Intro. to Computer Architecture

Daniel Page

Department of Computer Science,
University Of Bristol,
Merchant Venturers Building,
Woodland Road,
Bristol, BS8 1UB. UK.
⟨csdsp@bristol.ac.uk⟩

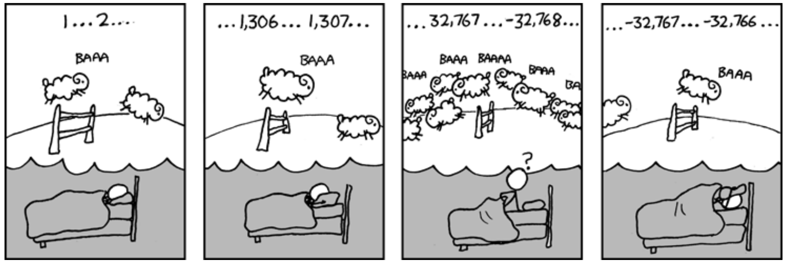
January 9, 2018

Keep in mind there are *two* PDFs available (of which this is the latter):

1. a PDF of examinable material used as lecture slides, and
2. a PDF of non-examinable, extra material:
   - the associated notes page may be pre-populated with extra, written explaination of material covered in lecture(s), plus
   - anything with a "grey'ed out" header/footer represents extra material which is useful and/or interesting but out of scope (and hence not covered).

Notes:

Notes:



http://xkcd.com/571/

Notes:

- 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!

Notes:

- 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)

  noting multiple representations may be able to capture the same (set of) values.

Notes:

An Aside: Properties of bit-sequences

Definition

A given literal, say

$$X = 1111011,$$

can be interpreted in *two* ways:

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.,

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

   where

   - 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).

Notes:

An Aside: Properties of bit-sequences

Definition

Following the idea of vectorial Boolean function, given an n-element bit-sequence X, and an m-element bit-sequence Y we can clearly

1. overload $\oslash \in \{\neg\}$, i.e., write
$$R = \oslash X,$$
to mean
$$R_i = \oslash X_i$$
for $0 \leq i < n$,
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 the $n = m$.

Notes:

- Although they *look* similar, take care not to confuse the bit-wise operators with the Boolean operators `!`, `&&` and `||`. It's reasonable to think of the former as being used for *computation* and the latter for *conditions* (i.e., when a decision is needed).

An Aside: Properties of bit-sequences

Definition

Following the idea of vectorial Boolean function, given an n-element bit-sequence X, and an m-element bit-sequence Y we can clearly

1. overload $\oslash \in \{\neg\}$, i.e., write
$$R = \oslash X,$$
to mean
$$R_i = \oslash X_i$$
for $0 \leq i < n$,
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 the $n = m$.

- Example: in C, we use the computational (or **bit-wise**) operators `~`, `&`, `|`, and `^` this way: they apply NOT, AND, OR, and XOR to corresponding bits in the operands.

Notes:

- Although they *look* similar, take care not to confuse the bit-wise operators with the Boolean operators `!`, `&&` and `||`. It's reasonable to think of the former as being used for *computation* and the latter for *conditions* (i.e., when a decision is needed).

Definition

Given two n-bit sequences X and Y, we can define some important properties named after Richard Hamming, a researcher at Bell Labs:

- The **Hamming weight** of X is the number of bits in X that are equal to 1, i.e., the number of times $X_i = 1$. This can be expressed as

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

- 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 $X_i \neq Y_i$. This can be expressed as

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

Note that both quantities naturally generalise to non-binary sequences.

Notes:

An Aside: Properties of bit-sequences

Definition

Given two n-bit sequences X and Y, we can define some important properties named after Richard Hamming, a researcher at Bell Labs:

- The **Hamming weight** of X is the number of bits in X that are equal to 1, i.e., the number of times $X_i = 1$. This can be expressed as

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

- 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 $X_i \neq Y_i$. This can be expressed as

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

Note that both quantities naturally generalise to non-binary sequences.

