COMP6700/2140 Bits and Bytes

Alexei B Khorev and Josh Milthorpe

Research School of Computer Science, ANU

February 2017

Bits and Arithmetic (on the Machine)

1. Floating Point numbers precision effects and rounding
2. Bitwise operations on numbers

Floating-Point Numbers

FP numbers are meant to represent *real* (non-integer) numbers of Mathematics using only a finite amount of memory. A finite amount of CPU also precludes operations on FP numbers to be mathematically precise. Given the IEEE Standards for `float` and `double` FP types (7 and 16 decimal digits for precision, respectively), and the fact that on the machine level numbers are stored as bits, we are bound to have only finite precision even for rational numbers (which require a finite amount of decimal symbols for their description):

1. Irrationals, *eg* $\sqrt{2} = 1.41421356237\ldots$ – no question: it cannot be represented exactly in decimal (and binary, and any other) form
2. "Lucky" rationals, *eg* $1\frac{5}{8} = 1.625 = 1 \cdot 2^0 + 1 \cdot \frac{1}{2^1} + 0 \cdot \frac{1}{2^{-2}} + 1 \cdot \frac{1}{2^{-3}} = (1.101)_2$ (exact binary representation)
3. "Unlucky" (majority) of rationals, *eg* $4.35 = 4\frac{7}{20} = (100.01011001100110011001\ldots)_2$ Truncated as required by the standard and converted back to decimal, 4.35 "unexpectedly" becomes 4.3499999999999964

Note: if curious, write a short Java program which calculates a binary (or, in arbitrary base `b`) string representation of a given positive rational number (my solution is ConvertRational.java)

Correct Rounding

Keeping the finite precision effects in mind, lets attempt a simple arithmetic computation:

```
double f = 4.35; int n = (int) ( 100 * f );
System.out.println(n); // prints 434: Wrong!
```

And that's understandable: since 4.35, it is stored as `4.349999999...` so `100 * f = 434.99999...` (etc) and when cast to an integer the fractional component is dropped (not rounded!) thus `n = 434`. The proper way to do rounding is to use a function call from the *Math* class:

```
double f = 4.35; int n = (int)Math.round( 100 * f );
System.out.println( n ); // prints 435! (Good)
```

Mechanised Arithmetic on Bits

Blaise Pascal invented a mechanical calculator in 17th century.

Bit-wise ops

Integer types (`byte, short, int, long`), `boolean` and `chars` values can be operated on by bitwise manipulations:

- `~` toggles each bit in its operand: `~0x00003333` → `0xffffcccc`
- `&` bitwise AND: `0xf00f & 0x0ff0` → `0x0000`
- `|` bitwise inclusive OR: `0xf00f | 0x0ff0` → `0xffff`
- `^` bitwise exclusive or (XOR): `0xaaaa ^ 0xffff` → `0x5555`

Together with three *shift* binary operators (only on integer types, type of the shift expression is determined by the first operand type); the LHS (first operand) is what is shifted, the RHS is how much to shift:

- `<<` signed left shift operator — shifts bits left filling with `0` bits on the RHS: `14 << 2` → `56`
- `>>` signed right (**arithmetic**) shift operator — shifts bits right filling vacated bits with the highest (sign) bit value: `14 >> 2` → `3` **and** `-14 >> 2` → `-4` (the leftmost position after `>>` depends on sign extension)
- `>>>` unsigned right (**logical**) shift operator — shifts bits right, filling vacated bits with zero: `14 >>> 2` → `3` **and** `-14 >> 2` → `1073741820`

they can be used as faster alternatives to standard arithmetic ops (and used in some numerically intensive codes, like image manipulations).

The Machine Level

The bitwise and shift ops is how the processor performs the basic operations:

- Addition: the ripple-carry adder (or alternatives) where sum s of every pair of corresponding bits in the $a + b$ expression (plus carry-in, c_{in}) is using `&`, `|` and `~`:

  $$s = a\bar{b}\bar{c}_{in} + \bar{a}\bar{b}c_{in} + \bar{a}b\bar{c}_{in} + abc_{in}$$
  $$c_{out} = ab + ac + bc$$

  where $\bar{a}$ means `~a`, ab is `a&b` and $a + b$ is `a|b`
- Subtraction: involves `~` and addition
- Multiplication: involves right shift and addition
- Divisions (both types): involve left shift and addition
- Shifts are implemented as bit copy and write

Example with bit and hex

One number example for binary and hexadecimal representations, and the bit-wise negation op:

```
int y = 2716; // ~y is -2717, BTW
String.format("%08x", y)  // hex string for  y is "00000a9c"
String.format("%08x", ~y) // hex string for ~y is "fffff563"
                          // 16*16*10 + 16*9 * 12 <--> a9c (test yourself!)
String.format("%32s", Integer.toBinaryString(y)).replace(' ','0')
      // "00000000000000000000101010011100"
String.format("%32s", Integer.toBinaryString(~y))
      // "11111111111111111111010101100011"
y << 4 // 43456, left shift by n amounts to multiplying y by 2^n (mod range)
String.format("%32s", Integer.toBinaryString(y<<4)).replace(' ','0')
      // "00000000000000001010100111000000"
y >> 4 // 169, right shift by amounts to integer division by 2^n
String.format("%32s", Integer.toBinaryString(y>>4)).replace(' ','0')
      // "00000000000000000000000010101001"
```

Where to look for this topic in the resources?

- Java Tutorial: Bitwise and Bit Shift Operators

- Arnold, Gosling and Holmes *The Java Programming Language, 4ed*, Ch. 9.25

- (for brave of heart) Henry S. Warren Jr. *Hacker's Delight*, Addison Wesley, 2013