第４回
演算命令（その２）

前回のまとめ

- 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

数の表現：２の補数表現

- 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。

$$y - x = y + (-x) = y + (2^n - x) = 2^n + (y - x)$$

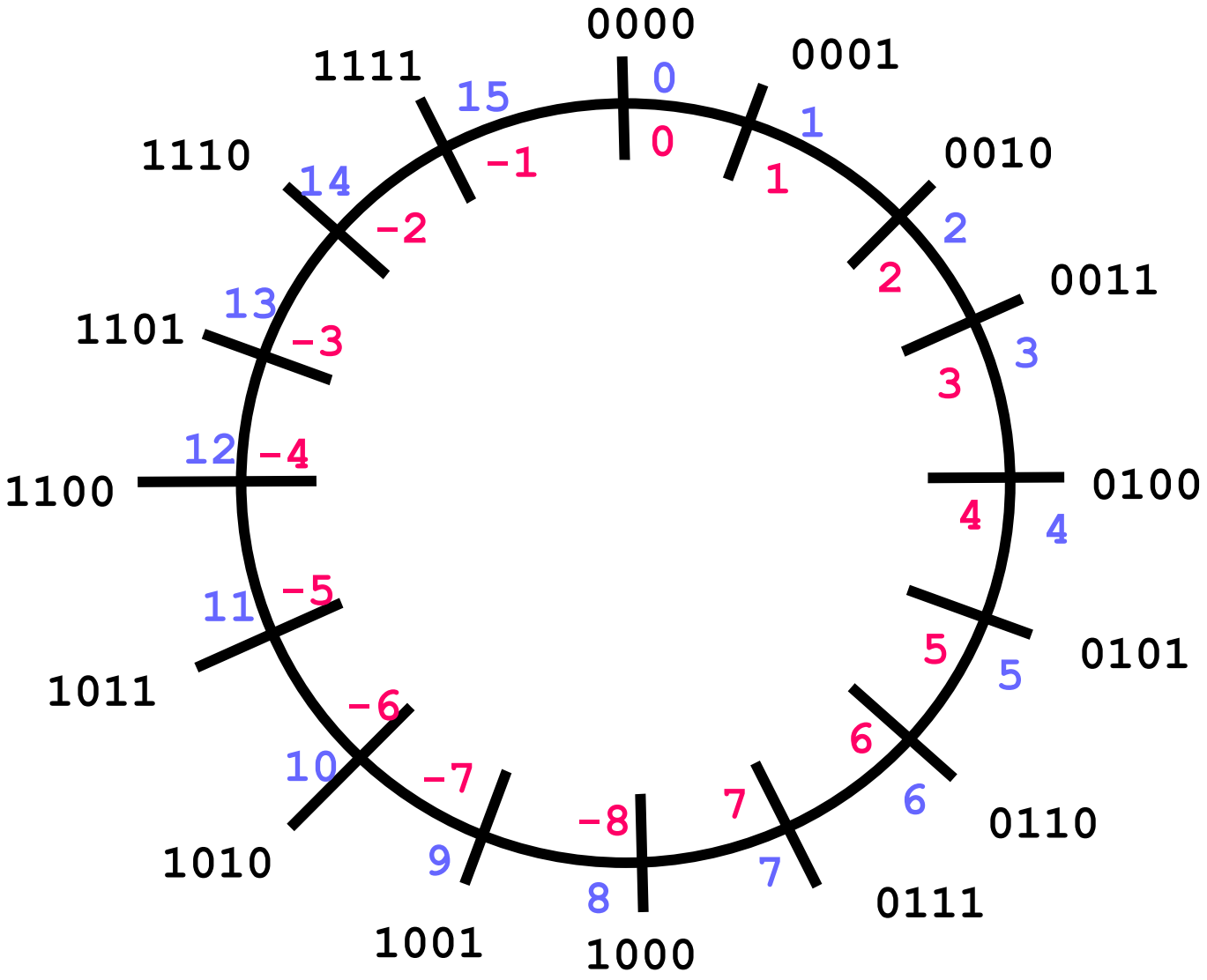
数の表現：２の補数表現

- 負の数の足し算

$$y - x = y + (-x) = y + (2^n - x) = 2^n + (y - x)$$

- 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。

数の表現：２の補数表現


0000
0
0
0001
1
1
0010
2
2
0011
3
3
0100
4
4
0101
5
5
0110
6
6
0111
7
7
1000
8
-8
1001
9
-7
1010
10
-6
1011
11
-5
1100
12
-4
1101
13
-3
1110
14
-2
1111
15
-1

数の表現：２の補数表現

- 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を加えて作ることができます。

$$x \rightarrow 2^n-1-x \text{（反転、足してすべて１）}$$
$$\rightarrow 2^n-1-x + 1 \text{（１を加える）}$$
$$= 2^n-x$$

数の表現：２の補数表現

- ２の補数表現では、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。

条件分岐命令と条件フラグ

- 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。

条件分岐命令と条件フラグ

- **符号つきの場合**

命令ニーモニック	条件（フラグの状態）	説明
符号付き条件付きジャンプ		
JG/JNLE	((SF xor OF) or ZF) =0	より大きい / より小さくなく等しくない
JGE/JNL	(SF xor OF)=0	より大きいか等しい / より小さくない
JL/JNGE	(SF xor OF)=1	より小さい / より大きくなく等しくない
JLE/JNG	((SF xor OF) or ZF)=1	より小さいか等しい / より大きくない
JNO	OF=0	オーバーフローなし
JNS	SF=0	符号なし（負でない）
JO	OF=1	オーバーフロー
JS	SF=1	符号（負）

