CS 261
Fall 2016

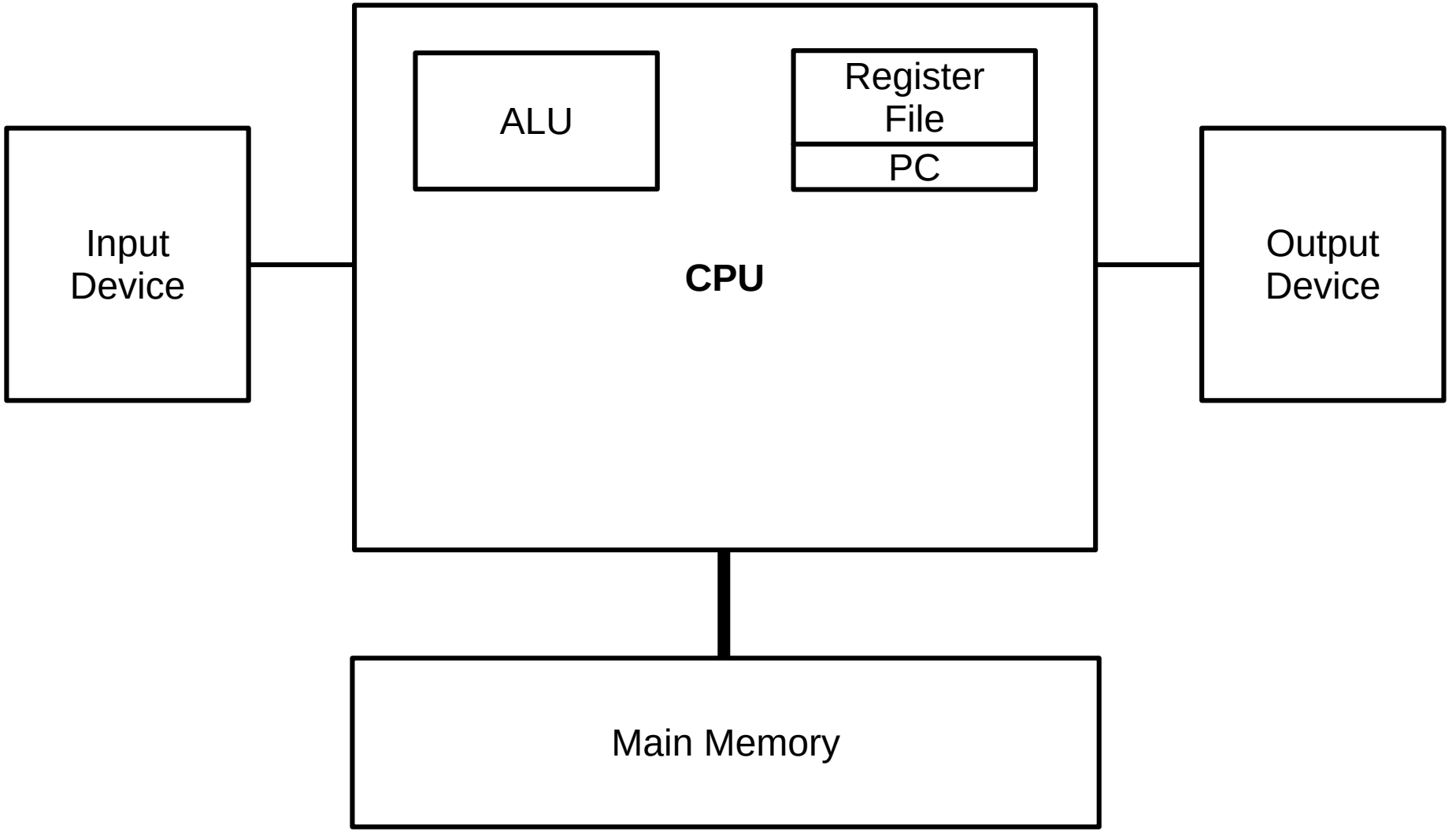
Mike Lam, Professor

x86-64 Assembly

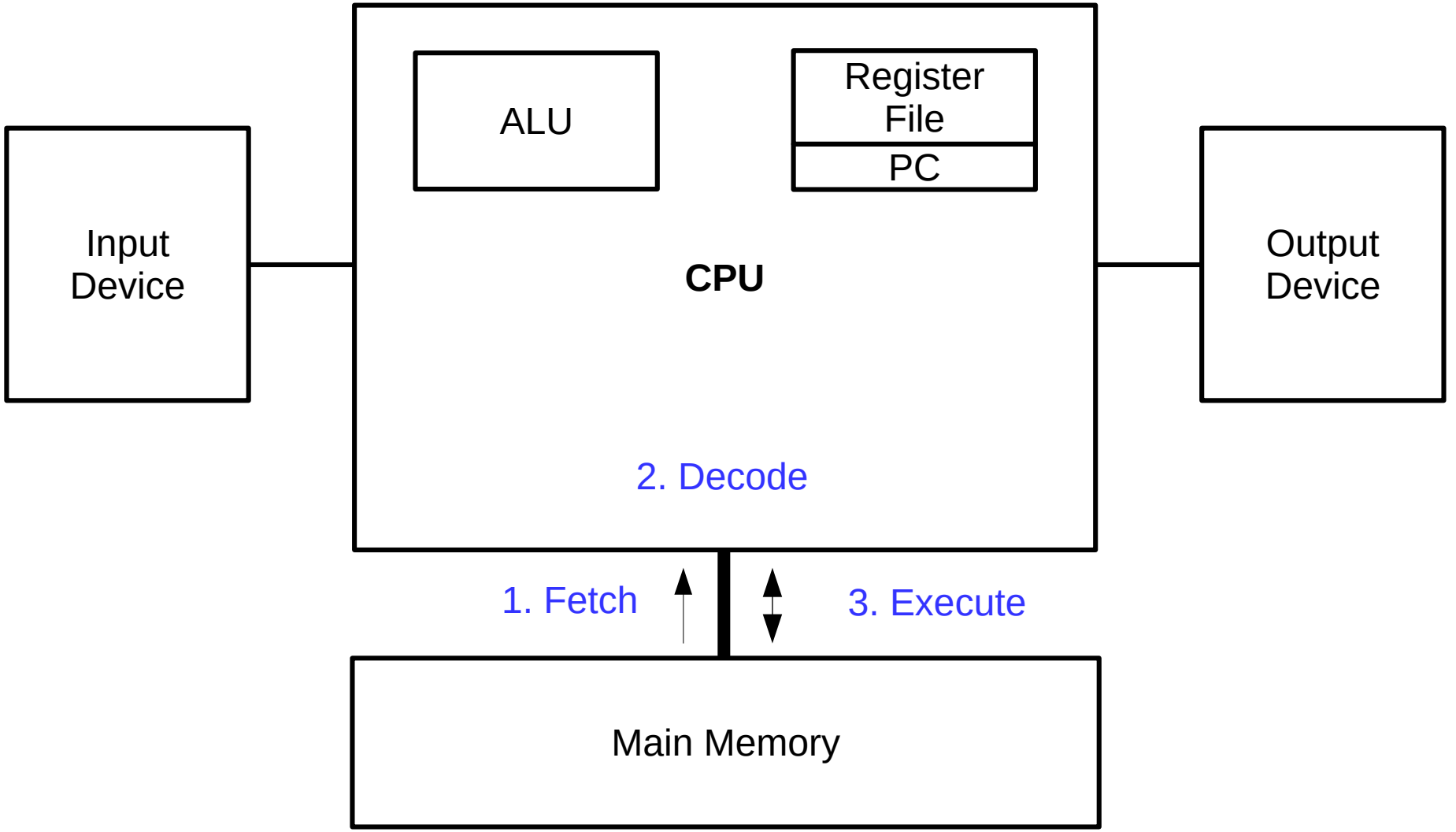
Topics

- Architecture/assembly intro
- Data formats
- Data movement
- Arithmetic and logical operations

