Scalable DSP optimized Montgomery Multiplier

Anonymous Author

Abstract—**This paper describes a scalable and efficient FPGA design for the Montgomery modular multiplication. Scheduling of operations and hardware organization are optimized for Ultrascale FPGA architectures featuring DSP48E2 slices on a Zynq Ultrascale+ platform. This results in a parallel systolic architecture inspired by previous work on the Finely Integrated Operand Scanning (FIOS) variant of the Montgomery multiplication algorithm. Our implementation can easily adapt to different width of the operands and performs a 256-bit, 512-bit and 1024-bit modular multiplications in 0.246 μs, 0.481 μs, and 0.953 μs using 5, 8 and 15 DSP blocks respectively.**

Index Terms—**Montgomery modular multiplication, arithmetic acceleration, Ultrascale FPGA, DSP**

I. Introduction

Modular multiplication and modular exponentiation using large numbers are critical to several popular cryptosystems such as RSA [1] or ECDSA [4] as well as applications such as the blockchain and postquantum cryptography protocols (CSIDH, ...). Unfortunately modular arithmetic requires divisions which have a significant cost in execution time for general purpose computers. In [5] Peter Montgomery unveiled an algorithm that would dramatically speed up repeated modular multiplications. It is especially well suited to the binary representation of numbers used by computers. There are many different ways to implement the Montgomery Multiplication (MM) algorithm in both software and hardware.

The Montgomery Multiplication algorithm has two main steps (see Section II): the multiplication of operands and the reduction of intermediate results. Since public key cryptography protocols usually involves large numbers, operands must be split into blocks of a given bit-width to fit general purpose processing hardware. This gives designers freedom as to how elementary block operations are scheduled and performed. In [2], Koç et al. described several implementations of the MM algorithm on a single symmetric processor. These implementations vary by the way operands are scanned during the process, how the multiplication and reduction steps are interleaved, and the order operations are performed in. Koç et al. found that the Coarlsy Integrated Operand Scanning (CIOS) variant was the most efficient on the general class of processor they considered. However when using ASIC[1] or reconfigurable hardware (FPGA[2]) designers can make use of parallel execution and relax data dependencies in arithmetic operations. Thus subsequent work on hardware implementations have essentially used asymmetric versions of the Finely Integrated Operand Scanning (FIOS) method from [2]. These implementations usually follow the naming convention "MWR2^kMM" (Multiple Word Radix 2^k Montgomery Multiplication): one of the operands is scanned using words of size k-bit while the other is scanned using words of standard sizes depending on the hardware used (16, 32-bit, ...). In [3] Tenca and Koç developed a scalable, parallel architecture for the Multiple Word Radix 2 Montgomery Multiplication algorithm (MWR2MM). This implementation replaced word-by-word multiplications from [2] by bit-by-word multiplications, which was especially suited to the logical operations of hardware implementations where multiplication can be a time-intensive operation. This work also laid down the foundations of the data dependencies within the Montgomery Multiplication algorithm and how operations could be parallelized in hardware. Improvements on this architecture have been made by decreasing its latency, execution time or area cost [7], [8]. Other high-radix asymmetric implementations have also been explored such as the MWR4MM, MWR8MM and MWR2^kMM [6] algorithms. Higher-radix scanning of the operands can lead to lower latency but they also increase the complexity of the block multiplications that must be performed, and can increase the hardware resource cost as well as decrease maximum frequency of the system. Asymmetric implementations also require careful consideration of the alignment and reunion of intermediate results which further increases complexity. Conversely a high-radix symmetric scanning of the operands can be particularly suited to the use of dedicated hardware such as arithmetic accelerators which can perform multiplications and additions using a high frequency. As such embedded multipliers (DSP slices) on FPGA have been used to implement variants of the Montgomery algorithm such as FIPS [9], CIOS [10] or FIOS [11].

The use of DSP blocks has sometimes been deemed too expensive, complex or slow for hardware implementations. However modern FPGA Ultrascale architectures feature efficient DSP48E2 arithmetic accelerator, which have a low equivalent area cost and provide high-frequency integer manipulation. We have taken advantage of modern DSP resources to build efficient and fast hardware FIOS Montgomery multipliers. Our design is sparing with hardware resource and scalable to operand sizes common in public key cryptography (128, 256, 512, 1024, 2048-bit, ...) without loss of performance. To the best of our knowledge, no other implementation is specifically fitted to this technology. All source, verification systems and hardware generation utilities are available at [17]

In this paper, we first describe the Montgomery multiplication algorithm and its FIOS variant in Section II. We then introduce the Processing Elements of our architecture and their scheduling in Section III. Our Implementation results are given in Section IV. Finally Section V concludes the paper.

[1]Application Specific Integrated Circuit

[2]Field Programmable Gate Array

II. The Montgomery Multiplication

A. Classical algorithm

Algorithm 1 Montgomery Multiplication

```
Input: $\bar{x}, \bar{y}, R, n, n'$
Output: $P = \overline{x.y} = x.y.R \mod n$
1: $P \leftarrow \bar{x}.\bar{y}$
2: $m \leftarrow P.n' \mod R$
3: $P \leftarrow P + m.n$
4: $P \leftarrow \frac{P}{R}$
5: if $P > n$ then
6:     $P \leftarrow P - n$
7: end if
8: return $P$
```

Algorithm 1 highlights the pseudocode of the Montgomery Multiplication as first described in [5].
Let n be a prime integer.
Let $R = 2^k$ with $k \geq \log_2 n$.
Let $n' \in \mathbb{N}$ such that $n.n' \mod R \equiv -1 \mod R$.
The Montgomery Multiplication algorithm manipulates representatives of large numbers in a n-residue system :
Let $(x, y) \in [0, n-1]^2$ and $\bar{x} = x.R \mod n$ be a n-residue of x (i.e $x = \bar{x}.R^{-1} \mod n$).
The Montgomery Multiplication algorithm computes $\overline{x.y} = \mathrm{MM}(\bar{x}, \bar{y}) = x.y.R \mod n$. For instance:

$$\begin{aligned} R &= 2^{64} \\ n &= \text{0xa7d6e790ae3067eb} \\ n' &= \text{0xe5bd3392613b773d} \\ \bar{x} &= \text{0xe894981b5d08deaf} \\ \bar{y} &= \text{0xf084efbc1492863d} \end{aligned}$$

$$P \leftarrow \bar{x}.\bar{y} = \text{0xda841505af2643e0928b3b307b7aa9b3} \quad (1)$$

$$\begin{aligned} P.n' &= \text{0xc419a656dd62}\ldots\text{ea91049b605da4a7} \\ m \leftarrow P.n' \mod R &= \text{0xea91049b605da4a7} \end{aligned} \quad (2)$$

$$P + m.n = \text{0x1744d986e513b9f760000000000000000} \quad (3)$$

$$P \leftarrow \frac{P}{R} = \text{0x1744d986e513b9f76} \quad (4)$$

Fig. 1: 64-bit Montgomery Multiplication example

Figure 1 demonstrates the operations which occur during a 64-bit Montgomery Multiplication. For the sake of the example numbers are represented in hexadecimal. Given n-residue operands $\bar{x}$ and $\bar{y}$, the multiplication steps occurs at (1). At that point $P \mod n = \bar{x}.\bar{y} \mod n$: the principle behind the Montgomery Multiplication is to derive a number from P which is a multiple of R but has the same value $\mod n$. During step (2) the m parameter is calculated through a partial multiplication and use of the $\mod R$ operation. Since R is taken to be a power of 2, the $\mod R$ operation consists in selecting the least significant bits of the operand, which can easily be done in hardware. A multiple of n is generated by adding $m.n$ to $\bar{x}.\bar{y}$ during (3). Since $m.n$ is a multiple of n, this operation does not change the value of $P \mod n$. The reduction step (4) concludes the algorithm. Since P is now a multiple of a power of 2, it can easily be divided by R without loss of information by selecting the most significant bits of the operand. At that point $P \equiv \bar{x}.\bar{y}.R^{-1} \equiv x.y.R \equiv \overline{x.y} \mod n$.
n-residues with shape $\bar{x} = x.R \mod n$ are said to be in the Montgomery domain. Since the result of the Montgomery Multiplication of elements in the Montgomery domain is still in the Montgomery domain, cryptographic protocols are typically carried out entirely within the Montgomery domain. Conversion inside and outside of the Montgomery domain can be performed by computing $\bar{x} = \mathrm{MM}(x, R^2)$ and $x = \mathrm{MM}(\bar{x}, 1)$ respectively. The MM algorithm is thus especially suited to protocols where multiple modular multiplications must be performed with respect to the same modulus n and few conversion operations are performed.

At the end of step (4) the only guarantee on the size of P is that $P \leq 2.n$, hence an additional subtraction might be required to further decrease the size of P and contain intermediate results in cryptographic protocols (see algorithm 1 line 5). Fortunately, [14] has shown that intermediate results in the Montgomery domain can be contained without resorting to a final subtraction by choosing $R > 4.n$, which is the model we have implemented.

The strength of the Montgomery Multiplication algorithm resides in the fact that it can perform modular multiplications without trial divisions, which are very costly to implement in hardware, requiring instead multiplications and $\mod R$ and $\frac{1}{R}$ operations, which are easily implemented in hardware.

B. The FIOS variant

In order to be general purpose, processing elements have intrinsic limitations on the width of the operands they can manipulate. To perform arithmetic operations on data of large width such as the ones used in cryptosystems (256, 516, 1024, 2048-bit, ...) operands must be sliced into blocks of smaller width. The algorithm requires on one hand the multiplication of the operand, and on the other hand the reduction of the result. This multiplication and reduction can be interleaved in several different ways which will have an effect on the performance and resource cost of the implementations (whether in software or hardware).

The Finely Integrated Operand Scanning (FIOS) method uses two loops to scan each block of the operands, and

performs a reduction operation as soon as a block of the result is available. (see Algorithm 2).
Let $w \in \mathbb{N}$. Let $W = 2^w$.
Let $\bar{x} = \sum_{i=0}^{s-1} \bar{x}_i.2^{i.w}$. Let $k = s.w \implies R = 2^{s.w}$ with s the number of w-bit blocks required to slice our operands.

Algorithm 2 FIOS

```
Input: $\bar{x}, \bar{y}$
Output: $P = \overline{x.y} = x.y.R \mod n$
 1: $P \leftarrow 0$
 2: for $i = 0$ to $s-1$ do
 3:     $P_0 \leftarrow P_0 + \bar{x}_i.\bar{y}_0$
 4:     $m_i \leftarrow P_0.n'_0 \mod W$
 5:     $P_0 \leftarrow P_0 + m_i.n_0$
 6:     $P_0 \leftarrow \frac{P_0}{W}$
 7:     for $j = 1$ to $s-1$ do
 8:         $P_{j-1} \leftarrow P_{j-1} + \bar{x}_i.\bar{y}_j + m_i.n_j + P_j$
 9:         $P_j \leftarrow \frac{P_{j-1}}{W}$
10:         $P_{j-1} \leftarrow P_{j-1} \mod W$
11:     end for
12: end for
13: if $P > n$ then
14:     $P \leftarrow$ SUB$(P, n)$
15: end if
16: return $P$
```

Iterations of the outer loop (at line 2) of the FIOS variant can be processed in parallel, which makes it a good candidate for hardware acceleration.

III. Systolic Hardware Architecture

A. The DSP48E2 Slice

FPGA hardware devices have collections of embedded multipliers called Digital Signal Processing blocks meant to accelerate arithmetic operations and Xilinx Ultrascale architectures introduce new improved DSP48E2 slices. Figure 2 highlights the main features used in the design. DSP48E2 slices have a $26 * 18$-bit signed multiplier used here as a $17 * 17$-bit unsigned multiplier. DSP48E2 slices also feature a 4-input 48-bit adder up from a 3-input adder available in the previous generation of DSP blocks. The OPMODE signal controls four multiplexers which allow for fine control of the inputs to the adder (see Table I). Finally, DSP blocks have a native support for 17-bit right-shift feedback of their output to the adder which allows for easy alignment of intermediate results. Thus a regular 17-bit slicing of our operands is simple and fits our hardware resources.

OPM	000100000_b	110000101_b	000000101_b
$P \leftarrow$	P	$A * B + C$	$A * B$
OPM	111100101_b	000100101_b	001100000_b
$P \leftarrow$	$P >> 17 + A * B + C$	$P + A * B$	$P >> 17$

TABLE I: DSP operations controlled by the OPMODE signal

DSP blocks are the fastest tool available to perform arithmetic operations on the FPGA device. Internal optional registers can be used but there must be at least 3 stages of registers on the multiplier datapath (as pictured Figure 2) for the DSP block to reach its maximum operating frequency (700 MHz up from about 400 MHz for 1 or 2 register stages). Thus maintaining that maximum frequency constrained the rest of the architecture to be finely pipelined and for no arithmetic operations to be performed outside of DSP blocks.

B. Processing Elements

Each Processing Element (PE) used in the design is responsible for an iteration i of the outer loop of the FIOS algorithm (see algorithm 2 line 2). Following the constraints that a single three-stages pipelined DSP block must handle all arithmetic operations within the PE, operations' scheduling is described in Algorithm 3 and is handled by an internal Finite State Machine which controls the logic of the PE. Since the m_i parameter is essential to the reduction process it must be computed as soon as possible. However it is dependent on the computation of the word which will be reduced, performed at line 1.

Algorithm 3 PE i operations

```
Input: $x_i, y, P_{i-1}(\mod 2^{17})$
 1: $\mathrm{P}_i \leftarrow x_i.y_0 + \mathrm{P}_{i-1}$                     }
 2: $\mathrm{P}_i \leftarrow x_i.y_1 + \mathrm{P}_{i-1}$                     }
 3: $\mathrm{P}_i \leftarrow x_i.y_2 + \mathrm{P}_{i-1}$                     }
 4: $\mathrm{P}_i \leftarrow m_i$                                            }
 5: $\mathrm{P}_i \leftarrow x_i.y_3 + \mathrm{P}_{i-1/dly}$                 } A
 6: $\mathrm{P}_i \leftarrow x_i.y_4 + \mathrm{P}_{i-1}$                     }
 7: $\mathrm{P}_i \leftarrow m.n_0 + C_i$                                    }
 8: $\mathrm{P}_i \leftarrow \mathrm{P}_i >> 17 + m_i.n_1 + C_i$             }
 9: $\mathrm{P}_i \leftarrow \mathrm{P}_i >> 17 + m_i.n_2 + C_i$             }
10: $\mathrm{P}_i \leftarrow \mathrm{P}_i >> 17 + m_i.n_3 + C_i$             } B
11: $\mathrm{P}_i \leftarrow \mathrm{P}_i >> 17 + m_i.n_4 + C_i$             }
12: for $j = 5$ to $s-6$ do
13:     $\mathrm{P}_i \leftarrow \mathrm{P}_i >> 17 + x_i.y_j + \mathrm{P}_{i-1/dly}$   } C
14:     $\mathrm{P}_i \leftarrow \mathrm{P}_i + m_i.n_j$                               }
15: end for
16: $\mathrm{P}_i \leftarrow \mathrm{P}_i >> 17$
```

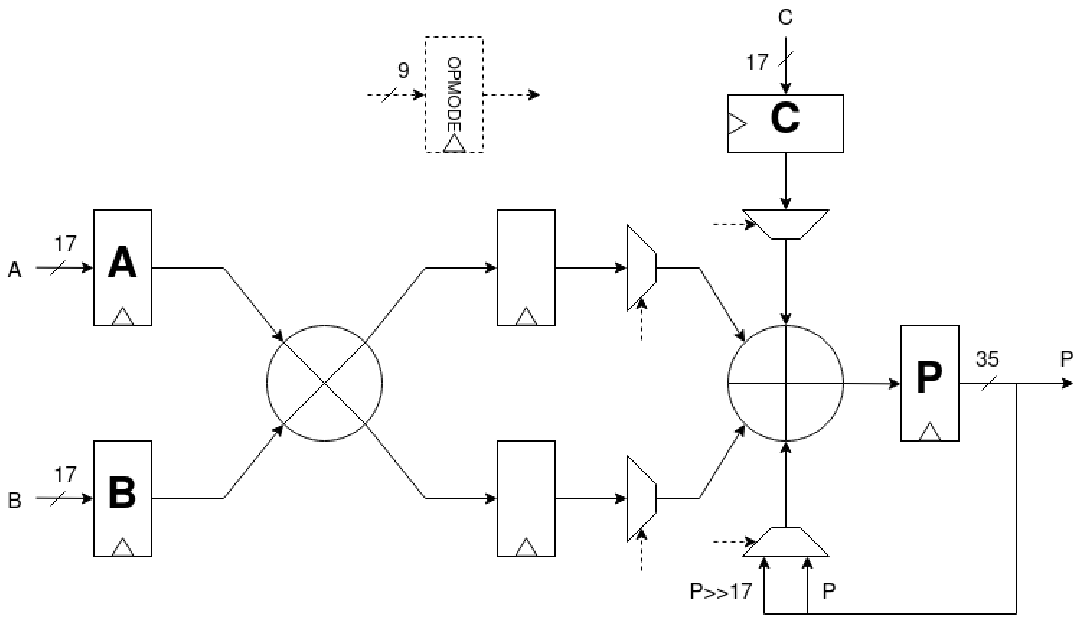

Fig. 2: DSP48E2 features

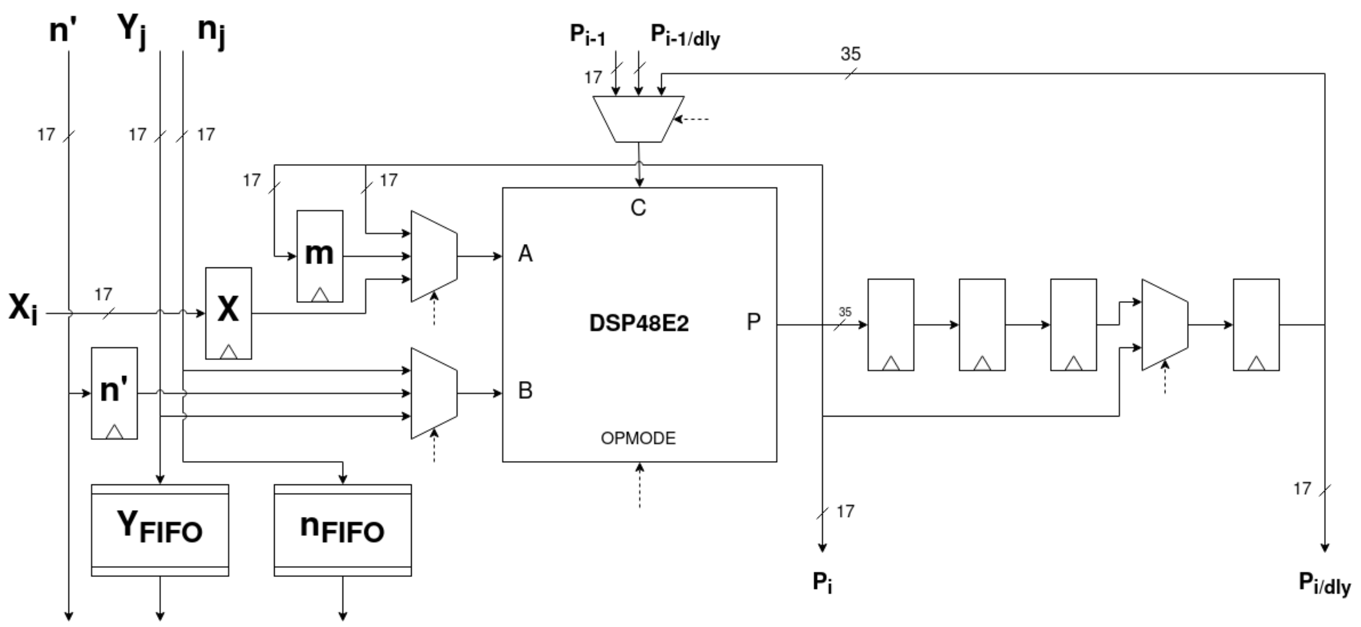

Fig. 3: Structure of a PE

Since data must travel through the pipeline and in order not to waste computation cycles, independent intermediate results are computed at lines 2, 3, 5 and 6. Lines 7 through 11 catch up to these intermediate results. Thereafter the PE element adopts a regular behavior until the end of computations.

It is critical that the operation at line 8 be performed as soon as possible since the next PE in the chain can only start processing data once this value has been computed and transmitted to the next DSP block. Figure 4 illustrates this data dependency which was first introduced in [3]. Consequently a new PE can only start processing data every 9 clock cycles. This means that different PEs will always manipulate different operands y_j and n_j at any given time, which is beneficial to the scalability and the performance of the design: operands can be circulated between the PEs through the use of First In First Out queues (see $\mathrm{Y_{FIFO}}, \mathrm{n_{FIFO}}$ Figure 3). Therefore signal density does not scale with the total number of data blocks s to be manipulated: congestion will not increase and maximum

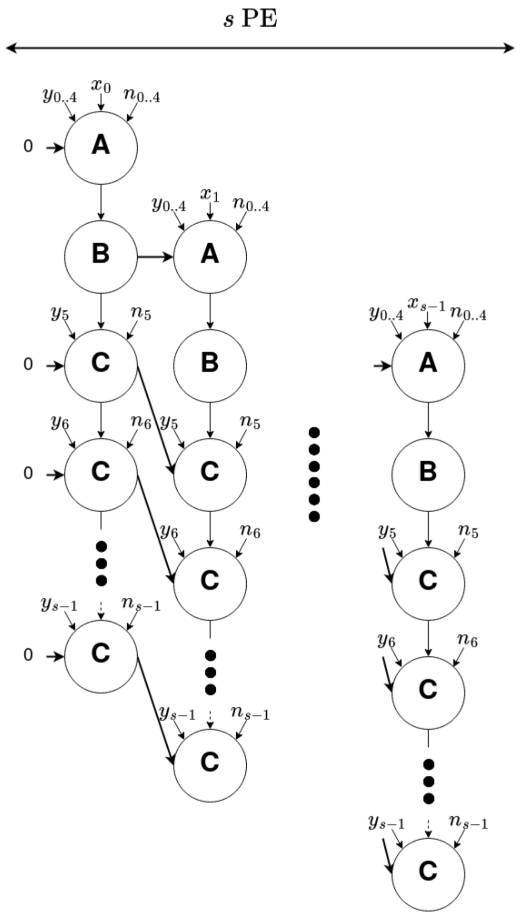

Fig. 4: Illustration of data dependencies in the FIOS algorithm

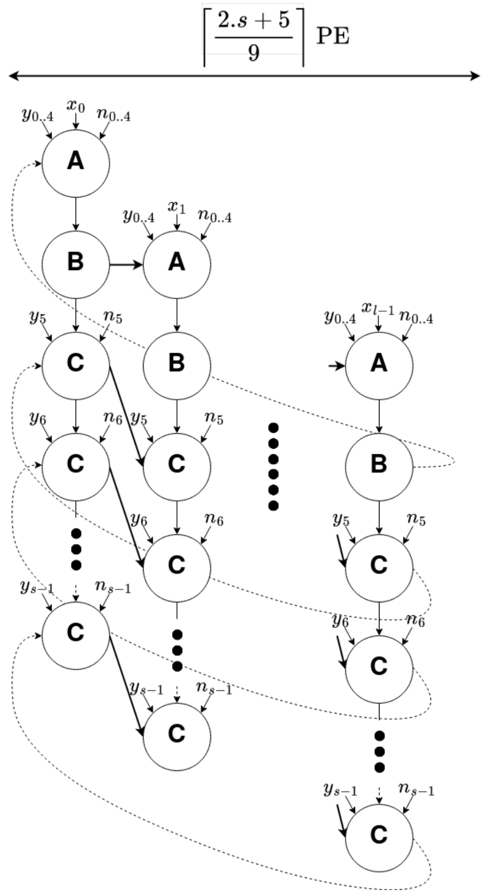

Fig. 5: Illustration of the feedback loop into the first processing element

frequency will not decrease significantly as the total width of the operands grows.

All logic within the PE essentially serves to accommodate these operations as is shown in Figure 3. Input registers are used to store fixed operands x_i, m_i, n' which are used all through the operations of the i-th PE. Input multiplexers are used to select between the different operands and feed them to either the multiplicative or additive inputs of the DSP block. Finally the output delay line is used to store the first few intermediate results calculated and to synchronize between different processing elements.

A PE takes a total of $2.s+5$ clock cycles to perform all of its operations. Considering the 9 cycles delay between PEs, only $l = \left\lceil \frac{2.s+5}{9} \right\rceil$ processing elements are required for the complete FIOS algorithm because the output of the last PE in the chain can be looped back as input to the first PE in the chain as soon as it has completed its loop iteration. This feedback loop is described Figure 5. This significantly cuts down on the resource cost of the architecture.

Finally a complete Montgomery Multiplication FIOS using this architecture requires a total of $11.s-4$ clock cycles.

IV. Results

The model has been developed using the Verilog hardware description language for implementations targeting a Zynq Ultrascale+ FPGA architecture (board ZCU104, part xczu7ev-ffvc1156-2-e). The correctness of calculations has been verified using post-synthesis simulations and reference python software implementations. The design has been synthesized, implemented and made compatible with the Zynq Ultrascale+ System on Chip (SoC) for co-design applications and communication with the processor through use of the AXI protocol. All design sources, testbenches, verification systems and results are open and available at [17].

In order to analyze relevant area metric for DSP blocks, the concept of equivalent LUT[3] cost[4] has been used: $\text{LUT}_{\text{eq}} = \frac{\text{LUT}_{\text{tot}}}{\text{DSP}_{\text{tot}}}$. Figure 6 highlights implementations results of the design's maximum frequency and the total equivalent LUT area cost as a function of the operand width on a logarithmic scale for common cryptographic sizes ranging from 128 bits to 4096 bits. This figure characterizes the quality

[3]Look-Up Tables are elementary components used in FPGA devices

[4]$\text{LUT}_{\text{eq}} = \frac{230400}{1728} = 133$ on the ZCU104 board

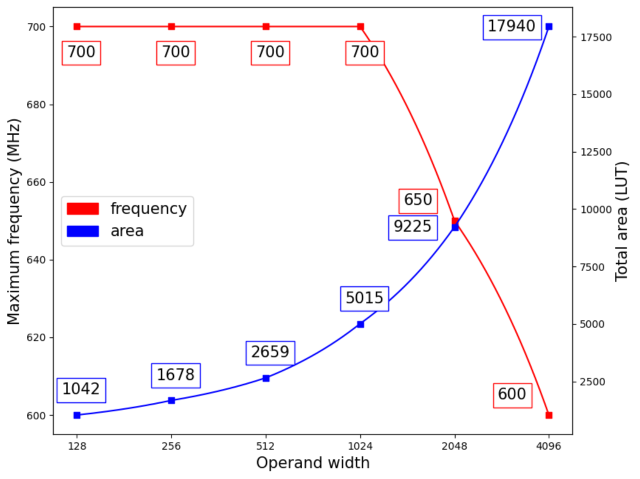

Fig. 6: Maximum Frequency and area metric as a function of operand width

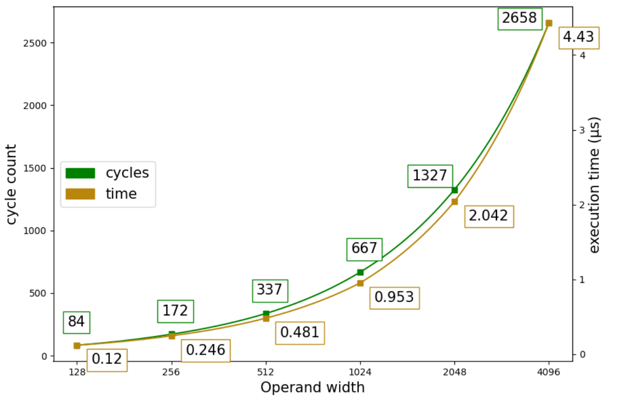

Fig. 7: Cycle count and execution time as a function of operand width

of the design's scalability. As expected there is little loss of performance for sizes ranging from 128 to 1024 bits: the maximum operating frequency remains stable at 700 MHz (close to the maximum frequency of DSP blocks), and the area cost grows slower than the bit-width as opposed to the exponential growth in resource cost which occurs in congested systems. However slight congestion starts to occur for very large sizes of the operands at 2048 and 4096 bits, which causes a slight drop in the maximum frequency. Figure 7 gives our cycles and execution time implementation results for multiple operand width.

We compare our implementation results against recent work on Montgomery multiplication implementations from the state of the art in Table II. Note that the last column is the equivalent LUT area-time product (AT), which is a measure of the efficiency of the design: the lower the AT, the more performance the design gets out of the resources it uses.

The first aspect to consider is that the maximum frequency of our design is the highest of all the designs we compare against. This is mainly due to the fine pipelining of the design and the fact that no arithmetic operations are performed outside of DSP blocks. This maximum frequency is also stable for different width of the operands (for sizes 256, 512 and 1024-bit), contrary to the designs from [10] and [8], whose maximum frequency decreases as operand width grows.

Some implementations use no DSP blocks : for instance [7] implements a 1024-bit Montgomery multiplication, but has a high latency, a low frequency and a high LUT cost, which leads to poor execution speed and very high area-time product. [8] which is implemented on an older FPGA architecture does not use DSP blocks either, but manages to have a low LUT cost. Thus in spite of its low frequency and high-latency, its area-time product is kept relatively low.

We compare our 512 and 1024-bit design with the design from [10], which exhibits a low and constant latency in exchange for low maximum frequency and high DSP use. Consequently our 1024-bit design is faster and has a 5.6 time smaller area-time product.

Several implementations seem to outperform our design. The 503-bit design from [12] is 4 times faster than our 512-bit design thanks to its extremely low latency and moderate maximum frequency. However it uses many more DSP blocks and LUTs which causes it to have a slightly higher area-time product. The 256-bit design is the closest from ours in terms of overall performance: its maximum frequency of 598 MHz and its 143 clock cycles latency make it about 3% faster than our 256-bit implementation. It also has a very low LUT count, however our use of 40% fewer DSP blocks grant us a slightly lower area-time product.

Finally we compare against the implementation from [16] which is the only one to use an Ultrascale FPGA architecture like we do. Its latency and maximum frequency are not given, but its execution time is about 2.8 times lower than ours, in exchange for the highest DSP cost and LUT cost of all the 256-bit implementations. This leads it to have an 8.4 times higher area-time product than our implementation.

V. Conclusion

We have successfully developed and tested a new scalable implementation of the Montgomery FIOS method, optimized to take advantage of the 4-input adder capabilities of the modern DSP48E2 slices used in Ultrascale FPGA architectures. This model has the highest maximum frequency, uses a low number of DSP blocks and LUTs and has the lowest area-time product compared with state of the art implementations, which makes it particularly suited to embedded devices where resources are sparse. Perspectives for future work include the use of this Montgomery Multiplier in a complete cryptographic protocol. Depending on the cryptographic protocol used, it might be advantageous to consider using the unfolded version

