CS330 - Computer Organization and Assembly Language Programming

Lecture 11

-Machine Level Programming-

Professor : Mahmut Unan – UAB CS

Agenda

- Assembly Basics: Registers, operands, move

x86-64 Integer Registers

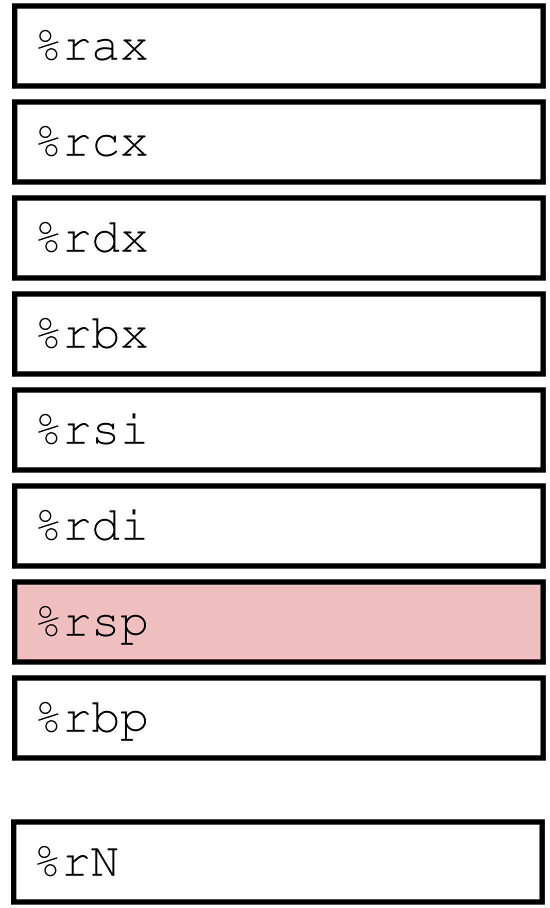
%rax	%eax	%r8	%r8d
%rbx	%ebx	%r9	%r9d
%rcx	%ecx	%r10	%r10d
%rdx	%edx	%r11	%r11d
%rsi	%esi	%r12	%r12d
%rdi	%edi	%r13	%r13d
%rsp	%esp	%r14	%r14d
%rbp	%ebp	%r15	%r15d

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

Some History: IA32 Registers

Moving Data

- Moving Data
  `movq` ***Source*, *Dest***:
- Operand Types
  - ***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"



`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;`

Cannot do memory-memory transfer with a single instruction

Simple Memory Addressing Modes

- **Normal (R) Mem[Reg[R]]**
  – Register R specifies memory address
  – Aha! Pointer dereferencing in C

