**DUBLIN CITY UNIVERSITY**
**SCHOOL OF ELECTRONIC ENGINEERING**

# Efficient FPGA Implementation of the lightweight cipher HIGHT

**Alex Doherty**
April 2019

BACHELOR OF ENGINEERING

IN

ELECTRONIC & COMPUTER ENGINEERING

Supervised by Dr. X. Wang

# Acknowledgements

I would like to thank my supervisor Dr. Xiaojun Wang for his guidance, enthusiasm and commitment to this project.

**IMPORTANT NOTE ON REFERENCES**
One area that some students fall down on is the area of references. The guideline here is quite simple: either you did the work and wrote the text, or *someone else did*. For any element of your dissertation, even the tiniest element, which falls into the latter category, you must provide as complete a reference as possible, so that another researcher can easily access exactly the same source of information as you have. The desired form for the reference data is usually that used in IEEE journals. Please examine carefully the references used in this document in the References section after section 6 that includes an example of how to reference a document from the Internet. (Please delete this note when using this document as a template). More recent versions of MS Word will allow you to 'Manage your Sources' under the 'References' tab and Bibtex is great if you are using Latex to prepare your document.

# Declaration

I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. I understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. I have read and understood the Assignment Regulations set out in the module documentation. I have identified and included the source of all facts, ideas, opinions, and viewpoints of others in the assignment references. Direct quotations from books, journal articles, internet sources, module text, or any other source whatsoever are acknowledged and the source cited are identified in the assignment references. This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study.

I have read and understood the DCU Academic Integrity and Plagiarism at https://www4.dcu.ie/sites/default/files/policy/1%20-%20integrity_and_plagiarism_ovpaa_v3.pdf and IEEE referencing guidelines found at https://loop.dcu.ie/mod/url/view.php?id=448779.

Name: Alex Doherty Date: 07/04/2019

# Abstract

In the modern world, with the increasing amount of devices being connected to the internet, there is a growing need for new and appropriate security. Lightweight cryptography has been introduced due to limitations in smaller devices containing data. In this report, the development of three different implementations of the lightweight cipher HIGHT are detailed. These implementations are built to run on hardware and encrypt a block of plaintext input into ciphertext.

These designs are written in the hardware description language VHDL and run on a Xilinx FPGA board. Results are obtained from these implementations which can be used to determine whether the cipher is suitable for devices or not.

# Table of Contents

**ACKNOWLEDGEMENTS** ..... **II**

**DECLARATION** ..... **II**

**ABSTRACT** ..... **III**

**CHAPTER 1 - INTRODUCTION** ..... **1**

**CHAPTER 2 - TECHNICAL BACKGROUND** ..... **3**

2.1 Equipment and Software ..... 3

*2.1.1 VHDL* ..... *3*

*2.1.2 Vivado* ..... *4*

*2.1.3 Zybo* ..... *4*

2.2 Cryptography ..... 6

2.3 HIGHT ..... 8

*2.3.2 Features of HIGHT* ..... *13*

*2.3.3 Operations* ..... *14*

2.4 Related Work ..... 15

2.5 Summary ..... 16

**CHAPTER 3 - DESIGN OF HIGHT** ..... **17**

3.1 Mixed Architecture ..... 18

3.2 FSM/Sequential Architecture ..... 20

3.3 Parallel Architecture ..... 22

3.4 Summary ..... 23

**CHAPTER 4- IMPLEMENTATION AND TESTING** ..... **24**

4.1 Implementation 1 ..... 24

4.2 Implementation 2 ..... 27

4.3 Implementation 3 ..... 28

4.4 Implementing on Hardware ..... 29

4.5 Testing ..... 30

*4.5.1 Testing of Implementation 1* ..... *30*

*4.4.2 Testing of Implementation 2 & 3* ..... *35*

4.5 Summary ..... 35

**CHAPTER 5 - RESULTS AND DISCUSSION** ..... **36**

5.1 SUMMARY .......... 39

**CHAPTER 6 – ETHICS .......... 40**

**CHAPTER 7 - CONCLUSIONS AND FURTHER RESEARCH .......... 41**

**REFERENCES .......... 42**

**APPENDIX 1 .......... 45**

# Table of Figures

TABLE 1: MEANING OF VARIABLES ........ 8
TABLE 2: MEANING OF OPERATOR SYMBOLS ........ 8
TABLE 3: XOR TRUTH TABLE ........ 14
TABLE 4: TEST VECTOR 1 INPUTS AND OUTPUT ........ 31
TABLE 5: TEST VECTOR 1 SUBKEYS ........ 31
TABLE 6: TEST VECTOR 1 ROUND RESULTS ........ 32
TABLE 7: TEST VECTOR 2 INPUTS AND OUTPUT ........ 32
TABLE 8: TEST VECTOR 2 SUBKEYS ........ 33
TABLE 9: TEST VECTOR 2 ROUND RESULTS ........ 34
TABLE 10: POWER USAGE COMPARISON ........ 36
TABLE 11: AREA UTILISATION COMPARISON ........ 37
TABLE 12: PERFORMANCE COMPARISON ........ 37
TABLE 13: RESOURCE USAGE AND PERFORMANCE COMPARISON ........ 38
TABLE 14: POWER AND ENERGY CONSUMPTION COMPARISON ........ 38

FIGURE 1.1: ZYBO ZYNQ-7000 DEVELOPMENT BOARD ........ 5
FIGURE 2.1: SUBKEY GENERATION PROCESS ........ 10
FIGURE 2.2: INITIAL TRANSFORMATION PROCESS ........ 11
FIGURE 2.3: ROUND FUNCTION PROCESS ........ 12
FIGURE 2.4: FINAL TRANSFORMATION PROCESS ........ 13
FIGURE 2.5: BITWISE SHIFT ROTATION ........ 15
FIGURE 2.6: COMPARISON OF DIFFERENT CIPHERS FOR PERFORMANCE AND RESOURCES 16
FIGURE 3.1: VISUAL REPRESENTATION OF ENCRYPTION PROCESS ........ 18
FIGURE 3.2: HIGHT DESIGN STRUCTURE AND MODULE INTERACTION ........ 19
FIGURE 3.3: FSM SEQUENCE OF OPERATION ........ 20
FIGURE 3.4: HIGHT DESIGN STRUCTURE AND FUNCTION INTERACTION ........ 22
FIGURE 3.5: PARALLEL ARCHITECTURE DESIGN FOR HIGHT ........ 23
FIGURE 4.1: HIGHT MODULE CONTAINING THE REST OF THE MODULES ........ 24
FIGURE 4.2: BIDIRECTIONAL_VECTOR SUBTYPE STRUCTURE ........ 26
FIGURE 4.3: HIERARCHY OF FILES ........ 27
FIGURE 4.4: HIGHT MODULE CONTAINING THE FUNCTIONS FILE, WHICH CONTAINS ALL THE FUNCTIONS ........ 27
FIGURE 4.5: FLOW OF DATA THROUGH IMPLEMENTATION 3 ........ 29
FIGURE 4.6: BLOCK DIAGRAM CONNECTING IP WITH ZYNQ ........ 30

# Chapter 1 - Introduction

This report addresses the implementation of the lightweight hardware block cipher called HIGHT (HIGH security and lightweighT). This was designed using the hardware description language VHDL. The cipher would then be implemented using a Zybo FPGA board. The encryption cipher had to be evaluated in terms of performance and resource usage as well as power and energy consumption.

With the rise of the Internet of Things (IoT) market and having a more connected world, there is an increase in devices which are connected to the internet and transmitting information. A lot of these devices are constrained and have less resources available to them. This means that there is a need for lightweight cryptography, to cater these kinds of devices. These devices can also be seen as an area to target by potential attackers, due to the lack of a standardised lightweight encryption cipher. This is due to it being a relatively new field. Cryptography will allow for the protection of your data and the usage of these devices.

Lightweight cryptography may be used in important industries such as healthcare, finance and military. It is of utmost importance that devices used by these industries cannot be breached. For this and other reasons outlined in the report, it is imperative to be able to design a suitable cipher for these demands.

This paper discusses the hardware implementation issues of the cipher HIGHT. Three different designs of the cipher have been implemented. These designs are presented and discussed, detailing their advantages and disadvantages. The technical background section details the software and equipment used to implement this design, along with background information on the area of cryptography and the encryption process of the HIGHT cipher. The design section discusses different possible methods of implementing the project, with reasoning as to why the design would be chosen. The implementations and testing section outlines how the designs of the project were created, with technical details on how they were implemented and the reasons for being created this way. This section also contains information on the testing of the project to ensure it encrypted the data correctly. The results are presented and reviewed in the results and discussion section, with a discussion as to what they mean and a comparison of the results between the different implementations. Finally, the ethics of lightweight cryptography is discussed, giving information on how the use of cryptography is, in certain areas, an ethical decision.

# Chapter 2 - Technical Background

This section addresses the HIGHT algorithm, along with the equipment and software used to design it. The area of cryptography is discussed with a focus on why lightweight cryptography is an important area for the future. A technical description on the different aspects in the encryption process is discussed. Finally, results from previous implementations of this cipher are reviewed.

## 2.1 Equipment and Software

### 2.1.1 VHDL

VHDL is the language used for this project to describe the hardware for the cipher. VHDL is a nested acronym which stands for VHSIC Hardware Description Language. VHSIC stands for Very High-Speed Integrated Circuit. It was developed by the US Government in the 1980's in an attempt to further research and develop high-speed integrated circuits. The first official standard for the language was released by IEEE in 1987.

VHDL is a hardware description language which is used to design and describe circuitry. As it is a language which describes a physical structure, much like a real structure there will be different modules to it which are running concurrently. It is used to design hardware for field-programmable gate arrays (FPGA) and application-specific integrated circuits (ASIC). Unlike software programming languages, rather than only executing commands sequentially, it can execute multiple modules which contain instructions simultaneously. [1]

VHDL simulation is event-driven. Event driven simulation is based on four main concepts. These concepts are simulation time, transaction generation, event processing and delta time. Simulation time is the time which the simulator models and it is an integral multiple of the resolution limit. The simulator cannot measure time delays less than the simulator resolution. A transaction is added to a queue when generated for a scheduled time. Each event generation phase consumes one time-steps transactions. Delta delay effects occur after an infinitesimally small delay. It can be used to handle zero delay transactions. Delta delay does not update the simulator time therefore an infinite amount of delta times is still zero for simulation time. [2] VHDL allows you to create logic systems, large or small scale. It does this using three primary design units and two secondary design units. The primary design units are entity declaration, package declaration and configuration declaration. The secondary design units are architecture body and package body. These design units can all be compiled independently. [3] IDE’s such as Vivado [4] and Modelsim [5] can be used for the writing, simulation and debugging of the model you are developing.

### 2.1.2 Vivado

Vivado is the integrated development environment (IDE) which will be used for this project. This can be used to write the VHDL that describes the project. It can also be used for other purposes, such as performing a power analysis on the system you have described, Register-Transfer Level (RTL) analysis and creating and packaging an intellectual property (IP) for the system you have created. The power analysis gives a detailed report on the power consumption of the system. It shows the amount of power consumed by different aspects in the system, such as the signals, the clocks, the inputs/outputs and the logic implemented in the system. RTL analysis can be used to create an elaborated design of the system you created. In this model, the system that has been described can be seen with details such as the registers used, the logic used and the wires which connect the ports. Creating the IP can be used to connect the design which has been created in VHDL to the Zynq processor. This can be done by creating a block design to connect your IP to the Zynq processor. Vivado can then generate the HDL for the block design which you have designed.

### 2.1.3 Zybo

The Field Programmable Gate Array (FPGA) board used in this project is a Zybo Zynq-7000 ARM/FPGA Development Board. [6] This FPGA is designed by the same company who created the Vivado IDE so it is suitable to be using them together. This board contains an Advanced RISC Machine (ARM) processor and an FPGA, which can be used to host a system design. FPGAs are semiconductor devices which are based on a matrix of configurable logic blocks. The hardware within an FPGA can be reprogrammed for the desired application. [7]

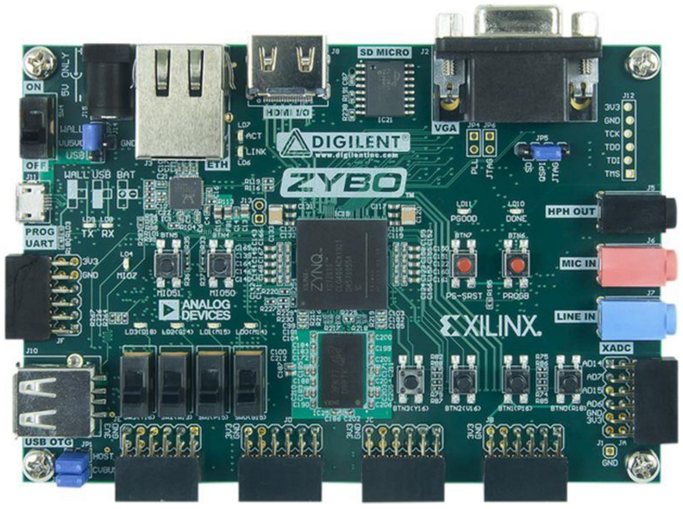


**Figure 1.1:** Zybo Zynq-7000 Development Board

## 2.2 Cryptography

This project deals with encrypting data which is input to the HIGHT cipher. This means that an algorithm is used to encrypt the original information so that it should not be able to be recovered by external sources. The data which goes into an encryption cipher is called the plaintext. The encrypted information which comes out of the cipher is referred to as the ciphertext. The purpose of cryptography is to keep your data secret. There are three areas which are closely related to each other. These areas are cryptography, cryptanalysis and cryptology.

