Computer Systems and Programming (CS367)

Lecture 0x07

Machine Level Representation I

Intro to x86-64 Assembly

Prof. Kevin Andrea

Lecture Overview (*Chapter 3.1-3.4*)

ASM + Machine	Display a Program as Source Code, Assembly, and Machine Code	Compile C Code to Assembly using gcc	3.1
		Disssemble an ELF file to Assembly / Machine Code using objdump	
		View Disassembly using gdb	
	Describe the Data Sizes in use by x86-64 Assembly	Describe what a Word is	2.1.2
		Describe Intel's 'word' data size	3.3
		List the Intel Data Types and their Assembly Suffixes	
		List the Relationships between C primitive types and Intel Data Types	
	Access Information using x86-64 Assembly Operations	Describe the Registers Available in x86-64 Assembly	3.4
		List the 64, 32, 16, and 8-bit Register Names	
		Describe the Operand Forms of x86-64 Assembly	
		Use the mov Instruction to Load Data	
		Use the movz/movs Instructions as Appropriate	
		Use the leaq Instruction to Access an Address	(+ 3.5.1)

Why Look at Machine Level Code? (3.1)

Our objective is to learn how compilers transform C down to Machine Code

- Compilers **optimize** constructs of C in many exciting ways:
  - Eliminate Redundant Code
  - Propagate Constants to Reduce Variables and Memory Usage
  - Replace Slow Operations with Faster ones (eg. **Division** vs. **>>**)

Reverse Engineering Machine Code through Assembly is like a great Puzzle!

- Before we can solve the puzzle, we need to learn about the pieces.

- We'll study the components of the x86-64 CPU Instruction Set so we can better understand how our programs work.

Assembly Languages (3.1)

An Assembly Language is a low-level language that usually has one instruction for each CPU Operation (Direct 1:1 Mapping)

- In C, we have high-level code, with complex instructions, like **?:**
- Compilers* will generate Machine Code to execute directly on the CPU
  - One Instruction in a High Level Language translates to Many on the CPU

Assembly Language is a human-readable version of Machine Instructions

- We can look at the compiled Machine Code in Assembly Language
  - We can see what any compiled program will execute on the computer.
- Each CPU Architecture has its own Assembly Language.

The Assembly here is used by Intel/AMD and the is AT&T Syntax of x86-64

x86-64 CPU Instruction Set (3.1)

This line of CPU Instructions is now 40 years old...

- Intel 8086 was created in 1978
  - This was followed by the 80286, i386, i486, Pentium, Core 2, Core i7
- The current Instruction Set is **x86-64** (debuted with the Pentium 4E)
  - * Actually Designed by AMD as the AMD64.
  - Both AMD and Intel use the x86-64 Instruction Sets Now.

This is the hardcopy of the 2100 page Programmer's Manual for x86-64.

(Don't worry, we're skipping the boring bits)

https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf

Some Definitions (3.1)

Architecture (ISA: Instruction Set Architecture)

