Computer Programming

Representing data

Robert Varga

Technical University of Cluj-Napoca
Computer Science Department

Seminar 1

Contents

Number systems

- modern computers use circuits with two states - representing data in base 2 is convenient
- humans use base 10
- we need conversion methods
- a *bit* is defined as a binary digit
- a *byte* consists of 8 bits

shorthand	name	bytes 10^x	bytes 2^x
KB	kilobyte	10^3	2^{10}
MB	megabyte	10^6	2^{20}
GB	gigabyte	10^9	2^{30}
TB	terabyte	10^{12}	2^{40}
PB	petabyte	10^{15}	2^{50}

Converting natural numbers

From decimal to binary

- repeatedly divide the current number by 2
- write the remainder on the right
- change to current number to the quotient
- repeat until quotient is 0
- the digits in binary are the remainders, in reverse order

Converting natural numbers - Examples

23	1
11	1
5	1
2	0
1	1
0	

$$23_{(10)} = 10111_{(2)}$$

100	0
50	0
25	1
12	0
6	0
3	1
1	1
0	

$$100_{(10)} = 1100100_{(2)}$$

Converting natural numbers

Algorithm S1

```
read x, b
while x > 0 do
    print x % b
    x = [ x / b ]
end while
```

- this works for strictly positive x
- general conversion for any base $b > 1$
- the digits are shown in reverse order
- this can be easily corrected by saving them in an array

Converting natural numbers

From binary to decimal

- each digit has a weight equal to $2^{position}$
- the digit position starts from zero and from the the right
- sum up the digits multiplied by their weights

$$x = \overline{b_{n-1} \ldots b_1 b_0}_{(2)}$$

$$x = 2^{n-1} b_{n-1} + \ldots + 2^1 b_1 + 2^0 b_0$$

Example:

$$101110_{(2)} = 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 32 + 8 + 4 + 2 = 46_{(10)}$$

Converting natural numbers - related bases

- because binary representation is long, programmers use octal and hexadecimal
- hexadecimal uses 16 digits, assigning letters to digits above 9:

$$A = 10, B = 11, C = 12, D = 13, E = 14, F = 15$$

- when converting from base b to base b^k there is a simple method
- group digits into groups of k, starting from the right (least significant)
- convert each group of k digits in base b to base b^k

Examples:

$y = 1010100101_{(2)} = 10\ 1010\ 0101_{(2)} = 2A5_{(16)}$

$z = 1010100101_{(2)} = 1\ 010\ 100\ 101_{(2)} = 1245_{(8)}$

Storing natural numbers

- positive natural numbers are stored in *unsigned integer* types
- use binary representation
- add leading zeros if necessary
- bits are grouped into bytes
- the least significant byte is stored first (little-endian)
- if n bits are used
  - the smallest representable value $= 0...000_{(2)} = 0$
  - the largest representable value $= 1...111_{(2)} = 2^n - 1$

Storing natural numbers - Example

Determine the memory representation of the number 314 as an *unsigned short int*.

- convert the number to binary $314_{10} = 100111010_2$
- unsigned short int has 2 bytes
- add leading zeroes up to 16 bits = 0000 0001 0011 1010
- store the bytes starting from the least significant one:
- byte 0: 0011 1010
- btye 1: 0000 0001

Storing natural numbers

Domain of unsigned types

type	size [bytes][1]	minimum	maximum
unsigned char	1	0	$2^8 - 1 = 255$
unsigned short int	2	0	$2^{16} - 1 = 65k$
unsigned int	4	0	$2^{32} - 1 = 4.3 \cdot 10^9$
unsigned long long int	8	0	$2^{64} - 1 = 1.8 \cdot 10^{19}$

[1] dimensions are given for Windows 64bit

Converting negative numbers

- the encoding should not change the rules of addition
- we want: $-x + x = 0$, for any x encoded on n bits
- we have the following property: $x + \tilde{x} = 1...111_{(2)}$ (n bits)
- where $\tilde{x}$ has all of x's bits flipped = *one's complement*
- furthermore, $1...111_{(2)} + 1 = 10...000_{(2)}$
- the last number has one active bit and n inactive bits
- but this cannot be represented on n bits
- we will ignore the leading active bit
  - this is called overflow
- so: $x + \tilde{x} + 1 = 0$ (with overflow)
- from this $\boxed{-x = \tilde{x} + 1}$

