LLMonPro: Low-Latency Montgomery Modular Multiplication Suitable for Verifiable Delay Functions

İsmail San
Department of Electrical and Electronics Engineering
Eskişehir Technical University
Eskişehir, Turkey, 26555
isan@eskisehir.edu.tr

Abstract

This study presents a method to perform low-latency modular multiplication operation based on both Montgomery and Ozturk methods. The design space exploration of the proposed method on a latest FPGA device is also given. Through series of experiments on the FPGA using an high-level synthesis tool, optimal parameter selection of the proposed method for the low-latency constraint is also presented for the proposed technique.

1 INTRODUCTION

Modular multiplication is a key operation in several cryptographic algorithms. Most of the public key cryptosystems such as RSA [1], Diffie-Helman [2], ElGamal [3] and DSA [4] are essentially based on modular arithmetic. The primary operation in such systems is the modular multiplication and it inherently requires a division which is very inefficient due the length of the modulus (e.g., 1024-bit).

Modular multiplication $Z = A \cdot B \bmod M$ simply consists of multiplying two n-bit operands, A and B and produces a result in a range of $[0, M-1]$. When M is also n-bit integer, a $2n$-bit result of plain multiplication needs to be reduced to n-bit. Thus, modular multiplication decreases the result of the multiplication into the operand length (n-bit) by performing a reduction process with respect to a given modulus M. Modular reduction inherently requires a division by a modulus M and finding the remainder of this division. There are many techniques to make this reduction area-efficient or throughput efficient. Montgomery modular multiplication method [5], which is proposed in 1985 by Peter Montgomery, first transforms representation of numbers from the ring $\mathbb{Z}_n$ to a Montgomery residual representation where the computations are more efficient since the division and modulo operations are done by a power of 2 and taking the remainder with respect to a number (R) that is power of 2. Montgomery reduction replaces division by multiplications and additions. Since there is no division, it makes Montgomery reduction advantageous in hardware implementations. In literature, there are several different hardware implementations from fastest to smallest, area- and throughput-efficient or compact implementations [6–10].

Emerging schemes like verifiable delay function (VDF) need a low-latency implementations of the modular arithmetic. VDFs are basically constructed to take a definite amount of time to perform its underlying function even if there are infinitely many concurrent execution units, that is, the operation is inherently sequential. Thus, low-latency modular multiplication algorithm and its hardware architecture with lowest possible latency are significantly important to make VDF schemes practical. Therefore, for hardware implementations of VDF, the critical path of the circuit and the number of total execution cycles are both important optimization constraints for modular multiplication.

Ozturk proposes a modular multiplication algorithm suitable for low-latency circuit implementations [11]. Modular multiplication algorithm proposed in [11] allows one to implement $O(\log n)$ depth circuits for the modular multiplication and squaring operations. One of the contributions in [11] to attain the low-latency is in the reduction step. After performing the multiplication in a redundant form to minimize the critical path, the partial products are reduced into the operand length by looking at a fixed table (look-up table). This is possible since a fixed modulus (M) is assumed to be used where this is practical for VDF like schemes.

In this paper, our objective is to find a computational model aimed at the low-latency Montgomery modular multiplication. In order to achieve the low-latency for a Montgomery multiplication, first, the multiplications should be performed with a lowest circuit depth. This is already be investigated thoroughly in [11] and redundant-representation polynomial multiplication has been presented in [11, Algorithm 7]. Thus, we adapt it to our method to perform the multiplications in the Montgomery modular multiplication (see the first 3 lines in Algorithm 1). Second, we observed some of the partial products in Montgomery method are not used in the final computation of the result. Hence, we change the algorithm to use only the necessary partial products and leave the job to the high-level synthesis (HLS) tool to remove unnecessary logic and multiplication circuits.

High Level Synthesis (HLS) tools enables an automatic translation from high level C/C++ descriptions into Register Transfer Level (RTL; e.g., Verilog, VHDL) hardware architecture designs. HLS can find the unused logic and unused multipliers by its compiler optimization passes. HLS also allows one to design at a high level of abstraction that offers one to focus on high level concepts within less amount of design time and it is easier to consider many design parameters.

