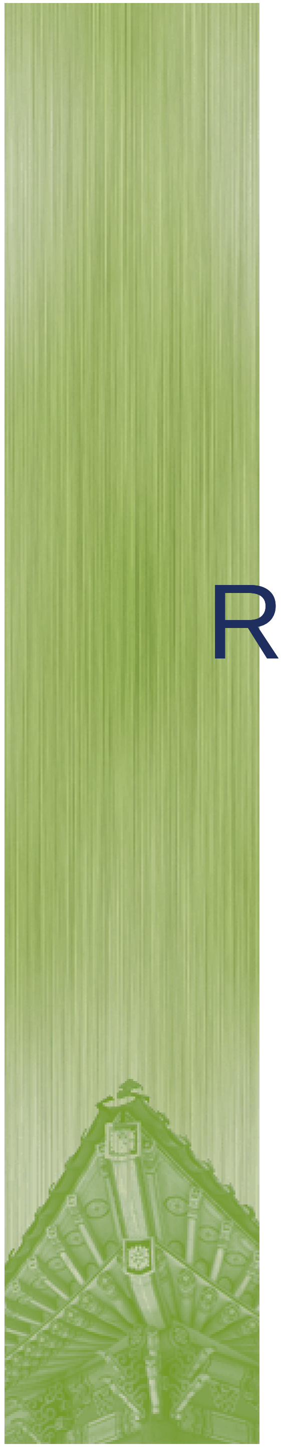
☆ AT&T S → D



[Chap.3-3] Machine-level Representation of Programs

Young Ik Eom (yieom@skku.edu, 031-290-7120)
Distributing Computing Laboratory
Sungkyunkwan University
http://dclab.skku.ac.kr

Contents

- Introduction
- Program encodings
- Data formats
- Intel processors
- Accessing information
- Primitive instructions
- Data movement instructions
- Arithmetic and logic instructions
- Control instructions
- Procedures
- …

Data Movement Instructions

3 classes of instructions

- **MOV** class
  - **movb**, **movw**, **movl**, **movq** (moving 1, 2, 4, and 8 bytes, respectively)
  - Copy their source values to their destinations
    - ✓ Immediate, register, memory for sources
    - ✓ Register or memory for destinations
- **MOVS** class
  - Copy a smaller amount of data to a larger data locations
  - Fill the upper bits by sign expansion
- **MOVZ** class
  - Copy a smaller amount of data to a larger data locations
  - Fill the upper bits by zero expansion

- Memory-to-memory data transfer is not allowed in x86-64

Data Movement Instructions

MOV class

Instruction		Effect	Description
MOV	S,D	D ← S	Move
movb	Mem→Mem은 X		Move byte
movw			Move word
movl			Move double word
movq			Move quad word
movabsq	I,R ☆	R ← I	Move absolute quad word

- Note)
  - The regular **movq** instruction can only have 32-bit 2's complement immediate source operands; The value is then sign-extended to produce 64-bit value for the destination
  - The **movabsq** instruction can have an arbitrary 64-bit immediate value as its source operand and can only have a register as a destination

Data Movement Instructions

- **5 possible combinations of src and dst types**
  - M-to-M data transfer is not allowed with 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;**

Data Movement Instructions

- Example)

`movl $0x4050,%eax`	Immediate → Register, 4 bytes
`movb $-17,(%rsp)`	Immediate → Memory, 1 byte
`movw %bp,%sp`	Register → Register, 2 bytes
`movq %rax,-12(%rbp)`	Register → Memory, 8 bytes
`movb (%rdi,%rcx),%al`	Memory → Register, 1 byte

Data Movement Instructions

■ MOVZ class

Instruction	Effect	Description
`MOVZ S,R`	R ← ZE(S)	Move with zero extension
`movzbw`		Move zero-extended byte to word
`movzbl`		Move zero-extended byte to double word
`movzwl`		Move zero-extended word to double word
`movzbq`		Move zero-extended byte to quad word
`movzwq`		Move zero-extended word to quad word

