[機械語序論　3 回目]

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://www5c.biglobe.ne.jp/~ecb/assembler/assembler00.html　x86 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ことができます。

% 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、即値、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、SCLAE 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 をつくり、ebx にその合計をいれて終了するプログラムです。

```
.data
.align 4
a:  .long1,2,3,4,5,6,7,8,9,10
.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
```

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 index にして、

```
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
```

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。edx に a をいれるためには、即値（ラベル a に与えられた番地を直接使う）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

のように書きます。（edx が指すメモリの内容を ebx にコピー）

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、右のオペランドには即値は使えない）。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）　結果が符号付整数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)　符合付整数の演算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または繰り下がりがある場合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。

$y - x = y + (-x) = \mathbf{y + (2^n - x)} = 2^n + (y - x)$

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の補数を加えればいいことになります。２の補数は、ビットを反転（これを１の補数という）し、これに１をに加えて作ることができます。

$x \rightarrow 2^n - 1 - x$ （反転、足してすべて１） $\rightarrow 2^n - 1 - x + 1$ （１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、演算結果がその大きさのビット（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、桁上が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、オーバーフローしているかをみています。オーバーフローしている場合は、その結果は反転しなくてはいけないことになります。

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

課題２

１、フィボナッチ数 fib(10)を、 ebx に格納して終了するプログラムを作りなさい。
フィボナッチ数 fib(n)とは、負でない数 n に対して、
n が０もしくは１のときは、1
それ以外の場合は、fib(n) = fib(n-1)+fib(n-2)
で定義される数である。
実行結果についても、提出すること。

ヒント： fib(n)を格納する配列を用いて、fib(2),fib(3), …と順番に求めていけばよい。

２、データ領域に３２ビットの２つの変数 x と変数ｙを宣言し、これを加算して、オーバーフローしている場合には、eax に１、オーバフローしていない場合には eax に０をセットして終わるプログラムを作りなさい。また、変数ｘと変数ｙの値を適当な値にセットし、オーバーフローフラグがセットされることを確かめなさい。提出はこのオーバフローフラグがセットされる値がセットされる例を、実行結果とともに提出すること。

アセンブラプログラムのデバックの方法

アセンブラプログラムのデバックは、gdb（gnu debugger）を使って行うことができます。

gdb の起動

gdb は、単独でも起動することができますが、emacs から起動すると便利です。実行プログラムを a.out とすると、まず、emacs から、

```
M-x gdb
```

と入力します。`Run gdb (like this): gdb` とプロンプトでるので、ここで、a.out と入力し、リターンします。そこで、gdb の window が開かれるはずです。

ブレークポイントの設定と実行開始

課題のプログラムは main から始まるので、まず、ここで停止するように、break コマンドで main にブレークポイントを設定します。(gdb)とプロンプトがでるので、ここで、

```
(gdb) break main
```

と入力します。次に、run コマンド main まで実行します。

```
(gdb) run
```

すると、実行が始まり、main で停止するはずです。

プログラムの disassemble

ここで、プログラムがどのようなコードになっているかについて、確認してみましょう。メモリ上の機械語になったプログラムをアセンブリプログラムで表示するのが disassemble コマンドです。disassemble とは、アセンブルの反対、つまり、機械語からアセンブラに直すことです。main から始まるプログラムを disassemble してみましょう。

```
(gdb) disassemble main
```

main のところに、任意のラベル名を書くことでそのプログラムを disassemble することができます。

プログラムのステップ実行

１命令づつ実行するコマンドが、stepi です。

```
(gdb) stepi
```

ここで、stepi コマンドを実行するごとに１命令づつ実行されているのがわかるはずです。

レジスタの表示

step 実行している途中で、レジスタの表示をして見ましょう。表示には２つの方法があります。

```
(gdb) info registers
```

では、すべてのレジスタの表示を行います。個別のレジスタを表示する場合には、

```
(gdb) print $レジスタ名
```

で表示させることができます。

実行の再開、ブレークポイントの設定

continue コマンドは実行を次のブレークポイントまで（もしくは終わりまで)、実行を再開するコマンドです。

```
(gdb) continue
```

さて、main にブレークポイントを設定しましたが、main の代わりにラベル名を書くことで、そのラベルの前で実行を止めることができます。また、アドレスを指定したい場合には

```
(gdb) break *アドレス
```

で任意のアドレスで実行を中断することができます。

データの表示

データの表示を行うコマンドが x コマンドです。

```
(gdb)  x  アドレス
```

で、アドレスの内容をプリントすることができます。x のあとには、データ表示のフォーマットができて、例えば、x/のあとに、表示するデータの数、10 進（ｄ)、１６進（ｘ)、８進（o）とそのあとに、b(byte), h(half), w(word)と指定します。たとえば、

```
(gdb) x/10dw  0x10000
```

では、0x10000 番地から、32 ビットごと（w）に１０進（ｄ）で、１０ワード表示するという意味になります。詳しくは、**help x** としてみてください。

他のコマンドについても、help コマンドで調べることができます。