x86-64 Programming I

CSE 351 Autumn 2017

Instructor:

Justin Hsia

Teaching Assistants:

Lucas Wotton

Michael Zhang

Parker DeWilde

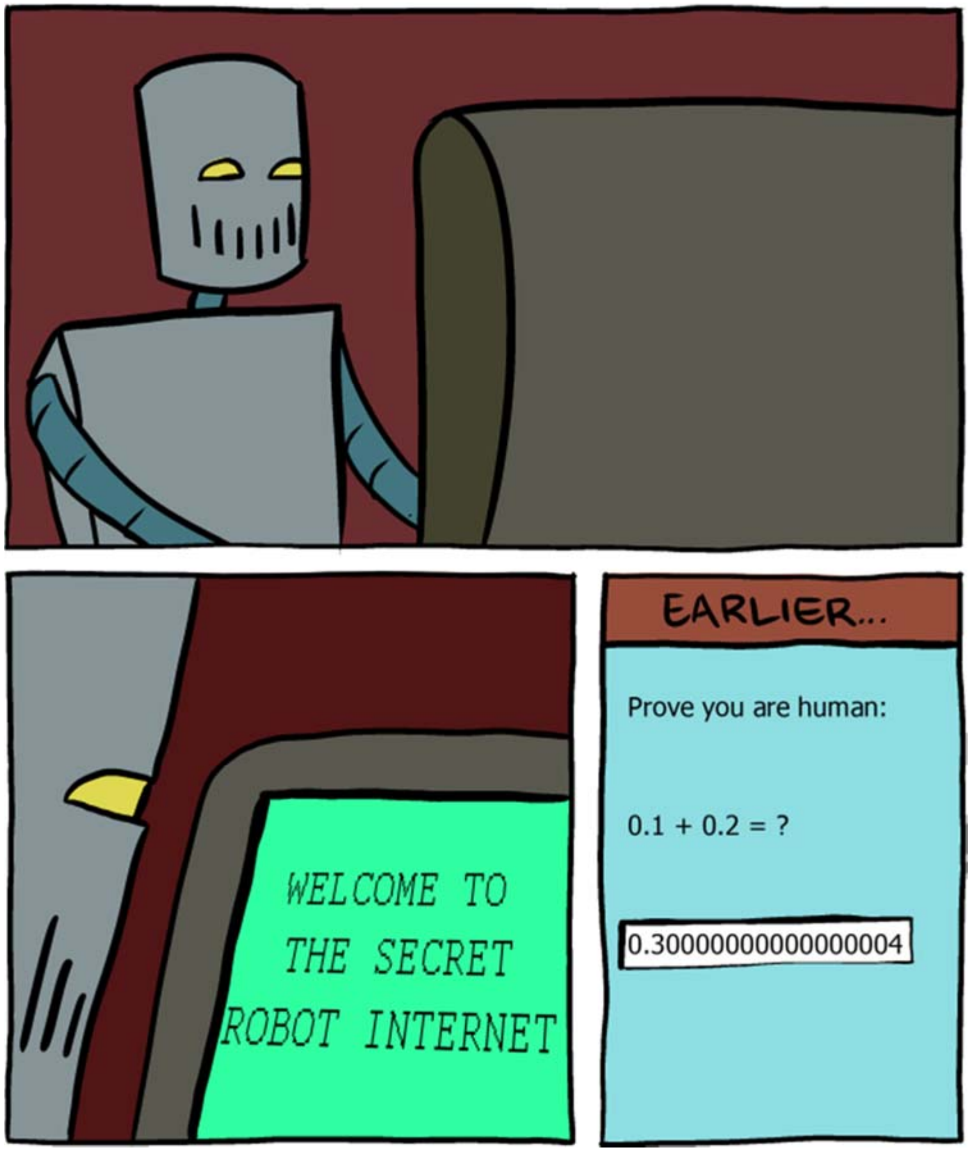
Ryan Wong

Sam Gehman

Sam Wolfson

Savanna Yee

Vinny Palaniappan



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

Administrivia

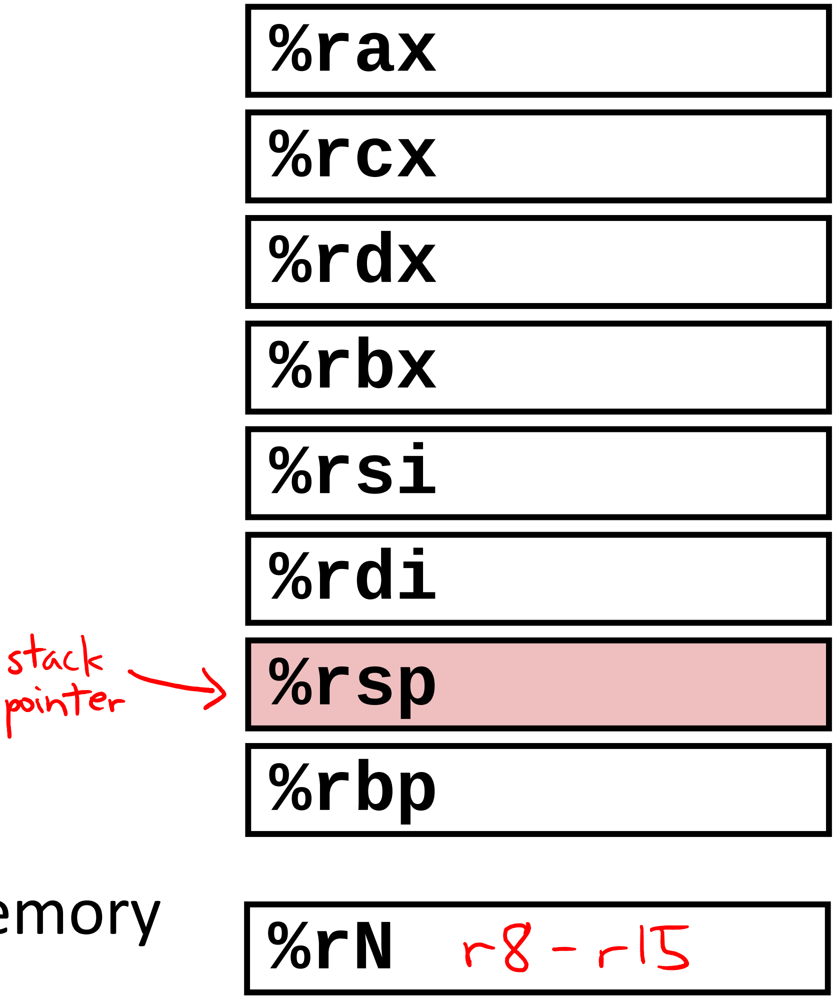
- Lab 1 due tonight at 11:59pm
  - You have *late days* available
- Homework 2 due next Friday (10/20)
- Lab 2 (x86-64) released on Tuesday (10/17)
  - Due on 10/27

Review: Operand types

- ***Immediate:*** Constant integer data
  - Examples: **`$0x400`**, **`$-533`**

    hex decimal
  - 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"

take data in %rax, treat as address, pull data at that address



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

x86 C
Imm ⟷ Constant
Reg ⟷ Variable
Mem ⟷ dereferencing a pointer

movq

Source	Dest	Src, Dest	C Analog
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?

① Mem → Reg movq (%rax), %rdx
② Reg → Mem movq %rdx, (%rbx)

x86-64 Introduction

- Arithmetic operations
- Memory addressing modes
  - `swap` example
- Address computation instruction (`lea`)

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`"?

Imm, Reg, or Mem

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*	

operation

operand size specifier (b,w,l,q)

① movq r2,r3 #r3=r2

② addq r1,r3 #r3=r2+r1

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

convention!

