Abstract— this paper proposes a design for a multiplier which can calculate complex floating point numbers of 32 bits using Vedic multiplication method. For multiplication, the Urdhva Tiryagbhyam sutra from Vedic mathematics is applied. The use of the Vedic mathematics and their application to the multiplier make certain significant reduction of propagation delay, successively improves speed. The design can handle underflow and overflow cases. The inputs to the multiplier are provided in IEEE 754, 32 bit binary format.

Key words: Complex Floating Point Numbers, Urdhva Tiryagbhyam, Vedic Mathematics, IEEE 754

I. INTRODUCTION

Floating Point number system is widely used in digital signal processing, in the field of medical imaging, image processing, biometrics, microprocessor, motion capture, audio applications, and multimedia [1] [2]. Floating point numbers are real valued numbers which has its own representation and standardization. The proposed architecture efficiently computes complex number in floating point format in order to perform multiplication, for wide range of numbers. The main focus is to improve the speed of the floating point multiplier. Therefore Vedic mathematics approaches are applied for multiplication.

A. IEEE 754 format:

An IEEE 754 format is a way in which a very large or very small numbers can be represented in binary format with high precision. It constitutes a set of representation of numerical values and symbols. It comprises of arithmetic formats, interchange formats, rounding rules, operations and exception handling. The Binary Floating point numbers are framed in single and double formats. The IEEE 754 standard describes a Single precision format with 3 sectors, that is 1 bit for showing the sign bit, 8 bit for showing the exponent part and 24 bit for showing mantissa , in which 23 bit are explicitly stored and 1 bit is hidden[3].

![Fig. 1: IEEE Format for single precision and double precision](image)

B. Urdhva tiryagbhyam Sutra

Sri Bharathi Krishna Tirthaji compiled 16 sutras from the ancient Vedic scripts in Sanskrit. The Urdhva tiryagbhyam sutra can be used for multiplication of any numbers. In Sanskrit urdhva means ‘vertically’ and tiryagbhyam means ‘crosswise’. [4]. Working of this sutra can be explained by considering an example. Consider three bit numbers let A=a2a1a0 and B=b2b1b0[5].

1) The least significant bit of A is multiplied with the least significant bit of B giving s0=a0b0;
2) Then a0 is multiplied with b1, and b0 is multiplied with a1 and the result are added together as: c1s1=a1b0+a0b1; Here c1 is carry and s1 is sum.
3) Next step is to add c1 with the multiplication results of a0 with b2, a1 with b1 and a2 with b0. c2s2=c1+a2b2+a1b1 + a0b2;
4) Next step is to add c3 with the multiplication results of a1 with b2 and a2 with b1. c3s3=c2+a1b2+a2b1;
5) Similarly the last step: c4s4=c3+a2b2;
6) The last stage of multiplication of A and B produces c4s4s3s2s1s0

![Fig. 2 The Urdhva Tiyagbhyam Multiplication](image)

II. PROPOSED ARCHITECTURE

The complex floating point multiplier can be described in 3 units.
Floating point multiplier
Floating point adder
Floating point subtractor
Multiplication of complex floating point can be represented by the equation as follows:

\[(A+B_i)*(C+D_i) = (A*C-B*D) + (A*D + B*C)i\]

Where \((A+B_i)\) & \((C+D_i)\) are complex numbers.

![Fig. 3: Proposed architecture for complex floating point multiplier](image)

A. Floating Point Multiplier

The floating point multiplier can be divided into three units. The first unit performs mantissa calculation. It uses 24 bits
starting from the least significant bit of 32 bit binary single precision format. In this paper we present a design for mantissa calculation which works on the principle of Vedic math. Urdhva Tiryagbyham sutra is most appropriate, since we have to perform multiplication of 24X 24 bits. In this method the partial products are computed simultaneously reducing the delay and thus increasing the computation speed of the algorithm. The 24 bit Vedic multiplier gives 48 bit.[6]

The second unit is the exponent unit in which the 8 bits starting from 23 bit to 30 bit from both the inputs are added using an adder. Before addition the exponents are represented in excess 127 format. After addition the result is biased to excess 127 form. The third unit is the sign calculation unit where the most significant bits of both the inputs are XORed to identify the sign of the result. If the XORing result in logic 1 means the resultant float point number is negative else if the resultant value is 0 then the number is positive.

