Basics of Computing – Chapter 1.4-1.7
Data Representation

Cory L. Strope

Overview of Computers

Symbols

People understand a large number of symbols:

- {a–z, A–z, 0–9, &, %, #, . . . }
- {a, aardvark, . . . , zulu, zygote}
- Pictures
- Sounds
- `Type` and **size** *of* **text** <u>written</u>
- i.e., e.g., et al, etc; etc; etc.

Computers do these processes using their symbol library:

$$\{0, 1\}$$

Overview of Computers

Binary

Binary is a **base-2** number system having only 2 symbols, $\{0,1\}$.

- Used to represent all information on a computer
- Combining binary symbols allow us to more representational
  - **bit:** 0/1, short for **b**_inary dig_**it**
  - **byte:** 8 bits
  - **kilobyte (KB):** 1024 (2^{10}) bytes
  - **megabyte (MB):** 1024 kilobytes, (2^{20}) bytes
  - **gigabyte (GB):** 1024 megabytes, (2^{30}) bytes
  - **terabyte (TB):** 1024 gigabytes, (2^{40}) bytes

$$2^{40} = 1\,099\,511\,627\,776$$

Overview of Computers

Binary / Hexadecimal

Binary strings can get very long and are difficult to "read" for people.

Hexadecimal is a **base-16** number system, which works as a more compact representation of binary:

Decimal	Bin.	Hex (0x)
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7

Decimal	Bin.	Hex (0x)
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Representing Information as Bit Patterns

The Binary System

Hexadecimal System

Storing Integers

Storing Fractions

Representing Information as Bit Patterns

Types of Information

Computers represent 4 major types of data:

1. Text
2. Numeric Values
3. Images
4. Sound

— · —··— — —·—· ——— —·· ·

01001000 01100101 01101100 01101100 01101111 00101110

Text is normally represented by means of a code.

- `ASCII` `http://www.lookuptables.com/`: Uses 8 bits to encode all keyboard characters, as well as non-visible characters, such as carriage returns (<enter>).
- `Unicode` `http://www.unicode.org/`: Uses 16 bits to represent up to 65536 bit patterns – enough for Chinese, Japanese, Hebrew, ...
- `ISO`: Uses 32 bits. Can encode billions of different symbols.

Why not use only `ISO` or `Unicode`?

Text Code

Hello.

- `ASCII`:

  01010100 01101010 01101101 01100101 00100110 01010011 01110000 01100001 01100011 01100101

- `Unicode`:

  0000000001010100 0000000001101001 0000000001101101 0000000001100101 0000000000100110

  0000000001010011 0000000001110000 0000000001100001 0000000001100011 0000000001100101

- `ISO`: no way.
- Symbols: Time&Space

Space: `ASCII` is 4× smaller than `ISO`; 2× smaller than `Unicode`.
Time: Often, one needs to use multiple keystrokes to make one symbol in larger character sets.

Representation of Numeric Values

Character encoding is inefficient when storing purely numeric values (not to mention, inconvenient).

- Using bits is simply a different way to count. For example, below we can match each bit pattern of a certain length to a corresponding decimal number:

Bit pattern	Decimal number
000	0
001	1
010	2
011	3

- 25 →

  00110010 00110101
  11001

Representation of Numeric Values

Binary numbers ⇌ Decimal numbers?

We need to be able to represent any (positive) integer in binary, and vice versa.

- The decimal system (base-10), representing 375:

$$\begin{array}{ccccccccc} & 0 & & 3 & & 7 & & 5 \\ \times & 10^3 & & 10^2 & & 10^1 & & 10^0 \\ \hline = & 0 & + & 300 & + & 70 & + & 5 \end{array}$$

- Each position is named, e.g. the 10^2 position is named the *hundreds* position.

Binary is **base-2**... What changes?

Representation of Numeric Values

Binary numbers $\rightleftharpoons$ Decimal numbers?

1. **Naming of positions**:

   Most significant bit $\rightarrow\boxed{1}0111011\boxed{1}\leftarrow$ *Least* significant bit.

