Fixed point algorithmic math package user's guide

By David Bishop (dbishop@vhdl.org)

The fixed point matrix math package was designed to be a synthesizable matrix math package. Because this package allows you to represent a number less than 1.0 it allows you to extend your hardware into areas you could not go before. This packages provides you with several "calculator" math functions. It was designed as a fixed point representation of the math_real package. There are several ways to do many of these functions. Several are represented here.

The VHDL matrix math packages can be downloaded at:
http://www.vhdl.org/fphdl/fixed_alg_pkg.zip

In the ZIP archive you will find the following files:

- "fixed_alg_pkg.vhdl" - Package definition
- "fixed_alg_pkg-body_real.vhdl" – Package body, implemented using the "real" type (unsynthesizable)
- "fixed_alg_pkg-body.vhdl" – Package body, implemented algorithmically.
- "fixed_lookup_pkg.vhdl" – Package definition for table lookup algorithms
- "fixed_lookup_pkg-body.vhdl" – Package body, builds a lookup table to compute the result.
- "test_fixed_alg.vhdl" which test the functionality of these packages.
- "test_fixed_lookup.vhdl" – basically a copy of "test_fixed_alg" used to test the lookup table functions.
- "compile.mti" – A compile script.
- "compile_93.mti" – Compile script for VHDL-93 (to 2002 rules for shared variables)

These packages have been designed for use in VHDL-2008. However, the compatibility version of the packages is provided that works for VHDL-1993. The VHDL-1993 versions of the packages have an "_93" at the end if their file names. These packages were designed to be compiled into the IEEE_PROPOSED library.

Dependencies:

- "fixed_alg_pkg" is dependent on the VHDL-2008 "numeric_std" and "fixed_pkg", and "math_real". The VHDL-1993 version of "fixed_alg_pkg" is dependent on the "IEEE_PROPOSED" library which can be downloaded at http://www.vhdl.org/fphdl/vhdl2008c.zip. It is also dependant on the "math_real"

Overview

The "fixed_alg_pkg" package defines no new types.

This package depends on the IEEE "numeric_std" and "fixed_pkg" packages. The VHDL-93 version is dependent on the "IEEE_PROPOSED" library which can be downloaded at http://www.vhdl.org/fphdl/vhdl2008c.zip. There is also a dependency on the "math_real".

Why two versions you ask? First as a check. To verify that the correct result is being returned. Basically to test the testbench. Second, these algorithms can take some time to run, so in batch simulations it may be faster to use the real math version. The results do not match exactly, but they are very close.

This package uses series to compute values. There is no pipelining done, so you need to do automatic pipelining, insert your own pipelines, or live with the long delays in a multi cycle clocking scheme.

These algorithms are not exhaustively debugged. Please e-mail me if you find a bug. Use at your own risk...

Lookup table package has functions which duplicate the functionality of the fixed_alg_pkg. However all of these functions are implemented via lookup tables. Care is taken to minimize the amount of memory needed.

Index:

Operators:

"**" – overloaded for sfixed ** integer, ufixed ** integer, sfixed ** sfixed, and ufixed ** ufixed

Please see the "power_of" function for documentation. These functions are implemented in the fixed_alg_pkg (real and synthesizable versions). There is also a lookup table version in which the power (second term) is assumed to be aconstant.

Functions:

Precision - This function rounds the input to a given number of binary bits. It is very useful to compare results to see if they are close to the predicted values. I hope to move this function into "fixed_pkg" in the next release of VHDL.
inputs:
arg : ufixed or sfixed
places : Natural number (starting at 1)
overflow_style : see fixed point package documentation
round_style : see fixed point package documentation
result will be the same size as the input argument.

Example:

variable x, y : ufixed (2 downto -15);

x := "000101010101010101"; -- 000.101010101010101, or 1/6
y := precision (x, 10);
The result will be rounded to 10 binary points, or:
y := "000101010101100000";

Floor - This function rounds the result down to the nearest integer. Similar to the floor function in C. I hope to move this function into "fixed_pkg" in the next release of VHDL.
inputs:
arg: ufixed or sfixed
overflow_style : see fixed point package documentation
round_style : see fixed point package documentation
result will be the same size as the input argument.

Ceil - This function rounds the result up to the nearest integer. Similar to the ceil function in C. I hope to move this function into "fixed_pkg" in the next release of VHDL.
inputs:
arg: ufixed or sfixed
overflow_style : see fixed point package documentation
round_style : see fixed point package documentation
result will be the same size as the input argument.

nr_divide - This function does a Newton Raphson divide (by using a loop, no real division involved). This function works by running the Newton Raphson algorithm on the reciprocal, then multiplying that by the "left" input. Yes, it has accuracy issues. Found only in the “fixed_alg_pkg”.
inputs:
l, r : ufixed or sfixed (not mixed)
round_style : see fixed point package documentation
guard_bits : see fixed point documentation
iterations : Number of times to go through the loop (see nr_reciprocal).
Result will be sized to either the sfixed or ufixed division operator as defined in the fixed point package documentation.

