A Beginner's Guide to Programming Digital Audio Effects in the kX Project environment

Martin Borisov (Tiger M)
Santiago Munoz (eYagos)
revised by
Max Mikhailov (Max M.)

Introduction

Digital audio effects are a small branch of the science of digital signal processing (DSP), which comprises many different types of applications such as image processing, communications, medical instrumentation, military instrumentation, deep sea and space exploration etc. All these deal with processing different signals (a signal being a stream of continuous data) and so does audio processing.
The digital signal processor of the sound card driven by kX gives us many possibilities for audio effects implementation and kX itself consists of tools that help us a lot in achieving this. Not many people comprehend the power that this combination puts in our hands. In fact we have a fully programmable digital effects processor, which we can command in the most subtle way possible by using the lowest level assembly instructions provided. We can program all possible types of effects known in the audio world. Some things might seem complicated, but don't be scared, I can assure you that you would be able to comprehend it.
In this guide I've tried to keep things as simple as possible and give examples, which are the essence of learning DSP techniques in kX. We willl use the kX Editor (Dane) which is a very convenient assembler. Although it doesn't give the flexibility of C++ programming, it will be enough for the basics and will give you a jump start on DSP programming. And if you are familiar with object oriented programming in C++, than you have all the power in your hands. But note: no C++ knowledge is required for understanding the information in this guide.
The structure of the guide is practicaly oriented. I'll give examples coupled with explanations of the 16 instructions used in programming the E-mu10kx chips. You can copy/paste and test them in Dane. The instructions are the programming language of the processor, often called the microcode/opcode.
In the end we'll "disect" several of the commonly used simple kX Project effects.

The digital signal processor

When a signal (in our case analog audio signal) enters the input of the soundcard it goes into an ADC (analog to digital converter), which transforms the voltage of the sound signal in regular intervals of time (in our case it does this 48000 times per second - 48KHz) to a number, thus making it digital (discrete). So, 48000 numbers form 1 second of audio signal or vice-versa. Then every number of this continuous array of values passes through the digital signal processor where it gets transformed by the effects. After that it goes into the DAC (digital to analog converter) where it gets analog again and goes to the sound card outputs. In some cases the signal might be extracted before it gets transformed by the DAC, and sent to the digital output of the sound card in digital form.
We transform the signal with audio effects when it is in the form of numbers and this means we have to

deal with math and nothing else! So, DSP effects are just mathematical functions, which most of us study in school. Actually each instruction is a small mathematical function all by itself. A combination of these forms a system - the audio effect. This system of mathematical functions is performed on each value of the transformed signal and the processor does this 48000 times per second.

Let's begin

Let's start by opening kX Editor. Rigt-click the kX icon in the taskbar and open kX Editor.
In the editor window you'll see lines starting with ; - these are just comments, in the upcomming examples there will be many of these. You'll see lines like "name", "copyright" and so on, which are only for providing information for the plugin and are not related to DSP. You can write your own information there for each effect. The "guid" line is important, because it guarantees the uniqueness of the plugin, so it doesn't overwrite existing effects when it is registered in kX. You can obtain an unique guid by going to the 'settings' tab and selecting 'generate guid'. This copies the new guid to the clipboard and you just have to paste it in your code. You'll also see an "end" word, which indicates where the code ends. It should always be there for the code to get compiled (parsed).

NOTE: In kX editor you can use both the decimal and hexadecimal (aka machine) number systems. It is advisable to use decimal, because this is the natural “human” number system and decimal numbers are automaticly transformed into hexadecimal.
For example, if you write 1 it will be transformed to 0x7fffffff. You don't have to know hexadecimal numbers to program audio effects in kX, in fact it will just be a waste of time. You can convert hexadecimal to decimal in windows calculator for instance, with the following formula: hex/(2^31 + 1) = dec, or decimal to hexadecimal: dec*(2^31 – 1) = hex.
Numbers processed by the E-mu10kx digital signal processor are fixed-point 32 bit fractional between -1 and 1 or integer values (whole numbers).

Registers

Unprocessed, intermediate and processed data of the audio signal stream is stored in physical registers. There are several types of registers:

1. **Input** and **output** registers. Incomming data unprocessed by the current effect is stored in the input register, so subsequently it can get processed. Already processed data is stored in the output register, so it can be rooted to physical outputs or other effects. Each microcode can have several input and output registers depending on how many channels we want it to have. For example, if it's going to be stereo we would need two of each type. If it's only mono we will need just one of each type. The use of such registers is not necessary for effects which don't need both inputs or outputs – for instance peak meters or wave generating effects. But at least one type is needed, otherwise the system would be meaningless and useless.

Declaration:

```
input in
output out ; in and out are just names, you can assign the registers
any name
```

NOTE: the value of the output register can be reused in the next sample cycle before the processor gets to calculating the operation, the result of which is written to the same output register, when it gets overwritten with the next output value.

2.**Static** and **temp** registers. They are used for storing intermediate data during instruction execution. The value of a static register is preserved until it is overwritten (next sample cycle), or the microcode is reinitialized (reloading or resetting the plugin). Temp registers are used for the present sample cycle only, so their last value can not be used in the next cycle – it will be zero. The idea behind the temp register is that it can be shared by all loaded effects, but such sharing is not supported by kX at the present time, so it is recommended that static be used most of the time. Despite that we'll use temp registers in the examples for learning purposes only.

Declaration:

```
static st
temp tmp ; st and tmp are again just names
```

Static and temp registers don't have to be initialized with a value, but if an initial value is needed you can assign such to a static register:

```
static st = 0.8
```

NOTE: when you use certain numbers (constants) directly in the microcode which are not present in hardware (in a special read-only memory), they are automatically transferred to static registers when the code is compiled.

3.**Control** register. This is a read-only register and has to be initialized with a value. When the code is compiled, a fader is automatically created for that register, so it can be user controlled. It can have values between 0 and 1 (although you can assign values greater than that, when you move the corresponding slider the value is automaticaly transformed between 0 and 1).

Declaration:

```
control volume = 0.2
```

The assigned value is the default value, so each time the code is reinitialized the control register will have this value.

4. **Constants**. There are certain constant values, which are defined in hardware in a read-only memory on the chip and can be used directly in the code (not with a static register) for the purpose of saving some resources. Such are 0, 0.125, 0.5, 0.75, -1, 1, 2, 3, 4, 8, 10, 20, 100, etc. If not hardware-defined constants are used, they will be automatically transformed to static registers, as mentioned before.

5. **Accum** register. This register gives us access to the dsp accumulator. When an instruction is executed, its result is automatically stored in it overwriting the previous value and then copied to the result register of that particular instruction. You can access the accumulator with the accum keyword and it can be used only as an A operand. It is 67 bit wide and has 4 guard bits. It can be used when we want an unsaturated or unwrapped intermediate result, for instance:

```
macs 0, 1, 0.5, 1 ;1+ 0.5 = 1.5

macsn out, accum, 0.6, 1 ;out=1.5-0.6=0.9
```

6.**CCR** (Condition Code Register). This register is used in the skip instruction. Its value is set after each instruction, based on its result. You can acces it with the ccr keyword.

7.**TRAM Access Data Registers**. These registers are for delay lines. There are write registers (which write samples in the delay line) and read registers (which read (extract) samples from the delay line). You can change the address of these registers within the declared delay line, thus changing the lenght of the delay line and the lenght of the actual delay. In Dane this is done by putting an & sign before the name of the corresponding register.

Instructions

All instructions have the following syntax:

instruction result R ,operand A, operand X, operand Y

Operands are registers.

