III. 応用編

5. ソフトウェアの実際 (I)

NHK 総合技術研究所 村上 敬之助
榎 並 和 雅
大 関 健 二

前回までは、マイクロコンピュータへのデータの入力の方法について述べて来た。今回からは、入って来たデータを処理するという事でソフトウェアを中心に述べる。その後、出力機器へのデータ出力制御、補助記憶装置、システム開発へと講座を進めていく予定である。

今回はデータ処理のうち、プログラミング技法と演算の基本について述べよう。

1. プログラミング技法

基礎編、ソフトウェア(II)にも述べてあるように、広い意味でのプログラミングは図1のように進めるのが普通である。この中でデバッグ作業は意外に時間がかかるもので、デバッグの時間の長短によってプログラミングの上手下手がわかるともいわれている。上手なプログラミングを行うコツ、いいかえればプログラミング技法についてふれるてみる。

1.1 良いプログラミングとは

良いプログラミングとは、だいたい次のようなことをいうのではないだろうか。

- (1) ソフトウェア開発期間が短い。
- (2) コンピュータ実行時間、使用メモリ容量が小さい。
- (3) 他人が見てもわかりやすい。

(1)については上述したように、デバッグ時間を減らすことによってソフトウェア開発期間の短縮に大いに寄与する。そのため、ゼネラルフロー作成の段階からデバッグのことを念頭に入れて進めることが重要である。

ゼネラルフローの段階では、できるかぎり細かくブロック化し、各ブロックをサブルーチンにする。そしてゼ

ネラルフローにそったメインプログラムから呼び出すとデバッグがしやすくなり、プログラムも簡潔になる(基礎編、ソフトウェアII参照)。あるブロックをサブルーチン化するという事は、同じプログラムを何回か使う時に用いるとメモリ容量が節約できるという有用性があるだけでなく(これは上述(2)にもつながり重要なことであるが)、プログラムの汎用化や、たとえ1回しか使わ

図1 プログラミングの開発手順

"Microcomputer Techniques for TV and Radio Engineers; 9. Programming Techniques (I)" by Keinosuke Murakami, Kazumasa Enami and Kenji Ohozeki (NHK Technical Research Laboratories, Tokyo)

れなくとも、サブルーチン化により上述のようにデバッグの容易さ、プログラムの簡潔さにつながる。

コーディングの段階では、プログラム中の随所に NOP 命令を入れておくとうい。デバッグ中、NOP を HALT 命令に変えれば、それ以前のプログラムのチェックが容易になる。さらにいえば NOP を 3 個連続して入れるとよい。そのプログラムに誤りがあって修正するとステップ数が長くなり、それ以下のプログラムの移動をせまられる場合がある。こうしたときに 3 個の NOP を 3 バイト命令である JMP 命令または CALL 命令に変更し、別のメモリエリアに延長することができる。デバッグが終了したら整理すれば良い。

その他、デバッグをしやすくするための工夫がいくつかあるが、ハードウェアとのからみもあるので後章のシステム開発のところで述べることにする。

(2)の実行時間或使用メモリ容量を小さくすることに対しては、使用マイクロコンピュータの命令を熟知していることが重要である。同じ目的のプログラムを作るにしてもいくつかの方法があるが、命令の使い方によってステップ数や実行時間が違ってくる。図 2 はその例である。8080 ではペアレジスタ操作命令をうまく使うと効率的なプログラムを書くことができる。

(3)の、わかりやすいプログラムを書くことに対しては、リストにおけるコメントが重要な役割を持つ。自分が作ったプログラムでも数週間もたてば何のプログラムだったか忘れてしまうこともある。リストにおいて、さらに機能を表わしたラベルの付け方(たとえば割算のプログラムなら“DIV”等)にも留意する必要がある。図 3 にリストの書き方の一例¹⁾を示した(この説明は後述する)。

また、コーディング時のフォーマットの統一も重要である。たとえば各レジスタの役割を決めておく。すなわち、BC レジスタはデータを仮りにしまっておくテンポラリレジスタに、DE レジスタはカウンタに割当て等である。これはプログラムを何人かで手分けして作る場合にも役立つ。

システムが完成したときには、リストの他に各種のフローチャートやメモリアップ、プログラムの説明文などのドキュメントの作成も忘れてはならない。ハードウェアにおける系統図、回路図、動作説明文などが必要であると同様に、ソフトウェアにおいても保守、改修のことを考慮しておかねばならない。

