ECE532

Prime Number Generator and RSA Encrypter/Decrypter

Group Report

Group 15
Qiaozhi (George) Zhao
Michael Fang
William Wen

April 9th, 2012
# Table of Contents

1. Overview
   1.1. Background and Motivation
   1.2. Math of RSA
   1.3. Design Goals
   1.4. Objectives
   1.5. Input/output
   1.6. Data Flow
   1.7. System Block Diagram
   1.8. Brief Description of IPs

2. Outcome
   2.1. Results
   2.2. Challenges and Problems
   2.2. Improvements and Future Work

3. Project Schedule

4. Detailed Description of IP Blocks
   4.1. Existing IPs
   4.2. Custom IPs
   4.3. Softcore Processor and software modules

5. Design Tree Description

Reference

Appendix A – Primality Test Verilog

Appendix B – Key Generation Module
1. Overview

1.1. Background and Motivation

The RSA cryptosystem is by far the most used public key encryption system. Its name is an abbreviation of the names of R. Rivest, A. Shamir and L. Adleman, who published it in 1978 [1] and is used to secure many commercial electronic transactions. The algorithm is also being developed into real product such as RSA secure token [2] to generate password for web server or VPN. We thought it would be interesting to implement it in hardware to see how these algorithms can be implemented in FPGA and possibly how much resource it may take to implement in an ASIC design.

1.2. The Math of RSA – a brief overview

We select two primes \( p \) and \( q \) and compute \( n = p \times q \). \( n \) is referred to as the modulus. Next, we choose an integer \( e \) that is relatively prime to \( (p-1)(q-1) \). \( e \) is referred to as the encryption exponent. The public encryption key consists of \( e \) and \( n \). A message \( M \) is encrypted by the modular exponentiation operation \( C = M^e \mod n \), producing the encrypted message \( C \). [3]

To perform decryption, we need to find the decryption exponent \( d \) that is a multiplicative inverse of \( e \mod (p-1)(q-1) \). The private decryption key consists of \( d \) and \( n \). The encrypted message \( C \) is decrypted by the modular exponentiation operation \( M = C^d \mod n \), producing the original message \( M \).

1.3. Design Goal

The team is proposing to implement a prime number generator and a RSA data encrypter and decrypter on the Xilinx Virtex Pro II FPGA board that is able to manipulated user stored text data.

The user data is retrieved from the Compact Flash and push buttons are used to select encrypt or decrypt option. The data is then passed through the RSA algorithm which is made of the following modules: a prime number generation module, a key generation module, an encrypter and a decrypter module. The data is then displayed via the VGA controller to show the public and private keys used, as well as decrypted and encrypted data.
1.4. Objectives

1.4.1. Features

- Ability to display encrypt and decrypt text data
- Ability to display public and private key used
- Ability to allow user to choose actions of either encryption or decryption

1.4.2. Functional Requirements

- Must generate sets of prime numbers
- Must be able to generate public and private sets of keys from the prime numbers.
- Must be able to encrypt and decrypt text data stored in CF card

1.4.3. Acceptance Criteria

- 128bits (number may alter depending on the complements available on Xilinx Virtex II Pro board) RSA key are generated successfully
- The decrypted data has not been altered in any way from the original
- Text data and cyphered text data display on VGA without data loss

1.5. Input/output

<table>
<thead>
<tr>
<th>Encrypting</th>
<th>Decrypting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>Original Text Data</td>
</tr>
<tr>
<td>Output</td>
<td><strong>Decrypted</strong></td>
</tr>
</tbody>
</table>

1.6. Data Flow
1.7 Block diagram

**Legend**

<table>
<thead>
<tr>
<th>Module Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Custom IP</td>
<td></td>
</tr>
<tr>
<td>Existing IP</td>
<td></td>
</tr>
</tbody>
</table>

**Diagram**

- VGA Display: Display of prime number (debug) and display encoded/decoded data
- SDRAM Original Data buffer
- SDRAM Encrypted Data buffer
- CF card Text Data
- TFT Controller: Interface to control SVGA
- MPMC: Multiport Interface logic for SDRAM access
- System ACE: Controller for CF access
- PLB
- MicroBlaze Processor
- FSL
- Prime Number Generator
- Key Generator
- Decryption
- Encryption
- Xilinx Virtex II FPGA
- XUP2P Development Board
- Push Buttons: User input to start encrypt or decrypt
### 1.8 System modules:

