

University of Texas Rio Grande Valley

ScholarWorks @ UTRGV

Theses and Dissertations - UTB/UTPA

8-2009

Asynchronous designs on FPGA with soft error tolerance for security algorithms

Deepya Reddy Nalubolu

University of Texas-Pan American

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



Part of the [Electrical and Computer Engineering Commons](#)

Recommended Citation

Nalubolu, Deepya Reddy, "Asynchronous designs on FPGA with soft error tolerance for security algorithms" (2009). *Theses and Dissertations - UTB/UTPA*. 1034.

https://scholarworks.utrgv.edu/leg_etd/1034

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

ASYNCHRONOUS DESIGNS ON FPGA WITH
SOFT ERROR TOLERANCE FOR
SECURITY ALGORITHMS

A Thesis

by

DEEPIA REDDY NALUBOLU

Submitted to the Graduate School of the
University of Texas-Pan American
In partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

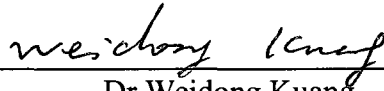
August 2009

Major Subject: Electrical Engineering

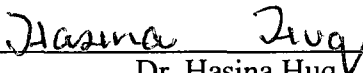
ASYNCHRONOUS DESIGNS ON FPGA WITH
SOFT ERROR TOLERANCE FOR
SECURITY ALGORITHMS

A Thesis
by
DEEPIA REDDY NALUBOLU

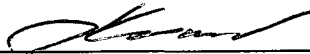
Approved as to style and content by:



Dr. Weidong Kuang
Chair of Committee



Dr. Hasina Huq
Committee Member



Dr. Sanjeev Kumar
Committee Member

August 2009

ABSTRACT

Nalubolu, Deepya Reddy., Asynchronous Designs on FPGA with Soft Error Tolerance for Security Algorithms. Master of Science (MS), August, 2009, 140 pp., references, 52 titles.

Asynchronous methodologies, such as Null Convention Logic (NCL), have tremendous potential in implementing digital logic. It is essential to design complex asynchronous circuits using commercial Electronic Design Automation (EDA) tools. The main focus of this thesis is to design NCL circuits using VHDL and implementing them on FPGAs. The major contributions of this thesis include:

- 1) Developing a methodology of designing NCL circuits with VHDL and applying it successfully to all practical designs in this thesis.
- 2) As an example, the NCL circuit for DES (Data Encryption Standard) algorithm has been designed and simulated using VHDL and the implementation issues on various FPGAs (Xilinx and Altera) have been investigated. Modification of the design has been done to minimize the amount of logic used.
- 3) An effective soft error tolerant scheme for asynchronous circuits on FPGAs is proposed, and successfully verified through software simulation and hardware implementation by introducing it into a DES round.

This thesis provides a starting point for further investigation of NCL circuits, in terms of VHDL modeling, FPGA implementations, and soft error tolerance.

DEDICATION

I am extremely overwhelmed to dedicate this thesis to my only four valuables. My mother, Sowjanya Nalubolu, my father, VenuGopal Reddy Nalubolu, my brother Vinay Reddy Nalubolu and my friend Vijay Ashwanth, who have been with me through thick and thin. I am made out of their discipline and commitment.

My parents always taught me the importance of education in one's life without whom I would have never anticipated for higher education abroad. Their kind guidance has helped me choose the right path in life. My brother and my friend were always encouraging and helped a lot in successfully finishing my thesis.

ACKNOWLEDGMENTS

The support for the project is a grant from the Department of Defense (DOD). I would like to thank the DOD for believing in the potential of students not only from highly reputed schools but also in schools for minority students. Without the backing up of the DOD, the ideas might not have come into existence. The Department of Electrical Engineering at UTPA would always be indebted to Government and private organizations for supporting its students. I would also like to thank Dr.Kuang, the grant principle investigator (PI) and thesis committee Chair. He is a kind and wonderful teacher who has given enough freedom to think and successfully implement things. Last but not the least I would thank the UTPA for giving me an excellent opportunity to study here without which I would not have met wonderful teachers and friends. I also thank the almighty for this being his will. I would like to convey my special thanks to Dr.Scott smith, Associate Professor, Department of Electrical Engineering, University of Arkansas for providing the basic material, the NCL library, based on which the thesis is constructed upon.

TABLE OF CONTENTS

	Page
ABSTRACT	iii
DEDICATION	iv
ACKNOWLEDGEMENT	v
TABLE OF CONTENTS	vi
LIST OF TABLES	x
LIST OF FIGURES	xi
CHAPTER I. INTRODUCTION	1
1.1 Data Encryption Standard Algorithm.....	2
1.2 Implementation of DES on FPGA	5
1.3 Asynchronous Design Methodology	7
1.4 Soft Errors and Digital Circuits	8
1.5 Thesis Objectives.....	10
CHAPTER II. BACKGROUND WORK ON NULL CONVENTION LOGIC	12
2.1 Completion Criteria	13
2.2 Threshold Gates with Hysteresis	15
2.3 NCL Pipeline	16
2.3.1 NCL Register.....	18
2.3.2 Completion Detection Circuitry	19

2.3.3 NCL Combinational Circuit	20
2.4 NCL Circuits Using CMOS Transistors	21
CHAPTER III. NCL CIRCUIT DESIGN WITH VHDL.....	24
3.1 FPGA	24
3.2 Design Flow used in the Thesis	26
3.3 NCL Circuits in VHDL	28
3.3.1 Data Type Called dual_rail_logic	28
3.3.2 Threshold Gates with Hysteresis in VHDL	29
3.3.3 NCL Dual-rail Registers & Completion Detection Circuits in VHDL.....	30
3.3.4 Constructing Computational Blocks	32
3.4 Simulation of a Simple NCL Pipeline	33
CHAPTER IV. ASYNCHRONOUS DES ALGORITHM USING NCL	38
4.1 Initial Register	39
4.2 Initial Round	40
4.3 NCL Registers1-15	41
4.4 Rounds1-14	42
4.5 Round 15	43
4.6 NCL Register17	44
4.7 Final Round and Final Register	44
4.8 NCL DES Design on FPGAs	44
4.8.1 Xilinx Device	45
4.8.2 Altera Devices.....	45
4.9 Improvements in Asynchronous DES Design	47

