UNM SCHOOL of ENGINEERING
Department of Computer Science

Assembly code: arithmetic operations and control

Lecture #8 (week 5)

Prof. Soraya Abad-Mota, PhD

(Chap. 3 – Secs. 3.4.2 to 3.4.4. and Sec 3.5)

Refresher Quiz (to do now on index cards)

- Name three of the 16 registers in the x86-64 microprocessor
- If an instruction has the following operand:
  - (r1,r2,4)
  - What can be r1 and r2?
  - What is its meaning? (or what does it translate into?)

Previously (secs. 3.2, 3.3 to 3.4.1)

- Turning C into Object Code
- Started learning details of Assembly code
  - Abstraction of the CPU components + interaction with Memory
  - Registers
  - Operand specifiers
- What does `w` stand for? What is its size?

Understand addressing modes (recall)

Address	Value	Register	Value
0x100	0xFF	%rax	0x100
0x104	0xAB	%rcx	0x1
0x108	0x13	%rdx	0x3
0x10C	0x11		

operand	value
4(%rax)	(0x100 + 0x4 = 0x104) content of = 0xAB
9(%rax, %rdx)	(0x10C) = 0x11
260(%rcx,%rdx)	0x104 + 0x1 + 0x3 = (0x108) = 0x13
0xFC(,%rcx,4)	(0xFC + 0x4) = (0x100) content of = 0xFF
(%rax,%rdx,4)	(0x100 + 0xC) = (0x10C) = 0x11

Learning objectives (for this portion)

After this session students should be able to:

1. Describe the data movement instructions.
2. Describe arithmetic and logical instructions.
3. Read assembly code containing move instructions as generated by a compiler.

Data movement instructions (figure 3.4)

Secs. 3.4.2 and 3.4.3

- `mov` class src,dest (from source to destination location)
  - **`movb`** move byte
  - **`movw`** move word
  - **`movl`** move double word (long word)
  - **`movq`** if source is immediate,
    has to repr. 32-bit two's-complement number
    that is sign extended to 64 bits (quad)

Data movement instructions

(special case)

- **movabsq** special instruction, can have arbitrary 64-bit immediate value in source but only register as dest.
- General rule: size of operand determines portion of the register to be updated. Exception: movl, also sets the high-order 4 bytes of the register to 0
- Check p. 184, aside blue rectangle to see an example of how each of the `mov` instructions work.

Moving Data with movq

- **`movq` *Source, Dest*:**
- Operand Types (from earlier)
  - ***Immediate:*** Constant integer data
    - Example: **`$0x400, $-533`**
    - Like C constant, but prefixed with **`'$'`**
    - Encoded with 1, 2, or 4 bytes
  - ***Register:*** One of 16 integer registers
    - Example: **`%rax, %r13`**
    - But **`%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 "address modes" (as we saw earlier)

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

%rN

More data movement instructions in Figures 3.5 and 3.6.

`movq` Operand Combinations (and examples)