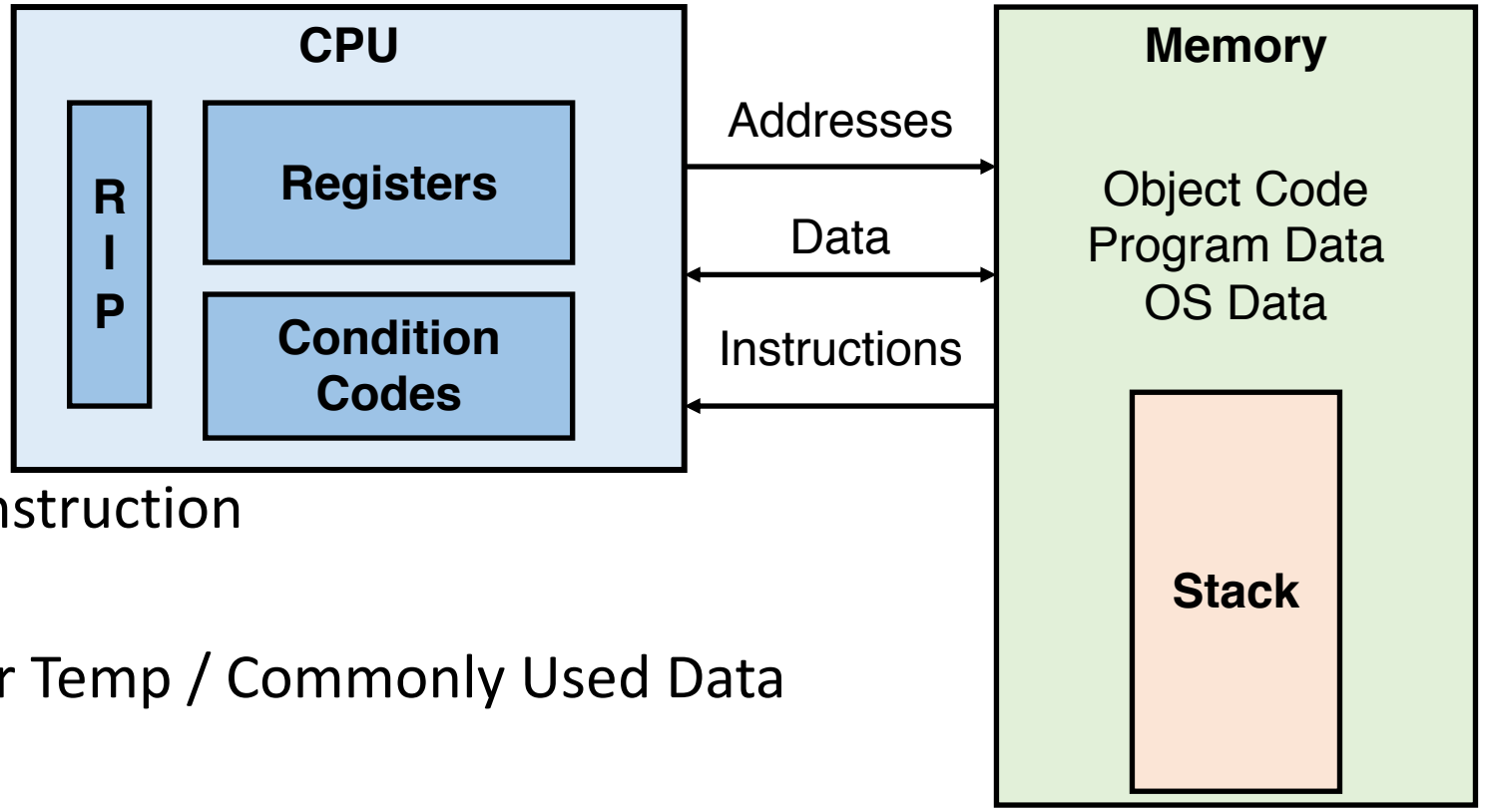
- The model for the Processor Design that we need to understand to read or write assembly/machine code.
- This includes the Design of Instruction Sets and Registers

Code Forms

- Machine Code: Byte-Level Programs that the CPU Executes Natively
- Assembly Code: A Text Representation of Machine Code

Example ISAs include x86, x86-64, Itanium, MIPS, ARM

Assembly Programmer's View (3.1)



Important Components

Programmer-Visible

RIP (Program Counter)

Address of the Next Instruction

Registers

Hardware Memory for Temp / Commonly Used Data

Condition Codes

Status about the Most Recent Arithmetic Operation (eg. Carry and Overflow Flags)

Memory

Byte Addressable Array that contains Program Code, Data, Stack and Heap

hello.* (.c, .i, .s, .o)

*Compiling (C -> Object Code) (3.2)

A Compiler Program usually Provides the Following Operations

Let's look at hello.c as an Example

Preprocessing: Remove comments, add Headers, replace Macros, ...

```
gcc -E hello.c -o hello.i
```

Compiling: Translating High-Level Code to Assembly

```
gcc -S hello.i -o hello.s
```

Assembling: Translate Assembly to Machine Code

```
gcc -c hello.s -o hello.o
```

Linking: Combine Objects (and Libraries) into an Executable

```
gcc hello.o -o hello
```

meaning.s

A Complete Assembly Program (3.2)

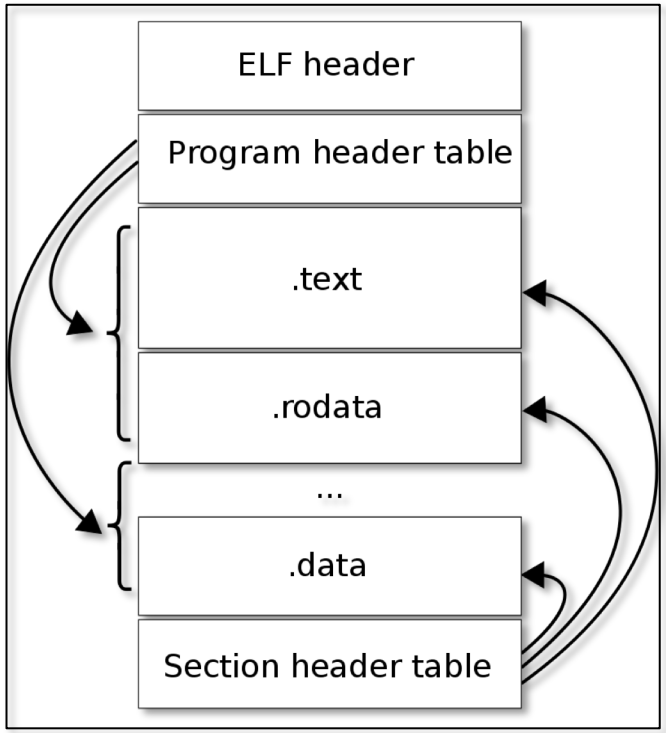
In this class, we're learning to Interpret Assembly

