理アドレス（C0000H-CFFFFH ）上に存在する共有メモリ構成となっている。したがって、ＤＳＰがストップ状態の時には、ＣＰＵは直接このメモリ空間にアクセス可能である（ＤＳＰが実行中はアクセスできない）。ＤＳＰのプログラムメモリ上にデータ転送のための空間を確保して、そこのメモリを使ってＣＰＵとの間でデータ転送を行うのが、「共有メモリによるデータ転送」である。もちろん、ＤＳＰはプログラム領域にあるデータを読み書きできる（これはＣＰＵのプログラムで、ＣＳ領域内（_TEXT 内）にデータ格納箇所を定義していることと同じである）。ＤＰＲ転送の場合にはＣＰＵが直接ＤＳＰのプログラム領域のメモリをアクセスすることはないのでＤＳＰをストップさせることはないが、共有メモリデータ転送の場合にはＣＰＵの方でＤＳＰをスタート、ストップさせなければならない。以下、具体的にその転送方法を述べる（なお、ＤＳＰのプログラミングについては後章参照）。

ＣＰＵからＤＳＰのプログラム領域に定義したデータ箇所にデータを転送するには、そのアドレス（ＣＰＵの物理アドレス）が決まらないといけない。すなわち、ＣＰＵの転送用プログラムを作成するためには、まず、ＤＳＰのデータアドレスを決定しておく必要がある。データ領域の定義は

ＢＳＳ　あるいは　ＤＳ

で行う。例えば、

```
TRNS:  BSS  1
```

と書けば、変数「TRNS」という名前の１ワード（２バイト）領域が確保される（これはＣＰＵの「DW」に対応）。しかし、問題はこの領域の「番地」である。通常、ＤＳＰのプログラムはＤＳＰアドレスで「０番地」から開始するようにしている。しかし、プログラムメモリの下位番地（３１番地まで）は、割り込みベクタなどに予約されているために使用できないので、通常はこれらの転送用データ領域はプログラム領域の「５０番地」から定義する。そのために、次のようにプログラミングする。

```
ORG   0
      BSS 50
TRNS: BSS 1
```

これにより、「TRNS」の１ワードデータの番地は「５０番地（ＤＳＰアドレス）」の定義となる。なお、「ＯＲＧ」は「オリジン・ロケーション・セット命令」であり、「プログラムのロケーションカウンタ（機械語に翻訳する時に指定番地から出発しなさいという命令）を「０」にする」ということを指定するものである。実際のメインプログラムは、このデータ定義の後からプログラミングする。

これで、データ転送のための領域（例では１ワード）をＤＳＰ側で確保したので、ＣＰＵからこの番地を通してデータ転送が可能となる。例では、ＤＳＰのアドレスで「５０番地」に定義した。これは、ＣＰＵから見るとセグメント「C000H 」でオフセット「 100番地」、すなわち、物理アドレスで「C0064H」である。したがって、ＣＰＵからＤＳＰにデータを転送する場合には

```
MOV  AX,C000H
MOV  ES,AX
MOV  AX,(データ)
MOV  BX,100        (MOV  BX,64H)
MOV  ES:[BX],AX
```

のようにプログラミングする。あるいは、ブロック転送命令を使用して、ＣＰＵのデータ領域に定義した「SDAT」の内容を転送するには

```
MOV  SI,OFFSET DGROUP:SDAT
MOV  AX,C000H
MOV  ES,AX
MOV  DI,100
```

```
    MOVSW
```
となる。

データ転送を終了した後に、ＤＳＰをスタートさせる。ＤＳＰのスタートは前に説明した「ＣＲ」を操作し、次のように行う。

```
    MOV  DX,02D2H
    IN   AL,DX
    AND  AL,10111111     (b6=0)
    OR   AL,00000001     (b0=1)
    OUT  DX,AL
```

