

## 中位 P C (16F84)留意点

---



目次

---

はじめに	3
------	---

---

ホ-初期化	4
条件分岐	5
テーブル処理	6
PC操作	7
プロット転送	8

---

演算	9
補数	9
ハ`付加算	12
ハ`付減算	14
ハ`付乗算	16
ハ`付除算	19
ハ`付リ ハ`付化 BCD変換	23
ハ`付化 BCD ハ`付変換	26
ハ`付化 BCD加算	30
ハ`付化 BCD減算	32

---

TMR0の使用法	34
----------	----

---

応用	36
入出力の拡張	36
RS-232C	38

---

命令一覧表	39
-------	----

はじめに

本書のプログラム部分は、以下のシンボル定義済みとして記述しています。プログラム部分を利用する場合は、以下をソース内に記述するか、以下を定義ファイルとして作成し、ソース内でインクルードしてください。

.16f84		;PIC 16F84に設定	
W	EQU	0	;作業レジスタ
F	EQU	1	;ファイルレジスタ
INDIRECT	EQU	0	;00: 間接アドレッシング
INDF	EQU	0	;00:
RTCC	EQU	1	; (RTCC = TMR0 = 1)
TMR0	EQU	1	;01: タイムテータ
OPTION	EQU	1	;81: 任意選択制御
RBPV	EQU	1.7	;OPTIONビット7
INTEDG	EQU	1.6	;OPTIONビット6
RTS	EQU	1.5	;OPTIONビット5
RTE	EQU	1.4	;OPTIONビット4
PSA	EQU	1.3	;OPTIONビット3
PS2	EQU	1.2	;OPTIONビット2
PS1	EQU	1.1	;OPTIONビット1
PS0	EQU	1.0	;OPTIONビット0
PC	EQU	2	;=PCL
PCL	EQU	2	;02:82: プログラムカウンタ下位8ビット
STATUS	EQU	3	;03:83: ステータス
IRP	EQU	3.7	;STATUSビット7 indirect page select
RP1	EQU	3.6	;STATUSビット6 page select 1
RP0	EQU	3.5	;STATUSビット5 page select 0
TO	EQU	3.4	;STATUSビット4 time out bit
PD	EQU	3.3	;STATUSビット3 power down bit
Z	EQU	3.2	;STATUSビット2 zero flag
DC	EQU	3.1	;STATUSビット1 digit carry/borrow flag
C	EQU	3.0	;STATUSビット0 carry/borrow flag
c	EQU	3.0	
FSR	EQU	4	;04:84: 間接アドレスポインタ
PORTA	EQU	5	;05: ポートA (RA)
RA	EQU	5	;05: ポートA (RA)
TRISA	EQU	5	;85: ポートA方向レジスタ
ra	EQU	5	
PORTB	EQU	6	;06: ポートB (RB)
RB	EQU	6	;06: ポートB (RB)
TRISB	EQU	6	;86: ポートB方向レジスタ
rb	EQU	6	
EEDATA	EQU	8	;08: EEPROMデータ
EEADR	EQU	9	;09: EEPROMアドレス
EECON1	EQU	8	;88: EEPROM制御 1
EEIF	EQU	8.4	;88: EEPROM書き込み終了フラグ
VRERR	EQU	8.3	;88: EEPROM書き込み打ち切りフラグ
WREN	EQU	8.2	;88: EEPROM書き込み許可フラグ
WR	EQU	8.1	;88: EEPROM書き込み開始フラグ
RD	EQU	8.0	;88: EEPROM読み出し開始フラグ
EECON2	EQU	9	;89: EEPROM制御 2
PCLATH	EQU	0AH	;0A:8A: プログラムカウンタ上位
INTCON	EQU	0BH	;0B:8B: 割り込み制御
GIE	EQU	0BH.7	;INTCONビット7
EEIE	EQU	0BH.6	;INTCONビット6
TOIE	EQU	0BH.5	;INTCONビット5
INTE	EQU	0BH.4	;INTCONビット4
RBIE	EQU	0BH.3	;INTCONビット3
TOIF	EQU	0BH.2	;INTCONビット2
INTF	EQU	0BH.1	;INTCONビット1
RBIF	EQU	0BH.0	;INTCONビット0

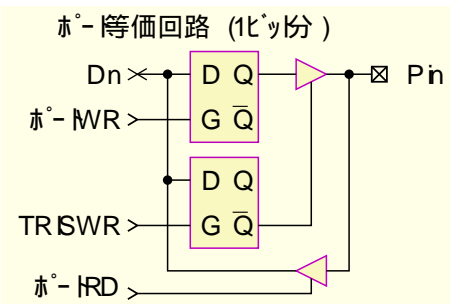
### 内蔵ポート初期化

PIC内蔵のポートはビット単位で入出力に設定可能です。通常、プログラム実行開始時に、入出力状態を決定するためにポートの初期化を行います。基本的な初期化手順を次に示します。

```

MOVW 00011100b ;RA ビットデータをWに取得
MOVF RA        ;RA⇒HHHLLに設定
MOVW 00000100b ;RB ビットデータをWに取得
MOVF RB        ;RB⇒LLLLLHLLに設定
BSF  RPO       ;レジスタファイル1に切替
MOVW 00000001b ;RA方向ビットデータを設定
MOVF TRISA     ;RA方向0000に設定
MOVW 11110001b ;RB方向ビットデータを設定
MOVF TRISB     ;RB方向1111000に設定
CLRWD T       ;ウォッチドッグ タイム初期化
MOVW 00001001b ;OPTION ビットデータを設定
MOVF OPTION    ;プルアップ有, 内部, WDT-4分周
BCF  RPO       ;レジスタファイル0に切替
    
```

ここでのポイントはRA RBの方向設定以前に出力データを設定することです。この順ですと、出力ポートの初期値を確定することが可能です。



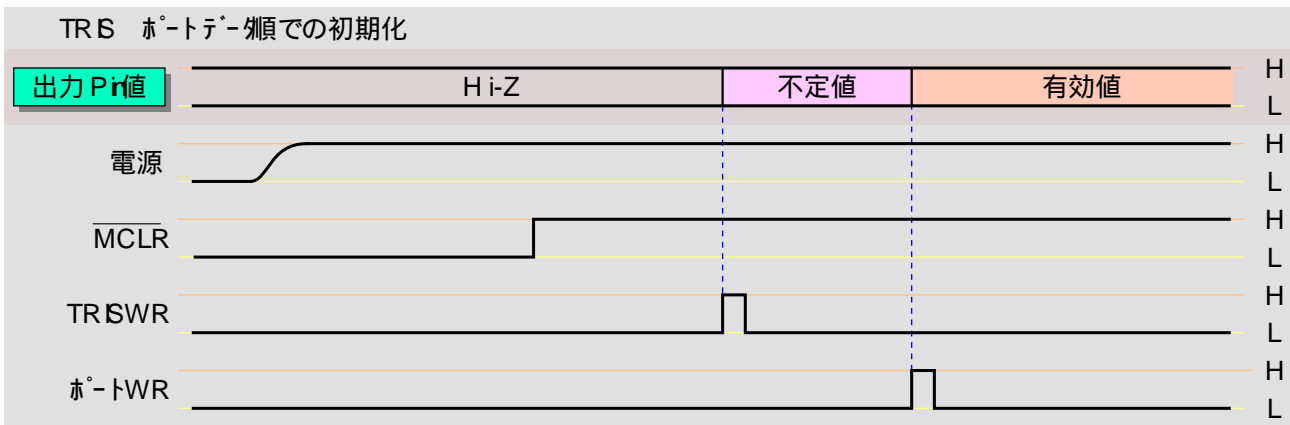
入出力Pinの入出力決定条件は次の2点です。

- リセット中は入力
- リセット後のTRISレジスタ値 (1=入力, 0=出力)

リセット時のTRISレジスタQ出力値は1(入力)で、ポートレジスタ値は不定です。この場合、出力に設定するポートでは、TRISレジスタの該当するビット=設定が行われるまで、対象ポートが入力であることに留意しなければなりません。通常、出力Pinが接続される回路の入力は、この期間の入力値を固定するため、抵抗によるプルアップまたはプルダウン処理を行います。

この処理を行っても、リセット時に予期せぬ動作となる場合があります。その原因はポートデータの初期化とTRISレジスタの入出力設定の順序に起因します。

方向が出力に設定されると、入力(Hi-Z)から、出力ラッチの出力をPinに出力するようになります。ポートデータの出力ラッチの初期値は不定ですから、先にTRISレジスタで入出力を設定すると、Pin出力は不定となってしまいます。出力ポートの出力データラッチ自身は常に有効ですので、出力として使用するビットには、TRISレジスタで方向を設定する前に、予め初期値を設定しておきます。



通常、この後にRAMの内部変数などの初期化を行い、割り込みの初期化を行います。

### 条件分岐

PCには、多くのCPUに存在する条件分岐命令がありません。唯一、無条件分岐命令 **GOTO** が存在するのみです。条件分岐は、この **GOTO** 命令と条件スキップ命令を組み合わせることで実現します。

中位PCには、条件スキップ命令として次の命令があります。

BTFSS	f, b	; f(ファイル RAM) のビット(b)が 1 で次命令をスキップ
BTFSC	f, b	; f(ファイル RAM) のビット(b)が 0 で次命令をスキップ
DECFSZ	f, d	; f(ファイル RAM) を -1し、結果が 0 でスキップ
INCFSZ	f, d	; f(ファイル RAM) を +1し、結果が 0 でスキップ

RAM上の1バイト変数 X の値を検査し、0 でラベル **ZERO** へ分岐する例を次に示します。

MOVF	X, F	変数 X の 0 検査
BTFSC	Z	; 0 以外 (Z=0) でスキップ
GOTO	ZERO	; 0 (Z=1) で分岐

このように、多くのCPUで記述する分岐条件を含む分岐命令の形式ではなく、分岐条件の否定を含むスキップ命令と無条件分岐命令の組み合わせになりますので、スキップ命令の条件記述には注意を要します。

このスキップ命令を積極的に利用すると、2つの条件のOR条件分岐が短縮して記述可能です。次は、フラグ変数 **FLG** のビットが 4 が 1 の場合に、ラベル **ACT** へ分岐する単純な記述です。

BTFSC	FLG, 0	; FLG のビット 0 = 0 でスキップ
GOTO	ACT	; FLG のビット 0 = 1 で ACT へ分岐
BTFSC	FLG, 4	; FLG のビット 4 = 0 でスキップ
GOTO	ACT	; FLG のビット 4 = 1 で ACT へ分岐

上記例は、次のように短縮して記述できます。

BTFSS	FLG, 0	; FLG のビット 0 = 1 でスキップ
BTFSC	FLG, 4	; FLG のビット 4 = 0 でスキップ
GOTO	ACT	; FLG のビット 4, 0 がともに 1 で分岐

上記例のようなビット単位の検査で3ビット以上となる場合は、Wレジスタを使用して一般の論理演算処理での分岐とする方が命令語数、速度共に有利です。

MOVF	FLG, W	; FLG 値を W に取得
ANDLW	01010101b	; ビット 6, 4, 2, 0 のみ有効
BTFSS	Z	; ビット 6, 4, 2, 0 が全て 0 でスキップ
GOTO	ACT	; ビット 6, 4, 2, 0 の何れかが 1 で ACT へ分岐

上記例は複数ビットのOR条件ですが、AND条件の場合は論理反転を行い、負論理 AND (正論理 OR) で判定します。

MOVF	FLG, W	; FLG 値を W に取得
<b>XORLW</b>	<b>OFFH</b>	論理反転 (負論理に変更)
ANDLW	01010101b	; ビット 6, 4, 2, 0 のみ有効
<b>BTFSC</b>	Z	; ビット 6, 4, 2, 0 の何れかが 1 元値で 0 でスキップ
GOTO	ACT	; ビット 6, 4, 2, 0 のすべてが 0 元値で 1 で ACT へ分岐

## テーブル分岐

テーブル分岐のテーブル部分は無条件分岐命令 **GOTO** を使用します。このテーブルを参照するには、テーブル番号を PC に加算することで実現します。PC への加算は、通常のハイキ加算命令 **ADDWF** を使用します。次にテーブル分岐の基本的な例を示します。

TBN	DS	1	分岐テーブル番号変数定義
TBJ	MOVF	TBN,W	;テーブル番号をWに取得
	ADDWF	PC	;プログラムカウンタに対応位置に変更
			;
	GOTO	JOB0	;テーブル番号=処理へ分岐
	GOTO	JOB1	;テーブル番号=処理へ分岐
	GOTO	JOB2	;テーブル番号=処理へ分岐

行実行時の PC は次の命令位置 先頭の **GOTO** を指しています。この PC に **TBN** を加算した値が新しい PC となり、その位置から実行されます。PC では全ての命令が 語長命令ですので、**TBN** の値はテーブルの番号 位置 を示すこととなります。

## 定数テーブル

中位 PIC では、その構造により 命令部とデータ部が明確に分離されています。このために、多くの CPU と同様な方法で命令部に定数を定義することができません。PIC で定数を定義するには、**RETLW** 命令を使用します。次に定数テーブルの例を示します。

TBN	DS	1	定数テーブル番号変数定義
CTB	MOVF	TBN,W	;テーブル番号をWに取得
	ADDWF	PC	;プログラムカウンタに対応位置に変更
			;
	RETLW	100	;W=100 呼び出し元へ復帰
	RETLW	150	;W=150 呼び出し元へ復帰
	RETLW	300	;W=300 呼び出し元へ復帰

基本的な動作は上記のテーブル分岐と同じです。計算されたテーブル アドレスで実行される **RETLW** 命令が、W レジスタに定数を設定し、呼び出し元へ復帰します。このルーチンを使用するには、予め **TBN** を設定し、**CALL** 命令で **CTB** を呼び出します。戻り値 W レジスタが目的の定数となります。

## 制限事項

上記例の方法には重要な制限事項があります。**PCLATH** の操作が行われていない場合、テーブルの最後が 00FFH 以下でなければいけません。**PCLATH** が操作されている場合は、**PCLATH** の値が上位アドレスとなる 256 語内にテーブルが存在していなければなりません。

これは PIC の構造上の制限で、**GOTO** や **CALL** 命令のように命令語内に実質的な PCH (プログラムカウンタ上位) 値が存在する場合には問題となりませんが、オペランドに直接 PC を指定する一般命令による PC 操作の場合、PCH には **PCLATH** の値が代入されることに起因します。

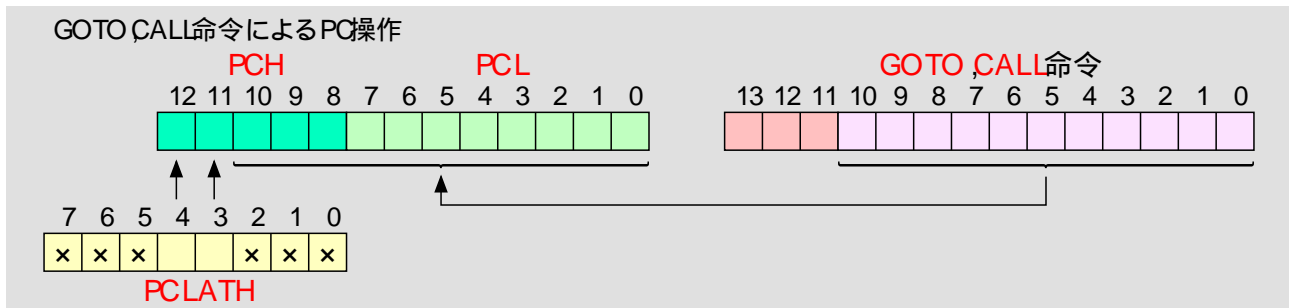
PC 操作に関する詳細と解決法については **PC 操作** を参照してください。

