

CS 261 Fall 2022

Mike Lam, Professor



$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

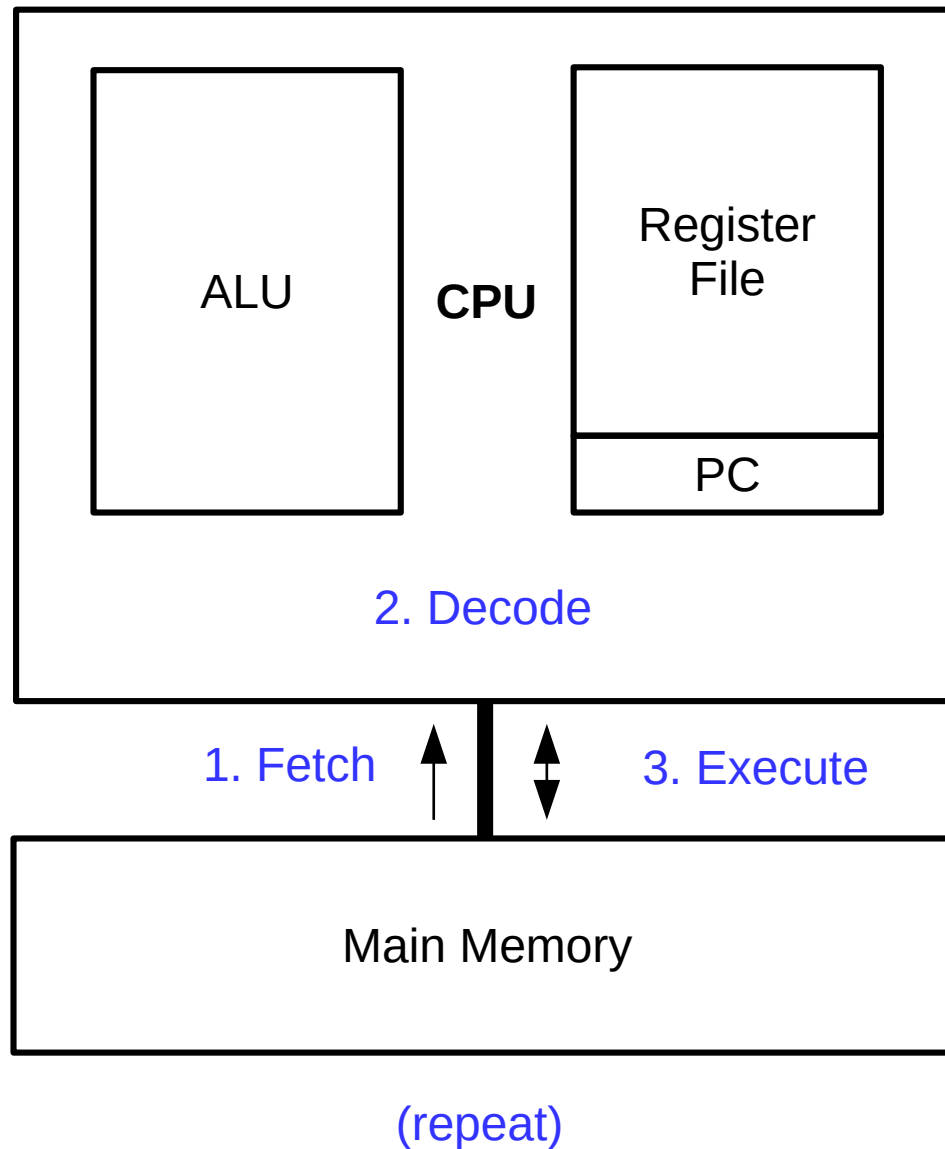
$$-\nabla p + \nabla \cdot \boldsymbol{\tau} + \rho \mathbf{g}$$

x86-64 Data Movement and Arithmetic

Topics

- Data movement
- Instruction validity
- Stack operations
- Arithmetic and logical operations

von Neumann architecture



Data movement

- Primary data movement instruction: "mov"
 - **Copies** data from first operand to second operand
- There are no “types” in assembly code
 - You must know how many bytes you want to move
 - Information = Bits + Context
 - Often, a “**class**” of machine instructions (e.g., “mov__”) will perform similar operations on different sizes of data
- Historical artifact: “**word**” in x86 is 16 bits
 - 1 byte (8 bits) = “byte” (**b** suffix)
 - 2 bytes (16 bits) = “word” (**w** suffix)
 - 4 bytes (32 bits) = “double/long word” (**l** suffix)
 - 8 bytes (64 bits) = “quad word” (**q** suffix)

