Algorithmic C (AC) Math Library Reference Manual

Software Version v1.0

March 2018

Table of Contents

Section 1. Introduction

The Algorithmic C Math Library (ac_math) contains synthesizable C++ functions commonly used in Digital Signal Processing applications. The functions use the Algorithmic C data types and are meant to serve as examples on how to write parameterized models and to facilitate migrating an algorithm from using floating-point to fixed-point arithmetic where the math functions either need to be computed dynamically or via lookup tables or piecewise linear approximations.

The input and output arguments of the math functions are parameterized so that arithmetic may be performed at the desired fixed point precision and provide a high degree of flexibility on the area/performance trade-off of hardware implementations obtained during Catapult synthesis.

The hardware implementations produced by Catapult on the math functions are bit accurate. Simulation of the RTL can thus be easily compared to the C++ simulation of the algorithm.

1.1. Types of Approximations

The following sections discuss the calculation methods used by the functions in the ac_math library and the trade-offs of using one method over another in your design.

1.1.1. Piecewise Linear

Some of the functions available in the ac_math library are implemented as piecewise linear approximations. <explain basics of PWL, reference example like ac_reciprocal_pwl() perhaps with a plot of actual vs pwl, briefly discuss method used to construct PWL model, errors in the approximation and warning about cascaded error due to chaining PWL function calls>

1.1.2. Lookup Table (LUT)

Some of the functions can be efficiently implemented as a lookup table. For example, the sine and cosine functions can be defined as a lookup table where the symmetry of the functions can be exploited to minimize the size of the table. <fill in more detail, especially magnitude of error>

1.1.3. CORDIC

Some of the hyperbolic and trigonometric functions have implementations based on the CORDIC algorithm. These implementations use an iterative approach that typically converges with one digit per iteration. The iterations may involve addition, subtraction, bit shifts and table lookups. Given the iterative nature of these implementations the resulting hardware will be larger and/or slower than the PWL or LUT implementations but offer the best accuracy.

1.2. Installing the ac_math library

The library consists of three directories, shown here:

```
.
|-- include
|   |-- ac_math
|   |   |-- ac_abs.h
|   |   |-- ac_inverse_sqrt_pwl.h
|   |   |-- ac_log_pwl.h
|   |   |-- ac_math_ns.h
|   |   |-- ac_normalize.h
|   |   |-- ac_pow_pwl.h
|   |   |-- ac_reciprocal_pwl.h
|   |   |-- ac_shift.h
|   |   |-- ac_sincos.h
|   |   `-- ac_sqrt_pwl.h
|   `-- ac_math.h
|-- pdfdocs
|   `-- ac_math_ref.pdf
`-- tests
    |-- Makefile
    |-- rtest_ac_inverse_sqrt_pwl.cpp
    |-- rtest_ac_log2_pwl.cpp
    |-- rtest_ac_log_pwl.cpp
    |-- rtest_ac_normalize.cpp
    |-- rtest_ac_pow_pwl.cpp
    |-- rtest_ac_reciprocal_pwl.cpp
    |-- rtest_ac_sincos_lut.cpp
    `-- rtest_ac_sqrt_pwl.cpp
```

In order to utilize this library you must have the AC Datatypes package installed and configure your software environment to provide the path to the AC Datatypes “include” directory as part of your C++ compilation arguments.

The ac_math library includes a series of unit tests that exercise each of the approximation functions across various fixed-point bit widths to ensure that the accuracy of the approximation is within a certain tolerance of the standard C++ math library equivalent (under the same input and output bit-width constraints). To exercise these tests from a Linux shell, use the following GNU make command line:

```
gmake all AC_TYPES_INC=<path to AC Datatypes include directory>
```

where the variable AC_TYPES_INC specifies the path to the install location of the AC Datatypes package. The results of the tests look something like this:

```
TEST: ac_inverse_sqrt_pwl()  INPUT: ac_float< 5, 3, 3, AC_RND>  OUTPUT: ac_float<64,32,10, AC_RND>   RESULT: PASSED , max error (0.083916)
TEST: ac_inverse_sqrt_pwl()  INPUT: ac_float< 5, 1, 3, AC_RND>  OUTPUT: ac_float<64,32,10, AC_RND>   RESULT: PASSED , max error (0.083916)
TEST: ac_inverse_sqrt_pwl()  INPUT: ac_float< 5, 0, 3, AC_RND>  OUTPUT: ac_float<64,32,10, AC_RND>   RESULT: PASSED , max error (0.083916)
TEST: ac_inverse_sqrt_pwl()  INPUT: ac_float< 5,-2, 3, AC_RND>  OUTPUT: ac_float<64,32,10, AC_RND>   RESULT: PASSED , max error (0.083916)
TEST: ac_inverse_sqrt_pwl()  INPUT: ac_float< 5, 9, 3, AC_RND>  OUTPUT: ac_float<60,30,11, AC_RND>   RESULT: PASSED , max error (0.083916)
```

The output shows the function under test, the input and output bit-widths and the maximum error observed over the tested range of minimum and maximum values expressible in the input type stepping by the smallest value expressible in the input type.

1.3. Using the ac_math library

In order to utilize any of the math functions, add the following include line to the source:

```
#include <ac_math.h>
```

Section 2. Piecewise Linear Functions

The *ac_math* package includes the following piecewise linear functions:

- Logarithm (ac_log_pwl)
- Power (ac_pow_pwl)
- Reciprocal (ac_reciprocal_pwl)
- Square Root (ac_sqrt_pwl)

The following subsections describes the implementation and usage of these functions in more detail.

2.1. Logarithm (ac_log_pwl)

The *ac_log_pwl* is piecewise linear implementation of logarithmic function, optimized to provide high performance with quick results. This function is implemented for ac_fixed datatypes and can take unsigned integers and fractional values. It provides great accuracy and it is observed that it is faster than other algorithms and consumes significantly lesser area. These qualities make it useful as a basic building block in high speed IP blocks.

2.1.1. The ac_log_pwl Implementation

The header file provides piecewise linear implementation of log base 2 via function ac_log2_pwl. On the other hand, log base e is computed using the change of base property. It is possible to convert other logarithmic computations to log base 2 piecewise linear implementation, by using change of base property, which is given by,

$$\log_b x = \frac{\log_a x}{\log_a b}$$

Note that, it will require one more additional multiplier/divider. Log base e can be also synthesized in piecewise manner, though after experimentation and comparison it was found out that above algorithm provides better algorithm than the piecewise linear version.