There are two major drawback of the approach presented in [11] which mainly uses look-up tables in the reduction step. One of them is that the modulus needs to be fixed. Offline computations of this look-up tables are required for each modulus change. They need to be separately modified for each modulus. The second one is that the look-up tables require high amount of on-chip memory, especially when the modulus size is getting larger. However, if the modulus is fixed and there are available on-chip memory for the look-up tables that are generated for the selected modulus, then the method proposed by Ozturk [11] is achieving lower-latency compared to the one presented in this paper. The main advantage of using Montgomery based approach presented in this study is that it is easy to replace the modulus at run-time and there is no need a big amount of look-up tables and its pre-computation.

2 Low-Latency Method for Montgomery Modular Multiplication

There are several different hardware architectures available in the open literature for Montgomery modular multiplication. However, a computational description aimed at the low-latency is missing. In this section, we present a low-latency computational model for Montgomery modular multiplication.

Algorithm 1 shows the Montgomery modular multiplication method (MonPro) which is a more efficient way of performing the necessary arithmetic with an overhead of transforming the numbers into Montgomery residual domain. If several modular multiplications are required to be performed, as in the case of VDF, i.e., many square operations that need to be done one after another, then this overhead is negligible in terms of latency. There are totally three large (e.g., 1024-bit) multiplications involved in Montgomery modular multiplication. However, we observed that some partial products are not used in the calculation of the modular multiplication result.

Algorithm 2 shows another computational description of Montgomery modular multiplication operation based on our observation that some partial products (Q_1 and S_0) of the multiplication results are not needed in the final summation (line 4 in Algorithm 2). After this observation, the computation of the partial products of the three multiplications are optimized. Only the logic to compute the necessary partial products required in the summation to get the result can be described with a low-level hardware description language. If it is described at high-level correctly, then logic

Algorithm 1 Montgomery Modular Multiplication.

Input: Odd n-bit modulus M, two operands $A, B < M$, Montgomery radix $R = 2^n$, and the pre-computed constant $M' = -M^{-1} \bmod R$.

Output: Montgomery multiplication result:
$Z = \texttt{MonPro}(A, B) = A \cdot B \cdot R^{-1} \bmod M$

1. $T \leftarrow A \cdot B$
2. $Q \leftarrow T \cdot M' \bmod R$
3. $Z \leftarrow (T + Q \cdot M)/R$
4. **if** $Z \geq M$ **then**
5. $\quad Z \leftarrow Z - M$
6. **end if**
7. **return** Z;

synthesizer via HLS do its task: (1) optimize the logic and (2) remove the unused parts of the execution (unused partial products) in terms of logic.

Algorithm 2 Our Reduced Montgomery Modular Multiplication.

Input: Odd n-bit modulus M, two operands $A, B < M$, Montgomery radix $R = 2^n$, and the pre-computed constant $M' = -M^{-1} \bmod R$.

Output: Montgomery multiplication result:
$Z = \texttt{RedMonPro}(A, B) = A \cdot B \cdot R^{-1} \bmod M$

1. $T \leftarrow A \cdot B$ ▷ $T = (T_1, T_0)$ where $T = T_1 \cdot 2^n + T_0$
2. $Q \leftarrow (T_1 \cdot 2^n + T_0) \cdot M' \mod R$
   $\leftarrow T_0 \cdot M' \mod R$
3. $S \leftarrow Q \cdot M = S_1 \cdot 2^n + S_0$
4. $Z \leftarrow \frac{(T_1 \cdot 2^n + T_0) + (S_1 \cdot 2^n + S_0)}{2^n}$
   $\leftarrow (T_1 + S_1 + \frac{T_0 + S_0}{2^n})$ ▷ $\frac{T_0 + S_0}{R} \in \{0, 1\}$ and $0 \leq T_0, S_0 \leq R - 1$
5. **if** $Z \geq M$ **then**
6. $\quad Z \leftarrow Z - M$
7. **end if**
8. **return** Z;

Efficient high-level implementation of `RedMonPro` is described in Algorithm 3. This description of the Montgomery modular multiplication is also efficient in terms of high-throughput and software implementations since there is a reduction in terms of a total computation of the original Montgomery product algorithm (Algorithm 1).

