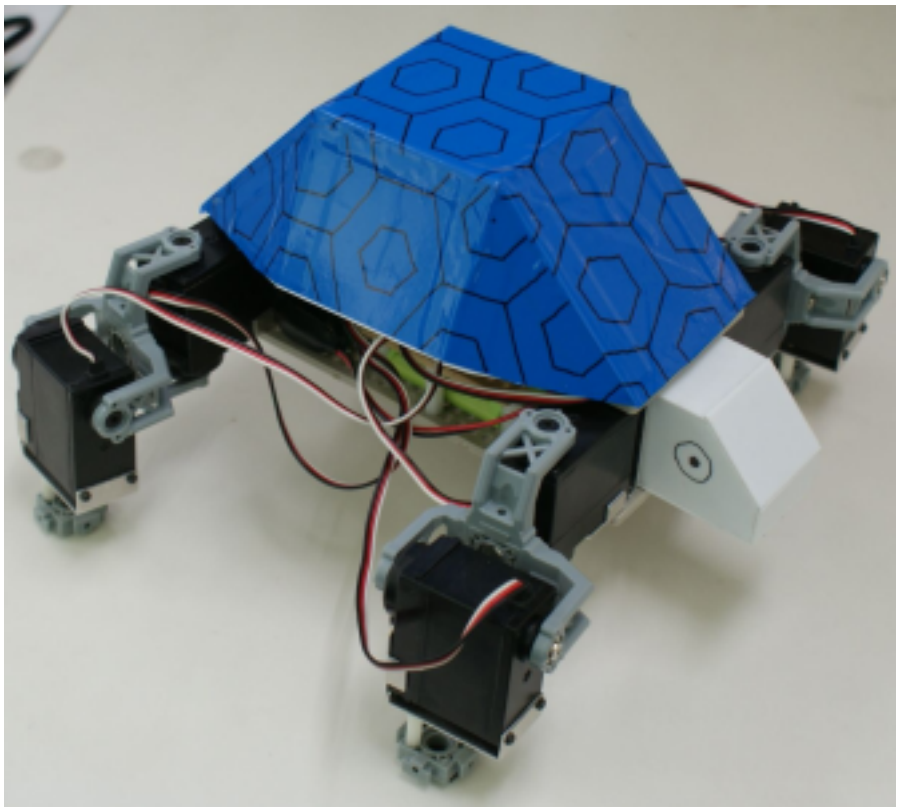
歩行ロボットを用いたマイコン実習(目次)

より移植性の高いプログラムを目指して

東北職業能力開発大学校 電子情報技術科

Tuesday, July 8, 2014



Yasuji Ono

製作・著作　小野泰二

目 次

0 初めにお読みください

0.1 この教材の目的・目標

本教材は、組込みマイコンのプログラミングを体験するとともに、歩行ロボットの動かし方を理解することを目標にしています。

対象とするロボットとして、近藤科学 (株) の KRS-788HV というモータを使ったロボットを考えています。

ただしモータが PWM で制御されるロボットでしたらプログラムの変更はほとんど必要ないでしょう。

プログラムとしては、

できるだけ少ない修正で異なるマイコンに移植できる

ことを目標にしています。

そのために、機能を関数化し、ハードウェアの依存性をできるだけ一か所に集めるように心がけました。また、関数化するに当たり、必要のないレジスタを設定しないということにも注意しました。

移植性を高めるために、開発環境に依存する部分はできるだけ使わないように心がけました。あらかじめ用意されている標準関数は開発環境が変わると使える保証がないので、できる限り一からプログラムを書くようにしました。最初は大変でしょうが、一度用意してしまえば再利用がきくので大変便利だと思います。

この教材の目的・目標：

- 組込みマイコンのプログラミングを体験
- 歩行ロボットを動かす
- 異なるマイコンに移植しやすいようにプログラムする

0.2 この教材の想定する読者

このテキストは職業能力開発大学校の専門課程、電子情報技術科 1 年次後期の学生と同等な知識を持つ読者を想定しています。

具体的な知識としては、C 言語の基本的なこと (ポインタ、構造体、共用体あたりまで) を理解していることが望ましいです。ただし、テキストの中でもポインタ、構造体、共用体、ビットフィールドについては簡単に説明しています。このテキストの内容で理解できないようでしたら参考文献などを勉強すると良いでしょう。

データ型、制御構造 (if,for,while など)、配列、関数に関する基本的な事柄は、前提としています。

これらの知識のない読者は、参考文献を勉強したり、必要に応じて参照したりしてください。

論理演算などに関しても、基本的なことは理解していることを前提に説明しています。こちらは少し詳しく説明しています。本テキストの内容を勉強して、分からないようでしたら参考文献等を勉強し直してください。

組込みマイコンに関しては、初心者を対象に考えています。

パソコンで C 言語の簡単なプログラムはできるけれども、マイコンというものを使ったことがない人が対象と言っても良いでしょう。

この教材を読むのに必要な知識：

- C 言語の基本

0.3 学習の流れ

本教材は、組込みマイコンの初心者を対象にし、とにかく実際にやってみて体験することを重視しています。

開発環境で用意されている設定等については、プログラムを作るのに必要最小限の説明にとどめ、スタートアップルーチンやスタックなどの詳細には触れません。本教材終了後に勉強を進めるとよいでしょう。

また、開発環境に関しては、平成 26 年 4 月時点の東北職業能力開発大学校電子情報技術科の環境を前提に書いています。インストールが必要な場合や開発環境が異なる場合には、適宜マニュアルなどを参照してください。

ロボットを動かすにあたっても、ロボット工学などの理論は用いません。ロボット用のモータをマイコンからいかに動かすかというレベルにとどめています。

市販のロボットキットを動かすだけではなく、自分でマイコンのプログラムをしてみたいという方のために、その第一歩となる教材という位置づけになります。

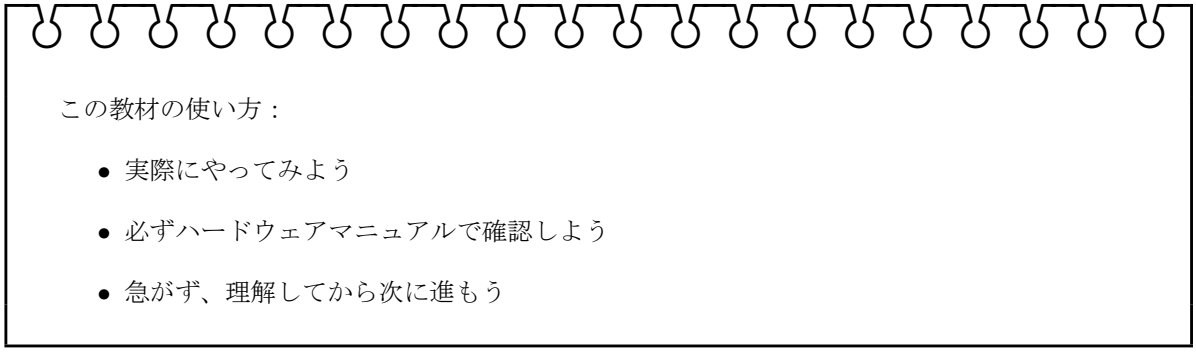
本教材の学習にあたっては、必ずハードウェアマニュアルで確認してください。本教材の範囲を超えて学習しようとした場合に、ハードウェアマニュアルは必須です。多くの機能の中からどのようにして目的の機能を実現しているかを確認しながら進めるようにしてください。

本テキストには課題も載せてあります。理解度を確認するためにも、自分でやってみてください。一部の解答例は付録として最後に載せておきました。必ずしもこの解答例だけが正解ではありません。また、最良のものであるともいえません。より良いプログラムになるように充分に考えてみてください。

さらに、自分で課題を考えてプログラミングしてみることをお勧めします。

分からなくなったり、忘れてしまった場合には、先を急がず前に戻って勉強しなおしましょう。

本教材によって、より多くの人に組込みマイコンの世界に興味をもっていただければ幸いです。



この教材の使い方：

- 実際にやってみよう
- 必ずハードウェアマニュアルで確認しよう
- 急がず、理解してから次に進もう

0.4 表示上の注意

プログラムのソースリスト等において、バックスラッシュ \ は円マーク ¥を意味しています。著者の組版ソフトに対する理解不足のため、現状ではこのような表示になってしまいました。ご了承ください。

1 組込みマイコンとは

1.1 組込みマイコン

近年、非常に多くの電化製品がコンピュータによって動いています。このように電化製品などに組込まれて、機器の制御をおこなうコンピュータのことを、「組込みマイコン」もしくは、ただ単に「マイコン」と呼んでいます。マイコンはマイクロコンピュータ (Micro Computer) もしくはマイクロコントローラ (Micro Controller) の略です。

本テキストでも「組込みマイコン」を、「マイコン」と呼ぶこともあります。

大まかに言ってしまえば、私たちが普段目にしているもののうち、パソコン以外のコンピュータは組込みマイコンであると言っても良いでしょう。

様々な機器の制御に専用の回路を作るよりもマイコンを用いたほうが、小さく安くでき、プログラムの変更により機能追加などもおこなえるというメリットがあります。このため、現在ではきわめて多くの機器がマイコンを使って動作しているのです。

マイコンにおいて重要なことは、必要な機能を備えていること以外に、

- 安価であること
- 小さいこと
- 消費電力が少ないこと (発熱が小さいこと)

があると考えられます。

パソコン以外のコンピュータは組込みマイコン

1.2 組込みマイコンを探してみよう

では、身の回りにある組込みマイコンを考えてみましょう。

機器に組込まれているために組込みマイコンを直接見ることは困難です。しかし、電気製品のほとんどは組込みマイコンによって制御されているのです。

以下に例を挙げてみましょう。

- 冷蔵庫
- 携帯電話、スマートフォン
- テレビ
- 炊飯器
- 自動販売機
- デジタルカメラ
- エアコン
- ポット
- 各種リモコン

などなど …

少し身の回りを見回しただけでも、マイコンによって制御されていそうなものは、たくさん見つかります。パソコンを一人で五台も十台も持っている人は珍しいでしょうが、組込みマイコンが家に十個以上あるというのは別に珍しいことではありません。自動車などは、高級車ですと、百個ぐらいの組込みマイコンが使われていることもあるようです。

上であげた家電製品のリストを見てみると、ずいぶん複雑そうなものから、比較的簡単な機能のものまで様々だということに気付きます。

しかも、同じ冷蔵庫でも大きさも機能も製品ごとに違っています。

表示機一つを取ってみても、ある機器は液晶表示機であり、あるものは LED だけで十分であったりします。安価なリモコンのように、表示機そのものが存在しないものすらあります。組込みマイコンにつながれているものは千差万別なのです。

このように、さまざまな機器を組込みマイコンは上手に動かしているわけです。

組込みマイコンはすべてプログラムで動いています。しかも新しい製品のためには新しくプログラムを作らなければなりません。組込みプログラマはますます必要になってくるのではないでしょうか。

さまざまな電気製品用のプログラムを行うにあたっては、一度作ったプログラムを他の家電製品にも使いまわせることは重要です。すでに正常に動作しているプログラムの一部を使うことで、開発期間を短くできるうえに、プログラムのバグが発生する可能性を少なくすることができるからです。

組込みマイコンが制御する機器は、機能も大きさも千差万別
移植性が重要

1.3 安く小さくということ

コンピュータにおいて中心的な処理装置を CPU(Central Processing Unit) といいます。コンピュータの頭脳に当たります。命令を読みこみ、解析し、実行します。家電製品などを制御するのには、CPU だけでは足りません。外部の情報を取り込むための A/D 変換の機能や、時間をはかるためのタイマなどの周辺機能が必要になってきます。これらの周辺機能をもつ IC などを CPU と接続することもできるのですが、それでは値段もかさみますし、大きくなってしまいます。必要な周辺機能を一つの IC の中に CPU と一緒に組込んでおくと、より小さく安い組込みマイコンができます。

必要のない機能は実装しないのがよいのですが、製品ごとに専用のマイコンを作っていたのでは、やはり高価になってしまいます。

必要な周辺機能は機器によって異なります。そこで、良く使う周辺機能をいくつか多めに用意しておき、必要に応じて使う機能を設定するという方法が考えられます (図 1.1)。

図 1.1 良く使う周辺機能から選択する

専用でもなく、完全に汎用でもなく、同規模の機器に必要な周辺機能を持ったマイコンを用意しておくのです。そうして、利用する際にその中から必要な機能だけ選びだすことになります。用意する機能が多ければ多いほど、一つのマイコンで多くの機器に対応できます。

しかし、周辺機能をたくさん入れれば入れるほど値段もかさみますし、機能を多く利用するためにはマイコンの端子 (脚) がその分必要になります。脚の間隔は決められているので、多くの脚を出すためには大きなものにならざるをえません。そこで、安価で小さなものを作ろうとすると周辺機能の個数にも制限が出てくるのです。

ところが、すでに述べたように、電子機器に必要な周辺機能というのは、機器によって違ってきます。ある機器にはネットワークの機能が複数必要かもしれませんし、他の機器はネットワークの機能は必要ないけれども A/D 変換は 8 本必要かもしれません。タイマが何本も必要な機器もあれば、必要のない機器もあるでしょう。そこで、用意しておく周辺機能もいくつかのバリエーションが必要になってきます。ラ

インアップを複数用意しておくことになるわけです。こうして、同じ CPU に違う周辺機能を持ったマイコンがいくつか存在することになります。

これらの中から、自分が開発する機器にあったマイコンを選び出し、用意されている周辺機能の中から必要な機能を設定することになります。

必要な機能を設定する際には、通信でしたら速度だとか、タイマでしたら計測時間などの設定も必要になってきます。

このように、マイコンを使うためには、周辺機能のさまざまな設定を行う必要があるのです。この設定はレジスタというものにプログラムを用いて値を代入することによって実現します。

1.4　周辺機能とレジスタ

では、あるマイコンを用意したとして、必要な周辺機能はどのようにして設定するのでしょうか？

本テキストで利用するルネサスエレクトロニクス株式会社のマイコンでは、内部 I/O レジスタというものを設定することで希望の周辺機能を実現することができます (図 1.2)。

