RSA Hardware & Software Implementation on Zynq Board

Noah Evantash, [Date]

EECE4632 HW-SW Design FPGA, Prof. Leeser

# Introduction:

The core of this project is the RSA algorithm, named after its inventors Ron Rivest, Adi Shamir, and Leonard Adleman in 1978 [1]. RSA is one of the most prominent encryption algorithms, and is used widely within the industry for encrypted communications, embedded RSA-enabled chips, and data security. The popularity of RSA is credited to the algorithms’ security and inability to be decrypted through iterative attempts. (Find source regarding the amount of time it takes for normal computer to crack RSA). RSA is so secure because it uses a complex formula with a pair of large prime numbers that make it difficult to reverse the value. However, the algorithm can be decrypted by knowing the prime values and the public key used to encrypt the original data. Due to the usefulness of this encryption and the potential to use iterative code, this algorithm was selected to be implemented on HW using a Zynq Pynq board.

# RSA Overview:

The RSA algorithm inputs use a public key, private key, and prime numbers, to encrypt and decrypt an input data stream. There are two sides to this algorithm: encryption and decryption. For the encryption, a value will be streamed into the function and will return an encrypted value. This is done by computing the equation:

Where *M* is the original data, *C* is the encrypted data, and *e* and *n* are computed with the prime numbers. The value *n* is equal to the product of the prime numbers *p* and *q*. The public key, *e* is equal to a value coprime and less than the totient (*t*) value that is the product of (*p*-1) \* (*q*-1). The primary inputs to this encryptor are therefore *p* and *q* for the prime number, *e* for the public key, and most importantly the data to be encrypted, *C*.

On the other side, the decryption is performed by computing the equation:

Where *d* is the only new variable representing the private key. The value *n* needs to be the same in the encryption and decryption for this to work since *p* and *q* need to be the same. The value *d* is calculated from the public key, *e*, using the identity:

Where *e* is the public key and t is the totient. Therefore, the only needed inputs to find *d* are *e* and *t*, which are both calculated from the prime numbers when generating the public key.

For this to work, there first needs to be calculations that create a value for *e* and *d* based on the prime values. For the public key, *e,* this can be done by checking if the value is coprime with *t* by seeing if the greatest common denominator is 1. The private key can be checked by looping the equation *e* \* *d* mod *t* for different values of d until the result is 1. Once these keys are discovered and checked, the function can encrypt an input value.

Therefore, there are really 4 iterations of functions occurring to encrypt and decrypt these values. The first function is to establish and check the public key with the input prime numbers. Once checked, the encryption function takes place with inputs *p, q, e,* and *M* (the original data). Next a private key needs to be developed and checked by finding the smallest value for *d* that satisfies the identity. The last function is therefore decryption, which takes the encrypted data *C,* the prime numbers *p* and *q* and the private key *d* to determine the decrypted message.

Diagram

Description automatically generated

# PS Implementation:

The first iteration of creating the RSA algorithm was built out using Python on the Processing system for the Pynq board. This iteration was simple and borrowed heavily from the source code available for introducing the RSA algorithm [x]. It began with evaluating for the public key using the greatest common denominator function to evaluate the totient and prime numbers to get the key. A value is then encrypted in the function “RSA\_Encryptor” by using the convenient “pow(a, e, m)” function on Python which evaluates “ae mod (m)”. The returned value becomes the encrypted cipher.

The next part of the code uses the inverted modulo function to evaluate the private key given the public key and primes. The inverted modulo function is evaluated using the function “pow(pubkey, -1, phi)” where pubkey is the public key and phi is the totient. With the private key, the decryption of the cipher is performed by evaluating pow(cipher, private\_key, n). The returned value is the decrypted cipher and should return the original value. However, due to the limitations of the integer value in Python, this iteration had limitations caused by the prime key and cipher data sizes. As a result, the integer value would be converted to a float and would therefore be inaccurate when evaluating the modulo.

