SIEMENS EDA

Algorithmic C (AC) Math Library Reference Manual

Software Version v3.4.1
October 2021

Table of Contents

Table of Contents

Table of Contents

Chapter 1: Introduction to ac_math

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. The following sections provide a summary of the *ac_math* library:

- Summary of Functions
- Types of Approximations
- Installing the ac_math library

1.1.1. 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>
```

1.2. Summary of Functions

The following sections summarize the functions and classes currently supported in the ac_math library. A discussion of the approximation methods follows.

1.2.1. Basic Math Functions

Function Type/Call	Approx. Method	Supported Data Types			
		ac_fixed	ac_float	ac_complex	Other float*
Absolute Value (*ac_abs()*)	N/A	Yes	Yes	Yes	No
Division (*ac_div()*)	N/A	Yes	Yes	Yes	No
Normalization (*ac_normalize()*)	N/A	Yes	No	Yes	No
Reciprocal (*ac_reciprocal_pwl()*)	PWL	Yes	Yes	Yes	Yes
Reciprocal (*ac_reciprocal_pwl_ha()*)	PWL	Yes	Yes	Yes	Yes

Function Type/Call	Approx. Method	Supported Data Types			
		ac_fixed	**ac_float**	**ac_complex**	**Other float***
Reciprocal *(ac_reciprocal_pwl_vha())*	PWL	Yes	Yes	Yes	Yes
Logarithm Base e (*ac_log_pwl()***)**	PWL	Yes	Yes	No	Yes
Logarithm Base e (*ac_log_cordic()***)**	CORDIC	Yes	No	No	No
Logarithm Base 2 (*ac_log2_pwl()***)**	PWL	Yes	Yes	No	Yes
Logarithm Base 2 (*ac_log2_cordic()***)**	CORDIC	Yes	No	No	No
Exponent Base e *(ac_exp_pwl())*	PWL	Yes	Yes	No	Yes
Exponent Base e *(ac_exp_cordic())*	CORDIC	Yes	No	No	No
Exponent Base 2 *(ac_pow2_pwl)*	PWL	Yes	No	No	No
Exponent Base 2 *(ac_exp2_cordic())*	CORDIC	Yes	No	No	No
Generic Exponent *(ac_pow_pwl())*	PWL	Yes	No	No	No
Generic Exponent *(ac_pow_cordic())*	CORDIC	Yes	No	No	No
Square Root *(ac_sqrt_pwl())*	PWL	Yes	Yes	Yes	Yes
Square Root *(ac_sqrt())*	N/A	Yes	Yes	No	Yes
Inverse Square Root *(ac_inverse_sqrt_pwl())*	PWL	Yes	Yes	Yes	Yes
Inverse Square Root *(ac_inverse_sqrt_pwl_vha())*	PWL	Yes	Yes	Yes	Yes
Sine/Cosine *(ac_sincos_lut())*	LUT	Yes	No	No	No
Sine/Cosine *(ac_sincos_cordic())*	CORDIC	Yes	No	No	No
Cosine *(ac_cos_cordic())*	CORDIC	Yes	No	No	No
Sine *(ac_sin_cordic())*	CORDIC	Yes	No	No	No
Tangent *(ac_tan_pwl())*	PWL	Yes	Yes	No	Yes
Arctangent *(ac_atan_pwl())*	PWL	Yes	Yes	No	Yes
Arctangent *(ac_atan_pwl_ha())*	PWL	Yes	Yes	No	Yes
Arctangent *(ac_atan_pwl_vha())*	PWL	Yes	Yes	Yes	Yes
Arctangent *(ac_arctan_cordic())*	CORDIC	Yes	No	No	No
Arccosine *(ac_arccos_cordic())*	CORDIC	Yes	No	No	No
Arcsine *(ac_arcsin_cordic())*	CORDIC	Yes	No	No	No
Shift Left *(ac_shift_left())*	N/A	Yes	No	Yes	No
Shift Right *(ac_shift_right())*	N/A	Yes	No	Yes	No
Hyperbolic Tangent *(ac_tanh_pwl())*	PWL	Yes	Yes	No	Yes
Sigmoid *(ac_sigmoid_pwl())*	PWL	Yes	Yes	No	Yes
Softmax *(ac_softmax_pwl())*	PWL	Yes	No	No	No
Barrel Shift (*ac_barrel_shift()*)	N/A	*ac_int support only			

* Note: “Other float” represents ac_ieee_float and ac_std_float support.

1.2.2. AC Matrix Class

The class ac_matrix implements a 2-D container class with a template parameter to specify the data type of the internal storage.

The class has member functions to implement some common operations including

- Assignment: operator=()
- Read-Only and Read-Write Element Access: *this(<row>,<col>)
- Comparison: operator!=(), operator==()
- Piecewise Addition: operator+(), operator+=()
- Piecewise Subtraction: operator-(), operator-=()
- Piecewise Multiplication: pwisemult()
- Matrix Multiplication (nested loops): operator*()
- Matrix Transpose: transpose()
- Sum All Elements: sum()
- Scale All Elements: scale(value)
- Formatted Stream Output: ostream &operator<<()

When using the computational functions with AC Datatypes, the form that returns a value is designed in such a way as to determine the full precision required in the output type in order to preserve accuracy during the operation. So using operator+ between two 10 bit ac_fixed matrices will return an 11 bit ac_fixed matrix. If you wish to prevent the bit growth and accept the truncation, you can use the compound operators +=,-=, etc. so that the target object receives the truncated values.

In addition to the built-in member functions, the ac_math library also includes stand-alone functions for more complicated linear algebra operations as described in the next section.

1.2.3. Linear Algebra Functions

The ac_math library includes several linear algebra functions that operate on either ac_matrix or plain C-style arrays. These functions, when used with AC Datatypes, are designed to give the user greater control over the bit precision of internal variables and the return value.

- Matrix Multiplication

- Matrix Determinant
- Cholesky Decomposition
- Cholesky Inverse
- QR Decomposition

1.3. 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.

- Piecewise Linear
- Lookup Table (LUT)
- CORDIC
- Miscellaneous Math Functions

1.3.1. Piecewise Linear

Some of the functions available in the ac_math library are implemented as piecewise linear (PWL) approximations.

A PWL approximation of a function essentially attempts to replicate a function as a set of line segments that are the closest fit to the actual function curve.

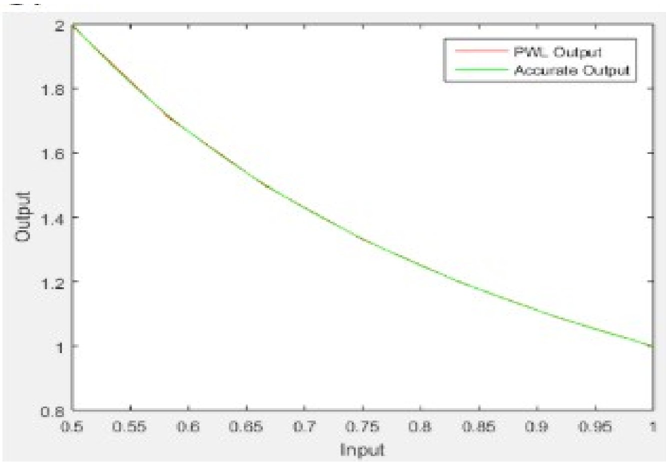


Illustration 1: Reciprocal PWL approximation

An example of a PWL approximation would be the *ac_reciprocal_pwl()* function, for the interval of [0.5, 1). A graphical depiction is given above. The interval is divided into 6 segments. As is evident from the graph and the close nature of the fit, where the red line (PWL output) is virtually indistinguishable from the green one (Accurate output), the approximate output is very close to the accurate output.

Constructing the PWL Lookup Table

In order to construct the PWL approximation, we need information on the accurate function output for the end points of each segment. Once that information is obtained, we calculated the slope and intercept value for that segment based on the input and output values corresponding to the segment end points. The segments are then shifted slightly in the direction opposite to the direction of concavity of the function in the interval of a particular segment, in order to further optimize the fit obtained. This is done by appropriately changing the intercept values for that segment. The slope and intercept values are then stored in separate lookup tables.

The errors obtained are generally small and tolerable for approximate applications. For instance, the reciprocal PWL approximation just mentioned has a maximum absolute error of 0.005511 over the interval of [0.5, 1), which corresponds to at least 7 error-free fractional bits in case of a fixed-point PWL output.

Pitfalls

However, it is important to note that if PWL function calls are cascaded, this error can build up and exceed tolerances. Such a situation is encountered, for instance, if the PWL functions are used for linear algebra functions that operate on matrices. Hence, the user must be aware of this pitfall and take steps to avoid it if necessary.

C++ Compiler

The PWL 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.

Rounding Mode for PWL Output

The internal variable to store the PWL output for all the functions is set to have rounding turned off by default (*AC_TRN*). This is because the default bitwidth of the PWL output operates at full precision, and no rounding is required. If, however, the user wishes to reduce the number of bits and use another rounding mode, they can pass it as a template argument. For an example on how to pass this argument for all the PWL functions that are implemented, please refer to the “Function Prototypes” and “Example Function Calls” sections for the various functions as explained later in the documentation.

1.3.2. 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. In most cases the table is described using literal values expressed as C++ double precision values. The context in which the function is then used (determined by the fixed-point output precision) will determine the amount of error in the function.

1.3.3. 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.

1.3.4. Miscellaneous Math Functions

A number of math functions that do not fall in the categories listed above are also provided. These are the absolute value, division, square root and shifting functions. These functions give an exact or very accurate output.

1.4. Installing the ac_math library

The library consists of three directories, shown here:

```
.
|-- include
|   |-- ac_math
|   |   |-- ac_abs.h
|   |   |-- ac_arccos_cordic.h
|   |   |-- ac_arcsin_cordic.h
|   |   |-- ac_atan2_cordic.h
|   |   |-- ac_atan_pwl.h
|   |   |-- ac_atan_pwl_ha.h
|   |   |-- ac_atan_pwl_vha.h
|   |   |-- ac_barrel_shift.h
|   |   |-- ac_chol_d.h
|   |   |-- ac_cholinv.h
|   |   |-- ac_determinant.h
|   |   |-- ac_div.h
|   |   |-- ac_hcordic.h
|   |   |-- ac_inverse_sqrt_pwl.h
|   |   |-- ac_inverse_sqrt_pwl_vha.h
|   |   |-- ac_log_pwl.h
```

```
|   |   |-- ac_matrixmul.h
|   |   |-- ac_normalize.h
|   |   |-- ac_pow_pwl.h
|   |   |-- ac_qrd.h
|   |   |-- ac_random.h
|   |   |-- ac_reciprocal_pwl.h
|   |   |-- ac_reciprocal_pwl_ha.h
|   |   |-- ac_reciprocal_pwl_vha.h
|   |   |-- ac_shift.h
|   |   |-- ac_sigmoid_pwl.h
|   |   |-- ac_sincos_cordic.h
|   |   |-- ac_sincos_lut.h
|   |   |-- ac_softmax_pwl.h
|   |   |-- ac_sqrt.h
|   |   |-- ac_sqrt_pwl.h
|   |   |-- ac_tan_pwl.h
|   |   `-- ac_tanh_pwl.h
|   |-- ac_math.h
|   `-- ac_matrix.h
|-- lutgen
|   |-- ac_atan_pwl_lutgen.cpp
|   |-- ac_atan_pwl_ha_lutgen.cpp
|   |-- ac_atan_pwl_vha_lutgen.cpp
|   |-- ac_inverse_sqrt_pwl_lutgen.cpp
```

```
|   |-- ac_inverse_sqrt_pwl_vha_lutgen.cpp
|   |-- ac_log_pwl_lutgen.cpp
|   |-- ac_pow_pwl_lutgen.cpp
|   |-- ac_reciprocal_pwl_lutgen.cpp
|   |-- ac_reciprocal_pwl_ha_lutgen.cpp
|   |-- ac_reciprocal_pwl_vha_lutgen.cpp
|   |-- ac_sigmoid_pwl_lutgen.cpp
|   |-- ac_sincos_lut_lutgen.cpp
|   |-- ac_sqrt_pwl_lutgen.cpp
|   |-- ac_tan_pwl_lutgen.cpp
|   |-- ac_tanh_pwl_lutgen.cpp
|   `-- helper_functions.h
|-- pdfdocs
|   `-- ac_math_ref.pdf
`-- tests
    |-- Makefile
    |-- rtest_ac_abs.cpp
    |-- rtest_ac_arccos_cordic.cpp
    |-- rtest_ac_arcsin_cordic.cpp
    |-- rtest_ac_atan2_cordic.cpp
    |-- rtest_ac_atan_pwl.cpp
    |-- rtest_ac_atan_pwl_ha.cpp
    |-- rtest_ac_atan_pwl_vha.cpp
    |-- rtest_ac_chol_d.cpp
```

|-- rtest_ac_cholinv.cpp

|-- rtest_ac_determinant.cpp

|-- rtest_ac_div.cpp

|-- rtest_ac_exp2_cordic.cpp

|-- rtest_ac_exp_cordic.cpp

|-- rtest_ac_exp_pwl.cpp

|-- rtest_ac_inverse_sqrt_pwl.cpp

|-- rtest_ac_inverse_sqrt_pwl_vha.cpp

|-- rtest_ac_log2_cordic.cpp

|-- rtest_ac_log2_pwl.cpp

|-- rtest_ac_log_cordic.cpp

|-- rtest_ac_log_pwl.cpp

|-- rtest_ac_matrix.cpp

|-- rtest_ac_matrixmul.cpp

|-- rtest_ac_normalize.cpp

|-- rtest_ac_pow2_pwl.cpp

|-- rtest_ac_pow_cordic.cpp

|-- rtest_ac_pow_pwl.cpp

|-- rtest_ac_qrd.cpp

|-- rtest_ac_reciprocal_pwl.cpp

|-- rtest_ac_reciprocal_pwl_ha.cpp

|-- rtest_ac_reciprocal_pwl_vha.cpp

|-- rtest_ac_shift.cpp

|-- rtest_ac_sigmoid_pwl.cpp

|-- rtest_ac_sincos_cordic.cpp

|-- rtest_ac_sincos_lut.cpp

|-- rtest_ac_softmax_pwl.cpp

|-- rtest_ac_sqrt.cpp

|-- rtest_ac_sqrt_pwl.cpp

|-- rtest_ac_tan_pwl.cpp

`-- rtest_ac_tanh_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.

Testing and Error Calculation

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.

In addition to the PWL approximations, the LUT and CORDIC implementations are also subject to error calculations on their output, in order to evaluate whether the error is tolerable.

For comparison, the input to the function being tested is converted to a double value (or ac_complex<double>, in case of complex inputs), and this double value is passed to the equivalent C++ math library function. The output, which is a double (or ac_complex<double>, in case of complex outputs), is subject to quantization according the output type of the function being tested.

It is this quantized output that is compared to the output of the function being tested. The comparison is done either in terms of relative error or absolute error. For real outputs, the relative/absolute error between the accurate (C++ math library) output and approximate output is calculated, based upon whether the accurate output lies above or below a pre-defined threshold; if the accurate output lies above the threshold, the relative error is calculated, if it lies below the threshold, the absolute error is calculated. For complex outputs, the error is calculated in a manner similar to the real output, with the differences being that (a) the quantization is carried out on the real and imaginary parts of the accurate output with respect to the type of the real and imaginary parts of the output of the function being tested and (b) the magnitude of the relative/absolute error is what is reported. This approach for comparison is followed for all the ac_math functions except for the:

- *ac_log_pwl, ac_log2_pwl, ac_atan_pwl, ac_atan_pwl_ha, ac_atan_pwl_vha, ac_tanh_pwl, ac_sigmoid_pwl, ac_sincos_lut* and *ac_softmax_pwl* functions where only the absolute error is calculated.
- *ac_abs, ac_shift_left, ac_shift_right* , *ac_barrel_shift* and *ac_normalize functions, which return an exact value at the output. The testing for these functions involves making sure that the expected values are the same as the corresponding actual values that the design returns.*

Chapter 2: Piecewise Linear Functions

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

- Logarithm (ac_log_pwl / ac_log2_pwl)
- Power (ac_exp_pwl / ac_pow2_pwl / ac_pow_pwl)
- Reciprocal (ac_reciprocal_pwl)
- High-accuracy Reciprocal (ac_reciprocal_pwl_ha)
- Very High-accuracy Reciprocal (ac_reciprocal_pwl_vha)
- Square Root (ac_sqrt_pwl)
- 2.7Inverse Square Root (ac_inverse_sqrt_pwl)
- Very High-Accuracy Inv Sqrt (ac_inverse_sqrt_pwl_vha)
- Tangent (ac_tan_pwl)
- Arctangent (ac_atan_pwl)
- High-accuracy Arctangent (ac_atan_pwl_ha)
- Very High-accuracy Arctangent (ac_atan_pwl_vha)
- Sigmoid (ac_sigmoid_pwl)
- Hyperbolic Tangent (ac_tanh_pwl)
- Softmax (ac_softmax_pwl)

Every function above, except for the softmax function, contains its own unique PWL implementation. The softmax function itself depends on the Power (specifically the *ac_exp_pwl* function) and Reciprocal PWL implementations.

Every function normalizes the function input to a value that is within the domain of the PWL approximation. The PWL approximation produces an output for this normalized value, and then applies a sort of "denormalization" that cancels out the effect of the previous normalization on the PWL output. This gives us an approximate output for the input that was supplied to the function. For each PWL approximation except for the softmax operation, the details are given in the table below:

PWL Function	Domain	Segments	Max. Abs. Error	Min. No. of Error-free Fractional Bits

Logarithm	[0.5, 1)	8	0.001251	9
Power	[0, 1)	4	0.003718	8
Reciprocal	[0.5, 1)	8	0.003274	8
High-accuracy Reciprocal	[0.5, 1)	32	0.000236429	12
Very High-accuracy Reciprocal	[0.5, 1)	64	6.33158e-05	13
Square Root	[0.5, 1)	4	0.000582	10
Inverse Square Root	[0.5, 1)	8	0.000893	10
Very High-accuracy Inv. Sqrt.	[0.5, 1)	32	6.56307e-05	13
Tangent	[0, π/4)	8	0.002300	8
Arctangent	[0, 1)	4	0.002885	8
High-accuracy Arctangent	[0, 1)	32	8.9351e-05	13
Very high-accuracy Arctangent	[0, 1)	64	4.59311e-05	14
Sigmoid	[0, 5)	8	0.002755	10
Softmax	N/A	N/A	N/A	N/A

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

2.1. Logarithm (ac_log_pwl / ac_log2_pwl)

The *ac_log_pwl* library provides a piecewise linear implementation for the base 2 and base e logarithmic functions, optimized to provide high performance with quick results. The following datatypes are supported by this IP design: (a) *ac_fixed* (b) *ac_float* (c) *ac_std_float* (d) *ac_ieee_float*. The *ac_fixed* version of the *ac_log2_pwl* function is the one that performs the actual PWL computation. The natural logarithm is computed using the log2 output and the change of base property.

2.1.1. The ac_log2_pwl Implementation

The fixed point *ac_log2_pwl* implementation normalizes the function input to the PWL domain, and performs a PWL approximation to find log2 of the normalized input. This normalized value is then denormalized to produce the final output.

PWL Approximation Graph

To explain the closeness of the PWL approximation to the actual, accurate implementation of the base 2 logarithm function, the following graph compares the PWL output against the accurate function output, where the red line (PWL output) is virtually indistinguishable from the green one (Accurate output):

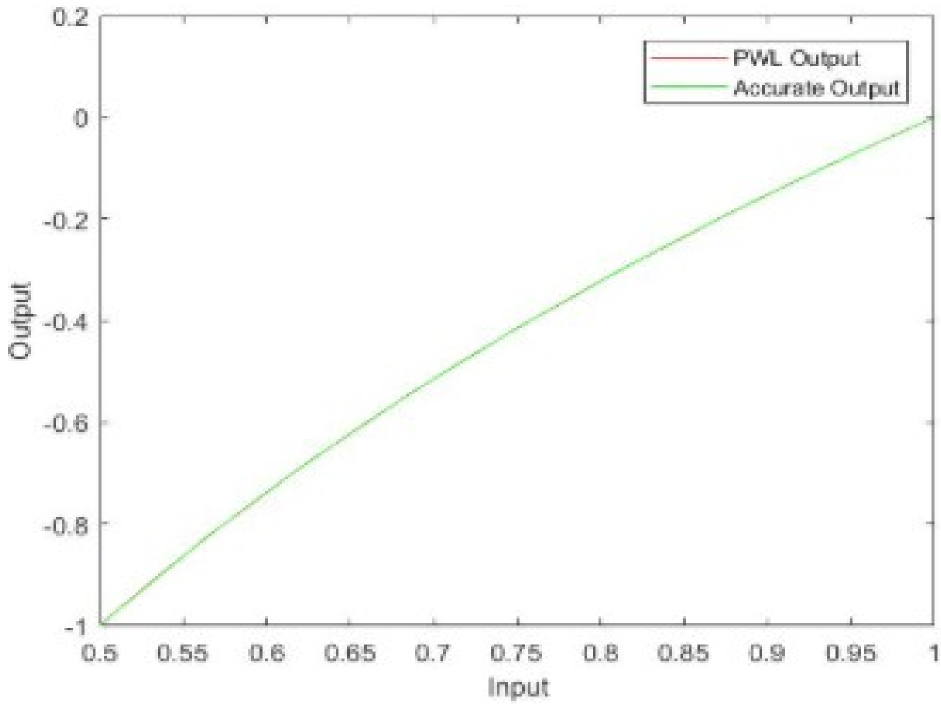


Illustration 2: PWL Output vs. Accurate Output for Base2 Logarithm

Floating Point Support

The design supports *ac_float*, *ac_std_float* and *ac_ieee_float* datatypes. The *ac_float* version uses the *ac_fixed* version to compute log2 of the fixed point input mantissa. The corresponding output is denormalized by taking into account the input exponent and integer width of the input mantissa.

It should be noted that, since the input to the fixed point version (i.e. the input mantissa) is already normalized, the fixed point version does not need to perform a normalization on the input. Therefore, to avoid unnecessary hardware, floating point version uses the default *call_normalize* parameter. For more details on this default parameter, refer to Function Prototypes.

The *ac_std_float* and *ac_ieee_float* version are essentially a wrappers around the ac_float version that perform a conversion between *ac_std_float*/*ac_ieee_float* and the intermediate *ac_float* values that are passed to the ac_float implementation.

Handling Zero and Negative Inputs

A macro-enabled *AC_ASSERT* is provided to alert the user if a zero input is passed to the PWL function. In addition, functionality is also provided to ensure that the ac_log2_pwl function passes the minimum negative value that can be represented with the output type when a zero input is encountered, to mimic an output of negative infinity.

Similarly, a macro-enabled AC_ASSERT is also provided with the floating point implementation to alert the user if a negative input is passed to the PWL function.

2.1.1. The ac_log_pwl Implementation

In order to calculate the natural logarithm of *ac_fixed* inputs, we rely upon the *ac_log2_pwl* implementation. As mentioned earlier, the change of base property is used. It can be represented by the following equation

