Useful x86 instructions

This is a very small subset of the available instructions but should be enough for your purposes.

Data movement

mov mem, reg/lit

example

```
mov [ebp+12],eax
mov dword[esp+4],12
```

means

store the right operand in the memory location on the left

mov reg, mem/reg/lit

example

```
    mov ebx,1
    mov ecx,[esi+ebp]
    mov eax,ebx
```

means

load the right operand into the register on the left

movss mem, reg

example

```
    movss [ebp+12],xmm0
```

means

store the right operand in the memory location on the left. The right operand is the bottom 32 bits of an xmm register.

movss reg, mem/reg

example

```
    movss xmm1,[esi+ebp]
    movss xmm1,xxm2
```

means

load the right operand into the register on the left, the left operand is the lower 32 bits of a xmm register and the data should be a 32 bit float

movups mem, reg

example

```
    movss [ebp+12],xmm0
```

means

store the right operand in the memory location on the left. The right operand is a 128 bit xmm register.

movss reg, mem

example

```
   movups xmm1,[esi+ebp]
```

means

load the right operand into the register on the left, the left operand is a 128 bit xmm register

push mem/reg/lit

example

```
    push dword 10
    push dword[esi+ebp+40]
    push ecx
```

means

push the operand on stack, pre-decrementing the esp register by 4

pop mem/reg

example

```
    pop dword[esi+ebp+40]
    pop ecx
```

means

the operand is assigned the value on the top of stack and the stack pointer is then incremented by 4

fld mem

example

```
    fld dword[esi+ebp+40]
```

means

the operand which is assumed to be a 32 bit floating point value is pushed on the fpu stack

fild mem

example

```
    fild dword[esi+ebp+40]
```

means

the 32bit integer operand is pushed on the fpu stack as a floating point number

fstp mem

example

```
    fstp dword[esi+ebp+40]
```

means

the operand is assigned the 32bit floating point value on the fpu stack the fpu stack is then popped

fistp mem

example

```
fistp dword[esi+ebp+40]
```

means

the 32bit floating point value on the fpu stack is converted to an integer and stored in the operand, the fpu stack is then popped.

Arithmetic

Integer arithmetic instructions can be divided into 3 classes

1. Add, subtract, and, or, xor.These are treated absolutely regularly as two operand instructions as shown below in section **??**.

2. Multiply, this comes in both 2 and 3 operand forms.

3. Divide and Modulus, these are irregular and make use of specific registers

Regular integer arithmetic

These take the form

operation *dest, src*

and mean *dest*:= *dest* operation *src*

the following operation codes are allowed

```
add, sub, and, or, xor
```

The table shows the allowed combinations of destination and source

Operand combinations for regular arithmetic

dest	src
register	register
register	constant
register	memory
memory	register
memory	constant

Examples

```
add esp,5
sub eax, ebx
and [eax+12],ebp
add dword[esi+edi],1
add esi,[edi]
```

Multiply

imul reg,reg/mem

This is functionally the same as the regular 2 operand integer arithmetic instructions.

Example

```
    imul ebx, dword [ebp-26]
```

imul reg,reg,const

This three operand form is particularly useful for computing array offsets.

Example

```
    imul esi,eax,16
```

Divide/modulus

A single instruction is used for both division and modulus.

idiv reg/mem

The 64 bit value in edx:eax is divided by the operand, the quotient is placed in eax, and the remainder is placed in edx.

Example

```
    idiv [ebp+64]
```

Floating point arithmetic

The floating point stack can be used to perform arithmetic in a postfix manner. The following fpu opcodes operate on the top two items on the fpu stack:

```
    faddp st1
    fsubp st1
    fdivp st1
    fmulp st1
```

These perfrom an operation between the top of the fpu stack (st0) and st1, store the result in st1, then pop the stack so that st1 becomes the new top of stack. Bear in mind that the maximum depth of the fpu stack is 8. Operations are performed using 80bit internal floating point representation.

