VHDL Modeling of Booth Radix-4 Floating Point Multiplier for VLSI Designer's Library

WAI-LEONG PANG, KAH-YOONG CHAN, SEW-KIN WONG, CHOON-SIANG TAN
Faculty of Engineering
Multimedia University
Persiaran Multimedia, Cyberjaya
MALAYSIA
wlpang@mmu.edu.my, kychan@mmu.edu.my, skwong@mmu.edu.my

Abstract: - Floating point arithmetic computation has been widely used today in graphics, digital signal processing, image processing and other applications. Multiplication is the most complex calculation that used in most digital electronic circuit. The multiplier may have large chip area density, high complexity, and is a time consuming computation because the output data size is twice larger than input data size. Complex floating point multiplication required more time to process data and is highly recommended to improve the computation speed. The performance in terms of computation and processing speed is one of the major factors in today's Very/Ultra Large Scale Integration (VLSI/ULSI) system design. The objective of this research is to design a 32-bit floating point multiplier for Very high speed integrated circuit Hardware Description Language (VHDL) designer's library that consists of mantissas multiplier, normalizer, exponent adder, and signer for VHDL designer's library that lack of floating point multiplier module. Booth radix-4 algorithm is used in the multiplier, mainly due to the simplicity of this algorithm to be modeled using VHDL and at the same time it provides good performance. The 32-bit floating point multiplier is tested on Arria II GX chip to determine their performance in terms of slack, maximum frequency and minimum clock period by using TimeQuest Timing Analyzer. Booth radix-4 multiplier in Arria II GX (EP2AGX45CU17I3) produces a maximum frequency of 206.14 MHz and minimum allowed clock period of 5 ns. Benchmarking has been carried out between the Booth radix-4 and Wallace Tree multipliers, since Wallace Tree multiplier can provide better performance to the VLSI system design. The resource consumption of Booth radix-4 multiplier is 88.8% less than the Wallace Tree multiplier and the performance of Booth radix-4 multiplier is almost equal to the Wallace Tree multiplier.

Key-Words: - VHDL, Booth Radix-4, Floating Point Multiplier

1 Introduction

Floating point computation has been widely used today in graphics, Digital Signal Processing (DSP), image processing and other applications. Floating point multiplication is a critical module in many applications especially for Graphic Processing Unit (GPU), image recognition, and digital signal processing applications such as wired and wireless communications involving a large dynamic range. One of the most demanding example applications that use floating point multiplier is Three-Dimensional (3D) GPU for gaming. Real time 3D graphics display is relied on the GPU's floating point unit to perform floating point calculations. The 3D object is rendered by many numbers of polygons to form. The floating point multiplier is used to calculate the changes of polygons, which involved some mathematical approach such as matrix and vector calculation. Floating point numbers gives high level of precision, produces much better detail of the 3D model. Another example that uses floating point multiplier is the navigation system in radar for identification, tracking and detection. The radar system may be scanned within a range between starting point to destination point for target acquisition. It requires wide dynamic ranges that use multiply or divide operation or matrix inversions to calculate polar coordinates to detect the exact target location precisely. Since the subset of the range must be determined in real time during operation, this is impossible to design this navigation system using fixed point DSP due to its limited range. Fortunately, floating point DSP is the perfect choice in designing this navigation system because it provides wide dynamic range and high precision [1].

Such complex multiplication required more time to process the data. Therefore, the high speed multiplication unit for floating point numbers is highly recommended to speed up such complex floating point multiplication. Some research

engineers have already started their research on designing a high speed floating point multiplier recently [2-6]. The performance in terms of processing speed of floating point data calculation is the main performance metric in today's VLSI or ULSI system design. The multiplier consumes large chip area density, high complexity, and time consuming process because the output data size is twice larger than input data size. Designing a high speed floating point multiplier in Field Programmable Grid Array (FPGA) [7] is still remains a major challenge. This inspired the idea to model a high speed floating point multiplier for VHDL designer's library. VHDL is one of the suitable languages to be used for complex system modelling [8-10].

There are many types of design models have been introduced to perform only for multiplication, and every multiplier design models have different algorithms which will give the same operation but different performance in terms of calculating speed and also resource consumption. Different type of multiplier algorithms is studied in this research. The suitable multiplication algorithm that is suitable for high speed 24-bit multiplication is identified. A new 32-bit single precision high speed floating point multiplier will be modelled in VHDL using the latest Engineering Design Automation (EDA) software, Altera Quartus II.

The main objective of this research is to model a 32-bit single precision floating point multiplier using VHDL. Since the floating point multiplier is not available in the VHDL designer's library, a new multiplication algorithm will also be created as a new module in the library. The Booth multiplication algorithm is proposed as a model for designing 24-bit multiplier. Other components including signer, exponent adder, and normalizer will also be modelled in VHDL to build a complete 32-bit floating point multiplier. Extensive simulation will be carried out on the multiplier modelled for functional verification. A high speed floating point multiplier is sampled by selecting the valid input data to produce the valid output results. Assumptions have been made to ignore the invalid data like overflow or underflow results produced by multiplying with infinite value or multiplying by zero respectively in this performance verification.

The timing performance, maximum operation frequency and resource consumption of 32-bit floating point multiplier are determined. Three Altera FPGA Arria II GX chip is used to evaluate the performance of the 32-bit floating point multiplier. The performance and resource usage comparison between newly proposed Booth multiplier and Wallace Tree multiplier are carried out.

2 Floating Point Multiplication Algorithms

The reason to have real numbers or fractional numbers is to obtain the result with better accuracy and precision. The binary representation is used to convert the real number into binary form that mostly supported by the machine. Such complex calculation required a huge amount of data and complex hardware to process desired output.

