MPASM

MPASM は Microchip Tecnology Inc.社の開発した PIC のためのアセンブリ言語である。ここでは MPASM の文法と使用法などについて記述する。

1. 文法

ソースコードファイルは ASCII テキストファイルエディターを使って作成する。そのように作られたソースコードは以下に示す基本的ガイドラインに従うべきである。

ソースファイルの各行は次の4つのタイプの情報を含んでよい。

・ラベル

・ニーモニック

・オペランド

・コメント

命令とこれらの位置づけは重要である。ラベルはカラム1から始まらなければならない。ニーモニックはカラム2かそれ以降から始まらなければならない。オペランドはニーモニックに続く。コメントはラベル、ニーモニック、オペランドに続くか、如何なるカラムから始まっても良い。カラムの幅の最大値は255文字である。

ホワイトスペースかコロンはラベルとニーモニック、ニーモニックとオペランドを区切らなければならない。複数オペランドはコンマによって区切られる。次に例を示す。

```
;
; Sample MPASM Source Code. For illustration only.
;
          list      p=16f54
Dest      equ       h'0B'

          org       H'01FF'
          goto      Start

          org       H'0000'

Start     movlw     H'0A'
          movwf     Dest
          bcf       Dest,3
          goto      Start

          end
```

1.1 ラベル

ラベルはカラム1から始まらなければならない。ラベルの後にはコロン、スペース、タブ、改行が続いてよい。

ラベルはアルファベットかアンダーバーで始まり、英数字、アンダーバー、クエスチョンマークを含んでよい。

ラベルは32文字までである。デフォルトではこれらは大文字と小文字を区別するが、コマンドラインで上書きできる。ラベルの定義でコロンが使われた場合、それはラベルオペレーターとして扱われ、ラベル自身としては扱われない。

1.2 ニーモニック

アセンブラ命令ニーモニック、アセンブラ指令、マクロ呼び出しはカラム2かそれ以上から始まらなければならない。同じ行でこれらがラベルである場合、命令はそのラベルからコロンか1つ以上の空白、もしくはタブで区切られなければならない。

1.3 オペランド

オペランドはニーモニックから一つ以上の空白かタブで区切られなければならない。複数オペランドはコンマで区切られる。

1.4 コメント

MPASM はセミコロン以降をコメントとみなす。セミコロンに続く全ての文字は改行が現れるまでの間無視される。セミコロンを含んでいる文字列定数は許されており、コメントとともに拒否されない。

2. 使用法

MPASM は前掲した命令セットを全て使うことが出来る。使用法は前掲したとおりである。

ここでは主に式構文とアセンブラ指令などについて記述する。

2.1 PIC16F8X 命令セット

PIC16F8X 命令セットは前掲したとおりのニーモニックで使用可能である。

オペランドは MPASM の式によって渡すわけだが、特殊機能レジスタのアドレスなどは予め P16F84A.INC というファイルでラベルとして定義されている。このファイルを include 指令で読み込むことによってより解りやすく、プログラムを記述することが出来る。

STATUS レジスタの Z フラグをテストすることを考えてみよう。

STATUS レジスタはデータメモリの bank0 0x03 番地にある。Z フラグはその3ビット目である。

このビットをテストするのに、例えば btfsz 命令を使うとすると、

```
        btfsz       0x03,3
```

となる。P16F84A.INC には次の定義がある。

```
STATUS  EQU     H'0003'
Z       EQU     H'0002'
```

EQU 指令は同義語を作る指令である。H'～'というのは16進数であること示す。(0x～と同じ。)

この定義を使って同じ内容を書くと以下のようになる。

```
        btfsz       STATUS,Z
```

このほうが「STATUS レジスタの Z ビットをテストしている」という事が解り易いだろう。

もう一例挙げてみよう。

addwf 命令を使ってワーキングレジスタと指定したレジスタの内容を足し合わせる。この命令にはオペランドが2つあり、1つ目は演算対象のレジスタのアドレス、2つ目は演算結果の格納先で 0 ならワーキングレジスタ、1 なら第一オペランドのアドレスに格納される。

いまワーキングレジスタと 0x0c 番地のレジスタの内容を足し合わせて結果をワーキングレジスタに格納したいとする。インクルードファイルによる定義なしで書くと以下のようになる。

```
        addwf       0x0c,0
```

P16F84A.INC には以下の定義がある。

```
W       EQU     H'0000'
F       EQU     H'0001'
```

これを用いて書くと

```
        addwf   0x0c,W
```

となる。W＝ワーキングレジスタ、F＝ファイルレジスタと覚えておけば、この方が解り易いだろう。

2.2 式構文

MPASM の式には演算子などを使って計算する機能などもあるが、今回の範囲ではそこまでは必要ない。ここでは基数ごとの数値リテラルの書き方について説明する。

以下の表にまとめた。

基数	構文	例
10進数	D'<digits>'	D'100'
16進数	H'<hex_digits>'	H'9f'
	0x<hex_digits>	0x9f
8進数	O'<octal_digits>'	O'777'
2進数	B'<binary_digits>'	B'00111001'