- No **`movzlq`** instruction !!!
  - Can be implemented with **`movl`** instruction

Data Movement Instructions

MOVS class

Instruction	Effect	Description
`MOVS S,R` ☆	`R ← SE(S)`	Move with sign extension
`movsbw`		Move sign-extended byte to word
`movsbl`		Move sign-extended byte to double word
`movswl`		Move sign-extended word to double word
`movsbq`		Move sign-extended byte to quad word
`movswq`		Move sign-extended word to quad word
`movslq`		Move sign-extended double word to quad word
`cltq` → convert double word to quad word	`%rax ← SE(%eax)`	Sign-extend **%eax** to **%rax**

Data Movement Instructions

- **Example)**

```
movabsq $0x0011223344556677,%rax   %rax = 0011223344556677
movb $0xAA,%dl                     %dl  = AA
movb %dl,%al                       %rax = 00112233445566AA
movsbq %dl,%rax                    %rax = FFFFFFFFFFFFFFAA
movzbq %dl,%rax                    %rax = 00000000000000AA
```

Data Movement Instructions

Example) Function testandset()

```
[C code]

long testandset(long *xp, long y)
{
   long x = *xp;
   *xp = y;
   return x;
}
```

```
[Assembly code]

   long testandset(long *xp, long y)
   xp in %rdi, y in %rsi

testandset:
   movq (%rdi),%rax   Get x at xp; Set as return value
   movq %rsi,(%rdi)   Store y at xp
   ret                Return
```

Data Movement Instructions

Example) Function swap()

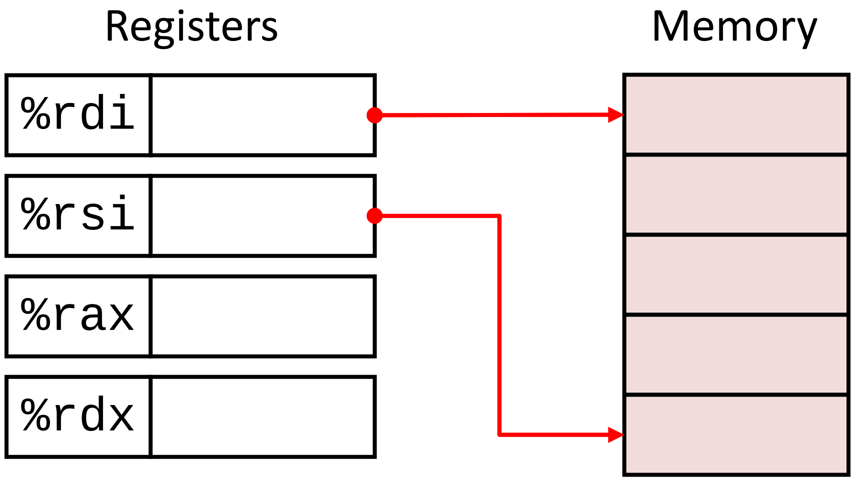
```
[C code]
void swap(long *xp, long *yp)
{
   long t0 = *xp;
   long t1 = *yp;
   *xp = t1;
   *yp = t0;
}
```

```
[Assembly code]
   void swap(long *xp, long *yp)
   xp in %rdi, yp in %rsi
swap:
   movq (%rdi),%rax
   movq (%rsi),%rdx
   movq %rdx,(%rdi)
   movq %rax,(%rsi)
   ret
```

Data Movement Instructions

Example) Function swap()