In the early stage, fixed point representation was the easiest method to convert the real number to binary because fixed-point adheres to the same basic arithmetic principles as integers. However, fixed point representation has limited range of values and once exceeding the range limit can cause data overflow. The floating point representation has better precision and support a much wider range of values compared to the fixed point representation. The size of the floating point representation that can be stored is either 32-bit (single-precision) or 64-bit (double-precision) defined by IEEE 754 Standard [11]. The IEEE Standard 754 single precision floating point format is widely implemented in digital systems uses 32 bits and 64 bits floating point number representation. In general, numbers are represented approximately to a fixed number of significant digits and scaled using an exponent. The base for the scaling is 2 for binary. For 32-bit floating point number representation, a floating point number in scientific notation as well as the IEEE 754 format are presented in Fig.1.

$$-1^{\text{Sign}} \times 1.\text{Mantissa} \times 2^{\text{Exponent-127}} \qquad (1)$$

Sign (1 bit)	Biased Exponent (8 bits)	Mantissa (23-bits)

Fig.1: IEEE 754 single precision (32-bit) floating point format

A decimal number needs to convert to binary number first follow by converting it to IEEE 754 single precision (32-bit) floating point format. The decimal point of the binary fractional number is moved to either left or right, so that only a single binary digit "1" is placed to the left of the binary decimal point. The exponent is used to record the adjustment of the decimal point. Next, the bias value, which is 127 is added to the exponent and then convert it into 8-bit binary. The "1." is omitted

from the mantissa. Once sign, biased exponent and mantissa are determined, 32-bit floating point representation is finally formed. The following example shows the conversion from decimal number to IEEE 754 32-bit floating point representation (Convert 12.0625_{10} to 32-bit floating point).

$12_{10} = 0000\ 1100_2$
$0.0625 \times 2 = 0.125$
$0.125 \times 2 = 0.25$
$0.25 \times 2 = 0.5$
$0.5 \times 2 = 1$
$12.0625_{10} = 000001100.0001_2 = 1.1000001 \times 2^{3}$

(+)	$3+127=130_{10}$ $=10000010_2$	1.1000001
0	10000010	10000010000000000000000

Multiplication is the mathematical operation of scaling one number by another. The common method to calculate the multiplication is using manual multiplication (Fig.2), which is multiply the multiplicand by each digit of the multiplier and then adds up all the properly shifted partial products. This method also applies in binary multiplication, a simple shift and add algorithm in base 2.

```
                                a3 a2 a1 a0    Multiplicand
X                               b3 b2 b1 b0    Multiplier
-------------------------------------------------------------
                          a3b0  a2b0  a1b0  a0b0    Partial Products
                    a3b1  a2b1  a1b1  a0b1
              a3b2  a2b2  a1b2  a0b2
        a3b3  a2b3  a1b3  a0b3
-------------------------------------------------------------
  P7    P6    P5    P4    P3    P2    P1    P0     Products
```

Fig.2: Manual multiplication method

This manual multiplication method can also be applied in circuit design by each part of multiplicand and multiplier is connected to AND gate to get partial products, and then adds up each partial product with adders. However, this design is impractical because almost all partial products are used and occupied more area density in a chip and also cause slow throughput. There are many multiplication methods that can reduce the number of partial products and speed up the process for better throughput in order to design the high speed multiplier. Wallace tree and Booth multiplier are widely used for implementing fast and efficient multiplier.

2.1 Wallace Tree Multiplier

Australian computer scientist Chris Wallace introduced the multiplication algorithm to implement the efficient circuit that multiplies two numbers, named Wallace tree multiplier in 1964 [12]. The partial products are arranged to form the Wallace Tree shown in Fig.3, and then compress the partial products using either a full adder (3:2 compressor) or half adder (2:2 compressor), depending on how many partial products are present in the same column.

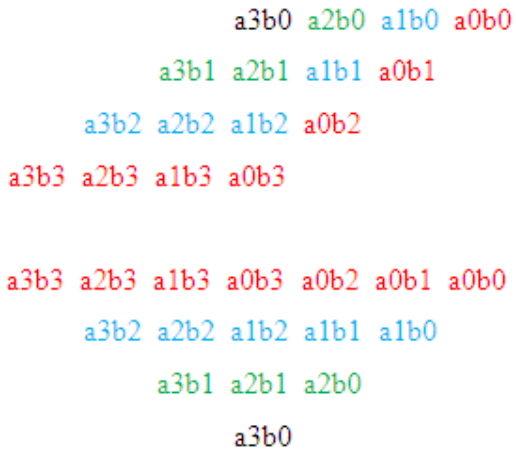


Fig.2: Partial products arrangement in the form of Wallace tree structure

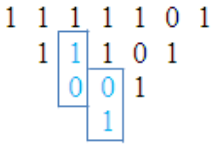
If there are three partial products within the same column, full adder is used to compress them and the output will be a sum output in the same column and a carry output that will be carried forward to the next higher column. If there are two partial products of the same column, half adder is used and the outputs are a sum in the same column and the carry output that will be carried forward to the next higher column. If there is just one partial product left, it will take as part of the result. The processes are repeated until only remain one partial product left in all columns to complete the Wallace Tree multiplication. The following example had shown the multiplication of 15×13.

