International Journal of Engineering Research and Applications (IJERA) ISSN: 2248-9622
International Conference on Industrial Automation and Computing (ICIAC- 12-13th April 2014)

RESEARCH ARTICLE OPEN ACCESS

Efficient Design of Floating Point Matrix Calculations Using Vhdl

Prof. Nicky H. Bellani, Prof. Pratik R. Hajare
SBJITMR, Nagpur bellaninicky@gmail.com, pratik_hajare74@rediffmail.com

ABSTRACT-
The paper describes the efficient design of IEEE 754 single precision floating point matrix calculations. The system provides a catalog of efficient user customizable cores,designed for FPGA implementation,ranging in six different matrix calculations categories:(i) Matrix Transpose (ii) Matrix Addition (iii) Matrix Subtraction (iv) Matrix Determinant (v) Matrix Multiplication (vi) Matrix Inverse. The generated cores are application core for 2x2 Matrix calculations. In order to prove its legality, the developed algorithm is simulated using the Xilinx 9.2i and Quartus software.
Keywords: Matrix calculations, exponent, significand, floating point

I. INTRODUCTION

High Performance systems are required by the developers for fast processing of computationally intensive applications. Reconfigurable hardware devices in the form of Field Programmable Gate Arrays(FPGAs) have been proposed as viable system building blocks in the construction of high performance systems at an economical price. Given the importance and the use of matrix calculations in scientific computing and data processing applications,they seem ideal candidates to harness and exploit the advantages offered by FPGAs. Floating point numbers are one possible way of representing real numbers in binary format; the IEEE 754 [1] 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.

II. FLOATING POINT ARITHMETIC

2.1 Floating Point:

A computer word is divided into two parts, an exponent and a significand. As an example, an exponent of (−3) and significand of 1.5 might represent the number 1.5 × 2–3 = 0.1875. The advantages of standardizing a particular representation are obvious. The semantics of floating-point instructions are not as clear-cut as the semantics of the rest of the instruction set, and in the past the behaviour of floating-point operations varied considerably from one computer family to the next. The variations involved such things as the number of bits allocated to the exponent and significand, the range of exponents, how rounding was carried out, and the actions taken on exceptional conditions like underflow and over- flow. IEEE arithmetic differs from much previous arithmetic in the following major ways:

2.2 Floating Point Rounding:

1. When rounding a “halfway” result to the nearest floating-point number, it picks the one that is even.
2. It includes the special values NaN, ∞, and −∞.
3. It uses denormal numbers to represent the result of computations whose value is less than 1.0 × 2Emin.
4. It rounds to nearest by default, but it also has three other rounding modes.
5. It has sophisticated facilities for handling exceptions.

To elaborate on (1), when operating on two floating-point numbers, the result is usually a number that cannot be exactly represented as another floating- point number. For example, in a floating-point system using base 10 and two significant digits, 6.1 × 0.5 = 3.05.This needs to be rounded to two digits. Should it be rounded to 3.0 or 3.1? In the IEEE standard, such halfway cases are rounded to the number whose low-order digit is even. That is, 3.05 rounds to 3.0, not 3.1.

The standard actually has four rounding modes. The default is round to nearest, which rounds ties to an even number as just explained. The other modes are round toward 0, round toward +∞, and round toward –∞. We will elaborate on the other differences in following sections.

2.3 Special Values and Denormals:

Probably the most notable feature of the standard is that by default a computation continues in the face of exceptional conditions, such as dividing by 0 or taking the square root of a negative number. For example, the result of taking the square root of a negative number is a NaN (Not a Number), a bit pattern that does not represent an ordinary number. As an example of how NaNs might be useful.

In IEEE arithmetic, if the input to an operation is a NaN, the output is NaN (e.g., 3 + NaN = NaN). Because of this rule, writing floating-point subroutines that can accept NaN as an argument rarely requires any special case checks.

The final kind of special values in the standard are denormal numbers. In many floating-point systems, if Emin is the smallest exponent, a number less than 1.0 *2Emin cannot be represented, and a floating-point operation that results in a number less than this is simply flushed to 0. In the IEEE standard, on the other hand, numbers less than 1.0 *2Emin are represented using significands less than 1. This is called gradual underflow. Thus, as numbers decrease in magnitude below 2Emin, they gradually lose their significance and are only represented by 0 when all their significance has been shifted out. For example, in base 10 with four significant figures, let x = 1.234 *10Emin. Then x/10 will be rounded to 0.123 * 10Emin, having lost a digit of precision. Similarly x/100 rounds to 0.012 *10Emin, and x/1000 to 0.001 *10Emin, while x/10000 is finally small enough to be rounded to 0. Denormals make dealing with small numbers more predictable by maintaining familiar properties such as x =y=>x-y = 0.

