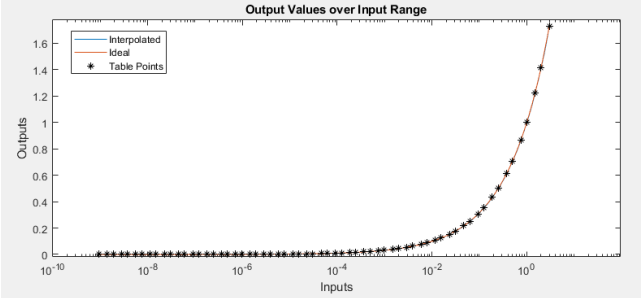
1.27.2020 Programmable LUT Final Design (closer look)

Thursday, September 19, 2019 4:30 PM

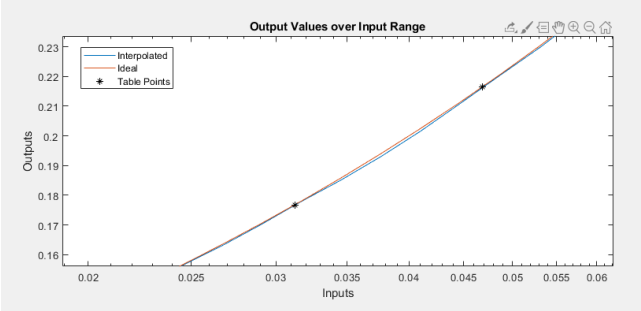
Record important design or data

- You should capture the following
  - Relationship to overall project
    Simplified the design to a self-contained block, working on testing and verification.
  - Requirement X: _____

  - Technical progress



Showing example function and table points



Showing example expected error between ideal and table output

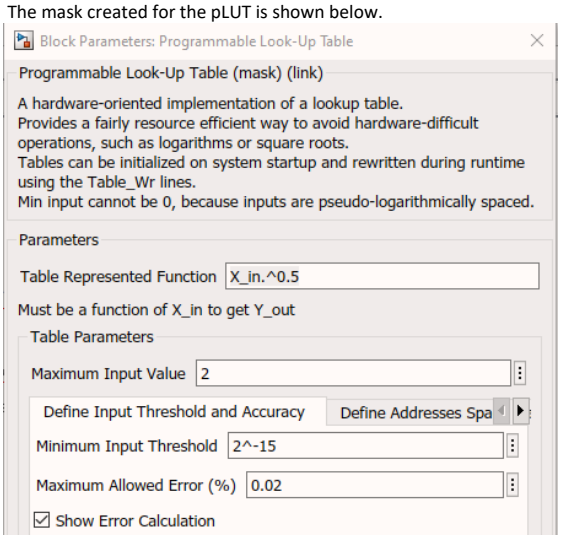
  - Next action items specific to the task you are working on

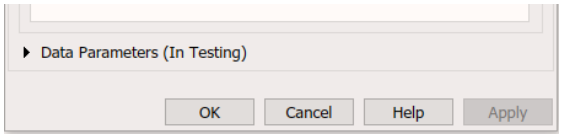
The purpose of this pLUT is to estimate a given function within a set degree of accuracy. A table will then be generated, along with logic to identify an input and choose the two surrounding pre-calculated table points and find an answer through linear interpolation between those two points.

In order to maximize accuracy for transcendental functions, the precalculated input values are pseudo logarithmically spaced. The input range is defined as a set of data points from some 2^a to 2^b. Note this cannot include 0, or negative input values.

This tech entry will focus on the math behind the logic, and how it all works together.

The mask created for the pLUT is shown below.





Let's move step by step.

Defining a range of Inputs

First, a set of inputs must be defined for the table to be based on.
By default, this will be determined automatically using the "Define Input Threshold and Accuracy" tab. A manual override can be set instead, but let's ignore that for now. To make calculations simpler, powers of 2 will be used to define the input bounds. Powers of 2 allow for a few tricks in binary, shown later.
'd' is defined as the smallest power of 2 greater than or equal to the Maximum Input Value. Effectively, 'd' will be used to define a hardware-optimized bound for the table's inputs.

```
d = ceil(log2(maxInput));
```

ceil rounds the input within it up to the next integer, and log2 finds the result of $\log_2(x)$.
Next, N_bits and M_bits are defined. N_bits will represent the number of bits in the table's addresses used to define input data range. A larger number of N_bits allows for smaller numbers to be included within the lookup table. M_bits represents the number of bits in the table's addresses used to increase precision between $\log_2$ spaced points. A higher number of M_bits increases the accuracy of the table's estimate.

