The Basics of FPGA Mathematics

by Adam Taylor
Principal Engineer
EADS Astrium
aptaylor@theiet.org

One of the main advantages of the FPGA is its ability to perform mathematical functions as desired. Here's a refresher on the basic rules and methods involved.

One of the many benefits of an FPGA-based solution is the ability to implement a mathematical algorithm in the best possible manner for the problem at hand. For example, if response time is critical, then we can pipeline the stages of mathematics. But if accuracy of the result is more important, we can use more bits to ensure we achieve the desired precision. Of course, many modern FPGAs also provide the benefit of embedded multipliers and DSP slices, which can be used to obtain the optimal implementation in the target device.

Let's take a look at the rules and techniques that you can use to develop mathematical functions within an FPGA or other programmable device.

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 straightforward arithmetic operations. The major drawback of the fixed-point system is that to represent larger numbers or to achieve a more accurate result with fractional numbers, you will need to use a larger number of bits. A fixed-point number consists of two parts, integer and fractional.

Floating-point representation allows the decimal point to "float" to different places within the number, depending upon the magnitude. Floating-point numbers, too, are divided into two parts, the exponent and the mantissa. This scheme is very similar to scientific notation, which represents a number as A times 10 to the power of 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 of B. The floating-point number is standardized by IEEE/ANSI standard 754-1985. The basic IEEE floating-point number utilizes an 8-bit exponent and a 24-bit mantissa.

Due to the complexity of floating-point numbers, we as designers tend wherever possible to use fixed-point representations. The above fixed-point number is capable of representing an unsigned number between 0.0 and 255.9906375 or a signed number between –128.9906375 and 127.9906375 using two's complement representation. Within a design you have the choice—typically constrained by the algorithm you are implementing—to use either unsigned or signed numbers. Unsigned numbers are capable of representing a range of 0 to $2^n - 1$, and always represent positive numbers. By contrast, the range of a signed number depends upon the encoding scheme used: sign and magnitude, one's complement or two's complement.

The sign-and-magnitude scheme utilizes the left-most bit to represent the sign of the number (0 = positive, 1 = negative). The remainder of the bits represent the magnitude. Therefore, in this system, positive and negative numbers have the same magnitude but the sign bit differs. As a result, it is possible to have both a positive and a negative zero within the sign-and-magnitude system.

One's complement uses the same unsigned representation for positive numbers as sign and magnitude. However, for negative numbers it uses the inversion (one's complement) of the positive number.

Two's complement is the most widely used encoding scheme for representing signed numbers. Here, as in the other two schemes, 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. You calculate a negative two's complement number by first taking the one's complement (inversion) of the positive number and then adding one to it. The two's complement number system allows you to subtract one number from another by performing an addition of the two numbers. The range a two's complement number can represent is given by

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

One way to convert a number to its two's complement format is to work right to left, leaving the number the same until you encounter the first "1." After this point, each bit is inverted.

FIXED-POINT MATHEMATICS

The normal way of representing the split between integer and 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 integer and 0 fractional. In many cases you will determine the correct number of integer and fractional bits required at design time, normally following conversion from a floating-point algorithm. Thanks to the flexibility of FPGAs, 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, 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

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

That means you would need 9 integer bits, allowing a range of 0 to 511 to be represented. Representing the number using 16 bits would allow for 7 fractional bits. The accuracy this representation would be capable of providing is given by

$$Accuracy = \left(\frac{Actual\_Value - FPGA\_Value}{2^{\text{Fractional Bits}}}\right) = 100$$

You can increase the accuracy of a fixed-point number by using more bits to store the fractional number. When designing, there are times when you may wish to store only fractional numbers (0,16), depending upon the size of the number you wish to scale up. Scaling up by 2^{16} may yield a number that still does not provide an accurate enough result. In this case you can multiply up by the power of 2, such that the number can be represented within a 16-bit number. You can then remove this scaling at a further stage within the implementation. For example, to represent the number $1.45309806319x10^{-4}$ in a 16-bit number, the first step is to multiply it by 2^{16}.

