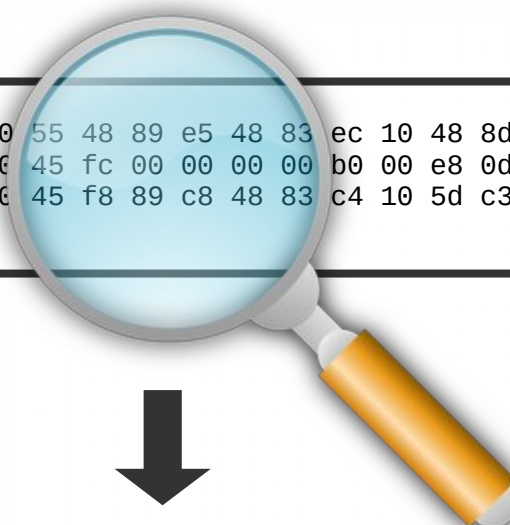
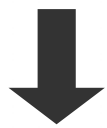


CS 261 Fall 2017

Mike Lam, Professor



```
00000000100000f50 55 48 89 e5 48 83 ec 10 48 8d 3d 3b 00 00 00 c7
00000000100000f60 45 fc 00 00 00 00 b0 00 e8 0d 00 00 00 31 c9 89
00000000100000f70 45 f8 89 c8 48 83 c4 10 5d c3
```



```
_main:
00000000100000f50    pushq    %rbp
00000000100000f51    movq     %rsp, %rbp
00000000100000f54    subq     $0x10, %rsp
00000000100000f58    leaq     0x3b(%rip), %rdi
00000000100000f5f    movl     $0x0, -0x4(%rbp)
00000000100000f66    movb     $0x0, %al
00000000100000f68    callq    0x100000f7a
00000000100000f6d    xorl     %ecx, %ecx
```