Cryptography is the protecting of information, generally when it is being stored or it is being communicated. Cryptography uses techniques derived from mathematical concepts to encrypt and transform these messages in a way which is difficult to recover the original information from them. There are four main objectives in modern cryptography. These objectives are confidentiality, integrity, non-repudiation and authentication. Confidentiality means the information cannot be recovered or understood by anyone who the information is not intended for. Integrity means the information cannot be changed in any way when either in storage or being transferred between a source and receiver. Non-repudiation means the sender/creator of the information cannot deny their intentions at a later stage as the information is exactly as it was when it was sent/created. Authentication means the sender/receiver can confirm the identity of the other. [8]

Cryptanalysis focuses on the ciphertext and ciphers in an attempt to understand them so that they can be weakened or defeated. This is done to recover information which has likely been encrypted for security or privacy reasons. Information being encrypted is generally valuable, which means there will sometimes be people attempting to access this information. Cryptanalysis can also be used by cryptographers to strengthen ciphers. [9]

Cryptology is the mathematics which is applied in the areas of cryptography and cryptanalysis. It is important that the information which is being stored or transmitted is difficult to decipher. This means the mathematics used to encrypt and secure them must be difficult to solve unless certain criteria are met. The level of difficulty of solving one of these equations is called its intractability. [10]

A cipher is the process which encrypts your data. These are used in electronic devices to prevent potential attackers from accessing your information. This report reviews the area of lightweight cryptography. Lightweight cryptography is a field of cryptography that aims at devices which have less resources available to them. These resources include power, memory, size, etc. These metrics have an effect on the performance of the cipher. Ciphers that have already been implemented in devices with more resources available to them are not optimisable. This has led to the need to develop ciphers specifically for these lightweight devices. Lightweight cryptography is increasingly becoming a more important area due to the rise of the "Internet of Things" (IoT), which is the connecting of devices in everyday objects so that they can send and receive data. This has led to the demand for lightweight cryptography. Due to the nature of IoT, it has limited computational abilities. Lightweight cryptography is extremely important for this industry as it may be used to withhold the privacy for industries such as finance, healthcare or the military. One of the main challenges in implementing security such as lightweight ciphers is finding a balance in the various requirements for the security. It is important to have a strong cipher but power consumption, execution speed and implementation costs must also be considered. [11] A variety of lightweight ciphers have been developed with different advantages. Due to the required balancing of requirements in lightweight cryptography, it is important that an appropriate cipher is chosen.

Lightweight cryptography can be used in ubiquitous sensor networks (USN) and radio-frequency identification (RFID). Cryptography is important in sensor networks. The reason for this is that the sensors have channels in which they connect to each other and a central hub. These channels can be used to transmit information between them. The cryptographic algorithms will come into use in these channels to uphold the sensor networks security and for authenticity. RFID tags can be used for identification of devices, people, etc. While RFID tags are being used in many applications already, there is the potential for them to be used in a lot more in the near future. RFID can be used in automated electronic toll systems, ID tags for food, pets, clothing, etc. [12] RFID tags can contain sensitive data which should not be accessible by unwanted personnel. With the potential for RFIDs and wireless sensor network nodes being more widely used in the modern world, this leads to a concern for the security and privacy of the information being stored and transmitted. Breaching the security of RFID tags could also lead to forgery of tags, tracking of consumers using the tags and many other problems. This ID will be encrypted. [13]

## 2.3 HIGHT

HIGHT [14] is a block cipher with a 64-bit block length and 128-byte key length. It was developed by a group from Korea University, National Security Research Institute (NSRI) and Korea Information Security Agency (KISA) in 2006. The 64-bit input is the plain text while the master key input is 128-bit. A block cipher encrypts blocks of text rather than encrypting one bit at a time, so HIGHT takes the 64-bit plaintext input and returns a 64-bit encrypted ciphertext. It is suitable for low-cost, low-power and ultra-light implementation. This is due to the absence of a traditional substitution layer, its Feistel structure and byte-oriented operations. [12] A Feistel structure for a cipher is a symmetric structure for a block cipher. This allows for the encryption and decryption algorithms to be almost identical, meaning there will be a large reduction in resources needed to implement it. HIGHT consists of simple operations to create its algorithm. These operations are XOR, left bitwise rotation and addition mod $2^8$.

**Table 1: Meaning of Variables**

| **Variable** | **Meaning** |
|---|---|
| P | The 64-bit plaintext input to be encrypted |
| C | The 64-bit ciphertext output |
| K | The 128-bit master key |
| WK | The 64-bit whitening keys block |
| SK | The 1024-bit subkey block |
| δ | Constant vector generated by LFSR |
| X | Intermediate block of output of each round |

**Table 2: Meaning of Operator Symbols**

| **Symbol** | **Operator** |
|---|---|
| $\oplus$ | XOR |
| $\lll_n$ | n-bit shift rotation |
| [+] | Addition in modular $2^8$ |

HIGHT has a thirty-two round iterative structure which is a variant of generalised Feistel Network. There are two main blocks in the cipher, the key schedule and the round block. The key schedule takes in the 128-bit master key and returns the master key once the key schedule is finished. The 128-bit master key is split into sixteen 8-bit bytes $K_{0,}K_{1,}\ldots,K_{15}$. The key schedule generates whitening keys and subkeys. Whitening is the process of XORing a key with the input to a block algorithm and then XORing other keys with the output of a block algorithm. This is to prevent a cryptanalyst from obtaining the plaintext/ciphertext pair. [15] Subkeys are keys which are bound to the master key. These are used in the thirty-two iterative rounds. The whitening keys are generated by taking bytes from the master key as shown in equations 1 and 2.

$$WK_i = K_{i+12}\ ; for\ i = 0, 1, 2, 3 \tag{1}$$

$$WK_i = K_{i-4}\ ; for\ i = 4, 5, 6, 7 \tag{2}$$

The 7-bit constant δ is used in subkey generation. The first 7-bit byte is pre-defined. The 7-bit constants $\delta_0$, $\delta_{1,\ldots,}$ $\delta_{127}$ are generated by the 7-bit linear-feedback shift register h. The characteristic polynomial of h is $x^7 + x^3 + 1$ with a period of $2^7$-1=127. This is done by concatenating the values of $S_i$. The initial value of h is set to $1011010_2$. [12] This process is done using the equations 3 and 4.

$$S_{i+6} = S_{i+2} \oplus S_{i-1}\ ; for\ i = 1\ to\ 127 \tag{3}$$

$$\delta_i = S_{i+6}||S_{i+5}||S_{i+4}||S_{i+3}||S_{i+2}||S_{i+1}||S_{i+0}; for\ i = 1\ to\ 127 \tag{4}$$

The subkeys are subsequently generated using equations 5 and 6.

$$SK_{16i+j} = K_{j-i\ mod\ 8}[+]\delta_{16i+j};\ i = 0\ to\ 7, j = 0\ to\ 7 \tag{5}$$

$$SK_{16i+j+8} = K_{(j-i\ mod\ 8)+8}[+]\delta_{16i+j+8};\ i = 0\ to\ 7, j = 0\ to\ 7 \tag{6}$$

The whitening keys and subkeys are the outputs of the key schedule as they are used in the encryption process.

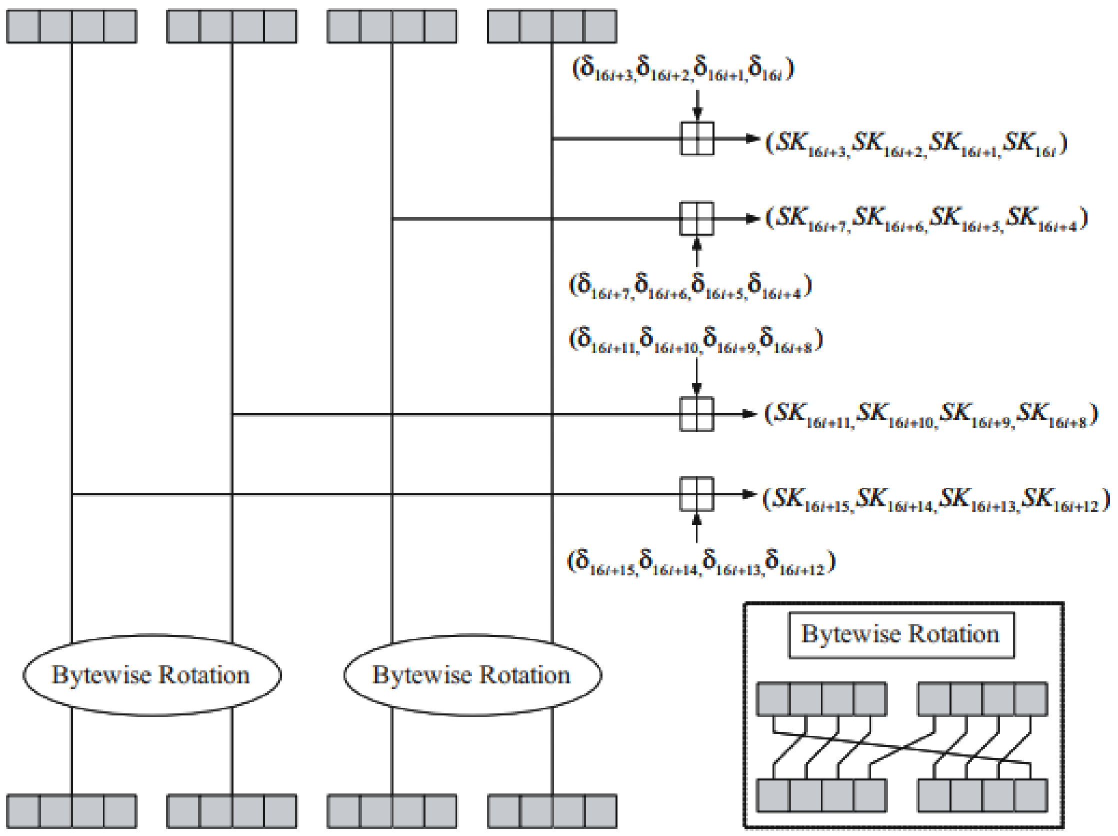

$(\delta_{16i+3},\delta_{16i+2},\delta_{16i+1},\delta_{16i})$
$(SK_{16i+3},SK_{16i+2},SK_{16i+1},SK_{16i})$
$(SK_{16i+7},SK_{16i+6},SK_{16i+5},SK_{16i+4})$
$(\delta_{16i+7},\delta_{16i+6},\delta_{16i+5},\delta_{16i+4})$
$(\delta_{16i+11},\delta_{16i+10},\delta_{16i+9},\delta_{16i+8})$
$(SK_{16i+11},SK_{16i+10},SK_{16i+9},SK_{16i+8})$
$(SK_{16i+15},SK_{16i+14},SK_{16i+13},SK_{16i+12})$
$(\delta_{16i+15},\delta_{16i+14},\delta_{16i+13},\delta_{16i+12})$
Bytewise Rotation
Bytewise Rotation
Bytewise Rotation

**Figure 2.1:** Subkey generation process [14]

The round block consists of three stages, the initial transformation, the round function and the final transformation. These three stages combine for the round block to make up the encryption process of the plaintext to ciphertext in this cipher. The 64-bit plaintext input is divided up into eight 8-bit bytes $P_0$, $P_1$,…,$P_7$. The initial transformation is the first round of encryption in the cipher. Four whitening key bytes $WK_0$, $WK_1$, $WK_2$ and $WK_3$ are used in the initial transform to transform the plaintext into the input for the first round of the round function. It uses the XOR and addition mod $2^8$ operations with the plaintext input and whitening keys to encrypt the input.

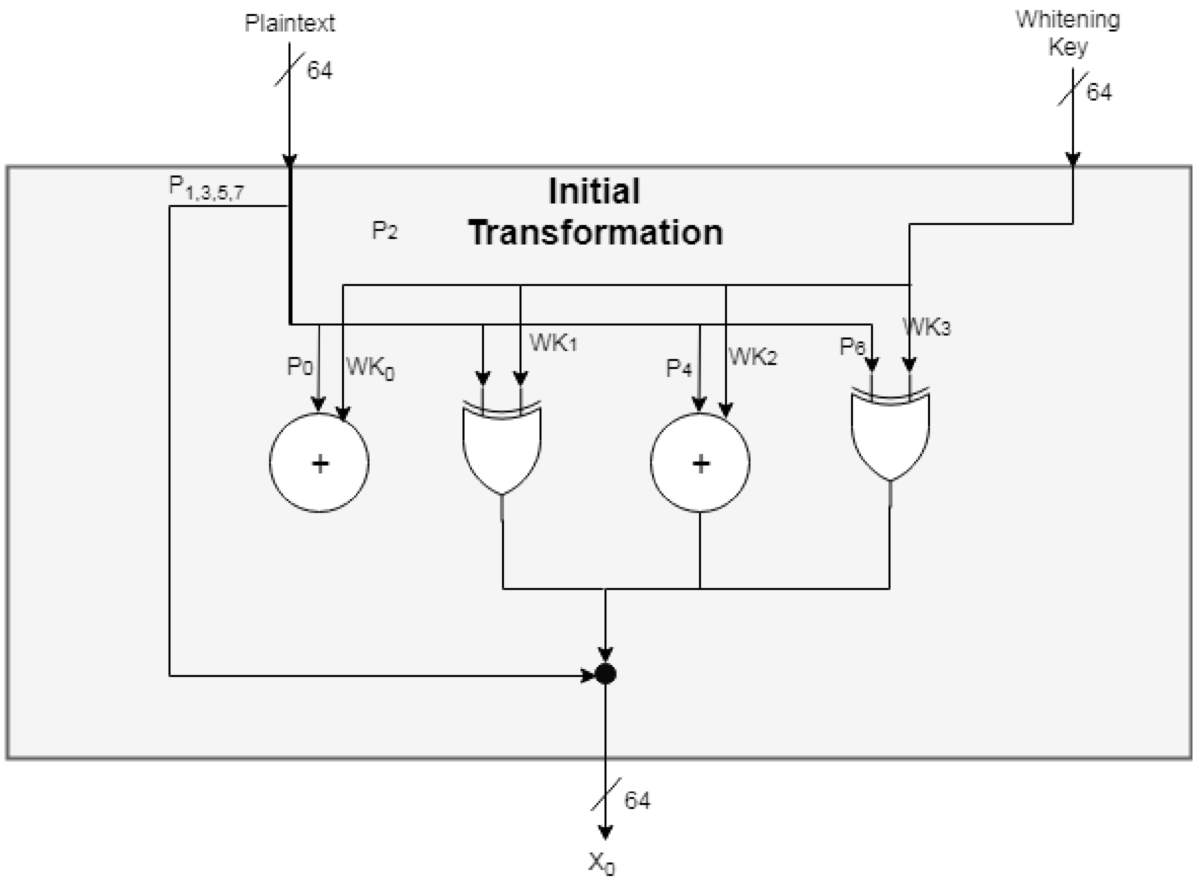


**Figure 2.2:** Initial Transformation process

The round function consists of thirty-two rounds to encrypt the input. The result of the initial transformation is the input to the round function. This goes through thirty-two rounds to further encrypt the plaintext. This is done using XOR and addition mod $2^8$ operations with the subkeys. Two supplementary functions are used by the round function. These functions provide bitwise diffusion by using left bitwise rotation and XORing the results of these operations. This is described by equations 7 and 8.

