x86 Programming I

CSE 351 Autumn 2016

Instructor:

Justin Hsia

Teaching Assistants:

Chris Ma

Hunter Zahn

John Kaltenbach

Kevin Bi

Sachin Mehta

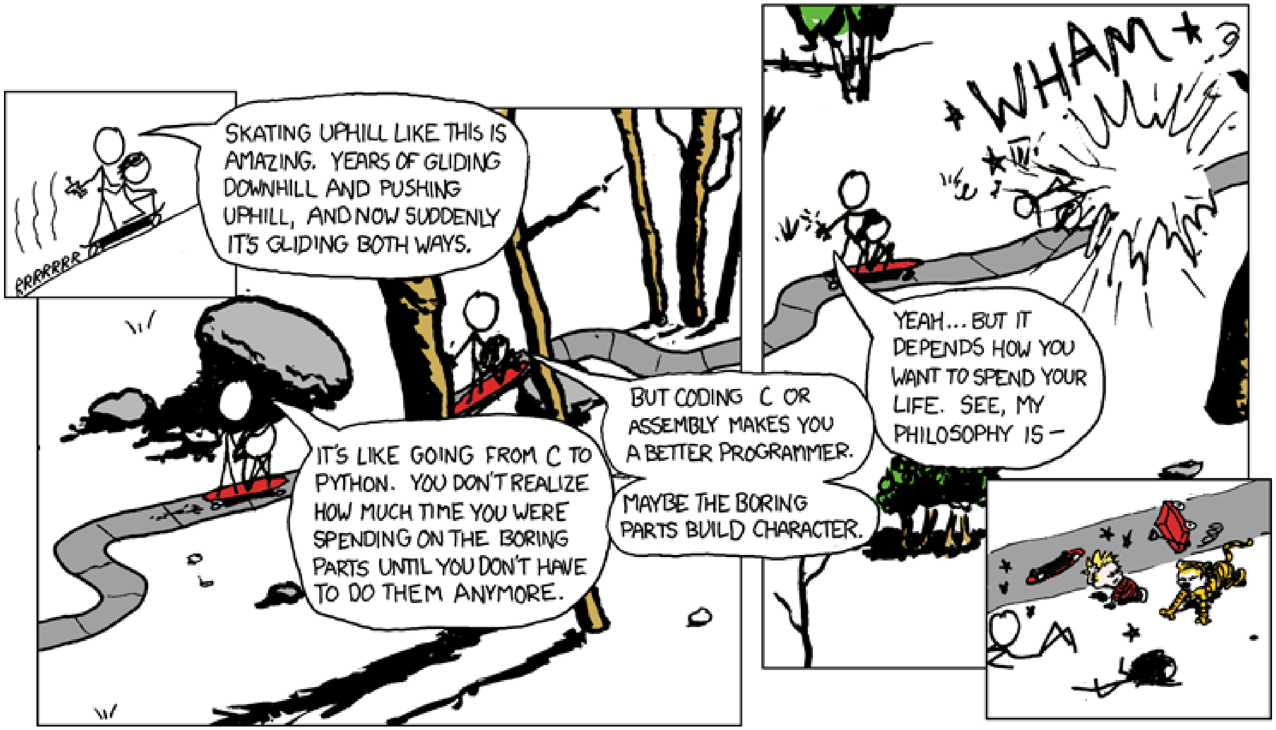
Suraj Bhat

Thomas Neuman

Waylon Huang

Xi Liu

Yufang Sun



http://xkcd.com/409/

Administrivia

- Lab 1 due today at 5pm
  - You have *late days* available
- Lab 2 (x86 assembly) released next Tuesday (10/18)
- Homework 1 due next Friday (10/21)

Roadmap

C:

```
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
    c.getMPG();
```

Assembly language:

```
get_mpg:
    pushq   %rbp
    movq    %rsp, %rbp
    ...
    popq    %rbp
    ret
```

Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

OS:

