x86-64 Programming II

CSE 351 Autumn 2024

Instructor:
Ruth Anderson

Teaching Assistants:
Alexandra Michael
Connie Chen
Chloe Fong
Chendur Jayavelu
Joshua Tan
Nikolas McNamee
Nahush Shrivatsa
Naama Amiel
Neela Kausik
Renee Ruan
Rubee Zhao
Samantha Dreussi
Sean Siddens
Waleed Yagoub

http://xkcd.com/99/

Give us some feedback on course so far

On your index card:

- Please **Keep** doing this:
- Please **Quit** doing this:
- Please **Start** doing this:

Relevant Course Information

- HW6 due TONIGHT, Friday (10/11) @ 11:59 pm
- Lab 1b, due Monday (10/14) @ 11:59pm
  - No major programming restrictions, but should **avoid magic numbers by using C macros (**`#define`**)**
  - Submit `aisle_manager.c`, `store_client.c`, and `lab1Bsynthesis.txt`
- HW7 due Monday (10/14) @ 11:59 pm
- Lab 2 (x86-64) coming soon!
  - Learn to read x86-64 assembly and use GDB

Lab extras

- All labs starting with Lab 2 have extra "components"
  - These are meant to be fun extensions to the labs
- Extra components are not graded, nor do we expect you to do them
  - Make sure you finish the rest of the lab before attempting any extra components

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;
}
```

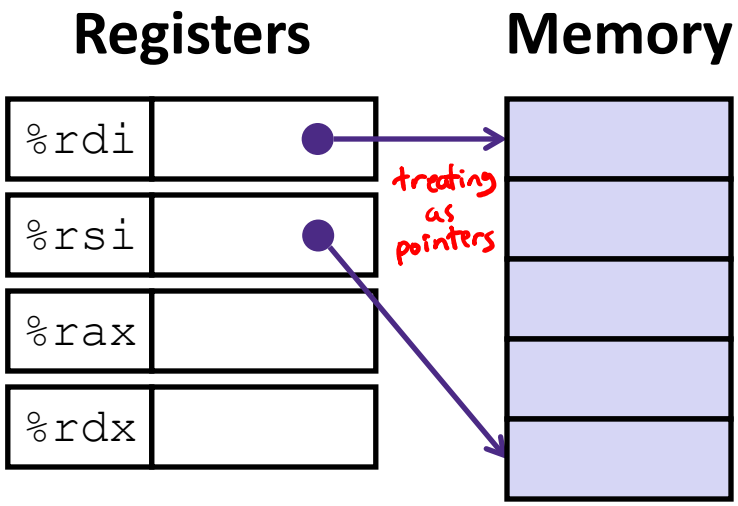
```
swap:
    movq   (%rdi), %rax
    movq   (%rsi), %rdx
    movq   %rdx, (%rdi)
    movq   %rax, (%rsi)
    ret
```

Mem operands

Compiler Explorer:
https://godbolt.org/z/zc4Pcq

Understanding swap()

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

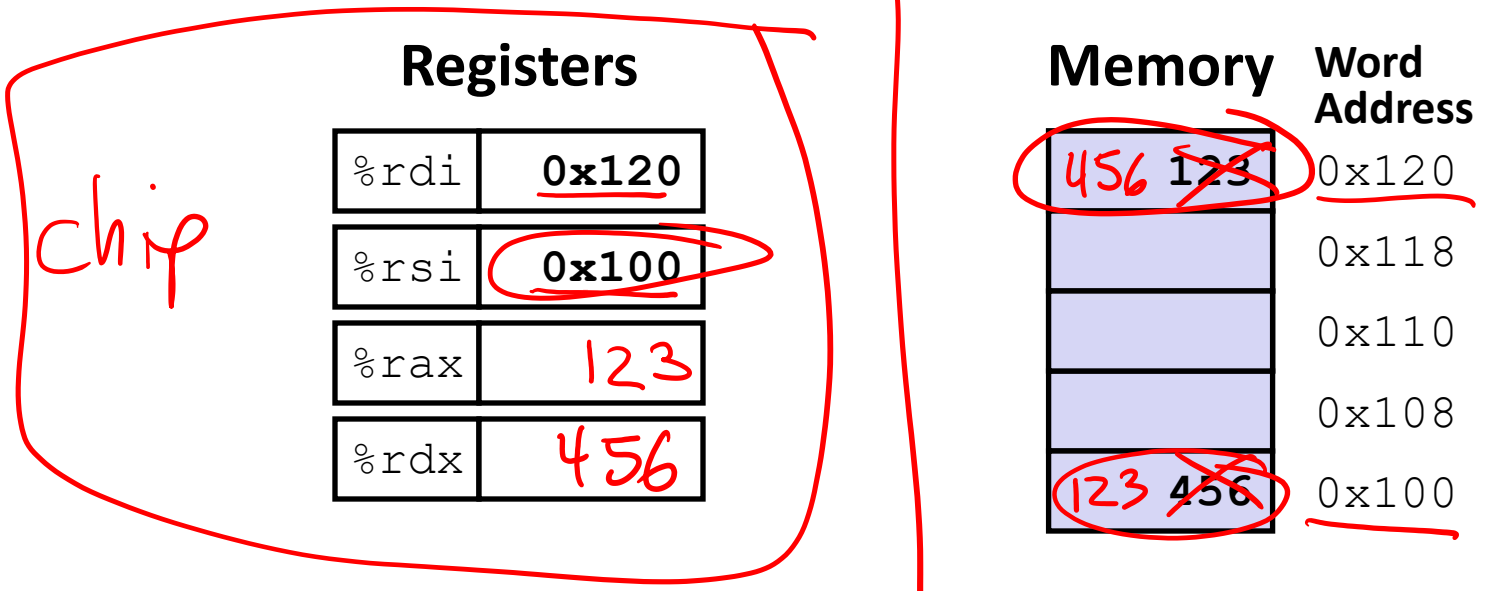