$15_{10} = 1111_2$
$13_{10} = 1101_2$

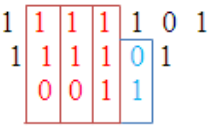
```
        1111   (15)
X       1101   (13)
-------------------
        1111
       0000
      1111
     1111
```

Half Adder (HA)
Full Adder (FA)

First stage:



Second stage:



Final stage:

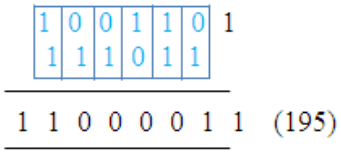


Fig.4 described the structure of Wallace tree multiplier that works similar to the example mentioned in previous part. Wallace tree multiplier speeds up the calculation by reducing the number of partial products. However, the disadvantage of the Wallace Tree multiplication is the digital circuit become more complicated as the number of bits extends. The consequence is the hardware wiring become more difficult to route, and large number of adders contribute longer delay to generate the final results thus slow down the processing speed. And that is the reason why Wallace Tree multiplication is not suitable to implement the high speed 24-bit multiplier.

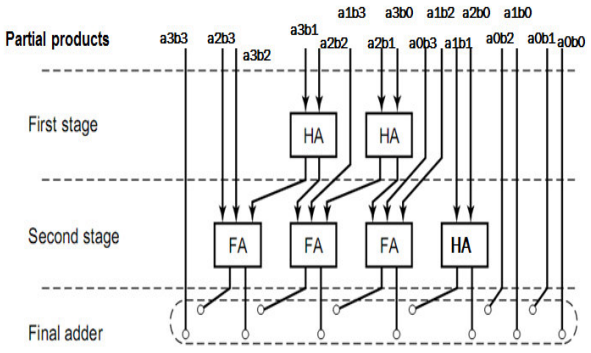


Fig.4: Partial products reduction in Wallace tree multiplier

2.2 Booth Radix-4 Multiplier

The Booth multiplication algorithm was developed by a British electrical engineer, physicist and computer scientist Andrew Booth in 1950 [13]. Booth formalized this observation and applied it to binary multiplication, where he started the first version known as radix-2. At the beginning, the first pair is created by appending least significant bit or Least Significant Bit (LSB) of the multiplier and a new bit '0' is added next to its right before examining each pair of bits. Then, Booth devises a simple rule for each step to complete multiplication:

- Add the multiplicand if the pair is 01,
- Subtract the multiplicand if the pair is 10,
- Do nothing if the pair is 00 or 11.

Once done, both partial product and multiplier are shifted one place to the right to allow the next pair of bits to be examined. Table 1 shows the radix-2 algorithm scheme, where A is multiplier, B is multiplicand and i represents bit position (e.g. i=0 indicates LSB). This process is repeated many times depending on the number of bits in the multiplier to complete the multiplication.

Table 1: Booth Algorithm Scheme (Radix-2)

A(i)	A(i-1)	B
0	0	+0
0	1	+B
1	0	-B
1	1	+0

For instance, mantissas multiplication required 24 bits for both inputs, and thus the examine pairs process in booth algorithm is repeated 24 times to complete mantissas multiplication. However, the radix-2 version takes too long to complete 24-bit mantissas multiplication process. Fortunately, the performance of Booth multiplier is improved in the enhanced version Booth radix-4. Instead of examining two least significant bits per step in radix-2, Booth radix-4 examines three least significant bits at a time in one step and performs actions before double right shifting. Initially, the last two bits of the multiplier are appended and a new bit '0' is added on the right of the LSB for Booth radix-4. The following actions are carried out to complete the computation.

- If 001 or 010, add the multiplicand only once.
- If 011, add the multiplicand twice.
- If 100, subtract the multiplicand twice.
- If 101 or 110, subtract the multiplicand only once.
- If 000 or 111, do nothing.

Both partial product and multiplier are then shifted two places to the right, allow the next three bits to be examined. This process is repeated depending on the number of bits in the multiplier to complete the Booth radix-4 multiplication. Table 2 shows the Booth radix-4 algorithm scheme. Notice that Booth radix-4 is able to reduce the number of

steps by half compared to radix-2 and higher processing speed can be achieved.

Table 2: Booth radix-4 Algorithm Scheme

A(i+1)	A(i)	A(i-1)	Action
0	0	0	Do nothing
0	0	1	A+B
0	1	0	A+B
0	1	1	A+2B
1	0	0	A-2B
1	0	1	A-B
1	1	0	A-B
1	1	1	Do nothing

Table 3 shows the computation that using Booth radix-4 multiplication.
1111 (15) x 1101 (13) = 11000011 (195)
Assuming 15 is multiplier, and 13 is multiplicand.
With Booth radix-4 multiplication, the number of partial products is greatly reduced, makes this hardware simple to implement and performs faster in data processing compared to Wallace Tree. In this research, a 24-bit Booth radix-4 multiplier is proposed in order to achieve high speed mantissas multiplication for high speed floating point multiplier. Booth radix-4 multiplier takes about 12 clock cycles to complete the 24-bit mantissas multiplication.

Table 3: Multiplication of 15 x 13 = 195 using Booth radix-4 algorithm

Step	8-bit + Temporary Register	Action	Procedure
1	0000 1111 0	Subtract 1101	0000 – 1101 = 0011
			0011
		Shift right twice	1100 11
2	1100 0111 1	Do nothing.	
		Shift right twice	1111 0011
3	0011 0000 1	Add 1101	1111 + 1101 = 1100
			1100 0011
		Shift right twice	0011 0000 11
4	0000 1100 0	Do nothing.	
		Shift right twice	0000 1100 0011
5	1100 0011 0	Execute Result	1100 0011 (195)

3 32-bit Booth Radix-4 Multiplier

The single-precision (32-bit) floating-point multiplier performs multiplication of two inputs which are floating-point numbers. At the beginning, both inputs must be converted from decimal number into floating point representation based from IEEE 754 standard before doing the multiplication. Once the floating point multiplication is complete, the output which is in IEEE 754 floating point representation will convert back to decimal number. A multiplication of two floating-point numbers is done in the following 5 steps:

Step 1: Multiplication of mantissas
Step 2: Normalization
Step 3: Addition of the exponents
Step 4: Calculation of the sign
Step 5: Composition of all results

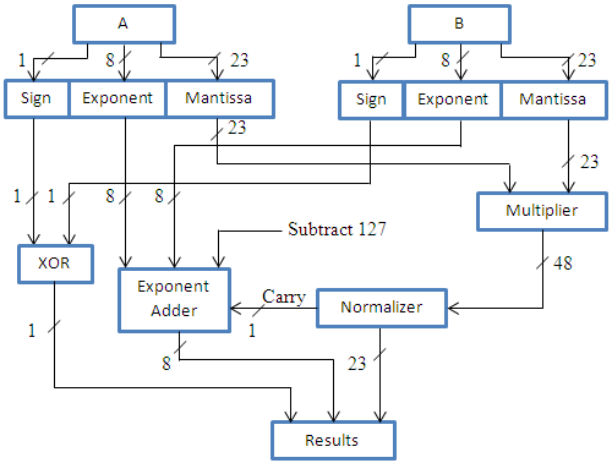


Fig.5: 32-bit floating point multiplier data flow

Fig.5 described the 32-bit floating point multiplier data process flow. Each input is split into three modules (sign, exponent, and mantissa) so that can be easily to route into corresponding components. Signs from input A and B are connected directly to XOR gate to generate the final sign result, either '0' indicates positive sign or '1' indicates the negative sign. Meanwhile, exponents and mantissas from input A and B are connected to exponent adder and multiplier respectively. The 48-bit output from multiplier must pass through to the normalizer to perform rounding to nearest 23-bit of mantissa. In exponent adder, both exponents from A and B are added before subtract to bias value which is 127. The carry signal from normalizer is also connected to exponent adder to adjust the exponent value, which will be the final 8-bit exponent result. All the output from signer (1-bit), exponent adder (8-bit), and normalizer (23-bit) are then combined to form 32-bit floating point multiplication product as the final results.

The last 23 bits of mantissa in 32-bit floating-point number is given by two operands for

multiplication. The explicit '1' added as the leading bit of both mantissas to fit into 24-bit multiplier unit. The 24-bit data multiply with another 24-bit data result twice the size of the operands which is a 48-bit data. Booth radix-4 multiplier is used as 24-bit mantissas multiplier. Later, only 23 bits are to be extracted in order to follow IEEE 754 standard rules. This can be done by normalization.

The extraction of 23-bit out of 48-bit output after multiplication, which is the final result for mantissa have come with 2 conditions. It may also involve some adjustment of the resultant exponent, depending on Most Significant Bit (MSB) of the 48-bit multiply product. If the MSB is '1', bit-25 to bit-47 will be selected as final 23-bit mantissa with rounding to nearest by adding bit-24, and add '1' to exponent. If MSB is '0', then bit-24 to bit-46 will be selected with rounding to nearest by adding bit-23 without adding carry to exponent.

The 8-bit exponent values from two operands are added to generate a sum of 9-bit result. The incoming values are biased, a constant value of 127 must be subtracted from the result. In addition, the carry signal from the normalizer is added to this exponent for adjustment. Only 8-bit exponent values forward to the final output, which will be the 8-bit exponent in IEEE 754 32-bit floating point. The formula for the sum of exponents is as follow.

$$\text{Sum of Exponent} = \text{Exponent A} + \text{Exponent B} - 127 + \text{Carry from Normalizer} \quad (2)$$

For special case reason, the MSB (bit-9) denotes that an exponent is either overflow or underflow has occurred. The MSB of the exponent is '1' means the exponent value is overflowed (infinite value), and '0' means this is under flowed (nearly zero value).

The left most significant bit in 32-bit floating-point format stores the sign of the number after multiplication, where '0' indicate positive sign (+) or '1' indicate negative sign (-). The result will generate a positive signed number when either positive signed numbers or both negative signed numbers are multiplied. If one input is a positive signed number, and other is a negative signed number, the result will generate a negative signed number after multiplies to two different signed input numbers. A simple method to determine the sign is using an exclusive-or gate (XOR gate). The XOR gate gives output logic '0' if both inputs are the same and logic '1' if both inputs are different.

4 Simulation Analyses

Altera Quartus II is the best EDA software to design digital logic circuit behaviour by schematic, or coding (VHDL or Verilog) on FPGA devices. VHDL is most commonly used to describe a logic circuit by writing text model. The text model is then compiled and synthesized into the gate level logic circuit. VHDL is also being used for writing simulation model to test logic circuit functionality, called test bench. The simulation model contains a number of random input vectors to generate the expected output during simulation.

TimeQuest Timing Analyzer is used as a timing analysis tool that validates the timing performance for digital logic design using industry standard constraint, analysis, and reporting methodology. TimeQuest analyses the clock constraint and I/O delay setting on the logic circuit design, and then generate the timing report once compilation is complete. The timing report will show the maximum frequency that the logic circuit can be supported on both 1200mV 0°C model and 85°C model. The slack (the margin which time is required to achieve at the longest and most critical path) is also determined. Slack is calculated by using the equation below.

$$\text{Slack} = \text{Time when data required} - \text{Time when data arrived} \quad (3)$$

The slack must be in positive value or the data must arrive before data required time, otherwise the clock setting does not meet the timing requirement and violates data setup and hold time. To achieve positive slack, the clock speed should be reduced to increase the time period. Clock cycle or time period can be set using Synopsys Design Constraints (SDC) file. TimeQuest generated a path summary report that shows the longest critical path, data arrival time and data required time.