2.4 Representation of Floating-Point Numbers:

Fig. 1 shows the IEEE 754 single precision binary format representation; it consists of a one bit sign (S), an eightbit exponent (E), and a twenty three bit fraction (M or Mantissa). An extra bit is added to the fraction to form what is called the 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 (1)

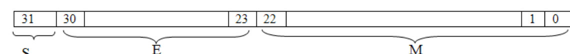

Figure: 1. IEEE single precision floating point format

Z = (-1S) * 2 (E - *Bias*) * (1.M) (1)

Where M = m22 2-1 + m21 2-2 + m20 2-3+…+ m1 2-22+ m0 2-23;

Bias = 127.

Multiplying two numbers in floating point format is done by 1- adding the exponent of the two numbers then subtracting the bias from their result, 2- multiplying the significand of the two numbers, and 3- 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 1 in the MSB ofthe result (leading one).

2.5 Floating-Point Addition / Subtraction

Floating-point addition is another essential step in matrix-vector multiplication. To perform the accumulation for matrix-vector multiplication a single precision floating point adder is used in this project. The algorithm for floating-point number addition is more complex than multiplication, as it involves more bit shifting and comparison. The floating-point adder performs calculations based on the algorithm described below :

1. Compare two numbers' exponent and keep the largest exponent.
2. Subtract exponents. Let d be the difference between two exponents
3. Align mantissas. Shift the mantissa to right by d bits. (Here the number that has a smaller exponent is the one that need to be shifted)
4. Add mantissas.
5. Test special case of the mantissa from the result. The exponent is set to -128 if the mantissa is zero.
6. Check for overflows and underflows.
7. Let k be the number of leading non-significant sign bits. The mantissa is left-shifted k bits. The exponent is subtracted by k.

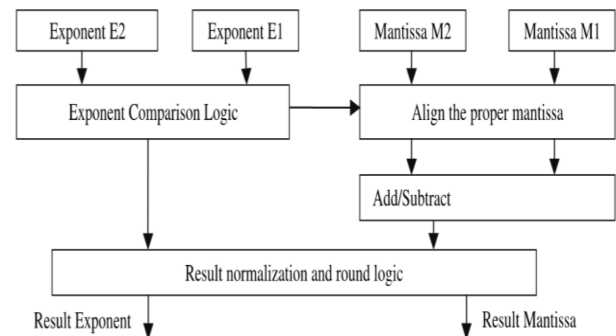

Figure:2 Floating Point Addition / Subtraction

2.6 Floating-Point Multiplication:

Floating-point multiplication is one of essential steps in matrix-vector multiplication, in this project a single precision floating-point multiplier is used. The floating-point multiplier performs calculations based on the algorithm described below:

1. Multiply mantissas.
2. Add exponents.
3. Test for special case of the mantissa. Set exponent to –128 if the mantissa equal to zero. If

normalization is needed, the mantissa is shifted right and exponent is increased accordingly.
4. Check for exponent overflow or underflow.
5. Check the sign bit, if both sign bit equal to 1 the result of the multiplication is positive. If one of the sign bit is 1 and the other is 0 then the result of the operation is negative.

The simplest floating-point operation is multiplication, so we discuss it first. A binary floating-point number x is represented as a significand and an exponent,

x = s*2e.

The formula

(s1 *2e1) • (s2 *2e2) = (s1 • s2) *2e1+e2

shows that a floating-point multiply algorithm has several parts. The first part multiplies the significands using ordinary integer multiplication. Because floating point numbers are stored in sign magnitude form, the multiplier need only deal with unsigned numbers (although we have seen that Booth recoding handles signed two's complement numbers painlessly). The second part rounds the result. If the significands are unsigned p-bit numbers (e.g., p = 24 for single precision), then the product can have as many as 2p bits and must be rounded to a p-bit number. The third part computes the new exponent. Because exponents are stored with a bias, this involves subtracting the bias from the sum of the biased exponents.
The interesting part of floating-point multiplication is rounding. Since the cases are similar in all bases, the figure uses human-friendly base 10, rather than base 2.
There is a straightforward method of handling rounding using the multiplier with an extra sticky bit. If p is the number of bits in the significand, then the A, B, and P registers should be p bits wide. Multiply the two significands to obtain a 2p-bit product in the (P,A) registers Using base 10 and p = 3, parts (a) and (b) illustrate that the result of a multiplication can have either 2p − 1 or 2p digits, and hence the position where a 1 is added when rounding up (just left of the arrow) can vary. Part (c) shows that rounding up can cause a carry-out.

