Machine-Level Programming: Controls

Prof. Hyuk-Yoon Kwon
http://bigdata.seoultech.ac.kr

Most parts are based on slides written by Brayant and O'Hallaon, CMU
(http://csapp.cs.cmu.edu/3e/instructors.html)

Homework Assignment #2

- **Released date: 10/11 (Mon.)**
- **Due date: 11/8 (Mon.)**
- **Where to submit: to e-class (http://eclass.seoultech.ac.kr)**
  - Late submission is not allowed.
- **Assigned score: 7 points**
- **CAUTION: penalty for cheating**
  - If cheating is detected, you will get an F
  - Some automatic tools are used to detect cheating
    - Copying and making some modifications will also be automatically detected

Makefile

```
edit : main.o kbd.o command.o display.o ₩
       insert.o search.o files.o utils.o
        cc -o edit main.o kbd.o command.o display.o ₩
                   insert.o search.o files.o utils.o

main.o : main.c defs.h
        cc -c main.c
kbd.o : kbd.c defs.h command.h
        cc -c kbd.c
command.o : command.c defs.h command.h
        cc -c command.c
display.o : display.c defs.h buffer.h
        cc -c display.c
insert.o : insert.c defs.h buffer.h
        cc -c insert.c
search.o : search.c defs.h buffer.h
        cc -c search.c
files.o : files.c defs.h buffer.h command.h
        cc -c files.c
utils.o : utils.c defs.h
        cc -c utils.c

clean :
        rm edit main.o kbd.o command.o display.o ₩
           insert.o search.o files.o utils.o
```

Execute Makefile

- **To compile all the files in “edit”**

  Linux> make edit

  Or

  Linux> make

- **To compile all the files in “clean”**

  Linux> make clean

Compress and Decompress Files

Compression and decompression

- Compression: tar –czvf 3.tar.gz /home/my/tar_dir
- Decompression: tar –xvf 3.tar.gz

