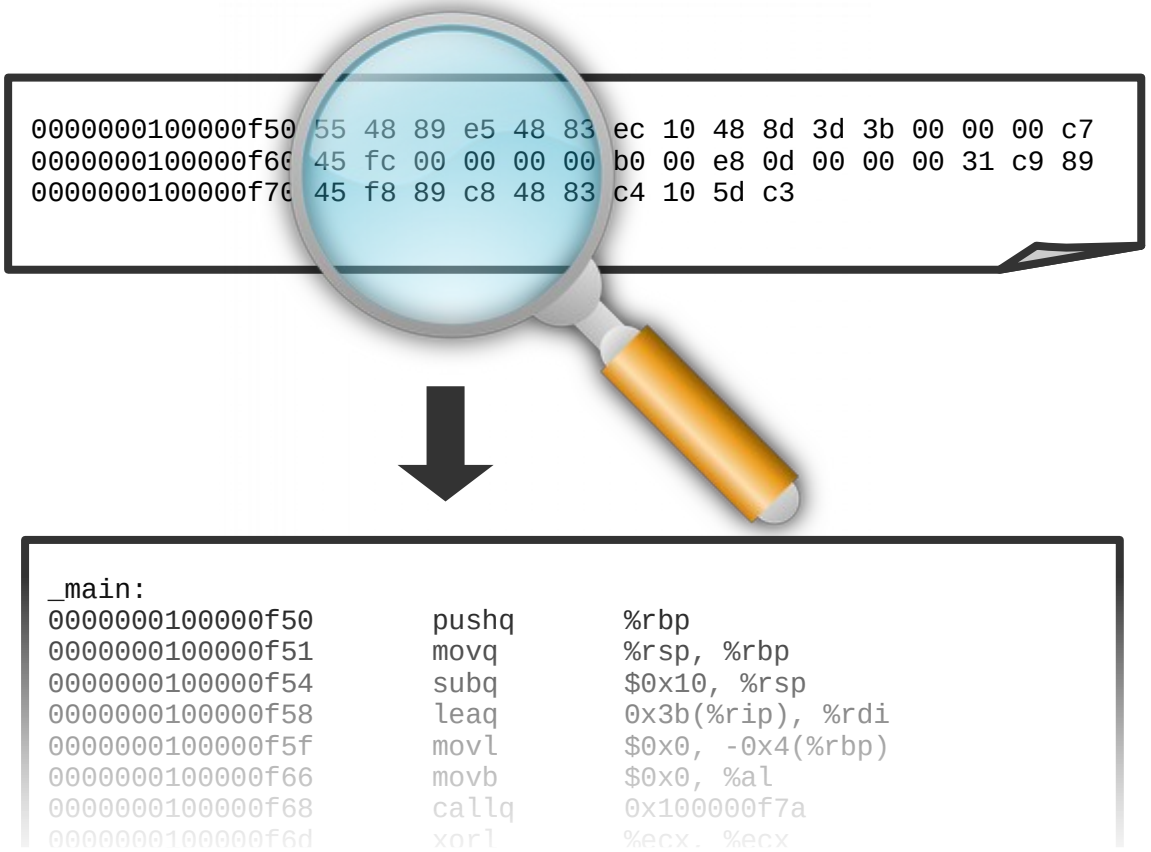
CS 261
Fall 2019

Mike Lam, Professor



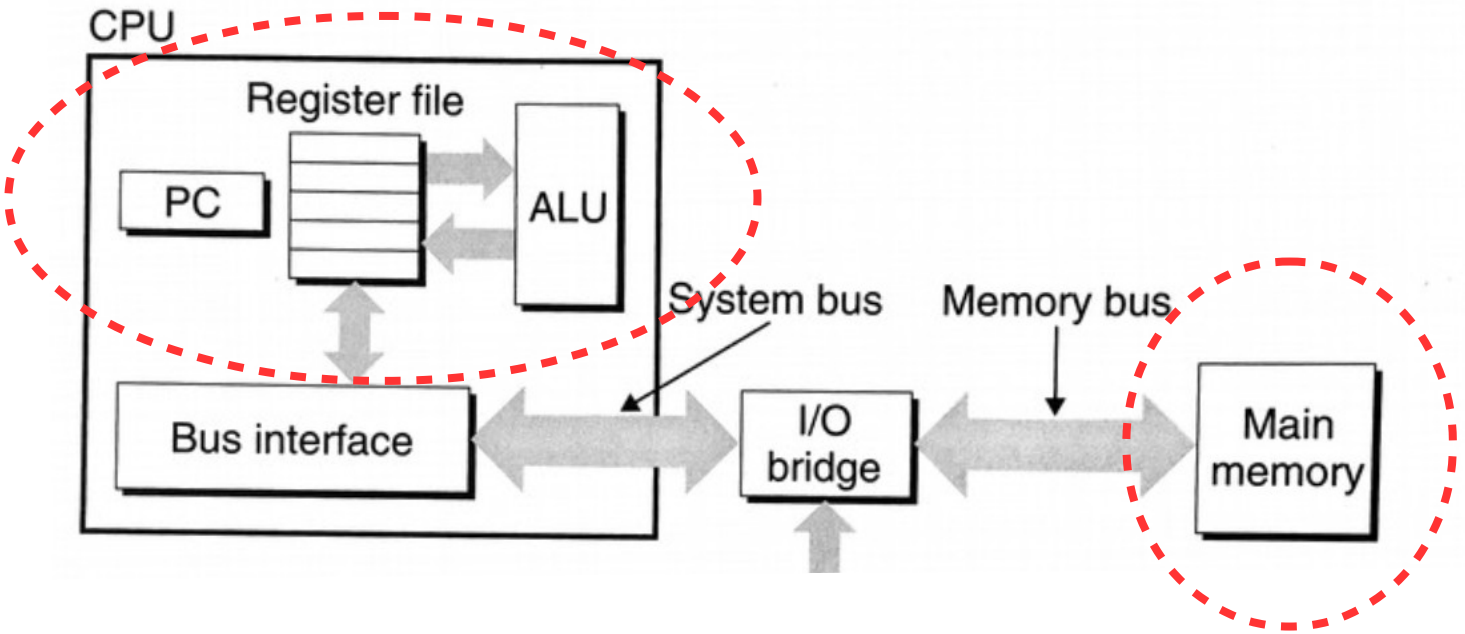
Machine and Assembly Code

