8086アセンブラASM86取扱説明書

(有)中日電工

目次

使い方 1
フォルダの作成 1
ソースプログラム（TEXTファイル）の作成 1
テキストエディタ 1
操作例 1
使用できる文字 3
ラベル 3
変数名 3
COMファイル 4
擬似命令 4
ORG 4
DB 4
DW 4
" " 4
; 4
: 4
＝ 4
数値について 5
8086のレジスタ 5
汎用レジスタ 5
CPUコントロールレジスタ 5
セグメントレジスタ 5
汎用ではない使い方 5
フラグレジスタ 6
CF キャリーフラグ 6
ZF ゼロフラグ 6
SF サインフラグ 6
OF オーバーフローフラグ 6
PF パリティフラグ 7
AF 補助キャリーフラグ 7
DF デスティネーションフラグ 7
IF インタラプトフラグ 7
TF トラップフラグ 7
セグメントレジスタ 7
セグメントとは 7
セグメントプリフィックス 9
メモリアドレスの指定方法 10
8086の命令 12
AAA ASCII Adjust to Add 12
AAD ASCII Adjust to Divide 12
AAM ASCII Adjust to Multiply 12
AAS ASCII Adjust to Subtract 13
ADC Add with Carry 13
ADD 13
AND 14
CALL 14
CALLF Call Far 15
CBW Convert Byte to Word 15
CLC Clear Carry flag 15
CLD Clear Direction flag 15
CLI Clear Interrupt flag 15
CMC Compiement Carry flag 15
CMP Compare 15
CMPSB／CMPSW Compare String Byte／Compare String Word 16
CWD Convert Word to Doubleword 17
DAA Decimal Adjust to Add 17
DAS Decimal Adjust to Subtract 17

DEC Decrement 17
DIV Divide 18
HLT Halt 18
IDIV Integer Divide 18
IMUL Integer Muitiply 19
IN 19
INC Increment 19
INT Interrupt 19
IRET Interrupt Return 20
JMP Jump 20
JMPF Jump Far 20
条件ジャンプ命令 20
JA Jump if Above 21
JNBE Jump if not Below and not Equal 21
JAE Jump if Above or Eual 21
JNB Jump if not Below 21
JNC Jump if not Carry 21
JB Jump if Below 21
JC Jump if Carry 21
JNAE Jump if not Above and not Equal 21
JBE Jump if Below or Eual 21
JNA Jump if not Abobe 21
JCXZ Jump if CX is Zero 22
JE Jump if Eual 22
JZ Jump if Zero 22
JG Jump if Greater 22
JNLE Jump if not Less and not Equal 22
JGE Jump if Greater or Equal 22
JNL Jump if not Less 22
JL Jump if Less 23
JNGE Jump not Greater and not Equal 23
JLE Jump if Less or Equal 23
JNG Jump if not Greater 23
JNE Jump if not Eual 23
JNZ Jump if not Zero 23
JNO Jump if not Overflow 23
JNP Jnmp if not Parity 23
JPO Jump if Parity Odd 23
JNS Jump if not Sign 24
JO Jump if Overflow 24
JP Jump if Parity 24
JPE Jump if Parity Even 24
JS Jump if Sign 24
LAHF Load AH from Flagregister 24
LDS Load pointer using DS 25
LEA Load Effective Address 25
LES Load pointer using ES 25
LOCK Lock bus25
LODSB／LODSW Load String Byte／Load String Word 25
LOOP 26
LOOPE Loop if Equal 26
LOOPZ Loop if Zero 26
LOOPNE Loop if not Equal 26
LOOPNZ Loop if not Zero 26
MOV Move 27
MOVSB／MOVSW Move String Byte／Move String Word 27
MUL Multiply 28
NEG Negative 28

NOP No Operation 28
NOT 29
OR 29
OUT 29
POP 30
POPF Pop Flagregister 30
PUSH 30
PUSHF Push Flagregister 31
RCL Rotate Left through Carry 31
RCR Rotate Righit through Carry 32
REP Repeat string operation 32
REPE Repeat string operation while Equal 32
REPZ Repeat string operation while Zero 32
REPNE Repeat string operation while not Equal 32
REPNZ Repeat string operation while not Zero 32
RET Return 33
RETF Return Far 33
ROL Rotate Left 33
ROR Rotate Righit 34
SAHF Save AH to Flagregister 34
SAL Shift Arithmetic Left 35
SHL Shift Left 35
SAR Shift Arithmetic Right 35
SBB Subtract with Borrow 36
SCASB／SCASW Scan String Byte／Scan String Word 36
SHR Shift Right 37
STC Set Carryflag 37
STD Set Directionflag 38
STI Set Interruptflag 38
STOSB／STOSW Store String Byte／Store String Word 38
SUB Subtract 38
TEST 39
WAIT 39
XCHG Exchange 39
XLAT Translate 40
XOR Exclusive Or 40
[参考]よく使われるMSDOSのファンクションコール 41
エラーコード 42

〒463-0067 名古屋市守山区守山2－8－14
パレス守山305
有限会社中日電工
TEL052-791-6254 Fax052-791-1391
E-mail thisida@alles.or.jp

8086アセンブラASM86操作説明書

使い方

WINDOWSのDOS窓で使用します。

AM188CPUボード付属CDROM（以下CDROMと略す）に入っているASM86．COMをハードディスクの適当なディレクトリ（フォルダ）にCOPYしてください。

フォルダの作成

適当な名前でよいのですがここでは、ASM86という名前のフォルダを作ってそこにASM86．COMプログラムをコピーします。

［マイコンピュータ］→［C:ドライブの順にマウスでクリックしてC:ドライブを開きます。

メニューバーの［ファイル］をクリックして［新規作成］→［フォルダ］の順にクリックします。

C:ドライブのウインドウの中に［新しいフォルダ］が作られます。その［新しいフォルダ］をクリックして選択しておいて［ファイル］→［名前の変更］の順にクリックすると［新しいフォルダ］の名前部分が変更可能になるので、ここでキーボードからASM86と入力して［Enter］を押します。

これでASM86という名前のフォルダがハードディスクに作られました。この［ASM86］フォルダをクリックしてASM86ウインドウを開いておきます。

次にCDROMドライブにCDROMをいれたあと、また［マイコンピュータ］をクリックして今度はCDROMディスクをダブルクリックします。するとCDROMのウインドウが開きます。中にASM86．COMプログラムアイコンが見えます。

このアイコンをマウスで左クリックしたまま離さないように注意してそのまま先ほどのASM86ウインドウまで引っ張って行って、そこで離します。コピーが完了してASM86ウインドウにASM86．COMアイコンが表示されます。

ソースプログラム（TEXTファイル）の作成

まずWINDOWSのメモ帳などの適当なテキストエディタでアセンブラソースプログラムを作成します。

作成したソースプログラムファイルはどのディレクトリ（フォルダ）に置いてもよいのですが、ASM86．COMと同じディレクトリに保存した方がよいでしょう。ここでは仮にTEST1．TXTというファイル名で保存したことにします（拡張子はTXTでなくてもつけなくてもかまいません。WINDOWSのメモ帳を使うと ．TXTがつけられます）

ASM86．COMのあるディレクトリに移動して、

ASM　TEST1．TXT

と入力して［Enter］を押すとアセンブル作業が開始され、正常に完了するとアセンブルリストファイルとCOMファイルが作成されます（この例では、TEST1．LST、TEST1．COMというファイル名で保存される）。

もしエラーがあるとアセンブル作業は中止されてエラーコードが表示されます。

テキストエディタ

WINDOWSのメモ帳（Notepad）でもよいのですが、お勧めはTeraPadです。

［注意］

テキストエディタでソースプログラムを作成したときは、最後は必ず改行して、できれば ；（セミコロン）で終わってください。改行しないで終了するとASM実行時に最終行でエラーが発生します。

［例］

```
END：MOVAH，4C
      INT　21              ……ここで［Enter］を入力
 ……カーソルがここに来てから終了する


END：MOVAH，4C
      INT　21    ……カーソルがこの位置のまま終了すると、正しくアセンブルされない
```

操作例

文法の説明は後にして、具体的な操作の例を示します。例を参考にして操作してください。

1．テキストエディタでソースプログラムを作成します。

```
ORG＝100
；
JMP　START
；
MSG1：“オハヨウ、フェルプスクン＄
MSG2：“コノテープハジドウテキニショウメツスル＄
```

```
CRLF:DB  0D
DB  0A
;
START:MOV  AH, 09;    display  STRING
MOV  DX, MSG1
INT  21
MOV  BX, *CRLF
MOV  DL, [BX]
MOV  AH, 02;              display  DL
INT  21
MOV  DL, [BX+1]
INT  21
MOV  DX, MSG2
MOV  AH, 09
INT  21
MOV  AH, 4C;          return  to  SYSTEM
INT  21
```

[注]プログラムの中で何箇所か INT 21 が使われています。これはMSDOSのファンクションコールです。
MSDOSではいろいろな働きをするシステムサブルーチンをこのような形でコールして使うことができます。
ファンクションコールについてはこの説明書の後ろの方で簡単に紹介しています。詳しくはMSDOSの解説書を参照して下さい。

2. プログラムを書き終わったら、SAVEします。ASM86. COMと同じディレクトリ(フォルダ)にSAVEした方が操作が楽です。ここではSAMPLE. TXTという名前でSAVEします。

3. DOS窓を開きます。

4. ASM86. COMがあるディレクトリへ行きます。

5. ASM SAMPLE. TXT[Enter]と入力する。

もしエラーがなければENDと表示されたあと、プログラムの開始アドレスと終了アドレスが表示されてアセンブル作業が完了し、SAMPLE. LSTファイルとSAMPLE. COMファイルが同じディレクトリ(フォルダ)に作成されます。

```
1999/1/7   15:17   SAMPLE.TXT
[00001]                         ORG=100
[00002]                         ;
[00003] 0100   EB2990             JMP START <012B>
[00004]                         ;
[00005] 0103   B5CAD6B3CCAA   MSG1:"ｵﾊﾖｳﾌｪﾙﾌﾟｽｸﾝ$
               D9CCDFBDB8DD
               24
[00006] 0111   BAC9C3B0CCDF   MSG2:"ｺﾉﾃｰﾌﾟﾊｼﾞﾄﾞｳﾃｷﾆｼｮｳﾒﾂｽﾙ$
               CABCDEC4DEB3
               C3B7C6BCAEB3
               D2C2BDD924
[00007] 0129   0D               CRLF:DB 0D
[00008] 012A   0A               DB 0A
[00009]                         ;
[00010] 012B   B409              START:MOV AH,09;     display STRING
[00011] 012D   BA0301             MOV DX,MSG1
[00012] 0130   CD21              INT 21
[00013] 0132   BB2901            MOV BX,*CRLF
[00014] 0135   8A17              MOV DL,[BX]
[00015] 0137   B402              MOV AH,02;              display DL
[00016] 0139   CD21              INT 21
[00017] 013B   8A5701            MOV DL,[BX+1]
[00018] 013E   CD21               INT 21
[00019] 0140   BA1101            MOV DX,MSG2
```

```
[00020] 0143   B409            MOV AH,09
[00021] 0145   CD21            INT 21
[00022] 0147   B44C            MOV AH,4C;          return to SYSTEM
[00023] 0149   CD21            INT 21
0100-014A
CRLF          =0129   MSG1          =0103   MSG2          =0111   START         =012B
```

LSTファイルはテキストファイルなので、プリンタに出力することができます。

使用できる文字

半角の英文字、数字、ASCIIコードで表せる記号、カタカナ以外は使用できません。全角文字、ひらがな、漢字はコメント行でも使用できません。

命令には英大文字以外は使用できません。小文字を使用するとエラーになります。

ラベルや変数には英大文字および数字と ＿ 以外は使用できません。先頭の１文字は英大文字に限ります。ラベル、変数の長さは最大19文字です。

コメントや文字列には英、数、カナ、記号が使用できます

ラベル

ラベルは命令の前に :（コロン）をつけて使います。

```
LOOP：MOV  [DI], AX
       INC  DI
       DEC  CL
       JNZ  LOOP
```

のように使います。普通はJMP命令などの飛び先やCALL命令で参照されるサブルーチンの先頭に置きます。またストリング命令で参照されるデータ列の先頭位置を示すのにも使います。

ADD CX, [XYZ]のように使うこともできます。この場合にはラベルXYZの置かれたメモリアドレスにあるデータ（2バイト）がCXに加算されます。

MOV DX, XYZのように使うこともできます。この場合にはラベルXYZの置かれたメモリアドレスがDXに代入されます。

```
MOV：MOV  AX, DX
AX：ADD  BX, DI
```

のように命令やレジスタ名と同じラベルをつけることもできますが、間違いやすいので避けたほうがよいでしょう。

なおセグメントレジスタCS, DS, ES, SSをラベル名にすることはできません（CS：、DS：、ES：、SS：はセグメントプリフィクスになります）。

変数名

変数名はワークアドレスや2バイト（1バイトは不可）の数値の代用として使います。

```
COUNT＝B038        ①
MAX＝100           ②
MOV  DI, ＊COUNT   ③
MOV  [DI]W, MAX    ④
MOV  AX, MAX       ⑤
```

のように使います。

①②は8086の命令ではなくて、アセンブラの約束事（擬似命令）で命令コードに変換はされません。

①②はプログラムのどこにおいてもよいのですが普通はプログラムの先頭部分にまとめて書くようにします。

命令の中でB038や100などの16進データ、16進アドレスの代わりに使います。したがって③～⑤は

```
MOV  DI, B038      ⑥
MOV  [DI]W, 100    ⑦
MOV  AX, 100       ⑧
```

というように直接、値を書くのと同じです。

プログラムの中で繰り返し使用するような場合に、③～⑤の方が⑥～⑧よりも、後から見たときにわかりやすい利点があります。

③はMOV DI, COUNT と書くとエラーになります。この場合アセンブラは、MOV DI, C と解釈します（DIレジスタに16進数000Cを代入）。

その後にOUNTが残るためここでエラーになってしまいます。

一般のアセンブラではこのようなトラブルを避けるため数値を示す場合には先頭に0（ゼロ）をつけたり、＆Hや＄をつけるか最後にHをつけたりします。

個人的な好みで申し訳ないのですが、このアセンブラでは反対に変数名のうちA～F、SIで始まる場合には前に＊をつけるルールにしています。
それ以外にレジスタ名（SI、SS）で始まる変数名にも前に＊をつけます。
①②は変数名の定義文でこの位置に変数名以外が置かれることはないので、＊をつける必要はありません。
この＊のルールはラベルについても同様です。
ただCALL命令やJMP命令などでは直接16進アドレスを指定せず、いつもラベルを使用するため、＊を使う必要はありません（CALL　＊ABC　とする必要はない。CALL　ABC　でよい）。
ラベルと変数名とは同じものです。使われ方が少し異なっているため、便宜的に区別して説明していますが、16進データやアドレスの代わりに使う点で全く同じものです（このアセンブラでは同じものとして扱っています）。
ABC：MOV　AX，BX　のように定義したときラベルであるといい、ABC＝1234　のように定義したとき変数名といいますが扱いは全く同じで両者を区別する制約はありません。
④⑦で[DI]ではなくて[DI]Wになっているのは、メモリに定数を代入したり（MOV）、メモリ内容と定数とを比較、演算する（ADD、ORなど）命令では8ビット（1バイト）と16ビット（2バイト）の2通りがあってそれを区別するため、1バイトは[　]の後ろにB（BYTEの意味）をつけ、2バイトのときはW（WORDの意味）をつけます。
⑦は2バイトの数値なのでWをつけなくてもよさそうなものですが、間違いを避けるため一律にこのルールを適用します。
このルールが必要なのはメモリと定数（変数、ラベルも同じ）との場合及びメモリ単独の場合だけで、MOV　[DI]，AX　のようにメモリとレジスタの場合にはアセンブラがレジスタタイプから判断するため、B、Wをつける必要はありません。