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

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

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

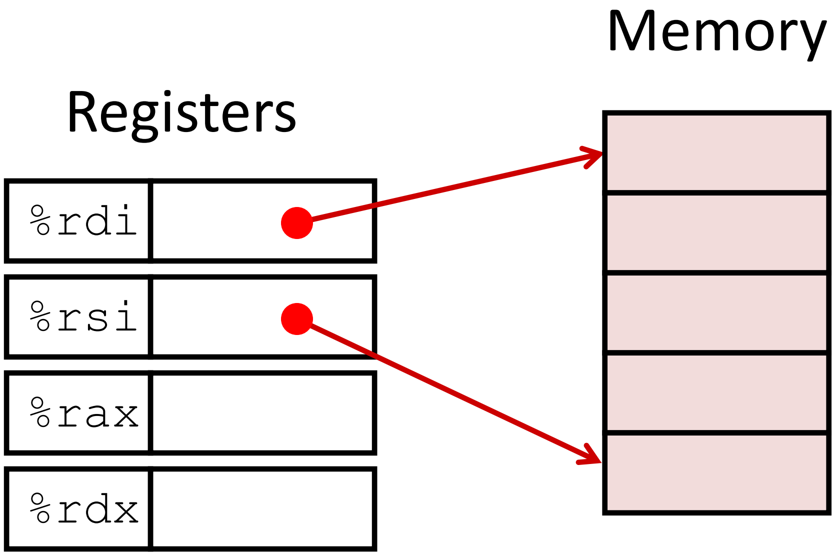
Example of Simple Addressing Modes

```
void 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()

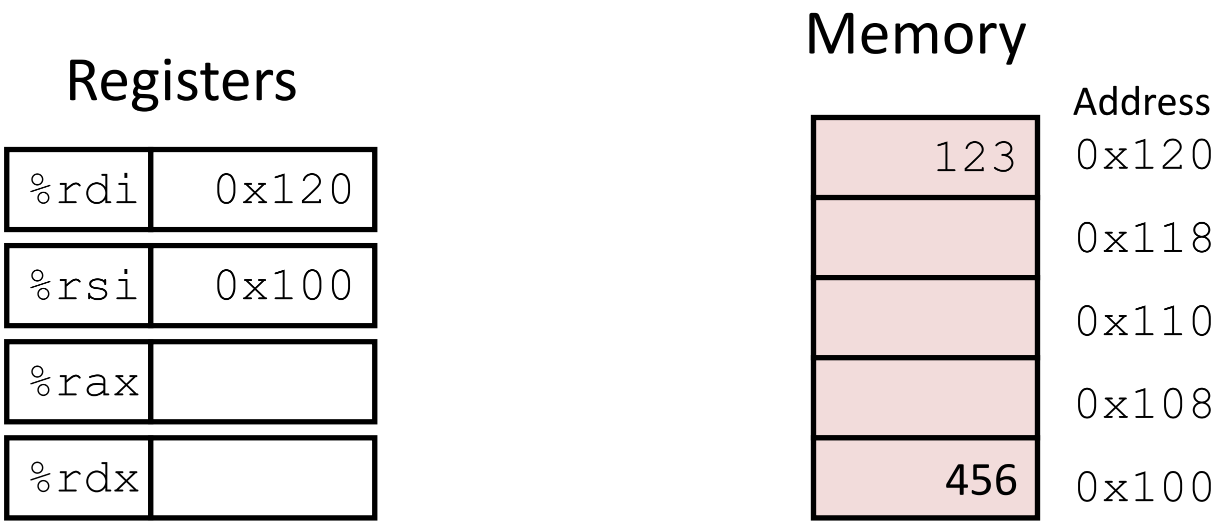
```
void swap
   (long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```



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

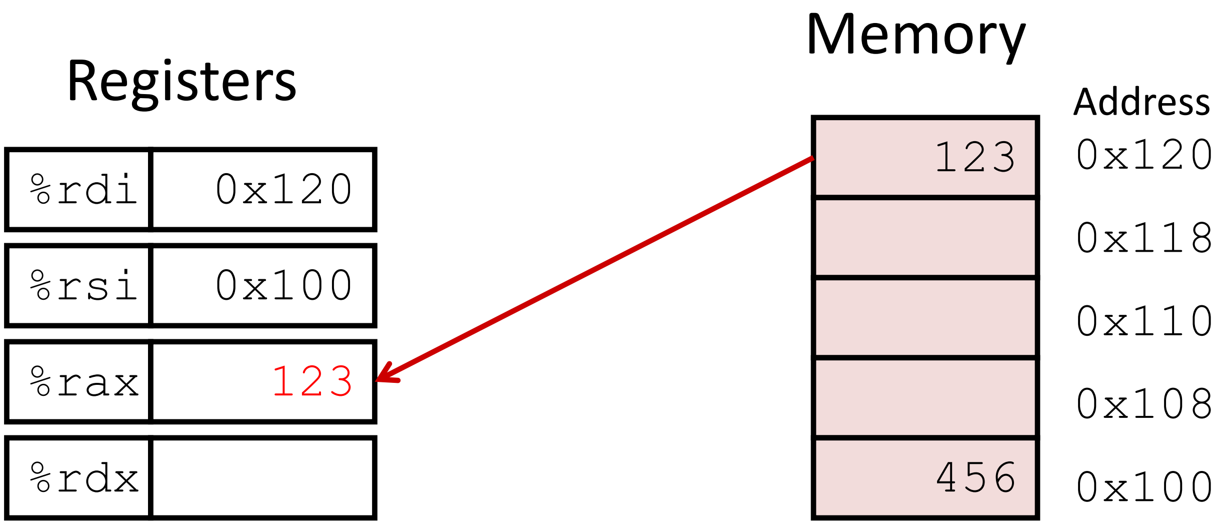
```
swap:
   movq    (%rdi), %rax   # t0 = *xp
   movq    (%rsi), %rdx   # t1 = *yp
   movq    %rdx, (%rdi)   # *xp = t1
   movq    %rax, (%rsi)   # *yp = t0
   ret
```

