

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

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

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

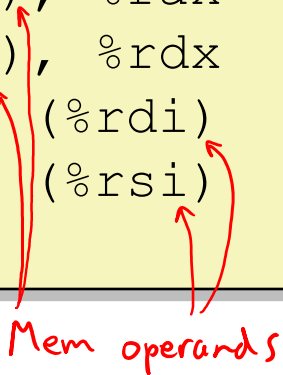
```
y += x;
y *= 3;
long r = y;
return r;
```

```
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
```



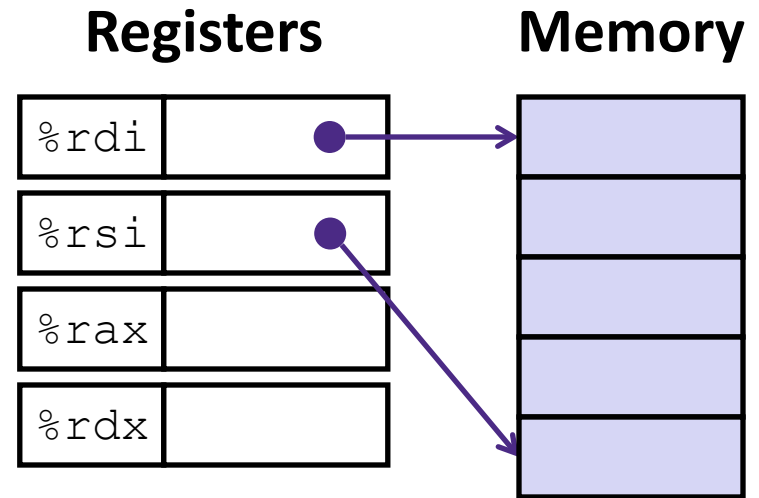
Mem operands

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

swap () walkthrough (1 of 5)

