資　料

マイクロコンピュータの
ソフトウェアの基礎

昭和59年3月

JIPDEC

財団法人　日本情報処理開発協会

この資料は，日本自転車振興会から競輪収益の一部である機械工業振興資金の補助を受けて，昭和５８年度に実施した「マイクロコンピュータの応用に関する調査研究」の一環としてとりまとめたものです。

序

マイクロコンピュータの出現以来はやくも十数年を経過したいま、その浸透はおそるべきものがある。当初はやみくもにその応用を考えたが、いまでは自然に、必然性を持って各種の機器に内蔵される型で使われるようになった。

世の中ではパーソナルコンピュータやワードプロセッサの方が人目につきやすいから、マイクロコンピュータの応用といえばそういったものを考えがちである。実際には、それらへの応用は数量比で言えばたいしたことはなく、大部分のマイクロコンピュータは人目に触れにくいかたちで機器のなかに内蔵されている。

こういったマイクロコンピュータ応用システムは、いまではほとんどあらゆる領域に進出しており、先端的な新製品ならば、ごく当然のようにマイクロコンピュータをいくつか埋め込んでいる。

問題はその種のシステムを開発する技術者が圧倒的に不足していることである。マイクロコンピュータがこのように普及し出してからわずか十数年であるから、技術者の不足は当然といえば当然である。しかし、だからといって放置しておくわけには行かない。マイクロコンピュータ応用技術はいまや日本の産業にとってもっとも重要な基盤になっているからである。国際競争力から見ても勿論、正しい技術にもとづいたシステムが出来なければ、その応用範囲がひろいだけに、社会にとって危険でさえある。人命に影響する場合も多い。正しい技術を持つマイクロコンピュータ応用システム技術者の育成はわが国にとって、地味ながら極めて重要な基本課題である。

本書はマイクロコンピュータ応用システム技術のためのカリキュラムを詳細化したものである。範囲はソフトウェア技術に限定してある。ハードウェアについては前年度、本協会より「マイクロコンピュータのハードウェアの基礎」という題名で教科書が刊行されている。これでハードウェアとソフトウェアと

が揃ったわけだが、両者を統合するシステム化技術が欠けている。機会があれば刊行されることを望みたい。

マイコン応用システムのソフトウェアの標準的な教材作りをめざして本書が作られた。この分野では類書がほとんどない。しかも技術そのものの進歩が早い。本書は相応の役割を果たし得ると思うが、わが国の標準教材となるためには広い範囲の読者からのご意見がぜひとも必要である。本書は読者からのご意見をいただくためのタタキ台である。是非忌憚のないご批判をお寄せいただきたい。それによって本書、つまりカリキュラムをよりよいものに育てて行きたいと考える。

なお、本書の作成にあたっては前田英明氏のご尽力が大きい。ここに謝意を表したい。

昭和５９年３月

マイクロコンピュータ基本問題委員会委員長
（電子技術総合研究所ソフトウェア部数理情報研究室長）
田　村　浩一郎

目　　次

第1部　システム開発のステップ

第1部　システム開発のステップ

〝９０円のＩＣが１個故障したためにシステムが誤動作をした。〟とニュースの種になった米国の防衛システムは１９６０年代の初期に開発されたシステムである。

このシステムが設置されて以来今日まで２０数年にわたって使用されている。

同じようにマイクロプロセッサを組み込んだ機器が設置された場合には、マイクロプロセッサを含む機械とこれで使用されているソフトウェアはこの機器が使用されている限り存在するわけである。

電子計算機が生まれてから今日まで約３０年、そしてマイクロプロセッサが生まれてからまだ１０年にも満たない。

このように歴史が浅い時代にはハードウェア、ソフトウェアを作るということに重点が置かれていた。しかし、システムの寿命が長くなってくるとシステムを拡張していかに長く使用することが出来るかということが大きな問題となってくる。

そこでシステムのライフ・サイクルに関心が持たれるようになって来た。

ここではシステムのライフ・サイクルを中心にして〝システムの開発〟をおこなう時にはどのような仕事が発生し、これをどのような手順でおこなっているか説明する。

1.　システムに対する要求の分析

客先がどのようなシステムが完成することを期待しているか検討する。

2.　システムの分析

客先が要求しているシステムがどこまで可能であるか検討する。

また、現在の手持ちの技術でどこまで可能か、またこのシステムを完成す

るためにはどのような新しい技術が必要であるか検討する。

この作業によって、システムの概要が完成する。

3. システムの定義

このシステムが持たなくてはならない機能（ベース・ライン）を設定する。

4. システムの設計

3.のシステムの定義で決定した機能を満足するシステムを開発するためにはどのようなハードウェア・システムが必要であるか、またソフトウェアが必要であるか検討する。この結果ハードウェア・システム仕様書、ソフトウェア仕様書が作成される。

以下の作業は、この作業によって作られた仕様書に基ずいて作業がおこなわれる。

5. システムの開発

この段階でハードウェアの開発とソフトウェアの開発に分かれて作業がおこなわれる。

ソフトウェアの開発では、4.の作業で作成されたソフトウェア作業書にもとずいて次のような作業をおこなう。

a) 機能別プログラム・モジュールの設計

ソフトウェア仕様書に述べられているソフトウェアを完成するために、これを機能別にいくつかのモジュールに分解する。

それぞれのモジュールについて仕様書を作成する。大きなプロジェクトの場合には1つのモジュールが更にいくつかのサブ・モジュールに分解される。

モジュール仕様書には、次のような項目が記載されている。

i) モジュールの目的

ii) 入力となるデータ、出力となるデータについて、名前、データ型、桁数などの必要な項目

iii) 処理方式（計算式や結果の精度など）

iv) 異常データの処理方法

v) 他のモジュールとの関連

b) プログラミング

先のa) で作られた機能別プログラム・モジュール仕様書にもとずいて作業をおこなう。

この作業では、仕様書に述べられている内容をプログラミング言語を使用して記述する。

c) プログラム・テスト

b）で作成されたプログラムをコンパイルし、テスト・データを使用してテストする。

この作業をくり返して、要求されている機能を満足するプログラムを作る。

この作業を一般に単体テスト（Unit test）と呼んでいる。

プログラムがいくつかのモジュールあるいはサブモジュールに分かれている場合には、いくつかの完成したモジュールを更に組み合わせてテストを続けていく。

次第にテストの対象となるモジュールは大きくなり、最後には1つのモジュールとなる。この1つのモジュールになったプログラムのテストは完成したハードウェアを利用しておこなうので一般には総合テスト（Integrated Test）と呼んでいる。

このテストは工場においておこなう場合が多いが、既設の機器を使用しなくてはならない場合には現地でおこなうことがある。

6. 設　置

完成したシステム（ハードウェア／ソフトウェア）を現地（客先）に設置し、開発者が用意した設置テスト、客先が用意した受入れテストを経て運用となる。

7. 保　守

設置されたあとに非常に長い作業が始まる。例えばシステムが設置されたあと現在の測定点を２０点から５０点に変更するという問題が発生する。このような問題が発生した場合にはハードウェアは勿論のことソフトウェアも変更しなくてはならない。この作業のことを保守と呼んでいる。

この作業はシステムが設置されている限り続く。したがって保守しやすいプログラムを作るということが重要である。

マイクロコンピュータ技術者育成コースの初級篇では5.のシステムの開発から7.の保守まで、あるいは4.のシステムの設計の一部にメンバーとして参加することが出来る技術者を育成することに目標をおいている。

尚、上級篇では1.の要求分析から4.のシステム開発までの作業に従事することが出来る技術者を育成することに努めている。

第2部　プログラミングの基礎

第2部　プログラミングの基礎

ＢＡＳＩＣで作られたプログラムを例に引くまでもなく、プログラミングを独自に学習した人は目的の結果が出るようにプログラムをこじつけてしまう傾向が強い。

このために、出来上がったプログラムは他人にとっても読み難いものであることは勿論のこと、たとえプログラムを作成した本人であってもこのプログラムに新しい機能を追加することが困難である。

プログラミングは過去に作成したプログラム、即ち経験をもとに発展させていくので、最初のプログラミングが以降のプログラミングに大きな影響を及ぼす。

したがって、最初のプログラミングに於いて系統立ったプログラミングの方法をきちんと学習させる必要がある。

このためにプログラミングの方法を指導するに当っては構造化プログラミングの考え方を導入する必要がある。

プログラミングがまったく初めての人にとっては構造化プログラミングも難なく受け入れることが出来るので問題は起きない。

しかし、過去にプログラミングについて経験を持っている人は構造化プログラミングを受け入れ難いものにする傾向がある。

プログラミングの指導に当ってはプログラムそのものも重要であるが、プログラムのテストの方法も併せて指導する必要がある。

構造化したプログラムは構造化しないプログラムよりもテストがおこない易いということを実証する。

このことによって、これ迄構造化プログラミングに対して疑問を持っていた人も自然と構造化プログラミングに対してなじんで来るので、マイクロコンピュータが普及した今日ではプログラミングの教育に当って〝プログラミングを

独学した人がかなり存在する”ということを充分考慮しないといけない。

構造化してないプログラムがあった場合には、プログラムを作成している途中でこの点を指摘して強制的に構造化させるよりも、出来上がった時点でプログラムのテストを通じて構造化プログラムの利点を徹した方が効果的である。

マイクロコンピュータのソフトウェア開発では、アセンブリ言語、ＰＬ／Ｍといったプログラミング言語が汎く使用されている。最近ではＣというプログラミング言語が使用されるようになって来た。

これらのプログラミング言語はいずれもアセンブリ言語に近い性格を持っているので、これらのプログラミング言語は使用する人の技量によって出来上がるプログラムが大きく変わって来る。

また、ＰＬ／Ｍは使用する機種によって利用できる機能が異なる。Ｃという言語は初心者にとってまぎらわしい記述法が色々と存在するなどの理由からこれらのプログラミング言語はプログラミングの基礎を学習する際に適した言語ではないという立場を取った。

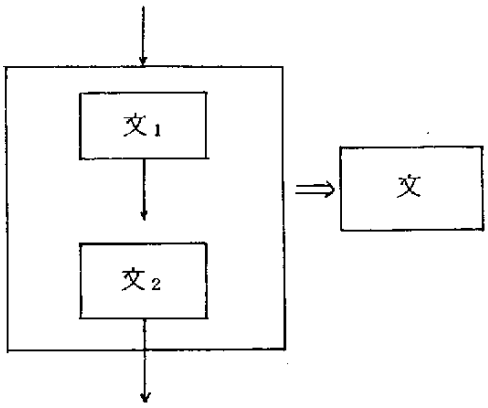
したがって、プログラミングの基礎を学習するに当ってはPascal を採用することにしたが、本コースはPascal を学習するためではなく、Pascal を通じてプログラムの作り方を修得することを目的としている。

Pascal をプログラミングの学習用言語とすると往々にしてPascal 風の例題を採用しがちになるが、本コースの目的はPascal を通じてプログラムの作り方、即ちプログラミングの方法を学習することにあるので、例題もマイクロコンピュータのプログラミングに於いて発生する問題を採用するように配慮した。

第１章　コースの進め方

問題を提示する場合には、構造化プログラミングの５つの要素を使用し、平文で説明をおこなうことが望ましい。

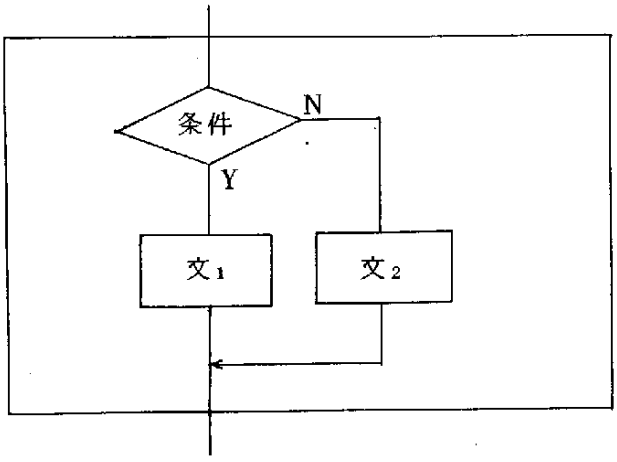
1)　sequence



文１を実行したあと文１のうしろにある文２を実行する。

文１と文２は組み合わさって文と等価になる。

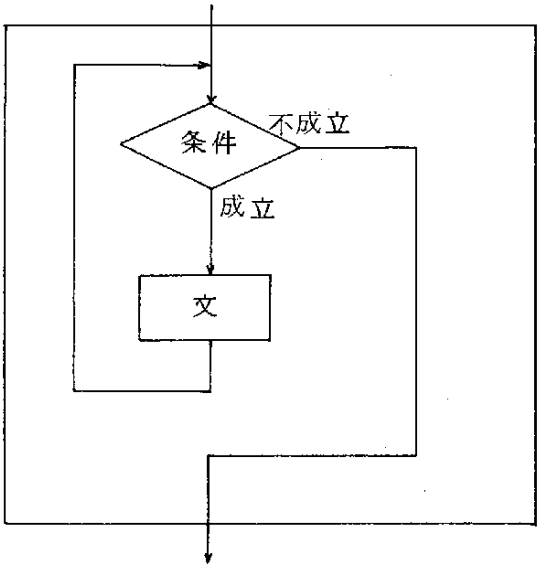
2)　if条件 then文 else文



条件が成立した時には
then のうしろにある文を
条件が成立しない時には
else のうしろにある文を
実行する。

if ～ then ～else は sequence 中の文と等価になる。

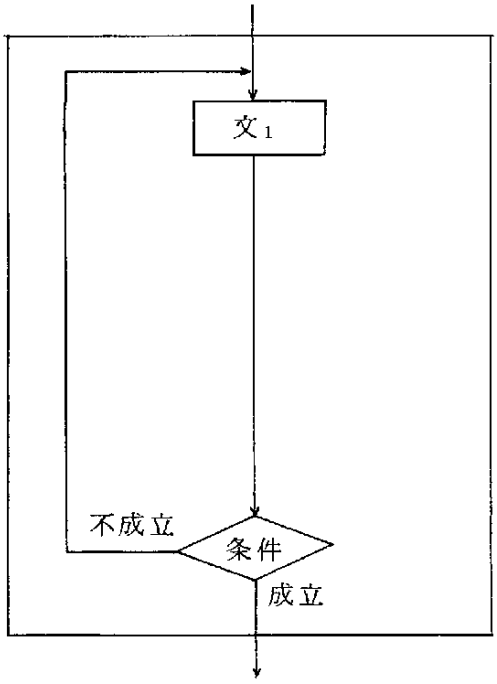
3) while 条件 do 文



条件が成立している間、文を実行する。

while ～do ～文は sequence 中の文と等価になる。

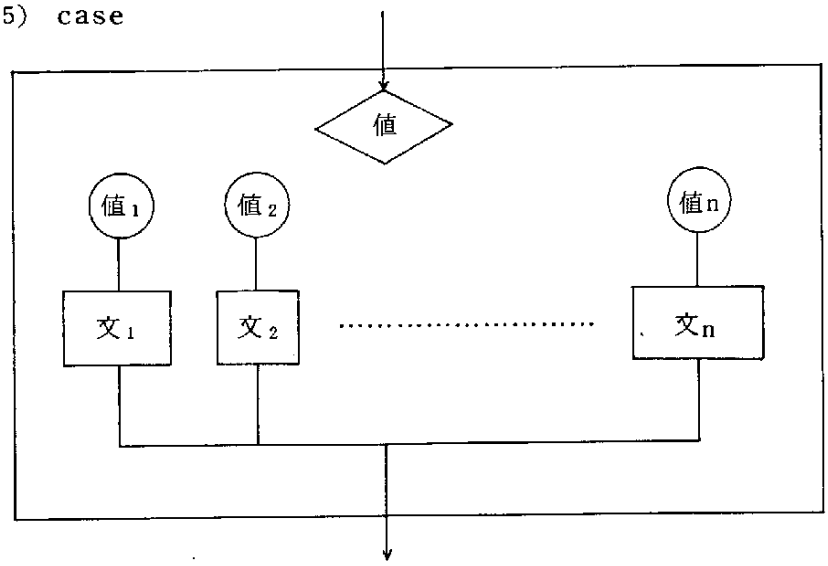
4) repeat 文 until 条件



条件が成立するまで文を繰返し実行する。

repeat ～until ～文は sequence 中の文と等価になる。

5) case



式の値によって値に対応する文を実行する。

case 文は sequence 中の文と等価になる。

第2章　プログラミング

2.1　簡単なプログラム

目的：メッセージを出力するプログラムを通じてPascalの構文図の読み方、ブロックの考え方を修得する。

writelnという関数を利用してメッセージを1行出力する。

このプログラムが完成したら数行にわたるメッセージを出力する。

例

```
program sample 1;
begin
      writeln ('This is first line');
      writeln ('This is second line');
      writeln ('This is third line')
end.
```

2.2　繰返しがあるプログラム

目的：for 文、while 文、repeat 文を利用して繰返し実行の方法を学習すると共に初期値の設定の方法を学ぶ。

併せてデータの型についても学習する。

2.1で学習した数行のメッセージを出力するプログラムを基礎にし、このメッセージを繰返して出力する。

例

```
program sample 2;
var  count : integer;
begin
      count := 1;
```

```
        while count <= 10 do
        begin
          writeln ('1st line') ;
          writeln ('2nd line') ;
          count := count + 1
        end
    end.
```

例

```
program sample 3 ;
var   count : integer;
begin
for count := 0 to 9 do
begin
  writeln ('this is message') ;
  writeln ('this is 2nd line')
  end
end.
```

例

```
program sample 4;
var   count : integer;
begin
    for 10 downto 1 do
    writeln ('this is pascal program')
end.
```

2.3　摂氏から華氏への温度変換

目的：摂氏から華氏への温度変換のプログラムを通して、値によっては正しく動かないプログラムがあるということを修得する。

最初に摂氏０度より１０℃おきに１００℃まで華氏の温度に変換するプログラムを作成する。

例

```
program sample 5;
var seshi, kashi : integer;
begin
seshi := 0;
while seshi <= 100 do
begin
kashi := seshi * 9 div 5 + 32;
writeln ( seshi, '  =  ', kashi ) ;
seshi := seshi + 10
end
end.
```

