第5章 WindowsのアセンブラMASMとLinuxのアセンブラgasの構文

本章から，WindowsのアセンブラMASMとLinuxのアセンブラgasの具体的なプログラミング方法や使い方，CPUの命令などについて説明します．ただし，前章でも述べたようにWindowsのアセンブラMASMとLinuxのアセンブラgasは，その生い立ちの違いから，同じx86系CPU用のアセンブラでありながら言語としての仕様が異なります．また，アセンブラで使われる用語もMASMとgasで異なる場合があります．

そこで，まずIntel表記のMASMの言語仕様を説明し，その後にAT&T表記のgasの言語仕様をIntel表記のMASMとの違いという形で説明することにします．

5.1 MASMのソース・ファイルの構成

アセンブラMASMのソース・ファイルは，複数のステートメント(statement)と呼ばれる文によって構成されます．

一つのステートメントは，1行で記述します．MASMでは，一つのステートメントは**図1**に示すようにname(名前)，operation(オペレーション)，operands(オペランド)，comment(コメント)の順に，この四つのフィールドから構成されています．

各フィールドは，commentのフィールドを除いて空白文字(スペースあるいはタブ)によって区切られます．commentのフィールドは，セミコロン(;)を区切りとし，セミコロンの次の文字から行末までをコメントとします．

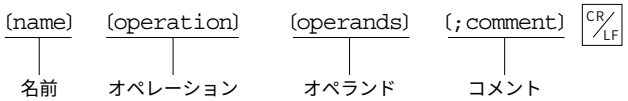
この四つのフィールドは，必要がなければ省略することができます．したがって，四つのフィールドをすべて省略した何の記述もないステートメントを作ることもできますし，四つのフィールドがすべて記述されたステートメントを作ることもできます．ただし，operandsは，operationのパラメータといった意味をもつフィールドなので，operationのフィールドがない場合，当然ですがoperandsのフィールドもありません．

5.1.1 name(名前)は変数がラベル

nameは，プログラマがデータを格納する変数や，あるいはジャンプ先のラベルを定義するのに使用します．そしてnameは，operationに記述されたディレクティブによって値や属性が定義されます(**図2**)．

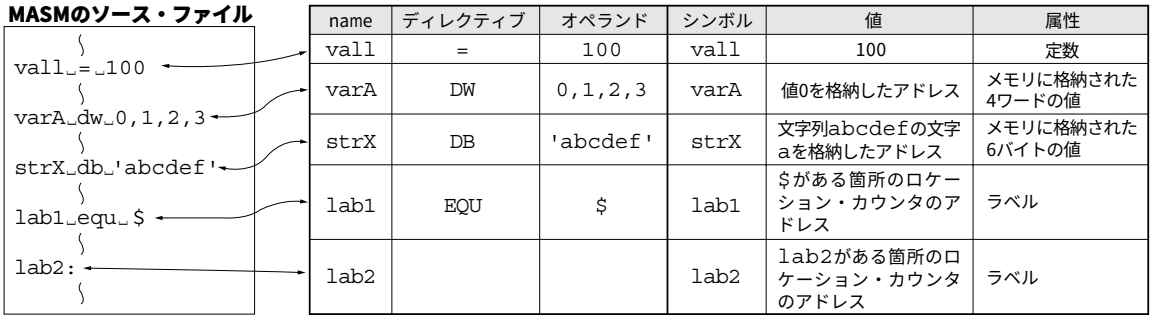
たとえば，nameが変数の場合，値は変数のアドレス，属性は変数の型(バイトかワード，ダブル・ワードなどのデータの種類，そしてデータの長さといった情報)となります．

nameがラベルの場合，値はジャンプ先となるコード(機械語命令)が格納されているアドレス，属性はラ



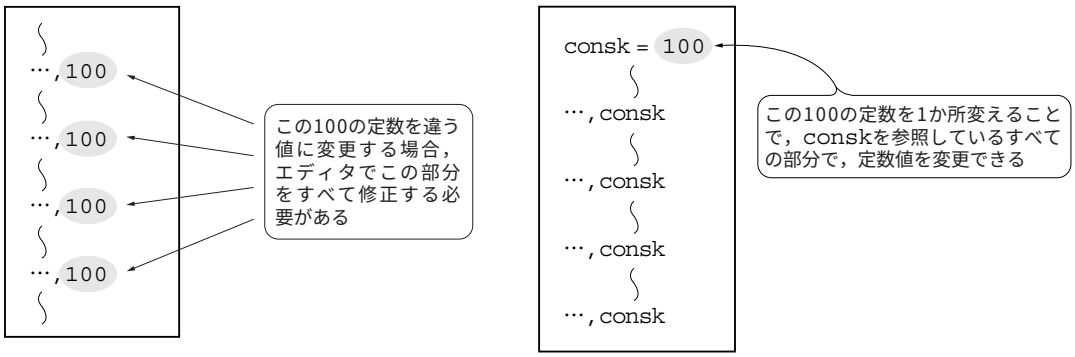
▶〔…〕の大カッコは，そのフィールドの内容を省略できることを表す
▶name，operation，operandsの各フィールドの区切りは一つ以上の空白文字(スペースあるいはタブ)で行う
▶commentの前のセミコロン(;)とほかのフィールドの間には0以上の空白文字(スペースあるいはタブ)を入れることができる

