CS107 Lecture 16
Assembly: Arithmetic and Logic Operations

Reading: B&O 3.5-3.6

Data Sizes: Not Everything is Eight Bytes

Data sizes in assembly use **slightly different names**.

- A **byte** is 1 byte.
- A **word** is 2 bytes.
- A **double word** is 4 bytes.
- A **quad word** is 8 bytes.

On early x86 processors, a "word" referred to the **natural register size** of 16 bits. 32-bit and 64-bit architectures **retained the association**—**word** and **two bytes** became synonymous—and introduced "double word" and "quad word" as **extensions** to the word.

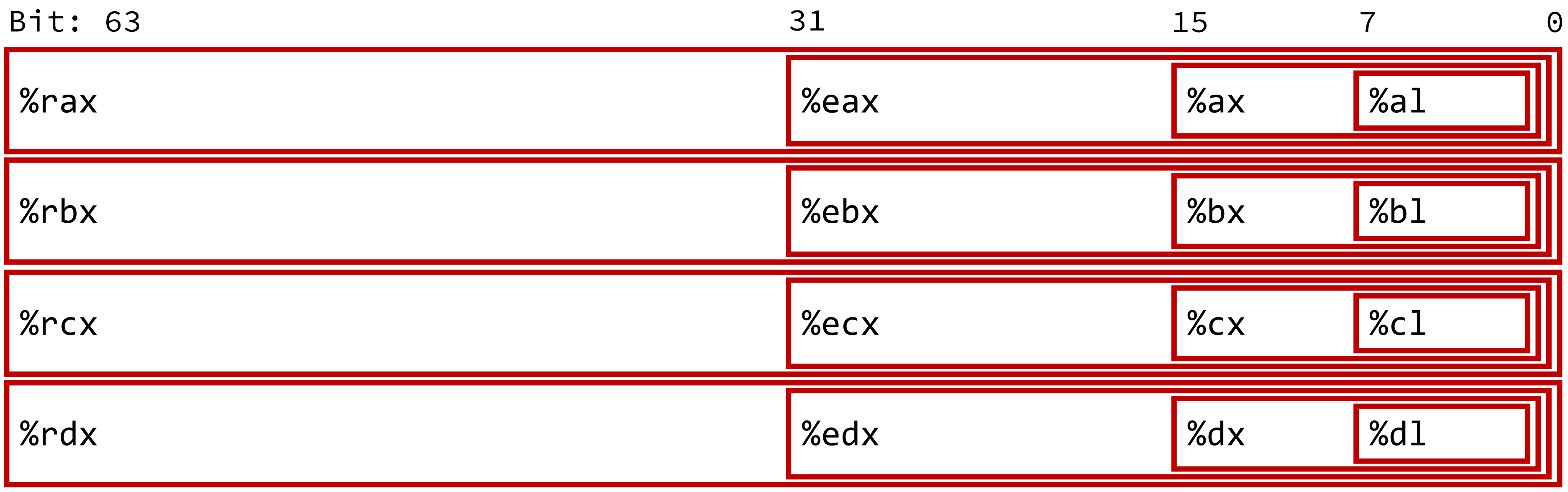
Assembly instructions—**most often `mov`, but others as well**—use suffixes to specify the size of the data being moved or manipulated.

- **`b`** means **byte**
- **`w`** means **word**
- **`l`** means **double word**
- **`q`** means **quad word**

Examples:

```
movb   $0x41, %al
subb   $1, (%rax)
xorw   %dx, %dx
leaq   (%rdi,%rsi,8), %rax
pushq  %rbp
```

Registers and Subregisters



- The 64-bit registers use an **r-prefix**: **`%rax`**, **`%rbx`**, **`%rcx`**, **`%rdx`**, and so on.
- The lower portions reuse older names: the low 32 bits use an **e-prefix** (**`%eax`**), the low 16 bits use the original names (**`%ax`**), and the low 8 bits use an **l-suffix** (**`%al`**).
- **The same subdivision applies to the other general-purpose registers** (**`%rsi`**, **`%rdi`**, **`%rbp`**, **`%rsp`**, **`%r8–%r15`**): each has 64-, 32-, 16-, and 8-bit names that refer to the **same underlying register**.
- These aren't separate registers, but different views into the **same 64-bit register**.

