第6章 ビット操作

ビット操作とは／排他的論理和の有用性／シフト演算／ビット操作の応用

この章では，ビットの情報を扱う手法を解説します．アセンブラ言語の処理では機械処理の中心がビット処理なので，あからさまに処理のプロセスが見えていたような気がします．しかしC言語になると中身が見えないような気がして少々不安な面もあります．そこで，ビット単位演算子と呼ばれる演算子を中心に，その機能と利用の仕方を具体的な例題をもとに解説します．

また通常の応用では，数値そのものの処理が多いのですが，制御系のプログラムでは，ディジタル的なスイッチのon/off入力やランプなどへの出力があります．ビットの操作は，C言語を制御系に応用した場合には欠かせないものです．しかしここでは具体的なスイッチやランプを用意するわけにもいきませんので，ビット操作を机上でできる例題にしています．

6.1 ビット操作とは

C言語でいうビット操作演算とは，アセンブラ言語では論理演算などと呼んでいることが多い命令による操作です．C言語ではビット単位演算と呼んでいます．C言語にも論理演算子はあるのですが，これは結果一つだけを真か偽か判定するもので，アセンブラでいう論理演算とは内容が異なります．C言語のビット演算は，ビットごとの論理演算という解釈ができます．

そしてC言語では，ビットごとにシフトをするシフト演算子もビット演算子の仲間です．コンピュータが扱っている情報がビットであるかぎり，どんな命令により操作しても，動いているのはビット単位の情報ですから，全ての命令がビットを操作はしているということはできますが，明示的に結果がビットの情報として表されているものをビット操作命令あるいは演算子と呼んでいるだけのことです．

C言語においては，**表6.1.1**に示すビット単位の演算子(bitwise operator)があります．このうち，ビット反転演算子

表6.1.1　ビット単位の演算子

演算子名	演算子	記述例	意　味	
ビット反転演算子	~	~a	aを否定	NOT
ビット積演算子	&	a & b	aとbをビット論理積	AND
ビット和演算子	|	a | b	aとbをビット論理和	OR
ビット差演算子	^	a ^ b	aとbの排他的論理和	XOR
右シフト演算子	>>	a >> n	aをnビット右シフト	
左シフト演算子	<<	a << n	aをnビット左シフト	

だけは単項演算子と呼ばれ，一つの変数にだけ処理を行います．ビット和，積，差などの演算子は2項演算子で，二つの変数に対する処理を行います．

6.1.1　ビット反転演算子

正式にはビット反転演算子(bitwise inversion operator)，ビット補数演算子(bitwise complement operator)あるいはたんに補数演算子といいますが，一般的にいう否定の演算を施す演算子です．否定ですからNOT演算とも呼びます．

演算子は変数の前に~(チルダ: tilde)を付けたものです．内容を**表6.1.2**に示します．変数aの内容が1ならば，~aは0，aが0ならば~aは1です．今後このような図で説明しますが，これは論理代数を勉強したときの，真理値表と同じものです．

表6.1.2　ビット反転演算

a	~a
0	1
1	0

```
   10100011   a
~)
   01011100   ~a
```

図6.1.1　ビット反転演算の例

ビット反転演算の対象は整数の変数です．変数の型はchar型で8ビット，int型で32ビット(16ビットの処理系もある)などです．それぞれのビットごとに，**表6.1.2**の処理が行われます．ビット演算は算術演算のように，桁上がりするなど，他のビットに干渉することは一切ありません．

もう少し具体的な例で見てみます．**図6.1.1**に示すように，char型の変数aに0xa3という値があったとします．この場合~aは0x5cを表すことになります．0xa3を2進の数に直すと，10100011です．この1と0を全て反転した形は，01011100ですから，16進表記で0x5cというわけです．

~aの内容は，もとの変数の内容をビット反転した内容になりますが，C言語では実際にこの値を変数に代入しなければ，実際の数値にはなりません．そこで，別の変数bに代入する場合には，

```
b = ~a;
```

