Machine-Level Programming: Operations

CSCI 237: Computer Organization
8th Lecture, Feb 26, 2025

Jeannie Albrecht

Administrative Details

- Lab 1 due yesterday/today
  - I hope you found the puzzles challenging but fun
  - We will grade your **last** submission (submitted using submit237)
- Lab 2 is posted
  - Try to read it before lab if possible
  - Lab 2 emphasizes understanding (but not writing!) x86-64 and gdb
- Glow HW 2 due Friday
  - Covers material from Ch 2

Last Time

- Floating point wrapup
- History of Intel processors and architectures
- Assembly Basics: Registers, operands, move
- Arithmetic & logical operations
- C, assembly, machine code
- Intro to data-dependent control

Today:
Machine Programming: Ops & Control

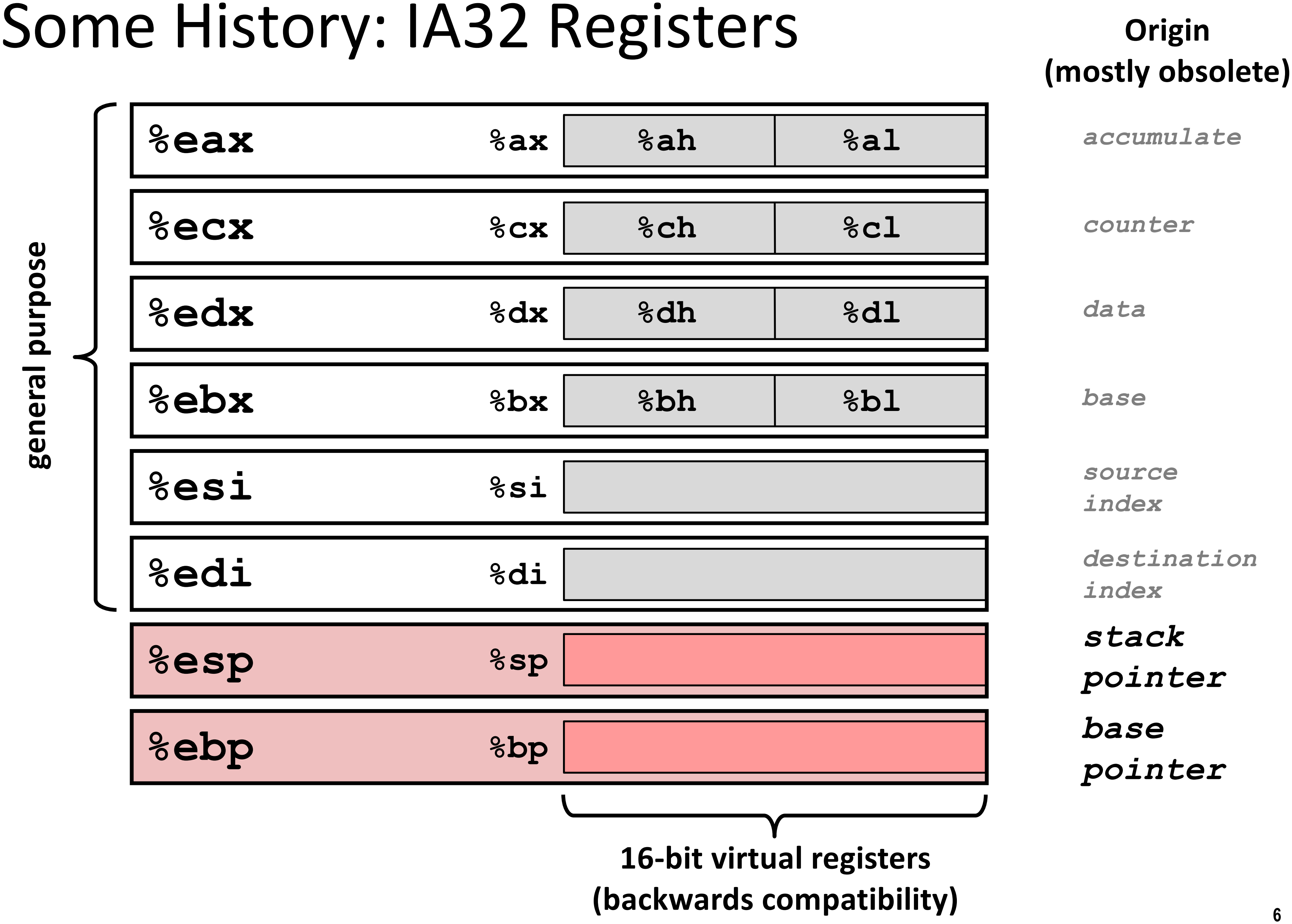
- History of Intel processors and architectures
- Assembly Basics: Registers, operands, move
- Arithmetic & logical operations
- C, assembly, machine code
- Intro to data-dependent control