2.3 よく使うアセンブラ指令

ORG　　Set Program Origin

構文

[<label>]　　ORG　　<expr>

説明

この指令に続くニーモニックを<expr>で指定されるアドレスにおく。例えば、

```
        org     0x04
        movlw   B'00110011'
        movwf   0x0c
        ...
```

とすると、org に続く movlw 命令以降の命令はプログラムメモリの 0x04 番地以降に置かれる。これを指定しなかった場合、命令は 0x00 番地から順に置かれる。

EQU　　Difine an Assembler Constant

構文

<label>　　EQU　　<expr>

説明

<label>の文字列を<expr>で指定した値として関連付ける。例えば次のようにする。

```
hoge    equ     0x0c
```

このようにすると、式を必要とする場所で hoge と書くと、hoge は 0x0c という数値に展開される。
汎用レジスタを変数として使いたい場合に便利である。

CBLOCK　　Define a Block of Constants

構文

```
cblock　　[<expr>]
<label>
<label>
...
endc
```

説明

<expr>で指定される値から始まる定数のリストを作る。<expr>が指定されない場合、最初の CBLOCK 指令なら 0 から、2番目以降の CBLOCK 指令なら前の CBLOCK 指令の最後の値から始まる。例えば以下のようにする。

```
cblock　　0x0c　　;汎用レジスタの開始アドレス
var_1
var_2
var_3
endc
```

これで、var_1 = 0x0c、var_2 = 0x0d、var_3 = 0x0e として展開される。大量の一続きの汎用レジスタを変数として使いたい場合に便利である。

3. イディオム

ここではプログラムの中でよく使う手法を幾つか説明する。

任意の汎用レジスタに任意の値を入れる

任意のレジスタに任意の値を入れる、といった命令は無い。しかし、ワーキングレジスタに任意の値をロードする命令とワーキングレジスタの値を任意のレジスタに転送する命令はあるので、この二者を使う。

```
;
;0x0c 番地のレジスタに、255 をいれる
;

        movlw   H'FF'
        movwf   0x0c
```

フラグをテストして分岐する

演算結果フラグは STATUS レジスタにある。また割り込み使用時も割り込み要因特定のためにフラグをテストすることがある。

任意のレジスタの任意のビットがクリアされているかセットされているかをテストする命令として、btfss 命令と btfsz 命令がある。

この両者の命令は、テスト結果が真であればその次の命令をスキップする。（次の命令を NOP 命令に置き換える。）

btfss 命令は指定したレジスタの指定したビットがセットされていればスキップする。

btfsz 命令は指定したレジスタの指定したビットがクリアされていればスキップする。

これらスキップ命令と goto 命令等のジャンプ命令を組み合わせて分岐を実現する。

例えば以下のように書く。

```
;
;C フラグが立っていればワーキングレジスタに 255 を、そうでなければ 0 をロードする
;
        btfss   STATUS,C
        goto    lb0

        movlw   H'FF'
        goto    lb1

lb0     movlw   H'00'
lb1     ...
```

大小比較

任意のレジスタの値と任意の値を比較する。比較命令は無いので、減算して演算結果フラグをテストする。

```
;
;0x0c 番地の内容が 10 より大きいか小さいか
;

                subwf       0x0c,W
                btfss       STATUS,DC
                goto        lb0

                ;大きいの場合の実行内容
                ...
                goto        lb1

                ;小さいの場合の実行内容
                ...

lb1             ...
```

N 回ループ

任意回数のループは decfsz 命令を使うと便利である。

decfsz 命令は指定したレジスタの値をデクリメントして値がゼロならば次の命令をスキップする。格納先は W と F で選べるが、この場合は F を指定して使う。

```
;
;10回ループする
;

cnt             equ         0x0c        ;0x0c 番地をカウンタ変数として使う

                movlw       D'10'
                movwf       cnt
lb0
                ;ループの処理内容
                ...
                decfsz      cnt,F
                goto        lb0
                ...
```

定数テーブルを作成する

定数テーブルを作るには retlw 命令を使うと便利である。

retlw 命令はオペランドとしてリテラルをとりそのリテラルをワーキングレジスタにロードし、戻り番地へ戻る。

この命令を使う方法では、プログラムカウンタを操作する必要がある。これについては、後述する。

```
;
;PORTA の下位2ビットを見て
;次のテーブルで与えられる値をワーキングレジスタにロードして戻るサブルーチン
;
;00    →    10
;01    →    20
;10    →    30
;11    →    40
;
table       movlw       B'00001'
            movwf       PCLATH
            movlw       B'00000011'
            andwf       PORTA,W     ;PORTA の下位2ビット以外をマスク
            movwf       PCL

            org         B'0000100000000'
            retlw       D'10'
            retlw       D'20'
            retlw       D'30'
            retlw       D'40'
```

プログラムカウンタの変更のしかた

プログラムカウンタは 13bit である。対応するレジスタは PCLATH レジスタと PCL レジスタで、それぞれプログラムカウンタの上位 5bit と下位 8bit である。

PCLATH レジスタをライトした値は次回 PCL レジスタをライトしたときにプログラムカウンタに反映される。PCLATH レジスタは書き込みバッファである。

プログラムカウンタを変更したいときは先ず PALATH レジスタにライトし、それから PCL レジスタにライトする。