Data Movement and Arithmetic

Topics

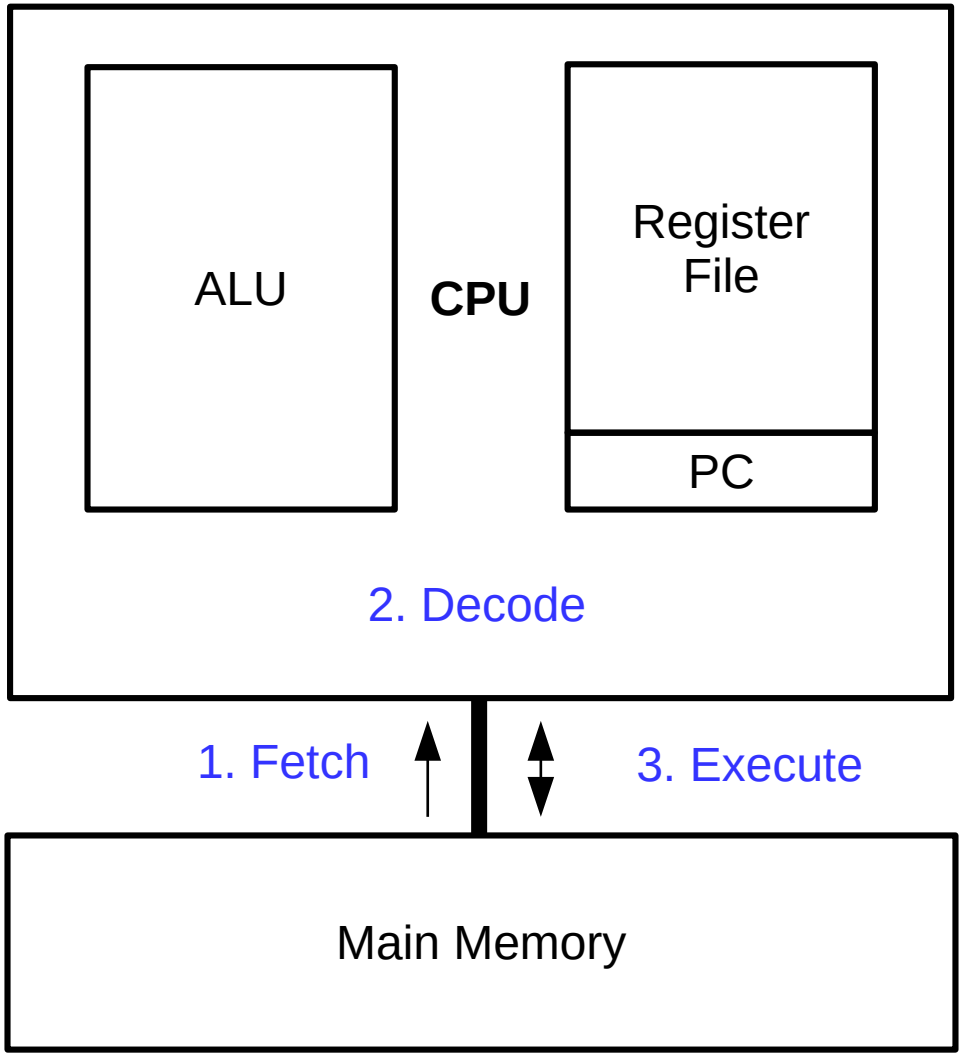
- Architecture/assembly intro
- Data formats
- Data movement
- Arithmetic and logical operations