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
<th>IP</th>
</tr>
</thead>
<tbody>
<tr>
<td>fsl_v20_[0-1]</td>
<td>Fast Simplex Link which performs uni-directional point-to-point communication between elements on the FPGA. Since its uni-directional, total two FIFO connections between the prime number generator/encrypter/decrypter modules carry processed data to and from the MicroBlaze processor.</td>
<td>Xilinx</td>
</tr>
<tr>
<td>Plb</td>
<td>Processor Local Bus is a high bandwidth bus used as an interface to devices on the FPGA. PLBs utilized in this project.</td>
<td>Xilinx</td>
</tr>
<tr>
<td><strong>mb_plb:</strong></td>
<td>Allows the connected soft processor to read from and write to other components on the FPGA.</td>
<td></td>
</tr>
<tr>
<td>DDR_SDRAM(mpmc)</td>
<td>Multi-Port Memory Controller which provides access to the DDR-SDRAM using multiple memory ports to interface with the RAM over PLB busses. Holds video data and encryption/decryption data</td>
<td>Xilinx</td>
</tr>
<tr>
<td>bram_block</td>
<td>Block RAM. A configurable memory module which is the local memory for the soft processor.</td>
<td>Xilinx</td>
</tr>
<tr>
<td>dlmb and dlmb_ctrl</td>
<td>Local Memory Bus and Controller interface connects and store Data memory for the MicroBlaze processor</td>
<td>Xilinx</td>
</tr>
<tr>
<td>ilmb and ilmb_ctrl</td>
<td>Instruction memory and controller for instruction memory.</td>
<td>Xilinx</td>
</tr>
<tr>
<td>rs232_uart_1</td>
<td>Serial communication link for debugging</td>
<td>Xilinx</td>
</tr>
<tr>
<td>xps_tft_0</td>
<td>XPS Thin Film Transistor Controller reads pixel data from memory and outputs it to video display via an SVGA port.</td>
<td>Xilinx</td>
</tr>
<tr>
<td>MicroBlaze</td>
<td>A multi-purpose soft processor core that controls the system and video interface</td>
<td>Xilinx</td>
</tr>
<tr>
<td>SystemACE</td>
<td>Controller for Compact Flash Memory</td>
<td>Xilinx</td>
</tr>
<tr>
<td>xps_gpio</td>
<td>XPS General Purpose Input/Output core for PLBs to control push buttons on the XUP board</td>
<td>Xilinx</td>
</tr>
<tr>
<td>miller_robin_ip_0</td>
<td>The three custom modules are combined into a single generated IP for simplicity and ease of connection with fsl Prime Number Generation Module: generates a random number and then test its primality using the miller robin algorithm.</td>
<td>Custom IP</td>
</tr>
<tr>
<td></td>
<td>Key Generation Module: generates exponent and modulus which is needed by the encrypter/decrypter module</td>
<td>Custom IP</td>
</tr>
</tbody>
</table>


**Encrypter/Decrypter Module:** Either encrypts or decrypts the data, based on the exponent through the exponentiation process.

<table>
<thead>
<tr>
<th>Custom IP</th>
</tr>
</thead>
</table>

### 2. Outcomes

Due to various implementation problems and complexity in the original vision to fit with the design of the XUP2P board specs, the team made some design changes to address issues with input data access and encryption/decryption complexity. The details of differences between the original design specification and the actual implementation are outlined as follows.

#### 2.1. Results

<table>
<thead>
<tr>
<th>Original Features</th>
<th>Current Features</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Display encrypt and decrypt text data</td>
<td>Display encrypt and decrypt text data</td>
<td>✓</td>
</tr>
<tr>
<td>Display public and private key used</td>
<td>Printed out through serial connection</td>
<td>For debugging only ✓</td>
</tr>
<tr>
<td>To allow user to choose actions of either encryption or decryption</td>
<td>Push button to select the prime number generated.</td>
<td>Encrypt and Decrypt done in sequence</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Functional Requirements</th>
<th>Results</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generate sets of prime numbers</td>
<td>Completed</td>
<td></td>
</tr>
<tr>
<td>Able to generate public and private sets of keys from the prime numbers</td>
<td>Completed</td>
<td></td>
</tr>
<tr>
<td>Encrypt and decrypt text data stored in CF card</td>
<td>Data is encrypted and decrypted.</td>
<td>CF card is not setup, input data file is written to the DDR memory instead. See “2.2 Challenges” for details</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Acceptance Criteria</th>
<th>Results</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>128 bit RSA key are generated successfully</td>
<td>32 bit</td>
<td>Sequential divider inefficient for larger number of bits, so we decided to use 64 bit instead. But that was still too much computations and we did not have enough slices on the FPGA, thus switched to 32bit</td>
</tr>
</tbody>
</table>
The decrypted data has not been altered in any way from the original

| Text data and cyphered text data display on VGA without data loss | Completed |