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

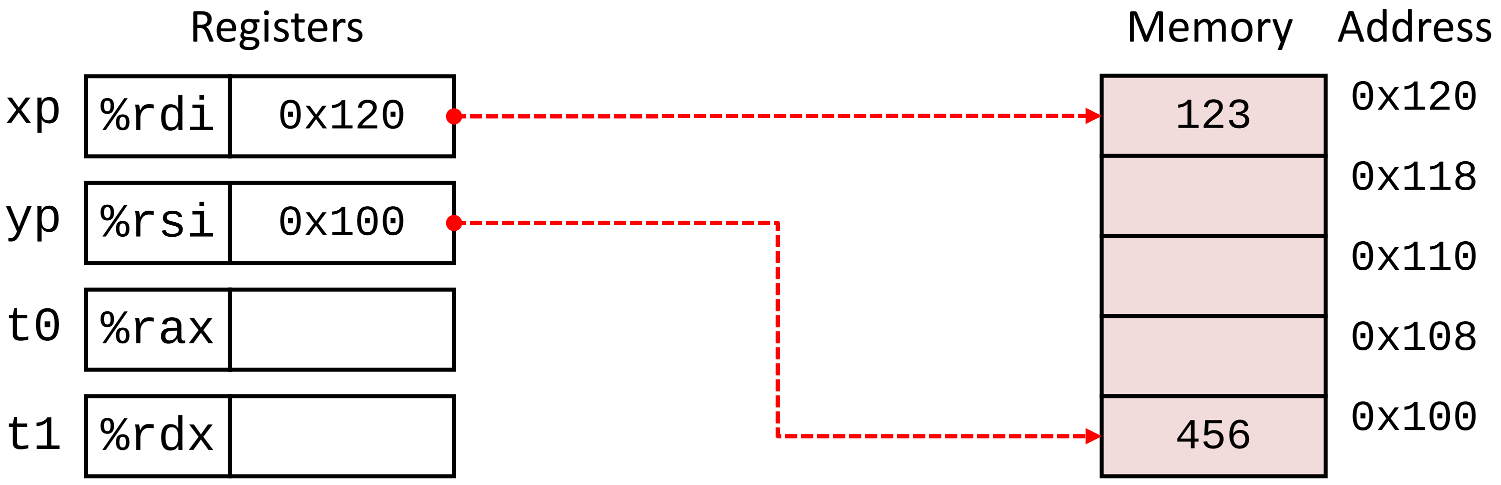


Register	Value
%rdi	xp
%rsi	yp
%rax	t0
%rdx	t1

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

Data Movement Instructions

Example) Function swap()



Registers

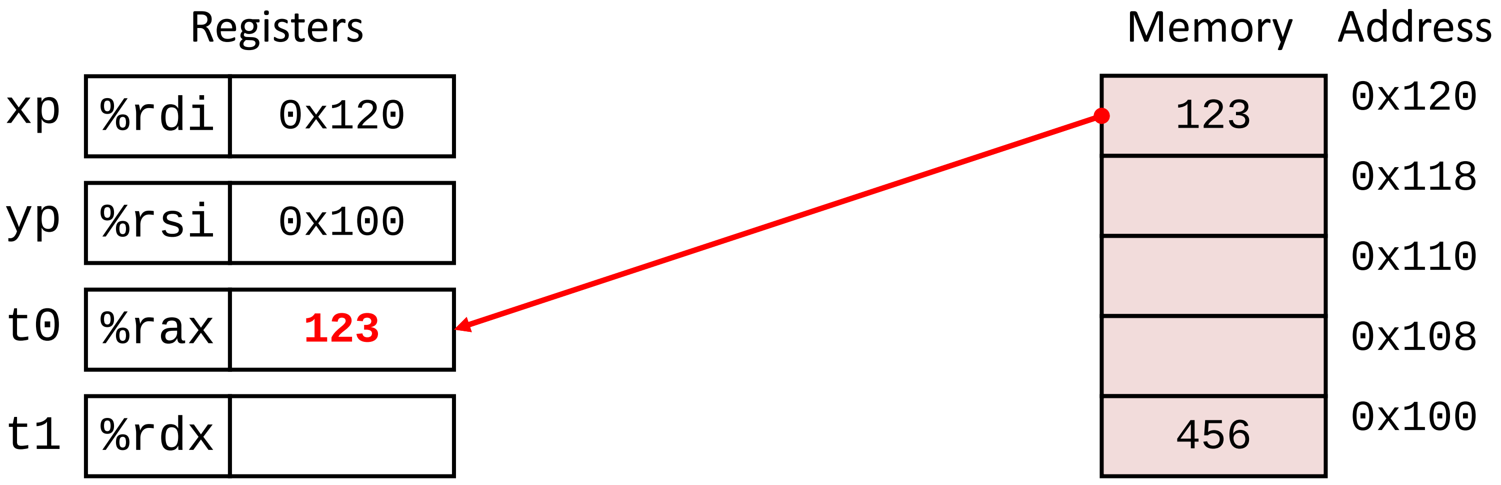
xp	%rdi	0x120
yp	%rsi	0x100
t0	%rax	
t1	%rdx	