このプログラムが完成したら、constを使用して定義を記号化する。

定数を記号化することによってプログラムの抽象化を教える。

このプログラムの下限、上限そして刻みをコンソールから入力するように改善する。出来上がったプログラムに色々な値を入れて見てこのプログラムが正しく動くか確認する。下限＜上限で刻みがマイナスの時、下限＞上限で刻みがプラスの時にはプログラムが止まらなくなるので、どちらかの場合に限定し、どのような値を入力してもプログラムが正常に終了するように作らせる。

例

```
program sample 6;
var  clow, chigh, step, f;
begin
  repeat
    write ('enter kagen'); readln (clow);
    write ('enter jogen'); readln (chigh);
    write ('enter kizami'); readln (step);
    until (clow>0 && chigh>0 && step>0);
    while (clow<=chigh) do
  begin
    f = clow * 9 div 5 + 32;
    writeln (clow, '  =' , f);
    clow := clow + step
  end
end.
```

出来上がったプログラムに値を入れて、このプログラムが正しく動くかどうかチェックをさせる。

2.4 ニュートン法による平方根の計算

目的：ニュートン法による平方根の計算を通じて計算誤差があるということを修得させる。

ニュートン法でAの平方根を求める式

$Xn = (Xn-1 + A/Xn-1) * 0.5$

Xn−1は前回の値、Xnは今回の値

最初Xn−1を(A+2)/3とおく。

例

```
begin
  x1＝(a＋2.0)／3.0；
  repeat
  x＝(x1＋a／x1)＊0.5；
  x1＝x；
  until  x1＊x1＝a；
  writeln('SQUARE ROOT OF', a, ' =' , x1);
end.
```

上記のプログラムを実行させると無限ループに入る。

このプログラムを改造し、毎回ｘ１の２乗の値を出力させ、計算誤差があるのでｘ１＊ｘ１≒ａであるということを認識させる。

またａの値にマイナスの値を入れた時にプログラムがどのように動くか確認する。

プログラムが正しく動くためにはそれぞれの変数がどのような値でなくてはならないか考えさせる。

出来上がったら誤差の範囲を色々変えて数表の値と比較して見る。

2.5　case文による偶数、奇数の判定と１６進数

目　的：　case文を利用し、１６進数で使用している文字を学習する。

キーボードより０から９までの１文字を入力し、この数字が偶数であるか奇数であるか判定する。

次に２進法、８進法、１６進法の関係を説明し、１６進法では各桁を表すのにどのような記号を使用しているか説明する。

この説明が済んだあと、コンソールから０から９、ＡからＦまでの１文字を入力し、この１６進数に相当する１０進数を出力する。

このプログラムで０から７ＦＦＦまでの値を入力する。

0から32767までの値が求まったら次に8000を入力すると−32768という値が出力されるので、ここで整数が計算機内部ではどのように表現されているか説明する。

これによって補数（Complement）を説明することが出来る。

2.6　16進数を10進数に変換する

目　的　：　10進数と16進数（2進数）の関係を学習する。Pascalでは16進数を入力することが出来ないので16進数は文字で入力するということを修得させる。

まず

$$a_1 \times 10^m + a_2 \times 10^{m-1} + \cdots\cdots + a_m \times 10^0 =$$

$$b_1 \times 16^n + b_2 \times 16^{n-1} + \cdots\cdots + b_m \times 16^0$$

という関数を教える。

次いで

$$b_1 \times 16^3 + b_2 \times 16^2 + b_3 \times 16^1 + b_4 \times 16^0 =$$

$$(((0 + b_1) \times 16 + b_2) \times 16 + b_3) \times 16 + b_4$$

が成立するということを教える。

次いでＡＳＣＩＩ文字のコレーティング・シーケンスを教えＯＲＤ（X）という関数の働らきをはっきりとさせる。

1桁の16進数（ＡＳＣＩＩ）文字を10進数に直す。

```
if ch > '9' then x = ord(ch) − ord('A') + 10
           else x = ord(ch) − ord('0');
```

ord('0')＝48、ord('A')は65である。

次いで4桁の16進数を10進数に変えるプログラムを教える。

例

```
program sample 7;
var
value:integer;  count:integer;  ch:char;
x:integer;
begin
value:=10;
for count:=1 to 4 do
begin
read(ch);
if ch>'9' then x:=ord(ch)-ord('A')-10
          else x:=ord(ch)-ord('0');
value:=value*16+x;
end;
writeln(value)
end.
```

2.7　16進数（2進数）から10進数への変換

目　的：16進数から10進数への変換方法を教える。ただし8000はマイナスの値となっているので、0から7FFFまでの数に限定する。

例

```
program sample8 ;
var
value:integer;  x:integer;  const:integer;
begin
  const:=4096;
  repeat
```

```
    x := value div const ;
    if x > 9 then x := x + 7 ;
    write ( chr ( x )) ;
    value := value mod const ;
    const := const div 16
    until const = 0 ;
    writeln ;
  end.
```

value の値を求めるために2.6で作成したプログラムを組み合わせると良いであろう。

１６進数８０００以上の値を取扱う場合には工夫が必要である。

即ち、－１（ＦＦＦＦ）から－３２７６８（８０００）は３２７６７から０までの値に変えなくてはならない。

この時、３２７６８＋ｘという計算が出来ないので、３２７６７＋ｘ＋１としなくてはならない。

例

```
  program sample9 ;
  var
  value : integer ;  x : integer ;  const : integer ;
  corr : integer ;
  const := 4096 ;
  if value < 0 then begin value := 32767 + value + 1 ;
                          corr := 8   end
               else corr := 0 ;
  repeat
    x := value div const ;
    value := value mod const ;
```

```
    if x>9 then x:=x+7;
    write(chr(x+corr));
    corr:=0;
    const:=const div 16
    until const=0;
    writeln;
  end.
```

2.8 関　数

目　的　：問題の抽象化と関係を学習する。学習するに当っては2.6、2.7で使用した問題を利用する。

例

```
  begin
  １０進数が入る場所を０にする。
  繰返しの制御変数ｉを１にする。
  while 繰返しの制御変数が４以内の時
    コンソールより１文字入力する。
    １０進相当数を１６倍した値に今読み込んだ文字を変換した値を加
    える。
    制御変数を１進ませる。
    while－end
    １０進相当数を出力する。
  end

  begin
  value:=0;
  i:=1;
```

```
while i <= 4 do
begin
  read(ch);
  value := value * 16 + aschex(ch);
  i := i + 1
  end;
  writeln(value)
end.
```

一方入力されたASCII文字をこれに対応する16進数に変換する関数aschexは次のように定義される。

```
function aschex(x : char) : integer;
begin
if x > '9' then aschex := ord(x) - ord('A' + 10)
          else aschex := ord(x) - ord('0')
end;
```

キーボードより4桁の16進数相当の文字を読む場合には次のようになる。

```
function hex1byte; integer;
var ch : char;
begin
  read(ch);
  if ch > '9' then hex1byte := ord(x) - 55
              else hex1byte := ord(x) - 48
end;
function hex2byte : integer;
begin
hex2byte := hex1byte * 16 + hex1byte
```

```
end;
function hex4byte:integer;
begin
hex4byte:=hex2byte*256+hex2byte
end;
```

Pascalでは関数を入れ子にすることができるので次のようにプログラミングできる。

```
function hex4byte:integer;
function hex2byte:integer;
function hex1byte:integer;
var
ch:char
begin
  read(ch);
  if ch>'9' then hex1byte:=ord(ch)-55
            else hex1byte:=ord(ch)-48
end (* of hex1byte *);
begin
  hex2byte:=hex1byte*16+hex1byte
end (* of hex2byte *)
begin
  hex4byte:=hex2byte*256+hex2byte
end;
```

しかし、アセンブリ言語のプログラミングではサブルーチンを入れ子に書くことが出来ないので、Pascalでも関数は入れ子にしない方がよいであろう。

2.9 手 続 き

目 的 ： １６進数を１０進数に変換するプログラムを通じて手続の方法について学習する。

```
procedure hex1asc(x : integer);
begin
  if x>9  then write(chr(x+55))
          else write(chr(x+48))
end;
procedure hex2asc(x : integer);
begin
  hex1asc(x div 16);
  hex1asc(x mod 16)
end;
procedure hex4asc(x : integer);
begin
  hex2asc(x div 256);
  hex2asc(x mod 256)
end;
```

2.10 エラー処理

目 的 ： エラー処理の仕方について学習する。

これまでの１６進数から１０進数への変換プログラムでは０から９、ＡからＦまでの文字しか入力されないという前提でプログラムを作成して来た。

このために０から９、ＡからＦ以外の文字が入力された時には適当な値に変換して処理をおこなってしまう。

このような文字が入って来た時にはこの文字を処理しないと共に、エラ

ーがあったことを知らせなくてはならない。

a) １６進数の場合に関数は０から１５までの値しか返さないので、エラーがあった場合にはこれ以外の値、例えば－１を返す。

b) 関数からは０から１５までの値を返し、特定の変数に状態を示す値を返す。

c) 関数からは状態を返し、特定の場所に変換した値を返す。

上記a)、b)、c)のうち、a)かc)の方法を採用するのが良いだろう。

例

１６進数を示す文字以外の文字が入って来た時には変換を止める。

```
begin
value:=0;   count:=1;
ch:=hex1byte;
while(ch>=0   and   count<=4) do
begin
  value:=value*16+ch;
  count:=count+1;
  ch:=hex1byte;
  end;
  if ch<0 then writeln('error')
          else writeln( value )
end.
function hex1byte:integer;
var
ch:char;
begin
read(ch);
if(ch>='0' and ch<='9')
```

```
        then hex1byte := ord(ch) - 48
    else if (ch >= 'A' and ch <= 'F')
            then hex1byte := ord(ch) - 55
            else hex1byte := -1
end;
```

2.11 偶数パリティ、奇数パリティ・ビットを生成する関数

目　的 ： 偶数パリティ、奇数パリティの意味、パリティ・ビットの生成法について学習する。

0から１２７までの値を受け取り、これから偶数パリティ・ビット（あるいは奇数パリティ・ビットが付いた値を返す関数を学習する。

2.12 配列の処理

目　的 ： Pascalでは文字列は文字データの配列として取扱うので、文字列データの処理を通して配列データの処理を学習する。

2.6、2.7のプログラムではコンソールから１文字入力をおこない、結果を１文字ずつコンソールに出力していたが、コンソールより配列に入力し、配列の中にあるデータを使用して１６進数から１０進数を作る。

更に１０進数から１６進数に変換し、変換した文字を配列に入れる。

出来上がった文字列を配列からコンソールに出力するようにしバッファ（配列）という考え方を導入する。

なお、それぞれの処理を関数化、手続き化し、プログラムの組み立て方を学習させる。

2.13 多倍精度演算

目　的 ： 配列を利用して多倍精度演算の方法を学習する。

１６ビット整数の場合には＋３２７６７までの値、８ビット整数の場合

には２５５までしか取扱うことが出来ない。

これより大きい整数を取扱う場合にはデータを４桁（あるいは２桁）毎に区切り、配列を利用して処理しなくてはならない。

配列を利用して多倍精度演算の方法を学習したあと、各桁をＡＳＣＩＩ文字で記憶し、ＡＳＣＩＩコードのままで演算する方法を学習する。

2.14 逆ポーランド記述による電卓プログラム

目　的：スタックとスタック・ポインタについて学習する。

逆ポーランド記述について説明をおこなうと共に、逆ポーランド記述されたデータをオペランド、オペランド、オペレータの対になるまでオペランドをオペランド・スタックに積んでおく。

この対になったらオペランド・スタックの上部より２つのオペランドを取り出し、指定されたオペレーションをおこなったあと結果をオペランド・スタックに返す。

利用できるオペレータを＋、－、＊、／だけではなく、余り、ＡＮＤ、ＯＲ、ＸＯＲと次第に増加していく。

2.15 通常の記述より逆ポーランド記述に変換

コンソールより通常使用しているかっこを含む式（ただし変数は取扱わない）を入力し、この式を逆ポーランド記述の式に変換する。

例

（２＋３）＊（４＋５）

は

２３＋４５＋＊

このテーマは省略することができるが、しかし2.13のテーマと併せて学習するとコンパイラが数式をどのように処理しているか理解する助けになる。

学習者が興味を引いた場合には自由書式で入力した一行の式を処理する電卓プログラムに発展させると良い。

2.16 メモリ・ダンプ・プログラム

目　的：配列を利用して１６進メモリ・ダンプと１６進ローダを学習する。

メモリ・アドレスをポインタとして参照することが出来るPascalコンパイラが利用出来る場合にはメモリの内容を１６進ダンプする。

メモリ・アドレスを参照することが出来ないPascalコンパイラの場合には配列をメモリに見立ててメモリの内容を１６進ダンプする。

配列をメモリの代わりに使用する場合には配列の内容は0で初期化されているので、配列の中に前もってデータを入力しておく必要がある。

このために次のコマンドを持ったモニタを用意する。

S（Substitute）命令

S変更を開始するアドレス↵

指定されたアドレスの内容１バイトを１６進数で表示する。

a) 変更をおこなう時にはコンソールより１６進２文字とリターンを入力し、この数値を現在対象となっているアドレスに書く。

b) 変更を必要としない場合にはリターンを入力する。

a)、b)いずれの場合にもアドレスは１進み、処理を繰返す。

c) .とリターンが入力された時にはS命令の処理を終了してモニタに帰る。

D（Dump）命令

D開始アドレス、終了アドレス↵

開始アドレスから終了アドレスまでの間にあるメモリの内容を１バイトずつ１行に１６バイトずつ出力する。

XXXX　XX　XX　……　……　……　XX

↑

この行の最初のバイトのアドレス

Q（Quit）命令

Q ↵

この命令が入力された時にはモニタの処理を終了してオペレーティング・システムにもどる。

メモリの内容をコンソールにダンプするプログラムを学習したあと、メモリにプログラムが入っていた場合には、これを再度ローディングすることが出来るようにするために

- バイナリ・ダンプ
- １６進ダンプ

Intel hex format

Motorola S format

について学習する。

2.17 メモリ・テスト・プログラム

目 的 : Nondestructive Memory test および destructive Memory test の方法について学習する。

Nondestructive Memory test は exclusive ORを利用しておこなう。

a) Memory i の内容をよむ。

b) この値を all 1 で exclusive OR をとる。

c) この値を Memory i に書く。

d) Memory i を読み出し、b)で作った値と比較する。
メモリにエラーがあるかどうかチェックする。

e) 読み出した値を反転して Memory に書く。

destructive test では次のことをおこなう。

peak test 1の bit を下から上に１ビットずつシフトしながらテ

ストしていく。

valley test　１ビットだけ０のビットを作り、このビットを下から上に移動させながらテストしていく。

例

```
(* valley test *)
  k := 1
   for i := 1 to 8 do
   begin
   data := 255 - k;
   memory[loc] := data
   if memory[loc] <> data
   then writeln('error at', loc, 'pattern = ', data);
   k := k * 2
  end;
```

2.18 逆アセンブラ・プログラム

目　的　：　情報をビット・パターンによって分類し、これに対応する他の情報に置き変える方法について学習する。

2.16で配列にデータを記憶させたり変更するプログラムを作成したので、これを利用して配列の中に指定したビット・パターンを記憶させる。

記憶させるビット・パターンは情報処理技術者試験でかって出題されたCOMP－Xのプログラムを利用する。

学習者にはCOMP－Xのプログラムであるということを意識させずに、決められたルールに従って情報を変換させる。

この学習が済んだあと８０８０、Ｚ８０、６８００、６５０２という８ビット・マイクロプロセッサの中から１つを選び逆アセンブラを作成させる。

この学習によって機械語の仕組、アドレシングの間にルールを見付け出すことができるのでハードウェアに対して親しみを持たすことが出来る。

2.19 CPUシミュレータ

目　的：COMP－Xのような簡単な構成のCPUをとり上げ、CPUの中で各命令がどのように処理されていくか学習する。

COMP－Xシミュレータを Pascal でプログラミングし、簡単なCOMP－Xプログラムが実行できるようにする。

このために、次のようなモニタ・プログラムを用意する。

S（Substitute）命令

D（Dump）命令

Q（Quit）命令

T（Trace）命令

シミュレータ・プログラムを学習すると高水準言語のレベルで各命令がCPUチップの中でどのように処理されていくか学習することが出来るので、高水準言語と機械語の命令との間が自然とつながる。

このためにアセンブリ言語のプログラミングに入っていった時に抵抗が少なくなる。またハードウェアとソフトウェアで記述する習慣が身についているとハードウェアを考える場合に無理なく入っていくことが出来るというメリットがある。

カリキュラムの拡張

本カリキュラムは、プログラムがまったく始めてという人が難なくアセンブラ・プログラミングに入っていくことが出来るようにしようという目的で作られてある。

このために、このカリキュラムには遊びがまったくなく、ある意味では非常

に堅苦しくなっている。

本カリキュラムは必修とも云えるものであるので、学習者の興味をそそるように色々な問題を追加する必要があろう。

例えば

関数では乱数の作り方を説明し、これを利用して色々なゲームを作成させるのも一案である。

配列では、

配列を利用したデータの分類、データの円滑化など測定データの処理をおこなうのも一つの方法である。

本カリキュラムではPascalのrecordについては全く除いてあるが、16ビット・マイクロコンピュータの時代になるとアセンブリ言語に於いてもrecordという考え方が導入されているので、必要に応じてrecordを採り上げてもよいだろう。

第3部　アセンブリ・プログラミング入門

第3部　アセンブリ・プログラミング入門

８ビット・マイクロプロセッサの中で一番多く使用されている８０８０のアセンブリ・プログラミングを学習する。

８０８０は8ビット・マイクロプロセッサの中では古典的とも云われるものであるが、このマイクロプロセッサの構造（アーキテクチュア）が１６ビット・マイクロプロセッサの８０８６でも引き継がれているので８０８０のプログラミングを通じてマイクロプロセッサのプログラミングを学習することにしよう。

