A High Speed Binary Single Precision Floating Point Multiplier Using Dadda Algorithm and Parallel Prefix Adder

Mr. Shailesh Kumar[1] Padmapriya Patil[2]
[1]Student, Department of VLSI Design and Embedded systems,
[2]Professor, Department of Electronics and Communication,
[1]VTU RC Gulbarga [2]PDA Gulbarga

Abstract--- Floating Point Arithmetic is extensively used in the field of Digital signal processing, Medical imaging, motion capture, audio application including broadcast and musical instruments .Many of these applications need to solve linear systems that use fair amount of matrix multiplication. The multiplier conforms to the IEEE 754 standard for single precision. The IEEE standard for Binary Floating point Arithmetic (IEEE 754)is most widely used standard for Floating point computation and is followed by many CPU and FPU implementation. An algorithm called Dadda algorithm is introduced here for reducing the partial product of multiplier unit single precision format. In this thesis to improve speed of multiplication of mantissa is done by using Dadda algorithm replacing carry save multiplier. The sub-module have been written in Verilog HDL and then synthesized simulated using Xilinx ISE 12.4. To find overall more speed of multiplication of two 32 bit Floating point numbers the exponents of two 32 bit Floating point numbers is calculated as Ea +Eb -127.Here exponent addition takes place by using kogge stone adder instead of Ripple carry adder which is used earlier. Earlier speed was 526 MHZ but the use of Dadda algorithm in mantissa multiplication and kogge stone adder in the exponent calculation increases the speed and reduces the delay time. So the kogge stone adder is faster adder and reduces delay. The Floating point multiplier is developed to handle overflow and underflow cases.

Keywords- Dadda algorithm; Single precision Floating point number; multiplication; Verilog HDL.

I. INTRODUCTION

Most of the DSP applications need floating point numbers multiplication. The possible ways to represent real numbers in binary format floating point numbers are; the IEEE 754 standard represents two floating point formats, Binary interchange format and Decimal interchange format. Single precision normalized binary interchange format is implemented in this design. Representation of single precision binary format is shown in Figure 1; starting from MSB it has a one bit sign (S), an eight bit exponent (E), and a twenty three bit fraction (M or Mantissa). Adding an extra bit to the fraction to form and is defined as significand1. If the exponent is greater than 0 and smaller than 255, and there is 1 in the MSB of the significand then the number is said to be a normalized number; in this case the real number is represented by

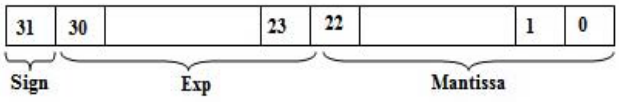

Fig. 1: IEEE single precision floating point format

Z= 1.M *2(E-bias) *(-1) s
Bias=127

Significand is the extra MSB bit appended to mantissa.

- Step1. Exponents of the two numbers are added directly, in this addition kogge stone adder is used extra bias is subtracted from the exponent result which is 127 for single precision Floating point numbers.
- Step 2.Significands multiplication of the two numbers using Dadda algorithm.
- Step3. To find the sign of result, XOR operation is done among sign bit of two numbers.
- Step 4.Finally the result is normalized such that there should be 1 in the MSB of the result.

II. FLOATING POINT MULTIPLIER ALGORITHM

The normalized floating point numbers have the form of Z= (-1)*2^ (E-Bias) *(1.M).The following algorithm is used to multiply two floating point numbers.

1. Significand multiplication: ie(1.M1*1.M2).
2. Give the decimal point in result.
3 .Exponent addition i.e.(E1+E2-Bias).
4. Getting the sign: i.e. s1XOR S2.
5. Normalizing the result: i.e. obtaining 1 at MSB of the results' significand.
6. Rounding implementation.
7. Verify for underflow/overflow occurrence.

Consider the following IEEE 754 Single precision floating point numbers to perform the multiplication, but the number of mantissa bits is reduced for simplification. Here only 5 bits are considered while still considering one bit for normalized numbers:

A = 0 10000001 01100 = 5.5, B = 1 10000100 00011 = -35

By following the algorithm the multiplication of A and B is

1. Significand Multiplication: 1.01100×1.00011=011000000100
2. Normalizing the result: 1.1000000100
3. Adding two exponents: 10000001+10000100=100000101

The result after adding two exponents is not true exponent and is obtained by subtracting bias value i.e 127. The same is shown in following equations.