$$F0(x) = x \lll_1 \oplus x \lll_2 \oplus x \lll_7 \tag{7}$$

$$F1(x) = x \lll_3 \oplus x \lll_4 \oplus x \lll_6 \tag{8}$$

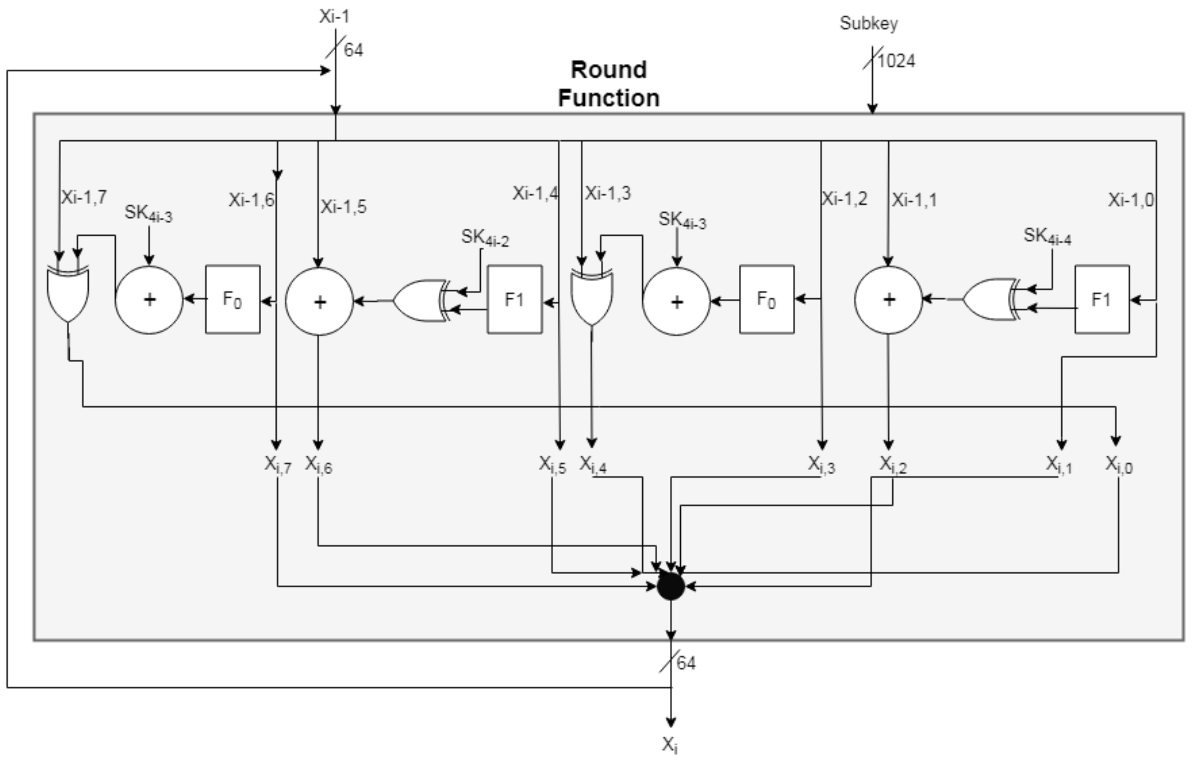


**Figure 2.3:** Round Function process

The output of the round function then enters as an input to the final transformation. Four whitening key bytes $WK_4$, $WK_5$, $WK_6$ and $WK_7$ are used in the final transform to transform the output of the round block into the ciphertext. Similarly to the initial transformation, the final transformation uses the XOR and addition mod $2^8$ operations with the input and whitening keys to attain the ciphertext. The result of this transformation will give you the ciphertext which is the encrypted output of the HIGHT system. The result is eight 8-bit bytes $C_0, C_1, \ldots, C_7$., which when concatenated will give you the 64-bit output ciphertext.

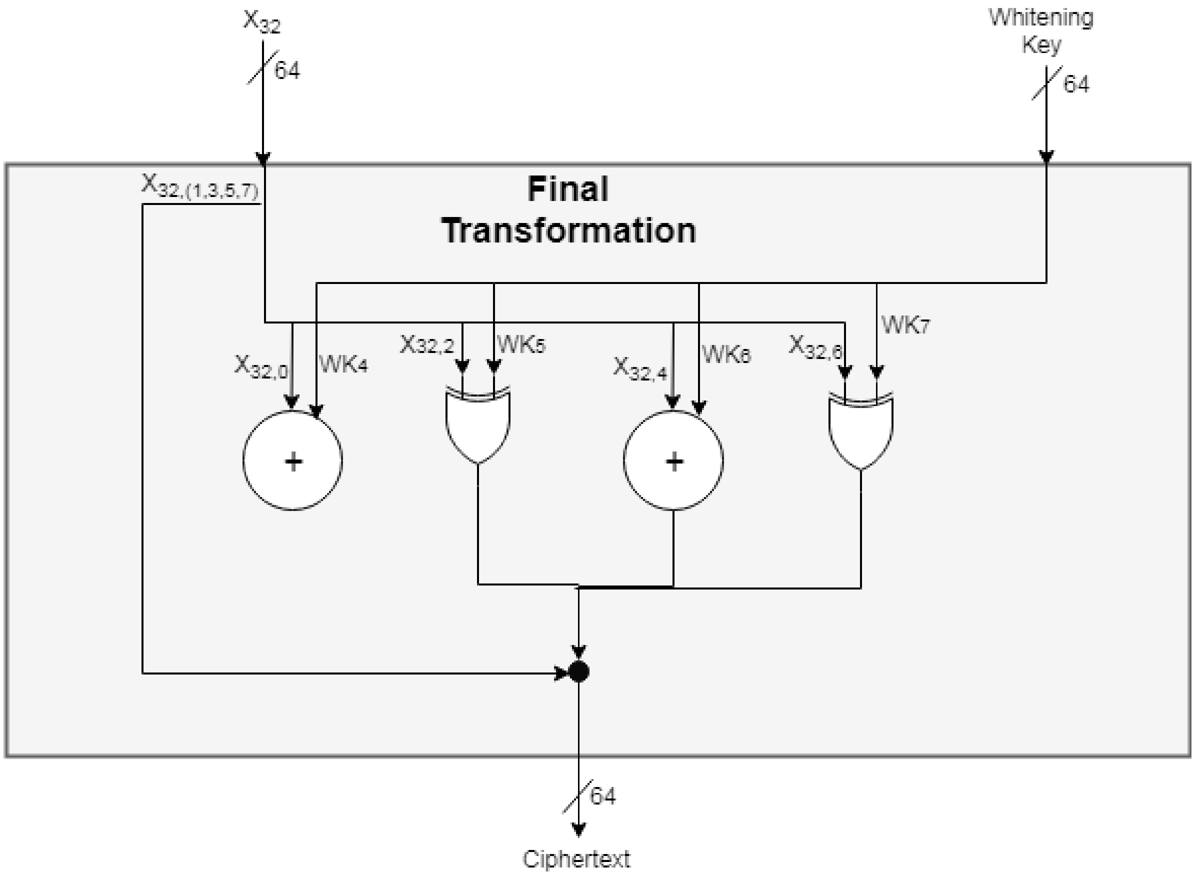


**Figure 2.4:** Final Transformation process

The HIGHT decryption process has two key changes to the encryption process. The first change is that the order in which the whitening keys and subkeys are applied is reversed. The second change is that the $2^8$ modular addition is replaced by $2^8$ modular subtraction, except for the modular addition operations that connect subkeys and outputs of auxiliary function F0.

## 2.3.2 Features of HIGHT

HIGHT uses a 64-bit input/output data block size. Being a block cipher is an advantage to have as it will encrypt more information at a time. There are many ciphers which will encrypt data bit-by-bit, which would take significantly longer. It has a 128-bit key length and outputs the same key from the key schedule. It is a thirty-two-round structure with XOR, modular addition and shift rotation operations. There is no S-box substitution in HIGHT unlike many other block ciphers. It is designed for low-resource devices, such as low power, data storage, etc.

### 2.3.3 Operations

The HIGHT cipher uses three different operations in its encryption process, these are bitwise exclusive OR (XOR), modular addition and shift rotation. The truth table for XOR is shown in the following table. This table uses two input bits with one output bit. It shows how the logic of the XOR operation works. For XOR gates, if one and only one of the inputs is set to high (1), the output will result in a high (1). For any other case, the output will result in a low (0).

**Table 3: XOR Truth Table**

| **INPUTS** | | **OUTPUTS** |
|---|---|---|
| **X** | **Y** | **Z** |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Modular addition is used in this project exclusively as addition mod $2^8$. This means it is 8-bit addition. In binary addition, you can get an extra bit called a carry bit as a result of adding two bits together. For this project, any carry bit which is the result of the 8th bit of the modular addition will be disregarded. The binary addition example in (9) shows 8-bit addition with the carry being disregarded.

$$\begin{array}{ccccccccc}
 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
+ & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\
\hline
\cancel{1} & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1
\end{array} \tag{9}$$

For the decryption process, the same operations are used except for the modular $2^8$ addition being replaced by modular $2^8$ subtraction.

The final operation which is used in this project is left circular shift rotation by n-bits in 8-bit values. This shifts the bits to the left by n-bits. If a bit is shifted past the 8-bits, it loops back to the start. The following diagram shows how the operation works for an 8-bit value being shifted by three bits.

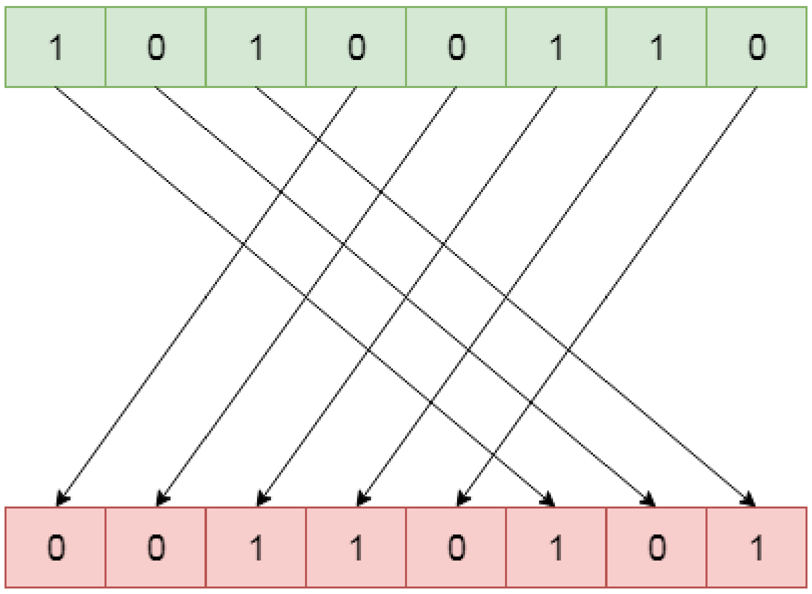
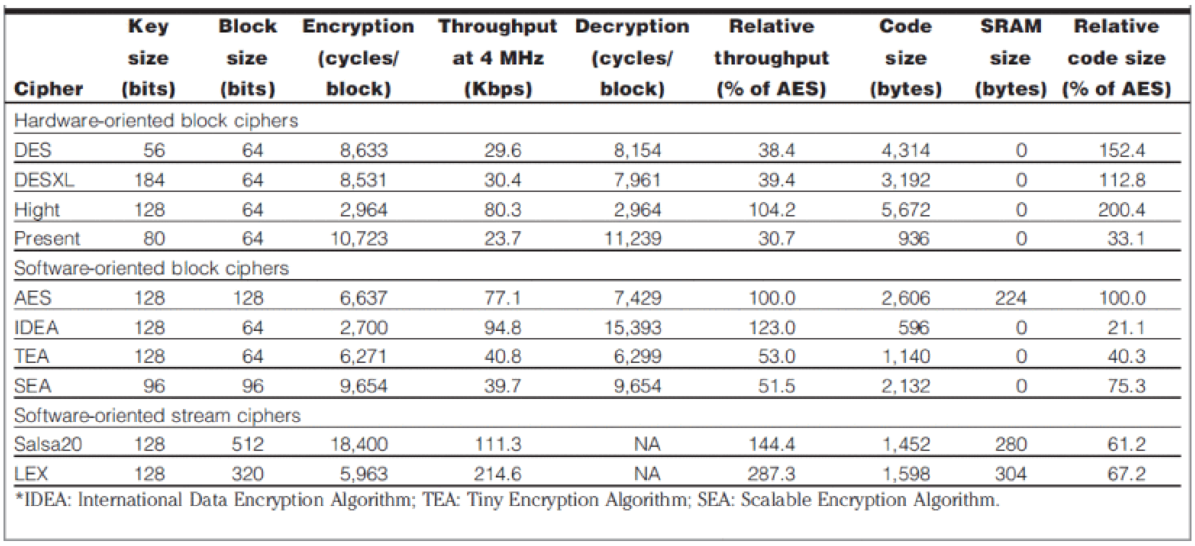


**Figure 2.5:** Bitwise Shift Rotation

## 2.4 Related Work

Eisenbarth et al. did a comparison of various lightweight cryptographic ciphers, analysing their performance in different aspects such as throughput, cycles/block for encryption/decryption, code size, etc. This comparison was done between different hardware, software and software-oriented ciphers. The software implementation comparison done shows that decreasing the code size of HIGHT had little effect on its encryption strength.

HIGHT also yielded the greatest throughput of the hardware ciphers at a clock rate of 4MHz. [16]



| Cipher | Key size (bits) | Block size (bits) | Encryption (cycles/ block) | Throughput at 4 MHz (Kbps) | Decryption (cycles/ block) | Relative throughput (% of AES) | Code size (bytes) | SRAM size (bytes) | Relative code size (% of AES) |
|---|---|---|---|---|---|---|---|---|---|
| Hardware-oriented block ciphers | | | | | | | | | |
| DES | 56 | 64 | 8,633 | 29.6 | 8,154 | 38.4 | 4,314 | 0 | 152.4 |
| DESXL | 184 | 64 | 8,531 | 30.4 | 7,961 | 39.4 | 3,192 | 0 | 112.8 |
| Hight | 128 | 64 | 2,964 | 80.3 | 2,964 | 104.2 | 5,672 | 0 | 200.4 |
| Present | 80 | 64 | 10,723 | 23.7 | 11,239 | 30.7 | 936 | 0 | 33.1 |
| Software-oriented block ciphers | | | | | | | | | |
| AES | 128 | 128 | 6,637 | 77.1 | 7,429 | 100.0 | 2,606 | 224 | 100.0 |
| IDEA | 128 | 64 | 2,700 | 94.8 | 15,393 | 123.0 | 596 | 0 | 21.1 |
| TEA | 128 | 64 | 6,271 | 40.8 | 6,299 | 53.0 | 1,140 | 0 | 40.3 |
| SEA | 96 | 96 | 9,654 | 39.7 | 9,654 | 51.5 | 2,132 | 0 | 75.3 |
| Software-oriented stream ciphers | | | | | | | | | |
| Salsa20 | 128 | 512 | 18,400 | 111.3 | NA | 144.4 | 1,452 | 280 | 61.2 |
| LEX | 128 | 320 | 5,963 | 214.6 | NA | 287.3 | 1,598 | 304 | 67.2 |

*IDEA: International Data Encryption Algorithm; TEA: Tiny Encryption Algorithm; SEA: Scalable Encryption Algorithm.

