機械語序論

第５回
演算命令（その３）
関数呼び出し（その１）

連絡

- 次回は中間試験をやります.
  - 日時：2月1日　2時限目（10:10 ～ 11:25）
  - 場所：3A402
  - 持ち込み：配付資料，及び手書きの資料のみ
    PCなどの電子機器は使用不可

前回のまとめ

- 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

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。

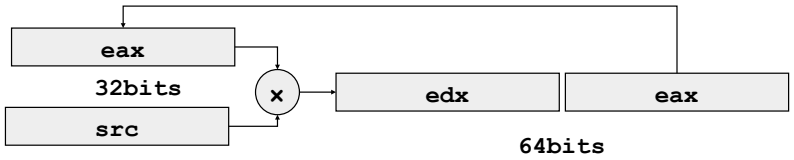
```
imul src,dst          # dst = dst * src
imul src1,src2,dst    # dst = src1 * src2
```

  - 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
```



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。

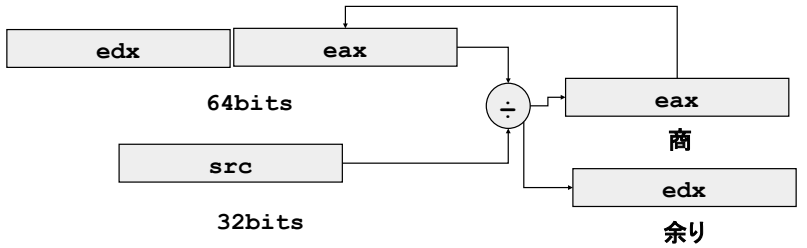
mul命令に対する注意

- 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。
  - 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、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。

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。
- 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
```



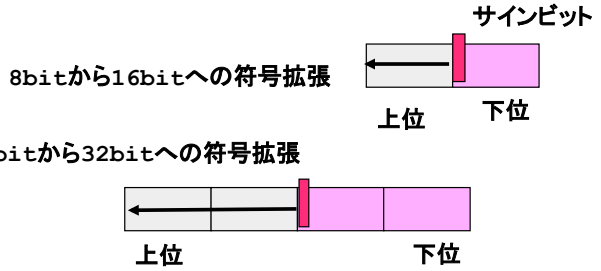
div命令

- 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
```


符号拡張とは

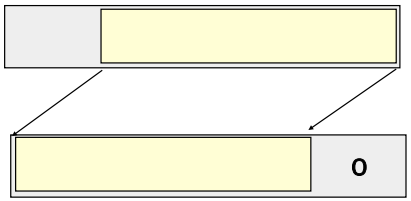
- 語長が違う符号つき数を扱う場合、符号拡張する必要がある。
- 最上位のサインビットで、上位のワードを埋めることを符号拡張(sign extension) という。



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

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。

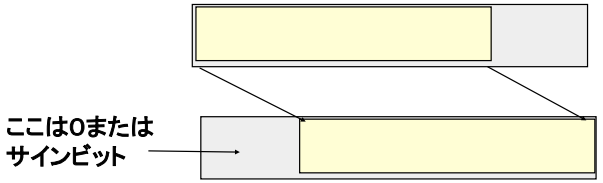


シフト命令

- 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。



シフト命令についての追加事項

- SAR命令は、シフトするときに符号拡張していることになる。
- 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、eax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。

ローテイト命令

- はみ出たビットを反対の側の空いたビットに移します。
- ローテイト命令には,

ROL (rotate left) ：CF を含めずに回転
ROR (rotate right)：CF を含めずに回転
RCL (rotate left through carry bit)：CF を含めて回転
RCR (rotate right through carry bit)：CF を含めて回転

があります。

【注意】配付資料と異なりますので，メモしておいてください.

CFを含めないローテイト命令

- ROR (rotate right) 命令の例：

%eax	CF
0111 1001 0110 1111 1001 0101 0100 0011	X

  - `ror $1, %eax`

%eax	CF
1011 1100 1011 0111 1100 1010 1010 0001	1

  - `ror $2, %eax`

%eax	CF
1101 1110 0101 1011 1110 0101 0101 0000	1

  - `ror $4, %eax`

%eax	CF
0011 0111 1001 0110 1111 1001 0101 0100	0

CF には，シフトされて最後にはみ出したビットが入ります。

CFを含めたローテイト命令

- RCR (rotate right through carry bit) 命令の例：

%eax	CF
0111 1001 0110 1111 1001 0101 0100 0011	X

  - `rcr $1, %eax`

%eax	CF
X011 1100 1011 0111 1100 1010 1010 0001	1

  - `rcr $2, %eax`

%eax	CF
1X01 1110 0101 1011 1110 0101 0101 0000	1

  - `rcr $4, %eax`

%eax	CF
011X 0111 1001 0110 1111 1001 0101 0100	0

シフトを行う際に，CF も含めて実行する。

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
```