```
swap:
    movq   (%rdi), %rax
    movq   (%rsi), %rdx
    movq   %rdx, (%rdi)
    movq   %rax, (%rsi)
    ret
```

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

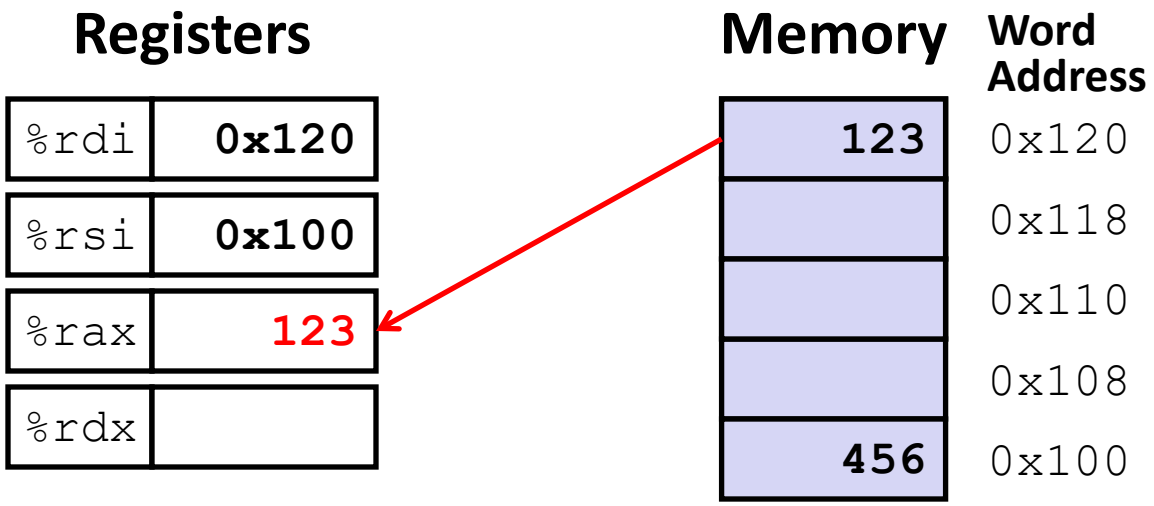
opcode src, dst

swap() walkthrough (1 of 5)



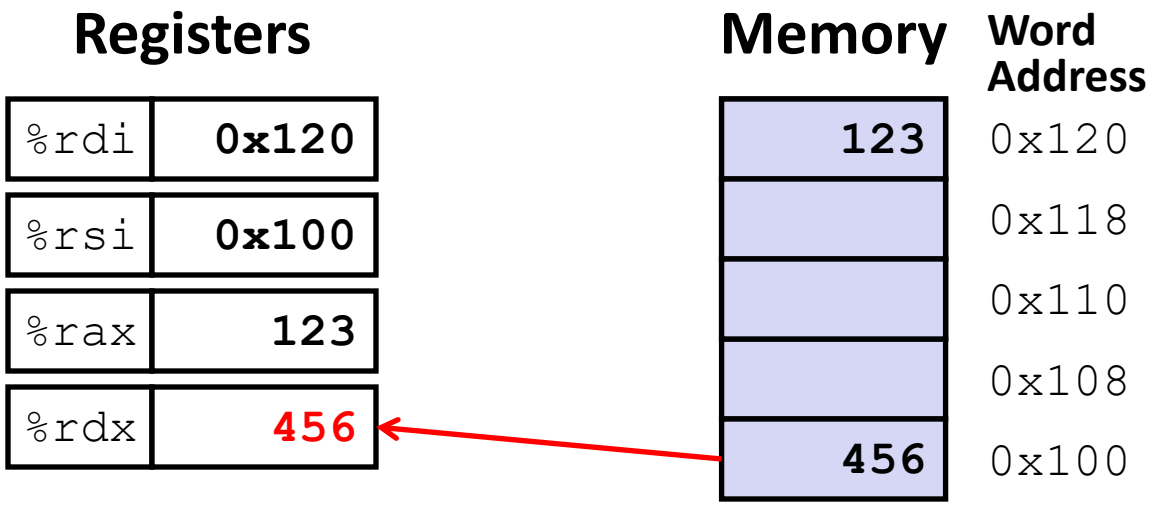