Algorithm 3 High-level implementation of the Algorithm 2

```
RedMonPro(A,B, M, Mprime, n){}
    R = 2**n
    T = A*B
    T0 = T % r
    T1 = T >> n

    if T0 == 0:
        return T1

    Q = (T0 * Mprime) % R

    S = Q*M
    S0 = S % R
    S1 = s >> n

    Z = T1 + S1 + 1
    if Z >= M:
        Z = Z-M
    return Z
```

Algorithm 4 is reprinted from [11] to show the computational model that we have adapted in the plain multiplications required in the Montgomery modular multiplication. Throughout this paper, it is denoted as `OzturkPolMul`. `OzturkPolMul` requires two operands in $k + 1$ digits and performs the plain multiplication with a low-latency (see [11] for circuit depth analysis of the algorithm). It produces $2k + 2$ digit multiplication result.

Algorithm 5 shows the proposed computational description for the low-latency approach to compute the Montgomery modular multiplication utilizing the Ozturk's redundant-representation polynomial multiplication. There are three invocations to redundant-representation polynomial Multiplication and each results in $2n$-bit result, however some parts of the results are utilized to get the final result. Line 4 in Algorithm 2 is implemented in Algorithm 5 from the Line 4 to 16. E is a internal variable to accumulate the T_1 and S_1 digits with a carry in value of 1. The variable S is used to contain the result of $Q_0 \cdot M$. Note that the lower part (S_0) is not used. T is used to contain the both parts of the result of $A \cdot B$. and the higher part (T_1) is accumulated in E.

Algorithm 4 Redundant-Representation Polynomial Multiplication proposed by Ozturk in [11, Algorithm 7].

Input: $A(x) = \sum_{i=0}^{k} A_i \cdot x^i,\ 0 \leq A_i < 2^{d+1}$
$B(x) = \sum_{i=0}^{k} B_i \cdot x^i,\ 0 \leq B_i < 2^{d+1}$
Output: Polynomial multiplication result:
$C = \texttt{OzturkPolMul}(A, B)$
$C(x) = \sum_{i=0}^{2k+2} C_i \cdot x^i,\ 0 \leq C_i < 2^{d+1}$

for $i \leftarrow 0$ **to** $2k+2$ **do**
  $D_i = 0$
end for
for $i \leftarrow 0$ **to** k **do**
  for $j \leftarrow 0$ **to** k **do**
    $T = A_i \cdot B_j$ ▷ $T = (T_2, T_1, T_0)$
    $D_{i+j} = D_{i+j} + T_0$
    $D_{i+j+1} = D_{i+j+1} + T_1$
    $D_{i+j+2} = D_{i+j+2} + T_2$ ▷ $0 \leq T_2 < 2^2$
  end for
end for
for $i \leftarrow 0$ **to** $2k+2$ **do**
  $C_i = 0$
end for
$C_{2k+2} = D_{2k+2}$ ▷ $0 \leq D_{2k+2} < 2^2$
for $i \leftarrow 0$ **to** $2k+1$ **do** ▷ $\forall i\ D_i = (D_{i_1}, D_{i_0})$
  $C_i = C_i + D_{i_0}$
  $C_{i+1} = C_{i+1} + D_{i_1}$
end for
return C; ▷ $0 \leq C_i < 2^{d+1}$ where $i \in [0, 2k+2]$

3 High-level synthesis of Low-Latency Montgomery Multiplication Method

In this section, the HLS descriptions of the proposed method that is introduced in the Section 2 are presented.