Converting negative numbers

- we call $\tilde{x} + 1$ the *two's complement* representation of -x
- all signed types in C use this representation
- types are signed, unless they are modified with the unsigned keyword

Properties:

- $-1 = 1...111_{(2)}$ all n bits equal 1 in two's complement
- $-10...000_{(2)} = 10...000_{(2)}$ according to the conversion rule
- this is impossible, so $-10...000_{(2)}$ doesn't have a valid C2 representation
- all negative numbers have the first bit active

Converting negative numbers

From decimal to two's complement:

- convert -x into binary
- flip all the bits from this representation
- add one (in binary) to the number

Example:
Find the representation of -24 in two's complement on n = 8 bits.

$-24 = \tilde{24} + 1$

$24 = 0001\ 1000$

$-24 = 1110\ 0111 + 1$

$-24 = 1110\ 1000\ _{C2}$

Converting negative numbers

From two's complement to decimal:

- each digit has a weight equal to $2^{position}$
- except the most significant digit, which has a weight of -2^{n-1}
- the digit position starts from zero and from the right
- sum up the digits multiplied by their weights

$$x = \overline{b_{n-1}b_{n-2}\ldots b_1 b_0}_{C2}$$
$$x = -2^{n-1}b_{n-1} + 2^{n-2}b_{n-2} + \ldots + 2^1 b_1 + 2^0 b_0$$

Example:

$10101010_{C2} = -1\cdot 2^7 + 1\cdot 2^5 + 1\cdot 2^3 + 1\cdot 2^1 = -128 + 32 + 8 + 2 = -86_{(10)}$

Storing negative numbers

- negative whole numbers are stored in C2
- the first bit (most significant) shows the sign of the number
  - first bit inactive = positive number
  - first bit active = negative number
- if we have a valid representation of the number x on m bits it can be simply extended to $n > m$ bits
  - by adding leading zeroes to positive numbers
  - by adding leading ones to negative numbers

Storing negative numbers

Case with n = 3 bits

C2	10
000	0
001	1
010	2
011	3
100	-4
101	-3
110	-2
111	-1

- numbers less than 2^{n-1} are represented like natural numbers
- after the largest representable number, there is a jump to the smallest representable number -2^{n-1}
- besides this jump, the numbers increase by one
- no duplicate representation for zero

Storing negative numbers

Domain of signed types

type	size [bytes][2]	minimum	maximum
char	1	$-2^7 = -128$	$2^7 - 1 = 127$
short int	2	-2^{15}	$2^{15} - 1 = 32k$
int	4	-2^{31}	$2^{31} - 1 = 2 \cdot 10^9$
long long int	8	-2^{63}	$2^{63} - 1 = 9 \cdot 10^{18}$

[2] dimensions are given for Windows 64bit

Converting the fractional part of numbers

From decimal to binary

- repeatedly multiply the current number by 2
- write the whole part on the right
- change to current number to the fractional part
- repeat until the fractional part is 0 or maximum precision is reached
- the digits in binary are the whole parts, in correct order

Converting the fractional part of numbers

Examples

0.15625	0
0.3125	0
0.625	1
0.25	0
0.5	1
0	

$$0.15625_{(10)} = 0.00101_{(2)}$$

0.1	0
0.2	0
0.4	0
0.8	1
0.6	1
0.2	0
0.4	0
...	...

$$0.1_{(10)} = 0.0(0011)_{(2)}$$

Converting the fractional part of numbers

Algorithm S2

read x, b, n
while $x > 0$ and $n > 0$ **do**
  d = [b x]
  print d
  x = b x - d
  n = n - 1
end while

- this works for $x \in (0, 1)$
- general conversion for any base $b > 1$
- prints the digits after the decimal point in correct order
- n stores the number of digits to calculate

