

A logo with a triangle and text

Description automatically generated with medium confidence

הפקולטה להנדסה ומדעי המחשב

החוג להנדסת חשמל ואלקטרוניקה

פרויקט גמר באלקטרוניקה

**מימוש בחומרה של אלגוריתם ההצפנה Simon**

Hardware Implementation of the Simon

Encryption Algorithm

|  |  |  |
| --- | --- | --- |
| מגיש: | ליאור ברנר | תשרי תשפ"ד  ירושלים |
| מנחה: | מר אורי שטרו |

# Abstract

The Simon encryption algorithm was 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. Along with Simon, The NSA developed the Speck algorithm, which is designed for optimal performance in software implementations. Both ciphers were accepted by the International Organization for Standardization (ISO) in 2018 as part of the RFID air interface standard.

The Simon algorithm is uses a Feistel network to encrypt data over rounds. The algorithm uses the user defined key to create a key schedule which is used for both encryption and decryption. There are 10 different combinations of Plaintext and Key sizes that can be used for the Simon cipher. This gives the cipher a lot of flexibility when it comes to security versus “heaviness” of the algorithm. Each combination of Plaintext and Key size employs a different number of rounds and possibly a slight difference in the algorithm used to generate the key schedule.

In order to implement the algorithm, I used the Basys3 board which comes with a Xilinx AMD FPGA chip. An FPGA is an integrated circuit that can be reprogrammed to have its hardware changed after it is created. Its hardware is designed using a Hardware Description Language (HDL), most commonly VHDL or Verilog. One of the great advantages of FPGA is that many operations can be done in parallel thus making it more time efficient. Since the hardware of the FPGA can be changed, it is good at performing various and unrelated tasks efficiently as per the user's need.

Table of Contents

[Abstract 1](#_Toc181964480)

[1 Introduction 5](#_Toc181964481)

[2 Cryptographic Theory 6](#_Toc181964482)

[3 Simon Cipher Algorithm 9](#_Toc181964483)

[4 Implementation 13](#_Toc181964484)

[4.1 Cipher Algorithm 13](#_Toc181964485)

[4.1.1 Block Diagram 13](#_Toc181964486)

[4.1.2 Main Modules 14](#_Toc181964487)

[*4.1.3* Python Controller 34](#_Toc181964488)

[5 Basys3 36](#_Toc181964489)

[5.1 Specs 36](#_Toc181964490)

[5.1.1 I/Os 36](#_Toc181964491)

[5.1.2 FPGA 38](#_Toc181964492)

[5.2 Tech Implementation 38](#_Toc181964493)

[5.2.1 JTAG 39](#_Toc181964494)

[5.2.2 USB 39](#_Toc181964495)

[6 Raspberry Pi 40](#_Toc181964496)

[7 Vivado 40](#_Toc181964497)

[8 Area and timing section (Vivado) 41](#_Toc181964498)

[8.1 Area Reports 41](#_Toc181964499)

[8.1.1 VHDL 41](#_Toc181964500)

[8.1.2 Verilog 44](#_Toc181964501)

[8.2 Timing Reports 48](#_Toc181964502)

[8.2.1 VHDL 48](#_Toc181964503)

[8.2.2 Verilog 49](#_Toc181964504)

[9 Discussions 49](#_Toc181964505)

[9.1 Simulation Results 49](#_Toc181964506)

[9.2 Actual Results 51](#_Toc181964507)

[9.3 Comparison of Verilog and VHDL 52](#_Toc181964508)

[9.3.1 Area 52](#_Toc181964509)

[9.3.2 Timing and Throughput 53](#_Toc181964510)

[9.3.3 Discrepancies and Explanations 54](#_Toc181964511)

[10 Conclusions 55](#_Toc181964512)

[11 Bibliography 56](#_Toc181964513)

[12 Appendix A 57](#_Toc181964514)

[13 Appendix B 58](#_Toc181964515)

Table of Figures

[Figure ‎3.1 - Diagram of the round function 10](#_Toc182343102)

[Figure ‎3.2 - U, V and W matrices for generating the Z constants 11](#_Toc182343103)

[Figure ‎3.3 - The five Z constants 11](#_Toc182343104)

[Figure ‎3.4 - Block diagrams of the subkey generator function for m = 2, 3 and 4 12](#_Toc182343105)

[Figure ‎4.1 - Block diagram of the system 13](#_Toc182343106)

[Figure ‎4.2 - I/Os for the round function 14](#_Toc182343107)

[Figure ‎4.3 - Technology schematic of the round function 15](#_Toc182343108)

[Figure ‎4.4 - RTL schematic of the round function 15](#_Toc182343109)

[Figure ‎4.5 - I/Os for subkey generator 16](#_Toc182343110)

[Figure ‎4.6 - Diagram of the subkey generator algorithm 16](#_Toc182343111)

[Figure ‎4.7 - Technology schematic of the subkey generator 17](#_Toc182343112)

[Figure ‎4.8 - RTL schematic of the subkey generator 17](#_Toc182343113)

[Figure ‎4.9 – I/Os for key schedule generator 18](#_Toc182343114)

[Figure ‎4.10 - First 4 subkeys 18](#_Toc182343115)

[Figure ‎4.11 - I/Os for encryption 20](#_Toc182343116)

[Figure ‎4.12 – Algorithm for the encryption module 20](#_Toc182343117)

[Figure ‎4.13 - I/Os for TxD module 23](#_Toc182343118)

[Figure ‎4.14 - Block diagram of TxD module 23](#_Toc182343119)

[Figure ‎4.15 - Block diagram of the TxD state machine 24](#_Toc182343120)

[Figure ‎4.16 - Technology schematic of TxD module 25](#_Toc182343121)

[Figure ‎4.17 - RTL schematic of TxD module 25](#_Toc182343122)

[Figure ‎4.18 - I/Os for RxD module 26](#_Toc182343123)

[Figure ‎4.19 - Block diagram of the RxD module 26](#_Toc182343124)

[Figure ‎4.20 - Block diagram of the RxD state machine 26](#_Toc182343125)

[Figure ‎4.21 - RTL schematic of the RxD module 28](#_Toc182343126)

[Figure ‎4.22 - I/Os for send data module 28](#_Toc182343127)

[Figure ‎4.23 - Algorithm in code of the Send Data module 29](#_Toc182343128)

[Figure ‎4.24 RTL schematic of Send Data module 30](#_Toc182343129)

[Figure ‎4.25 - I/Os for get data module 31](#_Toc182343130)

[Figure ‎4.26 - Algorithm in code of the Get Data module 31](#_Toc182343131)

[Figure ‎4.27 - RTL schematic of Send Data module 32](#_Toc182343132)

[Figure ‎4.28 - I/Os for seven segment data 32](#_Toc182343133)

[Figure ‎4.29 - Refresh period of Seven Segment Display 33](#_Toc182343134)

[Figure ‎4.30 - Code to open serial port 34](#_Toc182343135)

[Figure ‎4.31 - Code to open TKinter window 34](#_Toc182343136)

[Figure ‎5.1 - Overview of the Basys3 board 35](#_Toc182343137)

[Figure ‎5.2 - I/Os on the Basys3 board 36](#_Toc182343138)

[Figure ‎5.3 - Pmod ports 36](#_Toc182343139)

[Figure ‎5.4 - Basys3 configuration options 37](#_Toc182343140)

[Figure ‎8.1 – VHDL device technology placement 43](#_Toc182343141)

[Figure ‎8.2 – Verilog device technology placement 46](#_Toc182343142)

[Figure ‎8.3 - I/O placement on Basys3 board 47](#_Toc182343143)

[Figure ‎9.1 - Simulation of input wave 48](#_Toc182343144)

[Figure ‎9.2 - Simulation of cipher start after all data has been received 48](#_Toc182343145)

[Figure ‎9.3 - Simulation of ciphertext 49](#_Toc182343146)

[Figure ‎9.4 - Simulation of decrypting the ciphertext from above 49](#_Toc182343147)

[Figure ‎9.5 - Encryption and decryption using Raspberry Pi and Basys3 50](#_Toc182343148)

[Figure ‎9.6 - Encryption and decryption of multiple bytes using Raspberry Pi and Basys3 50](#_Toc182343149)

Table of Tables

[Table ‎4.1 – Design signals and their physical location on the Basys3 board 34](#_Toc182428755)

[Table ‎5.1 - Artix-7 FPGA family data 37](#_Toc182428756)

[Table ‎8.1 - Post synthesis area utilization 41](#_Toc182428757)

[Table ‎8.2 - Post implementation area utilization 41](#_Toc182428758)

[Table ‎8.3 - Full post implementation area utilization 43](#_Toc182428759)

[Table ‎8.4 - Post synthesis area utilization 44](#_Toc182428760)

[Table ‎8.5 - Post implementation area utilization 44](#_Toc182428761)

[Table ‎8.6 - Full post implementation area utilization 46](#_Toc182428762)

[Table ‎8.7 - Timing report table for VHDL implementation 48](#_Toc182428763)

[Table ‎8.8 - Timing report table for Verilog implementation 48](#_Toc182428764)

[Table ‎9.1 – Comparison of FFs and LUTs for the VHDL and Verilog implementations 52](#_Toc182428765)

[Table ‎9.2 – Comparison of Max Frequency and Throughput 53](#_Toc182428766)

# Introduction

Data security is one of the most important aspects of our time. Efficient and secure data transfer is critical in many cases. In order to transfer data securely, many different cryptographic algorithms are used. Included among these is the Advanced Encryption Standard (AES) which is currently the standard encryption algorithm for many organizations including the US government. The drawback to these algorithms is the fact that they require a lot of time, energy and memory to operate effectively. When dealing with more powerful machines such as computers this is fine but when we want to encrypt data in machines with lesser technological capabilities these algorithms become inefficient.

The IoT is a system of devices with communication technology which allows them to connect and communicate with each other. These devices can range from smartphones to laptops to lighting systems and more. Many of these devices do not have the ability to encrypt data with AES or similar algorithms as the technology that they use to communicate has limited abilities. Therefore, it is important to develop an encryption algorithm that requires very little power and memory in order to operate efficiently. For this the NSA created the Simon and Speck algorithms.

Field programmable gate arrays (FPGAs) are widely used to implement hardware circuitry. As the name suggests, FPGAs are programmable chips that are able to be reprogrammed from afar. This gives them a lot of flexibility in terms of their functionality. One of the advantages of FPGAs is their parallel processing which allows them to run multiple operations at once. FPGAs are optimized for hardware implementations that use many logic gate based operations. When designing a chip to do one specific thing such as encrypt data, with the ability to update or even completely change the existing design, the FPGA is one of the best options.

The main goal of the project is to implement the Simon Cipher Algorithm onto an FPGA using the Xilinx-AMD Basys3 board. To do this I employ the use of many other technologies and programs such as Rasberry Pi, Modelsim, Vivado and more.

Among the other goals of the project is to become proficient in using these technologies and to understand how they work for future projects. In addition I implemented the cipher in both VHDL and Verilog in order to both gain proficiency in theses languages and to compare the performance of both languages. The main areas I am interested in are area used and timing, including throughput.

# Cryptographic Theory

Cryptography [1] is a method of protecting information and communications using codes, so that only those for whom the information is intended can read and process it.

In computer science, cryptography refers to secure information and communication techniques derived from mathematical concepts and a set of rule-based calculations called [algorithms](https://www.techtarget.com/whatis/definition/algorithm), to transform messages in ways that are hard to decipher. These deterministic algorithms are used for cryptographic key generation, digital signing, verification to protect data privacy, web browsing on the internet and confidential communications such as credit card transactions and email.

**Cryptography techniques**

Cryptography is closely related to the disciplines of [cryptology](https://www.techtarget.com/searchsecurity/definition/cryptology) and [cryptanalysis](https://www.techtarget.com/searchsecurity/definition/cryptanalysis). It includes techniques such as microdots, merging words with images and other ways to hide information in storage or transit. In today's computer-centric world, cryptography is most often associated with scrambling [plaintext](https://www.techtarget.com/searchsecurity/definition/plaintext) (ordinary text, sometimes referred to as *cleartext*) into [ciphertext](https://www.techtarget.com/whatis/definition/ciphertext) (a process called [encryption](https://www.techtarget.com/searchsecurity/definition/encryption)), then back again (known as decryption). Individuals who practice this field are known as cryptographers.

Modern cryptography concerns itself with the following four objectives:

1. **Confidentiality.** The information cannot be understood by anyone for whom it was unintended.
2. **Integrity.**The information cannot be altered in storage or transit between sender and intended receiver without the alteration being detected.
3. **Non-repudiation.** The creator/sender of the information cannot deny at a later stage their intentions in the creation or transmission of the information.
4. **Authentication.** The sender and receiver can confirm each other's identity and the origin/destination of the information.

Procedures and protocols that meet some or all the above criteria are known as cryptosystems. Cryptosystems are often thought to refer only to mathematical procedures and computer programs; however, they also include the regulation of human behavior, such as choosing hard-to-guess passwords, logging off unused systems and not discussing sensitive procedures with outsiders.

Types of encryption algorithms

**Single-key or symmetric-key encryption algorithms** create a fixed length of bits known as a [block cipher](https://www.techtarget.com/searchsecurity/definition/block-cipher) with a secret key that the creator/sender uses to encipher data (encryption) and the receiver uses to decipher it. One example of [symmetric-key cryptography](https://www.techtarget.com/searchsecurity/feature/Cryptography-basics-Symmetric-key-encryption-algorithms) is the Advanced Encryption Standard ([AES](https://www.techtarget.com/searchsecurity/definition/Advanced-Encryption-Standard)). AES is a specification established in November 2001 by the National Institute of Standards and Technology (NIST) as a Federal Information Processing Standard (FIPS 197) to protect sensitive information. The standard is mandated by the U.S. government and widely used in the private sector.

**Public-key or asymmetric-key encryption algorithms** use a pair of keys, a public key associated with the creator/sender for encrypting messages and a private key that only the originator knows (unless it is exposed or they decide to share it) for decrypting that information. With this technique one person can encrypt messages for many people with only one key and each person will have his own distinct private key to decrypt the message.

To maintain data integrity in cryptography**,**[**hash functions**](https://www.techtarget.com/searchdatamanagement/definition/hashing)**,** which return a deterministic output from an input value, are used to map data to a fixed data size.