4.1 24-bit Booth Radix-4 Mantissa Multiplier

The Booth radix-4 algorithm is modeled using VHDL code to perform Booth multiplication. This algorithm is triggered by the positive edge triggered clock. This multiplication completed in 12 clock cycles plus 1 clock cycle for result adjustment. The enable signal (EN) is used to activate the normalizer once the multiplication is completed. Fig.6 shows the module of 24-bit Booth radix-4 mantissa multiplier.

A series of input data are shown in Table 4 are used for functional verification of the 24-bit mantissa multiplier. Table 4 provides all the input data and the corresponding output generated by multiplier. As shown in Fig.7, the 24-bit Booth radix-4 multiplier generated the expected output shown in Table 4. This shows that 24-bit Booth radix-4 multiplier is modeled successfully.

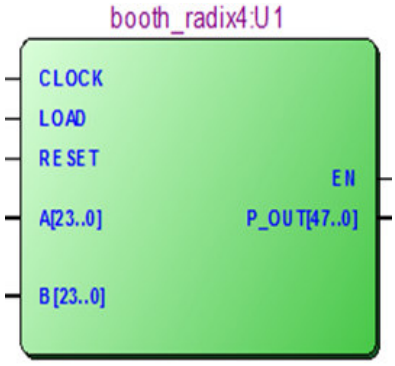


Fig.6: 24-bit Booth radix-4 mantissa multiplier module

Table 4: Test data used for 24-bit Booth radix-4 mantissa multiplier

Inputs (A x B)	Expected output
1 x 2	2
3 x 4	12
5 x 6	30
7 x 8	56
9 x 10	90
8785920 x 9437184	82914343649280
7991296 x 7340032	58656368361472

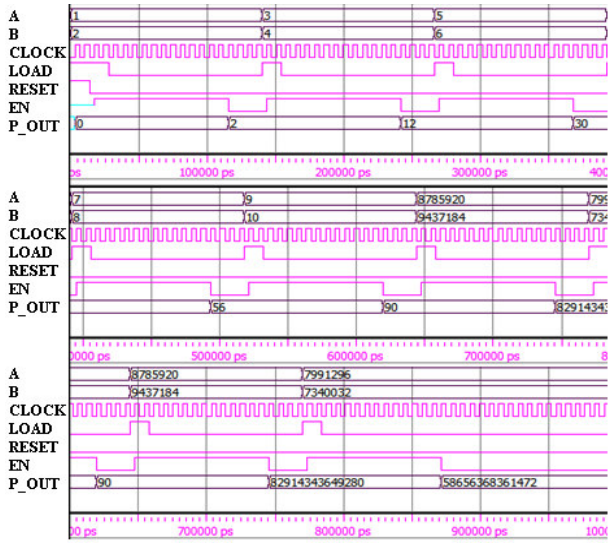


Fig.7: Output waveform of 24-bit Booth radix-4 mantissa multiplier

4.2 Normalizer

The function of normalizer is to extract 48-bit mantissa that has been generated by 24-bit Booth radix-4 mantissa multiplier into 23-bit mantissa as part of the final result. If the MSB (bit-48) is '1', bit-25 to bit-47 will be selected as final 23-bit mantissa with rounding to nearest by adding bit-24, and add carry '1' to exponent. If MSB is '0', then bit-24 to bit-46 will be selected with rounding to nearest by adding bit-23 without adding carry to exponent. All those conditions are written in VHDL using if-else condition. Fig.8 and Fig.9 show the RTL view and module of the normalizer.

Four tests below have been carried out to check the functionality of normalizer to extract 23-bit out from 48-bit, and also perform rounding to nearest algorithm. As a result, the output waveform in Fig.10 proved normalizer generates all the correct answers.

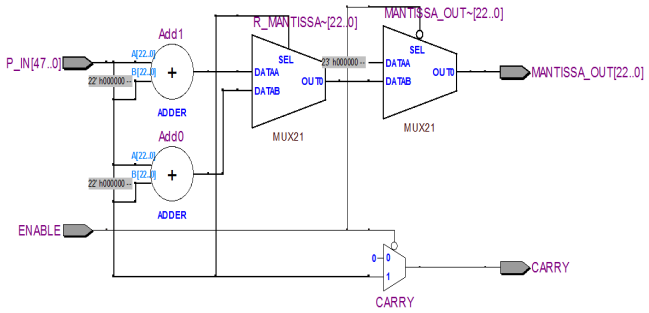


Fig.8: RTL view of the Normalizer

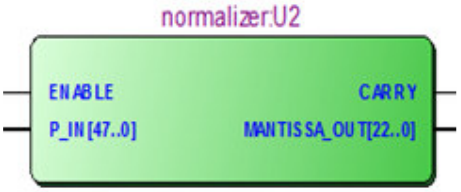


Fig.9: Normalizer module

Test 1:

0x345900000000 = 001101000101100100000000000000000000000000000000

MSB = 0,

23-bit Mantissa before rounding = 11010001011001000000000

23-bit Mantissa after rounding = 11010001011001000000000 + 0

= 11010001011001000000000 = 0x68B200

Test 2:

0xB45900000000 = 101101000101100100000000000000000000000000000000

MSB = 1,

23-bit Mantissa before rounding = 01101000101100100000000

23-bit Mantissa after rounding = 01101000101100100000000 + 0

= 01101000101100100000000 = 0x345900

Test 3:

0x345900C00000 = 00110100010110010000000011000000000000000000000000

MSB = 0,

23-bit Mantissa before rounding = 11010001011001000000001

23-bit Mantissa after rounding = 11010001011001000000001 + 1

= 11010001011001000000010 = 0x68B202

Test 4:

0xB45900000000 = 10110100010110010000000010000000000000000000000000

MSB = 1,

23-bit Mantissa before rounding = 01101000101100100000000

23-bit Mantissa after rounding = 01101000101100100000000 + 1

= 01101000101100100000001 = 0x345901

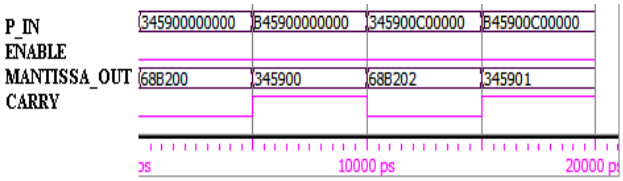


Fig.10: Output waveform of Normalizer

4.3 Exponent Adder

The operation of exponent adder is quite similar to the full adder except the constant biased value of 127 is included to be subtracted with the sum of input exponent A, B and the carry bit from normalizer. The 8-bit output SUM is the final result for the exponent part of 32-bit floating point multiplication result.