**Figure 2.6:** Comparison of different ciphers for performance and resources [16]

Yalla and Kaps implemented the block ciphers HIGHT and PRESENT for Xilinx Spartan3 FPGAs. They described these ciphers in VHDL. The results from their implementation yielded good results for HIGHT with an area of 91 slices and a throughput of 65.48Mbps. This meant that their implementation of HIGHT had a better throughput over area ratio than a previously published implementation of AES. [12]

## 2.5 Summary

This section gave an overview and explained the purpose of the equipment and software used for this project. A background on cryptography gives an idea of what it is and why it is needed. The description of the algorithm and how it encrypts the data that goes into it was given, with detailed information on how the algorithm is broken down and how each operation works. Finally, research was done into implementations and tests of this cipher which have previously been done. This section gives a better understanding of why the algorithm is needed, which helps to give an idea of how the system could be designed.

# Chapter 3 - Design of HIGHT

This section discusses different designs of the HIGHT algorithm, along with their respective advantages and disadvantages. Diagrams show visually how the plaintext is processed in the system to result in the ciphertext. The initial design of the HIGHT cipher was chosen to be implemented in the simplest way possible. The priority for this project is to have the cipher encrypting the text correctly so, resulting from this, different architectures for the algorithm and structures for how to design systems in VHDL are considered.

## 3.1 Mixed Architecture

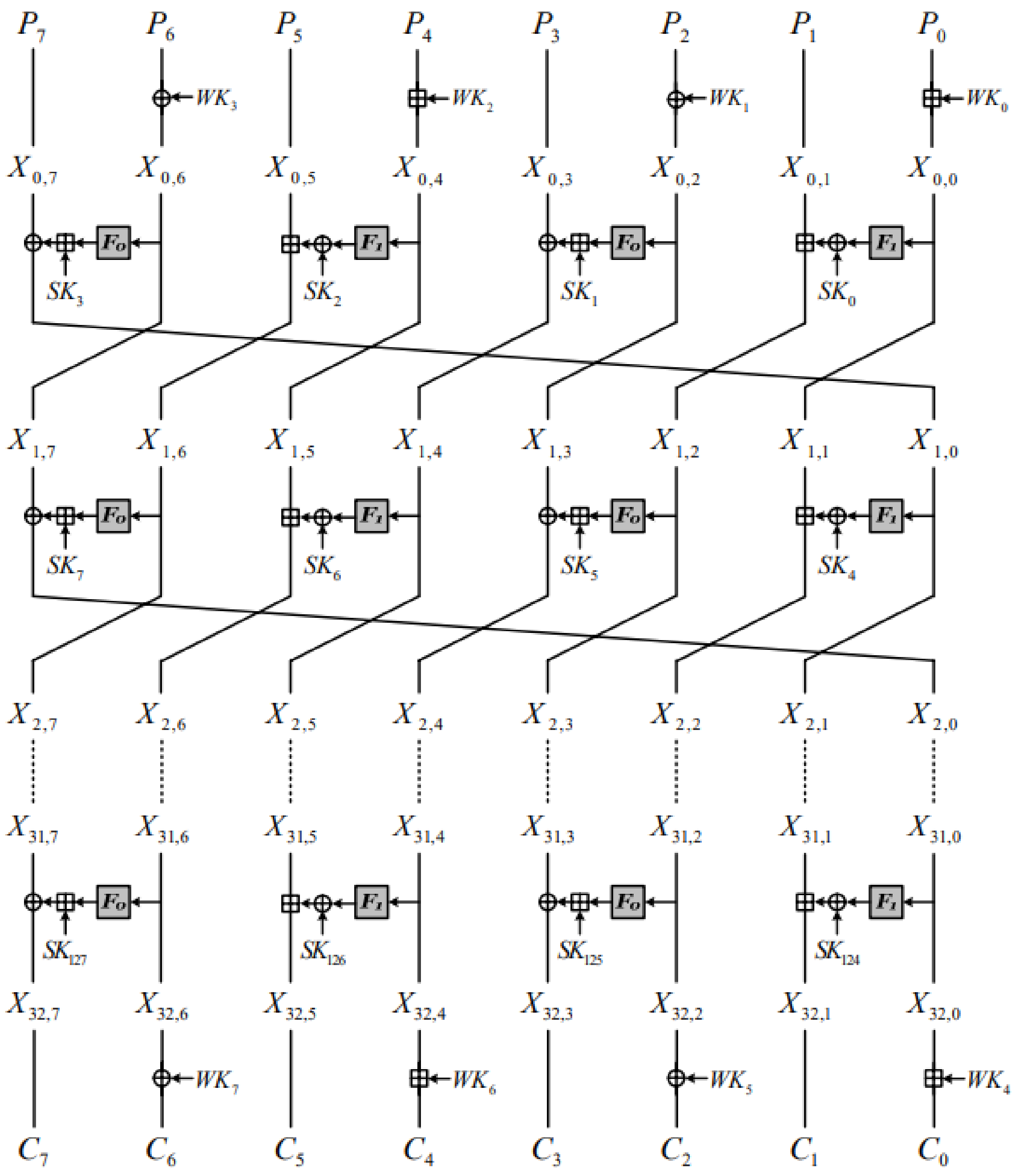

$P_7$ $P_6$ $P_5$ $P_4$ $P_3$ $P_2$ $P_1$ $P_0$
$WK_3$ $WK_2$ $WK_1$ $WK_0$
$X_{0,7}$ … $X_{0,0}$
$F_0$ $F_1$
$SK_3$ $SK_2$ $SK_1$ $SK_0$
$X_{1,7}$ … $X_{1,0}$
$SK_7$ $SK_6$ $SK_5$ $SK_4$
$X_{2,7}$ … $X_{2,0}$
$X_{31,7}$ … $X_{31,0}$
$SK_{127}$ $SK_{126}$ $SK_{125}$ $SK_{124}$
$X_{32,7}$ … $X_{32,0}$
$WK_7$ $WK_6$ $WK_5$ $WK_4$
$C_7$ $C_6$ $C_5$ $C_4$ $C_3$ $C_2$ $C_1$ $C_0$

**Figure 3.1:** Visual representation of encryption process [17]

This was built in a mixed style description. The project is structured in a hierarchical manner. The overlaying component in the design is the HIGHT component which will contain the key schedule component and the components for the different stages of the encryption process, the initial transformation, the round function and the final transformation. The key schedule will be used for the encryption process and the output of the encryption process will be returned to the overlaying HIGHT component. This modular design of the project allows low-level functionality to be in each of these modules. In each of these components, there is a data flow style as the order of execution of the statements is not important. Values in this construct would update in response to a clock signal. This means that the inputs and outputs for each component will be updated at the same time, based on the clock signal.

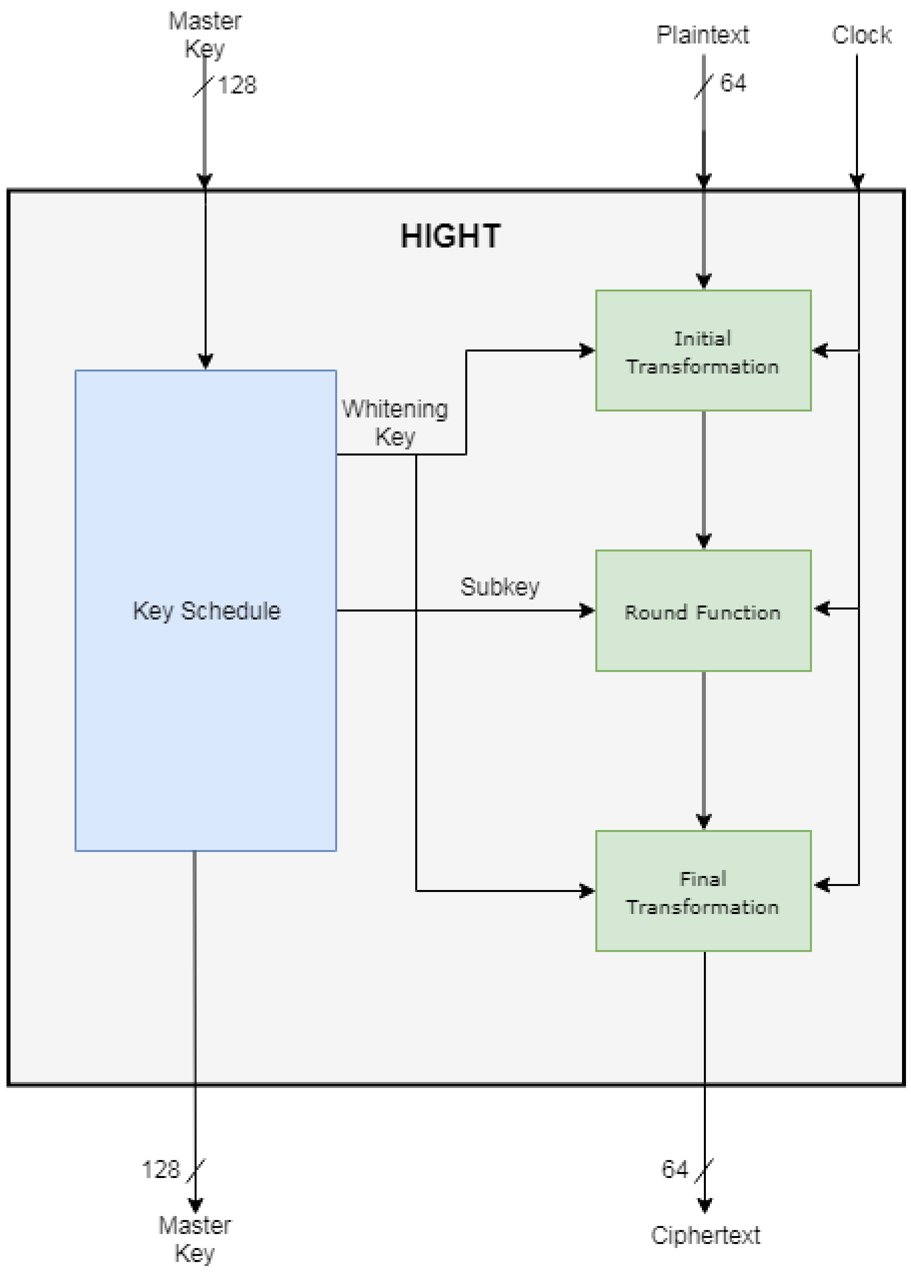


**Figure 3.2:** HIGHT design structure and module interaction

The simple modular design of the system serves as a good starting point for understanding the system and building it. As a counterpoint, the clock signal will be used in each individual module, which will lead to a longer processing time for the encryption.

## 3.2 FSM/Sequential Architecture

A Finite State Machine (FSM) architecture of this system is another approach to the modelling of the system. The FSM imposes an order on the structure of the functions. FSMs are sequential. The design described is a simple FSM which moves from one state to another in a sequential sequence. Figure 3.3 below shows the sequence of the FSM, with the states it has and changes in between.

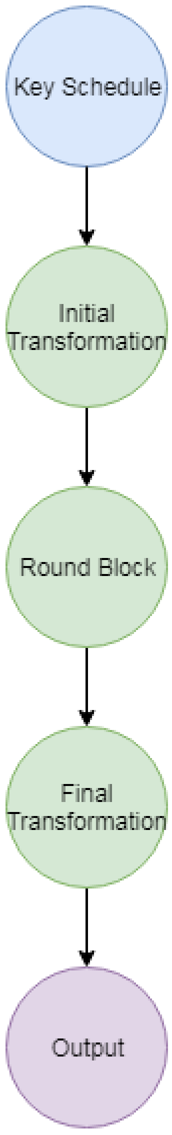


**Figure 3.3:** FSM Sequence of operation

The previous design is easily adaptable into an FSM system to control the flow of the system. An FSM has a state which the system is in. There is a fixed amount of states which it can be in. For this project, the different states the machine can be in are key schedule, initial transformation, round block, final transformation and output. The machine can only be in one of these states at a time. The next state of the system is reached depending on the current state. This architecture of the system can only process and encrypt 64 bits at a time as it will only accept a new input once the current encryption process is finished to produce the ciphertext. There are two inputs to the system in this architecture, the 64-bit plaintext input and the 128-bit master key input. The master key is an input to the key schedule while the plaintext is an input to the initial transformation, which is round zero of the encryption of the cipher. The key schedule will run first for the generation of the whitening and subkeys. These keys are needed for the encryption of the cipher, so it is run before the encryption. The encryption will run following the key schedule. The updating of the values is on the rising edge of the clock pulse. This means changes are all synchronised to begin at the same time and there are regular intervals. This is particularly useful for this project as it has a thirty-two round iterative encryption process, so it is important for the changes to update together and for there to be a consistent wait time between the updating of signals. An advantage of having a sequential logic system is in the idea behind it, in which it has a memory.