4.9.1 Design Modification to Utilize Internal RAM Elements.....	47
4.9.2 Resource Utilization with RAM elements	49
4.9.3 Resource Comparison Between Synchronous and Asynchronous Designs..	50
CHAPTER V. SOFT ERROR AND NCL CIRCUITS	52
5.1 SEUs in NCL	52
5.1.1 SEUS in Semiconductor Circuits	53
5.1.2 Generation and Propagation of Soft Errors in NCL	54
5.2 Study of NCL Pipeline for an SEU	55
5.3 Tackling Soft Errors Using NCL Methodology	57
5.3.1 Soft Error Mitigation and Correction	57
5.3.2 Soft Error Tolerant Schemes in NCL	59
CHAPTER VI. SOFT ERROR TOLERANT DESIGN USING FPGA	61
6.1 Introduction	61
6.2 Soft Error Tolerant Design	62
6.2.1 Circuit for Soft Error Tolerance	62
6.2.2 Case Study: NCL Full-adder as Computational Block	65
6.2.3 Generating Inputs and Strike	68
6.2.4 Simulation Results	72
6.3 Experiments on FPGA Device	78
6.3.1 Experimental Set up	79
6.3.2 FPGA Board	80
6.3.3 Results	81
CHAPTER VII. SOFT ERROR TOLERANT ASYNCHRONOUS DES DESIGN ...	86

7.1 Asynchronous DES with Soft Error Tolerance	86
7.2 Results Obtained	88
CHAPTER VIII. CONCLUSION AND FUTURE WORK	92
REFERENCES	94
APPENDIX A. DES ALGORITHM AND DESIGN UNITS IN VHDL	98
APPENDIX B. NCL LIBRARY	106
APPENDIX C. VHDL FILES USED FOR THE THESIS	116
BIOGRAPHICAL SKETCH.....	140

LIST OF TABLES

Table	Page
2.1: Dual-rail Encoding	13
2.2: Twenty-Seven Fundamental NCL Gates and their Boolean Functions	16
4.1: Devices and Companies	44
4.2: Virtex5 Resources	45
4.3: Resources used by DES Algorithm (dual-rail logic) on Xilinx Device	45
4.4: Altera Device Resources	46
4.5: Resources used by DES Algorithm (dual-rail logic) on Altera Device	46
4.6: Resources used by DES Algorithm with ROM on Altera Devices	50
4.7: FPGA Resources used by Different DES Designs	51
6.1: Truth Table of a 1-bit Full Adder with Different States	66

LIST OF FIGURES

Figure	page
1.1: Step by step procedure of DES	3
1.2: Metaphor of the bucket brigade a) synchronous b) asynchronous	7
2.1: Weak Conditions for NCL Completeness of Input	14
2.2: Different threshold gates	15
2.3: Basic NCL pipeline structure	17
2.4: 1-bit NCL register	18
2.5: n-bit completion detection circuitry	20
2.6: NCL implementation of a) Inverter b) Exor gate c) Full-adder	21
2.7: General Structure of a Static Gate	22
2.8: Go to NULL and hold DATA transistor blocks	22
2.9: Static th_{23} gate	23
3.1: Basic resources of an FPGA	25
3.2: FPGA design flow	25
3.3: Design flow	27
3.4: dual_rail_logic data-type	29
3.5: Behavioral description of th_{22} in VHDL	29
3.6: 1-bit NCL register in VHDL	31
3.7: Creating n-bit register from 1-bit register	31

3.8: VHDL code for a) Exor gate b) Full-adder	32
3.9: NCL pipeline with exor gate	33
3.10: VHDL code for the NCL pipeline with exor gate	34
3.11: a) initreg.vhd b) exor_dl.vhd c) finalreg.vhd	35
3.12: Quartus II software window	36
3.13: Simulation results of exor.vhd	37
4.1: DES pipeline in NCL dual-rail logic	39
4.2: Initial Register	39
4.3: Initial round in the DES pipeline	40
4.4: S-box inputs and outputs	41
4.5: The inside view of NCL register1-15	42
4.6: Internal structure of rounds1-14	43
4.7: NCL register17	43
4.8: Asynchronous DES round with S-boxes as RAM elements	47
4.9: S-box as ROM	48
5.1: Mechanism of soft errors in semiconductor circuits	53
5.2: SEU generation in th_{23} gate	55
5.3: Different outputs during different strike timings	57
6.1: Soft error tolerant design	62
6.2: Three scenarios to be tested for a full-adder	67
6.3: 1-bit dual-rail full-adder with strike	69
6.4: Modified full-adder incorporating strike	70
6.5: VHDL code for inducing a particle strike	70

6.6: Inputs and strike generator	71
6.7: Simulation waveform of inputs & strike generator	72
6.8: Simulation results without a strike	73
6.9: Simulation results during a strike at third clock cycle	73
6.10: Simulation results when strike is placed at the first clock cycle	74
6.11: Simulation results when strike is placed at the second clock cycle	74
6.12: Simulation results when strike is placed at the third clock cycle	75
6.13: Simulation results when strike is placed at the fourth clock cycle	76
6.14: Simulation results when strike is placed at the fifth clock cycle	76
6.15: Simulation results when strike is placed at the sixth clock cycle	77
6.16: Experimental set up	78
6.17: Laboratory experimental set up	79
6.18: The DE2 board	80
6.19: Actual results when strike is placed at the first clock cycle	82
6.20: Actual results when strike is placed at the second clock cycle	83
6.21: Actual results when strike is placed at the third clock cycle	83
6.22: Actual results when strike is placed at the fourth clock cycle	84
6.23: Actual results when strike is placed at the fifth clock cycle	84
6.24: Actual results when strike is placed at the sixth clock cycle	85
7.1: Asynchronous DES with embedded soft error tolerant circuit	87
7.2: Simulation result for strike at first clock cycle	88
7.3: Actual result for strike at first clock cycle	89
7.4: Simulation result for strike at second clock cycle	89

7.5: Actual result for strike at second clock cycle	89
7.6: Simulation result for strike at third clock cycle	91
7.7: Actual result for strike at third clock cycle	91

CHAPTER I

INTRODUCTION

Data Encryption Standard (DES) algorithm, a most widely used algorithm in the world is a cipher (a method for encrypting information) which was selected as an official Federal Information Processing Standard (FIPS) for the United States in 1976[1]. DES provided the basics for understanding block ciphers and their cryptanalysis. DES algorithm operates on a 64 bit plaintext using 56 bit key (actually 64 bit key, every 8th bit of the key is not used) thereby generating a 64 bit ciphertext which is the encrypted data.

