VHDL IMPLEMENTATION OF FLOATING POINT MULTIPLIER USING VEDIC MATHEMATICS

I.V.VAIBHAV1, K.V.SAICHARAN1, B.SRAVANTHI1, D.SRINIVASULU2

1Students of Department of ECE, SACET, Chirala, AP, India
2Associate Professor of the Dept ECE, SACET, Chirala, AP, India

E-mail: innamurivaibhav1@gmail.com, kota.saicharan@gmail.com, sravanthi.408b@gmail.com, seenu_dasari@rediffmail.com

Abstract: This project presents a binary floating point multiplier based on Vedic algorithm. To improve power efficiency a new algorithm called URDHVA-TRIYAKBHYAM has been implemented for 24 X 24 bit multiplier design. By using this approach number of components can be decreased and complexity of hardware circuit can be decrease. In this project, Vedic multiplication technique is used to implement IEEE 754 floating point multiplier. The Urdhava-Triyakhym sutra is used for the multiplication of Mantissa i.e., 24x24 bits. The sign bit of the result is calculated using one XOR gate and a Carry Save Adder is used for adding the two biased Exponents. The underflow and overflow cases are handled. The inputs to the multiplier are provided in IEEE 754, 32 bit format. The multiplier is implemented in VHDL and Spartan3E FPGA is used. The result consists of 32 bit binary number of which MSB represent sign, next 8 bit represents Exponent and remaining 23 bits represents Mantissa.

Keywords: Vedic Mathematics, Urdhva-Triyakhbyam sutra, Floating Point multiplier, Field Programmable Gate Array (FPGA), Carry Save Adder

I. INTRODUCTION:

Floating-point operations are useful for computations involving large dynamic range, but they require significantly more resources than integer operations. The rapid advance in Field-Programmable Gate Array (FPGA) technology makes such devices increasingly attractive for implementing floating-point arithmetic. FPGAs offer reduced development time and costs compared to application specific integrated circuits, and their flexibility enables field upgrade and adaptation of hardware to run-time conditions. Our main aims of designing floating point units (FPUs) for reconfigurable hardware implementation are: (a) to parameterise the precision and range of the floating-point format to allow optimizing the FPUs for specific applications, and (b) to support optional inclusion of features such as gradual underflow and rounding. An approach meeting these aims will achieve effective design trade-off between performance and required resources. The Binary Floating - Point representation is in two formats i.e., i) Single Precision ii) Double Precision.

![Single Precision Representation](image1)

![Double Precision Representation](image2)

**Figure I. Precision Representations**
The following Figure I represents the format of the IEEE 754 standard Single and Double Precision structure. In the Single Precision it contains 32 bits, and the Mantissa has the length 23 bits and 1 bit is added to the MSB for normalization, Exponent has the length of 8 bits which is biased to 127, actually the Exponent is represented in excess 127 bit format and MSB of Single is reserved for Sign bit.

If the sign bit is ‘1’ it represents that the number is negative and when the sign bit is ‘0’ it represents that the number is positive. In the Double Precision it contains 64 bits and the Mantissa has the length 52 bits and 1 bit is added to the MSB for normalization, the Exponent has the length of 11 bits which is biased to 1023 and the MSB of Double is reserved for sign bit.

Multiplication of two floating point numbers represented in IEEE 754 format is done by multiplying the normalized 24 bit mantissa, adding the biased 8 bit exponent and resultant is converted in excess 127 bit format and for the biasing result we are using Carry Save Adder, for the sign calculation the input sign bits are XORed.

II. FLOATING POINT MULTIPLIER

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) In the middle of the significand multiplier, and in the middle of the exponent adder (before the bias subtraction).
2) After the significand multiplier, and after the exponent adder.
3) At the floating point multiplier outputs (sign, exponent and mantissa bits).

Three pipelining stages mean that there is latency in the output by three clocks. 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.

The exponent can only represent positive numbers 0 through 255. To represent both positive and negative exponents, a fixed value, called a bias, is subtracted from the exponent field to obtain the true exponent. Bias-127 exponent, \( e = E + \text{bias} \): This gives us an exponent range from \( E_{\text{min}} = -126 \) to \( E_{\text{max}} = 127 \). The mantissa is the set of 0’s and 1’s to the right of the radix point of the normalized (when the digit to the left of the radix point is 1) binary number.

