A High-Performance Hardware Design for Computing Bézout's Coefficients Using Extended Stein's Algorithm

Ryan Massie and Firas Hassan
Ohio Northern University, r-massie, f-hassan@onu.edu

***Abstract* - Bézout's Coefficients are critical mathematical constructs in cryptography, particularly for public key cryptographic applications requiring increasingly large keys. As software-based methods for calculating these coefficients approach their performance limits, hardware implementations present a promising alternative for faster and more efficient computation. This paper proposes a scalable 64-bit hardware implementation for the computation of Bėzout's coefficients using a previously proposed Extended Stein's Algorithm Implementation which replaces the multiplication and division operations found in Extended Euclid's Algorithm with addition and binary shift operations. The algorithm uses a Controller and Datapath approach with the aim of shifting hardware complexity from the Datapath to the control logic, thus shortening the critical path and improving performance. The hardware implementation efficiency is evaluated, its advantages and limitations are discussed. The results demonstrate that the proposed implementation achieves efficient computation of Bézout's coefficients with a minimum clock period of 8 ns when synthesized on a Basys 3 Artix-7 FPGA. The design effectively balances performance and resource utilization, requiring 1628 LUTs and 535 FFs while maintaining scalability for larger bit sizes. This work establishes a foundation for future research into hardware accelerated cryptographic computations, particularly in applications requiring fast modular inverses and efficient key generation.**

Index Terms – Bézout's Coefficients, Binary Euclid, Hardware Implementation, Modular Inversion.

INTRODUCTION

The growing demand for secure communication systems has made cryptographic algorithms a cornerstone of modern technology. Efficient computation of mathematical constants, such as Bézout's coefficients, plays a crucial role in areas such as RSA encryption and Elliptic Curve Cryptography. Public and private key pairs in asymmetric cryptography heavily rely on the ability to compute the greatest common divisor (GCD) of large integers to establish coprimeness, which is fundamental to key generation and encryption processes [1] [2]. Bézout's Identity is an integral cornerstone for a multitude of widely used mathematics algorithms. The identity states that for any two integers a, b there exists two coefficients S_p, S_q such that $p * S_p + q * S_q = gcd(p, q)$ [3]. Bézout's coefficients play a crucial role in the computation of modular reciprocals, particularly in contexts where two integers are coprime, that is, their greatest common divisor equals 1. According to Bézout's identity, for any pair of coprime integers the identity can be rearranged to show that S_p serves as the modular inverse of p modulo q, meaning $p^{-1} \equiv S_p \, mod(q)$ or $p^{-1} * S_p \, mod(q) = 1$. The ability to compute modular inverses is essential in many cryptographic algorithms, including RSA and Elliptic Curve Cryptography (ECC), where secure key generation in RSA, and point multiplication in ECC depend on the modular inverse. As a result, Bézout's coefficients provide not only a theoretical foundation for these calculations but also a practical method for ensuring efficient and reliable cryptographic computations [4].

Algorithms for calculating Bézout's Coefficients trace the steps taken during the GCD computation and solving for how each intermediate remainder can be written in terms of p and q, thus are extensions of existing GCD algorithms. However, as the processing power of computers has increased, the need for larger keys, more resilient to brute force attacks, has grown with the desired key size reaching thousands of bits. Traditional software-based approaches to computing Bézout's coefficients are computationally expensive for large integers, limiting their scalability in cryptographic applications. To address this, hardware implementations have emerged as a promising alternative, but existing designs often trade off efficiency for complexity with the goal of achieving maximum performance [5]. While these approaches provide vastly superior performance relative to software implementations, the large areas and power consumption required could limit their applicability. There exists a space between these two extremes where the performance benefits of digital hardware can be leveraged while providing a more efficient implementation.

In this paper, we propose a novel hardware implementation of an Extended Stein's Algorithm (ESA) for the efficient computation of Bézout's coefficients. Although Stein's Algorithm necessitates more iterations than Euclid's Algorithm, its reliance on binary shifts, addition, and swapping instead of division and multiplication enables significant speedup in hardware. Our hardware further optimized the design by eliminating the need for the

hardware swap, instead utilizing register pointers to the register file in the controller. By replacing the complex multiplication and division operations in Euclid's algorithm with binary shifts [6], our design achieves improved performance and reduces hardware complexity. The proposed hardware is designed for applications requiring fast modular inverse computation and efficient handling of large cryptographic keys. This work focuses on optimizing hardware performance and analyzing trade-offs in complexity and execution speed. Alongside proving the design's scalability and optimization for cryptographic applications that return the GCD and Bézout's coefficients. The split controller and datapath approach allow the controller to remain the same regardless of bit size. Additionally, the performance and efficiency are discussed.

The remainder of the paper is organized as follows; Section II focuses on the intricacies of the Extended Stein's Algorithm with the adaptations required for hardware in section III. The hardware description and design review are found in Section IV. Section V focuses on implemented FSM and controller logic with simulations results in Section VI. Section VII highlights potential optimizations and improvements with concluding remarks in Section VIII.

RELATED WORK

There are several algorithms to compute the GCD of two numbers one of the most prominent, Euclid's Algorithm, is favored for requiring fewer iterations than other algorithms Euclid's algorithm takes an average of $\sim 0.58lg(N)$ iterations at scale with a maximum of $\sim 1.44lg(N)$. While beneficial for minimizing the total number of iterations, the multiple multiplications/divisions per iteration are challenging for traditional binary computers to perform. The Binary GCD algorithm, also known as Stein's Algorithm, was developed to eliminate the need for divisions at the expense of requiring more iterations on average, requiring an average of $\sim 0.70lg(N)$ iterations with the same maximum as Euclid's [7]. As a result of the complexity of binary division, it is beneficial for computers to utilize Stein's Algorithms as the decrease in iteration complexity and execution time outweighs the increase in the total number of iterations.

The Extended Euclidean Algorithm (EEA) is a well-known algorithm whose inner workings will not be discussed at length. The version of an Extended Stein's Algorithm (ESA) utilized to design the proposed hardware implements the algorithm in [8]. The algorithm utilizes a top-down approach like EEA where the divisors p and q with the constants S_p, S_q to maintain the invariant $p * \mathrm{Sp} + q * \mathrm{Sq} = gcd\ (p, q)$. Due to the need to swap p and q to ensure q > p, the algorithm introduces two additional coefficients associated with q, t_p and t_q. The algorithm's pseudocode is shown below.

The algorithm begins by handling base cases where either input is zero. If p = 0, the result is (q, 0, 1), and if q = 0, the result is (p,1,0). Otherwise, the algorithm proceeds by removing common factors of two from both numbers, reducing them while tracking the number of shifts applied. Once the algorithm establishes that both p and q are coprime, the algorithm initializes auxiliary variables to track Bézout's coefficients. These variables, p_0 and q_0, store the original values of p and q, while S_p, S_q, t_p, and t_q are initialized to represent the coefficients associated with each value. The next stage of the algorithm consists of iteratively reducing p and q while maintaining the relationships necessary to compute Bézout's Coefficients.

Algorithm 1. Extended Stein's Algorithm [8]

Inputs:
- p of length n with sign bit S_p
- q of length m with sign bit S_q

Outputs:
- gcd(p, q) of length max(n, m)
- Bézout's coefficients x and y

```
Initialize K = 0
If p = 0, return (q, 0, 1)
If q = 0, return (p, 1, 0)
While p and q are even:
    p = p / 2
    q = q / 2
    K = K + 1
Initialize p0 = p, q0 = q, Sp = 1, Sq = 0, tp = 0, tq = 1
While p is even:
  If Sp or Sq is odd:
      Sp = Sp – q0
      Sq = Sq + p0
  p = p / 2
  Sp = Sp / 2
  Sq = Sq / 2
While q is not zero:
  While q is even:
    If tp or tq is odd:
      tp = tp – q0
      tq = tq + p0
    q = q / 2
    tp = tp / 2
    tq = tq / 2
  If p is greater than q:
    Swap p and q
    Swap Sp and tp
    Swap Sq and tq
  q = q - p
  tp = tp - Sp
  tq = tq - Sq
Compute final output: gcd = p × 2^K
Return gcd, Sp, Sq
```

The first while loop eliminates factors of two from p. Since division by two is equivalent to a right shift, it is computationally efficient, but adjustments must be made when the corresponding coefficients S_p and S_q are odd. To maintain integer arithmetic, the algorithm subtracts q_0 from S_p and adds p_0 to S_q before performing the shift operation. This ensures that the coefficients remain valid while

maintaining the invariant. A similar process follows for q, but this reduction occurs inside the main loop that performs the core GCD computation. Within this loop, when q is even, the corresponding coefficient values are adjusted in the same manner as for p. Once q is no longer even, the algorithm proceeds to the subtraction step, ensuring that p always holds the smaller value. If p > q, a swap operation is performed on p and q as well as on their associated coefficients. This guarantees that the larger value is always reduced by the smaller one, maintaining the invariant and ensuring the correctness of the computation. The loop continues until q = 0, at which point p holds the final GCD. Since the algorithm previously removed common factors of two, the result must be multiplied by 2^K to restore these factors. The algorithm then returns the computed GCD along with the Bézout's Coefficients, which satisfy $p * \mathrm{Sp} + q * \mathrm{Sq} = gcd\ (p, q)$.

This extended version of Stein's Algorithm maintains the computational efficiency of the binary GCD method while incorporating the ability to compute Bézout's coefficients without requiring division operations. The avoidance of explicit division makes this method well-suited for hardware implementations, where division is costly and shift-based operations are preferable. The structure of the algorithm lends itself to implementation in digital hardware for cryptographic applications.

ADAPTATION TO DIGITAL HARDWARE

The goal of this implementation was to create a faithful adaptation of the original Extended Stein's Algorithm, optimizing and adapting where possible. A key inefficiency in implementations of the Extended Stein's Algorithm is the need for frequent variable swaps when ensuring that p always holds the smaller value. While software implementations can handle these swaps with minimal overhead, hardware implementations face additional complexity, as swapping requires additional control logic and its own dedicated step in a hardware state machine design, increasing resource utilization and execution time. To address this, an architecture that eliminates the need for explicit swaps by restructuring the algorithm's data flow.

Instead of performing conditional swaps, the proposed approach will employ selective assignment operations utilizing control signals to maintain the correct relationships between p and q without requiring explicit exchange operations. By dynamically adjusting which variables participate in subtraction, shift and coefficient updates, the design can ensure that the larger value is always reduced without reordering registers. This not only simplifies control logic but also improves computational efficiency by reducing cycle counts associated with conditional branching and temporary storage operations. By removing explicit swaps, the architecture can achieve a more streamlined and efficient execution while preserving the correctness of the Extended Stein's Algorithm. To accommodate these changes the p and q values are abstracted to X, and Y, along with the associated X_p, X_q, Y_p, Y_q allowing for a structured representation that eliminates the need for explicit swaps while preserving correctness. Since the aim is to remove the hardware swap the relationship between p and q must be tracked in the control logic.

An additional design decision is that our modified algorithm operates under the assumption that an external hardware check is in place to guarantee that neither input is zero before execution begins. This assumption simplifies the design by eliminating the need for internal zero checks, allowing the hardware to focus solely on performing the core computations of the extended Stein's binary GCD algorithm. By offloading this responsibility to an external pre-processing unit, we reduce logic overhead and streamline the control flow, ensuring efficient execution without unnecessary conditional branches. However, it is crucial that the system integrating this design enforces this requirement, as failure to do so could lead to undefined behavior or incorrect results.

PROPOSED HARDWARE

The proposed Extended Stein's hardware (Figure 1) takes two 64-bit inputs, INPUT1 and INPUT 2, and returns three 64-bit outputs, GCD, COEFFICIENT1 , and COEFFICENT2 which satisfies the Bézout's invariant $INPUT1 * COEFFICIENT1 + INPUT2 * COEFFICIENT2 = GCD$. It is controlled by an external clock and enable signal. The Datapath implementation, Fig. 3, utilizes a register file combined with hardware arithmetic circuits to compute the necessary values at each stage. The controller operates the hardware using the following signals:

- Load-Select
- Reset-Reg
- Done (2-bit)
- X-Check-Ahead
- Y-Check-Ahead
- Reg-Read (33-bit)
- Reg-Write (24-bit)
- Reg-Write-En

The core of the Datapath is the specialized register file. X, X_p, X_q, X_0, Y, Y_p, Y_q, Y_0 are stored. The register file contains 11 reads controlled by a 33-bit signal, Reg-Read, from the controller. There are 10 writes, 8 addressable, 2 fixed. A 24-bit signal, Reg-Write, sets the write destination. Write 8 and 9 are connected to the X/Y subtractors and can only write to the respective registers. All writes are controlled by the 10-bit Reg-Write-En signal.

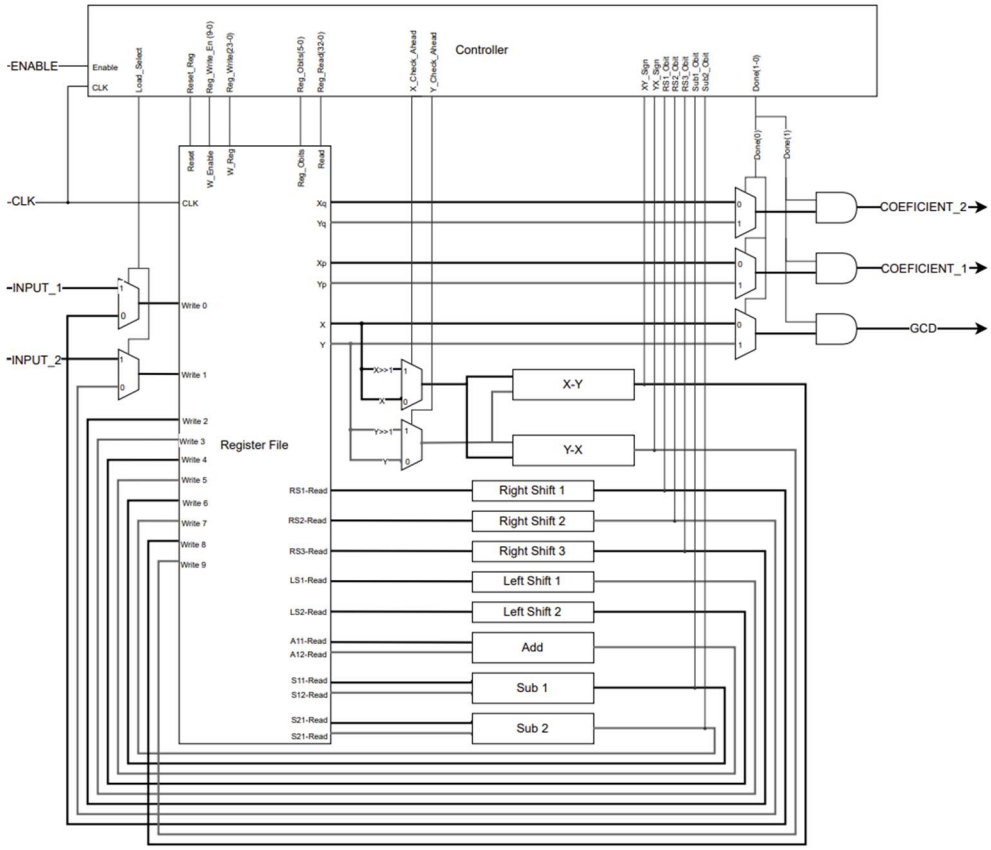

Fig. 1. Hardware Datapath

The Datapath must be carefully designed such that it provides all necessary information for the controller before the end of the current clock cycle such that the controller can determine the next step. The hardware utilizes 3 right shifts, 2 left shifts, 1 adder, and 4 subtractors of which 2 are dedicated to X minus Y and vice versa operations. The control signals needed by the controller are included where appropriate. The X/Y-Check-Ahead values are set high when the controller their respective SHIFT state to allow the controller to know whether the resultant X or Y value is larger than its counterpart. Lastly, the done signal controls which GCD/Coefficients are returned by the hardware.

FINITE STATE MACHINE IMPLEMENTATION

The hardware design follows a structured Controller and Datapath architecture, where the Controller manages the operation flow by implementing a Moore finite state machine (FSM). This FSM orchestrates the sequence of operations, with each state corresponding to a specific task in the design process. The states include:

- LOAD
- SHRINK
- ADJUST X
- SHIFT X
- REDUCE X
- ADJUST Y
- SHIFT Y
- REDUCE Y
- INFLATE
- DONE X
- DONE Y

In the context of the hardware implementation of Stein's binary GCD algorithm, focusing on the core of the state machine and examining the relationship between the states ADJUST X, SHIFT X, and REDUCE X is crucial for understanding how the algorithm progresses. These three states directly influence the manipulation of the X value, which is one of the primary operands in the GCD computation. Fig. 2. demonstrates the relationship between the three states.

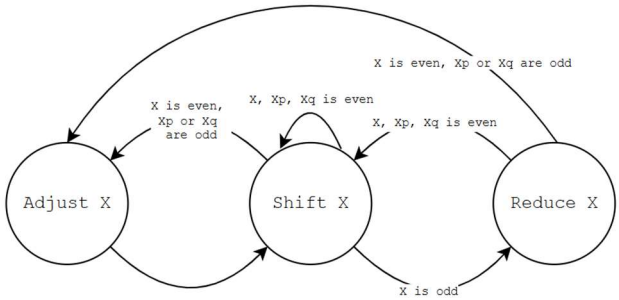

Fig. 2. Finite State Machine Core

Each step fulfills a core component of the original Extended Stein's Algorithm. ADJUST X modifies the X_p and X_q values by the operand by the original Y_0 and X_0 values, thus ensuring that both are even. This ensures that when SHIFT X divides all values by 2 via an arithmetic right shift to preserve sign, all three are even. However, this approach introduces an additional iteration compared to the original algorithm as the software can handle the adjustment within the shift iteration. Once X is no longer even the

reduce state subtracts Y from X, as both are negative the result is Y is even. This FSM is paired with a matching one for handling Y, Fig. 3 above.

A key breakthrough in design of the hardware was the realization that the only time the algorithm needs to transition from working on X to Y and vice versa is when a shift operation occurs, as it is the only time in the original algorithm where P could become larger than Q is when Q is divided by 2. In the context of our implementation the only jump from the "X" to "Y" side of the hardware occurs when going from a Shift to a Reduce State.

A key aspect of the design is the necessity for the next state to be determined before the end of each clock cycle, ensuring that the FSM can seamlessly transition between states without causing delays or errors. Eliminating the need for any additional states Minimizing the already increased iterations of Stein's algorithm wherever possible is critical to maintaining the algorithms' performance. Thus, the algorithm should be able to compute the next step before the end of the current clock cycle. The controller's inputs are as follows:

- **CLK:** Clock signal for the controller.
- **Enable:** Enables the state machine to process states on the rising edge of the clock.
- **XY-sign:** Input signal indicating the sign of the X minus Y resultant value.
- **YX-sign:** Input signal indicating the sign of the Y minus X resultant value.
- **RS1-obit:** Input signal for the RightShift1 resultant odd bit
- **RS2-obit:** Input signal for the RightShift1 resultant odd bit
- **RS3-obit:** Input signal for the RightShift1 resultant odd bit
- **Sub1-obit:** Input signal indicating the first subtraction operations odd bit.
- **Sub2-obit:** Input signal indicating the second subtraction operation odd bit
- **Reg-obits:** 6-bit input vector (5-0) representing register odd bits used to track X, Y, and their coefficients ordered (Y, Y_p, Y_q, X, X_p, X_q).

The LOAD state requires the controller to determine if it should jump straight to DONE, which occurs when both numbers are equal, whether to SHRINK because both X and Y are even or determine where in the core FSM it should continue from. Load is returned to upon enable being set low. The SHRINK state also must determine where in the core FSM to jump to whenever X and Y are both no longer even. SHRINK also highlights the controller's internal logic needed to track the number of shift values (K). The maximum value of K is equal to the unsigned bit size required to store the N-bit size of the hardware, and thus $K_{Size} = \log_2(N)$. Since the selected implementation for this publication is 64 bits, $K_{Size} = \log_2(64) = 6$-bits. When the state is SHRINK K is incremented and decremented when the state is INFLATE. The controller therefore maintains a 6-bit register, adder, and subtractor to perform these calculations.

The Controller knows the GCD is complete when both XY-Sign and YX-Sign are zero in a REDUCE state, represented in the FSM as "Complete". This mirrors the original algorithm checking if Q = 0 after reducing Q by P. If K > 0, The controller shifts X and Y by 2 utilizing left shifts to restore the GCD. Once K = 0, the hardware proceeds to either of the Done states which dictate whether X, X_p, X_q or Y, Y_p, Y_q are the resultant outputs.

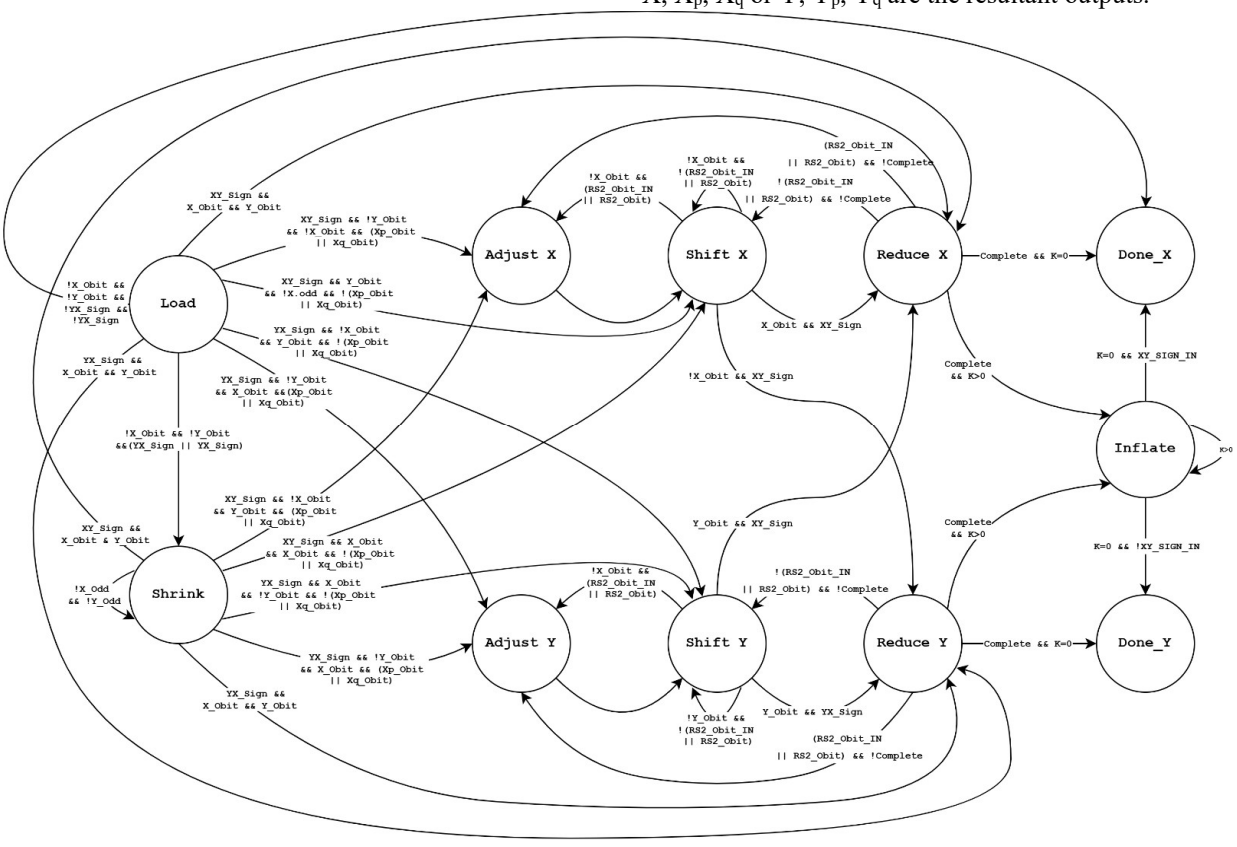

Fig. 3. Finite State Machine

SIMULATION RESULTS

The hardware implementation fulfills the needs of the described Extended Steins's FSM. Our hardware was described in Very High-Speed Integrated Circuit Hardware Description Language [9] and implemented on a Basys 3 Artix-7 Field Programmable Gate Array (FPGA). The synthesized hardware requires 1628 Look Up Tables and 535 Flip Flops. Our implementation was wrapped in a 192-bit shift registers to meet the I/O limitations of the FPGA device and the resultant minimum clock period was 8ns. Below Table 1 shows a running example of the implemented FSM with an initial p and q of 10 and 12 respectively and Fig. 4 demonstrates the resulting waveform from the hardware simulation. Each iteration of algorithm in the table directly corresponds with a matching clock on the waveform. The hardware beings with a shrink to ensure the values are coprime, mirrored with an inflate to ensure the final GCD is correct. The resulting waveform demonstrates the correct execution and coefficients of -1 and 1 and GCD of 2.

TABLE 1.
HARDWARE EXAMPLE

Step #	X	Y	Xp	Xq	Yp	Yq	Step
1	10	12	1	0	0	1	Load
2	5	6	1	0	0	1	Shrink
3	5	6	1	0	-6	6	Adjust Y
4	5	3	1	0	-3	3	Shift Y
5	2	3	4	-3	-3	3	Reduce X
6	2	3	-2	2	-3	3	Adjust X
7	1	3	-1	1	-3	3	Shift X
8	1	2	-1	1	-2	2	Reduce Y
9	1	1	-1	1	-1	1	Shift Y
10	1	0	-1	1	0	0	Reduce Y
11	2	0	-1	1	0	0	Inflate
12	2	0	-1	1	0	0	Done

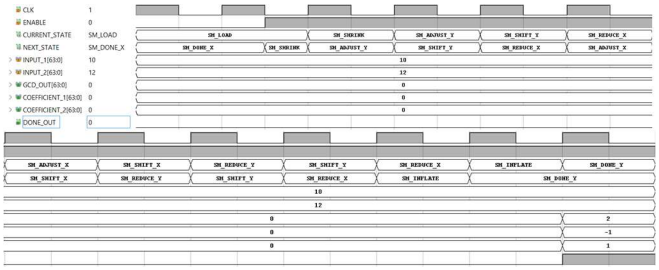

Fig. 4. Waveform Diagram

Several possibilities for improvement could allow this hardware design to be further optimized. The current implementation uses simple ripple carry adders. Substituting these with faster alternatives, including carry look-ahead adders, would improve hardware performance. Alternatively, it could be possible to substitute the two X/Y subtractors for a single subtractor that performs the relevant operation with necessary. The key limitation is devising a more efficient manner for determining when the hardware is done as it relies on the sign bit for both subtractors.

