# *A High Speed Binary Floating Point Multiplier Using Dadda Algorithm*

B. Jeevan,
Asst. Professor,
Dept. of E&IE,
KITS, Warangal.
jeevanbs776@gmail.com

S. Narender,
M.Tech (VLSI&ES),
KITS,
Warangal.
narender.s446@gmail.com

Dr C.V. Krishna Reddy,
Director, Nalla Narendra
Group of Institutions,
Ghatkesar, Hyderabad.
cvkreddy2@gmail.com

Dr K. Sivani,
Principal,
Talla Padmavati College
of Engg. Warangal.
k_sivani@yahoo.co.in

***Abstract*—This paper presents a high speed binary floating point multiplier based on Dadda Algorithm. To improve speed multiplication of mantissa is done using Dadda multiplier replacing Carry Save Multiplier. The design achieves high speed with maximum frequency of 526 MHz compared to existing floating point multipliers. The floating point multiplier is developed to handle the underflow and overflow cases. To give more precision, rounding is not implemented for mantissa multiplication. The multiplier is implemented using Verilog HDL and it is targeted for Xilinx Virtex-5 FPGA. The multiplier is compared with Xilinx floating point multiplier core.**

***Keywords—Dadda Algorithm; Floating point; multiplication; FPGA, 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 [1] 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 significand[1]. 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 (1).

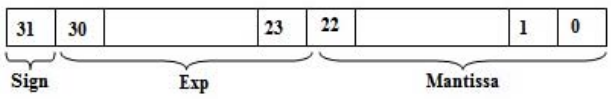


Fig. 1. IEEE single precision floating point format

$$Z = (-1^{S}) * 2^{(E - Bias)} * (1.M) \qquad (1)$$

Where $M = n_{22}2^{-1} + n_{21}2^{-2} + n_{20}2^{-3} + \ldots + n_1 2^{-22} + n_0 2^{-23}$;

Bias = 127.

[1] Significand is the extra MSB bit appended to mantissa.

Floating point multiplication of two numbers is made in four steps:

Step 1. Exponents of the two numbers are added directly, extra bias is subtracted from the exponent result.

Step 2. Significands multiplication of the two numbers using Dadda algorithm.

Step 3. 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 (leading one).

Most interesting area of many researchers is to implement Floating-point multipliers on FPGAs. In [2], the design of an efficient implementation of single precision floating point multiplier was verified against virtex-5 FPGA in which IEEE 754 single precision pipelined floating point multiplier was implemented on multiple FPGAs (4 Actel A1280). In [5], an another custom 16/18 bit three stage pipelined floating point multiplier was implemented that has no rounding modes. In [6], a single precision floating point multiplier that doesn't support rounding modes was implemented using a digit-serial multiplier: using the Altera FLEX 8000 it achieved 2.3 MFlops. In [7], a parameterizable floating point multiplier was implemented using the software-like language Handel-C, using the Xilinx XCV1000 FPGA; a five stages pipelined multiplier achieved 28MFlops. In [8], a latency optimized floating point unit using the primitives of Xilinx Virtex II FPGA was implemented with a latency of 4 clock cycles. The multiplier reached a maximum clock frequency of 100MHz.

## II. FLOATING POINT MULTIPLIER ALGORITHM

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

1. Significand multiplication; i.e. $(1.M_1 * 1.M_2)$.
2. Placing the decimal point in the result.
3. Exponent's addition; i.e. $(E_1 + E_2 - Bias)$.
4. Getting the sign; i.e. $s_1$ xor $s_2$.
5. Normalizing the result; i.e. obtaining 1 at the MSB of the results' significand.
6. Rounding implementation.
7. Verifying for underflow/overflow occurrence.

Consider the following IEEE 754 single precision

978-1-4673-5090-7/13/$31.00 ©2013 IEEE

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:

$$\begin{array}{r} 1.01100 \\ \times 1.00011 \\ \hline 101100 \\ 101100 \\ 000000 \\ 000000 \\ 000000 \\ 101100 \\ \hline 011000000100 \end{array}$$

2. Normalizing the result: 1.1000000100

3. Adding two exponents:

$$\begin{array}{r} 10000001 \\ +10000100 \\ \hline 100000101 \end{array}$$

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.

$$E_A = E_{A\text{-true}} + \text{bias}$$

$$E_B = E_{B\text{-true}} + \text{bias}$$

$$E_A + E_B = E_{A\text{-true}} + E_{B\text{-true}} + 2 \times \text{bias}$$

Therefore

$$E_{true} = E_A + E_B - \text{bias}.$$

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

$$\begin{array}{r} 100000101 \\ -001111111 \\ \hline 10000110 \end{array}$$

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:

1 10000110 10000.

