Design of High Speed Multiplier for Floating Point Numbers using DADDA Algorithm

Dondapati Subba Rao[1] S.Ravindra[2]
[1]P.G. Scholar [2]Assistant Professor
[1,2]Nimra Institute of Science and Technology Vijayawada, A.P. 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. So it is widely used in large set of scientific and signal processing computation. Multiplication is one of the common arithmetic operations in these computations. To improve speed multiplication of mantissa is done using Dadda multiplier replacing carry save multiplier which can be able to handle underflow and overflow cases. A high speed floating point Dadda multiplier is implemented in verilog HDL. Even Rounding is not implemented for high precision. This paper presents a high speed binary Dadda precession floating point multiplier based on Dadda Algorithm.*

Keywords—Dadda Algorithm, Floating point, Multiplier, IEEE-754, Verilog HDL.

I. Introduction

The real numbers represented in binary format are known as floating point numbers. Based on IEEE-754 standard, floating point formats are classified into binary and decimal interchange formats. Floating point multipliers are very important in DSP applications. This paper focuses on Dadda precision normalized binary interchange format. Figure I show the IEEE-754 Dadda precision binary format representation. Sign (S) is represented with one bit; exponent (E) and fraction (M or Mantissa) are represented with eleven and fifty two bits respectively. For a number is said to be a normalized number, it must consist of 'one' in the MSB of the significant and exponent is greater than zero and smaller than 1023. The real number is represented by equations (I) & (2).

Floating point implementation has been the interest of many researchers. In an IEEE-754 single precision pipelined floating point multiplier is implemented with custom 16/18 bit three stage pipelined floating point multiplier, that doesn't support rounding modes [1]. L.Louca, T.A.Cook, W.H. Johnson [2] implemented a single precision floating point multiplier by using a digit-serial multiplier. The design achieved 2.3 MFlops and doesn't support rounding modes. The multiplier handles the overflow and underflow cases but rounding is not implemented. The design achieves 30 I MFLOPs with latency of three clock cycles. The multiplier was verified against Xilinx floating point multiplier core.

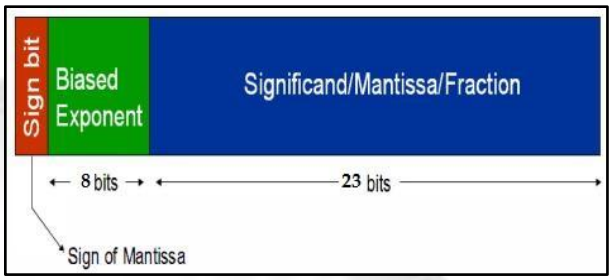


Fig. 1: IEEE 754 Dadda Floating Point Format.

The Dadda floating point multiplier presented here is based on IEEE-754 binary floating standard. We have designed a high speed Dadda precision floating point multiplier using Verilog language. It handles the overflow, underflow cases and rounding mode.

II. Floating Point Multiplication Algorithm

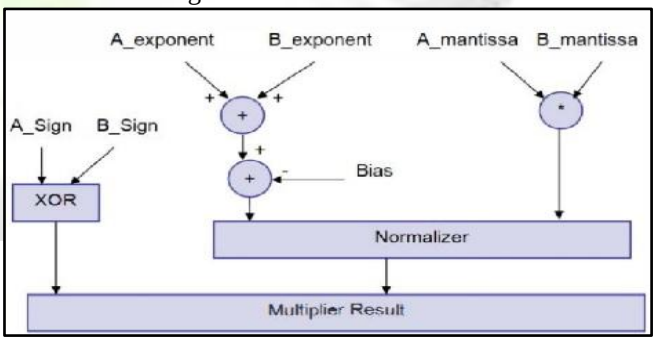
Multiplying two numbers in floating point format is done by

- Adding the exponent of the two numbers then subtracting the bias from their result.
- Multiplying the significant of the two numbers
- Calculating the sign by XORing the sign of the two numbers.

In order to represent the multiplication result as a normalized number there should be I in the MSB of the result (leading one).

The following steps are necessary to multiply two floating point numbers.

