SYSC 2006
Fall 2019

Carleton UNIVERSITY
Canada's Capital University

Data and Their Representation

Last edited by C.-H. Lung: Sept 11, 2019

Base 10 Number System

- How do we interpret the decimal value 742?
- "7 hundreds, 4 tens, 2 ones"
- $742 \Rightarrow 7 \times 10^2 + 4 \times 10^1 + 2 \times 10^0$
  - Weights associated with digit positions are powers of 10

Base N

- Different bases may be more natural in different circumstances
  - Base 10 (decimal) ⇒ 10 fingers
  - Base 2 (binary) ⇒ On/Off (Digital circuits)
- Base 10 has 10 symbols: 0 1 2 3 4 5 6 7 8 9
- Base 2 has 2 symbols: 0 1
- Base 16 (hexadecimal) has 16 symbols:

  0 1 2 3 4 5 6 7 8 9 <u>A B C D E F</u>

Counting in Binary and Decimal

Carleton University
Canada's Capital University

Value	Base 2	Base 10	Value	Base 2	Base 10
zero	0	0	seven	111	7
one	1	1	eight	1000	8
two	10	2	nine	1001	9
three	11	3	ten	1010	10
four	100	4	eleven	1011	11
five	101	5	...	...	...
six	110	6	fifteen	1111	15

Base N to Decimal Conversion

- Multiply each digit by its weight, sum the results
  - weights in binary are powers of 2
  - weights in base N are power of N
- $d_3 d_2 d_1 d_0$ (base N) =

  $d_3 \times N^3 + d_2 \times N^2 + d_1 \times N^1 + d_0 \times N^0$

Base 2 to Decimal Conversion

- $1001011001_2 = ?_{10}$
- Position of symbol determines exponent

2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	0	0	1	0	1	1	0	0	1_2

$$1 \times 2^9 + 0 \times 2^8 + 0 \times 2^7 + 1 \times 2^6$$
$$+ 0 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2$$
$$+ 0 \times 2^1 + 1 \times 2^0$$

Base 2 to Decimal Conversion

- Short form: $2^9 + 2^6 + 2^4 + 2^3 + 2^0$
- So, 1001011001_2

  $= 2^9 + 2^6 + 2^4 + 2^3 + 2^0$

  $= 512_{10} + 64_{10} + 16_{10} + 8_{10} + 1_{10}$

  $= 601_{10}$

Adding Binary Numbers

- 0 + 0 = 0
- 0 + 1 = 1
- 1 + 0 = 1
- 1 + 1 = 0 carry 1

Data Storage: Cells

- In a computer, data are stored as binary digits (bits) in fixed-size cells
- 8 bits ⇒ byte
  - common size for a computer's memory cells
- 4 bits ⇒ nibble (nybble)
- Cells always contain a pattern of 1s and 0's (are never empty)

Words and Cells

- Computer designers pick "word" size
  - 16 and 32 bits are now typical
    - 2 adjacent 8-bit memory cells are treated as a single 16-bit word
    - 4 adjacent 8-bit memory cells are treated as a single 32-bit word
- Example: 0110110011110010 ⇒ 16-bit word

Representing Values

- With a k-bit word, you can represent 2^k different values
  - 8-bit word (byte) ⇒ 256 different values
  - 9-bit word ⇒ 512 different values
  - 16-bit word ⇒ 65,536 different values
  - 32-bit word ⇒ 4,294,967,296 different values
- Increasing the word size by one bit doubles the number of values it can represent

Carleton UNIVERSITY
Canada's Capital University

Unsigned Integers

- Interpret an 8-bit cell as an unsigned (positive) integer

Cell contents (binary)	Value (base 10)
00000000	0
00000001	1
00000010	2
...	...
11111111	255

- Interpreted as unsigned integers, the values in a k-bit word range from 0 to $2^k - 1$

Signed Integers

- How do we represent signed (positive or negative) binary numbers in a computer if a cell can store only 0's and 1's (no minus sign)?

