x86-64

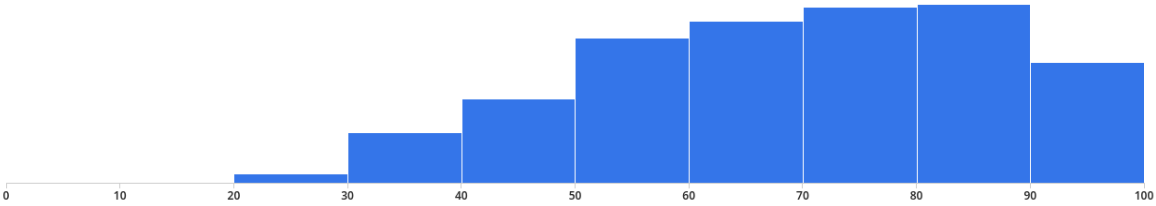
CS 2130: Computer Systems and Organization 1
October 15, 2025

Welcome!
Welcome!

Announcements

- Homework 5 available today, **due Monday at 11:59pm** on Gradescope
- Exam 1 scores released

Exam 1 Scores



	Mean	Median
Overall	68.92	70.0
Suspected Gen AI use on HW3	64.55	63.0
No suspected Gen AI use on HW3	70.18	73.0

Assembly

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!

Assembly

Features of assembly

- Automatic addresses - use **labels** to keep track of addresses
  - Assembler will remember location of labels and use where appropriate
  - Labels will not exist in machine code
- Metadata - data about data
  - Data that helps turn assembly into code the machine can use
- As complicated as machine instructions
  - There are a lot of instructions, and it is one-to-one!

Assembly Languages

There are many assembly languages

- But, they're backed by hardware!
- Two big ones these days: x86-64 and ARM
  - You likely have machines that use one of these
- Others: RISC-V, MIPS, ...

We will focus on **x86-64**

AT&T x86-84 Assembly

`instruction source, destination`

- Instruction followed by `0` or more operands (arguments)
- 4 types of operands:
  - Number (immediate value): `$0x123`
  - Register: `%rax`
  - Address of memory: `(%rax)` or `24` or `labelname`
  - Value at an address in memory: `(%rax)` or `24` or `labelname`

AT&T x86-84 Assembly

`mylabelname:`

- Label - remember the address of next thing to use later

`.something something`

- Metadirective - extra information that is not code
- How the code works with other things (i.e., talk to OS)
- Ex: `.globl main`

`// we can have comments!`

Addressing Memory

`2130(%rax, %rsp, 8)`   Cat c = new Cat(~)

c Cat
name
age
x

- Address can have up to 4 parts: 2 numbers, 2 registers
- Combines as: `2130 + %rax + (%rsp * 8)` ← math
- Common usage from this example:

c.x[3]

  - `rax` - address of an object in memory
  - `2130` - offset of an array into the object
  - `rsp` - index into the array
  - `8` - size of the values in the array

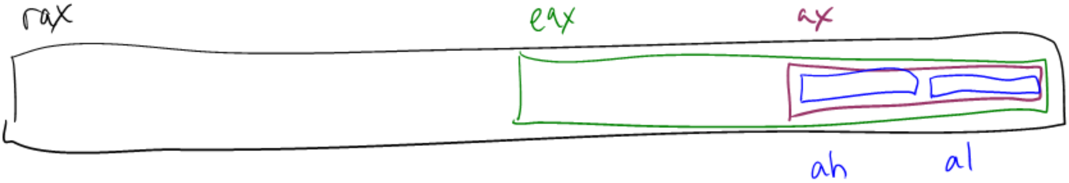
x[0] x[1] x[2]
0 +8 8 +8 16 24
8 byte 8 byte 8 byte

- Don't need all parts: `(%rax)` or `(%rax, 4)` or `4(%rax)`
- This is all one operand (one memory address)

hello.s example

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   inst src, dest
  - Remember our icode 0
- `movq (%rax), %rcx` - memory to register
  - Remember our icode ~~3~~ 4
- `movq %rax, (%rcx)` - register to memory
  - Remember our icode ~~4~~ 5
- `movq $21, %rax` - Immediate to register
  - Remember our icode 6 (b=0)

Note: at most one memory address per instruction

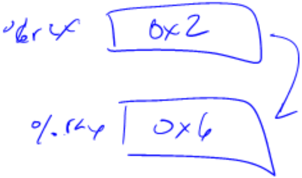
Other Instructions

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., +=, &=, …)

rcx = rcx + rax

Load Effective Address



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

>0 <0 s overflow

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

<0 $<=0$ $=0$ $!=0$ >0 $>=0$ unsigned

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
```

rax += -5

=

  - 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 operation should have been $= 0$

Jumps: compare...

rdx -= rax ⇒ rdx = rdx − rax

```
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

Jumps: … and test

```
testq %rax, %rdx
```

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

Neither save their result, just set condition codes!