- Example: given $X = \langle 1, 0, 0, 1 \rangle$ and $Y = \langle 0, 1, 1, 1 \rangle$ we find that

$$\mathcal{H}(X) = \sum_{i=0}^{n-1} X_i = 1 + 0 + 0 + 1 = 2$$

$$\mathcal{D}(X, Y) = \sum_{i=0}^{n-1} X_i \oplus Y_i = (1 \oplus 0) + (0 \oplus 1) + (0 \oplus 1) + (1 \oplus 1) = 1 + 1 + 1 + 0 = 3$$

Notes:

Positional Number Systems (1)

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

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

  where each x_i

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

Notes:

Positional Number Systems (2)

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.

Notes:

- For values of $b > 10$ we run into a problem: values such as 11 aren't a single digit, so it becomes awkward to say $x_i = 11$. To combat this, in hexadecimal or $b = 16$ for example, we use letters. This means it's important to remember

$$\begin{array}{ccc} A & \mapsto & 10 \\ B & \mapsto & 11 \\ C & \mapsto & 12 \\ D & \mapsto & 13 \\ E & \mapsto & 14 \\ F & \mapsto & 15 \end{array}$$

Positional Number Systems (2)

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.

Notes:

- For values of $b > 10$ we run into a problem: values such as 11 aren't a single digit, so it becomes awkward to say $x_i = 11$. To combat this, in hexadecimal or $b = 16$ for example, we use letters. This means it's important to remember

$$
\begin{array}{lcl}
A & \mapsto & 10 \\
B & \mapsto & 11 \\
C & \mapsto & 12 \\
D & \mapsto & 13 \\
E & \mapsto & 14 \\
F & \mapsto & 15
\end{array}
$$

Positional Number Systems (2)

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

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

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

Notes:

- For values of $b > 10$ we run into a problem: values such as 11 aren't a single digit, so it becomes awkward to say $x_i = 11$. To combat this, in hexadecimal or $b = 16$ for example, we use letters. This means it's important to remember

$$
\begin{array}{lcl}
A & \mapsto & 10 \\
B & \mapsto & 11 \\
C & \mapsto & 12 \\
D & \mapsto & 13 \\
E & \mapsto & 14 \\
F & \mapsto & 15
\end{array}
$$

Positional Number Systems (2)

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

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

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

Notes:

- For values of $b > 10$ we run into a problem: values such as 11 aren't a single digit, so it becomes awkward to say $x_i = 11$. To combat this, in hexadecimal or $b = 16$ for example, we use letters. This means it's important to remember

$$\begin{array}{lcl} A & \mapsto & 10 \\ B & \mapsto & 11 \\ C & \mapsto & 12 \\ D & \mapsto & 13 \\ E & \mapsto & 14 \\ F & \mapsto & 15 \end{array}$$

Positional Number Systems (3)

- **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

$$\begin{array}{lcl} 8AF & = & \langle F, A, 8 \rangle_{(16)} \\ & = & \langle \langle 1,1,1,1 \rangle_{(2)}, \langle 0,1,0,1 \rangle_{(2)}, \langle 0,0,0,1 \rangle_{(2)} \rangle_{(16)} \\ & = & \langle 1,1,1,1,0,1,0,1,0,0,0,1 \rangle_{(2)} \\ & \mapsto & 2223_{(10)} \end{array}$$

st. in C, for example,

$$\texttt{0x8AF} = 2223.$$

Notes:

Positional Number Systems (4)

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

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

st. in C, for example,

$$\texttt{0x8AF << 2} \mapsto 2223_{(10)} \cdot 2^2 = 8892_{(10)} \mapsto \texttt{0x22BC}.$$

Notes:

- For completeness, note that

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

st. in C, for example,

$$\texttt{0x8AF >> 2} \mapsto 2223_{(10)}/2^2 = 555_{(10)} \mapsto \texttt{0x022B}.$$

Positional Number Systems (5)

