Hardware Implementation of a Partially Homomorphic Cryptosystem

Rafael Gomez
EECS Department
Massachusetts Institute of Technology
ragoo@mit.edu

Luis Turino
EECS Department
Massachusetts Institute of Technology
turino14@mit.edu

Nicholas Gorbea
EECS Department
Massachusetts Institute of Technology
nggorbea@mit.edu

***Abstract*—This report presents the design of a Paillier cryptosystem implemented entirely on an FPGA fabric. Homomorphic cryptosystems, such as Paillier, are known to provide large computational overhead – we present advances to overcome some of these limitations. We utilize a *block stream* approach to handle the logic of the large numbers of our system, and we implement the hardware to exploit optimizations enabled by this approach in order to encrypt and decrypt high quantities of votes in a simulated election. Finally, we evaluate the performance of our system compared to a software implementation over a general purpose processor to demonstrate that our system is efficient – even compared to powerful CPUs.**

***Index Terms*—partially homomorphic encryption, field programmable gate arrays, Cryptography**

I. System Components

Our cryptosystem design consists of implementing the following components on our Spartan 7 Xilinx FPGAs

- 4096-bits operators: Multiplier, Divider/Modulo, Adder, Subtractor, Shifter, Bits Selector
- 4096-bits over 2048 bits Exponentiation Module
- 4096-bits Random Number Generator
- 8192-bits Montgomery Reducer
- UART Transmitter & Receiver
- SPI Transmitter & Receiver s
- Redundancy Check Modules

As of the date of this preliminary report, we have successfully implemented, simulated, and synthesized the Montgomery Reducer and all the 4096-bits components, save the 4096-bits Divider which is in active work. In III we go in detail of our design approaches for these components and their simulated results.

II. Encryption Scheme

A. Mechanism

The purpose of our system is to run a *secure election* demo, following the desirable properties of:

- **Vote secrecy** – no one with access to the vote database can determine what a given person voted for.
- **Confirmable votes** – any individual voter can confirm their vote is being counted correctly, at any point in time.
- **Vote-selling prevention** – voters can never prove they voted in a specific way, which otherwise would allow for a mechanism of selling and buying votes.

Our system achieves these properties through the implementation of Paillier Cryptosystem, a Partially Homomorphic Encryption (PHE) scheme. It allows us to compute the tally of an election without ever decrypting individual votes. In particular, the product of two ciphertexts will decrypt to the sum of their corresponding plaintexts.

$$D(E(m_1, r_1) \cdot E(m_2, r_2) \bmod n^2) = m_1 + m_2 \bmod n \quad (1)$$

Where m_1, m_2 are the messages in plaintext, r_1, r_2 are random 4096-bits numbers, n is the product of two 1024-bits primes, and *D* and *E* are our Paillier decrypt and encrypt functions.

Here, if a voter wants to vote for candidate A, they will cast a ballot with messages $m_A = 1$ for said candidate, and $m_{-A} = 0$ for the others.

A Paillier voting machine will encrypt the message using *E* and send it to a Tallier machine. For reference, a *ballot receipt*, consisting of the corresponding ciphertext, is given back to the voter. The Tallier will multiply the ciphertexts of all voters. By the homomorphic property (1), decrypting this product will be equivalent to adding 1 for every vote in favor for a candidate, and 0 not in favor.

Once an election is declared over, the Tallier sends the votes to the Decryptor that possesses the private key and the decryption function *D*, deciphering the result tallies for every candidate and announcing the winner.

To allow for *confirmable votes*, every constituent has access to the database of all votes – allowing them to check that their encrypted stored vote matches their ballot receipt and, if they were to perform the tally product, they would obtain the same encrypted tally announced by the official Tallier.

Paillier is based on the assumption that it is hard to determine whether a given number is an n-residue modulo n^2. In particular, a system implemented with a key size of 2048-bits, such as the one presented, will require a computational effort of $3 \cdot 10^{20}$ MIPS-year [2]. Hence, no one without the private key will be able to determine individual votes, granting *vote secrecy*.

Moreover, while voters may compare their ballot receipt with the ciphertext in the database, they can only verify that the vote is unchanged, but cannot confirm or demonstrate to whom the vote was casted for, as that would require the private key. This ensures *vote-selling prevention*.

Other valid security concerns – such as how to improve trustworthiness of the Decryptor machine (e.g. by distributing the decryption power over many trustworthy officials using Shamir's), how to ensure the voting machine does not output a modified vote (e.g. by implementing Benaloh's challenge), how to prevent fake or double votes (e.g. through a Validator Authority) or the general question of how to make the system more accessible to the public – will not be focused on in this demo, but are posed as future points of exploration.

B. System Design

For our current system, we will implement the case of a two-candidate election. A generalized system – as described above – is only a matter of scaling the number of messages by the number of candidates.

Our goal is to implement the above mechanism in an efficient manner. In figure 1, we demonstrate this by having an "Encryptor FPGA" acting as the Voting Machine, and a "Decryptor FPGA" acting as both the Tallier and Decryptor. The act of a voter casting a ballot is simulated and sent to the Voting Machine via UART. The Encryptor and Decryptor communicate through SPI and the data clock period is increased in order to prevent loss of ballots.

There are several essential modules required for implementing a complete Paillier cryptosystem [1].

Key Generation. From two large primes p,q and a large random number g, we calculate the public key for our Encryptor, $\boldsymbol{(n, g)}$ where n=pq. The private key for our Decryptor will be $\boldsymbol{(\lambda, \mu)} = (\varphi(n), \varphi^{-1}(n) \bmod n)$, where φ is Euler's totient function. Since our purpose is optimizing encryption and decryption, we precompute these one-time generated keys outside the boards, and provide them to the corresponding modules.

Encryptor – [Figure 2] A random 4096-bit number **r** is generated for every encryption. Then, given a message m and public key (n, g), the encryption function is

$$\boldsymbol{E(r, n) = g^m \cdot r^n \bmod n^2} \tag{2}$$

Decryptor – [Figure 3] Given a ciphertext c (or product of ciphertexts), and given public key (n, g), private key (λ, μ), the decryption function is [1]

$$\boldsymbol{D(c) = \frac{(c^\lambda \bmod n^2) - 1}{n} \cdot \mu \bmod n} \tag{3}$$

A major challenge of the system is implementing the large numbers operators. Evidently, it is infeasible to carelessly utilize 4096-bits wires and registers, as that would exhaust the resources and violate timing constraints. Instead, we took inspiration on how CPython implements large numbers [4]. We divide the 4096-bits number into 32-bits ***blocks***, and implement our operators by only working with a few blocks at a time.

To allow components to access blocks, we considered two different approaches:

1) *Centralized Memory*. Modules would send read and write requests to a *Memory Arbiter* utilizing pointers. Modules would interact by sending other modules pointers referencing to the desired numbers. However, the arbiter adds much complexity to the system, and utilizing memory in this manner adds overhead circumventable by the next approach. *We do not use this method.*
2) *Modules Data Streaming*. This is allowed by designing the system such that every module only depends on the module immediately before it. As blocks are processed, these modules either throw away the original input (e.g. addition) or "cache" away the entries to perform the computation if necessary (e.g. multiplication). The results are sent in blocks to the next components for processing, continuing the cycle in a serial fashion.

Henceforth, our design will follow a *block stream approach*.

C. System Optimization – Montgomery

The 4096-bits over 2048-bits Modular Exponentiation Operator is the main bottleneck of both encryption and decryption. A 4096-bits long-division Divider (acting also as a Modulo operator) requires at least 4 million cycles. If it were to be used in the modular exponentiation, it would need to be run 2048 times, resulting in an impractical lowerbound of 85.9 seconds[2] to encrypt a single vote.

To tackle this, we utilize *Montgomery Modular Multiplication* [3], which allows us to substitute almost every single divider with much faster multipliers, shifters, and bit selectors.

To compute $\boldsymbol{a^{2^u} \bmod N}$, the Montgomery approach is:

1) Compute $\overline{a} = aR \bmod N$, where $\boldsymbol{R = 2^{\lceil log_2 N \rceil}}$. Moreover, this can be optimized to avoid using the Divider by precomputing $\overline{R} = R^2 \bmod N$, since $\text{Redc}(a\overline{R}) = aRRR^{-1} \bmod N = aR \bmod N = \overline{a}$.

$$\text{Redc}(T) = TR^{-1} \bmod N \tag{4}$$

2) Calculate $T = \overline{a} \cdot \overline{a} = a^2R^2$.
3) Calculate $\text{Redc}(T) = \text{Redc}(a^2R^2) = a^2R \bmod N = \overline{a^2}$
4) Repeat last step with $T = \overline{x} \cdot \overline{x}$, where $\overline{x}$ is the output of the last step. Finish when you get the desired exponent.
5) We obtain $a^{2^u}R$. Reduce one more time to obtain a^{2^u}.

.

Ultimately, this design results in only needing to synthesize two dividers for the entire system! In III-B we go over how to efficiently implement Redc, and in III-C how to utilize it to implement a general $a^n \bmod N$ operator.

[1] $\frac{a}{b}$ denotes the quotient of a over b

[2] Assuming 100MHz Clock Speed

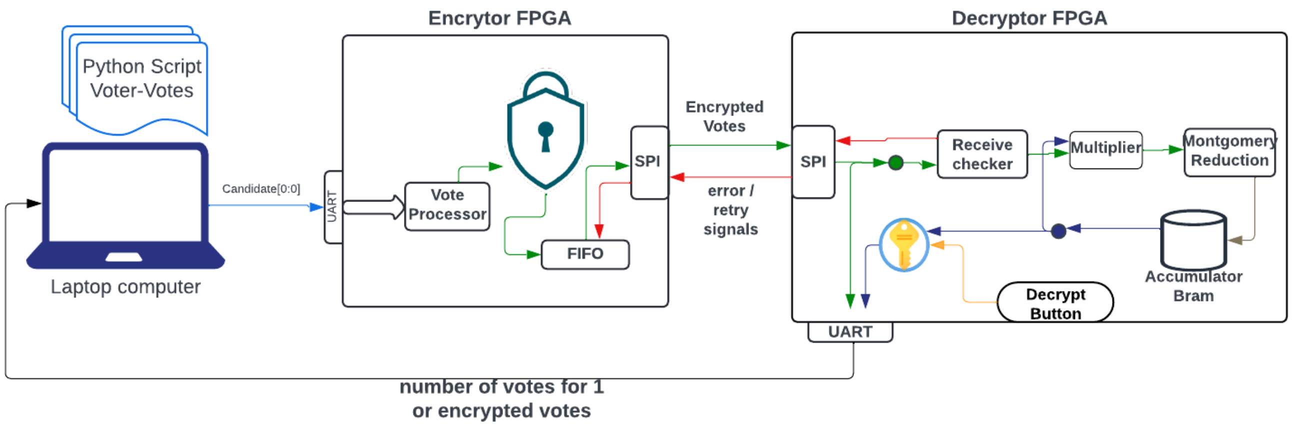

Fig. 1. High-level Block Diagram of the System

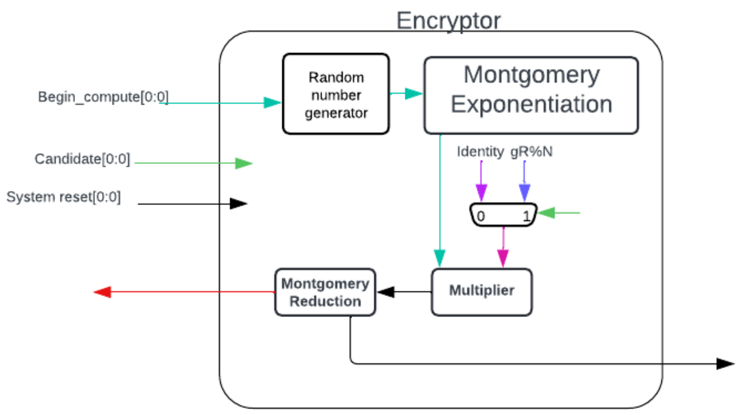

Fig. 2. Encryptor Block Diagram

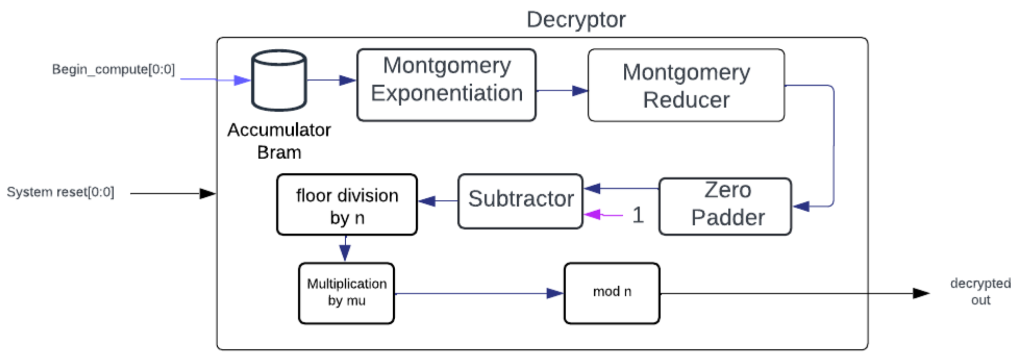

Fig. 3. Decryptor Block Diagram

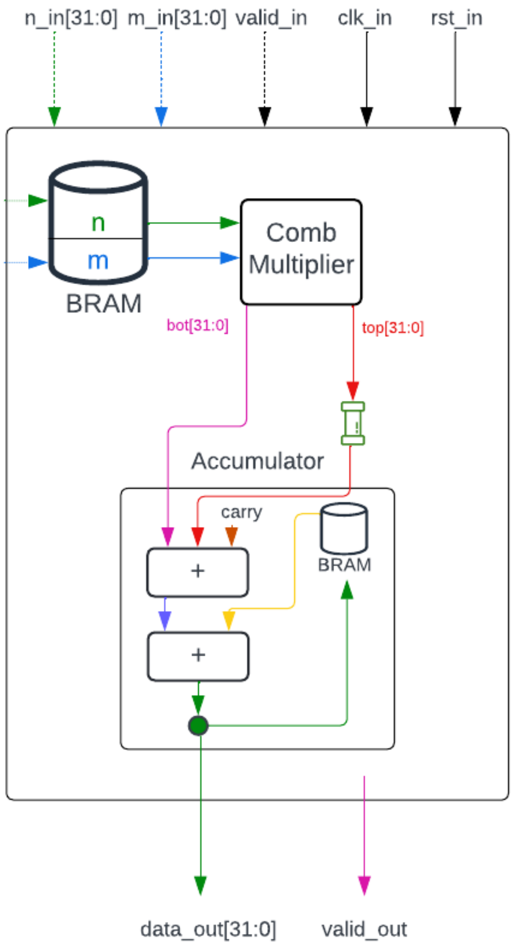

Fig. 4. FSM Multiplier Block Diagram

III. Hardware Implementation

A. *Multiplier*

The performance of our system is tightly dependent on the performance of the 4096-bit multiplier implementation. Montgomery modular exponentiation, the bottleneck of our system, requires looping around three multiplications 2048 times. Hence, we see that

$$t_{system} \approx t_{exp} \approx 2048 * 3 * \text{MultiplierCycles} \ * 10^{-8} \text{ seconds}$$

Historically, efficient multiplication algorithms have been vastly researched. A common theme is the trade-off between speed and resources – usually, calculating a product in less cycles requires utilizing greater number of resources.

Our system utilizes 4 DSPs in order to compute the product of 32-bit blocks combinationally – this allows us to divide our algorithm into the multiplication of two $4096/32 = 128$ blocks numbers.

To implement this, we considered several base algorithms, with potential optimizations:

1) *Grade-school multiplication*. Intuitive algorithm that runs in $O(n^2)$ and utilizes resonable resources.
2) *(Binomial) Karatsuba's*. Very expensive and complex, but runs in $O(n^{\log_2 3})$.
3) *Generalized-polynomial Karatsuba Multiplication*. Moderately expensive, requiring only $\frac{1}{2}n^2 + \frac{1}{2}n$ multiplications. [7]
4) Others, such as Schönhage–Strassen FFT. Proved to be too resource expensive or unpractical, despite significant speeds.

Where n represents the number of blocks, or single-cycle multiplications, of the numbers.

Figure 4 depicts our implementation of the Grade-school multiplication algorithm. It consists of multiplying every 128^2 pair of blocks of the two 4096-bits numbers. After every multiplication, we carefully add the results into an accumulator BRAM. This has to be done with great scrutiny in order to not waste cycles. Our original implementation, fpga_e/fsm_multipler_original.sv, required 82,689 cycles to complete, as it needed to record intermediate products into an additional BRAM. We optimized it, as per Figure 4, by doing the required accumulator additions at the same time as we output results from the multiplier, achieving 16,646 cycles, very close to the limit of Grade-school of $128^2 = 16,384$.

We further optimized in fpga_e/fsm_multipler_parallel.sv by making adding 4 Combinational Multipliers in parallel, to calculate four products simultaneously. This allows us to achieve a 4296 cycles multiplier. However, this multiplier must be used sparingly, as it requires 16 DSPs. Our Encryptor FPGA utilizes 6 parallel and 6 unparallel multipliers, reaching the maximum of 120 DSPs, and attaining

$$t_{system} \approx t_{exp} \approx 2048 * 3 * 4296 * 10^{-8} = 0.264 \text{ seconds}$$

per encryption. This is considerably efficient compared to similarly systems (see IV), and even to modern CPUs. Both Intel i7 and M1 CPUs were able to run the encryption in software in between 0.15 to 0.10 seconds.

As described above, Binomial Karatsuba's requires significant hardware resources that would exhaust our DSPs in our system, making this unfeasible. However, python_scripts/general_karatsuba_demo.py showcases a prototype of a generalized Karatusuba, requiring only one 32-bit Combinational Multiplier and only twice the amount of BRAMs. Implementing this system and parallelizing it would, in theory, allow our encryption to be 4 times faster. This is a future possible avenue of improving our system to outperform modern CPUs.

B. *Montgomery Reduction*

Recall II-C and (4):

$$\text{Redc}(T) = TR^{-1} \text{ mod } N$$

Where $N = n^2$ in our case (1).

To optimize Redc, figure 5 implements the following steps [3]:

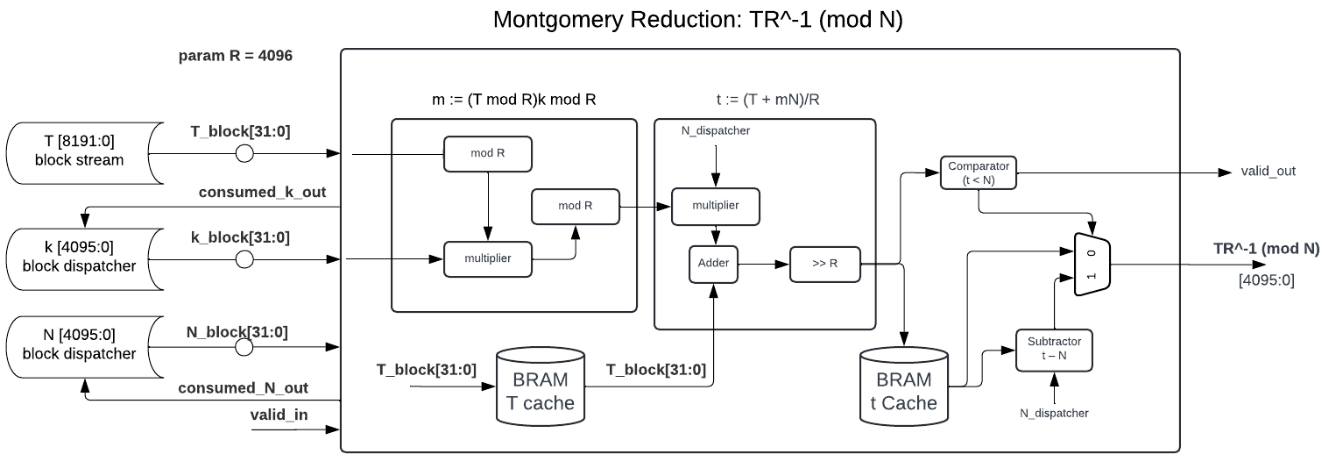

Fig. 5. Montgomery Reduction Block Diagram

TABLE I
4096-BITS MULTIPLIER APPROACHES

Component	***Cycles***	BRAM	DSPs
Grade-school Multiplier, 32-bits Blocks	16,646	1.5	4
Grade-school Multiplier, 64-bits Blocks	4230	3	16
Parallel grade-school Multiplier, 32-bits Blocks	4,296	3	16

1) Compute $m := (T \bmod R) \cdot k \mod R$, where k is a constant from Bezout's $RR^{-1} = kN + 1$
2) Compute $t := (T + mN)/R$
3) if $t < N$ return t, else t-N

Notice that $t \leq 2N$ and $t \equiv TR^{-1} \pmod N$. Moreover, since $R = 2^{\lceil log_2 N \rceil}$, all steps only require shifting, bit selects, addition, or multiplication – but no expensive division, as desired.

The workflow again follows our block stream approach – inputs come in 32 bit blocks representing sections of the 8192 and 4096 bit numbers. As R is just a power of 2, our mod module is invalidates the top half of the input, while the right shift module does the opposite. Addition and subtraction are likewise done in a classic ripple carry fashion whilst the comparator runs a finite state machine as per figure 7. There is some nuance with the signals, however which is why we need (for simplicity) the 2 BRAMs, so we can reuse the input after it has been consumed by other modules.

In terms of efficiency, our module takes 8,980 clock cycles (for the parallel version). Keeping in mind that each parallel multiplier takes 4,296 clock cycles and that they are blocking, we can see that all the non-multiplier logic just adds in $8,980 - 2 \times 4,296 = 114$ extra clock cycles for the rest of the logic.

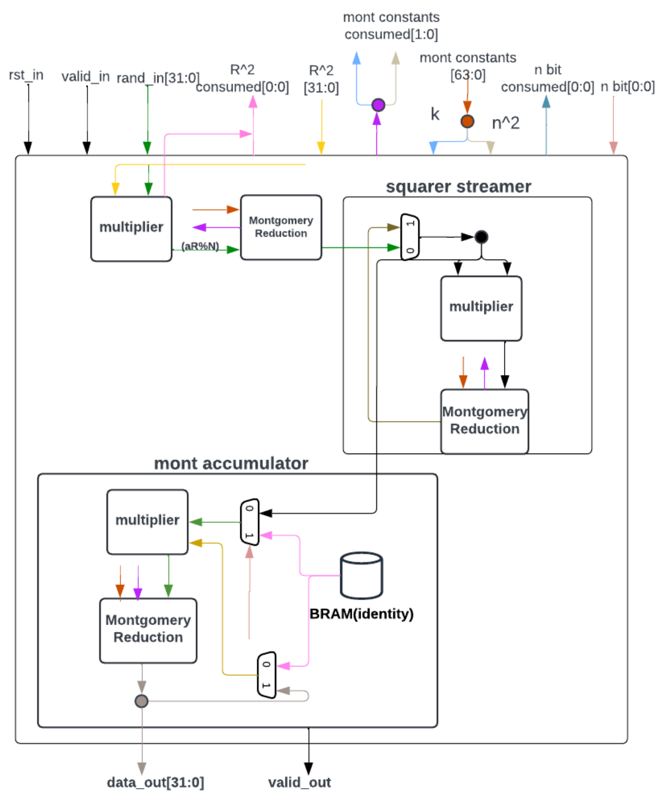

Fig. 6. Montgomery exponentiation Block Diagram

This means that if we want to increase the throughput of this module, the only feasible way is by increasing the throughput of the multiplier.

C. Montgomery Exponentiation

The squaring streamer implements the steps described in II-C to square the input values. For some a, it is initially fed in the value $a \cdot R \pmod N$ (in blocks stream approach). This input will be forwarded, likewise block-by-block, twice to a multiplier which will effectively square the input value. The multiplier now represents the value $a^2 \cdot R^2 \pmod N$ which is then fed into a Montgomery reducer, which cancels out one of the R terms, eventually outputting $a^2 \cdot R \pmod N$. This output, while sent out of the module, is also fed back into the multiplier and a loop repeats, having the squaring streamer effectively compute and send out $a^{2^i} \cdot R \pmod N$. The streamer, unlike in the preliminary report, does not have a specification on whether or not it must terminate after 2048, or some set number of, iterations. Instead, it is the responsibility of the module hooked up to it (in this case, the Montgomery accumulator) to reset the module before passing in its next input. This saves the headache of dealing with off-by-one or timing issues with internal counters.

The accumulator works similarly to the streamer but in addition to the initial state mux there is also an extra mux based on the bit values of n. This mux allows us to switch between the identity operation (multiplying by $R \mod n^2$) and the input of the squarer streamer, effectively allowing us to selectively update the running product with the value of a^{2^x} if 2^x is in the bitwise representation of n. We query the next n bit with the request signal, and when we iterated through the whole number we output the result from the Montgomery reduction, yielding a^n as desired. As hinted above, when the last valid out signal is sent from the accumulator, a reset signal is sent to the streamer.

D. Magic Number Division

Long division is an important algorithm for computing the quotient and residues when no special tricks apply. However, it is also very resource and time expensive – as it is $O(n^2)$ with respect to the number of bits. Moreover, implementing such an algorithm correctly in this scale, would be a real uphill battle with all the large, intermediate values needing to be updated mid-way.

However, from our encryption scheme we can tell we are only dividing by the fixed constant n. This allows us to instead implement algorithms for fixed integer division. In particular, there is a neat trick frequently used by compilers, commonly known as a *Magic Number Divider*. If the quotient of the division is expected to fall in a certain range, you can pre-compute $\frac{1}{n}$ left shifted by some number, and multiplied by some constant. Then, after multiplying and right shifting, you'd receive the result of the division "deleting" the decimal values. Thus, overall allowing division to be implemented via multiplication, shifting, and (depending on the constant) maybe 1 or 2 more cheap, restoring operations gives the expected value.

Another nice detail for our crypto-system is that the inputs to the divider are upper bounded by n^2. This allows us to generate a very convenient constant, making our restoring process single right-shift operation. Thus, division essentially only has the cost of a singular multiplication for our system :).

IV. Evaluation

Our system uses all 120 DSPs and about 50% of the boards block RAM. We also used 54% of the available slices and around 20% consumption for slice logic. There is a trade-off between computational efficiency and resource efficiency for our different modules. However by using more complex routing logic for reads and writes, there should be a way to reduce BRAM consumption for the multipliers to only 1 BRAM block *per* multiplier, as well as potentially decreasing the amount of constants in the top level.

With respect to timing, we have 0.423ns of slack, with a critical path coming just from a 32-bit multiplication and putting it in a register. This means that our system is optimal with respect to timing. We attempted increasing the register to 64, but this caused our system to not meet timing, with the same critical path as before.

Since the exponentiation is a our main bottleneck, our system approximately takes $2048 \cdot (4296 + 8980) \approx 27000000$ clock cycles. This results in about 3.7 encryptions per second.

We ran the same encryption process in a 125 MHz Raspberry Pi, and it took 91.2 seconds to complete, while an M1 MAC took about 0.12 seconds to complete, and our FPGA took about 0.27 seconds. This means that since our simplistic FPGA was able to reach near the speeds of such a modern system (which probably uses hardware accelerators), this indicates the good compatibility of a Paillier encryption scheme on an FPGA system.

Our system has achieved most of our goals, though we would have liked to have the decryption working in hardware rather than just simulation.

Our vast utilization of Montgomery for the system allows use to drastically reduce the instances where we need a divider, lowering the cycles per encryption (instead of the lowerbound of 86 seconds with Divider to just 5 seconds using Montgomery). Future optimizations focusing on the multiplier, or particular use cases of the multiplier, are being considered to further lower our Exponentiation, the main bottleneck of our system.

TABLE II
CYCLES PER COMPONENT

Component	***Cycles***	BRAM
4096-bits Adder, Subtractor, Shifter, Selector	∼128	0
4096-bits *grade-school* Multiplier	16,646	1.5
2048-bits *grade-school* Multiplier	4,230	1.5
4096-bits *Parallel grade-school* Multiplier	4,296	3
4096-bits Montgomery Reduction	33,680	4
4096-bits *Parallel* Montgomery Reduction	8,980	7
4096-bits over 2048-bits Modular Exponentiation	102,273,024	12
4096-bits over 2048-bits *Parallel* Modular Exponentiation	25,989,120	21
4096-bits by 2048-bits *magic number* Divider	16,902	2

If we were able to do things differently, we'd like to have payed more careful attention to register-assigns to avoid latches (this resulted in some nasty bugs, seemingly non-deterministic bugs!). We also would've like to have a better sense of the algorithms that we use and the specifics of how each component interacts with the rest of the system before beginning to implement modules. This would've allowed us to create less complex modules that would still be perfectly suitable for our purposes. -

V. Future Work

Currently we have only been able to synthesize the encryption pipeline, though the decryption one should be similar. From here, some future works would include figuring out further ways to restructure the multiplier pipeline to a non-blocking implementation, or using algorithms with higher performance such as Karatsuba's. Finally it would be ideal to have these votes be stored in some sort of permanent storage with some unique identifiers, so our voting system is more robust, and could implement Benaloh's challenge.

[redacted]

VI. Appendix

A. Source Files

All our source files are in
https://github.com/LuisGuille1729/fpga election
It includes:

- **fpga e/** – Encryptor system design & simulation files
- **fpga d/** – Decryptor system design & simulation files
- **python scripts/** – UART transmission and receiving, and prototypes for Paillier, Montgomery, Optimized Multiplications, Dividers.

B. Team Contributions

Everyone

- Diagrams, Write up, Proof Reading Each Other, Debugging
- Encryptor Top Level
- Decryptor Top Level
- Researching 4096-bit Adder, Multiplier, Divider, Modulo implementations, and Encryption Schemes

Rafa

- Montgomery Accumulator
- Multiplier
- Block Repeater
- Byte Repeater
- Padder
- Adder
- Accumulator
- Vote Processor
- Vote Accumulator

Luis

- Optimized multiplier (fsm multiplier.sv)
- Parallel multiplier (fsm multiplier parallel.sv)

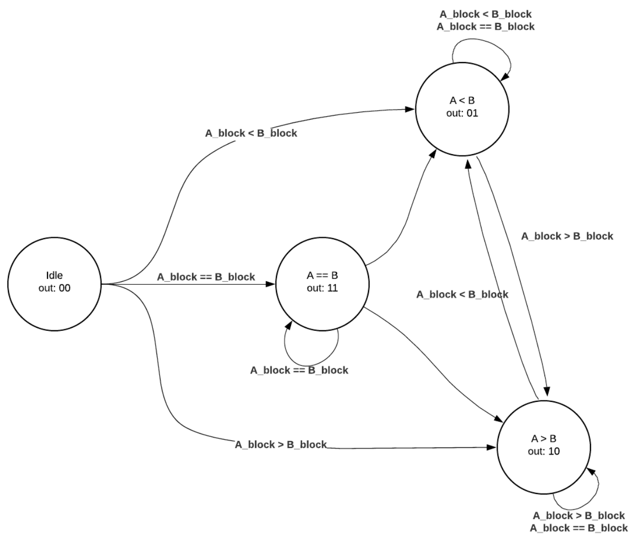

Fig. 7. Running Comparator – Finite State Machine

- montgomery reduce.sv
- Magic Divider (fixed divider.sv)
- UART, SPI communications (spi pe.sv)
- Communication Scripts
- Prototype and Math Scripts (python scripts)

Nico

- Block Repeater
- Montgomery Streamer
- Padder
- Revising Other Modules and Overall Logic

References

[1] Paillier, Pascal (1999). "Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. Advances in Cryptology" — EUROCRYPT 99. Lecture Notes in Computer Science. Vol. 1592. Springer. pp. 223–238. ISBN 978-3-540-65889-4.

[2] Roman N. Kvetny, Yevhenii A. Titarchuk, Volodymyr Y. Kotsiubynskyi, Waldemar Wójcik, and Nursanat Askarova "Partially homomorphic encryption algorithm based on elliptic curves", Proc. SPIE 10808, Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments 2018, 108082H (1 October 2018); https://doi.org/10.1117/12.2501583

[3] Montgomery, Peter (1985). "Modular multiplication without trial", Mathematics of Computation. Vol. 44. American Math Society. pp. 519-521.

[4] Golubin, Artem. "Python internals: Arbitrary-precision integer implementation" rushter.com. https://rushter.com/blog/python-integer-implementation (accessed Nov 27, 2024)

[5] Tobjorn Granlund, Peter Montgomery (1994). "Division by Invariant Integers using Multiplication" https://gmplib.org/ tege/divcnst-pldi94.pdf (accessed Dec 1, 2024)

[6] Ruben van Nieuwpoort."Division by constant unsigned integers". rubenvannieuwpoort.nl. https://rubenvannieuwpoort.nl/posts/division-by-constant-unsigned-integers (accessed Dec 1, 2024)

[7] Weimerskirch, Paar. "Generalizations of the Karatsuba Algorithm for Efficient Implementations". https://eprint.iacr.org/2006/224.pdf