As seen from the above tables, we have met most of our proposed features and functionalities, and have met all the basic criteria for the encryption and decryption modules. Some challenges were faced to properly implement a filing system to be able to store an external data file using the compact flash memory card slot. We also had lots of trouble implementing all the computations with in hardware: problem includes completing the algorithms in HDL, find a sufficient divider, and running out of slices/LUTS on the FPGA. Details of challenges will be described in the next section. Overall, most of original goals were met with a few minor changes to the algorithms required for practical implementation.

2.2. Challenges/Problems

2.2.1 SystemACE/CF reader

We were unable to configure the SystemACE controller to read and write data to the Compact Flash card. The original plan was to use the Xilinx EDK to setup the SystemACE IP and the software library XilFatFS for file I/O. SystemACE was designed to configure the FPGA at power-up, and CF was one of the several method for it to store configuration data externally. This was the main cause of problems we ran into. Due to the original nature for configuration purposes, the SystemACE controller requires very particular formatting of the compact flash. The following answer record was found on the Xilinx website which talks about formatting CF cards: http://support.xilinx.com/xlnx/xil_ans_display.jsp?iLanguageID=1&iCountryID=1&getPagePath=14456. But after many tries, we still receive the red error LED light on the board, indicating improper setup for the SystemACE CF controller.

2.2.2 IP generation and FSL connection

Because most of our design is composed of hardware modules, one of the biggest challenges was to generate our custom codes into IPs that can be connected to the soft processor via FSL. Because we had not design out custom IPs with the use FSL to connect these modules to Microbraze in mind, all three modules (prime number generator, key generator,
encryption/decryption module) had to be redesigned to comply with this. For the most part, the problem lies within the fact that FSL only offers a single bus connection, therefore, it cannot take in control or enable signals. This was problematic for us to distinguish the different input data for the custom IPs, which includes the input message, the prime numbers, and the modulus and encryption / decryption exponents. It was particularly difficult for the IP to separate the difference between doing an encryption or a decryption without and enable or control signals. In the end, we decided to complete the action in sequence so the decryption is carried out automatically after the encryption process.

2.2.3 Computation and the number of bits

We also encountered the issue that the Xilinx provided divider IP could only handle 32 bits. This was very far off from the originally propose 128 bit secure encryption keys. We made a switch in design and obtained a 64 bit divider from www.Opencore.org. However, towards the end of our design, upon integration, we found that the computations of all of the custom modules together were too much for the FPGA. An error during place and route stated that there was not enough slices for placement. In the end, we were forced to complete the design with a 32 bit encrypting keys.

2.3. Improvements and Future works

If we could start again, it would have been a better idea to plan everything out clearly first, this would for example eliminate a lot of wasted time and work because we had to redesign our custom block to work with FSL connections.

With more time permitting, the group would have the opportunity to debug and properly format the CF memory for external input of data. This would be an ideal scenario for a security encryption module which protects user’s data in a removable data storage device such as a memory card.

Other Improvement to be worked includes being able to have user controlling when to decrypt the encrypted data, although this might require the use of PLB instead to connect our custom IPs, which is much more complicated than FSL. In addition, with some kind of external storage system, a good feature to have is to be able to select which file to be encrypted out of several files.
3. Project Schedule

Week 1 (Feb 15)
Projected:
  • Develop the prime number generator using random number generator and prime numbers test in software
  • Write test cases to generate different sets of unique prime numbers
Actual:
  • Developed the prime number generator using random number generator and prime numbers test in HDL
  • Tested in simulation with test cases
On schedule, decided to work using HDL because it would be faster and more meaningful

Week 2 (Feb 22)
Projected:
  • Set up the VGA Interface module
  • Capable of showing different background colors
  • Instantiate SDRAM to store data on it
Actual:
  • Started on VGA controller
Stuck working on VGA controller

