University of Texas Rio Grande Valley

ScholarWorks @ UTRGV

Theses and Dissertations

12-2017

DES and TDES Performance Evaluation for Non-pipelined and Pipelined Implementations in VHDL Using the Cyclone II FPGA Technology

Edni Del Rosal
The University of Texas Rio Grande Valley

Follow this and additional works at: https://scholarworks.utrgv.edu/etd

Part of the Electrical and Computer Engineering Commons

Recommended Citation

Del Rosal, Edni, "DES and TDES Performance Evaluation for Non-pipelined and Pipelined Implementations in VHDL Using the Cyclone II FPGA Technology" (2017). *Theses and Dissertations*. 139.
https://scholarworks.utrgv.edu/etd/139

This Thesis is brought to you for free and open access by ScholarWorks @ UTRGV. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of ScholarWorks @ UTRGV. For more information, please contact justin.white@utrgv.edu, william.flores01@utrgv.edu.

DES AND TDES PERFORMANCE EVALUATION FOR NON-PIPELINED AND PIPELINED IMPLEMENTATIONS IN VHDL USING THE CYCLONE II FPGA TECHNOLOGY

A Thesis

by

EDNI DEL ROSAL

Submitted to the Graduate College of
The University of Texas Rio Grande Valley
In partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE ENGINEERING

December 2017

Major Subject: Electrical Engineering

DES AND TDES PERFORMANCE EVALUATION FOR NON-PIPELINED AND PIPELINED IMPLEMENTATIONS IN VHDL USING THE CYCLONE II FPGA TECHNOLOGY

A Thesis
by
EDNI DEL ROSAL

COMMITTEE MEMBERS

Dr. Sanjeev Kumar
Chair Of Committee

Dr. Weidong Kuang
Committee Member

Dr. Jun Peng
Committee Member

December 2017

Copyright 2017 Edni Del Rosal

All Rights Reserved

ABSTRACT

Del Rosal, Edni, DES and TDES Performance Evaluation For Non-Pipelined and Pipelined Implementations in VHDL Using The Cyclone II FPGA Technology, Master of Science (MS), December, 2017, 177 pp., 14 tables, 100 figures, references, 65 titles.

Two ongoing issues that engineers must face in the new era of data analytics are performance and security. Field Programmable Gate Arrays (FPGAs) offer a new solution for optimizing the performance of applications while the Data Encryption Standard (DES) and the Triple Data Encryption Standard (TDES) offer a mean to secure information. In this thesis we present a Non-Pipelined and Pipelined, in Electronic Code Book (EBC) mode, implementations in VHDL of these two commonly utilized cryptography schemes. Using Altera Cyclone II FPGA as our platform, we design and verify the implementations with the EDA tools provided by Altera. We gather cost and throughput information from the synthesis and timing results and compare the cost and performance of our designs to those presented in other literatures. Our designs achieve a throughput of 3.2 Gbps with a 50 MHz clock and our cost triples from DES to TDES.

DEDICATION

The completion of this life goal would not be possible without everyone who has greatly impacted my life. My family has been a blessing of support and uplifts every step of the way. Without their love, I would never have dared to shoot for the moon. Close friends and colleagues (too many to mention) who have witnessed the long days and nights of struggle, you have my outmost appreciation from the bottom of my heart. Siblings, this is for you mainly. You will find no greater satisfaction than working your most to accomplish your goals and dreams. Dr. Kumar, your guidance, knowledge, feedback and life has been a tremendous inspiration. Thank you for your time invested in me. My parents and I are grateful for you. My best wishes for you and your family.

ACKNOWLEDGMENTS

I would like to formally thank:

Dr. Sanjeev Kumar (advisor and chair of my thesis committee) for bestowing his knowledge on me. Among the students, he is known for his research, knowledge and accessibility. These were the key factors that helped determine my advisor. His networks security courses are utterly relevant in today's world. Through my industry experience, the concepts he discussed were of mayor impact. Once more, thank you for assisting to make this academic goal a reality.

I would like to thank Dr. Kuang and Dr. Peng for their wiliness to serve as committee members. Their input helped greatly to achieve the acceptance of my paper in the Journal of Circuit Systems. Furthermore, their advice, input, and comments on my thesis helped to ensure the quality of my work. I extend my deepest gratitude for their support and guidance.

TABLE OF CONTENTS

Page

ABSTRACT…...……………………………………………………………….………. iii

DEDICATION……….……………….…………………………………………...………. iv

ACKNOWLEDGMENTS………...…………….………………………………...………. v

TABLE OF CONTENTS……………………….……….…………………...………. vi

LIST OF TABLES…………………..………….……………………...….………..….….…. ix

LIST OF FIGURES………………..………………….………………………......….….…. x

CHAPTER I. INTRODUCTION………………..………………………………………. 1

1.1 FPGA History……………………………...…………………………………… 4

1.2 FPGA Business Market Model……………………….…………………………… 4

1.2.1 FPGA Vendors, Technology and Altera's DE2 Board EP2C35F672C6... 5

1.2.2 FPGA Based System Developers and Thesis Objective………..…….. 9

1.2.3 Conceptual Digital Design Flow and EDA Tools……………………..…. 10

1.3 Cryptography………………..…………………………………………………….. 14

1.3.1 Cryptography Current Events And History…………………………..….. 15

1.3.2 DES & TDES History…………………….…….…………………….. 18

1.3.3 Exhaustive Key Search Analysis…………………….……………….. 19

1.4 Survey On Exiting Work And Expected Results……………...…………………... 20

1.5 Thesis Outline……………..……………...…………………………………….. 22

CHAPTER II. DES & TDES NON-PIPELINED DESINGS IN VHDL……………..…..….. 23

2.1 DES Non-Pipelined Design For Encryption………...…………………..………… 24

2.1.1 DES Key Schedule Component For Encryption.……………….......…… 24

2.1.2 DES Encryption Initial Permutation, Final Permutation and 16 Rounds.... 30

2.2 DES Non-Pipelined Design For Decryption……………...............................…… 39

2.2.1 DES Key Schedule Component For Decryption.…………………..…… 40

2.2.2 DES Decryption Initial Permutation, Final Permutation and 16 Rounds... 42

2.3 TDES Non-Pipelined Design For Encryption and Decryption…………..……… 43

CHAPTER III. DES & TDES PIPELINED DESINGS IN VHDL………………………….... 46

3.2 DES Pipelined Design For Encryption…………........………………………… 47

3.1.1 DES Key Schedule Component For Encryption.………………………… 47

3.1.2 16 Stages In The DES Encryption Pipeline……………………….….…….. 48

3.2 DES Pipelined Design For Decryption…………........………………………… 50

3.2.1 DES Key Schedule Component For Decryption.…………..….………… 50

3.2.2 16 Stages In The DES Decryption Pipeline …………………………….….. 51

3.3 TDES Pipelined Design For Encryption and Decryption……………………..…… 52

CHAPTER IV. ECB Mode of Operation Validation System for 3DES & DES.……….....….. 57

4.1 Variable plaintext KAT…………........………………………………………… 59

4.2 Inverse Permutation KAT for Encryption …………........……………...……… 61

4.3 Variable key KAT…………........……………………………………………… 63

4.4 Permutation Operation KAT for Encryption.………..…........…………….…… 65

4.5 Substitution table KAT for Encryption.………….........……..……..…………… 67

4.6 Modes Test for encrypt (MONTECARLO)……...………………………………… 69

CHAPTER V. DES AND TDES NON-PIPELINED VS PIPELINED PERFORMANCE EVALUATION………..……....................…………......…………………………….….. 74

5.1 Hardware Setup and EDA Tools……………..………………………….…………… 74

5.2 Parameters of Performance Evaluation………...….........………………………..……… 76

5.3 Results and Discussion……………........……………………………………… 76

5.3.1 DES Encryption and Decryption Non Pipelined.…………..……………..… 77

5.3.2 TDES Encryption and Decryption Non Pipelined.…………..……...…… 90

5.3.3 DES Pipelined Encryption and Decryption.…………..……………..…… 99

5.3.4 TDES Pipelined Encryption and Decryption.………..…..………………… 108

5.3.5 DES and TDES Cost and Performance Comparison.…………….…..…… 117

5.3.6 Quartus II and Altera U.P. Simulator Observations.….………………...… 119

5.3.7 Performance and Cost Literature Review Comparisons.……………..… 121

CHAPTER VI. CONCLUSIONS AND FUTURE WORK………...…….....................……. 124

REFERENCES…………………………………...…..................…..……….......……….. 129

APPENDIX A……………………………………………...………..……........……….. 135

BIOGRAPHICAL SKETCH……………….………………...………..……........……….. 191

LIST OF TABLES

Page

Table 1: Comparison of Altera VS Xilinx High-End FPGA………………...…….…………. 6

Table 2: Computation Power And Time For Exhaustive Search ……………………………… 20

Table 3: Cost and Performance Comparison…………………………...…………………… 21

Table 4: Left Rotations…………………………...…………….………………………………… 27

Table 5: Variable Plaintext KAT for Encryption Cipher Results...……………………….……… 60

Table 6: Inverse Permutation KAT for Encryption Cipher Results...……..…………………… 62

Table 7: Variable Key KAT for Encryption Cipher Results...…………………….………… 64

Table 8: Permutation Operation KAT for Encryption Cipher Results...………….………… 66

Table 9: Substitution Table KAT for Encryption Cipher Results...……………..…………...… 67

Table 10: DES & TDES Hardware Cost Comparison...……….……………………...……… 118

Table 11: DES & TDES Delay Analysis and Throughput Comparison...……….………… 119

Table 12: DES & TDES Analysis and Synthesis Times...…………………………….…… 119

Table 13: DES & TDES Validation Simulation Times In Functional Mode...……….……… 120

Table 14: DES & TDES Designs Comparisons...…………………………………..……… 121

LIST OF FIGURES

Page

Figure 1: Microsoft's Prototype For Connecting FPGAs To The Rack and Processing Traffic…...……………………..……………………………………………………..... 2

Figure 2: IBM's FPGA Acceleration Approach via CAPI……...……...……………………….…… 3

Figure 3: IBM's Potential Markets For CAPI……...………………………...……………… 3

Figure 4: FPGA Market Demand and Supply Flow……..……...……...………………………… 5

Figure 5: LE Architecture……..…………………..………………………………….....…….. 7

Figure 6: LE's Interconnects in 1 LAB……...……...………………………..……………… 8

Figure 7: LABs Arranged Into Rows and Columns With Interconnect……...……...…….…… 9

Figure 8: Digital Design Flow Chart……..……...……....…………………………………… 11

Figure 9: TDES Pipelined Functionality Diagram …………………………...…...……….. 12

Figure 10: Equipment And EDA Tools ……...…………………………………...………… 13

Figure 11: Alice, Bob and Eve Encrypted Communication Example ………………...……… 14

Figure 12: Top Key Schedule Component For DES Encryption.…...………….……………. 25

Figure 13: Permutation Choice 1 Diagram.……………………...……………………..…….. 26

Figure 14: Left Rotation Outputs.……...……..………………..……………………...…… 28

Figure 15: Permutation Choice 2 Diagram……...………………………...……...………… 29

Figure 16: DES Encryption Top Component……...……...………………...………………… 30

Figure 17: Initial Permutation.…………………………...…...……...……………...…….. 31

Figure 18: Feistel Function Design……………………………………………………… 32

Figure 19: Expansion Permutation……………………………………………………… 33

Figure 20: Substitution Box S1………………………………………………………… 34

Figure 21: Substitution Box S2………………………………………………………… 34

Figure 22: Substitution Box S3………………………………………………………… 34

Figure 23: Substitution Box S4………………………………………………………… 35

Figure 24: Substitution Box S5………………………………………………………… 35

Figure 25: Substitution Box S6………………………………………………………… 35

Figure 26: Substitution Box S7………………………………………………………… 36

Figure 27: Substitution Box S8………………………………………………………… 36

Figure 28: Feistel Permutation………………………………………………………… 37

Figure 29: Left and Right 16 Round Outputs………………………………………… 38

Figure 30: Final Permutation ………………………………………………………… 39

Figure 31: Top Key Schedule Component For DES Decryption………………………… 41

Figure 32: DES Decryption Top Component…………………………………………… 43

Figure 33: TDES Non-Pipelined Encryption …………………………………………… 44

Figure 34: TDES Non-Pipelined Decryption…………………………………………… 44

Figure 35: Key Schedule Pipelined Design for Encryption……………………………… 48

Figure 36: DES Pipelined Encryption Top Component ………………………………… 49

Figure 37: Key Schedule Pipelined Design for Decryption …………………………… 51

Figure 38: DES Pipelined Decryption Top Component ………………………………… 52

Figure 39: DES Components Linked To Implement The TDES Encryption……………… 53

Figure 40: Key Bank Component For The TDES Design……………………………… 54

Figure 41: TDES And Key Bank Top Component Mapping For Encryption …..……..…… 54

Figure 42: DES Components Linked To Implement The TDES Decryption ……..………… 55

Figure 43: TDES And Key Bank Top Component Mapping For Decryption……………..… 55

Figure 44: ECB Mode of Operation for DES/TES Encryption…………………………..…… 58

Figure 45: ECB Mode of Operation for DES/TES Decryption…………………….....……… 59

Figure 46: DES Pipelined Encryption MOV Test 1……………………………..…………… 61

Figure 47: DES Pipelined Encryption MOV Test 1 Part 2…………………………………… 61

Figure 48: DES Pipelined Encryption MOV Test 2……………………………...………… 62

Figure 49: DES Pipelined Encryption MOV Test 2 Part 2…………………………………… 63

Figure 50: DES Pipelined Encryption MOV Test 3……………………………..…………… 65

Figure 51: DES Pipelined Encryption MOV Test 3 Part 2…………………………………… 65

Figure 52: DES Pipelined Encryption MOV Test 4……………………………...………… 67

Figure 53: DES Pipelined Encryption MOV Test 4 Part 2……………………..……...……… 67

Figure 54: DES Pipelined Encryption MOV Test 5……………………………...……… 68

Figure 55: DES Pipelined Encryption MOV Test 5 Part 2…………………………………… 68

Figure 56: DES Montecarlo Test For Encryption………………………….……….……… 70

Figure 57: DES Montecarlo Test Results For Encryption………………………………..…… 71

Figure 58: TDES Montecarlo Test For Encryption ……………………………..………… 72

Figure 59: TDES Montecarlo Test Results For Encryption……………………...……… 73

Figure 60: Design's Cost and Performance Evaluation Set Up …...…………………..……… 75

Figure 61: DES Encryption Analysis & Synthesis Cost Results…...………………………… 77

Figure 62: DES Decryption Analysis & Synthesis Cost Results……..……………………… 78

Figure 63: DES Encryption Analysis & Synthesis Running Time………………...……… 79

Figure 64: DES Decryption Analysis & Synthesis Running Time…………………………… 79

Figure 65: DES Encryption Substitution Table Known Answer Test In Timing Mode........… 80

Figure 66: DES Decryption Substitution Table Known Answer Test In Timing Mode…...… 80

Figure 67: DES Encryption Propagation Results………………………………..………… 82

Figure 68: DES Decryption Propagation Results……………………….…...……...…… 82

Figure 69: DES Encryption Node Finder Time………………………..………………...… 83

Figure 70: DES Decryption Node Finder Time……………………………………….…… 84

Figure 71: DES Encryption Generate Simulation Netlist Time…………………….……… 85

Figure 72: DES Decryption Generate Simulation Netlist Time……………………………… 86

Figure 73: DES Encryption Functional Simulation Time……………………..…...……… 87

Figure 74: DES Decryption Functional Simulation Time…………………………..……… 88

Figure 75: DES Encryption Timing Simulation Time……………………………...…..…… 89

Figure 76: DES Decryption Timing Simulation Time……………………………………...… 90

Figure 77: TDES Analysis & Synthesis Cost Results…………………………...………… 91

Figure 78: TDES Analysis & Synthesis Running Time…………………………...…..…… 92

Figure 79: TDES Substitution Table Known Answer Test In Timing Mode…………….…..… 93

Figure 80: TDES Propagation Results…………………………………………….…..……… 95

Figure 81: TDES Node Finder Time……………………………………………….………… 96

Figure 82: TDES Generate Simulation Netlist Time…………………………….………… 97

Figure 83: TDES Functional Simulation Time………………………………...…...……… 98

Figure 84: TDES Timing Simulation Time………………………………….....………… 99

Figure 85: DES Pipelined Analysis & Synthesis Cost Results………………...………...…... 100

Figure 86: DES Pipelined Analysis & Synthesis Running Time…………………...…..… 100

Figure 87: DES Pipelined Substitution Table Known Answer Test In Timing Mode……… 102

Figure 88: DES Pipelined Delay Results……………………………………....……… 104

Figure 89: DES Pipelined Node Finder Time…………………………..……………… 105

Figure 90: DES Pipelined Generate Simulation Netlist Time…………………….....… 106

Figure 91: DES Pipelined Functional Simulation Time……………………..………… 107

Figure 92: DES Pipelined Timing Simulation Time………………..…………..………… 108

Figure 93: TDES Pipelined Analysis & Synthesis Cost Results …………………………… 109

Figure 94: TDES Pipelined Analysis & Synthesis Running Time……………..………… 109

Figure 95: TDES Pipelined Substitution Table Known Answer Test In Timing Mode.......... 111

Figure 96: TDES Pipelined Delay Results………………………………………..…..… 113

Figure 97: TDES Pipelined Node Finder Time…………………………………...…… 114

Figure 98: TDES Pipelined Generate Simulation Netlist Time………………..…..……… 115

Figure 99: TDES Pipelined Functional Simulation Time…………………………………… 116

Figure 100: TDES Pipelined Timing Simulation Time…………………………...……… 117

CHAPTER I

INTRODUCTION

In the area of computing, Field Programmable Gate Arrays (FPGAs) offer a new solution for optimizing the performance of applications. FPGAs provide the flexibility of reconfiguring the silicon connection structure. This permits an algorithm needed for a specific application to be implemented directly on the FPGA [1]. The optimized architecture can execute the specific application in less cycles when compared to a CPU. In a CPU, the algorithm is broken down into a sequence of operations that take multiple cycles to execute [2]. An optimized algorithm implemented in FPGAs can execute in one or a few cycles [3]. Microsoft and Ryft have taken advantage of the accelerated computing, offered by FPGAs, to implement algorithms for their search engines [4] [5]. Microsoft's project Catapult, started by Doug Burger, "improved the operations per seconds of a critical components of Bing's search engine by nearly a factor of two" by loading some workload into Altera's Stratix V [6] [7]. Having had a success incrementing the performance, with the Catapult project, Microsoft used the same platform to process the massive amounts of traffic data from Azure and Office 365, thus assisting in deep machine learning. The PFGA implementation in Microsoft's racks can be seen in Figure 1 as presented in [8]. Prototype Version 2 shows every rack connecting to an FPGA that processes incoming traffic from the network.

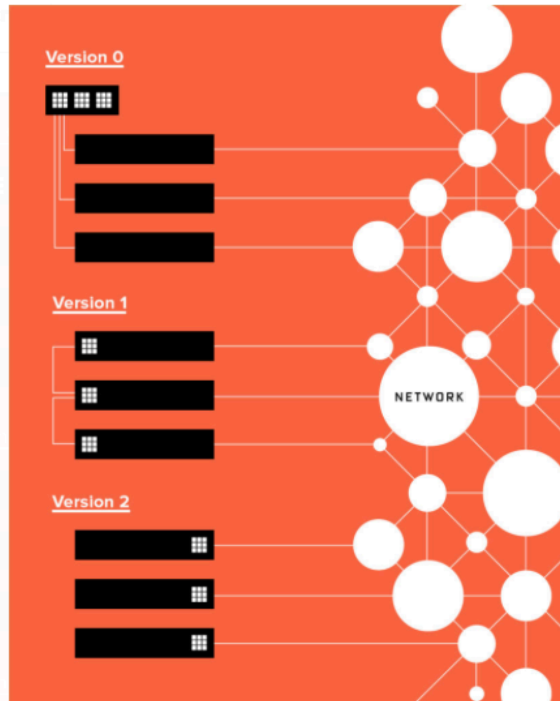


Figure 1: Microsoft's Prototype For Connecting FPGAs To The Rack and Processing Traffic [8]

Microsoft's success in using FPGA's, as reprogrammable architecture to meet their hardware needs, led Intel to acquire Altera for 16.7 billion dollars [9]. This is Intel's largest purchase in the history of the company. IBM has also taken this approach to meet the end user demands for processing massive amounts of data. IBM's solution incorporates a Coherent Accelerator Processor Proxy (CAPP) unit in their POWER 8 chip and a Power Service Layer (PSL) in the FPGA that allows coherency between the FPGAs and the chip via the Peripheral Component Interconnect Express (PCIE) [10] [11]. See Figure 2, provided in [11]. By incorporating coherency, the algorithms in the FPGA have access to the resources of the chip as well.

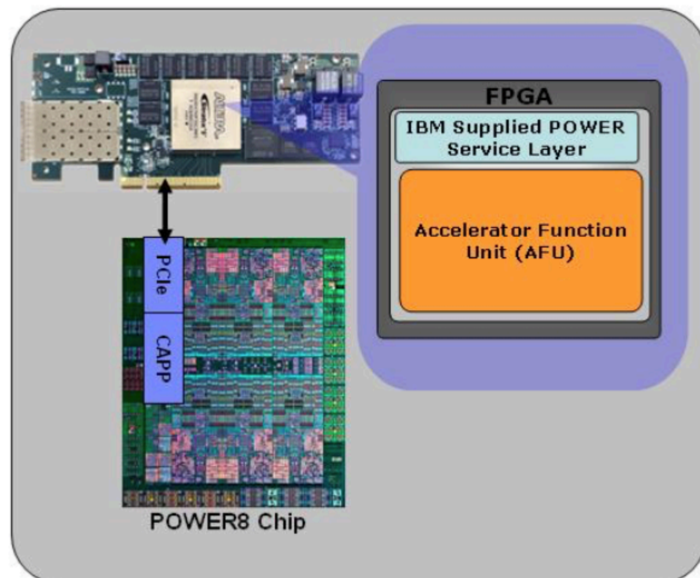


Figure 2: IBM's FPGA Acceleration Approach via CAPI [11]

From IBM's Coherent Accelerator Processor Interface (CAPI) documentation, in Figure 3, we see the potential market targets that can take advantage of this new resource [11].

Figure 3: IBM's Potential Markets For CAPI [11]

1.1 FPGA History

FPGAs derived from Programmable Logic Devices (PLDs) and Complex Programmable Logic Devices (CPLDs). PLDs consist of fully connected macros containing a few logic elements and a sequential element. The sequential element allows the PLD to be a reconfigurable device for digital circuits. CPLDs contain multiple PLDs and they may or may not be fully connected and they support more complex functions [12] [13]. The invention of the FPGA is accredited to Ross Freedman in 1984. Afterwards, Ross Freedman founded Xilinx, an American company supplier of FPGAs. Xilinx and Altera, both competitive American companies, dominate about 90 percent of the current FPGA market [14]. These two companies have been setting the FPGA technology standard for the past decades. Xilinx's current fastest technology, embedded in their Virtex and Kintex UltraScale+ FPGA series, is the 16 nm FinFET, meanwhile, Altera's fastest technology, embedded in the Stratix FPGA series, is the 14 nm Tri-Gate [15] [16] [17]. The Altera's Stratix series density can be up to 5.5 million Logic Elements (LEs) and the Xilinx's UltraScale+ series density can be up to 3.7 million System Logic Cells [18].

1.2 FPGA Business Market Model

As stated before, Altera and Xilinx are two FPGA vendors that dominate about 90 percent of the supply market. Intellectual Property (IP) logic block vendors license their designs to FPGA vendors. These designs are based on the specifications given by the FPGA vendors. The FPGA market and supply flow, shown in Figure 4, is extracted from [19].

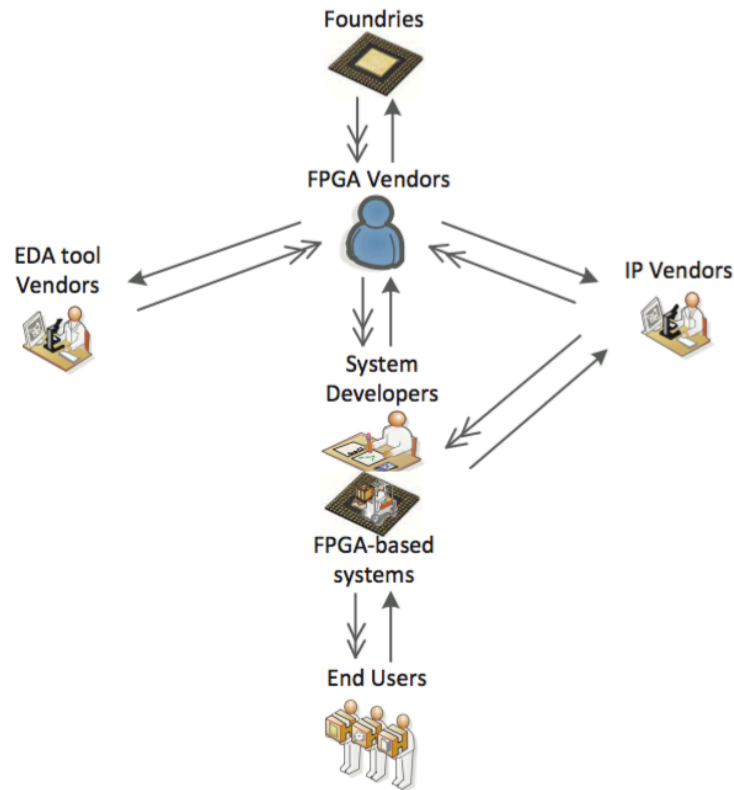


Figure 4: FPGA Market Demand and Supply Flow [19]

FPGA vendors define the specifications based on the needs of the System Developers, who must meet the specifications of FPGA-based systems, which are dictated by the needs of the End Users. With the assistance of Electronic Design Automation (EDA) tools, provided by EDA vendors, FPGA vendors can implement and verify their designs.

1.2.1 FPGA Vendors, Technology and Altera's DE2 Board EP2C35F672C6

Xilinx and Intel owned Altera dominate 90 percent of the FPGA market. The high-end technology embedded in Altera's Stratix FPGA series is the 14 nm Tri-Gate and the density can be up to 5.5 million LEs. Xilinx high-end technology embedded in the Virtex and Kintex UltraScale+ FPGA series is the 16 nm FinFETs and the density can be up to 3.7 million LCs. Table 1 compares the technology, maximum density and DSP logic operation of Altera's high-

end Stratix versus Xilinx high-end UltraScale+ [20] [21] [22].

Table 1: Comparison of Altera VS Xilinx High-End FPGA

	Technology	Density	Cost	DSP Operation	RELEASE
Altera Stratix 10	14 nm Tri-Gate	5.5 Million LEs	$24,995*	1 GHz	N/A***
Xilinx UltraScale+	16 nm FinFET	3.7 Million LCs	$15,995**	600 – 891 MHz	2016
Altera EP2C35	90 nm	33,216 LEs	$284 - $495	180 – 260 MHz	2005
	*price of 100G Development Kit Stratix V GX Edition **Xilinx Virtex UltraScale FPGA VCU110 Development Kit ***Stratix 10 Was Announced On 2013 But Has Not Been Released				

From Altera's website, the price of the 100G Development Kit Stratix V GX Edition is $24,000, and from Xilinx's website, the price of the Virtex UltraScale FPGA VCU110 Development Kit is $15,995. Altera's DSP Block operation frequency of 1GHz is greater than Xilinx's Slice frequency operation of 891 MHz.

In this thesis we use the Altera's Cyclone II DE2 Board EP2C35F672C6 platform. The technology in Cyclone II was realeased in 2005 [23]. The density, of model EP2C35F672C6, is 33,216 LEs and the technology is 90 nm. The DSP operation is about 200 MHz. This is the platform we have available in our Electrical Engineering department and we decided to evaluate our algorithms using the technology of this platform. This development board is available in Terasic's website for $495 [24]. The Altera Cyclone 2 FPGA contains the following: 33,216 Logic Elements, 105 M4K RAM blocks, 483,840 total RAM bits, 35 embedded multipliers, 4PLLs, 475 user I/O pins and the FineLine BGA 672-pin package. It contains an internal 50 MHz clock, an internal 27 MHz clock and an SMA external clock input [25].

The Logic Element (LE) is the unit that helps us determine the cost in Altera's Cyclone II

architecture.

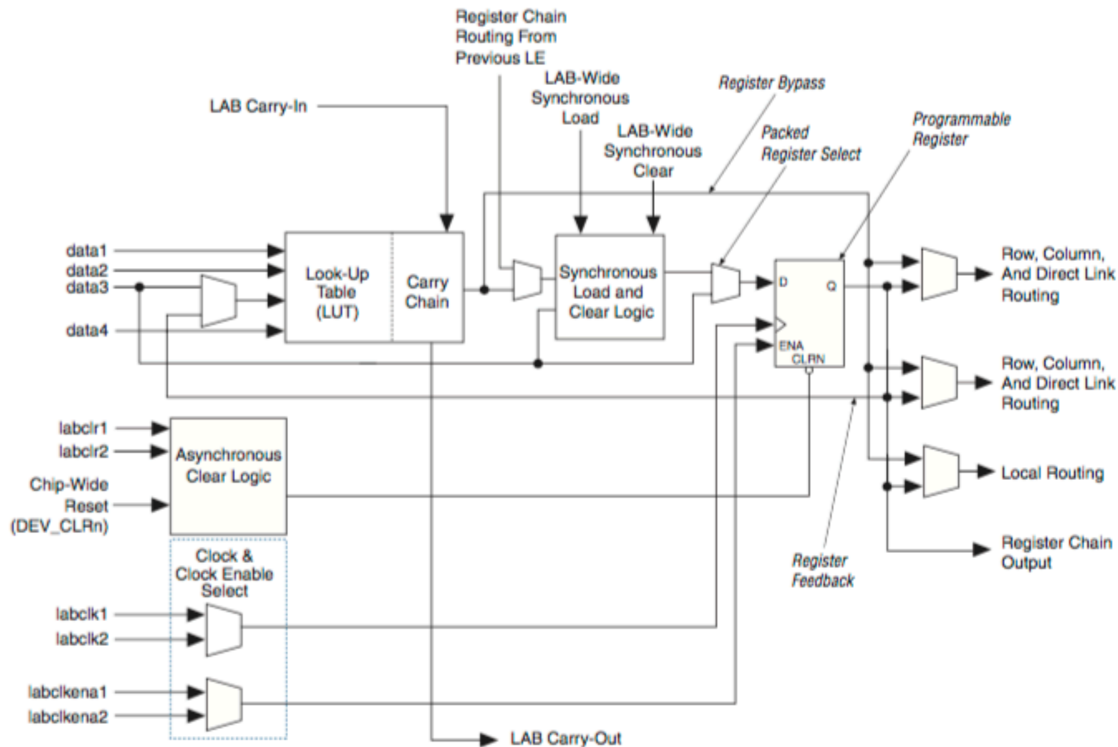


Figure 5: LE Architecture [25]

The LE is also smallest unit of logic that provides the following features, seen in Figure 5, as explained in [25]:

- A four-input look-up table (LUT), which is a function generator that can implement any function of four variables
- A programmable register
- A carry chain connection
- A register chain connection
- The ability to drive all types of interconnects: local, row, column, register chain, and direct link interconnects
- Support for register packing
- Support for register feedback

The LE can be programmed to be a sequential element or, if needed, the sequential element can be bypassed and the Look-Up Table output can be set as the direct output of the LE for combinational functions. A group of 16 LE's, as seen in Figure 6, are grouped into 1 Logic Array Block (LAB) and LABs are grouped into rows and columns.

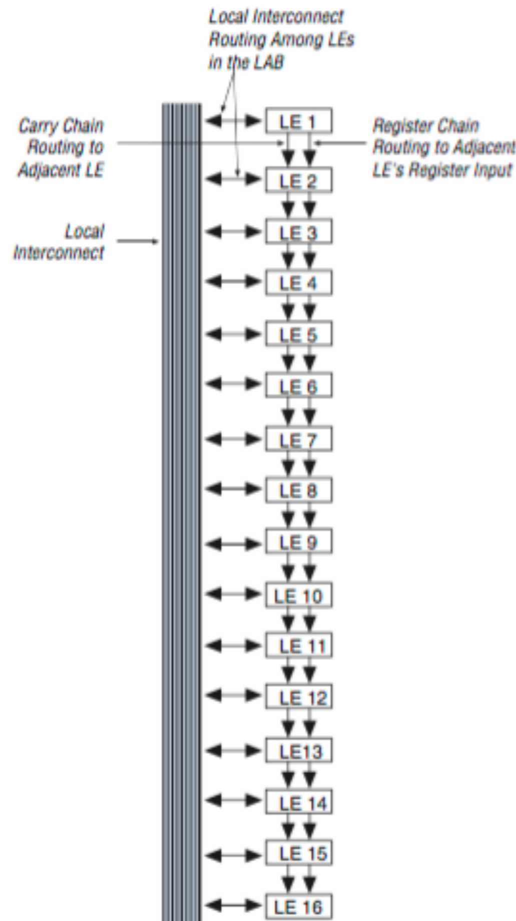


Figure 6: LE's Interconnects in 1 LAB [25]

Each LAB consists16 LEs, LAB control signals, LE carry chains, register channels and local interconnects. The LAB control signals consist of 2 clocks, 2 clock enables, two asynchronous clears, 1 synchronous clear and 1 synchronous load. Figure 7 shows LABs arranged in rows and columns with the proper interconnect.

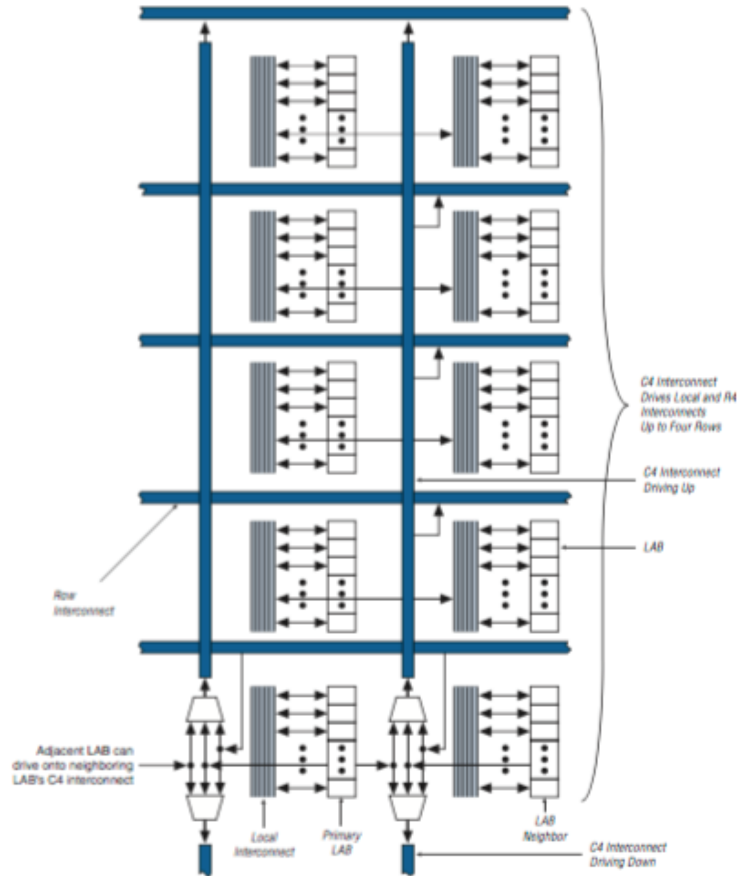


Figure 7: LABs Arranged Into Rows and Columns With Interconnect [25]

1.2.2 FPGA Based System Developers and Thesis Objective

The FPGA Based System Developers, as described in [19], “create commercial products on FPGA chips. The product is normally in the form of configuration bit stream file for a given family FPGA chips.” In this thesis, the function we implement, in VHSIC Hardware Description Language (VHDL), is the Data Encryption Standard (DES) and the Triple Data Encryption Standard (TDES) cryptographic algorithms for Altera’s Cyclone II DES2 EP2C35F672C6 FPGA Board. We implement and verify the pipelined and non-pipelined algorithms as stated in the National Institute of Standards (NIST) special publications 800-17 and 800-20 [26] [27]. The goal is to analyze the cost while increasing the cryptographic throughput by keeping a coherent

design that allows the non-pipelined designs to be pipelined with out major modifications. To achieve the coherent design, we implement different key scheduling schemes for the DES encryption and decryption algorithms. As presented in Section 2.1.1.1 and 2.1.2.1, the key scheduling for the DES encryption scheme performs left shift rotations and the key scheduling for the DES decryption scheme performs right shift rotations. Implementing the key schedules, this way, is different from what we have seen in the literature reviews presented in Section 1.4. The literature reviews implement left shift rotations only. Implementing left shifts rotations alone make it difficult to implement a fully TDES pipelined design since the pipelined encryptions and decryptions for DES are incoherent. As seen in Section 3.1.2, our coherent DES designs allow us to fully pipeline TDES by adding a key bank that buffers the keys and feeds them to the proper components in the proper cycle.

1.2.3 Conceptual Digital Design Flow and EDA Tools

The conceptual digital design flow is a top-down technique approach to solve a problem. The Design flow will help us implement the algorithms for both non-pipelined and pipelined cryptography schemes mentioned above. As seen in Figure 8, we begin with the design specifications.

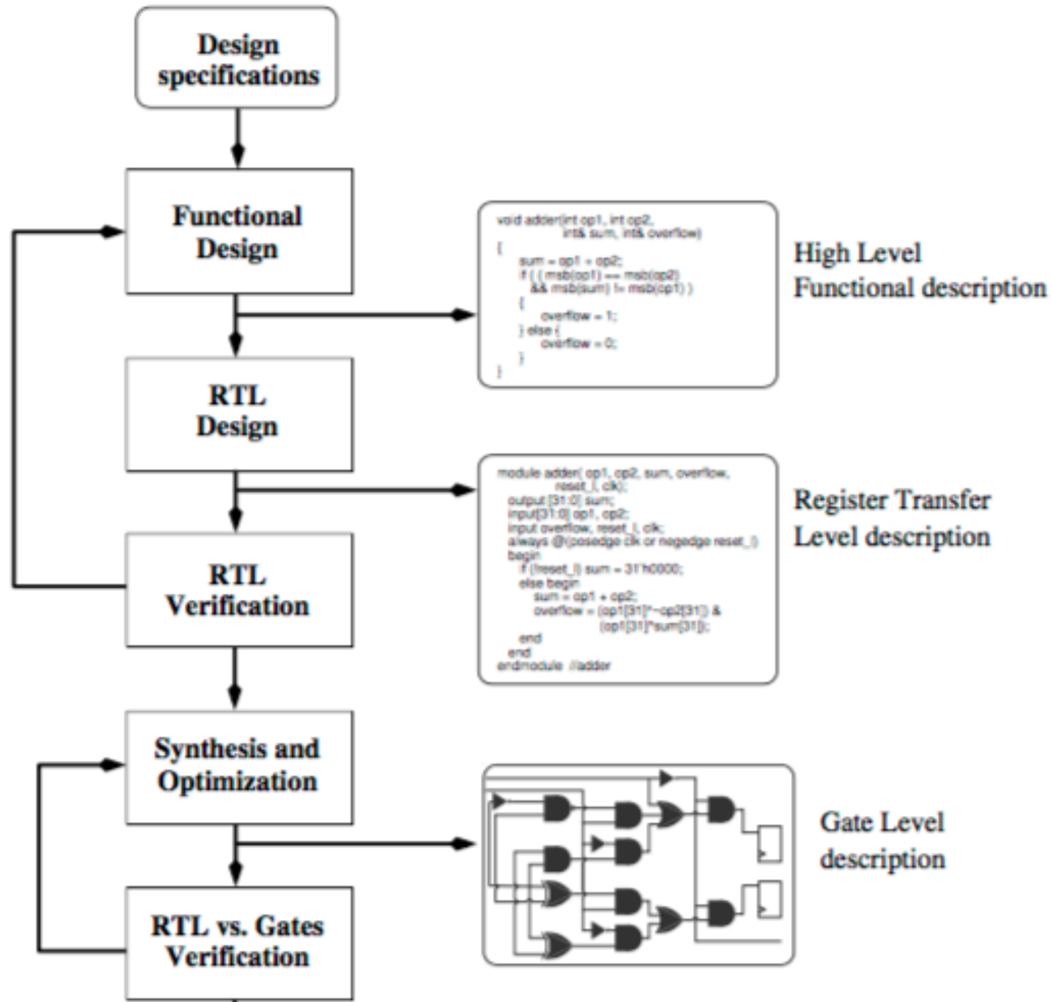


Figure 8: Digital Design Flow Chart [28]

The designs specifications consist on a list of requirements the design must meet. The requirements may include cost, power consumption, dimensions, functionality, and timing [28]. In our case, we are only constricted to functionality and cost because we are evaluating the performance (throughput). Our cost constraint is 33, 216 LEs. In Figure 9 we show a basic functionality overview of our TDES pipelined encrypting design. The functionality details of this design and all other designs are presented in Chapter 3.

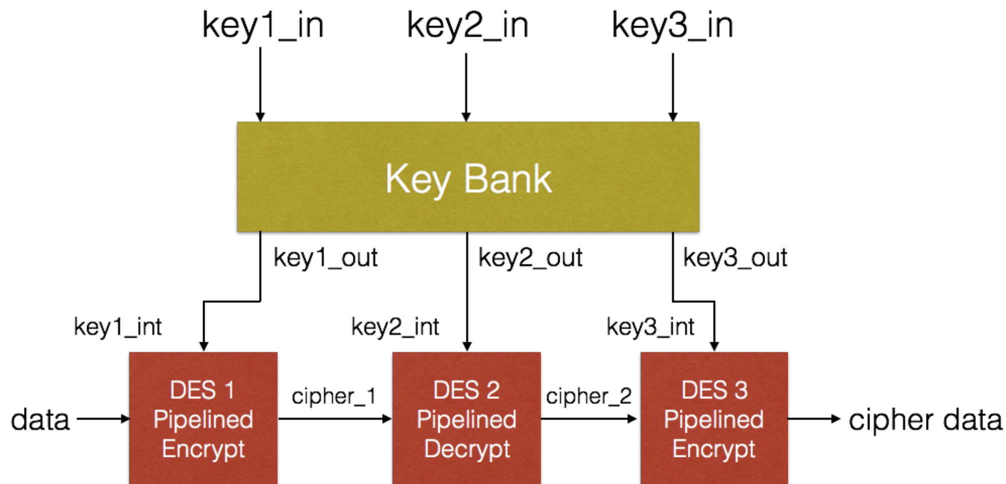


Figure 9: TDES Pipelined Functionality Diagram

The Functional Design makes use of High Level Synthesis (HLS) to implement the functional description of the design and generate the VDHL code. High-level languages such as C, C++ and Matlab are used in his process [29]. In industry, the functionalities of the designs are complex and HLS is required to minimize human error when generating the VHDL code. In our case, we don't use HLS and we directly go to the Register Transfer Level (RTL) Design. The Register Transfer Level consists of the VHDL code that describes the digital logic and sequential elements. The RTL Verification assures that the VHDL code functions as expected. Synthesis and Optimization looks in more detail at the gate level, sequential level and the netlist of the design to optimize for the requirements given. The RTL vs. Gates Verification ensures that, after optimizing the design, it still functions as expected with the requirements. The complete digital design flow chart continues to physical design/IC layout, fabrication and validation. For this thesis, we won't send our design for fabrication and therefore are not concerned with these other processes.

The Electronic Design Automation (EDA) tools aide in the implementation of the digital design flow. In this thesis we use Altera Quartus II software and the Altera University Program

(UP) Simulator. The version of Quartus used for implementing DES and TDES is Quartus II Web Edition Service Pack 1. The Quartus II software is installed on an iMac Mid 2011. The iMac is equipped with an Intel i5 core 2.5 GHz Processor. The operating system is Windows 8 Enterprise. See Figure 10. The RTL design and the Synthesis were done in Quartus II. Quartus II also provides the cost information for the designs we implemented. The cost information includes the number of Logic Elements (LEs), memory elements, pins and other information that is used to implement our design in the FPGA. We compare the cost information of our designs in Chapter 5. The verification was implemented in an extension program of Quartus II called the University Program Simulator (Altera U.P. Simulator). The simulator outputs a waveform image in which we visually analyze the time delays and propagation times as the inputs traverse the stages of the algorithms. The version of Altera's U.P. Simulator we use is the edition compatible with Quartus II 12 Web Edition Service Pack 1. Both of these Altera Programs can be downloaded from the Altera website free of charge. See reference [30].

Figure 10: Equipment And EDA Tools

1.3 Cryptography

Cryptography seeks to maintain the confidentiality of information. Confidentiality is one of the main components in the security triad [31] [32]. A system must ensure that the user's information is maintained inclusive to the parties that legitimately share it. Other users that seek to access the data without the adequate permission should only see an encrypted form of the data, which does not permit the non-legitimate user (hacker) to utilize the actual data.

Cryptography involves encryption and decryption. Encryption is the process of converting data to cipher data. Cipher data is an encrypted form of the data that is unreadable to anyone. Decryption is the process of converting cipher data to readable data that anyone can utilize. See Figure 11 below.

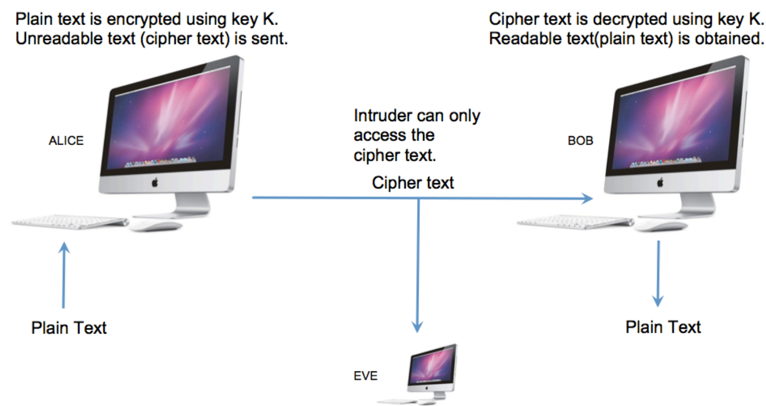


Figure 11: Alice, Bob and Eve Encrypted Communication Example

Alice seeks to send confidential data to Bob. Lets assume that the data is simple plain text. At Alice's end, her system encrypts the plain text using key K and is rendered unreadable. This cipher text is sent to Bob. Because Bob possesses the proper key K, he is able to decrypt the cipher text into plain text. Bob is now able read Alice's plain text. Eve is an eavesdropper that

gains access to the cipher text. However, since Eve does not have the proper key, Eve is not able to decrypt and read the message. To Eve, the intercepted cipher text is only a handful of random characters that make no sense.

To encrypt and decrypt, a key is needed. There exist two types of cryptography: symmetric and asymmetric key cryptography. Symmetric key cryptography requires the same key for the encryption and decryption processes and it is also known as the secret key cryptography [33]. Symmetric key cryptography schemes include the Data Encryption Standard (DES), Triple Dada Encryption Standard (TDES), Advanced Encryption Standard (AES) and Blowfish. Symmetric schemes can be implemented as a stream cipher or a block cipher. E.g. DES implemented in Electronic Codebook mode is considered a stream cipher but DES implemented in Cipher Block Chaining (CBC) mode is considered a block cipher. Applications for symmetric cryptography include government documents, email applications and securing credit card information (#!include reference). These ciphers are mainly used to encrypt large quantities of data because their required computation cost is less than the required computation cost for asymmetric schemes [31]. Asymmetric key cryptography requires different keys for the encryption and the decryption processes and it is also known as the public key cryptography. Asymmetric key cryptography schemes include Rivest Shamir Adleman (RSA) and Digital Signature Standard (DSS). A well known asymmetric scheme application is the communication over a public network environment. DSS is used to establish a secure network connection because it only requires small amounts of data to be encrypted.

1.3.1 Cryptography Current Events And History

Attacks in Paris, France and San Bernardino, California have triggered an ongoing debate

between tech companies and the government regarding the user's privacy. The topic in concern is encryption [34] [35]. With the ongoing election of 2016, the presidential candidates of the United States have demonstrated their concern regarding end-to-end encryption for social apps in mobile devices. End-to-end encrypted communication in mobile devices provides confidentiality and authenticity. Taking advantage of this features, attackers used encrypted apps such as Whatsapp and Telegram to communicate, plan and ultimately perpetrate the terrorist attacks of November 13, 2015 in Paris, which resulted in lost lives and hospitalized victims [36]. These events lead a position, mainly supported by the government and the 2016 US presidential candidates, that contemplates enforcing a backdoor on the security for mobile devices. As expected, the majority of the tech companies do not support this notion. Arguments for opposing a backdoor include users' privacy, undermining the purpose of technology and business diminution for domestic apps. Enforcing a backdoor would require developers to adhere to the domestic policy, thus, driving users to international platforms that don't append to the backdoor policy [37]. CEOs of top tech companies such as Tim cook, CEO of Apple, have been brought forth to congressional hearings regarding the possibility of implementing a backdoor on their devices [37]. Tim Cook states that without a warrant, his tech company has no intention or desire to give out its users data to anyone [38]. The strong opposition to a backdoor has led the government to seek other means of accessing a devices data in its decrypted form. In March 28 2016, the FBI announced they successfully decrypted the encrypted smartphone that belonged to the assassin of the San Bernardino attack after the smartphone maker refused to create software to decrypt the device [39].

The war between cryptographers and cryptanalysts has been ongoing. Among the first identified methods of encryption are coded messages. Caesar's cipher, named after Julius

Caesar, is one method that was used for encoding a message that consisted on replacing each letter with another by shifting down the alphabet. These early alphabetic methods are also known as substitution ciphers. To decrypt the encoded message the shift in the reverse direction is applied. This is the equivalent of decrypting a message using the same key that was used to encrypt it. Cryptography schemes that require the same key for encryption and decryption are known as symmetric key schemes. The method of encoding a message is simple to decrypt due to the low maximum number of possible combinations. As the complexity of encoding messages progressed, the methods evolved to substituting words by using mathematical schemes. The Advanced Encryption Standard (AES), an algebraic cryptographic scheme, along with the Triple Data Encryption Standard, have been set as the encryption standard by the National Institute of Systems and Technology (NIST) [40].

Decrypting messages has become as crucial as encrypting messages. Decryption has played an important role in changing the course of history. The US involvement in World War I was due to the interception and decryption of the Zimmerman telegraph. The Zimmerman telegram, intercepted by Britain, intended to encourage Mexico to raise arms against the US with the guarantee that Germany would provide them with full support and resources [41]. Offended by the German proposition, US entered the World War I and sided with the Allies. The US's involvement leaned the balance towards the Allies side and, needless to say, the Allies won the war. In World War II, the Germans communicated by radio communication. Since anyone was able to intercept the over the air communication, the Enigma machine was used to encrypt the messages. However, the British found a method for decrypting their messages [42]. By leaning the balance to the Allies side once more, the Allies won the war and it is estimated that this helped shorten the war by about two years.

Among the many applications for cryptography, one application that stands out is information safekeeping. Information safekeeping has changed from file cabinets and locks to online databases and automated tools for protecting them [43]. The advantage to safekeeping in databases is the feasibility to analyze the data at a fast rate and use it to increase productivity. The new business model infrastructure concerning computer consulting firms and computer architecture companies like Intel, IBM, Apple, Microsoft and Google includes the cloud market, processing and interpreting massive amounts of data [44]. Online databases undoubtedly provide a more convenient form of storing and accessing data, but keeping a database, makes the need for interconnects via a collection of interconnected networks. Thus, resulting in the need to establish secure encrypted connections.

