Area-Efficient Architectures for Large Integer and Quadruple Precision Floating Point Multipliers

Manish Kumar Jaiswal, Student Member, IEEE
Department of Electronic Engineering,
City University of Hong Kong, Hong Kong.
E-mail: mkjaiswal2@student.cityu.edu.hk

Ray C.C. Cheung, Member, IEEE
Department of Electronic Engineering,
City University of Hong Kong, Hong Kong.
E-mail: r.cheung@cityu.edu.hk

Abstract—Large integer multiplication and floating point multiplication are the two dominating operations for many scientific and cryptographic applications. Large integer multipliers generally have linearly but high area requirement according to a given bit-width. High precision requirements of a given application lead to the use of quadruple precision arithmetic, however its operation is dominated by large integer multiplication of the mantissa product. In this paper, we propose a hardware efficient approach for implementing a fully pipelined large integer multipliers, and further extending it to Quadruple Precision (QP) floating point multiplication. The proposed design uses less hardware resources in terms of DSP48 blocks and slices, while attaining high performance. Promising results are obtained when compared our designs with the best reported large integer multipliers and also QP floating point multiplier in literatures. For instance, our results have demonstrated a significant improvement for the proposed QP multiplier, for over 50\% improvement in terms of the DSP48 block usage with a penalty of slight additional slices, when compared to the best result in the literature on a Virtex-4 device.

Keywords—Large Integer Multiplier, FPGA, Quadruple Precision Arithmetic, Karatsuba Multiplication, High Performance Reconfigurable Computing, Cryptographic Arithmetics.

I. INTRODUCTION

Integer multipliers are the basic building blocks of many application areas, such as numerical processing, cryptography, and also as the main building block of floating point multiplication. Despite many advancement and many implementation strategies, the area requirements and performance numbers of this arithmetic operation is a bottleneck, specially when its size increases. Floating point number system is a common choice for many scientific computations due to its wide dynamic range feature. For instance, floating point arithmetic is widely used in many areas, especially in scientific computation, numerical processing [1], image processing [2], communications [3]. The single precision and double precision floating point arithmetic processing units are quite general on the current CPUs, GPUs and DSP processors. Increased precision requirements, leads to the switching from single precision to double precision in many application areas. A higher precision requirement than the double precision is needed, thus we can see that the use of quadruple precision arithmetic operation in many new applications [4], [5]. In this point of view, the IEEE-754 standard has introduced a higher precision, the 128-bit quadruple precision, in 2008 the revision of floating point format [6]. Still, the direct hardware support for the quadruple precision arithmetic is not yet available in the general purpose processors.

Very limited work is available in the literature for the quadruple precision arithmetic. [7], [8] have reported the ASIC implementations of quadruple precision arithmetic. A recent work [5] has reported the implementation of quadruple precision multiplier on FPGA platform. Obviously, our proposed work on efficient implementation of large integer multipliers and the quadruple precision floating point multiplication, is very desirable for FPGA as well as ASIC platforms.

In this work, we have proposed designs for large integers multipliers, and it’s generalization for even more larger integer multipliers. Further, the paper has extended their use for the mantissa multiplication of quadruple precision floating point numbers. The design has used “Karatsuba Multiplication” technique [9], a well-known multiplication theory, along with efficient use of FPGA resources. Implementation of the multipliers leads to effective use of DSP48 blocks on the FPGA along with other logic resources and promising performance. We have used Xilinx ISE synthesis tool, ModelSim simulation tool, and Xilinx Virtex-4 FPGA platforms for implementations and for result comparisons. The summarized key contributions of the presented work are as follows:

- Hardware efficient design of large integer multipliers and its generalization for even more larger integer multipliers.
- Extended above idea for Quadruple Precision floating point multiplier, which has shown significant improvement compared to prior designs in literature.
- Hardware evaluations of proposed designs on the FPGA platform.

Paper organization is as follows. Section II discusses the basic background and underlying approach for our designs. Section III presents the designs of several large integer multipliers, its generalization and the implementation of quadruple precision multiplier. Section IV gives the implementation details along with the comparisons with previously reported implementations, and finally concluded the paper in Section V.

II. BACKGROUND AND UNDERLYING APPROACH

The presented work has basically targeted the large integer multipliers with its implication on quadruple precision mantissa multiplication, which leads to the implementation of
QP floating point multiplier operation. The basic underlying concept used here for large integer multiplication is the Karatsuba multiplication technique [9], along with the efficient use of FPGA resources. Karatsuba Multiplication is a fast multiplication algorithm. It reduces the multiplication of two n-digit numbers from simple \( n^2 \) to at most \( 3n \log_2 3 \approx 3n^{1.585} \) single digit multiplication. The basic steps for this algorithm depend on the divide-and-conquer paradigm and proceed in the following ways.

