



AN FPGA BASED 64-BIT IEEE – 754 DOUBLE PRECISION FLOATING POINT ADDER/SUBTRACTOR AND MULTIPLIER USING VHDL

M. Aravind Kumar*, K. P. Mani*, P. Sai Teja**, G. Sowjanya**
& V. L. S. Prakash Raju**

* Associate Professor, Department of ECE, Grandhi Varalakshmi Venkata Rao Institute of Technology, West Godavari, Bhimavaram, Andhra Pradesh

** B.Tech Scholar, Department of ECE, Grandhi Varalakshmi Venkata Rao Institute of Technology, West Godavari, Bhimavaram, Andhra Pradesh

Abstract:

Floating point arithmetic is widely used in many areas, especially scientific computation and signal processing. For many signal processing, and graphics applications, it is acceptable to trade off some accuracy (in the least significant bit positions) for faster and better implementations. In this paper we describe an implementation of high speed IEEE 754 double precision floating point multiplier targeted for Xilinx Virtex-6 FPGA. VHDL is used to implement the design. Floating Point (FP) addition, subtraction and multiplication are more used in large set of scientific and signal processing computation. A high speed floating point double precision adder/subtractor and multiplier are implemented on a Virtex -6 FPGA. In addition, the proposed design is compliant with IEEE-754 format and handles overflow, under flow, rounding and various exception conditions.

Key Words: Double Precision, Floating Point, Adder/Subtractor, Multiplier, FPGA, IEEE 754 & Virtex-6.

Introduction:

Programming languages support numbers with fraction Called floating-point numbers. Scientific notation: $\pm d.f_1f_2f_3f_4 \dots \times 10^{\pm e_1e_2e_3}$

Examples:

3.14159265... (π)

2.71828... (e)

Floating Point Arithmetic is widely used in many areas, especially scientific computation and signal processing. The advantage of floating-point representation over fixed-point and integer representation is that it can support a much wider range of values. Floating point numbers are one possible way of representing real numbers in binary format; the IEEE 754 standard presents two different floating point formats, Binary interchange format and Decimal interchange format. Defined by IEEE Std 754-1985, floating points have 2 representations.

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

Two representations

- Single precision (32-bit)

1-bit sign + 8-bit exponent + 23-bit fraction

IEEE 754 uses biased representation for the exponent

- Value of exponent = $\text{val}(E) = E - \text{Bias}$.
- exponent field is 8 bits for single precision E can be in the range 0 to 255

- Bias = 127 (half of 254), $\text{val}(E) = E - 127$
- $\text{val}(E=1) = -126$, $\text{val}(E=127) = 0$, $\text{val}(E=254) = 127$

Decimal value of the Single Precision float

- Sign = 1 is negative
- Exponent =
 $(01111100)_2 = 124$, $E - \text{bias} = 124 - 127 = -3$
- Significand =
 $(1.0100 \dots 0)_2 = 1 + 2^{-2} = 1.25$ (1. is implicit).
- Value in decimal = $-1.25 \times 2^{-3} = -0.15625$.

IEEE-754 Encoding:

Single-Precision	Exponent = 8	Fraction = 23	Value
Normalized number	1 to 254	Any thing	$\pm (1.F)_2 \times 2^{E-127}$
Denormalized number	0	Non zero	$\pm (0.F)_2 \times 2^{-126}$
Zero	0	0	± 0
Infinity	255	0	$\pm \infty$
NAN	255	Non zero	NaN

- Double precision (64-bit)
 1-bit sign + 11-bit exponent + 52-bit fraction.

For double precision, exponent field is 11 bits

- E can be in the range 0 to 2047
- Bias = 1023 (half of 2046), $\text{val}(E) = E - 1023$
- $\text{val}(E=1) = -1022$, $\text{val}(E=1023) = 0$, $\text{val}(E=2046) = 1023$

Sign bit (1 bit)	Exponent bit(11 bits)	Fraction/ Mantissa (52 bits)
---------------------	--------------------------	------------------------------------

What is the decimal value of this Double Precision float?