以上のようにプログラミングとは、単に

0010 番地にデータ FF₍₁₆₎を入れるプログラム

冗長な例

```
MVI H, 00
MVI L, 10
MVI A, FF
MOV M, A
```

↓

簡潔な例

```
STA 0010
```

図 2 無駄なプログラムと簡潔なプログラムの比較例

コーディングしてコンピュータがうまく処理してくれれば終りというものではなく、いかに効率的に、いかに簡潔に行うかが重要なことである。

1.2 プログラミング上の注意点

プログラミングを行ううえで間違いやすいこと、いいかえればプログラミング上の注意点について筆者らの経験や文献²⁾等から述べてみよう(注:実際にこれから 8080 を使ってみようという人のための注意点であり、他のプロセッサを使おうとしている人や概念として勉強しようとしている人はこの項を読み飛ばしてかまわない)。

(1) フラグ

8080 にはフラグとしてよく使うものに C, Z, S, P があることは基礎編でおわかりいただけたと思う。ただし、命令によってフラグが影響をうけるものとうけないものがあるので、よく調べないと間違いのもとになる。特に INR, DCR は、Z, S, P の各フラグが影響されるが、C フラグは変らない。逆にシフト命令(RAL, RAR 等)は C フラグが影響を受け、Z, S, P フラグは不変である。

```
; 8BIT BINARY DIVISION
;
; CALLING SEQUENCE
; [(B,C)←DIVIDEND]
; [(D) ←DIVISOR]
; CALL DIV
; RETURN
; [(C) ←QUOTIENT]
; [(B) ←REMAINDER]
; USED REGISTER A,B,C,D,E
; FLAG S,C,P,Z
; STACK 1LEVEL
;
DIV ORG 0100
MVI E 09 ;BIT COUNT
MOV A B
MOV B A
MOV A C ;ROTATE CARRY INTO C REGISTER
RAL ;ROTATE NEXT MSB TO CARRY
MOV C A
DCR E
JZ DONE
MOV A B ;ROTATE MSB TO HIGH ORDER
RAL ;QUOTIENT
SUB D ;SUBTRACT DIVISOR,IF LESS THAN
JNZ LOOP ;HIGH ORDER QUOTIENT,GOTO LOOP
ADD D ;OTHERWISE ADD IT BACK
JMP LOOP
DONE RAL
MOV E A
MVI A 0FF ;COMPLEMENT THE QUOTIENT
XRA C
MOV C A
MOV A E
RAR
RET
END
```

図 3 計算プログラム DIV のリスト例

```
0100 0100
0101 1E09
0102 78
0103 47
0104 79
0105 17
0106 4F
0107 1D
0108 CA1501
0109 78
010A 17
010B 92
010C C20301
010D 82
010E C30301
010F 17
0110 5F
0111 3EFF
0112 A9
0113 4F
0114 7B
0115 1F
0116 C9
```

ERRORS 0000

```

00000110 ..... 06(16) ..... 00000110
+ ) 11111111 ..... 1(16) ..... 00000001
  00000101 ..... 5 ..... 00000101
Cフラグ1になる      Cフラグ0である(桁借りが無い).
(a) 加算命令のとき      (b) 減算命令のとき

```

図4 加減算命令とCフラグ

```

Aレジスタ 11110110 (F6(16))
Bレジスタ 10110100 (B4(16))
      ↓ CMP B
Aレジスタ 11110110 } 内容は変らない
Bレジスタ 10110100 }
Cフラグ 0 ..... 桁借りが無いから
Sフラグ 0 ..... 引いた結果が01000010 で正だから
Pフラグ 1 ..... " " で偶数だから
Zフラグ 0 ..... " " で0でないから

```

図5 比較命令とフラグ

図6 スタックは対で用いる

また論理命令 (ANA, ORA 等) では, Cフラグが強制的にリセットされる。DAD 命令ではアキュムレータを通さないが, Cフラグが影響をうける。

Zフラグは結果が0のとき1がセットされる。デバッグのとき, これを意外に間違いやすい。

加算のときのCフラグは理解しやすいが, 減算のときの動作は加算のときの逆になるので注意されたい。図4に例を示した。また比較命令 (CMP, CPI) のフラグの状態も図5に示した。Sフラグの状態に注目されたい。アキュムレータの内容のMSBだけに注目していると間違えることになる。