EA = EA-true + bias
EB = EB-true + bias
EA + EB = EA-true + EB-true + 2 x bias
Therefore Etrue= EA + EB – bias.

From the above analysis bias is added twice so bias has to be subtracted once from the result.

100000101-001111111=10000110

4. Sign bit of result is extracted by doing XOR operation of sign bit of two numbers:

1 10000110 01.1000000100

5. Then normalize the result so that there is a 1 just before the radix point (decimal point). Moving the radix point one place to the left increments the exponent by 1; moving one place to the right decrement the exponent by 1.
6. If the mantissa bits are more than 5 bits (mantissa available bits); rounding is needed. If we applied the truncation rounding mode then the stored value is: 11000011010000. In this we are presenting a floating point multiplier in which rounding support is not implemented. By this we get more precision in MAC unit and this will be accessed by the multiplier or by a floating point adder unit. calculator, Mantissa multiplier and sign bit calculator, using the pipelining concept here all processes are carried out in parallel. Two 24 bit significands are multiplied and the result is a 48 bit product, denoting this as Intermediate Result (IR). The IR width is 48-bit i.e. 47 down to 0 and the decimal point is located between bits 46 and 45 in the IR.

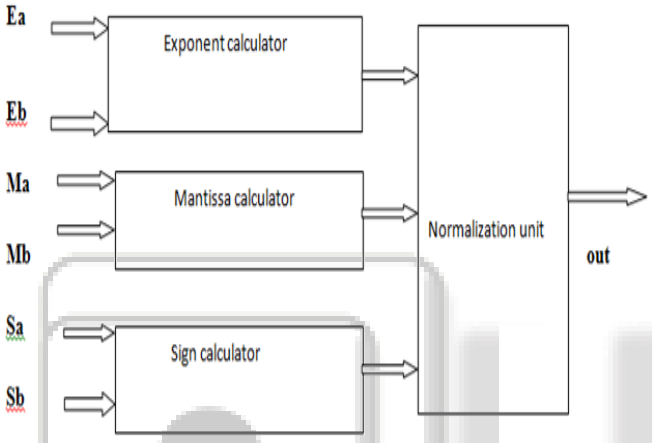

Fig. 2: Floating point multiplier Block Diagram

III. Problem Definition

Earlier the 8 bit addition of two exponents done by ripple carry adder which has more delay so overall multiplication of two single precision floating point number has less speed .Earlier for multiplication of two 23 bit mantissa has done by carry save multiplier. This mantissa multiplication have 48 partial product so used to many Half adder and Full adder. So time of mantissa multiplication have more delay.

A. Objective

In this thesis the overall speed of multiplication of two 32 bit single precision increases by using kogge stone adder which has less delay so multiplication time will decrease here Dadda algorithm is also used for reduction of partial product up to two row. By the use of Dadda algorithm and kogge stone adder the overall speed of two 32 bit multiplications will increase.

IV. Proposed Design

A. 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 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. 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

The number of (2, 2) counters used in Dadda's reduction method equals N-1.The calculation diagram for an 8X8 Dadda multiplier is shown in figure 2. 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

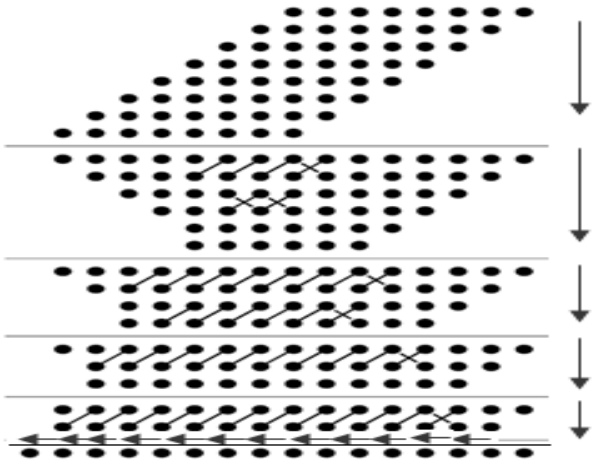

Fig. 3: Dot Diagram for 8*8 Dadda Multiplier

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 significand 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. EXPONENT CALCULATION