As the computation power has increased over the last century, so has the complexity of the cryptography schemes. The field of cryptography is broad due to the many components associated with it. Components in this field include: key distribution, algorithms, cryptanalysis and hardware implementation. The work presented here focuses on hardware implementation. We implement and evaluate the cost and performance of two commonly used schemes: The Data Encryption Standard (DES) and the Triple Data Encryption Standard (TDES). We successfully validate the implementation, in Electronic Codebook mode, of these two schemes in VHDL as described in [45] and [46].

1.3.2 DES & TDES History

The work presented in this thesis focuses on the symmetric DES and TDES cryptography schemes in Electronic Codebook (EBC) mode. Electronic payment systems are known to use the TDES scheme for the encryption/decryption of data, and hence faster implementations are of

great significance [3] [4]. Mail applications, such as Microsoft Outlook, make use of this scheme as well [5] The National Bureau of Standards (NBS) issued a request for an encryption standard in 1973. International Business Machines (IBM) Corporation originally introduced DES as LUCIFER in the 1960s. With the consultation of the National Security Agency (NSA) along with other outside consultants, and some adjustments, DES was eventually adopted as the Data Encryption Standard (DES) in 1976. Some of the modifications include the reduced bit size of the key from 128 to 56 bits and the design of the S-boxes that perform the non-linear bit substitutions was also kept classified. In 1999 DES was reaffirmed for the fourth time as triple DES (3DES) after DES was proven susceptible to a brute force attack in less than 24 hours. The triple DES increased the key size from 56 bits to 128 bits if two different keys are used [47] [48]. If three different keys are used in the scheme, the key size increases to 192 bits. The current cryptography schemes affirmed as the standards are TDES and Advanced Encryption Standard (AES). Although AES and 3DES have been reaffirmed as the standard now days, DES is still being used where a strong encryption in not needed.

1.3.3 Exhaustive Key Search Analysis

A brute force attack is a well-known method of cryptanalysis. The objective of the brute force attack is to run an exhaustive key search analysis, with all possible key combinations, on a cryptographic algorithm. An exhaustive search on DES would require a total of $2^{56} = 7.2 \text{ X } 10^{16}$ combinations. As the size of the key increases, the number of combination also increases exponentially.

Table 2: Computation Power And Time For Exhaustive Search [49]

Key Size (bits)	Cipher	Number of Alternative Keys	Time Required at 10^9 Decryptions/s	Time Required at 10^{13} Decryptions/s
56	DES	$2^{56} \approx 7.2 \times 10^{16}$	2^{55} ns = 1.125 years	1 hour
128	AES	$2^{128} \approx 3.4 \times 10^{38}$	2^{127} ns = 5.3×10^{21} years	5.3×10^{17} years
168	Triple DES	$2^{168} \approx 3.7 \times 10^{50}$	2^{167} ns = 5.8×10^{33} years	5.8×10^{29} years
192	AES	$2^{192} \approx 6.3 \times 10^{57}$	2^{191} ns = 9.8×10^{40} years	9.8×10^{36} years
256	AES	$2^{256} \approx 1.2 \times 10^{77}$	2^{255} ns = 1.8×10^{60} years	1.8×10^{56} years
26 characters (permutation)	Monoalphabetic	$2! = 4 \times 10^{26}$	2×10^{26} ns = 6.3×10^{9} years	6.3×10^{6} years

As seen in Table 2, a computation power of 10^{13} decryptions/s would complete an exhaustive search on DES in 1 hour [49]. However, an exhaustive search on TDES would take 5.8 X 10^{29} years. It is safe to say that TDES is a symmetric cipher that is unbreakable by means of a brute force attack.

1.4 Survey On Exiting Work And Expected Results

The performance of our implementation will be measured with throughput. We define throughput as the number of fully processed bits per second. In other words, the encrypted or decrypted bits per second. The cost is determined by the number of Logic Elements (LEs) needed to implement our designs. In Table 3 we compare the cost of the implementations presented in several publications.

We define the operable clock frequency as the frequency of one full encryption or decryption. In [53], the authors claim that their frequency for the DES 8 stage pipelined design is 20.79 MHz but their throughput is 665.28 Mbps. This means that the operable frequency is 20.79 MHz / 2 = 10.4q MHz. The highest DES non-pipelined throughput, 4.8 Gbps, is seen in [55]. From the information provided, in their publication, we can assume they implemented 1 permutation-substitution stage at 71.12 MHz. They also achieve the highest DES 16-stage

pipelined throughput of 18.2 Gbps at 294.031 MHz. From Table 2, we conclude that, to increase the throughput we can increase the operational frequency and increase the number of stages in the pipeline. The frequency threshold depends on the technology of the FPGA because of the propagation delay and time violations. The throughput is also proportional to the cost. To increase the throughput, more stages must be implemented, which means more logic elements or blocks.

Table 3: Cost and Performance Comparison

				DES Encrypt./Decrypt.		TDES Encrypt./Decrypt.		DES Pipelined Encrypt./ Decrypt.		TDES Pipelined Encrypt./ Decrypt.	
	Number Of Stages	Hardware	Operable Clock Frequency	Hardware Cost	Performance	Hardware Cost	Performance	Hardware Cost	Performance	Hardware Cost	Performance
Edni's Design	16	EP2C35F672C6	50 MHz	≈5,991 LEs		≈17,973 LEs		> 5,991 LEs	≈3.2 Gbps	> 17,973 LEs	≈3.2 Gbps
Andoni Et Al [50]	1	Xilinx Virtex XCV-1000	91/16 MHz	596 CLBs	347.1 Mbps						
	1		91/48 MHz			596 CLBs	115.7 Mbps				
FU Li, PAN Ming [51]	16	Altera Cyclone EP1C6Q240c8	100 MHz					-	6.4 GHz		
Liakot El Al [52]	-	FLEX 10K PLD EPF10K30BC356-3	-			*	*				
TOURIA ARICH EL AL [53]	1	EPF10K30BC356-3	17.39/16 MHz	851 LCs	69.56 Mbps						
	2	EPF10K30BC356-3	17.36/8 MHz					1305 LCs	138.88 Mbps		
	4	EPF10K50BC356-3	15.89/4 MHz					2095 LCs	254.36 Mbps		
	8	EP1K100FC484-3	20.79/2 MHz					3308 LCs	665.28 Mbps		
	16	EP1K100FC484-3**	16.47 MHz					5991 LCs	1.054 Gbps		
Ke Wang [54]	16	Xilinx Virtex4	228.6 MHz						16 Gbps		
Saeid Et Al [55]	1***	Xilinx Virtex-6	1201,923/16 MHz	-	4.8 Gbps						
	16***		294.031 MHz					-	18.2 Gbps		
Ji Yao, Hongo Kang [56]	16	Xilinx Virtex-II	-					3056 LE	3.2 Gbps		
Fang Ren ET AL [57]	1***	Xilinx Virtex4	215.165 MHz							-	800.66 Mbps

*Implemented **Device specs are inconsistent with the author's cost claim

***Assumption based on the data provided -Information is not provided

We cannot determine what the throughput of our non-pipelined designs would be because we don't have propagation delay data for implementing one permutation-substitution DES stage in the EP2C35F672C6 platform. However, we know that this platform has an internal 50 MHz clock. If we implement a 16-stage DES pipeline, we output 64 bits or processed data every 20 ns. We can achieve an approximate throughput of 64 bits / 20 ns = 3.2 Gbps. If we implement a 48-stage TDES pipeline we can also achieve an approximate throughput of 3.2 Gbps.

Regarding the cost of implementing our designs, we will gather this data from the synthesis of the EDA tools we discussed in Section 1.2.3. However, based on the results seen in [53], if we implement a 16-stage DES we expect the cost of our DES to be approximately 5,991 LEs and greater for the pipelined design. For our TDES design we expect the cost to be approximately 3 * 5,991 LEs = 17,973 LEs and greater for the pipelined design.

1.5 Thesis Outline

In this thesis we present a Non-Pipelined and Pipelined implementation in VHDL of two commonly used symmetric cryptography schemes: Data Encryption Standard (DES) and Triple Data Encryption Standard (TDES). We also evaluate the cost and performance of our implementations Using Altera's Quartus II and University Program (UP) simulator. The platform, for testing our designs, is based on Altera Cyclone II FPGA technology. This thesis report contains six chapters. Chapter 1 is the introduction. This chapter contains our statement of the problem, a background on cryptography, a description of the tools we used to implement and evaluate our schemes and our expected results. In chapter 2 we present a detailed description of our non-pipelined encryption and decryption schemes. In Chapter 3 we present in detail our pipelined schemes. Chapter 4 details the validation implementation and the results of the validation design schemes in Electronic Codebook (ECB) mode. In chapter 5 we present and evaluate the results obtained: the cost and performance of our schemes. We also compare our results to the ones presented in our literature reviews. Finally, chapter 6 contains our conclusion and future work.

CHAPTER II

DES & TDES NON-PIPELINED DESIGNS IN VHDL

In this chapter we present our DES & TDES Non-Pipelined designs. The DES scheme makes use of various techniques: permutation, XOR, substitution/transformation, and expansion. To encrypt data, the data and a 56-bit key are required. The actual length of the key is 64 bits, but in the first permutation in the key scheduling, 8 bits are discarded from the original 64-bit key, which leaves a 56-bit key standing. The discarded 8 bits are used for error detection. Most discussions of the DES algorithm index the string of bits from left right. The indexing for a 64-bit string would be [1 2 3 4 5 6 7 8 9 10 11 . . . 64]. However, in VHDL, indexing is quite the opposite. The indexing is done from right to left. The indexing for the key string of bits in VHDL is [63 62 61 . . . 10 9 8 7 6 5 4 3 2 1 0]. Note that VHDL begins with index 63 and ends with index 0. In this report we will abandon the indexing convention in most discussions and we will adopt the VHDL bit indexing convention. Further knowledge in VHDL is can be attained in [58] and [59].

The organization of this chapter is as follows: In Section 2.1 and 2.2 we present our DES non-pipelined designs and in Section 2.3 we present our TDES non-pipelined designs. In these sections we include our encryption schemes, decryption schemes and the specifications that our designs must meet. As mentioned earlier in Section1.2.2, a note of interest in this chapter is our

DES decryption key scheduling design. We perform right rotations in order to make our design coherent with our DES pipelined design in Chapter 3. The VHDL code for each component in the encryption and decryption designs is referenced in the text and provided in Appendix A.

2.1 DES Non-Pipelined Design For Encryption

Our DES non-pipelined encryption design has a key scheduling component, an initial permutation, a final permutation and 16 Feistel Function rounds. We present the algorithm in a similar way to [60]. The specifications for the DES non-pipelined encryption design are the following:

- The inputs to the top-level component are a 64-bit Key and 64-bit Data.
- The output to the top-level component is a 64-bit Cipher Data.
- The cost must be less than 33,216 Logic Elements.

2.1.1 DES Key Schedule Component For Encryption

The first part of the encryption algorithm involves scheduling 16 48-bit sub keys from the 64-bit key. These sub keys are introduced in order into the heart of the DES algorithm, which are called Feistel Function rounds. As seen in Figure 12, the key scheduler is presented in five steps: initial key permutation (PC_1), split into left and right halves (c0 and d0), left shift-rotate, concatenation and final key permutation (PC_2). The specifications for the DES key schedule component for encryption are the following:

- The input to the top-level key schedule component is the 64-bit Key.
- The outputs are 16-sub keys k1, k2, k3, …, k16.

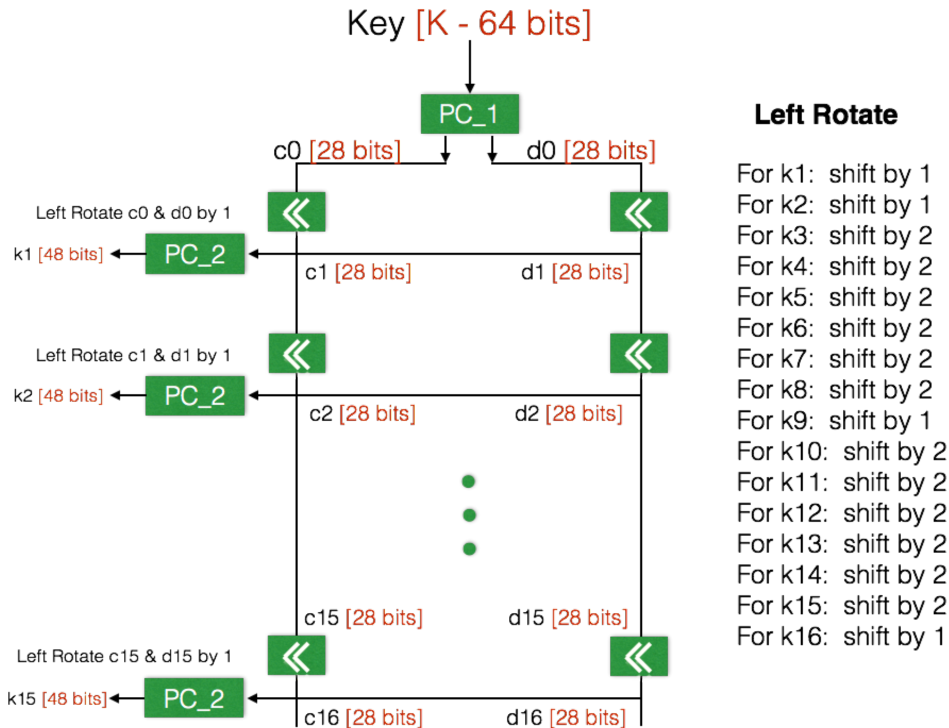


Figure 12: Top Key Schedule Component For DES Encryption.

Refer to the Appendix A section to view the VHDL code for the top key schedule component for encryption in DES_encrypt.vhd.

To provide a numerical example, lets assume the input key is 133457799BBCDFF1 in hexadecimal [D]. The 64-bit binary string is as follows:

Key = [0001 0011 0011 0100 0101 0111 0111 1001 1001 1011 1011 1100 1101 1111 1111 0001]

The permutation choice 1 is implemented as seen in Figure 13. A permutation is a scrambling of the bits. The initial key permutation reduces the key from the original 64 bits to 56 bits. The remaining 8 bits are not used in the key scheduling process but are rather used for error detection.

PC_1(key_permutation.vhd)						
7	15	23	31	39	47	55
63	6	14	22	30	38	46
54	62	5	13	21	29	37
45	53	61	4	12	20	28
1	9	17	25	33	41	49
57	2	10	18	26	34	42
50	58	3	11	19	27	35
43	51	59	36	44	52	60

Figure 13: Permutation Choice 1 Diagram [60].

From the numerical example stated above, the Key = 133457799BBCDFF1 in hex, the initial permutation would yield in the following 56-bit string:

[1111000011001100101010101111 0101010101100110011110001111]

The numbers inside the cells of Figure 13 represent the position of the bit in "Key". E.g. bit7 = 1, bit15 = 1, bit23 = 1, bit31 = 1, bit39 = 0, bit47 = 0, . . . , bit60 = 1.

The initial key permutation, which is implemented in the PC_1.vhd file, can be found in the appendix section of this report. The resulting 56-bit string is split in two halves c0 and d0. Continuing the example, c0 and d0 would yield the following:

c0 = [1111000011001100101010101111]

d0 = [0101010101100110011110001111]

c0 and d0 are 28-bit strings. These two strings of bits are left rotated by the amounts specified in Table 4.

Table 4: Left Rotations

Left Rotations		
For k1	Rotate c1 & d1 by :	1
For k2	Rotate c2 & d2 by :	1
For k3	Rotate c3 & d3 by :	2
For k4	Rotate c4 & d4 by :	2
For k5	Rotate c5 & d5 by :	2
For k6	Rotate c6 & d6 by :	2
For k7	Rotate c7 & d7 by :	2
For k8	Rotate c8 & d8 by :	2
For k9	Rotate c9 & d9 by :	1
For k10	Rotate c10 & d10 by :	2
For k11	Rotate c11 & d11 by :	2
For k12	Rotate c12 & d12 by :	2
For k13	Rotate c13 & d13 by :	2
For k14	Rotate c14 & d14 by :	2
For k15	Rotate c15 & d15 by :	2
For k16	Rotate c16 & d16 by :	1

Once the rotation is applied, each pair cn & dn is denoted as: c1, d1, c2, d2, c3, d3, c4, d4, . . . c15, d15. Each shift-rotation yields the sets of strings seen in Figure 14. The VHDL components for 1 left rotation and 2 left rotations are in Appendix A under SRL_1.vhd and SRL_2.vhd.

Left Rotation Outputs
c1 = [1110000110011001010101011111] d1 = [1010101011001100111100011110]
c2 = [1100001100110010101010111111] d2 = [0101010110011001111000111101]
c3 = [0000110011001010101011111111] d3 = [0101011001100111100011110101]
c4 = [0011001100101010101111111100] d4 = [0101100110011110001111010101]
c5 = [1100110010101010111111110000] d5 = [0110011001111000111101010101]
c6 = [0011001010101011111111000011] d6 = [1001100111100011110101010101]
c7 = [1100101010101111111100001100] d7 = [0110011110001111010101010110]
c8 = [0010101010111111110000110011] d8 = [1001111000111101010101011001]
c9 = [0101010101111111100001100110] d9 = [0011110001111010101010110011]
c10 = [0101010111111110000110011001] c10 = [1111000111101010101011001100]
c11 = [0101011111111000011001100101] c11 =[1100011110101010101100110011]
c12 = [0101111111100001100110010101] c12 = [0001111010101010110011001111]
c13 = [0111111110000110011001010101] c13 = [0111101010101011001100111100]
c14 = [1111111000011001100101010101] d14 = [1110101010101100110011110001]
c15 = [1111100001100110010101010111] d15 = [1010101010110011001111000111]
c16 = [1111000011001100101010101111] d16 = [0101010101100110011110001111]

Figure 14: Left Rotation Outputs

Each cn, dn pair is concatenated resulting in a 56-bit string. A final permutation (PC_2) is applied to each cn, dn concatenated pair. This permutation, as seen in Figure 15, reduces the bit string from 56 bits to 48 bits. The concatenation and permutation are processed in the PC_2.vhd file and can be found in Appendix A.

PC_2 (sub_key.vhd)					
42	39	45	32	55	51
53	28	41	50	35	46
33	37	44	52	30	48
40	49	29	36	43	54
15	4	25	18	9	1
26	16	5	11	23	8
12	7	17	0	22	3
10	14	6	20	27	24

Figure 15: Permutation Choice 2 Diagram.

The final permutation to the string pairs yield the following sixteen sub keys that are used in the Feistel functions of the DES scheme.

k1 = [000110110000001011101111111111000111000001110010]

k2 = [011110011010111011011001110110111100100111100101]

k3 = [010101011111110010001010010000101100111110011001]

k4 = [011100101010110111010110110110110011010100011101]

k5 = [011111001110110000000111111010110101001110101000]

k6 = [011000111010010100111110010100000111101100101111]

k7 = [111011001000010010110111111101100001100010111100]

k8 = [111101111000101000111010110000010011101111111011]

k9 = [111000001101101111101011111011011110011110000001]

k10 = [101100011111001101000111101110100100011001001111]

k11 = [001000010101111111010011110111101101001110000110]

k12 = [011101010111000111110101100101000110011111101001]

k13 = [100101111100010111010001111110101011101001000001]

k14 = [010111110100001110110111111100101110011100111010]

k15 = [101111111001000110001101001111010011111100001010]

k16 = [110010110011110110001011000011100001011111110101]

2.1.2 DES Encryption Initial Permutation, Final Permutation and 16 Rounds

The encryption process includes an initial permutation (IP), split into the left and right halves (L0, R0), 16 rounds of substitution (Feistel function, f) and XOR, and a final permutation (FP). Figure 16 depicts the top component in our DES encryption design. The DES encryption VHDL component is found in Appendix A under DES_encrypt.vhd.

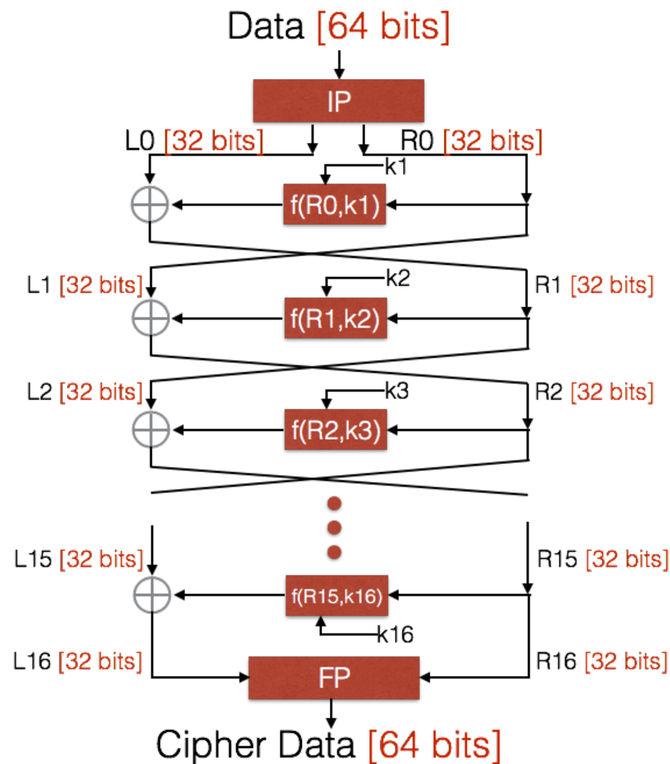


Figure 16: DES Encryption Top Component.

To provide a numerical example, lets assume the input data is 012345679ABCDEF in hexadecimal. The 64-bit binary string is as follows:

data = [0000 0001 0010 0011 0100 0101 0110 0111 1001 1010 1011 1100 1101 1110 1111]

The initial permutation is performed as seen in Figure 17.

IP (init_permutation.vhd)							
6	14	22	30	38	46	54	62
4	12	20	28	36	44	52	60
2	10	18	26	34	42	50	58
0	8	16	24	32	40	48	56
7	15	23	31	39	47	55	63
5	13	21	29	37	45	53	61
3	11	19	27	35	43	51	59
1	9	17	25	33	41	49	57

Figure 17: Initial Permutation.

It is important to note that in the initial permutation, no bits are dropped because this is the actual data that is encrypted. If we set the data equal to 012345679ABCDEF, in hex, as stated earlier, the initial permutation yields the following string of bits:

[1100110000000000110011001111111111110000101010101111000010101010]

The permuted data is split into the left and right 32-bit strings. These strings of bits are 32-bit long strings and we label them as L0 and R0 respectively. L0 and R0 are as follows:

L0 = [11001100000000001100110011111111]

R0 = [11110000101010101111000010101010]

The initial permutation VHDL code is found in the appendix section under init_permutation.vhd.

The Feistel function, denoted as f(Rn-1,kn) in Figure 18, is what makes the scheme nearly unbreakable. The Feistel function can be narrowed into 4 steps: Expansion Permutation, XOR with the sub key kn, Substitution/Transformation and P-Permutation.

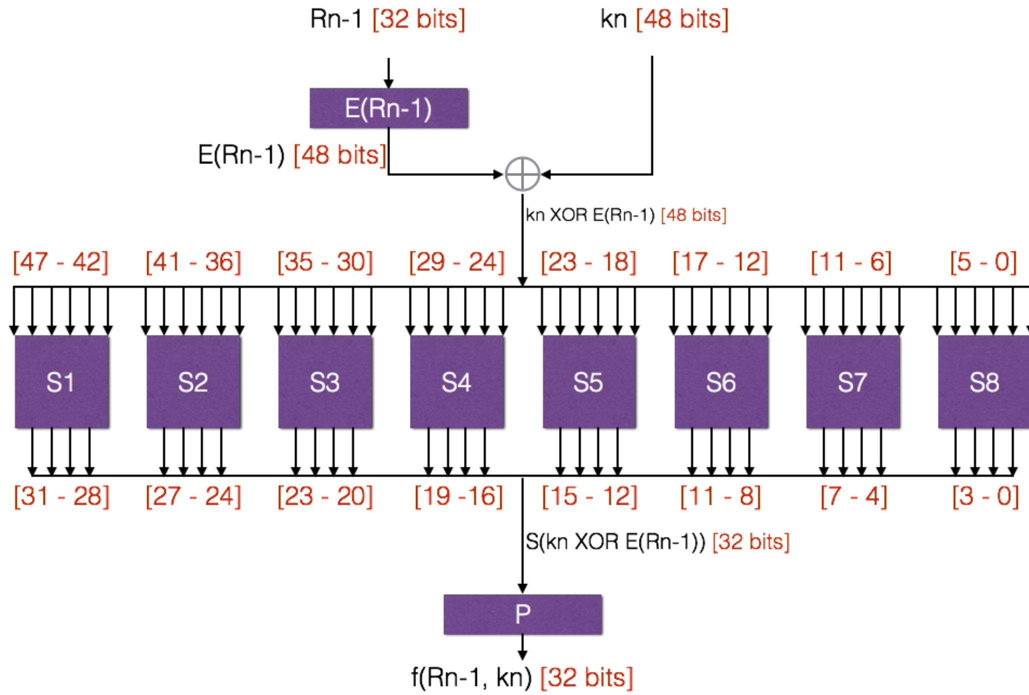


Figure 18: Feistel Function Design.

There are ten components in the Feistel Function Design: the expansion permutation, the permutation and eight substitution boxes. The Feistel function mapping in VHDL is provided in feistel_func.vhd in Appendix A.

The right side (Rn - 1), which is a 32-bit string, is expanded and permuted to 48 bits, E(Rn - 1). The expansion permutation is done as shown in Figure 19 and the VHDL code is under expan_perm.vhd in Appendix A.

E(Rn-1) expan_perm.vhd					
0	31	30	29	28	27
28	27	26	25	24	23
24	23	22	21	20	19
20	19	18	17	16	15
16	15	14	13	12	11
12	11	10	9	8	7
8	7	6	5	4	3
4	3	2	1	0	31

Figure 19: Expansion Permutation.

The expansion permutation stage results in a 48-bit string. The XOR operation is applied to this string along with its corresponding sub key kn. The expansion permutation stage and the XOR operation with the sub key kn is denoted as "kn XOR E(Rn-1)."

The result of the XOR is a 48-bit string that is divided into 8-6 bits strings as seen in Figure 18. The 6-bit strings are fed to the boxes S1 through S8. Every box has a 6-bit input and a 4-bit output. Lets take as an example the S1 Box. The six incoming bits are bits 47, 46, 45, 44, 43, 42. Bits 47 and 42 will determine the row in box 1 (box 1 is shown in Figure 20). Bits 46, 45, 44 and 43 will determine the column in box 1. The substitution box 1 is shown in Figure 20. E.g. if bits [47 42] = [0 0] and bits [46 45 44 43] = [1 1 0 0], then the chosen cell contains decimal value 5. 5 is [0101] in binary. Therefore [011000] will be replaced by [0101]. The similar scenario applies for all the eight substitution boxes. Figures 20 though 27 show the content of the remaining substitution boxes.

S1-Box Substitution																
	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
00	14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
01	0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
10	4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
11	15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

Figure 20: Substitution Box S1 [60].

S2-Box Substitution																
	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
00	15	1	8	14	6	11	3	4	9	7	2	13	12	0	5	10
01	3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	5
10	0	14	7	11	10	4	13	1	5	8	12	6	9	3	2	15
11	13	8	10	1	3	15	4	2	11	6	7	12	0	5	14	9

Figure 21: Substitution Box S2 [60].

S3-Box Substitution																
	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
00	10	0	9	14	6	3	15	5	1	13	12	7	11	4	2	8
01	13	7	0	9	3	4	6	10	2	8	5	14	12	11	15	1
10	13	6	4	9	8	15	3	0	11	1	2	12	5	10	14	7
11	1	10	13	0	6	9	8	7	4	15	14	3	11	5	2	12

Figure 22: Substitution Box S3 [60].

S4-Box Substitution																
	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
00	7	13	14	3	0	6	9	10	1	2	8	5	11	12	4	15
01	13	8	11	5	6	15	0	3	4	7	2	12	1	10	14	9
10	10	6	9	0	12	11	7	13	15	1	3	14	5	2	8	4
11	3	15	0	6	10	1	13	8	9	4	5	11	12	7	2	14

Figure 23: Substitution Box S4 [60].

S5-Box Substitution																
	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
00	2	12	4	1	7	10	11	6	8	5	3	15	13	0	14	9
01	14	11	2	12	4	7	13	1	5	0	15	10	3	9	8	6
10	4	2	1	11	10	13	7	8	15	9	12	5	6	3	0	14
11	11	8	12	7	1	14	2	13	6	15	0	9	10	4	5	3

Figure 24: Substitution Box S5 [60].

S6-Box Substitution																
	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
00	12	1	10	15	9	2	6	8	0	13	3	4	14	7	5	11
01	10	15	4	2	7	12	9	5	6	1	13	14	0	11	3	8
10	9	14	15	5	2	8	12	3	7	0	4	10	1	13	11	6
11	4	3	2	12	9	5	15	10	11	14	1	7	6	0	8	13

Figure 25: Substitution Box S6 [60].

S7-Box Substitution																
	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
00	4	11	2	14	15	0	8	13	3	12	9	7	5	10	6	1
01	13	0	11	7	4	9	1	10	14	3	5	12	2	15	8	6
10	1	4	11	13	12	3	7	14	10	15	6	8	0	5	9	2
11	6	11	13	8	1	4	10	7	9	5	0	15	14	2	3	12

Figure 26: Substitution Box S7 [60].

S8-Box Substitution																
	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
00	13	2	8	4	6	15	11	1	10	9	3	14	5	0	12	7
01	1	15	13	8	10	3	7	4	12	5	6	11	0	14	9	2
10	7	11	4	1	9	12	14	2	0	6	10	13	15	3	5	8
11	2	1	14	7	4	10	8	13	15	12	9	0	3	5	6	11

Figure 27: Substitution Box S8 [60].

Lets assume that the 48-bit string input to the S-boxes is [0110 0001 0001 0111 1011 1010 1000 0110 0110 0101 0010 0111]. Then the string will be divided as follows:

[011000 010001 011110 111010 100001 100110 010100 100111]

The result of the S-boxes will yield the following 32-bit string:

[0101 1100 1000 0010 1011 0101 1001 0111]

In this report, the resulting 32-bit string of the S-boxes is denoted as S(kn XOR E(Rn-1)). The implementation in VHDL of the all eight boxes can be found in Appendix A. Refer to the VHDL files S1.vhd, S2.vhd, S3.vhd, S4.vhd, S5.vhd, S6.vhd, S7.vhd and S8.vhd.

This 32-bit string S(kn XOR E(Rn-1)) will undergo another permutation (p-permutation) before it exists the Feistel function for the current round. The permutation is applied as shown in Figure 28.

P-Permutation							
16	25	12	11	3	20	4	15
31	17	9	6	27	14	1	22
30	24	8	18	0	5	29	23
13	19	2	26	10	21	28	7

Figure 28: Feistel Permutation

The implementation of the P-Permutation in VHDL and the update of Rn can be found in the permutation.vhd file in the appendix section of this report.

The result of this permutation we call f(kn,Rn-1), which is also the output of the Feistel function. In order to proceed to the next round, Ln and Rn must be updated. Rn = f(kn,Rn-1) XOR Ln-1. Ln = Rn-1. The new Rn and Ln are fed to the Feistel function in the next round. The sixteen rounds of the Feistel function yield sixteen left and right pair of 32-bit strings (L1, R1, L2, R2, L3, R3, . . . L16, R16). Continuing the example from above, the following left and right pairs, seen in Figure 29, are the results.

L1 = [11110000101010101111000010101010]
R1 = [11101111010010100110010101000100]

L2 = [11101111010010100110010101000100]
R2 = [11001100000000010111011100001001]

L3 = [11001100000000010111011100001001]
R3 = [10100010010111000000101111110100]

L4 = [10100010010111000000101111110100]
R4 = [01110111001000100000000001000101]

L5 = [01110111001000100000000001000101]
R5 = [10001010010011111010011000110111]

L6 = [10001010010011111010011000110111]
R6 = [11101001011001111100110101101001]

L7 = [11101001011001111100110101101001]
R7 = [00000110010010101011101000010000]

L8 = [00000110010010101011101000010000]
R8 = [11010101011010010100101110010000]

L9 = [11010101011010010100101110010000]
R9 = [00100100011111001100011001111010]

L10 = [00100100011111001100011001111010]
R10 = [10110111110101011101011110110010]

L11 = [10110111110101011101011110110010]
R11 = [11000101011110000011110001111000]

L12 = [11000101011110000011110001111000]
R12 = [01110101101111010001100001011000]

L13 = [01110101101111010001100001011000]
R13 = [00011000110000110001010101011010]

R14 = [00011000110000110001010101011010]
R14 = [11000010100011001001011000001101]

L15 = [11000010100011001001011000001101]
R15 = [01000011010000100011001000110100]

L16 = [01000011010000100011001000110100]
R16 = [00001010010011001101100110010101]

Figure 29: Left and Right 16 Round Outputs

Once the 16th round is executed, the final permutation, already discussed in Figure 16, is applied to the concatenation of L16 and R16 and the result is the 64-bit cipher data. The permutation is applied as seen in Figure 30.

FP (inv_init_permutation.vhd)							
24	56	16	48	8	40	0	32
25	57	17	49	9	41	1	33
26	58	18	50	10	42	2	34
27	59	19	51	11	43	3	35
28	60	20	52	12	44	4	36
29	61	21	53	13	45	5	37
30	62	22	54	14	46	6	38
31	63	23	55	15	47	7	39

Figure 30: Final Permutation

The final result is a 64-bit string that we call cipher data. In our example, where the key is 133457799BBCDFF1 and the data is 012345679ABCDEF, the final permutation yields the cipher data to be 85E813540F0AB405 in hex. The binary representation of the cipher data is:

cipher data = [1000 0101 1110 1000 0001 0011 0101 0100 0000 1111 0000 1010 1011 0100 0000 0101]

The implementation of the final permutation in VHDL is achieved in the file inv_init_permutation.vhd that can be found in Appendix A.

2.2 DES Non-Pipelined Design For Decryption

Our DES decryption non-pipelined design is very similar to the encryption design. The main difference is key scheduling process. The 16 Feistel rounds, the initial permutation and final permutation remain unchanged. The specifications for the DES non-pipelined encryption design are the following:

- The inputs to the top-level component are a 64-bit Key and 64-bit Cipher.

- The output to the top-level component is a 64-bit Data.
- The cost must be less than 33,216 Logic Elements.

2.2.1 DES Key Schedule Component For Encryption

The first part of the decryption algorithm involves scheduling 16 48-bit sub keys from the 64-bit key. These sub keys are supposed to be introduced in reverse order, into the heart of the DES algorithm, if the keys are generated as discussed in Section 2.1.1. As seen in Figure 31, the key scheduling performs right rotations instead. The specifications for the DES key schedule component for encryption are the following:

- The input to the top-level key schedule component is the 64-bit Key.
- The outputs are 16-sub keys k1, k2, k3, …, k16.

There are four components in the top key schedule component. Components PC_1 and PC_2 are discussed in Section 2.1.1.1. However, Right Rotate 1 and Right Rotate 2 replace the Left Rotate components. Refer to Appendix section to view the VHDL code of the top key schedule component for decryption DES_decrypt.vhd, the Right Rotate 1 component SRR_1.vhd and the Right Rotate 2 component SRR_2.vhd. Implementing right rotates in the decryption key schedule maintains a coherent mapping between the 16 sub keys and the 16 rounds in DES. This consistency makes it simple to implement our DES Pipelined decryption design seen in Section 3.1.2.

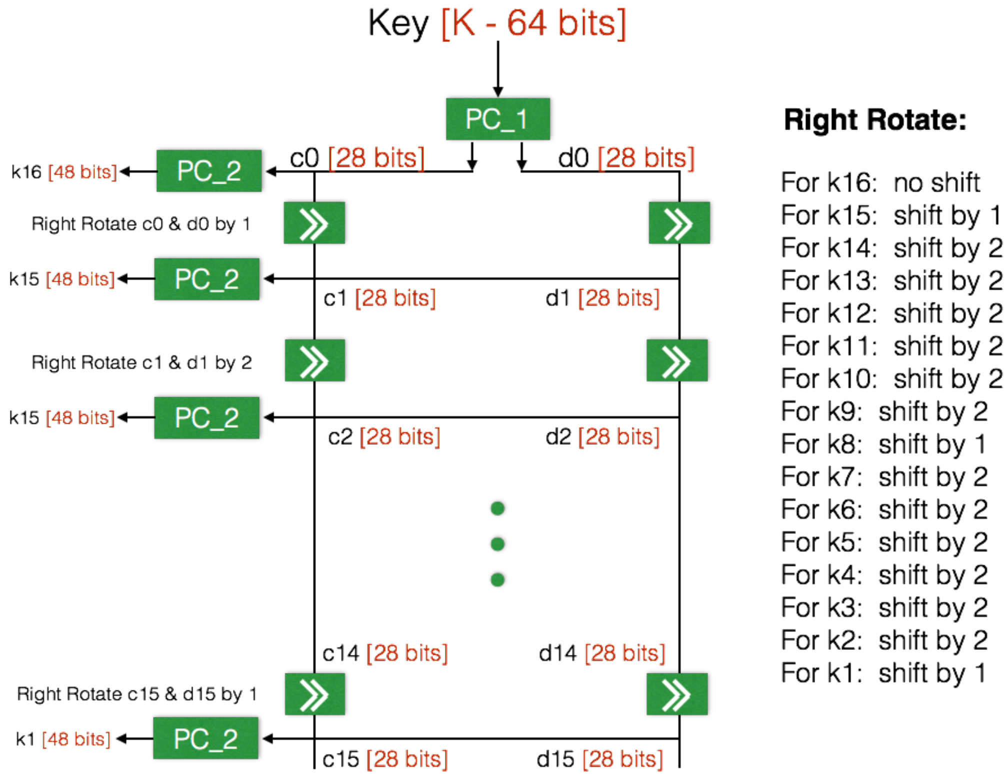


Figure 31: Top Key Schedule Component For DES Decryption.

It is important to note that the amount by which cn and dn right rotate is also changed as seen in Figure 31. The 16 48-bit sub keys generated with the decryption key scheduler are shown bellow. The decryption key schedule VHDL code is in Appendix A under KS_D.vhd.

k16 = [110010110011110110001011000011100001011111110101]

k15 = [101111111001000110001101001111010011111100001010]

k14 = [010111110100001110110111111100101110011100111010]

k13 = [100101111100010111010001111110101011101001000001]

k12 = [011101010111000111110101100101000110011111101001]

k11 = [001000010101111111010011110111101101001110000110]

k10 = [101100011111001101000111101110100100011001001111]

k9 = [111000001101101111101011111011011110011110000001]

k8 = [111101111000101000111010110000010011101111111011]

k7 = [111011001000010010110111111101100001100010111100]

k6 = [011000111010010100111110010100000111101100101111]

k5 = [011111001110110000000111111010110101001110101000]

k4 = [011100101010110111010110110110011010100011101]

k3 = [010101011111110010001010010000101100111110011001]

k2 = [011110011010111011011001110110111100100111100101]

k1 = [000110110000001011101111111111000111000001110010]

2.2.2 DES Decryption Initial Permutation, Final Permutation and 16 Rounds

Figure 32 depicts the top component in our DES decryption design. The DES decryption component is found in the appendix section under DES_decrypt.vhd. As seen in Figure 30, the 16 Feistel rounds, the initial permutation and the final permutation remain unchanged. The only change is the order in which the keys are fed in each round. The keys are fed in the reverse order from the way they were fed in the encryption process. Following the previous example, if the input Cipher Data is 85E813540F0AB405 in hex, then the output Data is 012345679ABCDEF in hex.

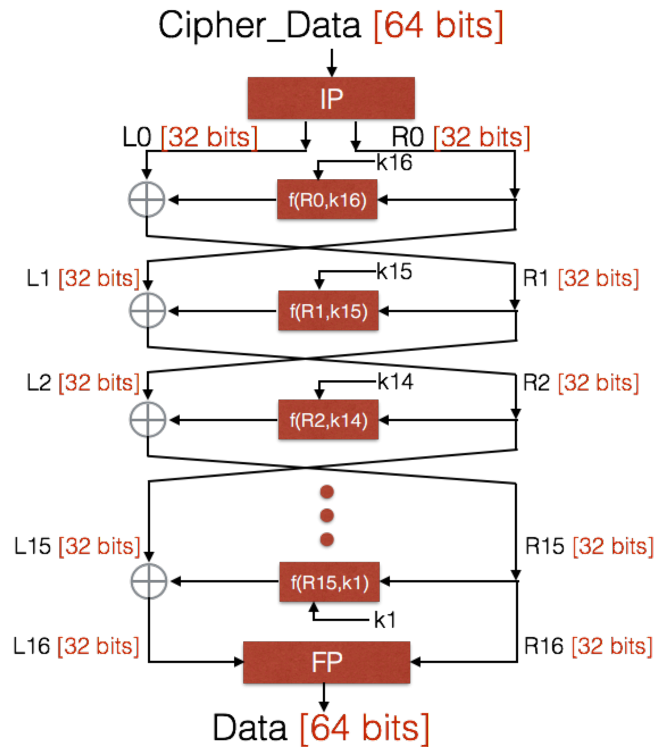


Figure 32: DES Decryption Top Component.

2.3 TDES Non-Pipelined Design For Encryption and Decryption

We present the TDES algorithms as described in [58]. The TDES top component for encrypting consists of three linked DES components as seen in Figure 33. The specification of the TDES encryption design must be the following:

- The inputs to the top-level component are 3 64-bit keys and 64-bit Data.
- The output to the top-level component is a 64-bit Cipher Data.
- The cost must be less than 33,216 Logic Elements.

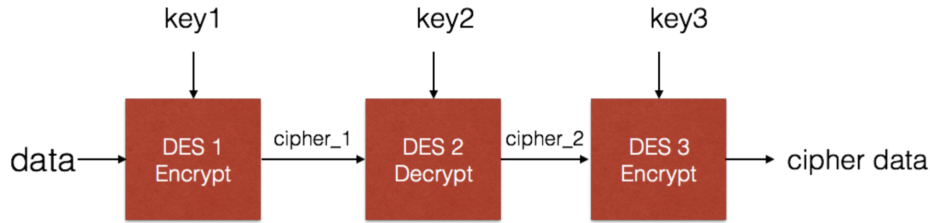


Figure 33: TDES Non-Pipelined Encryption.

The TDES encryption design is composed of a DES encryption component whose output is fed to a second DES decryption component whose output is fed to third DES encryption component. DES 1 is encrypted using key1, DES 2 is decrypted using key2 and DES 3 is encrypted using key 3. The output of DES 3 is the cipher data. The VHDL code, TDES_encrypt.vhd, for our TDES encryption component is in the Appendix A section. Similarly, the TDES top component for decrypting consists of three linked DES components as seen in Figure 34. The specification of the TDES decryption design must be the following:

- The inputs to the top-level component are 3 64-bit keys and 64-bit Cipher.
- The output to the top-level component is a 64-bit Data.
- The cost must be less than 33,216 Logic Elements.

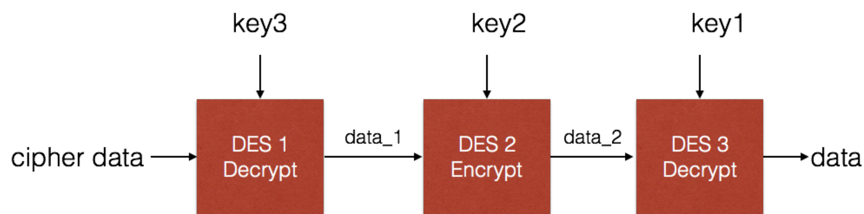


Figure 34: TDES Non-Pipelined Decryption.

The TDES decryption design is composed of a DES decryption component whose output is fed to a second DES encryption component whose output is fed to third DES decryption component. DES 1 is decrypted using key3, DES 2 is encrypted using key2 and DES 3 is

decrypted using key 1. The output of DES 3 is the data. The VHDL code, TDES_decrypt.vhd, for our TDES decryption component is in the Appendix A section.

An important note, made in [62], is that certain keys reduce the security provided by DES and TDES. The following keys, in hex, are considered weak keys:

- 01010101 01010101
- FEFEFEFE FEFEFEFE
- E0E0E0E0 F1F1F1F1
- 1F1F1F1F 0E0E0E0E

There are also 6 keys considered semi-weak and 48 possibly weak keys. These keys either generate 16 identical sub-keys or generate 4 distinct sub-keys. The outcome of the encryption and decryption, by using these keys, is identical. These keys should be avoided. The list of all the weak keys, semi-weak keys and possibly weak keys is found in [62].

CHAPTER III

DES & TDES PIPELINED DESINGS IN VHDL

In Chapter II we presented our non-pipelined designs. In this chapter, we continue optimizing the performance of the designs by means of pipelining. Without pipelining, the frequency at which the input is fed to the algorithm must be lower than the frequency at which the input propagates the entire logic of the scheme. By pipelining our design, we don't have to wait until the input propagates the entire logic to feed the next input. We only wait until the input reaches the next stage. In general, pipelining reduces the propagation time from one stage to the next. Increasing the depth of the pipeline decreases the propagation delay between stages. A pipeline stage in the design is achieved by applying a buffer, also known as a set of memory elements or registers, at the place in logic where the stage is desired. Increasing the pipeline depth increases the hardware cost by increasing the number of registers. Further information on pipelining can be obtained in [63].

To pipeline our designs we take advantage of the 16 Feistel function rounds in DES. We pipeline after every Feistel function round. The pipeline is also applied to the key schedules presented in Chapter 2. The pipelined depth of our DES design is 16 stages and the depth of our TDES design is 48 stages. This chapter is organized as follows: In Section 3.1 and 3.2 we present our DES Pipelined designs and in Section 3.3 we present our TDES pipelined designs. We also present a key bank that buffers the 3 input keys and feeds them to each DES component in the proper clock cycle.

In these sections we include our encryption scheme, decryption scheme and the specifications that our designs must meet. The VHDL code for each component in the encryption and decryption designs is referenced in the text and provided in Appendix A.

3.1 DES Pipelined Design For Encryption

The main change from the non-pipelined to the pipelined architecture is the buffers after every Feistel function round. The specifications for the DES pipelined encryption design are the following:

- The inputs to the top-level component are a 64-bit Key and 64-bit Data.
- The output to the top-level component is a 64-bit Cipher Data.
- The pipeline depth must be 16 stages and coherent to the 16 sub keys.
- The pipeline depth of the key scheduler must be 15 stages.
- The time propagations must meet the requirement for a 50 MHz clock.
- The cost must be less than 33,216 Logic Elements

3.1.1 DES Key Schedule Component For Encryption

To implement our encryption pipelined design, our DES non-pipelined design must incur two mayor modifications. The first modification concerns the key scheduler. The specifications for the key scheduler are the following:

- The input to the top-level key schedule component is the 64-bit Key.
- The outputs are 16-sub keys k1, k2, k3, …, k16.
- The pipeline depth of the key scheduler must be 15 stages.
- The time propagations must meet the requirement for a 50 MHz clock

Figure 35 shows the modification made to the key schedule component.

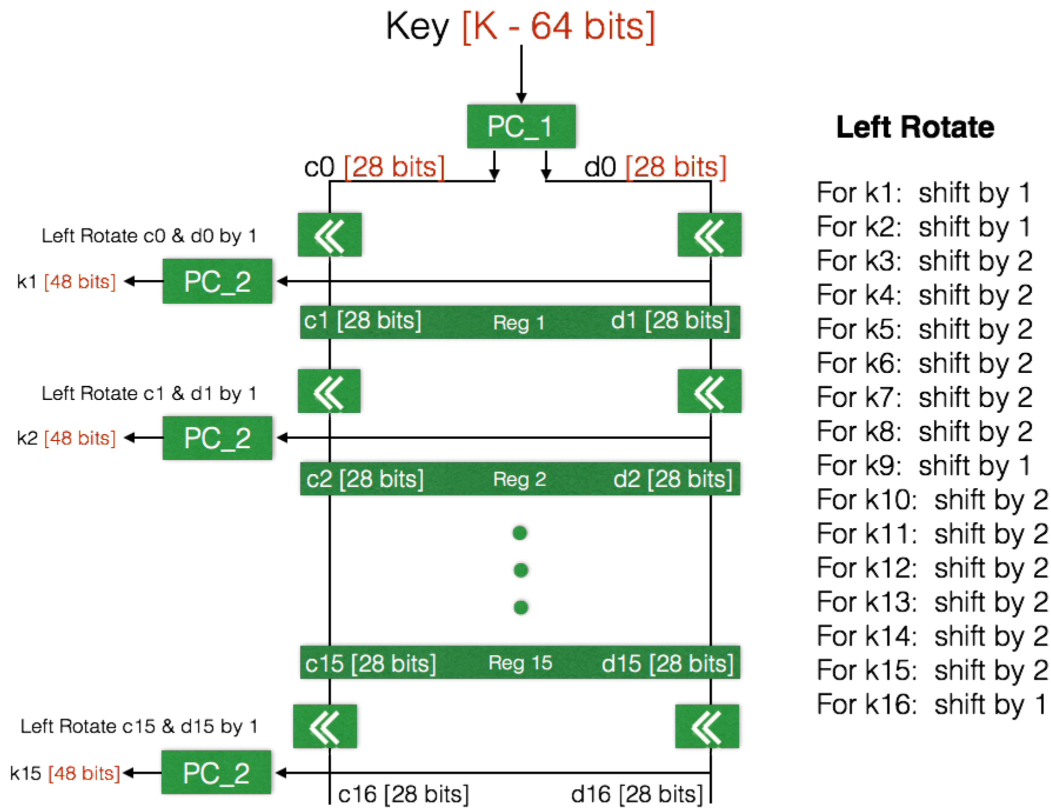


Figure 35: Key Schedule Pipelined Design for Encryption.

As seen in Figure 35, we pipelined the key schedule component by implementing 15 48-bit registers. We named these reg1, reg2, …, reg15. The frequency requirement is discussed in Chapter 5. Refer to Appendix A to view the VHDL code of the pipelined key schedule encryption design under KS_E_P.vhd.

3.1.2 16 Stages In The DES Encryption Pipeline

The second mayor modification is done on the top component in DES where the 16 Feistel function rounds are executed. Figure 36 shows the modifications made to the DES Encryption design.

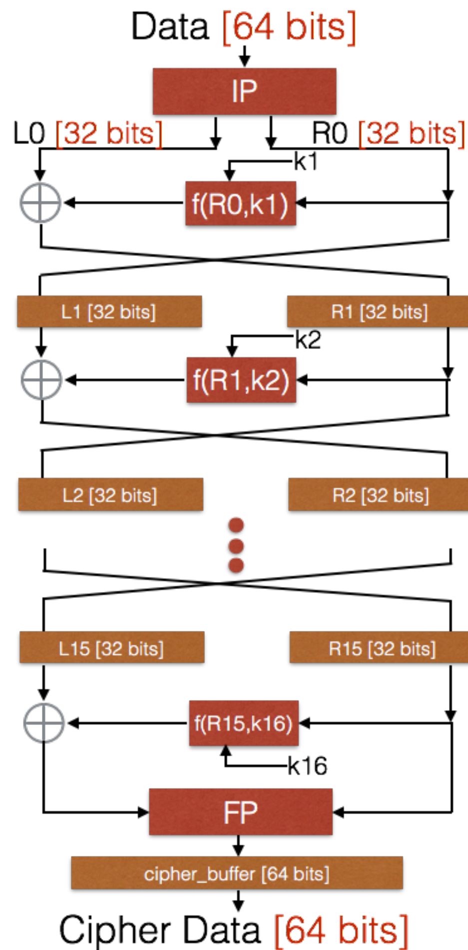


Figure 36: DES Pipelined Encryption Top Component.

