x86-64 Programming II

CSE 351 Autumn 2022

Instructor:

Justin Hsia

Teaching Assistants:

Angela Xu
Arjun Narendra
Armin Magness
Assaf Vayner
Carrie Hu
Clare Edmonds
David Dai
Dominick Ta
Effie Zheng
James Froelich
Jenny Peng
Kristina Lansang
Paul Stevans
Renee Ruan
Vincent Xiao

http://xkcd.com/99/

Relevant Course Information

- Lab submissions that fail the autograder get a **ZERO**
  - No excuses – make full use of tools & Gradescope's interface
  - Leeway on Lab 1a won't be given moving forward
- Lab 2 (x86-64) released today
  - Learn to trace x86-64 assembly and use GDB
- Midterm is in two weeks (take home, 11/3–5)
  - Open book; make notes and use midterm reference sheet
  - Individual, but discussion allowed via "Gilligan's Island Rule"
  - Mix of "traditional" and design/reflection questions
    - Form study groups and look at past exams!

Extra Credit

- All labs starting with Lab 2 have extra credit portions
  - These are meant to be fun extensions to the labs

- Extra credit points *don't* affect your lab grades
  - From the course policies: “they will be accumulated over the course and will be used to bump up borderline grades at the end of the quarter.”
  - Make sure you finish the rest of the lab before attempting any extra credit

Reading Review

- Terminology:
  - Address Computation Instruction (`lea`)
  - Condition codes: Carry Flag (`CF`), Zero Flag (`ZF`), Sign Flag (`SF`), and Overflow Flag (`OF`)
  - Test (`test`) and compare (`cmp`) assembly instructions
  - Jump (`j*`) and set (`set*`) families of assembly instructions

- Questions from the Reading?

Memory Addressing Modes (Review)

ar[i] ⟷ *(ar + i) → Mem[ar + i * sizeof(data type)]

- **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?*   data type widths
    - 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)

so reg name not interpreted as Rb

Address Computation Instruction (Review)

"Mem" Reg

- `leaq src, dst`
  - "`lea`" stands for *load effective address*
  - `src` is address expression (any of the formats we've seen)
    ↳ calculates Reg[Rb]+Reg[Ri]*S+D ~~Mem[ ]~~
  - `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];` address-of operator
  - Computing arithmetic expressions of the form `x+k*i+d`
    Reg[Rb]+Reg[Ri]*S+D
    - Though k can only be 1, 2, 4, or 8

Review Questions

- If `%rdx = 0xf000` and `%rcx = 0x100`, what addresses are dereferenced by the following memory operands?
  - `(%rdx, %rcx)` — Rb: `%rdx`, Ri: `%rcx`

    Reg[Rb]+Reg[Ri]*1

    $\boxed{0xf100}$
  - `0x80(,%rdx,2)` — D: `0x80`, Ri: `%rdx`, S: `2`

    Reg[Ri]*2+0x80

    $\boxed{0x1e080}$

    0xf000*2 = 0xf000 << 1 = 0x1e000

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

  no memory access, so must be lea

  S ∈ {1,2,4,8}

  - A. `leaq (,%rdi,~~9~~), %rax` — invalid syntax
  - B. `~~movq~~ (,%rdi,9), %rax` — invalid syntax
  - [✓ C.] `leaq (%rdi,%rdi,8), %rax` — %rax = 9*%rdi
  - D. `~~movq~~ (%rdi,%rdi,8), %rax` — %rax = Mem[9*%rdi]

Example: Basic Arithmetic

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

arbitrary! (for now...)

don't actually need new variables!

```
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;
```

