プログラミング基礎 第７回 ビット演算 補足資料と課題

1. n 進数

10 進数

- 日常的に使う（正の）整数は，各ケタの数値は〔 0 から 9 〕の〔 10 通り 〕の値をもつ．
- あるケタの数値が〔 10 〕になると，次のケタの数値が〔 1 増える 〕．
- **10** でケタが 1 つケタが**進**む**数**ということで，〔 10 進数 〕と呼ぶ．

2 進数

- 各ケタが持つ数値を最も少なくして，〔 0 と 1 〕の〔 2 通り 〕の値をもつものを考える．
- 例えば，0 の次は〔 1 〕，〔 1 〕の次（10 進数だと 2）は〔 10 〕となる．
- このように表現された数値を〔 2 進数 〕と呼ぶ．

ビット

- 1 ケタの 2 進数で表せる情報（〔　　0／1，　表／裏，HIGH／LOW，など　　〕）の単位を〔 bit（ビット）〕と呼ぶ．
- ビット数が増えると表現できる数値の範囲が広くなる．
  - 1 ビット … 〔 0 か 1 の 2 種類 〕
  - 2 ビット … 〔 00，01, 10, 11 の 4 種類 〕
  - 3 ビット … 〔 000, 001, 010, 011, 100, 101, 110, 111 の 8 種類 〕

8 進数

- 10 進数でケタが進むタイミング〔(　 9 → 10 　)〕は，2 進数ではケタが進むタイミングではない〔(　 1001 → 1010 　)〕ので，10 進数は中途半端である．
- 1 ケタの数が〔 　0～7 　〕の 8 種類の数値をもつ（2 進数だと〔 　3 ビット 　〕の数値に相当する）表現を，8 進数と呼ぶ．

16 進数

- 4 ビット（〔 　2 進数で 4 ケタで表せる数値 　〕）を 1 ケタに持つ表現を〔 　16 進数 　〕と呼ぶ
- 〔 　0～9 　〕，〔 　A～F 　〕の 16 種類の記号で 1 ケタの数値を表す．

10 進数	2 進数	8 進数	16 進数
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C

10 進数	2 進数	8 進数	16 進数
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	1 0000	20	10
17	1 0001	21	11
18	1 0010	22	12
19	1 0011	23	13
20	1 0100	24	14
21	1 0101	25	15
22	1 0110	26	16
23	1 0111	27	17
24	1 1000	30	18
25	1 1001	31	19

10 進数	2 進数	8 進数	16 進数
26	1 1010	32	1A
27	1 1011	33	1B
28	1 1100	34	1C
29	1 1101	35	1D
30	1 1110	36	1E
31	1 1111	37	1F
32	10 0000	40	20
33	10 0001	41	21
34	10 0010	42	22
35	10 0011	43	23
36	10 0100	44	24
37	10 0101	45	25
38	10 0110	46	26

課題 1

(1) 10 進数表現で 42，183 となる数値をそれぞれ，2 進数（8 桁），16 進数（2 桁）表現に直せ．
(2) 上記の二つの数値を，10 進数，2 進数，16 進数表現それぞれで足し算せよ．
(3) (2)の 2 進数表現による結果を 10 進数，16 進数表現に直せ．

C 言語での表現

- 10 進数 … 〔 0 以外の数字 〕で始める
- 8 進数 … 頭に〔 0 〕をつける
- 16 進数 … 頭に〔 0x 〕をつける

```
int x;
/* 下の代入式は全て同じ意味 */
x = 34;     /* 10 進数 */
x = 042;    /* 8 進数 */
x = 0x22;   /* 16 進数 */
```

図 1

printf での変換指定子

- 10 進数 … 〔 %d 〕
- 8 進数 … 〔 %o 〕
- 16 進数 … 〔 %x 〕

※int 型の場合はそのまま，short 型の場合は h（%hd など），long 型の場合は l（%ld など）を付ける．

2. ビット演算

ビットによる状態表現

- スイッチは，押されている（ON）か，押されていない（OFF）の 2 状態しかない．
- LED も，点灯（ON）か，消灯（OFF）の 2 状態しかない．つまり，〔 1 ビット 〕で表せる．
- 一方，int 型は〔 32 ビット 〕であり，約〔 40 億 〕通りの数値を表せるため無駄が多い．
- 32 ビットの数値は，1 ビットで表せる状態を 32 個分表せる．
- 例として，4 つの LED の状態を 4 ビットで表すことを考える．2 進数表現をしたときの各ケタの数値を LED1～4 に対応させる． ON のときを〔 1 〕，と OFF のときを〔 0 〕とする．
- 例えば，LED1 と LED4 は OFF で，LED2 と LED3 は ON だとすると，以下のように表せる．