Algorithm 6 shows the HLS implementation of Algorithm 4 that is the redundant-representation polynomial multiplication proposed by Ozturk in [11, Algorithm 7] where we used as a large multiplier in our method. This algorithm accepts two arguments `x` and `y` with $k+1$ digits and produces an output `z` with $2k+2$ digits. The arguments and the output arrays are decomposed to create a wide register by using `ARRAY_RESHAPE` pragma via Vivado HLS. For each multiplication in Algorithm 4, $T = A_i \cdot B_j$, is computed with calling a `mulhilo` function (see Algorithm 8). The necessary additions to accumulate the partial product on the variable `D` is described in Algorithm 6 between the line 34 to 36. Last accumulation is performed over the variable C and described between the lines after 40.

The HLS implementation of the proposed low-latency Montgomery modular multiplication is given in Algorithm 7. As one can see, it invocates `OzturkPolMul` method three times to compute T, Q, and S (see the `big_mul_wo_lut` function in Algorithm 6).

Algorithm 8 is the HLS description of the core function for $d+1$-bit multiplication operation used in HLS implementation of the proposed algorithm. It accepts two arguments `x` and `y` with $d+1$-bit and produces three results `hi`, `lo` and `redundant` with d, d, and 2-bit respectively. This function is simply implemented on FPGA with DSP building blocks.

4 Results

We captured our architectures in the C language and prototyped our hardware accelerators on a Xilinx Virtex Ultrascale+ FPGA (available on VCU118 FPGA Development Kit). Tables 1 and 2 summarize our synthesis results measured with Vivado HLS 2019.2. Table 3 shows the FPGA implementation results of the proposed Low-Latency Modular Multiplication based on Montgomery and Ozturk on the VCU118 Development Kit. Note that the corresponding results given in Tables 1 and 2 are further optimized when they are placed and routed on real FPGA platform with Vivado 2017.4 Design Suite.

Algorithm 5 Low-latency Montgomery Modular Multiplication using Ozturk based Redundant-Representation Polynomial Multiplication.

Input: Odd n-bit modulus M, two operands $A, B < M$, Montgomery radix $R = 2^n$, and the pre-computed constant $M' = -M^{-1} \bmod R$.

Output: Montgomery multiplication result:

$Z = \texttt{LLMonPro}(A, B) = A \cdot B \cdot R^{-1} \bmod M$

$T \leftarrow \texttt{OzturkPolMul}(A, B)$ ▷ $T = (T_1, T_0)$
$Q \leftarrow \texttt{OzturkPolMul}(T_0, M')$ ▷ $Q = (Q_1, Q_0)$
$S \leftarrow \texttt{OzturkPolMul}(Q_0, M)$ ▷ $S = (S_1, S_0)$
for $i \leftarrow 0$ **to** $k + 1$ **do**
    $E_i = 0$
    $Z_i = 0$
end for
$E_0 = T_{1_0} + (T_0 \,!= 0)$
for $i \leftarrow 0$ **to** k **do**
    $E_i = E_i + S_{1_i}$ ▷ $0 \leq S_{1_i} < 2^{d+1}$ $i \in [0, k+1]$
    $E_{i+1} = E_{i+1} + T_{1_{i+1}}$ ▷ $0 \leq T_{1_i} < 2^{d+1}$ $i \in [0, k+1]$
end for
for $i \leftarrow 0$ **to** k **do** ▷ $\forall i\ E_i = (E_{i_1}, E_{i_0})$
    $Z_i = Z_i + E_{i_0}$
    $Z_{i+1} = Z_{i+1} + E_{i_1}$
end for
return Z;

