Int. J. Adv. Res. Sci. Technol. Volume 5, Issue 1, 2016, pp.516-519.

IJARST

International Journal of Advanced Research in Science and Technology

journal homepage: www.ijarst.com

ISSN 2319 – 1783 (Print)

ISSN 2320 – 1126 (Online)

Implementation Of 32-Bit Floating Point Arithmetic Unit By Using Verilog.

K. V. Umma Maheshwara Rao*, P G Naveen, K. Usha Rani, G. Amani** and **A. M. Subhan.

Department of ECE, VITS College of Engineering, Visakhapatnam, Andhra Pradesh, India.

****Corresponding Author's E-mail: umeshkvuster@gmail.com***

A R T I C L E I N F O

Article history:

Received 09 Jan. 2015

Accepted 20 Feb. 2016

Available online 22 Feb. 2016

Keywords:

Addition,
Subtraction,
Multiplication,
Division.

A B S T R A C T

Arithmetic circuits form an important class of circuits in digital systems with the remarkable progress in the very large scale integration (VLSI) circuit technology, many complex circuits, unthinkable yesterday have become easily realizable today. Algorithms that seemed impossible to implement now have attractive implementation possibilities for the future. This means that not only the conventional computer arithmetic methods, but also the unconventional ones are worth investigation in new designs. In this thesis an ARITHMETIC UNIT based on IEEE standard for floating point numbers has been implemented on Spartan3E FPGA Board. The arithmetic unit implemented has a 32-bit processing unit which allows various arithmetic operations such as, Addition, Subtraction, Multiplication, Division and Square Root on floating point numbers.Each operation can be selected by a particular operation code. Simulation of FLOATING POINT ALU has been done by XILINX-ISE.

PAPER-QR CODE

IJARST Volume 5, Issue 1

Citation: *Umma Maheshwara Rao. et.al, Implementation Of 32-Bit Floating Point Arithmetic Unit By Using Verilog, Int. J. Adv. Res. Sci. Technol. Volume 5, Issue 1, 2016, pp.516-519.*

Introduction:

In computing a fixed-point number representation is a real data type for a number that has a fixed number of digits after (and sometimes also before) the radix point (e.g., after the decimal point '.' in English decimal notation).

Floating-point representations are easier to use than fixed-point representations, because they can handle a wider dynamic range and do not require programmers to specify the number of digits after the radix point.

Theory of floating point numbers:

Floating-point arithmetic is considered an esoteric (understood by only a particular group) subject by many people. This is rather surprising because floating-point is ubiquitous (existing) in computer systems. Almost every language has a floating-point data type; computers from PCs 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.

Due to rapid growth in financial, commercial, and Internet-based applications, there is an increasing desire to allow computers to operate on both binary and decimal floating-point numbers. Consequently, specifications for decimal floating-point support are being added to the IEEE-754 Standard for Floating-Point Arithmetic.

Essentially floating point numbers are used to represent very small to very large numbers with varying levels of precision. The normal set of mathematical operations such as addition, division, multiplication and subtraction are defined, but are slightly more complex than those for standard integer numbers. The speedup of these operations is quite important, because floating point numbers are used in a wide range of applications including CAD, games, graphics, multimedia, and scientific.

IEEE 754 Floating Point:

An IEEE 754 single-precision floating point value is 32 bits long. The bits are divided into fixed-sized fields as follows:

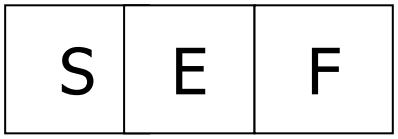

Fig. 1: Format of an IEEE 754 Floating Point Number

Bits 0 through 22 are for the mantissa; bits 23 through 30 are for the exponent; and bit 31 is the sign bit. The mantissa and exponent fields work like the similar parts in scientific notation (details follow). The sign bit gives the sign of the entire expression: a 0 bit means positive and a 1 bit means negative.

Storage layout:

IEEE floating point numbers have three basic components: the sign, the exponent, and the mantissa. The mantissa is composed of the fraction and an implicit leading digit (explained below). The exponent base (2) is implicit and need not be stored.

FLOAT Formula:

Here is a formula that summarizes the past several pages. In it, s is the sign bit (0 or 1), M is the mantissa (000...000 to 111...111) and E is the biased exponent. To convert a paper-and-pencil number into IEEE floating point, fill in each piece of the picture. Let us represent 1.0 as a 32-bit IEEE 754 floats.