Taking Prefix and Suffixes to Heart

`mov` can take an **optional suffix** (**`b`**/**`w`**/**`l`**/**`q`**) to specify the size of the move: **`movb`**, **`movw`**, **`movl`**, **`movq`**

- The **optional suffix can be omitted** if the size of the data transfer is implied by one or both arguments, e.g., **`mov $0x0, %al`** or **`mov %ebx, %edx`**
- **`mov`** only updates the specific register bytes or memory locations indicated.
  - **Exception**: Writing to a 32-bit register (e.g., **`movl %ebx,%eax`**) clears the upper 32 bits of the corresponding 64-bit register (e.g., **`%rax`**)

Examples:

```
movb $0xff, %bl
movl (%rsp,%rdx,4), %edx
movq (%rdx), %rax
```

The above **mov** instruction could have omitted the suffixes, as the **destination implies data size**.

```
movb $0x1, (%rcx)
movq $5, 8(%rsp)
movw $0x44, (%rax, %rbp, 4)
```

The suffix is **absolutely required** in situations where **immediates are written out to memory**, since memory operands have no size.

Big From Small: movz and movs

Two **mov** instructions are generally used **to copy from a smaller source to a larger destination**: **movz** and **movs**

- **movz** fills the remaining bytes with zeros

Examples:

```
movzbl   %al, %eax        # zero extend,  8 -> 32
movzwq   (%rdi), %rax     # zero extend, 16 -> 64
```

- **movs** fills the remaining bytes by sign-extending the most significant bit of the source

Examples:

```
movsbl   %al, %eax.       # sign extend,  8 -> 32
movzwq   (%rdi), %rax     # sign extend, 16 -> 64
```

  - And yes, these mnemonics encode both the source and destination sizes, e.g., **movzbl** = byte-to-long, **movswq** = word-to-quad

- The source must be memory or a register, and destination must be a register

Exhaustive List: movz

```
movz S,D          D ← ZeroExtend(S)
```

Instruction	Description
`movzbw`	Move zero-extended byte to word
`movzbl`	Move zero-extended byte to double word
`movzwl`	Move zero-extended word to double word
`movzbq`	Move zero-extended byte to quad word
`movzwq`	Move zero-extended word to quad word

Exhaustive List: movs

`movs S,D`   `D ← SignExtend(S)`

Instruction	**Description**
`movsbw`	Move sign-extended byte to word
`movsbl`	Move sign-extended byte to double word
`movswl`	Move sign-extended word to double word
`movsbq`	Move sign-extended byte to quad word
`movswq`	Move sign-extended word to quad word
`movslq`	Move sign-extended double word to quad word
`cltq`	Sign-extend **`%eax`** in place to fill **`%rax`**

Why the dedicated, in-place sign extender? 🤓

Well, signed division—as we'll see shortly—requires the dividend be **sign-extended to the full register width**. **`cltq`** provides a **compact, lickety-split** way to **extend** the 32-bit value in **`%eax`** to **occupy the full 64 bits** of **`%rax`**. Compact? How so? **`cltq`** is encoded in a ❤️ **single byte** ❤️ —0x98.

Baby’s First Assembly

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

More of our first **`objdump`** makes sense now.

- We see that a **sign extended copy** of **`i`**—backed by **`%eax`**—is written to **`%rcx`**.
    - Why do that? **Because memory operands can only be framed in terms of 64-bit registers**.
- Even though you don’t formally know about **add** yet, you see this one here **relies on one of the advanced addressing modes** to fetch **`arr[i]`**.

```
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
```

The `lea` Instruction

The **`lea`** instruction loads an effective address into a destination **without accessing memory**.

