# HIGH-SPEED, AREA-EFFICIENT FPGA-BASED FLOATING-POINT ARITHMETIC MODULES

M. Taher*, M. Aboulwafa**, A. Abdelwahab**, E. M. Saad**

*: Tibben Institute for Metallurgical Studies, **: Faculty of Engineering, Helwan University

**Abstract** In this paper, single-precision floating-point IEEE-754 standard Adder/Subtractor and Multiplier modules with high speed and area efficient are presented. These modules are designed, simulated, synthesized and optimized by using Mentor Graphics Tools, and they are implemented on an FPGA based system by using the Xilinx Tool (ISE). A comparison between the results of the proposed design and a previously reported one is provided. The effect of normalization unit at the single-precision floating-point multiplier and adder/Subtractor modules on the area, and speed is explained. An FIR filter is implemented on FPGA as an application example.

## 1. INTRODUCTION

The floating-point arithmetic modules were virtually impossible to be implemented on the older generations of FPGAs due to its limited density and speed.

Recently, the density and speed of FPGA are increased, so it becomes easy to implement floating-point arithmetic modules on it. With the appearance of high-level languages such as VHDL, rapid prototyping of floating point units has become feasible. Simulation and synthesis tools at a higher-level design aid the designer for a more controllable and maintainable product. Although low-level design specifications were alternately possible, the strategy used in the design that is presented here is to specify every aspect of the design in VHDL and rely on automated synthesis to generate the FPGA mapping.

The usage of floating point helps to manipulate the underflow and overflow problems often seen in fixed-point formats. This paper examines the implementations of floating-point arithmetic modules using single precision floating-point IEEE-754 standard format [1]. These modules have been synthesized on Xilinx Virtex-II XC2V6000bf957 FPGAs [2].

The general computing world has settled on floating-point formats, which conform to IEEE-754 standard [3]. These standards play a crucial role in ensuring numerical robustness and code compatibility among machines of vastly different architectures. However, the choice of floating-point format has such a dominant impact on FPGA implementation cost that the standards are often bent, giving the designer freedom to choose a custom floating-point format in order to spend FPGA resources as efficiently as possible. For example, work has been done to automatically determine custom floating-point bit widths for each node of computation [4]; others have demonstrated the suitability of very tiny floating-point formats with much less precision and range than IEEE single-precision [5].

## 2. FLOATING-POINT FORMAT REPRESENTATION

The floating-point format, which is used in this design, is the single-precision floating-point of IEEE-754 standard format [1] as shown in Fig. 1.

The floating-point value (V) is computed by

$$V = (-1)^{s} \times 2^{(e-bias)} \times (1.f) \qquad \textbf{(2.1)}$$

As illustrated in Figure 1, the sign field, s, is bit number 31 and is used to specify the sign of the number, if s equals one the value will be negative, but if s equals zero the value will be positive. Bits 30 down to

23 are the exponent field. This 8-bit quantity is a signed number represented by using a bias of 127. Bits 22 down to 0 are used to store the binary representation of the fraction for the floating-point number. The leading one in the mantissa, 1.f, does not appear in the representation; therefore the leading one is implicit. For example, -3.625 (decimal) or -11.101 (binary) will be normalized as illustrated in equation (2.2) [6] and the number is stored as in Figure 2.

$$V= (-1)1\ 2(128-127)\ (1.1101) \qquad \textbf{(2.2)}$$

Where s = 1, e = the times of right shifting for the number + the bias, the bias for single precision is 127, and the times of right shifting for this value " 11.101" is one, so e = 1+127 = 128(decimal) = 80 (hex), and f=680000 (hex). Therefore, -3.625 is stored as: C0680000 (hex) as illustrated in Fig. 2.

## 3. FLOATING POINT MULTIPLIER

Floating-point multiplication is similar to integer multiplication, because floating-point numbers are stored in signed-magnitude form; the multiplier needs only to deal with unsigned integer numbers and normalization. The optimized design of the single-precision floating-point multiplier has a latency of one clock cycle. The presented design has some ideas of that of Shirazi's 18-bits floating-point format multiplier [9]. The bottleneck of this design is the normalization unit. To normalize the multiplication results, only the most significant bit of the mantissa is checked: if it is 1 no shift is required, but if it is 0, one bit shift left is done. The optimization of the normalization allows the multiplier to run at slightly faster clock speed. It also helps in reducing the usage area [7].

