



**DUBLIN CITY UNIVERSITY  
SCHOOL OF ELECTRONIC ENGINEERING**

**Efficient FPGA Implementation of the Lightweight  
Cipher LED**

**Conor Sherlock**

April 2019

**BACHELOR OF ENGINEERING**

IN

**Electronic and Computer Engineering**

Supervised by Dr. X. Wang

Project Code XW19

# Acknowledgements

I would like to thank my supervisor Dr. Xiaojun Wang for his Guidance throughout the year and for giving me an excellent understanding of the VHDL language, and also for providing me with the equipment and hardware necessary for the project to be completed.

I would also like to thank Billy Roarty for providing me with a computer and a lab to work on the project and the report.

Finally, I would like to thank Brandon Walsh for his help throughout the year, and for generously offering his help with the project whenever it was needed.

# 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](https://www4.dcu.ie/sites/default/files/policy/1%20-%20integrity%20and%20plagiarism%20ovpaa%20v3.pdf) and IEEE referencing guidelines found at <https://loop.dcu.ie/mod/url/view.php?id=448779>.

Name: \_\_\_\_\_ Date: \_\_\_\_\_

# **Abstract**

This report researches the LED Block Cipher and implements the algorithm in the hardware description language VHDL. In addition to this, the algorithm was implemented on the Xilinx Zynq – 7000 Development board specifically targeting the Zynq ARM processor. The project also looks at different architecture version of the cipher and compares Iterative and Pipeline architectures to each other in terms of area utilisation and power consumption. Alongside the analysis, the report includes a step-by-step process of the inner working of LED to give future developers and researches an easy method of debugging and error checking. In addition to this, a tutorial on how to develop a custom IP in Vivado to run on the Zynq processor through the use of the Xilinx SDK.

# Table of Contents

Acknowledgements.....	ii
Declaration.....	ii
Abstract.....	iii
Table of Figures .....	vi
Chapter 1 – Introduction .....	1
Chapter 2 - Technical Background .....	2
2.1 VHDL .....	2
2.2 LED.....	3
2.2.1 addConstants .....	4
2.2.2 subCells.....	5
2.2.3 shiftRows .....	5
2.2.4 mixColumns.....	6
2.3 Zybo – Zynq-7000 Development Board .....	8
2.4 Finite Field Multiplication .....	9
2.5 LED Decryption.....	9
2.5.1 Cipher Feedback (CFB) .....	10
2.5.2 Output Feedback (OFB).....	11
2.5.3 Counter (CTR) .....	12
2.6 Summary .....	13
Chapter 3 – Designs of the Cipher.....	14
3.1 C++ .....	14
3.2 VHDL .....	14
3.2.1 Method 1: Serial/Iterative Architecture .....	14
3.2.2 Method 2: Pipeline Architecture .....	15

3.3 Vivado IP Creator .....	18
3.4 Developing for Zybo Board .....	18
3.5 Xilinx SDK (C code) .....	18
3.6 Summary .....	19
Chapter 4 – Implementation and Testing .....	20
4.1 Initial Testing and Groundwork .....	20
4.2 C++ .....	20
4.3 VHDL – Version 1 .....	21
4.4 VHDL – Version 2 (Iterative Architecture) .....	21
4.5 VHDL – Version 3 (Pipeline Architecture) .....	23
4.6 Versions 4 and 5 .....	24
4.7 IP Creation .....	24
4.8 C Code Development (With the Xilinx SDK) .....	25
4.9 Summary .....	27
Chapter 5 – Results and Discussion .....	28
5.1 Power Results .....	28
5.2 Area Utilisation Results .....	30
5.3 Discussion of Results .....	30
Chapter 6 – Ethics .....	32
6.1 Handling of Passwords and Personal Data .....	32
6.2 Energy Wastage and Cryptocurrency Mining .....	32
Chapter 7 – Conclusions and Further Research .....	34
7.1 Improvements and Further Research .....	34
References .....	35
Appendix A – Encryption Example .....	39
Appendix B – IP Integrator and SDK Tutorial .....	43

# Table of Figures

Figure 1 - VHDL Example .....	2
Figure 2 - State matrix .....	3
Figure 3 - Flow Diagram of the LED Algorithm .....	4
Figure 4 - Step Process .....	4
Figure 5 -addConstants Matrix and Output Result .....	5
Figure 6 - subCells S-Box.....	5
Figure 7 - shiftRows Output Matrix.....	6
Figure 8 - mixColumns Output Matrix .....	7
Figure 9 - Zybo Development Board .....	8
Figure 10 - CFB Encryption and Decryption.....	11
Figure 11 - Output Feedback (OFB).....	12
Figure 12 – Counter (CTR).....	13
Figure 13 - Iterative Architecture .....	15
Figure 14 - Pipeline Architecture.....	17
Figure 15 – Iterative VHDL File Hierarchy .....	22
Figure 16 - Overview of the Iterative VHDL Schematic.....	22
Figure 17 - Internal View of round0 .....	22
Figure 18 - Pipeline VHDL File Hierarchy .....	23
Figure 19 – Overview of the Pipeline VHDL Schematic .....	23
Figure 20 - IP Integrator Block Design.....	25
Figure 21 - Overview of the Xilinx SDK with the Output Terminal.....	26
Figure 22 - 64-bit LED Test Vectors .....	26

# Chapter 1 – Introduction

The aim of this project is to develop a VHDL implementation of the LED lightweight Block Cipher algorithm and to implement the design on an FPGA. The project will also evaluate the ciphers performance in terms of power consumption and resource requirements.

A cipher is a way of disguising data from sender to receiver using a specifically designed algorithm. The sender encrypts the data they want to protect using this algorithm and a user defined (or randomly generated) key, and the receiver decrypts (decodes) the encrypted data to bring it back to its original state. The most recognisable cipher is the Caesar Cipher [1], which shifts each letter of the alphabet by a certain number of positions, which is defined by the key.

While some ciphers encrypt parts of the input data separately, usually one bit at a time. This is generally referred to as a Stream Cipher [2]. A Block Cipher, however, applies its encryption algorithm to the entire block of text at the same time [3]. One advantage of this method is that implementing a Block Cipher design requires less hardware restrictions, which broadens the range of devices that can implement it [4].

Lightweight Block Ciphers, such as the one used in this project, are designed to use as little computing power and hardware resources as necessary. With the rise in different types of Internet of Things (IoT) smart devices such as smart plugs and smart thermostats, which are limited in hardware resources, the need for ways to protect lightweight devices connected to the internet has increased [5].

The LED (Lightweight Encryption Device) Block Cipher was presented at the 2011 Cryptographic Hardware and Embedded Systems (CHES) conference [6]. The cipher was designed to be as lightweight in hardware as possible, while still providing adequate levels of security and give reasonably good performance compared to ciphers of a similar nature. LED is a 64-bit block cipher, meaning it only deals with a 64-bit plaintext input. The key can vary in size from 64-bit to 128-bit, however this project will only deal with the 64-bit key version.

One aim for this project is to provide a resource for people developing their own designs for LED and for the Zynq ARM processor. Since the exact step-by-step process of LED is not provided, this report aims to create this process for future developers to use and to compare their designs to. This report also aims to provide a step-by-step process on developing for the Zynq ARM processor, and the steps taken from VHDL code to developing on the processor.

# Chapter 2 - Technical Background

This chapter lists and describes the fundamental theories underpinning the entire project. It will cover the background of the software side of the project as well as detail the hardware used and will briefly discuss some mathematical theory required to understand some sections of the cipher's algorithm.

## 2.1 VHDL

VHDL (VHSIC Hardware Description Language) is a hardware description programming language developed by the US Department of Defence in the early 1980s to standardise the way in which hardware designs were documented [7]. The language was standardised by the IEEE (Institute of Electrical and Electronics Engineers) in 1987 and again in 1993, with minor revisions occurring sporadically between then and 2008.

A typical VHDL program contains an “*Entity*” and an “*Architecture*”. The “*Entity*” describes the programs interface, which contains the ports used in the program, while the “*Architecture*” describes the function and implementation of the program. The example shown in Figure 1 is a simple 4-bit AND gate. The inputs A and B, as well as the output, are initialised as an “*std\_logic\_vector*”, which is a variable type in VHDL (Essentially array of bit values). In the architecture, the output is assigned to be equal to A AND B.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity and_gate is
    port ( a : in std_logic;
          b : in std_logic;
          output : out std_logic);
end and_gate;

architecture Behavioral of and_gate is
begin

    output <= a and b;

