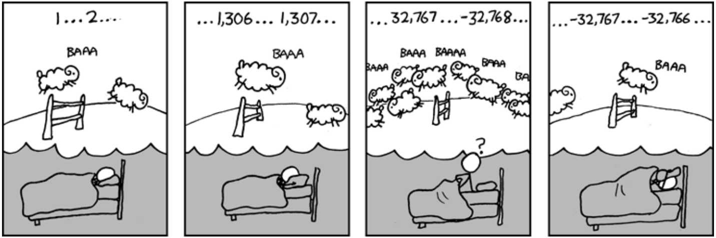
COMS12200 lecture: week #2



http://xkcd.com/571/

- Hopefully you agree that
$$123 \equiv \langle 3, 2, 1 \rangle$$
i.e., the decimal literal 123 is basically just a sequence of digits.
- The same is true elsewhere, e.g.,
  - a **bit** is a single binary digit, i.e., 0 or 1,
  - a **byte** is an 8-element sequence of bits, and
  - a **word** is a w-element sequence of bits

  and hence
$$01111011 \equiv \langle 1, 1, 0, 1, 1, 1, 1, 0 \rangle.$$
- Question: what do these things *mean* ... what do they *represent*?
- Answer: anything *we* decide they do!

- There's just *one* key concept here, namely

$$\underbrace{\hat{X}}_{\text{the representation of } X} \quad \underbrace{\mapsto}_{\text{maps to}} \quad \underbrace{X}_{\text{the value of } X}$$

- That is, we need
  1. a concrete representation that we can write down, and
  2. a mapping that means the right thing wrt. value, *plus* is consistent (in both directions).

An Aside: Properties of bit-sequences

- Although we can write

$$X = 1111011 \equiv \langle 1, 1, 0, 1, 1, 1, 1 \rangle,$$

there are actually *two* ways to interpret the same literal:

1. A **little-endian** ordering is where we read bits in a literal from right-to-left, i.e.,

$$X_{LE} = \langle X_0, X_1, X_2, X_3, X_4, X_5, X_6 \rangle = \langle 1, 1, 0, 1, 1, 1, 1 \rangle.$$

   where

   - the Least-Significant Bit (LSB) is the right-most in the literal (i.e., X_0), and
   - the Most-Significant Bit (MSB) is the left-most in the literal (i.e., $X_{n-1} = X_6$).

2. A **big-endian** ordering is where we read bits in a literal from left-to-right, i.e.,

$$X_{BE} = \langle X_6, X_5, X_4, X_3, X_2, X_1, X_0 \rangle = \langle 1, 1, 1, 1, 0, 1, 1 \rangle.$$

   Here,

   - the Least-Significant Bit (LSB) is the left-most in the literal (i.e., $X_{n-1} = X_6$), and
   - the Most-Significant Bit (MSB) is the right-most in the literal (i.e., X_0).

An Aside: Properties of bit-sequences

- Given an n-element bit-sequence X, and an m-element bit-sequence Y we can
  1. overload $\oslash \in \{\neg\}$, i.e., write
  $$R = \oslash X,$$
  to mean
  $$R_i = \oslash X_i$$
  for $0 \leq i < n$, and
  2. overload $\ominus \in \{\wedge, \vee, \oplus\}$, i.e., write
  $$R = X \ominus Y,$$
  to mean
  $$R_i = X_i \ominus Y_i$$
  for $0 \leq i < n = m$, where if $n \neq m$ we pad either X or Y with 0 until $n = m$.
- Example: in C, we use the **bit-wise** operators ~, &, | and ^ for exactly this: they apply NOT, AND, OR and XOR to corresponding bits in the operands.