As seen in Figure 36, we achieve the 16-stage pipeline by implementing 30 32-bit registers and one 64-bit buffer. These registers are L1 through L15 and R1 through R15. The buffer is cipher_buffer. The 16 sub keys are properly mapped to the 16 Feistel function rounds. The frequency requirement is discussed in Chapter 5. Refer to Appendix A to view the VHDL implementation of the pipelined DES encryption component under DES_encrypt_P.vhd.

3.2 DES Pipelined Design For Decryption

The specifications for the DES pipelined decryption design are the following:

- The inputs to the top-level component are a 64-bit Key and 64-bit Cipher.
- The output to the top-level component is a 64-bit Data.
- The pipeline depth must be 16 stages and coherent to the 16 sub keys.
- The pipeline depth of the key scheduler must be 15 stages.
- The time propagations must meet the requirement for a 50 MHz clock.
- The cost must be less than 33,216 Logic Elements

3.2.1 DES Key Schedule Component For Encryption

To implement our encryption pipelined design, our DES non-pipelined design must incur two mayor modifications. The first modification concerns the key scheduler. The specifications for the key scheduler are the following:

- The input to the top-level key schedule component is the 64-bit Key.
- The outputs are 16-sub keys k1, k2, k3, …, k16.
- The pipeline depth of the key scheduler must be 15 stages.
- The time propagations must meet the requirement for a 50 MHz clock

Figure 37 shows the modification made to the key schedule component.

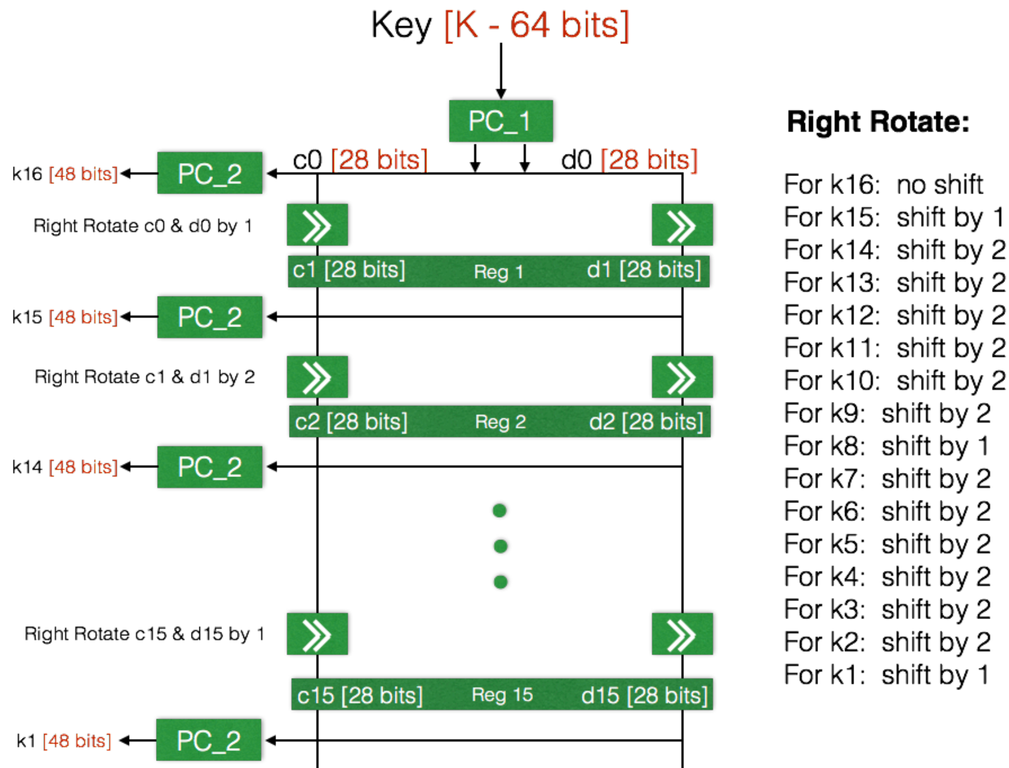


Figure 37: Key Schedule Pipelined Design for Decryption.

As seen in Figure 37, we pipelined the key schedule component by implementing 15 48-bit registers. We named these reg1, reg2, …, reg15. The left rotations, on cn and dn, as discussed in Chapter 2, allow the coherent mapping of the key to the proper Feistel round. The frequency requirement is discussed in Chapter 5. Refer to Appendix A to view the VHDL code of the pipelined key schedule encryption design under KS_D_P.vhd.

3.2.2 16 Stages In The DES Decryption Pipeline

The pipelined DES decryption top component remains the same as the component presented in 3.1.2. The only difference is the order in which the sub keys are fed to the 16 Feistel function rounds. See Figure 38. The order of the sub keys is discussed in Section 3.2.1.

The frequency requirement is discussed in Chapter 5. Refer to Appendix A to view the VHDL implementation of the DES decryption component under DES_decryption_P.vhd.

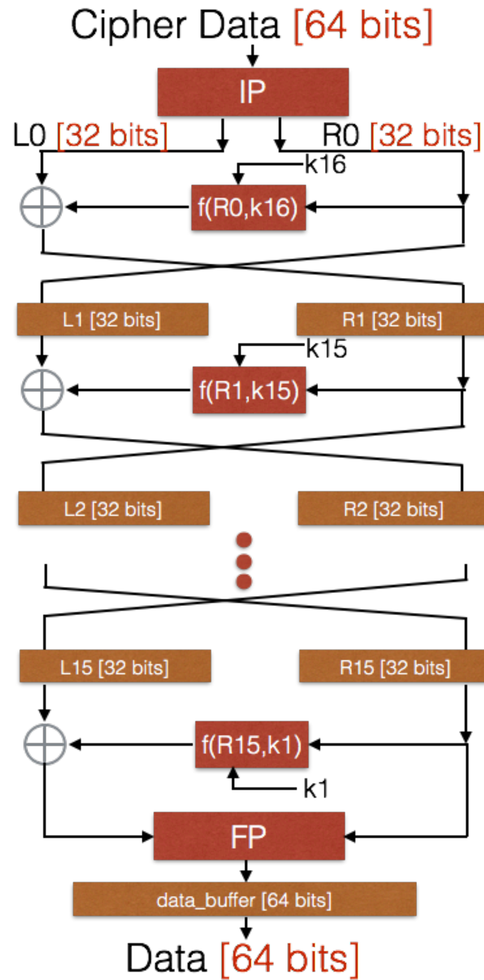


Figure 38: DES Pipelined Decryption Top Component.

3.3 TDES Pipelined Design For Encryption and Decryption

The specifications for the TDES pipelined encryption design must be the following:

- The inputs to the top-level component are 3 64-bit keys and 64-bit Data.
- The output to the top-level component is a 64-bit Cipher Data.
- The pipeline depth must be 48 stages.
- The time propagations must meet the requirement for a 50 MHz clock.

- The cost must be less than 33,216 Logic Elements

To implement our pipelined TDES encryption design, first we link 3 Pipelined DES components as shown in Figure 39.

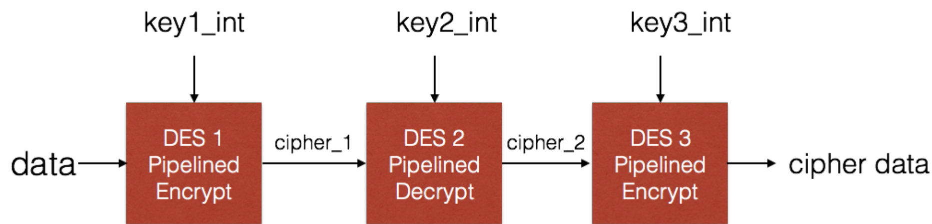


Figure 39: DES Components Linked To Implement The TDES Encryption.

From Figure 39 we see that it takes 16 cycles for Data to reach Data1, the output of DES 1 Pipelined Encrypt. It takes 32 cycles for Data to reach Data 2 and it takes 48 cycles to reach Cipher Data. Key1, Key2 and Key3 must be appropriately buffered before they are fed to their respective DES component, otherwise, Key2 and Key3 would be encrypting the incorrect input. Key1 is fed into DES 1 along with Data and therefore no buffering is needed for Key1. However Key2 must wait (be buffered) 15 cycles before it is fed to DES 2. Key3 must be buffered 31 cycles before it is fed to DES 3. This ensures that Data is processed with the correct key at the 3 stages of DES. Figure 40 shows how we buffer Key2 and Key3 in the key bank. Refer to Appendix A to view the VHDL implementation of the key bank component under key_bank.vhd. The frequency requirements and the timing analysis are discussed in Chapter 5.

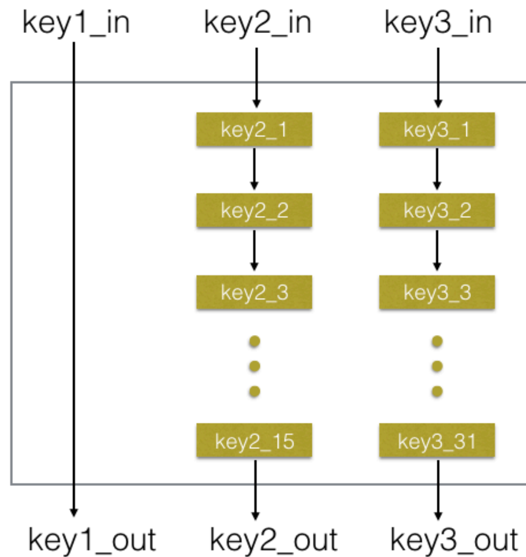


Figure 40: Key Bank Component For The TDES Design.

Figure 41 shows the mapping of the components TDES and the key bank.

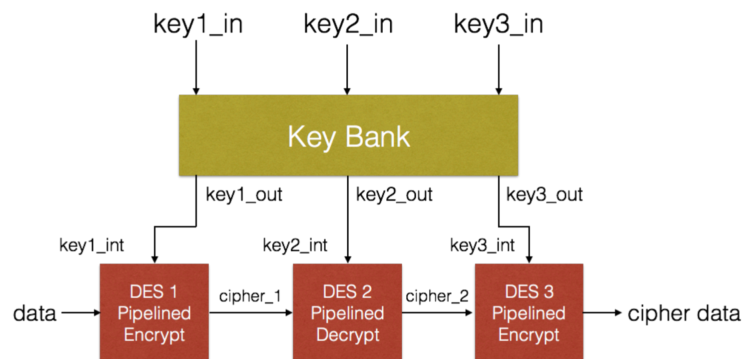


Figure 41: TDES And Key Bank Top Component Mapping For Encryption.

Refer to Appendix A to view the VHDL code of the TDES Encryption top component TDES_encrypt_P.vhd.

The TDES decryption is very similar to the encryption. To implement the decryption, the encrypting processes must be reversed. The keys must be inserted starting with the third key,

second key and first key. Figure 42 shows the map of the 3 DES components. The specifications for the TDES pipelined decryption design must be the following:

- The inputs to the top-level component are 3 64-bit keys and 64-bit Data.
- The output to the top-level component is a 64-bit Cipher Data.
- The pipeline depth must be 48 stages.
- The time propagations must meet the requirement for a 50 MHz clock.

  The cost must be less than 33,216 Logic Elements

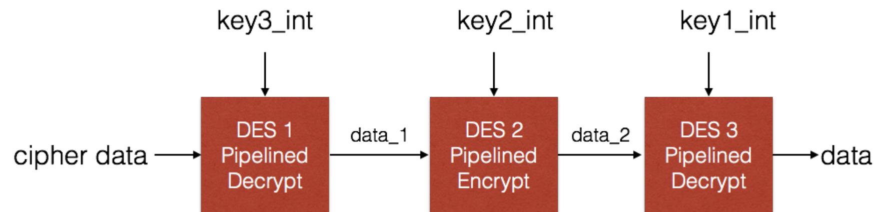


Figure 42: DES Components Linked To Implement The TDES Decryption.

Figure 43 shows the mapping of the components TDES and the key bank.

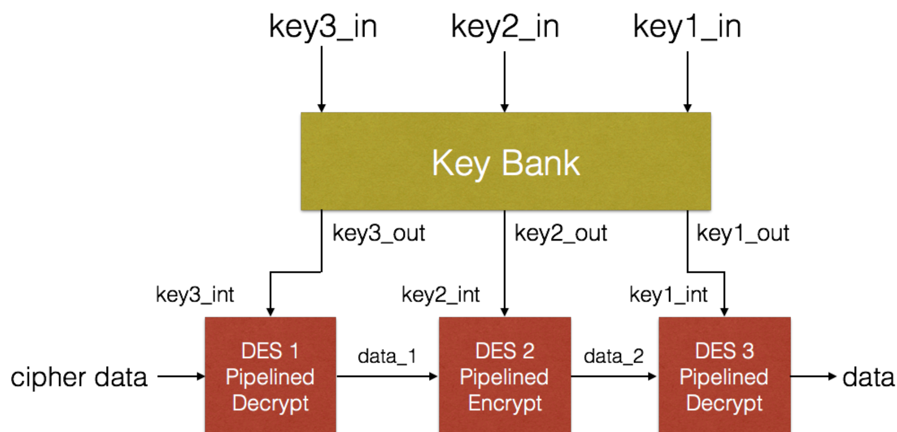


Figure 43: TDES And Key Bank Top Component Mapping For Decryption.

Refer to Appendix A to view the VHDL code of the TDES Decryption top component under TDES_decrypt_P.vhd. The frequency requirements and the timing analysis are discussed in Chapter 5.

CHAPTER IV

ECB MODE OF OPERATION VALIDATION SYSTEM FOR 3DES & DES

DES and TDES can be implemented in several modes: Electronic Codebook (ECB), Cipher Block Chaining (CBC), Cipher Feedback (CFB) and the Output Feedback (OFB). The Electronic Codebook (ECB) mode of operation validation for DES and 3DES is presented in this chapter. We also present the validation results for ECB mode. ECB mode only requires a key and the data as inputs. This mode of operation does not require additional input vector(s). ECB, being a non-feedback operation mode, does not require the scheme to perform a full cryptographic operation before proceeding to the next operation. This allows us to validate the non-pipelined and pipelined schemes presented in Chapter 3.

The validation, in this chapter, is presented as described in the National Institute of Standards and Technology (NIST) special publications 800-17 and 800-20 Modes of Operation Validation Systems (MOVS): Requirements and Procedures. "The MOVS is designed to perform automated testing on Implementations Under Test (IUTs)" [26] [27]. We implement the following six tests to validate our design:

- Test 1: Variable Plaintext KAT (Known Answer Test)
- Test 2: Inverse Permutation KAT
- Test 3: Permutation Operation KAT for Encryption/Decryption
- Test 4: Permutation Operation KAT for Encryption/Decryption

- Test 5: Substitution table KAT for Encryption/Decryption
- Test 6: Modes Test for Encrypt/Decrypt (MONTECARLO)

Each test is designed to validate a specific portion of the algorithm. E.g. the Variable Plaintext KAT validates the Initial and Final permutations of the schemes. It is important to note that in this work, we implement a slight variation of Tests 2 and 6. We discuss these variations in sections 4.3 and 4.6. respectively.

Figure 44 shows the setup for validating the ECB mode of operation for the encryption process.

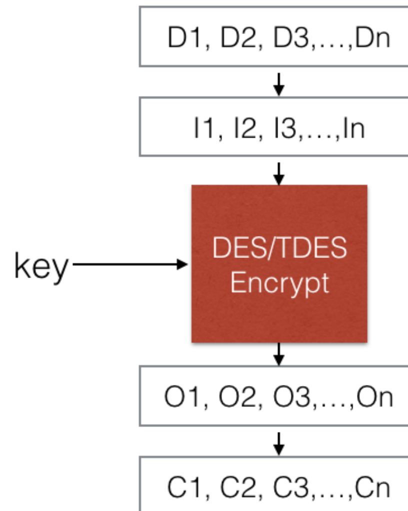


Figure 44: ECB Mode of Operation for DES/TES Encryption

In ECB mode of operation for encryption, the data block is defined as D0, D1, D2,…,Dn and the input block is defined as I0, I1, I2,…,In. The data block is directly used as the input block, which means that D0, D1, D2,…,Dn = I0, I1, I2,…,In. The input block is encrypted using the encryption scheme. The output block of the encryption scheme is defined as the O0, O1, O2,…,On and is the direct cipher block defined as C0, C1, C2,…,Cn. The ECB mode is straightforward. No additional input vectors are required and no feedback configuration is needed.

Figure 45 shows the setup for validating the ECB mode of operation for the decryption process. The setup is very similar to the encryptions setup. No additional input vectors and a feedback configuration are needed.

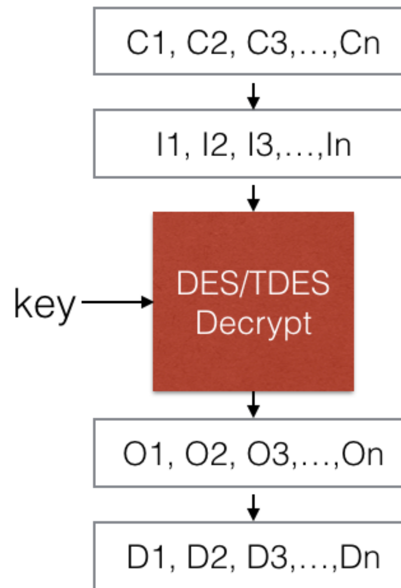


Figure 45: ECB Mode of Operation for DES/TDES Decryption

The organization of this chapter is presented by each validation test mentioned above. For each validation test section we overview the test scheme and present the results obtained using the EDA tool, Altera U.P. Simulator. For each test, we only show the results for our pipelined schemes to prevent this document from populating with many redundant figures. We are more concerned with validating the pipelined schemes due to the timing violations and delays. The only change made from the non-pipelined to the pipelined designs, are the addition of registers. If our pipelined designs successfully passed the validation tests, so will our non-pipelined designs.

4.1 Variable Plaintext KAT

The Variable Plaintext KAT consists on setting the key to zero and maintaining this value

though out the encryption of all the 64 data input vectors. The data consists of 64-64 bit vectors, provided in the lookup Table 5, and each vector is encrypted utilizing the initialized key. The standard definition of a vector applies here. One of the 64 bits is set to 1 and all other bits are set to zero.

Table 5: Variable Plaintext KAT for Encryption Cipher Results

Resulting Cipher Text for DES/TDES Variable Plaintext KAT	
key = 0x0000000000000000	
Plain Text (Data)	Cipher Text
D0 = 0x8000000000000000	C0 = 0x9A90BC0B75C73703
D1 = 0x4000000000000000	C1 = 0xCC6843598C732BBE
D2 = 0x2000000000000000	C2 = 0x1372953509B3C14C
D3 = 0x1000000000000000	C3 = 0x70AAAA8418E48930
.	.
.	.
.	.
D60 = 0x0000000000000008	C60 = 0x96B491C1FE443E9A
D61 = 0x0000000000000004	C61 = 0xD0E014CFEE94589D
D62 = 0x0000000000000002	C62 = 0x0B9E44B537AF2879
D63 = 0x0000000000000001	C63 = 0x22F428E3EC491E60

As seen in Table 5, encrypting the 64-64 bit data vectors with key 0x0000000000000000 yields 64-64 bit cipher text results. In Table 5 we show a few results, but the NIST publications 800-17 and 800-20 contain the list of all the 64 cipher results. In Figures 46 and 47 show partial cipher data results of Test 1 for our DES and TDES designs.

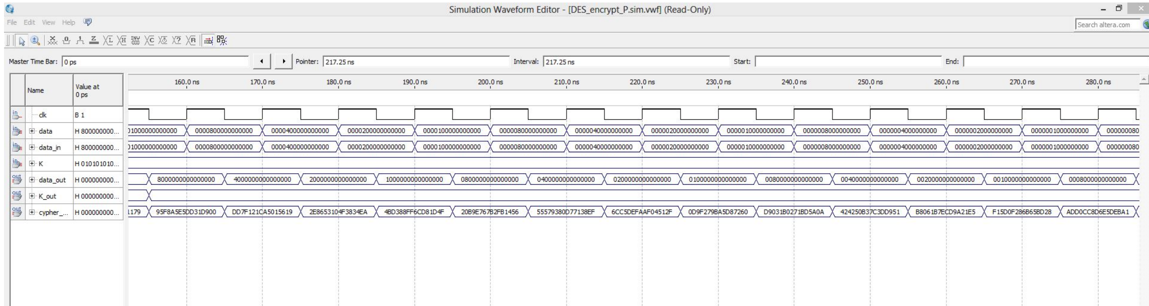


Figure 46: DES Pipelined Encryption MOV Test 1

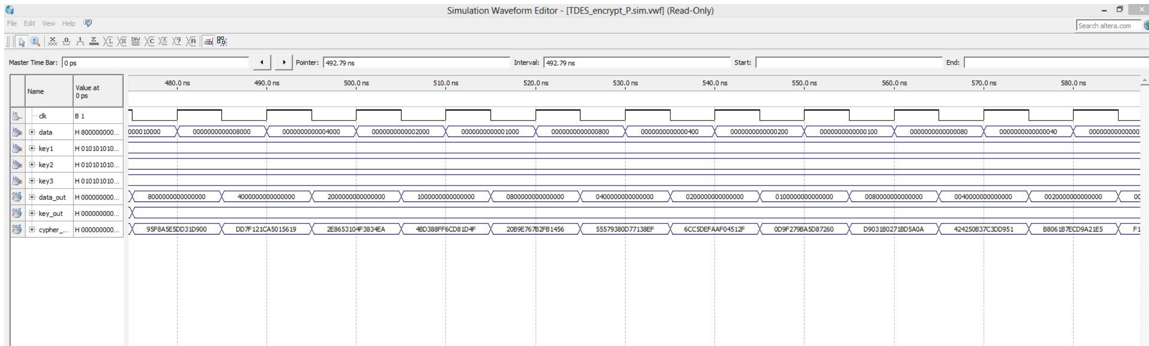


Figure 47: TDES Pipelined Encryption MOV Test 1 Part 2

As seen in Figures 46 and 47, the cipher results match the expected ciphers from Table 5. MOV Test 1 passed.

4.2 Inverse Permutation KAT for Encryption

The Inverse Permutation KAT consists on setting the key to zero and maintaining this value though out the encryption of all the 64 data inputs. The input data consists of the 64 cipher texts generated from the Variable Plaintext KAT.

Table 6: Inverse Permutation KAT for Encryption Cipher Results

Resulting Cipher Text for DES/TDES Variable Plaintext KAT	
key = 0x0000000000000000	
Plain Text (Data)	Cipher Text
D0 = 0x9A90BC0B75C73703	C0 = 0x8000000000000000
D1 = 0xCC6843598C732BBE	C1 = 0x4000000000000000
D2 = 0x1372953509B3C14C	C2 = 0x2000000000000000
D3 = 0x70AAAA8418E48930	C3 = 0x1000000000000000
.	.
.	.
.	.
D60 = 0x96B491C1FE443E9A	C60 = 0x0000000000000008
D61 = 0xD0E014CFEE94589D	C61 = 0x0000000000000004
D62 = 0x0B9E44B537AF2879	C62 = 0x0000000000000002
D63 = 0x22F428E3EC491E60	C63 = 0x0000000000000001

As seen in Table 6, encrypting the 64-64 bit data inputs with key 0x0000000000000000 yield 64-64 bit vectors. In Table 6 we only show a few results but NIST publications 800-17 and 800-20 contain the full results table. Figures 48 and 49 show partial cipher data results of Test 2 for our DES and TDES designs.

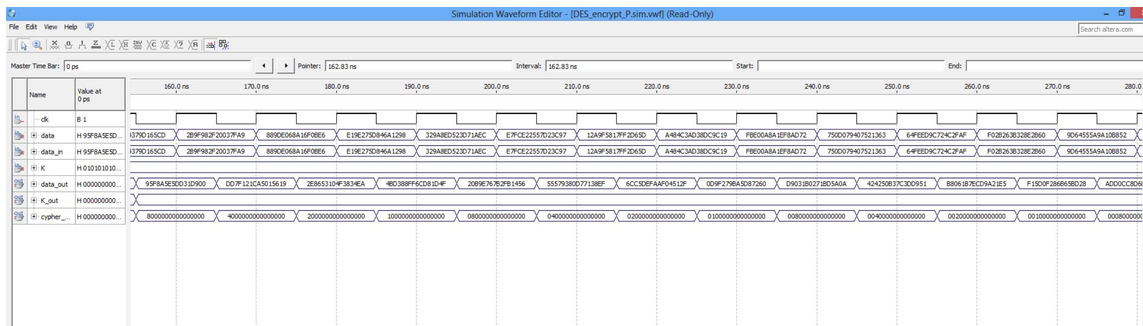


Figure 48: DES Pipelined Encryption MOV Test 2

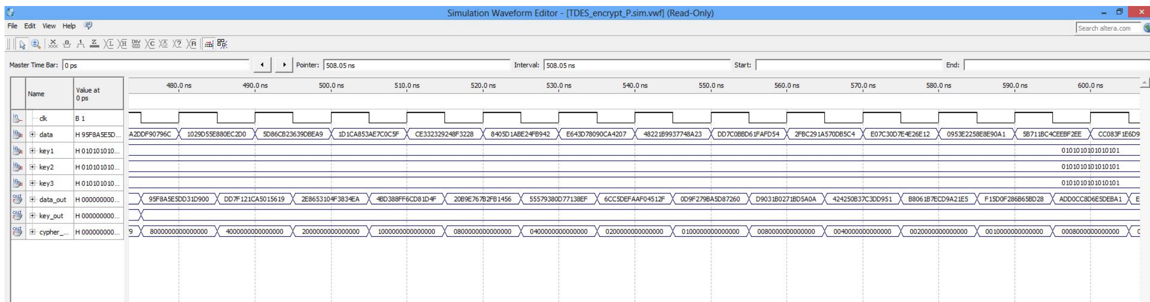


Figure 49: TDES Pipelined Encryption MOV Test 2 Part 2

As seen in Figures 5 and 6, the cipher results match the expected ciphers from Table 6. MOV Test 2 passed.

4.3 Variable Key KAT

The Variable Key KAT consists on setting the input data to zero and maintaining this value though out the encryption with all the 64 key input vectors. It is not necessary to input 64 key vectors. Since 8 bits in the key are used as parity, the key vectors containing a value vector in the parity can be discarded. This yields in 56 keys that produce 56 different ciphers. Encrypting with key vectors, whose value is in the parity, produce the same cipher as encrypting with key 0x0000000000000000. For the validation of Test 3, we encrypt using all 64 key vectors.

Table 7: Variable Key KAT for Encryption Cipher Results

Resulting Cipher Text for DES/TDES Variable Plaintext KAT	
D0,…,D63 = 0x0000000000000000	
Plain Text (Data)	Cipher Text
key0 = 0x8000000000000000	C0 = 0x95A8D72813DAA94D
key1 = 0x4000000000000000	C1 = 0x0EEC1487DD8C26D5
key2 = 0x2000000000000000	C2 = 0x7AD16FFB79C45926
key3 = 0x1000000000000000	C3 = 0xD3746294CA6A6CF3
.	.
.	.
.	.
key60 = 0x0000000000000008	C60 = 0x5A594528BEBEF1CC
key61 = 0x0000000000000004	C61 = 0xFCDB3291DE21F0C0
key62 = 0x0000000000000002	C62 = 0x869EFD7F9F265A09
key63 = 0x0000000000000001	C63 = 0x8CA64DE961B123A7

As seen in Table 7, encrypting the data 0x0000000000000000 with the 64 key vectors yield corresponding cipher text results. NIST publications 800-17 and 800-20 contain the full cipher text results table. The tables in publications 800-17 and 800-20 only contain 56 ciphers because the ciphers generated with keys containing a parity bit as the vector are not taken into account. As mentioned before, 8 bits are removed from the key for parity purposes. Figures 50 and 51 show partial cipher data results of Test 3 for our DES and TDES designs.

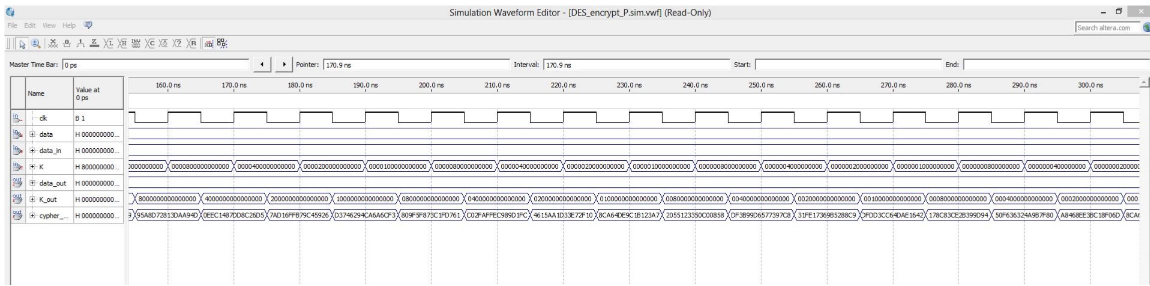


Figure 50: DES Pipelined Encryption MOV Test 3

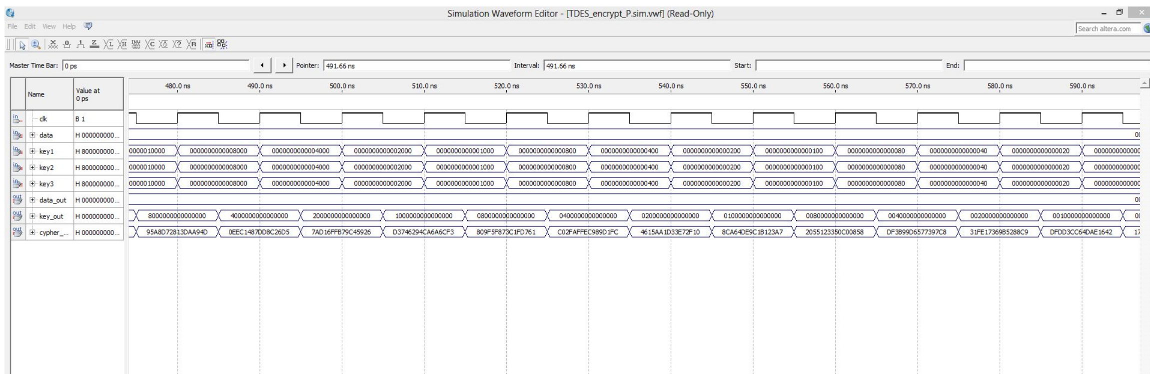


Figure 51: TDES Pipelined Encryption MOV Test 3 Part 2

As seen in Figures 50 and 51, the cipher results match the expected ciphers from Table 7. MOV Test 3 passed.

4.4 Permutation Operation KAT for Encryption

The Permutation Operation KAT for Encryption consists on setting input data to zero and maintaining this value though out the encryption with all the 32 different keys seen in Table 8.

Table 8: Permutation Operation KAT for Encryption Cipher Results

Resulting Cipher Text for DES/TDES Variable Plaintext KAT	
D0,…,D63 = 0x0000000000000000	
Plain Text (Data)	Cipher Text
key0 = 0x1046913489980131	C0 = 0x88D55E54F54C97 B4
key1 = 0x1007103489988020	C1 = 0x0C0CC00C83EA48 FD
key2 = 0x10071034C8980120	C2 = 0x83BC8EF3A6570183
key3 = 0x1046103489988020	C3 = 0xDF725DCAD94EA2E9
.	.
.	.
.	.
Key28 = 0x1002911598100104	C28 = 0xB3E35A5EE53E7B8D
Key29 = 0x1002911598190104	C29 = 0x61C79C71921A2EF8
Key30 = 0x1002911598100201	C30 = 0xE2F5728F0995013C
Key31 = 0x1002911698100101	C31 = 0x1AEAC39A61F0A464

As seen in Table 8, encrypting the data 0x0000000000000000 with the 32 different keys yields 32 cipher text results. NIST publications 800-17 and 800-20 contain the full cipher text results table. Figures 52 and 53 show partial cipher data results of Test 4 for our DES and TDES designs.

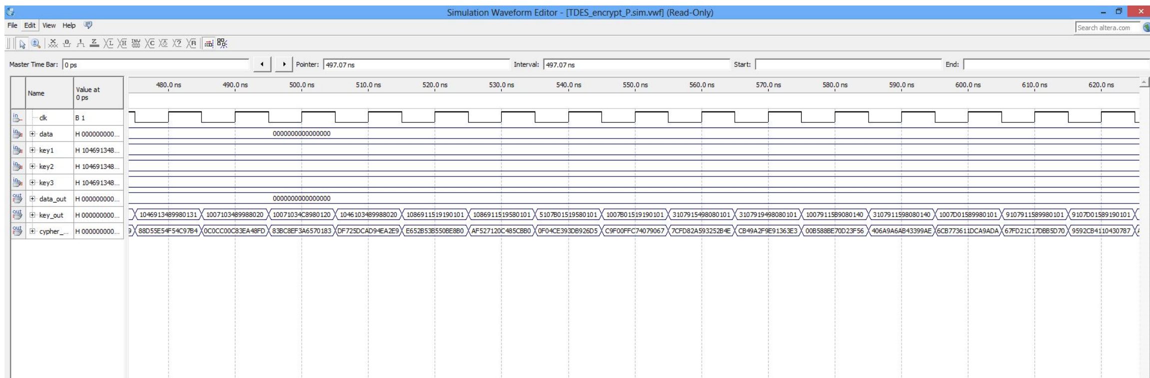


Figure 52: DES Pipelined Encryption MOV Test 4

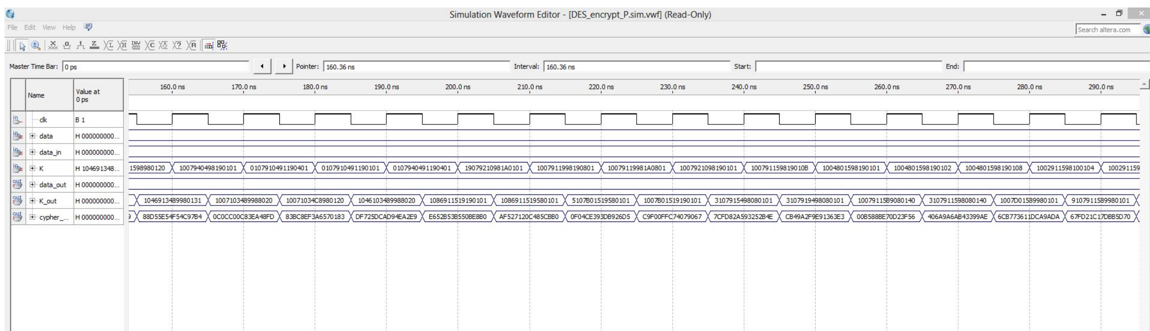


Figure 53: TDES Pipelined Encryption MOV Test 4 Part 2

As seen in Figures 52 and 53, the cipher results match the expected ciphers from Table 8. MOV Test 4 passed.

4.5 Substitution Table KAT for Encryption

The Substitution Table KAT for Encryption consists on validating the encryption with the respective data and key given in Table 9.

Table 9: Substitution Table KAT for Encryption Cipher Results

	Resulting Cipher Text for DES/TDES Variable Plaintext KAT	
Plain Text (Data)	key	Cipher Text
key0 = 0x01A1D6D039776742	key0 = 0x7CA110454A1A6E57	C0 = 0x690F5B0D9A26939B
key1 = 0x5CD54CA83DEF57DA	key1 = 0x0131D9619DC1376E	C1 = 0x7A389D10354BD271

key2 = 0x0248D43806F67172	key2 = 0x07A1133E4A0B2686	C2 = 0x868EBB51CAB4599A
key3 = 0x51454B582DDF440A	key3 = 0x3849674C2602319E	C3 = 0x7178876E01F19B2A
.	.	.
.	.	.
.	.	.
key60 = 0x072D43A077075292	key60 = 0x4FB05E1515AB73A7	C60 = 0x2F22E49BAB7CA1AC
key61 = 0x02FE55778117F12A	key61 = 0x49E95D6D4CA229BF	C61 = 0x5A6B612CC26CCE4A
key62 = 0x1D9D5C5018F728C2	key62 = 0x018310DC409B26D6	C62 = 0x5F4C038ED12B2E41
key63 = 0x305532286D6F295A	key63 = 0x1C587F1C13924FEF	C63 = 0x63FAC0D034D9F793

NIST publications 800-17 and 800-20 contain the full cipher text results table. Figures 54 and 55 show partial cipher data results of Test 5 for our DES and TDES designs.

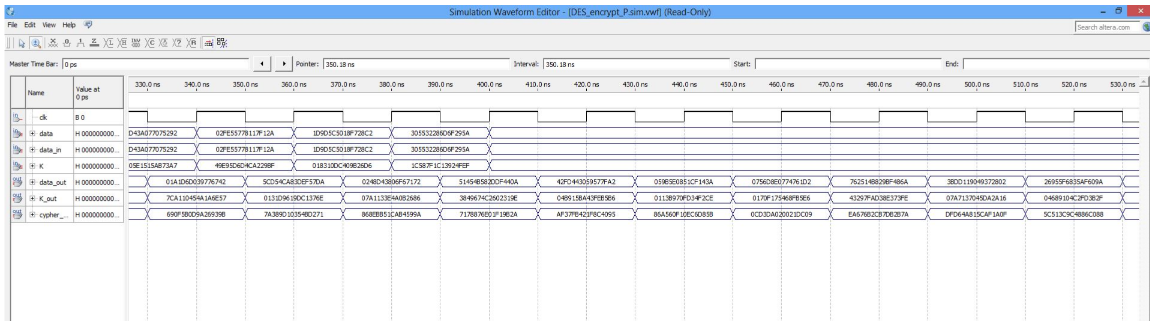


Figure 54: DES Pipelined Encryption MOV Test 5

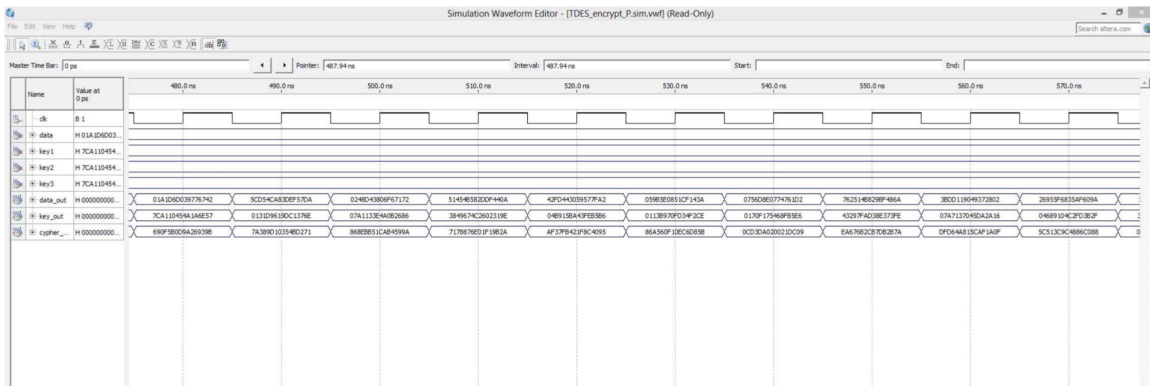


Figure 55: TDES Pipelined Encryption MOV Test 5 Part 2

As seen in Figures 53 and 54, the cipher results match the expected ciphers seen in Table

9. MOV Test 5 passed.

4.6 Modes Test for encrypt (MONTECARLO)

The Modes Test for Encrypt consists on initializing a 64-bit key and a 64-bit data input. In our case, we initialize the key to 0x0000000800000000 and the input data to 0x0000000000000000. The cipher data that yields from processing these two, 0x4F644C92E192DFED, is fed back to the encryption algorithm as the next 64-bit data input. Which means that, now the data input is 0x4F644C92E192DFED. This operation is performed 10000 times. At the 10000th time the key is updated by XOR-ing the current key with the current cipher. This operation is repeated for 400 keys. This yields four million ciphers that must be validated. These are too many ciphers. Instead of updating the key every 10000 ciphers, in our implementation of MONTECARLO, we update the key every 128 ciphers. Bellow, we show our MONTECARLO implementations and simulation results for our DES and TDES pipelined schemes.

Figure 13 shows how we implement the MONTECARLO simulation for the pipelined DES encryption scheme.

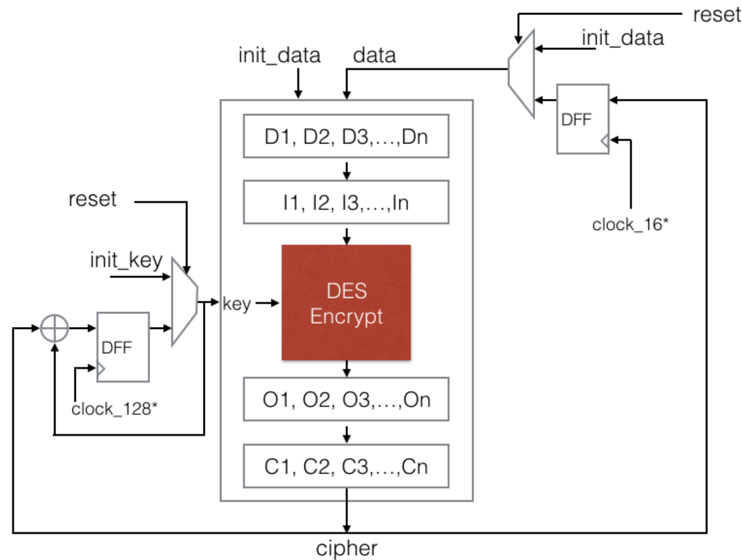


Figure 56: DES Montecarlo Test For Encryption

As seen in Figure 56, we feed the cipher back into the scheme every 16 clock cycles. Our pipelined scheme takes 16 clock cycles to process a 64-data input. Clock_16 clocks a D-Flip Flop every 16 cycles allowing us to feed the cipher back into the encryption scheme. Clock_128 clocks a D-Flip Flop every 128 clock_16 cycles. This means that every 128 clock_16 cycles, the key is updated. The new key equals to the current key XOR-ed with the current cipher. If at any time the reset is set, the key and the data input are reset to the initialized values. Figure 57 shows partial results of our MONTECARLO DES simulation.

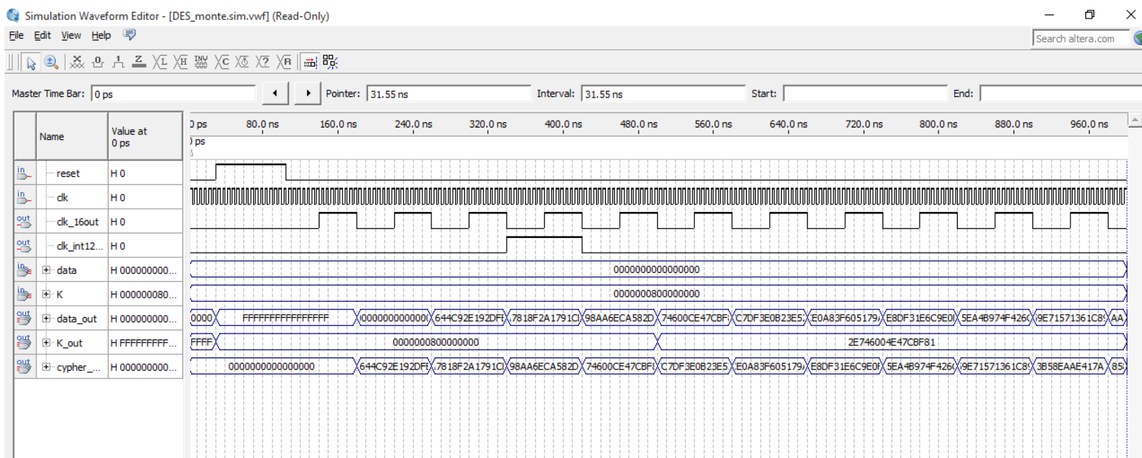
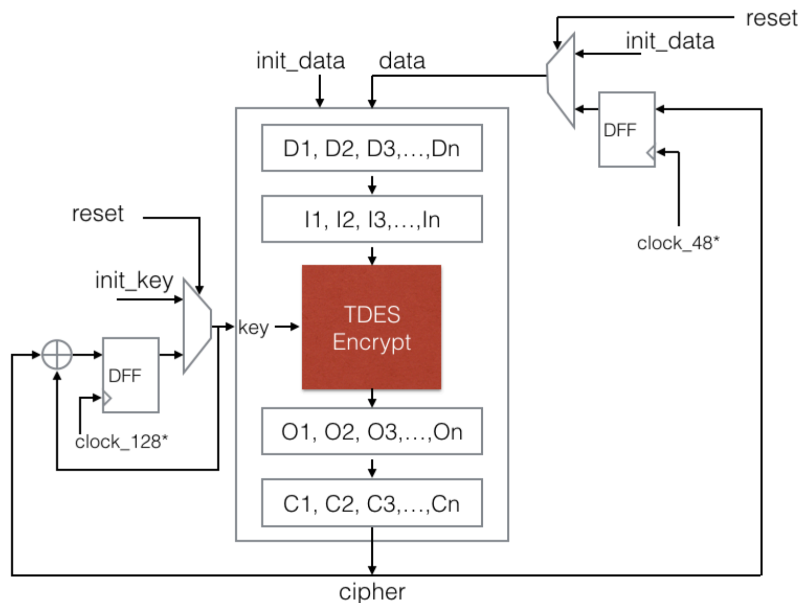


Figure 57: DES Montecarlo Test Results For Encryption

Figure 57 shows that the initial data input is 0x0000000000000000 and the initial key is 0x0000000800000000. The MONTECARLO simulation executes as expected. Every 16 clock cycles the cipher feedbacks to the input of the scheme. Once clk_int128 counts 128 clk_16 cycles, the key also updates. For the simulation shown in Figure 57, we also initialize clk_int128 to 125 clk_16 clock cycles. This allows us to see a key update at 50 ns.

Figure 58 shows how we implement the MONTECARLO simulation for the pipelined TDES encryption scheme.



* clock_48 counts 48 regular clock cycles
** clock_128 counts 128 clock_48 cycles

Figure 58: TDES Montecarlo Test For Encryption

As seen in Figure 58, we feed the cipher back into the scheme every 48 clock cycles. Our pipelined scheme takes 48 clock cycles to process a 64-data input. Clock_48 clocks a D-Flip Flop every 48 cycles allowing us to feed the cipher back into the encryption scheme. Clock_128 clocks a D-Flip Flop every 128 clock_48 cycles. This means that every 128 clock_48 cycles, the key is updated. The new key equals to the current key XOR-ed with the current cipher. If at any time the reset is set, the key and the data input are resent to the initialized values. Figure 59 shows partial results of our MONTECARLO TDES simulation.

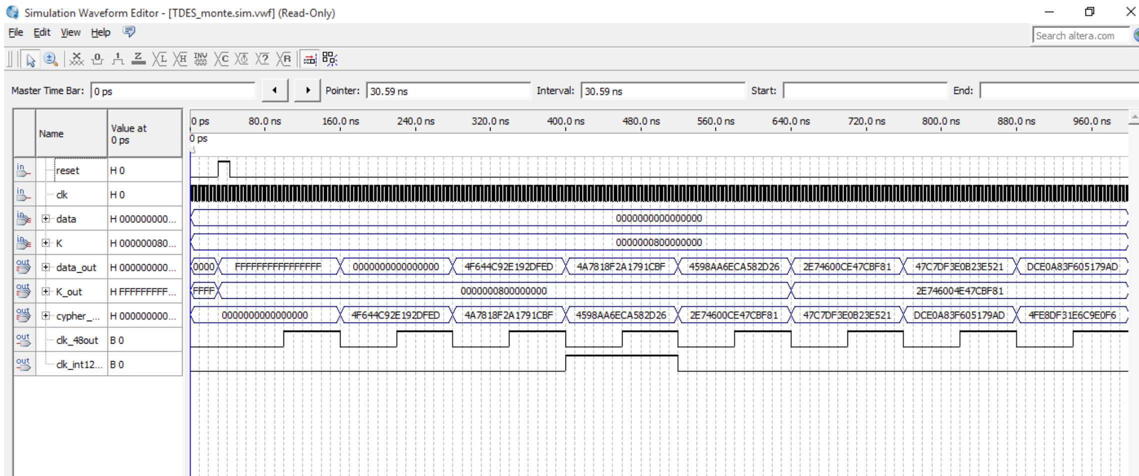


Figure 59: TDES Montecarlo Test Results For Encryption

Figure 59 shows that the initial data input is 0x0000000000000000 and the initial key is 0x0000000800000000. The MONTECARLO simulation executes as expected. Every 48 clock cycles the cipher feedbacks to the input of the scheme. Once clk_int128 counts 128 clk_48 cycles, the key also updates. For the simulation shown in Figure 59, we also initialize clk_int128 to 125 clk_48 clock cycles. This allows us to see a key update at 640 ns.

CHAPTER V

DES AND TDES NON-PIPELINED VS PIPELINED PERFORMANCE EVALUATION

In this chapter, we evaluate the performance and cost of the designs presented in Chapters 2 and 3. We present the performance and cost evaluation of our DES design, TDES design, DES Pipelined design and TDES Pipelined design. We compare our designs' performances and costs to those presented in [50] through [57]. To demonstrate our performance and cost results, we show the simulation results in Quartus II and the Altera University Program (U.P.) Simulator. The compilation results, in Quartus II, provide the cost to implement our design in hardware (Altera DE2 Board EP2C35F672C6). The waveform simulations, in the Altera U.P. Simulator, provide the throughput and delays in the hardware.

5.1 Hardware Setup and EDA Tools

The cost and performance tests took place in the controlled environment of the NRL (Network Research Lab) of Electrical/Computer Engineering at The University Of Texas Rio Grande Valley. Although our project does not pose a threat to networks outside of our controlled environment, the necessary EDA tools need to be available in the computer where the simulations are executed. An iMac was the element in which the performance of DES and TDES was evaluated. The iMac runs on Windows 8 Enterprise operating system.

Installed are the two applications that allow to the code to be generated, checked/verified, compiled, simulated and finally programed into the DE2hardware. These two programs are Quartus II Web Service Pack 1 edition and the Altera University Program Simulator. Once the blaster driver is installed in the computer, the DE2 Board can be linked to the computer via a USB connector. The setup is seen in Figure 60 below.

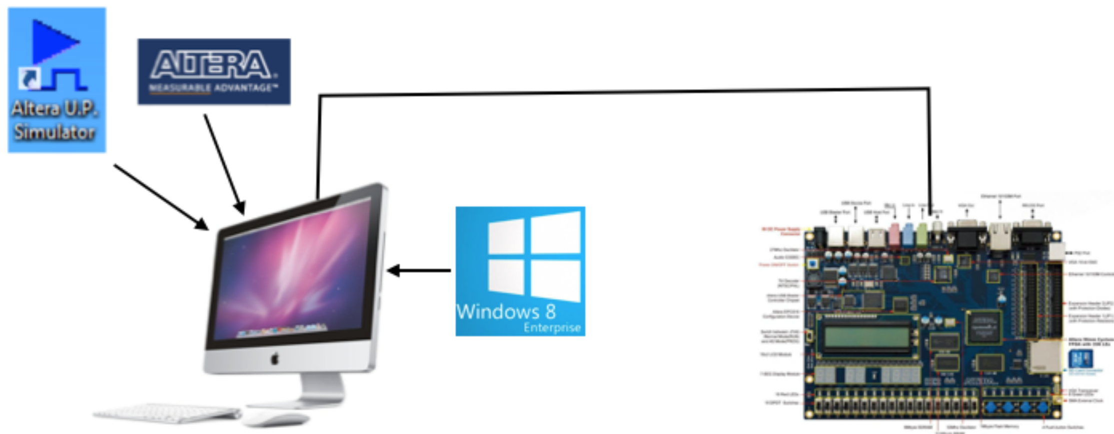


Figure 60: Design's Cost and Performance Evaluation Set Up.