Special Values:

The most significant bit of the significant (not stored) is determined by the value of biased exponent. If 0 < exponent < $2^e - 1$, the most significant bit of the significant is 1, and the number is said to be normalized. If exponent is 0 and fraction is not 0, the most significant bit of the significant is 0 and the number is said to be de-normalized. Three other special cases arise:

If exponent is 0 and fraction is 0, the number is ±0 (depending on the sign bit)

If exponent = $2^e - 1$ and fraction is 0, the number is ±infinity (again depending on the sign bit), and

If exponent = $2^e - 1$ and fraction is not 0, the number being represented is not a number (NaN).

Single-Precision 32-Bit:

A single-precision binary floating-point number is stored in 32 bits. Bit values for the IEEE 754 32bit float 0.15625, $(1+1/4)*2^{(124-127)}$. The exponent is biased by $2^{8-1} - 1 = 127$ in this case (Exponents in the range −126 to +127 are representable. See the above explanation to understand why biasing is done). An exponent of −127 would be biased to the value 0 but this is reserved to encode that the value is a de-normalized number or zero. An exponent of 128 would be biased to the value 255 but this is reserved to encode infinity or not a number (NaN). See the chart above.

For normalized numbers, which are the most common, the exponent is the biased exponent and fraction is the significant without the most significant bit.

Double-Precision 64 Bit:

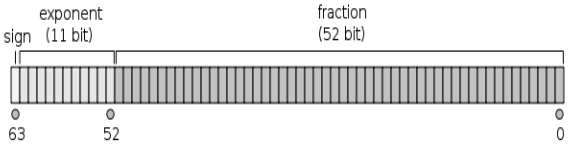

Fig. 2: IEEE 754 Double Precision Format

The three fields in a 64bit IEEE 754 float. Double is essentially the same except that the fields are wider. The fraction part is much larger, while the exponent is only slightly larger. NaNs and Infinities are represented with Exp being all 1s (2047). If the fraction part is all zero then it is Infinity, else it is NaN.

For Normalized numbers the exponent bias is +1023 (so e is exponent − 1023). For Denormalized numbers the exponent is (−1022) (the minimum exponent for a normalized number—it is not (−1023) because normalized numbers have a leading 1 digit before the binary point and denormalized numbers do not). As before, both infinity and zero are signed.

Floating-Point Algorithms:

The following sections look at algorithms defined by the IEEE 754 standard for conversion and base arithmetic calculations, and which are implemented in the FPU.

Calculations: The FPU supports the following calculations involving two floating-point values addition, subtraction, multiplication and division

Floating Point Addition:

Next we consider the design of an adder for floating point numbers. Two floating point will be added to form a floating point sum:

$$(F_1 \text{ X } 2^{E}{}_1) + (F_2 \text{ X } 2^{E}{}_2) = F \text{ X } 2^{E}$$

Again we will assume that the numbers to be added are properly normalized and that the answer should be put in normalized form. In order to add two fractions, the associated exponents must be equal. Thus, if the exponents E_1 and E_2 are different, we must un-normalize one of the fractions and adjust the exponent accordingly. The smaller number is the one that should be adjusted so that if significant digits are lost, the effect is not significant. To illustrate the process, we add

$$F_1 \text{ X } 2^{E}{}_1 = 0.111 \text{ X } 2^5$$
$$F_2 \text{ X} 2^{E}{}_2 = O.101 \text{ X } 2^3$$

Since $E_2 \neq E_1$, we un-normalize the smaller number F_2 by shifting right two times and adding 2 to the exponent:

$$0.101 \text{ x } 2^3 = 0.0101 \text{ x } 2^4 = 0.00101 \text{ x } 2^5$$

Note that shifting right one place is equivalent to dividing by 2, so each time we shift we must add 1 to

the exponent to compensate. When the exponents are equal, we add the fractions:

$$0.111 \times 2^5 + 0.00101 \times 2^5 = 01.00001 \times 2^5$$

The addition caused an overflow into the sign bit position, so we shift right and add 1 to the exponent to correct the fraction overflow. The final result is

$$F \times 2^E = 0.100001 \times 2^6$$

When one of the fractions is negative, the result of adding fractions may be normalized, as illustrated in the following example:

$$1.100 \times 2^{-2} + 0.100 \times 2^{-1} = 1.110 \times 2^{-1} + 0.1..$$

$$\times 2^{-1} \text{(after shifting } F_1) = 0.010 \times 2^{-1} = 0.100 \times 2^{-2}$$

