Floating Point Unit (FPU) for FPGA

Nikhil S. S.[1] Sheela Devi Aswathy Chandran[2]
[1]PG student [2]Assistant professor,
[1, 2,] Department of Electronics and Communication engineering,
[1, 2]TKM Institute of Technology, Karuvelil P.O, Kollam, Kerala-691505, India.

Abstract---Floating point describes a method of representing an approximation of a real number in a way that can support a wide range of values. The numbers are in general, represented approximately to a fixed number of significant digits (the mantissa) and scaled using an exponent. The base for the scaling is normally 2, 10 or 16. Floating point arithmetic and logic unit require many algorithms. Then again, many scientific applications require floating point arithmetic because of high accuracy in their calculations. FPGAs are becoming more suitable for supporting high speed floating point arithmetic units. Floating point units are widely used in digital applications such as digital signal processing, digital image processing and multimedia. Many algorithms depend on floating point arithmetic because floating point representation supports huge range. So many algorithms are used to define the arithmetic operations such as Addition, Subtraction, Multiplication and Division. In top-down design approach, four arithmetic modules, addition, subtraction, multiplication and division are combined to form a floating point arithmetic unit and the logical operations such as logic gates and shifting operations are combined together to form logic unit. For the synthesis and simulation Xilinx 13.2 is required.

I. Introduction

Floating point describes a method of representing an approximation of a real number in a Way that can support a wide range of values. The numbers are in general, represented approximately to a fixed number of significant digits (the mantissa) and scaled using an exponent. The base for the scaling is normally 2, 10 or 16. Floating point operations are very complex operations because they require many algorithms. By using Field Programmable Gate Arrays (FPGAs) the designers can build any logic device in hardware quickly and easily. Floating point arithmetic and logic unit require many algorithms. Consequently, FPGAs are becoming more suitable for supporting high speed floating point arithmetic units. Floating point units are widely used in digital applications such as digital signal processing, digital image processing and multimedia. Digital arithmetic operations are very important in the design of digital processors and application specific systems. 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 Conventional floating point units, the most frequently used floating point operations are multiplication and addition/subtraction counting for more than 94% of all floating point instructions. Hence the employment of highly performing divider, multiplier, adder and subtractor modules is of high importance.

Floating point addition is the most complex operation in floating point arithmetic and consists of many variable latency and area dependent sub-operations. In floating point addition implementations, latency is the primary performance bottleneck. Much work has been done to improve the overall latency of floating point adders. Various algorithms and design approaches have been developed by the VLSI community in the last two decades.

For the most part, digital design companies around the globe have focused on FPGA design instead of ASICs because of their effective time to market, adaptability and most importantly, low cost. The floating point unit is one of the most important custom applications needed in most hardware designs, as it adds accuracy, robustness to quantization errors and ease of use. The Figure.1 Shows the Floating point ALU Architecture.

II. Floating Point Arithmetic

Many applications require numbers that aren’t integers. There are a number of ways that non-integers can be represented. Adding two such numbers can be done with an integer add, whereas multiplication requires some extra shifting. There is various ways to represent the number systems. However, only one non-integer representation has gained widespread use, and that is floating point.

The floating point Arithmetic and Logic Unit (ALU) consist up of Arithmetic unit and Logic unit. The arithmetic operation consists up of Addition, Subtraction, Multiplication and Division. The Logic operation consists up of all logic gates and shifting operations.

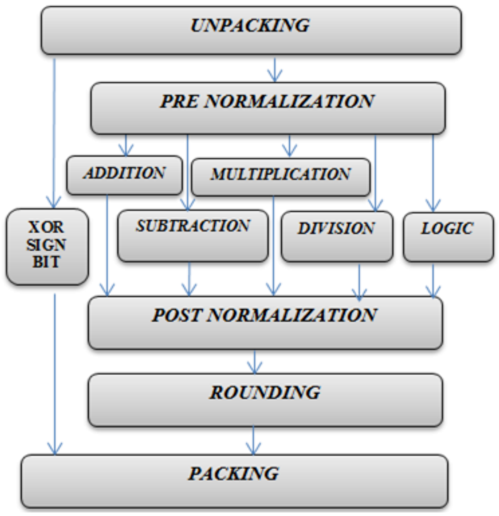

Fig.1: Floating point ALU Architecture

A. *Unpacking*

The inputs given are 32 bit wide and they are floating point numbers. So the Sign bit Exponent bits and Mantissa bits are separated. By the unpacking process the bits are separated as Sign bit, Exponent bits and Mantissa bits. The unpacking function is the first process and after the separation of the bits they are used to perform various functions.