a) 1.23
*6.78_______
8.3394 r=9>5 so round up rounds to 8.34.

b) 2.83
*4.47 _______
12.6501 r=5 and following digit !=0 so round up rounds to 1.27*101

P and A contain the product, case1 x0=0 shift needed, case2 x0=1 increment exponent.

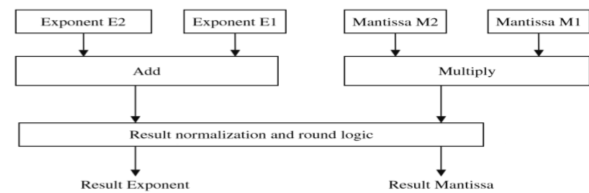

Figure: 3 Floating Point Multiplication

The top line shows the contents of the P and A registers after multiplying the significands, with p = 6. In case (1), the leading bit is 0, and so the P register must be shifted. In case (2), the leading bit is 1, no shift is required, but both the exponent and the round and sticky bits must be adjusted. The sticky bit is the logical OR of the bits marked s.

During the multiplication, the first p −2 times a bit is shifted into the A register, OR it into the sticky bit. This will be used in halfway cases. Let s represent the sticky bit, g (for guard) the most-significant bit of A, and r (for round) the second most-significant bit of A.
There are two cases:
1.The high-order bit of P is 0. Shift P left 1 bit, shifting in the g bit from A. Shifting the rest of A is not necessary.
2.The high-order bit of P is 1. Set s= s.v.r and r = g, and add 1 to the exponent.Now if r = 0, P is the correctly rounded product. If r = 1 and s = 1, then P + 1 is the product (where by P + 1 we mean adding 1 to the least-significant bit of P). If r = 1 and s = 0, we are in a halfway case, and round up according to the least significant bit of P. After the multiplication, P = 126 and A = 501, with g = 5, r = 0, s = 1. Since the high- order digit of P is nonzero, case (2) applies and r := g, so that r = 5, as the arrow indicates in Figure H.9. Since r = 5, we could be in a halfway case, but s = 1 indicates that the result is in fact slightly over 1/2, so add 1 to P to obtain the correctly rounded product. Note that P is nonnegative, that is, it contains the magnitude of the result.

2.6 Floating-Point Division

Division of a pair of FP numbers X ¼ mx * 2a and Y ¼ my * 2b is represented as X=Y ¼ (mx=my)* 2ab. A general algorithm for division of FP numbers consists of three basic steps:

1) Compute the exponent of the result by subtracting the exponents.
2) Divide the mantissa and determine the sign of the result.
3) Normalize and round the resulting value, if necessary.
4) Example Consider the division of the two FP numbers X ¼ 1.0000 * 222 and Y ¼21.0100 * 221.

1. Subtract exponents: 22 2 (21) ¼21.
2. Divide the mantissas: 1.0000 421.0100 ¼20.1101.
3. The result is 20.1101* 221.

Division of two FP numbers can be illustrated using the schematic shown in Figure.

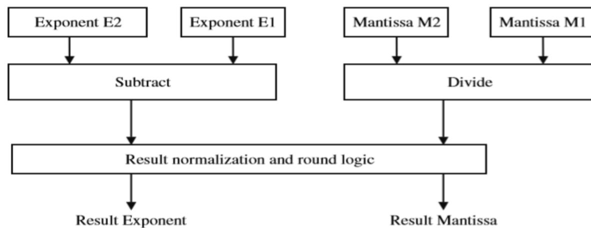

Figure: 4 Floating Point Division

III. RESULTS

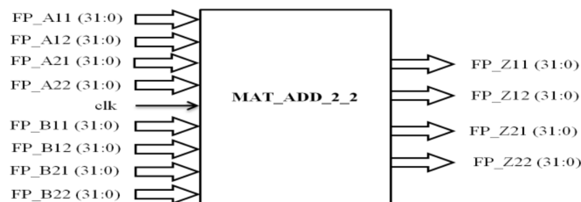

Figure 5: Block Diagram of 2x2 Matrix Addition

Figure 5 and 6 shows the RTL schematic and simulation results of 2x2 matrix addition with floating point numbers