### 3.1. Algorithm:

The flowchart for a single-precision semi-parallel floating-point multiplier is shown in Figure 3.
Where:

V: value represented in a single precision format
s: sign bit.
e: exponent.
f: fraction.
m: mantissa.
NaN: Not a Number.

### 3.2. Simulation Results:

These modules are designed, simulated, synthesized and optimized by using Mentor Graphics Tools, and they are implemented on an FPGA based system by using the Xilinx Tool (ISE).

Figure 4 illustrates a test sample for single-precision floating-point multiplier, where the output was updated for one clock cycle. The data are represented in hexadecimal form. Table 1 shows the test results of the multiplications of a set of input vectors in a hexadecimal radix format.

**Table 1** The test results of the presented Floating-Point Multiplier

| v1 | | v2 | | V | |
|---|---|---|---|---|---|
| Decimal | Hexadecimal | Decimal | Hexadecimal | Decimal | Hexadecimal |
| -3.625 | C0680000 | 20.052 | 41A06A7F | -72.6885 | C2916083 |
| 6.0037 | 40C01E4F | -10.1007 | C1219C78 | -60.641571 | C27290F8 |
| 0 | 00000000 | 35.0524 | 420C35A8 | 0 | 00000000 |
| 51.2305 | 424CEC08 | 4.0002 | 408001A3 | 204.93222 | 434CEEA6 |
| -7.0505 | C0E19DB2 | -8.1016 | C101A027 | 57.120327 | 42647B37 |

## 3.3. Synthesis Results:

The proposed single-precision semi-parallel floating-point multiplier module is implemented using VHDL Design Entry. It is mapped on the same FPGA chip that is used in [8] (Xilinx Virtex-II XC2V6000bf957). The synthesis results of the proposed configuration are compared with previous published results in [8] as shown in table 2.

By comparing the results of the proposed technique that are given in table 2 with those results in [8] also shown in the same table, it can be seen that the used area in our design is reduced by 55% while the speed is increased by 336.3%.

**Table 2** The comparison of the synthesis results

| The results of the proposed configuration | | The results in [8] | |
|---|---|---|---|
| Function generator (F.G.) | Speed | Function generator (F.G.) | Speed |
| 202 | 11.24 ns = 89 MHZ | 452 | 49 ns = 20.4 MHZ |

## 4. FLOATING POINT ADDER/SUBTRACTOR

An optimized design of the 32-bit floating-point Adder/Subtractor has a latency of one clock cycle is proposed. The presented design has some idea as that of Shirazi's 18-bit floating-point format Adder/Subtraction in [9]. But, the configuration of the normalization module allows the Adder/Subtraction to run at a slightly faster clock speed and also helps to reduce the used area [7]. The bottleneck of this design was the normalization unit.

## 4.1. Algorithm:

The flowchart of a single-precision cascaded floating-point Adder/Subtractor is shown in Figure 5.

Where:

V: value represented in a single precision format
s: sign bit.
e: exponent.
f: fraction.
m: mantissa.
e_sub: selection line to perform the addition or subtraction processes.
NaN: Not a Number.

### 4.1.1. Normalization:

Every four bits of the mantissa will be the inputs to an OR-gate. Then, "which of the six outputs of the OR-gates that is the leading one" can be detected rapidly, and the leading-one detection logic decides which of the six nibbles of the mantissa value contains the leading-one.

After that, the 5-bit shift value using the data word from the leading-one detection logic that determines which of the six nibbles in the resulting mantissa the one resides in. The data word can be used to determine what the upper three bits of the shift value are to be while the lower two bits are determined by the bit values in the nibble containing the leading one. The combinational logic to determine the lower two bits can be constructed from two, 4-variable logic equations:

$$S_0 = (\text{not } n_3) \text{ and } (n_2 \text{ or } ((\text{not } n_1) \text{ and } n_0)) \qquad \mathbf{(4.1)}$$
$$S_1 = (\text{not } n_3) \text{ and } (\text{not } n_2) \text{ and } (n_1 \text{ or } n_0) \qquad \mathbf{(4.2)}$$

Where: $s_0$ and $s_1$ are bits 0 and 1 of the constructed shift value, respectively. The $n_3$, $n_2$, $n_1$, and $n_0$ values represent bits 3 to 0, respectively, of the nibble containing the leading-one.

## 4.2. Simulation Results:

These modules are designed, simulated, synthesized and optimized by using Mentor Graphics Tools, and they are implemented on an FPGA based system by using the Xilinx Tool (ISE).

