CSEN102 – Introduction to Computer Science

Lecture 8:
Representing Information II

Prof. Dr. Slim Abdennadher
Dr. Aysha Alsafty
slim.abdennadher@guc.edu.eg,
aysha.alsafty@guc.edu.eg

German University Cairo, Department of Media Engineering and Technology

09.12.2017 - 14.12.2017

What you should have learned so far...

- We are used to a base 10 positional system
- Other bases (20, 60) were used through history
- Generally, a base n system encodes numbers as follows:

$$(x_i x_{i-1} \ldots x_1 x_0)_n = x_i \times n^i + x_{i-1} \times n^{i-1} + \ldots + x_1 \times n^1 + x_0 \times n^0$$

- We can convert any positional system into any other positional system
  1. Write down the digits
  2. Multiply each digit by its positional value (the respective power of the base)
  3. Add the products
- Some conversions are very convenient (binary–octal, binary–hexadecimal, ...)
- Binary is ideal for computers

Second summary

- You have learned to juggle with bases and numbering systems
- You understand why binary is convenient for computers

Unsigned numbers in binary representation

Range of n-bit unsigned integer

From 0 to $2^n - 1$.

Example

- 3-bits can represent the numbers from 0 to 7
- 4-bits can represent the numbers from 0 to 15
- ....
- 8-bits can represent the numbers from 0 to 255

Beyond positive integer values

Problem:

Just integer numbers are insufficient. We need also

- Fractions
- Negative values
- Non-numeric data (characters)
- Visual, audio, media, . . .

Solution:

- Binary encoding
  - We encode other data as numbers in binary
- Interpretation
  - We interpret data according to its encoding

In other words we distinguish internal and external representation.

Decimal and binary floating point numbers

- 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 (Convert 30.875)

$30 = 11110_2$ (by 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 $30.875 = 11110.111$

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 (Convert 43.828125)

$$43 = 101011_2 \text{ (by successive division)}$$

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

Therefore $43.828125 = 101011.110101$

Floating point numbers: normalized scientific notation

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

Example ($+.10111 \times 2^3$)

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

0	101110000	0	00011

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

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

1	101000000	1	00001

Simple arithmetics

Question

How do we add binary numbers

Answer: The same way as decimals.

Example

			1	0	1	0	1	1
+		1	1	0	0	0	1	0
(carries)	1	1	0	0	0	1	0	
	1	0	0	0	1	1	0	1

What about subtraction?

Negative numbers in binary representation

Question

How would you represent negative integer numbers?

We would like a representation, that...

- ...makes it simple to change the sign
- ...makes it simple to calculate sums
- ...uses the full capacity of the binary numbers (*i. e.*, no redundant encodings)

Negative numbers in binary representation

First idea: Sign/Magnitude Representation

We use the first bit as a sign.

Range of n-bit integer in Sign/Magnitude

From $-2^{n-1} - 1$ to $2^{n-1} - 1$.

Example (8-bit numbers)

- An 8 bit binary can represent the numbers from 0 to 255.
- If the first bit is a sign, then we can represent the numbers from -127 to 127.

Properties:

- It is easy to switch the sign. ☺
- The sums are overly complicated. ☹
- Instead of 256 numbers, we have only 255, with two

Negative numbers in binary representation

Second idea: One's Complement

Do not only change the sign bit. Instead, change every bit!

Range of n-bit integer in One's Complement

From $-2^{n-1} - 1$ to $2^{n-1} - 1$.

Example

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

Negative numbers in binary representation

Properties of One's Complement:

- It is easy to switch the sign (just flip every bit). ☺
- We can use the same algorithm for positive and negative sums. ☺

Example $(5 + (-9))$

	00000101	5
$+$	11110110	-9
carries	0000100	
	11111011	-4

- We still have two representations for 0.
  1. 00000000, or
  2. 11111111. ☹

Negative numbers in binary representation

Third idea: Two's Complement

Flip every bit (just like with One's Complement) and add 1.

Range of n-bit integer in Two's Complement

From -2^{n-1} to $2^{n-1} - 1$.

Example

Positive numbers	Negative numbers
$00000000 = 0_{10}$	$11111111 = -1_{10}$
$00000001 = 1_{10}$	$11111110 = -2_{10}$
$00000010 = 2_{10}$	$11111101 = -3_{10}$
$00000011 = 3_{10}$	$11111100 = -4_{10}$
$00000100 = 4_{10}$	$11111011 = -5_{10}$
...	...