第1章　8080の構造

1.1　レジスタ

８０８０はＣＰＵチップの中に次のようなレジスタを持っている。

a)　8ビット・レジスタ

Aレジスタ（アキュムレータとも云う。）

Bレジスタ

Cレジスタ

Dレジスタ

Eレジスタ

Hレジスタ

Lレジスタ

これらのレジスタのうちAレジスタは演算の中心となるレジスタであるので演算（Arithmetic）レジスタと呼んでいる。

このレジスタには演算している過程の数値が累計されているのでアキュームレータとも云われている。

B、C、D、E、H、Lレジスタはいずれも補助レジスタとして働らく。

即ち、演算に必要なデータを記憶しておくことが出来るが、＋1あるいは－1の演算しかおこなうことが出来ない。

Aレジスタを始めとして、これらのレジスタはプログラムの中で参照することができる。

b) フラッグ*・レジスタ

このレジスタには演算をおこなったあとのCPUの状態が記憶される。このレジスタには次の情報が記憶される。

キャリー・フラッグ

8ビット目からの桁上がりがあった時には1にセットされ、桁上がりがなかった時には0にリセットされる。

マイナス・フラッグ

演算の結果がマイナスになった時には1がセットされ、プラスの時には0にリセットされる。

プラス／マイナスはビット7、即ち最上位のビットで判断される。このビットが1の時にマイナス、0の時にプラスとなる。

ゼロ・フラッグ

演算の結果が0になった時に1にセットされる。それ以外の時には0にリセットされる。

インターデジット・キャリー・フラッグ

下の4ビット目から桁上がりがあった時あるいは上の4ビットから借りがあった時に1にセットされる。それ以外の時には0にリセッ

---

* それぞれの状態は1か0という状態しか持たない。
この状態は丁度旗（フラッグ）が上がっている、下がっているという状態に対応するので1と0という状態しか持たないものをフラッグと呼んでいる。

トされる。

このフラッグは１０進数を演算した時の補正に使用される。

イーブン・フラッグ

演算した結果の１のビットが偶数個あった場合には１にセットされ、奇数個であった場合には０にリセットされる。

フラッグ・レジスタの内容はプログラムでテストすることが出来るが、キャリー・フラッグ以外のフラッグの値はプログラムで変更することができない。

c) １６ビット・レジスタ

B、Cレジスタ、D、Eレジスタ、H、Lレジスタはそれぞれ対となって１６ビットのレジスタとして働らく。

このような使い方をレジスタ・ペアと呼んでいる。レジスタをペアで使用した場合にはB、D、Hレジスタそれぞれ上の８ビットのデータを保有し、C、E、Lレジスタはいずれも下の８ビット・データを保有している。

Aレジスタとフラッグ・レジスタは他のレジスタと同様に対となって使用されることがある。この場合にはＰＳＷ（Program Status Word）と呼んでいる。

d) コントロール・レジスタ

８０８０ではコントロール・レジスタとしてＰＣ（プログラム・カウンタ）とＳＰ（スタック・ポインタ）を持っている。

ＰＣレジスタは、これから実行しようとしている命令が入っているアドレスを指示する。

このレジスタの内容は命令を実行する都度命令の長さだけ進むが、ジャンプ命令などによってこのレジスタの内容を変更することができる。

ＳＰレジスタはメモリの一部をスタック・エリアとして使用する場合に、１６ビット・データを次にしまうことが出来る場所を示している。

データをスタック領域にしまった場合（ＰＵＳＨ）にＳＰの値は－２され、スタック領域からデータを呼び出した場合（ＰＯＰ）にＳＰの値は＋２される。

ＰＣ、ＳＰとも１６ビットの容量を持っているので８０８０が持つことが出来るメモリの全域を参照することが出来る。

８０８０では電源が入った時あるいはリセット・スイッチを押した時はＰＣとＳＰの値は０にリセットされる。

1.2 メモリ空間

８０８０では１６ビットのアドレス・ラインを持っているので2^{16}、即ち６５５３６バイトのメモリ空間を持っている。

したがって最大６５５３６バイトまでのメモリを実装することが出来る。

メモリ中の各記憶場所には０から６５５３５までの番地（アドレス）が付けられているので、この番地によってある記憶場所を別の記憶場所と区別することが出来る。

1.2.1 ＲＯＭとＲＡＭの使い分け

a) 組み込みシステムの場合

組み込みシステムではプログラムが固定しているのでプログラムをＲＯＭに書き込んでいる。一方、データはたえず変わるのでＲＡＭに書くようにしている。

８０８０では電源が入った時にＰＣの値が０になる、即ち０番地からプログラムが実行されるのでプログラムが入ったＲＯＭは０から始まる低い番地の方に置かれる。またＲＡＭはＲＯＭより高い番地に置かれる。

b) パーソナル・コンピュータの場合

パーソナル・コンピュータの場合には１つのコンピュータで色々な

処理をおこなうためにメモリの低い方の番地はＲＡＭにしている。オペレーティング・システムをフロッピー・ディスクより読み込むためのブート・プログラム、入出力装置を動かすためのプログラムはＲＯＭに書かれ高い方の番地に入っている。

ＲＯＭが高い方の番地に入っている時にはＣＰＵはこのプログラムを直接実行することが出来ない。

このために、ＣＰＵに電源が入った時にはＣＰＵの回路によってＲＯＭがある高い番地に飛ぶ命令を自動的に作り出すようになっている。このようにＣＰＵに電源が入った時に高い方の番地にあるプログラムを実行するようにした回路のことをPower on Jump 回路と呼んでいる。

1.3　Ｉ／Ｏ空間

８０８０ではそれぞれ８ビットの容量を持っている入力ポート、出力ポートを２５６個まで持つことが出来る。それぞれの入力ポート、出力ポートは０から２５５までの番地が付いている。

同じ番号の入力ポート、出力ポートは互に独立して使用することが出来る。

この外、メモリの一部を入力ポートあるいは出力ポートとして使用することも可能である。

独立した入力ポート、出力ポートのことを分離Ｉ／Ｏ（Isolated I／O）と呼ぶのに対してメモリの一部を入力、出力ポートとして使用することをメモリ・マップドＩ／Ｏ（Memory Mapped I／O）と呼んでいる。

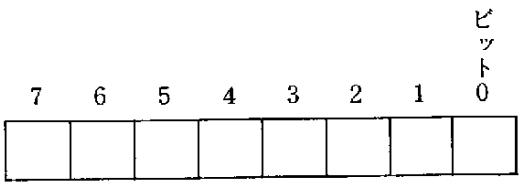
分離Ｉ／Ｏポートの場合には入力命令、出力命令の２つしか使用することが出来ない。これに対して、メモリ・マップド・Ｉ／Ｏ方式の場合には入出力のために８０８０が持っている総べての命令を使用することが出来る。

しかし、８０８０の場合には一般に分離Ｉ／Ｏポート方式を採用している。

第2章　情報の表現

2.1. 符号なしのデータ

８０８０の演算レジスタは１バイト（８ビット）の容量を持っている。



右側のビットをビット０と呼び、左側のビットをビット７と呼んでいる。

それぞれのビットに重みを掛けると右側のビットには2^0=１の重みが付いている。

これに対して左側のビットには2^7＝１２８の重みが付いている。

８ビットでは０から２５５までの値を表現することができることになる。

１６ビット・データの場合には０から2^{16}－１までの値、即ち０から６５５３５までの数値を表現することが出来る。

数値を表すために８ビットあるいは１６ビット全部を使用した場合には数値の符号を表すことが出来ないので、このようなデータのことを〝符号なしのデータ（Unsigned data）と呼んでいる。

2.2. 符号付きのデータ

符号付きのデータ（Signed data）の場合には符号を表すために１ビットを使用する。

符号付きのデータの場合に１は次のようにして表す。

0	0	0	0	0	0	0	1

＋1

一方、－１という値を表す場合には１と０のビットをそれぞれ反転させる。

このようにして負の数値を表す方式のことを〝１に対する補数（1's Complement）〟という。

1	1	1	1	1	1	1	0	－1

符号付きの数値では符号を表すために最上位のビットを利用する。このビットが０の場合にはこれを〝正の数値〟とし、このビットが１の時には〝負の数値〟とする。

１に対する補数の場合には同じ０であっても＋０、－０という２つの状態が存在することになるので、現在は符号付きの数値を表すのに〝２に対する補数（2's Complement）〟が使用されている。

この方式では負の数値は正の数値の各ビットを反転し１を加えることによって求まる。

0	0	0	0	0	0	0	1	＋1

1	1	1	1	1	1	1	1	－1

８０８０の演算回路は符号あり、符号なしに関係なく８ビットを単位として処理するので、符号付きの数値を処理する場合にはこれに応じた処理をプログラミングするように心掛けなくてはならない。

2.3　８進数と１６進数

８０８０は２進数（Binary）で処理をおこなう。しかし、$2^3 = 8$、$2^4 = 16$という関係があるので、この数値を８進数（Octal）あるいは１６進数（Hexadecimal）として取扱うことが出来る。

2進数	8進数	16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

2進数	8進数	16進数
1000	10	8
1001	11	9
1010	12	A
1011	13	B
1100	14	C
1101	15	D
1110	16	E
1111	17	F

2.4 BCDによる10進数

10進数の各桁を2進数で表現したものをBCD（Binary Coded Decimal）と呼んでいる。BCDでは4ビットで10進数1桁を表すことが出来るので1バイトに2桁収めることができる。

8080はBCDで表現されたデータであってもこれを2進数と同じように演算する。10進数では9から10になった時に桁上がりが起きるが2進数では15から16になった時に桁上がりが起きる。

BCDデータの桁上がりが自然であるようにするために8080はDAA（Decimal Adjust）命令が用意されている。

注意：アセンブリ・プログラムで10進数を使用した場合にはプログラムをアセンブルしている時に2進数に変換される。

BCD方式の10進数を使用する時には16進数で書く。

2.5 文字データ

文字データは1文字を表すのに8ビット（1バイト）使用される。

このデータは8080の中では2進数と全く同じように取り扱われるの

で、文字どうしの演算や文字と数値の演算をおこなうことが出来る。

文字を表すのにＪＩＳコードあるいはＡＳＣＩＩ（American Standard Code for Information Interchange）コードが使用されている。

第3章　命令語とアドレシング

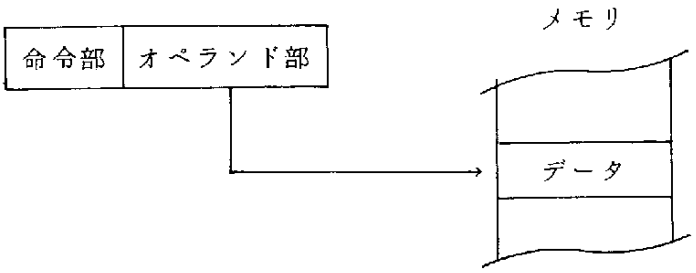
3.1　データのアドレシングの方法

命令語の中で処理の対象となるデータが在る場所を指定する方式のことをアドレシングと呼んでいる。

８０８０では次のアドレシングの方式を持っている。

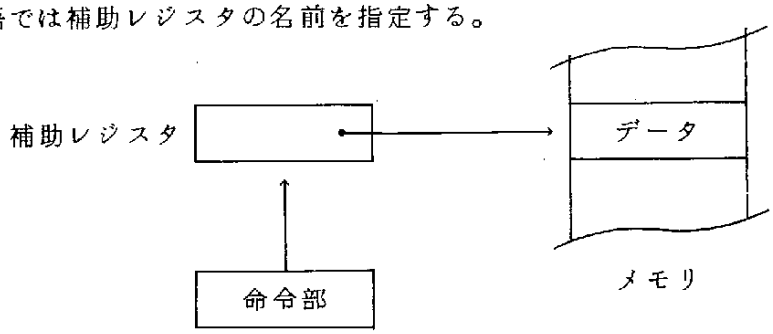
a）直接アドレシング（Direct addressing）

命令語の中で処理の対象となるデータの場所を直接示す。



b）レジスタ間接（Register indirect addressing）

処理の対象となるデータの場所を補助レジスタの中に入れておき、命令語では補助レジスタの名前を指定する。



c）即値（Immediate）

命令語の中で処理の対象となるデータを指定する。データ部はデータの長さによって１バイトあるいは２バイトの長さになる。

命令部	データ

d) 命令によって処理の対象となるデータが入っているレジスタの名前が決まる（Inherent）

命令語によっては命令語それ自体で処理の対象となるデータが入っているレジスタの名前が決まる。

3.2 命令語の長さ

８０８０の命令語の長さは１バイトから３バイトまでで、その長さは命令の種類によって異なる。

a) ３バイト命令

命令部	アドレス部

又は、

命令部	データ部

命令語が３バイトの場合には命令部が１バイト、アドレス部あるいはデータ部が２バイトである。

アドレス部、データ部は、下の８ビットが先に、上の８ビットがうしろの方に入る。

しかし、アドレス部、データ部が参照される場合には正しい順序（上の８ビットが先）になるので１６ビットを単位として取扱っている場合には気にする必要はない。しかし、８ビットずつ単独に処理する場合にはデータが並んでいる順序に注意を払う必要がある。

b) ２バイト命令

命令語	データ

命令語が２バイトの場合には命令部が１バイト、データ部が１バイトとなる。

この命令語は８ビットの即値データを取り扱う時に使用される。

c) １バイト命令

命令語

命令語が１バイトの場合には、命令語それ自体で処理は勿論のこと対象となるデータが入っているレジスタの名前が決まる。

第4章　アセンブリ・プログラミング

本テキストはＣＰ／Ｍというオペレーティング・システムで用意されているＡＳＭアセンブラを利用して学習することができるようになっている。

4.1. プログラムの書式

アセンブリ・プログラムではプログラムの各行は次の３つのフィールドで構成されている。

a) ラベル・フィールド

このフィールドは第１欄目から始まる。このフィールドにアイデンティファイヤを記入することが出来る。このアイデンティファイヤはこの行に付けられた名前として働らくので、このアイデンティファイヤによってこの行を参照することが出来る。

アイデンティファイヤは英文字で始まりこのうしろに英字あるいは数字が続く。

アイデンティファイヤの長さは１６字以上になることが出来ない。読み易い綴りとするために文字と文字の間に＄記号を挿入することが出来るが、この＄記号は読み易さを助けるためのものであり、これは無視される。アイデンティファイヤのうしろに：記号を付けることが出来る。

b) オペレーション・フィールド

アイデンティファイア・フィールドより１字以上の空白あるいはタブコードのうしろにオペレーション・フィールドが来る。

このフィールドには８０８０が持っている命令をインテル社で採用している表記法にしたがって書く。

オペレーション・コードの外にアセンブラに対する指示や疑似命令を書くことが出来る。

c) オペランド・フィールド

オペレーション・フィールドより１文字以上の空白あるいはタブ・コードのうしろからオペランド・フィールドが始まる。

オペランド・フィールドには次のものを書くことが出来る。

i) アイデンティファイア

ii) 数値定数

２進数の場合には２進数数値のうしろにＢを付ける。

８進数の場合には８進数数値のうしろにＯあるいはＱを付ける。

１０進数の場合には１０進数値をそのまま書く。また数値のうしろにＤを付けてもよい。

１６進数の場合には１６進数値のうしろにＨを付ける。

ただし最初の文字がＡからＦで始まっている時にはこの文字のすぐ左側に０を付ける。

iii) 文字定数

文字定数ではＡＳＣＩＩ文字からなる文字列の前後を’と’で囲む。

文字定数の中に’が含まれる場合には’を続けて２つ書く。

一時に書ける文字定数の長さは命令によって変ってくるが、最大64字までであり、１つの文字列は１行の中に収まっていなくてはならない。

4.2 注　釈

；記号を使用して注釈を書くことが出来る。；記号が現れてから１行の終りまでに書いてある文字が注釈として取り扱われる。

4.3 演 算 式

オペランド・フィールドには次の演算式を書くことが出来る。

a＋b、a－b、a＊b、a／b、a MOD b、＋b、－b、NOT B、a AND b、a OR b、a XOR b、a SHL b、a SHR b

4.4. 演算の優先順位

演算は次の優先順位によっておこなう。演算の優先順位が同一であった場合には左から順に演算がおこなわれる。

また(　)を利用し、演算を囲むことによって演算を先におこなわせることも出来る。

4.5. アセンブラに対する命令

ＣＰ／Ｍ　ＡＳＭでは次に示すアセンブラに対する指示命令を持っている。

ＯＲＧ

ＥＮＤ

ＥＱＵ

ＳＥＴ

ＩＦ，ＥＮＤＩＦ

ＤＢ

ＤＳ

ＤＷ

4.5.1. ＯＲＧ命令

ＯＲＧ命令は次の形をしている。

ラベル　　ＯＲＧ　式

ＯＲＧ命令は、アセンブラに対して次の命令を式で与えた１６ビットの値で示す番地からアセンブルするように指示する。

ＣＰ／Ｍの下で動くプログラムは必ず１００Ｈ番地からアセンブルさ

れていなくてはならないので、このプログラムには次の文が必ず必要である。

ＯＲＧ　　１００Ｈ

ＯＲＧ命令のある行にアイデンティファイヤを付けた場合には、アイデンティファイヤに対してオペランド・フィールドにある式が示す１６ビットの値が与えられる。

4.5.2. ＥＮＤ命令

ＥＮＤ命令は必要ではないが、使用する場合にはプログラムの最後の行に書かなくてはならない。

ＥＮＤ命令は次の２つの形をしている。

ラベル　ＥＮＤ

ラベル　ＥＮＤ　式

ラベル・フィールドにはアイデンティファイヤを書いてもよい。

第１の書式ではプログラムの実行開始番地が０番地に設定される。

第２の書式の場合にはプログラムの実行開始番地は式で与えた１６ビットの値で示す番地に設定される。

ＣＰ／Ｍの下ではプログラムの実行開始番地は１００Ｈ番地と決まっているので次のＥＮＤ文が必要である。

ＥＮＤ　　１００Ｈ

4.5.3. ＥＱＵ命令

ＥＱＵ命令は記号に対して数値を与えるために使用する。

ＥＱＵ命令は次の形をしている。

ラベル　ＥＱＵ　式

ラベルにはアイデンティファイヤを書く。アセンブラは式の値を評価し、この値をラベル・フィールドに書いてあるアイデンティファイヤ

に与える。

同じアイデンティファイヤに対しては何度もＥＱＵ命令を使用して値を与えることは出来ない。

4.5.4　ＳＥＴ命令

ＳＥＴ命令はＥＱＵ命令と同じ形をしている。

ラベル　ＳＥＴ　式

ＥＱＵ命令では１つのアイデンティファイアに対して１回しか値を与えることが出来なかったが、ＳＥＴ命令を使用すると１つのアイデンティファイヤに対して何度でも値を与えることが出来る。

