International Journal of Innovative Research in Electronics and Communications (IJIREC)
Volume. 1, Issue 1, April 2014, pp.45-50
www.arcjournal.org

Implementation of Double Precision Floating Point Multiplier in VHDL

Gargi S. Rewatkar
M.Tech Electronics
Wainganga College of Engineering and Management
Nagpur, India
gargis.rewatkar@gmail.com

Abstract: *Floating point numbers are one possible way of representing real numbers in binary format; the IEEE 754 standard presents two different floating point formats, Binary interchange format and Decimal interchange format. Multiplying floating point numbers is a critical requirement for DSP applications involving large dynamic range. Floating-point implementation on FPGAs has been the interest of many researchers. FPGAs are increasingly being used in the high performance and scientific computing community to implement floating-point based hardware accelerators. FPGAs are generally slower than their application specific integrated circuit (ASIC) counterparts, as they can't handle as complex a design, and draw more power. However, they have several advantages such as a shorter time to market, ability to re-program in the field to fix bugs, and lower nonrecurring engineering cost costs. Vendors can sell cheaper, less flexible versions of their FPGAs which cannot be modified after the design is committed. The development of these designs is made on regular FPGAs and then migrated into a fixed version that more resembles an ASIC. In this project we aim to implement double precision floating point multiplier in VHDL.*

Keywords: *Floating Point Multiplier, FPGA, VHDL.*

1. INTRODUCTION

Double precision floating point numbers are 64-bit binary numbers. The 64-bits are divided into 3 parts- sign, exponent and mantissa. The 52 least significant bits (LSBs) are used to represent the mantissa of the number. The next 11-bits are used to represent the exponent of the number. The most significant bit (MSB) of the number is used as a sign bit to represent the sign of the number.

- Sign bit '0' indicates positive number.
- Sign bit '1' indicates negative number.

2. FLOATING POINT MULTIPLIER

Multiplication of two floating point numbers is a complex task and is carried out in a series of steps. Since a floating point number consists of 3 parts- sign, exponent and mantissa, calculations for all the parts are carried out separately.

2.1. Calculation of Sign

The sign bit of the resultant is obtained by carrying out the EXOR operation of the sign bits of the two operands. Sign bit '0' represents a positive sign and sign bit '1' represents a negative sign.

2.2. Calculation of Exponent

The exponents of both the operands are represented in the IEEE 754 format, i.e., a bias of 1023 is added to both the exponents. To calculate the exponent of the resultant the bias of 1023 must be removed from the exponents. After removal of bias from the exponents, both are added to give the resultant exponent. This resultant exponent is in unbiased form. So to represent it in IEEE 754 format, it should be converted to the biased form by adding bias of 1023 to it.

2.3. Calculation of Mantissa

Mantissa calculation is the most complex part of floating point multiplication. A 64-bit number contains 52-bit mantissa. The resultant mantissa is calculated by multiplying the mantissas of both the operands. But before the multiplication is carried out, the mantissas of both the operands need to be normalized. Normalization is done in order to ensure that either of the numbers to be multiplied is not zero. If any one of the numbers is zero than the resultant will be zero. If both the numbers are zero than the resultant will be undefined or Not a Number (NaN). Normalization is done by adding a '1' as the MSB of the mantissa. By adding a '1' as MSB, the possibility of the number being a zero is eliminated. After normalization, the number of bits in the mantissa is increased by one, so the normalized mantissa contains 53-bits. The next step after normalization is multiplication of the normalized mantissas. Two 53-bits mantissas are multiplied and Mantissa calculation is the most complex part of floating point multiplication. A 64-bit number contains 52-bit mantissa. The resultant mantissa is calculated by multiplying the mantissas of both the operands. But before the multiplication is carried out, the mantissas of both the operands need to be normalized. Normalization is done in order to ensure that either of the numbers to be multiplied is not zero. If any one of the numbers is zero than the resultant will be zero. If both the numbers are zero than the resultant will be undefined or Not a Number (NaN). Normalization is done by adding a '1' as the MSB of the mantissa. By adding a '1' as MSB, the possibility of the number being a zero is eliminated. After normalization, the number of bits in the mantissa is increased by one, so the normalized mantissa contains 53-bits. The next step after normalization is multiplication of the normalized mantissas. Two 53-bits mantissas are multiplied and a resultant of 106-bits is obtained. There are several different algorithms which can be used to carry out the multiplication of the mantissas. As the size of the mantissa is very large it is convenient to use an algorithm rather than multiplying directly. This 106-bits resultant cannot be stored directly into the output because of its size. The output mantissa must contain only 52-bits. To obtain 52-bits mantissa, normalization of the 106-bits resultant is carried out. In normalized form the MSB of the number must be 1. Therefore all the '0' bits before the first '1' bit are discarded. Now the mantissa is in normalized form with a '1' as MSB. Now to extract the final 52-bits, de-normalization is carried out. This is done because the mantissa that is finally stored in IEEE 754 format is not in the normalized form as the integer part of the output is by default 1. So the first bit of the number, i.e. 1, is discarded and the next 52 bits are stored as the mantissa of the output, also discarding the remaining least significant bits.