条件分岐命令と条件フラグ

- 符号なしの場合

命令ニーモニック	条件（フラグの状態）	説明
符号なし条件付きジャンプ		
JA/JNBE	(CF or ZF)=0	より大きい / より小さくなく等しくない
JAE/JNB	CF=0	より大きいか等しい / より小さくない
JB/JNAE	CF=1	より小さい / より大きくなく等しくない
JBE/JNA	(CF or ZF)=1	より小さいか等しい / より大きくない
JC	CF=1	キャリー
JE/JZ	ZF=1	等しい / ゼロ
JNC	CF=0	キャリーなし
JNE/JNZ	ZF=0	等しくない / ゼロでない
JNP/JPO	PF=0	パリティなし / 奇数パリティ
JP/JPE	PF=1	パリティ / 偶数パリティ
JCXZ	CX=0	レジスタ CX がゼロ
JECXZ	ECX=0	レジスタ ECX がゼロ

もう一回、２の補数表現の例

- 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

- 8ビットつまりn =8とすると、
- 25(00011001)の負の数－25は、
- 2^8=256
- 256-25=231(11100111)

- 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。

２の補数表現の利点

- 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。

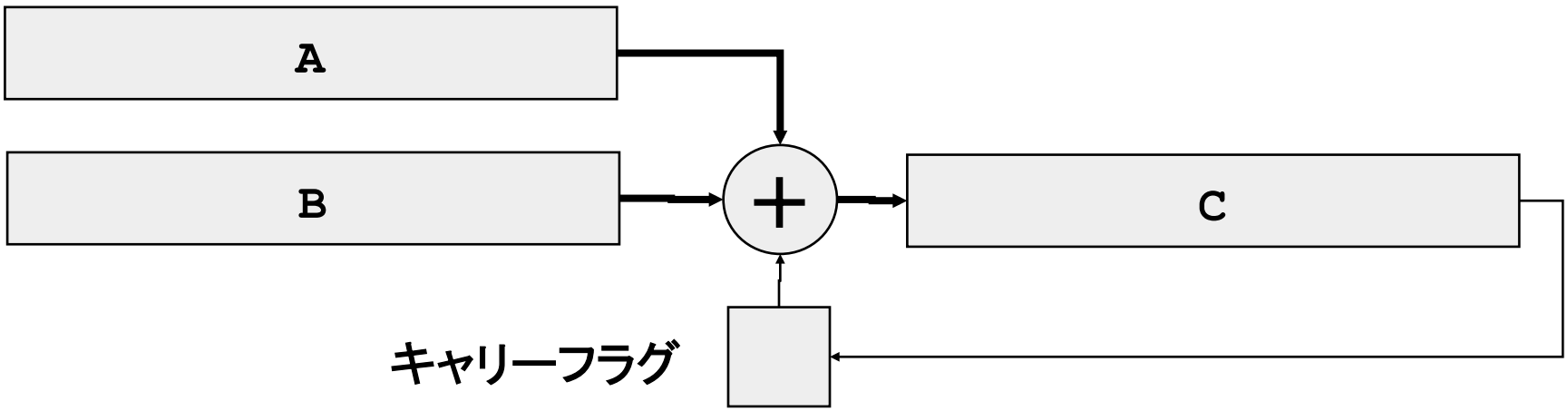
- どの表現でも、全部１の場合は-1で、最上位ビットのみが1の数字は負の数の最小値（絶対値が最大の負の数）で、最上位ビット以外が１の場合は正の数の最大値であることを覚えておきましょう。

add, sub命令（その２）

- 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
- 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
- 条件フラグ：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
- 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
  - 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。

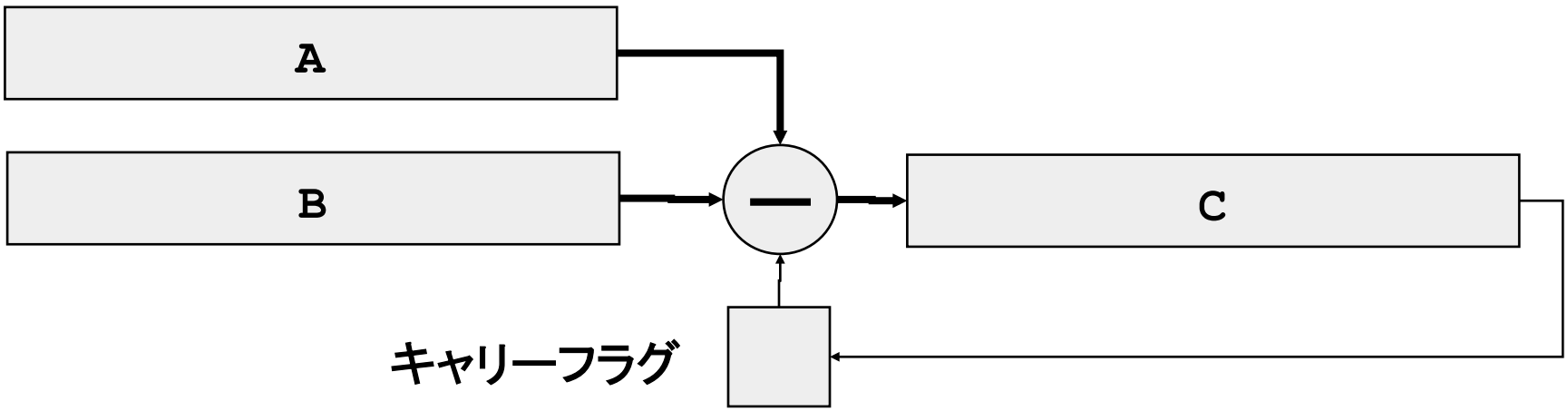
ADC（Add with carry）命令

- 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
  - 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。