end Behavioral;
```

Figure 1 - VHDL Example



One interesting thing to note about VHDL is that the language is *not* case sensitive, unlike most programming languages. Because of this, the standard case type programmers use for naming functions and variables is “*snake case*”, which separates words with underscores (For example, in “*std\_logic\_vector*” above). Other programming languages, such as Java and C++ for example, use “*camel case*”, which separates each word by capitalising the next word. For example, the function “*printOutputText*” would be considered camel case.

To program the cipher in VHDL, the Xilinx Vivado Design Suite will be used. This is a VHDL IDE which can perform high-level synthesis (Which essentially converts a description of a circuit into a logic circuit), run various types of analysis, and perform simulations of a program in response to various stimuli [8] [9]. Vivado has many useful tools which will be helpful during almost every stage of this project, such as the IP Integrator (Which synthesises VHDL designs for hardware use) and the previously mentioned analysis tools.

## 2.2 LED

As mentioned in the introduction section, LED takes a 64-bit input and a key size ranging from 64-bits to 128-bits. The input is viewed as a 4x4 matrix, where each cell of the matrix is 4-bits (also known as a nibble). This is referred to as the cipher’s “*State*” and is illustrated in Figure 2.

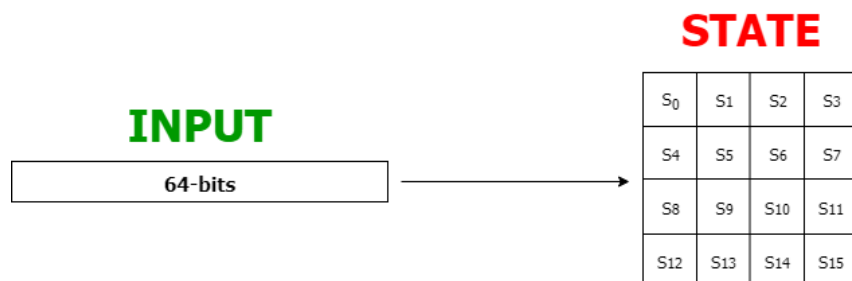


Figure 2 - State matrix

There are 32 overall rounds of encryption for the 64-bit key LED version. Each round consists of four functions; “*addConstants*”, “*subCells*”, “*shiftRows*”, and “*mixColumns*”, and the data flows through them in that order. These four functions are run four times in the order described, which makes up one “*Full Round*” of encryption. This is called the “*Step*” process and the functions are described in more detail in the sections below. Before each Step process, and

after the final Step process, the State and the key are combined using bitwise exclusive-or (Xor).

Figure 3 shows a flow diagram of the entire LED algorithm. To make the Step process easier to visualise, Figure 4 shows the inner working of the process and how the data flows through each Step process.

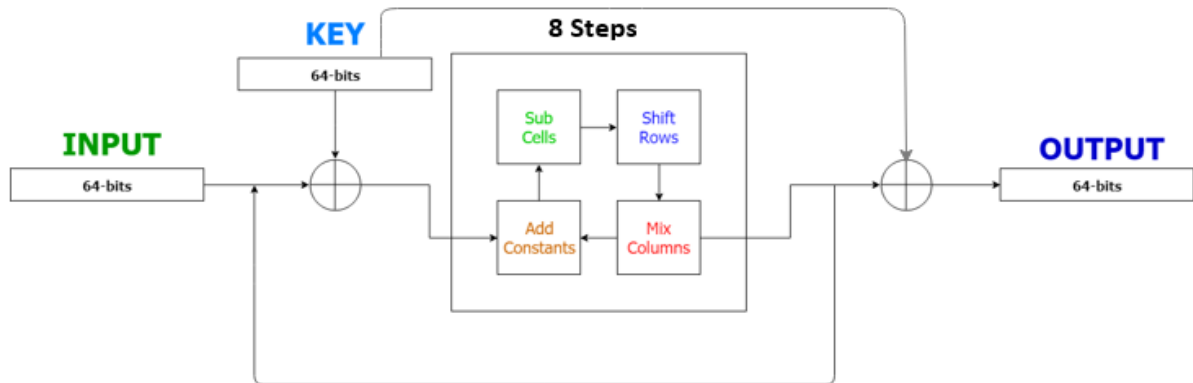


Figure 3 - Flow Diagram of the LED Algorithm

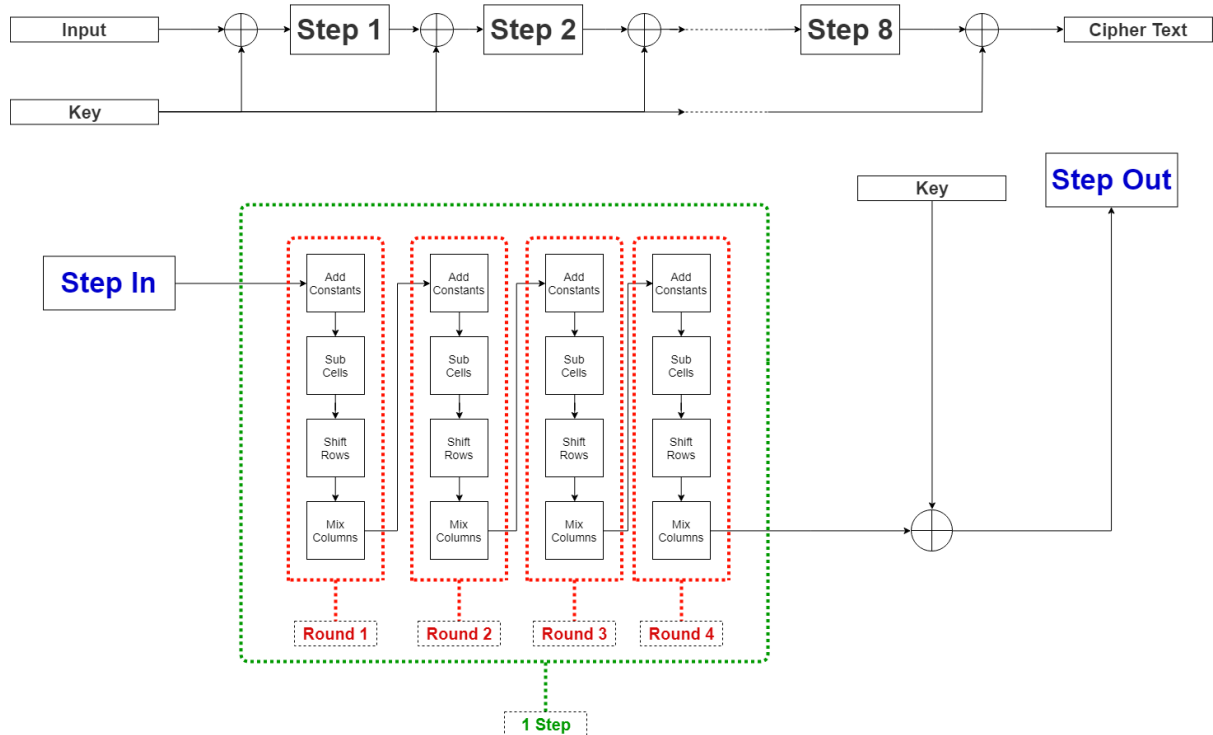


Figure 4 - Step Process

### 2.2.1 addConstants

The first of the Step process functions combines round-specific constant values and a key size value in a 4x4 matrix and combines it with the State using bitwise exclusive-Or. The complete matrix is shown in Figure 5. The  $KS_i$  figure represents the  $i^{\text{th}}$  bit of the key size value, which is

64 in this case (or 40 in hexadecimal), while the  $rc_i$  figure represents the  $i^{\text{th}}$  bit of the round constant value. The “||” notation represents the concatenation of bits.

$$AC = \begin{bmatrix} 0000 \text{ Xor } (KS_7||KS_6||KS_5||KS_4) & (rc_5|rc_4|rc_3) & 0 & 0 \\ 0001 \text{ Xor } (KS_7||KS_6||KS_5||KS_4) & (rc_2|rc_1|rc_0) & 0 & 0 \\ 0010 \text{ Xor } (KS_3||KS_2||KS_1||KS_0) & (rc_5|rc_4|rc_3) & 0 & 0 \\ 0011 \text{ Xor } (KS_3||KS_2||KS_1||KS_0) & (rc_2|rc_1|rc_0) & 0 & 0 \end{bmatrix}$$

$$\text{Cipher Text} = AC \text{ Xor State}$$

Figure 5 -addConstants Matrix and Output Result

### 2.2.2 subCells

The subCells function takes the output of the addConstants function and uses an S-Box to swap each cell in the State matrix with its corresponding S-Box value. The functionality of this S-Box is shown in Figure 6. LED uses the S-Box found in the PRESENT lightweight block cipher, which was designed for hardware optimisation [10].

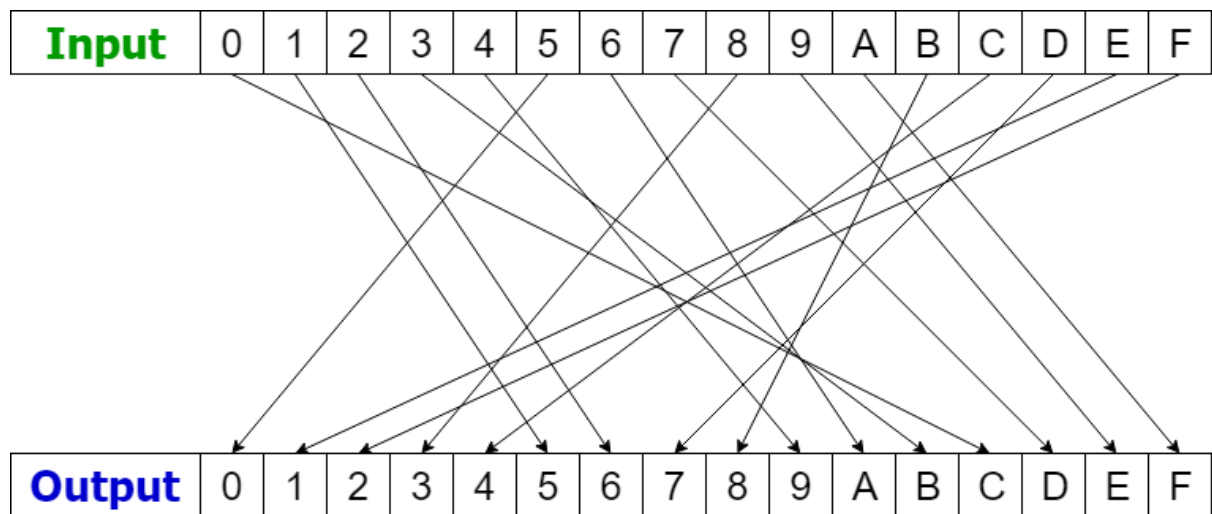


Figure 6 - subCells S-Box

### 2.2.3 shiftRows

The shiftRows function takes the output of subCells and shifts each cell in the matrix to the left by “ $n$ ” positions, where “ $n$ ” is the row number of the cell in the matrix. As can be seen in Figure 7, row zero shifts zero positions to the left, row one shifts one position to the left, and so on.

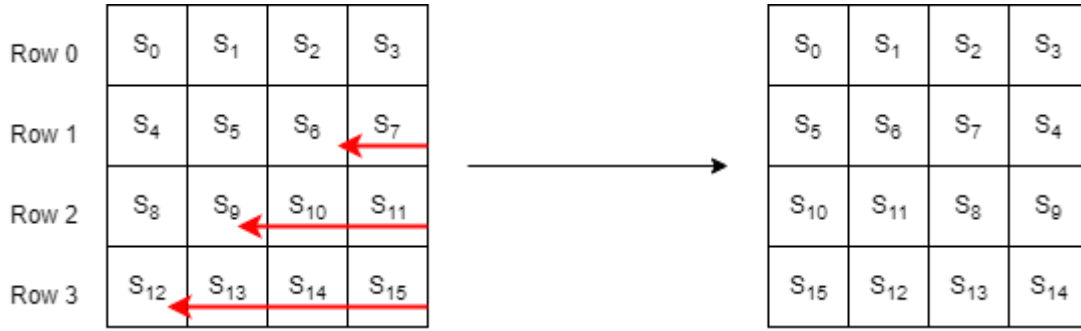


Figure 7 - shiftRows Output Matrix

## 2.2.4 mixColumns

The mixColumns function, as well as the shiftRows function, is used to achieve diffusion in the cipher, where diffusion is changing roughly half the output for every bit change in the input [11]. The function works by multiplying an MDS (Maximum Distance Separable) matrix “ $M$ ” with each column of the State. Finite Field Multiplication is used for multiplying the matrices, which is explained in a section below. The MDS matrix is a little complicated to derive, but it results in the following hardcoded matrix:

$$M = \begin{bmatrix} 4 & 1 & 2 & 2 \\ 8 & 6 & 5 & 6 \\ B & E & A & 9 \\ 2 & 2 & F & B \end{bmatrix}$$

Figure 8 shows how the mixColumns step multiplies the matrix “ $M$ ” by each column of the State. Instead of adding the products for each cell, the products are combined using bitwise exclusive-or (To prevent an overflow). The columns then are re-combined to give the output.

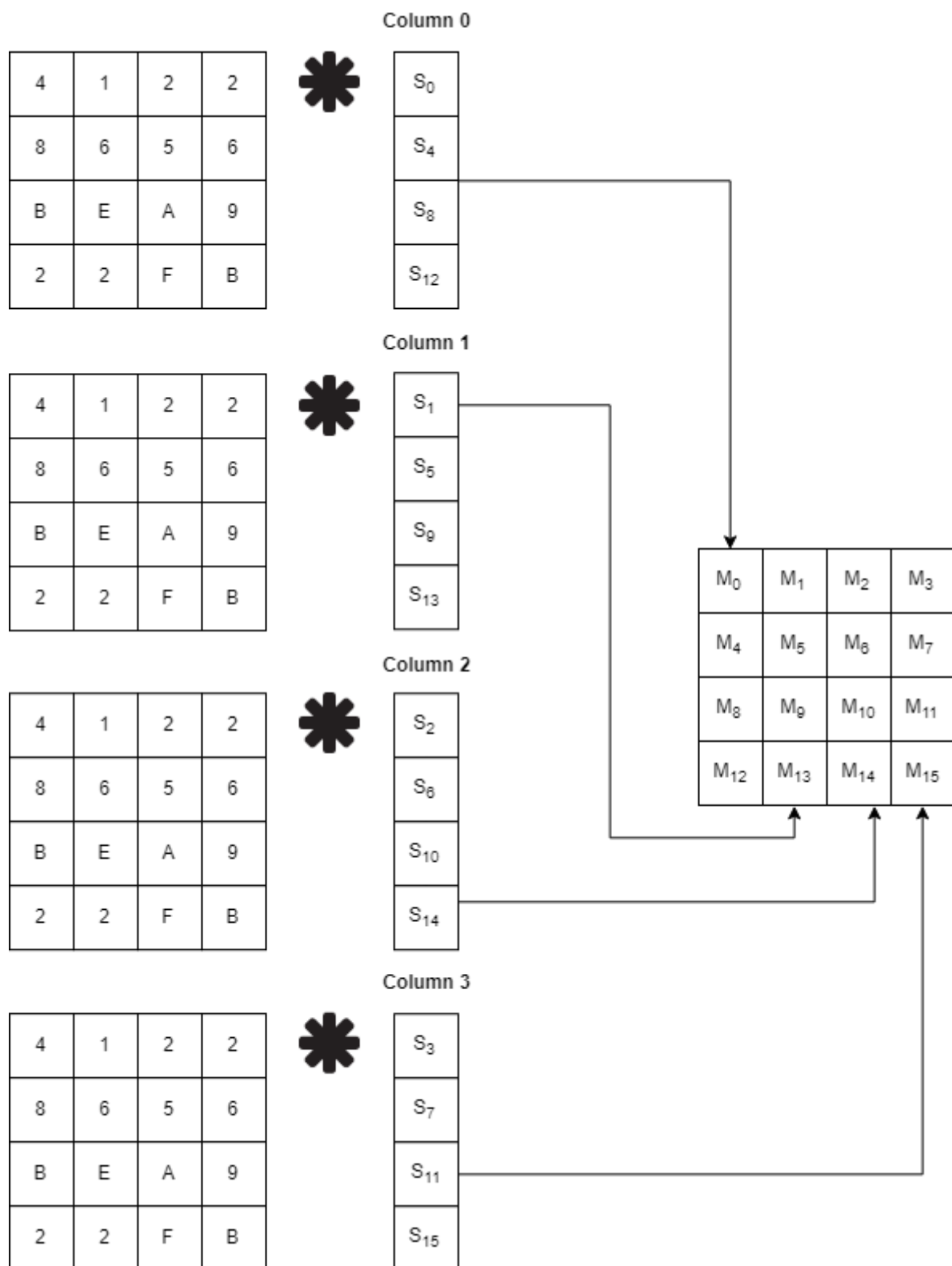


Figure 8 - mixColumns Output Matrix

## 2.3 Zybo – Zynq-7000 Development Board

The aim of this project is to implement the LED Block Cipher in VHDL targeting an FPGA. An FPGA (Field Programmable Gate Array) is an integrated circuit designed to be programmed and reprogrammed as many times as the user desires. They are semiconductor devices which use a matrix field of configurable logic blocks, which enables the reprogramming the of the device after manufacturing [12].

The FPGA used is Digilent's Zynq-7000 Development Board, and the project will specifically target the boards Zynq ARM processor [13] [14]. The Zybo Z7 is a powerful board which can be found from between €150 and €300 depending on the version and the model. The board contains a variety of ports for interfacing purposes, such as a HDMI and 32 Shield I/O ports. The board also contains a microSD slot, and ethernet port, and four USB slots. The main draw to this board for FPGA developers is the Zynq ARM processor. Figure 9 shows a picture of the Zybo development board.

The version of the board used in this project features the Xilinx Zynq Z-7010 AP SoC, which is a slightly older version of the Zybo board. The “SoC” part of the name stands for “*System on Chip*”, which means the Zynq ARM processor contains a processor core. In this case, it is a dual-core Cortex-A9 processor and an FPGA [15]. The ultimate goal of this project, should the main requirements be met early, would be to develop a program to interface between the FPGA and SoC, where the encryption would be achieved on the FPGA and the SoC would sent inputs and receive outputs from the FPGA.

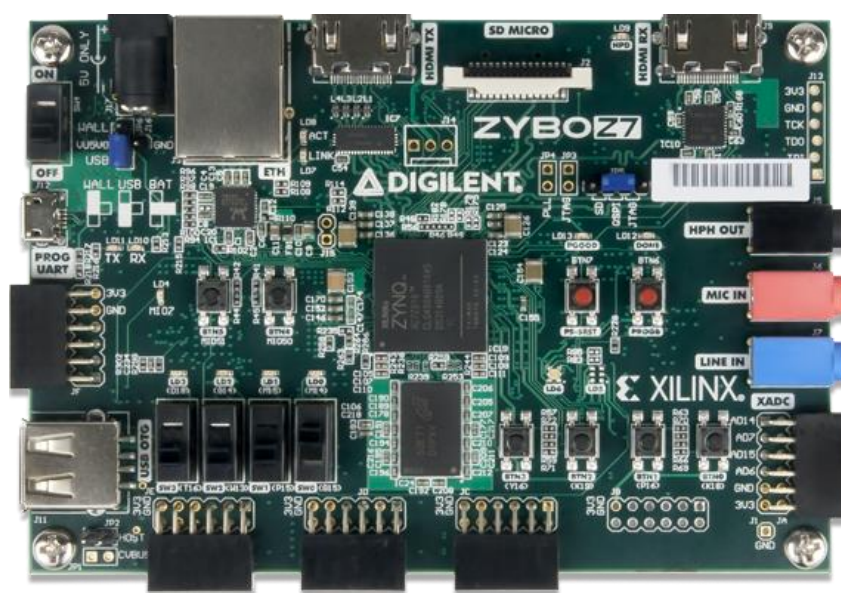


Figure 9 - Zybo Development Board

## 2.4 Finite Field Multiplication

Many AES-like ciphers achieve diffusion by using a multiplication method known as *Finite Field Multiplication* (Also known as a Galois Field). A finite field limits a number range, so addition and multiplication can be performed within that range. The field size is defined by the notation  $GF(2^n)$ , where  $n$  is a real number. For example, LED uses its MDS matrix over a finite field of  $GF(2^4)$ , which limits the number range from 0 to 15, or 0xF in hexadecimal. Other cases include AES which uses a finite field of  $GF(2^8)$ , which limits the range of numbers from 0 to 255, or 0xFF in hexadecimal [16].

The multiplication works by “*reducing*” the product once it exceeds a certain value, which is labelled a “*Reduction Polynomial*”. The simplest way of explaining is to show an example. Using a reduction polynomial of 0x13 (or 10011 in binary), the numbers 5 and 4 are multiplied together in a finite field of  $GF(2^8)$ .

5, which is 0101 in binary, is represented by the polynomial  $0X^3 + X^2 + 0X + 1$ , where each power of  $X$  represents its position in the binary number. Similarly, 4 (0100) is represented as  $0X^3 + X^2 + 0X + 0$ . Multiplying these polynomials together gives  $X^4 + X^2$ , which is represented in binary as 10100, or 20 in decimal. Since the field can only be in the range from 0 to 15, the result must be “*reduced*” back into the range. To do this, the reduction polynomial is subtracted from the product. This gives  $X^4 + X^2 - (X^4 + X + 1)$  which equals  $X^2 - X - 1$ . Converting this polynomial back into binary gives 0111, or 7 in decimal. So, the product of 5 and 4 in a finite field of  $GF(2^4)$  with a reduction polynomial of 0x13 gives 7.

## 2.5 LED Decryption

The paper proposing the design for LED did not state a specific decryption algorithm, so because of this the decryption was not a priority target for this project. The decryption would be a straight reverse of the encryption algorithm, which would be relatively simple to achieve for the first three functions. However, reverse mixColumns would be more challenging, as division in the finite field (reversing the finite field multiplication) is not a direct reverse of the multiplication algorithm and requires multiple tables of inverse multiplication as described in the document on finite field arithmetic [17]. Therefore, as this would be too time consuming and not a focus of the project, it was decided that analysis would only include the encryption process.

There are methods of deciphering Block Ciphers without the use of a direct decryption algorithm, however. A block cipher “*Mode of Operation*” is an algorithm used by block ciphers to decrypt a message larger than the standard input size (For example, LED has a standard input size of 64-bits) [18]. Using some of these algorithms, a large set of data can be encrypted and decrypted using only the encryption algorithm.

### **2.5.1 Cipher Feedback (CFB)**

Cipher Feedback essentially transforms the block cipher into a stream cipher. For encryption, as can be seen from Figure 10 below, the input to the cipher is not the input plain text, but instead an “*Initialisation Vector*” (IV), which is usually a randomly generated value the same as the input block size. After the encryption process is complete, the “*actual*” plain text to be encrypted is Xor-ed with the encrypted IV text to give the “*actual*” cipher text. This cipher text is then fed back into the encryption algorithm again. Once that encryption has been completed, the “*new*” plain text is Xor-ed with the encrypted text to give the next cipher text. This process is completed “*n*” number of times.

To decrypt the text, the procedure is reversed. Instead of the plain text Xor-ed with the encrypted text, the cipher text of the corresponding round is Xor-ed with the encrypted text. This will give the plain text of that round. This *cipher* text is then fed back into the encryption algorithm to decrypt the rest of the input data.



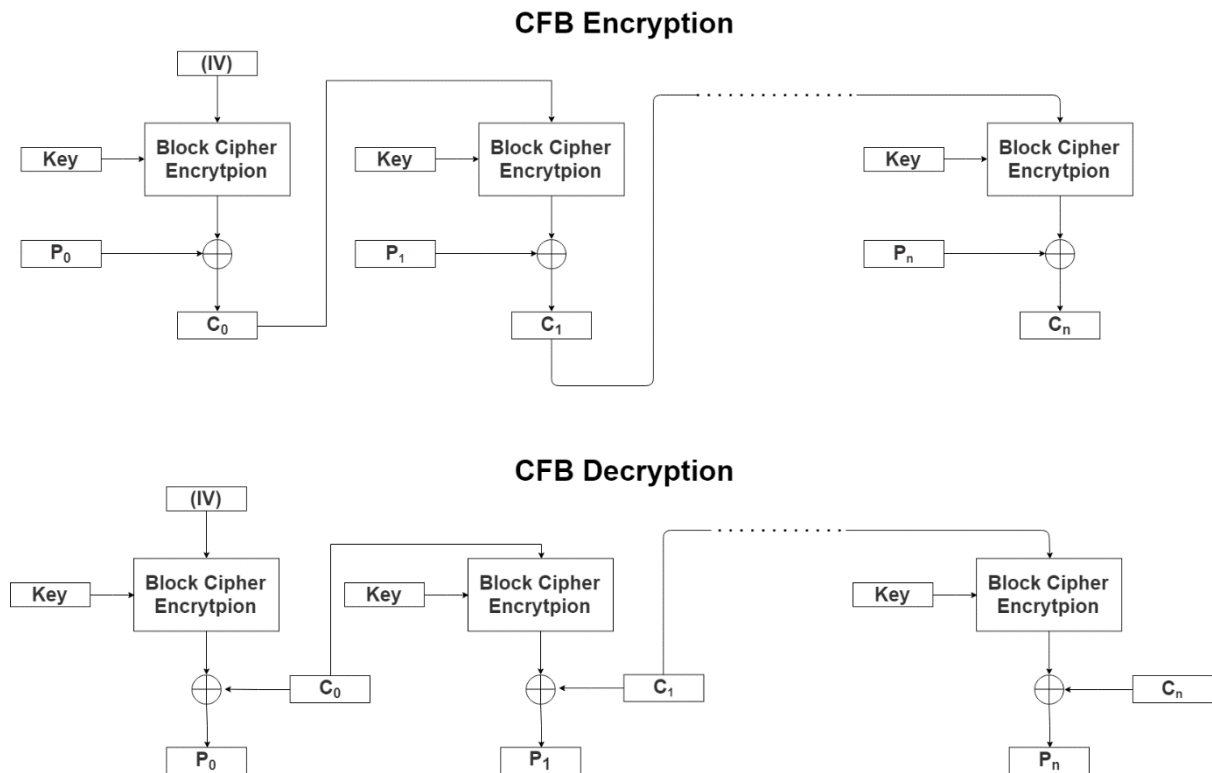


Figure 10 - CFB Encryption and Decryption

### 2.5.2 Output Feedback (OFB)

Output Feedback is almost identical to Cipher Feedback, with the only difference being what the data fed back is. In OFBs case, the feedback is taken *before* the plain text and the cipher text are Xor-ed. This is the same for the decryption as well. Figure 11 illustrates this process.

