Final Project Report: Digital Electronics - RSA Key Generation, Encryption & Decryption.

Matt Kenney and Jake Epstein

ABSTRACT

The aim of this project was to build a working RSA Cryptosystem using the VHDL hardware description language and to implement our system on the Basys3 FPGA Board. Cryptosystems are sets of cryptographic algorithms used for keeping digital communications confidential and verifying the authenticity of the parties involved in such communications. In RSA, three processes serve to satisfy the requirements of the cryptosystem. Firstly, the key generation process creates a unique set of numbers called the “key-pair” which are then used to carry out the core RSA operations. The RSA Encryption/ Decryption process facilitates secure communication channels between two parties, allowing each sender to encrypt messages such that they are only recoverable by the intended recipient. The RSA signature process satisfies the last obligation of the cryptosystem, providing a means to both create and audit digital signatures to enforce integrity in digital communications.

For the purposes of our project, we choose to implement the RSA key generation process and encryption/ decryption process. The function of the FPGA was to act as a dedicated cryptographic module that could be used as companion to a traditional microprocessor device—the microprocessor, using serial communication, could make calls to the FPGA to generate keys, encrypt a message, or decrypt a message. We aimed to take advantage of the specialized nature of the FPGA to perform cryptographic operations at higher speeds and throughput than a traditional processor.

The culmination of the project resulted in code that produced mathematically correct working key pairs and decrypted and encrypted properly in simulation. In hardware, however, the FPGA only generated the public key correctly, due to an error yet to be identified. The failure to generate an accurate private key prevented the FPGA from properly engaging in the encryption/decryption process in hardware, as the private key is needed to perform such operations accurately.

INTRODUCTION

The rise of blockchain technology and cryptocurrencies have given cryptography a significant media presence in recent years. Despite its associations with these newly developing technologies, however, cryptography plays broad and crucial roles in all forms of digital communication. Cryptography is, at the core, the study and practice of techniques that enforce security and fidelity in digital communications[1]. Cryptosystems are the functional unit of cryptography and represent a set of cryptographic algorithms that work together to create secure communications and verify the identity of the parties involved in these communications. The Rivest-Shamir-Adleman (RSA) cryptosystem, one of the oldest in use today (patented in 1983), and the first public-key encryption system ever invented, continues to remain extremely secure and widely used despite its age. To provide just a few examples of RSA's prominence today, RSA is the most commonly used cryptosystem used to implement Secure Sockets Layer (the 'S' in HTTPS) which prevents adversaries from snooping in on users' interactions with websites (as they enter bank credentials, for example)[2]. It is also the mostly widely used algorithm to produce digital signatures[3]. Digital signing and signature auditing are used as a means of verifying the identity of a digital entity. Signatures are used to prevent forged transaction statements, as a much stronger alternative to password-protected entry, and more. Finally, RSA is deemed secure up to the confidential level by the US Government[4] and is thus used to create extremely secure signatures and to encrypt digital documentation with confidential status. In short, RSA represents a crucial algorithm in maintaining secure communications today.

---

[1] Van Leeuwen, Jan, and Jan Leeuwen, eds. Handbook of theoretical computer science. Vol. 1. Elsevier, 1990.
[2] "What Is SSL (Secure Sockets Layer)?" DigiCert. Accessed June 05, 2019. https://www.digicert.com/ssl/.
[3] Katz, Jonathan, Alfred J. Menezes, Paul C. Van Oorschot, and Scott A. Vanstone. Handbook of applied cryptography. CRC press, 1996.
[4] Gallagher, P. "Federal information processing standards publication digital signature standard (DSS)." Fips pub 186-3 (2009).

Despite its advantages, working with RSA represents an extraordinary computational burden, as the cryptographic algorithms it employs require expensive computations. Further, as the processing speed of available hardware continues to rise and algorithms for cracking cryptosystems improve, users of RSA must adopt more rigorous standards for employing RSA to maintain its security over time[5]. Finally, cryptography experts now forecast that quantum computers will be able to break classical cryptographic algorithms as they are currently implemented[6], further increasing the need for security improvements (and thus computational power). Due to these considerations, researchers have begun constructing cryptographic modules on FPGAs and other embedded processors to enhance the speed and throughput of cryptographic operations in RSA and other cryptosystems

Following in the footsteps of these researchers, we aimed to produce a cryptographic FPGA chip using the Basys 3 FPGA and corresponding VHDL code. For this project, we decided to implement RSA key generation and encryption/decryption only. Since the final project code is intended for demonstration rather than full functional use, we opted to avoid implementing signature operations. Encryption and signature operations are identical operations (they merely use different operands), so we felt that we could adequately demonstrate the functionality of the system with encryption only. The function of the FPGA was to act as a dedicated cryptographic module that could be used as companion to a traditional microprocessor device—the microprocessor, using serial communication, could make calls to the FPGA to generate keys, encrypt a message, or decrypt a message.

[5] Barker, Elaine, and Quynh Dang. "NIST Special Publication 800-57 Part 1, Revision 4." NIST, Tech. Rep (2016).
[6] Chen, Lily, Lily Chen, Stephen Jordan, Yi-Kai Liu, Dustin Moody, Rene Peralta, Ray Perlner, and Daniel Smith-Tone. Report on post-quantum cryptography. US Department of Commerce, National Institute of Standards and Technology, 2016.

OVERVIEW OF RSA CRYPTOSYSTEM:

Symmetric Cryptosystems:

Prior to the invention of RSA and other public-key (or "asymmetric") cryptosystems that followed, secure communications were primarily carried out using *symmetric* cryptosystems. Symmetric cryptosystems use just one "key", a special type of operand used to perform cryptographic operations. Under symmetric cryptosystems, this single key performs all encryption/ decryption operations – For two entities to engage in secure communications with one another, they must each possess that same single key. Entity 1 encrypts a message using the key it shares with Entity 2, then Entity 2 decrypts the message using that same key. Security-wise, there is nothing inherently flawed about symmetric cryptosystems like this one; however, key distribution poses a significant problem. For two or more entities to begin engaging in secure communications in the first place, they must design a way to share the key with one another without an adversary obtaining the key. Without having a secure communication channel to begin with, distributing these keys over the internet without exposing them to adversaries can become exceedingly difficult.

Asymmetric (Public Key) Cryptosystems:

Public Key Cryptosystems solve the problem of key distribution by introducing the concept of a public-private key pair. In symmetric cryptosystems, keys are tied to a group of entities wishing to communicate with each other on secure channels. In public-key cryptosystems, instead, each *key-pair* is tied to a single entity on the internet (a person, company, etc.), and is comprised of a public key and a private key. The public key represents the entity's public identity and is visible to all other entities on the internet. Conversely, the private key is a tool to be held by the entity to which it belongs and kept secret. Because this form of cryptosystem is structured in such a way in which one of the keys can be made visible to adversaries without compromising security, it foregoes the key-distribution problem entirely.

In the RSA Encryption Process, an entity creates a secure communication by encrypting a message they would like to send using the recipient's public key. The recipient can then apply its private key to the encrypted message it received to recover the original message. The encryption is performed such that only the holder of the private key could possibly uncover the message.