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
```

Data Movement Instructions

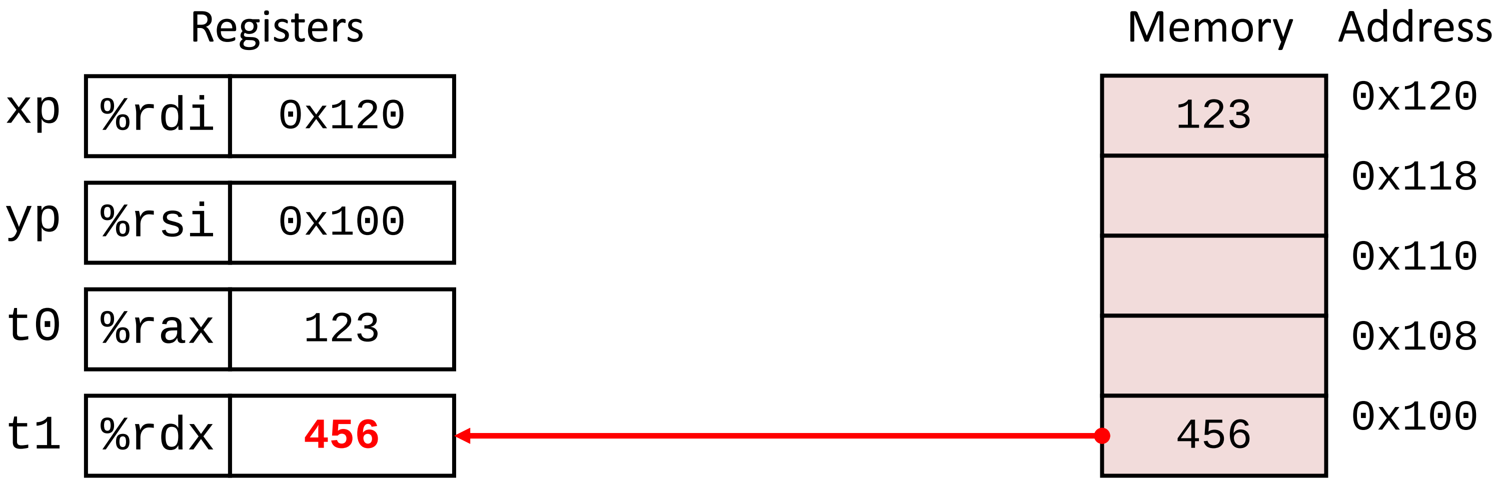
Example) Function swap()



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

Data Movement Instructions

Example) Function swap()



Registers

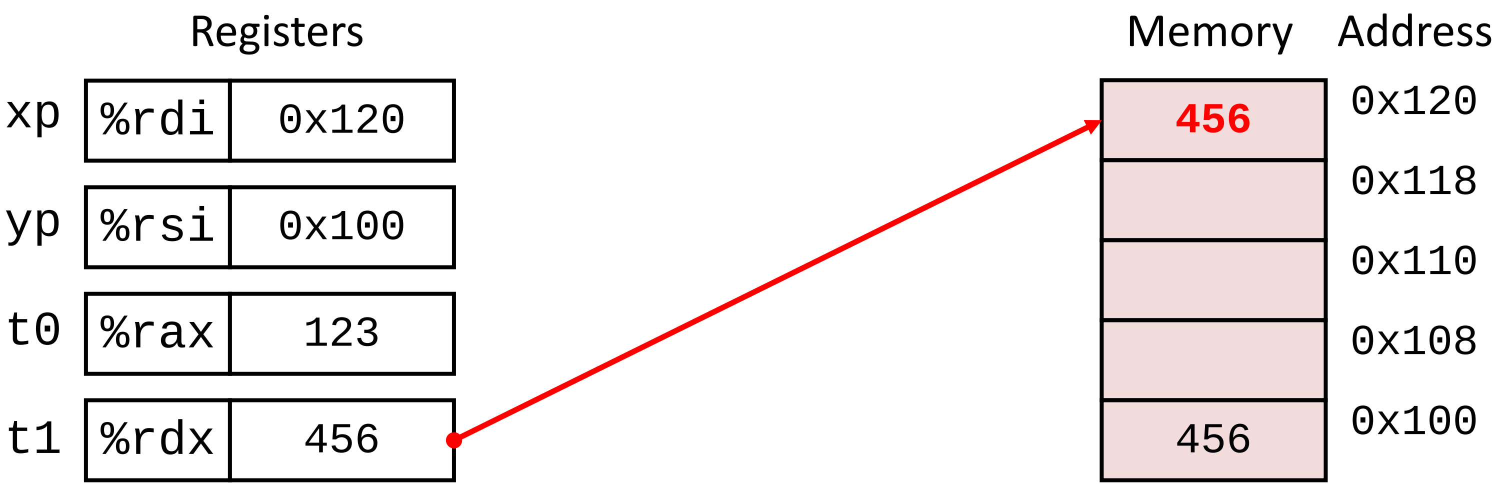
xp	%rdi	0x120
yp	%rsi	0x100
t0	%rax	123
t1	%rdx	456

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
```

