Information Coding

Lecturer: Guillaume Beslon
Original Author: Lionel Morel

Computer Science and Information Technology - INSA Lyon

Fall 2019
Content

Introduction

Natural Numbers in base 2

Conversion

Signed Integers

Notations and Other Interpretations
Where are we?

High-level language program (in C)

```
swap(int v[], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

Assembly language program (for MIPS)

```
swap:
    multi $2, $5,4
    add $2, $4,$2
    lw $15, 0($2)
    lw $16, 4($2)
    sw $16, 0($2)
    sw $15, 4($2)
    jr $31
```

Binary machine language program (for MIPS)

```
00000000001010001000100000010011000
000000001000001000100001000010001
10001101111001000000000000000000
1000111000010010000000000000000100
101011100001001000000000000000000
1010110111100100000000000000000100
00000011111000000000000000000000
00000011111000000000000000000000
```
Where are we?

Programmers...
... usually manipulate numbers (Relative Numbers, Reals, etc)

Machines ....
... only understand binary

⇒ We need to make the link between the two.
Bit vectors (aka “Words”)

Basic information: the \texttt{bit} \in \{0, 1\} (for binary digit)

Example of word: 01010101110010011100001111

We work with finite words!

\(l\) is the number of bits: in practice \(l = 8 \times n\)

\(n\) is the number of bytes (octets in french)

When programming (on a 32 or 64 bits machine):

\begin{itemize}
  \item \(n = 1\) is called a \texttt{byte} or a \texttt{char}
  \item \(n = 2\) is called a \texttt{short}
  \item \(n = 4\) is \textit{usually} called an \texttt{int} (or \texttt{float} if it represents a pseudo-real number)
  \item \(n = 8\) is called \texttt{long long} (or \texttt{double} for a pseudo-real)
\end{itemize}

Beware, these are naming \textit{conventions} only!!!

So, always make sure you know what you are talking about!
Consider the names chosen for the numeric types in C:

- `char` (the 8-bit integer) is an abbreviated noun (character) from **typography**
- `unsigned char` ???
- You can add two `char` ???
- `int` is an abbreviated noun (integer) from **mathematics**
- Although `2.147.483.647 +1 = -2.147.483.648`
- `short` and `long` are **adjectives**
- `float` is a verb, at least it is a **computer** term
- `double` means double **what?**
- `long double` is not even syntactically correct in English (and what about `long long`!)

After so much nonsense, if you’re lost, **it is not your fault**

---

1 courtesy of F. de Dinechin
Introduction

Natural Numbers in base 2

Conversion

Signed Integers

Notations and Other Interpretations
Natural numbers in base 2 (Unsigned integers)

Let $x$ be a vector of $n$ bits:
$x_{n-1}, x_{n-2}, \ldots, x_1, x_0$, with $x_i \in 0, 1$ (in base 2).
Using positional notation we can interpret the value of $x$ as a natural number:

$$x = \sum_{i=0}^{n-1} x_i \cdot 2^i$$

$2^n$ different values can be represented:

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

see the 'poly' for a generalization to base $\beta > 1$
Blackboard examples
**LSBit** is the bit position in a binary integer giving the units value, that is, determining whether the number is even or odd\(^2\).

\[
\begin{array}{cccccc}
1 & 0 & 0 & 1 & 0 & 1 & 0 & 1
\end{array}
\]

**LSByte** is the byte in that position of a multi-byte number which has the least potential value.

\(^2\)https://en.wikipedia.org/wiki/Least_significant_bit
**MSB**: Most Significant Bit (Byte)

**MSBit** is the bit position in a binary number having the greatest value.

![Binary Number](image)

**MSByte** is the byte (or octet) in that position of a multi-byte number which has the greatest potential value.

---

