[機械語序論　５回目　２０１１．１．２５]

前回まで、コンピュータの中での２進数による数の表現、２の補数表現について説明してきました。add,sub では、２の補数表現を使うことによって、負の数でも正の数でも同じ演算命令でできます。しかし、乗算、除算では符号つきの命令 imul,idiv と符号なしの命令 mul,div があります。これについて、ちょっとコメントしておきます。加減算命令では３２ビット同士の演算では結果は３２ビットになります（正確には３２ビット＋１ですが)。しかし、例えば乗算では３２ビット同士の結果は６４ビットになります。したがって、演算は原理的には３２ビットの数を６４ビットにして行われることになります。この時、６４ビットにするときに符号付の数の場合には符号を上位３２ビットに拡張して行い、符合なし数の場合は、上位３２ビットを０にして行います。したがって、この２つの命令があるわけです。これは除算でも同じです。しかし、３２ビット同士の乗算結果の３２ビットの範囲に収まり、その結果の３２ビットだけが必要な場合は、どちらの演算を使っても同じことになります。これについては、なぜなのかを考えてみてください。

シフト命令とローテイト命令

前回の資料では、シフト命令を説明しました。この命令も、右にシフトするときには符号付の数の場合の算術シフト **SAR(shift arithmetic right)**と、符号なしの場合の論理シフト **SHR(Shift logical right)**があります。

```
  shl  src,dst  # dst = dst << src
  shr  src,dst # dst = dst >> src 但し、シフトされた残りは０で埋める
  sar  src,dst # dst = dst >> src 但し、シフトされた残りは最上位ビットの値で埋める。
```

但し、**src は、即値もしくは、cl レジスタでなくてはなりません。**

SAR 命令は、シフトするときに符号拡張していることになります。

シフト命令の場合は、シフトされて外にはみ出した最後のビットがＣＦフラグに入ります。例えば、aex が 0111 1001 0110 1111 1001 0101 0100 0011 の場合は、

```
  shr $1,%eax
```

のあとでは CF は１になります。

これに似た命令に、ローテイト命令があります。この命令でははみ出たビットを反対の側の空いたビットに移します。左にローテイトする **ROL(rotate left)命令**と右にローテイトする **ROR(rotate right)命令**があります。例えば、上の ax の値の場合には、

```
  ror $4,%eax
```

とすると、011X 0111 1001 0110 1111 1001 0101 0100 になります。この場合、X にはこの命令を実行する直前のＣＦの値が入り、最後にはみ出た０がＣＦフラグに入ります。このような命令は、コンピュータグラフィックスでビットのイメージをシフトしたりする場合に使われます。

論理演算命令と条件分岐

論理演算命令には、AND, OR, XOR, NOT などの命令があります。オペランドは加算命令と同じです。

```
  and src,dst # dst = dst & dst
  or   src,dst # dst = dst | dst
  xor  src,dst # dst = dst ^ dst
  not  dst  # dst = ~ dst
```

さて、あるビットが１かどうかなど調べるときに便利な命令が、test 命令です。cmp 命令は、sub 命令をして結果を残さない命令であるように、test 命令は and 命令を行って、結果を残さない命令です。

```
  test src1,src2 # src1 & src2
```

この命令は、SF や ZF フラグをセットして、この後，jz 命令などで分岐します。

スタックと push/pop 命令

esp はスタックポインタと呼ばれるレジスタです。スタックはコンピュータでもっとも基本的なデータ構造であり、esp を使ってスタック操作を行うのが push/pop 命令です。

```
  push src  # src を push する
```

すなわち、src が３２ビットの場合（src が明示的にサイズを表さない場合は pushl と明示)、以下の操作がおこなわれます。

```
  esp = esp - 4
  (esp) = src
```

スタックはアドレスの下位の方向に伸びていきます。esp で指しているアドレスがスタックの top の要

素を指していることになります。この push と逆の操作を行うのが pop 命令です。

pop dst # dst にスタックから pop する。

dst は、当然、レジスタかメモリでなくてはなりません。この push/pop 命令は主に、関数呼び出しの引数渡しやレジスタの値などの待避に使われます。それでは関数呼び出しについて説明しましょう。

関数呼び出し：call 命令と ret 命令

esp で実現されているスタックは関数呼び出しに使われます。関数呼び出しは、以下のようにして行われます。

- 次の命令のアドレスをスタックに push する。
- 関数の先頭のアドレスに jump する。

これを行うのが、call 命令です。

call label # 関数呼び出し

この call 命令の次の命令のアドレス（戻り番地）を push して、label に jump します。

この逆、つまりスタックから pop して、そのアドレスに jump するのが ret 命令です。

ret # 関数からリターン

ここで、**ret 命令が実行されるときにはスタックの top に戻りアドレスがなければならない**ことに注意してください。

関数の引数の渡し方、関数の値の返し方

関数には引数がありますが、これを渡す方法として２つの方法があります。

- レジスタに入れて渡す方法、
- スタックに積んで渡す方法

レジスタにいれて渡す場合、呼び出す側と呼び出される側で決めておく必要があります。例えば、eax に第一引数、ebx に第二引数、といったように決めておくわけです。

```
mov  第一引数、%eax
mov  第２引数、%ebx
call foo
```

呼び出された側では、そのレジスタの値を使って計算します。しかし、さらに関数呼び出しをする場合にはまたレジスタが必要になりますので、必要な値はスタック push して待避しておき、関数呼び出しから帰ってきたら、pop して使います。

関数のリターン値については、値を返すレジスタを決めておき、それを使います。(大抵の場合、整数の値は eax を使って返します。)

しかし、x86 ではレジスタの数はそんなに多くはないので、引数が多い場合はスタックに積んで渡します。例えば、２引数をもつ foo という関数を考えると、

```
push 第２引数
push 第１引数
call foo
```

のようにします。呼び出された側では、スタックには、上から戻り番地、第１引数、第２引数と詰まれていますので、第１引数をアクセスするには、4(%esp)でアクセスすることができます。

```
foo:  …
      mov  4(%esp), %eax    #第１引数を eax にロードする
```

しかし、foo の中でさらに関数呼び出しをする場合には esp の値が変わってしまうことになります。またレジスタのちなどをスタックに待避するために push しておく場合にも、esp の値が変わってしまうことになります。そのために使うレジスタが ebp（ベースポインタ）です。ebp には呼び出された時点の esp の値（スタックのトップのアドレス）を入れておいて、この値を使って引数にアクセスします。

このような関数呼び出しについては、それぞれにマシンに決まった呼び出し方があり、例えばＣ言語とリンクする場合など、それに従わなくてはなりません。それについては、後半で。

今回やったことのまとめ

シフト命令とローテイト命令、論理演算命令、push/pop, call と ret 命令