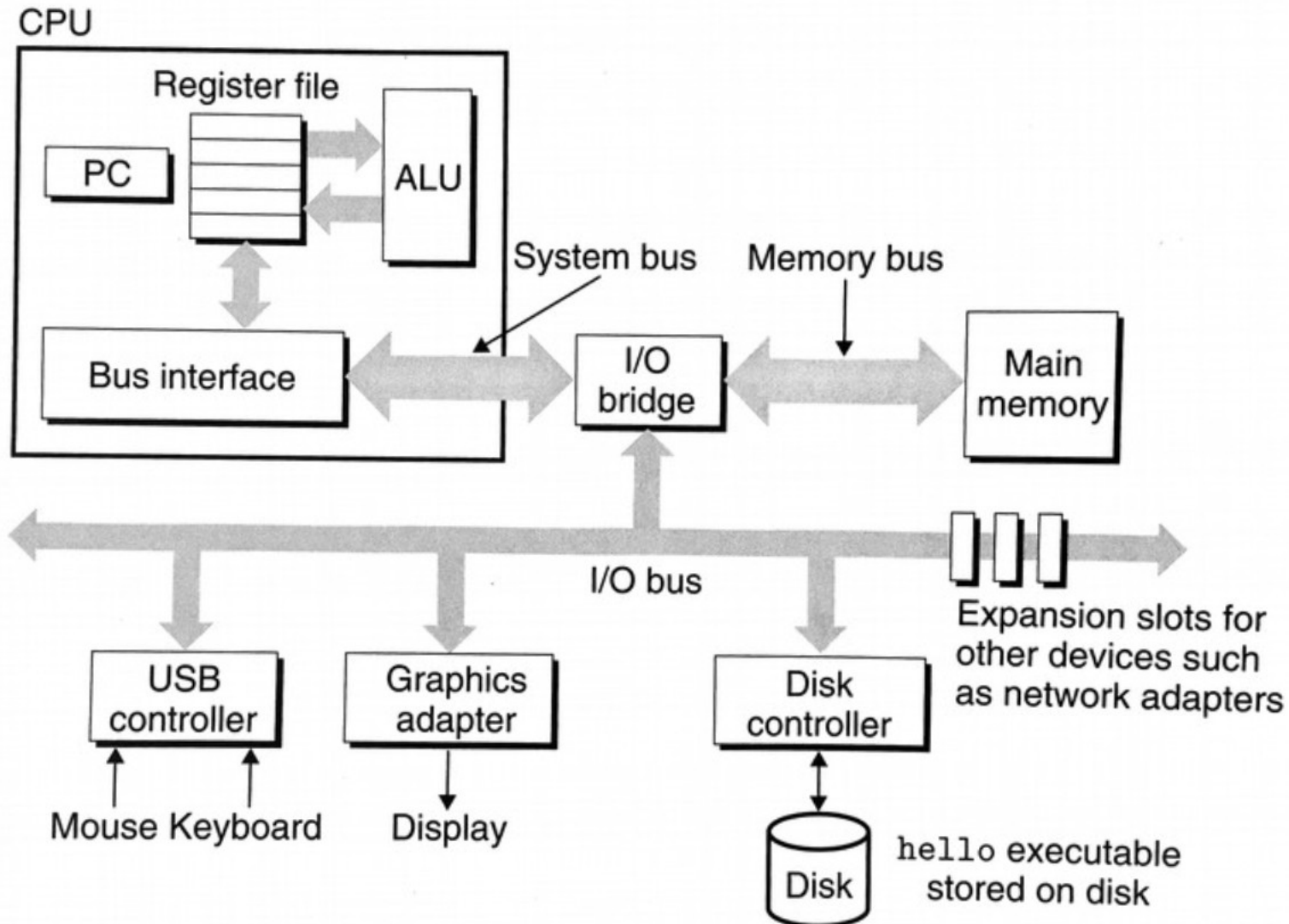
Machine and Assembly Code

Data Movement and Arithmetic

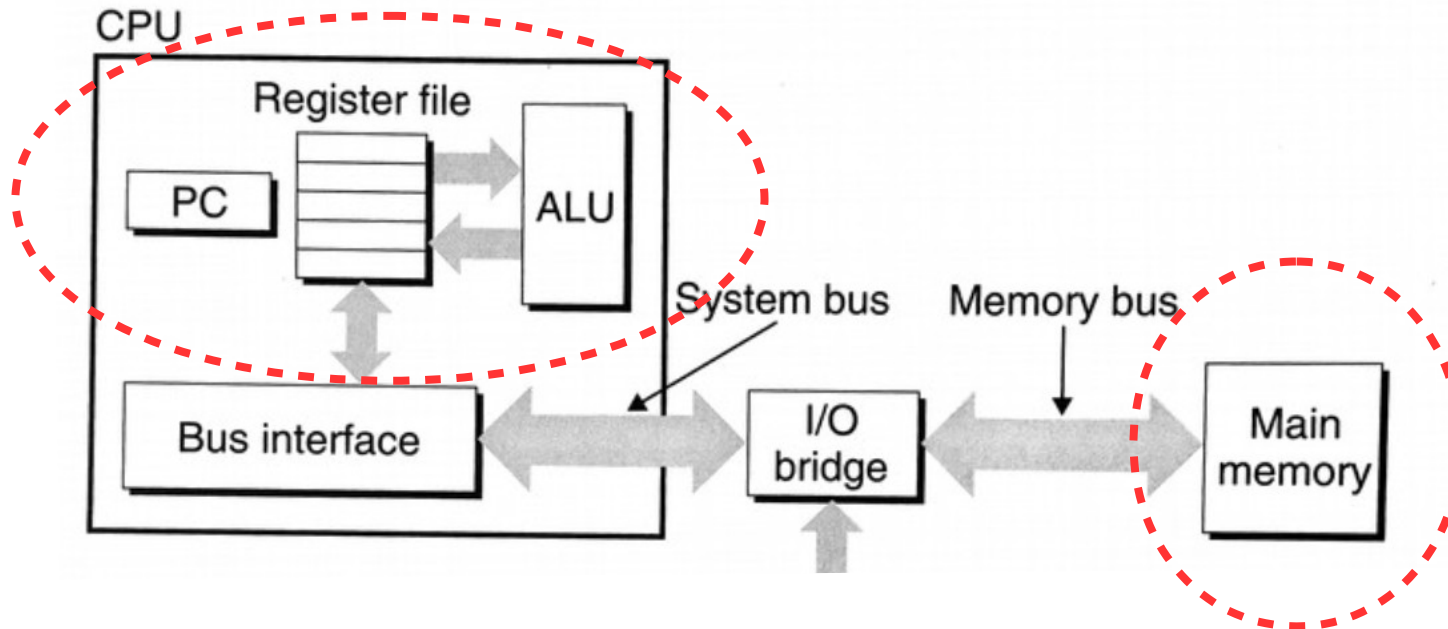
Topics

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

Computer systems

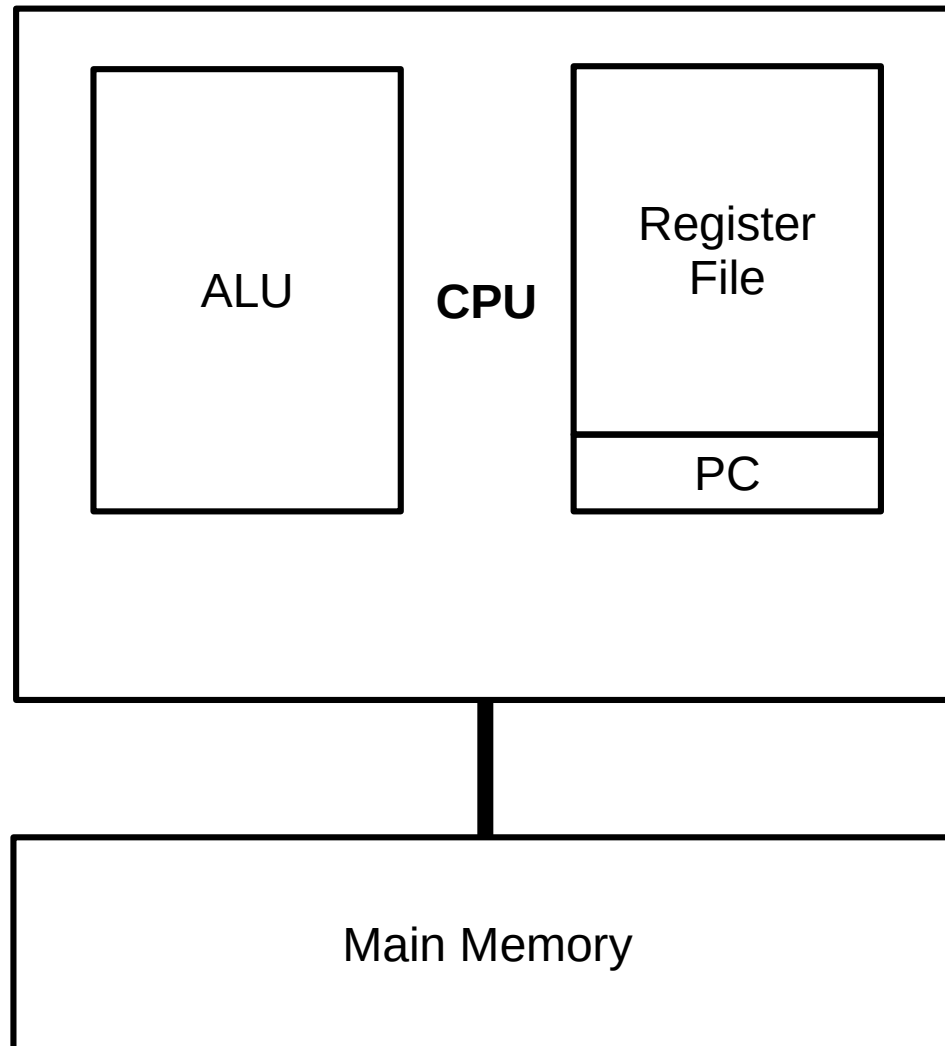


Computer systems

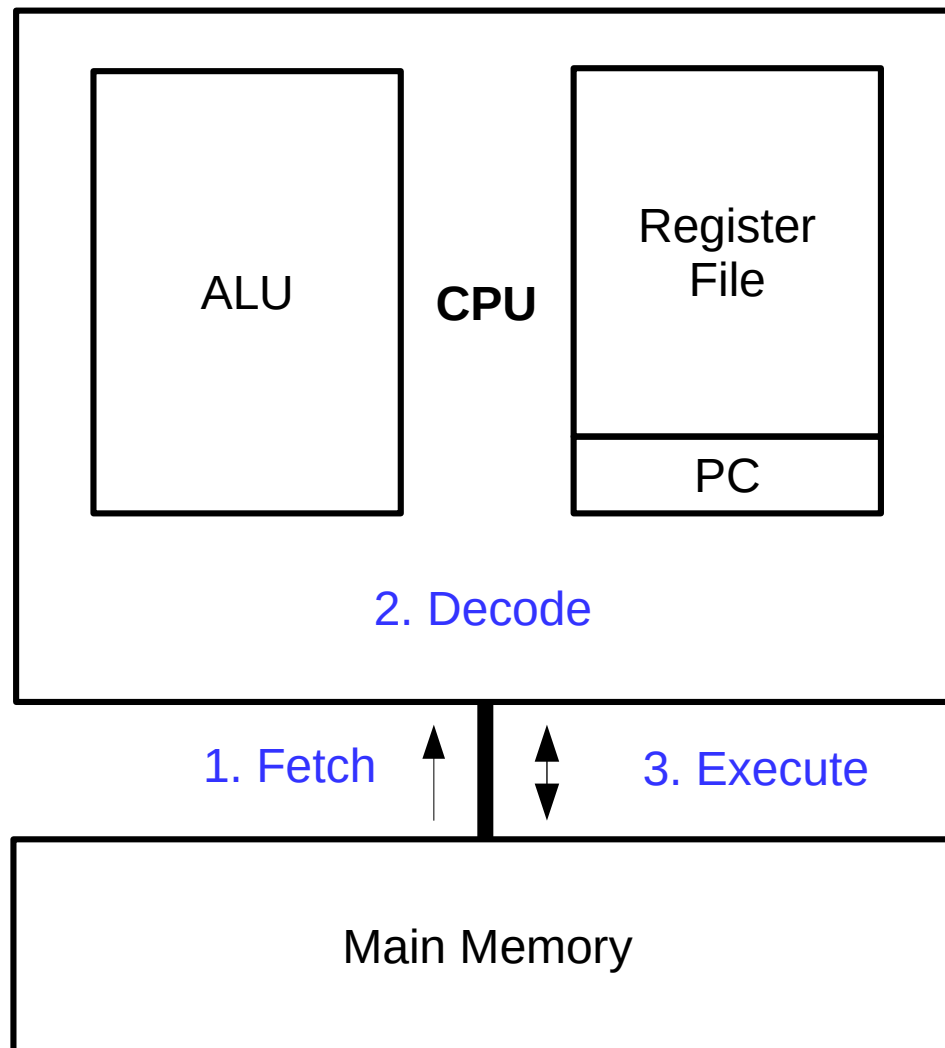


Let's focus for now on the single-CPU components

von Neumann architecture



von Neumann architecture



Machine code

- **Machine code**
 - Variable-length binary encoding of **opcodes** and *operands*
 - Program is stored in memory along with data
 - Specific to a particular CPU architecture (e.g., x86-64)
 - Looks very different than the original C code!

```
int add (int num1, int num2)
{
    return num1 + num2;
}
```



```
0000000000400606 <add>:
400606:    55
400607:    48 89 e5
40060a:    89 7d fc
40060d:    89 75 f8
400610:    8b 55 fc
400613:    8b 45 f8
400616:    01 d0
400618:    5d
400619:    c3
```