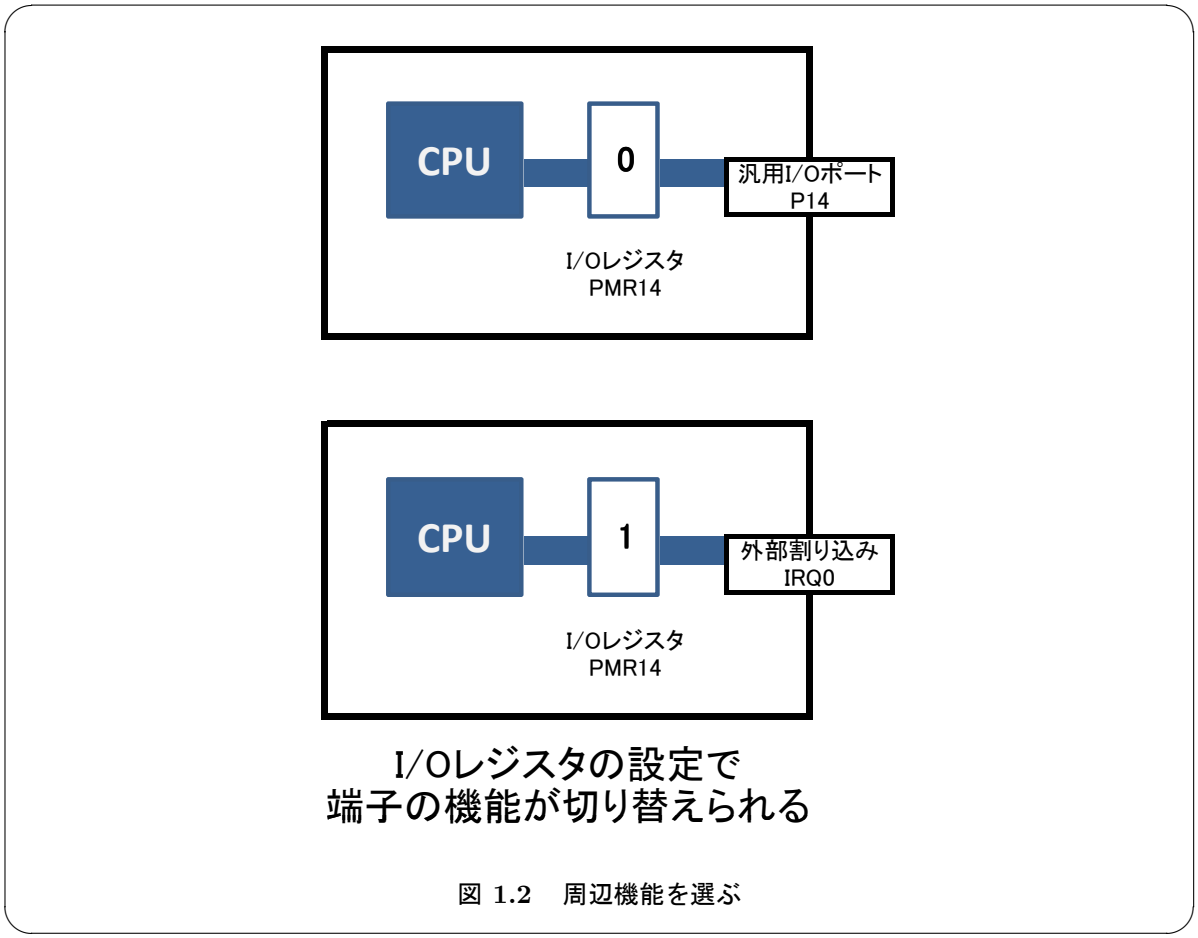


図 1.2　周辺機能を選ぶ

では、どのような周辺機能があり、それを実現するためには、どのような設定をすればよいのでしょうか？

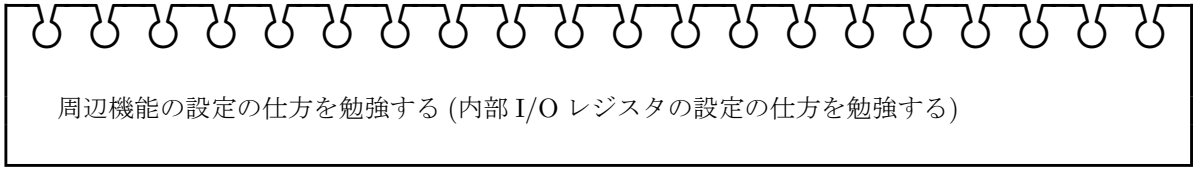
実は、周辺機能はさまざまで、機能によっては設定する内容も豊富です。そして、それらはすべてハードウェアマニュアルに説明されています。

しかし、ハードウェアマニュアルはすべての機能が説明されている反面、初心者には読みこなすのは大変です。

本テキストでは、具体的な例を用いて、希望する周辺機能を実現するために、内部 I/O レジスタをどのように設定するかということを学習します。

利用する周辺機能も、比較的良く使われるものばかりです。そして、それらの周辺機能は歩行ロボットでも利用されているのです。

本テキストとともに、ハードウェアマニュアルを参照しながら学習することで、異なる機能を実現しようとする場合や、マイコンが替わった場合でも同様の設定が自分でできるようになることと思います。



周辺機能の設定の仕方を勉強する (内部 I/O レジスタの設定の仕方を勉強する)

2 ビット演算

前の章では、マイコンを利用するのには I/O レジスタというものに値を設定する必要があるという説明をしました。レジスタ 1 つには 0 か 1 の値を設定することができます。2 つ (0 または 1) の選択肢からどちらか一方を選べる能力のことをビット (bit) と言います。レジスタ 1 つは 1 ビットということになります。

マイコンのレジスタは 8 ビット単位で操作します。8 ビットを 1 バイト (byte) といいます。

I/O レジスタは 1 バイト単位で扱いますが、周辺機能を選択する際には、1 ビット単位で値を設定する必要があります。したがって、バイト単位のデータを用いて、あるビットには値を設定し、他のビットには値を設定しないという操作が必須になってきます。

この章では、このような演算を行う方法について勉強していきましょう。

1 バイトのデータを用いて、1 ビット単位の設定をする方法を学ぶ。

2.1 論理演算

論理演算 (Logical Operation) はブール演算 (Boolean Operation) とも呼ばれ、真 (1) もしくは偽 (0) の二通りの元に対して、行われる演算です。

論理演算には入力が一つのものと二つのものがあります。いずれも、出力は一つです。それ以外のものは、これらを組み合わせて構成します。

ここでは、論理演算の例として、組込みプログラミングで良く利用する、論理積 (AND)、論理和 (OR)、否定 (NOT)、排他的論理和 (XOR) について学びましょう。

論理積 (AND)、論理和 (OR)、排他的論理和 (XOR) は入力が二つ、否定 (NOT) は入力が一つです。

すべての入出力の結果を表にしたものを真理値表 (Truth table) といいます。論理積 (AND)、論理和 (OR)、否定 (NOT)、排他的論理和 (XOR) について説明し、真理値表を確認しておきましょう。

2.1.1 論理積 (AND)

論理積 (Logical conjunction) は、入力値がいずれも 1 の場合に 1 を出力し、それ以外の場合には 0 を出力します。

表 2.1 論理積の真理値表

入力 1	入力 2	出力
0	0	0
0	1	0
1	0	0
1	1	1

2.1.2 論理和 (OR)

論理和 (Logical disjunction) は、入力値のいずれかが 1 の場合に 1 を出力し、いずれも 0 の場合には 0 を出力します。

表 2.2 論理和の真理値表

入力 1	入力 2	出力
0	0	0
0	1	1
1	0	1
1	1	1

2.1.3 否定 (NOT)

否定 (Logical Negation) は、入力値と反対の値が出力されます。

表 2.3 否定の真理値表

入力	出力
0	1
1	0

2.1.4 排他的論理和 (XOR)

排他的論理和 (Exclusive disjunction) は、二つの入力値が異なる場合に 1 を出力し、同じ場合には 0 を出力します。

表 2.4　排他的論理和の真理値表

入力 1	入力 2	出力
0	0	0
0	1	1
1	0	1
1	1	0

上記の論理演算は組込みプログラミングを勉強する際の基本ですから、入力に対する出力が即座に答えられるようにしてください。また、AND はどちらか片方が 0 ならば出力は 0、OR はどちらか片方が 1 ならば出力は 1 という覚え方も重要です。

演習 2.1- 1

下の図のような 100 マスの計算を、できるだけ短時間でできるようにしましょう。とりあえずの目標は 30 秒です。

排他的論理和(XOR)問題　名前　　時間　　正解数

	1	0	0	0	0	0	0	1	0	1
0										
1										
1										
0										
1										
1										
0										
1										
1										
1										

AND、OR、XOR の 100 マス計算用プリントを作成する EXCEL のファイルを、「ビット演算演習.xlsm」という名前で用意しておきました。これを利用して充分に論理演算に慣れてください。

2.2 論理演算をプログラムする

ここでは、今まで説明してきた論理演算を C 言語のプログラムで確認していきます。マイコンのプログラムでは結果の表示が大変ですので、パソコン上で確認してみましょう。

Borland C++ Compiler 5.5 など無償版のコンパイラなどを使って確認してみてください。

2.2.1 論理積 (AND) のプログラム

論理積 (Logical conjunction) は、C 言語のプログラムでは&を使います。

以下は、論理積の真理値表を確認するプログラムです。

02211and01.c

```
 1  /*************************************************************************/
 2  /*                                                                       */
 3  /*  DESCRIPTION :AND の演算                                              */
 4  /*  CPU TYPE    :PC                                                      */
 5  /*  NAME        :Ono Yasuji                                              */
 6  /*                                                                       */
 7  /*************************************************************************/
 8
 9  #include<stdio.h>
10
11  int main(void)
12  {
13
14    printf("0 \& 0 = %d\n", 0&0);
15    printf("0 \& 1 = %d\n", 0&1);
16    printf("1 \& 0 = %d\n", 1&0);
17    printf("1 \& 1 = %d\n", 1&1);
18
19    return(0);
20  }
```

End Of List

実行結果 (Borland C++ Compiler 5.5)

>bcc32 02211and01.c ↵

```
Borland C++ 5.5.1 for Win32 Copyright (c) 1993, 2000 Borland
02211and01.c:
Turbo Incremental Link 5.00 Copyright (c) 1997, 2000 Borland
```

>02211and01.exe ↵

```
0 & 0 = 0
0 & 1 = 0
1 & 0 = 0
1 & 1 = 1


>
```

✍ 課題 2.2.1 (提出) その他の論理演算のプログラム

上記のプログラム 02211and01.c と同様にして、論理和、否定、排他的論理和の真理値表を出力するプログラムを作ってみてください。

論理和は|を、否定は!を、排他的論理和は^を用いてプログラムすることができます。

論理和のプログラムを e02211or01.c、否定のプログラムを e02211not01.c、排他的論理和のプログラムを e02211xor01.c という名前で保存してください。

2.3 2 進数，16 進数

ここでは、組込みプログラミングを行う際に重要である、2 進数と 16 進数に関して説明をします。

マイコンの機能を設定する際に重要なのは、あるビットの値が 0 であるのか 1 であるのかということです。このような 0 と 1 の羅列は、2 進数で表現することができます。

組込みマイコンのプログラムでは、データは基本的に 1 バイト単位で扱います。C 言語では 2 進数が直接扱えませんし、見易さの点でも 16 進数で表現してプログラムすることが必要になります。

われわれは日常的に 10 進数を利用しています。

このように、組込みプログラミングを行う際には、10 進数、2 進数、16 進数を利用しなければなりません。これらの間の対応を充分に理解している必要があります。

10 進数、2 進数、16 進数の対応が即座に答えられるように、学習していきましょう。

2.3.1 2 進数 (Binary Number)

われわれが数を数える際には、0 から 9 までの数字を使い、9 の次は一桁増やして 10 としています。

どの桁も 0 から 9 までの 10 種類の数字を使って数値を表現しているわけです。このようにひとつの桁に対して 10 種類の数字を使って表現する数値を 10 進数と言います。

10 進数では、隣り合う上位 (右) の桁に対して、下位 (左) の桁の 10 倍の意味を持たせる位取り記数法を用います。つまり、10 進数で 1024 は、

$$1024 = 1 \times 10^3 + 0 \times 10^2 + 2 \times 10^1 + 4 \times 10^0$$

という意味を持つわけです。

マイコンの機能を利用するためには I/O レジスタなどの記憶領域に 1 ビット単位で値を設定する必要があります。1 ビットに設定する値は 0 もしくは 1 です。

例えば、PBDR という名前が付けられた 1 バイトの記憶領域に、$(01101001)_2$ という値を設定することを考えてみましょう。ここで、右下に 2 と書いたのは 2 進数であることを意味しています。アセンブラ言語では直接この値を扱えるものもあるのですが、C 言語では扱えません。そこで、これを 10 進数などで表現する必要があります。

10 進数に書きかえるにあたっては、$(01101001)_2$ という値を、8 桁の数値とみなす必要があります。各桁は 0 か 1 の 2 種類の値をとるので、これを 2 進数の 8 桁とみなすことができます。つまり、2 進数の位取り記数法を用いているとみなすわけです。10 進数の場合に上位の桁は下位の桁の 10 倍の意味を持ちましたが、2 進数の場合には、上位の桁は下位の桁の 2 倍の意味を持つことになります。したがって、$(01101001)_2$ は、

$$\begin{aligned}(01101001)_2 &= 0 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ &= 64 + 32 + 8 + 1\end{aligned}$$

$$= \quad 105$$

となります。

ここで 2 の 7 乗までの 10 進数との対応をみると、表 2.5 のようになります。

表 2.5　2 進数と 10 進数の対応

2 進数	10 進数	
1	1	2^0
10	2	2^1
100	4	2^2
1000	8	2^3
10000	16	2^4
100000	32	2^5
1000000	64	2^6
10000000	128	2^7

つまり、$(01101001)_2$ を設定するために、C 言語では 105 を指定しても良いわけです。しかし、2 進数と 10 進数の対応では、桁が多くなってくると計算が大変です。このような場合に便利なのが 16 進数です。

2.3.2 16 進数 (Hexadecimal Number)

2 進数と 10 進数の対応では、2 進数の桁と 10 進数の桁の間の対応が簡単ではありません。そこで、2 進数の複数の桁をひとまとまりにして、対応を簡単にすることが考えられます。

1 バイトを基本としていることから、2 進数 8 桁をひとまとまりにすることも考えられるのですが、この場合 1 桁に対して 256 種類の表記が必要になってしまいます。半分の 4 桁であれば 16 種類ですみますし、2 進数 8 桁も、16 進数 2 桁で表記できます。

そこで、C 言語のプログラムでは、2 進数であらわされたデータを扱う際に、16 進数が良く使われるのです。

16 進数では 0 から 9 までの数字と、A から F までのアルファベットで 1 桁を表記します。

2 進数 4 桁と 16 進数 1 桁との対応をみると、**表 2.6** のようになります。ここでは、対応する 10 進数も入れておきました。

表 2.6　2 進数 4 桁と 16 進数 1 桁の対応

2 進数	16 進数	10 進数
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

では、これを利用して $(01101001)_2$ を 16 進数に直してみましょう。すでに説明した通りに、2 進数の 4 桁が 16 進数の 1 桁に対応していますので、$(01101001)_2$ を 4 桁ずつに分けて、$(0110)_2 = (6)_{16}$、$(1001)_2 = (9)_{16}$ と変換することができます (ここで、右下に 16 と書いたのは 16 進数であることを意味しています)。

したがって、

$$(01101001)_2 = (69)_{16}$$

であることが分かります。

このように、2 進数を 16 進数に変換するには、4 桁ずつひとまとまりにして変換するだけですので、比較的容易にできます。桁がいくら長くなっても問題はありません。

そこで C 言語のプログラムではビットの設定などを行う際に、16 進数を使うのです。

演習 2.3- 1

下の図のような基数変換の計算を、できるだけ短時間でできるようにしましょう。とりあえずの目標は 1 分です。

基数変換問題

名前　　時間　　正解数

2進数	16進数	10進数
1111		
	A	
		11
111		
	1	
		0
1001		
	4	
		13
11		

基数変換の計算用プリントを作成する EXCEL のファイルを、「基数変換演習.xlsm」という名前で用意しておきました。これを利用して充分に基数変換に慣れてください。

2.4 基数変換をプログラムする

ここでは、2 進数、16 進数および 10 進数の変換を C 言語でプログラムしてみましょう。

2.4.1 2 進数、16 進数、10 進数で表示するプログラム

以下は、NUM で定義した数値を、2 進数、16 進数、10 進数で表示するプログラムです。

02411hyouji01.c

```
 1  /*************************************************************************/
 2  /*                                                                       */
 3  /*  DESCRIPTION :10 進数を 2 進数、16 進数で表示 (1 バイト)               */
 4  /*  CPU TYPE    :PC                                                      */
 5  /*  NAME        :Ono Yasuji                                              */
 6  /*                                                                       */
 7  /*************************************************************************/
 8
 9  #include<stdio.h>
10
11  #define NUM 200
12
13  int main(void)
14  {
15    int n=NUM;
16
17    printf("decimal %d: ", n);
18    printf("hexadecimal %x: ", n);
19    printf("binary ");
20    printf("%d", n/128);
21    n=n%128;
22    printf("%d", n/64);
23    n=n%64;
24    printf("%d", n/32);
25    n=n%32;
26    printf("%d", n/16);
27    n=n%16;
28    printf("%d", n/8);
29    n=n%8;
30    printf("%d", n/4);
31    n=n%4;
32    printf("%d", n/2);
33    n=n%2;
34    printf("%d", n);
35    printf("\n");
36
37    return(0);
38  }
```

