



FPGA Modular Multipliers using Hybrid Reduction Techniques

Sergey Gribok 
Intel Corporation, United States
sergey.gribok@intel.com

Martin Langhammer 
Intel Corporation, United Kingdom
martin.langhammer@intel.com

Bogdan Pasca 
Intel Corporation, France
bogdan.pasca@intel.com

Abstract—Modular multiplication is a key kernel in many computing fields. What makes this function so challenging are the very large word sizes – sometimes in the thousands of bits – that are typically required for the target applications. In this paper we propose a modular multiplication implementation based on a multi-stage hybrid reduction technique. Our proposed approach uses a parameterized number of multiplier-based reduction stages followed by a memory-based reduction. This construction allows for the multiplier-based stages to take advantage of Karatsuba multiplication, resulting in a reduced number of DSP Blocks. Our method also allows specifying the number of multiplier-based stages which adjusts the ratio of multipliers to memory blocks. The resource utilization of the proposed architecture outperforms the existing state-of-the-art modular multiplication designs while offering a user-defined way of distributing resources between memory and DSP Blocks.

Index Terms—Modular Multiplier; Karatsuba; multiplier-based reduction; lookup-table-based reduction; fine-grain reduction; FPGA

I. INTRODUCTION

Modular multiplication is the core function in the implementations of many cryptographic systems. These include traditional cryptosystems such as RSA or ElGamal [1], as well as recently introduced cryptographic algorithms such as multi-scalar multiplication (MSM) in Zero Knowledge Proofs (ZKP) [2], Verifiable Delay Functions (VDFs) [3], [4], or Time-lock puzzles [5], [6]. The size of the modular multiplication varies widely between these fields. For RSA, the size currently considered safe until the year 2030 is 2048 bits [7]. For MSMs the bitwidths are lower, ranging from 256 to 512 bits whereas for time-lock puzzles the size of the modular multiplication can be as high as 3072 bits [5].

The capability of FPGAs has increased over time, with the number of logic, DSP, and memory resources growing rapidly, allowing large modular multipliers to be implemented in a single pipelined structure (*i.e.* not requiring an iterative or multi-cycle implementation). In this paper we introduce a modular multiplication implementation based on a multi-stage hybrid modular reduction technique. The reduction uses a set of multiplier-based steps where each step consists of a multiplication followed by an addition. The stages use rectangular multipliers, with the size of the multipliers diminishing with every new stage. The proposed implementation uses a combination of DSP and Memory Blocks to implement the modular reduction. The resource utilization of our proposed

architecture outperforms the existing Barrett's-based state-of-the-art design while offering a parameterized and balanced resource utilization between logic, memory and DSP Blocks.

The main contributions of this work are:

- the combined use of a multiplier-based reduction approach (allowing for the use of Karatsuba-based multipliers) with table-based reduction techniques,
- a new fine-grain reduction technique allowing for an efficient final reduction step,
- a detailed implementation of a 130-bit Karatsuba-based multiplier that is used at the core of all proposed multiplier compositions,
- new state-of-the-art modular multiplier implementation based on the newly proposed techniques.

This paper is organized as follows. After discussing the related works in Section II, we describe the arithmetic that underlies our proposed method in Section III. In Section IV we describe the algorithm that transforms the math into a realizable structure and present the resulting architecture for a 2048-bit running example in Section V. We next focus on the integer multiplier implementation in Section VI and present the wordgrowth analysis for a 4-stage recursive Karatsuba decomposition on 2048 bits. The leaf-node multiplier core that is used by the large Karatsuba-based multiplier is then described in Section VII. The results are presented in Section VIII, along with a discussion of the normalized cost of the different resource combinations in an FPGA context. We conclude with Section IX.

II. RELATED WORKS

The best-known modular multiplier approaches are due to Barrett [8] and Montgomery [9]. There are several FPGA implementations of Barrett's-based reduction, each exploring different approaches. In one case [10], the authors focus on optimizing the reduction for specific moduli. The implementation is based on multiplication by constants, and is implemented using shift-and-add algorithms. The FPGA results are only presented for very low bitwidths (12 and 23 bits). Another approach is to use truncated multipliers (where only a lower or upper portion of the result is used for the next step, which leads to a partial – and sometimes requiring considerably fewer arithmetic and logic components – multiplier implementation) [11], [12]. Karatsuba-based techniques (which we use in this work) have also been reported for optimizing multipliers

[11], [13]. Montgomery-based implementations targeting a throughput of 1 result/clock cycle have been reported in [14] in an FPGA context.

One of the techniques we will use in this work is *folding*, a method that allows performing coarse-grain modular reduction using multiplications. This has previously been reported in a software context [15], where it was applied as a pre-processing stage before a Barrett's-style reduction. We base one of the reduction stages presented in this work on this technique.

Lookup-table-based modular reduction [16] has been widely used in FPGA designs [17], [18], [19], largely because of the high number of embedded memory blocks available in modern FPGAs. An enhancement to the basic lookup-table-based method using runs of 1s and 0s is presented in [20], and is similar to a CSD (canonical-signed-digit) approach.

A much higher radix approach that uses a DSP-based reduction is presented in [18]. The product bits overhanging the modulo bit-width are split into DSP-wide chunks, which are the used to drive several rectangular multiplications in parallel. The higher radix leads to a significantly lower latency.

All the previously known works focus on using a single modular reduction technique within the architecture (whether it be Barrett's, Montgomery, lookup-table-based, or multiplier-based). As a result, with increasing operand widths, one type of resource (Logic, DSPs, Memory) typically becomes the bottleneck for these implementations. To the best of our knowledge, our work is the first to combine different techniques within a single implementation. This allows to build more efficient implementations since some techniques are better suited for course-grain reduction, while others are better suited for a fine-grained reduction. Additionally, it allows to use different FPGA resources in a balanced way (*i.e.* reflecting the ratio found in the device).