Converting the fractional part of numbers

From binary to decimal

- each digit has a weight equal to $2^{position}$
- the digit position starts from -1 from the right of the decimal point and decreases as we go to the right
- sum up the digits multiplied by their weights

$$x = \overline{0.b_{-1}b_{-2}...b_{-k}}_{(2)}$$
$$x = 2^{-1}b_{-1} + 2^{-2}b_{-2} + ... + 2^{-k}b_{-k}$$

Example:

$$0.1001_{(2)} = 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 0 \cdot 2^{-3} + 1 \cdot 2^{-4}$$
$$= \frac{1}{2} + \frac{1}{16} = \frac{9}{16} = 0.5625_{(10)}$$

Storing rational numbers

- rational numbers are stored using the floating point representation
- the absolute value of the integer part can be converted to binary using Algorithm S1
- the fractional part can be converted to binary using the Algorithm S2
- concatenate the two parts
- the sign is stored in the sign bit
- does not use two's complement (C2)

Floating point representation

- the floating point representation of a number is of this form:

$$x = \pm\overline{a.bcde\ldots} \cdot b^{exponent}$$

- when working with numbers we care about its sign, the first digits and its magnitude
  - distances between two cities: $120\ \text{km} = 1.2 \cdot 10^5\ \text{m}$
  - the charge of an electron $= -1.60217662 \cdot 10^{-19}$ Coulomb
- the name reflects how multiplying a number by the base to a power changes the position of the decimal point

Floating point representation

- all real types in C (float, double, long double) use floating point representation
- each type has
  - 1 bit reserved for sign: 0 positive, 1 negative
  - a bits reserved for the exponent
  - b bits reserved for the mantissa (fraction)
- to find the representation of a number it must be written as:

$$x = \pm 2^{exponent - displacement} \cdot \overline{1.uvw\ldots}$$

- where:
  - the exponent is expressed using a bits
  - there are b digits after the decimal point

Floating point representation

Steps required the find the representation of a decimal number:

- convert the rational number from base 10 to base 2
- move the decimal point to the left or right by multiplying with 2^k
  - note, k can be negative, zero or positive
- calculate the exponent as k + *displacement* and express it in binary
- read the bits of the mantissa as the digits after the decimal point

Floating point representation - Example

- 132.57 represented as a float
- 1 sign bit, 8 bits for exponent, 23 bits for mantissa
- displacement for float is $127 = 0111\ 1111_{(2)}$

$$
\begin{aligned}
132.57 &= 10000100.1001000111101100\ldots_{(2)} \\
&\approx 10000100.1001000111101100_{(2)} \\
&= 2^{7} \cdot 1.00001001001000111101100_{(2)} \\
&= 2^{134-127} \cdot 1.00001001001000111101100_{(2)} \\
&= 2^{10000110_{(2)}-127} \cdot 1.00001001001000111101100_{(2)}
\end{aligned}
$$

- the representation in memory is:
- 0 100 0011 0 000 0100 1001 0001 1110 1100

Floating point representation

Domain of real types

type	size [bytes][3]	exponent	displacement	mantissa bits	minimum	maximum	precision
float	4	8	$127 = 2^7$-1	23	$1.17 \cdot 10^{-38}$	$3.4 \cdot 10^{38}$	6
double	8	11	$1023 = 2^{10}$-1	52	$2.22 \cdot 10^{-308}$	$1.79 \cdot 10^{308}$	15
long double	10	15	$16383 = 2^{14}$-1	64	$3.7 \cdot 10^{-4932}$	$1.2 \cdot 10^{4932}$	18

- normal minimum value $= 2^{-displacement+1}$
- normal maximum value $= 2^{+displacement+1}$
- precision = number of digits that can be accurately represented
- $10^{precision} = 2^{bits\ mantissa}$ so precision $= log_{10}(2)\ bits\ mantissa$
- all bits active in exponent: infinite and not-a-number
- all bits inactive in exponent: subnormal numbers = zero to the left of the decimal point
- IEEE 754 standard; online conversion tool

[3] dimensions are given for Windows 64bit