図1
MASMのステートメントの構成



name	ディレクティブ	オペランド	シンボル	値	属性
val1	=	100	val1	100	定数
varA	DW	0,1,2,3	varA	値0を格納したアドレス	メモリに格納された4ワードの値
strX	DB	'abcdef'	strX	文字列abcdefの文字aを格納したアドレス	メモリに格納された6バイトの値
lab1	EQU	$	lab1	$がある箇所のロケーション・カウンタのアドレス	ラベル
lab2			lab2	lab2がある箇所のロケーション・カウンタのアドレス	ラベル

図2　name(名前)の値と属性の定義の例



(**a**) ソース上に直接定数を記述した場合　　(**b**) ソース上のnameにディレクティブで定数を記述した場合

図3　ソース・ファイル上の定数の使用

表1　MASMのシンボル定義の規則

有効な長さ	1～247文字
使用できる文字	アルファベット(A～Z，a～z) @　_　$　? 10進数字(0～9)注

注：10進数字はシンボルの最初の文字には使用できない

ベルとなります．ラベルは，ディレクティブで定義することもできますが，operationがニモニックの場合，nameの後にコロン(:)を付けることで，ジャンプ先のラベルを定義することができます．また，ディレクティブの指定によってnameで表される定数を定義することができます．これにより，ソース・プログラムに直接，定数を記述する代わりに，その定数を示すnameで代用することができます．

ソース・プログラムに直接，定数を記述した場合，定数値を変更する際には，対象となる定数をすべてエディタで変更する必要があります．しかし，定数をnameで定義し，その定数を示すnameを使用していれば，定数値の変更があった場合でも，対象定数を定義しているnameの変更のみで済むので便利です(**図3**)．このほか，nameはoperationに記述されたディレクティブによっていろいろな値と属性をもつことになります．

nameのようにプログラマが定義する名前のほかに，アセンブラがすでに定義している名前もあります．これらはすべてシンボル(symbol)と呼びます．Ver6のMASMでは，プログラマがシンボルを定義する場合，**表1**のようにシンボルの長さと使用できる文字を決めています．

5.1.2　operation (オペレーション)はニモニックかディレクティブ

operationには，ニモニックあるいはディレクティブを記述します．

① ニモニック

ニモニック(mnemonic)は，2進数で表されるCPUの機械語命令を人間にわかりやすくするために作られた単語で，ニモニックを見れば機械語命令の動作が簡単に連想できるようになっています．ニモニックとして使われる単語は，機械語命令の動作を表す英単語が短い場合は，そのままその英単語がニモニックとして使用されます．しかし，英単語自体が長い場合は，英

単語の略語をニモニックとして使用します．また，動作が複雑でどうしても機械語命令の動作を表すのに文章となってしまう場合は，その頭文字をニモニックとしています．たとえば，加算命令の場合，ニモニックは加算を意味する英単語「ADD」がそのまま使われています．

しかし，キャリというフラグを含めた加算命令の場合，英語では「Add with carry」となり，これを略して「ADC」というニモニックになります．また，移動命令の場合，移動を意味する英単語は「MOVE」ですが，これを略した「MOV」がニモニックとして使われています．

② ディレクティブ

MASMで使用されるディレクティブには，外見上二つの種類があります．それはピリオド(.)から始まるディレクティブと，ピリオド(.)から始まらないディレクティブです．

ピリオドから始まるディレクティブは，MASMの後のバージョンで追加された新しいタイプのディレクティブです．それに対し，ピリオドから始まらないディレクティブは，MASMの初期のバージョンから使われている古いタイプのディレクティブです．

5.1.3　operands（オペランド）はパラメータか引き数

先に述べたように，operationにはパラメータ，あるいは引き数的な存在としてoperandsがあります．そのため，使用するoperationによって記述できるoperandsが異なります．

① ディレクティブとoperands

operationがディレクティブの場合，operandsはディレクティブに対する細かな動作を指定するパラメータとして機能します．

たとえば，DWというメモリ空間にワード変数を確保するディレクティブは，operandsの記述に従って，メモリ空間にワード変数のエリアを確保し，必要ならメモリ上の値を初期化します．