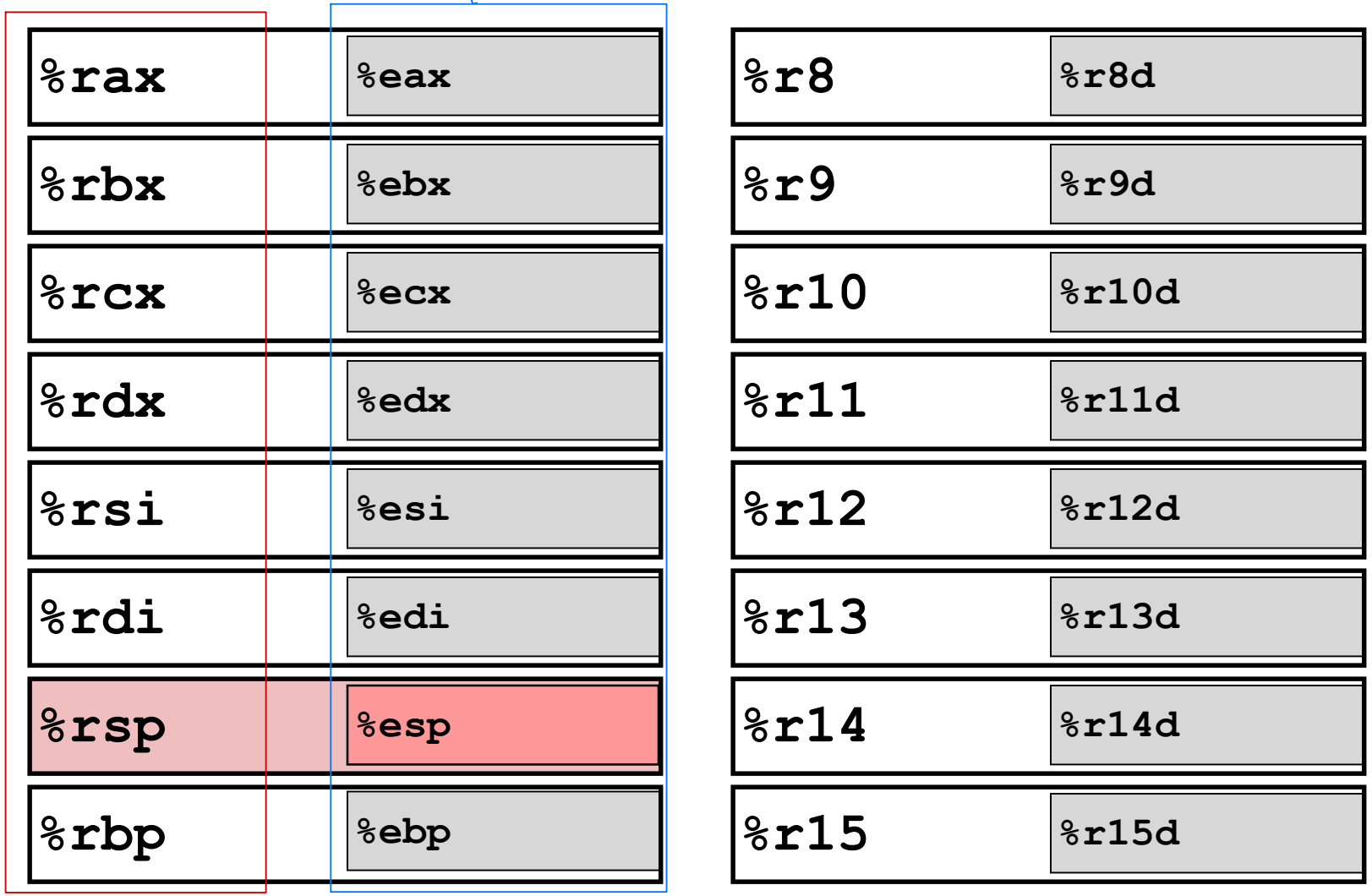
Basic Linux commands

- ls, mkdir, cd, pwd, and so on
- Refer to: https://maker.pro/education/basic-linux-commands-for-beginners

Today: Machine Programming I: Basics