2.1.2. Basic Algorithm

Before PWL implementation the input value is first given to the normalization function, which converts the input value into an exponent and a normalized value, the normalized value lies between 0.5-1 and exponent is a signed integer. This value is then subjected to the 9 points/8 segments piecewise linear implementation. After experimenting with multiple combinations, it was found that this combination provides a good QofR. Although user can replace the c and m values by generating his/her own version of piecewise linear implementation that will provide higher/lesser accuracy with higher/lesser ROM sizes based on his/her requirement. In other words, accuracy is directly proportional to the ROM size. LUT generator is included in the package which further explains procedure of doing that.

Since log of input is nothing but addition of log of normalized value and the exponent

```
( log(a*(b^c)) = log a + c log b
```

with b =2: log a + c log 2 = log a + c, for base 2---- equation

we get the result by using adder in the end. '1/Log base 2 e' is defined as constant in the beginning, in computation of log base e / natural log.

Lookup Table (LUT) Initialization

Lookup table values are computed using the LUT generator code provided in the package and user can replace them to change the accuracy and area based on his/her requirement. Lookup table values are defined as const values to make sure that they are synthesized as ROM values. 9 points/8 segments combination is used in this implementation which provides really good accuracy. ROM values are given total 11 fraction bits. This is computed using maximum LUT error, which as observed from the testbench code in LUT generator is 0.00125113.

Then log2 of this value is taken which is, -9.64259 and absolute value is 10. Also, we need to take additional addition bit due to mx+c addition and hence making total number of bits = 11.

LUT generator is non-synthesizable C++ code which writes values in a separate text file. Note that in LUT generator ROM points are taken a little below the curve, using segment midpoint shifting method. In this method, segments are stretched downwards by computing the maximum error and then adjusting the segment position such that this maximum error is minimized. This improves precision even further.

PWL Implementation

The following illustration compares the log base 2 function of MATLAB against pwl implementation of same (number of segments = 8, range = [0.5,1])

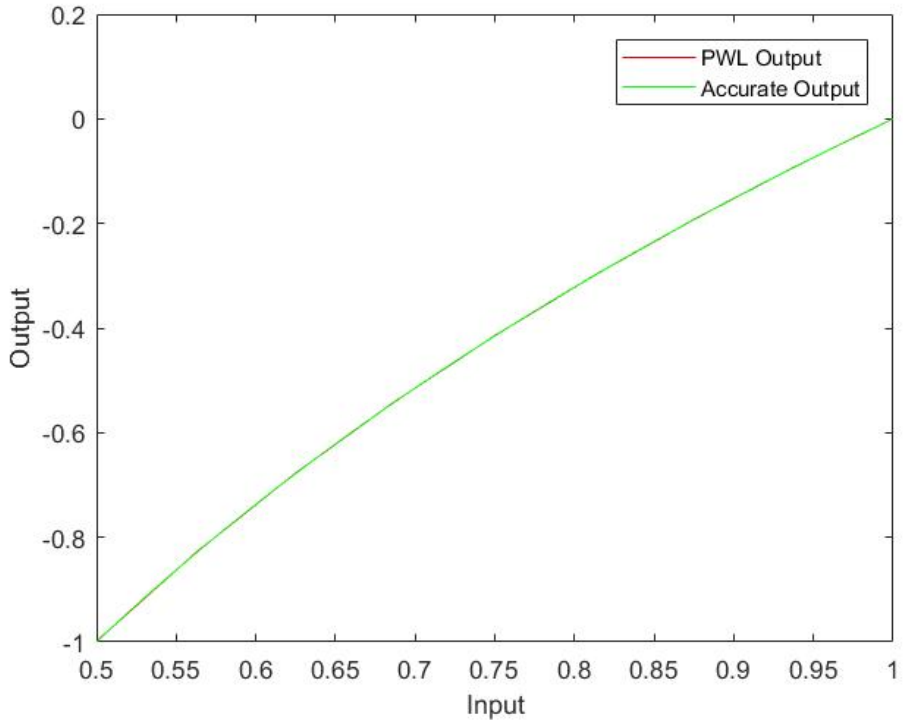


Figure 1. Comparison between log base 2 function of MATLAB and pwl implementation of same (number of segments = 8, range = [0.5,1])

As seen in the above graph, the actual implementation of the logarithmic function and its piecewise linear implementation version are significantly closer to each other. From the structure of the graph, it is observed that the range of 0.5-1 is enough to reflect the overall structure of the graph and hence in PWL implementation this range is used.

This range is then divided into 8 segments that are equally spaced from each other. Value of input is then converted into scaled input, whose integer part is used for determining index from the ROM.

As mentioned in the basic algorithm section, after log value of normalized value is computed, the exponent is then added in computation of log base 2 value of the input, which is our final output.

The maximum error was 0.19% in log base 2 function, for input range of (0,1234] with step of 0.05. After varying the input bitwidths and input values it was found out that error always stayed below 1%.

Note that function can also take negative integer bit widths as input and is also capable of handling total bitwidth < integer bitwidth of input scenario. Also, note in the below function headers that rounding of PWL output can be supplied as a template parameter with default parameter value as AC_RND. Function only take the unsigned ac_fixed input.

Note that, functions also have version that can return the output, rather than writing it at memory locations. This although, requires user to supply output_type as a template parameter. Log of zero returns minimum negative value (signed value with all bits 1).

2.1.3. Natural log or log base e handling

Natural log or log base e handling is done by using change of base property. As mentioned in the introduction section, log base e can be obtained from log base 2 in the following manner:

```
Log base e = log base 2 * constant
```

This constant is basically, 1/log base 2(e), hence this is internally defined as static const with precision of 12 bits. This precision is based on the internal precision of the log base 2 algorithm, which is given by 11 (internal precision of m and c values) + 1 (to take into account mx+c, addition component).

2.1.4. Function headers

As explained earlier, functions log base 2 and log base e have different pwl implementations and headers of them are given as follows:

- ```
  template <ac_q_mode q_mode_temp = AC_RND, int W1, int I1, ac_q_mode
  q_mode_in, ac_o_mode o_mode_in, int W2, int I2, ac_q_mode q_mode_out,
  ac_o_mode o_mode_out>
  void ac_log2_pwl (const ac_fixed <W1, I1, false, q_mode_in,
  o_mode_in> input, ac_fixed <W2, I2, true, q_mode_out, o_mode_out>
  &result);
  ```
- ```
  template< ac_q_mode q_mode_temp = AC_RND, int W1, int I1, ac_q_mode
  q_mode_in, ac_o_mode o_mode_in, int W2, int I2, ac_q_mode q_mode_out,
  ac_o_mode o_mode_out >
  void ac_log_pwl(const ac_fixed <W1, I1, false, q_mode_in, o_mode_in>
  input, ac_fixed <W2, I2, true, q_mode_out, o_mode_out> &result);
  ```

