Machine-Level Representation of Programs: Instruction set architectures

Yipeng Huang

Rutgers University

March 5, 2024
Table of contents

Announcements
  Quizzes and programming assignments
  Reading assignments

Floats: Understanding its design
  Deep understanding 1: Why is exp field encoded using bias?
  Deep understanding 2: Why have denormalized numbers?
  Deep understanding 3: Why is bias chosen to be $2^{k-1} - 1$?

Floats: Properties
  Floating point multiplication

Computer organization: A primer

Assembly code: Human readable representation of machine code

Instruction set architectures

swap.s: Assembly implementation of function that swaps memory contents

Data size and IA32, x86, and x86-64 registers
Quizzes and programming assignments

Short quiz 5
  ▶ Due Wednesday. All about floats.

Programming assignment 3
  ▶ Due Friday.
Reading assignments

CS:APP Chapters 3.1-3.4
Table of contents

Announcements
  Quizzes and programming assignments
  Reading assignments

Floats: Understanding its design
  Deep understanding 1: Why is exp field encoded using bias?
  Deep understanding 2: Why have denormalized numbers?
  Deep understanding 3: Why is bias chosen to be $2^{k-1} - 1$?

Floats: Properties
  Floating point multiplication

Computer organization: A primer

Assembly code: Human readable representation of machine code

Instruction set architectures

swap.s: Assembly implementation of function that swaps memory contents

Data size and IA32, x86, and x86-64 registers
The IEEE 754 number line

Figure: Full picture of number line for floating point values. Image credit CS:APP

Figure: Zoomed in number line for floating point values. Image credit CS:APP
## Floats: Summary

<table>
<thead>
<tr>
<th></th>
<th>normalized</th>
<th>denormalized</th>
</tr>
</thead>
<tbody>
<tr>
<td>value of number</td>
<td>$(-1)^s \times M \times 2^E$</td>
<td>$(-1)^s \times M \times 2^E$</td>
</tr>
<tr>
<td>E</td>
<td>$E = \text{exp-bias}$</td>
<td>$E = -\text{bias} + 1$</td>
</tr>
<tr>
<td>bias</td>
<td>$2^{k-1} - 1$</td>
<td>$2^{k-1} - 1$</td>
</tr>
<tr>
<td>exp</td>
<td>$0 &lt; \text{exp} &lt; (2^k - 1)$</td>
<td>$\text{exp} = 0$</td>
</tr>
<tr>
<td>M</td>
<td>$M = 1.\text{frac}$</td>
<td>$M = 0.\text{frac}$</td>
</tr>
<tr>
<td></td>
<td>$M$ has implied leading 1</td>
<td>$M$ has leading 0</td>
</tr>
<tr>
<td></td>
<td>greater range</td>
<td>greater precision</td>
</tr>
<tr>
<td></td>
<td>large magnitude numbers</td>
<td>small magnitude numbers</td>
</tr>
<tr>
<td></td>
<td>denser near origin</td>
<td>evenly spaced</td>
</tr>
</tbody>
</table>

**Table:** Summary of normalized and denormalized numbers
Deep understanding 1: Why is exp field encoded using bias?

exp field needs to encode both positive and negative exponents. Why not just use one of the signed integer formats? 2’s complement, 1s’ complement, signed magnitude?
Deep understanding 1: Why is exp field encoded using bias?

exp field needs to encode both positive and negative exponents. Why not just use one of the signed integer formats? 2’s complement, 1s’ complement, signed magnitude?

Answer: allows easy comparison of magnitudes by simply comparing bits.
Deep understanding 1: Why is exp field encoded using bias?

exp field needs to encode both positive and negative exponents.

Why not just use one of the signed integer formats? 2’s complement, 1s’ complement, signed magnitude?

Answer: allows easy comparison of magnitudes by simply comparing bits.

Consider hypothetical 8-bit floating point format (from the textbook)
1-bit sign, $k = 4$-bit exp, 3-bit frac.

What is the decimal value of $0b1_0110_111$?