A Field Programmable Gate Array (FPGA) is a semiconductor device that can be programmed or configured any number of times using a schematic design or a source code in HDL (hardware description language) that describe the user's hardware design [2]. FPGAs can be configured with dense logic and have very high logic capacity. Cryptographic algorithms like DES could be made to accommodate into the logic cells and the memory units of the FPGA since it also provides the advantage of updating and reprogramming any number of times in the future.

Asynchronous circuits [3] are digital circuits which operate without a clock unlike synchronous circuits whose operation is solely dependent on a clock signal. Asynchronous circuits use handshaking protocols to communicate between modules or

parts of the circuits for the operations to be done in sequence. As these circuits have additional handshaking protocols other than logic, the entire circuit design will be undoubtedly large compared to their synchronous counterparts. Hence, the challenge to optimize these designs on the FPGA logic is inevitable.

Soft errors, also called transient faults or single-event upsets (SEUs) are caused due to electrical noise or external radiation rather than design or manufacturing defects. As CMOS device sizes decrease, they are more easily affected by the low energy particles resulting from collisions between cosmic rays and particles in the atmosphere, potentially leading to a much higher rate of soft errors [4].

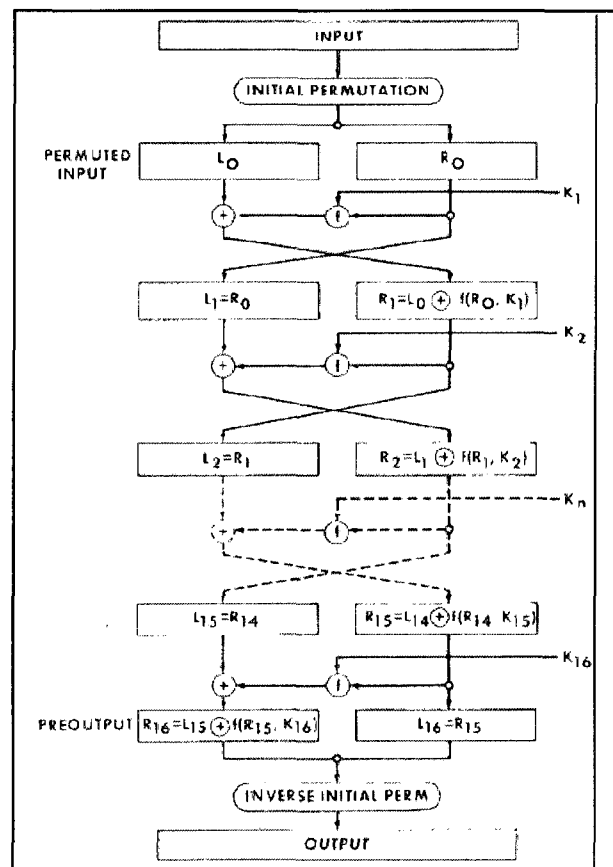
1.1 Data Encryption Standard Algorithm

DES is a block cipher - meaning it operates on plaintext blocks of a given size (64-bits) and returns ciphertext blocks of the same size. DES algorithm has been the foundation for many of the future cryptographic algorithms. Since its creation, DES was considered as a basic cryptographic algorithm by the academia and has been researched to crack the algorithm, create similar algorithms which are robust. The algorithm is believed to be practically secure in the form of Triple DES (TDEA) [1].

DES algorithm takes plain text as a block of 64-bits and generates a cipher using a 64-bit key. In general, cryptography is used to protect data while it is being communicated between two points or while it is stored in a medium vulnerable to physical theft. Communication security provides protection to data by enciphering it at the transmitting point and deciphering it at the receiving point. Enciphering is the process of generating a cipher (encrypted data) using the data and the key while deciphering is

just the opposite which is getting the original data using the cipher and the key. This thesis focuses only on enciphering procedure since deciphering is just the opposite.

File security provides protection to data by enciphering it when it is recorded on a stored medium and deciphering it when it is read back from the stored medium. In the first case, the key must be available at the transmitter and receiver simultaneously during communication. In the second, case the key must be maintained and accessible for the duration of the storage period. Figure 1 shown below pictorially explains the step by step procedure of DES.



and an example is illustrated for better understanding the algorithm. As mentioned previously DES algorithm operates on 64-bit plain text using 64-bit key to generate a 64-bit cipher. The 64-bit key is used to generate 16 sub-keys which are used at each step. Firstly an initial permutation PC-1 is done on the 64-bits which simultaneously eliminate every 8th bit of the key. Then this 56-bit permuted key is divided into two 28-bits, the left half denoted as C_0 and the right half as D_0 . $C_1, D_1, C_2, D_2, \dots, C_{16}, D_{16}$ are generated from previous C_i, D_i based on a left shift table (refer to Appendix A). For example, C_1, D_1 are generated by left shifting C_0 and D_0 respectively using the number of left shifts mentioned in the left shift table. Then each pair is concatenated and permuted using a permutation table PC-2, each generating a single sub-key K_1, K_2, \dots, K_{16} respectively.

The 64-bit data is applied with an initial permutation IP and the result is divided into two 32-bits, the left half as L_0 and the right half as R_0 . These L_0 and R_0 serve as the initial data from which further halves ($L_1, R_1, \dots, L_{16}, R_{16}$) are generated. The final cipher is generated from flipping and permuting L_{16} and R_{16} . The two equations which are used to generate L_i and R_i are $L_i = R_{i-1}$ and $R_i = L_{i-1} + f(R_{i-1}, K_i)$ which means next L is the present R and next R is obtained from present R and K. For example L_1 has the value of R_0 and R_1 is obtained from R_0 and K_1 , which is the first sub-key, after a series of steps mentioned below.