	Source	Dest	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;`

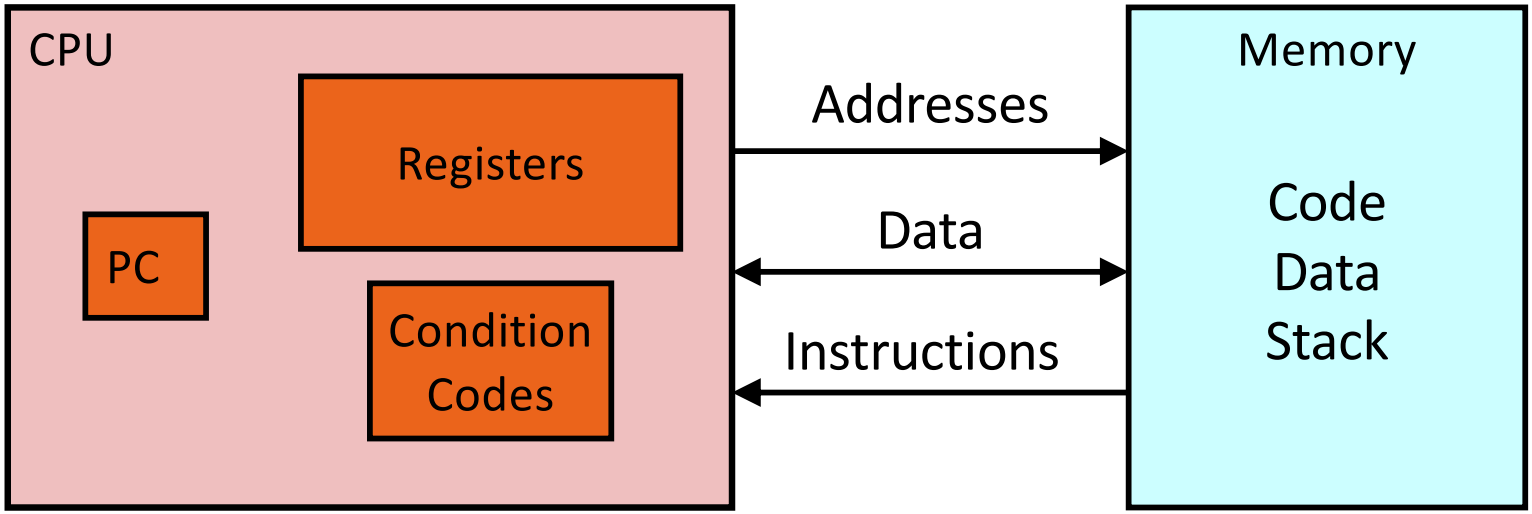
Cannot do memory-memory transfer with a single instruction, i.e. source and destination cannot be both references to memory locations.

`movq` Operand Combinations

	Source	Dest	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;`

Can the Dest operand be an Imm value?

Assembly/Machine Code View (recall)



Programmer-Visible State (only storage is shown here)

- **PC: Program counter**
  - Address of next instruction
  - Called "RIP" (instruction pointer register) x86-64
- **Register file**
  - Contains 16 named locations
  - Heavily used program data
- **Condition codes**
  - Store status information about most recent arithmetic or logical operation
  - Used for conditional branching
- **Memory**
  - Byte addressable array
  - Code and user data
  - Stack to support procedures

Simple Memory Addressing Modes

(recall and now applied to specific instructions)

- Normal (reg) M[R[reg]]
  - reg specifies memory address
  - Aha! Pointer dereferencing in C

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

- Displacement D(reg) M[R[reg]+D]
  - reg specifies start of memory region
  - Constant displacement D specifies offset

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

Practice Problems (keep list of registers handy)

- Problem 3.2 complete the suffix
  1. `mov? %eax,(%rsp)`
  2. `mov? (%rax),%dx`
  3. `mov? (%rdx),%rax`
  - students do the other 3 on their own
- **`Problem 3.3`** **`what is wrong with each?`**
  1. `movw (%rax),4(%rsp)`
  2. `movl %rax,(%rsp)`
  3. `movq %rax,$0x123`
  - students do the remaining 4 on their own