$$65536 \bullet 1.45309806319x10^{-4} = 9.523023$$

Storing the integer of the result (9) will result in the number being stored as $1.37329101563x10^{-4}$ (9 / 65536). This difference between the number required to be stored and the stored number is substantial and could lead to an unacceptable error in the calculated result. You can obtain a more accurate result by scaling the number up by a factor of 2. The result will be between 32768 and 65535, therefore still allowing storage in a 16-bit number. Using the earlier example of storing $1.45309806319x10^{-4}$, multiplying by a factor of 2^{28} will yield a number that can be stored in 16 bits and will be highly accurate of the desired number.

$$268435456 \bullet 1.45309806319x10^{-4} = 39006.3041205$$

The integer of the result will give you a stored number of $1.45308673382x10^{-4}$, which will provide for a much more accurate calculation, assuming you can address the scaling factor of 228 at a later stage within the calculation. For example, multiplying the scaled number with a 16-bit number scaled 4,12 will produce a result of 4,40 (28 + 12). The result, however, will be stored in a 32-bit result.

FIXED-POINT RULES

To add, subtract or divide, the decimal points of both numbers must be aligned. That is, you can only add to, subtract from or divide an x,8 number by a number that is also in an x,8 representation. To perform arithmetic operations on numbers of a different x,y format, you 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 fewest integer bits by 2^X. Division, however, will reduce your accuracy and may lead to a result that is outside the allowable tolerance. Since all numbers are stored in base-two scaling, you can easily scale the number up or down in an FPGA by shifting one place to the left or right for each power of 2 required to balance the two decimal points. To add together two numbers that 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 a 9,7 format, if the loss of a least-significant bit is acceptable.

For example, say you want to add 234.58 and 312.732, which are stored in an 8,8 and a 9,7 format respectively. The first step is to determine the actual 16-bit numbers that will be added together.

$$234.58 \bullet 2^8 = 60052.48$$

$$312.732 \bullet 2^7 = 40029.69$$

The two numbers to be added are 60052 and 40029. However, before you can add them you must align the decimal points. To align the decimal points by scaling up the number with a largest number of integer bits, you must scale up the 9,7-format number by a factor of 2^1.

$$40029 \bullet 2^1 = 80058$$

You can then calculate the result by performing an addition of

$$80058 + 60052 = 140110$$

This represents 547.3046875 in a 10,8 format ($140110 / 2^8$).

When multiplying two numbers together, you do not need to align the decimal points, as the multiplication will provide a result that is X1 + X2, Y1 + Y2 wide. Multiplying two numbers that are formatted 14,2 and 10,6 will produce a result that is formatted as 24 integer bits and 8 fractional bits.

You can multiply by a fractional number instead of using division within an equation through multiplying by the recip-

rocal of the divisor. This approach can reduce the complexity of your design significantly. 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.

$$\frac{1}{15} = 0.0666'$$

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

$$65536 \bullet 0.06666 = 4369$$

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

$$4369 \bullet 40029 = 174886701$$

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 you can scale up the reciprocal by a larger factor to produce a more accurate result. Therefore, never divide by a number when you can multiply by the reciprocal.

ISSUES OF OVERFLOW

When implementing algorithms, the result must not be larger than what is capable of being stored within the result register. Otherwise a condition known as overflow occurs. When that happens, the stored result will be incorrect and the most significant bits are lost. A very simple example of overflow would be if you added two 16-bit numbers, each with a value of 65535, and the result was stored within a 16-bit register.

$$65535 + 65535 = 131070$$

The above calculation would result in the 16-bit result register containing a value of 65534, which is incorrect. The simplest way to prevent overflow is to determine the maximum value that will result from the mathematical operation and use this equation to determine the size of the result register required.

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

If you were developing an averager to calculate the average of up to fifty 16-bit inputs, the size of the required result register could be calculated.

$$50 \bullet 65535 = 3276750$$

Using this same equation, this would require a 22-bit result register to prevent overflow occurring. You must also take care, when working with signed numbers, to ensure there is no overflow when using negative numbers. Using the averager example again, taking 10 averages of a signed 16-bit number returns a 16-bit result.

$$10 \bullet -32768 = -327680$$

Since it is easier to multiply the result by a scaled reciprocal of the divisor, you can multiply this number by $1/10 \bullet 65536 = 6554$ to determine the average.

$$-32768 \bullet 6554 = -2147614720$$

This number when divided by 216 equals -32770, which cannot be represented correctly within a 16-bit output. The module design must therefore take the overflow into account and detect it to ensure you don't output an incorrect result.