III. PROPOSED METHOD

In this work we propose a new modular multiplier architecture. Our goal is to compute:

$$Q \leftarrow XY \bmod M$$

or

$$Q \leftarrow P \bmod M$$

where X, Y and $P = XY$ are unsigned integers.

Let n denote the bit-width of both inputs X and Y , and of the modulus value M . Note that in this work we consider M to be a constant.

A. Multiplier-Based Coarse-Grain Reduction

The *folding*-based coarse reduction step introduced in [15] is based on the following congruence relations:

$$\begin{aligned} Q &= P \bmod M \\ &\equiv (P^H 2^\alpha + P^L) \bmod M \\ &\equiv ((P^H 2^\alpha \bmod M) + P^L) \bmod M \\ &\equiv \underbrace{(P^H \cdot (2^\alpha \bmod M) + P^L)}_{\text{new argument } P'} \bmod M \end{aligned}$$

By choosing an appropriate split between P^H and P^L in terms of number of bits, it can be ensured that the P' will be narrower than P . The cost of this reduction is the multiplication between the n -bit constant $(2^\alpha \bmod M)$ and P^H , followed by the addition of P^L . The *folding* reduction process may be repeated, with each iteration expecting to halve the size of the multiplier.

Once the size of the argument P' is within some threshold from the modulus bitwidth n , a subsequent lookup-table-based coarse-grain reduction may be performed.

B. Lookup-Table-Based Coarse-Grain Reduction

Let P' be the argument to be processed by the lookup-table-based modular reduction. The signal range that exceeds the n bits of the modulus is split into a number of limbs, such that the size of the limbs can be used to address either LUT-based or memory-based lookup tables. Let P' be written in terms of the n -bit base P'_{base} and a sum of β -bit-weighted chunks $2^{n+i\beta} P'_i$.

$$P' = \sum_i 2^{n+i\beta} P'_i + P'_{\text{base}}$$

The number of chunks is $c = \lceil (\text{length}(P') - n) / \beta \rceil$.

The following congruence relation can then be utilized:

$$\begin{aligned} P' \bmod M &\equiv \left(\sum_i 2^{n+i\beta} P'_i + P'_{\text{base}} \right) \bmod M \\ &\equiv \left(\sum_i \underbrace{(2^{n+i\beta} P'_i \bmod M)}_{\text{table lookup}} + P'_{\text{base}} \right) \bmod M \end{aligned}$$

Following the lookup-table-based reduction we are left with $c + 1$ n -bit terms that produce a result on $n + \lceil \log_2(c + 1) \rceil$ bits. A fine-reduction, described next, is used for reducing this to an n -bit value.

C. Fine-Grain Reduction

Let us denote by $\gamma = \lceil \log_2(c + 1) \rceil$, the number of bits that exceeds the n -bit word size at the end of the coarse-grain reductions. Our proposed fine-grain reduction uses the $\gamma + 1$ most significant bits of the coarse-grain result. A careful analysis of these $\gamma + 1$ bits allows for a good understanding of the magnitude of this value. This in turn allows for estimating the multiple of the modulus M that we need to subtract from the coarse-grain result to get the output Q in the $[0, M - 1]$ range.

We cannot exactly predict the exact multiple of M to perform the subtraction, since we are only looking at a small window of $\gamma + 1$ bits. Nonetheless, we can always determine two possible candidates of that multiple: K_1 and K_2 ($K_2 = K_1 + 1$). Based on these two multiples, two trial subtractions may be performed in parallel:

$$\begin{aligned} Q_1 &= P_{\text{coarse}} - MK_1, \\ Q_2 &= P_{\text{coarse}} - MK_2. \end{aligned}$$

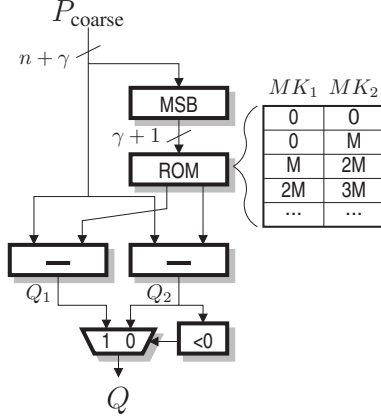


Fig. 1. Fine Reduction Architecture

Then, by checking the sign of Q_2 we can determine which of the two results to return: if Q_2 is negative then we return Q_1 , otherwise we return Q_2 . The two values MK_1 and MK_2 may be precomputed and stored in a ROM, as depicted in Figure 1. Finally, the $\gamma + 1$ MSB-bits of P_{coarse} may be used to access that ROM and fetch the two multiples in parallel. The method requires just a ROM access, two parallel subtractions, and a final multiplexer.

IV. ITERATIVE REDUCTION ALGORITHM

Algorithm 1 presents the high-level algorithm for implementing the n -bit modular multiplication using an s -step multiplier-based *folding* reduction followed by a table-based reduction. The algorithm inputs X and Y , which are n -bit unsigned operands to multiply, and M , the modulus, which is also an n -bit value that is constant. The algorithm performs s multiplier-based reduction steps. The size, in bits, of the data to be reduced at each step is denoted by w . Initially, this value is initialized to $w = 2n$ since at the first step of the reduction the input is the product XY . Variables P_i denote the current value to be reduced at step i , with $P_0 = XY$. At each step, the algorithm computes the variable k that will correspond to one of the dimensions of the multiplication being performed at the current step (the other dimension is n). The general rule of thumb, given the reduction step width w , is to compute k as half (floor) of the P_i bits that exceed n (half of $w - n$). Based on the value obtained for k , P_i is split into a high part (P_i^H) of k bits and a low part (P_i^L) of $w - k$ bits. The weight of the LSB of P_i^H is determined based on the value of k , and the constant $C_i = 2^{w-k} \bmod M$ is also calculated. The product, denoted by $T_i = P_i^H \cdot C_i$, holds on $n + k$ bits. This is then summed with P_i^L to produce the value to be reduced in the next iteration P_{i+1} , having the width $\max(n + k, w - k) + 1$.