Computer systems



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

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 the high-performance market (supercomputers!)
    - RISC-V is used in CPU research (and is growing in the industrial 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

```
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
```

opcode operands

```
push   %rbp
mov    %rsp,%rbp
mov    %edi,-0x4(%rbp)
mov    %esi,-0x8(%rbp)
mov    -0x4(%rbp),%edx
mov    -0x8(%rbp),%eax
add    %edx,%eax
pop    %rbp
retq
```

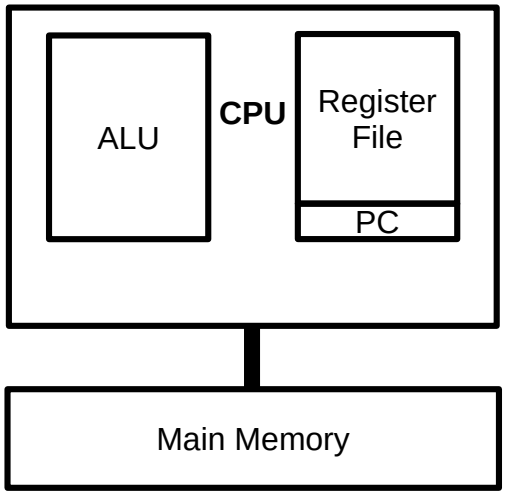
Assembly code

- 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

```
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
```

opcode operands

```
push   %rbp
mov    %rsp,%rbp
mov    %edi,-0x4(%rbp)
mov    %esi,-0x8(%rbp)
mov    -0x4(%rbp),%edx
mov    -0x8(%rbp),%eax
add    %edx,%eax
pop    %rbp
retq
```