Vector arithmetic

It is possible to perform parallel operations on vectors of 32 bit floats using the xmm registers. These instructions have the general format

*operation*PS *xmmreg,xmmreg*

For example

```
    mulps xmm0,xmm5
```

the suffix PS stands for Packed Single precison floats. In this case the 4 floats in xmm0 are multiplied by the corresponding floats in xmm5 and the result stored in xmm0.

The other useful vector arithmetic instructions in this context are:

```
addps, subps, divps
```

These instructions also exist in a memory to register form but for these to be used you have to guarantee that the operands are aligned on 16 byte memory boundaries. Since this is complicated to ensure, I suggest that you restrict yourself to the register to register forms of these instructions.

Scalar arithmetic

It is also possible to perform scalar arithmetic in the low order 32 bit words of the xmm registers. For instance, you can do all of the vector operations by using the subscript SS standing for Scalar Single precision after the operation thus:

```
addss xmm2,xmm0
```

would add the bottom 32 bit float in xmm0 to the bottom float in xmm2 and leave the result in xmm2.

Conversion instructions

operation	dest	src
cvtsi2ss	xmm register	general register
cvtsi2ss	xmm register	memory
cvtss2si	general register	xmm register
cvtss2si	memory	xmm register

If you are going to use these scalar instructions it is worth taking note of the conversion instructions `cvtsi2ss` and `cvtss2si` which convert signed doubleword integers to single precision floats and vice versa.

Examples

```
    cvtsi2ss xmm4, ebx
    cvtsi2ss xmm3, [ebp+20]
    cvtss2si eax,  xmm0
```

Integer comparisons

Comparison instructions exist which will place the results of comparison in the flags. The `cmp` instruction compares two integers.

Examples

```
    cmp eax, 12
    cmp eax, ecx
    cmp ebx, [ebx+16]
```

Set

The result of the comparison is written to the flags and can be used either by a SET instruction or by a conditional jump instruction.

For instance to test if the eax register was less then 10 we could write

```
    cmp eax,10
    setl bl
```

At the end of this the bl register will contain a boolean value of 1 if eax had been less than 10 and 0 if it had been greater than 10. The suffixes used by the SET instruction indicate which comparison is being tested. The suffixes that are most likely to be of use to you are L, G and E standing for Less than, Greater than, and Equal.

fcomip st0,st1

The instruction fcomip compares the top two elements of the floating point stack, popping the top one from the stack and placing the result of the comparison in the cpu flags.

It may be necessary to discard the next item on the fpu stack using an `fincstp` instruction which increments the floating point stack pointer.

cmpss

There are a family of comparison operations that work between scalar xmm registers. These leave an integer result in one of the registers thus:

```
    cmpltss xmm2,xmm4
```

would compare the float in the bottom 32 bits of xmm2 with the corresponding float in xmm4 and set xmm2 to all 1s if xmm2 was less than xmm4, otherwise it would set xmm2 to zero.

Scalar comparisons

instruction	means
cmpltss xmma,xmmb	xmma<xmmb
cmpeqss xmma,xmmb	xmma=xmmb
cmpnless xmma,xmmb	xmma>xmmb

comiss

Comiss is an alternative technique for performing scalar comparisons it compares the contents of two xmm registers and returns the results in the cpu flags.

Example

```
    comiss xmm1, xmm7
```

Branches

Branches can be unconditional and direct:

```
    jmp lab
```

or uncoditional and indirect:

```
    jmp dword[ebp+10]
```

or conditional on a condition code and direct:

```
    jl lab1
    jg lab3
    je lab4
```

Calls

Calls can be u direct:

```
    call lab
```

or indirect:

```
    call dword[ebp+10]
```

in either case the current value of the `eip` register is pushed on the stack and the `eip` register loaded from the operand. Returns are perfomed using the `ret` instruction which pops the top of stack into the `eip` register.