After as initial permutation, the 64-bit data is divided into 32-bits each as L_0 and R_0 . Now, $L_1 = R_0$ and in order to obtain R_1 , R_0 which is 32-bit is expanded using an expansion table (refer to Appendix A) to 48-bits. The resulting 48-bits is XORed with K_1 generating a 48-bit output which is divided into 8 group of 6-bits viz., B_1, B_2, \dots, B_8 . Each

6-bit value is mapped to 8, respective S-boxes S_1, \dots, S_8 generating a 4-bit value. The mapping is done by considering the first and the sixth bit of say B_1 , and the 2-bits are collectively considered to be row number and the remaining four bits from 2nd bit to 4th bit are collectively considered as column number. By using the row and the column numbers, a 4-bit value is taken from the S-box, S_1 for B_1 in this case. So, combining all the 8, 4-bit outputs of 8, single S-boxes, we get a 32 bit output. This 32-bit output is subjected to permutation by a table P and then exored with K_1 and is denoted as f . R_1 is obtained from f and L_0 by exoring both the 32-bits.

The above procedure represents one round of calculations. The entire DES algorithm has 16 such rounds using the two important equations mentioned above, starting from L_0, R_0 until the generation of L_{16}, R_{16} . L_{16} which represents the left 32-bits and R_{16} which represents the right 32-bits are flipped and subjected to final permutation IP^{-1} and the resulting output is the cipher text. As we know the entire procedure of DES, let's move on to investigate how DES is implemented on hardware.

1.2 Implementations of DES on FPGA

Since the creation of DES there have been many implementations of the algorithm. The most common implementation is the software implementation. Software implementation provides a great flexibility but it is slow for those applications where execution time is a crucial factor. Another implementation of DES is the ASIC (Application Specific Integrated Circuit) implementation. Execution speed is a major advantage of this method. ASICs have higher cost of implementation since they have to follow expensive and time-consuming fabrication process. Reconfigurable logic like the FPGA has the dual advantage of the software implementation and the ASIC

implementation. FPGAs are faster and have high flexibility and could be reconfigured or reprogrammed any number of times. They have the advantage to be programmed with different designs or modifications of the same design hence reducing the time and cost of implementing a design.

There are different synchronous DES designs which have been implemented on FPGAs [6-8]. In some of the designs, 16 sub-keys are pre-computed and multiplexers are used to select each sub-key. In comparison with the lookup table approach to implement the S-boxes, the direct implementation of Boolean functions increased the speed of processing, saved on the number of gates and was more suitable for FPGA architectures [6]. A design using pipelining approach and ROM elements present in the FPGA could achieve 1Gbps. DES with 16 pipelines gave the maximum speed but occupies more area in the FPGA [7]. The overall security of the design was improved by using different keys every clock cycle [8]. The fastest synchronous DES implementation on FPGA runs a data rate of 10.7 Gbps, utilizes Jbits on a FPGA. Jbits provides a Java-based Application Programming Interface (API) for the run-time creation and modification of the configuration bit-stream. This design is not a single-chip implementation of the full DES algorithm since the key schedule is computed in software. Also, it can only accommodate one key per data transfer session [9].

Here are some of the conclusions made when a DES design was implemented on a FPGA. S-Boxes should be implemented in ROM elements for maximum performance. Bigger chip results in a slower design. A migration from speed grades -4 to -3, -3 to -2 and -2 to -1 result in up to 20% higher performance. The Xilinx device is faster than the

Altera device even if the former is bigger than the later and had a slower speed grade (-4) than the second which had the faster speed grade (-1) [7].

1.3 Asynchronous Design Methodology

Logic design methods can be broadly classified into two categories, namely synchronous and asynchronous methodologies. Synchronous method is the commonly used method where the data is available to all the components of the design at the rising edge or at the falling edge of the clock. The clock signal is the dictator for the entire circuit's operation. In the asynchronous methodology there is mutual understanding between the neighboring units of the design. Sutherland [10] used a metaphor of the bucket brigade to explain the difference between the two methodologies.

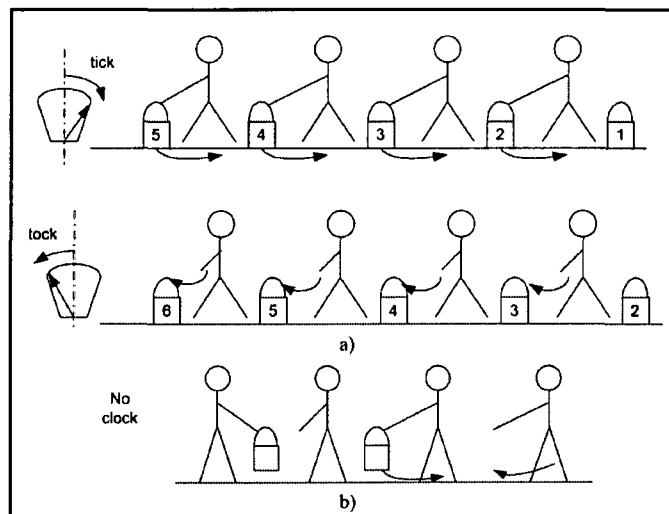


Figure 1.2 Metaphor of the bucket brigade a) Synchronous b) Asynchronous

Asynchronous circuits have several potential advantages over their synchronous counterparts. Clock skew is a matter of concern in synchronous logic but doesn't affect asynchronous logic since the methodology doesn't use a clock at all. Asynchronous circuits have average case performance unlike synchronous circuits whose clock is set according to worst case delay due to which asynchronous designs are faster than the

synchronous designs. Another advantage of asynchronous designs is that these have high energy efficiencies. Asynchronous circuits eliminate glitches and have transitions on the computation path where and when involved in the current computation thereby decreasing energy consumption. These designs also have robust external input handling capability since there is no clock which determines the inputs at a particular time. Hence, asynchronous circuits can accommodate inputs effectively. Asynchronous circuits have better noise and EM properties when used in mixed-signal circuits.

Asynchronous circuits can even utilize a synchronous wrapper, such that the end user does not know that the internal circuitry is actually asynchronous in nature. International Technology Roadmap for Semiconductors (ITRS) envisions a likely shift from synchronous to asynchronous design styles in order to increase circuit robustness, decrease power, and alleviate many clock-related issues. ITRS also states that asynchronous circuits will account for 19% of chip area within the next 5 years, and 30% of chip area within the next 10 years [11].

1.4 Soft Errors and Digital Circuits

Soft errors are random nonrecurring single bit errors in memory devices, including SRAM, DRAM, registers, and latches. Alpha particles from decaying uranium and thorium impurities in integrated circuit interconnect and packaging is a major source of soft errors at sea level and the neutron flux from cosmic rays is the major cause at higher altitudes [12]. The intensity of these soft errors depends on the energy of the incoming particle, location of the device, the geometry of the impact, the location of the strike, and the design of the logic circuit.