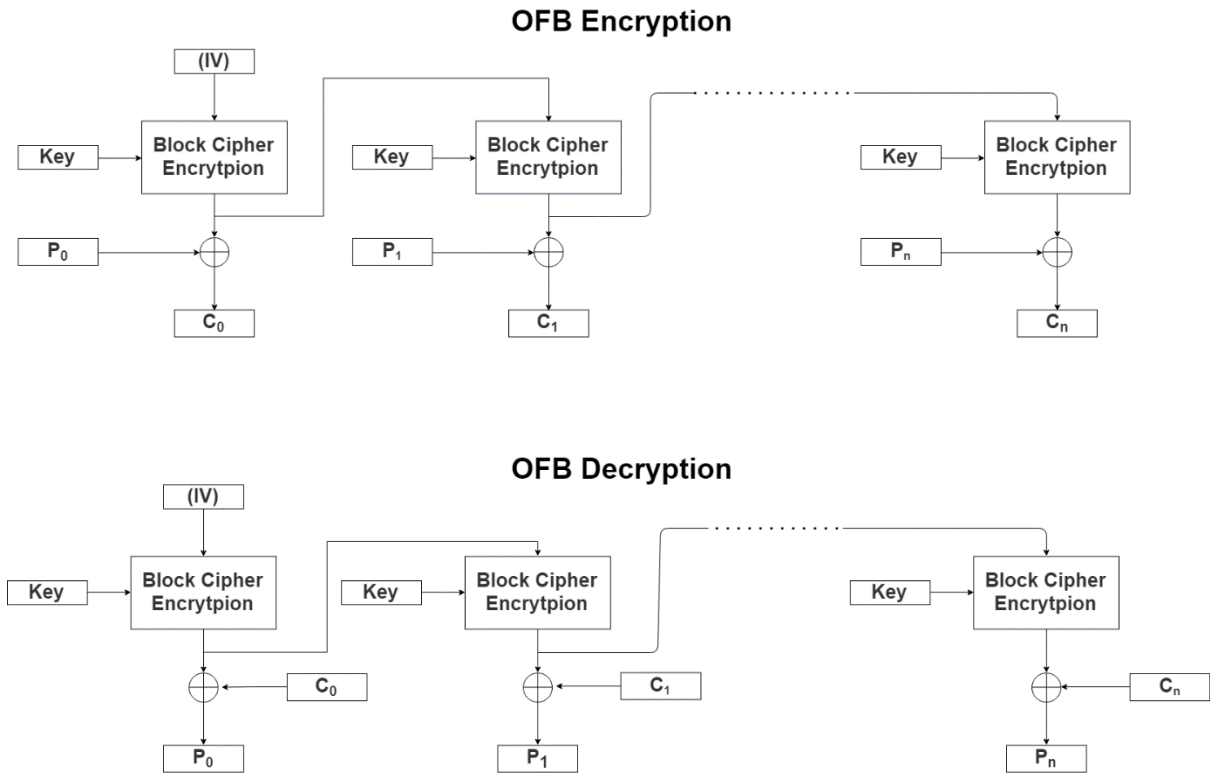
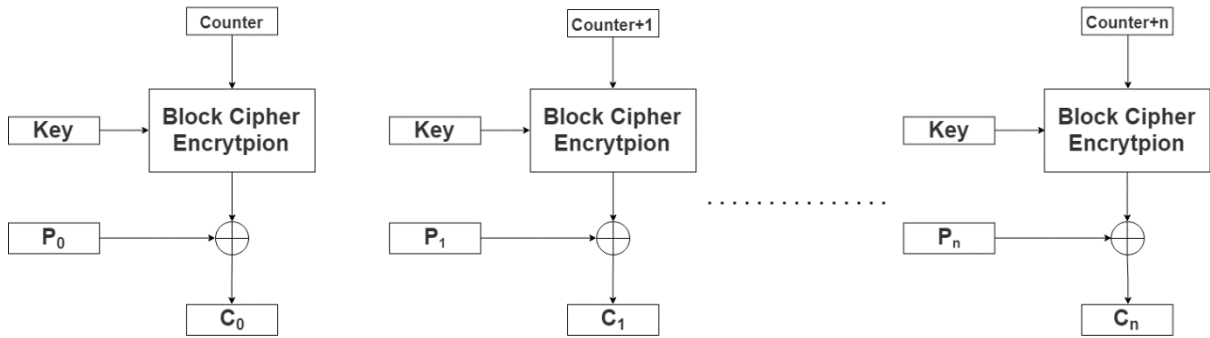


Figure 11 - Output Feedback (OFB)

### 2.5.3 Counter (CTR)

Contrary to the first two modes, Counter mode does not use any feedback loops or initialisation vectors. Instead, Counter mode runs using a counter that increments for each encryption that is run. The counter is initialised at a specified value, and after encrypting the input counter, it is Xor-ed with the plain text to generate the cipher text, just like the other modes of operation. For decryption, the counter is encrypted, and the cipher text is Xor-ed with the encrypted text to give the original plain text. Figure 12 shows the encryption and decryption process.

## CTR Encryption



## CTR Decryption

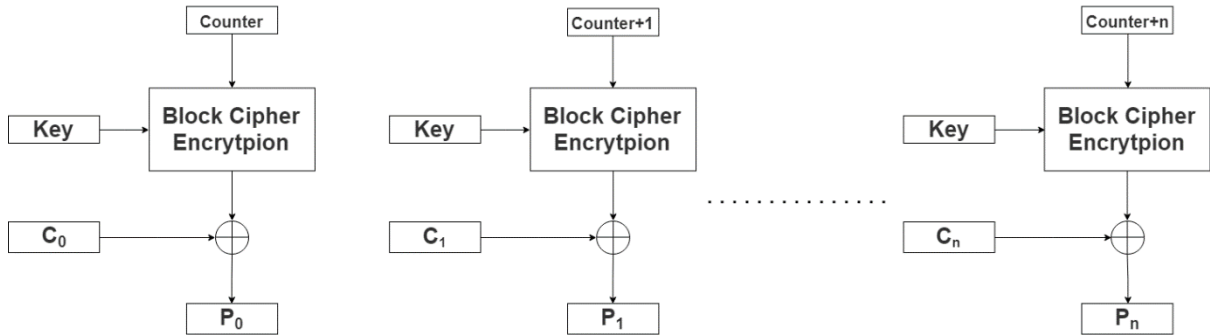


Figure 12 – Counter (CTR)

## 2.6 Summary

In this chapter the background theory necessary for this project was discussed. Hardware, software, and theoretical details were described in detail and how they would be used in the project. In terms of hardware, the Zybo board was introduced, along with a description of the Zynq ARM processor and how it will be used in this project. For the software side, the basics of VHDL were discussed, along with some example code, as well as the design of the LED Block Cipher algorithm. The various software needed to program the cipher, as well as implement the cipher on hardware, were presented and discussed. In regard to theoretical background, the mathematical principle of “*Finite Field Arithmetic*” was introduced and described, along with an example. Finally, the topic of block cipher “*Mode of Operation*” was introduced and explained how large blocks of data can be encrypted and decrypted using three techniques; “*Cipher Feedback*”, “*Output Feedback*”, and “*Counter*”.

## Chapter 3 – Designs of the Cipher

Chapter 3 discusses the proposed design choice of the project. The different architectures designed are explained, as well as why they were chosen and how they will be used. How the Zybo board will be programmed and how the VHDL code will translate to the Xilinx SDK is also discussed in this chapter.

### 3.1 C++

To give an initial reference point for the functionality of the cipher, a C++ version of the cipher will be developed before work on the VHDL version will take place. The goal was to make bug testing a more straightforward process, as well as to give a firm idea as to what the steps of the cipher involve.

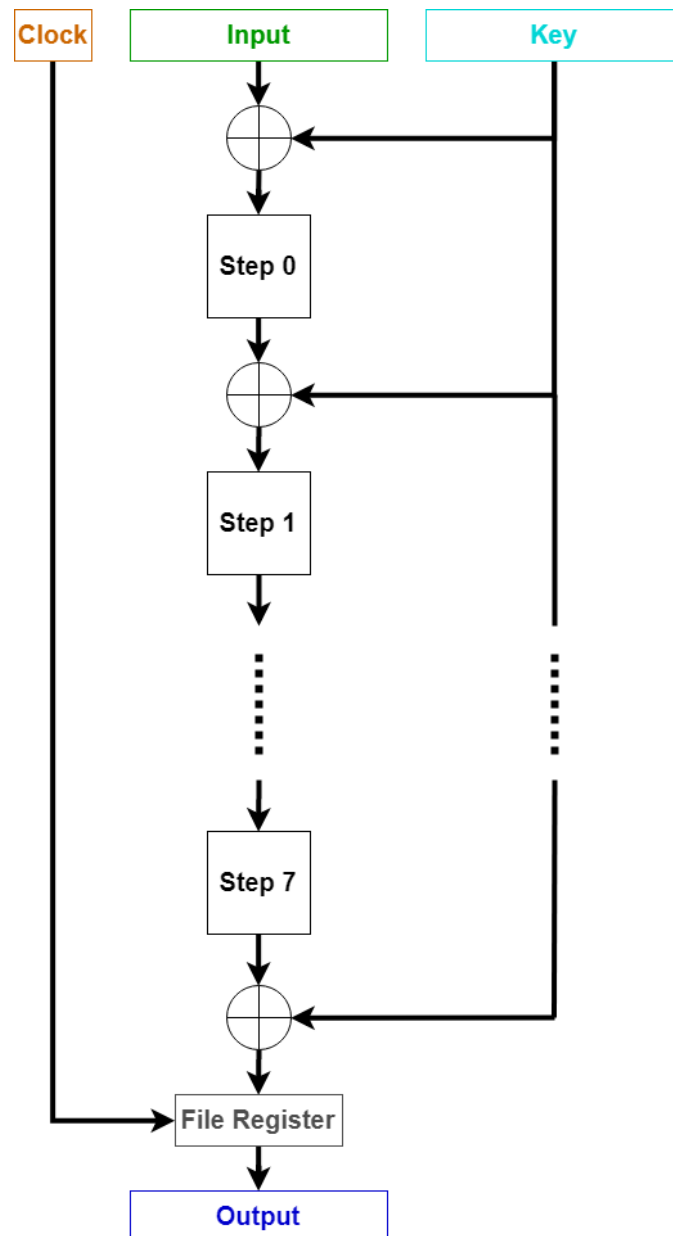
### 3.2 VHDL

The main aspect of the project was to create a VHDL version of LED and implement it on an FPGA device. Two different types of VHDL architectures for LED were designed, both having two versions of VHDL code themselves; one with look-up operations for the mixColumns function and one which calculates the results of the matrix multiplication. These two architectures were chosen as they are vastly different in terms of speed and efficiency.

#### 3.2.1 Method 1: Serial/Iterative Architecture

A program with an iterative architecture, sometimes known as serial architecture, processes one input at a time. This way the data is passed through the input and encrypted completely before the next input is read. The data flow diagram for this method is shown in Figure 13 below.

At the end of the encryption process, the output is sent to a file register. This file register gets updated on each rising edge clock cycle. While the file register is not completely necessary for this architecture method, it provides neat comparisons between it and the pipeline architecture method.



*Figure 13 - Iterative Architecture*

There are a few advantages and disadvantages of using this architecture method in design. One advantage is that the encryption process can be done in one clock cycle, so the speed at which a single input is processed is faster compared to other architectures. One disadvantage, however, is that the architecture is inefficient, as only one part of the design is being used at one time. In other words, most of the design lays dormant while the encryption process continues to run.

### **3.2.2 Method 2: Pipeline Architecture**

The next method aimed to use the space taken up by the cipher as efficiently as possible. Pipelining in ciphers do not wait for the encryption process to be finished before taking in new

data. Instead, data is being constantly processed for each round of encryption. For this design method, the result of each round of encryption is stored in a file register. After each round, the data from each register is sent to the next round of encryption. This is illustrated in Figure 14 below. The diagram shows how each *Step* block outputs its key and ciphertext to its own individual file register. When the register detects a rising edge in the clock signal, it outputs its data to the next pipeline stage register and accepts data from the previous pipeline stage register. The data in each the pipeline register flows one stage towards the final stage.

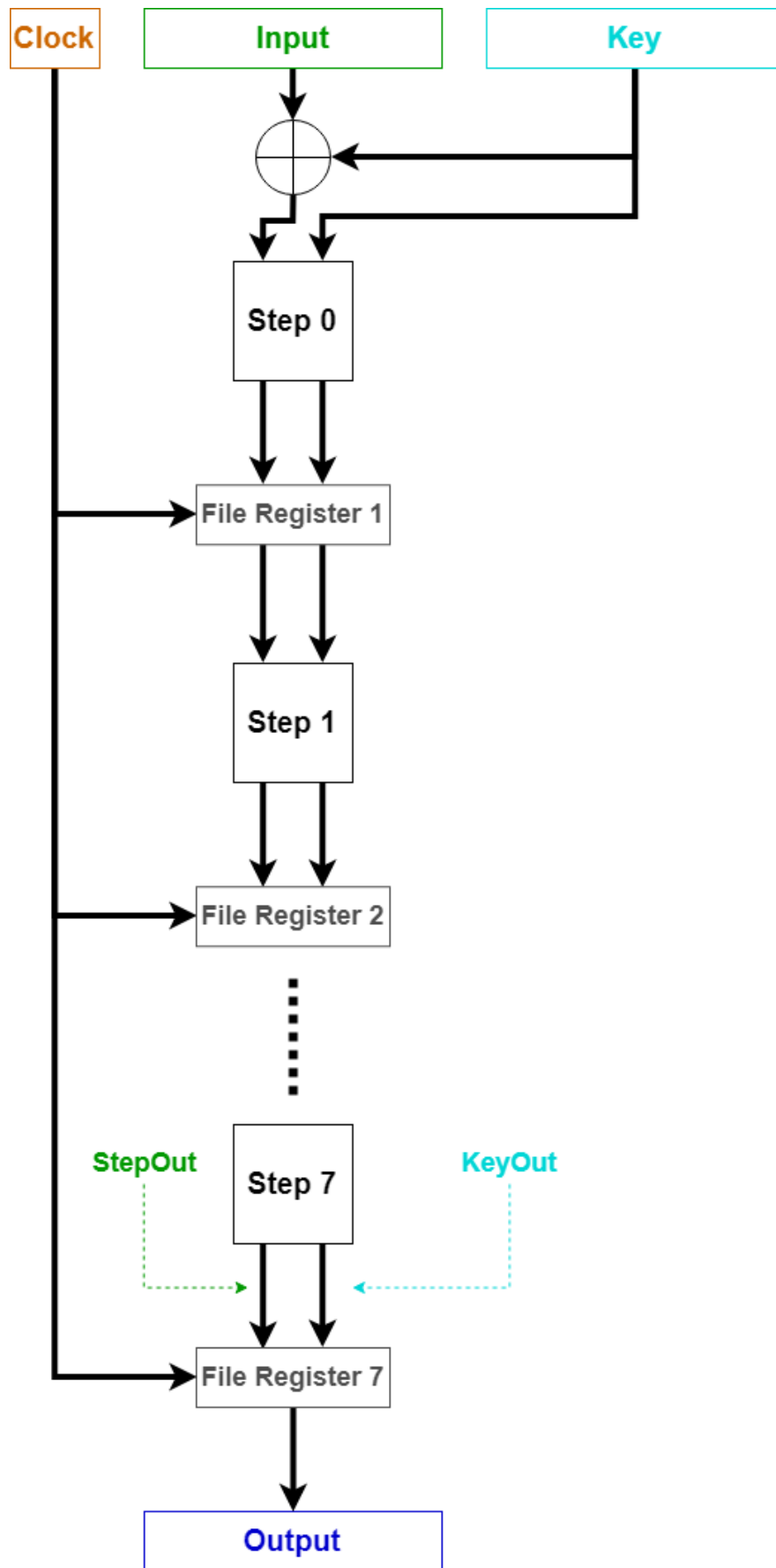


Figure 14 - Pipeline Architecture

One huge benefit of using a pipeline architecture is how efficient the design is compared to iterative architecture, as no section of the cipher is left unused and wasted at any given time. Another advantage is that the input can be updated at every clock cycle at a higher clock frequency, which allows an improvement in throughput over the previous method. One obvious disadvantage to the pipeline architecture method is that it takes 8 clock cycles to encrypt an input.

### **3.3 Vivado IP Creator**

To develop for the Zynq processor, and interface with the internal registers with the chip, the Vivado IP Integrator tool was to be used. This tool allows users to create their own hardware designs and use them with Vivado's block diagram creator. These features were used to create a block diagram which connects the VHDL code with the Zynq ARM processor. The custom IP gives the user access to the Zybo's internal "*Slave*" registers (512 registers which are 32-bits each) and allows inputs and outputs to come from other programs the user has created. In this case, the top level of the LED VHDL component's inputs and outputs were to be assigned to two Slave registers each, as the inputs and outputs are 64-bits and the registers are only 32-bits. Vivado automatically connects the components together in the block diagram, and once the file's HDL wrapper is created and the file exported, the SDK creation can begin.

### **3.4 Developing for Zybo Board**

The main target for this project was to implement the VHDL design on the Zynq ARM processor. The goal was to achieve this by interfacing the SoC part of the processor with the FPGA part, where the FPGA part would hold the encryption and decryption modules and the SoC part would send and receive the data to encrypt and decrypt.

### **3.5 Xilinx SDK (C code)**

The Xilinx Software Development Kit (SDK) handles the software side of the Zynq processor application development. The IDE allows the user to program the FPGA directly, so code can be tweaked, changed, and re-uploaded easily [13]. The C programming language is used to develop for the board, and libraries designed for FPGA development are included when exporting a file to the SDK.



### **3.6 Summary**

In this chapter, the design phase of the project was laid out and explained. The different architecture methods, Iterative and Pipeline, were described, along with the rational for why they were chosen and how they will be evaluated and compared. Other tools, such as the Vivado IP Integrator and the C code for the Xilinx SDK, were discussed, particularly how they will be used in conjunction with the VHDL code.

# Chapter 4 – Implementation and Testing

## 4.1 Initial Testing and Groundwork

Before starting on the C++ code, an on-paper version of the first round of LED was solved. This helped with debugging and testing the C++ code for any errors in the algorithm, as well as making sure the interpretation and understanding of the cipher was correct. The first three functions of the Step process were straightforward enough to write out on paper, but the mixColumns function was the first major hurdle. Upon an initial read and interpretation of the paper, it seemed that this function was just straightforward multiplication. However, finite field arithmetic was a complicated field of mathematics that was not on the syllabus of Electronic and Computer Engineering. Sources which properly described what a finite field is were difficult to find. However, the lecture notes from the University of Alaska on finite fields and the AES Lightweight Block Cipher came in handy and gave examples of how finite field multiplication worked for  $GF(2^4)$ , along with pseudocode for multiplying two numbers in a finite field [17]. Once this section of the cipher was understood, the groundwork of the project was laid out, and work could begin on the software side of the cipher.

## 4.2 C++

To give an initial reference point for the functionality of the cipher, a C++ version of the cipher was developed before work on the VHDL version took place. This was beneficial when development began on the VHDL version, as a fully complete version of the cipher was already created, which made bug testing a much more straightforward process.

Testing began with creating the four main functions, as well as the addRoundKey function. The implementation for the first three Step functions did not take too long, as they were relatively straightforward to code with the on-paper version readily available. The mixColumns step was slightly more complicated but using the document about finite field arithmetic from the previous section [17] as a reference made the debugging process easier.

The C++ code made use of the Vector class for dealing with LEDs matrices, as well as the integer types library [19] [20]. The Vectors were declared as 2D vectors, which acted like arrays. This made it easy to program the functions, as the LED paper describes the functions in terms of 4x4 matrices. No work was done to optimise or simplify the C++ version, as it was not the focus of the project.

An implementation in C by the cipher's creators helped with development, as figuring out which part of the cipher was causing problems was difficult. Their version, along with test vectors, can be found on the downloads section of their website [21].

### **4.3 VHDL – Version 1**

The initial milestone of the project was to develop a VHDL version of the project. The first version of the cipher used an Iterative Architecture and used many ideas from the C++ version described in the section above, making use of VHDL functions [22]. This version was made just to get a working LED VHDL program, so coding efficiency and optimisation were ignored for the time being.

Version 1 made use of VHDL's 2D arrays, which act much like those of the 2D vector arrays in C++, the advantage being that the size of each array element can exactly 4-bits. The process to debug the code was similar to that of the C++ version, where a testbench for each function was made and tested with different inputs.