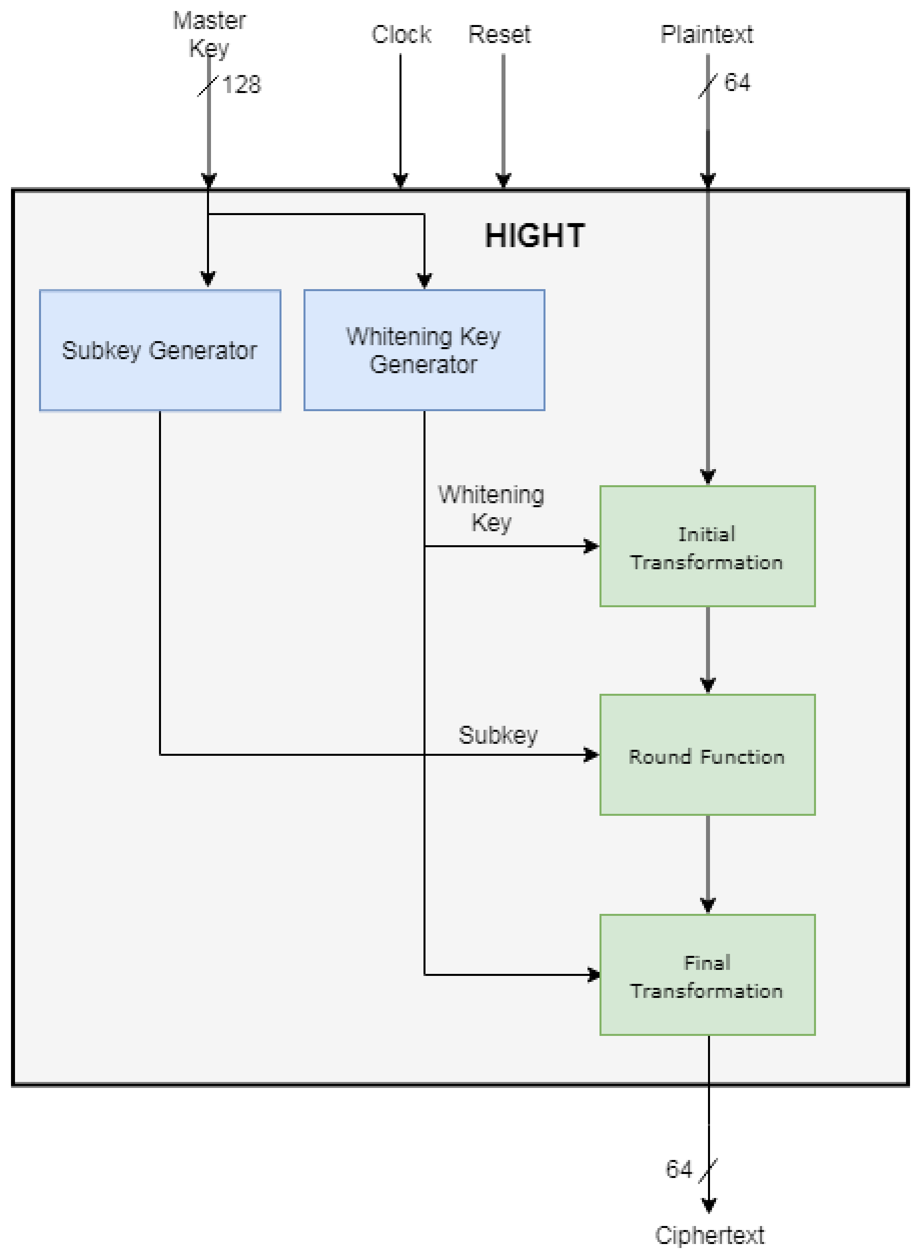


**Figure 3.4:** HIGHT design structure and function interaction

The FSM allows for a sequential execution of the process. This design uses a clock signal for the FSM, so the processing time for the system will be significantly reduced. As a counterpoint, this is a linear design which will use a greater amount of logic elements.

## 3.3 Parallel Architecture

As is clear from figure 3.5 for this architecture, this system will be able to process and encrypt more data than the standard setup for HIGHT. Due to the nature of HIGHT, it has a 64-bit plaintext input and 128-bit master key input. The logic size of the Key Schedule block is larger than that of the individual round blocks seen in the figure 3.5.

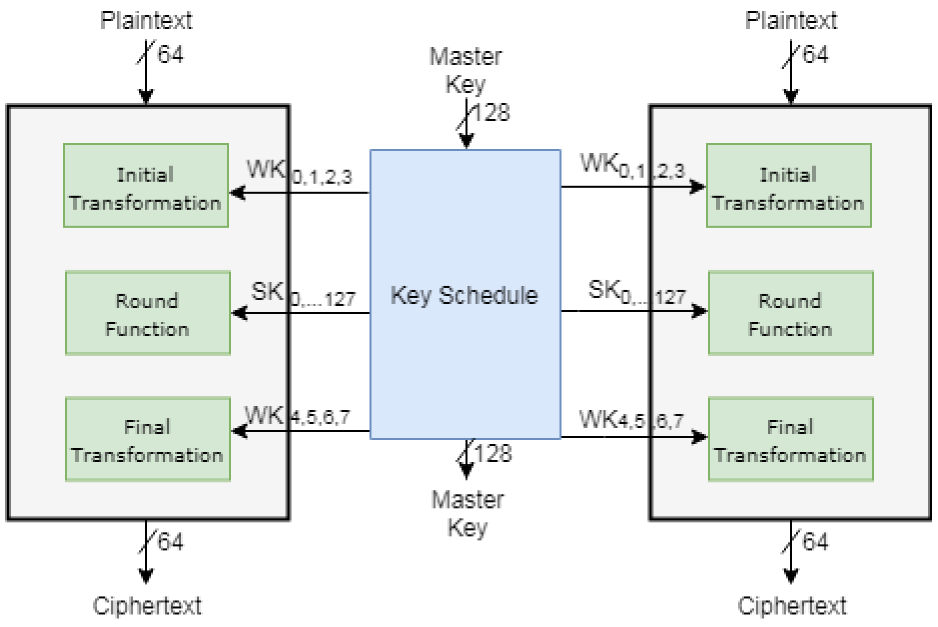


**Figure 3.5:** Parallel Architecture design for HIGHT [18]

This allows us to simultaneously run another encryption block while not breaching the RFID standard circuit area limits of 5000 gates. [19] This limit prevents the processing of more blocks simultaneously, as the cipher should adhere to these limits. This means rather than processing the standard 64-bits for each cycle of the cipher, it will process and encrypt 128-bits. The two parallel round blocks for the encryption of the data will accept 64-bits each to make up 128-bits. Both round blocks use the same key schedule block, so this will save power on running the key schedule twice for each 64-bits of data. An FSM is used for controlling the transfer of the whitening keys and subkeys from the key schedule to the round blocks. [18] This FSM can be built on from the second design, as they can be designed similarly. This design will use the largest amount of power, but it will make up for this sacrifice with its large throughput as it only needs one key schedule for 128-bits to be processed.

## 3.4 Summary

Three different designs for the HIGHT algorithm are discussed in this section, with their advantages and disadvantages. The first design was chosen to be implemented first as it is a relatively simple design and the structure gives a good picture of how the algorithm works. It makes sense to build the second design next as only a few extra changes need to be appended to it to implement the third design. The process of building the different implementations based on these designs is outlined in the next section.

# Chapter 4- Implementation and Testing

This section will address the three different architectures of the HIGHT block cipher which were implemented. The steps taken to write the VHDL description are described in this section. Also outlined in this section are the steps taken to analyse the performance of the individual implementations of the algorithm and creating an IP to package so that the VHDL description can be used on the Zybo Zynq-7000 development board.

## 4.1 Implementation 1

The first implementation of the project has a modularised structure. VHDL easily allows for this by being able to instantiate components which you have created in the same project in different modules.

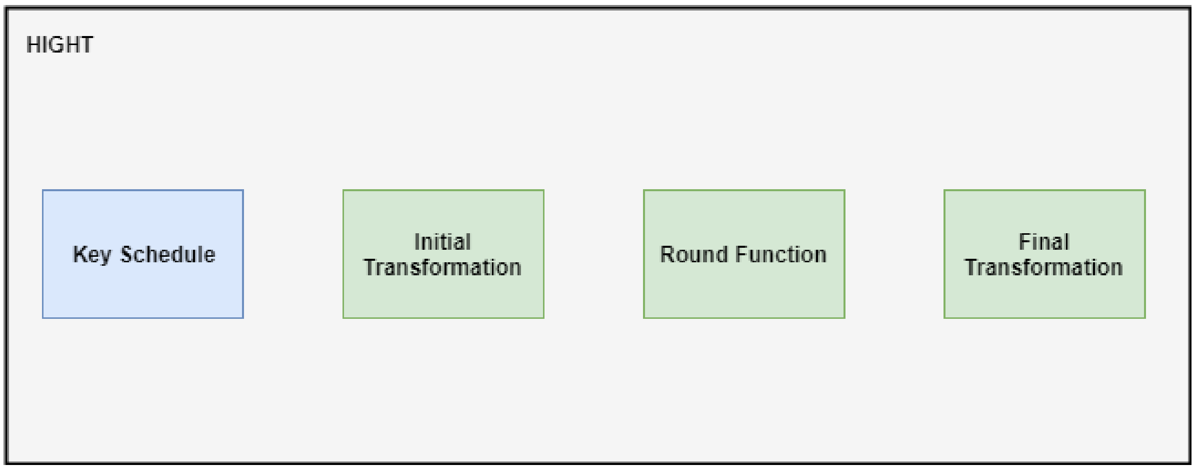


**Figure 4.1:** HIGHT module containing the rest of the modules

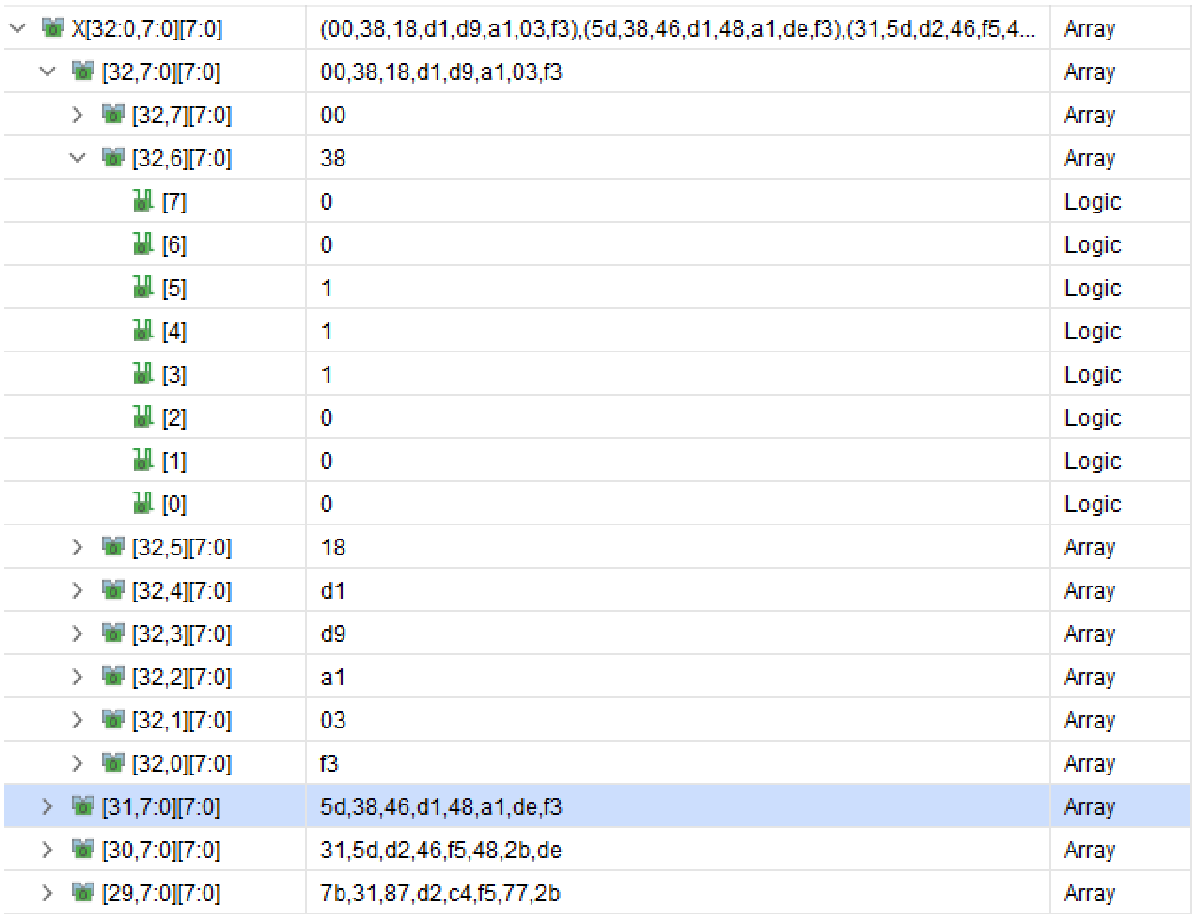
Figure 4.1 shows the project structure for this implementation in VHDL. The overlaying HIGHT module contains four other files within it. The HIGHT file, along with the files it contains are all VHDL files (.vhd). In HIGHT.vhd, there are components instantiated for each of the modules which it contains. The outputs for the Key Schedule are used in each of the Initial Transformation, the Round Function and the Final Transformation. The output of the Initial Transformation feeds into an input of the Round Function, while the output of the Round Function feeds into an input for the Final Transformation. We can connect these outputs and inputs using port maps in the overlaying HIGHT.vhd file by setting a signal to be an output of one of the modules and an input for another module. This allows for an interconnected design between the modules. The components of each of these files are declared and will run as a result of these instantiations. The output of the Final Transformation is also assigned to the output of the overall system in the HIGHT file.

As all the files depend on using keys generated from the key schedule, this file was designed first. This file generates whitening keys, which are used by the Initial and Final Transformation. It also creates subkeys, which are used by the Round Function file. There are three separate VHDL processes in the Key Generation file. The first process creates the whitening keys using ‘for’ loops. The use of the ‘for’ loop makes the code more compact rather than having to write out similar signal assignments one after the other, which can look messy. There are two other processes in the file, the constant generation and the subkey generation. The constants which are generated in its process are used for the generation of the subkeys. As processes in VHDL run concurrently, the order of these processes does not affect the result. When this file was implemented initially, the master key, subkey and whitening keys were all declared as bits in a large std_logic_vector. As a neater solution and an easier method of writing the VHDL, a subtype called ‘byte’ was created. This subtype is an 8-bit std_logic_vector. This allows the keys to be split up, making it easier to call the specific byte rather than attempting to address the correct bit range.

The second file designed was the Initial Transform, which is the first stage of the encryption process. This takes in the plaintext input and uses whitening keys zero to four from the key schedule to encrypt the plaintext. It does this by using the modular addition and XOR operations. The values are updated on the rising edge of a clock signal which is an input to this file.

The Final Transformation file was the next file designed as it was similar to the initial transformation. The only changes which had to be made were the input to the file was not the plaintext but the plaintext after going through thirty-two rounds of encryption. The other change was that whitening keys four to seven were used.