```
swap:
    movq   (%rdi), %rax   #  t0 = *xp
    movq   (%rsi), %rdx   #  t1 = *yp
    movq   %rdx, (%rdi)   # *xp =  t1
    movq   %rax, (%rsi)   # *yp =  t0
    ret
```

swap() walkthrough (2 of 5)



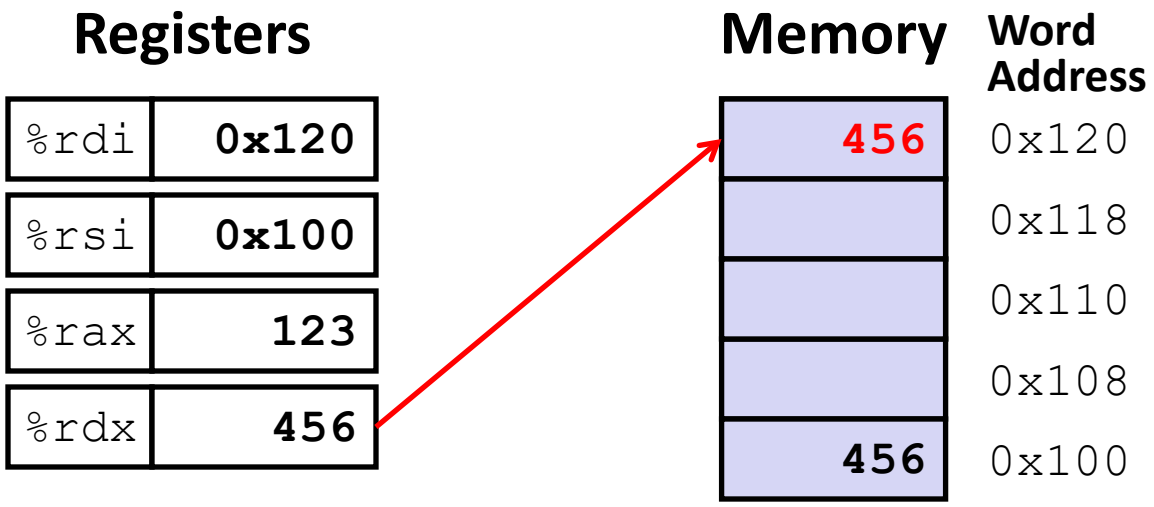

