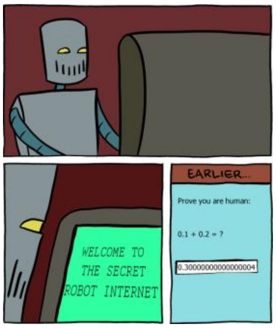
x86-64 Programming I

CSE 351 Winter 2018

Instructor:
Mark Wyse

Teaching Assistants:
Kevin Bi
Parker DeWilde
Emily Furst
Sarah House
Waylon Huang
Vinny Palaniappan



http://www.smbc-comics.com/?id=2999

Administrivia

- Homework 2 due Wednesday (1/24)
  - x86-64 part extended until Monday, January 29
- Lab 2 (x86-64) released today (1/22)
  - Due on Friday, February 2 @ 11:59 pm
- Don’t forget to read the book!
  - Ch. 3 will be very helpful!

---

x86-64 Programming I

- Operand types
- Moving data
- Arithmetic operations
- Memory addressing modes
  - `swap` example
- Address computation instruction (`lea`)

Operand types

- ***Immediate:*** Constant integer data
  - Examples: **`$0x400`**, **`$-533`**
  - Like C literal, but prefixed with **`‘$’`**
  - Encoded with 1, 2, 4, or 8 bytes *depending on the instruction*
- ***Register:*** 1 of 16 integer registers
  - Examples: **`%rax`**, **`%r13`**
  - But **`%rsp`** reserved for special use
  - Others have special uses for particular instructions
- ***Memory:*** Consecutive bytes of memory at a computed address
  - Simplest example: **`(%rax)`**
  - Various other “address modes”

`%rax`
`%rcx`
`%rdx`
`%rbx`
`%rsi`
`%rdi`
`%rsp`
`%rbp`

`%rN`

Moving Data

- General form: `mov_ source, destination`
  - Missing letter (_) specifies size of operands
  - Note that due to backwards-compatible support for 8086 programs (16-bit machines!), “word” means 16 bits = 2 bytes in x86 instruction names
  - Lots of these in typical code
- `movb src, dst`
  - Move 1-byte “**b**yte”
- `movw src, dst`
  - Move 2-byte “**w**ord”
- `movl src, dst`
  - Move 4-byte “**l**ong word”
- `movq src, dst`
  - Move 8-byte “**q**uad word”

`movq` Operand Combinations

	Source	Dest	Src, Dest	C Analog
`movq`	Imm	Reg	`movq $0x4, %rax`	`var_a = 0x4;`
		Mem	`movq $-147, (%rax)`	`*p_a = -147;`
	Reg	Reg	`movq %rax, %rdx`	`var_d = var_a;`
		Mem	`movq %rax, (%rdx)`	`*p_d = var_a;`
	Mem	Reg	`movq (%rax), %rdx`	`var_d = *p_a;`

- *Cannot do memory-memory transfer with a single instruction*
  - How would you do it?

Some Arithmetic Operations

- Binary (two-operand) Instructions:
  - **Maximum of one memory operand**
  - Beware argument order!
  - No distinction between signed and unsigned
    - Only arithmetic vs. logical shifts
  - How do you implement "`r3 = r1 + r2`"?

Format	Computation	
`addq` *`src, dst`*	*dst = dst + src*	(*dst += src*)
`subq` *`src, dst`*	*dst = dst – src*	
`imulq` *`src, dst`*	*dst = dst * src*	signed mult
`sarq` *`src, dst`*	*dst = dst >> src*	**A**rithmetic
`shrq` *`src, dst`*	*dst = dst >> src*	**L**ogical
`shlq` *`src, dst`*	*dst = dst << src*	(same as `salq`)
`xorq` *`src, dst`*	*dst = dst ^ src*	
`andq` *`src, dst`*	*dst = dst & src*	
`orq` *`src, dst`*	*dst = dst | src*	

↑ operand size specifier

Some Arithmetic Operations

- Unary (one-operand) Instructions:

Format	Computation	
`incq` *`dst`*	*dst = dst + 1*	increment
`decq` *`dst`*	*dst = dst – 1*	decrement
`negq` *`dst`*	*dst = –dst*	negate
`notq` *`dst`*	*dst = ~dst*	bitwise complement

- See CSPP Section 3.5.5 for more instructions: `mulq`, `cqto`, `idivq`, `divq`

Arithmetic Example

Register	Use(s)
`%rdi`	1st argument (`x`)
`%rsi`	2nd argument (`y`)
`%rax`	return value

```
long simple_arith(long x, long y)
{
  long t1 = x + y;
  long t2 = t1 * 3;
  return t2;
}
```

```
y += x;
y *= 3;
long r = y;
return r;
```

```
simple_arith:
  addq    %rdi, %rsi
  imulq     $3, %rsi
  movq    %rsi, %rax
  ret
```