Fig.11 and Fig.12 show the RTL view and module of the exponent adder respectively. A set of input vectors is used in the VHDL test bench shown in Fig.13 to test the functionality of the exponent adder. Fig.14 shows the output waveform and the simulation result indicates that the exponent adder is working successfully.

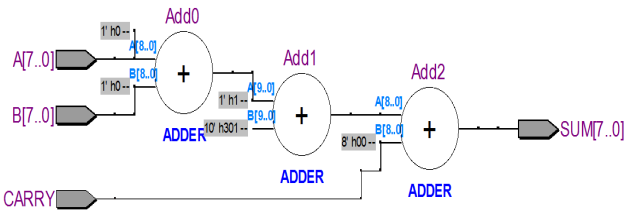


Fig.11: RTL view of Exponent Adder

Fig.12: Exponent Adder Module

```
A <= "00001100"; B <= "00001010"; CARRY <= '0';
wait for 5 ns;
CARRY <= '1'; wait for 5 ns;
A <= "01111100"; B <= "01100000"; CARRY <= '0';
wait for 5 ns;
CARRY <= '1'; wait for 5 ns;
A <= "10000001"; B <= "10000010"; CARRY <= '0';
wait for 5 ns;
CARRY <= '1'; wait for 5 ns;
A <= "10000110"; B <= "10000000"; CARRY <= '0';
wait for 5 ns;
CARRY <= '1'; wait for 5 ns;
A <= "10000010"; B <= "01111101"; CARRY <= '0';
wait for 5 ns;
CARRY <= '1'; wait for 5 ns;
```

Fig.13: VHDL test bench of Exponent Adder

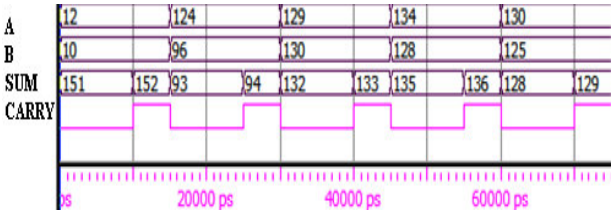


Fig.14: Output waveform of Exponent Adder

4.4 Signer

Signer determines the floating point multiplication results either positive number or negative number. Same signed numbers are multiplied will lead to a positive number, different signed numbers lead to a negative number. The solution is to use a simple logical operator XOR gate. The XOR gate output waveform is shown in Fig.15.

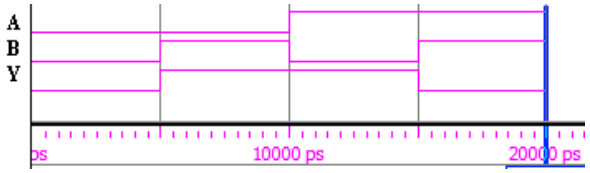


Fig.15: Output waveform of Signer or XOR gate

4.5 32-bit Both Radix-4 Multiplier

All four components are combined together to build 32-bit floating point multiplier. Fig.16 showed the RTL view of 32-bit floating point multiplier.

The following tests with desired inputs (A and B) and expected output R have been carried out to check the functionality of the 32-bit floating point multiplier. Special cases such as invalid data like overflow or underflow results produced by multiplying with infinite value or zero are ignored in this test.

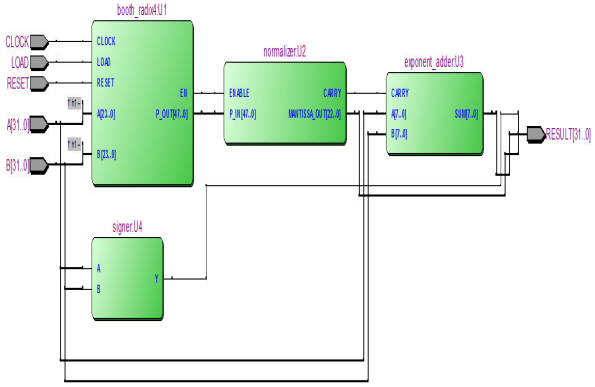


Fig.16: RTL view – Complete 32-bit floating point multiplier

Test 1:
134.0625 x -2.25 = -301.640625
A = 134.0625 = 1.00001100001 x 2^{7}
A = 0 10000110 00001100001000000000000
= 0x43061000
B = -2.25 = -1.001 x 2^{1}
B = 1 10000000 00100000000000000000000
= 0xC0100000
R = A x B = -301.640625
= -1.00101101101001 x 2^{8}
R = A x B
= 1 10000111 00101101101001000000000
= 0xC396D200

Test 2:
-14.5 x -0.375 = 5.4375
A = -14.5 = -1.1101 x 2^{3}
A = 1 10000010 11010000000000000000000
= 0xC1680000
B = -0.375 = -1.1 x 2^{-2}
B = 1 01111101 10000000000000000000000
= 0xBEC00000
R = A x B = 5.4375 = 1.010111 x 2^{2}
R = A x B
= 0 10000001 01011100000000000000000
= 0x40AE0000

Test 3:
7.5 x 15.5 = 116.25
A = 7.5 = 1.111 x 2^{2}
A = 0 10000001 11100000000000000000000
= 0x40F00000
B = 15.5 = 1.1111 x 2^{3}
B = 0 10000010 11110000000000000000000
= 0x41780000
R = A x B = 116.25 = 1.11010001 x 2^{6}
R = A x B
= 0 10000101 11010001000000000000000
= 0x42E88000

As a result, all the outputs RESULT from the waveform in Fig.17 are matched with outputs R computed above after 13 clock cycles. This is shown that the 32-bit floating point multiplier is modeled successfully.