```
swap:
    movq   (%rdi), %rax   #  t0 = *xp
    movq   (%rsi), %rdx   #  t1 = *yp
    movq   %rdx, (%rdi)   # *xp =  t1
    movq   %rax, (%rsi)   # *yp =  t0
    ret
```

swap() walkthrough (3 of 5)



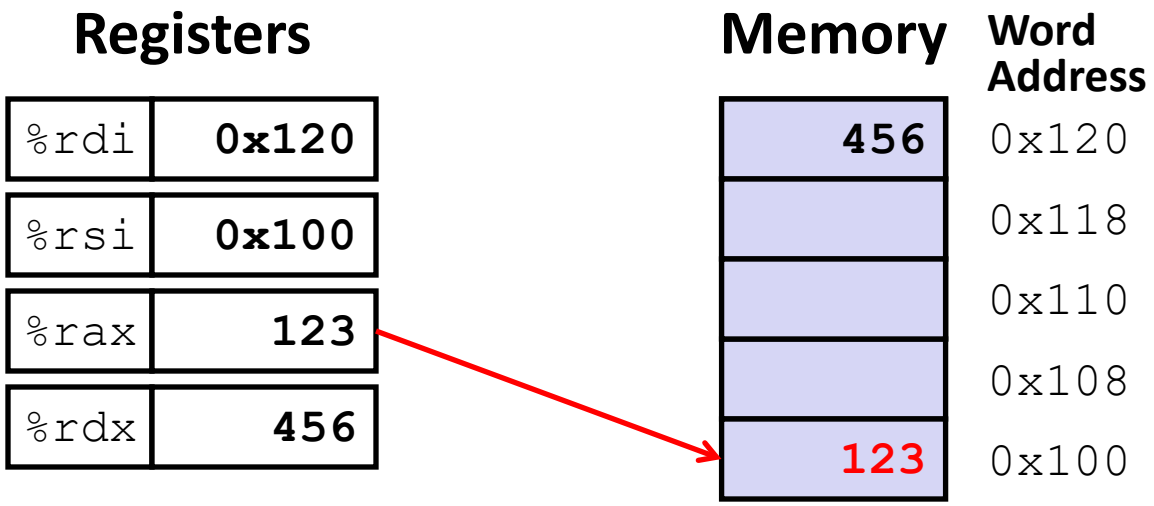

```
swap:
    movq   (%rdi), %rax   #  t0 = *xp
    movq   (%rsi), %rdx   #  t1 = *yp
    movq   %rdx, (%rdi)   # *xp = t1
    movq   %rax, (%rsi)   # *yp = t0
    ret
```

swap() walkthrough (4 of 5)




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

swap() walkthrough (5 of 5)




```
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)

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

data type widths

- **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 Examples

default values:
S = 1
D = 0
Reg[Rb] = 0
Reg[Ri] = 0

%rdx	0xf000
%rcx	0x0100

D(Rb,Ri,S) →
Mem[Reg[Rb]+(Reg[Ri]*S+D)]
ignore the memory access for now

Expression	Address Computation	Address (8 bytes wide)
0x8(%rdx)	0xf000 + 0x8	0xf008
(%rdx,%rcx)	0xf000 + 0x0100	0xf100
(%rdx,%rcx,4)	0xf000 + (0x100 * 4)	0xf400
0x80(,%rdx,2)	0x1e000 + 0x80	0x1e080