The **Simon Cipher Function** is a single key encryption algorithm. This means that the sender and receiver use the same key to encrypt and decrypt (respectively) the data being transferred.

There [2] are two types of symmetric encryption algorithms:

1. **Block algorithms.** Set lengths of bits are encrypted in blocks of electronic data with the use of a specific secret key. As the data is being encrypted, the system holds the data in its memory as it waits for complete blocks.
2. **Stream algorithms.** Data is encrypted as it streams instead of being retained in the system’s memory.

Our [3] cipher is a block algorithm meaning that it encrypts one block of data at a time. The cipher accepts a whole block of data including a key and a plaintext and then encrypts the data. When dealing with multiple blocks of data, for example when the cipher encrypts 32 bits of data at once and the message is 128 bits long, there are techniques for securing the data even more called block cipher modes of operation. A mode of operation describes how to repeatedly apply a cipher's single-block operation to securely transform amounts of data larger than a block. The input of each new block is the new block of plaintext XORed with some combination of the result and or input of the previous block depending on the mode. The first plaintext block is traditionally XORed a bitstring commonly referred to as an Initialization Vector (IV) before being input into the cipher itself.

**Design**

A [4] block cipher can be seen as a set of permutations on a plaintext influenced by a secret key. The goal is to create an efficient block cipher that on the one hand is easy to implement, understand and reverse, and on the other hand is difficult for an adversary to find any relationship between plaintext, ciphertext and the secret key. Nowadays, most block cipher found in practice use simple building blocks which are combined to get a complex function.

The Simon cipher algorithm, like most modern ciphers, is an iterative block cipher. This means that the data undergoes many transformations by putting it through many iterations of the same function. This function is often called a round. This is an algorithm which maps a plaintext of fixed size into a fixed size ciphertext using a key, by repeatedly applying a round transformation to the plaintext.

The main difficulty for a designer is now to choose a round function and method to factor in the key to get a complex relationship between the plaintext, ciphertext and key. The two most common ways to achieve this are so-called Feistel networks or Substitution Permutation Networks (SPNs).

The Simon cipher algorithm employs a Feistel Network in order to encrypt data. The [5] Feistel network is one of the most commonly used structures in iterated block ciphers. One round of a Feistel network is defined in Equation ‎2.1:

Equation ‎2.1

where (P1, P2) is the data input to the round, K is the round key and (C1, C2) is the data output of the round taken as input to the next round. The Feistel structure provides invertible transformations independently of whether the round function F is invertible or not and can be used to generate random permutations from random functions. P1 and P2 are two parts of the original plaintext or current plaintext if the plaintext has already been through rounds. After the last round, the outputs C1 and C2 are combined to produce the final ciphertext. When the sizes of P1 and P2 are equal, this is called a balanced Feistel Network. When they are not equal in size it is called an unbalanced Feistel Network.

Each round in a Feistel Network has an associated subkey. The group of subkeys is called a Key Schedule. The key schedule is generally generated by splitting up the original key into smaller subkeys and performing XOR, rotating, permutations and other operations on those subkeys and the newly generated subkeys. Each subkey then gets assigned to one of the rounds and the subkey is referred to as the round key.

There are many advantages for the Feistel Network including reversibility even when using a non-invertible round function, the ability to add and subtract as many rounds as needed depending on the security specifications and the simplicity of implementation.

The Simon cipher employs a balanced Feistel network. The number of rounds used depends on the size of the block. The Simon cipher is very flexible in terms of its ability to encrypt many different sizes of plaintexts. The size of a plaintext block can vary from 32 bits to 128 bits and the size of the key varies from 64 bits to 256 bits. Each unique plaintext key size combination will employ a different number of rounds.

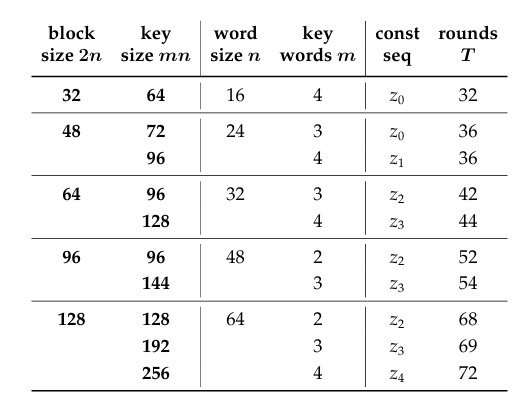
The Simon Cipher Encryption Algorithm is an encryption algorithm developed by the NSA in 2013 to be used in circuits where there are hardware limitations and bigger more standard encryption algorithms such as AES are to big to be implemented effectively on the specific technology. To give as much flexibility as possible, the cipher is able to be implemented with ten different plaintext and key size combinations.

Figure ‎2.1 - Characteristics of each variant of the Simon Cipher

The [6] Simon block cipher with an *n*-bit word (and hence a 2*n*-bit block) is denoted Simon2n, where *n* is required to be 16, 24, 32, 48 or 64 as described in Figure ‎2.1 as seen in source [6]. Simon2*n* with an *m*-word (*mn*-bit) key will be referred to as Simon2*n*/*mn*. For example, Simon64/128 refers to the version of Simon acting on 64-bit plaintext blocks and using a 128-bit key. Each instance of Simon uses the familiar Feistel rule of motion. The algorithm is engineered to be extremely small in hardware and easy to serialize at various levels, but care was taken so as not to sacrifice software performance.

When implementing the cipher, the round function inputs must be equal length. Since Simon uses a balanced Feistel network, the two plaintext inputs will be half the length of the original plaintext. This length is referred to as n and thus the variant of the cipher is Simon2*n*. Each round also receives a subkey of the same length. As explained earlier, the key length is *m* times *n* where *m* is equal to 2, 3 or 4.

# Simon Cipher Algorithm

**I/Os**

Plaintext – Data to be encrypted. Can be 32, 48, 64, 96 or 128 bits.

Key – Key that is used as part of the encryption process. Can be 64, 72, 96, 128, 144, 192 or 256 bits depending on the plaintext size. See Figure ‎2.1 for details.

Ciphertext – Encrypted data that is the same size as the Plaintext.

The Plaintext is divided into 2 equal parts and loaded into the first round. The plaintext then goes through many iterations of the round function (see Figure ‎3.1) and at the end the two outputs of the final round function are combined to make the ciphertext.

**Round Function**

The round function employs the use of the XOR operator, AND operator and left circular shift.

**Algorithm:**

A diagram of a algorithm

Description automatically generated

Figure ‎3.1 - Diagram of the round function

Figure ‎3.1, seen in source [6], presents a diagram of the 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 (Shown in Figure ‎3.1 as xi+1 and xi) of round i as Ui and Li respectively and to the outputs (Shown in Figure ‎3.1 as xi+2 and xi+1) as Ui+1 and Li+1 respectively. N-bit shift left functions will be referred to as S-N(X)

As described in Figure ‎3.1, Li becomes Ui+1.

As described in Equation ‎3.1, 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).

Equation ‎3.1

The round function is the same for every variant of the Simon Cipher.

The number of rounds that are used for each variant is shown in Figure ‎2.1.

**Key Schedule**

The Key Schedule is made up of subkeys equal to the number of rounds.

The first subkeys are generated by dividing the key into *m* parts. The subsequent subkeys are generated using one of three distinct algorithms depending on *m*.

Each of the three possible algorithms, which I refer to as subkey generators, makes use of the XOR operator and left circular shift operator.

Each subkey generator uses a constant called c which equals 0xFFFF…C. The size of the constant is equal to the size of the subkeys.

In addition, each subkey generator uses a bit string called a Z Constant. The (zj)i element in the formula refers to one bit in the z constant used for each cipher. In order to generate each z constant, we use a Linear Feedback Shift Register (LFSR). Each z constant is a repeating series of 31 bits. There are 5 distinct Z constants, and each variant of the cipher uses one of the 5 (see Figure ‎2.1).

A number and equation in a square

Description automatically generated with medium confidenceFigure ‎3.2, seen in source [6], shows three distinct LFSRs denoted as u, v and w to create three unique sequences. Z0 and Z1 are equivalent to u and v. Z2, Z3 and Z4 are created by taking u, v and w respectively and XORing them with the 2 period sequence of 01010101….

Figure ‎3.2 - U, V and W matrices for generating the Z constants

Each LFSR is generated by initializing a 5-bit vector to 00001 and iterating the vector over each matrix and the rightmost bit is the next bit in the string. The formula for finding the ith element is Equation ‎3.2:

Equation ‎3.2

Where z refers to the Z constant string and U refers to the matrix being used.

Figure ‎3.3, seen in source [6], A number of binary code

Description automatically generated with medium confidencepresents the five Z constants that are used for different sizes of the cipher.

Figure ‎3.3 - The five Z constants

The number of bits of the Z constant that are actually used depends on the variant of the cipher. For example, the Simon32/64 variant generates 28 new subkeys and therefore we only need to use 28 bits of the Z constant.

**Subkey Generators:**

**A group of black text

Description automatically generated**

Equation ‎3.3

In Equation ‎3.3, seen in source [6], ki+m refers to the subkey currently being generated and kx is a previous subkey already generated. c is the constant referenced above and (zj)i is the current bit in the Z constant. S-x represents a left rotation of x bits.

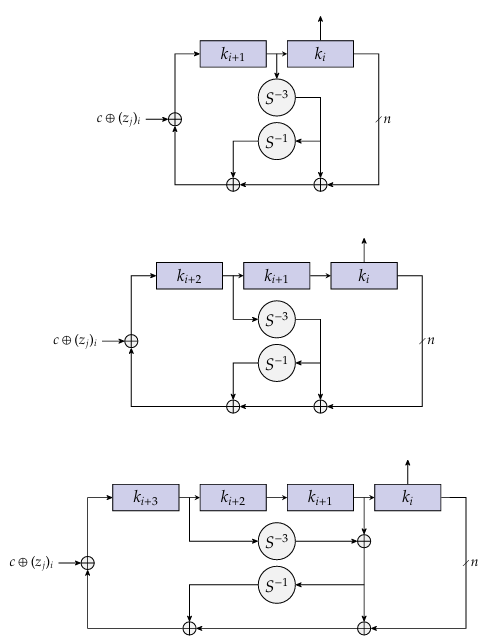
Figure ‎3.4, seen in source [6], shows the block diagram for generating subkeys based on *m*.

Figure ‎3.4 - Block diagrams of the subkey generator function for m = 2, 3 and 4

**This Project’s Simon Variant**

In this project I implemented the Simon32/64 variant. Therefore, *n* is equal to 16 and *m* is equal to 4. The Z constant used is Z0 and I use the subkey generator for *m* = 4. For this variant there are 32 rounds.

# Implementation

## Cipher Algorithm

### A diagram of a computer Description automatically generatedBlock Diagram

A computer screen with a circuit board and a circuit board

Description automatically generated with medium confidence

Figure ‎4.1 - Block diagram of the system

General Overview:

Figure ‎4.1 shows us the block diagram of the overall system. The Raspberry Pi sends the initial data to the Basys3. This data contains the plaintext, the key and the encryption/decryption setting. The final encrypted data is sent from the Basys3 back to the Raspberry Pi and is outputted on screen. Note: The diagram uses “Plaintext block” to refer to both plaintext or ciphertext depending on if the system is encrypting or decrypting the data.

### Main Modules

#### Simon Cipher

The cipher is the main part of the project. This is what encrypts the entered data. The cipher is implemented using four submodules. The round function, the subkey generator, the key schedule generator and the cipher algorithm itself which combines the other three modules to form the cipher.

##### Modules

For each module I will be presenting:

* The I/Os of the module and a short explanation for each I/O including purpose, number of bits and other general knowledge. Each module contains a high synchronous reset and a 100MHz clock that I didn’t include in the explanation of each module due to redundancy.
* The algorithm used - including a block diagram and written explanation.
* The technology schematic of the I/Os of the module.
* The RTL schematic of the module.

###### 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).

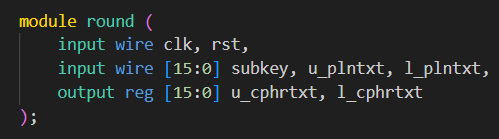
The sizes of all the inputs and outputs of the round function, as seen in Figure ‎4.2, are identical and equal the size of the original plaintext divided by 2 which is 16.

Figure ‎4.2 - I/Os for the round function

Figure ‎4.3shows technology schematic of the round function.

**A screenshot of a computer

Description automatically generated**

Figure ‎4.3 - Technology schematic of the round function

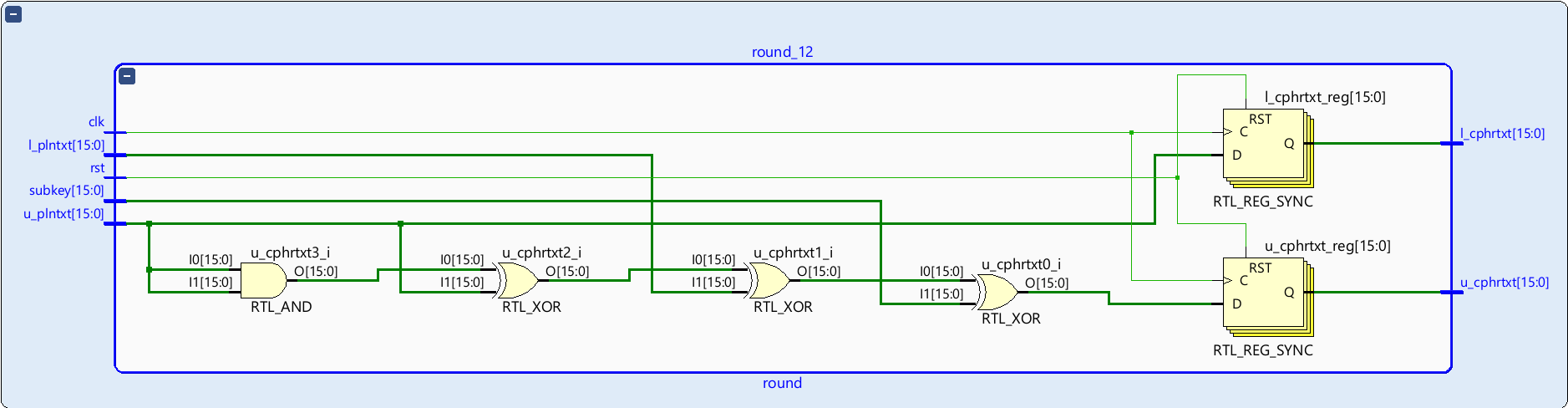
Figure ‎4.4 shows the RTL schematic for the round function.