Windows 8 Mac

Computer system:

Memory & data
Integers & floats
Machine code & C
x86 assembly
Procedures & stacks
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C

x86 Topics for Today

- Registers
- Move instructions and operands
- Arithmetic operations
- Memory addressing modes
- `swap` example

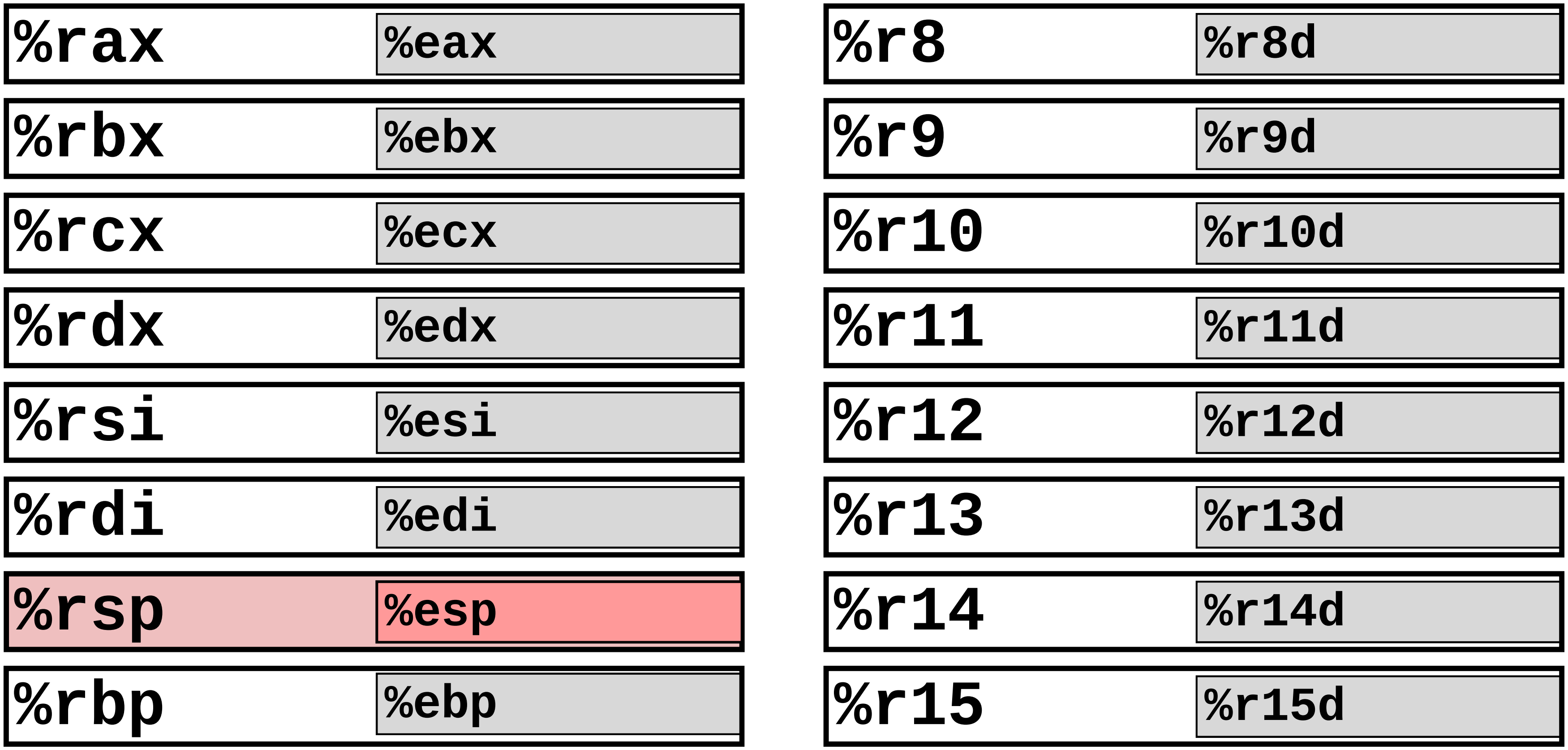
What is a Register?

