**Hardware Implementation of the Simon Encryption Algorithm**

**Simon Encryption Algorithm:**

The Simon Encryption Algorithm is a family of lightweight block ciphers which were developed in 2013 by the US National Security Agency (NSA) with the aim of developing a lightweight, flexible and easy-to-analyze cipher that offers excellent throughput, performance and low energy costs. The algorithm was designed specifically for use within the Internet of Things (IoT). Simon was designed to optimize performance in hardware implementations. The cipher was accepted by the International Organization for Standardization (ISO) in 2018 as part of the RFID air interface standard. Simon features 10 different key and plaintext size combinations giving it maximum efficiency for numerous applications. Simon employs a Balanced Feistel network and consists of up to 72 rounds for the largest key and plaintext size combination. Each combination of key and plaintext sizes is commonly referred to as Simon 2n/mn where 2n is the size of the plaintext and mn is the size of the key. Because the cipher is based off a Balanced Feistel network, the cipher is reversable and can be used to decrypt data.

In my project I have implemented the Simon encryption algorithm with a key size of 64 bits and a plaintext size of 32 bits. This version of the algorithm is referred to as Simon 32/64. I used the Basys3 FPGA board by Digilent Inc. to implement the algorithm. The board contains a Xilinx Artix-7 FPGA where the algorithm is stored. In order to program the FPGA I employed the use of the Vivado software suite developed by Xilinx Inc.

**Simon cipher full algorithm:**

In this section I will be describing the Simon cipher encryption algorithm. The decryption algorithm is very similar to the encryption algorithm with only minor differences which I will note at relevant points.

**I/Os:**

* Key. This is the key that will be used to generate a list of subkeys called a key schedule which will be used in each round of encryption. The length of the key between 64 and 256 bits
* Plaintext. This is the text that will be encrypted by the algorithm. The length of the plaintext is between 32 and 128 bits
* Ciphertext. This is the output of the algorithm. The length of the ciphertext is between 32 and 128 bits

A screenshot of a computer

Description automatically generated

Figure 1 - Simon Encryption Algorithm

**Algorithm:**

The full algorithm as seen in figure 1 consists of a chain of rounds whose individual outputs become the inputs of the next round in the chain. In addition, each round has a subkey that is matched to that round to encrypt the plaintext that is input into each round.

Each round is a function that employs the use of shifts and logic gates to encrypt the input data.

Each subkey is generated by a subkey generator function that also employs the use of shifts and logic gates to generate a string of subkeys to use for each round.

**Functions:**

**Round:**

**I/Os:**

* Subkeyi. This is one of the subkeys generated by the key schedule. Each round has a unique subkey dedicated to that round. When decrypting, the subkeys for each round are reversed meaning that for round i, the subkey that will be used is Knum\_of\_rounds – i.
* Plaintext - Ui andLi. As a balanced Feistel network, the plaintext of each round is divided into 2 parts – an Upper or Left plaintext and a Lower or Right plaintext. These 2 new plaintexts are 2 individual inputs to the round function.
* Ciphertext - Ui+1 andLi+1. As a balanced Feistel network, the ciphertext of each round is received as 2 parts – an Upper or Left ciphertext and a Lower or Right ciphertext. These 2 ciphertexts are 2 individual outputs of the round function. These values also serve as the upper and lower plaintexts for the next round function (see figure 1).

The sizes of all the inputs and outputs of the round function are identical and equal the size of the original plaintext divided by 2. This number is commonly referred to as *n*. In our case *n* = 16 bits.

**Algorithm:**

A diagram of a computer program

Description automatically generated

Figure 2 - Round Function

The round function employs the use of left shifts, an AND gate and XOR gates.

I will refer to upper and lower plaintext of round i as Ui and Li respectively and to the outputs as Ui+1 and Li+1 respectively. N-bit shift left functions will be referred to as S-N(X)

As described in figure 2, Li becomes Ui+1.

Li+1 is calculated by a bitwise AND between S-1(Ui) and S-8(Ui). This is then XORed with S-2(Ui). This result is XORed with Li and finally this result is XORed with the subkey of the current round (Ki).