SBB(subtract with borrow)命令

- 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。
  - 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。



INC命令とDEC命令

- INC(increment)命令は、１オペランドの命令で、dstに1加えます。
- DEC(decrement)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。

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

NEG命令

- 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。
- 符号付き二進数の最上位ビット（sign bit）だけを反転する命令ではないので注意。

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

imul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。

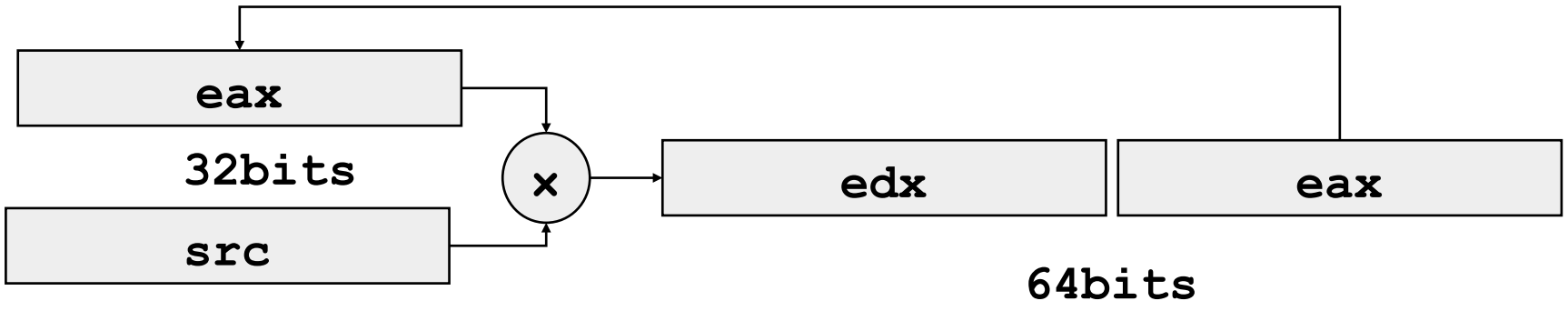
```
imul src,dst            # dst = dst * src
imul src1,src2,dst      # dst = src1 * src2
```

  - 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。

imul 演算命令

- 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一般に64ビットになります。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。使用されるレジスタeax, edxは暗黙に指定され、他のものに変更はできません。

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



mul演算命令

- 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。

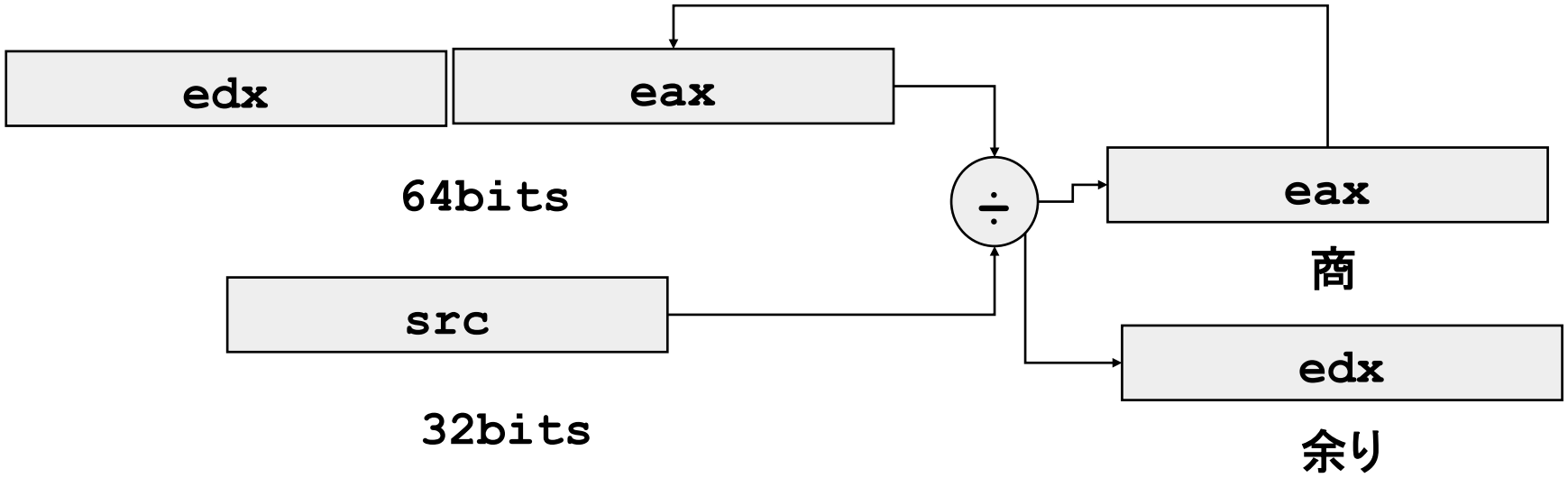
```
mul src   # edx:eax = eax * src
```

- 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。

div命令

- 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。
- 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。
- 商は32ビットのみ得られます。
- 符号なしの命令はdiv命令で、同じオペランドを持ちます。