INPUT :FP_A11	**0.24414062 => (00111110011110100000000000000000**
INPUT :FP_A12	0.24414062 => (00111110011110100000000000000000
INPUT :FP_A21	0.24414062 => (00111110011110100000000000000000
INPUT :FP_A22	0.24414062 => (00111110011110100000000000000000
INPUT :FP_B11	5.9127808E-5 => (00111000011110000000000000000000
INPUT :FP_B12	5.9127808E-5 => (00111000011110000000000000000000
INPUT :FP_B21	5.9127808E-5 => (00111000011110000000000000000000
INPUT :FP_B22	5.9127808E-5 => (00111000011110000000000000000000
OUTPUT :FP_Z11	0.24419975=>(0011111001111010000 0111110000000)$_2$
OUTPUT :FP_Z12	0.24419975=>(0011111001111010000 0111110000000)$_2$
OUTPUT :FP_Z21	0.24419975=>(0011111001111010000 0111110000000)$_2$
OUTPUT :FP_Z22	0.24419975=>(0011111001111010000 0111110000000)$_2$

Table1 : Results of 2x2 matrix addition

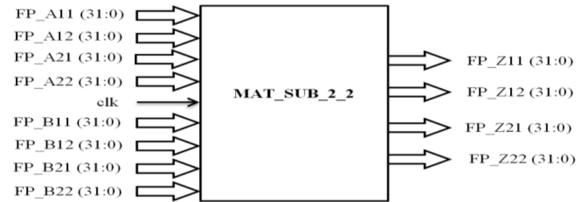

Figure 7: Block Diagram of 2x2 Matrix

SubtractionFigure 7 and 8 shows the RTL schematic and simulation results of 2x2 matrix subtraction with 32 bit floating point numbers.

INPUT :FP_A11	**0.24414062 => (00111110011110100000000000000000**
INPUT :FP_A12	0.24414062 => (00111110011110100000000000000000
INPUT :FP_A21	0.24414062 => (00111110011110100000000000000000
INPUT :FP_A22	0.24414062 => (00111110011110100000000000000000
INPUT :FP_B11	5.9127808E-5 => (00111000011110000000000000000000
INPUT :FP_B12	5.9127808E-5 => (00111000011110000000000000000000
INPUT :FP_B21	5.9127808E-5 => (00111000011110000000000000000000
INPUT :FP_B22	5.9127808E-5 => (00111000011110000000000000000000
OUTPUT	-0.2440834=> (10111110011110011111000100000000
OUTPUT	-0.2440834=> (10111110011110011111000100000000
OUTPUT	-0.2440834=> (10111110011110011111000100000000
OUTPUT	-0.2440834=> (10111110011110011111000100000000

Table2 : Results of 2x2 matrix subtraction

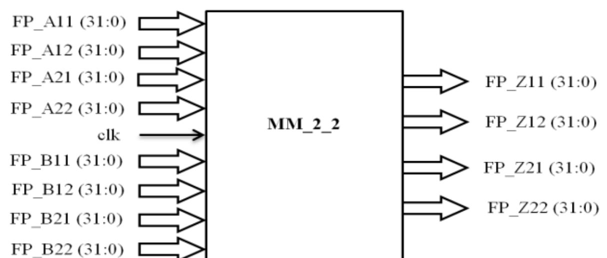

Figure 9: Block Diagram of 2x2 Matrix Multiplication

Figure 9 shows the RTL Schematic of 2x2 Matrix Multiplication. Figure 10 shows the Xilinx simulation results of 2x2 matrix multiplication in which the 4 elements of first 2x2 matrix are FP_A11, FP_A12, FP_A21, FP_A22 and the 4 elements of second 2x2 matrix are FP_**B**11, FP_**B**12, FP_**B**21, FP_**B**22. The result of this multiplication is assigned to four otput elements named FP_Z11, FP_Z12, FP_Z21 and FP_Z22.