### **4.4 VHDL – Version 2 (Iterative Architecture)**

The second version of the VHDL program made use of the languages "*Component Instantiation*". Components allow for large programs to be split into individual modules, allowing more structured and easier to follow code [23]. This design split each function into its own component, and each round into its own component again. The top level then combined all these components together to create the LED algorithm. The Hierarchy of the files are shown in Figure 15 below.

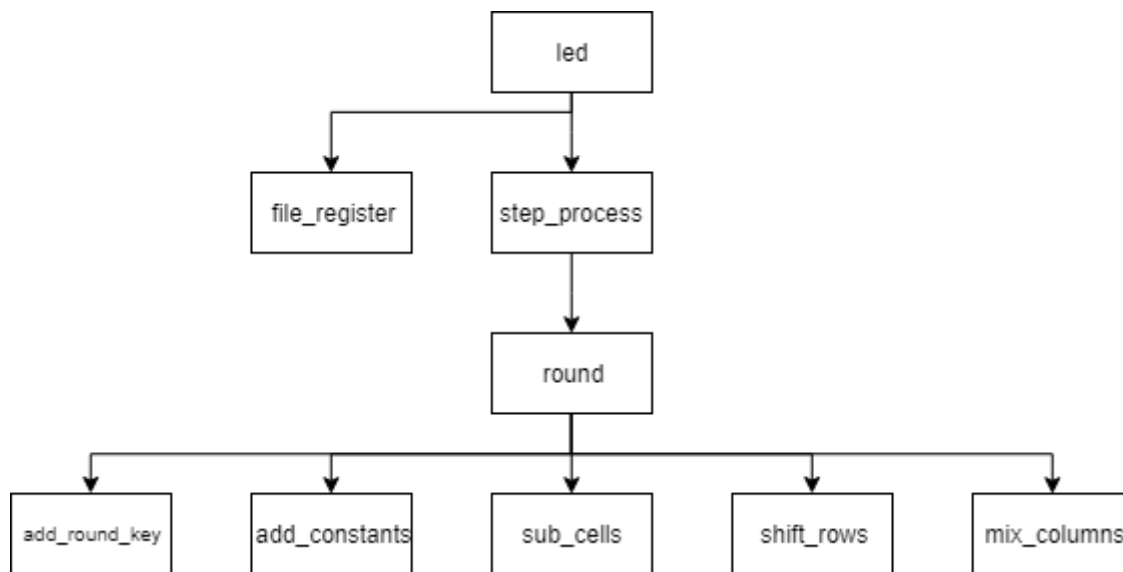


Figure 15 – Iterative VHDL File Hierarchy

The schematic for the program, shown in Figure 16, makes the hierarchy clearer to see. This shows how the components are laid out and how they connect to each other, while Figure 17 shows an internal view of one of the “round” blocks and how those components are connected.

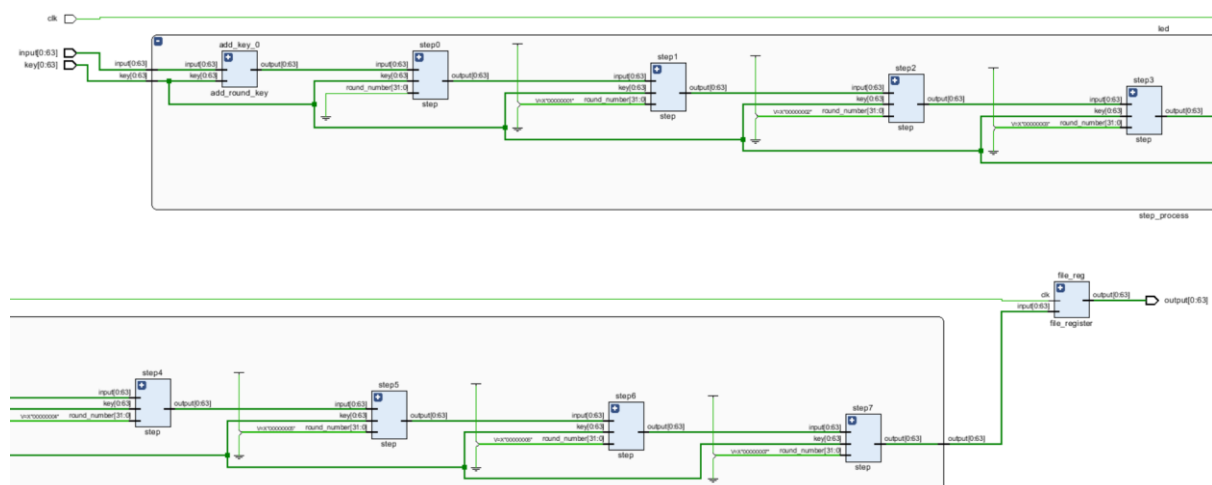


Figure 16 - Overview of the Iterative VHDL Schematic

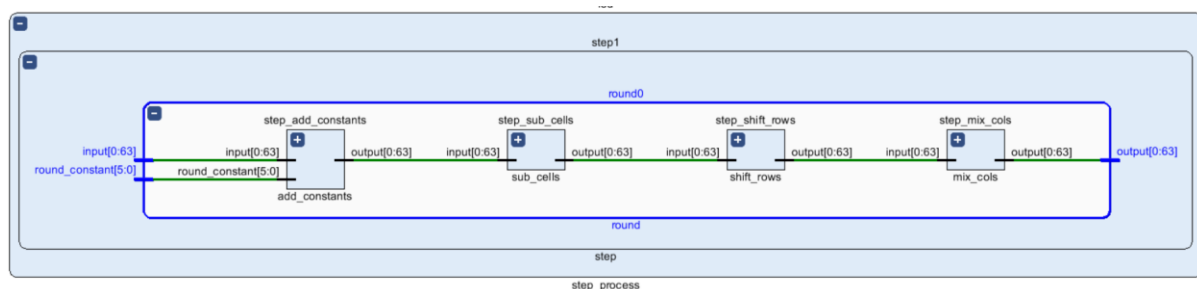


Figure 17 - Internal View of round0

## 4.5 VHDL – Version 3 (Pipeline Architecture)

The third version used the same elements as the second version, the only difference being that each round output its ciphertext and key to a file register. This is to allow the data to flow as described in the pipeline architecture section of chapter 3. The hierarchy and schematic are shown in Figure 18 and Figure 19 respectively. The only minor difference between the iterative version and the pipeline version is the file register component is on the same level as the round component.

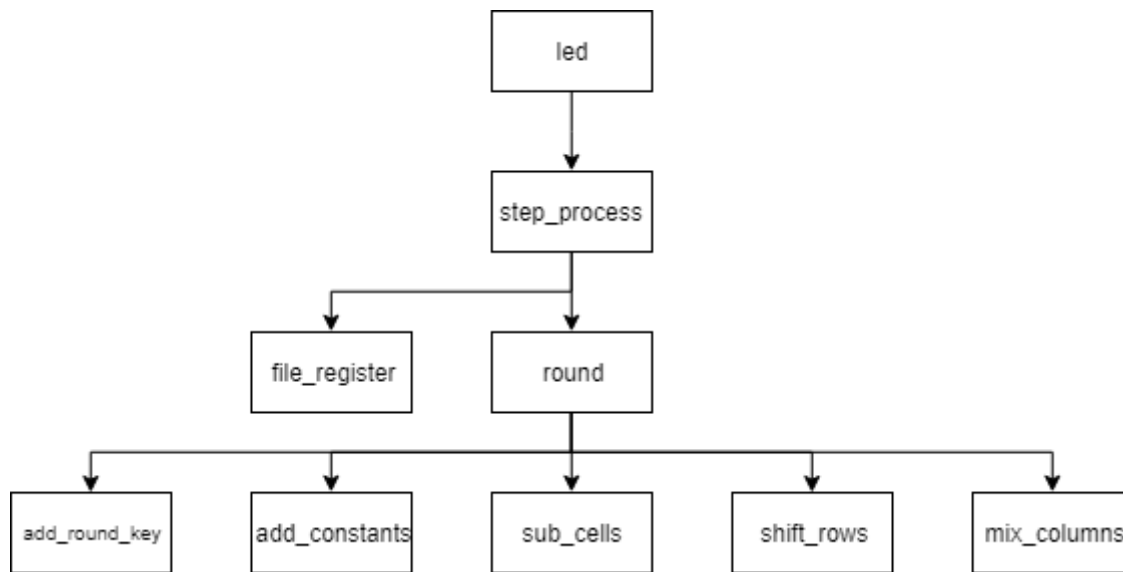


Figure 18 - Pipeline VHDL File Hierarchy

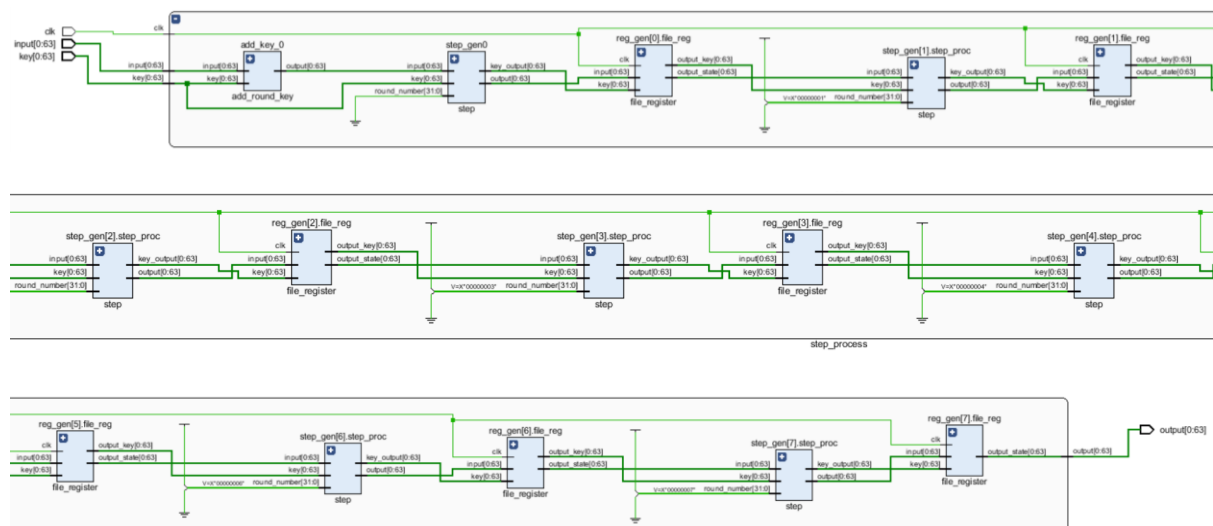


Figure 19 – Overview of the Pipeline VHDL Schematic

Each version of the VHDL files had a corresponding version which used a lookup table for the mixColumns function. These were made to highlight the differences between area and speed

compromises, where the lookup table method would be faster, and the non-lookup table method would require less area consumption. This will be explored further in the results and discussion section.

## 4.6 Versions 4 and 5

Versions 4 and 5 (or Methods 3 and 4) will be discussed briefly, as there's not many differences between them and versions 2 and 3. The difference between them is that they did not use a lookup table to determine the output of the finite field multiplication. Instead, they used an extra component – called “*gf*” – to determine the output. The functions from version 1 for this part were taken and modified slightly to work with the already created files.

## 4.7 IP Creation

There are little tutorials online about how to use the IP Integrator tool in Vivado and how it links to SDK creation and integrating a custom IP with the Zynq ARM processor. A document was received as part of the initial project meeting which detailed some tutorials about working with the tool and programming for the Zybo and Zynq processor, which helped with the initial stages of development [24]. However, the tutorials involving the IP Integrator tool did not all work, specifically in tutorial 5 – “*Adventures with IP Integrator*” where errors about pin assignments in the constraints file were unable to be resolved.

After much research online, a four-part YouTube tutorial by the channel “*ENGRTUTOR*” detailed the exact steps needed from creating a VHDL program to running it on the Zynq ARM processor [25] [26] [27] [28]. These tutorials clearly laid out the steps involved, working through an example of his own as a demonstration. Using this example, it was relatively simple to adjust this example to suit the VHDL program made in the previous section. The program takes two 64-bit inputs and a 64-bit output, along with an input clock signal. As described in a previous section, each register in the IP block is 32-bits, which meant 6 registers were required for the input, key, and output. The clock signal was assigned to the global clock signal “*S\_AXI\_ACLK*”.

Figure 20 shows the Vivado block diagram creation tool, where the Zynq processor and the custom LED IP and connected. Once the HDL wrapper was created, and the file exported, development could then begin on the C code in the Xilinx SDK.

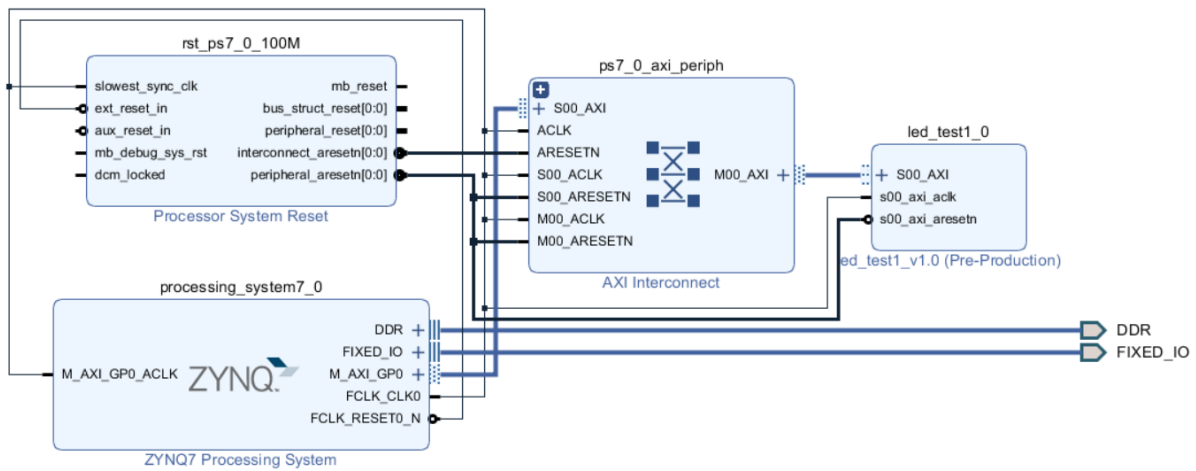


Figure 20 - IP Integrator Block Design

## 4.8 C Code Development (With the Xilinx SDK)

As explained in the previous chapter, the Xilinx SDK is used to directly program the Zybo's FPGA through C code. Using the tutorials mentioned in the previous section as a guide, a program was developed which assigned the input and key different values and sent the result (the output register) to the SDK terminal. To help with this, three libraries were used; *"xparameters.h"*, *"xil\_io.h"*, and *"xil\_types.h"*. The first library contains the base addresses of the file registers, which are required to read from and write to. The second library contains functions required for reading and writing to file registers, such as *"Xil\_Out32"* (which writes a value to a certain address) and *"Xil\_In32"* (which reads in a register value). Finally, the third library contains the variable types, such as U32, which can be used to store information about register values.

Using the *"Program to FPGA"* button on the IDE, the code is uploaded to the Zybo board. Running the program using the *"System Debugger"* tool, the output of the cipher gets printed to the terminal window. Figure 21 shows an overall view of the IDE and the output of the program to the terminal. Figure 22 shows two of the test vectors given by LEDs designers, which match the results shown in the terminal in Figure 21.

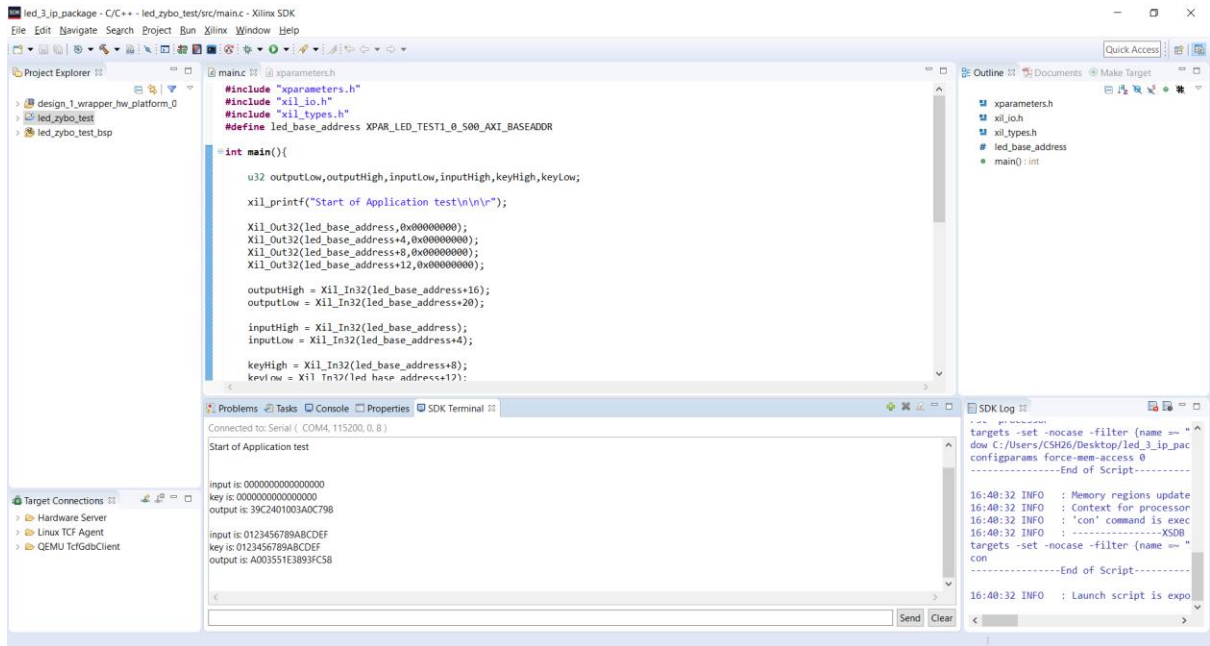


Figure 21 - Overview of the Xilinx SDK with the Output Terminal

	plaintext	key	ciphertext
LED-64	0 0 0 0	0 0 0 0	3 9 C 2
	0 0 0 0	0 0 0 0	4 0 1 0
	0 0 0 0	0 0 0 0	0 3 A 0
	0 0 0 0	0 0 0 0	C 7 9 8
	0 1 2 3	0 1 2 3	A 0 0 3
	4 5 6 7	4 5 6 7	5 5 1 E
	8 9 A B	8 9 A B	3 8 9 3
	C D E F	C D E F	F C 5 8

Figure 22 - 64-bit LED Test Vectors



## 4.9 Summary

In this chapter the designs of the project were put into place and version of the LED algorithm were created in C++ and VHDL. While the first version of the code was slow and unoptimized, the later versions of the code made use of hardware language techniques, such as components and file hierarchy. In addition to this, as version of the VHDL code was developed for the Zynq ARM processor and the correct results were output onto a terminal in the Xilinx SDK. The project goal was met after this was achieved, so work could begin on obtaining results for the area utilisation and power consumption.

## Chapter 5 – Results and Discussion

The two versions of the cipher were compared using Vivado's built-in analysis tools, specifically the power usage and utilisation reports. While the utilisation reports are accurate, the power usage tool is just an estimate, but for comparison between the different architectures and versions it works well, as the results will be consistent.

The results of the area utilisation are shown in Table 5 below. All values are in terms of the number of gates and registers used for the utilisation. The results of the power estimation are slightly different, as the clock period was decreased by a set amount each run to view the effects of clock frequency on power consumption. The values of the power consumption are shown in milliwatts (mW) and are separated into static and dynamic power usage, where dynamic power usage is the power used by the program and static power usage is the rest of the power usage in the device.

Method 1: Iterative Architecture with lookup table for mixColumns step.

Method 2: Iterative Architecture with logic functions for mixColumns step.

Method 3: Pipeline Architecture with lookup table for mixColumns step.

Method 4: Pipeline Architecture with logic functions for mixColumns step.

### 5.1 Power Results

*Table 1 - Method 1 Power Results*

Cipher Version	Clock frequency (KHz)	Clock Period (ns)	Dynamic Power (mW)	Device Static Power(mW)	Signals Power Usage (mW)	Logic Power Usage (mw)
<b>Method 1</b>	1000	1000	27	91	13	13
	2000	500	53	91	25	27
	2500	400	66	92	32	33
	3333	300	89	92	42	44
	5000	200	133	92	63	66
	10000	100	266	93	127	133

Table 2 - Method 2 Power Results

Cipher Version	Clock frequency (KHz)	Clock Period (ns)	Dynamic Power (mW)	Device Static Power(mW)	Signals Power Usage (mW)	Logic Power Usage (mw)
<b>Method 2</b>	1000	1000	26	91	12	14
	2000	500	52	91	24	27
	2500	400	65	92	30	34
	3333	300	87	92	40	45
	5000	200	131	92	50	68
	10000	100	262	93	119	136

Table 3 - Method 3 Power Results

Cipher Version	Clock frequency (KHz)	Clock Period (ns)	Dynamic Power (mW)	Device Static Power(mW)	Signals Power Usage (mW)	Logic Power Usage (mw)
<b>Method 3</b>	1000	1000	17	91	8	9
	2000	500	35	91	16	17
	2500	400	44	91	20	21
	3333	300	58	92	27	28
	5000	200	87	92	41	43
	10000	100	174	92	82	85

Table 4 - Method 4 Power Results

Cipher Version	Clock frequency (KHz)	Clock Period (ns)	Dynamic Power (mW)	Device Static Power(mW)	Signals Power Usage (mW)	Logic Power Usage (mw)
<b>Method 4</b>	1000	1000	18	91	8	9
	2000	500	35	91	16	18
	2500	400	44	91	20	22
	3333	300	59	92	26	30
	5000	200	88	92	39	45
	10000	100	176	92	79	90

## 5.2 Area Utilisation Results

Table 5 - Area Utilisation Results for All Methods

Cipher Version	LUT	FF	%Utilisation LUT	%Utilisation FF
Method 1	4635	64	26.34	0.18
Method 2	4673	64	26.55	0.18
Method 3	4576	960	26	2.73
Method 4	4714	960	26.78	2.73

## 5.3 Discussion of Results

As can be seen by the tables in the section above, the LUT utilisation is relatively similar for both architectures and all methods, with the method 3 using a little bit less space than Method 1 and Method 2. However, due to the use of multiple registers in the Methods 3 and 4, it uses more flip flop storage than Methods 1 and 2. This was expected, as the only component taking up flip flop space was the file register component. The register in Methods 1 and 2 only stored the output cipher text, which meant 64 bits of flip flop storage were required. Methods 3 and 4, however, stored the cipher text and the current key, which meant each register needed 128 bits of flip flop storage (Besides the 8<sup>th</sup> register which only stored the cipher text).

The designs for the non-LUT methods were intended to remove some excess utilisation from the program. However, the program was not optimised in the intended way, and it was not possible to cut down the utilisation in the timeframe of the project. This is why the non-LUT methods contains more area usage than the LUT methods.

As for the power design, it was expected that Methods 3 and 4 would uses less power than the others, as the design was made to be more efficient with its power consumption. As can be seen from the results, this expectation held true, and the pipeline architecture methods used significantly less power than the iterative versions. While there was no significant difference between the power consumption in Methods 3 and 4, there was a difference in where the power was used. As can be seen from the Table 3 and Table 4, more of the power went into the Signals in Method 3 than in the Method 4, and visa-versa for the Logic power usage. This was expected, as the non-LUT methods use a function to determine the result of the finite field multiplication

