Prologue

Hello dear readers, this is the second chapter of my tutorial on maths in assembler which was first published in GO64! magazine. The first chapter was re-published in Attitude #4.

Mathematics in assembly, part 2

- Written by Krill/Plush.
- Published in Domination #17
- Converted to ascii by Jazzcat/Onslaught.

Last time we discussed the basics, the representation of negative numbers, and a bit of elemental maths. This time, as promised, we pump up our number range of 8 bits a bit.

Our small number range of 8 bits may be sufficient for the beginning but soon it is not enough for one's purposes any more. But how to enlarge it? Quite simple. Let's say we want to have a 16 bits number, that's two bytes. So there are 16 bits now, that means the number can have values from 0 through 2~16-1, i.e. 65535. The second byte is just the logical continuation of the first one, it contains the bits 8 to 15. But how to compute using this number?

Computing with 16 bits

At first one should store the number like our computer proposes, in the order lobyte, hibyte. That means the first byte contains the less significant bits 0 through 7, the following byte the more significant bits 8 through 15. The simplest type of calculation, to add numbers, works just like this:

```
    CLC           ; clear carry bit
    LDA number1lo
    ADC number2lo ; add lobytes
    STA number3lo ; result's lobyte
    LDA number1hi
    ADC number2hi ; add hibytes
    STA number3hi ; result's hibyte.
```

The carry bit is cleared, the lobytes are added and the result is stored. But the carry bit is not cleared before adding the hibytes because of this: if there is an overflow while adding the lobytes, the result of the hibyte addition is automatically increased by one because of the previously set carry bit. Just an example - $0340 and $05C1 are to be added. So at first $40 and $C1 are added, afterwards the accu contains $09 because the carry bit has been set beforehand - ADC means add with carry bit. The correct result of $0901 is calculated. The subtraction is done in an analogous way; the only difference is to first set the carry bit instead of clearing it. If the numbers are signed it is just as easy, the results stay correct. The example routine may be extended for more than 16 bits, just add more ADCs without CLCs. Halving and doubling is again a little more complicated.

Shifting 16 bits

If the number $17C4, as an example, is to be doubled that would look like this:

```
    LDA number1lo ; $C4
    ASL
    STA number2lo ; $88
    LDA number1hi ; $17
    ROL
    STA number2hi ; $2F
```

The result of $2F88 is stored in number2. After shifting the lobyte left, the accu contains $88 and the carry bit is set. When rotating the hibyte left, the carry bit is rotated into the result which will then be correct afterwards. Again it's possible to extend the routine for more than 16 bits, just add some more rotations. When halving, the sign has to be taken care of again (see last chapter, published in Attitude #4). It is located in the MSB, like usual, which is bit 15 in this case. The hibyte has to be halved first because the bit falling out must be rotated into the lobyte. It should look like this:

```
    LDA number1hi
    CMP #$80
    ROR
    STA number2hi
    LDA number1lo
    ROR
    STA number2lo
```

With the gained knowledge we are able to perform simple calculations with 16 bits - but what exactly is it what we have gained with those 8 more bits?

The interpretation of our numbers depends on us because the computer just dumbly computes, while we interpret the values. As an example, we could say our numbers range from 0 through 65535, or, with sign, from -32768 through 32767. So we have a much larger number range. What if we interpreted the lobyte as fraction byte? We would have fixed point numbers with an accuracy of 1/256! In that case the 8 lowmost bits won't have the significances of 2~0 to 2~7 but 2~-8 to 2~-1 instead. What's the deal with that?

The usage of fraction bytes

Ofcourse at first more accuracy. As a simple example besides complicated calculations the following shall be mentioned: an animation is held in the memory. When showing another animation step each screen frame, the animation is too fast. When showing the same animation steps for two screen frames, it's too slow. So a following animation step must be shown 'each one a half frames.' This is a 16 bit fixed point number is useful. It's initialised with $0000. If a new animation step was to be shown each frame, $0100 would be added to the number, as the hibyte represents the number of the animation step. If the animation should be at only half that speed, $0080 would be added each frame. What's needed is a value right between these two, $00C0. Using it, the number will be $0000, $00C0,

$0180, $0240, $0300, $03C0, $0480, etc., successively. So the hi-bytes, i.e. the shown animation steps, are $00, $00, $01, $02, $03, $03, $04, etc. The animation speed is right between 1-framed and 2-framed.

More than 16 bits

As already mentioned above, the routines can be extended to more than 16 bits without any problems. Only the position of the decimal point has to be defined by us, nothing more. One could, for example, calculate using 24 bit numbers having two fraction bytes, which would mean an accuracy of 1/65536. But that wouldn't be very useful with additions, subtractions, shifts and rotations. From the next chapter on the multiplication will be discussed. That's were a high precision is rather needed.