計算機での数値表現

柏木 雅英
kashi@waseda.jp
http://verifiedby.me/

早稲田大学 基幹理工学部 応用数理学科

計算機のメモリ

- $0, 1$ の組み合わせにより全てのデータを表現している。
- 0 か 1 が覚えられる記憶素子が 1 つだけでは当然 2 通りの状態しか表せないが、n 個組み合わせれば 2^n 通りの状態が表現可能になる。
- 記憶容量の単位として、0 か 1 が覚えられる記憶素子 1 つを 1bit と呼ぶ。
- 伝統的に 8bit で 1 文字を表現してきたので、8bit を一組で扱うことが多く、8bit の塊を 1byte と言う。
- $2^{10} = 1024 \simeq 1000$ なので、計算機の世界では K(キロ)、M(メガ)、G(ギガ)、T(テラ) などは、それぞれ 2^{10}、2^{20}、2^{30}、2^{40} を表すことが多い。言うまでもなく物理ではそれぞれ 10^3、10^6、10^9、10^{12}。
- 例えば $16GB$ のメモリと言った場合、

$$16 \times 2^{30} \times 8 = 137438953472$$

個の $0, 1$ を記憶できる素子を持つ。

計算機での数値の表現

- 整数 (C 言語で言うところの `int`) と、実数 (C 言語で言うところの `float, double`) で表現の方法が違う。
- 基本的に固定長である。固定長でないと、配列アクセスの効率が非常に悪くなってしまう (`a[i]` が`*(a+i)` でアクセスできなくなってしまう)。
- 整数 (C 言語での `int`) は多くの場合 32bit=4byte。実数は、
  - 単精度 (C 言語での `float`): 32bit = 4byte
  - 倍精度 (C 言語での `double`): 64bit = 8byte
- 整数型には、符号なし (0 または正の数が表現できる) と、符号あり (負、0、正の数が表現できる) の 2 種類がある。

符号なし整数

C 言語での `unsigned int`。

bit pattern と数値の対応

b_{31}	b_{30}	b_{29}	$\cdots$	b_2	b_1	b_0	x
1	1	1	$\cdots$	1	1	1	$4294967295 = 2^{32} - 1$
1	1	1	$\cdots$	1	1	0	4294967294
				$\vdots$			
0	0	0	$\cdots$	0	1	1	3
0	0	0	$\cdots$	0	1	0	2
0	0	0	$\cdots$	0	0	1	1
0	0	0	$\cdots$	0	0	0	0

$$x = \sum_{i=0}^{31} b_i 2^i$$

要するに普通の 2 進数。

符号付き整数

C 言語での `int`、`signed int`。

bit pattern と数値の対応

b_{31}	b_{30}	b_{29}	$\cdots$	b_2	b_1	b_0	x
0	1	1	$\cdots$	1	1	1	$2147483647 = 2^{31} - 1$
				$\vdots$			
0	0	0	$\cdots$	0	1	0	2
0	0	0	$\cdots$	0	0	1	1
0	0	0	$\cdots$	0	0	0	0
1	1	1	$\cdots$	1	1	1	-1
1	1	1	$\cdots$	1	1	0	-2
1	1	1	$\cdots$	1	0	1	-3
				$\vdots$			
1	0	0	$\cdots$	0	0	0	$-2147483648 = -2^{31}$

$$x = -b_{31}2^{31} + \sum_{i=0}^{30} b_i 2^i \quad (b_{31} \text{ の重みだけ符号が逆})$$

2 の補数形式と呼ばれる独特の格納方法。

2 進数での加算の原理 (1)

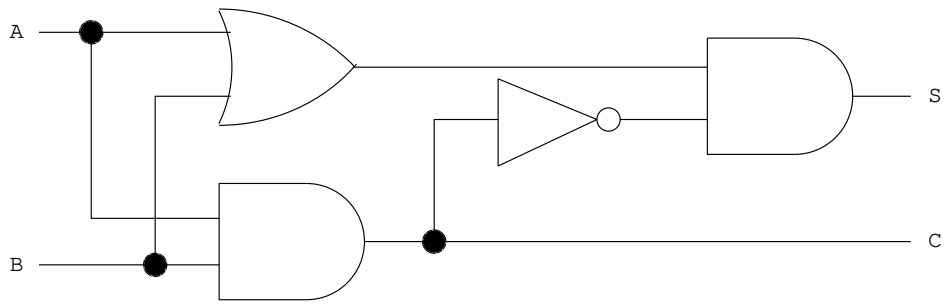
$7 + 3 = 10$ を 2 進数 4bit の筆算で行う様子:

$$\begin{array}{cccc} & & & \color{red}{1} & \\ & 0 & 1 & 1 & 1 \\ + & 0 & 0 & 1 & 1 \\ \hline & & & & \color{red}{0} \end{array} \Rightarrow \begin{array}{cccc} & & \color{red}{1} & & \\ & 0 & 1 & 1 & 1 \\ + & 0 & 0 & 1 & 1 \\ \hline & & & \color{red}{1} & 0 \end{array} \Rightarrow \begin{array}{cccc} & \color{red}{1} & & & \\ & 0 & 1 & 1 & 1 \\ + & 0 & 0 & 1 & 1 \\ \hline & & \color{red}{0} & 1 & 0 \end{array} \Rightarrow \begin{array}{cccc} & 0 & 1 & 1 & 1 \\ + & 0 & 0 & 1 & 1 \\ \hline & \color{red}{1} & 0 & 1 & 0 \end{array}$$

$\{0, 1\}$ を 3 つ受け取り、その合計 $(00, 01, 10, 11$ の 4 通りの出力$)$ を吐き出すような回路 (全加算器) があればこれを実行する回路が作れる。

2進数での加算の原理 (2) 半加算器

Input		Output	
A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

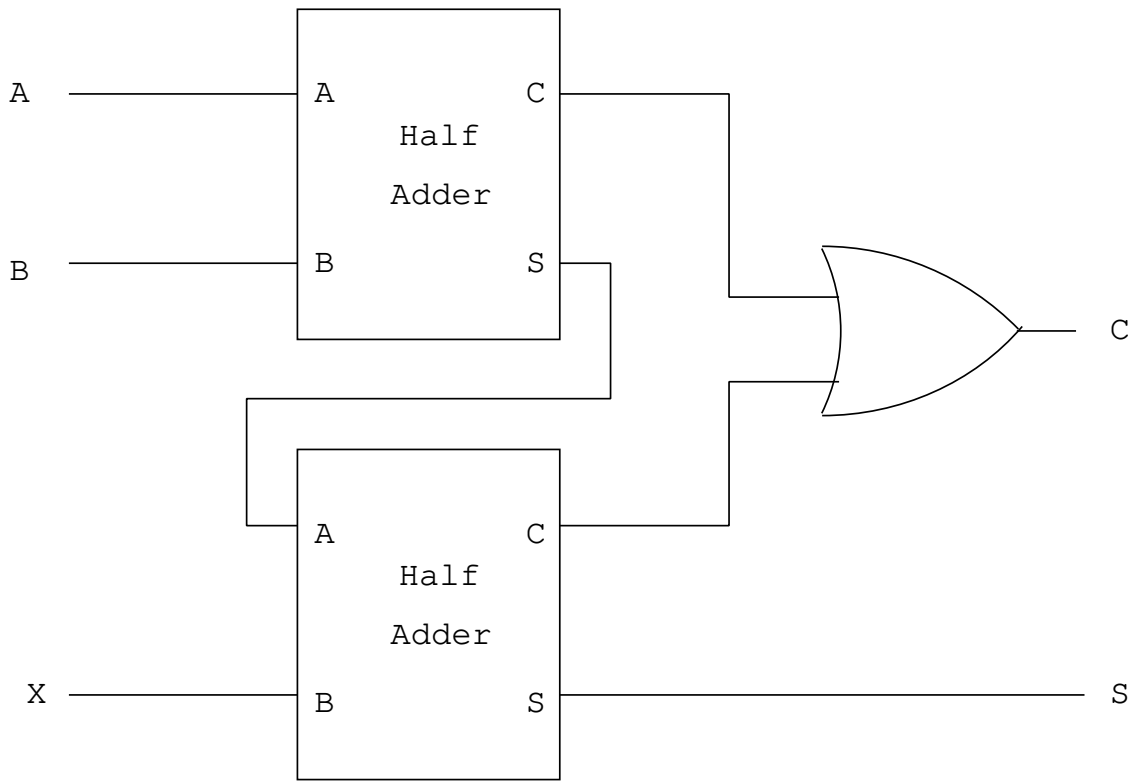


A,B が入力、S が演算結果、C が桁上がり情報である。

2進数での加算の原理 (3) 全加算器

桁上がりも入力として含めて加算を行う。半加算器 2 つで構成できる。

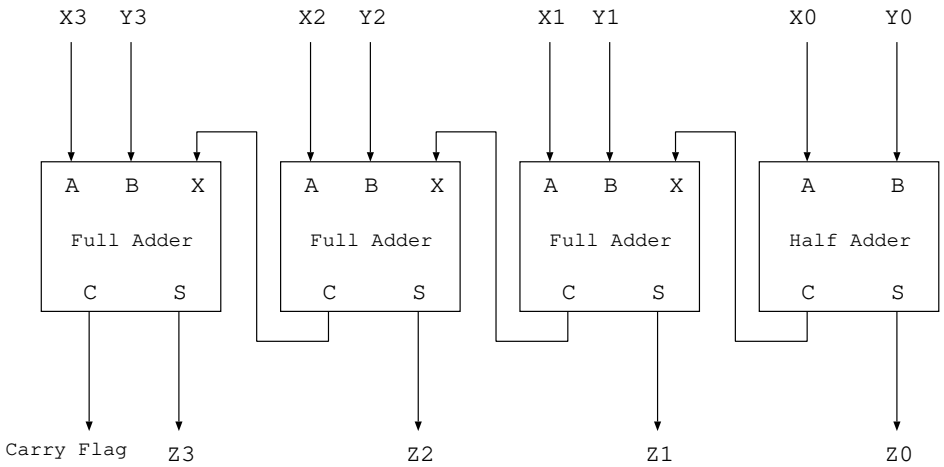
Input			Output	
A	B	X	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



A,B,X が入力、S が演算結果、C が桁上がり情報である。

2進数での加算の原理 (4) 4bit の加算器

(X_3, X_2, X_1, X_0) と (Y_3, Y_2, Y_1, Y_0) の和を計算し (Z_3, Z_2, Z_1, Z_0) に格納する 4bit 加算器の例。