- Multiplying the significant i.e. (I.MI * I.M2)
- Placing the decimal point in the result
- Adding the exponents i.e. (E I + E2 - Bias)
- Obtaining the sign i.e. sl xor s2
- Normalizing the result i.e. obtaining I at the MSB of the results "significant"
- Rounding the result to fit in the available bits
- Checking for underflow/overflow occurrence



III. Multiplier

Existing Multiplier: Carry Save Multiplier:

This unit is used to multiply the two unsigned significant numbers and it places the decimal point in the multiplied product. The unsigned significant multiplication

is done on 24 bit. The result of this significant multiplication will be called the IR. Multiplication is to be carried out so as not to affect the whole multiplier's performance. In this carry save multiplier architecture is used for 24X24bit as it has a moderate speed with a simple architecture. In the carry save multiplier, the carry bits are passed diagonally downwards (i.e. the carry bit is propagated to the next stage). Partial products are generated by ANDing the inputs of two numbers and passing them to the appropriate adder. Carry save multiplier has three main stages:

- The first stage is an array of half adders.
- The middle stages are arrays of full adders. The number of middle stages is equal to the significant size minus two.
- The last stage is an array of ripple carry adders. This stage is called the vector merging stage.

The count of adders (Half adders and Full adders) in each stage is equal to the significant size minus one. For example, a 4x4 carry save multiplier is shown in Figure 8 and it has the following stages:

- The first stage consists of three half adders.
- Two middle stages; each consists of three full adders.
- The vector merging stage consists of one half adder and two full adders.

The decimal point is placed between bits 45 and 46 in the significant multiplier result. The multiplication time taken by the carry save multiplier is determined by its critical path. The critical path starts at the AND gate of the first partial products (i.e. a1b0 and a0b1), passes through the carry logic of the first half adder and the carry logic of the first full adder of the middle stages, then passes through all the vector merging adders. The critical path is marked in bold in Figure 4.

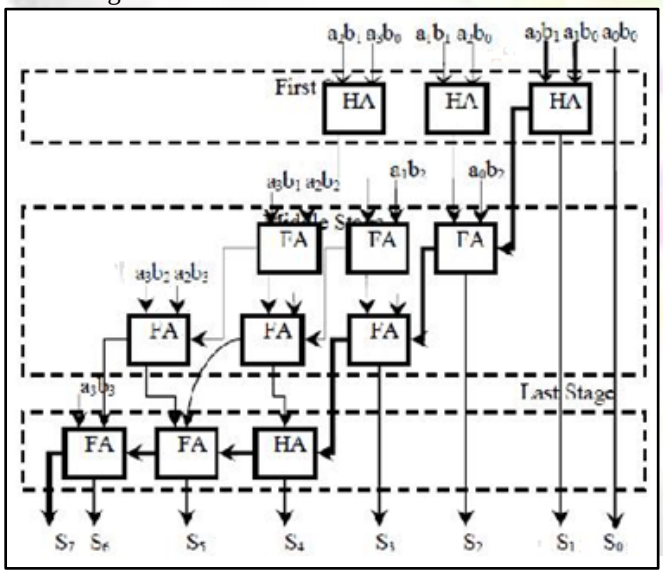


Fig. 4: 4x4 bit Carry Save multiplier

In Figure 4

- Partial product: aibj ai and bj
- HA: half adder.
- FA: full adder.

IV. PROPOSED MULTIPLIER DADDA MULTIPLIER

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 [3] is developed using the following recursive algorithm

- 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.
- 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.
- Let j = j-1 and repeat step 2 until a matrix with only two rows is generated.

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. {Therefore, the number of intermediate stages is set in terms of lower bounds: 2, 3, 4, 6, 9 . . .

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.

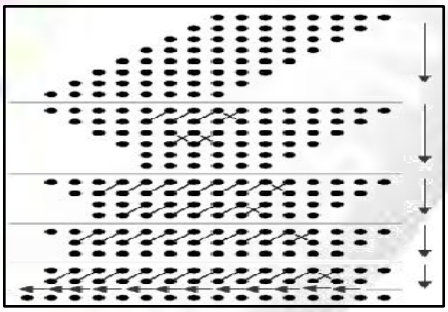


Fig. 5: Dot diagram for 8 by 8 Dadda Multiplier

The number of (2, 2) counters used in Dadda's reduction method equals N-1.The calculation diagram for an 8X8Dadda multiplier is shown in figure 5. Dot diagrams are useful tool for predicting the placement of (3, 2) and (2, 2) counter in parallel multipliers. Each IR bit is represented by a dot.

The output of each (3, 2) and (2, 2) counter are represented as two dots connected by a plain diagonal line. The outputs of each (2, 2) counter are represented as two dots connected by a crossed diagonal line.

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 than that.