The Quartus II program allows us to generate the VHDL code, test the code for errors, receive warning messages, compile the code and finally program the DE2 Board with the compiled code. The compilation results outputs information regarding the hardware's internal components such the total number of Logic Elements (LEs) and how many of them are utilized in our design.

The Altera U.P. Simulation shows the timing analysis for the input and output signals. By selecting functional simulation over timing simulation, simulations over many more clock cycles can be tested to ensure that the system works as it is intended. This setup is beneficial when simulating a Montecarlo simulation. By selecting timing simulation over functional

simulation, timing violations such as setup time violations and hold time violations can be troubleshoot and propagation delays can be analyzed. Making observations on the timing graphs allow us to ensure that our scheme will operate properly once it is implemented in hardware. Based on these observations we can calculate the throughput of our system.

5.2 Parameters of Performance Evaluation

To evaluate the performance of our DES and TDES designs, we discuss the observations in the EDA tools in which we built, simulated and tested our designs: the Quartus II software, the Altera University Program Simulator and the Altera DE2 board. The parameters that we analyze from the Quartus II software are the Total Logic Elements, the Total Combinational Functions, the Dedicated Logic Registers, the Total Registers, the Total Pins, the Total Memory bits and the time it takes Quartus II to run the Analysis and Synthesis. From the Altera U.P. Simulator, the parameters that we analyze are the time delays and signal propagation delays. Based on the Altera U.P. Simulator and the DE2 Board specs we also analyze the throughput of our designs.

5.3 Results and Discussion

The evaluation results from Quartus II, the Altera U.P. Simulator and the DE2 platform show the cost and performance of our designs. We begin by discussing the cost and performance of each design individually. Finally we compare our cost and performance against other designs mentioned in [50] through [57].

5.3.1 DES Encryption and Decryption Non Pipelined

Our DES Encryption and Decryption schemes are implemented in 22 VHDL files in Quartus II. The Implementation is previously discussed in Chapter 2. The Analysis and Synthesis on the Encryption and Decryption schemes, in Quartus II, generates the following report windows seen in Figures 61 & 62.

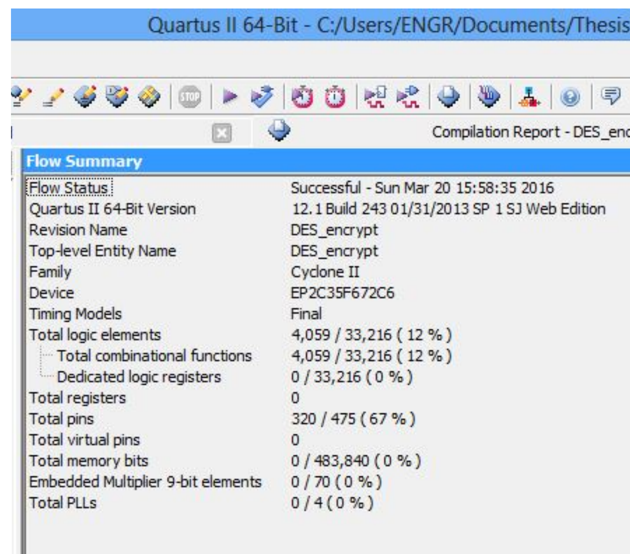


Figure 61: DES Encryption Analysis & Synthesis Cost Results.

Quartus II 64-Bit - C:/Users/ENGR/Documents/Thesis

Compilation Report - DES_dec

Flow Summary

Flow Status	Successful - Sun Mar 20 16:54:02 2016
Quartus II 64-Bit Version	12.1 Build 243 01/31/2013 SP 1 SJ Web Edition
Revision Name	DES_decrypt
Top-level Entity Name	DES_decrypt
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Final
Total logic elements	4,059 / 33,216 (12 %)
Total combinational functions	4,059 / 33,216 (12 %)
Dedicated logic registers	0 / 33,216 (0 %)
Total registers	0
Total pins	320 / 475 (67 %)
Total virtual pins	0
Total memory bits	0 / 483,840 (0 %)
Embedded Multiplier 9-bit elements	0 / 70 (0 %)
Total PLLs	0 / 4 (0 %)

Figure 62: DES Decryption Analysis & Synthesis Cost Results.

As seen in Figures 61 and 62, the DES encryption and decryption schemes utilize 12% of the Total Logic Elements: this is 4,059 of 33,216. 0% of the Dedicated Logic Registers were utilized: that is 0 of 33,216. The Total registers utilized were also 0. The total number of pins utilized was 320 of 475, which is 67%. And the Total Memory Bits utilized were 0 of 483,840, which is 0%. According to the Analysis and Synthesis, other parameters mentioned in the compilation report are Virtual pins, Embedded Multiplier 9-bit elements and PLLs (Phase Lock Loops). We didn't make used of these elements throughout our designs and we will not take these parameters into consideration in further discussions.

An observation made when running the Analysis and Synthesis is the amount it takes Quartus II to complete the analysis and synthesis process. Running the Analysis and Synthesis on the DES Non Pipelined encryption and decryption schemes takes about 3 minutes as seen in Figures 63 and 64. In Section 5.3.6 we compare the Analysis and Synthesis running time of all our designs.

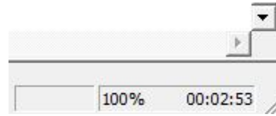


Figure 63: DES Encryption Analysis & Synthesis Running Time.

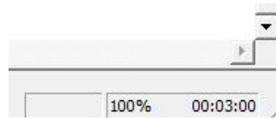


Figure 64: DES Decryption Analysis & Synthesis Running Time.

The timing diagrams from the Altera U.P. Simulator, seen in Figures 65 and 66, are obtained by running the Substitution Table Known Answer Test (Test 5) as stated in Chapter 4. From these timing diagrams we can make several observations.

Figure 65: DES Encryption Substitution Table Known Answer Test In Timing Mode.

Figure 66: DES Decryption Substitution Table Known Answer Test In Timing Mode.

Figures 65 and 66 depict three outputs, the data with its corresponding key and cipher data. The output of our encryption and decryption designs depend on the input data and cipher data respectively, and the input key. In the timing simulation results shown in the above Figures 65 and 66, we set the both input elements to change at the same time so that we can obtain the longest delay time.

As seen in Figures 67 and 68, we update the inputs at 100 ns. The output begins to update at 110 ns, 10ns after the inputs are updated and it displays unstable signals for about 80 ns. The output signal reaches stability after 90 ns. This information lets us know that our DES Non Pipelined system design has a propagation delay of 90 ns. This is the time it takes the input signal to propagate through the design from the time it is fed into the system until it reaches the output. A time period of 100 ns ensures that the output will be stable by time the next 64 input bits updates our system. Therefore, if T is the period at which we update the inputs, and T is 100 ns then the estimated throughput of our design is 640 Megabits per second as shown in Equation 1.

Figure 67: DES Encryption Propagation Results.

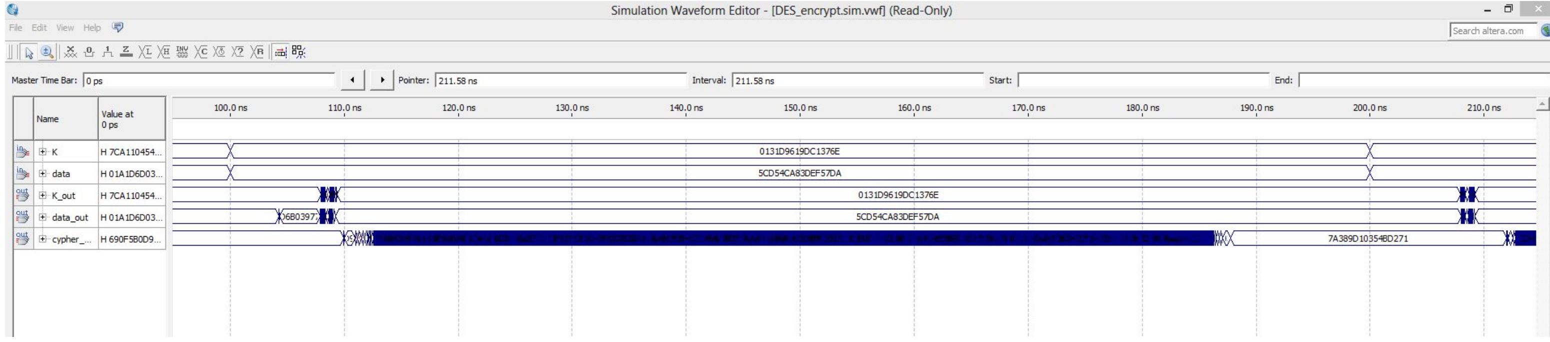
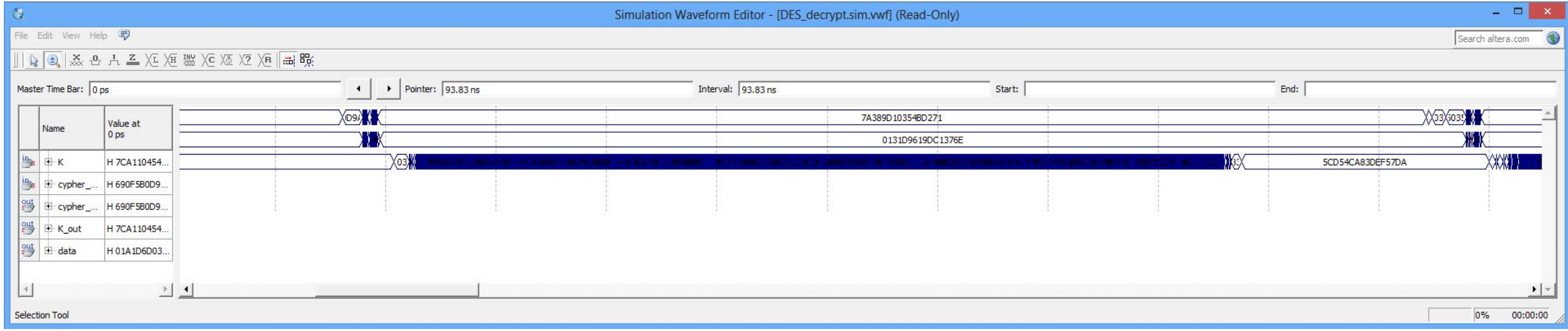


Figure 68: DES Decryption Propagation Results.



$$throughput = \frac{64\ bits}{T} = \frac{64\ bits}{100\ ns} = 640\ Megabits\ per\ second \qquad \text{Equation 1}$$

Another observation from the Altera U.P. Simulator is the time the simulator takes to run the node finder, and the time it takes to execute the simulation. These times increase or decrease in our different designs. We also note that the time to execute functional simulation is greater than executing a time simulation. In Section 5.3.6 we compare the execution times of our various designs.

For this non-pipelined design, the node finder time for both the encryption and decryption is about 4 minutes. These results can be seen in Figures 69 and 70 respectively.

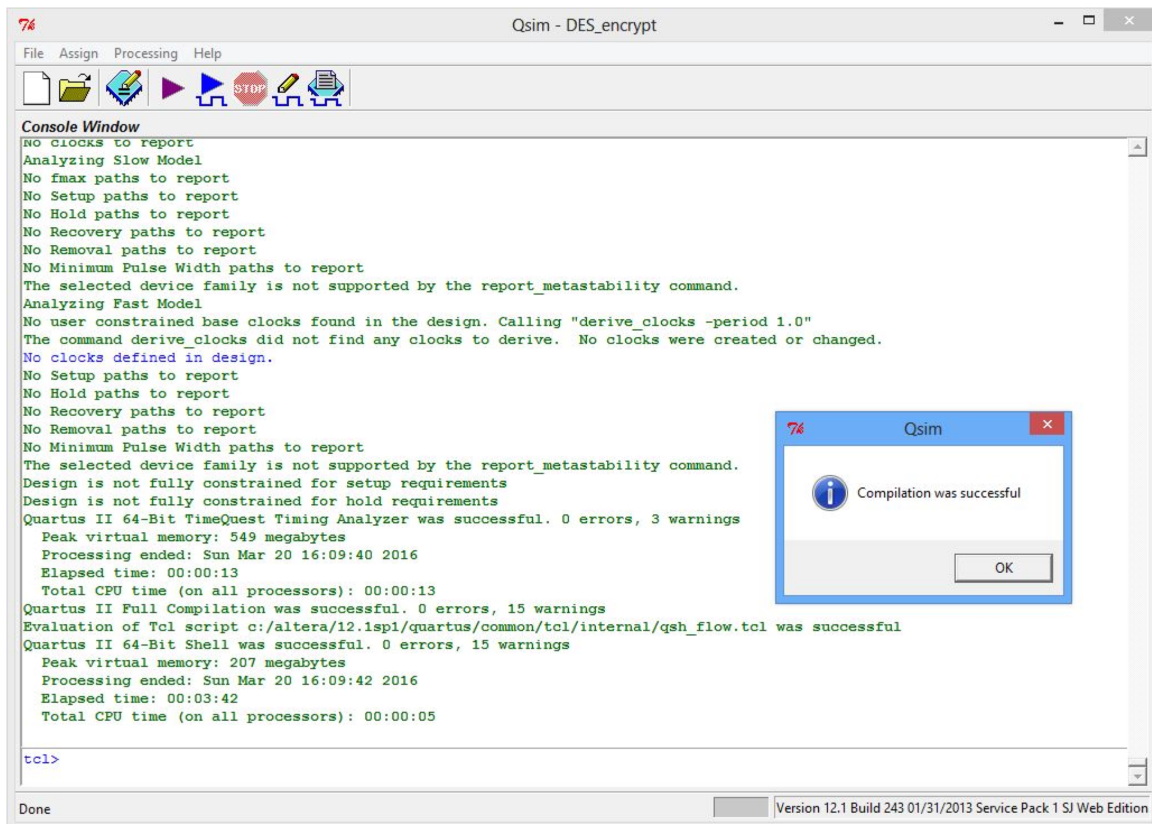


Figure 69: DES Encryption Node Finder Time.

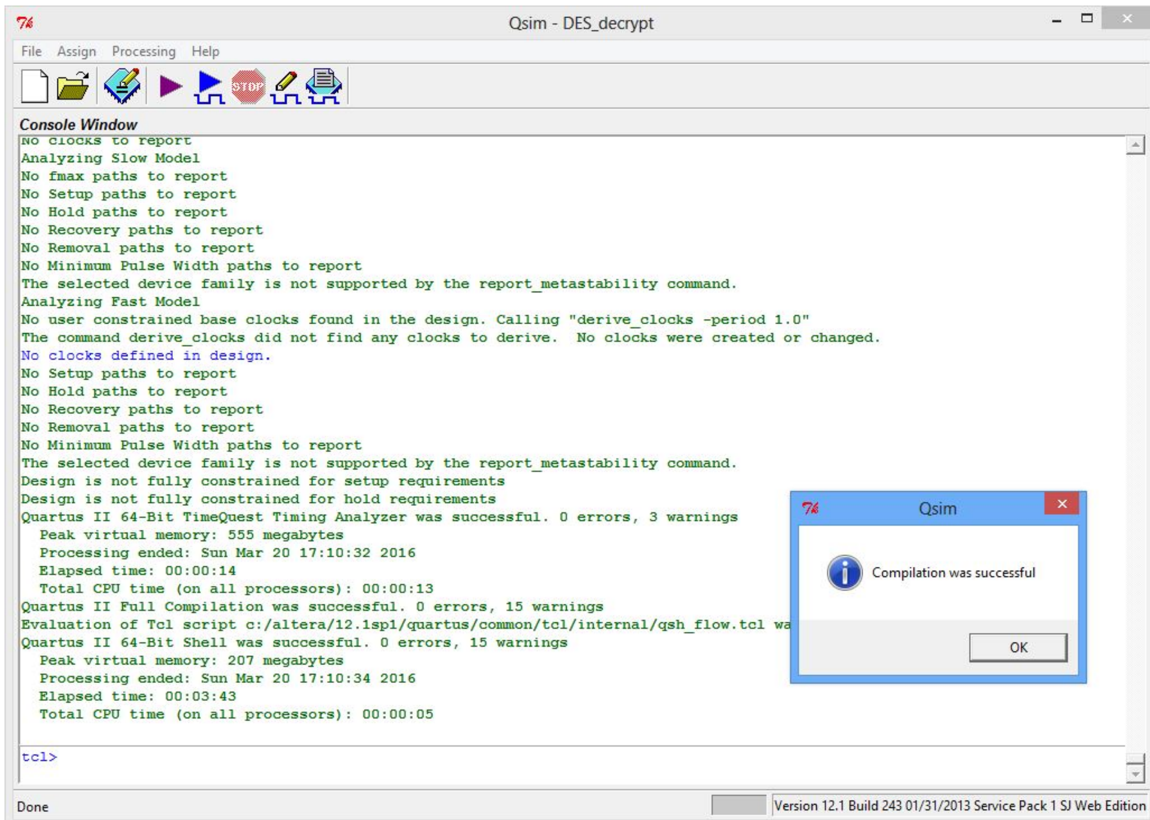


Figure 70: DES Decryption Node Finder Time.

To execute the simulation in functional mode, the simulation takes about 30 seconds to generate the netlist as seen in Figures 71 and 72.

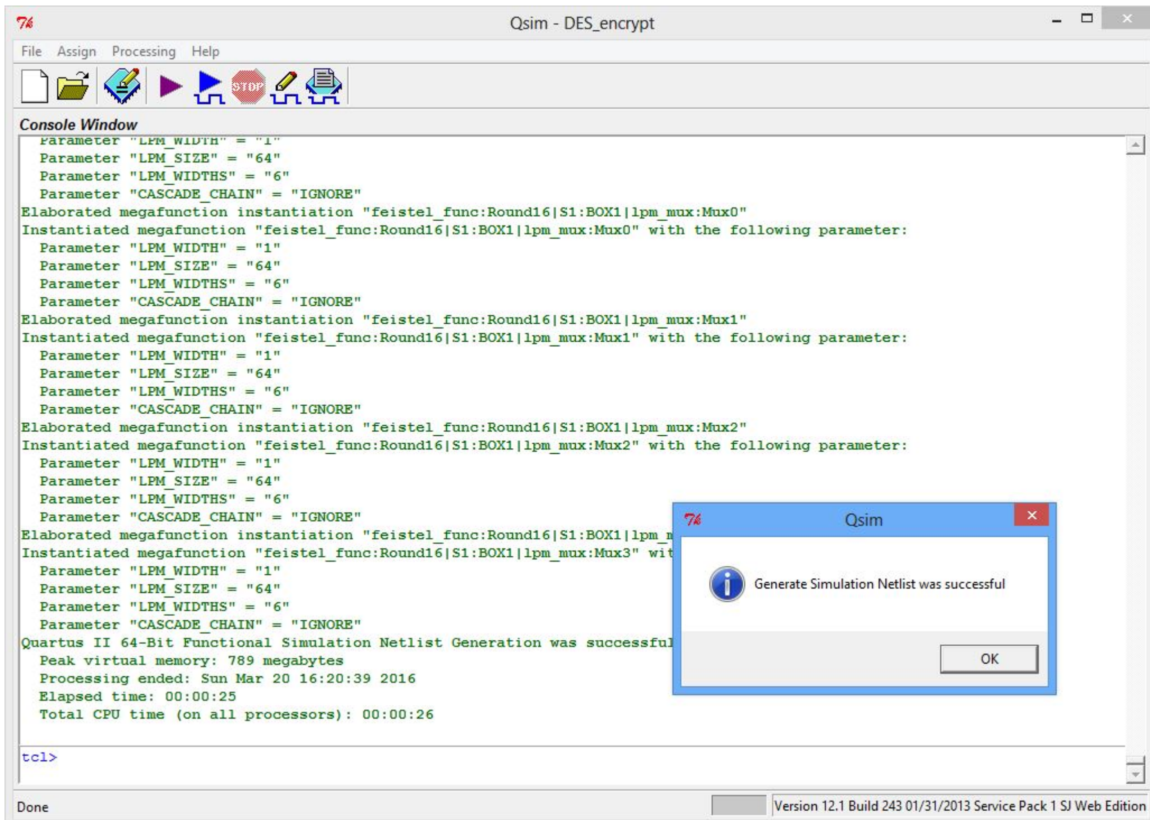


Figure 71: DES Encryption Generate Simulation Netlist Time.

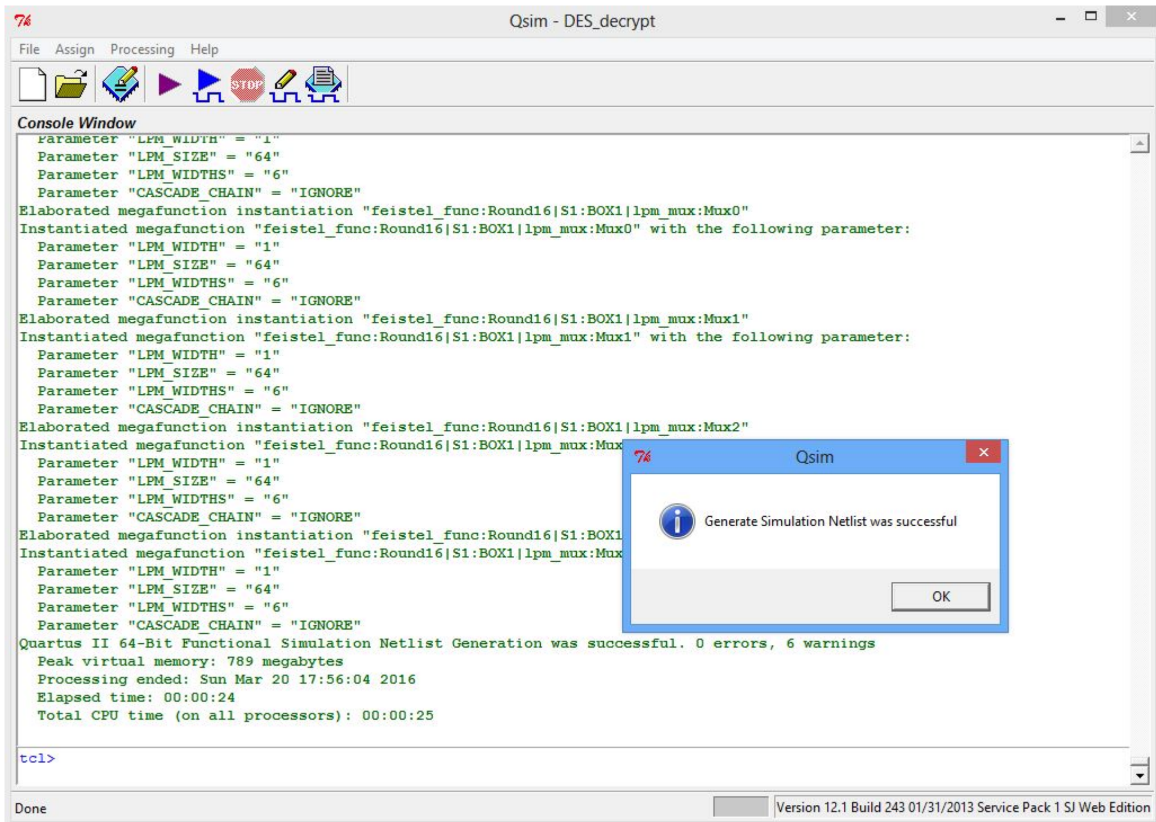


Figure 72: DES Decryption Generate Simulation Netlist Time.

After generating the netlist, the simulation spends about 4 minutes executing the simulation as seen in Figures 73 and 74.

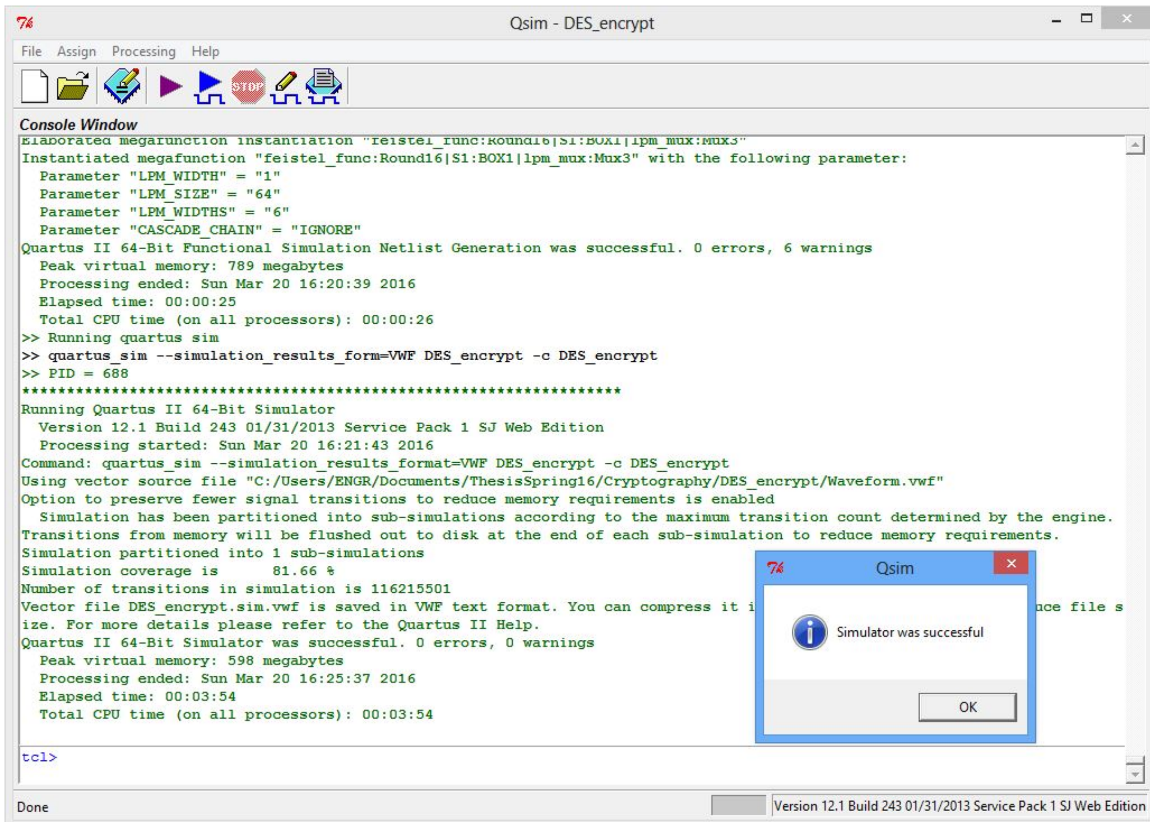


Figure 73: DES Encryption Functional Simulation Time.

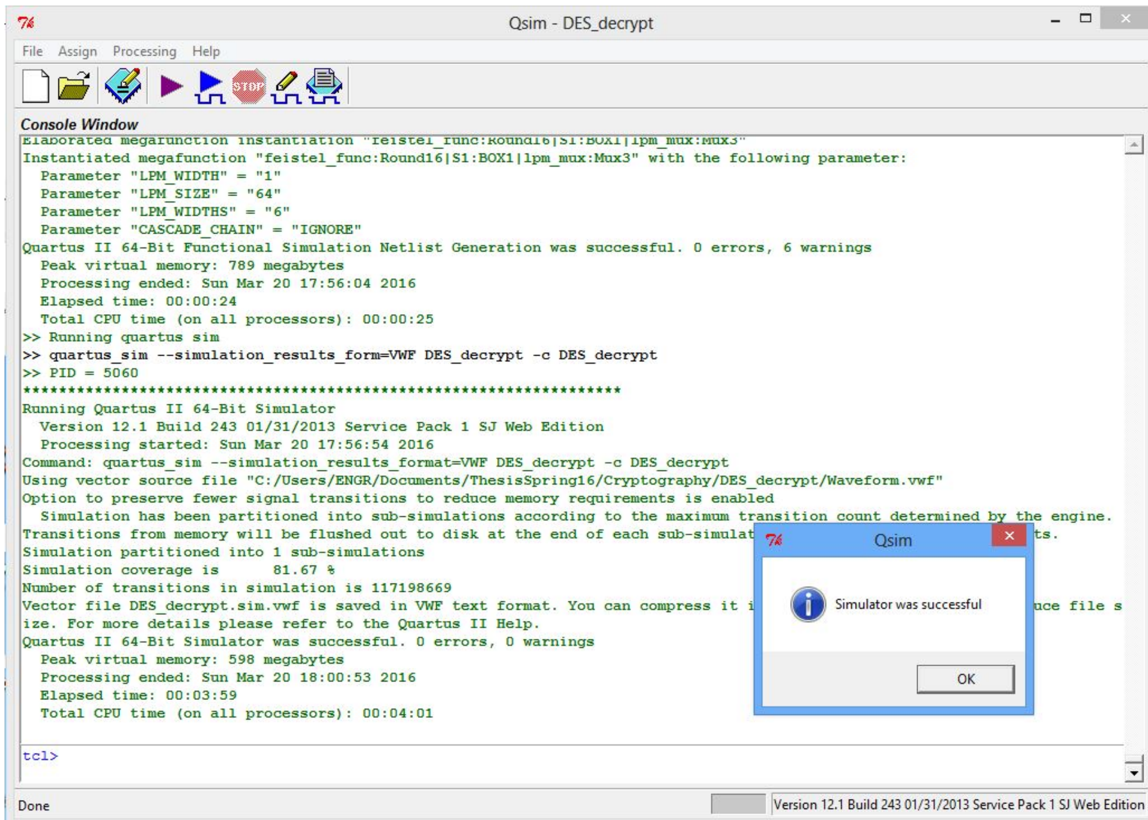


Figure 74: DES Decryption Functional Simulation Time.

Executing the simulation in timing mode only takes about 15 seconds as seen in Figures 75 and 76.

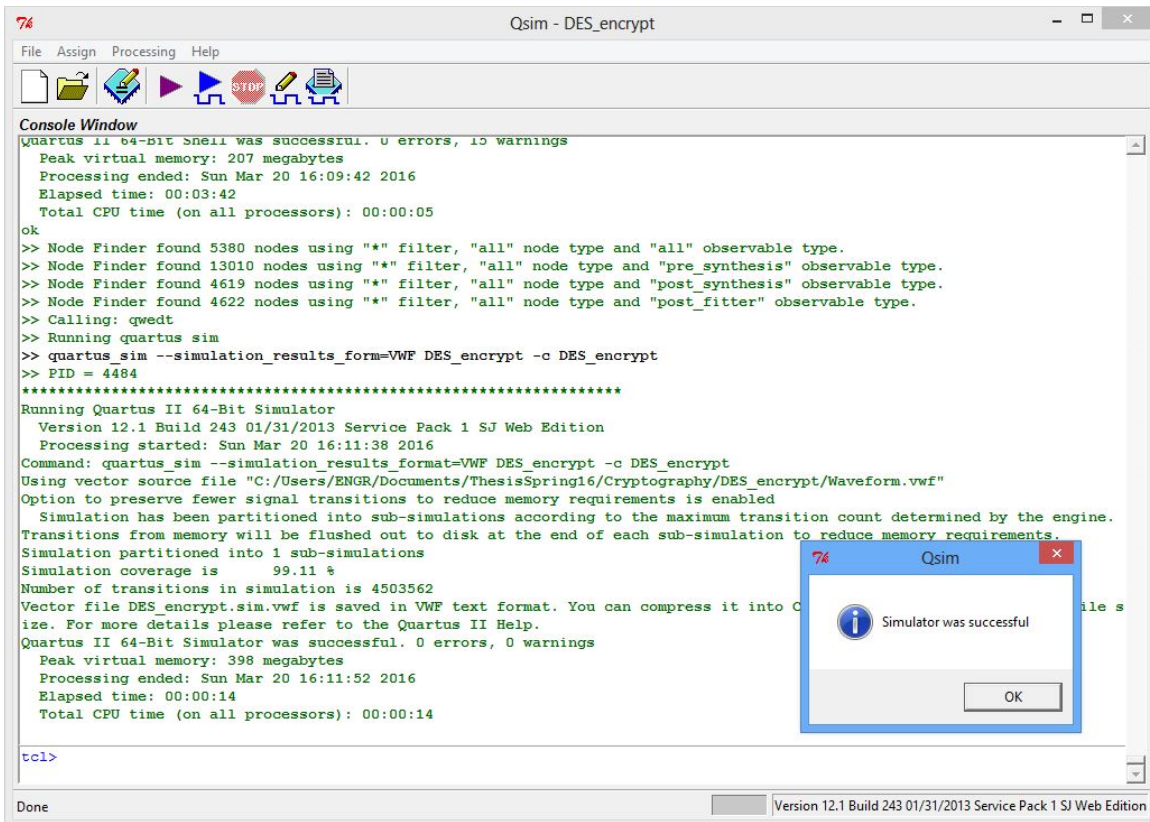


Figure 75: DES Encryption Timing Simulation Time.

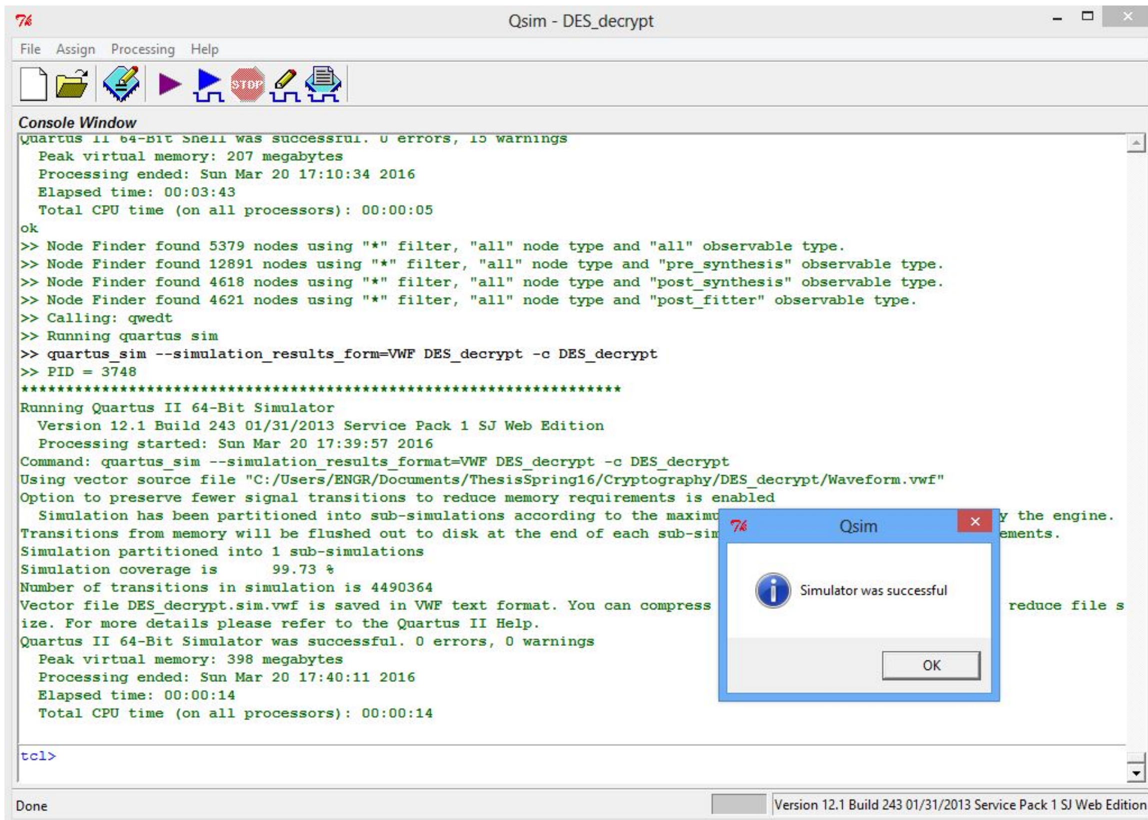


Figure 76: DES Decryption Timing Simulation Time.

Because the implementations for the encryption and decryption are very similar, we obtain results that are almost equal for both. We see the same relationship between the encryption and decryption for all our designs: that is DES pipelined encryption and decryption, TDES encryption and decryption and TDES pipelined encryption and decryption. In the following sections we only show the results for encryption and we keep in mind that the same results apply for the decryption scheme.

5.3.2 TDES Encryption and Decryption Non Pipelined

Our TDES Encryption and Decryption schemes are implemented in 25 VHDL files in Quartus II. The Implementation is previously discussed in Chapter 3. The Analysis and

Synthesis on the Encryption and Decryption schemes, in Quartus II, generates the following report window seen in Figure 77.

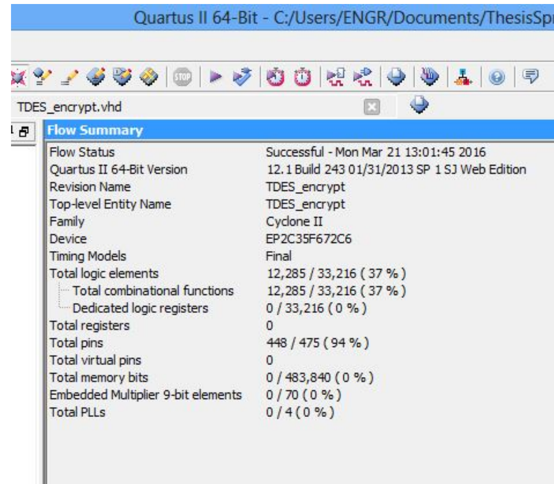


Figure 77: TDES Analysis & Synthesis Cost Results.

As seen in Figure 77, the TDES encryption and decryption schemes utilize 37% of the Total Logic Elements, which are 12,285 of 33,216. 37% of the Total Combinational Functions are utilized: that is 12,285 of 33,216. 0% of the Dedicated Logic Registers were utilized: that is 0 of 33,216. The Total registers utilized are also 0. The total number of pins utilized is 448 of 475, which is 94%. And the Total Memory Bits utilized are 0 of 483,840, which is 0%.

Running the Analysis and Synthesis on the TDES Non Pipelined encryption and decryption schemes takes about 13 minutes as seen in Figure 78. In Section 5.3.6 we compare the Analysis and Synthesis running time to the other designs we implemented.

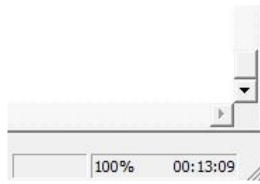


Figure 78: TDES Analysis & Synthesis Running Time.

The timing diagram from the Altera U.P. Simulator, seen in Figure 79, is obtained by running part of the Substitution Table Known Answer Test (Test 5) as stated in Chapter 4. From these timing diagrams we can make several observations.

Figure 79: TDES Substitution Table Known Answer Test In Timing Mode.

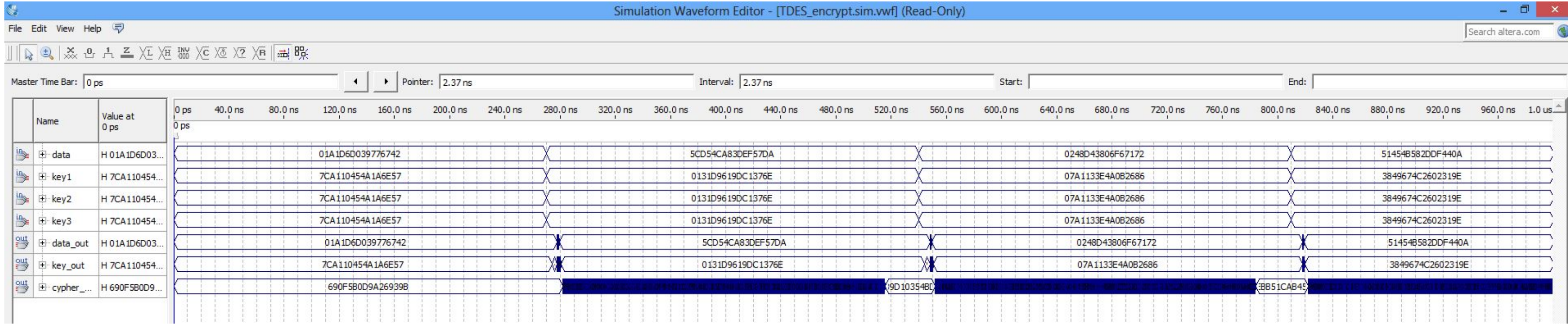


Figure 79 depicts three outputs, the data with its corresponding key and cipher data. The outputs of our encryption and decryption designs depend on the input data and cipher data respectively and the three input keys. In the timing simulation results shown in the above Figure 79 we set the all input elements to change at the same time so that we can obtain the longest delay time.

As seen in Figure 80 we update the inputs at 270 ns. The cipher data output begins to update at 280 ns, 10ns after the inputs are updated and it displays unstable signals for about 235 ns. The output signal reaches stability after 245 ns at the 515 ns mark. This information lets us know that our DES Non Pipelined system design has a propagation delay of 245 ns. Note that this delay is about three times more than our DES non-pipelined design. Since TDES is three times the hardware of DES we expect this increase in delay. A time period of 270 ns ensures that the output will be stable by time the next 64 input bits updates our system. Therefore, if T is the period at which we update the inputs, and T is 270 ns then the estimated throughput of our design is 237 Megabits per second as shown in Equation 2.

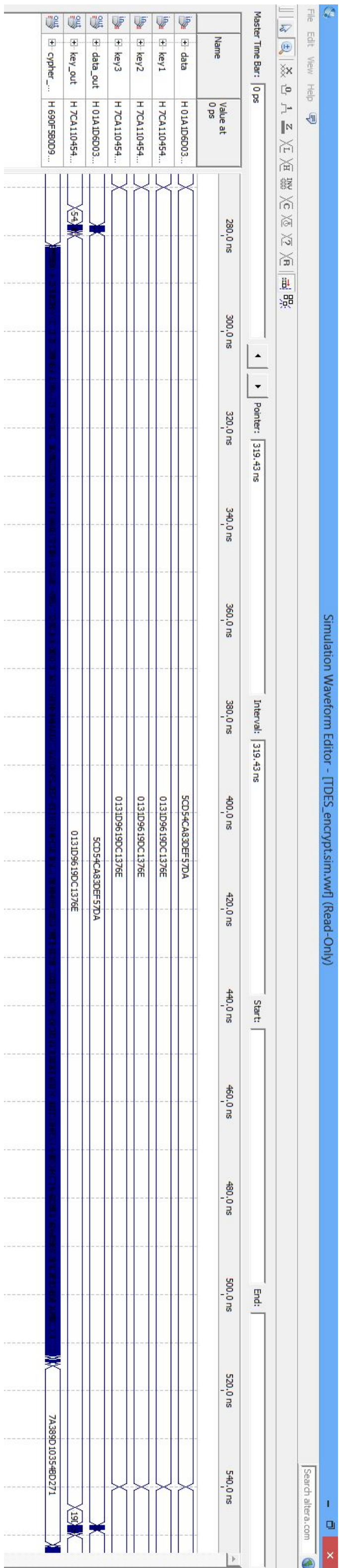


Figure 80: TDES Propagation Results.

$$throughput = \frac{64\ bits}{T} = \frac{64\ bits}{270\ ns} = 237\ Megabits\ per\ second \qquad \text{Equation 2}$$

In this non-pipelined design, the node finder time for both the encryption and decryption is about 15 minutes. This result can be seen in Figure 81.

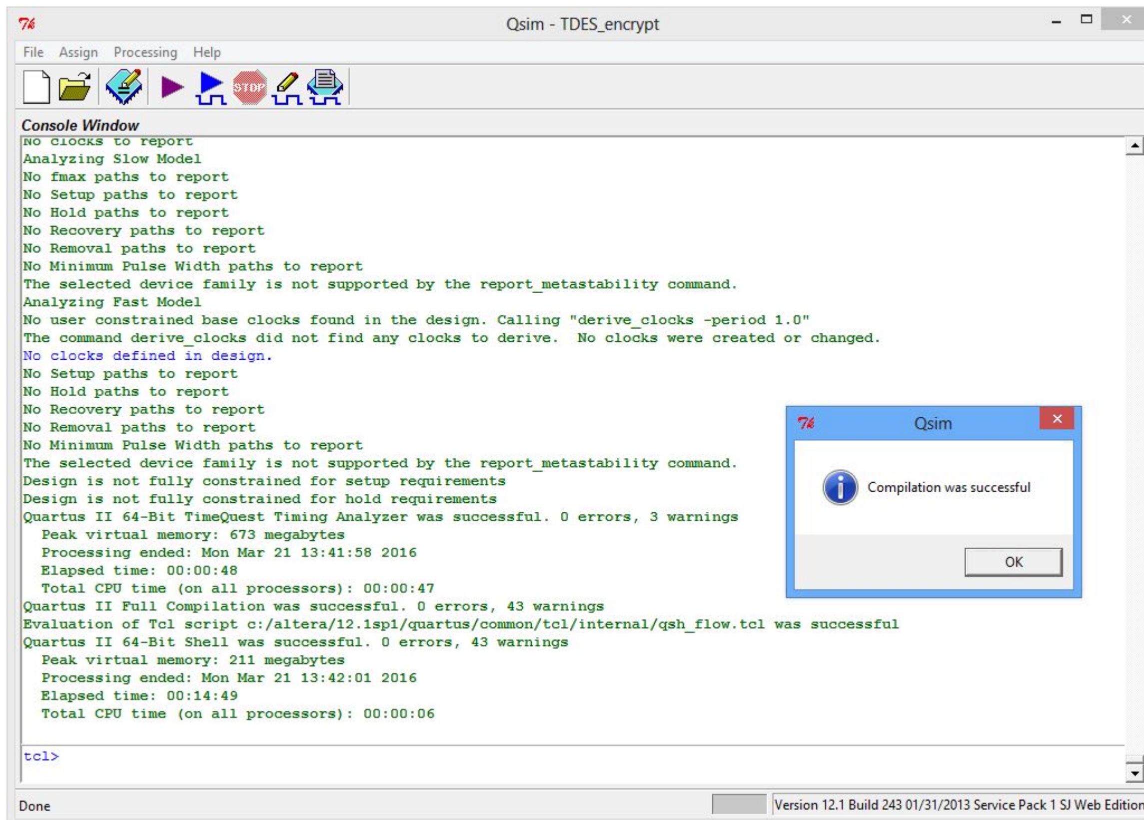


Figure 81: TDES Node Finder Time.

To execute the simulation in functional mode, the simulation takes about 1 minute and 15 seconds to generate the netlist as seen in Figure 82.

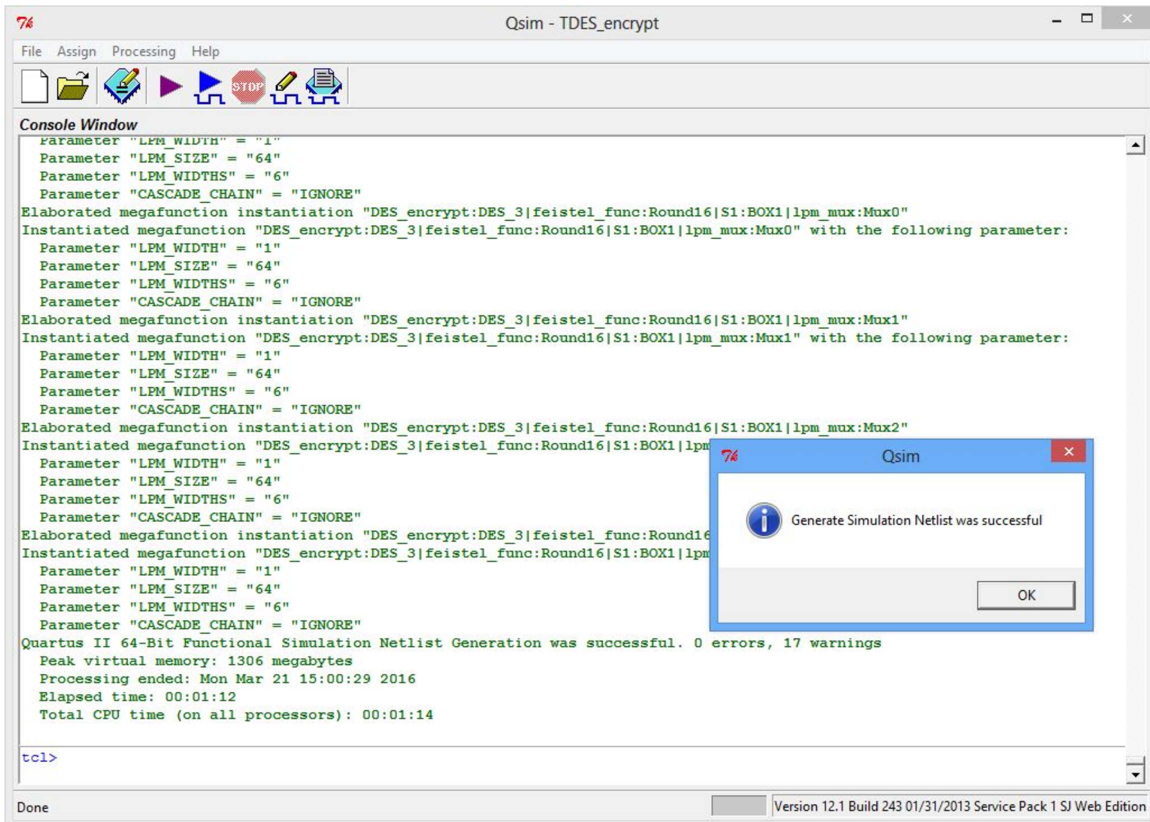


Figure 82: TDES Generate Simulation Netlist Time.

After generating the netlist, the simulation spends about 13 minutes executing the simulation as seen in Figure 83.