The RSA Signature process is identical to the encryption process except that the role of the public and private key is reversed. In the RSA Signature Process, entities use their private keys to "sign" (i.e. encrypt) digital messages and transaction statements they would like to issue to the public. They attach the original message, the "signature" (the encrypted message), and their public key when they want to issue a statement. Members of the public use the signing entity's corresponding public key to audit the validity of the signature—the sender really is who they claim to be if the attached "signature" decrypts to the original message. Alternatively, if the signature decrypts to something other than the original message, the signature is a forgery and the message must not have come from the attached public key identity.

RSA OPERATIONS

Key Generation: Key generation represents the most complex computation performed in RSA, although it is not typically the most computationally expensive. This is the process that generates a public and private key pair which can be used to carried out encryption/ decryption processes described above.

From a high-level perspective, the key generation algorithm generates a pair of keys which perform reverse operations—a message encrypted using the public key can be restored to original form using the private key, and a message encrypted using the private key can be restored to original form using the public key. Further, the key generation algorithm must also ensure that the private key is extremely difficult to derive even when the public key is known. The technical process is as follows:

Key Size: Key size is the number of bits found in each component of the keys. A larger key size increases the size of the message that can be encrypted and decreases the chances that a private key could be cracked but increases computational expense.

Public Key(e,n): The public key is composed of two key size numbers. In the encryption process, e is used as an exponent operand while n is used as the mod operand. See below for details.

Private Key(d): The private key is composed of just 1 number: d. d is also used as an exponent operand in the decryption process. See below for details.

* The below explanation is based on the explanation from the handbook of applied cryptography

Key Generation Steps:

1. Pick two large random prime integers p & q, each around half of key size in bits.
2. Compute public key component n: $n = p * q$.
3. Compute Euler's Totient $\phi = (p - 1) * (q - 1)$.
4. Pick a random integer $1 < e < \phi$ that is *coprime* to ϕ, i.e. $\gcd(e,\phi) = 1$
5. Compute the multiplicative inverse of e modulo $\phi(n)$. This is private key `d`
    a. *Multiplicative inverses a, b are numbers which yield a*b = 1. Thus, multiplicative inverses "mod x" a, b, are numbers such that (a*b) mod x = 1 mod x. In terms of the keys, (e * d) mod ϕ = 1 mod ϕ*

Encryption/ Decryption: The encryption and decryption operations are both simple modular exponentiation functions (a^b modulo x), differing only in the operands of the function. The operands include the message being sent, the ciphertext (encrypted message) being received, and the components of the public/ private keys.

Encryption: Entity B encrypts message m to be delivered to Entity A

1. Entity B obtains A's authentic public key(n,e)
2. Message (m) is represented in binary, and must be smaller than key component n
3. Ciphertext $C = (M^e) \bmod n$
4. Entity B sends Ciphertext C to Entity A.

Decryption:

1. Entity A uses its private key d to recover the message: $m = (c^{d}) \bmod n$

The operation of each of these three functions was implemented in the VHDL code generated through this project. The implementation is described in detail in the Theory of Operation subleaders of this report under the Design Solution Section.

DESIGN SOLUTION

SPECIFICATIONS

Our design solution was intended to allow a user of the device to interface with its cryptographic capabilities via SCI. On the board, we provided options for the user to request for the board to generate a public/private key pair, and options for the user to send in a message to be encrypted and/or decrypted by the board. Note that the top-level circuit designed on the board is merely meant to demonstrate the functionality of the code, and thus is not intended for a user attempting to leverage the cryptographic operations of the underlying modules for legitimate purposes. Further note that, because the hardware behavior of our code does not meet the intended behavior (or the behavior observed in simulation), the specifications described below reflect the intended behavior rather than the actual.

OPERATING INSTRUCTIONS:

*See Appendix AB for the front panel view of the board. All boldface terms below are defined on this Appendix.

1. To interface with the FPGA, plug in the PMOD RS232X part to the top row of pins on the JA interface (**SCI_connectors**) on the FPGA.
2. Using the appropriate cable for SCI, attach the PMOD part via USB-A, or via another port which can be used for serial communication

3. Open a serial terminal program. Set the program to use the following defaults: 115200 baud rate, 8 bits, no flow control, 1 stop bit, no parity, etc.
4. Press the **key_press** button to produce a key, wait for the **key_finished** LED to light up, and observe the first three most significant Hex values of public key component (**n_MSBs**) on the First 7-segment display
5. *After* generating a key (and only after), open a serial terminal, and type 4 characters on the keyboard. The FPGA will then encrypt the message you sent. After finishing the encryption process it should send back the resulting ciphertext to you via SCI. Note, however, that the encrypted version may not be representable in ASCII letter value codes and thus might not display. The FPGA will next perform the decryption operation to decrypt the ciphertext it generated using the private key in the key pair—the result of this decryption should be the original message text. At this step, if the message received in the Serial Terminal is the same as the message you originally sent, key generation, encryption, and decryption have all operated properly.

Top Level Theory of Operation:

Key Generation: The key generation process is prerequisite to the encryption or decryption processes and is triggered by a single press to the **key_press** button on the FPGA board. Key size is preset to 32 bits in the top-level file[7]. The key generation process produces a public key, **e** and **n**, and corresponding

[7] NOTE: The key generation module, treated as a component within our RSA top level file, was written using VHDL generic types to define the key size, and is therefore capable of producing keys of any length the user specifies. Because large keys produce extremely long simulation times when employed to carry out the encryption or decryption process, however, we hard-coded the key size to be set to 32 bits in the top-level file. Key generation took only about .1 or .2 seconds using 64- or 128-bit keys, however, subsequent encryption/ decryption processes regularly took several *seconds* of simulation time when run with 128 or 64-bit keys, due to slowdowns discussed under the

private key **d**. After the key generation process is complete, the FPGA flicks the **keygen_finsihed** LED, signaling the termination of the process, and displays the first 3 digits of public key value "n" on the 7-segment MUX display. After the successful termination of this process, the user is open to send a message to the device to be encrypted/ decrypted.

Encryption / Decryption: After the successful termination of the key generation process, the top-level module hosted on the FPGA begins listening for SCI input from the microcontroller to which it is attached. The SCI communications are received and sent through a PMOD interface on the FPGA, and our serial receiver and serial transmitter components facilitate the serial communication process. Because the key size chosen for this application is 32 bits, the maximum size of the message to be encrypted is 4 characters of text[8].

To begin the encryption/ decryption process, the user of the device opens a serial terminal and sends four characters of text to the FPGA. As the FPGA counts the number of characters it has received via SCI, displaying the counter on the leftmost digit of the 7-segment display, as it shifts these characters into an internal "message" register. The 7-segment display allows the user the to verify that their communications are being received correctly. After receiving all 4 characters of information, the device first attempts to encrypt the received 4-character message, running modular exponentiation to compute the value of the resulting ciphertext (Ciphertext C = M^e mod n). After performing encryption, the FPGA

"justification and evaluation" sections under the discussion on the modular exponentiation and modulus components of the project. A 32-bit key size hardly approaches the standard 2048-bit key size used in most practical applications, but this key size allowed us to test the behavior or all our circuits within a reasonable time frame in simulation and served our purposes well.

[8] Standard ASCII characters are represented by 8-bits, but always have a leading 0 in the Most Significant Bit position. RSA Enforces that the binary-value of the message to be encrypted must be less than the binary value of public key component n. For an n of key size 32, 4 characters of ASCII will always satisfy this requirement: Although 4 * 8 = 32, the MSB of the message will always be 0. Effectively, the ASCII value of a 4-character message is always 31 bits or less, and thus always less than the value of n.

sends the encrypted results back to the user via serial communication, using the serial transmitter to convey these results. The FPGA then decrypts the resulting ciphertext ($M = C^d \bmod n$) to recover the original message. Next, it sends the decrypted result back via serial communications to the user. If the user observes her original message was sent back, she has verified that the encryption and decryption processes operated properly, and that the key generation process produced a valid key-pair.