0 1 0 0 0 0 0 0 1 0 1 0 0 1 0 1 0 1 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0
0 0 0 0 0 0 0 0 0

Solution:

Value of exponent = $(10000000101)_2 - \text{Bias} = 1029 - 1023 = 6$

Value of double float = $(1.00101010 \dots 0)_2 \times 2^6$ (1. is implicit) = $(1001010.10 \dots 0)_2 = 74.5$.

IEEE-754 Encoding:

Double-Precision	Exponent = 11	Fraction = 52	Value
Normalized Number	1 to 2046	Anything	$\pm (1.F)_2 \times 2^{E-1023}$
Denormalized Number	0	Nonzero	$\pm (0.F)_2 \times 2^{-1022}$
Zero	0	0	± 0
Infinity	2047	0	$\pm \infty$
NaN(Not A Number)	2047	Nonzero	NaN

S is the Sign bit (0 is positive and 1 is negative)

- Representation is called sign and magnitude
- E is the Exponent field (signed)
- Very large numbers have large positive exponents
 - Very small close-to-zero numbers have negative exponent
- F is the Fraction field (fraction after binary point)
- More bits in fraction field improves the precision of FP numbers

Implementation of Floating Point Unit:

The two operands are already in double precision format.

All arithmetic operations have these three stages:

- **Pre-normalize:** the operands are transformed into formats that makes them easy and efficient to handle internally.
- **Arithmetic core:** the basic arithmetic operations are done here.
- **Post-normalize:** the result will be normalized if possible (leading bit before decimal point will be 1, if normalized) and then transformed into the format specified by the IEEE standard.

A. Addition and Subtraction:

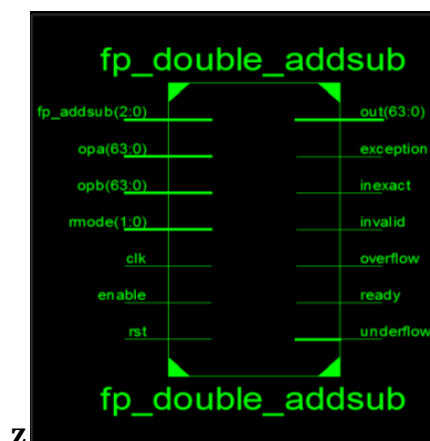
The conventional floating-point addition algorithm consists of five stages - exponent difference, pre-alignment, addition, normalization and rounding. Given floating-point numbers $X1 = (s1, e1, f1)$ and $X2 = (s2, e2, f2)$, the stages

For computing $X1 + X2$ are described as follows:

- Find exponent difference $d = e1 - e2$. If $e1 < e2$, swap position of mantissas. Set larger exponent as tentative exponent of result.
- Pre-align mantissas by shifting smaller mantissa right by d bits.
- Add or subtract mantissas to get tentative result for mantissa.
- Normalization. If there are leading-zeros in the tentative result, shift result left and decrement exponent by the number of leading zeros. If tentative result overflows, shift right and increment exponent by 1-bit.

Flow Chart for Addition/Subtraction:

Round mantissa result. If it overflows due to rounding, shift right and increment exponent by 1-bit. If the exponents differ by more than 53, the smaller number will be shifted right entirely out of the mantissa field, producing a zero mantissa. The sum will then equal the larger number. When adding numbers of opposite sign, cancellation may occur, resulting in a sum which is arbitrarily small, or even zero if the numbers are equal in magnitude. Normalization in this case may require shifting by the total number of bits in the mantissa, resulting in a loss of accuracy. Floating point subtraction is achieved simply by inverting the sign bit and performing addition of signed mantissas as outlined above.