- We will write Assembly Code, but not usually full Programs

```
# These two lines have to appear at the top of a Program
.text           # The E.L.F. Segment for Program Code
.globl main     # Declare main as a global Symbol

main:                   # Label for main (Address of main in Memory)
  movq $0x2a, %rax      # Move 0x2a (42) into Register RAX
  ret                   # Return (return value is whatever is in RAX)
```

```
kandrea@zeus-2$ gcc -o meaning meaning.s
kandrea@zeus-2$ ./meaning
kandrea@zeus-2$ echo $?
42
```



meaning_c.c

C to Assembly (Compiling) (3.2)

Let's take one step back and look at that program in C

```c
int main() {
  return 0x2a;
}
```

To Compile this down to Assembly, use the –S flag with gcc

- I'm also using the –O flag to optimize for output a little.

```
kandrea@zeus-2$ gcc –O –S –o meaning_c.s meaning_c.c
```

```
.file "meaning_c.c"
.text
.globl main
.type main, @function
main:
.LFB0:
.cfi_startproc
movl $42, %eax
ret
.cfi_endproc
.LFE0:
.size main, .-main
.ident "GCC: (GNU) 6.3.1 20170216 (Red Hat 6.3.1-3)"
.section .note.GNU-stack,"",@progbits
```

Let's compare that to Handcoded Assembly

```
# These two lines have to appear at the top of a Program
.text                       # The E.L.F. Segment for Program Code
.globl main  # Declare main as a global Symbol

main:                       # Label for main (Address of main in Memory)
  movq $0x2a, %rax          # Move 0x2a (42) into Register RAX
  ret                       # Return (return value is whatever is in RAX)
```

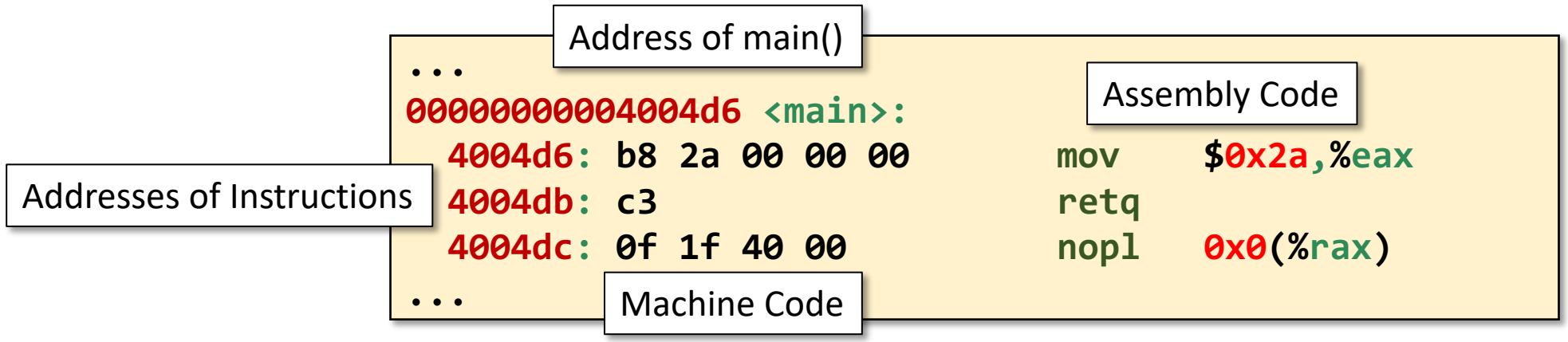
Disassembling Object Code (3.2)

Disassembler: objdump

- A disassembler is a program to translate Machine Code to Assembly
  - Can be run on any Executable (ELF file) or Object Code (.o file)
  - Use the –d option to disassemble the Object

Let's run objdump on an executable generated from our C code here

```
kandrea@zeus-2$ objdump –d meaning_c
```



Disassembly via GDB (3.2)

We can also use the disassemble command in GDB to view the Assembly

```
kandrea@zeus-2$ gdb meaning_c
...
Reading symbols from meaning_c...(no debugging symbols found)...done.
(gdb) b main
Breakpoint 1 at 0x4004d6
(gdb) run
...
Breakpoint 1, 0x00000000004004d6 in main ()
(gdb) disassemble
Dump of assembler code for function main:
=> 0x00000000004004d6 <+0>: mov    $0x2a,%eax
   0x00000000004004db <+5>: retq
End of assembler dump.
(gdb)
```

A Complete Assembly Program (3.2)

Today, we're going to examine this single line of Assembly

movX: Move (Copy) Data from a Source to a Destination.
`mov source, destination`

movq: q – Quadword (64-bit Value)

```
movq $0x2a, %rax        # Move 0x2a (42) into Register RAX
```

#X: # is for comments

$**0x2a**: Hex Value

%**rax**: RAX is used for Return Values

$0x2a: $ indicates an Immediate (constant)

%rax: % for Registers

Today, We'll Examine

- Data Formats
- Registers
- Operands
  - Forms
- Movement
  - movX
- Arithmetic

```
movq $0x2a, %rax      # Move 0x2a (42) into Register RAX
```

Data Formats (3.3)

In Assembly, we just have Data. No types, just data of different sizes.

