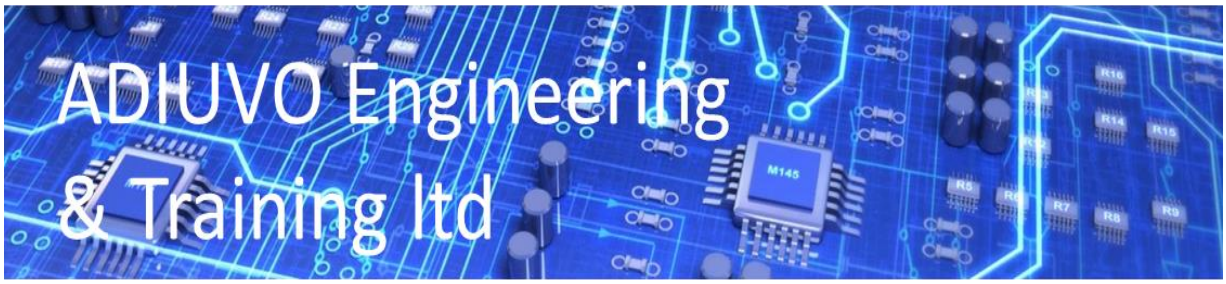

White Paper 1

How to do Math's in FPGA – Using VHDL 2008

Following the introduction of VHDL 93, which introduced the numeric_std package and the signed and unsigned types, implementing fixed point maths has been fairly straight forward. Using this package, we can implement mathematics using a fixed point representation. However, to implement a fixed point algorithm we need to understand the simple rules regarding fixed point operations.

VHDL 2008 introduced two new packages which support synthesis of fixed and floating point operations named fixed_pkg and float_pkg which significantly simplified how we can perform mathematics in FPGA's. In this article we will look at how we can use the fixed_pkg to implement fixed point maths.

Before I explain how we can use the functionality provided by VHDL 2008 to reduce the complexity, I think it is good idea to explain the basics such that we understand just how powerful these new packages are.

Representation of Numbers.

There are two methods of representing numbers within a design, fixed or floating-point number systems. Fixed-point representation maintains the decimal point within a fixed position allowing for straight forward arithmetic operations. The major drawback of fixed-point representation is that to represent larger numbers, or to achieve a more accurate result with fractional numbers, a larger number of bits are required. A fixed point number consists of two parts called the integer and fractional parts.

Floating point representation allows the decimal point to float to different places within the number depending upon the magnitude. Floating point numbers are divided into two parts the exponent and the mantissa. This is very similar to scientific notation which represents a number as A times 10 to the power B, where A is the mantissa and B is the exponent. However, the base of the exponent in a floating-point number is base 2, that is A times 2 to the power B. The floating-point number is standardized as IEEE / ANSI Standard 754 and utilises an 8-bit exponent and a 24-bit mantissa.

Due to the complexity of floating point numbers, as designers we tend wherever possible use fixed-point representation.

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}

The above fixed-point number is capable of representing an unsigned number of between 0.0 and 255.9906375, or a signed number of between –128.9906375 and 127.9906375, using twos complement representation. Within a design we have the choice to use either unsigned or signed numbers, typically this will be constrained by the algorithm being implemented. Unsigned numbers are capable of representing a range of 0 to 2n – 1, and always represent positive numbers. While the range of a Signed number depends upon the encoding scheme used, Sign and Magnitude, Ones Complement or Twos Complement.

Both the numeric_std and fixed_pkg use twos complement numbers to represent negative numbers. With Twos Complement representation positive numbers are represented in the same manner as unsigned numbers. While negative numbers are represented as the binary number you add to a positive number of the same magnitude to get zero. A negative twos complement number is calculated by first taking the ones complement (inversion) of the positive number and then adding one to it. The twos complement number system allows subtraction of one number form another by performing an addition of the two numbers. The range a twos complement number can represent is given by

$$-(2n\text{-}1) \text{ to } +(2n\text{-}1 - 1)$$

One method we can use to convert a number to its twos complement format is to work right to left leaving the number the same until the first one is encountered, after this each bit is inverted.

Fixed Point Mathematics.

The normal way of representing the split between integer, fractional bits within a fixed-point number is x,y where x represents the number of integer bits and y the number of fractional bits. For example, 8,8 represents 8 integer bits and 8 fractional bits while 16,0 represents 16 integers and 0 fractional. This format is often called Q format, this is given as Qm.n where m represents the number of integer bits and n represents the number of fractional bits. As such the examples above could be displayed as Q8.8 and Q16.0. Many applications use a format such as Q8, in this case it shows just the number of fractional bits such that the engineer understands where the decimal point in the vector resides.

In many cases the correct choice of the number of integer and fractional bits required will be undertaken at design time, normally following conversion from a floating point algorithm. Thanks to the flexibility of FPGA's, we can represent a fixed-point number of any bit length; the number of integer bits required depends upon the maximum integer value the number is required to store, while the number of fractional bits will depend upon the accuracy of the final result. To determine the number of integer bits required we can use the following equation

$$\text{Integer Bits Required} = Ceil\left(\frac{LOG_{10}\text{Integer\_Maximum}}{LOG_{10}2}\right)$$

For example, the number of integer bits required to represent a value between 0.0 and 423.0 is given by

$$9 = Ceil\left(\frac{\text{LOG}_{10}423}{\text{LOG}_{10}2}\right)$$

Meaning we would need 9 integer bits, allowing a range of 0 to 511 to be represented

Obviously using a fixed point number system does result in a quantisation error as is it not possible to encode the exact fractional value. In this case we have two options, first verify the loss of accuracy is acceptable and the performance of the algorithm is not adversely impacted, while the second is to increase the number of fractional bits used until the performance is acceptable.