The final result is obtained by combining the result of three units and hiding the most significant bit 1 from the Vedic multiplier. The result of sign calculation unit gives the most significant bit that is the 31 bit of the result indicating the sign of the floating point number. The exponent unit gives the exponent of the result and mantissa unit gives the mantissa part of the resultant floating point number.[7]

5) Step5. Normalize the mantissa (that is by hiding the most significant bit 1).
6) Step6. Arrange the result in IEEE 754 single precision format. Sign of the result will be same as the input p.
7) For floating point subtraction, invert the sign bit of the number to be subtracted.

C. Complex Floating Point Multiplier
Complex floating point multiplier uses the equation

\[(A+Bi) \times (C+Di) = (A*C - B*D) + (A*D + B*C)i\]

Where \((A+Bi)\) & \((C+Di)\) are complex numbers.

The output of floating point multipliers are given to the floating point adder and subtractor. \((A*C - B*D)\) will give the real part of the resultant complex floating point number and \((A*D + B*C)i\) gives the imaginary part of the resultant complex floating number.

III. RESULT
We have stimulated the VHDL code for complex floating point multiplier on vertex E. The result obtained from the synthesis report is shown below in table no1. The parameters such as number of slices, number of slice flip flop, GCLKs are outline in the synthesis report.

![Fig. 4: Floating point multiplier](image)

Fig. 4: Floating point multiplier

B. Floating Point Adder and Subtractor
1) Addition algorithm
Consider two inputs p and q, where p and q are two single precision floating point numbers. Each input has three fields sign, exponent, and mantissa. For input p, ps represents the sign bit, pe represents the exponent field and pm represents the mantissa field and for input q, qs represent the sign bit, qe represents the exponent field and qm represents the mantissa field.[8]

1) Step1. Check whether pe > qe, if yes proceed to the next step else swap p and q.
2) Step2. Then calculate the difference between pe and qe. If the difference is zero then there is no need to shift and if the difference is not zero then shift the mantissa of q that is qm to the right, with the same amount as the difference.
3) Step3. Amount of shifting is added to the exponent qe. Now the exponent pe and exponent qe becomes equal.
4) Step4. Check the sign of p and q, if they are same then add pm and qm else subtract pm and qm.

### Table 1: Device Utilization Summary

<table>
<thead>
<tr>
<th>Logic Utilization</th>
<th>Used</th>
<th>Available</th>
<th>Utilized</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Slice Registers</td>
<td>186</td>
<td>19200</td>
<td>0%</td>
</tr>
<tr>
<td>Number of Slice LUTs</td>
<td>5695</td>
<td>19200</td>
<td>29%</td>
</tr>
<tr>
<td>Number of fully used Bit Slices</td>
<td>184</td>
<td>5697</td>
<td>3%</td>
</tr>
<tr>
<td>Number of bonded IOBs</td>
<td>194</td>
<td>220</td>
<td>88%</td>
</tr>
<tr>
<td>Number of BUFG/BUFGCTRLs</td>
<td>6</td>
<td>32</td>
<td>18%</td>
</tr>
</tbody>
</table>

### Table 2: timing report

<table>
<thead>
<tr>
<th>Minimum period</th>
<th>No path found</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum input arrival time before clock</td>
<td>22.605ns</td>
</tr>
<tr>
<td>Maximum output required time after clock</td>
<td>6.486ns</td>
</tr>
<tr>
<td>Maximum combinational path delay</td>
<td>13.403ns</td>
</tr>
</tbody>
</table>

IV. CONCLUSION AND FUTURE WORK
We have described a way of implementing complex floating point multiplier on virtexE family which causes increase in speed and consume much less chip area. Further, the VHDL code for implementation of complex floating point multiplier can be optimized and then routed on Virtex 6 for further decrease in combinational delay and less consumption of chip area. In future, these modules can be converted to IEEE-754 double-precision format. The implementation of these units will ease the implementation of computationally intense scientific applications

REFERENCES
[2] Zaher Baidas, Andrew D. Brown, Senior Member, IEEE, and Alan Christopher Williams, Member, IEEE, “Floating-Point Behavioral Synthesis”, IEEE