At each step the algorithm requires implementing a rectangular multiplier, having one dimension set to n and the other to k . It is generally known that the discrete size of DSP Block multipliers yields certain multiplier sweet spots. Moreover, as it will be clarified in the upcoming section,

Algorithm 1: Modular multiplication algorithm for $Q \leftarrow AB \bmod M$ using s reduction steps

Input: X, Y - n -bit input values
Input: M - n -bit modulo value
Output: Q - the n -bit output value
 P, C, T are s -element arrays
 $R = \{\}$ // Reduction stack for coarse step
 $P_0 = X \cdot Y$
 $w = 2n$ // Step width
for i from 0 to $s - 1$ **do**
 $k = \lfloor (w - n)/2 \rfloor$ // Step mult width
 $g = \text{mult_sweet_spot}(k)$
if $k \leq g$ **then**
 $P_i^H = P_i(w - 1 : w - k)$
 $P_i^L = P_i(w - k - 1 : 0)$
 $C_i = 2^{w-k} \bmod M$
else
 $u_w = \text{find_trim_size}(w, n, g)$
 $U = P_i(w - 1 : w - u_w)$
 $R = R + \text{pair}(U, 2^{w-u_w})$ // Defer processing
 $w = w - u_w$
 $k = \lfloor (w - n)/2 \rfloor$ // Update mult width
 $P_i^H = P_i(w - 1 : w - k)$
 $P_i^L = P_i(w - k - 1 : 0)$
 $C_i = 2^{w-k} \bmod M$
end if
 $T_i = P_i^H \cdot C_i$ // Width $k+n$
 $P_{i+1} = T_i + P_i^L$ // $\max(k+n, w-k)+1$
 $w = \max(k + n, w - k) + 1$
end for
 $R = R + \text{pair}(P_s, 2^0)$ // Final item in the reduction stack
 $Q_{\text{coarse}} \leftarrow \text{table_based_reduction}(R)$ // Lookup-based reduction
 $Q \leftarrow \text{fine_grain_reduction}(Q_{\text{coarse}});$

when various Karatsuba techniques are used, additional multiplier sweet-spots are available. The general goal of the proposed algorithm is to always use the multiplier sweet-spots, and when a multiplier size (k) exceeds this value, to reduce the multiplication size to a sweet-spot. The function $\text{mult_sweet_spot}(k)$ returns a multiplier sweet-spot value close to k . For instance, if $k = 516$ the multiplier sweet-spot g returned by the function may equal 514. Since $k > g$, the function $\text{find_trim_size}(w, n, g)$, is called to find a new value w' for which $k' = (w' - n)/2 \leq g$. In this case the function may return $u_w = 4$, which yields $k' = 514$. The $u_w = 4$ bits from the top of P_i will be processed separately by the lookup-table-based reduction. The bits are first placed in the U variable, which is then added to the "to reduce" R list alongside their weight. The value of w is updated to reflect the new width to be processed by the next *folding* step, and the value k is recomputed based on this.

Once the s multiplier-based reduction steps are finalized, P_s is inserted to the reduction list R . This list of bits is then processed by a coarse-grain reduction, which is implemented based on table lookup. As previously explained, the result of the coarse-grain reduction will be somewhat wider than n . This result is then passed to a fine-grain reduction circuitry which reduces this back to n bits to produce Q , the output of this algorithm.

V. 2048-BIT MODULAR MULTIPLIER

In this section we use a 2048-bit modular multiplier to exemplify the execution of Algorithm 1 in the case of a multi-step *folding* reduction. We depict the resulting architecture in Figure 2 with a strong emphasis on the *folding* reduction part. The 2048 bitwidth results in a challenging implementation for a fully-pipelined implementation, as it will become clear in Section VIII where we list the performance of this implementation on a state-of-the-art contemporary device. The multiplication $P_0 \leftarrow XY$ is a full multiplication, with two 2048-bit operands producing a 4096-bit result. This product is split in two parts P_0^H and P_0^L , with P_0^H holding on 1024 bits and P_0^L holding on 3072 bits such that:

$$P_0 = 2^{2048+1024} P_0^H + P_0^L$$

The first step in reducing $P_0 \bmod M$ can be written as:

$$\begin{aligned} P_0 \bmod M &\equiv (2^{2048+1024} P_0^H + P_0^L) \bmod M \\ &\equiv \underbrace{((2^{3072} \bmod M) P_0^H + P_0^L)}_{P_1} \bmod M \\ &\equiv P_1 \bmod M \end{aligned}$$

For a known M , the value $C_0 = (2^{3072} \bmod M)$ is a constant that holds on 2048 bits. The product of this 2048-bit value with the 1024-bit P_0^H produces a 3072-bit value, which is then added to P_0^L (which also holds 3072 bits), to produce a 3073-bit stage result. This sum is denoted as P_1 in Figure 2. In order to add P_0^L to the multiplier output, it is delayed by a number of cycles. Quartus infers M20K-based shift registers for this logic.

Next, the 3073-bit P_1 is again decomposed in two parts: a high part P_1^H having a width of 513 bits and a low part P_1^L having a width of $512 + 2048 = 2560$ bits. The new constant $C_1 = (2^{2048+512} \bmod M)$ is multiplied by the 513-bit P_1^H to produce a 2561-bit T_1 product. This product is then added to P_1^L (a 2560-bit value), which results (after bit-growth) in a 2562-bit stage output value P_2 .

A third multiplier-based reduction step is performed starting with the 2562-bit value P_2 . This again is split in two parts, a high-part of 257-bits P_2^H and a low part P_2^L of $2048 + 257$ bits. The constant $C_3 = (2^{2048+257} \bmod M)$ is computed and is multiplied by P_2^H to produce a $2048+257=2305$ product T_2 . This is then added to P_2^L (which has the same width) to produce a 1-bit wider result P_3 , holding on 2306 bits.