An Aside: Properties of bit-sequences

- Given two n-element bit-sequences X and Y, we can define the following:
  1. The **Hamming weight** of X is the number of bits in X that are equal to 1, i.e., the number of times $X_i = 1$:

     $$\mathcal{H}(X) = \sum_{i=0}^{n-1} X_i.$$

  2. The **Hamming distance** between X and Y is the number of bits in X that differ from the corresponding bit in Y, i.e., the number of times $X_i \neq Y_i$:

     $$\mathcal{D}(X,Y) = \sum_{i=0}^{n-1} X_i \oplus Y_i.$$

Positional Number Systems (1)

- You *already* use **positional number systems** without thinking about it ...
- ... the idea is to express the value of a number x using a base-b expansion

$$\begin{aligned} \hat{x} &= \langle x_0, x_1, \ldots, x_{n-1} \rangle \\ &\mapsto \pm \sum_{i=0}^{n-1} x_i \cdot b^i \end{aligned}$$

  where each x_i

  - is one of n digits taken from the digit set $X = \{0, \ldots, b-1\}$,
  - and which is "weighted" by some power of of the base b.

Positional Number Systems (3)

Example

Consider an example where we

1. set $b = 10$, i.e., deal with **decimal** numbers, and
2. have $x_i \in X = \{0, 1, \ldots, 10 - 1 = 9\}$.

This means we can write

$$
\begin{array}{lcl}
\hat{x} = 123 & = & \langle 3, 2, 1 \rangle_{(10)} \\
 & \mapsto & \sum_{i=0}^{n-1} x_i \cdot 10^i \\
 & \mapsto & 3 \cdot 10^0 + 2 \cdot 10^1 + 1 \cdot 10^2 \\
 & \mapsto & 3 \cdot 1 \quad + 2 \cdot 10 \quad + 1 \cdot 100 \\
 & \mapsto & 123_{(10)}
\end{array}
$$

i.e., represent the value "one hundred and twenty three" in a variety of ways using different bases.

Positional Number Systems (3)

Example

Consider an example where we

1. set $b = 2$, i.e., deal with **binary** numbers, and
2. have $x_i \in X = \{0, 2-1 = 1\}$.

This means we can write

$$\begin{array}{lcl}
\hat{x} = 1111011 & = & \langle 1,1,0,1,1,1,1 \rangle_{(2)} \\
 & \mapsto & \sum_{i=0}^{n-1} x_i \cdot 2^i \\
 & \mapsto & 1 \cdot 2^0 + 1 \cdot 2^1 + 0 \cdot 2^2 + 1 \cdot 2^3 + 1 \cdot 2^4 + 1 \cdot 2^5 + 1 \cdot 2^6 \\
 & \mapsto & 1 \cdot 1 + 1 \cdot 2 + 0 \cdot 4 + 1 \cdot 8 + 1 \cdot 16 + 1 \cdot 32 + 1 \cdot 64 \\
 & \mapsto & 123_{(10)}
\end{array}$$

i.e., represent the value "one hundred and twenty three" in a variety of ways using different bases.

Positional Number Systems (3)

Example

Consider an example where we

1. set $b = 8$, i.e., deal with **octal** numbers, and
2. have $x_i \in X = \{0, 1, \ldots, 8 - 1 = 7\}$.

This means we can write

$$\begin{array}{lcl}
\hat{x} = 173 & = & \langle 3, 7, 1 \rangle_{(8)} \\
 & \mapsto & \sum_{i=0}^{n-1} x_i \cdot 8^i \\
 & \mapsto & 3 \cdot 8^0 + 7 \cdot 8^1 + 1 \cdot 8^2 \\
 & \mapsto & 3 \cdot 8 + 7 \cdot 64 + 1 \cdot 512 \\
 & \mapsto & 123_{(10)}
\end{array}$$

i.e., represent the value “one hundred and twenty three” in a variety of ways using different bases.

Positional Number Systems (3)

Example

Consider an example where we

1. set $b = 16$, i.e., deal with **hexadecimal** numbers, and
2. have $x_i \in X = \{0, 1, \ldots, 16 - 1 = 15\}$.