Example

Fractional components can be accommodated by tweaking the range of i in our original definition; we imagine there are m digits in the fractional component. 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

$$\begin{array}{rcl} \hat{x} = 123.3125 & = & \langle 5, 2, 1, 3, 3, 2, 1 \rangle \\ & \mapsto & \sum_{i=-m}^{n-m-1} x_i \cdot b^i \\ & \mapsto & 5 \cdot 10^{-4} + 2 \cdot 10^{-3} + 1 \cdot 10^{-2} + 3 \cdot 10^{-1} + \\ & & 3 \cdot 10^0 + 2 \cdot 10^1 + 1 \cdot 10^2 \\ & \mapsto & 5 \cdot 1/10000 + 2 \cdot 1/1000 + 1 \cdot 1/100 + 3 \cdot 1/10 + \\ & & 3 \cdot 1 + 2 \cdot 10 + 1 \cdot 100 \\ & \mapsto & 123.3125_{(10)} \end{array}$$

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

Notes:

Example

Fractional components can be accommodated by tweaking the range of i in our original definition; we imagine there are m digits in the fractional component. 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

$$\begin{array}{lcl}
\hat{x} = 1111011.0101 & = & \langle 1,0,1,0,1,1,0,1,1,1,1 \rangle \\
 & \mapsto & \sum_{i=-m}^{n-m-1} x_i \cdot b^i \\
 & \mapsto & 1 \cdot 2^{-4} + 0 \cdot 2^{-3} + 1 \cdot 2^{-2} + 0 \cdot 2^{-1} + \\
 & & 1 \cdot 2^{0} + 1 \cdot 2^{1} + 0 \cdot 2^{2} + 1 \cdot 2^{3} + 1 \cdot 2^{4} + 1 \cdot 2^{5} + 1 \cdot 2^{6} \\
 & \mapsto & 1 \cdot 1/16 + 0 \cdot 1/8 + 1 \cdot 1/4 + 0 \cdot 1/2 + \\
 & & 1 \cdot 1 + 1 \cdot 2 + 0 \cdot 4 + 1 \cdot 8 + 1 \cdot 16 + 1 \cdot 32 + 1 \cdot 64 \\
 & \mapsto & 123.3125_{(10)}
\end{array}$$

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

Notes:

Example

Fractional components can be accommodated by tweaking the range of i in our original definition; we imagine there are m digits in the fractional component. 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

$$\begin{array}{lcl}
\hat{x} = 173.24 & = & \langle 2,4,3,7,1 \rangle \\
 & \mapsto & \sum_{i=-m}^{n-m-1} x_i \cdot b^i \\
 & \mapsto & 4 \cdot 8^{-2} + 2 \cdot 8^{-1} + \\
 & & 3 \cdot 8^{0} + 7 \cdot 8^{1} + 1 \cdot 8^{2} \\
 & \mapsto & 4 \cdot 1/16 + 2 \cdot 1/8 + \\
 & & 3 \cdot 8 + 7 \cdot 64 + 1 \cdot 512 \\
 & \mapsto & 123.3125_{(10)}
\end{array}$$

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

Notes:

Example

Fractional components can be accommodated by tweaking the range of i in our original definition; we imagine there are m digits in the fractional component. 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

$$\begin{array}{lcl}
\hat{x} = 7B.5 & = & \langle 5, B, 7 \rangle \\
 & \mapsto & \sum_{i=-m}^{n-m-1} x_i \cdot b^i \\
 & \mapsto & 5 \cdot 16^{-1} + \\
 & & 11 \cdot 16^{0} + 7 \cdot 16^{1} \\
 & \mapsto & 5 \cdot 1/16 + \\
 & & 11 \cdot 1 \quad + 7 \cdot 16 \\
 & \mapsto & 123.3125_{(10)}
\end{array}$$

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

Notes:

$\mathbb{Z}$ (1)

- Problem: we want to represent and perform various operations on elements of $\mathbb{Z}$, *but*
  1. it's an an infinite set, and
  2. so far we've ignored the issue of sign.
- Solution: 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?