Week 3 (Feb 29)
Projected:
  • Set up CF Interface module
  • Able to read text file located on the CF card given the file name
  • Able to buffer the text file from CF card into BRAM module
Actual:
  • Found a existing VGA module
  • Completed the Key Generation module
Moved the Key Generation module ahead of schedule and pushed back work on CF one week due to personal schedules

Week 4 (March 7)
Projected:
  • Integrate BRAM interface module with VGA interface module
  • Display text file content on the monitor through VGA
  • Develop tests to test the functionality of VGA and CF module
Actual:
  • Started working on CF controller
  • Successfully display characters on VGA
On schedule

Week 5 (March 14)
Projected:
• Setup microprocessor
• Migrate the previous prime number generator codes to FPGA board
• Write standalone test to verify the functionality of both modules (simulation)

Actual:
• Completed the encryption/decryption module
• Trouble with CF card

Ran into our problems with the CF card, decided to finish writing the code for all three custom modules first before generating IP.

Week 6 (March 21)
Projected:
• Develop Key Generation module
• Develop testbench for Key Generation module
Actual:
• Generating IP for the custom modules
• Completed CF setup but still problem with the card

Gave up on CF memory, due to the fact we realized we need a lot of work and time to redesign our IPs in order to use FSL.

Week 7 (March 28)
Projected:
• Develop Encryption/Decryption module
• Develop testbench to test functionality of the module
Actual:
• FSL connection setup

Behind schedule, still setting up FSL

Week 7 (April 4)
Projected:
• Connect all modules together
• Develop tests to meet functional requirements
• Further debugging
Actual:
• Debugging of the integrated systems.

Behind Schedule in debugging and integrating

4. Detailed Description of IP Blocks

4.1. Existing IPs
4.1.1. FSL

FSL stands for Fast Simplex Link. It is a high speed FIFO for direct and simple unidirectional links between hardware cores and the MicroBlaze processor. The group chose to use FSL
because it is simple to setup and more importantly, it offers up to 8 FSL busses that greatly suit our design with many custom hardware modules. A dedicated FSL link was established to interface the MicroBlaze processor with each of the custom IP modules. Since the FSL bus is uni-directional, two links were required for each module. One is dedicated for receiving data from the Encryption/Decryption modules and the other is used exclusively for sending out from the MicroBlaze. The FIFO implementation of this bus is also ideal for the design, since overflow of data will not be lost and still is processed in the order it arrives. Setting up the system to use multiple FSL busses was very straightforward. Using the XPS’s GUI or manually by editing the MHS configuration file, multiple FSL buses were added by going into the “Add/Remove Cores” dialog and going to the "Bus Connections" tab. To connect it to the MB, we configure the ‘ bus’ option under config IP, and set the number of FLS links to 2.

Generating IP

4.1.2. MPMC

The multi-port memory controller is an existing Xilinx hardware module we used to create multiple interfaces with the SDRAM. There was a problem using our board as it was the 512MB RAM instead of the default 256MB one. We followed the following document to reconfigure our board for this memory:
http://www.eee.g.toronto.edu/~pc/courses/edk/doc/512MBfix.txt

In our project the DDR memory is used for:

- Store the VGA frame
- Store the text data
- Store the encrypted data

The address range allocated for each purpose is outlined in the following table.

<table>
<thead>
<tr>
<th>Address Range</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x90000000-0x9FFFFFFF</td>
<td>256 MB</td>
</tr>
<tr>
<td>0x90000000-0x90078000</td>
<td>1 MB</td>
</tr>
<tr>
<td>0x91000000-0x91078000</td>
<td>1 MB</td>
</tr>
<tr>
<td>0x92000000-0x94FFFFFFF</td>
<td>48MB</td>
</tr>
<tr>
<td>0x95000000-0x97FFFFFFF</td>
<td>48MB</td>
</tr>
<tr>
<td>0x98000000-0x9AFFFFFFF</td>
<td>48MB</td>
</tr>
</tbody>
</table>
Each module that requires memory access was given its own port and plb connection in order to allow simultaneous memory accesses to different addresses. For our design, we are using Port 0 in connection with \textit{mb\_plb}. The microblaze needs to be able to read the input data, write encrypted data and read/write decrypted data to and from memory.