Recap: x86-64 Integer Registers

`%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 1 & 2 bytes)
- Not part of memory (or cache); part of CPU

Some History: IA32 Registers

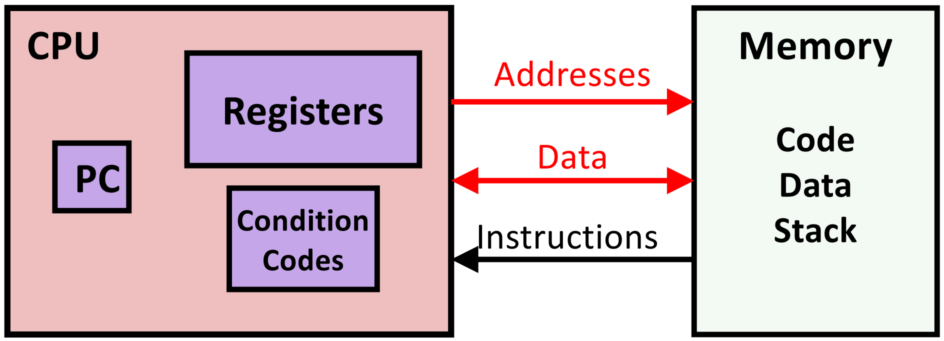


Assembly Characteristics: Types of Operations

- Transfer data between memory and register
  - Load data from memory into register
  - Store register data into memory
  - Data stored in registers is much faster to access than memory

- Perform arithmetic functions in CPU on register or memory data

- Transfer control
  - Unconditional jumps to/from procedures
  - Conditional branches



Moving Data (Ch 3.4)

- Moving data

  `movq` ***Source, Dest***

- Operand types for `source` and `dest`
  - ***Immediate:*** Constant integer data
    - Example: **`$0x400, $-533`**
    - Like C constants, but prefixed with **`'$'`**
    - Encoded with 1, 2, or 4 bytes
  - ***Register:*** One of 16 integer registers
    - Example: **`%rax, %r13`**
    - Note: **`%rsp`** reserved for special use
    - Others have special uses for particular instructions
  - ***Memory:*** 8 consecutive bytes of memory at ***address*** given by register
    - Simplest example: **`(%rax)`**
    - Various other "addressing modes"
    - Note the parentheses!

`%rax`
`%rcx`
`%rdx`
`%rbx`
`%rsi`
`%rdi`
`%rsp`
`%rbp`
`%rN`

Warning: Intel docs use mov *Dest, Source* 😩

Moving Data (Ch 3.4)

- Moving data

  `movq` ***Source, Dest***

Q = quad word (8 bytes)
L = double word (4 bytes)
W = word (2 bytes – historical!)
B = byte (1 byte)