- A location in the CPU that stores a small amount of data, which can be accessed very quickly (once every clock cycle)

- Registers have *names*, not *addresses*
  - In assembly, they start with % (e.g., `%rsi`)

- Registers are at the heart of assembly programming
  - They are a precious commodity in all architectures, but *especially* x86

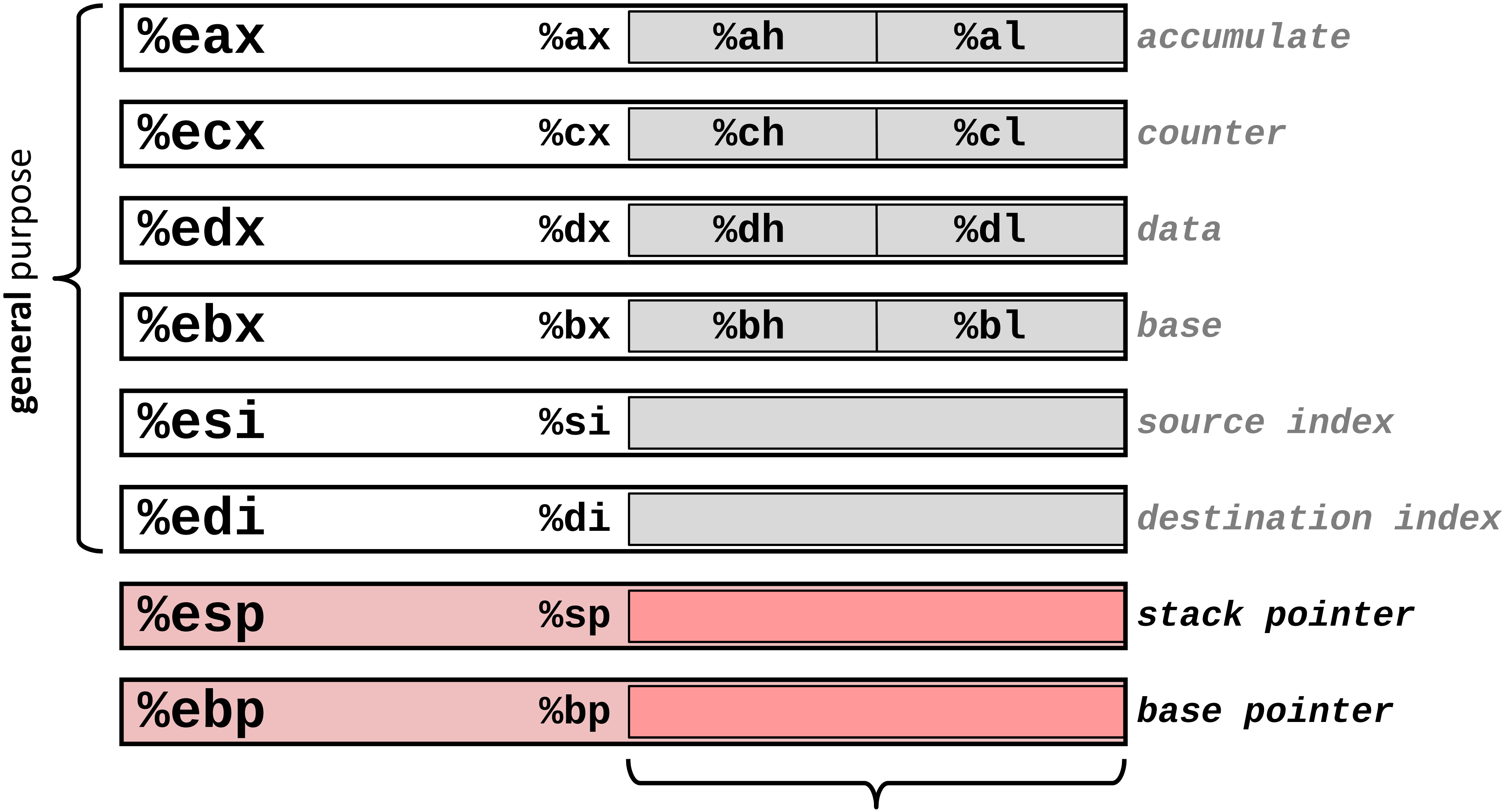
x86-64 Integer Registers – 64 bits wide



%rax	%eax	%r8	%r8d
%rbx	%ebx	%r9	%r9d
%rcx	%ecx	%r10	%r10d
%rdx	%edx	%r11	%r11d
%rsi	%esi	%r12	%r12d
%rdi	%edi	%r13	%r13d
%rsp	%esp	%r14	%r14d
%rbp	%ebp	%r15	%r15d

- Can reference low-order 4 bytes (also low-order 2 & 1 bytes)

Some History: IA32 Registers – 32 bits wide



general purpose

%eax	%ax	%ah	%al	*accumulate*
%ecx	%cx	%ch	%cl	*counter*
%edx	%dx	%dh	%dl	*data*
%ebx	%bx	%bh	%bl	*base*
%esi	%si			*source index*
%edi	%di			*destination index*
%esp	%sp			***stack pointer***
%ebp	%bp			***base pointer***

16-bit virtual registers
(backwards compatibility)