For applications in medical electronic devices this soft error mechanism may be extremely important. Neutrons are produced during high energy cancer radiation therapy using photon beam energies above 10 MV. These neutrons are moderated as they are scattered from the equipment and walls in the treatment room resulting in a thermal neutron flux that is about 40×10^6 higher than the normal environmental neutron flux. This high thermal neutron flux will generally result in a very high rate of soft errors and consequent circuit upset [13-14].

The incoming particle must be strong enough to induce a charge that can change the voltage value. The minimum charge required to change the logic level is called critical charge denoted as Q_{crit} . As device sizes are scaling down, the devices are prone to soft errors for lesser Q_{crit} .

The location of the device also influences the number of particle strikes and their effects. For example, the effect of soft error is worse in places farther to equator compared to places on the equator and worse in mountain tops rather than at sea level due to the density of cosmic rays.

Both synchronous and asynchronous circuits are affected by soft errors. In the case of synchronous circuits, if a soft error occurs during a clock tick, a wrong output is generated. Asynchronous circuits with dual-rail inputs and outputs have better advantages over synchronous circuits to detect and correct soft errors. This thesis focuses only on the soft errors generated in asynchronous combinational logic. Soft error rate (SER) is the rate at which a device or system encounters or is predicted to encounter soft errors. It is typically expressed as either number of failures-in-time (FIT), or mean-time-between-failures (MTBF). The unit adopted for quantifying failures in time is called FIT,

equivalent to 1 error per billion hours of device operation. MTBF is usually given in years of device operation. To put it in perspective, 1 year MTBF is equal to approximately 114,077 FIT [15].

Soft errors were first discovered in memory elements like DRAMs in 1970s. Since then DRAMs were the focus for soft errors also because it occupies most of the susceptible surface area. DRAMs of 256 Kb with 1980s technology had flips of five to six bits from a single alpha particle [15]. The present day devices must have many more flips for the same alpha particle. Error-correcting codes [16] are used to deal with soft errors in memory elements. Due to the continuous scaling down of the device sizes attention has been shifted from memories to combinational logic circuits [17]. Soft error detection and correction in combinational logic is an ongoing research topic since efficient soft error tolerant designs are not available. Logic soft errors are very significant contributors to system-level silent data corruption for designs manufactured in advanced technologies (90nm, 65nm, onwards) and targeted for enterprise computing and communications applications [18]. The model described by P. Sivakumar, to compute the SERs for existing and future microprocessor-style designs predicted that the SER per chip of logic circuits will increase nine orders of magnitude from 1992 to 2011 and at that point will be comparable to the SER per chip of unprotected memory elements [4].

1.5 Thesis objectives

This thesis concentrates purely on asynchronous methodology. The asynchronous methodology used here is Null-Convention Logic (NCL) and is explained in detail in Chapter II. The thesis also includes FPGA designs and usage of different FPGAs. The

topic of soft error detection and correction is also discussed. The major objectives of the thesis are:

- 1) Designing an asynchronous model of DES algorithm using NCL dual-rail logic and simulating the design and addressing different issues in synthesizing the asynchronous DES design on different FPGAs.
- 2) Improving the asynchronous DES design to optimally utilize the resources on the FPGA chip.
- 3) Designing an NCL dual-rail logic circuit which can efficiently detect and correct the occurrence of soft errors in asynchronous circuits.
- 4) Synthesizing the above mentioned circuit on FPGA and testing the circuit using hardware components.
- 5) Adding the soft error tolerant circuit to one of the asynchronous DES rounds and testing its operation.

CHAPTER II

BACKGROUND WORK ON NULL CONVENTION LOGIC

Asynchronous circuits can be grouped into two main categories: bounded-delay and delay-insensitive (DI) models. Bounded-delay models assume that delays in both gates and wires are bounded which leads to extensive timing analysis of worse-case behavior to ensure correct circuit operation. Delay-insensitive circuits assume delays in both logic elements and interconnects to be unbounded, although they assume that wire forks within basic components, such as a full adder, are isochronic, meaning that the wire delays within a component are much less than the logic element delays within the component, which is a valid assumption in the future nanometer technologies. Wire connecting components do not have to adhere to the isochronic fork assumption which enables them to operate in the presence of indefinite arrival times for the reception of inputs.

NCL [19] is a delay-insensitive asynchronous paradigm, which means that NCL circuits will operate correctly regardless of the delay of components and wires. NCL circuits utilize dual-rail or quad-rail logic to achieve delay-insensitivity. Throughout the thesis, the designs make use of dual-rail logic. This chapter explains the basics of NCL such as the components used to construct NCL circuits, criteria that each and every component of the NCL dual-rail signals must possess and transistor level construction of the basic components of NCL circuits.

2.1 Completion Criteria

NCL uses two completeness criteria to achieve its delay-insensitive behavior: symbolic completeness of expression and completeness of input. A symbolically complete expression is defined as an expression that only depends on relationships of the symbols presented in the expression. Dual-rail signals with three logic states (NULL, DATA0, and DATA1) are used to achieve symbolic completeness of expression. A dual-rail signal D consists of two wires, D^0 and D^1 . The value of a dual-rail signal is represented by a value from the set $\{\text{DATA0}, \text{DATA1}, \text{NULL}\}$, shown in Table 2.1. The DATA0 state ($D^0=1(\text{high})$, $D^1=0(\text{low})$) corresponds to a Boolean logic 0. The DATA1 state ($D^0=0(\text{low})$, $D^1=1(\text{high})$) corresponds to a Boolean logic 1. Null state ($D^0=0(\text{low})$, $D^1=0(\text{low})$) corresponds to non- data state. The state where $D^0=1(\text{high})$ and $D^1=1(\text{high})$ is forbidden.

Table 2.1 Dual-rail encoding

Logic Value	Encoding	
	D^0	D^1
DATA1	0	1
DATA0	1	0
NULL	0	0
Invalid	1	1

The second criterion, completeness of input, states that for an NCL combinational circuit, 1) the output may not transition from NULL to a complete set of DATA until the input values are completely DATA and 2) the output may not transition from DATA to a complete set of NULL values until the input values are completely NULL. The criterion, equivalent to Seitz's "weak condition" [20], is illustrated in figure 2.1. This criterion is a necessary condition for speed-independence. The orderings labeled in figure 2.1 are explained below.

