HDL Design of Efficient Floating Point Multiplier

Sadiya Fatima Sufi
Department of Electronics and Telecommunication Engineering
NUVA College of Engineering and Technology
Nagpur, India
Fatimasufi23@gmail.com

Pooja Thakre
Department of Electronics and Telecommunication Engineering
NUVA College of Engineering and Technology
Nagpur, India

Abstract—We present a VHDL implementation of a floating point multiplier. FPGA synthesis optimization techniques of pipelining and retiming have been used to improve the performance compared to the reference baseline design.

Keywords-*floating point multiplication, hardware description language, VHDL, digital logic, FPGA*

*****

I. Introduction

Floating point numbers are frequently used for numerical calculations in computing systems. Arithmetic units for floating point numbers are considerably more complex than those for fixed-point numbers. However, recent advances in VLSI technology have increased the feasibility of hardware implementations of floating point arithmetic units. The main advantage of floating point arithmetic is that its wide dynamic range virtually eliminates overflow and underflow in most applications. In this paper, we describe a simple representation of floating point numbers. Next, an algorithm is developed for floating point multiplication. This algorithm is implemented in VHDL. The design of the floating point multiplier is completed and implemented on an FPGA.

VHDL and Verilog are the two industry standard hardware description languages for digital ASIC and FPGA design. Figure 1 below summarizes a comparison between these two HDLs.

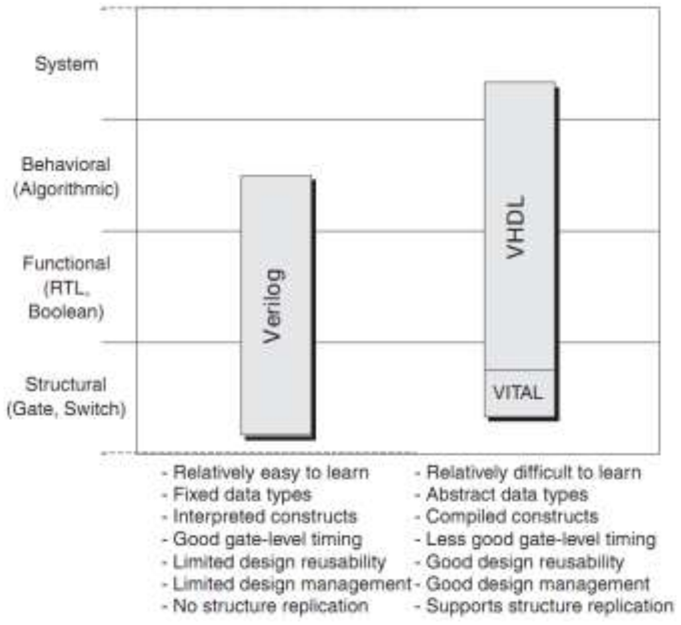

System
Behavioral (Algorithmic)
Functional (RTL, Boolean)
Structural (Gate, Switch)
Verilog
VHDL
VITAL
- Relatively easy to learn
- Fixed data types
- Interpreted constructs
- Good gate-level timing
- Limited design reusability
- Limited design management
- No structure replication
- Relatively difficult to learn
- Abstract data types
- Compiled constructs
- Less good gate-level timing
- Good design reusability
- Good design management
- Supports structure replication

Figure 1. Comparison between VHDL and Verilog

VHDL was chosen for this implementation because it is better suited for design reusability. The floating point multiplier is a subsystem that may be employed in a larger floating point arithmetic unit or a digital filter implementation.

The remainder of the paper is organized as follows: section 2 gives a brief overview of floating point number representations, which a focus on the IEEE 754 formats, which are widely used as a standard. Section 3 describes the algorithm used to implement floating point multiplication in digital hardware. Section 4 describes the implementation details and provides a discussion of the results.

