Modular Inverse for Integers using Fast Constant Time GCD Algorithm and its Applications

Sanjay Deshpande*, Santos Merino del Pozo†‡, Victor Mateu†, Marc Manzano†§, Najwa Aaraj†‡‡ and Jakub Szefer*
*Computer Architecture and Security Laboratory, Department of Electrical Engineering, Yale University, New Haven, CT, USA
Email: firstname.lastname@yale.edu
†Cryptography Research Centre, Technology Innovation Institute, Abu Dhabi, UAE
Email: firstname.lastname@tii.ae
‡Digital Security Group, Faculty of Science, Radboud University, Nijmegen, Netherlands
§Electronics and Computing Department, Faculty of Engineering, Mondragon Unibertsitatea, Mondragon, Spain
‡‡ Okinawa Institute of Science and Technology, Okinawa, Japan

***Abstract*—Modular inversion, the multiplicative inverse of an integer in the ring of integers modulo a prime number, is widely used in public-key cryptography. However, it is one of the most computationally intensive operations, thus, it remains the main performance bottleneck for many cryptographic algorithms.**

This paper presents to the best of the author's knowledge, the first FPGA-based hardware design for computing the multiplicative inverse using the recently proposed fast constant-time Greatest Common Divisor (GCD) algorithm. This paper introduces two distinct design architectures targeting different applications: (a) a full-width design and (b) a sequential design. The presented designs are compact, parameterizable, and scalable in terms of area and speed. The evaluation shows the proposed designs, which are constant-time and protect against timing-based attacks, outperform existing software and hardware implementations that use other modular inversion techniques. As a specific example, this work presents an evaluation focusing on the use of the multiplicative inverse hardware module to accelerate the ElGamal cryptosystem. The proposed design achieves a speed-up of 90% in the modular inverse calculation and a speed-up of 45% in the overall ElGamal decryption algorithm using our sequential hardware design of fast constant-time GCD algorithm.

In addition to developing the fast hardware implementation, this work potentially opens up a new direction for designing cryptosystems: the inverse operation is often avoided when designing algorithms, due to its complexity. With the new hardware module, using the inverse becomes more tractable, making it more appealing to use in the design of new cryptosystems.

I. Introduction

Public key cryptography schemes often use modular inversions, known to be computationally intensive operations. For example, in RSA [24], the secret key is obtained through the inversion of the public key. In Digital Signature Algorithm (DSA) [19], to generate a digital signature, the per-message random secret needs to be inverted. Meanwhile, in ElGamal encryption system [5], to decrypt the message, part of the ciphertext is inverted.

Apart from cryptographic applications, modular inversion also has its application in other fields. For example, it is used in communication and digital signal processing for residue number system applications [13], in image processing applications for image manipulation and production [30], and in musical scales and multiplicative groups to manipulate music written in pentatonic scale [29]. This paper mainly focuses on the cryptographic applications of modular inversion.

The two most well-known methods to compute the modular multiplicative inverse are the Fermat's method [27], and the Euclid's method [26] (also known as binary extended Euclidean algorithm (BEEA), or greatest common divisor (GCD) method). Out of these two, the most efficient approach to perform modular inversion is the BEEA which is derived from Euclid's method [26]. This approach is efficient because it replaces multi-precision divisions by simple right shifts, which makes this approach quite suitable for both hardware and software implementations – yet it is prone to side-channel attacks due to its non-constant time behavior. Therefore, in order to thwart timing-based attacks, cryptography researchers and practitioners have typically used Fermat's method based on modular exponentiation. Fermat's method can be implemented in constant time and a few of the efficient techniques to do so are described in [17].

Nevertheless, in 2019, Bernstein and Yang showed that similar (or even better) performance results can be achieved using the BEEA, while still being constant-time [3]. The BEEA variants in [3], which from now on will be referred to as "fast GCD algorithm" or "fast GCD inverse", are based on so called fast constant-time "division steps", whose efficiency was demonstrated in two software case studies using Intel desktop CPUs. However, the implementation and evaluation on CPUs with smaller multipliers (e.g., certain ARM CPUs), on Field Programmable Gate Arrays (FPGAs), and on Application-Specific Integrated Circuits (ASICs) was left as an open research problem.

Contributions: In this work we present, to the best of our knowledge, the first compact and scalable hardware designs for the modular multiplicative inverse for integers using the fast GCD algorithm. We introduce two architectures for the modular inverse unit which are further parameterizable allowing them to be adapted for various applications. The two architec-

This work was supported by a grant from the Technology Innovation Institute (TII).

Algorithm 1: `fast_gcd` module to compute $g^{-1} \pmod f$, assuming f is odd [3]

```
def iterations(d):
    if d < 46:
        rounds = (49*d+80)//17  #// does floor division
    else:
        rounds = (49*d+57)//17
    return rounds

def divsteps2(n,delta,f,g):
    v,r = 0,1
    m = n
    for n in range(m):
        mask1 = (delta > 0) and (g&1 ==1)
        mask0 = (delta <= 0) or (g&1 ==0)
        delta,f,g,v,r = (mask0*delta + mask1*(-delta)),
           (mask0*f + mask1*g), (mask0*g + mask1*(-f)),
           (mask0*v + mask1*r), (mask0*r + mask1*(-v))
        g0 = g&1
        delta,g,r = 1+delta,(g+g0*f)/2,(r+g0*v)/2
        g = ZZ(g) #zz() means Integer Ring
    v_out = sign(f)*ZZ(v*2^(m-1))
    return v_out

def fast_gcd(f,g):
    d = max(f.nbits(),g.nbits())
    m = iterations(d)
    precomp = Integers(f)((f+1)/2)^(m-1)
    v_out = divsteps2(m,1,f,g)
    inverse = ZZ(v_out*precomp)
    return inverse
```

tures we present are full-width design and sequential design. We then evaluate the designs on the ElGamal cryptosystem, where we present two lightweight hardware designs for ElGamal decryption, one using Fermat's method and other using fast GCD algorithm to compute modular inverse and demonstrate that the presented designs achieve a speed-up of 90% in the modular inverse calculation and a speed-up of 45% in ElGamal decryption algorithm, when using our sequential design of Fast GCD algorithm. Further, we profile other applications to estimate how much performance improvement our modular inverse hardware module would add to them. Additionally, we make all the code for the modular inverse and ElGamal decryption algorithm available under an open-source license at https://caslab.csl.yale.edu/code/fast-constant-time-gcd.

II. Related Work

In 2019, Bernstein and Yang proposed the fast constant-time GCD algorithm [3], providing the community with a new tool to compute modular inverse in an efficient manner, while remaining secure against timing-based attacks. In their work, the authors provided two variants of the fast GCD algorithm: a variant for the modular inversion of integers in the multiplicative group defined by the ring of integers modulo prime, and a variant for modular inversion of polynomials. the last one has recently been implemented in hardware [14] but, to the best of our knowledge, this work presents the first hardware implementation of the fast GCD algorithm for modular inversion of integers.

Other recent work on modular inversion includes work from Azarderakhsh et al. [18] which provides an area-time efficient architecture for ECC key exchange using Curve25519 targeting Xilinx `XC7Z020`. In their work, an implementation of constant-time modular inverse for 255-bit prime using Fermat's method was designed which takes approximately $89,911$ clock cycles for each modular inverse calculation, running at a frequency of 200 MHz. In [15], the authors provide multiple efficient hardware implementations for Supersingular Isogeny Key Encapsulation (SIKE) [11], a NIST Post Quantum Cryptography (PQC) competition candidate, targeting Xilinx Virtex-7 690T. Two implementations of constant-time modular inverse for 434-bit prime using Fermat's method are presented, first one is a 256-bit architecture which takes $47,098$ clock cycles for each modular inverse operation, running at 142 MHz, and second one is a 128-bit architecture which takes $111,646$ clock cycles for each modular inverse operation running at a frequency of 154 MHz. In [6], the author provides details on hardware design of a constant-time modular exponentiation unit for a field width of 2048-bits. This modular exponentiation can be used to compute modular inverse using Fermat's method, and the amount of clock cycles taken per each modular inverse is $113,880,000$ while running at 99.7 MHz on an Altera's Stratix FPGA. In [33], the authors provide a high-performance non constant-time modular inverse unit using GCD method, which for a prime of 2048 bits, in its best-case scenario, takes $98,198$ clock cycles for each inverse calculation running at 250 MHz on Xilinx `XC7VX485T-2`.

Regarding software implementations, in [3] the authors provide results for a software implementation of a modular inverse unit for a 255-bit prime using the constant-time fast GCD algorithm and they achieve each modular inverse computation in $35,277$ clock cycles on an ARM Cortex-A7 processor, $10,050$ clock cycles on Haswell, $8,778$ clock cycles on Skylake, and $8,543$ clock cycles on Kaby Lake. In [7] the authors provide a modular inverse targeting a specific prime of 255 bits using Fermat's method and they achieve each modular inverse computation in clock cycles ranging between $41,978$ to $151,997$, running at a frequency varying between 48 MHz and 2000 MHz depending on the target processor.

III. Hardware Design

The hardware designs for the constant-time fast GCD operations presented in this paper are based on the fast constant-time GCD algorithm by Bernstein and Yang [3]. The algorithm involves a different set of arithmetic operations, namely multiplications, additions, and divisions, as shown in Algorithm 1. Out of all the arithmetic operations, the most expensive operation is the division. The divisions and multiplications are done by means of shift operations due to the fact that they are conducted with powers of 2.

Our hardware constant-time fast GCD designs implement all the operations from the `fast_gcd` function in Algorithm 1. The sizes of inputs `f` and `g` are fixed once the size of the primes is selected (at compile time, based on the algorithm that is using the fast GCD module). Therefore, the values of `d`, `m`, `precomp` in `fast_gcd` function can be pre-computed and stored as constants in registers or memory. The final modular multiplication operation (`v_out*precomp`) is not computed inside our hardware fast GCD modules since it is assumed that all the cryptographic applications where the modular inverse is used already have an existing modular multiplication unit that can be used to perform this final modular multiplication step.

In this work, we present two fast GCD designs with two different time-area trade-offs: the first design provides a fast version that uses more resources while, the second one, provides a slightly slower version, but more area-efficient and with a resulting better time-area product.

A. *Fast GCD Algorithm - Full-Width Design*

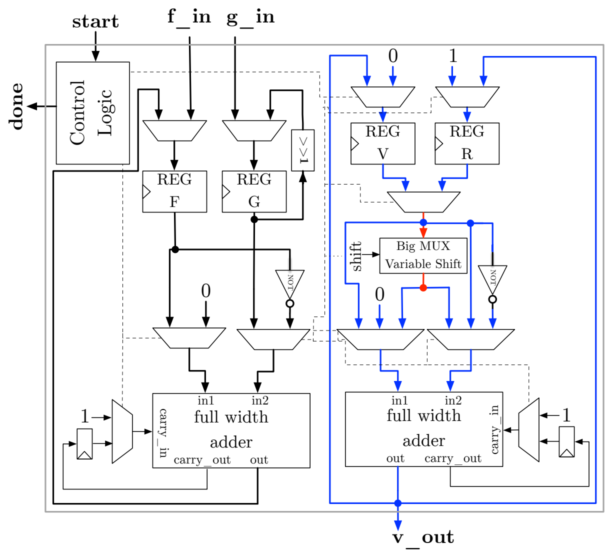

Fig. 1: Hardware design of fast GCD inverse - full-width design

Figure 1 shows the hardware design and dataflow of our first fast GCD algorithm implementation with a full-width design. As shown in Figure 1, different buses are segregated based on the colored black, blue, and red lines. Table I shows the bus widths for respective input width for all the solid lines from Figure 1. The dotted lines from Figure 1 represent the control signals. The design includes two adders and one multiplexer-based variable shifter to handle the multiplications and divisions. By design, we assume that the multiplications and divisions are by powers of 2. These three modules (two adders and one variable shifter) are the major contributors to the total area of the presented design. The control logic module consists of all the control signals and counters. The design follows a full-width datapath due to which the area is bigger. Keeping in mind the large data-width, the adder module is designed to register the carry and perform addition in multiple clock cycles, thus, reducing the critical path due to the carry propagation. The number of clock cycles taken by each addition is parameterized and can be decided upon, so that the design can be clocked at a higher frequency.

From the `divsteps2` (Algorithm 1), $g = (g + g0 * f)/2$ is implemented by rewiring the bits to achieve the right shift. Whereas implementing $r = (r + g0 * v)/2$ is not straight

Input	**Bus Widths**		
Width	**Black**	**Blue**	**Red**
255	256	739	1024
434	435	1255	2048
1279	1280	3690	4096
2048	2049	5907	8192

Table I: Bus widths of fast GCD inverse - full-width design

forward since r, v values remain as fractions (< 1) most of the time. Therefore, in order to handle this, a traditional division handling logic (which includes both multiplication and division) has been designed with a variable shift. The variable shift is designed using a barrel shift [22] and masking which can perform both right shift (for division by 2) and left shift (for multiplication by 2). The variable shifter's width depends on the width of the inputs to the modular inverse function and is calculated by the `iterations` function from Algorithm 1. For example, for an input width of 256 bits, used in Elliptic Curve Cryptography (ECC) [19], the width of the variable shifter is 741 bits. For an input width of 1279 bits, used in ElGamal cryptosystem [5], the width of the variable shifter is 3689 bits. Finally, for an input width of 2048 bits, used in RSA [24], the width of the variable shifter is 5906 bits. The barrel shift inside the variable shifter is constructed by a cascade of multiplexers and, as described in [22], an N-bit width shifter implemented as a $log_2 N$ cascade of multiplexers. This makes the variable shifter considerably large, and it becomes a major contributor to the overall area of the design.

The time and area results for the full-width design targeting the Zynq UltraScale+ `XCZU7EG` FPGA are presented in Table II. The time shown in Table II refers to the time taken by the hardware design to calculate one modular inverse operation. It can be seen that as the input width increases, the area increases considerably, which makes this design suitable for cases where a modular inverse of a comparatively smaller width (bit width) is used (for example, in ECC operations). In the case of input widths of 1279 and 2048 bits, the design becomes very large. Despite the large number of resources on the `XCZU7EG`, the full-width design for input widths 1279-bits and 2048-bits does not fit on it. The underlying cause is not the size of the whole module, but rather the fact that the designs of input widths 1279-bits and 2048-bits need variable shifters of width 4096-bits and 8192-bits, respectively. Additionally, each of the designs needs two big registers of size 3690-bit and 5907-bit. Due to placement and routing constraints, the synthesis tool could not route the design with such large registers. The '-' in Table II indicates that the configuration could not fit in the target FPGA. These large width inputs are handled easily by our next design, the sequential design presented next.

B. *Fast GCD Algorithm - Sequential Design*

The second hardware design of the fast GCD algorithm follows a Random Access Memory (RAM) based pipelined iterative architecture. The number of iterations is decided based on the width of the input, variable `m` in Algorithm 1,

Input Width	**Freq** (MHz)	**Time** (us)	**Area**			**Time x Area**
			Slices	FFs	% usage	$\times 10^3$
255	207	40.9	1847	6704	6.41%	76
434	202	93.1	3495	8856	12.14%	326
1279	-	-	-	-	-	-
2048	-	-	-	-	-	-

Table II: Time and area results for hardware implementation of fast GCD inverse - full-width design

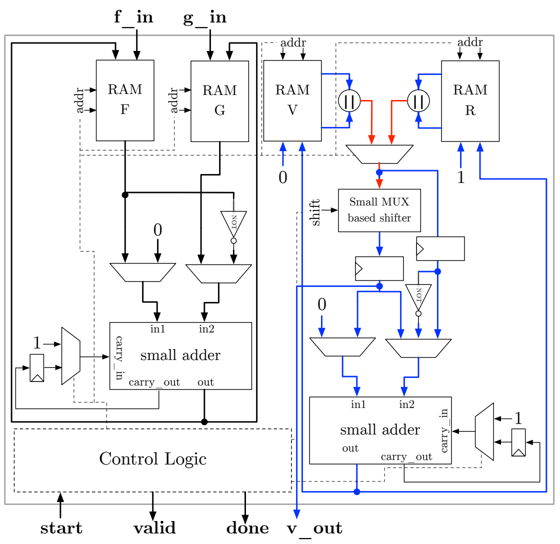

Fig. 2: Hardware design of fast GCD inverse - sequential design

which is public and can be precomputed. The design consists of four block RAM units, as shown in Figure 2, used to store the values of f, g, r, and v (from `divsteps2` function in Algorithm 1). As shown in Figure 2, different buses are segregated based on the colors black, blue, and red lines. Table III shows the bus widths for respective input width for all the solid lines from Figure 2. The dotted lines from Figure 2 represent the control signals. The drawback of the large variable shifter described in Section III-A is tackled by using a Sequential Variable Shifter (SVS).

The SVS is derived from the shifter described in Xilinx's document [8]. The variable shifter described in the document supports only left shifting the data when divided into four chunks. We extended the existing shifter's capability to support both left and right variable shifts on a user-defined number of data chunks. The number of data chunks and the width of each chunk is fixed before the design is synthesized. This variable shifter design is scalable from small (four) to large number (hundreds) of data chunks. The shift is performed sequentially by dividing the data (e.g. 1024-bits) in to smaller chunks (e.g. 256-bits). Inside the SVS, the input integer of size t-bits is loaded into a block RAM sequentially in smaller chunks of x-bits per clock cycle. Once the loading is completed, two smaller chunks of data are loaded out of the RAM in each clock cycle, and a partial shift is performed using the small shifter (of width $2x$) until the whole data is shifted.

Input Width	**Bus Widths**		
	Black	**Blue**	**Red**
255	64	128	256
434	73	128	256
1279	80	128	256
2048	86	128	256

(a) 256-bit architecture

Input Width	**Bus Widths**		
	Black	**Blue**	**Red**
255	32	64	128
434	44	64	128
1279	43	64	128
2048	43	64	128

(b) 128-bit architecture

Input Width	**Bus Widths**		
	Black	**Blue**	**Red**
255	22	32	64
434	22	32	64
1279	24	32	64
2048	22	32	64

(c) 64-bit architecture

Input Width	**Bus Widths**		
	Black	**Blue**	**Red**
255	11	16	32
434	11	16	32
1279	11	16	32
2048	11	16	32

(d) 32-bit architecture

Input Width	**Bus Widths**		
	Black	**Blue**	**Red**
255	6	8	16
434	6	8	16
1279	6	8	16
2048	6	8	16

(e) 16-bit architecture

Table III: Bus widths of fast GCD inverse - sequential design

Input Width	**Period** (ns)	**Time** (T) (us)	**Area**				**T×A** .10^3
			Slices (A)	FFs	BRAM	% usage	
256-bit architecture							
255	5.25	47	915	803	22.5	3.18%	43
434	5.30	106	920	851	22.5	3.19%	98
1279	5.34	710	922	625	22.5	3.20%	654
2048	5.47	1,680	933	932	22.5	3.24%	1567
128-bit architecture							
255	4.00	59	442	722	4	1.53%	26
434	4.00	121	482	638	4	1.67%	58
1279	4.20	992	531	650	4	1.84%	527
2048	4.30	2,540	554	564	12	1.92%	1407
64-bit architecture							
255	3.30	68	234	434	2	0.81%	16
434	3.40	188	256	453	2	0.89%	48
1279	3.40	1,506	282	412	2	0.98%	425
2048	3.80	4,399	291	390	6	1.01%	1280
32-bit architecture							
255	3.00	115	163	307	1	0.57%	19
434	3.00	316	166	318	1	0.58%	53
1279	3.00	2,613	235	320	1	0.82%	614
2048	3.20	7,335	220	370	1.5	0.76%	1614
16-bit architecture							
255	2.89	213	115	241	0.5	0.40%	25
434	2.95	608	120	265	0.5	0.42%	73
1279	2.98	5,148	128	254	1.5	0.44%	659
2048	2.98	13,592	136	275	1.5	0.47%	1848

Table IV: Time and area results for hardware implementation of fast GCD inverse - sequential design

The time and area results for the sequential implementation targeting the Zynq UltraScale+ `XCZU7EG` FPGA are presented in Table IV. In the sequential design, the datapath can be configured for different widths (256, 128, and so on) based on the application. The datapath width is dependent on the width of the shifter, if the width of the shifter is $2x$ then the width of the datapath is x.

From Table II and Table IV it can be seen that the time-area product for our sequential design is better than our full-width design. Therefore, we use our sequential design in the comparisons conducted in the rest of the paper.

Algorithm 2: ElGamal decryption algorithm

```
Input: Ciphertext 1 (c_1), Ciphertext 2 (c_2),
       SecretKey (x)
Output: Message (m)
Step 1: Compute s:=c_1^x
Step 2: Compute l:=s^(-1)
Step 3: Compute m:=c_2.l
```

Operations	Our Method - D_1	Our Method - D_2
$s := c_1^x$	Modular exponentiation	Modular exponentiation
$l := s^{-1}$	Modular exponentiation	Fast GCD algorithm
$m := c_2.l$	Modular multiplication	Modular multiplication

Table V: Operations used in ElGamal Decryption

IV. Applications

Cryptosystems based on the Discrete Logarithm Problem (DLP) usually require to compute the multiplicative inverse either in the encryption or in the decryption algorithm. In this paper, we first focus on ElGamal decryption, and then we analyze other possible applications. In all studied cases, the modular inverse is accelerated from 58% to 90%.

A. ElGamal Decryption

ElGamal cryptosystem [5] is a public-key encryption scheme defined over any finite cyclic group. Its security is based on DLP. The cryptosystem is defined by three algorithms: key generation, encryption, and decryption. The details of each can be found in [5]. We focus on the decryption algorithm as it is the only one using modular inverse, which we can accelerate with our hardware modules.

To the best of our knowledge, there are no existing efficient and suitable hardware implementations which are constant-time for ElGamal decryption. Thus, one of the contributions of this work is the hardware implementation of a lightweight constant-time ElGamal decryption unit itself.

Existing hardware designs from [32] are not publicly available. The only information about usage of modular inversion provided in [32] is that the authors use the modular inversion algorithm specified in [31] and the specified algorithm is a variation of BEEA which is not constant-time. Also, the authors do not specifically provide the timing details for the modular inverse calculation. To demonstrate and evaluate the impact of the inverse operation in the ElGamal decryption algorithm (shown in Algorithm 2), we implement our own hardware design of the ElGamal decryption unit. The minimum recommended key-size for ElGamal cryptosystem is 1024-bits [1]. We want to use Mersenne primes as they allow the computation of modular reductions using only additions as shown in [10] (Algorithm 2.31). This way, the circuit is smaller and more efficient. We use the Mersenne prime $2^{1279} - 1$ to define the integer ring in ElGamal because it is the smallest Mersenne prime bigger than 2^{1024}.

From Algorithm 2, we dissect the operations as shown in Table V. We design two separate constant-time decryption units, which we will refer to as D_1 and D_2 in the rest of the paper. In D_1, we use the Fermat's method [20] for the modular inverse calculation (which we consider to be the existing standard method for modular inverse calculation) and this serves as the reference design for us. In D_2 we use our modular inverse design described in Section III-B.

For both D_1 and D_2 we design a RAM-based ElGamal decryption processor. Our goal is to provide a lightweight architecture, so we design a datapath with a width of 16-bits. The individual components involved in the design of D_1 are Modular Multiplication (MM), and Modular Exponentiation (ME), and the individual components involved in the design of D_2 are MM, ME, and the fast GCD inverse (FGCD). One of the important components of both designs is the MM unit. We design our MM unit with a combination of Schoolbook Multiplication and Specific Reduction unit for the given prime ($2^{1279} - 1$, which is used in ElGamal). For ME, we use a constant-time left to right, square and multiply always algorithm described in [17]. For the D_1 design, we combine the MM and ME logic with some additional control logic, and we design a top-level module to calculate the s, l, and m values mentioned in Algorithm 2. The simplified hardware design for D_1 is shown in Figure 3. For the D_2 design, we use the same logic from D_1 to compute s and m values, however, for computing l we use the fast GCD inverse with some additional control logic. The simplified hardware design for D_2 is shown in Figure 4. All the bold lines in Figure 3 and Figure 4 represent 16-bit data buses. We design parameterizable modules where we provide a parameter to instantiate the Digital Signal Processing (DSP) functions or use the Look up Tables (LUTs) and register based logic for the two sub-modules `MUL & ACC` and `ADD`. Further details on extensions, results, and applications of our ElGamal decryption design are described in Section V.

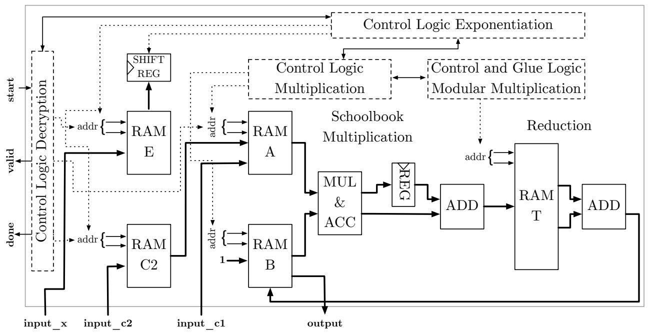

Fig. 3: Hardware design of ElGamal decryption module - D_1

B. Other Applications

We perform time profiling on other applications, namely DSA, ECC, SIKE [11], and RSA. There is a substantial performance improvement for the inverse operation. However, the total performance improvement for each algorithm is not significant since the usage of the inverse in this latter is limited as it is an expensive operation. Our new hardware modules significantly improve the computational time of the inverse operation (from 58% to 90%) and could thus allow for

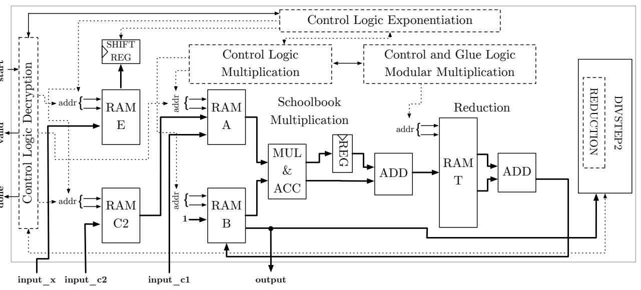

Fig. 4: Hardware design of ElGamal decryption module - D_2

the development of new or modified algorithms that use the inverse more widely.

The first additional application we analyze is DSA, where inversion is used in both signing and signature verification algorithms [25]. For a 2048-bit DSA, we consider the timing results provided for the modular multiplication unit in [9] and calculate the approximate timing for overall DSA signing and signature verification algorithms. To get the initial estimates, we consider a default design where the modular inverse is calculated using the Fermat's method [27]. Then, we consider timing by replacing the time taken for the Fermat's method modular inversion unit with the time taken by our modular inverse unit (described in Section III). The overall improvement in DSA signing and signature verification is 49% and 33% respectively, whereas improvement on the modular inverse is more than 90%.

The second additional application we analyze is ECC using Curve25519, where we conduct profiling on the lightweight design described in [18]. We notice that by replacing the existing inverse functionality with our inverse function (with a 16-bit architecture), the overall time improvement on each scalar multiplication is 5%, but note that there is a substantial improvement of 58% for the modular inverse calculation.

The third additional application we analyze is SIKE [11], a candidate from the on-going NIST PQC standardization process. We select a hardware implementation of SIKE given in [15] for deriving the timing profile. The authors provide two variations of the design: 128-bit and 256-bit. We profile both designs separately and notice that by replacing the existing inverse functionality with our inverse function: in the 128-bit design, the overall improvement in key generation and decapsulation is 4% and 3% respectively, with a substantial improvement of the modular inverse operation of around 82%. In the 256-bit design, the overall improvement in key generation and decapsulation is 3% and 2% respectively, with a substantial improvement of modular inverse operation of around 58%.

Finally, we also analyze the 2048-bit RSA case: we conduct time profiling on the key generation module provided in [12] and notice that by replacing the existing inverse functionality with our inverse function, the overall improvement in the key generation process is close to 3%, and the improvement for the modular inverse module is more than 90%.

V. Results and Analysis

Our hardware for the inverse operations and ElGamal decryption is implemented in VHDL. To test the Fast GCD Inverse implementation, we use the following methodology: the fast GCD inverse functionality was provided in a Sage script by its authors in [3]. The provided Sage implementation was not constant-time so, we re-wrote the Sage code to make it constant-time, as shown in Algorithm 1, and our Sage implementation mimics the procedures used by the hardware units. This way, we can generate suitable test vectors for the hardware design. This approach helps us debug the code during the development phase. After the behavioral simulation of the VHDL code, the code is synthesized and implemented. For testing the ElGamal decryption implementation, we write our own sage reference implementation based on Algorithm 2 and follow similar methodology as testing the fast GCD inverse module.

We obtain the post-place and route results for our hardware designs with Xilinx Vivado 2019.2 for Zynq UltraScale+ `XCZU7EG`. The results for our fast GCD inverse modules are shown in the Table II and Table IV. The results for our ElGamal decryption modules are shown in Table VII.

A. *Results: Fast GCD Algorithm*

Table VI shows timing results for our sequential design of the fast GCD algorithm (presented in Section III-B) with different widths and compares it with efficiently implemented modular inverse hardware designs from the literature. For brevity and a fair comparison, we only tabulate the results of the hardware designs. From the cited literature, only [33] uses Euclid's method to compute the modular inverse, while all the other works use Fermat's theorem. As it can be seen in Table II and Table IV, our design outperforms the other designs mentioned in Section II in terms of latency. There is only one exception where our design does not perform better and the reason for that is that in [33] the authors implemented a non-constant time modular inversion which leaks information about the inverted value and, therefore, is not suitable for cryptographic applications. In Table VI, we compare our 2048-bit input width design, which was implemented on `XCZU7EG` with the only existing 2048-bit design from literature [6], which was implemented on Stratix FPGA, and showcase an improvement of $339\times$. We compare results of FPGAs belonging to different process nodes, since: a) we do not have access to Stratix tools to generate the results for the design that targeted Stratix and b) we do not have access to the source code of the design from [6]. One of the observations that can be made from Table VI is that, when the field width becomes larger then the improvement caused by our inverse functionality increases significantly.

B. *Results: ElGamal Decryption*

The provided ElGamal decryption implementation is targeted for a specific prime: $2^{1279} - 1$. We achieve an improvement of 90% in the time taken to compute the modular inverse in D_2 when compared to D_1. Note that D_2 utilizes

Design	Device	Frequency (MHz)	Clock Cycles $\times 10^3$	Time (ms)	Speed-up
Input Width = 255 bits					
Our Work	XCZU7EG	190	9	0.05	0.5
Our Work	XC7Z020	100	9	0.09	1.0
[18]	XC7Z020	200	90	0.45	5.0
Input Width = 434 bits					
Our Work	XCZU7EG	188	20	0.11	0.64
Our Work	Virtex 7	112	20	0.17	1.0
[15]	Virtex 7	154	111	0.72	4.2
[15]	Virtex 7	141	47	0.33	1.9
Input Width = 2048 bits					
Our Work	XCZU7EG	182	307	1.68	1.0
[6]	Stratix	200	113,880	569.4	339.0

Table VI: Fast GCD inverse (sequential design) timing result comparison

our sequential design of the fast GCD algorithm (presented in Section III-B). Based on our lightweight architecture the time taken per each ciphertext decryption in D_1 is 0.11 seconds which is improved in our optimized design D_2 by 45%, taking 0.064 seconds for each decryption. Currently, designs D_1 and D_2 support two primes, namely $2^{127} - 1$ and $2^{1279} - 1$, and can be easily extended to support any given Mersenne prime [4] with some minor modifications in the modular reduction logic. Further, the designs can be made completely generic by implementing a generic modular reduction logic such as Barrett reduction [2].

In Table VII, we compare our implementation results with the most relevant ones in the literature. In [28], the authors provide a software implementation of ElGamal decryption unit, implemented using two techniques which they refer to as general and linear and they both take 3.2 and 2.8 seconds respectively for each decryption. The results presented in Table VII show that our designs D_1 and D_2 are both more efficient than the provided relevant implementations in literature. Both D_1 and D_2 are constant-time implementations, and consume less than 1% of the target FPGA resources, each. Hence, a user can focus on D_2 to achieve a better performance while still having good resource utilization. We developed both designs in order to allow a user the flexibility to choose between D_1 and D_2 depending on the requirements. It is worth remarking the fact that the field-width of the existing implementations from [28] and [32] is 512-bits whereas our field-width is 1279-bits. Additionally, all of our implementations are constant-time whereas no information about being constant-time is provided for [28] and [32].

VI. Conclusion and Future Work

In this paper, we offer two distinct constant-time hardware designs for the recently proposed multiplicative modular inverse algorithm, providing a different speed-area trade-off for different input sizes. Our proposed hardware designs provide a speed-up up to $339\times$ for the modular inverse calculation when compared to existing methods.

We also provide two novel lightweight hardware implementations for ElGamal decryption unit: one with a modular inverse using Fermat's method (D_1) and another one with our fast GCD inverse (D_2) implementation. We show that we gain a speedup of 45% for the overall decryption operation in our design D_2 when compared to D_1.

The compact fast GCD modular inverse module presented in this work can be embedded as a hardware accelerator in a RISC V processor such as [23] and modular inverse functionality can be added to its instruction set architecture. This would potentially provide acceleration benefits for all cryptographic applications developed on such processors, where modular inverse functionality is used.

The ElGamal decryption implementation can be extended to a complete ElGamal cryptosystem and ElGamal signature scheme. A faster implementation of ElGamal cryptosystem could potentially be used with other schemes in a hybrid cryptographic protocol or in other cryptographic applications such as within an electronic voting system as described in [16] and for homomorphic encryption operations as described in [21].

References

[1] Bryce D. Allen. Implementing several attacks on plain ElGamal encryption. Master's thesis, Iowa State University, 2008.

[2] Paul Barrett. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In *Advances in Cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986, Proceedings*, volume 263 of *Lecture Notes in Computer Science*, pages 311–323. Springer, 1986.

[3] Daniel J. Bernstein and Bo-Yin Yang. Fast constant-time GCD computation and modular inversion. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(3):340–398, May 2019.

[4] Chris K. Caldwell. Mersenne primes: History, theorems and lists. https://primes.utm.edu/mersenne/, 2020.

[5] T. Elgamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985.

[6] John Fry. RSA & public key cryptography in FPGAs. Technical report, Altera Corp., 2005.

[7] Hayato Fujii and Diego F. Aranha. Curve25519 for the Cortex-M4 and beyond. In Tanja Lange and Orr Dunkelman, editors, *Progress in Cryptology – LATINCRYPT 2017*, pages 109–127, Cham, 2019. Springer International Publishing.

[8] Paul Gigliotti. Implementing barrel shifters using multipliers. Technical Report XAPP195, Xilinx, 2004.

[9] B. Hanindhito, N. Ahmadi, H. Hogantara, A. I. Arrahmah, and T. Adiono. FPGA implementation of modified serial montgomery modular multiplication for 2048-bit RSA cryptosystems. In *2015 International Seminar on Intelligent Technology and Its Applications (ISITIA)*, pages 113–118, 2015.

[10] Darrel Hankerson, Alfred J. Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography*. Springer-Verlag, Berlin, Heidelberg, 2003.

[11] David Jao, Reza Azarderakhsh, Matthew Campagna, Craig Costello, Luca De Feo, Basil Hess, Aaron Hutchin-

Design	Device	Input Width	Logic	Frequency (MHz)	Time (s)	T×A
Our work-D_1	XCZU7EG	1279	Slices - 84 DSP - 3 % FPGA usage - 0.29%	300.00	0.11	9
Our work-D_2			Slices - 229 DSP - 5 % FPGA usage - 0.79%	300.00	0.06	14
V2 [32]	Virtex-E	512	Slices - 83,621	28.39	35.21	$2{,}944\times10^3$
V4 [32]			Slices - 83,442	27.58	36.24	$3{,}023\times10^3$

Table VII: ElGamal decryption time and area utilization comparison

son, Amir Jalali, Koray Karabina, Brian Koziel, Brian LaMacchia, Patrick Longa, Michael Naehrig, Geovandro Pereira, Joost Renes, Vladimir Soukharev, and David Urbanik. Supersingular isogeny key encapsulation. Submission to the NIST Post-Quantum Standardization Project, 2019.

[12] Matthew Kenney. Implementing an RSA key generation and encryption/decryption functionalities on the Basys-3 FPGA board. https://github.com/kenneym/cs56/tree/master/design-source, 2019.

[13] Pallab Maji. Application of residue arithmetic in communication and signal processing. Master's thesis, National Institute of Technology, 2011.

[14] Adrian Marotzke. A constant time full hardware implementation of streamlined NTRU prime. In *Smart Card Research and Advanced Applications - 19th International Conference, CARDIS 2020, Virtual Event, November 18-19, 2020, Revised Selected Papers*, volume 12609 of *Lecture Notes in Computer Science*, pages 3–17. Springer, 2020.

[15] Pedro Maat C. Massolino, Patrick Longa, Joost Renes, and Lejla Batina. A compact and scalable hardware/software co-design of SIKE. Cryptology ePrint Archive, Report 2020/040, 2020.

[16] M. Mikhail, Y. Abouelseoud, and G. Elkobrosy. Extension and application of ElGamal encryption scheme. In *2014 World Congress on Computer Applications and Information Systems (WCCAIS)*, pages 1–6, 2014.

[17] Christophe Negre and Thomas Plantard. Efficient Regular Modular Exponentiation Using Multiplicative Half-Size Splitting. *Journal of Cryptographic Engineering*, 7(3):245–253, 2017.

[18] M. B. Niasar, R. El Khatib, R. Azarderakhsh, and M. Mozaffari-Kermani. Fast, small, and area-time efficient architectures for key-exchange on curve25519. In *2020 IEEE 27th Symposium on Computer Arithmetic (ARITH)*, pages 72–79, 2020.

[19] NIST. The digital signature standard. *Commun. ACM*, 35(7):36–40, July 1992.

[20] O. Ore. *Number Theory and Its History*. Dover Books on Mathematics Series. Dover, 1988.

[21] Metehan Ozten. ElGamal cryptosystem with an application. http://koclab.cs.ucsb.edu/teaching/cren/project/2018/Ozten.pdf, 2018.

[22] Matthew R. Pillmeier, Michael J. Schulte, and Eugene George Walters III. Design alternatives for barrel shifters. In Franklin T. Luk, editor, *Advanced Signal Processing Algorithms, Architectures, and Implementations XII*, volume 4791, pages 436 – 447. International Society for Optics and Photonics, SPIE, 2002.

[23] PQSheild. The Pluto RISC-V core. https://pqsoc.com/#the-pluto-risc-v-core, 2021.

[24] R.L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21:120–126, 1978.

[25] Bruce Schneier and Phil Sutherland. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley & amp; Sons, Inc., USA, 2nd edition, 1995.

[26] Victor Shoup. *A Computational Introduction to Number Theory and Algebra*. Cambridge University Press, USA, 2005.

[27] Simon Singh. *Fermat's Last Theorem*. Fourth Estate, UK, 1997.

[28] Demba Sow, Leo Robert, and Pascal Lafourcade. Linear generalized ElGamal encryption scheme. Cryptology ePrint Archive, Report 2020/496, 2020.

[29] Donald Spector. Musical scales and multiplicative groups. In Eve Torrence, Bruce Torrence, Carlo Séquin, and Kristóf Fenyvesi, editors, *Proceedings of Bridges 2018: Mathematics, Art, Music, Architecture, Education, Culture*, pages 387–390, Phoenix, Arizona, 2018. Tessellations Publishing.

[30] Donald Spector. Images produced via modular multiplicative inverse filters. In Susan Goldstine, Douglas McKenna, and Kristóf Fenyvesi, editors, *Proceedings of Bridges 2019: Mathematics, Art, Music, Architecture, Education, Culture*, pages 367–370, Phoenix, Arizona, 2019. Tessellations Publishing.

[31] Loai Tawalbeh and Alexandre Tenca. An algorithm and hardware architecture for integrated modular division and multiplication in GF(p) and GF(2/sup n/). In *Proceedings. 15th IEEE International Conference on Application-Specific Systems, Architectures and Processors*, pages 247– 257, 2004.

[32] Loai A. Tawalbeh and Saadeh Sweidan. Hardware design and implementation of ElGamal public-key cryptography algorithm. *Information Security Journal: A Global Perspective*, 19(5):243–252, 2010.

[33] Xin Zhou, Koji Nakano, and Yasuaki Ito. Efficient implementation of FDFM approach for euclidean algorithms on the FPGA. *International Journal of Networking and Computing*, 6(2):420–435, 2016.