```
log(x) = log2(x) * log(2)
```

The output of the *ac_log2_pwl* implementation is multiplied by log(2), a constant value. The product is the output of the *ac_log_pwl* function.

Floating Point Support

The interaction between floating point and fixed point *ac_log_pwl* functions is the same as that outlined in for the corresponding *ac_log2_pwl* functions, with the exception being that an additional multiplication is required to perform a change of base as explained above, for *ac_log_pwl* functions.

2.1.2. Function Prototypes

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

```
template<ac_q_mode pwlQ = AC_TRN,
         int W, int I, ac_q_mode Q,  ac_o_mode O,
         int outW, int outI, bool outS, ac_q_mode outQ, ac_o_mode outO>
void ac_log2_pwl(
  const ac_fixed<W, I, false, Q, O> input,
  ac_fixed<outW, outS, outS, outQ, outO> &output,
  const bool call_normalize = true
)
```

The default *call_normalize* argument shown above allows the user to selectively call the *ac_normalize* function. As mentioned earlier, the floating point function calls to the fixed point *ac_log2_pwl* version set *call_normalize* to false in order to bypass the usage of *ac_normalize* and the generation of normalization hardware.

Bear in mind that to completely bypass the usage of *ac_normalize*, the *call_normalize* argument must stay false for every function call. This is done by passing a constant boolean parameter for call_normalize, as can be seen in Bypassing Normalization.

```
template <ac_q_mode pwlQ = AC_TRN,
          int W, int I, ac_q_mode Q, ac_o_mode O,
          int outW, int outI, bool outS, ac_q_mode outQ, ac_o_mode outO>
void ac_log_pwl(
  const ac_fixed <W, I, false, Q, O> input,
  ac_fixed <outW, outI, outS, outQ, outO> &output
 )
```

Note that the fixed point functions only accept unsigned inputs, which corresponds to the behavior of the actual logarithm function, which only accepts positive values for its domain. This is different than the behavior of the C math library logarithm functions, which use signed double datatypes for their inputs.

```
template <ac_q_mode pwlQ = AC_TRN,
          int W, int I, int E, ac_q_mode Q,
          int outW, int outI, int outE, ac_q_mode outQ>
void ac_log2_pwl(
  const ac_float<W, I, E, Q> input,
  ac_float<outW, outI, outE, outQ> &output
)
```

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

```
template <ac_q_mode pwl_Q = AC_TRN, int W, int E, int outW, int outE>
void ac_log2_pwl(
  const ac_std_float<W, E> &input,
  ac_std_float<outW, outE> &output
)
```

```
template <ac_q_mode pwl_Q = AC_TRN, int W, int E, int outW, int outE>
void ac_log_pwl(
  const ac_std_float<W, E> &input,
  ac_std_float<outW, outE> &output
)
```

```
template <ac_q_mode pwl_Q = AC_TRN,
          ac_ieee_float_format Format, ac_ieee_float_format outFormat>
void ac_log2_pwl(
  const ac_ieee_float<Format> &input,
  ac_ieee_float<outFormat> &output
)
```

```
template <ac_q_mode pwl_Q = AC_TRN,
          ac_ieee_float_format Format, ac_ieee_float_format outFormat>
void ac_log_pwl(
  const ac_ieee_float<Format> &input,
  ac_ieee_float<outFormat> &output
)
```

Returning by Value

The *ac_log2_pwl* and *ac_log_pwl* functions can return their 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 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 q_mode_temp = AC_TRN,
         class T_in>
T_out ac_log2_pwl(
  const T_in &input
)
```

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

2.1.3. Example Function Calls

An example of a function call to store the value of the base 2 and base e logarithms of a sample *ac_fixed* variable *x* in the variables y_log2 and y_log is shown below:

```
ac_fixed<20, 11, false, AC_RND, AC_SAT> x = 2.875;
typedef ac_fixed<30, 15,  true, AC_RND, AC_SAT> output_type;
output_type y_log, y_log2;

ac_log2_pwl(x, y_log2); // Approximates y_log2 = log2(x)
ac_log_pwl(x, y_log);   // Approximates y_log  = log(x)
```

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 logarithmic functions:

```
y = ac_log2_pwl<output_type>(x);
y = ac_log_pwl<output_type>(x);
```

If the user wants to change the default template parameters, e. g. they want to round the output of the PWL implementation, they can do so by using the following function calls as guidelines:

```
y = ac_log2_pwl<output_type, AC_RND>(x);
y = ac_log_pwl<output_type, AC_RND>(x);
```

Bypassing Normalization

As mentioned earlier in Function Prototypes, the calls to *ac_normalize* and hence the synthesis of normalization hardware can be bypassed by changing the *call_normalize* argument. An example on how to do so, for *ac_fixed* inputs that are already normalized, is shown below:

```
ac_fixed<16, 5, false> x = 3.5;
ac_fixed<18, 2, true> y;
const bool call_normalize = false;
ac_log2_pwl(x, y, call_normalize);
```

2.2. Power (ac_exp_pwl / ac_pow2_pwl / ac_pow_pwl)

The *ac_pow_pwl* library is designed to provide a quick approximation of the base 2, natural exponentials and generic power functions for real numbers using a piecewise linear (PWL) implementation of the base 2 exponential function with 5 points/4 segments along with the ac_log2_pwl function for the generic power function. This frees us of the burden of having to calculate the output using methods such as Taylor series expansion, which requires loop unrolling, a large number of adders and hence a comparatively large area, to give 100% throughput. By contrast, the ac_pow_pwl function can give 100% throughput by merely pipelining it with an II of 1, which results in a lower area for the hardware.

2.2.1. The ac_pow2_pwl Implementation

The *ac_pow_pwl* library provides three functions, one which calculates the base 2 exponential, one which calculates the natural exponential, and one more that calculates the generic power function (a^b). 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. The generic power function (henceforth called the *ac_pow_pwl* implementation) also depends on the *ac_pow2_pwl* implementation, in addition to the *ac_log2_pwl* function. All three functions only accept *real*, *ac_fixed* inputs and calculate *ac_fixed* outputs.

PWL Approximation Graph

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:

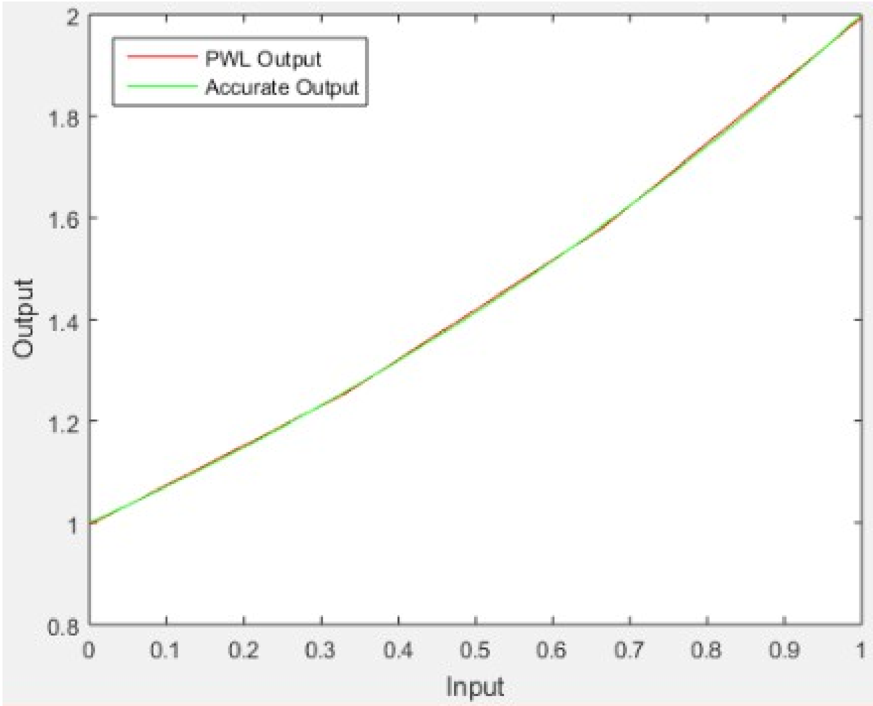


Illustration 3: PWL Output vs. Accurate Output for Base 2 Exponential

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. This can be required when the user uses an input *ac_fixed* type with an insufficient number of fractional bits.

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 *the input* and *log2(e)*, the user can pass a template value with the minimum they desire. By default this value is set to 11, because that minimum value was empirically seen to provide the best accuracy, up to 6 decimal places, for inputs that did not have enough fractional bits, for a PWL implementation with 4 segments and in the domain of [0, 1). For an example on how to pass this argument, please refer to the *Example Function Calls* section.

2.2.3. The ac_pow_pwl Implementation

In order to calculate the generic power function output for ac_fixed inputs, we use a relation similar to that used in the *ac_exp_pwl* implementation, namely

```
a^x = 2 ^ (x * log2(a))
```

The difference being that in this case, the base, i.e. *a*, is a dynamic input received from the user, hence we cannot use a constant value to represent *log2(a)*. We instead use the *ac_log2_pwl* function to calculate this value, and multiply the calculated value with the exponent, i.e. *x*. The product of this multiplication is passed to the *ac_pow2_pwl* implementation.

2.2.4. Function Prototypes

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

```
template<ac_q_mode pwl_Q = AC_TRN,
         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 = 11, ac_q_mode pwl_Q = AC_TRN,
          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
)
```

```
template<ac_q_mode pwl_Q = AC_TRN,
         int baseW, int baseI, ac_q_mode baseQ, ac_o_mode baseO,
         int exponW, int exponI, bool exponS, ac_q_mode exponQ, ac_o_mode ex-
ponO,
         int outW, int outI, ac_q_mode outQ, ac_o_mode outO>
void ac_pow_pwl(
  const ac_fixed<baseW, baseI, false, baseQ, baseO> &base,
  const ac_fixed<exponW, exponI, exponS, exponQ, exponO> &expon,
  ac_fixed<outW, outI, false, outQ, outO> &output
)
```

Note that the function only accepts unsigned ac_fixed datatypes for the output and base (in case of the *ac_pow_pwl* implementation). This corresponds to the behavior of the actual power functions, which have a range that covers only all positive real values, and whose base only covers the domain of positive real values. The behavior of our PWL functions in this sense is different than the behavior of the standard C math library functions, which accept signed doubles for both the input(s) and the output.

Returning by Value

All the power functions can return their 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 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_TRN, class T_in>
T_out ac_pow2_pwl(
    const T_in &input
)
```

```
template<class T_out, int n_f_b = 11, ac_q_mode pwl_Q = AC_TRN, class T_in>
T_out ac_exp_pwl(
    const T_in &input
)
```

```
template<class T_out, ac_q_mode pwl_Q = AC_TRN, class T_in_base, class T_in_ex-
pon>
T_out ac_pow_pwl(
  const T_in_base  &base,
  const T_in_expon &expon
)
```

2.2.5. 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<21, 12, false, AC_RND, AC_SAT> base_type;
typedef ac_fixed<20, 11,  true, AC_RND, AC_SAT> exp_type;
typedef ac_fixed<30, 15, false, AC_RND, AC_SAT> output_type;

base_type a = 2.5;
exp_type x = 2.875;
output_type y_exp, y_pow2, y_pow;
ac_pow2_pwl(x, y_pow2);  //Approximates y_pow2 = pow(2, x.to_double())
ac_exp_pwl(x, y_exp);    //Approximates y_exp  = exp(x)
ac_pow_pwl(a, x, y_pow); //Approximates y_pow  = pow(a, 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_exp);
```

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_pow2 = ac_pow2_pwl<output_type>(x);
y_exp  = ac_exp_pwl<output_type>(x);
y_pow  = ac_pow_pwl<output_type>(a, 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_pow2 = ac_pow2_pwl<output_type, AC_RND>(x);
y_exp  = ac_exp_pwl<output_type, 7, AC_RND>(x);
y_pow  = ac_pow_pwl<output_type, AC_RND>(a, 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 9 points/8 segments. 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 can be 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 six datatypes hence handled are (a) *ac_fixed*, (b) *ac_float*, (c) *ac_complex<ac_fixed>*,(d) *ac_complex<ac_float>*, (e) *ac_std_float* and (f) *ac_ieee_float*. It is the *ac_fixed* function that actually contains the code required for the PWL implementation. All the other functions rely upon the *ac_fixed* PWL implementation.

Handling Zero Input

The *ac_reciprocal_pwl* library provides a macro-enabled AC_ASSERT which will produce a run-time assert when a zero input is encountered. If this assert fails to kick in, additional functionality is provided that ensures output saturation when a zero input is encountered.

PWL Approximation Graph

To explain the closeness of the PWL approximation to the “real thing”, i.e. the actual, accurate implementation of the reciprocal function, the following graph compares the PWL output against the accurate function output:

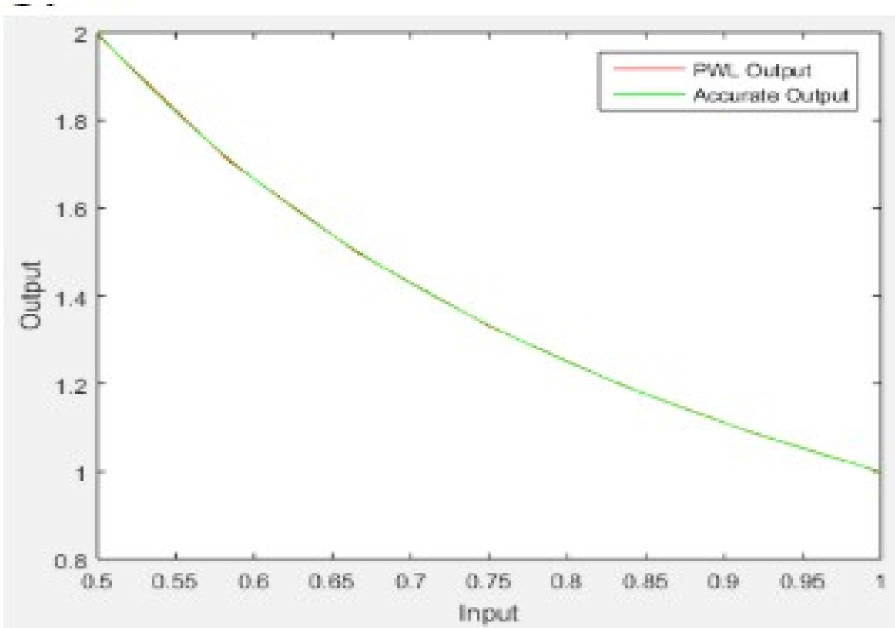


Illustration 4: PWL Output vs. Accurate Output for Reciprocal

ac_std_float and ac_ieee_float Support

The functions for *ac_std_float* and *ac_ieee_float* support serve as wrappers around the *ac_float* reciprocal designs, which perform conversions between the *ac_std_float*/*ac_ieee_float* float inputs/outputs and the intermediate *ac_float* variables that are passed to the *ac_float* design.