LED4	LED3	LED2	LED1
0	1	1	0

- 以下の例では int 型の変数 a にこのような数値が入っていることを想定する．

論理積

- LED2 が ON である，つまり〔 2 ケタ目 〕の数値が〔 1 〕であることを調べたい．
- この条件は，〔 0110 〕で真であるだけでなく，〔 0111，1011，1111 〕などでも真である．
- 全ての真の場合を挙げるのは効率が悪い．他のケタとは無関係に 2 ケタ目が 1 であるという条件式を表現したい．このようなときに〔 論理積（AND） 〕を使う．
- 〔 a & b 〕という演算は，2 進数表現したときの各ケタごとに，〔 両方の値 〕が 1 であれば 1，〔 片方でも 0 であれば 〕0 になる演算である．
  - 例）〔 0110 & 0010 → 0010 〕
- 2 進数表現で 0010 の数は，16 進数で〔 0x2 〕であるから，条件式〔 (a & 0x2) != 0 〕は，LED2 が ON である場合に真となる．
  ※ & の優先順位は，!= よりも低いため，() で括る必要がある．

論理和

- LED2 を ON にする，つまり〔 2 ケタ目 〕の数値を〔 1 〕にしたいとする．
- 単純に 0010 を〔 代入 〕すると，LED2 は ON になるが，その他が OFF になってしまう．例えば，元々0100 であれば〔 0110 〕にしたいし，1100 であれば〔 1110 〕にしたい．このような場合に〔 論理和（OR 演算） 〕を使う．
- 〔 a | b 〕という演算は，2 進数表現したときの各ケタごとに，〔 少なくともどちらか片方の値が 1 〕であれば 1，〔 両方の値 〕が 0 であれば 0 になる演算である．
  - 例）〔 0100 | 0010 → 0110 〕
- 2 進数表現で 0010 の数は，16 進数で〔 0x2 〕であるから，〔 a = a | 0x2 〕とすれば，LED2 が ON になる．他の LED の状態は変わらない．

排他的論理和

- LED2 の状態を〔　反転　〕させたいときがある．つまり 2 ケタ目の数値が〔　0　〕なら〔　1　〕に，〔　1　〕なら〔　0　〕にしたいとする．
- 論理積と論理和を組み合わせて，条件分岐を利用すれば実現可能だが，〔　排他的論理和（XOR 演算）　〕を使うとより簡潔に書ける．
- 〔　a ^ b　〕という演算は，2 進数表現したときの各ケタごとに，〔　両方とも同じ値　〕であれば 0，〔　違う値　〕であれば 1 となる演算である．
  - 例）〔　0100 ^ 0010 → 0110　〕，〔　0110 ^ 0010 → 0100　〕
- 2 進数表現で 0010 の数は，16 進数で〔 0x2 〕であるから，〔 a = a ^ 0x2 〕とすれば，LED2 の状態が反転する（ON なら OFF に，OFF なら ON に）．

補数

- 〔　全ビット　〕を反転させたい場合は，〔　補数　〕を使う．
- 〔　~a　〕という演算は，2 進数表現したときの各ケタごとに，反転を行う（0 であれば 1，1 であれば 0 になる）．
- これまでは 4 ケタの 2 進数を例に出したが，int 型は実際には 32 ケタの 2 進数となるので，例えば 0010 を反転させると，〔　1111 1111 1111 1111 1111 1111 1111 1101 〕となる．

課題 2

(1) 右の表の空欄を埋めなさい（解答用紙の表も埋めること）．a と b は 1 ビットの変数とする．

(2) **論理和**の説明とは逆に，LED1 と LED3 を消灯し，他の LED はそのままにしておきたい場合はどうすればよいか説明せよ．

a	b	a & b	a | b	a ^ b	~a
0	0				
0	1				
1	0				
1	1				

3. 2 の補数

コンピュータの中ではすべての数が 0 と 1 の組み合わせで表現されている．

正の整数（0 を含む）は，これまでに説明した単純な 2 進数で表すことができる．では，負の整数はどのように表されているだろうか？コンピュータは，符号（-）を直接扱うこともできないため，やはり 0 と 1 だけの組み合わせで表現したい．

