CS 261
Spring 2024

Mike Lam, Professor

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

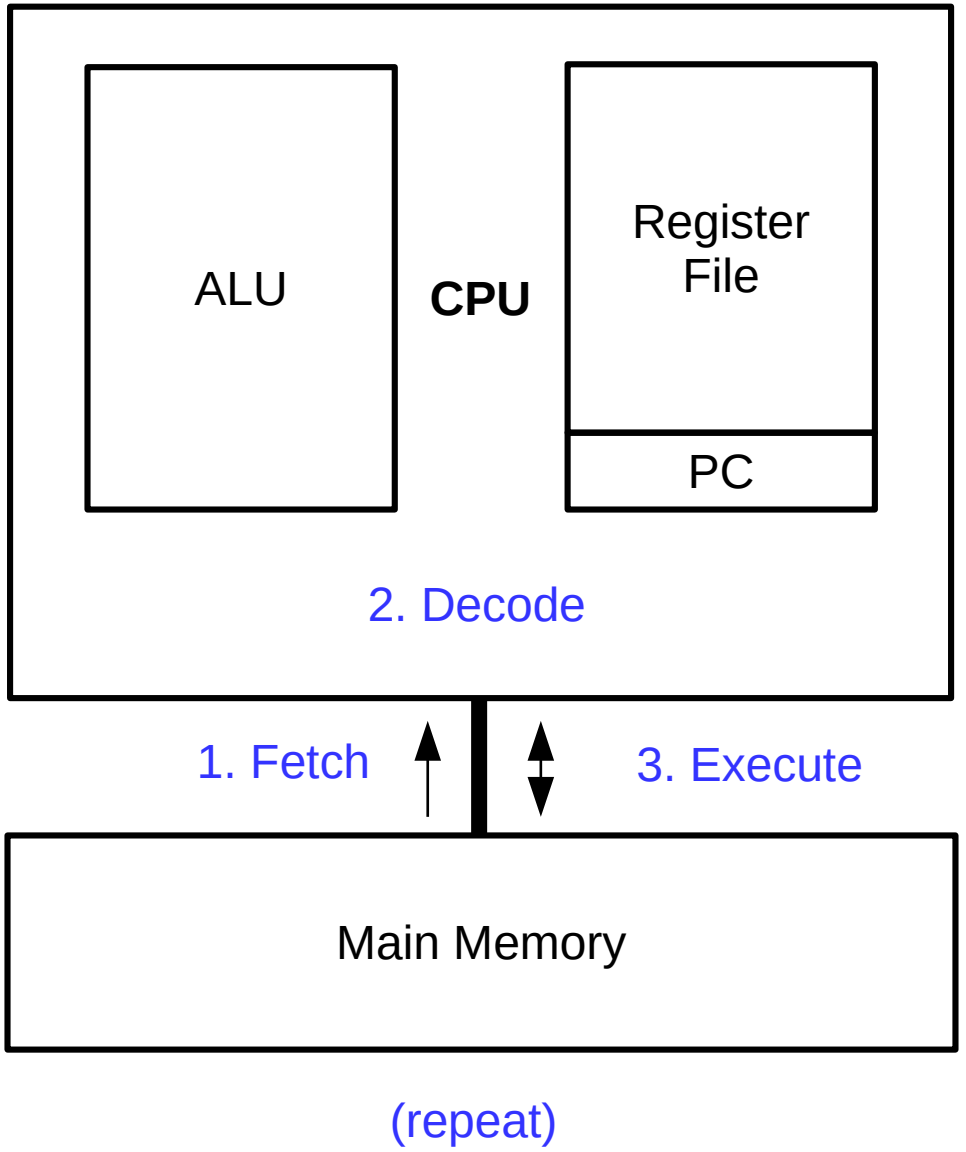
$$-\nabla p + \nabla \cdot \boldsymbol{\tau} + \rho \mathbf{g}$$

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)