## PC操作

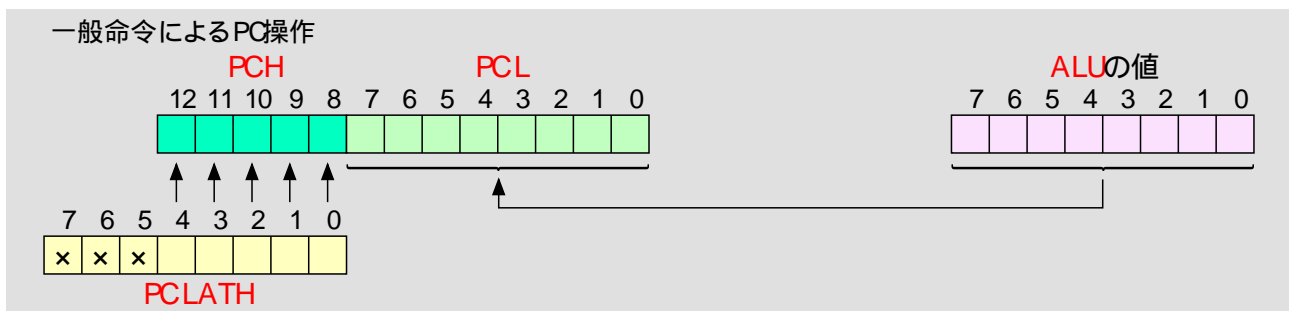
テーブル分岐や定数テーブルを使用する場合などに行われるPC操作では、PCH(プログラムカウンタ上位)の処理に注意しなければいけません。これはPCの処理がGOTO命令やCALL命令の場合と異なるためです。

中位PCのGOTO、CALL命令では、その命令語内に11ビットのPC値を保持しています。命令実行時には、この値がPCの下位11ビットになり、残りの上位ビットには、PCLATHの該当するビット値が代入されます。

これはPCLATHのビット4,3がプログラムメモリのバンク(アドレス)のような動作となることを意味します。但し、プログラム領域が11ビット(2K語)以下場合は、この動作を考慮せずに記述することができます。



一般命令を使用したPC操作書き込みでは、その命令自体で扱われるデータは8ビットですので、この8ビットデータがPCLになり、PCHにはPCLATHの低位5ビットが代入されます。



この場合、PC操作(演算)によるキャリーやホローはPCHに反映されませんので、予めPCLATHを設定する記述が必要になります。次にPCLに対する加算の場合の記述例を示します。

TBN	DS	1	分岐テーブル番号変数定義
TBJ	MOVLW	TBL/256	;テーブル基準アドレス上位をWに取得
	MOWF	PCLATH	;PCLATHを仮設定
	MOVLW	TBL	;テーブル基準アドレス下位をWに取得
	ADDWF	TBN,W	対応テーブルアドレスをWに取得
	BTFS	C	;キャリーなしでスキップ
	INCF	PCLATH,F	;PCLATH補正(+1)
			;
	MOWF	PC	;プログラムカウンタを対応位置に変更
			;
TBL	GOTO	JOB0	;テーブル番号=処理へ分岐
	GOTO	JOB1	;テーブル番号=処理へ分岐
	GOTO	JOB2	;テーブル番号=処理へ分岐

この例では、テーブルの先頭アドレス上位をPCLATHに仮設定し、次にテーブルアドレス下位を加算によって計算して、その結果のキャリーありでPCLATHを+1することで、PCへの書き込み時のPCLATHを予め設定しています。

テーブルが256境界内の場合は単純にPCLATHの設定だけの記述で済みます。また、プログラム領域が256未満(0000~00FFH)の場合には、これらの考慮が不要です。

一般的に、テーブル記述は256未満のアドレスに記述するのが最良です。

## ブロック転送

ブロック転送は間接ポインタを利用します。間接ポインタは体しかありませんので、転送すべき変数領域間のオフセット値を加減算することで対処します。次にブロック転送ルーチンの例を示します。

CNT	DS	1	転送バイト数カウンタ変数定義
OFS	DS	1	転送先-転送元オフセット変数定義
TMP	DS	1	;一時保存変数定義
BW	MOVWF	FSR	転送元基準アドレスを間接ポインタに設定
BW1	MOVF	INDF,W	転送元から1バイトをWに取得
	MOVWF	TMP	転送元データを一時保存
	MOVF	OFS,W	転送先-転送元オフセットをWに取得
	ADDWF	FSR,F	対応転送先アドレスを間接ポインタに設定
	MOVF	TMP,W	転送元データをWに復帰
	MOVWF	INDF	転送元データを転送先に設定
	DECWF	OFS,W	転送先-転送元オフセット-1をWに取得
	SUBWF	FSR,F	転送元間接ポインタを進行 設定
	DECFSZ	CNT,F	転送バイトカウンタを減数 =0でスキップ
	GOTO	BW1	転送バイト数分継続
			;
	RETURN		呼出元へ復帰

次に本転送ルーチンの使用例を示します。この例では変数領域Xから変数領域Yへ8バイトの転送を行っています。

X	DS	8	変数領域X定義
Y	DS	8	変数領域Y定義
	MOVLW	(Y-X)&0FFH	転送先-転送元オフセットをWに取得
	MOVWF	OFS	転送先-転送元オフセットを設定
	MOVLW	Y-X	転送バイト数をWに取得
	MOVWF	CNT	転送バイト数を設定
	MOVLW	X	転送元ポインタをWに取得
	CALL	BW	;ブロック転送

この転送ルーチンでは領域の下位アドレス側から複写を行います。このため、転送元と転送先が重複し、且つ転送元が転送先より下位側となる場合、内容が破壊されて正常な転送が行われませんので注意が必要です。

この場合は、転送ルーチンの **DECWF** を **NCF** に変更し、呼び出し時のWレジスタに設定する転送元ポインタを領域先頭ではなく、領域最終位置とすることで対処できます。この変更によって領域の上位アドレス側から複写を行うようになります。但し、この場合は転送元が転送先より上位側となる場合に正常な転送ができなくなります。

すべての状態で正常な転送を行うようにするには方向フラグ変数を定義し、その位置で **DECWF** か **NCF** のどちらかを使用する条件分岐の形式に変更します。とは言え、多くの場合、領域が重複し、且つ転送元と転送先の位置関係が両方存在する場合は殆どないでしょう。また、そのような状態にならないような変数構成などを考慮すべきです。

次に転送バイト数と実行サイクル数の関係を示します。

転送バイト数	命令語数	実行サイクル数	備考
2	12	24	実行命令サイクル数 = 転送バイト数 × 11 + 2
4		46	
6		68	
8		90	
10		112	
12		134	
14		156	
16		178	

### 1の補数

1の補数は単純に各ビットの論理反転値です。中位PICには1の補数を作成するCOMF命令があります。8ビット長の1の補数作成は、このCOMF命令で実現できます。

X	DS	1	;1バイト長変数 X定義
	COMF	X,F	変数 Xの 1の補数作成

多倍長の場合は単純に連続的なCOMF命令で実現します。

X	DS	3	;3バイト長変数 X定義
	COMF	X+0,F	変数 X最下位バイトの 1の補数作成
	COMF	X+1,F	変数 X第2バイトの 1の補数作成
	COMF	X+2,F	変数 X最上位バイトの 1の補数作成

### 2の補数

符号付き演算処理や減算を加算命令で行う場合などに、2の補数を作成する必要性が生じます。中位PICには1の補数(ビット反転)作成のCOMF命令はありますが、2の補数を作成する命令がありません。このため、2の補数を作成するにはCOMF命令を利用して複数の命令で行います。

X	DS	1	;1バイト長変数 X定義
	COMF	X,F	変数 Xの 1の補数作成
	INCF	X,F	変数 Xの 2の補数作成

多倍長の場合は1の補数時と異なり、直前のバイトからのキャリを含めて処理しなければなりません。

X	DS	2	;2バイト長変数 X定義
	COMF	X+0,F	変数 X最下位バイトの 1の補数作成
	INCF	X+0,F	変数 X最下位バイトの 1の補数作成
	;		
	BTFSC	Z	最下位バイトからのキャリなしでスキップ
	DECF	X+1,F	;キャリで第2バイト+1(実際には次補数のため -1)
	COMF	X+1,F	変数 X第2バイトの 2の補数作成

第2バイト目以降は上記第2バイト処理と同様な形式で記述します。何れの場合もキャリをZフラグで判定(Z=1)している点に注意してください。NCF命令ではキャリフラグが無効なので、直前値+1がキャリとなる場合に必ず結果が0になることを利用します。

命令語数と速度の関係

ビット数	バイト数	命令語数	実行サイクル数	備考
8	1	2	2	
16	2	5	5	
24	3	8	8	
32	4	11	11	
40	5	14	14	
48	6	17	17	
56	7	20	20	
64	8	23	23	

## ループ処理による10の補数

多倍長もビット数が多くなると命令語数的にループ処理が有利です。

X	DS	4	;4バイト長変数X定義
CNT	DS	1	;作業用カウンタ定義
	MOVLW	X	変数Xの基準ポインタをWに取得
	MOWF	FSR	変数Xの基準アドレスを間接ポインタに設定
	MOVLW	4	変数Xのバイト数をWに取得
	MOWF	CNT	;バイトカウンタを初期化
	;		
LOOP	COMF	INDF,F	対応バイトの10の補数作成
	INCF	FSR,F	間接ポインタを進行(+1)
	DECFSZ	CNT,F	;バイトカウンタを減数、=0でスキップ
	GOTO	LOOP	;0以外でバイト数分処理継続へ

次に上記ループ処理例と非ループ処理(必要数分のべた書き)での命令語数と実行サイクル数の関係を示します。

命令語数と速度の関係

ビット数	バイト数	命令語数		実行サイクル数		備考
		非ループ	ループ	非ループ	ループ	
8	1	1	8	1	8	
16	2	2		2	13	
24	3	3		3	18	
32	4	4		4	23	
40	5	5		5	28	
48	6	6		6	33	
56	7	7		7	38	
64	8	8		8	43	命令語数が同一

上記から64ビット(8バイト長を超える場合にループ処理での命令語数の方が少なくなることが判ります。然しながら実行速度は著しく遅くなり 同一命令語数となる64ビット長の場合で 倍以上の速度低下になります。

とは言え、同一プログラム内で異なるビット(バイト長)のデータを扱う場合、上記例のFSRとCOUNTの設定だけ変更すれば異なる長さの別データが扱えますので、命令語数的にかなり有利となります。

### ループ処理による2の補数

多倍長もビット数が多くなると命令語数的にループ処理が有利です。

X	DS	4	;4バイト長変数X定義
CNT	DS	1	作業用カウンタ定義
	MOVLW	X	変数Xの基準ポイントをWに取得
	MOWF	FSR	変数Xの基準アドレスを間接ポイントに設定
	MOVLW	4	変数Xのバイト数をWに取得
	MOWF	CNT	;バイトカウンタを初期化
	BSF	Z	初回用にZフラグを設定
;			
LOOP	BTFSC	Z	前桁からのキャリーなしでスキップ
	DECf	FSR,F	;キャリーで対応バイト+1実際には次補数のため-1)
	COMF	FSR,F	対応バイトの2の補数作成
	INCF	FSR,F	間接ポイントを進行(+1)
	DECFSZ	CNT	;バイトカウンタを減数、=0でスキップ
	GOTO	LOOP	;0以外でバイト数分処理継続へ

上記例の5行目のBSFは初回 最下位バイト処理時に7行目のDECfを実行させるために必要です。この5行目のBSFの代わりに、GOTOで8行目のDECfへ分岐させても、結果は同じです。この場合の命令語数と実行サイクル数も同じです。

10行目のNCFSZをNCFにはなりません。NCF命令はZフラグを更新します。このループでは9行目のCOMF命令の結果のZフラグを次の7行目のBTFSCに引き継ぐ必要があります。NCFSZ命令はZフラグを更新しません。

NCFSZ命令でFSRが0となった場合にスキップしますが、この場合のFSRは汎用レジスタGPRを指していますので、多くの場合、未実装領域となり現実的にはほぼ有り得ません。

次に上記ループ処理例と非ループ処理(必要数分のべた書き)での命令語数と実行サイクル数の関係を示します。

命令語数と速度の関係

ビット数	バイト数	命令語数		実行サイクル数		備考
		非ループ	ループ	非ループ	ループ	
8	1	2	11	2	11	命令語数が同一
16	2	5		5	18	
24	3	8		8	25	
32	4	11		11	32	
40	5	14		14	39	
48	6	17		17	46	
56	7	20		20	53	
64	8	23		23	60	

上記から32ビット(4バイト長を超える場合にループ処理での命令語数の方が少なくなることが判ります。しかし、実行速度は著しく遅くなり、同一命令語数となる32ビット長の場合で3倍以上の速度低下になります。

とは言え、同一プログラム内で異なるビット(バイト長)のデータを扱う場合、上記例のFSRとCOUNTの設定のみ変更すれば異なる長さの別データが扱えますので、命令語数的にかなり有利になります。

## バイト加算

中位PCIには8ビット(1バイト)長の加算命令 **ADDWF** 命令があります。8ビット(1バイト)長のバイト加算は、この **ADDWF** 命令で実現できます。次は1バイト長の  $X=X+Y$  の例です。

X	DS	1	;1バイト変数 X定義
Y	DS	1	;1バイト変数 Y定義
	MOVF	Y,W	変数 Y 加算値 を W に取得
	ADDWF	X,F	変数 X に変数 Y を加算

多倍長の場合は **ADDWF** 命令の連続で処理しますが、直前のバイトからのキャリを含めて処理しなければなりません。次は2バイト長の  $X=X+Y$  の例です。

X	DS	2	;2バイト変数 X定義
Y	DS	2	;2バイト変数 Y定義
	MOVF	Y+0,W	加算値最下位 バイトを W に取得
	ADDWF	X+0,F	最下位 バイト加算
	;		
	MOVF	Y+1,W	加算値第2バイトを W に取得
	BTFSZ	C	前桁からのキャリなしでスキップ
	INCFSZ	Y+1,W	;キャリで加算値 +1 を W に取得、=0 でスキップ
	ADDWF	X+1,F	前桁からのキャリなしで加算値 +1、0 で第2バイト加算

要点は5行目の **BTFSZ** と6行目の **INCFSZ** で2重の条件分岐 (スキップ) を行っていることです。前桁からのキャリなし時には7行目の **ADDWF** での加算となり、キャリあり時には加算値を+することで前桁からのキャリを反映しています。

更に、この加算値+1が0となる場合には7行目の **ADDWF** 命令をスキップしています。これは次桁用にキャリフラグ (=1) を保存するためです。加算値+1が0となる場合は今桁でキャリが発生することを意味します。7行目の **ADDWF** 命令を実行すると、今桁の結果は正しくありませんが、キャリなし (C=0) になってしまいます。

この対処のために7行目の **ADDWF** 命令をスキップする必要があります。直前の **BTFSZ** 命令の結果、**INCFSZ** 命令が実行される場合のキャリフラグは常に1です。

第2バイト目以降は上記第2バイト処理と同様な形式で記述します。次にビット(バイト)に対する命令語数と実行サイクル数の関係を示します。

命令語数と速度の関係

ビット数	バイト数	命令語数	実行サイクル数	備考
8	1	2	2	
16	2	6	6	
24	3	10	10	
32	4	14	14	
40	5	18	18	
48	6	22	22	
56	7	26	26	
64	8	30	30	