Figure 6 illustrates a simulation sample for single-precision floating-point adder/subtractor, where the output was updated for one clock cycle. The data are represented in hexadecimal form. Table 3 shows the results of the additions/subtractions operations of a set of input vectors in a hexadecimal radix format.

**Table 3** The simulation results of the presented Floating-Point Adder/Subtractor

| v1 | | v2 | | | V | |
|---|---|---|---|---|---|---|
| Decimal | Hexadecimal | Decimal | Hexadecimal | e_sub | Decimal | Hexadecimal |
| -3.625 | C0680000 | 20.052 | 41A06A7F | 0 | 16.427002 | 41836A80 |
| 6.0037 | 40C01E4F | -10.1007 | C1219C78 | 0 | -4.0970020 | C0831AA4 |
| 0 | 00000000 | 35.0524 | 420C35A8 | 0 | 35.052399 | 420C35A8 |
| 51.2305 | 424CEC08 | 4.0002 | 408001A3 | 0 | 55.230698 | 425CEC3C |
| -7.0505 | C0E19DB2 | -8.1016 | C101A027 | 0 | -15.152100 | C1726F00 |
| -3.625 | C0680000 | 20.052 | 41A06A7F | 1 | -23.677002 | C1BD6A80 |
| 6.0037 | 40C01E4F | -10.1007 | C1219C78 | 1 | 16.104401 | 4180D5D0 |
| 0 | 00000000 | 35.0524 | 420C35A8 | 1 | -35.052399 | C20C35A8 |
| 51.2305 | 424CEC08 | 4.0002 | 408001A3 | 1 | 47.230301 | 423CEBD4 |
| -7.0505 | C0E19DB2 | -8.1016 | C101A027 | 1 | 1.0510998 | 3F868A70 |

## 4.3. Synthesis Results:

The proposed single-precision cascaded floating-point adder/subtractor module is implemented using VHDL Design Entry. It is mapped on the same FPGA chip that is used in [8] (Xilinx Virtex-II XC2V6000bf957). The synthesis results of the proposed configuration are compared with previous published results in [8] as shown in table 4.

By comparing the results of the proposed technique given in table 4 with corresponding results in [8] that are shown in the same table, it can be seen that the used area in our design is reduced by 6% while the speed is increased by 68%.

**Table 4** The comparison of the synthesis results

| The results of the proposed configuration | | The results in [8] | |
|---|---|---|---|
| Function generator (F.G.) | Speed | Function generator (F.G.) | Speed |
| 490 | 30.67 ns = 32.6 MHZ | 521 | 51.5 ns = 19.4 MHZ |

## 5. ILLUSTRATIVE EXAMPLE

Implementation of an FIR filter on FPGA as an application example for the presented arithmetic modules is considered. The transfer function H (Z) of the considered FIR filter is given as follows [10]:

$$H(Z) = 1 + 2Z^{-1} + 3Z^{-2} + 4Z^{-3} + 3Z^{-4} + 2Z^{-5} + Z^{-6} \quad \textbf{(5.1)}$$

This transfer function is also verified using Matlab. Figure 7 shows the frequency responses of that FIR structure using both the proposed method and the Matlab [7]. Figure 8 shows the error ratio of the proposed method Where:

The frequency response of the proposed filter in Matlab is labeled by black line.

The frequency response of the proposed filter in VHDL is labeled by red dots.

This demonstrates that the proposed technique can be used to implement digital structures such as digital filters with reasonable accuracy. Moreover, this approach needs less chip area with increasing speed which is suitable for VLSI technology.

## 6. CONCLUSION

A design of the single-precision floating-point arithmetic modules with an optimized area and speed is presented. The effect of normalization on the area and speed has been examined experimentally. The design has been mapped on Xilinx vertex-II XC2V6000bf957. Comparisons of results between the proposed systems and previously published results have been demonstrated. The presented single-precision floating-point multiplier, adder, and subtractor modules run at slightly faster clock speed with used area less than that used previously.