Function versions that return the output and can be used to assign it to output variable:

- `template<class T_out, ac_q_mode q_mode_temp = AC_RND, class T_in> T_out ac_log2_pwl (const T_in &input)`
- `template<class T_out, ac_q_mode q_mode_temp = AC_RND, class T_in> T_out ac_log_pwl (const T_in &input)`

2.1.5. Example Function Call

Following is the example call for ac_log2_pwl function that returns log base 2 of input.

```
typedef ac_fixed<32, 16, false, AC_RND, AC_SAT> input_type;
typedef ac_fixed<20, 8, true, AC_RND, AC_SAT> output_type;
input_type x = 1.75;
output_type y;

ac_log2_pwl (x, y)  //Returns y = log2(x)

or

y = ac_log2_pwl <output_type> (x); //Returns log2(x)
```

Following is the example call for ac_log_pwl function that returns log base e of input.

```
typedef ac_fixed<32, 16, false, AC_RND, AC_SAT> input_type;
typedef ac_fixed<20, 8, true, AC_RND, AC_SAT> output_type;
input_type x = 1.75;
output_type y;

ac_log_pwl (x, y)  //Returns y = loge(x)

or

y = ac_log_pwl <output_type> ac_log_pwl(x); //returns ln(x)
```

Note that, both the functions will throw function mismatch error, when signedness isn't maintained as shown above and maximum negative value is returned at the output if 0 is supplied as the input to the above functions.

2.2. Power (ac_pow_pwl)

The *ac_pow_pwl* function is designed to provide a quick approximation of the base 2 and natural exponentials of real numbers using a piecewise linear (PWL) implementation with 4 points/3 segments, with high accuracy. This frees us of the burden of having to calculate a more accurate output using methods such as Taylor series expansion, which requires loop unrolling, a large number of adders and hence a large area, to give 100% throughput.

2.2.1. The ac_pow2_pwl Implementation

The *ac_pow_pwl* library provides two functions, one which calculates the base 2 exponential, and the other which calculates the natural exponential. The natural exponential version (henceforth called the *ac_exp_pwl*

implementation) depends upon the base 2 exponential version (henceforth called the *ac_pow2_pwl* implementation) for its computation, as will be explained later. Both functions only accept *real*, *ac_fixed* inputs and calculates *ac_fixed* outputs.

Normalization and De-normalization

The input to the exponentiation function is first normalized to a range of [0, 1). It is this normalized value that our PWL approximation produces an output corresponding to. The output is then “de-normalized”, i.e. the effects of the initial normalization are cancelled out to get the approximate output for the input supplied to our function.

PWL Approximation

To explain the closeness of the PWL approximation to the actual, accurate implementation of the base 2 exponential function, the following graph compares the PWL output against. the accurate function output. The graph values use double precision floating point values for the PWL computation in MATLAB.

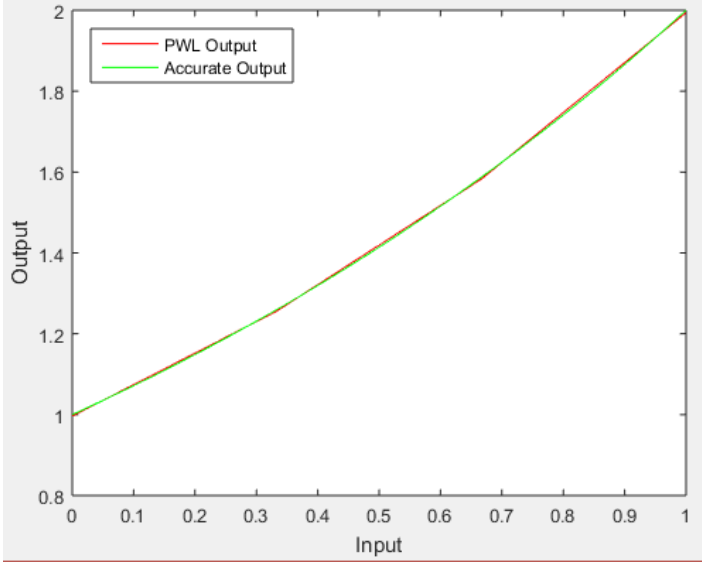


Figure 2. PWL Output vs. Accurate Output

The PWL segments closely follow the accurate base 2 exponential output. The maximum absolute error obtained in the domain of [0, 1) is roughly 0.001661, which corresponds to 9 error-free fractional bits for a fixed point PWL output.

2.2.2. The ac_exp_pwl Implementation

In order to calculate the natural exponential of ac_fixed inputs, we rely upon the *ac_pow2_pwl* implementation. The following relation is used:

```
exp(x) = 2 ^ ( x * log2(e) )
```

Hence, all that needs to be done is to multiply the input by log2(e)), store the product in a temporary variable, and then pass that to the *ac_pow2_pwl* implementation. While doing so, it is ensured that the temporary variable has enough fractional bits to represent the multiplication result.

2.2.3. Default Template Arguments

Rounding Mode for PWL Output

The internal variable to store the computation of the PWL output is set to have rounding (*AC_RND*) turned on by default. If, however, the user wishes to use another rounding mode, they can pass it as a template argument. For an example on how to pass this argument, please refer to the "Example Function Calls" section below.

Setting the Minimum Number of Fractional Bits

In order to change the minimum number of fractional bits that are used to store the product of *x* and *log2(e)*, the user can pass a template value with the minimum they desire. By default this value is set to 9, because that minimum value was empirically seen to provide the best accuracy for inputs that did not have enough fractional bits. For an example on how to pass this argument, please refer to the *Example Function Calls* section below.

2.2.4. Function Prototypes

The following code shows the template prototypes for the different implementations:

```
template<ac_q_mode pwl_Q = AC_RND,
        int W, int I, bool S,
        ac_q_mode Q, ac_o_mode O,
        int outW, int outI,
        ac_q_mode outQ, ac_o_mode outO>
void ac_pow2_pwl(
  const ac_fixed<W, I, S, Q, O> &input,
  ac_fixed<outW, outI, false, outQ, outO> &output
)
```

```
template<int n_f_b = 9,
        ac_q_mode pwl_Q = AC_RND,
        int W, int I, bool S,
        ac_q_mode Q, ac_o_mode O,
        int outW, int outI,
        ac_q_mode outQ, ac_o_mode outO>
void ac_exp_pwl(
  const ac_fixed<W, I, S, Q, O> &input,
  ac_fixed<outW, outI, false, outQ, outO> &output
)
```

Returning by Value

The *ac_pow2_pwl* and *ac_exp_pwl* functions can return their output by value as well as by reference. In order to return the value by value, the user must pass the information of the output type to the function as a template argument. For an example of how to do this, please refer to the *Example Function Calls* section below. The prototypes for the function call to return by value for both functions is shown below:

```
template<class T_out,
        ac_q_mode pwl_Q = AC_RND,
        class T_in>
T_out ac_pow2_pwl(
```

```
    const T_in &input
)
```

```
template<class T_out,
        int n_f_b = 9,
        ac_q_mode pwl_Q = AC_RND,
        class T_in>
T_out ac_exp_pwl(
    const T_in &input
)
```

2.2.5. C++ Compiler

The functions use default template arguments. In order to use a C++ compiler that supports this functionality, the user must use C++11 as the standard for their compilation, or a later standard, failing which a compile-time error is thrown.