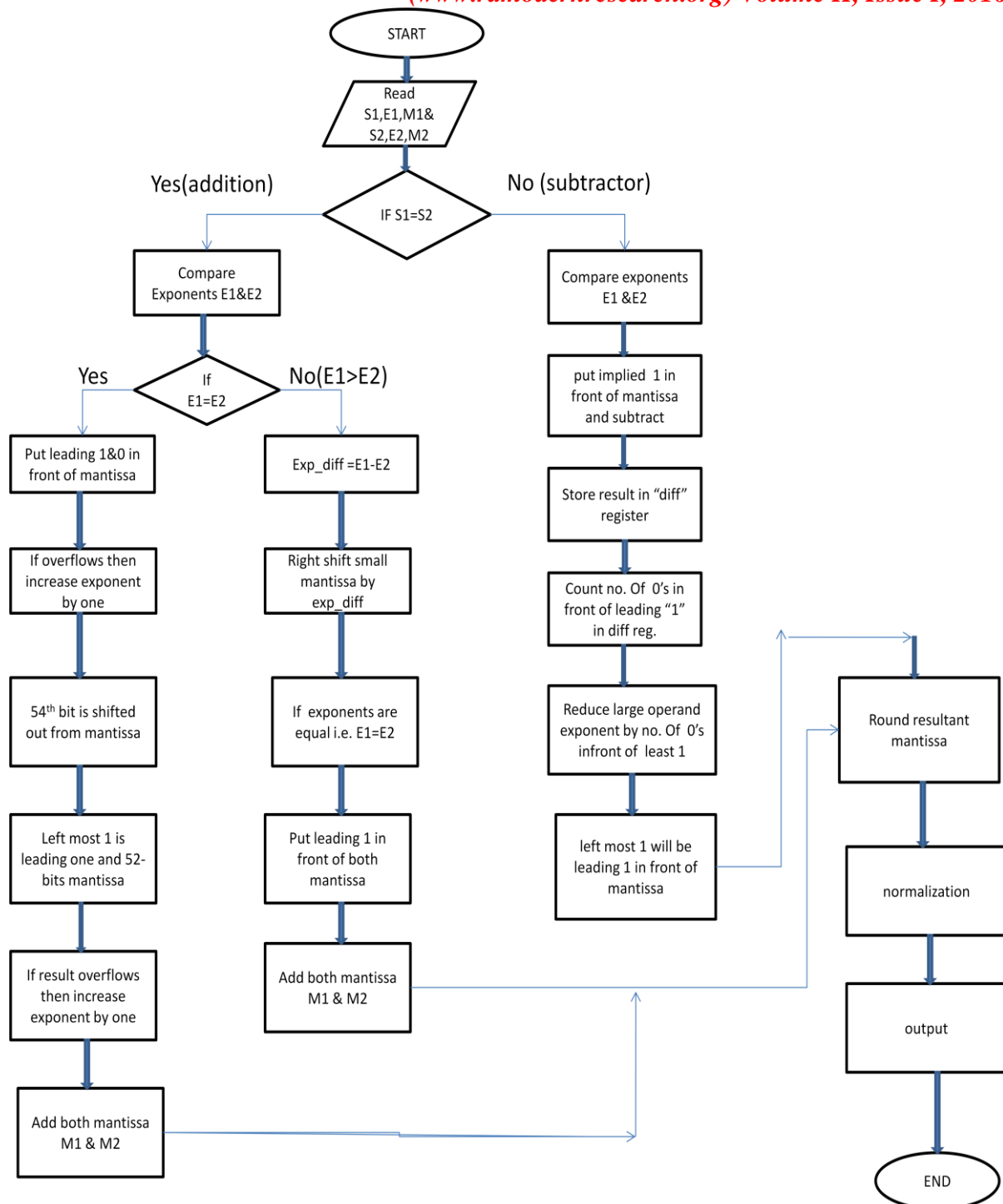


Figure 1: Black box view of double precision floating point adder / subtractor

Example for addition:

Consider adding: $(1.111)_2 \times 2^{-1} + (1.011)_2 \times 2^{-3}$

For simplicity, we assume 4 bits of precision (or 3 bits of fraction)

- exponents are not equal

Shift the significand of the lesser exponent right until its exponent matches the larger number

- $(1.011)_2 \times 2^{-3} = (0.1011)_2 \times 2^{-2} = (0.01011)_2 \times 2^{-1}$