2.3.2. 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_TRN,
         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_TRN,
         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_TRN,
         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_TRN,
         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
);
```

```
template <ac_q_mode pwl_Q = AC_TRN, int W, int E, int outW, int outE>
void ac_reciprocal_pwl(
  const ac_std_float<W, E> &input,
  ac_std_float<outW, outE> &output
);
```

```
template<ac_q_mode pwl_Q = AC_TRN,
           ac_ieee_float_format Format,
           ac_ieee_float_format outFormat>
void ac_reciprocal_pwl(
  const ac_ieee_float<Format> &input,
  ac_ieee_float<outFormat> &output
);
```

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_TRN,
         class T_in>
T_out ac_reciprocal_pwl(
  const T_in &input
);
```

2.3.3. 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*.

Returning by Value

As mentioned earlier, the *ac_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_RND>(x);
```

2.4. High-accuracy Reciprocal (ac_reciprocal_pwl_ha)

The high-accuracy reciprocal library is designed to be a higher-accuracy version of the *ac_reciprocal_pwl* function. The implementation, function prototypes and example function calls for the *ac_reciprocal_pwl_ha* library are almost the same as that for the *ac_reciprocal_pwl* library, with the only two major differences being as follows:

- The function name is different; the high-accuracy reciprocal library has the suffix *_ha* attached to denote that it is a *h*igh-*a*ccuracy implementation.

- The high-accuracy version uses 32 segments to enable a higher-accuracy output, as compared to 8 segments for the original implementation. Hence, the high-accuracy version will consume more area in hardware. The 32-element LUTs for the high-accuracy version might be too big to be mapped to a register bank, and hence be mapped to memories instead. The user might want to consider changing the memory mapping threshold if they do not wish for this to happen.

Keeping these differences in mind, The user can consult the documentation for *ac_reciprocal_pwl* for more details on the working of the high-accuracy version.

2.5. Very High-accuracy Reciprocal (ac_reciprocal_pwl_vha)

For applications where an ever higher accuracy output is desired, the user can utilize the *ac_reciprocal_pwl_vha* library, which uses a 64-segment PWL. Like its high-accuracy reciprocal counterpart, this library is also very similar to the *ac_reciprocal_pwl* library, with the only major differences being the name and the number of segments used. This library is designed to have a percentage relative error below 0.005%.

2.6. Square Root (ac_sqrt_pwl)

The *ac_sqrt_pwl* function is a piecewise linear implementation of a square root function that has been optimized to provide a fast, low-area implementation with minimal error, making it useful as a basic building block in high speed IP blocks.

2.6.1. The ac_sqrt_pwl Implementation

The *ac_sqrt_pwl* library provides three overloaded functions for the calculation of the square root of real and complex inputs, each handling a different input datatype. The three datatypes hence handled are (a) *ac_fixed*, (b) *ac_float* (c) *ac_std_float* (d) *ac_ieee_float* and (e) *ac_complex<ac_fixed>*. It is the *ac_fixed* implementation that actually contains the code required for the PWL approximation.

PWL Approximation Graph

To explain the closeness of the PWL approximation to the “real thing”, i.e. the accurate implementation of the square root function, the following graph is provided:

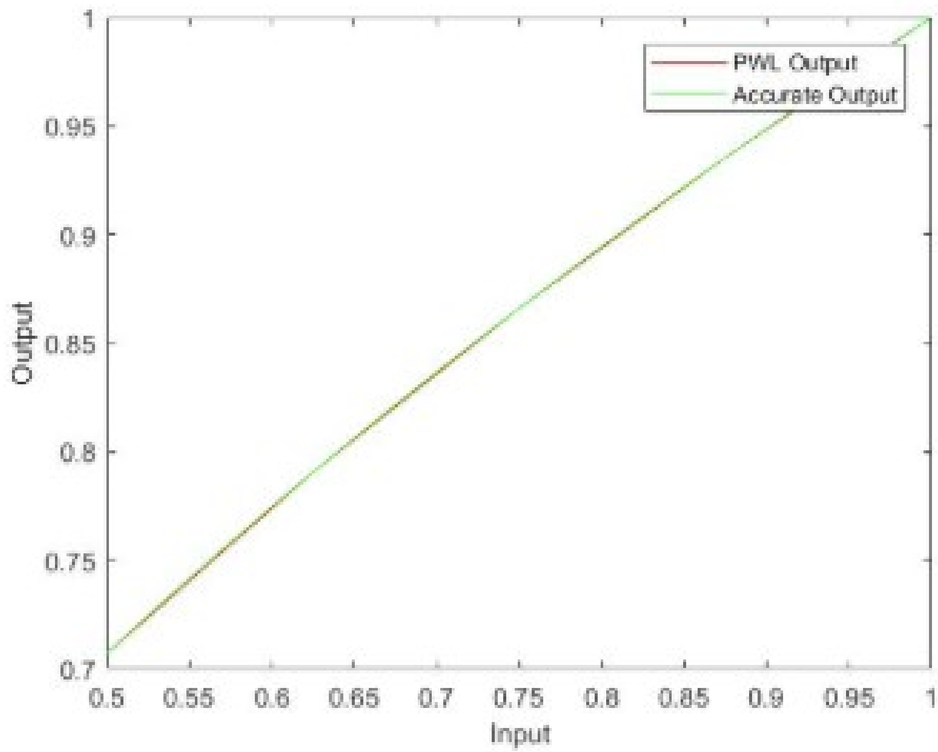


Illustration 5: PWL Output vs. Accurate Output for Square Root

Floating Point Support

The functions for *ac_std_float* and *ac_ieee_float* support serve as a wrapper around the *ac_float* square root design, and perform conversions between the *ac_std_float/ac_ieee_float* inputs/outputs and the intermediate *ac_float* variables that are passed to the *ac_float* design.

Handling Negative Inputs

A macro-enabled *AC_ASSERT* is provided in the *ac_float* implementation to alert the user to negative values that may accidentally be passed to the design.

Toggling Normalization

The floating point functions use the *ac_fixed* implementation to calculate the square root of the input mantissa. As the mantissa is already normalized, the fixed point function call is configured such that ac_normalize is not called and the generation of normalization hardware is bypassed. This is done by changing the value passed for the default *call_normalize* argument in the fixed point function interface. The interface, along with the *call_normalize* argument, is displayed in Function Templates. Refer to Example Function Calls for a usage example.

2.6.2. Function Templates

The following are the overloaded function prototypes for the *ac_sqrt_pwl* function for different datatypes:

```
template <ac_q_mode pwlQ = AC_TRN,
```

```
           int W, int I, ac_q_mode Q, ac_o_mode O,
           int outW, int outI, ac_q_mode outQ, ac_o_mode outO>
void ac_sqrt_pwl(
  const ac_fixed <W, I, false, Q, O> input,
  ac_fixed <outW, outI, false, outQ, outO> &output,
  const bool call_normalize = true
);
```

```
template <ac_q_mode pwlQ = AC_TRN,
          int W, int I, int E, ac_q_mode Q,
          int outW, int outI, int outE, ac_q_mode outQ>
void ac_sqrt_pwl(
  const ac_float <W, I, E, Q> input,
  ac_float <outW, outI, outE, outQ> &output
);
```

```
template <ac_q_mode pwl_Q = AC_TRN, int W, int E, int outW, int outE>
void ac_sqrt_pwl(
  const ac_std_float<W, E> &input,
  ac_std_float<outW, outE> &output
);
```

```
template <ac_q_mode pwl_Q = AC_TRN,
          ac_ieee_float_format Format, ac_ieee_float_format outFormat>
void ac_sqrt_pwl(
  const ac_ieee_float<Format> input,
  ac_ieee_float<outFormat> &output
);
```

```
template <ac_q_mode pwlQ = AC_TRN,
           int W, int I, ac_q_mode Q, ac_o_mode O,
          int outW, int outI, ac_q_mode outQ, ac_o_mode outO>
void ac_sqrt_pwl(
  const ac_complex <ac_fixed <W, I, true, Q, O> > input,
  ac_complex <ac_fixed <outW, outI, true, outQ, outO> > &output
);
```

Returning by Value

The *ac_sqrt_pwl* functions can return their output by value as well as by reference. To return the value, the user must pass the information of the output type to the function as a template argument. Refer to Example Function Calls for a usage example. The prototype for the function call to return by value is shown below:

```
template<class T_out, ac_q_mode pwlQ = AC_TRN, class T_in>
T_out ac_sqrt_pwl(
  const T_in &input
);
```

2.6.3. Example Function Calls

An example of a function call to store the value of the square root of a sample *ac_fixed* variable *x* in a variable *y*, by using both return by reference and return by value, is shown below. A line of code that illustrates the method to change the rounding mode of the intermediate PWL variable is also included.

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

input_type x = 1.75;
output_type y;

// Returns y = sqrt(x), and returns by reference.
ac_sqrt_pwl (x, y);

// Change the rounding mode for intermediate PWL variable to AC_RND
ac_sqrt_pwl<AC_RND> (x, y);

// The following line, returns by value instead of by reference.
y = ac_sqrt_pwl <output_type> (x);
```

If the input to the ac_fixed function is already normalized, the user can bypass normalization and save on hardware that way. To do so, they can pass *false* for the *call_normalize* argument, as shown below:

```
const bool call_normalize = false;
ac_sqrt_pwl(x, y, false);
```

2.7. Inverse Square Root (ac_inverse_sqrt_pwl)

The *ac_inverse_sqrt_pwl* function is a piecewise linear implementation of the inverse square root function (1/sqrt(x)) that has been optimized to provide a high-performance implementation with minimal error, making it useful as a basic building block in high speed IP blocks. This function is implemented as an overloaded function for the different datatypes it supports. It provides a more accurate and lower-area output as compared to using the square root and reciprocal PWL approximations together to calculate the inverse square root of a number.

2.7.1. The ac_inverse_sqrt_pwl Implementation

The *ac_inverse_sqrt_pwl* library provides four overloaded functions for the calculation of the inverse square root of real and complex inputs. Each overloaded function handles a different input datatype. The five datatypes hence handled are (a) *ac_fixed,* (b) *ac_float* (c) *ac_std_float*, (d) *ac_ieee_float* and (e) *ac_complex<ac_fixed>*. It is the *ac_fixed* function that actually contains the code required for the PWL implementation. All the other functions rely upon the *ac_fixed* PWL implementation.

Handling Invalid Inputs

The *ac_inverse_sqrt_pwl* library provides a macro-enabled AC_ASSERT which will produce a run-time assert when a zero input is encountered. If this assert fails to kick in, additional functionality is provided that ensures output saturation when a zero input is encountered.

A macro-enabled AC_ASSERT is also provided that is triggered in case of a negative floating point input.

PWL Approximation Graph

To explain the closeness of the PWL approximation to the “real thing”, i.e. the accurate implementation of the inverse square root function, the following graph is provided:

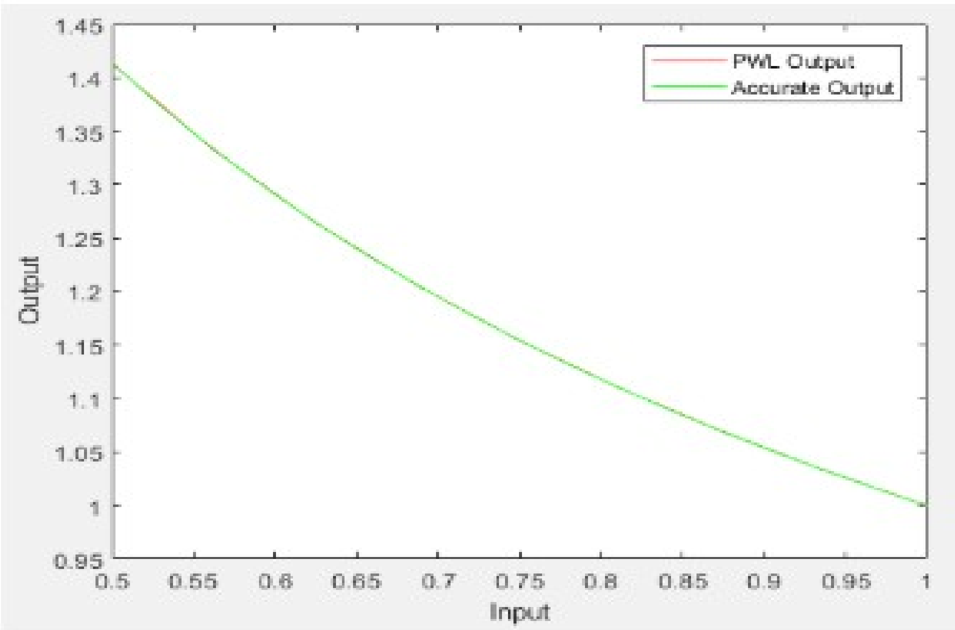


Illustration 6: PWL Output vs. Accurate Output for Inverse Square Root

Floating Point Support

The functions for *ac_std_float* and *ac_ieee_float* support serves as a wrapper around the *ac_float* inverse square root design, which performs conversions between the standard/IEEE float inputs/outputs and the intermediate *ac_float* variables that are passed to the *ac_float* design.

Toggling Normalization

The floating point functions use the *ac_fixed* implementation to calculate the square root of the input mantissa. As the mantissa is already normalized, the fixed point function call is configured such that ac_normalize is not called and the generation of normalization hardware is bypassed. This is done by changing the value passed for the default *call_normalize* argument in the fixed point function interface. The interface, along with the *call_normalize* argument, is displayed in Function Templates. Refer to Example Function Calls for a usage example.

2.7.2. Function Templates

The following are the overloaded function prototypes for the *ac_inverse_sqrt_pwl* function for different datatypes:

```
template<ac_q_mode q_mode_temp = AC_TRN,
         int W1, int I1, ac_q_mode q1, ac_o_mode o1,
         int W2, int I2, ac_q_mode q2, ac_o_mode o2>
void ac_inverse_sqrt_pwl(
  const ac_fixed <W1,I1, false, q1, o1> &input,
  ac_fixed <W2, I2, false, q2, o2> &output,
  const bool call_normalize = true
);
```

```
template <ac_q_mode q_mode_temp = AC_TRN,
          int W1, int I1, int E1, ac_q_mode q1,
          int W2, int I2, int E2, ac_q_mode q2>
void ac_inverse_sqrt_pwl(
  const ac_float <W1, I1, E1, q1> &input,
  ac_float <W2, I2, E2, q2> &output
);
```

```
template <ac_q_mode q_mode_temp = AC_TRN, int W1, int E1, int W2, int E2>
void ac_inverse_sqrt_pwl(
    const ac_std_float<W1, E1> &input,
    ac_std_float<W2, E2> &output
);
```

```
template<ac_q_mode q_mode_temp = AC_TRN,
         ac_ieee_float_format Format,
         ac_ieee_float_format outFormat>
void ac_inverse_sqrt_pwl(
```

```
  const ac_ieee_float<Format> input,
  ac_ieee_float<outFormat> &output
);
```

```
template<ac_q_mode q_mode_temp = AC_TRN,
         int W1, int I1, ac_q_mode q1, ac_o_mode o1,
         int W2, int I2, ac_q_mode q2, ac_o_mode o2>
void ac_inverse_sqrt_pwl(
  const ac_complex <ac_fixed <W1,I1,true, q1, o1> > &input,
  ac_complex <ac_fixed <W2, I2, true, q2, o2> > &output
);
```

Returning by Value

The *ac_inverse_sqrt_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 as follows:

```
template<class T_out, ac_q_mode q_mode_temp = AC_TRN, class T_in>
T_out ac_inverse_sqrt_pwl(
  const T_in &input
)
```

2.7.3. Example Function Calls

An example of a function call to store the value of the inverse square root of a sample *ac_fixed* variable *x* in a variable *y*, by using both return by reference and return by value, is shown below.