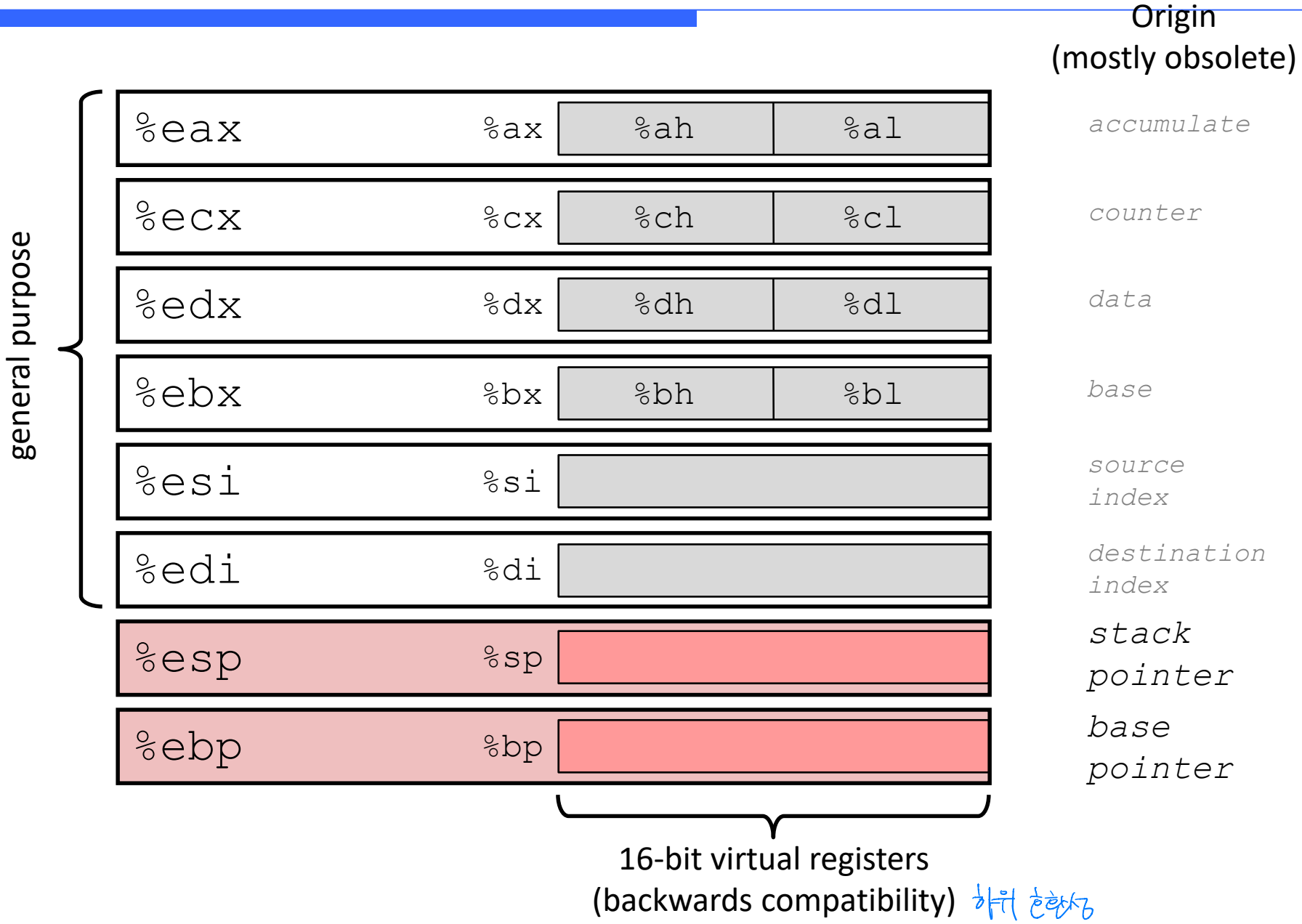
- History of Intel processors and architectures
- C, assembly, machine code
- **Assembly Basics: Registers, operands, move**
- Arithmetic & logical operations