Understanding `Swap()`



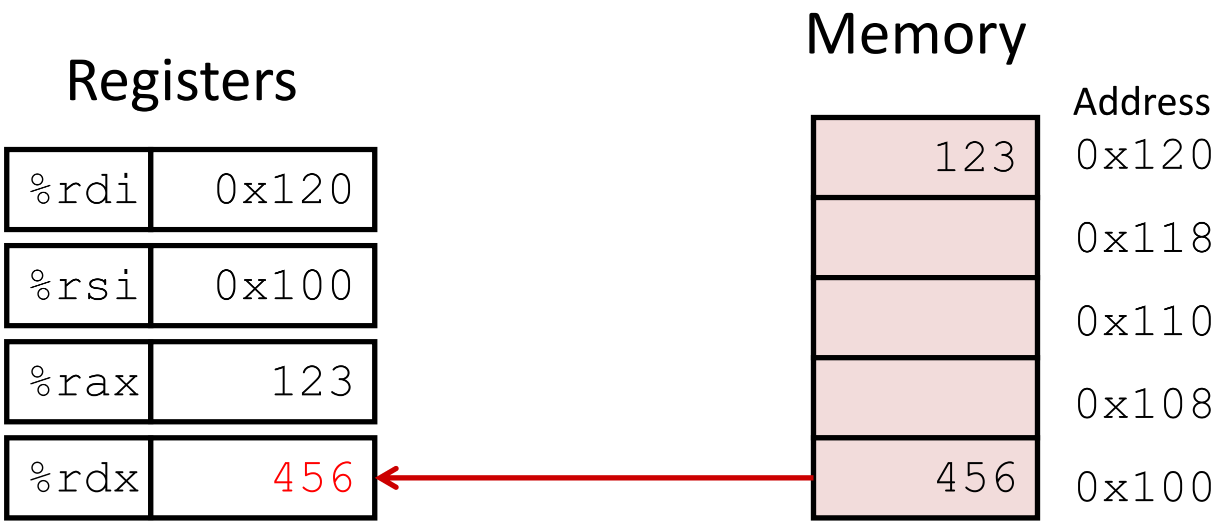

```
swap:
   movq    (%rdi), %rax  # t0 = *xp
   movq    (%rsi), %rdx  # t1 = *yp
   movq   %rdx, (%rdi)   # *xp = t1
   movq   %rax, (%rsi)   # *yp = t0
   ret
```

Understanding Swap()



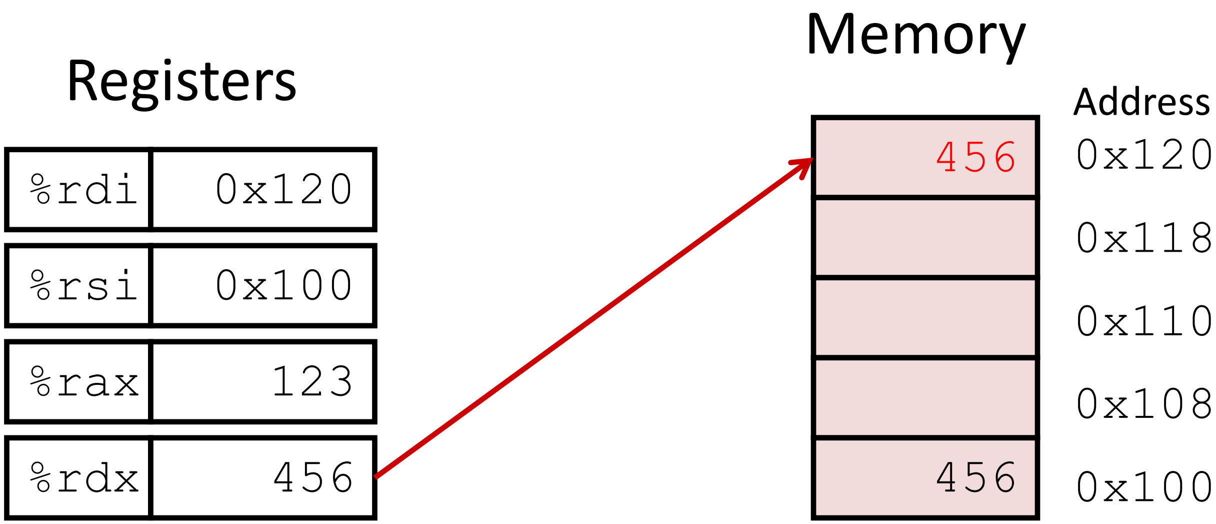

```
swap:
   movq     (%rdi), %rax   # t0 = *xp
   movq     (%rsi), %rdx   # t1 = *yp
   movq     %rdx, (%rdi)   # *xp = t1
   movq     %rax, (%rsi)   # *yp = t0
   ret
```