2. The (*unsigned*) binary system (base-2), representing 25:

$$\begin{array}{ccccccccccc} & 1 & & 1 & & 0 & & 0 & & 1 \\ \times & 2^4 & & 2^3 & & 2^2 & & 2^1 & & 2^0 \\ \hline & (16) & & (8) & & (4) & & (2) & & (1) \\ \hline = & 16 & + & 8 & + & 0 & + & 0 & + & 1 \end{array}$$

3. **Number of positions**: How many positions do we need to represent a decimal number? (i.e. What is the range of values that can be represented by n bits?)

Representation of Numeric Values

Binary numbers $\rightleftharpoons$ Decimal numbers?

The range of values that can be represented with n bits in the unsigned binary system:

- $n = 1$: $0, 1 \rightarrow 2$ values, $\{0, 1\}$.
- $n = 2$: $00, 01, 10, 11 \rightarrow 4$ values, $\{0, 1, 2, 3\}$.
- $n = 3$: $000, 001, 010, \ldots, 110, 111 \rightarrow 8$ values $\{0, 1, \ldots, 7\}$.

In general, n bits can be used to represent 2^n numbers ranging from $0 \ldots 2^n - 1$.

Representation of Numeric Values

Conversion of Numbers from Decimal to Binary

Converting a decimal number to binary is done by repeatedly dividing the number by 2 and writing down the remainder, until the number is equal to 0. Write remainders from bottom to top.

Given the number 13, convert to binary:

- $\frac{13}{2} = 6 \text{ r } \boxed{1}$
- $\frac{6}{2} = 3 \text{ r } \boxed{0}$
- $\frac{3}{2} = 1 \text{ r } \boxed{1}$
- $\frac{1}{2} = 0 \text{ r } \boxed{1}$

Thus, 13 in base-10 is 1101 in base-2.

Representation of Images

Given an image, how can we represent it in binary?

- *Bitmap*
  - Picture is broken up into small units, called picture elements (or *pixels*).
  - Disadvantage: Cannot be rescaled.
- *Vector*
  - Image is represented as a collection of lines and curves.

Representation of Images

Bitmaps – Black and White

http://www.exzooberance.com

- The smallest representation of pictures is 1 bit / pixel.

0	0	1	1	0	0
0	1	0	0	1	0
1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0

- 0 (white), 1 (black).

Representation of Images

Bitmaps – Grayscale

- A more accurate representation for images is grayscale, 1 byte / pixel:

00	00	FF	FF	00	00
00	FF	00	00	FF	00
FF	00	00	00	00	FF
00	FF	00	00	FF	00
00	00	FF	FF	00	00

- This is very popular for wedding pictures!

Representation of Images

Bitmaps – Color

- Color pictures, 3 bytes / pixel:

000000	000000	FFFFFF	FFFFFF	000000	000000
000000	FFFFFF	000000	000000	FFFFFF	000000
000000	FFFFFF	000000	000000	000000	FFFFFF
000000	FFFFFF	000000	000000	FFFFFF	000000
000000	000000	FFFFFF	FFFFFF	000000	000000

Representation of Images

Bitmaps – Issues

Size of each image:

- width = 512 pixels,
- height = 768 pixels.

$$size = w \times h \times bits$$

- $512 \times 768 \times 1 = 393216$ bits or 49152 bytes.
- $512 \times 768 \times 8 = 3145728$ bits or 393216 bytes.
- $512 \times 768 \times 8 \times 3 = 9437184$ bits or 1179648 bytes.

Representation of Images

Bitmaps – Issues

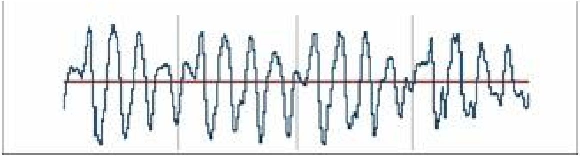
Bitmaps do not rescale well, giving the image a “grainy” appearance.

Representation of Sound

Types of Sound

Sound (IRL) is an **analog** signal.

- Continuous
- Wave-like



http://www.cs.wfu.edu/∼burg/

How can we convert a wave to a binary representation?

Representation of Sound

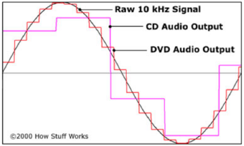
Analog to Digital Conversion – Sampling

Sampling the analog signal (wave) at regular intervals.