Construction and Debugging: The top-level life was designed by stitching three main components of the project together – key generation, the encryption/ decryption process, and SCI. Key generation represented the most complex operation of the bunch, involving several thousands of lines of code and including nearly all sub-modules that we constructed throughout the project. Encryption/ Decryption, conversely, involved only a single modular exponentiator component – one of the simplest/ lowest level components of the project. See Appendix H for more details about how all of the components wire-up in keygen and in RSA-top. Aside from determining how each of the three main components of the project would fit together, the designing of the top-level file involved the construction of an appropriate user-interface and decisions about how the user could interact with the module.

Debugging the top-level portion of the project represented a significant challenge. In simulation, we were able to trigger the RSA key generation process and produce a mathematically correct RSA public/ private key pair. We are also able to appropriately load in a message from simulated SCI. However, once the corresponding message was loaded, we lacked sufficient hardware resources to compute the result of the encryption or decryption of the message. Encryption involved the modular exponentiation process of raising the 31-bit message to a key up to 32 bits, and mudding by another 32-bit value n ($\text{cipher} = \text{msg}^e \bmod n$). Despite the performance design of our modular exponentiator, the simulation time lasted several seconds as we waited for the modular exponentiator to finish, causing multiple computers which we attempted to run the simulations on to run out of available hard-disk space and forcing simulations to stop.

Although we could have reduced the key size to see whether encryption or decryption would have resolved under a lower computational requirement, we felt this to be an unnecessary step – modular exponentiation components are littered throughout the key generation module and essential to the generation of a proper public/ private key pair. We did not feel the need to test the simple operation of the modular exponentiator component – the component required for encryption/ decryption, because we had already verified its capability to carry out its operations correctly when test benched as an independent entity, and had further verified its capability by producing a correct public/ private key pair. We instead focused our efforts on debugging the hardware behavior of the top level.

In order to obtain as much information as we could when testing the behavior of the FPGA, we first port mapped signals corresponding to each state in the RSA-top state machine to LED lights on the board. This allowed us to see whether the FPGA was getting "stuck" in one state or another. We were especially interested in debugging problems that resulted in the top-level state machine getting stuck in the generic "hold" state while it waited for other components to finish another task. We additionally port mapped the signals "keygen_ready, encrypt_start, and decrypt_start" to LEDs so we could see some of the underlying activity of the data path. Finally, we port mapped the counter that counts the number of characters received via SCI to the leftmost digit of the 7-segment display, and the 3 hex values representing the 12 MSBs of public key component n, to the 3 remaining digits on the 7-segment display.

After bit streaming the code, we pressed the gen_key button on the FPGA and found the first three hex values of n being displayed were B, 3, A. Converting the decimal value of n found in Appendix I of 3014598856 to hex yields B3AF20C9, indicating that the value for N found on the FPGA board was correct, and aligned with our simulation (we used the same random number seeds in each case, so N should have come out to the same value, as it did). Strangely, however, when we ran the same code, instead port mapping key components e or d to the 7-segment-display, key generation never finished, leaving the rsa-top level state machine stuck in the hold state and never receiving a done-tick from the keygen module.

We were unable to diagnose the reason for this problem. It may have been that both e and d were calculated incorrectly and represented metastable or unreadable signals. Alternatively, e and d may have been computed properly all along but cannot be read in some cases due to fan-out or fan-in problems. We believe that the problem likely had to do with an error preventing e and d from being properly read after being output from the keygen module. Whether that error arose from within or without of keygen is unclear.

We removed the mappings of e (or d) to the 7-segment display and replaced the mappings back to n in order to enable the board to escape the key generation process once again. This enabled us to test the board's encryption/ decryption capabilities over SCI. After pushing the keygen button to load a key-pair, and after observing that the correct n appeared on the 7-segment display, we began explaining the serial communications counter. Upon examining the message counter for serial communications, information appeared to be going into the FPGA via SCI properly—the counter ticked once for each character entered into the serial terminal on the computer connected to the FPGA. Once 4 characters were read in via SCI, however, the RSA-top state machine went to the encrypt state and again found itself stuck in hold, waiting for the modular exponentiator to finish. The modexp component, however, never finished, leaving RSA-top stuck in the hold state indefinitely. This failure to finish indicates once again that e or d may be unreadable for some undiscovered reason. Since e and d act as the exponents used in the encryption and decryption process, respectively, if they were not readable, it would make sense that the modexp component never finishes computing (since it had nothing to work with in the first place).

Because n is computed correctly in hardware, we know that, at the very least, the components contained in keygen pertaining to the calculation of n must be working properly (see project hierarchy in appendix H for details). The computation of n requires two large random prime numbers, p & q (n = p * q), so keygen calls upon a module called pqgen to obtain such numbers. Pqgen includes a random number generator and a Rabin-miller prime checking component – both must be used to compute random primes p & q. Moving down the levels of logic, the Rabin-miller prime checker requires a modular exponentiator component (the same one used in encryption/ decryption), and a random number generator. Both components are used extensively (i.e. at least 14 times for 14 different operations for the generation of p and q).[9] Finally, the modular exponentiation component uses or lowest-level component, the modulus. All of these components are necessary for the generation of p & q.

Because SCI signals were also being read correctly, based on the above assessment, we can conclude that all the components used in our design must have been performing correctly in hardware apart from one. Because we seemingly cannot read the value of d or e, we cannot know whether the extended GCD component works correctly. Extended GCD takes e and phi of n as input and calculates private key d as output. In conclusion, although the vast majority of our components bit streamed correctly, the inoperability of key components e and d on the FPGA board prevent anything other than the calculation of n form working properly.

Justification of Design: Although the design failed in hardware, we believe that the chosen design would have served its purpose well to demonstrate the key generation process and subsequent encryption/ decryption process over SCI. One major design shortcoming, however, was the way in which we set "seeds" used to generate random numbers. Seeds determine the chain of pseudo-random numbers that the random-number generators will produce. In their current state, we have left the seeds hard-

[9] The Rabin-Miller algorithm probabilistic prime checker we employed attempts to disprove the primacy of a number 7 times before declaring it prime. It uses random number generator and modular exponentiation operations on each attempt. Thus, if the random numbers pqgen tries for p & q happen to be prime on the first go, Rabin-Miller will still undergo 14 attempts. In the (much more likely) case that the random numbers tested for primacy are not prime on the first go, Rabin-Miller will keep being called until a prime p and q are discovered.

coded directly on the RSA top-level code file. This is poor design practice, because using the same seeds results in the generation of the same public/ private key pair. If we instead asked users to press random keys on their keyboards and read in those randomly entered ASCII values as the value of the random seeds to be used in key generation, our module would generate truly unique keys on each iteration. See more information on seeds under the description of the random number generator module below (LFSR).