In the next step, the 2306-bit P_3 value is passed through a lookup-table-based coarse-grain reduction. As previously explained, during the coarse-grain reduction the input is yet again split in two parts. The low part will consist of 2048-bits while the high part in this case will have a width $2306 - 2048 = 258$ bits. This 258-bit wide part extracted from the MSBs of P_3 is further decomposed in 29 chunks of $\beta = 9$ bits. Each chunk indexes a memory that stores 2048-bits. Using the M20K 9x40 configuration, this requires 52 M20Ks for tabulating each

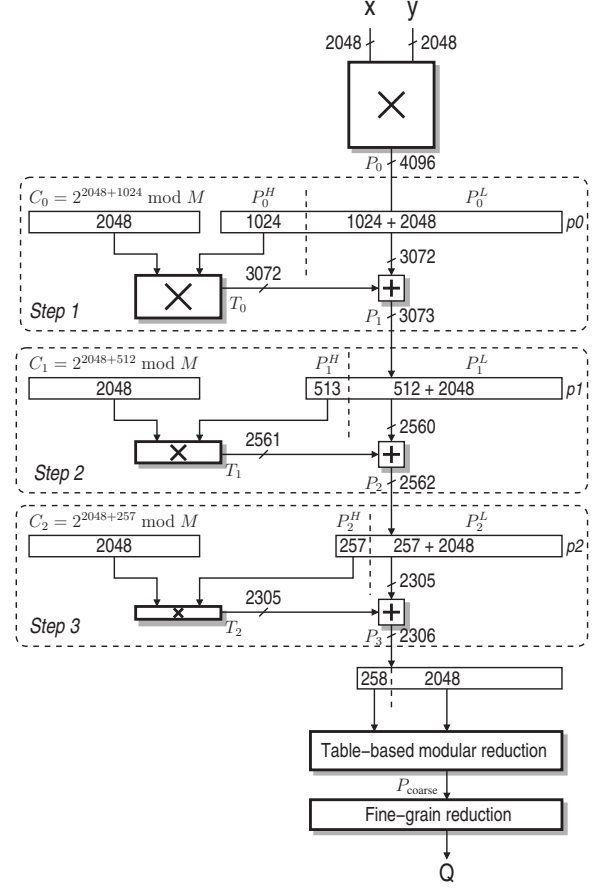


Fig. 2. 2048-bit modular multiplier using a 3-step multiplier-based modular reduction followed by a coarse and a fine-grain reduction

chunk. A total of $52 \times 29 = 1508$ M20Ks are used during the coarse-grain reduction. The 29 reduced terms are summed with the 2048-bit low part of P_3 . This sum of 30 terms produces P_{coarse} , that holds on $2048 + \lceil \log_2(30) \rceil = 2053$ bits, with $\gamma = \lceil \log_2(30) \rceil = 5$.

The fine-grain reduction utilizes $\gamma + 1 = 6$ bits from the top of P_{coarse} to fetch two close multiples of M : MK_1 and MK_2 . The required ROM bitwidth for the two multiples is $2(2048 + 6) = 4108$ bits. Note that since the ROM address size is only 6 bits, a LUT-based ROM implementation is more area efficient than a M20K-based one.

Efficiently constructing the multipliers for this implementation is critical to obtaining high-quality results. In the next section, we focus on the Karatsuba-based multipliers used throughout this implementation.

VI. 2048-BIT KARATSUBA GROWTH ANALYSIS

It is known that the Karatsuba decomposition allows trading-off multipliers for less costly operations such as additions or subtractions. Let A and B be the two operands of the multiplication of size $2k$. The additive Karatsuba technique expresses the product AB using two k -bit and one $k + 1$ -bit

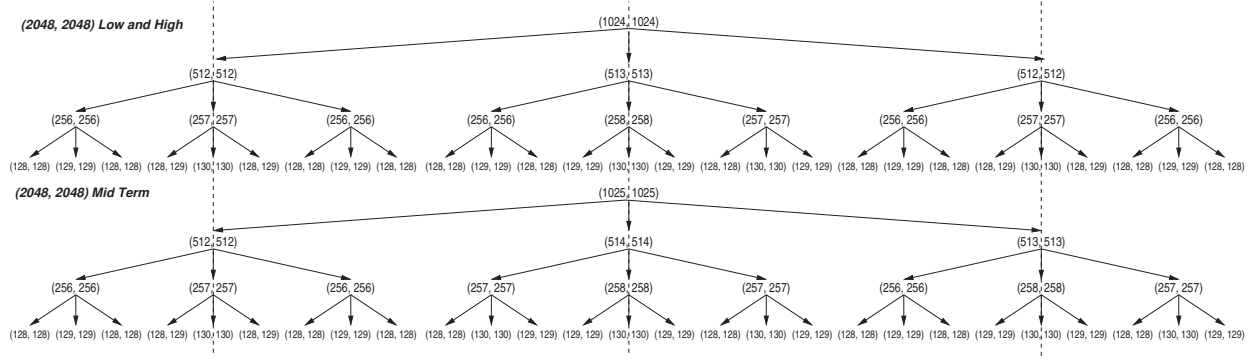


Fig. 3. 2048-bit multiplier, recursive Karatsuba multiplication sizes

unsigned multiplications, as opposed to 4 k -bit multiplications in the case of the schoolbook algorithm.

$$\begin{aligned}
 P &= A \cdot B \\
 &= (a_H 2^k + a_L) \cdot (b_H 2^k + b_L) \\
 &= 2^{2k} a_H b_H + a_L b_L \\
 &\quad + 2^k ((a_H + a_L)(b_H + b_L) - (a_H b_H + a_L b_L))
 \end{aligned}$$

