Implementation of RSA in Verilog

Rohith Rajasekaran[#1], Andrew Zarour[#2], Anish Junnarkar[#3], Mohammed Aly[#4]

[#]*ECE Department, California State Polytechnic University, Pomona*
3801 W Temple Ave, Pomona, CA 91768

[1]rrajasekaran@cpp.edu

[2]akzarour@cpp.edu

[3]adjunnarkar@cpp.edu

[4]mealy@cpp.edu

***Abstract*— RSA is one of the oldest and most widely used public-key algorithms for secure data transmission. The purpose of this paper is to implement the RSA algorithm using Verilog. This implementation consists of the following parts: Key Generation, Encryption, Decryption, VGA, and UART. The purpose of the Key Generation is to create a public key such that the data can be encrypted by anyone else, while having a private key to decrypt this message. Encryption is the actual process of encoding the message such that it cannot be read while decryption is the process of decoding the encoded message such that the message can be read. UART will be used in order to communicate to the Nexys A7 100T FPGA board and VGA will be used in order to display the encrypted and decrypted message on a monitor. This project as a whole is written entirely in Verilog and Vivado was used in order to perform a global synthesis of the RTL design.**

***Keywords*— FPGA, Verilog, UART, VGA, RSA**

I. Introduction

The acronym RSA comes from the last names of the men who had created the algorithm back in 1977: Ron Rivest, Adi Shamir, and Leonard Adleman. The idea of RSA is simple. A message that wants to be securely sent will be encrypted using a public key. This encrypted message would be sent to a user (receiver).The receiving end will have a private key that can decrypt the encrypted message. Through this method, no private keys are being shared, and without these keys, it becomes extremely difficult to actually encode the message. VGA, also known as the Video Graphics Array, was a popular display standard that was created by IBM back in 1987. The VGA generally provides a 640x 480 resolution color display and displays 16 colors at a time. However, if the resolution gets lowered, VGA can display around 256 colors. UART, also known as the Universal asynchronous receiver-transmitter, is a hardware component that allows for asynchronous serial communications. Unlike other serial communication protocols today, such as SPI and I2C, UART is a physical component on an FPGA board.

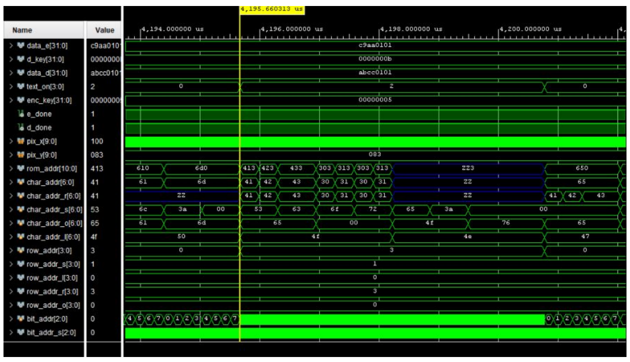
II. VGA

VGA is a connection type for monitors, also known as the Video Graphics Array connector. The VGA for this lab was mainly used to output the result of the encryption or decryption process, in 8-bit format.

A. Code Process

In order to get the right VGA output, we first worked from our lecture with Dr. Aly, where we found out how we need to sync the vertical and horizontal pixels at the right time. VGA only works at a twenty five Mhz clock, which means that the rest of the verilog VGA system needs to be synced to that. In order to get the text to display, we needed two different code files, a text rom file which contains the information of which pixels need to turn on for each letter or number, and a text generator, which has multiple address registers that display the text on the monitor when the address counter gets to that point. The VGA code was taken from a pong game on verilog github page, from Bogini. The code contained multiple text registers, but we isolated just one, the "ruleset text" case, and used its address register to display the 8 bit output, which would either be the encrypted or decrypted code.

The VGA is linked to the msg_in register, which is the 32 bit data input into the encryption and decryption code. When the reset for the encryption and decryption code is flipped high, the monitor will refresh the new output. Figure 1 shows the output of the testbench for both the encryption and decryption code, and the VGA output. The output of the VGA is very clear here, as we can see the values that the font rom code is outputting, along with the respective registers changing when the addresses match the locations of where the ruleset text should be. The 25 Mhz clock was created through the clock wizard available on the Vivado software, under IP. We implemented the clock using PLL, and renamed the clock signal to clk_25Mhz in the code.