Understanding Swap()



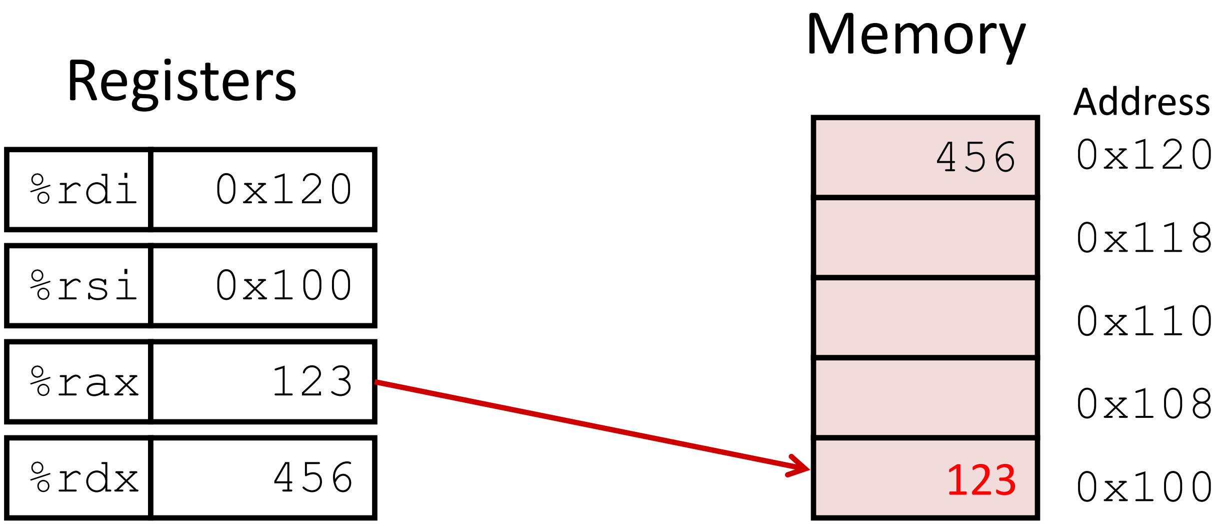

```
swap:
    movq     (%rdi), %rax   # t0 = *xp
    movq     (%rsi), %rdx   # t1 = *yp
    movq     %rdx, (%rdi)   # *xp = t1
    movq     %rax, (%rsi)   # *yp = t0
    ret
```

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 (R) Mem[Reg[R]]
  - Register R specifies memory address
  - Aha! Pointer dereferencing in C

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

- Displacement D(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” 1, 2, or 4 bytes

  – Rb: Base register: Any of 16 integer registers

  – Ri: Index register: Any, except for `%rsp`

  – S: Scale: 1, 2, 4, or 8

- **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]]

AT&T Syntax vs Intel Syntax

- AT&T Sytanx

  movl %esp, %ebp

  Instruction source destination

- Intel Syntax

  movq ebp, esp

  Instruction destination source

  No percent signs

Pushing and Popping Stack Data

- Stack plays a vital role in the handling of procedure calls.
- Stack = a data structure where values can be added or deleted → according to last in first out discipline

- Add data to stack → push
- Remove data from stack → pop

Push	Source	Push source onto stack
Pop	Destination	Pop top of stack into destination

EXAMPLES:

pushq %rbp

Popq M[R[%rsp]]