This means we can write

$$
\begin{array}{lcl}
\hat{x} = 7\text{B} & = & \langle B, 7 \rangle_{(16)} \\
 & \mapsto & \sum_{i=0}^{n-1} x_i \cdot 16^i \\
 & \mapsto & 11 \cdot 16^0 + 7 \cdot 16^1 \\
 & \mapsto & 11 \cdot 1 \quad + 7 \cdot 16 \\
 & \mapsto & 123_{(10)}
\end{array}
$$

i.e., represent the value "one hundred and twenty three" in a variety of ways using different bases.

Positional Number Systems (4)

- Fact:
  - each hexadecimal digit $x_i \in \{0, 1, \ldots 15\}$,
  - four bits gives $2^4 = 16$ possible combinations, so
  - each hexadecimal digit can be thought of as a short-hand for four binary digits.
- Example: we can perform the following translation steps

$$
\begin{array}{rcllllllll}
2223 & = & 1 \cdot 1 & + 1 \cdot 2 & + 1 \cdot 4 & + 1 \cdot 8 & + 0 \cdot 16 & + 1 \cdot 32 & + \\
 & & 0 \cdot 64 & + 1 \cdot 128 & + 0 \cdot 256 & + 0 \cdot 512 & + 0 \cdot 1024 & + 1 \cdot 2048 & \\
 & = & 1 \cdot 2^0 & + 1 \cdot 2^1 & + 1 \cdot 2^2 & + 1 \cdot 2^3 & + 0 \cdot 2^4 & + 1 \cdot 2^5 & + \\
 & & 0 \cdot 2^6 & + 1 \cdot 2^7 & + 0 \cdot 2^8 & + 0 \cdot 2^9 & + 0 \cdot 2^{10} & + 1 \cdot 2^{11} & \\
 & = & \multicolumn{6}{l}{\langle 1, 1, 1, 1, 0, 1, 0, 1, 0, 0, 0, 1 \rangle_{(2)}} \\
 & = & \multicolumn{6}{l}{\langle \langle 1, 1, 1, 1 \rangle_{(2)}, \langle 0, 1, 0, 1 \rangle_{(2)}, \langle 0, 0, 0, 1 \rangle_{(2)} \rangle_{(16)}} \\
 & = & \multicolumn{6}{l}{\langle 15_{(10)}, 10_{(10)}, 8_{(10)} \rangle_{(16)}} \\
 & = & \multicolumn{6}{l}{\langle F_{(16)}, A_{(16)}, 8_{(16)} \rangle_{(16)}} \\
 & = & \multicolumn{6}{l}{\langle F, A, 8 \rangle_{(16)}} \\
 & = & 15 \cdot 16^0 & + 10 \cdot 16^1 & + 8 \cdot 16^2 & & & \\
 & = & 15 \cdot 1 & + 10 \cdot 16 & + 8 \cdot 256 & & & \\
 & = & 2223 & & & & &
\end{array}
$$

so, for instance, if we write the literal `0x8AF` in C it has the same value as `2223`.

Positional Number Systems (5)

- Fact: left-shift (resp. right-shift) of some x by y digits is the same as multiplication (resp. division) by b^y.
- Example: taking $b = 2$ we find that

$$
\begin{array}{rcl}
x \cdot 2^y &=& \left(\sum_{i=0}^{n-1} x_i \cdot 2^i \right) \cdot 2^y \\
&=& \sum_{i=0}^{n-1} x_i \cdot 2^i \cdot 2^y \\
&=& \sum_{i=0}^{n-1} x_i \cdot 2^{i+y} \\
&=& x \ll y
\end{array}
$$

and

$$
\begin{array}{rcl}
x \,/\, 2^y &=& \left(\sum_{i=0}^{n-1} x_i \cdot 2^i \right) / \, 2^y \\
&=& \sum_{i=0}^{n-1} x_i \cdot 2^i / 2^y \\
&=& \sum_{i=0}^{n-1} x_i \cdot 2^{i-y} \\
&=& x \gg y
\end{array}
$$

so in C,

1. `0x8AF << 2` = `0x22BC` $\mapsto 8892_{(10)} = 2223_{(10)} \cdot 2^2 = 2223_{(10)} \cdot 4$, and
2. `0x8AF >> 2` = `0x022B` $\mapsto 555_{(10)} = 2223_{(10)} / 2^2 = 2223_{(10)} / 4$.