などの文となります．これでbにはaの内容の反転が代入されることになります．もちろん同じ元の変数(器)に代入することもできますので，

```
a = ~a;
```

ということもあります．さらに具体的な利用例は後述します．

■ 1の補数

ビットを反転するということは，2進数で1の補数を得たことと同じです．この場合の1というのは2進数の1のことです．複数桁(ビット)ある場合には，各桁が1ということで，各桁がすべて1の数値は10進数で表現した－1のことです．つまり，0xffffffffは－1ですから，ビット反転の~aは

```
(-1-a)
```

または(0xffffffff-a)と同じ結果になります．

6.1.2　ビット和演算子

論理代数を勉強したときと同じく，論理和もあります．C言語ではビット和演算子(bitwise inclusive OR operator)といいます．二つの変数の内容を，ビットごとに論理演算をして結果を出す演算です．C言語での演算子は，|(vertical bar：縦線)です．

ビット和演算子は2項演算子です．二つの変数または定数とのビットごとの論理和です．結果はまた変数に代入して得られます．**表6.1.3**にビットごとの和演算の内容を示します．具体的な内容はここでもchar型の変数で説明しますが，intやshort，longなどの型でも同様です．**図6.1.2**に変数aと変数bの内容をビット和演算した例を示します．いま変数aの内容が0xa3，変数bの内容が0x0eの例です．図のように数を2進で表記し，前掲の**表6.1.3**の内容で，ビットごとに論理和すれ

表6.1.3　ビット和演算

a	b	a | b
0	0	0
0	1	1
1	0	1
1	1	1

```
   10100011   a
|) 00001110   b
   10101111   結果
```

図6.1.2　ビット和演算の例

ば結果が得られます．結果を別の変数cに代入するのであれば，

```
c = a | b ;
```

と表記します．

もちろん自分自身にも演算の新しい結果を代入することもできますので，

```
a = a | b ;
```

という表記もできます．このときaの内容は新しい内容に書き換えられてしまいます．また2項演算子には，複合代入演算子もあって，

```
a |= b ;
```

という表記も許されます．

ビット和演算は，2数のビットのいずれかが，1のところが結果においても1になる演算です．そのために特定のビットを強制的に1にするような場合に利用されます．

6.1.3　ビット積演算子

前項の論理和に対する論理積の演算です．C言語ではビット積演算子（bitwise AND operator）には&（ampersand）を利用します．論理演算しとしても連続2個の&を使いますが，これとは意味がまったく違いますので注意してください．

ビットごとの演算の結果を**表6.1.4**に示します．論理代数のときの論理積（AND）と同様です．ここでもビットごとの演算で，桁上げなどは発生せずに他のビットへの干渉はありません．たとえばchar型の変数aが0xa3で，bが0x0fのとき，両者をビット積演算すると，**図6.1.3**に示すように，結果は0x03となります．

積演算は両者の1のところだけが1として残ります．残したい部分を1にしてANDします．逆にいえば，消したいところを0にします．積の操作はこのように抽出ができますので，通常マスク（mask）するなどといいます．不必要なビットを消去して，必要なビットのみを残すことができます．

この演算子も2項演算子で，

```
c = a & b;
```

などと利用します．また固定パターンでマスク操作をする場合には，

```
a &= 0x00ff;
```

などとビット積代入演算子としても利用できます．この場合は前述のように，下位8ビットのみが抽出されます．

表6.1.4　ビット積演算

a	b	a & b
0	0	0
0	1	0
1	0	0
1	1	1

```
  10100011   a
&)00001111   b
  00000011   結果
```

図6.1.3　ビット積演算の例

6.1.4　ビット差演算子

排他的論理和のビット演算はC言語ではビット差演算と呼びます．通称EX-OR（Exclusive OR）です．演算の論理を**表6.1.5**に示しますが，排他的論理和が反一致論理とも呼ばれているように，両者のビットが一致していないときに真になる論理だからです．次節も参照してください．