## REFERENCES

[1] IEEE Task P754, “A Proposed Standard for Binary Floating-Point Arithmetic,” IEEE Computer, Vol.14, No.12, pp.51-62, Mar.1981.
[2] Xilinx, Inc., the Programmable Logic Data Book, San Jose, California, 1993.
[3] IEEE Standards Board. IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE STD 754-1985 edition, 1985.
[4] A.A.Gaffar, O.Mencer, W.Luk, P.Y.Cheung, and N.Shirazi. "Floating Point Bit width Analysis via Automatic Differentiation". Proceedings of the International Conference on Field Programmable Technology, 2002.
[5] J.Dido et al. "A Flexible Floating-Point Format for Optimizing Data-Paths and Operators in FPGA Based DSPs". ACM/SIGDA Tenth ACM International Symposium on Field-Programmable Gate Arrays (FPGA’02), 2002.
[6] GH.A .At y, A. Hussein, I. Ashour, and M. Mones, “High-speed area-efficient FPGA-based floating-point multiplier", Proceedings of ICM 2003 Conference, Dec.2003, Cairo, EGYPT, pp.274-277.
[7] M. Taher, "Design and Implementation of High Performance FPGA Based Floating-Point DSP Functions", Master’s Thesis, Helwan University, Faculty of Engineering, Egypt, 2006.
[8] Bryan Cantanzaro, Brent Nelson, “Higher Radix Floating-Point Representations for FPGA-Based Arithmetic” in Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machiens, 2005.
[9] N.Shirazi, A. Walters, and P. Athanas, “Quantitative analysis of floating point arithmetic on FPGA based custom computing machines“, in Proceedings of IEEE Symposium on FPGAs for custom Computing machines, pp.155-162, 1995.
[10] John G. Proakis and Dimitris G. Manolakis, "Digital Signal Processing, Principles, Algorithms, and Applications", Prentice hall, 1996.

Bit # 31 30 23 22 0

| s | e | f |
|---|---|---|

**Fig.1.** 32 Bits Floating Point format

| S | e | | | f | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 100 | 0000 | 0 | 110 | 1000 | 0000 | 0000 | 0000 | 0000 |
| C | | 0 | 6 | | 8 | 0 | 0 | 0 | 0 |