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

don't actually need new variables!

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

must return in %rax

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

return

Example of Basic Addressing Modes

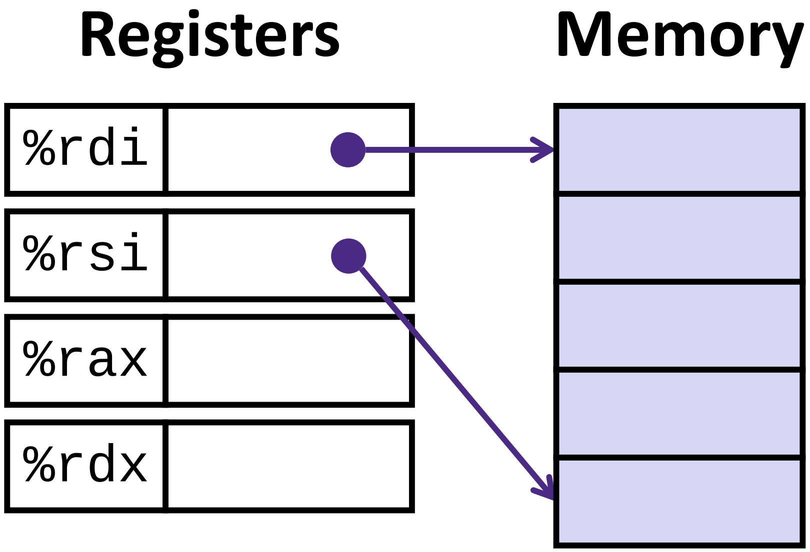
```
void swap(long *xp, long *yp)
{
   long t0 = *xp;
   long t1 = *yp;
   *xp = t1;
   *yp = t0;
}
```

src , dst (AT&T syntax)