Figure ‎4.4 - RTL schematic of the round function

###### 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 0xFFFC.
* Subkeyi. This is the current subkey being generated. This is the output of the subkey generator module.

The sizes of all the subkey inputs and outputs of the subkey generator module, as seen in Figure ‎4.5, are identical and equal the size of the original key divided by *m*. In our case 16 bits.

A computer code on a black background

Description automatically generated

Figure ‎4.5 - I/Os for subkey generator

**A screenshot of a computer

Description automatically generatedAlgorithm:**

Figure ‎4.6 - Diagram of the subkey generator algorithm

The algorithm shown in Figure ‎4.6 is the algorithm for our case of *m* = 4.

As explained above in Figure ‎2.1, Simon32/64 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 Equation ‎4.1, 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*.

Equation ‎4.1

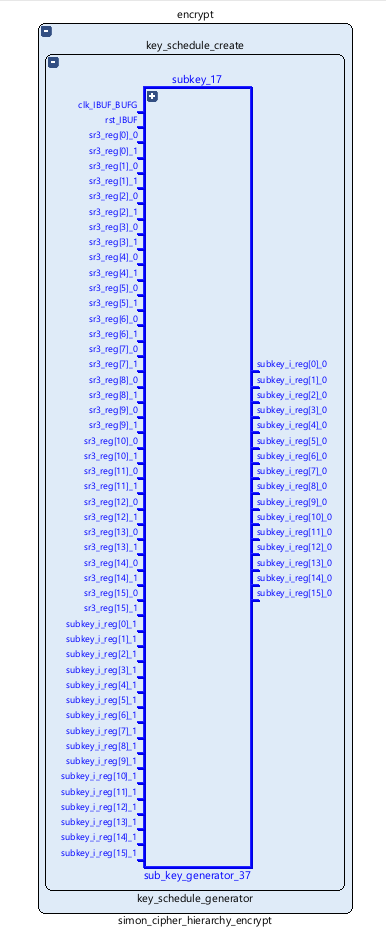
Figure ‎4.7 shows the technology schematic for the subkey generator module.

Figure ‎4.7 - Technology schematic of the subkey generator

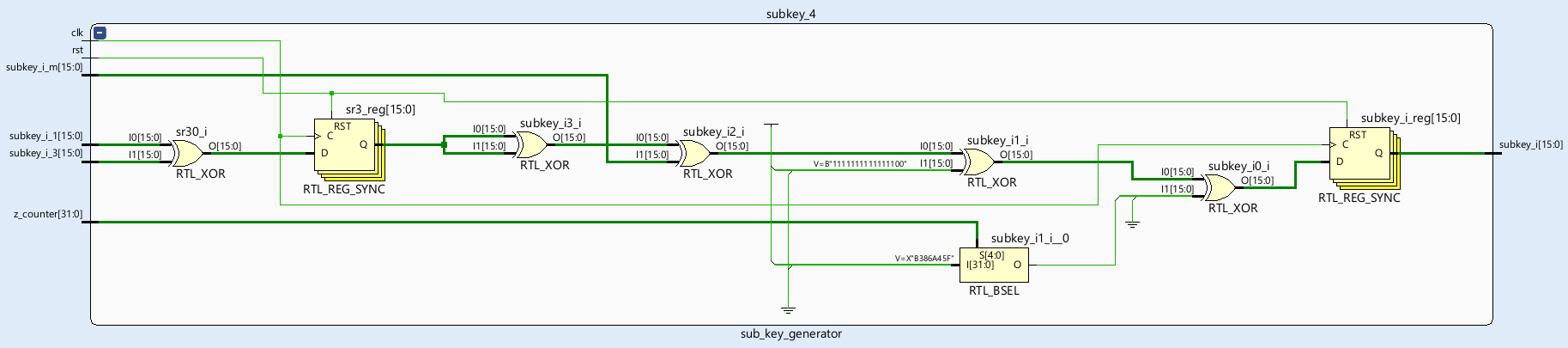
Figure ‎4.8 shows the RTL schematic for the subkey generator module.

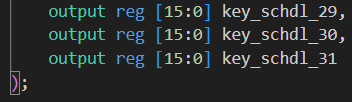
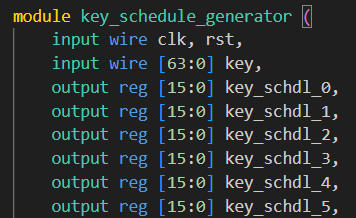
Figure ‎4.8 - RTL schematic of the subkey generator

###### Key Schedule Generator

**I/Os:**

* Key. This is the key that is used for the cipher.
* Subkeys. These are the 32 subkeys that are used for the round functions. These are the outputs of the module.

As seen in Figure ‎4.9, the size of the key is 64 bits. All the subkey outputs of the round function are equal to the size of the original key divided by *m*. In our case each subkey is 16 bits.



**…**

Figure ‎4.9 – I/Os for key schedule generator

**Algorithm:**

The first *m* = 4 subkeys are generated by taking the key and dividing it so that each subkey is *n* = 16 bits long.

A black text with black letters

Description automatically generated with medium confidence

Figure ‎4.10 - First 4 subkeys

In our case, where the key is 64 bits and *m* = 4, bit 63 represents the MSB of the key such that the 4th subkey is the 16 left-most bits of the key and the 1st subkey contains the rightmost bits of the key. See Figure ‎4.10.

Each subsequent subkey, in our case up to the 32nd, is generated using the subkey generator module that is described above.

Due to the size of the Technology Schematic, Figure ‎4.11 shows a section of the schematic that includes all of the inputs and some of the outputs. The full schematic can be accessed using Vivado.

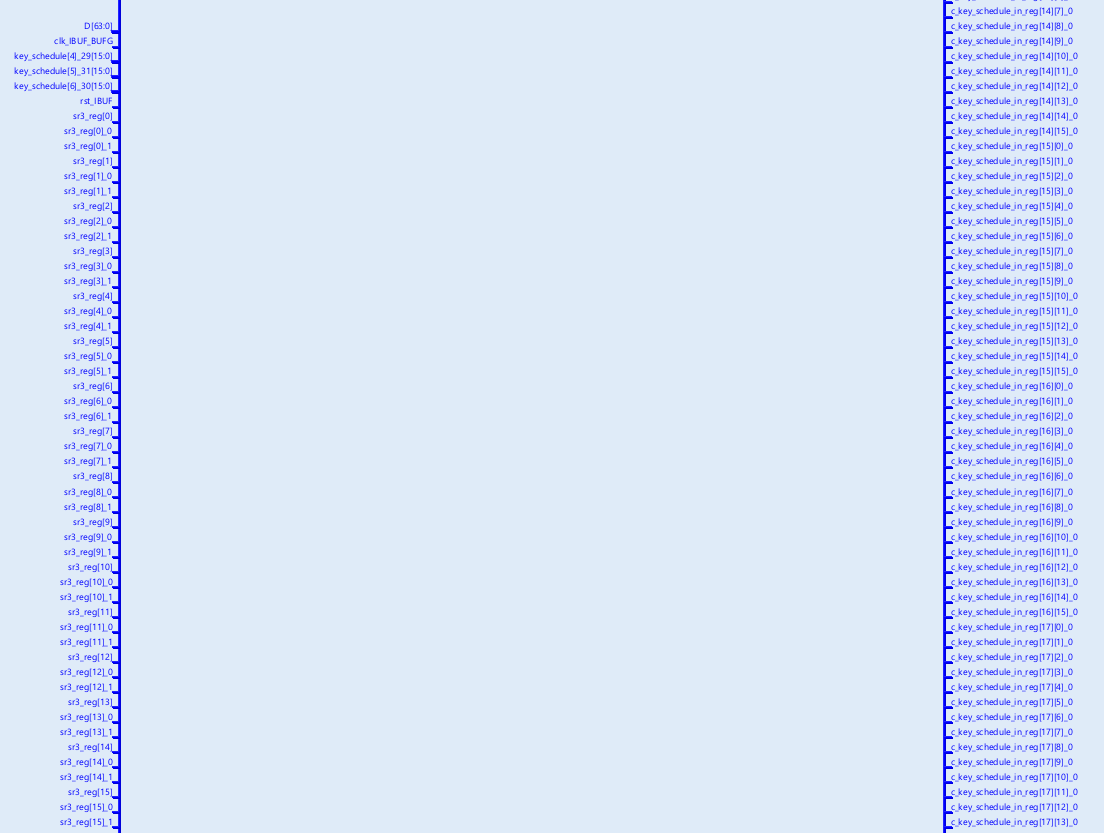


Figure ‎4.11 – Section of the technology schematic of key schedule generator

Due to the size of the RTL schematic, Figure ‎4.11 shows a section of the schematic that includes all of the inputs and some of the outputs. The full schematic can be accessed using Vivado.

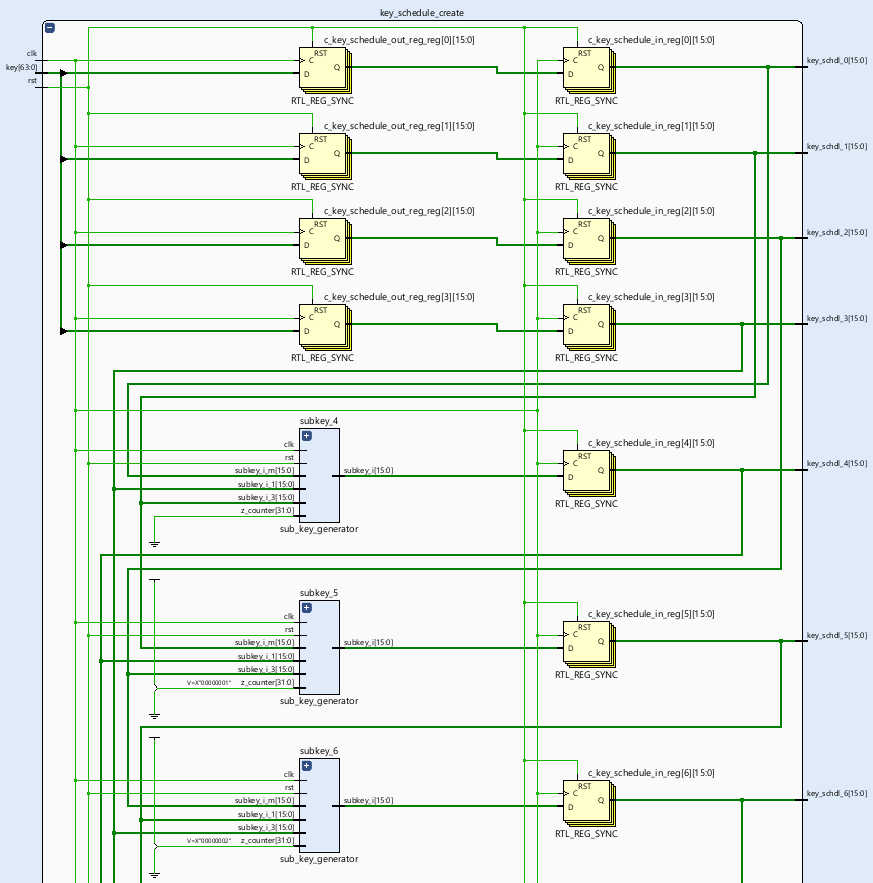


Figure ‎4.12 – Section of the RTL schematic of key schedule generator

###### Encrypt

**I/Os:**

* Plaintext. Block of data that is to be encrypted.
* Key. Key to be used to encrypt the plaintext.
* Load Plaintext. Flag that tells the cipher to input new plaintext data in the event that there is new data waiting to be encrypted.
* Load Key. Flag that tells the cipher to input new plaintext data in the event that there is new data waiting to be encrypted.
* Start Cipher. Flag to start the cipher with the current data.
* Ciphertext Ready. Flag when the ciphertext is finished being calculated and is ready to be displayed.
* Ciphertext. Block of encrypted data that is output to the screen.

As seen in Figure ‎4.13, the plaintext and ciphertext are equal in length to 2n – in our case 32 bits. The length of the key is mn – in our case 64 bits. All the flags are each 1 bit.

A screen shot of a computer

Description automatically generated

Figure ‎4.13 - I/Os for encryption

**Algorithm:**

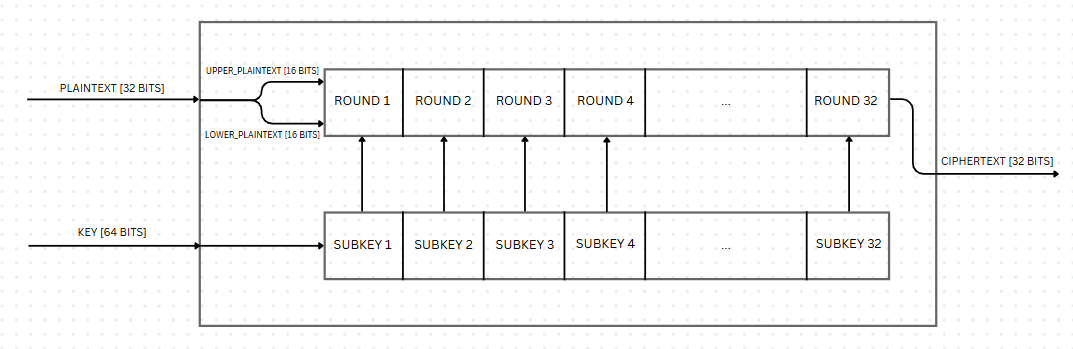


Figure ‎4.14 – Algorithm for the encryption module

As seen in Figure ‎4.14, after calculating each subkey, the plaintext is divided into 2 halves and each half is entered into the network of rounds. In our case the plaintext goes through 32 rounds (whose algorithm is described above) and the 2 outputs of the last round are combined together to form the ciphertext.