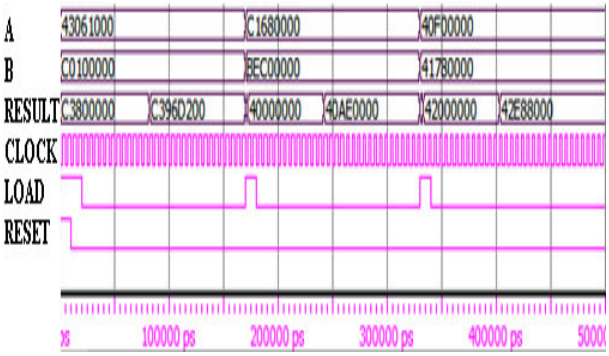


Fig.17: Output waveform of complete 32-bit floating point multiplier

4.6 TimeQuest Timing Analyzer

The analysis is targeted on Arria II GX EP2AGX45CU17I3 device. The minimum CLOCK period is able to be constrained until 5 ns under TimeQuest Timing Analyzer. The path summary report for 32-bit floating point multiplier using this low power mid-range chip is generated in Table 5. The critical path is located from pa[27] to P_OUT[35] inside the Booth radix-4 multiplier. The data arrival time is 7.276 ns and the data required time is 7.425 ns. The slack is the difference between data required time and data arrival time, which is 0.149 ns. The maximum frequency of Arria II GX EP2AGX45CU17I3 device is 206.14 MHz as shown in Fig.18.

Table 5: Path summary for 32-bit floating point multiplier using Arria II GX series EP2AGX45CU17I3

From Node	booth_radix4:U1|pa[27]
To Node	booth_radix4:U1|P_OUT[35]
Launch Clock	CLOCK
Latch Clock	CLOCK
Data Arrival Time	7.276
Data Required Time	7.425
Slack	0.149

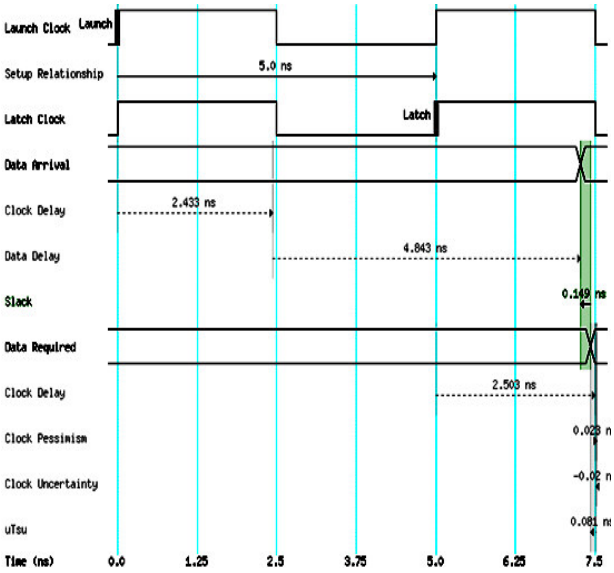


Fig.18: TimeQuest timing waveform of 32-bit floating point multiplier using Arria II GX Series EP2AGX45CU17I3 device

4.7 Booth Radix-4 and Wallace Tree Performance Comparison

Some research engineers claimed that Wallace Tree multiplier performs faster than Booth multiplier. The Wallace Tree multiplier modeled in [14] is used as 24-bit Wallace Tree mantissas multiplier. Their observation may be true but the question is how the Wallace Tree multiplier improves its performance compared to Booth multiplier. This experiment will prove that Booth multiplier still the best option to build a high speed multiplier rather than Wallace Tree multiplier.

The performance of the 32-bit Wallace Tree floating point multiplier is compared with 32-bit Booth radix-4 floating point multiplier in this test. The CLOCK signal is constrained by 5 ns for both types of multiplication and using Arria II GX (EP2AGX45CU17I3) FPGA is used to conduct this performance test. Table 6 summarizes the performance between Wallace Tree and Booth Radix-4 multiplier on the Arria II GX device.

Fig.19 shows the difference between 32-bit Wallace Tree and Booth radix-4 floating point multiplier by comparing the number of Adaptive Look-up Tables (ALUTs) and logic register used in Arria II GX FPGA chip. As shown in Fig.19, the resource consumption of Booth radix-4 multiplier is 88.8% less than the Wallace Tree multiplier. Wallace Tree multiplier consumes more resource usage on Arria II GX FPGA despite its performance is only slightly faster than Booth radix-4 multiplier. The 32-bit floating point multiplier with Wallace Tree multiplier used extra 88.8% of the total combinational ALUTs and 94.5% of the total logic registers to improve only 1% (2.76 MHz) faster than the Booth radix-4 multiplier. In contrast, the 32-bit floating point multiplier with Booth radix-4 multiplier consumes less resource usage and still able to perform with maximum frequency almost as fast as Wallace Tree multiplier.