- Operand types for s
  - ***Immediate:*** Constant integer data
    - Example: **`$0x400, $-533`**
    - Like C constants, but prefixed with **`'$'`**
    - Encoded with 1, 2, or 4 bytes
  - ***Register:*** One of 16 integer registers
    - Example: **`%rax, %r13`**
    - Note: **`%rsp`** reserved for special use
    - Others have special uses for particular instructions
  - ***Memory:*** 8 consecutive bytes of memory at ***address*** given by register
    - Simplest example: **`(%rax)`**
    - Various other “addressing modes”
    - Note the parentheses!

```
%rax
%rcx
%rdx
%rbx
%rsi
%rdi
%rsp
%rbp

%rN
```

Warning: Intel docs use mov *Dest, Source* 😩

Pointer Recap

```
int *p;            //variable p is a pointer to an integer
int i;             // integer value
```

- You dereference a pointer to get value with *:

```
int i2 = *p;       // integer i2 is assigned the value that
                   // pointer p is pointing to
```

- You find address of value with &:

```
int *p2 = &i;      // pointer p2 will point to the memory
                   // address of i
```

- Worth spending a few minutes digesting the concept of pointers!
- Check K&R for more info

`movq` Operand Combinations

Source	Dest	From,To Src,Dest	C Analog
Imm	*Reg*	`movq $0x4,%rax`	`temp = 0x4;`
	Mem	`movq $-147,(%rax)`	`*p = -147;`

movq

Move 0x4 into %rax

Move -147 into memory at address specified in %rax (note the parentheses)

`movq` Operand Combinations

	Source	Dest	From,To Src,Dest	C Analog
`movq`	*Imm*	*Reg*	`movq $0x4,%rax`	`temp = 0x4;`
		Mem	`movq $-147,(%rax)`	`*p = -147;`
	Reg	*Reg*	`movq %rax,%rdx`	`temp2 = temp1;`
		Mem	`movq %rax,(%rdx)`	`*p = temp;`

Move value in %rax into into memory at address specified in %rdx (note the parentheses)

Move value in %rax into %rdx

`movq` Operand Combinations

Source	Dest	From,To Src,Dest	C Analog
Imm	*Reg*	`movq $0x4,%rax`	`temp = 0x4;`
Imm	*Mem*	`movq $-147,(%rax)`	`*p = -147;`
Reg	*Reg*	`movq %rax,%rdx`	`temp2 = temp1;`
Reg	*Mem*	`movq %rax,(%rdx)`	`*p = temp;`
Mem	*Reg*	`movq (%rax),%rdx`	`temp = *p;`

movq

Move value from memory address specified in %rax into %rdx (note the parentheses)

movq Operand Combinations

	Source	Dest	From,To Src,Dest	C Analog
movq	*Imm*	*Reg*	`movq $0x4,%rax`	`temp = 0x4;`
		Mem	`movq $-147,(%rax)`	`*p = -147;`
	Reg	*Reg*	`movq %rax,%rdx`	`temp2 = temp1;`
		Mem	`movq %rax,(%rdx)`	`*p = temp;`
	Mem	*Reg*	`movq (%rax),%rdx`	`temp = *p;`

NOTE: Cannot do memory-memory transfer with a single instruction!

Simple Memory Addressing Modes

- Normal (Reg[R]) → Mem[Reg[R]]
  - Register R specifies memory address, note the parentheses
  - Like pointer dereferencing in C

```
movq (%rcx), %rax
```

- Displacement D(Reg[R]) → Mem[Reg[R]+D]
  - Register R specifies start of memory region
  - Constant displacement D specifies offset

```
movq 8(%rbp),%rdx
```

Example of Simple Addressing Modes

```
void whatAmI(<type> a, <type> b){
   ????


}
```