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

Understanding `swap()`

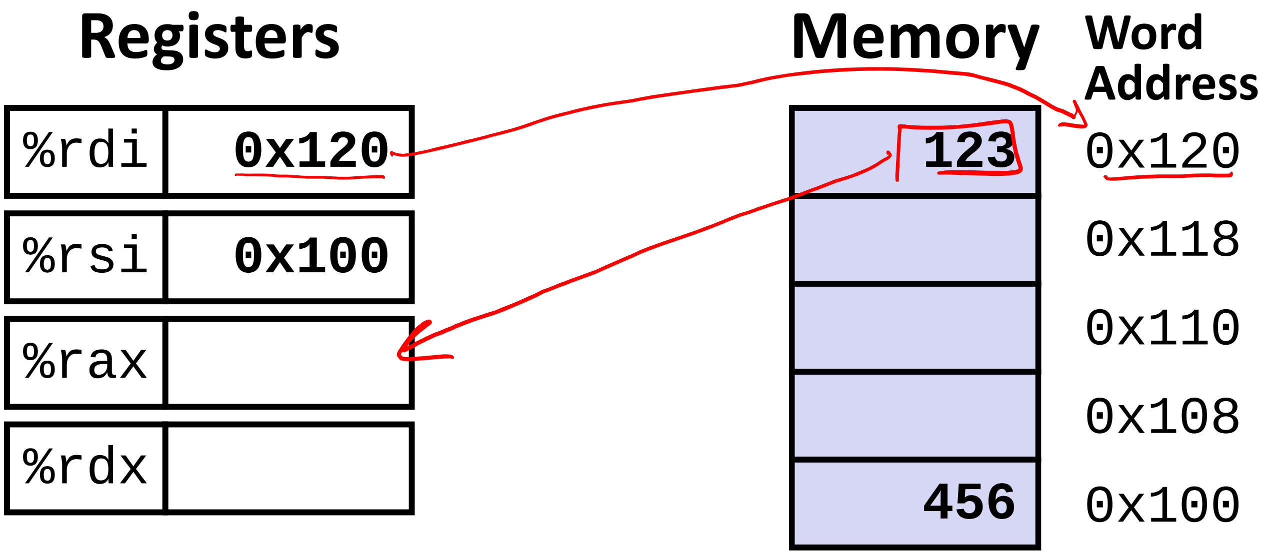
```
void swap(long *xp, long *yp)
{
   long t0 = *xp;
   long t1 = *yp;
   *xp = t1;
   *yp = t0;
}
```



```
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()`

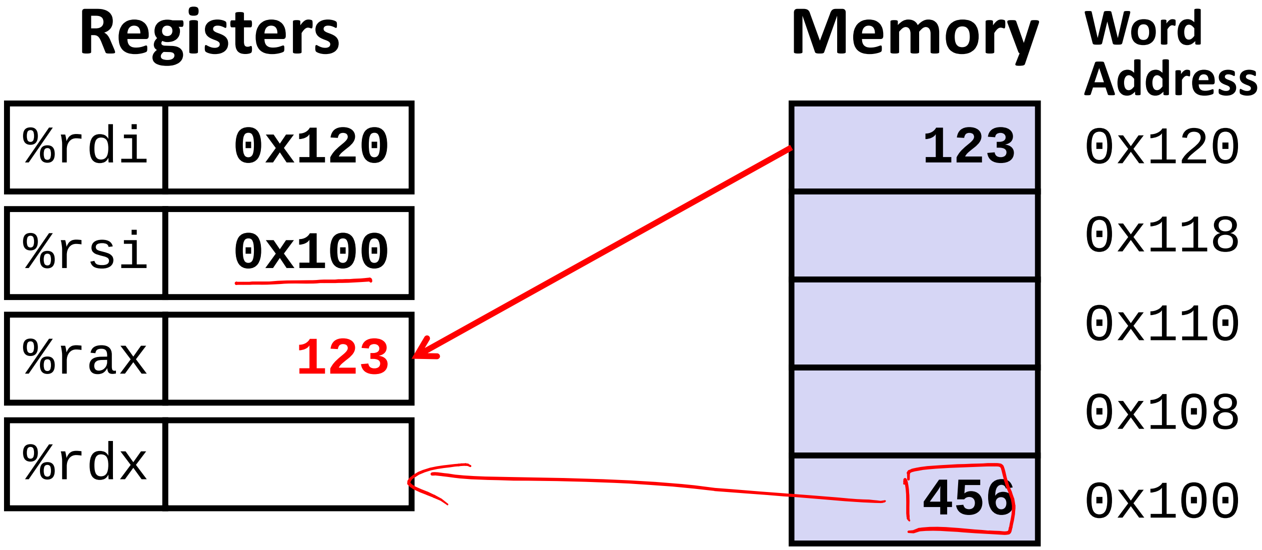


src dst

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

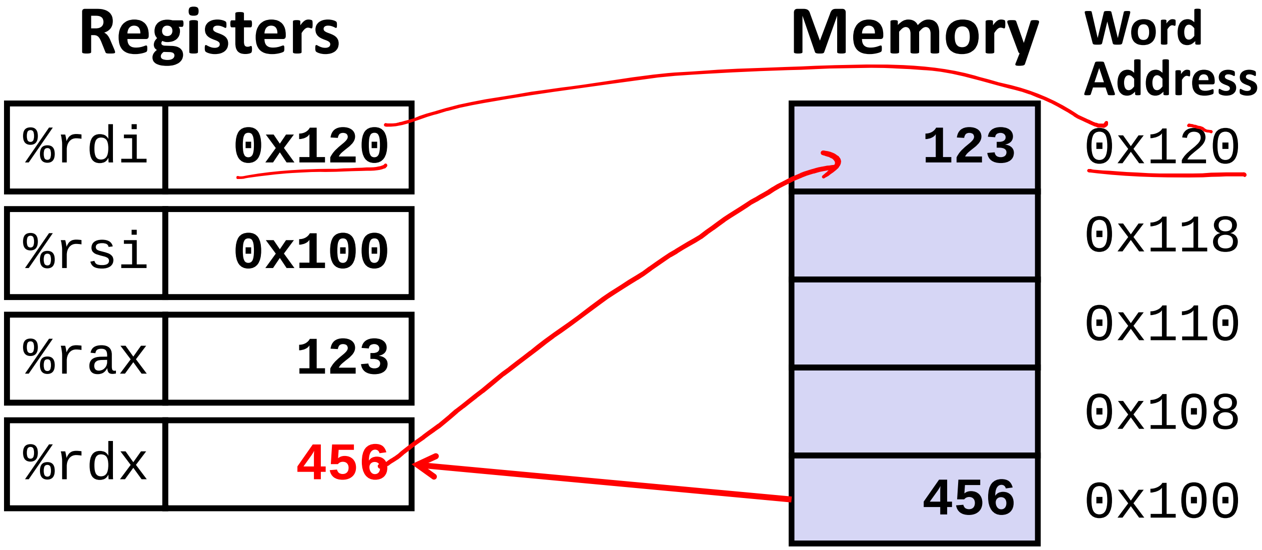
comment

Understanding swap()



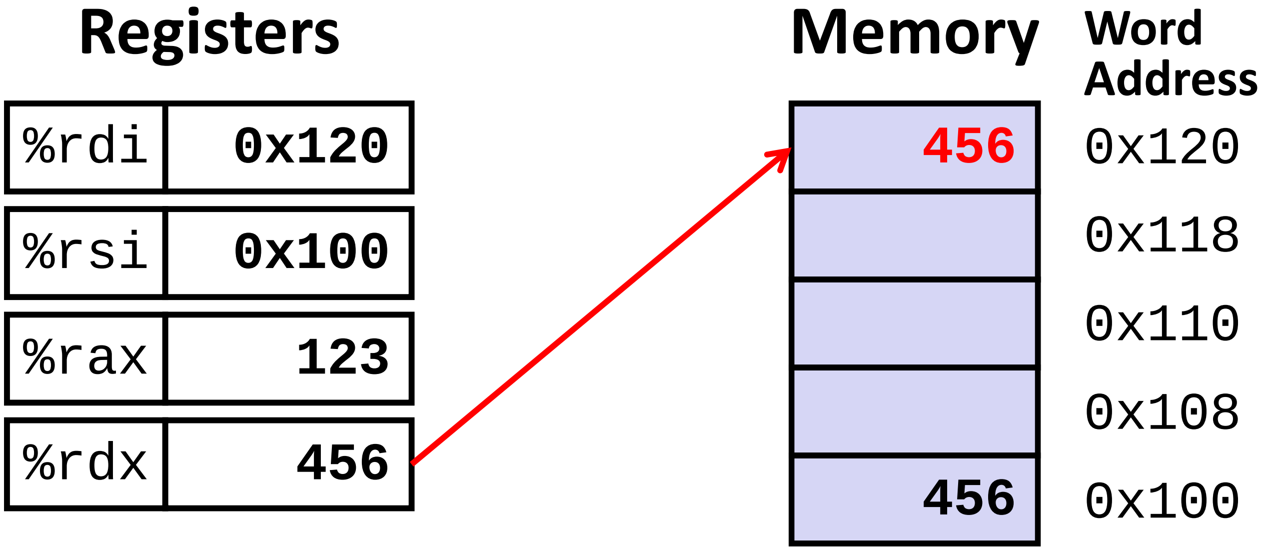
```
swap:
   movq  (%rdi), %rax  #  t0 = *xp
   movq  (%rsi), %rdx  #  t1 = *yp
   movq  %rdx, (%rdi)  # *xp =  t1
   movq  %rax, (%rsi)  # *yp =  t0
   ret
```

Understanding `swap()`



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

Understanding `swap()`