Normalized floating point numbers have the form of Z= (-1S) * 2 (E - Bias) * (1.M). To multiply two floating point numbers the following is done [1]:

1. Multiplying the significand; i.e. (1.M1*1.M2)
2. Placing the decimal point in the result
3. Adding the exponents; i.e. (E1 + E2 – Bias)
4. Obtaining the sign; i.e. s1 xor s2

5. Normalizing the result; i.e. obtaining 1 at the MSB of the results' significant
6. Rounding the result to fit in the available bits
7. Checking for underflow/overflow occurrence

In this paper a floating point multiplier in which rounding support isn't implemented. Rounding support can be added as a separate unit that can be accessed by the multiplier or by a floating point adder, thus accommodating for more precision if the multiplier is connected directly to an adder in a MAC unit. Figure 1 shows the multiplier structure; Exponents addition, Significant and multiplication, and Result's sign calculation are independent and are done in parallel. The significand multiplication is done on two 24 bit numbers and results in a 48 bit product, which we will call the intermediate product (IP). The IP is represented as (47 downto 0) and the decimal point is located between bits 46 and 45 in the IP [1].

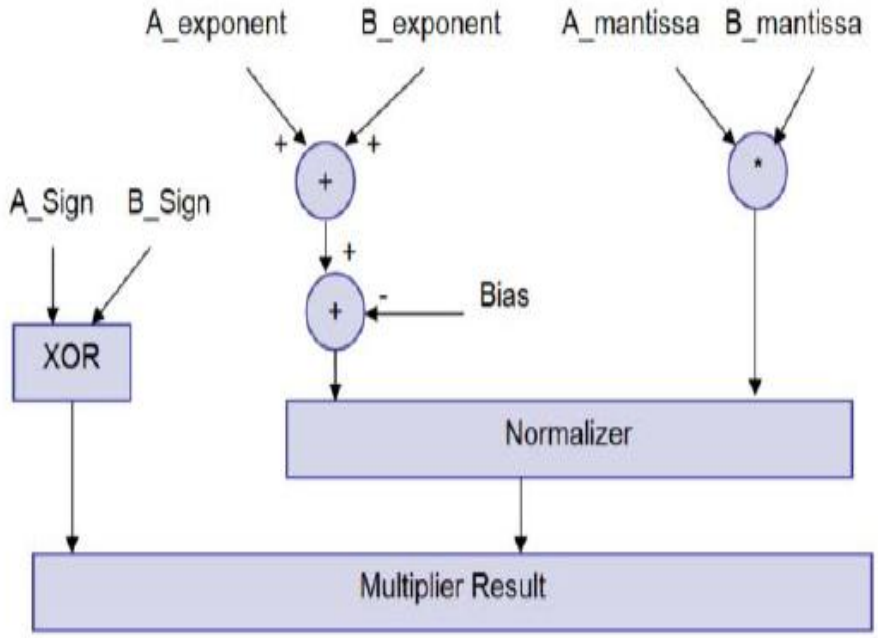


Figure1. *Floating Point Multiplier*

The FPMAC does not support denormalized numbers, infinity, or not-a-number (NaN). If f is the mantissa and exp is the biased exponent; the value of an IEEE-754 compliant FP number is (-1) sign × (1.f) × 2exp. The 1 is not represented in the format, but exists internally in the FPMAC's computation logic. Thus, the bitwidth of the mantissa within the FPMAC is actually 24 bits. The FPMAC, shown in Fig. 4, has 11 pipeline stages [3]. The notable features of this architecture are the "Base-32" conversion and de-conversion (Stages 6 and 11) and the accumulator (Stage 8). The 48-bit mantissa multiplier outputs produced in Stage 5, and retained in carry-save form, are converted to a non-traditional FP. The mantissa is truncated to 24 bits, 22 to the right of the decimal point and 2 to the left. The truncated mantissas are shifted left, as specified by the 5 least significant bits of the exponent, expanding them from 24 to 55 bits: 33 bits to the left of the decimal and 22 bits to the right. Afterwards, only the three most significant bits of the exponent are needed. The value of the FP number in this format is (-1) sign × f* × 2(32 × exp*), where f* is the shifted mantissa and exp* is the truncated exponent. The Base-32 representation eliminates shifters used for mantissa alignment, which are replaced with less costly conditional constant shifters. The outputs of the sign inversion stage are the incoming exponent (E7) and mantissa (C7