Data Movement Instructions

■ Example) Function swap()



Registers

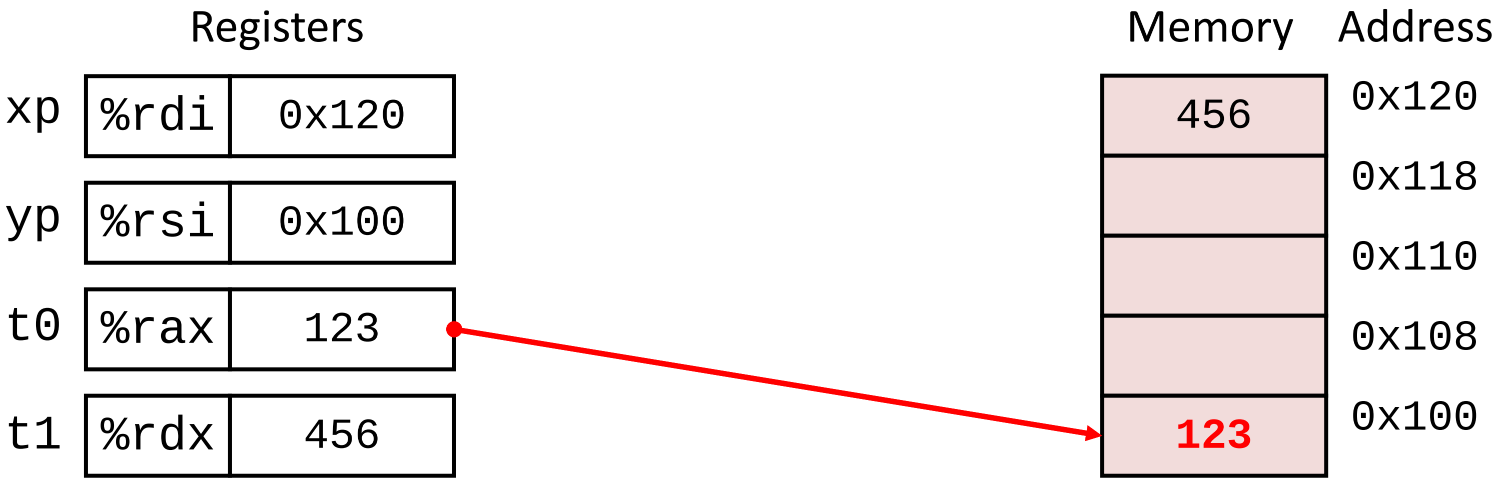
xp	%rdi	0x120
yp	%rsi	0x100
t0	%rax	123
t1	%rdx	456