- (1) Some inputs become DATA before some outputs become DATA.
- (2) All inputs become DATA before all outputs become DATA.
- (3) All outputs become DATA before some inputs become NULL.
- (4) Some inputs become NULL before some outputs become NULL.
- (5) All inputs become NULL before all outputs become NULL.
- (6) All outputs become NULL before some inputs become DATA.

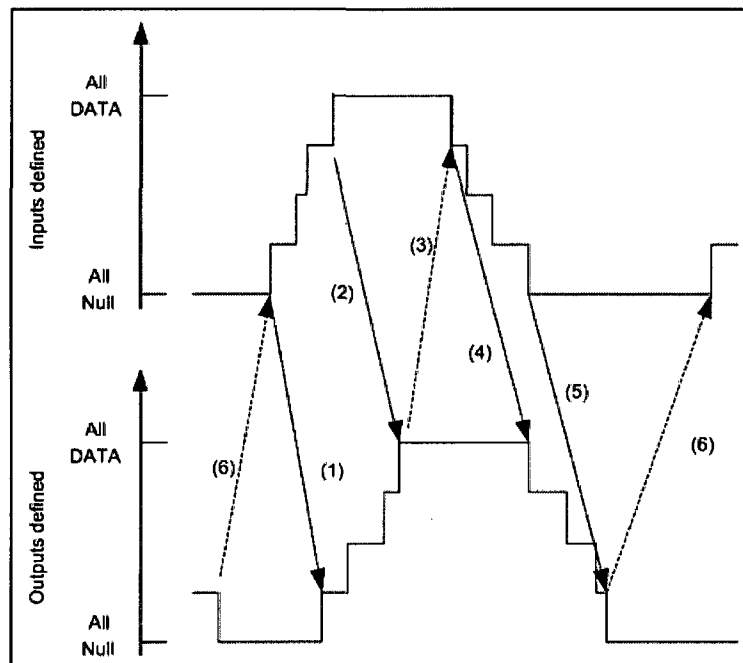


Figure 2.1 Weak conditions for NCL completeness of input.

An output is said to be input-complete with respect to a particular input if the output value (DATA) is not available until the input value (DATA) is available. And this input is a complete input of this particular output. A combinational circuit is input-complete if and only if each input at least has one output that is input-complete with respect to it.

2.2 Threshold Gates with Hysteresis

NCL uses a special type of gates, namely threshold gates with hysteresis [21] [22]. A general name of a fundamental threshold gate is described as $th_{mn}W_{n_1n_2\dots n_w}$, where 'th' means that the gate is a threshold gate, m is the threshold, n is the number of inputs, W means that the following number ' n_1 ', ' n_2 ', ... ' n_w ' are weights of the first ' W ' inputs and the weights of other inputs are one by default. Some of the threshold gates are shown in figure 2.2. In some threshold gates there could be variation for set, reset or inverted output. In that case letter 'd', 'n', or 'b' can be attached to the name of the threshold gate. For example, th_{22n} is a th_{22} gate with a control input 'reset' so that the output is initialized to low as long as 'reset' signal is active. Similarly, 'd' is used to initialize the output to high and 'b' indicates that the gate generates an inverted output.

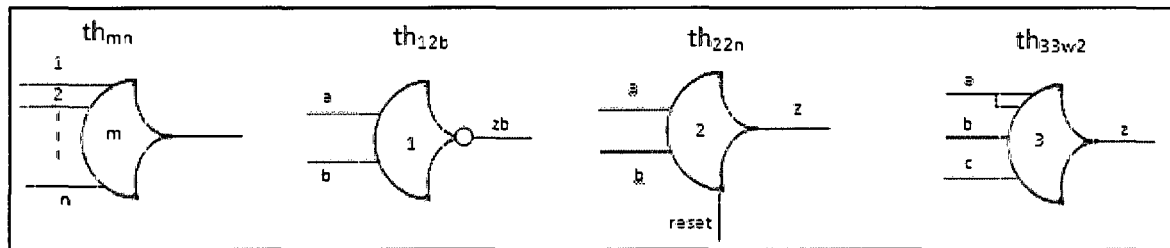


Figure 2.2 Different threshold gates

The threshold behavior of the threshold gate requires that the output become 1 if at least m of n inputs have become 1. The hysteresis behavior requires that the output only changes after a sufficiently complete set of input values have been established. In the case of a transition from a 0 to 1, the output remains at 0 until at least m of the n inputs become 1. In the case of a transition from 1 to 0, the output remains at 1 until all n inputs become 0. The hysteresis within each NCL gate ensures that all inputs must transition to NULL before a combinational circuit's output will transition to NULL, making the circuit input-complete with respect to NULL, assuming that the circuit is input-complete

with respect to DATA. These gates are the basic components upon which all the other essential components of an NCL pipeline (discussed in 2.3) are built.

There are 27 fundamental threshold gates in the NCL design library, as shown in Table 2.2, which constitute the set of all functions consisting of four or fewer variables.

Table 2.2 Twenty seven fundamental NCL gates and their Boolean functions

NCL gate	Boolean Function
th ₁₂	$A + B$
th ₂₂	AB
th ₁₃	$A + B + C$
th ₂₃	$AB + AC + BC$
th ₃₃	ABC
th _{23W2}	$A + BC$
th _{33W2}	$AB + AC$
th ₁₄	$A + B + C + D$
th ₂₄	$AB + AC + AD + BC + BD + CD$
th ₃₄	$ABC + ABD + ACD + BCD$
th ₄₄	$ABCD$
th _{24W2}	$A + BC + BD + CD$
th _{34W2}	$AB + AC + AD + BCD$
th _{44W2}	$ABC + ABD + ACD$
th _{34W3}	$A + BCD$
th _{44W3}	$AB + AC + AD$
th _{24W22}	$A + B + CD$
th _{34W22}	$AB + AC + AD + BC + BD$
th _{44W22}	$AB + ACD + BCD$
th _{34W22}	$ABC + ABD$
th _{34W32}	$A + BC + BD$
th _{34W32}	$AB + ACD$
th _{44W322}	$AB + AC + AD + BC$
th _{34W322}	$AB + AC + BCD$
th _{xor0}	$AB + CD$
th _{and0}	$AB + BC + AD$
th _{24comp}	$AC + BC + AD + BD$

2.3 NCL Pipeline

The framework for NCL systems consists of delay-insensitive combinational logic sandwiched between delay-insensitive registers. This combination of NCL registers along with completion detection circuitry and combinational logic is called NCL pipeline [23]. So, the basic components of any NCL circuit or system are NCL registers,

completion detection circuitry and NCL combinational logic like exor, full-adder, etc.