Data movement

- Primary data movement instruction: "mov"
 - **Copies** data from first operand to second operand
 - Multiple suffixes:
 - mov**b**, mov**w**, mov**l**, mov**q**, movabs**q**
 - movabsq is the only form that takes a 64-bit immediate
- Zero-extension variant: "movz"
 - movz**bw**, movz**bl**, movz**wl**, movz**bq**, movz**wq**
 - Note lack of movz**lq**; just use mov**l**, which sets higher 32-bits to zero
- Sign-extension variant: "movs"
 - movs**bw**, movs**bl**, movs**wl**, movs**bq**, movs**wq**, movs**lq**
 - ↑ ↑
 - byte-to-word

Registers

- Multiple names per register
 - Refers to different data sizes
 - **eXX** = lower 32-bits (e.g., **eax**)
 - **rXX** = full 64 bits (e.g., **rax**)
- Instruction suffixes and operand sizes must match!
 - E.g., `movq $1, %rax` is valid but `movq $1, %eax` is not

register encoding	zero-extended for 32-bit operands	not modified for 16-bit operands	not modified for 8-bit operands	low 8-bit	16-bit	32-bit	64-bit
0			AH*	AL	AX	EAX	RAX
3			BH*	BL	BX	EBX	RBX
1			CH*	CL	CX	ECX	RCX
2			DH*	DL	DX	EDX	RDY
6				SIL**	SI	ESI	RSI
7				DIL**	DI	EDI	RDI
5				BPL**	BP	EBP	RBP
4				SPL**	SP	ESP	RSP
8				R8B	R8W	R8D	R8
9				R9B	R9W	R9D	R9
10				R10B	R10W	R10D	R10
11				R11B	R11W	R11D	R11
12				R12B	R12W	R12D	R12
13				R13B	R13W	R13D	R13
14				R14B	R14W	R14D	R14
15				R15B	R15W	R15D	R15

63	32	31	16	15	8	7	0
----	----	----	----	----	---	---	---

63	32	31	0
----	----	----	---

RFLAGS
RIP

515-309.eps

* Not addressable when a REX prefix is used.
** Only addressable when a REX prefix is used.

Memory addressing modes

- Absolute: *addr*
 - Effective address: *addr*
- Indirect: (*reg*)
 - Effective address: $R[reg]$
- Base + displacement: *offset*(*reg*)
 - Effective address: $offset + R[reg]$
- Indexed: *offset*(reg_{base} , reg_{index})
 - Effective address: $offset + R[reg_{base}] + R[reg_{index}]$
- Scaled indexed: *offset*(reg_{base} , reg_{index} , *s*)
 - Effective address: $offset + R[reg_{base}] + R[reg_{index}] \cdot s$
 - Scale (*s*) must be 1, 2, 4, or 8

$R[reg]$ = value of register *reg*

useful for
pointers!

useful for
arrays!

(also, note that
offset and reg_{base}
are optional here)

Memory operands

- Addresses in x86-64 are always 32 or 64 bits
 - Thus, the registers used to calculate the effective address of a memory operand must be 32 or 64 bits
 - E.g., `movw %ax, (%ebp)` is valid
 - E.g., `movw %ax, (%rbp)` is valid
 - E.g., `movw %ax, (%bp)` is **not valid!**
 - E.g., `movw %ax, %rbp` is **not valid!**
- The size of data moved is determined by the size of the register operand or the instruction suffix
 - NOT the size of the register(s) used to calculate the effective address
 - Memory locations have no “type” in assembly/machine code

Validity summary

- Is an instruction valid?
 - Is the opcode valid?
 - Are all of the operands valid?
 - For immediate operands, is it a source register?
 - (cannot write to immediates!)
 - For register operands, is it a valid register?
 - (and does it match the width suffix?)
 - For memory operands, is it a valid addressing mode?
 - (and are all registers used 32- or 64-bits?)