1.**MACS** and **MACSN** (multiply-accumulate with saturation on overflow) That means that you multiply two numbers, then add them to a third number and store the value in the result register. These instructions operate with fractional numbers only (they perform fractional multiplication)! If the result exeeds -1 or 1 it is truncated to -1 or 1. This means that you can't use whole (integer) numbers with these two instructions except “fractional” 1.

Formulae:

MACS R = A + X*Y
MACSN R = A - X*Y

Example:

This is a simple volume control program.

```
name "Volume control";
copyright "Copyright (c) 2004.";
created "06/27/2004";
engine "kX";
guid "...It will be automatically generated!!!..."; Don't copy this


;we define the registers
input in
output out
control volume=1
```

```
macs out, 0, in, volume ;out = 0 + in * volume
end ;don't forget this
```

You can try it in kX Editor. Click on "Save Dane Source" (on the right of the window) not "Export to C++". Save it, then right click on the DSP window and select "Register Plugin". Open the file and it should now be with the other effects - you know where they are. Or you can find the file in windows explorer and double-click on it – it will automatically get registered.

2.**MACW** and **MACWN** (MAC with wraparound on overflow) Same as MACS and MACSN, but when the value exeeds -1 or 1 it wraps around. This is presumably to minimize noise when saturation occurs with MACS and MACSN. If you have 0.5 + 0.7, the result instead of 1 on saturation, will be -0.8 with warparound.

3.**MACINTS** (saturation) and **MACINTW** (wraparound). Same as MACS and MACW, but they perform integer multiplication. That means that you can multiply a fractional value with an integer value as well as integer with an integer. These two instructions always assume that the Y operand is an integer.

Formulae:

MACINTS R = A + X*Y
MACINTW R = A + X*Y

Example 1:

NOTE: I won't write the info part (name, copyright etc.) anymore. You can do that yourselves.

A simple gain. We multiply the input by 4 (that's a mono version of the x4 effect).

```
input in
output out

macints out, 0, in, 0x4 ; out = 0 + in * 4
end
```

Example 2 :

If we want to control the amount of gain:

```
input in
output out
control gain=0.25
temp t

macints t, 0, in, 0x4 ; t = 0 + in * 4
macs out, 0, t ,gain ; out = 0 + t * gain - its actually a level
control
```

```
end
```

4. **ACC3**. This instruction just sums three numbers. It saturates on overflow. The values of the operands can be all fractional (we treat the result as fractional) or all integer (we treat the result as integer).

Formula:

ACC3 R = A +X +Y

Example:

Mix of three mono sources plus a volume control.

```
input in1, in2, in3
output out
temp t
control volume = 0.33

acc3 t, in1, in2, in3 ;t=in1 + in2 + in3
macs out, 0 , t , volume ; out = 0 + t * volume
end
```

5. **MACMV** (MAC plus parallel move of operand A to operand R). The result of X*Y is added to the previous value of the accumulator and the result is again moved into the accumulator. At the same time the value of A is copied to R. This is very useful for filters, since it simultaneously accomplishes the MAC and the data shift required for filtering.

Formula:

R = A, accum += X * Y; or (accum = accum + X*Y)

Example:

See the EQ Lowpass disection at the end of the document.

6.**ANDXOR** used for generating standard logical instructions. I haven't had any experience with this instruction and there doesn't seem to be much use of it. If you want more details take a look at the As10k1 manual. If anyone wants to add to this section, please contact me (info in the end of the document).

7. **TSTNEG**, **LIMIT**, **LIMITN** give the possibility of using something close to "if... then..." statements in the microcode.

Formulae:

TSTNEG R = (A >= Y) ? X : ~X

If A>=Y, the result will be X, else (if A<Y) X is complemented (becomes negative).
This instruction could be used for obtaining the absolute value of a number:

```
tstneg out, in, in, 0 ;out = abs(in)
```

LIMIT R = (A >= Y) ? X : Y

If A>=Y the result will be X, else (if A<Y) the result will be Y.

LIMITN R = (A < Y) ? X : Y

If A<Y the result will be X, else (if A>Y) the result will be Y

Example:

A simple hard clipping fuzz. It cuts the wave over 0.05 and under -0.05, thus producing harmonics.

```
input in
output out
static negclip=-0.05, posclip=0.05 ;negative and positive clip limits
control volume=0.8
temp t

limitn t, posclip, posclip, in ;if in>posclip, t=posclip; else t=in
limit t,negclip, negclip, t ;if t<negclip, t=negclip; else t=t
macs out, 0, volume, t ;level control
end
```

8. **LOG** and **EXP**. LOG converts linear data into sign-exponent-mantissa (scientiffic/logarithmic) and EXP does the oposite. They have many uses as one E-mu/Creative Technology Center official states: "for data compression, dB conversion, waveshaping and log domain arithmetic approximating division and roots".

Formulae:

LOG R, Lin_data, Max_exponent, Sign_register
EXP R, Log_data, Max_exponent, Sign_register

Lin_data: Data to be converted. It would be interpreted as fractional format.
Max_exponent: Must be between 1 (0x1) and 31 (0x1F). This parameter controls, in simple words, the quantity of scale conversion made by the instruction. A value of 1 means no scale conversion. A value of 31 means maximum scale conversion (see gaphs 3, 4, 5, 6). For this reason, we sometimes refer to this parameter as 'Resolution' .
Sign_register: Must be between 0 (0x0) and 3 (0x3). This parameter is used to control the sign of the result:

0x0 - normal
0x1 - absolute value (allways positive)
0x2 - negative of absolute value (allways negative)
0x3 - negative (inverted)

Since the EXP instruction has exactly the opposite behaviour than LOG, we should only discuss the LOG

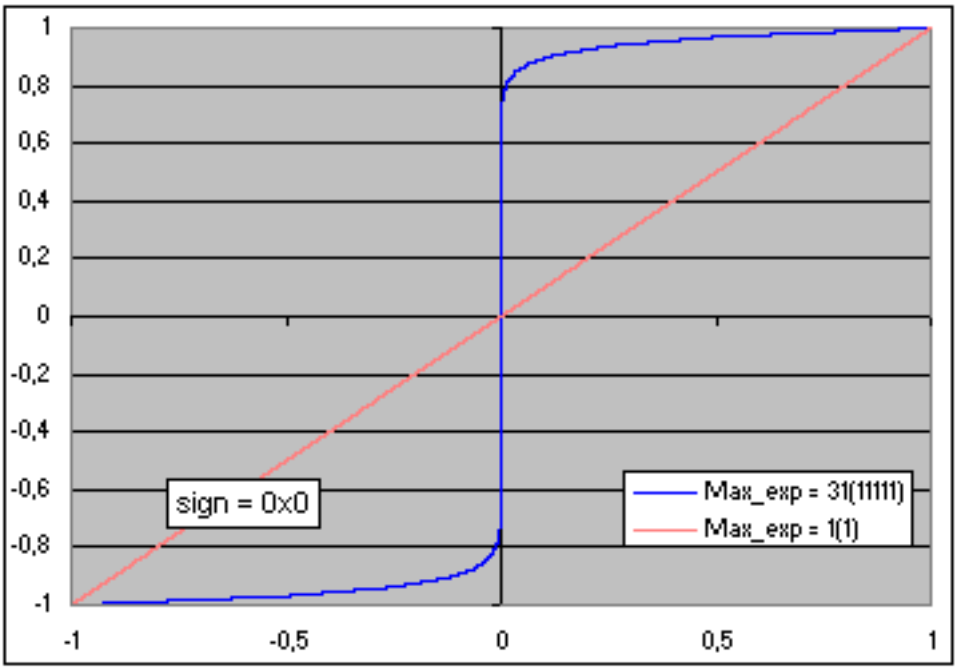
one. The opposite behaviour means that if you calculate the EXP of the result of a LOG instruction, you get the operand of the LOG instruction again (if the same resolutions are used):