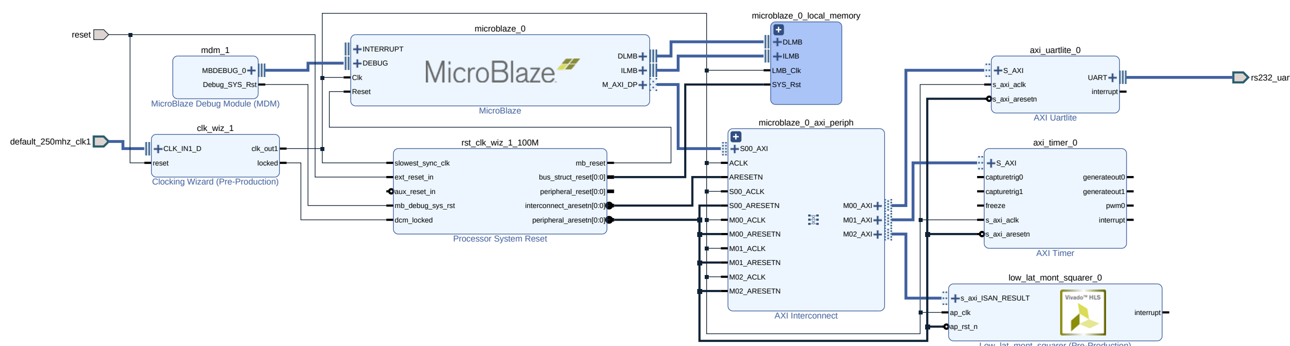

Figure 1: System-on-chip design used in the FPGA verification based on a MicroBlaze Processor attached with our proposed low-latency modular multiplication accelerators (see Table 3 for the parameters verified on real hardware based on this SoC).

Figure 1 illustrates the block diagram of the hardware verification model implemented on VCU118 development board. The accelerators for different parameter values based on the proposed method are implemented on the FPGA using the SoC and verified for the parameters given in Table 3. MicroBlaze is a soft-core processor and can be implemented in FPGA. It is used in this work to test the hardware architecture generated via HLS using our HLS descriptions of the proposed low-latency technique. A program written in C and run in the MicroBlaze processor provides the inputs and the modulus to the low-latency modular multiplication accelerators using slave AXI channel of the processor. The inputs and the outputs of the accelerators have special address values in the AXI memory address space. MicroBlaze writes and reads these memory addresses to send and receive values to the accelerators. Test vectors that are obtained by Sage implementations are used in this verification methodology.

The results obtained from the FPGA implementations running on the VCU118 are compared and verified with Sage [12] (Mathematical package used to implement the given algorithms) programs.

Table 1 summarizes (1) the results of the achieved clock frequency in terms of ns, (2) number of clock cycles to compute the modular multiplication result, (3) (4) LUT, (5) FF and (6) DSP resource utilization for the selected FPGA device (Xilinx VU9P FPGA) for various different parameter values n d, and k. Note that these are the estimated synthesis values measured by Vivado HLS 2019.2 version. As one can see the lowest-possible latency is usually

Algorithm 6 An high-level synthesis implementation of the low-latency redundant-representation polynomial multiplication Algorithm 4 proposed in [11, Algorithm 7] when $d = 16$.

```
void big_mul_wo_lut(ap_uint<(17)> z[2*k+2],
                    ap_uint<(17)> x[k+1],
                    ap_uint<(17)> y[k+1]){

#pragma HLS ARRAY_RESHAPE variable=x complete dim=1
#pragma HLS ARRAY_RESHAPE variable=y complete dim=1
#pragma HLS ARRAY_RESHAPE variable=z complete dim=1

    int i, j, l;

    ap_uint<32> D[2*k+3]; // width 2*d
#pragma HLS ARRAY_RESHAPE variable=D complete dim=1
    ap_uint<17> C[2*k+3]; // width d+1
#pragma HLS ARRAY_RESHAPE variable=C complete dim=1

  for(i=0; i<2*k+3; i++){
    D[i]=0;
  }
  for(i=0; i<2*k+3; i++){
    C[i]=0;
  }
  for(i=0; i<k+1; i++){
    for(j=0; j<k+1; j++){
      int ipj = i + j;
      ap_uint<16> lo[(k+1)*(k+1)];
      ap_uint<16> hi[(k+1)*(k+1)];
      ap_uint<2> redundant[(k+1)*(k+1)];
      mulhilo(x[i],
              y[j],
              &hi[i*(k+1)+j],
              &lo[i*(k+1)+j],
              &redundant[i*(k+1)+j]
             );
      D[ipj] = D[ipj] + lo[i*(k+1)+j];
      D[ipj+1] = D[ipj+1] + hi[i*(k+1)+j];
      D[ipj+2] = D[ipj+2] + redundant[i*(k+1)+j];
    }
  }

  C[2*k+2]=D[2*k+2];

  for(i=0; i<2*k+2; i++){
    C[i]=C[i]+(D[i]&(0xFFFF));
    C[i+1]=C[i+1]+(D[i]>>16);
  }

  for(i=0; i<2*k+2; i++){
    z[i]=C[i];
  }
}
```

achieved when d is selected as 16. This is rational since the FPGA inherently consists of 18x25-bit DSP48 units. However, the LUT utilization is significantly reduced when d is selected as 64. This is because of the decrease in k, which results in the reduction of the required logic to do the addition chain for the accumulations in the algorithm. If the latency is more important than the other constraints, then one should select d as 16 to target an FPGA implementation of the proposed algorithm in this paper.

Table 2 provides the same metrics as given in the Table 1 with different number of clock cycles achieved. This is especially important since a single C description is enough to generate many different hardware architecture by inserting different compiler (HLS) directives to explore the design space further. In this table, each iteration of the loop now has more than one cycle of initiation interval. That means one modular multiplication is now completed in more than one cycle while achieving less amount of clock period. However, none of the variants is better than the latency values of the architectures experimented in the Table 1.

Table 3 presents the placed and routed FPGA implementation results of the proposed Montgomery and Ozturk based low-latency modular multiplication method on VCU118 Development Kit. Note that the method has been verified on real hardware when n is equal to 128, 256 and 512. Each implementation completes the modular multiplication in a single cycle. The running frequency for the accelerators of the proposed technique on FPGA where n is equal to 128, 256 and 512 are 25MHz (40ns), 25MHz (40ns) and 20MHz (50ns), respectively. Modulo-square operation is also performed when $t = 10000$ and the elapsed clock cycles are measured and they are also given for the three different experiments in Table 3. As one can see, approximately 10000 clock cycles with a reading overhead are enough to complete the x^{2^t} modular square operation with 50ns for 512-bit operands. Total clock cycle latency achieved with the proposed technique for $n = 512$ is 46.72ns whereas it is 25.6ns in the Ozturk method. Remark that LUT utilization is reduced to 70k from 330k values given in [11] for $n = 512$.

5 CONCLUSION

In this paper, a new computational description aimed at low-latency for Montgomery modular multiplication is proposed. The method combines Montgomery reduction and polynomial multiplication with redundant-representation proposed by Ozturk in order to perform modular multiplication with a low-latency in hardware. We presented the design space exploration of the proposed approach on a latest FPGA device. Furthermore, the algorithm do not require look-up tables compared to Ozturk method. The proposed algorithm can be customized for different constraints, e.g., the algorithm can be customized to use a certain number of multiplication units via HLS, which however results in increasing the total latency of the modular multiplication.

ACKNOWLEDGMENT

The authors would like to thank Jean-Luc Beuchat for his valuable comments in terms of finding and fixing a bug in the first version of Algorithm 2 and his help to come up with an efficient description of the method described in Algorithm 3. The authors would like to acknowledge Supra National for its generous donation of materials in terms of license of Xilinx design tools and AWS credit used for the necessary hardware synthesis.

Table 1: Design space exploration of the proposed Montgomery and Ozturk based low-latency Modular multiplication (`LLMonPro`) method targeted to XILINX VU9P FPGA device via Vivado HLS 2019.2 to achieve single cycle latency.

n [bits]	d	k	Clock Freq. [ns]	# of clock cycles†	LUT	FF	DSP
64	16	4	17.77	1	12034	1622	75
	32	2	16.85	1	12921	1793	69
128	**16**	**8**	**19.44**	**1**	**31276**	**2992**	**243**
	32	4	20.86	1	26569	3222	300
	64	2	22.72	1	27789	3725	230
256	**16**	**16**	**21.15**	**1**	**77419**	**5763**	**867**
	32	8	22.86	1	58376	5776	972
	64	4	27.81	1	51583	6166	1200
512	**16**	**32**	**22.91**	**1**	**202906**	**10997**	**3267**
	32	16	24.86	1	149247	11251	3468
	64	8	30.36	1	102271	11056	3888
1024	16	64	24.70	1	636603	17155	12675
	32	32	26.85	1	414185	21605	13068
	64	16	32.93	1	253382	17727	13872

† Note that the number of clock cycles of a single Montgomery modular multiplication is always one.

References

[1] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[2] Whitfield Diffie and Martin Hellman. New directions in cryptography. *IEEE transactions on Information Theory*, 22(6):644–654, 1976.

[3] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE transactions on information theory*, 31(4):469–472, 1985.

[4] National Institute of Standards and Technology (NIST). Digital Signature Standard (DSS). `http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf`, July 2013.

[5] Peter L Montgomery. Modular multiplication without trial division. *Mathematics of computation*, 44(170):519–521, 1985.

[6] Thomas Blum and Christof Paar. High-radix montgomery modular exponentiation on reconfigurable hardware. *IEEE transactions on computers*, 50(7):759–764, 2001.

[7] Ismail San and Nuray At. Improving the computational efficiency of modular operations for embedded systems. *Journal of Systems Architecture*, 60(5):440–451, 2014.

[8] Miaoqing Huang, Kris Gaj, Soonhak Kwon, and Tarek El-Ghazawi. An optimized hardware architecture for the montgomery multiplication algorithm. In *International Workshop on Public Key Cryptography*, pages 214–228. Springer, 2008.

[9] Miaoqing Huang, Kris Gaj, and Tarek El-Ghazawi. New hardware architectures for montgomery modular multiplication algorithm. *IEEE Transactions on computers*, 60(7):923–936, 2010.

[10] Daisuke Suzuki. How to maximize the potential of fpga resources for modular exponentiation. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 272–288. Springer, 2007.

[11] Erdinç Öztürk. Modular multiplication algorithm suitable for low-latency circuit implementations. Cryptology ePrint Archive, Report 2019/826, 2019. `https://eprint.iacr.org/2019/826`.

[12] The Sage Developers. *SageMath, the Sage Mathematics Software System (Version 7.5.1)*, 2017. `https://www.sagemath.org`.

Algorithm 7 High-level synthesis implementation for the proposed low-latency Montgomery modular multiplication when $d = 16$.

```
void low_lat_mont_mul(ap_uint<(17)> z[k+1],
                      ap_uint<(17)> x[k+1],
                      ap_uint<(17)> y[k+1],
                      ap_uint<(17)> m[k+1],
                      ap_uint<(17)> mp[k+1]){

#pragma HLS ARRAY_RESHAPE variable=x complete dim=1
#pragma HLS ARRAY_RESHAPE variable=y complete dim=1
#pragma HLS ARRAY_RESHAPE variable=m complete dim=1
#pragma HLS ARRAY_RESHAPE variable=mp complete dim=1
#pragma HLS ARRAY_RESHAPE variable=z complete dim=1

#pragma HLS pipeline II=3
#pragma HLS LATENCY max=3 min=3
#pragma HLS INLINE

   ap_uint<17> t[2*k+2];
#pragma HLS ARRAY_RESHAPE variable=t complete dim=1
   ap_uint<17> q[2*k+2];
#pragma HLS ARRAY_RESHAPE variable=q complete dim=1
   ap_uint<17> tl[k+1];
#pragma HLS ARRAY_RESHAPE variable=tl complete dim=1
   ap_uint<17> ql[k+1];
#pragma HLS ARRAY_RESHAPE variable=ql complete dim=1
   ap_uint<17> res[2*k+2];
#pragma HLS ARRAY_RESHAPE variable=res complete dim=1
    ap_uint<34> E[k+1]; // width 2*d
#pragma HLS ARRAY_RESHAPE variable=E complete dim=1

  big_mul_wo_lut(t,x,y);  // computing t

  ap_uint<1> carry=0;
  for(int i=0;i<k;i++){ tl[i]=t[i]; if(t[i]==0 && carry==0) carry=0; else carry=1;}
  tl[k]=0;

  big_mul_wo_lut(q,tl,mp); // computing q

  for(int i=0;i<k;i++){ ql[i]=q[i];}
  ql[k]=0;

  big_mul_wo_lut(res,ql,m); // computing the result

  for(int i=0; i<k+1; i++){ z[i]=0;}
  for(int i=0; i<k+1; i++){ E[i]=0;}

  E[0] = t[k] + carry;
  for(int i=0; i<k; i++){
    E[i] = E[i] + res[i+k];
    E[i+1] = E[i+1] + t[i+k+1];}
  for(int i=0;i<k;i++){
    z[i] = z[i]+(E[i]&(0xFFFF));
    z[i+1] = z[i+1]+(E[i]>>16);}
}
```