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

input_type x = 1.75;
output_type y;

// Returns y = 1/sqrt(x), and returns by reference.
ac_inverse_sqrt_pwl (x, y);

// Change the rounding mode for intermediate PWL variable to AC_RND
ac_inverse_sqrt_pwl<AC_RND> (x, y);

// The following line, returns by value instead of by reference.
y = ac_inverse_sqrt_pwl <output_type> (x);
```

If the input to the ac_fixed function is already normalized, the user can bypass normalization and save on hardware that way. To do so, they can pass *false* for the *call_normalize* argument, as shown below:

```
const bool call_normalize = false;
ac_inverse_sqrt_pwl(x, y, false);
```

2.8. Very High-Accuracy Inv Sqrt (ac_inverse_sqrt_pwl_vha)

The very high-accuracy inverse square root library is designed to be a higher-accuracy version of the *ac_inverse_sqrt_pwl* function. The implementation, function prototypes and example function calls for the *ac_inverse_sqrt_pwl_vha* library are almost the same as that for the *ac_inverse_sqrt_pwl_vha* library, with the two major differences being as follows:

- The function name is different; the very high-accuracy inverse square root library has the suffix *_vha* attached to denote that it is a *v*ery *h*igh-*a*ccuracy implementation.
- The VHA version uses 64 segments to enable a higher-accuracy output, as compared to 8 segments for the original implementation. Hence, the high-accuracy version will consume more area in hardware. The 64-element LUTs for the high-accuracy version might be too big to be mapped to a register bank, and hence be mapped to memories instead. The user might want to consider changing the memory mapping threshold if they do not wish for this to happen.

Keeping these differences in mind, The user can consult the documentation for *ac_inverse_sqrt_pwl* for more details on the working of the very high-accuracy version.

2.9. Tangent (ac_tan_pwl)

The *ac_tan_pwl* library provides a fast, PWL implementation of the tangent function of first-quadrant angles with minimal inaccuracy. The implementation is faster than other algorithms such as the CORDIC algorithm, which require a significant number of iterations and stored table values to reach an acceptable accuracy in the output. The tangent PWL library supports (a) *ac_fixed* (b) *ac_float* (c) *ac_std_float* and (d) *ac_ieee_float* datatypes.

2.9.1. The ac_tan_pwl Implementation

Each supported datatype is handled separately through overloaded functions. The *ac_fixed* implementation contains the actual PWL implementation, and the *ac_float* implementation depends on the *ac_fixed* implementation to perform the PWL calculation.

Normalization

The actual tangent PWL implementation only covers the domain of [0, π/4). In order to cover a larger range of first-quadrant angle values, i. e. in order to extend the domain of the *ac_tan_pwl* function to [π/4, π/2), the input angle in radians is halved using a shift operation, and the following equation is used:

tan(x) = 2 * tan(x/2) * (1 / (1 – tan(x/2)^2))

In order to calculate the 1 / (1 – tan(x/2)^2) factor in the above equation, we use the *ac_reciprocal_pwl* library for ac_fixed inputs and outputs. Hence, the user must keep in mind that any changes made to that library will also affect the accuracy of the tangent approximation for values that equal or exceed π/4.

Saturation

It is important to note that the output will saturate to the maximum possible value once the input has crossed the value of 1.556640625 radians, which is roughly 89.2 degrees.

PWL Approximation Graph

To explain the closeness of the PWL approximation to the "real thing", i.e. the accurate implementation of the tangent function, the following graph is provided for the domain [0, π/4):

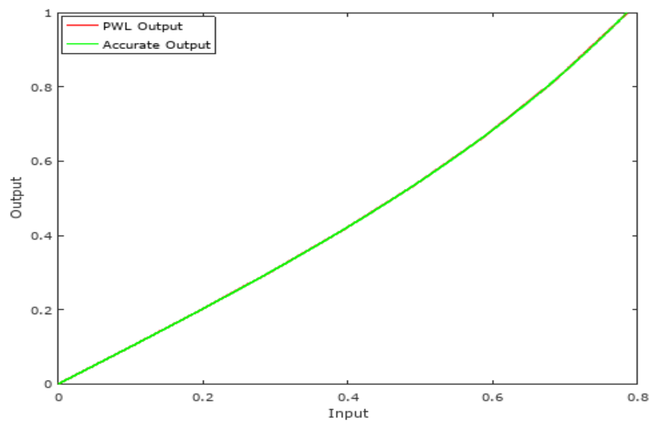


Illustration 7: PWL Output vs. Accurate Output for Tangent

Floating Point Support

The floating point designs use the *ac_shift_left* library to convert the values of input mantissa and exponent into a fixed point value that can be passed to the *ac_fixed* implementation. The bitwidth of the intermediate fixed point value can be reduced due to the fact that the input cannot exceed π/2 and because the *ac_fixed* PWL function truncates inputs that exceed a certain number of fractional bits.

The *ac_std_float* and *ac_ieee_float* functions serves as a wrapper around the *ac_float* function that converts *ac_std_float* and *ac_ieee_float* inputs/outputs to/from their equivalent *ac_float* representations.

Handling Invalid Inputs

Macro-enabled *AC_ASSERT*s are provided to alert the users to inputs that exceed π/2 or, in case of floating point implementations, negative inputs.

2.9.2. Function Templates

The following are the function prototypes for the *ac_tan_pwl* function for different datatypes:

```
template<ac_q_mode pwl_Q = AC_TRN,
         int W, int I, ac_q_mode Q, ac_o_mode O,
         int outW, int outI, ac_q_mode outQ, ac_o_mode outO>
void ac_tan_pwl(
  const ac_fixed<W, I, false, Q, O> &input,
  ac_fixed<outW, outI, false, outQ, outO> &output
);
```

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

```
template <ac_q_mode pwl_Q = AC_TRN, int W, int E, int outW, int outE>
void ac_sqrt_pwl(
  const ac_std_float<W, E> &input,
  ac_std_float<outW, outE> &output
);
```

```
template<ac_q_mode pwl_Q = AC_TRN,
         ac_ieee_float_format Format,
         ac_ieee_float_format outFormat>
void ac_tan_pwl(
  const ac_ieee_float<Format> &input,
  ac_ieee_float<outFormat> &output
);
```

Returning by Value

The *ac_tan_pwl* function can return its 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_TRN, class T_in>
T_out ac_tan_pwl(const T_in &input);
```

2.9.3. Example Function Calls

An example of a function call to store the value of the tangent of a sample *ac_fixed* variable *x* in a variable *y*, by using both return by reference and return by value, is shown below.

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

input_type x = 0.25;
output_type y;

// Returns y = tan(x), and returns by reference.
ac_tan_pwl (x, y);

// Change the rounding mode for intermediate PWL variable to AC_RND
ac_tan_pwl<AC_RND> (x, y);

// The following line, returns by value instead of by reference.
y = ac_tan_pwl <output_type> (x);
```

2.10. Arctangent (ac_atan_pwl)

The *ac_atan_pwl* library provides a PWL based approximation of the arctangent function for positive (a) *ac_fixed* (b) *ac_float* (c) *ac_std_float* and (d) *ac_ieee_float* inputs.

2.10.1. The ac_atan_pwl Implementation

The *ac_atan_pwl* has overloaded functions to support the different datatypes listed earlier. The *ac_fixed* version is the one that contains the actual PWL implementation, and the floating point versions depend on it for their functioning.

Normalization

The actual arctangent PWL implementation only covers the domain of [0, 1). This is done not only because the domain from [1, ∞) can be covered with the reciprocal function, but because attempting to cover a larger domain with saturation for input values that cross a certain limit means that:

- It's harder to obtain an adequate fit to the actual arctangent function, due to the nature of the function curve, necessitating more segments and a larger area to store LUT values.
- A comparator must be used for saturation, resulting in a further increase in area.

The formula used for any values that exceed unity is as follows:

$\text{atan}(x) = \pi/2 - \text{atan}(1/x)$

Where the value of 1/x is obtained using the reciprocal PWL function.

An input of unity is handled by storing it in a variable which saturates to a value slightly less than unity, hence ensuring that the normalized input value is still within the domain of the PWL function.

Floating Point Support

The functions for *ac_std_float* and *ac_ieee_float* support serves as a wrapper around the *ac_float* inverse square root design, which performs conversions between the standard/IEEE float inputs/outputs and the intermediate *ac_float* variables that are passed to the *ac_float* design.

PWL Approximation Graph

To explain the closeness of the PWL approximation to the "real thing", i.e. the accurate implementation of the arctangent function, the following graph is provided for the domain [0, 1):

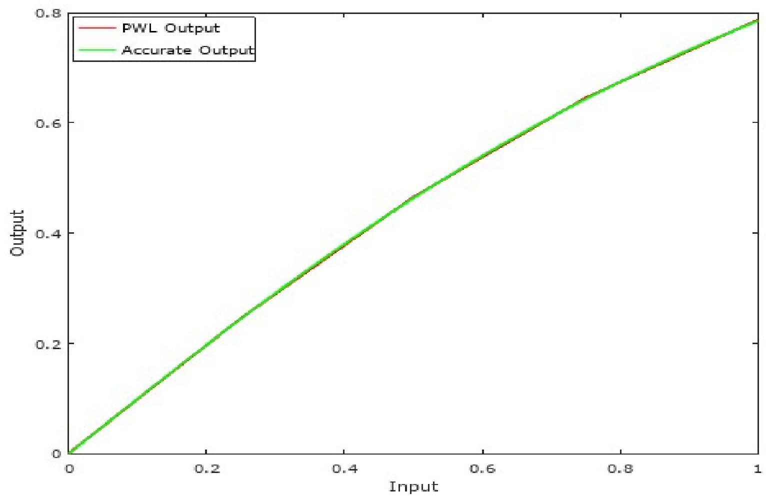


Illustration 8: PWL vs Accurate Output for Arctangent

2.10.2. Function Templates

The following is the function prototype for the *ac_atan_pwl* function for different datatypes:

```
template<ac_q_mode pwl_Q = AC_TRN,
         int W, int I, ac_q_mode Q, ac_o_mode O,
         int outW, int outI, ac_q_mode outQ, ac_o_mode outO>
void ac_atan_pwl(
  const ac_fixed<W, I, false, Q, O> &input,
  ac_fixed<outW, outI, false, outQ, outO> &output
);
```

```
template<ac_q_mode pwl_Q = AC_TRN,
         int W, int I, int E, ac_q_mode Q,
         int outW, int outI, int outE, ac_q_mode outQ>
```

```
void ac_atan_pwl(
  const ac_float<W, I, E, Q> &input,
  ac_float<outW, outI, outE, outQ> &output
);
```

```
template <ac_q_mode pwl_Q = AC_TRN, int W, int E, int outW, int outE>
void ac_atan_pwl(
    const ac_std_float<W, E> &input,
    ac_std_float<outW, outE> &output
);
```

```
template<ac_q_mode pwl_Q = AC_TRN,
         ac_ieee_float_format Format,
         ac_ieee_float_format outFormat>
void ac_atan_pwl(
  const ac_ieee_float<Format> &input,
  ac_ieee_float<outFormat> &output
);
```

Returning by Value

The *ac_atan_pwl* function can return its 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_TRN,
         class T_in>
T_out ac_atan_pwl(const T_in &input);
```

2.10.3. Example Function Calls

An example function call for *ac_fixed* variables is given below.

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

input_type x = 0.5;
output_type y;

// Returns y = atan(x), and returns by reference.
ac_atan_pwl (x, y);
```

```
// Change the rounding mode for intermediate PWL variable to AC_RND
ac_atan_pwl<AC_RND> (x, y);

// The following line returns by value instead of by reference.
y = ac_atan_pwl <output_type> (x);
```

2.11. High-accuracy Arctangent (ac_atan_pwl_ha)

The high-accuracy arctangent library is designed to be a higher-accuracy version of the *ac_atan_pwl* function. The implementation, function prototypes and example function calls for the *ac_atan_pwl_ha* library are almost the same as that for the *ac_atan_pwl* library, with the only three major differences being as follows:

- The function name is different; the high-accuracy arctangent library has the suffix *_ha* attached to denote that it is a *h*igh-*a*ccuracy implementation.
- The high-accuracy version uses 32 segments to enable a higher-accuracy output, as compared to 4 segments for the original implementation. Hence, the high-accuracy version will consume more area in hardware. The 32-element LUTs for the high-accuracy version might be too big to be mapped to a register bank, and hence be mapped to memories instead. The user might want to consider changing the memory mapping threshold if they do not wish for this to happen.
- The high-accuracy reciprocal PWL is called to enable the calculation of atan(1/x).

Keeping this differences in mind, the user can consult the documentation for *ac_atan_pwl* for more details on the working of the high-accuracy version.

2.12. Very High-accuracy Arctangent (ac_atan_pwl_vha)

For applications where an ever higher accuracy output is desired, the user can utilize the *ac_atan_pwl_vha* library, which uses a 64-segment PWL. Like its high-accuracy arctangent counterpart, this library is also very similar to the *ac_atan_pwl* library, with the only major differences being the name, the number of segments used, and the reciprocal PWL library called (*ac_reciprocal_pwl_vha*). This library is designed to have an absolute error below 5e-5.

2.13. Sigmoid (ac_sigmoid_pwl)

The *ac_sigmoid_pwl* library provides a PWL based approximation of the sigmoid function for *ac_fixed* inputs with minimal inaccuracy.

Normalization

The actual sigmoid PWL implementation only covers the domain of [0, 5). This pwl domain was selected because attempting to cover a larger domain makes it harder to obtain an adequate fit to the actual sigmoid function necessitating more segments and a larger area to store LUT values. As the input approaches infinity, the sigmoid function saturates and thus for inputs greater than 5, the output is saturated to the maximum possible pwl output. For negative inputs, the output will simply be 1 - the output of its positive conterpart taking advantage of the symmetry of the sigmoid function.

PWL Approximation Graph

To explain the closeness of the PWL approximation to the “real thing”, i.e. the accurate implementation of the sigmoid function, the following graph is provided for the domain [0, 5) :

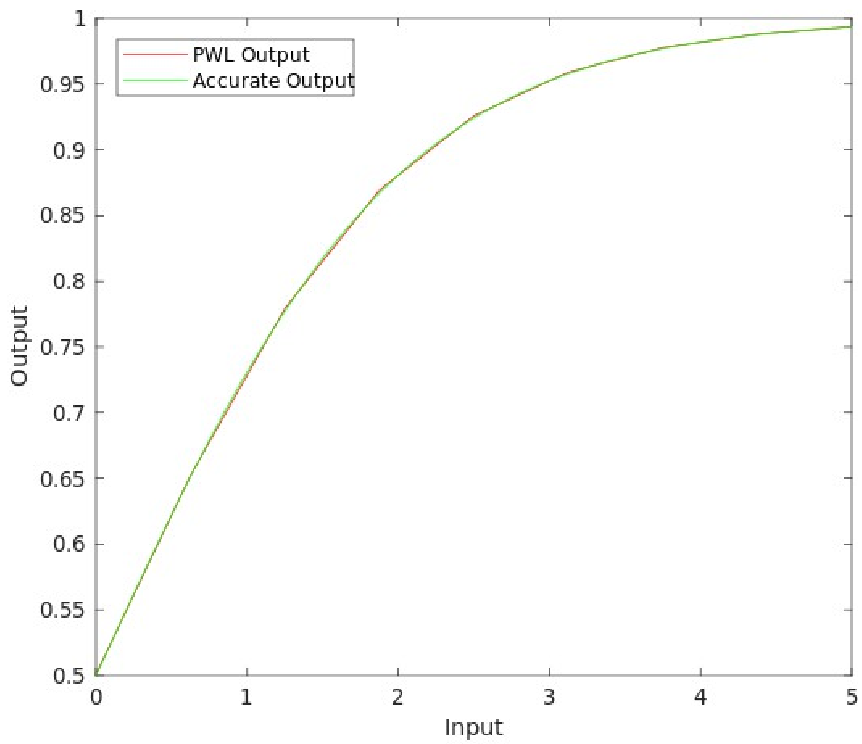


Illustration 9: PWL Output vs. Accurate Output for Sigmoid

2.13.1. Function Templates

The following is the function prototype for the *ac_sigmoid_pwl* function for *ac_fixed* datatypes:

```
template<ac_q_mode pwl_Q = AC_TRN,
         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_sigmoid_pwl(
  const ac_fixed<W, I, S, Q, O> &input,
  ac_fixed<outW, outI, false, outQ, outO> &output
);
```

Returning by Value

The *ac_sigmoid_pwl* function can return its 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 2.13.2Example Function Calls section below. The prototype for the function which returns by value is shown below:

```
template<class T_out,
         ac_q_mode pwl_Q = AC_TRN,
         class T_in>
T_out ac_sigmoid_pwl(const T_in &input);
```

2.13.2. Example Function Calls

An example of a function call to store the value of sigmoid of a sample *ac_fixed* variable *x* in a variable *y*, by using both return by reference and return by value, is shown below.

```
typedef ac_fixed<15, 7, true, AC_RND, AC_SAT> input_type;
typedef ac_fixed<30, 3, false, AC_RND, AC_SAT> output_type;

input_type x = 2.5;
output_type y;

// Returns y = sigmoid(x), and returns by reference.
ac_sigmoid_pwl (x, y);

// Change the rounding mode for intermediate PWL variable to AC_RND
ac_sigmoid_pwl<AC_RND> (x, y);

// The following line returns by value instead of by reference.
y = ac_sigmoid_pwl <output_type> (x);
```

2.14. Hyperbolic Tangent (ac_tanh_pwl)

The *ac_tanh_pwl* library provides a PWL based approximation of the hyperbolic tangent function for *ac_fixed* inputs with minimal inaccuracy.

Normalization

The actual hyperbolic tangent PWL implementation only covers the domain of [0, 3). This pwl domain was selected because it gives a good fit to the actual hyperbolic tangent function. As the input approaches infinity, the hyperbolic tangent function saturates and thus for inputs greater than 3, the output is saturated to the maximum possible pwl output. For negative inputs, the output will simply be the negation of the output of its positive conterpart taking advantage of the symmetry of the hyperbolic tangent function.

PWL Approximation Graph

To explain the closeness of the PWL approximation to the accurate implementation of the hyperbolic tangent function, the following graph is provided for the domain [0, 3) :

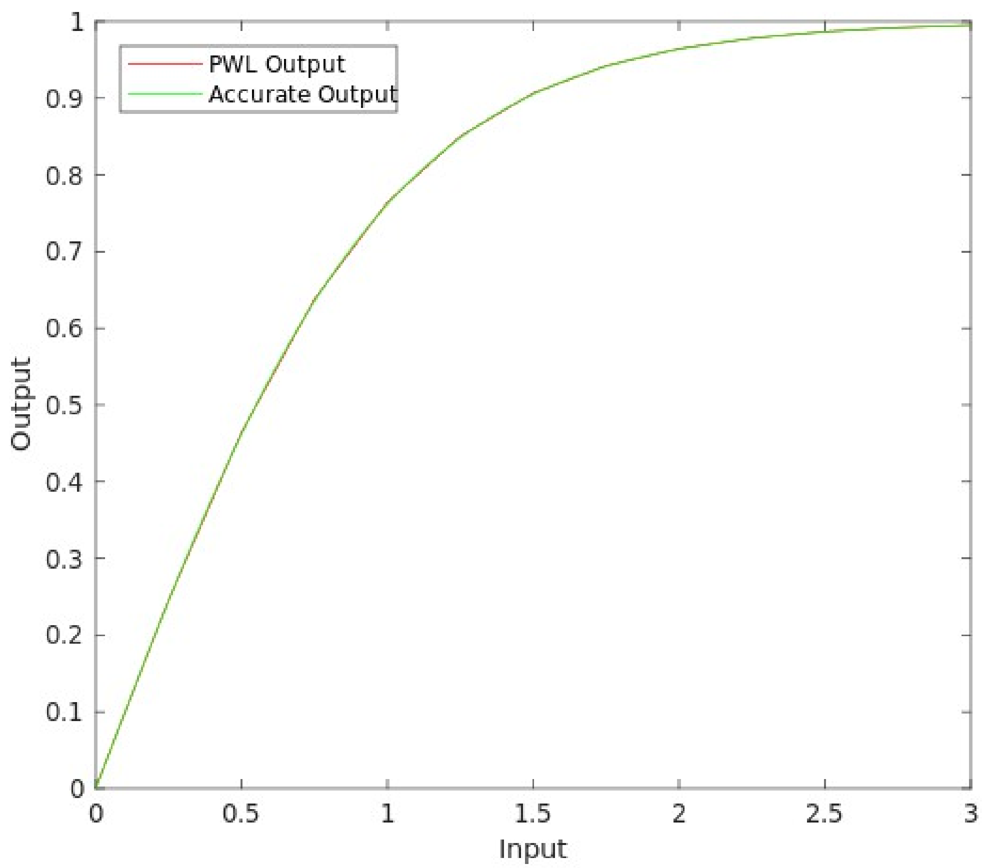


Illustration 10: PWL Output vs. Accurate Output for Hyperbolic Tangent

2.14.1. Function Templates

The following is the function prototype for the *ac_tanh_pwl* function for *ac_fixed* datatypes:

```
 template<ac_q_mode pwl_Q = AC_TRN,
         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_tanh_pwl(
  const ac_fixed<W, I, S, Q, O> &input,
  ac_fixed<outW, outI, outS, outQ, outO> &output
);
```

Returning by Value

The *ac_tanh_pwl* function can return its 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 2.13.2Example Function Calls section below. The prototype for the function which returns by value is shown below:

```
template<class T_out,
         ac_q_mode pwl_Q = AC_TRN,
         class T_in>
T_out ac_tanh_pwl(const T_in &input);
```

2.14.2. Example Function Calls

An example of a function call to store the value of hyperbolic tan of a sample *ac_fixed* variable *x* in a variable *y*, by using both return by reference and return by value, is shown below.