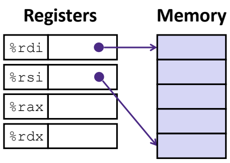
Example of Basic Addressing Modes

```
void swap(long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
   movq   (%rdi), %rax
   movq   (%rsi), %rdx
   movq   %rdx, (%rdi)
   movq   %rax, (%rsi)
   ret
```

Understanding `swap()`

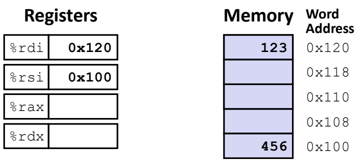
```
void swap(long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```



```
swap:
   movq   (%rdi), %rax
   movq   (%rsi), %rdx
   movq   %rdx, (%rdi)
   movq   %rax, (%rsi)
   ret
```

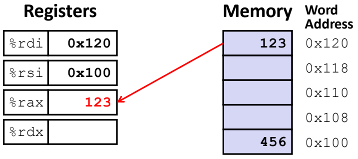
Register		Variable
`%rdi`	⇔	`xp`
`%rsi`	⇔	`yp`
`%rax`	⇔	`t0`
`%rdx`	⇔	`t1`

Understanding `swap()`



```
swap:
   movq   (%rdi), %rax   #  t0 = *xp
   movq   (%rsi), %rdx   #  t1 = *yp
   movq   %rdx, (%rdi)   # *xp =  t1
   movq   %rax, (%rsi)   # *yp =  t0
   ret
```

Understanding `swap()`

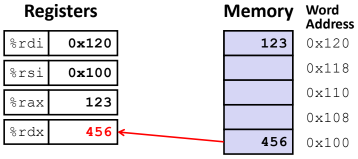


Registers	
%rdi	0x120
%rsi	0x100
%rax	123
%rdx	

Memory	Word Address
123	0x120
	0x118
	0x110
	0x108
456	0x100

```
swap:
   movq   (%rdi), %rax   #  t0 = *xp
   movq   (%rsi), %rdx   #  t1 = *yp
   movq   %rdx, (%rdi)   # *xp =  t1
   movq   %rax, (%rsi)   # *yp =  t0
   ret
```

Understanding `swap()`

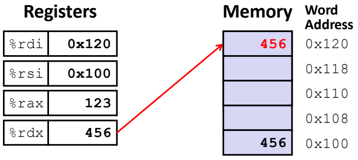


Registers	
%rdi	0x120
%rsi	0x100
%rax	123
%rdx	456

Memory	Word Address
123	0x120
	0x118
	0x110
	0x108
456	0x100

```
swap:
   movq   (%rdi), %rax   #  t0 = *xp
   movq   (%rsi), %rdx   #  t1 = *yp
   movq   %rdx, (%rdi)   # *xp =  t1
   movq   %rax, (%rsi)   # *yp =  t0
   ret
```

Understanding `swap()`

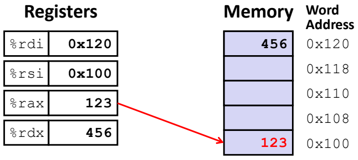


Registers	
%rdi	0x120
%rsi	0x100
%rax	123
%rdx	456

Memory	Word Address
456	0x120
	0x118
	0x110
	0x108
456	0x100

```
swap:
   movq   (%rdi), %rax   #  t0 = *xp
   movq   (%rsi), %rdx   #  t1 = *yp
   movq   %rdx, (%rdi)   # *xp =  t1
   movq   %rax, (%rsi)   # *yp =  t0
   ret
```

Understanding `swap()`



Registers	
%rdi	0x120
%rsi	0x100
%rax	123
%rdx	456

Memory	Word Address
456	0x120
	0x118
	0x110
	0x108
123	0x100