The Karatsuba algorithm can then be applied recursively on the 3 multiplications. Figure 3 presents the multiplier sizes during an additive recursive 4-level Karatsuba decomposition of a 2048-bit multiplier. At the first level, the 2048-bit input limbs get split into two 1024-bit parts. The high and low products $a_H b_H$ and $a_L b_L$ are 1024×1024-bit multipliers. Their respective recursive decomposition is similar, and is presented on the top part of Figure 3 only once for both high and low products. The middle multiplier, computing the product $(a_H + a_L) \times (b_H + b_L)$ operates on 1025 bits. The recursive decomposition of this multiplier is presented on the bottom of Figure 3. The 1025-bit operands of this multiplier will again be decomposed in two parts: a high part comprising of 512-bits and a low part comprising of 513 bits. The corresponding $a_H b_H$ and $a_L b_L$ multipliers now operate on 512×512 and 513×513 bits respectively. The middle multiplier, computing the product $(a_H + a_L) \times (b_H + b_L)$ will now operate on 514 bits. Focusing again on this widest multiplier (514×514 bits), the decomposition results in two 257-bit chunks, resulting in two 257-bit multipliers (for the high and low multipliers) and one 258-bit multiplier (for $(a_H + a_L) \times (b_H + b_L)$). The 258-bit multiplier has a decomposition such that the high and low multipliers operate on 129 bits and the middle multiplier operates on 130 bits. Since this is the widest multiplier in the decomposition tree we can conclude that the required multiplier size such that this decomposition is efficient is 130 bits. In general, we can deduce that given an input multiplier size of 2^p , the leaf-node multiplier sizes resulting through the recursive Karatsuba decomposition are bounded by $2^{p-l} + 2$, where l is the number of recursive decomposition layers.

VII. 130-BIT MULTIPLIER TILE IMPLEMENTATION

In the previous section we saw that a 4-level recursive Karatsuba decomposition for 2048-bits requires $2^{11-4} + 2 = 130$ -bit multipliers on the leaf nodes. In this section we briefly present a 5-part Karatsuba-based approach used to efficiently implement this in devices having 27-bit multipliers - such as those available in all modern Intel/Altera FPGA devices. This implementation is based on the techniques introduced in [21]. We start by splitting the 130-bit words in 5 chunks of 26 bits.

$$\begin{aligned}
 X &= 2^{4k} X_4 + 2^{3k} X_3 + 2^{2k} X_2 + 2^k X_1 + X_0 \\
 Y &= 2^{4k} Y_4 + 2^{3k} Y_3 + 2^{2k} Y_2 + 2^k Y_1 + Y_0
 \end{aligned}$$

Next, we compute the following products using using DSP blocks (note that the number of products equals 15).

$$\begin{aligned}
 P_{ii} &= X_i \cdot Y_i, i \in [0, 4] \\
 D_{ij} &= (X_i - X_j) \times (Y_i - Y_j), i \in [1, 4], j \in [0, i - 1]
 \end{aligned}$$

Based on the previously expressed terms we can express the product as seen below.

$$\begin{aligned}
 XY &= 2^{8k} P_{44} \\
 &+ 2^{7k} (P_{33} + P_{44} - D_{43}) \\
 &+ 2^{6k} (P_{22} + P_{33} + P_{44} - D_{42}) \\
 &+ 2^{5k} (P_{11} + P_{22} + P_{33} + P_{44} - D_{41} - D_{32}) \\
 &+ 2^{4k} (P_{00} + P_{11} + P_{22} + P_{33} + P_{44} - D_{40} - D_{31}) \\
 &+ 2^{3k} (P_{00} + P_{11} + P_{22} + P_{33} - D_{30} - D_{21}) \\
 &+ 2^{2k} \underbrace{(P_{00} + P_{11} + P_{22} - D_{20})}_{T^{2k}} \\
 &+ 2^k (P_{00} + P_{11} - D_{10}) \\
 &+ P_{00}
 \end{aligned}$$

Figure 4 presents one possible architecture for implementing the expression in the equation above. On Stratix10 (fastest speedgrade), the 130-bit multiplier has a latency of 12 cycles, requires 1,584 ALMs and 15 DSP Blocks, and runs at 595 MHz. The Agilex7 DSP Block 27-bit mode has a 1-cycle longer latency, increasing the total latency to 13 cycles. In terms of performance, the Agilex7 implementation as presented in Figure 4 (using the same datapath, only the multiplier instantiation updated) requires 1189 ALMs, 15 DSP Blocks and runs at 882 MHz.

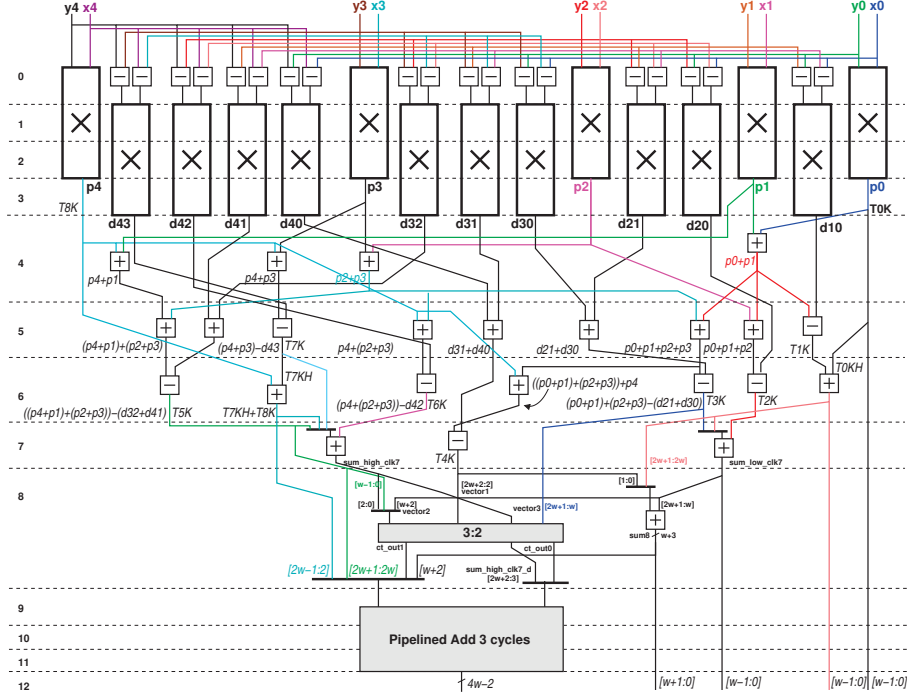


Fig. 4. Karatsuba-by-5 implementation of a 130-bit multiplier using 26-bit chunks on Stratix10