- CD: 44100 samples / second (i.e. 44.1 kHz), 16 bits / sample
- A CD has 700 MB capacity, so the maximum playing time of a CD should be:

$$\left(\frac{700\mathrm{MB} \times 1024\frac{\mathrm{KB}}{\mathrm{MB}} \times 1024\frac{\mathrm{B}}{\mathrm{KB}} \times 8\frac{\mathrm{b}}{\mathrm{B}}}{44100\frac{\mathrm{sample}}{\mathrm{s}} \times 16\frac{\mathrm{b}}{\mathrm{sample}}}\right)/60\frac{\mathrm{s}}{\mathrm{min}} = 138.7\mathrm{minutes}.$$

However, CDs only hold 80 minutes of audio. This is because CDs use 14 bits / byte.



Representing Information as Bit Patterns

The Binary System

Hexadecimal System

Storing Integers

Storing Fractions

Binary System

Overview

- Binary Notation
- Binary Addition
- Fractions in Binary

Binary Notation

Review

- A binary number of n bits can represent 2^n numbers with values ranging from $0 \ldots 2^n - 1$.
- Binary Format: $x_i \in \{0, 1\}$, for each position i

$$\begin{array}{cccccccccc}
 & x_{n-1} & & \ldots & & x_2 & & x_1 & & x_0 \\
\times & 2^{n-1} & & \ldots & & 2^2 & & 2^1 & & 2^0 \\
\hline
 & x_n \times 2^{n-1} & + & \ldots & + & x_3 \times 2^2 & + & x_2 \times 2^1 & + & x_1 \times 2^0
\end{array}$$

- 1111
  - $n = 4$, $2^{n-1} = 16$
  - $(1 \times 8) + (1 \times 4) + (1 \times 2) + (1 \times 1) = 15$
- 25

$$\begin{array}{cccccccccc}
16 & & 8 & & 4 & & 2 & & 1 & \\
\hline
16 & + & 8 & + & 0 & + & 0 & + & 1 & = 25
\end{array}$$

Binary Addition

Rules of Addition

There are four important rules we need for binary addition:

$$\boxed{\begin{array}{r} 0 \\ +0 \\ \hline 0 \end{array}} \qquad \boxed{\begin{array}{r} 0 \\ +1 \\ \hline 1 \end{array}} \qquad \boxed{\begin{array}{r} 1 \\ +0 \\ \hline 1 \end{array}} \qquad \boxed{\begin{array}{r} 1 \\ +1 \\ \hline 10 \end{array}}$$

Binary Addition

Addition Example: $11 + 14 = 25$

Binary addition proceeds much like base-10 addition:

- Proceeds from right to left (least significant bit to most significant bit)
- Has a carry bit

$$\begin{array}{r} \\ 1011 \\ +1110 \\ \hline 1 \end{array} \qquad \begin{array}{r} \\ 1011 \\ +1110 \\ \hline 01 \end{array} \qquad \begin{array}{r} 1 \\ 1011 \\ +1110 \\ \hline 001 \end{array} \qquad \begin{array}{r} 11 \\ 1011 \\ +1110 \\ \hline 11001 \end{array}$$

Fractions in Binary

Conversion

$$\boxed{110 \underbrace{\quad . \quad}_{*} 011}$$

* = radix point

- The “Integer” portion of binary is to the left of the radix point.
- Digits to the right represent the fractional part, and are interpreted with fractional values:

$$\begin{array}{ccccccccccccc} & 1 & & 1 & & 0 & . & 0 & & 1 & & 1 & \\ \times & 2^2 & & 2^1 & & 2^0 & & 2^{-1} & & 2^{-2} & & 2^{-3} & \\ \hline & 4 & + & 2 & + & 0 & + & 0 & + & \frac{1}{4} & + & \frac{1}{8} & = 6\frac{3}{8} \end{array}$$

Representing Information as Bit Patterns

The Binary System

Hexadecimal System

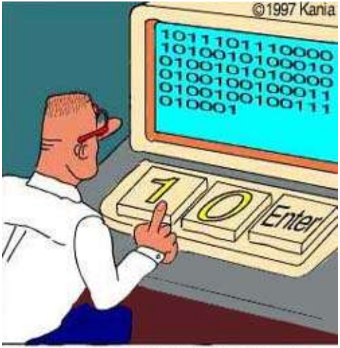
Storing Integers

Storing Fractions

Hexadecimal System

Computer representation of data often deals with long strings of bits...