COMファイル

MSDOSで実行されるプログラムファイルにはCOMファイルとEXEファイルがあります。
このアセンブラではCOMファイルを生成します。
両者の違いなどについては、ここでは説明を省きます。
COMファイルではセグメントの扱いに制約がある、と解説書などにありますが、通常のプログラムでは全く支障ありません。
アセンブラによって作成され、保存されたファイルには　．COM　の拡張子が付加されます。
．COM、　．EXE　ファイルはコマンドと同様の操作で実行されます（たとえばこのASM86．COMはASM86[Enter]で実行されます）。
COM形式のプログラムはセグメント（64KB）をまたいで作成することはできませんが、そんなに大きなプログラムをアセンブラで開発することはまず無いと思いますからCOMファイルで十分でしょう。
ちなみにこのASM86．COMでも8KBに満たないサイズです。
なおプログラムはセグメントをまたぐことができませんが、他のセグメントのプログラムにJMPしたり、CALLすることはできます（farジャンプ、farコール）。
またデータエリアとして他のセグメントを使用することも問題ありませんし、セグメント管理さえしっかり行えば64KB以上のデータエリアをアクセスすることもできます。

擬似命令

8086の命令ではありませんが、プログラムを書く上で便利なように、アセンブラが用意したいくつかの命令があります。このアセンブラでは以下の命令があります。

ORG

プログラムの開始アドレスを指定します。COMファイルは100H番地からはじめるという約束がありますから、プログラムの先頭は必ず　ORG＝100　にしなければなりません。
プログラムの途中にORG＝××××　と書くことで、それ以後のプログラムを××××から割り付けることができますが、COMファイルでは余り意味がありませんし、間違いのもとですから途中では使用しないほうがよいでしょう。

DB

1バイトのデータを定義します。DB　41　と書くとそのメモリ位置に41という値が書きこまれます。

DW

2バイトのデータを定義します。DW　1234　と書くとそのメモリ位置には下位バイト34が書かれ、その次のアドレスに上位バイト12が書きこまれます。

" "

” ”で挟んだ文字列を書くとそのメモリアドレスから後ろに、その文字列のASCIIコードが書き込まれます。
文字列は最大36文字で “ を文字列に含むことはできません。

;

行の先頭や命令の最後に　;（セミコロン）を書くとそれより後ろはコメントとみなされます。
後ろに何を書いてもアセンブラは無視します。

:

ラベルを書くときに使います。 XYZ：ADD　AX，CX　のように使います。
:はセグメントプリフィックスとしても使いますが、同じ記号でも使い方は全く異なります。

＝

変数を定義するのに使います。
一般のアセンブラではEQUを使いますが、＝の方がわかりやすいので、このアセンブラでは＝を用いています。

ABC＝1234 というように使います。
変数に代入できるのは2バイトの16進数のみです。
1バイトの16進数や10進数は代入できません。

数値について

このアセンブラでは10進数は使いません。
すべて16進数のみです。ABC＝100 とか MOV AX, 56 と書いた場合にはそれぞれ、0100H、0056Hの16進数として扱われます。
先頭の0は省略してもかまいません。
MOV AL, 56 と書いた場合には56Hなのでエラーにはなりませんが、MOV AL, 100 はエラーになります（100Hは16ビットの数なので8ビットレジスタALにはMOVできない）。

8086のレジスタ

8086には以下のレジスタがあります。

汎用レジスタ

AX （AH＋AL） アキュムレータ
BX （BH＋BL） ベースレジスタ
CX （CH＋CL） カウンタ
DX （DH＋DL） データレジスタ
SI ソースインデックスレジスタ
DI デスティネーションインデックスレジスタ
BP ベースポインタ
SP スタックポインタ

CPUコントロールレジスタ

F フラグレジスタ
IP インストラクションポインタ

セグメントレジスタ

CS コードセグメント
DS データセグメント
ES エクストラセグメント
SS スタックセグメント

これらのレジスタはすべて16ビットです。AX～SPを汎用レジスタといいますが、SPはスタックアドレスを管理しているので汎用レジスタとして使うことは無いと思います。
その意味ではAX～BPを汎用レジスタとした方が良いでしょう。
AX～BPは普通の命令（MOV、ADD、OR、INCなど）では同じように使うことができます。
AX、BX、CX、DXはそのまま16ビットとして使う以外に、上位8ビット、下位8ビットに分けて使うことができます。
AXレジスタの他にAHレジスタとALレジスタがあるのではなくて、AXレジスタの上位8ビットと下位8ビットを分離して別々に処理することができます。
たとえば MOV AH, 12 と MOV AL, 34 という命令を実行すると、AXレジスタには1234というデータが格納されます。
ここで INC AH を実行するとAHレジスタの内容が＋1されますから、AHレジスタの内容は13になります。
AXレジスタとして中身を確認すると、1334になっています。
このようにAXレジスタはAHレジスタとALレジスタという8ビットレジスタをつないだものだと理解してください。
プログラムの中でAXとして使うかAH、ALとして分けて使うかは全く自由で特別の宣言は不要です。
命令の中でAXと書けば16ビットレジスタとして処理され、AHまたはALと書けば、8ビットレジスタとして処理されます。
BX～DXも同様です。

汎用ではない使い方

汎用レジスタは汎用として使う以外に、ある命令では特別のレジスタとしても使われます。
AXレジスタ、DXレジスタは除算（DIV）や入出力（IN、OUT）などで演算レジスタ、データレジスタとして使われます。
これらの命令では他の汎用レジスタで代用することはできません。
BXレジスタはメモリアドレスを指定する場合のベースアドレスとしてBP、SI、DIと同じように使うことができますが、AX、CX、DXをこの用途に使うことはできません。
CXレジスタはストリング命令で回数カウンタとして使われますが他の汎用レジスタをCXの代わりに使うことはできません。
SI、DIレジスタはメモリアドレスを指定する場合のインデックスレジスタとして用いられます。
この用途にはBXレジスタ、BPレジスタも使うことができますが、AX、CX、DXをこの用途に使うことはできません。
またストリング命令（MOVS、COMPSなど）ではSIおよびDIレジスタを使いますが他の汎用レジスタでは代用できません。

フラグレジスタ

Fは16ビットのレジスタですが各ビットが独立していてそれぞれのビットに異なる情報（通常は直前に行われた演算の結果）が格納されています。

フラグは条件ジャンプ命令（JZ、JNCなど）で参照され、プログラムの流れをコントロールする重要な役目を担います。

また特別な命令や動作のコントロールにも使われます。

命令の実行によって自動的に設定されるフラグもありますが、いくつかのフラグはフラグを操作する命令でセット、リセットすることもできます。

フラグレジスタをスタックに保存するなどの操作以外では、通常フラグレジスタ全体をまとめて扱うことはありません。

下の図は参考までに示しましたが、各フラグのビット位置を覚える必要は全くありません。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	←ビットNo.
×	×	×	×	OF	DF	IF	TF	SF	ZF	×	AF	×	PF	×	CF	

×マークのビットは未使用

CF キャリーフラグ

ADDなどの演算の結果最上位ビットからオーバーフローしたり、SUBなどでボローが生じたときにセットされます。

またシフトやローティト命令ではキャリーを含めてシフト、ローティトを行うこともでき、特殊な目的に利用することができます。

条件ジャンプ命令で参照されます。

CFはCLC命令でクリア、STC命令でセットされ、CMC命令で反転します。

もっとも重要でよく使われるフラグです。

INC、DEC命令ではCFは変化しません。

ZF ゼロフラグ

演算の結果がゼロになったときにセットされます。

比較命令で一致したときもセットされます。

条件ジャンプ命令で参照されます。

CFと同様、重要でよく使われるフラグです。

SF サインフラグ

演算の結果がマイナス（最上位ビットが1）になったときにセットされます。

条件ジャンプ命令で参照されます。

OF オーバーフローフラグ

演算の結果がオーバーフローしたときにセットされます。

条件ジャンプ命令で参照されます。

SF、OFは演算結果を符号付の数とみなしてセット、リセットされます。

これに対してCFは符号無しとみなしてセット、リセットされます。

8ビットの数は16進で00～FFです。

これは10進で示すと0～255になります。

これが符号無しの数です。

同じ8ビットを符号付で示す考え方があります。

その場合には8ビットの最上位ビット（ビット7）を符号ビットとみなしてしまいます（16ビットならビット15が符号ビット）。

16進の00～7Fが0～＋127になります。

80～FFが－128から－1になります。

ここがよくわからないと言う方がいるかもしれません。

次のように考えてください。

00から1を引くと16進ではFFになりますからこれが－1です（これがよくわからない方はFF、2進表現で11111111の最下位ビットに1を加算してみてください。全部のビットが0になります。＋1すると0になる数は－1です）。

11111111に1を加算するとオーバーして8ビットのさらに上位のビットに1が立つではないかと言う方もいるでしょう。

8ビット符号無しの数ではオーバーしていますからCFがセットされます。

しかし符号付ではオーバーフローにはなりません。

これは同じ8ビットの数を下のような数直線に当てはめて扱っているからです。

16進数	80	FF 00	7F
符号付10進数	－128	－1　0	＋127

16進8ビットの最大値であるFFが符号付の数では真中に来ています。これがFFに＋1してCFがセットされてもOFがセットされない理由です。

同じ理由から7Fに＋1すると結果は80になって、CFはセットされませんが、OFフラグはセットされます。
符号付の数として扱うか、符号無しの数と考えるかは、プログラマの自由です。
結果のフラグについては2通り（符号付、符号無し）のフラグ（CF、SF、OF）が同時にセット、リセットされます。
ALに7Fが入っているときADD　AL，1を実行すると、ALは80になって符号付ではSF、OFがセットされますが符号無しで考えるとキャリーやボローは発生していませんからCFはセットされません（リセットされます）。

PF　パリティフラグ

演算結果の下位8ビット（16ビットの計算でも上位8ビットは無視される）の中に1のビットが偶数個あるときにセットされます。

AF　補助キャリーフラグ

8ビット数の下位4ビットのみに注目して、演算の結果下位4ビットでキャリーやボローが発生したときにセットされます。
BCD数の演算で利用されます。

DF　デスティネーションフラグ

MOVS、COMPSなどのストリング命令で使用します。
DFがクリアされているとストリング命令の実行後にアドレスを管理しているインデックスレジスタが＋1または＋2されます。
DFがセットされているとストリング命令の実行後にインデックスレジスタが－1または－2されます。
DFはCLD命令でクリアされSTD命令でセットされます。

IF　インタラプトフラグ

割り込みの許可、不許可をコントロールします。
IFがセットされているとマスク可能割り込みが受けつけられます。
IFがリセットされていると割り込みは受けつけられません。
IFはCLI命令でクリアされ、STI命令でセットされます。

TF　トラップフラグ

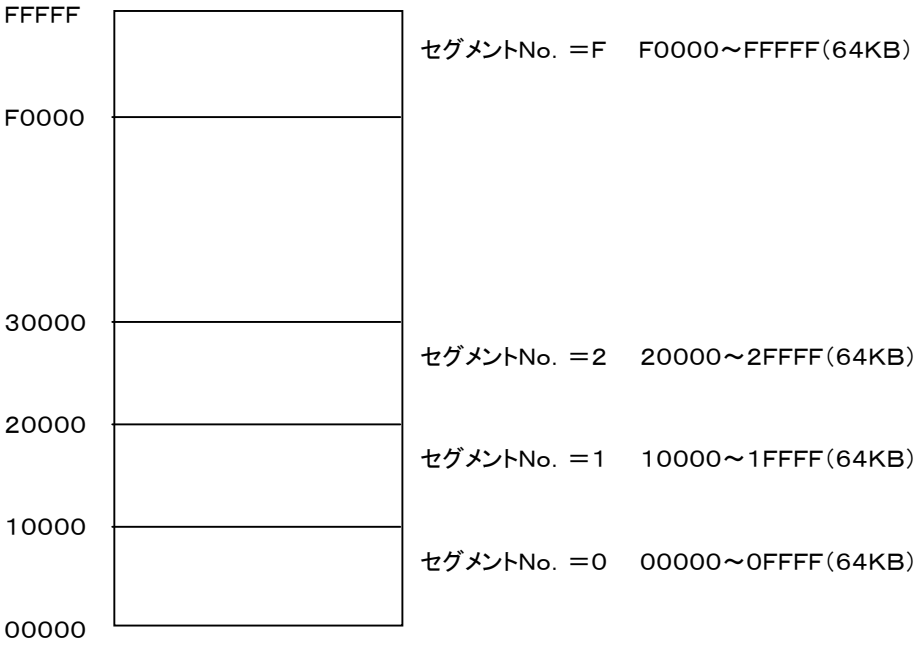
このフラグは命令のステップ実行に利用されます。
デバッグプログラムで使用します。
通常のプログラムでは使用しません。

セグメントレジスタ

セグメントを管理するレジスタで名前の通り、CSはプログラムの書かれているコードセグメントのセグメントアドレスを管理します。
DSは命令の対象になるデータメモリ領域のセグメントアドレスを管理します。
SSはスタックの置かれるセグメントを管理します。
ESはストリング命令で使用される他、DSの代用として使用されたりします。

セグメントとは

初めての方でよくわからないという方は読み飛ばしてください。
特別の命令以外ではセグメントについて理解していなくても、プログラムを書くことができます。
セグメントは、8086が8080やZ80と同じ16ビットレジスタしかもっていないのに、64KB以上のメモリをアクセスするための方法として考えられたものです（と思います）。
16ビットでアクセスできるメモリアドレスは2の16乗＝65536（64KB）が上限です。
8086はこの64KBをひとつのメモリ単位（セグメント）として扱うように命令が作られています。
仮に1MB（1024KB）のメモリ空間を64KBごとに区切って16個のセグメントとします。
するとこの16個のセグメントを指定するには4ビットあれば足りることになります（次ページ図参照）。



今、プログラム命令を最初のセグメントに書く事にし、データ領域をその次のセグメントに、そしてスタックをさらに次のセグメントに置き、残りのセグメントは予備とします。そしてそれぞれのセグメントナンバーを管理するレジスタとしてCS、DS、SSを使います。

これがモデル的なセグメントの概念です。

しかしこの考え方にはかなりムダがあります。

8086を組込み用のプロセッサとして考える（Z80や8080の代用）ならそれでもよいのですが、もっと大きなマルチタスク、マルチユーザーのオペレーティングシステムとしても使うことを考えると、限られた貴重な資源であるメモリを64KBごとに機械的に分割して使用するのは賢いやり方とは思えません。