ここで，2 進数の桁（けた）あふれ（オーバーフロー）を考えたい．桁が多いと扱いづらいので，簡単のために 4 ケタで考える．4 ケタの 2 進数は，0000～1111 までの 16 通りの数字を表すことができるが，1111 の次はどうなるだろうか．1111 に 0001 を足すことを考えると，5 桁あれば 1 0000 と表すことができるが，今回は残念ながら 4 ケタ分しか扱えないため桁あふれがおきる．あふれた桁は単純に無視される．したがって，1111 + 0001 の計算結果は 0000 となる．

0001 は 10 進数で 1 と考えられる．1 と足して 0 になる数は-1 と考えられる．つまり，1111 を-1 と考えると都合がよいことがわかる．

別の例を見てみよう．0101 と足した結果が 1 0000 となるのは，どんな数か．

0101 の 0 と 1 を反転させた 1010 は，0101 と足すと 1111 となる．

0101 + 1010 → 1111 で 1111 + 0001 → 1 0000 だから，0101 + 1010 + 0001 → 1 0000

つまり，0101（10 進数で 5）の負の数は，1010 + 0001 の結果である 1011 と考えると都合がよい．

以上をまとめると，

「ある 2 進数の負の数は，もとの数字の 0 と 1 を反転させた数に 1 を足した数とする．」

となる．これを，2 の補数と言う．

課題 3

(1) 8 桁の 2 進数 0110 0101 の負の数（2 の補数）を，同じ 8 桁の 2 進数で表せ．

(2) 10 進数で-23 となる 8 桁の 2 進数（2 の補数）を示せ．

4. シフト演算

- LED の表示を左右にずらしたいときがある．
  - 例えば，0001 を左に順番にずらして，〔　0010，　0100，　1000　〕としたり，0110 を右にずらして〔　0011，　0001，　0000　〕にしたりすることである．
- このようにケタをずらす演算をシフト演算と呼び，〔　左　〕にずらす場合は〔　左シフト演算　〕，〔　右　〕にずらす場合は〔　右シフト演算　〕と言う．
- 〔 a << n 〕は，n ケタずらす左シフト演算である．
  - 例）〔 0001 << 1 → 0010 〕，〔 0001 << 2 → 0100 〕，〔 0001 << 3 → 1000 〕
- 〔 a >> n 〕は，n ケタずらす右シフト演算である．
  - 例）〔 0110 >> 1 → 0011 〕，〔 0110 >> 2 → 0001 〕，〔 0110 >> 3 → 0000 〕
- シフト演算の結果，あふれたケタは消滅する．
- 左シフトの場合，下位のケタには 0 が詰められる．
- 右シフトの場合，上位のケタには，符号なしの変数の場合は 0，符号ありの変数の場合は元々の最上位のケタの値が詰められる．

ビット演算の代入演算子

- 算術演算子の代入演算子（ +=，-= など）と同様，ビット演算も代入演算子がある．

	演算子の名前	意味
a &= b;	論理積代入演算子	a = a & b;
a |= b;	論理和代入演算子	a = a | b;
a ^= b;	排他的論理和代入演算子	a = a ^ b;
a <<= n;	左シフト代入演算子	a = a << n;
a >>= n;	右シフト代入演算子	a = a >> n;

課題 4

(1) 右のプログラムにおいて， i の値が 8 である場合について，以下の問いに答えよ．
  ① 31 - i はいくつになるか答えよ．
  ② 1 << 31 - i を 32 ケタの 2 進数表現にしたとき，どのケタの数字が 1 になるか答えよ．
  ③ n & (1 << (31 - i)) が 0 以外の数字になるとき，n は 2 進数表現でどのケタが 1 であるか，理由も含めて答えよ．

(2) 右のプログラムで，n の値を-11 にして実行した結果を示し，n の値が 10 の場合と比較してどうなっているか説明せよ．

```
#include <stdio.h>

int main (void) {
  int i, n = 10;

  for (i = 0; i < 32; i++) {
    if ((n & (1 << (31 - i))) != 0)
      printf ("1");
    else
      printf ("0");
    if ((i + 1) % 4 == 0)
      printf (" ");
   }
   printf ("\n");

   return 0;
}
```