End Of List

実行結果 (Borland C++ Compiler 5.5)

```
>bcc32 02411hyouji01.c ↵

Borland C++ 5.5.1 for Win32 Copyright (c) 1993, 2000 Borland
02411hyouji01.c:
Turbo Incremental Link 5.00 Copyright (c) 1997, 2000 Borland

>02411hyouji01.exe ↵

decimal 200: hexadecimal c8: binaly 11001000


>
```

10 進数を 2 進数として表示する命令は printf にはありません。そこで自分で作らなければなりません。

ビットシフトを用いるともっとすっきりと書けるのですが、後で説明するのでここでは違う方法を使いました。

10 進数を 2 進数で表すために、P.17 の表 2.5 を利用しましょう。$(01101001)_2$ を 10 進数に直したときの逆のことをすれば良いわけです。10 進数の 200 の場合で考えると、

$$\begin{aligned} 200 &= 128 + 64 + 8 \\ &= 1 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 \\ &= (11001000)_2 \end{aligned}$$

であることが分かります。

これをプログラムにするには、与えられた 10 進数を $128(2^7)$ で割って結果を (整数部分だけで) 見ます。得られた値が 2^7 の係数になります。次に $128(2^7)$ で割ったときの余りを求めます (あるいは、割り算の結果が 1 だった時には元の値から 128 を引きます)。この結果は上式において、2^7 の項を取り除いたものになります。この結果に対して、$64(2^6)$ で割り算を行い、その結果を表示します。このようにして、表 2.5 の値が含まれているかを大きいほうから確認し、表示していくことで 2 進数を得ることができます。

以上をまとめると、次のようなアルゴリズムになります。

- 2 の 7 乗で割った結果を表示、2 の 7 乗で割った余りを求める。
- 2 の 6 乗で割った結果を表示、2 の 6 乗で割った余りを求める。
- 2 の 5 乗で割った結果を表示、2 の 5 乗で割った余りを求める。
- 2 の 4 乗で割った結果を表示、2 の 4 乗で割った余りを求める。
- 2 の 3 乗で割った結果を表示、2 の 3 乗で割った余りを求める。
- 2 の 2 乗で割った結果を表示、2 の 2 乗で割った余りを求める。
- 2 の 1 乗で割った結果を表示、2 の 1 乗で割った余りを求める。
- 結果を表示する。

2 のべき乗が違うだけで、同様な操作を繰り返しています。これは for 分などで分かりやすく書くことができるということです。

以下は、02411hyouji01.c の解説です。

プログラム解説 (02411hyouji01.c)

```
18  printf("hexadecimal %x: ", n);
```

%x は 16 進数で表示するための命令です。ここでは、変数 n の値が 16 進数に変換されて表示されます。

```
20  printf("%d", n/128);
```

10 進数の値 (n) を 128 で割っています。/は割り算で、整数 128 で割った場合には、結果は整数 (小数点以下切り捨て) になります。

```
21  n=n%128;
```

%を使って 128 で割った余りを求めています。最上位のビットを 0 にすることと同じです。

ここでは割り算と余りを求める演算を用いましたが、後で説明するシフトを用いるとさらに見通しの良いプログラムを組むことができます。

✍ 課題 2.4.1 (提出) 02411hyouji01.c を for ループを使って書きなおす

プログラム 02411hyouji01.c を for ループを用いて書きなおしてください。

e02411hyouji01.c という名前で保存してください。

2.5 マイコンプログラムのためのビット演算 (1 バイト)

ここでは、P.11 の 2.1 で学習した論理演算を、1 バイト二つの値の間に適用してみましょう。また、1 バイトの値を左右どちらかにずらすシフト演算も学習しましょう。

ここで学習する内容は、このテキストが目標とする移植性の良いプログラムを組むためにはきわめて重要な内容です。少々込み入った部分もありますが、しっかりと学習し、確実に理解してください。

2.5.1 論理積 (AND)

1 バイトの値二つの論理積は、対応する 1 ビットずつの論理積で計算できます (図 2.1)。

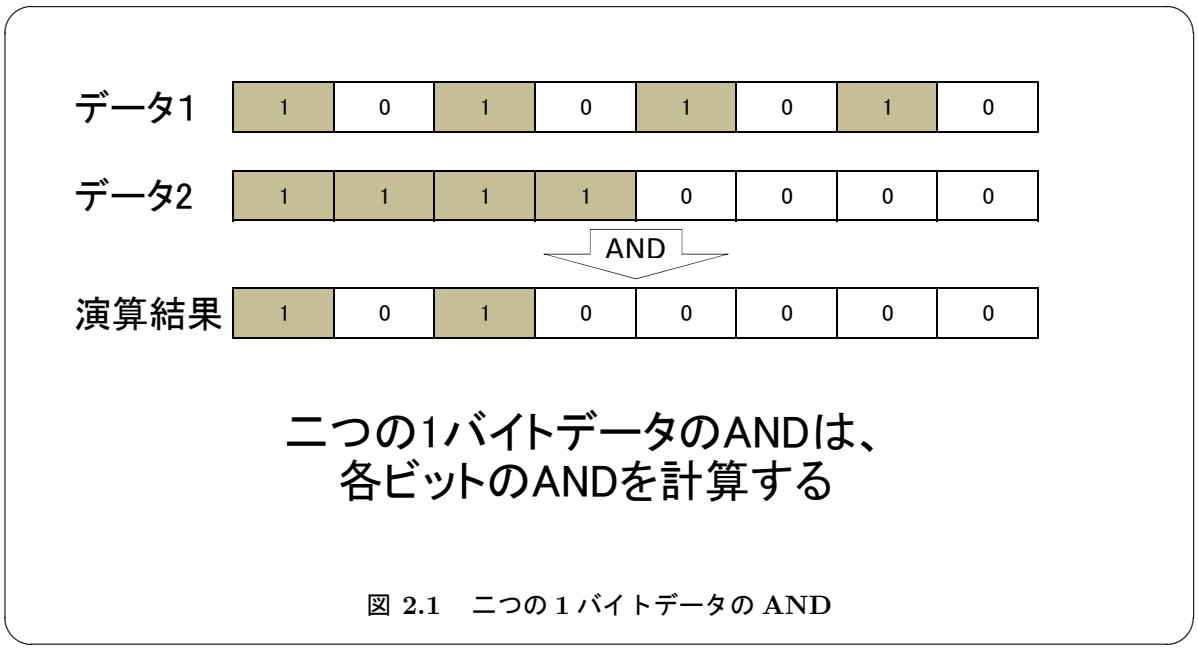


図 2.1　二つの 1 バイトデータの AND

図 2.1 で、一番左のビットを例に考えてみましょう。

データ 1 の一番左のビット値は「1」です。データ 2 の一番左のビット値も「1」ですから、両者の論理積「1」が、演算結果の一番左のビットの値になります。

同様に、左から二番目のビット値は「0」と「1」ですから、演算結果は「0」となるわけです。

このようにして、各ビットごとの論理積を取ることによって、図 2.1 の演算結果になることを確認してみてください。

以下では、この論理積がマイコンプログラムのどのような場面で利用されるかを考えてみましょう。

2.5.2 必要なビットだけ 0 に設定し、他は変えない (AND)

論理積の演算をマイコンプログラムでどのように使うかを考えてみましょう。すでに説明したように、マイコンのプログラムにおいてはレジスタの値を一部変更して、必要のないところは変更しないという操作が必要になってきます。直接値を代入してしまうと必要のないビットまで変更してしまいかねません。

このような場合、対象となっていない周辺機器に影響を与えてしまうかもしれないのです。たとえば、ブザーを鳴らすプログラムを書いているのに、LCD の設定を変更してしまうなどということが起こり得るわけです。このようなことが起こらないように、論理演算を使って必要なビットだけ変更し、必要のないビットを変更しないという操作を行います。

レジスタの下位 (図では右側)4 ビットを 0 に設定することを考えてみましょう。上位 4 ビットは変更しないことにします。図 2.1 の例を用い、データ 1 はレジスタの値だとします。変更される前のレジスタの値は 10101010 です。今の場合上位 4 ビットの値が分かっているので、16 進数で 0xA0 を代入すれば良いと思うかもしれません。しかし、実際にはマイコンが動作している途中でレジスタの値は変更されることもあり、プログラムの段階で特定できるとは限りません。論理積 (AND) を使えば必要なビットだけ 0 にして、他を変更しないことができますので、これを使うことにします。

1 バイトの論理積は 1 ビットずつの論理積に分けることができるという話をしました。そこでまず、1 ビットの論理積にどのような性質があるのかを見てみたいと思います。

1 ビットの論理積については、P.12 で説明しました。これをもとに、以下のように場合分けして考えてみましょう。

論理積の二つの入力のうち、片方が 0 であった場合には、演算の結果は 0 になります (図 2.2)。元データが 0 であっても 1 であっても演算結果は 0 になるわけです。

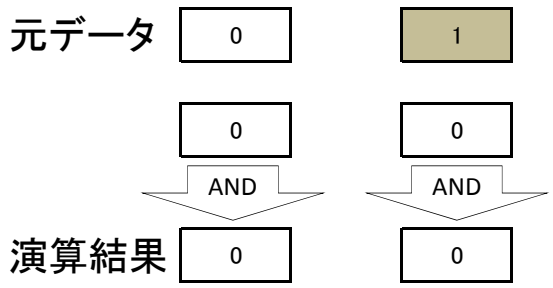


図 2.2 0 との論理積

では、片方が 1 であった場合はどうでしょう。この場合には、もう一方の値が 0 なら演算結果は 0。1 ならば 1 になることが分かります。つまり、1 と論理積を取ると、元の値と変わらないわけです (図 2.3)。

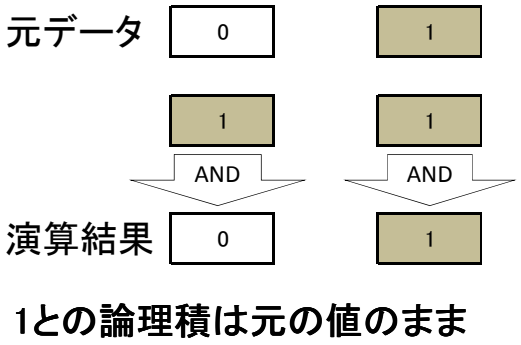


図 2.3　1 との論理積

この、「元の値を変更しない」という演算結果が重要なのです。

以上をまとめると、0 との論理積は 0 になり、1 との論理積は値を変えないということになります。

このことから、あるビットを 0 に設定し、他のビットは変更しないという目的のためには、0 に設定したいビットは 0 と、変更したくないビットは 1 と論理積を用いればよいということが分かります。

したがって下位 4 ビットを 0 に設定し、上位 4 ビットは変えないという演算は、図 2.4 のように 11110000 との論理積をとれば良いわけです。

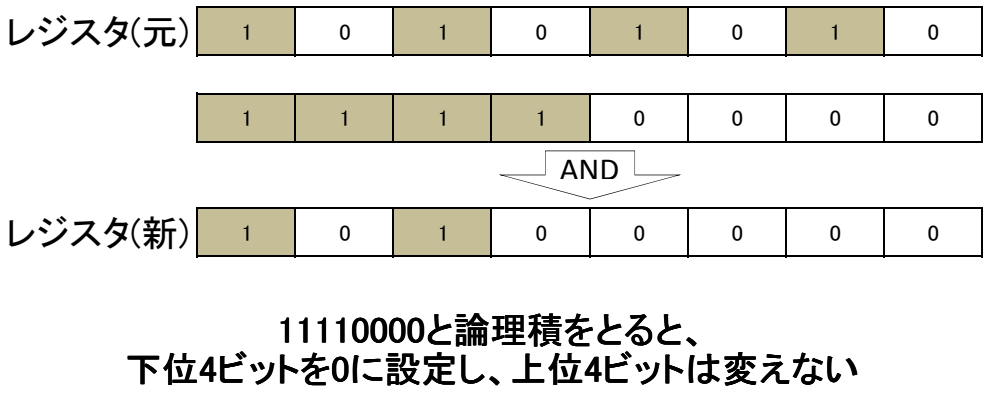


図 2.4　下位 4 ビットを 0 に設定し、上位 4 ビットは変えない演算

注意

ここで行っている演算は、レジスタの値に特に依存していないことに注意してください。図 2.4 では、レジスタの元の値に関係なく、下位 4 ビットを 0 に設定し、上位 4 ビットは変えない演算になっています。実際にレジスタの元の値を様々に変更して、そのようになっていることを確認してみてください。

演習 2.5- 1

1 バイトのレジスタの上位 2 ビットと下位 2 ビットを 0 に設定し、中の 4 ビットを変更しないようにするためには、どのような値とどのような演算を行えばよいか考えてみましょう。

まとめ：

0 に設定したいビットは 0 と、変更したくないビットは 1 と論理積 (AND) を取る。

2.5.3 必要なデータだけ取り出し、他は 0 に設定する (AND)

論理積（AND) は、必要なビットだけ 0 に設定し、他は変えない演算に利用できると説明をしました。0 に設定したいビットは 0 と、変更したくないビットは 1 と論理積 (AND) を取るわけです。

行うことは全く同じなのですが、解釈を変えることで異なる場面に利用することができます。必要なビットだけ 0 に設定して他は変えないということは、必要なデータだけをそのまま取り出して他は 0 にしてしまうと見ることもできるわけです (図 2.5)。

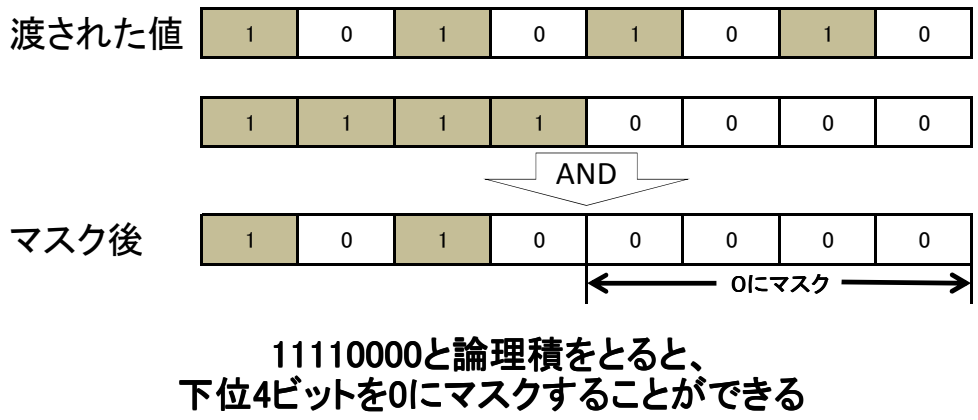


図 2.5　ビットを 0 にマスクする

こうして取り出したデータを何らかの形で利用することが考えられます。

具体的な例としては、受け渡されたデータのうち下位 3 ビットのデータだけを取り出して、その中の値

が 1 のビットをレジスタに反映させるという場面が想定されます (詳細は後ほど説明します)。

このように不要なデータを強制的に 0 にするときに利用するわけです。このような操作を「ビットを 0 にマスクする」と言ったりします。

演習 2.5- 2

1 バイトのレジスタの上位 3 ビットを 0 でマスクするには、どのような値とどのようなビット演算をすれば良いでしょうか。

2.5.4 2 のべき乗で割った余りを求める (AND)