test命令

- 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。

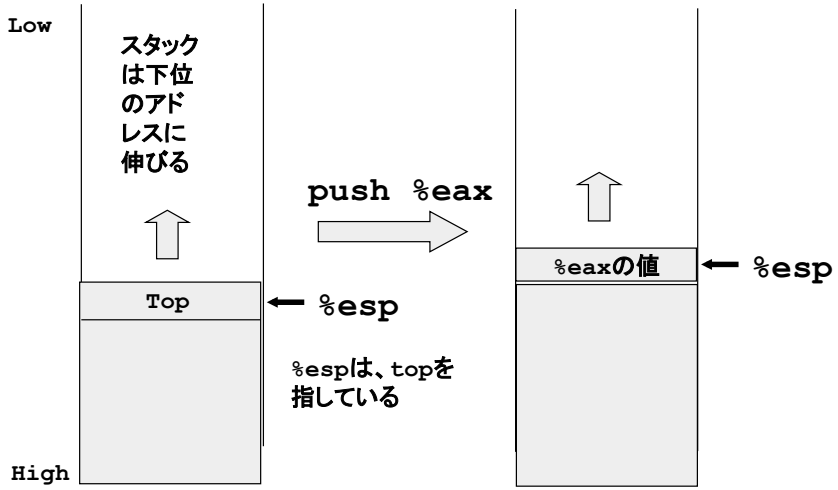
```
push src  # srcをpushする
```

- 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。

スタックとpush/pop命令



スタックとpush/pop命令

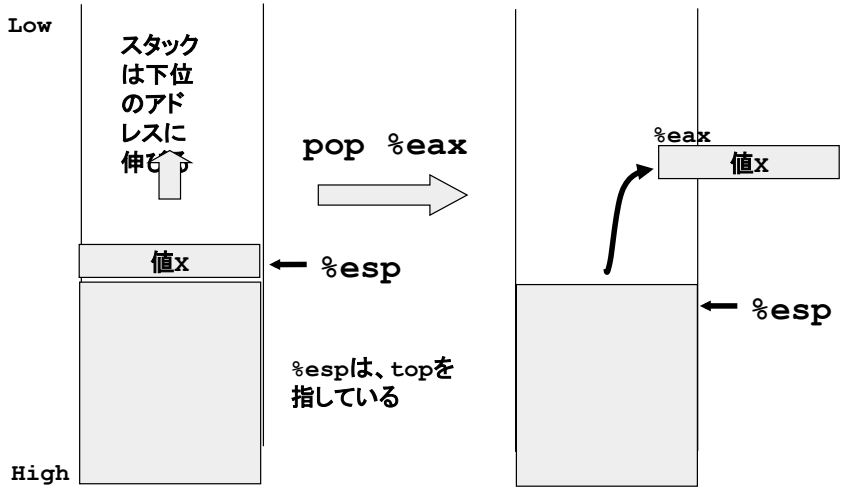
- pushと逆の操作を行うのがpop命令です。

```
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。

スタックとpush/pop命令



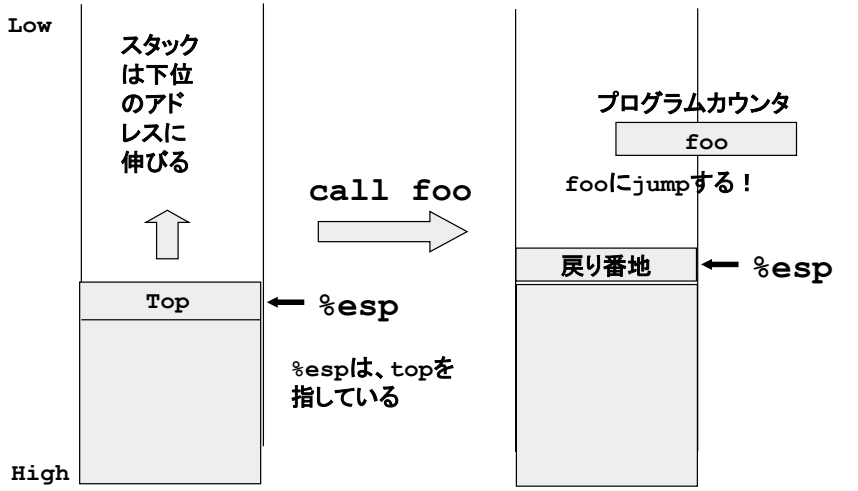
関数呼び出し：call命令とret命令

- 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。

関数呼び出し：call命令とret命令



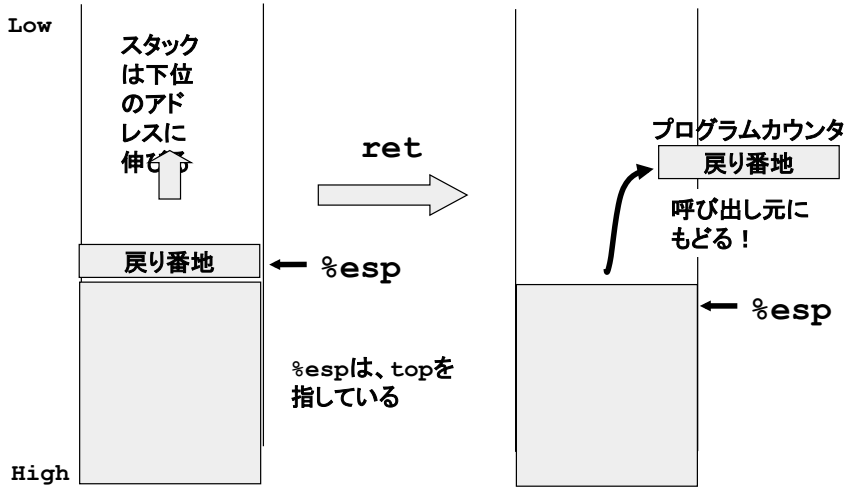
関数呼び出し：call命令とret命令

- call命令の逆、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。

関数呼び出し：call命令とret命令



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１引数、ebxに第２引数、といったように決めておくわけです。