In this example, d = 1, N_bits = 4, M_bits = 1. This is enough space to satisfy the max allowed error 0.02% of any input between 2^-14 (very small number) and 2^1 (2), as shown in the two graphs at the top of this entry.

Quick Explanation of Binary and Fixed-Point Numbers

To understand how this table's inputs can be "logarithmically spaced", we must first review fixed-point numbers in binary. Counting in binary is similar to counting in base 10. A sequence of numbers can be created in order, {00,01,02,03...09}. After reaching 09, the smallest digit becomes a 0, and the next highest digit is incremented, resulting in 10. With binary, there are only 1s and 0s, so counting to 10 in base 10 looks like this:
Binary: {0000, 0001, 0010, 0011, 0100, … 1010}
Base 10:{ 0, 1, 2, 3, 4, … 10}

Each digit in binary represents whether or not to include a power of 2. I.e. {8s, 4s, 2s, 1s}, similar to base 10's system of {1000s, 100s, 10s, 1s}.
Decimals in binary are represented in a similar fashion, but rather than each number beyond the decimal representing tenths or hundredths, they represent the 1/2s or 1/4ths place.

3.75 would look like 011.110, 2+1+½+¼
2.25 would look like 010.010, 2+0+0+¼
1.25 would look like 001.010, 0+1+0+¼

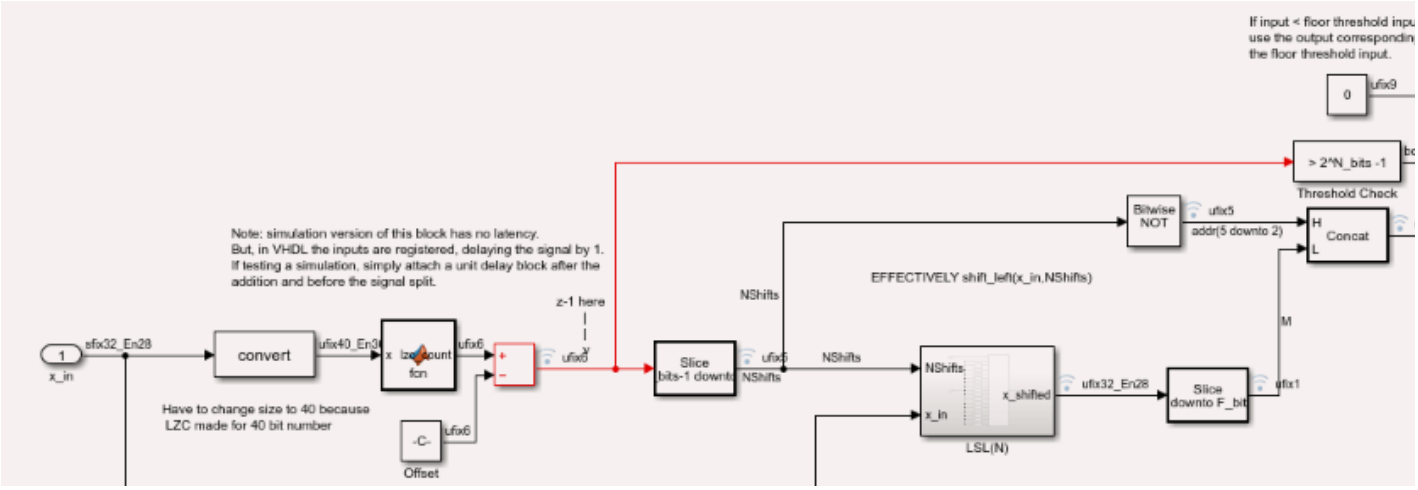
For convenience, the highest position of a binary number often represents it's sign. For now, let's assume all numbers are *unsigned*, or always positive.

