Jin-Soo Kim
(jinsoo.kim@snu.ac.kr)

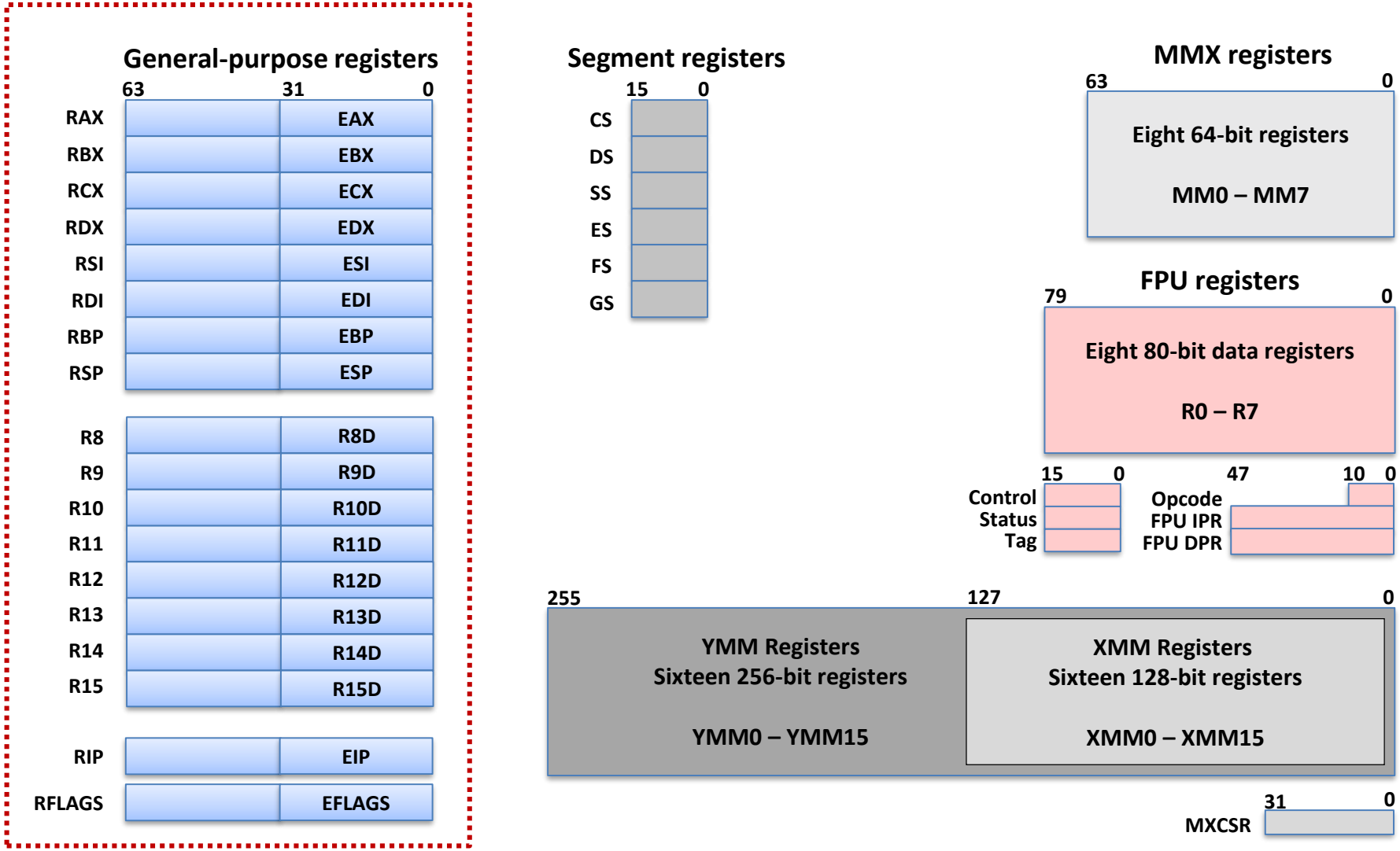
Systems Software
Research Lab.