### ループ処理によるバイト加算

多倍長もビット数が多くなると命令語数的にループ処理が有利です。

X	DS	4	;4バイト長変数 X定義
Y	DS	4	;4バイト長変数 Y定義
CNT	DS	1	作業用 カウンタ定義
TMP	DS	1	加算値一時保存変数定義
FLG	DS	1	;キャリー フラグ一時保存変数定義
	MOVLW	X	変数 Xの基準ポイントをWに取得
	MOWF	FSR	変数 Xの基準アドレスを間接ポイントに設定
	MOVLW	4	変数のバイト数をWに取得
	MOWF	CNT	;バイトカウンタを初期化
	BCF	C	初回用にキャリー フラグをクリア
;			
LOOP	RLF	FLG,F	前桁からのキャリー フラグを保存
	MOVLW	Y-X	変数間 オフセットをWに取得
	ADDWF	FSR,F	変数 Yの対応バイトのアドレスを間接ポイントに設定
	MOVF	INDF,W	加算値をWに取得
	MOWF	TMP	加算値を一時保存
	MOVLW	Y-X	変数間 オフセットをWに取得
	SUBWF	FSR,F	変数 Xの対応バイトのアドレスを間接ポイントに設定
	MOVF	TMP,W	加算値をWに復帰
	BTFS	FLG,0	前桁からのキャリーなしでスキップ
	INCF	TMP,W	;キャリーで加算値+1をWに取得 =0でスキップ
	ADDWF	INDF,F	前桁からのキャリーなし加算値+1 =0で対応バイト加算
;			
	INCF	FSR,F	間接ポイントを進行 (+1)
	DECFSZ	CNT,F	;バイトカウンタを減数、=0でスキップ
	GOTO	LOOP	;0以外でバイト数分処理継続へ

基本的には非ループ処理の基本形と同じ方法ですが、ループ処理を実現するために直接アドレッシングを行わず、間接アドレッシングによって変数アクセスを行っています。

間接アドレッシング用のポイントが1つだけですので、変数間オフセットの加減算で変数のポイントを切り替えています。このため、サブルーチン化して複数変数間で使用する場合は、対象となる変数間オフセットが同一でなければなりません。異なる場合は8行目と12行目のMOVLW Y-XをMOVWF OFSWのように変数OFSから対応する変数間オフセットを取得するようにし、呼び出しに先立ってこの変数OFSを設定します。

次に上記ループ処理例と非ループ処理 (必要数分のべた書き)での命令語数と実行サイクル数の関係を示します。

命令語数と速度の関係

ビット数	バイト数	命令語数		実行サイクル数		備考
		非ループ	ループ	非ループ	ループ	
8	1	2	19	2	19	
16	2	6		6	34	
24	3	10		10	49	
32	4	14		14	64	
40	5	18		18	79	命令語数がほぼ同一
48	6	22		22	94	以降、ループ処理の方が命令語数が少ない
56	7	26		26	109	
64	8	30		30	124	

上記から40ビット(5バイト長を超える場合にループ処理での命令語数の方が少なくなることが判ります。けれども実行速度は著しく遅くなり、ほぼ同一命令語数となる40ビット長の場合で4倍以上の速度低下になります。

## バイト減算

中位PCIには8ビット(1バイト長の減算命令 **SUBWF**命令があります。8ビット(1バイト長のバイト減算は、この **SUBWF**命令で実現できます。次は1バイト長の  $X=X-Y$ の例です。

X	DS	1	;1バイト長変数 X定義
Y	DS	1	;1バイト長変数 Y定義
	MOVF	Y,W	変数 Y減算値 をWに取得
	SUBWF	X,F	変数 Xから変数 Yを減算

多倍長の場合は **SUBWF**命令の連続で処理しますが、直前のバイトからの borrowを含めて処理しなければなりません。次は2バイト長の  $X=X+Y$ の例です。

X	DS	2	;2バイト長変数 X定義
Y	DS	2	;2バイト長変数 Y定義
	MOVF	Y+0,W	減算値最下位 バイトをWに取得
	SUBWF	X+0,F	最下位 バイト減算
	;		
	MOVF	Y+1,W	減算値第2バイトをWに取得
	BTFSS	C	前桁からの borrowなしでスキップ
	INCFSSZ	Y+1,W	; borrowで減算値 +1をWに取得、=0でスキップ
	SUBWF	X+1,F	前桁からの borrowなしに減算値 +1、0で第2バイト減算

減算での重要な点は borrow フラグです。borrow フラグは物理位置的に carry フラグと同一ですが、論理が異なります。このフラグを borrow として扱う場合、負論理 (0=borrowあり、1=borrowなし)になります。

上例の要点は5行目の **BTFSS**と7行目の **INCFSSZ**で2種の条件分岐 (スキップ)を行っていることです。前桁からの borrowなし時には7行目の **SUBWF**での減算となり borrowあり時には減算値を+1することで前桁からの borrowを反映しています。

更に、この減算値 +1が0となる場合には7行目の **SUBWF**命令をスキップしています。これは、次桁用に borrow フラグ (Cフラグ=0)を保存するためです。減算値 +1が0となる場合には今桁で borrowが発生することを意味します。7行目の **SUBWF**命令を実行すると、今桁の結果は正しくありませんが、borrowなし (Cフラグ=1)になってしまいます。

この対処のため、7行目の **SUBWF**命令をスキップする必要があります。直前の **BTFSS**命令の結果、**INCFSSZ**命令が実行される場合の borrow フラグは常に (Cフラグ=0)です。

第2バイト目以降は上記第2バイト処理と同様な形式で記述します。次にビット(バイト)に対する命令語数と実行サイクル数の関係を示します。

命令語数と速度の関係

ビット数	バイト数	命令語数	実行サイクル数	備考
8	1	2	2	
16	2	6	6	
24	3	10	10	
32	4	14	14	
40	5	18	18	
48	6	22	22	
56	7	26	26	
64	8	30	30	

### ループ処理によるバイト減算

多倍長もビット数が多くなると命令語数的にループ処理が有利です。

X	DS	4	;4バイト長変数 X定義
Y	DS	4	;4バイト長変数 Y定義
CNT	DS	1	作業用 カウンタ定義
TMP	DS	1	加算値一時保存変数定義
FLG	DS	1	;キャリー フラグ一時保存変数定義
	MOVLW	X	変数 Xの基準 ポインタをWに取得
	MOVWF	FSR	変数 Xの基準アドレスを間接 ポインタに設定
	MOVLW	4	変数のバイト数をWに取得
	MOVWF	CNT	;バイトカウンタを初期化
	BSF	C	初回用にゼロ フラグをクリア
;			
LOOP	RLF	FLG,F	前桁からのゼロ フラグを保存
	MOVLW	Y-X	変数間 オフセットをWに取得
	ADDWF	FSR,F	変数 Yの対応バイトのアドレスを間接 ポインタに設定
	MOVF	INDF,W	減算値をWに取得
	MOVWF	TMP	減算値を一時保存
	MOVLW	Y-X	変数間 オフセットをWに取得
	SUBWF	FSR,F	変数 Xの対応バイトのアドレスを間接 ポインタに設定
	MOVF	TMP,W	減算値をWに復帰
	BTFSS	FLG,0	前桁からのゼロなしでスキップ
	INCF	TMP,W	;ゼロで減算値 +1をWに取得、=0でスキップ
	SUBWF	INDF,F	前桁からのゼロなしが減算値 +1 0で対応バイト減算
;			
	INCF	FSR,F	間接 ポインタを進行 (+1)
	DECFSZ	CNT,F	;バイトカウンタを減数、=0でスキップ
	GOTO	LOOP	;0以外でバイト数分処理継続へ

基本的には非ループ処理の基本形と同じ方法ですが、ループ処理を実現するために直接アドレッシングを行わず、間接アドレッシングによって変数アクセスを行います。

間接アドレッシング用のポインタが1つだけですので、変数間オフセットの加減算で変数のポインタを切り替えます。このため、サブルーチン化して複数変数間で使用する場合は、対象となる変数間オフセットが同じでなければなりません。異なる場合は8行目と12行目のMOVLW Y-XをMOVF OFS,Wのように変数OFSから対応する変数間オフセットを取得するようにし、呼び出しに先立ってこの変数OFSを設定します。

次に上記ループ処理例と非ループ処理(必要数分のべた書き)での命令語数と実行サイクル数の関係を示します。

命令語数と速度の関係

ビット数	バイト数	命令語数		実行サイクル数		備考
		非ループ	ループ	非ループ	ループ	
8	1	2	19	2	19	
16	2	6		6	34	
24	3	10		10	49	
32	4	14		14	64	
40	5	18		18	79	命令語数がほぼ同一
48	6	22		22	94	以降、ループ処理の方が命令語数が少ない
56	7	26		26	109	
64	8	30		30	124	

上記から40ビット(5バイト長を超える場合にループ処理での命令語数の方が少なくなることが判ります。けれども実行速度は著しく遅くなり、ほぼ同一命令語数となる40ビット長の場合で4倍以上の速度低下になります。

### 加算によるハイ乗算

中位PCには乗算命令がありませんので、加算命令 **ADDWF** を基に作成します。基本的な方法は通常の筆算による乗算と同一です。大きく異なるのは通常の10進数の場合、乗数の各桁計算時に非乗数×乗数の対応桁(桁)の乗算がありますが、2進数の場合の桁は0か1だけですので、0でそのまま、1で非乗数の加算になり、加算のみで行えることです。

下図は8ビット(バイト長)の  $X \times Y = Z$  の例です。結果のZは16ビット(2バイト長)になります。例では中間結果を保持する作業領域をXの上位に連結し、中間結果とXのビットシフトを同時に行っています。この手法により、作業領域が最小となり、最終結果の移動もなくなります。また、各桁判定が常に変数Xの最下位ビットとなり、その結果の桁単位の加算位置も固定になりますので、ループ処理に最適な方法です。

変数Xの上位は初期値として0を設定しておく必要があります。

The diagram illustrates the bit-by-bit multiplication of two 8-bit numbers, X and Y, to produce a 16-bit result Z. X is 00000001 and Y is 11000100. The process involves shifting X to the right by one bit for each bit of Y that is 1, and then adding the shifted X to the current result. The carry is propagated to the next higher bit. The final result Z is 01111010.

**Step 1:** Xの最下位ビット=0なので結果とXを1ビット右シフトし、最上位ビット=0とします。

**Step 2:** Xのビット1現在の最下位ビット=1なので、結果の対応桁とYを加算し、結果とXを1ビット右シフトします。最上位ビットは加算結果のキャリー(例では0)です。

**Step 3:** Xのビット2現在の最下位ビット=0なので、結果とXを1ビット右シフトし、最上位ビット=0とします。

**Step 4:** Xのビット3現在の最下位ビット=1なので、結果の対応桁とYを加算し、結果とXを1ビット右シフトします。最上位ビットは加算結果のキャリー(例では0)です。

**Step 5:** Xのビット4現在の最下位ビット=1なので、結果の対応桁とYを加算し、結果とXを1ビット右シフトします。最上位ビットは加算結果のキャリー(例では1)です。

**Step 6:** Xのビット5現在の最下位ビット=0なので、結果とXを1ビット右シフトし、最上位ビット=0とします。

**Step 7:** Xのビット6現在の最下位ビット=0なので、結果とXを1ビット右シフトし、最上位ビット=0とします。

**Step 8:** Xの最上位ビット現在の最下位ビット=0なので、結果とXを1ビット右シフトし、最上位ビット=0とします。

この方法は任意のハイ長乗算が可能です。次に任意のハイ長乗算作成時の要点を示します。

- 変数Xの上位に連続して同一長の領域を確保します。
- 上記領域は0で初期化します。
- 加算はハイ長分の多倍長加算で行います。
- 右シフトはハイ長分のRRF命令の連続で行います。

### バイト乗算

中位PICには乗算命令がありませんので、加算命令 **ADDWF** を基に作成します。次は 16ビット(2バイト長の  $X \times Y = X$  (32ビット) の例です。

```

X      DS      2+2      ;2+2バイト長変数 X定義
Y      DS      2        ;2バイト長変数 Y定義
CNT    DS      1        ;作業用ビットカウンタ

      CLRF     X+2+0     ;結果上位を初期化
      CLRF     X+2+1     ;
;
      MOVLW   16        ;ビット数をWに取得
      MOVWF   CNT       ;ビットカウンタを初期化
LOOP  BCF     C          ;対応桁キャリーなしを仮設定 (キャリーフラグをクリア)
      BTFS   X+0,0      ;変数 Xの対応ビット=1でスキップ
      GOTO   LP1       ;変数 Xの対応ビット=0で分岐
;
      MOVF    Y+0,W     ;乗数 Y最下位バイトをWに取得
      ADDWF   X+2+0,F   ;対応桁結果最下位バイト計算
;
      MOVF    Y+1,W     ;乗数 Y第2バイトをWに取得
      BTFS   C          ;前桁からのキャリーなしでスキップ
      INCF   Y+1,W     ;キャリーで乗数第2バイト値+1をWに取得、=0でスキップ
      ADDWF   X+2+1,F   ;前桁からのキャリーなしが第2バイト値+1、0で第2バイト計算
;
LP1   RRF     X+2+1,F   ;対応桁キャリー-結果元データX右シフト
      RRF     X+2+0,F   ;
      RRF     X+1,F     ;
      RRF     X+0,F     ;
;
      DECF   CNT,F     ;ビットカウンタ減数、=0でスキップ
      GOTO   LOOP      ;0で、ビット数分処理継続へ
    
```

3バイト長以上の場合は、1,2行目の結果上位初期化部と18~2行目の右シフト部への追加、4行目のビット数の変更、多倍長加算の追加を行います。多倍長加算は17行目からへ13~16行目の第2バイト加算と同様の形式で記述します。ラベル **LP** からの **RRF** 命令は必ず上位側から記述しなければなりません。

次にビット(バイト)長に対する命令語数と実行サイクル数の関係を示します。

命令語数と速度の関係

ビット数	バイト数	命令語数	実行サイクル数	備考
8	1	12	74~ 82	8x 8 = 16ビット
16	2	19	179~ 259	16x 16 = 32ビット
24	3	26	316~ 532	24x 24 = 48ビット
32	4	33	485~ 901	32x 32 = 64ビット
40	5	40	686~ 1366	40x 40 = 80ビット
48	6	47	919~ 1927	48x 48 = 96ビット
56	7	54	1184~ 2584	56x 56 = 112ビット
64	8	61	1481~ 3337	64x 64 = 128ビット

## 兼用型ハ`イ乗算

基本形を変形した兼用型ハ`イ乗算ルーチンの例を次に示します。この例では8,16,24,32ビット長の乗算が行えます。但し、各々の結果は8x 8がX+3から、16x 16がX+2から、24x 24がX+1から、32x 32がX+0からになります。