```
mov   第１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。

関数の引数の渡し方、関数の値の返し方

- 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。）

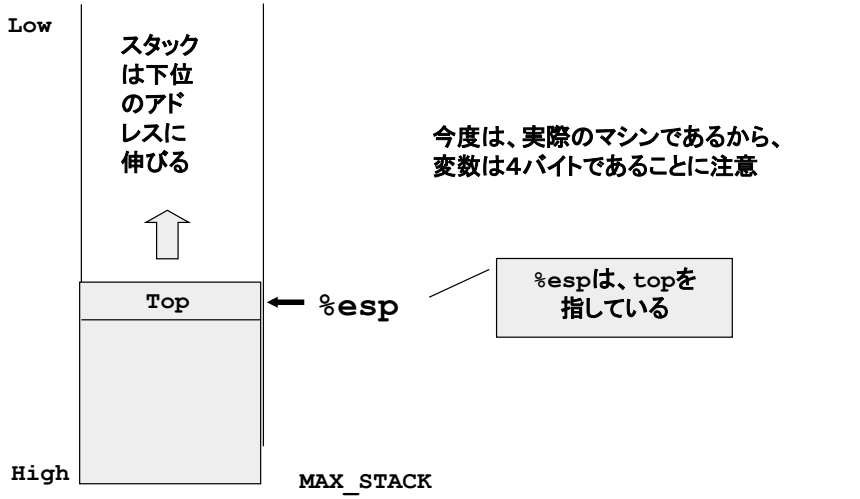
関数の引数の渡し方、関数の値の返し方

- 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、

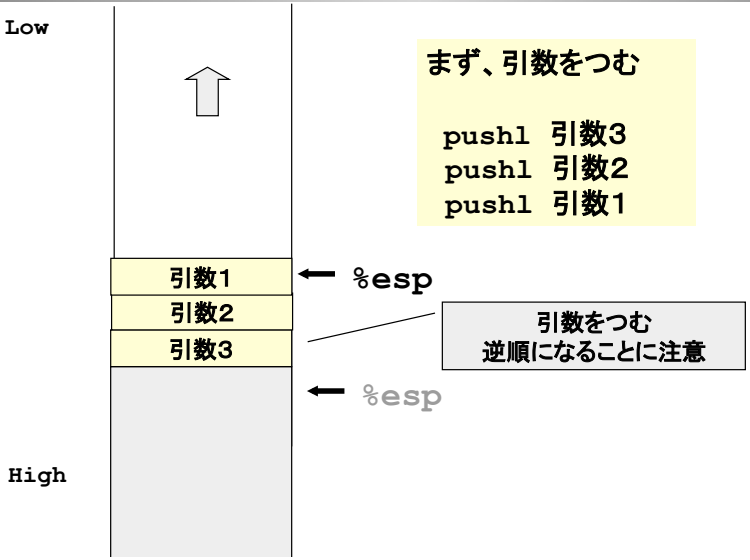
```
push 第２引数
push 第１引数
call foo
```

  - 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。

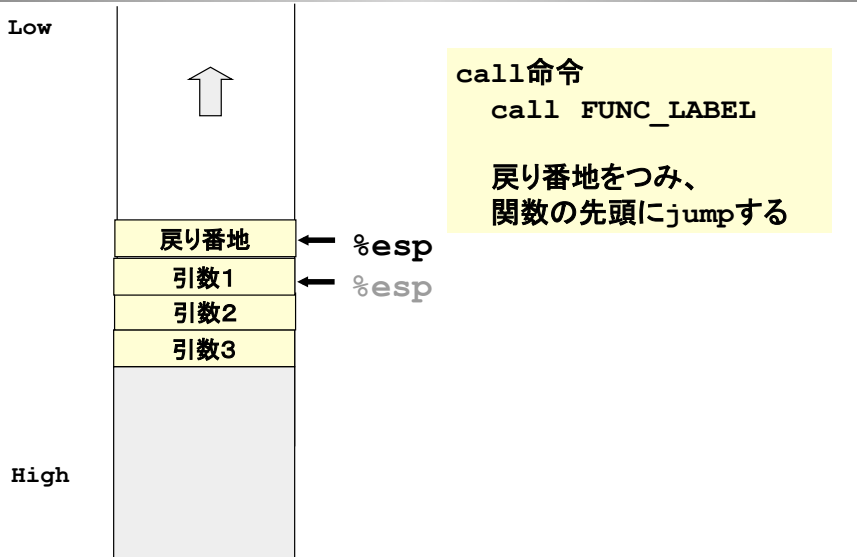
関数呼び出しの構造



関数呼び出しの構造



関数呼び出しの構造



関数呼び出しとベースポインタ（ebp)

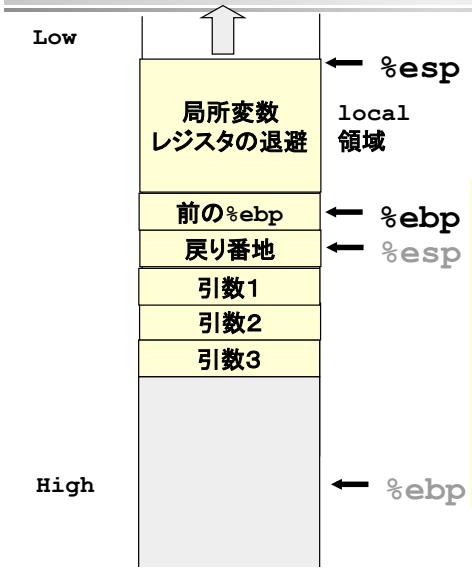
- 呼び出された側では、スタックには、上から戻り番地、第１引数、第２引数と詰まれていますので、第１引数をアクセスするには、4(%esp)でアクセスすることができます。

```
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。またレジスタの値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。

関数呼び出しの構造



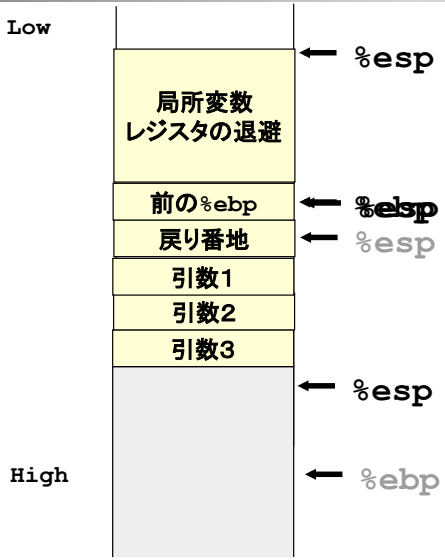
%ebp フレームポインタ