「ＣＲ」の内容は指定ビットだけの変更を行う。これによりＤＳＰは実行を開始し目的の処理を行い、演算結果を同じプログラムメモリ上に保存して（ここでは、同じメモリ「TRNS」に保存したとして）、演算終了の合図（ＤＰＲにより）をＣＰＵ側に送る。ＣＰＵはＤＳＰの演算が終了したことを確認して、まず、つぎの命令を実行してＤＳＰをストップさせる。

```
    MOV  DX,02D2H
    IN   AL,DX
    OR   AL,01000000     (b6=1)
    AND  AL,11111110     (b0=0)
    OUT  DX,AL
```

この場合も「ＣＲ」の指定ビットだけの変更を行う。その後、ＤＳＰのプログラムメモリの指定番地からデータを転送するために、次の命令を行う。

```
    MOV  AX,C000H
    MOV  ES,AX
    MOV  BX,100          (MOV  BX,64H)
    MOV  AX,ES:[BX]      (AX:データ)
```

あるいは、ブロック転送命令を使用して、ＣＰＵのデータ領域に定義した「RDAT」へ転送するには

```
    MOV  AX,DS
    MOV  ES,AX
    MOV  AX,C000H
    MOV  DS,AX
    MOV  DI,OFFSET DGROUP:RDAT
    MOV  SI,100
    MOVSW
    MOV  AX,ES
    MOV  DS,AX
```

とする。なお、この場合には「ＤＳ」の値を変えるので注意が必要である。

「共有メモリによるデータ転送」はこのようである。例では、１つのデータ転送の場合を説明しているが、複数の場合には転送するデータアドレスを順に並べて定義すれば、単にその数だけ「MOVSW 」命令を並べるだけとなる。この方法では、ＤＳＰとＣＰＵのデータアドレスが完全に一致していないと、とんでもない計算を行うことになるので、アドレスに十分注意して、慎重にプログラミングしなければならない。

４１．「ＤＳＰアセンブラプログラムの基本構成」

それでは、ＤＳＰアセンブラソースプログラムの構成を説明しよう。命令は異なるが、ＤＳＰアセンブラはＣＰＵアセンブラとほとんど同じである。ＤＳＰアセンブラ命令で使用する変数（オペランド）はほとんど「内部データメモリ」に定義した変数（ＣＰＵの「ＤＳ」内に定義した変数に対応）を対象としている。通常、ＤＳＰはＣＰＵの制御によって、スタート、ストップさせられる。この時、ＤＳＰの「内部データメモリ」の値は保持されないので、ＤＳＰで続けて使用する変数や定数は「プログラムメモリ」に保存しておく必要がある。また、「共有メモリによるデータ転送」では、ＣＰＵから直接アクセスできるメモリは「ＤＳＰプログラムメモリ」であるから、ＤＳＰでは、「プログラムメモリと内部データメモリとのデータ転送」が必要となる。ＤＳＰプログラム構成は以下のようである。

```
        （マクロ定義）

       ORG     0
       B    MPRO
       RET

       DSEG 4
        内部データ変数定義
       DEND

           BSS  47
        プログラムメモリ変数定義（共有メモリ）

MPRO:  DINT
       LDPK 4
         プログラムメモリから内部データメモリへデータ転送
         （ＤＰＲからデータ読みだし）
       「メインプログラム」
         （必要なデータのプログラムメモリへの保存）
         ＤＰＲへデータ書き込み（ＣＰＵへの演算終了合図）
 LOP:  B  LOP
       END
```

まず、必要ならば「マクロ定義」を行う。「マクロ定義」はＣＰＵの場合と同じであるが、「引き数名」は「＃１」、「＃２」のように決まっている。次に、「ＯＲＧ　０」を定義して、ソースプログラムのロケーションカウンタを「０」にし、開始アドレスを「０番地」とする。前述したように、ＤＳＰはＣＰＵによってスタート、ストップさせられる。ＤＳＰスタート時は「０番地」から実行を開始するが、「２番地から３１番地」は割り込みベクタ等に予約されているためプログラムは書けない。したがって、それ以降の番地からプログラミングしなければならない。そのため、「０番地」には「メインプログラム開始アドレス」にジャンプする命令を書いておく。ここでは、開始アドレス（ラベル）を「MPRO」としているので、「０番地」には「B MPRO」とする。これにより、ＤＳＰはスタートすると、「０番地」の命令（MPROに分岐しなさい）によって、「MPRO」の番地から実行を開始する。「２番地」は「割り込みベクタ」となっている。ＤＳＰでは割り込みは行わないが、ここでは「RET 」を定義しておく。