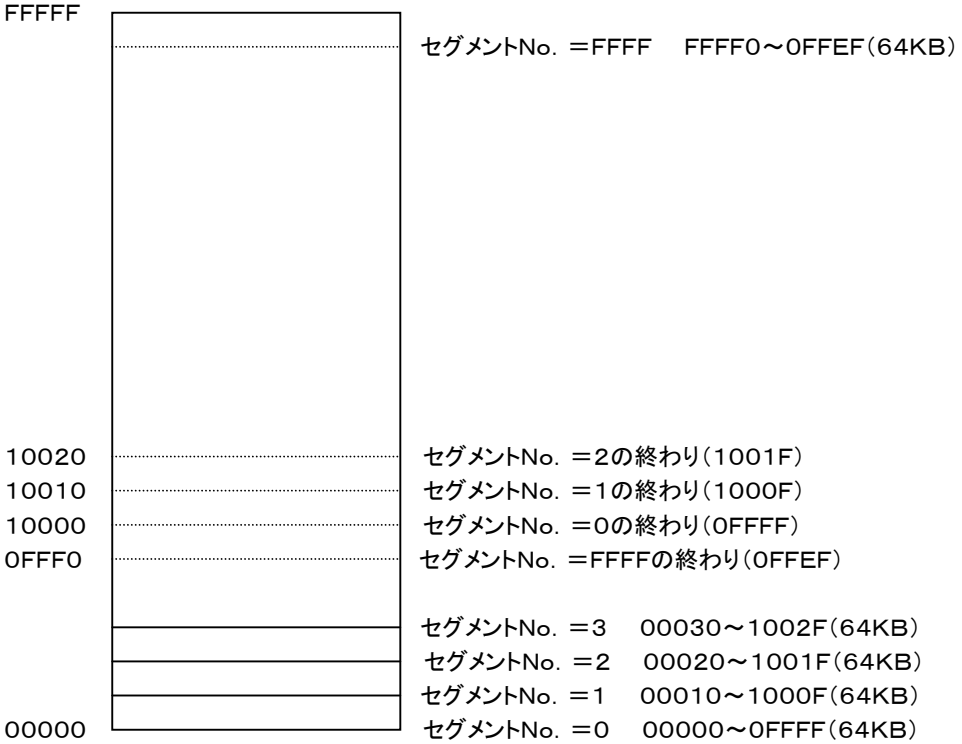
セグメントレジスタも他のレジスタと同じ16ビットであることを思い出してください。常識的なメモリサイズとして1MB（1024KB）のメモリ空間をアクセスすると考えたとき、モデル的なセグメントではそのうちわずか4ビットしか使用しませんでした。

そこで逆にこの16ビットのセグメントレジスタを最大限使用すると考えると、65536通りのセグメントを定義できることになります。

1MB／65536は16です。セグメントの開始アドレスに注目してください。64KBのサイズのセグメント65536個を1MBのメモリ空間に前から順に等間隔で（重なりを許して）割り付けたとすると、16バイトずつずらして配置できるはずです。

そしてこのようにして配置したセグメントに前から順にナンバーを与えていきます。

最初のセグメントが0000で次が0001というようにして最後はFFFFになって全部で65536個にナンバーが与えられます。



各セグメントの開始アドレスと実際のメモリアドレスの関係はどうなっているのでしょうか。

1MBのメモリ空間のアドレスは00000～FFFFFです。
最初のセグメント（ナンバー0000）の開始アドレスは実際のメモリ上では00000になります。次のセグメント（ナンバー0001）の開始アドレスは00010です。そして最後のセグメント（ナンバーFFFF）の開始アドレスはFFFF0になります（このセグメントは16バイトでメモリの最後になってしまいますが、こういうときは残りは先頭のメモリアドレス00000にもどって割り付けられます）。
このようにセグメントを示す数値（セグメントレジスタの値）とメモリ上でのそのセグメントの開始アドレスは4ビットシフトしたものになっています。
セグメントレジスタの値はメモリアドレス（00000～FFFFF。20ビットで示す）の上位16ビットに一致します。
セグメントの開始アドレスはそのときの20ビットの下位4ビットを0にしたアドレスになります。
一般的な命令でメモリを示す場合にはデータセグメントレジスタ（DS）の値がベースになります。
たとえば ADD ［BX］, AL という命令でBX＝1234だったとすると、Z80ならメモリアドレスも同じ1234番地です。
8086ではこのときのDSの値を16倍したアドレス＋BXレジスタ値、がメモリアドレスになります。
DS＝8000だったとすればメモリアドレスは80000＋1234＝81234になります。
上で説明したようにDSは0000～FFFFのどの値でも置くことができるので、DS＝5678などの場合も考えられます。
このときは56780＋1234＝579B4になります。
しかし、幸いこのアセンブラでは、普通の場合、DSの値や実際のメモリアドレスについて意識する必要は全くありません。
このアセンブラで作成されるCOMファイルについて考えてみます。
COMファイルでは実行開始時には各セグメントは同じメモリ空間に置かれます。CS＝DS＝SS＝ES。
そのセグメントアドレスはMSDOSが空いているメモリ空間に勝手に割り付けてしまいます。
つまりプログラム開始時点でCS（＝その他のセグメント）のアドレスがどこに置かれるかをあらかじめ知ることはできません。
どこになるのかは不明なのですが、CS＝DS＝SS＝ESでセグメントの大きさは64KBなのですから、とにかく使えるメモリはプログラム＋データ＋スタックを合わせて64KBなのだと考えてしまえばセグメクトを意識する必要はなくなります。
プログラムはセグメントの前の方（0100番地）から割り付けられますし、スタックは逆にセグメントの終わり（FFFF）から前に割り付けられますから、作業用のデータエリアはこの中間に置くように考えます。
たとえばプログラムサイズが10KBもあれば足りると考えたならば、そしてもしデータなどのエリアとして16KBくらいあればよいというのならば、プログラムは0100～2FFF（約12KB）におさまるので、データ用のアドレスとして3000～の部分を使えばよいことになります。
セグメントについては全く考える必要はありません。（普通のプログラムならこの考え方で済んでしまうはずです）。
セグメントを意識しなければならないプログラムとはどんなプログラムでしょうか？
プログラムとデータエリアを合わせると64KBを超えてしまう場合にセグメントが必要になってきます。
しかしこの場合でも本来のセグメントのあり方である、16バイトずつずらして配置するという部分はあまり活用できず、最初に説明した64KB毎に配置するモデル的な割り付け方をすることになります。
プログラムに32KBを使い、データに64KBを使うと考えた場合、DSにCSから32KB後ろの値をあたえてもよいと考えてしまうと、スタックがデータ領域の真中に置かれることになってSSをまた別のエリアにおかなければいけないことになってしまいます。そこでCS＝SSとしてプログラムとスタックに同じセグメント64KBを割り当てて、その後ろにデータセグメントを置くようにします。
CSがどこに割り当てられるかわからない状態でこれを実現するにはどうしたらよいでしょうか。
以外と簡単でプログラムの先頭で次のように書けばよいのです。

```
MOV  AX, CS       CSの内容をAXにコピーする
ADD  AX, 1000     AXに1000を加算する
MOV  DS, AX       DSにAX（CS＋1000を代入する。メモリアドレス上ではDSアドレス＝CS先頭アドレス
                  ＋10000の値となる）
MOV  ES, AX       必要ならESも更新する
```

もしもデータ領域として64KB以上を必要とする場合にはプログラム中で、DSやESの値を±1000することでデータ領域を切り替えることができます。
なお余りに大きなデータ領域を確保すると、システムのディスプレイ領域などと重なってしまう場合があります。すでに説明したように、プログラムがどのアドレスに割り付けられるかは指定できなくてMSDOSに任せてしまうことになります（通常20000番地台に割り付けられることが多いようです）からシステムのエリアと重ならないように余裕をもって設計する必要があります。

セグメントプリフィックス

上で考えたようにデータエリアとして64KBを超えるエリアを扱う場合にはセグメントに対する理解が必要になってきます。
例として、2つのメモリブロック（各64KB）の比較について考えてみます。Aブロックは40000～4FFFFでBブロックは60000～6FFFFであるとします。
比較するプログラムは次のように書けます。

```
MOV  AX, 4000
MOV  DS, AX
ADD  AX, 2000
MOV  ES, AX
MOV  CX, 8000
MOV  SI, 0
MOV  DI, SI
```

```
CLD
LOOP:CMPSW
JNZ  TIGAU
DEC  CX
JNZ  LOOP
```

ところでこの後の処理で(たとえばラベルTIGAUのルーチンで)、Bブロックのメモリを参照したいと考えます。

```
MOV  AL, [DI]
```

と書きたいところですが、これでは参照できません。なぜならすでに書いたように8086は一般の命令ではDSで指定するメモリブロックをアクセスするように働くからです。DS＝4000と指定しているため、[DI]はDSベースの[DI]になってしまうのです。COMPSWなどのストリング命令だけが例外的にESベースのメモリアクセスを行います。
したがってここでは

```
MOV  AX, 6000
MOV  DS, AX
MOV  AL, [DI]
```

としなければいけません。しかしBブロックへのアクセスはここ1箇所だけで、このあとはまたAブロックをアクセスすると考えると、またDSに4000を入れ直さなければならないことになります。
あるいは次のような方法も考えられます。

```
MOV  AX, 6000
PUSH  DS
MOV  DS, AX
MOV  AL, [DI]
POP  DS
```

もっと良い方法はないのでしょうか？
このような場合にセグメントプリフィックスを使用するとすっきりします。次のように書きます。

```
ES:
MOV  AL, [BX]
```

このES:はラベルではありません。「この次に書かれた命令のメモリアクセスはESベースで行う」という宣言なのです(次に書かれた1命令に限り有効です)。これをセグメントプリフィックスといいます。

```
ES:MOV  AL, [BX]
```

というように書くアセンブラが多いようですが、このアセンブラでは上の例のようにプリフィックスと次の命令は2行に分けて記述します。
セグメントプリフィックスには次の4種があります。

```
CS:
DS:
ES:
SS:
```

メモリアドレスの指定方法

すでに説明したようにメモリに対して演算を行うためのアドレスは通常はDSレジスタで指定する64KBのセグメント内のアドレスを0000～FFFFの範囲の数で指定します。普通はDSの値は意識する必要は無いので、16ビットのアドレス値のみでメモリを指定することになります。
アセンブラではメモリは[ ]で示します。
命令によっては8ビットのメモリか16ビットのメモリなのかをそれぞれ[ ]B、[ ]Wのように後ろにB、Wをつけて明示しなければいけません。

①[aaaa]

[1234]のようにメモリアドレス(16進数)で直接示します。[0123]や[0034]は[123]、[34]のように前のゼロを省略できます。10進数は扱えません。

②[nnn]

[XYZ]のように変数で書きます。アセンブラによってマシン語に変換された段階ではXYZに代入されている値が、メモリアドレスとして与えられます。XYZ＝3456のとき[XYZ]は[3456]と書くのと同じです。
[＊ABC]、[＊SIZE]のように最初の文字がA～F、SIで始るときは前に＊をつけなければいけません。この例ではもし＊をつけないと[0ABC](16進アドレス)、[SI…](エラー)と解釈されてしまいます。

③[SI]、[DI]、[BX]

SI、DI、BXにアドレスを入れて指定します。この3通りしかありません。[AX]、[CX]、[DX]はありません。また間違えやすいのですが[BP]もだめです。

④[BX＋SI]、[BX＋DI]、[BP＋SI]、[BP＋DI]

BXまたはBPにベースになるアドレスを入れておいて、SI、DIにそのベースアドレスからの増分を指定します。上記の4通り以外はありません。

⑤[SI＋D8]、[DI＋D8]、[BX＋D8]、[BP＋D8]

SI、DI、BX、BPをベースアドレスにして、そこからの増分を8ビットで指定します。D8は8ビットの数、01～FFの16進数を直接指定することを示しています。この8ビットの16進数は符号付の数として扱われます。01～7Fのときはアドレスを示すレジスタ（SI、DI、BX、BP）の値に01～7Fを加算してメモリアドレスとします。

80～FFのときはレジスタの値から減算して（レジスタの値にFF80～FFFFを加算する）アドレスが求められます。

このアセンブラでは10進数は扱わないので、MOV ［BX＋C3］, AXのように直接16進数で表記します。

⑥[BX＋SI＋D8]、[BX＋DI＋D8]、[BP＋SI＋D8]、[BP＋DI＋D8]

BX＋SI、BX＋DI、BP＋SI、BP＋DIをベースアドレスにして、そこからの増分を16進8ビットで指定します。

⑦[SI＋D16]、[DI＋D16]、[BX＋D16]、[BP＋D16]

SI、DI、BX、BPをベースアドレスにして、そこからの増分を16ビットで指定します。D16は16ビットの数、0001～FFFFの16進数を直接指定することを示しています。この16ビットの16進数は符号付の数として扱われます。0001～7FFFのときはアドレスを示すレジスタ（SI、DI、BX、BP）の値に0001～7FFFを加算してメモリアドレスとします。

8000～FFFFのときはレジスタの値から減算して（レジスタの値に8000～FFFFを加算する）アドレスが求められます

⑧[BX＋SI＋D16]、[BX＋DI＋D16]、[BP＋SI＋D16]、[BP＋DI＋D16]

BX＋SI、BX＋DI、BP＋SI、BP＋DIをベースアドレスにして、そこからの増分を16進16ビットで指定します。

8086の命令

8086の命令をアルファベット順に説明します。
慣れないうちは機能別に説明した方がわかりやすいかもしれません。しかし少し慣れてくるとアルファベット順の方が扱い易くなります。はじめは前から順にざっと目を通してどんな命令があるか大体のところを頭に入れてください。あとは実際にプログラムを書くときに詳しく参照して下さい。
見易さを考えてJMP命令と条件ジャンプ(J××)はアルファベット順にはしてありません。
[書式]で使う記号は次の通りです。

reg　8ビットレジスタ(AH、AL、BH、BL、CH、CL、DH、DL)、16ビットレジスタ(AX、BX、CX、DX、SI、DI、SP、BP)
reg8　8ビットレジスタ(AH、AL、BH、BL、CH、CL、DH、DL)
reg16　16ビットレジスタ(AX、BX、CX、DX、SI、DI、SP、BP)
mem　8ビットまたは16ビットメモリ [BX]B、[DI]Wなど。
segr　セグメントレジスタ CS、DS、ES、SS
data　1バイトまたは2バイトの16進数値

AAA　ASCII Adjust to Add

この命令より前に実行された1桁のアンパックBCD数どうしの加算結果がALレジスタに入っているとして、演算後の10進補正を行います。1桁のBCD(2進化10進数)同士の加算では、0～9に0～9を加算するので計算結果は10進数の0～18になります。結果が0～9のときはAAAを実行しても結果に変化はありません。結果が10進数の10～18のときは補正が必要になります。
計算の結果ALが9より大きいか、AFフラグがセットされていると、ALに6が加算されたあとALの上位4ビットに0が入れられます。さらにAHの内容に1が加算されます。AFフラグとCFフラグは1になります。
AL＜9でAF＝0のときはAFフラグとCFフラグが0になります。
[例]

```
MOV  AX, 0309     AHに上位桁のBCD数がはいっていてもよい
ADD  AL, 9
```

