Floating point topics
- Fractional binary numbers
- IEEE floating-point standard
- Floating-point operations and rounding
- Floating-point in C

There are many more details that we won’t cover
- It’s a 58-page standard...

Distribution of Values
- What ranges are NOT representable?
  - Between largest norm and infinity
    - Overflow (Exp too large)
  - Between zero and smallest denorm
    - Underflow (Exp too small)
  - Between norm numbers?
    - Rounding
- Given a FP number, what’s the bit pattern of the next largest representable number?
  - What is this “step” when Exp = 0?
  - What is this “step” when Exp = 100?
- Distribution of values is denser toward zero

Floating Point Encoding Summary

<table>
<thead>
<tr>
<th>Exponent</th>
<th>Mantissa</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>0</td>
<td>± 0</td>
</tr>
<tr>
<td>0x00</td>
<td>non-zero</td>
<td>± denorm num</td>
</tr>
<tr>
<td>0x01 – 0xFE</td>
<td>anything</td>
<td>± norm num</td>
</tr>
<tr>
<td>0xFF</td>
<td>0</td>
<td>± ∞</td>
</tr>
<tr>
<td>0xFF</td>
<td>non-zero</td>
<td>NaN</td>
</tr>
</tbody>
</table>

Floating Point Operations: Basic Idea

\[ \text{Value} = (-1)^{E} \times \text{Mantissa} \times 2^{\text{Exponent}} \]

- \( x + \epsilon \ y = \text{Round}(x + y) \)
- \( x \times \epsilon \ y = \text{Round}(x \times y) \)

Basic idea for floating point operations:
- First, compute the exact result
- Then round the result to make it fit into desired precision:
  - Possibly over/underflow if exponent outside of range
  - Possibly drop least-significant bits of mantissa to fit into M bit vector
Floating Point Addition

\[ (-1)^{S_1} \times \text{Man}_1 \times 2^{\text{Exp}_1} + (-1)^{S_2} \times \text{Man}_2 \times 2^{\text{Exp}_2} \]

- Assume \( \text{Exp}_1 > \text{Exp}_2 \)

- Exact Result: \( (-1)^{S} \times \text{Man} \times 2^{\text{Exp}} \)

- Sign \( S \), mantissa \( \text{Man} \):
  - Result of signed align & add
- Exponent \( E \): \( E_1 \)

- Adjustments:
  - If \( \text{Man} \geq 2 \), shift \( \text{Man} \) right, increment \( \text{Exp} \)
  - If \( \text{Man} < 1 \), shift \( \text{Man} \) left \( k \) positions, decrement \( \text{Exp} \) by \( k \)
  - Over/underflow if \( \text{Exp} \) out of range
  - Round \( \text{Man} \) to fit mantissa precision

Floating Point Multiplication

\[ (-1)^{S_1} \times \text{Man}_1 \times 2^{\text{Exp}_1} \times (-1)^{S_2} \times \text{Man}_2 \times 2^{\text{Exp}_2} \]

- Exact Result: \( (-1)^{S} \times \text{M} \times 2^{E} \)

- Sign \( S \): \( S_1 \wedge S_2 \)
- Mantissa \( \text{Man} \): \( \text{Man}_1 \times \text{Man}_2 \)
- Exponent \( E \): \( \text{Exp}_1 + \text{Exp}_2 \)

- Adjustments:
  - If \( \text{Man} \geq 2 \), shift \( \text{Man} \) right, increment \( \text{Exp} \)
  - Over/underflow if \( \text{Exp} \) out of range
  - Round \( \text{Man} \) to fit mantissa precision

Mathematical Properties of FP Operations

- Exponent overflow yields \( +\infty \) or \( -\infty \)
- Floats with value \( +\infty \), \( -\infty \), and NaN can be used in operations
- Result usually still \( +\infty \), \( -\infty \), or NaN; but not always intuitive
- Floating point operations do not work like real math, due to rounding
  - Not associative: \( 3.14+1e100-1e100 \neq 3.14+(1e100-1e100) \)
  - Not distributive: \( 100\times(0.1+0.2) \neq 100\times0.1+100\times0.2 \)
  - Not cumulative
    - Repeatedly adding a very small number to a large one may do nothing

Floating Point in C

- Offers two (well, 3) levels of precision
  - float 1.0f single precision (32-bit)
  - double 1.0 double precision (64-bit)
  - long double 1.0L ("double double" or quadruple) precision (64-128 bits)

- \#include <math.h> to get INFINITY and NaN constants
- Equality (==) comparisons between floating point numbers are tricky, and often return unexpected results, so just avoid them!