「DSEG」から「DEND」は「内部データメモリの変数」を定義する。ここで定義された変数は「プログラムメモリ」ではなく「ＤＳＰ内部データメモリ」である。したがって、ソースプログラム上では存在するが、最終的な機械語レベルのプログラムには存在しない。内部データメモリは「ページ４」を使用するので、「DSEG 4」として定義する。内部データメモリは「最大１２８ワード」であるので、これを越える変数の定義を行う場合には「別のページ（例えばページ５）」を定義する。

次に、ＣＰＵとの「共有メモリによるデータ転送」のためのデータ領域を定義するが、「３１番地」までは予約されているので、それ以降に定義する。ここでは、区切りのよい「５０番地」から「データ変数」を定義できるように、

「BSS　47」　（４７ワード確保）

とする。これにより、この次の番地は「５０番地」となる。プログラムメモリのデータ変数定義の後がメインプログラムとなる。

「DINT」命令は「割り込み不許可」を行うものである。割り込みは使用しないが一応書いておく。「LDPK 4」により、これから使用する「内部データメモリ」が「ページ４」であることを定義する。

プログラムではまず、「TBLR」命令によってＣＰＵから書き込まれたデータや使用する定数を内部データメモリに転送する。その後、必要ならばＤＰＲからデータを読み込む。そして、必要な処理のプログラムとなる。演算処理終了後、次回の演算に必要なデータやＣＰＵに送るデータ（内部データ領域に保存されている）をプログラムメモリのデータ領域に転送保存する。この時には「TBLW」命令を使用する。最後に、演算が終了したことをＣＰＵに知らせるために、ＤＰＲへデータを書き込む。この時、ＤＰＲへのデータとして何もなければダミーデータを出力する。あとは、ＣＰＵがＤＳＰの演算終了を確認してストップさせるので、ＤＳＰは「LOP: B LOP」の無限ループを実行し、ストップさせられるのを待つことになる。ＤＳＰソースプログラムの終了を「END 」で定義する。

ここで、よくわからない所はおそらく「内部データメモリ」と「プログラムメモリのデータ領域」の関係であろう。前述したように、ＤＳＰが命令として直接アクセスするデータはほとんど「内部データメモリ」である。「内部データメモリ」はＤＳＰ側だけに実装されている「高速ＲＡＭ」であり、ＣＰＵから直接アクセスできない。したがって、ＤＳＰプログラムメモリを通してＣＰＵとのデータのやり取りを行うわけである。すなわち、ＤＳＰはＣＰＵからのデータを一旦プログラムメモリで受取り、それを内部データメモリに転送することになる。ＤＳＰからＣＰＵにデータを送る場合も同様である。ＤＳＰのこの両者メモリの関係をＣＰＵに対応させると次のようになる。

```
（ＤＳＰ）                    （ＣＰＵ）
   DSEG                      DS内で定義した変数
VAR1:   BSS  1                 DVAR:   DW  ?
   DEND
                             CS内で定義した変数
XVAR1:  BSS  1                 CVAR:   DW  ?
  （ＰＭ定義）
```

例えば、変数データをアキュムレータ（ＣＰＵではAX）に移動する場合、

```
（ＤＳＰ）                    （ＣＰＵ）
 LAC  VAR1                    DS内変数  MOV  AX,DVAR
 (LAC  XVAR1 は不可)          CS内変数  MOV  AX,CS:CVAR
```

と書ける。ＤＳＰでは「プログラムメモリ」を直接アクセスできない。これはＣＰＵの「MOV 」命令が「セグメントオーバーライド」をしなければ、すべて「ＤＳ」

を使うことに対応する。したがって、「XVAR1 」のデータをＡＣＣに移動するには、まず、「XVAR1 」のデータを「VAR1」に転送しなければならない。そのために

```
LACK XVAR1
TBLR VAR1
```

