CMPT 295

Unit - Machine-Level Programming

Lecture 10 – Assembly language basics: `leaq` instruction, memory addressing modes and arithmetic & logical operations

Last Lecture

- As x86-64 assembly s/w dev., we now get to see more of the microprocessor (CPU) state: PC, registers, condition codes
- x86-64 assembly language – ***Data***
  - 16 integer registers of 1, 2, 4 or 8 bytes + memory address of 8 bytes
  - Floating point registers of 4 or 8 bytes
  - No aggregate types such as arrays or structures
- x86-64 assembly language – ***Instructions***
  - `mov*` instruction family
    - From register to register
    - From memory to register
    - From register to memory
  - Memory addressing modes
  - *Cannot do memory-memory transfer with a single mov* instruction*

Why *cannot do memory-memory transfer with a single* `mov*` *instruction?*

- No x86-64 assembly instructions that take 2 memory addresses as operands
- Such instruction would
  - Makes for very long machine instructions
  - Require more complex decoder unit (on microprocessor)
    in other words, require more complex microprocessor *datapath*
  - Memory only has one data bus and one address bus

  ⇒ *No appetite for instruction set architects to create such instructions*

  ⇒ *Registers very fast and can easily be used for such transfer*
- More info here:

https://stackoverflow.com/questions/33794169/why-isnt-movl-from-memory-to-memory-allowed

Last Lecture

- Requirement: When reading/writing assembly code …

… add a comment at the top of your function in your assembly code describing the parameter-to-register mapping

```
swap:
  # xp -> %rdi, yp -> %rsi
   movq     (%rdi), %rax   # L1 = *xp
   movq     (%rsi), %rdx   # L2 = *yp
   movq     %rdx, (%rdi)   # *xp = L2
   movq     %rax, (%rsi)   # *yp = L1
   ret
```

Comment each of your assembly language instruction by explaining what it does using corresponding C statement or pseudocode

Today's Menu

- Introduction
  - C program -> assembly code -> machine level code
- Assembly language basics: data, `move` operation
  - Memory addressing modes
- Operation `leaq` and Arithmetic & logical operations
- Conditional Statement – Condition Code + `cmov*`
- Loops
- Function call – Stack
- Array
- Buffer Overflow
- Floating-point operations

Various types of operands to x86-64 instructions

1. Integer value as operand directly in an instruction
   - This operand is called **immediate**
   - Operand syntax: `Imm`
   - Examples: `movq $0x4,%rax` and `movb $-17,%al`
2. Registers as operands in an instruction
   - Operand value: `R[`r_a`]`
   - Operand syntax: `%`r_a ← name of particular register
   - Example: `movq %rax,%rdx`
3. Memory address – using various memory addressing modes as operands in an instruction

These instructions copy immediate value to register

So far, this is the type of operands what we have seen!

This instruction copies the value of one register into another register

Memory addressing modes

We access memory in an x86-64 instruction by expressing a memory address through various ***memory addressing modes***

1. **Absolute** memory addressing mode
   - Use memory address as operand directly in instruction
     - The operand is also called **immediate**
   - Operand syntax: `Imm`
   - Effect: `M[Imm]`
   - Example: `call plus`
2. **Indirect** memory addressing mode

`plus` refers to the memory address of the first byte of the first instruction of the function called `plus` (see Demo)

2. **Indirect** memory addressing mode

- When a register contains an address
  - Similar to a pointer in C
- To access the data at the address contained in the register, we use parentheses **(...)**
- General Syntax: **(r_b)**
- Effect: **M[R[r_b]]**

2. **Indirect** memory addressing mode

- Example: | register to register: `movq %rdx,%rax` vs memory to register: `movq (%rdx),%rax`

Meaning or effect: `rax <- rdx` vs `rax <- M[rdx]`

or: `R[rax] <- R[rdx]` | `R[rax] <- M[R[rdx]]`

	Before	After	Before	After
	%rax = 15	**%rax = 6**	%rax = 15	**%rax = 11**
	%rdx = 6	%rdx = 6	%rdx = 6	%rdx = 6
not used	M[6] = 11	M[6] = 11	M[6] = 11	M[6] = 11