B. *Normalization*

The internal representation of a floating point number can be characterized in terms of the following parameters; the sign is either -1 or 1, the base or radix for exponentiation, an integer greater than 1. This is a constant for a particular representation, the exponent to which the base is raised. The upper and lower bounds of the exponent value are constants for a particular representation. For example, the number 123456.0 could be expressed in exponential notation as 1.23456e+05, a shorthand notation indicating that the mantissa 1.23456 is multiplied by the base 10 raised to power 5. The mantissa or significant is an unsigned integer which is a part of each floating point number. If the base of the representation is b, then the precision is the number of base b digits in the mantissa. This is a constant for a particular representation.

Many floating point representations have an implicit hidden bit in the mantissa. This is a bit which is present virtually in the mantissa, but not stored in memory because its value is always 1 in a normalized number. The mantissa of a floating point number represents an implicit fraction whose denominator is the base raised to the power of the precision. Since the largest representable mantissa is one less than this denominator, the value of the fraction is always strictly less than 1. The mathematical value of a floating point number is then the product of this fraction, the sign, and the base raised to the exponent.

The floating point number is normalized if the fraction is at least 1/b, where b is the base. In other words, the mantissa would be too large to fit if it were multiplied by the base. Non-normalized numbers are sometimes called denormal they contain less precision than the representation normally can hold. If the number is not normalized, then you can subtract 1 from the exponent while multiplying the mantissa by the base, and get another floating point number with the same value. Normalization consists of doing this repeatedly until the number is normalized. Two distinct normalized floating point numbers cannot be equal in value. (There is an exception to this rule: if the mantissa is zero, it is considered normalized. Another exception happens on certain machines where the exponent is as small as the representation can hold. Then it is impossible to subtract 1 from the exponent, so a number may be normalized even if its fraction is less than 1/b.)

In the normalization process two types of normalization procedure are done ie,

i. Pre-Normalization
ii. Post-Normalization

1) *Pre-Normalization*

In the pre-Normalization process, the procedure is done separately for addition/subtraction and multiplication/ division.

Pre-Normalize Block for Add/Subtract - Calculate the difference between the smaller and larger exponent. Adjust the smaller fraction by right shifting it, determine if the operation is add or subtract after resolving the sign bits. Check for NaNs on inputs.

Pre-Normalize Block for Mul/Div - Computes the sum/difference of exponents, checks for exponent overflow, underflow condition and INF value on an input.

2) *Post-Normalization*

Normalize the fractional part and the exponent part. Also do the rounding function in parallel and then pick the output corresponding to the chosen rounding mode.

III. FLOATING POINT ADDITION

The conventional floating-point addition algorithm consists of five stages exponent difference, pre alignment, addition, normalization and rounding. Given floating point numbers X_1 & X_2, $X_1 = (s_1, e_1, f_1)$ and $X2 = (s_2, e_2, f_2)$, X1+X2, the stages for computing are described as follows.

1) Find exponent difference $d = e_1 - e_2$. If $e_1 < e_2$, swap position of mantissas. Set larger exponent as tentative exponent of result.
2) Pre-align mantissas by shifting smaller mantissa right by d bits.
3) Add mantissas to get tentative result for mantissa.
4) 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.
5) Round mantissa result: If it overflows due to rounding, shift right and increment exponent by 1 bit.

Fig.1 shows the data path for a floating point addition. Only the main parts of the data path are shown for clarity.

The pre alignment and the normalization stages require large shifters. The prealignment stage requires a right shifter that is twice the number of mantissa bits (i.e., 48 bits for single-precision, 106 bits for double-precision) because the bits shifted out have to be maintained to generate the guard, round and sticky bits needed for rounding.