を行う。これは、ＣＰＵの

```
MOV  AX,CS:CVAR
MOV  DVAR,AX
```

を行うことと同じである。逆に、「VAR1」のデータを「XVAR1 」に転送するには

```
LACK XVAR1
TBLW VAR1
```

となる。ＤＳＰでは演算開始前および終了後にＣＰＵとの転送データ、保存しておく必要のあるデータは全て、この処理を行わなければならない。通常、内部データメモリ変数と同じものをプログラムメモリにも定義するので、プログラムメモリの変数名はデータメモリ変数名の先頭に「Ｘ」を付けた名前にしている。
では、共有メモリを用い、２つのデータ（Ａ，Ｂ）を取り込み、その積（Ｃ）を求めるＤＳＰプログラムを以下に示す。

```
 1:        ORG     0
 2:        B    MPRO
 3:        RET
 4:
 5:        DSEG 4
 6:         VARA  BSS 1  ;A
 7:         VARB  BSS 1  ;B
 8:         VARC  BSS 1  ;C
 9:        DEND
10:
11:               BSS 47
12:         XVARA  BSS 1  ;A (CPU物理アドレスC0032-C00033H)
13:         XVARB  BSS 1  ;B (CPU物理アドレスC0034-C00035H)
14:         XVARC  BSS 1  ;C (CPU物理アドレスC0036-C00037H)
15:
16: MPRO:  DINT
17:        LDPK 4
18:        LACK XVARA
19:        TBLR VARA
20:        LACK XVARB
21:        TBLR VARB
22:
23:        LT   VARA
24:        MPY  VARB
25:        PAC
26:        SACL VARC
27:
28:        LACK XVARC
29:        TBLW VARC
30:        OUT  VARC,PA1
31: LOP:   B    LOP
32:        END
```

なお、このプログラムではプログラムメモリの変数「VARC」に結果が保存され、また、ＤＰＲへも結果を出力している。このＤＳＰプログラムに対するＣＰＵプログラムの流れは次のようになる。

（ＤＳＰストップ）
↓
変数Ａ、ＢをＤＳＰプログラムメモリへ書き込み
↓
ＤＳＰスタート
↓
（ＣＰＵ別処理）
↓
ＤＳＰの演算終了確認
↓
ＤＳＰストップ
↓
結果ＣをＤＳＰプログラムメモリから読みだし
（または、結果ＣをＤＰＲから取り込み）
↓
終了

４２．「ＤＳＰもコンパイルとリンクなんだ」

ＣプログラムやＣＰＵマクロアセンブラソースプログラムと同様に、ＤＳＰソースプログラムもそのままでは実行できない。実行可能ファイルというよりＤＳＰメモリにローディング可能なファイルを作成するためには、「コンパイル」と「リンク」を行わなければならない。その手順を説明しよう。

まず、「コンパイル」を行う。これのために、「ＡＳＭ２５．ＥＸＥ」が用意されている。例えば、ＤＳＰファイル名を「ＤＳＦ．Ａ３２」とし、ドライブＢの子ディレクトリ「DFILE 」に保存されている時、次のように入力する。

カレントドライブＡの時

```
A:¥>D:ASM25 B:¥DFILE¥DSF[.A32] [-l] [-s]
```

カレントドライブＢの時

```
B:¥>D:ASM25 DFILE¥DSF[.A32] [-l] [-s]
```

カレントドライブＤの時

```
D:¥>ASM25 B:¥DFILE¥DSF[.A32] [-l] [-s]
```

[] 内は省略可能である。これによりドライブＢの子ディレクトリ「DFILE 」内に「リロケータブルファイル DSF.RL 」が作成される。

「－１」指定時は「リストファイル DSF.LST」
「－ｓ」指定時は「シンボルファイル DSF.SYM」

がそれぞれドライブＢの子ディレクトリ「DFILE 」中に作成される。ソースプログラム中にエラーがあれば表示される。

次に、「リンカ」により実行形式のオブジェクトファイルを作成する。このために「ＬＩＮＫ３２．ＥＸＥ」が用意されている。「ＡＳＭ２５」により作成されたファイルに対して次のように入力する（カレントドライブＤの場合）。

```
D:¥>LINK32 B:¥DFILE¥DSF[.RL] [他のファイル[.RL] ・・・]
```