Difference between the two exponent = $-1 - (-3) = 2$

So, shift right by 2 bits

Now, add the significant

$$\begin{array}{r} 1.111 \\ 0.01011 \\ \hline 10.00111 \end{array}$$

- So, $(1.111)_2 \times 2^{-1} + (1.011)_2 \times 2^{-3} = (10.00111)_2 \times 2^{-1}$
- However, result $(10.00111)_2 \times 2^{-1}$ is NOT normalized
- Normalize result: $(10.00111)_2 \times 2^{-1} = (1.000111)_2 \times 2^0$
 In this example, we have a carry
 So, shift right by 1 bit and increment the exponent
- Round the significand to fit in appropriate number of bits
 We assumed 4 bits of precision or 3 bits of fraction
- Round to nearest: $(1.000111)_2 \approx (1.001)_2$
 Renormalize if rounding generates a carry
- Detect overflow / underflow
 If exponent becomes too large (overflow) or too small (underflow).

Example for subtraction:

Consider: $((1.000)_2 \times 2^{-3}) - ((1.000)_2 \times 2^2)$

- Difference between the two exponents = $2 - (-3) = 5$
- Shift right by 5 bits: $(1.000)_2 \times 2^{-3} = (0.00001000)_2 \times 2^2$
- Convert subtraction into addition to 2's complement

$$\begin{array}{rcl} \text{Sign} & \rightarrow & + \quad 0.00001 \times 2^2 \\ & & - \quad 1.00000 \times 2^2 \end{array}$$

$$\begin{array}{r} 0 \quad 0.00001 \times 2^2 \\ 1 \quad 1.00000 \times 2^2 \\ \hline 1 \quad 1.00001 \times 2^2 \end{array}$$

Since result is negative, convert result from 2's complement to sign-magnitude

2's Complement = -0.11111×2^2

So, $(1.000)_2 \times 2^{-3} - (1.000)_2 \times 2^2 = -0.11111 \times 2^2$

- Normalize result: $-(0.11111)_2 \times 2^2 = -1.1111 \times 2^1$
- Round the significand to fit in appropriate number of bits
- Round to nearest: $(1.1111)_2 \approx (10.000)_2$
- Renormalize: rounding generated a carry

$$\begin{array}{rcl} & & 1.111|1 \\ & & 1 \\ (-1.1111)_2 \times 2^1 & \approx & (-10.000)_2 \times 2^1 = (-1.000)_2 \times 2^2 \end{array}$$

10.000

B. Multiplication:

In discussing floating-point multiplication, by complies with the IEEE 754 Standard, the two mantissas are to be multiplied, and the two exponents are to be added. The sign logic is a simple XOR. In order to perform floating point multiplication, a simple algorithm is realized:

- Add the exponents and subtract 1023 (bias).
- Multiply the mantissas and determine the sign of the result.
- Normalize the resulting value, if necessary.

Overflow occurs when the sum of the exponents exceeds 1023, the largest value which is defined in bias -1023 exponent representation. When this occurs, the exponent is set to 1024 ($E = 2047$) and the mantissa is set to zero indicating + or - infinity. Underflow occurs when the sum of the exponents is more negative than -1022, the most negative value which is defined in bias -1022 exponent representation.

Flow Chart of multiplication:

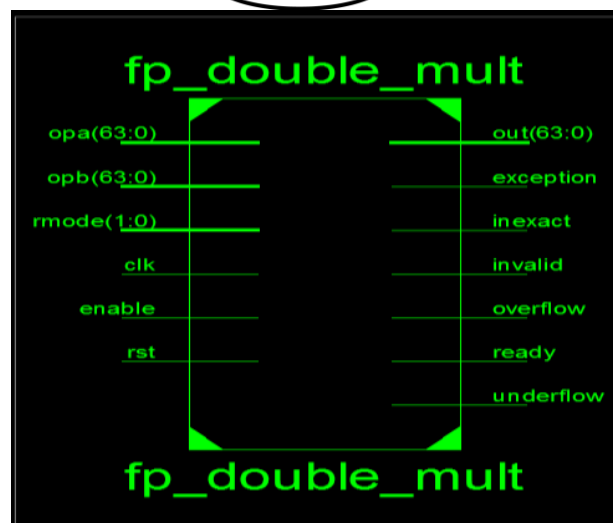
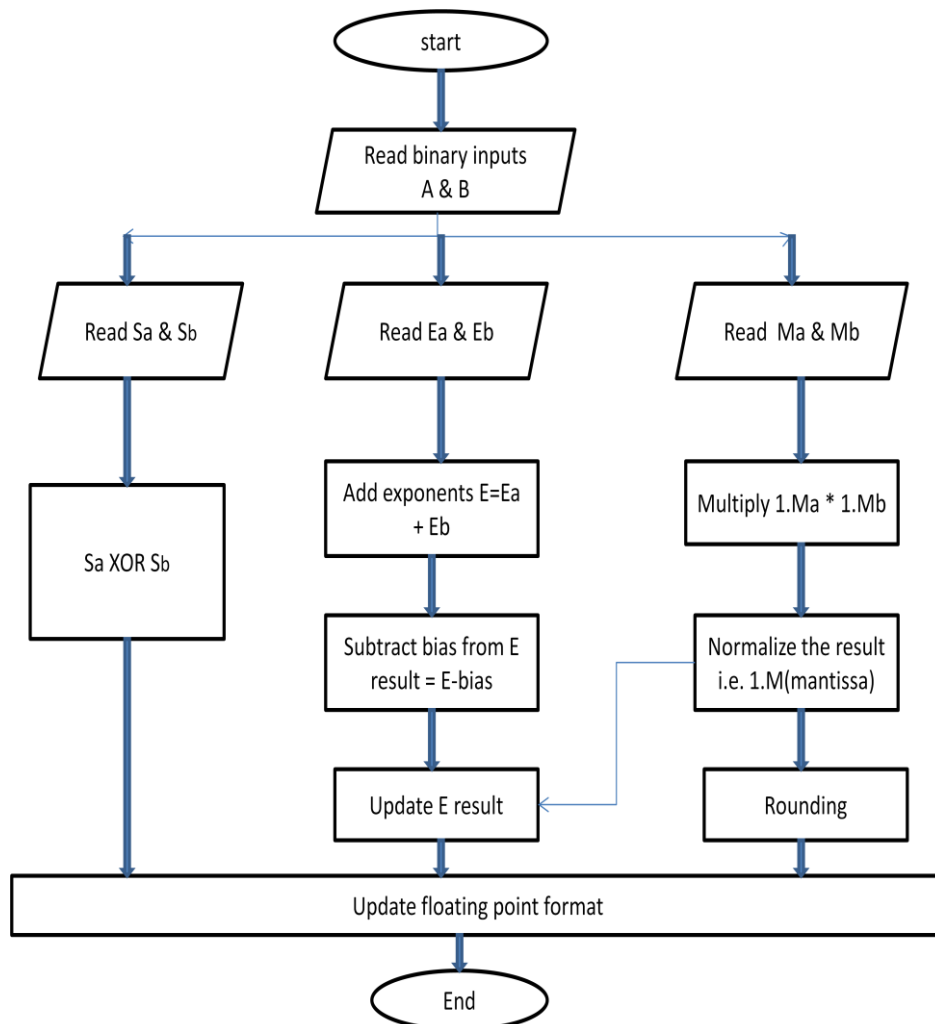


Figure 2: Black box view of floating point double precision multiplier.