Machine code

- Machine instructions are specified by an **instruction set architecture** (ISA)
 - **x86-64** (x64) is the current dominant workstation/server architecture
 - **ARM** is used in embedded and mobile markets
 - **POWER** is used in high-performance market
 - x86-64 has an **enormous**, complex instruction set
 - Lots of legacy features and support for previous ISAs
 - We'll learn a bit of it now, then later focus on a simplified form called **Y86**

```
0000000000400606 <add>:
400606:      55
400607:      48 89 e5
40060a:      89 7d fc
40060d:      89 75 f8
400610:      8b 55 fc
400613:      8b 45 f8
400616:      01 d0
400618:      5d
400619:      c3
```

Assembly code

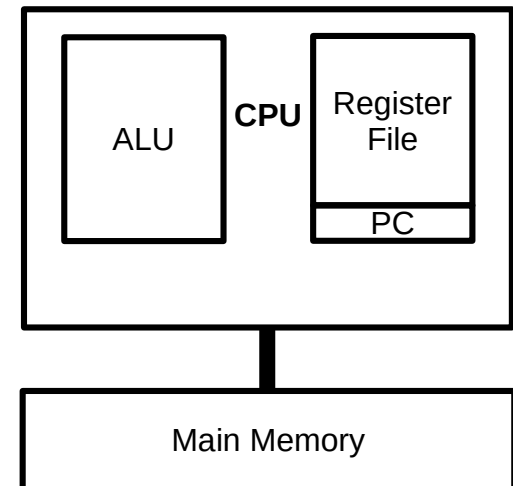
- **Assembly code**: human-readable form of machine code
 - Each indented line of text represents a single machine code instruction
 - Two main x86-64 formats: **Intel** and **ATT** (we'll use the latter)
 - Use "#" to denote comments (extends to end of line)
 - Generated from C code by compiler (not a simple process!)
 - **Disassemblers** like **objdump** can extract assembly from an executable
 - Understanding assembly helps you to debug, optimize, and secure your programs

		opcode	operands
0000000000400606	<add>:		
400606:	55	push	<i>%rbp</i>
400607:	48 89 e5	mov	<i>%rsp, %rbp</i>
40060a:	89 7d fc	mov	<i>%edi, -0x4(%rbp)</i>
40060d:	89 75 f8	mov	<i>%esi, -0x8(%rbp)</i>
400610:	8b 55 fc	mov	<i>-0x4(%rbp), %edx</i>
400613:	8b 45 f8	mov	<i>-0x8(%rbp), %eax</i>
400616:	01 d0	add	<i>%edx, %eax</i>
400618:	5d	pop	<i>%rbp</i>
400619:	c3	retq	

Assembly code

- Assembly provides low-level access to machine
 - **Program counter** (PC) tracks current instruction
 - Like a bookmark; also referred to as the **instruction pointer** (IP)
 - **Arithmetic logic unit** (ALU) executes **opcode** of instructions
 - Today, we'll focus on **data movement** and **arithmetic** opcodes
 - **Register file & main memory** store *operands*
 - Registers are faster but main memory is larger

		opcode	operands
00000000000400606	<add>:		
400606:	55	push	%rbp
400607:	48 89 e5	mov	%rsp, %rbp
40060a:	89 7d fc	mov	%edi, -0x4(%rbp)
40060d:	89 75 f8	mov	%esi, -0x8(%rbp)
400610:	8b 55 fc	mov	-0x4(%rbp), %edx
400613:	8b 45 f8	mov	-0x8(%rbp), %eax
400616:	01 d0	add	%edx, %eax
400618:	5d	pop	%rbp
400619:	c3	retq	