Seoul National University

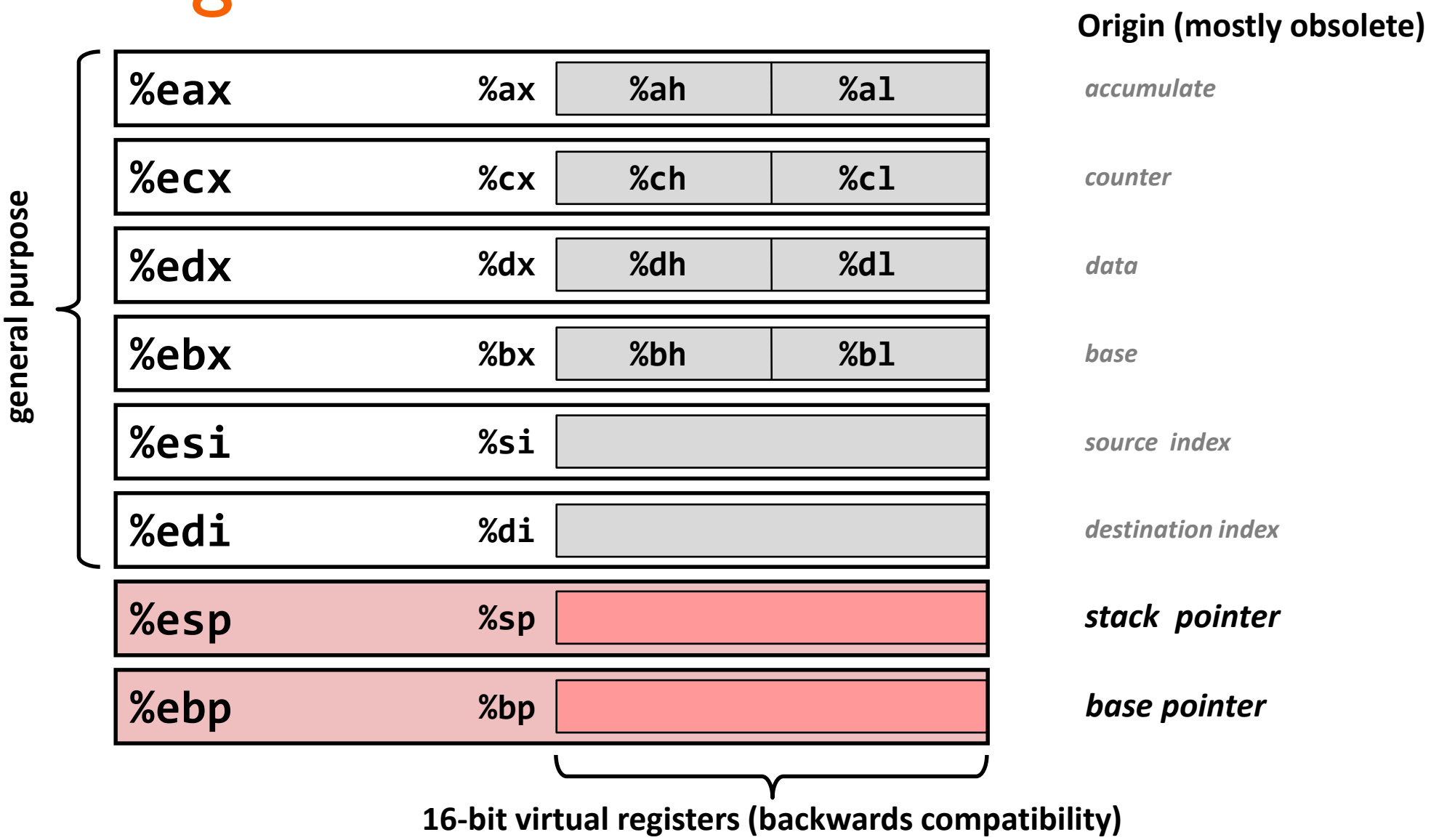
Spring 2018

Assembly I: Basic Operations

Basic Execution Environment



IA-32 Registers



Moving Data (1)

- Moving data: **`movq`** ***`Src, Dest`***
  - Move 8-byte ("quad") word
- Operand types
  - Immediate: constant integer data
    - Like C constant, but prefixed with '**`$`**'
    - Encoded with 1, 2, or 4 bytes
    - e.g. **`$0x400`**, **`$-533`**
  - Register: one of 16 integer registers
    - But **`%rsp`** reserved for special use
    - Others have special uses for particular instructions
  - Memory: 8 consecutive bytes of memory
    - Various "addressing modes"