MODULES COMPRISING THE TOP LEVEL:

The Modulus Module:

Specifications: The modulus's (mod's) functional purpose is fairly self-explanatory. It finds the remainder of a given division operation. For instance, *19 mod 9* = 1, because *19* divided by *9* is 2 remainder 1. However, if the divisor n (in this case 19) is greater than the dividend x, the result of the modulus function (the modulo) is the dividend. In other words, *9 mod 19 = 9.* For use in RSA key generation, the modulus function must also be capable of computing the modulus of a function with a negative dividend and positive divisor. An example of this is -3 mod 2 = 1. To create a universal modulus function that could be used for all cases, we derived that for a given function $-x \bmod n = n - (x \bmod n)$. By preprocessing all negative divisors into their positive equivalents, we avoided overcomplicated logic. The building of a modulus module was necessary as a key component of RSA encryption, decryption, and key generation is modular multiplication and exponentiation. Additionally, Xilinx's built in modulus function is only synthesizable for dividends of the power of 2 according to their manual. Our modulus function takes inputs clk, dividend a_in, divisor b_bin, and enable bit new_data. It outputs a done signal, the quotient (q_out) and the result of the modulo operation, the remainder, r_out.

Theory of Operation: More detailed documentation, the State diagram and block level schematic of the modulus can be found in Appendix P and Appendix S, but the following is a brief explanation of the logic. When the modulus receives a high new_data signal, it loads the inputs a_in (the dividend) and b_in (the divisor) into registers and enables computation of their modulo. A comparator checks if a < b, the end condition, at every rising clock edge. When a > b, a subtractor subtracts a from b on every rising clock edge until the end condition is met. A counter increments q at every subtraction, computing the quotient, the number of times b goes into a until it cannot anymore. At the end condition, the done signal is asserted as decremented a and incremented q are moved into output registers, becoming r_out and q_out respectively.

Construction and Debugging: The modulus function is fairly straightforward, and just had some minor timing issues with the done tick that were promptly able to resolve through simulation. The simulation waveform is attached in Appendix C.

Justification and Evaluation: Our modulo function takes a very conservative approach to modulus calculations. Although iterative subtraction on large numbers is certainly slow, this was somewhat intentional. The modulus component is arguable the most vital piece of the entire project. Nearly every process in RSA encryption, decryption, and key generation requires it. All of our upper level components instantiate it as a necessary component. We knew that a faulty modulus would result in completely faulty RSA, so we intended to make our modulus function as foolproof as possible. We also assumed that making a fast, synthesizable modulus function would be difficult as even Xilinx's mod function only synthesizes with powers of 2. That being said, multiple simulations have indicated that the modulus function is a significant performance bottle-neck. If we had to do it again, we would probably implement a simple bisection algorithm, where it picks a random multiple of a and subtracts it from b. If a > b, take half the multiple and subtract again. If a < b, take half the multiple and add it back. Do this iteratively until you are within a margin of a from b.

The Modular Exponentiation Module

Specifications: Modular Exponentiation is a very important and widely used function in RSA and cryptography as a whole. For this project, the function is used in Rabin-Miller's primality test, and directly in encryption and decryption, to compute $C = (M^{e}) \bmod n$ and $m = (c^{d}) \bmod n$. More specifically, modular exponentiation involves calculating the remainder of an integer x, raised to a power y, and divided by a positive integer n. As described in the Extended GCD section, calculation of the modular multiplicative inverse, in other words taking y in modular exponentiation to be a negative number, is considered relatively simple to compute. On the other hand, brute force modular exponentiation with positive coefficients is considered to be incredibly slow. Thus, modular exponentiation's one-way functional behavior makes it useful in cryptography. Our Modular Exponentiation module takes inputs clock, enable, x, y, and p, where the equation being calculated is x^y mod p. Its outputs are mod_exp, the solution to a given modular exponentiation problem and a done signal that's asserted when the module is finished.

Theory of Operation: Brute forcing modular exponentiation, as previously mentioned is incredibly resource intensive. While certainly possible on modern computers, the method still requires O(x^y) time to complete. There are a number of algorithms that were invented to improve the runtime efficiency of modular exponentiation. Some of the more common methods include the Right-to-left binary method, which uses exponentiation by squaring, Reduction via Modular arithmetic, and Montgomery's algorithm. For the purposes of this project, we eventually settled on Reduction via Modular arithmetic, which reduces runtime to O(log y). We did spend two days however trying to implement Mongomery's algorithm, without much success, to be further discussed in the Justification and Evaluation section. The fundamental idea behind Reduction via Modular arithmetic is the modular distributive property. When given an equation (ab) mod p = ((a mod p) (b mod p)) mod p. One modular multiplication can be factored and divided into two separate operations. When used iteratively, it can be used to reduce exponentiation into a series of multiplications. The C code snippet bellow courtesy of geeksforgeeks exemplifies this[10].

```
int power(int x, unsigned int y, int p)
{
   int res = 1;      // Initialize result

   x = x % p;  // Update x if it is more than or
             // equal to p

   while (y > 0)
   {
      // If y is odd, multiply x with result
      if (y & 1)
         res = (res*x) % p;

      // y must be even now
      y = y>>1; // y = y/2
      x = (x*x) % p;
   }
   return res;
}
```

If the exponent is even, it can be separated into two separate mods, so divide the exponent in half and square the argument, as seen with the above properties. If the exponent is odd, multiply x with result. In practice, the VHDL code follows this algorithm fairly closely. First, it loads the inputted x, y, and p values into registers and sets register res to 1. Next, it enables the modulus module to accept input. It sends in x % p and assign the output back to x. It

[10] "Modular Exponentiation (Power in Modular Arithmetic)." GeeksforGeeks. September 24, 2018. Accessed June 04, 2019. https://www.geeksforgeeks.org/modular-exponentiation-power-in-modular-arithmetic/.

then checks if the LSB of y is a 1 (odd), and if so, it multiplies by res * x. Then it multiplies by p. Then it assigns the result back to res. It checks if the LSB of y is not a 1 (even). If not, it multiplies x*x. Lastly it put it all back through the modulus p and iterates back through the FSM until y<= 0.

Construction and Debugging: Constructing the modular exponentiation unit was extremely challenging. We spent a long time focusing our efforts on constructing our algorithm using what most literature said was the fastest on FPGA's, Montgomery's algorithm. Without going into too much detail, Montgomery's performs a change of base operation to transpose modular exponentiation into "Montgomery's Domain" where modular exponentiation transforms into a series of multiplications and additions. While the algorithm was incredibly interesting to learn about, it turned into a massive time sink for us. It proved to be very difficult to implement in VHDL, and we spent many hours debugging code we failed to ever properly run. Eventually, we were advised to migrate to a new algorithm. We settled on Reduction via Modular Arithmetic as it did not require large divisions or multiplications and still had a run time complexity of O(log y). While this algorithm was easier than Montgomery's, we still struggled to implement it in VHDL as we hit numerous timing issues. Some of the larger multiplications required more than one clock cycle to complete. We ultimately sequentialized many of the operations, and essentially created an atomic module. If we had to do it over again, we would likely skip over spending so much time on Montgomery's Algorithm. It would have been better invested elsewhere.

Justification and Evaluation: While the algorithm we use for modular exponentiation has runtime complexity of O(log y), our struggle to satisfy timing constraints resulted in a sequential machine that performs one or two operations per clock cycle. This likely negatively impacts speed and performance. Additionally, while our solution was far easier to implement in VHDL than Montgomery's or other alternatives, these more sophisticated algorithms are slightly faster and more memory efficient. This worked in both simulation and hardware.