[] 内は省略可能である。これによりドライブＢの子ディレクトリ「DFILE 」内に

「オブジェクトコードファイル DSF.OJ 」が作成される。PUBLIC宣言されたシンボルがあれば「マップファイル DSF.MAP」が出力される。実際には、これらの操作はバッチファイル「ＶＸ．ＢＡＴ」で行う。
　ＤＳＰ実行可能ファイルは、前述したように、Ｃプログラムの子プロセス関数によってメモリへローディングされる。

４３．「データテーブルは有効な手段だ」

Ａ君：「これでほとんど、プログラムの内容やらはわかったような感じがするけど、僕たちが最終的に求めるのは、ＰＷＭパターンを発生するためのカウンタのカウント値でしょ」
Ｃ君：「そうたい」
Ｂ君：「ちょっと待って。ＰＷＭっちゃ何やったかいな」
Ｄ君：「あほ、それがわからんやったら卒論で何するか、わからんっちいうことやんか。「ＰＷＭ」は「パルス幅変調(Pulse Width Modulation)」のことたい。例えば、ラジオの「ＡＭ」は「振幅変調(Amplitude Modulation)」、「ＦＭ」は「周波数変調(Frequency Modulation)」とかいうやろ。また、「ＰＣＭ」は「パルスコード変調(Pulse Code Modulation)」などもあるやろ。こんなんの変調の一種たい」
Ａ君：「カウント値は、結局、何か計算して求めるわけやろ」
Ｃ君：「早い話が、数学的に２つの直線の交点を求める計算をＣＰＵかＤＳＰでするわけたい」
Ｄ君：「僕たちが作成するＰＷＭパターンは２つの信号で作るんたい。１つは三角波の搬送波と呼ぶ信号。もう一つは正弦波の変調波という信号たい。この２つの信号の交点を求めるわけだ。それを搬送波の１周期毎に区切って計算するんだ」
Ａ君：「パターンの作り方はだいたい知ってるけど、実際はどうやって計算するんや。三角波は直線だけど、変調波は直線じゃないから、ようわからん」
Ｃ君：「直線と正弦波の交点を求める計算式は超越関数になるんで、普通の方法じゃ求まらないんだ。すなわち、解がＸ＝とかいう形で一般的に表せんったい。こういうのは、ニュートン法なんかで求めるけど」
Ｄ君：「だから、実際の計算では、ある範囲で正弦波を直線近似してるんだ。そうすると、結局、２つの直線の交点を求めればいいということになるので、計算式が簡単になるっちゃ」
Ａ君：「でもね、三角波はもともと直線だから問題ないけど、正弦波の値はどうやって計算するわけ」
Ｂ君：「そんなもん、ＳＩＮを計算すればいいやんか」
Ａ君：「ＳＩＮを求める計算ってどうするんや」
Ｂ君：「そんなもん、ＳＩＮの所を押せばすぐ答えがでてくるぜ」
Ａ君：「ばかたれ。それは電卓の話やろが。機械語でどうやって計算するか、っちゅうことたい」
Ｂ君：「そんなん知らん」
Ｄ君：「機械語でも、ＳＩＮの計算なんかはできるけど、そんなことするととんでもない時間がかかるから、使えない」
Ａ君：「じゃぁ、どうするんじゃ」
Ｄ君：「それで、「データテーブル」というのを作るんたい」
Ｂ君：「「データテーブル」？、「データ」は「数値のデータ」やろ。「テーブル」は「飯なんか食うテーブル」やろ。データをテーブルに置いて何するんや」