4.1.3. TFT controller

The XPS TFT controller is another Xilinx IP block, it connects to the PLB bus and act as a PLB master that is connected to the MPMC in this project to control the SVGA. The tft controller reads image data previously stored in the DDR memory via a dedicated PLB bus. Then it converts these video data to VGA compatible data that is available to display on the monitor.

4.1.4. The System Advanced Configuration Environment (System ACE)

The SystemACE controller was designed to configure the FPGA at power-up using several methods of external memories. We are utilizing the SystemACE compact flash interface to provide external input data. It is an excellent choice for this project because it provides large amount of storage, and is removable which suit the projects motivations for a data secure encryption system. The PLB System Ace Controller is an IP core supplied by Xilinx that acts as a bridge between the microprocessor and the actual System ACE controller chip that also resides on the multimedia board. The System ACE controller chip in turn interfaces with the compact flash card for reading and writing data. We failed to implement this module successfully as previously described section 2.2.

4.2. Custom IP components

Public and private key generation is implemented by a Prime Number Generation component alone with a key generation module. The \textit{Prime Number Generation Module} generates two distinguish prime numbers $p$ and $q$ randomly that are used by the key generation module. The \textit{Key Generation Module} generates $n$, the prime modulus, $e$, the encryption exponent, and $d$, the encryption exponent using prime numbers $p$ and $q$. The \textit{Encrypter Module} is used to
alter the data based on the prime modulus $n$ and the encryption exponent $e$. The Decryption Module uses the decryption exponent $d$ and the prime modulus $d$ to restore the original data.

MB: MicroBraze;
PNG: Prime Number Generator;
KG: Key Generator;
EM/DM: Encryption/Decryption Module

Since we are combining all three modules into one IP, and FSL offers no control/enable signals, we decided to use a large FSM to assign states to which module the design will move to. Please see the abbreviated FSM in the following figure. When the design is ready, it will begin by sending a 0 on the bus as a flag to begin the PNG module. The PNG module will constantly generate primes numbers and sends it to the MB, the MB will stop it when the user has selected two prime numbers using the push button. Which will then send out 1 on to the bus as a flag to move to the KG module. The KG module will then send the encryption keys to the MB and in a similar process the MB will facilitate the transition to the final encryption module.
4.2.1 Prime Number Generator

Inputs a target accuracy ‘k’ and an odd integer ‘n’ generated through a random number generator which we wish to check for the primality. Outputs 1 if it is prime and sends out the integer n.

While repeating within a loop for ‘k’ times we check to see if n is a prime number. The algorithm is based the property of strong pseudo primes where an inequality is used to check if a number is composite, the more numbers of tests for the inequality fails, we more firm we are that the number is a prime. The number of tests, ‘k’ is set to a quarter the size of the integer ‘n’ to
make sure we test it enough times to confirm that it’s a prime number. On the other hand, once a test passes we know it is a composite number and move on to the next random number.

The inequality test relies on square roots of unity. Suppose $x$ is a square root of $1 \mod p$, where $p$ is a prime greater than 2, $x^2 \equiv 1 \mod p$, which results in $(x-1)(x+1) \equiv 0 \mod p$. So $x = (1 \mod p)$ or $x = (-1 \mod p)$. For a odd prime $n$, $n-1$ is an even number and can be written as $2^s \cdot d$ with $s$ and $d$ as positive integers and $d$ odd. For all $a$ within finite field $(\mathbb{Z}/n\mathbb{Z})^*$, then: $a^d \equiv 1 \mod n$ or $a^d \equiv -1 \mod n$ for some $0 \leq r \leq s-1$. Recall Fermat’s Little Theorem: $a^{n-1} \equiv 1 \mod n$. If we repeatedly take square roots of $a^{n-1}$, we will get either 1 or -1. This means that if $a^d \not\equiv 1 \mod n$ or $a^d \not\equiv -1 \mod n$ for some $0 \leq r \leq s-1$, then $n$ is not prime.

Therefore if the test passes with $a$, we are sure of $n$’s compositeness. We can call ‘$a$’ a witness for the compositeness of $n$. Otherwise, we can call ‘$n$’ a strong probable prime. We can generate our ‘$a$’ randomly in order to probabilistically determine $n$’s primality. [4] [5] The code is included in [Appendix A].