%rdi %rsi

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

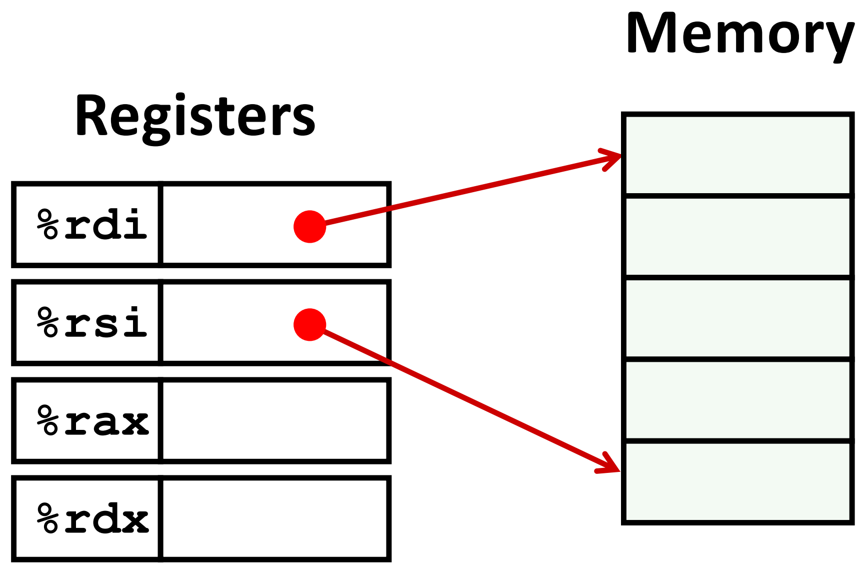
Example of Simple Addressing Modes

```
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;
}
```