C君：「テーブルには「表とか一覧表」とかいう意味があるやろが。だけん「データテーブル」は「データ一覧表」たい。B、おまえちょっと黙っとけ。おまえがしゃべると話が進まん」
D君：「それで、予め計算した正弦波のデータをメモリに保存しておいて、計算に必要な正弦波データをそのテーブルから読み出すわけたい」
A君：「でも、どこのデータを読み出すかわかると？」
D君：「だから、読み出すデータのアドレスとその内容を対応させて、テーブルを作っとかないといかん」
A君：「そりゃ、難しいぜ」
B君：「うん、難しい」
C君：「プログラムでは、その他に平方根を求めるためなんかにもデータテーブルを使いようよ」
B君：「あら、そう」
D君：「そしたら、この「データテーブル」の考え方を説明するね。そげん難しくないから」
D君は説明を始めた。
「ＣＰＵでもＤＳＰでも考え方は同じだから。じゃ、簡単な場合で、データテーブルを使って「０から１０の２乗を求める」ということをやろう。０から１０までの２乗の値は予めわかるので、それを下図のように「ＤＳ」内（ＣＰＵの場合）に書いておくんだ。その先頭番地をいま「SDAT」としておくね。
そこで、いま、ＡＸの２乗を求めようとする時、次のようにプログラムすればいいんだ。

```
    MOV   BX,OFFSET DGROUP:SDAT
    ADD   BX,AX
    MOV   AX,[BX]
```

あるいは、

```
    MOV   BX,AX
    MOV   SI,OFFSET DGROUP:SDAT
    MOV   AX,[SI+BX]
```

すなわち、求めようとする値をそのまま「SDAT」からのオフセットとすれば、「SDAT」にその値を加えた番地に必要データが保存されていることになるんだ。

ｱﾄﾞﾚｽ	ﾃﾞｰﾀ
SDAT+ 0	0
+ 1	1
+ 2	4
+ 3	9
+ 4	16
+ 5	25
+ 6	36
+ 7	49
+ 8	64
+ 9	81
+10	100

このテーブルの利用すれば、乗算命令を使わんで、２乗の値が求まるというわけだ。実際は計算に必要なデータの数だけテーブルを作成するということになるけどね。
ＤＳＰに場合には、プログラムメモリの空き領域にこんなデータテーブルを作成すればいいんだ。例えば、ＤＳＰアドレスの 2000Hから同じテーブルを作ったとすれば、プログラムは次のようになるんだ（ACCの２乗を求める）。

```
    ADLK  $2000
    TBLR  RES       (結果は変数 RESに保存)
```

わかったかなあ」
A君：「じゃ、正弦波データの時はどうなるの」
D君：「まず、決めることはどれだけの分解能のデータが必要かということだ。簡単な場合として、１度きざみのデータでよければ０度から３６０度までとして、それぞれの角度のデータを順番に保存しておけば、あとは、さっきと同じやり方で求められるでしょう」
A君：「そうか。わかった」
D君：「でも、正弦波の値は実際は－１から１までの値、要するに実数だからこれをどういった数値で保存しておくかを考えてないといけないね」

Ａ君：「そうか。例えば、0.25とか-0.6という値か」
Ｄ君：「そう。そういう場合に、前にもちょっと出てきたけど「固定小数点表示」が必要になるんだ。また、電圧や電流の大きさを表す場合にもこの表示が必要になるよ。この表示方法は別として、データテーブルの考え方はこんなもんかなあ」

データテーブルはある決まった計算を行う場合、１対１に対応した値が得られる時に非常に有効である。演算は結果が保存してあるアドレスを求めるだけであり、実際の演算は行わないので、演算時間の大幅な短縮が可能となる。だたし、テーブルのためのメモリが必要であり、分解能が高いほどデータテーブルメモリ領域は大きくなる。データテーブル作成のポイントは、演算するデータの値とテーブルのアドレス（テーブルの先頭番地からのオフセット）をいかにして対応させるかである。
　データテーブルはかなりのメモリ大きさとなるので、これらのデータ値は「ＢＡＳＩＣ」によって作成し、データをディスクに保存して実行時にローディングするようにしている。ＣＰＵに対するデータテーブルとＤＳＰに対するデータテーブルの形式は同じであるが、メモリへのローディング方法が異なるので、「ＢＡＳＩＣ」で作成する方法が違ってくる。ＣＰＵのテーブルは通常の「バイナリイメージデータ」（POKE命令でメモリに作成して BSAVEによって保存する）としてディスクに保存するが、ＤＳＰのテーブルローディングはＤＳＰプログラムローディングの「LD25.EXE」を使用するため、それに対応した「テキストイメージデータ」（PRINT #命令による直接ファイル書き込みで作成）としてディスクに保存する。詳しい作成方法は、実際のＢＡＳＩＣプログラムを参照のこと。