<u>The Linear Finite Shift Register Module</u>

Specifications: This project uses a linear finite shift register to a generate a stream of pseudo-random numbers for a number of modules needed to perform RSA key generation. As computers are entirely predictable machines, and cannot generate truly random numbers, the linear finite shift register and other pseudorandom number generators are commonly used sources of randomness within cryptographic systems. Encryption as a whole relies on sophisticated pseudorandom number generators to ensure there is no distinguishable pattern that may be abused by a bad actor. In the context of RSA encryption, random number generation is necessary when generating potential primes and performing Rabin-Miller's Primality to ensure such numbers have a high probability of primality. This process is part of the generation of *P* & *Q*. Additionally, random numbers are used in the generation of public key component *e*. While the linear finite shift register is neither secure nor sophisticated, it is a simple and easily implemented source of pseudo randomness. As the LFSR's input bit is a linear function of its previous state, and there are only a finite number of states for a given register of a fixed size and input seed, the LFSR repeats a given string. However, a well-designed function of reasonably large bit size can still generate a fairly long stream of random values. Our specific LSFR takes input clk, an enable/disable bit, a seed value of generic *num_bits* size, and a seed enable bit. It outputs a continuous sequence of pseudo random numbers as *data* of *num-bits* size until the enable/disable bit is toggled or it reaches the end of a given random stream. The later triggers a *data_done* mono-pulse, indicating the usable random number stream has ended, and is beginning to repeat.

Theory of Operation: The LFSR uses a single process and no state machine but a block diagram can be found in Appendix AA. As an LFSR takes input based on a linear function of previous states, it can generate a string of random values no greater that *2^N-1,* where N is the size of the seed. Our specific LFSR follows Xilinx's handbook recommendations[11]. At the rising edge of every clock, the LFSR checks if the enable/disable bit is set to high. If it is, it checks if the seed is being reset (the *seed_en* is toggled and a seed is waiting in seed). If the seed is being reset the data register gets the given seed value, otherwise it left shifts the data register value with a one-bit output XNOR function of bits within the register. Xilinx recommends "tapping" different locations within the register depending on its size. They provide a convenient lookup table of the most effective tapping locations for register sizes from 3 to 168. As our LFSR needs to generate random numbers of multiple different bit lengths, it takes a generic size. In order to take advantage of the significant improvements in performance when implementing bit length specific tapping schemes, we programmed Xilinx's tables, from 3 bits up to 64 bits, in a series of generate statements, as recommended by Surf VHDL[12]. On every clock cycle, the data register is compared to the seed, and if they are equivalent and it is not the first instance of seed (not immediately following its initial input), the *data_done* tick is asserted.

Construction and Debugging: The LFSR was debugged in simulation, and there were no noticeable bugs at first. However, when instantiating the component into another module, we noticed one frustrating issue that made it challenging to use. Done was asserted on the first clock cycle, as a new seed was instantiated. A start bit to insure done would never assert itself to the first seed resolved this problem. We're also not particularly fond of how we coded the seed input. Seed and seed_en must be asserted in

[11] Alfke, Peter. "Efficient shift registers, LFSR counters, and long pseudo-random sequence generators." http://www. xilinx. com/bvdocs/appnotes/xapp052. pdf (1998).

[12] Surf-VHDL. "How to Implement an LFSR in VHDL - Surf-VHDL." Surf. December 10, 2017. Accessed June 04, 2019. https://surf-vhdl.com/how-to-implement-an-lfsr-in-vhdl/.

parallel for a seed to be inserted, and this was occasionally challenging to time properly. We might change that if we were to did it over again, but overall no major changes.

Justification and Evaluation: This module is the VHDL standard for pseudo number generation. It's easy to code and implement and generates reasonably sized strings of random output. That being said, it is neither particularly secure nor sophisticated. Anybody referencing Xilinx's LFSR tables could emulate our design, and with the same seed, end up with the exact same outputs. If they took the first couple primes from the sequence, they would also likely know out *p* and *q*. This works in both simulation and hardware.

Extended GCD Module

Specification: The Extended Euclidean algorithm, or the extended greatest common denominator algorithm(extgcd), is an iterative algorithm that computes both the GCD(a,b)**,** and the coefficients of Bézout's identity of two univariate polynomials, x & y. Coefficients x and y are integers that satisfy the equation **ax + by = GCD(a,b)**[13]. The additional computation expense accrued by calculating these coefficients is negligible, making this algorithm a good choice for determining GCD in the case that these coefficients might prove helpful. The primary benefit to the RSA Cryptosystem that the extgcd algorithm produces arises from a property of the algorithm when its inputs, a and b, are coprime. Under this condition, x is the multiplicative inverse "mod b" of a, and y is the multiplicative inverse "mod a" of b. In other words: ((**x mod b) * a) mod b = 1 mod b**, and **((y mod a) * b) mod a = 1 mod a**. This property is extremely useful in RSA for the generation of key components e and d. Because e must be a random number that is coprime to ϕ, we continually test for a valid e by selecting random values for e and

[13] Extended Euclidean Algorithm. Accessed June 05, 2019. http://www-math.ucdenver.edu/~wcherowi/courses/m5410/exeucalg.html.

inputting ɸ and e into the extgcd algorithm. When the resulting gcd equals 1, we know that we have found a valid public key "e." Conveniently, however, we are also immediately able to calculate the private key "d" by using the coefficients of Bézout's identity. If **$x\phi + ye = \gcd(\phi, e) = 1$**, we can find **$d = y \bmod \phi$**[14]**.** This will yield a d that will effectively "undo" the effect of e in the RSA encryption/decryption process. In short, extgcd is the core component responsible for the calculation of key values e and d. The proof of the assignment of d is listed below and was derived by hand.

Proof:

Proof:	$x\phi + ye = 1$	* *e and ϕ are coprime*
	$= (x\phi + ye) \bmod \phi = \mathbf{1 \bmod \phi}$	* *mod each side by ϕ*
	$= (x\phi \bmod \phi) + (ye \bmod \phi)$* *distribute mod*	
	$= (ye \bmod \phi)$	* $x\phi \bmod \phi = 0$
	$= ((y \bmod \phi)\,(e \bmod \phi)) \bmod \phi$	* *Apply distributive mod property*
	$= (y \bmod \phi * e) \bmod \phi$	* *Since $e < \phi$ as a req. of RSA, $e \bmod \phi = e$*

Since $d * e \bmod \phi = 1 \bmod \phi$,
$d = y \bmod \phi$

Theory of Operation

Reference Code: As a means of better understanding the states and state transitions undertaken in this algorithm, consider the pseudocode excerpt from Dartmouth's CS30 course notes on the extended GCD Algorithm:

```
0 Procedure EXTGCD(a,b)
1 > Returns the GCD of a and b. Also returns x, y, such that xa + yb = gcd(a,b)
2 Divide a by b to get a = bq + r
3 if r = 0 then:
4         return (b,0,1) – gcd, x, y
5 else:
6         Let (g, x', y') = EXTGCD(b,r)
7         Return (g, y', x' – y'q)
```

[14] Katz, Jonathan, Alfred J. Menezes, Paul C. Van Oorschot, and Scott A. Vanstone. Handbook of applied cryptography. CRC press, 1996

The insight that gives rise to the Euclidean GCD algorithm is that the GCD of two numbers does not change if the larger of the two numbers is replaced by its difference with the smaller number. In the Euclidean algorithm, this process subtracts the value of the smaller number from the value of the larger number at each step, eventually reducing the two numbers to an equal amount. That equal amount is the GCD. The above code, using Euler's insight, reduces the GCD(a,b) to GCD(b, a mod b), thus reducing the value of the larger number at each recursive step. The proof behind this insight will be foregone here but note that this approach represents one of the more computationally efficient ways to compute GCD of large. The extended portions of the algorithm—the calculation of x and y, merely involve a series of flips ((x,y) <= (y,x)) and subtractions. The recursive algorithm above was adapted into an iterative one before being written in VHDL. The core operations, however (flip-flopping of y and x, deduction of y*q), remain consistent with the above code.