II. Representation of Floating-Point Numbers

A. *A simple floating point representation*

A simple representation of a floating point number (N) uses a binary fraction (F) and exponent (E), where $N = F \times 2^E$. We represent negative fractions and exponents using 2's complement form. In a typical floating point number system, F is 16 to 64 bits long and E is 8 to 15 bits long. We begin with a representation with a four-bit fraction (F) and a four-bit exponent (E).

In order to utilize all bits in F and have the maximum possible number of significant digits, F should be normalized so that its magnitude is as large as possible. If F is not normalized, we can normalize F by shifting it left until the sign bit and the next bit are different. Shifting F is equivalent to multiplication by 2, so every time we shift, we must decrement E by 1 to keep N the same. After normalization, the magnitude of F will be as large as possible, since any further shifting would change the sign bit.

Zero cannot be normalized so F = 0.000 when N = 0. Any exponent can then be used, but maintain uniformity, we will associate the negative exponent with the largest magnitude to represent zero. In a four-bit 2's complement number, -8 is the most negative number, and it is represented as 1000. Some floating point representations use a bias exponent so that E = 0 is associated with F = 0.

B. *The IEEE-754 floating point representation*

The IEEE 754 single precision (32-bit) floating point format, which is widely implemented, has an 8-bit biased integer exponent, which ranges between 0 and 255. The exponent is expressed in an excess-127 code so that its effective value is determined by subtracting 127 from the stored value. Thus the range of effective values of the exponent is -127 to 128, corresponding to the stored values of 0 to 255, respectively. A stored exponent value of 0 (E_{min}) serves as a flag indicating that the value of the number is 0 if the significand is 0, and for denormalized numbers if the significand is nonzero. A stored value of 255 (E_{max}) serves as a flag indicating that the value of the number is infinity if the significand is zero and for “not a number” if the significand is nonzero. The significand is a 25-bit sign magnitude mixed number where the binary point is to the right of the most significant bit. The leading bit of the significand is always a 1 except for denormalized numbers. As a result, when numbers are stored, the leading bit is omitted, giving an extra bit of precision.

Figure 2 depicts the single precision floating point format. The leftmost bit is the sign bit, 0 for positive and 1 for negative numbers. There is an 8-bit exponent field E, and a 23-bit mantissa field M. The exponent is with respect to the radix 2. Because it is necessary to be able to represent both very large and very small numbers, the exponent can be either positive or negative. Instead of simply using an 8-bit unsigned number as the exponent, which would allow exponent values in the range of -128 to 127, the IEEE standard specifies an exponent in the excess 127 format. In this format, a value of 127 is added to the value of the exponent so that E becomes a positive integer. This format is convenient for adding and subtracting floating point numbers because the first step in these operations involves comparing the exponents to determine whether the mantissas must be appropriately shifted to add or subtract the significant bits. The range of E is 0 to 255. The extreme values of E=0 and E=255 are used to denote exact zero and infinity respectively. Therefore the normal range of the exponent is -126 to 127, which is represented by the values of E from 1 to 254.

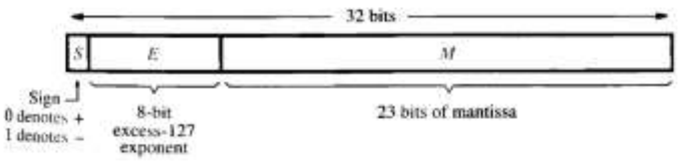


Figure 2. IEEE 754 single precision (32-bit) floating point number

The mantissa is represented using 23 bits. The IEEE standard calls for a normalized mantissa which means that the most significant digit is always equal to 1. This it is not necessary to include this bit explicitly in the mantissa field. Therefore, if M is a bit vector in the mantissa field, the actual value of the mantissa is 1.M, which gives a 24 bit mantissa. Consequently the floating point format in figure 1 represents the number

$$+/- 1.M \times 2^{E-127}$$

The mantissa field allows a precision of about 7 decimal digits, and the exponent field range corresponds to between 10^{-38} and 10^{38}

The double precision format is an extension of the single precision format and uses 64 bits (figure 3). Both the mantissa and exponent fields are larger, thus allowing for greater precision and range. The exponent field has 11 bits and it is specified in the excess 1023 format where

$$\text{Exponent} = E – 1023$$

The range of E is 0 to 2047, but the values of E=0 and E=2047 are used to indicate exact zero and infinity respectively. Thus the normal range of the exponent is from negative 1022 to positive 1023, which is represented by values of E from 1 to 2046.