The next iteration evaluated used a larger integer value for the prime values by generating a value with 512 bits. However, to do this, the bits had to be generated randomly and checked if they were prime through factorization. As a result, the delay of this iteration was significant at roughly 8 seconds. On the other hand, this iteration was more capable of larger values being encrypted/decrypted. This allowed it to encrypt strings rather than small integer values.

In the next iteration, the functionality of encrypting a string was kept, but the random generation of the string was altered. Instead, the string was broken down into an array by the ASCII value of the characters and each value was encrypted in the same manner. As a result, smaller values were guaranteed to be encrypted, and the resulting array of encrypted values could be decrypted and converted back to a string.

The last implementation on the PS was considered the best since it was capable of handling string values without taking an exceeding time to compute. Furthermore, it makes for a perfect implementation to develop an AXI stream overlay for in the PL implementation.

# PL Implementation:

For the PL implementation, the first step was to build the encrypting code on Vitis HLS and test the code. The early implementation did not use a function to generate the keys like the Python code, rather it used a static value for the prime numbers, public key, and private key. The values used for the keys were verified with the PS implementation, and furthermore allowed verification with the resulting encrypted value.

Unlike the Python implementation, the HLS code did not have a convenient method of evaluating aemod(n). The right-to-left binary method of evaluating modular exponentials was the best choice to use since it has reduced memory footprint and operations [x]. The following figure shows the code implemented to compute the value of the encrypted result.

Text

Description automatically generated

Figure 1: Right-to-left binary evaluation of modular exponent

To implement AXI stream to encrypt an array of values, this while loop in figure 1 is embedded in a while loop that reads the data from the stream. This while loop operates until the value the last value is sent, notified by the plain.last Boolean. In the while loop, each value is written to the dataOut stream after being encrypted. The results of this were tested with a testbench file and verified with the values from the PS implementation.

Lastly, this simple RSA implementation was tested on the Pynq board using an overlay. The overlay utilized the same setup as the PS implementation; however, the array of integers was inputted to the input buffer for the stream. To do this, the function runKernel has an input for the string message, where it allocates the input buffer and output buffer from the length of the message. From there, the message has each character converted into an integer and input into the buffer as an array. The buffer is then transmitted to the PL through the DMA where the values are encrypted. The output buffer then receives the output values as an array, which should be encrypted.

The next iteration of the PL implementation built off this last example by add the key generator function. To do this, much like the PS key generator, a function was required for the greatest common denominator, the co-prime checker, and the inverse modular function. The GCD function and co-prime checker were simple and used the same logic as the PS. However, much like the exponential modular, this was not readily available in Hardware. To perform this calculation, the extended Euclidian algorithm was used, which returns the private key value.

When building this, a common issue was the size used due to the number of iterations. Therefore, the data sizes were reduced until further optimization of pipelining and unrolling could be evaluated. Additionally, the prime values were still static, meaning the values would be identical. This allowed for validating the results of this implementation, as shown in the next section; however, it was a critical flaw in this iteration.

The final iteration accommodated for static prime values by implementing an axi lite connection to input primes. To implement this, two integer values were added to the encrypt function and were labeled with pragma as axi lite. On the PS side interacting with the overlay, the prime values are set by accessing the register map and setting the axi lite values for the primes.

In addition to the prime values being inputted, another key feature was added in this iteration which allowed the user to input a Boolean value to switch from encryption to decryption. This was simply done by setting the exponential value to the private key instead of the public key when evaluating for decryption. On the PS side, this was implemented in the same way as the prime values by accessing the register map and setting the Boolean to 1 for decryption and 0 for encryption. This allowed for testing of the entire RSA algorithm process and provided the functionality to read the decrypted message.

# Data Validation and Verification:

To operate and validate the operation of each PS and PL function and code, navigate to the repository in the link: [https://github.com/noahe7700/RSA\_Project/tree/main/Project%20Report](%20https:/github.com/noahe7700/RSA_Project/tree/main/Project%20Report). Here is where all the code for testing and running the implementation is held.

For the PS code, run the Jupyter notebook labeled notebook #1 to see the first iteration of the PS code.