Question

- Which of the following are valid x86-64 movement instructions?
 - A) `movb %eax, %ecx`
 - B) `movl %eax, %ecx`
 - C) `movl $8, %edx`
 - D) `movl $8, %rdx`
 - E) `movw $0x24, 0x4(%rsp)`
 - F) `movl $0x24, 0x4(%esp)`

Aside: suffixes

- Is the operand size suffix mandatory?
 - E.g., the "l" or "q" in "movl" or "movq"
- Technically, it is only required if it cannot be inferred
 - E.g., `mov %eax, %edi` is not ambiguous
 - We can infer that this is a 32-bit move because of the destination
 - However, `mov $2, (%rdx)` is ambiguous
 - Is it a 8-bit move? 32 bits? 64 bits?
 - A suffix is required here (e.g., `movl $2, (%rdx)` for 32 bits)
 - Generally, it is safer always to include the suffix

Question

- T/F: "movl (%rax), (%rdx)" is a valid x86-64 assembly instruction.

Aside: memory operands

- In x86-64, most opcodes have no memory -> memory form
 - You can't encode two memory operands in the same instruction
 - Invalid: `movl (%rax), (%rdx)`

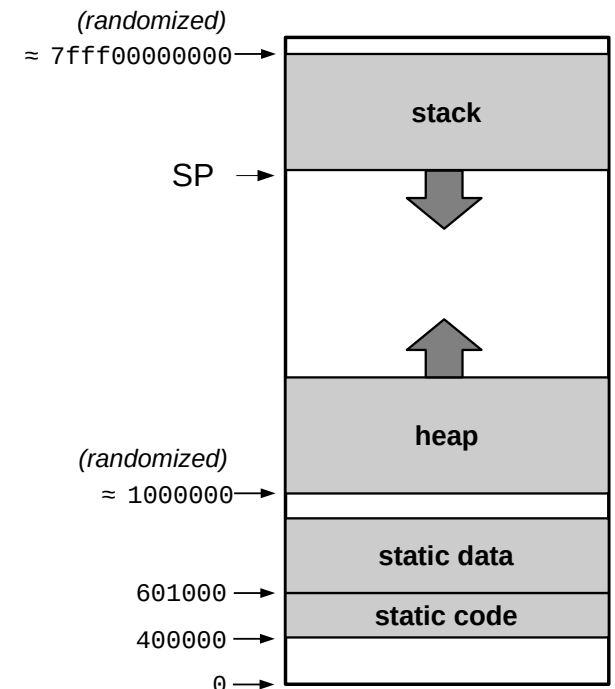
- Solution: use a temporary register

```
movl (%rax), %ecx
```

```
movl %ecx, (%rdx)
```

Stack operations

- The **system stack** holds 8-byte (quadword) slots, growing downward from high addresses to low addresses
 - **Stack Pointer** (SP) register stores address of "top" of stack
 - i.e., a pointer to the last value pushed (**lowest** address)
 - On x86-64, it is `%rsp` b/c addresses are 64 bits
 - `pushq <reg>` instruction
 - Subtract 8 from stack pointer
 - Store value of `<reg>` at `(%rsp)`
 - `popq <reg>` instruction
 - Retrieve value at `(%rsp)`
 - Save value in the given register
 - Increment stack pointer by 8



Exercise

- Given the following register state, what will the values of the registers be after the following instruction sequence?

- pushq %rax
- pushq %rcx
- pushq %rbx
- pushq %rdx
- popq %rax
- popq %rbx
- popq %rcx
- popq %rdx

Registers

<u>Name</u>	<u>Value</u>
%rax	0xAA
%rbx	0xBB
%rcx	0xCC
%rdx	0xDD

Exercise

- Given the following register state, what will the values of the registers be after the following instruction sequence?

