Turkish Journal of Electrical Engineering & Computer Sciences

http://journals.tubitak.gov.tr/elektrik/

Research Article

Turk J Elec Eng & Comp Sci
(2016) 24: 2961 – 2973

doi:10.3906/elk-1405-107

Single and multiple precision sequential large multipliers for field-programmable gate arrays

Ali ŞENTÜRK[1,*], **Mustafa GÖK**[2]
[1]Department of Computer Engineering, Çukurova University, Adana, Turkey
[2]Department of Electrical Engineering, Çukurova University, Adana, Turkey

Received: 16.05.2014 • **Accepted/Published Online:** 18.01.2015 • **Final Version:** 15.04.2016

Abstract: This paper presents single and multiple precision sequential large multiplier designs for field-programmable gate arrays. Both designs use the Karatsuba–Ofman method. They are pipelined and can generate a full size (double operand size) or a single size product. The syntheses results show that the sequential large Karatsuba–Ofman multiplier (SLKOM) implementations have up to 2.23 times less delay compared with the standard sequential large multipliers implementations presented in previous research. The 2048-bit multiple precision sequential Karatsuba–Ofman large multiplier (MPSLKOM) implementation can simultaneously execute eight 256-bit multiplications. The MPSLKOM implementations use roughly 1% more registers and up to 3% more LUTs than the SLKOM implementations.

Key words: Large multipliers, Karatsuba–Ofman, field-programmable gate array

1. Introduction

Large operands are widely used in scientific, cryptography, multimedia, and signal processing applications. Multiplication is one of the most used arithmetic operations in these applications [1,2]. General purpose processors do not contain large multipliers. To compensate for the lack of the hardware, special software routines or multiple-precision arithmetic libraries can be used to perform the multiplication of large operands (The GNU Multiple Precision Arithmetic Library). These routines decompose the large operands into standard size suboperands and perform multiple suboperand multiplications; the products of these multiplications are aligned and summed to generate the large product. There are algorithms faster than this simple method [3,4], however, they are constrained to use the standard size multipliers too. Thus, the software only approach becomes extremely time-consuming when a vast number of large multiplications are executed in applications. Therefore, there is a genuine need for large multipliers that work fast and use as little logic as possible.

The recent work on the design of large multipliers focuses on field-programmable gate array (FPGA) implementations due to their rapid design and flexibility advantages [5–12]. A brief discussion of the previous work is provided.

In [5], hybrid sequential large multipliers are designed using Broadcast (decomposition method's implementation) and Karatsuba–Ofman (KO) multiplier blocks. Various combinations of these multiplier blocks are tried out to implement 256-bit multipliers. Among these implementations, the one that uses four hierarchical stages of KO multipliers is the fastest, but uses the most logic resources; the implementation that uses two hierarchical stages of two broadcast multipliers is the slowest.

*Correspondence: asenturk@cu.edu.tr

In [6], a combinational large multiplier and squarer designs that use the decomposition method are presented. Both designs use fast adder trees to sum the partial products generated by suboperand multiplications; 20-bit to 85-bit multiplier implementations are mapped on Spartan-3 FPGAs.

In [7], another combinational large multiplier design that uses the decomposition method is presented. The design method exploits the structure of the arithmetic slices and the fast carry chains provided in Virtex 4 FPGAs; 16-bit to 221-bit implementations of the proposed design are mapped on the FPGAs.

In [8], a bit serial large multiplier design is presented. The design uses carry save adders to perform the addition of partial product bits. The product bits are converted on the fly from borrow save format to two's complement format; 128-bit to 1024-bit implementations are mapped on Virtex 2 FPGAs.

In [9], truncated large multiplier designs for high-precision floating-point multiplication are presented. The truncated multipliers can be used by applications that tolerate truncation error. The study modifies the KO method and applies it to both multiplication and squaring operations; 23-bit, 52-bit, and 112-bit pipelined implementations are synthesized and mapped on Virtex 4 FPGAs.

In [10], a large multiplier design that uses a modified KO method for high-precision floating-point multiplication is presented. A 128-bit quadruple-precision mantissa multiplier has been constructed using one 66-bit and two 65-bit multipliers instead of four 64-bit multipliers.

In [11], two combinational signed-large multipliers designs for FPGAs are presented. The first design uses symmetric multiplier blocks, while the second one uses asymmetric multiplier blocks; 51 by 68 to 51 by 190 multiplier implementations are mapped on Virtex 5 FPGAs.

In [12], three sequential large multiplier designs for FPGAs are presented. The paper uses the modified decomposition method and presents the speed-area tradeoff among those designs; 256-bit to 2048-bit implementations are synthesized and mapped on Virtex 5 FPGAs.

The main aspects of the previous work are summarized in Table 1. The columns of the table show the following: the reference of the work, the type of multiplication method, the target FPGA device, the size, delay, and the resource usage of the largest implementation mapped on the target platform. The resource usage is expressed in terms of the number of slices, LUTs, and utilized embedded multipliers.

Table 1. Previous work on large multipliers.

Publication	Mult. method	FPGA	Maximum implementation		
			Size (bits)	Delay (ns)	Hardware (Slices, LUTs, Muls.)
Quan et al., 2005, [5]	Hybrid	Virtex 2	256	380	17564 slices, 144 mul.
Gao et al., 2007, [6]	Decomposed	Spartan 3	85	22	400 LUT, 15 mul.
Athow and Al-Khalili, 2008, [8]	Decomposed	Virtex 4	221	16	60000 slices, 169 mul.
Bessalah et al., 2008, [8]	Serial	Virtex 2	1024	12091	2688 (CLBs)
Banescu et al., 2011,[9]	Truncated KO,	Virtex 5	112	3	2497 LUT, 19 mul.
Jaiswal and Cheung, 2012, [10]	KO, Div. Conq.	Virtex 4	130	3	2685 slices, 3505 LUTs, 27 mul.
Gao et al., 2012, [11]	Decomposed	Virtex 5	192	14	1150 LUT, 24 mul.
Senturk and Gok, 2012, [12]	Decomposed	Virtex 5	2048	570	8245 LUT, 121 DSP

The delays for the designs are rounded to the nearest integer and given in nanoseconds. Quan et al. [5] and Athow and Al-Khalili's [7] designs use excessive amounts of hardware resources. The multiplier proposed by Bessalah et al. [8] can support 1024-bit multiplication, but is extremely slow. The designs reported in Banescu et al. [9] and in Jaiswal and Cheung [10] are the fastest, but they are designed for floating point multiplication

and they cannot multiply operands larger than 130 bits. The design reported in Gao et al. [11] is approximately five times slower than the fastest implementations and suffers from the same limited operand size shortcoming.