$(-1)^{1} \cdot (1.111_2) \cdot 2^7 = -1.0\overline{11} \cdot 2^7$

$E = \text{exp-bias} = 6 - (2^{4-1}) = 6 - (2^3) = -1$

What is the decimal value of $0b1_0111_000$?

$(-1)^{1} \cdot (1.000)_2 \cdot 2^7 = -1.000 \cdot 2^7$

$E = \text{exp-bias} = 7 - 7 = 2^0$
Deep understanding 1: Why is exp field encoded using bias?

exp field needs to encode both positive and negative exponents. Why not just use one of the signed integer formats? 2’s complement, 1’s complement, signed magnitude?

Answer: allows easy comparison of magnitudes by simply comparing bits.

Consider hypothetical 8-bit floating point format (from the textbook)
1-bit sign, \( k = 4 \)-bit exp, 3-bit frac.

What is the decimal value of \( 0b1_0110_111 \)?
\(-1.875 \times 2^{-1}\)

What is the decimal value of \( 0b1_0111_000 \)?
\(-2.000 \times 2^{-1}\)
Deep understanding 2: Why have denormalized numbers?

Why not just continue normalized number scheme down to smallest numbers around zero?
Answer: makes sure that smallest increments available are maintained around zero.

Suppose denormalized numbers NOT used.

What is the decimal value of 0b0_0000_001? 1.125 \times 2^{-7}

What is the decimal value of 0b0_0000_111? 1.875 \times 2^{-7}

What is the decimal value of 0b0_0001_000? 2.000 \times 2^{-7}
Deep understanding 2: Why have denormalized numbers?

Why not just continue normalized number scheme down to smallest numbers around zero?
Answer: makes sure that smallest increments available are maintained around zero.

Suppose denormalized numbers ARE used.

What is the decimal value of 0b0_0000_001? 0.125 \times 2^{-6}
What is the decimal value of 0b0_0000_111? 0.875 \times 2^{-6}
What is the decimal value of 0b0_0001_000? 1.000 \times 2^{-6}
Floats: Special cases

<table>
<thead>
<tr>
<th>number class</th>
<th>when it arises</th>
<th>exp field</th>
<th>frac field</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0 / -0</td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>+infinity / -infinity</td>
<td>overflow or division by 0</td>
<td>$2^k - 1$</td>
<td>0</td>
</tr>
<tr>
<td>NaN not-a-number</td>
<td>illegal ops. such as $\sqrt{-1}$, inf-inf, inf*0</td>
<td>$2^k - 1$</td>
<td>non-0</td>
</tr>
</tbody>
</table>

Table: Summary of special cases

- $\frac{1}{\infty} = \infty$
- $\frac{1}{-\infty} = -\infty$
- $\frac{1}{0.0} = \infty$
- $\frac{1}{-0.0} = -\infty$
- $\frac{1}{\infty} = 0.0$
- $\frac{1}{-\infty} = -0.0$
Table of contents

Announcements
  Quizzes and programming assignments
  Reading assignments

Floats: Understanding its design
  Deep understanding 1: Why is exp field encoded using bias?
  Deep understanding 2: Why have denormalized numbers?
  Deep understanding 3: Why is bias chosen to be $2^{k-1} - 1$?

Floats: Properties
  Floating point multiplication

Computer organization: A primer

Assembly code: Human readable representation of machine code

Instruction set architectures

swap.s: Assembly implementation of function that swaps memory contents

Data size and IA32, x86, and x86-64 registers
How to multiply scientific notation?

Recall: $\log(x \times y) = \log(x) + \log(y)$
Floating point multiplication

**FP Multiplication**

- \((-1)^{s1} \ M_1 \ 2^{E_1} \times (-1)^{s2} \ M_2 \ 2^{E_2}\)

- **Exact Result:** \((-1)^{s} \ M \ 2^{E}\)
  - Sign \(s\): \(s1 \oplus s2\)
  - Significand \(M\): \(M_1 \times M_2\)
  - Exponent \(E\): \(E_1 + E_2\)

- **Fixing**
  - If \(M \geq 2\), shift \(M\) right, increment \(E\)
  - If \(E\) out of range, overflow
  - Round \(M\) to fit \(\text{frac}\) precision

- **Implementation**
  - Biggest chore is multiplying significands
Table of contents

Announcements
  Quizzes and programming assignments
  Reading assignments

Floats: Understanding its design
  Deep understanding 1: Why is exp field encoded using bias?
  Deep understanding 2: Why have denormalized numbers?
  Deep understanding 3: Why is bias chosen to be $2^{k-1} - 1$?