When all data is received via the get\_data module (see ‎4.1.2.2.1.4 Get Data), the load plaintext and load key and start cipher flags are raised. This signals the board to start encrypting the data. The cipher then begins generating a key schedule and calculating round outputs in parallel. This saves us time as once the first subkey is generated I can use it for the first round calculation.

In the framework of the project, I implemented both the encryption module and the decryption module. Overall it is almost exactly the same as the encryption module aside from 2 key points. First, after the last round of encryption in a Feistel network, we swap the right- and left-hand sides of the ciphertext and then combine them. This means that when decrypting, the starting right- and left-hand inputs are not the right- and left-hand parts of the ciphertext but rather we need to swap them. The second difference is that we need to use the reverse key schedule as encrypting. This means that the subkey for the first round will be the last subkey in the key schedule. This means that decrypting takes more time since we need to wait until all of the keys have been generated before starting to decrypt the ciphertext.

As there is a discrepancy between the amount of time it takes to encrypt and decrypt, I have chosen nonetheless to set the amount of time each module takes to 152 clock cycles as this is the longer time and I wanted the encryption and decryption times to match as now there is one set time for both.

As everything else is identical, I am choosing only explaining the encryption module.

The technology schematic is shown in Figure ‎4.15



Figure ‎4.15 – Technology schematic for encryption module

Figure ‎4.16 shows the RTL schematic for the encryption module.

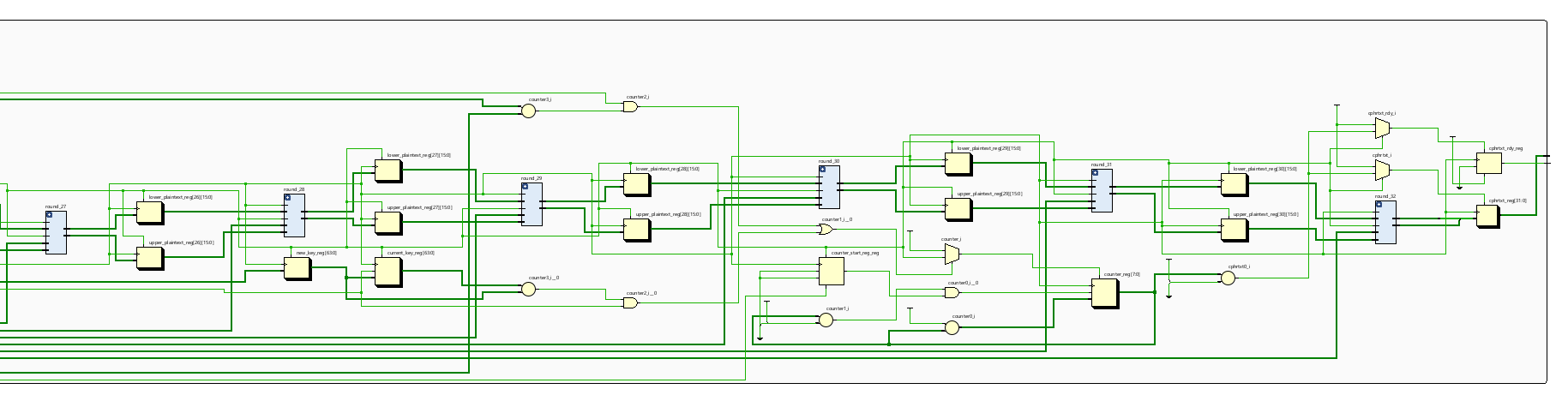
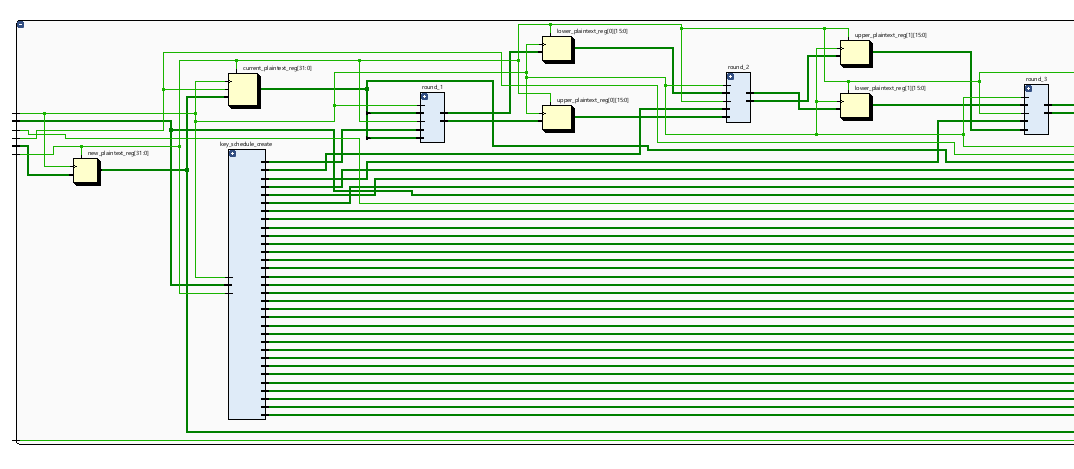
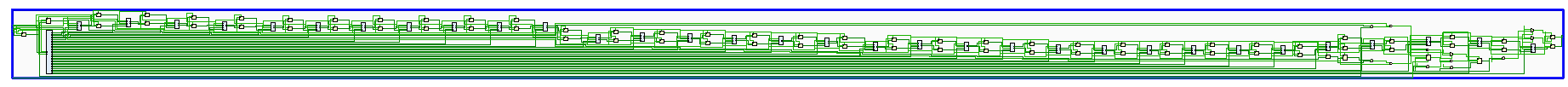


Figure ‎4.16 – RTL schematic of encryption module. Bottom figures are close ups of the beginning and end of the schematic

#### UART

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 input 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 from serial ports on the computer to the Basys3 and its FPGA.

The UART modules that I implemented 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.

##### Modules

###### TxD

**I/Os:**

* Input data. Data to be sent.
* Start txd. Flag when ready to start transmission.
* Txd data out. Individual bit sent to the computer.
* End of Txd. Flag when all data is sent.

As seen in Figure ‎4.17, input data has a length of 8 bits. All the other inputs and outputs are 1 bit long each.

A screen shot of a computer code

Description automatically generated

Figure ‎4.17 - I/Os for TxD module

**A diagram of a diagram

Description automatically generatedAlgorithm:**

Figure ‎4.18 - Block diagram of TxD module

**A diagram of a diagram

Description automatically generated**

Figure ‎4.19 - Block diagram of the TxD state machine

There are two main processes in the TxD module – the baudrate generator and the TxD state machine (see Figure ‎4.18)

**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. The Basys3’s clock ticks 100 million times per second whereas the computer’s clock ticks at 9600 times per second. In order to synchronize the clocks, I created a new clock that ticks one time every 100 million / 9600 = 10416 cycles of the Basys3’s clock. This way I can use the Basys3’s clock to simulate the computer’s baudrate.

**State Machine:**

The state machine (see Figure ‎4.19) 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

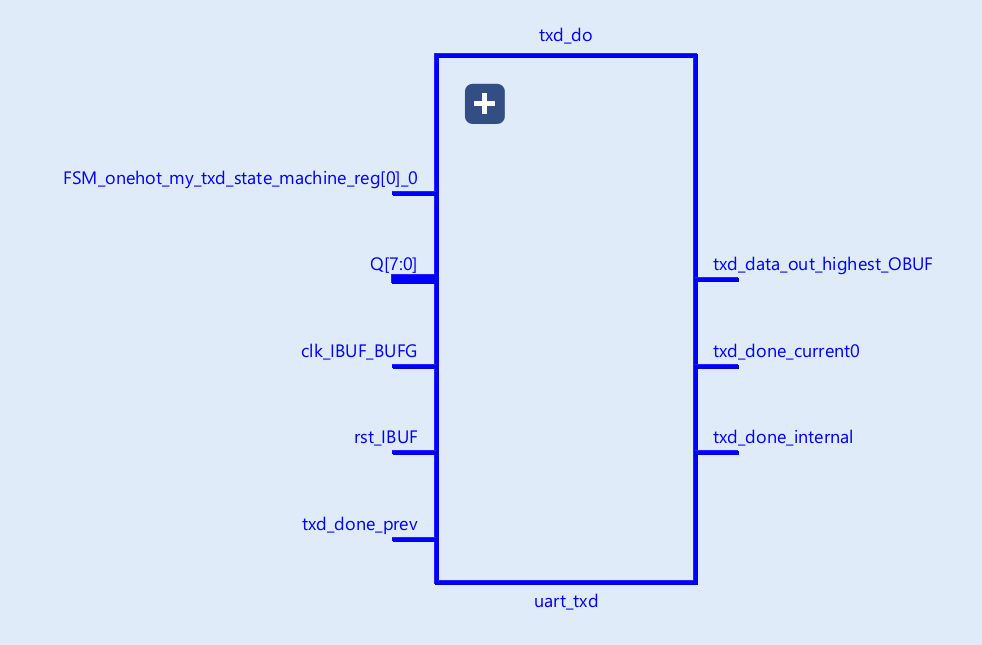
Figure ‎4.20 shows the technology schematic for the TxD module.

Figure ‎4.20 - Technology schematic of TxD module

Figure ‎4.21 shows the full RTL schematic of the TxD module along with a close up of the state machine.

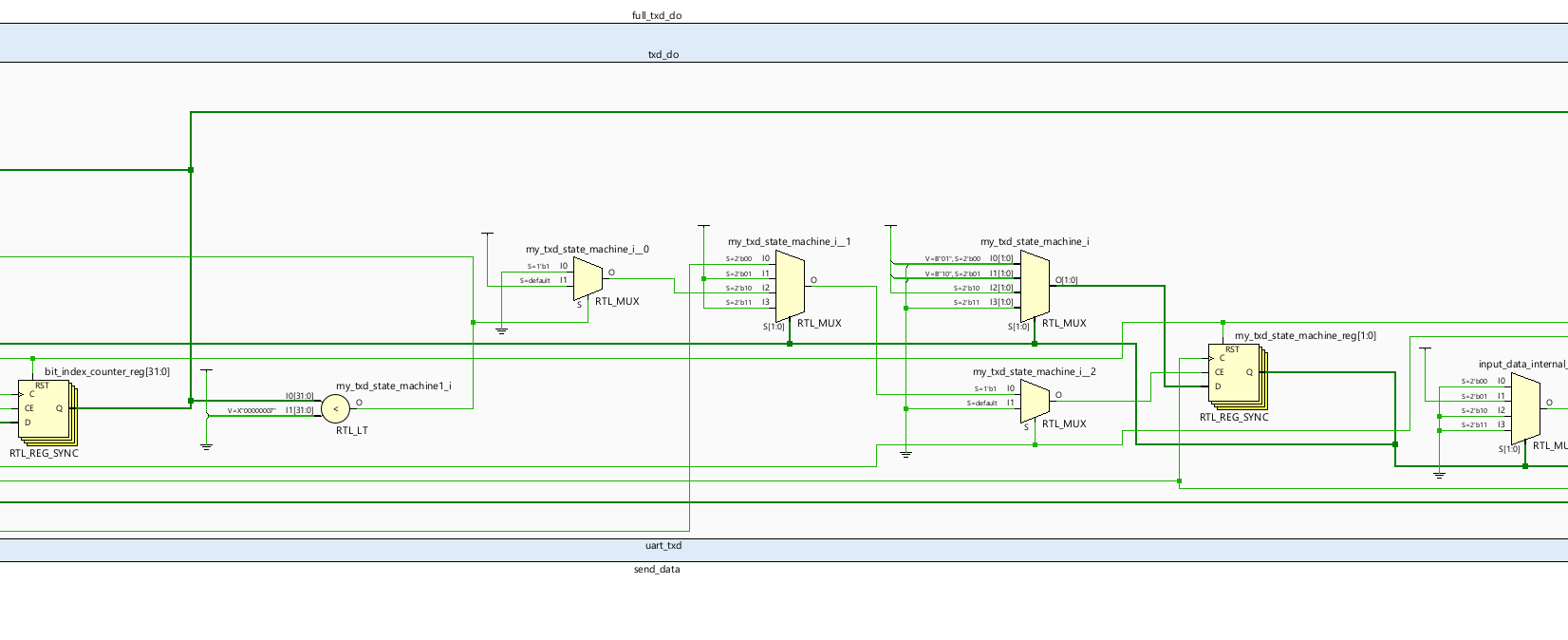
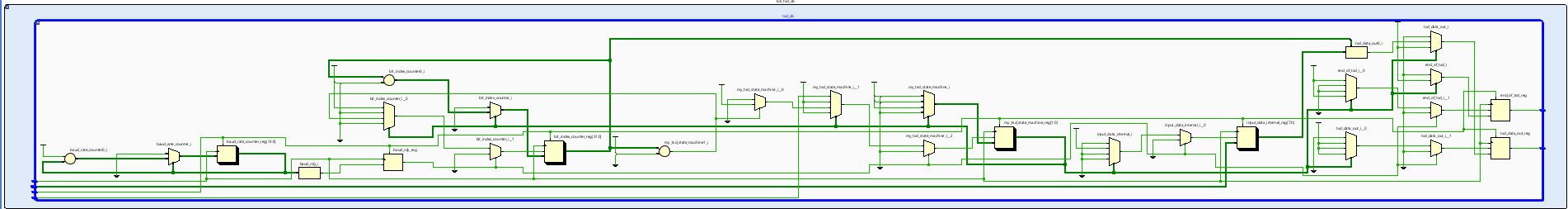


Figure ‎4.21 - RTL schematic of TxD module. The bottom figure shows the state machine in greater detail.

###### RxD

**I/Os:**

* Rxd data in. Individual bit of data being received.
* Rxd done. Flag to signal when all data is received.
* Data to cipher. Concatenated data that has been received.