The previous large multiplier designs are mostly combinational and achieve high execution speeds by liberally using FPGA resources. Currently, a 256-bit multiplier is the largest combinational implementation that can be mapped on a Virtex 5 FPGA by using all arithmetic slices. However, in practice all the arithmetic slices cannot be dedicated only to multiplier logic. Another issue is that the performances of the previous designs usually depend on some key attributes of the platforms such as the existence of the fast carry chains and the size of the built-in multipliers. Model-dependent optimization may not give the same results on all platforms, since even the members of the same FPGA family can have structural differences. Especially when the resources are very limited, the sequential designs are good alternatives to combinational designs. They require relatively small amount of resources, they can be mapped on any FPGA model, and they can multiply operands of any size as long as there exist enough resources for storage. Naturally, the sequential designs have higher latency compared with the combinational designs. On the other hand, pipelining and using fast methods such as the KO method can improve the performance of sequential designs. This paper presents single and multiple precision sequential large multiplier designs that explore this niche. The proposed designs decompose the large operands and use the KO algorithm to multiply the suboperands. The designs are pipelined to achieve maximum clock frequency. Both of them can generate full size products and this function is not even mentioned in most of the previous work; 256-bit, 512-bit, 1024-bit, and 2048-bit implementations of the proposed designs are mapped on FPGAs. The synthesis results are compared against the synthesis results given in previous large multiplier implementations. The rest of the paper is organized as follows: Section 2 presents the sequential large multiplication method and its implementation, Section 3 presents the multiple precision large multiplication method and its implementation, Section 4 gives delay and hardware usage results, and Section 5 presents the conclusion.

2. The sequential large KO multiplication (SLKOM)

The SLKOM algorithm first performs suboperand multiplications and then adds their products. A brief explanation for the decomposition method is given in the following: assume that w-bit large operands A and B are decomposed into n-bit suboperands. A and B can be expressed as the summation of the suboperands as:

$$A = \sum_{i=0}^{p-1} A_i \cdot 2^{ni}, B = \sum_{j=0}^{p-1} B_j \cdot 2^{nj} \tag{1}$$

where A_i and B_j represent i^{th} and j^{th} suboperands of A and B, respectively, and p represents the number of suboperands and is computed using $p = \lceil \frac{w}{n} \rceil$. The multiplication of A_i and B_j generates a $2w$-bit product, M, which can be also expressed as the sum of the suboperand multiplications as:

$$M = \sum_{i=0}^{p-1} \sum_{j=0}^{p-1} A_i \cdot B_j \cdot 2^{n(i+j)} \tag{2}$$

The computation of M using (2) requires $p^2 n$-bit multiplications and $(p^2-1)2n$-bit additions.

The decomposition method is modified for KO implementation as follows: let $A_{2i+1}A_{2i}$ and $B_{2j+1}B_{2j}$ be 2n-bit suboperands obtained by concatenating n-bit sub-operands A_{2i}, A_{2i+1} and B_{2j}, B_{2j+1}, respectively.

Eq. (1) is rewritten as:

$$A = \sum_{i=0}^{p/2-1} [A_{2i} + 2^n A_{2i+1}] \cdot 2^{2ni} B = \sum_{j=0}^{p/2-1} [B_{2j} + 2^n B_{2j+1}] \cdot 2^{2nj} \tag{3}$$

Three terms are defined using the suboperands as:

$$\begin{aligned} P\,(2i, 2j) &= A_{2i} \cdot B_{2j,} P\,(2i+1, 2j+1) = A_{2i+1} \cdot B_{2j+1} \\ P\,(2i+1, 2j) + P\,(2i, 2j+1) &= [A_{2i+1} + A_{2i}] \cdot [B\_(2j+1) + B\_2j] - P(2i+1, 2j+1) - P(2i, 2j) \end{aligned} \tag{4}$$

Eq. (2) is rewritten using these terms as:

$$M = \sum_{i=0}^{p/2-1} \sum_{j=0}^{p/2-1} [P(2i, 2j) + 2^{n+1}((P(2i+1, 2j) + P(2i, 2j+1)) + 2^{2n} P(2i+1, 2j+1)] \cdot 2^{2n(i+j)} \tag{5}$$

The computation of M using Eq. (5) requires $0.5p^2n$-bit multiplications, $0.25p^2n + 1$-bit multiplications, and $1.5p^2$ additions.

Algorithm 1 shows the steps and the data flow in time and space for the SLKOM. In this algorithm, '&' represents the concatenation operation, '$\{0\}_{n-3}$' represents a string of $n-3$ zeros, and the subscript notation '$x : p$' represents the string of bits from position x to p. For example, $M_{2n-1:n}$ represents the bits from positions $2n-1$ to n. The algorithm consists of two parts. The first part generates w less significant product bits. The second part generates w more significant product bits when needed. The inner loop in the first part is not iterated in time; the iterations in these loops show the inputs and outputs of the multipliers and adders. For example, the multiplication, $A_0 \cdot B_0$, is performed by Multiplier at iteration 0, and the multiplication, $A_0 \cdot B_{p-1}$, is performed by Multiplier $p-1$ at iteration 0. In the first part, each iteration of the outer loop generates $2n$ bits of the product. In the second part, the loop shows how the carry and sum values (C and S) are aligned and combined into two vectors, CN and SN, respectively. These vectors are added to generate w more significant product bits.