Example of Simple Addressing Modes

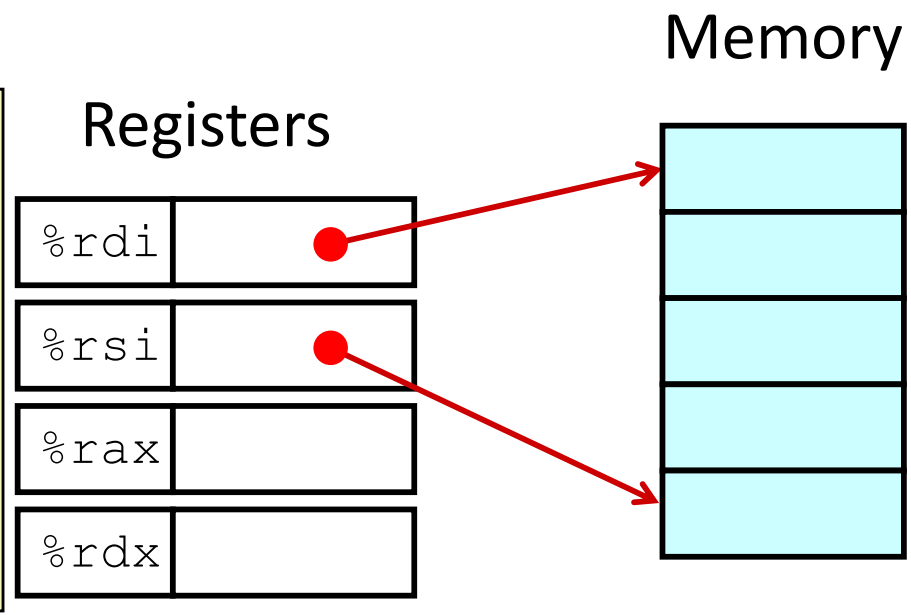
```
void swap
   (long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

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

(Details of function call and return in section 3.7.
For now, enough to know that 1st argument goes to %rdi and the 2nd to %rsi)
Now let's see the details of how swap operates.

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

Notice local vars stored in registers, arguments also in registers, specific ones. See table of registers fig. 3.2.

