Programming Design and Implementation

Lecture 9: Number Systems

Updated: 19th May, 2020

Mark Upston
Discipline of Computing
School of Electrical Engineering, Computing and Mathematical Sciences (EECMS)

Outline

Data Representation in Computer Memory

- Computer memory is made up of components that can be in one of two states (on or off)
- Other binary methods of information include:
  - On and Off (Flashing light)
  - Dot and Dash (Morse)
  - North and South (Magnet)
- Can be used in two ways:
  - Represented actual values in base 2
  - Hold an arbitrary series of states coded with particular meanings

Terminology

- Each <u>**b**</u>inary dig<u>**it**</u> is called a <u>**bit**</u>
- A group of eight (8) bits is a <u>**byte**</u>
- Memory is broken up into storage locations of a particular <u>**wordsize**</u>
- Wordsize is machine dependent and will be one or more bytes long
  - Now 64 bits (8 bytes)
- Each memory location is located through its <u>**memory address**</u>
- All data and programs are stored in memory using various interpretations of these groups of 1's and 0's

Data Types

- Manner of interpretation of the 1's and 0's varies for different data types stored:
- The way in which the 1's and 0's are interpreted depends on the data type being represented.
    - Addresses
    - Instructions
    - Integer values
    - Real values
    - Characters
        - Single characters
        - Character Strings
    - Boolean

Java's Primitive Data Types

- Java defines 8 primitive types:

Java Type	Memory Format	Range/Domain	Range/Domain
byte	8 bit integer	-2^7 to 2^7-1	-128 to 127
short	16 bit integer	-2^{15} to 2^{15}-1	-32768 to 32767
int	32 bit integer	-2^{31} to 2^{31}-1	-2147483648 to 2147483647
long	64 bit integer	-2^{63} to 2^{63}-1	±9.22337E+18
float	32 bit floating point	±6 sig. digits (10^{-46},10^{38})	
double	64 bit floating point	±15 sig. digits (10^{-324},10^{308})	
char	16 bit character	All Characters	
boolean	boolean	**true**, **false**	

Integer Data Types

- Integer: positive or negative value that consists of a whole number
- The Java primitive types **`byte`**, **`short`**, **`int`** and **`long`** are abstractions of integers from the mathematical world
- The range of integers is determined by the amount of storage available (memory) for a particular data type
- The accuracy is guaranteed
  - Stored as the exact base2 (`Binary`) equivalent of the base10 (`Decimal`) integer

Range of Integers

- Determined by how many distinct base2 values can be stored in the given number of bits: every additional bit doubles the size of the range
- For N bits, you always need 1 bit for the sign and the remaining N-1 bits can represent 2^{N-1} different combinations that directly relate to their binary value
- Note that the lack of symmetry is because of the need to represent zero (0) as one of the 2^{N-1} values:
  - {2^{N-1} negative, 0, 2^{N-1}-1 positive} values
    - Negative values stored as the 2's compliment of the number
- When an attempt to store a number which is larger/smaller than the maximum/minimum value then Integer Overflow occurs

Number Systems

- The number system we are used to is base-10 (`Decimal`)
  - Why:
  - $2546 = 2 * 10^3 + 5 * 10^2 + 4 * 10^1 + 6 * 10^0$
- A number system can have any base:
  - Base 8:
  - $2546 = 2 * 8^3 + 5 * 8^2 + 4 * 8^1 + 6 * 8^0$
- Computers use base-2 (`Binary`)
  - Often represented by base-8 (`Octal`) or base-16 (`Hexadecimal`)
    - Because they fall along the base-2 scale and allow for easy conversion

Decimal to Binary

- Divide the number by 2 - if there is a remainder, insert a 1, otherwise insert a 0 (starting from the right)

```
2546 ÷ 2 = 1273     | Remainder: 0        // 0
1273 ÷ 2 = 636      | Remainder: 1        // 10
 636 ÷ 2 = 318      | Remainder: 0        // 010
 318 ÷ 2 = 159      | Remainder: 0        // 0010
 159 ÷ 2 = 79       | Remainder: 1        // 10010
  79 ÷ 2 = 39       | Remainder: 1        // 110010
  39 ÷ 2 = 19       | Remainder: 1        // 1110010
  19 ÷ 2 = 9        | Remainder: 1        // 11110010
   9 ÷ 2 = 4        | Remainder: 1        // 111110010
   4 ÷ 2 = 2        | Remainder: 0        // 0111110010
   2 ÷ 2 = 1        | Remainder: 0        // 00111110010
   1 ÷ 2 = 0        | Remainder: 1        // 100111110010
```

Sanity Check

- **100111110010**
  - $0 * 2^0 + 1 * 2^1 + 0 * 2^2 + 0 * 2^3 + 1 * 2^4 + 1 * 2^5 + 1 * 2^6 + 1 * 2^7 + 1 * 2^8 + 0 * 2^9 + 0 * 2^{10} + 1 * 2^{11}$

Octal

- Not used much any more, but you need to know for Unix file permission setting
  - digits used are 0 to 7
- To convert binary to octal, group the binary number in sets of 3, from right to left, then convert each group to an octal digit (pad left with zeros if necessary)
  - e.g., Decimal 2546_{10} from previous

```
100 111 110 010
 4   7   6   2
```

  - e.g., 342391_{10}

```
001 010 011 100 101 110 111
 1   2   3   4   5   6   7
```