４４．「固定小数点演算」

　それでは、このテキストの最後として、「小数点演算」について少し説明しておこう。通常、コンピュータで扱う数値は整数であるが、それを整数とみなすか小数とみなすかはプログラマの考え方によっている。整数値とする時、小数点は０ビットの右側、すなわち、１６ビットでは
　００００　００００　００００　００００.
である。また、前述したように、８ビット目に小数点があると考えた場合には
　００００　００００.０００　０００
となる。いずれの表現でも「固定小数点表示(Fixed-Point Representation)」であるが、下の場合には、小数部も表現されており、これは「分数固定小数点表示(Fractional Fixed-Point Representation)」という。これらの表現による演算が「固定小数点演算(Fixed-Point Arithmetic)」である。卒論のプログラミングではこれを使用している。
　これに対し、実数を仮数部と指数部に分けて表示した場合を「浮動小数点表示(Floating-Point Representation)」という。これは、例えば、１６ビットの内、１２ビットを仮数部、４ビットを指数部に分けて表示する。すなわち、
　（仮数部）×２の（指数部）乗
として表す。この表現による演算は行っていないので、興味のある人はマニュアルなどで勉強して下さい。

　では、固定小数点演算について、実際に数値を使って説明しよう。分かりやすいように電圧、電流と抵抗を使ってオームの法則を計算する。固定小数点表示の場合、まず、それぞれの値に対して、小数点の位置を何処に置くか（小数部のビット数をいくらにするか）を決めなければならない。これにより、表現できる値の大きさと

分解能が決まる。各値を１６ビットで表現するとして、電流は小数部８ビットとしてみよう。すなわち、

（整数部）　　（小数部）

０００００００００．００００００００

正負表現のため、ＭＳＢは符号を表すので、この最大値は

01111111.11111111 = 127.99609375 (A)

単位は一応「Ａ」である。また、最小値は

10000000.00000000 = -128 (A)

となる。よって、この範囲の値を表現できることになる。この場合の分解能はＬＳＢが「２の（－８）乗」を表すので、

1 LSB = 0.00390625

となる。すなわち、この精度で表現可能となる。これは当然

128 - 0.00390625 = 127.9960375

である。実際に実験で用いる電流値は高々１０Ａ程度なので、電流の表現は小数部８ビットで十分である。小数部ビットを大きくすれば精度はよくなるが、そこまで精度をあげる必要はない。

抵抗値も電流と同様の精度としておく。

電圧は実際には３００Ｖ以上の値をとり得るので、小数部８ビットでは表現できない。したがって、小数部４ビットとすると、最大値は

01111111 1111.1111 = 2047.9375 (V)

最小値は

10000000 0000.0000 = -2048 (V)

となる。また、分解能は

1 LSB = 0.0625 (V)

である。以上のように小数部ビット数を決める。

では、電流 I=5.5(A)、抵抗 R=6.1(Ω) の場合の電圧Ｖを求める。

電圧は　V = IR = 5.5*6.1 = 33.55 (V)

固定小数点表示では

電流　1408D = 0580H = 00000101.10000000 = 5.5(A)

抵抗　1562D = 061AH = 00000110.00011010 = 6.1015625(Ω)

である。抵抗は 6.1(Ω)に最も近い表現を使う。これを例えばＣＰＵで計算するために、「IMUL」命令を行うと結果「DX･AX 」は

00218F00H = 00000000 00100001.10001111 00000000

となる。これは、このままの小数点位置であれば、

33.55859375(V)

を表している。ここで、小数部８ビット表現同士の積を行うと、得られる結果は小数部１６ビット表現となることに注意しないといけない。電圧は小数部４ビットとしているので、この表現に対応して、この３２ビットデータから取り出す（１２ビット右シフトして下位、あるいは４ビット左シフトして上位をとる）と