② ニモニックとoperands

operationがニモニックの場合，operandsはニモニックで示された命令の詳細な操作を指定するのに使用されます．

先に述べたように，ニモニックは機械語命令の動作を表すために作られた単語です．しかし，ニモニックのみで何千何万もある機械語命令をすべて規定することはできません．そこで，ニモニックでは機械語命令の機能までを表し，その詳細はoperandsを指定することで，アセンブラで使用されるニモニックの数を減らしています(**図4**)．

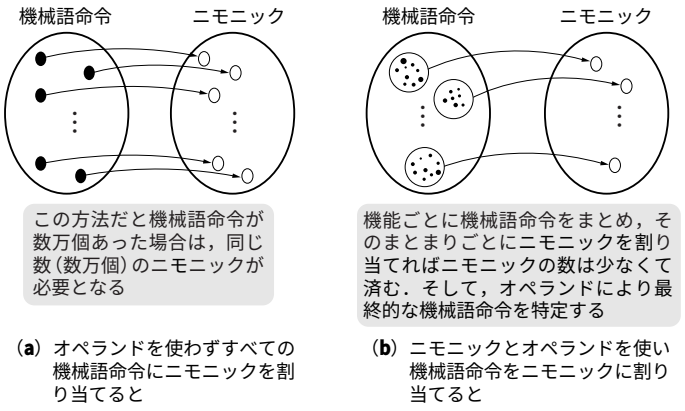
Intel社が発行しているPentiumプロセッサのマニュアルには，CPUで使用できる命令セットの記述があります．

このCPUのマニュアルによれば，16進数で表される機械語命令のことをOpcode(オペコード)，ニモニックとオペランドで表される表記をInstructionと呼んでいます．そして，マニュアルではニモニック別に，Opcodeに対応するInstructionが記述されています．

このマニュアルに記述されているInstructionは，MASMで使われているニモニックとオペランドの形式と一致しています．

③ operandsの数と順序

operationによっては，operandsが必要ない場合もありますし，逆に複数のoperandsを必要とす



(**a**) オペランドを使わずすべての機械語命令にニモニックを割り当てると

(**b**) ニモニックとオペランドを使い機械語命令をニモニックに割り当てると

図4
ニモニックとオペランド

① オペランドがない場合
operation

② オペランドが一つの場合
operation ␣ operand

③ オペランドが二つ以上の場合
operation ␣ operand,operand[,operand]…

operandとカンマ(,)との間には0以上の空白文字を入れることができる

図5　オペランドの記述

る場合もあります．operandsが必要ない場合は記述しません．複数のoperandsが必要な場合は，各operandsはカンマ(,)によって区切って記述します(**図5**)．

operationがディレクティブの場合，operandsの数は使用するディレクティブによって異なります．そのため，ディレクティブによってはoperandsがないものや，あるいは数個のoperandsを必要とするものまでさまざまです．

operationがニモニックの場合は，0～3個のoperandsが使用されます．ただし，3個のoperandsを使う命令というのはまれで，多くのニモニックでは0～2個のoperandsが使用されます．

たとえば，何の操作もしないNOPというニモニックの場合は，operandsを必要としないため，ソース・ファイルでは，

```
nop
```

とニモニックのみが記述されます．また，データを移動するMOVというニモニックは，移動という操作上，「元(source)」と「先(destination)」の指定が必要となります．そのため，MOVのニモニックの中には「元(source)」と「先(destination)」を示す二つのoperandsがあります．

MASMの場合，「先(destination)」を第1オペランド，「元(source)」を第2オペランドとします．たとえば，レジスタEDXの値をレジスタEAXに移動(実際にはコピー)する場合，ソース・ファイル上では，

```
mov    eax, edx
```

のように記述します．

また，二項演算のような命令では「元(source)」をsou，「先(destination)」をdes，演算をopとした場合，

des ← des　op　sou

のように演算します．たとえば，レジスタAXの値からメモリ上のワードの変数wval01の値を減算する場合，ソース・ファイル上では，

```
sub    ax, wval01
```

のように記述します．

逆に，単項演算のような命令では，「元(source)」と「先(destination)」は同じものになるため，operandsは一つになります．たとえば，レジスタALの値を1の補数の値にする場合，ソース・ファイル上では，

```
not    al
```

のようにoperandsを一つだけ記述します．

表2　アドレッシングとオペランド

operandの式の結果	アドレッシング
レジスタ名	レジスタ・オペランド
定数値	イミディエイト・オペランド
固定されたメモリのアドレス	直接メモリ・オペランド
固定されたメモリのアドレスまたは定数値と一つあるいは二つのレジスタに記憶されたアドレスを使ってメモリをアクセス	間接メモリ・オペランド