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. Figure 2 shows the block diagram of the multiplier structure; Exponents 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. Each block is elaborated in the following sections.

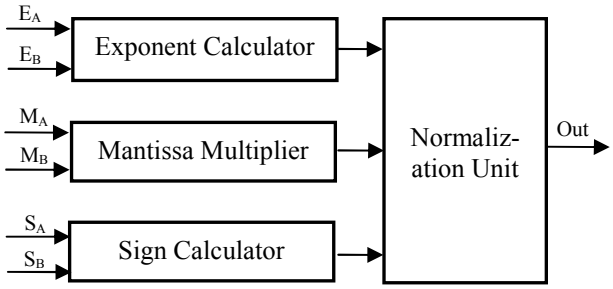


Fig. 2. Floating point multiplier block diagram

## III. MAIN BLOCKS OF FLOATING POINT MULIPLIER

### *A. Sign calculator*

The main component of Sign calculator is XOR gate. If any one of the numbers is negative then result will be negative. The result will be positive if two numbers are having same sign.

### *B. Exponent Adder*

This sub-block adds the exponents of the two floating point numbers and the Bias (127) is subtracted from the result to get true result i.e. $E_A + E_B - \text{bias}$. In this design the addition is done on two 8 bit exponents. In previous designs the most of the computational time is spending in the significand multiplication process (multiplying 24 bits by 24 bits); so quick result of the addition is not necessary. Thus we need a fast significand multiplier and a moderate exponent adder.

To perform addition of two 8-bit exponents an 8-bit ripple carry adder (RCA) is used. As shown in Figure 3 the ripple carry adder consists an array of one Half Adder (HA) (i.e. to which two LSB bits are fed) and Full Adders (FA) (i.e. to which two input bits and previous carry are given). The HA has two inputs and two outputs and FA has three inputs ($A_o$, $B_o$, $C_i$) and two outputs ($S_o$, $C_o$). The carry out ($C_o$) of each adder is fed to the next full adder (i.e. each full adder has to wait for carry out of preceding).

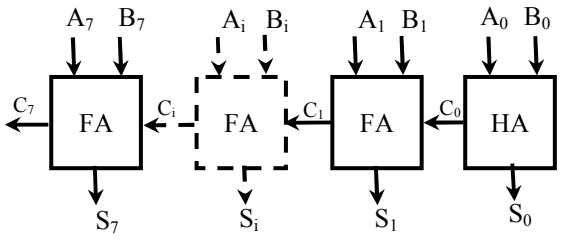


Fig. 3. Ripple Carry Adder

*i. One Subtractor (OS):*

The single bit subtractor is shown in fig.4; used for subtracting the bias. A normal subtractor has three inputs (minuend (X), subtrahend (Y), Borrow in ($B_i$)) and two outputs (Difference (D), Borrow out ($B_o$)). The subtractor logic can be optimized if one of its inputs is a constant value which is our case, where the Bias is constant ($127_{10} = 001111111_2$). Table I shows the truth table for a 1-bit subtractor with the input equal to 1 which we will call "one subtractor (OS)".

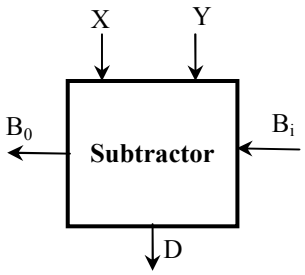


Fig. 4. Ripple Carry Adder

TABLE I. 1-BIT SUBTRACTOR WITH INPUT Y=1

| X | Y | $B_i$ | D | Bo |
|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

The Boolean equations (2) and (3) represent this subtractor

$$Difference\ (D) = \overline{X \oplus B_i}$$
$$Borrow\ \ (Bo) = \overline{X} + B_i$$

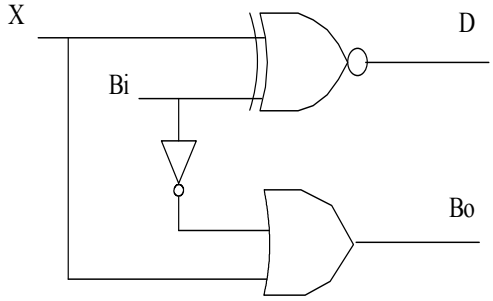


Fig. 5. 1-bit subtractor with the input Y = 1

ii. *Zero Subtractor (ZS)*

Table II shows the truth table for a 1-bit subtractor with the input Y equal to 0 which we will call "zero subtractor (ZS)"