4.2.2 Key Generator

This Key Generator is able to take two prime number ‘$p$’ and ‘$q$’ as input, generated by the prime_number_generator, and output the keys, ‘$e$’, the encryption exponent, ‘$d$’, the decryption exponent, and ‘$n$’ the modulus, which will be used to in the encryption_decryption module. ‘$n$’ is simply the product of ‘$p$’ and ‘$q$’.

This module performs the Extended Euclidean algorithm to find the greatest common divisor of ‘$e$’ and ‘$(p-1)(q-1)$’ as well as the modular inverse of ‘$e \mod (p-1)(q-1)$’. By finding the greatest common divisor we can determine wither ‘$e$’ and ‘$(p-1)(q-1)$’ are relatively prime to each other. Then the module returns the values of ‘$e$’ and the modular multiplicative inverse ‘$d$’ when $d$ is positive. Otherwise, $e$ is incremented by 2 and the algorithm executed again. ‘$e$’ will be used as the encryption exponent and $d$ as the decryption exponent.

The Code is attached in [Appendix B].

4.2.3 Encryption/Decryption Module

The basic algorithm for the encryption module is as follows: we have the encryption exponent $e$, modulus $n$, and decryption exponent $d$. To encrypt the message $M$ into cryptic $C$, $M^e$
mod \( n = C \). Reversely, \( M = C^d \mod n \). To carry out the modulo exponentiation function in HDL, we found a divider from www.Opencore.org. To implement the actual modulo exponentiation in an efficient way, the following method is used: given two integers \( a \) and \( b \), the following two equations are equivalent:

\[
    c \equiv (a \cdot b) \mod m \\
    c \equiv (a \cdot (b \mod m)) \mod m
\]

The algorithm is as follows, essentially, it increments a \( e' \) by one until \( e' \) reaches \( e \), while multiplying by \( b \) following the above equation.

1. Set \( c = 1, e' = 0 \).
2. Increase \( e' \) by 1.
3. Set \( c \equiv (b \cdot c) \mod m \).
4. If \( e' < e \), loop step 2 by continue to increase \( e' \). Else, \( c \) contains the correct solution to \( c \equiv b^e \mod m \).

The code for the modulo exponentiation itself is attached below:

```verilog
always @ (posedge clk) begin
    if (reset) begin
        base_reg <= base;
        modulo_reg <= modulo;
        exponent_reg <= exponent;
        result_reg <= 64'd1;
        divide_latency_counter <= DIVIDE_LATENCY;
        state <= DIVIDING;
    end
    else case (state)
        DIVIDING : begin
            // stall until dividers are done, since division takes DIVIDE_LATENCY cycles
            if (divide_latency_counter == 7'd0) state <= UPDATE;
            else divide_latency_counter <= divide_latency_counter - 5'd1;
        end
        UPDATE : begin
            if (exponent_reg != 64'd0) begin
                if (exponent_reg[0]) result_reg <= result_next;
                base_reg <= base_next;
                exponent_reg <= exponent_next;
                divide_latency_counter <= DIVIDE_LATENCY;
                state <= DIVIDING;
            end
            else state <= HOLD;
        end
        HOLD : begin
            end
        default : state <= HOLD;
    endcase
end
```

16
4.3 Software Modules

4.3.1 VGA Modules (VGA.h)

This module consisted of code used to write pixels to the screen. It was taken from http://www.eecg.toronto.edu/~pc/courses/432/2009/projects/tft-edk.txt, compose by Andrew Shorten. A new function named vga_print was created by the team to upgrade the existing header file so that the design is able to display characters directly using the softcore processor. Essentially, we created a bit map for all letters of the alphabet and symbols into 8 by 12 bits of pixel mapping. In turn, to display certain characters/letters, the function will display the pixel data for the required character/letter.

```c
void vga_print(struct Screen* in, char* c)
{
    int i=0;
    while ( c[i] != '\0'){
        if (c[i] == '\n'){
            VGA_x_pos =0;
            VGA_y_pos = (VGA_y_pos + 12) % VGA_Y_RES;
        }else{
            write_char(in, c[i], VGA_x_pos, VGA_y_pos);
            i=i+1;
            if ( VGA_x_pos == VGA_X_RES - 8 ){
                VGA_y_pos = (VGA_y_pos + 12) % VGA_Y_RES;
            }
            VGA_x_pos = (VGA_x_pos + 8) % VGA_X_RES;
        }
    }
}
```