[Images from results of first notebook\*]

The input value is 132 and is encrypted with the prime numbers 137 and 227. The encrypted value becomes 29741 with the public key (3, 31099). The decrypted value is then returned as 132 with the generated private key (20491, 31099) and confirming functionality.

For the results of the second attempt, run notebook #2. This attempt encrypted the value of “Hello World” as a 1024-bit value. The resulting cipher is an unrecognizable value. However, the timing is significant, and can fluctuate greatly.

[Images from results of 2nd notebook\*]

The third notebook is what the PL is based on and will be used to verify the results of the PL implementation. The message “Hello, World!” is entered into the encryption and is converted into its ASCII values as an array. The resulting cipher is an array of encrypted values shown below.

[Image of result 3rd notebook\*]

The values are then decrypted and reconstructed as a string, forming the original message. This example used prime numbers 2027 and 3011, which will be used to test the PL later to verify the encryption. The public key for this was (65537, 6103297) and the private key was (2416153, 6103297).

For the PL overlay code, download the simple overlay files in the PL code folder. The source code for the overlay is in the source code folder under the file RSA1.cpp and is tested with RSA\_test.cpp. For testing this, open the Interface Notebook in the PL Code folder. To test the first interface, upload the overlay files to the board from the Simple Overlay folder and change the text to the correct location in the notebook. Run the first few lines of code and the section Overlay Code. The resulting values are pulled from the PL and match the values from the previous attempt on PS.

[Image of RSA Simple\*]

For the implementation where the prime values are input, a similar method is performed. To implement this iteration, first download the overlay files from the final overlay folder and update the location in the notebook. Then, assign values to the prime numbers by accessing the register map with the code: hls\_ip.register\_map.prime1 = 2027

hls\_ip.register\_map.prime2 = 3011

hls\_ip.register\_map.encr = 0 for encryption, 1 for decryption

Run the code and the resulting values should be the same as the expected values from the previous examples.

[Image of RSA with inputs\*]

\*Images are missing due to files being on Pynq in lab.

# Optimization:

Testing with pipelining optimizations of the while loop used to encrypt/decrypt the values. The results of the different optimizations tested are as shown in table 1.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Design | Latency | DSP | FF | LUTs | Comments |
| RSA\_Simple - no pipeline | 32120 | 4 | 3183 | 2833 | Default Config |
| RSA\_Simple - pipelined | NA | 31 | 155753 | 119808 | ERROR - TOO LARGE |
| RSA\_Simple - Unroll 2 | 32120 | 4 | 3183 | 2833 | Same as default |
| RSA\_Simple - Unroll 8 | 32120 | 4 | 3183 | 2833 | Same as default |
| RSA\_Simple - Unroll 8/pipeline | NA | 31 | 155753 | 119808 | ERROR |
| RSA\_Simple - while(1) pipeline | NA | 31 | 157647 | 117491 | ERROR |
| RSA\_Simple - while(d > 0) pipeline | 49550 | 4 | 14453 | 11632 | II = 90, CLK = 25ns |
| RSA\_Simple - ^ with 32-bit data | 26000 | 3 | 11826 | 9389 | II = 90, clk = 20ns |
| RSA\_Simple - ^ with 32-bit data | 13440 | 3 | 12094 | 9427 | II = 90, clk = 10ns |
| RSA\_Simple - 32-bit no pipeline | 24420 | 3 | 2251 | 1977 | Expected result, but sacrificing security in RSA... |
| RSA\_Simple - 32-bit pipeline and unroll 2 | 13440 | 4 | 23576 | 18651 | No improvement… |
| RSA\_Simple - 32-bit pipeline with 8ns | 11100 | 3 | 12306 | 9476 | Latency speed-up |
| RSA\_Simple - 32-bit with 8ns | 20420 | 3 | 2443 | 2044 | Best so far! |

Table 1\*

\*not finished with optimizations… this is initial findings.

# Conclusion:

# References

|  |  |
| --- | --- |
| [1] | R. Rivest, A. Shamir and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," February 1978. [Online]. Available: http://people.csail.mit.edu/rivest/Rsapaper.pdf. |