Figure 2.3 provides the pictorial representation of an NCL pipeline.

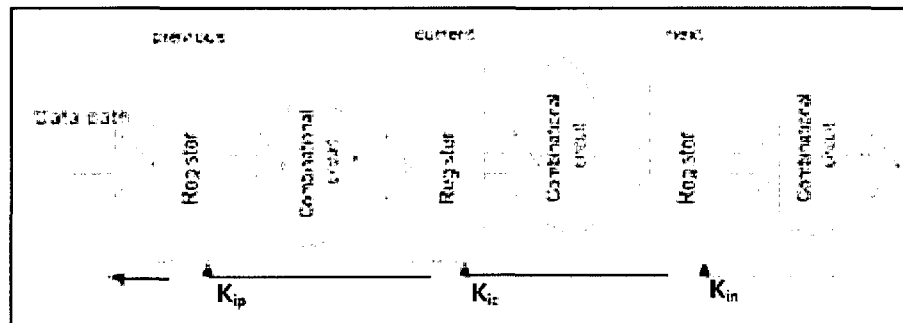


Figure 2.3 Basic NCL pipeline structure

DATA and NULL pairs pass through each of the components in the NCL pipeline consecutively. The presence of NULL is used as time reference in NCL circuits. The input request for each register comes from the completion detection circuit of the next register. Assume that all the circuits are in a NULL state and that the input request signals of the current register (K_{ic}) and the next register (K_{in}) are requesting a DATA wavefront and the previous register is presenting a complete DATA set to its combinational circuit. As the wavefront propagates through the previous combinational circuit to the current register, the current register passes the data since its control line is requesting DATA. When a complete data set is recognized by the current detection circuitry, it transitions its control line (K_{ip}) to the previous register to request NULL indicating that the current register has received and stored the data wavefront and the previous register can pass a NULL wavefront. The requested NULL wavefront from the previous register can arrive at the current register but, as long as its (current register's) control line (K_{ic}) is requesting DATA, the NULL wavefront will be blocked and the current register will maintain presentation the set of DATA values to the current combinational circuit. The control line for the current register will remain requesting DATA until the DATA wavefront has

propagated through the current circuit and has been received by the next register. When the next register receives and stores the DATA wavefront, the DATA set no longer needs to be maintained by the current register. The next completion detection circuit detects the complete DATA set and transitions its acknowledge line(K_{ic}) to request NULL indicating that it has received the DATA wavefront and the current register can allow a NULL wavefront. This is the entire operation of the whole NCL pipeline.

2.3.1 NCL Register

NCL systems contain at least two delay-insensitive registers, one at the input and the other at the output. Two adjacent register stages interact through their request and acknowledge signals; K_i and K_o , respectively, to prevent the current DATA wavefront from overwriting the previous DATA wavefront, by ensuring that the two DATA wavefronts are always separated by a NULL wavefront.

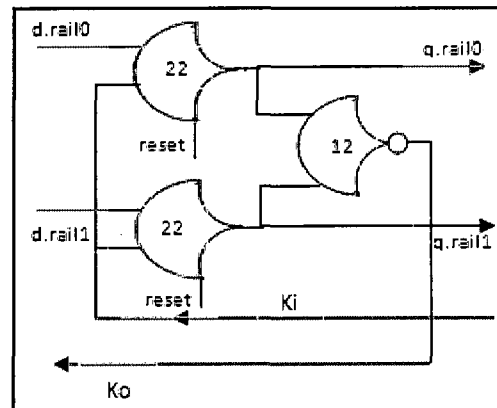


Figure 2.4 1-bit NCL register

Figure 2.4 shows a 1-bit NCL register. An n-bit register is realized through cascaded arrangements of n, 1-bit dual-rail registers. Each 1-bit NCL register used throughout the thesis comprises of two th_{22n} gates that pass a DATA value at the input only when K_i (request signal) is logic 1 and pass NULL only when K_i is logic 0. The

register also contains a th_{12b} gate which has two inputs one of which is connected to the output of one of the two th_{22n} gates and the other input is connected to the output of other th_{22n} gate. The output of th_{12b} is denoted as Ko , the acknowledge signal of the 1-bit register. Ko becomes logic 0 when the register receives complete DATA and has logic 1 when the register receives NULL. The acknowledge signals of each 1-bit register in an n -bit register are combined in the completion detection circuitry to produce the request signal to the previous register stage. The request signal Ki of the current register is from the output of the completion detection circuitry of the next register. Since both the th_{22} gates in the 1-bit register are reset to NULL (th_{22n}), the register outputs zeros when the reset signal or input is high. However, either register could be instead reset to a DATA value by replacing exactly one of the th_{22n} gates with a th_{22d} gate. But for the applications in this thesis only th_{22n} gates are used.

2.3.2 Completion Detection Circuitry

Completion detection circuitry consists of set of gates which determine the complete arrival of DATA or NULL at the registers. As mentioned in the previous section, all the Kos of each 1-bit register goes through the completion detection circuitry and produces the request signal for the previous register. Figure 2.5 is an example of an n -bit completion detection circuit.

Since the maximum input threshold gate is the th_{44} gate, the number of logic levels in the completion component for an n -bit register is given by $\log_4 n$. For example, suppose a 64-bit register. The completion detection circuitry has $\log_4 64 = 3$ levels. In the first level the circuit has 16, th_{44} gates, in the second level the circuit has 4, th_{44} gates and the third level has 1, th_{44} gate.