Register	Value
`%rdi`	`xp`
`%rsi`	`yp`
`%rax`	`t0`
`%rdx`	`t1`

```
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

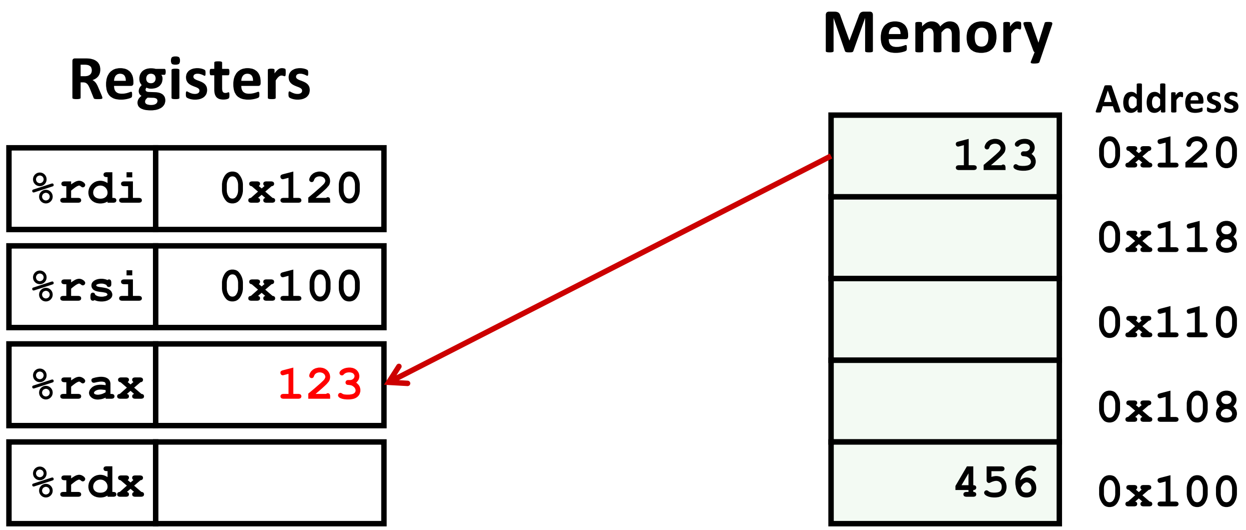
%rdi	0x120
%rsi	0x100
%rax	
%rdx	

Memory

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

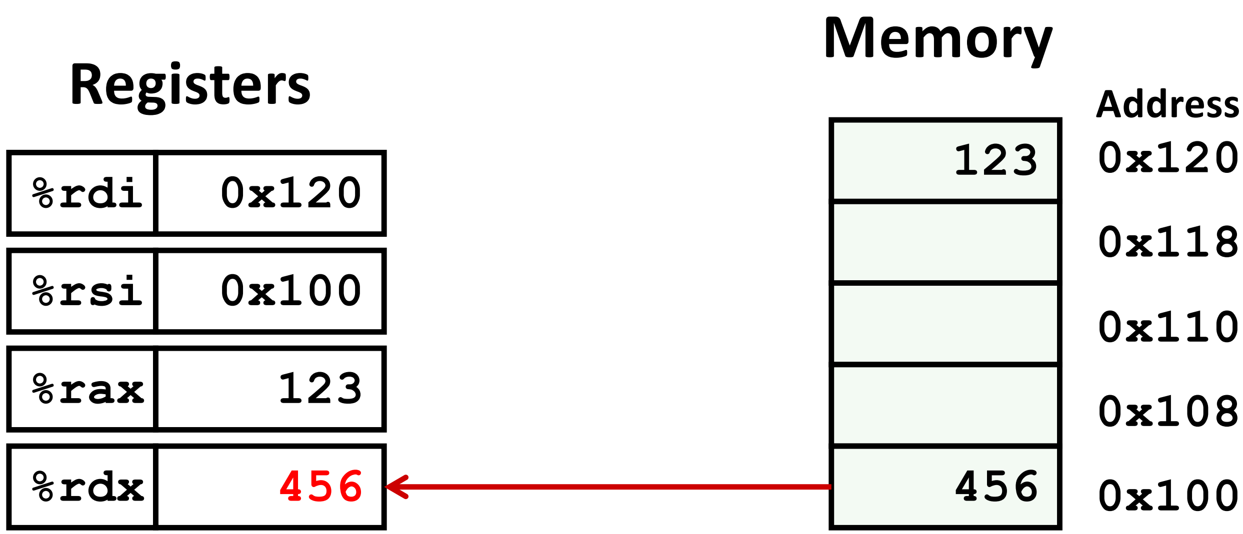
```
swap:
   movq    (%rdi), %rax   # t0 = *xp
   movq    (%rsi), %rdx   # t1 = *yp
   movq    %rdx, (%rdi)   # *xp = t1
   movq    %rax, (%rsi)   # *yp = t0
   ret
```

Understanding `swap()`



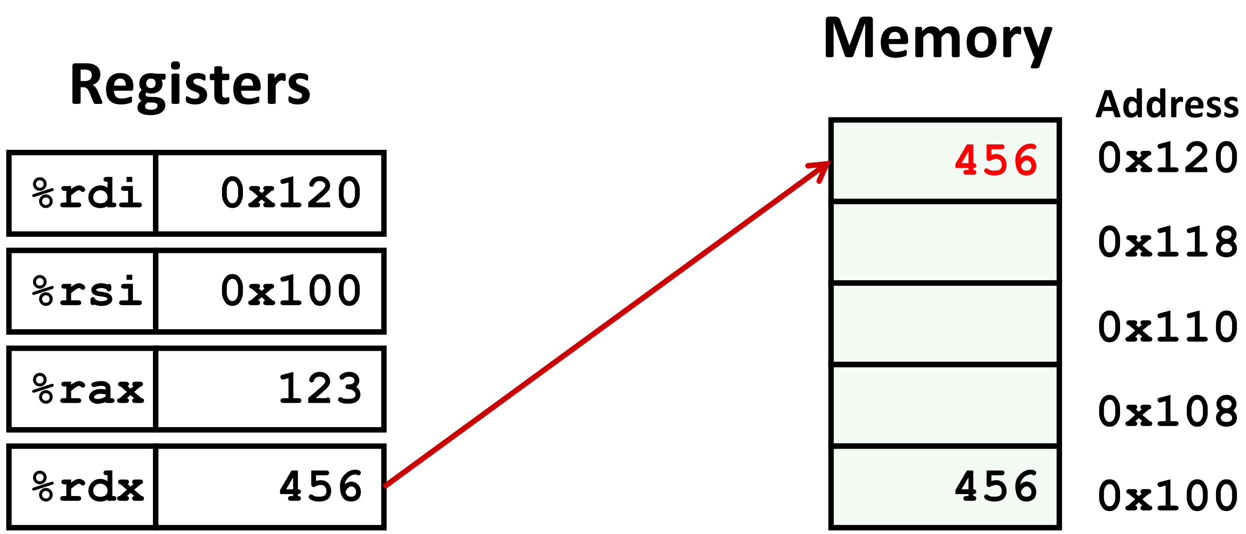