3. Implementation of a SLKOM

Figure 1 shows the block diagram for the SLKOM. The design has two main parts. The first part has five pipeline stages and when the pipeline is filled this part computes the less significant w-bits of the product in $p/2$ cycles. Moreover, an extra cycle is needed between large multiplications to reset the registers that hold values left by the previous multiplication. The second part is called "Align and add stage" and it computes the more significant w bits of the product. The units and their functions in all stages are explained as follows:

Stage 1: In this stage, two w-bit registers, $R1$ and R, are used to store operands A and B, respectively. $R1$ is a right-shift register, which shifts $2n$ bits in each cycle. Moreover, in the first stage $p/2 + 1n$-bit adders perform the additions $(A_{2j} + A_{2j+1}), (B_0 + B_1), (B_2 + B_3) \ldots (B_{p-4} + B_{p-3}), (B_{p-2} + B_{p-1})$.

Stage 2: In the second stage, the even numbered n-bit multipliers multiply the suboperand A_{2j} by the suboperands $B_0B_2, \ldots B_{p-4}B_{p-2}$. The product generated by an even numbered n-bit multiplier j is represented as PL_j. The odd numbered n-bit multipliers multiply the suboperand A_{2j+1} by suboperands $B_1B_3, \ldots, B_{p-3}B_{p-1}$. The product generated by an odd numbered n-bit multiplier j is represented as PH_j.

Algorithm 1. Sequential KO large multiplication.

Require: $S = 0, C = 0, CT = 0$

First Part

1. **for** i=0 **to** (p/2-1) **do**
2. **for** j=0 **to** (P/2-1) **do**
3. $PL = A_{2i} \cdot B_{2j}, PH = A_{2i+1} \cdot B_{2j+1},$
4. $PM = (A_{2i} + A_{2i+1}) \cdot (B_{2j} + B_{2j+1}) - PH - PL$
5. $PLL(i,j) = PL_{n-1:0}, PLH(i,j) = PL_{2n-1:n}$
6. $PML(i,j) = PM_{n-1:0}, PMH(i,j) = PM_{2n:n},$
7. $PHL(i,j) = PH_{n-1:0}, PHH(i,j) = PH_{2n-1:n}$
8. **if** j=0 **then**
9. $ST(i,0) = PLL(i,j) + S(i-1,2) + C(i-1,1) + CT$
10. $ST(i,1) = PLH(i,j) + PML(i,j) + S(i-1,3)_{n-1:0} + C(i-1,2)$
11. **else**
12. $ST(i,2j) = PLL(i,j) + PMH(i,j-1) + PHL(i,j-1) + S(i-1,2j+2) + C(i-1,2j+1)$
13. $ST(i,2j+1) = PLH(i,j) + PML(i,j) + PHH(i,j-1) + S(i-1,2j+3) + C(i-1,2j+2)$
14. **end if**
15. $S(i,2j) = ST(i,2j)_{n-1:0}, C(i,2j) = ST(i,2j)_{n+2:n}$
16. $S(i,2j+1) = ST(i,2j+1)_{n-1:0}, C(i,2j+1) = ST(i,2j+1)_{n+2:n}$
17. **end for**
18. $ST(i,2j+2) = PHL(i,j) + PMH(i,j)$
19. $S(i,2j+2) = ST(i,2j+2)_{n-1:0}, C(i,2j+2) = ST(i,2j)_{n+2:n}$
20. $S(i,2j+3) = PHH(i,j),$
21. $MT = (C(i,0) + S(i,1))\&S(i,0)$
22. $M_{[(i+1)\cdot 2n-1]:i\cdot 2n} = MT_{2n-1:0}$
23. $CT = MT_{2n}$
24. **end for**

Second Part

25. **for** k=0 **to** (p-1) **do**
26. $CN_{[(k+1)\cdot n-1]:k\cdot n} = \{0\}_{n-3}\&C(i,k+1)$
27. $SN_{[(k+1)\cdot n-1]:k\cdot n} = S(i,k+2)$
28. **end for**
29. $M_{2w-1:w} = CN + SN + CT$