[^3]: [https://en.wikipedia.org/wiki/Most_significant_bit](https://en.wikipedia.org/wiki/Most_significant_bit)
Warning: MSB/LSB not to be confused with “little endian” and “big endian”

- MSB/LSB are to be understood relatively to the **Binary code**.
- “Little-Endian” and “Big-Endian” are ways to organise the bytes into the **physical memory** of a computer...

→ Comics vs. Mangas

- “Big-Endian”: The most significant byte is stored first in the lowest memory address (or big in first).
- “Little-Endian”: Stores the least significant bytes in the lowest memory address.

→ When you directly look at the memory content, be careful to the “endianness” (aka “boutisme”)...
Notations and important values

We write: \((x_{n-1}, x_{n-2}, \ldots, x_1, x_0)_\beta\)
when writing \(x\) in base \(\beta\).

eg:

- \((101)_2 = (5)_{10}\)
- \((1010)_2 = (10)_{10}\)

for \(n = 8\):

- max: \(2^n - 1 = 255\)
- 256 different values

for \(n = 32\):

- \(2^n - 1 = 4.294.967.295\)
- \(\approx 4\) billions different values

for \(n = 16\):

- \(2^n - 1 = 65535\)
- 65536 different values

for \(n = 64\):

- \(2^n - 1 = \) a lot!
- Still not a infinite number of values...

\(^4\)Probably one of the rare things to be known by heart
Content

Introduction

Natural Numbers in base 2

Conversion

Signed Integers

Notations and Other Interpretations
Binary $\rightarrow$ Decimal Conversion

Any number $p \in \mathbb{N}$ can be represented in a unique positional form in base 2, using $n$ bits (with $n = \lceil \log_2(p) \rceil + 1$):

$$(x_{n-1}x_{n-2}\cdots x_1x_0)_2 := \sum_{i=0}^{n-1} x_i2^i.$$ 

**NB:** bits are numbered from 0 to $n - 1$
Binary $\leftarrow$ Decimal Conversion

The remainder of the euclidean division of $x$ by 2 gives the right-most digit of its representation in base 2:

$$x = x_{n-1} \cdot 2^{n-1} + x_{n-2} \cdot 2^{n-2} + \cdots + x_2 \cdot 2^2 + x_1 \cdot 2^1 + x_0$$

$$= \left( x_{n-1} \cdot 2^{n-2} + x_{p-2} \cdot 2^{n-3} + \cdots + x_2 \cdot 2^1 + x_1 \right) \cdot 2 + x_0 \quad \text{quotient}$$

so, $x_0 = n \mod 2$.

We get all the digits of the binary representation of a natural number $n$ by applying euclidean divisions to the successive quotients until we reach 0 as a quotient.

**NB:** This gives least-significant bits first!

Again, see “poly” for a generalization to any base $\beta > 1$
Binary ← Decimal Conversion - example

To convert $n = (423)_{10}$ to binary, we

\[
423 = 211 \times 2 + 1 \\
211 = 105 \times 2 + 1 \\
105 = 52 \times 2 + 1 \\
52 = 26 \times 2 + 0 \\
26 = 13 \times 2 + 0 \\
13 = 6 \times 2 + 1 \\
6 = 3 \times 2 + 0 \\
3 = 1 \times 2 + 1 \\
1 = 0 \times 2 + 1
\]

From this we deduce that: $(423)_{10} = (110100111)_{2}$
Blackboard examples
Fast equivalence: the trick!

\[10^3 \approx 2^{10}\]
Blackboard examples
Content

Introduction

Natural Numbers in base 2

Conversion

Signed Integers

Notations and Other Interpretations
Signed Integers

Two ways of representing $-x$:

- 1 bit for the sign, the rest for $|x|$

$$x_{n-1} = \begin{cases} 
0 & \text{if } x \geq 0, \\
1 & \text{if } x < 0. 
\end{cases} \quad \text{and} \quad (x_{n-2}, x_{n-3}, \ldots, x_1, x_0) = |x|$$

- Pros: Simple to understand
- Cons: 2 writings for 0
- Cons: hardware implementation ≠ unsigned integers

Two’s complement

- Cons: Less easy to understand
- Pros: numbering and hardware implementation is unchanged
  → Used in 99.99% of digital circuits\(^5\)

\(^5\)But other solutions are also used, e.g. for floating values...
Two’s Complement - equation

Let $x$ be a vector of $n$ bits: $x_{n-1}, x_{n-2}, \cdots, x_1, x_0$, with $x_i \in 0, 1$

The value of $x$ interpreted as a signed integer is:

$$x = -x_{n-1}2^{n-1} + \sum_{i=0}^{n-2} x_i2^i.$$ 