Floats: Properties
  Floating point multiplication

Computer organization: A primer

Assembly code: Human readable representation of machine code

Instruction set architectures

swap.s: Assembly implementation of function that swaps memory contents

Data size and IA32, x86, and x86-64 registers
Computer organization

Layer cake: remember the first day of class, we discussed what are parts of a computer?

- Society
- Human beings
- Applications
- Algorithms
- High-level programming languages
- Interpreters
- Low-level programming languages
- Compilers
- Architectures
- Microarchitectures
- Sequential/combinational logic
- Transistors
- Semiconductors

ISA - Instruction set architecture
Stored program computer

Stored program:
Instructions reside in memory, loaded as needed.

von Neumann architecture:
Data and instructions share same connection to memory.

Assembly/Machine Code View

Programmer-Visible State

- **PC**: Program counter
  - Address of next instruction
  - Called “RIP” (x86-64)
- **Register file**
  - Heavily used program data
- **Condition codes**
  - Store status information about most recent arithmetic or logical operation
  - Used for conditional branching

Memory

- Byte addressable array
- Code and user data
- Stack to support procedures
## Memory hierarchy

<table>
<thead>
<tr>
<th></th>
<th>Capacity</th>
<th>Access speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internet</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tape</td>
<td>250Pb</td>
<td></td>
</tr>
<tr>
<td>Hard drives</td>
<td>16TB</td>
<td>2Mb/s</td>
</tr>
<tr>
<td>Solid state drives</td>
<td>4TB</td>
<td>2Gb/s</td>
</tr>
<tr>
<td>DRAM</td>
<td>8Gb - 1Tb+</td>
<td>8Gb/s</td>
</tr>
<tr>
<td>Last-level cache</td>
<td>64Mb</td>
<td></td>
</tr>
<tr>
<td>Level-1 cache</td>
<td>1Mb</td>
<td></td>
</tr>
<tr>
<td>Registers</td>
<td>1Kb</td>
<td>1000 · 4 · Gb/s</td>
</tr>
</tbody>
</table>

- Registers (.25ns; 4GHz => .25e-9s)
Table of contents

Announcements
   Quizzes and programming assignments
   Reading assignments

Floats: Understanding its design
   Deep understanding 1: Why is exp field encoded using bias?
   Deep understanding 2: Why have denormalized numbers?
   Deep understanding 3: Why is bias chosen to be $2^{k-1} - 1$?

Floats: Properties
   Floating point multiplication

Computer organization: A primer

Assembly code: Human readable representation of machine code

Instruction set architectures

swap.s: Assembly implementation of function that swaps memory contents

Data size and IA32, x86, and x86-64 registers
Unraveling the compilation chain

Turning C into Object Code

- Code in files `p1.c p2.c`
- Compile with command: `gcc -Og p1.c p2.c -o p`
  - Use basic optimizations (`-Og`) [New to recent versions of GCC]
  - Put resulting binary in file `p`

![Diagram of compilation stages]

- `text` -> `C program (p1.c p2.c)` -> `Compiler (gcc -Og -S)`
- `text` -> `Asm program (p1.s p2.s)` -> `Assembler (gcc or as)`
- `binary` -> `Object program (p1.o p2.o)` -> `Linker (gcc or ld)`
- `binary` -> `Executable program (p)`

Static libraries (.a)

Figure: Stages of compilation. Image credit CS:APP
Assembly

Human readable machine code

- Very limited
- Not much control flow
- Any more complex functionality is built up
- for loops, while loops, turn into assembly sequence

Choice of what assembly to experiment with

- MIPS
- ARM
- x86 / x86-64 (not ideal for teaching, but it allows us to experiment on ilab)
Assembly instructions

Instructions for the microarchitecture

- Binary streams that tell an electronic circuit what to do
- Fetch, decode, execute, memory, writeback
A preview of microarchitecture

Figure: Stages of compilation. Image credit Wikimedia
Table of contents

Announcements
  Quizzes and programming assignments
  Reading assignments

Floats: Understanding its design
  Deep understanding 1: Why is exp field encoded using bias?
  Deep understanding 2: Why have denormalized numbers?
  Deep understanding 3: Why is bias chosen to be $2^{k-1} - 1$?