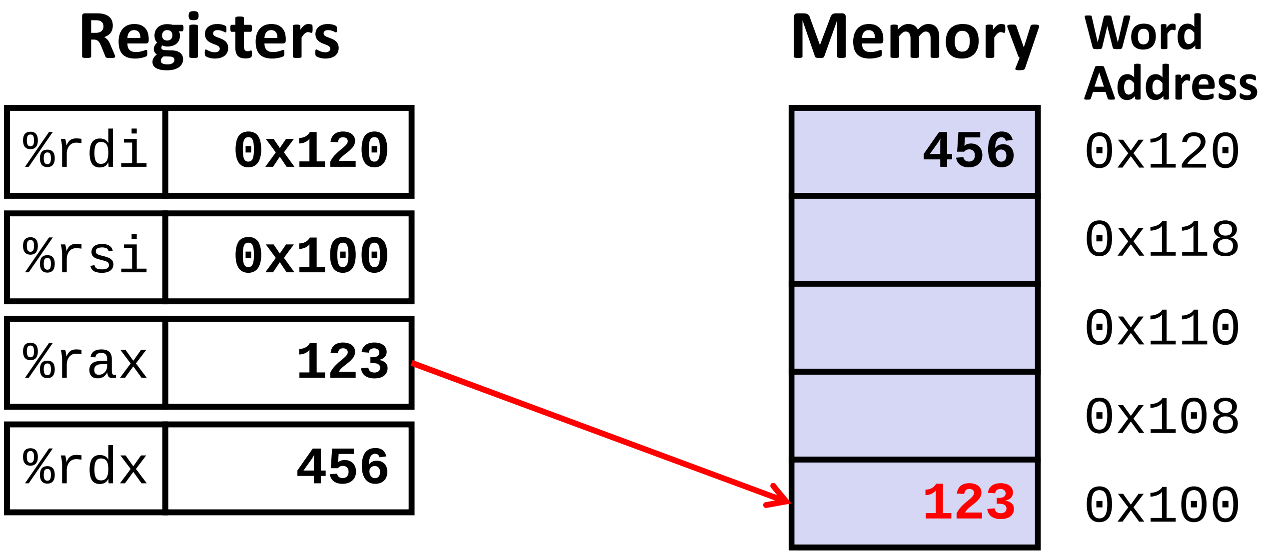
Registers

%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

name of register

treat Mem as an array

- **Indirect:** (R) Mem[Reg[R]]

value stored in register

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

no space

- **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

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?*
    - 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 Examples

(if not specified) 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]

Expression	Address Computation	Address
$\overset{D}{\texttt{0x8}}(\overset{Rb}{\texttt{\%rdx}})$	Reg[Rb]+D = 0xf000 + 0x8	0xf008
$(\overset{Rb}{\texttt{\%rdx}},\overset{Ri}{\texttt{\%rcx}})$	Reg[Rb]+Reg[Ri]*1	0xf100