x86-64 Integer Registers



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

Some History: IA32 Registers



Moving Data

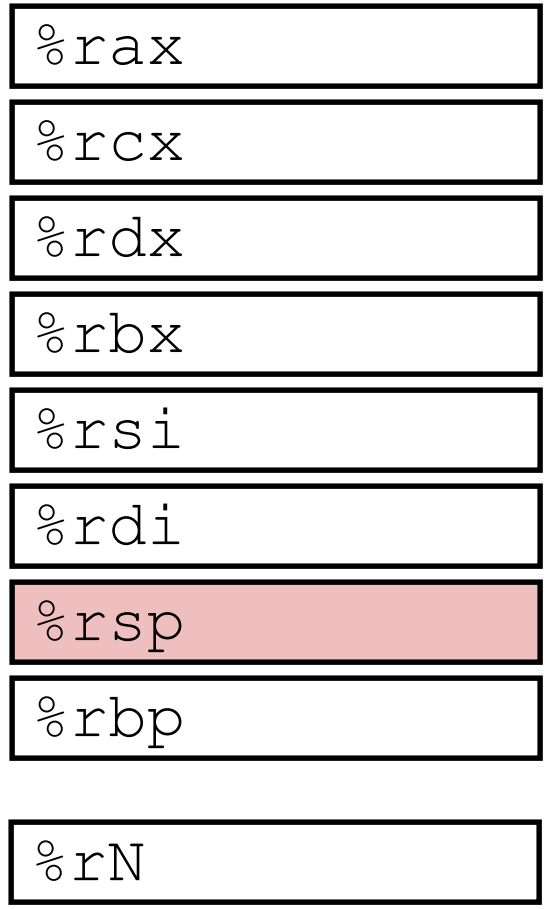
Moving Data

Source to Destination

`mov` *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

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
```

int i = *p

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

int arr[10] = {...};

int i = *(arr+8)

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

Example of Simple Addressing Modes

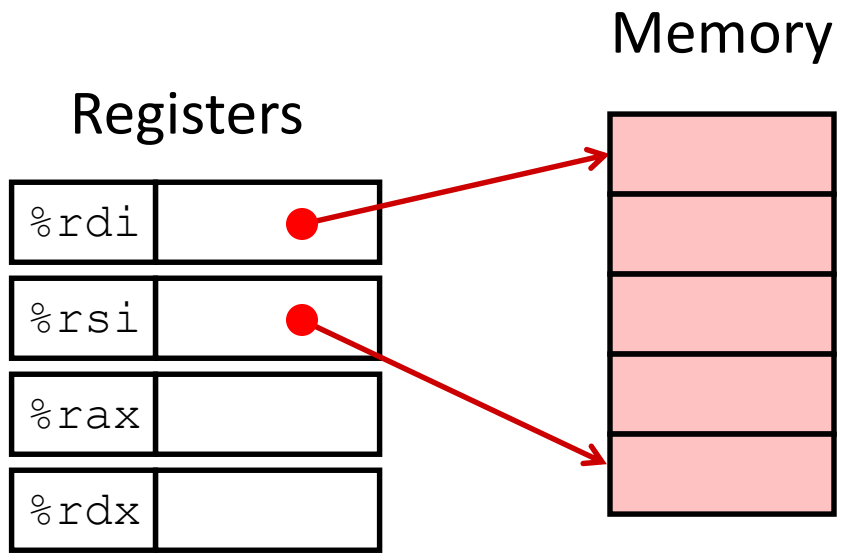
```
void swap
    (long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

