Data Representation

Topics:

Representing Floating Point Numbers

Representing Text

Representing Signed Integers

Arithmetic: Addition and Subtraction

Prof. Dr. Slim Abdennadher

27.12.2007

Decimal Floating Points

- A **floating point number** is a number that can contain a fractional part, e.g. 30.875.
- In the decimal system, digits appearing in the right of the floating point represent a value between zero and nine, times an increasing negative power of ten.
- For example the value 30.875 is represented as follows:

$$3 \times 10^1 + 0 \times 10^0 + 8 \times 10^{-1} + 7 \times 10^{-2} + 5 \times 10^{-3}$$

- Similarly, the value 10.110011_2 is represented as follows:

$$1 \times 2^1 + 0 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 0 \times 2^{-4} + 1 \times 2^{-5} + 1 \times 2^{-6}$$

Converting Decimal Floating Points to Binary

- **Integer part**: successive division
- **Fraction Part**: Multiply decimal fraction by 2 and collect resulting integers from top to bottom

Example 1: Convert 30.875

$$30 = 11110_2 \text{ (successive division)}$$

$$\begin{aligned} 0.875 \times 2 &= 1.750 \\ 0.75 \times 2 &= 1.5 \\ 0.5 \times 2 &= 1.0 \end{aligned}$$

Therefore $30.875 = 11110.111$

Example 2: Convert 43.828125

- $43 = 101011_2$

$$
\begin{aligned}
0.828125 \times 2 &= 1.65625 \\
0.65625 \times 2 &= 1.3125 \\
0.3125 \times 2 &= 0.625 \\
0.625 \times 2 &= 1.25 \\
0.25 \times 2 &= 0.5 \\
0.5 \times 2 &= 1.0
\end{aligned}
$$

Therefore $43.828125 = 101011.110101$

Representing Floating Point Numbers

- Three steps process:
  - **Convert the decimal number to a binary number**
  - **Write binary number in "normalized" scientific notation**
  - **Store the normalized binary number**
- Look at an example:
  - How do we store the number 5.75

Converting Decimal Floating Points to Binary

- Integer part: with one of the presented algorithms
- Fraction Part: Multiplication decimal fraction by 2 and collect resulting integers

  Example: Convert 5.75

$$5 = 101_2$$

$$0.75 = \frac{1}{2} + \frac{1}{4} = 2^{-1} + 2^{-2} \text{ which is binary } 0.11$$

$$\begin{aligned} 0.75 \times 2 &= 1.5 \\ 0.5 \times 2 &= 1.0 \end{aligned}$$

Therefore $5.75 = 101.11$

Floating Point Numbers: Normalized Scientific Notation

- **Scientific notation:** $\pm M \times B^{\pm E}$
  - B is the **base**, M is the **mantissa**, E is the **exponent**.
  - Example (decimal, base 10):
    $3 = 3 \times 10^0$
    $2020 = 2.02 \times 10^3$
- Easy to convert to scientific notation:
  - 101.11×2^0
- **Normalize** to get the "." in front of first (leftmost) "1" digit
  - Increase exponent by one for each location "." moves left (decreases if we have to move right)
  - $101.11 \times 2^0 = 10.111 \times 2^1 = 1.0111 \times 2^2 = .10111 \times 2^3$
  - $\underline{101011}.110101 \times 2^0 = .101011110101 \times 2^6$
  - $.\underline{0}1101 \times 2^0 = .1101 \times 2^{-1}$

Floating Point Numbers: Storing the Normalized Number

Storing decimal numbers using 16 bits:

Sign of mantissa	Mantissa	Sign of exponent	Exponent
1 bit	9 bits	1 bit	5 bits

Example 1: $+.10111 \times 2^3$

- Mantissa: $+.10111$
- Exponent: 3

0	101110000	0	00011

Example 2: $-.101 \times 2^{-1}$

- Mantissa: $-.101$
- Exponent: -1

1	101000000	1	00001

Representing Text

- How can we represent text in binary form?
  - Assign to each character a positive integer value (e.g. A is 65, B is 66, a is 97, . . .)
  - Then we can store the numbers in their binary form
- The mapping of text to numbers: **Code Mapping**
- Various conventions for representing characters.
  - American Standard Code for Information Interchange (**ASCII**): each letter 8 bits (only $2^7 = 128$ different characters can be represented)
  - **Unicode**: each letter 16 bits

Representing Text: ASCII Code

Binary	**Decimal**	**Character**
0010 0001	33	!
0010 0010	34	“
...	...	...
0010 1000	40	(
0010 1001	41	)
...	...	...
0100 0001	65	A
0100 0010	66	B
...	...	...
0110 0001	97	a
...	...	...

$$BAD! = 01000010 \quad 01000001 \quad 01000100 \quad 00100001$$

Representing Unsigned Integers

- For a 3-bit unsigned integer, the **range** is $[0, 7]$ which has a binary representation of

$$\begin{aligned} 000 &= 0 \\ 001 &= 1 \\ 010 &= 2 \\ \ldots \\ 110 &= 6 \\ 111 &= 7 \end{aligned}$$

- For a 16-bit unsigned integer, the **range** is $[0, 2^{16} - 1] = [0, 65535]$ which has a binary representation of

$$\begin{aligned} 0000000000000000 &= 0 \\ 0000000000000001 &= 1 \\ &\ldots \\ 1111111111111111 &= 65535 \end{aligned}$$