2.2.6. Example Function Calls

An example of a function call to store the value of the natural exponential of a sample *ac_fixed* variable *x* in a variable *y* is shown below:

```
typedef ac_fixed<20, 11, true, AC_RND, AC_SAT> input_type;
typedef ac_fixed<30, 15, true, AC_RND, AC_SAT> output_type;

input_type  x = 2.875;
output_type y_exp;
output_type y_pow2;

ac_pow2_pwl(x, y_pow2); //Approximates y_pow2=pow(2,x.to_double())
ac_exp_pwl(x, y_exp);   //Approximates y = exp(x)
```

The variable *y* hereafter contains the approximate value of the natural exponential of x.

Changing the Minimum Number of Fractional Bits

As mentioned earlier, the minimum number of fractional bits for the result of the multiplication of the input to the natural exponent functions and log2(e) can be varied. An example of this is shown below, where the user allocates a minimum of 7 fractional bits.

```
ac_exp_pwl<7>(x, y);
```

Changing Rounding/Truncation Mode for the PWL Output Variable

As mentioned earlier, the temporary variable that stores the PWL output has rounding turned on by default. An example in which the user changes the rounding/truncation mode is shown as follows:

```
ac_pow2_pwl<AC_TRN>(x, y);
ac_exp_pwl<9, AC_TRN>(x, y);
```

The latter function call changes the rounding/truncation mode while keeping the minimum number of fractional bits same as the default, i.e. 9.

Returning by Value

In order to have the function return by value, the user must pass the output type information as a template parameter. This is done as follows, for the base 2 and base e exponential functions:

```
y = ac_pow2_pwl<output_type>(x);
y = ac_exp_pwl<output_type>(x);
```

If the user wants to change the default template parameters, e.g. they want to truncate the output of the PWL implementation and/or change the minimum no. of fractional bits to, say, 7, they can do so by using the following function calls as guidelines:

```
y = ac_pow2_pwl<output_type, AC_TRN>(x);
y = ac_exp_pwl<output_type, 7, AC_TRN>(x);
```

2.3. Reciprocal (ac_reciprocal_pwl)

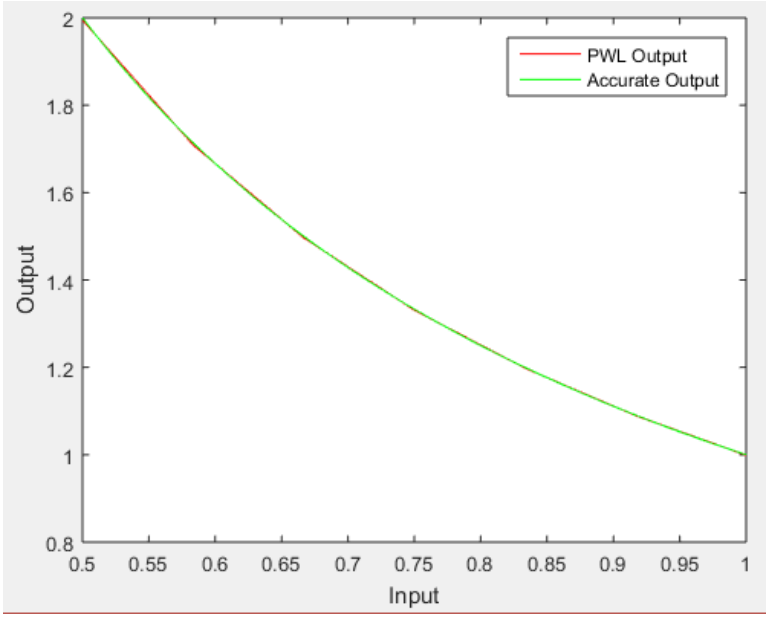
The ac_reciprocal_pwl function is designed to provide a quick approximation of the reciprocal of real and complex numbers using a Piecewise Linear (PWL) implementation with 7 points/6 segments, optimized to give a high-accuracy implementation. Many hardware division operations are calculated indirectly by first obtaining the value of the reciprocal of the denominator, and then multiplying that with the numerator. The calculation of the reciprocal is generally done using PWL approximation, which is faster and requires lesser area than actual division hardware.

2.3.1. The ac_reciprocal_pwl Implementation

The *ac_reciprocal_pwl* library provides four overloaded functions for the calculation of the reciprocal of real and complex inputs. Each overloaded function handles a different input datatype. The four datatypes hence handled are (a) ac_fixed, (b) ac_float, (c) ac_complex<ac_fixed> and (d) ac_complex<ac_float>. It is the ac_fixed function that actually contains the code required for the PWL implementation. All the other functions rely upon ac_fixed PWL implementation in one way or another, as will be explained later.

PWL Approximation

To explain the closeness of the PWL approximation to the “real thing”, i.e. the actual, accurate implementation of the reciprocal function, the following is a graph of PWL output vs. the accurate function output, using double precision floating point values for the PWL computation in MATLAB.



The PWL segments closely follow the accurate reciprocal output. The maximum absolute error obtained in the domain of [0.5, 1) is roughly 0.005511, which corresponds 7 error-free fractional bits in case of a fixed point PWL output.

Normalization and De-normalization

Before we can implement the PWL approximation, normalization is conducted on the input using the ac_normalize library. The normalization scales the input to the range of [0.5, 1) depending upon the bit content in the input. The reciprocal of this normalized output is calculated through PWL approximation. This reciprocal is then passed through a sort of "de-normalization" that cancels the effect of the previous scaling.

2.3.2. Default Template Arguments

The internal variable used to store the computation of the PWL output is set to have rounding turned on by default. If, however, the user wishes to not use other rounding/truncation modes for the intermediate variable, they can do so by passing the desired round/truncation mode as a template argument. For an example on how to pass this argument, please refer to the "Example Function Calls" section below.