V. .ROUNDING AND EXCEPTIONS

The IEEE standard specifies four rounding modes round to nearest, round to zero, round to positive infinity, and round to negative infmity. Table 1 shows the rounding modes selected for various bit combinations of rmode. Based on the rounding changes to the mantissa corresponding changes has to be made in the exponent part also.

Bit combination	Rounding Mode
00	round_nearest_even
01	round_to_zero
10	round_up
11	round_down

Table1: Rounding modes selected for various bit combinations of rmode in the exceptions module, all of the special cases are checked for, and if they are found, the appropriate output is created, and the individual output signals of underflow, overflow, inexact, exception, and invalid will be asserted if the conditions for each case exist.

VI. IMPLEMENTATION AND TESTING

The Dadda floating point multiplier design was simulated in Modelsim and synthesized using Xilinx ISE Table 2 shows the area and operating frequency of dadda floating point multiplier Xilinx core respectively.

	Proposed Floating point Multiplier	Existing Floating Point Multiplier	Xilinx Core
LUT & FF Pairs Used	1,146	1,110	235
CLB Slices	1,149	1,112	732
Max Frequency (MHz)	526.857	401.711	206.299

Table 2: Area and Frequency Comparison between the Implemented Floating Point Multiplier and Xilinx Core

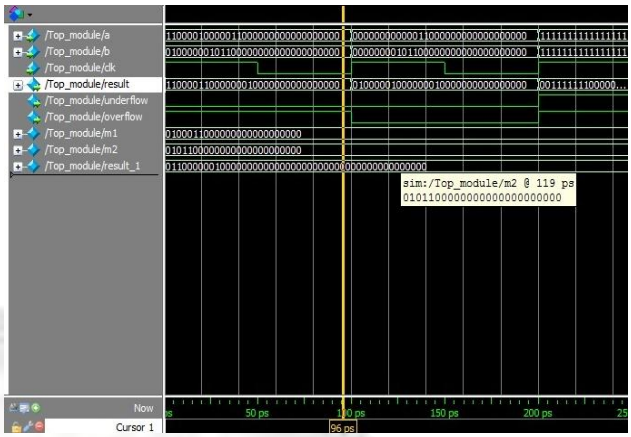


Fig. 6:Simulation Result of Dadda Floating Point Multiplier

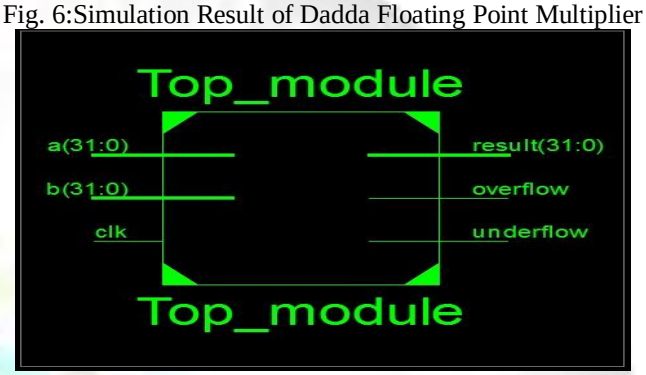


Fig.7: RTL Schematic for top level module

VII. CONCLUSION

The Dadda floating point multiplier high speed multiplication of mantissa. The design achieves the operating frequency of 526 MHz MFLOOPS with area of 676 slices. The implemented design is verified with Xilinx Virtex core, which gives more accuracy compared to single precession. This design handles the overflow, underflow mode.

REFERENCES

[1] 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.

[2] L. Louca, T. A. Cook, and W. H. Johnson, "Implementation of IEEE Single Precision Floating Point Addition and Multiplication on FPGAs," Proceedings of 83rd IEEE Symposium on FPGAs for Custom Computing Machines (FCCM"96), pp. 107-116,1996.

[3] Whytney J. Townsend, Earl E. Swartz, "A Comparison of Dadda and Wallace multiplier delays". Computer Engineering Research Center, The University of Texas.

[4] Mohamed AI-Ashraf)', Ashraf Salem, Wagdy Anis., "An Efficient Implementation of Floating Point Multiplier ", Saudi International Electronics, Communications and Photonics Conference (SIECPC), pp. 1-5,24-26 April 2011.

[5] B. Lee and N. Burgess, "Parameterisable Floating-point Operations on FPG A," Conference Record of the ThirtySixth Asilomar Conference on Signals, Systems, and Computers, 2002.
[6] Xilinx13.4, Synthesis and Simulation Design Guide", UG626 (v13.4) January 19, 2012.
[7] 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.