Name	Value
data_e[31:0]	c9aa0101
d_key[31:0]	0000000b
data_d[31:0]	abcc0101
text_on[3:0]	2
enc_key[31:0]	00000005
e_done	1
d_done	1
pix_x[9:0]	100
pix_y[9:0]	083
rom_addr[10:0]	413
char_addr[6:0]	41
char_addr_r[6:0]	41
char_addr_s[6:0]	53
char_addr_o[6:0]	65
char_addr_l[6:0]	4f
row_addr[3:0]	3
row_addr_s[3:0]	1
row_addr_l[3:0]	0
row_addr_r[3:0]	3
row_addr_o[3:0]	0
bit_addr[2:0]	0
bit_addr_s[2:0]	0

Fig. 1 A little hard to read here, but this is the testbench of the VGA and encryption/decryption code working together. You can see the char registers changing very quickly when the address register matches with the location of the rule text.

III. UART

UART, or Universal asynchronous receiver-transmitter, is a computer hardware device for serial communication in which the data format and the transmission speeds are configural. In UART communication, two UARTs communicate directly with each other through two wires. Each UART has a tx, or transmitter, and an rx, or receiver. In order to make the two UARTS communicate with one another, you must connect the tx of one UART with the rx of the other, receiving UART. It is framed not with a clock, but rather with start and stop bits that denote the beginning and end of the data packet being transferred. Once the tx and rx are synchronized, and the tx begins to send data, the transmitting UART adds the start bit, the parity bit, and the stop bit to the data frame. The start bit signals to the rx that it needs to begin to read the data being sent over, the parity bit ensures that all data was correctly received, and the stop bit denotes the end of the data stream, doing so by changing the voltage across the transmission line from a low voltage to a high voltage for at least two bit durations. Note: This data is sent at a specific rate, known as the baud rate, or the speed of data transfer, expressed in bits per second. This baud rate must be synchronized between the two UARTS, otherwise the rx may receive incorrect data. Once the entire packet is sent serially from the transmitting UART to the receiving UART, the receiving UART samples the data line at the pre-configured baud rate. It then disregards the start bit, parity bit, and stop bit from the data frame. The receiving UART then converts the serial data back into parallel and transfers it to the data bus on the receiving end.

A. Code Process

In our project, we utilized a UART to send data from our keyboard to the board. Here, the keyboard acts as a transmitter, and the FPGA acts as a receiver. Through putty, we are able to transmit data to the board. After receiving the data, the FPGA would simply display the data received, doing so by utilizing 7 segment displays, as well as LEDS.

Approaching this project made us realize that we needed a couple pieces. Firstly, we realized that for this project, we did not need to incorporate a tx for our project. The FPGA was only responsible for displaying the data already transmitted from the computer. It did not need to send any data back to the computer, therefore it did not need a tx. In our project, we did not utilize a baud rate generator, but rather preselected the speed of the data transmission/reception by calculating how many clock cycles would be run per uart bit. This was found by dividing the bitrate at which the we wanted the transmitter and receiver to run at (9600 pulses per second) by the hertz at which the internal clock on the FPGA runs at. Our rx module contained six inputs and one output. The inputs include “clk,” “resetn,” “uart_rxd,” “uart_rx_en,” “uart_rx_break,” “uart_rx_valid,” and one output of 8 bits length: “uart_rx_data.” The clk is a clock that is utilized in checking certain flags, such as whether or not the reset was enabled or not. “resetn” was the reset, which worked as a boolean value. The data would only be read if the reset was not on. “uart_rxd” represents the input that comes from the keyboard, in the form of an ASCII value, one bit at a time. This bit is then loaded into an 8-bit register, uart_rx_data, which is used to hold the binary equivalent of this ASCII value. "uart_rx_en" is used as a switch that will not allow the inputted data to be read or stored unless turned to 1. uart_rx_break works as an extra precaution. It is utilized in the code to inform us if there was any issue in receiving the data, primarily by comparing the value of data being received with uart_rx_valid, another variable that is able to verify if the data is valid, received, and available. It does this by utilizing a finite state machine that verifies the data does not change when loading it into the register.