また全フラグをクリアする命令として XRA A がある。これをフラグ操作の前に入れておくこととフラグの状態が明瞭になる。

(2) スタック操作

プログラムの最初にスタックポインタのセットを忘れてがちなので注意したい。

スタックを有効に使うと効率的なプログラムが書けることがあるが, 充分注意しないと間違いが発生しやすい。PUSH と POP, CALL と RET, RST (リスタート) と RET は対で用いるのが通常のやり方で, 数式に

```

/*PARAMETER      ORG @ 1100
EQU PARA1=@11
EQU PARA2=@22
EQU PARA3=@33
EQU PARA4=@44      } パラメータの定義

1100 CD0811      CALL LDS      /サブルーチン コール命令
1103 11          PARA1
1104 22          PARA2
1105 33          PARA3
1106 44          PARA4
1107 00          0      } パラメータの設定

/.....
/
1108 E3          LDS :XTHL      /リターンアドレス(#1103)をHLに
1109 7E          MOV A M      /PARA1をACCにとり込み処理
      :
110A 23          INX HL      /HL ← #1104
110B 7E          MOV A M
      :
110C 23          INX HL      /ACC ← PARA 2
110D 7E          MOV A M
      :
110E 23          INX HL
110F 7E          MOV A M
      :
1110 23          INX HL
1111 E3          XTHL      /リターンアドレス(#1107)を
1112 C9          RET      スタックに入れる

END

```

図7 スタックを積極的に使った例 (サブルーチンで使うパラメータをコール命令の後に入れて受け渡すプログラム)

おける(と)の関係のようなものである(図6参照)。サブルーチンに飛んだ後, 不注意に POP B などとやるとリターンアドレスがどこかに行ってしまうと, RET 命令を実行したときにはまったく異なったアドレスがPC(プログラムカウンタ)に入り, あらぬ方向に飛んでしまうことになる。

ただし, 先程述べたように, スタックを使うと便利ことがある。たとえば, サブルーチン内で使うパラメータ(引き数)の受け渡しをレジスタやRAM領域に入れて行うことはよくあるが, その代わりに図7のような方法もある³⁾。メインプログラムのCALL命令の次にパラメータを入れておく方式である。サブルーチンLDSに入るとスタックにはコール命令の次のアドレス(#1103₍₁₆₎)が入る。普通のコール命令の場合, 戻り番地であるが, ここではパラメータ PARA 1 のアドレスである。そして XTHL の命令で HL の内容とスタックポインタで示される内容 (1103₍₁₆₎ 番地の PARA 1) とを交換し, HL に 1103₍₁₆₎ を入れる。これで MOV A M とすれば PARA 1 をアキュムレータに引いて来ることができる。PARA 2 以下を引くときは, HL を1ずつインクリメントすればよい。パラメータをすべて引き終ったら, HL をさらに1増やし, メインプログラムの戻り番地(#1107)にして再び XTHL でスタックに戻す。最後のリターン命令でこの戻り番地がPCに入りメインプロ

グラムを続行する。スタック操作は間違いが発生しやすいということを認識しておけば、積極的に使い効率的なプログラムが書ける一例である。

(3) 割込み処理

割込みでよく使うリスタート命令 RST も 1 バイトの CALL 命令であるから、上述のようにスタック操作には注意を要する。

割込み許可、禁止命令 EI, DI は、8080 内部のインタラプトイネーブルフリップフロップ (INTE F/F) の操作命令である。EI 命令は、INTE F/F をセットするが、割込みがかかると自動的にリセットされる。だから何度も割込みを許可した場合は、そのつど EI 命令を入れておかねばならない。あるいは割込みサービスルーチンの最後に EI を入れておけば、次の割込みが許可される。

また DI の使い方は、割込みがかかってはいけないところ、たとえば決められた時間幅のパルスを出すプログラム (基礎編、ソフトウェア II 7, 入出力) 等々ときには入れておきたい。

また割込みサービスルーチンの最初と最後の部分にレジスタを図 6 のように PUSH B, POP B などとしセーブしておくといよい。いつの時点で割込みがかかってもバググラウンドジョブのレジスタはこわれない。