Floating Point Conversions in C

- Casting between int, float, and double changes the bit representation
  - int \( \rightarrow \) float
    - May be rounded (not enough bits in mantissa: 23)
    - Overflow impossible
  - int or float \( \rightarrow \) double
    - Exact conversion (all 32-bit int representable)
  - long \( \rightarrow \) double
    - Depends on word size (32-bit is exact, 64-bit may be rounded)
  - double or float \( \rightarrow \) int
    - Truncates fractional part (rounded toward zero)
    - "Not defined" when out of range or NaN: generally sets to \( \text{Tmin} \)
      (even if the value is a very big positive)
Floating Point and the Programmer

```c
#include <stdio.h>
int main(int argc, char* argv[]) {
    float f1 = 1.0;
    float f2 = 0.0;
    int i;
    for (i = 0; i < 10; i++)
        f2 += 1.0/10.0;
    printf("0x%08x 0x%08x\n", *(int*)&f1, *(int*)&f2);
    printf("f1 = %10.9f\n", f1);
    printf("f2 = %10.9f\n\n", f2);
    f1 = 1E30;
    f2 = 1E-30;
    float f3 = f1 + f2;
    printf("f1 == f3? %s\n", f1 == f3 ? "yes" : "no" );
    return 0;
}
```

Floating Point Summary

- Floats also suffer from the fixed number of bits available to represent them
- Can get overflow/underflow
- “Gaps” produced in representable numbers means we can lose precision, unlike `int`
- Some “simple fractions” have no exact representation (e.g. 0.2)
- “Every operation gets a slightly wrong result”
- Floating point arithmetic not associative or distributive
- Mathematically equivalent ways of writing an expression may compute different results
- Never test floating point values for equality!
- Careful when converting between `ints` and `floats`!

Number Representation Really Matters

- 1991: Patriot missile targeting error
  - clock skew due to conversion from integer to floating point
- 1996: Ariane 5 rocket exploded ($1 billion)
  - overflow converting 64-bit floating point to 16-bit integer
- 2000: Y2K problem
  - limited (decimal) representation: overflow, wrap-around
- 2038: Unix epoch rollover
  - Unix epoch = seconds since 12am, January 1, 1970
  - signed 32-bit integer representation rolls over to Tmin in 2038
- Other related bugs:
  - 1982: Vancouver Stock Exchange 10% error in less than 2 years
  - 1994: Intel Pentium FDIV (floating point division) HW bug ($475 million)
  - 1997: USS Yorktown “smart” warship stranded: divide by zero
  - 1998: Mars Climate Orbiter crashed: unit mismatch ($193 million)

Translation

<table>
<thead>
<tr>
<th>Code Time</th>
<th>Compile Time</th>
<th>Run Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>User program in C</td>
<td>C compiler</td>
<td>Hardware</td>
</tr>
<tr>
<td>.c file</td>
<td>.o file</td>
<td>.exe file</td>
</tr>
</tbody>
</table>

What makes programs run faster?

Roadmap

Memory & data
Integers & floats
x86 assembly
Executables
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C

Translation

Different applications or algorithms
Perform optimizations, generate instructions
Different implementations

HW Interface Affects Performance

C Language
Program A
GCC
x86-64
Intel Pentium 4
Intel Core 2
Intel Core i7
AMD Opteron
AMD Athlon
ARMv8 (AArch64/A64)
ARM Cortex-A53
Apple A7
Instruction Set Architectures

- The ISA defines:
  - The system’s state (e.g. registers, memory, program counter)
  - The instructions the CPU can execute
  - The effect that each of these instructions will have on the system state

![Diagram of CPU, PC, Registers, Memory]

Instruction Set Philosophies

- **Complex Instruction Set Computing (CISC):** Add more and more elaborate and specialized instructions as needed
  - Lots of tools for programmers to use, but hardware must be able to handle all instructions
  - x86-64 is CISC, but only a small subset of instructions encountered with Linux programs

- **Reduced Instruction Set Computing (RISC):** Keep instruction set small and regular
  - Easier to build fast hardware
  - Let software do the complicated operations by composing simpler ones

General ISA Design Decisions

- **Instructions**
  - What instructions are available? What do they do?
  - How are they encoded?

- **Registers**
  - How many registers are there?
  - How wide are they?

- **Memory**
  - How do you specify a memory location?

Mainstream ISAs

<table>
<thead>
<tr>
<th>Intel</th>
<th>ARM</th>
<th>MIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>x86</strong></td>
<td><strong>ARM Architecture</strong></td>
<td><strong>MIPS</strong></td>
</tr>
<tr>
<td>Home: Desktop, Mobile</td>
<td>Internet/Server, Embedded</td>
<td>Internet/Server, Embedded</td>
</tr>
<tr>
<td>Design: RISC</td>
<td>Design: RISC</td>
<td>Design: RISC</td>
</tr>
<tr>
<td>Type: Register-Register</td>
<td>Type: Register-Register</td>
<td>Type: Register-Register</td>
</tr>
<tr>
<td>Endurance: Low</td>
<td>Endurance: High</td>
<td>Endurance: High</td>
</tr>
</tbody>
</table>

Macbooks & PCs (Core i3, i5, i7, M)
- x86-64 Instruction Set

Smartphone-like devices (iPhone, iPad, Raspberry Pi)
- ARM Instruction Set

Digital home & networking equipment (Blu-ray, PlayStation 2)
- MIPS Instruction Set

Definitions

- **Architecture (ISA):** The parts of a processor design that one needs to understand to write assembly code
  - “What is directly visible to software”

- **Microarchitecture:** Implementation of the architecture
  - CSE/EE 469, 470

Are the following part of the architecture?
- Number of registers?
- How about CPU frequency?
- Cache size? Memory size?