ＳＥＴ命令はこの命令が現れる都度式の値を評価してこれをアイデンティファイヤに与える。したがってプログラムをアセンブルしている時に計算をおこなう必要がある場合にこのＳＥＴ命令を使用する。

4.5.5　ＩＦ，ＥＮＤＩＦ命令

ＩＦ命令とＥＮＤＩＦ命令は対となって使用される。

ＩＦ命令とＥＮＤＩＦ命令は次の形をしている。

ＩＦ　　式

８０８０の命令

指示命令

ＥＮＤＩＦ

ＩＦ命令は式の値を評価し、この値が０以外（真）の時にはこの命令からＥＮＤＩＦ命令の前までにある命令をアセンブルする。

しかし、式の値が０の時（偽）にはＩＦ命令からＥＮＤＩＦ命令までにある命令をアセンブルしない。

4.5.6　ＤＢ命令

ＤＢ（Define Byte）命令はバイト単位でメモリ領域を確保し、ここに初期値を設定する場合に使用する。

ＤＢ命令は次の形をしている。

ラベル　ＤＢ　式$_1$，式$_2$，…… ，式n

式１から式ｎまでの値は０から２５５までの範囲でなくてはならない。

ＤＢ命令では式１から順にメモリ領域に割当てていく。

式が文字列であった場合には文字列を構成する文字を順番にメモリ領域に割当てていく。ＤＢ命令ラベル・フィールドにアイデンティファイヤが書かれている時には最初のバイトが割当てられた番地にこの名前が付く。

4.5.7　ＤＷ命令

ＤＷ（Define Word）命令はワード（１６ビット）単位でメモリ領域を確保し、ここに初期値を設定する場合に使用する。

ＤＷ命令は次の形をしている。

ラベル　ＤＷ　式$_1$，式$_2$，…… ，式n

ＤＢ命令と異なってＤＷ命令では文字列を含めそれぞれの式の値を16ビットとして取り扱う。アセンブラはＤＷ命令に出合うと２バイトの領域を確保し、ここに式の値をしまう。式の値は他の８０８０の命令と同じように下の８ビットを番地の低い方に、上の８ビットを高い番地にしまう。

ＤＷ命令のラベル部にアイデンティファイヤが書かれている場合にはこの命令で確保した最初のバイトが在る番地にこの名前が付く。

4.5.8　ＤＳ命令

ＤＳ命令はメモリ上の領域を確保するために使用する。しかしＤＳ命

令はＤＢ命令やＤＷ命令と異なって確保した領域は初期化しない。

ＤＳ命令は次の形をしている。

ラベル　ＤＳ　式

ＤＳ命令では式で与えたバイト数だけ領域を確保する。

ラベル部にアイデンティファイヤが記入されている場合には、確保した領域の最初のバイトにこの名前が付く。

4.5.9　特別な名称

ＡＳＭアセンブラは８０８０が持っているレジスタに次の名前と値を付けている。

A	7	H	4
B	0	L	5
C	1	M	6
D	2	SP	6
E	3	PSW	7

またアセンブラが持っているロケーション・カウンタ（次の命令をアセンブルする番地を指定する）には＄という名前が付いている。

レジスタの名前、ロケーション・カウンタの名前はオペランド・フィールドで使用することが出来る。

ちなみにＤＳ命令は次の命令と同じ働らきをする。

ラベル　ＥＱＵ　＄

　　ＯＲＧ　＄＋式

第5章　8080　プログラミング

5.1　データの転送プログラム

5.1.1　8ビット・データ転送

領域ＡＢＣにある１バイトのデータを領域ＤＥＦに転送する。

```
program xfer1;
var  abc,def:byte;
begin   def:=abc   end.
```

８０８０ではメモリの中にあるデータを別の場所に転送する場合にはＡレジスタを介しておこなう。

メモリにある１バイトをＡレジスタに持ってくるためにＬＤＡという命令が、またＡレジスタにあるデータをメモリに転送するためにＳＴＡという命令がある。

ＬＤＡ、ＳＴＡの命令は次の形をしている。

LABEL	OP	OPERAND	
	LDA	アドレス	(アドレス)→A
	STA	アドレス	A→アドレス

(アドレス)はアドレスの内容を示す。

アセンブラ・プログラムではデータが入る領域は一般にプログラムの領域のうしろにとる。

アセンブラのプログラムでは番地の代わりに名前を使用することが出来るので、これを利用すると上記のプログラムは次のようになる。

```
LABEL     OP      OPR
          ORG     100H
          LDA     ABC
          STA     DEF
          JMP     0
ABC:      DS      1
DEF:      DS      1
          END     100H
```

このプログラムをＣＰ／Ｍで実行するためにはプログラムは１００Ｈから割当てなくてはならない。このためにＯＲＧ命令によってロケーション・カウンタの値を１００Ｈに設定する。

ＬＤＡ命令ではＡＢＣという場所に入っている１バイトがメモリより呼び出されてＡレジスタに入る。

ＳＴＡ命令ではＡレジスタの内容がＤＥＦという場所にしまわれる。

ＪＭＰ ０という命令は、このプログラムの実行を終了してＣＰ／Ｍにもどるということを指示する。

ＡＢＣ：　ＤＳ　１，ＤＥＦ：　ＤＳ　１という命令はそれぞれ１バイトの大きさの領域を確保し、それぞれにＡＢＣ、ＤＥＦという名前を付けるという意味である。

ＥＮＤ　１００Ｈという命令は、プログラムの終了を示すと共にプログラムの開始番地を指示している。

5.1.2　１６ビット・データの転送

領域ＡＢＬＥにある２バイトのデータをＢＡＫＥＲから始まる２バイトに転送する。

```
program xfer2;
var able, baker : integer;
begin  baker := able  end.
```

これまでに学習したＬＤＡ、ＳＴＡの命令を利用すると次のようなプログラムになる。

LABEL	OP	OPR
ＣＰＭ	ＥＱＵ	0
	ＯＲＧ	１００Ｈ
ＳＴＡＲＴ：	ＬＤＡ	ＡＢＬＥ
	ＳＴＡ	ＢＡＫＥＲ
	ＬＤＡ	ＡＢＬＥ＋１
	ＳＴＡ	ＢＡＫＥＲ＋１
	ＪＭＰ	ＣＰＭ
ＡＢＬＥ：	ＤＳ	２
ＢＡＫＥＲ：	ＤＳ	２
	ＥＮＤ	ＳＴＡＲＴ

ＬＤＡ，ＳＴＡ命令では１バイトしか取り扱うことが出来ないので、２バイトのデータを転送する場合には１バイトずつ２回に分けておこなう。

ＡＢＬＥ　ＤＳ　２という命令で２バイト確保した時には、最初のバイトにはＡＢＬＥという名前が付いているので、次のバイトはＡＢＬＥ＋１で参照することが出来る。

ＣＰＭ　ＥＱＵ　０という命令はＣＰＭという名前に０という値を与える。

したがってＪＭＰ　ＣＰＭという命令はＪＭＰ　０と書いたと同じ働

らきをする。

同様にＯＲＧ　１００Ｈという命令はロケーション・カウンタの値を１００Ｈに設定し、次の命令を１００Ｈ番地よりアセンブルするように指定する。

このために次の行のラベル・フィールドに書かれているＳＴＡＲＴという名前には１００Ｈという値が与えられるのでＥＮＤ　ＳＴＡＲＴという命令はＥＮＤ　１００Ｈと書いたと同じことになる。

８０８０ではメモリにある２バイトをＨＬに持って来るためにＬＨＬＤという命令が、またＨＬレジスタにある２バイトをメモリに転送するためにＳＨＬＤという命令がある。

LABEL	OP	OPR	
	LHLD	アドレス	(アドレス) → L
			(アドレス＋1) → H
	SHLD	アドレス	(L) → アドレス
			(H) → アドレス＋1

この命令を利用すると次のようにプログラミングすることが出来る。

```
LABEL     OP      OPR
CPM       EQU     0
          ORG     100H
START:    LHLD    ABLE
          SHLD    BAKER
          JMP     CPM
ABLE:     DS      2
BAKER:    DS      2
          END     START
```

5.1.3　8ビット・データの代入

領域ableに8ビット・データを代入する。

```
program assign8;
var  able:byte;
begin  able:=127    end.
```

領域に8ビット・データを代入する場合には、Aレジスタに代入すべき値を作っておき、この値をSTA命令で領域にしまう。

8ビット・データを各種レジスタに設定するためにMVIという命令が用意されている。

LABEL	OP	OPR
	MVI	A, 値8
	MVI	B, 値8
	MVI	C, 値8
	MVI	D, 値8
	MVI	E, 値8
	MVI	H, 値8
	MVI	L, 値8

MVIの命令を使用すると次のようにプログラミングすることが出来る。

LABEL	OP	OPR
CPM	EQU	0
	ORG	100H
START:	MVI	A, 127
	STA	ABLE
	JMP	CPM

```
ABLE:    DS      1
         END
```

ＤＢ命令を使用すると領域の確保とこの領域の初期化をおこなうことができるので、上記のプログラムが領域の初期化だけのものであれば、次の命令だけで初期化をおこなうことができる。

```
LABEL    OP    OPR
ABLE:    DB    127
```

5.1.4. １６ビット・データの代入

領域bakerに１６ビットのデータを代入する。

```
program assign16;
var  baker:integer;
begin  baker:=16532   end.
```

１６ビット・データは８ビット・データが積み重なったものであるので次のようにプログラミングすることができる。

```
LABEL     OP      OPR
CPM       EQU     0
          ORG     100H
START:    MVI     16532 MOD 256
         (MVI     16532 AND 255)
          STA     BAKER
          MVI     16532 / 256
          (MVI    16532 SHR 8)
          STA     BAKER+1
          JMP     CPM
```

```
BAKER:    DS      2
          END     START
```

このプログラムの場合注意しなくてはならないのは下の８ビットを先にしまわなくてはならないということである。

１６５３２という１６ビット・データの下８ビットは１６５３２を２５６で割った余り、あるいは１６５３２と２５５でＡＮＤを計算した結果で求まる。

同様に１６５３２の上の８ビットは１６５３２を２５６で割った時の商で求まる。

ある値を２５６で割るということはこの値を右に８ビット桁送りをしたと同じことになる。

８０８０では対のレジスタに１６ビットの値をしまうためにＬＸＩという命令を持っている。

LABEL	OP	OPR
	LXI	B，値16
	LXI	D，値16
	LXI	H，値16
	LXI	SP，値16

ＬＸＩ命令ではオペランド部で指定した１６ビットの値を指定されたレジスタ対にしまう。

この命令を利用すると次のようになる。

LABEL	OP	OPR
CPM	EQU	0
	ORG	100H
	LXI	H，16532
	SHLD	BAKER
	JMP	0

```
BAKER:   DS      2
         END     100H
```

先の例と同じようにこのプログラムが領域の初期化だけである場合にはＤＷ命令だけでおこなうことが出来る。

```
BAKER:   DW      16532
```

5.1.5　8ビット・データを補助レジスタを使用して転送する場合

メモリにあるデータを転送する場合にＬＤＡ、ＳＴＡ命令のようにデータが入っている番地を直接指示する代わりに、補助レジスタの助けを借りておこなう命令がいくつか用意されている。

この補助レジスタの助けを借りる命令を使用する場合には補助レジスタに前もってデータが入っている番地を入れておかなくてはならない。

この仕事をするためにＬＸＩ命令が使用される。

```
LABEL    OP      OPR
         LDAX    B
         LDAX    D
         MOV     A,M
         STAX    B
         STAX    D
         MOV     M,A
```

ＬＤＡＸ　Ｂ（又はＤ）命令ではＢＣレジスタ（あるいはＤＥレジスタ）で示されるメモリ番地の内容がＡレジスタに読み出される。

ＳＴＡＸ　Ｂ（又はＤ）命令ではＡレジスタの内容がＢＣレジスタの内容（あるいはＤＥレジスタ）の内容で示される番地に書き込まれる。

ＭＯＶ　Ａ，Ｍ命令ではＨＬレジスタの内容で示されるメモリ番地の内容がＡレジスタに読み出される。

ＭＯＶ　Ｍ，Ａ命令は逆にＡレジスタの内容がＨＬレジスタの内容で

示されるメモリ番地に書き込まれる。

ＬＤＡ，ＬＤＡＸ，ＳＴＡ，ＳＴＡＸ命令は対象となるレジスタがＡレジスタであったが、ＭＯＶ命令では各種８ビット・レジスタ間及びメモリと各種８ビット・レジスタとの間のデータの転送にも利用できる。

補助レジスタの助けを借りて８ビット・データをABLEよりBAKERに転送するプログラムは次のようにプログラミングすることが出来る。

```
LABEL     OP      OPR
CPM       EQU      0
          ORG     100H
          LXI     B,ABLE
          LDAX    B
          LXI     B,BAKER
          STAX    B
          JMP     CPM
ABLE:     DS      1
BAKER:    DS      1
          END     100H
```

ＬＸＩ　Ｂ，ＡＢＬＥによってＡＢＬＥの番地がＢＣレジスタに入る。ＬＤＡＸ　Ｂの命令によって、ＢＣレジスタの内容によって示されるメモリ番地の内容、即ちＡＢＬＥの内容がレジスタに入る。

以下同様にしてＡレジスタの内容がＢＡＫＥＲにしまわれる。

ＨＬレジスタを使用した場合には次のようになる。

```
LABEL     OP      OPR
CPM       EQU     0
          ORG     100H
          LXI     H,ABLE
          MOV     A,M
```

```
        LXI     H,BAKER
        MOV     M,A
        JMP     CPM
ABLE:   DS      1
BAKER:  DS      1
        END     100H
```

5.1.6 補助レジスタを使用して8ビット・データを代入する。

ＭＶＩ命令は８ビット・データを各種レジスタに代入する命令であったが、ＨＬレジスタの助けを借りると８ビット・データをメモリ番地に直接書き込むことが出来る。

LABLE	OP	OPR
	MVI	M,値8

例　ableに１２７を代入する。

LABEL	OP	OPR
	ORG	100H
	LXI	H,ABLE
	MVI	M,127
	JMP	0
ABLE:	DS	1
	END	100H

5.2 ブロック転送

5.2.1 ２５６バイト以内の転送

ableより始まる１０バイトをbakerから順次転送する。

```
program xfer1;
const   size＝10;
var   able, baker：array〔1‥ size〕of byte;
      i：integer;
begin    for i：＝0 to size－1 do
         baker〔i〕：＝able〔i〕
end.
```

転送すべきバイト数が２５６バイト以内の場合には空いているレジスタを使用して転送したバイト数を数える。

この問題を解くために次の命令を使用する。

LABEL	OP	OPR	
	INR	A	(A)＋1→A
	INR	B	(B)－1→B
	INR	C	(C)－1→C
	INR	D	(D)－1→D
	INR	E	(E)－1→E
	INR	H	(H)－1→H
	INR	L	(L)－1→H
	INR	M	((HL))－1→(HL)

この命令を使用するとキャリー・フラッグ以外のフラッグが影響を受ける。

レジスタの内容あるいはメモリの内容から１を引く命令ＤＣＲがある。この命令もＩＮＲ命令と同様にキャリー・フラッグ以外のフラッグに影響を与える。

レジスタの内容を＋１、－１する命令の外にレジスタ対の内容を＋１、－１する命令がある。

LABEL	OP	OPR	
	INX	B	(BC) + 1 → BC
	INX	D	(DE) + 1 → DE
	INX	H	(HL) + 1 → HL
	INX	SP	(SP) + 1 → SP
	DCX	B	(BC) − 1 → BC
	DCX	D	(DE) − 1 → DE
	DCX	H	(HL) − 1 → HL
	DCX	SP	(SP) − 1 → SP

ＩＮＸ、ＤＣＸ命令はいずれもフラッグには影響を与えない。

データを比較するためにＣＭＰ、ＣＰＩという命令が用意されている。

ＣＭＰ命令はＡレジスタにある値と指定されたレジスタあるいはメモリの内容とを比較する。ＣＰＩ命令はＡレジスタの値と命令語で示した値と比較する。

ＣＭＰ命令、ＣＰＩ命令とも比較した結果をフラッグ・レジスタにセットする。

比較をおこなったあとも各レジスタの値は変わらない。

プログラムの遂行順序を変えるために次の命令が用意されている。

LABEL	OP	OPR	
	ＪＭＰ	アドレス	アドレス→ＰＣ
	ＪＮＺ	アドレス	ゼロ・フラッグ＝リセット アドレス→ＰＣ それ以外の時は次の命令
	ＪＺ	アドレス	ＪＮＺの逆
	ＪＮＣ	アドレス	キャリー・フラッグ＝リセット アドレス→ＰＣ それ以外のとき次の命令
	ＪＣ	アドレス	ＪＮＣの逆
	ＪＰＯ	アドレス	イーブン・フラッグ＝リセット

		アドレス→ＰＣ それ以外のとき次の命令
ＪＰＥ	アドレス	ＪＰＯの逆
ＪＭ	アドレス	マイナス・フラッグ＝リセット アドレス→ＰＣ それ以外の時次の命令
ＪＰ		ＪＭの逆

この命令を使用して次のようにプログラミングする。

LABEL	OP	OPR	
SIZE	EQU	10	
	ORG	100H	
	LXI	B, ABLE	
	LXI	D, BAKER	
	MVI	L, 0	転送したバイト数を0にする。
LOOP：	LDAX	B	データを転送する。
	STAX	D	
	INX	B	送る方のアドレスを＋1
	INX	D	受ける方のアドレスを＋1
	INR	L	転送したバイト数を教える。
	MOV	A, L	
	CPI	SIZE	
	JNZ	LOOP	全部転送し終らない時にはＬＯＯＰにもどって転送を続ける。
	JMP	0	
ABLE：	DS	SIZE	
BAKER：	DS	SIZE	
	END	100H	

解　説

順番に並んでいるデータを一つずつ取扱う場合には補助レジスタの助けを借りる。

補助レジスタにデータが始まるアドレス（１６ビット）を入れておき、１バイト処理し終る都度ＩＮＸ命令によって＋１していくと、補助レジスタは次に処理するデータがある場所を示す。

転送すべきバイト数が２５６バイト以内であるので、アドレスを進める時にＩＮＸ命令でなくＩＮＲ命令が利用できそうに思える。ＩＮＸ Ｂ命令の代わりにＩＮＲ　Ｃ命令とするとＣレジスタの値が２５５（０ＦＦＨ）のあと０になる。ＩＮＸ命令であるとＣレジスタからの桁上がりがＢレジスタに加わるが、ＩＮＲ命令であるとこの桁上がりがＢレジスタに加わらない。