Input data has a length of 8 bits. All the other inputs and outputs are 1 bit long each (see Figure ‎4.22)

A screen shot of a computer code

Description automatically generated

Figure ‎4.22 - I/Os for RxD module

**A diagram of a diagram

Description automatically generatedAlgorithm:**

Figure ‎4.23 - Block diagram of the RxD module

A diagram of a diagram

Description automatically generated

Figure ‎4.24 - Block diagram of the RxD state machine

The receive module works by testing each bit at the middle of its cycle. When a new bit is sent to the board, in order to receive the bit in the most accurate manner we need to wait until the bit is stable. In the beginning of the waveform of the bit there is often a short period of time where the bit fluctuates or glitches out. In order to receive the correct data I use a counter which counts until we get the middle of the bit where it is already stable and I input the bit based on this information.

As in the TxD module, the RxD module (see Figure ‎4.23) contains both a baudrate generator and a state machine.

**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 (see Figure ‎4.24) 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.

Figure ‎4.25 shows the technology schematic.

A screen shot of a computer code

Description automatically generated

Figure ‎4.25 – Technology schematic of the RxD module

Figure ‎4.26 shows the RTL schematic for the RxD module:

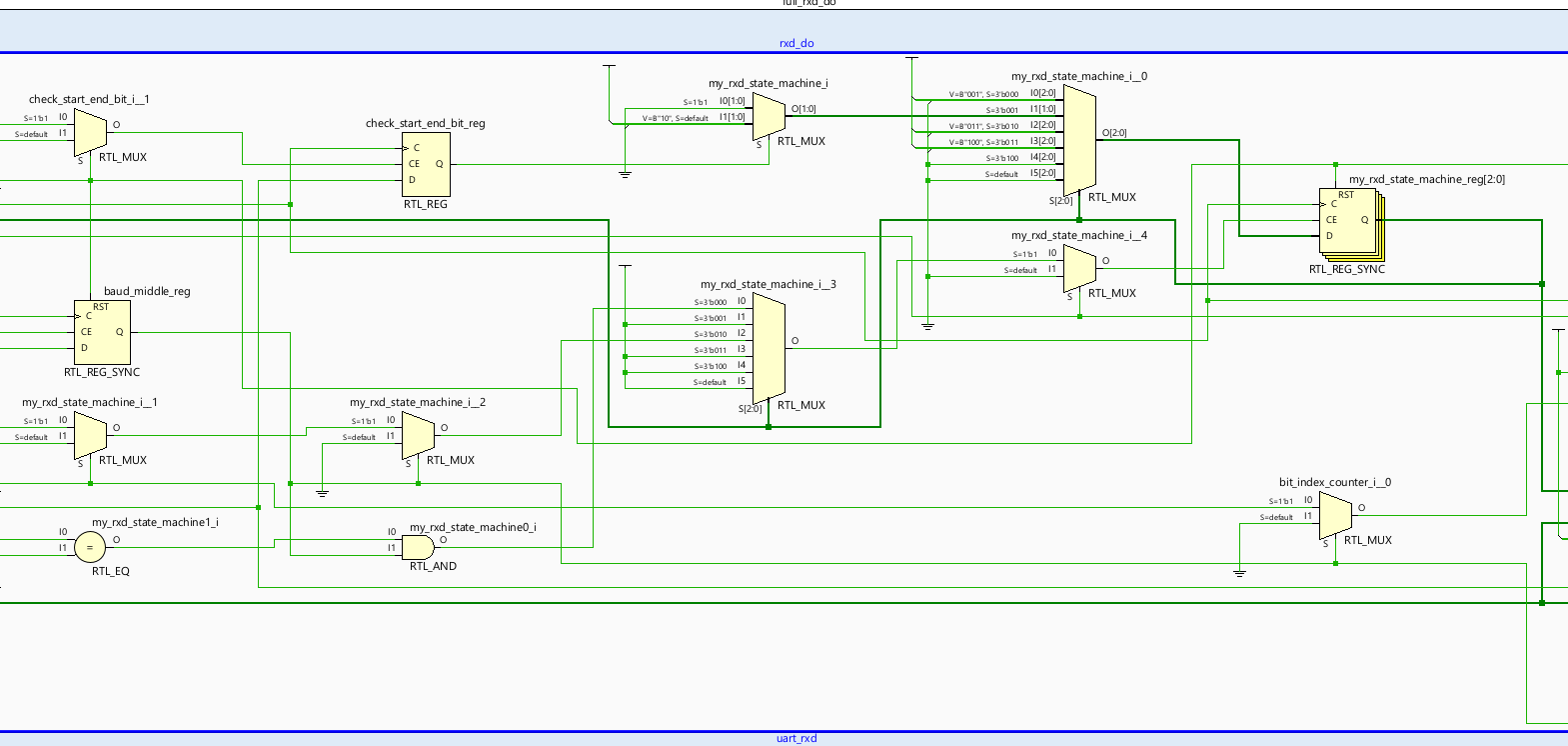
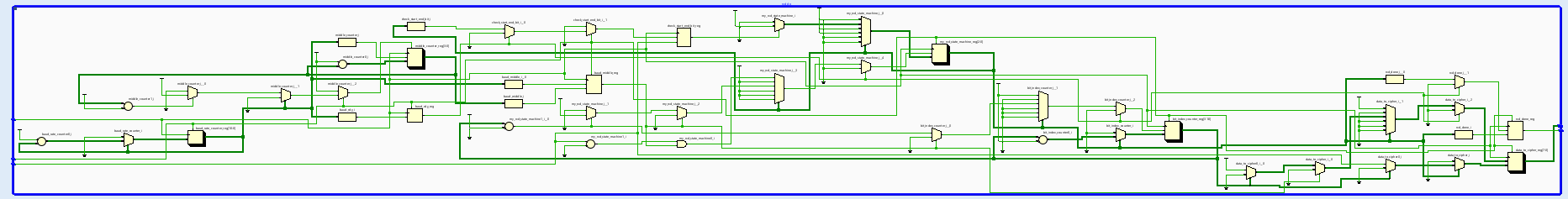


Figure ‎4.26 RTL schematic of RxD module. The bottom figure shows the state machine in greater detail.

###### Send Data

**I/Os:**

* Ciphertext send. This is a flag that tells us if the ciphertext is ready to be sent.
* Data. Total data to be output
* Txd\_data\_out\_top. Individual bit being output each baud cycle.

A screen shot of a computer code

Description automatically generatedAs seen in Figure ‎4.27, the length of the data is 32 bits as this is the ciphertext being output. The other two I/Os are one bit each.

Figure ‎4.27 - I/Os for send data module

**Algorithm:**

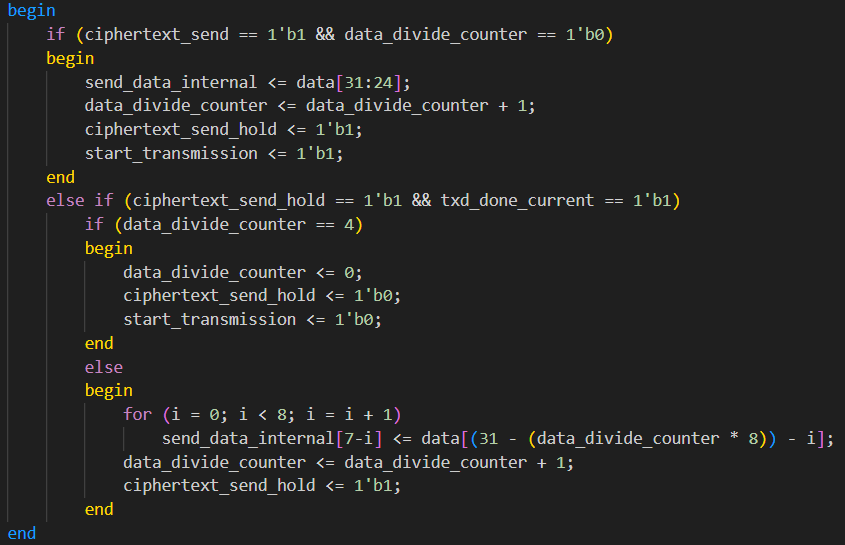
****

Figure ‎4.28 - Algorithm in code of the Send Data module

When the ciphertext is ready to be sent, the algorithm will send 8 bits of the total 32-bit ciphertext each time using the TxD module which operates on 8 bits of data as described earlier. The algorithm will send out 4 packages of 8 bits each time a ciphertext is ready to be sent. Once all 32 bits are sent, the algorithm waits for the flag that signals that a new ciphertext is ready to be sent (see Figure ‎4.28).

Figure ‎4.29A diagram of a computer code

Description automatically generated shows the technology schematic for the send data module.

Figure ‎4.29 – Technology schematic for send data module

Figure ‎4.34(below) shows the RTL schematic for the send data module.

###### Get Data

**I/Os:**

* Rxd\_data\_in\_top. Individual bit being received by the board.
* Start cipher. Flag that signals to the cipher that all data has been received and cipher can be started.
* Data. All bits of data.

As seen in Figure ‎4.30, the length of the data is 104 bits as this is the ciphertext being output. The other two I/Os are one bit each.

**A screen shot of a computer code

Description automatically generated**

Figure ‎4.30 - I/Os for get data module

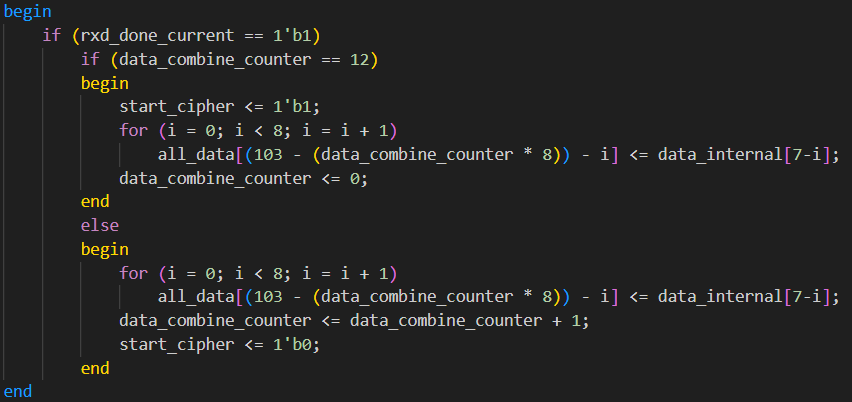
**Algorithm:**

Figure ‎4.31 - Algorithm in code of the Get Data module

When each batch of 8 bits of data is received using the RxD module described earlier, the algorithm concatenates the current set of data into one big string of bits. When the necessary amount of bits have been received, the algorithm signals to the cipher that it should operate on the current set of data (see Figure ‎4.31).

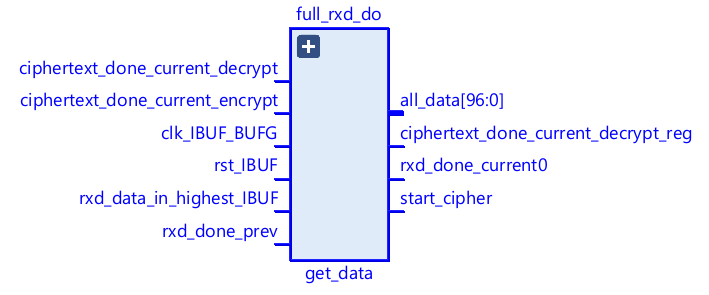
Figure ‎4.32 shows the technology schematic for the get data module.

Figure ‎4.32 – Technology schematic of the get data module

Figure ‎4.33 shows the RTL schematic for the get data module.

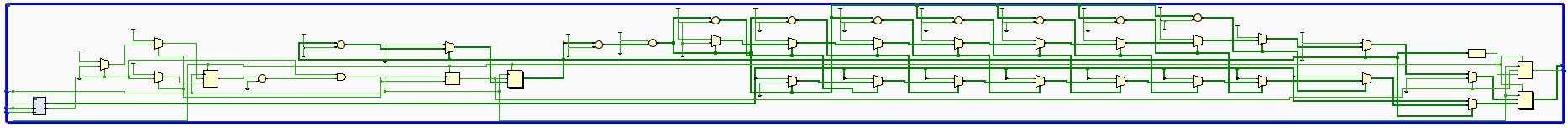


Figure ‎4.33 - RTL schematic of get data module

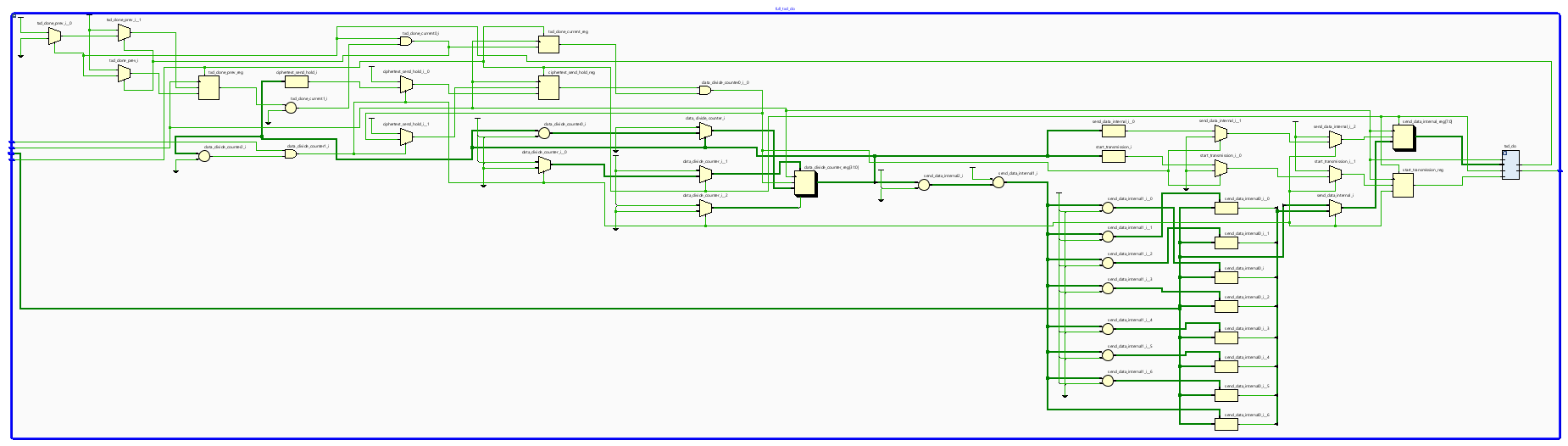


Figure ‎4.34 - RTL schematic of Send Data module

#### Seven Segment Display

**I/Os:**

* Trigger. Flag that triggers the display when the cipher is done.
* Seven segment data. Controls the data being displayed.