その他、まだたくさんの注意事項があるが、紙面の都合上このへんで終える。ともかく、基本さえしっかりつかんでいれば、上記のような間違いは犯さないであろう。

2. 演算処理

2.1 数値演算の基礎

通常のマイクロコンピュータでは、演算命令は加減算とシフト命令しか持っていないので、その他の演算はこれを組合せ、プログラムを作って行わなければならない。三角関数や指数関数等もすべて同様である。

数値演算処理について次のような例題で説明していく。

(例題) 0~1V フルレンジの 8 ビット A/D コンバータに入ってきた 2 つのアナログ信号の電圧比を計算せよ。

図 8 にこのゼネラルフローを示した。図中のサブルーチン A/DC は、A/D コンバータを駆動し、デジタル値をアキュムレータに入れるもので、前回の入力制御のところで説明したのでここでは省略する。

ここでたとえば、2 つの入力アナログ信号を、X が 0.7V, Y が 0.3V とすると、そのデジタル値は、0V が 00₍₁₆₎, 1V が FF₍₁₆₎ とすれば、

$$X = B3_{(16)}, Y = 4D_{(16)}$$

である。次にこれらを使って Y/X の割算を行うわけで

図 8 電圧比を求めるプログラムゼネラルフロー

左肩の数字は図 3 リストのアドレス番号

図 9 DIV プログラムのフローチャート

あるが、前出のリスト例 DIV (図 3) を用いることにしよう。図 9 にこの詳細フローを示した。これは、分子として 16 ビットのデータを B, C レジスタに入れ、分母の 8 ビットを D レジスタに入れてコールすれば、商 8 ビットが C レジスタに、余りが B レジスタに入ってきてくるものである。ただし分母、分子の条件として、分母

$$\begin{array}{r}
 \begin{array}{l} 1101110 \leftarrow \text{商(Cレジスタ)} \\ 01011001 \sqrt{0010011010000000} \leftarrow \text{分子(B,C レジスタ)} \\ \text{分母} \\ \text{(Dレジスタ)} \end{array} \\
 \begin{array}{r}
 1011001 \\
 10000010 \\
 1011001 \\
 10100100 \\
 1011001 \\
 10010110 \\
 1011001 \\
 1111010 \\
 1011001 \\
 1000010
 \end{array} \\
 \text{余り(Bレジスタ)}
 \end{array}$$

図 10 2進数割算(筆算による方法)

分子ともに正数で分母は $7F_{(16)}$ 以下、分子は上位バイトは、分母の $1/2$ より小であることが必要である。

この DIV サブルーチンの動作は図 10 に示す 2 進数の割算の筆算によるものと同じである。分子から分母を引いて、引ければ(負にならなければ)商に 1 を立て、引なければ 0 を立てる。引けた場合は、引いた結果を 1 つ桁上げて再び分母を引き、もし引けない場合は、そのまま桁上げて分母を引く。これを 8 回繰り返すわけである。このプログラムでは引いてみて負になれば直し直している。

1.2 節プログラム上の注意点(1)フラグのところ述べたように、引き算して引けた場合は桁借りがないので C フラグが 0 である。そしてこれがある桁の商としている。逆に負になれば、直し直した結果はオーバーフローを生じるので C フラグが 1 となる。すなわち、商の結果は 0 と 1 が逆転するためプログラムの最後に商を反転している。また桁上げはシフト命令を使い、しかも商のシフトと分子のシフトを一挙に行っている。

上述の 2 つの電圧値の比を求めるのに、このサブルーチンの条件を満たすように、分母の $B3_{(16)}$ を $1/2$ 倍して(右へ 1 ビットシフト) $59_{(16)}$ にし $7F_{(16)}$ 以下にして D レジスタに入れる。分子もこれに合わせ $1/2$ 倍して、B レジスタに $26_{(16)}$ 、C レジスタに $80_{(16)}$ を入れてコールする。そして分子は分母の $1/2$ より小さいという条件を満足しているのでこれでよい。もし $1/2$ 以上の場合は分子を $1/4, 1/8$ と右へシフトして答を逆に 4 倍、8 倍(左へシフト)すればよい。結果として商が $6E_{(16)}$ 、余りが $42_{(16)}$ となる。