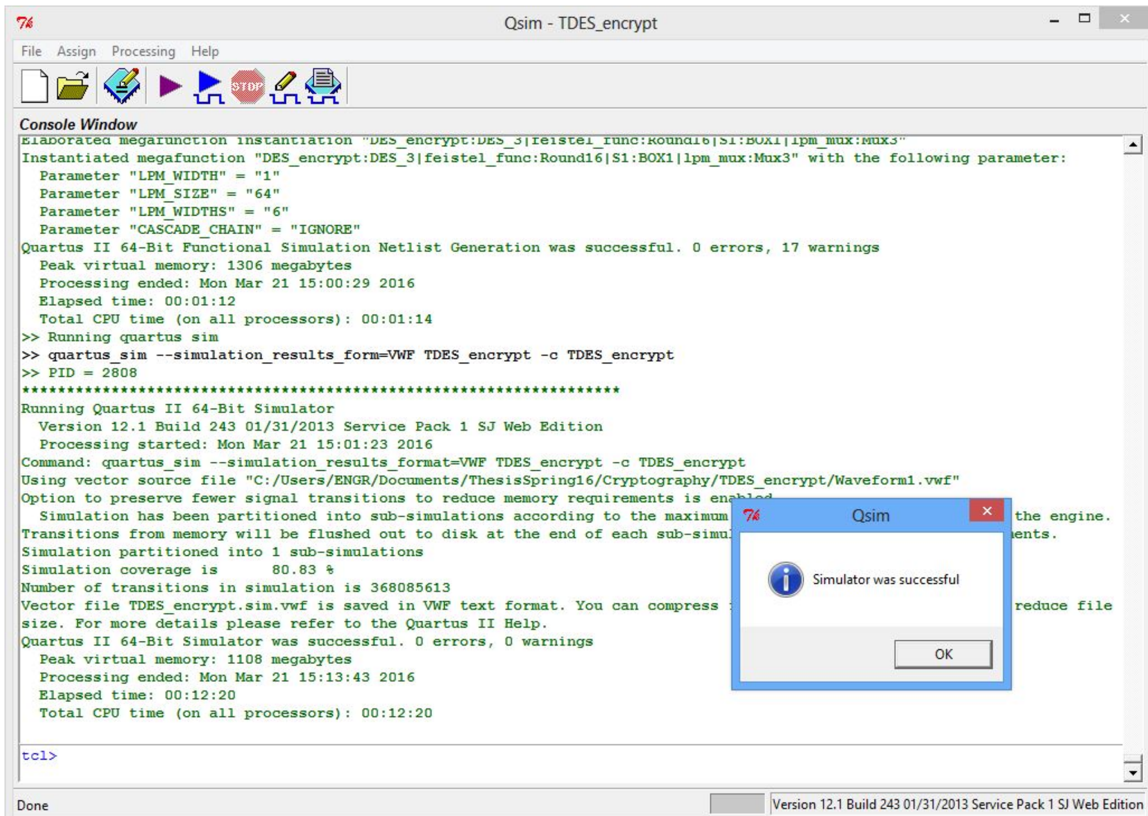


Figure 83: TDES Functional Simulation Time.

Executing the simulation in timing mode only takes about 1 minute as seen in Figure 84.

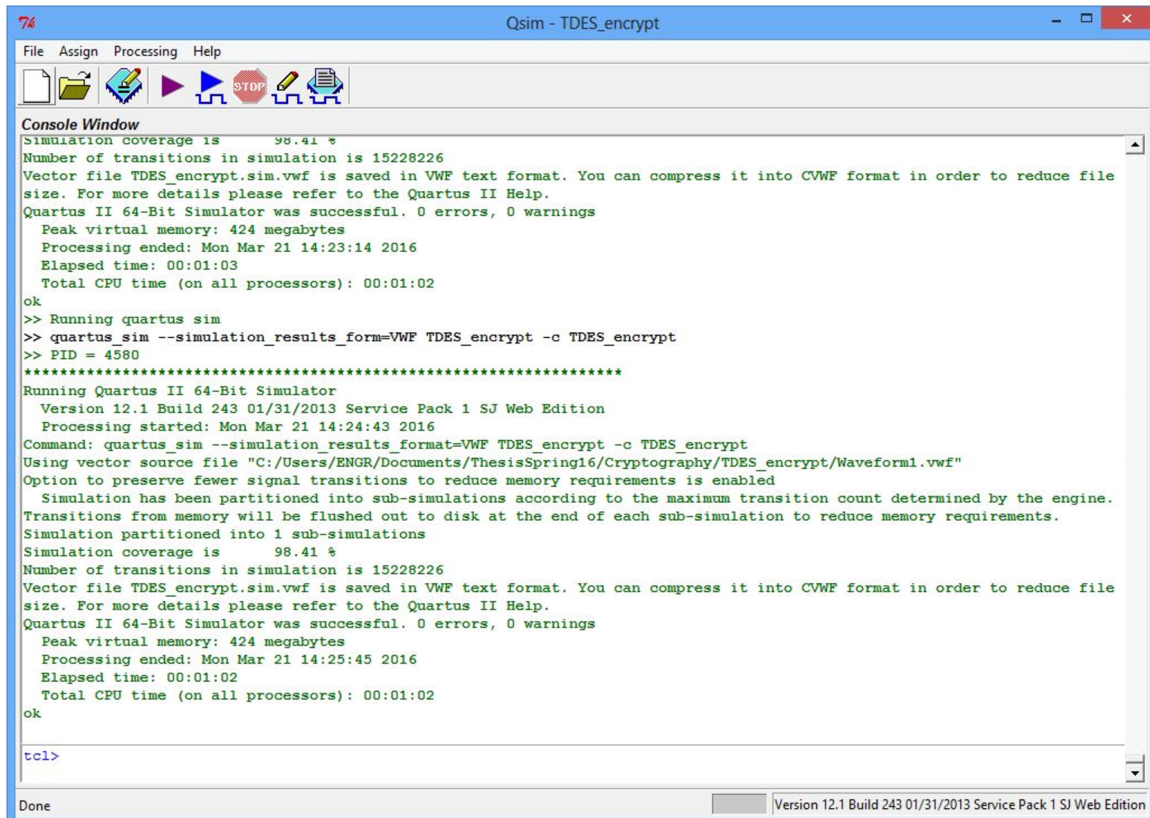


Figure 84: TDES Timing Simulation Time.

5.3.3 DES Pipelined Encryption and Decryption

Our DES Pipelined Encryption and Decryption schemes are implemented in 22 VHDL files in Quartus II. The Implementation is previously discussed in Chapter 4. The Analysis and Synthesis on the Encryption and Decryption schemes, in Quartus II, generates the following report window seen in Figure 85.

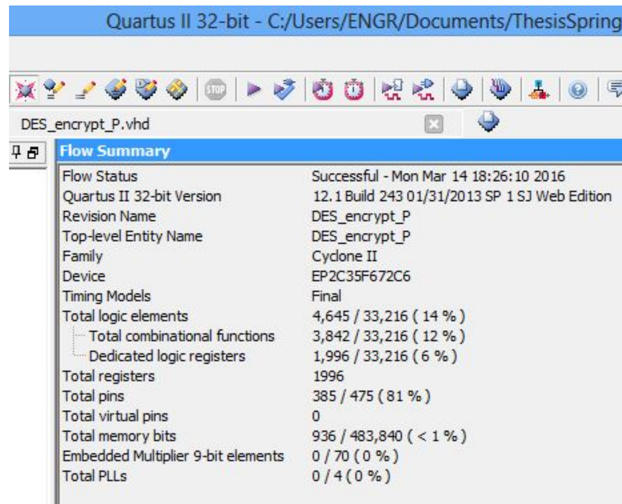


Figure 85: DES Pipelined Analysis & Synthesis Cost Results.

As seen in Figures 85, the DES pipelined encryption and decryption schemes utilize 14% of the Total Logic Elements, which is 4,645 and 4,570 of 33,216 respectively. 12% of the Total Combinational Functions are utilized: that is 3,842 of 33,216. 6% of the Dedicated Logic Registers were utilized: that is 1,996 and 1,988 of 33,216 respectively. The Total registers utilized are also 1,996 and 1,998 respectively. The total number of pins utilized is 385 of 475, which is 81%. And the Total Memory Bits utilized are 936 and 1,664 of 483,840 respectively, which is less than 1%.

Running the Analysis and Synthesis, on the DES pipelined encryption and decryption schemes, takes about 2 minutes as seen in Figure 86. In Section 5.3.6 we compare the Analysis and Synthesis running time to the other designs we implemented.

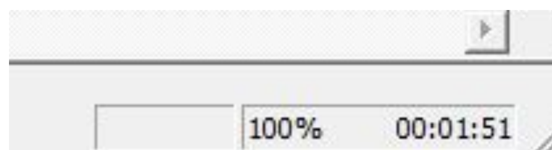


Figure 86: DES Pipelined Analysis & Synthesis Running Time.

The timing diagram from the Altera U.P. Simulator, seen in Figure 87, is obtained by running the Substitution Table Known Answer Test (Test 5) as stated in Chapter 4. From these timing diagrams we can make several observations.

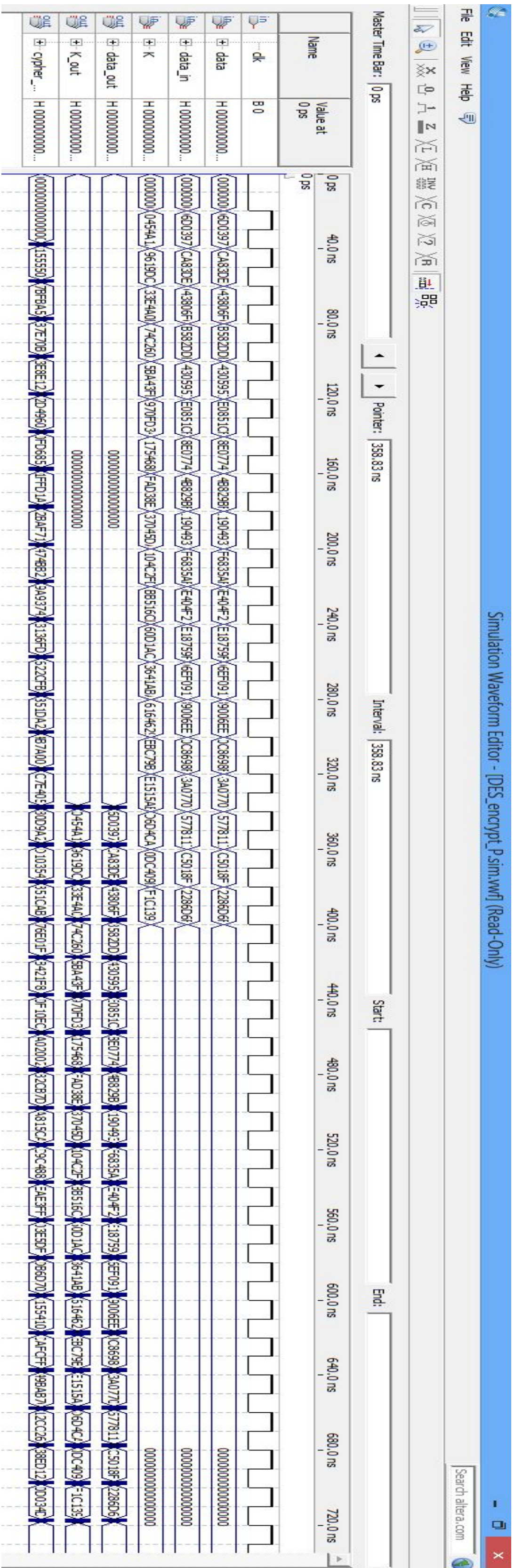


Figure 87: DES Pipelined Substitution Table Known Answer Test In Timing Mode.

Figure 87 depicts three outputs, the data with its corresponding key and cipher data. The output of our encryption and decryption designs depend on the input data and cipher data respectively and the input key. Because these designs are pipelined as presented in Chapter 3, we update the inputs at the same time that our clock strikes the positive edge. Since both, our encryption and decryption designs are negative edge triggered, to avoid any set up time or hold time violations we modify the inputs on the positive edge of the clock. After the negative edge strikes, the input data will go through our pipelined design. Since there are 16 rounds in the encryption/decryption scheme, and we have pipelined our design at every round, it will take 16 clock cycles for the input data to reach the output of our design. After the initial 16-cycle delay, our systems outputs at every clock cycle. Therefore, if T is the period for one clock cycle, and T is 20 ns (since the DE2 board's internal clock is 20 ns), then the estimated throughput of our design is 3.2 Gigabits per second as shown in Equation 3.

$$throughput \approx \frac{64\ bits}{T} \approx \frac{64\ bits}{20\ ns} \approx 3.2\ Gigabits\ per\ second \qquad \text{Equation 3}$$

As seen in Figure 88, the negative edge of the clock triggers at 330 ns. The output begins to update 6 ns after the clock triggers the negative edge, and displays unstable signals for about 2 ns seconds. The output signal reaches stability after 8 ns seconds at the 338 ns mark. This information lets us know that our DES pipelined system design has a propagation delay of 8 ns seconds. Note that this delay is much less than our non-pipelined designs because the amount of hardware that processes the input signal in every pipelined stage is much less. In the case of the non-pipelined designs, the input signal must traverse the entire designs. Every clock cycle, the pipeline design only processes the signal to an equivalent of one round of hardware in DES. This allows us to achieve a shorter delay and a higher throughput.

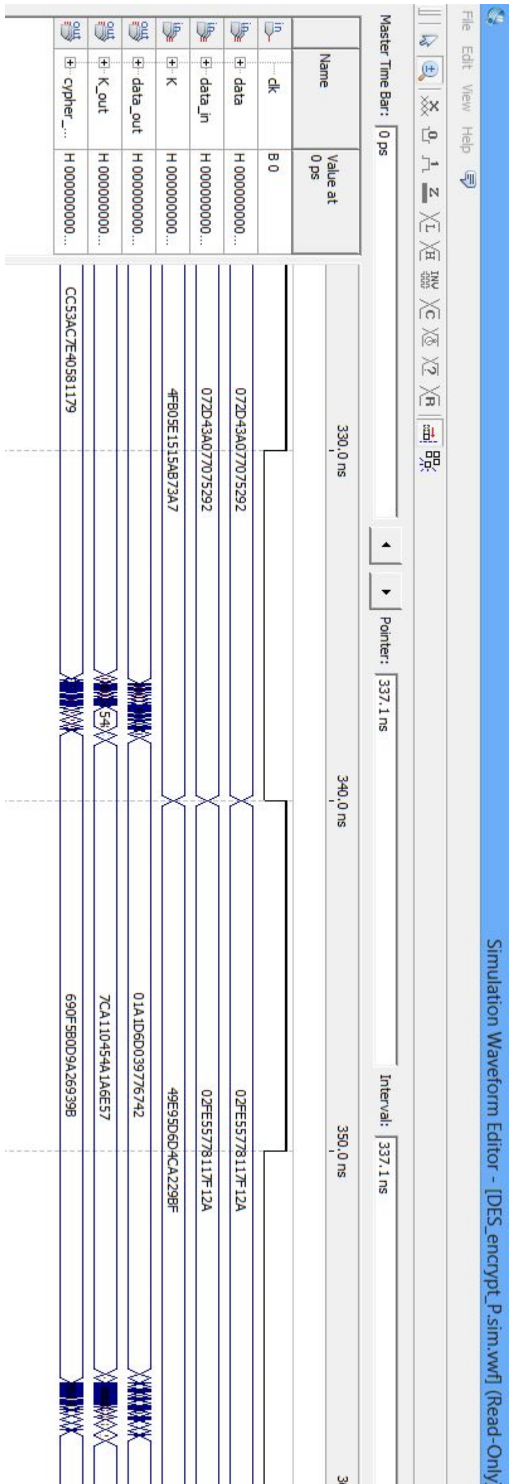


Figure 88: DES Pipelined Delay Results.

In this Pipelined design, the node finder time for both the encryption and decryption is about 3 minutes. This result can be seen in Figure 89.

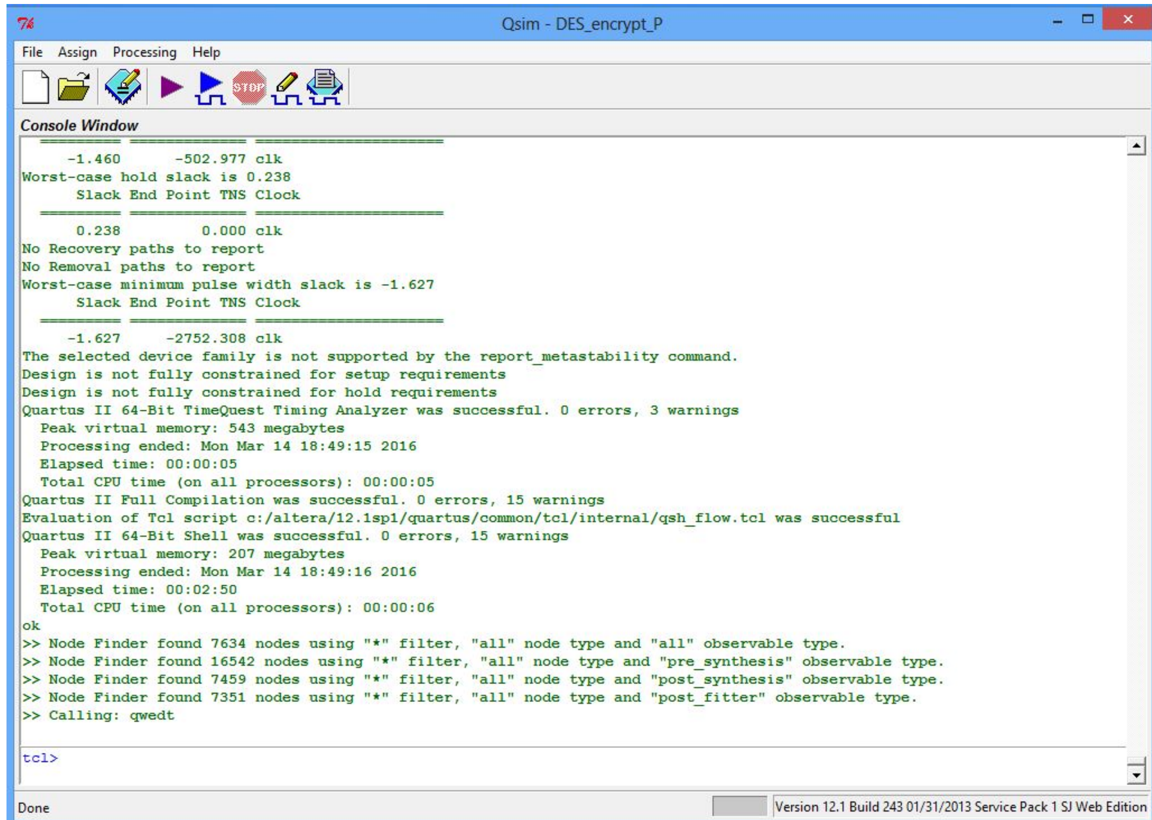


Figure 89: DES Pipelined Node Finder Time.

To execute the simulation in functional mode, the simulation takes about 26 seconds to generate the netlist as seen in Figure 90.

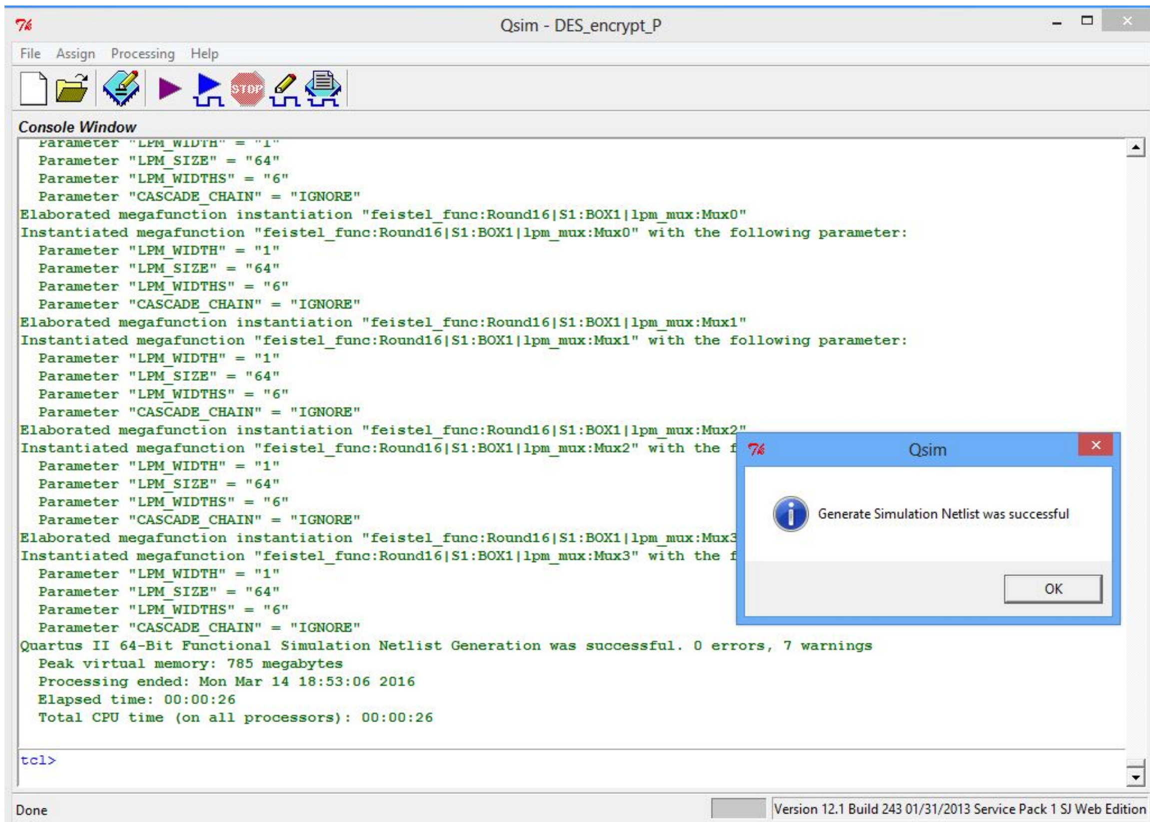


Figure 90: DES Pipelined Generate Simulation Netlist Time.

After generating the netlist, the simulation spends about 1 minute to execute the simulation as seen in Figure 91.

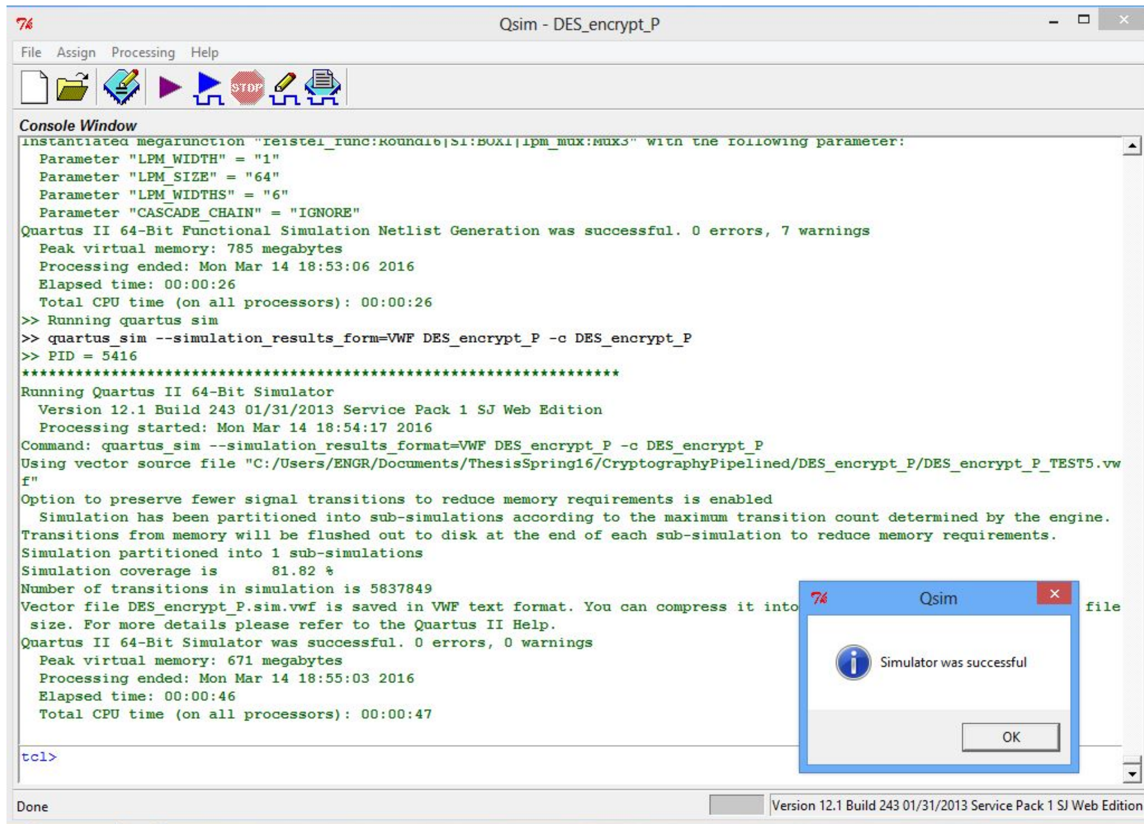


Figure 91: DES Pipelined Functional Simulation Time.

Executing the simulation in timing mode only takes a few seconds as seen in Figure 92.

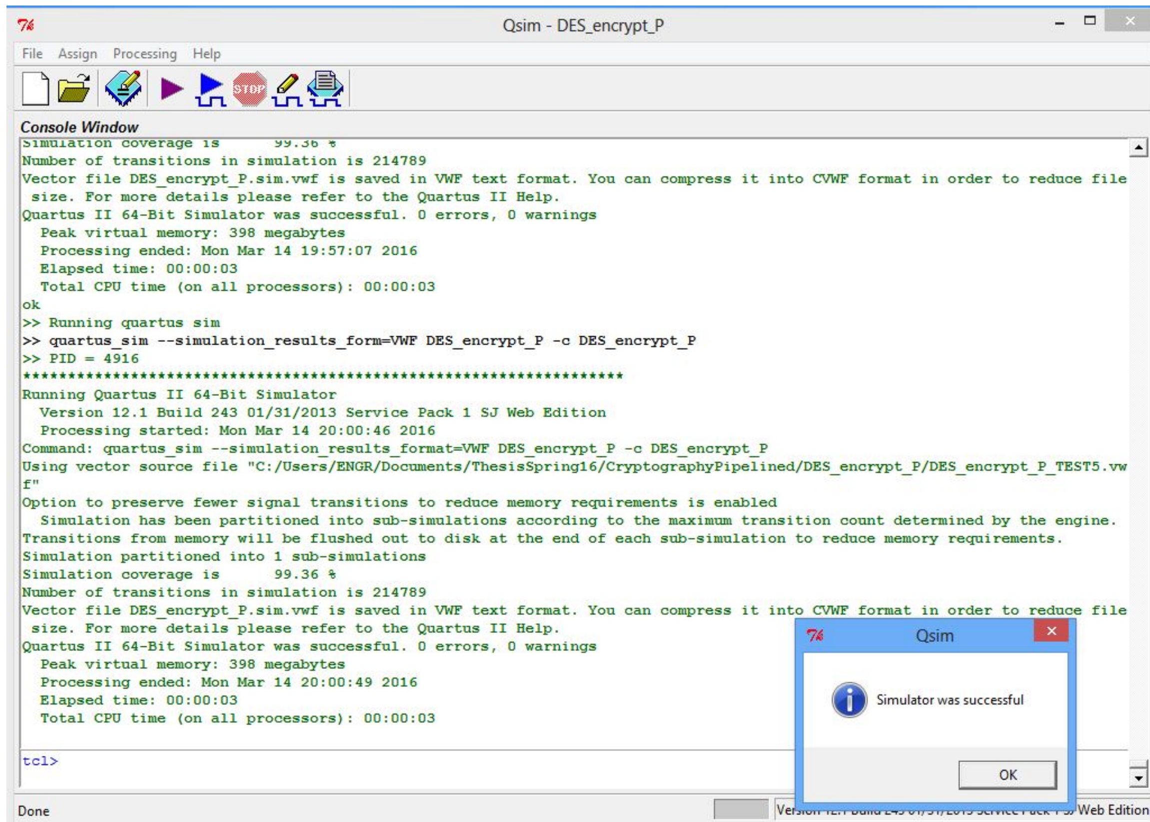


Figure 92: DES Pipelined Timing Simulation Time.

5.3.4 TDES Pipelined Encryption and Decryption

Our TDES Pipelined Encryption and Decryption schemes are implemented in 25 VHDL files in Quartus II. The Implementation is previously discussed in Chapter 4. The Analysis and Synthesis on the Encryption and Decryption schemes, in Quartus II, generates the compilation report window seen in Figure 93.

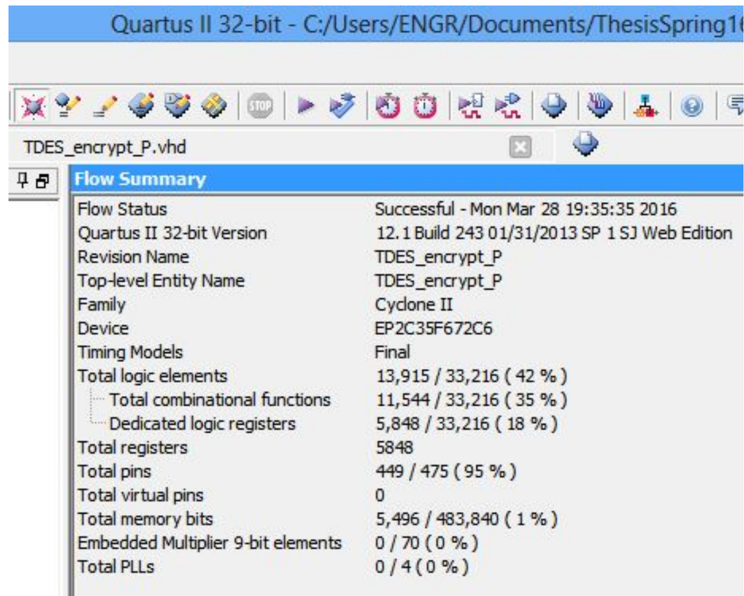


Figure 93: TDES Pipelined Analysis & Synthesis Cost Results.

As seen in Figure 93, the TDES pipelined encryption and decryption schemes utilize 42% of the Total Logic Elements, which is 13,216 of 33,216. 35% of the Total Combinational Functions are utilized: that is 11,544 of 33,216. 18% of the Dedicated Logic Registers are utilized: that is 5,848 of 33,216. The Total registers utilized are also 5,848. The total number of pins utilized is 449 of 475, which is 95%. And the Total Memory Bits utilized are 5,496 of 483,840, which is 1%.

Running the Analysis and Synthesis, on the TDES pipelined encryption and decryption schemes, takes about 4 minutes as seen in Figure 94. In Section 5.3.6 we compare the Analysis and Synthesis running time to the other designs we implemented.

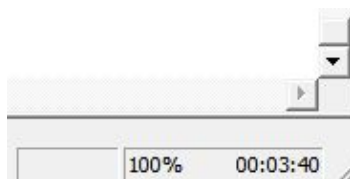


Figure 94: TDES Pipelined Analysis & Synthesis Running Time.

The timing diagram from the Altera U.P. Simulator, seen in Figure 95, is obtained by running part of the Substitution Table Known Answer Test (Test 5) as stated in Chapter 4. From these timing diagrams we can make several observations.

Figure 95: TDES Pipelined Substitution Table Known Answer Test In Timing Mode.

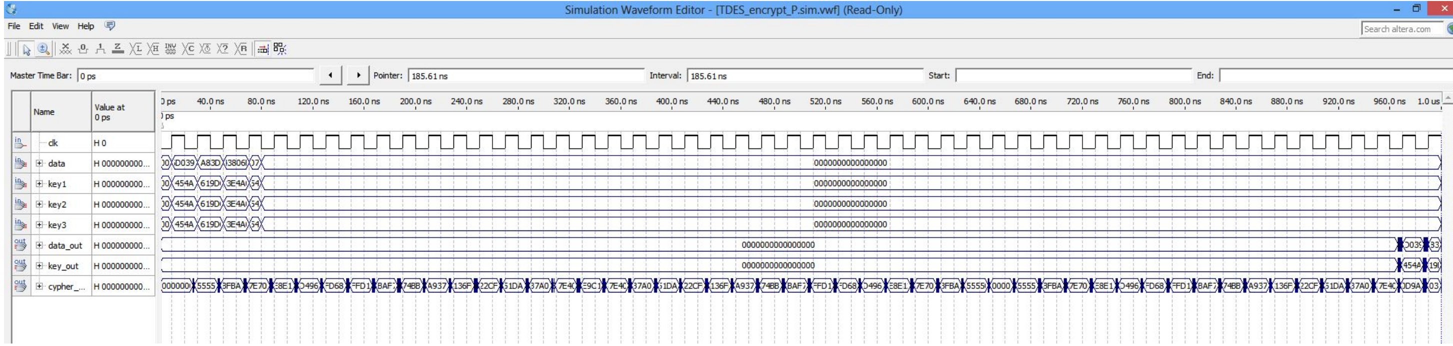
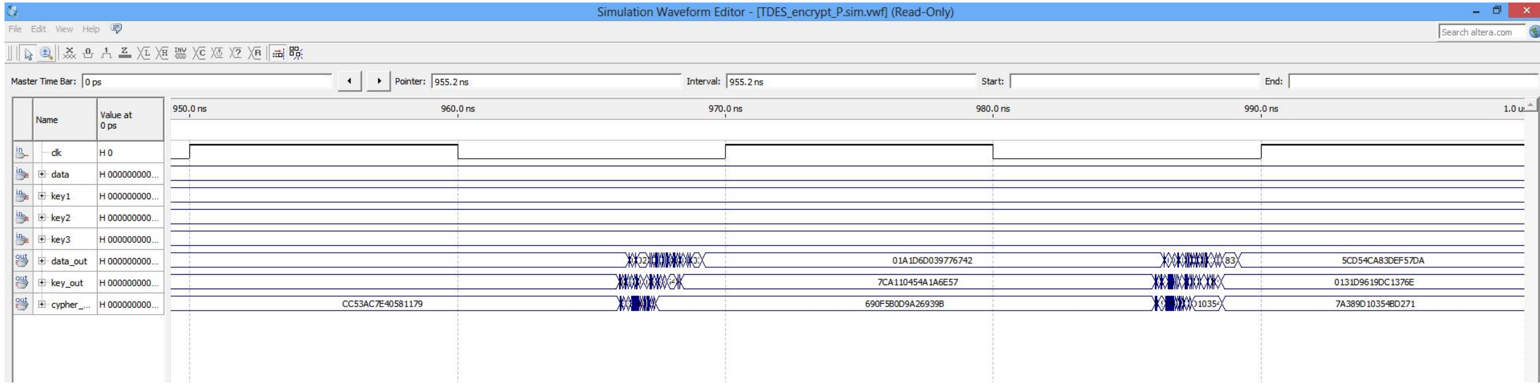


Figure 95 depicts three outputs, the data with its corresponding key and cipher data. Notice that only one key is shown since all three keys are the same in this example. The outputs of our encryption and decryption designs depend on the input data and cipher data respectively and the three input keys. Because these designs are pipelined as presented in Chapter 3, we update the inputs at the same time that our clock strikes the positive edge. Since both, our encryption and decryption designs are negative edge triggered, to avoid any set up time or hold time violations we modify the inputs on the positive edge of the clock. After the negative edge strikes, the input data will be go through our pipelined design. Since there are 48 rounds in the encryption/decryption scheme, and we have pipelined our design at every round in the DES scheme, it will take 48 clock cycles for the input data to reach the output of our design. After the initial 48-cycle delay, our systems outputs at every clock cycle. Therefore, if T is the period for one clock cycle, and T is 20 ns (since the DE2 board's internal clock is 20 ns), then the estimated throughput of our design is 3.2 Gigabits per second as shown in Equation 4.

$$throughput \approx \frac{64\ bits}{T} \approx \frac{64\ bits}{20\ ns} \approx 3.2\ Gigabits\ per\ second \qquad \text{Equation 4}$$

As seen in Figure 86, the negative edge of the clock triggers at 960 ns. The output begins to update 6 ns after the clock triggers the negative edge, and displays unstable signals for about 3 ns seconds. The output signal reaches stability after 9 ns seconds at about 969 ns. This information lets us know that our DES pipelined system design has a propagation delay of 9 ns seconds. Again it is important to note that this delay is much less than our non-pipelined designs because the amount of hardware that processes the input signal in every pipelined stage is much less as discussed in Section 5.3.4.

Figure 96: TDES Pipelined Delay Results.



In this Pipelined design, the node finder time for both the encryption and decryption is about seven and a half minutes. This result can be seen in Figure 97.

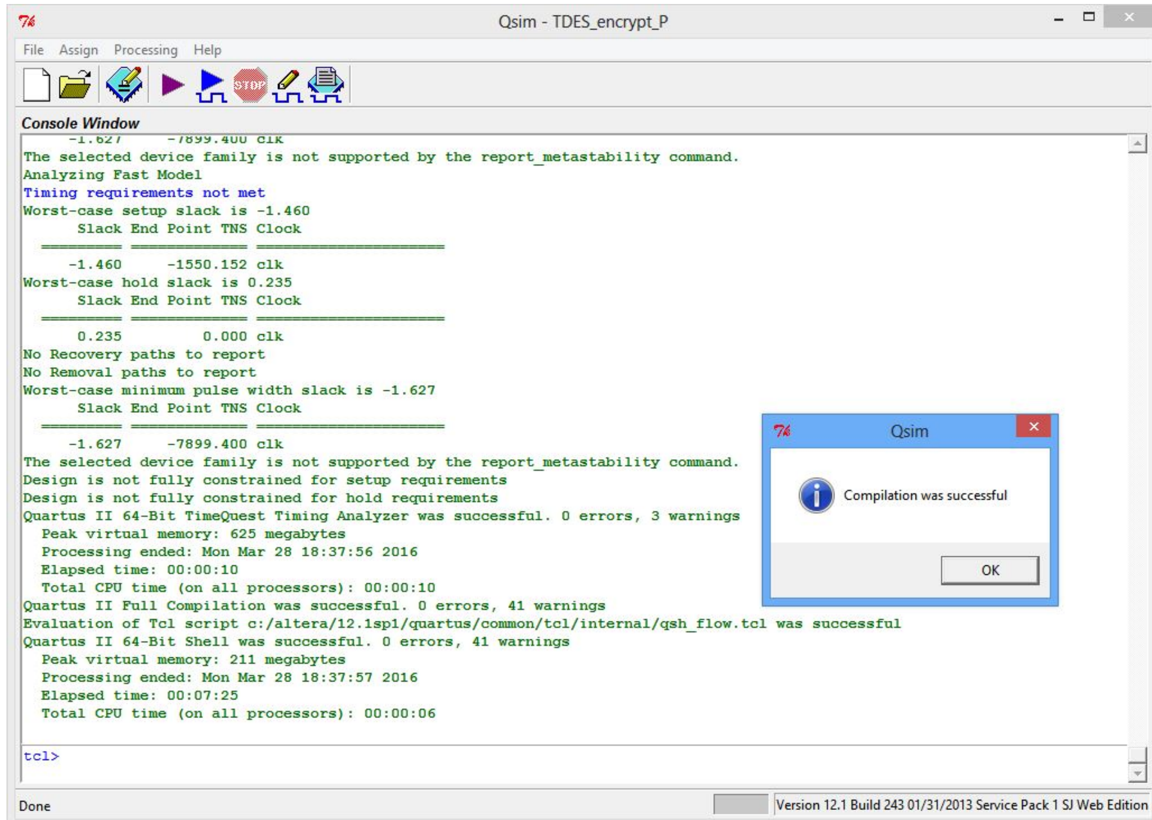


Figure 97: TDES Pipelined Node Finder Time.

To execute the simulation in functional mode, the simulation takes about a minute and a half to generate the netlist as seen in Figure 98.

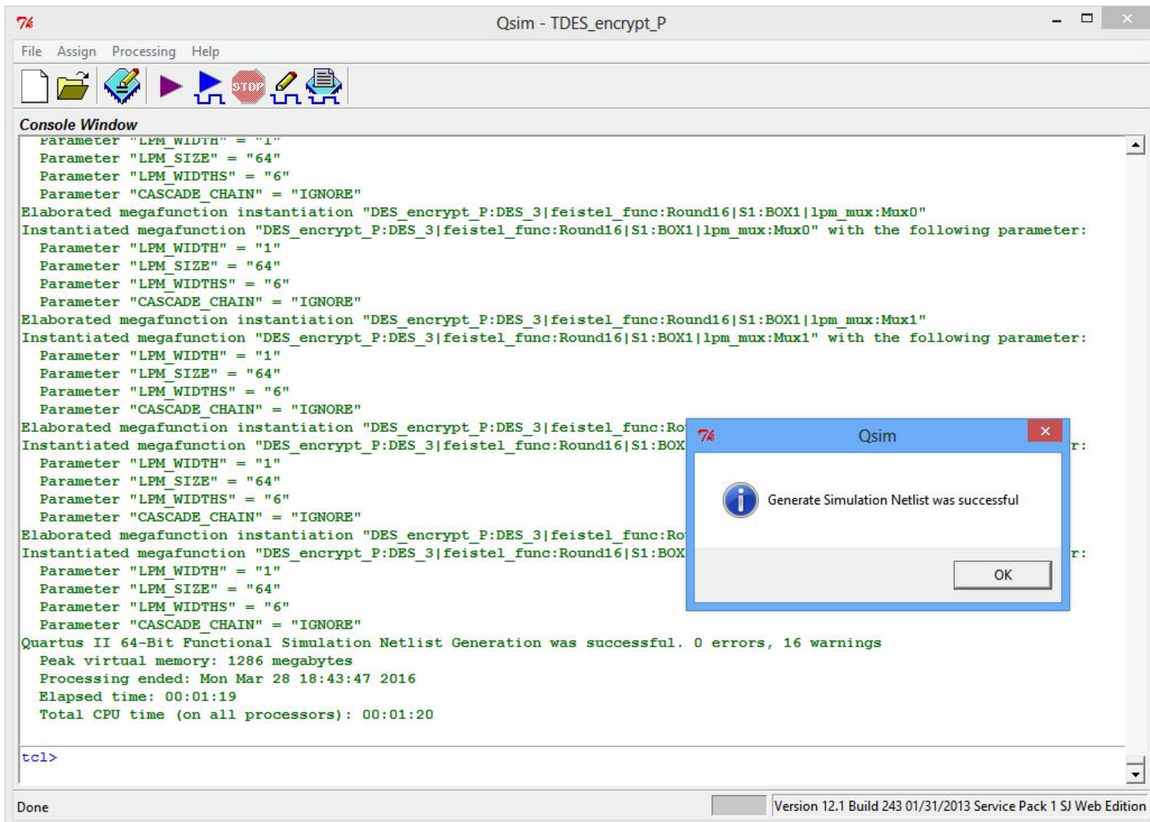


Figure 98: TDES Pipelined Generate Simulation Netlist Time.

After generating the netlist, the simulation spends about two and a half minutes to execute the simulation as seen in Figure 99.

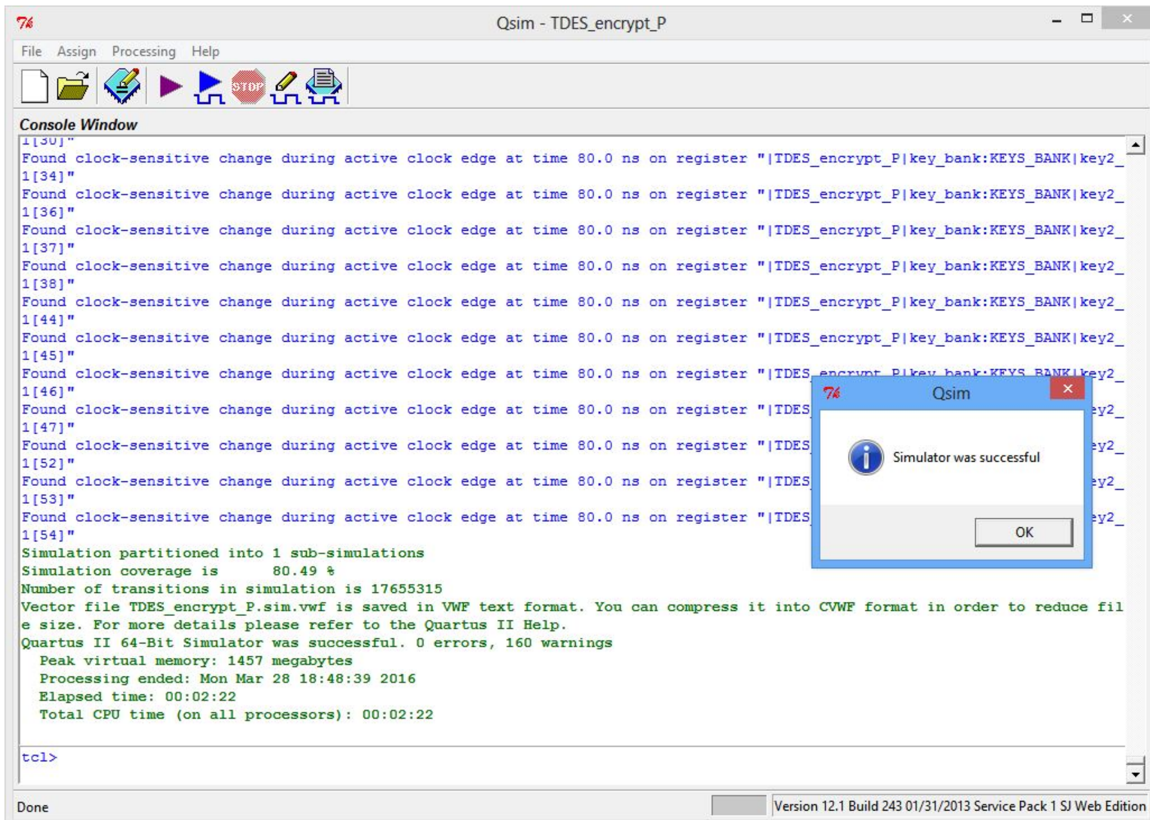


Figure 99: TDES Pipelined Functional Simulation Time.

Executing the simulation in timing mode only takes a few seconds as seen in Figure 100.

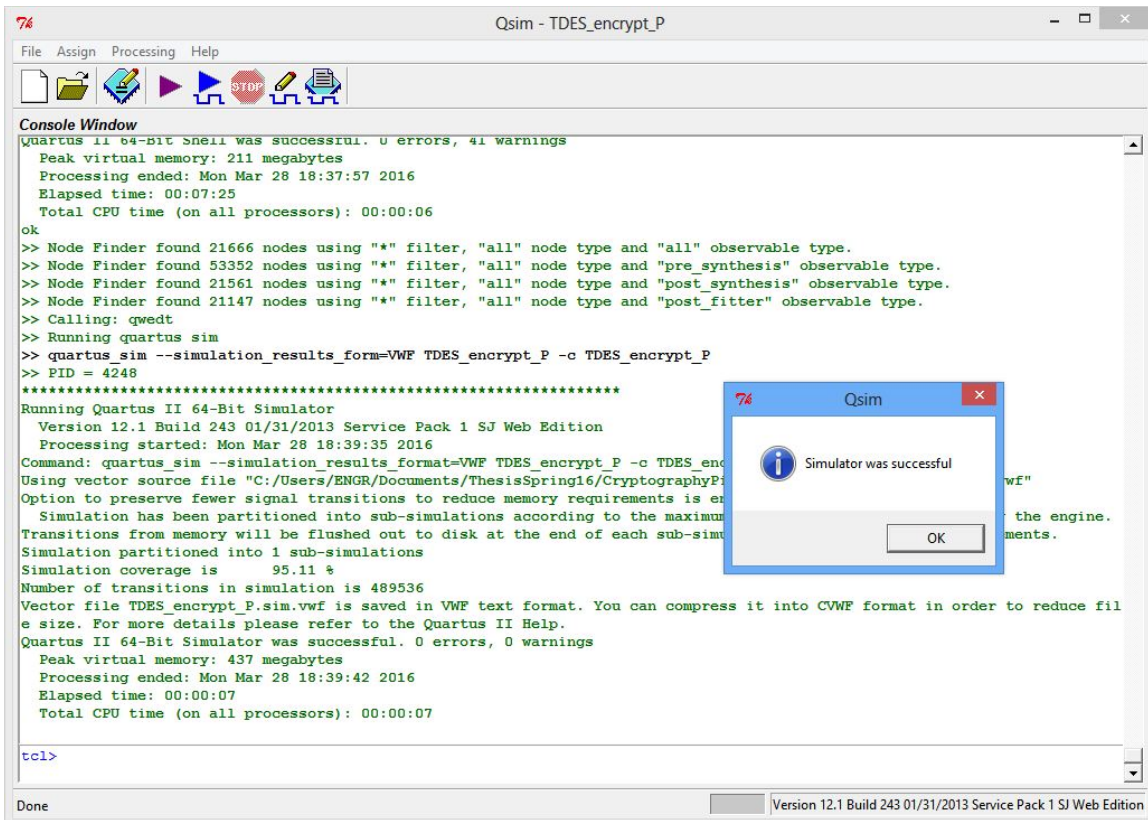


Figure 100: TDES Pipelined Timing Simulation Time.

5.3.5 DES and TDES Cost and Performance Comparison

In the previous sections, 5.3.1 through 5.3.5, we discuss the cost and performance of each of our designs separately. In Table 10 we compare each design side by side.

Table 10: DES & TDES Hardware Cost Comparison.

		DES Encrypt./ Decrypt.	TDES Encrypt./ Decrypt.	DES Pipelined Encrypt./Decrypt.	TDES Pipelined Encrypt./Decrypt.	Total Number Of Items Available
ALTERA DE2 BOARD (EP2C35F672C6) HARDWARE COST	Total Logic Elements	4,059	12,285	4,645	13,915	33,216
	Total Combinational Functions	4,059	12,285	3,842	11,544	33,216
	Dedicated Logic Registers	0	0	1,996	5,848	33,216
	Total Registers	0	0	1,996	5,848	
	Total Pins	320	448	385	449	475
	Total Memory Bits	0	0	936	5,496	483,840

The total number of logic elements in our device lets us know how much hardware space our design occupies. In the non-pipelined design, we only used about 4,000 logic elements from the 33,000 total to implement the DES encryption. If we wish to implement both, the encryption and decryption, a total of about 8,000 logic elements would be used. Since TDES is the DES scheme three times, we see that the space triples as well for the TDES scheme. Implementing both TDES encryption and decryption would take about 25,000 logic elements.

Comparing our non-pipelined vs. our pipelined designs, the number of logic elements used increases by 600 in DES and by 1,600 in TDES. The number of logic elements increase because our pipelined designs implements registers at every round in the DES scheme. As seen in Table 10, the total number of registers is zero in our non-pipelined designs, but in our pipelined DES and TDES designs the total number of registers is 1,196 and 5,848 registers respectively.

The throughput performances of our designs vary from the non-pipelined to the pipelined designs as seen in Table 11. TDES has the lowest throughput with 237 Mbps. This throughput is about 1/3 of the throughput achieved with DES. DES achieves a throughput of 640 Mbps.

Pipelining both designs, TDES and DES, greatly increased the throughput to 3.2 Gbps. That is 13.5 times more than the non-pipelined TDES design and 5 times more than the non-pipelined DES design. The throughput of our designs is based on the time delays and propagation delays obtained.

Table 11: DES & TDES Delay Analysis and Throughput Comparison.

		DES Encrypt./ Decrypt.	TDES Encrypt./ Decrypt.	DES Pipelined Encrypt./Decrypt.	TDES Pipelined Encrypt./Decrypt.
Delay Analysis & Throughput	Initial Output Delay	10 ns	10 ns	6 ns	6 ns
	Propagation Delay	90 ns	245 ns	8 ns	9 ns
	Throughput	640 Mbps	237 Mbps	3.2 Gbps	3.2 Gbps

5.3.6 Quartus II and Altera U.P. Simulator Observations

In Table 12 we compare the time executions for the Analysis and Synthesis in Quartus II.

Table 12: DES & TDES Analysis and Synthesis Times.

		DES Encrypt./ Decrypt.	TDES Encrypt./ Decrypt.	DES Pipelined Encrypt./Decrypt.	TDES Pipelined Encrypt./Decrypt.
Quarts II Observation	Analysis and Synthesis Time	3 Minutes	13 Minutes	2 Minutes	4 Minutes

It takes 3 minutes for Quartus II to execute the Analysis and Synthesis on our DES code and the time quadruples for the TDES code. TDES executes in about 13 minutes. However, the time decreases for the DES and TDES pipelined designs to 2 minutes and 4 minutes respectively.