The Round Function file was the most complex aspect of the algorithm to design. Similar to how the Initial Transformation was initially designed, the original design for this used large std_logic_vectors. An std_logic_vector of size 2112 bits was used to hold the results from all thirty-two rounds of encryption. This made any attempt to troubleshoot or read the values this signal contained awkward. This was made a lot clearer once a multidimensional std_logic_vector subtype was created called bidirectional_vector. This creates a three-dimensional array. This is used to create a range of thirty-two, within each of these thirty-two are eight 8-bit bytes. Figure 4.2 visually displays this concept.



| | | |
|---|---|---|
| X[32:0,7:0][7:0] | (00,38,18,d1,d9,a1,03,f3),(5d,38,46,d1,48,a1,de,f3),(31,5d,d2,46,f5,4... | Array |
| [32,7:0][7:0] | 00,38,18,d1,d9,a1,03,f3 | Array |
| [32,7][7:0] | 00 | Array |
| [32,6][7:0] | 38 | Array |
| [7] | 0 | Logic |
| [6] | 0 | Logic |
| [5] | 1 | Logic |
| [4] | 1 | Logic |
| [3] | 1 | Logic |
| [2] | 0 | Logic |
| [1] | 0 | Logic |
| [0] | 0 | Logic |
| [32,5][7:0] | 18 | Array |
| [32,4][7:0] | d1 | Array |
| [32,3][7:0] | d9 | Array |
| [32,2][7:0] | a1 | Array |
| [32,1][7:0] | 03 | Array |
| [32,0][7:0] | f3 | Array |
| [31,7:0][7:0] | 5d,38,46,d1,48,a1,de,f3 | Array |
| [30,7:0][7:0] | 31,5d,d2,46,f5,48,2b,de | Array |
| [29,7:0][7:0] | 7b,31,87,d2,c4,f5,77,2b | Array |

**Figure 4.2:** Bidirectional_vector subtype structure

Using this subtype allowed the code to be a lot cleaner and easier to read. A ‘for’ loop is used to set the values for rounds one to thirty-one of the encryption process. The thirty-second round uses values which are hardcoded, as the process used is slightly different, therefore it cannot be used in the same ‘for’ loop. The modular addition operation is used along with XOR and the functions F0 and F1 to encrypt the text. Functions F0 and F1 contain bitwise rotation and XOR operations. The values for the result of each round of the encryption are also updated on the rising edge of a clock signal which is an input to this file.

The HIGHT file was the final one to be designed as it overlays and contains all the other files created. Declaring components and port maps are used to connect the files correctly for the algorithm.

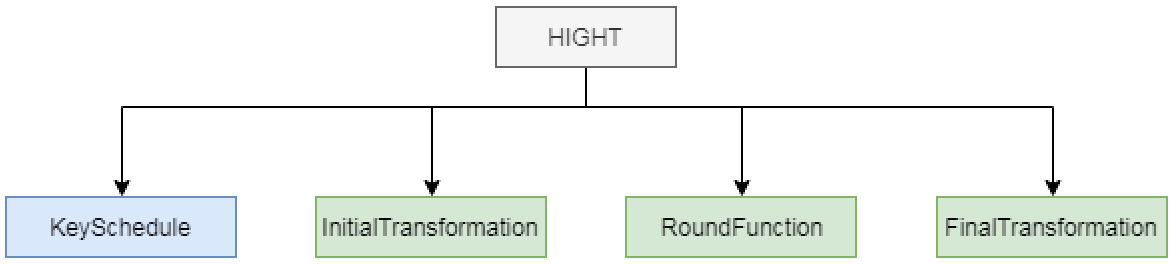


**Figure 4.3:** Hierarchy of files

Not shown in these figures is another file created called ‘constants.vhd’. This is a VHDL file which creates a package. This package contains several things which are reusable such as the message length which is an integer constant. It also contains subtypes which have been defined in here, as they are being used in multiple files and this prevents there being multiple definitions for these subtypes in different files.

## 4.2 Implementation 2

The second implementation of the project has a modularised type structure, similar to the first but it is not created in the same way. There are only two files in this project which are used for the system, which are ‘HIGHT.vhd’ and ‘functions.vhd’. The HIGHT module contains the logic for the FSM. It is using the ‘functions.vhd’ file in it. The functions for the different parts of the encryption process are called within the different states of the FSM.

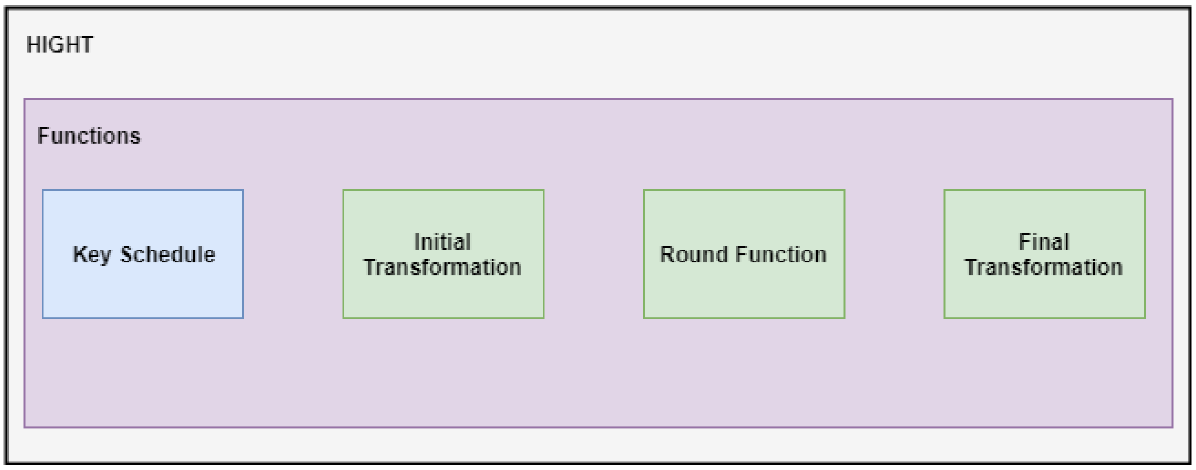


**Figure 4.4:** HIGHT module containing the functions file, which contains all the functions

The logic used in these functions are the same as in the first implementation, the main difference being they are using variables rather than signals. The outputs of these functions are assigned to signals which are created in the HIGHT module. The functions return the value which will be used in the next stage of the encryption process. The system is initialised into the key schedule state when the reset input is high ('1'). A case statement is used for when the system has not been reset, to transition to the next state of the system on the rising edge of the clock signal. When in the key schedule state, the whitening key function is called and assigned to a signal and the subkey function is called and returned to a signal. The state of the machine also updates to initial and the plaintext input is assigned to a signal. The system then enters the initial case state. This just calls the initial transform and returns its value to a signal. The round state returns the round function to a signal similarly. The final state returns the final transformation result, which is the ciphertext and the state changes to output. The output state simply assigns the ciphertext to the output of the HIGHT system.

## 4.3 Implementation 3

The third implementation of this project was building on the second implementation. The structure of this architecture was very similar, with just a few key changes made to allow for the parallel processing. A FSM dictated the sequence of the algorithm, similarly to the second implementation.

One of the key differences between this implementation and the second implementation was the input and output of the system. This architecture of HIGHT accepts a 128-bit block of plaintext, which will then be split into two 64-bit blocks. Likewise, the ciphertext returned by the system is a 128-bit block. In initial state, the initial transformation function is called twice, to pass the two different 64-bit plaintext blocks. These individual results are returned to different signals. In similar fashion, the round block and the final transformation are called twice, with the results from the stages of the encryption of the different blocks being passed to them. The outputs of the two parallel final transformations are concatenated together to result in the 128-bit ciphertext output.

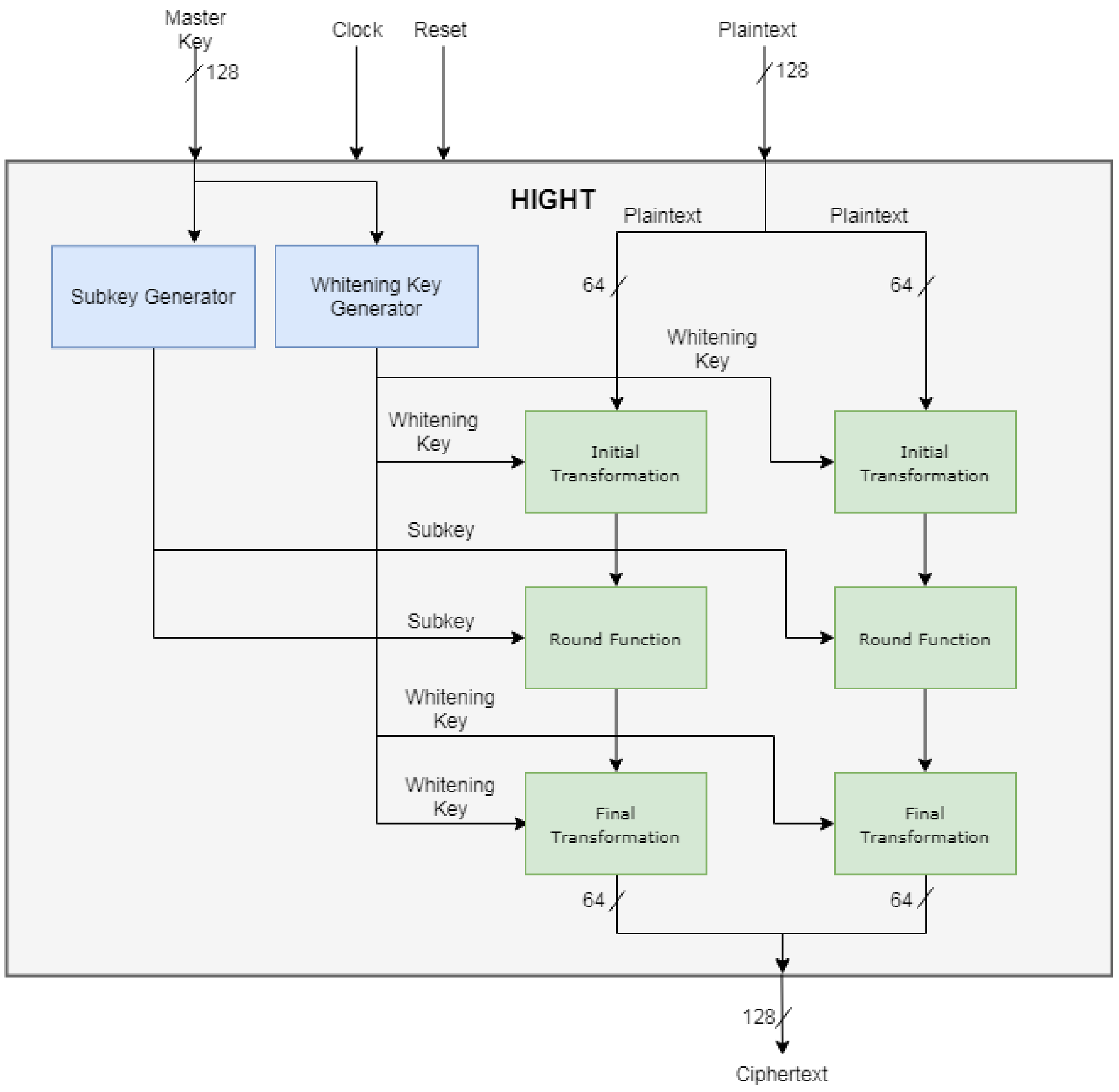


**Figure 4.5:** Flow of data through implementation 3

As is described in this section, with a few simple changes added to the second implementation, a parallel architecture can be built to process twice the data.

## 4.4 Implementing on Hardware

The implementation of the hardware was built similarly for the three implementations. Once the system had been fully described in VHDL and it had been tested to ensure it was working as intended, the project was packaged so that it could run on the FPGA. Once the IP is created, it generates VHDL code which can be appended in some sections to add your own logic to it. A component for the HIGHT module is created and instantiated. As the slave registers created for the IP are thirty-two bits, two slave registers must be used for the plaintext input in the first two implementations. For the third implementation, four slave registers are used to store the plaintext input. The 128-bit master key input requires four slave registers. Separate signals are created to store the inputs. The registers needed to store these inputs are concatenated into their respective signals. This is done so that the registers can be mapped to inputs of the HIGHT system. Two of the registers are set to write the value in them into another signal which was created for the first two implementations. The third implementation uses four of these registers as the output of the system is 128-bit. This signal will be mapped to the output ciphertext of the HIGHT system. Creating the IP can be used to connect the design which has been created in VHDL to the Zynq processor. This can be done by creating a block design to connect your IP to the Zynq processor. For this project, the UART1 peripheral will be used to connect to the device.

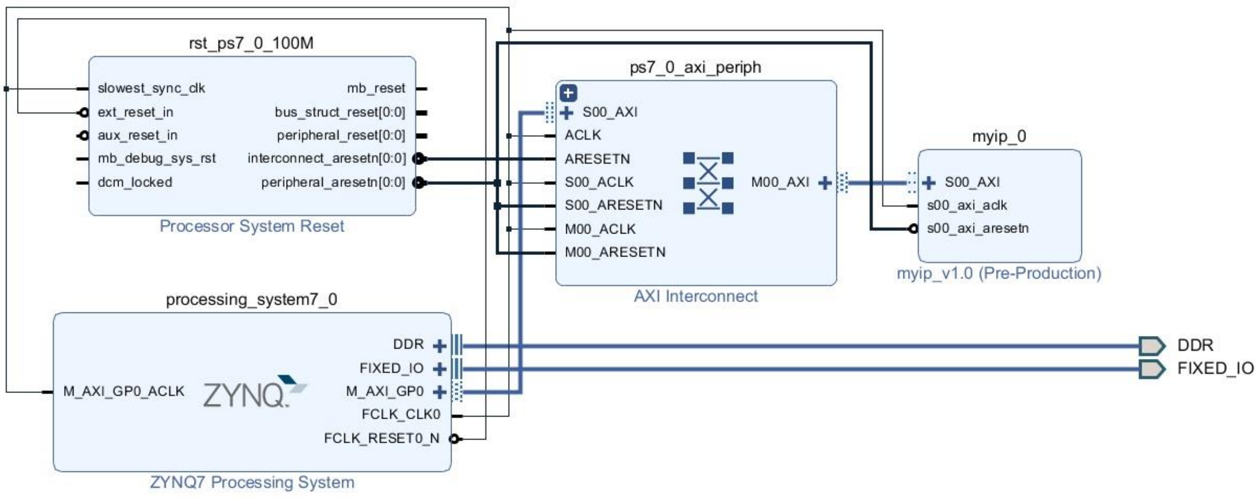


**Figure 4.6:** Block diagram connecting IP with Zynq

The AXI interconnect is used for memory mapped transfers between the processor and the IP that has been created. Vivado can then generate the HDL for the block design which you have designed. At this point, the hardware is exported, and an SDK is launched. The SDK is used to set the values for the inputs to the system. It can also be used to read and display the output of the system to ensure it works correctly. The FPGA should be connected to the device running the SDK so that the bitstream can be downloaded to the FPGA. The program is then launched on the hardware.