`lea src, dst`

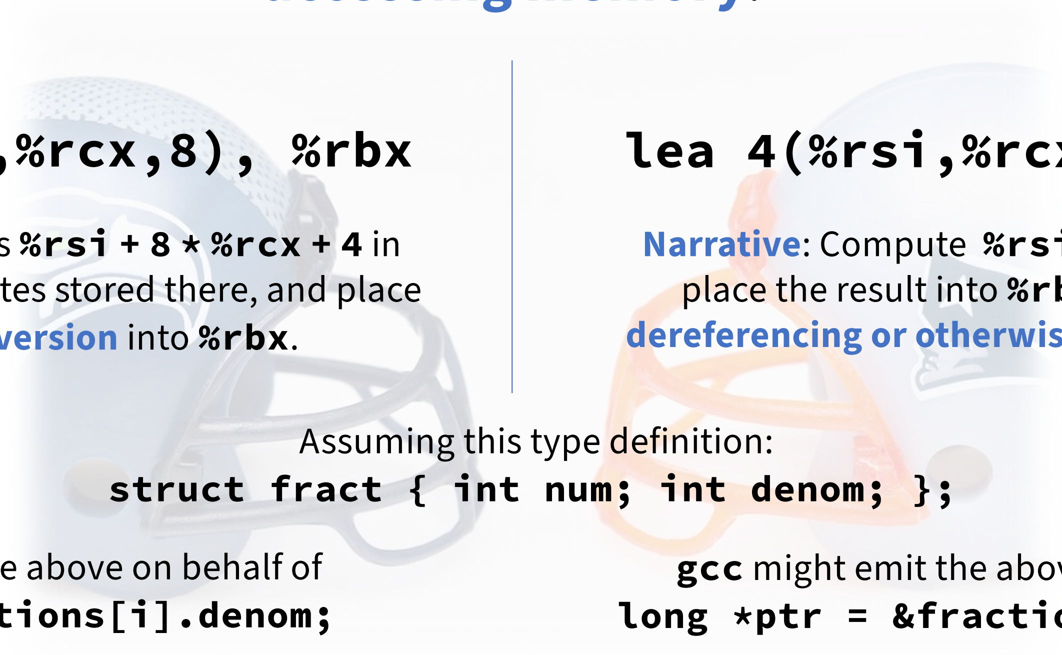
Unlike **`mov`**, which copies data from the location identified by **`src`** to **`dst`**, **`lea`** **copies the computed address described by** **`src`** to **`dst`**. **`src`** must be a memory-style address expression and **`dst`** must be a register.

Examples:

```
lea 8(%rsp), %rax              # rax = rsp + 8
lea (%rdi,%rsi), %rax          # rax = rdi + rsi
lea 16(%rdi,%rsi,4), %rax      # rax = rdi + 4 * rsi + 16
lea -0x20(%rbp,%rcx,8), %r10   # r10 = rbp + 8 * rcx - 32
```

Competing Narratives: `lea` and `mov`

The **`lea`** instruction loads an effective address into a destination **without accessing memory**.



`movslq 4(%rsi,%rcx,8), %rbx`

Narrative: Go to address **`%rsi`** + **8** * **`%rcx`** + **4** in memory, grab the four bytes stored there, and place a **sign-extended version** into **`%rbx`**.

`lea 4(%rsi,%rcx,8), %rbx`

Narrative: Compute **`%rsi`** + **8** * **`%rcx`** + **4** and place the result into **`%rbx`**. **Do so without dereferencing or otherwise accessing memory**.

Assuming this type definition:

```
struct fract { int num; int denom; };
```

`gcc` might emit the above on behalf of

```
long d = fractions[i].denom;
```

`gcc` might emit the above on behalf of

```
long *ptr = &fractions[i].denom;
```

Cool, useful Trivia: **`lea`** is often used for **elementary school arithmetic** because it computes addresses without touching memory. Because **nothing's ever dereferenced**, the "addresses" don't have to be real memory addresses.

Special Purpose Registers

Some registers take on **special responsibilities** during program execution.

- **`%rax`** typically stores the **return value**
- **`%rdi`** stores the **first** parameter passed to a function
- **`%rsi`** stores the **second** parameter passed to a function
- **`%rdx`** stores the **third** parameter passed to a function
- **`%rcx`**, **`%r8`**, and **`%r9`** store parameters **four**, **five**, and **six**
  - Additional parameters **beyond a sixth** aren't passed in registers
- **`%rip`** stores the **address of the next instruction** to execute
- **`%rsp`** points to the **top of the current stack frame**

What are you doing creating functions with **seven** or more parameters, though?

