X86_64, Assembly

CS 2130: Computer Systems and Organization 1

Xinyao Yi Ph.D.
Assistant Professor

Announcements

- Homework 4 due tonight on Gradescope
- Homework 5 available soon, due Monday at 11:59pm on Gradescope

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!

- ISA is like the grammar and vocabulary of a language.
- Assembly code is a sentence written in that language.

Assembly

Features of assembly

- Automatic addresses - use **labels** to keep track of addresses
  - Assembler will remember location of labels and use where appropriate

    It's going to replace them with the actual addresses when it builds the binary that we're going to run.
  - Labels will not exist in machine code

    (.text .data .byte)
- Metadata - data about data (extra information)
  - 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

Each CPU family has its own unique set of machine instructions – therefore it's own assembly language.

- But, they're backed by hardware!
- Two big ones these days: x86-64 and ARM
  most computers
  M1/M2 chips on MAC and cellphones
  – You likely have machines that use one of these
- Others: RISC-V, MIPS, ... PowerPC

We will focus on **x86-64**

UNIVERSITY of VIRGINIA

x86-64

x86-64 has a weird and long history

- Expansion of the 8086 series (Intel)

8 bits / 16 bits / 32 bits

8008 – 8086, 8286, 8386, 8486, <u>x86</u>

- AMD expanded it with <u>AMD64</u> A 64-bit that was backward compatible with X86.
- Intel decide to use same build, but called it x86-64
- Backwards compatible with the 8086 series

x86-64

Two dialects - two ways to write the same thing

- Intel - likely using with Windows

  ```
  mov QWORD PTR [rdx+0x227],rax
  ```

- AT&T - likely using with anything else

  ```
  movq %rax,0x227(%rdx)
  ```

We will use AT&T dialect

UNIVERSITY of VIRGINIA

AT&T x86-84 Assembly

`instruction source, destination`

- Instruction followed by 0 or more operands (arguments)
- 4 types of operands: (typically we will not see more than 2)
  - Number (immediate value): `$0x123`
  - Register: `%rax`
  - lea — Address of memory: `(%rax)` or 24 or `labelname`
  - Value at an address in memory: `(%rax)` or 24 or `labelname`

loading the addresses

In most of the cases, we are doing something using the value. Except for

AT&T x86-84 Assembly

`mylabelname:` end with a colon

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

`.something something` start with a dot

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

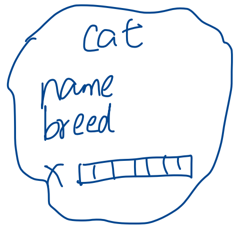
`// we can have comments!`

Addressing Memory

`2130(%rax, %rsp, 8)`

- Address can have up to 4 parts: 2 numbers, 2 registers
- Combines as: $\boxed{\texttt{2130 + \%rax + (\%rsp * 8)}}$
- Common usage from this example:
  - `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 (used to calculate the offset)
- Don't need all parts: (%rax) or 4(%rax)
- This is all one operand (one memory address)

object c



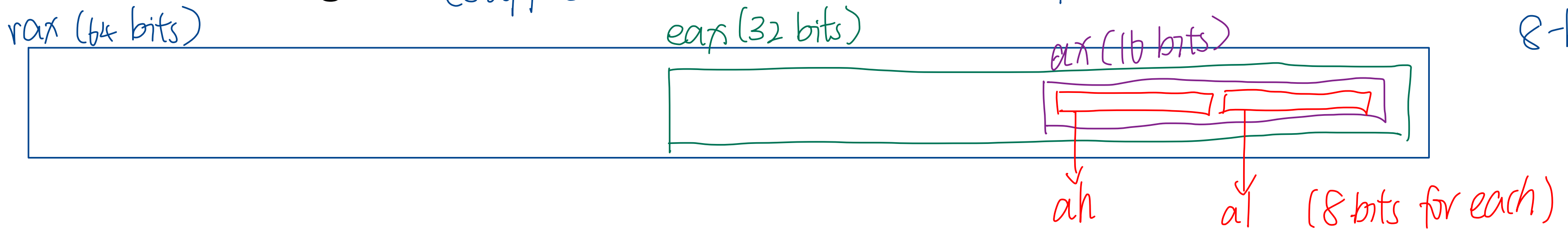
c. x[15]

If I don't have all the pieces, it will calculate what it can.

hello.s example

Registers

rax is a 64-bit register (supposed to be backwards compatible with x86 (32-bit), 16-bit, 8-bit)



If I look at 32-bit version, it will just zero out the top 32 bits.

We'll see this with all our registers, in slightly different way.

(Check the reading)

UNIVERSITY of VIRGINIA

Instructions (short acronyms for what we want to do, like mov, add, and, or, xor, neg)

Instructions have different versions depending on number of bits to use

- `movq` - 64-bit move (similar for addq, subq)
  - q = quad word $\xrightarrow{}$ The instruction followed by how wide of the thing we want to do.
- `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*

We cannot do memory to memory calculations.