`%rax`
`%rbx`
`%rcx`
`%rdx`
`%rsi`
`%rdi`
`%rsp`
`%rbp`
`%r8`
`%r9`
`%r10`
`%r11`
`%r12`
`%r13`
`%r14`
`%r15`

Moving Data (2)

- **movq** operand combinations
  - Cannot do memory-memory transfer with a single instruction

	Source	Destination		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;

Simple Addressing Modes

- Normal `(R)` Mem[Reg[R]]
  - Register **R** specifies memory address
  - Pointer dereferencing in C
  - e.g. `movq (%rcx), %rax`

- Displacement `D(R)` Mem[Reg[R]+D]
  - Register **R** specifies start of memory region
  - Constant displacement **D** specifies offset
  - e.g. `movq 8(%rbp), %rdx`

General Addressing Modes

- **`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]]
  - Useful to access arrays and structures

Addressing Modes: Example

- Address computation

%rdx	0xf000
%rcx	0x0100

Expression	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

Swap Example

- Source code in C:

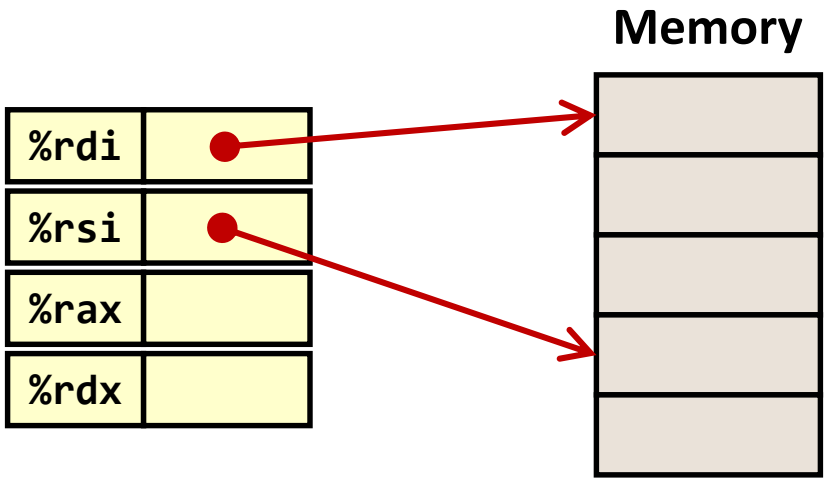
```
void swap(long *xp, long *yp)
{
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

- Corresponding assembly code:

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

Understanding Swap (1)

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



Register Allocation (By compiler)