ビット差演算子（bitwise non-equivalence operator）には，^（circumflex accent）を使います．これは英記号を流用しただけでとくに意味はありません．これも当然ながら2項演算子で，変数または定数2個を演算子で結び演算します．**図6.1.4**に例を示します．変数aの内容が0xa3で，変数bの内容が0x0fのときには，結果は0xacとなります．ビット差演算の詳細については，次節でまとめて解説します．

表6.1.5　ビット差演算

a	b	a ^ b
0	0	0
0	1	1
1	0	1
1	1	0

```
  10100011   a
^)00001111   b
  10101100   結果
```

図6.1.4　ビット差演算の例

6.2　排他的論理和の有用性

前述までのビット反転，ビット和，ビット積そしてビット差演算は，論理代数の中で最も基本的な演算です．その中でもビット差演算と呼ばれる排他的論理和演算はとくに有用なビット演算です．プログラムの中でもよく利用します．論理の学習の鍛錬をかねて，排他的論理和の有効性や利用例について補足しておきます．**図6.2.1**に排他的論理和のシンボルと真理値表を示します．この真理値表をいろいろな見方で見直してみます．

6.2.1　一致を検出

排他的論理和は別名「反一致回路」と呼ばれることは前述しました．出力Yは，入力ABが一致している00や11では偽で，01や10のときに真となっています．Yを否定(反転)すれば，一致回路ということもできます．

排他的論理和はディジタル値の一致を検出することができるので，二つの値が等しいか否かの判定に利用できます．ハードウェアでは，コンピュータが出力したアドレスと，メモリや入出力装置に与えられたアドレスが等しいかどうかの検出，つまりアドレス・デコーダなどにも利用されます．

6.2.2　変化を検出

排他的論理和の論理を時系列的に使うと，また有効な面が出てきます．Aに時間的に過去の情報を入力し，Bには現在の情報を入力したとします．過去の情報は変数に確保しておいて，新しい情報を入力したときに，ビット差演算をするということになります．**図6.2.2**にその例を示します．

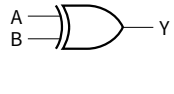
図(**a**)は排他的論理和の真理値表です．出力が1となっているのは，網かけで示したとおり，入力ABが01のときと10のときです．いま，Aは過去の情報，Bは現在の情報としましたから，出力が1となったのは，過去と現在の状態が異なるということです．つまり状態が変化したということになります．offしていたスイッチがonに変化した，あるいはonしていたスイッチがoffになったという状況を検出できます．この機能はハードウェアで使うよりも，プログラムの中で使うことが多い機能です．

6.2.3　変化の方向を検出

変化を検出するだけでなく，変化の方向も知ることができます．**図6.2.2**(**b**)は，排他的論理和をして得られた結果と，入力Aの論理積をした真理値表です．ここで結果Zが1となっているのは，網かけで示した部分だけです．この行が何を意味するのかといえば，入力ABが10のときですから，過去に情報が1であって，現在は0である変化をしたところです．

一方，同**図**(**c**)は，排他的論理和をして得られた結果と，入力Bの論理積をした真理値表です．ここで結果Zが1となっているのは，網かけで示した部分だけで，この行は，過去の情報が0で，現在の情報が1と変化をした行です．

このように，新旧の情報を排他的論理和した結果と，その入力のいずれかをさらに論理積することによって，変化の方向も知ることができます．スイッチが押されたときだけを検出し，離されたときは無視するという使い方です．この機能もプログラミングの上で使うことの多い機能です．



Y＝A・$\overline{B}$＋$\overline{A}$・B

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

図6.2.1　排他的論理和

(**a**)AとBの排他的論理和

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

(**b**)YとAの論理積

Y	A	Z
0	0	0
1	0	0
1	1	1
0	1	0

(**c**)YとBの論理積

Y	B	Z
0	0	0
1	1	1
1	0	0
0	1	0

図6.2.2　変化の検出