データがＸＸ００Ｈ番地から割当てられている時には２５６バイト以内の転送であればＩＮＲ命令を利用することが出来る。

しかし、アドレスの下８ビットが００Ｈでない場合にはアドレスはＸＸＦＦＨ番地からＸＸ００Ｈ番地に変わるのでＩＮＲ命令は利用できない。

比較は引き算でおこなわれる。

今１バイト転送した時にはＬレジスタには１が入っている。Ｌレジスタの内容をＡレジスタに移して比較すると次のようになる。

```
   0  0  0  0  0  0  0  1          1
-) 0  0  0  0  1  0  1  0       -)1  0
-------------------------      ---------
   1  1  1  1  0  1  1  1          - 9
   |
   └→マイナス・フラッグ＝１
      ゼロ・フラッグ　　＝０
      キャリー・フラッグ＝１
```

１０バイト転送した時の状態

```
    0  0  0  0  1  0  1  0
-)  0  0  0  0  1  0  1  0
-------------------------
    0  0  0  0  0  0  0  0
    └→マイナス・フラッグ＝0
       ゼロ・フラッグ　　＝1
       キャリー・フラッグ＝0
```

このフラッグの状態を見て処理をくり返すかどうか判断する。

このプログラムは１０バイト転送したかどうかが判れば良いので転送したバイト数を０から１０まで数えていく代わりに１０から０まで数えても同じことになる。

この考え方に基づいて次のようにプログラミングすることが出来る。

```
LABEL       OP      OPR
SIZE        EQU     10
            ORG     100H
            LXI     B, ABLE
            LXI     D, BAKER
            MVI     L, SIZE
LOOP:       LDAX    B
            STAX    D
            INX     B
            INX     D
            DCR     L
            JNZ     LOOP
            JMP     0
ABLE:       DS      SIZE
BAKER:      DS      SIZE
```

```
END        100H
```

解　説

ＤＣＲ命令はこの命令を実行したあとキャリー・フラッグを除く他のフラッグをセットするので、この命令を使用すると比較をおこなわなくてもよいので大変楽になる。

5.2.2.　２５６バイト以上の転送

転送すべきバイト数が２５６バイト以上になった場合には転送すべきバイト数をレジスタ対に入れておき、ＤＣＸ命令によって－１して来る。

しかし、ＤＣＸ命令はＤＣＲ命令と異なってフラッグをセットしないために両方のレジスタの内容が０になった時にデータ転送を終了するようにプログラミングする必要がある。

８０８０ではＡＤＤ、ＳＵＢという算術演算命令とＡＮＡ、ＯＲＡ、ＸＲＡという論理演算命令がある。

LABEL	OP	OPR
	ADD	A, B, C, D, E, H, L, M
	SUB	A, B, C, D, E, H, L, M
	ANA	A, B, C, D, E, H, L, M
	ORA	A, B, C, D, E, H, L, M
	XRA	A, B, C, D, E, H, L, M

を書くことが出来る。

ＡＤＤ命令はＡレジスタの内容から指定されたレジスタあるいはメモリの内容を加えて、結果をＡレジスタに入れる。

ＳＵＢ命令はＡレジスタの内容から指定されたレジスタあるいはメモリの内容を引き、結果をＡレジスタに入れる。

ＡＮＡ、ＯＲＡ、ＸＲＡ命令はＡレジスタの内容と指定したレジスタあるいはメモリの内容との間で次に示す論理演算をおこない結果をＡレジスタにもどす。いずれの命令も演算の状況を各フラッグにセットする。

ＡＮＡ		
A	B	C
0	0	0
1	0	0
0	1	0
1	1	1

ＯＲＡ		
A	B	C
0	0	0
0	1	1
1	0	1
1	1	1

ＸＲＡ		
A	B	C
0	0	0
0	1	1
1	0	1
1	1	0

ＨＬレジスタの内容が０であるか判定する場合に、両方のレジスタの内容が０である時にＡＤＤ命令とＯＲＡ命令では０になる。

しかしＳＵＢ、ＸＲＡ、ＡＮＡ命令では両方のレジスタの内容が０でなくても演算の結果０になることが有り得るのでこれらの命令は終りの判定に利用することが出来ない。

このことを考慮に入れて次のようにプログラミングする。

```
LABEL     OP      OPR
SIZE      EQU     16000
          ORG     100H
          LXI     B,ABLE
          LXI     D,BAKER
          LXI     H,SIZE
LOOP:     LDAX    B
          STAX    D
          INX     B
          INX     D
          DCX     H
          MOV     A,H
          ORA     L
          JNZ     LOOP
          JMP     0
```

```
ABLE:    DS      SIZE
BAKER:   DS      SIZE
         END     100H
```

5.3　1行のメッセージを出力する。

5.3.1　1行出力を利用する。

```
program message;
begin writeln('This is message')  end;
```

ＣＰ／Ｍは１行のメッセージをコンソールに出力するという仕事をおこなう仕事をするルーチンを持っている。

このルーチンは次のように使用する。

```
LXI     D, メッセージのアドレス
MVI     C, 9
CALL    5
```

（サブルーチンを使用するときの取り決めのことをサブルーチンのコーリング・シーケンスと呼んでいる。）

このサブルーチンを使用する場合にはメッセージの終りに＄文字を入れておかなくてはならないが、この文字は出力されない。

ＣＡＬＬ命令は指定されたアドレスに飛ぶ前に、ＣＡＬＬ命令の次にある命令のアドレスをスタックに入れる。このためにスタック領域を設定しておかなくてはならない。８０８０はＣＡＬＬ文があるとスタック・ポインタの内容を－１し、ＣＡＬＬ文の次にある命令のアドレスの下８ビットをしまう。

スタック領域を設定するに当ってはこのことを考慮に入れなくてはならない。

```
LABEL     OP      OPR
BDOS      EQU     5
          ORG     100H
          LXI     SP,STACK      スタック領域の設定
          LXI     D,MSG
          MVI     C,9
          CALL    BDOS
          JMP     0
MSG:      DB      'This is message'
          DB      13,10,'$'
          DS      20            スタック領域
STACK:    EQU     $
          END     100H
```

解　説

スタック領域はＤＳ命令を使用して確保する。ＤＳ命令のある行にＳＴＡＣＫという名前を付けると、＄、１０（復帰コード）、１３（改行コード）の領域をスタック・エリアとして使用することになる。

８０８０ではスタック・エリアに値をしまう時に－１するので、スタック・エリアのうしろに名前を付けなくてはならない。

5.3.2　１文字出力を利用して１行出力をおこなう。

ＣＰ／Ｍの中にある１文字出力をおこなうサブルーチンを使用して１行を出力する。このサブルーチンのコーリング・シーケンスは次のようになっている。

```
          Eレジスタに出力する文字を入れておく
          MVI     C,2
```

CALL　5

詳細が判らないサブルーチンを使用する場合にはメイン・ルーチンで使用しているレジスタと同じレジスタをサブルーチンで使用しているかも知れない。メイン・ルーチンで使用しているレジスタをサブルーチンで使用しているとサブルーチンから戻って来た時にレジスタの値が変わっているのでうまく働らかなくなる。このような事故を防ぐにはメイン・ルーチンで使用しているレジスタの内容をサブルーチンに入る前に保存しておき、サブルーチンから戻って来た時にレジスタに戻すようにする。

８０８０ではＨＬレジスタとメモリとの間でデータをやり取りするための命令は用意されているが、他のレジスタ対とメモリとの間でデータをやりとりする命令はない。

このためにレジスタ対の内容をスタック領域にしまうようにする。

この処理をおこなうためにＰＵＳＨ、ＰＯＰという命令が用意されている。

LABEL	OP	POP	
	PUSH	H	スタックにしまう
	PUSH	B	
	PUSH	D	
	PUSH	PSW	
	POP	H	スタックからもどす
	POP	B	
	POP	D	
	POP	PSW	

この命令を利用して次のようにプログラミングする。

```
LABEL     OP      OPR
BDOS      EQU     5
          ORG     100H
START:    LXI     SP,STK
          LXI     D,MSG
LOOP:     LDAX    D
          CPI     '$'
          JZ      0           '$'が読まれれば終了
          PUSH    D           DEを一時しまう
          MOV     E,A         読んだ文字をEレジスタに
          MVI     C,9
          CALL    5
          POP     D           DEをもどす。
          INX     D           DEを進める。
          JMP     LOOP
MSG:      DB      '*** MESSAGE',13,10,'$'
          DS      20
STK:      EQU     $
          END     START
```

5.4 値によって処理を分ける

5.4.1 正常なデータが入力された時

キー・ボードより '0' から '9' までの文字を入力し、この文字が偶数を示す文字であればＥＶＥＮと、また奇数を示す文字であればＯＤＤと出力する。

```
program  oddeven;
var  ch:char;
begin
  read(ch);
  case  ch  of
'0','2','4','6','8' : writeln('even');
'1','3','5','7','9' : writeln('odd');
 end
end.
```

ＡＳＣＩＩコードを１６進数で示すと０は３０Ｈ、１は３１Ｈ、２は３２Ｈであるから下の１ビットが０の時には偶数、１の時は奇数という判定をする方法があるが、ここでは別の方法を採用する。

８０８０にはＰＣＨＬという命令がある。この命令はＨＬレジスタの内容で示されるアドレスに飛ぶ。

値によって飛び先が異なる場合にはＣ３００００Ｈ（Ｃ３はＪＭＰ命令の機械語）というデータを用意し、値によって飛び先を求め、この値を００００のところにうめたあとこの命令を実行すれば、任意のアドレスに飛ぶことが出来る。しかし、このプログラムがＲＯＭにあった場合には００００のところに値を書くことが出来ないので値をうめる必要がないＰＣＨＬを使用する。

ＨＬレジスタとＤＥレジスタの内容を変換するためにＸＣＨＧという命令がある。

Ａレジスタに他のレジスタの内容を加えるためにＡＤＤという命令があったと同じようにＨＬレジスタに他の１６ビット・レジスタの内容を加えるＤＡＤ命令がある。

LAB	OP	OPR	
	DAD	H	(HL)+(HL)→HL
	DAD	B	(BC)+(HL)→HL
	DAD	D	(DE)+(HL)→HL
	DAD	SP	(SP)+(HL)→HL

キー・ボードより1文字入力するために次のCP/Mルーチンを利用する。

```
MVA     C, 1
CALL    5
```

このサブルーチンによってキ　・ボードから入力された文字がAレジスタに入る。

```
LABEL   OP      OPR
BDOS    EQU     5
READ    EQU     1
        ORG     100H
START:  LXI     SP,STK
        MVI     C,READ
        CALL    BDOS
        SUI     30H
        MOV     E,A   }
        MVI     D,0   }   16ビットに延長
        LXI     H,TBL
        DAD     D
        DAD     D
        MOV     E,M
        INX     H
        MOV     D,M
```

```
        XCHG
        PCHL
TBL:    DW      EVEN    ; 0
        DW      ODD     ; 1
        DW      EVEN    ; 2
        DW      ODD     ; 3
        DW      EVEN    ; 4
        DW      ODD     ; 5
        DW      EVEN    ; 6
        DW      ODD     ; 7
        DW      EVEN    ; 8
        DW      ODD     ; 9
EVEN:   LXI     D,MSGE
        MVI     C,9
        CALL    5
        JMP     0
ODD:    LXI     D,MSGO
        MVI     C,9
        CALL    5
        JMP     0
MSGE:   DB      'EVEN',13,10,'$'
MSGO:   DB      'ODD',13,10,'$'
        DS      20
STK:    EQU     $
        END     100H
```

解　説

キー・ボードより２という文字が入力された場合を例にとって、このプログラムがどのように動くか眺めてみよう。

Ａレジスタに３２Ｈという値で入ってくるので、この値から３０Ｈを引いて２とする。

飛び先に関する情報が入っているアドレスを求めるために、この値を１６ビットに拡張してＤＥレジスタに入れる。

テーブルの中の要素はそれぞれ２バイトずつであるのでテーブルの始まりのアドレスＴＢＬをＨＬレジスタに入れ、ＤＥレジスタの値をＨＬレジスタに２回加えると、目的の情報が入っているアドレスを求めることができる。飛び先はＤＷで定義してあるので下のアドレスが先、上のアドレスがうしろに入っている。今ＨＬレジスタは下のアドレスが入っている場所を示しているので、このアドレスの内容をＥレジスタに入れ、ＨＬレジスタの値を１進めて、このアドレスの内容をＤレジスタに入れるとＤＥレジスタには飛び先のアドレスが組立つ。ＸＣＨＧ命令によってＤＥレジスタの内容をＨＬレジスタに入れたあとＰＣＨＬ命令によって目的のアドレスに飛ぶ。

5.4.2　異常なデータが入力された時

5.4.1のプログラムは0から9までのデータが入力された場合には正常に働く。

しかし、0から9以外のデータが入力された場合にはこのプログラムはどのように動くか判らない。

例えば：という文字を入力した場合を考えてみよう。：という文字の値は３ＡＨである。３０Ｈを引くと０ＡＨとなり、飛び先に関する情報はＴＢＬ＋１４Ｈのところにあるということになる。

この時はＬＸＩ　Ｄ，ＭＳＧＥという３バイト命令のうち２バイトを

飛び先として取り上げ、ここに飛んでしまう。

したがって正しく動かない。

プログラムはどのような値が入って来ても正常に動くようにしておかなくてはならない。プログラムが正常に動くためには、値の範囲チェックを入れる必要がある。

このためにデータの入力の部分を次のように変更する。

```
        JMP     SKIP
MSG:    DB      'ENTER DATA',13,10,'$'
SKIP:   LXI     D,MSG
        MVI     C,9
        CALL    5
        MVI     C,1
        CALL    5
        CPI     '0'
        JC      SKIP
        CPI     '9'+1
        JNC     SKIP
        SUI     30H
```

解　説

／という文字２ＦＨが入力された時にはＣＰＩ　３０Ｈは次のようになる。

```
  0 0 1 0   1 1 1 1
  0 0 1 1   0 0 0 0
-------------------
↘ 1 1 1 1   1 1 1 1
  └──→マイナス＝1
        ゼロ　　＝0
  └──────→キャリー＝1
```

0という文字より小さい値を持つ文字が入力された場合にはキャリーが立つので、この場合には再入力を指示する。

：という文字３ＡＨが入力された時にはＣＰＩ　'9'＋１（ＣＰＩ　３ＡＨと同じ）という命令は次のように働く。

```
  0 0 1 1 1 0 1 0
  0 0 1 1 1 0 1 0
-------------------
  0 0 0 0 0 0 0 0
    |
    └──→マイナス＝0
          ゼロ　　＝1
  |
  └──────→キャリー＝0
```

したがってキャリーが立っていない時に入力をやり直すようにプログラミングする。

上記のデータの範囲チェックを5.4.1のプログラムに挿入することによって、このプログラムはどのような値を入力しても異常な行動はとらなくなった。

5.4.3　再度入力が可能のようにする

ＣＡＬＬ命令は、サブルーチンに飛ぶ前にこの命令の次のアドレスをスタックに入れる。

サブルーチンから戻る時にはＲＥＴ命令を使用する。この命令はスタックより１６ビットの値をポップし、この値をプログラム・カウンタに入れる。したがって、スタックの上部に入っている値で示されるアドレスに飛ぶことが出来る。

しかし、ＪＭＰ命令などあるいはＰＣＨＬ命令でサブルーチンに飛んだ場合には、スタックに戻りアドレスが入っていないので、メイン・ルーチンに戻ることが出来ない。

しかし、ＰＵＳＨ命令によって戻りアドレスを強制的にスタックの上

部につんでおくと、たとえＪＭＰ命令やＰＣＨＬ命令によってサブルーチンに飛んだ場合でもＲＥＴ命令によってメイン・ルーチンに戻って来ることが出来る。戻りアドレスは強制的に設定するので、ＪＭＰ命令あるいはＰＣＨＬ命令のうしろだけではなく、任意のアドレスに戻ることが出来る。

5.4.1のプログラムを加工する場合には、ＥＶＥＮ、ＯＤＤサブルーチンの中にあるＪＭＰ命令をＲＥＴ命令に変える。

5.4.1プログラムの中のＸＣＨＧ命令の前に次の２行を入れる。

```
LXI     H,SKIP
PUSH    H
```

5.4.2のプログラムを次のように変える。

```
CALL    5
CPI     'Q'
JZ      0
CPI     '0'
JC      SKIP
```

このように改造した5.4.1プログラムと5.4.2プログラムを組合わせると何度でも繰返して実行できるプログラムになる。

このように戻り口をスタックにセットする方式は、１個所から多個所に分岐し、ここから１個所に集まるというプログラムを作成する時に有効な手法である。

5.5　１６ビット・データの処理

5.5.1　符号なしのデータ

８０８０では１６ビット・データを加算するためにＤＡＤの命令を持っているが、この命令はキャリー・フラッグしかセットしない。したがって符号なしのデータ処理をおこなう。

減算は補数を利用しておこなう。

補数を作るためにＣＭＡという命令がある。

LABEL	OP	OPR	
	CMA		Ａレジスタの内容のビットを反転する。

ＨＬレジスタの内容からＤＥレジスタの内容を引くプログラムは次のように作る。

LABEL	OP	OPR	
SUB16：	MOV	A，D	
	CMA		１の補数
	MOV	D，A	
	MOV	A，E	
	CMA		１の補数
	MOV	E，A	
	INX	D	２の補数に直す
	DAD	D	（HL）－（DE）
	RET		

5.5.2　符号付きのデータ

符号付きのデータの場合には演算した結果によって、各種フラッグがセットされるようにする。

演算した時に、下の桁からの桁上がりが上の桁に影響を及ぼすように、次の命令が用意されている。

LABEL	OP	OPR
	ADC	レジスタ
	ADC	M
	ACI	値8
	SBB	レジスタ

SBB　　M

SBI　　値8

ＡＤＣ、ＡＣＩ命令では加算をおこなう時に桁上がりを加える。

ＳＢＢ、ＳＢＩ命令は減算をおこなう時に下の桁からの借り(borrow)を引く。

ＨＬレジスタ、ＤＥレジスタの内容を中心とした場合、それぞれの処理は次のようになる。

a) 加算の場合

LABEL	OP	OPR
	MOV	A，L
	ADD	E
	MOV	L，A
	MOV	A，H
	ADC	D
	MOV	H，L

b) 減算の場合

LABEL	OP	OPR
	MOV	A，L
	SUB	E
	MOV	L，A
	MOV	A，H
	SBB	D
	MOV	H，A

c) 比較の場合

LABEL	OP	OPR
COMP：	MOV	A，H
	SUB	D

```
        RNZ
        MOV     A,L
        SUB     E
        RET
```

比較の場合にはまず上の桁を比較する。上の桁が等しくない場合には判定が付いた。

等しい場合について下の桁を比較する。

比較の場合には減算した時の結果をＨＬレジスタに返さない。

ＲＥＴ命令は無条件に戻る命令であったがＲＮＺ命令はゼロ・フラッグが０（リセット）の時にもどる。このフラッグが１にセットされている時には次の命令を実行する。８０８０には次のような条件によって処理をおこなう命令を持っている。