**Subkey Generator:**

**I/Os:**

* Subkeyi-1. This is the previous subkey generated by the key schedule.
* Subkeyi-m. This is the subkey that is *m* spots before the current subkey that is being generated. *M* is the ratio between the size of the key and the size of the plaintext multiplied by 2. In our case *m* = 4.
* Subkeyi-3. This is the subkey that is 3spots before the current subkey that is being generated. This input only exists when *m* is equal to 4 as is our case.
* Z counter. This is an integer that is used to track which bit of the z constant should be used for each subkey generator round.
* Constant. This is equal to 0xFF..FC and its size is *n*.
* Subkeyi. This is the current subkey being generated. This is the output of the subkey generator function.

The sizes of all the subkey inputs and outputs of the round function are identical and equal the size of the original key divided by *m*. This number is commonly referred to as *n*. In our case *n* = 16 bits

A screenshot of a computer

Description automatically generated**Algorithm:**

Figure 3- Subkey Generator Function

There is a total of 3 different subkey generator functions based on the value of *n*. See figure 4. (In my diagram I refer to the subkey ki+m in figure 3 as ki.) We will be using the function for the case of *m* = 4.

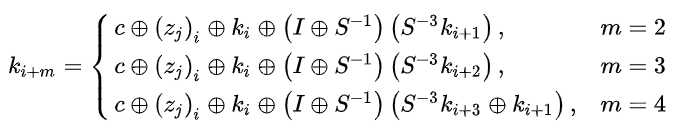
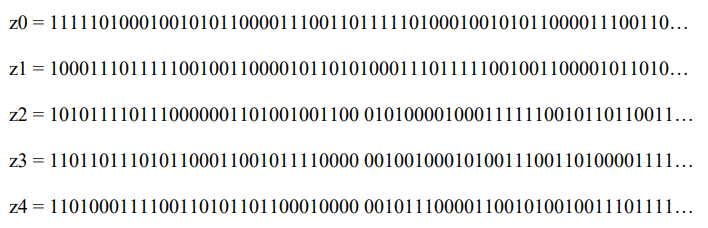


Figure 4 - Subkey Function Equation

In each of the algorithms, the first *m* subkeys are generated by dividing the original key into *m* parts. Each subkey will therefore be *n* bits. In our case *m* = 4 and *n* = 16 bits.

There are a total of five possible z constants that are used for different sizes of the cipher.



Simon 32/64 uses Z0 as its z constant.

I will refer to subkey inputs as Ki-x and the output as Ki. N-bit shift right functions will be referred to as SN(X). The current bit of the z constant will be Zj and the constant will be *c*.

As described in figure 3, Ki is calculated by performing a XOR on S3(Ki-1) and Ki-3. This value is then shifted right by 1 bit and XORed with its original value. This result is XORed with Ki-4 and then XORed with Zj and *c*.

**UART Connection:**

In order to use the Basys3 to encrypt data we need to input data onto the board and into the algorithm. The Basys3 contains 16 switches that can be used to input data. Simon 32/64 requires at least 96 bits of data to run. 32 bits for the plaintext and 64 for the key, in addition to any other data that needs to be inputted manually to the algorithm. Aside from the fact that the Basys3 doesn’t have enough external inputs, ideally, we would be able to input data from a computer or a different external data source. For this we will use a UART (Universal Asynchronous Receiver Transmitter) to input and output data to the Basys3 and its FPGA.

The UART modules that I implemented using VHDL include modules for both receiving (RxD) and transmitting (TxD) data. Each module uses a state machine that receives or sends 8 bits of data at one time. This way we can adhere to the standard practice of sending and receiving one byte of data at a time from standard terminals such as RealTerm or the pySerial library.

In total we need to receive at least 96 bits or 12 bytes of data and send 32 bits or 4 bytes of data. Since the Rx and Tx modules only receive and send 8 bits or 1 byte at a time, we need a way to receive and store 12 bytes of data at once and send 4 bytes of data at once. To do this I have included modules that concatenate every byte of data that is received into 1 big register which is later sent to the appropriate inputs of the cipher and divide the data that is to be sent into singular bytes so that they can be outputted to the computer.

