Perl scripts for assembler sources

The perl scripts in this directory are my 'hack' to generate multiple different assembler formats via the one original script.

The way to use this library is to start with adding the path to this directory and then include it.

```
push(@INC,"perlasm","../../perlasm");
require "x86asm.pl";
```

The first thing we do is setup the file and type of assembler

```
&asm_init($ARGV[0]);
```

The first argument is the 'type'. Currently `cpp`, `sol`, `a.out`, `elf` or `win32`. The second argument is the file name.

The reciprocal function is `&asm_finish()` which should be called at the end.

There are two main 'packages'. `x86ms.pl`, which is the Microsoft assembler, and `x86unix.pl` which is the unix (gas) version.

Functions of interest are:

```
&external_label("des_SPtrans");  declare and external variable
&LB(reg);                        Low byte for a register
&HB(reg);                        High byte for a register
&BP(off,base,index,scale)        Byte pointer addressing
&DWP(off,base,index,scale)       Word pointer addressing
&stack_push(num)                 Basically a 'sub esp, num*4' with extra
&stack_pop(num)                  inverse of stack_push
&function_begin(name,extra)      Start a function with pushing of
                                 edi, esi, ebx and ebp. extra is extra win32
                                 external info that may be required.
&function_begin_B(name,extra)    Same as normal function_begin but no
                                 pushing.
&function_end(name)              Call at end of function.
&function_end_A(name)            Standard pop and ret, for use inside
                                 functions.
&function_end_B(name)            Call at end but with pop or ret.
&swtmp(num)                      Address on stack temp word.
&wparam(num)                     Parameter number num, that was push in
                                 C convention.  This all works over pushes
                                 and pops.
&comment("hello there")          Put in a comment.
&label("loop")                   Refer to a label, normally a jmp target.
&set_label("loop")               Set a label at this point.
&data_word(word)                 Put in a word of data.
```

So how does this all hold together? Given

```
int calc(int len, int *data)
{
    int i,j=0;

    for (i=0; i<len; i++)
    {
        j+=other(data[i]);
    }
}
```

So a very simple version of this function could be coded as

```
push(@INC,"perlasm","../../perlasm");
require "x86asm.pl";

&asm_init($ARGV[0]);

&external_label("other");

$tmp1=   "eax";
$j=      "edi";
$data=   "esi";
$i=      "ebp";

&comment("a simple function");
&function_begin("calc");
&mov(    $data,     &wparam(1)); # data
&xor(    $j,        $j);
&xor(    $i,        $i);

&set_label("loop");
&cmp(    $i,        &wparam(0));
&jge(    &label("end"));

&mov(    $tmp1,     &DWP(0,$data,$i,4));
&push(   $tmp1);
&call(   "other");
&add(    $j,        "eax");
&pop(    $tmp1);
&inc(    $i);
&jmp(    &label("loop"));

&set_label("end");
&mov(    "eax",     $j);

&function_end("calc");

&asm_finish();
```

The above example is very very unoptimised but gives an idea of how things work.

There is also a cbc mode function generator in cbc.pl

```
&cbc($name,
     $encrypt_function_name,
     $decrypt_function_name,
     $true_if_byte_swap_needed,
     $parameter_number_for_iv,
     $parameter_number_for_encrypt_flag,
     $first_parameter_to_pass,
     $second_parameter_to_pass,
     $third_parameter_to_pass);
```

So for example, given

```
void BF_encrypt(BF_LONG *data,BF_KEY *key);
void BF_decrypt(BF_LONG *data,BF_KEY *key);
void BF_cbc_encrypt(unsigned char *in, unsigned char *out, long length,
                    BF_KEY *ks, unsigned char *iv, int enc);

&cbc("BF_cbc_encrypt","BF_encrypt","BF_encrypt",1,4,5,3,-1,-1);

&cbc("des_ncbc_encrypt","des_encrypt","des_encrypt",0,4,5,3,5,-1);
&cbc("des_ede3_cbc_encrypt","des_encrypt3","des_decrypt3",0,6,7,3,4,5);
```