Multiplying by more than 1.0 in the FXCore

Preface

Math in processors always has limits, in fixed point it is the range but maintains high resolution. In floating point the range is wide but we trade off resolution as we need bits for the exponent. As a result each has its place, floating point is great for sending a probe to Pluto as it works with a large range of numbers while fixed point is great for things like audio as we work in a limited range. But there are times we need a feature from the other number type, like being able to multiply by a large number that cannot normally be expressed in a fixed point number so over the years programmers have developed tricks to get around these limits

Multiplying by numbers larger than 1.0

When looking at the FXCore it appears there is no way to multiply by a number larger than 0.999 except by shifting which limits us to powers of 2. So it seems we are limited by fixed point math that we cannot multiply by a value like 3.16 but the truth is we can if we break up the multiply into a few steps:

Let us say we want to multiply a value “X” by a coefficient “Y” which is larger than can be represented in our number system but we have the ability to multiply by some large constant “Z” in the number system, as a result we can break up the number and do it in the following manner:

ACCUMULATOR = (Y/Z)*X
ACCUMULATOR = Z*ACCUMULATOR

“Y” must always be less than “Z”

So in FXCore we could multiply a number by something like 3.16 by calculating the following

Y/Z = 3.16/4 = 0.79 – note we are dividing by a power of 2 so we can use 2,4,8, etc. Use the smallest power required

After the multiply of 0.79*X we shift the result 2 bits left to do a multiply of 4 and we have the result.

As the multiply is now <0.999 we can do it in the FXCore and as we have a saturating left shift function in FXCore we can shift 2 bits left for the multiply of 4.

Combining with LOG2 function

If you have read the other app notes and looked at the available sample code you will have seen that:

X^N = EXP2(N*LOG2(x))

As FXCore has both LOG2 and EXP2 functions it is very easy to calculate X^N. In most cases we are interested in taking a root such as square root or cube root of a number so we see N=0.5 or 0.333 in these cases. But as we now know how to multiply by values greater than 1.0 it means we can raise number to any value so if we wanted X^1.6 we take LOG2(X), use the trick of:

ACCUMULATOR = (1.6/2)*LOG2(X)

Left shift ACCUMULATOR 1 bit to multiply by 2

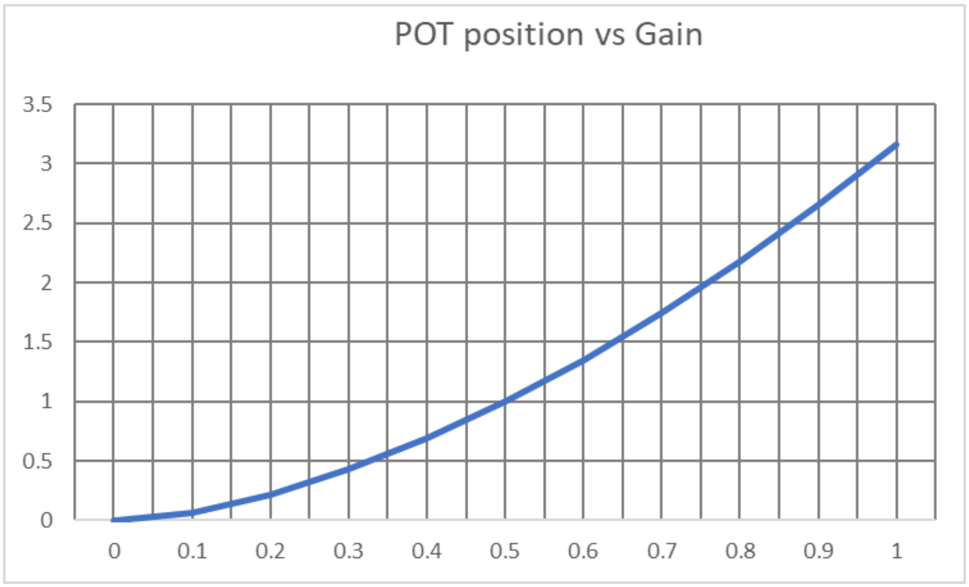
EXP2(ACCUMULATOR) and we have the result.

Practical example

If we look at the equation:

K=3.16*(X^1.66)

We can see that for X=0 K=0 and for X=0.5 K=1.0 and for X=1.0 K=3.16. It is obvious this is not a linear curve and is actually the curve for a mixer pot that controls the volume of an audio signal. At x=0 we would have no output (-inf db), at x=0.5 we have unity gain (+0db) and at X=1.0 we have a gain of 3.16 (+10db).



So we know we can now calculate a coefficient K between 0 and 3.16 but we now have a problem, we cannot represent a value of 3.16 in our fixed point system nor do we have a multiply instruction that can accept a coefficient >+0.999, luckily the commutative property of multiplication allows us to get around this limit. The commutative property of multiplication states that the order of operation in multiplication does not change the result so A*B*C is the same as C*B*A, as such we can do the shift after we multiply by the coefficient. For example we want to multiply IN0 by the above equation such that:

OUT0 = IN0*3.16*(X^1.66)

We can do:

OUT0 = 4*(IN0*(3.16/4)*(X^1.66))

The multiply by 4 can be done as the final step and is just a left shift of 2 bits. The shifter in FXCore allows for shifts of 0 to 31 bits so if one of the numbers is <1.0 you can use a shift of 0 as 2^0 = 1. As the FXCore has a saturating shift function it will clip if the result would have exceeded +0.99999/-1.0.

The new “SIG*A*X^B” block in the utility library implements this function, it has 4 parameters that must be set by the user and these are:

A : the actual value so in this case 3.16
A_shift : The number of bits to shift for a power of 2, in this case set it to 2 as 2^2 = 4 and that is greater than 3.16
B : The actual value, 1.66 in this case
B_shift : The number of bits to shift for a power of 2, in this case set it to 1 as 2^1 = 2 and that is greater than 1.66

This block has an input pin for a signal and an output pin for the result. You can see this block used in the mix_4_2_lshs_V2.gcf file where we use it to be able to add gain to signals.

Experimental Noize Inc.

Scottsdale, AZ USA

www.xnoize.com

sales@xnoize.com