2.3.3. Function Templates

The following are the overloaded function prototypes for the ac_reciprocal_pwl function for different datatypes:

```
template<ac_q_mode pwl_Q = AC_RND,
        int W,
        int I,
        bool S,
        ac_q_mode Q,
        ac_o_mode O,
        int outW,
        int outI,
        bool outS,
        ac_q_mode outQ,
```

```
        ac_o_mode outO>
void ac_reciprocal_pwl(
  const ac_fixed<W, I, S, Q, O> &input,
  ac_fixed<outW, outI, outS, outQ, outO> &output
)
```

```
template<ac_q_mode pwl_Q = AC_RND,
        int W,
        int I,
        int E,
        ac_q_mode Q,
        int outW,
        int outI,
        int outE,
        ac_q_mode outQ>
void ac_reciprocal_pwl(
  const ac_float<W, I, E, Q> &input,
  ac_float<outW, outI, outE, outQ> &output
)
```

```
template<ac_q_mode pwl_Q = AC_RND,
        int W,
        int I,
        bool S,
        ac_q_mode Q,
        ac_o_mode O,
        int outW,
        int outI,
        bool outS,
        ac_q_mode outQ,
        ac_o_mode outO>
void ac_reciprocal_pwl(
  const ac_complex<ac_fixed<W, I, S, Q, O> > &input,
  ac_complex<ac_fixed<outW, outI, outS, outQ, outO> > &output
)
```

```
template<ac_q_mode pwl_Q = AC_RND,
        int W,
        int I,
        int E,
        ac_q_mode Q,
        int outW,
        int outI,
        int outE,
        ac_q_mode outQ>
void ac_reciprocal_pwl(
     const ac_complex<ac_float<W, I, E, Q> > &input,
     ac_complex<ac_float<outW, outI, outE, outQ> > &output
)
```

2.3.4. Returning by Value

The *ac_reciprocal_pwl* functions can return their output by value as well as by reference. In order to return the value, the user must pass the information of the output type to the function as a template argument. For

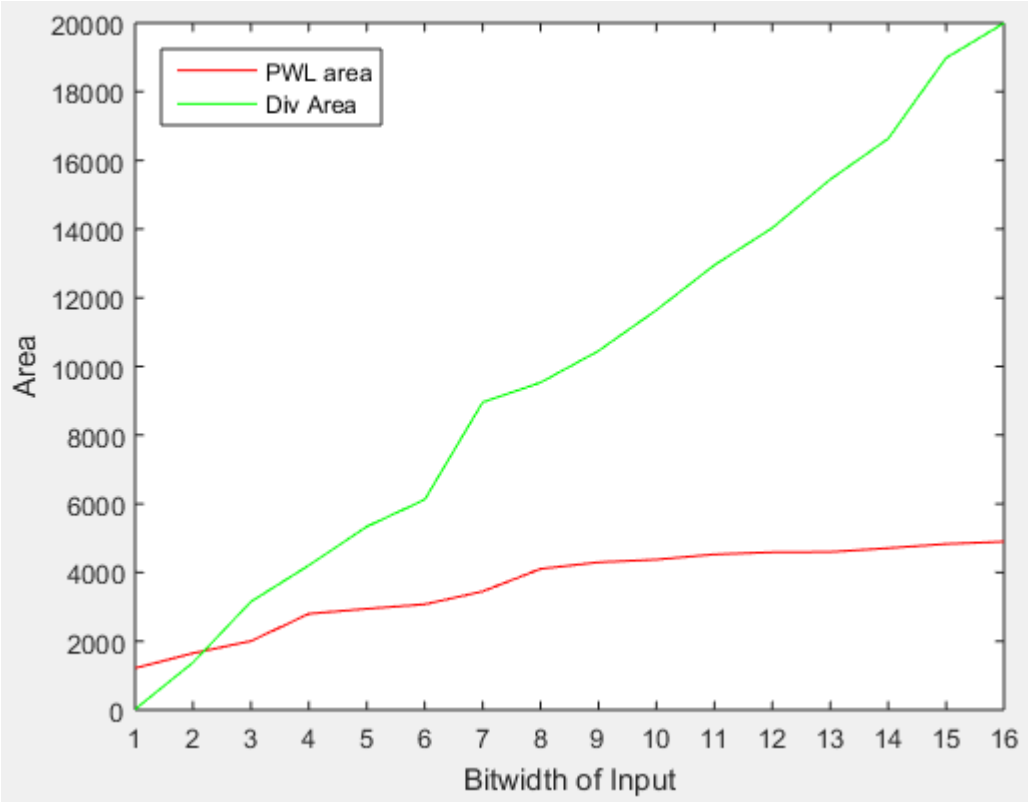
an example of how to do this, please refer to the *Example Function Calls* section below. The prototype for the function call to return by value is shown below:

```
template<class T_out,
        ac_q_mode pwl_Q = AC_RND,
        class T_in>
T_out ac_reciprocal_pwl(
    const T_in &input
)
```

2.3.5. Comparison with the div() Function

The *mgc_ac_math* library already provides the *div()* function for exact division of *ac_fixed* values, but in order to make it achieve 100% throughput, it is necessary to completely unroll the loops in that function. This generally results in a high area for implementation. In order to illustrate this disadvantage, the following graph was plotted while considering the following:

- Input, output and temporary values for the PWL were all ac_fixed variables.
- Input bit-width was varied from 1 to 16 bits.
- The input used 5 integer bits, signedness (true), with rounding and saturation modes (*AC_RND* and *AC_SAT*) constant.
- The bit-width (64), number of integer bits (32), signedness (true), rounding and saturation modes (AC_RND and AC_SAT) were kept constant for the output.
- The loop for the *div()* function was completely unrolled, and the PWL function was pipelined with an *II = 1* for 100% throughput.
- The number of fractional bits for all the internal PWL variables was set to 9. The variable storing the output of the PWL approximation had rounding turned on.
- Every value that could be represented for the given input precision and signedness was passed as an input to the testbench.
- The clock period was set to 0.8 ns for all cases. Decreasing the clock period beyond 0.8 nS produces feedback timing violations when the *div* function exceeds an input width of 16.



For low (< 3) values of bitwidth, the div function takes up lesser area than the PWL. However, when the bitwidth goes beyond those values, the PWL function takes up significantly lesser area for its implementation. e. g. When the input bitwidth is at its highest value, i.e. 16 in this case, the PWL function takes up only a quarter of the area that the div function occupies.

Of course, some accuracy is sacrificed. The maximum relative error suffered while using PWL approximation (compared to the direct evaluation of 1/x with the double representation being used for the input and its result) for an input bit-width of 16 is 0.3212%, while the maximum relative error for the div function is 1.85275e-7%, which is essentially negligible.