Name Origin
(mostly obsolete)

x86-64 Assembly Data Types

- "Integer" data of 1, 2, 4, or 8 bytes
    - Data values
    - Addresses (untyped pointers)
- Floating point data of 4, 8, 10 or 2x8 or 4x4 or 8x2
    - Different registers for those (e.g. `%xmm1`, `%ymm2`)
    - Come from *extensions to x86* (SSE, AVX, …)
    - Probably won't have time to get into these ☹
- No aggregate types such as arrays or structures
    - Just contiguously allocated bytes in memory
- Two common syntaxes
    - "AT&T": used by our course, slides, textbook, gnu tools, …
    - "Intel": used by Intel documentation, Intel tools, …
    - Must know which you're reading

Three Basic Kinds of Instructions

1) Transfer data between memory and register
   - *Load* data from memory into register
     - `%reg` = Mem[address]
   - *Store* register data into memory
     - Mem[address] = `%reg`

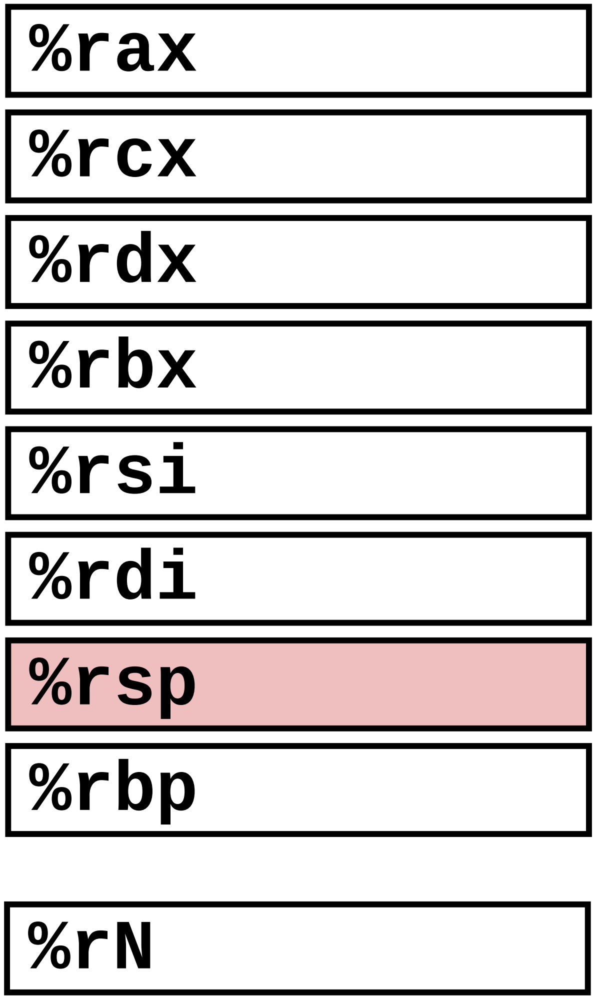
Remember: Memory is indexed just like an array of bytes!