Register	Variable
%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 (2)

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

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

Memory

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

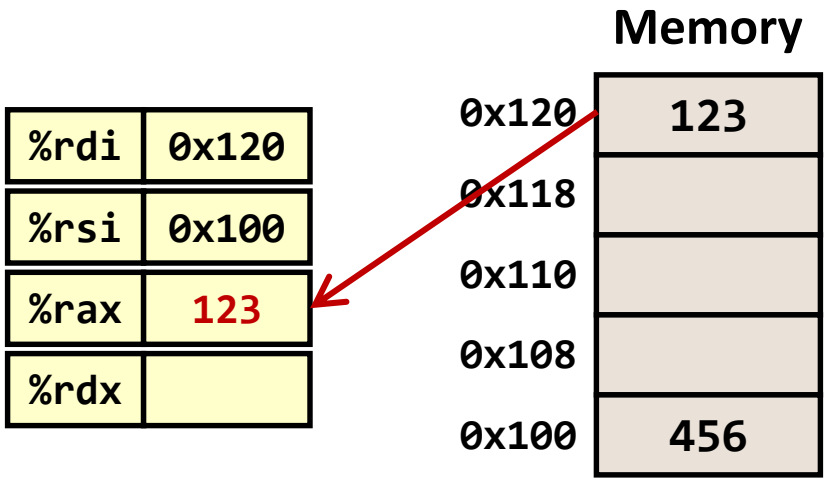
Register Allocation (By compiler)

Register	Variable
%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 (3)

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



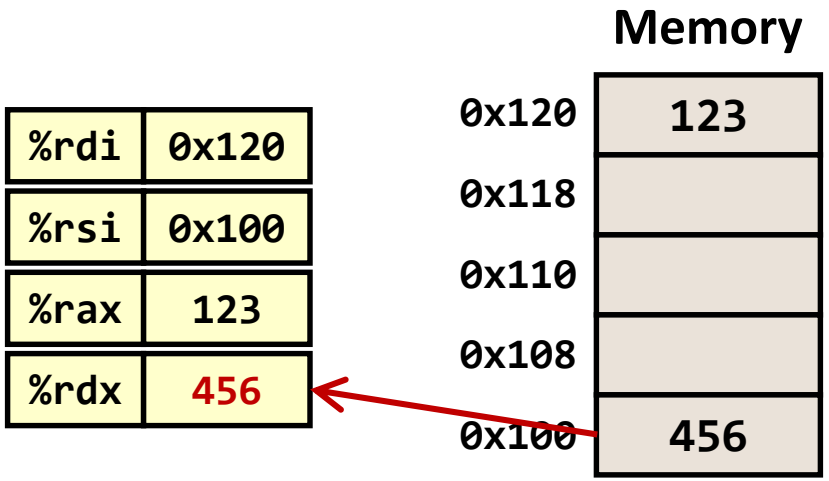
Register Allocation (By compiler)

Register	Variable
%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 (4)

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



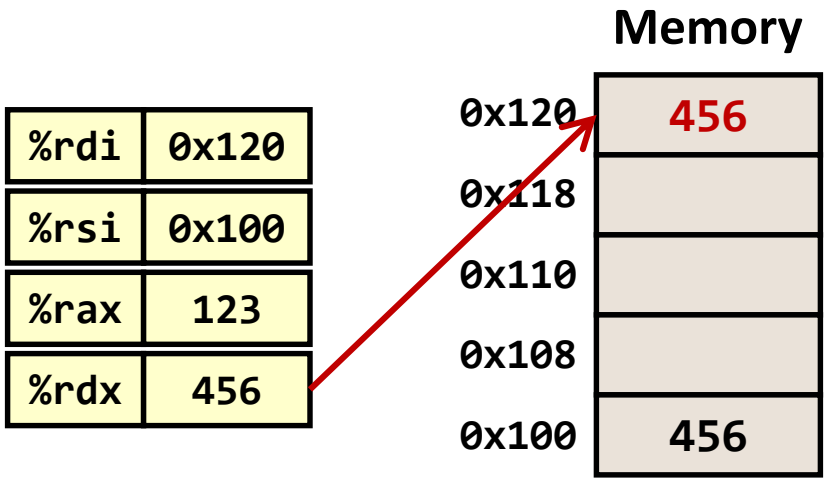
Register Allocation (By compiler)

Register	Variable
%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 (5)

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



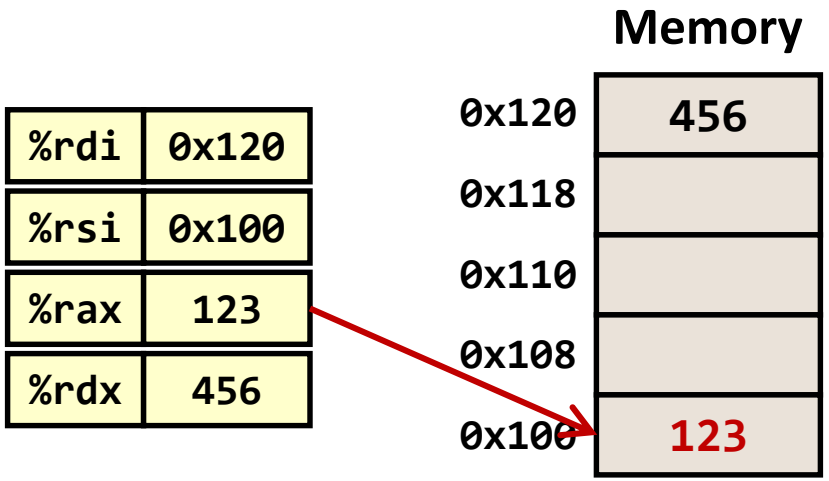
Register Allocation (By compiler)

Register	Variable
%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 (5)

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



Register Allocation (By compiler)

Register	Variable
%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
```

Arithmetic/Logical Operations (1)