Additionally, this project utilizes a module that is connected to the 7-segment display. This module, "ssd_gen" loads the input into a 32-bit register. It then takes this input and, according to a multiplexer, will take 4-bits of the input and display the corresponding number or letter on the 7-segment display. The output of this module includes a 8-bit output, "an," and a 7-bit register "a_to_g." For every 4-bits, loaded into the module, one "an", or 7 segment is utilized. In our specific code, loading our 8 bit output from the rx module turns on two 7-segment displays, each having their own "a_to_g" values that specify which leds need to be turned on in order to display the correct digit.

Our top module, impl_top utilizes nine inputs and outputs: "clk," "sw_0," "rst," "sw_1," "uart_rxd," "a_to_g," "AN," "DP," and "led2." The clock was used to act as a reminder for the system to check for certain changes in signals at each cycle. "sw_0" and "sw_1" were connected to the constraints file of our FPGA, and acted as the enable for the receiver. If our sw_0 was off, then the FPGA simply would not display anything on the 7 segment or the leds (mentioned later). a_to_g, AN, and DP were all used to allow the 7 segment display to properly reflect the data input, and the "led2" output register was connected to the .xdc (constraints) file of the project, allowing for the output of the top module to be represented in leds on the board. To elaborate, our top module utilized an instance of the rx module, the ssd_gen module, and another module titled "mux_binary," which uses a decoder to convert the hexadecimal ASCII input to a binary number instead. This allows the data to be fed into the encryption / decryption code of the project. The output of this module, being the binary number that is reflected on leds on the FPGA.

IV. Key Generation

In cryptography, key generation refers to the process of generating keys. In RSA, key generation is a very valuable topic because it is the basis of the entire encryption and decryption process.

A. Generating the Public and Private Keys

To begin the process of finding the public key, one must generate two large prime numbers, which would be called as p and q. The size of these p and q values determine how difficult it will be to hack the code. A general rule of thumb is the larger the prime numbers, the harder it is to crack. From there, one would generate the RSA modulus, which is denoted by n. The calculation to solve for n is $n=p*q$. From here the derived number (denoted as e) is selected. This derived number must be such that $1<e<(p-1)(q-1)$ as well as being coprime with e. From this, we have calculated the RSA public key (n,e).

The Private Key d is calculated based on the values of p, q, and e. The value of the number d is the inverse of e modulo (p-1)(q-1). The relationship of all of these values is as follows: $e*d= 1 \bmod (p-1)*(q-1)$.

B. Encryption and Decryption

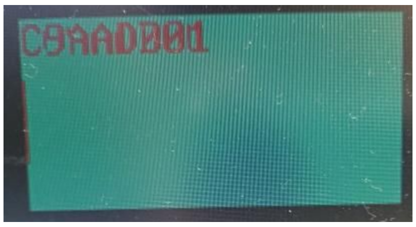
Encryption is the process of ciphering information. The process of encrypting a plaintext message, p, is shown through the equation of $C= p^e \bmod n$, where C is the ciphertext, p is the plaintext message, e is the derived number, and n is the derived modulus.

Decryption is the process of deciphering information. The process of decrypting an encrypted message can be shown through the equation $p= C^d \bmod n$, where C is the ciphertext, d is the private key value, and n is the derived modulus.

Although the process of encrypting and decrypting itself is really simple, it has proven to be a secure way of transferring messages. The inverter code was used to generate the keys for the cipher code, and it used an FSM and bit manipulation to do the calculations mentioned above.

C. Implementation of RSA

Essentially, the implementation process is extremely similar to what we talked about earlier. We created a generator which calculated the values of the modulus and derived numbers. Additionally, we had an inverter that was used to find the private key d, and the public key e. From here, we created an encryptor and decryptor which is used for encryption as well as decryption. However, we ran into a few issues without code. It turns out that operations such as modulo, division, and exponents are not synthesizable through verilog, meaning that our project would not work. To solve this issue, we used sections of code made by Rajandeep on Github and manipulated it such that it would better fit our requirements for our project (reference of the Github is shown in the references page). His project had a modulo exponentiation, which was a method that can be used to replace calculations made by modulus and exponents. Additionally, he also had code for an inverter, modulus (by itself) and divider which were all synthesizable. Using this additional help, we also created a control file which would deal with all of the resets, clocks, and switches that we used for the Encryption/Decryption process. Here figure 2 displays the output of a decryption test we did, which we used to make sure that the cipher algorithm was working as expected.



```
C9AADB01
```

Fig. 2 Monitor Display of the encrypted code, outputting to the VGA connection to the monitor.