Registers

- General-purpose
 - AX**: accumulator
 - BX**: base
 - CX**: counter
 - DX**: address
 - SI**: source index
 - DI**: dest index
- Special
 - BP**: base pointer
 - SP**: stack pointer
 - FLAGS**: status info
 - "Condition codes" in CS:APP
 - IP**: instruction pointer
 - This is the PC on x86-64

eXX = lower 32-bits (e.g., eax)
 rXX = full 64 bits (e.g., rax)

register encoding	not modified for 8-bit operands				low 8-bit	16-bit	32-bit	64-bit
	not modified for 16-bit operands							
	zero-extended for 32-bit operands							
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

0

63

32 31

0

RFLAGS

513-309.eps

RIP

* Not addressable when a REX prefix is used.

** Only addressable when a REX prefix is used.

Operand types

- Immediate
 - Operand embedded in instruction itself
 - Extends the size of the instruction by the width of the value
 - 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** calculated from instruction components
 - Written in assembly using a variety of **addressing modes**

Memory addressing modes

- Absolute: **addr**
 - Effective address: **addr**
 - Indirect: (**reg**)
 - Effective address: $R[\text{reg}]$
 - Base + displacement: **offset**(**reg**)
 - Effective address: $\text{offset} + R[\text{reg}]$
 - Indexed: **offset**(**reg**_{base}, **reg**_{index})
 - Effective address: $\text{offset} + R[\text{reg}_{\text{base}}] + R[\text{reg}_{\text{index}}]$
 - Scaled indexed: **offset**(**reg**_{base}, **reg**_{index}, **s**)
 - Effective address: $\text{offset} + R[\text{reg}_{\text{base}}] + R[\text{reg}_{\text{index}}] \cdot s$
 - Scale (s) must be 1, 2, 4, or 8
- $R[\text{reg}]$ = value of register **reg**
- useful for pointers!
- useful for arrays!
- (also, note that **offset** and **reg**_{base} are optional here)

Exercise

- Given the following machine status, what is the value of the following assembly operands? (assume 32-bit memory locations)

- \$42
- \$0x10
- %rax
- 0x104
- (%rax)
- 4(%rax)
- 2(%rax, %rdx)
- (%rax, %rdx, 4)

Registers

<u>Name</u>	<u>Value</u>
%rax	0x100
%rdx	0x2

Memory

<u>Address</u>	<u>Value</u>
0x100	0xFF
0x104	0xAB
0x108	0x13

Exercise

- Given the following machine status, what is the value of the following assembly operands? (assume 32-bit memory locations)

- \$42 42
- \$0x10 16
- %rax 0x100
- 0x104 0xAB
- (%rax) 0xFF
- 4(%rax) 0xAB
- 2(%rax, %rdx) 0xAB
- (%rax, %rdx, 4) 0x13

Registers

<u>Name</u>	<u>Value</u>
%rax	0x100
%rdx	0x2


Memory

<u>Address</u>	<u>Value</u>
0x100	0xFF
0x104	0xAB
0x108	0x13

Data sizes

- 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 word" (d suffix)
 - 8 bytes (64 bits) = "quad word" (q suffix)
- Often, a "class" of instructions will perform similar jobs, but on different sizes of data
 - There are no "types" in assembly code
 - Thus, instruction suffixes and operand sizes must match!
 - E.g., `movq $1, %rax` is valid but `movq $1, %eax` is not