INPUT :FP_A11	0.24414062 => (00111110011110100000000000000000
INPUT :FP_A12	0.24414062 => (00111110011110100000000000000000
INPUT :FP_A21	0.24414062 => (00111110011110100000000000000000
INPUT :FP_A22	0.24414062 => (00111110011110100000000000000000
INPUT :FP_B11	5.9127808E-5 => (00111000011110000000000000000000
INPUT :FP_B12	5.9127808E-5 => (00111000011110000000000000000000
INPUT :FP_B21	5.9127808E-5 => (00111000011110000000000000000000
INPUT :FP_B22	5.9127808E-5 => (00111000011110000000000000000000
OUTPU T	2.8871E-5 => (00110111111100100011000000000000
OUTPU T	2.8871E-5 => (00110111111100100011000000000000
OUTPU T	2.8871E-5 => (00110111111100100011000000000000
OUTPU T	2.8871E-5 => (00110111111100100011000000000000

Table:3 Result of 2x2 Matrix multiplication of floating point numbers

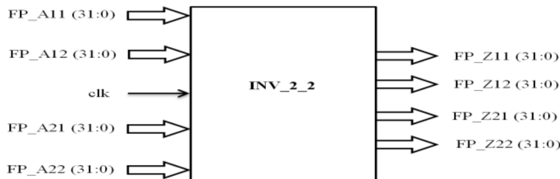

Figure 10: Block Diagram of 2x2 Matrix Inverse

Figure 10 and 11 shows the RTL schematic and simulation results of 2x2 matrix with floating point numbers which provides the inverse of a matrix.

INPUT :FP_A11	5.9127808E-5 => (00111000011110000000000000000000
INPUT :FP_A12	0.24414062 => (00111110011110100000000000000000
INPUT :FP_A21	0.24414062 => (00111110011110100000000000000000
INPUT :FP_A22	5.9127808E-5 => (00111000011110000000000000000000)
OUTPUT :FP_Z11	-9.920001E-4 =>(101110101000001000000110000000 00)
OUTPUT :FP_Z12	4.096=>(010000001000001100010010 0 1101111)$_2$
OUTPUT :FP_Z21	4.096=>(010000001000001100010010 0 1101111)$_2$
OUTPUT :FP_Z22	-9.920001E-4 =>(101110101000001000000110000000

Table4 : Results of 2x2 matrix Inverse

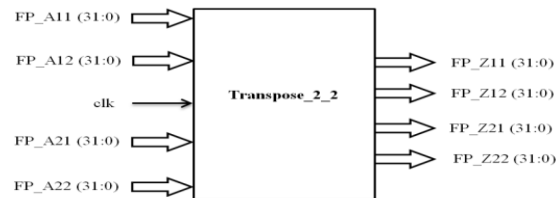

Figure 12: Block Diagram of 2x2 Transpose of a Matrix

Figure 12 and 13 shows the respective RTL schematic and simulation results of 2x2 matrix with floating point which is producing the transpose of a 2x2 Matrix at the output.

INPUT :FP_A1	5.9127808E-5=>(00111000011110000000000000000000
INPUT :FP_A1	5.9127808E-5=>(00111000011110000000000000000000
INPUT :FP_A2	0.24414062 => (00111110011110100000000000000000
INPUT :FP_A2	0.24414062 => (00111110011110100000000000000000
OUTPU T	5.9127808E-5=>(00111000011110000000000000000000
OUTPU T	0.24414062 => (00111110011110100000000000000000
OUTPU T	5.9127808E-5=>(00111000011110000000000000000000
OUTPU T	0.24414062 => (00111110011110100000000000000000