という命令を実行するとALには16進数の12が入ります(8086は数値を16進数として扱います)。16進数の12は10進数の18ですから計算は正しく行われています。ただBCD同士の加算なので、結果もBCDで欲しいところです。
そこでこの後にAAAを実行します。その結果AXには0408が入ります。
[フラグ]
実行前にAL＞9またはAFフラグ＝1のとき、実行の結果AF＝1、CF＝1になります。それ以外のとき実行の結果AF＝0、CF＝0になります。
OF、SF、ZF、PFは不明です。
DF、IF、TFは変化しません。

AAD　ASCII Adjust to Divide

2桁のアンパックBCD数の上位1桁がAHに、下位1桁がALに入っているとき、このBCD数を16進数に直してAXレジスタに入れます。2桁のアンパックBCD数を1桁のBCD数で割るための準備として使われます。
[例]

```
MOV  AX, 0506
AAD                  この結果AX＝38になる(56D＝38H)
MOV  BL, 07
DIV  BL              結果はAL＝8になる
```

[フラグ]
実行の結果のAXの値によって、SF、ZF、PFがセット、リセットされます。
OF、AF、CFは不明です。
DF、IF、TFは変化しません。

AAM　ASCII Adjust to Multiply

この命令より前に実行された1桁のアンパックBCD数どうしの乗算結果がALレジスタに入っているとして、演算後の10進補正を行います。1桁のBCD(2進化10進数)同士の乗算では、計算結果は10進数の0～81になります。乗算結果は16進数になりますから、ALには00～51が入っていることになります。これを補正して2桁のBCD数としてAXに入れます(AX＝0000～0801)。
[例]

```
MOV  AL, 6
MOV  BL, 9
MUL  BL              AL＝36になる(36H＝54D)
AAM                  AH＝05、AL＝04になる
```

[フラグ]
実行の結果のAXの値によって、SF、ZF、PFがセット、リセットされます。

OF、AF、CFは不明です。
DF、IF、TFは変化しません。

AAS　ASCII Adjust to Subtract

この命令より前に実行された1桁のアンパックBCD数どうしの減算結果がALレジスタに入っているとして、演算後の10進補正を行います。1桁のBCD(2進化10進数)同士の減算では、結果は10進数の－9～＋9になります。結果がマイナスのときはボローが発生するので、AFフラグ＝1になります。このときAASを実行するとBCD上位桁(AH)から1が引かれて、ALに結果のBCD数の下位1桁の値が入ります。

[例]

```
MOV  AX, 0304
SUB  AL, 8              AL＝FC(04－08の2進演算の結果が2の補数で入れられる)
AAS                     AX＝0206になる
```

[フラグ]

実行前にAFフラグ＝1のとき、実行の結果AF＝1、CF＝1になります。それ以外のとき実行の結果AF＝0、CF＝0になります。
OF、SF、ZF、PFは不明です。
DF、IF、TFは変化しません。

ADC　Add with Carry

[書式]

①ADC reg1, reg2
②ADC reg, mem
③ADC mem, reg
④ADC reg, data
⑤ADC mem, data

第1オペランドと第2オペランドとCFフラグを加算して第1オペランドのregまたはmemに格納します。第2オペランドの値は変化しません。
①はreg1＝reg2＝8ビットまたはreg1＝reg2＝16ビットです。8ビットと16ビットの混合はできません。
⑤のmemの表記では
ADC [BX]B, data8
ADC [BX]W, data16
のようにB(Byte)またはW(Word)をつける必要があります。

[例]

```
①ADC  DL, AH
  ADC  BX, SI
②ADC  BH, [BX]     [BX]Bにする必要は無い
  ADC  CX, [DI]     [DI]Wにする必要は無い
③ADC  [BP], CL      [BP]Bにする必要は無い
  ADC  [SI], AX     [SI]Wにする必要は無い
④ADC  BL, 45
  ADC  DI, 3000
⑤ADC  [BP＋DI]B, 12   [ ]Bが必要
  ADC  [DI]W, 78AB    [ ]Wが必要
```

[フラグ]

計算の結果の値によってOF、SF、ZF、AF、PF、CFが変化します。
DF、IF、TFは変化しません。

ADD

[書式]

①ADD reg1, reg2
②ADD reg, mem
③ADD mem, reg
④ADD reg, data
⑤ADD mem, data

第1オペランドと第2オペランドを加算して第1オペランドのregまたはmemに格納します。第2オペランドの値は変化しません。
①はreg1＝reg2＝8ビットまたはreg1＝reg2＝16ビットです。8ビットと16ビットの混合はできません。
⑤のmemの表記では
ADD [BX]B, data8
ADD [BX]W, data16
のようにB(Byte)またはW(Word)をつける必要があります。

[例]
①ADD DL, AH
ADD BX, SI
②ADD BH, [BX]　[BX]Bにする必要は無い
ADD CX, [DI]　[DI]Wにする必要は無い
③ADD [BP], CL　[BP]Bにする必要は無い
ADD [SI], AX　[SI]Wにする必要は無い
④ADD BL, 45
ADD DI, 3000
⑤ADD [BP＋DI]B, 12　[ ]Bが必要
ADD [DI]W, 78AB　[ ]Wが必要
[フラグ]
計算の結果の値によってOF、SF、ZF、AF、PF、CFが変化します。
DF、IF、TFは変化しません。

AND
[書式]
①AND reg1, reg2
②AND reg, mem
③AND mem, reg
④AND reg, data
⑤AND mem, data
第1オペランドと第2オペランドのAND（論理積）を計算して第1オペランドのregまたはmemに格納します。第2オペランドの値は変化しません。
①はreg1＝reg2＝8ビットまたはreg1＝reg2＝16ビットです。8ビットと16ビットの混合はできません。
②③ではレジスタが8ビットのときはメモリ1バイトとの間で、レジスタが16ビットの場合にはメモリ2バイトとの間で演算が行われます。
⑤のmemの表記では
AND [BX]B, data8
AND [BX]W, data16
のようにB（Byte）またはW（Word）をつける必要があります。
[例]
①AND DL, AH
AND BX, SI
②AND BH, [BX]　[BX]Bにする必要は無い
AND CX, [DI]　[DI]Wにする必要は無い
③AND [BP], CL　[BP]Bにする必要は無い
AND [SI], AX　[SI]Wにする必要は無い
④AND BL, 45
AND DI, 3000
⑤AND [BP＋DI]B, 12　[ ]Bが必要
AND [DI]W, 78AB　[ ]Wが必要
[フラグ]
OF＝CF＝0になります。
計算の結果の値によってSF、ZF、PFが変化します。
AFは不明です。
DF、IF、TFは変化しません。

CALL
[書式]
①CALL label
labelで示されるサブルーチンをCALLします
[例]
CALL SUB01
　　|
　　|
SUB01：MOV AX, 0
[フラグ]
変化しません。

CALLF　　Call Far

［書式］

①CALLF label1:label2

②CALLF seg16:adr16

セグメント外にあるサブルーチンをCALLします。label1はセグメントアドレス、label2はサブルーチンのアドレスを示すラベル名です。

②のように16進数で直接示すこともできます。

通常のCALLはラベルのみで16進数での直接指定はしないので、A～Fで始るラベル名を指定しても＊をつける必要はありませんが、CALLFは16進数での指定もできるため、A～Fで始るラベル名を指定するときは＊をつける必要があります。

［例］

FSEG＝3000

FADRS＝1234

CALLF ＊FSEG:＊FADRS　　（CALLF 3000:1234と書くこともできます）

FADRS:MOV AX, 0　　（3000:1234）

［フラグ］

変化しません。

CBW Convert Byte to Word

ALにはいっている8ビットのデータを符号付きとみなして16ビットに拡張してAXレジスタに入れます。

［例］

MOV AL, 01

CBW　　AX＝0001になる

MOV AL, 80　　（80H＝－128D）

CBW　　AX＝FF80になる（FF80Hは16ビットの－128）

［フラグ］

変化しません。

CLC Clear Carry flag

CFフラグ＝0にします。

［フラグ］

CF＝0になる以外は変化しません。

CLD Clear Direction flag

DFフラグ＝0にします。

DFフラグが0のとき、ストリング命令（MOVS、CMPSなど）でインデックスレジスタの値が＋1または＋2されていきます。

［フラグ］

DF＝0になる以外は変化しません。

CLI Clear Interrupt flag

IFフラグ＝0にします。

IFフラグが0のときは、マスク可能割り込みはうけつけられません。

［フラグ］

IF＝0になる以外は変化しません。

CMC Compiement Carry flag

CFフラグの値を反転します。CF＝0をCF＝1に、CF＝1をCF＝0にします。

［フラグ］

CF以外は変化しません。

CMP Compare

［書式］

①CMP reg1, reg2

②CMP reg, mem

③CMP mem, reg

④CMP reg, data

⑤CMP mem, data

第1オペランドと第2オペランドの値を比較します。
比較は第1オペランドから第2オペランドを減算して行いますが第1オペランド、第2オペランドの値は変化しません。
①はreg1＝reg2＝8ビットまたはreg1＝reg2＝16ビットです。8ビットと16ビットの混合はできません。
⑤のmemの表記では
CMP ［BX]B, data8
CMP ［BX]W, data16
のようにB（Byte）またはW（Word）をつける必要があります。
［例］
①CMP DL, AH
CMP BX, SI
②CMP BH, ［BX］ ［BX]Bにする必要は無い
CMP CX, ［DI］ ［DI]Wにする必要は無い
③CMP ［BP], CL ［BP]Bにする必要は無い
CMP ［SI], AX ［SI]Wにする必要は無い
④CMP BL, 45
CMP DI, 3000
⑤CMP ［BP＋DI]B, 12 [ ]Bが必要
CMP ［DI]W, 78AB [ ]Wが必要
［フラグ］
SUB命令と同じ変化をします。
計算の結果の値によってOF、SF、ZF、AF、PF、CFが変化します。
DF、IF、TFは変化しません。
OP1＝OP2のときZF＝1になります。
符号なしの数として比較した結果、OP1＜OP2のときCF＝1になります。OP1＞PO2のときCF＝0になります。
符号付きとして比較した結果、OP1＜OP2のときSF＝1になります。OP1＞OP2のときはSF＝0になります。

CMPSB／CMPSW Compare String Byte／Compare String Word
［書式］
CMPSB
CMPSW
DS：SIで示されるアドレスのメモリデータとES：DIで示されるアドレスのメモリデータを比較し、結果によってフラグをセット、リセットしたあとSIレジスタとDIレジスタを更新します。DFフラグ＝0のとき、SIとDIは増加し、DFフラグ＝1のときSIとDIは減少します。
特別な場合を除いて、比較を行うメモリブロックは両方とも同じセグメントにあると思われますから、COMプログラム開始時の初期状態のCS＝DS＝ESのままでセグメントレジスタを意識する必要はないでしょう。
CMPSBは1バイトずつ比較を行います。実行後SIとDIは＋1または－1されます。
CMPSWはワード単位（2バイト）で比較を行います。実行後SIとDIは＋2または－2されます。
メモリのデータはどちらも変化しません。
REPプリフィックスと組み合わせることで任意のバイト数のメモリブロックの比較を行うプログラムが簡単に記述できます。
［例］
8000番地からのメモリ内容とA000番地からの内容を1000H（4096）バイト比較します。不一致を発見したらERR1にジャンプします。
MOV SI, 8000
MOV DI, A000
MOV CX, 1000
CLD DF＝0にする。次にSTDが実行されるまでDF＝0のままになる。
LP1：CMPSB
JNZ ERR1
DEC CX
JNZ LP1
下は上のプログラムをREPプリフィックスを使って書いたものです
MOV SI, 8000
MOV DI, A000
MOV CX, 1000
CLD
REPZ CMPSBをCX＝0になるまで、またはZF＝0になるまで繰り返す。
CMPSB
JNZ ERR
［フラグ］
SUB命令と同じ変化をします。

計算の結果の値によってOF、SF、ZF、AF、PF、CFが変化します。
DF、IF、TFは変化しません。

CWD　Convert Word to Doubleword

AXにはいっている16ビットのデータを符号付きとみなして32ビットに拡張してDXレジスタに上位16ビットを、AXレジスタに下位16ビットを入れます。

［例］

```
MOV AX, 1
CWD              DX＝0000、AX＝0001になる
MOV AX, FF80     （FF80Hは16ビットの－128）
CWD              DX＝FFFF、AX＝FF80になる
```

［フラグ］

変化しません。

DAA　Decimal Adjust to Add

ADD命令でパックBCD数の加算を行ったあとの10進補正を行います。パックBCD数は8ビットを4ビット2桁に分けて、各桁に0～9を当てはめたものです。

たとえばALレジスタに45が入っているとき、8086はこれを45H（＝69D）として計算を行います。

```
MOV AL, 45　（45H＝69D）
ADD AL, 27　（27H＝39D）
```

この計算の結果はAL＝6C（108D）になります。これで計算はあっているのですが、そうではなくて10進数の45＋27＝72の計算を行いたい場合にDAAを使います。

上での計算の次に DAA を実行すると、AL＝48（72D）に補正されます。

```
MOV AL, 45
ADD AL, 58
DAA
```

この例ではADD命令によってAL＝9Dになったあと、DAA命令によって、AL＝03に補正されCF＝1になります。CFフラグが上位桁へのキャリーになっています。

上位桁へのキャリーが発生しないときはCF＝0になります。

DAAはALレジスタに対してのみ働きます。

［フラグ］

結果の値によってSF、ZF、AF、PF、CFが変化します。

10進数に直した結果上位桁へのキャリーがあるときCF＝1になります。それ以外のときCF＝0になります。

OFは不明です。

DF、IF、TFは変化しません。

DAS　Decimal Adjust to Subtract

SUB命令でパックBCD数の減算を行ったあとの10進補正をします。

DASはALレジスタに対してのみ働きます。

［例］

```
①MOV AL, 75
 SUB AL, 26      AL＝4F
 DAS             AL＝49, CF＝0
②MOV AL, 12
  SUB AL, 43     AL＝CF
  DAS            AL＝69, CF＝1
```

②のように10進補正の結果上位桁からのボロー（借り）が発生したときはCF＝1になります。

［フラグ］

結果の値によってSF、ZF、AF、PF、CFが変化します。

10進数に直した結果上位桁からのボローがあるときCF＝1になります。それ以外のときCF＝0になります。

OFは不明です。

DF、IF、TFは変化しません。

DEC　Decrement

［書式］

①DEC reg

②DEC mem

　レジスタまたはメモリの内容を－1します。

［例］

①DEC AL
②DEC BX
③DEC [SI]B
④DEC [XYZ]W
③④のようにメモリを指定するときはB(byte)、W(word)の指定が必要です。
[フラグ]
結果の値によってOF、SF、ZF、AF、PFが変化します。
CFは変化しません。注意してください。
DF、IF、TFは変化しません。

DIV Divide
[書式]
①DIV reg8
②DIV mem8
③DIV reg16
④DIV mem16
①②ではAXレジスタの内容をオペランドで示した8ビットレジスタまたはメモリ1バイトの値で除算します。商はALに、余りはAHに入れられます。
③④ではDXレジスタが上位16ビット、AXレジスタが下位16ビットとして示される32ビットの数をオペランドで示した16ビットレジスタまたはメモリ2バイトの値で除算します。商はAXに、余りはDXに入れられます。
オペランドの値が0であった場合、または被除数≫除数でオーバーフローが発生した場合にはINT0割り込みが発生します。
[例]
①DIV CL
②DIV [SI]B
③DIV BX
④DIV [XYZ]W
②④のようにメモリを指定するときはB(byte)、W(word)の指定が必要です。
[フラグ]
OF、SF、ZF、AF、PF、CFは不明です。
DF、IF、TFは変化しません。