Let \( W \& X \) be the two n-digit numbers. By breaking these number in two parts, for some base \( B \), we can rewrite them as, \( W = W_1.B^m + W_0 \) and \( X = X_1.B^m + X_0 \) where \( W_0 & X_0 \) are of \( m \)-digit. Now, we can write the product of \( W \& X \) as follows,

\[
WX = W_1.X_1.B^{2m} + (W_1.X_0 + W_0.X_1).B^m + W_0.X_0
\]

The eq.(1) requires four multiplications to get the complete result, whereas, using Karatsuba method, by re-writing \( \beta \) as in eq.(2), it can be reduced to only three multipliers.

\[
\beta = (W_1 + W_0)(X_1 + X_0) - \alpha - \gamma \quad (2)
\]

or

\[
\alpha = \gamma = (W_1 - W_0)(X_1 - X_0) \quad (3)
\]

Similarly by extended this technique, i.e. by splitting the operands into three parts, we reduce number of multiplier from 9 to 6. The details are described as follows:

We can divide the operands \( W \& X \), and do the multiplication \( W.X \) as follows,

\[
W = W_2.B^{2m} + W_1.B^m + W_0, \quad X = X_2.B^{2m} + X_1.B^m + X_0
\]

\[
W.X = \alpha_2.B^{4m} + \alpha_1.B^{2m} + \alpha_0 + \beta_2.B^{3m} + \beta_1.B^{2m} + \beta_0.B^m
\]

where,

\[
\alpha_2 = W_2.X_2, \quad \beta_2 = W_2.X_1 + W_1.X_2, \quad \alpha_1 = W_1.X_1,
\]

\[
\beta_1 = W_2.X_0 + W_0.X_2, \quad \alpha_0 = W_0.X_0, \quad \beta_0 = W_1.X_0 + W_0.X_1
\]

Up to this level we need 9 multiplier to accomplish the task. The number of multiplication can be reduced to 6 by modifying the \( \beta_2, \beta_1, \) and \( \beta_0 \), as below.

\[
\beta_2 = (W_2 + W_1)(X_2 + X_1) - \alpha_2 - \alpha_1
\]

\[
\beta_1 = (W_2 + W_0)(X_2 + X_0) - \alpha_2 - \alpha_0
\]

\[
\beta_0 = (W_1 + W_0)(X_1 + X_0) - \alpha_1 - \alpha_0
\]

or

\[
\beta_2 = \alpha_2 + \alpha_1 - (W_2 - W_1)(X_2 - X_1)
\]

\[
\beta_1 = \alpha_2 + \alpha_0 - (W_2 - W_0)(X_2 - X_0)
\]

\[
\beta_0 = \alpha_1 + \alpha_0 - (W_1 - W_0)(X_1 - X_0)
\]

III. MULTIPLIER DESIGN METHODOLOGY

In this section, we discuss the details for the implementation of the multiplier design. The designing of 34-bit, 51-bit, 66-bit, 130-bit, then 113-bit multiplier for quadruple precision mantissa multiplication multiplier has been discussed, which finally leads to the implementation quadruple precision floating point multiplier. At present, the implementations of our designs focus on the Xilinx Virtex-4 FPGA platform.

A. 34-bit Multiplier Design

In general, the 34-bit multiplication on Xilinx FPGA platform requires 4 DSP48 block to accomplish the multiplication result. Whereas the incorporation of the Karatsuba method will reduce the this number from 4 to 3 DSP48. Because of this, we have used eq.(1) with eq.(3) to utilize the two partitioning approach of Karatsuba for 34-bit multiplication. This has been used to utilize the \( 18 \times 18 \) signed multiplication capability of DSP48 for \((W_1 - W_0)\times(X_1 - X_0)\), instead of \( 18 \times 18 \) unsigned multiplier for \((W_1 + W_0)\times(X_1 + X_0)\) in eq.(2). This approach will have a hardware benefit on the FPGA platform.

Fig. 1 has shown the implementation of the 34-bit multiplication. Both operands here have been partitioned in two parts of 17-bit each. We have tried to efficiently use available DSP48 IP core on FPGA. The DSP48 IP core on Xilinx Virtex-4 FPGA is powered with 17x17 unsigned multiplier along with a 48-bit adder/subtractor and some more additional operations. In Fig. 1, unsigned multiplier block \( m00 \) and \( m11 \) are used for \( \alpha \)'s generation and signed multiplier block \( m10 \) is used for \( \beta \) generation in eq.(1). In order to use the inherent 48-bit adder on the DSP48 block, the pair of \( m11 \) and \( m0 \) have been ported on a single DSP48 block, and \( m10 \) and \( s1 \) has been ported on another single DSP48 block. This method has helped us to reduce some extra adder/subtractor logic in the implementation.