**TxD:**

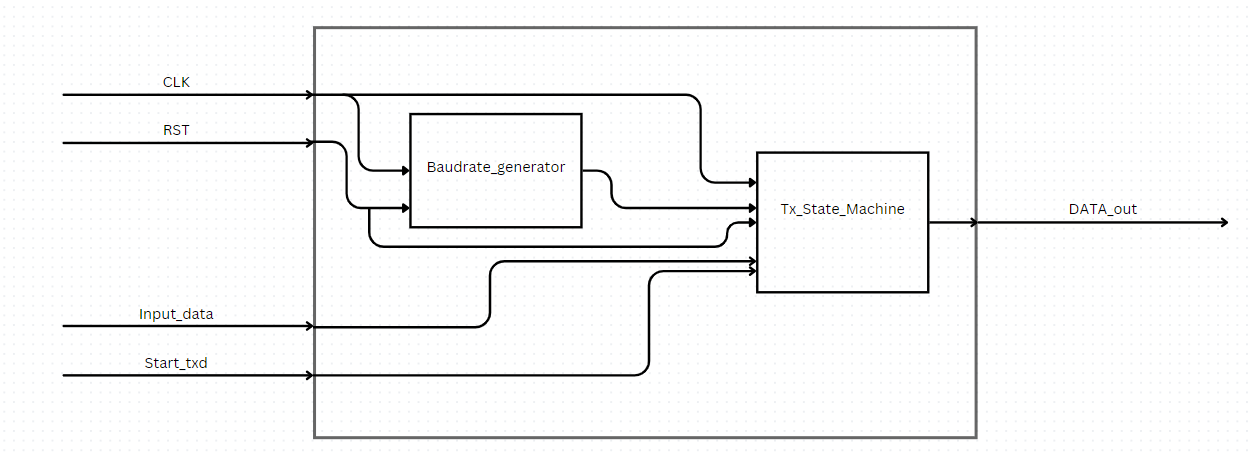


Figure 5 - TxD Module

Figure 5 shows the block diagram of the TxD module.

**Baudrate Generator:**

The computer processes data at a frequency (in bits per second) called a baudrate. The standard baudrates are 4800, 9600, 19200, 38400, 57600, 115200, 230400, 460800, 921600 bits per second. In my project I am using a baudrate of 9600 bps. The Basys3 contains an internal clock that runs at a frequency of 100MHz. In order to allow accurate data transfer between the computer and the Basys3 we need to synchronize the computer’s baudrate and the Basys3’s internal clock. For this we will create a “baudrate clock” which will activate the state machine 9600 times per second. This will allow me to transmit data using the Basys3 at a rate of 9600 bits per second which will allow accurate transmission to the computer.

**State Machine:**

The state machine is made up of four states. IDLE, START, TRANSMIT and FINISH

* IDLE: In this state we are waiting for the algorithm to tell us to start transmitting
* START: In this state we send a ‘0’ to the computer as a start bit and load the input data into the state machine and move to the TRANSMIT stage.
* TRANSMIT: In this state we go through the data using a 3 bit counter and send 1 bit at every tick of the baud clock until all of the bits have been sent.
* FINISH: In this state we send a ‘1’ to the computer as a stop bit and go back to the IDLE state

A diagram of a diagram

Description automatically generated

Figure 6- TxD state machine

**RxD:**

A diagram of a diagram

Description automatically generated

Figure 7 - RxD module

Figure 7 shows the block diagram of the RxD module.

**Baudrate Generator:**

As explained above, in order to synchronize between the clock of the Basys3 and the clock of the terminal we need to create a baud clock. In order to receive the most accurate reading of the current bit being transmitted, we want to sample the bit in the middle of the current pulse or bit being sent. To do this we will create an additional baud clock for sampling the current bit, which is sixteen times the frequency of the baud clock in the TxD module. This way we will be able to sample the current bit with high accuracy.

**State Machine:**

The state machine is made up of four states. IDLE, START, RECEIVE, FINISH\_RX and RDY\_TX

* IDLE: In this state we are waiting for the algorithm to tell us to start receiving
* START: When the start bit is detected, we receive it to check that it is a ‘0’ and that it is actually a start bit and not just a glitch.
* RECEIVE: In this state we receive the data one bit at a time and put it into an 8-bit register.
* FINISH\_RX: In this state we move onto the next state.
* RDY\_TX: In this state we raise a flag that signifies that the current bit has been received and we go back to the IDLE state.