2) Perform arithmetic operation on register or memory data
   - `c = a + b;`   `z = x << y;`   `i = h & g;`

3) Control flow: what instruction to execute next
   - Unconditional jumps to/from procedures
   - Conditional branches

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?

Memory vs. Registers

Addresses	**vs.**	Names
`0x7FFFD024C3DC`		`%rdi`
Big	**vs.**	Small
~ 8 GiB		(16 x 8 B) = 128 B
Slow	**vs.**	Fast
~50-100 ns		sub-nanosecond timescale
Dynamic	**vs.**	Static
Can “grow” as needed while program runs		fixed number in hardware

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

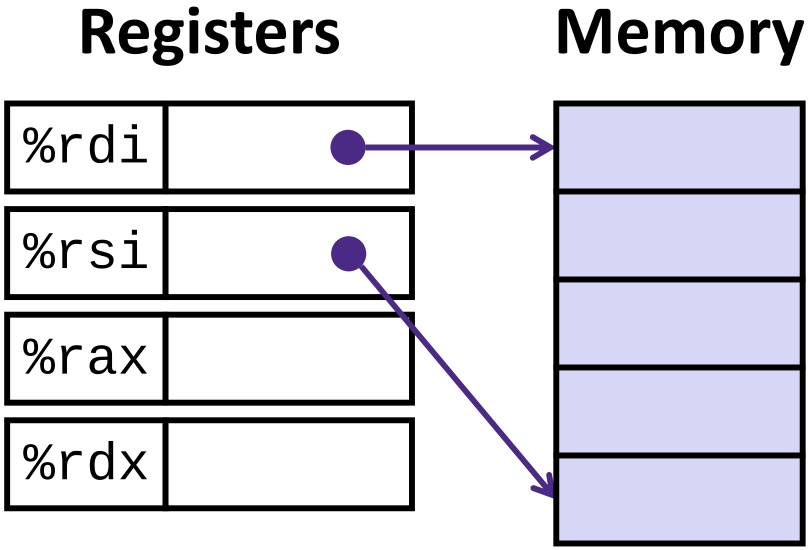
```
void swap(long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

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

Understanding swap()

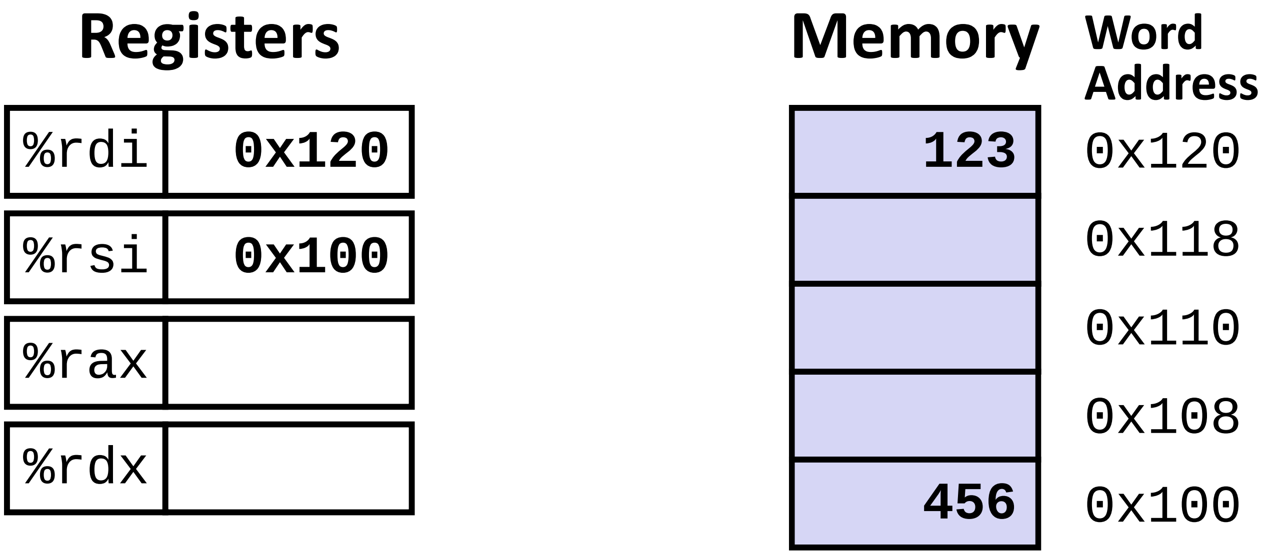
```
void swap(long *xp, long *yp)
{
   long t0 = *xp;
   long t1 = *yp;
   *xp = t1;
   *yp = t0;
}
```