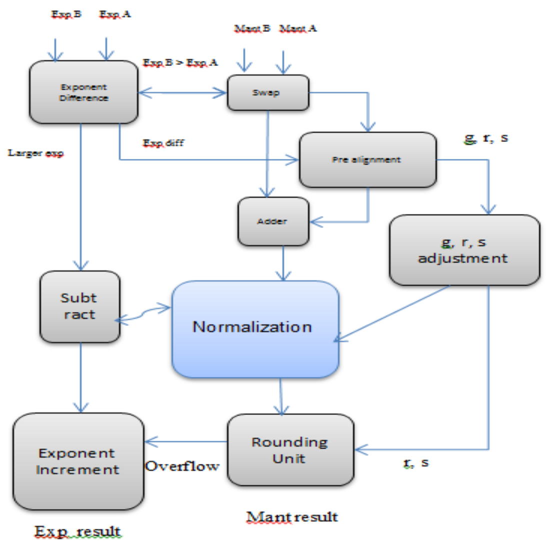

Fig. 2: Data path for Floating point Addition and Subtraction

The shifter only needs to shift right by up to 24 places for single precision or 53 places for double precision. The normalization stage requires a left shifter equal to the number of mantissa bits plus 1 (to shift in the guard bit), i.e. 25 bits for single precision and 54 bits for double precision. The shift amount is determined by the leading one detector (LOD) circuit, which outputs the number of leading zeros before the first one in the bit string. The final stage of the floating-point adder is the rounding unit. It makes a rounding decision based on the rounding mode, the LSB of the mantissa, the round bit and the sticky bit.

IV. FLOATING POINT SUBTRACTION

The conventional floating point addition algorithm consists of five stages exponent difference, pre-alignment, addition, normalization and rounding. Given floating-point numbers $X_1 = (s_1, e_1, f_1)$ and $X_2 = (s_2, e_2, f_2)$, the stages for computing X1-X2 are described as follows.

1) Find exponent difference $d = e_1 - e_2$. If $e_1 < e_2$, swap position of mantissas. Set larger exponent as tentative exponent of result.
2) Pre-align mantissas by shifting smaller mantissa right by d bits.
3) Subtract mantissas to get tentative result for mantissa.
4) 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.
5) Round mantissa result: If it overflows due to rounding, shift right and increment exponent by 1 bit.

V. FLOATING POINT MULTIPLICATION

Algorithmically, floating point multiplication is much simpler than floating point addition. However, a very wide integer multiplier is required. Given floating point numbers X_1 & X_2, $X_1 = (s_1, e_1, f_1)$ and $X_2 = (s_2, e_2, f_2)$, X1 × X2, can be computed using

S= S1 XOR S2

$e = e_1+e_2$-bias

$f= f_1 \times f_2$

The floating point multiplication algorithm is as follows

1) Multiply the mantissas, M1 x M2.
2) Placing the decimal point in the result.
3) Adding the exponent.
4) Obtaining the sign bit, S1 XOR S2.
5) Normalizing the result.
6) Rounding the result.
7) Check underflow or overflow.

Figure.3 shows the data path for a floating point multiplier. Only the main parts of the data path are shown for clarity. If the result from the multiplier has two bits left of the binary point, the mantissa has to be shifted right to compensate and the exponent is incremented. If the rounding of the mantissa results in an overflow, the mantissa is shifted right by one and the exponent is incremented. Equation calls for a very wide multiplier 53 x 53 bit unsigned multiplier for double-precision and 24 x 24 bit for single precision.

