

Endianness

CS 2130: Computer Systems and Organization 1

Xinyao Yi Ph.D.
Assistant Professor

Announcements

- Homework 4 **due tonight at 11:59pm** on Gradescope
 - Note the earlier deadline!
 - You have written most of this code already
 - Lab 6 may provide a fast way to get started
- No quiz this weekend!

64-bit Machines

How much can we address with 64-bits?

- 16 EiB (2^{64} addresses = $2^4 \times 2^{60}$)
- But I only have 8 GiB of RAM

A Challenge

There is a disconnect:

- Registers: 64-bits values
- Memory: 8-bit values (i.e., **1 byte** values)
 - Each address addresses an 8-bit value in memory
 - Each address points to a 1-byte slot in memory

A Challenge

There is a disconnect:

- Registers: 64-bits values
- Memory: 8-bit values (i.e., **1 byte** values)
 - Each address addresses an 8-bit value in memory
 - Each address points to a 1-byte slot in memory
- How do we store a 64-bit value in an 8-bit spot?

Rules

Rules to break “big values” into bytes (memory)

1. Break it into bytes
2. Store them adjacently
3. Address of the overall value = smallest address of its bytes
4. Order the bytes
 - If parts are ordered (i.e., array), first goes in smallest address
 - Else, hardware implementation gets to pick (!!)
 - Little-endian
 - Big-endian

Ordering Values

Little-endian

- Store the low order part/byte first
- Most hardware today is little-endian

Big-endian

- Store the high order part/byte first

Example

Store [0x1234, 0x5678] at address 0xF00

Endianness

Why do we study endianness?

- It is **everywhere**
- It is a source of weird bugs
- Ex: It's likely your computer uses:
 - Little-endian from CPU to memory
 - Big-endian from CPU to network
 - File formats are roughly half and half

Moving up!

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**

x86-64

x86-64 has a weird and long history

- Expansion of the 8086 series (Intel)
 - 8086, 8286, 8386, 8486, x86
- AMD expanded it with AMD64
- 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

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)`

- Address can have up to 4 parts: 2 numbers, 2 registers
- Combines as: `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
- 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
 - 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., `+=`, `&=`, ...)

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 operation should have been $= 0$

Jumps: compare...

`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!

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

Debugger - step through code!

- You will be using this for lab 7
- Experience seeing results of these instructions step-by-step
- **Please read the x86-64 summary reading before lab!**