X	DS	4+4	;4+4ハ`イ長変数X定義	
Y	DS	4	;4ハ`イ長変数Y定義	
CNT	DS	1	;作業用ビットカウンタ	
MUL8	MDVLW GOTO	8 MUL	;ビット数 (=8をWに取得 実処理へ	[8x 8入口]
;				
MUL16	MDVLW GOTO	16 MUL	;ビット数 (=16をWに取得 実処理へ	[16x 16入口]
;				
MUL24	MDVLW GOTO	24 MUL	;ビット数 (=24をWに取得 実処理へ	[24x 24入口]
;				
MUL32	MDVLW	32	;ビット数 (=32をWに取得	[32x 32入口]
;				
MUL	CLRF CLRF CLRF CLRF	X+4+0 X+4+1 X+4+2 X+4+3	結果上位を初期化 ; ; ;	
;				
MUL1	MDWVF BCF BTFSS GOTO	CNT C X+0,0 MUL2	;ビットカウンタを初期化 対応桁キャリなしを仮設定 (キャリ フラグをクリア) 変数Xの対応ビット=1でスキップ 変数Xの対応ビット=0で分岐	
;				
	MOVF ADDWF	Y+0,W X+4+0,F	乗数Y最下位ハ`イをWに取得 対応桁結果最下位ハ`イ計算	
;				
	MOVF BTFSC INCFSZ ADDWF	Y+1,W C Y+1,W X+4+1,F	乗数Y第2ハ`イをWに取得 前桁からのキャリなしでスキップ ;キャリで乗数第2ハ`イ値 +1をWに取得、=0でスキップ 前桁からのキャリなしが第2ハ`イ値 +1、0で第2ハ`イ計算	
;				
	MOVF BTFSC INCFSZ ADDWF	Y+2,W C Y+2,W X+4+2,F	乗数Y第3ハ`イをWに取得 前桁からのキャリなしでスキップ ;キャリで乗数第3ハ`イ値 +1をWに取得、=0でスキップ 前桁からのキャリなしが第3ハ`イ値 +1、0で第3ハ`イ計算	
;				
	MOVF BTFSC INCFSZ ADDWF	Y+3,W C Y+3,W X+4+3,F	乗数Y第4ハ`イをWに取得 前桁からのキャリなしでスキップ ;キャリで乗数第4ハ`イ値 +1をWに取得、=0でスキップ 前桁からのキャリなしが第4ハ`イ値 +1、0で第4ハ`イ計算	
;				
MUL2	RRF RRF RRF RRF RRF RRF RRF RRF	X+4+3,F X+4+2,F X+4+1,F X+4+0,F X+3,F X+2,F X+1,F X+0,F	対応桁キャリ 結果元データ右シフト ; ; ; ; ; ; ;	
;				
	DECFSZ GOTO	CNT,F MUL1	;ビットカウンタ減数、=0でスキップ ; 0で、ビット数分処理継続へ	
;				
	RETURN		呼び出し元へ復帰	

### 減算によるハイト除算

中位PICには除算命令がありませんので減算命令 **SUBWF** を基に作成します。基本的な方法は通常の筆算による除算と同じです。大きく異なるのは通常の10進数の場合、除数の各桁計算時に非除数の対応桁÷除数の除算がありますが、2進数の場合の桁は **0**か**1** だけですので、**0**でそのまま、**1**で非除数の減算になり、減算のみで行えることです。

下図は8ビット(1バイト)長の  $X \div Y = X$  の例です。商は8ビット(1バイト)長で  $X+0$  に、剰余も8ビット(1バイト)長で  $X+1$  に格納されます。例では中間結果を保持する作業領域を  $X$  の上位に連結し、中間結果と  $X$  のビットシフトを同時に行っています。この手法によって作業領域が最小となり、最終結果の移動もなくなります。また、各桁での減算位置が常に固定となり、その結果の商ビットは常に変数  $X$  の最下位ビットになりますので、ループ処理に最適な方法です。

変数  $X$  の上位は初期値として **0** を設定しておく必要があります。

キャリーフラグは非除数ビット数 > 除数ビット数の場合のみ考慮しなければなりません。その他の場合のキャリーフラグは常に **0** となりますので、剰余 - 除数の対象にはなりません。

The diagram illustrates the bit-by-bit division of 15 (00001111) by 7 (00000111). The process starts with the dividend in the register and the divisor below it. The quotient register is initially 0. The algorithm proceeds by shifting the dividend left and subtracting the divisor if the current bit is 1. The steps are as follows:

- Step 1: Dividend is 00001111, divisor is 00000111. The 15th bit is 0, so the quotient bit is 0. The dividend is shifted left to 00011110.
- Step 2: Dividend is 00011110, divisor is 00000111. The 14th bit is 0, so the quotient bit is 0. The dividend is shifted left to 00111100.
- Step 3: Dividend is 00111100, divisor is 00000111. The 13th bit is 0, so the quotient bit is 0. The dividend is shifted left to 01111000.
- Step 4: Dividend is 01111000, divisor is 00000111. The 12th bit is 0, so the quotient bit is 0. The dividend is shifted left to 11110000.
- Step 5: Dividend is 11110000, divisor is 00000111. The 11th bit is 1, so the quotient bit is 1. The dividend is shifted left to 11100000, and the divisor is subtracted from it, resulting in 11000000.
- Step 6: Dividend is 11000000, divisor is 00000111. The 10th bit is 1, so the quotient bit is 1. The dividend is shifted left to 10000000, and the divisor is subtracted from it, resulting in 00000000.
- Step 7: Dividend is 00000000, divisor is 00000111. The 9th bit is 0, so the quotient bit is 0. The dividend is shifted left to 00000000.
- Step 8: Dividend is 00000000, divisor is 00000111. The 8th bit is 0, so the quotient bit is 0. The dividend is shifted left to 00000000.
- Step 9: Dividend is 00000000, divisor is 00000111. The 7th bit is 0, so the quotient bit is 0. The dividend is shifted left to 00000000.
- Step 10: Dividend is 00000000, divisor is 00000111. The 6th bit is 0, so the quotient bit is 0. The dividend is shifted left to 00000000.
- Step 11: Dividend is 00000000, divisor is 00000111. The 5th bit is 0, so the quotient bit is 0. The dividend is shifted left to 00000000.
- Step 12: Dividend is 00000000, divisor is 00000111. The 4th bit is 0, so the quotient bit is 0. The dividend is shifted left to 00000000.
- Step 13: Dividend is 00000000, divisor is 00000111. The 3rd bit is 0, so the quotient bit is 0. The dividend is shifted left to 00000000.
- Step 14: Dividend is 00000000, divisor is 00000111. The 2nd bit is 0, so the quotient bit is 0. The dividend is shifted left to 00000000.
- Step 15: Dividend is 00000000, divisor is 00000111. The 1st bit is 0, so the quotient bit is 0. The dividend is shifted left to 00000000.

The final quotient is 00000010 (2) and the remainder is 00000001 (1).

この方法は任意のバイト長除算が可能です。次に任意のバイト長除算作成時の要点を示します。

- 変数  $X$  の上位に連続して同一長の領域を確保します。
- 上記領域は **0** で初期化します。
- 減算はバイト長の多倍長減算で行います。
- 左シフトはバイト長の **RLLF** 命令の連続で行います。

## ハイ除算

中位PICには除算命令がありませんので、減算命令 **SUBWF** を基に作成します。次は 16ビット(2バイト長の  $X \div Y = X$  (16ビット) の例です。

X	DS	2+2	;2+2バイト長変数 X定義	
Y	DS	2	;2バイト長変数 Y定義	
CNT	DS	1	作業用ビットカウンタ	
	CLRF	X+2+0	非除数上位を初期化	
	CLRF	X+2+1	;	
			;	
	MOVLW	16	;ビット数をWに取得	
LOOP	MOVF	CNT	;ビットカウンタを初期化	
	BCF	C	商の対応ビット=0を仮設定	
	RLF	X+0,F	非除数 商を左シフト桁移動)	
	RLF	X+1,F	;	
	RLF	X+2+0,F	;	
	RLF	X+2+1,F	;	
			;	
	BTFSC	C	非除数対象最上位桁 =0でスキップ	(非除数ビット数 >
	GOTO	LP2	非除数対象最上位桁 =1で減算へ	除数ビット数時必要)
	MOVF	Y+1,W	除数第2バイトをWに取得	
	SUBWF	X+2+1,W	第2バイト減算可/不可検査	
	BTFSS	Z	同一値で下位桁検査へスキップ	
	GOTO	LP1	可/不可判定可で分岐	
	MOVF	Y+0,W	除数最下位バイトをWに取得	
LP1	SUBWF	X+2+0,W	最下位バイト減算可/不可検査	
	BTFSS	C	減算可でスキップ	
	GOTO	LP3	減算不可で分岐	
LP2	BSF	X+0,0	商の対応ビット=1を設定	
	MOVF	Y+0,W	除数最下位バイトをWに取得	
	SUBWF	X+2+0,F	最下位バイト減算	
	MOVF	Y+1,W	除数第2バイトをWに取得	
	BTFSS	C	前桁からのゼロなしでスキップ	
	INCFSZ	Y+1,W	;ゼロで減数値+1をWに取得,=0でスキップ	
	SUBWF	X+2+1,F	第2バイト減算	
LP3	DECFSZ	CNT,F	;ビットカウンタ減数,=0でスキップ	
	GOTO	LOOP	; 0で、ビット数分処理継続へ	

3バイト長以上の場合は1,2行目の非除数上位初期化部と7~10行目の左シフト部への追加、4行目のビット数の変更、多倍長比較と多倍長減算の追加を行います。多倍長比較は14行目から15~18行目の第2バイト比較と同様の形式で記述し、多倍長減算は33行目から29~32行目の第2バイト減算と同様の形式で記述します。7行目からの **RLF** 命令は必ず下位側から記述しなければなりません。

次にビット(バイト)長に対する命令語数と実行サイクル数の関係を示します。

命令語数と速度の関係

ビット数	バイト数	命令語数	実行サイクル数	備考
8	1	15	90~ 106	8÷ 8 = 8ビット 剰余 =8ビット)
16	2	26	259~ 371	16÷ 16 = 16ビット 剰余 =16ビット)
24	3	37	436~ 796	24÷ 24 = 24ビット 剰余 =24ビット)
32	4	48	645~ 1381	32÷ 32 = 32ビット 剰余 =32ビット)
40	5	59	886~ 2126	40÷ 40 = 40ビット 剰余 =40ビット)
48	6	60	1159~ 3031	48÷ 48 = 48ビット 剰余 =48ビット)
56	7	71	1464~ 4096	56÷ 56 = 56ビット 剰余 =56ビット)
64	8	82	1801~ 5321	64÷ 64 = 64ビット 剰余 =64ビット)

兼用型ハイレ除算

基本形を変形した兼用型ハイレ除算ルーチンの例を次に示します。この例では8~64ビット長の除算が行えます。結果は商がX+0から、剰余がX+8めらになります。但し、非除数Xの設定値は次表の位置から設定しなければなりません。

非除数Xビット長	8	16	24	32	40	48	56	64
非除数X設定位置	X+7	X+6	X+5	X+4	X+3	X+2	X+1	X+0
備考	8÷8	16÷8 16÷16	24÷16 24÷24	32÷16 32÷32	40÷24 40÷32	48÷24 48÷32	56÷32	64÷32

X	DS	8+4	;8+4ハイレ長変数X定義	
Y	DS	4	;4ハイレ長変数Y定義	
CNT	DS	1	;作業用ビットカウンタ	
DV6432	MDVLW GOTO	64 DIV32	;ビット数 (=64)をWに取得 実処理へ	[64÷ 32入口]
; ;				
DV5632	MDVLW GOTO	56 DIV32	;ビット数 (=56)をWに取得 実処理へ	[56÷ 32入口]
; ;				
DV4832	MDVLW GOTO	48 DIV32	;ビット数 (=48)をWに取得 実処理へ	[48÷ 32入口]
; ;				
DV4824	MDVLW GOTO	48 DIV24	;ビット数 (=48)をWに取得 実処理へ	[48÷ 24入口]
; ;				
DV4032	MDVLW GOTO	40 DIV32	;ビット数 (=40)をWに取得 実処理へ	[40÷ 32入口]
; ;				
DV4024	MDVLW GOTO	40 DIV24	;ビット数 (=40)をWに取得 実処理へ	[40÷ 24入口]
; ;				
DV3232	MDVLW GOTO	32 DIV32	;ビット数 (=32)をWに取得 実処理へ	[32÷ 32入口]
; ;				
DV3216	MDVLW GOTO	32 DIV16	;ビット数 (=32)をWに取得 実処理へ	[32÷ 16入口]
; ;				
DV2424	MDVLW GOTO	24 DIV24	;ビット数 (=24)をWに取得 実処理へ	[24÷ 24入口]
; ;				
DV2416	MDVLW GOTO	24 DIV16	;ビット数 (=24)をWに取得 実処理へ	[24÷ 16入口]
; ;				
DV1616	MDVLW GOTO	16 DIV16	;ビット数 (=16)をWに取得 実処理へ	[16÷ 16入口]
; ;				
DV1608	MDVLW GOTO	16 DIV8	;ビット数 (=16)をWに取得 実処理へ	[16÷ 8入口]
; ;				
DV0808	MDVLW	8	;ビット数 (=8)をWに取得	[8÷ 8入口]
; ;				
DIV8	CLRF	Y+1	除数上位を初期化	(8ビット除数移行点)
DIV16	CLRF	Y+2	; ;	(16ビット除数移行点)
DIV24	CLRF	Y+3	; ;	(24ビット除数移行点)
DIV32	CLRF	X+8+0	非除数 (仮想) 止位を初期化	(32ビット除数移行点)
; ;	CLRF	X+8+1	; ;	
; ;	CLRF	X+8+2	; ;	
; ;	CLRF	X+8+3	; ;	
; ;	MDWF	CNT	;ビットカウンタを初期化	
; ;				

次頁へ続く

;			
DIV1	BCF	C	;商の対応ビット=0を仮設定
	RLF	X+0,F	;非除数 商を左シフト桁移動)
	RLF	X+1,F	;
	RLF	X+2,F	;
	RLF	X+3,F	;
	RLF	X+4,F	;
	RLF	X+5,F	;
	RLF	X+6,F	;
	RLF	X+7,F	;
	RLF	X+8+0,F	;各桁減算 剰余 部左シフト桁移動)
	RLF	X+8+1,F	;
	RLF	X+8+2,F	;
	RLF	X+8+3,F	;
;			
	BTFSC	C	;非除数対象最上位桁=0でスキップ°
	GOTO	DIV3	;非除数対象最上位桁=1で減算へ
;			
	MOVF	Y+3,W	;除数第4バイトをWに取得
	SUBAF	X+8+3,W	;第4バイト減算可 /不可検査
	BTFSS	Z	;同一値で下位桁検査へスキップ°
	GOTO	DIV2	;可 /不可判定可で分岐
;			
	MOVF	Y+2,W	;除数第3バイトをWに取得
	SUBAF	X+8+2,W	;第3バイト減算可 /不可検査
	BTFSS	Z	;同一値で下位桁検査へスキップ°
	GOTO	DIV2	;可 /不可判定可で分岐
;			
	MOVF	Y+1,W	;除数第2バイトをWに取得
	SUBAF	X+8+1,W	;第2バイト減算可 /不可検査
	BTFSS	Z	;同一値で下位桁検査へスキップ°
	GOTO	DIV2	;可 /不可判定可で分岐
;			
	MOVF	Y+0,W	;除数最下位バイトをWに取得
	SUBAF	X+8+0,W	;最下位バイト減算可 /不可検査
DIV2	BTFSS	C	;減算可でスキップ°
	GOTO	DIV4	;減算不可で分岐
;			
DIV3	BSF	X+0,0	;商の対応ビット=1を設定
	MOVF	Y+0,W	;除数第1バイトをWに取得
	SUBAF	X+8+0,F	;第1バイト減算
;			
	MOVF	Y+1,W	;除数第2バイトをWに取得
	BTFSS	C	;前桁からのゼロなしでスキップ°
	INCFSZ	Y+1,W	;ゼロで減数値+1をWに取得 =0でスキップ°
	SUBAF	X+8+1,F	;前桁からのゼロなしが減算値+1 0で第2バイト減算
;			
	MOVF	Y+2,W	;除数第3バイトをWに取得
	BTFSS	C	;前桁からのゼロなしでスキップ°
	INCFSZ	Y+2,W	;ゼロで減数値+1をWに取得 =0でスキップ°
	SUBAF	X+8+2,F	;前桁からのゼロなしが減算値+1 0で第3バイト減算
;			
	MOVF	Y+3,W	;除数第4バイトをWに取得
	BTFSS	C	;前桁からのゼロなしでスキップ°
	INCFSZ	Y+3,W	;ゼロで減数値+1をWに取得 =0でスキップ°
	SUBAF	X+8+3,F	;前桁からのゼロなしが減算値+1 0で第4バイト減算
;			
DIV4	DECFSZ	ONT,F	;ビットカウン減数 =0でスキップ°
	GOTO	DIV1	; 0で、ビット数分処理継続へ
;			
	RETURN		;呼び出し元へ復帰