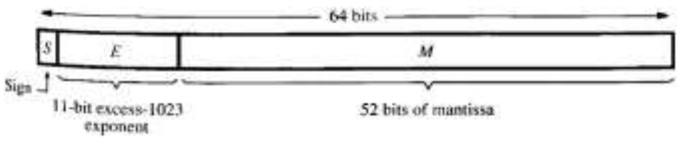


Figure 3. IEEE 754 double precision (64-bit) floating point number

The mantissa field has 52 bits. Since the mantissa is assumed to be normalized, its actual value is again 1.M. Therefore the value of the floating point number is

$$\text{Value} = +/- 1.M \times 2^{E-1023}$$

This format allows representation of numbers that have a precision of about 16 decimal digits and a range of approximately 10^{-308} to 10^{308}.

III. FLOATING POINT MULTIPLICATION

Given two floating point numbers, the product is

$$(F1 \times 2^{E1}) \times (F2 \times 2^{E2}) = (F1 \times F2) \times 2^{(E1 + E2)}$$

The fraction part of the product is the product of the fractions, and the exponent part of the product is the sum of the exponents. We assume that F1 and F2 are normalized to begin with, and that the final result will also be normalized.

Basically, all that is required is that we multiply the fractions and add the exponents. However, several special cases must also be considered. First, if F = 0, then we must set the exponent E to the largest negative value 1000. Second, if we multiply -1 by -1 (1.000 x 1.000) the result should be +1. Since we cannot represent +1 as a 2’s complement fraction, we call this special case a fraction overflow. To correct this situation, we set F = ½ (0.100) and add 1 to the exponent E. This is justified, since $1 \times 2^{E} = ½ \times 2^{E+1}$

The algorithm for floating point multiplication forms the product of the operand significands and the sum of the operand exponents. For radix-2 floating point numbers, the significand values are greater than or equal to 1 and less than 2. The product of two such numbers will be greater than or equal to 1 and less than 4. At most a single right shift is required to normalize the product.

The flowchart for the floating point multiplication algorithm is given in figure 3 below.

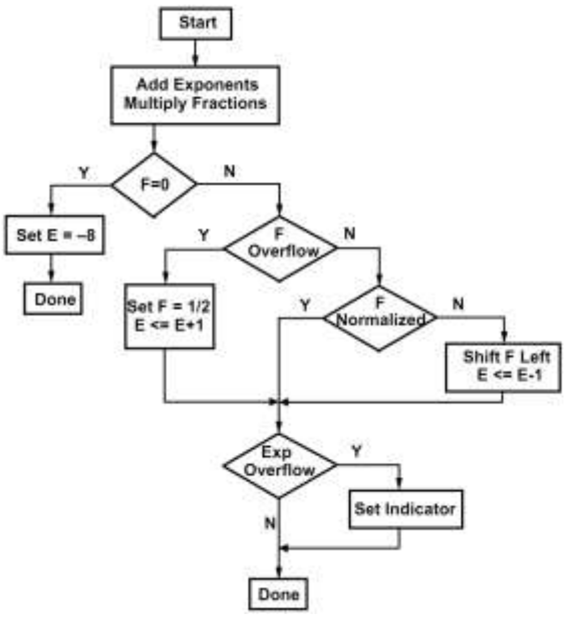


Figure 4. Flowchart for floating point multiplication

IV. IMPLEMENTATION

The floating point multiplier is made up of a control unit, an exponent adder, and a fraction multiplier. The exponent adder is given in figure 5, the fraction multiplier is given in figure 6, and the main control unit state machine chart is given in figure 7. The state diagram for the multiply control unit used by the fraction multiplier is given in figure 8.