- Other examples: `movq %rax,(%rdx)` <- register to memory

  `movq $-147,(%rax)` <- immediate to memory *leaq* has the form of an instruction that reads from memory to a register (because of the parentheses), however it ***does not*** reference **memory at all!**

`leaq` - *Load effective address* instruction

- Often used for ***address computations*** and ***general arithmetic computations***
- **Syntax:** **`leaq`** ***`Source, Destination`***
- Example:
  1. Computing addresses **if %rax <- 0x0000000000000008 and %rcx <- 16**

     `leaq (%rax, %rcx), %rdx`

     Once executed, rdx will contain 0x18
  2. Computing arithmetic expressions of the form **x + k*y** where k ∈ {1,2,4,8} **if %rdi <- variable a**

     C code: `return a*3;`

     x y k

     `leaq (%rdi, %rdi, 2), %rax`

     Once executed, rax will contain 3a
- Operand ***Destination*** is a register
- Operand ***Source*** is a ***memory addressing mode*** expression

3. "**Base + displacement**" memory addressing mode

- General Syntax: `Imm(rb)` ($Imm(r_b)$)
- Effect: $M[Imm + R[r_b]]$
- Examples: `movq %rax, -8(%rsp)`

  `leaq 7(%rdi), %rax`
- **Careful here!**
  - When dealing with `leaq`, the effect is $Imm + R[r_b]$

    not $M[Imm + R[r_b]]$

4. **Indexed** memory addressing mode

1. General Syntax: (r_b, r_i)

- Effect: $M[R[r_b] + R[r_i]]$
- Example: **`movb (%rdi, %rcx), %al`**

2. General Syntax: $Imm(r_b, r_i)$

- Effect: $M[Imm + R[r_b] + R[r_i]]$
- Example: **`movw 0xA(%rdi, %rcx), %r11w`**

Careful here!

- When dealing with **`leaq`**, the effect is
  1. $R[r_b] + R[r_i]$ ***not*** $M[R[r_b] + R[r_i]]$
  2. $Imm + R[r_b] + R[r_i]$ ***not*** $M[Imm + R[r_b] + R[r_i]]$

5. **Scaled** indexed memory addressing mode

1. General Syntax: `(,`r_i`,s)` Effect: `M[R[`r_i`] * s]`
   - Example: `(, %rdi, 2)`
2. General Syntax: *Imm*`(,`r_i`,s)` Effect: `M[`*Imm* `+ R[`r_i`] * s]`
   - Example: `3(, %rcx, 8)`
3. General Syntax: `(`r_b`,`r_i`,s)` Effect: `M[R[`r_b`] + R[`r_i`] * s]`
   - Example: `(%rdi, %rsi, 4)`
4. General Syntax: *Imm*`(`r_b`,`r_i`,s)` Effect: `M[`*Imm* `+ R[`r_b`] + R[`r_i`] * s]`
   - Example: `8(%rdi, %rsi, 4)`

Again, careful here!

- When dealing with `leaq`, the effect is ***not*** to reference **memory at all!**

Summary - Memory addressing modes

We access memory in an x86-64 instruction by expressing a memory address through various ***memory addressing modes***

1. **Absolute**
2. **Indirect**
3. **"Base + displacement"**
4. **2 indexed**
5. **4 scaled indexed**

General Syntax: `Imm(`r_b`, `r_i`, s)`

Effect: `M[Imm + R[`r_b`]+ R[`r_i`] * s]`

See Table of x86-64 Addressing Modes

on Resources web page of our course web site

Let's try it!

