Introduction to Computing Systems
from bits & gates to C & beyond

Chapter 2

Bits, Data Types & Operations

- *Integer Representation*
- *Floating-point Representation*
- *Logic Operations*

Data types

- **Our first requirement is to find a way to represent information (data) in a form that is mutually comprehensible by human and machine.**
  - **Ultimately, we will have to develop schemes for representing all conceivable types of information - language, images, actions, etc.**
  - **We will start by examining different ways of representing *integers*, and look for a form that suits the computer.**
  - **Specifically, the devices that make up a computer are switches that can be on or off, i.e. at high or low voltage. Thus they naturally provide us with two <u>symbols</u> to work with: we can call them *on* & *off*, or (more usefully) *0* and *1*.**

Decimal Numbers

- "decimal" means that we have ten digits to use in our representation (the symbols 0 through 9)

- What is 3,546?
  - it is *three* thousands plus *five* hundreds plus *four* tens plus *six* ones.
  - i.e. $3,546 = 3.10^3 + 5.10^2 + 4.10^1 + 6.10^0$

- How about negative numbers?
  - we use two more symbols to distinguish positive and negative:

    + and -

Slides prepared by Walid A. Najjar & Brian J. Linard, University of California, Riverside

Unsigned Binary Integers

$$Y = \text{“abc”} = a.2^2 + b.2^1 + c.2^0$$

(where the digits a, b, c can each take on the values of 0 or 1 only)

N = number of bits

Range is:
$0 \leq i < 2^N - 1$

	3-bits	5-bits	8-bits
0	000	00000	00000000
1	001	00001	00000001
2	010	00010	00000010
3	011	00011	00000011
4	100	00100	00000100

Problem:

- How do we represent *negative* numbers?

Slides prepared by Walid A. Najjar & Brian J. Linard, University of California, Riverside

1^{st} attempt: Signed Magnitude

- Leading bit is the <u>sign</u> bit

$Y = \text{“abc”} = (-1)^a (b.2^1 + c.2^0)$

Range is:
$-2^{N-1} + 1 < i < 2^{N-1} - 1$

Problems:

- How do we do addition/subtraction?
- We have two numbers for zero (+/-)!

-4	10100
-3	10011
-2	10010
-1	10001
-0	10000
+0	00000
+1	00001
+2	00010
+3	00011
+4	00100

2^{nd} attempt: One's Complement

- **Invert all bits**

**If msb (most significant bit) is 1 then the
number is negative (same as signed magnitude)**

Range is:
$$-2^{N-1} + 1 < i < 2^{N-1} - 1$$

Problems:

- **Same as for signed magnitude**

-4	11011
-3	11100
-2	11101
-1	11110
-0	11111
+0	00000
+1	00001
+2	00010
+3	00011
+4	00100

Finally: Two's Complement

- **Transformation**
  - **To transform a into –a (and vice versa), invert all bits in a and add 1 to the result**

Range is:
$-2^{N-1} < i < 2^{N-1} - 1$

<u>Advantages:</u>

- **Operations need not check the sign**
- **Only one representation for zero**
  (try it: take the two's complement of 00000)
- **Efficient use of all the bits**

-16	10000
…	…
-3	11101
-2	11110
-1	11111
0	00000
+1	00001
+2	00010
+3	00011
…	…
+15	01111

Manipulating Binary numbers - part 1

- **Binary to Decimal conversion & vice-versa**
  - A 4 bit binary number $A = a_3a_2a_1a_0$ corresponds to:

    $a_3.2^3 + a_2.2^2 + a_1.2^1 + a_0.2^0 = a_3.8 + a_2.4 + a_1.2 + a_0.1$

    (where $a_i = 0$ or 1 only)

  - A decimal number can be broken down by iterative division by 2, assigning bits to the columns that result in an odd number:

    e.g. $(13)_{10} \Rightarrow ((((13 - 1)/2 - 0)/2 - 1)/2 - 1) = 0 \Rightarrow (01101)_2$

  - In the 2's complement representation, leading zeros <u>do not</u> affect the value of a positive binary number, and leading ones <u>do not</u> affect the value of a negative number. So:

    01101 = 00001101 = 13   and   11011 = 11111011 = -5

Manipulating Binary numbers - part 2

- **Binary addition simply consists of applying, to each column in the sum, the rules:**

  <u>0 + 0 = 0</u> <u>1 + 0 = 0 + 1 = 1</u> <u>1 + 1 = 10</u>

  - With 2's complement representation, this works for both positive and negative integers *so long as both numbers being added are represented with the same number of bits.*

e.g. to add the number 13 => 00001101 (8 bits) to -5 => 1011 (4 bits):
we have to *<u>sign-extend</u>* (SEXT) the representation of -5 to 8 bits:

00001101
<u>11111011</u>
00001000 => 8 (as expected!)

Manipulating Binary numbers - part 3

- **Overflow**
  - If we add the two (2's complement) 4 bit numbers representing 7 and 5 we get :

    0111 => +7

    <u>0101</u> => <u>+5</u>

    1100 => -4 (in 4 bit 2's comp.)

    - ***We get -4, not +12 as we would expect !!***
    - ***We have <u>overflowed</u> the range of 4 bit 2's comp. (-8 to +7), so the result is invalid.***
    - ***Note that if we add 16 to this result we get back 16 - 4 = 12***
      - this is like "stepping up" to 5 bit 2's complement representation
  - In general, if the sum of two positive numbers produces a negative result, or vice versa, an overflow has occurred, and the result is invalid in that representation.

Slides prepared by Walid A. Najjar & Brian J. Linard, University of California, Riverside

Limitations of integer representations

- **Most numbers are not integer!**
  - Even with integers, there are two other considerations:
- **Range:**
  - The magnitude of the numbers we can represent is determined by how many bits we use:
    - *e.g. with 32 bits the largest number we can represent is about +/- 2 billion, far too small for many purposes.*
- **Precision:**
  - The exactness with which we can specify a number:
    - *e.g. a 32 bit number gives us 31 bits of precision, or roughly 9 figure precision in decimal repesentation.*
- **We need another data type!**

Rational numbers

- **Our decimal system handles non-integer *rational* numbers by adding yet another symbol - the decimal point (.) to make a *fixed point* notation:**
  - e.g. $3,546.78 = 3.10^3 + 5.10^2 + 4.10^1 + 6.10^0 + 7.10^{-1} + 8.10^{-2}$
- **The *floating point*, or scientific, notation allows us to represent very large and very small numbers (integer or rational), with as much or as little precision as needed:**
  - Unit of electric charge $e = 1.602\ 176\ 565 \times 10^{-19}$ Coul.
  - Volume of universe = 3×10^{86} cm^3
    - ***the two components of these numbers are called the mantissa (3) and the exponent (86)***

Rational numbers in binary

- **We mimic the decimal floating point notation to create a "hybrid" binary floating point number:**
  - **We first use a "binary point" to separate whole numbers from fractional numbers to make a fixed point notation:**
    - ***e.g. 00011001.110 = $1x2^4 + 1x2^3 + 1x2^0 + 1x2^{-1} + 1x2^{-2}$ => 25.75***

      (2^{-1} = 0.5 and 2^{-2} = 0.25, etc.)
  - **We then "float" the binary point:**
    - ***00011001.110 => 1.1001110 x 2^4***

      mantissa = 1.1001110, exponent = 4
  - **Now we have to express this without the extra symbols (x, 2, .)**
    - ***by convention, we divide the available bits into three fields:***

      sign, mantissa, exponent

IEEE-754 fp numbers - 1

32 bits:	1	8 bits	23 bits
	s	biased exp.	fraction

$$N = (-1)^{s} \times 1.\text{fraction} \times 2^{(\text{biased exp.} - 127)}$$

- Sign: 1 bit
- Mantissa (1.fraction): 23 bits
  - We "normalize" the mantissa by dropping the leading 1 and recording only its fractional part (why?)
- Exponent: 8 bits
  - In order to handle both +ve and -ve exponents, we add 127 to the actual exponent to create a "biased exponent":
    - *2^{-127} => biased exponent = 0000 0000 (= #0)*
    - *2^{0} => biased exponent = 0111 1111 (= #127, as an "unsigned magnitude")*
    - *2^{+127} => biased exponent = 1111 1110 (= #254, as an "unsigned magnitude")*