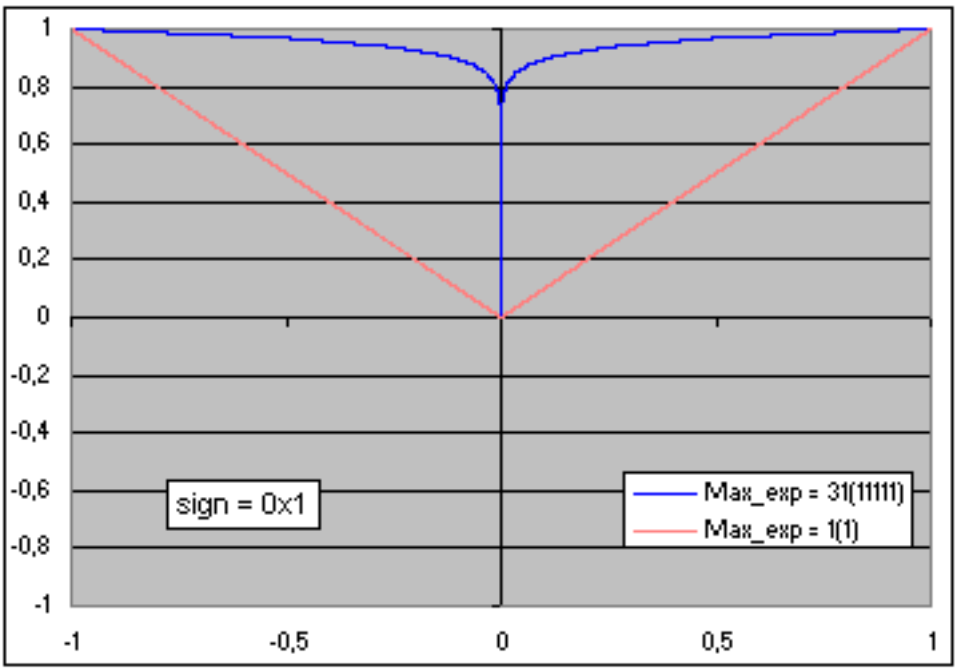
```
log tmp1, x, res, sign ;tmp1 = LOG (x, res, sign)
exp tmp2, tmp1, res, sign ;tmp2 = EXP (tmp1, res, sign) = x
```

Graphs

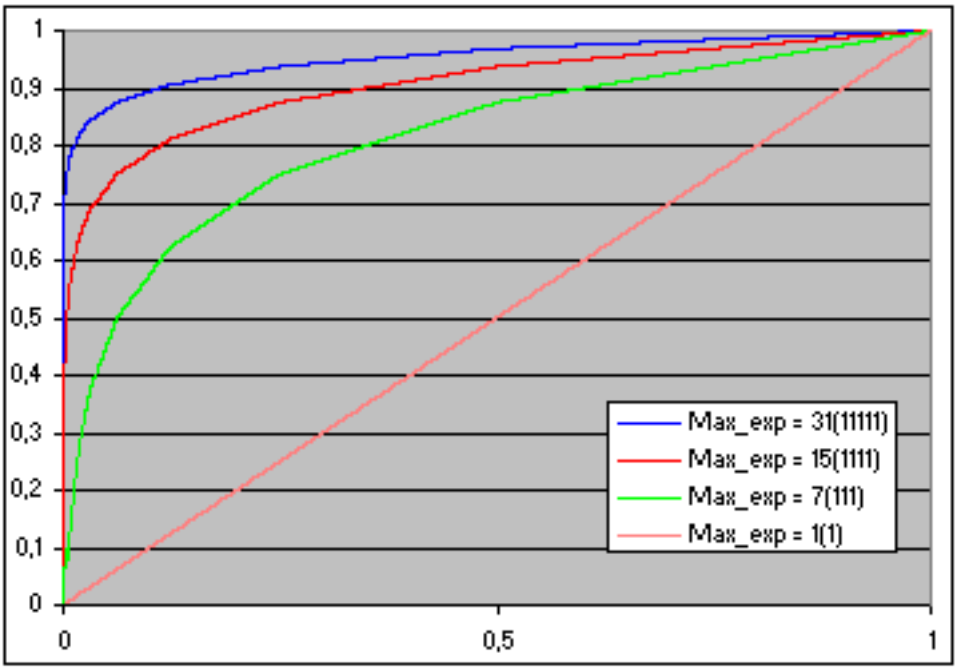
Here you have some plots that can help to understand the behaviour of the parameters of these two instructions.

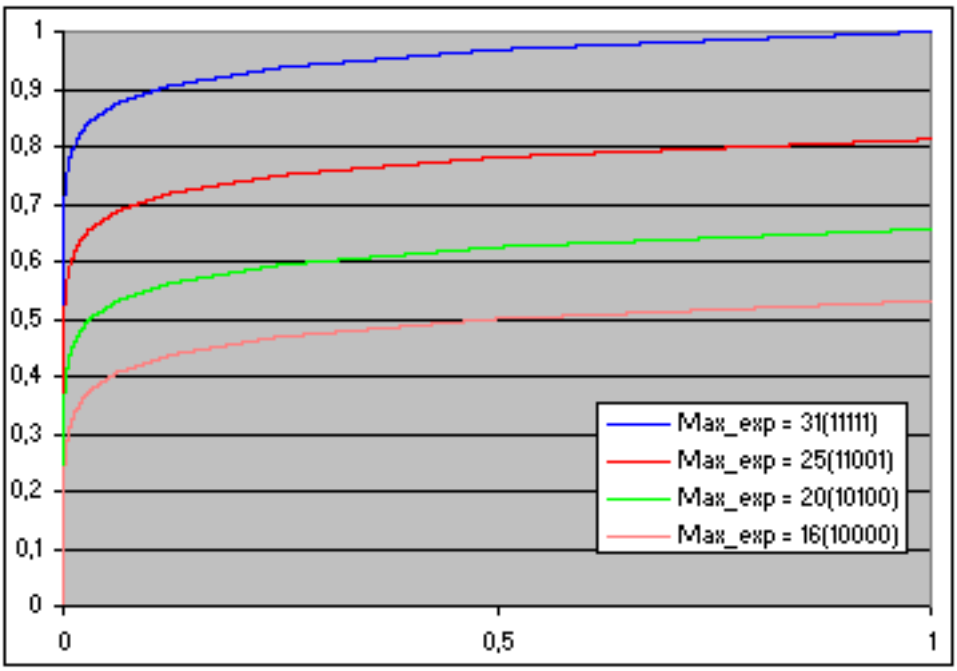
Plots of LOG for the full input range. We can see the difference between Max_exp=0x1F (maximum allowed value) and Max_exp=0x1 (minimum allowed value). We can also see, comparing the two graphs, how the sign_register parameter acts.