Algorithm 8 Core multiplication function used in the low-latency `LLMonPro` implementation when $d = 16$.

```
void mulhilo(ap_uint<17> x,    // width d+1
             ap_uint<17> y,    // width d+1
             ap_uint<16> *hi, // width d
             ap_uint<16> *lo, // width d
             ap_uint<2> *redundant // width 2-bit
             ){
  // width 2*d+2
  ap_uint<34> res = (ap_uint<34>)x * (ap_uint<34>)y;
  *lo = res;
  *hi = res >> d;
  *redundant = res >> 2*d;
}
```

Table 2: Design space exploration of the proposed Montgomery and Ozturk based low-latency Modular multiplication (`LLMonPro`) method targeted to XILINX VU9P FPGA device via Vivado HLS 2019.2 to achieve various different execution clock cycles.

n [bits]	d	k	Clock Freq. [ns]	# of clock cycles†	LUT	FF	DSP
64	16	4	5.93	16	12429	5002	38
	32	2	11.04	2	12126	2559	42
128	16	8	6.50	16	31726	12218	108
	32	4	8.13	3	24098	4253	200
	64	2	15.10	2	25783	5375	140
256	16	16	8.51	3	69150	8647	434
	16	16	8.52	5	68904	9621	386
	32	8	8.79	3	50062	7559	648
	64	4	10.90	3	46105	11005	800
512	16	32	8.50	8	186605	26588	818
	16	32	9.10	5	168620	20246	1452
	32	16	9.22	5	149262	16750	3468
	64	8	11.25	3	84735	14097	2592
1024	16	64	16.90	3	511958	40303	5634
	32	32	8.61	9	307049	54991	3272
	32	32	10.491	5	302360	38646	5808
	32	32	9.89	8	307031	50156	3272
	32	**32**	**9.19**	**16**	**294266**	**74620**	**1640**
	64	16	12.11	16	171356	59821	1760
	64	16	8.61	18	172305	65471	2080

† It denotes the number of clock cycles of a single Montgomery modular multiplication.

Table 3: FPGA Implementation Results of the proposed Montgomery and Ozturk based Low-Latency Modular Multiplication (`LLMonPro`) on VCU118 Development Kit

				Area				Critical path [clock freq.]					Execution Time	
	n	d	k	LUT	FF	DSP	BRAM	Target Freq. [ns]	Logic [ns]	Route [ns]	Total [ns]	Logic %	# of clock cycles†	$t = 10000$ cycles‡
Our Work	128	16	8	6498	2278	181	1	40	10.99	15.50	26.49	41.49	1	10086
	256	16	16	18741	4121	621	1	40	13.05	19.17	32.22	40.50	1	10103
	512	16	32	69302	7833	2269	1	50	14.65	32.07	46.72	31.36	1	10137
	1024	32	32	146075	68499	1640	2	20	6.49	13.49	19.98	32.50	16	160144

† It denotes the number of clock cycles of a single Montgomery modular multiplication.
‡ It denotes the total number of clock cycles of the modular squarer when $t = 10000$.