

## [機械語序論 4回目]

### 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、-2147483648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が、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。符号つき乗算命令は、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 ビットのみ得る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、符号付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 ビットの符号付き数を 64 ビットに符号拡張するには、`cldq(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
cldq          # 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、符号付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 # 符号付きのバイトを 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