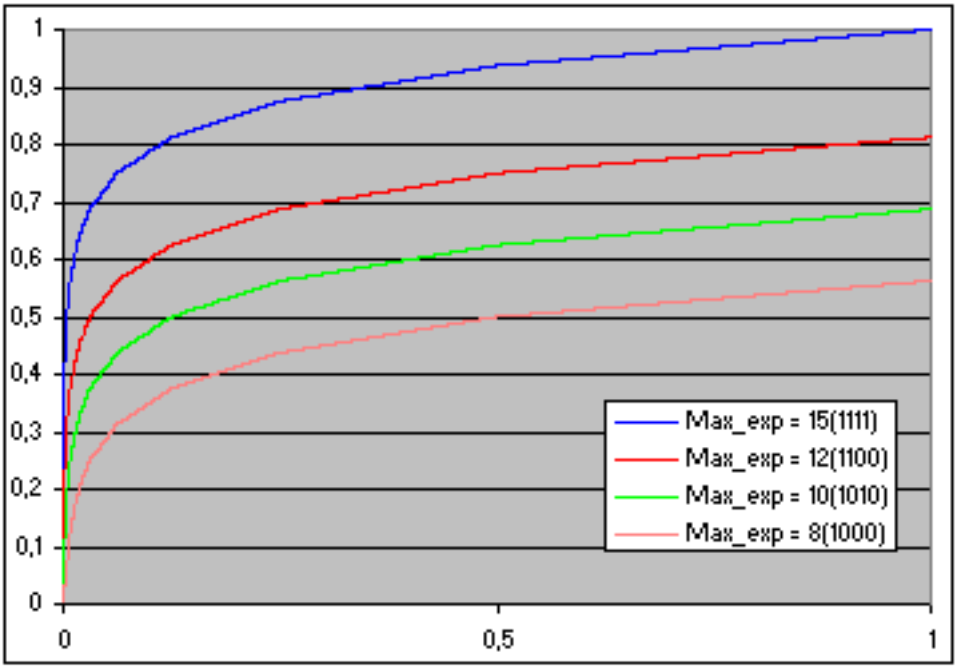


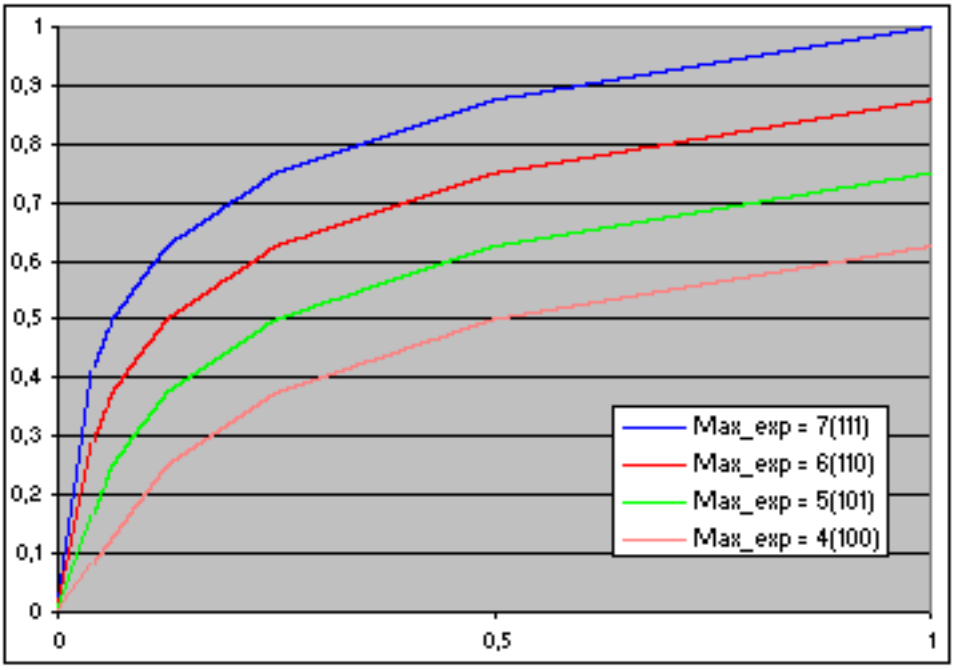


Plots of the various max_exponent values.


1
0,9
0,8
0,7
0,6
0,5
0,4
0,3
0,2
0,1
0
0
0,5
1
Max_exp = 31(11111)
Max_exp = 15(1111)
Max_exp = 7(111)
Max_exp = 1(1)


1
0,9
0,8
0,7
0,6
0,5
0,4
0,3
0,2
0,1
0
0
0,5
1
Max_exp = 31(11111)
Max_exp = 25(11001)
Max_exp = 20(10100)
Max_exp = 16(10000)





Why are these instructions called LOG and EXP?

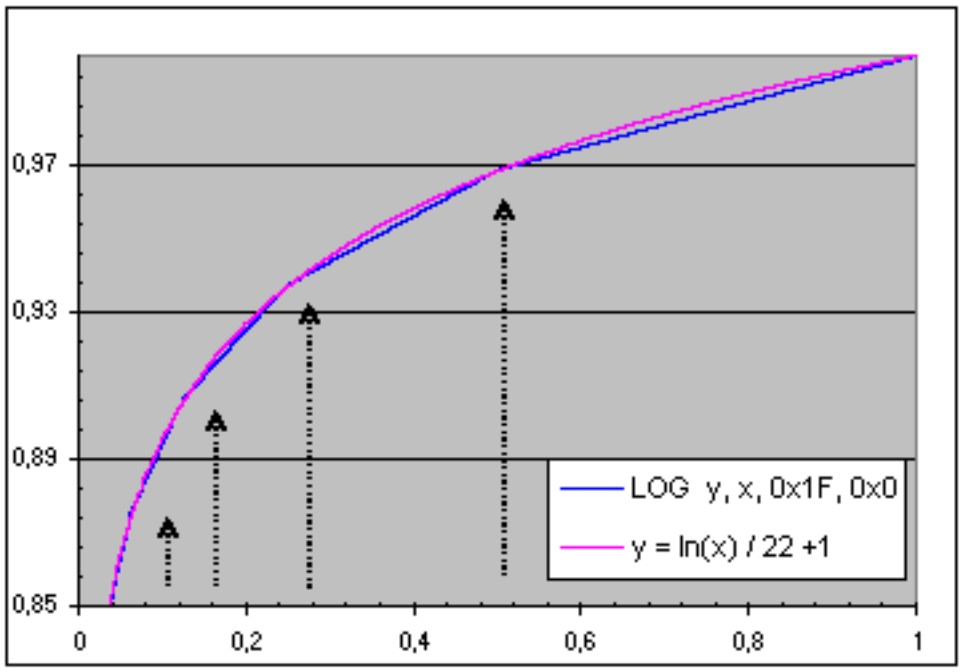
Let's talk about the difference between LOG/EXP instructions, and the log/exp functions in math (I will use upper case for the dane instructions, and lower case for the mathl functions). The LOG/EXP dane instructions are instructions to convert linear data into exponent + mantissa data, and exponent + mantissa data into linear data, respectively. But then..., why are they named LOG and EXP? Well, there is a powerful reason for this - these two functions are good approximations to the mathematical log/exp functions. And the greater the resolution parameter (Max_exp_size), the better the approximation.

