S2 - Integer and Floating Point Numbers

Date: 2013-05-01
tags: hwswint

Unsigned Integers

Possible number of values for N bits = 2^N - 1

Addition/Subtraction can be done with the normal "carry/borrow" rules.

Signed Integers

For unsigned integers example:

8 bits: `0x00`=0, `0x01`=1, ..., `0x7F`=127

But we need half of them to be negative.

The high order bit is used to indicate negative. This is called "sign-and-magnitude" representation.

Example:

- `0x00` = 00000000_2 is non-negative, because the sign bit is 0
- `0x7F` = 01111111_2 is non-negative
- `0x85` = 10000101_2 is negative
- `0x80` = 10000000_2 is negative

But there are two values for 0_{10} : `0x00` (+0) and `0x80` (-0).

This can create problems for example:

4 - 3 = 0100_2 - 0011_2 = 0001_2 = 1

4 + (-3) = 0100_2 + 1011_2 = 1111_2 = -7

Therefore, 4 - 3 != 4 + (-3)

We do not want this! Instead what is used is two's complement negatives.

Two's Complement Negatives

Rather than having a sign bit as in Signed integers, let the most significant bit have the same value but negative weight.

Example:

- unsigned 1010_2 : $1*2^3 + 0*2^2 + 1*2^1 + 0*2^0 = 10_{10}$
- 2's comp 1010_2 : $-1*2^3 + 0*2^2 + 1*2^1 + 0*2^0 = -6_{10}$

-1 is represented as 1111_2, all negative integers still have MSB = 1. Also, there is only one zero.

To get the negative representation of any integer, take bitwise compliment and then add one!

`~x + 1 = -x`

Arithmetic example:

- 4 + 3 = `0100 + 0011 = 0111` = 7

- -4 + 3 = 1100 + 0011 = 1111 = -1
- 4 - 3 = 0100 + 1101 = 10001 = 0001 = 1

Here in the last example, the highest carry bit is dropped. This is called modular addition.

Signed and Unsigned Numeric Values

Both signed and unsigned integers have limits. For 4 bit values:

- If you compute a number that is too big like 6 + 4 = ? or 15U + 2U = ?
- If you compute a numbet that is too small like -7 -3 = ? or 0U - 2U = ?

The CPU may be capable of "throwing an exception" for overflow on signed values but it won't for unsigned. C & Java don't give a fuck when this happens and silently cruise along. We need to be careful about this and explicitly check.

Values to remember

Unsigned Values:

- UMin = 0 (000...0)
- UMax = 2^w - 1 (111...1)

Two's Complement Values:

- TMin = -2^{w-1} (100...0)
- TMax = 2^{w-1} - 1 (011..1)
- Negative 1 = 111...1 or 0xFFFFFFFF (32-bits)

Values for W = 16 (16-bit systems)

	Decimal	Hex	Binary
UMax	65535	FF FF	11111111 11111111
TMax	32767	7F FF	01111111 11111111
TMin	-32768	80 00	10000000 00000000
-1	-1	FF FF	11111111 11111111
0	0	00 00	00000000 00000000

Some observations:

- | TMin | = TMax + 1
- UMax = 2 * TMax + 1

Integers in C

C has support for unsigned and signed numbers with #include<limits.h>. In this header file, there are some contants pre-declared for us such as:

- ULONG_MAX
- LONG_MAX
- LONG_MIN

These values are platform specific.

Constants

- By default, constants are considered signed integers
- Use "U" suffix to force unsigned, e.g. `0U`, `4294967259U`

Casting

- `int tx, ty;` - signed integer
- `unsigned ux, uy;` - unsigned integer
- `tx = (int) ux;` - unsigned casted as signed integer
- `uy = (unsigned) ty;` - signed casted as unsigned integer
- `tx = ux` - implicit casting also occurs via assignments and function calls
- `uy = ty` - same as above
- The `gcc` flag `-Wsign-convention` produces warning for implicit casts, but `-Wall` doesn't.

Here, the bits are unchanged, they are just interpreted differently.