- Binary representation is not very readable.
- To minimize the number of symbols, binary streams are often represented in *Hexadecimal*.



Real programmers code in binary.

Hexadecimal System

Hexadecimal Notation

Hexadecimal (**hex**) is a `base-16` number system → 16 symbols:

$$\{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$$

Conversion from binary to hex is simple:

- Split binary string into 4 bit segments (Starting from *least* significant bit.)
- Find base-10 value of each segment
- Replace base-10 value with hex equivalent.

Hexadecimal System

Hexadecimal Conversion

Binary	↔	Decimal	↔	Hex
0000	↔	0	↔	0
0001	↔	1	↔	1
0010	↔	2	↔	2
0011	↔	3	↔	3
0100	↔	4	↔	4
0101	↔	5	↔	5
0110	↔	6	↔	6
0111	↔	7	↔	7
1000	↔	8	↔	8
1001	↔	9	↔	9
1010	↔	10	↔	A
1011	↔	11	↔	B
1100	↔	12	↔	C
1101	↔	13	↔	D
1110	↔	14	↔	E
1111	↔	15	↔	F

Binary to Hex:

$$\underbrace{\underbrace{0010}_{2}}_{2}\underbrace{\underbrace{1001}_{9}}_{9}\underbrace{\underbrace{1111}_{15}}_{F}\underbrace{\underbrace{0110}_{6}}_{6}\underbrace{\underbrace{1101}_{13}}_{D}\underbrace{\underbrace{0000}_{0}}_{0}\underbrace{\underbrace{1011}_{11}}_{B}$$

Hex to Binary:

$$\underbrace{\underbrace{5}_{5}}_{0101}\underbrace{\underbrace{A}_{10}}_{1010}\underbrace{\underbrace{6}_{6}}_{0110}\underbrace{\underbrace{E}_{14}}_{1110}\underbrace{\underbrace{C}_{12}}_{1100}\underbrace{\underbrace{1}_{1}}_{0001}\underbrace{\underbrace{3}_{3}}_{0011}$$

Representing Information as Bit Patterns

The Binary System

Hexadecimal System

Storing Integers

Storing Fractions

Storing Numbers

- Storing Integers
    - Two's Complement
    - Excess Notation
- Storing Fractions
    - Floating Point Notation
    - Truncation Error

Storing Integers

To this point, we have only been concerned with *positive*, or unsigned, binary integers.

How can we make negative numbers?

- In Decimal:

$$15 \xrightarrow{\text{voila!}} \boxed{-}15$$

  The minus sign can denote a negative value.
- In binary, however, we do not have a '$-$' symbol...

Two's Complement

Two's Complement is one solution to this problem.

- Interpret the *most significant bit* as a positive or negative:

  $\boxed{1}$ 00101100010110

  This bit is called the **sign bit**,
  - '1' denotes negative,
  - '0' denotes positive.
- 10110 is negative, 010110 is positive.

Are we done?

Two's Complement

It is a little more complex than that. The actual method is:

Definition

Start at the least significant bit (right) and copy down each bit until you reach the first '1'. Copy down that '1' as well.
For the rest of the binary number, copy down the complement of each bit for the remainder of the bit pattern.

The **complement** of a bit pattern is:

Bit	Complement
1	0
0	1

Two's Complement

The procedure for **negating**

Bit pattern	Value represented
0111	7
0110	6
0101	5
0100	4
0011	3
0010	2
0001	1
0000	0
1111	-1
1110	-2
1101	-3
1100	-4
1011	-5
1010	-6
1001	-7
1000	-8

Example: 0110→1010 $(6 \rightarrow -6)$

Original	Action	Comp?	2's Comp.
0110	0: Copy.	N	0
0110	1: *First* 1, copy.	N→Y	10
0110	1: Complement.	Y	010
0110	0: Complement.	Y	1010

Example: 0101→1011, $(5 \rightarrow -5)$

Original	Action	Comp?	2's Comp.
0111	1: *First* 1, copy	N→Y	1
0101	0: Complement.	Y	11
0101	1: Complement.	Y	011
0101	0: Complement.	Y	1011

Two's Complement

Conversion from `base-10`

There are two cases:

1. `base-10` number ≥ 0:
    - Convert to unsigned binary

$$14 = 01110$$

☐

2. `base-10` number < 0:
    - Convert to unsigned binary

$$-14 = -01110$$

    - Take 2's complement

$$01110 \xrightarrow{2's\ comp.} 10010$$

☐

Two's Complement

Conversion to base-10

Again, two cases:

1. 2's complement number most significant bit = 0:
   - Convert to base-10 as done for unsigned binary

$$01101 = 13$$

☐

2. 2's complement number most significant bit = 1:
   - Take 2's complement

$$10011 = 01101$$

   - Convert to base-10 as done for unsigned binary

$$01101 = 13$$

   - Place a '−' to the front of the base-10 number

$$-13$$

☐

Two's Complement

Two's Complement vs. Unsigned Binary Notation

Given a bit pattern with n bits:

	Unsigned Binary	2's Complement
Number of integers represented	2^n	2^n
Range of integers represented	$[0, 2^n - 1]$	$[-2^{n-1}, 2^{n-1} - 1]$

Two's Complement

Addition in Two's Complement

Addition in 2's complement is very similar to Binary addition, except that all bit patterns (even the answer) must be the same length:

$$\boxed{\begin{array}{r} \\ 1011 \\ +1110 \\ \hline 1 \end{array}} \qquad \boxed{\begin{array}{r} \\ 1011 \\ +1110 \\ \hline 01 \end{array}} \qquad \boxed{\begin{array}{r} 1 \\ 1011 \\ +1110 \\ \hline 001 \end{array}} \qquad \boxed{\begin{array}{r} 11 \\ 1011 \\ +1110 \\ \hline \cancel{1}1001 \end{array}}$$

Any values beyond our range are dropped. Above: $-5 + -2 = -7$!

$$\boxed{\begin{array}{r} \\ 0111 \\ +1011 \\ \hline 0 \end{array}} \qquad \boxed{\begin{array}{r} 1 \\ 0111 \\ +1011 \\ \hline 10 \end{array}} \qquad \boxed{\begin{array}{r} 11 \\ 0111 \\ +1011 \\ \hline 001 \end{array}} \qquad \boxed{\begin{array}{r} 111 \\ 0111 \\ +1011 \\ \hline \cancel{1}0001 \end{array}}$$

$7 - 5 = 2$.

Two's Complement

Overflow – Computers make mistakes.

$$\begin{array}{r} 0111 \\ +0110 \\ \hline 1 \end{array}$$

$$\begin{array}{r} 0111 \\ +0110 \\ \hline 01 \end{array}$$

$$\begin{array}{r} 1 \\ 0111 \\ +0110 \\ \hline 101 \end{array}$$

$$\begin{array}{r} 11 \\ 0111 \\ +0110 \\ \hline 1101 \end{array}$$

$7 + 6 = -3$... What Happened???

- One problem with 2's complement notation is the idea of **overflow**.
  - 2's complement can represent the range $[-2^{n-1}, 2^{n-1} - 1]$. Any addition/subtraction that results in a number other than that range will cause overflow.
- Occurs when adding 2 positive or 2 negative numbers.
- Can be detected by checking the sign bit after an operation.

Fix: Use larger bit patterns.

Excess Notation

Another method for representing integer values is **excess** notation.
All bit patterns must be of the same length.

- Choose a pattern length to be used.
- Write down all the different bit patterns of that length (as done in counting)
- The pattern with a '1' followed by all zeroes is the zero value.

Two's complement vs. Excess: **Sign bits are reversed**.

Excess-8 notation:

Bit pattern	Value represented
1111	7
1110	6
1101	5
1100	4
1011	3
1010	2
1001	1
1000	0
0111	-1
0110	-2
0101	-3
0100	-4
0011	-5
0010	-6
0001	-7
0000	-8

Binary value − decimal value = 8.

Excess Notation

Excess-C

- Excess-2 → 2 bits
- Excess-4 → 3 bits
- Excess-8 → 4 bits
- Excess-16 → 5 bits
- Excess-32 → 6 bits
- and so on...

Excess-C 0 value	Binary value	Difference
10	2	2
100	4	4
1000	8	8
10000	16	16
100000	32	32
⋮	⋮	⋮

Rule: The number, ‘C’, is equal to the value of the most significant bit in the excess-C bit pattern!

Excess Notation

Excess-C vs Two's Complement

Given a bit pattern with n bits:

	2's Complement	Excess
Number of integers represented	2^n	2^n
Range of integers represented	$[-2^{n-1}, 2^{n-1} - 1]$	$[-2^{n-1}, 2^{n-1} - 1]$

Excess Notation

Excess-C ⇌ `Base-10`

`base-10` ⟶ Excess-C

1. Choose appropriate length bit pattern
2. Add the value 'C' to the `base-10` number
3. Convert the resulting `base-10` number to unsigned binary.

☐

Excess-C ⟶ `base-10`

1. Convert excess-C number to `base-10`
2. Subtract 'C' from the `base-10` number.

☐

Representing Information as Bit Patterns

The Binary System

Hexadecimal System

Storing Integers

Storing Fractions

Storing Fractions

Overview

Using integers, we only needed to represent a pattern of 1's and 0's. Fractions also need to represent the position of the radix point.

Floating-point notation: Based on *Scientific notation* (i.e. $-4.357 \times 10^{-2} = -0.04357$)

- There are three parts to Scientific notation:
  1. *Sign*: ±
  2. *Mantissa*: Number pattern
  3. *Exponent*: Power

$$\underbrace{-}_{sgn}\ \underbrace{4.357}_{mantissa} \times 10^{\overbrace{-2}^{exp}}$$

Floating Point Notation

Binary Representation

The general format of a floating point number in binary is:

$$\textit{sgn mantissa} \times 2^{\textit{exponent}}$$

Sign: (±)

Exponent:

- Positive **or** negative
- Integer

Mantissa:

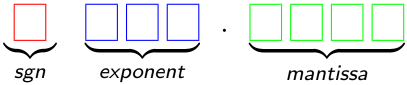
- Positive number
- Where is the radix point?!?

How can we represent each of these parts?

Floating Point Notation

Binary Representation

We will use one byte to store fractions[1]:



The eight bits are represented using:

- Sign: 1 bit, ‘0’ = positive, ‘1’ negative
- Exponent: 3 bits, using excess-4 notation
- Mantissa: 4 bits... Radix point is placed on the left side of the mantissa!

---

[1] Machines often use much larger representations, 32- or 64-bit.

Floating Point Notation

Floating-point → base-10 – 11011101

$$\underbrace{\boxed{1}}_{sgn}\quad \underbrace{\boxed{1}\,\boxed{0}\,\boxed{1}}_{exponent}\quad . \quad \underbrace{\boxed{1}\,\boxed{1}\,\boxed{0}\,\boxed{1}}_{mantissa}$$

Sign bit $= 1 \rightarrow \boxed{-}$

Mantissa $= .1101 \xrightarrow{shift} \boxed{1.101}$

$$\begin{array}{cccccccccc} & 1 & . & 1 & & 0 & & 1 & & \\ \times & 2^{0} & & 2^{-1} & & 2^{-2} & & 2^{-3} & & \\ \hline & 1 & + & \frac{1}{2} & + & 0 & + & \frac{1}{8} & & = 1\frac{5}{8} \end{array}$$

Exp	val	shift
111	3	3
110	2	2
101	1	$\boxed{1}$
100	0	0
011	-1	-1
010	-2	-2
001	-3	-3
000	-4	-4

Finally, adding the sign bit,

$$11011101 = -1\frac{5}{8}$$

Floating-point Notation

base-10 → Floating-point – $2\frac{1}{4}$

$2\frac{1}{4} \xrightarrow{\text{binary}} \boxed{1}0.01$

Copy to mantissa, starting from leftmost 1: ($\boxed{1}0.01$)

$\boxed{}\ \boxed{}\boxed{}\boxed{}\ .\ \boxed{1}\boxed{0}\boxed{0}\boxed{1}$

$.1001 \longrightarrow 10.01$, shift $= +2$.