```
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()`



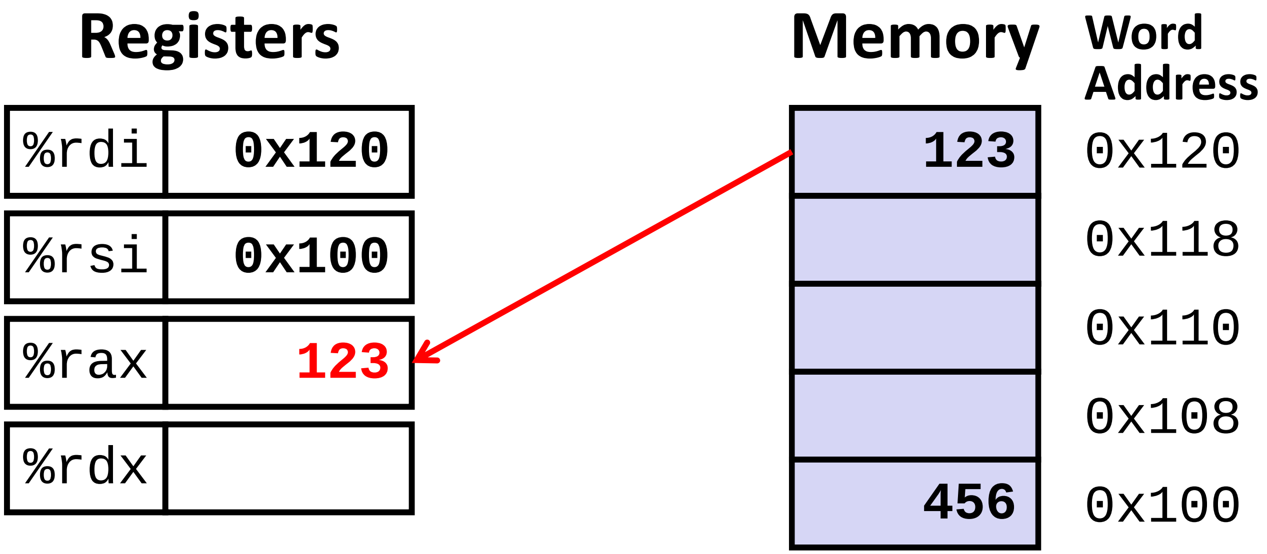
Registers

%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
```

Understanding `swap()`



Registers

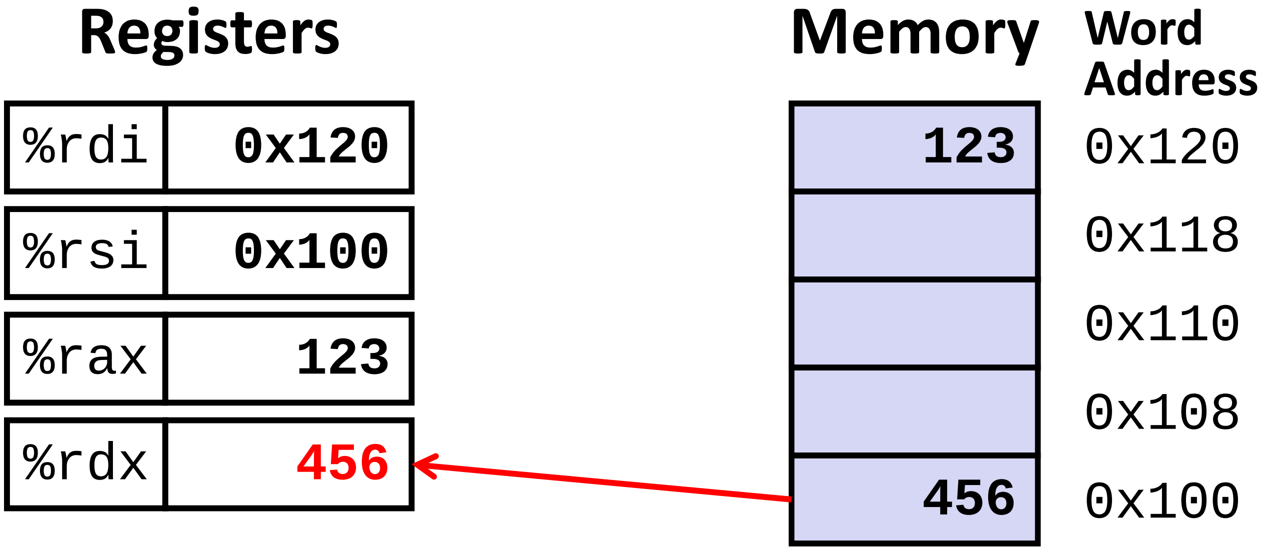
%rdi	**0x120**
%rsi	**0x100**
%rax	**123**