P.20 の 02411hyouji01.c において、10 進数を 2 進数に変換するプログラムを紹介しました。

そこで用いたアルゴリズムは、

- 2 の 7 乗で割った結果を表示、2 の 7 乗で割った余りを求める。
- 2 の 6 乗で割った結果を表示、2 の 6 乗で割った余りを求める。
- 2 の 5 乗で割った結果を表示、2 の 5 乗で割った余りを求める。
- 2 の 4 乗で割った結果を表示、2 の 4 乗で割った余りを求める。
- 2 の 3 乗で割った結果を表示、2 の 3 乗で割った余りを求める。
- 2 の 2 乗で割った結果を表示、2 の 2 乗で割った余りを求める。
- 2 の 1 乗で割った結果を表示、2 の 1 乗で割った余りを求める。
- 結果を表示する。

というものでした。

このとき、割り算は/で、余りは%で計算しました。しかし、一般にビット演算は乗除算などと比べて高速です。そこで、ビット演算で置き換えることを考えてみましょう。

2 のべき乗で割った余りの例として、$8(=2^3)$ で割ったときの余りを考えてみましょう。

0 から 15 までを、8 で割ったときの余りを表にしてみます (表 2.7)。

表 2.7　0 から 15 までを、8 で割ったときの余り

10 進数	2 進数表示	8 で割った余り	8 で割った余りを 2 進数表示 (3 桁)
0	0	0	000
1	1	1	001
2	10	2	010
3	11	3	011
4	100	4	100
5	101	5	101
6	110	6	110
7	111	7	111
8	1000	0	000
9	1001	1	001
10	1010	2	010
11	1011	3	011
12	1100	4	100
13	1101	5	101
14	1110	6	110
15	1111	7	111

これから、8 で割ったときの余りは 2 進数で下位 3 桁を取り出すことと等しいということが分かるでしょうか。これは、$8-1=7=(111)_2$ と論理積を取ることによって得ることができます。

同様にして、

- $2(2^1)$ で割った余りは、$1(2^1-1)$ との論理積、
- $4(2^2)$ で割った余りは $3(2^2-1=(11)_2)$ との論理積、
- $8(2^3)$ で割った余りは $7(2^3-1=(111)_2)$ との論理積、
- $16(2^4)$ で割った余りは $15(2^4-1=(1111)_2)$ との論理積、
- $32(2^5)$ で割った余りは $31(2^5-1=(11111)_2)$ との論理積、
- $64(2^6)$ で割った余りは $63(2^6-1=(111111)_2)$ との論理積、
- $128(2^7)$ で割った余りは $127(2^7-1=(1111111)_2)$ との論理積

を取れば良いことが分かります。

このようにして、2^n で割ったときの余りは、2^n-1 と論理積を取ることによって計算できるわけです (図 2.6)。

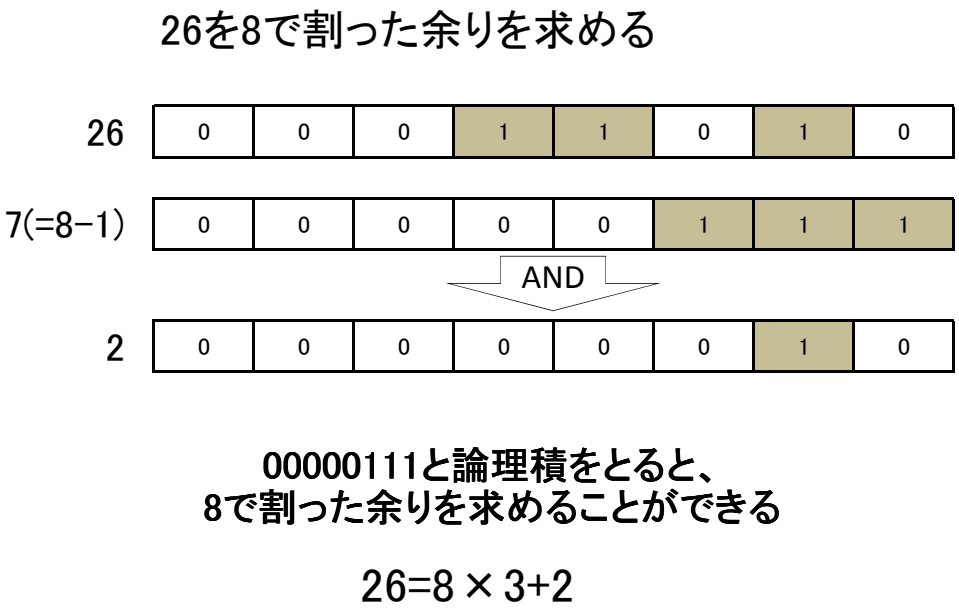


図 2.6　2^n で割ったときの余りを求める

ただし、注意すべきこととして、この方法で計算可能なのは 2 のべき乗で割ったときの余りだけだということです。2 のべき乗ではない、たとえば 5 で割ったときの余りはこの方法では計算できません (実際に 4 と論理積を取ってみれば分かると思います)。

✍ 課題 2.5.1　(提出) 02411hyouji01.c を論理積を使って書きなおす

プログラム 02411hyouji01.c を論理積を使って書きなおしてください。

e02511hyouji01.c という名前で保存してください。

2.5.5 論理和 (OR)

1 バイトの値二つの論理和は、対応する 1 ビットずつの論理和で計算できます (図 2.7)。

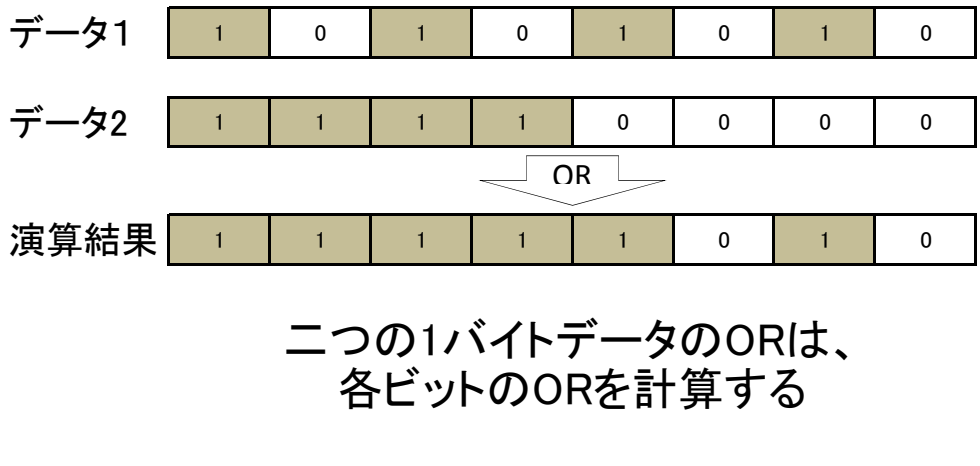


図 2.7 二つの 1 バイトデータの OR

図 2.7 で、一番左のビットを例に考えてみましょう。

データ 1 の一番左のビット値は「1」です。データ 2 の一番左のビット値も「1」ですから、両者の論理和「1」が、演算結果の一番左のビットの値になります。

同様に、左から二番目のビット値は「0」と「1」ですから、演算結果は「1」となるわけです。

このようにして、各ビットごとの論理和を取ることによって、図 2.7 の演算結果になることを確認してみてください。

以下では、この論理和がマイコンプログラムのどのような場面で利用されるかを考えてみましょう。

2.5.6 必要なビットだけ 1 に設定し、他は変えない (OR)

論理和の演算をマイコンプログラムでどのように使うかを考えてみましょう。すでに説明したように、マイコンのプログラムにおいてはレジスタの値を一部変更して、必要のないところは変更しないという操作が必要になってきます。論理演算を使って必要なビットだけ変更し、必要のないビットを変更しないという操作を行います。論理積と論理和をどのように使い分けるかも把握しておきましょう。

レジスタの上位 (図では左側)4 ビットを 1 に設定することを考えてみましょう。下位 4 ビットは変更しないことにします。図 2.7 の例を用い、データ 1 はレジスタの値だとします。変更される前のレジスタの値は $(10101010)_2$ です。

1 バイトの論理和は 1 ビットずつの論理和で計算できるという説明をしました。そこでまず、1 ビットの論理和にどのような性質があるのかを見てみたいと思います。

論理和の二つの入力のうち、片方が 1 であった場合には、演算の結果は 1 になります (図 2.8)。

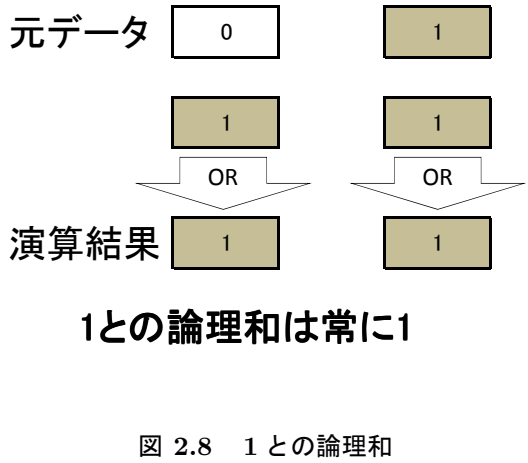


図 2.8　1 との論理和

では、片方が 0 であった場合はどうでしょう。この場合には、もう一方の値が 0 なら演算結果は 0。1 ならば 1 になることが分かります。つまり、0 と論理和を取ると、元の値と変わらないわけです (図 2.9)。

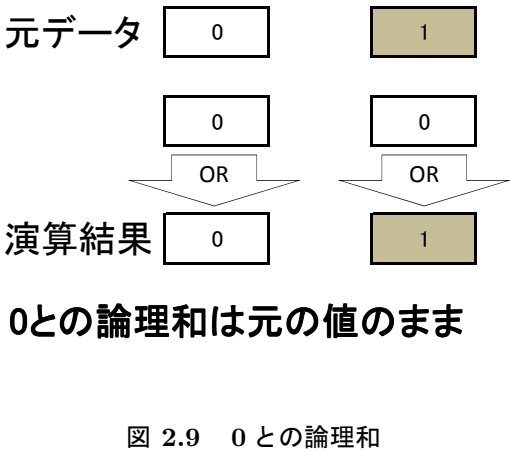


図 2.9　0 との論理和

以上をまとめると、1 との論理和は 1 になり、0 との論理和は値を変えないということになります。

このことから、あるビットを 1 に設定し、他のビットは変更しないという目的のためには、1 に設定したいビットは 1 と、変更したくないビットは 0 と論理和を取ればよいということが分かります。

したがって上位 4 ビットを 1 に設定し、下位 4 ビットは変えないという演算は、図 2.10 のように $(11110000)_2$ との論理和をとれば良いわけです。

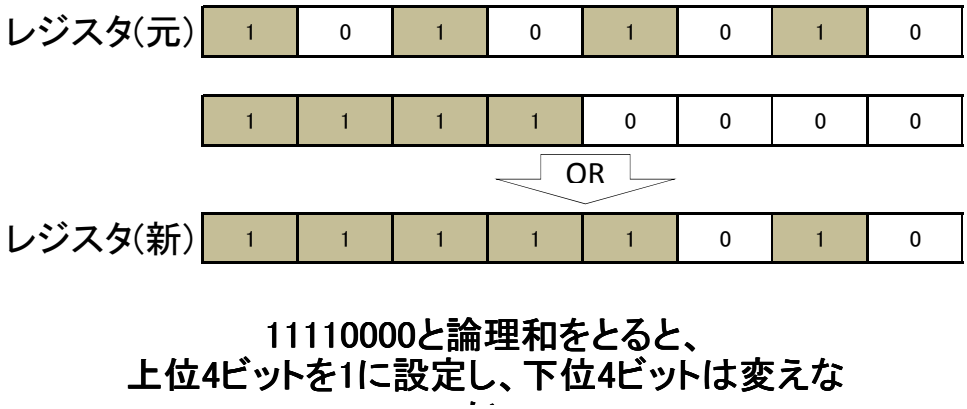


図 2.10　上位 4 ビットを 1 に設定し、下位 4 ビットは変えない演算

注意

ここで行っている演算は、レジスタの値に特に依存していないことに注意してください。図 2.10 では、レジスタの元の値に関係なく、上位 4 ビットを 1 に設定し、下位 4 ビットは変えない演算になっています。実際にレジスタの元の値を様々に変更して、そのようになっていることを確認してみてください。

演習 2.5- 3

1 バイトのレジスタの上位 2 ビットと下位 2 ビットを 1 に設定し、中の 4 ビットを変更しないようにするためには、どのような値とどのような演算を行えばよいか考えましょう。

まとめ :
1 に設定したいビットは 1 と、変更したくないビットは 0 と論理和 (OR) を取る。

P.23 の 2.5.2 では、0 に設定したいビットは 0 と、変更したくないビットは 1 と論理積 (AND) を取るという説明をしました。ここでは、1 に設定したいビットは 1 と、変更したくないビットは 0 と論理和 (OR) を取るという説明をしています。これらを使うと、ある範囲の値を変更し、それ以外を変更しないという操作が可能になります。

たとえば、下位 4 ビットを $(1010)_2$ に変更し、上位 4 ビットは変更しない演算を考えてみましょう。これは一種類のビット演算で実現することはできませんので、論理積と論理和を使うことになります。

まず、$(11111010)_2$ を論理積を取り、その後に $(00001010)_2$ と論理和を取れば良いことが分かります (図 2.11)。論理積と論理和の順番は逆でもかまいません。

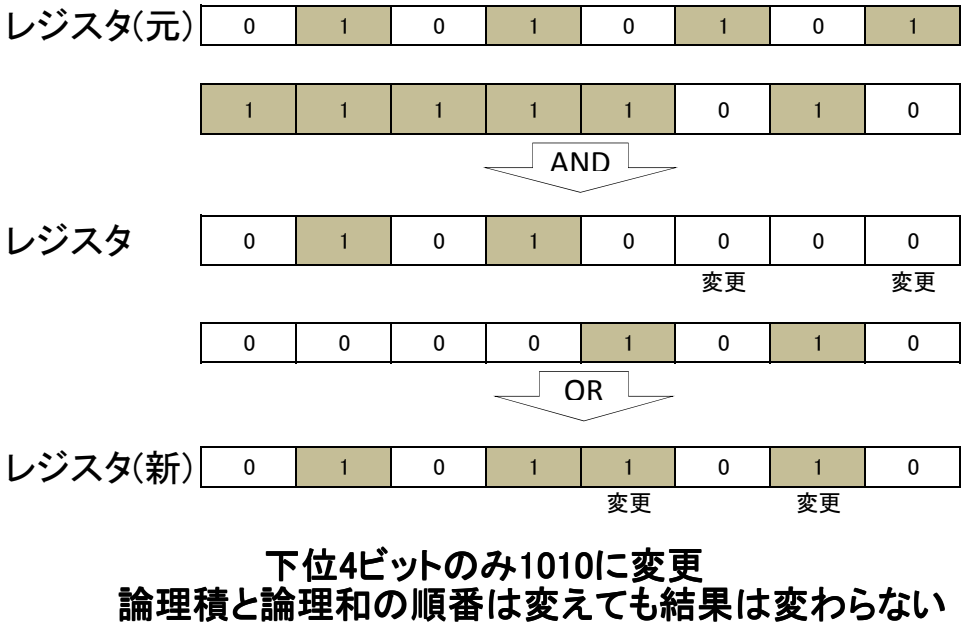


図 2.11　下位 4 ビットのみ 1010 に変更

ほかの書き方、たとえば、まず、$(11110000)_2$ と論理積を取り、その後に $(00001010)_2$ と論理和を取っても良いことが分かります。しかし、この場合は論理積と論理和の順番を入れ替えると結果が変わってしまいます。順番を入れ替えても結果が変わらないように、必要のない書き換えを行わないほうが良いでしょう。後でプログラムを変更した場合に、順番を入れ替えてしまわないとも限らないからです。こうして引き起こされた誤動作は原因を見つけるのが大変なものです。何か特別な事情がない限り、論理積では必要なビットのみを 0 に設定し、論理和では必要なビットのみを 1 に設定するようにしましょう。

演習 2.5- 4

1 バイトのレジスタの上位 6 ビットを $(110010)_2$ に設定し、下位の 2 ビットを変更しないようにするためには、どのような値とどのような演算を行えばよいか考えてください。