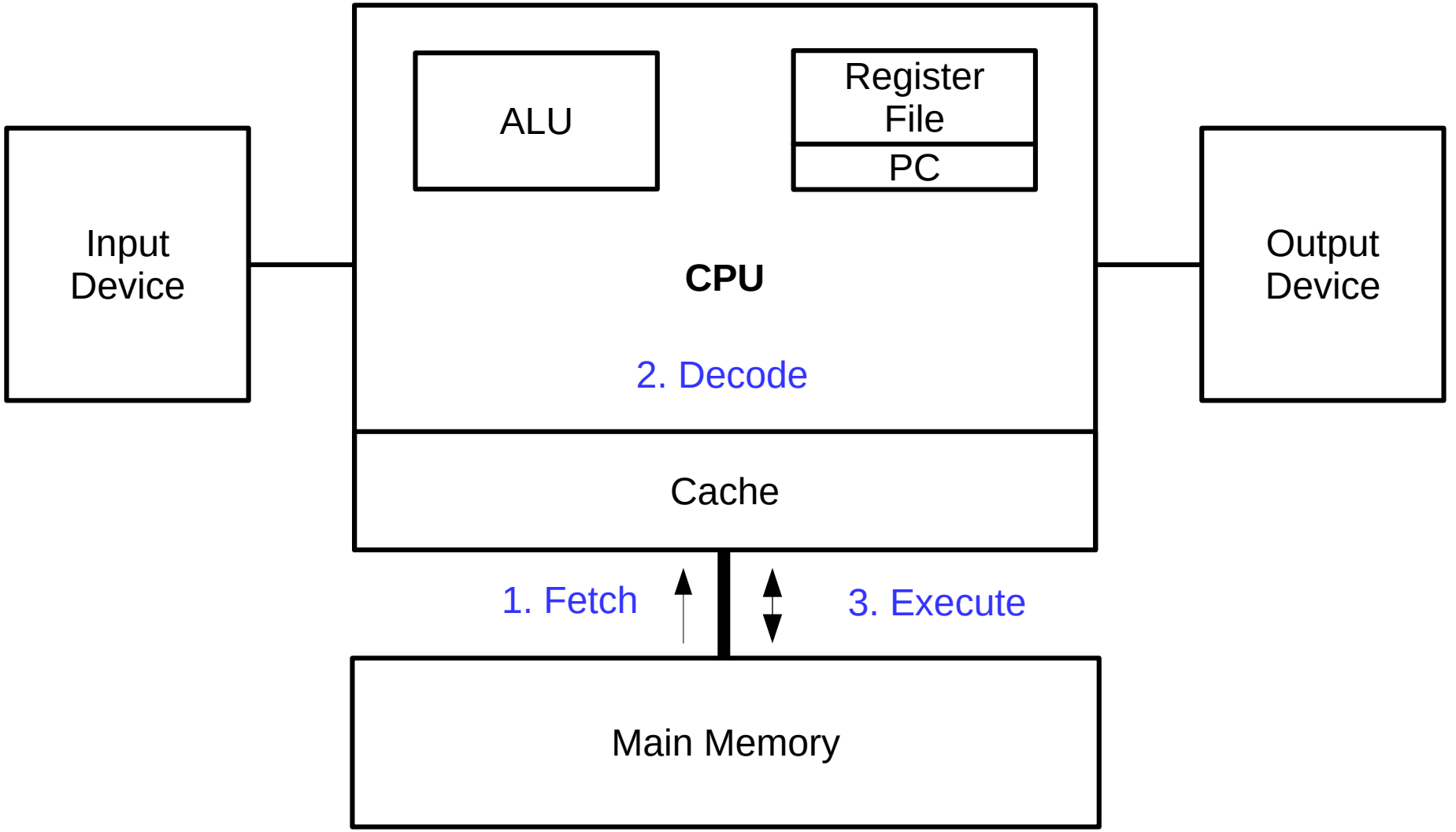
von Neumann architecture



von Neumann architecture



von Neumann architecture



Assembly programming

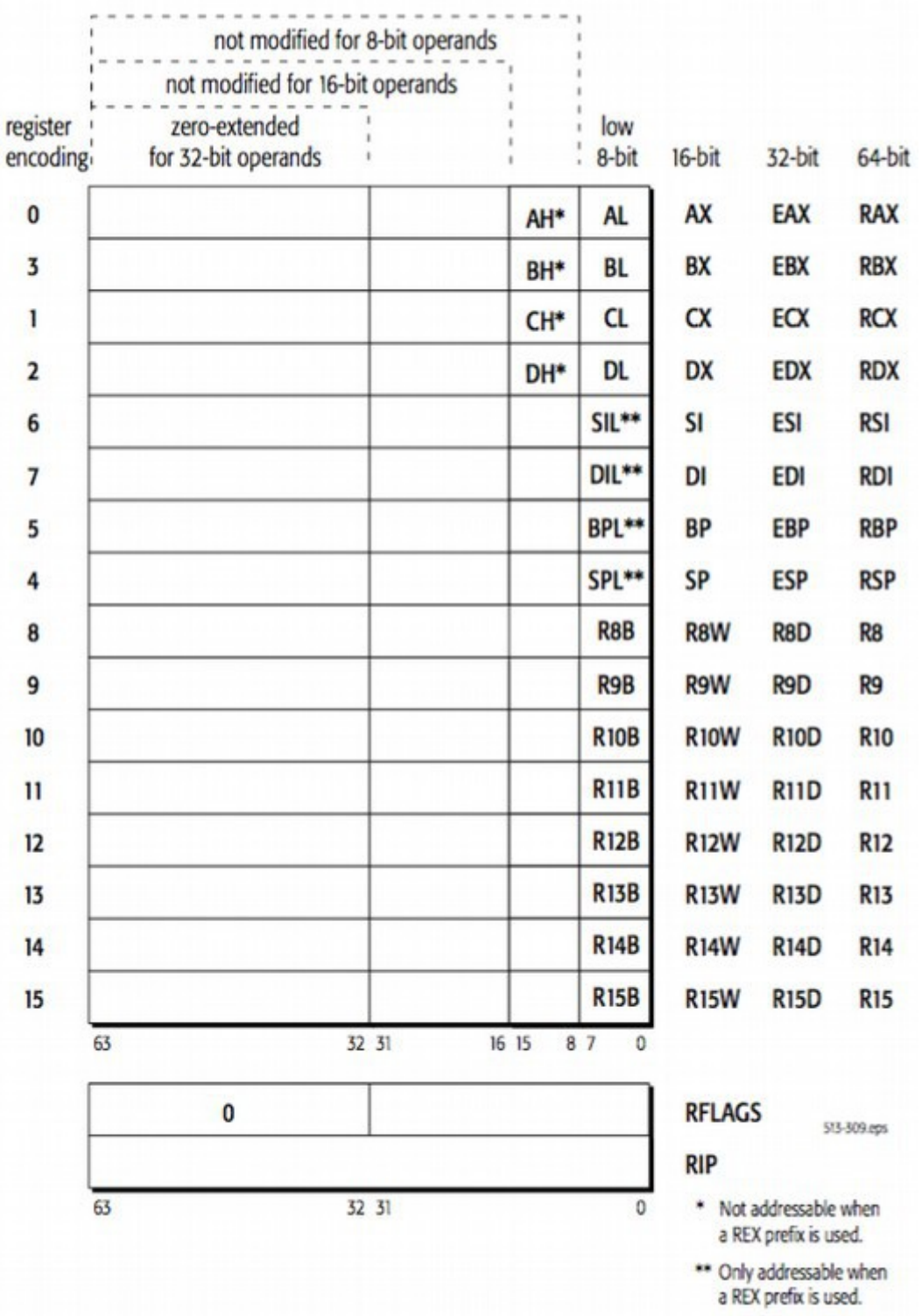
- Assembly: simple, CPU-specific programming language
  - However, x86-64 has become the industry standard
  - Based on fetch/decode/execute execution loop
  - Program is stored on disk along with data
  - Low-level access to machine (memory, I/O, etc.)
  - Each instruction = opcode and operands
  - Compilers often target assembly code instead of machine code for increased portability
  - Understanding assembly code can help you optimize and secure your programs