The state diagram and block diagram describing the extgcd module can be viewed under Appendix Q and Appendix T. The logic of extgcd is as follows:

Extgcd remains in a resting state (nop) until the enable bit, "new_data," goes high. When this occurs, extgcd loads data in and resets relevant registers/ logic signals before moving to the **divide** state. In this state, extgcd makes a simple call to the modulus component to compute the value of a mod n. After the mod finishes, throwing a **mod_finished** signal, quotient q and remainder r are retrieved from the mod machine and the next state is set to **check.** In the check state, extgcd determines whether it should iterate or terminate (if a mod b = 0). The check is equivalent to line 3 in the above pseudocode. If r fails the check, **iterate_en** is set to 1 and the next state get **update_xy**. When the **iterate_en** bit is thrown, a is set to b, and b is set to remainder r. This process satisfies the same obligation as line 6 of the above pseudocode, effectively moving from searching for gcd(a,b) to equivalent gcb(b,r). The extended portion of the algorithm is performed almost identically in the iterative method when compared to the above recursive method, only slight changes in this calculation are made. The **update_xy** state simply updates the values of x and y before sending next state to divide once again. The states cycle between divide, hold, and check until r = 0, at which point GCD gets b (line 4 above) and output is triggered. Note that, because either x or y can exhibit a negative value, we used signed values of x and y throughout the extgcd algorithm, and merely cast these signed values to type STD_LOGIC_VECTOR before output.

Construction and Debugging:

Before realizing that the recursive algorithm above could be implemented in an iterative fashion, we foolishly attempted to build a similar recursive algorithm in VHDL. To make this process possible, we even attempted to use a stack memory to hold the values of variables uncovered at the deeper phases of recursion, just as a computer would use stack memory to hold each "layer" of computation. To make matters worse, we even derived an algorithm to determine the maximum possible number of layers of stack memory required to find the gcd of two numbers a and b, based on the value of larger value a. Interestingly, we found and proved that the maximum number of possible recursions that could take place during the course of the extgcd algorithm was equal to the index (in the Fibonacci sequence) of the smallest Fibonacci number larger than a. Nevertheless, once we discovered a means to implement the algorithm iteratively, we had a much easier time, resulting in code that was much simpler and cleaner. The Debugging process for this module was relatively straightforward. It merely involved verifying that the GCD output of the algorithm was correct and verifying that xa + yb = 1.

Justification and Evaluation: The extgcd algorithm represents a crucial component of our design and is implemented in such a way that its computational cost is about as low as possible. Its usage in the keygen—in which d is computed immediately after e is found – represents a streamlined and effective way of generating key components e and d. Overall, this component represented one of the more successful components of our code.

Rabin-Miller Primality Test:

Specifications: The Rabin- Miller primality test is used to ensure that operands p and q (used to compute n) are prime. In order for the key to remain secure, p and q must be prime, or ϕ could be easily compromised. If ϕ is uncovered, the secret key d can be cracked by finding the multiplicative inverse mod ϕ of e. A primality test is a probabilistic algorithm that determines whether a given number is prime. Because primality tests are not deterministic, they cannot be expected to accurately assess the primality of a number with 100% certainty. For assessing the primality of large primes (very high bit counts), however, employing deterministic prime-checking algorithms becomes computationally infeasible, and the benefits of using a probabilistic algorithm outweigh the costs[15]. The Rabin-Miller prime test is used to compute probabilistic primes by using a series of "attempts" to disprove the primality of the number it is operating on. If, during one of these attempts, the compositeness of the number is uncovered, Rabin-Miller states that if n is not prime. Alternatively, if none of the attempts succeeds in disproving the primality of the number under testing, the number is declared as prime. The formula to calculate the probability of Rabin-Miller passing a non-prime as prime is $(1/4)^{\text{attempts}}$. According to the the prevailing opinion in the field, anumber of tries of 7 represents the most secure number of attempts to be employed by Rabin-Miller[16], with no additional benefit gained beyond this number. For this reason, we built a Rabin-Miller algorithm defaulted to 7 attempts to disprove primality. The prime_test entity takes an *odd* number as input and outputs a Boolean value "prime."

[15] Katz, Jonathan, Alfred J. Menezes, Paul C. Van Oorschot, and Scott A. Vanstone. Handbook of applied cryptography. CRC press, 1996.
[16] Park, Heejin, Sang Kil Park, Ki-Ryong Kwon, and Dong Kyue Kim. "Probabilistic analyses on finding optimal combinations of primality tests in real applications." In International Conference on Information Security Practice and Experience, pp. 74-84. Springer, Berlin, Heidelberg, 2005.

Theory of Operation: The Rabin-Miller algorithm was built by transferring the below python script to VHDL. All states in the Miller-Rabin entity simply represent operations in this python code and involve a significant amount of looping. The behavior of the code is better understood in a language such as python. The ideological basis behind the Miller-Rabin algorithm is beyond the scope of this report, as it involves delving deeply into other mathematical proofs such as Fermat's Primality Test and a discussion on other concepts regarding pseudoprime numbers. Nevertheless, note that the Miller-Rabin algorithm makes use of the random generator (to generate a normal distribution of operators to run its tests with), and a modular exponentiator component. It interfaces with these components in the block-level diagram.

```
# http://inventwithpython.com/hacking (BSD Licensed)

 import
 random


            def rabinMiller(num):
              # Returns True if num is a prime number.

              if num % 2 == 0:
                return False;

              s = num - 1
              t = 0
              while s % 2 == 0:
                # keep halving s while it is even (and use t
                # to count how many times we halve s)
                s = s // 2
                t += 1

              for trials in range(5): # try to falsify num's primality 5 times
                a = random.randrange(2, num - 1)
                v = pow(a, s, num)

                if v != 1: # this test does not apply if v is 1.
```

```
        i = 0
        while v != (num - 1):
            if i == t - 1:
                return False
            else:
                i = i + 1
                v = (v ** 2) % num
    return True
```

Construction/ Debugging: Because of the complexity of this algorithm, the construction of the prime tester was primarily performed by reconciling the above python code to a VHDL state machine and datapath. Testing of the prime tester was relatively straightforward—we simply input values which we knew to be prime, and values we knew to be composite, and observed the prime signal the prime tester issued during its output state. In all the cases we tested, the prime tester correctly assessed the primacy of the number. The chance of failure of this algorithm is only about 1/16384, so the chance we observed such a failure in simulation was negligible.

Justification and Evaluation: Although the code worked properly, the state machine result of this code was not particularly clean. Reconciling code from Python to VHDL resulted in a complex state machine that performed little to no operations in parallel. On the other hand, the component worked correctly in simulation, and is one of the components we can certify working in hardware. The intent was a working design rather than an efficient one; however, if we had additional time to improve the design, we'd have attempted to clean up the state machine or designed the algorithm from scratch in VHDL so as to make the algorithm more native to the language.