2.3.6. Example Function Calls

An example of a function call to store the value of the reciprocal of a sample ac_fixed variable x in a variable y is shown below:

```
typedef ac_fixed<20, 11, true, AC_RND, AC_SAT> input_type;
typedef ac_fixed<30, 15, true, AC_RND, AC_SAT> output_type;
input_type x = -1.75;
output_type y;
ac_reciprocal_pwl(x, y);  //Approximates y = 1 / x
```

The variable y hereafter stores the approximate value of the reciprocal of x.

Changing Rounding/Truncation Mode for PWL Output Variable

As mentioned earlier, the temporary variable that stores the PWL output has rounding turned on by default. An example of passing AC_TRN as the rounding/truncation mode instead, for the same data-types of x and y as in the previous example is shown below:

```
ac_reciprocal_pwl<AC_TRN>(x, y);
```

Returning by Value

As mentioned earlier, the reciprocal_pwl functions can also return by value. In order to do so, the type information for the output must be passed explicitly to function as shown below.

```
y = ac_reciprocal_pwl<output_type>(x);
```

If the user also wishes to change the rounding mode for the temporary variable and have the function return by value, they can call the function as follows:

```
y = ac_reciprocal_pwl<output_type, AC_TRN>(x);
```

2.4. Square Root (ac_sqrt_pwl)

The *ac_sqrt_pwl* function is piecewise linear implementation of a square root function that has been optimized to provide high performance with accurate results. This function is implemented as an overloaded function for *ac_fixed*, *ac_float*, *ac_complex of ac_fixed* and takes positive values as inputs. If negative input is provided to it, it will throw a compilation error. It provides really good accuracy. It is observed that the pwl implementation is faster than other algorithms and consumes significantly smaller area making it useful as a basic building block in high speed IP blocks.

2.4.1. The ac_sqrt_pwl Implementation

The header file provides square root implementation for *ac_fixed*, *ac_float* and *ac_complex* datatypes and is optimized to provide high internal accuracy. User has been given freedom to choose the input and output bit widths, signedness (for complex datatype implementation), saturation and rounding types. Additionally, you can supply internal rounding of PWL implementation using a template argument. By default, the output value is rounded using the standard *AC_RND* option.

2.4.2. Basic Algorithm

Before PWL implementation the input value is first given to the normalization function, which converts the input value into an exponent and a normalized output. The normalized output is then subjected to the 5 points/4 segments piecewise linear implementation. After experimenting with multiple combinations, it was found that this combination provides a great accuracy, with significantly lesser area. Although user can replace the c and m values by generating his/her own version of piecewise linear implementation that will provide different accuracy with different ROM sizes based on his/her requirement of accuracy, speed, and area. LUT generator is included in the package which includes code of achieving that.

Square root of the input is then product of square root of normalized value multiplied by 2 to the power of half of normalized exponent obtained at the output of the normalization function.

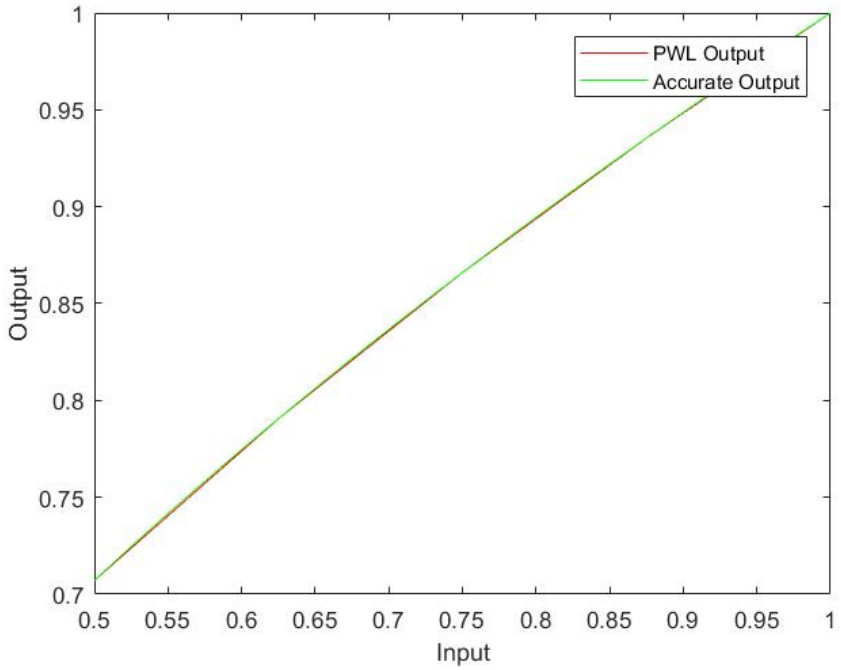
Lookup Table (LUT) Initialization

Lookup table values are computed using the LUT generator code provided in the package and user can replace them to change the ROM values and precision based on his/her requirement. Lookup table values are defined as const values to make sure that they are synthesized as ROM by Catapult. The ROMs store values of ac_fixed type, with total bit width as 12 and integer bit width as 0. It can be mathematically proven that 12 bits are enough to achieve sufficient precision.

LUT generator is non-synthesizable C++ code which writes values in a separate text file.

PWL Implementation

The following illustration shows a graphical comparison between Square root function of MATLAB and pwl implementation (number of segments = 4, range = [0.5,1])



The PWL implementation of square root function for fixed point datatype takes domain/input values from 0 onwards and is monotonically increasing. Square root graph of negative values is a complex value and can be simply built on top square root implementation of positive values API. Mathematically, it can be proven that the range of 0.5-1 is enough to reflect the overall structure of the graph and hence in PWL implementation this range is used.

This range is then divided into 4 segments that are equally spaced from each other. Value of input to the pwl is then converted into the scaled input, which is converted into integer for determining index from the ROM.

As mentioned in the basic algorithm section, after square root value of normalized value is computed, the exponent obtained from normalized function is then halved (based on if it is even or not).

If the exponent is uneven, then the output of square root of normalized input is then multiplied square root of 2 and then exponent is reduced by one and then is halved.

Localized two raised to index is then computed by using left shift.