$2^n$ different values can be represented:

$$-2^{n-1} \leq x < 2^{n-1} - 1$$
Two’s Complement - intuition

credit: Benoît Lopez
Two’s Complement - compute the opposite of $x$

- The complement of a bit $a \in \{0, 1\}$ is:

$$\overline{a} = \begin{cases} 
1, & \text{when } a = 0 \\
0, & \text{when } a = 1 
\end{cases}$$

- if

$$x = x_{n-1}, x_{n-2}, \cdots, x_1, x_0$$

then

$$-x = (\overline{x_{n-1}}, \overline{x_{n-2}}, \cdots, \overline{x_1}, \overline{x_0}) + 1$$
Blackboard examples
Number Extension

How do we assign a \( n \)-bits vector to an \( m \)-bits vector?

- \( n > m \), truncate → lose most significant bits
- \( n < m \), requires a sign extension

Consider \((x_{n-1}, x_{n-2}, \ldots, x_1, x_0)\). Let’s write it as 
\((y_{m-1}, y_{m-2}, \ldots, y_n, y_{n-1}, y_{n-2}, \ldots, y_1, y_0)\). The value of \( x \) is preserved if we take:

\[
y_{m-1} = x_{n-1}, y_{m-2} = x_{n-1}, \ldots, y_n = x_{n-1}, y_{n-1} = x_{n-1},
\]

\[
y_{n-2} = x_{n-2}, \ldots, y_1 = x_1, y_0 = x_0
\]
Content

Introduction

Natural Numbers in base 2

Conversion

Signed Integers

Notations and Other Interpretations
Useful notations

Reading binary is a pain in the ass! In practice, everybody uses **hexadecimal**: 

<table>
<thead>
<tr>
<th>binary</th>
<th>hexa</th>
<th>decimal</th>
<th>binary</th>
<th>hexa</th>
<th>decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0x0</td>
<td>0</td>
<td>1000</td>
<td>0x8</td>
<td>8</td>
</tr>
<tr>
<td>0001</td>
<td>0x1</td>
<td>1</td>
<td>1001</td>
<td>0x9</td>
<td>9</td>
</tr>
<tr>
<td>0010</td>
<td>0x2</td>
<td>2</td>
<td>1010</td>
<td>0xA</td>
<td>10</td>
</tr>
<tr>
<td>0011</td>
<td>0x3</td>
<td>3</td>
<td>1011</td>
<td>0xB</td>
<td>11</td>
</tr>
<tr>
<td>0100</td>
<td>0x4</td>
<td>4</td>
<td>1100</td>
<td>0xC</td>
<td>12</td>
</tr>
<tr>
<td>0101</td>
<td>0x5</td>
<td>5</td>
<td>1101</td>
<td>0xD</td>
<td>13</td>
</tr>
<tr>
<td>0110</td>
<td>0x6</td>
<td>6</td>
<td>1110</td>
<td>0xE</td>
<td>14</td>
</tr>
<tr>
<td>0111</td>
<td>0x7</td>
<td>7</td>
<td>1111</td>
<td>0xF</td>
<td>15</td>
</tr>
</tbody>
</table>

Max values:  

- on 1 byte: \(0xFF = 255_{10}\)
- on 2 bytes: \(0xFFFF = 65.535_{10}\)
- on 4 bytes: \(0xFFFFFFFF = 4.294.967.295_{10}\)
Other interpretation of bit vectors

- (Pseudo-)real numbers
- Bit fields, indicators, booleans, ...
- Characters/symbols: encodings such as ASCII or ISO or UTF-8. Useful for writing human-readable messages
- Encoded/compressed data (bzip, ...), music (mp3, ogg, ...), video (mpeg2, h264, ...)
- Encrypted data
- Instructions = programs in their “final” form (that which is interpreted by HW).
- ...

All this necessitates interpreting these bit vectors!
Integers have a constant precision but a very low dynamics...

- Difficult to represent very large values (e.g. mass of the universe in kilograms)
- Impossible to represent very small values (e.g. mass of an electron in kilograms)
- Encoding of useless (and sometimes misleading) LSB

Solution: scientific notation ($3 \times 10^{52}$ and $9, 109 \times 10^{-31}$)

Floating points are the binary analogs to scientific notation. Pseudo-real numbers are represented by a mantissa (or fraction) and an exponent coded by (still finite) bit vectors.
Floating points

Floating points are normalised by the IEEE 754 format. Mainly two formats are used:

- Single precision (32 bits): sign (1 bit), exponent (8 bits, including exponent sign), fraction (23 bits) → float
- Double precision (64 bits): sign (1 bit), exponent (11 bits, including exponent sign), fraction (52 bits) → double