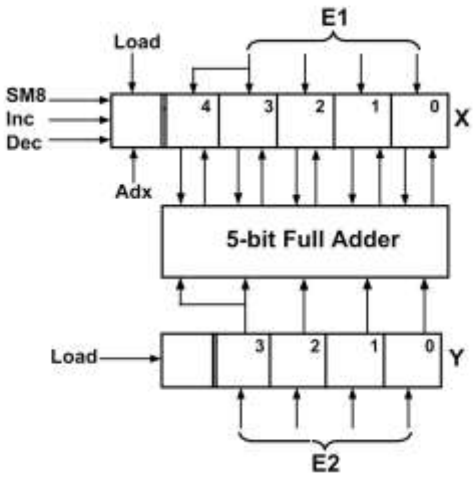


Figure 5. Exponent adder

The function of the exponent adder, as stated previously, is to add the exponents of the two floating point numbers, as multiplication entails the addition of exponents. The fraction multiplier uses the shift-and-add method to multiply the mantissas of the two numbers. This shift and add operation is controlled by the multiply control unit.

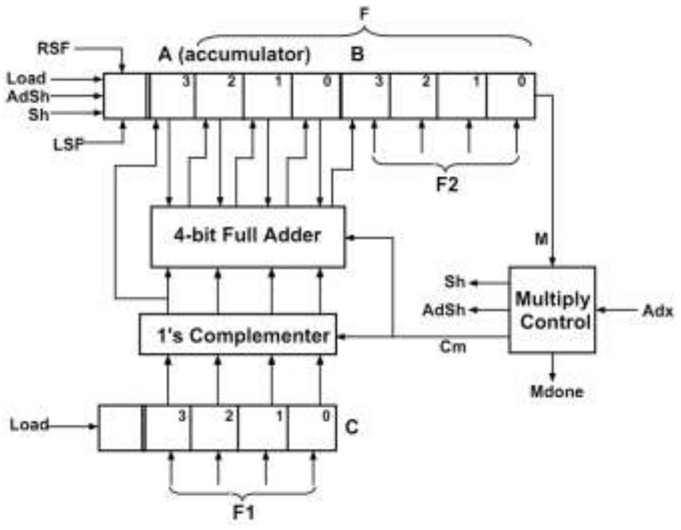


Figure 6. Fraction multiplier

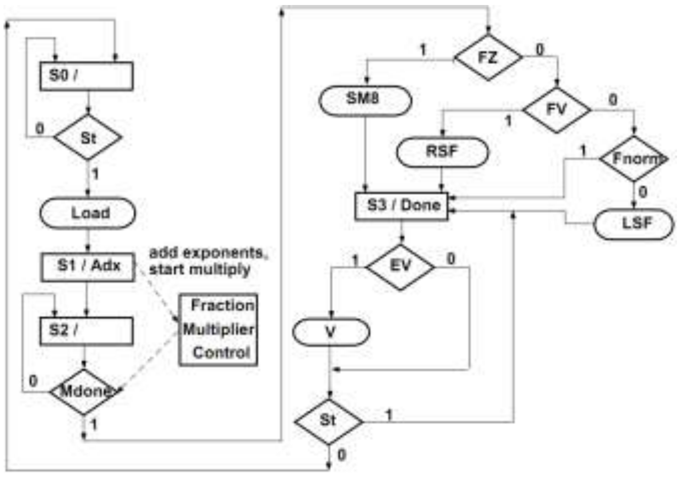


Figure 7. State machine chart for the floating point multiplier

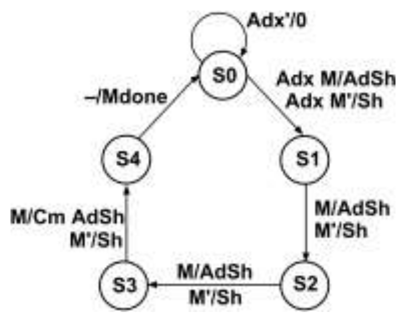


Figure 8. State diagram for the multiplier control unit

The top-level schematic for the floating point multiplier is given in figure 9 below.