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



div命令

- 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

- 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。
- 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。

オペランドサイズについて

- 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。

```
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。
```

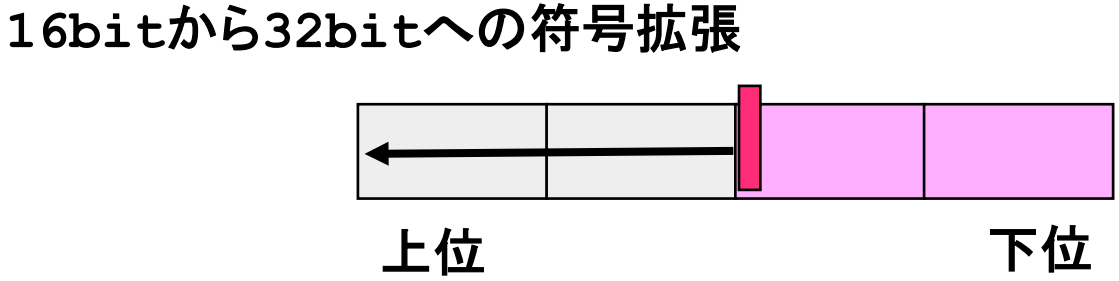
オペランドサイズについて

- 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)
```

符号拡張とは

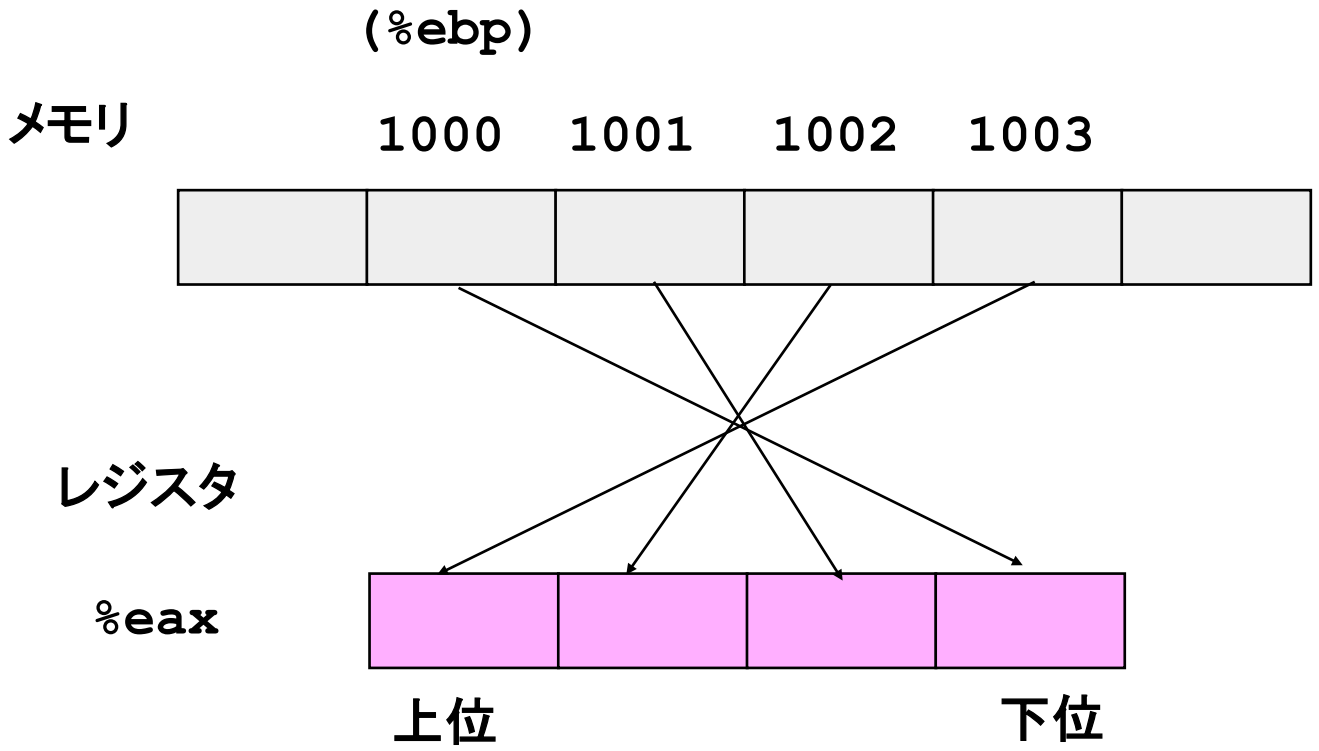
- 語長が違う符号つき数を扱う場合、符号拡張する必要がある。
- 最上位のサインビットで、上位のワードを埋めることを符号拡張(sign extension) という。



16bitの “-3” = 0xfffd → 32bitでは0xfffffffd

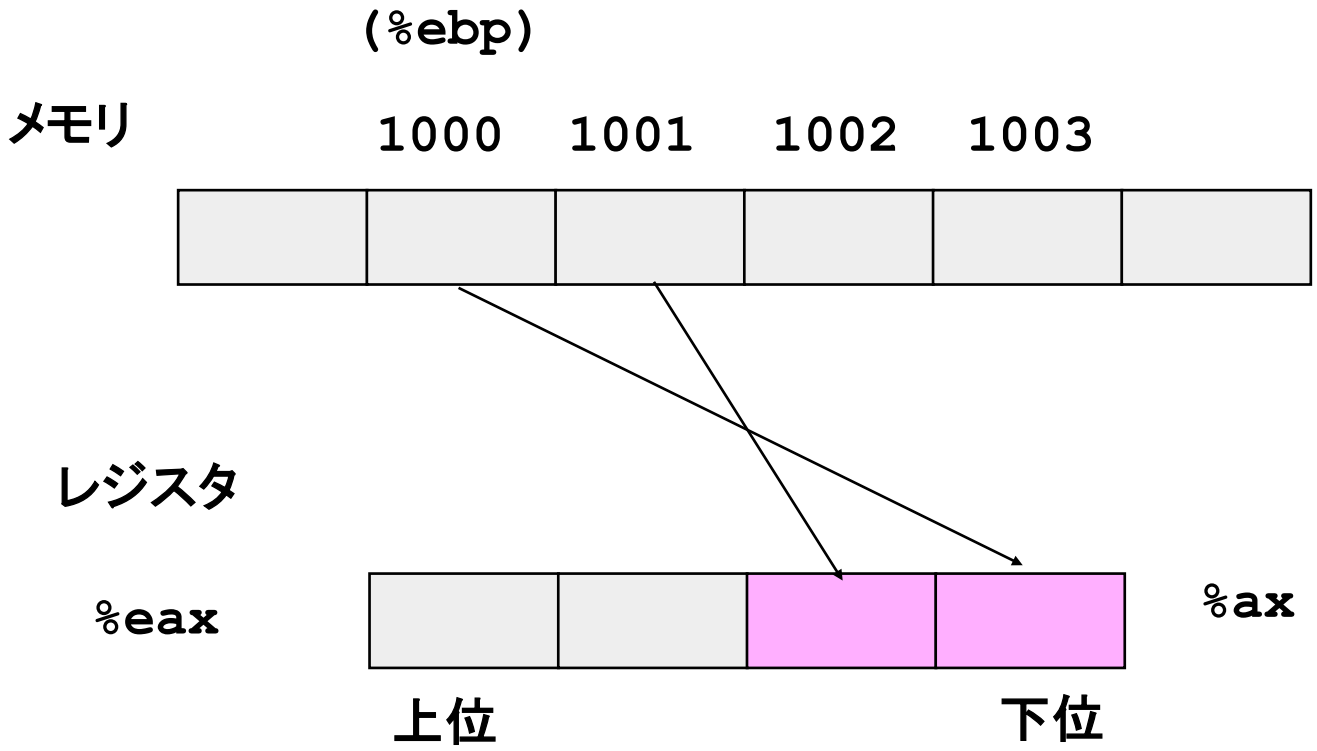
movl

movl (%ebp), %eax　　# 1000-1003番地の4byteの値をeaxにコピー



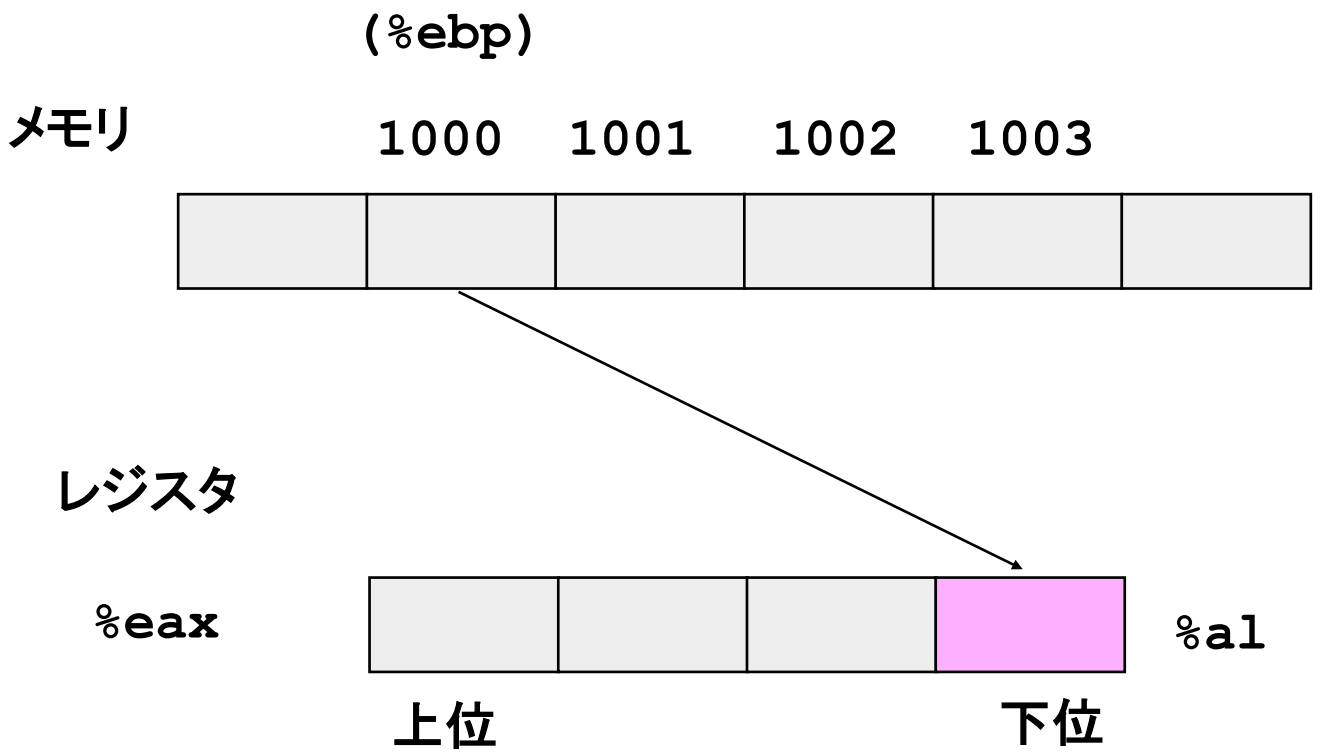
movw

movw (%ebp), %ax　　# 1000-1001番地の2byteの値をaxにコピー



movb

movb (%ebp), %al　　# 1000番地の1byteの値をalにコピー



