# 電子計算機 **2E**

3E　2003.5.21

## 本日の授業のテーマ

本日の事業のテーマは、以下のとおりです。

(1) 数の正負の表現

本日の授業のゴールは、以下のとおり。

- コンピューター内部での負の数の表現方法がわかる。

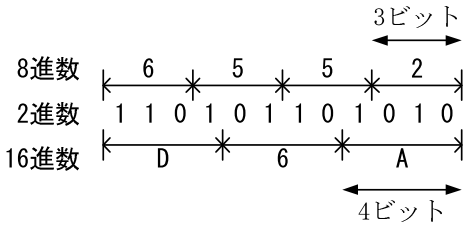
## 0. 先週の復習

・基数の変換(8, 16→10 進数)

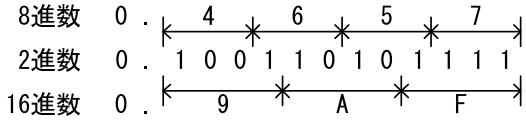
$$(376)_8 = (3\times10^2 + 7\times10^1 + 6\times10^0)_8 = (3\times8^2 + 7\times8^1 + 6\times8^0)_{10} = (3\times64 + 7\times8 + 6\times1)_{10} = (254)_{10}$$

$$(376)_{16} = (3\times10^2 + 7\times10^1 + 6\times10^0)_{16} = (3\times16^2 + 7\times16^1 + 6\times16^0)_{10} = (3\times256 + 7\times16 + 6\times1)_{10} = (886)_{10}$$

・整数の基数の変換(2→8, 16 進数)。8 進数への変換は第 0 ビット(最小桁)から 3 桁ずつ区切り、16 進数への変換は 4 桁ずつ区切り計算します。

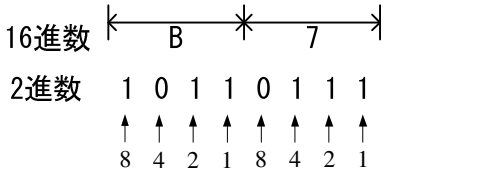



・小数の基数の変換(2→8, 16 進数)。8 進数への変換は小数点の次の桁から 3 桁ずつ区切り、16 進数への変換は 4 桁ずつ区切り計算します。




・桁数が合わない場合は、先頭に必要なだけ、ゼロを書き足して考えます。例えば、16 進数へ変換する場合、$(101100)_2 = (00101100)_2$ や $(0.111111)_2 = (0.11111100)_2$ として計算します。

・小数、整数の基数の変換(8, 16→2 進数)。16 進数から 2 進数への変換は、16 進数の各桁を(1, 2, 4, 8)の和に分けて、それぞれのビットに対応させます。8 進数の場合は、(1, 2, 4)の和に分けます。




$(B)_{16} = (11)_{10} = (8+2+1)_{10}$　　$(7)_{16} = (7)_{10} = (4+2+1)_{10}$

・基数の変換(10→進数 8, 16)。2 つの方法が有ります。

①一旦、2 進数へ変換した後、8 及び 16 進数へ変換する。

②整数の場合、8 又は 16 で割って、その余りが各桁になる。小数の場合は、8 又は 16 を乗じて、整数部が各桁になる。

# 1 負の数の表現

整数や小数の正の数の表現は非常に簡単でした。しかし、負の数の場合、事情が異なります。負の数の小数となると更にややこしい[1]ので、ここでは負の整数のみを取り扱うことにします。

通常は、数の前に‘ - ‘の符号をつけて、負の数を表します。コンピューター内部には、マイナスの符号などありませんから、それに対応した何かが必要です。負の数を表す約束事を作る必要があります。

いろいろな負の数を表す約束事(表現方法)が考えられます。数多くの負の表現方法がある中で、われわれが使うのは、

- 便利であること
- 表現が簡単であること
- ハードウェアーの実現が簡単であること

などを基準にして、実際に使うものを選択します。通常は、最期に紹介する 2 の補数表現を使いますが、ほかの表現方法も紹介しておきます。

コンピューターの情報の単位は 8 ビットの場合が多いです。そのため、これ以降は 8 ビットで話を進めます。

## 1.1 絶対値表示

絶対値表示とは、符号のためにビットを 1 つ設けて、あとは絶対値で表現する方法です。たとえば、図 1 のような表現です。人間にとって、この表現は分かりやすいですが、機械にとって良いか否かは別です。

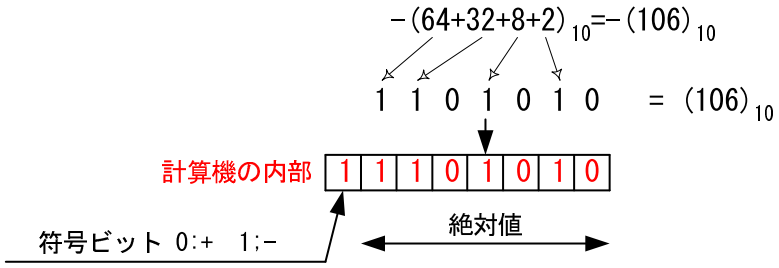



図 1 計算機内部の負の整数の表現(絶対値表示)