The maximum error was reduced to as low as 0.09% for fixed point, with

- **Range**: (0,1234]
- **Step**: 1
- **Allowed Error**: 1%
- **Input type**: *ac_fixed <54,27,false,AC_RND, AC_SAT>*
- **Output type**: *ac_fixed <42, 15, false, AC_RND, AC_SAT>*

***NOTE**: The implementation can also take negative integer bit widths as input bit width or integer width can be greater than total width.*

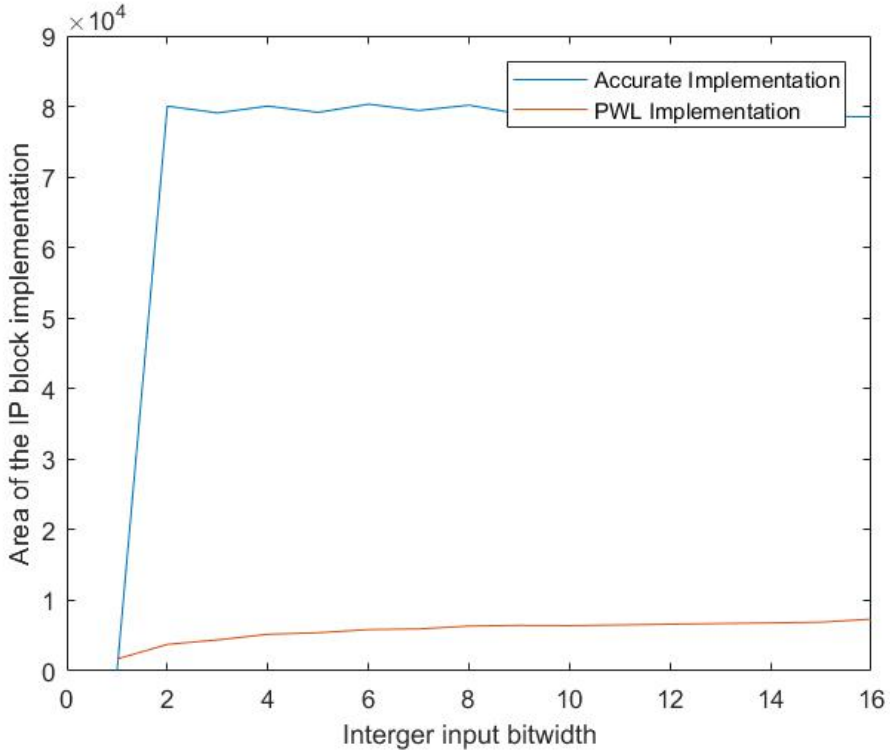
There is also an additional optional template parameter which lets user chose the rounding type of the normalized_output, with default value set to AC_RND.

2.4.3. Comparison with square root accurate implementation

The mgc_ac_math library already provides the sqrt function for exact division of ac_fixed values, but in order to make it achieve 100% throughput, it is necessary to completely unroll the loop in that function. This generally results in a high area for implementation. In order to illustrate this disadvantage, the graph shown in Figure 2. was plotted while considering the following:

- Input, output and temporary values for the PWL were all ac_fixed variables.
- Input bit-width was varied from 1 to 16 bits.
- The number of integer bits (5), signedness (false), rounding and saturation modes (AC_RND and AC_SAT) were kept constant too for the input.
- The bit-width (64), number of integer bits (32), signedness (false), rounding and saturation modes (AC_RND and AC_SAT) were kept constant for the output.
- The loop for the sqrt () function (square root) was completely unrolled, and the PWL function was pipelined with an II = 1 for 100% throughput.
- The number of fractional bits for all the internal PWL variables was set to 12. The variable storing the output of the PWL approximation had rounding turned on.
- Every value that could be represented for the given input precision was passed as an input to the testbench.
- The clock period was set to 1.65 ns for all cases. This was done because a clock period any lesser than that would results in increase in area, for the case when input width = 16

The following illustration plots the Area vs. Bitwidth for the *PWL* and *sqrt()* functions.



For 1 integer bitwidth, the sqrt function takes up lesser area than the PWL. However, when the bitwidth goes beyond those values, the PWL function takes up significantly lesser area for its implementation. e. g. When the input bitwidth is at its highest value, i.e. 16 in this case, the PWL function takes up only one tenth of the area that the sqrt function occupies. Of course, some accuracy is sacrificed. The maximum error suffered while using PWL approximation (compared to the direct evaluation of square root function with the double representation being used for the input and its result) for an input bit-width of 16 is 0.115936%, while the maximum error for the sqrt function is 3.95915e-7%, which is essentially negligible.

2.4.4. Function headers

There are total three implementations of the square root function that handle ac_fixed, ac_float and ac_complex of ac_fixed. The headers of the functions are as follows:

For ac_fixed:

```
template<ac_q_mode q_mode_temp = AC_RND,
        int input_width,
        int input_int,
        ac_q_mode q_mode,
        ac_o_mode o_mode,
        int output_width,
        int output_int,
        ac_q_mode q_mode_out,
        ac_o_mode o_mode_out>
void ac_sqrt_pwl(
  const ac_fixed<input_width, input_int, false, q_mode, o_mode> input,
  ac_fixed<output_width, output_int, false, q_mode_out, o_mode_out> &output
);
```

For ac_float:

```
template<ac_q_mode q_mode_temp = AC_RND,
        int input_width,
        int input_int,
        int input_exp,
        ac_q_mode q_mode,
        int output_width,
        int output_int,
        int output_exp,
        ac_q_mode q_mode_out>
void ac_sqrt_pwl(
  const ac_float<input_width, input_int, input_exp, q_mode> input,
  ac_fixed<output_width, output_int, output_exp, q_mode_out> &output
);
```

For ac_complex of ac_fixed:

```
template <ac_q_mode q_mode_temp = AC_RND,
        int input_width,
        int input_int,
        ac_q_mode q_mode,
        ac_o_mode o_mode,
        int output_width,
        int output_int,
        ac_q_mode q_mode_out,
        ac_o_mode o_mode_out>
void ac_sqrt (
     const ac_complex <ac_fixed <input_width,input_int, true, q_mode, o_mode>  input,
     ac_complex <ac_fixed <output_width, output_int, true, q_mode_out, o_mode_out>
&output);
```

Apart from above header, there is one more header which is created to return the values of the output, in which case output_type has to be supplied as the template parameter.

The header is as follows:

```
template< typename output_type,
        ac_q_mode q_mode_temp = AC_RND,
        typename input_type>
void ac_sqrt_pwl(
  const input_type input,
  output_type &output
);
```

2.4.5. Handling different datatypes

As mentioned earlier, ac_fixed datatype implementation is the one subjected to piecewise linear operations, where as other datatypes give it call directly or indirectly as explained below:

Handling ac_float

Handling of ac_float is simply done by separating the mantissa and the exponent and then operating on them separately. First mantissa is subjected to a piecewise linear implementation of ac_fixed by calling ac_fixed implementation.

Exponent of the ac_float is then either divided by two (if it is even) or is reduced by one and then divided by zero (if it is odd), the result is assigned to the exponent of the output. If exponent is even, output mantissa is

assigned with output of pwl, where as in case of odd exponent, output mantissa is multiplication of output of pwl and square root of 2, which is declared as constant.