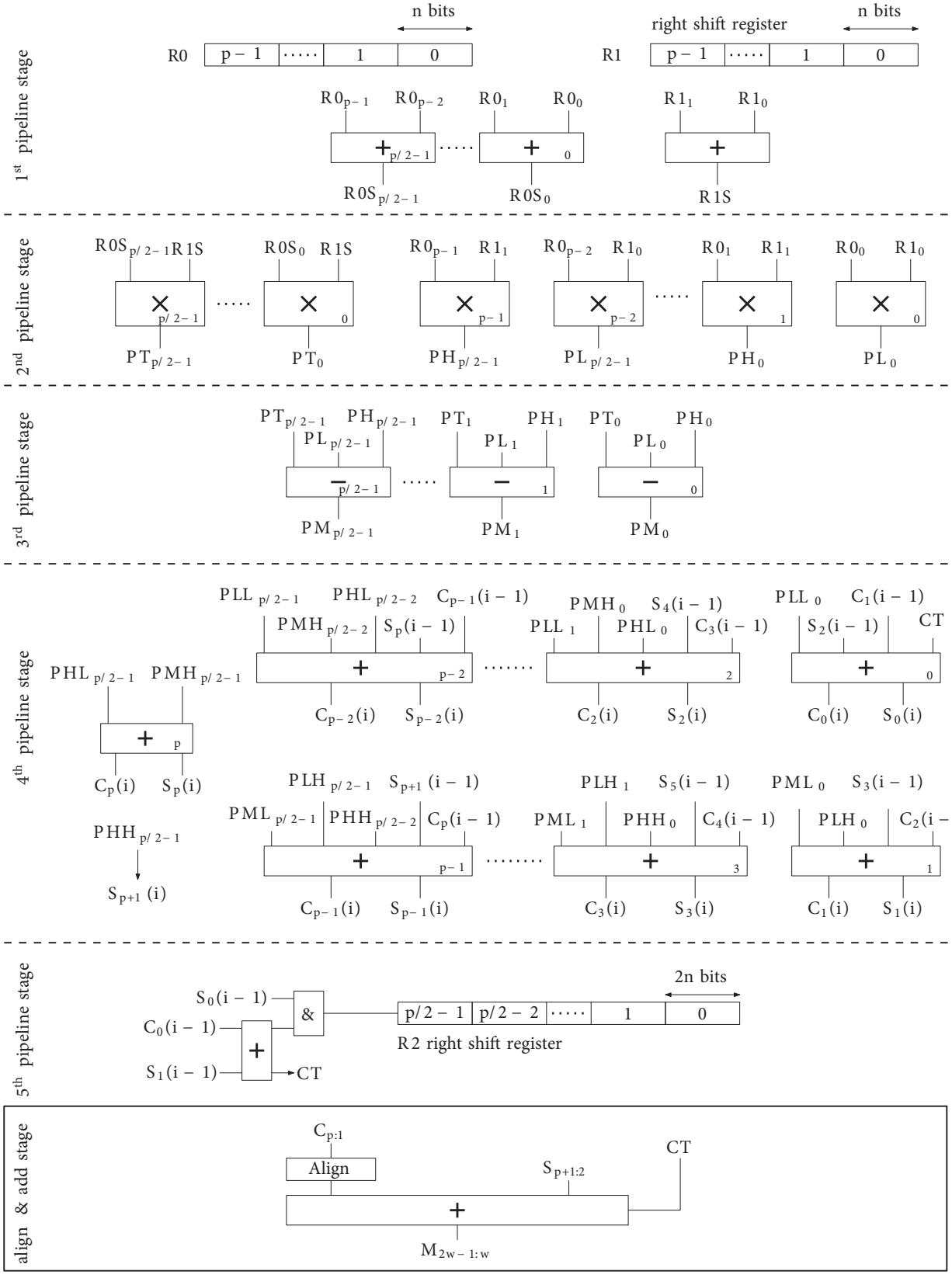

Figure 1. Block diagram for the sequential KaratsubaOfman large multiplier.

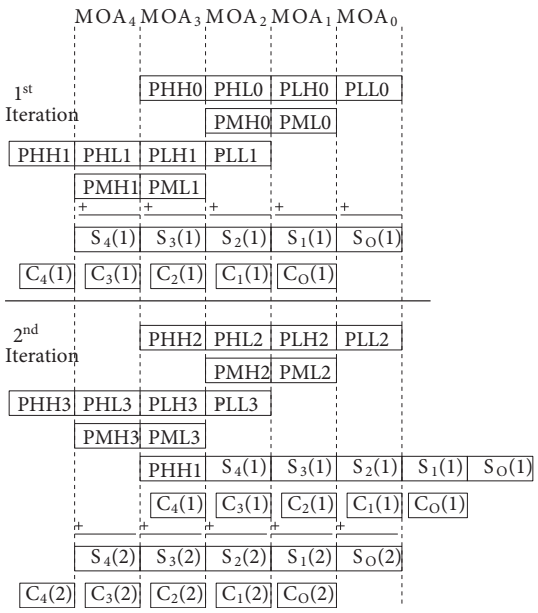

Figure 2. An example of addition of partial products.

Furthermore, $p/2(n+1)$-bit multipliers multiply the outputs of the adders generated in Stage 1. The output of the first adder, $R1S$, is multiplied by the outputs of the adders, RS_js. The products generated by these multiplications are represented as PT_js.

Stage 3: This stage consists of $p/2$ 3-operand subtractors. Each subtractor j computes $PM_j = PT_j - PH_j - PL_j$.

Stage 4: This stage consists of p multioperand adders ($MOAs$) that sum the products, PL, PM, and PH, generated in the the previous stages and the outputs of MOAs generated in the previous cycle. The sum and carry outputs of MOA_j at cycle i are represented as $S_j(i)$ and $C_j(i)$. To align the inputs of the MOAs, PL, PM, and PH values are further divided into low and high parts as PLL, PLH, PML, PMH, PHL, and PHH, respectively. MOA_0 adds PLL_0, $S_2(i-1)$, $C_1(i-1)$ and a carry bit, CT, which is generated by the adder located in Stage 5. MOA_1 adds PML_0, PLH_0, $S_3(i-1)$ and $C_2(i-1)$. The rest of the MOAs except MOA_p have five inputs. The even numbered MOAs add $PLL_{j/2}$, $PMH_{j/2-1}$, $PHL_{j/2-1}$, $S_{j+2}(i-1)$ and $C_{j+1}(i-1)$; the odd numbered MOAs add $PML_{(j-1)/2}$, $PLH_{(j-1)/2}$, $PHH_{(j-3)/2}$, $S_{j+2}(i-1)$, and $C_{j+1}(i-1)$, where $2 \leq j \leq p-1$. MOA_p adds $PHL_{p/2-1}$ and $PMH_{p/2-1}$. All MOAs generate 3-bit carries.

Example 1 *Figure 2 shows an example for the alignment and addition of MOA inputs. It is assumed that there are five MOAs and the partial products are generated by the multiplication of operands* $A = A(3)A(2)A(1)(A0)$ *and* $B = B(3)B(2)B(1)B(0)$. *The* $S_j(i)$ *and* $C_j(i)$ *values represent the sum and carry outputs of a MOA* j *at the iteration* i, *respectively.* $S_1(i), S_0(i)$, *and* $C_0(i)$ *are added in the fifth stage.* PHH_3 *and* $C_4(2)$ *are added in the next iteration.*

Stage 5: This stage consists of an n-bit adder and a w-bit right-shift register ($R2$). In every cycle, $2n$ bits of the product are generated by adding $S_1(i-1)$ and $C_0(i-1)$, and concatenating their sum with $S_0(i-1)$. The sum output of this adder is shifted into $R2$. The carry-out of the n-bit adder, CT, is added by the MOA_0 in Stage 4. After $p/2$ iterations, the w-bit right shift-register, $R2$ holds the less significant half

of the product. Then, S_is and C_is generated in this stage can be used to calculate the more significant half of the product in the "Align and add stage".