```
swap:
   movq    (%rdi), %rax   # t0 = *xp
   movq    (%rsi), %rdx   # t1 = *yp
   movq    %rdx, (%rdi)   # *xp = t1
   movq    %rax, (%rsi)   # *yp = t0
   ret
```

Understanding swap()



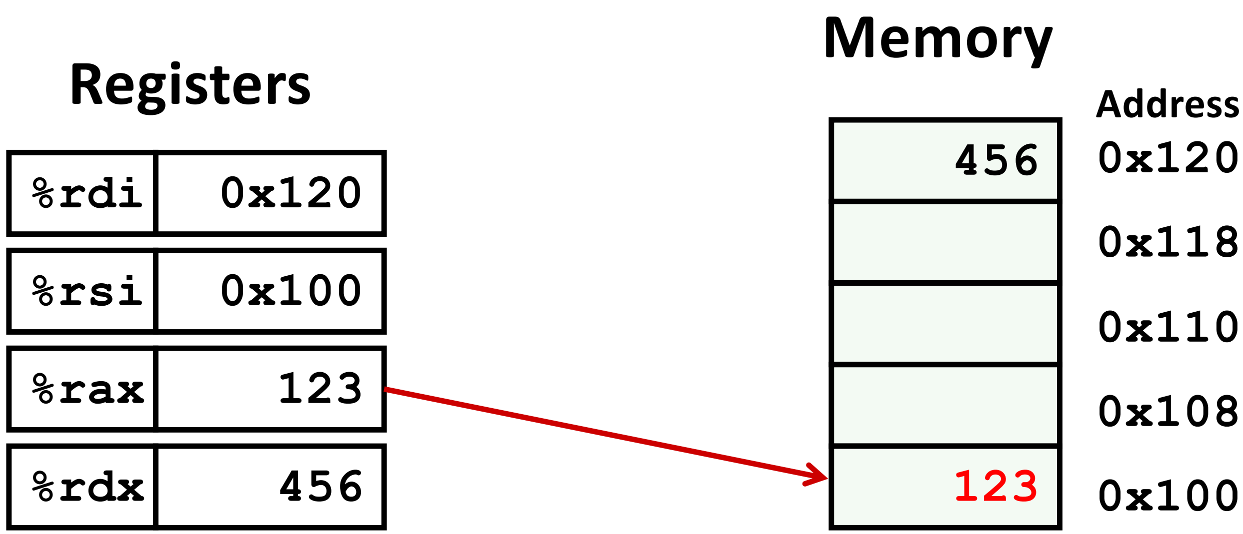

```
swap:
   movq     (%rdi), %rax   # t0 = *xp
   movq     (%rsi), %rdx   # t1 = *yp
   movq     %rdx, (%rdi)   # *xp = t1
   movq     %rax, (%rsi)   # *yp = t0
   ret
```

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()`




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

Simple Memory Addressing Modes

- Normal (Reg[R]) → Mem[Reg[R]]
  - Register R specifies memory address
  - Like pointer dereferencing in C

```
movq (%rcx), %rax
```

- Displacement D(Reg[R]) → Mem[Reg[R]+D]
  - Register R specifies start of memory region
  - Constant displacement D specifies offset