Memory	Address
456	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
```

Data Movement Instructions

Example) Function swap()



Registers

xp	%rdi	0x120
yp	%rsi	0x100
t0	%rax	123
t1	%rdx	456

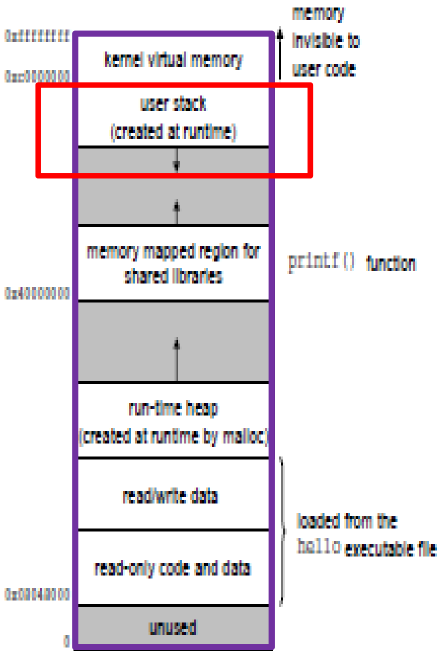
Memory	Address
456	0x120
	0x118
	0x110
	0x108
123	0x100

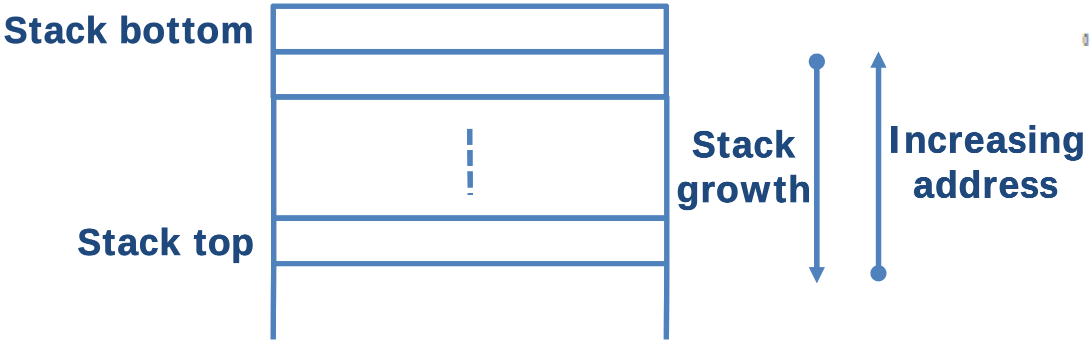
```
swap:
   movq     (%rdi),%rax        # t0 = *xp
   movq     (%rsi),%rdx        # t1 = *yp
   movq     %rdx,(%rdi)        # *xp = t1
   movq     %rax,(%rsi)        # *yp = t0
   ret
```

Data Movement Instructions

Pushing and popping stack data

- Stack
  - LIFO (Last-In-First-Out) data structure
  - Insert (push) and delete (pop) at the **top** of the stack
  - In x86-64
    - ✓ Program stack in some region of memory