A. *Prefix Adder*

Binary addition is the most fundamental and frequently used arithmetic operation. A lot of work on adder design has been done so far and much architecture has been proposed. When high operation speed is required, tree structures like parallel-prefix adders are used. In Sklansky proposed one of the earliest tree-prefix is used to compute intermediate signals. In the Brent-Kung approach designed the computation graph for area-optimization. The KS architecture is optimized for timing. The LF architecture is proposed, where the fan-out of gates increased with the depth of the prefix computation tree. The HC adder architecture is based on BK and KS is proposed. In an algorithm for back-end design is proposed. The area minimization is done by using bitwise timing constraints. In which is targeted to minimize the total switching activities under bitwise timing constraints. The architecture saves one logic level implementation and reduces the fan-out requirements of the design. A fast characterization process adders is proposed using matrix representation The Parallel Prefix addition is done in three steps, which is shown in fig3. The fundamental generate and propagate signals are used to generate the carry input for each adder. Two different operators black and gray are used here.

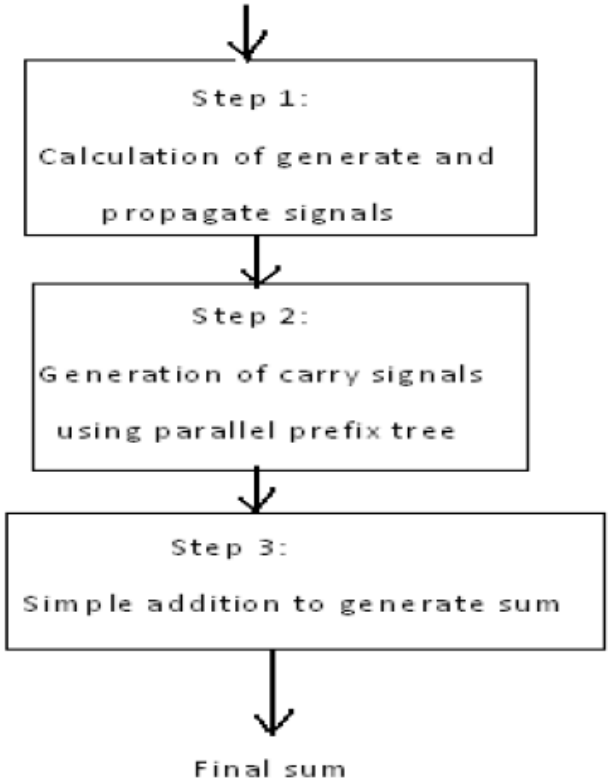

Fig. 4: Addition procedure using parallel prefix Tree structure

VI. PRELIMINARIES

In every bit (*i*) of the two operand block, the two input signals (*ai* and *bi*) are added to the corresponding carry-in signal (*carryi*) to produce sum output (*sumi*) The equation to produce the sum output is:

$$sum_i = a_i \oplus b_i \oplus carry_i$$

Computation of the carry-in signals at every bit is the most critical and time – consuming operation. In the carry- look ahead scheme of adders (CLA), the focus is to design the carry-in signals for an individual bit additions. This is achieved by generating two signals, the generate (*gi*) and propagate (*pi*) using the equations: -

$$g_i = a_i \wedge b_i$$

$$p_i = a_i \oplus b_i$$

The carry in signal for any adder block is calculated by using the formula) –

$$c_{i+1} = g_i \vee (p_i)$$

Where ci must be expanded to calculate ci+1 at any level of addition. Parallel Prefix adders compute carry-in at each level of addition by combining generate and propagate signals in a different manner. Two operators namely *black* and *gray* are used in parallel prefix trees are shown in fig 5(a), fig 5(b) respectively.

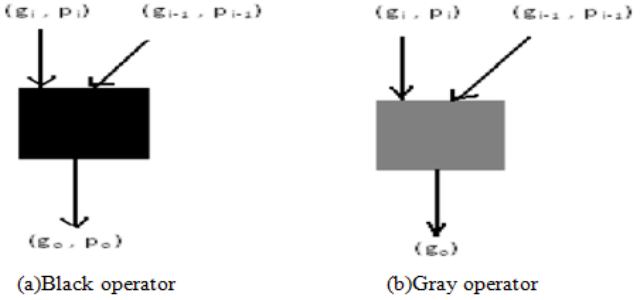

Fig. 5: Operators used in Prallel Prefix Tree

The black operator receives two sets of generate and propagate signals (gi , pi),(gi-1 ,pi-1), computes one set of generate and propagate signals (go , po) by the following equations:

$$g_o = g_i \vee (p_i \wedge g_{i-1})$$

$$p_o = p_i \wedge p_{i-1}$$

The gray operator receives two sets of generate and propagate signals (gi, pi),(gi-1 ,pi-1) computes only one generate signal with the same equation as in equation. The logic diagram of black operator and gray operator is shown in fig 6(a), fig 6(b) respectively.