%rdx	

Memory

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

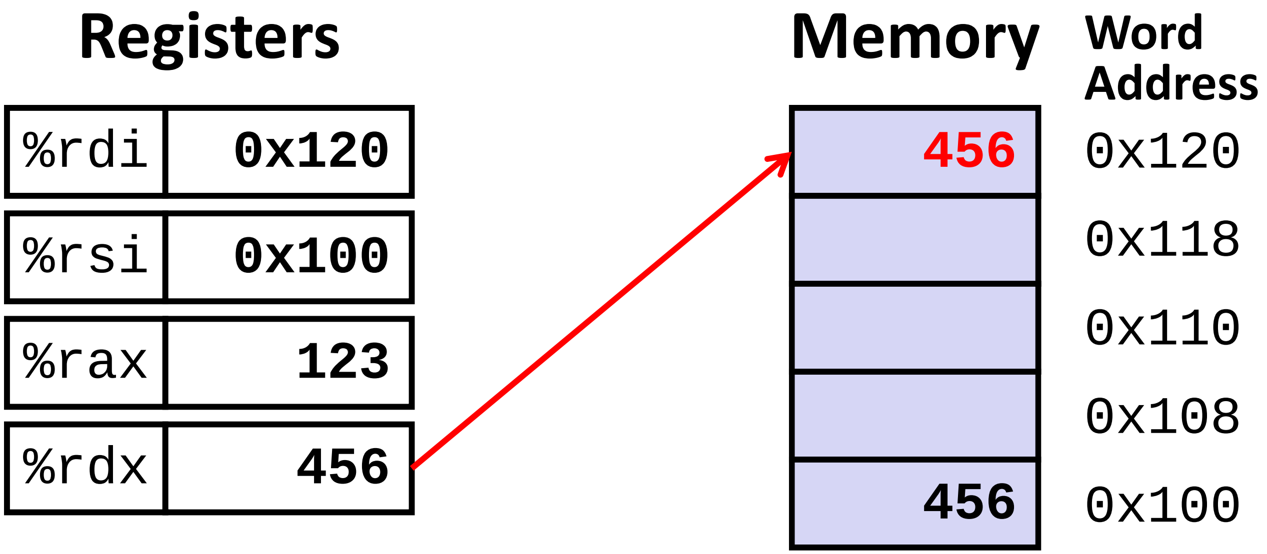
```
swap:
   movq  (%rdi), %rax  #  t0 = *xp
   movq  (%rsi), %rdx  #  t1 = *yp
   movq  %rdx, (%rdi)  # *xp =  t1
   movq  %rax, (%rsi)  # *yp =  t0
   ret
```

Understanding swap()