    - ✓ Grows downward (top at the lowest address)
    - ✓ Stack pointer **%rsp** holds the address of top element





Stack bottom

Stack top

Stack growth

Increasing address

Data Movement Instructions

- Push and pop instructions

Instruction	Effect	Description

`pushq    S`	`R[%rsp] ← R[%rsp]-8;`
	`M[R[%rsp]] ← S`
		Push quad word

`popq     D`	`D ← M[R[%rsp]];`
	`R[%rsp] ← R[%rsp] + 8`
		Pop quad word

Data Movement Instructions

■ Push and pop instructions

Initially

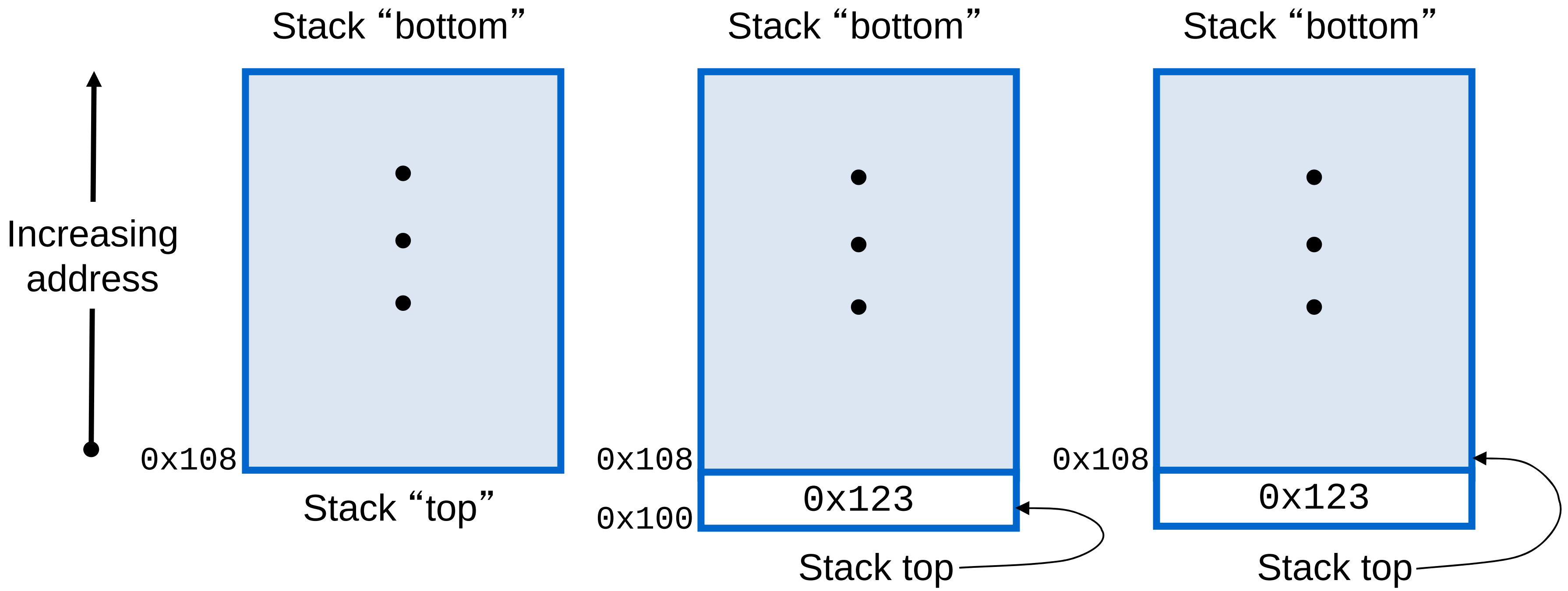
%rax	0x123
%rdx	0
%rsp	0x108

pushq %rax

%rax	0x123
%rdx	0
%rsp	0x100

popq %rdx

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



Data Movement Instructions

Push and pop instructions

- Example)