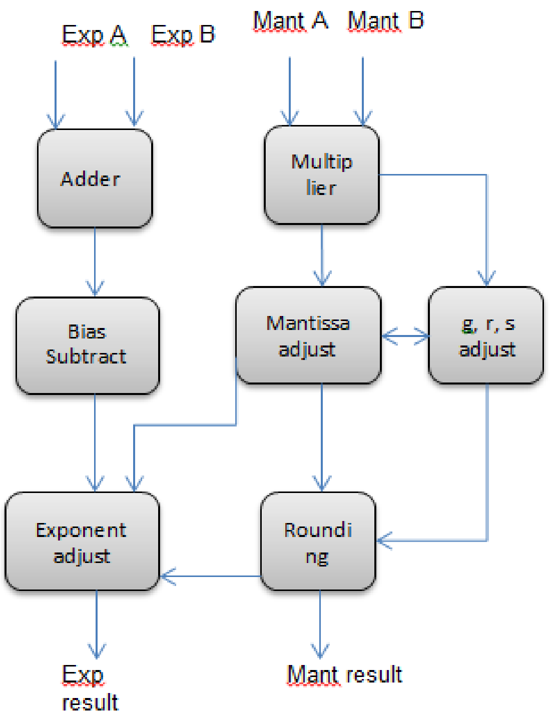

Fig.3: Data path for Floating point Multiplication

A. *Multiplier architecture*

The multiplier used here is Vedic multiplier. The main advantage of the Vedic multiplication algorithm (Urdhva-Tiryakbyam Sutra) stems from the fact that it can be easily realized in hardware. The hardware realization of a 32-bit multiplier using this Sutra is shown in Figure.4; the hardware design is very similar to that of the famous array multiplier where an array of adders is required to arrive at the final product. All the partial products are calculated in parallel and the delay associated is mainly the time taken by the carry to propagate through the adders which form the multiplication array.

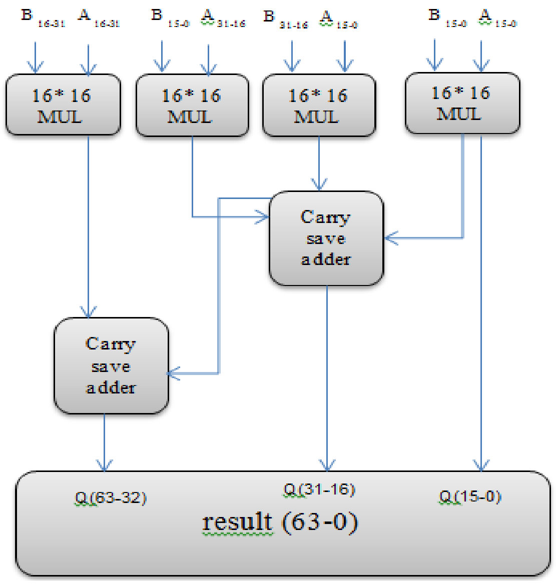

Fig.4: 32-bit Vedic multiplier

VI. Floating Point Division

Algorithmically, floating point multiplication is much simpler than floating point addition. The floating point division algorithm is similar to that of the floating point multiplier algorithm.

Given floating point numbers X_1 & X_2, $X_1 = (s_1, e_1, f_1)$ and $X_2 = (s_2, e_2, f_2)$, X1 / X2, can be computed using
S= S1 XOR S2

$e = e_1+e_2$-bias

$f= f_1 / f_2$

The floating point division algorithm is as follows

1) Divide the mantissas,
   M1 / M2
2) Placing the decimal point in the result.
3) Subtracting the exponent.
4) Obtaining the sign bit, S1 XOR S2.
5) Normalizing the result.
6) Rounding the result.
7) Check underflow or overflow.

VII. Logic Operations

Digital logic is a methodology for dealing with expressions and state tables containing discrete (usually two-state) variables, in this sense the term is synonymous with Boolean algebra. The term is also applied to the hardware components and circuits in which such expressions and tables are implemented. The logic operations done here are,

1. Logic gates
2. Shifting operations

A. *Logic Gates*

Logic gates perform basic logical functions and are the fundamental building blocks of digital integrated circuits. Most logic gates take an input of two binary values, and output a single value of a 1 or 0. Some circuits may have only a few logic gates, while others, such as microprocessors, may have millions of them. There are seven different types of logic gates.

In the following examples, each logic gate except the NOT gate has two inputs, A and B, which can either be 1 (True) or 0 (False). The resulting output is a single value of 1 if the result is true or 0 if the result is false.

i) AND - True if A and B are both True
ii) OR - True if either A or B are True
iii) NOT - Inverts value: True if input is False; False if input is True
iv) XOR - True if either A or B are True, but False if both are True
v) NAND - AND followed by NOT: False only if A and B are both True
vi) NOR - OR followed by NOT: True only if A and B are both False
vii) XNOR - XOR followed by NOT: True if A and B are both True or both False

By combining thousands or millions of logic gates, it is possible to perform highly complex operations. The maximum number of logic gates on an integrated circuit is determined by the size of the chip divided by the size of the logic gates. Since transistors make up most of the logic gates in computer processors, smaller transistors mean more complex and faster processors.