**Figure 2** Representation of –3.625 in a single precision floating-point format

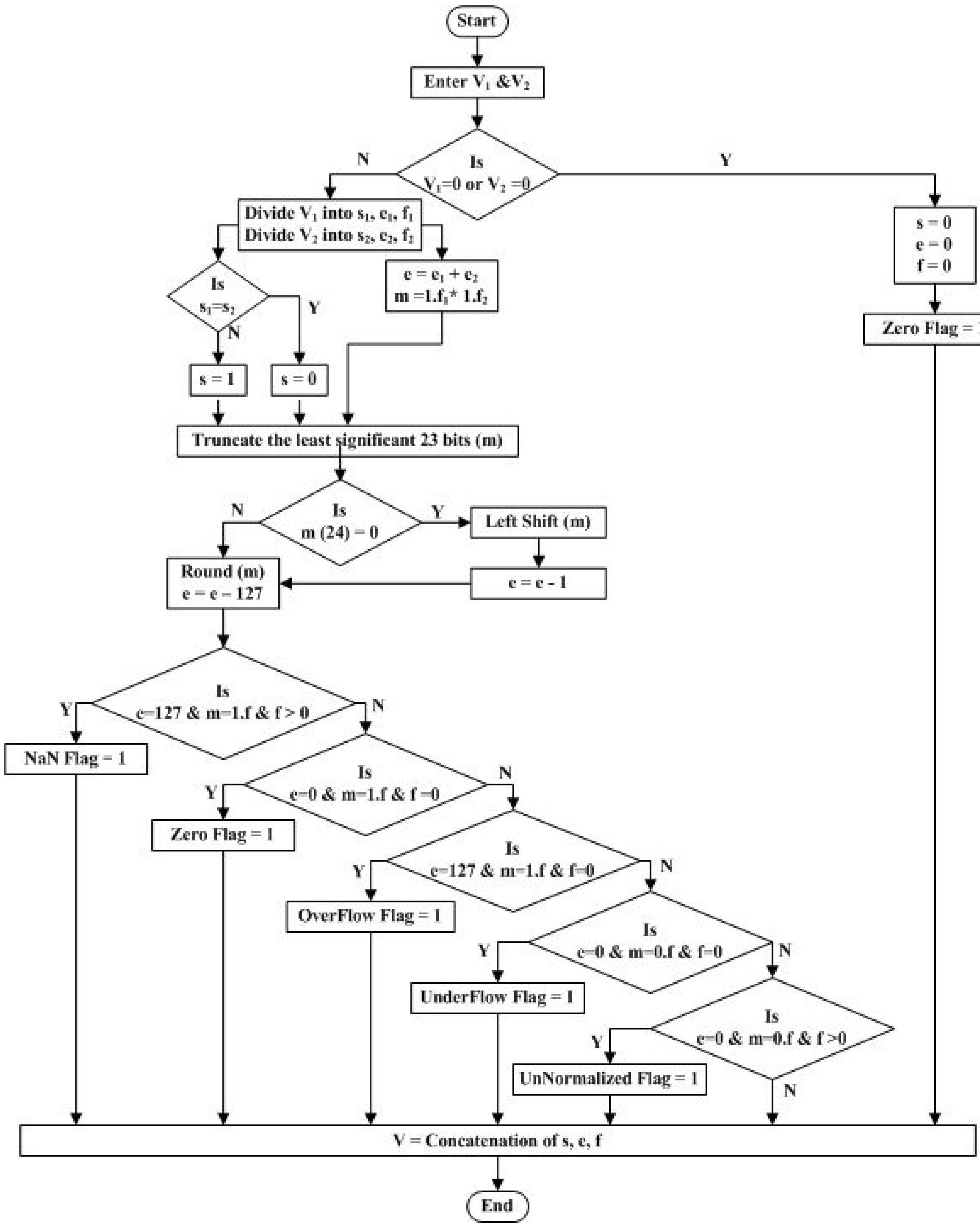

**Figure 3** Flowchart for single precision floating-point multiplier

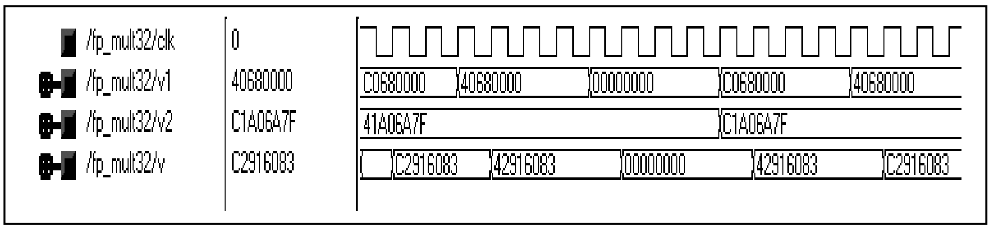
```
/fp_mult32/clk   0
/fp_mult32/v1    40680000   C0680000  40680000  00000000  C0680000  40680000
/fp_mult32/v2    C1A06A7F   41A06A7F                      C1A06A7F
/fp_mult32/v     C2916083             C2916083  42916083  00000000  42916083  C2916083
```