語長と符号拡張

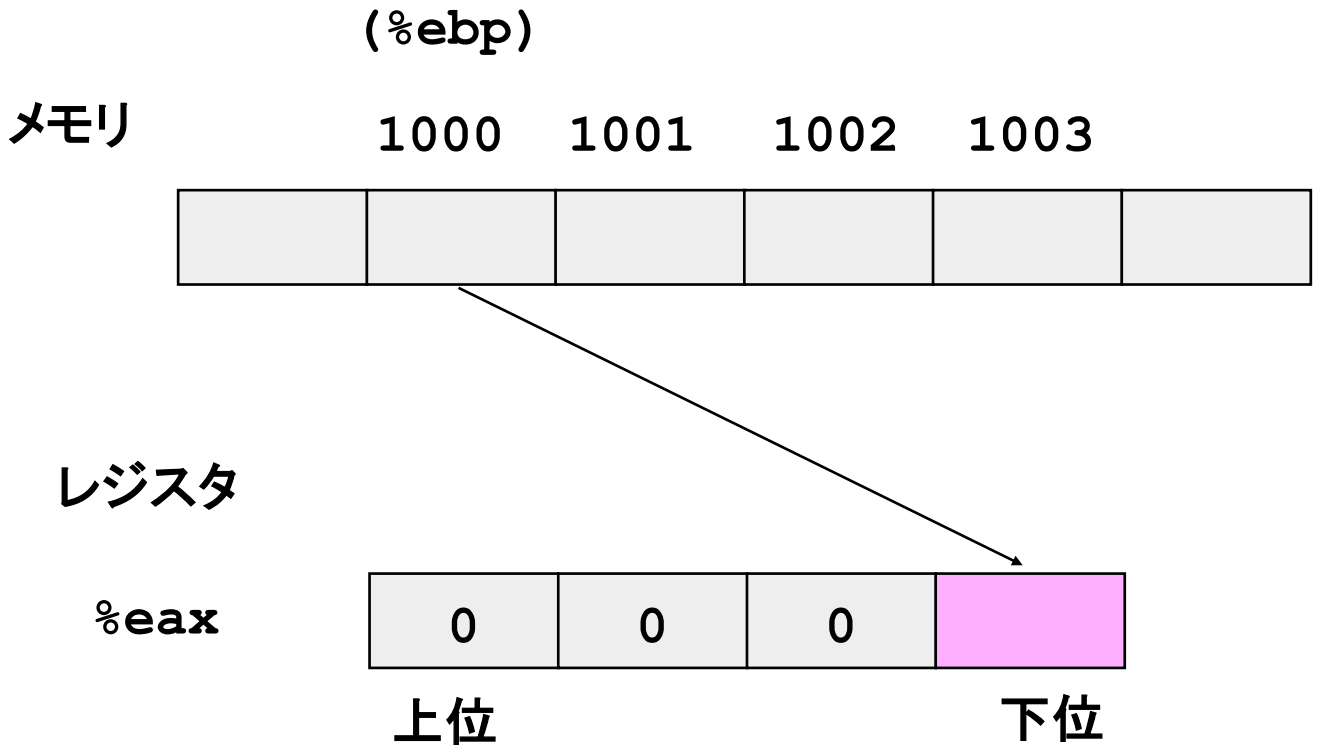
- movb, movwは、ロードの場合は、ax, al以外の値を変えない
- ストアの場合は、この命令を用いる

- ロードして、符号拡張、または０で埋める場合はmovs, movz命令を用いる
  - バイトの場合は、movsbl, movzbl
  - ワードの場合は、movswl, movzwl

movzbl

movzbl (%ebp), %eax　　# 1000番地の1byteの値をeaxにコピー, eaxの上位を０にする

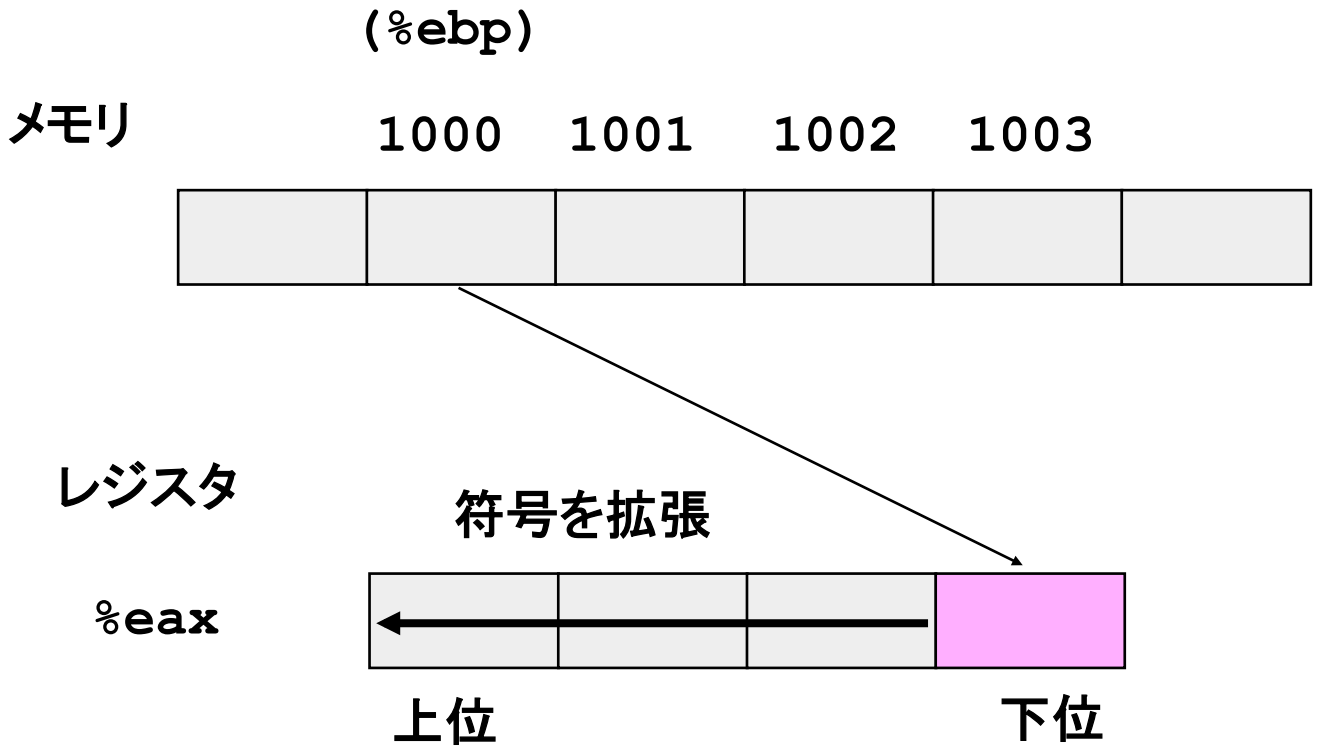
unsigned の場合に使う



movsbl

movsbl (%ebp), %eax　# 1000番地の1byteの値をeaxにコピー , 符号を拡張

signed の場合に用いる



movzblとmovsblの実行例

プログラム