NOTE: We allways remember trigonometry, but not always remeber logarithms. Take a look at your old math notes if you don't remember what a neperian logarithm is. -;)

LOG y, x, res, 0x1 approximates to y = log[base](x) + 1 = ln(x) / ln(base) + 1 with:

ln(base)	res
~ 22.18	0x1F (31
~ 11.09	0xA (15)
~ 5.68	0x7 (7

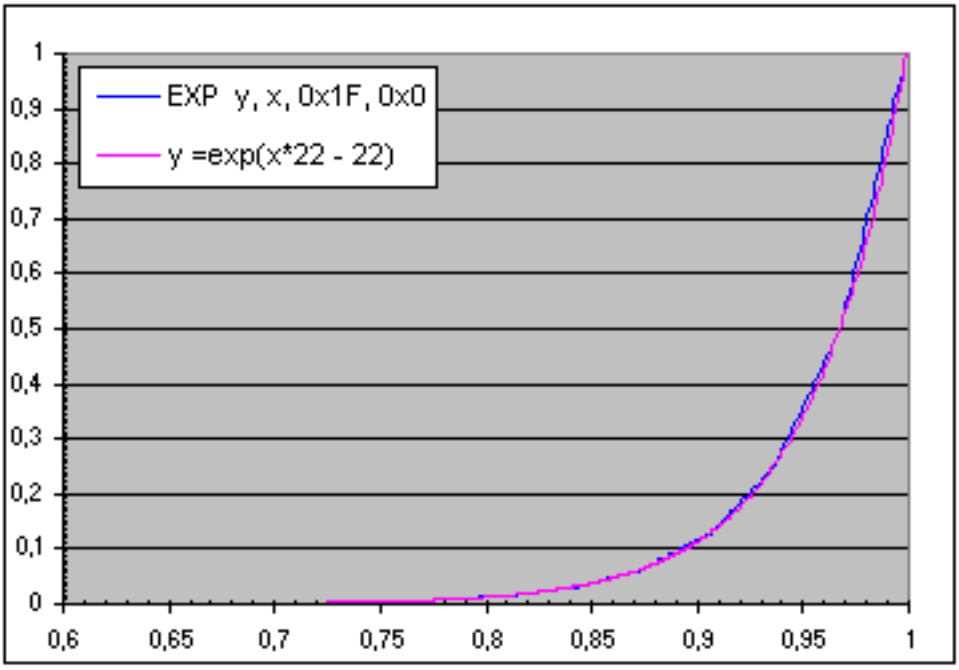
)



Once again, as the EXP instruction has the opposite behaviour than LOG, it can be approximated to the mathematical exp instruction.

EXP y, x, res, 0x1 approximates to y = base^(x - 1) = exp((x - 1) * ln(base)) with:

ln(base)	res
~ 22.18	0x1F (31)
~ 11.09	0xA (15)
~ 5.68	0x7 (7)



Absolute value with the LOG instruction

The LOG instruction can perform the absolute value of the input data using Max_exp_size=1 and the sign_register parameter:

Operation: |x|

```
log y, x, 0x1, 0x1; y = abs (x)
```

Operation: -|x|

```
log y, x, 0x1, 0x2 ;y = - abs (x)
```

There is a loss of two (maybe tree) bits with these expressions. Although the loss of two or tree bits is really insignificant (we still have 32-3 = 29 bits), it is better to use the TSTNEG instruction to do the normal absolute value operation.

Log domain arithmetic

We can use the LOG and EXP instructions to perform operations like divisions and/or roots. But don't forget these two important things:

1.Since LOG and EXP are not exactly the log and exp functions, the next is only an approximation, and may not be enough in many cases.
2.In general, we must use always the max allowed value of Max_exp_size (0x1F) parameter to get a better approximation.

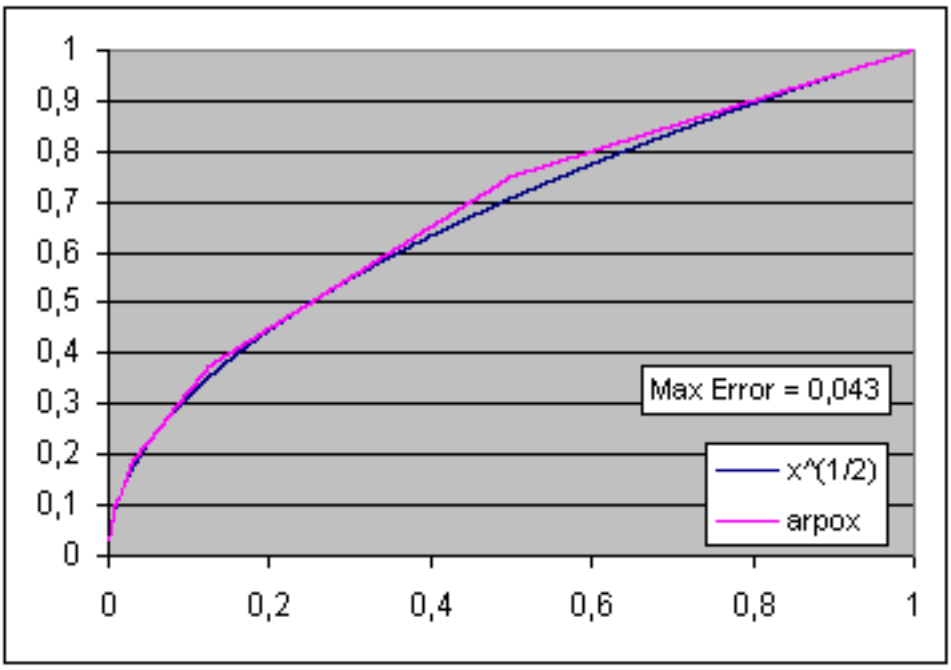
These algorithms are based on the formulas:

log (a ^ b) = b * log (a)
a ^ b = exp (log(a^b)) = exp (b*log(a))

Square root approximation:
Operation: a ^ (1/2)

```
log tmp1, a, 0x1F, 0x1 ;tmp1 = log[base] (a) + 1
macs tmp1, 0.5, tmp1, 0.5 ;tmp1 = 1/2 + ( log[base] (a) + 1) / 2= log
[base] (a) / 2 + 1
exp result, tmp1, 0x1F, 0x1 ;result = base ^ ( log[base] (a) / 2 + 1 -
1 ) = base ^ ( log[base] (a) / 2 )= base ^ ( log[base] ( a^(1 / 2) ) =
a ^ (1/2)
```

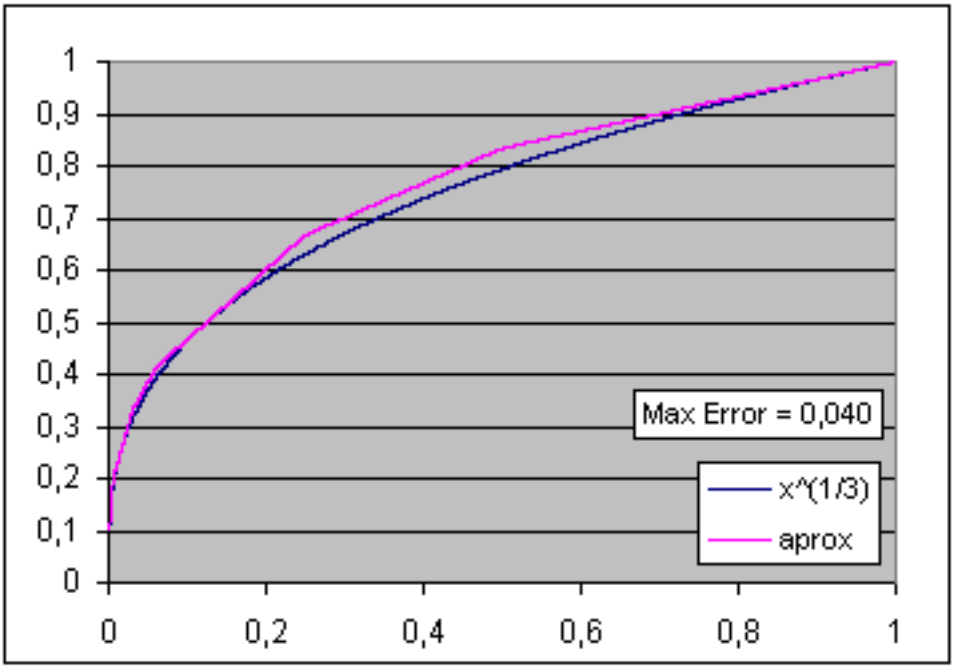
As we can see in the graph, this algorihtm gives a very good aproximation to the square root:



Cubic root approximation:

```
log tmp1, a, 0x1F, 0x1 ;tmp1 = log[base] (a) + 1
macs tmp1, 0.6666666666, tmp1, 0.3333333333 ;tmp1 = 2 /3 + ( log[base]
(a) + 1 ) / 3
;= log[base] (a) / 3 + 1
exp result, tmp1, 0x1F, 0x1 ;result = base ^ ( log[base] (a) / 3 + 1 -
1 )
;= base ^ ( log[base] (a) / 3 )
;= base ^ ( log[base] ( a^(1 / 3) ) = a ^ (1/3)
```

Wors aproximation to the cubic root than to the square root, but still very good.

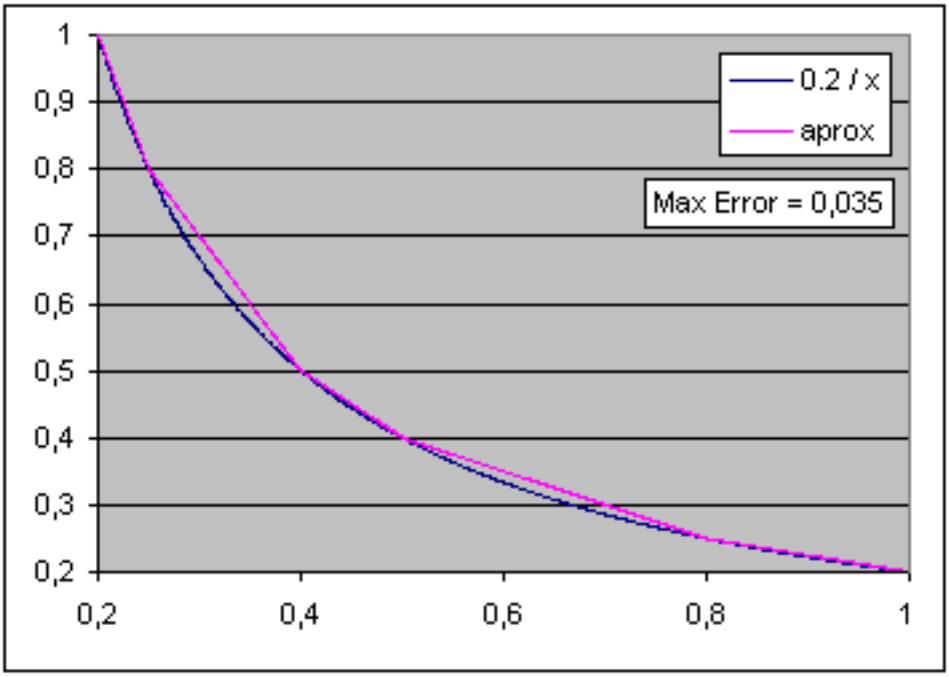


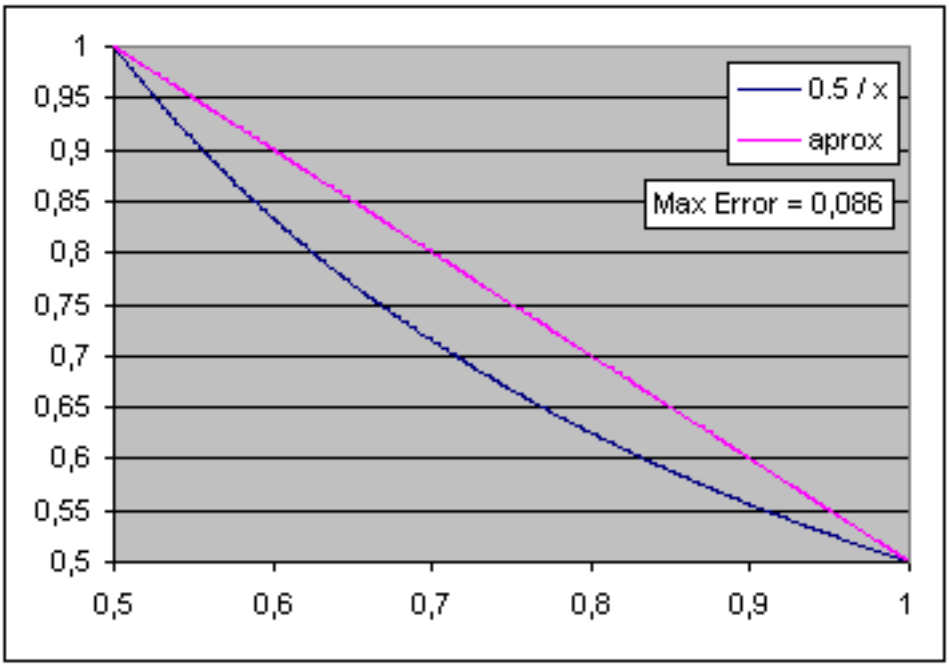
Division approximation with an unsigned result:

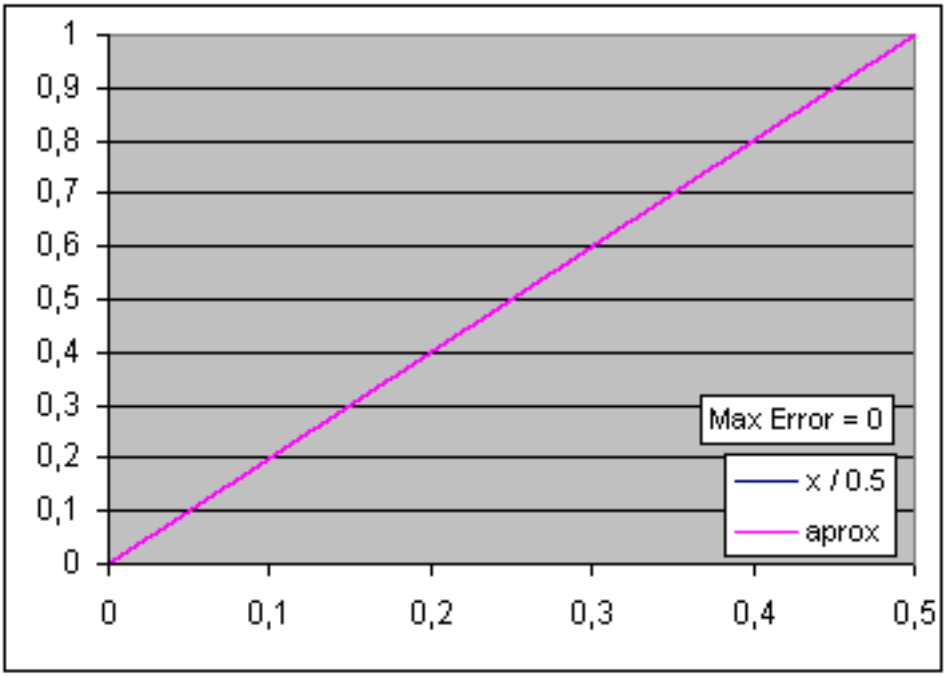
Based on the following formula:

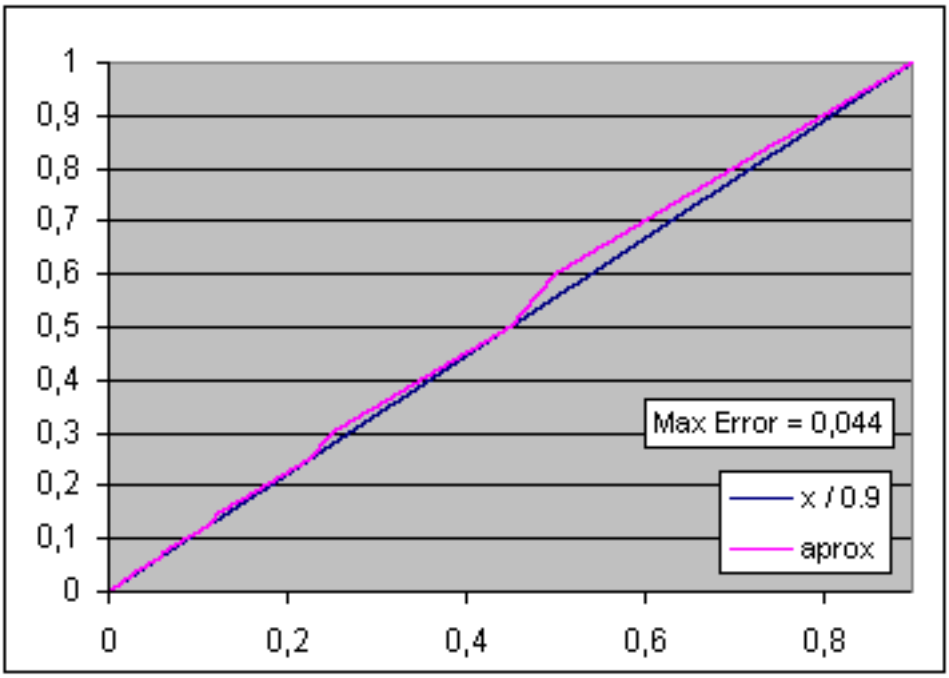
a / b = exp (log(a / b)) = exp (log(a) - log(b))

```
log tmp1, a, 0x1F, 0x1 ;tmp1 = log[base] (a) + 1
log tmp2, b, 0x1F, 0x1 ;tmp2 = log[base] (b) + 1
macsn tmp2, tmp2, 1, 1 ;tmp2 = log[base] (b)
macsn tmp3, tmp1, tmp2, 1 ;tmp3 = log[base] (a) - log[base] (b) + 1
exp result, tmp3, 0x1F, 0x1 ;result = base ^ ( log[base] (a) - log
[base] (b) + 1 - 1 )
;= base ^ ( log[base] (a) - log[base] (b) )
;= base ^ ( log[base] ( a / b ) ) = a / b
```


1
0,9
0,8
0,7
0,6
0,5
0,4
0,3
0,2
0,2
0,4
0,6
0,8
1
0.2 / x
aprox
Max Error = 0,035


1
0,95
0,9
0,85
0,8
0,75
0,7
0,65
0,6
0,55
0,5
0,5
0,6
0,7
0,8
0,9
1
0.5 / x
aprox
Max Error = 0,086





Observations:

-We can't store a fractional number greater than 1.0, so the operand **b** must be greater than operand **a** in order to get a result smaller than 1.0.
-We must take the sign = 0x1, since logarithm of a negative number is not defined, and the mathematical formula would not be valid. This restricts the result to be allways positive (the operands still can be positive or negative).
-The above restriction can be solved adding two more lines to the algorithm (see next paragraph).
-The precission of this algorithm oscillates a lot, depending on the value of the two operands (see above graphs) . The maximum error is 0.086.

Division approximation with a signed result:

```
macints tmp_sign, 0, a, b ;tmp_sign = +1, if a and b has the same sign.
;tmp_sign = -1, if a and b has opposite signs.
```

```
limitl tmp_sign, tmp_sign, 0x3, 0x1 ;If tmp_sign = +1 Then tmp_sign =
0x1
;If tmp_sign = -1 Then tmp_sign = 0x3
log tmp1, a, 0x1F, 0x1 ;tmp1 = log[base] (a) + 1
log tmp2, b, 0x1F, 0x1 ;tmp2 = log[base] (b) + 1
macsn tmp2, tmp2, 1, 1 ;tmp2 = log[base] (b)
macsn tmp3, tmp1, tmp2, 1 ;tmp3 = log[base] (a) - log[base] (b) + 1
exp result, tmp3, 0x1F, tmp_sign ;result = base ^ ( log[base] (a) - log
[base] (b) + 1 - 1 )
;= base ^ ( log[base] (a) - log[base] (b) )
;= base ^ ( log[base] ( a / b ) ) = a / b
```

9. **INTERP** - this instruction performs linear interpolation between two points. The main use of this operation is for simple single instruction low-pass filter for reverberation and other not very demanding filtering tasks. It is also useful for wet/dry signal mixing or pan control, a simple high-pass filter, envelopes, waveshaping and many other things.

Formula:

INTERP R = (1 - X) * A + X * Y

Example 1:

This is very useful for one instruction one-pole low-pass filter, which has the following formula:

out = coef * in + (1 - coef) * out
where coef is the filter coefficient.

The coefficient can be calculated with the following formula:
coef = 2 – cos(2*pi*(Fc/Fs)) – sqrt((cos(2*pi*(Fc/Fs)) – 2)^2 – 1)
where Fc is the cutoff frequency, Fs is the sampling frequency (which is fixed at 48000Hz for the Emu10kx signal processors), pi is 3,14 which we know from pre-highschool math. In this function the only parameter is the cutoff frequency. This is the frequency at which the frequency range is cut and all frequencies above the cutoff will be removed (filtered), all below – passed.

```
input in
output out
control filter=0.5

interp out, out, filter, in ;the value of out is preserved for the next
sampe cycle
end ;thus forming a one sample delay line
```

We can also utilize this instruction for a simple high-pass filter:

```
input in
output out
control filter=0.5
static st

interp st, st, filter, in
macsn out, in, st, 1
end
```

Example 2:

We can combine the low-pass filter with the soft clipping log instruction.

```
input in
output out
static exponent=0.35
control volume=0.2, filter=0.5
temp t

log t, in, exponent, 0x0 ;log soft clipping
macs t, 0, t, volume ;volume control
interp out, out, filter, t ;low-pass filter
end
```

We get a simple soft clipping fuzz/overdrive effect.

10. The **SKIP** instruction provides flow control in the dsp code. It skips a given number of instructions under certain conditions, making use of the already mentioned CCR.

SKIP R, CCR, TEST_VALUE, number of instructions to skip

TEST_VALUE is a register containing a value which indicates under which conditions to skip.
CCR is the address of the CCR register. This can be any register, thus a previously saved CCR can be reused, or a constant can be used to implement an always skip (a NOP).

The CCR is set after each instruction based upon the result of the instruction (R operand). Thus, to make a skip into our program we need two instructions. The first can be any instruction that sets the value to be tested. The second is the skip instruction which tests the value, and sets the amount of skipped instructions. So, simply said, the result of the instruction before the skip instruction is tested by the skip instruction, and based on it a given number of instructions below the skip one are omited.