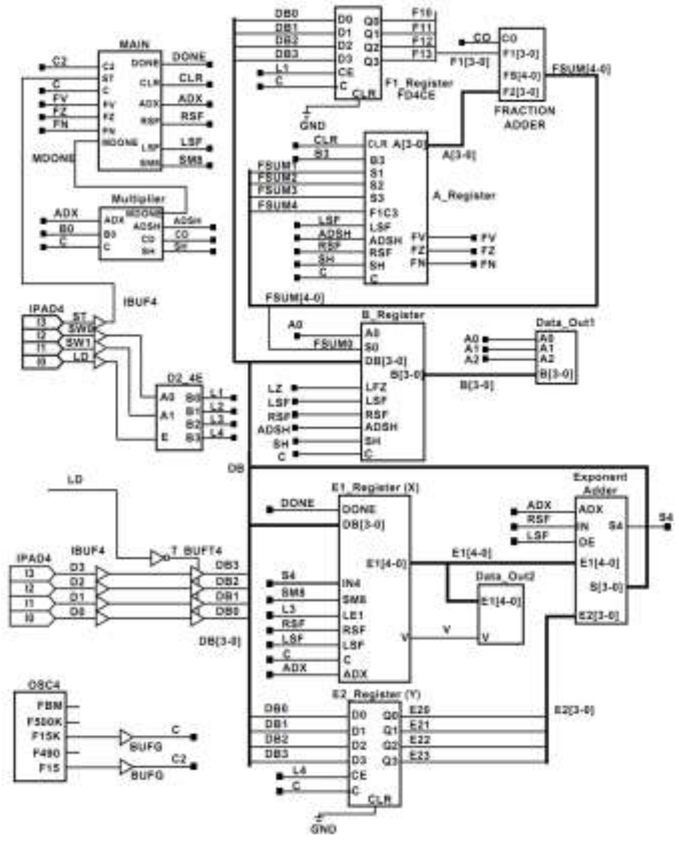


Figure 9. Top-level schematic of the floating point multiplier

Finally, the bus structure for the multiplier is given in figure 10 below.

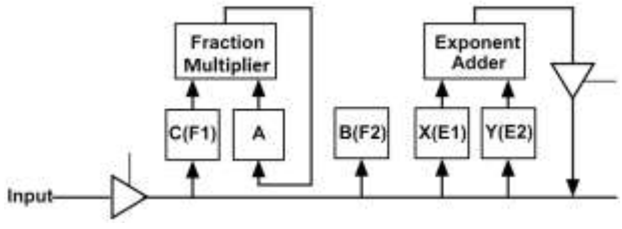
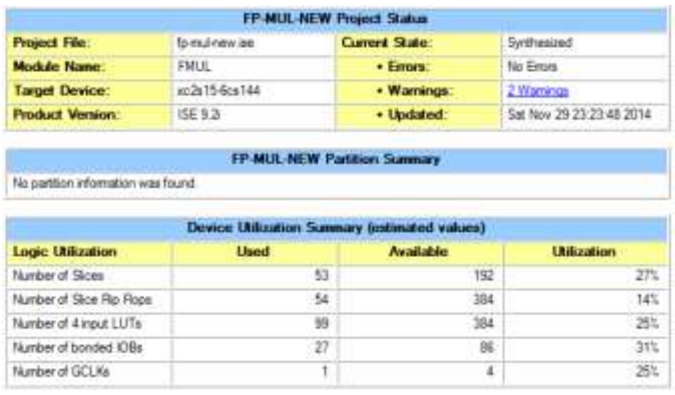


Figure 10. Multiplier bus structure

V. Simulation and Synthesis

The floating point multiplier has been implemented on a Xilinx Spartan 2 FPGA. Simulations were carried out on the Xilinx Integrated Simulation Environment. The simulation waveforms are shown in figure 11. The synthesis results are shown in table 1.