Range of Unsigned Integers

In general, the **range** for an r-bit integer in base n

$$[0, n_{10}^{r} - 1]$$

Binary Arithmetic: Unsigned Addition

- The algorithm of binary addition is the same as that for decimal addition except that we use the binary addition table:

A	B	Carry Digit	Unit Digit
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

- **Example:**

$$\begin{array}{ccccccccccc} & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & \text{Carries} \\ & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & \\ + & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & \\ \hline & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & \end{array}$$

Binary Numbers: Signed Integers

There are different representations for signed integers

- **Sign/magnitude representation:**
  - 1010 0001 0110 1111
- **One's complement representation:**
  - 1101 1110 1001 0000
- **Two's complement representation:**
  - 1101 1110 1001 0001
- The above examples are all the same number, -8559_{10}

Signed Integers: Sign/Magnitude Representation

- The leftmost bit is used as a sign bit, where 0 indicates a positive integer and 1 indicates a negative integer (**Sign/magnitude**).
- A 16-bit computer represents the value -14 as

$$1000000000001110$$

- For a 16-bit signed integer, the **range** is $[-(2^{16-1}-1),(2^{16-1}-1)] = [-32767, 32767]$
- In general, the **range** for an r-bit signed integer in base n is $[-(n_{10}^{r-1}-1),(n_{10}^{r-1}-1)]$
- **Problem:** The value 0 has two representations
  - Most computers use a different representation (**two's complement**)

Signed Numbers: One's Complement

- Any pattern whose sign bit is 1 represents a negative value.
- Change every 1 to 0 and every 0 to 1.

Positive numbers	**Negative numbers**
$00000000 = 0_{10}$	$11111111 = -0_{10}$
$00000001 = 1_{10}$	$11111110 = -1_{10}$
$00000010 = 2_{10}$	$11111101 = -2_{10}$
$00000011 = 3_{10}$	$11111100 = -3_{10}$
$00000100 = 4_{10}$	$11111011 = -4_{10}$
...	...

- **Advantage over sign/magnitude:** easier to perform arithmetic
- **Problem:** Two distinct representations of zero (still!)

Signed Numbers: Two's Complement

- Any pattern whose sign bit is 1 represents the negative of the value obtained by:
  - invert each 1 to a 0 and each 0 to a 1,
  - then adding 1 to the final value.
- **Example:**

$$
\begin{array}{cccccccccl}
1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & = & -5_{10} \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & & \text{invert step} \\
 & & & & & & + & 1 & & \text{add 1 step} \\
\hline
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & = & 5_{10}
\end{array}
$$

- **Example:**

$$
\begin{array}{cccccccccl}
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & = & 8_{10} \\
1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & & \text{invert step} \\
 & & & & & & + & 1 & & \text{add 1 step} \\
\hline
1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & = & -8_{10}
\end{array}
$$

Signed Numbers: Two's Complement

Positive numbers	**Negative numbers**
$00000000 = 0_{10}$	$00000000 = 0_{10}$
$00000001 = 1_{10}$	$11111111 = -1_{10}$
$00000010 = 2_{10}$	$11111110 = -2_{10}$
$00000011 = 3_{10}$	$11111101 = -3_{10}$
$00000100 = 4_{10}$	$11111100 = -4_{10}$
...	...

- Solves the problem of two representations of zero
- However the negative value (8 bit-value version) 10000000 does not have an equivalent positive representation
- Therefore, for a 16-bit signed integer, the **range** is $[-2^{16-1}, 2^{16-1} - 1] = [-32768, 32767]$

Binary Addition: Two's Complement

- **Assumption:** Fixed number of bits available
- **Addition:** adding the 2's complement representations regardless their sign.
- If there is any carry from the leftmost digits it is ignored.
- **Example:** $(-1)_{10} + (-1)_{10}$

$$\begin{array}{cccccccc} & & 1 & 1 & 1 & 1 & = & \text{-1} \\ + & & 1 & 1 & 1 & 1 & = & \text{-1} \\ \hline & 1 & 1 & 1 & 1 & 0 & & \end{array}$$

Discard the overflow: 1110 (-2_{10})

Binary Subtraction: Two's Complement

- Use Addition to do subtraction: $A - B = A + (-B)$
- Convert the second number to two's complement and perform the binary addition
- **Example:** $17_{10} - 10_{10} = 7_{10}$.

17	=	010001	
10	=	001010	
Complement:		110101	
+1:		110110	(-10 in two's complement)

$$
\begin{array}{lr}
 & 010001 \\
+ & 110110 \\
\hline
 & 1000111 \\
\text{cut off the 1} & 000111
\end{array}
$$

Binary Subtraction: Two's Complement

- **Example:** $11_{10} - 15_{10} = -4_{10}$.

11	=	001011	
15	=	001111	
Complement:		110000	
+1:		110001	(-15 in two's complement)

$$\begin{array}{rl} & 001011 \\ + & 110001 \\ \hline & 111100 \end{array}$$

- To get the decimal value of the output, look first at the **leftmost bit**:
  - If it is a 0, convert the number to decimal directly (multiplication by weights)
  - If it is a 1, find the 2's complement of the number, then convert to decimal (multiplication by weights)
- 111100: 2's complement is $000100 = 4_{10}$, decimal value of the result is -4_{10}