$\mathbb{Z}$ (1)

- Problem: we'd like to represent and perform various operations on elements of $\mathbb{Z}$, but it's an an infinite set!
- Solution: we approximate, in C for example we get

$$\begin{array}{lclcl}
\texttt{unsigned char} & \mapsto & \mathbb{Z}_{\texttt{unsigned char}} & = & \{ \quad 0, \ldots, +2^{8} - 1 \} \\
\texttt{unsigned int} & \mapsto & \mathbb{Z}_{\texttt{unsigned int}} & = & \{ \quad 0, \ldots, +2^{32} - 1 \} \\
\texttt{char} & \mapsto & \mathbb{Z}_{\texttt{char}} & = & \{ -2^{7}, \ldots, 0, \ldots, +2^{7} - 1 \} \\
\texttt{int} & \mapsto & \mathbb{Z}_{\texttt{int}} & = & \{ -2^{31}, \ldots, 0, \ldots, +2^{31} - 1 \}
\end{array}$$

but why *these*, and how do they work?

$\mathbb{Z}$ (2)

Definition

An unsigned integer can be represented in n bits by using the natural binary expansion. That is, we have

$$\begin{array}{rcl} \hat{x} & = & \langle x_0, x_1, \ldots, x_{n-1} \rangle \\ & \mapsto & \sum_{i=0}^{n-1} x_i \cdot 2^i \end{array}$$

for $x_i \in \{0, 1\}$, and

$$0 \leq x \leq 2^n - 1.$$

$\mathbb{Z}$ (3) – Sign-magnitude

Definition

A signed integer can be represented in n bits by using the **sign-magnitude** approach; 1 bit is reserved for the sign (0 means positive, 1 means negative) and $n - 1$ for the magnitude. That is, we have

$$\begin{array}{rcl} \hat{x} & = & \langle x_0, x_1, \ldots, x_{n-1} \rangle \\ & \mapsto & -1^{x_{n-1}} \cdot \sum_{i=0}^{n-2} x_i \cdot 2^i \end{array}$$

for $x_i \in \{0, 1\}$, and

$$-2^{n-1} - 1 \leq x \leq +2^{n-1} - 1.$$

Note there are two representations of zero (i.e., $+0$ and -0).

$\mathbb{Z}$ (4) – Sign-magnitude

Example

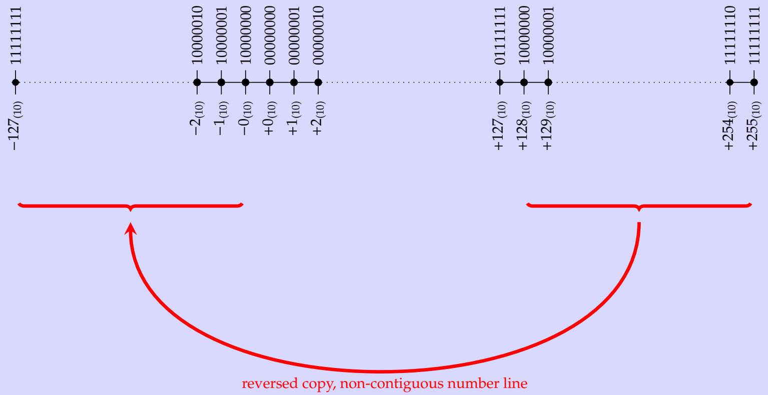
If $n = 8$ for example, we can represent values in the range $-127 \ldots +127$; selected cases are as follows:

$$
\begin{array}{lcccclllllllcl}
01111111 & \mapsto & -1^0 & \cdot & (& 1 \cdot 2^6 & + 1 \cdot 2^5 & + 1 \cdot 2^4 & + 1 \cdot 2^3 & + 1 \cdot 2^2 & + 1 \cdot 2^1 & + 1 \cdot 2^0 &) & = & +127_{(10)} \\
\vdots & & & & & & & & & & & & & \vdots & \\
01111011 & \mapsto & -1^0 & \cdot & (& 1 \cdot 2^6 & + 1 \cdot 2^5 & + 1 \cdot 2^4 & + 1 \cdot 2^3 & + 0 \cdot 2^2 & + 1 \cdot 2^1 & + 1 \cdot 2^0 &) & = & +123_{(10)} \\
\vdots & & & & & & & & & & & & & \vdots & \\
00000001 & \mapsto & -1^0 & \cdot & (& 0 \cdot 2^6 & + 0 \cdot 2^5 & + 0 \cdot 2^4 & + 0 \cdot 2^3 & + 0 \cdot 2^2 & + 0 \cdot 2^1 & + 1 \cdot 2^0 &) & = & +1_{(10)} \\
00000000 & \mapsto & -1^0 & \cdot & (& 0 \cdot 2^6 & + 0 \cdot 2^5 & + 0 \cdot 2^4 & + 0 \cdot 2^3 & + 0 \cdot 2^2 & + 0 \cdot 2^1 & + 0 \cdot 2^0 &) & = & +0_{(10)} \\
10000000 & \mapsto & -1^1 & \cdot & (& 0 \cdot 2^6 & + 0 \cdot 2^5 & + 0 \cdot 2^4 & + 0 \cdot 2^3 & + 0 \cdot 2^2 & + 0 \cdot 2^1 & + 0 \cdot 2^0 &) & = & -0_{(10)} \\
10000001 & \mapsto & -1^1 & \cdot & (& 0 \cdot 2^6 & + 0 \cdot 2^5 & + 0 \cdot 2^4 & + 0 \cdot 2^3 & + 0 \cdot 2^2 & + 0 \cdot 2^1 & + 1 \cdot 2^0 &) & = & -1_{(10)} \\
\vdots & & & & & & & & & & & & & \vdots & \\
11111011 & \mapsto & -1^1 & \cdot & (& 1 \cdot 2^6 & + 1 \cdot 2^5 & + 1 \cdot 2^4 & + 1 \cdot 2^3 & + 0 \cdot 2^2 & + 1 \cdot 2^1 & + 1 \cdot 2^0 &) & = & -123_{(10)} \\
\vdots & & & & & & & & & & & & & \vdots & \\
11111111 & \mapsto & -1^1 & \cdot & (& 1 \cdot 2^6 & + 1 \cdot 2^5 & + 1 \cdot 2^4 & + 1 \cdot 2^3 & + 1 \cdot 2^2 & + 1 \cdot 2^1 & + 1 \cdot 2^0 &) & = & -127_{(10)}
\end{array}
$$

$\mathbb{Z}$ (5) – Sign-magnitude

Example

For $n = 8$, consider the following number line:



$\mathbb{Z}$ (6) – Two's-Complement

Definition

A signed integer can be represented in n bits by using the **two's-complement** approach. The basic idea is to weight bit $n-1$ using -2^{n-1} rather than $+2^{n-1}$, and all other bits as normal. That is, we have

$$\begin{array}{rcl} \hat{x} & = & \langle x_0, x_1, \ldots, x_{n-1} \rangle \\ & \mapsto & x_{n-1} \cdot -2^{n-1} + \sum_{i=0}^{n-2} x_i \cdot 2^i \end{array}$$

