www.ijecs.in
International Journal Of Engineering And Computer Science ISSN:2319-7242
Volume 2 Issue 6 June 2013 Page No. 2098-2101

Design and implementation of efficient 32-bit floating point multiplier using Verilog

***Surendra Singh Rajpoot*[1], *Nidhi Maheshwari*[2], *D.S. Yadav*[3]**

[1]Lord Krishna College of technology
Rau-Pithampur By-pass road,
Opp.STI(I)Ltd. Indore(MP), India
errajpoot@email.com
[2] Lord Krishna College of technology
Rau-Pithampur By-pass road,
Opp.STI(I)Ltd. Indore(MP), India
dsyadav@gmail.com
[3] Lord Krishna College of technology
Rau-Pithampur By-pass road,
Opp.STI(I)Ltd. Indore(MP), India
Er.nidhi17@ymail.com

Abstract: ***A Binary multiplier is an integral part of the arithmetic logic unit (ALU) subsystem found in many processors. Floating Point Arithmetic is extensively used in the field of banking, tax calculation, currency conversion, and other financial areas including broadcast, musical instruments, conferencing, and professional audio. Many of these applications need to solve sparse linear systems that use fair amounts of matrix multiplication.***
The objective of this thesis is to design and implement single precision (32-bit) floating-point cores for multiplication. The multiplier conforms to the IEEE 754 standard for single precision. The IEEE Standard for Binary Floating Point Arithmetic (IEEE 754) is the most widely used standard for floating point computation, and is followed by many CPU and FPU implementation. The standard defines formats for representing floating point numbers (including negative zero and denormal numbers) and special values (infinites and NaNs) together with a set of floating point operation that operate on these values. It also specifies four rounding modes and five exceptions.

In this thesis, I have used VERILOG as a HDL and Xilinx ISE has been synthesized on same tool. Timing and correctness properties were verified. Instead of writing Test- Benches & Test-Cases we used Wave-Form Analyzer which can give a better understanding of Signals & variables and also proved a good choice for simulation of design. In order to perform floating point multiplication a VERILOG program is realized. The fixed-point design is extended to support floating-point multiplication by adding several components including exponent generation, rounding, shifting, and exception handling.

Keywords: Binary multiplier, denormal numbers, IEEE 754, NaNs, Xilinx ISE

1. General

Many people consider floating-point arithmetic an esoteric subject. This is rather surprising because floating-point is ubiquitous in computer systems. Almost every language has a floating-point data type; computers from PC's to supercomputers have floating-point accelerators; most compilers will be called upon to compile floating-point algorithms from time to time; and virtually every operating system must respond to floating-point exceptions such as overflow. There are some aspects of floating point that have a direct impact on designers of computer systems. It begins with background on floating-point representation and rounding error, continues with a discussion of the IEEE floating-point standard, and concludes with numerous examples of how computer builders can better support floating-point.[1]

Every computer has a floating point processor or a dedicated accelerator that fulfills the requirements of precision using detailed floating point arithmetic. The main applications of floating points today are in the field of medical imaging, biometrics, motion capture and audio applications, including broadcast, conferencing, musical instruments and professional audio. Their importance can be hardly over emphasized because the performances of computers that handle such applications are measured in terms of the number of floating point operations they perform per second. [2]

2. Introduction

There are several ways to represent real numbers on computers. Fixed point places a radix point somewhere in the middle of the digits, and is equivalent to using integers that represent portions of some unit. For example, one might represent 1/100ths of a unit; if you have four decimal digits, you could represent 10.82, or 00.01. Another approach is to use rational, and represent every number as the ratio of two integers. [18]

Floating-point representation - the most common solution - basically represents reals in scientific notation. Scientific notation represents numbers as a base number and an exponent. For example, 123.456 could be represented as 1.23456×10^2. In hexadecimal, the number 123.abc might be represented as $1.23abc \times 16^2$.

Floating-point solves a number of representation problems. Fixed-point has a fixed window of representation, which limits it from representing very large or very small numbers. Also, fixed-point is prone to a loss of precision when two large numbers are divided.

Floating-point, on the other hand, employs a sort of "sliding window" of precision appropriate to the scale of the number. This allows it to represent numbers from 1,000,000,000,000 to 0.0000000000000001 with ease.