(Which contributes to logic), while the LUT methods use a lookup table (Which contributes to more signal usage). This held true for Methods 1 and 2 as well.

## **Chapter 6 – Ethics**

### **6.1 Handling of Passwords and Personal Data**

On March 21<sup>st</sup> of this year, Facebook revealed that, due to a bug in their password management systems, hundreds of millions of user's passwords were stored in plain text on internal servers [29]. This meant that a number of Facebook employees had access to these user's passwords, which are estimated to be over 20,000 people. However, after internal investigations about the incident from Facebook, there were no cases found about any employees that used this data maliciously. Shockingly, this is not the first case of a mishap like this; on the 3<sup>rd</sup> of May 2018, twitter CTO Parag Agrawal revealed in a blog that a similar bug to the one Facebook experienced occurred in Twitter's password encryption process [30]. Even though in both cases no data was compromised or stolen, this kind of issue breaks the trust the consumer has with the company.

The GDPR piece on encryption recommends companies encrypt the personal data that they store, as this limits the severity of a data breach. However, encrypting passwords before they are stored should be a legal requirement, as a data breach with plain text passwords would lead to serious issues for the website's users. While it may not be feasible or necessary to encrypt every piece of data given by a user, it is in the authors opinion that personal data must always be kept encrypted in the case of a data breach. This would be beneficial to all parties involved, as users have less of a chance get their personal data compromised, and companies would prevent the loss of trust of their users and incur a less substantial fee from a data breach.

### **6.2 Energy Wastage and Cryptocurrency Mining**

One aspect briefly touched upon in this project is how the first cipher design was less energy efficient than the second design. Developers and researchers strive to make cipher designs as efficient and fast as possible, while using as little space and energy as possible, but it's difficult to find the perfect balance. Often sacrifices must be made in one design area to make full use of another. The designers of LED mention this in the CHES 2011 paper, in which they describe their design as one of the lowest demanding ciphers in terms of hardware and takes up as little space as possible [21]. Energy consumption is a widely researched topic in the area of cryptography, as developers and companies strive to lower their carbon footprint. As a result of this, lightweight block ciphers, such as LED, are more globally used, in addition to the growing trend in smart devices as discussed in chapter 1.

Cryptocurrency has quickly become a global phenomenon in recent years. The process to obtain bitcoin, for example, involves a process called mining. The process to obtain a token, depending on the cryptocurrency, can take a large amount of energy to achieve. As a result of this, Bitcoin mining accounted for 0.33% of the entire world's energy consumption – roughly 22TWh – as described in an article by Alex de Vries [31]. This much energy consumption most definitely contributes to global warming and will continue to grow as long as Bitcoin mining is still feasible.

However, one positive thing found during research is that 77.6% of the energy used by Bitcoin mining is renewable energy [32]. Their figures were based off locations of Bitcoin miners and renewable energy availability in China, where a large majority of Bitcoin mining operations are located. This would make Bitcoin greener than most industries in the world, if the figures are accurate. An increase in eco-friendly Bitcoin mining setups would be a significant contribution to lowering global emissions.

## Chapter 7 – Conclusions and Further Research

The aim of this project, being the FPGA implementation of LED, was successfully achieved within the project timeframe. The LED Block Cipher was successfully implemented in VHDL, making four different versions for comparison purposes, and the program was successfully implemented on the Zynq ARM processor.

### 7.1 Improvements and Further Research

There were certain aspects of the project that could have gone smoother, and other ideas and designs that did not have time to be completed. One of these ideas, and one of the initial plans for the project, was to have a direct connection between the Zynq processors cores and its FPGA. One project that was found while researching was a version of the AES Lightweight Block Cipher that interfaced the cores and the FPGA with the use of a Linux kernel [33]. The Linux kernel sends the data to the FPGA, where the data is then encrypted and sent back to the kernel. However, developing the Linux kernel would have taken too much time, and not a lot of information about running Linux on the Zybo board was available online.

Another idea for improving the code came from another project found online. This project implemented the TwoFish encryption algorithm on the Zynq ARM processor using the Vivado IP Integrator and the Xilinx SDK [34]. It went further into the development for the Zybo by having the program read the input and key from a file on an SD card and writing the output to a different file.

A large amount of time was spent researching how to upload code onto the Zybo board, specifically targeting the Zynq processor. Thanks to the YouTube tutorials mentioned in the Implementation section the primary goal of this project was achieved. However, if the project had been given these as a starting point, more time and effort could have been done to optimise the VHDL code and work on a better version of the code to run on the Zybo board.



# References

- [1] J. Lyons, “Practical Cryptography - Caesar Cipher,” [Online]. Available: <http://practicalcryptography.com/ciphers/caesar-cipher/>. [Accessed 26 March 2019].
- [2] L. A. Dale Liu, “Stream Ciphers,” 2009. [Online]. Available: <https://www.sciencedirect.com/topics/computer-science/stream-ciphers>.
- [3] M. Rouse, “Block Cipher,” January 2006. [Online]. Available: <https://searchsecurity.techtarget.com/definition/block-cipher>.
- [4] K. D. Howard Poston, “Block Ciphers and Modes of Operation,” 2018. [Online]. Available: <https://www.commonlounge.com/discussion/6747358d828a45c99f61f4c09ff2f371>.
- [5] S. M. Masanobu Katagi, “Lightweight Cryptography for the Internet of Things,” Sony Corporation, 2012.
- [6] T. T. Bart Preneel, “Cryptographic Hardware and Embedded Systems - CHES 2011,” in *The 13th International Workshop on Cryptographic Hardware and Embedded*, Todai-ji Cultural Center, Nara, Japan, 2011.
- [7] D. H. Sarah Harris, *Digital Design and Computer Architecture: ARM Edition 1st Edition*, Morgan Kaufmann, 2015.
- [8] X. Wang, “VHDL Introduction (slides 1-27) File,” 2018.
- [9] Xilinx Inc, “Vivado Design Suite HLx Editions,” 2015. [Online]. Available: <https://www.xilinx.com/support/documentation/backgrounders/vivado-hlx.pdf>. [Accessed 1 April 2019].
- [10] L. K. G. L. C. P. A. P. M. R. Y. S. d. C. V. A. Bogdanov, “PRESENT: An Ultra-Lightweight Block Cipher,” 2007.
- [11] L. C. W. Wade Trappe, “Diffusion and Confusion,” in *Introduction to Cryptography with Coding theory*, 2006, p. 38.

- [12] Xilinx Inc, “Field Programmable Gate Array (FPGA),” [Online]. Available: <https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html>. [Accessed 29 March 2019].
- [13] Xilinx Inc, “Xilinx Software Development Kit (XSDK),” [Online]. Available: <https://www.xilinx.com/products/design-tools/embedded-software/sdk.html>. [Accessed 28 March 2019].
- [14] Xilinx, “Zynq-7000 SoC Data Sheet: Overview,” 2 July 2018. [Online]. Available: [https://www.xilinx.com/support/documentation/data\\_sheets/ds190-Zynq-7000-Overview.pdf](https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf).
- [15] I. N. Torsvik, “SoC FPGA Evaluation Guidelines,” [Online]. Available: <https://www.datarespons.com/soc-fpga-evaluation-guidelines/>. [Accessed 29 March 2019].
- [16] V. R. Joan Daemen, “AES Proposal: Rijndael,” 2001.
- [17] D. O. Lawlor, “Groups, Fields, and AES,” 2013. [Online]. Available: [https://www.cs.uaf.edu/2013/spring/cs463/lecture/02\\_11\\_groups\\_fields.html](https://www.cs.uaf.edu/2013/spring/cs463/lecture/02_11_groups_fields.html). [Accessed 28 March 2019].
- [18] M. Rosulek, “Block Cipher Modes of Operation,” in *The Joy of Cryptography*, Oregon State University, 2019, pp. 135-136.
- [19] “<stdint> (stdint.h),” 2011. [Online]. Available: <http://www.cplusplus.com/reference/cstdint/>. [Accessed 28 March 2019].
- [20] “std::Vector,” 2011. [Online]. Available: <http://www.cplusplus.com/reference/vector/vector/>. [Accessed 28 March 2019].
- [21] T. P. A. P. M. R. Jian Guo, “The LED Block Cipher,” [Online]. Available: <https://sites.google.com/site/ledblockcipher/>. [Accessed 2019 March 28].
- [22] “Functions,” 15 October 2009. [Online]. Available: <https://www.ics.uci.edu/~jmoorkan/vhdlref/function.html>. [Accessed 29 March 2019].

- [23] “Component Instantiation,” [Online]. Available: <https://www.ics.uci.edu/~jmoorkan/vhdlref/compinst.html>. [Accessed 30 March 2019].
- [24] “Tutorials for Hardware and Software Co-design on Zybo Development Board,” 2017.
- [25] ENGRTUTOR, “Vivado 2015.2 CUSTOM IP PART I - Creating and Packaging Your IP Vivado,” 2015.
- [26] ENGRTUTOR, *Vivado 2015.2 CUSTOM IP - PART II Creating Vivado Design with Custom IP*, 2015.
- [27] ENGRTUTOR, *Vivado 2015.2 CUSTOM IP PART III - Creating Software for your custom IP Xilinx SDK*, 2015.
- [28] ENGRTUTOR, *Vivado 2015.2 CUSTIOM IP PART IV - Editing your Custom IP Vivado*, 2015.
- [29] b. krebs, “Facebook Stored Hundreds of Millions of User Passwords in Plain Text for Years,” 21 March 2019. [Online]. Available: <https://krebsonsecurity.com/2019/03/facebook-stored-hundreds-of-millions-of-user-passwords-in-plain-text-for-years/>. [Accessed 3 April 2019].
- [30] P. Agrawal, “Keeping your account secure,” 3 May 2018. [Online]. Available: [https://blog.twitter.com/official/en\\_us/topics/company/2018/keeping-your-account-secure.html](https://blog.twitter.com/official/en_us/topics/company/2018/keeping-your-account-secure.html). [Accessed 3 April 2019].
- [31] A. d. Vries, “Bitcoin's Growing Energy Problem,” *Joule*, vol. 2, pp. 801 - 805, 2018.
- [32] CoinShares Research, “The Bitcoin Mining Network - Trends, Composition, Marginal Creation Cost, Electricity Consumption & Sources,” CoinShares Research, 2018.
- [33] L. Vösandi, “Accelerating AES cipher on Zynq SoC-s,” 10 August 2014. [Online]. Available: <https://lauri.xn--vsandi-pxa.com/tub/aep/applied-embedded-systems-project-report.html>. [Accessed 4 April 2019].
- [34] bdimciu, “Twofish Encryption Algorithm on ZYBO,” 2017. [Online]. Available: <https://www.instructables.com/member/bdimciu/>. [Accessed 6th April 2019].