In Table 13 we compare the times it took the Altera U.P. Simulator to execute the first five validation tests mentioned in Chapter 4. We executed the simulations in functional mode.

Table 13: DES & TDES Validation Simulation Times In Functional Mode

	DES Non-Pipelined		TDES Non-Pipelined		DES Pipelined		TDES Pipelined	
	Generate Netlist Time	Simulation Time	Generate Netlist Time	Simulation Time	Generate Netlist Time	Simulation Time	Generate Netlist Time	Simulation Time
TEST 1	25 Seconds	17 Minutes	1 Min 17 Sec	3 Hrs 14 Mins	25 Seconds	1 Minute	1 Min 16 Sec	2 Min 58 Sec
TEST 2	24 Seconds	24 Minutes	1 Min 18 Sec	3 Hrs 35 Mins	23 Seconds	1 Minute	1 Min 15 Sec	3 Min 18 Sec
TEST 3	24 Seconds	18 Minutes	1 Min 18 Sec	3 Hrs 12 Min	25 Seconds	1 Minute	1 Min 12 Sec	2 Min 53 Sec
TEST 4	25 Seconds	12 Minutes	58 Sec	1 Hr 24 Min	24 Seconds	41 Seconds	1 Min 6 Sec	2 Min 15 Sec
TEST 5	24 Seconds	8 Minutes	1 Min 14 Sec	1 Hr 5 Min	25 Seconds	37 Seconds	1 Min 19 Sec	2 Min 31 Sec

We notice that the time it takes to execute a simulation in our non-pipelined designs take longer than our pipelined designs. To execute the first three tests in the TDES non-pipelined design take more than three hours each. However, this time decreased to three minutes in our TDES pipelined design.

Executing the validation tests in timing mode only takes a minute or two. However, the zoom in function in the wave figure windows has a very large delay. Zooming into the waveform causes a stall for about a minute and a simple scroll produces stalls of about the same time. It is inconvenient to check the waveforms this way because of the stall time. It is convenient to check the waveforms in functional mode because we don't have stalls. Referring to the execution times seen in Table 13, to check the tests results of our designs we prefer running the simulation in timing mode for our TDES non-pipelined design. We prefer running the simulation in functional mode for our DES non-pipelined, DES pipelined and TDES pipelined.

5.3.7 Performance and Cost Literature Review Comparisons

In this section we compare our design's cost and performance to the designs discussed in [50] through [57]. In Table 14 we see the cost and performance of various designs, including ours, side by side.

Table 14: DES & TDES Designs Comparisons

				DES Encrypt./Decrypt.		TDES Encrypt./Decrypt.		DES Pipelined Encrypt./ Decrypt.		TDES Pipelined Encrypt./ Decrypt.	
	Number Of Stages	Hardware	Operable Clock Frequency	Hardware Cost	Performance	Hardware Cost	Performance	Hardware Cost	Performance	Hardware Cost	Performance
Edni's Design	16	EP2C35F672C6	10 MHz	4,059 LEs	640 Mbps						
	16		3.7 MHz			12,285 LEs	237 Mbps				
	16		50 MHz					4,645 LEs	3.2 Gbps	13,915 LEs	3.2 Gbps
Andoni Et Al [50]	1	Xilinx Virtex XCV-1000	91/16 MHz	596 CLBs	347.1 Mbps						
	1		91/48 MHz			596 CLBs	115.7 Mbps				
FU LI, PAN Ming [51]	16	Altera Cyclone EP1C6Q240c8	100 MHz					-	6.4 GHz		
Liakot El Al [52]	-	FLEX 10K PLD EPF10K30BC356-3	-			*	*				
TOURIA ARICH EL AL [53]	1	EPF10K30BC356-3	17.39/16 MHz	851 LCs	69.56 Mbps						
	2	EPF10K30BC356-3	17.36/8 MHz					1305 LCs	138.88 Mbps		
	4	EPF10K50BC356-3	15.89/4 MHz					2095 LCs	254.36 Mbps		
	8	EP1K100FC484-3	20.79/2 MHz					3308 LCs	665.28 Mbps		
	16	EP1K100FC484-3**	16.47 MHz					5991 LCs	1.054 Gbps		
Ke Wang [54]	16	Xilinx Virtex4	228.6 MHz						16 Gbps		
Saeid Et Al [55]	1***	Xilinx Virtex-6	1201,923/16 MHz	-	4.8 Gbps						
	16***		294.031 MHz					-	18.2 Gbps		
Ji Yao, Hongo Kang [56]	16	Xilinx Virtex-II	-					3056 LE	3.2 Gbps		
Fang Ren ET AL [57]	1***	Xilinx Virtex4	215.165 MHz							-	800.66 Mbps

*Implemented **Device specs are inconsistent with the author's cost claim

***Assumption based on the data provided -Information is not provided

As seen in Table 14, not all publications contain the cost or performance of their designs. In [52], the authors did not provide information relating to the cost and performance of their TDES design.

For the most part, we found publications in which DES, DES pipelined and TDES designs are implemented. Publications in which TDES pipelined designs are implemented are scarce.

Form Table 14, we see that the performance of DES varies from 59.56 Mbps to 4.8 Gbps. The throughput performance depends on the system's delay and the number of stages implemented. The system's delay is inversely proportional to the systems operable clock speed and frequency. The operable frequency is the frequency at which the system outputs a 64-bit cipher block. Faster operable clock speeds and a higher number of stages results in higher throughput. Our design's performance is 640 Mbps. We achieve this throughput by implementing a 16-stage design at an operable frequency of 10 MHz. Although a faster clock is implemented in [40], the authors implement only one stage. This yields the low throughput of 347.1Mbps and their operable frequency is 5.69 MHz. [53] has the lowest throughput of 69.56 Mbps because their operable frequency is 17.39 MHz and the design is one-stage also. In [55] the authors claim that the clock speed is 1201.923 MHz, but their throughput is 4.8 Gbps. This means that their operable frequency is 75.120 MHz = 1201.923 MHz / 16. We can infer from this that and the information provided that their system is a one-stage. Also, the authors in [55] do not provide any cost information on their design. In [50] the cost of their design is 596 CLBs. The cost in [53] is 851 LCs. Our cost is 4,059 LEs. Our cost is higher because we implement 16 stages as opposed to one in [50] and [53].

The performance of DES Pipelined varies from 138.88 Mbps to 18.2 Gbps. Our design's performance is 3.2 Gbps. We achieve this throughput by implementing a 16-stage pipelined design at an operable frequency of 50 MHz. In [53], the authors implemented several pipelined stages designs from 2 to 16. A two stage pipelined design yields a low throughput of 138.88 Mbps. A 16 stage pipelined design yields a throughput of 1.054 Gbps and their operable frequency is about 17 MHz. In [55], a high throughput of 18.2 Gbps is seen and the operable frequency is 294.031 MHz. No information regarding the cost of the design in [55] is provided.

In [53], the cost varies from 1305 LCs for the two-stage to 5991 LCs for the 16-stage design. Our cost is 4645 LEs. More stages available in the design yield a higher cost and a higher throughput.

Our TDES design achieves a throughput of 237 Mbps and has a cost of 12,285 LEs. Our operating frequency is 3.7 MHz. In [50], a throughput of 115.7 Mbps is achieved, and their design has a cost of 596 CLBs. Their cost is low because only one DES stage is implemented as opposed to our 16 DES stage. [52] only mentions that they implement the TDES design but no information regarding the performance and cost of their system is given.

Our TDES pipelined design achieves a throughput of 3.2 Gbps and has a cost of 13,915 LEs. Our operating frequency is 50 MHz. In [60] the authors claim that the clock speed is 215.165 MHz, but their throughput is 800.66 Gbps. This means that their operable frequency is about 13.45 MHz ≈ 215.165 MHz / 16.

CHAPTER VI

CONCLUSIONS AND FUTURE WORK

Undoubtedly, cryptography maintains the confidentiality component of the security triad. Without confidentiality, the black hat hacker community would have their way with ease, implementing attacks on computer systems around the world gaining personal information and causing great financial losses. In March 2015, Target was ordered to compensate 10 million dollars to the victims affected by Target's security breach in 2013. "The data breach at Target, which took place during the height of the 2013 holiday shopping season, was one of the largest in U.S. corporate history [64]." In this breach, the personal and financial information of approximately 40 million customers was exposed yielding in card frauds, inconvenient charges and fees. Target was not the only company that suffered losses; most card companies offer to reimburse 100% of card frauds and they also experienced a loss. Sony also suffered millions in loss due to a breach by a famous hack group, named Anonymous, in 2014. Anonymous hacked Sony, threatened to leak information, incur higher damages and acts of terror if Sony's film, The Interview, was released. The release only yielded revenue of 31 million by January 2015 as the budget for the film was 44 million [65]. The market demands include reliability, security and fast performance. Although the cost increases with performance, more concern is given to acquiring higher performance and security,

rather than saving some extra cash. The results of not having a strong security outweighs saving on the cost.

In this thesis we focused on improving the performance of two commonly used encryption schemes: DES and TDES. Alteras DE2 board technology EP2C35F672C6 is the hardware platform that allowed us to measure the performance of our implementations, which were coded and tested in Altera's Quartus II software and Altera's University Program Simulator. These programs also provided information regarding the cost of our implementations.

In Chapter I we demonstrated the common Alice, Bob and Eve eavesdropping scenario that shows the need for secure communications. We overviewed the importance of cryptography, the up to day relevance, the history, the applications and how it has chanced the course of history. In Table 2 we compared the computational power and the time required to run a brute force attack on these schemes. The chart shows that DES and TDES are still reliable. DES is the only scheme that has been susceptible to a successful brute force attack in less than 24 hours. However, this is achieved at the expense of great cost and resources, and since this scheme is only used for applications of minor importance, a brute force attack on this scheme is not worth the hassle. TDES, on the other hand, is not susceptible to brute force attack. It would take thousands of years to traverse through all the possible key combinations. Further more, we discussed the EDA tools we used to code and test the logic designs: Altera's Quartus II and Altera's University Program Simulator. We also showed that, even though the technology of our hardware platform, Altera's EP2C35F672C6, only supports a 50 MHz clock speed, we may be able achieve a throughput of 3.2 Gbps if we experience no setup time or hold time violations.

We presented the logic designs of our non-pipelined implementations of DES and TDES in Chapter 2 for both, the encryption and decryption processes. We presented all the different

components of the cryptography schemes including the components in the key scheduling as presented in [60] and [62]. In [60] and [62], the authors show the key scheduling for the encryption processes only. They state that, for the decryption, the keys are applied in reverse order. Instead of using the same encryption key-scheduling scheme, for the decryption key scheduling, we implemented a right rotation on the half strings of bits. By applying the right rotations, we had flexibility to pipeline our designs as presented in chapter 3. The logic designs of all our components in VHDL, for the non-pipelined designs, are overviewed in this chapter as well. Every VHDL file is referenced in Appendix A. We also followed though the schemes with a numerical example using the key 0x133457799BBCDFF1 and the data input 0x012345679ABCDEF. At every stage of the key scheduling processes and the encryption/decryption processes we show the results as the inputs traverse the algorithms. The total number of VHDL files generated to implement our non-pipelined schemes is 25.

In Chapter 3 we covered in detail the logic designs in VHDL, the components and the modifications incurred to achieve the pipelined designs for DES and TDES. The VHDL files are referenced in the appendix section. The main changes in the DES pipelined design only concerned the addition of memory elements (Flip Flops) also known as sequential elements or registers. These registers buffered the data as it traversed the 16 rounds of DES. This means that the pipeline of our DES design was 16 stages. As discussed in chapter 2, our LEFT shifts implementation, in the decryption key scheduling process, allowed us to pipeline through the 48 rounds of TDES with ease. The pipeline for our TDES design is 48 stages. The literature we have reviewed shows that the pipeline depth for their TDES is 3 stages [57]. They pipelined after every DES component and did not sub-pipeline the DES components. Three keys were required for TDES. To properly match the keys with each DES component in TDES pipelined,

we implemented a key bank that buffers the keys for either zero cycles, 15 cycles or 31 cycles. The total number of VHDL files generated to implement our pipelined schemes is 25.

We successfully validated our encryption and decryption schemes as specified in the NIST special publications 800-17 and 800-20 [26] [27]. In chapter 4 we demonstrated that the results of our simulations were consistent with the expected results given in 800-17 and 800-20. Using Altera's University Program Simulator, we showed the timing waveforms of the six validation tests. To prevent this document from rapidly populating with many figures, we showed the figures for the pipelined schemes only. The only validation test that required extra logic design, to implement, was the MONTECARLO test. We demonstrated the feedback logic configuration that updates the input data and the key. A scaled down version of the MONTECARLO presented in the NIST publications was implemented. Instead of updating the key every 10000 ciphers, we updated the key every 128 ciphers. Our pipelined logic designs passed the MONTECARLO validation test along with the other five validation tests.

We presented the cost and performance of our implementation in Chapter 5. Based on the cost information gathered from Altera's Quartus II software compilation results, the cost tripled from DES to TDES logic implementation in VHDL. The cost information yielded a cost of 4,059 Logic Elements for DES non-pipelined and a cost of 12,285 Logic Elements for TDES non-pipelined. This is consistent with what we know because TDES involves 3 DES components. The cost from DES to TDES triples. In our pipelined designs we saw an increase in the cost due to the memory elements we added in every stage of the pipeline. The cost for our DES pipelined is 4,645 and our TDES pipelined is 13,915. From the cost results seen in Table 14, we see that the cost of our logic designs is higher that those seen in the other literatures. However, their low cost designs yield a lower performance even when their clock frequency is

higher. As explained before, the low cost designs require more cycles per full encryption, thus decreasing the throughput. Altera's EP2C35F672C6 technology allowed us run the clock at the maximum speed of 50 Mhz. This is a lower frequency compared to the clock speed of the devices in the other literatures. Our non-pipelined designs achieved low throughputs of 640 Mbps and 237 Mbps for our DES and TDES. Pipelining both designs allowed us to achieved a higher throughput of 3.2 Gbps. In Table 14, we see that our DES performed better than most of the designs in the other literatures while our TDES outperformed all the other designs.

In this thesis we have successfully implemented, and evaluated two common encryption schemes in ECB mode: DES and TDES. Even though our designs operate at the low clock speed of 50 MHz, by increasing the cost (adding more stages), we were able to increase the throughput up to 3.2 Gbps. Future work for this thesis will be to implement, validate and test the DES and TDES schemes in Cipher Block Chaining (CBC) mode. CBC mode takes advantage of the feedback configuration and extra input vectors to create a more secure encryption. CBC guarantees that, as long the input vector is different, the cipher generated will be different even when the same data is encrypted again.

REFERENCES

[1] "FPGA Acceleration | Synective Labs." Synective Labs. N.p., n.d. Web. 03 Oct. 2016. <http://www.synective.se/index.php/acceleration/fpga-acceleration/>

[2] "The CPU, Machine Language, and the Stored Program Model." Introduction to Computers (Aut 01). N.p., n.d. Web. 03 Oct. 2016. <http://www.cs.brandeis.edu/~tim/Classes/Autumn01/CS2a/Notes/pc.html>

[3] Electrical Engineering Stack Exchange. N.p., n.d. Web. 3 Oct. 2016. <http://electronics.stackexchange.com/questions/101472/how-can-an-fpga-outperform-a-cpu>

[4] Garry, Pat. "Truths and Myths About FPGA Acceleration." Ryft. N.p., n.d. Web. 3 Oct. 2016. <http://www.ryft.com/blog/truths-and-myths-about-fpga-acceleration>

[5] "Project Catapult." Microsoft. N.p., 1 June 2011. Web. 3 Oct. 2016. <https://www.microsoft.com/en-us/research/project/project-catapult/>

[6] @TDaytonPM. "Microsoft Extends FPGA Reach From Bing To Deep Learning." The Next Platform. N.p., 01 Sept. 2015. Web. 03 Oct. 2016. <http://www.nextplatform.com/2015/08/27/microsoft-extends-fpga-reach-from-bing-to-deep-learning/>

[7] Research, Microsoft. "Microsoft Research Faculty Summit 2015 - Co-located Events." Microsoft Research Faculty Summit 2015 - Co-located Events. N.p., n.d. Web. 03 Oct. 2016. <http://research.microsoft.com/en-us/um/redmond/events/fs2015/co-located-events.aspx>

[8] Metz, Cade. "Microsoft Bets Its Future on a Reprogrammable Computer Chip." Wired. N.p., 25 Sept. 2016. Web. 3 Oct. 2016. <https://www.wired.com/2016/09/microsoft-bets-future-chip-reprogram-fly/?yptr=yahoo>

[9] Knight, Shawn. "Microsoft's Project Catapult Is Why Intel Bought FPGA-maker Altera for $16.7 Billion Last Year." Techspot. N.p., 26 Sept. 2016. Web. 3 Oct. 2016. <http://www.techspot.com/news/66461-microsoft-project-catapult-why-intel-bought-fpga-maker.html>

[10] "POWER8 Coherent Accelerator Processor Interface (CAPI)." IBM. N.p., n.d. Web. 03 Oct. 2016. <https://www-304.ibm.com/webapp/set2/sas/f/capi/home.html>

[11] Wile, Bruce. Coherent Accelerator Processor Interface (CAPI) for POWER8 Systems White Paper (n.d.): n. pag. IBM. 29 Sept. 2014. Web. 3 Oct. 2016. <https://www-304.ibm.com/webapp/set2/sas/f/capi/CAPI_POWER8.pdf>

[12] "Programmable Logic." Future Electronics. N.p., n.d. Web. 3 Oct. 2016. <http://www.futureelectronics.com/en/programmable-logic/programmable-logic.aspx>

[13] Barr, Michael. "How Programmable Logic Works." Neutrino. N.p., 1999. Web. 3 Oct. 2016. <https://cyphunk.files.wordpress.com/2006/02/Programmable%20Logic%20Overview%20-%20PLD,%20CPLD,%20FPGA.pdf>

[14] Morris, Kevin. "Xilinx vs. Altera." Xilinx vs. Altera. N.p., 25 Feb. 2014. Web. 03 Oct. 2016. <http://www.eejournal.com/archives/articles/20140225-rivalry/>

[15] "Highest Performance & Integration on FinFET." Virtex UltraScale+. N.p., n.d. Web. 03 Oct. 2016. <https://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale-plus.html>

[16] By Logging In, You Agree to Our Terms of Service. "STRATIX 10 FPGA AND SOC." Stratix 10. N.p., n.d. Web. 03 Oct. 2016. <https://www.altera.com/products/fpga/stratix-series/stratix-10/overview.html>

[17] Pangrle, By Barry. "FinFET Vs. Tri-Gate." Semiconductor Engineering. N.p., n.d. Web. 03 Oct. 2016. <http://semiengineering.com/finfet-vs-tri-gate/>

[18] "UltraScale Architecture and Product Overview." (n.d.): n. pag. Xilinx. 27 Sept. 2016. Web. 3 Oct. 2016. <http://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf>

[19] J. Zhang and G. Qu, "A survey on security and trust of FPGA-based systems," Field-Programmable Technology (FPT), 2014 International Conference on, Shanghai, 2014, pp. 147-152. doi: 10.1109/FPT.2014.7082768

[20] Davidson, Allan. A New FPGA Architecture and Leading-Edge FinFET Process Technology Promise to Meet Next- Generation System Requirements (n.d.): n. pag. Altera. June 2015. Web. 3 Oct. 2016. <https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/wp/wp-01220-hyperflex-architecture-fpga-socs.pdf>

[21] Xilinx, Inc. Kintex UltraScale+ FPGAs Data Sheet: DC and AC Switching Characteristics (DS922) (n.d.): n. pag. Xilinx. 9 May 2016. Web. 3 Oct. 2016. <http://www.xilinx.com/support/documentation/data_sheets/ds922-kintex-ultrascale-plus.pdf>

[22] Morris, Kevin. "Xilinx 1, Intel 0 Big X Scores First in the New Rivalry." Electronic Engineering Journal. N.p., 3 Feb. 2016. Web. 3 Oct. 2016. <http://www.eejournal.com/archives/articles/20160203-xilinx/>

[23] Lazzaro, John. "Altera Parts History." University Of California, Berkeley. N.p., n.d. Web. 3

Oct. 2016. <http://www-inst.eecs.berkeley.edu/~cs294-59/fa10/resources/Altera-history/Altera-history.html>

[24] Terasic. N.p., n.d. Web. 3 Oct. 2016. <https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=13&List=Simple#Category53>

[25] Altera. "Cyclone II Device Handbook, Volume 1." (n.d.): n. pag. Altera. Feb. 2007. Web. 3 Oct. 2016. <https://www.altera.com/en_US/pdfs/literature/hb/cyc2/cyc2_cii5v1.pdf>

[26] S. Keller and M. Smid, "Modes of operation validation system (MOVS): Requirements and procedures."

[27] S. Keller, "Modes of operation validation system for the triple data encryption algorithm (TMOVS): Requirements and procedures."

[28] "Design and Verification of Digital Systems." Scalable Hardware Verification with Symbolic Simulation (n.d.): 7-33. Web. 3 Oct. 2016. <http://web.eecs.umich.edu/~valeria/research/thesis/thesis2.pdf>

[29] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers and Z. Zhang, "High-Level Synthesis for FPGAs: From Prototyping to Deployment," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 30, no. 4, pp. 473-491, April 2011. doi: 10.1109/TCAD.2011.2110592

[30] "Quartus Prime Design Software." Altera. N.p., n.d. Web. 03 Oct. 2016. <https://dl.altera.com/13.0sp1/?edition=web>

[31] Wade, Trappe & Lawrence Washington. "Introduction to Cryptography with Coding Theory"

[32] S. Kumar. "ELEE 6399.03: Network Systems Security Module 1 Introduction/ Security Triad CIA," August 2014

[33] S. Kumar. "ELEE 6399.03: Network Systems Security Module 10 Cryptography concepts Symmetric/Asymmetric Cryptography," August 2014

[34] Higgins, Andrew, & De Freytas-Tamura, Kimiko. "Attacks in Paris" The New York Times, 19 November 2015. Web. 17 December 2015. <http://www.nytimes.com/news-event/attacks-in-paris>

[35] Serrano, Richard A. "Tashfeen Malik messaged Facebook friends about her support for jihad." Los Angeles Times, 14 December 2015. Web. 17 December 2015. <http://www.latimes.com/local/lanow/la-me-ln-malik-facebook-messages-jihad-20151214-story.html >

[36] Birnbaum, Michael, Sound Mekhennet, and Ellen Nakashima. "Paris Attack Planners Used Encrypted Apps, Investigators Believe." Washington Post. The Washington Post, 17 Dec. 2015. Web. 03 Oct. 2016. <https://www.washingtonpost.com/world/europe/paris-attack-planners-used-encrypted-apps-investigators-believe/2015/12/17/e798d288-a4de-11e5-8318-bd8caed8c588_story.html>

[37] Bereznak, Alyssa. "Where do the presidential candidates stand on encryption? A handy guide." Yahoo, 17 December 2015. Web. 17 December 2015. <https://www.yahoo.com/politics/where-do-the-presidential-candidates-stand-on-134651347.html>

[38] Chokkattu, Julian. "BlackBerry CEO criticizes Apple for refusal to decrypt data for law enforcement." Yahoo, 17 December 2015. Web. 17 December 2015. <http://sports.yahoo.com/news/blackberry-ceo-criticizes-apple-refusal-183417613.html;_ylt=AwrXnCelWnRWCQYAI4jQtDMD;_ylu=X3oDMTByb2lvbXVuBGNvbG8DZ3ExBHBvcwMxBHZ0aWQDBHNlYwNzcg-->

[39] Ferrigno, Lorenzo, and Charles Riley. "FBI Director: We Bought 'a Tool' to Hack Terrorist's IPhone." CNNMoney. Cable News Network, 7 Apr. 2016. Web. 03 Oct. 2016. <http://money.cnn.com/2016/04/07/technology/fbi-iphone-hack-san-bernardino/>

[40] "Advanced Encryption Standard – AES." Lecture Notes in Computer Science (2005): n. pag. NIST. 26 Nov. 2001. Web. 3 Oct. 2016. <http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf>

[41] "Teaching With Documents: The Zimmermann Telegram." National Archives. N.p., n.d. Web. 3 Oct. 2016. <https://www.archives.gov/education/lessons/zimmermann/>

[42] Gladwin, Lee. "Alan Turing, Enigma, and the Breaking of German Machine Ciphers in World War II." (n.d.): n. pag. National Archives. Web. 3 Oct. 2016. <https://www.archives.gov/publications/prologue/1997/fall/turing.pdf>

[43] Stallings, Williams. Cryptography Network and Security: Principals and Practices 4th Edition. < http://www.inf.ufsc.br/~bosco.sobral/ensino/ine5680/material-cripto-seg/2014-1/Stallings/Stallings_Cryptography_and_Network_Security.pdf>

[44] http://www.businesscloudnews.com/2016/08/02/aws-microsoft-google-and-ibm-continue-cloud-market-dominance/

[45] S. Keller and M. Smid, "Modes of operation validation system (MOVS): Requirements and procedures."

[46] S. Keller, "Modes of operation validation system for the triple data encryption algorithm (TMOVS): Requirements and procedures."

[47] M. Smid and D. Branstad, "Data encryption standard: past and future," vol. 76, no. 5, pp. 550–559.

[48] S. Mukhopadhyay, “Chapter 2: Data encryption standard (DES)..”

[49] Stallings, William. Cryptography and Network Security: Principles and Practice. Sixth ed. Upper Saddle River: Pearson, 2014. Print.

[50] I. Andoni, P. Chodowiec, and J. Radzikowski, “Hardware implementation of IPSeccryptographic transformations,” 2001, < http://ece.gmu.edu/ courses/ECE636/project/reports/IAn PCh JRa.pdf.>

[51] L. Fu and M. Pan, "A Simplified FPGA Implementation Based on an Improved DES Algorithm," Genetic and Evolutionary Computing, 2009. WGEC '09. 3rd International Conference on, Guilin, 2009, pp. 227-230. doi: 10.1109/WGEC.2009.11

[52] L. Ali, N. A. M. Yunus, H. Jaafar, R. Wagiran and E. Low, "Implementation of triple data encryption algorithm using VHDL," Semiconductor Electronics, 2004. ICSE 2004. IEEE International Conference on, 2004, pp. 5 pp.-. doi: 10.1109/SMELEC.2004.1620907

[53] T. Arich and M. Eleuldj, "Hardware implementations of the data encryption standard," Microelectronics, The 14th International Conference on 2002 - ICM, 2002, pp. 100-103.

doi: 10.1109/ICM-02.2002.1161506

[54] K. Wang, "An Encrypt and Decrypt Algorithm Implementation on FPGAs," Semantics, Knowledge and Grid, 2009. SKG 2009. Fifth International Conference on, Zhuhai, 2009, pp. 298-301. doi: 10.1109/SKG.2009.74

[55] S. Taherkhani, E. Ever and O. Gemikonakli, "Implementation of Non-Pipelined and Pipelined Data Encryption Standard (DES) Using Xilinx Virtex-6 FPGA Technology," Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on, Bradford, 2010, pp. 1257-1262. doi: 10.1109/CIT.2010.227

[56] J. Yao and H. Kang, "FPGA implementation of dynamic key management for DES encryption algorithm," Electronic and Mechanical Engineering and Information Technology (EMEIT), 2011 International Conference on, Harbin, Heilongjiang, 2011, pp. 4795-4798. doi: 10.1109/EMEIT.2011.6024111

[57] F. Ren, L. Chen, and T. Zhang, “3des implementation based on FPGA,” in Emerging Research in Web Information Systems and Mining (G. Zhiguo, X. Luo, J. Chen, F. L. Wang, and J. Lei, eds.), vol. 238, pp. 218–224, Springer Berlin Heidelberg.

[58] Brown, Stephen D., and Zvonko G. Vranesic. Fundamentals of Digital Logic with VHDL Design. New York, NY: McGraw-Hill, 2009. Print.

[59] Perry, Douglas L. VHDL: Programming by Example. New York: McGraw-Hill, 2002. Print.

[60] "DATA ENCRYPTION STANDARD (DES)." NIST. FIPS, 25 Oct. 1999. Web. 6 Oct.

2016. <http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf>

[61] S. Kumar. “ELEE 6399.03: Network Systems Security Module 11B Symmetric Cryptography Data Encryption Standand – DES & 3DES,” August 2014

[62] Barker, William C., and Elaine Barker. "Recommendation for the Triple Data Encryption Algorithm (TDEA) Block Cipher." NIST Special Publication 800-67 (n.d.): n. pag. NIST. National Institute of Standards and Technology, Jan. 2012. Web. 8 Oct. 2016.

[63] Patterson, David A., and John L. Hennessy. Computer Organization and Design: The Hardware/software Interface. San Francisco, CA: Morgan Kaufmann, 2007. Print.

[64] Thursday, On. "Target Will Pay Hack Victims $10 Million." CNNMoney. Cable News Network, 19 Mar. 2015. Web. 11 Oct. 2016. < http://money.cnn.com/2015/03/19/technology/security/target-data-hack-settlement/>

[65] Isidore, Chris. "'The Interview' Was Expected to Bring in $100 Million for Sony." CNNMoney. Cable News Network, 14 Dec. 2014. Web. 11 Oct. 2016. < http://money.cnn.com/2014/12/18/media/sony-the-interview-box-office/>

APPENDIX A

APPENDIX A

VHDL FILES

DES_decrypt.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;

ENTITY DES_decrypt IS
      PORT(cipher_data: IN std_logic_vector(63 DOWNTO 0);
                 K: IN std_logic_vector(63 DOWNTO 0);

                 cipher_out: OUT std_logic_vector(63 DOWNTO 0);
                 K_out: OUT std_logic_vector(63 DOWNTO 0);
                 data: OUT std_logic_vector(63 DOWNTO 0));
END DES_decrypt;

ARCHITECTURE decryption OF DES_decrypt IS

      COMPONENT KS_D
      PORT( KEY: IN std_logic_vector(63 DOWNTO 0);

      K1,K2,K3,K4,K5,K6,K7,K8,K9,K10,K11,K12,K13,K14,K15,K16: OUT
std_logic_vector(47 DOWNTO 0));
      END COMPONENT;

      COMPONENT init_permutation
      PORT(data: IN std_logic_vector(63 DOWNTO 0);
                 Ln: OUT std_logic_vector(31 DOWNTO 0);
                 Rn: OUT std_logic_vector(31 DOWNTO 0));
      END COMPONENT;

      COMPONENT feistel_func
      PORT(Rn: IN std_logic_vector(31 DOWNTO 0);
                 keyx: IN std_logic_vector(47 DOWNTO 0);
                 Rn_1: OUT std_logic_vector(31 DOWNTO 0));
      END COMPONENT;

      COMPONENT inv_init_permutation
      PORT(Ln: IN std_logic_vector(31 DOWNTO 0);
                 Rn: IN std_logic_vector(31 DOWNTO 0);
                 cipher_data: OUT std_logic_vector(63 DOWNTO 0));
      END COMPONENT;
```

```
      SHARED VARIABLE
K1,K2,K3,K4,K5,K6,K7,K8,K9,K10,K11,K12,K13,K14,K15,K16:
std_logic_vector(47 DOWNTO 0);

      SHARED VARIABLE
fR0_K1,fR1_K2,fR2_K3,fR3_K4,fR4_K5,fR5_K6,fR6_K7,fR7_K8,fR8_K9,fR9_K
10,fR10_K11,fR11_K12,fR12_K13,fR13_K14,fR14_K15,fR15_K16:
std_logic_vector(31 DOWNTO 0);

      SIGNAL
L0,L1,L2,L3,L4,L5,L6,L7,L8,L9,L10,L11,L12,L13,L14,L15,L16:
std_logic_vector(31 DOWNTO 0);
      SIGNAL
R0,R1,R2,R3,R4,R5,R6,R7,R8,R9,R10,R11,R12,R13,R14,R15,R16:
std_logic_vector(31 DOWNTO 0);
      SIGNAL data_int: std_logic_vector(63 DOWNTO 0);

BEGIN

      Key_schedule : KS_D PORT
MAP(K,K1,K2,K3,K4,K5,K6,K7,K8,K9,K10,K11,K12,K13,K14,K15,K16);

      Inital_Perm : init_permutation PORT MAP(cipher_data,L0,R0);

      ---------------------------16 ROUNDS--------------------------
----------
      Round1 : feistel_func PORT MAP(R0,K1,fR0_K1);
      R1 <= L0 XOR fR0_K1;
                                          L1 <= R0;

      Round2 : feistel_func PORT MAP(R1,K2,fR1_K2);
      R2 <= L1 XOR fR1_K2;
                                          L2 <= R1;

      Round3 : feistel_func PORT MAP(R2,K3,fR2_K3);
      R3 <= L2 XOR fR2_K3;
                                          L3 <= R2;

      Round4 : feistel_func PORT MAP(R3,K4,fR3_K4);
      R4 <= L3 XOR fR3_K4;
                                          L4 <= R3;

      Round5 : feistel_func PORT MAP(R4,K5,fR4_K5);
      R5 <= L4 XOR fR4_K5;
                                          L5 <= R4;

      Round6 : feistel_func PORT MAP(R5,K6,fR5_K6);
      R6 <= L5 XOR fR5_K6;
                                          L6 <= R5;

      Round7 : feistel_func PORT MAP(R6,K7,fR6_K7);
      R7 <= L6 XOR fR6_K7;
                                          L7 <= R6;

      Round8 : feistel_func PORT MAP(R7,K8,fR7_K8);
```

```
      R8 <= L7 XOR fR7_K8;
                                        L8 <= R7;

      Round9 : feistel_func PORT MAP(R8,K9,fR8_K9);
      R9 <= L8 XOR fR8_K9;
                                        L9 <= R8;

      Round10 : feistel_func PORT MAP(R9,K10,fR9_K10);
      R10 <= L9 XOR fR9_K10;
                                        L10 <= R9;

      Round11 : feistel_func PORT MAP(R10,K11,fR10_K11);
      R11   <= L10 XOR fR10_K11;
                                        L11 <= R10;

      Round12 : feistel_func PORT MAP(R11,K12,fR11_K12);
      R12 <= L11 XOR fR11_K12;
                                        L12 <= R11;

      Round13 : feistel_func PORT MAP(R12,K13,fR12_K13);
      R13 <= L12 XOR fR12_K13;
                                        L13 <= R12;

      Round14 : feistel_func PORT MAP(R13,K14,fR13_K14);
      R14 <= L13 XOR fR13_K14;
                                        L14 <= R13;

      Round15 : feistel_func PORT MAP(R14,K15,fR14_K15);
      R15 <= L14 XOR fR14_K15;
                                        L15 <= R14;

      Round16 : feistel_func PORT MAP(R15,K16,fR15_K16);
      R16 <= L15 XOR fR15_K16;
      L16 <= R15;
      ------------------------------------------------------------

      Inverse_Init_Perm : inv_init_permutation PORT MAP
(R16,L16,data_int);

      data<=data_int;
      cipher_out <= cipher_data;
      K_out <= K;

END decryption;
```

DES_decrypt_P.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;

ENTITY DES_decrypt_P IS
      PORT( clk: IN std_logic;
            cipher_data: IN std_logic_vector(63 DOWNTO 0);
            cipher_in: IN std_logic_vector(63 DOWNTO 0);
```

```
                 K: IN std_logic_vector(63 DOWNTO 0);

                 cipher_out: OUT std_logic_vector(63 DOWNTO 0);
                 K_out: OUT std_logic_vector(63 DOWNTO 0);
                 data: OUT std_logic_vector(63 DOWNTO 0));
END DES_decrypt_P;

ARCHITECTURE decryption OF DES_decrypt_P IS

      COMPONENT KS_D_P
      PORT( clk: IN std_logic;
                 KEY: IN std_logic_vector(63 DOWNTO 0);

      K1,K2,K3,K4,K5,K6,K7,K8,K9,K10,K11,K12,K13,K14,K15,K16: OUT
std_logic_vector(47 DOWNTO 0));
      END COMPONENT;

      COMPONENT init_permutation
      PORT(data: IN std_logic_vector(63 DOWNTO 0);
                 Ln: OUT std_logic_vector(31 DOWNTO 0);
                 Rn: OUT std_logic_vector(31 DOWNTO 0));
      END COMPONENT;

      COMPONENT feistel_func
      PORT(Rn: IN std_logic_vector(31 DOWNTO 0);
                 keyx: IN std_logic_vector(47 DOWNTO 0);
                 Rn_1: OUT std_logic_vector(31 DOWNTO 0));
      END COMPONENT;

      COMPONENT inv_init_permutation
      PORT(Ln: IN std_logic_vector(31 DOWNTO 0);
                 Rn: IN std_logic_vector(31 DOWNTO 0);
                 cipher_data: OUT std_logic_vector(63 DOWNTO 0));
      END COMPONENT;

      SHARED VARIABLE
K1,K2,K3,K4,K5,K6,K7,K8,K9,K10,K11,K12,K13,K14,K15,K16:
std_logic_vector(47 DOWNTO 0);
      SHARED VARIABLE
fR0_K1,fR1_K2,fR2_K3,fR3_K4,fR4_K5,fR5_K6,fR6_K7,fR7_K8,fR8_K9,fR9_K
10,fR10_K11,fR11_K12,fR12_K13,fR13_K14,fR14_K15,fR15_K16:
std_logic_vector(31 DOWNTO 0);

      SIGNAL
L0,L1,L2,L3,L4,L5,L6,L7,L8,L9,L10,L11,L12,L13,L14,L15,L16:
std_logic_vector(31 DOWNTO 0);
      SIGNAL
R0,R1,R2,R3,R4,R5,R6,R7,R8,R9,R10,R11,R12,R13,R14,R15,R16:
std_logic_vector(31 DOWNTO 0);
      SIGNAL data_int,data_buffer: std_logic_vector(63 DOWNTO 0);

      SIGNAL n: INTEGER := 0;
```

```
      SIGNAL
cipher_int,cipher_int1,cipher_int2,cipher_int3,cipher_int4,cipher_in
t5,cipher_int6,cipher_int7,

      cipher_int8,cipher_int9,cipher_int10,cipher_int11,cipher_int12
,cipher_int13,cipher_int14,cipher_int15: std_logic_vector(63 DOWNTO
0);

      SIGNAL K_int,K_int1,K_int2,K_int3,K_int4,K_int5,K_int6,K_int7,

      K_int8,K_int9,K_int10,K_int11,K_int12,K_int13,K_int14,K_int15:
std_logic_vector(63 DOWNTO 0);

BEGIN

      Key_schedule : KS_D_P PORT
MAP(clk,K,K1,K2,K3,K4,K5,K6,K7,K8,K9,K10,K11,K12,K13,K14,K15,K16);

      Inital_Perm : init_permutation PORT MAP(cipher_data,L0,R0);

      ---------------------------16 ROUNDS--------------------------
----------
      Round1 : feistel_func PORT MAP(R0,K1,fR0_K1);

      Round2 : feistel_func PORT MAP(R1,K2,fR1_K2);

      Round3 : feistel_func PORT MAP(R2,K3,fR2_K3);

      Round4 : feistel_func PORT MAP(R3,K4,fR3_K4);

      Round5 : feistel_func PORT MAP(R4,K5,fR4_K5);

      Round6 : feistel_func PORT MAP(R5,K6,fR5_K6);

      Round7 : feistel_func PORT MAP(R6,K7,fR6_K7);

      Round8 : feistel_func PORT MAP(R7,K8,fR7_K8);

      Round9 : feistel_func PORT MAP(R8,K9,fR8_K9);

      Round10 : feistel_func PORT MAP(R9,K10,fR9_K10);

      Round11 : feistel_func PORT MAP(R10,K11,fR10_K11);

      Round12 : feistel_func PORT MAP(R11,K12,fR11_K12);

      Round13 : feistel_func PORT MAP(R12,K13,fR12_K13);

      Round14 : feistel_func PORT MAP(R13,K14,fR13_K14);

      Round15 : feistel_func PORT MAP(R14,K15,fR14_K15);

      Round16 : feistel_func PORT MAP(R15,K16,fR15_K16);
      R16 <= L15 XOR fR15_K16;
      L16 <= R15;
```

```
        ------------------------------------------------------------

        Inverse_Init_Perm : inv_init_permutation PORT MAP
(R16,L16,data_int);

        Process(clk,cipher_data)
        BEGIN
                IF(falling_edge(clk)) THEN
                        FOR n IN 0 TO 15 LOOP
                                CASE n IS
                                        WHEN 0 => R1 <= L0 XOR fR0_K1;
                                                        L1 <= R0;
                                                        cipher_int1 <=
cipher_in;
                                                        K_int1 <= K;
                                        WHEN 1 => R2 <= L1 XOR fR1_K2;
                                                        L2 <= R1;
                                                        cipher_int2 <=
cipher_int1;
                                                        K_int2 <= K_int1;
                                        WHEN 2 => R3 <= L2 XOR fR2_K3;
                                                        L3 <= R2;
                                                        cipher_int3 <=
cipher_int2;
                                                        K_int3 <= K_int2;
                                        WHEN 3 => R4 <= L3 XOR fR3_K4;
                                                        L4 <= R3;
                                                        cipher_int4 <=
cipher_int3;
                                                        K_int4 <= K_int3;
                                        WHEN 4 => R5 <= L4 XOR fR4_K5;
                                                        L5 <= R4;
                                                        cipher_int5 <=
cipher_int4;
                                                        K_int5 <= K_int4;
                                        WHEN 5 => R6 <= L5 XOR fR5_K6;
                                                        L6 <= R5;
                                                        cipher_int6 <=
cipher_int5;
                                                        K_int6 <= K_int5;
                                        WHEN 6 => R7 <= L6 XOR fR6_K7;
                                                        L7 <= R6;
                                                        cipher_int7 <=
cipher_int6;
                                                        K_int7 <= K_int6;
                                        WHEN 7 => R8 <= L7 XOR fR7_K8;
                                                        L8 <= R7;
                                                        cipher_int8 <=
cipher_int7;
                                                        K_int8 <= K_int7;
                                        WHEN 8 => R9 <= L8 XOR fR8_K9;
                                                        L9 <= R8;
                                                        cipher_int9 <=
cipher_int8;
                                                        K_int9 <= K_int8;
```

```
                              WHEN 9 => R10 <= L9 XOR fR9_K10;
                                                L10 <= R9;
                                                cipher_int10 <=
cipher_int9;
                                                K_int10 <= K_int9;
                              WHEN 10 => R11    <= L10 XOR fR10_K11;
                                                L11 <= R10;
                                                cipher_int11 <=
cipher_int10;
                                                K_int11 <= K_int10;
                              WHEN 11 => R12 <= L11 XOR fR11_K12;
                                                L12 <= R11;
                                                cipher_int12 <=
cipher_int11;
                                                K_int12 <= K_int11;
                              WHEN 12 => R13 <= L12 XOR fR12_K13;
                                                L13 <= R12;
                                                cipher_int13 <=
cipher_int12;
                                                K_int13 <= K_int12;
                              WHEN 13 => R14 <= L13 XOR fR13_K14;
                                                L14 <= R13;
                                                cipher_int14 <=
cipher_int13;
                                                K_int14 <= K_int13;
                              WHEN 14 => R15 <= L14 XOR fR14_K15;
                                                L15 <= R14;
                                                cipher_int15 <=
cipher_int14;
                                                K_int15 <= K_int14;
                              WHEN 15 => data_buffer <= data_int;
                                                cipher_int <=
cipher_int15;
                                                K_int <= K_int15;
                        END CASE;
                  END LOOP;
            END IF;
      END PROCESS;

      data<=data_buffer;
      cipher_out <= cipher_int;
      K_out <= K_int;

END decryption;
```

DES_encrypt.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;

ENTITY DES_encrypt IS
      PORT(data: IN std_logic_vector(63 DOWNTO 0);
                  K: IN std_logic_vector(63 DOWNTO 0);
```

```
                  data_out: OUT std_logic_vector(63 DOWNTO 0);
                  K_out: OUT std_logic_vector(63 DOWNTO 0);
                  cipher_data: OUT std_logic_vector(63 DOWNTO 0));
END DES_encrypt;

ARCHITECTURE encryption OF DES_encrypt IS

      COMPONENT KS_E
      PORT(KEY: IN std_logic_vector(63 DOWNTO 0);

      K1,K2,K3,K4,K5,K6,K7,K8,K9,K10,K11,K12,K13,K14,K15,K16: OUT
std_logic_vector(47 DOWNTO 0));
      END COMPONENT;

      COMPONENT init_permutation
      PORT(data: IN std_logic_vector(63 DOWNTO 0);
                  Ln: OUT std_logic_vector(31 DOWNTO 0);
                  Rn: OUT std_logic_vector(31 DOWNTO 0));
      END COMPONENT;

      COMPONENT feistel_func
      PORT(Rn: IN std_logic_vector(31 DOWNTO 0);
                  keyx: IN std_logic_vector(47 DOWNTO 0);
                  Rn_1: OUT std_logic_vector(31 DOWNTO 0));
      END COMPONENT;

      COMPONENT inv_init_permutation
      PORT(Ln: IN std_logic_vector(31 DOWNTO 0);
                  Rn: IN std_logic_vector(31 DOWNTO 0);
                  cipher_data: OUT std_logic_vector(63 DOWNTO 0));
      END COMPONENT;

      SHARED VARIABLE
K1,K2,K3,K4,K5,K6,K7,K8,K9,K10,K11,K12,K13,K14,K15,K16:
std_logic_vector(47 DOWNTO 0);
      SHARED VARIABLE
fR0_K1,fR1_K2,fR2_K3,fR3_K4,fR4_K5,fR5_K6,fR6_K7,fR7_K8,fR8_K9,fR9_K
10,fR10_K11,fR11_K12,fR12_K13,fR13_K14,fR14_K15,fR15_K16:
std_logic_vector(31 DOWNTO 0);

      SIGNAL
L0,L1,L2,L3,L4,L5,L6,L7,L8,L9,L10,L11,L12,L13,L14,L15,L16:
std_logic_vector(31 DOWNTO 0);
      SIGNAL
R0,R1,R2,R3,R4,R5,R6,R7,R8,R9,R10,R11,R12,R13,R14,R15,R16:
std_logic_vector(31 DOWNTO 0);
      SIGNAL cipher_int: std_logic_vector(63 DOWNTO 0);

      SIGNAL n: INTEGER := 0;

      SIGNAL
data_int,data_int1,data_int2,data_int3,data_int4,data_int5,data_int6
,data_int7,
```

```
      data_int8,data_int9,data_int10,data_int11,data_int12,data_int1
3,data_int14,data_int15: std_logic_vector(63 DOWNTO 0);

      SIGNAL K_int,K_int1,K_int2,K_int3,K_int4,K_int5,K_int6,K_int7,

      K_int8,K_int9,K_int10,K_int11,K_int12,K_int13,K_int14,K_int15:
std_logic_vector(63 DOWNTO 0);

BEGIN

      Key_schedule : KS_E PORT
MAP(K,K1,K2,K3,K4,K5,K6,K7,K8,K9,K10,K11,K12,K13,K14,K15,K16);

      Inital_Perm : init_permutation PORT MAP(data,L0,R0);

      ---------------------------16 ROUNDS--------------------------
----------
      Round1 : feistel_func PORT MAP(R0,K1,fR0_K1);
      R1 <= L0 XOR fR0_K1;
      L1 <= R0;

      Round2 : feistel_func PORT MAP(R1,K2,fR1_K2);
      R2 <= L1 XOR fR1_K2;
      L2 <= R1;

      Round3 : feistel_func PORT MAP(R2,K3,fR2_K3);
      R3 <= L2 XOR fR2_K3;
      L3 <= R2;

      Round4 : feistel_func PORT MAP(R3,K4,fR3_K4);
      R4 <= L3 XOR fR3_K4;
      L4 <= R3;

      Round5 : feistel_func PORT MAP(R4,K5,fR4_K5);
      R5 <= L4 XOR fR4_K5;
      L5 <= R4;

      Round6 : feistel_func PORT MAP(R5,K6,fR5_K6);
      R6 <= L5 XOR fR5_K6;
      L6 <= R5;

      Round7 : feistel_func PORT MAP(R6,K7,fR6_K7);
      R7 <= L6 XOR fR6_K7;
      L7 <= R6;

      Round8 : feistel_func PORT MAP(R7,K8,fR7_K8);
      R8 <= L7 XOR fR7_K8;
      L8 <= R7;

      Round9 : feistel_func PORT MAP(R8,K9,fR8_K9);
      R9 <= L8 XOR fR8_K9;
      L9 <= R8;

      Round10 : feistel_func PORT MAP(R9,K10,fR9_K10);
```

```
      R10 <= L9 XOR fR9_K10;
      L10 <= R9;

      Round11 : feistel_func PORT MAP(R10,K11,fR10_K11);
      R11   <= L10 XOR fR10_K11;
      L11 <= R10;

      Round12 : feistel_func PORT MAP(R11,K12,fR11_K12);
      R12 <= L11 XOR fR11_K12;
      L12 <= R11;

      Round13 : feistel_func PORT MAP(R12,K13,fR12_K13);
      R13 <= L12 XOR fR12_K13;
      L13 <= R12;

      Round14 : feistel_func PORT MAP(R13,K14,fR13_K14);
      R14 <= L13 XOR fR13_K14;
      L14 <= R13;

      Round15 : feistel_func PORT MAP(R14,K15,fR14_K15);
      R15 <= L14 XOR fR14_K15;
      L15 <= R14;

      Round16 : feistel_func PORT MAP(R15,K16,fR15_K16);
      R16 <= L15 XOR fR15_K16;
      L16 <= R15;

      ------------------------------------------------------------

      Inverse_Init_Perm : inv_init_permutation PORT MAP
(R16,L16,cipher_int);

      cipher_data<=cipher_int;
      data_out <= data;
      K_out <= K;