Align and add stage: This stage is independent from the pipelined structure. The align and add stage can compute the w more significant bits of the product while the first part is multiplying another large operand. The w-bit CPA located in this stage adds $S_{p+1:2}$s and $C_{p:1}$ vectors with the carry bit CT. This addition can also be carried sequentially as long as the delay for the computation is less than the delay for the first part. By this way a smaller adder than the current one can be used in the implementation.

4. Implementation of a multiple-precision SLKOM (MPSLKOM)

The SLKOM design can also be used to perform low precision multiplications. For example, a 2048-bit SLKOM can multiply operands smaller than 2048 bits by setting the unused inputs to zeroes and decreasing the number of iterations. However, this method is not very efficient, since the hardware that processes the zero inputs does not really contribute to the computation. This problem is solved by modifying the SLKOM design. The modified design is called MPSLKOM.

Figure 3 shows the block diagram for the MPSLKOM design. Similar to the SKOLM implementation, the design has five pipeline stages. Each stage consists of k blocks that can process (w/k)-bit operands. At the lowest precision, each column functions as an independent (w/k)-bit multiplier and executes k parallel multiplications. When the operand precision is doubled, columns are paired and each pair of columns functions as a $(2w/k)$-bit multiplier. At the highest precision, all the columns are combined and function as a single w-bit multiplier. In general, the MPSLKOM design can multiply (cw/k)-bit operands, where c is any power of 2 that is less than or equal to k. The precision of the multiplier is set by using the control signal sp.

In general, the logic design of the MPSLKOM is almost identical to the logic design of the SLKOM. Thus, only the details of the modified stages are shown in Figure 4. The logic designs of the blocks in Stages 2 and 3 are exactly the same as the logic design of the SKOLM's Stages 2 and 3. In Figure 4 $[t-1]$ and $[t+1]$ represent the previous and the next blocks, respectively. The details of the modifications are explained as follows:

Stage 4: As in the SLKOM implementation, an array of MOAs is used to perform the addition of partial products and the outputs of MOAs generated in the previous cycle. In the MPSLKOM implementation, the MOAs $0, 1, p-2$ and $p-1$ are modified. They have extra inputs designated by dotted and dashed boxes. The extra inputs are mutually exclusive. The dotted boxes show the extra inputs that exist only in block 0 or block -1 ; the dashed boxes show the extra inputs that exist in the rest of the blocks. The multiplexers in the boxes select one of the two input signals based on the precision of the operands. As explained above, the blocks are combined when the precision is increased. The details of the modifications made in the MOAs are given in the following:

- MOA_0 : The first input of MOA_0 is always CT in block 0. It can be either CT or 0 in the other blocks. When the blocks are combined, the first input of the MOA_0 is CT in the right most block of the group, and it is 0 in the other blocks of the group. The second input exists in blocks 1 to $k-1$. When the blocks are combined, the second input of the MOA_0 is 0 in the right-most block of the group, and it is $[t-1](C_p\&S_p)$ in the other blocks of the group.

- MOA_1 : The first input of this adder exits in blocks 1 to $k-1$. When the blocks are combined, the first input of the MOA_1 is $[t-1]S_{p+1}(i)$ in the right most block of the group, and it is 0 in the other blocks of the group.

- MOA_{p-2} : The second input of this adder in block $k-1$ is $S_p(i-1)$. In blocks 0 to $k-2$, when the blocks are combined, the second input of the MOA_{p-2} is $S_p(i-1)$ in the left most block of the group, and it is $[t+1]S_0(i-1)$ in the other blocks of the group.

- MOA_{p-1} : The first input of this adder in block $k-1$ is $C_p(i-1)$. It can be either $C_p(i-1)$ or $[t+1]C_0(i-1)$ in the other blocks. When the blocks are combined in a group, the first input of the (MOA_{p-1}) is $C_p(i-1)$ in the left most block of the group, and it is $[t+1]C_0(i-1)$ in the other blocks of the group. The second input of the (MOA_{p-1}) in block $k-1$ is $S_{p+1}(i-1)$, and it can be either $S_{p+1}(i-1)$ or $[t+1]S_1(i-1)$ in the other blocks. When the blocks are combined, the second input of the MOA_{p-1} is $S_{p+1}(i-1)$ in the left most block, and it is $[t+1]S_1(i-1)$ in the other blocks of the group.

Stage 5: The block of this stage consists of an n-bit adder and a (w/k)-bit right shift register $R2$. In each cycle, $2n$ bits of the product are generated by adding $S_1(i-1)$ and $C_0(i-1)$ and concatenating the sum with $S_0(i-1)$. This value is shifted into $R2$. The carry-out of the adder is added by the MOA_0. In blocks 0 to $k-2$, when the blocks are combined, the stored value is changed to $R2_0(i+1)$ in the left most block of the group, and it is kept the same in the other blocks of the group.

Align and add stage: Similar to the SLKOM design, the blocks in this stage are independent from the blocks in the pipelined part. The blocks contain (w/k)-bit adders that compute the more significant half of the products. The inputs of the n-bit adder are modified as follows: the first input is CT in block 0. It can be either CT or $[t-1]CO$ in the other blocks. When the blocks are combined, the first input is CT in the right most block of the group, and it is $[t-1]CO$ in the other blocks of the group. In block $k-1$, the second input is $S_{p+1:2}$. In blocks 0 to $k-2$, when the blocks are combined, the second input is $S_{p+1:2}$ in the left most block of the group, and it is $[t+1]S_{1:0}\&S2_{p-1}$ in the other blocks of the group. The third input is the same in all blocks. When the blocks are combined, the third input of the adder is aligned $C_{p:1}$ in the left most block of the group and it is $0\&C_{p-1:1}$.