```
movq 8(%rbp), %rdx
```

Complete Memory Addressing Modes

- Most General Form

  D(Rb,Ri,S) → Mem[Reg[Rb]+S*Reg[Ri]+ D]

  - D: Constant “displacement”
  - Rb: Base register Reg[Rb]: Any of 16 integer registers
  - Ri: Index register Reg[Ri]: Any, except for **`%rsp`**
  - S: Scale: 1, 2, 4, or 8 (no other values allowed!)

- Special Cases

  (Rb,Ri) → Mem[Reg[Rb]+Reg[Ri]]

  D(Rb,Ri) → Mem[Reg[Rb]+Reg[Ri]+D]

  (Rb,Ri,S) → Mem[Reg[Rb]+S*Reg[Ri]]

Address Computation Examples

`%rdx`	`0xf000`
`%rcx`	`0x0100`

Most General Form

D(Rb,Ri,S) ⟶ Mem[Reg[Rb]+S*Reg[Ri]+ D]

- D: Constant "displacement"
- Rb: Base register: Any of 16 integer registers
- Ri: Index register: Any, except for %rsp
- S: Scale: 1, 2, 4, or 8

Expression	Address Computation	Address
`0x8(%rdx)`		
`(%rdx,%rcx)`		
`(%rdx,%rcx,4)`		
`0x80(,%rdx,2)`		

Address Computation Examples

`%rdx`	`0xf000`
`%rcx`	`0x0100`

Most General Form

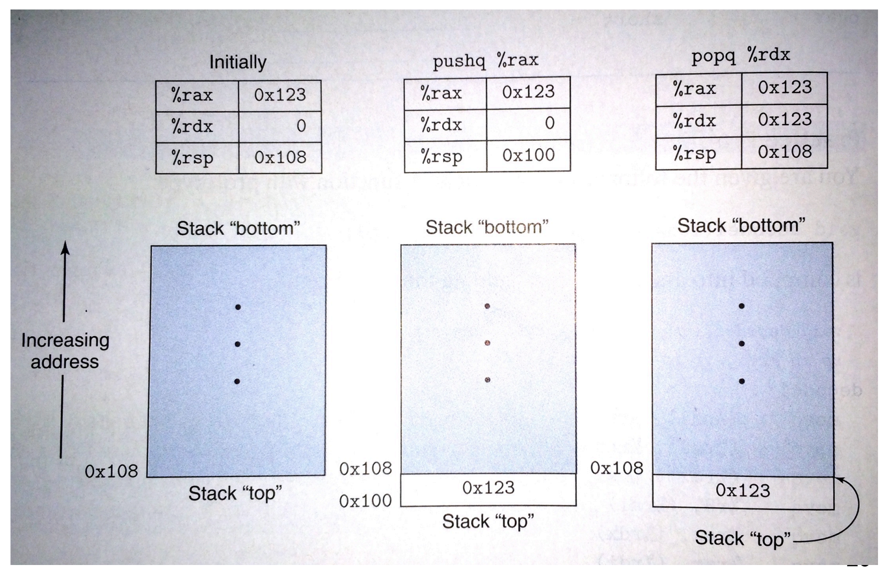
D(Rb,Ri,S) ⟶ Mem[Reg[Rb]+S*Reg[Ri]+ D]

- D: Constant “displacement”
- Rb: Base register: Any of 16 integer registers
- Ri: Index register: Any, except for %rsp
- S: Scale: 1, 2, 4, or 8

Expression	Address Computation	Address
`0x8(%rdx)`	`0xf000 + 0x8`	`0xf008`
`(%rdx,%rcx)`		
`(%rdx,%rcx,4)`		
`0x80(,%rdx,2)`		

Pushing and Popping Stack Data

- In addition to **mov**, can move data to and from program stack using **push** and **pop** (or pushq, popq, etc)
  - Recap: Stacks are LIFO (Last In First Out)
  - Usually drawn upside down ("top" of stack is on bottom of pic) 🤷‍♀️
  - The stack is part of memory
  - Registers are part of CPU
- **%rsp** holds address of top element of stack (aka stack pointer)
- **push**: Add data to top of stack
- **pop**: Remove data from stack



Address Computation Instruction

- **`leaq`** Src, Dst
  - "Load Effective Address" – copy memory **address** in src to dst
  - Src is an address
  - Set Dst to address denoted by expression
  - Unlike `mov`, which copies data at the address src to dst, `lea` copies the *value* of src itself to dst
- Uses
  - Computing addresses without a memory reference
    - E.g., translation of **`p = &x[i];`**
  - Computing arithmetic expressions of the form x + k*y
    - k = 1, 2, 4, or 8
- Example

