CS107, Lecture 16
Assembly: Arithmetic and Logic

Reading: B&O 3.5-3.6

Ed Discussion: https://edstem.org/us/courses/46162/discussion/3748926

Data Sizes

Data types in assembly are managed via a slightly different set of names:

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

Assembly instructions can include suffixes to refer to these types:

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

Register Sizes

Bit: 63 31 15 7 0

%rax	%eax	%ax	%al
%rbx	%ebx	%bx	%bl
%rcx	%ecx	%cx	%cl
%rdx	%edx	%dx	%dl
%rsi	%esi	%si	%sil
%rdi	%edi	%di	%dil

Register Sizes

Bit: 63	31	15	7 0
%rbp	%ebp	%bp	%bpl
%rsp	%esp	%sp	%spl
%r8	%r8d	%r8w	%r8b
%r9	%r9d	%r9w	%r9b
%r10	%r10d	%r10w	%r10b
%r11	%r11d	%r11w	%r11b

Register Sizes

Bit: 63	31	15	7	0
%r12	%r12d	%r12w	%r12b	
%r13	%r13d	%r13w	%r13b	
%r14	%r14d	%r14w	%r14b	
%r15	%r15d	%r15w	%r15b	

Register Responsibilities

Some registers take on special responsibilities during program execution.

- **%rax** stores the return value
- **%rdi** stores the first parameter to a function
- **%rsi** stores the second parameter to a function
- **%rdx** stores the third parameter to a function
- **%rip** stores the address of the next instruction to be executed
- **%rsp** stores the address of the stack frame of the currently executing function

Reference Sheet: `cs107.stanford.edu/resources/x86-64-reference.pdf`
See more guides on Resources page of course website!

mov Variants

- **mov** can take an optional suffix (b/w/l/q) that specifies the size of data to move: `movb, movw, movl, movq`
- **mov** only updates the specific register bytes or memory locations indicated.
  - **Exception: movl** writing to a register will also set high order 4 bytes to 0.

Practice: mov And Data Sizes

For each of the following **mov** instructions, determine the appropriate suffix based on the operands (e.g., **movb**, **movw**, **movl** or **movq**).

```
1. mov__ %eax, (%rsp)
2. mov__ (%rax), %dx
3. mov__ $0xff, %bl
4. mov__ (%rsp,%rdx,4),%dl
5. mov__ (%rdx), %rax
6. mov__ %dx, (%rax)
```

Practice: mov And Data Sizes

For each of the following **mov** instructions, determine the appropriate suffix based on the operands (e.g., **movb**, **movw**, **movl** or **movq**).

```
1. movl %eax, (%rsp)
2. movw (%rax), %dx
3. movb $0xff, %bl
4. movb (%rsp,%rdx,4),%dl
5. movq (%rdx), %rax
6. movw %dx, (%rax)
```

mov

- The **movabsq** instruction is used to write a 64-bit immediate (constant) value.
- The regular **movq** instruction can only take 32-bit immediates.
- 64-bit immediate as source, only register as destination.

movabsq $0x0011223344556677, %rax

movz and movs

- There are two **mov** instructions that can be used to copy a smaller source to a larger destination: **movz** and **movs**.
- **movz** fills the remaining bytes with zeros
- **movs** fills the remaining bytes by sign-extending the most significant bit of the source.
- The source must be from memory or a register, and the destination must be a register.

movz and movs

```
MOVZ S,R            R ← 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

movz and movs

```
MOVS S,R          R ← 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 all of %rax `%rax <- SignExtend(%eax)`

Register Sizes

- The operand forms with parentheses (e.g., **mov (%rax), %rdi**) require that registers in parentheses be the 64-bit registers.
- For that reason, you may see smaller registers extended with e.g., **movs** into the larger registers before these kinds of instructions.

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          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
```

lea

The **lea** instruction copies an "effective address" from one place to another.