and S7). The accumulated result is a feedback exponent (E8) and mantissa (C8 and S8). The updated exponent and mantissa are also stored to E8, C8 and S8. The incoming and feedback mantissas are shifted by the difference between the incoming and feedback exponents. The exponents must be equalized before mantissa addition. The Base-32 representation ensures that all shifts are by 0 or 32 bits [3].

The flow graph of overall algorithm for floating point multiplier including rounding is shown in figure 2 [2].

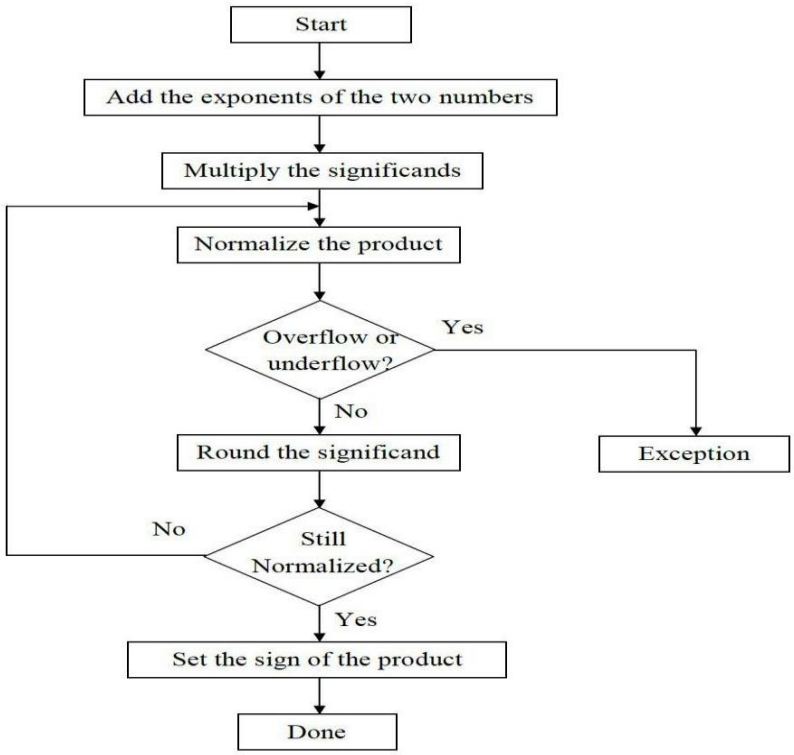


Figure2. *Flow Graph of Floating Point Multiplier*

3. DADDA ALGORITHM

Dadda proposed a sequence of matrix heights that are predetermined to give the minimum number of reduction stages. To reduce the N by N partial product matrix, dada multiplier develops a sequence of matrix heights that are found by working back from the final two-row matrix. In order to realize the minimum number of reduction stages, the height of each intermediate matrix is limited to the least integer that is no more than 1.5 times the height of its successor. The process of reduction for a dadda multiplier [4] is developed using the following recursive algorithm.

1. Let d1=2 and dj+1 = [1.5*dj], where dj is the matrix height for the jth stage from the end. Find the smallest j such that at least one column of the original partial product matrix has more than dj bits.
2. In the jth stage from the end, employ (3, 2) and (2, 2) counter to obtain a reduced matrix with no more than dj bits in any column.
3. Let j = j-1 and repeat step 2 until a matrix with only two rows is generated.