## 4.5 Testing

### 4.5.1 Testing of Implementation 1

To test that the cipher was working correctly, the developers of the cipher released two test vectors which can be used as an input to your system, to generate a given output. Along with this, the developers also released detailed results that you should achieve along the way to ensure it is encrypting the system correctly and not just reaching the correct result by chance. They have given the subkeys that you should get from the master key input. They have also given the step by step encryption, showing the result of the cipher after each round of encryption. This means the results achieved could be compared to this for the Initial Transformation, each round in the Round Function and the Final Transformation. The following tables show the vectors given, along with their result.

**Table 4: Test Vector 1 Inputs and Output**

| | |
|---|---|
| **Key** | 00 11 22 33 44 55 66 77 88 99 aa bb cc dd ee ff |
| **Plaintext** | 00 00 00 00 00 00 00 00 |
| **Ciphertext** | 00 f4 18 ae d9 4f 03 f2 |

**Table 5: Test Vector 1 Subkeys**

| **Sub Key** | **Value** | **Sub Key** | **Value** |
|---|---|---|---|
| *SK3* \|\| *SK2* \|\| *SK1* \|\| *SK0* | e7135b59 | *SK67* \|\| *SK66* \|\| *SK65* \|\| *SK64* | cfa7c7f6 |
| *SK7* \|\| *SK6* \|\| *SK5* \|\| *SK4* | c99cb0c8 | *SK71* \|\| *SK70* \|\| *SK69* \|\| *SK68* | 48555f62 |
| *SK11* \|\| *SK10* \|\| *SK9* \|\| *SK8* | 906d96d7 | *SK75* \|\| *SK74* \|\| *SK73* \|\| *SK72* | 1f50a1b1 |
| *SK15* \|\| *SK14* \|\| *SK13* \|\| *SK12* | 2c6a5599 | *SK79* \|\| *SK78* \|\| *SK77* \|\| *SK76* | a5986d86 |
| *SK19* \|\| *SK18* \|\| *SK17* \|\| *SK16* | 27032ade | *SK83* \|\| *SK82* \|\| *SK81* \|\| *SK80* | 0706f33c |
| *SK23* \|\| *SK22* \|\| *SK21* \|\| *SK20* | b5e32d31 | *SK87* \|\| *SK86* \|\| *SK85* \|\| *SK84* | fb2b7aff |
| *SK27* \|\| *SK26* \|\| *SK25* \|\| *SK24* | ced9de4e | *SK91* \|\| *SK90* \|\| *SK89* \|\| *SK88* | 7a755a93 |
| *SK31* \|\| *SK30* \|\| *SK29* \|\| *SK28* | 48919180 | *SK95* \|\| *SK94* \|\| *SK93* \|\| *SK92* | 7bb39134 |
| *SK35* \|\| *SK34* \|\| *SK33* \|\| *SK32* | f915b5f4 | *SK99* \|\| *SK98* \|\| *SK97* \|\| *SK96* | bcdf15f0 |
| *SK39* \|\| *SK38* \|\| *SK37* \|\| *SK36* | fadc0ee2 | *SK103* \|\| *SK102* \|\| *SK101* \|\| *SK100* | ef018ca2 |
| *SK43* \|\| *SK42* \|\| *SK41* \|\| *SK40* | bba15439 | *SK107* \|\| *SK106* \|\| *SK105* \|\| *SK104* | 2a436495 |
| *SK47* \|\| *SK46* \|\| *SK45* \|\| *SK44* | 9fadb9bf | *SK111* \|\| *SK110* \|\| *SK109* \|\| *SK108* | ae882255 |
| *SK51* \|\| *SK50* \|\| *SK49* \|\| *SK48* | 16b7f8e8 | *SK115* \|\| *SK114* \|\| *SK113* \|\| *SK112* | c7e50f52 |
| *SK55* \|\| *SK54* \|\| *SK53* \|\| *SK52* | e41e0239 | *SK119* \|\| *SK118* \|\| *SK117* \|\| *SK116* | 67d9bcf0 |
| *SK59* \|\| *SK58* \|\| *SK57* \|\| *SK56* | d9451b36 | *SK123* \|\| *SK122* \|\| *SK121* \|\| *SK120* | 61a18fda |
| *SK63* \|\| *SK62* \|\| *SK61* \|\| *SK60* | a9b0ad97 | *SK127* \|\| *SK126* \|\| *SK125* \|\| *SK124* | d1357c79 |

Table 6: Test vector 1 Round Results

| **Round** | **Value** | **Round** | **Value** |
|---|---|---|---|
| *Initial* | 0000001100220033 | *Round 17* | 2c93a90ddd0283ae |
| *Round 1* | 00ce1138223f33e7 | *Round 18* | 93570db102d9aec4 |
| *Round 2* | cee138ef3fa3e78a | *Round 19* | 57b7b1dbd998c4e4 |
| *Round 3* | e14fef91a3708a8a | *Round 20* | b7bedb55989ae458 |
| *Round 4* | 4f8a91cd70518ad1 | *Round 21* | be87559d9a515868 |
| *Round 5* | 8a53cd0951c3d1ee | *Round 22* | 87ce9d5351786873 |
| *Round 6* | 534609c7c3e4ee7d | *Round 23* | ceab53d6784b73bc |
| *Round 7* | 4673c7c5e41b7dd7 | *Round 24* | ab30d6d74ba8bc69 |
| *Round 8* | 7359c58c1b33d79c | *Round 25* | 30bfd7f7a83369df |
| *Round 9* | 595f8cf333d59c07 | *Round 26* | bf13f71733bfdf7d |
| *Round 10* | 5f0cf317d507073f | *Round 27* | 134617f1bfd57db2 |
| *Round 11* | 0ca0173007033fb6 | *Round 28* | 467bf187d5c4b277 |
| *Round 12* | a03a3043030bb63e | *Round 29* | 7b3187d2c4f5772b |
| *Round 13* | 3a7943b40b2b3e37 | *Round 30* | 315dd246f5482bde |
| *Round 14* | 7920b47a2b7c37b5 | *Round 31* | 5d3846d148a1def3 |
| *Round 15* | 20637a797ce4b5d0 | *Round 32* | 003818d1d9a103f3 |
| *Round 16* | 632c79a9e4ddd083 | *Final* | 00f418aed94f03f2 |

Table 7: Test Vector 2 Inputs and Output

| | |
|---|---|
| **Key** | ff ee dd cc bb aa 99 88 77 66 55 44 33 22 11 00 |
| **Plaintext** | 00 11 22 33 44 55 66 77 |
| **Ciphertext** | 23 ce 9f 72 e5 43 e6 d8 |

**Table 8: Test Vector 2 Subkeys**

| **Sub Key** | **Value** | **Sub Key** | **Value** |
|---|---|---|---|
| *SK3* \|\| *SK2* \|\| *SK1* \|\| *SK0* | 4e587e5a | *SK67* \|\| *SK66* \|\| *SK65* \|\| *SK64* | be74727f |
| *SK7* \|\| *SK6* \|\| *SK5* \|\| *SK4* | b8695b51 | *SK71* \|\| *SK70* \|\| *SK69* \|\| *SK68* | af9a8263 |
| *SK11* \|\| *SK10* \|\| *SK9* \|\| *SK8* | 07c2c9e8 | *SK75* \|\| *SK74* \|\| *SK73* \|\| *SK72* | 1e2d5c4a |
| *SK15* \|\| *SK14* \|\| *SK13* \|\| *SK12* | 2b471032 | *SK79* \|\| *SK78* \|\| *SK77* \|\| *SK76* | 1ceda097 |
| *SK19* \|\| *SK18* \|\| *SK17* \|\| *SK16* | 6c262bcd | *SK83* \|\| *SK82* \|\| *SK81* \|\| *SK80* | d4b17ca3 |
| *SK23* \|\| *SK22* \|\| *SK21* \|\| *SK20* | 828eb698 | *SK87* \|\| *SK86* \|\| *SK85* \|\| *SK84* | 404e7bee |
| *SK27* \|\| *SK26* \|\| *SK25* \|\| *SK24* | 230cef4d | *SK91* \|\| *SK90* \|\| *SK89* \|\| *SK88* | 5730f30a |
| *SK31* \|\| *SK30* \|\| *SK29* \|\| *SK28* | 254c2af7 | *SK95* \|\| *SK94* \|\| *SK93* \|\| *SK92* | d0e6a233 |
| *SK35* \|\| *SK34* \|\| *SK33* \|\| *SK32* | 1c16a4c1 | *SK99* \|\| *SK98* \|\| *SK97* \|\| *SK96* | 67687c35 |
| *SK39* \|\| *SK38* \|\| *SK37* \|\| *SK36* | a5657527 | *SK103* \|\| *SK102* \|\| *SK101* \|\| *SK100* | 12027b6f |
| *SK43* \|\| *SK42* \|\| *SK41* \|\| *SK40* | eeb25316 | *SK107* \|\| *SK106* \|\| *SK105* \|\| *SK104* | e5dcdbea |
| *SK47* \|\| *SK46* \|\| *SK45* \|\| *SK44* | 5a463014 | *SK111* \|\| *SK110* \|\| *SK109* \|\| *SK108* | e1992132 |
| *SK51* \|\| *SK50* \|\| *SK49* \|\| *SK48* | 17a6c593 | *SK115* \|\| *SK114* \|\| *SK113* \|\| *SK112* | 504c5475 |
| *SK55* \|\| *SK54* \|\| *SK53* \|\| *SK52* | 6d85475c | *SK119* \|\| *SK118* \|\| *SK117* \|\| *SK116* | 68c8899b |
| *SK59* \|\| *SK58* \|\| *SK57* \|\| *SK56* | ea44f8f1 | *SK123* \|\| *SK122* \|\| *SK121* \|\| *SK120* | fa18e40d |
| *SK63* \|\| *SK62* \|\| *SK61* \|\| *SK60* | 422702ca | *SK127* \|\| *SK126* \|\| *SK125* \|\| *SK124* | e2345934 |

**Table 9: Test Vector 2 Round Results**

| **Round** | **Value** | **Round** | **Value** |
|---|---|---|---|
| *Initial* | 00ee222144886643 | *Round 17* | db63ca6b6e9dfaaf |
| *Round 1* | ee2d21b1880a435f | *Round 18* | 63776b6b9d09af72 |
| *Round 2* | 2db4b11c0acc5fde | *Round 19* | 77856b93091172c5 |
| *Round 3* | b4951c9fcca3dec5 | *Round 20* | 851793871106c58c |
| *Round 4* | 95c19fe4a30fc556 | *Round 21* | 17a7878206f18c48 |
| *Round 5* | c115e4730f545645 | *Round 22* | a7598251f1c64855 |
| *Round 6* | 15e27386540d45b7 | *Round 23* | 597d5119c6e85575 |
| *Round 7* | e26486c30dabb777 | *Round 24* | 7d4a196ee8e775d8 |
| *Round 8* | 6424c35bab9d7772 | *Round 25* | 4a7f6ef7e7bdd882 |
| *Round 9* | 24725b8c9d607282 | *Round 26* | 7fadf729bdcb8284 |
| *Round 10* | 72458c7b602d829d | *Round 27* | ad442985cb29845f |
| *Round 11* | 458c7bab2dc69d59 | *Round 28* | 44b58548296e5f31 |
| *Round 12* | 8cc6ab08c6ba5982 | *Round 29* | b51d488f6e0231f3 |
| *Round 13* | c60f0841ba688280 | *Round 30* | 1df78ff802f8f39d |
| *Round 14* | 0fd3413668f280d4 | *Round 31* | f7fdf850f8529dd8 |
| *Round 15* | d35c3627f2afd4e4 | *Round 32* | 23fd9f50e552e6d8 |
| *Round 16* | 5cdb27caaf6ee4fa | *Final* | 23ce9f72e543e6d8 |

To test whether the cipher is working correctly, along with these test vectors, you can create a test bench in VHDL. A test bench instantiates a component of the module which is being tested. This allows you to be able to set the values for the inputs of the module which is being tested in the test bench. Test benches were created for each individual file, to ensure that they were working correctly. The test bench for the HIGHT module could test the input and output for the whole system, ensuring that the modules are connected to each other correctly and the correct result is achieved. When the incorrect output was resulting from the Round Function module, pass/fail tests were put into the test bench to check which line the error was occurring at. This can help narrow down specifically what the problem is. The operations in the lines which are incorrect can be calculated on paper, to see the result which should be achieved and compare to the result which is achieved. This helps narrow down specifically what the problem is. During testing of the Round Function, this process was used to eventually conclude that the functions F0 and F1 were incorrect. The reason these functions were not working as intended was that the shift left operation was being used instead of the rotate left operation. The objects tab in Vivado is also useful for troubleshooting. This tab, along with the scope tab allows you to check the results for the different ports and signals in each individual module. The individual bits for these ports and signals can be seen, so this can help narrow down specifically where a problem is occurring. Another tool which was used for the debugging of problems in the project was the breakpoint/step debugging tool. This can be used to see, step by step, what is happening in the program that has been written.

This process was used to fix any problems that occurred in the building of this design. Once the system achieved the correct result, it is important to compare each round of encryption to the results given. If all the results are the same as the results given, it can be concluded that the HIGHT cipher has been designed correctly.

### 4.4.2 Testing of Implementation 2 & 3

There is only one entity which can be tested in these structures of the project. This means a test bench can only be created for the HIGHT top module. This is another reason why it makes sense to build these versions of the project after building the previous version. From encrypting the project in a similar way, it is known that the logic behind the encryption process is correct. The results can be compared to the ones given from the test vectors. These can be seen in tables 4, 5, 6, 7, 8 and 9. This confirms that this implementation of the project is working correctly.

## 4.5 Summary

To summarise, this chapter outlines the steps taken to implement three different designs of the project. These designs were described in VHDL, before they were implemented on an FPGA device. The three implementations were tested, with reference to the test vectors provided by the cipher creators, to ensure the system was working as intended. After confirming the cipher was working correctly, results could be derived from the individual implementations and compared.