4.3.2 MicroBlaze Processor

All of the functionality of the processor block is contained main.c, the main software component of the system. The main responsibility of the MicroBlaze to be to facilitate the communication of data as the actual implementation of our functions is done through the hardware. The MicroBlaze writes the input data through the PLB to the MPMC into the SDRAM. Similarly, it also writes and read encrypted and decrypted data to the SDRAM through separate PLB buses.
5. Design Tree Description

**VGA_Module** - root directory

- `system.xmp` - XPS Project File
- `system.mss` - System Hardware Specification file
- `system.mhs` - System Software Specification file
- `_xps` - Option files for bitinit, libgen, simgen and platgen
- `blkdiagram` - Block diagram generated by Xilinx tools
- `data` - System Constraints File
  - `system.ucf`
- `etc` - Option files for bitgen and downloading
- `pcores` - Custom hardware peripherals
  - `miller_robin_ip`
- `c_source` - Custom software
  - `main.c`
  - `VGA.h`
References


module miller_rabin(
    start_number,
    accuracy,
    ready,
    clk,
    reset,
    prime,
    finish,
    base,
    modulo,
    exponent,
    mod_exp_reset,
    mod_exp_finish,
    mod_exp_result
);

localparam WORDSIZE = 32;

input [WORDSIZE-1:0] start_number;
input [WORDSIZE-1:0] accuracy;
input ready;
input clk;
input reset;
output reg prime;
output finish;

output reg [WORDSIZE-1:0] base;
output reg [WORDSIZE-1:0] modulo;
output reg [WORDSIZE-1:0] exponent;
output reg mod_exp_reset;
input mod_exp_finish;
input [WORDSIZE-1:0] mod_exp_result;

reg [WORDSIZE-1:0] n;
reg [31:0] accuracy_reg;
reg [WORDSIZE-1:0] r;
reg [WORDSIZE-1:0] d;
reg [WORDSIZE-1:0] s;
reg [3:0] state;
reg [31:0] k;
reg [WORDSIZE-1:0] count_to_s;

//state definitions
localparam FACTORING = 4'd1;
localparam GET_RANDOM = 4'd2;
localparam MOD_EXP_WAIT = 4'd3;
localparam CHECK = 4'd4;
localparam R_LOOP = 4'd5;
localparam HOLD = 4'd6;
assign finish = (state == HOLD) ? 1'b1 : 1'b0;

reg [31:0] baseshift;
reg [1:0] twotimes;
wire [15:0] rand_out;
wire [31:0] myrandnum = {rand_out, baseshift[15:0]};
wire [WORDSIZE-1:0] nminus2 = n - 2'd2;

wire [126:0] seed_in = {{7{16'haaaa}},15'h5aaa};

reg rand_reset;

// always
// #10 rand_out = $random(4'b1111);
// this generates a random number
rand127 myrand(
    .rand_out(rand_out),
    .seed_in (seed_in),
    .state_in(4'd0),
    .clock_in(clk),
    .reset_in(rand_reset)
);

always @ (posedge clk) begin
    if (reset) begin
        n <= start_number;
        d <= start_number - 32'd1;
        s <= 32'd0;
        k <= 32'd0;
        base <= 32'd0;
        baseshift <= 32'd0;
        modulo <= 32'b0;
        exponent <= 32'b0;
        count_to_s <= 32'd0;
        accuracy_reg <= accuracy;
        state <= FACTORING;
        mod_exp_reset <= 1'b1;
        rand_reset <= 1'b1;
        prime <= 1'b0;
        twotimes <= 2'b00;
    end
    else case (state)
    FACTORING : begin
        if (ready == 1'b1) begin
            rand_reset <= 1'b0;
            if (d[0]) begin //d % 2 == 1
                state <= GET_RANDOM;
                baseshift <= 2'd2; //rawr
            end
            else begin //d % 2 == 0
                d <= d >> 1; //d = d / 2
                s <= s + 32'd1;
            end
        end
        GET_RANDOM: begin