Carleton UNIVERSITY
Canada's Capital University

Signed Magnitude

- A simple technique for representing signed integers is to use the most significant bit as a sign bit, and the remaining bits to represent the magnitude
- Sign bit
  - 0 $\Rightarrow$ positive
  - 1 $\Rightarrow$ negative

Signed Magnitude

Cell contents (binary)	Value (base 10)
00000000	+0
00000001	+1
00000010	+2
...	...
01111111	+127
10000000	-0
10000001	-1
...	...
11111111	-127

Signed Magnitude

- 8-bit word can represent 128 positive and 128 negative integers
- Drawbacks:
  - 2 representations for 0
  - relatively complex circuits to perform arithmetic operations

2's Complement

Carleton University — Canada's Capital University

- Use most significant bit as a sign bit
  - 0 ⇒ positive
  - 1 ⇒ negative
- If a decimal integer is positive, it is represented in binary as if it was an unsigned integer
- If a decimal integer is negative, we negate the binary representation of the equivalent positive value (how?)

Negation Rule

- How do we negate a binary number?

1. Complement all bits (change all 0's to 1's and all 1's to 0's)
2. Then add 1, ignoring any carry out of the most significant bit

Negating Binary Numbers

- $-1_{10} = ?_2$

```
+1₁₀ = 00000001₂ ⇒    complement ⇒ 11111110₂
                                 add 1        1₂
                                              --------
                                              11111111₂
```

- So, -1_{10} is represented as 11111111_2

Negating Binary Numbers

- $-127_{10} = ?_2$

$$
\begin{array}{lllr}
+127_{10} = 01111111_2 \Rightarrow & \text{complement} & \Rightarrow & 10000000_2 \\
 & \text{add } 1 & & 1_2 \\
 & & & \text{---------} \\
 & & & 10000001_2
\end{array}
$$

- So, -127_{10} is represented as 10000001_2

Negating Binary Numbers

- $-2_{10} = ?_2$

```
+2₁₀ = 00000010₂ ⇒     complement ⇒ 11111101₂
                   add 1                 1₂
                                  ---------
                   11111110₂
```

- So, -2_{10} is represented as 11111110_2

2's Complement

Cell contents (binary)	Value (base 10)
00000000	+0
00000001	+1
...	...
01111111	+127
10000000	-128
10000001	-127
...	...
11111110	-2
11111111	-1

2's Complement

- Interpreted as 2's complement integers, the values in a k-bit word range from -2^{k-1} to $2^{k-1} - 1$

Negating Binary Numbers

- We can negate negative 2's complement numbers using the same technique
- Example: negate $-1_{10} = 11111111_2$

$$
\begin{array}{llr}
11111111_2 \Rightarrow & \texttt{complement} \Rightarrow & 00000000_2 \\
& \texttt{add 1} & 1_2 \\
& & \texttt{---------} \\
\end{array}
$$

$00000001_2 = +1_{10}$

Negating Binary Numbers

- Example: negate $-127_{10} = 10000001_2$

$$
\begin{array}{llr}
10000001_2 \Rightarrow & \texttt{complement} \Rightarrow & 01111110_2 \\
& \texttt{add 1} & 1_2 \\
& & \texttt{---------} \\
& & 011111111_2 = +127_{10}
\end{array}
$$

Negating Binary Numbers

- What happens if we negate 0?

$$0_{10} = 00000000_2 \Rightarrow \text{complement} \Rightarrow 11111111_2$$

$$\text{add 1} \quad 1_2$$

$$\text{---------}$$

$$00000000_2$$

- Using 2's complement notation, there's only 1 representation for 0

Negating Binary Numbers

- What happens if we negate -128_{10}?

```
-128₁₀ = 10000000₂ ⇒   complement ⇒ 01111111₂
                       add 1               1₂
                                    --------
                                    10000000₂
```