Data movement

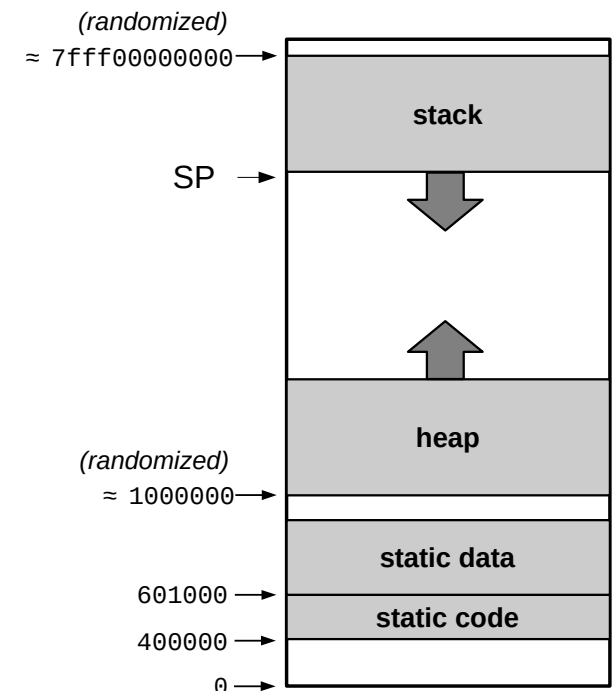
- Primary data movement instruction: "mov"
 - Copies data from first operand to second operand
 - E.g., `movq $1, %rax` will set the value of RAX to 1
 - `movb`, `movw`, `movl`, `movq`, `movabsq`
- Zero-extension variant: "movz"
 - `movzbw`, `movzbl`, `movzwl`, `movzbq`, `movzww`
 - Note lack of `movzlq`; just use `movl`, which sets higher 32-bits to zero
- Sign-extension variant: "movs"
 - `movsbw`, `movsbl`, `movswl`, `movsbq`, `movswq`, `movslq`
 - 
byte-to-word

x86-64 addresses

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

Stack management

- 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 stack top
 - `popq <reg>` instruction
 - Retrieve value at current stack top (`%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
- popq %rbx
- popq %rcx
- popq %rdx

%rax = 0xDD

%rbx = 0xBB

%rcx = 0xCC

%rdx = 0xAA

Registers

<u>Name</u>	<u>Value</u>
%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	$D \leftarrow D \wedge S$	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</code>	S, D	$D \leftarrow \&S$	Load effective address
<code>INC</code>	D	$D \leftarrow D+1$	Increment
<code>DEC</code>	D	$D \leftarrow D-1$	Decrement
<code>NEG</code>	D	$D \leftarrow -D$	Negate
<code>NOT</code>	D	$D \leftarrow \sim D$	Complement
<code>ADD</code>	S, D	$D \leftarrow D + S$	Add
<code>SUB</code>	S, D	$D \leftarrow D - S$	Subtract
<code>IMUL</code>	S, D	$D \leftarrow D * S$	Multiply
<code>XOR</code>	S, D	$D \leftarrow D \wedge S$	Exclusive-or
<code>OR</code>	S, D	$D \leftarrow D S$	Or
<code>AND</code>	S, D	$D \leftarrow D \& S$	And
<code>SAL</code>	k, D	$D \leftarrow D \ll k$	Left shift
<code>SHL</code>	k, D	$D \leftarrow D \ll k$	Left shift (same as SAL)
<code>SAR</code>	k, D	$D \leftarrow D \gg_A k$	Arithmetic right shift
<code>SHR</code>	k, D	$D \leftarrow D \gg_L k$	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 all registers after the following instructions?

```
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
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

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
```

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.

```
%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	$D \leftarrow D \wedge S$	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

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

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

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
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

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.

Hand-writing assembly

- Minimal template (returns 0; known to work on stu):

```
.globl main
main:
```

```
    movq $0, %rax    # your code goes here
```

```
    ret
```

- Save in .s file and build with gcc as usual (don't use "-c" flag)
 - Run program and view return value in bash with `echo $?`
- Use gdb to trace execution
 - `start`: begin execution and pause at main
 - `disas`: print disassembly of current function
 - `ni`: next instruction (step over function calls)
 - `si`: step instruction (step into function calls)
 - `p/x $rax`: print value of RAX (note "\$" instead of "%")
 - `info registers`: print values of all registers