

# x86-64 Assembly

---

CS 2130: Computer Systems and Organization 1

March 13, 2023

# Announcements

- Homework 5 available, due next Monday at 11pm on Gradescope
- Updated extension policy available on the website
- Results from regrade requests should be available by next week
- No Prof. Hott office hours this week!
- Lab tomorrow: using the lldb debugger

## General principle of all **assembly languages**

- Code (text, not binary!)
- 1 line of code = 1 machine instruction
- One-to-one reversible mapping between binary and assembly
  - We do not need to remember binary encodings!
  - A program will turn text to binary for us!

# AT&T x86-84 Assembly

- `instruction source, destination`
  - Instruction followed by 0 or more operands (arguments)
- `mylabelname:`
  - Label - remember the address of next thing to use later
- `.something something`
  - Metadirective - extra information that is not code
- `// comments!`
- Address Calculations: `2130(%rax, %rsp, 8)`
  - Combines as:  $2130 + \%rax + (\%rsp * 8)$
  - This is all one operand (one memory address)

# Registers

`rax` is a 64-bit register

# Instructions

Instructions have different versions depending on number of bits to use

- **movq** - 64-bit move
  - q = quad word
- **movl** - 32-bit move
  - l = long
- There are encodings for shorter things, but we will mostly see 32- and 64-bit

# More powerful than our ISA

Instructions can move/operate between memory and register

- `movq %rax, %rcx` - register to register
  - Remember our icode 0
- `movq (%rax), %rcx` - memory to register
  - Remember our icode 3
- `movq %rax, (%rcx)` - register to memory
  - Remember our icode 4
- `movq $21, %rax` - Immediate to register
  - Remember our icode 6 (b=0)

*Note: at most one memory address per instruction*

# Other Instructions

Other instructions work the same way

- `addq %rax, %rcx` — `rcx += rax`
- `subq (%rbx), %rax` — `rax -= M[rbx]`
- `xor`, `and`, and others work the same way!
- Assembly has virtually no 3-argument instructions
  - All will be modifying something (i.e., `+=`, `&=`, ...)

Load effective address: `leaq 4(%rcx), %rax`

- Performs memory address calculation
- Stores address, not value at the address in memory



# Jumps

`jmp foo`

- Unconditional jump to `foo`
- `foo` is a label or memory address
- Need `jmp*` to use register value

Conditional jumps

- `jl, jle, je, jne, jg, jge, ja, jb, js, jo`

Unlike our Toy ISA, these do not compare given register to 0

# Jumps

**Condition codes** - 4 1-bit registers set by every math operation, `cmp`, and `test`

- Result for the operation compared to 0 (if no overflow)

- Example:

```
addq $-5, %rax
```

```
// ...code that doesn't set condition codes...
```

```
je foo
```

- Sets condition codes from doing math (subtract 5 from rax)
- Tells whether result was positive, negative, 0, if there was overflow, ...
- Then jump if the result of that operation should have been = 0

# Jumps: compare and test

`cmpq %rax, %rdx`

- Compare checks result of `- =` and sets condition codes
- How `rdx - rax` compares with 0
- Be aware of ordering!
  - if `rax` is bigger, sets `<` flag
  - if `rdx` is bigger, sets `>` flag

`testq %rax, %rdx`

- Sets the condition codes based on `rdx & rax`
- Less common

*Neither save their result, just set condition codes!*

# Function Calls: Calling Conventions

## `callq myfun`

- Push return address, then jump to myfun
- Convention: Store arguments in registers and stack before call
  - First 6 arguments (in order): `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`
  - If more arguments, pushed onto stack (last to first)

## `retq`

- Pop return address from stack and jump back
- Convention: store return value in `rax` before calling `retq`

*This is similar to our Toy ISA's function calls in homework 4*

Debugger - step through code!

- Similar experience to our ToyISA simulators
- You will be using `lldb` for lab tomorrow
- Experience seeing results of these instructions step-by-step
- **Please read the x86-64 summary reading before lab!**

example with lldb