00218F00H = 00000000 00100001.10001111 00000000

↓

0000 00100001.1000 = 0218H

これは「33.5(V) 」である。実際値と「0.05(V) 」の誤差を生じる。これは、小数部８ビットから４ビットにおとしたためである。誤差を小さくするためには、小数部ビット数を大きくしなければならない。乗算では小数部ＮビットとＭビットの積は小数部（Ｎ＋Ｍ）ビット表現の数値となる。

次に、電圧 V=123.7(V) 、抵抗 R=8.5(Ω)の場合で、電流を求めてみよう。

電流は　I = V/R = 123.7/8.5 = 14.55294118 (A)

固定小数点表示では

電圧　1979D = 07BBH = 00000111 1011.1011 = 123.6875(V)
抵抗　2176D = 0880H = 00001000.10000000 = 8.5(Ω)
である。電圧は123.7(V)に最も近い表現を使う。これをＣＰＵで計算するために、「IDIV」命令を使うが、抵抗は１６ビット表現なので、電圧を３２ビット表現に拡張して行うと
0000 07BBH / 0880H = 0
となる。これは問題である。どうしてこういう結果になるかと言うと、小数部ビットの対応を考えていないからである。すなわち、除算の場合には乗算とは逆に商の小数部は、被除数が小数部Ｎビットで除数が小数部Ｍビットでは、商は小数部（Ｎ－Ｍ）ビットとなるためである。上の計算では商の小数部ビット数は（－４）となっている。したがって、小数部８ビットの商を得るためには、予め被除数を１２ビット上げておく必要がある。すなわち、電圧は３２ビットで
0000 0111 1011.1011
↓
0000 0000 0111 1011.1011 0000 0000 0000 = 007BB000H
とする。これで演算を行うと
007BB000H / 0880H = 0E8DH
これは小数部８ビットとなっているので、
0000 1110.1000 1101 = 14.55078125(A)
となる。演算による誤差は、電圧の表現誤差によるものである。除算の場合には商として得る値の小数部ビット数を考慮して演算する必要がある。

固定小数点演算は以上のようなことになる。これらの計算においては演算誤差すなわち有効桁数に注意しておかないといけない。

上の演算をもう少し分かりやすく書くと次のようになる。すなわち、電流と抵抗は小数部８ビットとしている。これらの１６ビット値は単に
「１」を「２５６＝２の８乗」に対応（実際の値の２５６倍）
させているだけである。電圧は
「１」を「１６＝２の４乗」に対応（実際の値の１６倍）
している。
乗算では
I = 5.5 (A) = 5.5*256 = 1408
R = 6.1(Ω) = 6.1*256 = 1561.6 → 1562
とし、
IR = (1408/256)*(1562/256) = (1408*1562)/(256*256)
= 2199296/(256*256)
= 00218F00H/(256*256)
結果は実際値の６５５３６倍となる。１２ビット右シフト（１／４０９６倍）するので、
V = 2199296/4096
= 536.9375 → 536 = 0218H
となり、実際値の１６倍が得られる。
除算では
V = 123.7 (V) = 123.7*16 = 1979.2 → 1979
R = 8.5 (Ω) = 8.5*256 = 2176
とし、電圧は１２ビット左算術シフトをするので
V = 1979*4096
となる。この時点で、電圧は実際値の（２の１６乗）倍である。これを２５６倍の

抵抗値で割るので
　　V/R = (1979*4096)/2176 = 3725.176471 → 3725 = 0E8DH
となり、実際値の２５６倍の電流が得られる。

実際の値で考えると
乗算は
　　Ｉ×（８ビット）×Ｒ×（８ビット）＝ＩＲ×（１６ビット）
１２ビット右シフトして
　　ＩＲ×（１６－１２ビット）＝ＩＲ×（４ビット）
となる。
除算は、まず、電圧を１２ビット左シフトするので、
　　Ｖ×（４＋１２ビット）＝Ｖ×（１６ビット）
よって、
　　Ｖ／Ｒ＝Ｖ／Ｒ（１６－８ビット）＝Ｖ／Ｒ×（８ビット）
となる。

このような演算では、乗算ではオーバーフローは起こらないが、除算では商が１６ビットに納まるよう、扱う数値の大きさに注意しないといけない。