まとめ :

一部のビットだけを変更する方法。

0 に設定したいビットは 0 と、変更したくないビットは 1 と論理積 (AND) を取る。

1 に設定したいビットは 1 と、変更したくないビットは 0 と論理和 (OR) を取る。

2.5.7 必要なデータだけ取り出し、他は 1 に設定する (OR)

論理積の場合と同様に、論理和も解釈を変えることで異なる場面に利用することができます。必要なビットだけ 1 に設定して他は変えないということは、必要なデータだけをそのまま取り出して他は 1 にしてしまうと見ることもできるわけです (図 2.12)。

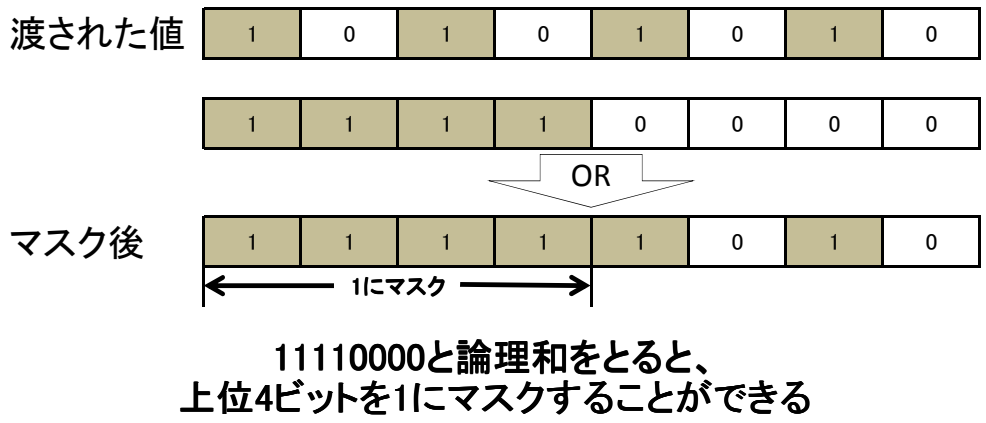


図 2.12　ビットを 1 にマスクする

こうして取り出したデータを何らかの形で利用することが考えられます。

具体的な例としては、受け渡されたデータのうち下位 3 ビットのデータだけを取り出して、その中の値が 0 のビットをレジスタに反映させるという場面が想定されます (詳細は後ほど説明します)。

このように不要なデータを強制的に 1 にするときに利用するわけです。このような操作を「ビットを 1 にマスクする」と言ったりします。

演習 2.5- 5

1 バイトのレジスタの上位 3 ビットを 1 でマスクするには、どのような値とどのようなビット演算をすれば良いかを答えなさい。

2.5.8 排他的論理和 (XOR)

1 バイトの値二つの排他的論理和も、対応する 1 ビットずつの排他的論理和で計算できます (図 2.13)。

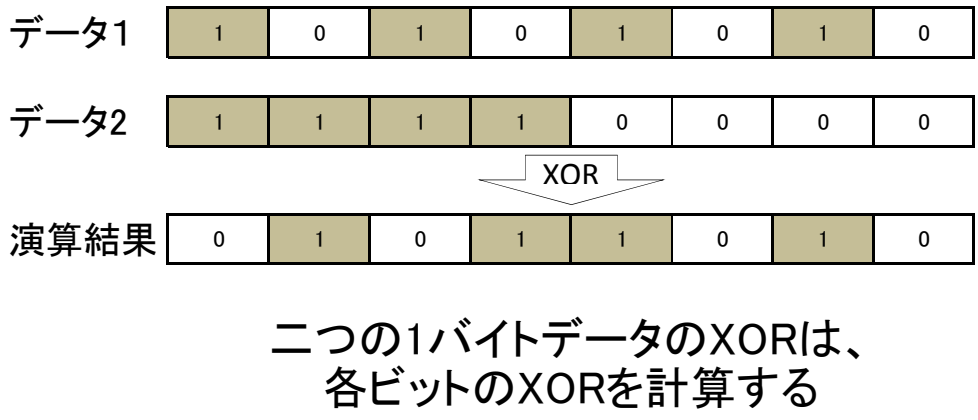


図 2.13　二つの 1 バイトデータの XOR

今回も図 2.13 で、一番左のビットを例に考えてみましょう。

データ 1 の一番左のビット値は「1」です。データ 2 の一番左のビット値も「1」ですから、両者の排他的論理和「0」が、演算結果の一番左のビットの値になります。

同様に、左から二番目のビット値は「0」と「1」ですから、演算結果は「1」となるわけです。

このようにして、各ビットごとの排他的論理和を取ることによって、図 2.7 の演算結果になることを確認してみてください。

以下では、この排他的論理和がマイコンプログラムのどのような場面で利用されるかを考えてみましょう。

2.5.9 必要なビットだけ値を反転させる、他は変えない (XOR)

排他的論理和の演算をマイコンプログラムでどのように使うかを考えてみましょう。

1 バイトの排他的論理和も 1 ビットずつの排他的論理和で計算できることを確認しました。そこでここでも、1 ビットの排他的論理和にどのような性質があるのかを見てみたいと思います。

排他的論理和の二つの入力のうち、片方が 0 であった場合には、演算の結果は元の値と同じです (図 2.14)。

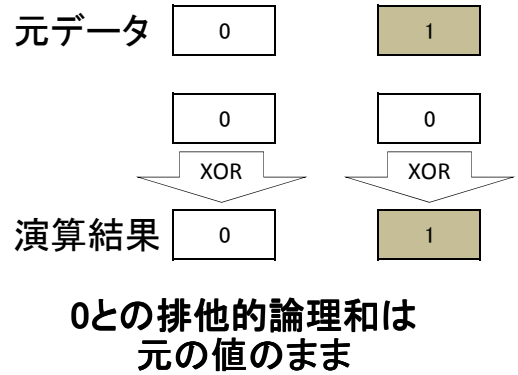


図 2.14　1 との論理和

では、片方が 1 であった場合はどうでしょう。この場合には、もう一方の値が 0 なら演算結果は 1、1 ならば 0 になることが分かります。つまり、1 と排他的論理和を取ると、値が反転するわけです (図 2.15)。

1 と排他的論理和を 2 度取ると、元の値に戻ります。

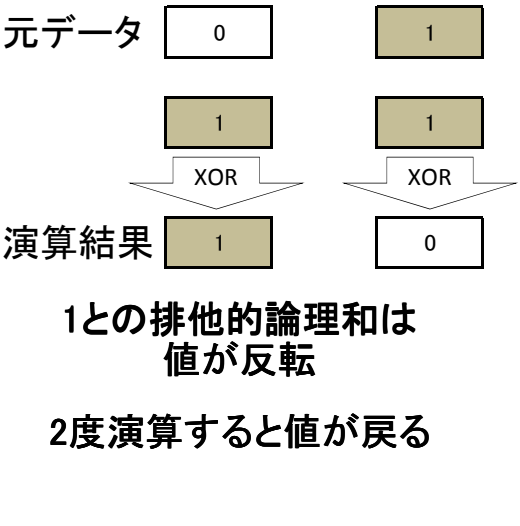


図 2.15　1 との排他的論理和

以上をまとめると、1 との排他的論理和は値が反転し、0 との排他的論理和は値を変えないということになります。

このことから、あるビットを反転し、他のビットは変更しないという目的のためには、反転させたいビットは 1 と、変更したくないビットは 0 と排他的論理和を用いればよいということが分かります。

したがって上位 4 ビットを反転させ、下位 4 ビットは変えないという演算は、図 2.16 のように $(11110000)_2$

との排他的論理和をとれば良いわけです。

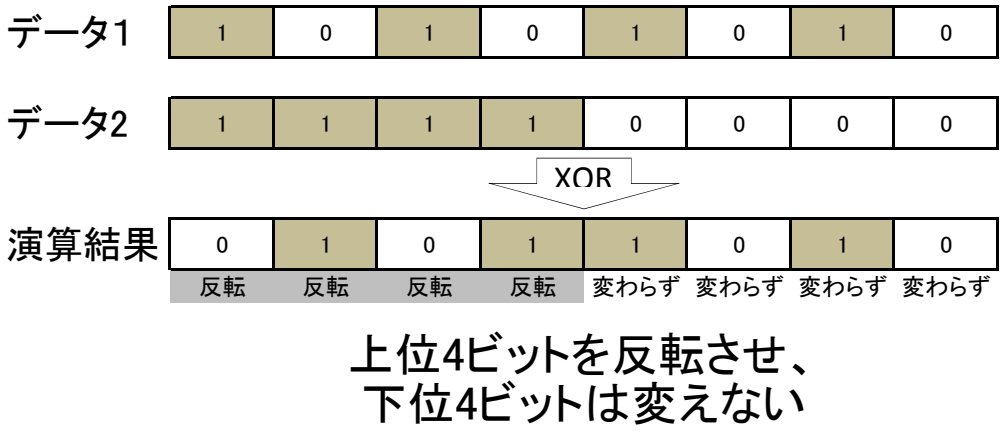


図 2.16　排他的論理和で一部の値を反転させる

一部のビットだけ値を反転させるという操作は、LED の点滅をプログラムする際に役立ちます。

また、PWM 出力をソフトウェアで実装するなどにも利用できます。

マイコンプログラムでは値を反転させるという操作は少なくありません。if 文など条件文を使って、現在の値が 1 であるか 0 であるかを調べて反転させることもできるのですが、1 ビットずつ条件文を書かなくてはなりません。プログラムが長くなりますし、実行速度も遅くなります。

そのような場合に、元の値が何であるかに関係なく、必要な部分だけ一度に複数ビットの値を反転させることができる排他的論理和は大変便利な演算です。

演習 2.5- 6

1 バイトのレジスタの上位 2 ビットと下位 2 ビットの値を反転させ、中の 4 ビットを変更しないようにするためには、どのような値とどのような演算を行えばよいか考えてください。

まとめ :

反転させたいビットは 1 と、変更したくないビットは 0 と排他的論理和 (XOR) を取る。

2.5.10 値を反転させる

値の反転は、P.12 で否定 (NOT) を説明しました。しかし、C 言語では否定 (NOT)!を 1 バイトの値に演算した場合、それぞれのビットを反転することにはなりません。

否定 (NOT)!は本来真を偽に、偽を真に変換する論理否定演算です。C 言語で偽は 0 であり、真は 1 以外の値になります。したがって、1 バイトの値 0 に否定 (NOT)!を演算した場合、1 になるだけでビットごとに反転はしません。0 以外の値を否定すると 0 になります。

ビットごとの反転にはチルダ演算子~を用います。

~演算子は 1 の補数を求める演算で、それぞれのビットの値を反転させます (図 2.17)。

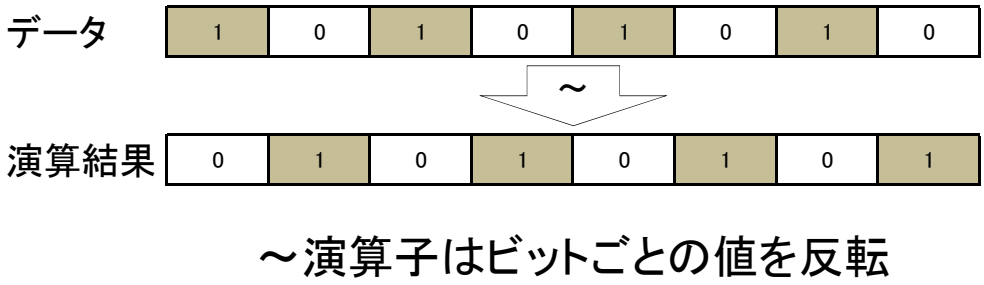


図 2.17 ～演算子

~演算子は、全てのビットを反転させるので、同じ値が入っている変数でも、変数の大きさが違う場合には結果が違ってきます。

例えば、unsigned char 型 (1 バイト) の変数 a の中身が 1 だった場合、~演算子を作用させると結果は 0xFE です。

ところが、unsigned short 型 (2 バイト) の変数 a の中身が 1 だった場合、~演算子を作用させた結果は 0xFFFE なのです (図 2.18)。

~演算子を使う場合には、対象となる変数等の大きさにも気をつけてください。

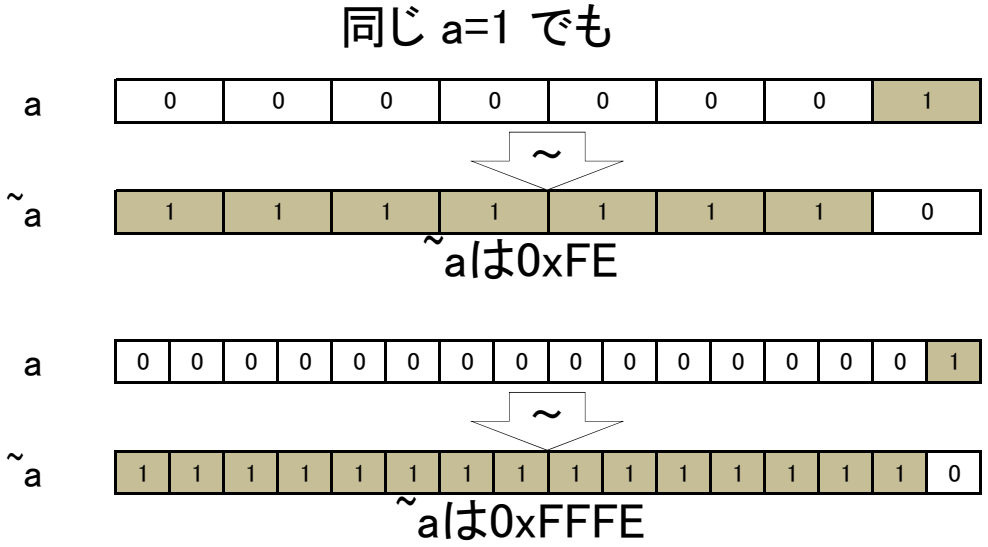


図 2.18　～演算子とビット幅

注意

~演算子と同じ結果は排他的論理和を使っても得られます。
全てのビットを 1 にした値と排他的論理和を取れば良いのです。
以下では~演算子は、関数の引数として受け取った値を、反転させるときに利用することにします。
引数の値が 1 の場所のレジスタを 0 に設定するとか、逆に 0 の値のところを 1 に設定するという場合です。
詳細は後ほど詳しく説明します。

2.5.11 シフト

シフト (shift) はデータを左右どちらかにずらす操作です。おもにレジスタのデータ操作に利用しますので、このテキストでは非負 (0 以上) の値で考えることにします。

左シフトは、指定されたビット数だけ左にずらす操作です。右側のあいたビットには 0 を詰めます (図 2.19)。

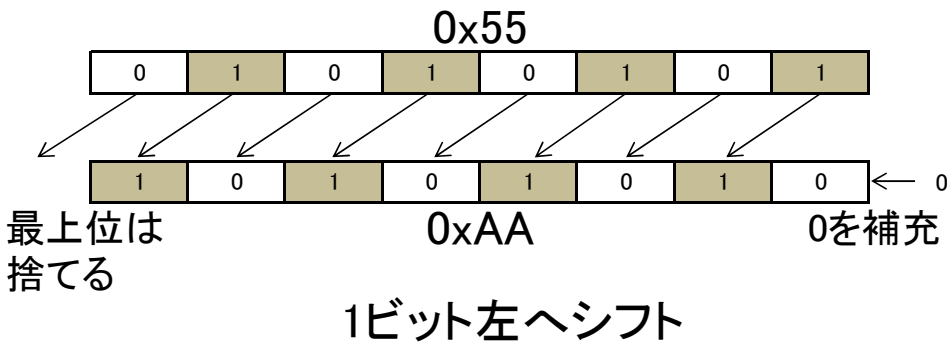


図 2.19 左シフト

左シフトはどのような場面で使われるのでしょう。

もちろん、LED の点灯をずらすというような用途にも利用できるのですが、それ以外にも以下のような場面でよく使われます。

組込みマイコンではレジスタを 1 バイト単位で扱います。マイコンの各足にレジスタが対応している場合には、8 本の足のデータを一度に扱うことになります。

たとえば LED はマイコンの足の一か所に接続することになるのですが、LED を 8 個制御するシステムというのはそれほど多くありません。通常は 8 本の足のうちの一部を使うことになります。