$(\overset{Rb}{\texttt{\%rdx}},\overset{Ri}{\texttt{\%rcx}},\overset{S}{\texttt{4}})$	*4	0xf400
$\overset{D}{\texttt{0x80}}(,\overset{Ri}{\texttt{\%rdx}},\overset{S}{\texttt{2}})$	Reg[Ri]*2+0x80	0x1e080

0xf000*2

0xf000 << 1 = 0x1e000

1111 0000

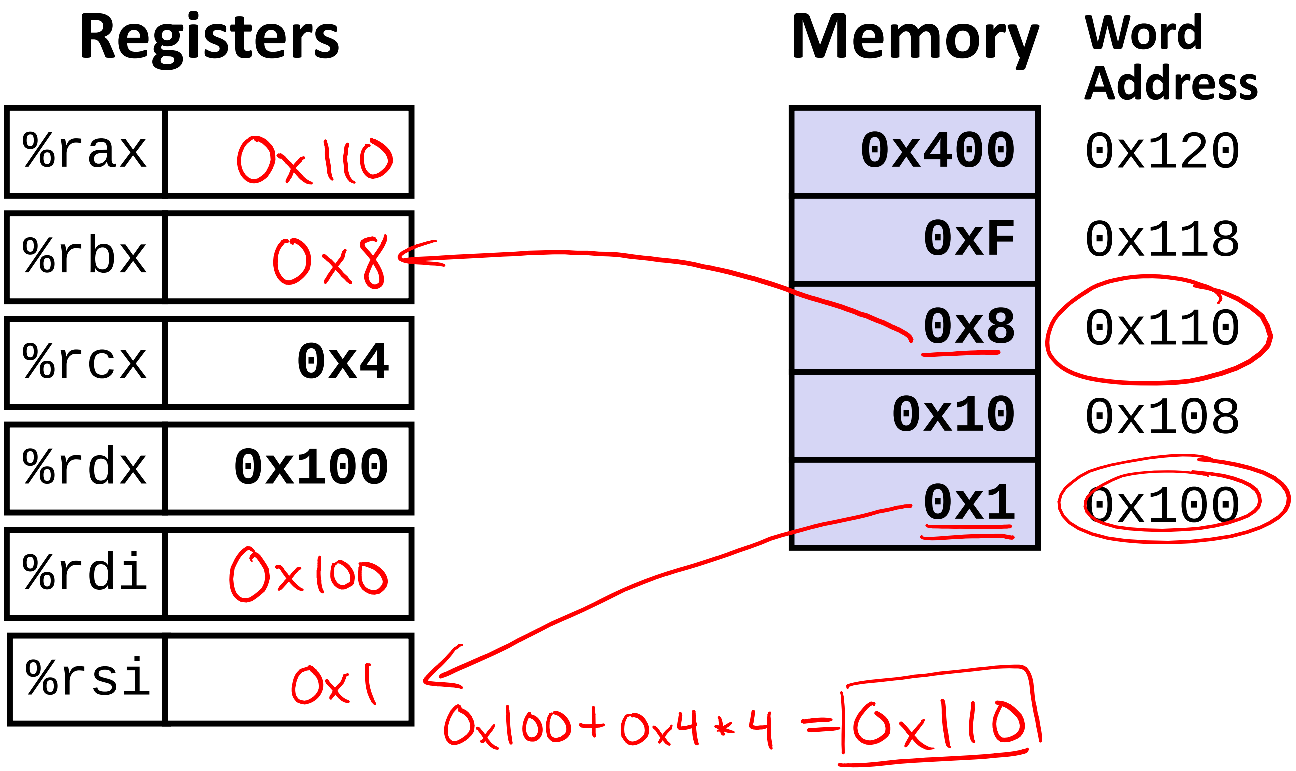
1 1110 0000...0

Address Computation Instruction

"Mem" Reg

- `leaq src, dst`
  - “`lea`” stands for *load effective address*
  - `src` is address expression (any of the formats we’ve seen)
  - `dst` is a register ↳ calculates Reg[Rb]+Reg[Ri]*S+D ~~Mem[ ]~~
  - 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`
    - Though k can only be 1, 2, 4, or 8

Example: `lea` vs. `mov`



Registers

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

Memory

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)

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

← multiplying two variables

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

Arithmetic Example

```c
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!

comment (AT&T syntax)

s ∈ {1,2,4,8}

```
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`?
  - Vote at http://PollEv.com/justinh

s ∈ {1,2,4,8}

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

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

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

D. `movq (%rdi,%rdi,8), %rax` %rax = *(9 * %rdi)

E. We're lost…

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
- `lea` is address calculation instruction
    - Does NOT actually go to memory
    - Used to compute addresses or some arithmetic expressions