Negative numbers in binary representation

Properties of Two's Complement:

- It is somewhat easy to switch the sign (just flip every bit and add one). ☺
- We can still use the same algorithm for positive and negative sums. ☺

Example $(5 + (-9))$

	00000101	5
$+$	11110111	-9
carries	0000111	
	11111100	-4

- The range is now -128 (10000000) to 127 (01111111), so all positions are used. ☺

Summary

- What is the range of a n-bit unsigned binary integer?
  from 0 to $2^n - 1$
- What is the range of an n-digit unsigned base-b integer?
  from 0 to $b^n - 1$
- What is the range of a n-bit binary two's complement integer?
  from -2^{n-1} to $2^{n-1} - 1$
- What is the advantage of the two's complement representation for negative numenrals over others?
  - Addition of negative numbers is exactly the same a with positive numbers
  - It uses the full capacity of an n-bit numeral
- How do I convert a positive binary into the corresponding two's complement negative?
  - Flip every bit
  - Add 1

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

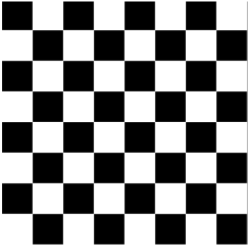
Representing text: ASCII Code

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 01000001 01000100 00100001

Representing Images

- How can we represent images in binary form?
  - Images are also stored in binary to be processed by the computer, so that they can be seen on the screen
  - Images are made up of pixels. Each pixel in an image has a color, and the color is stored in binary
  - We can create a simple black and white image by using a sequence of 0's (for black pixels) and 1's (for white pixels). Thus, using only 1-bit for the pixels color



Black/White and RGB Images

- To have more than two colors (black and white), more bits will be needed to represent a pixel. Each pixel's color sample has three numerical RGB components (Red, Green, Blue) to represent the color of that tiny pixel area. These three RGB components are three 8-bit numbers for each pixel. Thus, we can create colored (RGB) images using 24 bits per pixel

Representing Colors

- How are colors represented in computers?
  - Computers represent everything in binary, including colors
  - The color is defined by its mix of Red, Green and Blue, each of which can be represented using 8-bits. Thus, each having a range:
    - 0 to 255 (in decimal)
    - 00 to FF (in hexadecimal)
  - In total there are: $256 \times 256 \times 256 = 256^3 = 16,777,216$ possible color combinations

Representing Colors

- How are colors represented in computers?
  - The web pages and some computer programs use the hexadecimal format to represent a color. #RRGGBB, where RR is how much Red (using two hexadecimal digits), GG is how much Green, and BB how much Blue.
    - #FF0000 : Red
    - #00FF00 : Green
    - #0000FF : Blue
    - #000000 : Black
    - ...
    - #FFFFFF : White
    - For example: (64,48,255) in decimal, which is equal to (40,30,FF) in hexadecimal and would be coded as #4030FF

Connection to the world of hardware

- Internally, computers represent all information using the binary number system
- Why? Electronic devices that are based on only two easily distinguishable states are cheaper and more reliable than devices based on more than two states.
- It does not matter if the two states are 0 and 1, or “off” and “on”, or “closed” and “open” or “low” and “high”, or whatever.
- All that matters is that the two states be distinguishable and stable.

Summary

- You learned base conversions between different numbering systems
- You learned why computers use binary
- You have seen representations of
    - Unsigned, positive, numbers
    - Characters
    - Signed, normalized, floating-point numbers
    - Two's complement signed integers
- Other data is also represented numerically

Summary

- Operations you should know:
  - Converting a base-*n* number into base-*m*.
  - Converting a floating-point number into normalized form.
  - Representing a binary normalized floating-point number in a 16-bit word
  - Converting a negative number into binary Two's Complement
  - Adding Two's Complement numbers.

Question

- Algorithms often use true/false flags
- Such flags are internally represented by a 8-bit Two's Complement number
- "False" is represented by 0
- How would you represent "true"?

Summary

- Question the numbering systems you know. Decimal is not always ideal.
- Computers represent information internally as binary numbers
- Numbers are build out of two states: on, off
- Storing data requires encoding and interpretation. Distinguish internal and external representation.
- We saw how to represent as binary data:
  - Numbers (integers, floating point)
  - Text (code mappings as ASCII and Unicode)
  - Images
  - Colors