VIII. RESULTS AND DISCUSSION

A. Setup and Methodology

We report performance numbers for modular multipliers ranging from 256 to 2048 bits on Agilex7 and for 1024 bits on Stratix10 in Table I. For Agilex7 we target the AGIB027R31A1E1VB part, that is the fastest speedgrade device having 912,800 ALMs, 8,528 DSPs, and 13,272 M20Ks available. For Stratix10 we target the 1SM21CHU1F53E1VG part, fastest speedgrade device having a total of 702,720 ALMs, 3,960 DSPs and 6,847 M20Ks available. The reported results are obtained from single-seed runs using Intel Quartus Prime 23.4, without any placement or timing constraints.

For each bitwidth in the table, we report (a) the resource utilization of the main XY multiplier of the design, alongside (b) the modular reduction part (including the *folding*-based, lookup-table-based coarse-grain and fine-grain reduction), and (c) the complete modular multiplier design containing the two components and synthesized together. For the main-multiplier results (a) all 4 multipliers (256, 512, 1024 and 2048-bit) are built using the recursive decomposition approach shown in Figure 3. The 256-bit multiplier uses a single decomposition level, the 512-bit multiplier uses two levels, and three levels are used for the 1024-bit multiplier. In the case of the modular reduction part (b) we report the number of folding steps performed for every modular multiplier size. The adder trees used to sum-up the coarse-grain lookup-table outputs are sliced to 20 bits to match the LAB adder size.

B. Prior Work Comparison

Table I contains entries from prior works that we compare and contrast our results against.

1) *Stratix10*: In the case of the Stratix10 results, we compare our work against another Stratix10 implementation where 3 multipliers are used with a fine-grain reduction stage to implement a Barrett's-style modular multiplier [11]. The paper only reports the resource-utilization breakdown of the multipliers and fine-grain reduction stages, and does not report performance results for the full modular multiplier. We have tabulated the 3 major components alongside an estimated total resource utilization (in *italics*) in Table I for comparison. We selected the lowest reported frequency of these components as the *Total* frequency (although in our experiments this is unlikely to be achievable due to the size of the design). For the same 1024-bit configuration, our proposed modular multiplier utilizes fewer ALMs (219K vs 255K), fewer DSPs (975 vs 1183) but more M20Ks (392 vs 0). Our reported frequency is 442 MHz, which is 42 MHz lower than the frequency of the reduction-only component. Our design provides a more balanced resource usage of the FPGA; we provide a 15% reduction in logic and a 18% reduction in DSP which will have a proportional reduction in the footprint of these rather large designs. The 392 M20K memory blocks will have no impact on the footprint of this design, as they will be enveloped by the other resources (the 219K ALMs will span an area containing approximately 3200 M20Ks in the target device).

2) *Agilex7*: In the case of the Agilex7 results, the table contains two entries from a recent paper [18], Config #1

TABLE I
AREA UTILIZATION AND PERFORMANCE FOR THE PROPOSED IMPLEMENTATIONS FOR VARYING BIT-WIDTHS

Device	Width	Method	Breakdown	Latency	Resource Utilization			FMax (MHz)	Normalized Area
					ALMs	DSPs	M20Ks		
Virtex 6	256	[14]	Total	-	187K LUTs	0	0	68	18,700
Virtex Ultrascale+	256	[17]	Total	-	87K LUTs	289	0	17.55ns	12,139
	512	[17]	Total	-	329K LUTs	1089	0	25.68ns	45,927
Agilex7	256	ours	Multiplier	23	5,147	45	0	865	1,049
			Reduction, 1 step	34	6,681	30	105	796	1,693
			Total, 1 step	57	11,505	75	105	804	2,647
	512	ours	Multiplier	35	18,200	135	0	808	3,426
			Reduction, 1 step	47	22,580	90	377	791	5,730
			Total, 1 step	82	40,127	225	377	758	9,090
			Reduction, 2 steps	63	23,496	150	195	780	5,367
			Total, 2 steps	98	41,261	285	195	771	8,759
	1024	ours	Multiplier	47	60,236	405	0	747	10,842
			Reduction, 2 steps	88	81,475	450	754	608	18,302
			Total, 2 steps	135	140,831	855	756	635	29,072
			Reduction, 3 steps	104	83,721	570	390	610	17,638
			Total, 3 steps	151	142,972	975	392	623	28,396
		[18]	Total Config #1	1	375K	820	0	33.53	47,258
			Total Config #2	2	160K	820	3,014	73.35	44,957
	2048	ours	Multiplier	60	192,866	1,215	0	667	33,744
			Reduction, 3 steps	141	277,344	1,710	1586	557	58,185
			Total, 3 steps	201	470,627	2,925	1587	569	91,978
Stratix10	1024	ours	Multiplier	47	93,495	405	0	507	14,164
			Reduction, 2 steps	88	122,202	450	754	460	22,377
			Total, 2 steps	135	217,106	855	756	447	36,699
			Reduction, 3 steps	104	127,347	570	390	484	22,148
			Total, 3 steps	151	219,871	975	392	442	36,086
		[11]	Multiplier	46	93K	405	0	504	14,119
			Multiplier LSB	43	82K	405	0	512	13,019
			Multiplier MSB	46	73K	373	0	500	11,738
			Total	143	255K	1,183	0	500	39,577

and Config #2 (we note that another 7 configurations are reported in [18], however these configurations are for squarers – and hence not comparable with the results reported for the multiplier). Although both our work and the compared work implement modular multiplication, the application goals are different, which we will now discuss. Our work targets throughput – the maximum number of operations per second – and can therefore be deeply pipelined. Pipelining on FPGAs is generally very inexpensive, as every logic level has an associated pipeline register. The other work is architected for latency – the minimum time until the result is available. This is because the targeted application are VDFs (Variable Delay Functions), which require many iterative modular multiplications. We will assume that these can easily be pipelined for higher throughput if required, and achieve the same clock frequencies as our work. The cost of the pipelining will be negligible, as the registers are already available in the FPGA fabric. There will be a small amount of logic needed for inserting delays in unbalanced portions of the datapaths, but we will not consider these here as the additional cost is likely to be minimal.