Assembly Programmer’s View

- **Programmer-visible state**
  - **PC:** the Program Counter (Rip in x86-64)
    - Address of next instruction
  - **Named registers**
    - Together in “register file”
    - Heavily used program data
  - **Condition codes**
    - Store status information about most recent arithmetic operation
    - Used for conditional branching

- **Memory**
  - Byte-addressable array
  - Code and user data
  - Includes the Stack (for supporting procedures)
x86-64 Assembly “Data Types”

∫ Integral data of 1, 2, 4, or 8 bytes
  • Data values
  • Addresses (untyped pointers)
∫ Floating point data of 4, 8, 10 or 2x8 or 4x4 or 8x2
  • Different registers for those (e.g. %xmm1, %ymm2)
  • Come from extensions to x86 (SSE, AVX, …)
∫ No aggregate types such as arrays or structures
  • Just contiguously allocated bytes in memory
∫ Two common syntaxes
  • “AT&T”: used by our course, slides, textbook, gnu tools, …
  • “Intel”: used by Intel documentation, Intel tools, …
  • Must know which you’re reading

What is a Register?

∫ A location in the CPU that stores a small amount of data, which can be accessed very quickly (once every clock cycle)
∫ Registers have names, not addresses
  • In assembly, they start with % (e.g. %rsi)
∫ Registers are at the heart of assembly programming
  • They are a precious commodity in all architectures, but especially x86

x86-64 Integer Registers – 64 bits wide

<table>
<thead>
<tr>
<th>trax</th>
<th>tecx</th>
<th>trcx</th>
<th>trdx</th>
<th>trsi</th>
<th>trdi</th>
<th>trsp</th>
<th>trbp</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>%ecx</td>
<td>%ecx</td>
<td>%edx</td>
<td>%esi</td>
<td>%edi</td>
<td>%esp</td>
<td>%ebp</td>
</tr>
</tbody>
</table>

∫ Can reference low-order 4 bytes (also low-order 2 & 1 bytes)

Some History: IA32 Registers – 32 bits wide

<table>
<thead>
<tr>
<th>leax</th>
<th>texx</th>
<th>tecx</th>
<th>tedx</th>
<th>tesi</th>
<th>tedi</th>
<th>lesp</th>
<th>lebp</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>%ecx</td>
<td>%ecx</td>
<td>%edx</td>
<td>%esi</td>
<td>%edi</td>
<td>%esp</td>
<td>%ebp</td>
</tr>
</tbody>
</table>

16-bit virtual registers (backwards compatibility) Name Origin

(mostly obsolete)

Memory vs. Registers

∫ Addresses
  • 0x7FFFFFFD024C3DC
∫ Big
  • ~ 8 GiB
∫ Slow
  • ~50-100 ns
∫ Dynamic
  • Can “grow” as needed while program runs
∫ Names

Three Basic Kinds of Instructions

1) Transfer data between memory and register
  • Load data from memory into register
    • %reg = Mem[address]
  • Store register data into memory
    • Mem[address] = %reg

2) Perform arithmetic operation on register or memory data
  • c = a + b;  z = x << y;  i = h & g;

3) Control flow: what instruction to execute next
  • Unconditional jumps to/from procedures
  • Conditional branches

Remember: Memory is indexed just like an array of bytes!
Operand types

- **Immediate**: Constant integer data
  - Examples: $0x400$, $-533$
  - Like C literal, but prefixed with `$`
  - Encoded with 1, 2, 4, or 8 bytes depending on the instruction

- **Register**: 1 of 16 integer registers
  - Examples: `%rax`, `%rdx`
  - But `%rsp` reserved for special use
  - Others have special uses for particular instructions

- **Memory**: Consecutive bytes of memory at a computed address
  - Simplest example: `%rax`
  - Various other “address modes”

Summary

- Converting between integral and floating point data types **does** change the bits
  - Floating point rounding is a HUGE issue!
    - Limited mantissa bits cause inaccurate representations
    - Floating point arithmetic is NOT associative or distributive
  - x86-64 is a complex instruction set computing (CISC) architecture
  - Registers are named locations in the CPU for holding and manipulating data
  - x86-64 uses 16 64-bit wide registers
  - Assembly operands include immediates, registers, and data at specified memory locations