```
typedef ac_fixed<15, 7, true, AC_RND, AC_SAT> input_type;
typedef ac_fixed<30, 3, false, AC_RND, AC_SAT> output_type;

input_type x = 2.5;
output_type y;

// Returns y = tanh(x), and returns by reference.
ac_tanh_pwl (x, y);

// Change the rounding mode for intermediate PWL variable to AC_RND
ac_tanh_pwl<AC_RND> (x, y);

// The following line returns by value instead of by reference.
y = ac_tanh_pwl <output_type> (x);
```

2.15. Softmax (ac_softmax_pwl)

The *ac_softmax_pwl* library provides a PWL based approximation of the softmax function for an array of *ac_fixed* inputs with minimal inaccuracy.

Intermediate Datatypes

Three intemediate datatypes are important for the computation of the softmax output:

- *T_exp*: Determines the precision for the exponent values of each input, calculated through the *ac_exp_pwl* function. Depends on the input bitwidth, by default. The default derived bitwidth, specifically the derived integer bitwidth, can be very large, hence a *static_assert* is provided in the function to alert the user if it exceeds 64 bits, at compile time and thereby limit the area.
- *T_sum*: Determines the precision for the sum of exponent values. Depends on the bitwidth of *T_exp*.
- *T_recip*: Determines the precision of the reciprocal of the sum above, calculated through the ac_reciprocal_pwl function. Depends on the bitwidth of *T_sum*, by default.

All the above datatypes are unsigned, fixed-point datatypes.

Loops and Architectural Exploration

The function contains three loops with the following labels:

- *CALC_EXP_LOOP*: Loops through all input array elements and calculates exponent value of each.
- *SUM_EXP_LOOP*: Loops through all exponent values calculated above and finds their sum.
- *CALC_SOFTMAX_LOOP*: Finds final softmax value by multiplying the exponent values of each input with the reciprocal of their sum.

All these loops can be pipelined and/or unrolled, contingent upon other factors like clock constraints and datatypes involved. If the main function as well as all loops in the design are pipelined with an II of 1, one can achieve a throughput of *2K* clock cycles, where *K* is the length of the input/output vector. If the main function is pipelined with an II of 1 and all loops fully unrolled, a throughput of one clock cycle can be achieved. Care should be taken in doing the latter, especially in unrolling the *CALC_SOFTMAX_LOOP*. Doing means that the design now uses *K* multipliers which may be very large, depending on the size of *T_recip* and *T_exp*. If the area is very large, the user is advised to either consider partially or completely rolling *CALC_SOFTMAX_LOOP*, or to use template parameters to override and reduce the default *T_recip* and *T_exp* bitwidths and hence reduce the area. Refer to the Changing Intermediate Bitwidths section below, for more details.

2.15.1. Function Templates

The following is the function prototype for the *ac_softmax_pwl* function for *ac_fixed* datatypes:

```
template<ac_q_mode pwl_Q=AC_TRN,
         bool or_e=false,
         int iW_e=0, int iI_e=0, ac_q_mode iQ_e=AC_TRN, ac_o_mode iO_e=AC_WRAP,
         bool or_r=false,
         int iW_r=0, int iI_r=0, ac_q_mode iQ_r=AC_TRN, ac_o_mode iO_r=AC_WRAP,
         int K,
         int W, int I, bool S, int Q, ac_q_mode Q, ac_o_mode O,
         int outW, int outI, ac_q_mode outQ, ac_o_mode outO
         >
void ac_softmax_pwl(
  const ac_fixed<W, I, S, Q, O> (&input)[K],
  const ac_fixed<outW, outI, false, outQ, outO> (&output)[K]
);
```

Changing Intermediate Bitwidths

As mentioned above, the user might want to modify the bitwidths of *T_exp* or *T_recip*. To do so, they must set the *or_e* (for *T_exp*) or *or_r* (for *T_recip*) parameters to true, and assign values to the template parameters with the suffix of *_e* (for *T_exp*) or the suffix of *_r* (for *T_recip*). Please refer to the Example Function Calls section for an example.

2.15.2. Example Function Calls

The following code snippet shows the user how to call the *ac_softmax_pwl* function in various different configurations, for an array of 4 inputs/outputs.

```
ac_fixed<20, 4, true> input[4];
ac_fixed<17, 1, false> output[4];
// Initialize input array.
input[0] = 0.5;
input[1] = -2.4;
input[2] = 3.2;
input[3] = -1.9;
// Call ac_softmax_pwl function with default configuration
ac_softmax_pwl(input, output);
// Call ac_softmax_pwl function, with AC_RND as the rounding type for the PWL
output,
```

```
// ac_fixed<32, 16, false, AC_TRN, AC_SAT> as the type for T_exp and the de-
fault
// T_recip type.
ac_softmax_pwl<AC_RND, true, 32, 16, AC_TRN, AC_SAT>(input, output);
```

Chapter 3: CORDIC Math Functions

The CORDIC-based functions in the *ac_math* library include the following operations:

- Sine/Cosine (ac_sin_cordic/ ac_cos_cordic)
- Arcsin/Arccos (ac_arcsin_cordic/ ac_arccos_cordic)
- Arctangent (ac_atan2_cordic)
- Exponential (ac_exp_cordic/ ac_exp2_cordic)
- Logarithm (ac_log_cordic/ ac_log2_cordic)
- Power (ac_pow_cordic)

3.1. Sine/Cosine (ac_sin_cordic/ ac_cos_cordic)

There are algorithms that require the computation of the *ac_sin_cordic()* and/or ac_*cos_cordic()* functions for dynamic angles (as opposed to a set of predefined angles as in FFTs which are best implemented as a table lookup). In such cases, the CORDIC algorithm is often used as a way to compute the *ac_sin_cordic()* or *ac_cos_cordic()* functions. The math library contains a sample CORDIC implementation using fixed-point data types. The available functional interfaces compute sin, cos or both scaled sin and cos with one call (more efficient hardware than having separate calls to sin and cos).

The first argument is the angle scaled by (*1/PI*). The advantage of the prescaled angle is that it makes the quadrant computation inside the CORDIC algorithm trivial and it also simplifies the complexity of the caller as in most cases the calls to sin and cos are made with angles that are multiples of PI.

AC: ac_fixed
`void ac_sin_cordic ( ac_fixed<AW,AI,true,AQ,AO> angle_over_pi, ac_fixed<OW,OI,true,OQ,OO> &sin )`
`void ac_cos_cordic ( ac_fixed<AW,AI,true,AQ,AO> angle_over_pi, ac_fixed<OW,OI,true,OQ,OO> &cos )`
`void ac_sincos_cordic ( ac_fixed<AW,AI,true,AQ,AO> angle_over_pi, ac_fixed<OW,OI,true,OQ,OO> C, ac_fixed<OW,OI,true,OQ,OO> &sin, ac_fixed<OW,OI,true,OQ,OO> &cos )`
SystemC: sc_fixed
`void ac_sin_cordic ( sc_fixed<AW,AI,AQ,AO> angle_over_pi, sc_fixed<OW,OI,OQ,OO> &sin )`
`void ac_cos_cordic ( sc_fixed<AW,AI,AQ,AO> angle_over_pi, sc_fixed<OW,OI,OQ,OO> &cos )`
`void ac_sincos_cordic ( sc_fixed<AW,AI,AQ,AO> angle_over_pi, sc_fixed<OW,OI,OQ,OO> C, sc_fixed<OW,OI,OQ,OO> &sin, sc_fixed<OW,OI,OQ,OO> &cos )`

Table 1: Functions for Sine, Cosine and Combined Sine and Cosine with Amplitude

Note that the interface that computes both *sine* and *cosine*, the data types for *C*, *sin*, *cos* parameters must be identical.

3.2. Arcsin/Arccos (ac_arcsin_cordic/ ac_arccos_cordic)

The *ac_math* library contains sample CORDIC implementation using fixed-point data types. The available functional interfaces compute the inverse trigonometric functions arcsin and arccos.

The result of these functions is scaled by (1/PI), which is often the case in practice, maintaining consistency with the sin/cos implementations.

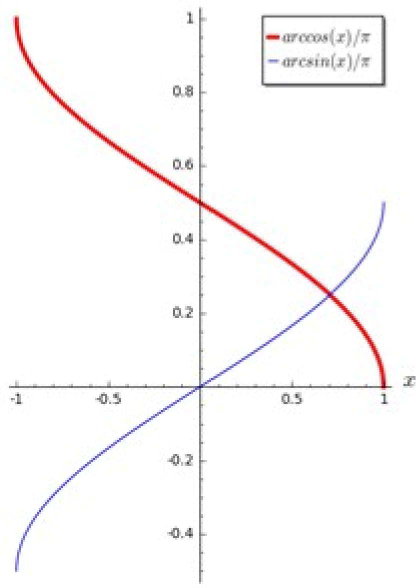


Illustration 11: Graph of scaled arcsin and arccos functions

AC: ac_fixed

```
void ac_arccos_cordic(
  ac_fixed<AW,AI,true,AQ,AO> x,
  ac_fixed<OW,OI,false,OQ,OO> &arccos_x_over_pi
);

void ac_arcsin_cordic(
  ac_fixed<AW,AI,true,AQ,AO> x,
  ac_fixed<OW,OI,true,OQ,OO> &arcsin_x_over_pi
);
```

SystemC: sc_fixed*

```
void ac_arccos_cordic(
  sc_fixed<AW,AI,AQ,AO> x,
  sc_ufixed<OW,OI,OQ,OO> &arccos_x_over_pi
```

```
);

void ac_arcsin_cordic(
  sc_fixed<AW,AI,AQ,AO> x,
  sc_fixed<OW,OI,OQ,OO> &arcsin_x_over_pi
);
```

*Note: only supports up to 20 fractional bits, i.e., AW - AI <= 20.

3.3. Arctangent (ac_atan2_cordic)

The *ac_atan2_cordic* function is a fixed-point CORDIC implementation of the functionality of the corresponding *ac_math.h* function. It takes two signed fixed-point arguments called *y* and *x* and it returns the arc tangent in the range *-PI* to *PI*. The accuracy of the computation is directly dependent on the precision of the fixed-point *atan* variable passed to the function.

AC: ac_fixed	SystemC: sc_fixed

`void ac_atan2_cordic(`
`  ac_fixed<YW,YI,true,YQ,YO> y,`
`  ac_fixed<XW,XI,true,XQ,XO> x,`
`  ac_fixed<OW,OI,true,OQ,OO> &atan`
`)`
	`void ac_atan2_cordic(`
	`  sc_fixed<YW,YI,YQ,YO> y,`
	`  sc_fixed<XW,XI,XQ,XO> x,`
	`  sc_fixed<OW,OI,OQ,OO> &atan`
	`)`

Table 2: Functions for Atan2

The behavior when one or both of the arguments is zero is identical with that of the *ac_atan2_cordic* in *ac_math.h*:

- 0 when x == 0 and y == 0
- -PI/2 when x == 0 and y < 0
- +PI/2 when x == 0 and y > 0

3.4. Exponential (ac_exp_cordic/ ac_exp2_cordic)

The *ac_math* library contains sample CORDIC implementations of exponentials. The *exp* function evaluates the exponential of a fixed point argument *x*, i.e., the transcendental number '*e*' raised to the power of *x*. The *exp2* function evaluates the number 2 raised to the power of a fixed point argument *x*.

The following code shows the function declarations for the AC Numerical Datatypes (*ac_fixed*). There are no SystemC variants.

```
void ac_exp_cordic( const ac_fixed<AW,AI,AS,AQ,AV> &x,
                    ac_fixed<ZW,ZI,false,ZQ,ZV> &z);
```

```
void ac_exp2_cordic( const ac_fixed<AW,AI,AS,AQ,AV> &x,
                     ac_fixed<ZW,ZI,false,ZQ,ZV> &z);
```

3.5. Logarithm (ac_log_cordic/ ac_log2_cordic)

The *ac_math* library contains sample CORDIC implementations of logarithms. The 'log' function evaluates the natural logarithm of a fixed point argument x, i.e., the logarithm with transcendental number '*e*' as its base. The log2 function evaluates the base 2 logarithm of argument *x*.

The following code shows the function declarations for the AC Numerical Datatypes (*ac_fixed*). There are no SystemC variants.

```
void ac_log_cordic( const ac_fixed<AW,AI,false,AQ,AV> &x,
                    ac_fixed<ZW,ZI,ZS,ZQ,ZV> &z);

void ac_log2_cordic( const ac_fixed<AW,AI,false,AQ,AV> &x,
                     ac_fixed<ZW,ZI,ZS,ZQ,ZV> &z);
```

3.6. Power (ac_pow_cordic)

The *ac_pow_cordic* function evaluates the result of raising a fixed point number *x* to the power of another fixed point number *y*.

The following code shows the function declarations for the AC Numerical Datatypes (*ac_fixed*). There are no SystemC variants.

```
void ac_pow_cordic( const ac_fixed<AW,AI,AS,AQ,AO> &a,
                    const ac_fixed<BW,BI,BS,BQ,BO> &b,
                    ac_fixed<ZW,ZI,false,ZQ,ZV> &z);
```

***Note**: The logarithm and exponential cordic functions have limitations on the types they can accept for their input/output. The logarithm functions will only accept unsigned inputs, as the actual logarithm functions operate on the positive real-valued domain, while the exponential functions only accept unsigned types for the output, as they output values in the range of positive real values. This is different than the standard C math functions for logarithm and exponential functions, which accept signed doubles at their input and output.*

Chapter 4: Lookup Table (LUT) Functions

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

- Sine/Cosine (ac_sincos_lut)

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

4.1. Sine/Cosine (ac_sincos_lut)

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

4.1.1. The ac_sincos_lut Implementation

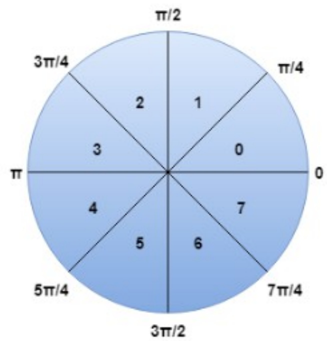
The *ac_sincos_lut* function accepts *ac_fixed* datatypes as input. The domain for the input is (0,1) radians/2PI. The function returns an *ac_complex*<*ac_fixed*> variable where the real part represents the *cosine* value and the imaginary part represents the *sine* value. The number of table entries is 512 values. A prototype of the ac_*sincos_lut* function is shown below:

```
template<class T_in, class T_out>
void ac_sincos_lut(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 an *ac_fixed* datatype. The class *T_out* represents the datatype of the output which should also be an *ac_fixed* datatype. Here, *input* represents the input to the *ac_sincos_lut* function, the *output* is an *ac_complex* variable whose real part is the output *cosine* value and whose imaginary part is the output *sine* value.

Algorithm

All the entries for the sine and cosine lookup tables were generated using the C++ math library sin() and cos() functions and the values are represented as constant doubles. The table contains only those values for the range of angles from (0-PI/4] radians and symmetry is used to adjust the output accordingly. By leveraging the symmetry the table can have more entries for greater precision while still covering the full range of angle inputs. To compute the index into the table, the first 3 bits are extracted from the MSB side of the input to determine in which octant the input lies. The octants can be defined as:

Input Angle (1/2PI radians)	Octant	
[0.000 – 0.125)	0	
[0.125 – 0.250)	1	
[0.250 – 0.375)	2	
[0.375 – 0.500)	3	
[0.500 – 0.625)	4	
[0.625 – 0.750)	5	
[0.750 – 0.875)	6	
[0.875 – 1.000)	7	

The look up table index in this implementation uses the input bits (MSB-3 : LSB). When the input bitwidth is exactly 12 bits, this results in 512 possible indexes. If the input bitwidth is greater than 12 bits, then the closest table entry is found. If the input bitwidth is less than 12 bits, then a stride is implemented.

Handling negative inputs

If the input angle is a negative value (for example -0.125) it would be represented in 2's complement form as 1.1110 0000. Ignoring the integer portion of the result, the fractional portion is 0.875 which is identical to -0.125.