④ operandsの内容

operationによっては，operandsに記述される内容も違ってきます．operationがディレクティブの場合，前から述べているようにoperandsに記述する内容は，使用するディレクティブの仕様によって決まります．

operationがニモニックの場合は，operandsにはアドレッシングを表すoperandsが記述されます．その場合，operandsには式が記述されます．そして，式の結果によって，**表2**に示すようにアドレッシング別に「レジスタ・オペランド」や「イミディエイト・オペランド」，「直接メモリ・オペランド」，そして「間接メモリ・オペランド」の四種類のoperandsに分類されます．

また，operandsから扱うバイトやワードといったデータのサイズも決定されます．

▶レジスタ・オペランド

operandsにCPUのレジスタ名を記述することで，このレジスタ・オペランドとなります．たとえば，32ビット・レジスタならEAXやEDX，EDIといった32ビット・レジスタ名を記述することによって32ビット・レジスタをアクセスすることができます．

同じように，16ビット・レジスタならAX，CX，SIといった16ビット・レジスタ名，8ビット・レジスタならAH，AL，BL，DHといった8ビット・レジスタ名を記述することにより，16あるいは8ビット・レ

ジスタをアクセスすることができます．

▶イミディエイト・オペランド

operandsに記述された式の結果が値なら，イミディエイト・オペランドとなります．式には直接定数を指定することもできますし，定数を表すシンボルも使用できます．たとえば，レジスタEAXに10進数の230を設定する場合なら，

```
mov   eax, 230
```

と記述します．この例の場合，第1オペランドはレジスタ・オペランド，第2オペランドがイミディエイト・オペランドということになります．

▶直接メモリ・オペランド

operandsに記述された式の結果がメモリ内の固定アドレスの変数を示している場合，直接メモリ・オペランドになります．たとえば，レジスタAXの値をメモリ上のワード変数waryから8ワード離れたメモリにストアするのであれば，

```
mov   wary[2*8], ax
```

あるいは，

```
mov   wary+2*8, ax
```

と記述します．この例の場合，第1オペランドが直接メモリ・オペランドになります．

▶間接メモリ・オペランド

operandsに記述された式の結果，一つあるいは二つのレジスタに記憶されているアドレスと定数を実行時に参照演算し，アクセスするメモリのアドレスを求める方法を間接メモリ・オペランドといいます．

たとえば，レジスタAXの値をワード変数waryからレジスタEBXが示したワード離れたメモリにストアするのであれば，

```
mov   wary[ebx*2], ax
```

と記述します．この例の場合，第1オペランドが間接メモリ・オペランドになります．

5.1.4　comment（コメント）は注釈

commentのフィールドには，注釈として任意の文字列を記述できます．MASMでは，commentには8ビットJISコードのほかに，シフトJISで表された日本語も注釈として記述できます．

5.2 gasのアセンブラのソース・ファイルの構成

アセンブラgasのソース・ファイルも，複数のステートメント（statement）と呼ばれる文によって構成されます．やはり，一つのステートメントは，1行で記述します．

gasでは一つのステートメントは，**図6**のようにlabel（ラベル）から始まり，その後にdirective（ディレクティブ），あるいはinstruction（インストラクション）の記述を行います．

instructionの後には，0個以上のoperand（オペランド）が続きます．

5.2.1　注釈の記述方法は/*と*/で囲む

複数行にわたる注釈は，C言語と同じように注釈の部分を/* */で囲みます．

たとえば，

```
/*----------
   comment
-----------*/
```

のように記述することによって，複数行にわたる注釈を書くことができます．

MASMのcommentに相当する注釈を，gasでは“line comment”と呼びます．“line comment”は，注釈開始を表す文字から行末までが注釈となり，任意の文字列を記述することができます．

gasの場合，ターゲットのCPUによって注釈開始を表す文字が異なります．x86系CPU用のgasでは，シャープ記号（#）が注釈開始を表す文字となります．

gasの注釈では，7ビットASCIIコードのほかに，EUCで表された漢字も注釈として記述することができます．

5.2.2　label（ラベル）はシンボルを表す

label（ラベル）の記述は任意ですが，labelを記述する場合は必ずlabelの後にコロン（:）を付けます

〔label:〕〔# comment〕 LF
〔label:〕.directive〔# comment〕 LF
〔label:〕instruction〔# comment〕 LF

▶〔…〕の大カッコは，そのフィールドの内容を省略できることを表す
▶各フィールドの間には必要なら空白文字（スペースあるいはタブ）を入れることができる

図6　gasのステートメントの構成

(ラベルの後がディレクティブでも).