Table5 : Results of 2x2 matrix transpose

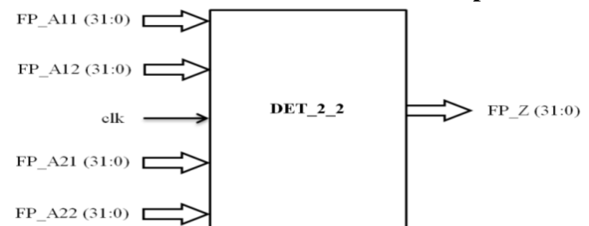

Figure 14: Block Diagram of Determinant of a Matrix

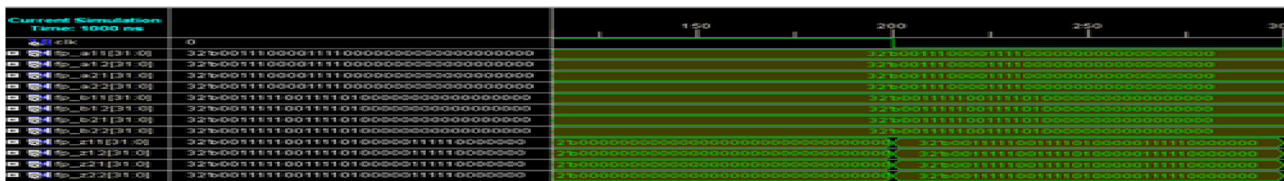

Figure 6: Simulation Results for Floating point 2*2 Matrix Addition

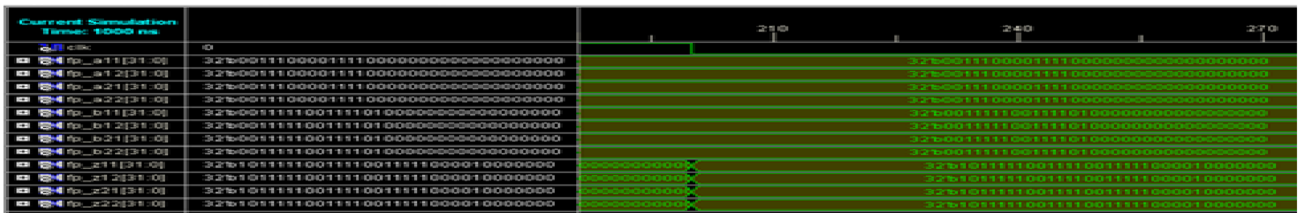

Figure 8: Simulation Results for Floating point 2*2 Matrix Subtraction

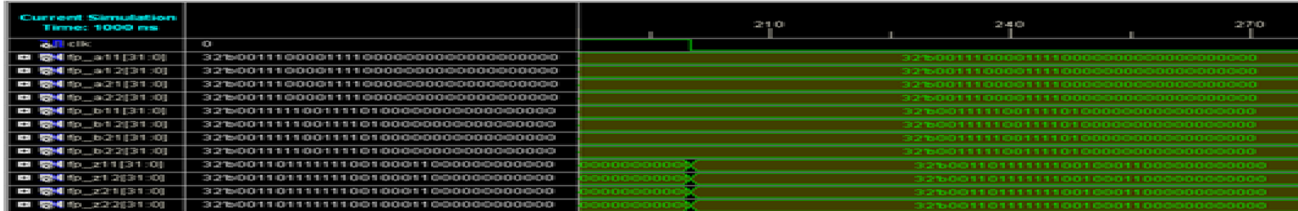

Figure 10: Simulation Results for Floating point 2*2 Matrix Multiplication

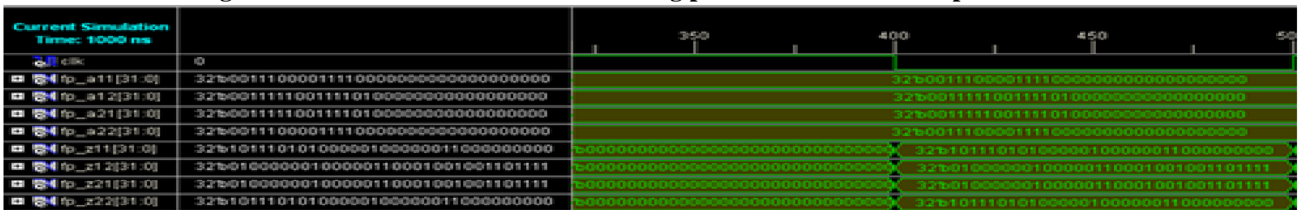

Figure 11: Simulation Results for Inverse of Matrix

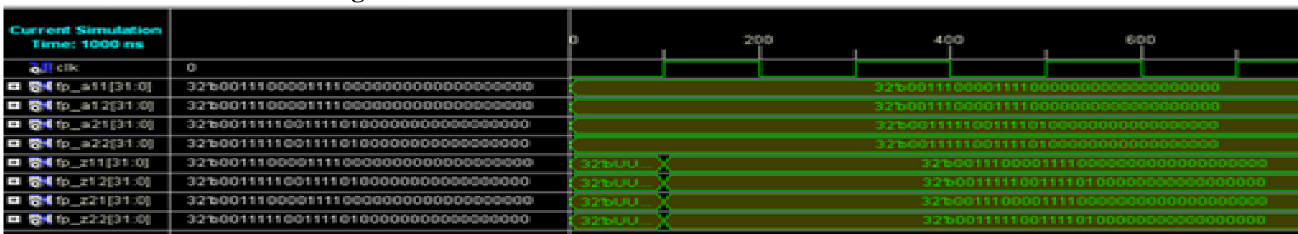

Figure 13: Simulation Results for Floating point Matrix Transpose

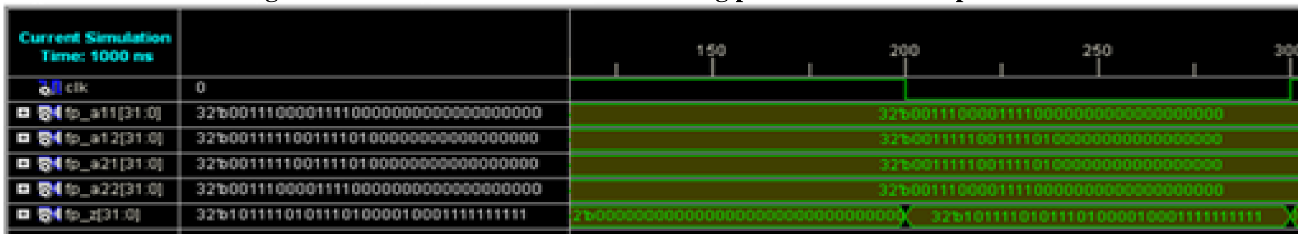

Figure 15: Simulation Results for Floating point Matrix Determinant

Figure 14 and 15 shows the respective RTL schematic and simulation results of 2x2 matrix with floating point which is producing the determinant of a 2x2 Matrix at the output FP_Z.

INPUT :FP_A11	5.9127808E-5 => (001110000111100000000000000000000
INPUT :FP_A12	0.24414062 => (001111100111101000000000000000000
INPUT :FP_A21	0.24414062 => (001111100111101000000000000000000
INPUT :FP_A22	5.9127808E-5 => (001110000111100000000000000000000
OUTPU T :FP_Z	-0.05960464 => (10111101011101000010001111111111

Table6 : Results of 2x2 matrix Determinant

IV. CONCLUSION