```
pushq %rbp
```

≡

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

1-byte instruction

Total 8-bytes

```
popq %rax
```

≡

```
movq (%rsp),%rax
addq $8,%rsp
```

Arithmetic/Logic Instructions

- **Integer arithmetic/logic instructions**
  - 4 classes
    - Load effective address (**leaq**)
    - Unary
    - Binary
    - Shift
  - Each unary, binary, and shift instructions have different variants with different operand sizes (except **leaq**)
    - Byte, word, double word, quad word
    - Eg) **addb**, **addw**, **addl**, **addq**

Arithmetic/Logic Instructions

Integer arithmetic/logic instructions

Instruction	Effect	Description	
`leaq S,D` register	D ← &S	Load effective address	LEA
`INC D`	D ← D + 1	Increment	Unary
`DEC D`	D ← D - 1	Decrement	
`NEG D`	D ← -D	Negate	
`NOT D`	D ← ~D	Complement	
`ADD S,D`	D ← D + S	Add	Binary
`SUB S,D`	D ← D - S	Subtract	
`IMUL S,D`	D ← D * S	Multiply	
`XOR S,D`	D ← D ^ S	Exclusive-or	
`OR S,D`	D ← D | S	Or	
`AND S,D`	D ← D & S	And	
`SAL k,D`	D ← D << k	Left shift	Shift
`SHL k,D`	D ← D << k	Left shift (same as **SAL**)	
`SAR k,D`	D ← D >>$_A$ k	Arithmetic right shift	
`SHR k,D`	D ← D >>$_L$ k	Logical right shift	

Arithmetic/Logic Instructions

Integer arithmetic/logic instructions

- **leaq** instruction
  - "Load effective address" instruction
  - Copies the effective address of the source operand to the destination
    - ✓ No memory access ☆
    - ✓ The destination operand must be a register

Arithmetic/Logic Instructions

Integer arithmetic/logic instructions

- **leaq** instruction
  - Usage
    - ✓ To generate pointers (for later memory references)
      Eg) `movq 8(%rbx,%rcx,4),%rax`
      `leaq 8(%rbx,%rcx,4),%rax`
    - ✓ To compactly describe some arithmetic operations
      Eg) Assume **`%edx`** has value **x**, then the instruction
      `leaq 7(%rdx,%rdx,4),%rax`
      sets the value of register **`%rax`** to **5·x+7**

Arithmetic/Logic Instructions

- Example) Function scale()

```
[C code]

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

```
[Assembly code]

   long scale(long x, long y, long z)
   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) = x + 4*y + 12*z
   ret
```

Arithmetic/Logic Instructions

Integer arithmetic/logic instructions

- Unary instructions
  - Single operand
    - ✓ The operand can be either a register or a memory location
  - Eg) **`incq (%rax)`** vs incq %rax
- Binary instructions
  - Two operands
    - ✓ First operand (source)
      - An immediate value, a register, or a memory location
    - ✓ Second operands (both source and destination)
      - A register or a memory location
    - ✓ The two operands cannot both be memory locations
  - Eg) **`subq %rax,%rdx`**

Arithmetic/Logic Instructions

Integer arithmetic/logic instructions

- Shift instructions
  - Two operands
    - ✓ First operand is shift amount
      - Immediate value or **`%cl`** (1 byte, max 255)
      - With x86-64, a shift instruction operating on data values that are ω bits long determines the shift amount from the low-order m bits of register **`%cl`**, where $2^m = \omega$
    - ✓ Second operand is the value to be shifted
      - In register or memory
  - Logical or arithmetic

Arithmetic/Logic Instructions

- **Example) Function arith()**

```
[C code]

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;
}
```

Arithmetic/Logic Instructions

Example) Function arith() (Cont'd)

[Assembly code]

```
   long arith(long x, long y, long z)
   x in %rdi, y in %rsi, z in %rdx

arith:
   xorq %rsi,%rdi                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
```

[C code]

```
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;
}
```

Arithmetic/Logic Instructions

Special arithmetic instructions

- 64-bit multiplication for 128-bit product
- Integer division
  - Generates quotient and remainder

Arithmetic/Logic Instructions

- Special arithmetic instructions

Instruction	Effect	Description
`imulq S`	`R[%rdx]:R[%rax] ← S × R[%rax]`	Signed full multiply
`mulq S`	`R[%rdx]:R[%rax] ← S × R[%rax]`	Unsigned full multiply
`cqto`	`R[%rdx]:R[%rax] ← SE(R[%rax])`	Convert to oct word

`idivq S`	`R[%rdx] ← R[%rdx]:R[%rax] mod S`
	`R[%rax] ← R[%rdx]:R[%rax] ÷ S`
		Signed divide

`divq S`	`R[%rdx] ← R[%rdx]:R[%rax] mod S`
	`R[%rax] ← R[%rdx]:R[%rax] ÷ S`
		Unsigned divide

Summary