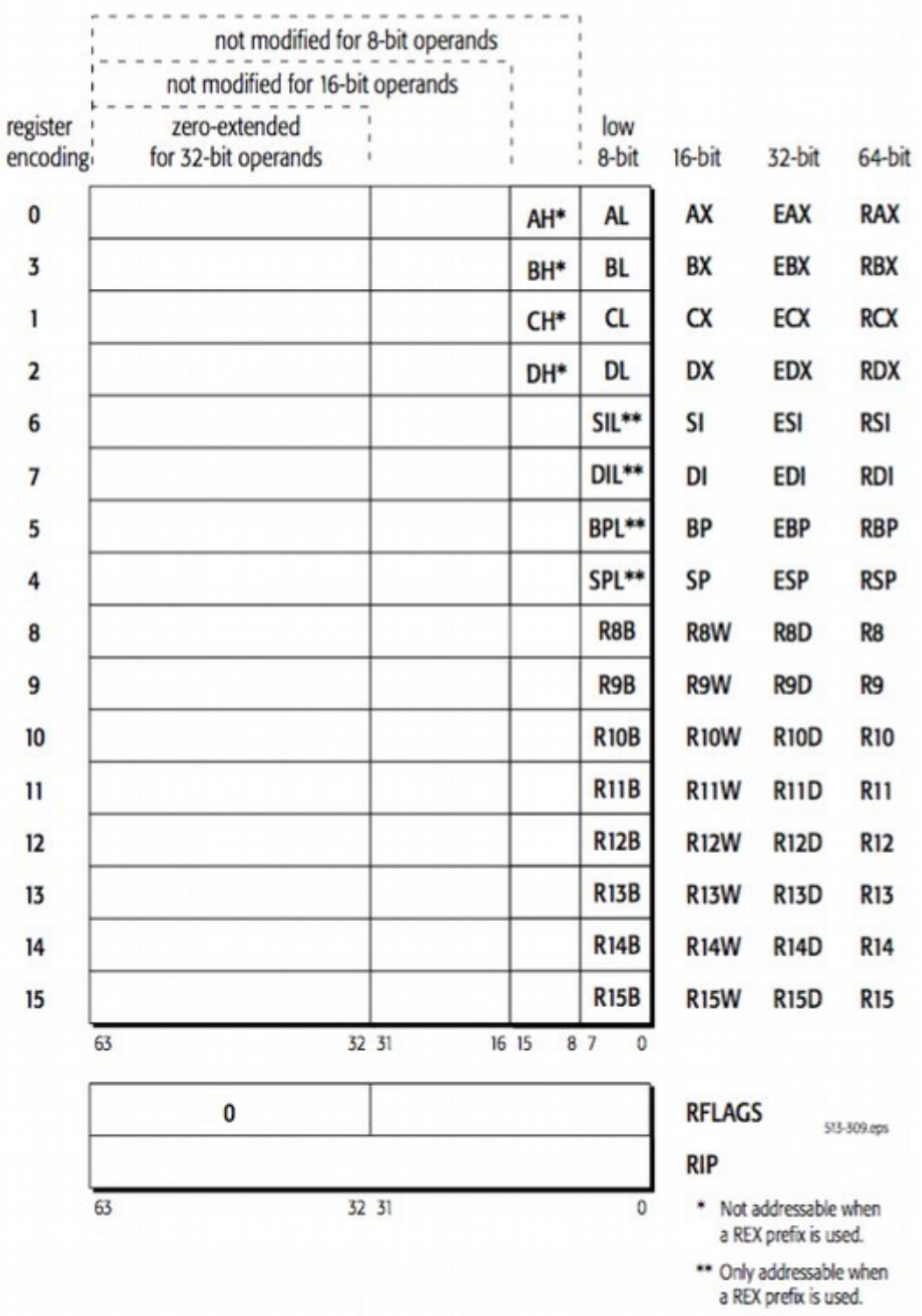
Operand types

- Immediate
  - Operand value 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

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`)



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)

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

Name	Value
%rax	0x100
%rdx	0x2

Memory

Address	Value
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" (`l` 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`
  - `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`

byte-to-word

x86-64 addresses

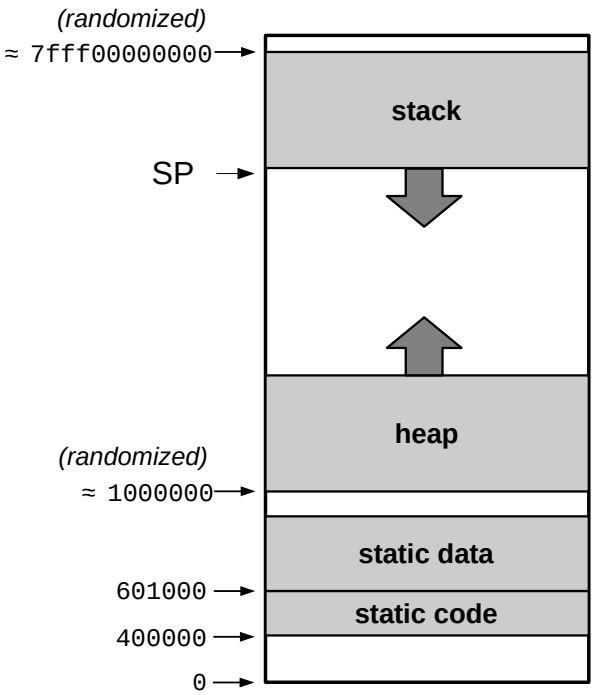
- Addresses in x86-64 are always 32 or 64 bits
  - Thus, the registers used to calculate the effective address of a memory operand must be 32 or 64 bits
    - E.g., `movw %ax, (%ebp)` is valid
    - E.g., `movw %ax, (%rbp)` is valid
    - E.g., `movw %ax, (%bp)` is **not valid**!
    - E.g., `movw %ax,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

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?)

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 (`%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

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 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
```

Hand-writing x86_64 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