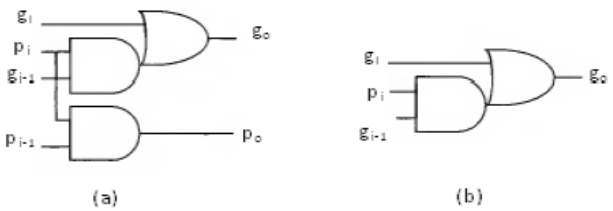

Fig. 6: Logic Diagram of (a) Black operator (b) Gray operator

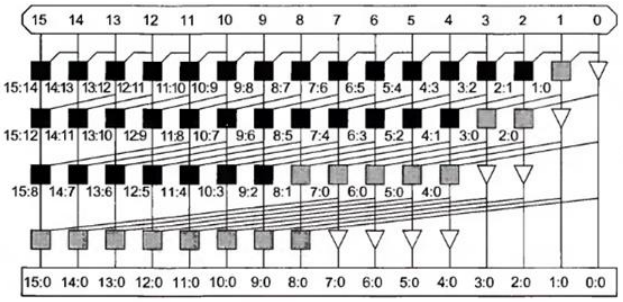

Fig. 7: Kogge stone adder

A. Result and discussion

By the use of Dadda algorithm and Prefix adder(Kogge stone) the speed of multiplication is increased of single precision floating point number.