最上位 bit からの桁あふれは捨てられる。こういう加算回路を用いると、符号ありと符号無しで全く同じ回路が使える。例えば、4bit の回路において $6 + (-1)$ は、$0110 + 1111 = (1)0101$ で、溢れた bit を捨てると 5 が得られる。

浮動小数点数

C 言語で言うところの `float`、`double`。

浮動小数点形式

例えば、「1234.5」に対する「1.2345×10^3」のように、小数点の位置を一番左の数値と左から 2 番目の数値の間に移動し (この作業を正規化と呼ぶ)、それに基数 (radix) のベキを掛けた形式。この「1.2345」の部分を仮数部といい、「10^3」の部分を指数部という。

浮動小数点数は仮数部の長さ、指数部の長さ、基数が 2 か 10 か 16 か、など、様々なバリエーションが考えられ、実際昔は計算機メーカー毎に様々なフォーマットが乱立していた。そこで、1985 年に William Kahan が中心となって

IEEE 754: Standard for Binary Floating-Point Arithmetic

という標準規格が制定された。幸いなことにこれ以降に世に出たハードウェアのほぼ全てがこの規格に従うこととなった。

IEEE 754 倍精度浮動小数点数 (1)

64bit=8byte を、符号部 s (1bit)、指数部 e (11bit)、仮数部 m (52bit) に分割する。

s	e_{10}	e_9	$\cdots$	e_0	m_{51}	m_{50}	$\cdots$	m_0

e を 2 進整数 $(e = \sum_{i=0}^{10} e_i 2^i)$、$m$ を 2 進小数 $(m = \sum_{i=0}^{51} m_i 2^{i-52})$ とする。このとき、$0 \leq e \leq 2047$、$0 \leq m < 1$ である。
$1 \leq e \leq 2046$ の範囲のとき、これを正規化数といい、

$$(-1)^s \times (1 + m) \times 2^{e-1023}$$

で計算される数値と対応する。

- 仮数部 m に 1 を加えていることに注目。正規化した 2 進数は (0 を除いて) 必ず先頭が 1 になるので、その 1 をメモリに格納しないことによって、52bit 分のメモリ消費で 53bit 長の仮数部を表現できる。
- e の両端の $e = 0$ と $e = 2047$ は特殊用途に使われている。

IEEE 754 倍精度浮動小数点数 (2)