labelを記述すると，それはシンボルとしてgasに記憶されます．シンボルで使用できるのは，a～zあるいはA～Zの英字と0～9の数字，そしてアンダスコア(_)，ピリオド(.)，ドル記号($)の三文字です．

シンボルに使われる文字の長さに制限はなく，記述した文字列すべてが有効なシンボルとなります．

gasはMASMと異なり，labelはコロン(:)で定義しただけではアドレスしか記憶しません．labelに属性をもたせるためにはコロン(:)による定義とは別に，ディレクティブによって属性を設定する必要があります．

5.2.3 directive (ディレクティブ)は .から始まる

gasのディレクティブは，必ずピリオド(.)から始まります．そのため，ソース・リストを見る場合，リスト上のディレクティブを見つけやすいという利点があります．

MASMの場合，nameフィールドのシンボルとディレクティブには関係がありました．しかし，gasのディレクティブは，左側に記述されているlabelとは，何の関係ももつことはありません．つまり，ディレクティブの記述が，左側に記述されているlabelに影響を与えないということです．

5.2.4 instruction（インストラクション）

gasのinstructionは，MASMでいうニモニックと基本的には同じです．ただし，MASMのニモニックと大きく異なるのは，instructionの末尾の文字で扱うデータのサイズを指定することです．

末尾の文字がbならバイト，wならワード，lならロング(ダブル・ワード)ということになります．たとえば，MOVの場合，データ・サイズを示す文字を付けて，

```
movb  ……  バイトMOV命令
movw  ……  ワードMOV命令
movl  ……  ロングMOV命令
```

と記述します．

これは，ラベル自体がバイトやワードといった属性をもたないため，オペランドから扱うデータのサイズを得ることができないので，このような記述になっています．

5.2.5 operand (オペランド)の記述

MASMとgasでかなり異なっているのが，このoperandの記述です．

① operandの順序

operandが複数ある場合，operandとoperandの区切りはカンマ(,)によって行われます．ここで注意しなければならないことは，operandが複数ある場合，gasでは「元(source)」と「先(destination)」の指定がMASMとは逆になるということです．つまり，第1オペランドとして「元(source)」が記述されます．そして，第2オペランドとして「先(destination)」が記述されることになります(**図7**)．

たとえば，レジスタEDXの値をレジスタEAXに移動(実際にはコピー)する場合，gasソース・ファイル上では，

```
movl   %edx, %eax
```

のように記述します．

② operandの内容

instructionでは，データをアクセスするためにアドレッシング別に「レジスタ」や「イミディエイト」，「直接メモリ参照」，そして「間接メモリ参照」の四種類のoperandをもちます．この点はMASMと同じです．ただし，その記述方法はMASMとは大きく異なります．

▶レジスタ

レジスタ名の前にパーセント記号(%)を付けます．たとえば，レジスタEAXなら%eax，レジスタBXなら%bx，レジスタCHなら%chのようになります．

▶イミディエイト

値の前にドル記号($)を付けます．たとえば，レジスタAXに10進数の200を設定する場合，gasのソー

※⇐はデータの移動方向を表している

(**a**) Intel表記　(**b**) AT&T表記

図7 gasのinstructionのオペランドの記述

ス・ファイル上では，

```
movw    $200, %ax
```

のように記述します．また，labelで定義されたdat1というラベルのアドレスをイミディエイトの値として，レジスタEBXに欲しい場合は，

```
movl    $dat1, %ebx
```

と記述します．

▶直接メモリ参照

参照するメモリのアドレスを示すラベルを記述することで，直接メモリ参照となります．たとえば，labelで定義されたdat1というラベルが示すアドレス上の値をリードし，レジスタEAXにロードする場合は，

```
movl    dat1, %eax
```

と記述します．

▶間接メモリ参照

MASMでは，

```
disp[base+index*scale]
```

あるいは，

```
[base+index*scale+disp]
```

となる間接メモリ参照ですが，gasの場合は，

```
disp(base,index,scale)
```

と記述します．

ここで，dispはディスプレースメント(displacement)，baseはベースとなるレジスタ，indexはインデックスとなるレジスタ，そしてscaleはindexの値に乗算する倍率(1，2，4，8のいずれかの値のみ使用可能)となります．

disp，base，index，scaleは，必要がなければ省略することができます．たとえば，MASMでは，

```
mov    eax, [ebp-8]
```