TABLE I. SYNTHESIS RESULTS



FP-MUL-NEW Project Status			
Project File:	fp-mul-new.ise	**Current State:**	Synthesized
Module Name:	FMUL	**• Errors:**	No Errors
Target Device:	xc2s15-6cs144	**• Warnings:**	2 Warnings
Product Version:	ISE 9.2i	**• Updated:**	Sat Nov 29 23:23:48 2014

FP-MUL-NEW Partition Summary
No partition information was found

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slices	53	192	27%
Number of Slice Flip Flops	54	384	14%
Number of 4 input LUTs	99	384	25%
Number of bonded IOBs	27	86	31%
Number of GCLKs	1	4	25%

References

[1] S. Brown and Z. Vranesic, “Fundamentals of Digital Logic with VHDL Design,” Second Edition, McGraw Hill Publishing Company, 2005.

[2] D. M. Harris and S. L. Harris, “Digital Design and Computer Architecture,” Morgan Kaufmann Publishers, 2005.

[3] C. H. Roth, Jr., “Digital Systems Design Using VHDL,” PWS Publishing Company, 1998.

[4] D. A. Patterson and J. L. Hennessey, “Computer Organization and Design – The Hardware Software Interface”, Third Edition, Morgan Kaufmann Publishers, 2005.

[5] U. Meyer-Baese, “Digital Signal Processing with Field Programmable Gate Arrays,” Fourth Edition, Springer-Verlag, 2014

[6] 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), pp. 155-162, 1995.

[7] B. Fagin and C. Renard, “Field Programmable Gate Arrays and Floating Point Arithmetic,” IEEE Transactions on VLSI, Vol. 2, No. 3, pp. 365-367, 1994.

[8] M. Al-Ashrafy, A. Salem, W. Anis, “An Efficient Implementation of a Floating Point Multiplier,” Saudi International Electronics, Communications, and Photonics Conference (SIECPC), pp. 1-5, 25-26, 2011.

.

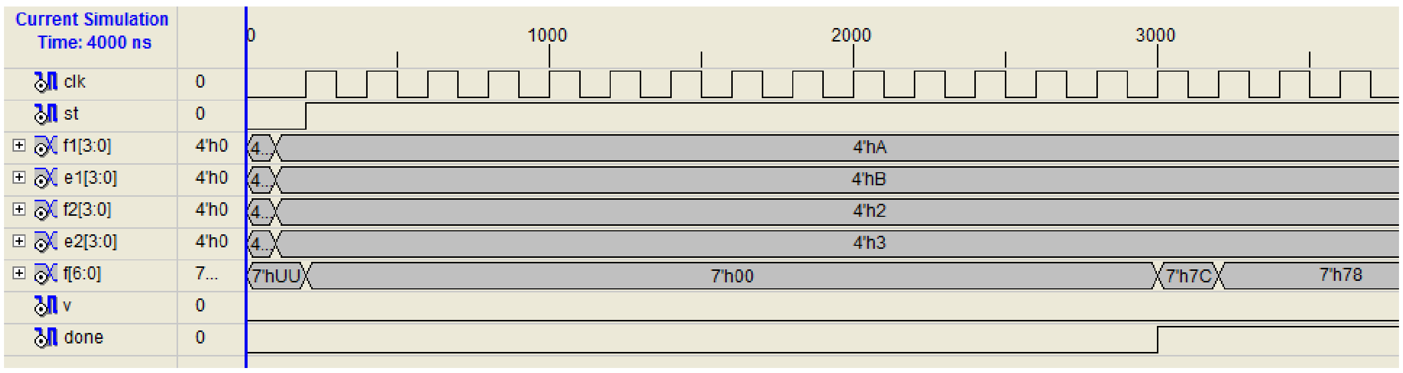


Figure 11. Simulation results of floating point multiplication