for $x_i \in \{0, 1\}$, and

$$-2^{n-1} \le x \le +2^{n-1} - 1.$$

$\mathbb{Z}$ (7) – Two's-Complement

Example

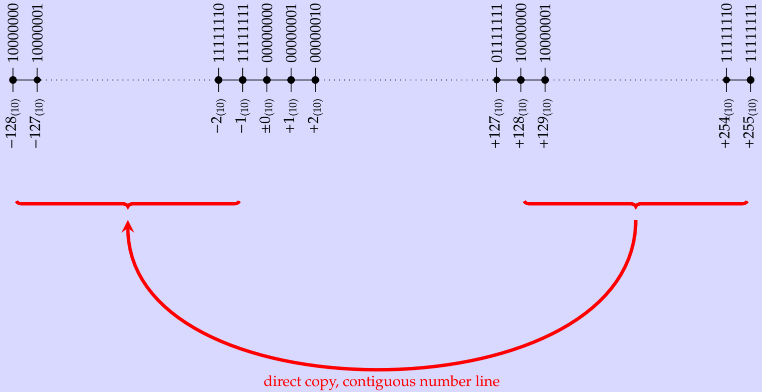
If $n = 8$ for example, we can represent values in the range $-128 \ldots +127$; selected cases are as follows:

$$
\begin{array}{lclcr}
01111111 & \mapsto & 0 \cdot -2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 & = & +127_{(10)} \\
 & \vdots & & \vdots & \\
01111011 & \mapsto & 0 \cdot -2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 & = & +123_{(10)} \\
 & \vdots & & \vdots & \\
00000001 & \mapsto & 0 \cdot -2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 & = & +1_{(10)} \\
00000000 & \mapsto & 0 \cdot -2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 & = & +0_{(10)} \\
11111111 & \mapsto & 1 \cdot -2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 & = & -1_{(10)} \\
 & \vdots & & \vdots & \\
10000101 & \mapsto & 1 \cdot -2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 & = & -123_{(10)} \\
 & \vdots & & \vdots & \\
10000000 & \mapsto & 1 \cdot -2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 & = & -128_{(10)}
\end{array}
$$

$\mathbb{Z}$ (8) – Two's-Complement

Example

For $n = 8$, consider the following number line:



Conclusions

- Take away points:
  1. *We* control what bit-sequences mean ...
  2. ... a representation of some X *isn't* the same thing as the value of X: *we* decide how one maps to the other so without knowing *how* X is represented, it has no sane meaning.
  3. For example, we can view the C `int` data-type as mapping to
     3.1 a signed 32-bit integer, or
     3.2 a generic object which can take one of 2^{32} states.
  4. With the second mind-set, *we* assign meaning to each bit or state; as a result we can represent *anything*, e.g.,
     - an RBG-based pixel within an image,
     - an ASCII character within a text file, or
     - a network IP address.

References and Further Reading

[1] Wikipedia: Bit manipulation.
http://en.wikipedia.org/wiki/Bit_manipulation.

[2] Wikipedia: Fixed-point arithmetic.
http://en.wikipedia.org/wiki/Fixed-point_arithmetic.

[3] Wikipedia: Floating-point arithmetic.
http://en.wikipedia.org/wiki/Floating_point.

[4] Wikipedia: One's-complement.
http://en.wikipedia.org/wiki/One's_complement.

[5] Wikipedia: Two's-complement.
http://en.wikipedia.org/wiki/Two's_complement.

[6] D. Goldberg.
What every computer scientist should know about floating-point arithmetic.
ACM Computing Surveys, 23(1):5–48, 1991.

[7] D. Page.
Chapter 1: Mathematical preliminaries.
In *A Practical Introduction to Computer Architecture*. Springer-Verlag, 1st edition, 2009.

[8] B. Parhami.
Part 1: Number representation.
In *Computer Arithmetic: Algorithms and Hardware Designs*. Oxford University Press, 1st edition, 2000.

References and Further Reading

[9] A.S. Tanenbaum.
Appendix A: Binary numbers.
In *Structured Computer Organisation* [11].

[10] A.S. Tanenbaum.
Appendix A: Floating-point numbers.
In *Structured Computer Organisation* [11].

[11] A.S. Tanenbaum.
Structured Computer Organisation.
Prentice-Hall, 6th edition, 2012.