CONCLUSION

This work presents an easily 64-bit hardware implementation of the Extended Stein's Algorithm that scales to large bit sizes without affecting control logic or overall hardware architecture. It proves effective for computing Bézout's coefficients, leveraging binary shift operations to improve efficiency over traditional software-based approaches. By shifting computational complexity from the Datapath to the control logic, the proposed design optimizes performance while maintaining a scalable and practical architecture for cryptographic applications. The results demonstrate that although Stein's Algorithm requires more iterations than Euclid's Algorithm, its reduced cycle time enables overall performance gains, particularly in hardware implementations where division and multiplication are costly. Improvements can also be made for applications where the GCD is unnecessary such as RSA where the GCD is odd, and the input numbers are large primes. This would allow for the elimination of the SHRINK and INFLATE states and several components from the Datapath.

Overall, this work demonstrates the feasibility of using hardware acceleration for Bézout's coefficient computation, balancing efficiency and resource utilization. By leveraging the Extended Stein's Algorithm, the implementation mitigates the performance limitations of traditional software-based approaches while maintaining a manageable hardware footprint. This study serves as a foundation for future research into optimizing GCD-based computations in cryptographic systems, reinforcing the role of specialized hardware in secure and efficient key management.

REFERENCES

[1] V. Kapoor, V. S. Abraham, and R. Singh, "Elliptic Curve Cryptography," Ubiquity, vol. 2008, no. May, pp. 1–8, May 2008, doi: https://doi.org/10.1145/1386853.1378356

[2] R. L. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," Communications of the ACM, vol. 21, no. 2, pp. 120–126, Feb. 1978, doi: https://doi.org/10.1145/359340.359342. Available: http://people.csail.mit.edu/rivest/Rsapaper.pdf.

[3] E. Bézout, "Théorie générale des équations algébriques," *Internet Archive*, 1779. Available: https://archive.org/details/thoriegnra00bz

[4] "Disquisitiones arithmeticae : Gauss, Carl Friedrich, 1777-1855 : Free Download, Borrow, and Streaming : Internet Archive," *Internet Archive*, 2018. https://archive.org/details/disquisitionesa00gaus.

[5] K. Sreedhar, M. Horowitz, and C. Torng, "Fast Large-Integer Extended GCD Algorithm and Hardware Design for Verifiable Delay Functions and Modular Inversion," IACR Transactions on Cryptographic Hardware and Embedded Systems, pp. 163–187, Aug. 2022, doi: https://doi.org/10.46586/tches.v2022.i4.163-187. Available: https://eprint.iacr.org/2021/1292.

[6] R. P. Brent, "Twenty years' analysis of the binary Euclidean algorithm," in Millennial Perspectives in Computer Science: Proceedings of the 1999 Oxford-Microsoft Symposium in Honour of Professor Sir Antony Hoare, J. Davies, A. W. Roscoe, and J. Woodcock, Eds. New York: Palgrave, 2000, pp. 41-53.

[7] R. P. Brent, "Analysis of the Binary Euclidean Algorithm," ACM SIGSAM Bulletin, vol. 10, no. 2, pp. 6–7, May 1976, doi: https://doi.org/10.1145/1093397.1093399

[8] J. R. Barkema, "Extending Stein's GCD Algorithm and a Comparison to Euclid's GCD Algorithm," Studenttheses.uu.nl, Jun. 2019, Available: https://studenttheses.uu.nl/handle/20.500.12932/33194.

[9] Ryan-B-Massie, "GitHub - Ryan-B-Massie/ExtendedBinaryGCDHardware," GitHub, Feb. 12, 2025. Available: https://github.com/Ryan-B-Massie/ExtendedBinaryGCDHardware/tree/main.