As seen in Figure ‎4.35, the trigger is one bit and the seven segment data is 11 bits long.

A screen shot of a computer code

Description automatically generated

Figure ‎4.35 - I/Os for seven segment data

The seven-segment display is capable of displaying up to 4 seven segment characters including numbers 0-9 and representations of letters including d, o, n, E. The purpose of the seven segment display in my project is to signal that the cipher is done encrypting by displaying ‘d0nE’ when the cipher finishes running.

A diagram of a number of digits

Description automatically generatedThe seven segment display works by activating up one of the 4 seven segment characters and activating any of the 7 LEDs in that character. Since it is only possible to activate one of the 4 displays at once, we activate them one at a time in a cycle that is faster than the eye can process meaning that the display is flickering very fast but to the naked eye it seems to be one continuous light source. In Figure ‎4.36, seen in source [6], we see that the total refresh period meaning the cycle of turning on and off all four displays must be between 1 and 16 ms. I am using a refresh period of 2.62 ms. The display works by an active low.

Figure ‎4.36 - Refresh period of Seven Segment Display

#### Constraints File

In order to place the highest level of I/Os in the design onto the board in specific places, we use a constraints file. The constraints file has a .xdc (Xilinx design constraints) extension.

Using the file, we can attach any of the input signals to any of the inputs on the Basys3 board such as buttons and the output signals to outputs on the Basys3 board such as LEDs. Both input and output signals can be attached to Pmod ports.

See Table ‎4.1 for overview of the ports for the outer signals.

|  |  |
| --- | --- |
| Signal | Basys3 I/O |
| clk | 100MHz Crystal Ocillator |
| rst | Middle push button |
| Seven segment top | Seven segment display |
| TxD and RxD | USB-RS232 interface |

Table ‎4.1 – Design signals and their physical location on the Basys3 board

### Python Controller

#### Modules

##### PySerial

The PySerial python library allows users to connect a serial port on the Raspberry Pi or PC to a serial port on a hardware board. Using the connection, the user can send and receive bytes of data to and from the board. In our case we send 13 bytes of data, representing the mode (encryption or decryption), the 8-byte key and the 4-byte plaintext. The Raspberry Pi receives the 4-byte output ciphertext from the board.

Figure ‎4.37 - Code to open serial port

The code in Figure ‎4.37 opens a serial port through’/dev/ttyUSB1’ (one of the USB ports on the Raspberry Pi) with a baudrate of 9600 bps. There are other parameters that can be altered such as use of parity bits, byte sizes and more.

In order to read incoming data and write outgoing data, the ser.read(num\_of\_bytes) and the ser.write(‘byte’) commands (respectively) are used.

##### TKinter

The TKinter python library allows users to create user interfaces for inputting and displaying data.

In order to input plaintexts and keys and display the output, I used the library to create a window where the user can choose to either encrypt or decrypt and this opens up a new window where the user can input the data he wants to encrypt and the ciphertext is displayed in the window.

The code in Figure ‎4.38 opens a TKinter window.

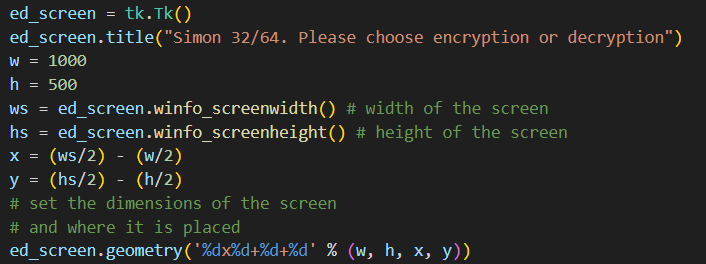


Figure ‎4.38 - Code to open TKinter window

# Basys3

## Specs

Figure ‎5.1 - Overview of the Basys3 board

### I/Os

Figure ‎5.1, taken from source [7], shows an overview of the Basys3 board. The Basys3 [7] board includes sixteen slide switches, five push buttons, sixteen individual LEDs, and a four-digit seven segment display. The pushbuttons and slide switches are connected to the FPGA via series resistors to prevent damage from inadvertent short circuits (a short circuit could occur if an FPGA pin assigned to a pushbutton or slide switch was inadvertently defined as an output). The five pushbuttons, arranged in a plus-sign configuration, are "momentary" switches that normally generate a low output when they are at rest, and a high output only when they are pressed. Slide switches generate constant high or low inputs depending on their position.

This gives us a total of 21 inputs, 16 LED outputs and 28 Seven Seg outputs (7 per digit).

Figure ‎5.2, taken from source [7], shows the physical locations of each input and output on the Basys3 board.

A diagram of a circuit

Description automatically generated

Figure ‎5.2 - I/Os on the Basys3 board

The Pmod ports are arranged in a 2x6 right-angle and are 100-mil female connectors that mate with standard 2x6 pin headers. Each 12-pin Pmod port provides two 3.3V VCC signals (pins 6 and 12), two Ground signals (pins 5 and 11), and eight logic signals, as shown in Fig. 20. The VCC and Ground pins can deliver up to 1A of current. Pmod data signals are not matched pairs, and they are routed using best-available tracks without impedance control or delay matching. are shown in Figure ‎5.3, taken from source [7], shows the pin assignments for the Pmod I/Os connected to the Basys3 board.

This gives us a total of 32 I/Os

A diagram of a computer chip

Description automatically generated

Figure ‎5.3 - Pmod ports

In total the Basys3 contains up to 53 inputs and up to 76 outputs and a total of 97 I/Os.

### FPGA

The Basys3 includes an on-board Xilinx-AMD Artix-7 FPGA (XC7A35T-1CPG236C)

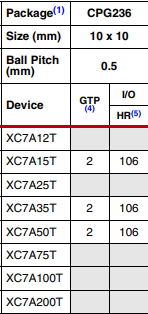
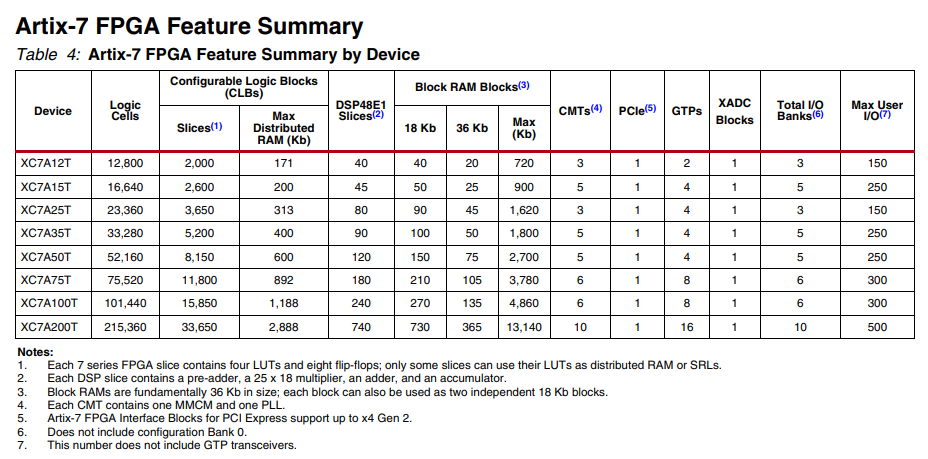


Figure ‎5.4 - Artix-7 FPGA family data

In Figure ‎5.4, taken from source [8], we see the specs for the XC7A35T family. Although the max user I/O in the XC7A35T family is 250 I/Os, we see that our particular device, the XC7A35T-1CPG236C has only 106 I/Os.

In this project, the primary concerns are the number of Flip-Flops and LUTs. We can see in note one that each slice contains eight Flip-Flops and four LUTs for a total of 5200x8=41,600 Flip-Flops and 5200x4=20,800 LUTs

## Tech Implementation

There are 3 main ways to program the Basys3 board. SPI Flash, JTAG and USB. In order to choose which mode to use we can set the pins of the JP1 pin set to set the mode.

Figure ‎5.5, taken from source [7], shows the physical connections for each programming mode.

A diagram of a computer

Description automatically generated

Figure ‎5.5 - Basys3 configuration options

Within this project I programmed the board using the USB and JTAG methods.

### JTAG

The [7] Xilinx Tools typically communicate with FPGAs using the Test Access Port and Boundary-Scan Architecture, commonly referred to as JTAG. During JTAG programming, a .bit file is transferred from the PC to the FPGA using the onboard Digilent USB-JTAG circuitry (port J4) or an external JTAG programmer, such as the Digilent JTAG-HS2 attached to port J5 (located below port JA). You can perform JTAG programming any time after the Basys 3 has been powered on regardless of what the mode jumper (JP1) is set to. If the FPGA is already configured, then the existing configuration is overwritten with the bitstream being transmitted over JTAG. Setting the mode jumper to the JTAG setting (seen in Fig. 3) is useful to prevent the FPGA from being configured from any other bitstream source until a JTAG programming occurs. Programming the Basys 3 with an uncompressed bitstream using the on-board USB\_JTAG circuitry usually takes around five seconds. JTAG programming can be done using the hardware server in Vivado. The demonstration project available at digilentinc.com provides an in-depth tutorial on how to program your board.

### USB

You [7] can program the FPGA from a pen drive attached to the USB-HID port (J2) by doing the following:

In order to program the board, we must load the .bit file through the USB-A port located next to the JP1 pin set. To find the .bit file we must go into the .runs folder > impl\_1 > .bit file. We then transfer the file to an external storage drive such as a flash drive. We put the file into the root folder of the drive and then plug the flash drive into the Basys3. We then turn on the board, set the JP1 to the correct setting and load the file onto the board using the “Prog” button (see image above).

The FPGA will automatically be configured with the .bit file on the selected storage device. Any .bit files that are not built for the proper Artix-7 device will be rejected by the FPGA.

The Auxiliary Function Status, or "BUSY" LED (LD16), gives visual feedback on the state of the configuration process when the FPGA is not yet programmed:

• When steadily lit, the auxiliary microcontroller is either booting up or currently reading the configuration medium (pen drive) and downloading a bitstream to the FPGA.

• A slow pulse means the microcontroller is waiting for a configuration medium to be plugged in.

• In case of an error during configuration, the LED will blink rapidly.

When the FPGA has been successfully configured, the behavior of the LED is application-specific. For example, if a USB keyboard is plugged in, a rapid blink will signal the receipt of an HID input report from the keyboard.

# Raspberry Pi

I am using a Raspberry Pi for my user interface and for inputting and outputting data to the Basys3 board.

The [9] Raspberry Pi is a minicomputer that has a Broadcom BCM2711 quad-core Cortex-A72 (ARM v8) 64-bit SoC chip running at 1.5 GHz. It also includes 4 USB-A ports, 2 micro-HDMI outputs, a 3.5 mm headphone jack, a USB-C port used for power supply and 40 GPIOs.

In my project I used only the USB ports to connect the Raspberry Pi to the Basys3 board for input and output purposes.

# Vivado

In order to implement the code on the board, I have chosen to use the Xilinx-AMD Vivado Design Suite version 2023.2.

To do this, we need two sets of code. Source code and a constraints file. The source code includes all modules mentioned above. The constraints file is a file with a which defines the physical outputs of certain signals onto the Basys3 board.

Once we add the requisite files to Vivado we need to run 3 operations. Design Synthesis, Design Implementation and Bitstream Generation.

**Synthesis**

Synthesis [10] is the process of transforming a Register Transfer Level (RTL) specified design into a gate-level representation. AMD Vivado™ synthesis is timing-driven and optimized for memory usage and performance.

The main purpose of synthesis is to generate a netlist of components including wiring and logic gates from the original code.

**Implementation**

Vivado [11] implementation includes all steps necessary to place and route the netlist onto device resources, within the logical, physical, and timing constraints of the design.

The implementation takes the synthesized netlist of components and places each component in a physical DFFs, LUTs and others that exist on the board.

**Bitstream Generation**

In order to program the board Vivado creates a file with a .bit extension that is able to program the Basys3 board according to the implemented design and the constraints.

# Area and timing (Vivado)

## Area Reports

### VHDL

#### Post Synthesis

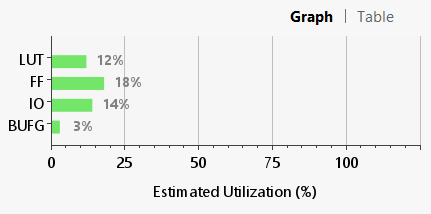
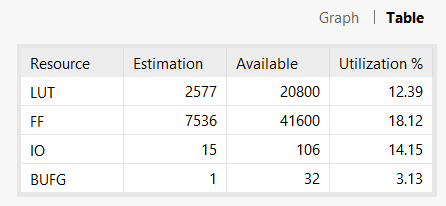


Figure ‎8.1 - Post synthesis VHDL area utilization

#### Post Implementation

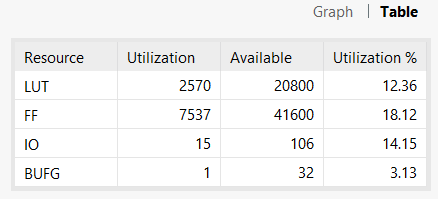
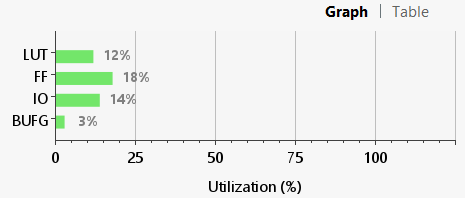


Figure ‎8.2 - Post implementation VHDL area utilization

Figure ‎8.1, taken from Vivado, show the post synthesis area report in graph and table form.

Figure ‎8.2, taken from Vivado, show the post implementation area report in graph and table form.

#### Device

Figure ‎8.3, taken from Vivado, shows the technology placement on the FPGA on the Basys3.