さて、16 進の $6E$ とはどういう値なのだろうか。X と Y の値は同じ A/D コンバータで変換されたものであるから、両者同じ重みを持っている。そこでこの計算を小数点を用いて表現すると、A/D コンバータ入力が $1V$ のとき $FF_{(16)}$ あるいは 2 進数では 11111111 、これは約 1.00000000 である。したがって $0.7V$ では 0.1011011 、 $0.3V$ では 0.01001101 となる。これらを書き直すと

1.011011×2^{-1} および 1.001101×2^{-2} である。割算の前に $1/2$ 倍したのでそれぞれ 1.001101×2^{-2} 、 1.001101×2^{-3} で、その割算の結果は $0.1101110 \times 2^{(-3)-(-2)}$ となり、 $0.01101110 (0.6E_{(16)})$ が得られる。あるいは下のように表わして、

$$\begin{array}{r}
 4D_{(16)} \\
 .00100110 \ 10000000 \\
 \underline{.01011001} \\
 59_{(16)} = B_{(16)} \ 3/2
 \end{array} = \frac{.01101110}{6E_{(16)}} \quad (1)$$

であり、結果の小数点の位置も明瞭になって、10 進数で表わせば、 $2^{-2} + 2^{-3} + 2^{-5} + 2^{-6} + 2^{-7} = 0.4296875$ である。ちなみに、実際の値は、 $0.3/0.7 = 0.4285\cdots$ である。

次に精度の検討をしてみよう。一般に、計算機によって数値計算する場合に検討を要することとして⁴⁾、

- (1) データ固有の誤差
- (2) 丸めの誤差
- (3) 打ち切り誤差
- (4) 誤差の伝搬

がある。

(1)は、先程の例でいけば、AD コンバータの分解能が 8 ビットであるため、 $1(V)/2^8 \approx 0.004(V)$ から AD コンバータ出力のデータは $\pm 0.002V$ の絶対誤差を持っている(応用編、データ入力制御(Ⅲ) 3.2 A/D コンバータ参照)。計算機入力のデータにすでに含まれている誤差を固有(inherent)誤差という。

(2)は、計算機の扱えるデータの桁数には限度がある。たとえば 8 ビットどうしの乗算を行うと 16 ビットになり、さらに 16 ビットどうしの乗算を行うと 32 ビットとなる。このように繰り返していくと手に負えない桁数となり、どこかでデータを丸め(roundoff)しなければならない。丸めには四捨五入や切り上げ、切り捨てがあるが、一般の計算機では切り捨てる。

上の例では、商 8 ビット分の演算結果が得られたら計算を終了しているが、実際の商の値は $0.\overbrace{0110111010111}^{6E}$ …と下線部のように続いているのである。下線部が切捨てられたわけで、10 進数表示では約 0.0028 (5 桁目以下四捨五入)に相当する。これが丸めによって生ずる誤差である。

(3)は、今回の例題では出て来ないが、たとえば $\sin x$ の数値演算の場合、テーラー展開を用いて、

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$

により処理することがあるが、 x^9 以下の項の計算を打ち切って(truncate)しまえば、そこに誤差を生じる。

(4)は、データが誤差を持っている場合、演算によって誤差がどのようになっていくかを検討しなければなら

表 1 四則演算と誤差

$x_1 = x_1^* + \Delta x_1$, $x_2 = x_2^* + \Delta x_2$, $y = y^* + \Delta y$ とする. (ただし $*$ は真値, Δ は誤差とし, $\simeq$ は $|\Delta x_1| \ll x_1^*$, $|\Delta x_2| \ll x_2^*$ のとき成立)

	絶対誤差 (Δy)	相対誤差 ($\Delta y/y^*$)
和, 差 ($y = x_1 \pm x_2$)	$\Delta y = \Delta x_1 + \Delta x_2 $	$\Delta y/y^* = \frac{x_1^*}{x_1^* \pm x_2^*} \left \frac{\Delta x_1}{x_1^*} \right + \frac{x_2^*}{x_1^* \pm x_2^*} \left \frac{\Delta x_2}{x_2^*} \right $
積 ($y = x_1 \cdot x_2$)	$\Delta y \simeq x_1^* \Delta x_2 + x_2^* \Delta x_1 $	$\Delta y/y^* \simeq \left \frac{\Delta x_2}{x_2^*} \right + \left \frac{\Delta x_1}{x_1^*} \right $
商 ($y = x_1/x_2$)	$\Delta y \simeq \left \frac{\Delta x_1}{x_2^*} \right + \left \frac{x_1^* \cdot \Delta x_2}{x_2^{*2}} \right $	$\Delta y/y^* \simeq \left \frac{\Delta x_1}{x_1^*} \right + \left \frac{\Delta x_2}{x_2^*} \right $

ない. 四則演算による誤差の伝搬 (propagation) を表 1 に示した.