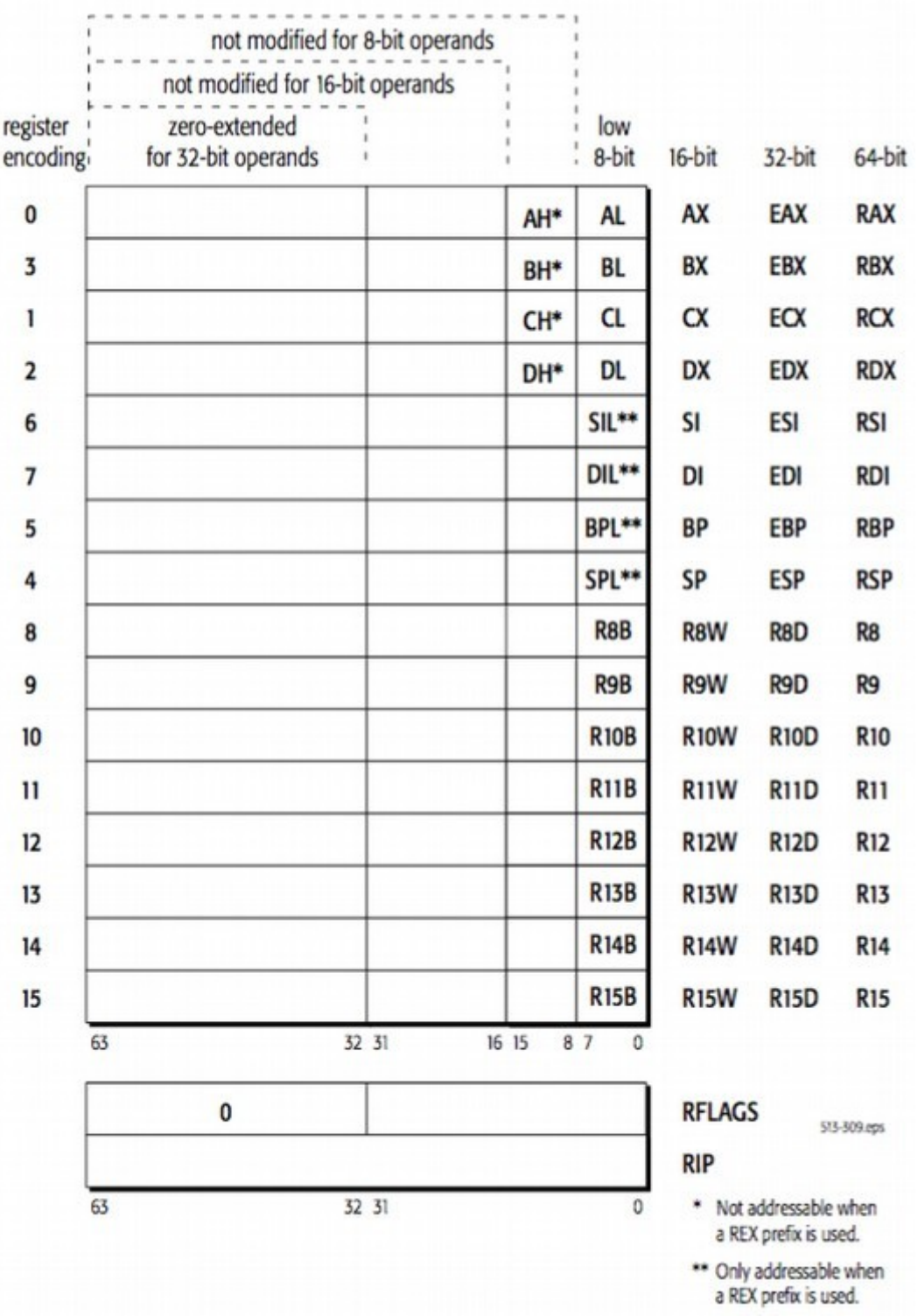
Data movement

- Primary data movement instruction: "`mov`"
  - **Copies** data from first operand to second operand
  - Multiple suffixes:
    - `movb`, `movw`, `movl`, `movq`, `movabsq`
    - `movabsq` is the only form that takes a 64-bit immediate
- Zero-extension variant: "`movz`"
  - `movzbw`, `movzbl`, `movzwl`, `movzbq`, `movzwq`
  - Note lack of `movzlq`; just use `movl`, which sets higher 32-bits to zero
- Sign-extension variant: "`movs`"
  - `movsbw`, `movsbl`, `movswl`, `movsbq`, `movswq`, `movslq`

Registers

- Multiple names per register
  - Refers to different data sizes
  - **e**XX = lower 32-bits (e.g., **e**ax)
  - **r**XX = full 64 bits (e.g., **r**ax)

- Instruction suffixes and operand sizes must match!
  - E.g., `movq $1, %rax` is valid but `movq $1, %eax` is not



Memory addressing modes

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

- 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***$_{index}$] · ***s***
  - Scale (*s*) must be 1, 2, 4, or 8