となるステートメントは，gasなら，

```
movl   -8(%ebp), %eax
```

となります．

また，MASMでは，

```
mov    eax, var1[edx*4]
```

となるステートメントは，gasでは，

```
movl   var1(, %edx, 4), %eax
```

となります．

5.3 x86系CPUのアドレッシング・モード

CPUがメモリをアクセスするとき，アクセス対象となるアドレスの生成方法を指定するのがアドレッシング・モードです．

x86系CPUのアドレッシング・モードは，イミディエイト(即値)オペランド，レジスタ・オペランド，そしてメモリ・オペランドの3種類です．

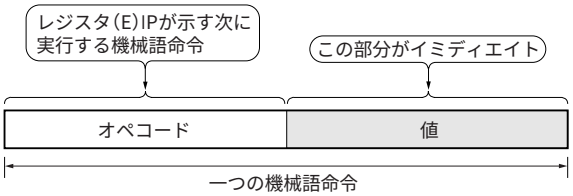


図8　イミディエイト・オペランド

5.3.1　イミディエイト(即値)オペランド

Pentiumのマニュアルでは「即値オペランド」という用語が使われていますが，「即値」とは「イミディエイト」の日本語訳です．現在，「即値」という言葉よりも「イミディエイト」という言葉が一般的によく知られているので，ここでも「イミディエイト」を使うことにします．

イミディエイト・オペランドでは，アクセス対象となる値が機械語命令の内部にあります(**図8**)．つまり，イミディエイト・オペランドでは，機械語命令内にアクセス対象の値が埋め込まれているわけです．そのため，イミディエイト・オペランドでは，値はリードのみで，ライトによって値の変更はできません．したがって，イミディエイト・オペランドは定数を必要とする場合に使用されます．

386以降のx86系CPUでは，イミディエイト・オペランドとしてバイト，ワード，ダブル・ワードの値が使用できます．これらの値の符号の扱いは，機械語命令(オペコード)により「符号なし」か「符号付き」かが区別されます．

5.3.2　レジスタ・オペランドはCPUのレジスタをアクセス対象にする

CPUのレジスタをアクセス対象にするのがレジスタ・オペランドです(**図9**)．CPUのレジスタの構成については，第3章を参照してください．

レジスタ・オペランドとしてアクセスできるレジスタには，**表3**のような汎用レジスタ，セグメント・レジスタ，命令ポインタ，フラグ・レジスタや浮動小数点演算ユニットのレジスタ，そしてシステム・レジスタなどがあります．

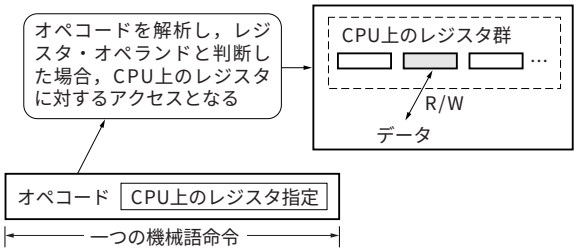


図9　レジスタ・オペランド

表3　レジスタ・オペランドでアクセスできるレジスタ

汎用レジスタ	32ビット・レジスタ EAX，EBX，ECX，EDX，ESI，EDI，EBP，ESP
	16ビット・レジスタ AX，BX，CX，DX，SI，DI，BP，SP
	8ビット・レジスタ AH，AL，BH，BL，CH，CL，DH，DL
セグメント・レジスタ	CS，DS，ES，SS，FS，GS
命令ポインタ	EIP，IP
フラグ・レジスタ	EFLAGS，FLAGS
浮動小数点演算ユニット	レジスタ・スタック［ST(0)～ST(7)］ コントロール・レジスタ ステータス・レジスタ，タグ・ワード 命令ポインタ，データ・ポインタ
システム・レジスタ	システム・フラグ GDTR，IDTR，TR，LDTR 制御レジスタ［CR0～CR4］ デバッグ・レジスタ［DR0～DR7］

① 汎用レジスタは3種類ある

386以降のCPUでは，汎用レジスタとして32ビット(EAX，EBX，ECX，EDX，ESI，EDI，EBP，ESP)，16ビット(AX，BX，CX，DX，SI，DI，BP，SP)，8ビット(AH，AL，BH，BL，CH，CL，DH，DL)の3種類のレジスタが，レジスタ・オペランドとしてアクセスできます．

また，MULやDIVといった乗除算命令では，16ビット・レジスタあるいは32ビット・レジスタを二つ使って，32ビット長の値や64ビット長の値を扱います(**図10**)．

② セグメント・レジスタ

セグメント・レジスタ(CS，DS，ES，SS，FS，GS)も，レジスタ・オペランドとしてアクセスできます．

x86系CPUでは，セグメントという塊でメモリ上の機械語命令やデータを管理しています．このセグメントを示すのがセグメント・レジスタの役割です．そのため，機械語命令がメモリ上の値をアクセスする場合，かならずこのセグメント・レジスタが参照されます(**図11**)．したがって，セグメント・レジスタはプログラムで自由に読み書きできるようになっています．ただし，実際問題としてOSによりセグメント・レジスタの書き込みが許可されていない場合があります．