さて, 先に得られた結果 6E はどれほどの精度を持っているのであろうか, 具体的に検討してみよう.

まず A/D コンバータ出力 X, Y は, 真値を X^*, Y^* とすれば

$$X = X^* \pm 0.002 \quad (2)$$

$$Y = Y^* \pm 0.002 \quad (3)$$

である. そして分母の切捨てを行ったため, サブルーチン DIV の入力分母 X' は, 最悪の場合 LSB 分すなわち 0.004 だけ切捨てられて (切捨てが 0 の場合もある),

$$X' = X - 0.004_{(\max)} = \begin{cases} X^* + 0.002 \\ X^* - 0.006 \end{cases} \quad (4)$$

である. 次に割算による誤差の変動 (伝搬) を検討すると, 表 1 によって,

$$\frac{Y}{X} \simeq \frac{Y^*}{X^*} + \left| \frac{\pm 0.002}{X^*} \right| + \left| \frac{X^* \times \left(\frac{+0.002}{-0.006} \right)}{Y^{*2}} \right| \quad (5)$$

となる. そして最後に商の結果を切り捨てた. (1) 式の商の小数点の位置に留意しながら, 切り捨てによって生じる最大の誤差を見ると, 8 ビットで計算を打ち切ったから 2^{-8} すなわち約 0.004 となる.

結局, 次の (6) 式となる.

$$\frac{Y}{X} \simeq \frac{Y^*}{X^*} + \left| \frac{\pm 0.002}{X^*} \right| + \left| \frac{X^* \times \left(\frac{+0.002}{-0.006} \right)}{Y^{*2}} \right| + \begin{pmatrix} 0 \\ -0.004 \end{pmatrix} \quad (6)$$

具体的に $X=0.7$, $Y=0.3$ を入れて得られる電圧比の取り得る範囲を求めると, $0.4286 \pm \frac{0.0065}{0.0081}$ (小数点以下 4 桁四捨五入) である. 16 進数で表わせば, 6F₍₁₆₎ ~ 6C₍₁₆₎ で, ほぼ LSB 2 ビットは誤差であることがわかった.

より精度を高くとりたい場合には, A/D コンバータの語長を長くしたり, 演算語長を増やしたり (通常はバイト単位で増やし, 倍長, 3 バイト長へ拡張) する必要がある. 実際のプログラミングにおいては, 前述のように厳密な精度の検討をしなくてもよいが, 倍長演算をす

る必要があるかどうか (倍長演算にすると処理時間, メモリ容量が増える欠点がある) ぐらいの目安はつけたいものである.

図 3 のプログラム DIV を倍長演算に変更したい場合は, 原理的には同じであるが分子の 4 バイト, 分母の 2 バイトをメモリ内に入れてサブルーチンに受け渡し (レジスタの数が不足するので), 2 バイト単位で分子と分母を比較し, シフトも 4 バイトで行う. 繰り返し回数は 16 回である.

またこの DIV は負の数は扱えない. もし負の数も扱いたい場合は, まず分母, 分子のおおのの絶対値を取り (0 から負数を引く), このプログラムで正数どうして演算する. 同時に分子, 分母の符号の排他論理和を取り, 結果が 1 なら (符号が異なれば) 演算結果を負に変換する.

ところで, アナログ信号を測定する場合に, その信号の S/N が悪く精度良く測ることができないことがある. こうしたときに有効な手段として平均化処理がある. A/D 変換したデータを積算して, 積算回数で割ると測定精度が上がる. 必要な精度を σ_L , 入力検出誤差を σ_s とすれば, 必要な積算回数 N は次式で与えられる.

$$N = \left(\frac{\sigma_s}{\sigma_L} \right)^2 \quad (7)$$

通常は N を 2 の倍数にするとよい. 何故なら積算回数で割算するときにシフト命令だけですむからである.