- Originally, the 8086 (16-bit processor) has a 16-bit Word Size
  - So, Intel called a 16-bit Data Size a '**Word**'
  - Since then, they've kept this definition and use **Double Word** for 32 bits, and **Quad Word** for 64 bit data sizes.

C Type	Intel Data Type	Assembly *Suffix*	Size (Bytes)
`char`	Byte	`b`	1
`short`	Word	`w`	2
`int`	Double Word	`l`	4
`long`	Quad Word	`q`	8
`char *`	Quad Word	`q`	8
`float`	Single Precision	`s`	4
`double`	Double Precision	`l`	8

Assembly Operations use **Suffixes** to indicate what Size of data they operate on.

movX (Copies from src to dst)
movb (Copies 8 bits)
movw (Copies 16 bits)
movl (Copies 32 bits)
movq (Copies 64 bits)
mov *Num bits based on dest*

```
movq $0x2a, %rax     # Move 0x2a (42) into Register RAX
```

Registers

(3.4)

In x86-64, we have 16 General Registers

- Used for temp storage
- Used for common data
- Used for arguments
- Used for return values

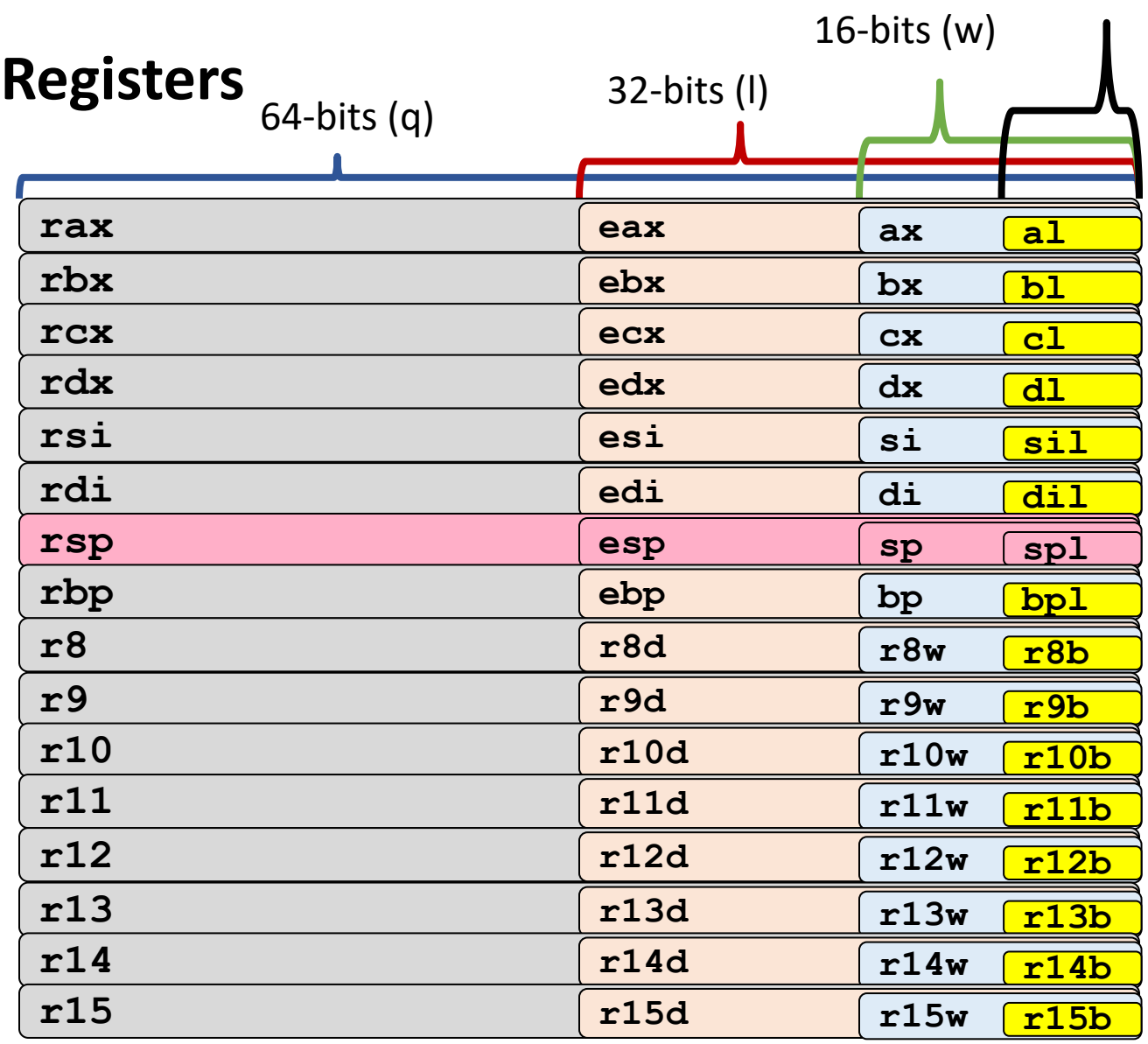
RAX – Return Value

RSP – Stack Pointer

RBP – Stack Base Pointer

Special Registers:

RIP – Program Counter



```
movq $0x2a, %rax      # Move 0x2a (42) into Register RAX
```

Operand Specifiers (3.4.1)

All operations in Assembly use suffixes and have zero or more Operands

The General Operand Format is:

`operation` **`source, dest`** (eg. addq %rax, %rbx)

`retq`	Zero Operands	
`incq %rax`	One Operand	(Dest = Dest + 1)
`movq $0x2a, %rax`	Two Operands	(Dest = Source)

***WARNING**: This is the **AT&T Syntax**. Most online guides are **BACKWARDS**!

The Intel Syntax is more common for programmers and online guides.

```
movq $0x2a, %rax      # Move 0x2a (42) into Register RAX
```

Operand Forms (3.4.1)

Just like in C, there are many different forms Operands can have.

Before we look at the forms, let's look at some Terms in the Book's Notation

Term	Notation	Meaning
Register	$\%r_x$	A Hardware Register
Register Value	$R[\%r_x]$	The value in Register r_x
Immediate Value	`$Imm`	Imm is an Immediate (Hardcoded Constant)
Address	`Imm`	Treat Imm as an Address in Memory
Dereference	`M[X]`	Memory Lookup of Address in X (Reg/Addr)

Operand Forms (3.4.1)

We're going to walk through examples of each type (C and Assembly)

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`$_i$`] * s]`	Scaled Indexed
Memory	`Imm(, %r`$_i$`, s)`	`M[Imm + R[r`$_i$`] * s]`	Scaled Indexed
Memory	`(%r`$_b$`, %r`$_i$`, s)`	`M[R[r`$_b$`] + R[r`$_i$`] * s]`	Scaled Indexed
Memory	`Imm(%r`$_b$`, %r`$_i$`, s)`	`M[Imm + R[r`$_b$`] + R[r`$_i$`] * s]`	Scaled Indexed

Remember, these forms represent a *single* operand (either Source or Dest)

Operand Forms (3.4.1)

We're going to walk through examples of each type (C and Assembly)

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] * s]`	Scaled Indexed
Memory	`Imm(, %r_i, s)`	`M[Imm + R[r_i] * s]`	Scaled Indexed
Memory	`(%r_b, %r_i, s)`	`M[R[r_b] + R[r_i] * s]`	Scaled Indexed
Memory	`Imm(%r_b, %r_i, s)`	`M[Imm + R[r_b] + R[r_i] * s]`	Scaled Indexed

This set deals with accessing Values Directly

Immediate Operands (3.4.1)

Immediate	$Imm	Imm	Immediate

An Operand in this form is just a numerical constant.

The Operand form is: **$Imm**

This is equivalent in C to a hardcoded value.

Immediate Values can only be the **Source**, never a Destination

Example:

Assembly: `movq $42, %rax`

English: *Copy the immediate value 42 into RAX*

C Analog: *a = 42;*

Register Operands (3.4.1)

Register	$\%r_a$	$R[r_a]$	Register

An operand in this form is the Value contained in that Register.

The Operand form is: $\%r_a$

This just refers to any arbitrary register.

Example:

Assembly: `movq $42, %rax`

English: *Copy the immediate value 42 into RAX*

Assembly: `movq %rbx, %rax`

English: *Copy the value contained in RBX into RAX*

Operand Forms (3.4.1)

We're going to walk through examples of each type (C and Assembly)

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] * s]`	Scaled Indexed
Memory	`Imm(, %r_i, s)`	`M[Imm + R[r_i] * s]`	Scaled Indexed
Memory	`(%r_b, %r_i, s)`	`M[R[r_b] + R[r_i] * s]`	Scaled Indexed
Memory	`Imm(%r_b, %r_i, s)`	`M[Imm + R[r_b] + R[r_i] * s]`	Scaled Indexed

This set deals with Dereferencing a Single Register

Dereferencing Operands (3.4.1)

Memory	Imm	M[Imm]	Absolute

An Operand in this form is just a hardcoded Address.

The Operand form is: **Imm**

Note that there are no $ in this format!

Imm in this format is always treated as an Address Dereference!

Example:

Assembly: `movq 0x400496, %rax`

English: *Copy 64-bits of data from Memory, starting at address 0x400496 and write them into RAX*

C Analog: `a = *((int *)0x400496);`

Dereferencing Operands (3.4.1)

Memory	(%r_a)	M[R[r_a]]	Indirect

An Operand in this form is for dereferencing an address stored in a Register

The Operand form is: **(%r_a)**

This is when r_a's **value** is an Address and you want to Dereference it.