# Chapter 5 - Results and Discussion

The metrics used to evaluate the HIGHT cipher design were energy, area and performance. The performance is measured by the throughput and the latency. The energy required to process a plaintext block is given. This is found using the total power consumed, which is measured using Vivado's power report tool. These measurements are tested for the different architectures of the HIGHT system that were implemented. The area usage is provided in terms of Look-Up Tables (LUT) and flip-flops (FF). In the tables in this section, for each metric, the implementation result which performed the best in the area has been displayed in bold. A comparison between Implementation 1 (I1), Implementation 2 (I2) and Implementation 3 (I3) is done in this section.

The HIGHT cipher was designed for devices such as RFIDs, for this reason, a standard frequency for RFID of 13.56 MHz which is used for appropriate applications was chosen as the clock rate. [20] Using this common frequency, a fair comparison can be made between the different architectures of the cipher. The power measured for the different architectures is presented below, with a breakdown of where the power is being used.

**Table 10: Power usage comparison**

| | **Total Power (W)** | **Dynamic Power (W)** | **Clocks Power (W)** | **Signals Power (W)** | **Logic Power (W)** | **I/O Power (W)** | **Device Static Power (W)** |
|---|---|---|---|---|---|---|---|
| I1 | 0.11 | 0.019 | 0.003 | **0.004** | **0.003** | 0.008 | **0.092** |
| I2 | **0.104** | **0.012** | **0.001** | **0.004** | 0.006 | **0.001** | **0.092** |
| I3 | 0.114 | 0.022 | 0.002 | 0.009 | 0.011 | **0.001** | **0.092** |

It should be noted that the static power remains constant across the different architectures built for the same FPGA board. The dynamic power depends on the switching activity of the circuit. The sum of the static and dynamic power results in the total power. [21] I2 has a 5.5% reduction in the total power consumed compared to I1, while I3 uses the most amount of power. I3 processes more bits using two round blocks, which leads to an increase in power.

The area utilisation for the architectures has been compiled, with details such as look-up-tables, flip-flops and inputs/outputs (I/O) being included for a more detailed comparison of where the area is being utilised.

**Table 11: Area utilisation comparison**

| | **LUT** | **FF** | **I/O** |
|---|---|---|---|
| I1 | **1953** | 2208 | **257** |
| I2 | 2578 | **1069** | 258 |
| I3 | 4629 | 1325 | 386 |

I1 has a significant amount more LUTs used than I2 and I3 uses the most LUTs. There is a 24.2% reduction in LUTs in I1 when compared to I2. I3 uses almost twice the amount of LUTs as I2. This is due to I1s modularised structure containing simple reusable logic in comparison to I2 and I3s FSM using functions with linear logic. This leads to a greater hardware usage. There is a greater number of flip-flops in I1 than I2 and I3. This is due to more registers being used for memory, as a result of having ports and signals in each module.

The performance of the encryption process of the cipher designs is analysed by the throughput and the latency. The latency is calculated by the number of cycles required to process a 64-bit block in the HIGHT system. These have been calculated using the following formulas and compiled into the table 12.

$$Throughput = \frac{13.56MHz \times blockSize}{latency}$$

**Table 12: Performance comparison**

| | **Block Size (bits)** | **Latency (clock cycles)** | **Throughput (Mbps)** |
|---|---|---|---|
| I1 | 64 | 35 | 24.8 |
| I2 | 64 | **4** | 216.96 |
| I3 | 128 | **4** | **433.92** |

I2 yields far superior results than I1 for the throughput and latency. As I3 is built the same way, except processes twice the bits, the throughput is twice that of I2. There is an 88.6% reduction in throughput in I1 when compared to I2. This is due to the clock in the FSM of I2 running to call each function rather than running inside the encryption round process.

**Table 13: Resource usage and performance comparison**

| **Design** | **Block Size (bits)** | **Key Size (bits)** | **LUT** | **FF** | **I/O** | **Clock Rate (MHz)** | **Latency (clock cycles)** | **Throughput (Mbps)** |
|---|---|---|---|---|---|---|---|---|
| I1 | 64 | 128 | **1953** | 2208 | **257** | 13.56 | 35 | 24.8 |
| I2 | 64 | 128 | 2578 | **1069** | 258 | 13.56 | **4** | 216.96 |
| I3 | 128 | 128 | 4629 | 1325 | 386 | 13.56 | **4** | **433.92** |

I1, although it uses more FFs, it generally outperformed in terms of hardware usage. I2 and I3 yielded much greater results in terms of throughput, with a smaller latency.

The energy required to process a 64-bit block is found by using the following formula.

$$Energy = \frac{power \times latency}{13.56MHz}$$

The energy per bit can then be found using the following formula.

$$Energy\ per\ bit = \frac{Energy}{BlockSize}$$

Table 14 has compiled the key information regarding the power and energy consumption to compare the two ciphers side by side.

**Table 14: Power and energy consumption comparison**

| **Design** | **Block Size (bits)** | **Key Size (bits)** | **Latency (clock cycles)** | **Static Power (W)** | **Dynamic Power (W)** | **Total Power** | **Energy (µJ)** | **Energy/bit (nJ/b)** |
|---|---|---|---|---|---|---|---|---|
| I1 | 64 | 128 | 35 | **0.092** | 0.019 | 0.11 | 0.284 | 4.44 |
| I2 | 64 | 128 | **4** | **0.092** | **0.012** | **0.104** | **0.032** | 0.507 |
| I3 | 128 | 128 | **4** | **0.092** | 0.022 | 0.114 | 0.034 | **0.266** |

The energy per bit measurement is a measure of the efficiency of the architectures of the cipher, measuring the amount of energy used for each bit processed in the cipher design. I2 has a 5.5% reduction in the total power consumed compared to I1. This in turn contributed to a reduction in the energy consumed of 88.7%. While the FSM in I2 means the encryption processes are running sequentially rather than concurrently, the power is significantly reduced due to the clock signal being used for the FSM rather than updating each round, like in I1. I3 has a slightly larger energy consumption result when compared to I2, but when comparing the efficiency of that energy, I3 had the most successful results with a 47.5% reduction in energy per bit when compared to I2.

## 5.1 Summary

This section compared and contrasted the implementation results of three different architectures. I3 proved to have the best results for the use of energy per bit processed in the system while I1 used the least amount of hardware logic. A decision can be made as to which implementation would be appropriate to use for the needs of the device.

# Chapter 6 – Ethics

The topic of ethics is an important one for the area of cryptography. It can come into play when making the decision of if a device should require cryptography to keep the data secure. In certain areas such as the military, healthcare and finance, you could argue that using cryptography in certain situations is the ethical decision to make, even if it will be more expensive. Information that is used and transferred in these sectors can be very important and could affect many people's lives significantly if the data is breached. It is important to be able to implement strong cryptography so that this information can remain private. A strong cipher can lead to a range of things, such as increased power consumption, increased encryption/decryption or key generation time, an increase in area, etc. It is important to note that there could be sources with malicious intent who are attempting to interfere with the transfer of data in these sectors. Encryption uses a lot of resources such as power, time and hardware. If we could rely on an ethical world, there would be no need for encryption and this would increase the amount of resources that could be dedicated to communication of data rather than using resources on encrypting the data.

Another case where ethics comes into play with cryptography is when a company is entrusted with personal information of a customer/user. For example, on many websites, the passwords of users will be stored but it is also important that the company and people working at the company cannot access this information. In this kind of scenario, the encryption of the data is important as the user/customer should be able to trust the company to be able to keep their information private. [22] A recent story has come out about Facebook storing millions of users' passwords as plaintext. An investigation stated that between 200 and 600 million users passwords were stored as plaintext, which was accessible by more than 20,000 employees. [23] This is an example of the importance of encrypting data as, particularly at as large of a company as Facebook, there are many people who may be able to access this information.

While generally, you can say that it is ethical to use cryptography, there are exceptions. This depends on the usage of the cryptography, for example, if a criminal was using cryptography to protect themselves, cryptography can be seen to be used to the advantage of someone who is using it for unethical activities. [22]

# Chapter 7 - Conclusions and Further Research

In conclusion, this report demonstrates a successful implementation of the lightweight cipher HIGHT. The aim of this project was to design this cipher using VHDL, targeting a Xilinx FPGA board and analyse its performance and resource requirements. This was successfully achieved for three different implementations of the cipher. Experimental results were obtained for each of the three architectures, using parameters that made it a fair and equal comparison. A 64-bit modular architecture was presented, which could be applied to devices which can accept a trade-off in performance for less hardware usage. Another 64-bit architecture with a FSM structure was developed, which acts as an in between as it did not result in the greatest performance or resource utilisation, but achieved results between the other architectures. The final architecture presented was a 128-bit architecture, structured as an FSM also. This is designed for application where low resource utilisation is not a priority, but rather a high performance, while still upholding HIGHTs lightweight nature.

Future work could be done to optimise the VHDL code to reduce the amount of logic elements used. One way to do this would be to optimise the FSM. While the first architecture implemented had less logic elements used, this could be further improved upon. Additional subtypes could be created to further reduce the hardware.

Comparisons between various different FPGAs could be made to have a more wide and informative analysis of the cipher. The cipher could also be implemented on a microprocessor or ASIC.

There could also be further work done into analysing side-channel attacks on the cipher and designing suitable counter-measures for these attacks. A hardware Trojan attack analysis was performed on HIGHT by [24]. This is a circuit which can be inserted as firmware modifications to FPGA bit streams to deactivate the host circuit, change its functionality or provide covert channels through which information can be leaked. This can be used to recover the master key of the cipher. [24]

# References

[1] A. M. Wood, “Get Started with VHDL Programming: Design Your Own Hardware,” 12 February 2019. [Online]. Available: https://www.whoishostingthis.com/resources/vhdl/. [Accessed 13 March 2019].

[2] University of Southampton, “ELEC3017 - Simulation,” University of Southampton, Southampton.

[3] X. Wang, *VHDL Introduction,* Dublin, 2018.

[4] Xilinx, “Vivado Design Suite,” Xilinx, [Online]. Available: https://www.xilinx.com/products/design-tools/vivado.html. [Accessed 13 03 2019].

[5] Mentor, “ModelSim PE Student Edition,” Mentor, [Online]. Available: https://www.mentor.com/company/higher_ed/modelsim-student-edition. [Accessed 13 03 2019].

[6] Xilinx, “Digilent Zybo Zynq-7000 ARM/FPGA SoC Trainer Board,” Xilinx, [Online]. Available: https://www.xilinx.com/products/boards-and-kits/1-4azfte.html. [Accessed 14 03 2019].

[7] Xilinx, “What is an FPGA?,” Xilinx, [Online]. Available: https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html. [Accessed 06 April 2019].

[8] M. Rouse, “What is cryptography?,” September 2018. [Online]. Available: https://searchsecurity.techtarget.com/definition/cryptography. [Accessed 01 April 2019].

[9] M. Rouse, “What is cryptanalysis?,” January 2018. [Online]. Available: https://searchsecurity.techtarget.com/definition/cryptanalysis. [Accessed 01 April 2019].

[10] “What is cryptology?,” September 2005. [Online]. Available: https://searchsecurity.techtarget.com/definition/cryptology. [Accessed 01 April 2019].

[11] C. A. Lara-Nino, A. Diaz-Perez and M. Morales-Sandoval, “Lightweight Hardware Architectures for the Present Cipher in FPGA,” *IEEE Transactions on Circuits and Systems I: Regular Papers,* vol. 64, no. 9, pp. 2544 - 2555, 2017.

[12] P. Yalla and K. J, “Lightweight Cryptography for FPGAs,” in *2009 International Conference on Reconfigurable Computing and FPGAs*, Quintana Roo, 2009.

[13] M. Feldhofer, S. Dominikus and J. Wolkerstorfer, "Strong Authentication for RFID Systems Using the AES Algorithm," *CHES'04,* pp. 357-370, 2004.

[14] D. Hong, J. Sung, S. Hong and e. al, "HIGHT: A new block cipher suitable for low-resource device," in *International Workshop on Cryptographic Hardware and Embedded Systems*, Yokohama, 2006.

[15] B. Schneier, in *Applied Cryptography Second Edition*, New York, Wiley, 1996, pp. 366-367.

[16] T. Eisenbarth, S. Kumar, C. Paar, A. Poschmann and L. Uhsadel, "A Survey of Lightweight-Cryptography Implementations," *IEEE Design & Test of Computers,* vol. 24, no. 6, pp. 522-533, 2007.

[17] H. Lee, H. Jeong, B. Han and Y. Won, "The HIGHT Encryption Algorithm," 24 June 2011. [Online]. Available: https://tools.ietf.org/html/draft-kisa-hight-00. [Accessed 03 April 2019].

[18] J.-H. Lee and D.-G. Lim, "Parallel Architecture for High-Speed Block Cipher, HIGHT," *International Journal of Security and Its Applications,* vol. 8, no. 2, pp. 59-66, 2014.

[19] K. Finkenzeller and M. D., RFID-Handbook: Fundamentals and Applications in Contactless SmartCards, Radio Frequency Identification and Near-Field Communication, 3rd ed., Wiley, 2010.

[20] "RFID Frequency Bands & Spectrum | Electronics Notes," Electronics Notes, [Online]. Available: https://www.electronics-notes.com/articles/connectivity/rfid-radio-frequency-identification/frequency-bands-spectrum.php. [Accessed 05 April 2019].

[21] C. A. Lara-Nino, A. Diaz-Perez and M. Morales-Sandoval, "Lightweight Hardware Architectures for the Present Cipher in FPGA," *IEEE Transactions on Circuits and Systems I: Regular Papers,* vol. 64, no. 9, pp. 2544-2555, September 2017.

[22] C. Falk, "The Ethics of Cryptography," West Lafayette, 2005.

[23] B. Krebs, "Facebook Stored Hundreds of Millions of User Passwords in Plain Text for Years," KrebsonSecurity, 21 March 2019. [Online]. Available: https://krebsonsecurity.com/2019/03/facebook-stored-hundreds-of-millions-of-user-passwords-in-plain-text-for-years/. [Accessed 07 April 2019].

[24] H. Chen, T. Z. F. Wang and e. al., "Stealthy Hardware Trojan Based Algebraic Fault Analysis of HIGHT Block Cipher," *Security and Communication Networks,* vol. 2017, p. 15, 2017.

# Appendix 1

The code for this project can be found at the following Google Drive link:

https://drive.google.com/open?id=1CMr3UowMg6yFD_War02wL7l8wcACQsX8