HLT Halt
プロセッサを停止させます。パソコンのアセンブラプログラムでは通常は使用しません。この命令によってCPUは停止してしまいます。
HLTを解除するにはリセットか外部割り込み信号の入力が必要です。マスク可能割り込みは事前にSTI(割り込み許可)が実行されていなければ無視されます。NMI(ノンマスカブルインタラプト)は受け付けられます。
[フラグ]
変化しません。

IDIV Integer Divide
[書式]
①IDIV reg8
②IDIV mem8
③IDIV reg16
④IDIV mem16
符号付き除算を行います(DIVは符号なし16進数の除算を行います)。
①②ではAXレジスタの内容をオペランドで示した8ビットレジスタまたはメモリ1バイトの値で除算します。商はALに、余りはAHに入れられます。
③④ではDXレジスタが上位16ビット、AXレジスタが下位16ビットとして示される32ビットの数をオペランドで示した16ビットレジスタまたはメモリ2バイトの値で除算します。商はAXに、余りはDXに入れられます。
オペランドの値が0であった場合、または被除数≫除数でオーバーフローが発生した場合にはINT0割り込みが発生します。
[例]
①IDIV CL
②IDIV [SI]B
③IDIV BX
④IDIV [XYZ]W
②④のようにメモリを指定するときはB(byte)、W(word)の指定が必要です。
[フラグ]
OF、SF、ZF、AF、PF、CFは不明です。
DF、IF、TFは変化しません。

IMUL Integer Muitiply

［書式］

①IMUL reg8

②IMUL mem8

③IMUL reg16

④IMUL mem16

符号付き乗算を行います（MULは符号なし16進数の乗算を行います）。

①②ではALレジスタの内容とオペランドで示した8ビットレジスタまたはメモリ1バイトの値との乗算を行います。結果はAXレジスタに入れられます。

③④ではAXレジスタとオペランドで示した16ビットレジスタまたはメモリ2バイトの値との乗算を行います。結果の上位16ビットはDXに、下位16ビットがAXに入れられます。

［例］

①IMUL CL

②IMUL ［SI]B

③IMUL BX

④IMUL ［XYZ]W

②④のようにメモリを指定するときはB（byte）、W（word）の指定が必要です。

［フラグ］

書式①②でAH＝0になったとき、または書式③④でDX＝0になったときはOF＝CF＝0になります。それ以外のときはOF＝CF＝1になります。

SF、ZF、AF、PFは不明です。

DF、IF、TFは変化しません。

IN

［書式］

①IN AL, adr8

②IN AX, adr8

③IN AL, DX

④IN AX, DX

①②は直接8ビットで示すI／Oアドレス（00～FF）から8ビットまたは16ビットのデータをALまたはAXに入力します。

8086は0000～FFFFのI／Oアドレスをアクセスできますが、書式①②はそのうちの0000～00FFの範囲しかアクセスできません。

上位8ビットのアドレスは常に00になります。

③④ではDXで示される16ビットのI／Oアドレス（0000～FFFF）から8ビットまたは16ビットのデータをALまたはAXに入力します。

［例］

①IN AL, 80

②IN AX, 05

③MOV DX, 3067

④IN AL, DX

［フラグ］

変化しません。

INC　Increment

［書式］

①INC reg

②INC mem

　レジスタまたはメモリの内容を＋1します。

［例］

①INC AL

②INC BX

③INC ［SI]B

④INC ［XYZ]W

③④のようにメモリを指定するときはB（byte）、W（word）の指定が必要です。

［フラグ］

結果の値によってOF、SF、ZF、AF、PFが変化します。

CFは変化しません。注意してください。

DF、IF、TFは変化しません。

INT　Interrupt

［書式］
INT data8
8ビットの数値（00～FF）で指定する番号のソフトウェア割り込みを発生させます。ROM BIOSのルーチンやMSDOSのINT 21割り込みルーチンを利用するのに使用します。
［例］
MOV AH, 02
MOV AL, 41
INT 21
［フラグ］
INT命令そのものはフラグに影響を与えませんが、INTによって呼び出され実行されるルーチンがメインプログラムに戻るときに、結果の情報をメインプログラムに伝えるために、意識的にフラグを変化させることがよくあります。特にCFフラグがよく使われます。

IRET Interrupt Return
割り込みプログラムからメインプログラムにリターンします。
割り込みプログラムの終わりにはRET命令は使用できません。
INT命令または外部割り込みによって割り込みプログラムに制御が移る時点で、戻り先のアドレスとフラグレジスタがスタックに待避されます。
通常のRET命令はスタックから戻り先アドレスしか取り出さないため、割り込みルーチンの最後にRET命令を使うとスタックが食い違ってしまいます。
［フラグ］
IRET命令が実行された時点でスタックに待避されていた割り込み直前のフラグレジスタの値を元に戻すため、IRET実行前のフラグの状態はこわされてしまいます。
（このことは、割り込みプログラムの先頭でメインプログラムのフラグレジスタを待避する必要がないことを示しています）

JMP Jump
［書式］
JMP label
labelで示されるプログラム位置に無条件でジャンプします。
［例］
JMP XYZ
　　|
　　|
XYZ:MOV AX, 0
［フラグ］
変化しません。

JMPF Jump Far
［書式］
①JMPF label1:label2
②JMPF seg16:adr16
セグメント外にあるプログラムにジャンプします。label1はセグメントアドレス、label2はジャンプ先のアドレスを示すラベル名です。
②のように16進数で直接示すこともできます。
通常のJMPはラベルのみで16進数での直接指定はしないので、A～Fで始るラベル名を指定しても＊をつける必要はありませんが、JMPFは16進数での指定もできるため、A～Fで始るラベル名を指定するときは＊をつける必要があります。
［例］
FSEG＝3000
FADRS＝1234
JMPF ＊FSEG:＊FADRS　　（JMPF 3000:1234と書くこともできます）
　　|
　　|
FADRS:MOV AX, 0　　（3000:1234）
［フラグ］
変化しません。

条件ジャンプ命令
以下に説明するJ××タイプの条件ジャンプ命令は通常CMP命令やADD、SUBなどの演算命令、ローティトなどのCFを変化させる命令の後ろにおいて、処理の流れを変えるのに使います。
条件が満足されると指定したラベルで示されるアドレスにジャンプします。
条件が満足されないときは条件ジャンプ命令の次の命令が実行されます。

条件ジャンプ命令の飛び先はそのジャンプ命令の次のアドレスから数えて＋127から－128バイトの範囲に限られます。それよりも遠いアドレスにジャンプしたいときは一旦近くにジャンプしておいて、そこに目的のアドレスへの無条件ジャンプ命令（JMP）を置きます。
8086の条件ジャンプ命令は異常な程たくさんあります。
じつは8086の条件ジャンプ命令は同じマシン語コードに複数のニーモニックがつけられているのです。
たとえばJCはJBやJNAEと同じマシン語コードに翻訳されます。
この説明書では同じマシン語コードになる命令は1箇所にまとめて説明してあります。同じところに複数の命令が併記してある場合はそのうちのどのニーモニックを書いても同じマシン語コードに翻訳されることを示しています。
そのうちで覚えやすいと思うものだけを使うようにすれば良いでしょう。

JA　Jump if Above
JNBE　Jump if not Below and not Equal
［書式］
JA label
JNBE label
CF＝ZF＝0（not carry and not zero）のときにlabelで示すアドレスにジャンプします。
符号なしの2数をCMP命令で比較するとき、またはSUB命令で減算するとき、第1オペランド＞第2オペランドだとこの条件になります。
［例］
CMP AL, 40
JA ALPHA　　AL＞40 のときALPHAへジャンプします
［フラグ］
変化しません。

JAE　Jump if Above or Eual
JNB　Jump if not Below
JNC　Jump if not Carry
［書式］
JAE label
JNB label
JNC label
CF＝0のときにlabelで示すアドレスにジャンプします。
符号なしの2数をCMP命令で比較するとき、またはSUB命令で減算するとき、第1オペランド≧第2オペランドだとこの条件になります。
［例］
CMP BL, 30
JAE SUUJI　　BL≧30 のときSUUJIへジャンプします
［フラグ］
変化しません。

JB　Jump if Below
JC　Jump if Carry
JNAE　Jump if not Above and not Equal
［書式］
JB label
JC label
JNAE label
CF＝1のときにlabelで示すアドレスにジャンプします。
符号なしの2数をCMP命令で比較するとき、またはSUB命令で減算するとき、第1オペランド＜第2オペランドだとこの条件になります。
［例］
CMP CL, 5B
JB ALPHA　　CL＜5B のときALPHAへジャンプします
［フラグ］
変化しません。

JBE　Jump if Below or Eual
JNA　Jump if not Abobe
［書式］

JBE label
JNA label
CF＝1またはZF＝1のときにlabelで示すアドレスにジャンプします。
符号なしの2数をCMP命令で比較するとき、またはSUB命令で減算するとき、第1オペランド≦第2オペランドだとこの条件になります。
［例］
CMP DL, 39
JBE SUUJI　　DL≦30 のときSUUJIへジャンプします
［フラグ］
変化しません。

JCXZ Jump if CX is Zero
［書式］
JCXZ label
CXレジスタの値が0のときにlabelで示すアドレスにジャンプします。
他の条件ジャンプ命令がフラグの状態によってジャンプするのと違って、フラグに関係なくCXレジスタが0のときにジャンプします。
［例］
JCXZ JP1　　CX＝0 のときJP1へジャンプします
INC DX
JP1：MOV AX, DX
［フラグ］
変化しません。

JE Jump if Eual
JZ Jump if Zero
［書式］
JE label
JZ label
ZF＝1のときにlabelで示すアドレスにジャンプします。
2数をCMP命令で比較するとき、またはSUB命令で減算するとき、2数が同じだとこの条件になります。
［例］
CMP AH, 0D
JE DTEND　　AH＝0D のときDTENDへジャンプします
［フラグ］
変化しません。

JG Jump if Greater
JNLE Jump if not Less and not Equal
［書式］
JG label
JNLE label
ZF＝0でOF＝SFのときにlabelで示すアドレスにジャンプします。
符号付きの2数をCMP命令で比較するとき、またはSUB命令で減算するとき、第1オペランド＞第2オペランドだとこの条件になります。
符号付きの2数の減算では、OF＝1、SF＝1になる場合が出てきます（下例）。
［例］
SUB BH, FF　　BH＝7F（＋127）のとき、実行した結果はBH＝80（－128、OF＝1、SF＝1）
JG PLUS　　減算前のBH＞－1（FF） のときPLUSへジャンプします
［フラグ］
変化しません。

JGE Jump if Greater or Equal
JNL Jump if not Less
［書式］
JGE label
JNL label
ZF＝1またはOF＝SFのときにlabelで示すアドレスにジャンプします。
符号付きの2数をCMP命令で比較するとき、またはSUB命令で減算するとき、第1オペランド≦第2オペランドだとこの条件になります。

[例]
CMP CH, F0
JGE DATA　　BH≧F0（－16）のときDATAへジャンプします
[フラグ]
変化しません。

JL Jump if Less
JNGE Jump not Greater and not Equal
[書式]
JL label
JNGE label
SF≠OFのときにlabelで示すアドレスにジャンプします。
符号付きの2数をCMP命令で比較するとき、またはSUB命令で減算するとき、第1オペランド＜第2オペランドだとこの条件になります。
符号付きの2数の減算では、OF＝1、SF＝0になる場合が出てきます（下例）。
[例]
SUB BH, 1　　BH＝80（－128）のとき、実行した結果はBH＝7F（＋127、OF＝1、SF＝0）
JL DATA　　減算前のBH＜1のときDATAへジャンプします
[フラグ]
変化しません。

JLE Jump if Less or Equal
JNG Jump if not Greater
[書式]
JLE label
JNG label
ZF＝1またはSF≠OFのときにlabelで示すアドレスにジャンプします。
符号付きの2数をCMP命令で比較するとき、またはSUB命令で減算するとき、第1オペランド≦第2オペランドだとこの条件になります。
[例]
CMP BH, FF
JLE DATA　　BH≦1のときDATAへジャンプします
[フラグ]
変化しません。

JNE Jump if not Eual
JNZ Jump if not Zero
[書式]
JNE label
JNZ label
ZF＝0のときにlabelで示すアドレスにジャンプします。
2数をCMP命令で比較するとき、またはSUB命令で減算するとき、2数が同じでない場合にこの条件になります。
[例]
CMP AH, FF
JNE DETA　　AH≠FF のときDETAへジャンプします
[フラグ]
変化しません。

JNO Jump if not Overflow
[書式]
JNO label
OFフラグ＝0のときにlabelで示すアドレスにジャンプします。
この命令より前に実行された演算命令でオーバーフローが起きていなければ、OF＝0になります。
符号付数の加減算などで結果が扱える数の範囲を超えるとオーバーフローが発生します（JO命令の[例]参照）。
[フラグ]
変化しません。

JNP Jnmp if not Parity
JPO Jump if Parity Odd

［書式］
JNP　label
JPO　label
PF（パリティフラグ）＝0のときにlabelで示すアドレスにジャンプします。
この命令より前に行われた演算の結果の値の下位8ビットのうち１のビットが奇数であるときPF＝0になります。
［例］
OR　BL, BL
JNP　KISU
［フラグ］
変化しません。

JNS　Jump　if　not　Sign
［書式］
JNS　label
SF（サインフラグ）＝0のときlabelで示すアドレスにジャンプします。
この命令より前に行われた演算の結果を符号付の数としてみたとき、正の数か0ならばSF＝0になります。
［例］
ADD　AX, BX
JNS　PLUS
［フラグ］
変化しません。

JO　　Jump　if　Overflow
［書式］
JO　label
OFフラグ＝１のときにlabelで示すアドレスにジャンプします。
この命令より前に実行された演算命令でオーバーフローが起きるとOF＝１になります。
符号付数の加減算などで結果が扱える数の範囲を超えるとオーバーフローが発生します。
［例］
ADD　AL, 1　　AL＝7F（＋127）のとき、加算の結果AL＝80（－128）になってOF＝１になります
JO　OVER
［フラグ］
変化しません。

JP　　Jump　if　Parity
JPE　Jump　if　Parity　Even
［書式］
JP　label
JPE　label
PF（パリティフラグ）＝１のときにlabelで示すアドレスにジャンプします。
この命令より前に行われた演算の結果の値の下位8ビットのうち１のビットが偶数であるときPF＝１になります。
［例］
OR　BL, BL
JP　GUSU
［フラグ］
変化しません。

JS　　Jump　if　Sign
［書式］
JS　label
SF（サインフラグ）＝１のときlabelで示すアドレスにジャンプします。
この命令より前に行われた演算の結果を符号付の数としてみたとき、負の数ならばSF＝１になります。
［例］
ADD　AX, BX
JS　MINUS
［フラグ］
変化しません。