ＪＭＰ	ＣＡＬＬ	ＲＥ
ＪＮＺ	ＣＮＺ	ＲＮＺ
ＪＺ	ＣＺ	ＲＺ
ＪＮＣ	ＣＮＣ	ＲＮＣ
ＪＣ	ＣＣ	ＲＣ
ＪＰＯ	ＣＰＯ	ＲＰＯ
ＪＰＥ	ＣＰＥ	ＲＰＥ
ＪＭ	ＣＭ	ＲＭ
ＪＰ	ＣＰ	ＲＰ

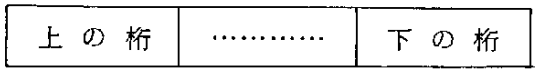
5.6　多倍精度演算

８ビット・データより長い桁のデータを演算することを多倍精度演算（Multiple Precision Arithmetic）と呼んでいる。

この演算は下の桁からの桁上がり、借りを考慮してくり返し処理する。

この種のデータを処理する場合にはデータの並び方が重要になってくる

が、この例ではデータが次のように並んでいると仮定する。



アドレス　　アドレス＋ｎ－１

```
program multi add;
var  data 1, data 2 : array〔1..10〕of byte;
     i : integer;  carry : integer;
begin
    carry := 0;
    for i := 10 downto 1 do
    begin
    data1〔i〕:=data1〔i〕+data2〔i〕;
    if data1〔i〕>=10 then
        begin  data1〔i〕:=data1〔i〕-10;
               carry := 1   end
        else   carry := 0;
    end
end.        注：１バイト１桁とした場合
```

ＤＡＴＡ１、ＤＡＴＡ２にあるデータを加えてＤＡＴＡ１にしまうＭＵＬＴＩＡＤＤというサブルーチンは次のようにプログラミングすることができる。

```
LABEL       OP      OPR
SIZE        EQU     10
MULT$16:
            LXI     H,DATA1+SIZE-1
            LXI     D,DATA2+SIZE-1
```

```
        MOV     C,SIZE
        XRA     A
LOOP:   LDAX    D
        ADC     M
        MOV     M,A
        DCX     H
        DCX     D
        DCR     C
        JNZ     LOOP
        RET
DATA1:  DS      SIZE
DATA2:  DS      SIZE
```

解　説

このプログラムではデータはアドレスの低い方に上の桁が入っているので、下の桁はアドレスが高い方になる。

各バイトを処理する度にアドレスを１つずつ減らしていかなくてはならない。

上の桁はデータが始まるアドレス＋０に入っているので、下の桁のアドレスはデータが始まるアドレス＋桁数－１であることに注意しなくてはならない。

最初の桁を加算する時には下の桁からの桁上がりが無いので、演算に先立ってキャリー・フラッグを０にしておかなくてはならないが、８０８０にはキャリー・フラッグを１にセットするＳＴＣ命令があるが、０にリセットする命令が無い。

このために、結果が０となる演算をおこなってキャリー・フラッグを０にリセットする。Ａレジスタの内容からＡレジスタの内容を引く、あ

るいはAレジスタの内容とAレジスタの内容をＸＲＡすることによってキャリー・フラッグを0にする。

（ＸＲＡでは両方のビットが同じ時にこのビットの結果は0になるということを利用する。）

このような準備をおこなったあと、下の桁からの桁上がりを考慮し、ＡＤＣ命令によって各桁の加算をくり返して処理する。

メモリの内容を直接演算する場合には必ずＨＬレジスタの助けを借りなくてはならないのでＡＤＣ　Ｍ命令を利用する。

注意：Ａ＋Ｂ＝Ｃという演算をおこなう場合には、Ａ、Ｂ、Ｃというデータを参照するために補助レジスタを３ケ使用しなくてはならない。

このようにすると繰返しをコントロールする時にレジスタが使用できなくなるので、スタック領域を利用して見掛け上のレジスタの数を多くしなければならない。

繰返しの判定においてキャリー・フラッグの内容を変更するような命令を使用しないようにする。

また変更されるような場合には下の桁からの桁上がりが入っているＰＳＷを一時的に保存しなくてはならない。

5.7　ＢＣＤ演算

ＢＣＤデータを２進法で加算すると結果はＢＣＤ以外のものになってしまう。

この値をＢＣＤにもどすためにＤＡＡ命令がある。

LABEL	OP	OPR
	DAA	

ＤＡＡ命令は次のことをおこなう。

Ａレジスタの下4ビットが９以上のとき、あるいはインタ・デジット・キャリイが１のとき、Ａレジスタの下４ビットに６を加える。

Ａレジスタの上4ビットが９以上のとき、あるいはキャリー・フラッグが１のときに上８ビットに６を加える。

例　２３　　0 0 1 0 : 0 0 1 1　＝２３

＋）０５　　＋）0 0 0 0 : 0 1 0 1　＝０５

0 0 1 0 : 1 0 0 0

↳0　　↳0

ＤＡＡ

0 0 1 0 : 1 0 0 0　＝２８

３８　　0 0 1 1 : 1 0 0 0　＝３８

＋）９３　　＋）1 0 0 1 : 0 0 1 1　＝９３

1 1 0 0 : 1 0 1 1

↳0　　↳0

ＤＡＡ

0 0 1 1 : 0 0 0 1　＝３１

↳1　　（１３１）

０８　　0 0 0 0 : 1 0 0 0　＝０８

＋）１９　　0 0 0 1 : 1 0 0 1　＝１９

0 0 1 0 : 0 0 0 1　＝２１

↳0　　↳1

ＤＡＡ

0 0 1 0 : 0 1 1 1　　２７

5.8 乗除算

８０８０は乗除算の命令を持っていないので乗除算のルーチンをプログラムしなくてはならない。

乗算の場合には乗数の値が小さい場合には加算の繰り返しを、また除算の場合に商の値が小さい場合には減算の繰り返しで答を求めることが出来る。

しかし、乗数の値が大きい場合あるいは商の値の総和が大きい場合には別の方法で答を求めなくてはならない。

Ａレジスタの内容を１０倍する場合

８０８０は２進法の計算機であるので、１０倍は８倍した値と２倍した値を加えることによって求める。

```
ＡＤＤ     Ａ      (Ａ) ＋ (Ａ) ＝ ２Ａ
ＭＯＶ     Ｂ，Ａ
ＡＤＤ     Ａ      ２Ａ ＋ ２Ａ ＝ ４Ａ
ＡＤＤ     Ａ      ４Ａ ＋ ４Ａ ＝ ８Ａ
ＡＤＤ     Ｂ      ８Ａ ＋ ２Ａ ＝ １０Ａ
```

ＨＬレジスタの内容を１０倍する場合

```
ＤＡＤ     Ｈ      (ＨＬ) ＋ (ＨＬ) ＝ ２ＨＬ
ＰＵＳＨ   Ｈ
ＤＡＤ     Ｈ      ２ＨＬ ＋ ２ＨＬ ＝ ４ＨＬ
ＤＡＤ     Ｈ      ４ＨＬ ＋ ４ＨＬ ＝ ８ＨＬ
ＰＯＰ     Ｄ       ＤＥ ＝ ２ＨＬ
ＤＡＤ     Ｄ      ８ＨＬ ＋ ２ＨＬ ＝ １０ＨＬ
```

１０を２進法で書くと１０１０Ｂであるから１が立っているところを加えれば良いということになる。

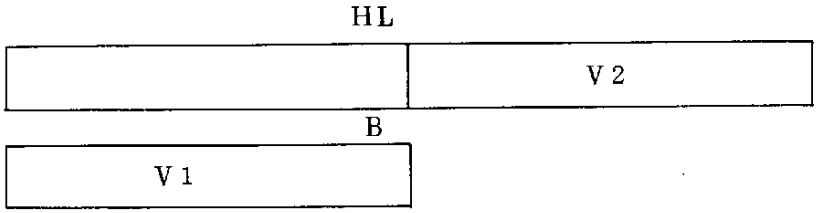
この方法をPascalで記述すると次のようになる。

```
function multiply (x , y : integer) : integer ;
var  v1, v2, r : integer ;  i : integer ;
begin
  v1 := x * 256 ;  v2 := y     r := 0 ;
  for  i := 0  to 7 do
  begin
    if ( v2 & $1 ) = 1   then r := r + v1 ;
    v2 := v2  div 2 ;      r := r div 2
  end ;
   multiply := r
  end ;
```

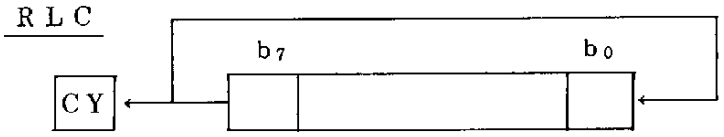
Ｖ２の下の桁が１のときにはＲにＶ１を加える。このあとＶ２とＲを１ビットずつ右にシフトしていく。

この処理を８回繰返すとＲにＶ１とＶ２の積が求まる。

これを図示すると次のようになる。



８０８０では１６ビット・レジスタの内容を $1/2$（右に１ビット・シフト）する命令が無いので利用できるシフト命令を利用しておこなう。

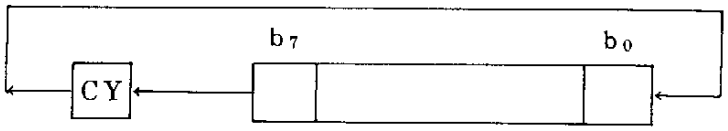


Aレジスタの値を左に1ビット・シフトする。あふれたビットをキャリー・フラッグとb_0に入れる。

RRC

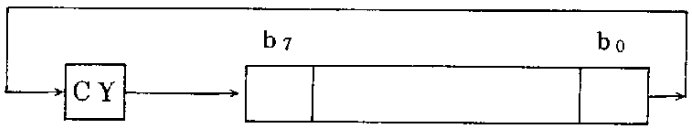
Aレジスタの値を右に1ビット・シフトする。あふれたビットはキャリー・フラッグとb_7に入る。

RAL



Aレジスタの値をキャリー・フラッグを含めて左に1ビット・シフトする。

RAR



Aレジスタの値をキャリー・フラッグを含めて右に1ビット・シフトする。

次の方法で16ビット・レジスタの内容を右に1ビット・シフトすることが出来る。

```
LABEL      OP      OPR
SHIFTR:    XRA     A       キャリー・フラッグ＝0
           MOV     A,H
           RAR
           MOV     H,A
           MOV     A,L
```

```
        RAR
        MOV     L,A
        RET
```

上記サブルーチンを利用してＨＬレジスタに入っているＶ２とＢレジスタに入っているＶ１とを掛け、積をＨＬレジスタに入れるサブルーチンＭＵＬは次のようにプログラミングすることが出来る。

```
LABEL   OP      OPR
MUL:    MVI     C,9
        CALL    SHIFT16
        JNC     SKIP    下のビット＝0
        MOV     A,H
        ADD     B
        MOV     H,A
SKIP:   DCR     C
        JNZ     SHIFT
        RET
```

除算は乗算と同様に次の方法でプログラミングできる。

```
function  divide(v1,v2:integer):integer;
var  v1,v2,r,i:integer;
begin
 v1:=x;  v2:=y*256;  r:=0;
 for i:=0  to 8 do
 begin  r:=r*8
  if(v1-v2)>=0  then
     begin  v1:=v1-v2;  r:=r+1  end;
     v1:=v1*2
  end;
```

```
    divide := r
end;
```

第6章　プログラムの組み立て方

メモリの中に入っている情報を見やすい形にして外部に表示する仕事をおこなうプログラム（Dump program）を開発する場合を例にとってプログラムの作り方を考えてみることにしよう。

メモリに入っている内容を読み易い形で出力するためには、メモリのどこからどこまでという情報を外部から入力しなくてはならない。

この場合、どこまでという情報がどこからという情報よりも大きいかあるいは等しくなくてはいけない。

```
begin
repeat
    ダンプの開始アドレスを入力する。
    ダンプの終了アドレスを入力する。
until 終了アドレス>＝開始アドレス
    メモリのダンプ
end;
```

ダンプすべき範囲が狭い場合にはすべての情報をコンソールの１行に出力することができる。しかし範囲が広い場合には１行に出力することが出来ないので何行かに分けて出力しなくてはならない。

１行には、最左端の情報が入っているメモリ・アドレスを出力し、各メモリの内容を空白とＡＳＣＩＩ文字で出力する。

```
repeat
  address を４桁で出力する。
  １行に表示するバイト数を１６にセット。
  repeat
    空白を出力する。
```

address で示された memory の内容を１バイト表示する。

address を１進める。

出力すべきバイト数を1減らす。

until 最後のアドレスになる。あるいは出力のバイト数が０になる。

復帰改行する。

until 最後のアドレスになる。

ダンプの開始アドレス、終了のアドレス入力はコンソールから１６進数を示すASCII文字４桁でおこなうので次のようになる。

結果を０にする。

入力する文字数をセットする。

repeat

コンソールより１文字入力する。

入力した文字を０から１５までの値にする。

結果を１６倍し、これに変換した値を加える。

入力すべき文字数を１減らす。

until 入力すべき文字数が０

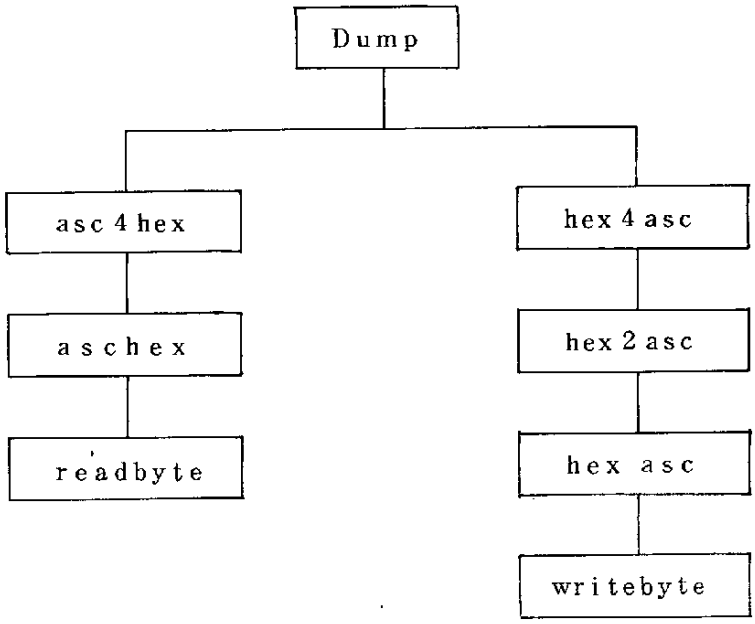
一方出力の方では、addressをＡＳＣＩＩ文字の４桁で出力しなくてはならない。

これは上の１バイト、下の１バイトと分けて出力する。

このためにhex2ascという関数を用意する。

１バイトの内容を１６進数に相当するＡＳＣＩＩ文字２桁で出力するために上の４ビット（ニブルという）と下の４ビットに分け、この値に相当する０～９、Ａ～Ｆという文字に変換する。

メモリ・ダンプのプログラムを完了するためには、次の関数が必要になる。



ダンプ・プログラムのメイン・プログラムは次のようになる。

```
program dump;
var  start-adr, end-adr : integer;
begin
repeat
   write ('enter address');
   star-adr := asc 4 hex;
   write ('enter address');
   end-adr := asc 4 hex;
until  start-adr <= end-adr;
   dump-memory;
end.
```

上記プログラムを８０８０でプログラミングすると次のようになる。

```
LABEL       OP      OPR
            ORG     100H
            LXI     SP,STK
RPT1:       LXI     D,HEADING
            MVI     C,9
            CALL    5
            CALL    ASC4HEX
            SHLD    STRT$ADR
            LXI     D,HEADING
            MVI     C,9
            CALL    5
            CALL    ASC4HEX
            SHLD    END$ADR
            XCHG
            LHLD    STRT-ADR
            CALL    COMPARE
            JNC     REPEAT1
            CALL    DUMP$MEMORY
            JMP     0
```

ここに以下で説明するサブルーチンを挿入する。

```
STRT$ADR:
            DW      2
END$ADR:
            DW      2
```

```
                DS          100         ;STACK
STK:            EQU         $
HEADING:
                DB          'ENTER ADDRESS   $'
                END         100H
```

asc4hexという関数は次のように定義する。

```
function  asc4hex:integer;
var  value, i:integer;   ch:char;
begin
    value:=0;
    for i:=4  downto 1  do
      value:=value*16+aschex(readbyte);
          crlf ;
      asc4hex:=value
end;
```

この関数は16倍があるので、16ビット加算に便利なHLレジスタを変数valueで使用する。

```
LABEL           OP          OPR
ASC4HEX:
                LXI         H,0
                MVI         C,4
LOOP11:
                DAD         H
                DAD         H
                DAD         H
```

```
        DAD     H
        CALL    READBYTE
        CALL    ASCHEX
        MOV     E,A
        MVI     D,0
        DAD     D
        DCR     C
        JNZ     LOOP11
        CALL    CRLF
        RET
```

ＲＥＡＤＢＹＴＥというサブルーチンではＣＰ／Ｍのリード・ルーチンを使用してコンソールから１文字入力する。

このルーチンはＣＰ／Ｍのルーチンを使用するので使用しているレジスタをセーブ、リストアする。

```
LABEL       OP      OPR
READ$BYTE:
            PUSH    H       ┐
            PUSH    D       ├  セーブ
            PUSH    B       ┘
            MVI     C,1
            CALL    5
            POP     B       ┐  リストア
            POP     D       ├  セーブした時と逆の順序で
            POP     H       ┘  取り出す。
            RET
```