- $e = 0, m = 0$ なら、± 0
- $e = 2047, m = 0$ なら、$\pm\infty$
- $e = 2047, m \neq 0$ なら、NaN (Not a Number)。負数の平方根など、不可能な演算の結果を表すのに使われる。
- $e = 0, m \neq 0$ は、「非正規化数」を表す。正規化をするならアンダーフローして 0 に落ちてしまうような極めて 0 に近い数を、正規化を諦めることによって精度を落としてでも何とか表現しようという意図で設けられている。

	$m = 0$	$m \neq 0$
$e = 0$	± 0	$(-1)^s \times (0 + m) \times 2^{-1022}$ (非正規化数)
$1 \leq e \leq 2046$	$(-1)^s \times (1 + m) \times 2^{e-1023}$ (正規化数)	
$e = 2047$	$\pm\infty$	NaN (Not a Number)

倍精度浮動小数点数の例

10 進数の 5.25 は IEEE754 の倍精度でどう表現されるか。

- $5.25_{(10)} = 101.01_{(2)}$　(2 進数に)
- $101.01 = 1.0101 \times 2^2$　(正規化)
- (指数部)　$2 + 1023 = 1025_{(10)} = 10000000001_{(2)}$　(1023 の下駄を履かせ、11bit 符号なし 2 進数に)
- (仮数部)　1.0101 から先頭の 1 を取り除いたもの。長さは 52bit なので、後ろは 0 で埋める。
  0101000000000000000000000000000000000000000000000000
- (符号部)　正なら 0 負なら 1 なので、0。
- 符号部、指数部、仮数部の順に結合して、
  0|10000000001|0101000000000000000000000000000000000000000000000000
  となる。

bit pattern を実際に覗いてみる

プログラム

```
#include <stdio.h>

void peek(void *p, int size)
{
    int i, j;
    unsigned char c;

    for (i=size-1; i>=0; i--) {
        c = ((unsigned char *)p)[i];
        for (j=7; j>=0; j--) {
            if ((c & (1 << j)) != 0) {
                printf("1");
            } else {
                printf("0");
            }
        }
    }
    printf("\n");
}

int main()
{
    unsigned int u = 3;
    int i = -3;
    double x1 = 5.25;
    double x2 = 0.1;

    peek((void *)&u, sizeof(u));
    peek((void *)&i, sizeof(i));
    peek((void *)&x1, sizeof(x1));
    peek((void *)&x2, sizeof(x2));
}
```

実行結果

```
00000000000000000000000000000011
11111111111111111111111111111101
01000000000101010000000000000000000000000000000000000000000000000000000000000000
00111111101110011001100110011001100110011001100110011001100110011001100110011010
```

なお、このプログラムは Little Endian なシステム (Intel x86 等) でのみ正しく動作し、Big Endian なシステムの場合は"`for (i=size-1;i>=0;i--)`" を"`for (i=0;i<size;i++)`" と逆順に直す必要がある。

非正規化数

- アンダーフロー (指数部が小さすぎて 0 になってしまうこと) の発生を少しでも緩和するための悪あがき。
- (正の正規化数の最小数)
  $1.00000000000000000000000000000000000000000000000000 \times 2^{-1022} = 2^{-1022}$
- (最終 bit から 1 を減じたこれよりわずかに小さい数)
  $0.11111111111111111111111111111111111111111111111111 \times 2^{-1022}$
  $1.11111111111111111111111111111111111111111111111110 \times 2^{-1023}$
  は、指数部 -1023 が小さすぎて表現不能。普通はアンダーフローさせて 0。
- (非正規化数) 2^{-1022} を下回った数は正規化せずに指数部を 2^{-1022} に固定し、(正規化されていない仮数部の小数点以下を) そのまま格納する。
  $0.11111111111111111111111111111111111111111111111111 \times 2^{-1022}$
  $\vdots$
  $0.00000000000000000000000000000000000000000000000001 \times 2^{-1022} = 2^{-1074}$
  が表現可能に。仮数部の bit 数が 52 ～ 1 と本来の 53 より減少してしまっている点に注意。