2.2 浮動小数点

先に述べた例題の答 6E₍₁₆₎ は, C レジスタの中に入れてサブルーチンから戻って来た. しかし実際は 0.6E₍₁₆₎ であることを述べた. また別の例題の時は 4 ビット小数点がずれて 6.E₍₁₆₎ かも知れない. レジスタやメモリの中の数値は単なる数字のら列でしかなく, 使用者が適当に仮想の小数点を決めて位取りをしていた. このような数値の表現を固定小数点方式 (fixed point) という. 固定小数点方式では, 前述のような割算 1 回だけの処理ならまだなんとか見当をつけることができたが, より複雑

な計算になると、この位取りの作業はかなりやっかいになってくる。そこで、この位取りの作業を楽にしようと工夫したのが浮動小数点方式である。すなわちある数値 N を、

$$n = m \times b^x \quad (x \text{ は整数}) \quad (8)$$

の形で表現するもので、 m を仮数部 (mantissa)、 x を指数部 (exponent) という。 b は底 (base) と呼び、これを 2 とし、 m に (10) 式のような条件をつけると、 n は正規化 (normalize) 数という。

$$n = m \times 2^x \quad (x: \text{整数}) \quad (9)$$

$$n \text{ が } \begin{cases} \text{正のとき} & \frac{1}{2} \leq m < 1 \\ \text{負のとき} & -\frac{1}{2} > m \geq -1 \end{cases} \quad (10)$$

このような表現の例を図 11 に示した。仮数部、指数部および符号を指定すれば実数を表現することができる。演算の入出力において常にこのような表現をしておけば、位取りの作業がずっと楽になることは明らかである。

浮動小数点形式の計算機内部での表現形式は、種々のものがあるが、例えば図 12 のようなものがある。1つのデータを表現するのに 4 バイトで構成されており、1 バイトの指数部と 3 バイトの仮数部で表現され、図 13 のようにそれぞれ重みづけされる。指数部、仮数部はともに 2 の補数表示である。図 13 に 10 進数に変換する

$$\begin{aligned} -1101.1011 &\rightarrow -(0.11011011) \times 2^4 \\ +0.00011001 &\rightarrow +(0.11001) \times 2^{-3} \end{aligned}$$

↑
符号 仮数部 指数部

図 11 固定小数点と浮動小数点の例

図 12 4 語浮動小数点形式のデータ構成

上のような場合の 10 進数表現

指数部: 2 の補数をとると 00001010 となり -10 (10 進) である

仮数部: 2 の補数をとると 010010011 となり $-(2^{-1} + 2^{-4} + 2^{-7} + 2^{-8}) = -0.57421875$ したがって $-0.57421875 \times 2^{-10} = -0.5607605 \times 10^{-3}$

図 13 図 12 のデータ構成の場合の重みづけと数値例

方法の一例を示した。

この形式で表わされる数値の許容範囲は、約 $\pm(2^{127} \sim 2^{-128})$ すなわち $\pm(1.701412 \times 10^{38} \sim 2.938735 \times 10^{-39})$ である。これぐらいの範囲の数値が扱えれば、通常の演算には充分である。一方、固定小数点方式では、4 バイト分を使えば約 $\pm 1.34 \times 10^8$ の数値許容範囲であるので、それこそ桁違いにその範囲は違う。ただし、有効桁を比較すると、浮動小数点方式では指数部に 1 バイト分とられるので、この点では固定小数点方式の方が優っている。

なお、浮動小数点の内部データ表現の仕方として、指数部を最下位バイトとする場合もある。また仮数部は (10) 式のように正規化されるので、 2^{-1} の項は常に 1 になることがわかっている。したがってこの 2^{-1} 項を省略する表現形式もある。

通常 A/D コンバータ等の外部から入って来るデータは、2 進整数形である場合が多い。これを浮動小数点に変換する方法について述べてみよう。図 14 にそのフローチャートを示した。これは整数を正規化することによって他ならない。すなわち整数が正であれば $1/2$ から 1 の間に、負なら $-1/2$ から -1 の間に入るまで整数を右にシフトし (2 の倍数で割り)、シフトした回数により指数部を変更することである。3 バイトの整数を例にとって説