Assembly code operand types

- Immediate
  - Operand embedded in instruction itself
  - Written in assembly using “$” prefix (e.g., `$42` or `$0x1234`)
- Register
  - Operand stored in register file
  - Accessed by register number
  - Written in assembly using name and “%” prefix (e.g., `%eax` or `%rsp`)
- Memory
  - Operand stored in main memory
  - Accessed by effective address
  - Written in assembly using a variety of addressing modes

Registers

- General-purpose
  - AX: accumulator
  - BX: base
  - CX: counter
  - DX: address
  - SI: source index
  - DI: dest index
- Special
  - **BP: base pointer**
  - **SP: stack pointer**
  - **IP: instruction pointer**
  - **FLAGS: status info**



Memory addressing modes

- Absolute: `mov $1, x`
  - Moves to $M[x]$
- Indirect: `mov $1, (r)`
  - Moves to $M[R[r]]$
- Base + displacement: `mov $1, x(r)`
  - Moves to $M[x + R[r]]$
- Indexed: `mov $1, x(r_b, r_i)`
  - Moves to $M[x + R[r_b] + R[r_i]]$
- Scaled indexed: `mov $1, x(r_b, r_i, s)`
  - Moves to $M[x + R[r_b] + R[r_i] \cdot s]$
  - Scale (s) must be 1, 2, 4, or 8

Exercise

- Given the following machine status, what is the value for the following assembly operands?
  - `$42`
  - `$0x10`
  - `%rax`
  - `0x104`
  - `(%rax)`
  - `4(%rax)`
  - `2(%rax, %rdx)`
  - `(%rax, %rdx, 4)`

Registers

Name	Value
%rax	0x100
%rdx	0x2

Memory

Address	Value
0x100	0xFF
0x104	0xAB
0x108	0x13

Exercise

- Given the following machine status, what is the value for the following assembly operands?
  - `$42` `42`
  - `$0x10` `16`
  - `%rax` `0x100`
  - `0x104` `0xAB`
  - `(%rax)` `0xFF`
  - `4(%rax)` `0xAB`
  - `2(%rax, %rdx)` `0xAB`
  - `(%rax, %rdx, 4)` `0x13`

Registers

Name	Value
%rax	0x100
%rdx	0x2

Memory

Address	Value
0x100	0xFF
0x104	0xAB
0x108	0x13

Brief aside: data formats

- Historical artifact: "word" in x86 is 16-bit
  - 1 byte (8 bits) = "byte" (`b`)
  - 2 bytes (16 bits) = "word" (`w`)
  - 4 bytes (32 bits) = "double word" (`l`)
  - 8 bytes (64 bits) = "quad word" (`q`)

Data movement

- Often, a “class” of instructions will perform similar jobs, but on different sizes of data
- Primary data movement instruction: "`mov`"
  - `movb`, `movw`, `movl`, `movq`, `movabsq`
- Zero-extension variant: "`movz`"
  - `movzbw`, `movzbl`, `movzwl`, `movzbq`, `movzwq`
- Sign-extension variant: "`movs`"
  - `movsbw`, `movsbl`, `movswl`, `movsbq`, `movswq`, `movslq`

Stack management

- Push/pop instructions: `pushq` and `popq`
  - 8-byte (quadword) slots, growing “downward” from high addresses to low addresses
- Register `%rsp` stores address of top of stack
  - i.e., a pointer to the last value pushed
- `pushq`
  - Subtract 8 from stack pointer
  - Store value at new stack top location (`%rsp`)
- `popq`
  - Retrieve value at current stack top (`%rsp`)
  - 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 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 all registers after the following instructions?

```
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 all registers after the following instructions?

```
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
```

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)