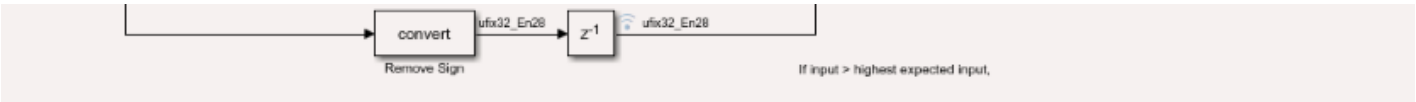
A fixed-point number is simply a binary number where there is a set number of digits, and a defined location for the decimal point (called "radix point" in binary). In the design, the word size is defaulted to 32 bits, with 28 of those bits representing fractional numbers. This provides a high amount of precision, so long as the number never reaches about 8, or goes below –8. Since the design works with an audio codec, where inputs are bound between –1 and 1, this far more than covers the range of expected inputs, but for some calculations the size or amount of fractional bits may need to be adjusted.

Back to the design.

Logarithmic Spacing of Table Inputs

With this understanding of binary, we can use a few tricks to "logarithmically" space a look-up table.





Once an input is received (always treated as unsigned by going through the "convert" blocks), the signal is split into two paths. The upper paths counts the number of "Leading Zeros". Effectively, this identifies the highest value "1" of the fixed point string. Keep in mind 4 digits are used for non-fractional bits and 28 digits are used for fractional bits. Only the first 4 will be shown for simplicity.

For example, given 0.75 or 0000.1100... , the number of leading zeros is 4.
Given 0.125, or 0000.0010... , the number of leading zeros is 6.

This is number is then subtracted by an offset. "W_bits-1-F_bits-d". The derivation and full explanation of 'd', the effective upper input bound, can be seen above. This integer value is then sent to two locations. We will come back to the upper path in a moment. The "Slice" block copies the integer, but only certain bits. While the text is cut off, bits "N_bits-1" down to 0 are read. This effectively limits the number between 0 and (2^N_bits) -1. We now know Nshifts, which is the power of 2 "x_in" must be multiplied by to be equal or greater than 2^d.

This sliced integer is sent up to the "Bitwise NOT" block, which flips the bits of the integer as represented in binary. More on this later.

The original input signal "x_in" is then rotated left by Nshifts. This is one of the tricks of binary, rotating bits can be used to multiply or divide by a power of 2 with minimal calculations needed.
For example:

000.100 = ½
001.000 = 1
000.010 = ¼

This rotation forces the highest 1 within "x_in" to move to the same location every time, so long as "x_in" is within the predefined boundaries.
For example:

x_in = 0.75, 00.110; NShifts = 2, x_shifted = 011.000 = 3
x_in= 0.5, 00.100; NShifts = 2, x_shifted = 010.000 = 2
x_in= 1.0, 01.000; NShifts = 1, x_shifted = 010.000 = 2

Note that x_shifted is always greater than or equal to 2^d.
Another bit slice occurs on x_shifted, taking bits (F_bits-1+d) down to (F_bits-M_bits+d). Effectively, this grabs "M_bits" bits from below the guaranteed 1. In this example, this only grabs the bit representing 1's in x_shifted.

Signals M and NOT(NShifts) are concatenated, resulting in the address used to read from the tables. The farthest upper path, mentioned earlier, is used as a check to see if the input is below the minimum value of x_in recognized by the table, and if it is, will use return address 0 instead.

Full example of address definitions:

x_in = 0.75,00.110...	NShifts = 2	x_shifted = 011.000 = 3	M = 1	NOT(NShifts) = 1101	addr_r = 11011
x_in = 0.5, 00.100...	NShifts = 2	x_shifted = 010.000 = 2	M = 0	NOT(NShifts) = 1101	addr_r = 11010
x_in = 1.0, 01.000...	NShifts = 1	x_shifted = 010.000 = 2	M = 0	NOT(NShifts) = 1110	addr_r = 11100
x_in = 2^-14,00. … 10 …	NShifts = 15	x_shifted = 010.000 = 2	M = 0	NOT(NShifts) = 0000	addr_r = 00000

This shows that there are an equal amount of entries in the table between any power of 2, from minimum input to maximum input, providing for maximum accuracy of any size number allowed when using transcendental functions.

The most important takeaways here are that an **increase of a power of 2 increases NOT(NShifts) by 1**, and that **all entries in the table are in order from least to greatest**.