REAL-WORLD IMPLEMENTATION

Let's say that you are designing a module to implement a transfer function that is used to convert atmospheric pressure, measured in millibars, into altitude, measured in meters.

$$-0.0088x2 + 1.7673x + 131.29$$

The input value will range between 0 and 10 millibars, with a resolution of 0.1 millibar. The output of the module is required to be accurate to +/-0.01 meters. As the module specification does not determine the input scaling, you can figure it out by the following equation.

$$4 = Ceil\left(\frac{LOG^{10}\,10}{LOG^{10}2}\right)$$

Therefore, to ensure maximum accuracy you should format the input data as 4 integer and 12 fractional bits. The next step in the development of the module is to use a spreadsheet to calculate the expected result of the transfer function across the entire input range using the unscaled values. If the input range is too large to reasonably achieve this, then calculate an acceptable number of points. For this example, you can use 100 entries to determine the expected result across the entire input range.

Input (millibar)	Output (meters)
0	131.2900
0.1	131.4666
0.2	131.6431
0.3	131.8194
0.4	131.9955
0.5	132.1715
0.6	132.3472

Once you have calculated the initial unscaled expected values, the next step is to determine the correct scaling factors for the constants and calculate the expected outputs using the scaled values. To ensure maximum accuracy, each of the constants used within the equation will be scaled by a different factor.

The scaling factor for the first polynomial constant (A) is given by

$$8 = Ceil\left(\frac{LOG^{10}\,133.29}{LOG^{10}2}\right)$$

The second polynomial constant (B) scaling factor is given by

$$1 = Ceil\left(\frac{LOG^{10}\,1.7673}{LOG^{10}2}\right)$$

The final polynomial constant (C) can be scaled up by a factor of 2^{16}, as it is completely fractional.

Polynomial Constant	Unscaled	Scaled
A	133.29	33610
B	1.77	57910
C	-0.01	-577

These scaling factors allow you to calculate the scaled spreadsheet, as shown in Table 1. The results of each stage of the calculation will produce a result that will require more than 16 bits.

The calculation of the Cx^2 will produce a result that is 32 bits long formatted 4,12 + 4,12 = 8,24. This is then multiplied by the constant C, producing a result that will be 48 bits long formatted 8,24 + 0,16 = 8,40. For the accuracy required in this example, 40 bits of fractional representation is excessive. Therefore, the result will be divided by 2^{32} to produce a result with a bit length of 16 bits formatted 8,8. The same reduction to 16 bits is carried out upon the calculation of Bx to produce a result formatted 5,11.

The result is the addition of columns Cx^2, Bx and A. However, to obtain the correct result you must first align the radix points, either by shifting up A and Cx^2 to align the numbers in an x,11 format, or shifting down the calculated Bx to a format of 8,8, aligning the radix points with the calculated values of A and Cx^2.

In this example, we shifted down the calculated value by 2^3 to align the radix points in an 8,8 format. This approach simplified the number of shifts required, thus reducing the logic needed to implement the example. Note that if you cannot achieve the required accuracy by shifting down to align the radix points, then you must align the radix points by shifting up the calculated values of A and Cx^2. In this example, the calculated result is scaled up by a power of 2^8. You can then scale down the result and compare it against the result obtained with unscaled values. The difference between the calculated result and the expected result is then the accuracy, using the spreadsheet commands of MAX() and MIN(), for the maximum and minimum error of the calculated result that can be obtained across the entire range of spreadsheet entries.

Once the calculated spreadsheet confirms that you can achieve the required accuracy, you can write and simulate the RTL code. If desired, you could design the testbench such that the input values are the same as those used in the spreadsheet. This allows you to compare the simulation outputs against the spreadsheet-calculated results to ensure the correct RTL implementation.