### ハイレリ ハッグ化 BCD変換

8ビット長ハイレリ ハッグ化 BCD変換の例を以下に示します。

X	DS	1	;ハイレリ値 元値 変数定義
Y	DS	2	;ハッグ化 BCD値 変換値 変数定義
CNT1	DS	1	;ハイレリ用ビットカウン定義
CNT2	DS	1	;BCD用バイトカウン定義
B2BCD	CLRF	Y+0	結果 BCD変数を初期化
	CLRF	Y+1	;
	MOVLW	8	;ビットカウン値を設定
	MOWF	CNT1	;
	GOTO	B2D3	初回開始へ
;			
B2D1	MOVLW	Y+0	;BCD最下位バイトポインタに取得
	MOWF	FSR	;BCD基準アドレスを間接ポインタに設定
	MOVLW	3	;BCDバイト数をWに取得
	MOWF	CNT2	;BCDバイトカウン値を設定
B2D2	MOVLW	8-10/2	;下桁で8次回 10以上となる値 (5)を取得
	ADDWF	INDF, F	;下桁次回 10か検査 次桁 LSB, 3ビット補整)
	BTFSZ	INDF, 3	;下桁次回 10以上でスキップ
	SUBWF	INDF, F	次回 9以下で元値復帰
;			
	MOVLW	(8-10/2)*16	上桁で8次回 10以上となる値 (5)を取得
	ADDWF	INDF, F	上桁次回 10か検査 次桁 LSB, 3ビット補整)
	BTFSZ	INDF, 7	上桁次回 10以上でスキップ
	SUBWF	INDF, F	次回 9以下時、元値復帰
;			
	INCF	FSR, F	間接ポインタを進行
	DECFSZ	CNT2, F	;バイトカウン値減数 =0でスキップ
	GOTO	B2D2	;BCDバイト数分継続
;			
	RLF	X+0, F	;ハイレリ止位ビットから処理
	RLF	Y+0, F	;ハイレリ止位ビットをBCD最下位ビットへ
	RLF	Y+1, F	;
	DECFSZ	CNT1, F	;ビットカウン値減数 =0でスキップ
	GOTO	B2D1	;ビット数分継続

上記例は変数定義と以下の点を変更することで、8ビット長以外でも使用できます。

- 結果 BCD値のバイト数分を初期化します。
- ビットカウン値はハイレリ値のビット数を設定します。
- BCDバイト数は結果となるハッグ化 BCDのバイト数を設定します。
- ハイレリ値のバイト数分、下位バイトから RLF 命令を記述します。
- 結果 BCD値バイト数分、下位バイトから RLF 命令を記述します。

ハイレリ値のビット長とBCD桁数、ハッグ化 BCDバイト数の関係を次に示します。

ハイレリビット長	8	16	24	32	40	48	56	64
BCD桁数	3	5	8	10	13	15	17	20
ハッグ化 BCDバイト数	2	3	4	5	7	8	9	10

#### 命令語数と速度の関係

ビット数	バイト数	命令語数	実行サイクル数	備考
8	1	25	240	0 ~ 255
16	2	28	717	0 ~ 65535
24	3	31	1418	0 ~ 16777215
32	4	34	2343	0 ~ 4294967295
40	5	39	4001	0 ~ 1099511627775
48	6	42	5478	0 ~ 281474976710655
56	7	45	7179	0 ~ 72057594037927935
64	8	48	9104	0 ~ 18446744073709551615

### 兼用型ハ`イリ ハ`ッ化BCD変換

基本形を变形した兼用型ハ`イリ ハ`ッ化BCD変換ルーチンの例を次に示します。基本形に対し、既に終了(シフトした無効なハ`イリ値分の処理をハ`イリ単位で行わないこと)で高速化を計っています。この手法で基本形に対して平均で2倍程度速くなりますが、命令語数が増大します。また、ハ`イリ値によって実行速度が変化し、基本形に対して約1.5~3倍程度の速度になります。

本例では8~64ビット長のハ`イリ ハ`ッ化BCD変換が行えます。結果のハ`ッ化BCD値はY+0からになります。元値のハ`イリ値は次表の位置から設定しなければなりません。

ハ`イリ値ビット長	8	16	24	32	40	48	56	64
ハ`イリ値X設定位置	X+7	X+6	X+5	X+4	X+3	X+2	X+1	X+0
BCD桁数	3	5	8	10	13	15	17	20
ハ`ッ化BCDハ`イ数	2	3	4	5	7	8	9	10

```

X      DS      8      ;ハ`イリ値 元値 変数定義
Y      DS      10     ;ハ`ッ化BCD値 変換値 変数定義
ONT1   DS      1      ;ハ`イリ用ビットカウン定義
ONT2   DS      1      ;BCD用ハ`イリカウン定義
TMP    DS      1      処理BCDハ`イ数(高速化用)

B2D8   MOVLW   8      ;ハ`イリビット数 (=8をWに取得      [8ビットハ`イリ入口 ]
      GOTO    BD8     実処理へ
;
B2D16  MOVLW   16     ;ハ`イリビット数 (=16をWに取得     [16ビットハ`イリ入口 ]
      GOTO    BD16    実処理へ
;
B2D24  MOVLW   24     ;ハ`イリビット数 (=24をWに取得     [24ビットハ`イリ入口 ]
      GOTO    BD24    実処理へ
;
B2D32  MOVLW   32     ;ハ`イリビット数 (=32をWに取得     [32ビットハ`イリ入口 ]
      GOTO    BD32    実処理へ
;
B2D40  MOVLW   40     ;ハ`イリビット数 (=40をWに取得     [40ビットハ`イリ入口 ]
      GOTO    BD40    実処理へ
;
B2D48  MOVLW   48     ;ハ`イリビット数 (=48をWに取得     [48ビットハ`イリ入口 ]
      GOTO    BD48    実処理へ
;
B2D56  MOVLW   56     ;ハ`イリビット数 (=56をWに取得     [56ビットハ`イリ入口 ]
      GOTO    BD56    実処理へ
;
B2D64  MOVLW   64     ;ハ`イリビット数 (=64をWに取得     [64ビットハ`イリ入口 ]
;
BD64   CLRF    Y+9    結果BCD変数を初期化
BD56   CLRF    Y+8    ;
BD48   CLRF    Y+7    ;
BD40   CLRF    Y+6    ;
      CLRF    Y+5    ;
BD32   CLRF    Y+4    ;
BD24   CLRF    Y+3    ;
BD16   CLRF    Y+2    ;
BD8    CLRF    Y+1    ;
      CLRF    Y+0    ;
      MOVWF   ONT1    ;ビットカウン設定
      MOVLW   1        ;Wに1を取得
      MOVWF   TMP     処理BCDハ`イ数を初期化
      GOTO    B2D5    初回開始へ
;

```

次頁へ続く

;			
B2D1	MOVLW	Y+0	;BCD最下位バイトポイントを取得
	MOXWF	FSR	;BCD最下位バイトアドレスを間接ポイント外に設定
	MOVF	TMP,W	処理 BCDバイト数をWに取得
	MOXWF	CNT2	;BCDバイトカウンタを設定
B2D2	MOVLW	8-10/2	;下桁で8次回 10以上となる値 (5)を取得
	ADDWF	INDF,F	;下桁次回 10か検査 次桁 LSB,3ビット補整)
	BTFSS	INDF,3	;下桁次回 10以上でスキップ
	SUBWF	INDF,F	次回 9以下で元値復帰
;			
	MOVLW	(8-10/2)*16	上桁で8次回 10以上となる値 (5)を取得
	ADDWF	INDF,F	上桁次回 10か検査 次桁 LSB,3ビット補整)
	BTFSS	INDF,7	上桁次回 10以上でスキップ
	SUBWF	INDF,F	次回 9以下で元値復帰
	BTFSS	INDF,7	上桁次回 10以上でスキップ
	GOTO	B2D3	次回 9以下で分岐
;			
	DECF	CNT2,W	最下位 BCDバイトか検査
	BTFSC	Z	最下位バイト以外でスキップ
	INCF	TMP,F	最下位バイトで処理 BCDバイト数を増加 (+1)
	GOTO	B2D4	次へ
B2D3	SUBWF	INDF,F	次回 9以下時、元値復帰
;			
B2D4	INCF	FSR,F	間接ポイントを進捗
	DECFSZ	CNT2,F	;バイトカウンタを減数、=0でスキップ
	GOTO	B2D2	;BCDバイト数分継続
;			
B2D5	RLF	X+0,F	;バイト止位ビットから処理
	RLF	X+1,F	;
	RLF	X+2,F	;
	RLF	X+3,F	;
	RLF	X+4,F	;
	RLF	X+5,F	;
	RLF	X+6,F	;
	RLF	X+7,F	;
	RLF	Y+0,F	;バイト止位ビットをBCD最下位ビットへ
	RLF	Y+1,F	;
	RLF	Y+2,F	;
	RLF	Y+3,F	;
	RLF	Y+4,F	;
	RLF	Y+5,F	;
	RLF	Y+6,F	;
	RLF	Y+7,F	;
	RLF	Y+8,F	;
	RLF	Y+9,F	;
	DECFSZ	CNT1,F	;ビットカウンタを減数、=0でスキップ
	GOTO	B2D1	;ビット数分継続へ
;			
	RETURN		呼出元へ復帰

## 16ビット BCD 16ビット変換

16ビット BCDを16ビットに変換するにはBCDの各桁の16ビット値を加算します。BCDの各桁を下位桁からd1,d2,...とする  
と、i桁のBCD値は次式で表せます。

$$d1 + d2 \times 10^1 + d3 \times 10^2 + \dots + d_n \times 10^{n-1}$$

この式を変形すると次式になります。

$$d1 + (d2 + (d3 + \dots + d_n \times 10 \dots) \times 10) \times 10$$

この式からBCDの各桁毎に10倍と加算を行うことで変換できることが判ります。

次に4桁16ビット BCD 16ビット長16ビット変換の例を示します。

X	DS	2+2	;16ビット値 変換値 )一時保存変数定義
Y	DS	2	;16ビット BCD値 元値 変数定義
D2B	CLRF	X+1	結果16ビット変数を初期化
			;
	SWAPF	Y+1,W	;BCD最上位桁値をWに取得
	ANDLW	0FH	;下位桁のみ有効
	MOVF	X+0	;BCD最上位桁値設定
	CALL	M10	;BCD最上位桁値を10倍
	MOVF	Y+1,W	;BCD第3桁値をWに取得
	CALL	C1D	;BCD第3桁値を積算後10倍
	SWAPF	Y,W	;BCD第2桁値をWに取得
	CALL	C1D	;BCD第2桁値を積算後10倍
	MOVF	Y,W	最下位桁値をWに取得
			;
		(BCD桁加算)	
A1D	ANDLW	0FH	;下位桁のみ有効
	ADDAF	X+0,F	最下位16ビットに加算
	MOVLW	1	;Wに1を設定
	BTFS	C	;キャリーなしでスキップ
	ADDAF	X+1,F	;キャリーで第216ビット+1
	RETURN		呼出元へ復帰
			;
		(BCD桁を加算後10倍)	
C1D	CALL	A1D	;BCD桁加算
			;
M10	BCF	C	;キャリーフラグを解除
	RLF	X+0,F	最下位16ビット×2
	RLF	X+1,F	第216ビット×2
			;
	MOVF	X+0,W	最下位16ビット×2値をWに取得
	MOVF	X+2+0	最下位16ビット×2値を一時保存
	MOVF	X+1,W	第216ビット×2値をWに取得
	MOVF	X+2+1	第216ビット×2値を一時保存
			;
	BCF	C	;キャリーフラグを解除
	RLF	X+0,F	最下位16ビット×2(×4)
	RLF	X+1,F	第216ビット×2(×4)
			;
	BCF	C	;キャリーフラグを解除
	RLF	X+0,F	最下位16ビット×2(×8)
	RLF	X+1,F	第216ビット×2(×8)
			;
	MOVF	X+2+0,W	加算値最下位16ビットをWに取得
	ADDAF	X+0,F	最下位16ビット加算
			;
	MOVF	X+2+1,W	加算値第216ビットをWに取得
	BTFS	C	前桁からのキャリーなしでスキップ
	INCF	X+2+1,W	;キャリーで加数第216ビット+1をWに取得、=0でスキップ
	ADDAF	X+1,F	前桁0第216ビット+1のキャリーなしで第216ビット加算
			;
	RETURN		呼出元へ復帰

前記の例は、変数と以下の点を変更することで、BCD桁以外の桁数にも使用できます。

結果ハッキリ値ハイ数 - 份、初期化します。

同様の形式で必要なBCD桁数分を記述します。

この2行と同様な形式で、結果ハッキリ値のハイ数 - 份、加算処理を記述します。

必要なハッキリ値のハイ数分、下位ハイからRLF命令を記述します。

結果ハッキリ値ハイ数分の一時保存処理を記述します。

同様な形式で、結果ハッキリ値ハイ数 - 份、加算処理を記述します。

ハッキリ値のビット長とBCD桁数、ハッキリ化BCDハイ数の関係を次に示します。

BCD桁数	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
ハッキリBCDハイ数	1	2	3	4	5	6	7	8	9	10								
ハッキリビット長	7	10	14	17	20	24	27	30	34	37	40	44	47	50	54	57	60	64
ハッキリハイ数	1	2	3	4	5	6	7	8										

命令語数と速度の関係

BCD桁数	ハッキリハイ数	命令語数	実行サイクル数	備考
2	1	19	23	0 ~ 99
3	2	37	68	0 ~ 999
4	2	41	101	0 ~ 9999
5	3	57	179	0 ~ 99999
6	3	61	223	0 ~ 999999
7	3	65	267	0 ~ 9999999
8	4	81	389	0 ~ 99999999
9	4	85	444	0 ~ 999999999
10	5	101	599	0 ~ 9999999999
11	5	105	665	0 ~ 99999999999
12	5	109	731	0 ~ 999999999999
13	6	121	930	0 ~ 9999999999999
14	6	125	1007	0 ~ 99999999999999
15	7	141	1239	0 ~ 999999999999999
16	7	145	1327	0 ~ 9999999999999999
17	8	161	1592	0 ~ 99999999999999999
18	8	165	1691	0 ~ 999999999999999999
19	8	169	1790	0 ~ 9999999999999999999

## 兼用型ハック化BCD ハイリBCD変換

基本形を变形した兼用型ハック化BCD ハイリ変換ルーチンの例を次に示します。本例では2~10桁(1~4バイト)のハック化BCD ハイリ変換が行えます。但し、結果のハイリ値を最大4バイトとして扱っているため、10桁のハック化BCDの場合の最大値は4294967295に制限されます。結果のハイリ値はX+0からになります。元値のハック化BCD値はY+0が最下位バイトです。

BCD桁数	2	3	4	5	6	7	8	9	10
ハック化BCDバイト数	1	2	2	3	3	4	4	5	5
ハイリ値バイト数(ビット長)	1 (7)	2 (10)	2 (14)	3 (17)	3 (20)	3 (24)	4 (27)	4 (30)	5 (34)