Example:

Assembly: `movq (%rcx), %rax`

English: *Copy 64-bits of data from Memory, starting at the address in RCX and write them into RAX*

C Analog: `a = *c;`

Dereferencing Operands (3.4.1)

Memory	Imm(%r_b)	M[Imm + R[r_b]]	Base + Displacement

An Operand in this form is for dereferencing using Pointer Arithmetic

The Operand form is: **Imm(%r_a)**

This lets you add an Offset to an Address before Dereferencing.

In Assembly, all arithmetic is done in ***BYTES***.

Example:

Assembly: `movq 8(%rcx), %rax`

English: *Copy 64-bits of data from Memory, starting at the address in RCX + 8 bytes, then write it into RAX*

C Analog: `a = *(c + 8); // c + 8 bytes!`

Operand Forms (3.4.1)

We're going to walk through examples of each type (C and Assembly)

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`$_i$`] * s]`	Scaled Indexed
Memory	`Imm(, %r`$_i$`, s)`	`M[Imm + R[r`$_i$`] * s]`	Scaled Indexed
Memory	`(%r`$_b$`, %r`$_i$`, s)`	`M[R[r`$_b$`] + R[r`$_i$`] * s]`	Scaled Indexed
Memory	`Imm(%r`$_b$`, %r`$_i$`, s)`	`M[Imm + R[r`$_b$`] + R[r`$_i$`] * s]`	Scaled Indexed

This set deals with Dereferencing a Two Registers

Dereferencing Operands (3.4.1)

Memory	(r_b, r_i)	M[R[r_b] + R[r_i]]	Indexed

An Operand in this form is for dereferencing an address stored in a Register

The Operand form is: **(r_b, r_i)**

This adds the values of r_b and r_i before dereferencing.

Example:

Assembly: `movq (%rcx,%rdx), %rax`

English: *Copy 64-bits of data from Memory, starting at the address equal to RCX+RDX , then write into RAX*

C Analog: `a = *(c+d); // In Bytes`

Dereferencing Operands (3.4.1)

Memory	Imm(r_b, r_i)	M[Imm + R[r_b] + R[r_i]]	Indexed

An Operand in this form is for dereferencing an address stored in a Register

The Operand form is: **Imm(r_b, r_i)**

Adds the values of an Imm, r_b, and r_i together before dereferencing.

Example:

Assembly: `movq 8(%rcx,%rdx), %rax`

English: *Copy 64-bits of data from Memory, starting at address equal to 8+RCX+RDX , then write into RAX*

C Analog: `a = *(8+c+d); // In Bytes`

Operand Forms (3.4.1)

We're going to walk through examples of each type (C and Assembly)

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`$_i$`] * s]`	Scaled Indexed
Memory	`Imm(, %r`$_i$`, s)`	`M[Imm + R[r`$_i$`] * s]`	Scaled Indexed
Memory	`(%r`$_b$`, %r`$_i$`, s)`	`M[R[r`$_b$`] + R[r`$_i$`] * s]`	Scaled Indexed
Memory	`Imm(%r`$_b$`, %r`$_i$`, s)`	`M[Imm + R[r`$_b$`] + R[r`$_i$`] * s]`	Scaled Indexed

These are like the previous, but with a Scaling Factor (s)

Dereferencing Operands (3.4.1)

Memory	(, %r_i, s)	M[R[r_i] * s]	Scaled Indexed

An Operand in this form is for dereferencing an address stored in a Register

The Operand form is: **(, %r_i, s)**

This multiplies the value in r_i by s before dereferencing it.

S can only be **1, 2, 4, or 8**

Example:

Assembly: `movq (,%rdx,4), %rax`

English: *Copy 64-bits of data from Memory, starting at the address equal to 4*RDX , then write into RAX*

C Analog: `a = *(4*d); // In Bytes`

Dereferencing Operands (3.4.1)

Memory	Imm(, %r_i, s)	M[Imm + R[r_i] * s]	Scaled Indexed

An Operand in this form is for dereferencing an address stored in a Register

The Operand form is: **Imm(, %r_i, s)**

This multiplies the value in r_i by s, then adds Imm before dereferencing

S can only be **1, 2, 4, or 8**

Example:

Assembly: `movq 8(,%rdx,4), %rax`

English: *Copy 64-bits of data from Memory, starting at the address equal to 8 + 4*RDX , then write into RAX*

C Analog: `a = *(8+(4*d)); // In Bytes`

Dereferencing Operands (3.4.1)

Memory	(%r_b, %r_i, s)	M[R[r_b] + R[r_i] * s]	Scaled Indexed

An Operand in this form is for dereferencing an address stored in a Register

The Operand form is: **(%r_b, %r_i, s)**

This multiplies the value in r_i by s, then adds r_b before dereferencing

S can only be **1, 2, 4, or 8**

Example:

Assembly: `movq (%rbx,%rdx,4), %rax`

English: *Copy 64-bits of data from Memory, starting at the address equal to RBX + 4 * RDX , then write the fetched bits into RAX as its value.*