Input Scaled	C	B	A	Result	Result Scaled	Expected Result	Difference
0	0	0	33610	33610	131.289	131.2900	0.0009
409	-6	361	33610	33655	131.465	131.4666	0.0018
819	-24	723	33610	33700	131.641	131.6431	0.0025
1228	-52	1085	33610	33745	131.816	131.8194	0.0030
1638	-93	1447	33610	33790	131.992	131.9955	0.0033
2048	-145	1809	33610	33835	132.168	132.1715	0.0035
2457	-208	2171	33610	33880	132.344	132.3472	0.0035
2867	-283	2533	33610	33925	132.520	132.5228	0.0033

Table 1 – Real results against the fixed-point mathematics

RTL IMPLEMENTATION

The RTL example uses signed parallel mathematics to calculate the result within four clock cycles. Because of the signed parallel multiplication, you must take care to correctly handle the extra sign bits generated by the multiplications.

```
ENTITY transfer_function IS PORT(
sys_clk : IN std_logic;
reset : IN std_logic;
data : IN std_logic_vector(15 DOWNTO 0);
new_data : IN std_logic;
result : OUT std_logic_vector(15 DOWNTO 0);
new_res : OUT std_logic);
END ENTITY transfer_function;

ARCHITECTURE rtl OF transfer_function IS
-- this module performs the following
transfer function -0.0088x2 + 1.7673x +
131.29
-- input data is scaled 8,8, while the
output data will be scaled 8,8.
-- this module utilizes signed parallel
mathematics

TYPE control_state IS (idle, multiply,
add, result_op);
CONSTANT c : signed(16 DOWNTO 0) :=
to_signed(-577,17);
CONSTANT b : signed(16 DOWNTO 0) :=
to_signed(57910,17);
CONSTANT a : signed(16 DOWNTO 0) :=
to_signed(33610,17);
SIGNAL current_state : control_state;
SIGNAL buf_data : std_logic; --used to
detect rising edge upon the new_data
SIGNAL squared : signed(33 DOWNTO 0); --
register holds input squared.
SIGNAL cx2 : signed(50 DOWNTO 0); --regis-
ter used to hold Cx2
SIGNAL bx : signed(33 DOWNTO 0); -- regis-
ter used to hold bx
SIGNAL res_int : signed(16 DOWNTO 0); --
register holding the temporary result

BEGIN
fsm : PROCESS(reset, sys_clk)
BEGIN
IF reset = '1' THEN
 buf_data <= '0';
 squared <= (OTHERS => '0');
 cx2 <= (OTHERS => '0');
 bx <= (OTHERS => '0');
 result <= (OTHERS => '0');
 res_int <= (OTHERS => '0');
 new_res <= '0';
 current_state <= idle;
ELSIF rising_edge(sys_clk) THEN
buf_data <= new_data;
CASE current_state IS
WHEN idle =>
 new_res <= '0';
 IF (new_data = '1') AND (buf_data = '0')
 THEN --detect rising edge new data
   squared <= signed( '0'& data) *
   signed('0'& data);
   current_state <= multiply;
 ELSE
   squared <= (OTHERS =>'0');
   current_state <= idle;
END IF;
WHEN multiply =>
 new_res <= '0';
 cx2 <= (squared * c);
 bx <= (signed('0'& data)* b);
 current_state <= add;
WHEN add =>
 new_res <= '0';
 res_int <= a + cx2(48 DOWNTO 32) +
   ("000"& bx(32 DOWNTO 19));
 current_state <= result_op;
WHEN result_op =>
 result <= std_logic_vector(res_int
 (res_int'high -1 DOWNTO 0));
 new_res <= '0';
 current_state <= idle;
END CASE;
END IF;
END PROCESS;
END ARCHITECTURE rtl;
```

The architecture of FPGAs makes them ideal for implementing mathematical functions, although the implementation of your algorithm may take a little more initial thought and modeling in system-level tools such as MATLAB® or Excel. You can quickly implement mathematical algorithms once you have mastered some of the basics of FPGA mathematics.