LAHF　Load　AH　from　Flagregister

［書式］
LAHF
SF、ZF、AF、PF、CFフラグの状態ををAHレジスタにコピーします。
OF、DF、IF、TFはコピーされません。

7	6	5	4	3	2	1	0	←ビットNo.
SF	ZF	×	AF	×	PF	×	CF	

×マークのビットは不定

［フラグ］
変化しません。

LDS　Load pointer using DS
［書式］
LDS reg16, label
labelで示されるメモリアドレスから2バイトのデータを16ビットレジスタに格納し、その次の2バイトのデータをDSに格納します。
［例］
ADRS1＝1000
　　｜
　　｜
LDS BX, ADRS1　1000～1001番地の内容がBXに、1002～1003番地の内容がDSに入れられます
［フラグ］
変化しません。

LEA　Load Effective Address
［書式］
LEA reg16, ladel
labelで示されるアドレスを16ビットアドレスに格納します。
昔のアセンブラではこの命令が必要だったのでしょうか。
このアセンブラでは MOV reg16, label で同じ働きになるため、この命令を使う必要はありません。
［フラグ］
変化しません。

LES　Load pointer using ES
［書式］
LES reg16, label
labelで示されるメモリアドレスから2バイトのデータを16ビットレジスタに格納し、その次の2バイトのデータをESに格納します。
［例］
ADRS1＝1000
　　｜
　　｜
LES BX, ADRS1　1000～1001番地の内容がBXに、1002～1003番地の内容がESに入れられます
［フラグ］
変化しません。

LOCK　Lock bus
［書式］
LOCK
この命令の次の命令の実行が完了するまでバスを他のプロセッサからアクセスできないようにロックします。
普通のシステムでは使用しません。
［フラグ］
変化しません。

LODSB／LODSW　Load String Byte／Load String Word
［書式］
LODSB
LODSW
LODSBはDS：SIで示されるメモリアドレスから1バイトのデータをALに格納し、その後にDF＝0のときはSIを＋1します。DF＝1のときはSIを－1します。
LODSWはDS：SIで示されるメモリアドレスとその次のアドレスから2バイトのデータをAXに格納し、その後にDF＝0のときはSIを＋

2します。DF＝1のときはSIを－2します。
この命令はたとえば
MOV AL,［SI］
INC SI
という命令で置き換えができますから、あえて使わなければならない命令とはいえません。
DFの設定が必要なことを考慮するとむしろMOV命令を使う方が賢明と言えるでしょう。
［例］
CLD　　次にSTD命令が実行されるまではDF＝0になります
LODSB　DS：SIの内容がALに入ります
［フラグ］
変化しません。

LOOP
［書式］
LOOP label
CXレジスタの値を－1し、CX＝0になるまで、labelで示されるアドレスにジャンプします。
labelはこのLOOP命令の次の命令のアドレスから＋127～－128バイトの間になければいけません。
［例］
MOV AL, FF
MOV DI, 1000
MOV CX, 100
LP1：MOV［DI］, AX
INC DI
LOOP LP1　　LP1からLOOPまでの命令がCX＝0になるまで100H＝256回繰り返し実行されます
［フラグ］
変化しません。

LOOPE　Loop if Equal
LOOPZ　Loop if Zero
［書式］
LOOPE label
LOOPZ label
ZF＝1ならばCXレジスタの値を－1し、CX＝0になるまで、labelで示されるアドレスにジャンプします。
labelはこのLOOP命令の次の命令のアドレスから＋127～－128バイトの間になければいけません。
［例］
MOV CX, 100
LP1：INC SI
CMP［SI］B, 00
LOOPE LP1　　LP1からLOOPまでの命令がCX＝0になるまで100H＝256回繰り返し実行されます。
　　ただし［SI］≠00であるとLOOPEの次の命令に制御が移ります
［フラグ］
変化しません。

LOOPNE　Loop if not Equal
LOOPNZ　Loop if not Zero
［書式］
LOOPNE label
LOOPNZ label
ZF＝0ならばCXレジスタの値を－1し、CX＝0になるまで、labelで示されるアドレスにジャンプします。
labelはこのLOOP命令の次の命令のアドレスから＋127～－128バイトの間になければいけません。
［例］
MOV CX, 100
LP1：INC SI
CMP［SI］B, 00
LOOPE LP1　　LP1からLOOPまでの命令がCX＝0になるまで100H＝256回繰り返し実行されます。
　　ただし［SI］＝00であるとLOOPEの次の命令に制御が移ります
［フラグ］
変化しません。

MOV　　Move

［書式］

①MOV reg1, reg2
②MOV reg, mem
③MOV mem, reg
④MOV reg, data
⑤MOV mem, data
⑥MOV segr, reg16
⑦MOV segr, mem
⑧MOV reg16, sreg
⑨MOV mem, sreg

第1オペランドで指定するレジスタ、メモリに第2オペランドで指定する値が代入されます。

①はレジスタ間のデータ転送を行います。8ビットレジスタ←8ビットレジスタか16ビットレジスタ←16ビットレジスタに限ります。8ビットと16ビットとの間での転送はできません。

②③はレジスタとメモリとの間のデータ転送を行います。16ビットレジスタの場合には指定するメモリとその次のアドレスのメモリとの間で2バイトのデータが転送されます。

④はレジスタに直接データの値を入れます。8ビットレジスタには1バイトの16進数の値を直接指定します。10進数や変数の指定はできません。16ビットレジスタには2バイトの16進数の値を直接指定するほか、変数やラベルを指定することもできます。ラベルを指定するとそのラベル位置のプログラムアドレスが16レジスタに入れられます。

⑤はメモリに直接データの値を入れます。[ ]B（8ビット）か[ ]W（16ビット）の区別を表記しなければエラーになります。[ ]B（8ビット）には1バイトの16進数の値を直接指定します。10進数や変数の指定はできません。[ ]W（16ビット）には2バイトの16進数の値を直接指定するほか、変数やラベルを指定することもできます。ラベルを指定するとそのラベル位置のプログラムアドレスがメモリに入れられます。

⑥⑦はセグメントレジスタに16ビットレジスタかメモリの値を代入します。セグメントレジスタには直接値を代入することができない（④⑤に相当する命令はない）のでこの命令を使います。

⑧⑨はセグメントレジスタの値を16ビットレジスタかメモリに代入します。

［例］

```
MOV AL, BL
MOV AX, BX
MOV CL, [SI]
MOV CX, [DI]
MOV [BX], AL
MOV [DI], DX
MOV AL, 38
MOV BX, 345
MOV DI, LBL1
MOV [BX]B, 45
MOV [SI]W, HENSU
MOV DS, AX
MOV ES, [MEMO1]
MOV BX, CS
MOV [WORK], ES
```

［フラグ］

変化しません。

MOVSB／MOVSW　　Move String Byte／Move String Word

［書式］

MOVSB
MOVSW

DS:SIで示されるアドレスのメモリデータからES:DIで示されるアドレスにデータを転送します。1バイトまたは2バイトのデータを転送したあと、SIレジスタとDIレジスタを更新します。DFフラグ＝0のとき、SIとDIは増加し、DFフラグ＝1のときSIとDIは減少します。

MOVSBは1バイトずつ転送します。実行後SIとDIは＋1または－1されます。

MOVSWはワード単位（2バイト）で転送します。実行後SIとDIは＋2または－2されます。

この命令より前にDSとESに異なるセグメントアドレスを入れておくことで64KB単位のデータ転送が簡単に記述できます。ただ不用意にセグメントレジスタ（特にDS）を変更するとトラブルのもとになりますから、十分理解できるまではセグメントレジスタは変更しない方がよいでしょう。

REPプリフィックスと組み合わせることで任意のバイト数のメモリ間ブロック転送を行うプログラムが簡単に記述できます。

［例］

8000～8FFFのメモリ内容1000H（4096）バイトをA000～AFFFに転送します。

```
MOV  SI, 8000
MOV  DI, A000
MOV  CX, 1000
CLD                    DF＝0にする。次にSTDが実行されるまでDF＝0のままになる。
LP1:MOVSB
DEC  CX
JNZ  LP1
```
下は上のプログラムをREPプリフィックスを使って書いたものです
```
MOV  SI, 8000
MOV  DI, A000
MOV  CX, 1000
CLD
REPZ              MOVSBをCX＝0になるまで繰り返す。
MOVSB
JNZ  ERR
```
[フラグ]
変化しません。

MUL　　Multiply

[書式]
①MUL reg8
②MUL mem8
③MUL reg16
④MUL mem16
符号なしの乗算を行います(IMULは符号つき16進数の乗算を行います)。
①②ではALレジスタの内容とオペランドで示した8ビットレジスタまたはメモリ1バイトの値との乗算を行います。結果はAXレジスタに入れられます。
③④ではAXレジスタとオペランドで示した16ビットレジスタまたはメモリ2バイトの値との乗算を行います。結果の上位16ビットはDXに、下位16ビットがAXに入れられます。
[例]
①MUL CL
②MUL [SI]B
③MUL BX
④MUL [XYZ]W
②④のようにメモリを指定するときはB(byte)、W(word)の指定が必要です。
[フラグ]
書式①②でAH＝0になったとき、または書式③④でDX＝0になったときはOF＝CF＝0になります。それ以外のときはOF＝CF＝1になります。
SF、ZF、AF、PFは不明です。
DF、IF、TFは変化しません。

NEG　　Negative

[書式]
NEG reg
NEG mem
レジスタまたはメモリの値を符号付2進数とみなしてその正負を反転させます。NOTとの違いを理解してください。
オペランドにメモリを指定するときは[ ]B(Byte)、[ ]W(Word)を明示しなければエラーになります。
もとの値が0のときは結果も0になります(CF＝0になります)。
もとの値が負の最大値(80Hまたは8000H)のときは値は変わりません(OF＝1になります)
[例]
```
NEG  AL          AL＝1のとき、実行の結果はAL＝FF(－1)
NEG  [BX]W     [ ]B、[ ]Wの指定が必要
```
[フラグ]
結果の値によってSF、ZF、AF、PFが変化します。
CFは通常は1、もとの値が0のときCF＝0になります。
OFは通常は0、もとの値が80または8000のときOF＝1になります。

NOP　　No Operation

[書式]

NOP
何の操作も行いません。フラグも変化しません。
通常アセンブラでは使用する機会はありません。マシン語で直接命令コードを書いてプログラムを作成する場合にはよく使います。
不要になった命令をNOP（マシン語コードは90）で置き換えるのに利用します。
［フラグ］
変化しません。

NOT
［書式］
NOT　reg
NOT　mem
レジスタまたはメモリの値の各ビットを反転します。NEGとの違いを理解してください。
オペランドにメモリを指定するときは［　］B（Byte）、［　］W（Word）を明示しなければエラーになります。
［例］
NOT　BH　　BH＝AE（10101110）のとき、実行の結果はBH＝51（01010001）
NEG　［BX］W　　［　］B、［　］Wの指定が必要
［フラグ］
変化しません。

OR
［書式］
①OR　reg1，reg2
②OR　reg，mem
③OR　mem，reg
④OR　reg，data
⑤OR　mem，data
第1オペランドと第2オペランドのOR（論理和）を計算して第1オペランドのregまたはmemに格納します。第2オペランドの値は変化しません。
①はreg1＝reg2＝8ビットまたはreg1＝reg2＝16ビットです。8ビットと16ビットの混合はできません。
②③ではレジスタが8ビットのときはメモリ1バイトとの間で、レジスタが16ビットの場合にはメモリ2バイトとの間で演算が行われます。
⑤のmemの表記では
OR　［BX］B，data8
OR　［BX］W，data16
のようにB（Byte）またはW（Word）をつける必要があります。
［例］
①OR　DL，AH
　OR　BX，SI
②OR　BH，［BX］　　［BX］Bにする必要は無い
　OR　CX，［DI］　　［DI］Wにする必要は無い
③OR　［BP］，CL　　［BP］Bにする必要は無い
　OR　［SI］，AX　　［SI］Wにする必要は無い
④OR　BL，45
　OR　DI，3000
⑤OR　［BP＋DI］B，12　　［　］Bが必要
　OR　［DI］W，78AB　　［　］Wが必要
［フラグ］
OF＝CF＝0になります。
計算の結果の値によってSF、ZF、PFが変化します。
AFは不明です。
DF、IF、TFは変化しません。

OUT
［書式］
①OUT　adr8，AL
②OUT　adr8，AX
③OUT　DX，AL
④OUT　DX，AX
①②は直接8ビットで示すI／Oアドレス（00～FF）にALまたはAXの値を出力します。

8086は0000～FFFFのI／Oアドレスをアクセスできますが、書式①②はそのうちの0000～00FFの範囲しかアクセスできません。
上位8ビットのアドレスは常に00になります。
③④ではDXで示される16ビットのI／Oアドレス（0000～FFFF）にALまたはAXの値を出力します。
［例］
①OUT 80, AL
②OUT B3, AX
③OUT DX, AL
④OUT DX, AX
［フラグ］
変化しません。

POP
［書式］
①POP reg16
②POP mem16
③POP segr
PUSH命令によってスタックに退避されていた16ビットの値をレジスタ、メモリに戻します。
この命令の実行後SP（スタックポインタ）が＋2されます。
PUSHとPOPを必ず対で使う必要があります。PUSHを 2 回行って、POPは1回だけというようにしておくと、その後はスタックが食い違ったままになるためプログラムが暴走してしまったりします。
PUSHとPOPは順番が必要でレジスタ名には依存しません。最後にPUSHしたデータがその後に最初にPOPされたレジスタに入れられます。
［例］
①PUSH AX
PUSH BX
　　|
　　|
POP CX　　CXにBXの値が入ります
POP DX　　DXにAXの値が入ります
テクニックとして意識的にこのような使い方をすることがあります。
②POP ［BX］　　［BX］Wとする必要はありません
③POP ES
［フラグ］
変化しません。

POPF　Pop Flagregister
［書式］
POPF
PUSHFによってスタックに退避されていたフラグレジスタの値をもとに戻します。
この命令の実行後SP（スタックポインタ）が＋2されます。
［フラグ］
全フラグが変化します（PUSHFによって保存されていた内容に置き換えられます）。

PUSH
［書式］
①PUSH reg16
②PUSH mem16
③PUSH segr
レジスタ、メモリの値（16ビット）をスタックに格納します。レジスタ、メモリの値は変化しません。
この命令の実行後SP（スタックポインタ）が－2されます。
PUSHしたものは必ずPOPしなければいけません。通常はPUSHしたレジスタに対してPOPで元に戻す処理が多いのですが、必ずレジスタ名を合わせる必要はありません。テクニックでわざと違うレジスタにPOPすることもあります（POPの［例］参照）。
スタックには積み重ねるという意味があります。
SPで指定したアドレス（通常はメモリの一番後ろに指定する）から前に向かって積み上げる形でデータを格納していきます。
COMファイルの場合、MSDOSによってプログラムの実行開始時点ではSP＝FFFEになっていますからプログラムの先頭でSPを設定する必要はありません。
AX＝1234、BX＝5678、CX＝90ABのとき
PUSH AX
PUSH BX

PUSH CX
の順で実行すると、スタックの状態は下のようになります（実行前のSP＝FFFE）。

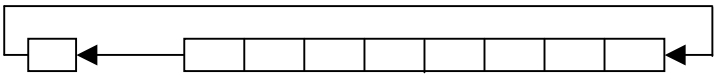
FFFF	
FFFE	
FFFD	34
FFFC	12
FFFB	78
FFFA	56
FFF9	AB
FFF8	90
FFF7	
FFF6	