- pushq %rax

- pushq %rcx

- pushq %rbx

- pushq %rdx

- popq %rax

%rax = 0xDD

- popq %rbx

%rbx = 0xBB

- popq %rcx

%rcx = 0xCC

- popq %rdx

%rdx = 0xAA

Registers

<u>Name</u>	<u>Value</u>
%rax	0xAA
%rbx	0xBB
%rcx	0xCC
%rdx	0xDD

Arithmetic and logic operations

Instruction		Effect	Description
leaq	S, D	$D \leftarrow \&S$	Load effective address
INC	D	$D \leftarrow D+1$	Increment
DEC	D	$D \leftarrow D-1$	Decrement
NEG	D	$D \leftarrow -D$	Negate
NOT	D	$D \leftarrow \sim D$	Complement
ADD	S, D	$D \leftarrow D + S$	Add
SUB	S, D	$D \leftarrow D - S$	Subtract
IMUL	S, D	$D \leftarrow D * S$	Multiply
XOR	S, D	$D \leftarrow D \wedge S$	Exclusive-or
OR	S, D	$D \leftarrow D S$	Or
AND	S, D	$D \leftarrow D \& S$	And
SAL	k, D	$D \leftarrow D \ll k$	Left shift
SHL	k, D	$D \leftarrow D \ll k$	Left shift (same as SAL)
SAR	k, D	$D \leftarrow D \gg_A k$	Arithmetic right shift
SHR	k, D	$D \leftarrow D \gg_L k$	Logical right shift

Figure 3.10 Integer arithmetic operations. The load effective address (leaq) instruction is commonly used to perform simple arithmetic. The remaining ones are more standard unary or binary operations. We use the notation \gg_A and \gg_L to denote arithmetic and logical right shift, respectively. Note the nonintuitive ordering of the operands with ATT-format assembly code.

Exercise

Instruction	Effect	Description
<code>leaq S, D</code>	$D \leftarrow \&S$	Load effective address
<code>INC D</code>	$D \leftarrow D+1$	Increment
<code>DEC D</code>	$D \leftarrow D-1$	Decrement
<code>NEG D</code>	$D \leftarrow -D$	Negate
<code>NOT D</code>	$D \leftarrow \sim D$	Complement
<code>ADD S, D</code>	$D \leftarrow D + S$	Add
<code>SUB S, D</code>	$D \leftarrow D - S$	Subtract
<code>IMUL S, D</code>	$D \leftarrow D * S$	Multiply
<code>XOR S, D</code>	$D \leftarrow D \wedge S$	Exclusive-or
<code>OR S, D</code>	$D \leftarrow D S$	Or
<code>AND S, D</code>	$D \leftarrow D \& S$	And
<code>SAL k, D</code>	$D \leftarrow D \ll k$	Left shift
<code>SHL k, D</code>	$D \leftarrow D \ll k$	Left shift (same as SAL)
<code>SAR k, D</code>	$D \leftarrow D \gg_A k$	Arithmetic right shift
<code>SHR k, D</code>	$D \leftarrow D \gg_L k$	Logical right shift

Registers	
Name	Value
<code>%rax</code>	0x12
<code>%rbx</code>	0x56
<code>%rcx</code>	0x02
<code>%rdx</code>	0xF0

What are the values of the destination registers after each of the following instructions executes in sequence?

```
addq %rax, %rax
subq %rax, %rbx
imulq %rcx, %rax
andq %rbx, %rdx
shrq $4, %rdx
```

Figure 3.10 Integer arithmetic operations. The load effective address (`leaq`) instruction is commonly used to perform simple arithmetic. The remaining ones are more standard unary or binary operations. We use the notation \gg_A and \gg_L to denote arithmetic and logical right shift, respectively. Note the nonintuitive ordering of the operands with ATT-format assembly code.