Swap

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

Understanding `Swap()`

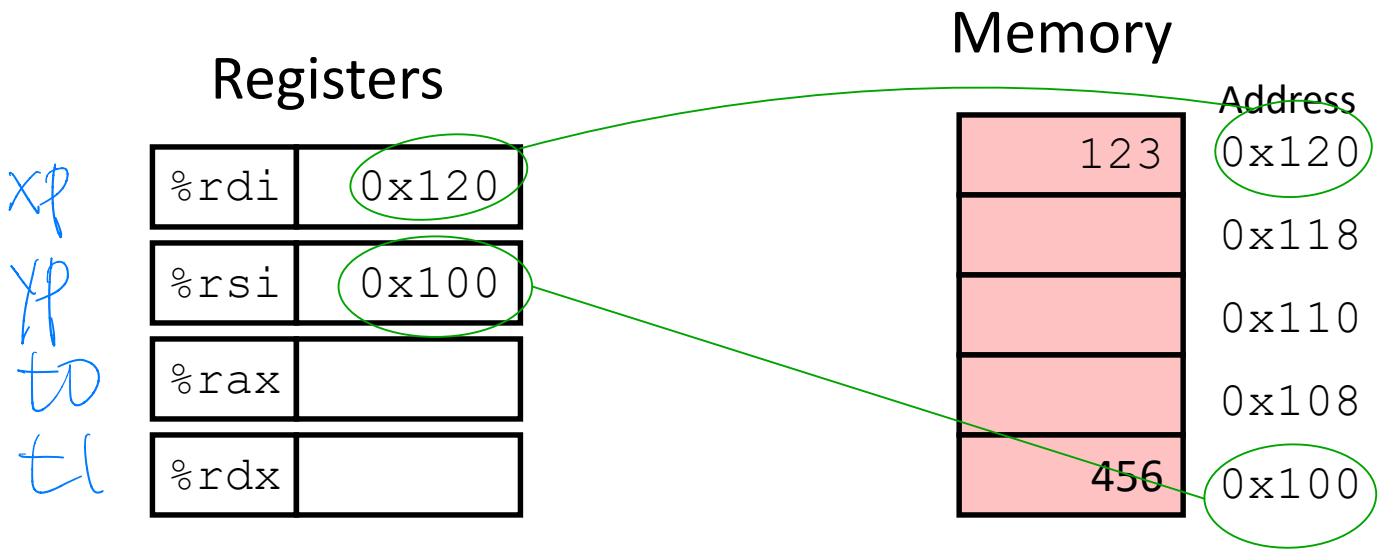
```
void swap
    (long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```



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

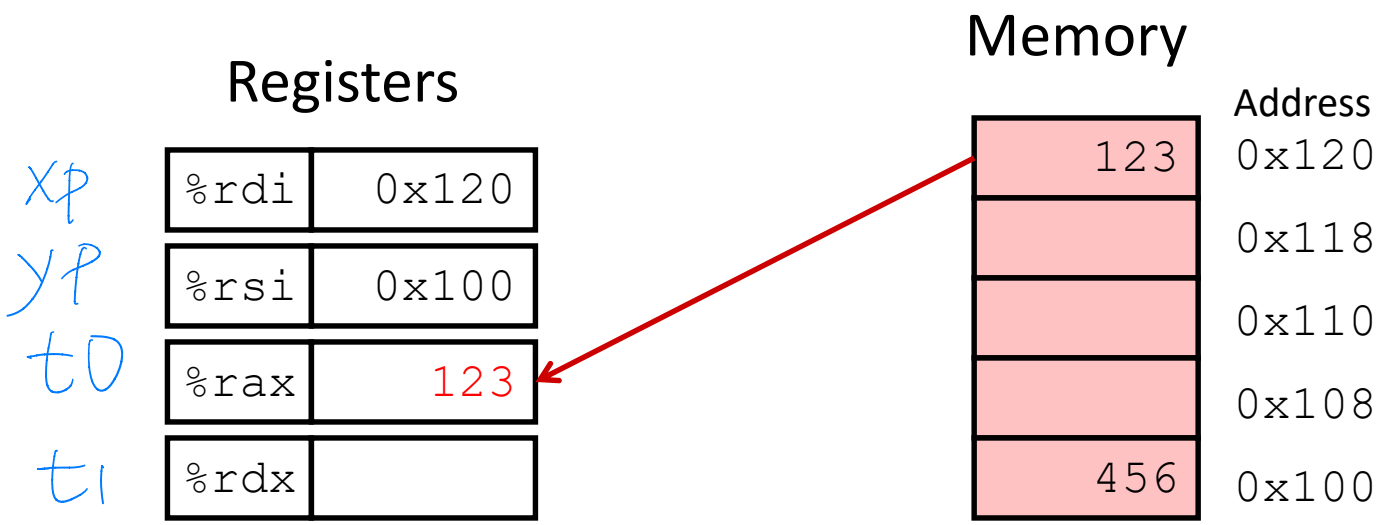
```
swap:
        movq    (%rdi), %rax  # t0 = *xp
        movq    (%rsi), %rdx  # t1 = *yp
        movq    %rdx, (%rdi)  # *xp = t1
        movq    %rax, (%rsi)  # *yp = t0
        ret
```

Understanding `Swap()`



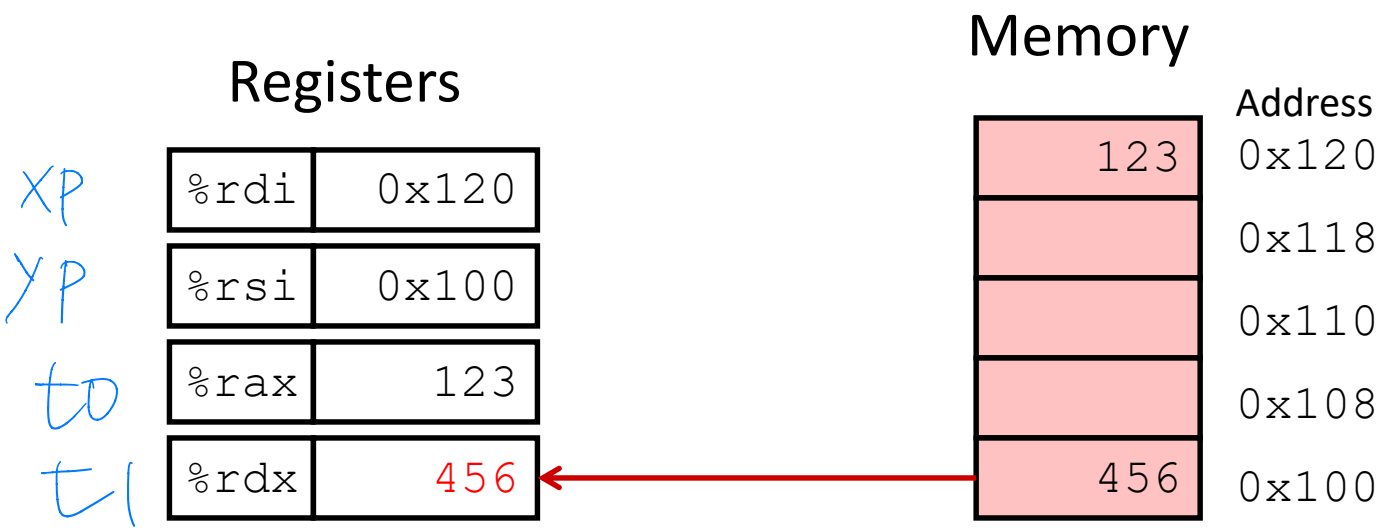