SP＝FFF8になる

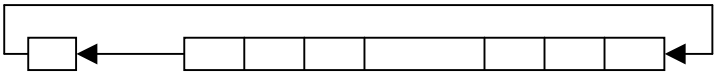
［フラグ］
変化しません。

PUSHF　Push Flagregister
［書式］
PUSHF
フラグレジスタの内容をスタックに退避します。退避したスタックの値はPOPFもとに戻すことができます。
この命令の実行後SP（スタックポインタ）が－2されます。
［フラグ］
変化しません。

RCL　　Rotate Left through Carry
［書式］
①RCL reg, 1／RCL reg
②RCL mem, 1／RCL mem
③RCL reg, CL
④RCL mem, CL
①②はCF（キャリーフラグ）を含めて9ビットまたは17ビットのデータを左に1ビット回転します。
8ビットの場合



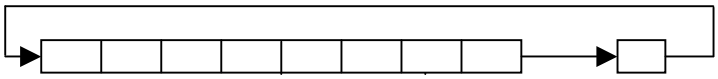
CF　7　6　5　4　3　2　1　0　ビット
8ビットデータを左に回転し、ビット7がCFに入り、CFの値がビット0に入ります。
16ビットの場合



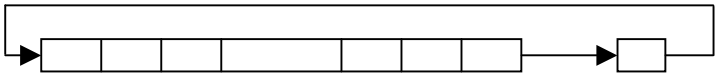
CF　15　14　13　2　1　0　ビット
16ビットデータを左に回転し、ビット15がCFに入り、CFの値がビット0に入ります。
③④ではCLの値が左回転のビット数（回数）になります。
②④では[ ]B（8ビット）、[ ]W（16ビット）の指定が必要です。
［例］
RCL AL
RCL ［BX]W
RCL BX, CL
RCL ［DI]B, CL
［フラグ］
CFには回転前のレジスタ、メモリの最上位ビットの値が入ります。
①②では回転の対象になる値が符号付の数と考えたとき、回転前と回転後で符号が変わった場合にOF＝1になります。符号が変化しなかったときはOF＝0になります。
回転前の値が80（－128），CF＝1のとき、回転後は01（＋1），CF＝1，OF＝1になります。

③④ではOFは不明です。
その他のフラグは変化しません。

RCR　　Rotate Righit through Carry
［書式］
①RCR reg, 1／RCR reg
②RCR mem, 1／RCR mem
③RCR reg, CL
④RCR mem, CL
①②はCF（キャリーフラグ）を含めて9ビットまたは17ビットのデータを右に1ビット回転します。
8ビットの場合



ビット7　6　5　4　3　2　1　0　　CF
8ビットデータを右に回転し、ビット0がCFに入り、CFの値がビット7に入ります。
16ビットの場合



ビット15　14　13　　2　1　0　　CF
16ビットデータを右に回転し、ビット0がCFに入り、CFの値がビット15に入ります。
③④ではCLの値が右回転のビット数（回数）になります。
②④では[ ]B（8ビット）、[ ]W（16ビット）の指定が必要です。
［例］
RCR AL
RCR ［BX]W
RCR BX, CL
RCR ［DI]B, CL
［フラグ］
CFには回転前のレジスタ、メモリの最下位ビットの値が入ります。
①②では回転の対象になる値が符号付の数と考えたとき、回転前と回転後で符号が変わった場合にOF＝1になります。符号が変化しなかったときはOF＝0になります。
回転前の値が80（－128）, CF＝0のとき、回転後は40（＋64）, CF＝0, OF＝1になります。
③④ではOFは不明です。
その他のフラグは変化しません。

REP　　Repeat string operation
REPE　　Repeat string operation while Equal
REPZ　　Repeat string operation while Zero
［書式］
REP
REPE
REPZ
ストリング命令の前につけて、繰り返しを指示するプリフィックス命令です。REPEは単独では使わずに、MOVS、CMPSなどのストリング命令の前につけて、繰り返し回数を指定します。
ZF＝1である間、CXレジスタの値で示す回数、次のストリング命令を繰り返し実行します（CMPS、MOVSの［例］参照）。
［フラグ］
変化しません。

REPNE　　Repeat string operation while not Equal
REPNZ　　Repeat string operation while not Zero
［書式］
REPNE
REPNZ
ストリング命令の前につけて、繰り返しを指示するプリフィックス命令です。REPNEは単独では使わずに、MOVS、COMPSなどのストリング命令の前につけて、繰り返し回数を指定します。
ZF＝0である間、CXレジスタの値で示す回数、次のストリング命令を繰り返し実行します。
REPZ（REPE）に比べると使用することは少ない命令です。

［フラグ］
変化しません。

RET　　Return

［書式］
RET
CALLで呼び出されたサブルーチンから元のメインプログラムにリターンします。
CALL命令でスタックに保存されていた16ビットの戻り先アドレスをIP（インストラクションポインタ）に戻すことでメインプログラムにもどります。この結果SP（スタックポインタ）は＋2されます。
CALLとPUSHはスタックに値を保存し、POPとRETはスタックから値を取り出します。組み合わせ、順序が食違うとプログラムが暴走してしまいますから十分注意が必要です。
［例］

```
CALL  SUB1
    |
SUB1:PUSH  AX
    |
POP  AX
RET
```

［フラグ］
変化しません。

RETF　　Return Far

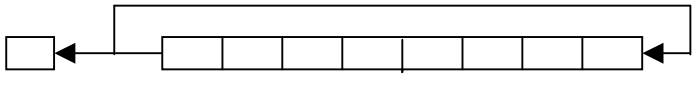
［書式］
RETF
セグメント外のプログラムからCALLF命令で呼び出されたサブルーチンから元のメインプログラムにリターンします。
CALLF命令でスタックに保存されていた32ビットの戻り先アドレスをCS（コードセグメント）とIP（インストラクションポインタ）に戻すことでメインプログラムにもどります。この結果SP（スタックポインタ）は＋4されます。
CALLとPUSHはスタックに値を保存し、POPとRETはスタックから値を取り出します。組み合わせ、順序が食違うとプログラムが暴走してしまいますから十分注意が必要です。
［例］

```
CALLF  SUBSEG:SUBADRS
  |
SUBADRS:PUSH  AX   （CS＝SUBSEG）
  |
POP  AX
RETF
```

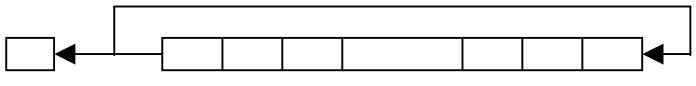
［フラグ］
変化しません。

ROL　　Rotate Left

［書式］
①ROL reg, 1／ROL reg
②ROL mem, 1／ROL mem
③ROL reg, CL
④ROL mem, CL
①②は8ビットまたは16ビットのデータを左に1ビット回転するとともに最上位ビットがCF（キャリーフラグ）に入ります。
8ビットの場合



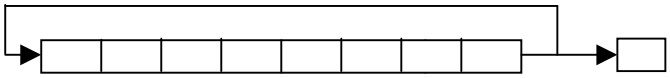
8ビットデータを左に回転し、ビット7はCFとビット0に入ります。
16ビットの場合



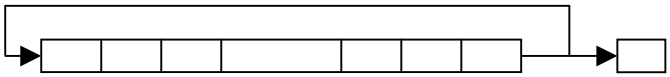
16ビットデータを左に回転し、ビット15はCFとビット0に入ります。
③④ではCLの値が左回転のビット数（回数）になります。

②④では[ ]B(8ビット)、[ ]W(16ビット)の指定が必要です。
[例]
ROL AL
ROL [BX]W
ROL BX, CL
ROL [DI]B, CL
[フラグ]
CFには回転前のレジスタ、メモリの最上位ビットの値が入ります。
①②では回転の対象になる値が符号付の数と考えたとき、回転前と回転後で符号が変わった場合にOF＝1になります。符号が変化しなかったときはOF＝0になります。
回転前の値が80(－128), CF＝0のとき、回転後は01(＋1), CF＝1, OF＝1になります。
③④ではOFは不明です。
その他のフラグは変化しません。

ROR　　Rotate Righit
[書式]
①ROR reg, 1／ROR reg
②ROR mem, 1／ROR mem
③ROR reg, CL
④ROR mem, CL
①②は8ビットまたは16ビットのデータを右に1ビット回転するとともにビット0がCF(キャリーフラグ)に入ります。
8ビットの場合



ビット7　6　5　4　3　2　1　0　　CF
8ビットデータを右に回転し、ビット0はCFとビット7に入ります。
16ビットの場合



ビット15　14　13　　2　1　0　　CF
16ビットデータを右に回転し、ビット0はCFとビット15に入ります。
③④ではCLの値が右回転のビット数(回数)になります。
②④では[ ]B(8ビット)、[ ]W(16ビット)の指定が必要です。
[例]
ROR AL
ROR [BX]W
ROR BX, CL
ROR [DI]B, CL
[フラグ]
CFには回転前のレジスタ、メモリの最下位ビットの値が入ります。
①②では回転の対象になる値が符号付の数と考えたとき、回転前と回転後で符号が変わった場合にOF＝1になります。符号が変化しなかったときはOF＝0になります。
回転前の値が80(－128), CF＝0のとき、回転後は40(＋64), CF＝0, OF＝1になります。
③④ではOFは不明です。
その他のフラグは変化しません。

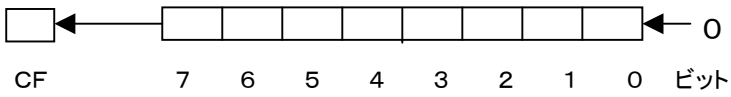
SAHF　　Save AH to Flagregister
[書式]
SAHF
AHの特定ビットの値をSF、ZF、AF、PF、CFフラグにコピーします。
OF、DF、IF、TFにはコピーされません。

7	6	5	4	3	2	1	0	←ビットNo.
SF	ZF	×	AF	×	PF	×	CF	

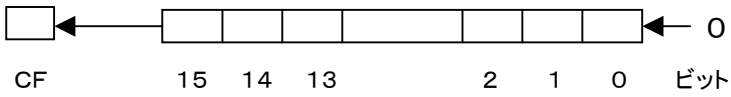
×マークのビットは使われません

[フラグ]
SF、ZF、AF、PF、CFフラグが変化します。
OF、DF、IF、TFは変化しません。

SAL　　Shift Arithmetic Left
SHL　　Shift Left
［書式］
①SAL reg, 1／SAL reg
　SHL reg, 1／SHL reg
②SAL mem, 1／SAL mem
　SHL mem, 1／SHL mem
③SAL reg, CL
　SHL reg, CL
④SAL mem, CL
　SHL mem, CL
①②は8ビットまたは16ビットのデータを左に1ビットシフトします。最上位ビットはCFに入ります。ビット0には0が入れられます。左シフトの場合にはSALとSHLは同じ結果になります。
8ビットの場合

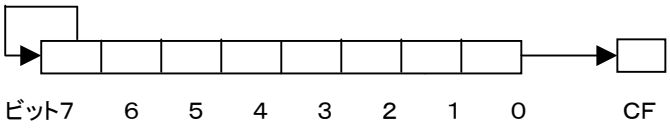


8ビットデータを左にシフトし、ビット7がCFに入り、0がビット0に入ります。
16ビットの場合



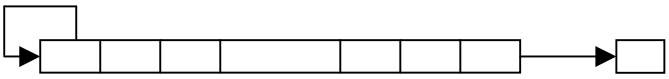
16ビットデータを左にシフトし、ビット15がCFに入り、0がビット0に入ります。
③④ではCLの値が左シフトのビット数（回数）になります。
②④では[ ]B（8ビット）、[ ]W（16ビット）の指定が必要です。
［例］
SAL AL
SAL ［BX]W
SAL BX, CL
SAL ［DI]B, CL
SHL AL
SHL ［BX]W
SHL BX, CL
SHL ［DI]B, CL
［フラグ］
CFにはシフト前のレジスタ、メモリの最上位ビットの値が入ります。
①②ではシフトの対象になる値が符号付の数と考えたとき、シフト前とシフト後で符号が変わった場合にOF＝1になります。符号が変化しなかったときはOF＝0になります。
シフト前の値が81（－127）のとき、シフト後は02（＋2），CF＝1，OF＝1になります。
③④ではOFは不明です。
結果の値によってSF、ZF、PFが変化します。
AFは不明です。
DF、IF、TFは変化しません。

SAR　　Shift Arithmetic Right
［書式］
①SAR reg, 1／SAR reg
②SAR mem, 1／SAR mem
③SAR reg, CL
④SAR mem, CL
①②は8ビットまたは16ビットのデータを右に1ビットシフトするとともにビット0がCF（キャリーフラグ）に入ります。最上位ビット（符号ビット）はそのまま残ります。この結果最上位ビットとそのすぐ右のビットは同じ値になります。
8ビットの場合



8ビットデータを右にシフトし、ビット0はCFに入ります。ビット7は変化しません。

16ビットの場合



ビット15　14　13　　2　1　0　　CF
16ビットデータを右にシフトし、ビット0はCFに入ります。ビット15は変化しません。
③④ではCLの値が右シフトのビット数（回数）になります。
②④では[ ]B（8ビット）、[ ]W（16ビット）の指定が必要です。
[例]
SAR AL
SAR [BX]W
SAR BX, CL
SAR [DI]B, CL
[フラグ]
CFにはシフト前のレジスタ、メモリのビット0の値が入ります。
①②ではOFフラグが影響をうけると、8086の解説書には記載してあります。しかしSARではシフト前とシフト後で符号の変化は無いのでOFは起きないのではないかと思われます。ここではOFは不明としておきます。
③④ではOFは不明です。
結果の値によってSF、ZF、PFが変化します。
AFは不明です。
DF、IF、TFは変化しません。

SBB　Subtract with Borrow
[書式]
①SBB reg1, reg2
②SBB reg, mem
③SBB mem, reg
④SBB reg, data
⑤SBB mem, data
第1オペランドから第2オペランドとCFフラグを減算して第1オペランドのregまたはmemに格納します。第2オペランドの値は変化しません。
①はreg1＝reg2＝8ビットまたはreg1＝reg2＝16ビットです。8ビットと16ビットの混合はできません。
⑤のmemの表記では
SBB [BX]B, data8
SBB [BX]W, data16
のようにB（Byte）またはW（Word）をつける必要があります。
[例]
①SBB DL, AH
　SBB BX, SI
②SBB BH, [BX]　[BX]Bにする必要は無い
　SBB CX, [DI]　[DI]Wにする必要は無い
③SBB [BP], CL　[BP]Bにする必要は無い
　SBB [SI], AX　[SI]Wにする必要は無い
④SBB BL, 45
　SBB DI, 3000
⑤SBB [BP＋DI]B, 12　[ ]Bが必要
　SBB [DI]W, 78AB　[ ]Wが必要
[フラグ]
計算の結果の値によってOF、SF、ZF、AF、PF、CFが変化します。
DF、IF、TFは変化しません。

SCASB／SCASW　Scan String Byte／Scan String Word
[書式]
SCASB
SCASW
SCASBはES：DIで示されるアドレスのメモリデータとALレジスタとを比較し結果のフラグをセットした後、DIレジスタを更新します。D