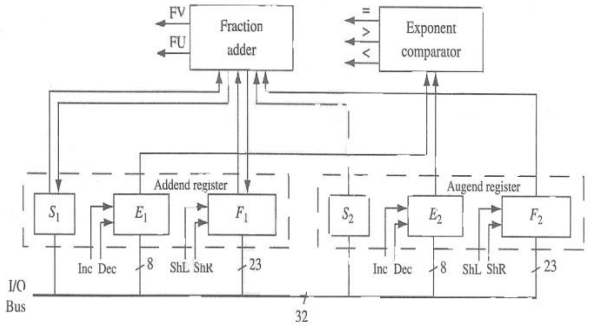

Fig. 3: Overveiw of Floaing Point Adder

Floating-Point Multiplication:

Given two floating point numbers,$(F_1 \times 2^{E}_1)$ and $(F_2 \times 2^{E}_2)$, the product is$(F_1X2^{E1}) X (F_2X2^{E2}) = (F_1XF_2) X 2^{(E1+E2)} = F X 2^{E}$.The fraction part of the product is the product of the fraction, and the exponent part of the product is the sum of the exponents. Hence, a floating point consists of two major components: a fraction multiplier, and an exponent adder. The details of floating point multiplication will depend on the precise formats in which the fraction multiplication and exponent addition are performed.

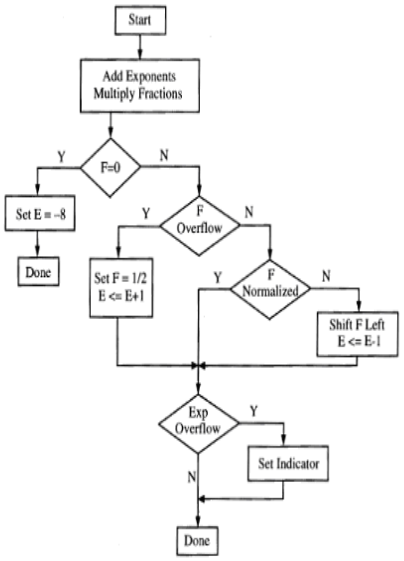

Fig. 4: Flow Chart for Floating Point Multiplication

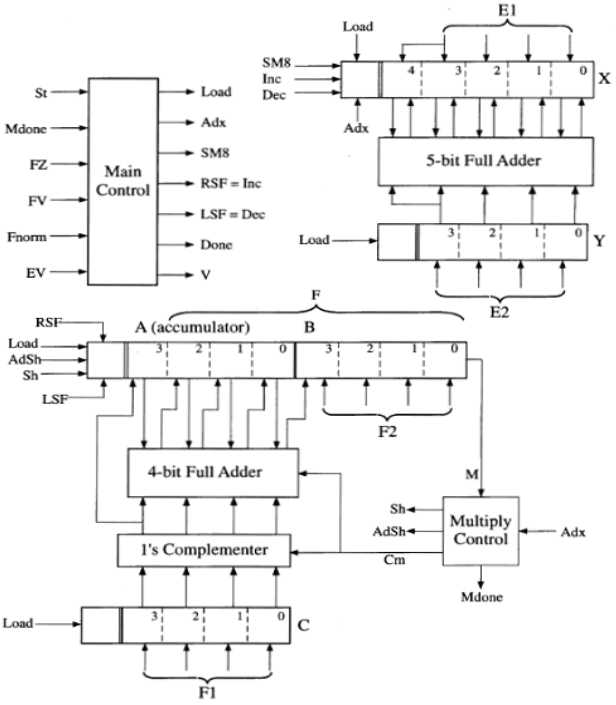

Fig. 5: Exponent Adder and Fraction Multiplier

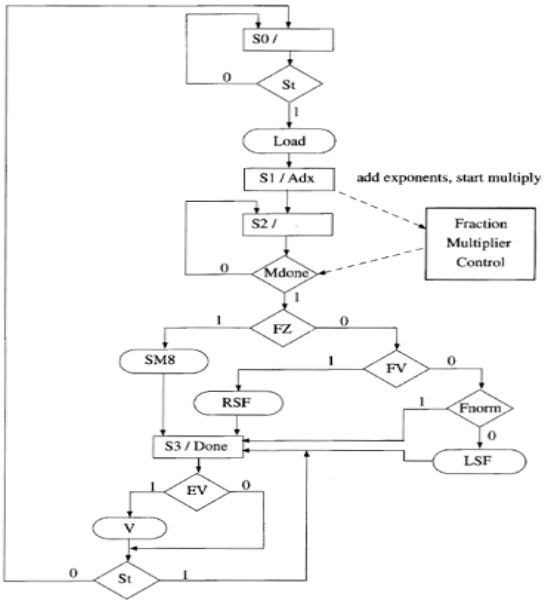

Fig. 5: SM chart for Floating point multiplication

Other floating-point operations:

Subtraction:

Subtraction is similar to addition. It involves making the exponents of both operands equal and then subtracting the resulting significands. In order to make things less complicated, care is taken to ensure that the smaller operand is always subtracted from the larger one.

Floating point subtraction is achieved simply by inverting the sign bit and performing addition of signed mantissas as outlined above. Like addition as far as alignment of radix points then the algorithm for subtraction of sign mag. numbers takes over. Before

subtracting, compare magnitudes. Change sign bit if order of operands is changed. Don't forget to normalize number afterward.

Division:

The quotient of two floating-point numbers is

$$(F_1 \text{ X } 2^{E}_{1}) \ (F_2 \text{ X } 2^{E}_{2}) = (F_1/F_2) \text{ X } 2^{(E_1 - E_2)} = F \text{ X } 2^{E}$$

Thus, the basic rule for floating point division is to divide the fractions and subtract the exponents. In addition to considering the same special cases as for multiplication, we must test for divide by 0before dividing. If F_1 and F_2 are normalized, then the largest positive quotient (F) will be

$0.1111\ldots/0.1000\ldots = 01.111\ldots$

Which is less than 10_2, so the fraction overflow is easily corrected? For example,

$(0.110101\text{x } 2^{2})\div(0.101 \text{ x } 2^{-3}) = 01.010 \text{ x } 2^{5} = 0.101 \text{ x } 2^{6}$

Alternatively, if $F_1 \geq F_2$, we can shift F_1 right before is to dividing and avoid fraction overflow in the first place. In the IEEE format, when divide by 0 is involved, the result can be set to NAN (Not a number).

Observations:

Addition of two floating point numbers:

I added 2 floating point numbers.

They are 3.5 + 3.5

3.5 is represented as

0 10000000 11000000000000000000000.

And the output is 7.

7 is represented as

0 10000001 11000000000000000000000

0 represents sign so this is positive number

10000001= 128+1=129

11000000000000000000000=0.5+0.25=0.75 2^ (129-127)x1+ (0.5 + 0.25) = 2^ (2) x 1.75=4 x 1.75 = 7

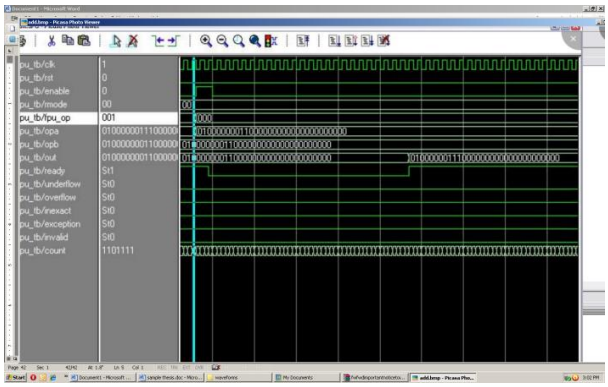

Fig. 6: Simulation Results of Addition

Similarly simulation results occur for:

Subtraction, Multiplication, Division.

Conclusion:

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 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.

In this project we developed a VERILOG code for floating point addition, subtraction, multiplication and division using XILINX tools and simulated the results with the use of MODELSIM simulator.

References:

1. Principles of digital systems design using vhdl by Roth and john
2. Tanenbaum, Andrew S., Structured Computer Organization, Third Edition, Prentice-Hall, 1990.
3. *ANSI/IEEE Standard 754-1985, Standard for Binary Floating Point Arithmetic.*
4. Computer Organization and Architecture, William Stallings, pp. 222-234 Macmillan Publishing Company, ISBN 0-02-415480-6
5. IEEE Computer Society (1985), IEEE Standard for Binary Floating-Point Arithmetic, IEEE Std 754-1985.
6. Intel Architecture Software Developer's Manual,
7. Volume 1: Basic Architecture, (a PDF document downloaded from intel.com.)