

$0x4(\%rax, \%rdx, 2), \underline{\hspace{2cm}}$

mov

$\underline{\hspace{2cm}}, 0x4(\%rax, \%rdx, 2)$



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.” 49

Practice: 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 on our 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  
4005cb:    7c 04 8f        cmp    %esi,%edx  
4005ce:    7c 04 8f        jl    4005c2 <sum_array+0xc>  
4005d1:    9f                repz  retq
```

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



From Assembly to C

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`



From Assembly to C

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!