3. IEEE 754 Floating Point Standard

IEEE 754 floating point standard is the most common representation today for real numbers on computers. The IEEE (Institute of Electrical and Electronics Engineers) has produced a Standard to define floating-point representation and arithmetic. Although there are other representations, it is the most common representation used for floating point numbers. The standard brought out by the IEEE come to be known as IEEE 754. The IEEE Standard for Binary Floating-Point Arithmetic (IEEE 754) is the most widely- used standard for floating point computation, and is followed by many CPU and FPU implementations.[1] The standard defines formats for representing floating-point numbers including negative numbers and denormal numbers special values i.e. infinities and NAN's together with a set of floating-point operations that operate on these values. It also specifies four rounding modes which are round to zero, round to nearest, round to infinity and round to even and five exceptions including when the exceptions occur, and what happens when they do occur. Dealing with fixed-point arithmetic will limit the usability of a processor. If operations on numbers with fractions (e.g. 10.2445), very small numbers (e.g. 0.000004), or very large numbers (e.g. 42.243×10^5) are required, then a different one representation is in order is the floating-point arithmetic.[14] The floating point is utilized as the binary point is not fixed, as is the case in integer (fixed-point) arithmetic. In order to get some of the terminology out of the way, let us discuss a simple floating-point number, such as -2.42×10^3. The '-' symbol indicates the sign component of the number, while the '242' indicate the significant digits component of the number, and finally the '3' indicates the scale factor component of the number. It is interesting to note that the string of significant digits is technically termed the *mantissa* of the number, while the scale factor is appropriately called the *exponent* of the number. The general form of the representation is the following:

$$(-1)^S * M * 2^E \qquad (1)$$

Where

S represents the sign bit,
M represents the mantissa and
E represents the exponent

4. FLOATING POINT ARITHMETIC

The IEEE Standard for Binary Floating-Point Arithmetic (IEEE 754) is the most widely- used standard for floating-point computation, and is followed by many CPU and FPU implementations.[1] The standard defines formats for representing floating-point number (including ±zero and denormals) and special values (infinities and NaNs) together with a set of floating-point operations that operate on these values. It also specifies four rounding modes and five exceptions.

IEEE 754 specifies four formats for representing floating-point values: single-precision (32-bit), double-precision (64-bit), single-extended precision (≥ 43-bit, not commonly used) and double-extended precision (≥ 79-bit, usually implemented with 80 bits). Many languages specify that IEEE formats and arithmetic be implemented, although sometimes it is optional. For example, the C programming language, which pre-dated IEEE 754, now allows but does not require IEEE arithmetic (the C floattypically is used for IEEE single-precision and doubleuses IEEE double-precision).[18]

5. Single Precision Floating Point Numbers

The single-precision number is 32 bit wide. The single-precision number has three main fields that are sign, exponent and mantissa. The 24-bit mantissa (the leading one is implicit) can approximately represent a 7-digit decimal number, while an 8-bit exponent to an implied base of 2 provides a scale factor with a reasonable range. Thus, a total of 32 Bit is needed for single-precision number representation. To achieve a bias equal to $2^{n-1} - 1$ is added to the actual exponent in order to obtain the stored exponent. This equals 127 for an eight-bit exponent of the single-precision format. The addition of bias allows the use of an exponent in the range from −127 to +128, corresponding to a range of 0-255 for single precision number. The single-precision format offers a range from 2^{-127} to 2^{+127}, which is equivalent to 10^{-38} to 10^{+38}

Sign: 1-bit wide and used to denote the sign of the number i.e. 0 indicate positive number and 1 represent negative number.
Exponent: 8-bit wide and signed exponent in excess-127 representation.
Mantissa: 23-bit wide and fractional component.

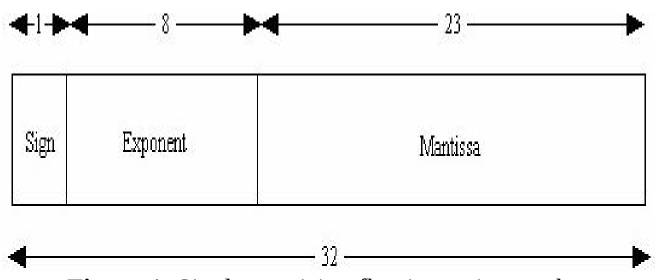

Figure 1: Single-precision floating-point number representation

The excess-127 representation mentioned when discussing the exponent portion above, is utilized to efficiently compare the relative sizes of two floating point numbers. Instead of storing the exponent (E) as a signed number, we store its unsigned integer representation ($E' = E + 127$). This gives us a range for

E' of 0 <= *E'* <= 255. While the 0 and 255 end values are used to represent special numbers (exact 0, infinity and denormal numbers), the operating range of *E0* becomes 1 <= *E'*<=" 254, thus, limiting the range of *E* to -126<= *E* <= 127. In double-precision numbers, an excess-1023 representation is utilized.