Lookup_divide – This function assumes that the divisor is a constant. Function works by computing all positive possible values and storing them in a lookup table.
l, r : ufixed or sfixed (not mixed)
round_style : see fixed point package documentation
guard_bits : see fixed point documentation
iterations : Number of times to go through the loop (see nr_reciprocal).
Result will be sized to either the sfixed or ufixed division operator as defined in the fixed point package documentation.

nr_reciprocal - This function does a Newton Raphson reciprocal. The algorithm used is:
c1 = c0 * (2 - c0*arg)
and looping until a given accuracy is achieved. This loop is shortened by creating the correct seed, or "c0". This is done shifting the argument so the correct power of 2 of the result is achieved. Then running the loop. For anything less than 10 bits, 3 loops will work. 6 loops will give an accurate result for a 50 bit number. Thus there will be a trade off as to which division algorithm is best.
arg: ufixed or sfixed
round_style : see fixed point package documentation
guard_bits : see fixed point documentation
iterations : Number of times to go through the loop.
If the number of iterations is set to "0" then it will make an assumption given the length of the number fed into the algorithm. A number of 6 will grantee accuracy, the the expense of logic.
Result will be sized to either the sfixed or ufixed division operator as defined in the fixed point package documentation.

sqrt - Performs a square root using Newton's iteration:
root := (1 + arg) / 2, then root := (root + (arg/root))/2 until done.
Yes, this function does involved a divide, making it fairly slow. I recommend that you use the inverse_sqrt function which if much more hardware efficient if possible. The number of iterations necessary can be assumed correctly by the length of the argument.
arg : sfixed or ufixed. A negative input will cause an error
overflow_style : see fixed point package documentation
round_style : see fixed point package documentation
guard_bits : see fixed point documentation
Result will be the same size and type as the input.

cbrt - Performs a cube root using Newton's iteration:
root := (arg + 2) / 3, then root := ((arg/root**2) + 2(root)))/3 until done.
Another fairly slow function. Every number has 3 cube roots, but this function only delivers the positive one. A negative input will result in a negative result.
arg : sfixed or ufixed.
overflow_style : see fixed point package documentation
round_style : see fixed point package documentation
guard_bits : see fixed point documentation
Result will be the same size and type as the input.

inverse_sqrt - This function returns 1/sqrt(x) using a Newton Raphson iteration:

y1 = (y0*(3-y0*y0*arg))/2, where the seed "y0" is computed figuring out the power of 2 of the result. No division is involved in this algorithm. The number of iterations can be assumed by the number of bits in the result. However you can override this value if necessary. 4 iterations are usually good enough depending on the seed. 6 will be good enough for 50 bits of resolution.
arg : sfixed or ufixed.
overflow_style : see fixed point package documentation
round_style : see fixed point package documentation
guard_bits : see fixed point documentation
iterations : Number of times to go through the loop.
Result will be of the same size and type as the input.

exp - This function returns E^X. It is done by using the series:
e**x := 1 + x + (x**2 / 2!) + (x**3 / 3!) ...
This algorithm is done without using division (1/3! is a constant), however it needs one "term" for every 2 bits of accuracy.
arg: sfixed or ufixed
overflow_style : see fixed point package documentation
round_style : see fixed point package documentation
guard_bits : see fixed point documentation
iterations : Number of times to go through the loop.
Result will be of the same size and type as the input.

Log - This function takes in two arguments and returns the Log of the first argument using the second argument as the base. The algorithm used is: LogY(X) = ln(X) / ln(y), so it isn't very efficient.
arg : sfixed or ufixed (must be a positive number)
base : POSITIVE, sfixed or ufixed (must be a positive number)
overflow_style : see fixed point package documentation
round_style : see fixed point package documentation
guard_bits : see fixed point documentation
iterations : Number of times to go through the loop, passed to "ln" function
Result will be of the same size and type as the "arg" input.

power_of - This function performs l^r. This is done via a log based operation which does involve some division. Loop iterations are calculated from the length of the output.
l : sfixed or ufixed
r : sfixed or ufixed
overflow_style : see fixed point package documentation
round_style : see fixed point package documentation
guard_bits : see fixed point documentation
Result will be of the same size and type as the left or "l" input.

ln - Performs the natural log of the argument. The series performed
here is: TERM = ((x-1)/(x+1)), ln(x) = 2(TERM + (1/3)*TERM**3 + ...)
There is one divide, at the beginning of the routine.
arg : sfixed or ufixed (must be a positive number)
overflow_style : see fixed point package documentation
round_style : see fixed point package documentation
guard_bits : see fixed point documentation
iterations : Number of times to go through the loop, You need about
1 iteration for every 4 bits of result. If a "0" is passed as the
number of iterations then iterations is arg'length/4.