Design	FPGA	Width	Frequency (MHz)	Latency	DSP	LUT	Time (μs)	AT
Our	**Zynq+**	**256**	**700**	**172**	**5**	**1013**	**0.246**	**413**
		512	**700**	**337**	**8**	**1595**	**0.481**	**1279**
		1024	**700**	**667**	**15**	**3020**	**0.953**	**4779**
[10]	Artix 7	512	106	66	57	1789	0.622	5828
		1024	65.0	66	161	5242	1.01	26921
[7]	Spartan VII	1024	104	1056	0	5310	10.0	53100
[8]	Virtex 6	256	176	257	0	620	1.46	905
		512	99.7	515	0	1222	5.16	6305
[12]	Virtex 7	503	226	23	44	13775	0.102	1405
[13]	Virtex 7	256	598	143	9	634	0.239	437
[16]	Kintex+	256	-	-	248	9450	0.0852	3480

TABLE II: Comparison of implementation results

of our implementation (see Figure 4) which uses more resources, but is able to compute multiple Montgomery Multiplications simultaneously. Finally our design is a good candidate for co-design applications since it supports communication with the processor of a development board.

REFERENCES

[1] R. L. Rivest, A. Shamir, and L. Adleman. 1978. A method for obtaining digital signatures and public-key cryptosystems. Commun. ACM 21, 2 (Feb. 1978), 120–126.

[2] C. Kaya Koc, T. Acar and B. S. Kaliski, ”Analyzing and comparing Montgomery multiplication algorithms,” in IEEE Micro, vol. 16, no. 3, pp. 26-33, June 1996.

[3] Tenca, Alexandre F., and Cetin K. Koç. ”A scalable architecture for montgomery nultiplication.” International Workshop on Cryptographic Hardware and Embedded Systems. Springer, Berlin, Heidelberg, 1999.

[4] Koblitz, Neal. ”Elliptic curve cryptosystems.” Mathematics of computation 48.177 (1987): 203-209.

[5] Montgomery, Peter L. ”Modular multiplication without trial division.” Mathematics of computation 44.170 (1985): 519-521.

[6] Tenca, Alexandre F., Georgi Todorov, and Cetin K. Koç. ”High-radix design of a scalable modular multiplier.” International Workshop on Cryptographic Hardware and Embedded Systems. Springer, Berlin, Heidelberg, 2001.

[7] M. Huang, K. Gaj and T. El-Ghazawi, ”New Hardware Architectures for Montgomery Modular Multiplication Algorithm,” in IEEE Transactions on Computers, vol. 60, no. 7, pp. 923-936, July 2011.

[8] 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,” in IEEE Transactions on Circuits and Systems II: Express Briefs, vol. 68, no. 6, pp. 2137-2141, June 2021.

[9] Gastaldo, Paolo, Giovanni Parodi, and Rodolfo Zunino. ”Enhanced montgomery multiplication on dsp architectures for embedded public-key cryptosystems.” EURASIP Journal on Embedded Systems 2008 (2008): 1-9.

[10] Mrabet, Amine, et al. ”A scalable and systolic architectures of Montgomery modular multiplication for public key cryptosystems based on DSPs.” Journal of Hardware and Systems Security 1.3 (2017): 219-236.

[11] Mondal, Arpan, et al. ”Efficient FPGA implementation of Montgomery multiplier using DSP blocks.” Progress in VLSI Design and Test. Springer, Berlin, Heidelberg, 2012. 370-372.

[12] S. M. -H. Farzam, S. Bayat-Sarmadi, H. Mosanaei-Boorani and A. Alivand, ”Fast Supersingular Isogeny Diffie–Hellman and Key Encapsulation Using a Customized Pipelined Montgomery Multiplier,” in IEEE Transactions on Circuits and Systems I: Regular Papers, vol. 69, no. 3, pp. 1221-1230, March 2022

[13] G. Gallin and A. Tisserand, ”Generation of Finely-Pipelined GF(PP) Multipliers for Flexible Curve Based Cryptography on FPGAs,” in IEEE Transactions on Computers, vol. 68, no. 11, pp. 1612-1622, 1 Nov. 2019

[14] Walter, Colin D.. “Montgomery exponentiation needs no final subtractions.” Electronics Letters 35 (1999): 1831-1832.

[15] El Khatib, R., Azarderakhsh, R., Mozaffari-Kermani, M. (2019). Optimized Algorithms and Architectures for Montgomery Multiplication for Post-quantum Cryptography. In: Mu, Y., Deng, R., Huang, X. (eds) Cryptology and Network Security.

[16] Javad Ahsan, Mohammad Esmaeildoust, Amer Kaabi, Vahid Zarei, Efficient FPGA implementation of RNS Montgomery multiplication using balanced RNS bases (2022)

[17] https://github.com/anonymHost2023/Scalable-DSP-opt-FIOS-MM.git