B. 51-bit Multiplier Design

The implementation of 51-bit multiplier has used the three partitioning approach of the Karatsuba method. As com-
pared to the general approach, our approach requires only 6 multiplier blocks instead of 9 blocks to complete a 51-bit multiplication. Here eq.(4) has been used with eq.(6) for this implementation. Again as above (for 34-bit multiplier), the computation of $\beta's$ in eq.(4) have used the signed multiplication capability of DSP48, in eq.(6), instead of $18 \times 18$ unsigned multiplications in eq.(5), to save some area required for additional adders in it.

The implementation of 51-bit multiplier has been carried out as Fig. 2(a). Both 51-bit operands have been partitioned in to three parts, in which the width of each of them is 17-bit. The unsigned multiplier block $m00$, $m11$ and $m22$ have been used to produce $\alpha_0$, $\alpha_1$ and $\alpha_2$ in eq.(4), whereas signed multiplier blocks $m10$, $m20$ and $m21$ used for the generation of $\beta_0$, $\beta_1$ and $\beta_2$ of eq.(4). Here, the pair of $m10$ & $s10$, $m20$ & $s20$, and $m21$ & $s21$ has been ported individually on a single DSP48 unit, in order to use the available adder/subtractor on chip. The final adder unit has been further pipelined to achieve even better performance.

C. 66-bit Multiplier Design

In the design of a 66-bit multiplier, we use the implementation of a 34-bit multiplier as a primary building block. Using two partitioning approach of Karatsuba multiplier, we can build a maximum of a 66-bit unsigned multiplier using a 34-bit unsigned multiplier, because of the reason mentioned below. For the implementation of a 66-bit multiplier, eq.(1) with eq.(2) has been utilized. For this, we need two unsigned multipliers of width $m$-bit and one unsigned multiplier of width $(m + 1)$-bit.

The Implementation strategy of 66-bit operands multiplication has been shown in the Fig. 2(b). Each stage in the Fig. 2(b) has been further pipelined for better performance numbers. Here, the operands have been partitioned in to two parts with width 33-bit. So, here we need two unsigned multiplier of 33-bit and one unsigned multiplier of 34-bit. In Fig. 2(b), $m00$ and $m11$ are 33-bit unsigned multiplier blocks used for generation of $\alpha_0$ and $\alpha_1$, respectively, and $m10$ is a 34-bit unsigned multiplier block used for the generation of $\beta$ in eq.(1). Typically, all $m00$, $m11$ and $m01$ have used 34-bit multiplier design discussed above. In comparison to the general multiplication approach, this implementation requires only 9 DSP48 multiplier blocks instead of 16 DSP48 blocks.

D. 130-bit Multiplier Design

The design approach of a 130-bit multiplier is similar to that of the 66-bit multiplier discussed above. For the case of the 130-bit multiplier, the architecture is similar to that of the 66-bit multiplier, Fig. 2(b). Here, using two partitioning approach, the 130-bit operands is divided into two parts (65-bit each). The 66-bit multiplier is used here as building block. For 130-bit multiplier, we need two unsigned multipliers of 65-bit each, for $\alpha$'s generation (analogous $m00$ and $m11$ in Fig. 2(b)) and one 66-bit unsigned multiplier for $\beta$ generation (analogous $m01$ in Fig. 2(b)), in eq.(1). As, previous all the stages and bigger adders/subtractors are further pipelined to achieve a better performance. The total number of DSP48 used 130-bit multiplier is 27, as contrast to using 64 DSP48 for the general approach.

E. 113-bit Multiplier (Quadruple Precision)

For the quadrant precision multiplication, it requires a $113 \times 113$ multiplier. Here, operands has been divided into two unequal parts, 51-bit and 62-bit. Thus, it need one 51-bit multiplier ($W_0 \times X_0$), one 62-bit multiplier ($W_1 \times X_1$) and one 63-bit multiplier ($((W_1 + W_0) \times (X_1 + X_0))$). Implemented of 51-bit multiplier is done by three partitioning extension of Karatsuba method, needs 6 DSP48 blocks. The 62-bit & 63-bit multiplier have been implemented by two partitioning method, and basically done using 66-bit multiplier architecture, which requires three 34-bit multipliers, for each, a total of 9 DSP48 blocks. Thus, a total of 24, $17 \times 17$ multiplier blocks are required to do $113 \times 113$ multiplication, which has large saving in terms of area. Whereas, by general approach $113 \times 113$ multiplication do need 49, $17 \times 17$ multiplier blocks. The architecture of 113-bit multiplier is similar to that of Fig. 2(b), with $W_1=51$-bit, $X_1=51$-bit, $W_0=62$-bit, $X_0=62$-bit, $m00$ as a 62-bit multiplier, $m11$ as 51-bit multiplier, and $m10$ as 63-bit multiplier. Using this architecture of the complete 113-bit multiplication can be achieved.

