How Computers Represent Numbers

John Finigan

November 20, 2015

Typical modern computer hardware

- PC, iPhone, IBM Mainframe
- All information stored in memory as base-2 digits
- Every byte (8 base-2 digits) has a unique address
  - A byte is our fundamental unit of information

Bytes and Words

- 1 byte is 8 bits (8 base-2 digits)
- 1 byte can store $2^8 = 256$ unique values
- So, we group bytes into words
- 4 byte word can store $2^{32} = 4,294,967,296$ unique values
- 2, 4, and 8 byte words are common sizes for numbers

Representing $\mathbb{Z}$ - The Integers

- Consider a word as one base-2 number
- A 2 byte word is then one 16 digit base-2 number
- Range: 0000000000000000_2 to 1111111111111111_2
- In base-10, that's 0_{10} to $65,535_{10}$

First problem: Limited Range

- $\mathbb{Z}$ is infinitely large
- No getting around this mismatch
- Minimum value: 0
- Maximum values:
  - 2 byte word - $65,535_{10}$
  - 4 byte word - $4,294,967,295_{10}$
  - 8 byte word - $18,446,744,073,709,551,615_{10}$

Second problem: No Negative Numbers

- Solution: Reserve half of the range for negative numbers
- Zero is placed in the center of our finite number line
- Largest absolute value is cut in half
- One bit is consumed by the sign, one way or the other

Representing Negative Numbers, Take 1

- Sign/Magnitude representation
  - Most significant bit indicates sign
  - Rest indicate magnitude (absolute value)
  - Example in 4 bits: $0111_2 = 7_{10}$ negates to $1111_2 = -7_{10}$
- Ones Complement
  - Invert every bit in a positive number to make a negative number with same absolute value
  - Example in 4 bits: $0111_2 = 7_{10}$ negates to $1000_2 = -7_{10}$

Problem! Negative Zero

- Not impossible to work around, just awkward
- 4 bit examples:
  - Sign/Magnitude: $\textcolor{red}{0}000_2$ and $\textcolor{red}{1}000_2$
  - Ones Complement: 0000_2 and 1111_2
- To test equality with zero, must test both
  - Either in hardware, or in software

Representing Negative Numbers, Take 2

- Twos Complement
  - To make a negative number, invert every bit in its positive counterpart and then add 1
  - Example in 4 bits: $0111_2 = 7_{10}$
  - Negates to $1000_2 + 1_2 = 1001_2 = -7_{10}$
- No Negative Zero:
  - 0000_2 inverts to 1111_2
  - $1111_2 + 1_2 = 0000_2$, discarding carry
  - Twos complement negative of 0000_2 is 0000_2
- Adding twos complement numbers "works":
  - $7_{10} + -7_{10} = 0$
  - $0111_2 + 1001_2 = 0000_2$, discarding carry

Overflow

- Notice the "discarding carry" from previous slide
- A word can't grow, so carry is lost
- In other words, modular arithmetic
- For a two byte word, $65,535_{10} + 1_{10} = 0$
- C guarantees modular arithmetic for unsigned integers, but guarantees nothing for signed integers: result is undefined.

Unsigned Overflow demo - C language

```c
unsigned int u = UINT_MAX;
printf("UINT_MAX is %u\n", u);
printf("UINT_MAX + 1 is %u\n", u + 1);
printf("UINT_MAX + 2 is %u\n", u + 2);
```

Output:

```
UINT_MAX is 4294967295
UINT_MAX + 1 is 0
UINT_MAX + 2 is 1
```

Signed overflow in C is undefined!

Representing $\mathbb{R}$ - The Reals

- Range still limited by finite word size
- New problem: real number line is infinitely dense
  - Base-2 word is fundamentally an integer
  - Unavoidably, it can represent only a finite set of values
- Not all real values can have unique/exact representations

Floating Point

- Divide the word into two major parts
- One part stores a binary value with a fixed radix point
  - Called the Significand or Fraction or Mantissa
  - Example: 1.01010101_2
- The other part stores a binary value representing an integer exponent
  - An implicit base, β, is raised to this exponent
- Multiplying the significand by the exponentiated base yields the word's real number value
  - Value is computed as $\pm d.dddddddd \times \beta^e$

Exact representation of a real number is possible

- But only for certain reals:
  - If the desired value is not too large or too small
  - If the desired value and the base share the same prime factors
    - Base-10 example: $0.1 = \frac{1}{10} = \frac{1}{2 \cdot 5}$
    - If $\beta = 2$, 5 is not a common prime factor!
- Uniqueness isn't guaranteed either:
  - $1.0 \times 10^1 = 0.1 \times 10^2$
  - But, if we require that the significand ≥ 1.0, uniqueness is guaranteed
  - A floating point number with this property is *normalized.*

Example Encoding

- 4 byte word:
  - 1 bit of sign, 8 bits of exponent, 23 bits of fraction
  - SEEE EEEE EFFF FFFF FFFF FFFF FFFF FFFF
- 8 byte word:
  - 1 bit of sign, 11 bits of exponent, 52 bits of fraction
  - SEEE EEEE EEEE FFFF FFFF FFFF FFFF FFFF
  - FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF

Encoding Choices

- How to represent sign of fraction and sign of exponent
- What base, β, is raised to the exponent?
- How to represent special values, like 0, infinity, or undefined
- Many competing obsolete defacto standards
  - IBM Mainframe HFP, $\beta = 16$
  - DEC VAX
  - Cray
  - dozens more

IEEE 754 Standard (Basic Formats)

- 4 byte word (single precision):
  - 1 bit of sign, 8 bits of exponent, 23 bits of fraction
- 8 byte word (double precision):
  - 1 bit of sign, 11 bits of exponent, 52 bits of fraction
- For each:
  - $\beta = 2$
  - Sign/Magnitude sign representation for fraction
  - *biased* sign representation for exponent
  - Exponent's unsigned value is added to a constant *bias*

IEEE 754 Limits

- Single precision
  - $e_{max} = 127, e_{min} = -126$
  - 23 bits of fraction plus implict most significant bit (1.)
  - Normalized range: about $\pm 1.175 \times 10^{-38} to \pm 3.403 \times 10^{38}$
- Double precision
  - $e_{max} = 1023, e_{min} = -1022$
  - 53 bits of fraction plus implict most significant bit (1.)
  - Normalized range: about $\pm 2.225 \times 10^{-308} to \pm 1.798 \times 10^{308}$

IEEE 754 Special Values (Single Precision)

- If raw exponent is 255 (all 1s) and fraction is not 0
  - NaN - Not a Number, sign irrelevant
- If raw exponent is 255 (all 1s) and fraction is 0
  - Infinity - sign bit determines sign
- If raw exponent is 0 (all 0s) and fraction is not 0
  - Denormalized numbers, sign bit determines sign
- If raw exponent is 0 (all 0s) and fraction is 0
  - Zero - sign bit determines sign

Error

- "Squeezing infinitely many real numbers into a finite number of bits requires an approximate representation." - Goldberg
- Units in the last place (ulps)
  - How much does the rounded result differ from the ideal result?
  - 3.14 approximated to 3.12 - 2 ulps
  - 3.14159 approximated to 3.14 - 0.159 ulps
  - Due to rounding, approximation can differ from ideal by up to 0.5 ulps
- Relative error:
  - (ideal - approx) / ideal
  - Relative error corresponding to 0.5 ulps varies by at most a factor of β (Goldberg)

Overflow and Underflow

- IEEE 754 mandates that overflow and underflow can be detected
  - a flag is set and, optionally, an exception can be raised
- Gradual Underflow vs. Store-0
  - Recall that the fraction's most significant bit is implicitly a 1 before the radix point
  - This leaves a BIG gap of unrepresentable numbers between $2^{e_{min}}$ and zero
  - Old non-IEEE 754 implementations typically store 0 for these numbers

Gradual Underflow

- IEEE 754 introduces a special case: if exponent is $e_{min} - 1$ and fraction is non-zero, implicit digit before radix point is 0.
  - called a subnormal or denormalized number
- We can then use all of the fraction's bits to represent values between $2^{e_{min}}$ and 0
- Calculations do not have to be scaled to avoid tiny values that, with Store-0, would be represented *less accurately* than normalized numbers
- Makes error analysis and avoidance easier
  - Otherwise, algorithms have to detect when values get too small, and if possible, scale to avoid them

Error Example

```
double a = 1.015;
printf("%19.16f\n", a);
```

Output

```
1.0149999999999999
```

1.015_{10} is not exactly representable in base-2 floating point.

Error can break your program

```
//Goldberg's constants
double x = (3.34*3.34) - (4.0*1.22*2.28);
printf("%19.16f\n", x);

if (x == 0.0292)
   printf("equal to 0.0292!\n");
else
   printf("not equal to 0.0292!\n");
```

Output:

```
0.0292000000000012
not equal to 0.0292!
```

Sources of Error beyond 0.5 ulps

- Error accumulates
- Expressions may not be associative or distributive
- Summations can accumulate error
  - Using naive approach, can grow proportional to number of terms summed
  - My own experiment: relative error of summing 1.015
  - Summed 100 times vs 1 trillion times: relative error 29 billion times larger!
- Cancellation: subtracting two numbers that are almost equal can cancel most of the correct digits, leaving mostly incorrect digits.

Solution

It's an entire field of study!

- Many recipes have been developed, and many proofs done
- Active field since at least the 1960s
- Informed IEEE 754 deeply
- Many "simple" equations can be rearranged in potentially complex ways to avoid error.
- Example: Kahan's summation algorithm reduces error growth to a constant factor

Decimal Floating Point

- IEEE 854
- $\beta = 10$
- We (mostly) think in base-10, so it's better for human generated numbers
- Such as money
- Same underlying approximation issues though
- $\beta > 2$ can make error analysis more difficult (Goldberg), but 10 is a special case for human reasons

Software defined precision

- IEEE 754 happens in hardware (typically)
- We can do higher precision more slowly using software
- GNU Multiple Precision Arithmetic Library (GMP)
  - " There is no practical limit to the precision except the ones implied by the available memory in the machine GMP runs on."

References

- David Goldberg. 1991. What every computer scientist should know about floating-point arithmetic. ACM Comput. Surv. 23, 1 (March 1991)
- IEEE STANDARD 754-1985 - IEEE Standard for Binary Floating-Point Arithmetic
- Sun Numerical Computation Guide (2005, Sun Microsystems inc.)