Fixed Point Rules

To perform addition, subtraction the decimal points of both numbers must be aligned, for division they do not have to be aligned but it can open up some issues as the scaling of the result will be the difference between the two numbers and it is possible to send them negative. As such it is a good idea if you can align your division decimal points.

That is a x,8 number can only be added to, subtracted from or divided by a number which is also in a x,8 representation. To perform arithmetic operations on numbers of different x,y format we must first ensure the decimal points are aligned. To align a number to a different format you have two choices, either multiply the number with more integer bits by 2^X or divide the number with the least number of integer bits by 2^X. When dividing by 2^X the accuracy will be reduced and may lead to a result which is outside the allowable tolerance. As all numbers are stored in base two scaling up or down can be achieved easily in a FPGA through shifting one place to the left or right for each power of 2 required to balance the two decimal points. To add together two number which are scaled 8,8 and 9,7 you can either scale up the 9,7 number by a factor or 2^1 or scale the 8,8 format down to an 9,7 format if the loss of a least significant bit is acceptable. For example, adding 234.58 and 312.732, which are stored in an 8,8 and 9,7 formats respectively. The first step is to determine the actual 16 bit numbers, which will be added together.

$$234.58 * 2^8 = 60052.48$$

$$312.732 * 2^7 = 40029.69$$

The two numbers to be added together are 60052 and 40029 however, before the two numbers can be added together the decimal point must be aligned. To align the decimal points by scaling up the number with a largest number of integer bits, the 9,7 format number must be scaled up by a factor of 2^1

$$40029 * 2^1 = 80058$$

The result can then be calculated by performing an addition of

$$80058 + 60052 = 140110$$

This represents 547.3046875 in a 10,8 format ($140110 / 2^8$) the result becomes 10,8 not 9,8 as we have to take into account the results of the MSB's being added. If we do not account for this, then we do not have a correct result.

When multiplying two numbers together the decimal points do not need to be aligned as the multiplication will provide a result which is X1 + X2, Y1 + Y2 wide. Multiplying two numbers, which are formatted 14,2 and 10,6, will produce a result, which is formatted 25 integer bits and 8 fractional bits. The additional bit on the integer again comes from the potential growth in side due to the multiplication.

Multiplication is very useful and we can use it to save the requirement to perform a division in some instances where we can multiply by the reciprocal of the divisor instead. Using this approach, we can reduce the complexity of the design significantly but only if the division is fixed. For example, to

divide the number 312.732 represented in 9,7 (40029) format by 15 the first stage is to calculate the reciprocal of the divisor.

$$\frac{1}{15} = 0.066666'$$

This reciprocal must then be scaled up, to be represented within a 16-bit number

65536 * 0.06666 = 4369

This will produce a result which is formatted 9, 23 when the two numbers are multiplied together

4369 * 40029 = 174886701

The result of this multiplication is thus

$$\frac{174886701}{8388608} = 20.8481193781$$

While the expected result is 20.8488, if the result is not accurate enough then the reciprocal can be scaled up by a larger factor to produce a more accurate result. Therefore, never divide by a number when you can multiply by the reciprocal.

Overflow.

When implementing algorithms, we must ensure that the result is not larger than what is capable of being stored within the result register. When this condition occurs it is known as overflow, when overflow occurs the stored result will be incorrect and the most significant bits are lost.

IEEE Fixed Package

This package introduces two new signal types

- ufixed – for unsigned numbers
- sfixed – for signed numbers

What is important about how we can use these types is in how they represent where the decimal point is located. We can declare both of these signals using the following format

- integer bits are represented in the range MSB down to 0
- fractional bits are represented in the range -1 down to LSB

With the decimal point located between the 0 and -1 bit as would be normal, this means a signal declaration looks like the following

SIGNAL example : ufixed(3 DOWNTO -3);

Which represents the vector of 000.000 allowing for a range of 0.0 to 7.875 when representing unsigned number.

When we declare signals like this we can declare them as either all integer or all fractional as necessary for our algorithm implementation.

To help initialise signals, variables and constants in our algorithm we can use the to_ufixed and to_sfixed, these can be used with integers, real, ufixed, sfixed and std_logic_vectors.

The packages should be supported with most tools which support VHDL 2008 however, as many of these tools sometimes take a little time to implement the latest language features, the IEEE has made available both the fixed and float packages as a IEEE_proposed library which can be downloaded and used in your VHDL 1993 designs if they are not already included with your tool chain.

Within Vivado (2015.4) we can use these libraries quite easily the first thing we need to do is declare the libraries within our design file using the syntax below.

```
library ieee_proposed;
use ieee_proposed.fixed_float_types.all;
use ieee_proposed.fixed_pkg.all;
```

We can then use the signed and unsigned types and its functions to implement the algorithm we desire.

The package provides support for all mathematical, logical, comparison and other operators commonly contained within type package definitions. Some of the more exciting functions provided are

- Resize - Resizes the word
- Add Carry - Eases the implementation of accumulators
- Salb – Scale to a power of two
- Modulo – Return the modulus of two numbers
- Divide – Provides more user control on rounding style and guard bits than the “/” operator
- Reciprocal – Calculates the reciprocal of a number
- Remainder – Provides the remainder from

Conclusion

Implementing fixed point math’s in FPGA has always been pretty simple however the IEEE 2008 fixed library introduces a new package which makes the use of these even simpler as the location of the decimal point is easier to determine for each vector. However, we must still follow the rules outlined above for performing fixed point mathematics.

Links

https://standards.ieee.org/downloads/1076/1076-2008/