Floating points are NOT reals. The maximum number of values one can encode with a float is still $2^{32} = 4.294.967.296$! → Floating points have precision issues that are (1) too difficult to be listed here and (2) extremely dangerous! (e.g., in general $a + b \neq b + a$ and $b + a - b - a \neq 0$)
## TACTL, Timer_A Control Register

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-10</td>
<td>Unused</td>
<td>15-10</td>
</tr>
<tr>
<td>9-8</td>
<td><strong>TASSELx</strong> Timer_A clock source select</td>
<td>9-8</td>
</tr>
<tr>
<td>7-6</td>
<td><strong>IDx</strong> Input divider. These bits select the divider for the input clock.</td>
<td>7-6</td>
</tr>
<tr>
<td>5-4</td>
<td><strong>MCx</strong> Mode control. Setting MCx = 00h when Timer_A is not in use conserves power.</td>
<td>5-4</td>
</tr>
</tbody>
</table>

### TASSELx Bits
- **00**: TACLK
- **01**: ACLK
- **10**: SMCLK
- **11**: Inverted TACLK

### IDx Bits
- **00**: /1
- **01**: /2
- **10**: /4
- **11**: /8

### MCx Bits
- **00**: Stop mode: the timer is halted
- **01**: Up mode: the timer counts up to TACCR0
- **10**: Continuous mode: the timer counts up to 0FFFFh
- **11**: Up/down mode: the timer counts up to TACCR0 then down to 0000h

### Unused Bit
- Bit 3: Unused
## Character encoding — UTF-8

<table>
<thead>
<tr>
<th>Character</th>
<th>Binary code point</th>
<th>Binary UTF-8</th>
<th>Hexadecimal UTF-8</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>U+0024</td>
<td>010 0100</td>
<td>00100100</td>
</tr>
<tr>
<td>€</td>
<td>U+00A2</td>
<td>000 1010 0010</td>
<td>11000010 10100010</td>
</tr>
<tr>
<td>€</td>
<td>U+20AC</td>
<td>0010 0000 1010 1100</td>
<td>11100010 10000010 10101100</td>
</tr>
<tr>
<td>🕒</td>
<td>U+10348</td>
<td>0 0001 0000 0011 0100 1000</td>
<td>11110000 10010000 10001101 10001000</td>
</tr>
</tbody>
</table>
3.4.1 Double-Operand (Format I) Instructions

Figure 3–9 illustrates the double-operand instruction format.

**Figure 3–9. Double-Operand Instruction Format**

![Diagram of double-operand instruction format]

Table 3–11 lists and describes the double operand instructions.

**Table 3–11. Double-Operand Instructions**

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>S-Reg, D-Reg</th>
<th>Operation</th>
<th>V</th>
<th>N</th>
<th>Z</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV (.B)</td>
<td>src, dst</td>
<td>src → dst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD (.B)</td>
<td>src, dst</td>
<td>src + dst → dst</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>ADDC (.B)</td>
<td>src, dst</td>
<td>src + dst + C → dst</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>SUB (.B)</td>
<td>src, dst</td>
<td>dst + .not.src + 1 → dst</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>SUBC (.B)</td>
<td>src, dst</td>
<td>dst + .not.src + C → dst</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>CMP (.B)</td>
<td>src, dst</td>
<td>dst - src</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>DADD (.B)</td>
<td>src, dst</td>
<td>src + dst + C → dst (decimally)</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>BIT (.B)</td>
<td>src, dst</td>
<td>src .and. dst</td>
<td>0</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>BIC (.B)</td>
<td>src, dst</td>
<td>.not.src .and. dst → dst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BIS (.B)</td>
<td>src, dst</td>
<td>src .or. dst → dst</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XOR (.B)</td>
<td>src, dst</td>
<td>src .xor. dst → dst</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>AND (.B)</td>
<td>src, dst</td>
<td>src .and. dst → dst</td>
<td>0</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>
Next lecture...

Combinatorial Logic and Circuits!!

ie, how to process information encoded in bit vectors?