V. Conclusions

This project was completed with the board functions/modules UART, VGA, LED, switches, buttons, and the Seven Segment. The Encryption/Decryption code from Rajandeep's Github was used, but we also have original code for the process, which we had to scratch due to the code being unoptimized. The code was linked to the VGA code, which output the cipher code or the deciphered code correctly, up to 256 bits, with a p and q limitation of 256 bits as well. The input also worked correctly from UART, where we were able to generate the 8 bit input as ASCII, and used a multiplexer to get the binary equivalent of the keyboard inputs. We were able to output both the binary and the ASCII information to the LEDs and Seven Segment, respectively. While these parts of the lab worked as expected, we did run into timing issues when we connected the UART modules to the encryption/decryption and VGA code.

Next steps in this project would be to sort out these timing issues, and allow for multiple hex inputs from PuTTY, which we were using for the UART inputs. When running the VGA and the cipher code together, no issues occur with the input saved in the ROM. The UART also doesn't suffer from any issues when run independently of the cipher and VGA code. Another problem that we had faced was related to the VGA, where we were only able to output a total of 32 bits. In cases where the p and q values are very large, or the input is very large, the cipher output to the monitor could be larger than 32 bits. These special cases are another step that could be done in the future, given more time to work on this project. We designed both a FIFO and manual memory for the UART, and the FIFO was the section of the code where the timing errors in the code popped up. The manual memory that we designed had manual inputs and read/write registers for access, and while it worked better, there were more issues on the timing on when the input would reach the cipher algorithm.

Acknowledgment

We would like to thank Dr. Aly for providing us with this challenge as well as mentoring us throughout the project.

References

[1] Bogini, B. "Bogini/Pong." GitHub, 2012, github.com/bogini/Pong/blob/master/pong_text.v.

[2] Bogini, B. "Bogini/Pong." GitHub, 2012, github.com/bogini/Pong/blob/master/font_rom.v.

[3] Bogini, B. "Bogini/Pong." GitHub, 2012, github.com/bogini/Pong/blob/master/pong_top.v.

[4] E., Chiranth. "Implementation of RSA Cryptosystem Using Verilog." Implementation of RSA Cryptosystem Using Verilog.pdf, 2011, www.ijser.org/researchpaper/Implementation_of_RSA_Cryptosystem_Using_Verilog.pdf.

[5] Ireland, David. The Euclidean Algorithm and the Extended Euclidean Algorithm, 2019, www.di-mgt.com.au/euclidean.html.

[6] Marshall, Ben. "Ben-Marshall/Uart." GitHub, 12 Aug. 2020, github.com/ben-marshall/uart/blob/master/rtl/uart_rx.v.

[7] Merrick, Russel. "Go Board - UART (Universal Asynchronous Receiver/Transmitter)." The Go Board - UART Project (Part 1, Receiver), 2019, www.nandland.com/goboard/uart-go-board-project-part1.html.

[8] N/A, Rajandeep. "Rajandeep/RSA-CRYPTOSYSTEM-Using-Verilog." GitHub, 2016, github.com/Rajandeep/RSA-CRYPTOSYSTEM-using-verilog/tree/master/RSA.srcs/sources_1/new.

[9] Nanasekaran, Lavanya G., "Reconfigurable-Computing-CalPoly-Pomona/Quantum-Computing." GitHub, 2019, github.com/Reconfigurable-Computing-CalPoly-Pomona/Quantum-Computing/tree/master/src/rsa/vhdl.

[10] Shams, Rehan. "Cryptosystem an Implementation of RSA Using Verilog." TechRepublic, 2013, www.techrepublic.com/resource-library/whitepapers/cryptosystem-an-implementation-of-rsa-using-verilog/.

[11] Torng, Christopher. "Prime Number Generator and RSA Encrypter/Decrypter." Cornell University Website Template - Two Column, 2005, people.ece.cornell.edu/land/courses/ece5760/FinalProjects/f2011/clt67_yl478/clt67_yl478/index.html.

[12] Vipin, N/A. Synthesisable Verilog Code for Division of Two Binary Numbers, 1 Jan. 1970, verilogcodes.blogspot.com/2015/11/synthesisable-verilog-code-for-division