A diagram of a diagram

Description automatically generated

Figure 8 - RxD state machine

**Python GUI:**

I built a python application which is used as a wrapper for the hardware implementation. The python module employs the use of the Tkinter python library to create a GUI to be used as a simple and intuitive interface for entering data and receiving encrypted data through the computer. The python module utilizes the pySerial library to connect to the serial port of the computer and through this to the Basys3 board.

When data is received from the user, the module formats the data to make sure that the plaintext and key are the correct size for the cipher. This is either done through the use of automatic left zero padding or error messages to tell the user to input different data. The code then converts each character of data entered to byte form through UTF-8 encoding. This allows us to send the information as a string of binary numbers to the Basys3. After receiving the ciphertext, the module will then convert the ciphertext into a string of characters which will be presented on the screen.

**Implementation:**

To implement the design onto the Basys3 board, I used the Vivado Design Suite (2023.2) by Xilinx AMD for multiple tasks such as design synthesis, implementation, placement and bitstream generation.

**Synthesis and Implementation:**

Figure 9 is an RTL description of the top level of the design created by the Vivado Design Suite. The VHDL code is converted to logic gates and wires that connect them. This is optimized for performance, power consumption, space and other things. The RTL description is used as a high-level model for actual implementation onto the Basys3 board.

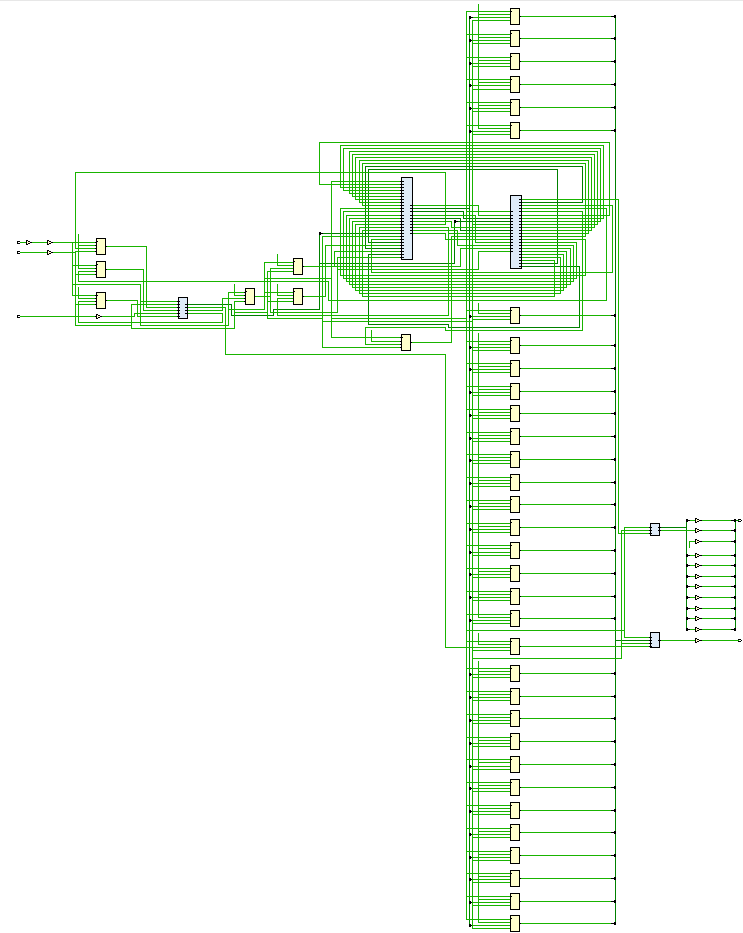
**

Figure 9 - RTL schematic

**Placement:**

Figure 10 represents the physical placement of different elements on the FPGA of the Basys3 board. The elements include Flip-Flops, LUTs, wires, I/Os and other programmable elements.

A screen shot of a computer

Description automatically generated

Figure 10- Placement on Basys3 FPGA

**Package:**

Figure 11 describes the I/O ports on the Basys3 board including the clock input, seven segent outputs and the UART I/O pins.

*A screenshot of a video game

Description automatically generated*

Figure 11- I/O planning on Basys3

A computer with a circuit board

Description automatically generated