```
swap:
   movq   (%rdi), %rax   #  t0 = *xp
   movq   (%rsi), %rdx   #  t1 = *yp
   movq   %rdx, (%rdi)   # *xp =  t1
   movq   %rax, (%rsi)   # *yp =  t0
   ret
```

Memory Addressing Modes: Basic

- **Indirect:** `(R)` Mem[Reg[R]]
  - Data in register `R` specifies the memory address
  - Like pointer dereference in C
  - Example: `movq (%rcx), %rax`
- **Displacement:** `D(R)` Mem[Reg[R]+D]
  - Data in register `R` specifies the *start* of some memory region
  - Constant displacement `D` specifies the offset from that address
  - Example: `movq 8(%rbp), %rdx`

Complete Memory Addressing Modes

- **General:**
  - `D(Rb,Ri,S)` Mem[Reg[Rb]+Reg[Ri]*S+D]

`Rb`:	Base register (any register)
`Ri`:	Index register (any register except `%rsp`)
`S`:	Scale factor (1, 2, 4, 8) – *why these numbers?*
`D`:	Constant displacement value (a.k.a. immediate)

- **Special cases** (see CSPP Figure 3.3 on p.181)

`D(Rb,Ri)`	Mem[Reg[Rb]+Reg[Ri]+D]	`(S=1)`
`(Rb,Ri,S)`	Mem[Reg[Rb]+Reg[Ri]*S]	`(D=0)`
`(Rb,Ri)`	Mem[Reg[Rb]+Reg[Ri]]	`(S=1,D=0)`
`(,Ri,S)`	Mem[Reg[Ri]*S]	`(Rb=0,D=0)`

Address Computation Examples

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

`D(Rb,Ri,S)` →
Mem[Reg[Rb]+Reg[Ri]*S+D]

Expression	Address Computation	Address
`0x8(%rdx)`		
`(%rdx,%rcx)`		
`(%rdx,%rcx,4)`		
`0x80(,%rdx,2)`		

Address Computation Instruction

- `leaq src, dst`
  - "`lea`" stands for *load effective address*
  - `src` is address expression (any of the formats we've seen)
  - `dst` is a register
  - Sets `dst` to the *address* computed by the `src` expression (does not go to memory! – it just does math)
  - Example: `leaq (%rdx,%rcx,4), %rax`
- Uses:
  - Computing addresses without a memory reference
    - *e.g.* translation of `p = &x[i];`
  - Computing arithmetic expressions of the form `x+k*i+d`
    - Though `k` can only be 1, 2, 4, or 8

Example: `lea` vs. `mov`

Registers

`%rax`	
`%rbx`	
`%rcx`	**0x4**
`%rdx`	**0x100**
`%rdi`	
`%rsi`	

Memory	Word Address
0x400	0x120
0xF	0x118
0x8	0x110
0x10	0x108
0x1	0x100

```
leaq (%rdx,%rcx,4), %rax
movq (%rdx,%rcx,4), %rbx
leaq (%rdx), %rdi
movq (%rdx), %rsi
```

Arithmetic Example

Register	Use(s)
`%rdi`	1st argument (`x`)
`%rsi`	2nd argument (`y`)
`%rdx`	3rd argument (`z`)

```
long arith(long x, long y, long z)
{
  long t1 = x + y;
  long t2 = z + t1;
  long t3 = x + 4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

```
arith:
  leaq    (%rdi,%rsi), %rax
  addq    %rdx, %rax
  leaq    (%rsi,%rsi,2), %rdx
  salq    $4, %rdx
  leaq    4(%rdi,%rdx), %rcx
  imulq   %rcx, %rax
  ret
```

- Interesting Instructions
  - `leaq`: "address" computation
  - `salq`: shift
  - `imulq`: multiplication
    - Only used once!

Arithmetic Example

Register	Use(s)
`%rdi`	`x`
`%rsi`	`y`
`%rdx`	`z, t4`
`%rax`	`t1, t2, rval`
`%rcx`	`t5`

```
long arith(long x, long y, long z)
{
  long t1 = x + y;
  long t2 = z + t1;
  long t3 = x + 4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

```
arith:
  leaq    (%rdi,%rsi), %rax     # rax/t1   = x + y
  addq    %rdx, %rax            # rax/t2   = t1 + z
  leaq    (%rsi,%rsi,2), %rdx   # rdx      = 3 * y
  salq    $4, %rdx              # rdx/t4   = (3*y) * 16
  leaq    4(%rdi,%rdx), %rcx    # rcx/t5   = x + t4 + 4
  imulq   %rcx, %rax            # rax/rval = t5 * t2
  ret
```

Peer Instruction Question

- Which of the following x86-64 instructions correctly calculates `%rax=9*%rdi`?

A. `leaq (,%rdi,9), %rax`

B. `movq (,%rdi,9), %rax`

C. `leaq (%rdi,%rdi,8), %rax`

D. `movq (%rdi,%rdi,8), %rax`

E. We're lost…

Summary

- **Operands:** Assembly operands include *immediates*, *registers*, and *data at a specified memory location*.
- **Memory Addressing Modes:** The addresses used for accessing memory in `mov` (and other) instructions can be computed in several different ways
  - *Base register*, *index register*, *scale factor*, and *displacement* map well to pointer arithmetic operations
- `lea` is address calculation instruction
  - Does NOT actually go to memory
  - Used to compute addresses or some arithmetic expressions