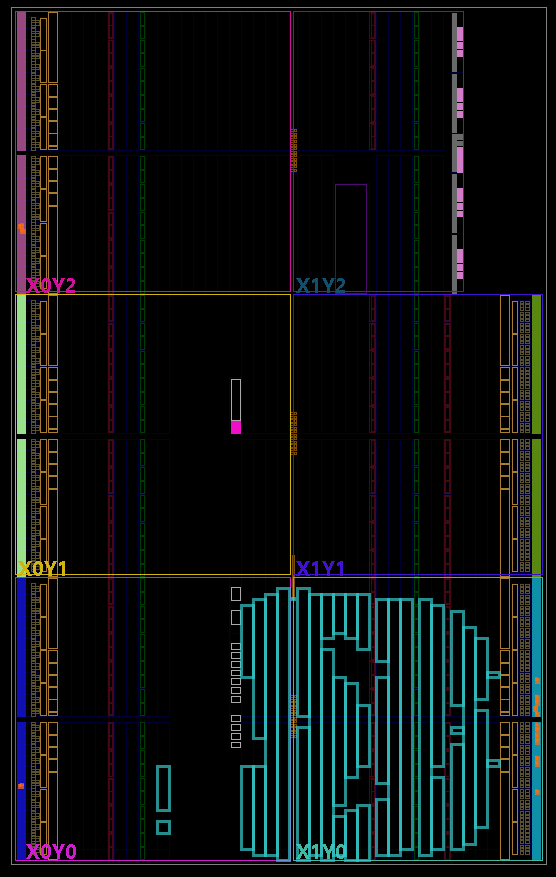


Figure ‎8.3 – VHDL device technology placement

### Verilog

#### Post Synthesis

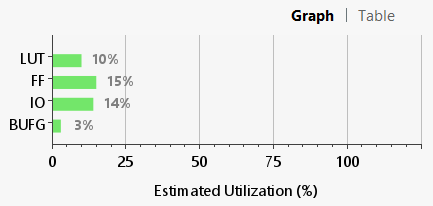
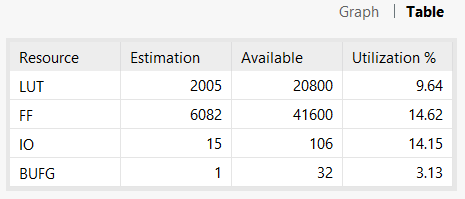


Figure ‎8.4 - Post synthesis Verilog area utilization

#### Post implementation

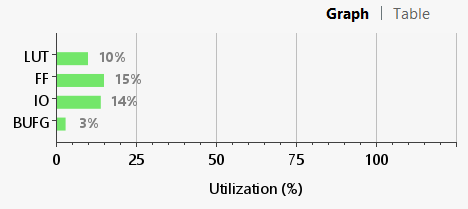


Figure ‎8.5 - Post implementation Verilog area utilization

Figure ‎8.4, taken from Vivado, show the post synthesis area report in graph and table form.

Figure ‎8.5, taken from Vivado, show the post implementation area report in graph and table form.

#### Device

Figure ‎8.6, taken from Vivado, shows the technology placement on the FPGA on the Basys3 board.

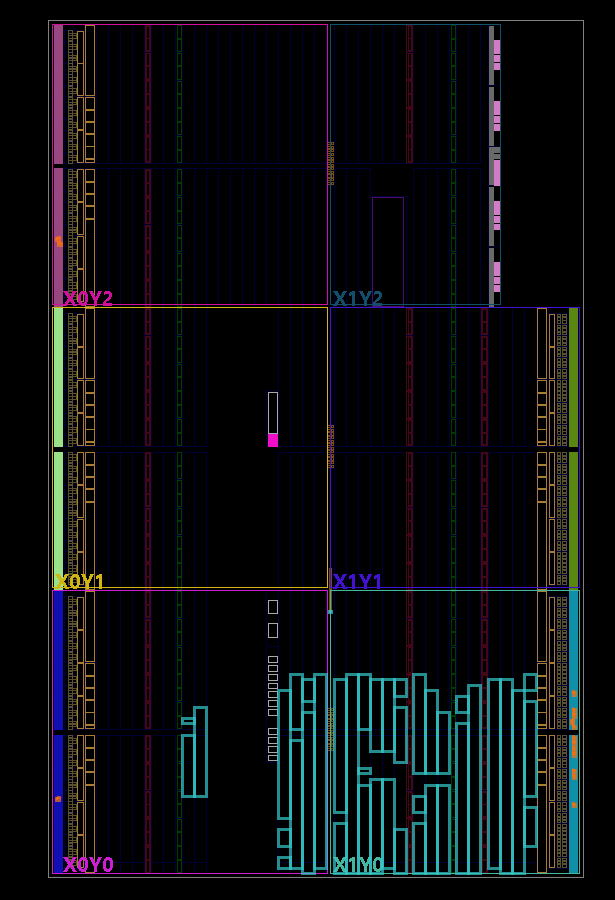


Figure ‎8.6 – Verilog device technology placement

#### I/O plan

Figure ‎8.7, taken from Vivado, shows the I/O placement on the Basys3 board.

A screenshot of a video game

Description automatically generated

Figure ‎8.7 - I/O placement on Basys3 board

## Timing Reports

### VHDL

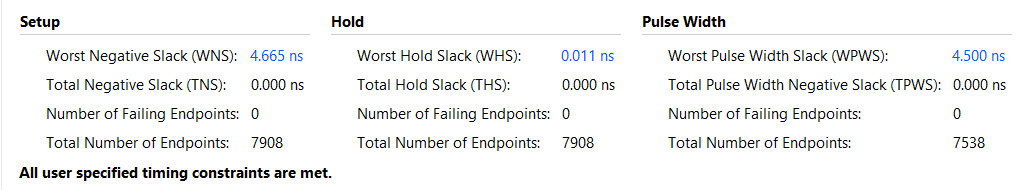


Figure ‎8.8 - Timing report table for VHDL implementation

### Verilog

Figure ‎8.9 - Timing report table for Verilog implementation

Worst negative slack represents the longest time it takes for a signal to be set. Since the clock cycle is 5 ns and we are working with synchronous signals, the signal must be set before the clock changes its state meaning that the max time can be 5ns. In both VHDL and Verilog this condition is upheld.

Worst negative slack represents the time it takes for a signal to set. As long as the signal is set before the clock changes it is good.

Worst pulse width slack represents the minimum amount of time given by the pulse of the clock over the required time of the design. When this time is positive it means that the clock pulse is always wide enough for the design.

Figure ‎8.8 and Figure ‎8.9, both taken from Vivado, show the timing report for the VHDL and Verilog (respectively) implementations.

# Discussions

## Simulation Results

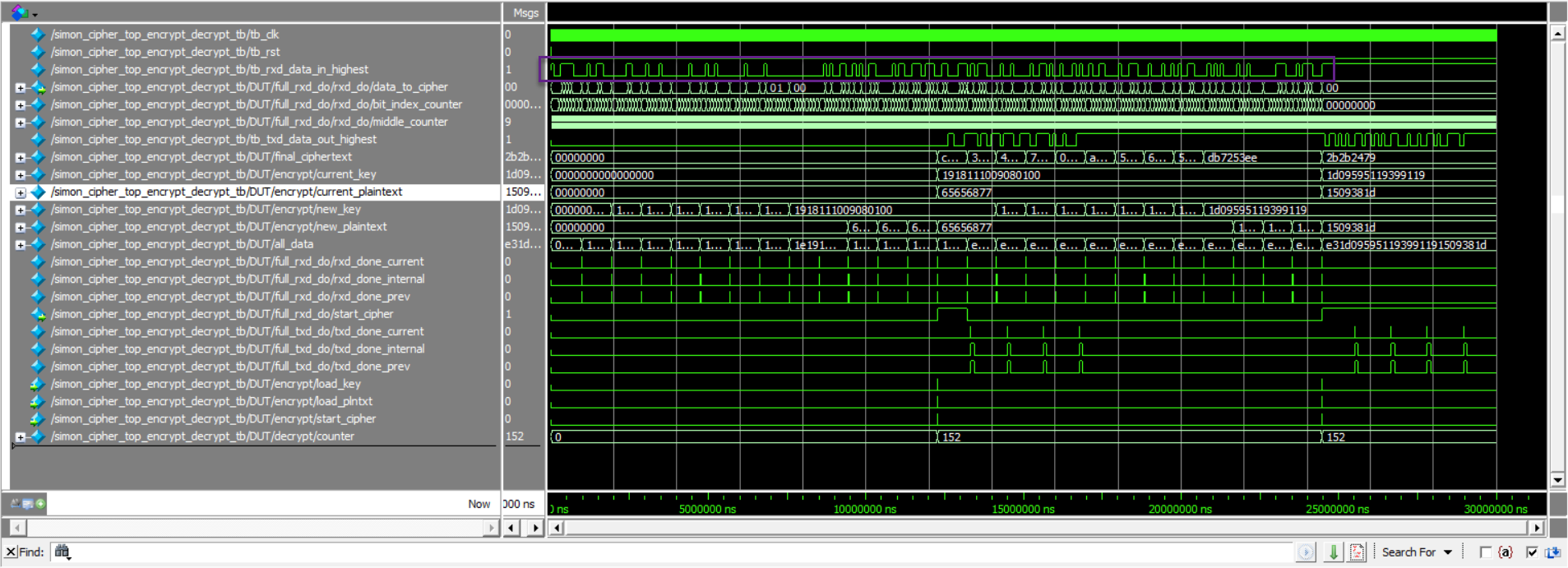
Figure ‎9.1 shows the simulated wave input in Modelsim:

Figure ‎9.1 - Simulation of input wave

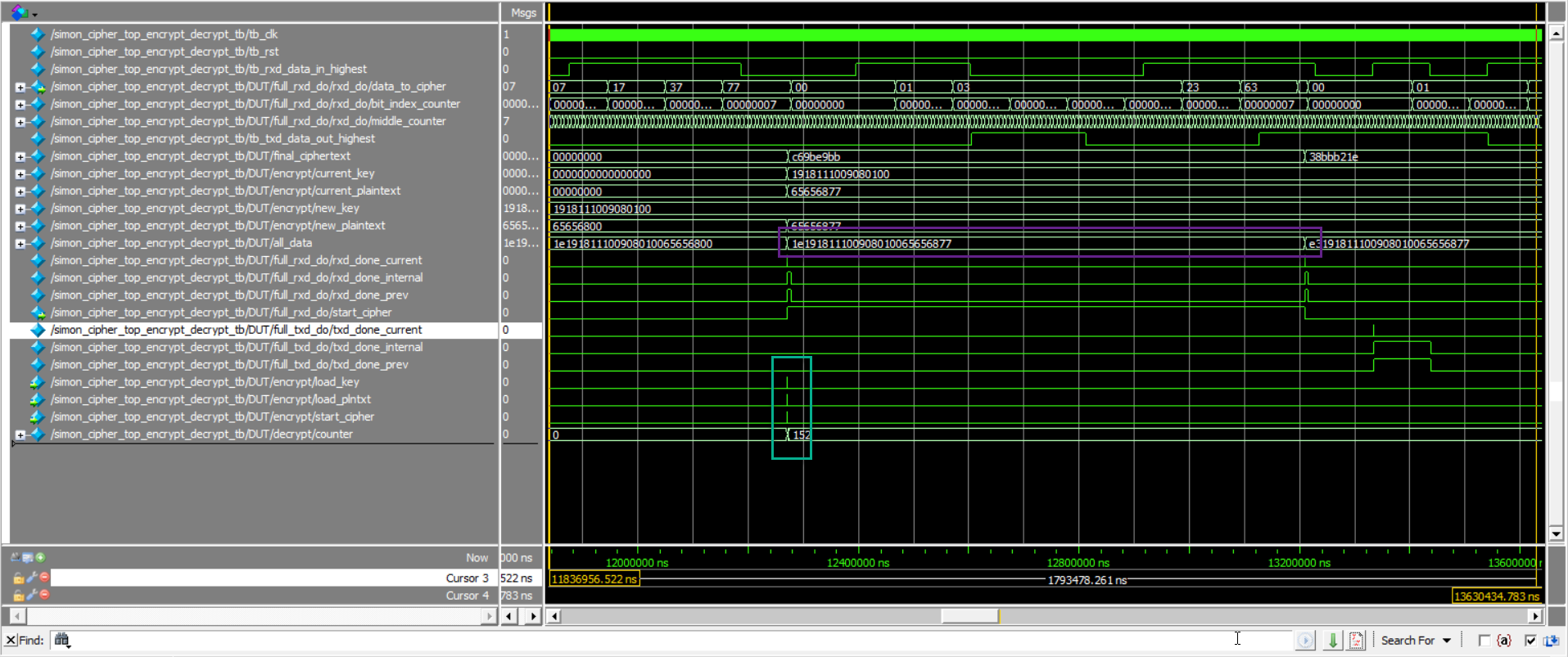
Figure ‎9.2 shows the simulated hexadecimal input (including encryption byte) and cipher start in Modelsim:

Figure ‎9.2 - Simulation of cipher start after all data has been received

Figure ‎9.3A screenshot of a computer

Description automatically generated shows the simulated encrypted ciphertext and wave output in Modelsim:

Figure ‎9.3 - Simulation of ciphertext output

Figure ‎9.4A screenshot of a computer program

Description automatically generated shows a simulation of the encrypted plaintext and wave output based on the plaintext and key in Modelsim:

Figure ‎9.4 – Simulation of ciphertext of key and plaintext

Figure ‎9.5A screenshot of a computer program

Description automatically generated shows a simulation of the decrypted ciphertext and wave output based on the ciphertext and key in Modelsim:

Figure ‎9.5 - Simulation of decrypting the ciphertext from above

In the simulation I was able to successfully input and output data and encrypt data and then decrypt it with the expected results.

## Actual Results

In order to use the cipher, we must load the technology onto the Basys3 board using Vivado and then connect the board to the Raspberry Pi.