X	DS	4+4	;ハイリ値 変換値 変数 /一時保存変数定義	
Y	DS	5	;ハック化BCD値 元値 変数定義	
QNT	DS	1	;ハック化BCD桁数 カウン定義	
D2B2	MOVLW GOTO	2-1 DB1	;BCD桁数 (=2)-1をWに取得 ;ハイリ1バイト変換処理へ	[2桁ハック化BCD入口]
;				
D2B3	MOVLW GOTO	3-1 DB2	;BCD桁数 (=3)-1をWに取得 ;ハイリ2バイト変換処理へ	[3桁ハック化BCD入口]
;				
D2B4	MOVLW GOTO	4-1 DB2	;BCD桁数 (=4)-1をWに取得 ;ハイリ2バイト変換処理へ	[4桁ハック化BCD入口]
;				
D2B5	MOVLW GOTO	5-1 DB3	;BCD桁数 (=5)-1をWに取得 ;ハイリ3バイト変換処理へ	[5桁ハック化BCD入口]
;				
D2B6	MOVLW GOTO	6-1 DB3	;BCD桁数 (=6)-1をWに取得 ;ハイリ3バイト変換処理へ	[6桁ハック化BCD入口]
;				
D2B7	MOVLW GOTO	7-1 DB3	;BCD桁数 (=7)-1をWに取得 ;ハイリ3バイト変換処理へ	[7桁ハック化BCD入口]
;				
D2B8	MOVLW GOTO	8-1 DB4	;BCD桁数 (=8)-1をWに取得 ;ハイリ4バイト変換処理へ	[8桁ハック化BCD入口]
;				
D2B9	MOVLW GOTO	9-1 DB4	;BCD桁数 (=9)-1をWに取得 ;ハイリ4バイト変換処理へ	[9桁ハック化BCD入口]
;				
D2B10	MOVLW	10-1	;BCD桁数 (=10)-1をWに取得	[10桁ハック化BCD入口]
;				
DB4	CLRF	ANKX+3	結果ハイリ変数を初期化	
DB3	CLRF	ANKX+2	;	
DB2	CLRF	ANKX+1	;	
DB1	CLRF	ANKX+0	;	
	MOVWF	QNT	;BCD桁 カウン設定	
;				
D2B1	RRF	QNT,W	;ハック化BCD値バイトオフセットをWに取得	
;				
	ADDLW	Y	;ハック化BCD対応ハイリ位置を取得	
	MOVWF	FSR	;ハック化BCD対応ハイリ用間接ポインタ設定	
	MOVF	INDF,W	対応ハイリ下位桁をWに仮取得	
	BTFS	QNT,0	;下位桁 (=0)でスキップ	
	SWAPF	INDF,W	上位桁で対応ハイリ上位桁をW下位に取得	

次頁へ続く

;	CALL	A1D	;BCD桁加算
	CALL	M2B	現在のハ`付け値を2倍
	MOVF	X+0,W	最下位ハ`イトx 2値をWに取得
	MOXWF	X+4	最下位ハ`イトx 2値を一時保存
	MOVF	X+1,W	第2ハ`イトx 2値をWに取得
	MOXWF	X+5	第2ハ`イトx 2値を一時保存
	MOVF	X+2,W	第3ハ`イトx 2値をWに取得
	MOXWF	X+6	第3ハ`イトx 2値を一時保存
	MOVF	X+3,W	第4ハ`イトx 2値をWに取得
	MOXWF	X+7	第4ハ`イトx 2値を一時保存
	CALL	M2B	現在のハ`付け値を2倍 (x 4)
	CALL	M2B	現在のハ`付け値を2倍 (x 8)
	MOVF	X+4,W	加算値最下位ハ`イトをWに取得
	ADDAF	X+0,F	最下位ハ`イト加算
;	MOVF	X+5,W	加算値第2ハ`イトをWに取得
	BTFSC	C	前桁からのキャリなしでスキップ°
	INCFSZ	X+5,W	;キャリで加数第2ハ`イト+1をWに取得 ,=0でスキップ°
	ADDAF	X+1,F	前桁°第2ハ`イト+1のキャリなしで第2ハ`イト加算
;	MOVF	X+6,W	加算値第3ハ`イトをWに取得
	BTFSC	C	前桁からのキャリなしでスキップ°
	INCFSZ	X+6,W	;キャリで加数第3ハ`イト+1をWに取得 ,=0でスキップ°
	ADDAF	X+2,F	前桁°第3ハ`イト+1のキャリなしで第3ハ`イト加算
;	MOVF	X+7,W	加算値第4ハ`イトをWに取得
	BTFSC	C	前桁からのキャリなしでスキップ°
	INCFSZ	X+7,W	;キャリで加数第4ハ`イト+1をWに取得 ,=0でスキップ°
	ADDAF	X+3,F	前桁°第4ハ`イト+1のキャリなしで第4ハ`イト加算
	DECFSZ	CNT,F	;BCD桁 カウンタを減数 ,=0でスキップ°
	GOTO	D2B1	;BCD桁数 -1份継続
;	MOVF	Y+0,W	;BCD最下位ハ`イトをWに取得
;	(BCD桁加算 )		
A1D	ANDLW	0FH	;下位桁 (BCD)の桁のみ有効
	ADDAF	X+0,F	結果ハ`付けの最下位ハ`イトに加算
	MOVLW	1	;Wに 1を設定
	BTFSC	C	;キャリなしでスキップ°
	ADDAF	X+1,F	;キャリありで第2ハ`イトに+1
	BTFSC	C	;キャリなしでスキップ°
	ADDAF	X+2,F	;キャリありで第3ハ`イトに+1
	BTFSC	C	;キャリなしでスキップ°
	ADDAF	X+3,F	;キャリありで第4ハ`イトに+1
	RETURN		呼出元へ復帰
;	(ハ`付け値2倍 )		
M2B	BCF	C	;キャリ フラグを解除
	RLF	X+0,F	結果ハ`付けの最下位ハ`イトx 2
	RLF	X+1,F	第2ハ`イトx 2
	RLF	X+2,F	第3ハ`イトx 2
	RLF	X+3,F	第4ハ`イトx 2
	RETURN		呼出元へ復帰

## ハック化 BCD加算

中位 PICにはハック化 BCD加算命令がありませんので8ビット(1バイト長のハック加算命令 **ADDWF**命令と桁キャリーフラグを利用して作成します。1バイト長のハック化 BCD変数 XとWレジスタをハック化 BCD加算する例  $(X+W) \Rightarrow X$  を以下に示します。

X	DS	1	;1バイト長ハック化 BCD変数 X定義
FLG	DS	1	;ハック化 BCDキャリーフラグ変数定義
PADD	ADDWF	X,F	;ハック加算
	MOVF	STATUS,W	;フラグをWに取得
	MOVF	FLG	;フラグを保存
			;
	MOVLW	16-10	加算で桁キャリーとなる値をWに取得
	ADDWF	X,F	;下位桁 9以下か検査 (下位桁補整)
	BTFSC	C	;下位桁補整でのハイトキャリーなしでスキップ
	BSF	FLG,0	;下位桁補整でのハイトキャリーで結果キャリー=1
			;
	BTFSC	FLG,1	;以前のハック加算での桁キャリーなしでスキップ
	GOTO	PADD1	;ハック加算での桁キャリーで分岐
			;
	BTFSS	DC	;下位桁 10以上でスキップ
	SUBWF	X,F	;下位桁 9以下で元値復帰
			;
PADD1	MOVLW	(16-10)*16	加算でキャリーとなる値をWに取得
	ADDWF	X,F	;上位桁 9以下か検査
	BTFSC	C	;上位桁 9以下でスキップ
	BSF	FLG,0	;上位桁 10以上で結果キャリーフラグを設定
			;
	BTFSS	FLG,0	;最終的な上位桁からのキャリーありでスキップ
	SUBWF	X,F	;キャリーなしで元値復帰
			;

上記形式の連続で多倍長のハック化 BCD加算が行えますが、命令語数的に不利です。通常、多倍長のハック化 BCD加算はループ処理で記述します。ループ処理形式では任意のバイト数のハック化 BCDが扱え、実行速度は概ねバイト数に比例します。

ルーフ処理によるハック化BCD加算

ハック化BCD演算は桁当りの命令数が多いので、ルーフ処理による演算が一般的です。次の例は任意のハック化BCD加算ルーチンです。このルーチンの呼び出し時にはWレジスタにハック化BCD変数を設定する必要があります。

X	DS	1~	;ハック化BCD変数X定義
Y	DS	1~	;ハック化BCD変数Y定義
CNT	DS	1	;バイトカウンタ定義
TMP			;一時保存データ(FLGと領域を共有)
FLG	DS	1	;BCD最終桁からのキャリーフラグ変数定義
PADD	MOVF	CNT	;バイトカウンタを設定
	MOVLW	X	変数X基準ポイントをWに取得
	MOVF	FSR	変数X基準位置を間接ポイントに設定
	BCF	FLG,0	結果キャリーフラグを初期化
PADD1	MOVLW	Y-X	変数YオフセットをWに取得
	ADDWF	FSR,F	対応変数Y位置を間接ポイントに設定
	MOVF	INDF,W	加算値をWに取得
	BTFSC	FLG,0	前回桁からのキャリーなしでスキップ
	ADDLW	1	;キャリーありでBCD下位桁+1
	MOVF	TMP	加算値を一時保存
	MOVLW	Y-X	対応変数XオフセットをWに取得
	SUBWF	FSR,F	対応変数X位置を間接ポイントに設定
	MOVF	TMP,W	加算値をWに復帰
	ADDWF	INDF,F	;ハック化加算
	MOVF	STATUS,W	;フラグをWに取得
	MOVF	FLG	;フラグを保存
	MOVLW	16-10	加算で桁キャリーとなる値をWに取得
	ADDWF	INDF,F	;下位桁9以下か検査(下位桁補整)
	BTFSC	C	;下位桁補整でのバイトキャリー無しでスキップ
	BSF	FLG,0	;下位桁補整でのバイトキャリーで結果キャリー=1
	BTFSC	FLG,1	;以前のハック化加算での桁キャリー検査
	GOTO	PADD2	;ハック化加算での桁キャリーで分岐
	BTFSS	DC	;下位桁10以上9でスキップ
	SUBWF	INDF,F	;下位桁9以下で元値復帰
PADD2	MOVLW	(16-10)*16	加算でキャリーとなる値をWに取得
	ADDWF	INDF,F	上位桁9以下か検査
	BTFSC	C	上位桁9以下でスキップ
	BSF	FLG,0	上位桁10以上で結果キャリーフラグを設定
	BTFSS	FLG,0	最終的な上位桁からのキャリーありでスキップ
	SUBWF	INDF,F	;キャリーなしで元値復帰
	INCF	FSR,F	間接ポイントを進捗
	DECFSZ	CNT,F	;バイトカウンタを減数=0でスキップ
	GOTO	PADD1	;ハック化分岐継続
	RETURN		呼出元へ復帰

変数定義は必要とする最大ハック化BCD変数を定義します。

次にハック化BCDの各ハック化BCD変数に対する実行サイクル数の関係を示します。

ハック化BCD長と速度の関係

BCD桁数	BCDハック化変数	命令語数	実行サイクル数	BCD桁数	BCDハック化変数	命令語数	実行サイクル数
1,2	1	35	34 ~ 35	11,12	6	35	179 ~ 185
3,4	2		63 ~ 65	13,14	7		208 ~ 215
5,6	3		92 ~ 95	15,16	8		237 ~ 245
7,8	4		121 ~ 125	17,18	9		266 ~ 275
9,10	5		150 ~ 155	19,20	10		295 ~ 305

## ハック化 BCD減算

中位 PICにはハック化 BCD減算命令がありませんので、8ビット(1バイト長のハイナリ減算命令 **SUBWF** と桁ホロ (キャリー) フラグを利用して作成します。1バイト長のハック化 BCD変数 XからWレジスタをハック化 BCD減算し、結果を変数 Xに格納する例  $X-W=X$  を以下に示します。

X	DS	1	;1バイト長ハック化BCD変数X定義
FLG	DS	1	;ハック化BCDホロ フラグ変数定義
PSUB	SUBAF	X,F	;ハイナリ減算
	BSF	FLG,0	結果ホロ フラグを初期化
	RLF	FLG,F	;ホロ フラグを保存
			;
	MOVF	X,W	;ハイナリ減算結果をWに取得
	BTFSC	DC	;ハイナリ減算の桁ホロでスキップ
	SUBLW	9	;下位桁9以下か検査
			;
	MOVLW	16-10	;下位桁補整値 (=6)をWに取得
	BTFSS	DC	;9以下 桁ホロ無しでスキップ
	SUBAF	X,F	;下位桁9以下に補整
			;
	RRF	FLG,F	;ハイナリ減算でのホロ フラグを復帰
	MOVF	X,W	;下位桁補整後データをWに取得
	BTFSC	C	;ハイナリ減算でのホロでスキップ
	SUBLW	9*16	上位桁9以下か検査
	BTFSC	C	;10以上ホロでスキップ
	GOTO	PSUB1	;9以下で分岐
			;
	MOVLW	(16-10)*16	上位桁補整値 (=60H)をWに取得
	SUBAF	X,F	上位桁9以下に補整
	BCF	FLG,0	結果ホロ フラグを設定
			;
	PSUB1		
			;

上記形式の連続で多倍長のハック化 BCD減算が行えますが、命令語数的に不利です。通常、多倍長のハック化 BCD減算はルーフ処理で記述します。ルーフ処理形式では任意のバイト数のハック化 BCDが扱え、実行速度は概ねバイト数に比例します。

ルーフ処理によるハック化BCD減算

ハック化BCD演算は桁当りの命令数が多いのでルーフ処理による演算が一般的です。次の例は任意のハイ数のハック化BCD減算ルーチンです。このルーチンの呼び出し時にはWレジスタにハイ数を設定する必要があります。

X	DS	1~	;ハック化BCD変数X定義
Y	DS	1~	;ハック化BCD変数Y定義
ONT	DS	1	;ハイトカウン定義
TMP			;一時保存データ(FLGと領域を共有)
FLG	DS	1	;BCD最終桁からのゼロフラグ変数定義
PSUB	MOVF	ONT	;ハイトカウンを設定
	MOVLW	X	変数X基準ポインタをWに取得
	MOVF	FSR	変数X基準位置を間接ポインタに設定
	BSF	FLG,0	結果ゼロフラグを初期化
PSUB1	MOVLW	Y-X	変数間オフセットをWに取得
	ADDWF	FSR,F	対応変数Y位置を間接ポインタに設定
	MOVF	INDF,W	減算値をWに取得
	BTFS	FLG,0	前回桁からのゼロなしでスキップ
	ADDLW	1	;ゼロありで減算値BCD下位桁+1
			;
	MOVF	TMP	減算値を一時保存
	MOVLW	Y-X	変数間オフセットをWに取得
	SUBWF	FSR,F	対応変数X位置を間接ポインタに設定
	MOVF	TMP,W	減算値をWに復帰
	SUBWF	INDF,F	;ハック化減算
	BSF	FLG,0	結果ゼロフラグを初期化
	RLF	FLG,F	;ゼロフラグを保存
			;
	MOVF	INDF,W	;ハック化減算結果をWに取得
	BTFS	DC	;ハック化減算の桁ゼロでスキップ
	SUBLW	9	;下位桁9以下か検査
			;
	MOVLW	16-10	;下位桁補整値 (=6をWに取得
	BTFS	DC	;9以下桁ゼロなしでスキップ
	SUBWF	INDF,F	;下位桁9以下に補整
			;
	RRF	FLG,F	;ハック化減算でのゼロフラグを復帰
	MOVF	INDF,W	;下位桁補整後データをWに取得
	BTFS	C	;ハック化減算でのゼロでスキップ
	SUBLW	9*16	;上位桁9以下か検査
	BTFS	C	;10以上(ゼロ)でスキップ
	GOTO	PSUB2	;9以下で分岐
			;
	MOVLW	(16-10)*16	;上位桁補整値 (=60HをWに取得
	SUBWF	INDF,F	;上位桁9以下に補整
	BCF	FLG,0	結果ゼロフラグを設定
			;
PSUB2	INCF	FSR,F	間接ポインタを進行
	DECFSZ	ONT,F	;ハイトカウン減数 =0でスキップ
	GOTO	PSUB1	;ハイ数分継続
			;
	RETURN		呼出元へ復帰