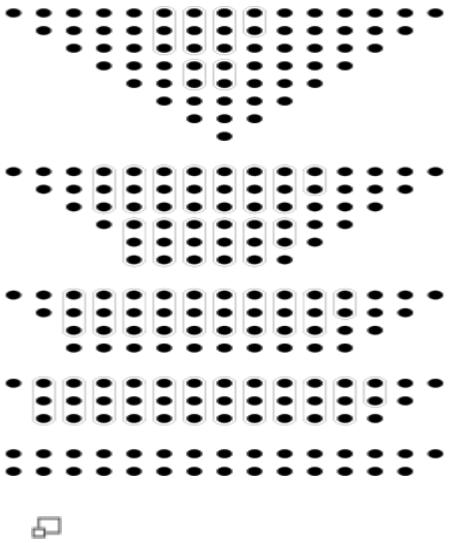


Fig3. *Example of Dadda reduction on 8x8 multiplier*

Basic principle known from manual multiplication

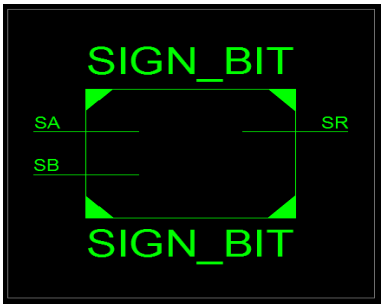
This method of reduction, because it attempts to compress each column, is called a column compression technique. Another advantage of utilizing Dadda multipliers is that it utilizes the minimum number of (3, 2) counters. For Dadda multipliers there are N2 bits in the original partial product matrix and 4.N-3 bits in the final two row matrix. Since each (3, 2) counter takes three inputs and produces two outputs, the number of bits in the matrix is reduced by one with each applied (3, 2) counter therefore, the total number of (3,2) counters is #(3, 2) = N2 – 4.N+3 the length of the carry propagation adder is CPA length = 2.N–2. The 8 by 8 multiplier takes 4 reduction stages, with matrix height 6, 4, 3 and 2. The reduction uses 35 (3, 2) counters, 7 (2, 2) counters, reduction uses 35 (3, 2) counters, 7 (2, 2) counters, and a 14-bit carry propagate adder. The total delay for the generation of the final product is the sum of one AND gate delay, one (3, 2) counter delay for each of the four reduction stages, and the delay through the final 14-bit carry propagate adder arrive later, which effectively reduces the worst case delay of carry propagate adder. The decimal point is between bits 45 and 46 in the significant IR. Critical path is used to determine the time taken by the Dadda multiplier. The critical path starts at the AND gate of the first partial products passes through the full adder of the each stage, then passes through all the vector merging adders. The stages are less in this multiplier compared to the carry save multiplier and therefore it has high speed [4]. As per my project concert I have 52 bit mantissa, so the output result of multiplier contain 104 bit. The stages required are total 9 reduction stages are required and which are reduces the rows in the order or 42, 28, 19, 13, 9, 6, 3 and 2. We required total 2499 (3, 2) counter and also 51 (2, 2) counter. The CPA length is 102.

Methodology

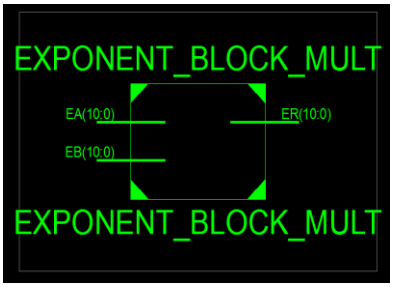
- Theoretical design of floating point multiplier
- Design of floating point multiplier in VHDL
- Simulation and Synthesis of circuit
- Verification using test benches
- Comparison of obtained results with available literature
- Modification if required to improve results
- Thesis writing

4. RESULTS

SIGN BIT



EXPONENT BLOCK



5. CONCLUSION

Double precision floating point multiplier implemented in VHDL may be used applications such as digital signal processors, general purpose processors and controllers and hardware accelerators.

Tools

Xilinx Synthesis Tool Web Pack ISE 10.1i

REFERENCES

[1] M. A. Ashrafy, et al, "An Efficient Implementation of Floating Point Multiplier", in IEEE International Conference on Electronics, Communication & Photonics, 2011, pp 1 – 5.
[2] Pradeep Sharma, et al, "Analysing Single Precision Floating Point Multiplier on Virtex 2P Hardware Module", in International Journal of Engineering Research and Applications, vol 2, no. 5, 2012, pp 2016 – 2020.
[3] Arun Paidimarri, et al, "FPGA Implementation of Single Precision Floating Point Multiply Accumulator with Single Cycle Accumulation", in 17th IEEE Symposium on Field Programmable Custom Computing Machines, 2009.
[4] B. Jeevan, et al, "A High Speed Binary Floating Point Multiplier using Dadda Algorithm", in IEEE International MultiConference on Automation, Computing, Communication, Control and Compressed Sensing, 2013.