must return in %rax

instr src , dst

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

Example: Using Memory

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

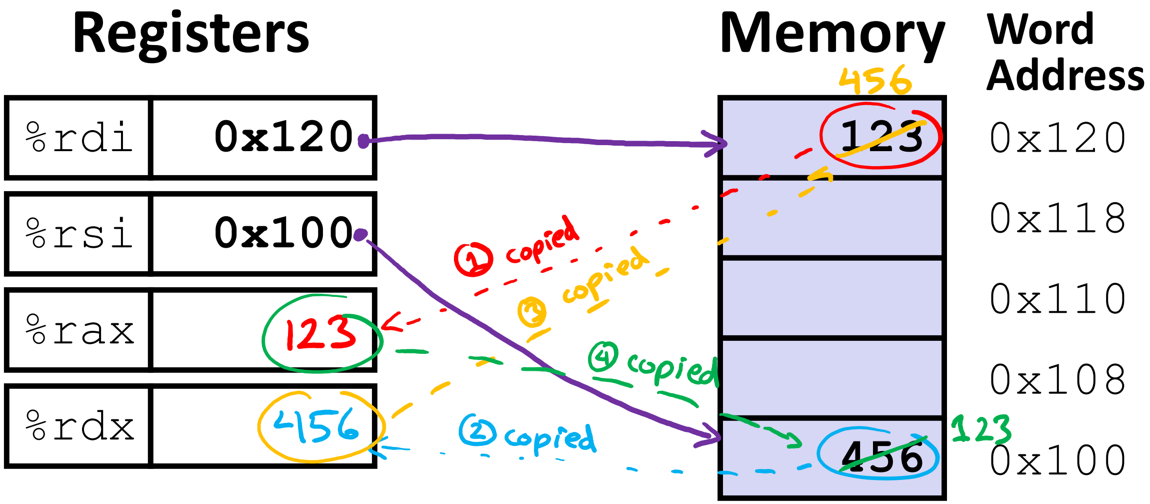
Compiler Explorer:
https://godbolt.org/z/c9M9fMefa

memory operands register operands

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

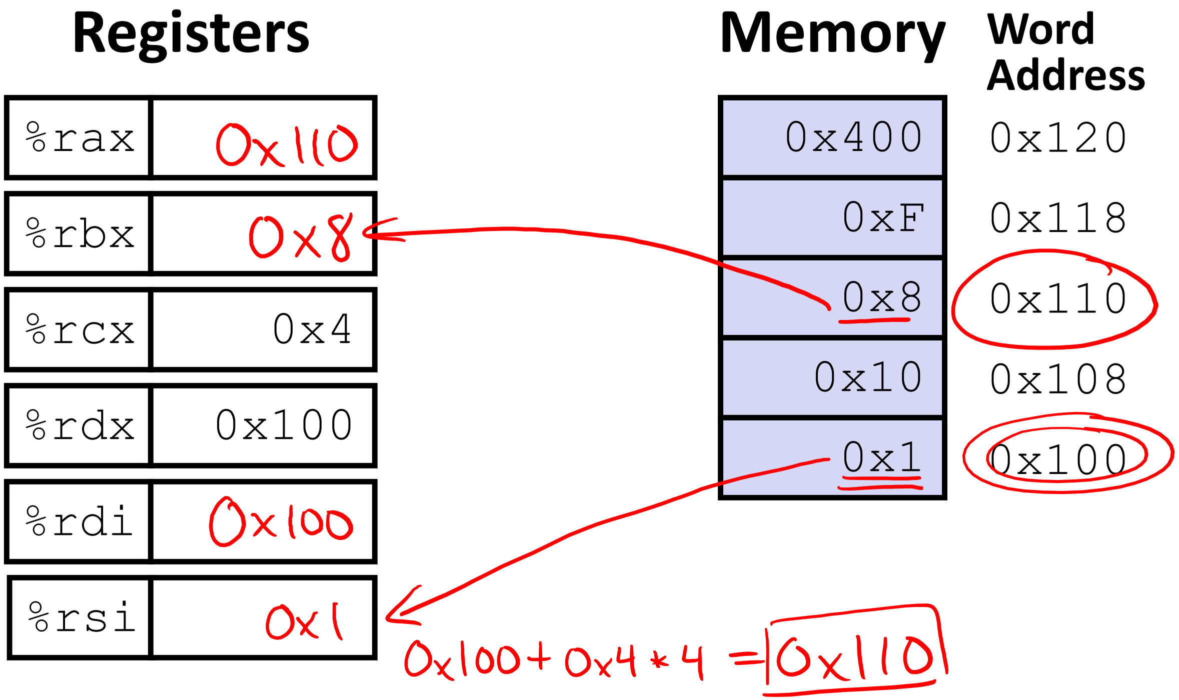
Register		Variable
%rdi	⇔	xp
%rsi	⇔	yp
%rax	⇔	t0
%rdx	⇔	t1

Example: Using Memory



```
swap:
①  movq   (%rdi), %rax    #  t0 = *xp
②  movq   (%rsi), %rdx    #  t1 = *yp
③  movq   %rdx, (%rdi)    # *xp =  t1
④  movq   %rax, (%rsi)    # *yp =  t0
    ret
```

Example: lea vs. mov



Registers

%rax	0x110
%rbx	0x8
%rcx	0x4
%rdx	0x100
%rdi	0x100
%rsi	0x1

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

$0x100 + 0x4 * 4 = \boxed{0x110}$

Rb Ri S

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

0x100

→ 0x110 ("addr")

→ 0x8 (data)

→ 0x100 ("addr")