B. Top module:

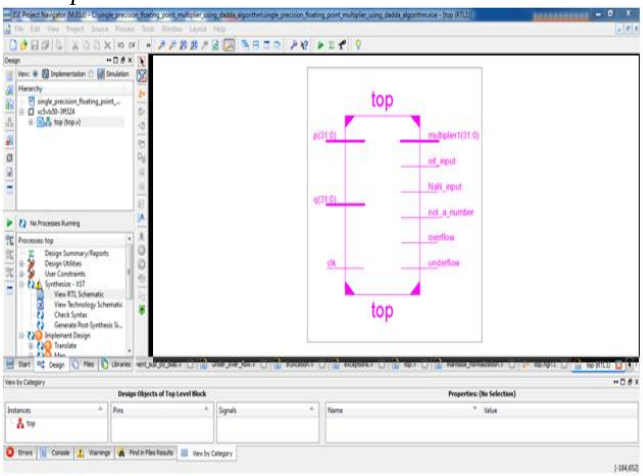

Fig. 8: Top Module of Multiplier

C. RTL Schematic

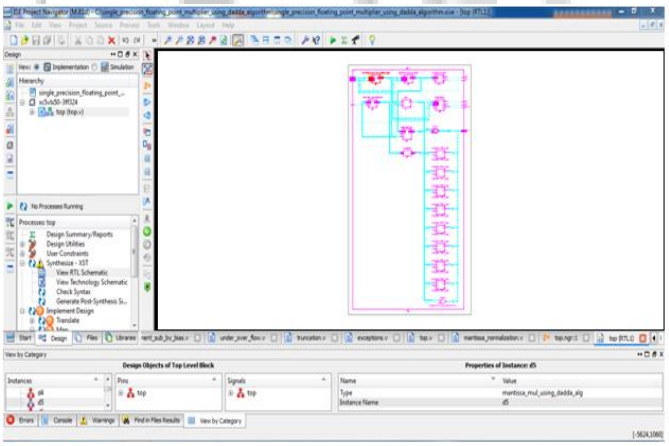

Fig. 9: RTL Schematic of Multiplier

D. Frequency

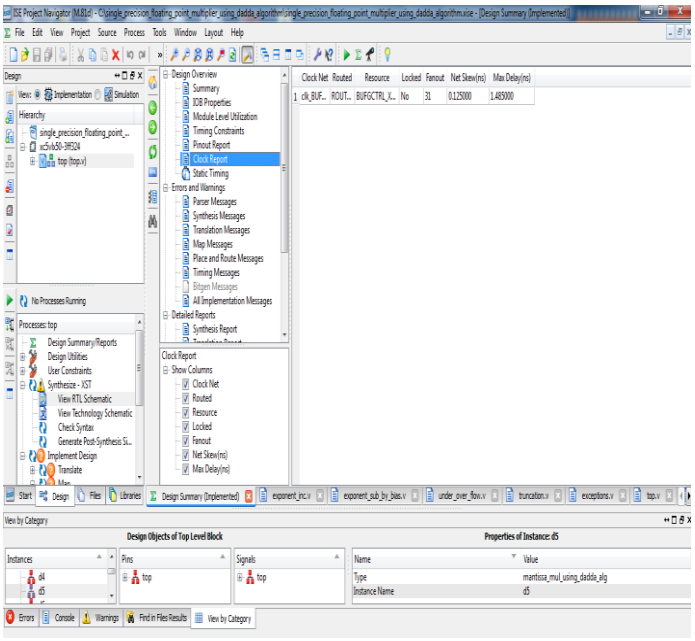

Fig. 10: calculation of frequency by the use of delay

E. Simulation result:

The simulation results for corresponding input shown in figure. The simulation is done using Xilinx.

Inputs: p=32'b 01111111100000000000000000000000
Q=32'b 01111111100000000000000000000000

Output: result = 01111111100000000000000000000000

Here this is an exeption handling case so overflow takes place and exponent is not a number because all 8'b are one.

Frequency can be calculated by using formula: F=1/T where T is delay.

The maximum speed will be 674 MHZ. The simulation output is shown in Fig.11.

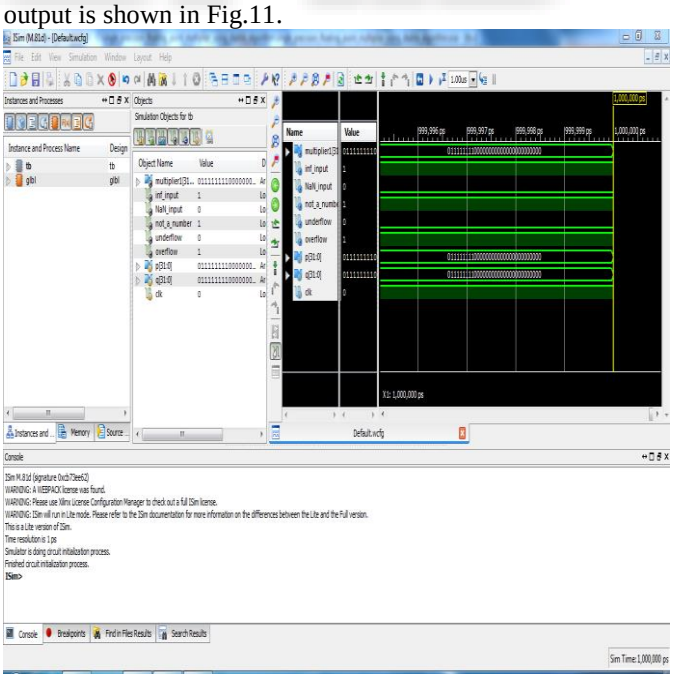

Fig. 11: Floating point multiplier simulation output

F. Future scope:

This paper describes simulation of floating point multiplier of single precision that supports IEEE 754 -2008[1] binary

interchange format; this multiplier can be used for double precision floating point which is 64*64 bit and at high speed.

REFERENCES

[1] IEEE 754-2008 Standard for floating point Arithmetic 2008.
[2] Mohamed AL-Ashrfy, Asharaf Salem and Wagdy Anis "An Efficient Implementation of Floting Point Multiplier" IEEE Transaction on VLSI 978-1-4577-0069-9/11@2011 IEEE, Mentor Graphics.
[3] B. Fagin and C. Renard, "Field Programmable Gate Arrays and Floating Point Arithmetic," IEEE Transactions on VLSI, vol. 2, no. 3, pp. 365- 367, 1994.
[4] 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
[5] L. Louca. T.A. cook, and W.H. Johnson," Implementation of IEEE Single Precision Floating point Addition and Multiplication on FPGA"Proceeding of 83 the IEEE Symposium on FPGA for custom computing Machine pp 107-116 1996.
[6] Xilinx 13.4 , Synthesis and simulation Design Guide ,UG (V13.4) January 19,2012.
[7] Whytney j.Townsend , Earl E.Swartz " A comparision of Dadda and Wallace multiplier delays". Computer Engineering Research Center , The university of Texas.