```
swap:
        movq    (%rdi), %rax  # t0 = *xp
        movq    (%rsi), %rdx  # t1 = *yp
        movq    %rdx, (%rdi)  # *xp = t1
        movq    %rax, (%rsi)  # *yp = t0
        ret
```

Understanding `Swap()`



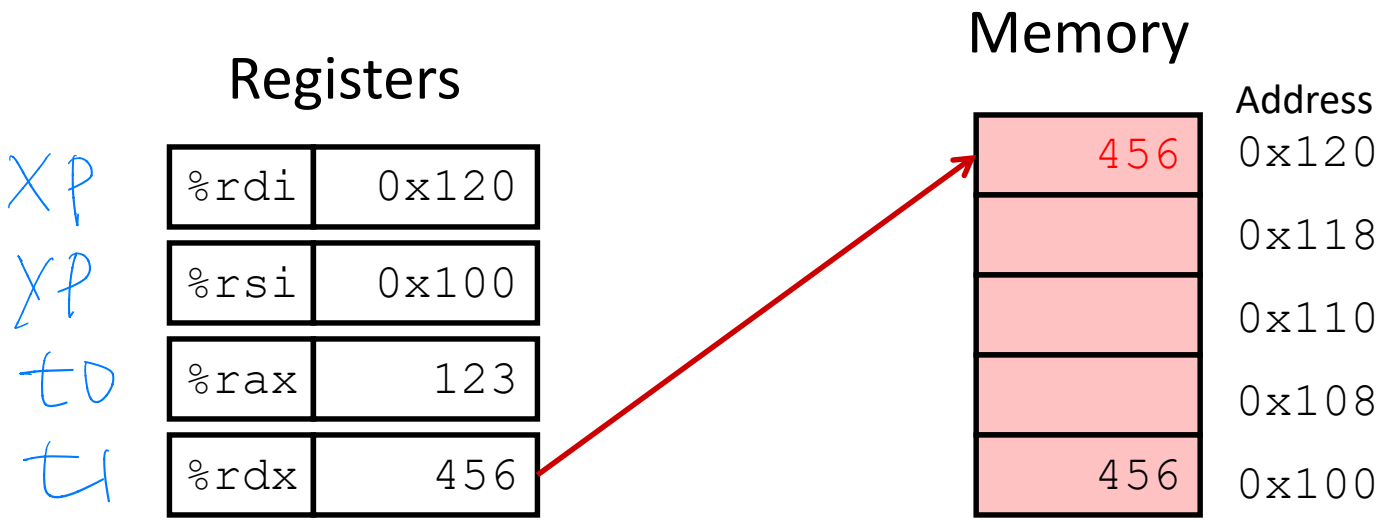