**Figure 4** The simulation results of the proposed Multiplier

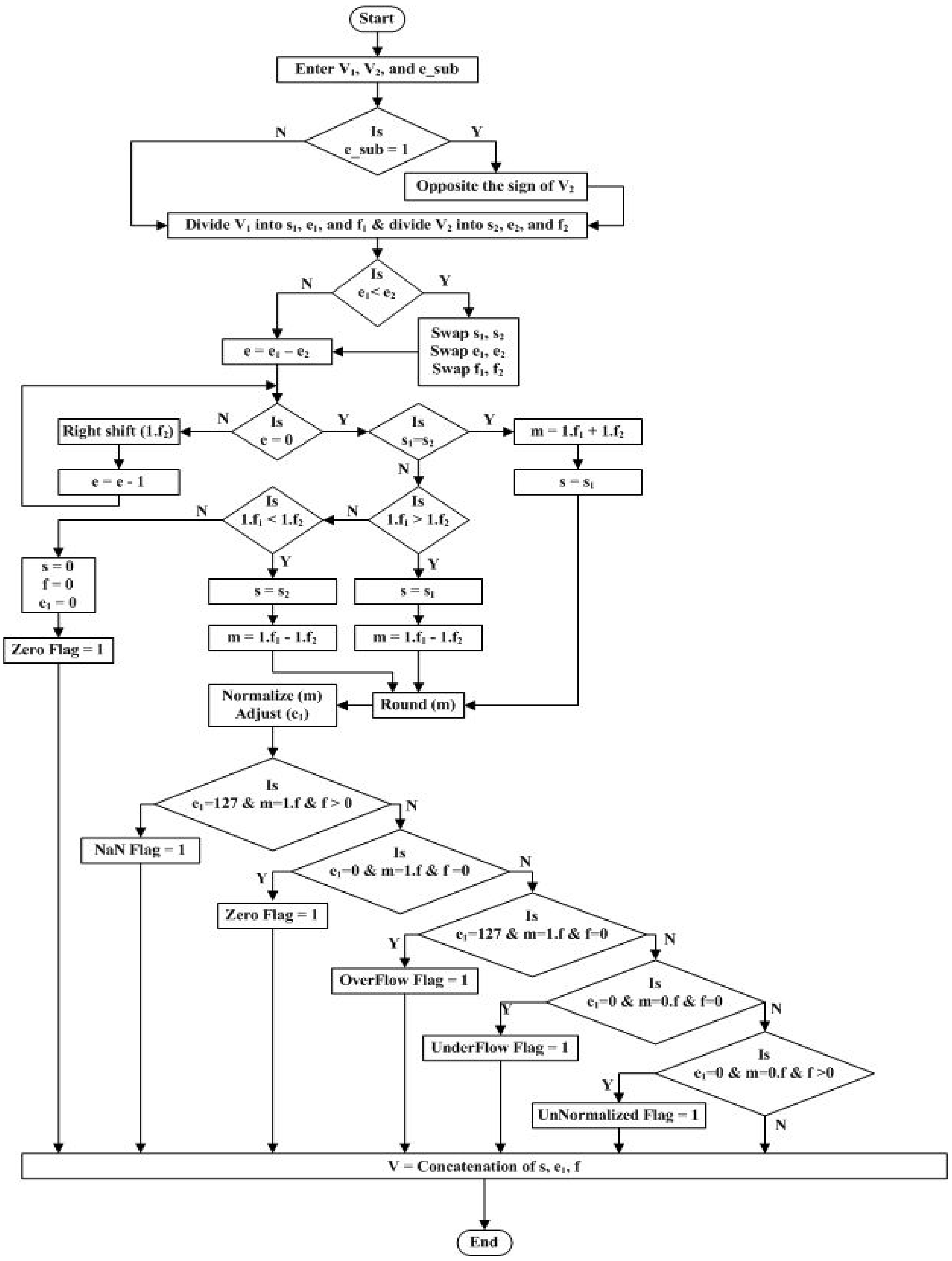

**Figure 5** Flowchart for single precision floating-point adder/subtraction

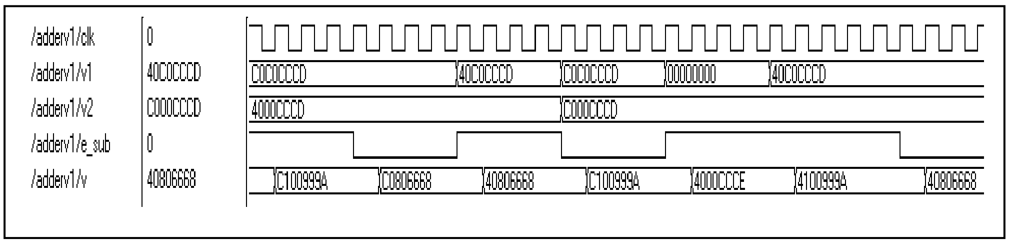

**Figure 6** The simulation results of the proposed Adder/Subtractor

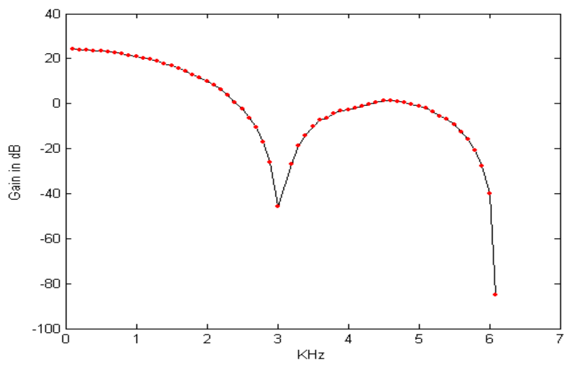

**Figure 7** The transfer responses of the presented FIR filter in Matlab & VHDL

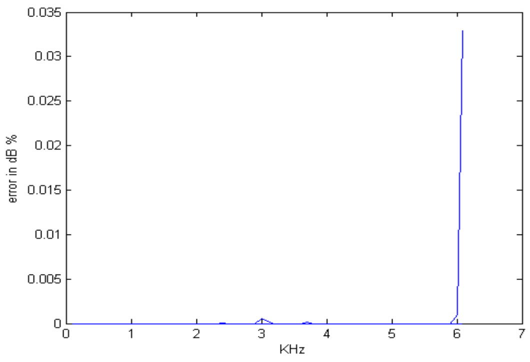

**Figure 8** The error ratio