[機械語序論　２回目]

今回はアセンブリ言語のプログラムの書き方と x86 のもっとも基本的な命令の使い方について説明します。

アセンブラ基本的な記法

前回、機械語とは何かということについて説明しましたが、機械語に１対１に記述するのがアセンブリ言語です。アセンブリ言語は、基本的に以下のような形式で記述します。

命令コードのニーモニック　オペランド１、オペランド２、オペランド３、...

オペランドの数や使えるオペランドの種類については、命令コードによって異なります。

x86 のアセンブリ記法については、AT&Ｔ記法と Intel 記法がありますが、Linux で使えるアセンブラは gnu のアセンブラ gas で、これは AT&T 記法を使っていますので、ここでは AT&T 記法について説明していきます。

x86 アセンブリ言語のオペランドの種類と記法（その１）

オペランドには、以下の種類があります。

- **レジスタ**：x86 には３２ビットの汎用レジスタとして、eax, ebx, ecx, edx, esi, edi, esp, ebp の８個のレジスタがあります。このうち、esp はスタックポインター、ebp はベースレジスタと呼ばれているもので使い方が決まっていますので（これは、関数呼び出しで説明する）、当面使えるレジスタは６個です。オペランドにレジスタを指定する場合には、レジスタ名の前に%をつけます。例えば、eax をオペランドに指定する場合には、%eax と書きます。
- **即値（イミディエト：immediate）**：これは、定数のことです。定数をオペランドに指定する場合には、最初に＄をつけます。例えば、１をオペランドに指定する場合には、$1 と書きます。これは、１０進数です。１６進数を指定する場合には、0x をつけます。また、＄のあとには＋、－などの簡単な式をかくことができます。
- **アドレス参照：**＄をつけないで、単なる数値をオペランドとして書いた場合はその数値で示されるアドレスに対するメモリ参照になります。例えば、100 と書いた場合にはアドレス 100 を参照します。また、データ領域につけた変数のラベル（名前）を書いた場合にも、そのラベルのアドレスを参照します。

このほかにも、レジスタによるメモリ参照（アドレッシングモードと呼ばれる）や ax や、al,ah などレジスタの部分指定がありますが、これについては、（その２）で解説することにします。

mov 命令

もっとも基本的な命令は mov 命令です。

mov　src、dst

src で指定されたオペランドから、dst で指定されたオペランドにコピーします。

```
mov $1,%eax    #  レジスタ eax に１をセット
mov %eax,%edi  #  eax の内容を edi にコピー
mov 100, %ebx  #  アドレス 100 の３２ビットワードを ebx にロード
mov %edx, 200  #  アドレス 200 に、edx の内容をストア
mov  x, %ecx   #  ラベル x のデータを ecx にロード
```

上説明した大抵の組み合わせはつかえますが、以下の制限があります。

src と dst の両方がアドレス参照であってはならない

ということです。例えば、mov 100,200 はエラーになります。

add 命令、sub 命令

加算を行う add 命令と、減算を行う sub 命令はもっとも基本的な演算命令です。

```
add  src, dst    # dst = dst + src
sub  src, dst    # dst = dst - src
```

x86 の命令は dst について、加算、減算を行う２オペランドの命令であることを注意してください。

```
add $1, %eax  # レジスタ eax に１を加算する。
sub 100, %edx  #  レジスタ edx から、アドレス１００の内容を減算する
add %edx, %ebx # レジスタ ebx に edx の内容を加算する。
sub %eax, x    #   ラベル x のデータから、eax の値を減算する。
```

ここでも、src と dst の両方がアドレスであってはなりません。

cc –S で得られたコードで、movl とか addl など、ｌがついている場合があります。これは、命令が扱

うデータのサイズをあらわすもので、l は 32 ビット、w は 16 ビット、b は８ビットをあらわします。たとえば、movl は 32 ビットのデータの mov 命令、movw は１６ビットのデータの mov 命令です。オペランドのどれかがレジスタで eax などの 32 ビットのレジスタの場合はデータのサイズがわかりますので、mov 命令をつかっても自動的に movl と同じと解釈されます。

アセンブラ擬似命令 (その１)

アセンブリ言語には、命令をあらわす部分のほか、記法上便宜的に導入された擬似命令があります。

- **ラベル**： 名前のあとに:をつけたものはラベルで、プログラム中の位置やアドレスを示します。これは、C 言語のラベルと同じです。
- **.text** ： プログラムのコードであることを示します。これによって、コードはまとめられて、メモリ上に格納されます。
- **.data** ： プログラムのデータ部分であることをしめします。データもデータだけまとめられて、メモリ中に格納されます。
- **.globl ラベル**： ラベルをリンク時に見えるようにします。他のファイルから参照される名前は.globlで宣言しておかなくてはなりません。
- **.word n** : 16 ビットの n という値を格納する領域を確保します。
- **.long n** : 32 ビットの n という値を格納する領域を確保します。
- **.align 4** : 4 バイトごとの境界にあわせます。

```
.data
.align 4
x: .long 100
.text
.align 4
.globl main
main: mov x,%eax
      add $1,%eax
      mov %eax, x
    ...
```

このほかにもありますが、それはこれからの講義で説明することにすることにして、右のは x という 32 ビットの領域を確保して、その領域にラベル x をつけておきます。ｘに 1 を加えるプログラムの１部です。C 言語と同じように main から始まるものとして、main は、.globl で外から参照できるようにしておきます。.align はなくても動きますが、４バイト境界にないと、効率が下がりますので、つけておきます。

あと、アセンブリ言語でのコメントは#で始めるか、Ｃと同じように/* */で囲った部分がコメントになります。

簡単な分岐命令（その１）

では、この時点で書けるプログラムを面白くするために最後に簡単な分岐命令を説明しましょう。Ｃ言語で if に相当する分岐命令は、cmp 命令と条件分岐命令の組み合わせでかきます。

```
cmp opd1, opd2
je  L
```

```
      mov $0, %eax
L1:  cmp $10,%eax
      je L2
      … # ループ本体
      add $1,%eax
      jmp L1
L2:    ....
```

je 命令は、前の cmp 命令で、比較した結果が同じであればコードにあるラベル L に分岐する命令です。例えば、

```
cmp $0,%eax
je L
```

では、eax が 0 だったら、ラベル L に分岐します。ちなみに、同じでなければ分岐する命令は jne です。なお、cmp 命令は sub 命令と同じで被減算数が右に来ることになっていますので、イミディエトの値（$10 等）は左のオペランドに書かなくてはいけません。

また、無条件に分岐する命令は、jmp 命令です。

```
jmp L
```

これは、Ｃ言語でいえば goto 文で、コード中のラベルはＣ言語と同じように、プログラム中に L:という形式で書いておきます。

右の例は、eax を 0 にしておいて、eax を１づつ加えて、eax が 10 になるまで、ループする例です。

今回やったことのまとめ：

オペランド（レジスタ、即値、アドレス）、mov 命令、add/sub 命令、アセンブラ擬似命令（.text, .data, .globl, .align, .long, .word）, cmp 命令、je/jne 命令、jmp 命令