[35] Digilent, “Zybo Z7 Board Reference Manual,” 21 February 2018. [Online]. Available: [https://reference.digilentinc.com/\\_media/reference/programmable-logic/zybo-z7/zybo-z7\\_rm.pdf](https://reference.digilentinc.com/_media/reference/programmable-logic/zybo-z7/zybo-z7_rm.pdf).

## Appendix A – Encryption Example

Round No.	Step No.	Function	Message in to Stage	Message out of Stage
0	0	addRoundKey	0x0123456789abcde	0x0000000000000000
1	0	addConstants	0x0000000000000000	0x4000510020003100
1	0	subCells	0x4000510020003100	0x9ccc05cc6cccb5cc
1	0	shiftRows	0x9ccc05cc6cccb5cc	0x9ccc5cc0cc6ccb5c
1	0	mixColumns	0x9ccc5cc0cc6ccb5c	0x77f5e24d84bae15e
2	0	addConstants	0x77f5e24d84bae15e	0x37f5b14da4bad25e
2	0	subCells	0x37f5b14da4bad25e	0xbd208597f98f7601
2	0	shiftRows	0xbd208597f98f7601	0xbd2059788ff91760
2	0	mixColumns	0xbd2059788ff91760	0xebe9263ec76c56fd
3	0	addConstants	0xebe9263ec76c56fd	0xabe9713ee76c61fd
3	0	subCells	0xabe9713ee76c61fd	0xf81ed5b11da4a527
3	0	shiftRows	0xf81ed5b11da4a527	0xf81e5b1da41d7a52
3	0	mixColumns	0xf81e5b1da41d7a52	0x52dd9d612448fde4
4	0	addConstants	0x52dd9d612448fde4	0x13ddca610548cae4
4	0	subCells	0x13ddca610548cae4	0x5b774fa5c0934f19
4	0	shiftRows	0x5b774fa5c0934f19	0x5b77fa5493c094f1
4	0	mixColumns	0x5b77fa5493c094f1	0x8ec92ad0c0805afd
4	0	addRoundKey	0x8ec92ad0c0805afd	0x8fea6fb7492b9712
5	1	addConstants	0x8fea6fb7492b9712	0xccea38b76a2ba012
5	1	subCells	0xccea38b76a2ba012	0x441fb38daf68fc56
5	1	shiftRows	0x441fb38daf68fc56	0x441f38db68af6fc5
5	1	mixColumns	0x441f38db68af6fc5	0x055569a512fc59a3
6	1	addConstants	0x055569a512fc59a3	0x42553fa535fc6fa3
6	1	subCells	0x42553fa535fc6fa3	0x9600b2f0b024a2fb
6	1	shiftRows	0x9600b2f0b024a2fb	0x96002f0b24b0ba2f
6	1	mixColumns	0x96002f0b24b0ba2f	0x1b16dfdb81361a66
7	1	addConstants	0x1b16dfdb81361a66	0x5c168adba6362f66
7	1	subCells	0x5c168adba6362f66	0x045a3f78faba62aa
7	1	shiftRows	0x045a3f78faba62aa	0x045af783bafaa62a
7	1	mixColumns	0x045af783bafaa62a	0xdf6dc4182d5ec56f
8	1	addConstants	0xdf6dc4182d5ec56f	0x986d97180a5ef66f
8	1	subCells	0x986d97180a5ef66f	0xe3a7ed53cf012aa2
8	1	shiftRows	0xe3a7ed53cf012aa2	0xe3a7d53e01cf22aa
8	1	mixColumns	0xe3a7d53e01cf22aa	0x4f1bdf50367636b9
8	1	addRoundKey	0x4f1bdf50367636b9	0x4e389a37bfddfb56
9	2	addConstants	0x4e389a37bfddfb56	0x0838cd3799ddcc56
9	2	subCells	0x0838cd3799ddcc56	0xc3b347bdee77440a
9	2	shiftRows	0xc3b347bdee77440a	0xc3b37bd477eea440
9	2	mixColumns	0xc3b37bd477eea440	0xb107a7737775c23b
10	2	addConstants	0xb107a7737775c23b	0xf407f0735275f53b
10	2	subCells	0xf407f0735275f53b	0x29cd2cdb06d020b8
10	2	shiftRows	0x29cd2cdb06d020b8	0x29cdcdb2d006820b
10	2	mixColumns	0x29cdcdb2d006820b	0xebea405ce75cfde0

Round No.	Step No.	Function	Message in to Stage	Message out of Stage
11	2	addConstants	0xebea405ce75cfde0	0xa8ea165cc45ccbe0
11	2	subCells	0xa8ea165cc45ccbe0	0xf31f5a044904481c
11	2	shiftRows	0xf31f5a044904481c	0xf31fa0450449c481
11	2	mixColumns	0xf31fa0450449c481	0x8cbf671132cc7542
12	2	addConstants	0x8cbf671132cc7542	0xcbbf331115cc4142
12	2	subCells	0xcbbf331115cc4142	0x4882bb5550449596
12	2	shiftRows	0x4882bb5550449596	0x4882b55b44506959
12	2	mixColumns	0x4882b55b44506959	0xca3295eff7b0bcd
12	2	addRoundKey	0xca3295eff7b0bcd	0xcb11d0887e1b7102
13	3	addConstants	0xcb11d0887e1b7102	0x8c118188591b4002
13	3	subCells	0x8c118188591b4002	0x345535330e589cc6
13	3	shiftRows	0x345535330e589cc6	0x3455333580e69cc
13	3	mixColumns	0x3455333580e69cc	0xf2f031a9a9605314
14	3	addConstants	0xf2f031a9a9605314	0xb4f062a98f606014
14	3	subCells	0xb4f062a98f606014	0x892ca6fe32acac59
14	3	shiftRows	0x892ca6fe32acac59	0x892c6fea329ac5
14	3	mixColumns	0x892c6fea329ac5	0x61b1c0040d57f640
15	3	addConstants	0x61b1c0040d57f640	0x25b197042957c140
15	3	subCells	0x25b197042957c140	0x6085edc96e0d459c
15	3	shiftRows	0x6085edc96e0d459c	0x6805dc9e0d6ec459
15	3	mixColumns	0x6805dc9e0d6ec459	0xdd9739fc3d21867c
16	3	addConstants	0xdd9739fc3d21867c	0x9c976ffc1c21b07c
16	3	subCells	0x9c976ffc1c21b07c	0xe4eda22454658cd4
16	3	shiftRows	0xe4eda22454658cd4	0xe4ed224a655448cd
16	3	mixColumns	0xe4ed224a655448cd	0xb88a3de4c5715dc1
16	3	addRoundKey	0xb88a3de4c5715dc1	0xb9a978834cda902e
17	4	addConstants	0xb9a978834cda902e	0xf9a92d836fdaa52e
17	4	subCells	0xf9a92d836fdaa52e	0x2ffe673ba27ff061
17	4	shiftRows	0x2ffe673ba27ff061	0x2ffe73b67fa21f06
17	4	mixColumns	0x2ffe73b67fa21f06	0x3a53c9a3303ea241
18	4	addConstants	0x3a53c9a3303ea241	0x7d539ba3173e9041
18	4	subCells	0x7d539ba3173e9041	0xd70be8fb5db1ec95
18	4	shiftRows	0xd70be8fb5db1ec95	0xd70b8fb5db15d5ec9
18	4	mixColumns	0xd70b8fb5db15d5ec9	0x6dacbe3a6ca484e1
19	4	addConstants	0x6dacbe3a6ca484e1	0x2baceb3a4aa4b1e1
19	4	subCells	0x2baceb3a4aa4b1e1	0x68f418bf9ff98515
19	4	shiftRows	0x68f418bf9ff98515	0x68f48bf1f99f5851
19	4	mixColumns	0x68f48bf1f99f5851	0x4fddbd301e814ffb
20	4	addConstants	0x4fddbd301e814ffb	0x0addee303b817cfb
20	4	subCells	0x0addee303b817cfb	0xcf771bcb835d428
20	4	shiftRows	0xcf771bcb835d428	0xcf771bc135b88d42
20	4	mixColumns	0xcf771bc135b88d42	0x11e96499a004c8c8
20	4	addRoundKey	0x11e96499a004c8c8	0x10ca21fe29af0527

Round No.	Step No.	Function	Message in to Stage	Message out of Stage
21	5	addConstants	0x10ca21fe29af0527	0x52ca77fe0baf3327
21	5	subCells	0x52ca77fe0baf3327	0x064fdd21c8f2bb6d
21	5	shiftRows	0x064fdd21c8f2bb6d	0x064fd21df2c8dbb6
21	5	mixColumns	0x064fd21df2c8dbb6	0x98cb6c609b955cba
22	5	addConstants	0x98cb6c609b955cba	0xddcb3860bc9568ba
22	5	subCells	0xddcb3860bc9568ba	0x7748b3ac81e0a38f
22	5	shiftRows	0x7748b3ac81e0a38f	0x77483acbe081fa38
22	5	mixColumns	0x77483acbe081fa38	0xe2ac0dc3d791ebce
23	5	addConstants	0xe2ac0dc3d791ebce	0xa1ac5dc3f491dbce
23	5	subCells	0xa1ac5dc3f491dbce	0xf5f4074b29e57841
23	5	shiftRows	0xf5f4074b29e57841	0xf5f474b0e5291784
23	5	mixColumns	0xf5f474b0e5291784	0x375a5616028dd02c
24	5	addConstants	0x375a5616028dd02c	0x715a0616248de02c
24	5	subCells	0x715a0616248de02c	0xd50fca5a69371c64
24	5	shiftRows	0xd50fca5a69371c64	0xd50fa5ac376941c6
24	5	mixColumns	0xd50fa5ac376941c6	0x5ed8fda3f89160e7
24	5	addRoundKey	0x5ed8fda3f89160e7	0x5ffbb8c4713aad08
25	6	addConstants	0x5ffbb8c4713aad08	0x1bfbe9c4553a9c08
25	6	subCells	0x1bfbe9c4553a9c08	0x58281e4900bfe4c3
25	6	shiftRows	0x58281e4900bfe4c3	0x5828e491bf003e4c
25	6	mixColumns	0x5828e491bf003e4c	0xa09c73b4010f89fc
26	6	addConstants	0xa09c73b4010f89fc	0xe09c21b4210fbbfc
26	6	subCells	0xe09c21b4210fbbfc	0x1ce4658965c28824
26	6	shiftRows	0x1ce4658965c28824	0x1ce45896c2654882
26	6	mixColumns	0x1ce45896c2654882	0x2abb702fb72da2d7
27	6	addConstants	0x2abb702fb72da2d7	0x6abb252f972d97d7
27	6	subCells	0x6abb252f972d97d7	0xaf886062ed67ed7d
27	6	shiftRows	0xaf886062ed67ed7d	0xaf88062667edded7
27	6	mixColumns	0xaf88062667edded7	0xbe27acb64514524c
28	6	addConstants	0xbe27acb64514524c	0xff27ffb66414614c
28	6	subCells	0xff27ffb66414614c	0x226d228aa959a594
28	6	shiftRows	0x226d228aa959a594	0x226d28a259a94a59
28	6	mixColumns	0x226d28a259a94a59	0x86c36456cca1cb6f
28	6	addRoundKey	0x86c36456cca1cb6f	0x87e02131450a0680
29	7	addConstants	0x87e02131450a0680	0xc5e07631670a3180
29	7	subCells	0xc5e07631670a3180	0x401cdab5adcfb53c
29	7	shiftRows	0x401cdab5adcfb53c	0x401cab5dcfadcb53
29	7	mixColumns	0x401cab5dcfadcb53	0x93c786c4b8b4a65b
30	7	addConstants	0x93c786c4b8b4a65b	0xd6c7d0c49db4905b
30	7	subCells	0xd6c7d0c49db4905b	0x7a4d7c49e789ec08
30	7	shiftRows	0x7a4d7c49e789ec08	0x7a4dc49789e78ec0
30	7	mixColumns	0x7a4dc49789e78ec0	0x34e88d8bbdd9391c
31	7	addConstants	0x34e88d8bbdd9391c	0x77e8d98b9ed90d1c

Round No.	Step No.	Function	Message In to Stage	Message Out of Stage
31	7	subCells	0x77e8d98b9ed90d1c	0xdd137e38e17ec754
31	7	shiftRows	0xdd137e38e17ec754	0xdd13e3877ee14c75
31	7	mixColumns	0xdd13e3877ee14c75	0x96d335f2c7e37706
32	7	addConstants	0x96d335f2c7e37706	0xd1d365f2e0e34706
32	7	subCells	0xd1d365f2e0e34706	0x757ba0261c1b9dca
32	7	shiftRows	0x757ba0261c1b9dca	0x757b026a1b1ca9dc
32	7	mixColumns	0x757b026a1b1ca9dc	0xa1201079b13831b7
32	7	addRoundKey	0xa1201079b13831b7	0xa003551e3893fc58

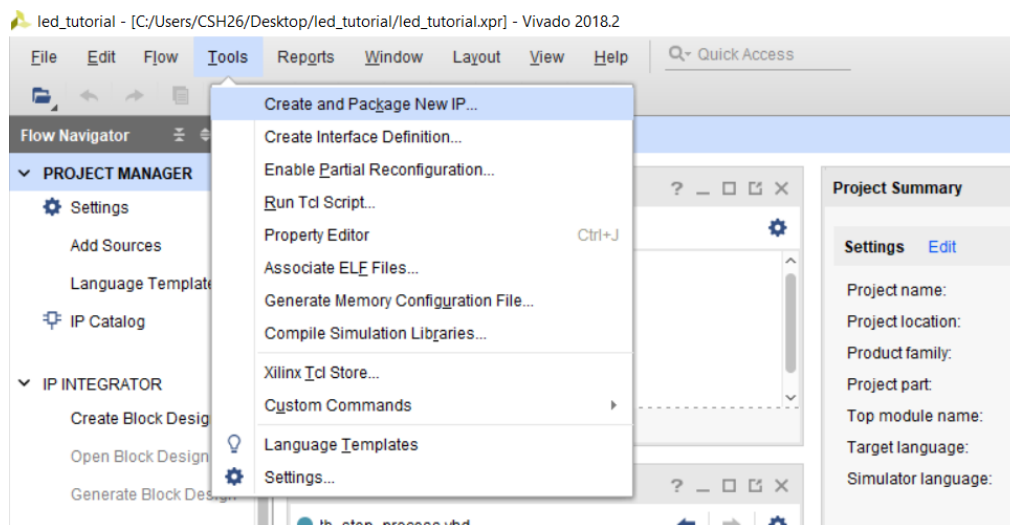
This table shows, with an input and key of 0x0123456789abcdef, the steps going from the input to the output of the LED 64-bit algorithm.



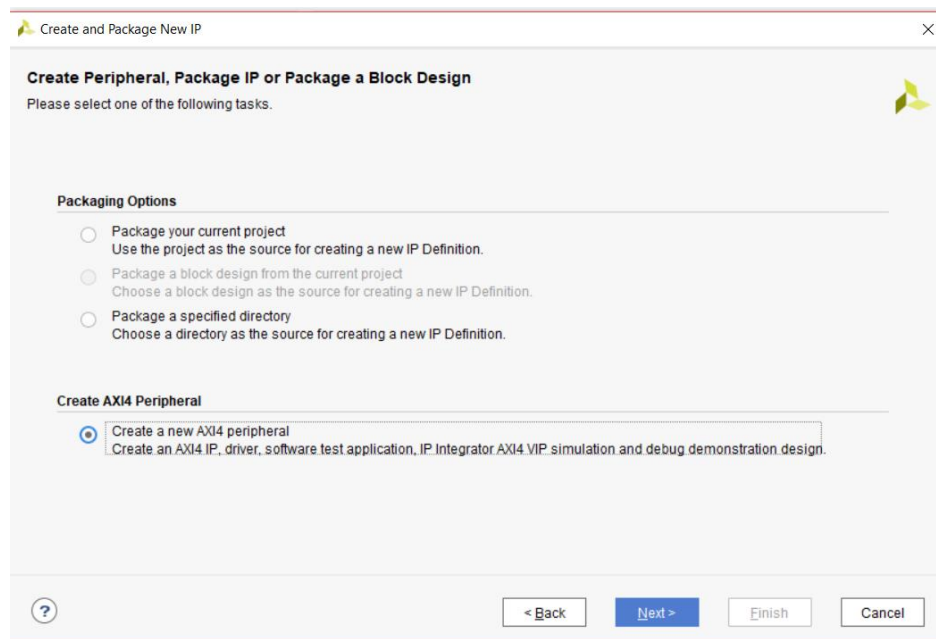
## Appendix B – IP Integrator and SDK Tutorial

Appendix B shows the steps from creating a custom IP from a VHDL program to implementing that program on the Zybo board. This section is aimed to help designers quickly implement their designs on the Zynq ARM processor and will show the steps from creating an IP and a custom block diagram to successfully running the program on the Xilinx SDK.

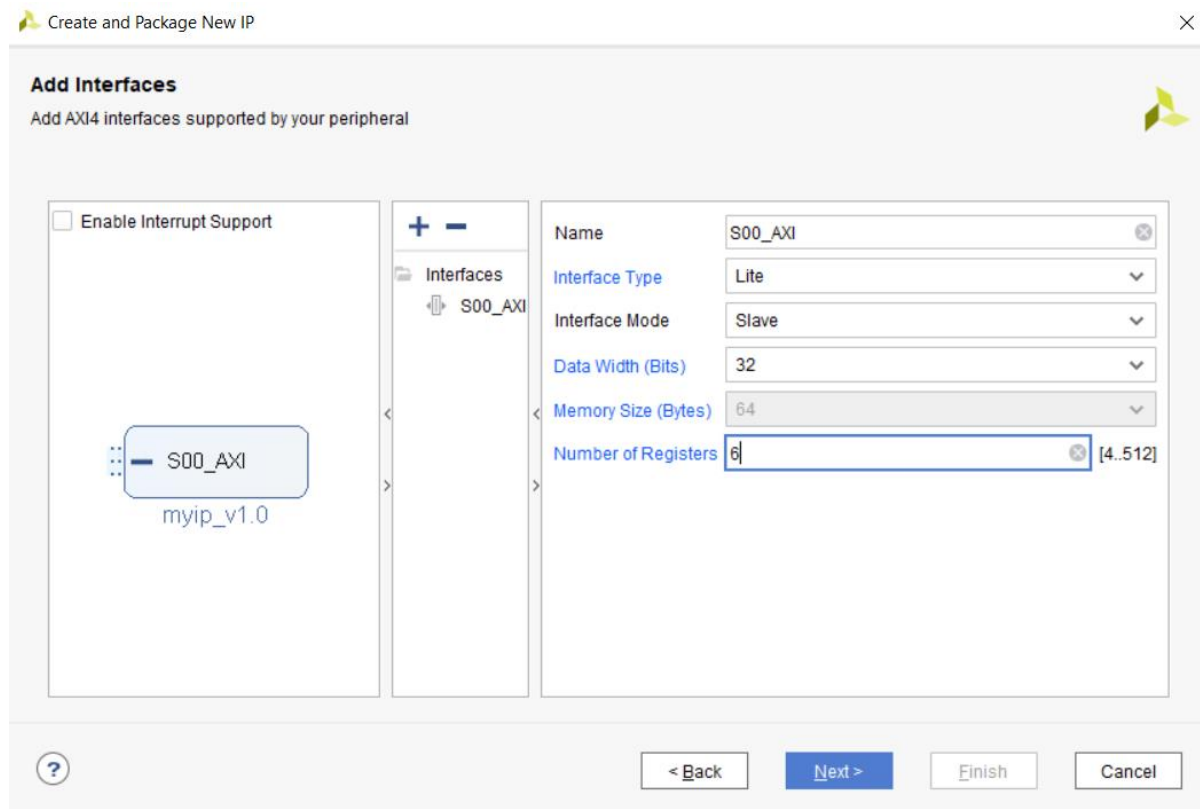
Once a project has been set up, program been made, and the Synthesis has been run, select Tools -> Create and package new IP.



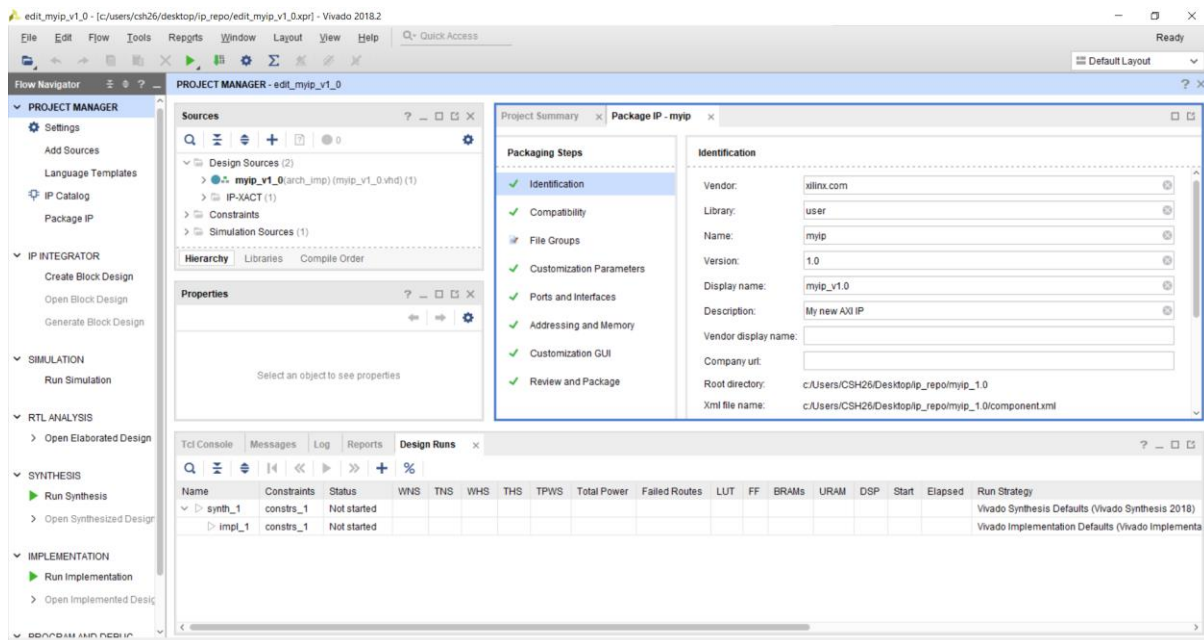
Select next, then select “Create AXI4 Peripheral”, then select next again.



Once the name and description of the IP has been set and next has been selected, the Add Interfaces window will pop up. This screen will decide the number of interfaces and registers the IP will contain. Each Slave register is 32-bits wide, so chose a number based on the number of inputs and outputs required for the program. In this case, 6 registers were chosen, as LED needs 192 bits of input and output ports.

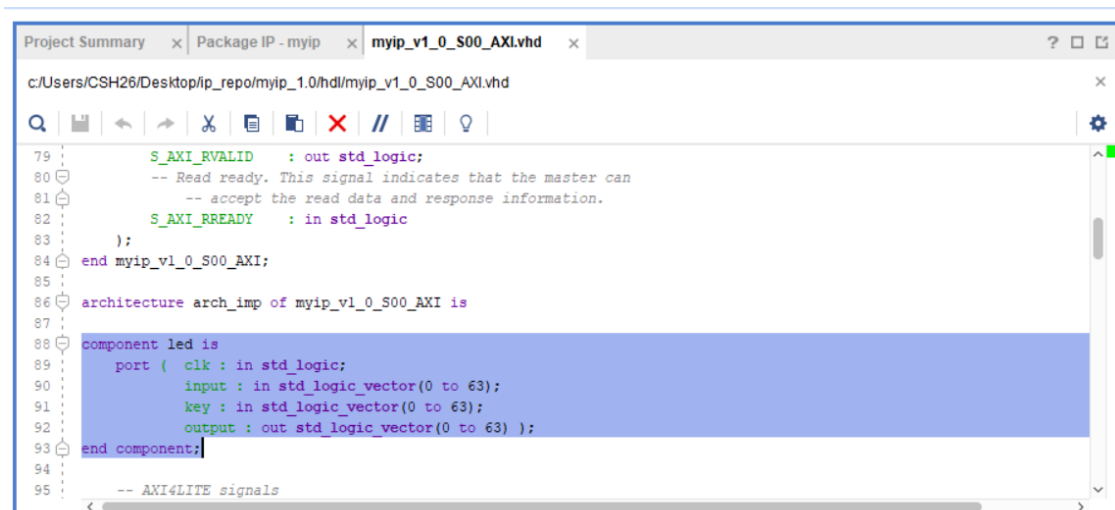


On the next screen, select “edit IP” and select finish. After this, the IP has successfully been created and a new project should pop up.



The next step is to import the VHDL program files to implement on the Zybo board. Select the “+” in the sources tab, locate the files required, and import them into the project.

The next step is to create a component of the file that was imported. Expand the file in the sources tab and open the file called “\*IP name\*\_s00\_AXI\_INST” (The second file in the hierarchy). This file lets the user select what inputs of the IP go to a specified slave address. Scroll until the beginning of the Architecture is found and declare the component.

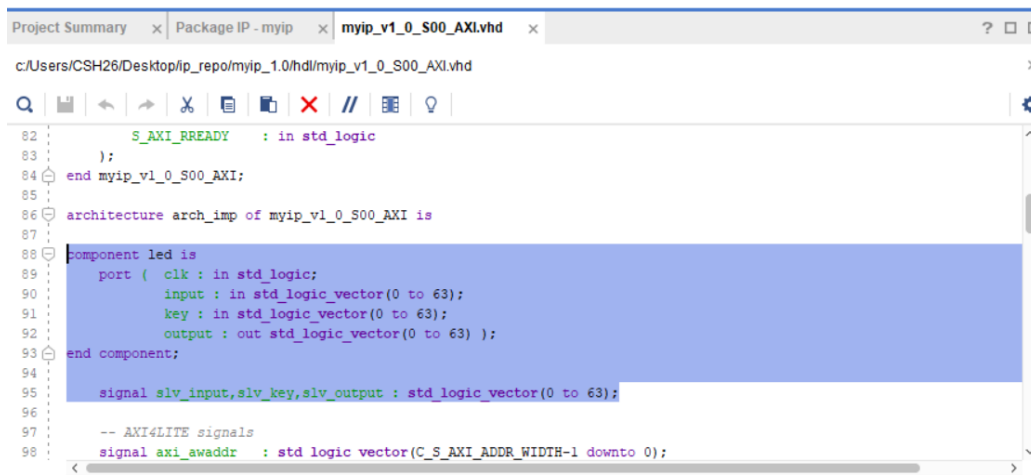


Scroll down to the very end of the code where the commented line “Add user logic here” is found and add the instance of the component created. This is where the inputs and outputs will be assigned to registers.

Above the user logic the local addresses for the slave registers can be found. This is what addresses the inputs and outputs will be assigned. In this case, the registers are smaller than the ports, so signals must be made to assign the ports values. Above the component for the program, create signals for the inputs and outputs. In this case, the signals “slv\_input”, “slv\_output”, and “slv\_key” were create of type std\_logic\_vector(0 to 63). In the user logic section, these signals were assigned to the registers.

Where the slave registers are declared, the output signal needs to replace two of the slave registers. Change one of the registers to slv\_output(0 to 31) and the other to slv\_output(32 to 63). The Clock signal, if there is one in the program, should be set to “S\_AXI\_ACLK” which is an internal AXI clock signal.

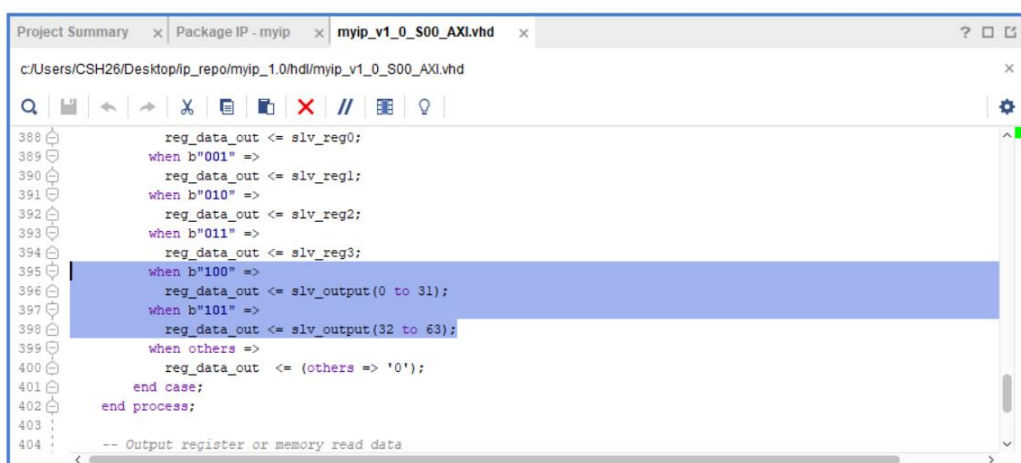
The screenshots below show the necessary changes to the code.