- Two operand instructions
  - **`addq`** Src, Dest — Dest = Dest + Src
  - **`subq`** Src, Dest — Dest = Dest – Src
  - **`mulq`** Src, Dest — Dest = Dest * Src (unsigned)
  - **`imulq`** Src, Dest — Dest = Dest * Src (signed)
  - **`salq`** Src, Dest — Dest = Dest << Src (= **`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

Arithmetic/Logical Operations (2)

- One operand instructions
  - `incq` Dest Dest = Dest + 1
  - `decq` Dest Dest = Dest – 1
  - `negq` Dest Dest = – Dest
  - `notq` Dest Dest = ~ Dest

- See books for more instructions

Address Computation Instruction

- **`leal` *`Src, Dest`***
  - ***`Src`*** is address mode expression
  - Set ***`Dest`*** to address denoted by expression
- Uses
  - Computing address without a memory reference
    - e.g. translation of `p = &x[i];`
  - Computing arithmetic expression of the form `x + k*y`
    - k = 1, 2, 4, or 8

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

```
leaq  (%rdi, %rdi, 2), %rax
salq  $2, %rax
```

Example: arith

```
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;
}
```

```
x in %rdi
y in %rsi
z in %rdx
```

```
arith:
    leaq   (%rdi, %rsi), %rax        # %rax = x + y (t1)
    addq   %rdx, %rax                # %rax = z + t1 (t2)
    leaq   (%rsi, %rsi, 2), %rdx     # %rdx = 3 * y
    salq   $4, %rdx                  # %rdx = 48 * y (t4)
    leaq   4(%rdi, %rdx), %rcx       # %rcx = x + t4 + 4 (t5)
    imulq  %rcx, %rax                # %rax = t5 * t2 (rval)
    ret
```

Example: logical

```
int logical (int x, int y) {
    int t1 = x^y;
    int t2 = t1 >> 17;
    int mask = (1 << 13) - 7;
    int rval = t2 & mask;
    return rval;
}
```

x in %edi
y in %esi

```
logical:
    movl  %edi, %eax          # %eax = x
    xorl  %esi, %eax          # %eax = x ^ y (t1)
    sarl  $17, %eax           # %eax = t1 >> 17 (t2)
    andl  $8185, %eax         # %eax = t2 & 8185 (rval)
    ret
```

CISC Properties

- Complex Instruction Set Computer (CISC)
  - Instruction can reference different operand types
    - Immediate, register, memory
  - Arithmetic operations can read/write memory
    - Source or destination can be a memory address
  - Memory reference can involve complex computation
    - Rb + S * Ri + D
    - Useful for arithmetic expressions, too
  - Instructions can have varying lengths
    - x86-64 instructions can range from 1 to 15 bytes

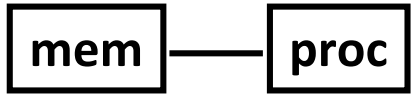
Machine-level Programming

- Assembly code is textual form of binary object code
- Low-level representation of program
  - Explicit manipulation of registers
  - Simple and explicit instructions
  - Minimal concept of data types
  - Many C control constructs must be implemented with multiple instructions

Summary

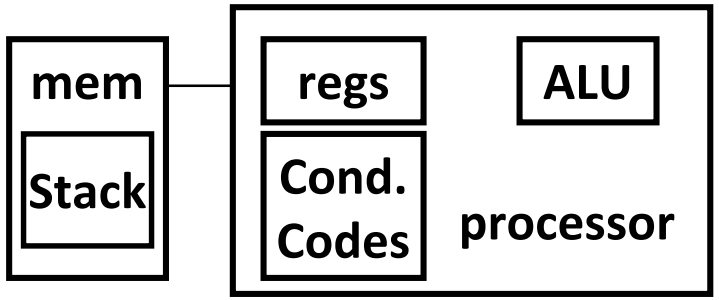
Machine Models

C



Compiler

Assembly



Data

1) char
2) int, float
3) double
4) struct, array
5) pointer

1) byte
2) 2-byte word
3) 4-byte long word
4) 8-byte quad word
5) contiguous byte allocation
6) address of initial byte

Control

1) loops
2) conditionals
3) switch
4) Proc. call
5) Proc. return

1) branch/jump
2) call
3) ret