END encryption;
```

DES_encrypt_P.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;

ENTITY DES_encrypt_P IS
      PORT( clk: IN std_logic;
                data: IN std_logic_vector(63 DOWNTO 0);
                data_in: IN std_logic_vector(63 DOWNTO 0);
                K: IN std_logic_vector(63 DOWNTO 0);

                data_out: OUT std_logic_vector(63 DOWNTO 0);
                K_out: OUT std_logic_vector(63 DOWNTO 0);
                cipher_data: OUT std_logic_vector(63 DOWNTO 0));
END DES_encrypt_P;
```

```
ARCHITECTURE encryption OF DES_encrypt_P IS

      COMPONENT KS_E_P
      PORT( clk: IN std_logic;
                 KEY: IN std_logic_vector(63 DOWNTO 0);

      K1,K2,K3,K4,K5,K6,K7,K8,K9,K10,K11,K12,K13,K14,K15,K16: OUT
std_logic_vector(47 DOWNTO 0));
      END COMPONENT;

      COMPONENT init_permutation
      PORT(data: IN std_logic_vector(63 DOWNTO 0);
                 Ln: OUT std_logic_vector(31 DOWNTO 0);
                 Rn: OUT std_logic_vector(31 DOWNTO 0));
      END COMPONENT;

      COMPONENT feistel_func
      PORT(Rn: IN std_logic_vector(31 DOWNTO 0);
                 keyx: IN std_logic_vector(47 DOWNTO 0);
                 Rn_1: OUT std_logic_vector(31 DOWNTO 0));
      END COMPONENT;

      COMPONENT inv_init_permutation
      PORT(Ln: IN std_logic_vector(31 DOWNTO 0);
                 Rn: IN std_logic_vector(31 DOWNTO 0);
                 cipher_data: OUT std_logic_vector(63 DOWNTO 0));
      END COMPONENT;

      SHARED VARIABLE
K1,K2,K3,K4,K5,K6,K7,K8,K9,K10,K11,K12,K13,K14,K15,K16:
std_logic_vector(47 DOWNTO 0);
      SHARED VARIABLE
fR0_K1,fR1_K2,fR2_K3,fR3_K4,fR4_K5,fR5_K6,fR6_K7,fR7_K8,fR8_K9,fR9_K
10,fR10_K11,fR11_K12,fR12_K13,fR13_K14,fR14_K15,fR15_K16:
std_logic_vector(31 DOWNTO 0);

      SIGNAL
L0,L1,L2,L3,L4,L5,L6,L7,L8,L9,L10,L11,L12,L13,L14,L15,L16:
std_logic_vector(31 DOWNTO 0);
      SIGNAL
R0,R1,R2,R3,R4,R5,R6,R7,R8,R9,R10,R11,R12,R13,R14,R15,R16:
std_logic_vector(31 DOWNTO 0);
      SIGNAL cipher_int,cipher_buffer: std_logic_vector(63 DOWNTO
0);

      SIGNAL n: INTEGER := 0;

      SIGNAL
data_int,data_int1,data_int2,data_int3,data_int4,data_int5,data_int6
,data_int7,

      data_int8,data_int9,data_int10,data_int11,data_int12,data_int1
3,data_int14,data_int15: std_logic_vector(63 DOWNTO 0);

      SIGNAL K_int,K_int1,K_int2,K_int3,K_int4,K_int5,K_int6,K_int7,
```

```
      K_int8,K_int9,K_int10,K_int11,K_int12,K_int13,K_int14,K_int15:
std_logic_vector(63 DOWNTO 0);

BEGIN

      Key_schedule : KS_E_P PORT
MAP(clk,K,K1,K2,K3,K4,K5,K6,K7,K8,K9,K10,K11,K12,K13,K14,K15,K16);

      Inital_Perm : init_permutation PORT MAP(data,L0,R0);

      ---------------------------16 ROUNDS--------------------------
----------
      Round1 : feistel_func PORT MAP(R0,K1,fR0_K1);

      Round2 : feistel_func PORT MAP(R1,K2,fR1_K2);

      Round3 : feistel_func PORT MAP(R2,K3,fR2_K3);

      Round4 : feistel_func PORT MAP(R3,K4,fR3_K4);

      Round5 : feistel_func PORT MAP(R4,K5,fR4_K5);

      Round6 : feistel_func PORT MAP(R5,K6,fR5_K6);

      Round7 : feistel_func PORT MAP(R6,K7,fR6_K7);

      Round8 : feistel_func PORT MAP(R7,K8,fR7_K8);

      Round9 : feistel_func PORT MAP(R8,K9,fR8_K9);

      Round10 : feistel_func PORT MAP(R9,K10,fR9_K10);

      Round11 : feistel_func PORT MAP(R10,K11,fR10_K11);

      Round12 : feistel_func PORT MAP(R11,K12,fR11_K12);

      Round13 : feistel_func PORT MAP(R12,K13,fR12_K13);

      Round14 : feistel_func PORT MAP(R13,K14,fR13_K14);

      Round15 : feistel_func PORT MAP(R14,K15,fR14_K15);

      Round16 : feistel_func PORT MAP(R15,K16,fR15_K16);
      R16 <= L15 XOR fR15_K16;
      L16 <= R15;

      ------------------------------------------------------------

      Inverse_Init_Perm : inv_init_permutation PORT MAP
(R16,L16,cipher_int);

      Process(clk,data)
      BEGIN
            IF(falling_edge(clk)) THEN
```

```
                    FOR n IN 0 TO 15 LOOP
                          CASE n IS
                                WHEN 0 => R1 <= L0 XOR fR0_K1;
                                                  L1 <= R0;
                                                  data_int1 <= data_in;
                                                  K_int1 <= K;
                                WHEN 1 => R2 <= L1 XOR fR1_K2;
                                                  L2 <= R1;
                                                  data_int2 <=
data_int1;
                                                  K_int2 <= K_int1;
                                WHEN 2 => R3 <= L2 XOR fR2_K3;
                                                  L3 <= R2;
                                                  data_int3 <=
data_int2;
                                                  K_int3 <= K_int2;
                                WHEN 3 => R4 <= L3 XOR fR3_K4;
                                                  L4 <= R3;
                                                  data_int4 <=
data_int3;
                                                  K_int4 <= K_int3;
                                WHEN 4 => R5 <= L4 XOR fR4_K5;
                                                  L5 <= R4;
                                                  data_int5 <=
data_int4;
                                                  K_int5 <= K_int4;
                                WHEN 5 => R6 <= L5 XOR fR5_K6;
                                                  L6 <= R5;
                                                  data_int6 <=
data_int5;
                                                  K_int6 <= K_int5;
                                WHEN 6 => R7 <= L6 XOR fR6_K7;
                                                  L7 <= R6;
                                                  data_int7 <=
data_int6;
                                                  K_int7 <= K_int6;
                                WHEN 7 => R8 <= L7 XOR fR7_K8;
                                                  L8 <= R7;
                                                  data_int8 <=
data_int7;
                                                  K_int8 <= K_int7;
                                WHEN 8 => R9 <= L8 XOR fR8_K9;
                                                  L9 <= R8;
                                                  data_int9 <=
data_int8;
                                                  K_int9 <= K_int8;
                                WHEN 9 => R10 <= L9 XOR fR9_K10;
                                                  L10 <= R9;
                                                  data_int10 <=
data_int9;
                                                  K_int10 <= K_int9;
                                WHEN 10 => R11    <= L10 XOR fR10_K11;
                                                  L11 <= R10;
                                                  data_int11 <=
data_int10;
```

```
                                        K_int11 <= K_int10;
                        WHEN 11 => R12 <= L11 XOR fR11_K12;
                                        L12 <= R11;
                                        data_int12 <=
data_int11;
                                        K_int12 <= K_int11;
                        WHEN 12 => R13 <= L12 XOR fR12_K13;
                                        L13 <= R12;
                                        data_int13 <=
data_int12;
                                        K_int13 <= K_int12;
                        WHEN 13 => R14 <= L13 XOR fR13_K14;
                                        L14 <= R13;
                                        data_int14 <=
data_int13;
                                        K_int14 <= K_int13;
                        WHEN 14 => R15 <= L14 XOR fR14_K15;
                                        L15 <= R14;
                                        data_int15 <=
data_int14;
                                        K_int15 <= K_int14;
                        WHEN 15 => cipher_buffer <= cipher_int;
                                        data_int <=
data_int15;
                                        K_int <= K_int15;
                    END CASE;
                END LOOP;
            END IF;
        END PROCESS;

        cipher_data<=cipher_buffer;
        data_out <= data_int;
        K_out <= K_int;

END encryption;
```

expan_perm.vhd

```
--EDNI DEL ROSAL----expan_perm.vhd---------------
--Thesis Spring 2015----------------------------
--This file performs the expansion permutation of
--Rn, E(Rn), in the Feistel function.  It also---
--performs the XOR between the expanded Rn, E(Rn),
--and the sub key kn.---------------------------
--The inputs are Rn and kn and the output is-----
--Rn_expan, E(Rn)---------------------
library IEEE;
use IEEE.std_logic_1164.all;

ENTITY expan_perm IS
      PORT(Rn: IN std_logic_vector(31 DOWNTO 0);
            Rn_expan: OUT std_logic_vector(47 DOWNTO 0));
END expan_perm;
```

```
ARCHITECTURE expansion OF expan_perm IS

BEGIN

      Rn_expan <= Rn(0) & Rn(31) & Rn(30) & Rn(29) & Rn(28) & Rn(27)
&
                        Rn(28) & Rn(27) & Rn(26) & Rn(25) & Rn(24) &
Rn(23) &
                        Rn(24) & Rn(23) & Rn(22) & Rn(21) & Rn(20) &
Rn(19) &
                        Rn(20) & Rn(19) & Rn(18) & Rn(17) & Rn(16) &
Rn(15) &
                        Rn(16) & Rn(15) & Rn(14) & Rn(13) & Rn(12) &
Rn(11) &
                        Rn(12) & Rn(11) & Rn(10) & Rn(9) & Rn(8) &
Rn(7) &
                        Rn(8) & Rn(7) & Rn(6) & Rn(5) & Rn(4) & Rn(3)
&
                        Rn(4) & Rn(3) & Rn(2) & Rn(1) & Rn(0) &
Rn(31);

END expansion;
```

Feistel_func.vhd

```
--EDNI DEL ROSAL----feistel_func.vhd-------------
--Thesis Spring 2015----------------------------
--This file performs the Feistel function and----
--updates the right side Rn.---------------------
--The inputs are Ln, Rn, keyx and the output is--
--Rn_1-------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;

ENTITY feistel_func IS
      PORT(Rn: IN std_logic_vector(31 DOWNTO 0);
            keyx: IN std_logic_vector(47 DOWNTO 0);
            Rn_1: OUT std_logic_vector(31 DOWNTO 0));
END feistel_func;

ARCHITECTURE func OF feistel_func IS

      COMPONENT expan_perm
      PORT(Rn: IN std_logic_vector(31 DOWNTO 0);
            Rn_expan: OUT std_logic_vector(47 DOWNTO 0));
      END COMPONENT;

      COMPONENT S1
      PORT(B1: IN std_logic_vector(5 DOWNTO 0);
            S_B1: OUT std_logic_vector(3 DOWNTO 0));
      END COMPONENT;

      COMPONENT S2
      PORT(B2: IN std_logic_vector(5 DOWNTO 0);
```

```
            S_B2: OUT std_logic_vector(3 DOWNTO 0));
      END COMPONENT;

      COMPONENT S3
      PORT(B3: IN std_logic_vector(5 DOWNTO 0);
            S_B3: OUT std_logic_vector(3 DOWNTO 0));
      END COMPONENT;

      COMPONENT S4
      PORT(B4: IN std_logic_vector(5 DOWNTO 0);
            S_B4: OUT std_logic_vector(3 DOWNTO 0));
      END COMPONENT;

      COMPONENT S5
      PORT(B5: IN std_logic_vector(5 DOWNTO 0);
            S_B5: OUT std_logic_vector(3 DOWNTO 0));
      END COMPONENT;

      COMPONENT S6
      PORT(B6: IN std_logic_vector(5 DOWNTO 0);
            S_B6: OUT std_logic_vector(3 DOWNTO 0));
      END COMPONENT;

      COMPONENT S7
      PORT(B7: IN std_logic_vector(5 DOWNTO 0);
            S_B7: OUT std_logic_vector(3 DOWNTO 0));
      END COMPONENT;

      COMPONENT S8
      PORT(B8: IN std_logic_vector(5 DOWNTO 0);
            S_B8: OUT std_logic_vector(3 DOWNTO 0));
      END COMPONENT;

      COMPONENT permutation
      PORT(S_concat: IN std_logic_vector(31 DOWNTO 0);
            Rn_1: OUT std_logic_vector(31 DOWNTO 0));
      END COMPONENT;

      SHARED VARIABLE E_Rn: std_logic_vector(47 DOWNTO 0);
      SIGNAL E_xor_keyx: std_logic_vector(47 DOWNTO 0);
      SHARED VARIABLE S_out: std_logic_vector(31 DOWNTO 0);


BEGIN

      expansionPerm: expan_perm PORT MAP(Rn,E_Rn);

      E_xor_keyx <= E_Rn XOR keyx;

      BOX1: S1 PORT MAP(E_xor_keyx(47 DOWNTO 42),S_out(31 DOWNTO
28));
      BOX2: S2 PORT MAP(E_xor_keyx(41 DOWNTO 36),S_out(27 DOWNTO
24));
      BOX3: S3 PORT MAP(E_xor_keyx(35 DOWNTO 30),S_out(23 DOWNTO
20));
```

```
      BOX4: S4 PORT MAP(E_xor_keyx(29 DOWNTO 24),S_out(19 DOWNTO
16));
      BOX5: S5 PORT MAP(E_xor_keyx(23 DOWNTO 18),S_out(15 DOWNTO
12));
      BOX6: S6 PORT MAP(E_xor_keyx(17 DOWNTO 12),S_out(11 DOWNTO
8));
      BOX7: S7 PORT MAP(E_xor_keyx(11 DOWNTO 6),S_out(7 DOWNTO 4));
      BOX8: S8 PORT MAP(E_xor_keyx(5 DOWNTO 0),S_out(3 DOWNTO 0));

      Perm: permutation PORT MAP(S_out,Rn_1);


END func;
```

init_permutation.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;

ENTITY init_permutation IS
      PORT(data: IN std_logic_vector(63 DOWNTO 0);
                  Ln: OUT std_logic_vector(31 DOWNTO 0);
                  Rn: OUT std_logic_vector(31 DOWNTO 0));
END init_permutation;

ARCHITECTURE permutation OF init_permutation IS
BEGIN

      Ln <= data(6) & data(14) & data(22) & data(30) & data(38) &
data(46) & data(54) & data(62) &
            data(4) & data(12) & data(20) & data(28) & data(36) &
data(44) & data(52) & data(60) &
            data(2) & data(10) & data(18) & data(26) & data(34) &
data(42) & data(50) & data(58) &
            data(0) & data(8) & data(16) & data(24) & data(32) &
data(40) & data(48) & data(56);

      Rn <= data(7) & data(15) & data(23) & data(31) & data(39) &
data(47) & data(55) & data(63) &
            data(5) & data(13) & data(21) & data(29) & data(37) &
data(45) & data(53) & data(61) &
            data(3) & data(11) & data(19) & data(27) & data(35) &
data(43) & data(51) & data(59) &
            data(1) & data(9) & data(17) & data(25) & data(33) &
data(41) & data(49) & data(57);

END permutation;
```

inv_init_permutation.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
```

```
ENTITY inv_init_permutation IS
      PORT(Ln: IN std_logic_vector(31 DOWNTO 0);
                  Rn: IN std_logic_vector(31 DOWNTO 0);
                  cipher_data: OUT std_logic_vector(63 DOWNTO 0));
END inv_init_permutation;

ARCHITECTURE permutation OF inv_init_permutation IS
      SHARED VARIABLE Ln_Rn: std_logic_vector(63 DOWNTO 0) := Ln &
Rn;
BEGIN

       cipher_data <= Ln_Rn(24) & Ln_Rn(56) & Ln_Rn(16) & Ln_Rn(48)
& Ln_Rn(8) & Ln_Rn(40) & Ln_Rn(0) & Ln_Rn(32) &
           Ln_Rn(25) & Ln_Rn(57) & Ln_Rn(17) & Ln_Rn(49) & Ln_Rn(9)
& Ln_Rn(41) & Ln_Rn(1) & Ln_Rn(33) &
           Ln_Rn(26) & Ln_Rn(58) & Ln_Rn(18) & Ln_Rn(50) & Ln_Rn(10)
& Ln_Rn(42) & Ln_Rn(2) & Ln_Rn(34) &
           Ln_Rn(27) & Ln_Rn(59) & Ln_Rn(19) & Ln_Rn(51) & Ln_Rn(11)
& Ln_Rn(43) & Ln_Rn(3) & Ln_Rn(35) &

           Ln_Rn(28) & Ln_Rn(60) & Ln_Rn(20) & Ln_Rn(52) & Ln_Rn(12)
& Ln_Rn(44) & Ln_Rn(4) & Ln_Rn(36) &
           Ln_Rn(29) & Ln_Rn(61) & Ln_Rn(21) & Ln_Rn(53) & Ln_Rn(13)
& Ln_Rn(45) & Ln_Rn(5) & Ln_Rn(37) &
           Ln_Rn(30) & Ln_Rn(62) & Ln_Rn(22) & Ln_Rn(54) & Ln_Rn(14)
& Ln_Rn(46) & Ln_Rn(6) & Ln_Rn(38) &
           Ln_Rn(31) & Ln_Rn(63) & Ln_Rn(23) & Ln_Rn(55) & Ln_Rn(15)
& Ln_Rn(47) & Ln_Rn(7) & Ln_Rn(39);

END permutation;
```

key_bank.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;

ENTITY key_bank IS
      PORT( clk: IN std_logic;
                  key1_in,key2_in,key3_in: IN std_logic_vector(63
DOWNTO 0);
                  key1_out,key2_out,key3_out: OUT std_logic_vector(63
DOWNTO 0));
END key_bank;

ARCHITECTURE bank OF key_bank IS

      SIGNAL
key2_1,key2_2,key2_3,key2_4,key2_5,key2_6,key2_7,key2_8,

      key2_9,key2_10,key2_11,key2_12,key2_13,key2_14,key2_15,key2_16
:
                        std_logic_vector(63 DOWNTO 0);
```

```
	SIGNAL
key3_1,key3_2,key3_3,key3_4,key3_5,key3_6,key3_7,key3_8,

	key3_9,key3_10,key3_11,key3_12,key3_13,key3_14,key3_15,key3_16
,

	key3_17,key3_18,key3_19,key3_20,key3_21,key3_22,key3_23,key3_2
4,

	key3_25,key3_26,key3_27,key3_28,key3_29,key3_30,key3_31,key3_3
2:
			std_logic_vector(63 DOWNTO 0);

BEGIN

	key1_out <= key1_in;
	key2_out <= key2_15;
	key3_out <= key3_31;

	PROCESS(clk)
	BEGIN
		IF(falling_edge(clk)) THEN
			FOR n IN 0 TO 31 LOOP
				CASE n IS
					WHEN 0 => key2_1 <= key2_in;
							key3_1 <=
key3_in;
					WHEN 1 => key2_2 <= key2_1;
							key3_2 <= key3_1;
					WHEN 2 => key2_3 <= key2_2;
							key3_3 <= key3_2;
					WHEN 3 => key2_4 <= key2_3;
							key3_4 <= key3_3;
					WHEN 4 => key2_5 <= key2_4;
							key3_5 <= key3_4;
					WHEN 5 => key2_6 <= key2_5;
							key3_6 <= key3_5;
					WHEN 6 => key2_7 <= key2_6;
							key3_7 <= key3_6;
					WHEN 7 => key2_8 <= key2_7;
							key3_8 <= key3_7;
					WHEN 8 => key2_9 <= key2_8;
							key3_9 <= key3_8;
					WHEN 9 => key2_10 <= key2_9;
							key3_10 <=
key3_9;
					WHEN 10 => key2_11 <= key2_10;
							key3_11 <=
key3_10;
					WHEN 11 => key2_12 <= key2_11;
							key3_12 <=
key3_11;
					WHEN 12 => key2_13 <= key2_12;
							key3_13 <=
key3_12;
```

```
                              WHEN 13 => key2_14 <= key2_13;
                                              key3_14 <=
key3_13;
                              WHEN 14 => key2_15 <= key2_14;
                                              key3_15 <=
key3_14;
                              WHEN 15 => key2_16 <= key2_15;
                                              key3_16 <=
key3_15;

                              WHEN 16 => key3_17 <= key3_16;
                              WHEN 17 => key3_18 <= key3_17;
                              WHEN 18 => key3_19 <= key3_18;
                              WHEN 19 => key3_20 <= key3_19;
                              WHEN 20 => key3_21 <= key3_20;
                              WHEN 21 => key3_22 <= key3_21;
                              WHEN 22 => key3_23 <= key3_22;
                              WHEN 23 => key3_24 <= key3_23;
                              WHEN 24 => key3_25 <= key3_24;
                              WHEN 25 => key3_26 <= key3_25;
                              WHEN 26 => key3_27 <= key3_26;
                              WHEN 27 => key3_28 <= key3_27;
                              WHEN 28 => key3_29 <= key3_28;
                              WHEN 29 => key3_30 <= key3_29;
                              WHEN 30 => key3_31 <= key3_30;
                              WHEN 31 => key3_32 <= key3_31;
                        END CASE;
                  END LOOP;
            END IF;
      END PROCESS;

END bank;
```

KS_D.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;

ENTITY KS_D IS
      PORT( KEY: IN std_logic_vector(63 DOWNTO 0);

      K1,K2,K3,K4,K5,K6,K7,K8,K9,K10,K11,K12,K13,K14,K15,K16: OUT
std_logic_vector(47 DOWNTO 0));
END KS_D;

ARCHITECTURE keys OF KS_D IS

      COMPONENT PC_1
      PORT( key: IN std_logic_vector(63 DOWNTO 0);
                  c0: OUT std_logic_vector(27 DOWNTO 0);
                  d0: OUT std_logic_vector(27 DOWNTO 0));
      END COMPONENT;

      COMPONENT SRR_1
```

```
            PORT( C,D: IN std_logic_vector(27 DOWNTO 0);
                        CS,DS: OUT std_logic_vector(27 DOWNTO 0));
      END COMPONENT;

      COMPONENT SRR_2
            PORT( C,D: IN std_logic_vector(27 DOWNTO 0);
                        CS,DS: OUT std_logic_vector(27 DOWNTO 0));
      END COMPONENT;

      COMPONENT PC_2
      PORT(CS,DS: IN std_logic_vector(27 DOWNTO 0);
            K: OUT std_logic_vector(47 DOWNTO 0));
      END COMPONENT;

      SHARED VARIABLE
C0,C1,C2,C3,C4,C5,C6,C7,C8,C9,C10,C11,C12,C13,C14,C15,C16:
std_logic_vector(27 DOWNTO 0);
      SHARED VARIABLE
D0,D1,D2,D3,D4,D5,D6,D7,D8,D9,D10,D11,D12,D13,D14,D15,D16:
std_logic_vector(27 DOWNTO 0);

      SIGNAL
Reg1,Reg2,Reg3,Reg4,Reg5,Reg6,Reg7,Reg8,Reg9,Reg10,Reg11,Reg12,Reg13
,Reg14,Reg15: std_logic_vector(55 DOWNTO 0);
      SIGNAL cipher_int: std_logic_vector(63 DOWNTO 0);

BEGIN

      PermChoice_1: PC_1 PORT MAP(KEY,C0,D0);
      PermChoice1_2: PC_2 PORT MAP(C0,D0,K1);
      Reg1 <= C0 & D0;

      ShiftRotate1_1 : SRR_1 PORT MAP(Reg1(55 DOWNTO 28),Reg1(27
DOWNTO 0),C1,D1);
      PermChoice2_2: PC_2 PORT MAP(C1,D1,K2);
      Reg2 <= C1 & D1;

      ShiftRotate2_2 : SRR_2 PORT MAP(Reg2(55 DOWNTO 28),Reg2(27
DOWNTO 0),C2,D2);
      PermChoice3_2: PC_2 PORT MAP(C2,D2,K3);
      Reg3 <= C2 & D2;

      ShiftRotate3_2 : SRR_2 PORT MAP(Reg3(55 DOWNTO 28),Reg3(27
DOWNTO 0),C3,D3);
      PermChoice4_2: PC_2 PORT MAP(C3,D3,K4);
      Reg4 <= C3 & D3;

      ShiftRotate4_2 : SRR_2 PORT MAP(Reg4(55 DOWNTO 28),Reg4(27
DOWNTO 0),C4,D4);
      PermChoice5_2: PC_2 PORT MAP(C4,D4,K5);
      Reg5 <= C4 & D4;

      ShiftRotate5_2 : SRR_2 PORT MAP(Reg5(55 DOWNTO 28),Reg5(27
DOWNTO 0),C5,D5);
      PermChoice6_2: PC_2 PORT MAP(C5,D5,K6);
```

```
      Reg6 <= C5 & D5;

      ShiftRotate6_2 : SRR_2 PORT MAP(Reg6(55 DOWNTO 28),Reg6(27
DOWNTO 0),C6,D6);
      PermChoice7_2: PC_2 PORT MAP(C6,D6,K7);
      Reg7 <= C6 & D6;

      ShiftRotate7_2 : SRR_2 PORT MAP(Reg7(55 DOWNTO 28),Reg7(27
DOWNTO 0),C7,D7);
      PermChoice8_2: PC_2 PORT MAP(C7,D7,K8);
      Reg8 <= C7 & D7;

      ShiftRotate8_1 : SRR_1 PORT MAP(Reg8(55 DOWNTO 28),Reg8(27
DOWNTO 0),C8,D8);
      PermChoice9_2: PC_2 PORT MAP(C8,D8,K9);
      Reg9 <= C8 & D8;

      ShiftRotate9_2 : SRR_2 PORT MAP(Reg9(55 DOWNTO 28),Reg9(27
DOWNTO 0),C9,D9);
      PermChoice10_2: PC_2 PORT MAP(C9,D9,K10);
      Reg10 <= C9 & D9;

      ShiftRotate10_2 : SRR_2 PORT MAP(Reg10(55 DOWNTO 28),Reg10(27
DOWNTO 0),C10,D10);
      PermChoice11_2: PC_2 PORT MAP(C10,D10,K11);
      Reg11 <= C10 & D10;

      ShiftRotate11_2 : SRR_2 PORT MAP(Reg11(55 DOWNTO 28),Reg11(27
DOWNTO 0),C11,D11);
      PermChoice12_2: PC_2 PORT MAP(C11,D11,K12);
      Reg12 <= C11 & D11;

      ShiftRotate12_2 : SRR_2 PORT MAP(Reg12(55 DOWNTO 28),Reg12(27
DOWNTO 0),C12,D12);
      PermChoice13_2: PC_2 PORT MAP(C12,D12,K13);
      Reg13 <= C12 & D12;

      ShiftRotate13_2 : SRR_2 PORT MAP(Reg13(55 DOWNTO 28),Reg13(27
DOWNTO 0),C13,D13);
      PermChoice14_2: PC_2 PORT MAP(C13,D13,K14);
      Reg14 <= C13     & D13;

      ShiftRotate14_2 : SRR_2 PORT MAP(Reg14(55 DOWNTO 28),Reg14(27
DOWNTO 0),C14,D14);
      PermChoice15_2: PC_2 PORT MAP(C14,D14,K15);
      Reg15 <= C14 & D14;

      ShiftRotate15_1 : SRR_1 PORT MAP(Reg15(55 DOWNTO 28),Reg15(27
DOWNTO 0),C15,D15);
      PermChoice16_2: PC_2 PORT MAP(C15,D15,K16);

END keys;
```

KS_D_P.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;

ENTITY KS_D_P IS
      PORT( clk: IN std_logic;
                  KEY: IN std_logic_vector(63 DOWNTO 0);

      K1,K2,K3,K4,K5,K6,K7,K8,K9,K10,K11,K12,K13,K14,K15,K16: OUT
std_logic_vector(47 DOWNTO 0));
END KS_D_P;

ARCHITECTURE keys OF KS_D_P IS

      COMPONENT PC_1
      PORT( key: IN std_logic_vector(63 DOWNTO 0);
                  c0: OUT std_logic_vector(27 DOWNTO 0);
                  d0: OUT std_logic_vector(27 DOWNTO 0));
      END COMPONENT;

      COMPONENT SRR_1
            PORT( C,D: IN std_logic_vector(27 DOWNTO 0);
                        CS,DS: OUT std_logic_vector(27 DOWNTO 0));
      END COMPONENT;

      COMPONENT SRR_2
            PORT( C,D: IN std_logic_vector(27 DOWNTO 0);
                        CS,DS: OUT std_logic_vector(27 DOWNTO 0));
      END COMPONENT;

      COMPONENT PC_2
      PORT(CS,DS: IN std_logic_vector(27 DOWNTO 0);
            K: OUT std_logic_vector(47 DOWNTO 0));
      END COMPONENT;

      SHARED VARIABLE
C0,C1,C2,C3,C4,C5,C6,C7,C8,C9,C10,C11,C12,C13,C14,C15,C16:
std_logic_vector(27 DOWNTO 0);
      SHARED VARIABLE
D0,D1,D2,D3,D4,D5,D6,D7,D8,D9,D10,D11,D12,D13,D14,D15,D16:
std_logic_vector(27 DOWNTO 0);

      SIGNAL
Reg1,Reg2,Reg3,Reg4,Reg5,Reg6,Reg7,Reg8,Reg9,Reg10,Reg11,Reg12,Reg13
,Reg14,Reg15: std_logic_vector(55 DOWNTO 0);
      SIGNAL cipher_int: std_logic_vector(63 DOWNTO 0);

BEGIN

      PermChoice_1: PC_1 PORT MAP(KEY,C0,D0);
      PermChoice1_2: PC_2 PORT MAP(C0,D0,K1);

      ShiftRotate1_1 : SRR_1 PORT MAP(Reg1(55 DOWNTO 28),Reg1(27
DOWNTO 0),C1,D1);
```

```
      PermChoice2_2: PC_2 PORT MAP(C1,D1,K2);

      ShiftRotate2_2 : SRR_2 PORT MAP(Reg2(55 DOWNTO 28),Reg2(27
DOWNTO 0),C2,D2);
      PermChoice3_2: PC_2 PORT MAP(C2,D2,K3);

      ShiftRotate3_2 : SRR_2 PORT MAP(Reg3(55 DOWNTO 28),Reg3(27
DOWNTO 0),C3,D3);
      PermChoice4_2: PC_2 PORT MAP(C3,D3,K4);

      ShiftRotate4_2 : SRR_2 PORT MAP(Reg4(55 DOWNTO 28),Reg4(27
DOWNTO 0),C4,D4);
      PermChoice5_2: PC_2 PORT MAP(C4,D4,K5);

      ShiftRotate5_2 : SRR_2 PORT MAP(Reg5(55 DOWNTO 28),Reg5(27
DOWNTO 0),C5,D5);
      PermChoice6_2: PC_2 PORT MAP(C5,D5,K6);

      ShiftRotate6_2 : SRR_2 PORT MAP(Reg6(55 DOWNTO 28),Reg6(27
DOWNTO 0),C6,D6);
      PermChoice7_2: PC_2 PORT MAP(C6,D6,K7);

      ShiftRotate7_2 : SRR_2 PORT MAP(Reg7(55 DOWNTO 28),Reg7(27
DOWNTO 0),C7,D7);
      PermChoice8_2: PC_2 PORT MAP(C7,D7,K8);

      ShiftRotate8_1 : SRR_1 PORT MAP(Reg8(55 DOWNTO 28),Reg8(27
DOWNTO 0),C8,D8);
      PermChoice9_2: PC_2 PORT MAP(C8,D8,K9);

      ShiftRotate9_2 : SRR_2 PORT MAP(Reg9(55 DOWNTO 28),Reg9(27
DOWNTO 0),C9,D9);
      PermChoice10_2: PC_2 PORT MAP(C9,D9,K10);

      ShiftRotate10_2 : SRR_2 PORT MAP(Reg10(55 DOWNTO 28),Reg10(27
DOWNTO 0),C10,D10);
      PermChoice11_2: PC_2 PORT MAP(C10,D10,K11);

      ShiftRotate11_2 : SRR_2 PORT MAP(Reg11(55 DOWNTO 28),Reg11(27
DOWNTO 0),C11,D11);
      PermChoice12_2: PC_2 PORT MAP(C11,D11,K12);

      ShiftRotate12_2 : SRR_2 PORT MAP(Reg12(55 DOWNTO 28),Reg12(27
DOWNTO 0),C12,D12);
      PermChoice13_2: PC_2 PORT MAP(C12,D12,K13);

      ShiftRotate13_2 : SRR_2 PORT MAP(Reg13(55 DOWNTO 28),Reg13(27
DOWNTO 0),C13,D13);
      PermChoice14_2: PC_2 PORT MAP(C13,D13,K14);

      ShiftRotate14_2 : SRR_2 PORT MAP(Reg14(55 DOWNTO 28),Reg14(27
DOWNTO 0),C14,D14);
      PermChoice15_2: PC_2 PORT MAP(C14,D14,K15);
```

```
      ShiftRotate15_1 : SRR_1 PORT MAP(Reg15(55 DOWNTO 28),Reg15(27
DOWNTO 0),C15,D15);
      PermChoice16_2: PC_2 PORT MAP(C15,D15,K16);

      PROCESS(clk,KEY)
      BEGIN
            IF (falling_edge(clk)) THEN
                  FOR n IN 0 TO 14 LOOP
                        CASE n IS
                              WHEN 0 => Reg1 <= C0 & D0;
                              WHEN 1 => Reg2 <= C1 & D1;
                              WHEN 2 => Reg3 <= C2 & D2;
                              WHEN 3 => Reg4 <= C3 & D3;
                              WHEN 4 => Reg5 <= C4 & D4;
                              WHEN 5 => Reg6 <= C5 & D5;
                              WHEN 6 => Reg7 <= C6 & D6;
                              WHEN 7 => Reg8 <= C7 & D7;
                              WHEN 8 => Reg9 <= C8 & D8;
                              WHEN 9 => Reg10 <= C9 & D9;
                              WHEN 10 => Reg11 <= C10 & D10;
                              WHEN 11 => Reg12 <= C11 & D11;
                              WHEN 12 => Reg13 <= C12 & D12;
                              WHEN 13 => Reg14 <= C13      & D13;
                              WHEN 14 => Reg15 <= C14 & D14;
                        END CASE;
                  END LOOP;
            END IF;
      END PROCESS;
END keys;
```

KS_E.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;

ENTITY KS_E IS
      PORT(KEY: IN std_logic_vector(63 DOWNTO 0);

      K1,K2,K3,K4,K5,K6,K7,K8,K9,K10,K11,K12,K13,K14,K15,K16: OUT
std_logic_vector(47 DOWNTO 0));
END KS_E;

ARCHITECTURE keys OF KS_E IS

      COMPONENT PC_1
      PORT( key: IN std_logic_vector(63 DOWNTO 0);
                  c0: OUT std_logic_vector(27 DOWNTO 0);
                  d0: OUT std_logic_vector(27 DOWNTO 0));
      END COMPONENT;

      COMPONENT SRL_1
            PORT( C,D: IN std_logic_vector(27 DOWNTO 0);
                        CS,DS: OUT std_logic_vector(27 DOWNTO 0));
      END COMPONENT;
```

```
      COMPONENT SRL_2
            PORT( C,D: IN std_logic_vector(27 DOWNTO 0);
                        CS,DS: OUT std_logic_vector(27 DOWNTO 0));
      END COMPONENT;

      COMPONENT PC_2
      PORT(CS,DS: IN std_logic_vector(27 DOWNTO 0);
            K: OUT std_logic_vector(47 DOWNTO 0));
      END COMPONENT;

      SHARED VARIABLE
C0,C1,C2,C3,C4,C5,C6,C7,C8,C9,C10,C11,C12,C13,C14,C15,C16:
std_logic_vector(27 DOWNTO 0);
      SHARED VARIABLE
D0,D1,D2,D3,D4,D5,D6,D7,D8,D9,D10,D11,D12,D13,D14,D15,D16:
std_logic_vector(27 DOWNTO 0);

      SIGNAL
Reg1,Reg2,Reg3,Reg4,Reg5,Reg6,Reg7,Reg8,Reg9,Reg10,Reg11,Reg12,Reg13
,Reg14,Reg15: std_logic_vector(55 DOWNTO 0);
      SIGNAL cipher_int: std_logic_vector(63 DOWNTO 0);

BEGIN

      PermChoice_1: PC_1 PORT MAP(KEY,C0,D0);

      ShiftRotate1_1 : SRL_1 PORT MAP(C0,D0,C1,D1);
      PermChoice1_2: PC_2 PORT MAP(C1,D1,K1);
      Reg1 <= C1 & D1;

      ShiftRotate2_1 : SRL_1 PORT MAP(Reg1(55 DOWNTO 28),Reg1(27
DOWNTO 0),C2,D2);
      PermChoice2_2: PC_2 PORT MAP(C2,D2,K2);
      Reg2 <= C2 & D2;

      ShiftRotate3_2 : SRL_2 PORT MAP(Reg2(55 DOWNTO 28),Reg2(27
DOWNTO 0),C3,D3);
      PermChoice3_2: PC_2 PORT MAP(C3,D3,K3);
      Reg3 <= C3 & D3;

      ShiftRotate4_2 : SRL_2 PORT MAP(Reg3(55 DOWNTO 28),Reg3(27
DOWNTO 0),C4,D4);
      PermChoice4_2: PC_2 PORT MAP(C4,D4,K4);
      Reg4 <= C4 & D4;

      ShiftRotate5_2 : SRL_2 PORT MAP(Reg4(55 DOWNTO 28),Reg4(27
DOWNTO 0),C5,D5);
      PermChoice5_2: PC_2 PORT MAP(C5,D5,K5);
      Reg5 <= C5 & D5;

      ShiftRotate6_2 : SRL_2 PORT MAP(Reg5(55 DOWNTO 28),Reg5(27
DOWNTO 0),C6,D6);
      PermChoice6_2: PC_2 PORT MAP(C6,D6,K6);
      Reg6 <= C6 & D6;
```

```
      ShiftRotate7_2 : SRL_2 PORT MAP(Reg6(55 DOWNTO 28),Reg6(27
DOWNTO 0),C7,D7);
      PermChoice7_2: PC_2 PORT MAP(C7,D7,K7);
      Reg7 <= C7 & D7;

      ShiftRotate8_2 : SRL_2 PORT MAP(Reg7(55 DOWNTO 28),Reg7(27
DOWNTO 0),C8,D8);
      PermChoice8_2: PC_2 PORT MAP(C8,D8,K8);
      Reg8 <= C8 & D8;

      ShiftRotate9_1 : SRL_1 PORT MAP(Reg8(55 DOWNTO 28),Reg8(27
DOWNTO 0),C9,D9);
      PermChoice9_2: PC_2 PORT MAP(C9,D9,K9);
      Reg9 <= C9 & D9;

      ShiftRotate10_2 : SRL_2 PORT MAP(Reg9(55 DOWNTO 28),Reg9(27
DOWNTO 0),C10,D10);
      PermChoice10_2: PC_2 PORT MAP(C10,D10,K10);
      Reg10 <= C10 & D10;

      ShiftRotate11_2 : SRL_2 PORT MAP(Reg10(55 DOWNTO 28),Reg10(27
DOWNTO 0),C11,D11);
      PermChoice11_2: PC_2 PORT MAP(C11,D11,K11);
      Reg11 <= C11 & D11;

      ShiftRotate12_2 : SRL_2 PORT MAP(Reg11(55 DOWNTO 28),Reg11(27
DOWNTO 0),C12,D12);
      PermChoice12_2: PC_2 PORT MAP(C12,D12,K12);
      Reg12 <= C12 & D12;

      ShiftRotate13_2 : SRL_2 PORT MAP(Reg12(55 DOWNTO 28),Reg12(27
DOWNTO 0),C13,D13);
      PermChoice13_2: PC_2 PORT MAP(C13,D13,K13);
      Reg13 <= C13 & D13;

      ShiftRotate14_2 : SRL_2 PORT MAP(Reg13(55 DOWNTO 28),Reg13(27
DOWNTO 0),C14,D14);
      PermChoice14_2: PC_2 PORT MAP(C14,D14,K14);
      Reg14 <= C14 & D14;

      ShiftRotate15_2 : SRL_2 PORT MAP(Reg14(55 DOWNTO 28),Reg14(27
DOWNTO 0),C15,D15);
      PermChoice15_2: PC_2 PORT MAP(C15,D15,K15);
      Reg15 <= C15 & D15;

      ShiftRotate16_1 : SRL_1 PORT MAP(Reg15(55 DOWNTO 28),Reg15(27
DOWNTO 0),C16,D16);
      PermChoice16_2: PC_2 PORT MAP(C16,D16,K16);

END keys;
```

KS_E_P.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;

ENTITY KS_E_P IS
      PORT( clk: IN std_logic;
                  KEY: IN std_logic_vector(63 DOWNTO 0);

      K1,K2,K3,K4,K5,K6,K7,K8,K9,K10,K11,K12,K13,K14,K15,K16: OUT
std_logic_vector(47 DOWNTO 0));
END KS_E_P;

ARCHITECTURE keys OF KS_E_P IS

      COMPONENT PC_1
      PORT( key: IN std_logic_vector(63 DOWNTO 0);
                  c0: OUT std_logic_vector(27 DOWNTO 0);
                  d0: OUT std_logic_vector(27 DOWNTO 0));
      END COMPONENT;

      COMPONENT SRL_1
            PORT( C,D: IN std_logic_vector(27 DOWNTO 0);
                        CS,DS: OUT std_logic_vector(27 DOWNTO 0));
      END COMPONENT;

      COMPONENT SRL_2
            PORT( C,D: IN std_logic_vector(27 DOWNTO 0);
                        CS,DS: OUT std_logic_vector(27 DOWNTO 0));
      END COMPONENT;

      COMPONENT PC_2
      PORT(CS,DS: IN std_logic_vector(27 DOWNTO 0);
            K: OUT std_logic_vector(47 DOWNTO 0));
      END COMPONENT;

      SHARED VARIABLE
C0,C1,C2,C3,C4,C5,C6,C7,C8,C9,C10,C11,C12,C13,C14,C15,C16:
std_logic_vector(27 DOWNTO 0);
      SHARED VARIABLE
D0,D1,D2,D3,D4,D5,D6,D7,D8,D9,D10,D11,D12,D13,D14,D15,D16:
std_logic_vector(27 DOWNTO 0);

      SIGNAL
Reg1,Reg2,Reg3,Reg4,Reg5,Reg6,Reg7,Reg8,Reg9,Reg10,Reg11,Reg12,Reg13
,Reg14,Reg15: std_logic_vector(55 DOWNTO 0);
      SIGNAL cipher_int: std_logic_vector(63 DOWNTO 0);

BEGIN

      PermChoice_1: PC_1 PORT MAP(KEY,C0,D0);

      ShiftRotate1_1 : SRL_1 PORT MAP(C0,D0,C1,D1);
      PermChoice1_2: PC_2 PORT MAP(C1,D1,K1);

      ShiftRotate2_1 : SRL_1 PORT MAP(Reg1(55 DOWNTO 28),Reg1(27
DOWNTO 0),C2,D2);
```

```
      PermChoice2_2: PC_2 PORT MAP(C2,D2,K2);

      ShiftRotate3_2 : SRL_2 PORT MAP(Reg2(55 DOWNTO 28),Reg2(27
DOWNTO 0),C3,D3);
      PermChoice3_2: PC_2 PORT MAP(C3,D3,K3);

      ShiftRotate4_2 : SRL_2 PORT MAP(Reg3(55 DOWNTO 28),Reg3(27
DOWNTO 0),C4,D4);
      PermChoice4_2: PC_2 PORT MAP(C4,D4,K4);

      ShiftRotate5_2 : SRL_2 PORT MAP(Reg4(55 DOWNTO 28),Reg4(27
DOWNTO 0),C5,D5);
      PermChoice5_2: PC_2 PORT MAP(C5,D5,K5);

      ShiftRotate6_2 : SRL_2 PORT MAP(Reg5(55 DOWNTO 28),Reg5(27
DOWNTO 0),C6,D6);
      PermChoice6_2: PC_2 PORT MAP(C6,D6,K6);

      ShiftRotate7_2 : SRL_2 PORT MAP(Reg6(55 DOWNTO 28),Reg6(27
DOWNTO 0),C7,D7);
      PermChoice7_2: PC_2 PORT MAP(C7,D7,K7);

      ShiftRotate8_2 : SRL_2 PORT MAP(Reg7(55 DOWNTO 28),Reg7(27
DOWNTO 0),C8,D8);
      PermChoice8_2: PC_2 PORT MAP(C8,D8,K8);

      ShiftRotate9_1 : SRL_1 PORT MAP(Reg8(55 DOWNTO 28),Reg8(27
DOWNTO 0),C9,D9);
      PermChoice9_2: PC_2 PORT MAP(C9,D9,K9);

      ShiftRotate10_2 : SRL_2 PORT MAP(Reg9(55 DOWNTO 28),Reg9(27
DOWNTO 0),C10,D10);
      PermChoice10_2: PC_2 PORT MAP(C10,D10,K10);

      ShiftRotate11_2 : SRL_2 PORT MAP(Reg10(55 DOWNTO 28),Reg10(27
DOWNTO 0),C11,D11);
      PermChoice11_2: PC_2 PORT MAP(C11,D11,K11);

      ShiftRotate12_2 : SRL_2 PORT MAP(Reg11(55 DOWNTO 28),Reg11(27
DOWNTO 0),C12,D12);
      PermChoice12_2: PC_2 PORT MAP(C12,D12,K12);

      ShiftRotate13_2 : SRL_2 PORT MAP(Reg12(55 DOWNTO 28),Reg12(27
DOWNTO 0),C13,D13);
      PermChoice13_2: PC_2 PORT MAP(C13,D13,K13);

      ShiftRotate14_2 : SRL_2 PORT MAP(Reg13(55 DOWNTO 28),Reg13(27
DOWNTO 0),C14,D14);
      PermChoice14_2: PC_2 PORT MAP(C14,D14,K14);

      ShiftRotate15_2 : SRL_2 PORT MAP(Reg14(55 DOWNTO 28),Reg14(27
DOWNTO 0),C15,D15);
      PermChoice15_2: PC_2 PORT MAP(C15,D15,K15);
```

```
      ShiftRotate16_1 : SRL_1 PORT MAP(Reg15(55 DOWNTO 28),Reg15(27
DOWNTO 0),C16,D16);
      PermChoice16_2: PC_2 PORT MAP(C16,D16,K16);

      PROCESS(clk,KEY)
      BEGIN
            IF (falling_edge(clk)) THEN
                  FOR n IN 0 TO 14 LOOP
                        CASE n IS
                              WHEN 0 => Reg1 <= C1 & D1;
                              WHEN 1 => Reg2 <= C2 & D2;
                              WHEN 2 => Reg3 <= C3 & D3;
                              WHEN 3 => Reg4 <= C4 & D4;
                              WHEN 4 => Reg5 <= C5 & D5;
                              WHEN 5 => Reg6 <= C6 & D6;
                              WHEN 6 => Reg7 <= C7 & D7;
                              WHEN 7 => Reg8 <= C8 & D8;
                              WHEN 8 => Reg9 <= C9 & D9;
                              WHEN 9 => Reg10 <= C10 & D10;
                              WHEN 10 => Reg11 <= C11 & D11;
                              WHEN 11 => Reg12 <= C12 & D12;
                              WHEN 12 => Reg13 <= C13 & D13;
                              WHEN 13 => Reg14 <= C14 & D14;
                              WHEN 14 => Reg15 <= C15 & D15;
                        END CASE;
                  END LOOP;
            END IF;
      END PROCESS;
END keys;
```

PC_1.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;

ENTITY PC_1 IS
PORT( key: IN std_logic_vector(63 DOWNTO 0);
            c0: OUT std_logic_vector(27 DOWNTO 0);
            d0: OUT std_logic_vector(27 DOWNTO 0));
END PC_1;

ARCHITECTURE permutation OF PC_1 IS
BEGIN
      c0 <= key(7) & key(15) & key(23) & key(31) & key(39) & key(47)
& key(55) &
                                    key(63) & key(6) & key(14) &
key(22) & key(30) & key(38) & key(46) &
                                    key(54) & key(62) & key(5) &
key(13) & key(21) & key(29) & key(37) &
                                    key(45) & key(53) & key(61) &
key(4) & key(12) & key(20) & key(28);

      d0 <= key(1) & key(9) & key(17) & key(25) & key(33) & key(41)
& key(49) &
```

```
                                  key(57) & key(2) & key(10) &
key(18) & key(26) & key(34) & key(42) &
                                  key(50) & key(58) & key(3) &
key(11) & key(19) & key(27) & key(35) &
                                  key(43) & key(51) & key(59) &
key(36) & key(44) & key(52) & key(60);
END
permutation;166166166166166166166166166166166166166166166166166166
166166
```

PC_2.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;

ENTITY PC_2 IS
PORT(CS,DS: IN std_logic_vector(27 DOWNTO 0);
      K: OUT std_logic_vector(47 DOWNTO 0));
END PC_2;

ARCHITECTURE permutation OF PC_2 IS

      SHARED VARIABLE CS_DS: std_logic_vector(55 DOWNTO 0) := CS &
DS;

BEGIN
            K <= CS_DS(42) & CS_DS(39) & CS_DS(45) & CS_DS(32) &
CS_DS(55) & CS_DS(51) &
                 CS_DS(53) & CS_DS(28) & CS_DS(41) & CS_DS(50) &
CS_DS(35) & CS_DS(46) &
                 CS_DS(33) & CS_DS(37) & CS_DS(44) & CS_DS(52) &
CS_DS(30) & CS_DS(48) &
                 CS_DS(40) & CS_DS(49) & CS_DS(29) & CS_DS(36) &
CS_DS(43) & CS_DS(54) &
                 CS_DS(15) & CS_DS(4) & CS_DS(25) & CS_DS(19) &
CS_DS(9) & CS_DS(1) &
                 CS_DS(26) & CS_DS(16) & CS_DS(5) & CS_DS(11) &
CS_DS(23) & CS_DS(8) &
                 CS_DS(12) & CS_DS(7) & CS_DS(17) & CS_DS(0) &
CS_DS(22) & CS_DS(3) &
                 CS_DS(10) & CS_DS(14) & CS_DS(6) & CS_DS(20) &
CS_DS(27) & CS_DS(24);

END permutation;
```

S1.vhd

```
--EDNI DEL ROSAL----S1.vhd-----------------------
--Thesis Spring 2015-----------------------------
--This file performs the substitution of the box-
--S1.--------------------------------------------
--The input is B1 and the output is S_B1---------
library IEEE;
```

```
use IEEE.std_logic_1164.all;

ENTITY S1 IS
      PORT(B1: IN std_logic_vector(5 DOWNTO 0);
                 S_B1: OUT std_logic_vector(3 DOWNTO 0));
END S1;

ARCHITECTURE substitute OF S1 IS

            TYPE mem IS ARRAY (0 to 15) OF std_logic_vector (3 DOWNTO
0);
            constant sub_out : mem := (
            0 => "0000",
            1 => "0001",
            2 => "0010",
            3 => "0011",
            4 => "0100",
            5 => "0101",
            6 => "0110",
            7 => "0111",
            8 => "1000",
            9 => "1001",
            10 => "1010",
            11 => "1011",
            12 => "1100",
            13 => "1101",
            14 => "1110",
            15 => "1111");

BEGIN
            PROCESS (B1)
            BEGIN
                  CASE B1 IS
                        WHEN "000000" => S_B1 <= sub_out(14);
                        WHEN "000010" => S_B1 <= sub_out(4);
                        WHEN "000100" => S_B1 <= sub_out(13);
                        WHEN "000110" => S_B1 <= sub_out(1);
                        WHEN "001000" => S_B1 <= sub_out(2);
                        WHEN "001010" => S_B1 <= sub_out(15);
                        WHEN "001100" => S_B1 <= sub_out(11);
                        WHEN "001110" => S_B1 <= sub_out(8);
                        WHEN "010000" => S_B1 <= sub_out(3);
                        WHEN "010010" => S_B1 <= sub_out(10);
                        WHEN "010100" => S_B1 <= sub_out(6);
                        WHEN "010110" => S_B1 <= sub_out(12);
                        WHEN "011000" => S_B1 <= sub_out(5);
                        WHEN "011010" => S_B1 <= sub_out(9);
                        WHEN "011100" => S_B1 <= sub_out(0);
                        WHEN "011110" => S_B1 <= sub_out(7);

                        WHEN "000001" => S_B1 <= sub_out(0);
                        WHEN "000011" => S_B1 <= sub_out(15);
                        WHEN "000101" => S_B1 <= sub_out(7);
                        WHEN "000111" => S_B1 <= sub_out(4);
                        WHEN "001001" => S_B1 <= sub_out(14);
```