Returning by Value

The implementations of the ac_sincos_lut() function can return the output by value as well as by reference. See the sample code below for examples of each form. The prototype for the function to return by value is shown below:

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

4.1.2. 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;

 // returning value by reference
ac_sincos_lut(x, y);
cout << "Sine: " << y.i() << endl;
cout << "Cosine: " << y.r() << endl;
```

```
// returning by value
y = ac_sincos_lut<output_type>(x);
```

4.1.3. Increasing look up table entries

In the current implementation, the number of lookup table entries for the sine and cosine lookup tables is 512. This means that the implementation gives an accurate output for the cases when input bitwidth is less than or equal to 12 bits. So for cases when input bitwidth is greater than 12 bits, the closest table entry is found (as mentioned above) which might incur some error (maximum error is approximately equal to the largest difference between consecutive lookup table entries) and therefore if a more accurate implementation is desired, then the current lookup table entries can be replaced. The formula for the number of lookup table entries for sine and cosine is as follows:-

Number of lookup table entries = 2 ^ (input bitwidth - 3)

For example, the number of lookup table entries to get an accurate output for the following input bitwidths are:-

- 13 bits – 1024
- 4 bits – 2048
- 15 bits – 4096

The *lutgensincos.cpp* file under the *.../src/examples/Math/ac_sincos_lut* directory can be referred for generating lookup table entries. Also the *ac_sincos_lut.h* library header file has to be accordingly modified if the number of lookup table entries are changed.

Synthesizing the ac_sincos_lut function

In High Level Synthesis, the *ac_sincos_lut* function can be completely pipelined as there are no data dependencies thus ensuring high throughput and low latency.

Chapter 5: Linear Algebra Functions

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

- Cholesky Decomposition (ac_chol_d)
- Cholesky Inverse (ac_cholinv)
- Determinant (ac_determinant)
- Matrix Multiplication (ac_matrixmul)
- QR Decomposition (ac_qrd)

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

5.1. Cholesky Decomposition (ac_chol_d)

The *ac_chol_d* library is designed to provide a Cholesky Decomposition of a square, positive definite input matrix using the Cholesky-Crout algorithm. The user can utilize either accurate math functions or the piece-wise linear (pwl) math library for the internal calculations involved. Cholesky Decomposition is an important linear algebra operation that has applications in solving linear equations.

5.1.1. The ac_chol_d Implementation

The *ac_chol_d* library provides ten overloaded functions for computing the Cholesky Decomposition of real and complex matrices, and returns the lower triangular matrix result of the Cholesky Decomposition. Combined, the ten functions handle five datatypes, which are (a) ac_fixed (b) ac_float (c) ac_std_float (d) ac_ieee_float and (e) ac_complex<ac_fixed>. The matrix of data elements of each type can either be passed as standard two-dimensional C++ arrays, or can be packaged in the ac_matrix class.

Algorithm

As discussed earlier, the Cholesky-Crout algorithm is used to compute the Cholesky Decomposition. The computation for this algorithm is done in a column-wise manner. We first compute the diagonal element for each matrix either using the accurate or the approximate (PWL) *sqrt* function. After we calculate the diagonal element, we store its inverse in a separate variable, which is then reused for the computation of the non-diagonal elements below the diagonal. This inverse square root value can be calculated using the accurate *ac_div* function, or the approximate PWL version from the *ac_inverse_sqrt_pwl* library.

The Cholesky-Crout algorithm is used due to its simplicity and the reusability of the reciprocal value in calculating the non-diagonal elements.

Accurate Math Functions vs. PWL Approximations

The user has the option of being able to choose the accurate versions of the reciprocal and the sqrt functions, or their PWL approximations, as mentioned earlier, depending upon how much accuracy they may desire. Both have their advantages and disadvantages. The accurate math functions, while providing high precision, can also add a lot of overhead in terms of throughput/area. The PWL functions, while providing higher throughput at a lesser cost in area, are also imprecise.

To use PWL approximations, the user has to override a default template parameter (*use_pwl*) for the *ac_chol_d* function call. An example of doing so is giving in the Example Function Calls section.

Floating Point Implementations

As mentioned earlier, the library provides support for *ac_float*, *ac_std_float* and *ac_ieee_float* intermediate variables. The *ac_std_float* and *ac_ieee_float* implementations serve as a wrapper around and depend on the *ac_float* implementation for the actual calculations, with temporary arrays provided in both these wrapper implementations to allow for interfacing with the *ac_float* implementation.

Type of Intermediate Variables

Intermediate variables are used within the function to store the result of repeated subtractions in the process of calculating each element, as well as the reciprocal of diagonal elements. The default precision of these intermediate variables is set to be equal to the precision of the output variables. However, the user can choose to add any number of bits to these default precision values (in case of complex intermediate variables, these extra bits are used for the real/imaginary parts). The precision can be changed by overriding the default *delta_** template parameters. The two possible sets of *delta_** parameters are given as follows:

- For real/complex fixed point variables: *delta_w* and *delta_i* (added to word and integer width of fixed point intermediate variables, respectively).
- For real floating point variables: delta_w, delta_i and delta_e (added to word, integer and exponent width of the *ac_float* intermediate variables, respectively)

The user can also choose the rounding mode of the intermediate variables used for both fixed and floating point intermediate variables by overriding the *imd_Q* default template parameter (set to *AC_RND* by default). Similarly, the *imd_O* template parameter (set to *AC_SAT* by default) can be overridden to change the saturation mode of fixed-point intermediate types.

Note that, because the *ac_std_float* and *ac_ieee_float* implementations are a wrapper around the *ac_float* implementation, any information passed through the *delta_** and *imd_** template parameters for either is in turn passed to the *ac_float* implementation.

Examples on how to override these parameters are given in the Example Function Calls section below.

Input Checking

As explained earlier, the input matrix must be positive definite. While calculating the values for the diagonal elements of the matrix, a square root operation is performed, either using a PWL approximation or the accurate math function. If the input to the *sqrt* function is negative/zero for any value, that means that the input matrix is not positive definite, and a macro-enabled *AC_ASSERT* is provided that will throw a run-time error in such a case. In case the *AC_ASSERT* does not kick in, additional, synthesizable functionality is also provided to output a matrix of zeros.

For certain input matrices when the PWL implementations are used for calculation, the inaccuracy incurred during computations might be large enough to result in the input matrix being wrongly perceived as not positive definite even when it is such. This particularly occurs when the diagonal values of the output matrix are very small and hence, as a result, even a small absolute error in the calculation of the reciprocal value of this diagonal quickly blows up and results in a large error when the remaining elements of that column are calculated. As this error builds up during the calculation of the value for the next diagonal element, the value calculated as the input to the square root function can turn out to be negative, hence resulting in the input checking failing for this particular case even though the input matrix is positive definite.

5.1.2. Function Prototypes

The following are the overloaded function prototypes for the *ac_chol_d* function to handle different datatypes:

```
template<bool use_pwl = false,
         int delta_w = 0, int delta_i = 0,
          ac_q_mode imd_Q = AC_RND, ac_o_mode imd_O = AC_SAT,
          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,
         unsigned M>
void ac_chol_d(
  const ac_fixed<W, I, S, Q, O> A[M][M],
  ac_fixed<outW, outI, outS, outQ, outO> L[M][M]
)
```

```
template<bool use_pwl = false,
         int delta_w = 0, int delta_i = 0, int delta_e = 0,
         ac_q_mode imd_Q = AC_RND,
         int W, int I, int E, ac_q_mode Q,
         int outW, int outI, int outE, ac_q_mode outQ,
         unsigned M>
void ac_chol_d(
  const ac_float<W, I, E, Q> A[M][M],
  ac_float<outW, outI, outE, outQ> L[M][M]
)
```

```
template<bool use_pwl = false,
         int delta_w = 0, int delta_i = 0, int delta_e = 0,
         ac_q_mode imd_Q = AC_RND,
         int W, int E,
```

```
         int outW, int outE,
         unsigned M>
void ac_chol_d(
  const ac_std_float<W, E> A[M][M],
  ac_std_float<outW, outE> L[M][M]
)
```

```
template<bool use_pwl = false,
         int delta_w = 0, int delta_i = 0, int delta_e = 0,
         ac_q_mode imd_Q = AC_RND,
         ac_ieee_float_format Format,
         ac_ieee_float_format outFormat,
         unsigned M>
void ac_chol_d(
  const ac_ieee_float<Format> A[M][M],
  ac_ieee_float<outFormat> L[M][M]
)
```

```
template<bool use_pwl = false,
         int delta_w = 0, int delta_i = 0,
         ac_q_mode imd_Q = AC_RND, ac_o_mode imd_O = AC_SAT,
         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,
         unsigned M>
void ac_chol_d(
  const ac_complex<ac_fixed<W, I, S, Q, O> > A[M][M],
  ac_complex<ac_fixed<outW, outI, outS, outQ, outO> > L[M][M]
)
```

```
template<bool use_pwl = false,
          int delta_w = 0, int delta_i = 0,
          ac_q_mode imd_Q = AC_RND, ac_o_mode imd_O = AC_SAT,
         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,
          unsigned M>
void ac_chol_d(
  const ac_matrix<ac_fixed<W, I, S, Q, O>, M, M> &A,
  ac_matrix<ac_fixed<outW, outI, outS, outQ, outO>, M, M> &L
)
```

```
template<bool use_pwl = false,
         int delta_w = 0, int delta_i = 0, int delta_e = 0,
         ac_q_mode imd_Q = AC_RND,
         int W, int I, int E, ac_q_mode Q,
         int outW, int outI, int outE, ac_q_mode outQ,
         unsigned M>
void ac_chol_d(
  const ac_matrix<ac_float<W, I, E, Q>, M, M> &A,
  ac_matrix<ac_float<outW, outI, outE, outQ>, M, M> &L
)
```

```
template<bool use_pwl = false,
         int delta_w = 0, int delta_i = 0, int delta_e = 0,
         ac_q_mode imd_Q = AC_RND,
         int W, int E,
         int outW, int outE,
         unsigned M>
void ac_chol_d(
  const ac_matrix<ac_std_float<W, E>, M, M> &A,
  ac_matrix<ac_std_float<outW, outE>, M, M> &L
)
```

```
template<bool use_pwl = false,
         int delta_w = 0, int delta_i = 0, int delta_e = 0,
         ac_q_mode imd_Q = AC_RND,
         ac_ieee_float_format Format,
         ac_ieee_float_format outFormat,
         unsigned M>
void ac_chol_d(
  const ac_matrix<ac_ieee_float<Format>, M, M> &A,
  ac_matrix<ac_ieee_float<outFormat>, M, M> &L
)
```

```
template<bool use_pwl = false,
          int delta_w = 0, int delta_i = 0,
          ac_q_mode imd_Q = AC_RND, ac_o_mode imd_O = AC_SAT,
```

```
         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,
          unsigned M>
void ac_chol_d(
  const ac_matrix<ac_complex<ac_fixed<W, I, S, Q, O> >, M, M> &A,
  ac_matrix<ac_complex<ac_fixed<outW, outI, outS, outQ, outO> >, M, M> &L
)
```

5.1.3. C++ Compiler

The functions use default template arguments. This requires using a C++ compiler that supports the C++11 standard. Failing to use such a compiler will result in compilation errors.

5.1.4. Example Function Calls

An example of a function call to store the lower triangular Cholesky Decomposition matrix of a sample 8x8 fixed point, positive definite matrix A in a matrix L is shown below:

```
typedef ac_matrix<ac_fixed<20,11,true,AC_RND,AC_SAT>,8,8> i_type;
typedef ac_matrix<ac_fixed<30,15,true,AC_RND,AC_SAT>,8,8> o_type;
i_type A;
o_type L;
//Hypothetical function that generates a positive definite matrix
//and stores the output in A.
gen_pos_def_matrix(A);
ac_chol_d(A, L);
```

The ac_matrix L hereafter stores the lower triangular matrix result of the Cholesky decomposition.

Choosing PWL approximation

By default, accurate math functions are used for the calculation of internal variables. However, as mentioned earlier, the user can override the default. They can use PWL approximation functions by giving a Boolean template argument, as follows:

```
ac_chol_d<true>(A, L);
```

Configuring the Type for Temporary Variables

The user can add extra bits to the default precision of the temporary variables in the function, by overriding the default template parameters. They can also add other rounding and saturation modes for the temporary variables for fixed point variables. They can do it as follows:

```
ac_chol_d<false, 10, 5, AC_TRN, AC_WRAP>(A, L);
```

By doing this, the user will add 10 bits to the default value for the temporary variable word width, 5 bits to the default value for the temporary variable integer width and switch off rounding and saturation for the temporary variables. Note that the user must also explicitly specify whether they want to use PWL approximation or the accurate math functions in such a case. (In this case, the user specifies that they want to use the accurate math functions)

Similarly, the user can also subtract bits from the default precision values. To do so, they merely need to pass negative parameters for the same. If the user wants to subtract 3 bits from word width and 2 bits from integer width, they can pass the following template parameters:

```
ac_chol_d<false, -3, -2>(A, L);
```

An example function call where the user changes the type configuration for floating point intermediate variables is shown below:

```
ac_chol_d<false, 10, 5, 2, AC_TRN>
```

Doing so will add 10 bits to the bitwidth, 5 bits to the integer width and 2 bits to the exponent width of temporary variables while setting their rounding type to *AC_TRN* and using accurate math functions.

5.2. Cholesky Inverse (ac_cholinv)

The *ac_cholinv* function provides matrix inversion of a positive definite matrix using forward substitution and Cholesky Decomposition which uses the Cholesky-Crout algorithm. The user can utilize either accurate math functions or the PWL math library for the internal calculations involved.

5.2.1. The ac_cholinv Implementation

The *ac_cholinv* library provides two overloaded functions for computing the inverse of input matrices and returns the inverted matrix. Each overloaded function handles a different input datatype. The two datatypes hence handled are *ac_fixed* and *ac_complex<ac_fixed>*. The matrix of data elements of each type are packaged in the *ac_matrix* class.

Algorithm

As mentioned earlier, the Cholesky-Crout algorithm is used to compute the Cholesky Decomposition which is done in the *ac_chol_d.h* library and it returns a lower triangular matrix. Please refer to the documentation of the library *ac_chol_d.h* for Cholesky Decomposition. Then the inverse of the lower triangular matrix is computed using forward substitution and then this inverse is multiplied with its conjugate transpose to obtain the inverse matrix.

Accurate Math Functions vs. PWL Approximations

The user has the option of being able to choose the accurate versions of the reciprocal and the *sqrt* functions, or their PWL approximations, as mentioned earlier, depending upon how much accuracy they may desire. Both have their advantages and disadvantages. The accurate math functions, while accurate, can also add a lot of overhead in terms of throughput/area. The PWL functions, while providing higher throughput, are a bit imprecise.

To use the pwl functions, the user has to override a default template parameter (*use_pwl1*) for the *ac_chol_d* function call in the *ac_cholinv* function and (use_pwl2) for the *ac_cholinv* function call. An example of doing so is giving in the *Example Function Calls* section.

Type of Intermediate Variables

Intermediate variables are used within the function to store the result of repeated additions in the process of calculating each element, as well as the reciprocal of diagonal elements. The default precision of these temporary variables is set to be equal to the precision of the output variables. However, the user can choose to add any number of bits to these default values for word width as well as the integer width of the temporary variables (in case of complex temporary variables, these extra bits are used for the real/imaginary parts). They can do this by overriding the default template parameters (add2w and add2i for word and integer width, respectively). Furthermore, the user can also choose the rounding and saturation modes for the temporary variables by overriding the appropriate template parameters (temp_Q for rounding mode and temp_O for saturation). An example of how to do so is giving in the Example Function Calls section.

5.2.2. Function Prototypes

The following are the overloaded function prototypes for the ac_chol_d function to handle different datatypes:

```
template<bool use_pwl1 = false,
        bool use_pwl2 = false,
        int add2w = 0,
        int add2i = 0,
        ac_q_mode temp_Q = AC_RND,
        ac_o_mode temp_O = AC_SAT,
        unsigned M,
        class T_in,
         class T_out>
 void ac_chol_d(
  const ac_matrix<T_in, M, M> &A,
  ac_matrix<T_out, M, M> &L
)
template<bool use_pwl1 = false, bool use_pwl2 = false, int      add2w = 0, int
add2i = 0, ac_q_mode temp_Q = AC_RND, ac_o_mode temp_O = AC_SAT, unsigned M,
class T_in, class T_out>
 void ac_chol_d(
  const ac_matrix<ac_complex<T_in>, M, M> &A,
  ac_matrix<ac_complex<T_out>, M, M> &L
)
```

5.2.3. 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 or a later standard as the standard for their compilation, failing which a compile-time error is thrown.

5.2.4. Example Function Calls

An example of a function call to store the lower triangular Cholesky Decomposition matrix of a sample 8x8 positive definite matrix A in a matrix L is shown below:

```
typedef ac_matrix<ac_fixed<20,11,true,AC_RND,AC_SAT>,8,8> i_type;
typedef ac_matrix<ac_fixed<30,15,true,AC_RND,AC_SAT>,8,8> o_type;
i_type A;
o_type Ainv;
//Hypothetical function that generates a positive definite matrix
//and stores the output in A.
gen_pos_def_matrix(A);
ac_cholinv(A, Ainv);
```

Choosing PWL approximation

By default, accurate math functions are used for the calculation of internal variables. However, as mentioned earlier, the user can override the default. The first Boolean template parameter if true uses PWL approximation functions for calculation of internal variables and if false uses accurate math functions in the ac_chol_d library and similarly if the second Boolean template parameter if true uses PWL approximation functions for calculation of internal variables and if false uses accurate math functions in the ac_cholinv library.

1. If the user wants to use the PWL approximation functions for both the ac_chol_d and ac_chol_inv libraries, then the following is the way the function should be called

    ```
       ac_chol_inv<true, true>(A, L);
    ```

2. If the user wants to use the PWL approximation functions for the ac_chol_d library and accurate math functions for the ac_chol_inv library, then the following is the way the function should be called.

    ```
       ac_chol_inv<true, false>(A, L);
    ```

    Note: If the user wants to specify the template parameter for choosing the PWL approximation functions only for the ac_cholinv library, then the user has to explicitly specify whether they want to use PWL approximation or the accurate math functions for the ac_chol_d library as well.

Configuring the Type For Temporary Variables

The user can add extra bits to the default precision of the temporary variables in the function, by overriding the default template parameters. They can also add their own quantization and overflow modes for the temporary variables. For example, they can do it as follows:

```
ac_chol_inv<false, false, 10, 5, AC_RND, AC_SAT>(A, Ainv);
```

By doing this, the user will add 10 bits to the default value for the temporary variable word width, 5 bits to the default value for the temporary variable integer width and switch on rounding and saturation for the temporary variables. Note that the user must also explicitly specify whether they want to use PWL approximation or the accurate math functions in such a case. (In this case, the user specifies that they want to use the accurate math functions)

Similarly, the user can also subtract bits from the default precision values. To do so, they merely need to pass negative parameters for the same. If the user wants to subtract 3 bits from word width and 2 bits from integer width, they can pass the following template parameters:

```
ac_chol_d<false, false, -3, -2>(A, Ainv);
```

5.3. Determinant (ac_determinant)

The *ac_determinant* library implementation is a fully parallelized and scalable implementation for determinant computation using template recursion functionality, with the user being able to choose between using an internally determined datatype or adding their own parameters for intermediate type precision, signedness, rounding and saturation.

5.3.1. The ac_determinant Implementation

The *ac_determinant* library provides four overloaded functions for computing the determinant of real and complex matrices. Combined, the four functions handle two datatypes, which are (a) ac_fixed and (b) ac_complex<ac_fixed>. The matrix of data elements of each type can either be passed as standard two-dimensional C++ arrays, or can be packaged in the ac_matrix class.

Algorithm

Determinant computation is a recursive process. This implementation leverages the recursive tendency of this function to design a hardware-efficient implementation that is scalable and is highly parallelizable. Hence, higher order matrices are reduced to 2x2 matrices by computing minors recursively on them, with each recursion reducing the row and column size by 1. A specialization is also defined for a 1x1 matrix, in which case, we just return the sole value stored in the matrix.

In order to implement a fully parallelizable and synthesizable architecture, template recursion is used, with the 2x2 matrix being the specialized case for recursion.

Internal Precision Adjustment

The ac_determinant implementation is such that it maintains full internal precision. To make sure that it does that, bitwidths are computed at every step of the coding.

Using full internal precision can result in a very large bitwidth, however. In case full internal precision might not be required, the user can set their own internal precision by overriding the default template parameters and providing their own internal types.

5.3.2. Function headers

The following are the overloaded function prototypes for the ac_determinant function to handle different datatypes:

```
template <bool override = false,
          int internal_width = 16, int internal_int = 8,
          bool internal_sign = true, ac_q_mode internal_rnd = AC_RND,
          ac_o_mode internal_sat = AC_SAT,
          unsigned M,
          int W1, int I1, bool S1, ac_q_mode q1, ac_o_mode o1,
          int W2, int I2, ac_q_mode q2, ac_o_mode o2>
```

```
void ac_determinant(
  ac_matrix <ac_fixed <W1, I1, S1, q1, o1>, M, M> &input,
  ac_fixed<W2, I2, true, q2, o2> &result
)
```

```
template <bool override = false,
          int internal_width = 16, int internal_int = 8,
          bool internal_sign = true, ac_q_mode internal_rnd = AC_RND,
          ac_o_mode internal_sat = AC_SAT,
          unsigned M,
          int W1, int I1, bool S1, ac_q_mode q1, ac_o_mode o1,
          int W2, int I2, ac_q_mode q2, ac_o_mode o2>
void ac_determinant(
  const ac_matrix<ac_complex<ac_fixed<W1,I1,S1,q1,o1> >, M, M> &input,
  ac_complex<ac_fixed<W2, I2, true, q2, o2> > &result
)
```

```
template <bool override = false,
          int internal_width = 16, int internal_int = 8,
          bool internal_sign = true, ac_q_mode internal_rnd = AC_RND,
          ac_o_mode internal_sat = AC_SAT,
          unsigned M,
          int W1, int I1, bool S1, ac_q_mode q1, ac_o_mode o1,
          int W2, int I2, ac_q_mode q2, ac_o_mode o2>
void ac_determinant(
  const ac_fixed<W1, I1, S1, q1, o1> a[M][M],
  ac_fixed<W2, I2, true, q2, o2> &result
)
```

```
template <bool override = false,
          int internal_width = 16, int internal_int = 8,
          bool internal_sign = true, ac_q_mode internal_rnd = AC_RND,
          ac_o_mode internal_sat = AC_SAT,
          unsigned M,
          int W1, int I1, bool S1, ac_q_mode q1, ac_o_mode o1,
          int W2, int I2, ac_q_mode q2, ac_o_mode o2>
void ac_determinant(
  const ac_complex<ac_fixed<W1, I1, S1, q1, o1> > a[M][M],
  ac_complex<ac_fixed<W2, I2, true, q2, o2> > &result
)
```

The library also provides the following two functions to allow the user to return by value.

```
template<class T_out,
         bool override = false,
         int internal_width = 16, int internal_int = 8,
         bool internal_sign = true, ac_q_mode internal_rnd = AC_RND,
         ac_o_mode internal_sat = AC_SAT,
         unsigned M, class T_in>
T_out ac_determinant(const ac_matrix <T_in, M, M> &input)
```

```
template<class T_out,
         bool override = false,
         int internal_width = 16, int internal_int = 8,
         bool internal_sign = true, ac_q_mode internal_rnd = AC_RND,
         ac_o_mode internal_sat = AC_SAT,
         unsigned M, class T_in>
T_out ac_determinant(const T_in input[M][M])
```

5.3.3. C++ Compiler

The functions use default template arguments. This requires using a C++ compiler that supports the C++11 standard, or a later standard. Failing to use such a compiler will result in compilation errors.

5.3.4. Example Function Call

The following give examples of using the determinant function with both ac_matrix and C style array inputs.

```
// Using the ac_matrix wrapper class
ac_matrix <ac_fixed <8, 5, true, AC_RND, AC_SAT>, 2, 2> input;
typedef ac_fixed <27, 18, true, AC_RND, AC_SAT> output_type;
output_type output;
// Assign elements to input
input(0, 0) = 1;
input(0, 1) = 1.5;
input(1, 0) = 0.5;
input(1, 1) = 1.75;
ac_determinant (input, output);
// The above function call can also be replaced by:
output = ac_determinant <output_type> (input);
```

```
// Using C-style arrays
ac_fixed <8, 5, true, AC_RND, AC_SAT> input[2][2];
typedef ac_fixed <27, 18, true, AC_RND, AC_SAT> output_type;
```

```
output_type output;
// Assign elements to input
input[0][0] = 1;
input[0][1] = 1.5;
input[1][0] = 0.5;
input[1][1] = 1.75;
ac_determinant (input, output);
// The above function call can also be replaced by:
output = ac_determinant <output_type> (input);
```

5.4. Matrix Multiplication (ac_matrixmul)

The ac_matrixmul function is designed to provide the multiplication of two matrices with full precision by default and if user wants to specify his own precision then a provision is provided for this purpose.

5.4.1. The ac_matrixmul Implementation

The ac_matrixmul library provides implementation for ac_fixed and ac_complex<ac_fixed> datatypes. There is one generalized function which handles the computation for both the datatypes. A prototype of the ac_matrixmul function is shown below:

```
template<int mw = 0, int mi = 0, ac_q_mode mq = AC_RND,
         ac_o_mode mo= AC_SAT,
         int sw = 0, int si = 0, ac_q_mode sq = AC_RND,
         ac_o_mode so = AC_SAT,
         unsigned M, unsigned N, unsigned P, class T_in_A,
         class T_in_B, class T_op>
void
ac_matrixmul(
    const ac_matrix<T_in_A, M, N> &A,
    const ac_matrix<T_in_B, N, P> &A,
    ac_matrix<T_op, M, P> &C
)
```

Definition of matrices:

In the above prototype, ac_matrix is a container class that helps to define two dimensional matrices. The following is the way to define a matrix using the ac_matrix class for the ac_fixed datatype:

```
ac_matrix<ac_fixed<20, 10, true, AC_RND, AC_SAT>, M, N> A;
```

In the definition above, A is a matrix of dimension M x N where its elements are of ac_fixed datatype.

Similarly, the following is the way to define a matrix using the ac_matrix class for the ac_complex<ac_fixed> datatype:

```
ac_matrix<ac_complex<ac_fixed<40, 20, true, AC_RND, AC_SAT> >, M, N> A;
```

In the definition above, A is a matrix of dimension M x N where its elements are of ac_fixed datatype.

Default Template Arguments:

By default, the internal variables have full precision turned on. There are two internal variables mult and sum used in the computation of matrix multiplication. The code snippet where the sum and mult variables are used in the matrix multiplication block is as follows:

```
for (unsigned i=0; i<M; i++) {
  for (unsigned j=0; j<P; j++) {
     mult = 0;
     sum = 0;
     for (unsigned k=0; k<N; k++) {
           mult = A(i,k) * B(k,j);
           sum = sum + mult;
     }
     C(i,j) = sum;
   }
 }
```

The default template arguments in the order which they have been mentioned in the list of template parameters in the function prototype for this block are:

- Width of the mult variable – mw
- Integer width of the mult variable – mi
- Quantization mode of the mult variable – mq
- Overflow mode of the mult variable – mo
- Width of the sum variable – sw
- Integer width of the sum variable – si
- Quantization mode of the sum variable – sq
- Overflow mode of the sum variable - so

For the ac_fixed datatype

From the above code snippet, in order to achieve full precision for the mult variable its width and integer width is calculated as the sum of the widths and the sum of the integer widths of the elements of A and B matrices respectively. For the sum variable, the full precision is dependent on the number of times it is accumulated by the mult variable which is in turn decided by the number of iterations in the for loop in which it is accumulated. As the number of iterations are N in this for loop, the width and the integer width for the sum variable is calculated as the width of the mult variable + log2_ceil(N) and the integer width of the mult variable + log2_ceil(N) respectively. The default quantization and overflow modes for the mult and the sum variables are defined as AC_RND and AC_SAT respectively.

For the ac_complex<ac_fixed> datatype

Everything above applies for the ac_complex<ac_fixed> datatype except that the width and integer of the mult variable is calculated as the sum of the widths + 1 and the sum of the integer widths + 1 of the elements of A and B matrices respectively. This is because when two complex numbers are multiplied for example:

```
(a1+b1i) x (a2+b2i) = a1a2 – b1b2 + (a1b2 + a2b1)i
```

The computation of the real part and the imaginary part involves an extra addition or subtraction and thus 1 is added. The user must keep in mind that in order to allow the usage of default template arguments for functions, they must use a version of C++ that allows this usage, such as C++ 11, in case they wish to compile and execute the code for the matrixmul library, failing which an error will be thrown by the compiler.

If the user wants to specify his/her own precision for the internal variables instead of the default, then please refer to the “Example Function Calls” section below for the same.

Wrong Datatypes

If any of the two inputs and the output have different datatypes from each other, then a static assert is thrown stating that ‘*Both arguments need to be of the same datatype*’ in case of C++11 version of the compiler and in versions prior to that a compiler error is generated.

5.4.2. Example Function Call

An example of a function call to store the result matrix of the two input matrices A and B who have elements of ac_fixed datatype is shown below:

```
typedef ac_matrix<ac_fixed<20, 10, true, AC_RND, AC_SAT>, M, N> input_type1;
typedef ac_matrix<ac_fixed<20, 10, true, AC_RND, AC_SAT>, N, P> input_type2;
typedef ac_matrix<ac_fixed<61, 31, true, AC_RND, AC_SAT>, M, P> output_type;
input_type1 A;
input_type2 B;
output_type C;
ac_matrixmul(A, B, C);    //Function call
```

The two input matrices here are A and B and the output matrix here is C.

Changing default template parameters through function call:

- If the user wants a different precision from the default precision, then the default

  template parameters have to be modified. If the user wants to modify all the template parameters then the user has to modify the above function call in the following way:

```
ac_matrixmul<mw, mi, mq, mo, sw, si, sq, so> (A, B, C);
```

The above order in which the template arguments are specified has to be maintained.

- Suppose the user only wants to modify the width of the mult variable i.e mw and keep all the other template arguments as it is, then the function call can be written as:

```
ac_matrixmul<mw> (A, B, C);
```

- But if the user wants to modify only sw for instance, then in order to do this the user has to make a function call like this:

```
ac_matrixmul<mw, mi, mq, mo, sw> (A, B, C);
```

5.4.3. Debug

Here is a provision to enable debug mode in the *ac_matrixmul.h* file. It can be done by defining the macro as

```
#define MATRIXMUL_DEBUG
```

5.5. QR Decomposition (ac_qrd)

The *ac_qrd* function provides QR decomposition of square input matrix A. In linear algebra, QR decomposition is a decomposition of matrix A into product A = QR where Q is an orthogonal matrix and R is an upper triangular matrix.

This function uses systolic array based implementation where each iteration performs a Given's rotation algorithm to convert given matrix into Q and R matrices. Note that this function uses ac_matrix container class and input and output matrices are either supplied in the form of ac_matrix objects or via standard AC datatype 2D arrays. This QR decomposition implementation supports ac_fixed and ac_complex<ac_fixed> datatypes. If the function is called with other datatypes, compilation error is thrown.

5.5.1. The ac_qrd Implementation

The ac_qrd implementation used Givens rotation algorithm to convert any given matrix into an upper triangular matrix which is R. Computation of Q requires same set of sequence of rotations and can be computed simultaneously. Givens rotation has the property of implementing the QRD decomposition in systolic manner, which makes it possible to synthesize parallel architecture and simplify the design.

The process of QR decomposition can be divided into two different functions (processing elements) that are called as off-diagonal and diagonal processing elements (PEs). Diagonal PEs are used to compute rotational parameters, whereas off-diagonal PEs are used to apply them to the rows and compute the new values of the rows in the final R and Q matrices.

Systolic Array Structure of QRD

QRD can be massively parallelized if only it performs rotation on two different set of rows.

The process of applying rotations and computing rotational parameters c and s can be divided into two different functions (processing elements) that are called as off-diagonal and diagonal processing elements (PEs). Diagonal PEs are used to compute c and s, where as off-diagonal PEs are used to apply them to the rows and compute the new values of the rows in the final R matrix.

- **Diagonal Processing Elements** are expressed in the form of function, diagonal_PE which takes element to be zeroed (b) and element in the row just above that in the same column (a) as inputs and returns the rotational matrix parameters. It also takes the default Boolean parameter ispwl, which allows user to do architectural exploration by switching between PWL functions and accurate ac_math functions. By default, PWL functions are used to make sure that the error is minimum, although this increases the area of the Diagonal Processing elements.

- **Off-Diagonal Processing Elements are expressed using function offdiagonal_PE, which applies Givens rotation to the relevant rows of the Q and R matrices.**
  Givens rotation for complex matrices produces a complex output for the bottom right portion of the R matrix by default. However, certain applications might require the R matrix to have real elements throughout to enable operations such as matrix inversion during downstream processing. The user can obtain such outputs by toggling the real_diag template parameter to "true". The calculation of the parameters required to do this final rotation can be done through PWL functions or accurate ac_math functions. This choice is determined via the "ispwl" template parameter.

5.5.2. Function prototypes

For ac_matrix of ac_fixed:

```
template<
  bool ispwl = true, unsigned M,
  int W1, int I1, ac_q_mode q1, ac_o_mode o1,
  int W2, int I2, ac_q_mode q2, ac_o_mode o2
>
void ac_qrd(
  ac_matrix<ac_fixed <W1, I1, true, q1, o1>, M, M> &A,
  ac_matrix<ac_fixed <W2, I2, true, q2, o2>, M, M> &Q,
  ac_matrix<ac_fixed <W2, I2, true, q2, o2>, M, M> &R
)
```

For ac_matrix of ac_complex<ac_fixed>:

```
template<
  bool real_diag = false, bool ispwl = true, unsigned M,
  int W1, int I1, ac_q_mode q1, ac_o_mode o1,
  int W2, int I2, ac_q_mode q2, ac_o_mode o2
>
void ac_qrd(
  ac_matrix<ac_complex<ac_fixed<W1, I1, true, q1, o1> >, M, M> &A,
  ac_matrix<ac_complex<ac_fixed<W2, I2, true, q2, o2> >, M, M> &Q,
  ac_matrix<ac_complex<ac_fixed<W2, I2, true, q2, o2> >, M, M> &R
```

```
)
```

For C++ ac_fixed array:

```
template<
  bool ispwl = true, unsigned M,
  int W1, int I1, ac_q_mode q1, ac_o_mode o1,
  int W2, int I2, ac_q_mode q2, ac_o_mode o2
>
void ac_qrd(
  ac_fixed <W1, I1, true, q1, o1> A[M][M],
  ac_fixed <W2, I2, true, q2, o2> Q[M][M],
  ac_fixed <W2, I2, true, q2, o2> R[M][M]
)
```

For C++ ac_complex<ac_fixed> array:

```
template<
  bool real_diag = false, bool ispwl = true, unsigned M,
  int W1, int I1, ac_q_mode q1, ac_o_mode o1,
  int W2, int I2, ac_q_mode q2, ac_o_mode o2
>
void ac_qrd(
  ac_complex<ac_fixed <W1, I1, true, q1, o1> > A[M][M],
  ac_complex<ac_fixed <W2, I2, true, q2, o2> > Q[M][M],
  ac_complex<ac_fixed <W2, I2, true, q2, o2> > R[M][M]
```

)

5.5.3. Example Function Calls

An example function call to calculate the QR decomposition of a 3x3 complex ac_matrix object is given below:

```
ac_matrix<ac_complex<ac_fixed<16, 8,false,AC_TRN,AC_WRAP> >, 3, 3> A;
ac_matrix<ac_complex<ac_fixed<32,16,false,AC_TRN,AC_WRAP> >, 3, 3> Q, R;
// Hypothetical function that initializes matrix “A”
init_matrix(A);
ac_qrd(A, Q, R);
```

An example of the same function, but for a 3x3 complex 2D C++ array is given below:

```
ac_complex<ac_fixed<16, 8,false,AC_TRN,AC_WRAP> A[3][3];
ac_complex<ac_fixed<32,16,false,AC_TRN,AC_WRAP> Q[3][3], R[3][3];
// Hypothetical function that initializes matrix A
init_matrix(A);
```

```
ac_qrd(A, Q, R);
```

By default, the design does not perform the final rotation to obtain a real bottom-right element. To make the design do that, set the “real_diag” template parameter to “true”, as shown below:

```
ac_qrd<true>(A, Q, R);
```

The design also uses PWL functions by default. To enable the use of accurate ac_math functions, set the “ispwl” template parameter to “false”, as shown below:

```
ac_qrd<true, false>(A, Q, R);
```

Chapter 6: Miscellaneous Functions

The *ac_math* package also provides the following functions, in addition to the PWL, LUT and CORDIC functions:

- Absolute Value (ac_abs)
- Division (ac_div)
- Square Root (ac_sqrt)
- Shifts (ac_shift_left/ac_shift_right)
- Barrel Shift (ac_barrel_shift)

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

6.1. Absolute Value (ac_abs)

The absolute value (ac_abs) operation produces a positive result by negating values less than zero.

- Integer Signed

```
void ac_abs(ac_int<XW,true> x, ac_int<YW,false> &y)
void ac_abs(ac_int<XW,true> x, ac_int<YW,true> &y)
```

- Fixed Point Signed

```
void ac_abs(ac_fixed<XW,XI,true,XQ,XO> x, ac_fixed<YW,YI,false,YQ,YO> &y)
void ac_abs(ac_fixed<XW,XI,true,XQ,XO> x, ac_fixed<YW,YI,true,YQ,YO> &y)
```

- Float