This project describes a system for matrix algorithm cores generation used in image processing applications. The system provides a catalogue of user-customizable cores ranging in three different matrix algorithm categories: (i) matrix operations, (ii) matrix transforms and (iii) matrix decomposition. The system includes a GUI to help the users customize the cores to be generated to meet the requirements of their applications.

REFERENCES

[1]. Faycal Bensaali, Abbes Amira, Reza Sotudeh, A General Framework for Efficient FPGA Implementation of Matrix Product, University of Hertfordshire, Brunel University, West London.

[2]. Syed M. Qasim, Ahmed A. Telba, Abdulhameed Y. Almazroo, FPGA Design and Implementation of Matrix Multiplier Architectures for Image and Signal Processing Applications : IJCSNS, VOL 10 No. 2, Feb. 2010.

[3]. Michael deLorimier, Andre deHon, Floating-Point Sparse Matrix-Vector Multiply for FPGAs, Dept. of CS, 256-80, California Institute of Technology, Pasadena, CA 91125.

[4]. C. S.Wallace, “A suggestion for fast multipliers,” IEEE Trans. Electron. Comput., no. EC-13, pp. 14–17, Feb. 1964.

[5]. M. R. Santoro, G. Bewick, and M. A. Horowitz, “Rounding algorithms for IEEE multipliers,” in Proc. 9th Symp. Computer Arithmetic, 1989, pp. 176–183.

[6]. D. Stevenson, “A proposed standard for binary floating point arithmetic,” IEEE Trans. Comput., vol. C-14, no. 3, pp. 51-62, Mar.

[7]. Naofumi Takagi, HirotoYasuura, and Shuzo Yajima. “High-speed VLSI multiplication algorithm with a redundant binary addition tree”. IEEE Transactions on Computers, C-34(9), Sept 1985.

[8]. Hough, D., "Applications of the Proposed IEEE 754 Standard for Floating-Point Arithmetic", *Computer*, Vol. 13, No. 1, Jan. 1980, pp 70-74.