ＣＰ／Ｍのキャラクタ・リード・サブルーチンはコンソールから入力したデータをＡレジスタに入れて帰って来るので、ＡＳＣＨＥＸサブルーチンではこ

れをそのまま利用する。

```
LABEL     OP      OPR
ASCHEX:
          SUI     30H
          CPI     10
          RC
          SUI     7       英字なら更に7を引く
          RET
```

改行のルーチンは次のように作る。

```
LABEL     OP      OPR
CRLF:
          MVI     A,13
          CALL    WRITE$BYTE
          MVI     A,10
          CALL    WRITE$BYTE
          RET
```

WRITE$BYTEというサブルーチンはREAD$BYTEと同じようにCP/Mのルーチンを使用する。

```
LABEL     OP      OPR
WRITE$BYTE:
          PUSH    H       ┐
          PUSH    D       │
          PUSH    B       │ レジスタのセーブ
          PUSH    PSW     ┘
          MOV     E,A
          MVI     C,2
          CALL    5
```

```
        POP     PSW  ┐
        POP     B    │
        POP     D    ├ レジスタのリストア
        POP     H    │
        RET          ┘
```

COMPAREサブルーチンは次のようになる。

```
LABEL     OP      OPR
COMPARE:
          MOV     A,H
          CMP     D
          RNZ
          MOV     A,L
          CMP     E
          RET
```

以上で、ダンプの開始アドレス、終了アドレスが入力できたので、次にダンプの本体を定義する。

```
procedure dump-memory;
var i:integer;
begin
  repeat
    crlf;
    hex4asc(start-address);
     i:=16;  (* 16 bytes/line *)
  repeat
      space;
      hex2asc(memory[start-address]);
        start-address:=start-address+1;
```

```
        i := i - 1;
    until (start-address > end-address) or i = 0;
    until  start-address > end-address
        crlf ;
  end;
```

この手続きに対応する８０８０のプログラムは次のようになる。

```
LABEL       OP      OPR
DUMP$MEMORY:
REPEAT30:
            CALL    CRLF
            LHLD    STRT$ADR
            CALL    HEX4ASC
            MVI     C,16
REPEAT31:
            CALL    SPACE
            MOV     A,M
            CALL    HEX2ASC
            XCHG
            LHLD    END$ADR
            XCHG
            CALL    COMPARE
            JNC     EXIT
            INX     H
            DCR     C
            JNZ     REPEAT31
            JC      REPEAT30
EXIT:
```

```
                CALL    CRLF
                RET
```

ＳＰＡＣＥというサブルーチンはＣＲＬＦと同様に作成する。

```
LABEL           OP      OPR
SPACE:
                MVI     A,20H
                CALL    WRITE$BYTE
                RET
```

hex4asc という手続きは次のように定義される。

```
procedure hex4asc(x:integer);
begin
      hex2asc(x div 256);
      hex2asc(x mod 256)
end;
```

このプログラムでは対象となるデータはＨＬレジスタに入っている。

したがってx div 256はHレジスタの内容ということになる。同様にx mod 256はＬレジスタの内容ということになる。

したがってＨＥＸ４ＡＳＣというサブルーチンは次のように定義できる。

```
LABEL           OP      OPR
HEX4ASC:
                MOV     A,H
                CALL    HEX2ASC
                MOV     A,L
                CALL    HEX2ASC
                RET
```

更にhex2ascという手続きは次のように定義できる。

```
procedue  hex2asc(x:integer);
begin
        hex1asc (x div 16);
        hex1asc (x mod 16)
end;
```

８０８０には割算の命令がないので、１６で割る処理は右に４ビット・シフトするということで代用しなくてはならない。

ビットが循環する形のシフトであるので、右に４ビット・シフトするだけでは１６で割ったことにならない。

そこで下の４ビットを抽出して上の４ビットを０にし、１６で割ったと同じことをしなくてはならない。

同様に mod １６は下の４ビットを抽出することになる。

この抽出の処理はhex1ascの方にまかせる。

この手続きに対応するサブルーチンは次のようになる。

```
LABEL    OP       OPR
HEX2ASC:
         PUSH     PSW
         RRC   ┐
         RRC   │
         RRC   ├  1/16
         RRC   │
         RRC   ┘
         CALL     HEX1ASC
         POP      PSW
         CALL     HEX1ASC
```

RET

アセンブリ・プログラムではサブルーチンに値を渡す場合にレジスタを介して渡す。呼ばれたサブルーチンの方でこのレジスタを使用している場合には、この値はなくなってしまうので、この手続きのように渡された値を２度使用するような場合にはサブルーチンにコントロールを渡す前に自分で保存しておかなくてはならない。

ＰＵＳＨ、ＰＯＰ命令でＰＳＷを指定するとＡレジスタとフラッグが処理の対象となる。

１桁の１６進数をＡＳＣＩＩ文字に変換する手続き hex1asc

この手続きは次のように定義される。

```
procedure  hex1asc(x : integer);
begin
if x > 9 then  x:=x+7;  write-byte(chr(x))
end;
```

アセンブリ言語のプログラムではパラメータはレジスタ渡しとなるので、これを利用して次のようにプログラミングする。

```
HEX1ASC:    ANI   0FH      下４ビット抽出
            CPI   0AH      ０～９まででキャリーが１にセットさ
            JC    SKIP1    れるので、この時はSKIP1に飛ぶ。
            ADI   7        Ａ～Ｆの時に補正する。
SKIP1;      ADI   30H
            CALL  WRITE$BYTE
            RET
```

８０８０にはＤＡＡという命令があるので、この命令を利用すると次のようにプログラミングすることが出来る。

```
HEX1ASC:  ANI     0FH     下４ビット抽出
          ADI     90H
          DAA
          ACI     40H
          DAA
          CALL    WRITE$BYTE
          RET
```

上記のプログラムはＡレジスタが０から９までの時には次のように働く。

即ち、９０Ｈを加えて９０Ｈから９９Ｈになる。ＤＡＡでは値が変わらないのでキャリーが立たない。したがってＡＤＣ命令を実行した時にＤ０ＨからＤ９Ｈとなる。

次いでＤＡＡを実行した時に３０Ｈから３９Ｈになる。

一方、Ａレジスタの値がＡからＦの時には次のように働く。

９０Ｈを加えると９ＡＨから９ＦＨになる。ＤＡＡで補正するとＡ０ＨからＡ５Ｈとなりこの時にキャリーが発生する。

次のＡＣＩ命令はＡレジスタの内容に３０Ｈとキャリーの内容を加えるので、この結果はＥ１ＨからＥ６Ｈになる。

これに対してＤＡＡ命令による補正をおこなうと４１Ｈから４６Ｈという値になる。

ＨＥＸ１ＡＳＣというサブルーチンではＡレジスタの値が０から９までの時には"０"（３０Ｈ）から"９"（３９Ｈ）までの値を返し、値がＡＨからＦＨまでの時には"Ａ"（４１Ｈ）から"Ｆ"（４６Ｈ）までの値を返す。

以上でメモリ・ダンプ・プログラムとアセンブリ・プログラムが完成した。

Ｐａｓｃａｌを使用して記述した処理の定義と実際のアセンブラ・プログラムとの間に若干の相違が出てくる。

例えばダンプしようとしているメモリのアドレスが０ＦＦＦＦＨ番地であった時に、これに１を加えると、１００００Ｈ番地でなく００００Ｈ番地になっ

てしまう。

したがってアドレスを進めたあとstart－address ＞ end－address という比較は成立しなくなる。このために1を加える前に比較しなくてはならない。

Pascalで定義したプログラムが正しいかどうか検証しようとした場合にも同様のことが起きる。

Pascalの整数は符号付きであるので、7FFFH（32767）から8000H（－32768）に変わった時にも比較が成立しなくなる。

1行を表示したあと、終りになったか比較すると、ダンプの開始アドレスの下の4ビットが0か8以外の値で始まっていると、0FFFFHを越えてしまうことがある。

これを避ける為には、毎バイトを表示した時に終ったかどうか判定しなくてはならない。

判定はCOMPAREというサブルーチンでおこなう。終りまで進んだ時にはキャリー・フラッグを0、まだ終っていない時には1にしている。

終っていない時には、出力すべきバイト数を1減らし、残りのバイト数が0でない時には同一行の処理を続けるが、この判定のためにDCR命令を使用している。

DCR命令はフラッグの内容を変更する、しかしキャリー・フラッグには影響を与えない。

このために、1行の処理が終った時にもう一度、全部のダンプが終ったか比較しなくてもキャリー・フラッグだけ調べればよい。

第7章　サブルーチンの作成法

7.1　サブルーチンへのパラメータの渡し方

サブルーチンにパラメータを渡す時に値そのものを渡す場合（Call by Value）と値が入っているアドレスを渡す場合（Call by reference）という2つの方法がある。

アドレスを渡した場合にはデータを間接的に参照することになるので、まず値渡しの場合について説明することにしよう。

i）レジスタ渡し

サブルーチンで必要とする値を各種レジスタに入れた上でサブルーチンを参照する方法である。一般にこの方法が使用されるが、この方法では8ビット・データの場合には7個までのデータ、16ビット・データの場合には3個までのデータしか渡すことが出来ない。

ii）スタック渡し

サブルーチンで必要とする値をスタックに積んだ上でサブルーチンを参照する方法である。この方法では渡すことが出来るパラメータの数に制限がないのでコンパイラで生成したプログラムではこの方法を採用している。

スタックの中に入っているパラメータの参照の仕方はどのような命令が利用できるかによって変わってくる。

8080の場合にはスタック・ポインタに入っている値を中心としてこれからの偏位でスタックの中に入っているデータを参照するという命命を持っていない。またインデックス・レジスタを持っていない。

このためにスタックの上部にある16ビットの値と（TOS＝ Top of Stack）とHLレジスタにある値とを交換するXTHLという命令を使用してスタックに入っているパラメータを利用する。

ＸＴＨＬ命令

（HL）⟷ＴＯＳ　（Top of Stack）

この命令を利用する場合にはスタックの中に値がどのような順番で入っているかたえず頭の中に入れておくか、あるいは図に書いておかなくてはならない。

この方法を利用して２つの値を加え、この結果を返すサブルーチンであるＡＤＤ２をプログラミングしてみよう。

このサブルーチンは次の方法で参照される。

```
        ＬＸＩ      Ｈ，５
        ＰＵＳＨ    Ｈ
        ＬＸＩ      Ｈ，６
        ＰＵＳＨ    Ｈ
        ＣＡＬＬ    ＡＤＤ２
```

ＡＤＤ２というサブルーチンが参照された時にはスタックには次の順序で値が入っている。

ＴＯＳ	戻りアドレス
ＴＯＳ－１	６
ＴＯＳ－２	５

一方、ＡＤＤ２サブルーチンは次のようにプログラミングされる。

```
ADD2:   POP     H     戻りアドレス→ＨＬ
        XTHL          ６→ＨＬ，戻りアドレス→ＴＯＳ
        XCHG          ６→ＤＥ
        POP     H     戻りアドレス→ＨＬ
        XTHL          ５→ＨＬ，戻りアドレス→ＴＯＳ
        DAD     D     ５＋６＝１１→ＨＬ
```

```
        XTHL            戻りアドレス→ＨＬ，１１→ＴＯＳ
        PUSH    H       戻りアドレス→ＴＯＳ
                        １１→ＴＯＳ－１
        RET             戻りアドレス→ＰＣ，１１→ＴＯＳ
```

このサブルーチンの実行が終ってメイン・ルーチンに戻って来た場合にはスタックの上部に結果が入っている。

この方法はサブルーチンから値をいくつか返す場合にも利用することができる。

ⅲ) サブルーチンのコーリング・シーケンスの中にパラメータを列記する。

例

```
            ＣＡＬＬ    ＡＤＤ２
    Ｘ１：  ＤＷ        Ｖ１
    Ｘ２：  ＤＷ        Ｖ２
    Ｘ３：  メインの処理        結果はＨＬレジスタに残る。
```

この方法はアセブリ言語のプログラミングで良く見られる方法である。

サブルーチンＡＤＤ２を参照した時にはスタックの上部にＶ１という値が入っているアドレスがあるということを利用する。

```
ADD2：  POP     H       Ｘ１→ＨＬ
        MOV     C,M
        INX     H
        MOV     B,M     Ｖ１→ＢＣ，Ｘ１＋１→ＨＬ
        INX     H       Ｘ２→ＨＬ
        MOV     E,M
        INX     H
        MOV     D,M     Ｖ２→ＤＥ
        INX     H       Ｘ３→ＨＬ
        PUSH    H       戻りアドレス→ＴＯＳ
```

```
        XCHG
        DAD     B       V1＋V2→HL
        RET
```

パラメータの数が多い場合には、スタック・ポインタの値を入れるとパラメータがスタックの中に並んでいると同じように処理することができる。

```
ADD2:   POP     D       X1アドレス→DE, TOS＝空
        LXI     H,0
        DAD     SP
        SHLD    TEMPSTK SPのセーブ
        XCHG
        SPHL    X1→SP
        POP     B       V1→BC, X2→SP
        POP     H       V2→HL, X3→SP
        DAD     B       V1→V2→HL
        SHLD    RESULT
        LXI     H,0
        DAD     SP
        XCHG            X3→DE
        LHLD    TEMPSTK
        SPHL            SPリストア
        PUSH    D       X3→TOS
        LHLD    RESULT
        RET
```

この処理の変形として、パラメータだけを一まとめにして、このパラメータが入っている箱のアドレスをサブルーチンに送るという方法がある。

８０８０では乗除算の命令を持っていないので、見掛けのレジスタをメモリ上に用意しこのレジスタで演算をおこなうといった場合に、このような使い方をする。

例

```
                LXI     H, PACKET
                CALL    MULT
                  :
                  :
PACKET:         DW      0, 0
                DW      0
```

この場合にはスタック・ポインタの値を入れかえることによってパラメータがコーリング・シーケンスの中に並んでいる場合と同様に処理する。

7.2 再帰的なサブルーチン

Pascalでｎ！を計算する関数は次のようにプログラミングする。

```
function fact(n:integer):integer;
begin
if  n=0  then  fact:=1
         else  fact:=fact(n-1)* n
end;
```

マイクロプロセッサはスタックを持ち、サブルーチンを参照した時に戻りアドレスが自動的にスタックの上部にしまわれる。

このために、サブルーチンがサブルーチンの中から参照されてもサブルーチンがどこで終るかはっきりとしていれば、自分で自分を参照する（再

帰的な）サブルーチンをアセンブリ言語でプログラミングすることができる。この場合にはパラメータをスタックを介して渡す場合に使用したと同じ方法でプログラミングする。

データのスタックと戻りアドレスを８０８０のハードウェア・スタックを利用するとｎ！を計算するサブルーチンＦＡＣＴは次のようにプログラミングすることができる。

```
FACT:
;           ENTRY
;           A-REG = VALUE
;           EXIT
;           TOS    = N!
;
            ORA     A           ; N＝0？
            JNZ     NON$ZERO
            LXI     H,1             N＝0の時は1をスタッ
            XTHL                    に入れてもどる。
            PUSH    H
RET1:       RET
NON$ZERO:
            MVI     H,0         Nの値を16ビットに拡張し
            MOV     L,A         スタックにしまう。
            PUSH    H
            DCR     A             N－1を算出
            CALL    FACT          FACT(N-1)
RET3:       POP     H             HL＝(N-1)
            POP     D             DE＝N
            XRA     A           結果を0にしておく。
```

```
MLOOP:  ADD     D           N＊FACT(N−1)！を
        DCR     L           算出する。
        JNZ     MLOOP
        MVI     H,0         結果を１６ビットに拡張し
        MOV     L,A         スタックにしまう。
        XTHL
        PUSH    H
RET2:
        RET
```

解　説

０！のとき

ＴＯＳ＝戻りアドレス、Ａ＝０で入ってくるので、ＴＯＳ＝戻りアドレス、ＴＯＳ－１＝１としてＲＥＴ１でもどる。

１！のとき

ＴＯＳ＝戻りアドレス、Ａ＝２で入ってくる。

Ａレジスタの値が０でないので、１をスタックに入れ、Ａレジスタの値を－１してＦＡＣＴに入れる。

ＴＯＳ　＝　戻りアドレス（RET3）

ＴＯＳ－１＝１（N）

ＴＯＳ－２＝戻りアドレス

この時Ａレジスタの値が０であるので、スタックを次のようにする。

ＴＯＳ　＝　戻りアドレス

ＴＯＳ－１＝１（FACT（N−1））

ＴＯＳ－２＝１（N）

ＴＯＳ－３＝戻りアドレス

ＲＥＴ１にあるＲＥＴ命令によってＲＥＴ３にもどる。

ＴＯＳ　＝　１（FACT（N−1））

ＴＯＳ－１＝１（Ｎ）

ＴＯＳ－２＝戻りアドレス

ＲＥＴ３よりＲＥＴ２までの間でＮ＊ＦＡＣＴ（Ｎ－１）の計算をおこなう。

このあとＸＴＨＬ命令とＰＵＳＨ　Ｈ命令によってスタックの内容を次のように修正する。

ＴＯＳ　＝　戻りアドレス

ＴＯＳ－１＝Ｎ！

ＦＡＣＴを参照したメイン・ルーチンに戻って来た時にはスタックの上部にＮ！の値が入っている。

第8章　I／Oのプログラミング

I／Oプログラミングは使用するインタフェース、即ちどのようなI／Oサポート・チップを使用しているか、どのような制御方式を採用しているかによって変わってくる。

I／O制御の方式には次の3通りがある。

a) プログラムI／O

b) プログラム割込み

c) DMA（Direct Memory Access）

プログラムI／O方式は入出力装置に命令を出したあと、次の命令を受け入れることが出来るようになるまでプログラムで監視する方式である。

この方式ではI／Oの監視のためにCPUの時間がすべて取られてしまうので、高速データの処理をおこなうことが出来ない。

プログラム割込み方式ではデータが入力された時、あるいは次のデータを出力することが出来るようになった時に、I／Oの動作の完了をプログラムの自動中断機構（割込み）によって知らされるので、I／Oの状況をたえず監視する必要はない。