C Analog: `a = *(b+(4*d)); // In Bytes`

Dereferencing Operands (3.4.1)

Memory	Imm(%r_b, %r_i, s)	M[Imm + R[r_b] + R[r_i] * s]	Scaled Indexed

An Operand in this form is for dereferencing an address stored in a Register

The Operand form is: **Imm(%r_b, %r_i, s)**

Multiplies the value in r_i by s, then adds r_b and Imm before dereferencing

S can only be **1, 2, 4, or 8**

Example:

Assembly: `movq 2(%rbx,%rdx,4), %rax`

English: *Copy 64-bits of data from Memory, starting at the address equal to 2 + RBX + 4 * RDX , then write the fetched bits into RAX as its value.*

C Analog: `a = *(2+b+(4*d)); // In Bytes`

Operand Form Patterns (3.4.1)

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`$_i$`] * s]`	Scaled Indexed
Memory	`Imm(, %r`$_i$`, s)`	`M[Imm + R[r`$_i$`] * s]`	Scaled Indexed
Memory	`(%r`$_b$`, %r`$_i$`, s)`	`M[R[r`$_b$`] + R[r`$_i$`] * s]`	Scaled Indexed
Memory	`Imm(%r`$_b$`, %r`$_i$`, s)`	`M[Imm + R[r`$_b$`] + R[r`$_i$`] * s]`	Scaled Indexed

Operand Form Patterns (3.4.1)

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`$_i$`] * s]`	Scaled Indexed
Memory	`Imm(, %r`$_i$`, s)`	`M[Imm + R[r`$_i$`] * s]`	Scaled Indexed
Memory	`(%r`$_b$`, %r`$_i$`, s)`	`M[R[r`$_b$`] + R[r`$_i$`] * s]`	Scaled Indexed
Memory	`Imm(%r`$_b$`, %r`$_i$`, s)`	`M[Imm + R[r`$_b$`] + R[r`$_i$`] * s]`	Scaled Indexed

These are just Values, no Dereferencing

Operand Form Patterns (3.4.1)

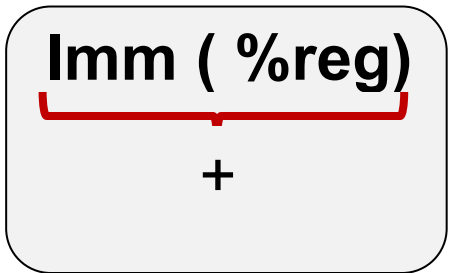
Type	Form	Operand Value	Name
Immediate	`$Imm`	`Imm`	Immediate
Register	`%r`$_a$	`R[r`$_a$`]`	Register
Memory	`Imm`	`M[Imm]`	Absolute
Memory	`(%r`$_a$`)`	`M[R[r`$_a$`]]`	Indirect
Memory	`Imm(%r`$_b$`)`	`M[Imm + R[r`$_b$`]]`	Base + Displacement
Memory	`(%r`$_b$`, %r`$_i$`)`	`M[R[r`$_b$`] + R[r`$_i$`]]`	Indexed
Memory	`Imm(%r`$_b$`, %r`$_i$`)`	`M[Imm + R[r`$_b$`] + R[r`$_i$`]]`	Indexed
Memory	`(, %r`$_i$`, s)`	`M[R[r`$_i$`] * s]`	Scaled Indexed
Memory	`Imm(, %r`$_i$`, s)`	`M[Imm + R[r`$_i$`] * s]`	Scaled Indexed
Memory	`(%r`$_b$`, %r`$_i$`, s)`	`M[R[r`$_b$`] + R[r`$_i$`] * s]`	Scaled Indexed
Memory	`Imm(%r`$_b$`, %r`$_i$`, s)`	`M[Imm + R[r`$_b$`] + R[r`$_i$`] * s]`	Scaled Indexed

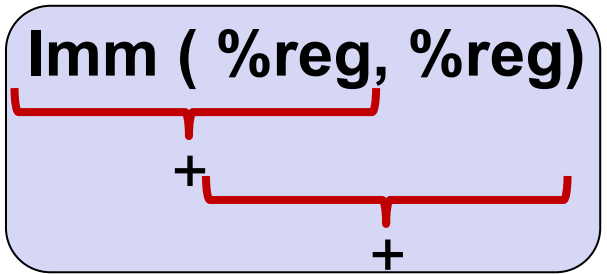
Imm (%reg)

+

Operand Form Patterns (3.4.1)

