General Operations

Because, in the end, you're still to write something useful.

ALU instructions

Instruction	Example	Description
add	add eax, 0x04	Add 4 to the content of eax.
and	and eax, 0x01	Do a binary **and** between eax and 0x01.
or	or eax, 0xFF	Do a binary **or** between eax and 0xFF.
sub	sub eax, 0x01	Subtract 0x01 from eax.
test	test eax, eax	This is equivalent to cmp and does not change the register values.
xor	xor eax, eax	Do a binary **xor** between eax and eax.

SHIFT instructions

Instruction	Example	Description
sal	sal eax, 0x01	Proceed to a left shift while keeping the sign of the data in eax
sar	sar eax, 0x01	Same as sal but shift is done to the right.
shl	shl eax, 0x01	Proceed to a left shift. The sign bit is not kept.
shr	shr eax, 0x01	Same as shl but sift is done to the left.
rol	rol eax, 0x01	Rotate the bits to the left.
ror	ror eax, 0x01	Rotate the bits to the right.

Shift arithmetic VS. Shift VS. Rotation

Shift arithmetic (on a byte):

Let's pretend we have the following value

0b00000001

After applying a 1 bit left shift the value will be:

0b00000010

Had the value been **0b10000001** the new value would have been

0b10000010

Shift (on a byte):

Let's pretend we have the following value

0b00000001

After applying a 1 bit left shift the value will be:

0b00000010

Had the value been 0b10000001 the new value would have been

0b00000010

Rotation (on a byte):

Let's pretend we have the following value

0b00000001

After applying a 1 bit left rotation the value will be:

0b00000010

Had the value been 0b10000001 the new value would have been

0b00000011

Multiplication - unsigned (mul)

```
0x080483e3 <+3>:     mov    eax,0x2
0x080483e8 <+8>:     mov    ecx,0x3
0x080483ed <+13>:    mul    ecx
```

```
eax            0x2      2
ecx            0x3      3
edx            0xbffff0a4       -1073745756
```

Both terms of the multiplication are set in **EAX and** in **another register**. After the multiplication, the **result** will be in **edx:eax** where **eax** contain the **lower 32 bits** of the result.

```
eax            0x6      6
ecx            0x3      3
edx            0x0      0
```

Multiplication - signed (imul)

```
0x080483e3 <+3>:     mov    eax,0x2
0x080483e8 <+8>:     mov    ecx,0x3
0x080483ed <+13>:    imul   eax,eax,0x3
```

```
eax            0x2      2
ecx            0x3      3
edx            0xbffff0a4       -1073745756
```

```
eax            0x6      6
ecx            0x3      3
edx            0xbffff0a4       -1073745756
```

imul allows for this form:
imul <dst>, <src1>, <src2>
There are 3 variants of the instructions, you should have a look at intel documentation should you be interested! (volume2)

Division - unsigned (div) AND signed (idiv)

There is a catch using idiv.
Can you guess it?

```
0x080483e3 <+3>:     mov    eax,0x5
0x080483e8 <+8>:     xor    edx,edx
0x080483ea <+10>:    mov    ecx,0x2
0x080483ef <+15>:    div    ecx
```

```
eax            0x5      5
ecx            0x2      2
edx            0x0      0
```

```
eax            0x2      2
ecx            0x2      2
edx            0x1      1
```

Both div and idiv work the same way!

The **div** instruction uses **edx:eax** to hold the **number to be divided**. The **divisor** is then determined by the **operand** used with the div instruction. After the div, **eax** will **contain** the result (**quotient**) of the division while **edx** will **contain** the **remainder**. Be careful!, AH:AL will be used for division on bytes and DX:AX for division on words (16 bits).

Lets write some code!