```
          .data
          .align 4
a:        .long 0x0E34837F
b:        .long 0x0E348380
          .text
          .globl main
main:     movzbl a, %eax
          movsbl a, %ebx
          movzbl b, %ecx
          movsbl b, %edx
          call stop
```

a, b の番地の1byteの2進表現

0x7F = 0b01111111

0x80 = 0b10000000

結果

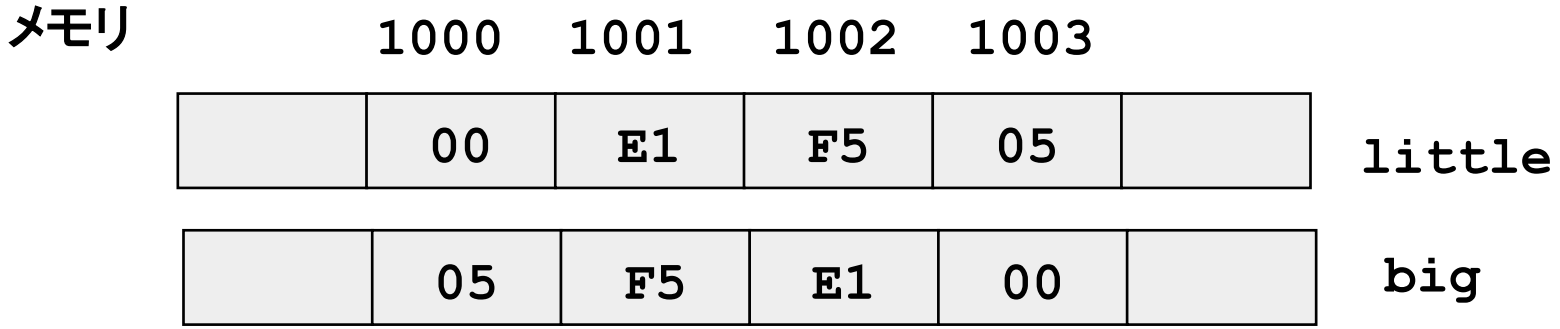
```
eax=0x0000007f   ebx=0x0000007f