→ 0x1 (data)

Example: lea Arithmetic

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;      ← replaced by lea & shift
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

```
arith:
            x     y
  leaq     (%rdi,%rsi), %rax      # rax = x+y (t1)
            z
  addq     %rdx, %rax             # rax = x+y+z (t2)
            y     y
  leaq     (%rsi,%rsi,2), %rdx    # rdx = 3y
  salq     $4, %rdx               # rdx = 48y (t4)
  leaq     4(%rdi,%rdx), %rcx
  imulq    %rcx, %rax    ← multiplying two variables
  ret
```

- Interesting Instructions
  - leaq: “address” computation
  - salq: shift
  - imulq: multiplication
    - Only used once!

Example: lea Arithmetic

```
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;
}
```

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

limited registers means they often get reused!

```
arith:                                       comment (AT&T syntax)
  leaq    (%rdi,%rsi), %rax       # rax/t1   = x + y
  addq    %rdx, %rax   S∈{1,2,4,8} # 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
```

Control Flow

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

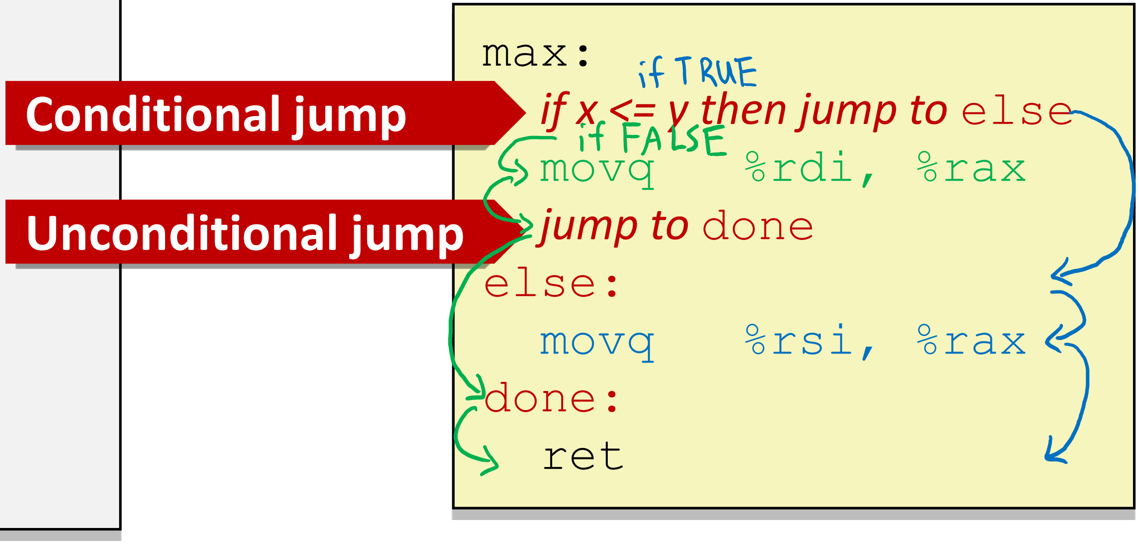
```
long max(long x, long y)
{
   long max;
   if (x > y) {
      max = x;
   } else {
      max = y;
   }
   return max;
}
```

```
max:
   ???
   movq    %rdi, %rax   # if case
   ???
   ???
   movq    %rsi, %rax   # else case
   ???
   ret
```

Control Flow

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

```
long max(long x, long y)
{
  long max;
  if (x > y) {
    max = x;
  } else {
    max = y;
  }
  return max;
}
```



Conditional jump

Unconditional jump

```
max:
  if x <= y then jump to else
  movq    %rdi, %rax
  jump to done
else:
  movq    %rsi, %rax
done:
  ret
```

Conditionals and Control Flow

- Conditional branch/*jump*
  - Jump to somewhere else if some *condition* is true, otherwise execute next instruction
- Unconditional branch/*jump*
  - *Always* jump when you get to this instruction

- Together, they can implement most control flow constructs in high-level languages:
  - **`if`** `(`*condition*`)` **`then`** `{…}` **`else`** `{…}`
  - **`while`** `(`*condition*`)` `{…}`
  - **`do`** `{…}` **`while`** `(`*condition*`)`
  - **`for`** `(`*initialization*`;` *condition*`;` *iterative*`)` `{…}`
  - **`switch`** `{…}`

Summary

- **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
- **Load effective address** (`lea`) instruction used to compute addresses and perform basic arithmetic
  - *Doesn't* dereference the source memory operand, unlike all other instructions!
- Control flow in x86 determined by Condition Codes