6. Ranges Of Floating-Point Numbers

Table 1.1: Effective Range of single precision and double precision float numbers

	Binary	**Decimal**
Single	$\pm (2-2^{-23}) \times 2^{127}$	$\sim \pm 10^{38}$
Double	$\pm (2-2^{-32}) \times 2^{1023}$	$\sim \pm 10^{308}$

7. Floating Point Multiplication

Given two FP numbers n1 and n2, the product of both, denoted as n, can be expressed as:

$$n = n1 \times n2$$
$$= (-1)^{S1}.\ p1.\ 2^{E1} \times (-1)^{S2}.\ p2.2^{E2}$$
$$\mathbf{= (-1)^{S1+S2}.\ (p1.p2).\ 2^{E1+E2}} \qquad \textbf{(2)}$$

This means that the result of multiplying two FP numbers can be described as multiplying their significands and adding their exponents.[14] The resultant sign S is S1 +S2, the resultant significand p is the adjusted product of p1. p2 and the resultant exponent E is the adjusted E1+E2+bias. In order to perform floating-point multiplication, a simple algorithm is realized:

- Add the exponents and subtract 127.
- Multiply the mantissas and determine the sign of the result.
- Normalize the resulting value, if necessary.

8. Number Representation Using Single Precision Format

Let us try and represent the decimal number $(-0.75)_{10}$ in IEEE floating-point format. First of all, we notice that $(-0.75)_{10} = (-3/4)_{10}$.

In binary notation, we have $(-0.11)_2 = (-0.11)_2 \times 2^0 = (-1.1)_2 \times 2^{-1}$ Referring to equation (2), we can represent our number as: $(-1)^1 * (1 + .10000000000000000000000_2) * 2^{126-127}$ Thus, our single-precision representation of the number is given as $(10111111010000000000000000000000)_2$,where The sign bit is $(1)_2$,for negative numbers; The exponent is $(01111110)_2$, to represent $(126)_{10}$, The mantissa is $(10000000000000000000000)_2$, to represent the fractional part $(.1)_2$.

9. Conclusion

Single precision floating point multiplier is designed and implemented using xilinx in this paper. The designed multiplier conforms to IEEE 754 single precision floating point standard. In this implementation exceptions (like invalid, inexact, infinity, etc) are considered. In this implementation rounding modes like round to positive infinity, round to negative infinity, round to zero and round to even. The designed is verified using fpu_test test bench. The design is also verified for overflow and underflow cases.

References

1. IEEE standard for binary-floating point arithmetic, ANSI/IEEE Std 754-1985, The Institute of Electrical and Electronic Engineers Inc., New York, August 1985.
2. David Goldberg: What Every Computer Scientist Should Know About Floating- Point Arithmetic, 1991.
3. I. Koren, Computer Arithmetic Algorithms, Second Edition, prentice Hall, 2002.
4. An ANSI/ IEEE Standard for Radix-Independent Floating-Point Arithmetic, Technical Committee on microprocessor of IEEE computer society, October, 1987.
5. Steve Hollasch, IEEE Standard 754 Floating Point Numbers, February 2005.
6. BROWN, Stephen D. Fundamentals of Digital Logic with VHDL design. Boston: McGraw-Hill, 2000.
7. John L Hennesy & David A. Patterson 'Computer Architecture A Quantitative Approach' Second edition; A Harcourt Publishers International Company
8. J. Bhasker, ***A VHDL Primer***, Third Edition, Pearson, 1999.
9. M. Ercegovac and T. Lang, Digital Arithmetic, Morgan Kaufmann Publishers, 2004.
10. John. P. Hayes, 'Computer Architecture and Organization', McGraw Hill, 1998.
11. Peter J . Ashenden, *The Designer's Guide to VHDL*, Morgan Kaufmann Publishers, 95 Inc. , 1996.
12. Prof. W. Kahan, Lecture Notes on the Status of IEEE Standard 754 for Binary Floating-Point Arithmetic. Link: www.cs.berkeley.edu/~wkahan/ieee754status.html
13. Wikipedia, the free encyclopedia, IEEE 754-1985. Link: http://en.wikipedia.org/wiki/floating _point.html
14. Behrooz Parhami, Computer Arithmetic, Algorithms and Hardware Design Oxford University Press.2000.
15. IEEE Floating Point Representation of Real Number, Fundamentals of Computer Science. Link: http://www.math.grin.edu/~stone/courses/fundamentals/ieee- reals.html
16. M. J. Flynn and S. F. Oberman, Advanced Computer Arithmetic Design, John Wiley and Sons, 2001.
17. N. Weste, D. Harris, CMOS VLSI Design, Third Edition, Addison Wesley, 2004.

18. Beebe, H.F.Nelson, Floating Point Arithmetic, Computation in Modern Science & Technology, December, 2007.
19. P. Karlstrom, A. Ehliar, High Performance Low Latency Floating Point Multiplier, November 2006
20. www.ieee.org

Author Profile

Surendra Singh Singh received the B.E. degree in Electronics and communication Engineering from Ujjain engineering college Ujjain in 2008 and persuing m.tech degree in session (2011-2013) from Lord Krishna college of Technology. This is my research paper.