0xf000
0 .. .. 0000 1111 0000 0000 0000 0
1 e 0 0 0

Reading Review

- Terminology:
  - Address Computation Instruction (`lea`)

On Monday:

- 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

Review Question

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

no memory access, so must be lea

$S \in \{1, 2, 4, 8\}$

Not a valid scaling factor

A. `leaq (,%rdi,9), %rax` — invalid syntax

B. `movq (,%rdi,9), %rax` — invalid syntax

C. `leaq (%rdi,%rdi,8), %rax` — %rax = 9 * %rdi

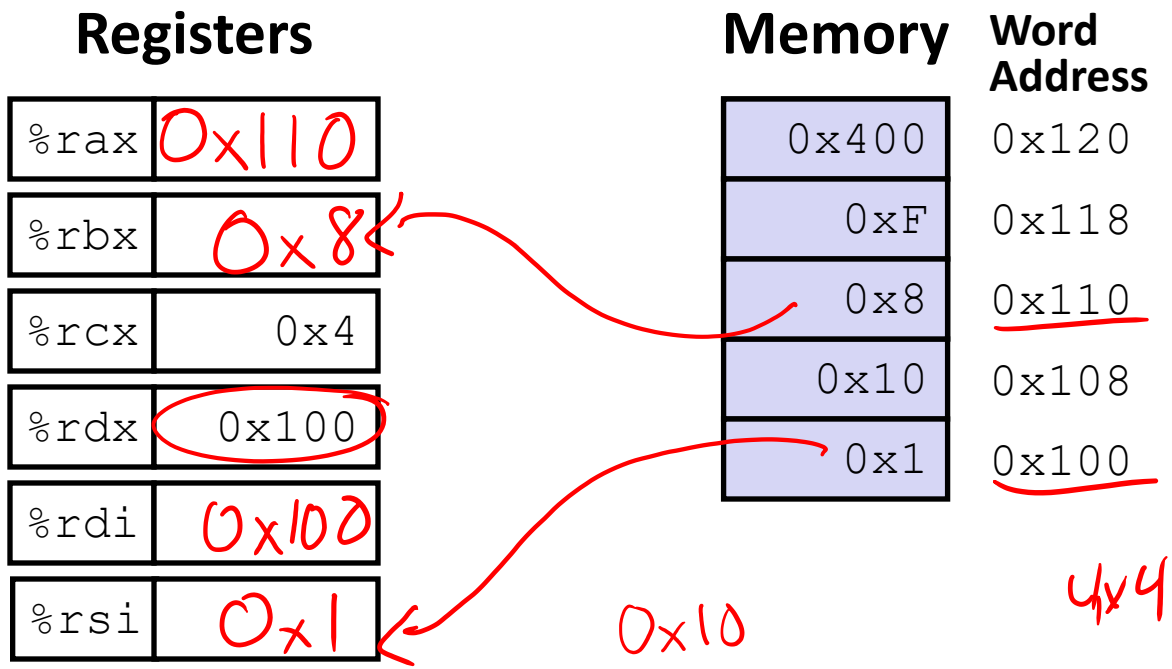
8 * rdi

D. `movq (%rdi,%rdi,8), %rax` — %rax = Mem[9 * %rdi]

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



`lea` – "It just does math"

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;
}
```

← replaced by lea & shift

```
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
```

rax = x+y (t1)
rax = x+y+z (t2)
rdx = 3y
rdx = 48y (t4)

multiplying two variables

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

Arithmetic Example

```
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:
  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
```

comment (AT&T syntax)

S ∈ {1,2,4,8}

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