Initially

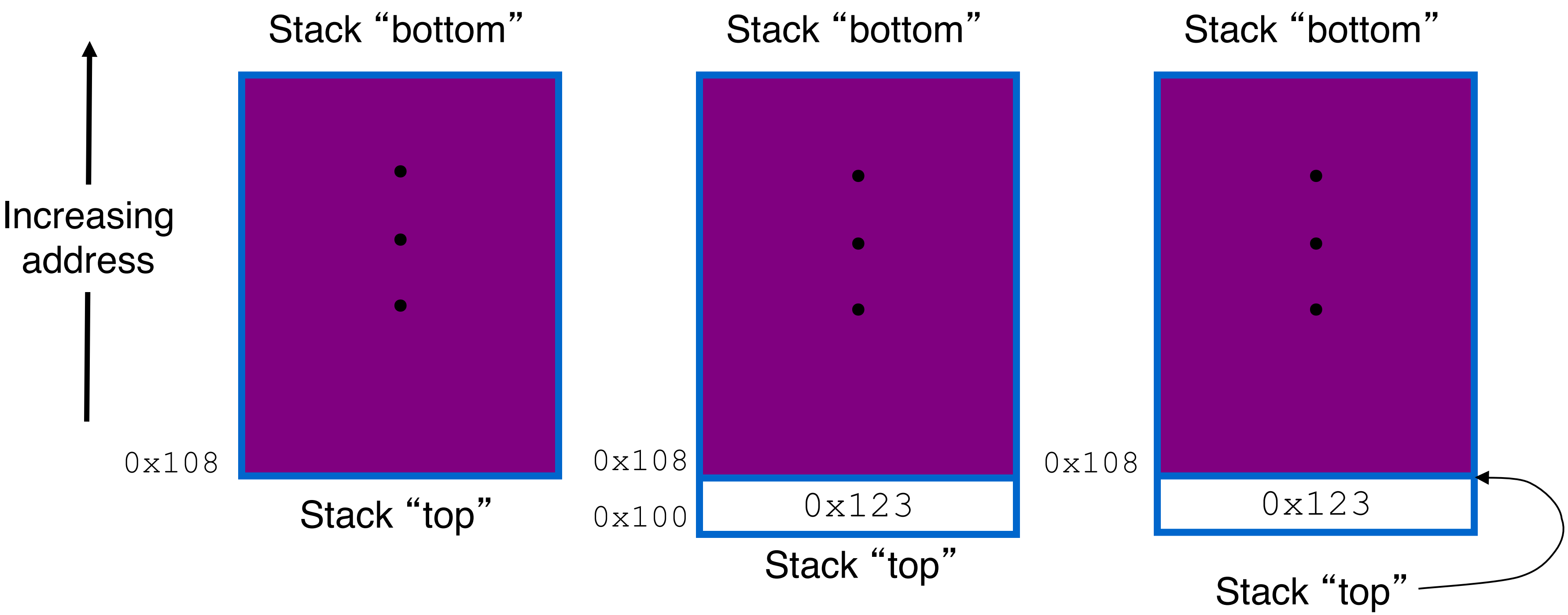
%rax	0x123
%rdx	0
%rsp	0x108

`pushq %rax`

%rax	0x123
%rdx	0
%rsp	0x100

`popq %rdx`

%rax	0x123
%rdx	0x123
%rsp	0x108



- The stack pointer %rsp holds the address of the top stack element.
- Pushing a quad word value onto the stack involves first decrementing the stack pointer by 8 and then writing the value at the new top of stack address.
- `pushq %rbp` is equal to

```
subq &8, %rsp
movq %rbp, (%rsp)
```

recall -Complete Memory Addressing Modes

- **Most General Form**

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

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

- **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**

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

Address Computation Examples

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

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`

Arithmetic

&

Logical Operations

Address Computation Instruction

- **`leaq`** Source, Destination
  - Source is address mode expression
  - Set Destination to address denoted by expression

leaq S, D D ← &S Load effective address

- 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
```

More on **LEAQ**

- The destination operand must be a register

`leaq 7(%rdx, %rdx,4), %rax`

→set register `%rax` to `5x+7`

If `%rax` has the value x, and `%rcx` holds the value y

```
leaq 6(%rax), %rdx
leaq (%rax, %rcx), %rdx
```

More on **LEAQ**

- The destination operand must be a register