変数定義は必要とする最大ハイ数を定義します。

次にハック化BCDの各ハイ数に対する実行サイクル数の関係を示します。

ハック化BCD長と速度の関係

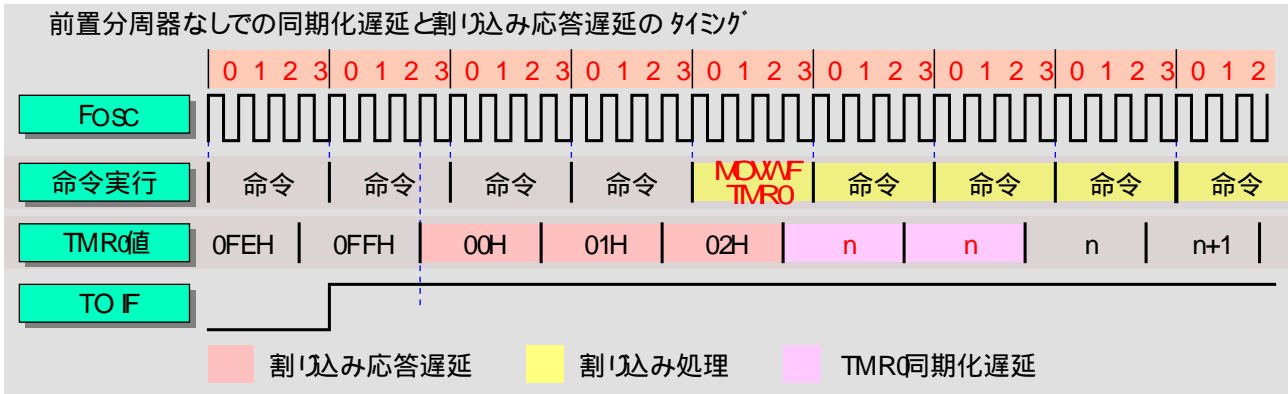
BCD桁数	BCDハイ数	命令語数	実行サイクル数	BCD桁数	BCDハイ数	命令語数	実行サイクル数
1,2	1	35	34 ~ 36	11,12	6	35	179 ~ 191
3,4	2		63 ~ 67	13,14	7		208 ~ 222
5,6	3		92 ~ 98	15,16	8		237 ~ 253
7,8	4		121 ~ 129	17,18	9		266 ~ 284
9,10	5		150 ~ 160	19,20	10		295 ~ 315

## TMR0の使用法

TMR0の使用で注意しなければいけないのは設定値です。基本的なことですが、TMR0は上昇エッジでOFFHからのオーバーフローで割り込みが発生します。このため、設定値は256-目的の計数値になります。実際のプログラム上では単に目的の計数値に負符号を追加するだけです。例えば目的の計数値が16の場合、設定値は256-16=240ですが、-16との記述は内部でOFFFFFFFF0H(32ビット表現)になり、その下位8ビットが採用され、結果は0E0Hで10進数の240になります。

実際に注意が必要なのは同期化のための停止時間と割り込み応答時間です。これらによって結果的にタイマ間隔が長くなる問題が発生します。特に前置分周器を有効(2分周以上)にする場合は大変複雑になります。

次に前置分周器なしでの上記問題点を示すタイミングを示します。



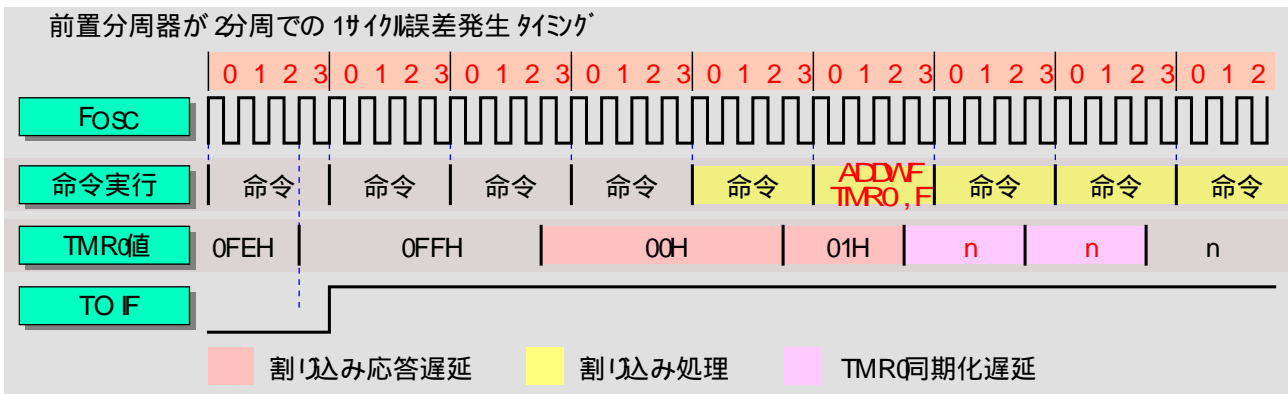
この場合、TMR0同期化遅延の2サイクルと割り込み応答遅延の3サイクル分を目的計数値に対して補正すれば正確なタイマ周期が得られます。両方共に遅延要素ですので、実際の設定値は上記での設定値 +5になります。上記例の16の場合は5-16のようになります。

現実には、この例の割り込み処理のようにいきなりTMR0書き込み命令となることはなく、レジスタ保存やタイマ設定値の取得命令などの後になります。その場合、それらの命令分の遅延が割り込み応答遅延に加算されます。これらを含めて最終的な設定値を求めます。

しかし、割り込み処理ルーチン先頭からTMR0書き込み命令間の命令実行サイクル数に応じて設定値を都度調整することになり非常に不便です。また、2サイクル命令実行時に割り込みが発生した場合は割り込み応答遅延が+1されるので、この方法では常に正確なタイマ周期が得られません。これを解決するにはTMR0に書き込む(MOVSF TMR0)ではなく、TMR0に加算する(ADDWF TMR0,F)方法にします。TMR0オーバーフロー後、TMR0は継続して00Hからの上昇計数を行っていますので、これはオーバーフロー後の経過サイクル数を表します。このため、補正値は常にTMR0同期化遅延の2サイクル分だけになります。

この方法でも前置分周器を使用する場合は更に話が複雑になります。上記2項目の遅延自体は変わりませんが、TMR0書き込み命令とTMR0計数クロック前置分周器出力との同期化も行わなければなりません。TMR0書き込みによって前置分周器がリセットされますので、TMR0書き込み命令はTMR0計数時と一致させる必要があります。ところがTMR0同期化遅延は常に2サイクルですので、実際にはTMR0計数時の2サイクル前で同期化を行わなければなりません。前置分周器が2分周の場合はTMR0計数時と一致させ、設定値を+1することでも行なえます。

前置分周器が2分周で問題(1サイクル誤差)となるタイミング例を次に示します。



前置分周器が2分周時のTMR0書き込み命令とTMR0計数クック前置分周器出力との同期化プログラム例を次に示します。

SYNC	MOVF	TMR0,W	現在のTMR0値をWに取得
	SUBWF	TMR0,W	;このサイクルで計数が検査
	BTFSZ	Z	;このサイクルが計数でスキップ
	GOTO	SYNC1	;このサイクルが計数で分岐
;			
SYNC1	MOVLW	1-16	;設定値-TMR0同期化遅延補正値をWに取得
	ADDWF	TMR0,F	;TMR0再設定補正(このサイクルが計数サイクル)

この例ではTMR0同期化遅延の2サイクル分が2分周の1カウンに相当するので、設定値を-1することで補正しています。4分周以上では上記例のような簡単な記述では行なえず、かなり複雑になります。次に4分周時の例を示します。

SYNC	MOVF	TMR0,W	現在のTMR0値をWに取得
	SUBWF	TMR0,W	;このサイクルで計数が検査
	BTFSZ	Z	;直前の命令サイクルが計数でなければスキップ
	GOTO	SYNC1	;2前の命令サイクルが計数で分岐
;			
	NOF		;SUBWF命令間が4+1サイクルとなるための補正
	MOVF	TMR0,W	現在のTMR0値をWに取得
	SUBWF	TMR0,W	;このサイクルで計数が検査
	BTFSZ	Z	;直前の命令サイクルが計数でなければスキップ
	GOTO	SYNC1	;2前の命令サイクルが計数で分岐
;			
	NOF		;SUBWF命令間が4+1サイクルとなるための補正
	MOVF	TMR0,W	現在のTMR0値をWに取得
	SUBWF	TMR0,W	;このサイクルで計数が検査
	BTFSZ	Z	;直前の命令サイクルが計数でなければスキップ
	GOTO	SYNC1	;2前の命令サイクルが計数で分岐
;			
	NOF		最終同期位置用補正
	NOF		;直前のGOTO SYNCより+1サイクルとなる補正)
;			
SYNC1	NOF		;一致時のSUBWF命令からADDWF命令間サイクル数補正
	MOVLW	-16	;4の倍数-TMR0同期化遅延の2サイクル正の最小値
	ADDWF	TMR0,F	;TMR0再設定補正(このサイクルが計数サイクル)

この例では計数サイクルを検査するSUBWF命令間が4+命令サイクルとなるようにしています。この計数サイクル検査を3回行うことで計数サイクル位置を得ています。3回目で一致しなかった場合は次のサイクルが該当位置であることが分かっていますので、単にNOF命令でサイクル数調整を行っています。この例のMOVLWの取得値は負符号を付加した目的の計数値そのものです。TMR0同期化遅延はプログラム実行サイクルで補正しています。

これ以上の分周値でも上記例と同様な手法で対応可能ですが、命令語数と実行時間からあまり実用的とは言えません。むしろプログラムによるカウンタを併用し、TMR0自体はできるだけ前置分周器なしで使用の方が得策です。

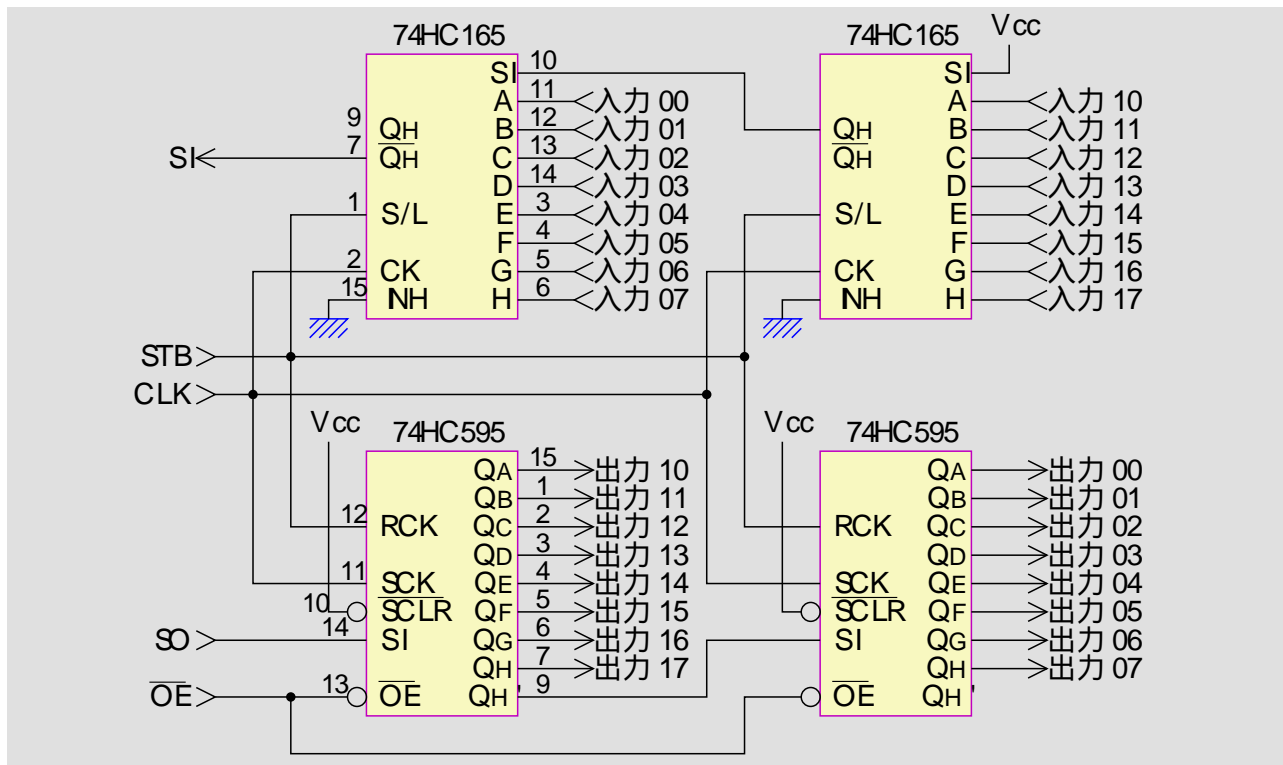
何れの方法を採っても、割り込み発生時に実行中の命令の実行サイクルが1の場合と2の場合でTMR0と割り込み処理ルーチン開始タイミングは1サイクルの誤差を生じますので、プログラムによるホ-出力などでのTMR0との同期化を完全正確に行うことは不可能です。

## 入出力の拡張

入出力の拡張方法には多くの方法がありますが、ここでは一般的な同期シリアル接続による方法を示します。パラレル接続の方が速度的に有利ですが、内蔵ポートを大量に消費しますので得策ではありません。シリアル接続で8ビットの入出力処理を行う場合のプログラム実行時間は10MHzクロックで約50μ程度になります。

シリアル接続の入力デバイスには74HC165,166などが使用できます。出力には74HC595が最適です。また、特定メーカーでは74HC595の互換や機能相当で出力形態がハイポートやFET駆動の大電流ドライバとなっているデバイスもありますので、入手が可能ならば目的に応じて選択します。

次に74HC165と74HC595を使用する場合の例を示します。



上記例は入出力共に16ビットの拡張例です。この方法ではIC個が8ビット分ですので、8ビット単位で任意の拡張ができます。3個目以降は2個目同様に縦列接続で結線します。

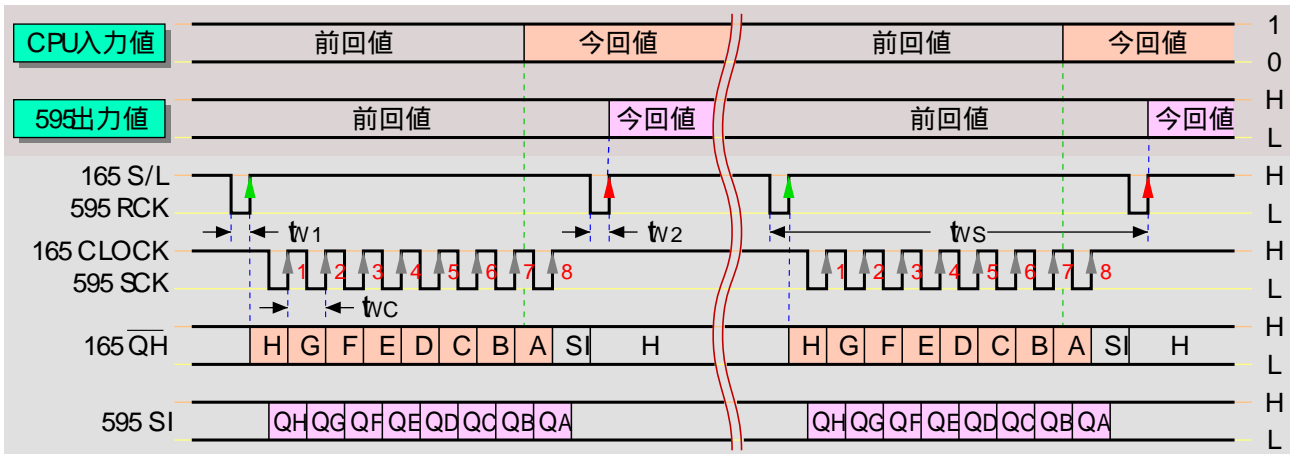
注意しなければならないのは順番で、図上のSIは左端のデバイスからの入力が入力され、SOは左端のデバイスが最後の出力となることです。