倍精度浮動小数点数の限界値

正の倍精度浮動小数点数の大きさの限界値をまとめる。

- (最大数) $e = 2046, m = 111\ldots111$
  $2^{1024} - 2^{971} \simeq 1.7976931348623157 \times 10^{308} \simeq 10^{308.25}$
- (正規化数の最小数) $e = 1, m = 000\ldots000$
  $2^{-1022} \simeq 2.2250738585072014 \times 10^{-308} \simeq 10^{-307.65}$
- (非正規化数の最大数) $e = 0, m = 111\ldots111$
  $2^{-1022} - 2^{-1074} \simeq 2.2250738585072201 \times 10^{-308} \simeq 10^{-307.65}$
- (非正規化数の最小数) $e = 0, m = 000\ldots001$
  $2^{-1074} \simeq 4.9406564584124654 \times 10^{-324} \simeq 10^{-323.31}$

-2^{1024}	-2^{-1022}	-2^{-1074}		2^{-1074}	2^{-1022}	2^{1024}	
$-\infty$	正規化数	非正規化数	-0	$+0$	非正規化数	正規化数	∞

単精度浮動小数点数

32bit=4byte を、符号部 s (1bit)、指数部 e (8bit)、仮数部 m (23bit) に分割する。指数部長、仮数部長の違いはあるが、基本的には倍精度と同じ。

s	e_7	e_6	$\cdots$	e_0	m_{22}	m_{21}	$\cdots$	m_0

e を 2 進整数 $(e = \sum_{i=0}^{7} e_i 2^i)$、$m$ を 2 進小数 $(m = \sum_{i=0}^{22} m_i 2^{i-23})$ とする。

	$m = 0$	$m \neq 0$
$e = 0$	± 0	$(-1)^s \times (0 + m) \times 2^{-126}$ (非正規化数)
$1 \leq e \leq 254$	$(-1)^s \times (1 + m) \times 2^{e-127}$ (正規化数)	
$e = 255$	$\pm\infty$	NaN (Not a Number)

丸め誤差

- 計算結果などの数値をメモリに格納するとき、仮数部の長さに限界があるため (倍精度なら 53bit、単精度なら 24bit)、はみ出た部分は切り捨てるか切り上げるかするしかない。ここで発生する誤差を丸め誤差という。
- 切り捨てるか切り上げるかは、誤差の小さくなる方を採用する (10 進数で言うところの四捨五入のようなもの)。2 進数なので 0 捨 1 入。
- (偶数丸め) 切り捨てと切り上げの誤差が完全に等しいとき、すなわち、格納したい数が 2 つの隣り合った浮動小数点数のちょうど中間だったとき、仮数部の末尾の bit が 0 である方に丸めるというルールになっている。これを偶数丸めという。

$\frac{1}{10}$ はどのように格納されるか

$\frac{1}{10} = 0.1$ は、10 進数だと有限小数だが、2 進数だと無限小数。

$$0.1_{(10)} = 0.0001100110011001100\cdots_{(2)}$$

$1.10011001100110011001100110011001100110011001100110011001|1001100\cdots \times 2^{-4}$
のうち先頭の 53bit が格納可能なので、54bit 目を 0 捨 1 入して、
$1.10011001100110011001100110011001100110011001100110011010 \times 2^{-4}$
を格納する。$-4 + 1023 = 1019$ で、$1019_{(10)} = 01111111011_{(2)}$ なので、
$0|01111111011|1001100110011001100110011001100110011001100110011010$
のような bit 列で格納される。この数は、10 進数に直すと

$$0.1000000000000000055511151231257827021181583404541015625$$

で、0.1 より少し大きい数になっている。

10 進数に慣れている我々は 0.1 はきれいな割り切れる数という思い込みがあるので、0.1 に誤差が混入していることによって引き起こされる事故はとても多い。

絶対誤差, 相対誤差, machine epsilon

ある値 x になんらかの誤差が入って $\tilde{x}$ になったとき、

$$|x - \tilde{x}|$$

を絶対誤差、

$$\frac{|x - \tilde{x}|}{|x|} \quad \text{あるいは} \quad \frac{|x - \tilde{x}|}{|\tilde{x}|}$$

を相対誤差という。浮動小数点形式は、相対誤差をほぼ一定値に保つように設計されている。

machine epsilon

ある浮動小数点数の体系において、

$$(1 \text{ より大きな最小の浮動小数点数}) - 1$$

を、machine epsilon という。

相対誤差の大きさの目安として使われる。IEEE 754 倍精度では $2^{-52} \simeq 2.22 \times 10^{-16}$、単精度では $2^{-23} \simeq 1.19 \times 10^{-7}$。