Notes:

$\mathbb{Z}$ (2)
Unsigned

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

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

Notes:

$\mathbb{Z}$ (3)
Signed, 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

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

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

Notes:

Example

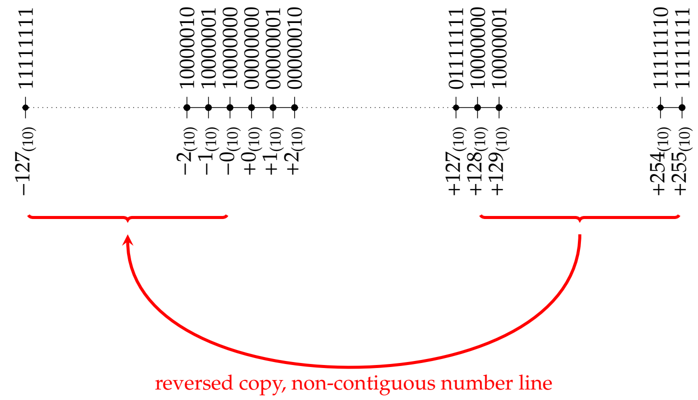
If $n = 8$, for example, we can represent values in the range $-127 \ldots +127$; selected cases are as follows:

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

Notes:

Example

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



Notes:

Definition

A signed integer can be represented in n bits by using the **two's-complement** approach. The basic idea is to weight the $(n-1)$-th bit 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

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

Notes:

Example

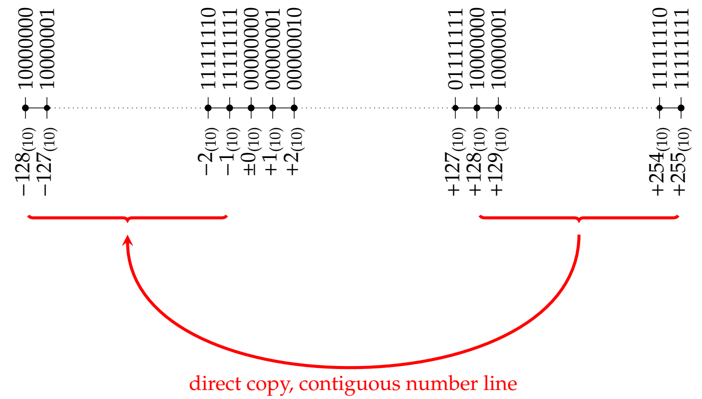
If $n = 8$ for example, we can represent values in the range $-128 \ldots +127$; selected cases are as follows:

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

Notes:

Example

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



Notes:

An Aside: The shift-and-mask paradigm

- Problem: set the i-th bit of some x, i.e., x_i, to 1.
- Solution: compute

$$x \vee (1 \ll i).$$

Example

If $x = 0011_{(2)}$ and $i = 2$ then we compute

$$\begin{array}{lclllll}
x & \vee & (& 1 & \ll & i &) \\
0011_{(2)} & \vee & (& 1 & \ll & 2 &) \\
0011_{(2)} & \vee & & 0100_{(2)} & & & \\
0111_{(2)} & & & & & &
\end{array}$$

meaning initially $x_2 = 0$, then we changed it so $x_2 = 1$.

Notes:

- Problem: set the i-th bit of some x, i.e., x_i, to 0.
- Solution: compute

$$x \wedge \neg(1 \ll i).$$

Example

If $x = 0111_{(2)}$ and $m = 2$ then we compute

$$\begin{array}{lccclcc}
x & \wedge & \neg & (& 1 & \ll & i &) \\
0111_{(2)} & \wedge & \neg & (& 1 & \ll & 2 &) \\
0111_{(2)} & \wedge & \neg & (& 0100_{(2)} & & &) \\
0111_{(2)} & \wedge & & & 1011_{(2)} & & & \\
0011_{(2)} & & & & & & &
\end{array}$$

meaning initially $x_2 = 1$, then we changed it so $x_2 = 0$.