"**" - Operator, calls "power_of" function using the default
parameters.

sin - Performs a "sin" function using this series:
sin(x) = x - x**3/3! + x**5/5! - x**7/7! ....
no divides involved, and is fairly efficient. The function does one
iterations for every 4 bits of precision. The input is expected to be
in radians. If the input is larger that 2*PI, or less than 0, then a
function is called with normalizes the input to the 0 to 2*PI range.
arg : sfixed - input in Radians
overflow_style : see fixed point package documentation
round_style : see fixed point package documentation
guard_bits : see fixed point documentation
result will be the same size as the "arg" input, and will be a number
between -1 and 1.

cos - Performs a "cos" function using this series:
cos(x) = 1 - x**2/2! + x**4/4! - x**6/6! ....
no divides involved, and is fairly efficient. The function does one
iterations for every 4 bits of precision. The input is expected to be
in radians. If the input is larger that 2*PI, or less than 0, then a
function is called with normalizes the input to the 0 to 2*PI range.
arg : sfixed - input in Radians
overflow_style : see fixed point package documentation
round_style : see fixed point package documentation
guard_bits : see fixed point documentation
result will be the same size as the "arg" input, and will be a number
between -1 and 1.

tan - Performs a tangent function using this series:
This function is currently performed by calculating sin(x)/cos(x), it
works, but it is probably not as efficient as I would like.
arg : sfixed - input in Radians
overflow_style : see fixed point package documentation

round_style : see fixed point package documentation
guard_bits : see fixed point documentation
result will be the same size as the "arg" input, and will saturate as the number goes to infinity.

arcsin - Performs an "arcsin" function (angle who's "sin" is). This is done by calculating: arcsin(x) = arctan (x / sqrt (1 - x**2)), which is defiantly not the most efficient mechanism, but it works.
arg : sfixed - between -1.0 and 1.0
overflow_style : see fixed point package documentation
round_style : see fixed point package documentation
guard_bits : see fixed point documentation
iterations: passed to the arctan function.
result will be the same size as the input, and returned in radians.

arccos - Performs an "arccos" function (angle who's "cos" is). This is done by calculating: arccos(x) = PI/2 - arcsin(arg), which is defiantly not the most efficient mechanism, but it works.
arg : sfixed - between -1.0 and 1.0
overflow_style : see fixed point package documentation
round_style : see fixed point package documentation
guard_bits : see fixed point documentation
iterations: passed to the arctan function.
result will be the same size as the input, and returned in radians.

arctan - Performs an "arctan" function (angle who's tangent is). This one is a bit complicated. After several trials with standard equations I came up with:
arctan (x) = PI/2 - arctan(1/x)
which only works for small angles, so I use a "half_angle" formula to compute arguments larger than sqrt(2)/4.
arg : sfixed
overflow_style : see fixed point package documentation
round_style : see fixed point package documentation
guard_bits : see fixed point documentation
iterations: one iteration for every 6 bits of precision.
result will be the same size as the input, and returned in radians.

sinh - Hyperbolic sin function. This one also took a few trials. The most efficient function I could come up with was:
sinh(x) = (e^x - e^(-x))/2 which uses the very efficient "exp" function.
overflow_style : see fixed point package documentation
round_style : see fixed point package documentation

guard_bits : see fixed point documentation
iterations: passed to the "exp" function.
result will be the same size as the input.

cosh - Hyperbolic cos function.
The most efficient function I could come up with was:
cosh(x) = (e^x + e^(-x))/2 which uses the very efficient "exp"
function.
overflow_style : see fixed point package documentation
round_style : see fixed point package documentation
guard_bits : see fixed point documentation
iterations: passed to the "exp" function.
result will be the same size as the input.

tanh - Hyperbolic tan function.
The most efficient function I could come up with was:
tanh(x) = (e^2x - 1) / (e^2x + 1) which uses the very efficient "exp"
function, but this time involving a divide.
overflow_style : see fixed point package documentation
round_style : see fixed point package documentation
guard_bits : see fixed point documentation
iterations: passed to the "exp" function.
result will be the same size as the input.