Table 6: Performance comparison between the 32-bit Wallace Tree and Booth radix-4 floating point multiplier on Arria II GX (EP2AGX45CU17I3)

24-bit Mantissas Multiplier Type	Wallace Tree	Booth Radix-4	Difference
Chip Family	Arria II GX EP2AGX45CU17I3		
Device			
Combinational ALUTs	2122 / 36100 (6%)	238 / 36100 (< 1%)	88.8%
Logic Registers	1939 / 36100 (5%)	106/ 36100 (< 1%)	94.5%
Total Pins	98 / 176 (56%)	99/ 176 (56%)	
Minimum Clock Period	5 ns		
Maximum Frequency	208.9 MHz	206.14 MHz	1%

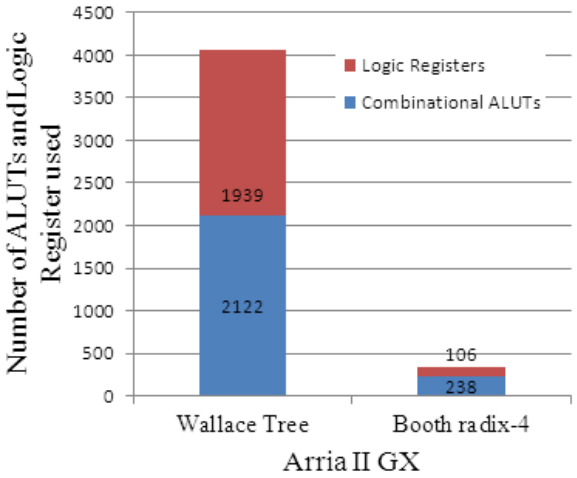


Fig.19: Resource usage comparison graph between 32-bit Wallace Tree and Booth radix-4 floating point multiplier on Arria II GX FPGA device

5 Conclusion

The high speed single precision 32-bit Booth radix-4 floating point multiplier has been modeled using

VHDL. It consists of 4 modules, i.e. Booth Radix-4 mantissa multiplier, normalizer, exponent adder and the signer. Booth radix-4 multiplication is one of the suitable algorithm to be used to design the high speed 24-bit mantissas multiplier because this algorithm is much simpler than the complex Wallace Tree multiplier, thus less gate delay and able to perform such complex multiplication faster. In addition, Booth radix-4 performance is doubled compared to Booth radix-2 that allows high speed multiplication can be achieved. The 32-bit floating point Wallace Tree multiplier operates up to 208.9 MHz with same constrained clock period of 5 ns on Arria II GX FPGA. However, Wallace Tree multiplier consumes more than 90% extra resources compare to Booth radix-4 multiplier to gain only 1% performance improvement. The 32-bit floating point Booth radix-4 multiplier design is a better option because it consumes much lesser resource on FPGA and supports the maximum frequency of 206.14 MHz.

References:

[1] A. Malinowski, Hao Yu, Comparison of Embedded System Design for Industrial Applications, *IEEE Transactions on Industrial Informatics*, Vol.7, 2011, pp. 244-254.

[2] M. Marius, S. Marius, O. Onisifor, A FPGA Floating Point Interpolator, *Advances in Intelligent Systems and Computing*, Vol.195, 2013, pp. 331-336.

[3] B. Hickmann, A Parallel IEEE P754 Decimal Floating-Point Multiplier, *IEEE 25th International Conference on Computer Design*, 2007, pp. 296 -303.

[4] M. Kumar Jaiswal, Area-Efficient FPGA Implementation of Quadruple Precision Floating Point Multiplier, *IEEE International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, 2012, pp. 376-382.

[5] P. Wu, B.B. Wang, C.H. Ji, Design and Realization of Short Range Defense Radar Target Tracking System Based on DSP/FPGA, *WSEAS Transactions on Systems*, Vol.10, 2011, pp. 376-386.

[6] L.F. Rahman, Md. Mamun, M.S. Amin, VHDL Environment for Pipeline Floating Point Arithmetic Logic Unit Design and Simulation, *Journal of Applied Sciences Research*, Vol.8, 2012, pp. 611-619.

[7] X. Yang, J. Zhao, J. Jiang, An improved dc capacitor voltage detection technology and its FPGA implementation in the CHB-based STATCOM, *WSEAS Transactions on Systems*, Vol. 9, 2010, pp. 20-30.

[8] Z.Y. Lam, W.L. Pang, C.P. Ooi, S.K. Wong, K.Y. Chan, VHDL Modelling of Reed Solomon Decoder, *Research Journal of Applied Sciences*, Vol. 4, 2012, pp. 5193-5200.

[9] W.L. Pang, M. B. I. Reaz, M.I. Ibrahim, L.C. Low, F.M. Yasin, R.A. Rahim, Handwritten character recognition using fuzzy wavelet: a VHDL approach, *WSEAS Transactions on Systems*, Vol. 5, 2006, pp. 1641-1647.

[10] W. L. Pang, K. W. Chew, F. Choong, E.S. Teoh, VHDL Modeling of the IEEE802.11b DCF MAC, *6th WSEAS International Conference on Instrumentation, Measurement, Circuits & Systems*, 2007, pp. 28-33.

[11] IEEE 754-2008, IEEE Standard for Floating-Point Arithmetic, IEEE Computer Society, Aug 2008.

[12] C. Wallace, A Suggestion for a Fast Multiplier, *IEEE Transactions on Electronic Computers*, Vol.13, 1964, pp. 14-17.

[13] A. Booth, A Signed Binary Multiplication Technique, *Journal of Mechanics and Applied Mathematics*, Vol. 4, 1951, pp. 236-240.

[14] VHDL Component: Wallace Tree Multiplier (Generic), *Verilog HDL Discussion Forum*, http://www.openhdl.com/vhdl/655-vhdl-component-wallace-tree-multiplier-generic.html.