Excess-4 representation: 110

$\boxed{}\ \boxed{1}\boxed{1}\boxed{0}\ .\ \boxed{1}\boxed{0}\boxed{0}\boxed{1}$

$2\frac{1}{4}$ is non-negative:

$\boxed{0}\ \boxed{1}\boxed{1}\boxed{0}\ .\ \boxed{1}\boxed{0}\boxed{0}\boxed{1}$

$$2\frac{1}{4} = \boxed{0}\ \boxed{1}\boxed{1}\boxed{0}\ .\ \boxed{1}\boxed{0}\boxed{0}\boxed{1}$$

Floating-point Notation

base-10 → Floating-point – $-\frac{1}{8}$

$$\frac{1}{8} \xrightarrow{\text{binary}} 0.001$$

Copy to mantissa, starting from leftmost 1: $(0.00\boxed{1})$

$$\boxed{}\ \boxed{}\boxed{}\boxed{}\ .\ \boxed{1}\boxed{0}\boxed{0}\boxed{0}$$

$0.1000 \longrightarrow 0.001$, shift $= -2$.

Excess-4 representation: 010

$$\boxed{}\ \boxed{0}\boxed{1}\boxed{0}\ .\ \boxed{1}\boxed{0}\boxed{0}\boxed{0}$$

$-\frac{1}{8}$ is negative:

$$\boxed{1}\ \boxed{0}\boxed{1}\boxed{0}\ .\ \boxed{1}\boxed{0}\boxed{0}\boxed{0}$$

$$-\frac{1}{8} = \boxed{1}\ \boxed{0}\boxed{1}\boxed{0}\ .\ \boxed{1}\boxed{0}\boxed{0}\boxed{0}$$

Remember: Leftmost 1, NOT leftmost bit!

Floating-point Notation

Truncation Errors

$2\frac{5}{8} \xrightarrow{\text{binary}} \boxed{1}0.101$

Copy to mantissa, starting from leftmost 1: ($\boxed{1}0.101$)

$\boxed{}\ \boxed{}\,\boxed{}\,\boxed{}\ .\ \boxed{1}\,\boxed{0}\,\boxed{1}\,\boxed{0}\,\boxed{1}$

0.1010 $\longrightarrow$ 10.10, shift $= +2$.

Excess-4 representation: 110

$\boxed{}\ \boxed{1}\,\boxed{1}\,\boxed{0}\ .\ \boxed{1}\,\boxed{0}\,\boxed{1}\,\boxed{0}$

$2\frac{5}{8}$ is non-negative:

$\boxed{0}\ \boxed{1}\,\boxed{1}\,\boxed{0}\ .\ \boxed{1}\,\boxed{0}\,\boxed{1}\,\boxed{0}$

$$2\frac{5}{8} = 2\frac{1}{2} = \boxed{0}\ \boxed{1}\,\boxed{1}\,\boxed{0}\ .\ \boxed{1}\,\boxed{0}\,\boxed{1}\,\boxed{0}$$

Truncation errors or **round-off errors** (gray box) occur when a number is too large to represent!

Floating-point Notation

Order of operations

When adding fractions using floating-point notation, we can avoid truncation errors. For example, adding the three numbers $\left\{\frac{1}{8}, \frac{1}{8}, 2\frac{1}{2}\right\}$:

$$\begin{array}{r} 10.10 \\ +\ 0.001 \\ \hline 10.10\,1 \end{array}$$

$$\begin{array}{r} 10.10 \\ +\ 0.001 \\ \hline 10.10\,1 \end{array}$$

$$\boxed{2\frac{1}{2} + \frac{1}{8} + \frac{1}{8} = 2\frac{1}{2}}$$

$$\begin{array}{r} 0.001 \\ +\ 10.10 \\ \hline 10.10\,1 \end{array}$$

$$\begin{array}{r} 10.10 \\ +\ 0.001 \\ \hline 10.10\,1 \end{array}$$

$$\boxed{\frac{1}{8} + 2\frac{1}{2} + \frac{1}{8} = 2\frac{1}{2}}$$

$$\begin{array}{r} 0.001 \\ +\ 0.001 \\ \hline 0.010 \end{array}$$

$$\begin{array}{r} 0.010 \\ +\ 10.10 \\ \hline 10.11 \end{array}$$

$$\boxed{\frac{1}{8} + \frac{1}{8} + 2\frac{1}{2} = 2\frac{3}{4}}$$

By adding the small numbers first, we can avoid truncation errors!

Representing Information as Bit Patterns

The Binary System

Hexadecimal System

Storing Integers

Storing Fractions

Questions over Chapters 1.4–1.7?