IEEE-754 fp numbers - 2

- **Example:**
  - *25.75 => 00011001.110 => 1.1001110 x 2^4*
  - *sign bit = 0 (+ve)*
  - *normalized mantissa (fraction) = 100 1110 0000 0000 0000 0000*
  - *biased exponent = 4 + 127 = 131 => 1000 0011*
  - *so 25.75 => 0 1000 0011 100 1110 0000 0000 0000 0000 => x41CE0000*
- **Values represented by convention:**
  - Infinity (+ and -): exponent = 255 (1111 1111) and fraction = 0
  - NaN (not a number): exponent = 255 and fraction ≠ 0
  - Zero (0): exponent = 0 and fraction = 0
    - *note: exponent = 0 => fraction is* de-normalized, *i.e no hidden 1*

IEEE-754 fp numbers - 3

- **Double precision (64 bit) floating point**

64 bits:	1	11 bits	52 bits
	s	biased exp.	fraction

$$N = (-1)^s \times 1.\text{fraction} \times 2^{(\text{biased exp.} - 1023)}$$

- **Range & Precision:**
  - **32 bit:**
    - mantissa of 23 bits + 1 => approx. 7 digits decimal
    - $2^{+/-127}$ => approx. $10^{+/-38}$
  - **64 bit:**
    - mantissa of 52 bits + 1 => approx. 15 digits decimal
    - $2^{+/-1023}$ => approx. $10^{+/-306}$

Other Data Types

- **Other numeric data types**
  - e.g. BCD *(binary coded decimal)*
- **Bit vectors & masks**
  - we frequently deal with the individual bits themselves
    e.g. bitwise logic operations AND, NOT
- **Text representations**
  - ASCII: uses 8 bits to represent main Western alphabetic characters & symbols, plus several “control codes”,
  - Unicode: 16 bit superset of ASCII providing representation of many different alphabets and specialized symbol sets.
  - EBCDIC: IBM’s mainframe representation.

Hexadecimal Representation

- **Base 16 (hexadecimal)**
  - More a convenience for us humans than a true data type
  - 0 to 9 represented as such
  - 10, 11, 12, 13, 14, 15 represented by A, B, C, D, E, F
  - $16 = 2^4$: i.e. every hexadecimal digit can be represented by a 4-bit binary (unsigned) and vice-versa.

- **Example**

$$(16AB)_{16} = x16AB$$
$$= 1.16^3 + 6.16^2 + 10.16^1 + 11.16^0$$
$$= (5803)_{10} = \#5803$$
$$= b0001\ 0110\ 1010\ 1011$$

Slides prepared by Walid A. Najjar & Brian J. Linard, University of California, Riverside

Another use for bits: Logic

- Beyond numbers
  - *logical variables* can be *true* or *false*, *on* or *off*, etc., and so are readily represented by the binary system.
  - A logical variable A can take the values *false = 0* or *true = 1* only.
  - The manipulation of logical variables is known as Boolean Algebra, and has its own set of operations - which are not to be confused with the arithmetical operations of the previous section.
  - Some basic operations: NOT, AND, OR, XOR

Slides prepared by Walid A. Najjar & Brian J. Linard, University of California, Riverside

Basic Logic Operations

- **Truth Tables of Basic Operations**

NOT

A	A'
0	1
1	0

AND

A	B	A.B
0	0	0
0	1	0
1	0	0
1	1	1

OR

A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1

- **Equivalent Notations**
  - not A = A' = $\overline{A}$
  - A and B = A.B = A $\wedge$ B = A intersection B
  - A or B = A+B = A $\vee$ B = A union B

Slides prepared by Walid A. Najjar & Brian J. Linard, University of California, Riverside

More Logic Operations

- **XOR and XNOR**

XOR

A	B	A⊕B
0	0	0
0	1	1
1	0	1
1	1	0

XNOR

A	B	(A⊕B)'
0	0	1
0	1	0
1	0	0
1	1	1

- **Exclusive OR (XOR): either A or B is 1, not both**
- **$A \oplus B = A.B' + A'.B$**