Reverse Engineering Etude

Examine the **assembly code** emitted on behalf of the **`dolores_park`** function and **reconstruct an equivalent, two-line implementation** in C.

Recall that the **first three parameters** are supplied via **`%rdi`**, **`%rsi`**, and **`%rdx`**, and the **return value** is placed in **`%rax`** just before exit.

Assembly:

```
dolores_park:
    leaq (%rsi,%rdx,2), %rax
    movq %rax, (%rdi)
    movq (%rdi,%rsi,8), %rax
    subq %rdx, %rax
    ret
```

Key Insights:

- The second line of assembly writes the contents of **`%rax`** to the address stored in **`%rdi`**.
  - **`%rdi`** stores **`arr`**, so line 1 is likely **`arr[0] = <expr>`**.
  - Since **`%rsi`** and **`%rdx`** store the values of **`x`** and **`y`**, it looks like **`lea`** is being leveraged to compute **`x + 2 * y`**.
- Line 3 places **`arr[x]`** into **`%rax`** as if it's the return value.
- Line 4, however, quickly subtracts **`y`** from **`%rax`** before returning.

C:

```
long dolores_park(long arr[], long x, long y) {
    arr[0] = x + 2 * y;
    return arr[x] - y;
}
```

Math: Unary and Binary Operators

The following instructions operate on a **single operand** (register or memory):

Instruction	Effect	Description
`inc D`	`D ← D + 1`	Increment
`dec D`	`D ← D - 1`	Decrement
`neg D`	`D ← -D`	Negate
`not D`	`D ← ~D`	Complement

Examples:

```
inc %rax                    # rax++
decq 8(%rsi)                # arr[1]--
negq (%rbx,%rcx,8)          # arr[k] *= -1
incq (%rdi)                 # (*p)++
```

These instructions operate on **two operands**. At most one of the operands can be a memory location, and the source can be an immediate. Note the **destination is also the first of the two arguments**.

Instruction	Effect	Description
`add S, D`	`D ← D + S`	Add
`sub S, D`	`D ← D - S`	Subtract
`imul S, D`	`D ← D * S`	Multiply
`xor S, D`	`D ← D ^ S`	Exclusive-or
`or S, D`	`D ← D | S`	Or
`and S, D`	`D ← D & S`	And

Examples:

```
add %rsi, %rax              # rax += rsi
subq %rax, 8(%rdi)          # p[1] -= rax
imulq $4, (%rsi,%rdx,8)     # arr[j] *= 4
xor %eax, %eax              # eax to 0
or $1, %rax                 # lsb to 1
andq $-8, %rsp              # align to 8
```

Bitwise Shift Instructions

Shift instructions have **two operands**: shift amount **k** and shiftee **D**.
k can be **an immediate** or the byte register **%cl** (and only that register)

Instruction	Effect	Description
`sal k, D`	`D ← D << k`	Left shift
`shl k, D`	`D ← D << k`	Left shift (i.e., `sal`)
`sar k, D`	$D \leftarrow D >>_A k$	Arithmetic right shift
`shr k, D`	$D \leftarrow D >>_L k$	Logical right shift

Examples:

```
shl $1, %rax       # multiply by 2
shr $2, %rbx       # unsigned divide by 4
sar $5, %rcx       # signed divide by 32
sarq $3, 8(%rdi)   # signed arr[1] /= 8
mov %rsi, %rcx     # stage shift amount
shl %cl, %rax      # rax <<= rsi
```

With **%cl**, the **width of the shiftee determines which part of %cl matters**.

For a **w**-bit shiftee, the low-order $\texttt{log}_2\texttt{w}$ bits of **`%cl`** determine the shift amount.

If **`%cl = 0xff`**, **`shlb`** shifts by 7 because the lowest $\log_2 8 = 3$ bits of **`0xff`** are **`0b111 = 7`**.

`shlw` shifts by 15 because the lowest $\log_2 16 = 4$ bits of **`0xff`** are **`0b1111 = 15`** .