`%rdx`	`0xf000`
`%rcx`	`0x0100`

Expression	Address Computation	Address
`8(%rdx)`		
`(%rdx,%rcx)`		
`(%rdx,%rcx,4)`		
`0x80(,%rdx,2)`		
`0x80(%rdx, 2)`		
`0x80(,%rdx, 3)`		

* -> Size designator

```
q -> long  64
l -> int   32
w -> short 16
b -> char   8
```

Two-Operand Arithmetic Instructions

Syntax	Meaning	Examples	in C
`add* Src, Dest`	`Dest ← Dest + Src`	`addq %rax, %rcx`	`x += y`
`sub* Src, Dest`	`Dest ← Dest - Src`	`subq %rax, %rcx`	`x -= y`
`imul* Src, Dest`	`Dest ← Dest * Src`	`imulq $16,(%rax,%rdx,8)`	`x *= y`

- "destination" and "first operand" are the same
  - "2 operand" assembly language (machine)
- **`mem ← mem OP mem`** usually not supported
- 2 assembly code formats: **ATT and Intel format** (see Aside in Section 3.2 P. 177)
  - We are using the **ATT format**
  - Both order the operands of their instructions differently - Watch out!

* -> Size designator

```
q -> long  64
l -> int   32
w -> short 16
b -> char   8
```

Two-Operand Logical Instructions

Syntax	Meaning	Examples
`and* Src, Dest`	`Dest ← Dest & Src`	`andl $252645135, %edi`
`or* Src, Dest`	`Dest ← Dest | Src`	`orq %rsi, %rdi`
`xor* Src, Dest`	`Dest ← Dest ^ Src`	`xorq %rsi, %rdi`

- `xorq` special purpose:
  - `xorq %rax, %rax` <- *zeroes* register `%rax`
  - `movq $0,  %rax` <- also *zeroes* register `%rax`
- x86-64 convention:
  - Any instruction updating the lower 4 bytes will cause the higher-order bytes to be set to 0
  - `xorl %eax, %eax` and `movl $0, %eax` <- also *zeroes* register `%rax`

* -> Size designator

```
q -> long  64
l -> int   32
w -> short 16
b -> char   8
```

Two-Operand Shift Instructions

Syntax	Meaning	Examples
`sal* Src, Dest`	`Dest ← Dest << Src`	`salq $4, %rax`
`sar* Src, Dest`	`Dest ← Dest >> Src`	`sarl %cl, %rax`
`shr* Src, Dest`	`Dest ← Dest >> Src`	`shrq $2, %r8`

- *Left shift* - also called **shlq**: filling **Dest** with 0, from the right
- *Right arithmetic Shift*: filling **Dest** with sign bit, from the left
- *Right logical Shift*: filling **Dest** with 0, from the left

* -> Size designator

```
q -> long  64
l -> int   32
w -> short 16
b -> char   8
```

One-Operand Arithmetic Instructions

Syntax	Meaning	Examples
`inc*` *Dest*	Dest ← Dest + 1	`incq (%rsp)`
`dec*` *Dest*	Dest ← Dest – 1	`decq %rsi`
`neg*` *Dest*	Dest ← –Dest	`negl %eax`
`not*` *Dest*	Dest ← ~Dest	`notq %rdi`

Summary

- `leaq` - load effective address instruction
- Various types of operands to x86-64 instructions
  - Immediate (constant integral value)
  - Register (16 registers)
  - Memory address (various memory addressing modes)
    - General Syntax: **Imm(r_b, r_i, s)**
- Arithmetic & logical operations
  - Arithmetic instructions: **add*, sub*, imul* inc*, dec*, neg*, not***
  - Logical instructions: **and*, or*, xor***
  - Shift instructions: **sal*, sar*, shr***

Next lecture

- Introduction
  - C program -> assembly code -> machine level code
- Assembly language basics: data, `move` operation
  - Memory addressing modes
- Operation `leaq` and Arithmetic & logical operations
- Conditional Statement – Condition Code + `cmov*`
- Loops
- Function call – Stack
- Array
- Buffer Overflow
- Floating-point operations

Practice and DEMO!