```
swap:
        movq    (%rdi), %rax  # t0 = *xp
        movq    (%rsi), %rdx  # t1 = *yp
        movq    %rdx, (%rdi)  # *xp = t1
        movq    %rax, (%rsi)  # *yp = t0
        ret
```

Understanding `Swap()`



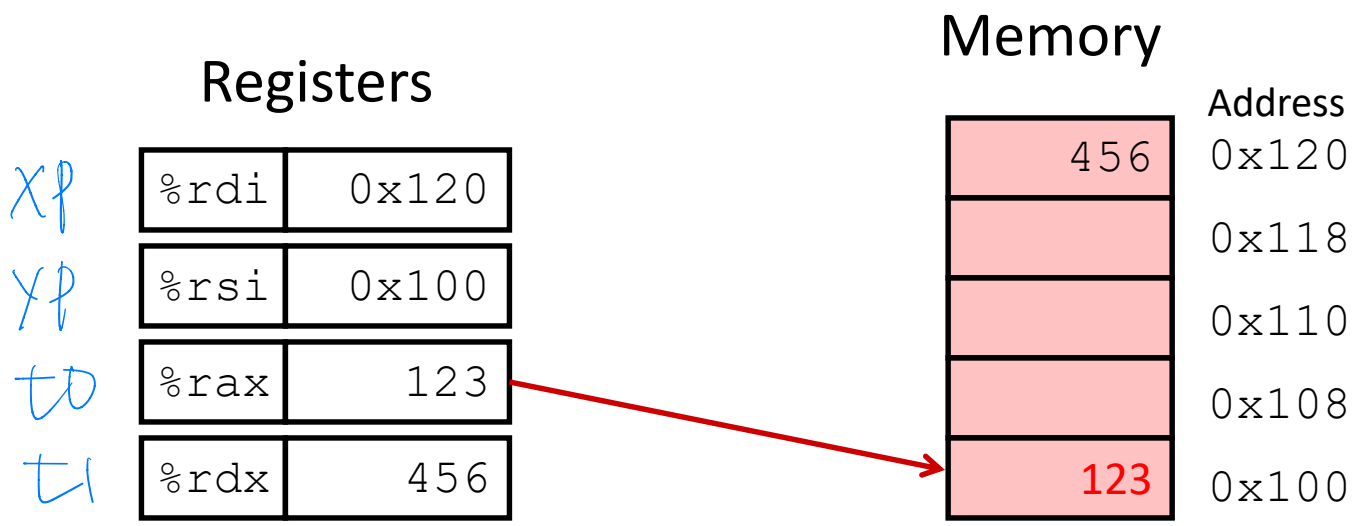

```
swap:
        movq    (%rdi), %rax  # t0 = *xp
        movq    (%rsi), %rdx  # t1 = *yp
        movq    %rdx, (%rdi)  # *xp = t1
        movq    %rax, (%rsi)  # *yp = t0
        ret
```

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

주소값 연산 방식.

- **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 시작점.
- Ri: Index register: Any, except for **`%rsp`** 몇번째 element ?
- S: Scale: 1, 2, 4, or 8 (*why these numbers?*)

size of data types.
such as char, int, long, and etc.

Specific Instances

(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

$D(Rb, Ri, S) = Mem[Reg[Rb] + S \cdot Reg[Ri] + D]$

2 x 1111 0000 0000 0000 + 1000 0000

1 1110 0000 1000 0000

0001 1110 0000 1000 0000

Expression	Address
0x8(%rdx)	Q1
(%rdx,%rcx)	Q2
(%rdx,%rcx,4)	Q3
0x80(,%rdx,2)	Q4

$0xf000 + 0x8 = 0xf008$

$0xf000 + 0x0100 = 0xf100$

$0xf000 + 4 \cdot 0x0100 = 0xf400$

$2 \cdot 0xf000 + 0x80 = 0x1e080$

Today: Machine Programming I: Basics

- History of Intel processors and architectures
- C, assembly, machine code
- Assembly Basics: Registers, operands, move
- **Arithmetic & logical operations**

Address Computation Instruction

low effective address

- **`leaq` *Src*, *Dst***
  - *Src* is address mode expression
  - Set *Dst* to address denoted by expression

- **Uses**
  - Computing addresses
    - E.g., translation of `p = &x[i];`
  - Computing arithmetic expressions of the form x + k*y + d (**leaq d(x, y, k), Dst**)
    - 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 =
```

(1+2) X

= 4X

= 12 X

Some Arithmetic Operations

- **Two Operand Instructions:**

suffix q => qword => 64 bits

cf)
byte = 8 bits
word = 16 bits
dword = 32 bits
qword = 64 bits

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	

shlq

shift - arithmetic - left/right
shift – logical – left/right

- **Watch out for argument order!**

Src first

Some Arithmetic Operations

- **One Operand Instructions**

Increase	`incq`	*Dest*	*Dest = Dest + 1*
decrease	`decq`	*Dest*	*Dest = Dest – 1*
negative	`negq`	*Dest*	*Dest = – Dest*
not	`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
```

constant

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
```

t3

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

Machine Programming: 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