    end
end
if (twotimes == 2'b0) begin
    baseshift <= {16'b0,rand_out};
twotimes <= 2'b01;
end
else begin
    // need to limit base to [2,n-2]
    if ((twotimes == 2'b01) & (myrandnum < 32'd2)) begin
twotimes <= 2'b00; // redo if less than 2
end
    // shift right until in range
    else if ((twotimes == 2'b01) & (myrandnum > nminus2)) begin
baseshift <= myrandnum >> 1;
end
    else if ((twotimes == 2'b11) & (baseshift > nminus2)) begin
    end
else begin
twotimes <= 2'b00;
    base <= (twotimes == 2'b11) ? baseshift : myrandnum;
modulo <= n;
exponent <= d;
state <= MOD_EXP_WAIT;
end
end
MOD_EXP_WAIT: begin
    mod_exp_reset <= 1'b0;
    if (~mod_exp_reset & mod_exp_finish) begin
        mod_exp_reset <= 1'b1;
        // if x=1 or x=n-1 then do next loop
        if (mod_exp_result == {{(WORDSIZE-1){1'b0}},1'b1} ||
            mod_exp_result == n - 1'b1) begin
            state <= CHECK;
        end
else if (s == 0) begin
    // skip r_loop (ie. return composite)
    state <= HOLD;
    prime <= 1'b0;
end
else begin
    // do r_loop
    state <= R_LOOP;
    count_to_s <= 32'd1;
    // do x <= x^2 mod n
    base <= mod_exp_result; // base is x
    modulo <= n;
exponent <= 32'd2;
end
end
R_LOOP : begin
    mod_exp_reset <= 1'b0;
    if (~mod_exp_reset & mod_exp_finish) begin
        mod_exp_reset <= 1'b1;
end
// if x=1 then return composite
if (mod_exp_result == {(WORDSIZE-1){1'b0},1'b1}) begin
  state <= HOLD;
  prime <= 1'b0;
end

// if x=n-1 then do next loop
else if (mod_exp_result == n - 1'b1) begin
  state <= CHECK;
end

// if r_loop ends, return composite
else if (count_to_s == s) begin
  state <= HOLD;
  prime <= 1'b0;
end

// otherwise, do next r_loop
else begin
  count_to_s <= count_to_s + 1'b1;
  base <= mod_exp_result;
  modulo <= n;
  exponent <= 32'd2;
end

// check loop condition
CHECK : begin
  // last loop, signal prime
  if (k + 1'b1 >= accuracy_reg) begin
    state <= HOLD;
    prime <= 1'b1;
  end
  // next loop
  else begin
    state <= GET_RANDOM; // next loop
    k <= k + 1'b1; // increment accuracy count
  end
end

HOLD : begin
  // endless loop
end

endcase

endmodule
Appendix B

#include <stdio.h>

void eea(int a, int b, int* gcd, int* x, int* y){
    *x=0, *y=1;
    int u=1, v=0, m, n, q, r;
    *gcd = b;
    while(a!=0){
        q=(*gcd)/a;
        r=(*gcd)%a;
        m=(*x)-u*q;
        n=(*y)-v*q;
        (*gcd)=a;
        a=r;
        (*x)=u;
        (*y)=v;
        u=m;
        v=n;
    }
}

int eea(int a, int b){
    int x=0, y=1;
    int u=1, v=0, m, n, q, r;
    int gcd = b;
    while(a!=0){
        q=gcd/a;
        r=gcd%a;
        m=x-u*q;
        n=y-v*q;
        gcd=a;
        a=r;
        x=u;
        y=v;
        u=m;
        v=n;
    }
    return gcd;
}

int isprime(int p)
{
    int d;
    for (d = 2; d < p; d = d + 1)
        if (p % d == 0)
            return 0;
    return 1;
}

void keygen(int p, int q, int *e, int *d, int *n){
    //compute modulus
    *n = p*q;
    int phi = (p-1)*(q-1);
    ...
int eP;
for( eP= 3; eP<phi; eP+=2){
    if(eea(eP, phi)==1){
        *e = eP;
        break;
    }
}
*e = 17;
eP = 23;
//generate public key exponent
int dP = 0;
int s;
do{
    dP++;
    s = (dP*eP)%phi;
}while(s!=1);
*d = dP;
}
int main() {
    unsigned int *e = (int *) malloc(sizeof (int));
    unsigned int *d = (int *) malloc(sizeof (int));
    unsigned int *n = (int *) malloc(sizeof (int));
    keygen(149, 701, e, d, n);
    printf("%d %d %d \n", *e, *d, *n);
    keygen(821, 733, e, d, n);
    printf("%d %d %d \n", *e, *d, *n);
    keygen(877, 673, e, d, n);
    printf("%d %d %d \n", *e, *d, *n);
    return 0;
}