B. *Shifting Operations*

The shift operations allow bits to be moved to the left or right in a word. There are three types of shift operations: logical, rotate and arithmetic. A logical shift moves bits to the left or right. The bits which fall off the end of the word are discarded and the word is filled with 0's from the opposite end. A logical right shift of the 8 bit binary number 1000 1011 gives 0100 0101, as shown below:

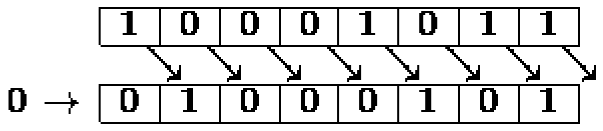

Fig.5: Right shift

Shift instructions include a repeat value, which is the number of times the single bit shift operation is repeated.

A *rotate* operation is a circular shift in which no bits are discarded. A rotate right of the 8 bit binary number 1000 1011 gives 1100 0101, as shown below.

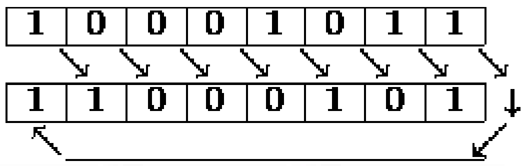

Fig.6: Rotation shift

A right rotation by n bits of an n bit word returns the original word unchanged. A right rotation by n-1 bits is equivalent to a left rotation of 1 bit. The left rotation instruction is redundant because a left rotation of j bits is equivalent to a right rotation of n-j bits. On positive integers, a logical left shift is equivalent to multiplication by 2 and a logical right shift is equivalent to division by 2. The *arithmetic shift* extends this operation to negative 2's complement integers.

An arithmetic right shift is similar to a logical right shift, except that the leftmost bits are filled with the sign bit of the original number instead of 0's.The shifting can be done in single bit shift and multiple bits shift.

VIII. Conclusion

This paper presents a single precision floating point arithmetic and logic unit. The arithmetic operations include addition, subtraction, multiplication and division. The multiplier used here is Vedic multiplier, which reduces the number of partial products. The logic operations are also done here. The future scopes of this project are to implement the proposed floating point arithmetic unit using Field-Programmable Gate Arrays (FPGAs).

References

[1] Yedukondala Rao Veeranki, R. Nakkeeran, “Spartan 3E Synthesizable FPGA Based Floating-Point Arithmetic Unit”, International Journal of Computer Trends and Technology (IJCTT) - volume4, Issue4 –April 2013.
[2] Yee Jern Chong and Sri Parameswaram, “Configurable Multimode Embedded units floating-point for FPGAs”,

IEEE Transactions on VLSI systems, pp. 2033-2044, Vol.19, No.11, November 2011.

[3] Rajit Ram Singh, Asish Tiwari, Vinay Kumar Singh, Geetam "VHDL environment for floating point Arithmetic Logic Unit - ALU design and simulation" International Conference on Communication Systems and Network Technologies, June 2011.

[4] C. C. M. K. Jaiswal and R. Cheung, "High Performance FPGA Implementation of Double Precision Floating Point Adder/Subtractor", International Journal of Hybrid Information Technology, vol. 4, no. 4, (2011) October.

[5] D. Sangwan and M. K. Yadav, "Design and Implementation of Adder/Subtractor and Multiplication Units for Floating-Point Arithmetic", International Journal of Electronics Engineering, pp. 197-203 (2010).

[6] A. R. Lopes, A. Shahzad, G. A. Constantinides, and E. C. Kerrigan, "More flops or more precision? Accuracy parameterizable linear equation solvers for model predictive control", IEEE Symposium on Field Programmable Custom Computing Machines, Napa, California, 2009

[7] P. M. Seidel, G. Even, "Delay-Optimization Implementation of IEEE Floating-Point Addition," IEEE Transactions on computers, pp. 97-113, vol. 53, no. 2 February 2004.

[8] J. Liang, R. Tessier and O. Mencer, "Floating Point Unit Generation and Evaluation for FPGAs," IEEE Symp. On Field-Programmable Custom Computing Machines, pp. 185-194, April 2003.

[9] J. Liang, R. Tessier, and O. Mencer. "Floating Point Unit Generetion and Evaluation for FPGAs", Proc. of IEEE Symposium on Field Programmable Custom Computing Machines (FCCM'03), California, USA, April 2003.

.