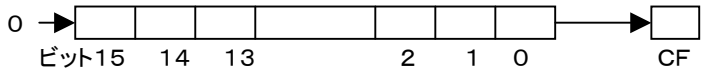
Fフラグ＝0のとき、DIは＋1し、DFフラグ＝1のときDIは－1します。
SCASWはES:DIで示されるアドレスのメモリデータとその次のアドレスのメモリデータをAXレジスタと比較し結果のフラグをセットした後、DIレジスタを更新します。DFフラグ＝0のとき、DIは＋2し、DFフラグ＝1のときDIは－2します。
REPプリフィックスと組み合わせることもできますが、MOVSと異なりそれほど簡単にはなりません。REPとは組み合わせないほうがよいでしょう。
［例］
8000～8FFFのメモリ1000H（4096）バイトを検索し最初にAL（ここでは0D）と一致したアドレスを検出する。

```
MOV  DI, 8000
MOV  CX, 1000
MOV  AL, 0D
CLD                    DF＝0にする。次にSTDが実行されるまでDF＝0のままになる。
LP1:SCASB
JZ  STEND
DEC  CX
JNZ  LP1
```

［フラグ］
比較の結果OF、SF、ZF、AF、PF、CFが変化します。
DF、IF、TFは変化しません。

SHR　　Shift Right
［書式］
①SHR reg, 1／SHR reg
②SHR mem, 1／SHR mem
③SHR reg, CL
④SHR mem, CL
①②は8ビットまたは16ビットのデータを右に1ビットシフトするとともにビット0がCF（キャリーフラグ）に入ります。最上位ビットには0が入れられます。
8ビットの場合

8ビットデータを右にシフトし、ビット0はCFに入ります。ビット7には0が入ります。
16ビットの場合



16ビットデータを右にシフトし、ビット0はCFに入ります。ビット15には0が入ります。
③④ではCLの値が右シフトのビット数（回数）になります。
②④では[ ]B（8ビット）、[ ]W（16ビット）の指定が必要です。
［例］

```
SHR  AL
SHR  [BX]W
SHR  BX, CL
SHR  [DI]B, CL
```

［フラグ］
CFにはシフト前のレジスタ、メモリのビット0の値が入ります。
①②ではシフトの対象になる値が符号付の数と考えたとき、シフト前とシフト後で符号が変わった場合にOF＝1になります。符号が変化しなかったときはOF＝0になります。
シフト前の値が80（－128）のとき、シフト後は40（＋64），CF＝0，OF＝1になります。
③④ではOFは不明です。
結果の値によってSF、ZF、PFが変化します。
AFは不明です。
DF、IF、TFは変化しません。

STC　　Set Carryflag
CFフラグ＝1にします。
［フラグ］
CF＝1になる以外は変化しません。

STD　　Set Directionflag

DFフラグ＝1にします。
DFフラグが1のとき、ストリング命令（MOVS、CMPSなど）でインデックスレジスタの値が－1または－2されていきます。
［フラグ］
DF＝1になる以外は変化しません。

STI　　Set Interruptflag

IFフラグ＝1にします。
IFフラグが1にセットされると、マスク可能割り込みが受け付けられるようになります。
［フラグ］
IF＝1になる以外は変化しません。

STOSB／STOSW　　Store String Byte／Store String Word

［書式］
STOSB
STOSW
STOSBはALの値をES：DIで示されるアドレスに転送した後、DIレジスタを更新します。DFフラグ＝0のとき、DIは＋1し、DFフラグ＝1のときDIは－1します。
STOSWはAXの値をES：DIで示されるアドレスおよびその次のアドレスに転送した後、DIレジスタを更新します。DFフラグ＝0のとき、DIは＋2し、DFフラグ＝1のときDIは－2します。
REPプリフィックスと組み合わせることで任意のバイト数のメモリブロックを指定値で埋めるプログラムが簡単に記述できます。
［例］
2000～2FFFのメモリ内容1000H（4096）バイトにFFを書きこむ。

```
MOV  DI, 2000
MOV  CX, 800
CLD                    DF＝0にする。次にSTDが実行されるまでDF＝0のままになる。
MOV  AX, FFFF
LP1：STOSW
DEC  CX
JNZ  LP1
```

下は上のプログラムをREPプリフィックスを使って書いたものです

```
MOV  DI, 2000
MOV  CX, 800
CLD
MOV  AX, FFFF
OR   CL, CL       ZF＝1にする。REPZとSTOSとの組み合わせでは必要（ZF＝0ではREPZは実行されない）
REPZ              ZF＝1のとき、STOSWをCX＝0になるまで繰り返す。
STOSW
```

［フラグ］
変化しません。

SUB　　Subtract

［書式］
①SUB reg1, reg2
②SUB reg, mem
③SUB mem, reg
④SUB reg, data
⑤SUB mem, data
第1オペランドから第2オペランドを減算して第1オペランドのregまたはmemに格納します。第2オペランドの値は変化しません。
①はreg1＝reg2＝8ビットまたはreg1＝reg2＝16ビットです。8ビットと16ビットの混合はできません。
⑤のmemの表記では
SUB ［BX］B, data8
SUB ［BX］W, data16
のようにB（Byte）またはW（Word）をつける必要があります。
［例］

```
①SUB  DL, AH
  SUB  BX, SI
②SUB  BH, ［BX］     ［BX］Bにする必要は無い
  SUB  CX, ［DI］     ［DI］Wにする必要は無い
```

③SUB ［BP］, CL　　［BP］Bにする必要は無い
　SUB ［SI］, AX　　［SI］Wにする必要は無い
④SUB BL, 45
　SUB DI, 3000
⑤SUB ［BP＋DI］B, 12　［ ］Bが必要
　SUB ［DI］W, 78AB　　［ ］Wが必要
［フラグ］
計算の結果の値によってOF、SF、ZF、AF、PF、CFが変化します。
DF、IF、TFは変化しません。

TEST
［書式］
①TEST reg1, reg2
②TEST reg, mem
③TEST mem, reg
④TEST reg, data
⑤TEST mem, data
第1オペランドと第2オペランドのAND（論理積）を計算して結果をフラグレジスタに示します。ANDと異なり第1オペランド、第2オペランドの値は変化しません。
レジスタ、メモリの内容を壊さないで、特定ビットが1であるか否かをテストします。
①はreg1＝reg2＝8ビットまたはreg1＝reg2＝16ビットです。8ビットと16ビットの混合はできません。
②③ではレジスタが8ビットのときはメモリ1バイトとの間で、レジスタが16ビットの場合にはメモリ2バイトとの間で演算が行われます。
⑤のmemの表記では
TEST ［BX］B, data8
TEST ［BX］W, data16
のようにB（Byte）またはW（Word）をつける必要があります。
［例］
①TEST DL, AH
　TEST BX, SI
②TEST BH, ［BX］　［BX］Bにする必要は無い
　TEST CX, ［DI］　［DI］Wにする必要は無い
③TEST ［BP］, CL　　［BP］Bにする必要は無い
　TEST ［SI］, AX　　［SI］Wにする必要は無い
④TEST BL, 45
　TEST DI, 3000
⑤TEST ［BP＋DI］B, 12　［ ］Bが必要
　TEST ［DI］W, 78AB　　［ ］Wが必要
［フラグ］
OF＝CF＝0になります。
計算の結果の値によってSF、ZF、PFが変化します。
AFは不明です。
DF、IF、TFは変化しません。

WAIT
［書式］
WAIT
コプロセッサが命令を実行し終わるまで待機します。このアセンブラはコプロセッサを対象にしていないため、WAITは使用しません。
［フラグ］
変化しません。

XCHG　　Exchange
［書式］
①XCHG reg, reg
②XCHG reg, mem
第1オペランドと第2オペランドの値を交換します。2つのオペランドのサイズ（8ビットまたは16ビット）は同じでなければいけません。
［例］
XCHG AX, BX
XCHG CL, DH

XCHG DI, [BX]　　[BX]Wにする必要はありません
[フラグ]
変化しません。

XLAT　　Translate
[書式]
XLAT
メモリに置かれたテーブルからデータを取り出してALレジスタに格納します。
この命令の実行前にテーブルの先頭アドレスをBXレジスタに、またそのテーブルの先頭から何番目のデータかを示す数値をALに入れておきます。1バイトのコード変換のための命令です。
[例]
TBL1:“0123456789ABCDEF”

MOV BX, TBL1
XLAT　　ALの値(00～0F)をASCIIコード(30～39、41～46)に変換します
[フラグ]
変化しません。

XOR　　Exclusive Or
[書式]
①XOR reg1, reg2
②XOR reg, mem
③XOR mem, reg
④XOR reg, data
⑤XOR mem, data
第1オペランドと第2オペランドのXOR(排他的論理和)を計算して第1オペランドのregまたはmemに格納します。第2オペランドの値は変化しません。
①はreg1＝reg2＝8ビットまたはreg1＝reg2＝16ビットです。8ビットと16ビットの混合はできません。
②③ではレジスタが8ビットのときはメモリ1バイトとの間で、レジスタが16ビットの場合にはメモリ2バイトとの間で演算が行われます。
⑤のmemの表記では
XOR [BX]B, data8
XOR [BX]W, data16
のようにB(Byte)またはW(Word)をつける必要があります。
[例]
①XOR DL, AH
　XOR BX, SI
②XOR BH, [BX]　　[BX]Bにする必要は無い
　XOR CX, [DI]　　[DI]Wにする必要は無い
③XOR [BP], CL　　[BP]Bにする必要は無い
　XOR [SI], AX　　[SI]Wにする必要は無い
④XOR BL, 45
　XOR DI, 3000
⑤XOR [BP＋DI]B, 12　[ ]Bが必要
　XOR [DI]W, 78AB　　[ ]Wが必要
[フラグ]
OF＝CF＝0になります。
計算の結果の値によってSF、ZF、PFが変化します。
AFは不明です。
DF、IF、TFは変化しません。

[参考]よく使われるMSDOSのファンクションコール

MSDOSにはマシン語プログラムで利用できる基本的なシステムサブルーチンがINT××の形で組込まれています。BIOSの一種ですがMSDOSではファンクションコールと呼んでいます。

ここではそのうちでも良く使われると思われるINT21タイプの代表的なものを示します。詳細についてはMSDOSの解説書を参照して下さい。

ここで説明するINT21はすべてAHレジスタに機能を示す数値を入れてINT21命令を実行します。他のレジスタにも引数を入れる必要があるものもあります。レジスタに値が入ってリターンしてくるものもあります。

INT××はソフトウェア割り込みですがサブルーチンコールだと思えば良いでしょう。

[例]文字の画面表示(DLのASCIIコードによって示される文字が画面に表示される)

```
MOV  DL, 41
MOV  AH, 02
INT  21
```

AH＝01　　キーボードから1文字入力し、同時に画面に表示する。キーが押されるまでリターンしない
　　CTRL＋Cでシステムブレイクする
リターンデータ　AL＝入力コード(ASCII)

AH＝02　DL＝文字コード(ASCII)　　文字を画面に表示する

AH＝05　DL＝文字コード(ASCII)　　文字をプリンタに出力する

AH＝06　DL＝FF　　キーボードから1文字入力する。画面には表示しない。CTRL＋Cは無視する
リターンデータ　キーが押されていたとき　AL＝入力コード(ASCII)　ZF＝0
　　キーが押されていないとき　AL＝00　ZF＝1

AH＝06　DL＝文字コード(≠FF)　　文字を画面に表示する

AH＝08　　キーボードから1文字入力する。画面には表示しない。キーが押されるまでリターンしない
　　CTRL＋Cでシステムブレイクする
リターンデータ　AL＝入力コード(ASCII)

AH＝09　DX＝文字列の先頭アドレス　　文字列を画面に表示する(文字列の終わりは＄が必要。＄は表示されない

AH＝2A　　日付を求める
リターンデータ　CX＝年　DH＝月　DL＝日　AL＝曜日

AH＝2C　　時刻を求める
リターンデータ　CH＝時　CL＝分　DH＝秒　DL＝0

AH＝4C　　システムに戻る

ここに挙げたものはファンクションコールの一部です。実際にはファイルアクセスを含めてたくさんの機能が利用できます。ファンクションコールの解説だけで本が1冊できてしまうほどの量があります。限られたスペースではとてもそのすべてを説明することはできません。詳しく知りたいかたはMSDOSのファンクションコールについての専門書を参照して下さい。

エラーコード

ASM86コマンド実行時にエラーが発生すると、ERR に続いて2桁のエラーコードを表示したあと、エラーの発生した行を表示します。エラーの内容によってはアセンブル作業をただちに中止してしまうため、すべてのエラーが一時に表示されるとは限りません。エラーを修正してASMコマンドを再度実行させると、次に発見されたエラーが表示されます。
エラーの意味は大体は下に説明する通りですが、必ずしもこの通りとは限りません。表示された行に、ここでの表現とは異なるエラーが含まれている場合もあります。

01 命令のつづりがまちがっている
02 ラベル、変数名が2重に定義された
03 ラベル、変数名が未定義か、条件ジャンプ命令の指定アドレスが±127バイトよりも遠くにあるか、JMP、CALLの指定アドレスが±32767バイトよりも遠くにある
04 ラベルが多すぎる。使用可能ラベル数最大2048個
05 ソースファイルのファイル名として指定されたファイルが無いかまたはパス名が間違っている
06 ラベル、変数名の長さが19バイトを超えている
07 文字列 “ ” が36バイトを超えているか、右の “ がない
08 条件ジャンプ命令の飛び先ラベルが±127バイトよりも遠くにある
09 欠番
10 PUSH [ ]B、POP [ ]Bは間違い。PUSH、POPは16ビットを対象とするため、PUSH [ ]W、POP [ ]Wでなければならない
11 メモリ指定[ ]の中に誤りがある。A～FまたはSIで始るラベル名に＊がついていないか16進数表現に誤りがある
12 メモリ指定[ ]の後ろにB, Wが無いかB, Wの後ろに誤りがある
13 メモリ指定[ ]Bまたは[ ]Wの後ろに誤りがある
14 MOV, ADDなどの第1パラメタに誤りがある。またはその他の命令のパラメタに誤りがある
15 MOV, ADDなどの第2パラメタに誤りがある
16 MOV, ADDなどの第1パラメタの後ろに ,（カンマ）がないか、誤りがある
17 MOV, ADDなどの第1、第2パラメタのサイズが不一致（片方が8ビットで他方が16ビット）
18 セグメントレジスタに対するMOV命令では数値の直接代入はできない
19 MOV, ADDなどの第1パラメタの後ろに ,（カンマ）がないか、誤りがある
20 ROLなどのローティト命令のパラメタに ,１ ,CL 以外が使われているか誤りがある
21 ジャンプ命令のラベル指定に誤りがある
22 行の終わりに不要な文字列がある
23 IN、OUT命令のI／Oアドレスに誤りがある。16進8ビット以外は不可
24 OUT命令の第2パラメタにAL、AX以外があるか誤りがある
25 LEA、LDS、LESの第1パラメタは16ビットレジスタ以外が使われているか誤りがある
26 メモリ指定で[BP]は誤り
27 RETの後ろに数値、変数名以外がある
28 CALLF、JMPFのアドレス指定に誤りがある
29 CALLF、JMPFのセグメント指定に誤りがある