5. Results

This section presents the syntheses results for the SLKOM and the MPSLKOM implementations and their comparisons with previous large multiplier designs. VHDL models for the implementations of the proposed designs are written. The functional verification of all models is tested by exhaustive simulation. The models are synthesized using Xilinx ISE tool set and mapped on Virtex FPGAs. For all syntheses the models are optimized for speed and the target FPGA speed grades are set to –2.

Table 2 presents the comparison between the standard sequential large multiplier (SSLM) implementations presented in [12] and the SLKOM and MPSLKOM implementations presented in the present study. VHDL models of these implementations are mapped on Virtex 5 xc5vfx100t FPGAs. The columns in Table 2 show the operand sizes, the multiplier types, the number of clock cycles, the delays in nanoseconds, and the number and utilization percentages of registers, LUTs, and DSPs. In [12], the clock periods for all SSLM implementations are given in the range of 4.143 to 4.157 ns. The clock periods for all SLKOM and MPSLKOM implementations are equal to 4.159 ns. The total delay for each implementation is equal to $(p/2+1)$ clock periods, where p is the number of the suboperands. The delay for the "Align and add stage" is not taken into account for the calculation of the total delay since this stage is independent from the other stages and it can run while the other stages perform the next large multiplication. The SLKOM and MPSLKOM implementations use more

hardware resources and require fewer cycles to generate the product than the SSLM implementations. The synthesis results show that the SLKOM implementations are 2.11 to 2.23 times faster and use 55% to 59% more DSP slices than the SSLM implementations. The MPSLKOM implementations have up to 3% more register and LUT utilization compared to the SLKOM implementations, while both designs' implementations use the same number of DSP slices.

Table 2. Comparison of the SSLM, SLKOM, and MPSLKOM implementations (Virtex 5).

Size	Type	Cycles	Delay (ns)	Register		LUT		DSP	
				Num.	Ut.%	Num.	Ut.%	Num.	Ut.%
512	SSLM	38	157.434	2732	4%	2125	3%	31	12%
	SLKOM	17	70.703	6248	10%	3913	6%	48	19%
	MPSLKOM	14	70.703	6445	10%	4217	7%	48	19%
1024	SSLM	71	294.508	5342	8%	4165	7%	61	24%
	SLKOM	33	137.247	12392	19%	7801	12%	96	38%
	MPSLKOM	33	137.247	12931	20%	8603	13%	96	38%
2048	SSLM	137	569.509	10562	16%	8245	12%	121	47%
	SLKOM	65	270.335	24680	39%	15609	24%	192	75%
	MPSLKOM	65	270.335	25903	40%	17180	27%	192	75%

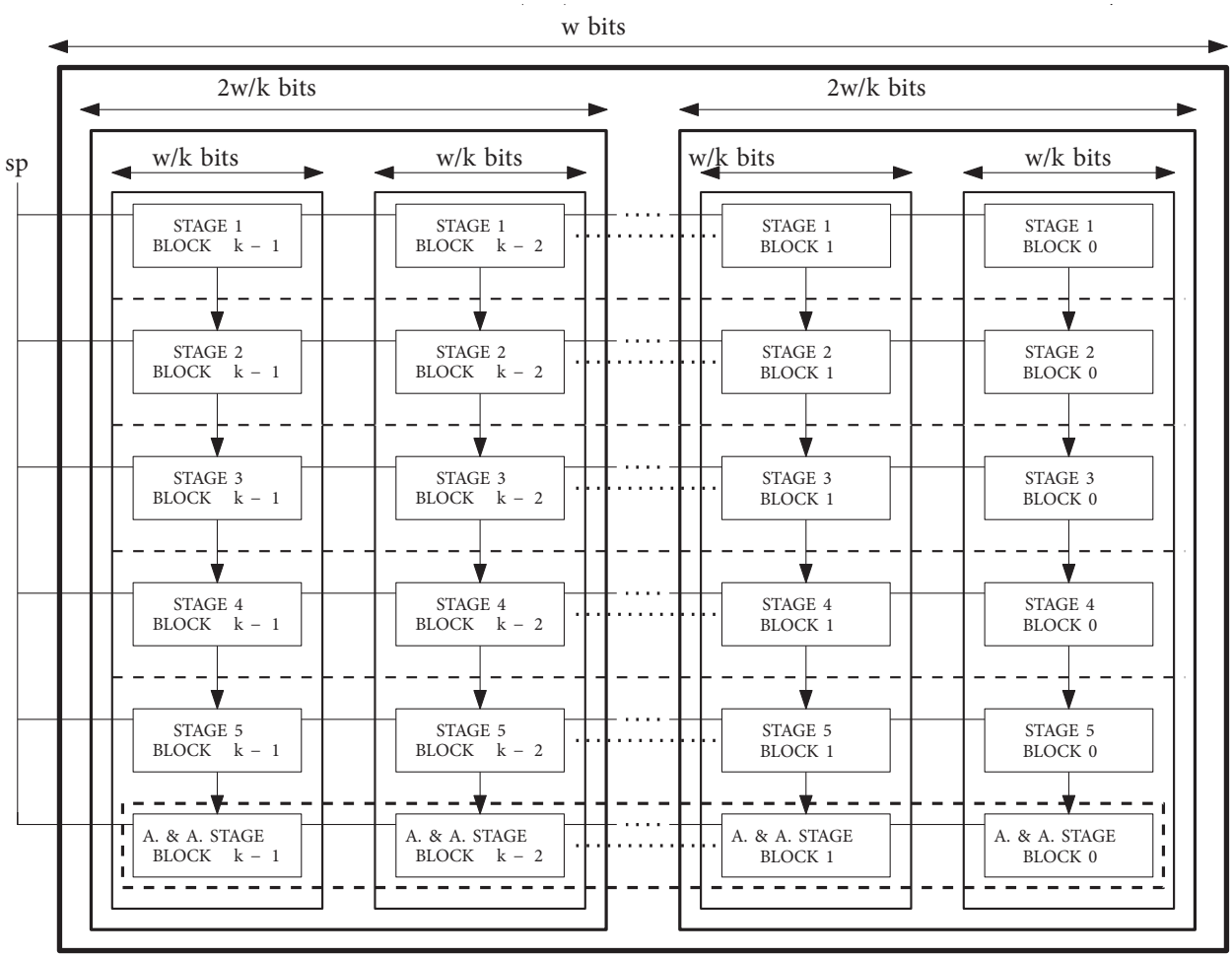

Figure 3. Block diagram for the multiple-precision sequential KaratsubaOfman large multiplier.