Floats: Properties
  Floating point multiplication

Computer organization: A primer

Assembly code: Human readable representation of machine code

Instruction set architectures

swap.s: Assembly implementation of function that swaps memory contents

Data size and IA32, x86, and x86-64 registers
Why are instruction set architectures important

Interface between computer science and electrical and computer engineering

- Software is varied, changes
- Hardware is standardized, static

Computer architect Fred Brooks and the IBM 360

- IBM was selling computers with different capacities,
- Compile once, and can run software on all IBM machines.
- Backward compatibility.
- An influential idea.
CISC vs. RISC

Complex instruction set computer

- Intel and AMD
- Have an extensive and complex set of instructions
- For example: x86’s extensions: x87, IA-32, x86-64, MMX, 3DNow!, SSE, SSE2, SSE3, SSSE3, SSE4, SSE4.2, SSE5, AES-NI, CLMUL, RDRAND, SHA, MPX, SGX, XOP, F16C, ADX, BMI, FMA, AVX, AVX2, AVX512, VT-x, VT-d, AMD-V, AMD-Vi, TSX, ASF
- Can license Intel’s compilers to extract performance
- Secret: inside the processor, they break it down to more elementary instructions
CISC vs. RISC

Reduced instruction set computer

- MIPS, ARM, RISC-V (can find Patterson and Hennessy Computer Organization and Design textbook in each of these versions), and PowerPC
- Have a relatively simple set of instructions
- For example: ARM’s extensions: SVE;SVE2;TME; All mandatory: Thumb-2, Neon, VFPv4-D16, VFPv4 Obsolete: Jazelle
- ARM: smartphones, Apple ARM M1 Mac
Post-ISA world

- Increasingly, the CPU is not the only character
- It orchestrates among many pieces of hardware
- Smartphone die shot
- GPU, TPU, FPGA, ASIC

**Figure:** Apple A13 (2019 Apple iPhone 11 CPU). Image credit AnandTech
Table of contents

Announcements
  Quizzes and programming assignments
  Reading assignments

Floats: Understanding its design
  Deep understanding 1: Why is exp field encoded using bias?
  Deep understanding 2: Why have denormalized numbers?
  Deep understanding 3: Why is bias chosen to be $2^{k-1} - 1$?

Floats: Properties
  Floating point multiplication

Computer organization: A primer

Assembly code: Human readable representation of machine code

Instruction set architectures

swap.s: Assembly implementation of function that swaps memory contents

Data size and IA32, x86, and x86-64 registers
Unraveling the compilation chain

Turning C into Object Code

- Code in files `p1.c` `p2.c`
- Compile with command: `gcc -Og p1.c p2.c -o p`
  - Use basic optimizations (`-Og`) [New to recent versions of GCC]
  - Put resulting binary in file `p`

<table>
<thead>
<tr>
<th>text</th>
<th>C program (<code>p1.c</code> <code>p2.c</code>)</th>
<th>Compiler (<code>gcc -Og -S</code>)</th>
</tr>
</thead>
<tbody>
<tr>
<td>text</td>
<td>Asm program (<code>p1.s</code> <code>p2.s</code>)</td>
<td>Assembler (<code>gcc or as</code>)</td>
</tr>
<tr>
<td>binary</td>
<td>Object program (<code>p1.o</code> <code>p2.o</code>)</td>
<td>Linker (<code>gcc or ld</code>)</td>
</tr>
<tr>
<td>binary</td>
<td>Executable program (<code>p</code>)</td>
<td>Static libraries (<code>.a</code>)</td>
</tr>
</tbody>
</table>
Data movement instructions

Does unsigned / signed matter?