Registers

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

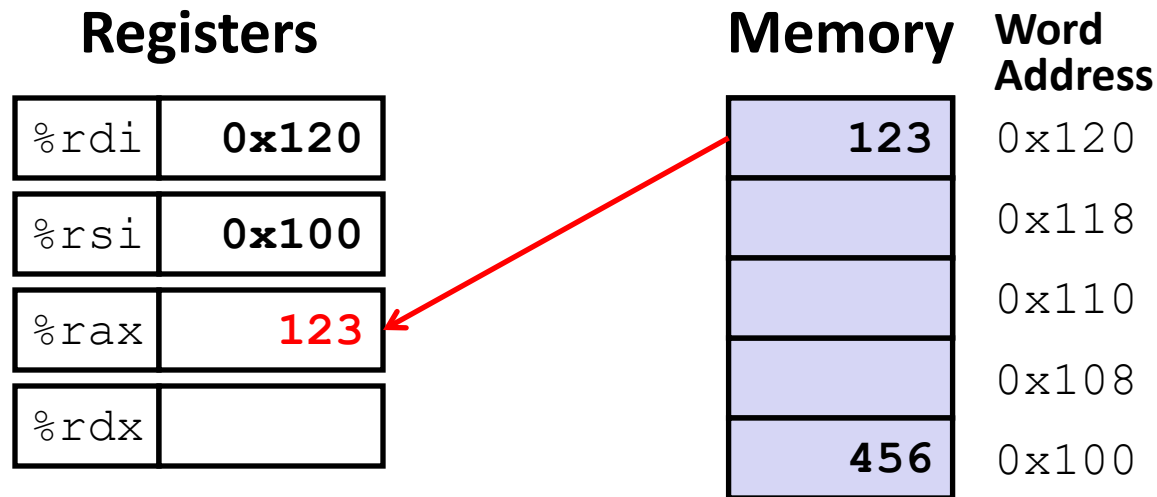
Memory

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
```

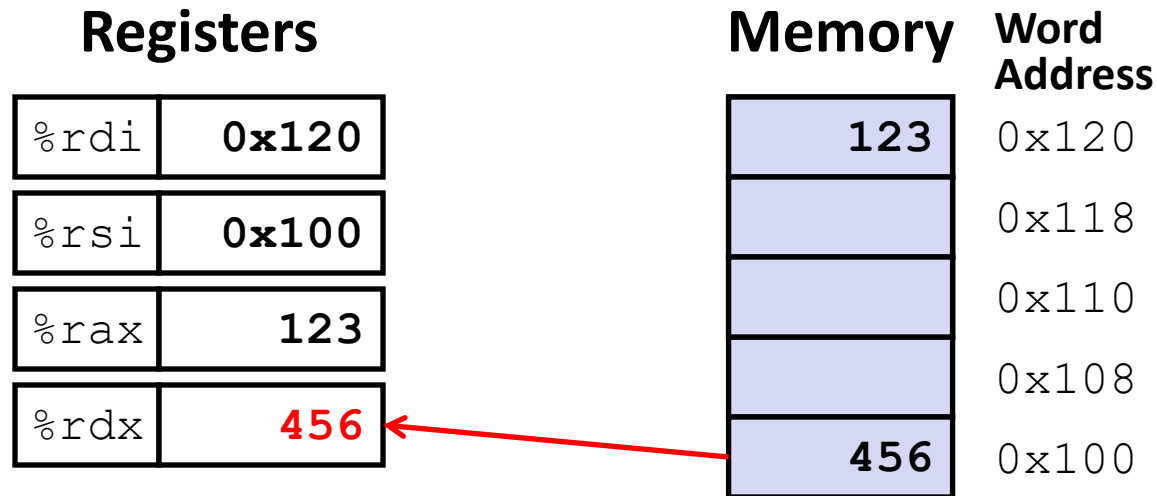
swap () walkthrough (2 of 5)



swap:

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

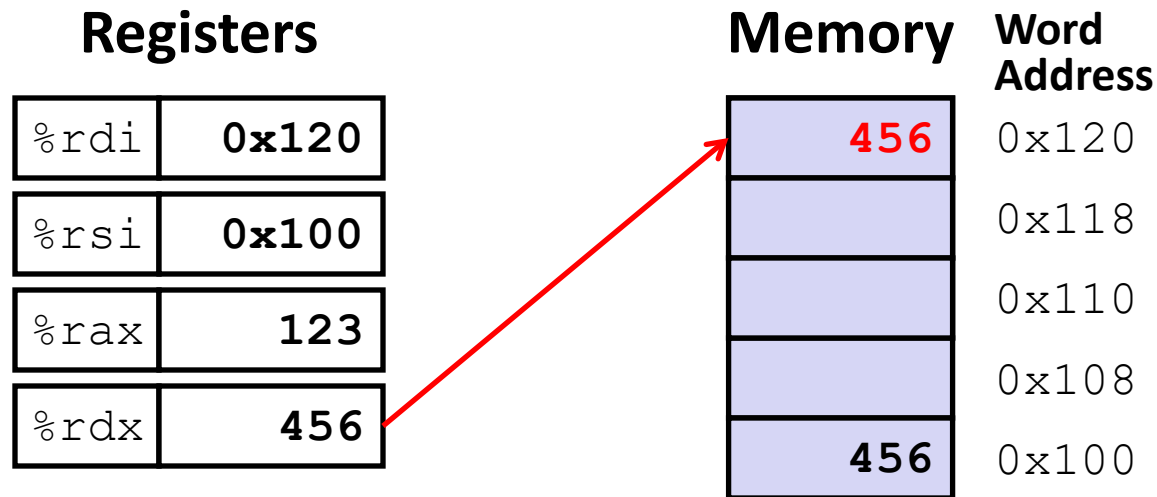
swap () walkthrough (3 of 5)



swap:

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

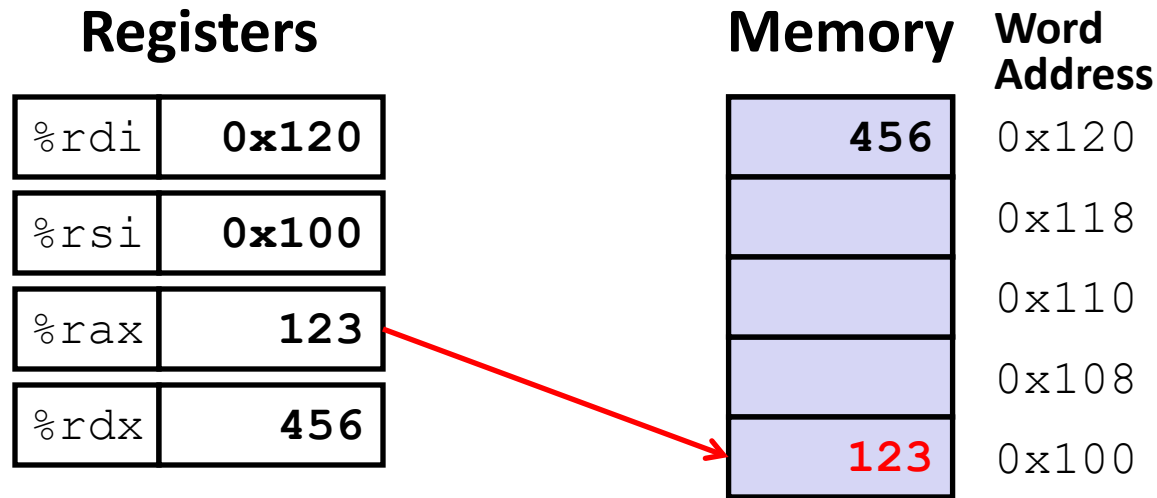
swap () walkthrough (4 of 5)



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) $\text{Mem}[\text{Reg}[R]]$

- Data in register R specifies the memory address
- Like pointer dereference in C
- Example: `movq (%rcx), %rax`

❖ Displacement: $D(R)$ $\text{Mem}[\text{Reg}[R]+D]$

- 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:

- $D(Rb, Ri, S) \quad \text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri] * S + D]$
 - 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)

- $D(Rb, Ri) \quad \text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri] + D] \quad (S=1)$
- $(Rb, Ri, S) \quad \text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri] * S] \quad (D=0)$
- $(Rb, Ri) \quad \text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri]] \quad (S=1, D=0)$
- $(, Ri, S) \quad \text{Mem}[\text{Reg}[Ri] * S] \quad (Rb=0, D=0)$

Address Computation Examples

default values:

$S = 1$

$D = 0$

$\text{Reg}[Rb] = 0$

$\text{Reg}[Ri] = 0$

<code>%rdx</code>	<code>0xf000</code>
<code>%rcx</code>	<code>0x0100</code>

$D(Rb, Ri, S) \rightarrow$

$\text{Mem}[\text{Reg}[Rb] + \text{Reg}[Ri] * S + D]$

↑ ignore the memory access for now

Expression	Address Computation	Address (8 bytes wide)
<code>0x8(%rdx)</code>		
<code>(%rdx,%rcx)</code>		
<code>(%rdx,%rcx,4)</code>		
<code>0x80(,%rdx,2)</code>		

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: $\%rax = 9 * \%rdi$

- A. `leaq (,%rdi,9), %rax`
- B. `movq (,%rdi,9), %rax`
- C. `leaq (%rdi,%rdi,8), %rax`
- D. `movq (%rdi,%rdi,8), %rax`

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
```

lea – “It just does math”

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;
}
```

```
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
```

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

❖ 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

```

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

```

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