The input signals to the top level module are

1. Clk
2. Rst
3. Enable
4. Opa (64 bits)
5. Opb (64 bits)

Example for Multiplication:

- Consider multiplying: 1.0102×2^{-1} by -1.1102×2^{-2}
 As before, we assume 4 bits of precision (or 3 bits of fraction)
- Unlike addition, we add the exponents of the operands
 Result exponent value = $(-1) + (-2) = -3$
- Using the biased representation: $EZ = EX + EY - Bias$
 $EX = (-1) + 127 = 126$ ($Bias = 127$ for SP)
 $EY = (-2) + 127 = 125$
 $EZ = 126 + 125 - 127 = 124$ (value = -3)
- Now, multiply the significands:

$$\begin{array}{r}
 1.010 \\
 1.110 \\
 \hline
 0000 \\
 1010 \\
 1010 \\
 1010 \\
 \hline
 10001100
 \end{array}$$

$$(1.010)2 \times (1.110)2 = (10.001100)2.$$

- Since sign $SX \neq SY$, sign of product $SZ = 1$ (negative)
- So, $1.010 \times 2^{-1} \times -1.110 \times 2^{-2} = -10.001100 \times 2^{-3}$
- However, result: -10.001100×2^{-3} is NOT normalized
- Normalize: $10.001100 \times 2^{-3} = 1.0001100 \times 2^{-2}$
 Shift right by 1 bit and increment the exponent, At most 1 bit can be shifted right
- Round the significand to nearest:

$$\begin{array}{l}
 1.000 \mid 1100 \\
 1.0001100 \approx 1.001 \text{ (3-bit fraction)}
 \end{array}$$

$$+ \quad 1 \quad \text{Result} \approx -1.001 \times 2^{-2} \text{ (normalized)}$$

- Detect overflow / underflow 1.001
 No overflow / underflow because exponent is within range.

Over flow:

An over flow occurs when the number is too large to be stored.

Example:

$$\begin{array}{r}
 1101 \\
 1010 \\
 \hline
 10111
 \end{array}$$

Under flow:

Under flow refer to floating point under flow, where an operation results in a number that is too small to be represent able. For example ,if exponent part can

represent values in -1023 to +1023 ,then any number with absolute value less than 2^{-1023} may cause underflow.

Rounding:

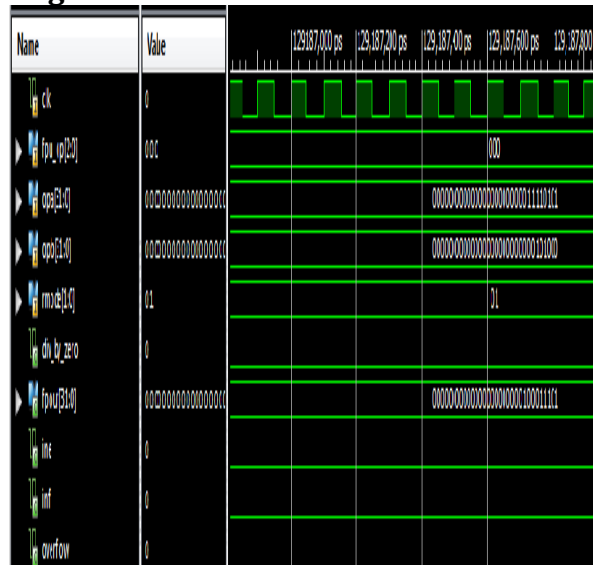
The IEEE standard specifies four rounding modes round to nearest, round to zero, round to positive infinity, and round to negative infinity. Table 1 shows the rounding modes selected for various bit combinations of rmode. Based on the rounding changes to the mantissa corresponding changes has to be made in the exponent part also.

TABLE 1: Rounding Modes Selected for Various Bit Combinations of Mode

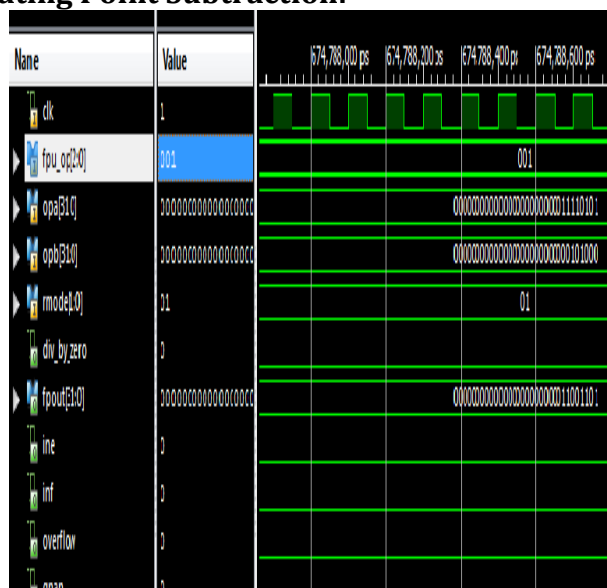
Bit Combination	Rounding Mode
00	Round_Nearest_Even
01	Round_To_Zero
10	Round_Up
11	Round_Down