[1] これまでとここでの負の数の学習が理解できていれば、その応用なので、自分で考えることは出来るでしょう。

**1.2 1** の補数表示

1 の補数表示とは、絶対値のビットを反転する方法です。したがって、第 7 ビットが符号を表すことになります。この方法の欠点は、00000000 と 11111111 がともにゼロを表すことになります。

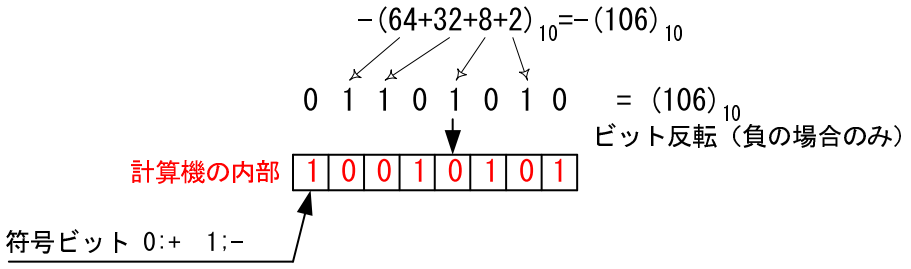



図 2 計算機内部の負の整数の表現(1 の補数表示)

**2.3 2** の補数表示

2 の補数の表現が、現在、計算機内部で普通に使われている方法です。これは、非常に重要です。図 3 に示しますが、手順は、

① 表したい負の数の絶対値を **2** 進数で表現し、ビットを反転する。
② それに、**+1** 加算する。

です。絶対に覚えてください。

要するに、先に示した 1 の補数表示に 1 を加えた表示方法です。

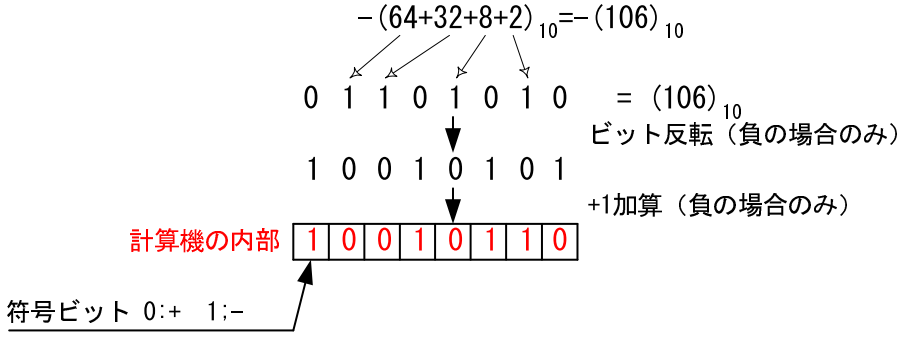



図 3 計算機内部の負の整数の表現(2 の補数表示)

この表現方法のイメージを図 4 に示します。ここで重要なことは、第 8 ビットを無視すると、2 進数が連続していることです。

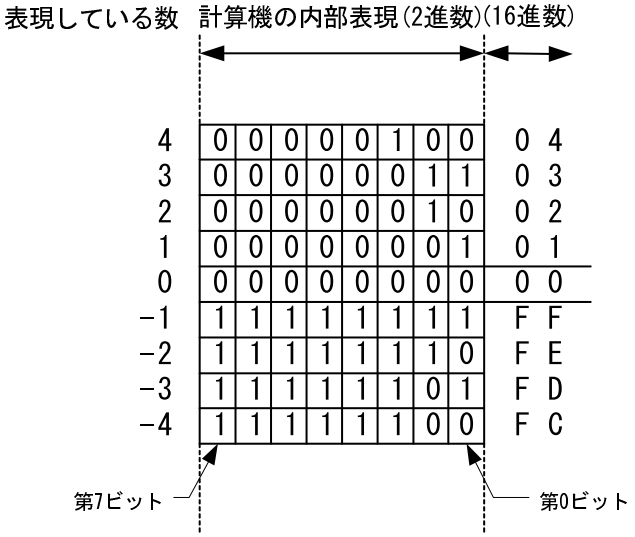



図 4 距離計イメージ(2 の補数表示)

この方法のメリットは、負の数の加算が加算器でできることです。これが可能だと、減算はその 2 の補数を計算して、加算器で出来ます。負の数の加算と正の数の減算は同じなので、負の数の加算が加算器で出来ることが重要です。

それでは、なぜ、2 の補数表現だと、負の数の加算が加算器で可能なのでしょうか?。理論的に説明をしましょう。ある正の整数を $x$ としましょう。その負の数、-x は 2 の補数表現では、

$$[-x]=(FF-x+1)_{16} \tag{1}$$

となります。左辺の[-x]が-x の意味です。[ ]の意味は、括弧内の負の整数を計算機内部の表現を意味しています。これは、私が作った表記ですので、一般には用いられていません。右辺の *FF-x* がビット反転です。それに 1 を加えて、2 の補数の表現としています。