These are the most common cases and they are used for almost all situations:

```
skip    ccr, ccr, 0x8, n ;skip n number of instruction if previous
```

```
result is =0

  skip    ccr, ccr, 0x100, n ;if previous result is !=0 (different from
0)

  skip    ccr, ccr, 0x4, n ;<0

  skip    ccr, ccr, 0x1008, n ;<=0 (less or equal to 0)

  skip    ccr, ccr, 0x180, n ;>0

  skip    ccr, ccr, 0x80, n ;=>0 (equal or more than 0)

  skip    ccr, ccr, 0x10, n ;on saturation

  skip    ccr, ccr, 0x7FFFFFFF, n ;skip always
```

Delay lines

They are a necessary part of effects like delay, echo, reverb, chorus and many others.
For complete information on delay lines look at the As10k1 manual and Dane help (in the kX help file). Here I'll discuss the practical side only.
In Dane we declare delay lines by specifying the number of samples in internal (on chip) or external memory (ram) . Remember that 48000 samples form 1 second of audio data, so a delay line of 48000 will give us 1 second of delay.
We declare delay lines like this:

```
itramsize 6000 ;(0.125sec delay)
```
or
```
xtramsize 12000 ;(0.25 sec of delay)
```

We declare delay data registers like that:

```
idelay write wr at 0 ;wr is just a name;the write register is generally
at 0
idelay read rd at 12000 ;rd is just a name
```
or
```
xdelay write... etc.
```

We can have as many read and write registers as the dsp has and they should not overlap. We can also have several read registers following one write register, so we have one point at which data is entered into the delay line and many points at which it is read/extracted. This is the best way to think of delay lines and their registers – as points. The further one read point is away from the write point, the more delayed the data at it will be.

Example:
Simplest static 0.25 sec delay.

```
input in
output out
xtramsize 12000 ;we declare the number of samples(the delay line)

xdelay write wr at 0 ;write data register at 0
xdelay read rd at 12000 ;read data register at 12000

macs wr, in, 0, 0 ;we write the value of in to wr
macs out, rd, in, 1 ;we mix the input with the 0.25 sec delayed input
in rd
end
```

Example2:
Now we will make a feedback echo with a damping filter.

```
input in
output out
control filter, feedback

xtramsize 12000

xdelay write wr at 0
xdelay read rd at 12000

macs out, in, rd, feedback
interp wr, out, filter, wr ;we re-use the output register
end
```

Some simple effects "disection" and explanation

Let's start simple.

1.Stereo Mix

```
; Registers
input in1L, in1R, in2L, in2R;
output outL, outR;
control In1 Level=0x0, In2 Level=0x0;
temp tmp

; Code
macs tmp, 0x0, in1L, In1 Level; ;tmp = 0 + in1L * In1 Level
```

```
;this is a level control for the left input of channel 1

macs outL, tmp, in2L, In2 Level;;oulL = tmp + in2L * In2 Level
;here we mix the left input of channel 1 with the left input of channel
2
; + level control for the left input of channel 2

macs tmp, 0x0, in1R, In1 Level;
;level control for the right input of channel 1

macs outR, tmp, in2R, In2 Level
;here we mix the right input of channel 1 with the right input of
channel 2
;+ level control for the right input of channel 2


end
```

It is as simple as that. For more input channels the concept is the same - we mix all left channels to the output left channel and all right channels to the output right channel.

2.Delay Old

```
; Registers
input in;
output out;
control level=0x7fffffff, feedback=0x40000000, delay=0x1c20000;


xtramsize 14400 ;we declare external tram size (number of samples)
; External TRAM delay line (14400 samples; ~0.300000 msec)

xdelay write wrt at 0x0; ;write data register
xdelay read rd at 0x0; ;read data register



; Code
acc3 &rd, delay, &wrt, 0x0;
;here we produce the delay time by offsetting the rd addres register
;by the value of "delay", which is user controllable. We use the wrt
register as
;the initial point.
;We acces the adress register with "&rd"


macs out, in, level, rd;
;out = in + level * rd
;here we mix the input with the delayed rd with level control for rd

macs wrt, in, rd, feedback;
;wrt = in + rd * feedback
;here we create the feedback amount


end
```

3. EQ Lowpass, Bandpass, Highpass, Notch

The example is a biquadratic (aka biquad) lowpass, but all four are basically the same, olny the coefficients are different. Although better methods exist, we'll use this one because of it's simplicity.

NOTE: The values of the registers containing the filter coefficients are C++ controlled. You can't control them with faders created by Dane only by copying the code. For our example we'll assume that they are static and not user controllable.

These filters are IIR (Infinite Impulse Response) and their difference equation is:

y[n] = b0*x[n] + b1*x[n-1] + b2*x[n-2]
- a1*y[n-1] - a2*y[n-2]

x[n]= input
y[n]=output
x[n-1] = delayed input by one sample
x[n-2] = delayed input by two samples
y[n-1] = delayed output by one sample
y[n-2] = delayed output by two samples

Because the effect is stereo there are actually two filters, so we can get rid of the second (right channel) part, because it is the same thing. Of course we're doing this for learning purposes, if we want the effect to remain stereo we have to use one filter for each channel – left and right.

NOTE:. Remember that the values of static registers are preserved for the next sample cycle and can be reused until they are overwritten. Thus they form a delay line from which we get the delayed samples for the above formula.

```
; Registers
input inl ; inr
output outl ; outr
static b0=0x1b40d9, b1=0x3681b2, b2=0x1b40d9; filter coefficients
static a1=0x7b4cfa5d, a2=0xc446023e, sca=0x2;
static lx1=0x0, lx2=0x0, ly1=0x0;static registers for the delayed
samples
static ly2=0x0 ; rx1=0x0, rx2=0x0; we don't need this
;static ry1=0x0, ry2=0x0; and this, because they are for the right
cahnnel
temp t1, t2

; Code
macs 0x0, 0x0, 0x0, 0x0;
;we null the accumulator (note that the result of that operation is 0)
;so previous values won't get mixed with the ones we need

macmv lx2, lx1, lx2, b2;
;b2*x[n-2]
macmv lx1, inl, lx1, b1;
;b1*x[n-1]
```

```
macmv t1, t1, inl, b0;
;b0*x[n]
macmv ly2, ly1, ly2, a2;
;a2*y[n-2]
macmv t1, t1, ly1, a1;
;a1*y[n-1]
macs t2, accum, 0x0, 0x0;
;we copy the value of the accumulator to t2, so we can use it
macints ly1, 0x0, t2, sca;
;we scale it up by 2, because the coefficients have been initially
scaled down
;to prevent overflow
macs outl, ly1, 0x0, 0x0;
;we copy everything to the output
end
;we get rid of the second part, because it's the same as the first
; macs 0x0, 0x0, 0x0, 0x0;
; macmv rx2, rx1, rx2, b2;
; macmv rx1, inr, rx1, b1;
; macmv t1, t1, inr, b0;
; macmv ry2, ry1, ry2, a2;
; macmv t1, t1, ry1, a1;
; macs t2, accum, 0x0, 0x0;
; macints ry1, 0x0, t2, sca;
; macs outr, ry1, 0x0, 0x0;
```

That's all for now. I hope this guide helps you start programming dsp effects. Ater you learn the basics, you might want to start studying more complex kX effects and search for dsp code on the web, which you can port to the kX environment.

Feel free to ask anything related to digital audio.
You can send me an e-mail - martintiger@abv.bg, or start a thread on the kX forum. I'm also open for suggestions, new ideas, criticism, and of course, if anyone finds something that's wrong in this document, please contact me.

Have fun with DSP.