Exercise

Instruction	Effect	Description
<code>leaq S, D</code>	$D \leftarrow \&S$	Load effective address
<code>INC D</code>	$D \leftarrow D+1$	Increment
<code>DEC D</code>	$D \leftarrow D-1$	Decrement
<code>NEG D</code>	$D \leftarrow -D$	Negate
<code>NOT D</code>	$D \leftarrow \sim D$	Complement
<code>ADD S, D</code>	$D \leftarrow D + S$	Add
<code>SUB S, D</code>	$D \leftarrow D - S$	Subtract
<code>IMUL S, D</code>	$D \leftarrow D * S$	Multiply
<code>XOR S, D</code>	$D \leftarrow D \wedge S$	Exclusive-or
<code>OR S, D</code>	$D \leftarrow D S$	Or
<code>AND S, D</code>	$D \leftarrow D \& S$	And
<code>SAL k, D</code>	$D \leftarrow D \ll k$	Left shift
<code>SHL k, D</code>	$D \leftarrow D \ll k$	Left shift (same as SAL)
<code>SAR k, D</code>	$D \leftarrow D \gg_A k$	Arithmetic right shift
<code>SHR k, D</code>	$D \leftarrow D \gg_L k$	Logical right shift

Figure 3.10 Integer arithmetic operations. The load effective address (`leaq`) instruction is commonly used to perform simple arithmetic. The remaining ones are more standard unary or binary operations. We use the notation \gg_A and \gg_L to denote arithmetic and logical right shift, respectively. Note the nonintuitive ordering of the operands with ATT-format assembly code.

Registers

Name	Value
<code>%rax</code>	<code>0x12</code>
<code>%rbx</code>	<code>0x56</code>
<code>%rcx</code>	<code>0x02</code>
<code>%rdx</code>	<code>0xF0</code>

What are the values of the destination registers after each of the following instructions executes in sequence?

```
addq %rax, %rax  %rax:0x24
subq %rax, %rbx  %rbx:0x32
imulq %rcx, %rax %rax:0x48
andq %rbx, %rdx  %rdx:0x30
shrq $4, %rdx   %rdx:0x03
```

```
%rax = 0x48
%rbx = 0x32
%rcx = 0x02
%rdx = 0x03
```

Exercise

Instruction		Effect	Description
leaq	<i>S, D</i>	$D \leftarrow \&S$	Load effective address
INC	<i>D</i>	$D \leftarrow D+1$	Increment
DEC	<i>D</i>	$D \leftarrow D-1$	Decrement
NEG	<i>D</i>	$D \leftarrow -D$	Negate
NOT	<i>D</i>	$D \leftarrow \sim D$	Complement
ADD	<i>S, D</i>	$D \leftarrow D + S$	Add
SUB	<i>S, D</i>	$D \leftarrow D - S$	Subtract
IMUL	<i>S, D</i>	$D \leftarrow D * S$	Multiply
XOR	<i>S, D</i>	$D \leftarrow D \wedge S$	Exclusive-or
OR	<i>S, D</i>	$D \leftarrow D S$	Or
AND	<i>S, D</i>	$D \leftarrow D \& S$	And
SAL	<i>k, D</i>	$D \leftarrow D \ll k$	Left shift
SHL	<i>k, D</i>	$D \leftarrow D \ll k$	Left shift (same as SAL)
SAR	<i>k, D</i>	$D \leftarrow D \gg_A k$	Arithmetic right shift
SHR	<i>k, D</i>	$D \leftarrow D \gg_L k$	Logical right shift

What does the following instruction do if `%rax = 0x100`?

```
leaq (%rax, %rax, 2), %rax
```

`%rax = 0x300`
(multiply by three)

Note: `leaq` does not actually read/write memory!

Figure 3.10 Integer arithmetic operations. The load effective address (`leaq`) instruction is commonly used to perform simple arithmetic. The remaining ones are more standard unary or binary operations. We use the notation \gg_A and \gg_L to denote arithmetic and logical right shift, respectively. Note the nonintuitive ordering of the operands with ATT-format assembly code.