TABLE II. 1-BIT SUBTRACTOR WITH THE INPUT Y=0

| X | Y | Bi | D | Bo |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 0 | 0 |

$$Difference\ \ (D) = S \oplus B_i$$
$$Borrow\ \ (Bo) = \overline{S}.\, B_i$$

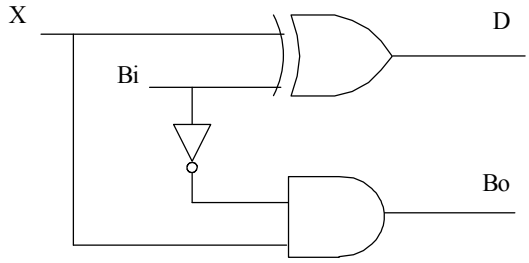


Fig. 6. 1-bit Subtractor with input T= 0.

Fig. 7 shows the Bias subtractor which is a chain of 7 one subtractors (OS) followed by 2 zero subtractors (ZS); the borrow output of each subtractor is fed to the next subtractor. If an underflow occurs then $E_{result} < 0$ and the number is out of the IEEE 754 single precision normalized numbers range; in this case the output is signaled to 0 and an underflow flag is asserted.

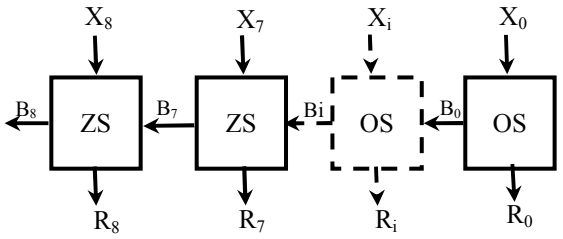


Fig. 7. Ripple Borrow Subtractor

*C. Significand multiplication using Unsigned Multiplier*

*i. Existing Multiplier:*

*Carry Save Multiplier:*

This unit is used to multiply the two unsigned significand numbers and it places the decimal point in the multiplied product. The unsigned significand multiplication is done on 24 bit. The result of this significand 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 24X24 bit 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:

1. The first stage is an array of half adders.
2. The middle stages are arrays of full adders. The number of middle stages is equal to the significand size minus two.
3. 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 significand size minus one. For example,

a 4x4 carry save multiplier is shown in Figure 8 and it has the following stages:

1. The first stage consists of three half adders.
2. Two middle stages; each consists of three full adders.
3. 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 significand 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. $a_1b_0$ and $a_0b_1$), 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 8.

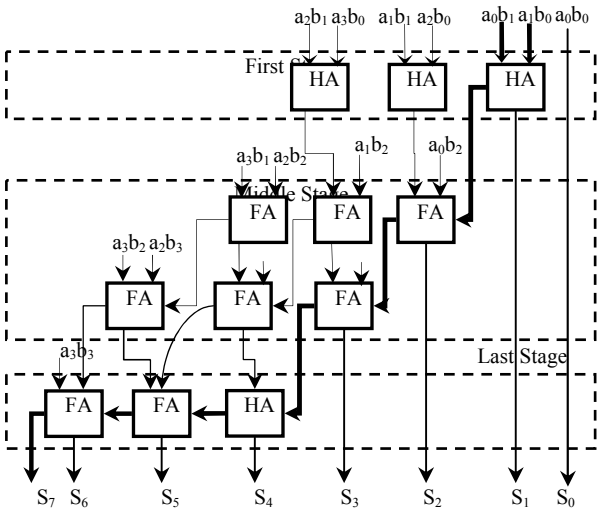


Fig. 8. 4x4 bit Carry Save multiplier

In Figure 8

1. Partial product: $a_ib_j$ $a_i$ and $b_j$
2. HA: half adder.
3. FA: full adder.

### *ii. 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 [7] is developed using the following recursive algorithm

1. Let d1=2 and $d_j$+1 = [1.5*$d_j$], 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 $j^{th}$ stage from the end, employ (3, 2) and (2, 2) counter to obtain a reduced matrix with no more than $d_j$ 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 $N^2$ 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) = $N^2$ – 4.N+3 the length of the carry propagation adder is CPA length = 2.N–2.

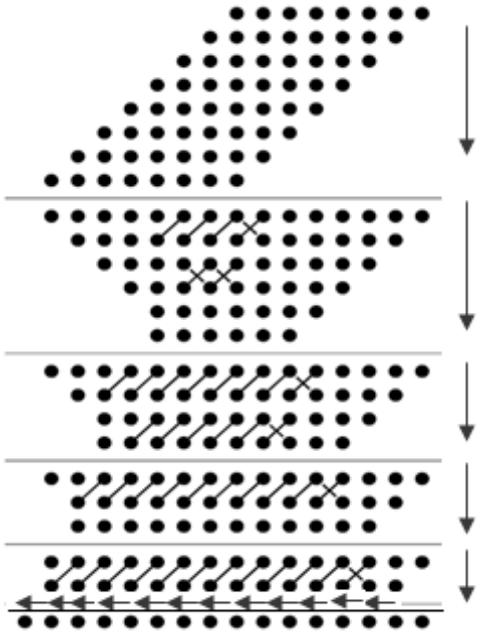

Fig. 9. Dot diagram for 8 by 8 Dadda Multiplier

The number of (2, 2) counters used in Dadda's reduction method equals N-1.The calculation diagram for an 8X8 Dadda multiplier is shown in figure 9.

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 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.

*D. Normalizing Unit:*

The result of the significand multiplication (intermediate product) must be normalizing. Having a leading '1' just immediate to the left of the decimal point (i.e. in the bit 46 in the intermediate product) is known as a normalized number. Since the inputs are normalized numbers then the intermediate product has the leading one at bit 46 or 47

1. No shift is needed the intermediate product is known to be a normalized number when the one is at bit 46 (i.e. to the left of the decimal point) .
2. The exponent is incremented by 1 if the leading one is at bit 47 then the intermediate product is shifted to the right.

Multiplexers are used to perform combinational shift logic for the shift operation.

## IV. UNDERFLOW/OVERFLOW PREDICTION

Underflow/Overflow means that the result's exponent is too small/large to be represented in the exponent field. The result of exponent must be 8 bits in size, and must be between 1 and 254 otherwise the value is not a normalized one. While adding the two exponents adding or during normalization the overflow may occur. Overflow due to exponent addition may be compensated during subtraction of the bias; resulting underflow that can never be compensated; if the intermediate exponent = 0 then during normalization it's an underflow that may be compensated by adding 1 to it.

If an overflow occurs an overflow signal is made high and the result turns to ±Infinity (sign determined according to the sign of the floating point multiplier inputs). If an underflow occurs an underflow signal goes high and the result turns to ±Zero (sign determined according to the sign of the floating point multiplier inputs). An underflow flag is raised after denormalized numbers are made to Zero with the appropriate sign calculated from the inputs. Assume that $E_1$ and $E_2$ are the exponents of the two numbers A and B respectively; the result's exponent is calculated by (6)

$$E_{result} = E_1 + E_2 - 127 \quad (6)$$

$E_1$ and $E_2$ can have the values from 1 to 254; resulting in $E_{result}$ having values from -125 (2-127) to 381 (508-127); but for normalized numbers, $E_{result}$ can only have the values from 1 to 254. Table III summarizes the $E_{result}$ different values and the effect of normalization on it.

TABLE III. NORMALIZATION EFFECT ON RESULT'S EXPONENT AND OVERFLOW/UNDERFLOW DETECTION

| $E_{result}$ | Category | Comments |
|---|---|---|
| $-125 \le E_{result} < 0$ | Underflow | Can't be compensated during normalization. |
| $E_{result} = 0$ | Zero | May turn to normalized number during normalization (by adding 1 to it) |
| $1 < E_{result} < 254$ | Normalized Number | May result in overflow during normalization. |
| $255 \le E_{result}$ | Overflow | Can't be compensated |

## V. MULTIPLIER PIPELINING

In order to enhance the performance of the multiplier, three pipelining stages are used to divide the critical path thus increasing the maximum operating frequency of the multiplier.

The pipelining stages are imbedded at the following locations:

1. Before the bias subtraction; in the middle of the significand multiplier and in the middle of the exponent adder.
2. After the significand multiplier and the exponent adder.
3. Sign, exponent and mantissa bits; at the floating point multiplier outputs.

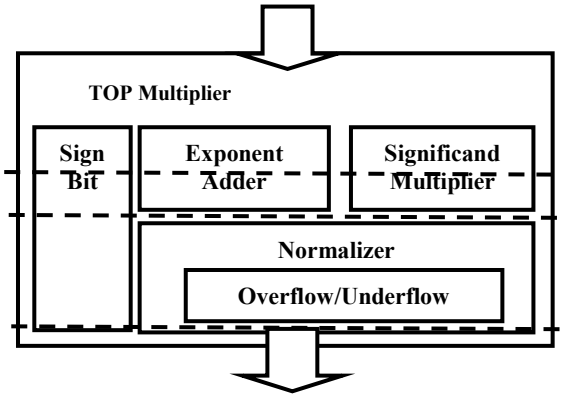


Fig. 10. Figure shows the pipelining stages as dotted lines.

The synthesis tool "retiming" option was used so that the synthesizer uses its optimization logic to better place the pipelining registers across the critical path.

## VI. IMPLEMENTATION AND TESTING

The whole multiplier (top unit) was tested against the Xilinx floating point multiplier core. Xilinx core was customized to have two flags to indicate overflow and underflow, and to have a maximum latency of three cycles. Xilinx core implements the "round to nearest" rounding mode.

A stimulus program is written and applied it to the implemented floating point multiplier and to the Xilinx core then results are compared. The floating point multiplier code was also checked using Xilinx13.4 [9]. The design was synthesized using Xilinx synthesis XST tool [9] and it is targeted on Virtex-5 FPGA (xc5vlx20t-2ff323). Table IV shows the improvement in the implemented floating point multiplier compared to Xilinx core.

*Simulation Results:*

The simulation results for corresponding inputs are shown in Fig.10. The simulation is done using Xilinx 13.4 [9]. Considering the random floating point numbers,

Inputs: a = 32'HC20C0000;
b = 32'H40B00000;
Output: result = 32'HC3408000;

Here the input 'a' is negative, input 'b' is positive, so the output result will be negative whose sign bit is '1'.

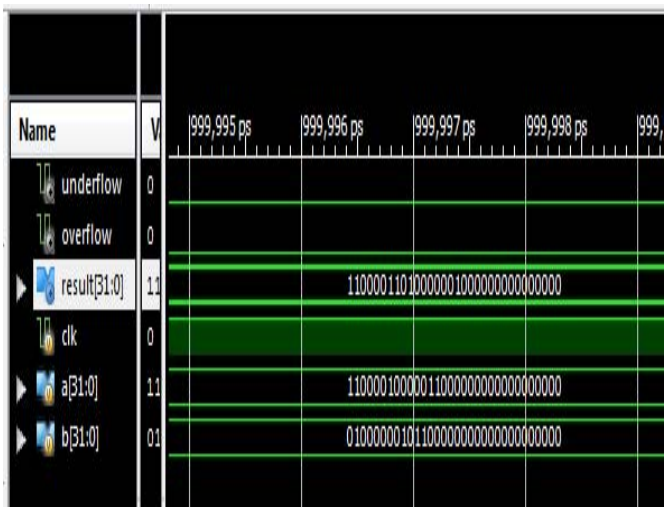


Fig. 11. Floating point multiplier Simulation

TABLE IV. AREA AND FREQUENCY COMPARISON BETWEEN THE IMPLEMENTED FLOATING POINT MULTIPLIER AND XILINX CORE

| | **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 |

## VII. CONCLUSION AND FUTUREWORK

This paper describes an implementation of a floating point multiplier using Dadda Multiplier that supports the IEEE 754-2008 binary interchange format; the multiplier is more precise because it doesn't implement rounding and just presents the significand multiplication result as is (48 bits). The significand multiplication time is reduced by using Dadda Algorithm. The design has been implemented on a Xilinx Virtex5 FPGA and achieved the speed of 526MHz.

## REFERENCES

[1] IEEE 754-2008, IEEE Standard for Floating-Point Arithmetic, 2008.

[2] Mohamed Al-Ashrfy, Ashraf Salem and Wagdy Anis "An Efficient implementation of Floating 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 FPGAs," Proceedings of 83 the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'96), pp. 107-116, 1996.

[6] Jaenicke and W. Luk, "Parameterized Floating-Point Arithmetic on FPGAs", Proc. of IEEE ICASSP, 2001, vol. 2, pp.897-900.

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

[8] B. Lee and N. Burgess, "Parameterisable Floating-point Operations on FPGA," Conference Record of the Thirty-Sixth Asilomar Conference on Signals, Systems, and Computers, 2002.

[9] Xilinx13.4, Synthesis and Simulation Design Guide", UG626 (v13.4) January 19, 2012.