上図では入力ポートのSをQHに接続しています。これは入力が負論理入力の場合の接続方法で、正論理の場合はQH(番ピン)に接続します。このように入力の論理により論理を選択できる点が74HC165は74HC166より有利です。

この出力ポートの出力が出力ポート初期化まで不定でもよい場合はOEをGNDに接続できます。また、出力ポートの初期出力値がLならば、SCLRをリセット信号に接続することで、OEをGNDに接続できます。

PIC内蔵ポートの使用を少しでも減らしたい場合にはSとSOを接続し、1ビットで処理することもできます。その場合のSIは10k程度の抵抗を介して接続します。プログラムでは方向レジスタで内蔵ポートの該当ビットの方向を切り替えながら入出力を行います。この方法では1つのシフトレジスタ内で入出力を同時に処理できず、入力と出力を個別に処理しなければならないので、入出力処理時間が入出力同時処理に比べて倍近くかかります。

拡張入出力制御 タイミング図

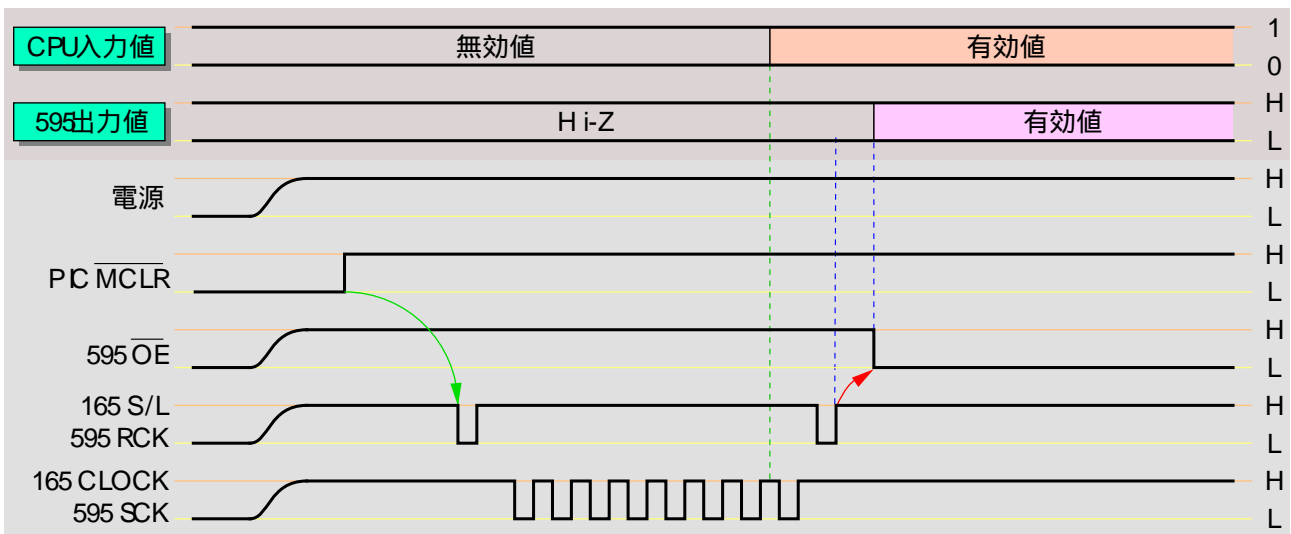


74HC165はS/L=L 実際には立ち上がりで 165のA~ H入力をラッチします。 のタイミングで代用可能ですが、この場合は前回の時点での入力状態になります。最新の状態を入力するためには のタイミングで行います。同時に74HC595でもシフトレジスタ内のデータがラッチに取り込まれますが、シフトレジスタの状態が変化していませんので、これによって595の出力は変化しません。

74HC595のRCKの立ち上がりで595のシフトレジスタ値をラッチレジスタにラッチします。この結果、595の各出力が同時に変化します。同時に74HC165でもA~ H入力のラッチが行われます。を行わない場合、これが165入力ラッチ動作になります。

74HC165のCLOCKと74HC595のSCKはシフトレジスタに対するシフトクロックで、共に立ち上がりでシフト動作を行います。

拡張入出力初期化 タイミング図



リセット後のプログラム開始で通常のようにホップ初期化を行います。その後に接続した拡張入出力ビットの入出力処理を行います(上図例は8ビット分) この時の出力データが74HC595の初期値になります。また、入力データも有効になります。

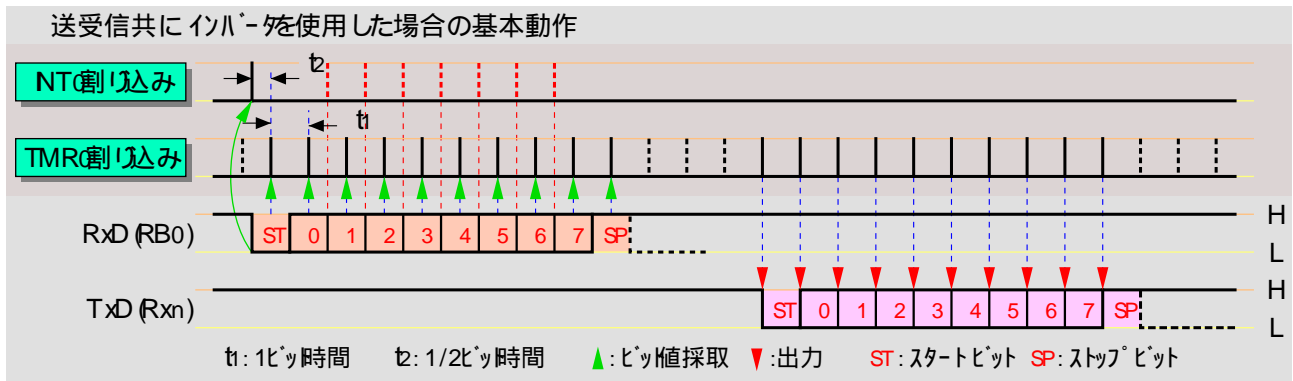
74HC595のRCKの立ち上がりで595の内部レジスタに指定値がラッチされますが、この時点での595出力端子はまだHi-Z状態です。この後に595のOEをLにすることで595出力端子が有効になります。

## NT0,TMR0利用のRS-232C

PC (16F84)にはハードウェアによるシリアル インターフェース機能がありませんので、外部割込みとタイマを利用し、ソフトウェアで実現しますが、同時送受信は不可で交互双方向通信になります。

送信の場合、ビット時間毎に出力信号 (TxD)を設定すればよいので、タイマの利用だけで実現できます。このタイマの設定値は1ビット時間です。

受信の場合は1バイト受信の起動となるスタートビット開始の検出が必要です。具体的にはスタートビットの立ち上がりは外部割込み (NT0)で検出し、ビット値採取用タイマを起動します。この立ち上がりは回線上での場合で、受信側にインパルスが存在する場合の入力信号 (RxD)では立ち下りになります。



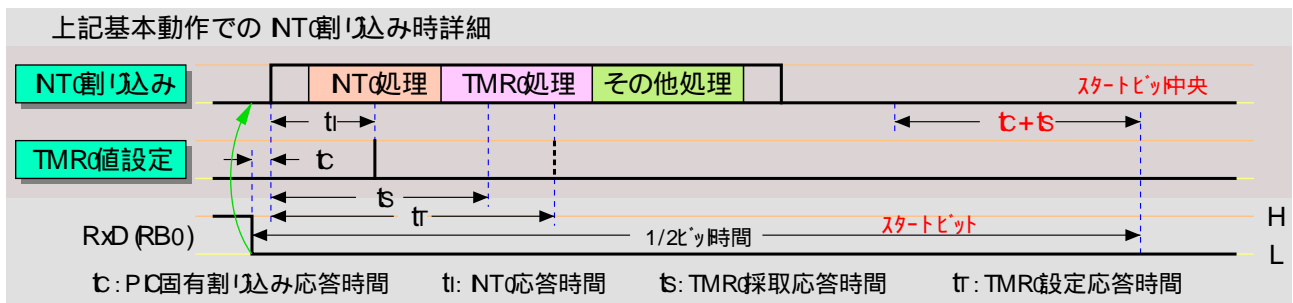
NT割込みでTMR0の計数値を1/2ビット時間に設定します。設定後、必ずTMR割込み要求フラグ(TOIF)をクリアします。以降のTMR割込みでは計数値を1ビット時間に設定します。

スタートビット開始後も殆どの場合、NT割込みが発生します。これらのNT割込みは1バイト受信完了までNT0割込みを禁止するか、または無視する必要があります。

TMR0の計数値は基本的に1/2または1ビット時間ですが、実際には割込み応答時間を考慮する必要があります。

送信時は各ビット出力処理間の最大誤差が実際の出力信号での誤差となりますが、通常、この差はプログラム数ステップ分程度ですので、転送レタが低い場合は無視できる値です。

受信時の各ビット受信処理も同様ですが、全体の基準となるNT割込みでのTMR計数値への1/2ビット時間設定は厳密さが要求されます。



NT0入力の有効エッジから割込み処理ルーチンが起動されるまでに(または2)+3サイクル(×4クロック)の遅延( $t_c$ )があります。

スタートビット検出のTMR割込みは $t_c+t_s$ 分の遅延が存在するため、位置で発生させなければなりません。また、TMR計数値設定時まで $t_c+t_r$ 分の遅延もあります。

これらのため、での設定値は1/2ビット時間 - ( $t_c+t_i$ ) - ( $t_c+t_s$ )になります。同様にTMR割込みでの1ビット時間も1ビット時間 - ( $t_c+t_r$ )にしなければなりません。

その他の割込み処理がNT0やTMR0と非同期に発生する場合、その他の割込み処理中にNT0やTMR0割込みを検査し、これらの割込みがある場合にその処理を優先するなどの処置をすることで、 $t_i, t_s, t_r$ などの応答時間を一定とする工夫が必要です。

同様にプログラム内で割込み禁止期間が存在すると、その期間分 $t_c$ に対する誤差要因となり、全体に影響を及ぼします。原則的に割込み禁止の使用は不可としてプログラムします。

命令一覧表

二-モニク/オペランド	意味	サイクル数	14ビットオペコード	更新ステータス	注意
ハイ単位のファイルレジスタ命令					
ADDWF f, d	Add W and f	1	00 0111 dfff ffff	Z, DC, C	1, 2
ANDWF f, d	AND W with f	1	00 0101 dfff ffff	Z	1, 2
CLRF f	C lear f	1	00 0001 1fff ffff	Z	2
CLRWF	C learW	1	00 0001 0xxx xxxx	Z	
COMF f, d	Complement f	1	00 1001 dfff ffff	Z	1, 2
DECf f, d	Decrement f	1	00 0011 dfff ffff	Z	1, 2
DECFSZ f, d	Decrement f, Skip if 0	1/2	00 1011 dfff ffff		1, 2, 3
INCF f, d	Increment f	1	00 1010 dfff ffff	Z	1, 2
INCFSZ f, d	Increment f, Skip if 0	1/2	00 1111 dfff ffff		1, 2, 3
ICRF f, d	Inclusive OR W with f	1	00 0100 dfff ffff	Z	1, 2
MOVF f, d	Move f	1	00 1000 dfff ffff	Z	1, 2
MOVWF f	Move W to f	1	00 0000 1fff ffff		
NOP	No Operation	1	00 0000 0xx0 0000		
RLF f, d	Rotate Left f through Carry	1	00 1101 dfff ffff	C	1, 2
RRF f, d	Rotate Right f through Carry	1	00 1100 dfff ffff	C	1, 2
SUBWF f, d	Subtract W from f	1	00 0010 dfff ffff	Z, DC, C	1, 2
SWAPF f, d	Swap nibbles in f	1	00 1110 dfff ffff		1, 2
XCRWF f, d	Exclusive OR W with f	1	00 0110 dfff ffff	Z	1, 2
ビット単位のファイルレジスタ命令					
BCF f, b	Bit C lear f	1	01 00bb bfff ffff		1, 2
BSF f, b	Bit Set f	1	01 01bb bfff ffff		1, 2
BTFSC f, b	Bit Test f, Skip if C lear	1/2	01 10bb bfff ffff		3
BTFSS f, b	Bit Test f, Skip if Set	1/2	01 11bb bfff ffff		3
リテラル制御命令					
ADDLW k	Add literal and W	1	11 111x kkkk kkkk	Z, DC, C	
ANDLW k	AND literal with W	1	11 1001 kkkk kkkk	Z	
CALL k	Call subroutine	2	10 0kkk kkkk kkkk		
CLRWDT	C lear Watchdog Timer	1	00 0000 0110 0100	$\overline{TO}$ , $\overline{PD}$	
GOTO k	Go to address	2	10 1kkk kkkk kkkk		
ICLW k	Inclusive OR literal with W	1	11 1000 kkkk kkkk	Z	
MOVLW k	Move literal to W	1	11 00xx kkkk kkkk		
RETFIE	Return from interrupt	2	00 0000 0000 1001		
RETLW k	Return with literal in W	2	11 01xx kkkk kkkk		
RETURN	Return from Subroutine	2	00 0000 0000 1000		
SLEEP	Go into standby mode	1	00 0000 0110 0011	$\overline{TO}$ , $\overline{PD}$	
SUBLW k	Subtract W from literal	1	11 110x kkkk kkkk	Z, DC, C	
XORLW k	Exclusive OR literal with W	1	11 1010 kkkk kkkk	Z	
シンボル 意味					
b	8ビットファイルレジスタ内のビットアドレス				
d	結果格納先指定子 ; d = 0 結果は W に格納 ; d = 1 結果は f (ファイルレジスタ) に格納 )				
f	ファイルレジスタのアドレス (00H ~ 7FH)				
k	リテラル 定数 テーまたはラベル				
W	作業レジスタ (累積器)				
x	無効 (または 1)				

- 注意 : 1. I/Oレジスタ外に対する Read modify Write (MOVWF RB, F など) はピンの入力レベルが使用されます。例えば入力ピンのデータラッチが 1、そのピンが外部デバイスにより Low レベルとなっている場合、データラッチには 0 が書き込まれます。
2. 前置分周器が TMRQ に割り当てられている場合、これらの命令を TMR0 レジスタ外に対して実行すると (TMRQ に対して書き込みを伴う場合) TMRQ に割り当てられている前置分周器がクリアされます。
3. プログラムカウンタ (PC) を変更したり条件付き試験の結果が真になると命令実行は 2 サイクル掛かり、2 番目のサイクルは NOP として実行されます。



---

© HERO 2006.

本書は中位PC (16F84留意点の記録です。

青字の部分はリクとなっています。一般的に赤字の0,1は論理0,1を表します。その他の赤字は重要な部分を表します。