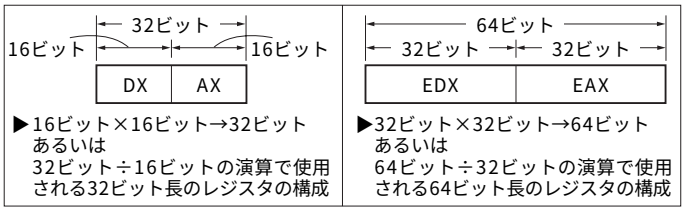


図10
乗除算とレジスタ・オペランド

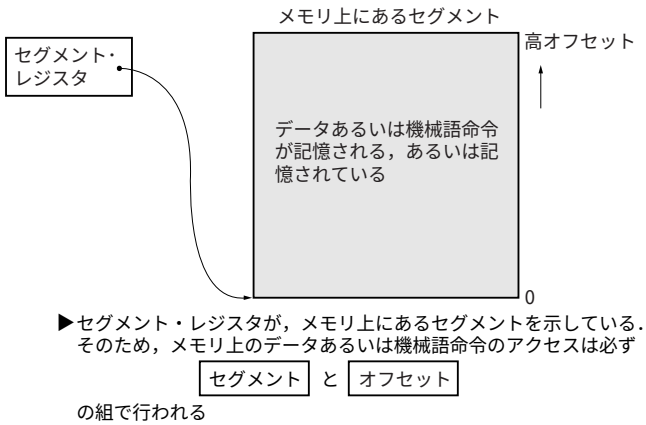


図11
セグメントとセグメント・レジスタ

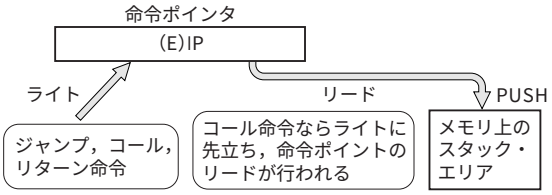


図12　命令ポインタのアクセス

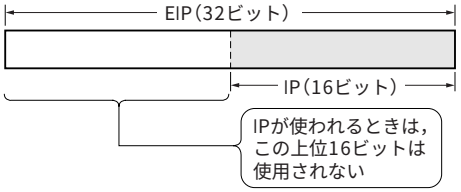


図13　EIPとIP

たとえば，MS-DOSやWindows95以前のWindowsの場合は，アプリケーションによるセグメント・レジスタの書き込みを許していました．しかし，Windows 95以降のWindows(Windows NT/2000/XPを含む)やLinuxでは，アプリケーションによるセグメント・レジスタの書き込みを許していません．

このセグメント・レジスタのアクセスは，OSに合ったコンパイラ言語を使用している分には，コンパイラが自動的に処理してくれるため，ほとんど注意する必要がありません．しかし，アセンブラでプログラミングする場合，セグメント・レジスタのアクセスもプログラマが記述する必要があるため，OSがセグメント・レジスタの書き込みを許しているかどうかには注意する必要があります．

誤ってセグメント・レジスタを書き換えてしまうと，別のアドレスをアクセスすることになり，誤ったデータの読み書きやプログラムの暴走といった不具合が発生することになります．

③ 命令ポインタとフラグ・レジスタ

命令ポインタ(EIPあるいはIP)のレジスタは，ジャンプ命令やコール命令(割り込みを含む)，リターン命令によってアクセスされます．ジャンプ，コール，リターンのすべての命令は，命令ポインタへの書き込みとなります．

コール命令(割り込みを含む)は，命令ポインタに新しい値を書き込む前に，それまでの値を読み出してスタックに保存します．この操作により，中断した処理に後でまた戻れるようにしています(**図12**)．

命令ポインタとしてEIPの32ビット・レジスタが使われるのか，あるいはIPの16ビット・レジスタが使われるのかは，CPUの実行モードによって異なります．

CPUがプロテクト・モードで動作し，32ビット属性をもったコード・セグメントで実行されている場合は，EIPの32ビット・レジスタが命令ポインタとして使われます．CPUがリアル・モードで動作している場合やプロテクト・モードで動作している場合でも，16ビット属性をもったコード・セグメントで実行されているときや仮想8086モードで実行されているときは，IPの16ビット・レジスタが命令ポインタとして使われます(**図13**)．

フラグ・レジスタ(EFLAGSあるいはFLAGS)は，演算結果の状態を表すステータス・フラグと，命令の実行を制御するためのコントロール・フラグ，そしてCPUを制御するためのシステム・フラグから構成されています．

使用頻度の高いフラグについては，フラグのセット/クリア，テストを行う機械語命令が用意されているため，フラグ・レジスタのアクセスは，その命令を使います．また，フラグ・レジスタの下位8ビットは，レジスタAHとの間でロード/ストアすることができます．