Compared to Config #1, our proposed multiplier uses fewer ALMs (142K vs 375K), but more DSPs (975 vs 820) and more M20Ks (392 vs none). Despite utilizing 155 more DSP blocks

and 392 M20Ks, the 233K ALMs savings make the proposed multiplier a better choice in most contexts. Compared to Config #2, we still utilize fewer ALMs (142K vs 160K), 155 more DSP Blocks, but consume significantly fewer M20Ks (392 vs 3014).

3) *Other devices*: A Montgomery-style multiplier based on a 3-part Karatsuba implementation is reported for a Virtex-6 device [14]. Their results show that a 256-bit modular reduction requires 187K LUT6 and runs at a frequency of 68 MHz. Compared to this implementation, our Agilex7 256-bit design uses 12K ALMs, 75 DSPs and 105 M20Ks, again yielding a better resource utilization balance. Although the older Virtex-6 device (40nm process node) will be less performant, we might expect a 2:1 frequency ratio, rather than the 12:1 ratio shown here.

Another low-latency modular multiplier implementation, targeting similar applications as [18], is presented in [17] targeting an Ultrascale+ device. Results are reported for 128, 256 and 512-bit modular multipliers. For both 256 and 512 bits (on which we report results), our proposed implementations outperform the multipliers from [17] in terms of resource utilization (both LUTs and DSP Blocks). This is somewhat expected as the goal of [17] is to obtain as low of a latency as possible – at the expense of a higher resource utilization.

C. Normalized Area Comparison

In this section we compare the proposed modular multiplier implementation to related works by using a *normalized area* metric. This value is extrapolated based on the physical die area values reported in [22] for a Stratix-III device which states that the cost of 1 DSP Block is 11.9x the cost of 1 LAB, and the cost of 1 M9K block is 2.87x the cost of 1 LAB. Since Stratix10 and Agilex7 devices contain M20Ks (as opposed to M9Ks in Stratix-III), we scale up the memory block cost by 2.22x to compensate for this difference. Therefore, we estimate the cost of 1 M20K to be 6.37x the cost of 1 LAB. The last column of Table I tabulates this normalized area for all proposed implementations.

Using this metric, we can compare the earlier Stratix10 modular multiplier [11] against the two variations proposed in this work (2 folding step and a 3 folding step multiplier). First, among the two proposed variations, the 3-folding step variant shows a lower normalized area of 36,086 compared to 36,699 for the 2-folding step variant. The 36,086 area is roughly 10% lower than the 39,577 area calculated for the earlier work [11]. Our proposed implementation provides both a better balance of resources and normalized silicon area.

When applying the same comparison on the Agilex7 1024-bit modular multiplier, our 3-step implementation outperforms the earlier latency-optimized cores [18] by a significant margin (Config #2 [18] is 58% larger than our 3-folding step config.).

Finally, comparing against [14] and [17] is difficult, since they report results on Xilinx devices (Virtex-6 and Virtex Ultrascale+ respectively). We used the *1 LUT is equivalent to 1 ALM* metric to compute the normalized area, but arguably this may not be very precise. Given this, the area for the 256-bit Montgomery multiplier proposed in [14] is 18,700, that of the 256-bit low-latency modular multiplier proposed in [17] is 12,139 whereas the Agilex7 256-bit multiplier area is only 2,647. A similar advantage can be observed when analyzing the 512-bit modular multiplier, where [17] is converted to 45,927 area units, whereas the smallest Agilex-7 design has an area of 8,759.

D. Floorplan analysis

In Figure 5 we present the floorplan of the proposed 2048-bit modular multiplier on an Agilex7 device. Several components are highlighted including the integer multiplier in pink, the 3 *folding*-based reduction stages in red, purple and light blue as well as the lookup-table-based coarse-grain reduction in light green and the final fine-grain reduction in dark blue. The area of the highlighted regions corresponds to our expectation: the integer multiplier is 2048-bit and takes the largest area of the device. Next, the first stage of the multiplier-based reduction is 1024x2048-bits which is roughly equivalent to 2/3 of the 2048-bit multiplier size. Subsequent stages will consume 2/3 of the DSP count of the previous stage, but the assembling of the rectangular multipliers will have a cost. Nonetheless, we can observe that subsequent multiplier-based stages have an increasing lower area in the design. Both the



Fig. 5. Floorplan of the 2048-bit modular multiplier. In pink (on the top) we highlight the 2048-bit integer multiplier; in red (bottom left) reduction, stage 1; purple (bottom center) reduction, stage 2; light-blue (bottom right) reduction, stage 3; green (right-center) table-based reduction; dark-blue (right-center) fine-grain reduction.

table-based coarse-grain reduction as well as the fine-grain reduction occupy relatively small regions.

Finally, we need to highlight that the push-button floorplan returned by Quartus is interesting. Since the size of the components is decreasing, and the components are all organized in a chain, Quartus arranges them in a spiral allowing easy communication between subsequent components and avoiding the need to route wires across components.