つぎに、ある整数 y を考えて、*y-x* を計算してみましょう。

$$[y-x]=(y+FF-x+1)_{16} \tag{2}$$

*FF-x*+1 は、あらかじめ計算されて、コンピューター内部のメモリーに格納されているので、[y-x]は加算器で可能です。

これは、あたりまえです。重要なことは、この結果が、負の場合、2 の補数表現になっており、正の場合、そのままの値になっていることです。最初は、*y-x* が負になる場合を考えましょう。すると(2)式は、

$$[y-x]=(FF-(x-y)+1)_{16} \tag{3}$$

と変形できます。この場合、絶対値が *(x-y)* なので、①絶対値のビット反転②+1 加算となっています。つぎに、*y-x* が正になる場合を考えましょう。すると(2)式は、

$$[y-x]=(y-x+FF+1)_{16}$$
$$=(y-x+100)_{16} \qquad (4)$$

となります。100 は計算機内部では、桁上がりを示します。8 ビットの表示では無視されます。したがって、内部の表現は、正しく表せます。

## 3 どの方法が有利か

絶対値表示で負の数を減算する場合、減算器が必要です。場合によっては、比較器も必要かも知れません。1 の補数表示と 2 の補数表示は似ていますが、0 が 2 通りで表示する 1 の補数表示は良くないように思えます。2 の補数表示は、減算をする場合、加算器が使えます。ただし、2 の補数への変換が必要です。

減算の場合、

| | |
|---|---|
| 絶対値表示 | 減算器 |
| 2 の補数表示 | 加算器　ビット変換と+1 |

となります。じつは、減算器よりはビット操作の方がずっと簡単です。そのような理由から、2 の補数表現が使われるようになったのです。

負の数の加算は、圧倒的に 2 の補数表示の方が簡単です。

このような理由から、回路構成の簡単な 2 の補数の表現が使われるようになりました。単に、補数といえば 2 の補数と考えてください。

この方法で負の数を表すことは、1970 年頃には常識となったようです。コンピューターの発明から 20〜30 年も要したことを考えると、これに到達するまでに、技術者はかなりの試行錯誤を行ったと思います。

驚いたことに、負の数をこの補数で表すアイディアは、パスカルが最初です。パスカルは、パスカリーヌという歯車式計算機を 1642 年に製作しています。減算、あるいは負の数の加算を加算器(歯車)で行うために、補数というものを考えたようです。

## 4 ビット反転と+1 加算の意味

正の整数 x として、-x を計算機の内部で表現する場合、

① x をビット反転する。
② +1 加算する。

の操作を行いました。式で表すと、

$$[-x] = (FF - x + 1)_{16} \tag{5}$$

でしたよね。ここでは、この操作の意味を調べます。結論から言うと、この操作は、符号反転の操作(-1 乗算)です。

それでは、もう一回この操作を繰り返しましょう。すると

$$\begin{aligned} &(FF - (FF - x + 1) + 1)_{16} = (x)_{16} \\ &= [x] \end{aligned} \tag{6}$$

となります。このことから、この操作は、符号反転であることが分かります。(5)式は、x の符号反転を示しており、(6)式は-x の符号反転を示しています。

-x のコンピューター内部表現を求める式は、(5)です。したがって、元の x を求めるためには、(5)の反対の演算をすればよく、

① 1 減算(-1 加算)する。
② ビットを反転する。

となります。式で表すと、

$$\begin{aligned} &(FF - (FF - x + 1 - 1))_{16} \\ &= (x)_{16} \\ &= [x] \end{aligned} \tag{7}$$

です。

しかし、元の表現を得るためには、(6)式の演算でも良いはずです。(7)式を変形すると(6)式が容易に導くことが出来ます。

これらのことから、以下の結論を導くことが出来ます。

- ①符号の反転②+1 加算の操作は、コンピューターの内部表現の符号反転である。
- この符号の反転と反対の操作①1 減算②符号反転は、まったく同じ操作である。

## 5 練習問題(2 の補数)

8 ビットで、以下の計算を実施してみよう。

(1) 次の負の 10 進数を補数で表現してみましょう。

例 $(-14)_{10}$ 00001110 ← 14 の 2 進数表現
11110001 ← ビット反転
11110010 ← +1 加算

Q1 $(-18)_{10}$

(2) 次の 2 の補数表現を 10 進数に変換してみましょう。

例 11101011
11101010 ← 1 減算
00010101 ← ビット反転
16+4+1=21 ← 10 進数に変換
$-(21)_{10}$

例の別解法 11101011
00010100 ← ビット反転
00010101 ← +1 加算
16+4+1=21 ← 10 進数に変換
$-(21)_{10}$

Q1 11001011

(3) 次の計算を、10 進数の演算を、2 の補数を使って計算してみましょう。

Q1 21-14 Q2 14-21