```

82      S_AXI_RREADY : in std_logic
83    );
84  end myip_v1_0_S00_AXI;
85
86  architecture arch_imp of myip_v1_0_S00_AXI is
87
88  component led is
89    port (
90      clk : in std_logic;
91      input : in std_logic_vector(0 to 63);
92      key : in std_logic_vector(0 to 63);
93      output : out std_logic_vector(0 to 63) );
94  end component;
95
96  signal slv_input,slv_key,slv_output : std_logic_vector(0 to 63);
97
98  -- AXI4LITE signals
99  signal axi_awaddr : std_logic_vector(C_S_AXI_ADDR_WIDTH-1 downto 0);

```



```

388      reg_data_out <= slv_reg0;
389      when b"001" =>
390        reg_data_out <= slv_reg1;
391      when b"010" =>
392        reg_data_out <= slv_reg2;
393      when b"011" =>
394        reg_data_out <= slv_reg3;
395      when b"100" =>
396        reg_data_out <= slv_output(0 to 31);
397      when b"101" =>
398        reg_data_out <= slv_output(32 to 63);
399      when others =>
400        reg_data_out <= (others => '0');
401      end case;
402    end process;
403
404    -- Output register or memory read data

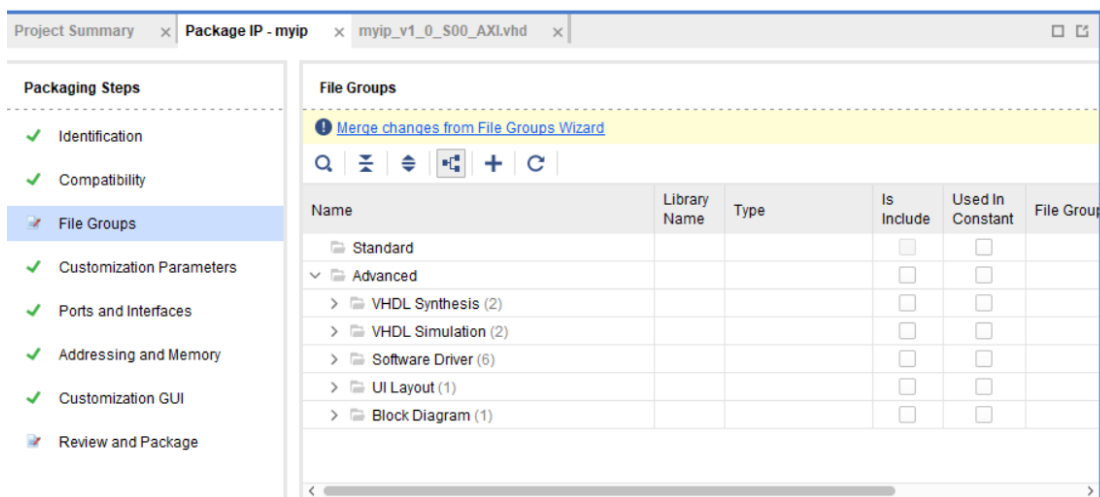
```

```

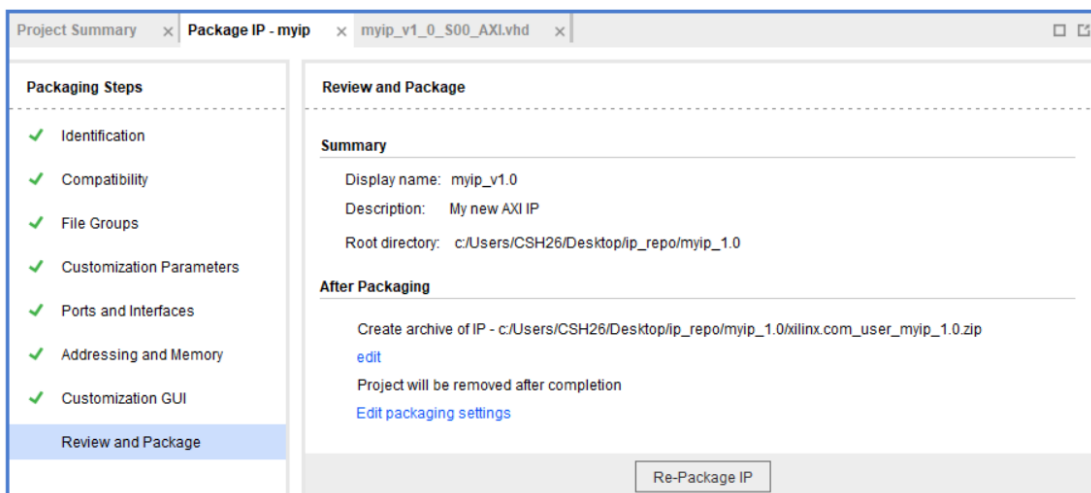
422
423    -- Add user logic here
424
425    led_zybo : led
426    port map(
427        clk => S_AXI_ACLK,
428        input => slv_input,
429        key => slv_key,
430        output => slv_output
431    );
432    slv_input <= slv_reg0 & slv_reg1;
433    slv_key <= slv_reg2 & slv_reg3;
434
435    -- User logic ends
436
437 end arch_imp;
438

```

Once these changes have been made, return to the Package IP tab and select “File Groups” and “Merge Changes from File Groups Wizard”

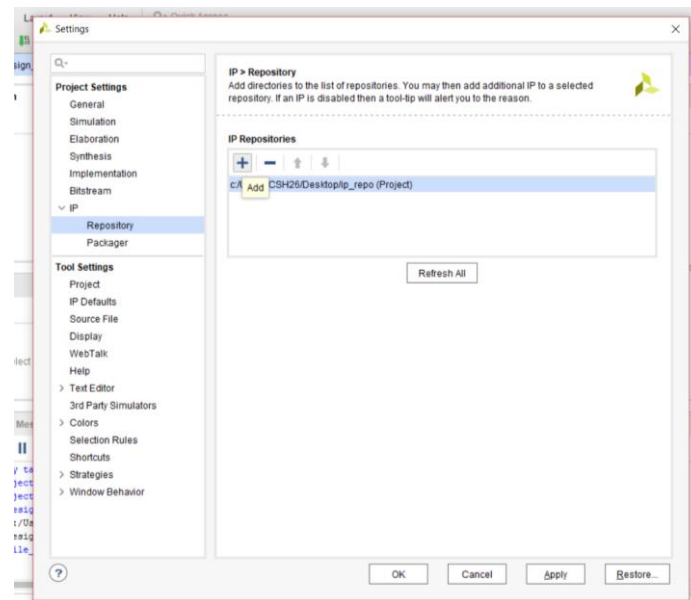
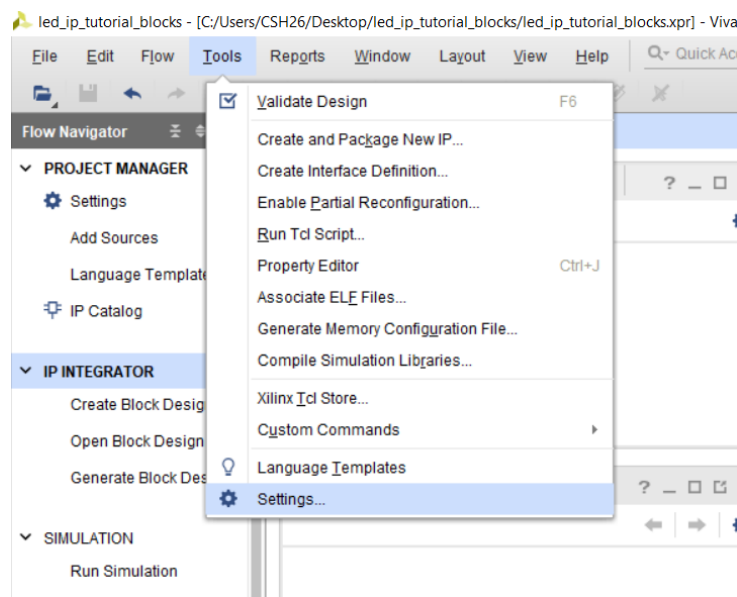


After this, select “Review and Package” and select “Re-Package IP” on the bottom.

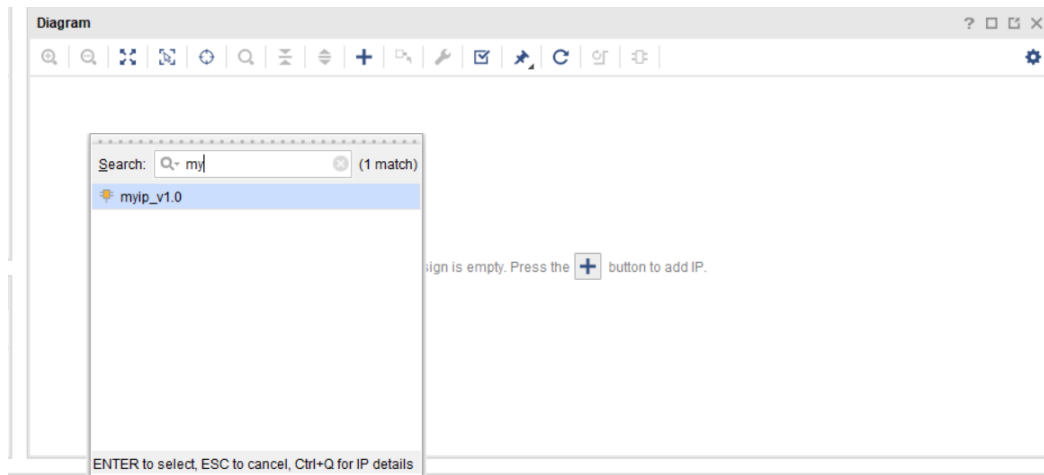


A dialogue box asking to close the project will appear. Select “yes”, and the IP creation portion will be complete.

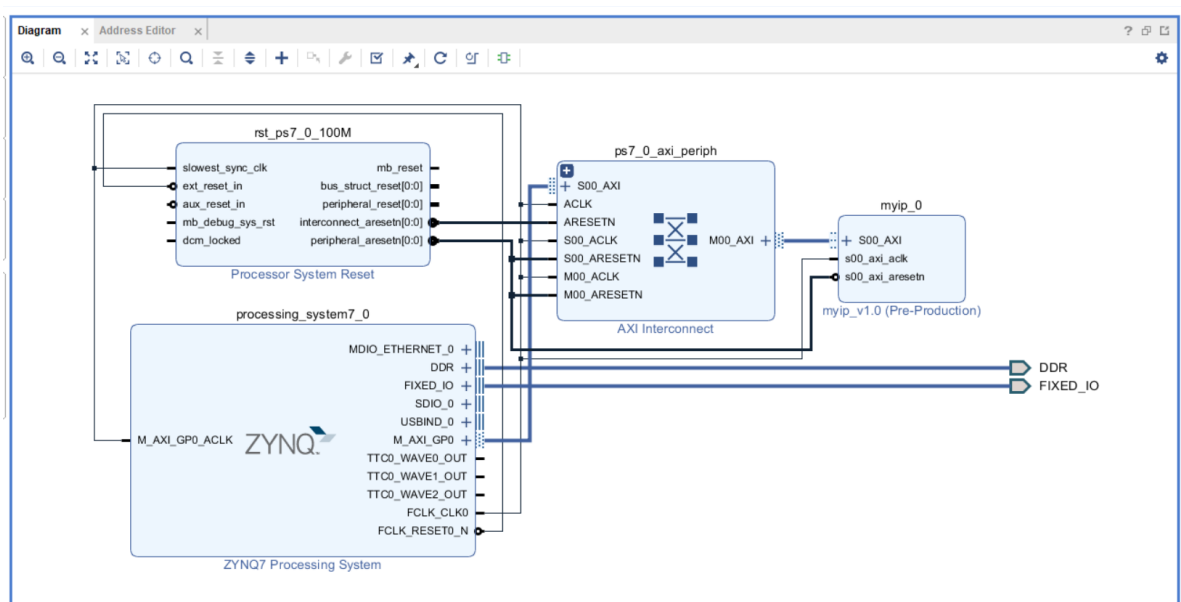
Next, a new project must be created, will be for the block diagram. Create a new project, and once it has been created select the “Create Block Diagram” option on the left of the screen. The IP just created must be added to the IP Repo. Select tools -> settings -> IP -> Repository and select the “+” button. Find the correct IP Repo folder and hit select, and the IP should be available in the project.



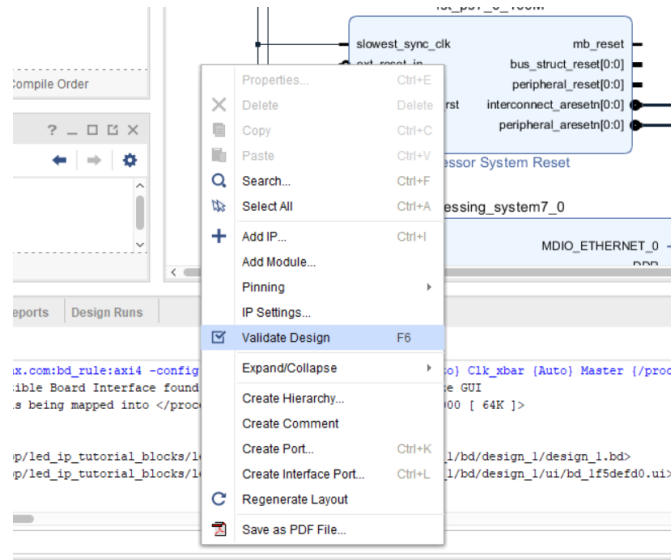
Go back to the block diagram, select the “+” button in the centre and search for the custom IP. In this case the IP was named “myip\_v1.0”. Double click to add to the project.



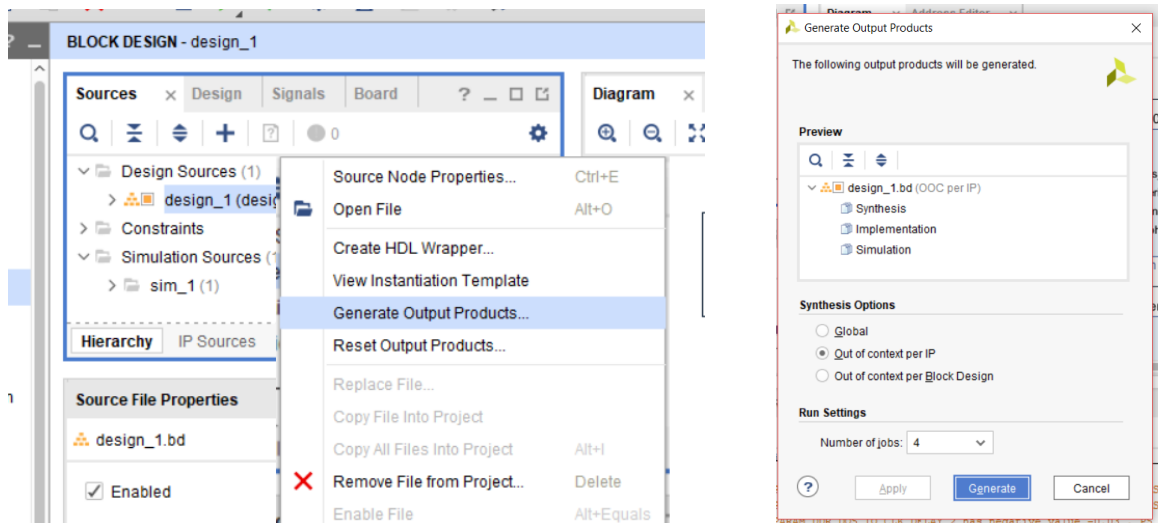
Next, search for the Zynq processor and add it to the diagram. A header will appear on the top of the diagram saying to run the block and connection automations. Select these options and press ok to both. The block diagram should now look like the figure below.



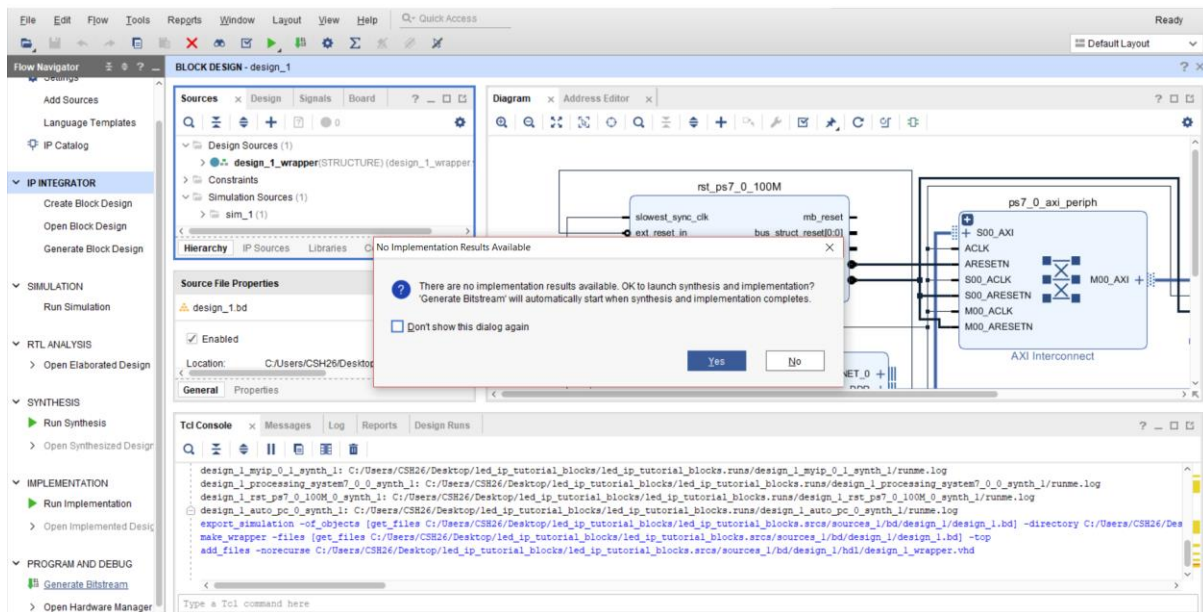
Right click on the diagram and select “Validate Design”. A notification saying the design was successfully validated should appear.



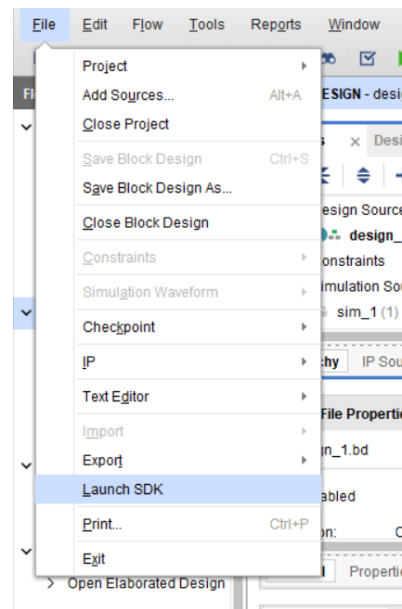
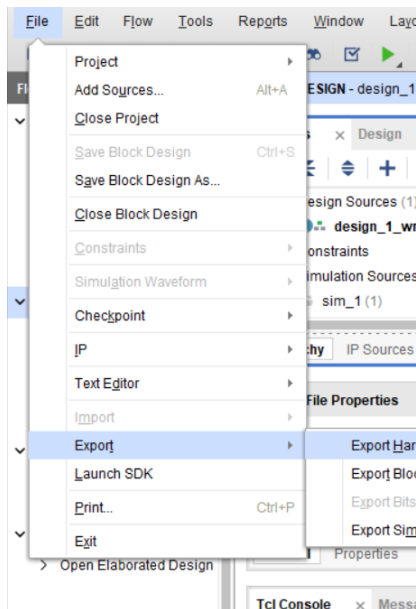
Once this is complete, right click the design in the source tab and select “Generate Output Ports.” A dialogue box will appear, select generate. This may take a little bit of time depending on the program.





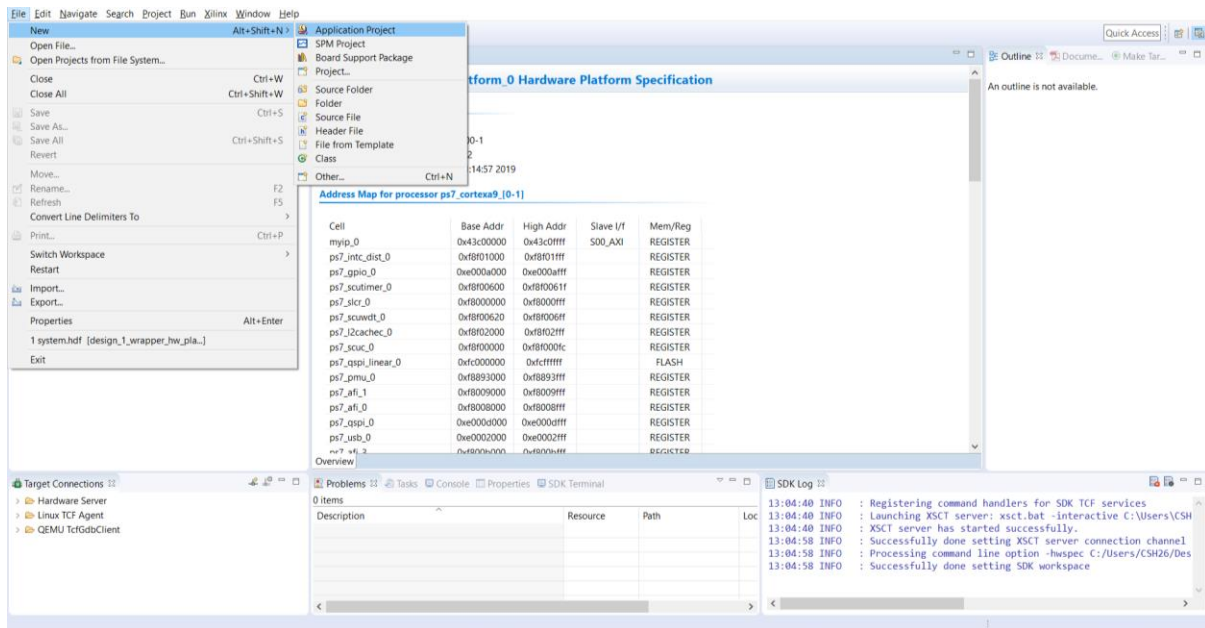


Next, select “Generate Bitstream” from the left had pane and select yes to the dialogue box. This process may take some time (this took 13 minutes just to run the implementation). Once that has completed, right click in the sources tab again and select “Create HDL Wrapper” and select ok. Once this is complete, go to file -> export -> export hardware (make sure include bitstream is checked), then go to file -> Launch SDK.

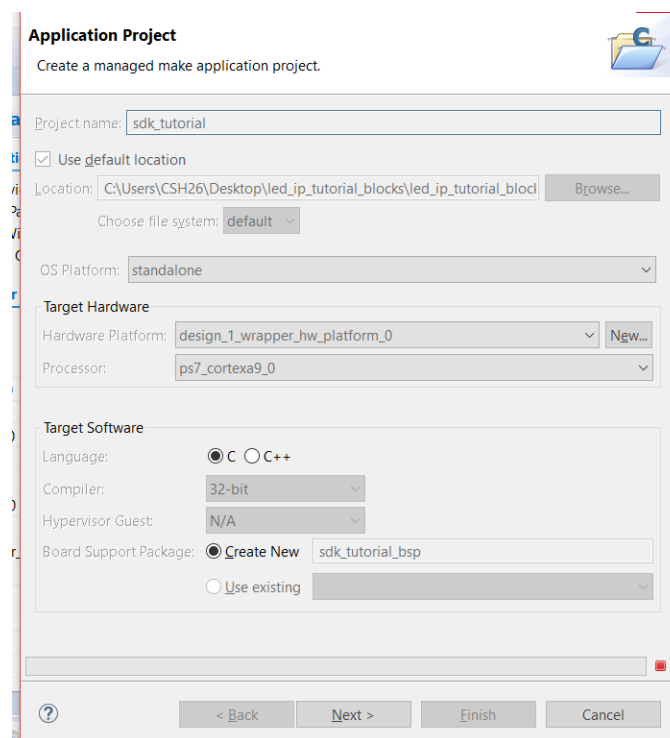


This concludes the Vivado section. The SDK should have launched one the “Launch SDK” option has been pressed.

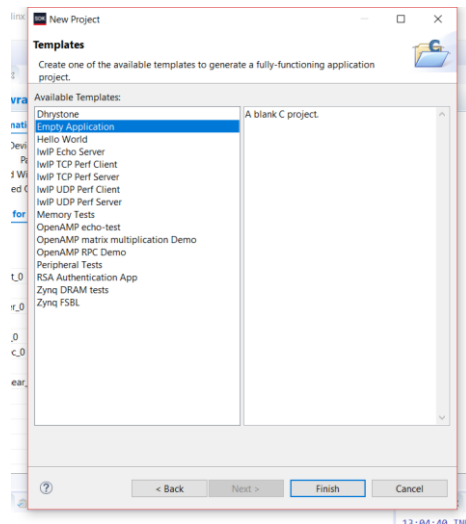
Once the SDK is launched, select file -> new -> application project.



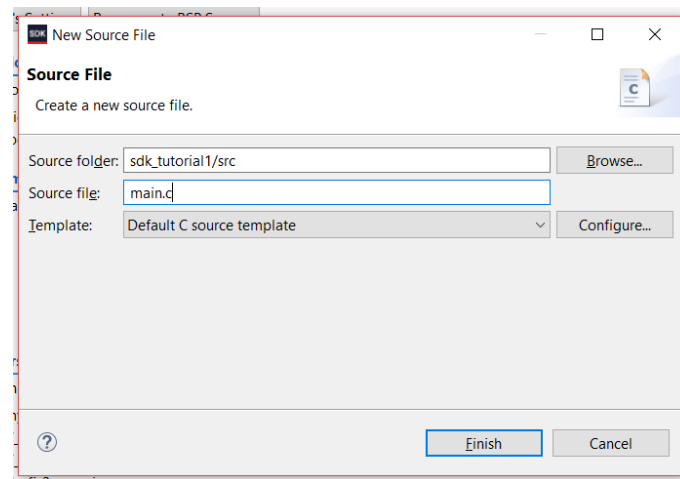
Make sure the hardware platform is set to “design\_1\_wrapper\_hw\_platform\_0” and select .



Then, select “empty application” and select finish.



Next the C code will be programmed. Right click on the src folder in the newly created project and select new -> source file. Name the file with the “.c” extension and select finish.



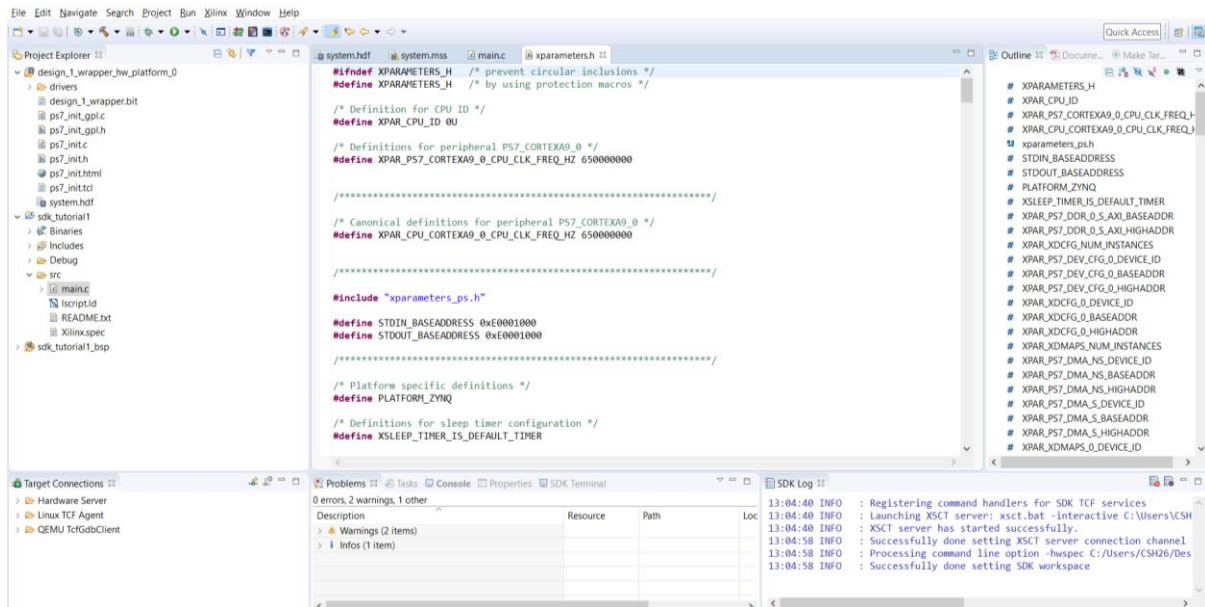
Open the file and include the following libraries:

```
#include "xparameters.h"
```

```
#include "xil_io.h"
```

```
#include "xil_types.h"
```

Create an “int main(){}” method and the programming can begin. Open the “xparameters.h” file by CTRL + left clicking on the text. This will open the file.



Using CTRL + f, search for the name of the file. There you will find the base address of the registers. Each register is the base address + 4 in the order that they were assigned. The base address in this case is “0x43C00000” and is defined as:

“XPAR\_MYIP\_0\_S00\_AXI\_BASEADDR”.

```

system.hdf system.mss main.c xparameters.h
#define XPAR_PS7_GPIO_0_DEVICE_ID 0
#define XPAR_PS7_GPIO_0_BASEADDR 0xE000A000
#define XPAR_PS7_GPIO_0_HIGHADDR 0xE000AFFF

/* Canonical definitions for peripheral PS7_GPIO_0 */
#define XPAR_XGPIOPS_0_DEVICE_ID XPAR_PS7_GPIO_0_DEVICE_ID
#define XPAR_XGPIOPS_0_BASEADDR 0xE000A000
#define XPAR_XGPIOPS_0_HIGHADDR 0xE000AFFF

/* Definitions for driver MYIP */
#define XPAR_MYIP_NUM_INSTANCES 1

/* Definitions for peripheral MYIP_0 */
#define XPAR_MYIP_0_DEVICE_ID 0
#define XPAR_MYIP_0_S00_AXI_BASEADDR 0x43C00000
#define XPAR_MYIP_0_S00_AXI_HIGHADDR 0x43C0FFFF

/* Definitions for driver QSPIPS */
#define XPAR_XQSPIPS_NUM_INSTANCES 1

/* Definitions for peripheral PS7_QSPI_0 */

```

Using this address, a number will be written to the input and keys (in this case). Using the method “Xil\_Out32”. The following code writes the number 0x0123456789ABCDEF to the input and key:

```

//input
Xil_Out32(XPAR_MYIP_0_S00_AXI_BASEADDR,0x01234567);
Xil_Out32(XPAR_MYIP_0_S00_AXI_BASEADDR+4,0x89abcdef);
//key
Xil_Out32(XPAR_MYIP_0_S00_AXI_BASEADDR+8,0x01234567);

```

```
Xil_Out32(XPAR_MYIP_0_S00_AXI_BASEADDR+12,0x89abcdef);
```

To see if the code worked, the output is going to be sent to a terminal. To do this, a variable must be created to view this output. The following code creates a variable and stores the output:

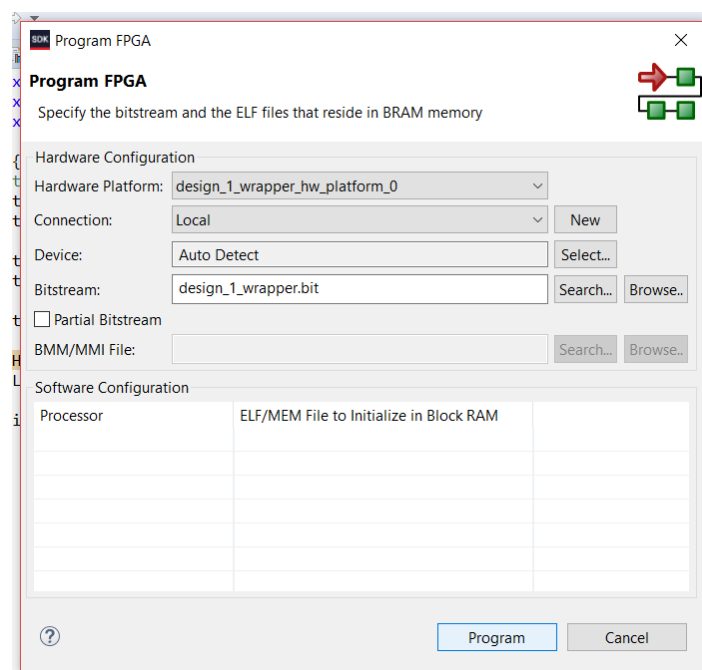
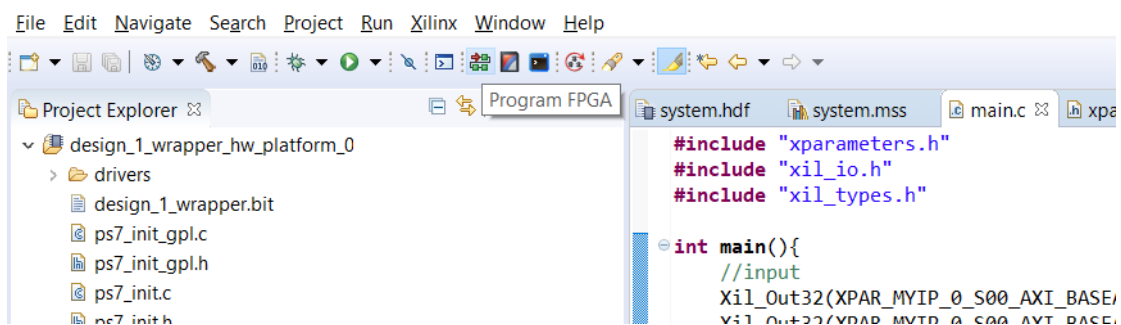
```
u32 outputLow,outputHigh;
```

```
outputHigh = Xil_In32(XPAR_MYIP_0_S00_AXI_BASEADDR+16);
outputLow = Xil_In32(XPAR_MYIP_0_S00_AXI_BASEADDR+20);
```

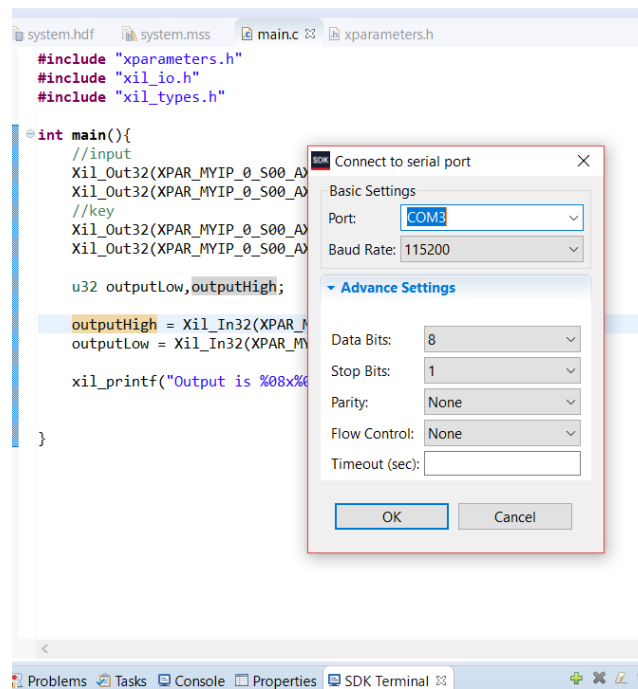
To view these values, the addresses must be printed to the terminal. The following code prints these variables:

```
xil_printf("Output is %08x%08x\n",outputHigh,outputLow);
```

The FPGA can now be programmed. Click on the “program FPGA” icon on the top left of the screen. Make sure the correct hardware platform and bitstream are selected and click program.



Next, go to the bottom of the screen and select the SDK terminal. Click on the “+” icon to create a new terminal. Select the options in the figure below and select ok. If it doesn’t work first time after debugging, try a different port.



Next, right click on the project and select run as -> 1. Launch on hardware (system debugger). Click on the terminal and if the output is correct, the Zynq processor has been successfully programmed.