Some casting surprises:

- If signed and unsigned are mixed in a single expression, then signed values are implicitly casted to unsigned.
- Including comparision operators <, >, ==, <=, >=

Shift operations for Unsigned Integers

Left Shift: x<<y

- Shift bit-vector x left by y positions.
- Throw away extra bits on the left and fill with `0s` on the right.

Right Shift: x>>y

- Shift bit-vector x right by y positions.
- Throw away extra bits on the right and fill with `0s` on the left.

x	00000110 (6)
<< 3	00110000 (48)
>> 2	00000001 (1)

Here, when we left-shifted binary of 6 by 3 positions, we multiplied 6 by 2^3 which yielded 48.

When we right-shifted binary of 6 by 2 positions, we divided it by 2^2 which should have yielded 1.5 but since we can't represent fraction, so the result was 1.

x	11110010 (242)
<< 3	10010000 (144 - wrong)
>> 2	00111100 (60)

Here, when binary of 242 was left-shifted by 3 positions, the result should have been 242 * 2^3 = 1936 but instead we got 144 since 1936 doesn't fits in 8 bit and additional bits on the left were dropped.

When binary of 242 was right-shifted by 2 positions, we got the correct result of 242 / 2^2 = 60.5 (rounded down to 60).

Shift operations for signed integers

Left Shift: x<<y

- Equivalent to multiplying x by 2^y
- (if resulting value fits, no 1s are lost)

Right Shift: x>>y

- Logical Shift (for unsigned values) - fill with 0s on the left
- Arithmetic Shift (for signed values) - replicate MSB on left and maintains sign of x
- Equivalent to dividing x by 2^y

x	`01100010` (98)
<< 3	`00010000` (16 - wrong)
Logical >> 2	`00011000` (24)
Arithmetic >> 2	`00011000` (24)

Now with a negative integer.

x	`10100010` (-94)
<< 3	`00010000` (16 - wrong)
Logical >> 2	`00101000` (40 - wrong)
Arithmetic >> 2	`11101000` (-24)

Here, in the last arithmetic right shift, the MSB 1 instead of 0 was added to the left.

Undefined behavior occurs in C when `y < 0` or `y >= word_size`.

Using Shifts and Masks

Extract the 2nd most significant byte of an integer.

- Let x be `00110101 01100010 10011010 01010010`.
- First we do right shift operation `x >> 16` and get `00000000 00000000 00110101 01100010`.
- And then we do bitwise `&` operation with `0xFF` which in binary is `00000000 00000000 00000000 11111111`.
- The result of the above `&` operation is `00000000 00000000 00000000 01100010`.

Extract the sign bit of signed integer.

- `(x>>31) & 1` - need the `& 1` to clear out all other bits except LSB.

Conditionals as Boolean expressions (assuming x is 0 or 1)

- `if (x) a=y else a=z;` which is same as `a = x ? y : z;`
- **Can be re-written (assuming arithmetic right shift) as:**

```
a = ((x<<31) >> 31) & y + (((!x)<<31) >> 31) & z;
```

Sign Extension Example

Here, we will convert from smaller to larger integer data type. C automatically performs sign extension.

```
short int x = 12345; // 16 bits
int ix = (int) x; // 32 bits
short int y = -12345;
int iy = (int) y;
```

	Decimal	Hex	Binary
x	12345	30 39	00110000 01101101
ix	12345	00 00 30 39	00000000 00000000 00110000 01101101
y	-12345	CF C7	11001111 11000111
iy	-12345	FF FF CF C7	11111111 11111111 11001111 11000111

Fractional Binary Numbers

Some examples:

- 1011.101_2 is equal to 8 + 2 + 1 + 1/2 + $1/8_{10}$.
- 5 and 3/4 = 101.11_2
- 2 and 7/8 = 10.111_2
- 63/64 = 0.111111_2

Observations:

- Divide by 2 by shifting right
- Multiply by 2 by shifting left
- **Numbers of form $0.111111111...._2$ are just below 1.0**
  1/2 + 1/4 + 1/8 + ... + $1/2^i$ + ... ---> 1.0 Shorthand notation for all 1 bits to the right of binary point: 1.0 - e (epsilon)

Limitations:

- Can only exactly represent numbers that can be written as x * 2^y
- **Other rational numbers have repeating bit representations, e.g.:**
  - **1/3 = $0.0101010101[01]_{...2}$**
    - 1/5 = $0.001100110011[0011]_{...2}$
    - 1/10 = $0.0001100110011[0011]_{...2}$

Fixed Point Representations

We might try representing fractional binary numbers by picking a fixed place for an implied binary point.

Lets do that, using 8-bit fixed point numbers as an example:

- **#1 the binary point is between bits 2 and 3**
  b_7 b_6 b_5 b_4 b_3 [.] b_2 b_1 b_0 - The maximum value that can be represented with this is 31 and 7/8.
- **#2: the binary point is between bits 4 and 5**
  b_7 b_6 b_5 [.] b_4 b_3 b_2 b_1 b_0 - The maximum value that can be represented with this is 7 and 31/32.

The position of the binary point affects the range and precision of the representation.

- range: diff between largest and smallest numbers possible
- precision: smallest possible difference between any two numbers

One of the pros of fixed point representations is that it's simple. The same hardware that does integer arithmetic can do fixed point arithmetic.

But there is a bigger con to it. There is no good way to pick where the fixed point should be. Either the precision or the range has to be sacrificed each time. This is why fixed point representations are not used.

IEEE Floating Point

It is analogous to scientific notation. For example, we represent 12000000 as 1.2×10^7 ; and 0.0000012 as 1.2×10^{-6} . In C, these can be written as `1.2e7` and `1.2e-7` respectively.

This is an IEEE standard established in 1985 for floating point arithmetic and is supported by all major CPUs today. This is fast at hardware level but is numerically well behaved.

Floating Point Representation

Numerical form:

$V_{10} = (-1)^s * M * 2^E$

- Sign bit s determines whether the number is negative or positive
- Significand (mantissa) M normally a fraction value in range [1.0, 2.0)
- Exponent E weights value by a (possibly negative) power of two

Representation in memory

- MSB s is sign bit s
- exp field encodes E (but is not equal to E)
- frac field encodes M (but is not equal to M)

s	exp	frac

Single precision: 32 bits

s (1)	exp (8)	frac (23 bits)

Double precision: 64 bits

s (1)	exp (11)	frac (52 bits)

Normalization and Special Values

Normalized means the mantissa M has the form 1.xxxxxx

- 0.0011×2^5 and 1.1×2^3 represent the same number, but the latter makes better use of the available bits.
- Since we know that mantissa starts with a 1, we don't bother to store it thus saving a bit.

Now, how do we represent `0.0`? Or special/undefined values like `1.0/0.0`?

For these cases, there are some special values reserved for `exp` and `frac`.

- A bit pattern `00...0` represents zero

- **If `exp == 11...1` and `frac == 00...0`, it represents infinity**
  e.g. 1.0/0.0 = -1.0/-0.0 = +infinity, 1.0/-0.0 = -1.0/0.0 = -infinity
- **If `exp == 11...1` and `frac != 00...0`, it represents `NaN`: Not a Number.**
  Results from operations with undefined results, e.g. sqrt(-1), infinity, -infinity, infinity * 0

Since, `000...0` and `111...1` are already reserved above as special values, we can't use them.

So, the condition: `exp != 000...0` and `exp != 111...1`.

Exponent is coded as biased value: `E = exp - Bias`

- exp is an unsigned value ranging from 1 to 2^k - 2 (k == # bits in the exp)
- **Bias = 2^{k-1} - 1**
  - Single precision: 127 (so exp: 1...254, E: -126...127)
  - Double precision: 1023 (so exp: 1...2046, E: -1022...1023)
- This enables negative values for E, for representing very small values.