The PQ Generator Module

Specifications: This module generates two random prime numbers p and q. These two random primes are used to calculate n = pq and ɸ= (p – 1) *(q – 1). For RSA, generating a random pair of prime seeds is vital to the cryptographic value and security of RSA. If an adversary had a high probability of guessing p and q (meaning they're not random), they could easily calculate ɸ. As they already know public key component e, they can compute e modulo ɸ(n) to find private key d, breaking the RSA system. To generate the two random numbers, The pq Generator module generates random numbers using the LFSR and tests if they are prime using Rabin Miller's primality test. To enforce the key size, the pq generator outputs p and q as the first two randomly generated primes whose most significant bit is '1'. Our pq generator takes inputs clock, enable, and a seed value for the random number generator. It outputs p, q, and a done signal.

Theory of Operation: Upon initialization, the pq generator takes a seed and sends it to the LFSR random number generator. To ensure that the seed for Rabin-Miller and the LFSR generating random potential primes are not identical, the seed of Rabin-Miller always gets the inputted seed + 3. As the LFSR generates numbers, the FSM watches the done signal, resetting the seed to a function of a previous seed value if it is asserted. A random number from the LFSR is fed into a rand_num register. The FSM checks if the value is odd, a necessary specification for test primes sent to Rabin-Miller. It also checks if the most significant bit is a one, as this is a RSA condition. P and q must have a one as the most significant bit to ensure that n = pq is of the promised keysize. If these conditions are met, the number is fed into Rabin-Miller for prime number testing. If the number is in fact a prime, it is outputted as p and the process is repeated for q. If any conditions are not met along the way, a new random number is retrieved and tried again.

Justification and Evaluation: This is one way to execute pq generation that is widely accepted amongst industry experts. However, we've seen some people encourage the use of flexible length keys that sum to the desired key size. Our pq module generates a p or q for a given number only if the number has a most significant bit of 1. While this works, it limits the potential set of valid primes significantly, making the value less random and easier to infer. By using flexible key lengths, if you are looking for a 32 bit key you could generate a 24 bit p and an 8 bit q and still possibly have a valid prime, widening the set of valid primes and making p and q more random. We might try to implement something like this if we were to do this project again.

Construction and Debugging: We debugged using simulation. There were no significant problems to note. We have verified that pq generation works in hardware and generates valid primes.

Key Generation Module

Specifications: The purpose of the key generation module (entity keygen) is to generate a valid public/private key pair for use in RSA encryption and decryption. It takes as input an enable bit, and two numbers of key size termed "seeds." One seed has bits equal to key size, while the other has a number of bits equal to half the key size. These seeds are used as inputs into the various random number generators the keygen requires to function properly. The keygen module caries out all steps identified in the key generation process under the "overview of RSA cryptosystems" section of the Introduction in order to generate a public/private key-pair. Its outputs include a mono-pulse done signal to indicate when is finished computing the key-pair. As the done signal is pulsed, the keygen component outputs the two components comprising the public key, e and n, and corresponding private key d.

Theory of Operation: The state diagram and block diagram describing the keygen module can be viewed under Appendix L and Appendix Z. The logic of keygen is as follows:

Keygen remains in a resting state (nop) until called upon by another module with the enable bit input. Once enabled, keygen loads the random seeds to be passed to the Linear Finite Shift Register (LFSR) components (random number generators) that it uses in the process of key generation. Next, it moves moves to the gen_pq state. In this state, it calls upon the pqgen module to generate the large primes p & q required for computing the rest of the key. Pqgen uses an LFSR to generate the randomness it needs to pick two random primes, and thus requires a seed to pass to the LFSR. LSFR seeds must be the same bit size as the bit size of the random numbers to be generated. Accordingly, keygen passes pqgen the smaller of the two seeds it possesses[17].

After waiting for pqgen in a generic hold state, keygen reaches the compute_n state and multiplies the outputs of pqgen to compute $n = p * q$ and $\phi = (p - 1) *(q - 1)$. Next, keygen loops back and forth between the states try_n and test_e until it finds a public key value e that satisfies the requirement that e is a random integer $1 < e < \phi$ that is *coprime* to ϕ, i.e. $\gcd(e,\phi) = 1$. In the try_e state, keygen simply generates random numbers with a bit length of key size. In the test_e state, keygen first tests if e is greater than 1 and less than ϕ. If e indeed satisfies these criteria, keygen enables the Extended GCD component (extgcd) to check if **gcd(e, ϕ) = 1**. E and ϕ are port-mapped to the extgcd component as the two inputs to the algorithm. When the extgcd algorithm concludes, keygen stops waiting in the hold state and moves back to the test_e state, this time checking whether the GCD outputted by the extgcd component equaled 1. If it did not, the try_e, test_e loop continues in the same fashion. If **gcd(e, ϕ)** did come out as 1, however, keygen has found a valid e and moves on to perform the calculation of d.

[17] P and Q, in our implementation, each get key size/ 2 bits. Setting them as such ensures that their product, n, is guaranteed to have key_size bits (no 0 in the MSB). Pqgen, under this implementation, needs to generate random numbers of key_size/ 2 bits. Because the bit length of the seed passed to the LSFR component must equal the size of the random numbers to be generated, pqgen therefore needs to be passed a seed of size key_size / 2.

As explained in the proof under the description of the extgcd component, when ϕ and e are inputs to the extgcd algorithm, the extgcd output "y" can be used to compute the value of private key d. In all cases, **d = y mod ϕ.** Therefore, after an appropriate value "e" has been uncovered using the extgcd algorithm, d can be computed immediately. Many cases, however, exist where the value of y is negative. Because our modulus function cannot appropriately perform calculations using signed numbers or compute the value of a negative number mod a positive one, in the check_y_sign state, keygen performs a simple reconciliation to ensure the modulus component performs as intended. (-x mod n) is equivalent to n – (x mod n) (see proof below). Leveraging this property of a modulus calculation with a negative dividend, keygen flips the value of y before passing it to the modulus component if y happens to be negative. Later, in the compute_d state, keygen subtracts the value produced by the modulus from ϕ**.** In brief, this reconciliation allows the proper calculation of a negative y mod ϕ. In the case that y is positive, y is passed to the modulus normally in the check_y_sign state and d gets the result of that modulus operation in the compute_d state. The done tick is finally issued, and the key generation is complete in the output state that follows.

Proof: *(-x mod n) = n – (x mod n)*
(-x mod n) mod n = (n – (x mod n)) mod n * *mod each side by n*
-x mod n mod n = (n mod n) – (x mod n mod n) * *distribute mod operations*
-x mod n = - x mod n * *double mods reduce to single mods; n mod n gets 0*

Construction and Debugging: The keygen module performs very few logic operations of its own – it primarily relies on the operations of other prebuilt modules to calculate each key component, n, e, and d. Building keygen, therefore, simply involved the stitching together of other modules. In simulation, keygen produced an accurate Public(e,n)/ Private(d) key pair. We verified the accuracy of these keys by ensuring that (e*d) mod ϕ = 1 mod ϕ = 1. We performed further verification by testing a random message m, and ensuring that $(msg^e \text{ mod } n)^d$ mod n = msg. Wolfram Alpha was used as a powerful calculator to test these results.

