[機械語序論　４回目　２０１１．１．１２]

２進数による数の表現（その２）

前回、コンピュータの中での２進数による数の表現について説明しました。負の数は２の補数表現という方法で表現されています。もう一度繰り返すと、ｎビットで表現する場合、正の数をｘとすると、2^n-x で負の数を表現するものです。例えば、8 ビットつまり n=8 とすると、25(00011001)の負の数－25 は、2^8=256 ですから、256-25=231(11100111)となります。これは、00011001 のビット反転 11100110 に 1 を足したものと同じになります。この２の補数表現の利点を以下にあげておきます。

1．負の数と正の数を同じように扱える。負の数と正の数を区別することなしに加算すればよい。
2．最上位のビットを見るだけで正の数か負の数かわかる。0 ならば正、1 ならば負の数である。
3．正の数から２進数で-1 を繰り返していけば、自然と負の数になる。

２の補数表現では、８ビットでは、-128 から 127 まで、16 ビットでは、-32768 から 32767 まで、３２ビットでは、-27147483648 から 2147483647 までの範囲の数を表すことができます。どの表現でも、全部１の場合は-1 で、最上位ビットのみが 1 の数字は負の数の最小値（絶対値が最大の負の数）で、最上位ビット以外が１の場合は正の数の最大値であることを覚えておきましょう。

add，sub 命令（その２）ADC，SBB，INC，DEC，NEG

さて、２の補数表現はきまりごとですから、数は符号なしでも扱うことができます。これを符号なし数といいます。例えば、Ｃ言語で unsigned をつけたものがそれです。加算命令 ADD は符号つき数（２の補数表現）でも符号なし数でも同じです。それは、上の性質 1 によるものです。前回、条件フラグに、ZF(ゼロフラグ)、SF(サインフラグ)、OF(オーバーフローフラグ)、CF（キャリーフラグ）の４つがあると説明しました。このうち、SF と OF はオペランドを符号つきの数としてみたときに意味のあるフラグです。特に、OF は「数を符号つき数としてみたときに結果が範囲を超えた」ことを示します。それに対して、CF は「符号なし数としてみたときに、最上位のビットから繰り上がりがあった」ことを示します。

ADC（Add with carry）命令は、オペランドは ADD 命令と同じですが、演算前にキャリーフラグがセットされているときに、加算結果にさらに１を足す命令です。これは、32 ビット以上の数を加算するときに便利な命令です。例えば、64 ビットの整数を足したりするときにはこの命令が必要になります。

SUB 命令にも同じことが言えます。SUB 命令も命令自体は符号つき数でも、符号なし数でも同じです。今度は、ボロー(繰り下がり)が必要になるときに CF フラグがセットされます。これに対して、**SBB(subtract with borrow)命令**は、sub 命令と同じですが、CF フラグがセットされている場合には結果からさらに-1 されます。

加算、減算に関してはあと、INC 命令と DEC 命令があります。

```
inc dst      # dst = dst+1
dec dst      # dst = dst-1
```

INC(increment)命令は、１オペランドの命令です、dst に 1 加えます。また、DEC 命令は１引きます。この命令では、CF フラグは影響されません。

```
neg dst      # dst = -dst
```

NEG 命令は、0 からオペランドを減算した結果をセットします。つまり、dst を負の数に変換します。

mul,div 演算命令

乗算と除算命令では、符号なしと符号ありと時には別々の命令をつかわなくてはなりません。その意味は、符号なしとして解釈したときと、符号ありと解釈して乗算した結果が 2 の補数表現上異なってしまうからです。符合つき乗算命令は、IMUL 命令です。IMUL 命令には３つの使い方があります。

```
imul src,dst    # dst = dst * src
```

この２オペランド形式では、これまでの add や sub 命令と同じです。

```
imul src1,src2,dst   # dst = src1 * src2
```

この３オペランド形式では、dst はレジスタでなくてはなりません。

実は、乗算の場合にはその結果を正しく格納するためには倍のビットが必要になります。つまり、３２ビットと３２ビットの掛け算の結果は６４ビットのはずです。結果として６４ビットを得るためには次の１オペランド形式を使います。

```
imul src  # edx:eax = eax * src
```