```
lea        src,dst
```

Unlike **mov**, which copies data at the address src to the destination, **lea** copies the value of src *itself* to the destination.

The syntax for the destinations is the same as **mov**. The difference is how it handles the src.

lea vs. mov

Operands	mov Interpretation	lea Interpretation
`6(%rax), %rdx`	Go to the address (6 + what's in %rax), and copy data there into %rdx	Copy 6 + what's in %rax into %rdx.

lea vs. mov

Operands	mov Interpretation	lea Interpretation
`6(%rax), %rdx`	Go to the address (6 + what's in %rax), and copy data there into %rdx	Copy 6 + what's in %rax into %rdx.
`(%rax, %rcx), %rdx`	Go to the address (what's in %rax + what's in %rcx) and copy data there into %rdx	Copy (what's in %rax + what's in %rcx) into %rdx.

lea vs. mov

Operands	mov Interpretation	lea Interpretation
`6(%rax), %rdx`	Go to the address (6 + what's in %rax), and copy data there into %rdx	Copy 6 + what's in %rax into %rdx.
`(%rax, %rcx), %rdx`	Go to the address (what's in %rax + what's in %rcx) and copy data there into %rdx	Copy (what's in %rax + what's in %rcx) into %rdx.
`(%rax, %rcx, 4), %rdx`	Go to the address (%rax + 4 * %rcx) and copy data there into %rdx.	Copy (%rax + 4 * %rcx) into %rdx.

lea vs. mov

Operands	mov Interpretation	lea Interpretation
`6(%rax), %rdx`	Go to the address (6 + what's in %rax), and copy data there into %rdx	Copy 6 + what's in %rax into %rdx.
`(%rax, %rcx), %rdx`	Go to the address (what's in %rax + what's in %rcx) and copy data there into %rdx	Copy (what's in %rax + what's in %rcx) into %rdx.
`(%rax, %rcx, 4), %rdx`	Go to the address (%rax + 4 * %rcx) and copy data there into %rdx.	Copy (%rax + 4 * %rcx) into %rdx.
`7(%rax, %rax, 8), %rdx`	Go to the address (7 + %rax + 8 * %rax) and copy data there into %rdx.	Copy (7 + %rax + 8 * %rax) into %rdx.

Unlike **mov**, which copies data at the address src to the destination, **lea** copies the value of src *itself* to the destination.

Reverse Engineering Practice

```
void calculate(int x, int y, int *ptr) {
    ____?____;
}

----------

calculate:
  leal (%rdi,%rsi,2), %eax
  movl %eax, (%rdx)
  ret
```

Note: assume x is in %rdi, y is in %rsi and ptr is in %rdx.

Reverse Engineering Practice

```
void calculate(int x, int y, int *ptr) {
    *ptr = x + 2 * y;
}

----------

calculate:
  leal (%rdi,%rsi,2), %eax
  movl %eax, (%rdx)
  ret
```

A Note About Operand Forms

- Many instructions share the same address operand forms that **mov** uses.
  - e.g., 7(%rax, %rcx, 2).
- These forms work the same way for other instructions (exception, **lea**):
  - It interprets this form as just the calculation, *not the dereferencing*
  - lea 8(%rax,%rdx),%rcx -> Calculate 8 + %rax + %rdx, put it in %rcx

Unary Instructions

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:

```
incq 16(%rax)
dec %rdx
not %rcx
```

Binary Instructions

The following instructions operate on two operands (both can be register or memory, source can also be immediate). Both cannot be memory locations. Read it as, e.g., 'subtract S from D':

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:

```
addq %rcx,(%rax)
xorq $16,(%rax, %rdx, 8)
subq %rdx,8(%rax)
```

Shift Instructions

The following instructions have two operands: the shift amount **k** and the destination to shift, **D**. **k** can be either an immediate value, or the byte register **%cl** (and only that register!)

Instruction	Effect	Description
`sal k, D`	$D \leftarrow D \ll k$	Left shift
`shl k, D`	$D \leftarrow D \ll k$	Left shift (same as `sal`)
`sar k, D`	$D \leftarrow D \gg_A k$	Arithmetic right shift
`shr k, D`	$D \leftarrow D \gg_L k$	Logical right shift

Examples:

```
shll $3,(%rax)
shrl %cl,(%rax,%rdx,8)
sarl $4,8(%rax)
```

Shift Amount

Instruction	Effect	Description
`sal k, D`	$D \leftarrow D \ll k$	Left shift
`shl k, D`	$D \leftarrow D \ll k$	Left shift (same as `sal`)
`sar k, D`	$D \leftarrow D \gg_A k$	Arithmetic right shift
`shr k, D`	$D \leftarrow D \gg_L k$	Logical right shift

- When using **%cl**, the width of what you are shifting determines what portion of **%cl** is used.
- For **w** bits of data, it looks at the low-order **log2(w)** bits of **%cl** to know how much to shift.
  - If **%cl** = 0xff, then: **shlb** shifts by 7 because it considers only the low-order log2(8) = 3 bits, which represent 7. **shlw** shifts by 15 because it considers only the low-order log2(16) = 4 bits, which represent 15.