Algorithms and Architecture 1

Representing and Manipulating Numbers

Alexandre David

Outline

- Introduction
- Information storage
- Integer coding
- Basic arithmetics
- Hexadecimal notation
- Endianness
- Floats
- IEEE FP
- Implementation of functions
- Numerical Precision

Introduction

Natural/Real Numbers

- Base 10
- Infinite
- Exact

Numbers in Computers

- Base 2
- Finite representation
- Rounding - overflow

In this lecture

- How to represent numbers – range, encoding
- Arithmetics
- How to use these numbers

Examples

- Overflow:

```
main() {
 printf("%d\n", 200*300*400*500);
}
```

outputs -884901888

```
main() {
 printf("%lld\n",200LL*300LL*400LL*500LL);
}
```

outputs 12 000 000 000

- Loss of precision:

(3.14+1e20)-1e20 == 0.0

3.14+(1e20-1e20) == 3.14

Information Storage

- Basic unit is the byte (=8 bits).

C-declaration	Typical 32-bit	Compaq Alpha
char	1	1
short int	2	2
int	4	4
long int	4	8
char*	4	8
float	4	4
double	8	8

- Note1: beware of addressing.
- Note2: allocated memory is 32/64 bits aligned.

Integer Coding

- Unsigned integers: $UB = \sum_{i=0}^{i=w-1} x_i 2^i$
- Signed integers: $SB = -x_{w-1} 2^{w-1} + \sum_{i=0}^{i=w-2} x_i 2^i$

with w being the size of a word (in bits), x the bits.
Coding for signed integers is called 2 complement.

- The highest bit codes the sign.
- Overflow rounds up

Basic Arithmetics

- Logical operations (bitwise): & | ^ ~ >> <<
- Arithmetics operations: + - * /
- Careful with shifts on signed integers.
- Do not mess up with boolean operations (&& ||).
- Properties:
  - Operations are the same on int/unsigned int.
  - Commutativity, associativity, distributivity, identities, annihilator, cancellation, idempotency, absorption, De Morgan laws.
  - Identity: -a == ~a + 1

Arithmetics Cont.

- Applications:
  - Machine code of + - * / same for int/uint
  - Howto set/unset/read bits? Swap example.
- Integer convertion == type casting
  - Padding for the sign (int)
  - Convertion is modulo the size of the new int.
  - Beware of implicit conversions in C.
- Optimizations for some operations:
  - 2*a == a+a == a << 1, a/2 == a >> 1
  - a *= 2^i == x <<= i, a /= 2^i == x >>= i
  - a % 2^i == a & ((1 << i)-1), 2^i == 1 << i

Hexadecimal Notation

- Get used to know by heart first powers of 2.
- Very useful to manipulate bits.
  - A digit codes 4 bits (0..F, ie, 0..15) = 16 numbers.
  - 0..9, obvious. A..F = 10..15.
  - C notation 0x.. for hexadecimal.
- Examples:
  - 0xa57e: (10=8+2)(5=4+1)(7=4+2+1)(14=8+4+2)
    1010 0101 0111 1110
  - Useful to know 0xf 0x7 0x3.
  - Individual bits accessed by shifts and masks.
  - uint: 0..0xffffffff, int: 0x80000000..0x7fffffff

Endianness: Beware

- When I write “0xa57e”, it is a notation for humans. In base 2, it is “1010 0101 0111 1110”.
- Little endian: stored as 0111111010100101
  Big endian: stored as 1010010101111110
  in memory, at the bit level on the chip.
- It does not matter in C, you never need to pay attention to it except:
  - For bitmap manipulation
  - Device drivers
  - Network transferts
  - When the bit ordering matters where you are writing

Example

```
main() {
 int a = 0xf0000000;
 char *c = &a;
 printf("%x\n", *c);
}
```

- Intel: outputs 0
- Sun: outputs ffffff0

Why?

- What if `a = 0x70000000` ?

Floats

- How to code a real number with bits?
  - Finite precision -> approximation
  - Represent very small and very large numbers -> “density” of encoding varies.
- Scientific notation used, eg (base 10), 3.141e12 but in base 2.
- Fractional numbers (bad for large numbers)
  - Decimal: $d = \sum_{i=-n}^{m} 10^i d_i$
  - Binary: $b = \sum_{i=-n}^{m} 2^i b_i$

IEEE FP

- IEEE floating point standard
  - $V=(-1)^s M 2^E$
  - Number of bits (float/double) for s: 1, m: 23/52, e: 8/11
  - Normalized and denormalized values
  - Bit fields: s, m, e to code respectively S, M, E
- Normalized values (e!=0, e!=111...)
  - E=e-bias (-126..127/-1022..1023)
  - M=1+f $1 \leqslant M < 2$
    - Trick for more precision: implied leading 1 representation.

IEEE FP Cont.

- Denormalized values (E: only 0s or 1s)
  - e=0, $E=1-\text{bias}, \text{bias}=2^{k-1}-1$
    - Coding compensates for M not having an implied leading 1.
    - M=f
    - Coding for numbers very close to 0 and 0 (+0.0,-0.0)
  - e=111..
    - f=0, (signed) infinite
    - f!=0, NaN
- Properties
  - +0.0 == 0
  - If interpreted as unsigned integers, floats can be sorted (+x ascending, -x descending)

Properties

- Operations not associative
- Not always inverse (infinity)

  Important for compilers and programmers.
- Loss of precision.
- Example: x=a+b+c; y=b+c+d;

  Optimize or not?
- Monotonicity $a \geqslant b \Rightarrow a+x \geqslant b+x$
- Cast:
  - int2float rounded, double2float rounded/overflow
  - int/float2double OK
  - float/double2int truncated/rounded/overflow

IA32: The Good and The Bad

- Good: uses internally 80 bit extended registers for more precision.
- Bad:
  - Stack based FP
  - Side effects like changing values when loading or saving numbers in memory whereas register transferts are OK. Memory accesses may imply rounding (to float or double).

Implementation of Functions

- Integers
  - Addition/substraction simple
  - Multiplication based on
    `r=a*b: r=0; while(b) do { if (b&1) r+=a; a+=a; b>>=1; }`
  - Division iterative like pen and paper
- Floats
  - Addition/substraction require
    - Check for 0, align the significands, +/-, normalize the result
    - Guard bits (ALU reg larger, padd with 0) to avoid losing precision on numbers that are very close (1.0000..*2^1-1.1111..2*0)
  - Multiplication/division principle simpler
    - multiply/divide significands, add/sub exponents, detect over/under-flow

Complex Functions

- Lagrange polynomials $P_n(x)=\sum_{i=0}^{n}[f(x_i)\prod_{k=0,k\neq i}^{n}\frac{(x-x_k)}{(x_i-x_k)}]$
- Taylor series $f(x)=\sum_{i=0}^{\infty}f^{(i)}(x_0)\frac{(x-x_0)^i}{i!}$
- Other numerical methods that *converge rapidly*
- Special (int): random generator (linear congruence generator). $x_{i+1}=(ax_i+c)\bmod m$
  - Simple
  - But correlation between successive values
  - High bits of better quality

Numerical Precision

- Evalutation of precision
  - Absolute: $x\pm\alpha$
  - Relative: $x*(1\pm\delta)$
- Be careful with division by very small values: can amplify numerical errors
  - Numerical justification for Gauss' method to solve equations.