```
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

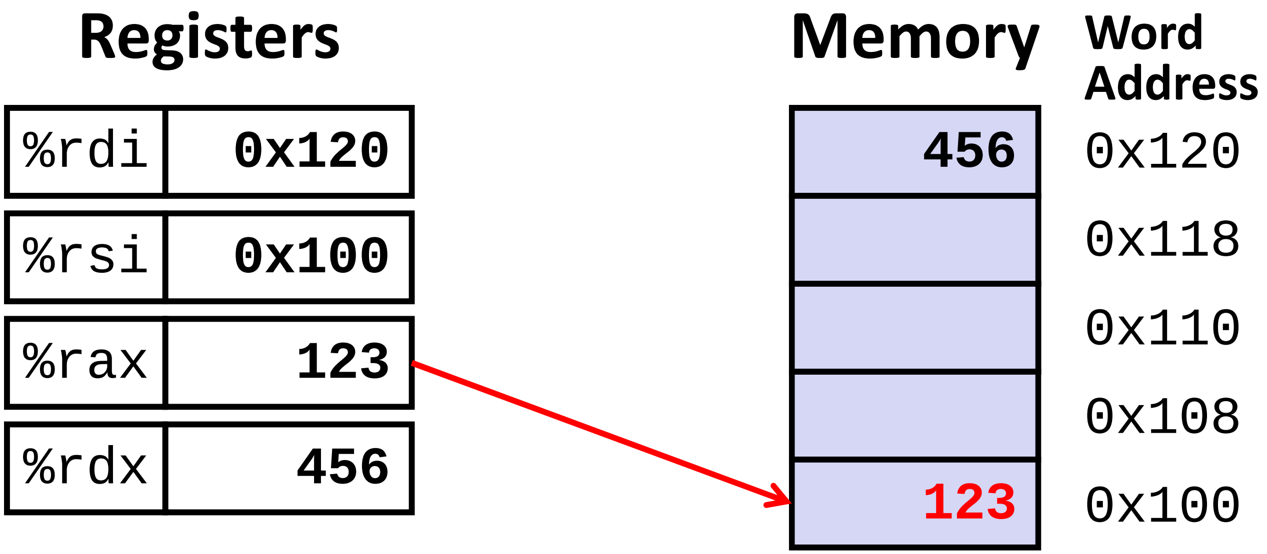
%rdi	**0x120**
%rsi	**0x100**
%rax	**123**
%rdx	**456**

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**

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)`		

Peer Instruction Question

- Which of the following statements is TRUE?
  - Vote at http://PollEv.com/justinh

(A) The program counter (%rip) is a register that we manually manipulate

(B) There is only one way to compile a C program into assembly

(C) Mem to Mem (src to dst) is the only disallowed operand combination

(D) We can compute an address without using any registers

Summary

- **Registers** are named locations in the CPU for holding and manipulating data
  - x86-64 uses 16 64-bit wide registers
- Assembly instructions have rigid form
  - Operands include immediates, registers, and data at specified memory locations
  - Many instruction variants based on size of data
- **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