Step 1. Load Verilog and Constraint files into Vivado

Step 2. Run synthesis and implementation

Step 3. Copy the .bit file to the root folder of an external hard drive

Step 4. Using the USB port, load the bitstream to the Basys3

Step 5. Connect one of the USB ports of the Raspberry Pi to the micro usb port on the Basys3

Step 6. Reset the board with the reset button

Step 7. Run the python code on the Raspberry Pi

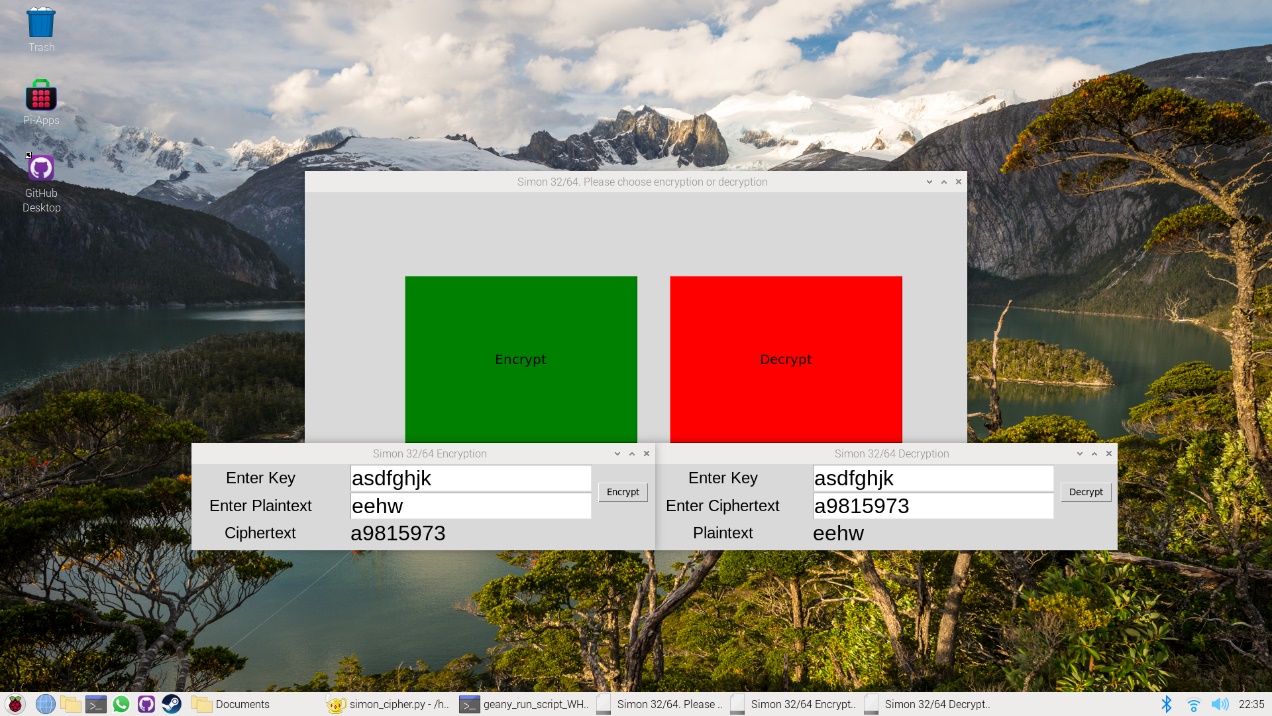


Figure ‎9.6 - Encryption and decryption using Raspberry Pi and Basys3

The plaintext and key used in Figure ‎9.6 are ascii representations of the plaintext and key used in Figure ‎9.3.

A screenshot of a computer

Description automatically generated

Figure ‎9.7 - Encryption and decryption of multiple bytes using Raspberry Pi and Basys3

As seen in Figure ‎9.6 there is an option to Encrypt or Decrypt the data. When using the same key to encrypt and decrypt the data we see that the original data is encrypted and decrypted successfully.

In Figure ‎9.7 I successfully encrypt and decrypt multiple bytes of data. When the original plaintext has less than a whole number of bytes, the subsequent decrypted plaintext will be left padded with zeros as the cipher accepts and returns a whole number of bytes. In our case 4 bytes or 32 bits.

## Comparison of Verilog and VHDL

### Area

Table ‎9.1 – Comparison of FFs and LUTs for the VHDL and Verilog implementations

|  |  |  |
| --- | --- | --- |
|  | Flip-Flops | LUTs |
| VHDL | 7537 | 2570 |
| Verilog | 6082 | 1996 |

#### Differences

In terms of FFs, Verilog uses about 1500 less flip flops (see Table ‎9.1). Comparing the area reports in Figure ‎B.1 and Figure ‎B.2 we see that VHDL uses unique FFs for each iteration of the key schedule (both encrypting and decrypting) while Verilog uses one set of FFs as we use the same key schedule for both encrypting and decrypting.

In terms of LUTs, Verilog uses about 600 less LUTs (see Table ‎9.1). Comparing the area reports (see Figure ‎B.1 and Figure ‎B.2) we see that VHDL uses unique LUTs for each iteration of the key schedule (both encrypting and decrypting) while Verilog uses one set of LUTs as we use the same key schedule for both encrypting and decrypting.

### Timing and Throughput

In order to calculate the maximum frequency of for the cipher to run properly we use the formula in Equation ‎9.1:

Equation ‎9.1

To calculate the maximum throughput, we use the formula in Equation ‎9.2:

Equation ‎9.2

As described above, the cipher takes 152 clock cycles to run and the cipher uses 96 bits, 32 for the plaintext and 64 for the key.

Plugging this into the formula we get Equation ‎9.3:

Equation ‎9.3

#### VHDL

In Figure ‎8.8 we see that the worst negative slack is 4.665 ns.

When we plug the numbers into Equation ‎9.3, we get Equation ‎9.4.

Equation ‎9.4

The maximum throughput will be Equation ‎9.5:

Equation ‎9.5

#### Verilog

In Figure ‎8.9 we see that the worst negative slack is 3.821 ns.

When we plug the numbers into Equation ‎9.3, we get Equation ‎9.6.

Equation ‎9.6

The maximum throughput will be Equation ‎9.7:

Equation ‎9.7

|  |  |  |
| --- | --- | --- |
|  | Max Frequency | Max Throughput |
| VHDL | 187.441 MHz | 118.38 Mbps |
| Verilog | 161.838 MHz | 102.21 Mbps |

Table ‎9.2 – Comparison of Max Frequency and Throughput

Looking at Table ‎9.2 we see that that the VHDL implementation has a wider range of frequencies in which to operate successfully and thus the maximum throughput is higher.

### Challenges

Every time the user enters a new plaintext and key combination, I expect the data to be entered correctly. In simulations, this works properly. In reality, this is not happening. My instinct is that the registers do not reset properly, and this is causing the data to be entered into the basys3 incorrectly. In order to combat this, when sending data to the Basys3 I send a string of 0s to effectively reset the registers properly before sending the actual data.

When creating the UART, at first, I used a one-bit counter that counts at double the speed of the baud rate in order to test the bit at the middle of its cycle but often the data was very inaccurate when coming to the board. In order to increase accuracy, I tried making the counter run at higher speeds relative to the baud rate and I eventually settled on a four-bit counter which runs at sixteen times the baud rate. The accuracy of the UART is around eighty-five percent.

# Conclusions

In conclusion, I think that I was able to successfully meet most of my goals of the project which include implementing the cipher algorithm on both VHDL and Verilog and gaining useful insight into the advantages and disadvantages of each language.

Despite challenges that arose, I was able to successfully implement a UART for transmitting data between the Raspberry Pi and the Basys3. These challenges offered a unique perspective on the advantages and disadvantages of using different techniques for data transfer, with the main considerations being accuracy versus space used and time taken to build the necessary code.

Using the different technologies and tools helped improve my proficiency and understanding of these tools. This includes the use of Vivado, Raspberry Pi, Modelsim and Basys3. At times there was a lot of trial and error in figuring out how everything works but I was able to overcome and successfully complete the objective.

Over the course of the project, I implemented the Simon32/64 variant of the cipher using the loop unrolling technique for implementing the Feistel Network. My implementation hopefully can be used for future inspiration and improvements to this specific algorithm and other cryptographic algorithms.

Possible future improvements and endeavors would be to implement the algorithm using an iterative implementation for the Feistel Network. Additional goals would be to implement all versions of the cipher either at once or separately and programming the board anew for each variant.

Other improvements would include maximizing the UART module. This could be by optimizing the current state machine-based module or by creating a new UART module using a different technique.

Overall, I think the project was successful and I gained much insight into the world of cryptography and FPGAs. I think that the project also can give insight about the Simon Cipher Algorithm and its advantages and disadvantages in addition to general understanding of cryptographic algorithms.

# Bibliography

|  |  |
| --- | --- |
| [1] | K. Richards, "What is cryptography?," [Online]. Available: https://www.techtarget.com/searchsecurity/definition/cryptography. |
| [2] | "Symmetric Key Encryption - why, where and how it’s used in banking," Jan 2020. [Online]. Available: https://www.cryptomathic.com/blog/symmetric-key-encryption-why-where-and-how-its-used-in-banking. |
| [3] | "Block cipher mode of operation," [Online]. Available: https://en.wikipedia.org/wiki/Block\_cipher\_mode\_of\_operation. |
| [4] | S. Kölbl, "Design and analysis of cryptographic algorithms," Technical University of Denmark, 2017. |
| [5] | K. Nyberg, "Generalized Feistel Networks," in *Advances in Cryptology - ASIACRYPT '96*, 1996, p. 91. |
| [6] | National Security Agency, "The Simon and Speck Families of Lightweight Block Ciphers," 2013. |
| [7] | Digilent, "Basys 3™ FPGA Board Reference Manual," Revised July 10, 2019 . |
| [8] | Xilinx-AMD, "7 Series FPGAs Data Sheet: Overview," September 2020. [Online]. Available: https://docs.amd.com/v/u/en-US/ds180\_7Series\_Overview. |
| [9] | Raspberry Pi Ltd, "Raspberry Pi 4 Model B," April 2024. |
| [10] | Xilinx-AMD, "Vivado Design Suite User Guide: Synthesis (UG901)," June 2024. |
| [11] | Xilinx-AMD, "Vivado Design Suite User Guide: Implementation (UG904)," June 2024. |

Links and explanations

GitHub for the project files - <https://github.com/LeorBrennerWork/Final_Project>

Explanations of each folder in the project:

Code drafts – versions of each individual module that I created before combining them

Documents – All non-code files used throughout the project

Python code – Code created using python (see ‎4.1.3 Python Controller)

Verilog Final – Final version of the Verilog implementation along with its Vivado implementation

VHDL Final – Final version of the VHDL implementation along with its Vivado implementation

Short Video about the project - <https://youtu.be/oei0a2Ahavg>

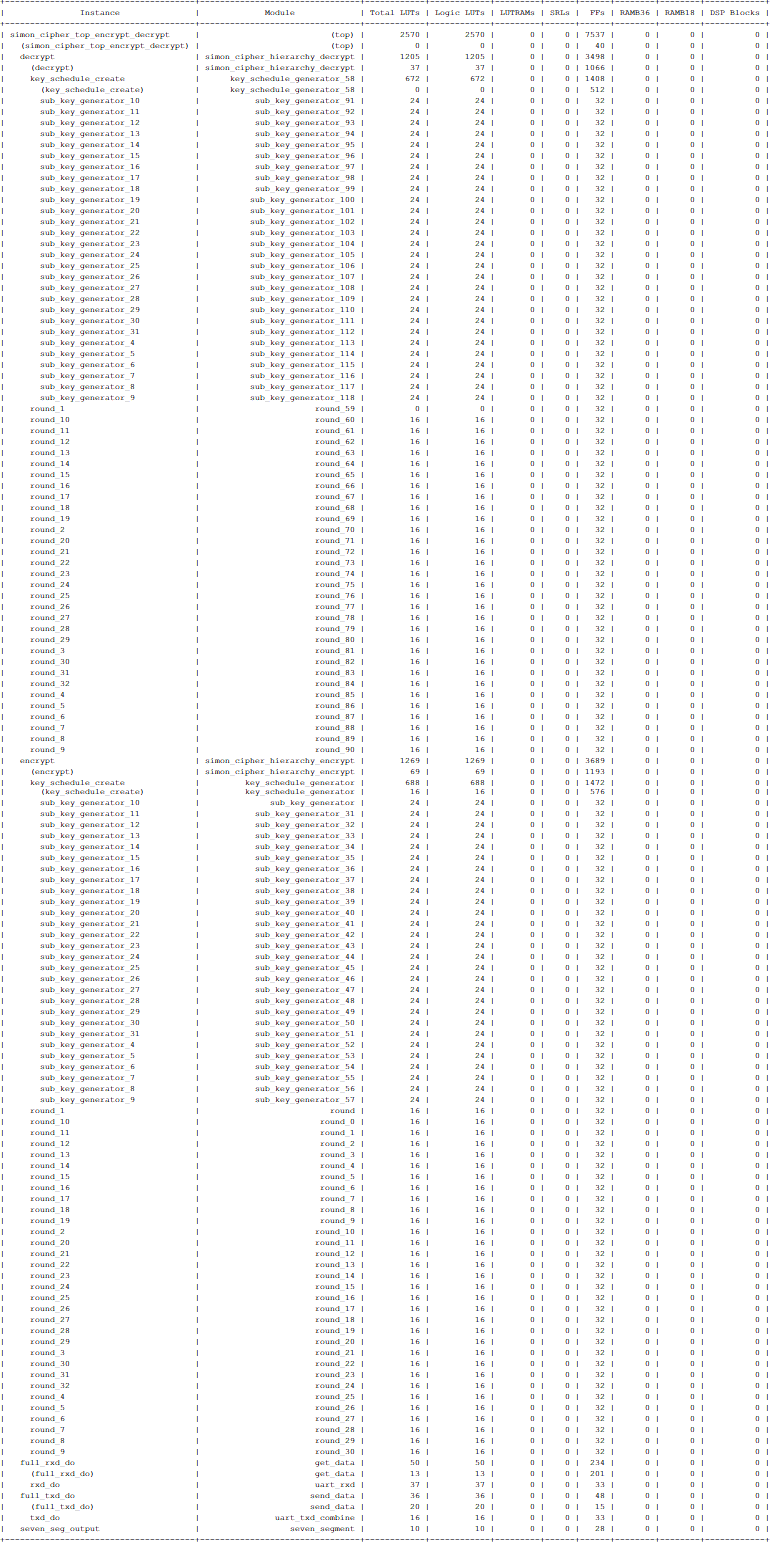


Figure ‎B.1 - Full VHDL post implementation area utilization



Figure ‎B.2 – Full Verilog post implementation area utilization