1. void swap_uc ( unsigned char*a, unsigned char*b );
2. void swap_sc ( signed char*a, signed char*b );

Swapping different data sizes

1. void swap_c ( char*a, char*b );
2. void swap_s ( short*a, short*b );
3. void swap_i ( int*a, int*b );
4. void swap_l ( long*a, long*b );
Table of contents

Announcements
  Quizzes and programming assignments
  Reading assignments

Floats: Understanding its design
  Deep understanding 1: Why is exp field encoded using bias?
  Deep understanding 2: Why have denormalized numbers?
  Deep understanding 3: Why is bias chosen to be $2^{k-1} - 1$?

Floats: Properties
  Floating point multiplication

Computer organization: A primer

Assembly code: Human readable representation of machine code

Instruction set architectures

swap.s: Assembly implementation of function that swaps memory contents

Data size and IA32, x86, and x86-64 registers
Data size and x86 / x86-64 registers

**Assembly syntax**

Instruction Source, Dest

```assembly
swap_l:
    movq (%rsi), %rax
    movq (%rdi), %rdx
    movq %rdx, (%rsi)
    movq %rax, (%rdi)
    ret
```

<table>
<thead>
<tr>
<th>swap</th>
<th>data type</th>
<th>mov operation</th>
<th>registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>swap_uc</td>
<td>unsigned char</td>
<td>movb (move byte)</td>
<td>%al, %dl</td>
</tr>
<tr>
<td>swap_sc</td>
<td>signed char</td>
<td>movb (move byte)</td>
<td>%al, %dl</td>
</tr>
<tr>
<td>swap_c</td>
<td>char</td>
<td>movb (move byte)</td>
<td>%al, %dl</td>
</tr>
<tr>
<td>swap_s</td>
<td>short</td>
<td>movw (move word)</td>
<td>%ax, %dx</td>
</tr>
<tr>
<td>swap_i</td>
<td>int</td>
<td>movl</td>
<td>%eax, %edx</td>
</tr>
<tr>
<td>swap_l</td>
<td>long</td>
<td>movq</td>
<td>%rax, %rdx</td>
</tr>
</tbody>
</table>
Data size and IA32, x86, and x86-64 registers

Some History: IA32 Registers

<table>
<thead>
<tr>
<th>General Purpose</th>
<th>Origin (mostly obsolete)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>%ax</td>
</tr>
<tr>
<td>%ecx</td>
<td>%cx</td>
</tr>
<tr>
<td>%edx</td>
<td>%dx</td>
</tr>
<tr>
<td>%ebx</td>
<td>%bx</td>
</tr>
<tr>
<td>%esi</td>
<td>%si</td>
</tr>
<tr>
<td>%edi</td>
<td>%di</td>
</tr>
<tr>
<td>%esp</td>
<td>%sp</td>
</tr>
<tr>
<td>%ebp</td>
<td>%bp</td>
</tr>
</tbody>
</table>

- Accumulate
- Counter
- Data
- Base
- Source
- Index
- Destination
- Stack
- Pointer
- Base
- Pointer

Note the backward compatibility.
Data size and IA32, x86, and x86-64 registers

<table>
<thead>
<tr>
<th>data type</th>
<th>registers</th>
<th>x86-64 Integer Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>%al, %dl</td>
<td>%rax, %eax</td>
</tr>
<tr>
<td>short</td>
<td>%ax, %dx</td>
<td>%rbx, %ebx</td>
</tr>
<tr>
<td>int</td>
<td>%eax, %edx</td>
<td>%rcx, %ecx</td>
</tr>
<tr>
<td>long</td>
<td>%rax, %rdx</td>
<td>%rdr, %rdx</td>
</tr>
</tbody>
</table>

Note the backward compatibility.

- Can reference low-order 4 bytes (also low-order 1 & 2 bytes)
Data size and IA32, x86, and x86-64 registers

Figure: x86-64 with SIMD extensions registers. Image credit: https://commons.wikimedia.org/wiki/File:Table_of_x86_Registers_svg.svg