Handling of ac_complex

Square root of a complex number can be obtained by following formula:

```
Output_real_part = sqrt ((sqrt (mod) + input_real_part)/2)
Output_imag_part = sqrt ((sqrt (mod) - input_real_part)/2)
```

This square root computation requires computation of intermediate square roots of fixed values, which is achieved by using implementations for ac_fixed.

2.4.6. Example Function Call

Following is the example call for ac_sqrt_pwl function for ac_fixed datatype.

```
typedef ac_fixed<20, 11, false, AC_RND, AC_SAT> input_type;
typedef ac_fixed<10, 8, false, AC_RND, AC_SAT> output_type;

input_type x = 1.75;
output_type y;

// Returns y = sqrt(x)
ac_sqrt_pwl (x, y)

//Above part of call can be replaced by
y = ac_sqrt_pwl <output_type> (x);
```

Following is the example call for ac_sqrt_pwl function for ac_float datatype.

```
typedef ac_float<32, 16, 4, AC_RND> input_type;
typedef ac_float<16, 8, 4, AC_RND> output_type;

input_type x = 1.75;
output_type y;

//Returns y = sqrt(x)
ac_sqrt_pwl (x, y)

//Above part of call can be replaced by
y = ac_sqrt_pwl <output_type> (x);
```

Following is the example call for ac_sqrt_pwl function for ac_complex <ac_fixed> datatype.

```
typedef ac_complex <ac_fixed<20, 11, true, AC_RND, AC_SAT > > input_type;
typedef ac_complex <ac_fixed<10, 6, true, AC_RND, AC_SAT > > output_type;

input_type x = (1.75,1.7);
output_type y;

//Returns y = sqrt(x)
ac_sqrt_pwl (x, y)

//Above part of call can be replaced by
y = ac_sqrt_pwl <output_type> (x);
```

Section 3. Lookup Table (LUT) Functions

The *ac_math* package includes the following Lookup Table functions:

- Sine/Cosine (ac_sincos)

The following subsections describes the implementation and usage of these functions in more detail.

3.1. Sine/Cosine (ac_sincos)

The *ac_sincos* function is designed to provide the sine and cosine values using a lookup table (LUT).

3.1.1. The ac_sincos Implementation

The *ac_sincos* library is used to accept *ac_fixed* datatype as input. The domain for the input is (0,1) radians/2PI. The function returns an *ac_complex<ac_fixed>* variable where the real part represents the *cos* value and the imaginary part represents the *sin* value. The number of table entries are 512 values. A prototype of the *sincos* function is shown below:

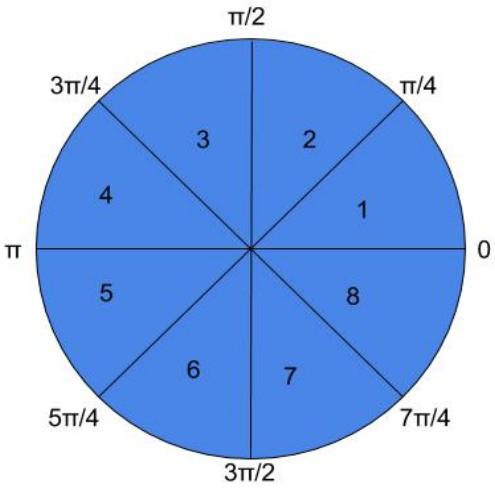
```
template<class T_in,
         class T_out>
void ac_sincos(const T_in &input,
               ac_complex<T_out> &output
)
```

In the above prototype, the class *T_in* represents the datatype of the input which should be *ac_fixed* datatype. The class *T_out* represents the datatype of the output which should be *ac_fixed* datatype. Here, *input* represents the input to the *ac_sincos* function, the *output* is an *ac_complex* variable whose real part is the output *cos* value and whose imaginary part is the output *sin* value.

Algorithm

All the entries for the sin and cos lookup tables were generated using the C++ math library sin and cos functions.

Now, as the lookup table contains the sin and cos values for (0-PI/4] radians, so firstly 3 bits are extracted from the MSB side of the input to determine in which octant the input lies. The octant is defined by the ‘range’ variable in the function. The octants can be defined as: [0 – 0.125) - 0, [0.125 – 0.25) - 1 , [0.25 - 0.375) - 2 , [0.375 – 0.5) - 3 ,[0.5 – 0.625) – 4, [0.625 – 0.75) – 5, [0.75 – 0.875) - 6, [0.875 – 1) - 7. So for example if the input is 0.3, then range i.e the octant is 2. A circle representing the octants is shown below. In the case of fractional bits being less than or equal to 12, the look up table index is determined by the (number of fractional bits – 3 from the LSB side) bits which are extracted from the input. Whereas when the number of fractional bits are more than 12, then the closest lookup table entry is found for the input. There are 2 lookup tables provided for sin and cos which have 512 entries corresponding to the maximum number of fractional bits. Then for the number of fractional bits smaller than 12, a stride is implemented.



The circle divided into octants. The corresponding trigonometric identities are used depending on the range and a lookup using the look up table index is carried out to determine the output.

Handling negative inputs

The two's complement encoding on the angle 0.125 would be:

0.0010 0000 (pre-negated) -> 1.1101 1111 (inverted) -> 1.1110 0000 (after adding 1 for 2's complement form)

If I then take the fractional bits and treat it as a positive number -> .1110 0000 = 0.875, which is exactly the same angle as -0.125 and therefore negative inputs are handled.

Returning by Value

The ac_sincos function can return the output by value as well as by reference. In order to return the output by value, the user must pass the information of the output type to the function as a template argument. For an example of how to do this, please refer to the *Example Function Call* section below. The prototype for the function to return by value is shown below:

```
    template<class T_out, class T_in>
  T_out ac_sincos(
    const T_in &input
  )
```

3.1.2. Example Function Call

An example of a function call is as follows where x represents the input angle and y represents a complex variable whose real part is the cos value and imaginary part is the sin value:

```
typedef ac_fixed<12, 1, true, AC_RND, AC_SAT> input_type;
typedef ac_complex<ac_fixed<23, 1, true, AC_RND, AC_SAT> > output_type;

input_type x = 0.37;
output_type y;

ac_sincos(x, y);
```

Returning by Value

As mentioned earlier, the ac_sincos function can also return by value. In order to do so, the type information for the output must be passed explicitly to function as shown below.

```
y = ac_sincos<output_type>(x);
```

where x is input and y is output.

Synthesizing the ac_sincos function

In HLS Synthesis, the *ac_sincos* function can be completely pipelined as there are no dependencies.