```
void ac_abs(ac_float<XW,XI,XE,XQ> x, ac_float<YW,YI,YE,YQ> &y)
```

6.2. Division (ac_div)

The division functions are supported for integer, fixed-point, complex, and float data types. The division functions takes two inputs: dividend and divisor and computes the quotient. If the inputs are integer there is a version of the function that also computes the remainder of the division. In all cases the div function returns true if the computed remainder is nonzero.

NOTE*: Division by zero triggers an assertion failure during simulation.*

6.2.1. Integer Division

There are several integer division functions defined. Some of the functions compute just the quotient, some compute both the quotient and the remainder. All return a bool flag to indicate whether the remainder is non zero.

The first argument is the dividend (or numerator), the second is the divider (or denominator), the third is the quotient (output argument) and the fourth (optional) argument is the remainder (output argument). All the arguments are either all signed or all unsigned. The computed quotient is equivalent to the resulted computed by the operator `'/'`:

```
ac_int<8,false> n;
ac_int<5,false> d;
ac_int<6,false> q;
ac_div(n, d, q);
ac_int<6,false> q1 = n/d;    // q == q1
```

While the behavior of the last two lines is identical, Catapult will produce different hardware for each case. Catapult will allocate a component from the library for the '/' whereas calling the div function will inline the function.

`bool ac_div(`
`  ac_int<NW,false> dividend,`
`  ac_int<DW,false> divisor,`
`  ac_int<QW,false> &quotient,`
`  ac_int<RW,false> &remainder`
`);`
	`bool ac_div(`
	`  ac_int<NW,true> dividend,`
	`  ac_int<DW,true> divisor,`
	`  ac_int<QW,true> &quotient,`
	`  ac_int<RW,true> &remainder`
	`);`

`bool ac_div(`
`  ac_int<NW,false> dividend,`
`  ac_int<DW,false> divisor,`
`  ac_int<QW,false> &quotient`
`);`
	`bool ac_div(`
	`  ac_int<NW,true> dividend,`
	`  ac_int<DW,true> divisor,`
	`  ac_int<QW,true> &quotient`
	`);`

Table 3: Functions for Integer Division that Return Remainder != 0

For the div functions, dividend == divisor*quotient + remainder provided that both the quotient and the remainder have sufficient precision so that they don’t overflow.

6.2.2. Fixed-point Division

There are several fixed-point division functions defined. Each function returns a *bool* flag to indicate whether the remainder is non zero (assumes that the quotient has enough precision that it does not overflow).

The first argument is the *dividend* (or numerator), the second is the *divider* (or denominator), the third is the *quotient* (output argument). All the arguments are either all signed or all unsigned. The computed quotient takes into account the target bitwidth, integer bitwidth, quantization and overflow modes. All quantization modes work correctly since they are based on the computation of the remainder. The operator ‘/’, on the other hand, is forced to truncate the result without knowing the data type of the target. For example, the operator ‘/’ for *ac_fixed* returns a result that is dependent on the type of both dividend and divisor and the bits (possibly infinite bits) to the right of the result are truncated before the quantization mode of the target is known.

The hardware produced by Catapult for a call to the `‘/’` and `'/='` operators and for a call to the div function are different. In the first case, Catapult allocates a component from the library, whereas in the second case the div function is inlined.

```
bool ac_div(
  ac_fixed<NW,NI,false,NQ,NO> divi-
dend,
  ac_fixed<DW,DI,false,DQ,DO> divisor,
  ac_fixed<QW,QI,false,QQ,QO> &quo-
tient
);
```

```
bool ac_div(
  ac_fixed<NW,NI,true,NQ,NO> dividend,
  ac_fixed<DW,DI,true,DQ,DO> divisor,
  ac_fixed<QW,QI,true,QQ,QO> &quotient
);
```

Table 4: Functions for Fixed-Point Division that Return Remainder != 0

The *div* functions return the *bool* value remainder != 0 which is equivalent to (divisor*quotient != dividend, provided that the quotient does not overflow) and may be dependent on the width and integer width of the quotient. For example:

```
ac_fixed<3,3,false> q; ac_fixed<4,3> q2;
ac_fixed<2,2,false> a = 3; ac_fixed<2,2,false> b = 2;
bool nonzero_rem = ac_div(a, b, q);   // nonzero_rem == true,  q == 1
bool nonzero_rem2 = ac_div(a, b, q2); // nonzero_rem2 == false, q2 == 1.5
```

6.2.3. Float Division

Inlined *ac_float* division is implemented using *ac_fixed* division and has an equivalent return value. The quotient may lose precision as neither the dividend or divisor are normalized. The resulting quotient will overflow if the result of division cannot be represented in the quotient type, otherwise the result is representable and normalized.

```
bool ac_div(
  ac_float<NW,NI,NE,NQ> dividend,
  ac_float<DW,DI,DE,DQ> divisor,
  ac_float<QW,QI,QE,QQ> &quotient
);
```

6.2.4. Complex Division

The *ac_div* library also provides for the division of complex inputs.

```
void ac_div(
  ac_complex<NT> dividend,
```

```
  ac_complex<DT> divisor,
  ac_complex<QT> &quotient
)
```

6.3. Square Root (ac_sqrt)

The square root function has only input argument and one output argument both of which are unsigned. There is an integral version and a fixed-point version.

6.3.1. Integer square root

The integer square root computes the largest integer value such that when squared it is equal or less than the argument. The prototype of the function is given as follows:

```
void ac_sqrt(
  ac_int<XW,false> x,
  ac_int<OW,false> &sqrt
)
```

6.3.2. Fixed-point square root

The fixed-point *ac_sqrt* function allows for full flexibility on both the input and the output precision. The bitwidth, integer bitwidth, quantization and overflow modes of the target (second argument) determine the bits of precision for computing the square root and the quantization and overflow that is performed on the result. All quantization modes are computed exactly based on the remainder of the square root computation. The prototype of the function is given as follows:

```
void ac_sqrt(
  ac_fixed<XW,XI,false,XQ,XO> x,
  ac_fixed<OW,OI,false,OQ,OO> &sqrt
)
```

An example of how the square root functions is used, consider the case where the input value is *ac_fixed<4,4, false>* (4 bit unsigned integer), and the required precision for the result is *ac_fixed<8,3, false>* (3 bit integral, 5 fractional: xxx.xxxxx), the call code would look like:

```
ac_fixed<4,4,false> x = 13;
ac_fixed<8,3,false> sqrt_x;
ac_sqrt(x, sqrt_x); // sqrt_x == (ac_fixed<8,3,false>) sqrt( x.to_double() );
```

6.3.3. Floating-point square root

The *ac_sqrt* function also allows for *ac_float*, *ac_std_float* and *ac_ieee_float* inputs and outputs. The ac_std_float and ac_ieee_float implementations are a wrapper around the ac_float implementation, with temporary variables present to ensure compatibility with the ac_float implementation. The ac_float implementation calculates the square root of the input mantissa, stores it in a temporary variable and then factors in the input exponent value to calculate the final output, in accordance to the equations below:

```
sqrt(floating_point_input) = sqrt(mantissa*2^exp)
                           = sqrt(mantissa)*2^(exp/2)
                           = sqrt(mantissa)*2^(exp>>1)
```

For odd exponent values, we can still use 2^(exp>>1) to factor in 2^(exp/2), provided that we left-shift the mantissa by 1 first. For instance, consider an input with exp = 9.

```
sqrt(floating_point_input) = sqrt(mantissa*2^9)
                           = sqrt(mantissa*2*2^8)
                           = sqrt((mantissa<<1)*2^8)
                           = sqrt(mantissa<<1) * 2^(8<<1)
                           = sqrt(mantissa<<1) * 2^(9<<1)
                             (Since 8<<1 = 9<<1 = 4)
```

The *ac_fixed* temporary variable used to store the value of sqrt(mantissa) (or, in the case of odd exponents, sqrt(mantissa<<1)) bases its fractional bits and rounding mode on default template parameters, as seen in the prototype:

```
template<bool OR_TF = false, int TF_ = 32, ac_q_mode TQ = AC_TRN,
         int XW, int XI, int XE, ac_q_mode XQ, int OW, int OI, int OE,
ac_q_mode OQ>
void ac_sqrt(
  ac_float<XW,XI,XE,XQ> x,
  ac_float<OW,OI,OE,OQ> &sqrt
)
```

If the OR_TF variable is set to false (default value), the temporary variables use the same fractional bitwidth as the output mantissa, if not, they use the fractional bitwidth supplied by the TF_ parameter (32 by default). The rounding mode for temp variables is set to that supplied by the TQ parameter (AC_TRN by default). An example ac_float function call that uses a temporary variable with 16 fractional bits and AC_RND as the rounding mode is given below:

```
ac_float<25, 2, 8> input, output;
ac_sqrt<true, 16, AC_RND>(input, output);
```

The ac_std_float and ac_ieee_float implementations use the same default template parameters, which they in turn supply to the ac_float implementation. The prototypes are given below:

```
template<bool OR_TF = false, int TF_ = 32, ac_q_mode TQ = AC_TRN,
         int XW, int XE, int OW, int OE>
void ac_sqrt(
  ac_std_float<XW, XE> x,
  ac_std_float<OW, OE> &sqrt
)

template<bool OR_TF = false, int TF_ = 32, ac_q_mode TQ = AC_TRN,
         ac_ieee_float_format Format, ac_ieee_float_format outFormat>
```

```
void ac_sqrt(
  ac_ieee_float<Format> x,
  ac_ieee_float<outFormat> &sqrt
)
```

6.3.4. Special input handling

The ac_std_float and ac_ieee_float implementations can also handle negative and NaN inputs:

Input	Output
-0.0	-0.0
Negative non-zero	-nan
+nan	+nan
-nan	-nan

This handling mirrors that provided by the sqrt() function in the math.h standard C++ library. While negative zero handling is provided by default, the AC_SQRT_NAN_SUPPORTED macro must be defined to enable NaN output generation (for negative non-zero and +/-nan inputs).

6.4. Shifts (ac_shift_left/ac_shift_right)

The *ac_shift* functions allow the specification of precision and quantization and overflow modes of the target. This gives a simple way to get around the issue of the fixed-point shift operations >> and << for *ac_fixed* returning the type of the first operand. The shifting provided by these functions is “arithmetic” in nature, that is, these functions provide the same result as multiplying the input by 2^(shift_count), all the while keeping in mind the quantization and overflow modes of the target. In this sense, they allow for saturation and rounding, something that is not provided by the >> and << operators.

6.4.1. Bidirectional shifts

For ac_fixed data-types, both the right shift >> and the left shift << operators shift in the opposite direction when the shift value is negative. The equivalent functionality is provided by the following functions:

<table>
<tr><td><pre>void ac_shift_right(
  ac_fixed<XW,XI,false,XQ,XO> x,
  int n,
  ac_fixed<OW,OI,false,OQ,OO> &sr
)</pre></td><td><pre>void ac_shift_right(
  ac_fixed<XW,XI,true,XQ,XO> x,
  int n,
  ac_fixed<OW,OI,true,OQ,OO> &sr
)</pre></td></tr>
<tr><td><pre>void ac_shift_left(
  ac_fixed<XW,XI,false,XQ,XO> x,
  int n,
  ac_fixed<OW,OI,false,OQ,OO> &sl
)</pre></td><td><pre>void ac_shift_left(
  ac_fixed<XW,XI,true,XQ,XO> x,
  int n,
  ac_fixed<OW,OI,true,OQ,OO> &sl
)</pre></td></tr>
</table>

Table 5: Functions for Fixed-Point Bidirectional Shifts

6.4.2. Unidirectional shifts

If the shift value is know to be non-negative, it is best to cast it to (unsigned int) so that the following functions are inlined. These functions will deliver better quality of results during Catapult synthesis.

`void ac_shift_right(`
`  ac_fixed<XW,XI,false,XQ,XO> x,`
`  unsigned int n,`
`  ac_fixed<OW,OI,false,OQ,OO> &sr`
`)`
	`void ac_shift_right(`
	`  ac_fixed<XW,XI,true,XQ,XO> x,`
	`  unsigned int n,`
	`  ac_fixed<OW,OI,true,OQ,OO> &sr`
	`)`

`void ac_shift_left(`
`  ac_fixed<XW,XI,false,XQ,XO> x,`
`  unsigned int n,`
`  ac_fixed<OW,OI,false,OQ,OO> &sl`
`)`
	`void ac_shift_left(`
	`  ac_fixed<XW,XI,true,XQ,XO> x,`
	`  unsigned int n,`
	`  ac_fixed<OW,OI,true,OQ,OO> &sl`
	`)`

Table 6: Functions for Fixed-Point Unidirectional Shift

6.4.3. Complex shifts

Wrapper functions for the unidirectional *ac_shift_left* and *ac_shift_right* are provided for complex types to perform the corresponding shift for the real and imaginary parts of the underlying type.

`void ac_shift_right(`
`  ac_complex<XT> x,`
`  unsigned int n,`
`  ac_complex<OT> &sr`
`)`
	`void ac_shift_left(`
	`  ac_complex<XT> x,`
	`  unsigned int n,`
	`  ac_complex<OT> &sl`
	`)`

Table 7: Functions for Complex Unidirectional Shift

6.5. Barrel Shift (ac_barrel_shift)

Barrel shifter is a digital circuit that can shift a data word by a specified number of bits without the use of any sequential logic, only pure combinational logic, i.e. it inherently provides a binary operation. The way barrel shifter implemented here is as a sequence of multiplexers where the output of one multiplexer is connected to the input of the next multiplexer in a way that depends on the shift distance. A barrel shifter is often used to shift and rotate n-bits in modern microprocessors, typically within a single clock cycle.

6.5.1. Generic Block Diagram

Block diagram below show the mux based implementation of 4 bit barrel shifter for example, take a four-bit number inputs (MSB)ABCD(LSB) table below show rotation in each stage of 4 bit barrel shift. Shifting in various stages based on control bits are illustrated in table below.

SHIFT	Shift = 0	Shift = 1	Shift = 2	Shift = 3

INPUT	C1=0	C0=0	C1=0	C0=1	C1=1	C0=0	C1=0	C0=0
A	A	A	A	D	C	C	C	B
B	B	B	B	A	D	D	D	C
C	C	C	C	B	A	A	A	D
D	D	D	D	C	B	B	B	A

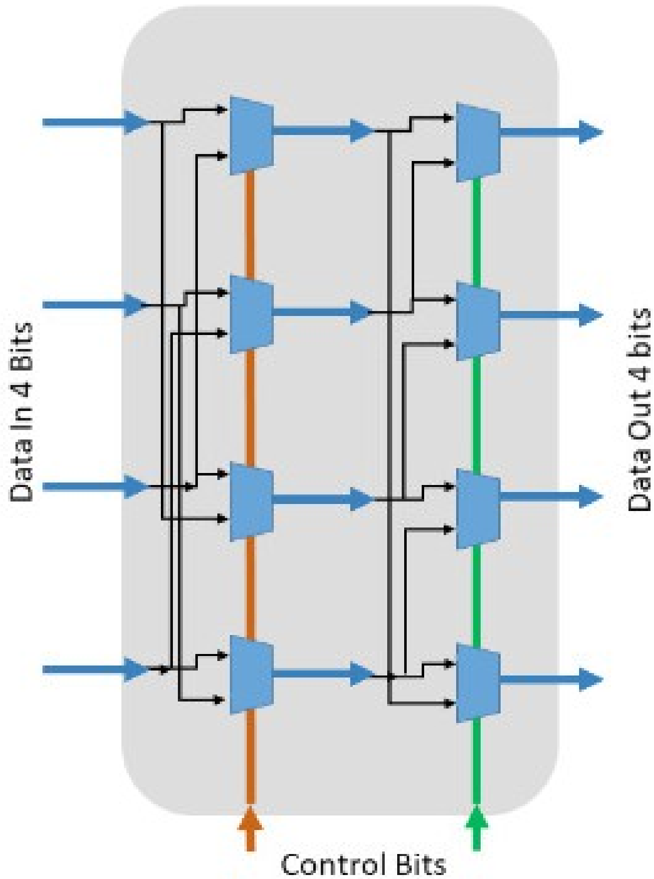


Illustration 1 Mux 2:1 based implementation for 4 bit Barrel Shifter

6.5.2. Implementation Details

Barrel shifter implemented in ac_math.h. Barrel shifter is can be used as combo C-Core or a top design based on user requirements.

Architectural Details

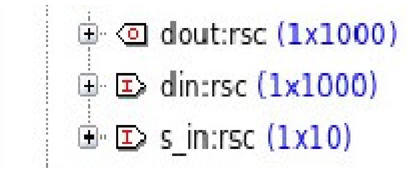
We need to reduce clock overhead to 0 which make sure use of complete cycle for the combo logic. Directive below is to be used set clock overhead.

```
directive set /barrel_shift<1000>::run/core -CLOCK_OVERHEAD 0.000000
```

As design is divided into combinational ccore to reduce the runtime catapult will synthesize each stage in the barrel shifter.

Port Descriptions

Barrel Shifter interface for N bits input is input port N bit along with a select input of width $\lceil \log 2(N) \rceil$ bits and similarly output port of N bit. Illustration below shows ports for N =1000 bits.l.



6.5.3. Using ac_barrel_shift

3. To instantiate barrel_shifter in design user have include ac_math.h and follow the below steps.

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

4. Call of ac_barrel_shifter providing number of bits as template parameter

```
data_ot = ac_barrel_shift<1000>(data_in, ctrl_in) ;
```

as ac_barrel_shift only provide ability to rotate right with the value on control port i.e. if control port have value of 1 that impose input LSB become output MSB and rest change accordingly. User can modify control input or input itself to match the output as rotate left. Create variables for input output and control ports.

```
        const CTL_BITS =   CTRL_BITS ac::nbits<NUM_BITS-1>::val;
  ac_int<NUM_BITS,false> data_in;
  ac_int<CRL_BITS,false> ctrl_in;
  ac_int<NUM_BITS,false> data_ot;
```

5. After assigning the data and control bit, user can run the barrel shifter instance by calling. Is the top interface as well for shifter. data_ot will collect the output data.

6.5.4. Output

Output of the core is right rotated version of input word. Design is feed forward makes F-max only lemmatized by the technology library used.

Design limitation

- Core is designed to rotate right as rotate left can be derived from rotate right itself.
- Design is tested under 1024 bits.

Scheduling, Resource and Area Reports

Below are the RTL synthesis results of Barrel shifter of 1000 bits based on 45nm lib area is 30630.7 µm2

Solution	Area	Delay	Slack	Area(Co...	AreaSeq	TID	MinClkPrd	InputDe...	InputSe...	R2RDelay	R2RSetup
barrel_shift<1000>::run.v1 (extract)	30630.70	0.15	0.38	21555.04	9075.65	Oasys19.1-s007	0.62	0.43	0.04	0.58	0.04