Table 3 presents a comparison of the SLKOM implementations with the previous implementations. Since the previous designs were mapped on different Virtex FPGAs, to make fair comparisons, 256-bit and 512-bit SLKOM implementations were mapped on the same models of Virtex 2, Virtex 4, and Virtex 5 families. The 256-bit SLKOM implementation had better delay than the referenced previous implementations, except the ones presented in [7] and [11]. Compared with the 256 by 256 SKOLM, the 221 by 221 design reported in Athow and Al-Khalili [7] was 2.75 times faster and used 7 times more DSPs; the 51 by 192 design reported in Gao et al. [11] was 2.64 times faster and used the same number of DSPs. On the other hand, at least six 51 by 192 multipliers are needed to multiply 256-bit operands. The register usage values for most of the previous implementations have not been reported, and thus this parameter is not shown in the resource usage column. However, the pipelined designs are expected to use much more registers than the combinational designs. The register utilization percentages for 256-bit SLKOM implementations are roughly 5% for all Virtex 5 platforms.

Table 3. Comparison of SLKOM with previous implementations.

Presented in	FPGA	Size	Hardware	Delay
Quan et al., [5]	Virtex2	256 × 256	17564 Slices	380
SLKOM	Virtex2	256 × 256	2539 Slices, 24 18 × 18 Mults	67
Jaiswal and Cheung, [10]	Virtex4	130 × 130	2685 Slices 24 DSPs	47
Athow and Al-Khalili, [7]	Virtex4	221 × 221	60000 Slices 169 DSPs	16
SLKOM	Virtex4	256 × 256	2541 Slices 24 DSPs	44
Gao et al., [11]	Virtex5	51 × 192	1150 LUTs 24 DSPs	14
Senturk and Gok, [12]	Virtex5	256 × 256	816 LUTs 16 DSPs	78
SLKOM	Virtex5	256 × 256	2018 LUTs 24 DSPs	37

Table 4 presents the syntheses results for MPSLKOM 512-bit, 1024-bit, and 2048-bit implementations on Virtex 5 xc5vfx100t FPGAs. The table presents the following values for each supported precision: the total number of cycles per multiplication, the number of parallel multiplications, the total delay for a single operation, and the delay per multiplication. For each implementation, the minimum operand precision is 256 bits, the delay/multiplication is calculated by dividing the delay for a single multiplication by the number of parallel multiplications. The results show that the 2048-bit MPSLKOM's delay/multiplication is less than the delay/multiplication of the fastest 256-bit combinational multiplier's delay/multiplication [10]. The 2048-bit implementation can also perform 512-bit, 1024-bit multiplications 4 and 2 times faster than an SLKOM implementation, respectively. In general, all the MPSLKOM implementations have higher throughput than the SLKOM implementations in low precision operation modes. Since a small amount of extra hardware is enough

Table 4. Synthesis results for MPSLKOM implementations.

Max Size	Size	Cycles	# Muls.	Delay (ns)	Delay/Mult
512	256	9	2	37.431	18.716
	512	17	1	70.703	70.703
1024	256	9	4	37.431	9.358
	512	17	2	70.703	35.352
	1024	33	1	137.247	137.247
2048	256	9	8	37.431	4.679
	512	17	4	70.703	17.676
	1024	33	2	137.247	68.624
	2048	65	1	270.335	270.335