Significand coded with implied leading 1: `M = 1.xxx...x`$_2$

- `xxx...x`: the n bits of frac
- Minimum when 000...0 (M = 1.0)
- Maximum when 111...1 (M = 2.0 - e(epsilon))
- Get extra leading bit for "free"

Normalized Encoding Example

Value: `float f = 12345.0;` (with single precision, 32-bits)

$12345_2 = 11000000111001_2 = 1.1000000111001_2 \times 2^{13}$

Singnificand:

$M = 1.1000000111001_2$ frac = $10000001110010000000000_2$

Exponent: E = exp - Bias, so exp = E + Bias

E = 13, Bias = 127, so exp = 140 = 10001100_2

Result

s (1)	exp (8)	frac (23 bits)
0	10001100	10000001110010000000000

Floating Point Operations

Unlike the representation for integers, the representation for floating-point number is not exact because mantissa doesn't go on forever and stops at 23 bits or even 52 bits in double precision.

Basic idea:

- First, compute the exact result
- **Then, round the result to make it fit into desired precision:**

- Possibly overflow if the exponent is too large
- Possibly drop least significant bits of significand to fit into `frac`

For rounding a value, there are many possible rounding modes like round towards zero, round down, round up, round to nearest value and round to even. With most of these values, errors start accumulating when rounding is repeated on the same value and this causes statistical bias. However, with round to even avoids this bias by rounding up about half the time and rounding down half the time. The default rounding mode for IEEE floating point is Round-to-even.

If overflow of the exponent occurs, result will be infinity or -infinity. Floats with value of inifinity, -inifinity and NaN can be used in operations but the result is usually still inifinity, -inifinity, or NaN.

Floating point operations are not always associative or distributive, due to rounding!

- (3.14 + 1e10) - 1e10 != 3.14 + (1e10 - 1e10)
- 1e20 * (1e20 - 1e20) != (1e20 * 1e20) - (1e20 * 1e20)

Here, in the first example, in the LSH, when `(3.14 + 1e10)` is computed, `3.14` so small against `1e10` that after adding these two and trying to fit it into the 23 bits, the least significant bits are dropped and thus the result is `1e10`. So, LSH computes to `1e10 - 1e10 = 0`. While on the RHS, `3.14 + (1e10 - 1e10) = 3.14`.

In the second example, LSH simply computes to `0`. While on the RHS, `(1e20 * 1e20)` causes overflow and results infinity. So, RHS computes to `infinity - infinity = infinity`.

Floating Point in C

C offers two levels of precision

- `float` - single precision (32-bit)
- `double` - double precision (64-bit)

Default rounding mode is round-to-even. There is a header file `#include <math.h>` to get INFINITY and NAN constants.

Equality comparisions ``==`` between floating point numbers are tricky, and ofter return unexpected results. AVOID THEM!!! Rather, subtract them and test if the value is small.

Coversion between data types

Casting between int, float, and double changes the bit representation.

- `int --> float` - May be rounded but overflow not possible since float can accomodate much larger values than int.
- `int --> double` - Exact conversion as long as int has <= 53-bit word size since double has 52 bits for mantissa part (the leading 1 is not stored to the int can be 52+1 bit long) else rounding occurs.
- `float --> double` - Exact conversion since float is 32-bits and 64-bits, so double can definitely store a float representation in it.
- `double or float --> int` - Fractional part is truncated (rounded towards zero). And if the double/float representation is too large or too small, then int is generally set as Tmin or NaN or infinity...

Summary

Zero

0	00000000	00000000000000000000000

Normalized values

s	1 to 2^k -2	significand = 1.M

Infinity

s	11111111	00000000000000000000000

NaN

s	11111111	non-zero

Denormalized values

s	00000000	significand = 0.M

As with integers, float suffers from fixed number of bits available to represent them.

- Can get overflow/underflow, just like ints
- Some "simple fractions" have no exact representations (e.g. 0.2)
- Can also lose precision, unlike ints. Every operation gets a slightly wrong result.

Mathematically equivalent ways of writing an expression may compute differet results.

Never test floating point values for equality.