F. Larger Multiplier Designs

Using a similar method as discussed above, we can generally design even larger integer multiplier with much less required area. We discuss some more cases below in brief:
TABLE I: Implementation details

<table>
<thead>
<tr>
<th>Design</th>
<th>Latency</th>
<th>Slice</th>
<th>LUT</th>
<th>FF</th>
<th>DSP48</th>
<th>Freq(M)</th>
</tr>
</thead>
<tbody>
<tr>
<td>34-bit Mult</td>
<td>4</td>
<td>120</td>
<td>118</td>
<td>172</td>
<td>5</td>
<td>324</td>
</tr>
<tr>
<td>51-bit Mult</td>
<td>6</td>
<td>333</td>
<td>420</td>
<td>362</td>
<td>6</td>
<td>324</td>
</tr>
<tr>
<td>66-bit Mult</td>
<td>8</td>
<td>627</td>
<td>796</td>
<td>1126</td>
<td>9</td>
<td>310</td>
</tr>
<tr>
<td>130-bit Mult</td>
<td>14</td>
<td>2685</td>
<td>3505</td>
<td>4955</td>
<td>27</td>
<td>295</td>
</tr>
<tr>
<td>113-bit Mult</td>
<td>14</td>
<td>2373</td>
<td>3050</td>
<td>5998</td>
<td>24</td>
<td>310</td>
</tr>
<tr>
<td>QP FP Mult</td>
<td>17</td>
<td>2464</td>
<td>3211</td>
<td>3961</td>
<td>24</td>
<td>310</td>
</tr>
</tbody>
</table>

1) 84-bit Multiplier: By using a combination of the 34-bit multiplier and the 51-bit multiplier, we can build a 84-bit multiplier, using two partitioning method of eq.(1) with eq.(2). The partitioning of operands needs to be of 34-bit and 50-bit each. Thus, it will require one 34-bit unsigned multiplier and one 50-bit unsigned multiplier for $\alpha$’s generation and one 51-bit unsigned multiplier for $\beta$ generation. In this way, it requires only 15 DSP48 blocks, instead of 25 DSP48 using general method.

2) 100-bit Multiplier: Similar to the design of the 66-bit multiplier using 34-bit multipliers, we can build a 100-bit multiplier using the 51-bit multiplier and two partitioning method. This will require only 18 DSP48 multiplier blocks instead of 36 DSP48 multiplier blocks as a whole.

3) 258-bit Multiplier: Furthermore, even for very large multipliers such as 258-bit multipliers, we can use above designed multipliers as basis. To design a 258-bit multiplier we need two 129-bit multipliers and one 130-bit multiplier, effectively all are 130-bit multipliers. So, we can accomplish the 258-bit multiplication result using only 81 DSP48.

Likewise we can build more larger integer multipliers, using the various combination of above discussed designs with much better use of hardware resources.

IV. RESULTS AND COMPARISONS

Hardware utilization and performance for proposed implementation is shown in Table-I on Virtex-4 FPGAs. All hardware reported results are post place-and-route data from the FPGA design tools. Effective use of DSP48 in the design has also helped in reducing some hardware resources.

Table-II has shown the comparison with the best reported implementation of quadruple precision floating point multiplication on FPGA platform. [5] (extension of [10]), is very recent result, has shown the implementation of quadruple precision multiplication on Virtex-4 FPGA platform. They require 49 DSP48 block for 113-bit multiplication. On a Virtex-5 device, they have reported an use of 34 DSP48 blocks, by using 24x17 unsigned multiplication capability of DSP48 on Virtex-5. Even on this, our design uses only 24 DSP48, which is a significant benefit in terms of hardware usage. [11] has reported the implementation large integer multiplier with bit width of 128, 256, and 512-bit. For 512-bit integer multiplier they have used 324 DSP48 blocks for computations, whereas, using our approach, we need only 243 DSP48 blocks, with fully pipelined computation. Thus, the presented design methodology is providing much area efficient integer multipliers.

V. CONCLUSIONS

We have presented a divide-and-conquer based hardware-efficient methodology for performing large integer multiplication on FPGA platforms, which has been extended to design an efficient architecture for quadruple precision multiplication on FPGAs. The proposed work has shown attractive area-efficient designs for several large integer multipliers, and it can be extended for even much larger integer multipliers. The proposed architectures achieve high performance with smaller area and shorter latency. The presented design of the QP multiplier is able to achieve approximately 50% area reduction in terms of the number of DSP48 usage compared with the best design reported in literature, along with less required latency and high performance.

REFERENCES