```
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	

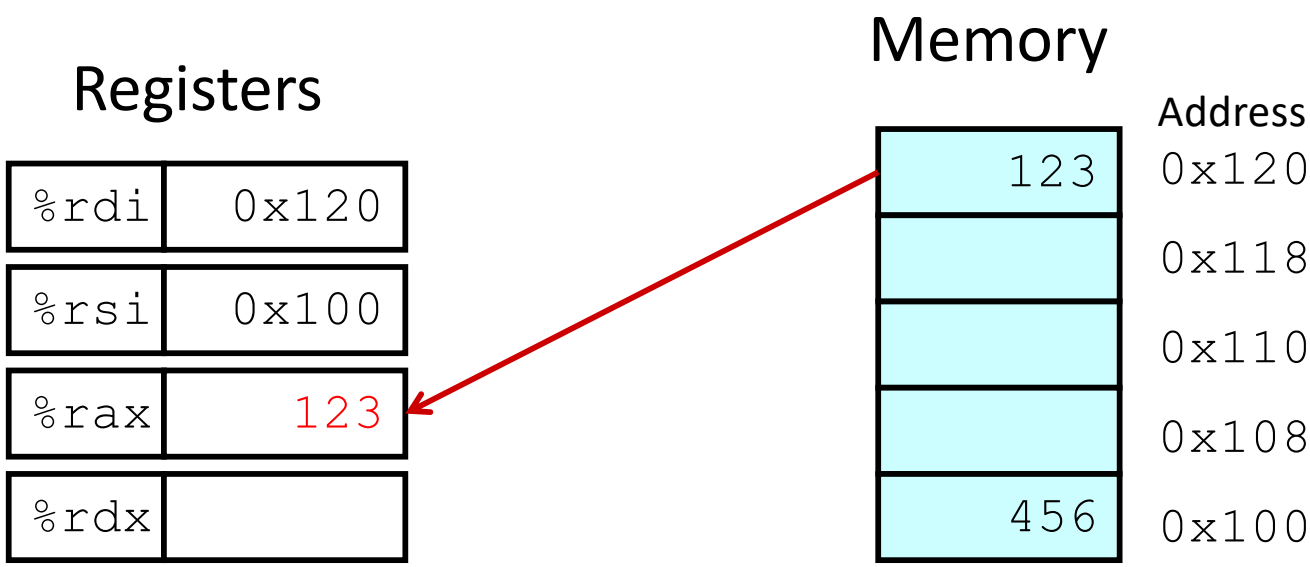
Memory

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

Notice order of addresses

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

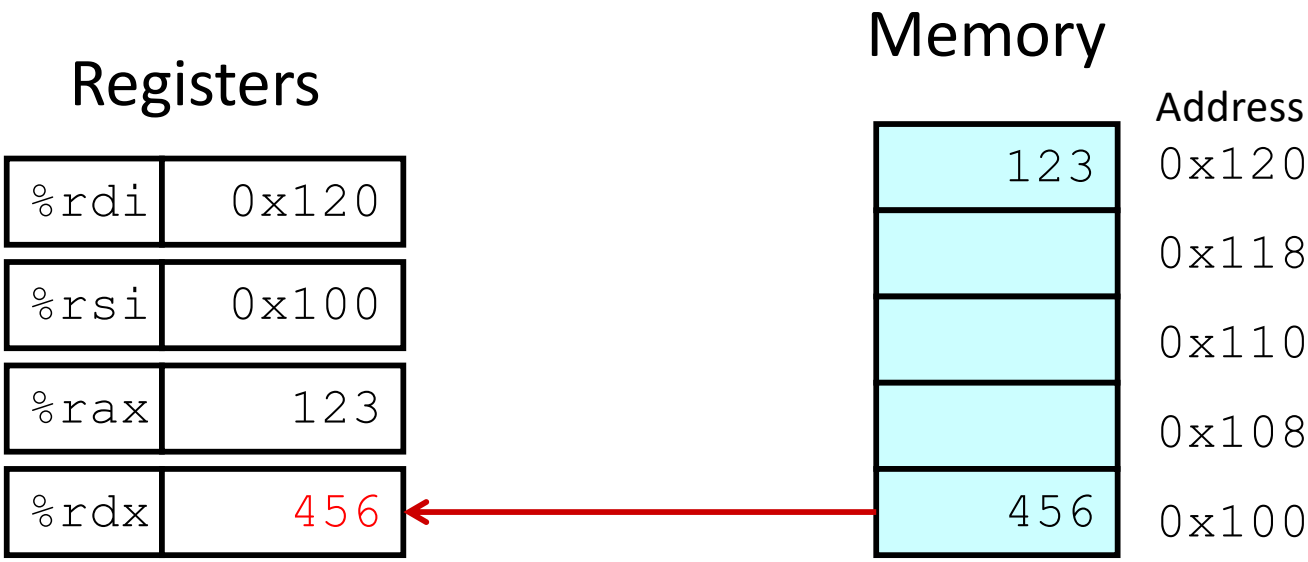
Understanding `Swap()`



Notice order

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

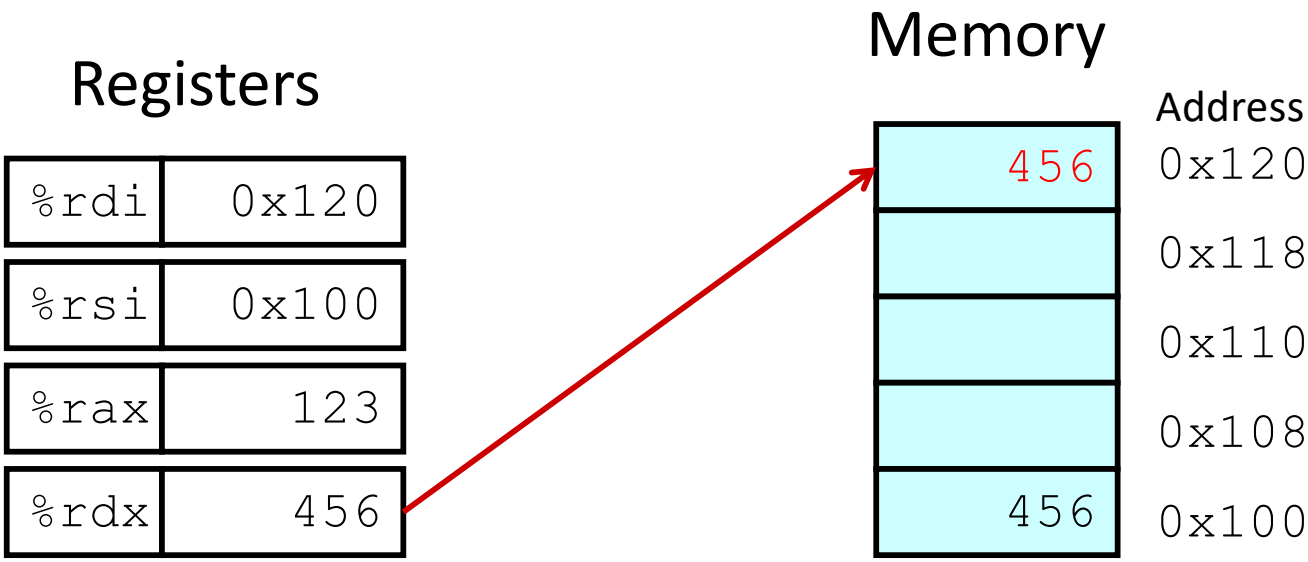
Understanding `Swap()`



Notice order

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

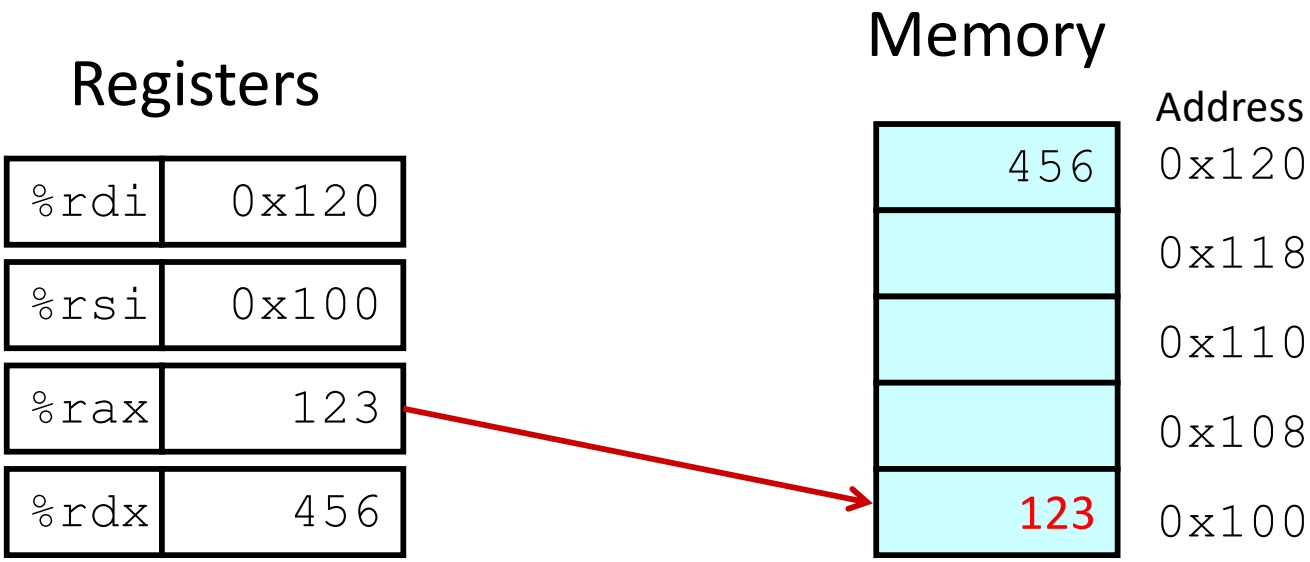
Understanding `Swap()`



Notice order

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

Understanding `Swap()`



Notice order

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

Complete Memory Addressing Modes (recall)

- Most General Form

  $D(r_b,r_i,S)$ $\quad$ $M[R[r_b]+S*R[r_i]+ D]$

  - D: Constant "displacement" 1, 2, or 4 bytes
  - r_b: Base register, any of 16 integer registers
  - r_i: Index register: any, except for **%rsp**
  - S: Scale: 1, 2, 4, or 8 (*why these numbers?*)
- Special Cases

  (r_b,r_i) $\quad$ $M[R[r_b]+R[r_i]]$

  $D(r_b,r_i)$ $\quad$ $M[R[r_b]+R[r_i]+D]$

  (r_b,r_i,S) $\quad$ $M[R[r_b]+S*R[r_i]]$

More address computation examples

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

We did a few previously
May do these additional ones

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

More address computation examples

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

We did a few previously
May do these additional ones

Expression	Address Computation	Address
`$0x8(%rdx)`	`0xf000 + 0x8`	`0xf008`
`(%rdx,%rcx)`	`0xf000 + 0x100`	`0xf100`
`(%rdx,%rcx,4)`	`0xf000 + 4*0x100`	`0xf400`
`$0x80(,%rdx,2)`	`2*0xf000 + 0x80`	`0x1e080`

Learning objectives (for this portion)

After this session students should be able to:

1. Describe the data movement instructions (sec 3.4).
2. Describe arithmetic and logical instructions (sec 3.5).
3. Read assembly code containing move instructions as generated by a compiler.