III. MULTIPLICATION STEPS OF URDHVA-TRIYAKBHYAM

The Vedic multiplication system is based on 16 Vedic sutras or aphorisms, which describes natural ways of solving a whole range of mathematical problems. Out of these 16 Vedic Sutras the Urdhva-triyakbhyam sutra is suitable for this purpose. In this method the partial products are generated simultaneously which itself reduces delay and makes this method fast. The method for multiplication of two, 3 BITS number. Consider the numbers A and B where \( A = a_2a_1a_0 \) and \( B = b_2b_1b_0 \). The LSB of A is multiplied with the LSB of B: \( s_0 = a_0b_0 \); Then \( a_0 \) is multiplied with \( b_1 \), and \( b_0 \) is multiplied with \( a_1 \) and the results are added together as:

\[
c_1s_1 = a_1b_0 + a_0b_1;
\]

Here \( c_1 \) is carry and \( s_1 \) is sum. Next step is to add \( c_1 \) with the multiplication results of \( a_0 \) with \( b_2 \), \( a_1 \) with \( b_1 \) and \( a_2 \) with \( b_0 \).

\[
c_2s_2 = c_1a_2b_0 + a_1b_1 + a_0b_2;
\]

Next step is to add \( c_3 \) with the multiplication results of \( a_1 \) with \( b_2 \) and \( a_2 \) with \( b_1 \).

\[
c_3s_3 = c_2a_1b_2 + a_2b_1;
\]

Similarly the last step
\[ c_4 s_4 = c_3 + a_2 b_2; \]

Now the final result of multiplication of A and B is 
\[ c_4 s_4 s_3 s_2 s_1 s_0. \]

### Algorithm Steps:
1. Multiplying the significand \((1.M1 \times 1.M2)\)
2. Placing the decimal point in the result
3. Adding the exponents \(i.e., (E1 + E2 - \text{Bias})\)
4. Obtaining the sign; \(i.e., s_1 \text{ XOR } s_2\)
5. Normalizing the result; \(i.e., \text{ obtaining 1 at the MSB of the result's significand.}\)
6. Checking for underflow/overflow occurrence

The result of the significand multiplication (intermediate product) must be normalized to have a leading '1' just to the left of the decimal point \(i.e., \text{ in the bit 46 in the intermediate product.}\) Since the inputs are normalized numbers then the intermediate product has the leading one at bit 46 or 47

1. If the leading one is at bit 46 \(i.e., \text{ to the left of the decimal point}\) then the intermediate product is already a normalized number and no shift is needed.

2. If the leading one is at bit 47 then the intermediate product is shifted to the right and the exponent is incremented by 1.

The shift operation is done using combinational shift logic made by multiplexers. Fig shows a simplified logic of a Normaliser that has an 8 bit intermediate product input and a 6 bit intermediate exponent input.

After addition the result is again biased to excess 127 bit code. For this purpose 127 is subtracted from the result. Two's complement subtraction using addition is incorporated for this purpose. If \(ER\) is the final resultant exponent then, \(ER = EA + EB - 127\) where \(EA\) and \(EB\) are the exponent parts of operands \(A\) and \(B\) respectively. In this case \(ER = 10000110\).

### Mantissa Multiplication
Mantissa multiplication is done using the 24 bit Vedic Multiplier. The mantissa is expressed in 23 bit which is normalized to 24 bit by adding a 1 at MSB.

The block diagram of 24x24 BIT multiplier is shown in Figure 5. This multiplier is modelled using structural style of modelling using VHDL. In this paper first a 3x3 Vedic multiplier is implemented using the above mentioned method.

The 6x6 is designed using four 3x3 multipliers. After that the 12x12 is implemented using four 6x6 BIT multipliers. Finally the 24x24 BIT multiplier is made using four 12x12 BIT multipliers. The 24x24 BIT multiplier requires four 12x12 BIT multipliers and two 24 BIT ripple carry adders and 36 BIT ripple carry adders.

### A 24*24 Vedic multiplier is design by using four
12*12 Vedic multipliers based urdhva Triyakbhyam.

Here first block 12X12 multiplier consists of lower 12 bits of \(x\) \(i.e., x(11 \text{ down to 0})\) and \(y(11 \text{ down to 0})\), second block 12*12 Vedic multiplier inputs are \(x(23 \text{ down to 12})\) and \(y(11 \text{ down to 0})\) out off 24 bit output of first block lower adder 12 bits are separated and higher order bits are appended as 12 lower bits in front augmented with “000000000000” now second block 24 bits and above 24 generate bits are added and 24 bits sum is generated using 24 bit ripple carry adder.

Higher order 12 bit of \(y(23 \text{ down to 12})\) and lower order 12 bits of \(x\ i.e., x(11 \text{ down to 0})\) are multiplier and appended infront with “000000000000” to make 36 data similarly both higher order 12 bits of \(x\) and \(y\)
VHDL Implementation of Floating Point Multiplier Using Vedic Mathematics

The number of LUTs and slices required for the Vedic Multiplier is less and due to which the power consumption is reduced.

Also the repetitive and regular structure of the multiplier makes it easier to design. And the time required for computing multiplication is less than the other multiplication techniques.

An Overflow or Underflow case occurs when the result Exponent is higher than the 8 BIT or lower than 8 BIT respectively. Overflow may occur during the

The above two 36 bits are added to generate a 36 bit data in the right side resultant 36 bits are added to generate final 36 bits the result of 24\times24 multiplier is 48 bits consists of 36 higher bits from 36 bit adder output and lower order 12 bits 36+12=48 bits.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{architecture.png}
\caption{Architecture for Floating point Multiplier.}
\end{figure}
addition of two Exponents which can be compensated at the time of subtracting the bias from the exponent result. When overflow occurs the overflow flag goes up. The under flow can occur after the subtraction of bias from the exponent, it is the case when the number goes below 0 and this situation can be handled by adding 1 at the time of normalization. When the underflow case occur the under flow flag goes high.

**Example of 32 bit Multiplier**
The Mantissa Calculation Unit requires a 24 bit multiplier if 32 bit single IEEE 754 format is considered. In this paper we propose the Multiplication of two, 24 bit mantissa is done using the Vedic Multiplier. In this case 48 bit result obtained after the multiplication of mantissa is 25.375----010000011100101100000000000000000000000000000000

00000

Mantissa Output----0 10001000 01000001111100100000000000000000000000000000

This result is deduced as

\[
A \times B = 25.375 \times 25.375 = (643.890625)_{10} = (1010000011.111001)_{2}
\]

**CONCLUSION AND RESULT**
The multiplier is designed in VHDL and simulated using I- Simulator. The design is synthesized using Xilinx ISE 12.1 tool targeting the Xilinx Spartan3E FPGA. A test bench is used to generate the stimulus and the multiplier operation is verified. The over flow and under flow flags are incorporated in the design in order to show the over flow and under flow cases. The result shows the Vedic multiplication method is the efficient way of multiplying two floating point numbers. The lesser number of LUTs shows that the hardware requirement is reduced, thereby reducing the power consumption. The power is reduced affectively still not compromising delay so much.

This project presents an implementation of a floating point multiplier that supports the IEEE 754-32 format; the multiplier doesn’t implement rounding and just presents the significant multiplication result as is (48 bits); this gives better precision if the whole 48 bits are utilized in another unit. The ripple carry addition design has been implemented on a Xilinx Spartan3E FPGA.

The below table shows the summary of the multiplier tested:

<table>
<thead>
<tr>
<th>Parameters</th>
<th>VHDL</th>
<th>IHZ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device</td>
<td>Spartan 3E</td>
<td>Spartan 3E</td>
</tr>
<tr>
<td>Power consumption</td>
<td>11.9mW</td>
<td>29.7mW</td>
</tr>
<tr>
<td>Time delay</td>
<td>81.2ns</td>
<td>12.4ns</td>
</tr>
<tr>
<td>Number of LUTs</td>
<td>974</td>
<td>1012</td>
</tr>
<tr>
<td>Power delay</td>
<td>100.9pJ</td>
<td>122.5pJ</td>
</tr>
</tbody>
</table>

Table 1: Comparison of results.

**REFERENCES**