Notes:

An Aside: The shift-and-mask paradigm

- Problem: extract the i-th bit of some x, i.e., x_i.
- Solution: compute

$$(x \gg i) \wedge 1.$$

Example

If $x = 0011_{(2)}$, then

1. if $i = 2$ we compute

$$\begin{array}{clccccc}
(& x & \gg & i &) & \wedge & 1 \\
(& 0011_{(2)} & \gg & 2 &) & \wedge & 1 \\
(& 0000_{(2)} & & &) & \wedge & 1 \\
 & 0000_{(2)} & & & & &
\end{array}$$

   meaning $x_2 = 0$, or
2. if $i = 0$ we compute

$$\begin{array}{clccccc}
(& x & \gg & i &) & \wedge & 1 \\
(& 0011_{(2)} & \gg & 0 &) & \wedge & 1 \\
(& 0011_{(2)} & & &) & \wedge & 1 \\
 & 0001_{(2)} & & & & &
\end{array}$$

   meaning $x_0 = 1$.

Notes:

- Problem: extract an m-bit sub-word (i.e., m contiguous bits) starting at the i-th bit of some x.
- Solution: compute

$$(x \gg i) \wedge ((1 \ll m) - 1).$$

Example

If $x = 1011_{(2)}$, $m = 2$ and $i = 1$ then we want to extract the sub-word $\langle x_1, x_2 \rangle$

$$\begin{array}{ccccccccccccccc}
(& x & \gg & i &) & \wedge & (& (& 1 & \ll & m &) & - & 1 &) \\
(& 1011_{(2)} & \gg & 1 &) & \wedge & (& (& 1 & \ll & 2 &) & - & 1 &) \\
(& 0101_{(2)} & & &) & \wedge & (& (& 0100_{(2)} & & &) & - & 1 &) \\
(& 0101_{(2)} & & &) & \wedge & (& & 0011_{(2)} & & & & & &) \\
 & 0001_{(2)} & & & & & & & & & & & & &
\end{array}$$

meaning $\langle x_1, x_2 \rangle = \langle 1, 0 \rangle$ as expected.

Notes:

Conclusions

- Take away points:
  1. *We* control what bit-sequences mean: we can interpret an instance of the C `int` data-type as
     - a signed 32-bit integer, *or*
     - a generic object which can take one of 2^{32} states,

     and, as a result, can represent *anything*, e.g.,
     - a pixel within an image,
     - a character within a text file,
     - a network IP address,
     - ...
  2. Beyond this, knowing about various standard representations is important and useful in a general sense.

Notes:

Additional Reading

- *Wikipedia: Numeral system*. URL: http://en.wikipedia.org/wiki/Numeral_system.
- D. Page. "Chapter 1: Mathematical preliminaries". In: *A Practical Introduction to Computer Architecture*. 1st ed. Springer-Verlag, 2009.
- B. Parhami. "Part 1: Number representation". In: *Computer Arithmetic: Algorithms and Hardware Designs*. 1st ed. Oxford University Press, 2000.
- W. Stallings. "Chapter 9: Number systems". In: *Computer Organisation and Architecture*. 9th ed. Prentice-Hall, 2013.
- A.S. Tanenbaum and T. Austin. "Appendix A: Binary numbers". In: *Structured Computer Organisation*. 6th ed. Prentice-Hall, 2012.

Notes:

References

[1] *Wikipedia: Numeral system*. URL: http://en.wikipedia.org/wiki/Numeral_system (see p. 69).

[2] D. Page. "Chapter 1: Mathematical preliminaries". In: *A Practical Introduction to Computer Architecture*. 1st ed. Springer-Verlag, 2009 (see p. 69).

[3] B. Parhami. "Part 1: Number representation". In: *Computer Arithmetic: Algorithms and Hardware Designs*. 1st ed. Oxford University Press, 2000 (see p. 69).

[4] W. Stallings. "Chapter 9: Number systems". In: *Computer Organisation and Architecture*. 9th ed. Prentice-Hall, 2013 (see p. 69).

[5] A.S. Tanenbaum and T. Austin. "Appendix A: Binary numbers". In: *Structured Computer Organisation*. 6th ed. Prentice-Hall, 2012 (see p. 69).

Notes: