RSA FPGA Implementation Preliminary Report

1st Joseph Kim
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
joekim02@mit.edu

2nd Stephen Campbell
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
sjcr@mit.edu

***Abstract*— We present an implementation of RSA in hardware on an FPGA (Field-Programmable Gate Array), as well as a communications protocol between two FPGAs. This design allows for two computers to establish a secure channel to send and receive messages between each other, without an eavesdropping third party being able to discern the plaintext messages. The system is intended for use as a demonstration in classroom settings, supporting a side-by-side view of encrypted and decrypted media.**

Keywords*— *RSA, cryptography, FPGA, Montgomery Multiplication, Euclidean Algorithm, UART, SPI

I. Introduction (Campbell)

This project uses FPGAs (Field-Programmable Gate Arrays) to facilitate a demonstration of RSA cryptography. Traditional methods of setting up a secure channel between two parties include non-trivial amounts of networking configurations, which can be avoided by using a physical connection between two FGPAs. FPGAs are flexible enough to accommodate a variety of hardware designs and reap the benefits of specialization.

However, using them in the context of RSA cryptography comes with some challenges. RSA cryptography involves numerous mathematical operations that are non trivial to implement on hardware, such as modular exponentiation and inverse modulus. When attempting to expand the RSA bit-depth, it is easy to run out of available space on the FPGA and encounter timing issues with modules that work at a lower size. Another issue is with entering and receiving messages from the FPGA. Though this project uses FPGAs packaged with a dev board, the switches, buttons, and lights available are cumbersome and are limited in their expressibility. For this reason, we elected to use communication protocols to send and receive data via computers, which provide a much more convenient user experience.

Our design set out to meet the following requirements:

1) Implement RSA to some non-trivial bit-depth
2) Allow users to send encrypted data between computers
3) Implementation of at least one of a custom multiplier or a modular exponentiation module

Our implementation is parametrized by the desired RSA bit-depth M. All of the mathematical operations are performed with modules of sufficient size to accommodate numbers of size M. Additionally, the packetization scheme for sending messages between computers and FGPAs used packets of length M. We were able to verify that the modules worked in simulation with sizes up to 512 bits. However, we encountered sizing and timing issues when synthesizing our design for deployment on the FPGA. We were able to synthesize our system with up to 128 bits of depth, but only the system with 64 bits of depth was able to meet timing constraints.

Our design can send messages from FPGAs to computers via the UART (Universal Asynchronous Receiver-Transmitter) protocol and between FPGAs via the SPI (Serial Peripheral Interface) protocol. Though we were able to have FPGAs receive data from computers via UART, persistent issues when scaling this to the packetization scheme prevented us from implementing this in our final design.

We chose to implement modular exponentiation because we believed it would be a more interesting challenge. The operation itself seemed initially simple, but the modulo operation presented a significant obstacle in improving throughput, as the traditional method of division would cause a significant slowdown of the entire messaging pipeline.

II. Top level design (Kim)

The top level design of the project can be seen in figure 1.

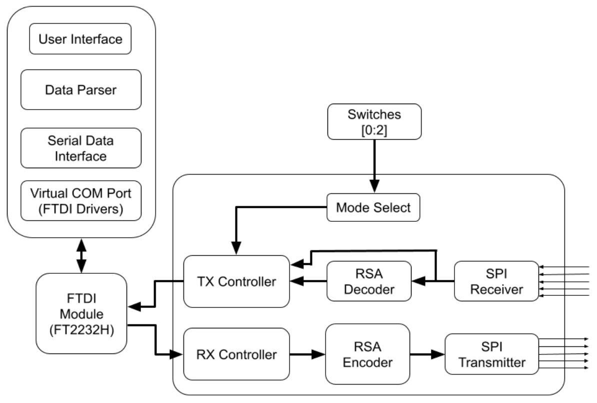


Figure 1: Top level design

The setup of the project will involve 2 laptop computers, each connected to a separate FPGA, which will be the mode in which these devices will send messages back and forth. The FPGA's will have a direct connection and utilize the SPI protocol in order to transmit data between each other, and the messages that are sent will be rendered on one of the screens. The laptops will communicate by utilizing the USB-UART handler and the PySerial library API. The FPGAs themselves will hold an implementation of RSA, an asymmetric cryptosystem, in order to encrypt and decrypt the exchanged messages.

III. Cryptographic engine (Kim)

We selected RSA-512, which utilizes 512-bit modulus in its cryptographic algorithms. While RSA-512 cryptography can be broken quickly and cheaply today [1], larger key sizes were deemed to be too large to fit onto an FPGA, and the goal is to show the feasibility of implementing cryptographic algorithms on FPGAs. If size turns out to be less of an issue than predicted, more secure key sizes may be considered.

An overall view of the cryptographic engine can be seen in figure 2. The engine consists of three parts: key derivation, encryption, and decryption.

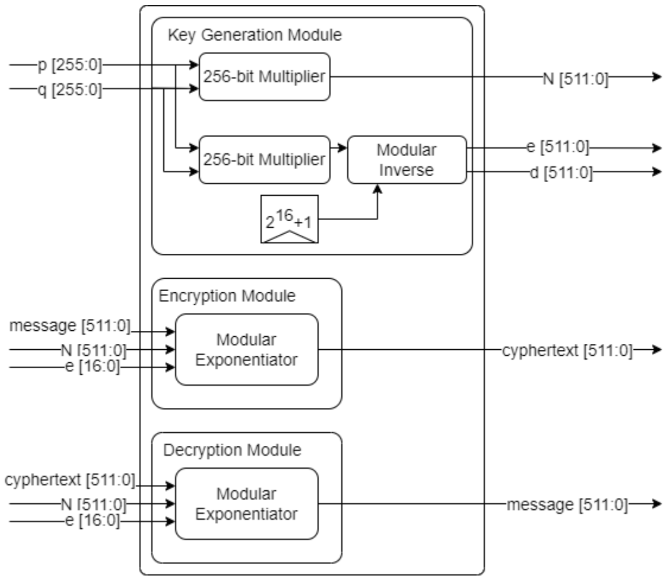


Figure 2: Cryptographic Engine

A. *Key Derivation*

Key derivation of RSA takes two 256 bit prime numbers as inputs, and returns an asymmetric key pair consisting of a public key (N,e) and a private key (N,d). The public key is known to everyone, but the private key is kept secret and is only known by the receiver.

Key derivation for RSA begins with the selection of two 256 bit prime numbers, p and q. These are deterministically chosen for now. However, an implementation for randomly generating prime numbers will be completed in the future. These are then multiplied together using a 256 bit multiplier to produce the modulus N. The numbers (p-1) and (q-1) are also multiplied together in order to derive the private exponent d.

The public exponent e and the private exponent d must be related in that $e * d \equiv 1 \bmod (p-1)(q-1)$. This also implies that e and d must additionally be relatively prime to (p-1)(q-1), otherwise they would not have a modular multiplicative inverse. In practice, the public exponent e is fixed to be $2^{16}+1$, and d is computed by applying the extended euclidean algorithm on the exponent e and the base (p-1)(q-1).

Given that this is only run once during startup, we can choose to sacrifice performance here in favor of a smaller area.

B. *Encryption and Decription*

Encryption and decryption rely on an efficient modular exponentiation module to achieve high throughput. However, reducing intermediate products is extremely expensive as they require an operation akin to a division. As a result, we use Montgomery multiplication [2] to reduce the reliance on division of the modulus and achieve a high throughput, as well as a repeated squares approach for exponentiation to reduce the runtime further.

Montgomery multiplication utilizes a constant R such that R and the modulus N are coprime, and R > N. In practice, R can be set to a power of 2, given that N is a product of two primes. The operation to transform a number x into Montgomery form is defined as

$\bar{x} = x * R \bmod N$.

Addition between numbers in Montgomery form can be performed without change. However, multiplication between two numbers requires the removal of the extra factor of R by Montgomery Reduction. The following algorithm performs this Montgomery Reduction, which can also be used to transform a number in Montgomery form back to its original value:

```
function REDC($\bar{x}$):
        m = (($\bar{x}$ mod R) * N') mod R
        t  = ($\bar{x}$ - m*N) // R
        if t < 0: return t + N, else: return t.
```

This algorithm also requires the derivation of N', which is derived by using the extended euclidean algorithm on N and R to find numbers N', R' such that NN' - RR' = 1 = gcd(N,R). Despite the additional cost from this computation, this runs much faster than the naïve approach to multiplication by utilizing a power of 2 for the constant R, making division operations much faster.

With this, we present the algorithm for modular exponentiation, which uses a repeated squares approach with Montgomery Multiplication as the intermediate multiplications:

```
function mod_exponential(x, e, N)
    // computes $x^e$ mod N
    // x and e are assumed to be bit strings of length n
    running_prod = 1
    current_base = x % N
    for i in range(1, n):
        if e[i] == 1:
            running_prod=REDC(running_prod * current_base)
        current_base=REDC(current_base * current_base)
    return running_prod
```

Encryption of a message m utilizes the recipient's public key pair (N,e), and uses the algorithm above to compute m^e mod N. Decryption of an encrypted message c involves the recipient's private key (N,d), and uses the algorithm above to compute c^d mod N, which is equivalent to the original message.

IV. Communication Protocols (Campbell)

An important goal of the system is the ability to be able to easily use it in a classroom environment. We elected to use computers as the starting point of the data transmission to enable a variety of media to be sent.

The overall data path goes from one computer to an FPGA, then to another FPGA and computer. This system has the advantage of being full-duplex, as each FPGA has its own encoding and decoding modules and independent receive and transmit modules.

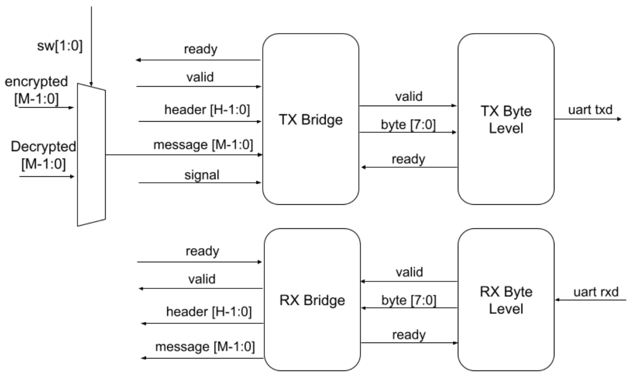


Figure 3: UART Modules

A. Packetization

Packetization accomplishes several objectives: it provides framing and context for the data being sent, it helps avoid overwhelming the FGPA with too much data at once while maintaining throughput, it serves as a convenient way to group data for encryption and decryption, and it provides for data verification. Packetization was achieved via "bridge" modules (inspired by Fischer Moseley's Manta UART implementations), which converted data from packets to bytes.

To meet these goals, the data sent from computer to computer is packetized into M-bit (where M is the message bit-depth) chunks. This size was chosen since it is the maximum size the RSA module can process and thus maximizes the effective throughput of the system. Each is accompanied by a 32-bit header. When data is encrypted or decrypted, the header is preserved and reattached to the processed message.

The header uses the following format, by bits:

0	: data flag
1	: start flag
2	: raw flag
3-6	: reserved
7-14	: transmission ID
15-22	: packet #
23-31	: packet data length - 1

A computer may want to send a transmission that is more than the M bits long. To make this possible, a packet-level protocol is needed. The first step is sending a START packet. This packet is differentiated by having the start flag of its header be 1. Subsequent packets have 0 for their START flag and have the same transmission ID as the START packet. The contents of the START packet include information about the data type, and the total transmission size in bytes and packets, in the following format, by bits:

0-7	: data type
8-15	: reserved
16:	: number of packets

Each packet is encoded by the RSA encryption module and sent to the other FGPA. This FGPA then decrypts it and sends it to the other computer. It can then recover the complete transmission by concatenating the contents of each packet. It can also verify that it received all the packets in the transmission by checking that each packet number less than the number of packets specified in the START packet was received. Using the data type bits in the START packet, it can interpret the data it received and parse it into a known representation.

B. *Computer with FPGA Communication*

Communication between the FPGAs and the computers is achieved through the FPGA's on-board USB-UART handler (FT2232HQ). The FPGAs implement 8N1 UART which the computers receive/transmit via a Python script using the PySerial library API. Switches on the FPGA determine whether the data sent to the computers is decrypted, encrypted, or both at once. For maximum bandwidth, the 'both' option should not be selected. The FT22232HQ supports up to 12MBaud, and therefore the computers and FPGAs use 12MBaud UART communication.

Because the FPGA is subject to memory limitations, it may send a signal to the computer if it is currently unable to accept more incoming packages. Signals are a special kind of packet with no body; they use the same bits as the package headers but have data_flag set to 0 and their transmission_id identifies what kind of signal is sent. The system uses two signals, "stall" and "unstall" to manage data coming into the FPGAs. When the FPGA's pipeline is clogged, it sends a stall signal after finishing its current transmission. Once it becomes unclogged, the FPGA sends an "unstall" signal which the computer recognizes and begins sending data once more.

C. *FPGA with FPGA Communcation*

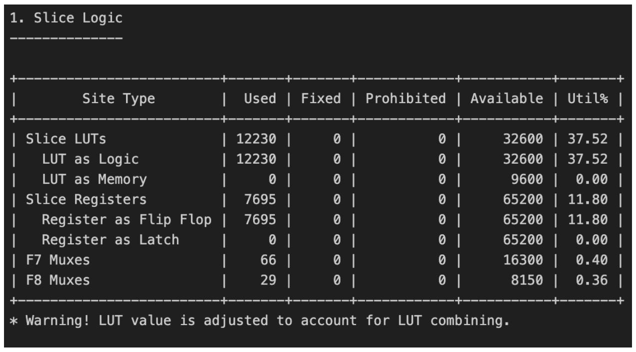
Communication between the FPGA's is achieved via two SPI busses driven at 24MHz to avoid bottlenecking the 12MBaud UART. Every time a packet is sent, the SPI bus's select line is driven low, which allows for easy data framing. Packets are sent bit-wise, starting from the header, from MSB to LSB (Most Significant to Least Significant Bit). This achieves higher throughput than UART, which requires grouping by bytes with start and stop bits. The receiving FPGA is aware of the header and message sizes and stores each as the bits arrive. SPI proved to be much more simple to implement than UART due to its ease of data framing.

IV. Evaluation (Kim)

In this section, we will evaluate our design based on resources utilized and timing metrics, and offer insights into where problems arose and how we overcame them.

Table 1 contains the resource utilization of our design, in terms of Slice logic. Presently, we were only successful in synthesizing our design using key and message sizes of 128 bits. However, we ran into issues with the Vivado software attempting to synthesize large-scale designs. We were successful in synthesizing 128-bit RSA, but increasing this size to 256 caused the software to crash. This could be due to the sheer size of multiple registers and implementations of some of our modules, but the actual cause is unknown. Vivado reported that no DSPs or BRAMs were utilized in our design, which could be changed in the future with better implementations.

Table 1: Utilization of slice logics



```
1. Slice Logic
--------------

+-------------------------+-------+-------+------------+-----------+-------+
|        Site Type        |  Used | Fixed | Prohibited | Available | Util% |
+-------------------------+-------+-------+------------+-----------+-------+
| Slice LUTs              | 12230 |     0 |          0 |     32600 | 37.52 |
|   LUT as Logic          | 12230 |     0 |          0 |     32600 | 37.52 |
|   LUT as Memory         |     0 |     0 |          0 |      9600 |  0.00 |
| Slice Registers         |  7695 |     0 |          0 |     65200 | 11.80 |
|   Register as Flip Flop |  7695 |     0 |          0 |     65200 | 11.80 |
|   Register as Latch     |     0 |     0 |          0 |     65200 |  0.00 |
| F7 Muxes                |    66 |     0 |          0 |     16300 |  0.40 |
| F8 Muxes                |    29 |     0 |          0 |      8150 |  0.36 |
+-------------------------+-------+-------+------------+-----------+-------+
* Warning! LUT value is adjusted to account for LUT combining.
```

Our timing requirements utilized a 100 MHz clock, and we were almost able to meet the timing requirement with 64 bit depth; however, the modular inverse computation seemed to have a negative slack after synthesis, and we were unable to determine the cause of this. However, given that the modular inverse was only utilized once during startup, we deemed this to be a minor issue. Scaling the design to 128 bits, however, additionally introduced negative slack in our decryption module. However, the rest of the modules were able to meet timing requirements. The current problem, however, is that implementing faster algorithms and modules will likely incur an area cost that could be problematic when synthesizing the entire design. As a result, we had to optimize for cheaper modules for area, at the cost of better latency and throughput

Currently, our design has met the minimum requirements of being able to send and receive messages, as well as being able to encrypt and decrypt these messages to be displayed on a laptop, albeit with some minor issues. Meeting the ideal goals would take much more effort as we discuss in the next section. With minimal changes, we would be able to emulate a small conversation between the two boards, as well as fix the underlying issues regarding negative slack and some resource problems.

V. Further developments (Kim)

Here, we discuss how our implementation could be further modified, given additional time, to account for various factors and use cases.