ecx=0x00000080   edx=0xffffff80
```

edx = 0xffffffff80 = 0b1111111111111111111111110000000

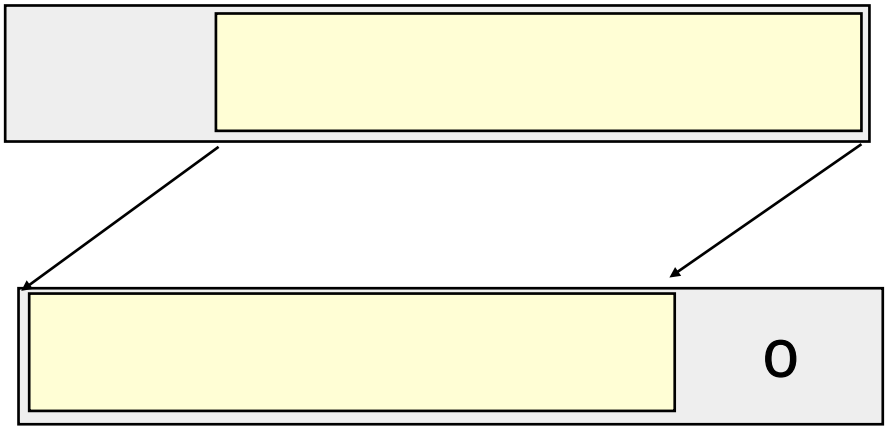
Big EndianとLittle Endian

- 1バイト以上のデータを扱う際、メモリ上でそれがどの順番で格納されているかに気をつける必要があります。
- データをビットの列として見た場合、低いアドレスから高いアドレスに向けて、上位のビットを含むデータから格納する方法をbig endian、下位のビットを含むデータから格納する方法をlittle endianと呼びます
  - （”endian”を”indian”と間違えている人がよくいるので注意）。これはCPUアーキテクチャに依存し、x86やAlphaはlittle endian、M68000やPowerはbig endian、MIPSはそのどちらかを実行時に選べるようになっています。
  - 100000000 = 0x05F5E100



シフト命令

- 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。
- srcは、即値もしくは、clレジスタでなくてはなりません。

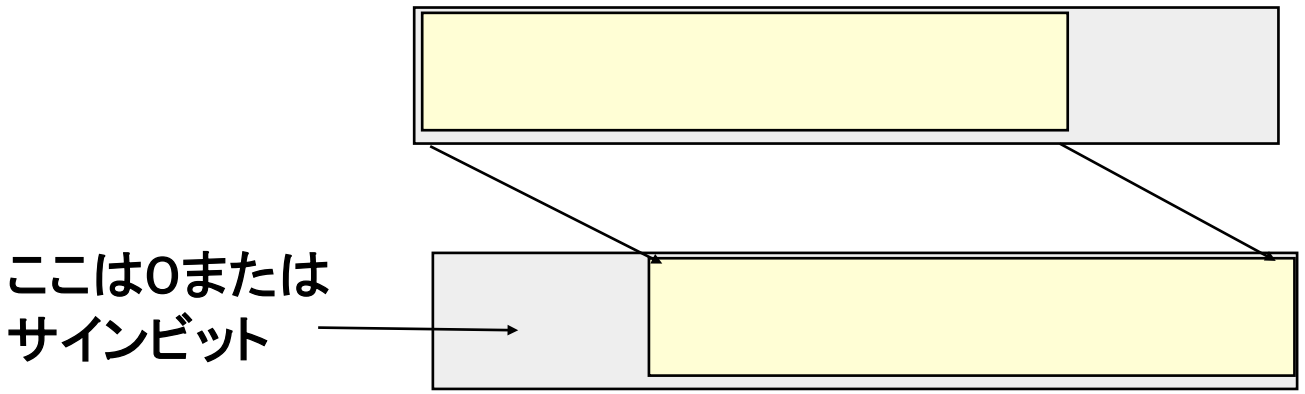


シフト命令

- 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。



論理演算

- 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。

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

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

課題３

１、データ領域に３２ビットの２つの変数xとｙを宣言し、
これを乗算命令を使わずに、シフト命令と加算命令で、
xとyの符号無し乗算結果をeaxに格納して終了するプログラム
を作りなさい。結果は３２ビットに収まるものとしてよい。
xとyに適当な初期値をセットし、その結果を提出すること。

ヒント：　２進法の乗算のやり方を考えること。
使う命令は以下のとおり、

```
shl   $1,src   # srcを１ビット左にシフト
shr   $1,src   # srcを右にシフト
and   src,dst   # dst = dst & src
```

課題３

2. １２８ビットの符号なしの整数の加算プログラムを作りなさい。データ領域に、それぞれ４つの３２ビットのデータを持つ、配列xとｙを以下のように宣言する。

```
        .data
        .align 4
x:      .long 0x87001240,0x00124011,0x8130FFFF,0x1234
y:      .long 0x07001245,0x12f01348,0x8230FFFF,0x12
        .text
```

ここからメインプログラム….

- これをアドレス下位にあるものを下位の桁として、128ビットの符号なし整数を表しているものとする。
- 例えば、xは、0x12348130FFFF0012401187001240という数を表している。
- この２つの数を加算して、加算結果を下位の桁から順にeax,ebx,ecx,edxの４つの３２ビットデータとして格納して終わること。

ヒント：　キャリーを使った加算を使うこと。