Prologue

Hello dear readers. I noticed a need for some tutorials on coding and decided to re-publish the texts i have been writing fo “GO64!” magazine. Cactus gave me the impulse to do that. “Attitude” is the first diskmag to publish a chapter of the tutorial on maths in assembly. The following chapters will be published in this mag and also other ones like “Vandalism News” and “Domination”. Just watch out. Many thanks so Cosowi/Plush, the publisher of “GO64!”, for the kind of permission to re-publish these articles.

Mathematics in Assembly

Calculations in machine language are anything else than trivial: the command set of the C-64 supplies all powerful operations - addition, subtraction, shift and rotate, and that's it. That's sufficient for the beginning, but when it comes to coding demos it isn't any more. An alternative would be to use the accurate floating point routines of the Kernel, but they are far from what we call fast and optimized. What we need are selfmade, fast and sufficiently accurate routines.

Fixed point arithmetic routines are fast and accurate enough. They enable us to easily define the necessary accuraxy (i.e. the amount of fraction bits) and the computational range that means the range of numbers the routines ar appliable on. A common example would be 0 through 65535 ($0000 through $FFFF. without sign) or -32768 through 32767 ($8000 through $7fff, with sign), respectively. But more on that later.

The Basics: Binary Representation of Numbers

This tutorial assumes basic knowledge of assembly language and the hexadecimal/ binary number system. It should be clear how such a hex number looks like. Alright, so we know common hex numbers on the C-64. Usually they range from 0 through 255, which equals $00 through $FF. But whar if we also need negative numbers for our computations?

Negative Numbers in Assembly?

Very simple, we just divide the range in half. This means we use one half of the range for positive, the other one for negative numbers. The positive half ranges lfrom 0 through 127 (0 is positive for a computer) and the negative from -128 through -1. Basically there are two ways to binary represent signed numbers where it turned out to be most useful to use the upmost bit (MSB, Most Significant Bit), which is bit 7 in the simplest case, as the sign. Acleared bit means a positive number, a set bit a negative number. One could represent for instance -3 as $83 (binary %10000011), but this is not a very clever solution. The so-called two's complement is the way to go. Basically this means if we want

to have a negative number, we subtract its absolute value from zero. Sounds logical, as -3 is 0-3. This means in hex: -3 = $00-$03 = $FD, for 8 bit large numbers. Because of that, -1 is not $81, but $FF. An alternative way of calculating negative numbers would be th negate them bit-wise (i.e. perform an EOR #$FF) and afterwards add 1. The result is the same while the later method is faster. So why do we do it like this? It's because this system makes additions and subtractions very easy. Just imagine we'd use the mentioned naive system where -1 would be $81. For a simple addition, the signs had to be checked first, the numbers made positive accordingly (in case they are negative) and afterwards their absolute values had to be subtracted or added, depending on the numbers' signs. Not very effective.

Signed Addition and Subtraction

Using the two's complement system, the whole deal is extremely simple: -5+4 is -1. In assembly this would be: $FB+$04=$FF. Analogous to that 8-12=-4, so $08-$0C=$FC. That should be clear. But the command set of the 6502 processor family also features the shiftand rotate commands. So how to take care of the sign there?

Signed Shift and Rotate

Shifting left doubles, shifting right halves a number. Now we should be a bit more cautious. Doubling 5 makes 10, so $05+$05=$0A. Shifting left gives exactly this result (the number to be doubled must not be larger than 127, as 2×128 would already be 256 (=$0100), which doesn't fit into our number range of $00 through $ff - using the carry bit here would help us, but more on that later). Shifting right (halving) for instance $24 makes $12 which is also correct (but here we have to take care off odd numbers, bit 0 is pushed into the carry bit). Obviously there are no problems with positive numbers. Rather with negative ones. Shifting a number like $EC (-20 or %11101100) left results in $D8 (-40 or %11011000). Exactly. But trying to halve it again using LSR, the result is not $EC but $6C(108 or %01101100).

The problem with halving and a negative sign

The reason is clear, the upmost bit was lost before. Now it becomes obvious why the command shifting left has different “properties” than the one shifting right: left is “arithmetic” (ASL), and right is only “logical” (LSR) because the correct sign is preserved when shifting left in opposite to shifting right. So what to do? Very simple. The sign just has to be preserved when shifting right. Shifting a negative number right, the result has to be negative, too. This is the first time the rotate commands are useful. Some small example code:

```
LDA #$EC ; -20
CMP #$80 ; MSB to carry bit
```

```
ROR      ; halve
```

Now the correct result, $F6 (-10, %11110110), is computed. This routine of course also gives correct results for positive numbers. The absolute value of a doubled number must not be larger than 127 ($7F), otherwise the number range had to be enlarged to more than 8 bits. As mentioned before, the carry bit can be used to help if doubled number exceeds the number range, as an ASL or ROL commands pushes the upmost bit to the carry bit which can be taken care of afterwards.

This was just the beginning. Enough for now. Just take your time to digest the whole thing. In the next part of this turorial the small 8-bit number range will be pumped up a little, up- and downwards.