現在の関数のフレームの
位置を覚えているレジスタ

関数フレームのセットアップ

```
push %ebp
movl %esp,%ebp
subl local領域のサイズ、%esp
```

前の%ebpを保存し局所変数の分だけ%espを移動させる
⇒ 局所領域の確保

関数戻りの手順



%ebp フレームポインタ
現在の関数のフレームの
位置を覚えているレジスタ

関数から帰る場合には、戻り値を%eaxにセットする。

%ebpのところに%espを戻して、まず、前の%ebpを戻す（leave命令）

戻り番地を取り出して、そこにjump （RET命令）

戻ったら、引数を積んだ部分を%espを戻す

関数の呼び出し規則

- 実際のマシンでは呼び出し規則は決められており、命令を組み合わせて行わなくてはならない
- 呼び出し側では、スタック上に引数をpushし、call命令を用いる。
  - ラベルfooにjumpした時には、スタック上に戻り番地がpushされる。
- 関 数呼び出しが終わって、戻ってきたときには、スタックポインタを元に戻して おかなくてはならない。
  - 従って、pushした引数個数分だけ、%espを加算して戻 す。なお、関数のもどり値は、eaxに入れることになっている。

```
pushl 引数2
pushl 引数1
call foo
addl 引数の個数*4, %esp
```

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

- 関数呼び出し
  - 詳しい内容は、後半で！

連絡（再掲）

- 次回は中間試験をやります.
  - 日時：2月1日　2時限目（10:10 ～ 11:25）
  - 場所：3A402
  - 持ち込み：配付資料，及び手書きの資料のみ
    PCなどの電子機器は使用不可