

【機械語序論 3 回目 2002・12・17】

今回の説明を始める前に、google で探した便利なサイトの情報の紹介です。

<http://www.bekkoame.ne.jp/~bero/docj/gas.txt> gnu のアセンブラのマニュアルです。

<http://paran0ia.virtualave.net/documents/asm.html> アセンブラ入門講座です。

ちなみに、gnu assembler (gas) については、info コマンドを使ってみる事ができます。

```
% info -s as.info
```

また、Pentium などの IA32 アーキテクチャの命令セットのマニュアルは、インテルのサイト

<http://www.intel.co.jp/jp/developer/download/index.htm>

から、ダウンロードすることができます。命令セットのマニュアルだけでも、800 ページになる膨大なものなので、くれぐれも、プリントしないように! ここには、いろいろなマニュアルがあります。

x86 アセンブリ言語のオペランドの種類と記法 (その2)

前回、オペランドについて、レジスタ、測地、アドレス参照について説明しました。アドレス参照では、単にアドレスを参照する場合を説明しました。例えば、

```
mov 100,%eax
```

とした場合には、100 番地から始まる 4 バイト整数を eax レジスタにロードします。アドレスの代わりにラベルを使った場合にはラベルが定義されたアドレスを意味します。例えば、

```
.data
x: .long 0
.text
mov x, %eax
```

これは、データ領域に定義された変数に x というラベルをつけて、これを参照しています。x というシンボルはアセンブラではアドレスを意味します。

さて、これだけだと配列のように、計算した値でメモリを参照することができません。x86 は非常に強力なアドレスを計算することができます。メモリ参照にどのような計算を行ってメモリ参照を行うかという指定をアドレッシングモードといいます。x86 では、以下のアドレッシングができます。

BASE+(INDEX*SCALE)+DISPLACEMENT

- **DISPLACEMENT:** アドレス、もしくはベースレジスタからのオフセットを示す。定数。
- **BASE:** ベースレジスタ
- **INDEX:** インデックスレジスタ、配列の添え字のようなもの
- **SCALE:** 2,4,8 のいずれか。配列要素のサイズを指定する。

これを、アセンブラでは、以下のように記述します。

DISPLACEMENT(BASE,INDEX,SCALE)

これらの一部でもかまいません。例えば、前回説明したアドレス指定とは、DISPLACEMENT のみの場合です。ベースレジスタとインデックスレジスタには、どのレジスタを指定してもかまいません。いくつか例をしめしましょう。実際、アクセスされるアドレスを実効アドレスといいます。

(%ebx) ebx の値をアドレスとしてメモリにアクセスする。

4(%ebx) ebx の値に 4 を加えたものが実効アドレス

a(%edi) ラベル a のアドレスに、edi の値を加えたものが実効アドレス

(%edi,%ebx,2) edi+ebx*2 が実効アドレス

a(,%ecx,4) a+ecx*4 が実効アドレス

(%ebx,%edi) ebx+edi が実効アドレス

SCALE が省略された場合は、SCALE は 1、つまり、SCALE は無効になります。

配列とアドレッシングモード

アドレッシングモードを使って配列の計算を考えてみましょう。右のプログラムは、データ領域に、1 から 10 までのはいった配列 a を作り、ebx にその合計をいれて終了するプログラムです。

前回説明した.long は 1 つのデータだけでしたが、複数個書くことによって、データ全部が.long で確保されます。この先頭に、ラベル a を置いています。eax は 1 ずつ増えていきますので、これを

```
.data
.align 4
a: .long 1,2,3,4,5,6,7,8,9,10
.text
.globl main
main:
    mov $0,%eax
    mov $0,%ebx
L1: cmp $10,%eax
    je L2
    add a(,%eax,4),%ebx
    add $1,%eax
    jmp L1
L2: call stop
```

index にして、

```
add a(%eax,4),%ebx
```

で、ebx に加算しています。

同じプログラムを別の形で書いてみました。まず、edx に a のアドレスをいれて、これを 4 ずつ増やしていきます。ebx に a を入れるためには、即値ですから、

```
mov $a,%edx
```

とします。edx の内容でメモリ参照するには

```
add (%edx), %ebx
```

と書きます。

```
main:
    mov $0,%eax
    mov $0,%ebx
    mov $a,%edx
L1:  cmp $10,%eax
    je L2
    add (%edx),%ebx
    add $4,%edx
    add $1,%eax
    jmp L1
L2:  call stop
```

状態フラグと分岐命令 (その2)

前回では、簡単な分岐命令 je だけを説明しました。x86 では、cmp 命令を行って、je 命令で分岐します。cmp 命令は、実際には sub 命令と同じく、右オペランドから左オペランドを減算し、結果を捨てる命令です (そのために、左のオペランドは即値は使えない?)。実は CPU の中には演算結果について、以下の状態を保持する状態フラグという隠れたレジスタがあります。状態フラグには以下のものがあります。

- OF (オーバフローフラグ) 結果が数字の範囲の正または負の限度を超えたとき 1
- SF (サインフラグ、符号) 結果が負である場合 1 (結果の最上位ビットが 1)
- ZF (ゼロフラグ) 結果がゼロである場合 1 (結果の全部のビットが 0)
- CF (キャリーフラグ) 最上位のビットからの繰り上がりがある場合 1

cmp 命令のあとで実行する命令 je は、ZF が 1 の時分岐する命令なのです。mov 命令以外のほとんどの演算命令では、その結果で上のフラグをセットするので、演算命令を実行したあとで、条件分岐命令を実行することができます。例えば、

```
add $1,%eax
```

```
je L
```

では、%eax が 0 になったときに分岐することになります。

フラグの意味を理解するためにはコンピュータの中でどのように数が表現されているかを理解する必要があります。コンピュータの中では、負の数は 2 の補数表現という方法を使って、表現されています。これは数 x の負の数を n ビットで表現する場合に、 $2^n - x$ で表現する方法です。

$$y - x = y + (-x) = y + (2^n - x) = 2^n + (y - x)$$

となるので、n ビットで表現している場合、 2^n は無視 (キャリー、桁上がりになる)すると、減算は 2 の補数を加えればよいこととなります。2 の補数は、ビットを反転 (これを 1 の補数という)に加えて作ることができます。

$$x \quad 2^n - 1 - x \quad (\text{反転、足してすべて } 1) \quad 2^n - 1 - x + 1 \quad (1 \text{ を加える}) = 2^n - x$$

この表現については、資料 1 を参考にしてください。

2 の補数表現では、このように減算を加算で行うことができるほか、最上位ビットが 1 の場合は負、0 の場合は正になります。これを示すのがサインフラグです。オーバフローフラグは、演算結果がビット (32 ビット) で表現できる範囲を超えたことを示すもので、例えば、正の数と正の数を加えて、結果が負になったり、その逆の場合に 1 になります。

資料 2 にこのフラグを使った分岐命令について示します。ここで、符号付の数の比較と符号なし (C 言語で unsigned) の数の比較が違う命令になっていることに注意してください。符号なしの場合には、桁上りを示すキャリーフラグを判定しているのに対して、符号付の数の場合には結果のサインフラグを見るほか、オーバフローしているかをみています。オーバフローしている場合は、その結果は反転しなくてはならないこととなります。

今回やったことのまとめ

アドレッシングモード、2 の補数表現、条件フラグといろいろな分岐命令