IX. CONCLUSIONS

In this work we have proposed a new class of modular multipliers allowing to better balance the resource utilization trade-off between logic, DSP Blocks and Memory Blocks. The proposed approach uses a 2-stage coarse-grain modular reduction followed by a fine-grain reduction. Modular *folding*, used by the first coarse-grain reduction stage, is multiplier-based, and allows for the use of Karatsuba-based multipliers that primarily use DSP Blocks. The lookup-table-based reduction used in the second coarse-grain stage, along with the proposed fine-grain reduction, utilize primarily Memory Blocks, available in abundance on recent devices. The combination of reduction techniques balances the resource utilization between DSP and Memory Blocks. This balance can be further adjusted by varying the number of modular *folding* steps employed. We have demonstrated that the proposed techniques yield multipliers that scale to 2048-bit (and beyond) in fully parallel implementation style, while operating at over 550 MHz in an Agilex7 device (in the case of the 2048-bit multiplier) in a single-seed compile. Finally, we apply a normalized cost metric to compare our work and previously published works, and find that our architectures are considerably more efficient.

REFERENCES

- [1] W. N. A. Ruzai, M. R. K. Ariffin, and M. A. Asbullah, "On the variants of RSA cryptosystem and its related algebraic cryptanalysis," *Embracing Mathematical Diversity*, p. 67, 2019.
- [2] D. Čapko, S. Vukmirović, and N. Nedić, "State of the art of zero-knowledge proofs in Blockchain," in *2022 30th Telecommunications Forum (TELFOR)*. IEEE, 2022, pp. 1–4.
- [3] D. Boneh, J. Bonneau, B. Bünz, and B. Fisch, "Verifiable delay functions," in *Annual international cryptology conference*. Springer, 2018, pp. 757–788.
- [4] J. Xu, C. Wang, and X. Jia, "A survey of Blockchain consensus protocols," *ACM Comput. Surv.*, vol. 55, no. 13s, jul 2023. [Online]. Available: <https://doi.org/10.1145/3579845>
- [5] "Description of the CSAIL2019 Time Capsule Crypto-Puzzle," <https://people.csail.mit.edu/rivest/pubs/Riv19f.new-puzzle.txt>, accessed: 2024-03-21.
- [6] "Description of the LCS35 Time Capsule Crypto-Puzzle," <http://people.csail.mit.edu/rivest/pubs/Riv99b.lcs35-puzzle-description.txt>, accessed: 2024-03-21.
- [7] H. Ferraiolo and A. Regenscheid, "Cryptographic algorithms and key sizes for personal identity verification," National Institute of Standards and Technology, Tech. Rep., 2023.
- [8] P. Barrett, "Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor," in *Conference on the Theory and Application of Cryptographic Techniques*. Springer, 1986, pp. 311–323.
- [9] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of computation*, vol. 44, no. 170, pp. 519–521, 1985.
- [10] R. Müller, W. Meier, and C. F. Wildfeuer, "Area efficient modular reduction in hardware for arbitrary static moduli," *CoRR*, vol. abs/2308.15079, 2023. [Online]. Available: <https://doi.org/10.48550/arXiv.2308.15079>
- [11] M. Langhammer and B. Pasca, "Efficient FPGA modular multiplication implementation," in *FPGA '21: The 2021 ACM/SIGDA International Symposium on Field Programmable Gate Arrays, Virtual Event, USA, February 28 - March 2, 2021*, L. Shannon and M. Adler, Eds. ACM, 2021, pp. 217–223. [Online]. Available: <https://doi.org/10.1145/3431920.3439306>
- [12] C. H. Lim, H. S. Hwang, and P. J. Lee, "Fast modular reduction with precomputation," in *Proceedings of Korea-Japan Joint Workshop on Information Security and Cryptology (JWISC'97)*. Citeseer, 1997, pp. 65–79.
- [13] A. A. H. Abd-Elkader, M. Rashdan, E.-S. A. M. Hasaneen, and H. F. A. Hamed, "FPGA-based optimized design of Montgomery modular multiplier," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 68, no. 6, pp. 2137–2141, 2021.
- [14] R. Liu and S. Li, "A design and implementation of Montgomery modular multiplier," in *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2019, pp. 1–4.
- [15] W. Hasenplaugh, G. Gaubatz, and V. Gopal, "Fast modular reduction," in *18th IEEE Symposium on Computer Arithmetic (ARITH'07)*. IEEE, 2007, pp. 225–229.
- [16] S. ichi Kawamura and K. Hirano, "A fast modular arithmetic algorithm using a residue table (extended abstract)," in *Advances in Cryptology - EUROCRYPT '88, Workshop on the Theory and Application of Cryptographic Techniques, Davos, Switzerland, May 25-27, 1988, Proceedings*, ser. Lecture Notes in Computer Science, vol. 330. Springer, 1988, pp. 245–250.
- [17] E. Öztürk, "Design and implementation of a low-latency modular multiplication algorithm," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, no. 6, pp. 1902–1911, 2020.
- [18] M. Langhammer, S. Gribok, and B. Pasca, "Low-latency modular exponentiation for FPGAs," *IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines*, 2022.
- [19] A. Opasatian and M. Ikeda, "Lookup table modular reduction: A low-latency modular reduction for fast ECC processor," in *2023 IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS)*, 2023, pp. 1–6.
- [20] Z. Cao, R. Wei, and X. Lin, "A fast modular reduction method," *Cryptology ePrint Archive*, Paper 2014/040, 2014, <https://eprint.iacr.org/2014/040>. [Online]. Available: <https://eprint.iacr.org/2014/040>
- [21] B. Pasca, "High-performance floating-point computing on reconfigurable circuits," Ph.D. dissertation, École Normale Supérieure de Lyon, Lyon, France, Sep. 2011. [Online]. Available: http://tel.archives-ouvertes.fr/docs/00/65/41/21/PDF/Bogdan_PASCA-Calcul_flottant_haute_performance_sur_circuits_reconfigurables_2011.pdf
- [22] H. Wong, V. Betz, and J. Rose, "Comparing FPGA vs. custom CMOS and the impact on processor microarchitecture," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, 2011, pp. 5–14.