- So, negating -128_{10} yields -128_{10}

Base 2 to decimal

- Positive 2's complement binary numbers are converted to decimal using the technique shown earlier for converting unsigned binary numbers to decimal
- For a negative 2's complement number, convert it to a positive value (use the negation rule), perform binary to decimal conversion, then place a minus sign in front of the result

Base 2 to decimal

- Assuming 2's complement representation, $10001100_2 = ?_{10}$
- First, positive or negative?
- The sign bit is 1, so the value is negative, so negate it:

```
10001100₂ ⇒  complement ⇒ 01110011₂
              add 1               1₂
                              ---------
                              01110100₂
```

- Convert to decimal: $01110100_2 = 116_{10}$
- Affix minus sign: $10001100_2 = -116_{10}$

Characters

- Characters are represented as binary values (character codes)
- ASCII - *American Standard Code for Information Interchange*
  - 7-bit code (2^7 = 128 possible values)
  - typically stored in 8-bit bytes, most significant bit set to 0

Characters

- 95 codes
  - upper & lowercase alphabet
  - decimal digits 0 .. 9
  - 32 punctuation characters (., !, @, *, /, }, etc.)
- 33 *control* codes (ring bell, carriage return, line feed, etc.)

Selected ASCII Character Codes

Binary	Char	Binary	Char
00000000	NUL	01000001	A
00100000	space	01000010	B
00100001	!	10011010	Z
00101000	(	01100001	a
00101111	/	01100010	b
00110000	0	01111010	z
00110001	1	01111101	}
00111001	9	01111111	DEL

Floating Point

- Binary numbers can have a *binary point*
- Digits to the right represent a fractional value
- Each digit is weighted by a negative power of 2
- 0.1011_2

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

  $= 0.5_{10} + 0.125_{10} + 0.0625_{10}$

  $= 0.6875_{10}$

Floating Point

- Some decimal fractions produce a repeating fraction when converted to binary
- $0.1_{10} = 0.000110011001100110011..._2$
- To store these values in a fixed-size cell, it must be truncated, introducing a small error
  - e.g., 0.0001100110011_2 is a close approximation of 0.1_{10} but isn't equal to 0.1_{10}

Floating Point

- That's why

```
// Calculate 1.0 as
//   0.1 + 0.1 + 0.1 + ... + 0.1
double x = 0.0;
for (int i = 0; i < 10; i++) {
    x = x + 0.1;
}
```

sets x to `0.99999999999999998181`, not `1.0`, and why `x == 1.0` is false

Representing Floating Point

- A simple technique: store the value in an n-bit word, with an implicit binary point
- Example: use a 24-bit word, implicit binary point after the 12th bit

Contents	Interpretation	Decimal
000000000000000000000000	000000000000.000000000000	0
000000000000000000000001	000000000000.000000000001	1/4096 = 0.00024414
000000000001000000000000	000000000001.000000000000	1
111111111111111111111111	111111111111.111111111111	4095.99975586

Representing Floating Point

- Drawbacks
  - can't represent negative values
  - range of values is too limited for practical purposes

Scientific Notation

- Represent values as a base N mantissa multiplied by the base raised to a power
- $7123660000000_{10} = 71.2366_{10} \times 10^{11}$
- $93450.0_{10} = 9.345_{10} \times 10^{4}$
- $-0.0456_{10} = -45.6_{10} \times 10^{-3}$
- $111010.0_{2} = 11.101_{2} \times 2^{4}$
- $0.01001_{2} = 1.0012 \times 2^{-2}$
  - exponent indicates how far the point is to be shifted, right or left

Normalized Scientific Notation

- Represent real numbers in binary using the format:

  $$+/-\ 1.bbbb...bbbb \times 2^{exponent}$$

  - 0 is a special case

Normalized Scientific Notation

- To store a normalized binary number in a 24-bit word
  - use 8 bits for the exponent
  - use 16 bits for the mantissa
    - 1 bit for the sign (0 = +ve, 1 = -ve)
    - don't store the mantissa's leading one or binary point
    - 15 digits for the fractional part of the mantissa
- See *Principles of Computer Systems*, Fig 1.18