プログラムI／O方式、プログラム割込み方式ともデータの入力、出力はプログラムによっておこなう。

DMA方式はCPUの助けを借りることなくI／O装置とメモリとの間で直接やりとりがおこなわれる。このために高速度でデータの授受をおこなうことが出来る。

また、I／O動作の完了はプログラムの自動中断によって知らされるので、プログラムでI／O装置の状況を監視する必要がない。

次の表はデータの転送速度とCPUの占有率を各種I／O制御方式毎にまとめたものである。

ＣＰＵ占有率

データ速度	プログラムＩ／Ｏ％	ＩＯ割込み％	ＤＭＡ　％
１ＫＢ／Ｓ	１００％	2.5％	0.2
３ＫＢ／Ｓ	１００％	7.5％	0.6
５ＫＢ／Ｓ	１００％	１2.5％	1.0
１０ＫＢ／Ｓ	１００％	２5.0％	2.0
３０ＫＢ／Ｓ	１００％	７5.0％	6.0
５０ＫＢ／Ｓ	－	－	１0.0
１００ＫＢ／Ｓ	－	－	２0.0
３００ＫＢ／Ｓ	－	－	６0.0
５００ＫＢ／Ｓ	－	－	１００

8.1　シリアル・インタフェース

マイクロコンピュータのハードウェアの基礎　ページ１６２に紹介されている８２５１というプログラマブル通信インタフェース・チップを使用して、ＣＲＴディスプレイ・ターミナルをプログラムＩ／Ｏ方式でコントロールする方法について説明することにしよう。

ここでは次のようなＣＲＴディスプレイ・ターミナルを使用する。

1. データ速度　　９６００ボー
2. データ長　　７ビット
3. パリティ　　偶数
4. ストップ・ビット　　1
5. 通信方式　　全二重

８２５１は次の方式で使用する。

1. 制御方式　　非同期
2. クロック　　データ速度の１６倍のクロックを使用

3. データ・ポート　　８０Ｈ

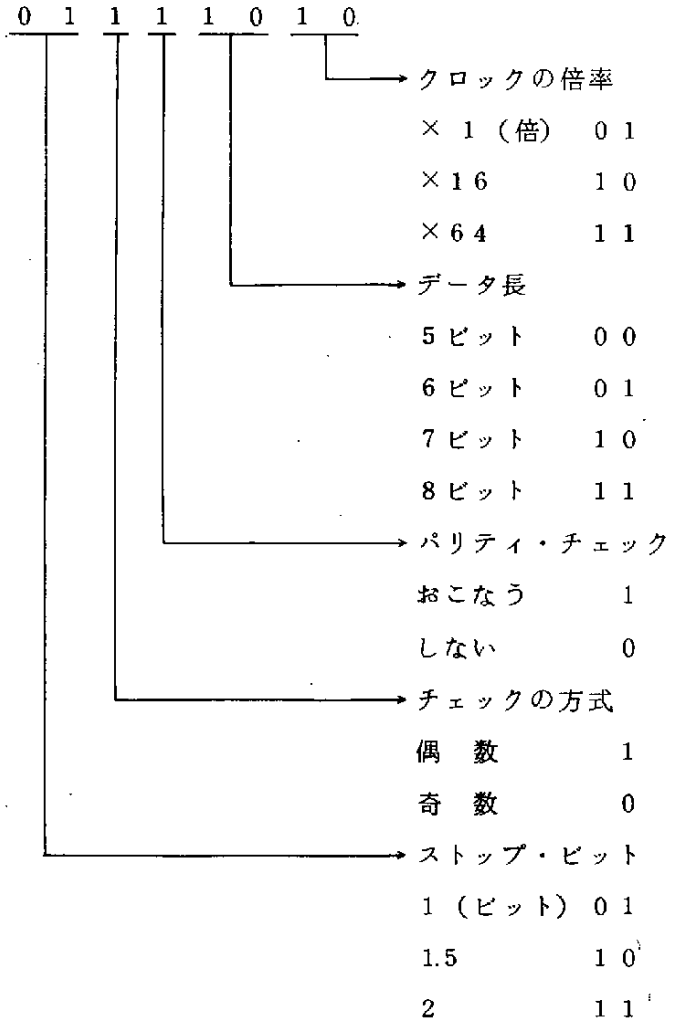
4. ステータス・ポート　　８１Ｈ

5. バス・ライン　　正論理

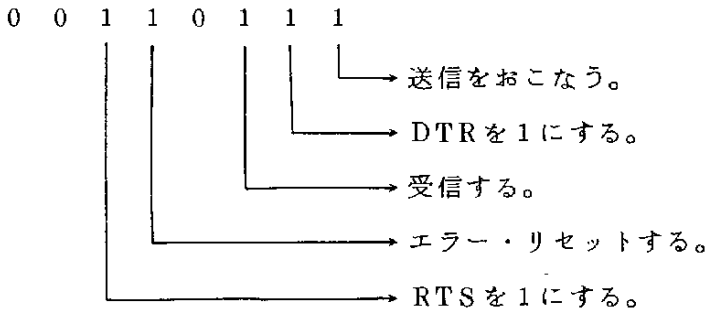
8.1.1　８２５１の初期化

８２５１を使用する場合にはモード・コマンドの初期設定をおこなわなくてはならない。

８２５１のモード設定をおこなう時の形式は次のようになっている。



８２５１のコマンド設定をおこなう時の形式は次のようになっている。



８２５１の初期化をおこなうサブルーチンは次のようになる。

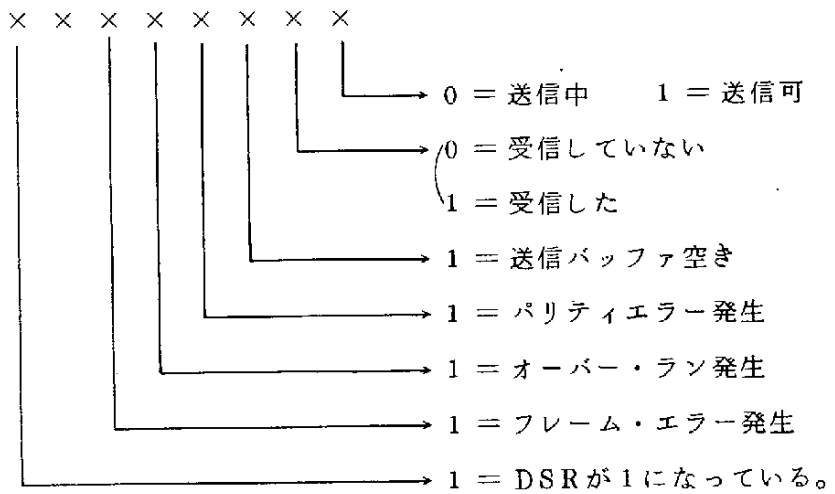
LABEL	OP	OPR
	MVI	A, 01111010B
	OUT	81H
	MVI	A, 00110111B
	OUT	81H
	RET	

8.1.2 ８２５１のステータス

８２５１からのデータ読み込みは次のようにする。

```
while   データが受信されなければ
        何もしない。
        受信したデータを読む。
```

８２５１の状態は次のステータスによって示される。



8.1.3　READサブルーチン

８２５１より１バイト・データを受信するためのＲＥＡＤサブルーチンは次のようになる。

```
LABEL    OP      OPR
READ：
         IN      81H     データを受信するまで待つ
         ANI     2
         JZ      READ
         IN      80H
         OUT     80H     エコー・バック
         ANI     7FH     下７ビット抽出
         RET
```

端末装置を全二重方式で使用している場合には、キー・ボードからデータを入力してもこのデータはＣＲＴディスプレイには表示されない。キー・ボードから入力したデータをＣＲＴディスプレイに表示したい場合には８２５１が読んだデータ、即ちＡレジスタにあるデータを送り返さなくてはならない。

8.1.4 ＷＲＩＴＥサブルーチン

８２５１からデータを１バイト送出する場合には出力すべきデータをＡレジスタに入れたあとＯＵＴ命令を出す。

Ａレジスタに入っているデータを出力するＷＲＩＴＥサブルーチンは次のようになる。

```
LABEL     OP      OPR
WRITE:
          PUSH    PSW
WAIT:     IN      81H
          ANI     1
          JZ      WAIT
          POP     PSW
          OUT     80H
          RET
```

8.1.5 端末の制御

端末装置がデータを受け取れない場合が往々にして発生する。

この場合に、8.1.4で述べたＷＲＩＴＥサブルーチンでは端末装置の状態に関係なくデータを出力してしまうので端末装置で受信しそこなう。

これを防ぐために次の２つの方法がある。

a) ＤＳＲを０にする

端末装置の状態は端末装置のＤＴＲ（Data Terminal Ready）端子に出ている。

端末装置が受信可能な時に１、受信不可能な時に０となっている。

端末からのＤＴＲ信号は、ＣＰＵ側のＤＳＲ（Data Set Ready）端子につながっている。

この端子の信号が８２５１のステータスのビット７に反映している。

したがって、送信可と共にこの情報をチェックしたあとデータの出力をおこなうようにプログラムする。

注意：端末側のＤＴＲが０になってからＤＳＲがテストされるまでにデータが送られている場合がある。

端末側に受信バッファが無いとＤＳＲで送信をコントロールしてもデータが落ちることがある。

b) ＸＯＮ／ＸＯＦＦ

ＤＳＲ／ＤＴＲでコントロールをおこなうとデータの信号線の外に制御用の信号線が必要になる。

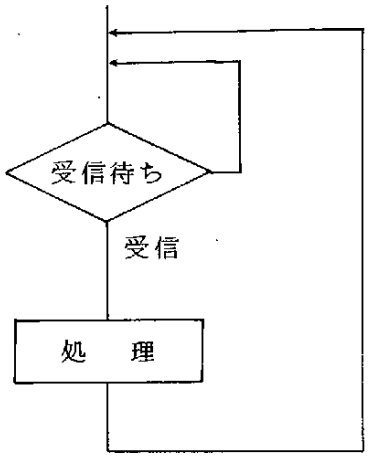
データの信号線だけでコントロールをおこなうためにＸＯＮ／ＸＯＦＦ方式がある。

この方式では端末装置が受信バッファがいっぱいなどの状態になった時に端末装置からＣＰＵにＸＯＦＦ（又はＤＣ３＝１３Ｈ）を送る。端末装置は受信可能になった時にＸＯＮ（又はＤＣ１＝１１Ｈ）を送る。

したがってＣＰＵはＸＯＦＦを受信してからＸＯＮを受信するまでデータの送信を中断する。

8.2 プログラム割込み方式の制御

プログラムＩ／Ｏ方式でＩ／Ｏ装置を働らかせた場合には、処理は次の図のようになるので、プログラムの処理時間が長い場合にはデータが続けて入って来た場合には先のデータを失う。



データがどのような高速で入力されてもデータを失わないで受信するためにはＩ／Ｏサポート・チップでデータを受信した場合に、サポート・チップからＣＰＵに知らせ、ＣＰＵは今までの処理を中断してこのデータを取り込むようにする。ＣＰＵはデータを取り込んだあと中断した処理を実行する。

プログラム割込み方式では受信したデータを一時的に保存しておくために受信バッファを用意する。

割込みによって起動される受信プログラムではデータを受信した時に、このデータを受信バッファの一番うしろにおく。

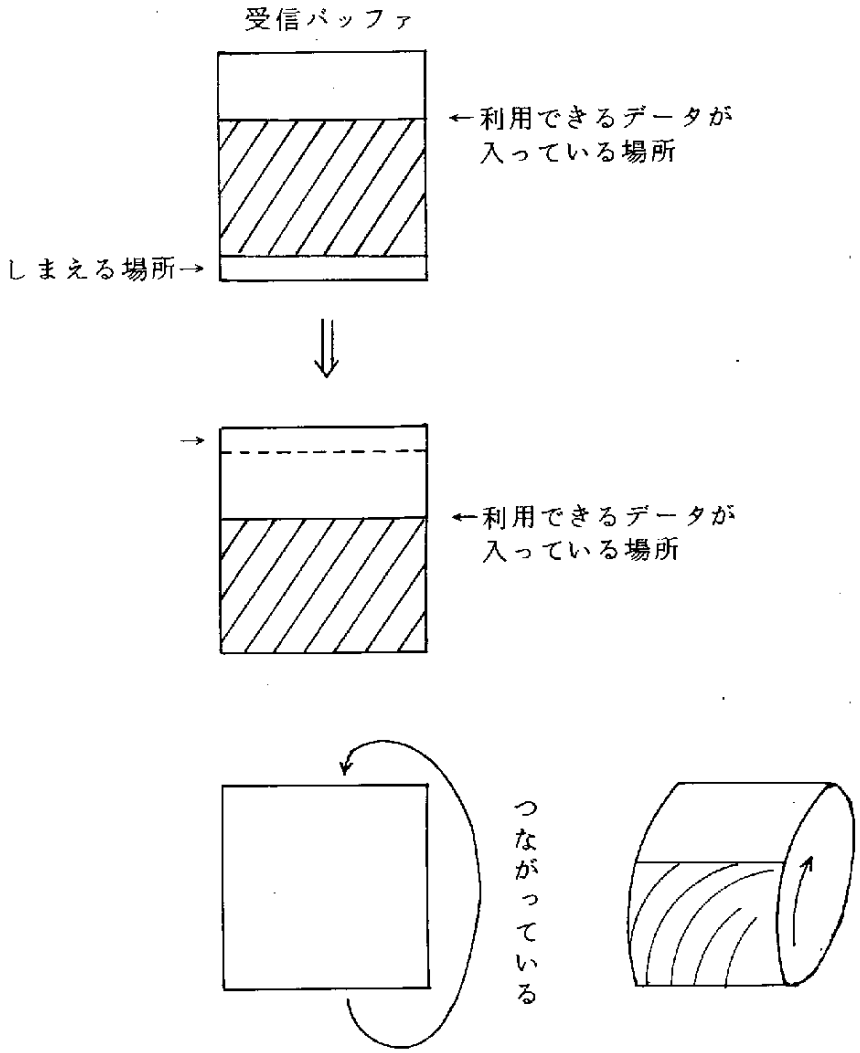
一方、プログラムでデータが必要になった場合には受信バッファの先頭にあるデータから取り出す。

受信バッファはＦＩＦＯ（First In First Out）バッファである。即ち先に受信したデータが先に取り出される。

ＦＩＦＯバッファではプログラムでデータを１つ取り出したあと、残りのデータをバッファの先頭につめなくてはならない。

データの転送をおこなわずに、バッファの先頭にあるデータを取り出すことが出来るようにするために、一般にはバレル・バッファという方式を採用している。

この方式ではバッファの終りまでデータを入力したあと、次のデータをバッファの先頭の方にしまうので、バッファはあたかもビールの樽をころがしているように見える。



受信プログラムではバッファの外に次の情報を用意する。

a) 入力ポインタ（INPTR）

受信したデータをバッファのどこにしまえるか示す情報、最初は0

b) 出力ポインタ（OUTPTR）

利用できる最初のデータが入っている場所を示す情報、最初は0

c) 受信したバイト数（INCH）

最初は0。受信する度に＋1し、取り出した時に－1する。

d) バッファの空き数（EMPTY）

最初はバッファの大きさを示すバイト数。

受信する度に－1し、データを読み出す度に＋1する。

次の手続きは割込みによって起動する。

```
procedure  receive  interrupt〔n〕;
begin
  if empty!=0  then
    begin
        empty:=empty-1;
        inch  :=  inch+1;
        buffer〔inptr〕:read-data(port);
        inptr:=inptr  mod  size;
        (*  wrap  around  *)
    end
end;
```

一方、この受信バッファからのデータの読み出しは次の関数によっておこなう。

```
function  read:char;
begin
  while  inch=0  do  (*  nothing  *);
  read:=buffer〔outptr〕;
  inch:=inch-1;
```

```
    empty := empty + 1;
    outptr := outptr  mod  size
  end;
```

この関数は次のことをおこなう。

1) 受信バッファが空のときには受信バッファにデータが入るまで待つ。

2) 受信バッファにデータがある場合には、最初のデータを取り出す。

3) バッファに受信できるバイト数、受信バッファにあるバイト数を更新する。

4) 次に取り出すことができるデータがある場所を更新する。

次のデータがある場所を

```
outptr  mod  size
```

によって求めているので、バッファの終りに到達した時には自動的にバッファの先頭にもどる。

受信したバイト数、受信できるバイト数はreceiveとreadの両方で参照している。

readの説明2)で述べた処理を実行している時にデータを受信しreceiveが起動すると受信したバイト数と受信できるバイト数がシステム全体を通して合わなくなるので注意が必要である。

処理の方法については応用篇で説明することにしたい。

8.3 DMA方式による制御

DMA方式によるI／Oの制御はプログラムの割込み処理を含むので、この処理方法については応用篇で説明することにしたい。

8.4 I／Oシミュレータ

組込みシステムは特殊な入出力装置を使用するので、このシステムで使

用するプログラムはシステムが完成するまでテストに入れない。

このために使用する入出力装置に近い標準の入出力装置を使用して出来るだけ早い時期からテストに入れるようにＩ／Ｏシミュレータを使用する。

Ｉ／Ｏシミュレータは状況によってハードウェアでおこなう場合もあるし、ソフトウェアでおこなう場合がある。

ここでは8.1で述べているシリアル・インタフェースを開発中にＣＰ／Ｍのサブルーチンを Ｉ／Ｏシミュレータとして利用する方法を説明する。

```
FALSE   EQU     0
TRUE    EQU     NOT FALSE
TEST    EQU     TRUE
        IF      TEST
        MOV     E,A
        MVI     C,2
        CALL    5
        ENDIF
        IF      NOT TEST
        PUSH    PSW
WAIT:   IN      81H
        ANI     1
        JZ      WAIT
        POP     PSW
        OUT     80H
        ENDIF
        RET
```

テストしている時には、ＴＥＳＴ　ＥＱＵ　ＴＲＵＥ　でアセンブルし、実機用のプログラムをアセンブルする時にはＴＥＳＴ　ＥＱＵ　ＮＯＴ　ＴＥＳＴと変えてアセンブルすることが出来る。

昭和59年3月発行

発行所　財団法人　日本情報処理開発協会
東京都港区芝公園3丁目5番8号
機械振興会館内
Tel（434）8211（代表）

印刷所　株式会社　昌文社
東京都港区芝5丁目26番30号
Tel　（452）4931（代）