```
long m12(long x){
  return x*12;
}
```

Converted to x86 by compiler:

```
leaq (%rdi,%rdi,2), %rax   # t = x+2*x
salq $2, %rax              # return t<<2
```

(salq is a left shift)

Understanding mov vs. lea

Operands	mov interpretation	lea interpretation
`6(%rax), %rdx`	Go to the address (6 + what’s in %rax), and **copy data** there into %rdx	Copy (6 + what’s in %rax) into %rdx.
`(%rax, %rcx), %rdx`	Go to the address (what’s in %rax + what’s in %rcx) and **copy data** there into %rdx	Copy (what’s in %rax + what’s in %rcx) into %rdx.
`(%rax, %rcx, 4), %rdx`	Go to the address (%rax + 4 * %rcx) and **copy data** there into %rdx.	Copy (%rax + 4 * %rcx) into %rdx.
`7(%rax, %rax, 8), %rdx`	Go to the address (7 + %rax + 8 * %rax) and **copy data** there into %rdx.	Copy (7 + %rax + 8 * %rax) into %rdx.

Unlike mov, which copies data at the address src to the destination dst, lea copies the value of src itself to dst. It does not “follow the pointer” and copy data from memory.

Today:
Machine Programming: Ops & Control

- History of Intel processors and architectures
- Assembly Basics: Registers, operands, move
- Arithmetic & logical operations
- C, assembly, machine code
- Intro to data-dependent control

Arithmetic Operations

- Two Operand Instructions:

Format		Computation	
`addq`	Src, Dest	Dest = Dest + Src	
`subq`	Src, Dest	Dest = Dest – Src	
`imulq`	Src, Dest	Dest = Dest * Src	
`salq`	Src, Dest	Dest = Dest << Src	(Also called `shlq`)
`sarq`	Src, Dest	Dest = Dest >> Src	Arithmetic right shift
`shrq`	Src, Dest	Dest = Dest >> Src	Logical right shift
`xorq`	Src, Dest	Dest = Dest ^ Src	
`andq`	Src, Dest	Dest = Dest & Src	
`orq`	Src, Dest	Dest = Dest | Src	

- Remember argument order! *Src, Dest*
  (Warning (again): Intel docs use "op *Dest,Src*")
- No distinction between signed and unsigned int in x86

Arithmetic Operations

- One Operand Instructions

`incq`	Dest	Dest = Dest + 1
`decq`	Dest	Dest = Dest – 1
`negq`	Dest	Dest = – Dest
`notq`	Dest	Dest = ~Dest

- See book for a more complete list!

Arithmetic Expression 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
```

Instructions

- **`leaq`**: address computation
- **`salq`**: left shift
- **`imulq`**: multiplication
  - Only used once

Arithmetic Expression 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    # t1
  addq    %rdx, %rax           # t2
  leaq    (%rsi,%rsi,2), %rdx
  salq    $4, %rdx             # t4
  leaq    4(%rdi,%rdx), %rcx   # t5
  imulq   %rcx, %rax           # rval
  ret
```

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