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

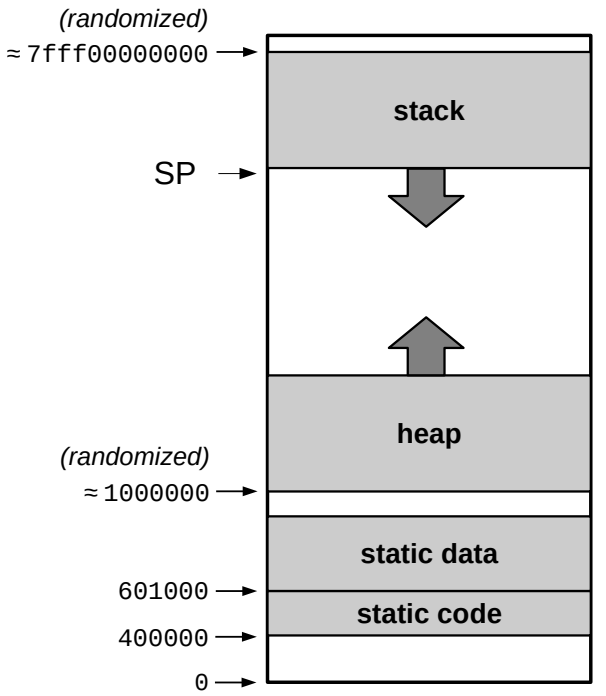
- In x86-64, most opcodes have no memory -> memory form
  - You can't encode two memory operands in the same instruction
  - Invalid: `movl (%rax), (%rdx)`

- 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

Name	Value
%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`
  - `popq %rbx`
  - `popq %rcx`
  - `popq %rdx`

`%rax = 0xDD`
`%rbx = 0xBB`
`%rcx = 0xCC`
`%rdx = 0xAA`

Registers

Name	Value
%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	$D \leftarrow D \hat{} S$	Exclusive-or
OR	S, D	$D \leftarrow D \mid S$	Or
AND	S, D	$D \leftarrow D \,\&\, S$	And
SAL	k, D	$D \leftarrow D << k$	Left shift
SHL	k, D	$D \leftarrow D << k$	Left shift (same as SAL)
SAR	k, D	$D \leftarrow D >>_A k$	Arithmetic right shift
SHR	k, D	$D \leftarrow D >>_L k$	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 $>>_A$ and $>>_L$ to denote arithmetic and logical right shift, respectively. Note the nonintuitive ordering of the operands with ATT-format assembly code.

Exercise

Instruction		Effect	Description
`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	$D \leftarrow D \hat{} S$	Exclusive-or
OR	S, D	$D \leftarrow D \mid S$	Or
AND	S, D	$D \leftarrow D \,\&\, S$	And
SAL	k, D	$D \leftarrow D << k$	Left shift
SHL	k, D	$D \leftarrow D << k$	Left shift (same as SAL)
SAR	k, D	$D \leftarrow D >>_A k$	Arithmetic right shift
SHR	k, D	$D \leftarrow D >>_L k$	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 $>>_A$ and $>>_L$ to denote arithmetic and logical right shift, respectively. Note the nonintuitive ordering of the operands with ATT-format assembly code.

Registers

Name	Value
%rax	0x12
%rbx	0x56
%rcx	0x02
%rdx	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
```

Exercise

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	$D \leftarrow D \hat{} S$	Exclusive-or
OR	S, D	$D \leftarrow D \mid S$	Or
AND	S, D	$D \leftarrow D \,\&\, S$	And
SAL	k, D	$D \leftarrow D << k$	Left shift
SHL	k, D	$D \leftarrow D << k$	Left shift (same as SAL)
SAR	k, D	$D \leftarrow D >>_A k$	Arithmetic right shift
SHR	k, D	$D \leftarrow D >>_L k$	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 $>>_A$ and $>>_L$ to denote arithmetic and logical right shift, respectively. Note the nonintuitive ordering of the operands with ATT-format assembly code.

Registers

Name	Value
%rax	0x12
%rbx	0x56
%rcx	0x02
%rdx	0xF0

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`	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	$D \leftarrow D \hat{} S$	Exclusive-or
OR	S, D	$D \leftarrow D \mid S$	Or
AND	S, D	$D \leftarrow D \,\&\, S$	And
SAL	k, D	$D \leftarrow D << k$	Left shift
SHL	k, D	$D \leftarrow D << k$	Left shift (same as SAL)
SAR	k, D	$D \leftarrow D >>_A k$	Arithmetic right shift
SHR	k, D	$D \leftarrow D >>_L k$	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 $>>_A$ and $>>_L$ to denote arithmetic and logical right shift, respectively. Note the nonintuitive ordering of the operands with ATT-format assembly code.

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!