In hardware, it's unclear how much of keygen worked as expected. The key generation process clearly finished in hardware, as evidenced by a "keygen_done" tick that lights up an LED on the board. Further, n was properly computed. We checked this by port mapping several hex values of n to the 7-segment display and verifying that the n-value displayed on hardware matched the corresponding one found in simulation. Strangely, when we port-mapped values of e or d to the 7-segment display, the code no longer behaved as expected and the key generation process no longer finished. This may have been due to a problem in key generation in generating the values of e and/or d, or it may have been due to some other undefined problem, such as too-much fan out/ out problem having to do with the e or d registers. Despite several days of debugging, we were unable to resolve this issue.

Justification and Evaluation: The key generation module is a relatively straightforward process, comprised of several steps which take place serially, one at a time. We believe the design was satisfactory yet are puzzled by the potential problems it may undergo when bit-streamed to hardware.

The SerialRx Module

Specifications: The SerialRx module is an SCI interface that communicates from a computer via an SCI cable to PMODRS232 board connector. This module follows the specifications from the old lab 5. We use SerialRx, at 115,200 Baud, no parity, one stop bit, no flow control. Our SerialRx module has inputs clock, RsRx (serial input), rx_data (parallel output), and rx_done_tick. Input from the computer enters the board via a serial cable to the upper row of JA via RsRx. The SerialRx interprets the signal appropriately, and outputs it in parallel via rx_data. For RSA specifically, Rx_data proceeds through RSA encryption/decryption logic and is outputted via SerialTx back serially to the computer. The RSA encryption/decryption logic accepts four letters due to the small cryptographic key size. The SCI client we used for this project is SCREEN.

Theory of Operation: SerialRx and serial source communicate via a predetermined baud rate of 115200. SeriaRx waits for serial input RsRx to go low upon initialization. Low RsRx indicates the start of a transmission. It also initializes the counter c_count. Counting to c_count = N/2, where N is the clock frequency/baud rate, marks the center of the first data bit. It also enables s_count and resets c_count. S_count increments every N count and shifts the RsRx value into a shift register (can also think about it as counting shifts). When s_count = "1010", when 10 values have been shifted in, the output register gets the middle 8 bits and the done tick is asserted.

Construction and Debugging: We based our design on the Old Lab 5 specifications. We experienced some minor issues with the done-tick, but some simulation testing and minor debugging fixed it.

Justification and Evaluation: This is a simple design that directly follows the old lab 5. The only potential significant source of improvement is likely increasing the baud rate so SCI has a faster sample rate and happens faster. Additionally, the lab should use terminal package SCREEN rather the PuTTy. We had many problems with PuTTy and SCREEN always worked flawlessly. This worked in both simulation and hardware.

CONCLUSIONS

The goal of this project was to build a full RSA cryptosystem to be implemented on an FPGA, capable of key generation and encryption/ decryption processes. As of current, our aims were matched in simulation, but fell short in hardware. The simulations verifying the accuracy of key generation (Apendex

I) and the simulations verifying the capabilities of the modular exponentiator collectively validate the system's functionality as a working cryptosystem. On the other hand, although we were able to verify that all components aside from the extended gcd component synthesized and bit streamed properly (see discussion on construction and debugging of RSA Top level), our code was only capable of demonstrating partial key generation. Because of the problems we experienced with using key components e and d in the RSA top-level file, encryption and decryption of messages sent via SCI additionally could not be performed on hardware.

Overall, although the failure of the project to work on hardware was disappointing, it appeared that the majority of our project modules – pq_generator, prime_tester, modexp, LSFR (random number generator), and mod – all worked on the board. The remaining problems on the board could have been caused by errors in keygen that caused portions of the key generation to fail, errors in extgcd, or miscellaneous errors due to the wiring of the components, fan-in or fan-out errors, or others. While the errors caused problems with the overall operability of the board, having such a large portion of components synthesize correctly still represents a major success.

At the time of the original project proposal, we drastically underestimated the difficulty of implementing an RSA cryptosystem. The primary difficulty in this problem came in trying to work with the constrained resources and capabilities of the FPGA and the VHDL coding language. Having access to only simple operations such as addition, multiplication, and subtraction, the .vhd files we used to implement our algorithms were often 10 or 20-fold the size of their java/python counterparts. The working code in the project at the time of writing includes 4277 lines of *current* working VHDL code (design files and testbenches), and 1405 lines of code that has been discarded and abandoned.

The extraordinary time commitment of writing and debugging this code resulted in both authors of this report spending approx. 100 hours per person researching algorithms, writing and debugging code, and preparing this report. The time commitment to just generating this quantity of code reduced our ability to debug the final top-level circuit. Therefore, although we found our project to be extremely enriching and satisfying, we would recommend that groups choose projects with more feasible timeframes; or, alternatively, choose projects which can be easily "scaled back" if they become overbearing.

In terms of general coding and design advice, we noticed that our designs benefited substantially and that our work progressed much more quickly when made sure we had very thoroughly researched the algorithms we were attempting to build before starting to code. Further, we found that building testbenches that grow alongside the design file (testing each new feature iteratively, every time a feature is added), was the most effective way to debug. As we learned to adopt this method over the course of the project, we became substantially more efficient. Finally, we recommend that groups begin focusing on function right away—focus on getting a working product first, rather than making it efficient or adding features. Even when there's plenty of time left in the project, focus your efforts on function over form.

Acknowledgements

We'd like to acknowledge and thank Professor Luke and Dave Picard for their term long effort in teaching us all about digital electronics and VHDL. We really enjoyed the class!

Both of us contributed equal time to the project and worked well on it for the past couple weeks. We both contributed equally on coding/debugging. Matt focused on Extended GCD, Key Generation, RSA Top, Modulus, and Rabin-Miller. Jake focused on Modular Exponentiation, SCI, LFSR, and pq generation. For this paper, Jake focused on the block diagrams, waveforms, and state diagrams. Matt focused on the writing, but Jake also contributed.

REFERENCES

Alfke, Peter. "Efficient shift registers, LFSR counters, and long pseudo-random sequence generators." http://www. xilinx. com/bvdocs/appnotes/xapp052. pdf (1998).

Barker, Elaine, and Quynh Dang. "NIST Special Publication 800-57 Part 1, Revision 4." NIST, Tech. Rep (2016).

Chen, Lily, Lily Chen, Stephen Jordan, Yi-Kai Liu, Dustin Moody, Rene Peralta, Ray Perlner, and Daniel Smith-Tone. Report on post-quantum cryptography. US Department of Commerce, National Institute of Standards and Technology, 2016.

Gallagher, P. "Federal information processing standards publication digital signature standard (DSS)." Fips pub 186-3 (2009).

Katz, Jonathan, Alfred J. Menezes, Paul C. Van Oorschot, and Scott A. Vanstone. Handbook of applied cryptography. CRC press, 1996.

Park, Heejin, Sang Kil Park, Ki-Ryong Kwon, and Dong Kyue Kim. "Probabilistic analyses on finding optimal combinations of primality tests in real applications." In International Conference on Information Security Practice and Experience, pp. 74-84. Springer, Berlin, Heidelberg, 2005.

"Modular Exponentiation (Power in Modular Arithmetic)." GeeksforGeeks. September 24, 2018. Accessed June 04, 2019. https://www.geeksforgeeks.org/modular-exponentiation-power-in-modular-arithmetic/.

Van Leeuwen, Jan, and Jan Leeuwen, eds. Handbook of theoretical computer science. Vol. 1. Elsevier, 1990.

"What Is SSL (Secure Sockets Layer)?" DigiCert. Accessed June 05, 2019. https://www.digicert.com/ssl/.