3 個だけ使う場合を考えてみましょう。

このとき、1 バイトのレジスタの最下位ビットから 3 ビット分につながっていれば良いのですが、実際には必ずしもそうできるわけではありません。最下位から 3 ビットをスイッチに使い、その次の 3 ビットを LED に使うなどということが普通に行われます。マイコンの足の数は限られているので、このように混在して利用するしかないのです。

こうしたとき、点灯する LED を指定するのに、3 ビットずらしたデータで指定するよりも、最下位ビットから 3 ビット分のデータで指定するほうが分かりやすく間違えも少なくなるでしょう。そこで、LED を指定するデータとしては最下位から 3 ビット分で記述し、レジスタに設定するときには左に 3 ビットシフトして設定するというプログラムの仕方が考えられます (図 2.20)。

このような場面でも左シフトは利用されるわけです。

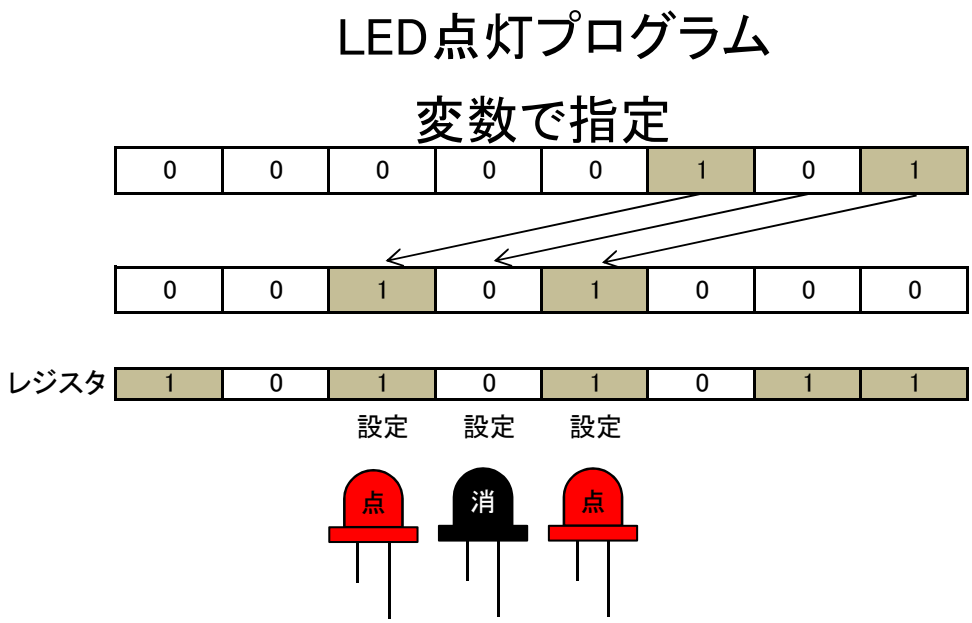


図 2.20 左シフトの利用例

右シフトは、指定されたビット数だけ右にずらす操作です。左側のあいたビットには 0 を詰めます (図 2.21)。

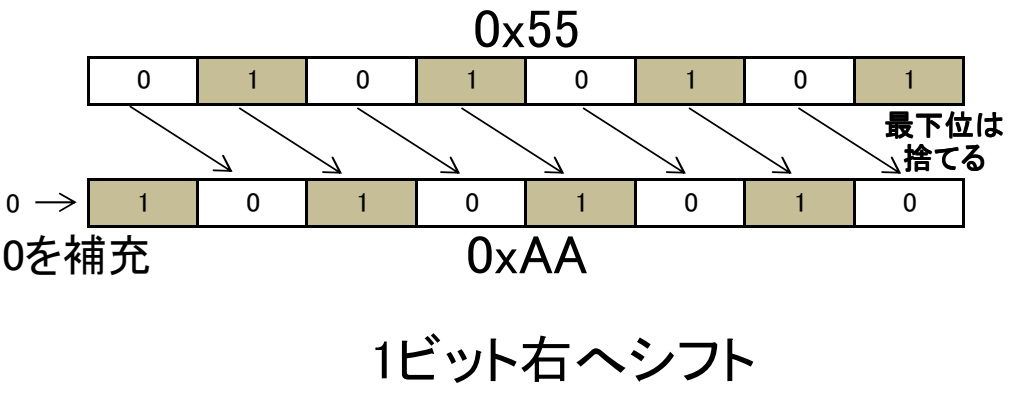


図 2.21 右シフト

注意

変数の値を右シフトした場合、符号付きの場合と符号なしの場合で、左側のあいたビットに詰める値が違います。
符号なしの場合にはあいたビットには 0 が詰められます。
符号ありの場合には、負の数は最上位ビットが 1 ですので、符号を保持するために 1 を詰めるのが一般的です。つまり、0 以上の数については 0 が、負の数については 1 が代入されるのです。
このテキストでは、符号付きデータの右シフトは利用しないこととし、符号なしのデータのみを考えることとしました。

シフトの利用例として、P.20 の 02411hyouji01.c をシフトで書きなおしてみましょう。

02512hyouji01.c

```
 1  /*********************************************************************/
 2  /*                                                                   */
 3  /*  DESCRIPTION :10 進数を 2 進数、16 進数で表示 (1 バイト)           */
 4  /*  CPU TYPE    :PC                                                  */
 5  /*  NAME        :Ono Yasuji                                          */
 6  /*                                                                   */
 7  /*********************************************************************/
 8
 9  #include<stdio.h>
10
11  #define NUM 200
12
13  int main(void)
14  {
15    int i, n=NUM;
16
17    printf("decimal %d: ", n);
18    printf("hexadecimal %x: ", n);
19    printf("binary ");
20    for(i=7; i>=0; i--){
21      printf("%d", (n>>i)&1);
22    }
23    printf("\n");
24
25    return(0);
26  }
```

End Of List

実行結果 (Borland C++ Compiler 5.5)

```
>bcc32 02512hyouji01.c ↵

Borland C++ 5.5.1 for Win32 Copyright (c) 1993, 2000 Borland
02512hyouji01.c:
Turbo Incremental Link 5.00 Copyright (c) 1997, 2000 Borland

>02512hyouji01.exe ↵

decimal 200: hexadecimal c8: binaly 11001000

>
```

実行結果は P.20 の 02411hyouji01.c の場合と同じです。NUM の値を 200 から他の数値に変更して結果を確認してみてください。

以下はプログラム 02512hyouji01.c の解説です。

プログラム解説 (02512hyouji01.c)

```
20  for(i=7; i>=0; i--){
21    printf("%d", (n>>i)&1);
22  }
```

21 行目で抜き出したいビットを最下位ビットにシフトします。そのビットだけ抜き出すために、1 と論理積&を取ります。

20 行目と 22 行目でループを回しています。上位ビットから順に最下位ビットにまでシフトして、論理積でビットの値を抜き出しています。

2.5.12 値を 2 のべき乗倍する、2 のべき乗で割る (シフト)

シフトは指定されたビット数データを右方向もしくは左方向にずらす演算でした。

このずらす前のデータとずらした後のデータを計算してみましょう。

$(00000100)_2 = 4$ を 1 ビット左シフトする場合で計算してみます。

1 ビット左シフトすると、$(00001000)_2 = 8$ になります。元の値の 2 倍になっているわけです。2 ビット左シフトすると、$(00010000)_2 = 16$ と 4 倍になります。

このように n ビット左シフトすると、2^n 倍することになるのです。

ただし、n ビット左シフトする際に、上位 n ビットの値は 0 でないと値が捨てられてしまうことになるので注意が必要です。

C 言語では*で掛け算ができますが、ビットをシフトしたほうが高速に演算ができます。

そこで、2^n 倍するのにこの演算を使ったりします。

次に $(00000100)_2 = 4$ を 1 ビット右シフトする場合を考えてみましょう。

1 ビット右シフトすると、$(00000010)_2 = 2$ になります。元の値の 1/2 倍になっているわけです。2 ビット右シフトすると、$(00000001)_2 = 1$ と 1/4 倍になります。

このように n ビット右シフトすると、$1/2^n$ 倍することになるのです。

ただし、n ビット右シフトする際に、下位 n ビットの値はが 0 でない場合には、余りの値が捨てられてしまうことになるので注意が必要です。たとえば、$(00000011)_2 = 3$ を 1 ビット右シフトすると $(00000001)_2 = 1$ となり、余りの 1 は捨てられているわけです。

2.5.13 1 から 0 への変化をみる

スイッチが押されたのを確認したり、レジスタのあるビットが 1 に設定されたのを確認したりと、ビットの変化を認識する必要がある場面は多いものです。これも、if 文などの条件文を用いて書くこともできるのですが、ビット演算を用いて変化を調べることもできます。

ここでは、あるビットが 1 から 0 に変化したことを感知する演算を考えてみましょう。

ある値からある値への変化を見るには、以前の値と現在の値を比較する必要があります。

以前の値を old、現在の値を new とします。いずれも 1 ビットで 0 もしくは 1 を取るとしましょう。

場合としては old が 0 もしくは 1、new が 0 もしくは 1 が考えられますから、4 通りがあるわけです。論理演算を利用して、old が 1 で new が 0 のときだけ他と異なる結果を得たいわけです。このとき利用できそうな演算は、二つの入力に対してひとつだけ異なる結果を出力するものでなければなりません。

この条件に合いそうなものは論理積と論理和であるということが分かります。

ここでは、論理積を用いてこれを実現してみましょう。

論理積で出力が他と異なるのは、二つの入力が 1 の場合で、このとき出力は 1 になります。今知りたいのは 1 から 0 に変化した場合なので、old が 1、new が 0 です。ともに 1 になるようにするには new の値を反転させれば良いことが分かります。

以上のことから、1 から 0 への変化をみるためには、old に以前の値を、new に現在の値を代入して、`old & (~new)` という演算を行えば良いことが分かります。値が 1 から 0 に変化したときに演算結果は 1 になり、それ以外の場合には演算結果は 0 になります (図 2.22)。

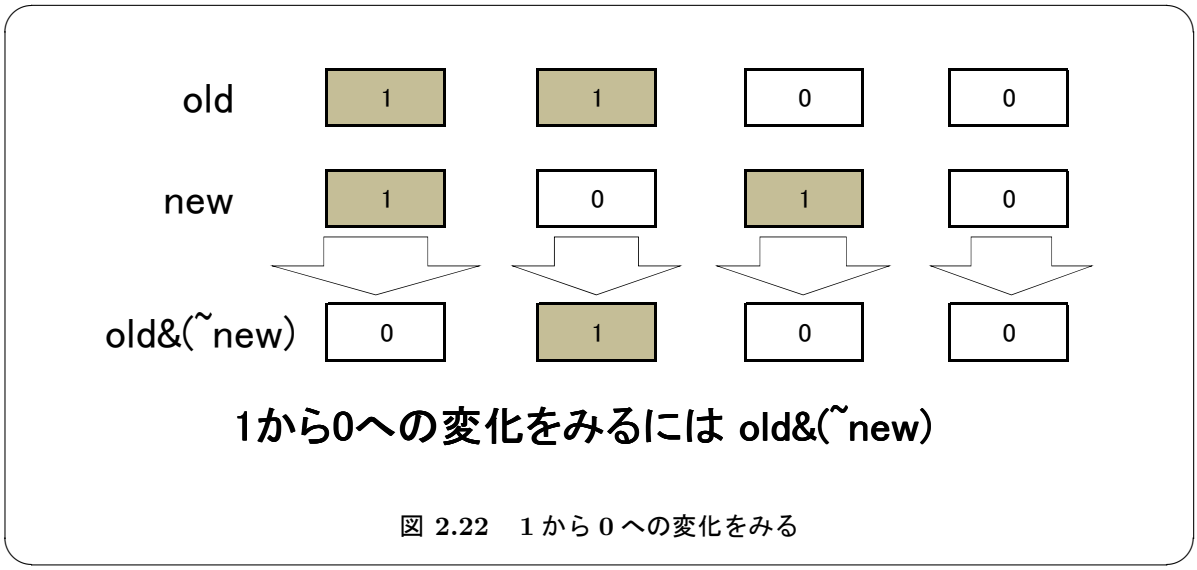


図 2.22　1 から 0 への変化をみる

上の説明では 1 ビットだけを考えましたが、これは 1 バイト (あるいは複数バイト) の場合にも適用できます。1 バイトであれば、8 ビットのうちのどこか 1 ビットでも、1 から 0 に変化した場合に演算結果が 0 以外の数値になります。値を調べることでどのビットが変化したかも分かります。

例えば、結果が 128 だったら、最上位ビットが 1 から 0 に変化したということになるわけです。結果が 3 であれば、最下位の 2 ビットが 1 から 0 に変化したことになります。

もちろんバイト単位や 1 ビットだけでなく、複数ビット (ビットが隣り合っていなくてもよい) について、1 から 0 への変化をみることもできます。

論理積を使って、不要なビットに関してはビットを 0 にマスクしてしまえば良いわけです (図 2.23)。

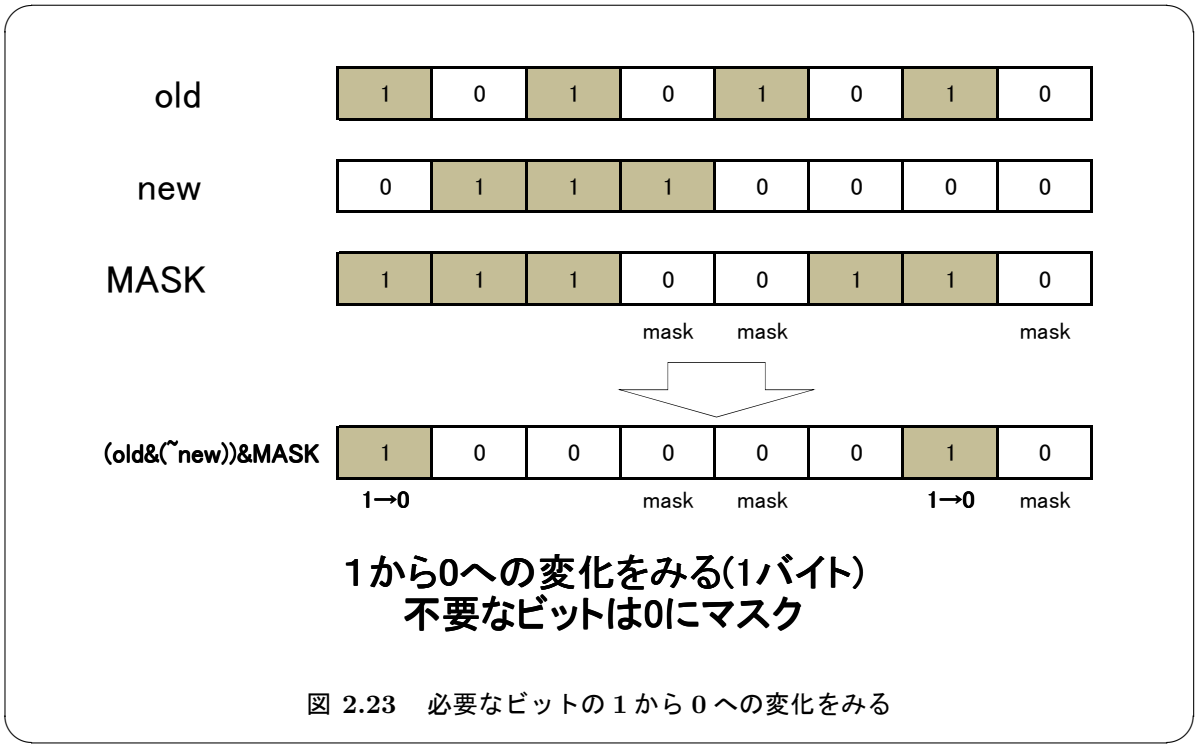


図 2.23　必要なビットの 1 から 0 への変化をみる

図 2.23 では、old が $(10101010)_2$ で、値が変化して new が $(01110000)_2$ になった場合を考えています。この場合には、右から 2 番目のビットと右から 4 番目のビット、一番左のビットが 1 から 0 に変化しています。しかし、MASK$(11100110)_2$ を用いて、一番右、右から 4 番目、右から 5 番目のビットは 0 にマスクしています。そのために、`(old&(~new))&MASK` の結果は右から 2 番目と一番左のビットだけが 1 になっているのです。