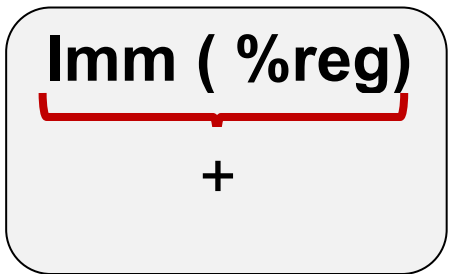
Type	Form	Operand Value	Name
Immediate	`$Imm`	`Imm`	Immediate
Register	`%r_a`	`R[r_a]`	Register
Memory	`Imm`	`M[Imm]`	Absolute
Memory	`(%r_a)`	`M[R[r_a]]`	Indirect
Memory	`Imm(%r_b)`	`M[Imm + R[r_b]]`	Base + Displacement
Memory	`(%r_b, %r_i)`	`M[R[r_b] + R[r_i]]`	Indexed
Memory	`Imm(%r_b, %r_i)`	`M[Imm + R[r_b] + R[r_i]]`	Indexed
Memory	`(, %r_i, s)`	`M[R[r_i] * s]`	Scaled Indexed
Memory	`Imm(, %r_i, s)`	`M[Imm + R[r_i] * s]`	Scaled Indexed
Memory	`(%r_b, %r_i, s)`	`M[R[r_b] + R[r_i] * s]`	Scaled Indexed
Memory	`Imm(%r_b, %r_i, s)`	`M[Imm + R[r_b] + R[r_i] * s]`	Scaled Indexed

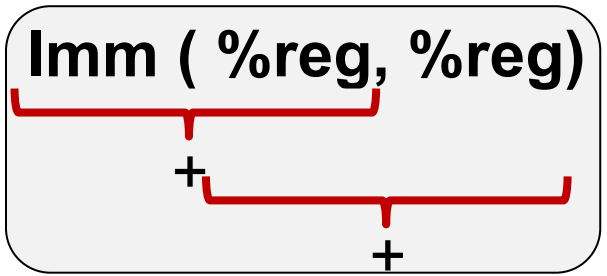


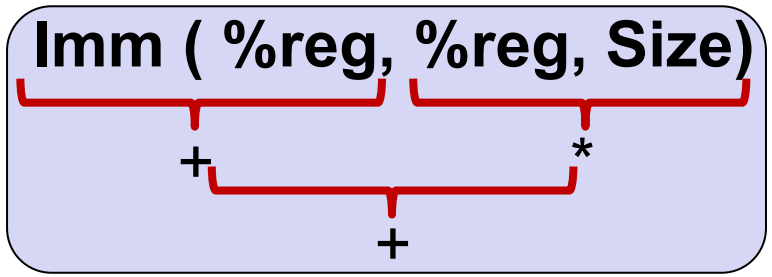


Operand Form Patterns (3.4.1)

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`$_i$`] * s]`	Scaled Indexed
Memory	`Imm(, %r`$_i$`, s)`	`M[Imm + R[r`$_i$`] * s]`	Scaled Indexed
Memory	`(%r`$_b$`, %r`$_i$`, s)`	`M[R[r`$_b$`] + R[r`$_i$`] * s]`	Scaled Indexed
Memory	`Imm(%r`$_b$`, %r`$_i$`, s)`	`M[Imm + R[r`$_b$`] + R[r`$_i$`] * s]`	Scaled Indexed







Operand Forms

These forms can be either Source or Destination

- If Source, generally they are used to read in a Source Value
- If Destination, they are used to calculate the Address of where to Write the Output to
- Arithmetic will read both Source and Dest, then write the result to Dest

Complicated Example:

```
movq %rax, (%rdx, %rcx, 8)
```

1) Read a 64-bit **Value** from RAX
2) Multiply 8 by RCX's Value, then add RDX to it to generate an **Address**
3) Store the 64-bit **Value** read from RAX into the generated **Address**.

Operand Forms (Lightning Round) (3.4.1)

Given the following, how would each Operand Evaluate?

All values in Memory are in Hex

1) `%rdx`	`0x02`
2) `$0x110`	`0x110`
3) `0x110`	`0xdef`
4) `(%rax)`	`0x123`
5) `8(%rbx)`	`0x1701`
6) `8(%rbx,%rcx)`	`0xdef`
7) `0x108(,%rdx,8)`	`0xabc`
8) `0x10(%rbx,%rdx,8)`	`0x123`
9) `%rax`	`0x120`

Registers

`%rax`	`0x120`
`%rbx`	`0x100`
`%rcx`	`0x08`
`%rdx`	`0x02`

Memory

	Address
123	0x120
abc	0x118
def	0x110
1701	0x108
456	0x100

Homework

Before your next class....

Reading:

1. **Review Chapters 3.1-3.5 and Start Reading Chapter 3.6**

   The next section introduces Control Flow (Conditionals and Loops!)

2. **Continue to Review Chapter 2 – A Wandering Midterm Approaches**

   For Chapter 2, make sure you are comfortable with:

   - Integer Encoding (Signed and Unsigned)
     - Boolean Operations and Shifting in C for Signed and Unsigned Integers
     - Integer Arithmetic
   - Floating Point Encoding (Any given Bit Widths)
     - FP Encoding -> Decimal
     - Decimal -> FP Encoding
     - Floating Point Arithmetic (Multiplication and Addition) and Rounding