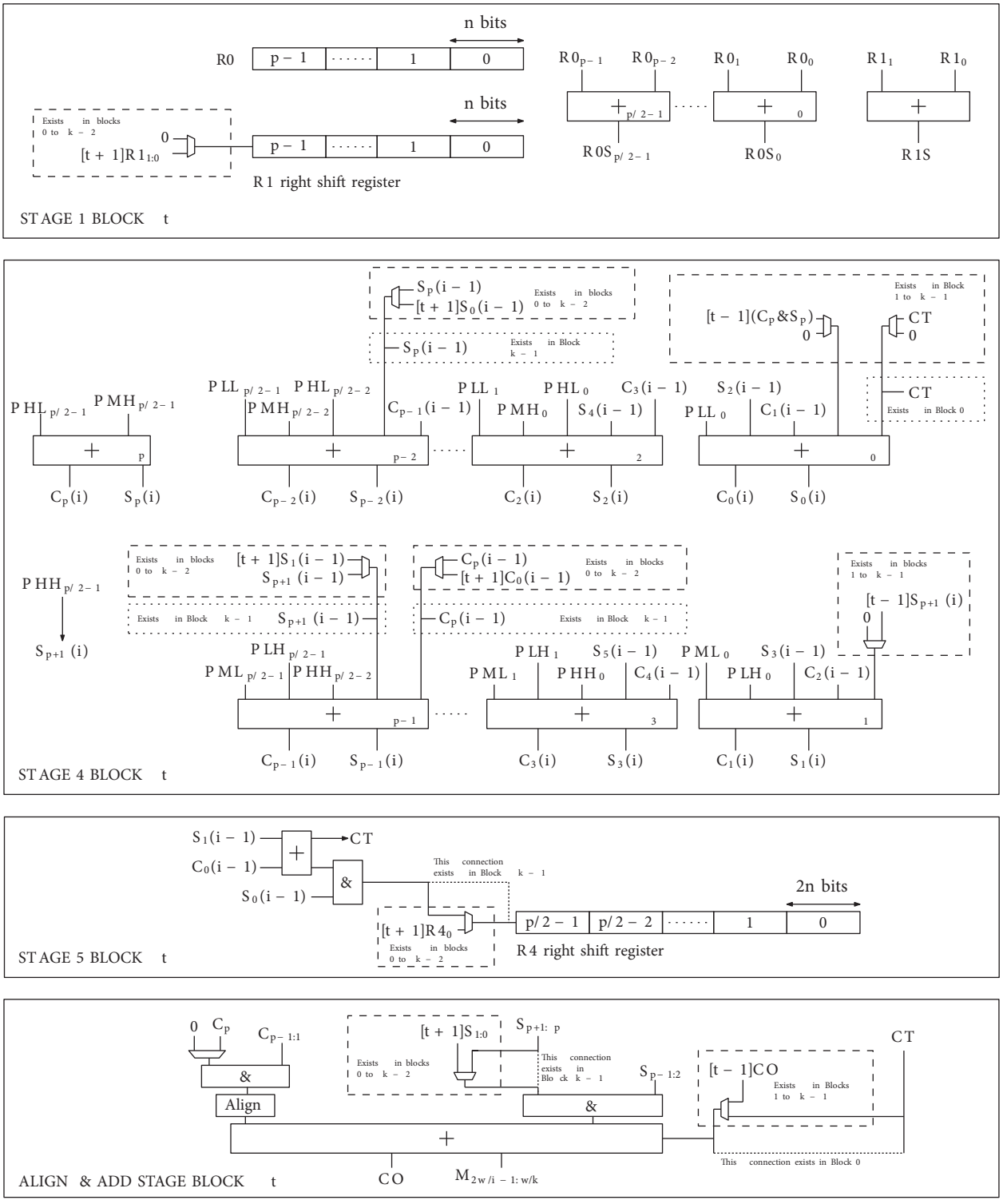

Figure 4. Details of the blocks in the MPSLKOM.

to convert an SLKOM to a MPSLKOM, they are expected to be preferred more than the SLKOMs. Note that instead of a MPLSKOM, multiple low precision SLKOMs can be mapped on an FPGA by using approximately

the same amount of hardware, but those low precision SLKOMs cannot be used to multiply higher precision operands.

6. Conclusion

This paper presented single and multiple precision sequential large multiplier designs for FPGAs (SLKOM and MPSLKOM). Both designs offer significant hardware savings compared with combinational designs, and thus, much larger sequential implementations than the combinational ones can be mapped on FPGAs. For example, 2048-bit SLKOM and MPSLKOM implementations use 75% DSP slices of a Virtex 5 FPGA. We modeled and synthesized 256-bit to 2048-bit implementations of SLKOM and MPSLKOM designs. The syntheses results show that the speed disadvantage of the sequential implementations can be solved by increasing the throughput. This can be observed from the results of MPSKOLM implementations. For example, the delay per multiplication for a 2048-bit MPSLKOM implementation was 4.679 ns at 256-bit multiplication mode, which was less than the delay for the fastest combinational implementation. The 2048-bit MPSLKOM implementation can also perform two 1024-bit multiplications and four 512-bit multiplications in parallel.

References

[1] FIPS P. 186-2. Digital Signature Standard (DSS). Gaithersburg, MD, USA: National Institute of Standards and Technology (NIST), 2000.

[2] Menezes AJ, Van Oorschot PC, Vanstone SA. Handbook of Applied Cryptography. Boca Raton, FL, USA: CRC press, 1996.

[3] Bodrato M. Towards optimal Toom-Cook multiplication for univariate and multivariate polynomials in characteristic 2 and 0. In: Arithmetic of Finite Fields, June 21–22 2007; Madrid, Spain: Springer. pp. 116-133.

[4] Karatsuba A, Ofman Y. Multiplication of multidigit numbers on automata. English translation in Soviet Physics-Doklady 1963; 7: 595-596.

[5] Quan G, Davis JP, Devarkal S, Buell DA. High-level synthesis for large bit-width multipliers on FPGAs: a case study. In: Proceedings of the 3rd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis; 19–21 September 2005; Jersey City, NJ, USA: ACM. pp. 213-218.

[6] Gao S, Chabini N, Al-Khalili D, Langlois P. Optimised realisations of large integer multipliers and squarers using embedded blocks. IET Comput Digit Tec 2007; 1: 9-16.

[7] Athow JL, Al-Khalili AJ. Implementation of large-integer hardware multiplier in Xilinx FPGA. In: 15th IEEE International Conference on Electronics, Circuits and Systems, ICECS; 31 August–3 September 2008; St. Julian's, Malta: IEEE. pp. 1300-1303.

[8] Bessalah H, Messaoudi K, Issad M, Anane N, Anane M. Left to right serial multiplier for large numbers on FPGA. In: The 3rd International Design and Test Workshop; 20–22 December 2008; Monastir, Tunisia: IEEE. pp. 288-293.

[9] Banescu S, De Dinechin F, Pasca B, Tudoran R. Multipliers for floating-point double precision and beyond on FPGAs. ACM Comp Ar 2011; 38: 73-79.

[10] Jaiswal M, Cheung RC. Area-efficient architectures for large integer and quadruple precision floating point multipliers. In: The 20th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM); 29 April–1 May 2012; Toronto, Canada: IEEE. pp. 25-28.

[11] Gao S, Al-Khalili D, Chabini N, Langlois P. Asymmetric large size multipliers with optimised FPGA resource utilisation. IET Comput Digit Tec 2012; 6: 372-383.

[12] Senturk A, Gok M. Pipelined large multiplier designs on FPGAs. In: The 15th Euromicro Conference on Digital System Design; 5–8 September 2012; Izmir, Turkey: IEEE. pp. 809-814.