このようにして、必要なビットの変化だけを見ることもできるわけです。

演習 2.5- 7

論理和を用いて、1 から 0 への変化をみる方法を考えてください。
また、必要のないビットをマスクする方法についても考えてみましょう。

2.5.14 0 から 1 への変化をみる

0 から 1 への変化をみる方法も同様にして得ることができます。

ここでも以前の値を old、現在の値を new とします。今回は old が 0、new が 1 のときを見たいのですから、old を反転させれば両方が 1 になります。

したがって、`(~old) & new` という演算結果が 1 のときに、0 から 1 に値が変化したということが分かります。それ以外の場合には演算結果は 0 になります (図 2.24)。

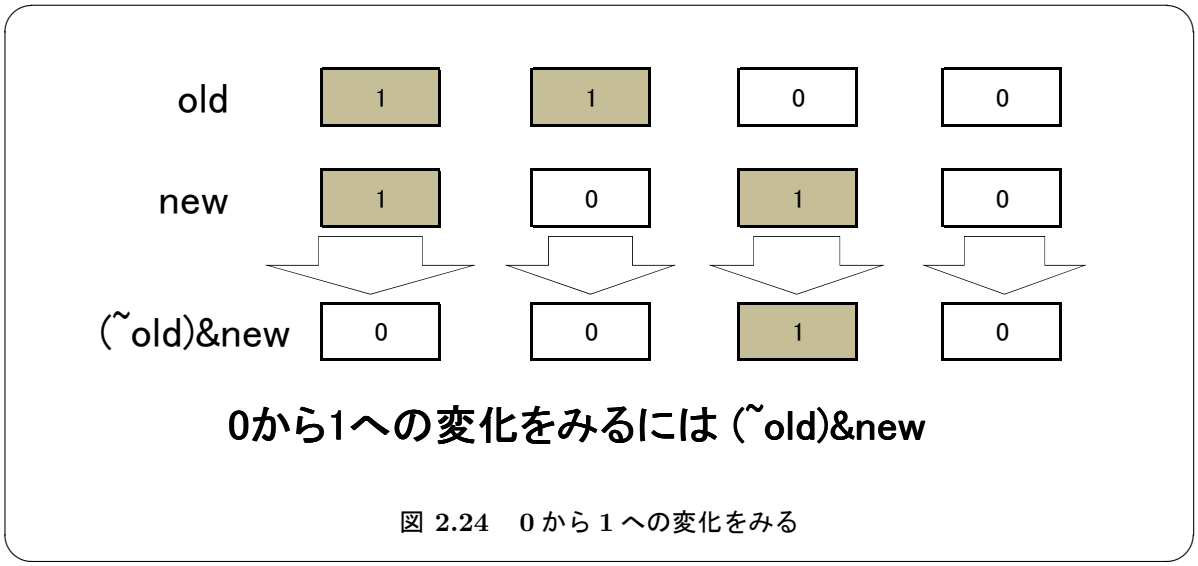


図 2.24　0 から 1 への変化をみる

もちろん論理積を使って、不要なビットに関してはビットを 0 にマスクして、特定のビットについて 0 から 1 への変化をみることができます。

演習 2.5- 8

論理和を用いて、0 から 1 への変化をみる方法を考えましょう。

2.6 ビット演算の利用例

ここまでは、ビット演算の基本とその利用例を説明してきました。ここでは、これらのビット演算を用いて移植性の高いプログラムを組む際に必要と思われる事柄をまとめておきましょう。

ここで想定するのは以下の事柄です。

周辺機器の機能を使うための関数の仕様を決めておき、ハードウェアが違っても同じ関数が使えるようにします。

関数にはレジスタを設定するためのデータなどを引数として与えますが、引数の使い方は常に同じにしておきます。ハードウェアの違いなどは関数内のプログラムで吸収するように作りましょう (図 2.25)。

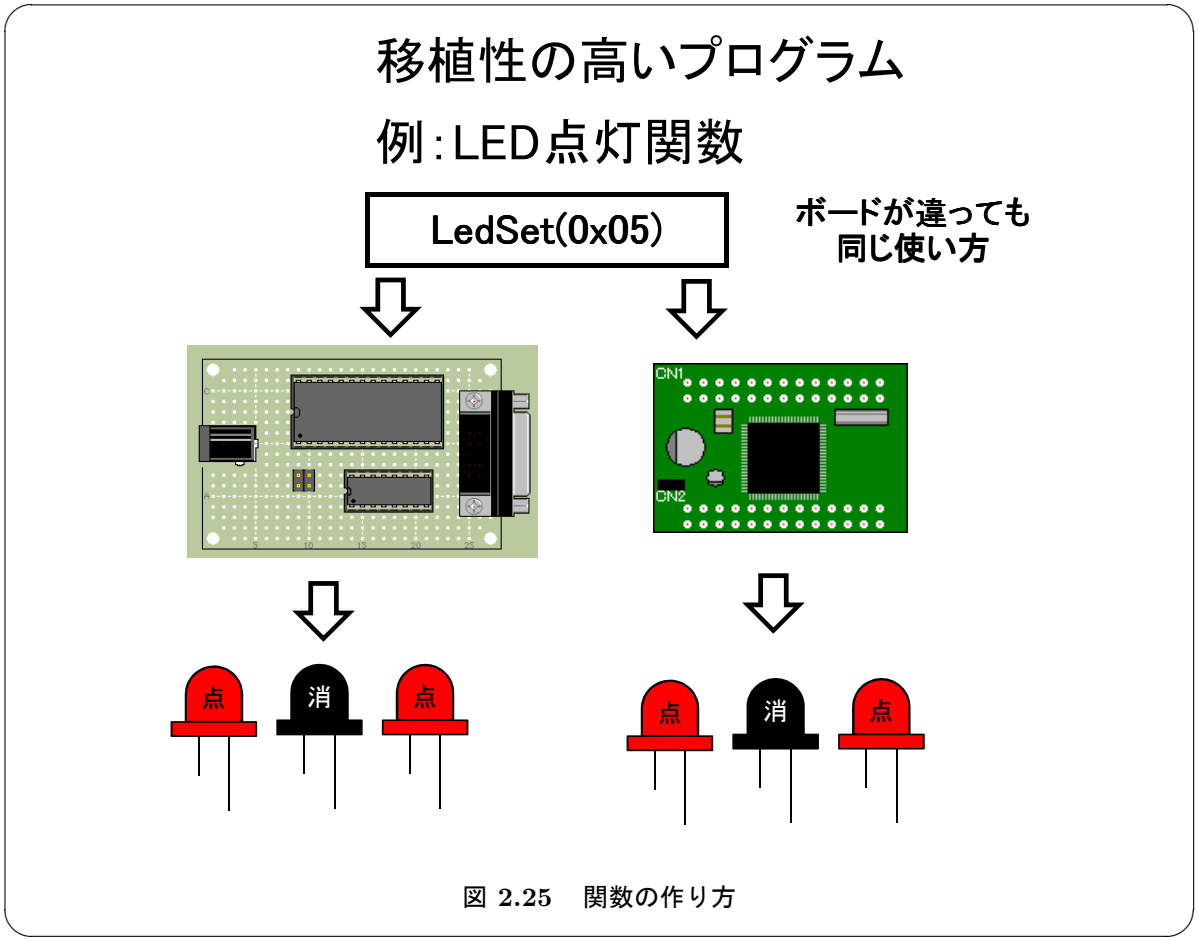


図 2.25 関数の作り方

このような関数を作る場合、どのようなビット演算を利用する可能性があるか考えてみましょう。

具体例として、LED を点灯させる関数 LedOn を考えてみましょう。

ここで作るのは引数として 1 バイト分を渡し、ビットの値が 1 の LED を点灯させるものです。0 のところは何もしません。引数の渡し方は、最下位ビットが 1 であれば LED0 が、2 番目のビットが 1 であれば LED1 が、3 番目のビットが 1 であれば LED2 が点灯することにします。例えば、5 を渡せば LED0 と LED2 が点灯することになるわけです。

引数の値が 0 のビットは元の状態のままで、点灯状態であれば点灯状態のまま変わらず、消灯状態であ

れば消灯状態のまま変わりません。

LED の回路は図 2.26 であるとします。

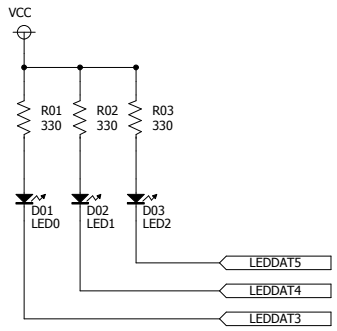


図 2.26 LED の回路

LED の点灯消灯はレジスタ LEDDAT に値を代入することで指定するとします。また、LEDDAT3 は LEDDAT の 4 ビット目、LEDDAT4 は LEDDAT の 5 ビット目、LEDDAT5 は LEDDAT の 6 ビット目をあらわしています。

なお、レジスタ LEDDAT の最下位ビットは LEDDAT0 です。

レジスタ LEDDAT3(LEDDAT の 4 ビット目) を 0 に設定すると端子は 0V になり LED0 が点灯します。同様に LEDDAT4(LEDDAT の 5 ビット目) を 0 に設定すると LED1 が、LEDDAT5(LEDDAT の 6 ビット目) を 0 に設定すると LED2 が点灯します。

この回路に対して LedOn を作るためには、引数として 1 を渡された場所のレジスタを 0 に設定する必要があるわけです。引数の値とレジスタに設定する値の関係は、回路が変わると変わることに注意してください。この対応はプログラムを用いて行うことになります。

このような条件で関数 LedOn をプログラムすると、以下のようになります。

LED 点灯関数の例

```
1  /*******************************
2    1 を指定した LED を点灯
3  *******************************/
4  void LedOn(unsigned char led_data)
5  {
6    LEDDAT &= (~((led_data<<LED_SHIFT) & LED_MASK)); /* 0 をセット */
7  }
```

End Of List

4 行目の引数 unsigned char led_data には、1 バイトのデータが渡されます。LedOn(0x05) のように関数を呼び出すと、unsigned char led_data には 5 が入ることになります。

ただし、渡されるデータが常に正しく 3 ビット分だけである保証はありません。それ以外のビットにも値を設定される可能性がないとはいえません。そのために関数の中で、不要なビットはマスクして 0 もしくは 1 に設定してしまうほうが、プログラムが誤動作する可能性が少なくなるでしょう。

上記の LedOn の演算を図示してみましょう。引数 led_data とレジスタ LEDDAT の関係は図 2.27 のようになります。

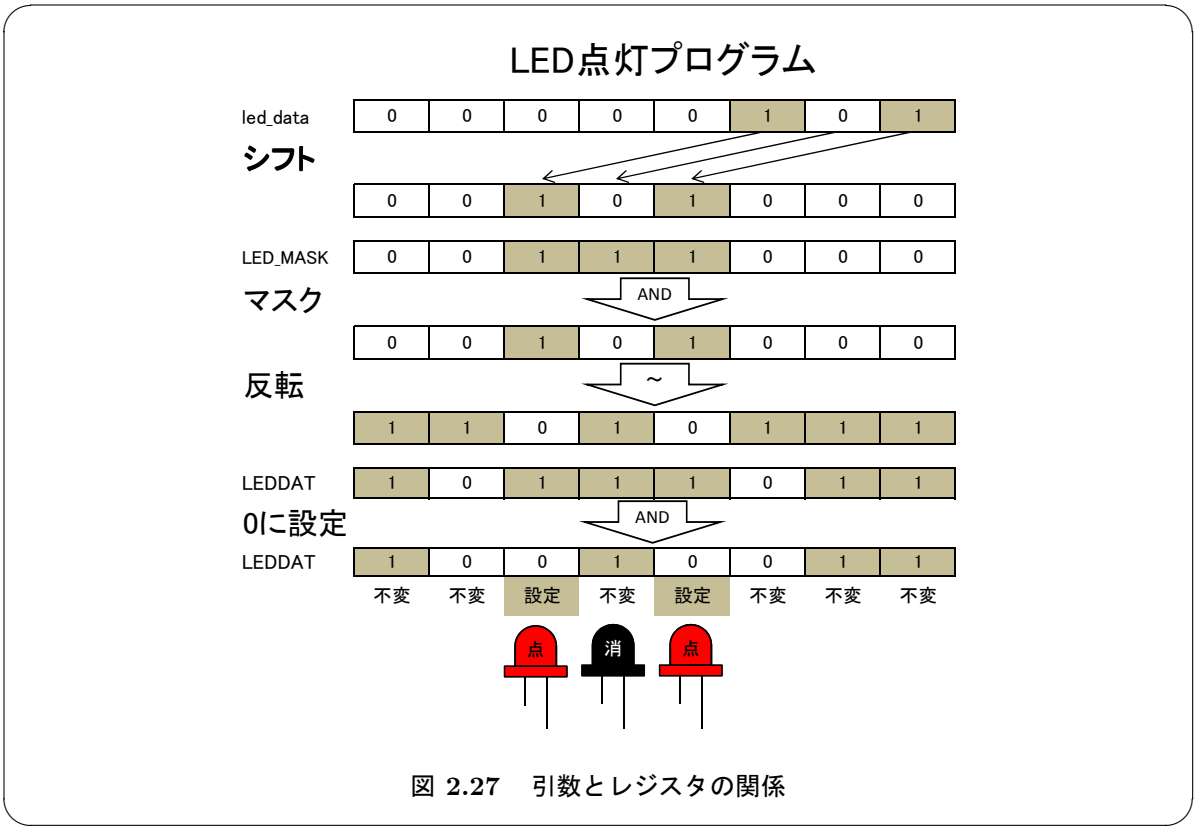


図 2.27　引数とレジスタの関係

ここでは、以下のような操作が行われているわけです。

- 渡されたデータ led_data を 3 ビット左にシフト
- 必要なビット以外を 0 にマスク
- ビットの値をすべて反転
- レジスタ LEDDAT と AND を取り、0 のところを 0 に設定

それぞれの操作は以下のような場合に必要になります。

- シフトは周辺機器 (LED など) が最下位ビットから連続して使われていないため
- マスクは 1 バイト全体が一つの周辺機器に使われていないため
- 反転は渡されたデータの 1 の場所を 0 にあるいは 0 の場所を 1 に設定するため

ここではもっとも複雑な場合を考えてみましたが、いきなりこれを考えるのではなく、以下は簡単なものから徐々に複雑な演算へとステップアップしていくことにしましょう。

2.6.1 値が 0 だったらレジスタを 1 に設定する

P.23 の 2.5.2 で論理積 (AND) を用いると、必要なビットだけ 0 に設定し、他のデータを変えないことができるという説明をしました。今考えている関数の例では、引数の値が 0 のビットに対してレジスタの値を 0 に設定する場合に論理積を用いるということになります。

同様に、P.30 で説明した、論理和 (OR) の、必要なビットだけ 1 に設定し、他は変えないという使い方はどうでしょうか。今考えている関数の例では、引数の値が 1 のビットに対してレジスタの値を 1 に設定する場合に用いればよいことが分かります。