1234567_8

Octal	Binary
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Hexadecimal

- Almost always used to display memory in computers
- Digits used:

Hex	Binary	Hex	Binary
0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

- To convert binary to hex, group the binary number in sets of 4, from right to left, then convert each group to an hex digit (pad left with zeros if necessary)

Binary to Hex Conversion

- e.g., 229656673502_{10}

0011	0101	0111	1000	1001	1010	1011	1100	1101	1110
3	**5**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**

Positive and negative in memory

- The most significant bit determines the sign.
  - 0 for positive and 1 for negative
- The weight of each remaining bit is a power of two.
- For negative the weight is the negative of the corresponding power of two.
  - Calculated and stored using two's complement
- Must always know how many bits we are storing the number in.
  - (8, 16, 32 or 64)

2's Compliment

- To get the two's complement, the bits are inverted by using the bitwise NOT operation; the value of 1 is then added to the resulting value.
- e.g., Decimal 85_{10}

```
8 bit
Pos Binary:     01010101
 Flip Bits:     10101010
                      +1
Neg Binary:     10101011

16 bit
Pos Binary:     0000000001010101
Neg Binary:     1111111110101011
```

2's Compliment (2)

- e.g., Decimal 80_{10}

```
8 bit
Pos Binary:     01010000
 Flip Bits:     10101111
                      +1
Neg Binary:     10110000
```

Real Numbers

- Positive or negative value that consists of a whole number plus a fractional part (expressed in floating point, or scientific notation)
- The Java types **`float`** and **`double`** are an abstraction of the real numbers that exist in the mathematical world
- The range and accuracy of real numbers are limited in any computing system
  - Why? How would you store $\frac{1}{3}$ or $\sqrt{2}$?

Range and Accuracy of Real Numbers

- Determined by number of bits and the split up of the **mantissa** and **exponent**
- There has to be a limit on the range, by definition, you need an infinite number of bits to represent infinity (∞)
- Accuracy is obviously limited
  - The number of significant digits is limited
    - There are an infinite number of real values between any two points on the number line
  - Irrational numbers
  - Recurring decimals
  - IEEE 754 form (binary conversion)

IEEE 754 (Floating Point) Numbers

- Comprises of sign, exponent and mantissa
  - Mantissa A.K.A Significand
- Single precision - binary32
  - Sign bit: Most Significant bit; 0 pos, 1 neg
  - Exponent width: 8 bits biased to 127
  - Significand precision: 24 bits (23 explicitly stored)
    - Normalised
    - $0.1 = (1/2) = 2^{-1}, 0.01 = (1/4) = 2^{-2}, 0.001 = (1/8) = 2^{-3}$ etc

Real Conversion

- To convert a base 10 real number into an IEEE 754 binary32 format use the following outline:
  - In general, refer to the IEEE 754 standard itself for the strict conversion including the rounding behaviour
- Two real numbers with integer and a fraction parts 12.375 and 0.085
  - Determine the sign bit
  - Both 0 (positive)
- Convert the integer part into binary
  - $12 = 1100$
  - $0 = 0$
- Convert the fraction part into binary
  - $.375 = .011$
  - $0.085 = .00010101110000101000111101\ 1$ etc
  - refer to next 2 slides...

Fraction Conversion

- Multiply the fraction by 2
- If value >= 1, write a 1 to the significand , then subtract 1 from the value, otherwise write 0 to the significand.
- Repeat the preceding steps for the appropriate number significant bits.
  - Stop if the value is 0 (after subtracting 1) or when the required number if significant bits has been reached.

Fraction Conversion (2)

- **0.085**

0.17	0
0.34	0
0.68	0
1.36	1
0.72	0
1.44	1
0.88	0
1.76	1
1.52	1
1.04	1
0.08	0
0.16	0
0.32	0
0.64	0
1.28	1
0.56	0
1.12	1

Conversion

- Add the two results
  - 1100.011
  - 0.000101011100001010001111011
- Normalise them
  - $1.100011 * 2^3$
  - $1.01011100001010001111011 * 2^{-4}$
- Produce the exponent (add to bias)
  - $127 + 3 = 130 = 10000010$
  - $127 - 4 = 123 = 01111011$
- Produce significand
  - 100011
  - 01011100001010001111011
  - Important: Integer part (always 1) is not stored.
- Encoding is:
  - 0 10000010 10001100000000000000000 `// Exact`
  - 0 01111011 01011100001010001111011 `// Not exact`

0.1 Base 10

- Determine the sign bit | 0 (Positive)
- convert the integer part into binary | $0 = 0$
- convert the fraction part into binary
  - $0.1 = 0.0001100110011001100110011001100110011001100$
- Add the two results
  - $0.0001100110011001100110011001100110011001100$
- Normalise them
  - $1.100110011001100110011001100110011001100 * 2^{-4}$
- Produce the exponent (add to bias)
  - $127 - 4 = 123 = 01111011$

0.1 Base 10 (2)

- Produce significand
  - 10011001100110011001101
    - 1100110011001100
      `//we lost this (and more) as only allowed 23 bits`
  - Important: Integer part (always 1) is not stored.
- Encoding is:
  - 0 01111011 10011001100110011001101 `// Not exact`
  - 0.100000001490116119384765625 base-10

Next Week

- The next Lecture will address the following:
  - Real World Applications