```
                    WHEN "001011" => S_B1 <= sub_out(2);
                    WHEN "001101" => S_B1 <= sub_out(13);
                    WHEN "001111" => S_B1 <= sub_out(1);
                    WHEN "010001" => S_B1 <= sub_out(10);
                    WHEN "010011" => S_B1 <= sub_out(6);
                    WHEN "010101" => S_B1 <= sub_out(12);
                    WHEN "010111" => S_B1 <= sub_out(11);
                    WHEN "011001" => S_B1 <= sub_out(9);
                    WHEN "011011" => S_B1 <= sub_out(5);
                    WHEN "011101" => S_B1 <= sub_out(3);
                    WHEN "011111" => S_B1 <= sub_out(8);

                    WHEN "100000" => S_B1 <= sub_out(4);
                    WHEN "100010" => S_B1 <= sub_out(1);
                    WHEN "100100" => S_B1 <= sub_out(14);
                    WHEN "100110" => S_B1 <= sub_out(8);
                    WHEN "101000" => S_B1 <= sub_out(13);
                    WHEN "101010" => S_B1 <= sub_out(6);
                    WHEN "101100" => S_B1 <= sub_out(2);
                    WHEN "101110" => S_B1 <= sub_out(11);
                    WHEN "110000" => S_B1 <= sub_out(15);
                    WHEN "110010" => S_B1 <= sub_out(12);
                    WHEN "110100" => S_B1 <= sub_out(9);
                    WHEN "110110" => S_B1 <= sub_out(7);
                    WHEN "111000" => S_B1 <= sub_out(3);
                    WHEN "111010" => S_B1 <= sub_out(10);
                    WHEN "111100" => S_B1 <= sub_out(5);
                    WHEN "111110" => S_B1 <= sub_out(0);

                    WHEN "100001" => S_B1 <= sub_out(15);
                    WHEN "100011" => S_B1 <= sub_out(12);
                    WHEN "100101" => S_B1 <= sub_out(8);
                    WHEN "100111" => S_B1 <= sub_out(2);
                    WHEN "101001" => S_B1 <= sub_out(4);
                    WHEN "101011" => S_B1 <= sub_out(9);
                    WHEN "101101" => S_B1 <= sub_out(1);
                    WHEN "101111" => S_B1 <= sub_out(7);
                    WHEN "110001" => S_B1 <= sub_out(5);
                    WHEN "110011" => S_B1 <= sub_out(11);
                    WHEN "110101" => S_B1 <= sub_out(3);
                    WHEN "110111" => S_B1 <= sub_out(14);
                    WHEN "111001" => S_B1 <= sub_out(10);
                    WHEN "111011" => S_B1 <= sub_out(0);
                    WHEN "111101" => S_B1 <= sub_out(6);
                    WHEN "111111" => S_B1 <= sub_out(13);
                END CASE;
            END PROCESS;
END architecture substitute;
```

S2.vhd

```
--EDNI DEL ROSAL----S2.vhd-----------------------
--Thesis Spring 2015-----------------------------
--This file performs the substitution of the box-
```

```
--S2.-------------------------------------------
--The input is B2 and the output is S_B2---------
library IEEE;
use IEEE.std_logic_1164.all;

ENTITY S2 IS
      PORT(B2: IN std_logic_vector(5 DOWNTO 0);
                 S_B2: OUT std_logic_vector(3 DOWNTO 0));
END S2;

ARCHITECTURE substitute OF S2 IS

           TYPE mem IS ARRAY (0 to 15) OF std_logic_vector (3 DOWNTO
0);
           constant sub_out : mem := (
           0 => "0000",
           1 => "0001",
           2 => "0010",
           3 => "0011",
           4 => "0100",
           5 => "0101",
           6 => "0110",
           7 => "0111",
           8 => "1000",
           9 => "1001",
           10 => "1010",
           11 => "1011",
           12 => "1100",
           13 => "1101",
           14 => "1110",
           15 => "1111");

BEGIN
           PROCESS (B2)
           BEGIN
                 CASE B2 IS
                       WHEN "000000" => S_B2 <= sub_out(15);
                       WHEN "000010" => S_B2 <= sub_out(1);
                       WHEN "000100" => S_B2 <= sub_out(8);
                       WHEN "000110" => S_B2 <= sub_out(14);
                       WHEN "001000" => S_B2 <= sub_out(6);
                       WHEN "001010" => S_B2 <= sub_out(11);
                       WHEN "001100" => S_B2 <= sub_out(3);
                       WHEN "001110" => S_B2 <= sub_out(4);
                       WHEN "010000" => S_B2 <= sub_out(9);
                       WHEN "010010" => S_B2 <= sub_out(7);
                       WHEN "010100" => S_B2 <= sub_out(2);
                       WHEN "010110" => S_B2 <= sub_out(13);
                       WHEN "011000" => S_B2 <= sub_out(12);
                       WHEN "011010" => S_B2 <= sub_out(0);
                       WHEN "011100" => S_B2 <= sub_out(5);
                       WHEN "011110" => S_B2 <= sub_out(10);

                       WHEN "000001" => S_B2 <= sub_out(3);
                       WHEN "000011" => S_B2 <= sub_out(13);
```

```
                    WHEN "000101" => S_B2 <= sub_out(4);
                    WHEN "000111" => S_B2 <= sub_out(7);
                    WHEN "001001" => S_B2 <= sub_out(15);
                    WHEN "001011" => S_B2 <= sub_out(2);
                    WHEN "001101" => S_B2 <= sub_out(8);
                    WHEN "001111" => S_B2 <= sub_out(14);
                    WHEN "010001" => S_B2 <= sub_out(12);
                    WHEN "010011" => S_B2 <= sub_out(0);
                    WHEN "010101" => S_B2 <= sub_out(1);
                    WHEN "010111" => S_B2 <= sub_out(10);
                    WHEN "011001" => S_B2 <= sub_out(6);
                    WHEN "011011" => S_B2 <= sub_out(9);
                    WHEN "011101" => S_B2 <= sub_out(11);
                    WHEN "011111" => S_B2 <= sub_out(5);

                    WHEN "100000" => S_B2 <= sub_out(0);
                    WHEN "100010" => S_B2 <= sub_out(14);
                    WHEN "100100" => S_B2 <= sub_out(7);
                    WHEN "100110" => S_B2 <= sub_out(11);
                    WHEN "101000" => S_B2 <= sub_out(10);
                    WHEN "101010" => S_B2 <= sub_out(4);
                    WHEN "101100" => S_B2 <= sub_out(13);
                    WHEN "101110" => S_B2 <= sub_out(1);
                    WHEN "110000" => S_B2 <= sub_out(5);
                    WHEN "110010" => S_B2 <= sub_out(8);
                    WHEN "110100" => S_B2 <= sub_out(12);
                    WHEN "110110" => S_B2 <= sub_out(6);
                    WHEN "111000" => S_B2 <= sub_out(9);
                    WHEN "111010" => S_B2 <= sub_out(3);
                    WHEN "111100" => S_B2 <= sub_out(2);
                    WHEN "111110" => S_B2 <= sub_out(15);

                    WHEN "100001" => S_B2 <= sub_out(13);
                    WHEN "100011" => S_B2 <= sub_out(8);
                    WHEN "100101" => S_B2 <= sub_out(10);
                    WHEN "100111" => S_B2 <= sub_out(1);
                    WHEN "101001" => S_B2 <= sub_out(3);
                    WHEN "101011" => S_B2 <= sub_out(15);
                    WHEN "101101" => S_B2 <= sub_out(4);
                    WHEN "101111" => S_B2 <= sub_out(2);
                    WHEN "110001" => S_B2 <= sub_out(11);
                    WHEN "110011" => S_B2 <= sub_out(6);
                    WHEN "110101" => S_B2 <= sub_out(7);
                    WHEN "110111" => S_B2 <= sub_out(12);
                    WHEN "111001" => S_B2 <= sub_out(0);
                    WHEN "111011" => S_B2 <= sub_out(5);
                    WHEN "111101" => S_B2 <= sub_out(14);
                    WHEN "111111" => S_B2 <= sub_out(9);
                END CASE;
            END PROCESS;
END architecture substitute;
```

S3.vhd

```
--EDNI DEL ROSAL----S3.vhd----------------------
--Thesis Spring 2015----------------------------
--This file performs the substitution of the box-
--S3.-------------------------------------------
--The input is B3 and the output is S_B3---------
library IEEE;
use IEEE.std_logic_1164.all;

ENTITY S3 IS
      PORT(B3: IN std_logic_vector(5 DOWNTO 0);
                 S_B3: OUT std_logic_vector(3 DOWNTO 0));
END S3;

ARCHITECTURE substitute OF S3 IS

            TYPE mem IS ARRAY (0 to 15) OF std_logic_vector (3 DOWNTO
0);
            constant sub_out : mem := (
            0 => "0000",
            1 => "0001",
            2 => "0010",
            3 => "0011",
            4 => "0100",
            5 => "0101",
            6 => "0110",
            7 => "0111",
            8 => "1000",
            9 => "1001",
            10 => "1010",
            11 => "1011",
            12 => "1100",
            13 => "1101",
            14 => "1110",
            15 => "1111");

BEGIN
            PROCESS (B3)
            BEGIN
                  CASE B3 IS
                        WHEN "000000" => S_B3 <= sub_out(10);
                        WHEN "000010" => S_B3 <= sub_out(0);
                        WHEN "000100" => S_B3 <= sub_out(9);
                        WHEN "000110" => S_B3 <= sub_out(14);
                        WHEN "001000" => S_B3 <= sub_out(6);
                        WHEN "001010" => S_B3 <= sub_out(3);
                        WHEN "001100" => S_B3 <= sub_out(15);
                        WHEN "001110" => S_B3 <= sub_out(5);
                        WHEN "010000" => S_B3 <= sub_out(1);
                        WHEN "010010" => S_B3 <= sub_out(13);
                        WHEN "010100" => S_B3 <= sub_out(12);
                        WHEN "010110" => S_B3 <= sub_out(7);
                        WHEN "011000" => S_B3 <= sub_out(11);
                        WHEN "011010" => S_B3 <= sub_out(4);
                        WHEN "011100" => S_B3 <= sub_out(2);
                        WHEN "011110" => S_B3 <= sub_out(8);
```

```
                WHEN "000001" => S_B3 <= sub_out(13);
                WHEN "000011" => S_B3 <= sub_out(7);
                WHEN "000101" => S_B3 <= sub_out(0);
                WHEN "000111" => S_B3 <= sub_out(9);
                WHEN "001001" => S_B3 <= sub_out(3);
                WHEN "001011" => S_B3 <= sub_out(4);
                WHEN "001101" => S_B3 <= sub_out(6);
                WHEN "001111" => S_B3 <= sub_out(10);
                WHEN "010001" => S_B3 <= sub_out(2);
                WHEN "010011" => S_B3 <= sub_out(8);
                WHEN "010101" => S_B3 <= sub_out(5);
                WHEN "010111" => S_B3 <= sub_out(14);
                WHEN "011001" => S_B3 <= sub_out(12);
                WHEN "011011" => S_B3 <= sub_out(11);
                WHEN "011101" => S_B3 <= sub_out(15);
                WHEN "011111" => S_B3 <= sub_out(1);

                WHEN "100000" => S_B3 <= sub_out(13);
                WHEN "100010" => S_B3 <= sub_out(6);
                WHEN "100100" => S_B3 <= sub_out(4);
                WHEN "100110" => S_B3 <= sub_out(9);
                WHEN "101000" => S_B3 <= sub_out(8);
                WHEN "101010" => S_B3 <= sub_out(15);
                WHEN "101100" => S_B3 <= sub_out(3);
                WHEN "101110" => S_B3 <= sub_out(0);
                WHEN "110000" => S_B3 <= sub_out(11);
                WHEN "110010" => S_B3 <= sub_out(1);
                WHEN "110100" => S_B3 <= sub_out(2);
                WHEN "110110" => S_B3 <= sub_out(12);
                WHEN "111000" => S_B3 <= sub_out(5);
                WHEN "111010" => S_B3 <= sub_out(10);
                WHEN "111100" => S_B3 <= sub_out(14);
                WHEN "111110" => S_B3 <= sub_out(7);

                WHEN "100001" => S_B3 <= sub_out(1);
                WHEN "100011" => S_B3 <= sub_out(10);
                WHEN "100101" => S_B3 <= sub_out(13);
                WHEN "100111" => S_B3 <= sub_out(0);
                WHEN "101001" => S_B3 <= sub_out(6);
                WHEN "101011" => S_B3 <= sub_out(9);
                WHEN "101101" => S_B3 <= sub_out(8);
                WHEN "101111" => S_B3 <= sub_out(7);
                WHEN "110001" => S_B3 <= sub_out(4);
                WHEN "110011" => S_B3 <= sub_out(15);
                WHEN "110101" => S_B3 <= sub_out(14);
                WHEN "110111" => S_B3 <= sub_out(3);
                WHEN "111001" => S_B3 <= sub_out(11);
                WHEN "111011" => S_B3 <= sub_out(5);
                WHEN "111101" => S_B3 <= sub_out(2);
                WHEN "111111" => S_B3 <= sub_out(12);
            END CASE;
        END PROCESS;
END architecture substitute;
```

S4.vhd

```
--EDNI DEL ROSAL----S4.vhd----------------------
--Thesis Spring 2015----------------------------
--This file performs the substitution of the box-
--S4.------------------------------------------
--The input is B4 and the output is S_B4---------
library IEEE;
use IEEE.std_logic_1164.all;

ENTITY S4 IS
      PORT(B4: IN std_logic_vector(5 DOWNTO 0);
                  S_B4: OUT std_logic_vector(3 DOWNTO 0));
END S4;

ARCHITECTURE substitute OF S4 IS

            TYPE mem IS ARRAY (0 to 15) OF std_logic_vector (3 DOWNTO
0);
            constant sub_out : mem := (
            0 => "0000",
            1 => "0001",
            2 => "0010",
            3 => "0011",
            4 => "0100",
            5 => "0101",
            6 => "0110",
            7 => "0111",
            8 => "1000",
            9 => "1001",
            10 => "1010",
            11 => "1011",
            12 => "1100",
            13 => "1101",
            14 => "1110",
            15 => "1111");

BEGIN
            PROCESS (B4)
            BEGIN
                  CASE B4 IS
                        WHEN "000000" => S_B4 <= sub_out(7);
                        WHEN "000010" => S_B4 <= sub_out(13);
                        WHEN "000100" => S_B4 <= sub_out(14);
                        WHEN "000110" => S_B4 <= sub_out(3);
                        WHEN "001000" => S_B4 <= sub_out(0);
                        WHEN "001010" => S_B4 <= sub_out(6);
                        WHEN "001100" => S_B4 <= sub_out(9);
                        WHEN "001110" => S_B4 <= sub_out(10);
                        WHEN "010000" => S_B4 <= sub_out(1);
                        WHEN "010010" => S_B4 <= sub_out(2);
                        WHEN "010100" => S_B4 <= sub_out(8);
                        WHEN "010110" => S_B4 <= sub_out(5);
                        WHEN "011000" => S_B4 <= sub_out(11);
                        WHEN "011010" => S_B4 <= sub_out(12);
```

```
            WHEN "011100" => S_B4 <= sub_out(4);
            WHEN "011110" => S_B4 <= sub_out(15);

            WHEN "000001" => S_B4 <= sub_out(13);
            WHEN "000011" => S_B4 <= sub_out(8);
            WHEN "000101" => S_B4 <= sub_out(11);
            WHEN "000111" => S_B4 <= sub_out(5);
            WHEN "001001" => S_B4 <= sub_out(6);
            WHEN "001011" => S_B4 <= sub_out(15);
            WHEN "001101" => S_B4 <= sub_out(0);
            WHEN "001111" => S_B4 <= sub_out(3);
            WHEN "010001" => S_B4 <= sub_out(4);
            WHEN "010011" => S_B4 <= sub_out(7);
            WHEN "010101" => S_B4 <= sub_out(2);
            WHEN "010111" => S_B4 <= sub_out(12);
            WHEN "011001" => S_B4 <= sub_out(1);
            WHEN "011011" => S_B4 <= sub_out(10);
            WHEN "011101" => S_B4 <= sub_out(14);
            WHEN "011111" => S_B4 <= sub_out(9);

            WHEN "100000" => S_B4 <= sub_out(10);
            WHEN "100010" => S_B4 <= sub_out(6);
            WHEN "100100" => S_B4 <= sub_out(9);
            WHEN "100110" => S_B4 <= sub_out(0);
            WHEN "101000" => S_B4 <= sub_out(12);
            WHEN "101010" => S_B4 <= sub_out(11);
            WHEN "101100" => S_B4 <= sub_out(7);
            WHEN "101110" => S_B4 <= sub_out(13);
            WHEN "110000" => S_B4 <= sub_out(15);
            WHEN "110010" => S_B4 <= sub_out(1);
            WHEN "110100" => S_B4 <= sub_out(3);
            WHEN "110110" => S_B4 <= sub_out(14);
            WHEN "111000" => S_B4 <= sub_out(5);
            WHEN "111010" => S_B4 <= sub_out(2);
            WHEN "111100" => S_B4 <= sub_out(8);
            WHEN "111110" => S_B4 <= sub_out(4);

            WHEN "100001" => S_B4 <= sub_out(3);
            WHEN "100011" => S_B4 <= sub_out(15);
            WHEN "100101" => S_B4 <= sub_out(0);
            WHEN "100111" => S_B4 <= sub_out(6);
            WHEN "101001" => S_B4 <= sub_out(10);
            WHEN "101011" => S_B4 <= sub_out(1);
            WHEN "101101" => S_B4 <= sub_out(13);
            WHEN "101111" => S_B4 <= sub_out(8);
            WHEN "110001" => S_B4 <= sub_out(9);
            WHEN "110011" => S_B4 <= sub_out(4);
            WHEN "110101" => S_B4 <= sub_out(5);
            WHEN "110111" => S_B4 <= sub_out(11);
            WHEN "111001" => S_B4 <= sub_out(12);
            WHEN "111011" => S_B4 <= sub_out(7);
            WHEN "111101" => S_B4 <= sub_out(2);
            WHEN "111111" => S_B4 <= sub_out(14);
        END CASE;
    END PROCESS;
```

```
END architecture substitute;
```

S5.vhd

```
--EDNI DEL ROSAL----S5.vhd----------------------
--Thesis Spring 2015----------------------------
--This file performs the substitution of the box-
--S5.-------------------------------------------
--The input is B5 and the output is S_B5---------
library IEEE;
use IEEE.std_logic_1164.all;

ENTITY S5 IS
      PORT(B5: IN std_logic_vector(5 DOWNTO 0);
                  S_B5: OUT std_logic_vector(3 DOWNTO 0));
END S5;

ARCHITECTURE substitute OF S5 IS

            TYPE mem IS ARRAY (0 to 15) OF std_logic_vector (3 DOWNTO
0);
            constant sub_out : mem := (
            0 => "0000",
            1 => "0001",
            2 => "0010",
            3 => "0011",
            4 => "0100",
            5 => "0101",
            6 => "0110",
            7 => "0111",
            8 => "1000",
            9 => "1001",
            10 => "1010",
            11 => "1011",
            12 => "1100",
            13 => "1101",
            14 => "1110",
            15 => "1111");

BEGIN
            PROCESS (B5)
            BEGIN
                  CASE B5 IS
                        WHEN "000000" => S_B5 <= sub_out(2);
                        WHEN "000010" => S_B5 <= sub_out(12);
                        WHEN "000100" => S_B5 <= sub_out(4);
                        WHEN "000110" => S_B5 <= sub_out(1);
                        WHEN "001000" => S_B5 <= sub_out(7);
                        WHEN "001010" => S_B5 <= sub_out(10);
                        WHEN "001100" => S_B5 <= sub_out(11);
                        WHEN "001110" => S_B5 <= sub_out(6);
                        WHEN "010000" => S_B5 <= sub_out(8);
                        WHEN "010010" => S_B5 <= sub_out(5);
                        WHEN "010100" => S_B5 <= sub_out(3);
```

```
WHEN "010110" => S_B5 <= sub_out(15);
WHEN "011000" => S_B5 <= sub_out(13);
WHEN "011010" => S_B5 <= sub_out(0);
WHEN "011100" => S_B5 <= sub_out(14);
WHEN "011110" => S_B5 <= sub_out(9);

WHEN "000001" => S_B5 <= sub_out(14);
WHEN "000011" => S_B5 <= sub_out(11);
WHEN "000101" => S_B5 <= sub_out(2);
WHEN "000111" => S_B5 <= sub_out(12);
WHEN "001001" => S_B5 <= sub_out(4);
WHEN "001011" => S_B5 <= sub_out(7);
WHEN "001101" => S_B5 <= sub_out(13);
WHEN "001111" => S_B5 <= sub_out(1);
WHEN "010001" => S_B5 <= sub_out(5);
WHEN "010011" => S_B5 <= sub_out(0);
WHEN "010101" => S_B5 <= sub_out(15);
WHEN "010111" => S_B5 <= sub_out(10);
WHEN "011001" => S_B5 <= sub_out(3);
WHEN "011011" => S_B5 <= sub_out(9);
WHEN "011101" => S_B5 <= sub_out(8);
WHEN "011111" => S_B5 <= sub_out(6);

WHEN "100000" => S_B5 <= sub_out(4);
WHEN "100010" => S_B5 <= sub_out(2);
WHEN "100100" => S_B5 <= sub_out(1);
WHEN "100110" => S_B5 <= sub_out(11);
WHEN "101000" => S_B5 <= sub_out(10);
WHEN "101010" => S_B5 <= sub_out(13);
WHEN "101100" => S_B5 <= sub_out(7);
WHEN "101110" => S_B5 <= sub_out(8);
WHEN "110000" => S_B5 <= sub_out(15);
WHEN "110010" => S_B5 <= sub_out(9);
WHEN "110100" => S_B5 <= sub_out(12);
WHEN "110110" => S_B5 <= sub_out(5);
WHEN "111000" => S_B5 <= sub_out(6);
WHEN "111010" => S_B5 <= sub_out(3);
WHEN "111100" => S_B5 <= sub_out(0);
WHEN "111110" => S_B5 <= sub_out(14);

WHEN "100001" => S_B5 <= sub_out(11);
WHEN "100011" => S_B5 <= sub_out(8);
WHEN "100101" => S_B5 <= sub_out(12);
WHEN "100111" => S_B5 <= sub_out(7);
WHEN "101001" => S_B5 <= sub_out(1);
WHEN "101011" => S_B5 <= sub_out(14);
WHEN "101101" => S_B5 <= sub_out(2);
WHEN "101111" => S_B5 <= sub_out(13);
WHEN "110001" => S_B5 <= sub_out(6);
WHEN "110011" => S_B5 <= sub_out(15);
WHEN "110101" => S_B5 <= sub_out(0);
WHEN "110111" => S_B5 <= sub_out(9);
WHEN "111001" => S_B5 <= sub_out(10);
WHEN "111011" => S_B5 <= sub_out(4);
WHEN "111101" => S_B5 <= sub_out(5);
```

```
                        WHEN "111111" => S_B5 <= sub_out(3);
                  END CASE;
            END PROCESS;
END architecture substitute;
```

S6.vhd

```
--EDNI DEL ROSAL----S6.vhd-----------------------
--Thesis Spring 2015----------------------------
--This file performs the substitution of the box-
--S6.------------------------------------------
--The input is B6 and the output is S_B6---------
library IEEE;
use IEEE.std_logic_1164.all;

ENTITY S6 IS
      PORT(B6: IN std_logic_vector(5 DOWNTO 0);
                  S_B6: OUT std_logic_vector(3 DOWNTO 0));
END S6;

ARCHITECTURE substitute OF S6 IS

            TYPE mem IS ARRAY (0 to 15) OF std_logic_vector (3 DOWNTO
0);
            constant sub_out : mem := (
            0 => "0000",
            1 => "0001",
            2 => "0010",
            3 => "0011",
            4 => "0100",
            5 => "0101",
            6 => "0110",
            7 => "0111",
            8 => "1000",
            9 => "1001",
            10 => "1010",
            11 => "1011",
            12 => "1100",
            13 => "1101",
            14 => "1110",
            15 => "1111");

BEGIN
            PROCESS (B6)
            BEGIN
                  CASE B6 IS
                        WHEN "000000" => S_B6 <= sub_out(12);
                        WHEN "000010" => S_B6 <= sub_out(1);
                        WHEN "000100" => S_B6 <= sub_out(10);
                        WHEN "000110" => S_B6 <= sub_out(15);
                        WHEN "001000" => S_B6 <= sub_out(9);
                        WHEN "001010" => S_B6 <= sub_out(2);
                        WHEN "001100" => S_B6 <= sub_out(6);
                        WHEN "001110" => S_B6 <= sub_out(8);
```

```
WHEN "010000" => S_B6 <= sub_out(0);
WHEN "010010" => S_B6 <= sub_out(13);
WHEN "010100" => S_B6 <= sub_out(3);
WHEN "010110" => S_B6 <= sub_out(4);
WHEN "011000" => S_B6 <= sub_out(14);
WHEN "011010" => S_B6 <= sub_out(7);
WHEN "011100" => S_B6 <= sub_out(5);
WHEN "011110" => S_B6 <= sub_out(11);

WHEN "000001" => S_B6 <= sub_out(10);
WHEN "000011" => S_B6 <= sub_out(15);
WHEN "000101" => S_B6 <= sub_out(4);
WHEN "000111" => S_B6 <= sub_out(2);
WHEN "001001" => S_B6 <= sub_out(7);
WHEN "001011" => S_B6 <= sub_out(12);
WHEN "001101" => S_B6 <= sub_out(9);
WHEN "001111" => S_B6 <= sub_out(5);
WHEN "010001" => S_B6 <= sub_out(6);
WHEN "010011" => S_B6 <= sub_out(1);
WHEN "010101" => S_B6 <= sub_out(13);
WHEN "010111" => S_B6 <= sub_out(14);
WHEN "011001" => S_B6 <= sub_out(0);
WHEN "011011" => S_B6 <= sub_out(11);
WHEN "011101" => S_B6 <= sub_out(3);
WHEN "011111" => S_B6 <= sub_out(8);

WHEN "100000" => S_B6 <= sub_out(9);
WHEN "100010" => S_B6 <= sub_out(14);
WHEN "100100" => S_B6 <= sub_out(15);
WHEN "100110" => S_B6 <= sub_out(5);
WHEN "101000" => S_B6 <= sub_out(2);
WHEN "101010" => S_B6 <= sub_out(8);
WHEN "101100" => S_B6 <= sub_out(12);
WHEN "101110" => S_B6 <= sub_out(3);
WHEN "110000" => S_B6 <= sub_out(7);
WHEN "110010" => S_B6 <= sub_out(0);
WHEN "110100" => S_B6 <= sub_out(4);
WHEN "110110" => S_B6 <= sub_out(10);
WHEN "111000" => S_B6 <= sub_out(1);
WHEN "111010" => S_B6 <= sub_out(13);
WHEN "111100" => S_B6 <= sub_out(11);
WHEN "111110" => S_B6 <= sub_out(6);

WHEN "100001" => S_B6 <= sub_out(4);
WHEN "100011" => S_B6 <= sub_out(3);
WHEN "100101" => S_B6 <= sub_out(2);
WHEN "100111" => S_B6 <= sub_out(12);
WHEN "101001" => S_B6 <= sub_out(9);
WHEN "101011" => S_B6 <= sub_out(5);
WHEN "101101" => S_B6 <= sub_out(15);
WHEN "101111" => S_B6 <= sub_out(10);
WHEN "110001" => S_B6 <= sub_out(11);
WHEN "110011" => S_B6 <= sub_out(14);
WHEN "110101" => S_B6 <= sub_out(1);
WHEN "110111" => S_B6 <= sub_out(7);
```

```
                    WHEN "111001" => S_B6 <= sub_out(6);
                    WHEN "111011" => S_B6 <= sub_out(0);
                    WHEN "111101" => S_B6 <= sub_out(8);
                    WHEN "111111" => S_B6 <= sub_out(13);
              END CASE;
        END PROCESS;
END architecture substitute;
```

S7.vhd

```
--EDNI DEL ROSAL----S7.vhd----------------------
--Thesis Spring 2015----------------------------
--This file performs the substitution of the box-
--S7.-------------------------------------------
--The input is B7 and the output is S_B7---------
library IEEE;
use IEEE.std_logic_1164.all;

ENTITY S7 IS
      PORT(B7: IN std_logic_vector(5 DOWNTO 0);
              S_B7: OUT std_logic_vector(3 DOWNTO 0));
END S7;

ARCHITECTURE substitute OF S7 IS

        TYPE mem IS ARRAY (0 to 15) OF std_logic_vector (3 DOWNTO
0);
        constant sub_out : mem := (
        0 => "0000",
        1 => "0001",
        2 => "0010",
        3 => "0011",
        4 => "0100",
        5 => "0101",
        6 => "0110",
        7 => "0111",
        8 => "1000",
        9 => "1001",
        10 => "1010",
        11 => "1011",
        12 => "1100",
        13 => "1101",
        14 => "1110",
        15 => "1111");

BEGIN
        PROCESS (B7)
        BEGIN
              CASE B7 IS
                    WHEN "000000" => S_B7 <= sub_out(4);
                    WHEN "000010" => S_B7 <= sub_out(11);
                    WHEN "000100" => S_B7 <= sub_out(2);
                    WHEN "000110" => S_B7 <= sub_out(14);
                    WHEN "001000" => S_B7 <= sub_out(15);
```

```
WHEN "001010" => S_B7 <= sub_out(0);
WHEN "001100" => S_B7 <= sub_out(8);
WHEN "001110" => S_B7 <= sub_out(13);
WHEN "010000" => S_B7 <= sub_out(3);
WHEN "010010" => S_B7 <= sub_out(12);
WHEN "010100" => S_B7 <= sub_out(9);
WHEN "010110" => S_B7 <= sub_out(7);
WHEN "011000" => S_B7 <= sub_out(5);
WHEN "011010" => S_B7 <= sub_out(10);
WHEN "011100" => S_B7 <= sub_out(6);
WHEN "011110" => S_B7 <= sub_out(1);

WHEN "000001" => S_B7 <= sub_out(13);
WHEN "000011" => S_B7 <= sub_out(0);
WHEN "000101" => S_B7 <= sub_out(11);
WHEN "000111" => S_B7 <= sub_out(7);
WHEN "001001" => S_B7 <= sub_out(4);
WHEN "001011" => S_B7 <= sub_out(9);
WHEN "001101" => S_B7 <= sub_out(1);
WHEN "001111" => S_B7 <= sub_out(10);
WHEN "010001" => S_B7 <= sub_out(14);
WHEN "010011" => S_B7 <= sub_out(3);
WHEN "010101" => S_B7 <= sub_out(5);
WHEN "010111" => S_B7 <= sub_out(12);
WHEN "011001" => S_B7 <= sub_out(2);
WHEN "011011" => S_B7 <= sub_out(15);
WHEN "011101" => S_B7 <= sub_out(8);
WHEN "011111" => S_B7 <= sub_out(6);

WHEN "100000" => S_B7 <= sub_out(1);
WHEN "100010" => S_B7 <= sub_out(4);
WHEN "100100" => S_B7 <= sub_out(11);
WHEN "100110" => S_B7 <= sub_out(13);
WHEN "101000" => S_B7 <= sub_out(12);
WHEN "101010" => S_B7 <= sub_out(3);
WHEN "101100" => S_B7 <= sub_out(7);
WHEN "101110" => S_B7 <= sub_out(14);
WHEN "110000" => S_B7 <= sub_out(10);
WHEN "110010" => S_B7 <= sub_out(15);
WHEN "110100" => S_B7 <= sub_out(6);
WHEN "110110" => S_B7 <= sub_out(8);
WHEN "111000" => S_B7 <= sub_out(0);
WHEN "111010" => S_B7 <= sub_out(5);
WHEN "111100" => S_B7 <= sub_out(9);
WHEN "111110" => S_B7 <= sub_out(2);

WHEN "100001" => S_B7 <= sub_out(6);
WHEN "100011" => S_B7 <= sub_out(11);
WHEN "100101" => S_B7 <= sub_out(13);
WHEN "100111" => S_B7 <= sub_out(8);
WHEN "101001" => S_B7 <= sub_out(1);
WHEN "101011" => S_B7 <= sub_out(4);
WHEN "101101" => S_B7 <= sub_out(10);
WHEN "101111" => S_B7 <= sub_out(7);
WHEN "110001" => S_B7 <= sub_out(9);
```

```
                    WHEN "110011" => S_B7 <= sub_out(5);
                    WHEN "110101" => S_B7 <= sub_out(0);
                    WHEN "110111" => S_B7 <= sub_out(15);
                    WHEN "111001" => S_B7 <= sub_out(14);
                    WHEN "111011" => S_B7 <= sub_out(2);
                    WHEN "111101" => S_B7 <= sub_out(3);
                    WHEN "111111" => S_B7 <= sub_out(12);
               END CASE;
          END PROCESS;
END architecture substitute;
```

S8.vhd

```
--EDNI DEL ROSAL----S8.vhd-----------------------
--Thesis Spring 2015----------------------------
--This file performs the substitution of the box-
--S8.--------------------------------------------
--The input is B8 and the output is S_B8---------
library IEEE;
use IEEE.std_logic_1164.all;

ENTITY S8 IS
     PORT(B8: IN std_logic_vector(5 DOWNTO 0);
               S_B8: OUT std_logic_vector(3 DOWNTO 0));
END S8;

ARCHITECTURE substitute OF S8 IS

          TYPE mem IS ARRAY (0 to 15) OF std_logic_vector (3 DOWNTO 
0);
          constant sub_out : mem := (
          0 => "0000",
          1 => "0001",
          2 => "0010",
          3 => "0011",
          4 => "0100",
          5 => "0101",
          6 => "0110",
          7 => "0111",
          8 => "1000",
          9 => "1001",
          10 => "1010",
          11 => "1011",
          12 => "1100",
          13 => "1101",
          14 => "1110",
          15 => "1111");

BEGIN
          PROCESS (B8)
          BEGIN
               CASE B8 IS
                    WHEN "000000" => S_B8 <= sub_out(13);
                    WHEN "000010" => S_B8 <= sub_out(2);
```

```
WHEN "000100" => S_B8 <= sub_out(8);
WHEN "000110" => S_B8 <= sub_out(4);
WHEN "001000" => S_B8 <= sub_out(6);
WHEN "001010" => S_B8 <= sub_out(15);
WHEN "001100" => S_B8 <= sub_out(11);
WHEN "001110" => S_B8 <= sub_out(1);
WHEN "010000" => S_B8 <= sub_out(10);
WHEN "010010" => S_B8 <= sub_out(9);
WHEN "010100" => S_B8 <= sub_out(3);
WHEN "010110" => S_B8 <= sub_out(14);
WHEN "011000" => S_B8 <= sub_out(5);
WHEN "011010" => S_B8 <= sub_out(0);
WHEN "011100" => S_B8 <= sub_out(12);
WHEN "011110" => S_B8 <= sub_out(7);

WHEN "000001" => S_B8 <= sub_out(1);
WHEN "000011" => S_B8 <= sub_out(15);
WHEN "000101" => S_B8 <= sub_out(13);
WHEN "000111" => S_B8 <= sub_out(8);
WHEN "001001" => S_B8 <= sub_out(10);
WHEN "001011" => S_B8 <= sub_out(3);
WHEN "001101" => S_B8 <= sub_out(7);
WHEN "001111" => S_B8 <= sub_out(4);
WHEN "010001" => S_B8 <= sub_out(12);
WHEN "010011" => S_B8 <= sub_out(5);
WHEN "010101" => S_B8 <= sub_out(6);
WHEN "010111" => S_B8 <= sub_out(11);
WHEN "011001" => S_B8 <= sub_out(0);
WHEN "011011" => S_B8 <= sub_out(14);
WHEN "011101" => S_B8 <= sub_out(9);
WHEN "011111" => S_B8 <= sub_out(2);

WHEN "100000" => S_B8 <= sub_out(7);
WHEN "100010" => S_B8 <= sub_out(11);
WHEN "100100" => S_B8 <= sub_out(4);
WHEN "100110" => S_B8 <= sub_out(1);
WHEN "101000" => S_B8 <= sub_out(9);
WHEN "101010" => S_B8 <= sub_out(12);
WHEN "101100" => S_B8 <= sub_out(14);
WHEN "101110" => S_B8 <= sub_out(2);
WHEN "110000" => S_B8 <= sub_out(0);
WHEN "110010" => S_B8 <= sub_out(6);
WHEN "110100" => S_B8 <= sub_out(10);
WHEN "110110" => S_B8 <= sub_out(13);
WHEN "111000" => S_B8 <= sub_out(15);
WHEN "111010" => S_B8 <= sub_out(3);
WHEN "111100" => S_B8 <= sub_out(5);
WHEN "111110" => S_B8 <= sub_out(8);

WHEN "100001" => S_B8 <= sub_out(2);
WHEN "100011" => S_B8 <= sub_out(1);
WHEN "100101" => S_B8 <= sub_out(14);
WHEN "100111" => S_B8 <= sub_out(7);
WHEN "101001" => S_B8 <= sub_out(4);
WHEN "101011" => S_B8 <= sub_out(10);
```

```
                    WHEN "101101" => S_B8 <= sub_out(8);
                    WHEN "101111" => S_B8 <= sub_out(13);
                    WHEN "110001" => S_B8 <= sub_out(15);
                    WHEN "110011" => S_B8 <= sub_out(12);
                    WHEN "110101" => S_B8 <= sub_out(9);
                    WHEN "110111" => S_B8 <= sub_out(0);
                    WHEN "111001" => S_B8 <= sub_out(3);
                    WHEN "111011" => S_B8 <= sub_out(5);
                    WHEN "111101" => S_B8 <= sub_out(6);
                    WHEN "111111" => S_B8 <= sub_out(11);
                END CASE;
            END PROCESS;
END architecture substitute;
```

SRL_1.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;

ENTITY SRL_1 IS
      PORT( C,D: IN std_logic_vector(27 DOWNTO 0);
                  CS,DS: OUT std_logic_vector(27 DOWNTO 0));
END SRL_1;

ARCHITECTURE shift OF SRL_1 IS

      SHARED VARIABLE Cint: bit_vector(27 DOWNTO 0):=
to_bitvector(C);
      SHARED VARIABLE Dint: bit_vector(27 DOWNTO 0):=
to_bitvector(D);

BEGIN

      CS <= to_stdlogicvector(Cint rol 1);
      DS <= to_stdlogicvector(Dint rol 1);

END shift;
```

SRL_2.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;

ENTITY SRL_2 IS
      PORT( C,D: IN std_logic_vector(27 DOWNTO 0);
                  CS,DS: OUT std_logic_vector(27 DOWNTO 0));
END SRL_2;

ARCHITECTURE shift OF SRL_2 IS

      SHARED VARIABLE Cint: bit_vector(27 DOWNTO 0) :=
to_bitvector(C);
```

```
      SHARED VARIABLE Dint: bit_vector(27 DOWNTO 0) :=
to_bitvector(D);

BEGIN

      CS <= to_stdlogicvector(Cint rol 2);
      DS <= to_stdlogicvector(Dint rol 2);

END shift;
```

SRR_1.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;

ENTITY SRR_1 IS
      PORT( C,D: IN std_logic_vector(27 DOWNTO 0);
                  CS,DS: OUT std_logic_vector(27 DOWNTO 0));
END SRR_1;

ARCHITECTURE shift OF SRR_1 IS

      SHARED VARIABLE Cint: bit_vector(27 DOWNTO 0):=
to_bitvector(C);
      SHARED VARIABLE Dint: bit_vector(27 DOWNTO 0):=
to_bitvector(D);

BEGIN

      CS <= to_stdlogicvector(Cint ror 1);
      DS <= to_stdlogicvector(Dint ror 1);

END shift;
```

SRR_2.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;

ENTITY SRR_2 IS
      PORT( C,D: IN std_logic_vector(27 DOWNTO 0);
                  CS,DS: OUT std_logic_vector(27 DOWNTO 0));
END SRR_2;

ARCHITECTURE shift OF SRR_2 IS

      SHARED VARIABLE Cint: bit_vector(27 DOWNTO 0) :=
to_bitvector(C);
      SHARED VARIABLE Dint: bit_vector(27 DOWNTO 0) :=
to_bitvector(D);

BEGIN
```

```
      CS <= to_stdlogicvector(Cint ror 2);
      DS <= to_stdlogicvector(Dint ror 2);

END shift;
```

TDES_decrypt.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;

ENTITY TDES_decrypt IS
PORT( cipher_data: IN std_logic_vector(63 DOWNTO 0);
            key1,key2,key3: IN std_logic_vector(63 DOWNTO 0);

            key_out: OUT std_logic_vector(63 DOWNTO 0);
            cipher_out: OUT std_logic_vector(63 DOWNTO 0);
            data: OUT std_logic_vector(63 DOWNTO 0));
END TDES_decrypt;

ARCHITECTURE encrypt OF TDES_decrypt IS

      SIGNAL data_1,data_2,data_3: std_logic_vector(63 DOWNTO 0);

      SIGNAL cipher_out1,cipher_out2,cipher_out3:
std_logic_vector(63 DOWNTO 0);
      SIGNAL key_out1,key_out2,key_out3: std_logic_vector(63 DOWNTO
0);

      COMPONENT DES_encrypt
            PORT( data: IN std_logic_vector(63 DOWNTO 0);
                        K: IN std_logic_vector(63 DOWNTO 0);

                        data_out: OUT std_logic_vector(63 DOWNTO 0);
                        K_out: OUT std_logic_vector(63 DOWNTO 0);
                        cipher_data: OUT std_logic_vector(63 DOWNTO
0));
      END COMPONENT;

      COMPONENT DES_decrypt
            PORT( cipher_data: IN std_logic_vector(63 DOWNTO 0);
                        K: IN std_logic_vector(63 DOWNTO 0);

                        cipher_out: OUT std_logic_vector(63 DOWNTO
0);
                        K_out: OUT std_logic_vector(63 DOWNTO 0);
                        data: OUT std_logic_vector(63 DOWNTO 0));
      END COMPONENT;

BEGIN

      DES_1 : DES_decrypt PORT
MAP(cipher_data,key3,cipher_out3,key_out3,data_3);
      DES_2 : DES_encrypt PORT
MAP(data_3,key2,cipher_out2,key_out2,data_2);
```

```
      DES_3 : DES_decrypt PORT
MAP(data_2,key1,cipher_out1,key_out1,data_1);

                  data <= data_1;
                  cipher_out <= cipher_data;
                  key_out <= key1;

END encrypt;
```

TDES_decrypt_P.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;

ENTITY TDES_decrypt_P IS
PORT(clk: IN std_logic;
            cipher_data: IN std_logic_vector(63 DOWNTO 0);
            key1,key2,key3: IN std_logic_vector(63 DOWNTO 0);

            key_out: OUT std_logic_vector(63 DOWNTO 0);
            cipher_out: OUT std_logic_vector(63 DOWNTO 0);
            data: OUT std_logic_vector(63 DOWNTO 0));
END TDES_decrypt_P;

ARCHITECTURE encrypt OF TDES_decrypt_P IS

      SIGNAL data_1,data_2,data_3: std_logic_vector(63 DOWNTO 0);

      SIGNAL key1_int,key2_int,key3_int: std_logic_vector(63 DOWNTO
0);

      SIGNAL cipher_out1,cipher_out2,cipher_out3:
std_logic_vector(63 DOWNTO 0);
      SIGNAL key_out1,key_out2,key_out3,key_buffer_1:
std_logic_vector(63 DOWNTO 0);

      COMPONENT key_bank
            PORT( clk: IN std_logic;
                        key1_in,key2_in,key3_in: IN
std_logic_vector(63 DOWNTO 0);
                        key1_out,key2_out,key3_out: OUT
std_logic_vector(63 DOWNTO 0));
      END COMPONENT;

      COMPONENT DES_encrypt_P
            PORT( clk: IN std_logic;
                        data: IN std_logic_vector(63 DOWNTO 0);
                        data_in: IN std_logic_vector(63 DOWNTO 0);
                        K: IN std_logic_vector(63 DOWNTO 0);

                        data_out: OUT std_logic_vector(63 DOWNTO 0);
                        K_out: OUT std_logic_vector(63 DOWNTO 0);
                        cipher_data: OUT std_logic_vector(63 DOWNTO
0));
```

```
      END COMPONENT;

      COMPONENT DES_decrypt_P
            PORT( clk: IN std_logic;
                        cipher_data: IN std_logic_vector(63 DOWNTO
0);
                        cipher_in: IN std_logic_vector(63 DOWNTO 0);
                        K: IN std_logic_vector(63 DOWNTO 0);

                        cipher_out: OUT std_logic_vector(63 DOWNTO
0);
                        K_out: OUT std_logic_vector(63 DOWNTO 0);
                        data: OUT std_logic_vector(63 DOWNTO 0));
      END COMPONENT;


BEGIN

      KEYS_BANK : key_bank PORT
MAP(clk,key3,key2,key1,key3_int,key2_int,key1_int);
      DES_1 : DES_decrypt_P PORT
MAP(clk,cipher_data,cipher_data,key3_int,cipher_out3,key_out3,data_3
);
      DES_2 : DES_encrypt_P PORT
MAP(clk,data_3,cipher_out3,key2_int,cipher_out2,key_out2,data_2);
      DES_3 : DES_decrypt_P PORT
MAP(clk,data_2,cipher_out2,key1_int,cipher_out1,key_out1,data_1);

      Process(clk)
      BEGIN
            IF(falling_edge(clk)) THEN
                  key_buffer_1 <= key_out1;
            END IF;
      END PROCESS;

                  data <= data_1;
                  cipher_out <= cipher_out1;
                  key_out <= key_buffer_1;

END encrypt;
```

TDES_encrypt.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;

ENTITY TDES_encrypt IS
PORT( data: IN std_logic_vector(63 DOWNTO 0);
            key1,key2,key3: IN std_logic_vector(63 DOWNTO 0);

            key_out: OUT std_logic_vector(63 DOWNTO 0);
            data_out: OUT std_logic_vector(63 DOWNTO 0);
            cipher_data: OUT std_logic_vector(63 DOWNTO 0));
END TDES_encrypt;
```

```
ARCHITECTURE encrypt OF TDES_encrypt IS

      SIGNAL cipher_1,cipher_2,cipher_3: std_logic_vector(63 DOWNTO
0);

      SIGNAL data_out1,data_out2,data_out3: std_logic_vector(63
DOWNTO 0);
      SIGNAL key_out1,key_out2,key_out3: std_logic_vector(63 DOWNTO
0);

      COMPONENT DES_encrypt
            PORT( data: IN std_logic_vector(63 DOWNTO 0);
                        K: IN std_logic_vector(63 DOWNTO 0);

                        data_out: OUT std_logic_vector(63 DOWNTO 0);
                        K_out: OUT std_logic_vector(63 DOWNTO 0);
                        cipher_data: OUT std_logic_vector(63 DOWNTO
0));
      END COMPONENT;

      COMPONENT DES_decrypt
            PORT( cipher_data: IN std_logic_vector(63 DOWNTO 0);
                        K: IN std_logic_vector(63 DOWNTO 0);

                        cipher_out: OUT std_logic_vector(63 DOWNTO
0);
                        K_out: OUT std_logic_vector(63 DOWNTO 0);
                        data: OUT std_logic_vector(63 DOWNTO 0));
      END COMPONENT;


BEGIN

      DES_1 : DES_encrypt PORT
MAP(data,key1,data_out1,key_out1,cipher_1);
      DES_2 : DES_decrypt PORT
MAP(cipher_1,key2,data_out2,key_out2,cipher_2);
      DES_3 : DES_encrypt PORT
MAP(cipher_2,key3,data_out3,key_out3,cipher_3);


                  cipher_data <= cipher_3;
                  data_out <= data;
                  key_out <= key1;

END encrypt;
```

TDES_encrypt_P.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;

ENTITY TDES_encrypt_P IS
```

```
PORT(clk: IN std_logic;
           data: IN std_logic_vector(63 DOWNTO 0);
           key1,key2,key3: IN std_logic_vector(63 DOWNTO 0);

           key_out: OUT std_logic_vector(63 DOWNTO 0);
           data_out: OUT std_logic_vector(63 DOWNTO 0);
           cipher_data: OUT std_logic_vector(63 DOWNTO 0));
END TDES_encrypt_P;

ARCHITECTURE encrypt OF TDES_encrypt_P IS

      SIGNAL cipher_1,cipher_2,cipher_3: std_logic_vector(63 DOWNTO
0);

      SIGNAL key1_int,key2_int,key3_int,key_buffer_1:
std_logic_vector(63 DOWNTO 0);

      SIGNAL data_out1,data_out2,data_out3: std_logic_vector(63
DOWNTO 0);
      SIGNAL key_out1,key_out2,key_out3: std_logic_vector(63 DOWNTO
0);

      COMPONENT key_bank
            PORT( clk: IN std_logic;
                        key1_in,key2_in,key3_in: IN
std_logic_vector(63 DOWNTO 0);
                        key1_out,key2_out,key3_out: OUT
std_logic_vector(63 DOWNTO 0));
      END COMPONENT;
      COMPONENT DES_encrypt_P
            PORT( clk: IN std_logic;
                        data: IN std_logic_vector(63 DOWNTO 0);
                        data_in: IN std_logic_vector(63 DOWNTO 0);
                        K: IN std_logic_vector(63 DOWNTO 0);

                        data_out: OUT std_logic_vector(63 DOWNTO 0);
                        K_out: OUT std_logic_vector(63 DOWNTO 0);
                        cipher_data: OUT std_logic_vector(63 DOWNTO
0));
      END COMPONENT;

      COMPONENT DES_decrypt_P
            PORT( clk: IN std_logic;
                        cipher_data: IN std_logic_vector(63 DOWNTO
0);
                        cipher_in: IN std_logic_vector(63 DOWNTO 0);
                        K: IN std_logic_vector(63 DOWNTO 0);

                        cipher_out: OUT std_logic_vector(63 DOWNTO
0);
                        K_out: OUT std_logic_vector(63 DOWNTO 0);
                        data: OUT std_logic_vector(63 DOWNTO 0));
      END COMPONENT;
```

```
BEGIN

      KEYS_BANK : key_bank PORT
MAP(clk,key1,key2,key3,key1_int,key2_int,key3_int);
      DES_1 : DES_encrypt_P PORT
MAP(clk,data,data,key1_int,data_out1,key_out1,cipher_1);
      DES_2 : DES_decrypt_P PORT
MAP(clk,cipher_1,data_out1,key2_int,data_out2,key_out2,cipher_2);
      DES_3 : DES_encrypt_P PORT
MAP(clk,cipher_2,data_out2,key3_int,data_out3,key_out3,cipher_3);

      Process(clk)
      BEGIN
            IF(falling_edge(clk)) THEN
                              key_buffer_1 <= key_out3;
            END IF;
      END PROCESS;

                  cipher_data <= cipher_3;
                  data_out <= data_out3;
                  key_out <= key_buffer_1;



END encrypt;
```

BIOGRAPHICAL SKETCH

Edni Del Rosal was born on November 28, 1988. His educational background consists of a BS of Engineering and Computer Science in Electrical Engineering from the University Of Texas Pan-American on December 2011. His industrial experience includes IBM and AMD, where he worked as a CPU/GPU Validation Intern, Power Integrity Intern and System Test Intern. He has also earned his MS in Electrical Engineering from the University Of Rio Grande Valley in December 2017. The focus of his thesis work was improving the performance of the Triple Data Encryption Standard scheme in reconfigurable logic. His permanent address is:

10,000 Aconcagua St.

Edinburg, TX 78541

Email: del.rosal.ed@gmail.com

His publication achieved during his Masters is:

- Del Rosal, E. and Kumar S. "A Fast FPGA Implementation For Triple DES Encryption Scheme." Journal of Circuit and Systems. 2017.