P.47 の 2.6 で LED の回路について説明しました。そこで説明した関数では、引数として 1 を渡されたビットのレジスタを 0 に設定する必要がりました。

このように、引数の値が 0 のビットに対してレジスタの値を 1 に設定するとか、あるいは、引数の値が 1 のビットに対してレジスタの値を 0 に設定するという操作が必要になってきます。

以下では、これまで学んできたことを組み合わせて、この方法について学習しましょう。

そこでまずは、引数の値が 0 のビットに対してレジスタの値を 1 に設定する演算を考えてみましょう。あるビットを 1 に設定して他を変えないという操作には論理和 (OR) を使えばよいのでした。今の場合には引数が 0 のビットに対してレジスタを 1 に設定したいので、引数の値を反転させる必要があります。

したがって、設定するレジスタを PBDR、引数を data とすると、実行するべきビット演算は次のようになります。

$$\mathrm{PBDR} \mid = (\sim \mathrm{data})$$

このとき、data の値が 1 になるビットに対しては、PBDR の値は変更されないことを確認しておいてください。図 2.28 では、引数の値が $(10101010)_2$ の場合を考えてみました。

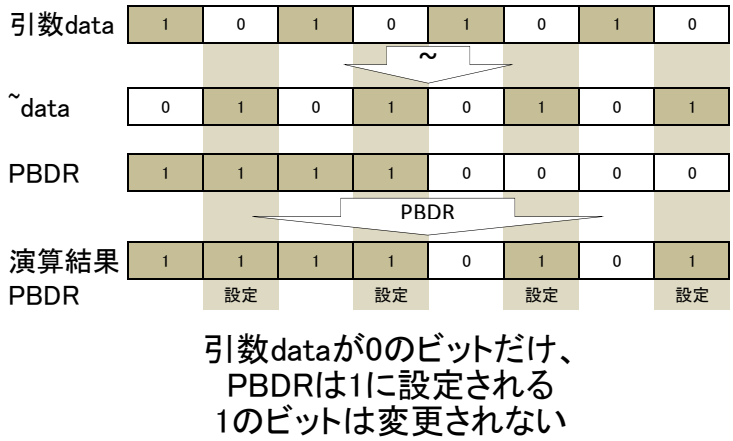


図 2.28 引数が 0 のビットに対してレジスタを 1 に設定

演算結果が変更されているビットは、引数 data の値が 0 であること、設定された値は 1 であることを確認してください。また、引数 data の値が 1 であるビットは、演算結果がもとの PBDR の値と変わってないことも確認しておいてください。

2.6.2 値が 1 だったらレジスタを 0 に設定する

2.6.1 と同様に、引数の値が 1 のビットに対してレジスタの値を 0 に設定する演算を考えてみましょう。あるビットを 0 に設定して他を変えないという操作には論理積 (AND) を使えばよいのでした。今の場合にも引数が 1 のビットに対してレジスタを 0 に設定したいので、引数の値を反転させる必要があります。

したがって、設定するレジスタを PBDR、引数を data とすると、実行するべきビット演算は次のようになります。

$$\mathrm{PBDR}\ \&= (\sim \mathrm{data})$$

このとき、data の値が 0 になるビットに対しては、PBDR の値は変更されないことを確認しておいてください。図 2.29 では、引数の値が $(10101010)_2$ の場合を考えてみました。

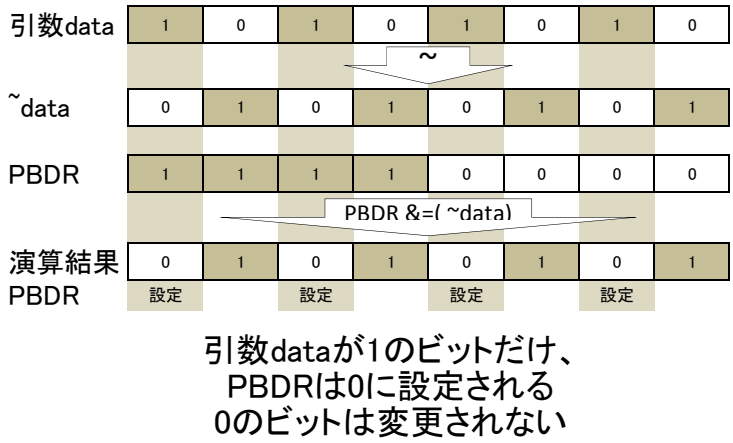


図 2.29 引数が 1 のビットに対してレジスタを 0 に設定

演算結果が変更されているビットは、引数 data の値が 1 であること、設定された値は 0 であることを確認してください。また、引数 data の値が 0 であるビットは、演算結果がもとの PBDR の値と変わってないことも確認しておいてください。

以上、引数で与えられたビットの値をレジスタに反映させる演算を説明してきました。まとめると以下の表 2.8 のようになります。

表 2.8 引数で与えられたビットの値をレジスタに反映させる演算

引数 (data)	レジスタ (PBDR)	演算
0	0	PBDR &= data
1	変更されず	
0	変更されず	PBDR |= data
1	1	
0	1	PBDR |= (~data)
1	変更されず	
0	変更されず	PBDR &= (~data)
1	0	

2.6.3 不要なビットをマスクし、値が 0 だったらレジスタを 0 に設定する

表 2.8 に引数で与えられたビットの値をレジスタに反映させる演算をまとめました。そこでの演算では、引数で与えられるデータは 1 バイト全体だということに注意してください。

必要なビットだけ 0 に設定し、他は変えない論理積や、必要なビットだけ 1 に設定し、他は変えない論理和は、I/O ポートの入出力を設定する際などに良く使われます。

関数内で利用される場合には、引数で与えられたデータをレジスタに反映させる演算に使うことができます。たとえば、0 のところは 0 に 1 のところは 1 に設定するというような使い方です。この場合には、`PBDR &= data` と `PBDR |= data` の演算を両方使うことになります。ただし、ここでは 1 バイト全体を設定しているので、このように演算を 2 回行う必要はなく、引数の値を直接 `PBDR = data` とレジスタに入れてしまえば充分でしょう。値を反転させて設定する場合も同様です。

このような演算が重要なのは、一部のビットにのみ値を反映させるという場合です。

たとえば以前に LED が 3 個だけつながっている場合を考えました。この場合には引数のデータのうち 3 ビット分だけが意味があり、他のデータはレジスタに反映させたくないわけです。

そこで、ここでは引数として受け取ったデータの一部だけを反映させる場合を考えてみましょう。

表 2.8 のそれぞれについて、マスクをかけて一部のデータを使うように拡張します。

仕様として、マスクをかけるための値を MASK という名前で定義するものとし、1 のビットはデータを残し、0 のビットはデータを反映させないものとします。たとえば、MASK の値が $(00001111)_2$ であるとき、上位 4 ビットは値をマスクして残さず、下位 4 ビットは残すものとします。0 でマスクするか 1 でマスクするかは、引数で与えた値が仕様通りに反映されるように決めます。

まずは不要なビットをマスクし、引数の値が 0 のビットに対してレジスタの値を 0 に設定する演算を考えてみましょう。この演算は、表 2.8 からも分かるように、`PBDR &= data` でした。

ここではこの演算に、data にマスクをかける演算を追加すればよいことになります。

注意しなければならないのは、データをマスクするための値 MASK を、1 のビットはデータを残し、0

のビットはマスクすると仕様を決めたことです。最終的に PBDR と論理積を取ることになるので、0 でマスクしてしまうとその値が PBDR に反映されてしまいます。そこで 1 でマスクするように MASK の値は反転させる必要がありそうです。この場合 MASK を反転させた値と data との論理和を取れば良いことが分かります。

以上より、不要なビットをマスクし、引数の値が 0 のビットに対してレジスタの値を 0 に設定する演算は、

$$\text{PBDR} \ \&= \ (\ \text{data} \mid (\ \sim\text{MASK}\ )\ )$$

とすれば良いことが分かります。ここで、MASK はデータを残すビットを 1 に設定した値です。

注意

ここでは演算の方法を上記のようにしましたが、書き方は一つだけではありません。たとえば、上記の方法以外に

```
PBDR &=(~((~data) & MASK))
```

と書くこともできるでしょう。MASK を反転させず、data と論理積を取っているのでこのほうが良いと感じる人もいるかもしれません。
ここでは、単に演算の少ない上記の方法を使いました。
以下の演算でも、結果が同じになる演算は他にもあることに注意してください。
このテキストでは、data を加工し最後にレジスタと論理積もしくは論理和を取るようにし、できるだけ演算が少ない方法を利用するようにしています。

2.6.4 不要なビットをマスクし、値が 1 だったらレジスタを 1 に設定する

マスクをかけるための値 MASK の仕様は 2.6.3 と同じにして、不要なビットをマスクし、引数の値が 1 のビットに対してレジスタの値を 1 に設定する演算を考えてみましょう。

引数の値が 1 のビットに対してレジスタの値を 1 に設定する演算は、P.52 の表 2.8 からも分かるように、`PBDR |= data` でした。この演算に、data にマスクをかける演算を追加すればよいことになります。

最終的に PBDR と論理和を取ることになるので、MASK と data との論理積を取れば良いことが分かります。

したがって、不要なビットをマスクし、引数の値が 1 のビットに対してレジスタの値を 1 に設定する演算は、

$$\text{PBDR} \mid = \ (\ \text{data} \ \&\ \text{MASK}\ )$$

となります。MASK はデータを残すビットを 1 に設定した値です。

引数の値の一部をレジスタに反映させることを考えてみましょう。ここでは引数の特定のビットのデータを抜き出し、それが 0 なら対応するレジスタを 0 に、1 なら対応するレジスタを 1 に設定する場合です。

一部だけ値を反映し、他は何もしないようにしたいわけです。2.6.3 で説明した不要なビットをマスクし、値が 0 だったらレジスタを 0 に設定する演算と、ここで説明した不要なビットをマスクし、値が 1 だったらレジスタを 1 に設定する演算を続けて行えばよいわけです。

まとめ :
与えられたデータ data の一部を抜き出し、レジスタ PBDR に反映させるためには、
PBDR & = (data | (~MASK))
PBDR | = (data & MASK)
を行えばよい。
ただし、MASK は 1 のビットはデータを残し、0 のビットはデータを反映させないものとする。

2.6.5 不要なビットをマスクし、値が 0 だったらレジスタを 1 に設定する

2.6.3 と 2.6.4 では引数の値が 0 のビットに対してレジスタの値を 0 に、引数の値が 1 のビットに対してレジスタの値を 1 に設定しました。

ここでは、不要なビットをマスクし、引数の値が 0 のビットに対してレジスタの値を 1 に設定する演算を考えてみましょう。

2.6.3 や 2.6.4 と同様に、マスクしない演算から考えてみす。

引数の値が 0 のビットに対してレジスタの値を 1 に設定する演算は、P.52 の表 2.8 では、`PBDR |= (~data)` でした。この演算に、data にマスクをかける演算を追加すればよいことになります。

したがって、不要なビットをマスクし、引数の値が 0 のビットに対応したレジスタを 1 に設定する演算は、

$$\text{PBDR} \mid = ((\sim\text{data}) \,\&\, \text{MASK})$$

とすれば良いことが分かります。

2.6.6 不要なビットをマスクし、値が 1 だったらレジスタを 0 に設定する

ここでは、不要なビットをマスクし、引数の値が 1 のビットに対してレジスタの値を 0 に設定する演算を考えてみましょう。

今回もマスクしない演算から考えてみます。

引数の値が 1 のビットに対してレジスタの値を 0 に設定する演算は、P.52 の表 2.8 では、`PBDR &= (~data)` でした。この演算に、data にマスクをかける演算を追加すればよいことになります。

引数の値が 0 のビットに対してレジスタの値を 0 に設定する演算は、`PBDR &= data` でした。これに対して、不要なビットをマスクし、引数の値が 0 のビットに対してレジスタの値を 0 に設定する演算は、`PBDR &= ( data | (~ MASK))` でした。

この類推からすると、不要なビットをマスクし、引数の値が 0 のビットに対してレジスタの値を 1 に設定する演算は、`PBDR &=(( ~data) | (~MASK))` とすれば良いことが分かります。ただし、これはド・モルガンの法則から、

$$\text{PBDR \&= (~(data \& MASK))}$$

と書くことができます。ここでは、演算の少ないこちらを利用することにします。

まとめ :
与えられたデータ data の一部を抜き出し、レジスタ PBDR に反転して反映させるためには、
PBDR |= ((~data) & MASK)
PBDR &= (~(data & MASK))
を行えばよい。ただし、MASK は 1 のビットはデータを残し、0 のビットはデータを反映させないものとする。

引数で与えられたビットの値を値 MASK でマスクしてレジスタに反映させる演算を説明してきました。まとめると以下の表 2.9 のようになります。

表 2.9 引数で与えられたビットの値を値 MASK でマスクしてレジスタに反映させる演算

引数 (data)	レジスタ (PBDR)	演算
0	0	PBDR &= (data | (~MASK))
1	変更されず	
0	変更されず	PBDR |= (data & MASK)
1	1	
0	1	PBDR |= ((~data) & MASK)
1	変更されず	
0	変更されず	PBDR &= (~(data & MASK))
1	0	

ただし、MASK は 1 のビットはデータを残し、0 のビットはデータを反映させないものとする。

2.6.7 値をシフトして不要なビットをマスクし、レジスタを設定する

さて、最後は値をシフトさせることを考えます。すでに説明したように、関数に渡された値は、レジスタを設定する際にシフトする必要がある場合があります。最下位ビットからレジスタを使ってない場合です。

なお、ここではシフトはするけれどマスクはしないという場合は考えません。シフトをする場合には必ずマスクをかけるということにしておきます。

また、シフトとマスクの順番ですが、引数として渡されたデータをまずシフトすることにします。その後マスクをかけるという手順です。この場合、マスクの値もシフトしてから利用することにします。詳しいことは、以下の説明で確認してください。

引数で渡されたデータをまずシフトしてからマスクして、レジスタを設定するとしました。このため、マスクしてレジスタの値を設定するという前述の演算に、シフトを入れれば今目的の演算が得られます。引数のビットが 0 か 1 か、それに対応してレジスタを 0 と 1 のどちらに設定するかで 4 通りの場合があります。具体的に書き下してみると、

- 値をシフトして不要なビットをマスクし、値が 0 だったらレジスタを 0 に設定する
- 値をシフトして不要なビットをマスクし、値が 1 だったらレジスタを 1 に設定する
- 値をシフトして不要なビットをマスクし、値が 0 だったらレジスタを 1 に設定する
- 値をシフトして不要なビットをマスクし、値が 1 だったらレジスタを 0 に設定する

です。それぞれの演算は、シフトがない場合は P.55 の表 2.9 にまとめたとおりです。ここでは data に対してシフトをしてやれば良いだけです。同様に表にすると、表 2.10 のようになります。ただし、MASK は 1 のビットはデータを残し、0 のビットはデータを反映させないものとします。また、SHIFT は左シフトするビット数をあらわすものとします。通常このような使い方では右シフトを行うことは無いので、ここでは左シフトのみを想定しています。

表 2.10 引数で与えられたビットの値を値 MASK でマスクしてレジスタに反映させる演算

引数 (data)	レジスタ (PBDR)	演算
0	0	PBDR &= ((data << SHIFT) | (~MASK))
1	変更されず	
0	変更されず	PBDR |= ((data << SHIFT) & MASK)
1	1	
0	1	PBDR |= ((~(data << SHIFT)) & MASK)
1	変更されず	
0	変更されず	PBDR &= (~((data << SHIFT) & MASK))
1	0	

ただし、MASK は 1 のビットはデータを残し、0 のビットはデータを反映させないものとする。
また、SHIFT は左シフトするビット数をあらわすものとする。