Results:

Single Precision Floating Point Addition:



Single Precision Floating Point Subtraction:



Slice Logic Utilization:

Number of Slice Registers	19 out of 301440
Number of Slice LUTs	231 out of 150720
Number used as Logic	231 out of 150720

Slice Logic Distribution:

Number of LUT Flip Flop pairs used	231
Number with an unused Flip Flop	212 out of 231 91%
Number with an unused LUT	0 out of 231 0%
Number of fully used LUT-FF pairs	19 out of 231 8%
Number of unique control sets	1

IO Utilization:

Slice Logic Distribution:	247
Number of bonded IOBs	246 out of 400 61%

Specific Feature Utilization:

Number of BUFG/BUFGCTRL/BUFHCEs	1 out of 176 0%
Number of DSP48E1s	15 out of 768 1%

Conclusion:

The double precision floating point adder/subtractor and multiplier support the IEEE-754 binary interchange format, targeted on a Xilinx Virtex-6 FPGA. It provides more accuracy when compared to single precision floating point. These designs handle the overflow, underflow, rounding mode and various exception conditions.

References:

1. V. Oklobdzija, "An algorithmic and novel design of a leading zero detector circuit: comparison with logic synthesis", IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 2, no. 1, (1994) March, pp. 124–128.
2. P. Belanovic and M. Leaser, "A Library of Parameterized Floating-Point Modules and Their Use", in 12th International Conference on Field-Programmable Logic and Applications (FPL-02). London, UK: Springer-Verlag, (2002) September, pp. 657–666.
3. K. Hemmert and K. Underwood, "Open Source High Performance Floating-Point Modules", in 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM-06), (2006) April, pp. 349–350.
4. Malik and S. -B. Ko, "A Study on the Floating-Point Adder in FPGAs", in Canadian Conference on Electrical and Computer Engineering (CCECE-06), (2006) May, pp. 86–89.
5. D. Sangwan and M. K. Yadav, "Design and Implementation of Adder/Subtractor and Multiplication Units for Floating-Point Arithmetic", in International Journal of Electronics Engineering, (2010), pp. 197-203.
6. M. K. Jaiswal and R. C. C. Cheung, "High Performance FPGA Implementation of Double Precision Floating Point Adder/Subtractor", in International Journal of Hybrid Information Technology, vol. 4, no. 4, (2011) October.
7. B. Fagin and C. Renard, "Field Programmable Gate Arrays and Floating Point Arithmetic", IEEE Transactions on VLSI, vol. 2, no. 3, (1994), pp. 365–367.
8. N. Shirazi, A. Walters and P. Athanas, "Quantitative Analysis of Floating Point Arithmetic on FPGA Based Custom Computing Machines", Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'95), (1995), pp. 155–162.

9. L. Louca, T. A. Cook and W. H. Johnson, "Implementation of IEEE Single Precision Floating Point Addition and Multiplication on FPGAs", Proceedings of 83rd IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'96), (1996), pp. 107-116.
10. Jaenicke and W. Luk, "Parameterized Floating-Point Arithmetic on FPGAs", Proc. of IEEE ICASSP, vol. 2, (2001), pp. 897-900.
11. Lee and N. Burgess, "Parameterisable Floating-point Operations on FPGA", Conference Record of the Thirty-Sixth Asilomar Conference on Signals, Systems, and Computers, (2002).
12. M. Al-Ashrasfy, A. Salem, W. Anis, "An Efficient Implementation of Floating Point Multiplier", Saudi International Electronics, Communications and Photonics Conference (SIEPC), (2012)