EFLAGSあるいはFLAGS全体のアクセスは，PUSH/POP命令を使い，スタックを経由する方法でアクセスします(**図14**)．

④ 浮動小数点演算ユニットのレジスタ

浮動小数点演算ユニット(FPU)には，実数値を記憶するためのレジスタ・スタックとFPUを制御するためのコントロール・レジスタ，そして状態を表すステータス・レジスタなどがあります．レジスタ・スタックには，80ビットの拡張精度実数が格納できるレジスタが8本あります．レジスタ・スタックのアクセスは「スタック」という名のとおり，値のロード/ストアはスタック動作を行います．そして，二項演算は，スタック・トップの値とスタック上の指定レベルの値とで行います(**図15**)．

コントロール・レジスタとステータス・レジスタにはアクセス専用の命令があり，その命令を使って状態を読み書きします．

FPUには，このほかにレジスタ・スタック上の8本のレジスタの内容を示しているタグ・ワードというレジスタもあります．また，FPUが最後に実行した機械語命令のアドレスを示している命令ポインタ，

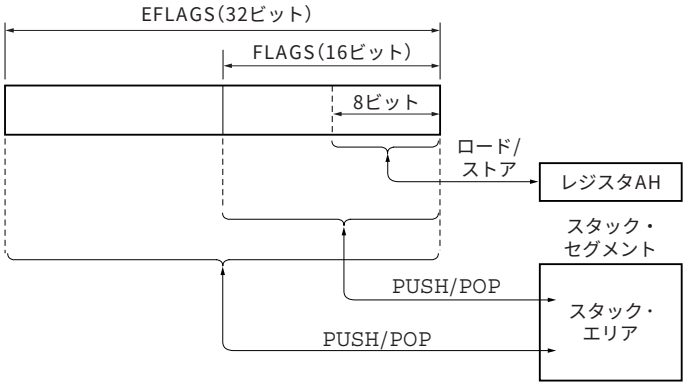


図14
フラグ・レジスタのアクセス

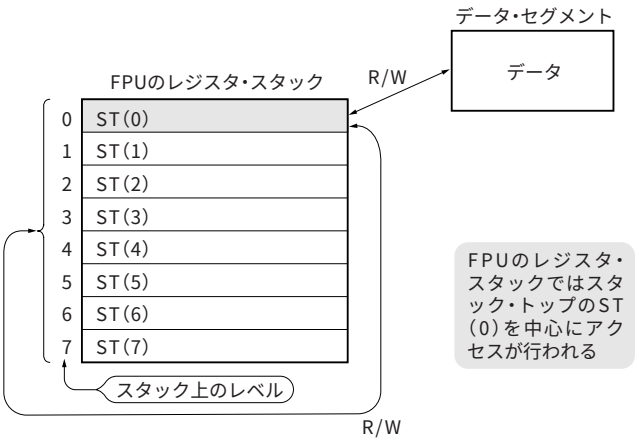


図15
FPUのレジスタ・スタックのアクセス

FPUが最後にメモリ上のデータをアクセスしたときのアドレスを示すデータ・ポインタがあります．

これらのレジスタへは，単体ではアクセスできません．FPUには，制御関係のレジスタ全部に対するロード/ストア命令，あるいはFPU上の全レジスタに対するロード/ストア命令を使うことで，アクセスできます(**図16**)．

⑤ システム・レジスタ

システム・レジスタは，CPUを制御するためのレジスタです．システム・レジスタによってCPUの実行モードや例外の管理，仮想記憶のためのページングのサポートなどが行えます．そのため，システム・レジスタは，OSなどのシステム・プログラムで使用し，OSの下で実行される一般アプリケーション・プログラムでは使用できません．

CPUは，システム・レジスタのアクセスに対し制限を設けています．CPUがリアル・モードか，プロテクト・モードの場合は，特権レベルの高いプログラムでないとアクセスできないようになっています．したがって，WindowsやLinux上で動作するアプリケー

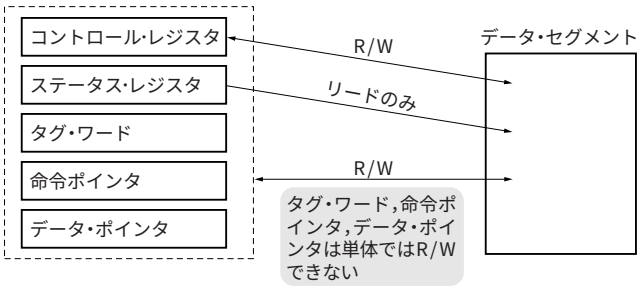


図16
FPU上の制御関係のレジスタのアクセス