One issue with our current design is space limitations. The size of the modules, due to the large size of the various quantities, prevented the design from being scaled to quantities larger than 128 bits. Trying to fit this requirement would require more research on different algorithms to conserve area, at the cost of speed, or fundamental changes to our top level design. Utilization of more space in BRAM may also alleviate space issues, as the various values could be stored in there while not in use, reducing the need for large sized registers. Another possible way to save space would be to modify the top level design to reuse certain modules, such as reusing the modular exponent module to both encrypt and decrypt messages.

Another future development would be the generation of random primes for key generation. Currently, our design is very simplified, and the primes are deterministically selected for each board; however, this is obviously insecure as recycling keys in this manner makes the attacker's job easier in determining the private key of either party. This also means that once a private key is recovered, messages from previous sessions can also be decrypted. Nondeterministically generating primes for key generation solves this issue; however, due to the complex algorithms that generate random primes, this work would likely have to be done on the laptops and sent to the boards using the communications protocol. This would require minor modifications to the setup phase of our top level design to receive these randomly generated primes.

Our stretch goals would require major changes and refactoring of the top level to accommodate. One of our goals was to attempt to implement a symmetric key encryption scheme, which would also require a key exchange protocol to negotiate a shared secret between the two parties. The simplest method would be the Diffie-Hellman key exchange, which can be implemented with changes to the communications pipeline as well as the reutilization of our modular exponent function. However, symmetric encryption algorithms such as AES would require further research and experimentation to be implemented. Sending audio and video was also another of these goals, which would require additional modules and functionality to be implemented.

VI. Retrospective

This project has given us much insight into the world of digital design. It highlighted many subtle issues that arise when attempting to write implementations that will eventually be flashed onto a physical board, and it is a very different experience compared to writing the same thing in software. This section aims to detail some of these revelations that we encountered.

- Because debugging hardware can be a frustrating task to do, utilizing simulations can give valuable insight into the behavior of a specific system or module. However, even the simulation has its limits, as we discovered especially in debugging our various cryptography modules. Verilog can surprisingly handle incredibly large numbers to use in its simulations, as it had no complaints when we tested our modules with 512 bit numbers in some places. However, even that has its limits, as discerning outputs in gtkwave was a horrible experience because the software had a lot of problems in displaying these 512 bit numbers.

- Area proved to be a significant problem, as some of the operations within modules were attempting to multiply two extremely large numbers within a single clock cycle. This would, of course, explode our resource utilization, and we had many instances where the build of our design attempted to utilize more resources than were physically present on the FPGAs, which would obviously not build. The experience would have been a lot smoother had we continuously reminded ourselves to keep this idea in mind as we designed the modules.

- The AXI protocol of managing FSMs was extremely helpful in making sure each module began working when they were supposed to. Many computations were often stalled by other computations; for example, computing a modular inverse required the repeated use of a divider to obtain the quotient and remainder, and utilizing AXI made the logic of waiting for the divider to finish much simpler. If given the opportunity to redo this project, we would continue to use this protocol given its familiarity, though there may be better suited protocols.

VII. Acknowledgements & Contributions

The workload was split between the two major components and based on each person's familiarity with each. Joseph handled the cryptographic engine and its related modules, while Stephen worked on communications, including the UART and SPI protocol and modules.

During the development of the low level UART modules, Fischer Moseley's Manta UART implementations were referenced [3]. We would also like to thank Joe Steinmeyer and the rest of the 6.2050 staff for making the class fun and interesting, as well as helping debug various components of the project.

V. Source Code

The repository containing all code and testbenches can be found at https://github.com/Jierark/EncryptED.

VI. References

[1] D. Goodin, "Breaking 512-bit RSA with Amazon EC2 is a cinch. so why all the weak keys?," Ars Technica, https://arstechnica.com/information-technology/2015/10/breaking-512-bit-rsa-with-amazon-ec2-is-a-cinch-so-why-all-the-weak-keys/#:~:text=The%20cost%20and%20time%20required,even%20computing%20novices%20can%20follow. (accessed Nov. 22, 2023).

[2] P. L. Montgomery, "Modular Multiplication Without Trial Division," Mathematics of Computation, vol. 44, no. 170, pp. 519–521, Apr. 1985. Accessed: Nov. 22, 2023. [Online]. Available: https://www.ams.org/journals/mcom/1985-44-170/S0025-5718-1985-0777282-X/S0025-5718-1985-0777282-X.pdf

[3] F. Moseley, "Manta: An In-Situ Debugging Tool for Programmable Hardware," Manta Documentation, https://fischermoseley.github.io/manta/ (accessed Nov. 22, 2023).