Operation groups

- load effective address: `leaq` (variant of `movq`; q is for quadruple word)
- unary operations
- binary operations
- shifts

- All the operations are listed in figure 3.10, p. 192 (keep this figure handy)

Address Computation Instruction

- **`leaq`** S, D (does not refer to memory at all, copies effective address to D)
  - S is address mode expression
  - Set D to address denoted by expression. Must be a register
- 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 ASM by compiler:

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

Problem 3.6 (p. 192)

- %rax holds x and %rcx holds y

Instruction	Result
leaq 6(%rax), %rdx	
leaq (%rax,%rcx), %rdx	
leaq (%rax,%rcx,4), %rdx	
leaq 7(%rax,%rax,8), %rdx	

Problem 3.6 (p. 192)

- %rax holds x and %rcx holds y

Instruction	Result (is an address stored in %rdx)
leaq 6(%rax), %rdx	6+x
leaq (%rax,%rcx), %rdx	x+y
leaq (%rax,%rcx,4), %rdx	
leaq 7(%rax,%rax,8), %rdx	

Some Arithmetic Operations (binary)

- 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 (3 shifts use %cl)
`sarq`	Src,Dest	Dest = Dest >> Src	Arithmetic
`shrq`	Src,Dest	Dest = Dest >> Src	Logical
`xorq`	Src,Dest	Dest = Dest ^ Src	
`andq`	Src,Dest	Dest = Dest & Src	
`orq`	Src,Dest	Dest = Dest | Src	

- Watch out for argument order! (D is used as src and dest)
- No distinction between signed and unsigned int

Some Arithmetic Operations (unary)

- One Operand Instructions

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

- See textbook and summary document (posted on canvas) for more machine instructions and how to study them. We cannot cover them all in lectures.

Practice (in class)

- Problem 3.8, p. 194

Practice in class: Problem 3.8, p. 194

Address	Value	Register	Value
0x100	0xFF	%rax	0x100
0x108	0xAB	%rcx	0x1
0x110	0x13	%rdx	0x3
0x118	0x11		

Fill in the following table showing the effects of the following instructions, in terms of both the register or memory location that will be updated and the resulting value:

Instruction	Destination	Value
addq %rcx,(%rax)	_____	_____
subq %rdx,8(%rax)	_____	_____
imulq $16,(%rax,%rdx,8)	_____	_____
incq 16(%rax)	_____	_____
decq %rcx	_____	_____
subq %rdx,%rax	_____	_____

Practice (after the lecture)

- Problem 3.9, p. 195
- Problem 3.11, p. 197
- And follow the example in the next two slides on your own, so that we can continue to Condition Codes

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
```

Interesting Instructions

- **`leaq`**: address computation
- **`salq`**: shift
- **`imulq`**: multiplication
  - But, only used once

%rdi holds x, %rsi holds y, %rdx holds z (see next slide)

Understanding 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**
%rax	**t1**, **t2**, **rval**
%rdx	**t4**
%rcx	**t5**

Follow details on your own

Learning objectives (stated at the beginning)

After this session students should be able to:

1. Describe the data movement instructions.
2. Describe arithmetic and logical instructions.
3. Read assembly code containing move instructions as generated by a compiler.

Practice with this problem (on your own)

- Problem 3.7 p. 193

```
long scale2(long x, long y, long z){
    long t =                          ;
    return t;
}
scale2: leaq (%rdi,%rdi,4),%rax
        leaq (%rax,%rsi,2),%rax
        leaq (%rax,%rdx,8),%rax
        ret
```

End lecture #8

End of section 3.5