```
leaq 7(%rdx, %rdx,4), %rax
```

→set register `%rax` to `5x+7`

If `%rax` has the value x, and `%rcx` holds the value y

```
leaq 6(%rax), %rdx     → 6 + x
leaq (%rax, %rcx), %rdx → x + y
```

Basic Arithmetic Operations

- ADD
- SUB
- IMUL
- IDIV

ADD

```
ADDQ  %rbx,  %rax
```

- adds %rbx to %rax, and overwrites the result in %rax

```
ADDQ  &8,  %rsp
```

adds 8 to the stack pointer %rsp, (incrementing)

- c = a + b;

```
MOVQ  a, %rax
MOVQ  b, %rbx
ADDQ  %rbx, %rax
MOVQ %rax, c
```

leaq for basic arithmetic operations

```
long scale(long x, long y, long z){
long t= x  + 4 * y + 12 * z;
return t;
}

scale:
leaq (%rdi, %rsi, 4), %rax
leaq (%rdx, %rdx, 2), %rdx
leaq (%rax, %rdx, 4), %rax
```

leaq for basic arithmetic operations

```
long scale(long x, long y, long z){
long t= x  + 4 * y + 12 * z;
return t;
}
x in %rdi, y in %rsi, z in %rdx
scale:
leaq (%rdi, %rsi, 4), %rax    x+4*y
leaq (%rdx, %rdx, 2), %rdx   z+2*z=3*z
leaq (%rax, %rdx, 4), %rax (x+4*y)+4*(3*z)
```

Some 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
`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	

imulq

Imul S, D   D ← D*S

`return a*b + c*d;`

*Assume a in %edi, b in %sil, c in %rdx, d in %ecx

```
movslq %ecx,%rcx
movsbl %sil,%esi
imulq %rdx, %rcx
imull %edi, %esi
leal (%rsi,%rcx), %eax
ret
```

salq, shrq, sarq, shlq….

- Left/right shift operations
- salq $4, %rax → left shift rax by 4 bits
- For example

```
movw   $ff00,%ax         # ax=1111.1111.0000.0000 (0xff00, unsigned 65280, signed -256)
shrw   $3,%ax            # ax=0001.1111.1110.0000 (0x1fe0, signed and unsigned 8160)
                         # (logical shifting unsigned numbers right by 3
                         #   is like integer division by 8)
shlw   $1,%ax            # ax=0011.1111.1100.0000 (0x3fc0, signed and unsigned 16320)
                         # (logical shifting unsigned numbers left by 1
                         #   is like multiplication by 2)
```


https://en.wikibooks.org/wiki/X86_Assembly/Shift_and_Rotate

Binary Operations

`xorq, andq, orq, notq…`

```
movl $0x1, %eax ; eax ≔ 1
movl $0x0, %ebx ; ebx ≔ 0
orl  %eax, %ebx ; ebx ≔ eax ∨ ebx
```

`ebx` would be 1 because 1 ∨ 0 ⇔ 1

```
notl %ebx   ; ebx ≔ 0
```

Unary Operations

- Single operand serving as both source and destination
- Operand can be either a register or memory location

`incq(%rsp)` similar to ++ or – in C

`decq (%rcx)`

Other 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 more instructions

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

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`

```
long arith(long x, long y, long z){
long t1 = x ^ y;
long t2= z*48;
long t3= t1 & 0x0F0F0F0F;
long t4= t2-t3;
return t4;
} // Assume x in %rdi, y in %rsi , and z in %rdx
arith:
    xorq %rsi, %rdi              long t1 = x ^ y;
    leaq  (%rdx, %rdx,2), %rax        3*z
    salq   $4, %rax          t2 = 16*(3*z)=48*z
    andl   $252645135, %edi     t3 = t1 & 0x0F0F0F0F
    subq   %rdi, %rax           return t2-t3;
    ret
```

Machine Programming I: Summary

- History of Intel processors and architectures
  - Evolutionary design leads to many quirks and artifacts
- C, assembly, machine code
  - New forms of visible state: program counter, registers, ...
  - Compiler must transform statements, expressions, procedures into low-level instruction sequences
- Assembly Basics: Registers, operands, move
  - The x86-64 move instructions cover wide range of data movement forms
- Arithmetic
  - C compiler will figure out different instruction combinations to carry out computation

Next Class

- Read Chapter 3.6