この１オペランド形式の場合には、暗黙のうちに片方のオペランドに eax を使って,src を掛け算して、その結果の６４ビットを上位３２ビットを edx に、下位３２ビットを eax に格納します。

符号なし演算命令は mul です。これは、imul の１オペランド形式と同じものしかありません。

mul src # edx:eax = eax * src

実は、３２ビット同士の乗算で、３２ビットのみの結果を得る場合には、符号付でも符号なしでも同じなので、この場合には符号なしの乗算に imul を使うことができます。

除算命令も同じように、符号つきのものと符号なしのものがあります。

idiv src # eax = (edx: eax) / src , edx = 余り

この命令では、被除数の上位３２ビットを edx、下位３２ビットを eax において、これを src で割った商を eax に、余りを edx に格納します。符号なしの命令は div 命令で、同じオペランドを持ちます。

なお、32 ビットの符号つき数を 64 ビットに符号拡張するには、cltd(cdq)命令を使うことができます。

```
mov  src1,%eax # eax に被除数をいれる
cltd           # 符号を拡張して、eax -> edx:eax
idiv src2      # src2 で割る。  商は eax 余りは edx に入る
```

オペランドサイズについて

これまで、オペランドは３２ビット（ロング、もしくはダブルワード）のみを説明してきました。しかし、x86 アーキテクチャで扱う数は１６ビット（ワード）、８ビット（バイト）の場合があります。汎用のレジスタである eax,ebx,ecx,edx は、その中に下位１６ビットをあらわす名前 ax,bx,cx,dx があります。さらに、そのうち上位８ビットには ah,bh,ch,dh, 下位８ビットには al,bl,cl,dl という名前がつけられています。また、edi、esi も下位１６ビット部分には di,si という名前がつけられています。x86 のアセンブラでは、レジスタオペランドにこれらの１６ビット、あるいは８ビットのレジスタの名前が指定されたときには、そのサイズの命令になります。例えば、

```
mov  src1,%ax  #  src1 の１６ビットを ax にロードする。
add  src1,%al  # src１の８ビットを al に加算する。
```

ここで、例えば、src が即値で、dst がメモリの場合にはどのオペランドサイズであるかわかりません。そのときには、命令の最後に l(３２ビット、ロングワード) w(１６ビット、ワード)、b(８ビット、バイト)をつけて、明示して指定します。

```
movb  $1,(%edi)
```

では、%edi のさす番地に１バイトで数値 1 を格納することになります。また、imul や idiv 命令で、３２ビット命令 imull, idivl では eax と edx をつかいますが、１６ビット命令 imulw, idivw では、ax と dx、８ビット命令 imulb, idivb では、al と ah を使います。

サインビットの符号拡張とシフト命令・論理演算命令

メモリ上の値をサイズの異なる 2 進数としてロードしたい場合があります。この場合、符号つきか符号なしかによって２つの命令 **movs と movz 命令**があります。movs 命令の場合は、最上位のビットで残りの上位のビットを埋めます。これを**符号拡張（sign extension）**といいます。movz の場合は無条件に 0 を埋めるだけです。これらの命令では、src のオペランドサイズと dst のオペランドサイズの２つを書き加えます。

```
movsbl  (%di),%eax  # 符号つきのバイトを 32 ビットの eax に符号拡張してロードする。
movzwl  (%si),%edx  # 符号なしのワードを edx にロードする。上位 16 ビットは 0。
```

シフト命令は、1bit 単位のシフトを行います。左にシフトする命令は、SHL(shift logical left)命令です。

shl src,dst # dst = dst << src

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。**

右にシフトする場合には、サインビットを拡張するかどうかで、**SHR(Shift logical right)と SAR(shift arithmetic right)**の２つの命令があります。

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。

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。

これらに関連し、ビットを操作する論理演算命令として、**AND, OR, XOR, NOT** があります。それぞれビット毎の論理演算を行います。最初の 3 つは 2 つのオペランド、NOT は１つのオペランドを取ります。

今回やったことのまとめ

キャリーフラグの意味と加算減算命令、乗算除算命令、オペランドサイズと符号拡張、論理演算

_