図 14 2 進整数を浮動小数点に変換するためのフローチャート

明しよう。いま、与えられた整数が下のようであれば、

これは 0.1×2^{23} であることはわかるであろう。もし 1 の位置が右に 1 ビットずれていれば 0.1×2^{22} であり、2 ビットずれていれば 0.1×2^{21} である。一般的に MSB の次のビットから右へ数えて x 個目に 1 が初めて出て来たとすれば、指数部は $23-x$ で表わされることになる。この考え方が図 14 のフローチャートであり、最初の指数部の初期値は、3 バイトの整数のときは $23_{(10)}$ をセットすればよいことになる。このように整数を正規化して浮動小数点に変換することができた。この正規化の原理が浮動小数点演算の基本となる。

さて、次に浮動小数点どうしの演算について述べよう。高級なコンピュータにおいては、浮動小数点の演算を専用の演算回路で行うものがあるが、一般のマイクロコンピュータでは、このようなハードウェアを備えていることはまれで、ソフトウェアで行うのが普通である。浮動小数点演算のプログラムは、固定小数点演算のものにくらべ相当ステップ数が多くなる。たとえば単なる加算でさえ百数ステップに及び、その実行時間は $1 \sim 2 \text{ ms}$ とかなり遅くなる。位取りで頭を悩まさないですむかわりに、このような欠点があることを覚悟しておかねばならない。

浮動小数点による四則演算の概略⁵⁾をここで述べることにし、より詳しくは末尾の参考文献⁶⁾⁷⁾を参考にされたい。

2 つの正規化された浮動小数点を $n_1 = m_1 \times b^{x_1}$, $n_2 = m_2 \times b^{x_2}$ とすると、

(1) 乗除算

乗算は

$$n = n_1 \times n_2 = (m_1 \times m_2) \cdot b^{x_1+x_2}$$

であり、除算は

$$n = n_1/n_2 = (m_1/m_2) \cdot b^{x_1-x_2}$$

であるから、

- (i) 仮数部どうしを固定小数点方式の乗除算で演算を行う。
- (ii) 指数部を加減する。
- (iii) 仮数部を正規化し、指数部を補正する。

(2) 加減算

実は浮動小数点の場合、乗除算より加減算の方がめん

どうなのである。加減算は、

$$\begin{aligned} n &= n_1 \pm n_2 \\ &= m_1 \cdot b^{x_1} \pm m_2 \cdot b^{x_2} \\ &= (m_1 \pm m_2 b^{x_2-x_1}) b^{x_1} \quad (x_1 \geq x_2) \end{aligned}$$

で与えられるため、桁合せをする必要がある。すなわち、

- (i) 指数部を比較し、小さい方が大きい方に合うまで仮数部の桁をずらしていく。上の式は $x_1 \geq x_2$ の場合で、 $(x_2 - x_1)$ だけ左へずらしていく。
- (ii) 仮数部の加減算を行う。
- (iii) 指数部に、指数部の大きい方を、上の例では x_1 を入れる。
- (iv) 仮数部を正規化し、指数部を補正する。

仮数部の演算の結果 ± 1 を越えてしまう場合があるが、正規化の手順と同様これを修正する必要がある。また指数部がその許容範囲を越えた場合は、大きすぎるとしてオーバーフロー（図 12 の場合 $\pm 2^{+127}$ 以上）あるいは小さすぎるとしてアンダーフロー（図 12 では $\pm 2^{-128}$ 以下）として演算を終了し、その由を表示するような手段が必要である。

なお 0 を浮動小数点で表わす場合は、正規することができないので、正規化数の正の数でとりうる最小値すなわち、仮数部を 0、仮数符号を正、指数部を最小値にしたもので代用する場合がある。

〔参考文献〕

- 1) Adam Osborne: 8080 Programming For Logic Design
- 2) 渡辺和夫: 私のプログラミング, インターフェース, 8 (1976) 140
- 3) 実用マイコンプログラム集, インターフェース, 9 (1977) 33-43
- 4) Daniel D. McRacken: Numerical Methods and Fortran Programing, Wiley International Edition
- 5) 高橋茂: デジタル電子計算機, 日刊工業新聞社
- 6) 田辺皓正: 数値演算プログラム, 電子科学, 9 (1976) 39-46
- 7) そのまま使えるサブルーチン集, 電子科学, 9 (1977), 綴込付録

用語集

絶対誤差

これに対し相対誤差がある。近似値を x , 真値を x^* とすると絶対誤差 Δx は, $\Delta x = x - x^*$ で表わされ、誤差そのものをいう。一方、相対誤差は $\Delta x/x^*$ で表わされ、これを 100 倍したものをパーセント誤差という。