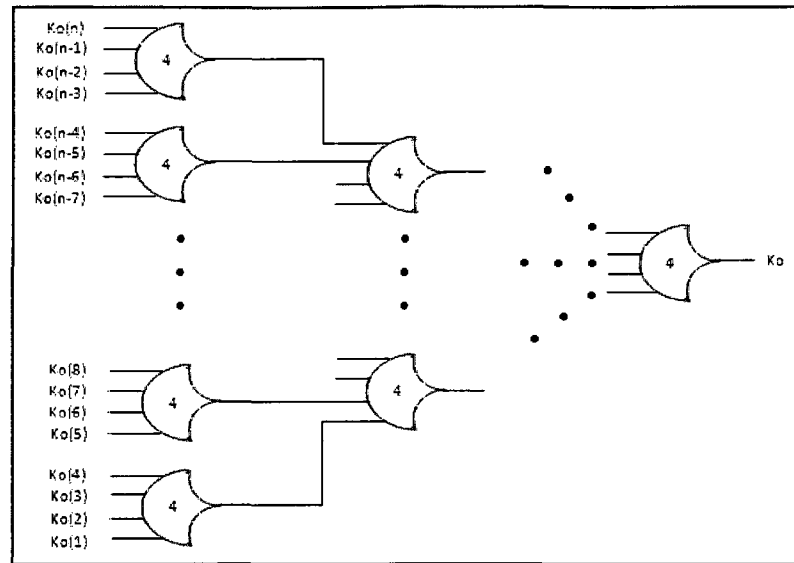


Figure 2.5 n-bit completion detection circuitry

Here is another example which clarifies the process of constructing the completion detection circuitry in a better way. Consider a 44-bit NCL register. The completion detection circuitry of this 44-bit register has three logic levels and is constructed using 11, th_{44} gates in the first logic level, 2, th_{44} gates and 1, th_{33} gate in the second logic level and 1, th_{33} gate in the third logic level.

2.3.3 NCL Combinational Circuit

The functionality of NCL combinational circuits are similar to Boolean combinational logic circuits except that NCL circuits are made from the 27 threshold gates mentioned in table 2.2. All the NCL combinational circuits must maintain two vital properties viz., input-completeness and observability.

Input-completeness requires that all outputs of a combinational circuit may not transition from NULL to DATA until all inputs have transitioned from NULL to DATA, and that all outputs of a combinational circuit may not transition from DATA to NULL until all inputs have transitioned from DATA to NULL.

Observability requires that no orphans may propagate through a gate. An orphan is defined as a wire that transitions during the current DATA wavefront, but is not used in the determination of the output. Orphans are caused by wire forks and can be neglected through the isochronic fork assumption (i.e. gate delays are much longer than wire delays within a component), as long as they are not allowed to cross a gate boundary. This observability condition, also referred to as indicatability or stability, ensures that every gate transition is observable at the output; which means that every gate that transitions is necessary to transition at least one of the outputs.

Figure 2.6 shows some of the NCL combinational circuits a) inverter, b) exor and c) full-adder using threshold gates. Exor is used in the thesis while constructing DES algorithm using dual-rail logic and full-adder is taken as an example for combinational circuit to demonstrate the working of soft-error detection and correction circuitry.

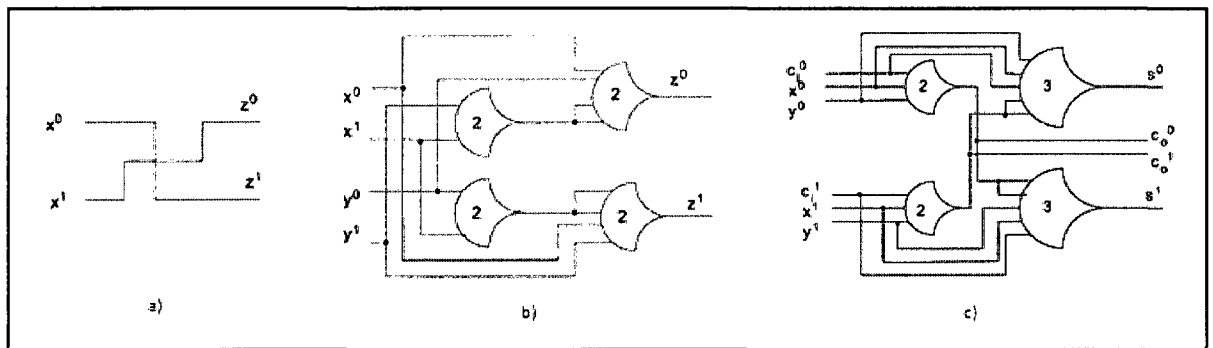


Figure 2.6 NCL implementation of a) Inverter b) Exor gate c) Full-adder [24]

2.4 NCL Circuits using CMOS Transistors

An efficient method is to design the threshold gates with hysteresis using CMOS technology at transistor level. There are three different ways of realizing a threshold gate using transistors. These are static, semi-static and dynamic. Static gates are the stable transistor level threshold gates while semi-static and dynamic can reduce the amount of

area occupied by them. The three designs differ in their structure. The general structure of a static threshold gate is shown in figure 2.7. This thesis doesn't use the transistor level designs of the threshold gates but makes use of their behavioral description using VHDL. This section is basically for understanding the concepts of threshold gates at transistor level.

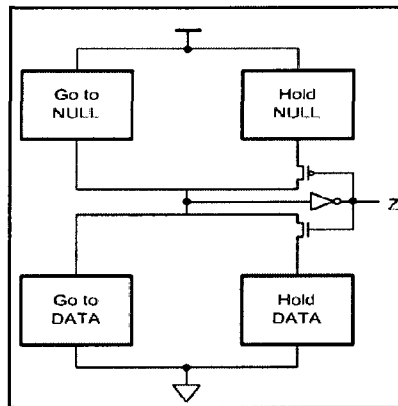


Figure 2.7 General structure of a static gate

The Go to NULL and Hold DATA blocks are complementary to each other and have the universal forms shown in figure 2.8. Go to NULL block is only ON when all N inputs are 0 and Hold DATA block is ON if one or more of the inputs are 1. Because of the series chain in the Go to NULL block, speed considerations will limit these structures to a maximum number of inputs, typically less than six. The structures of Go to DATA and Hold NULL are complementary to each other and depends on the particular threshold gate.

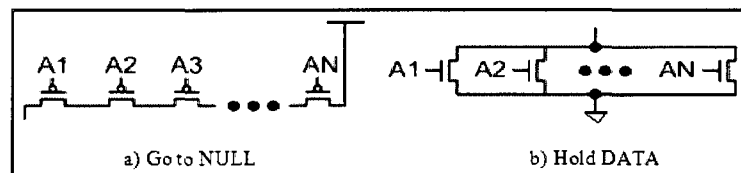
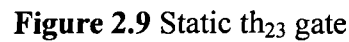


Figure 2.8 Go to NULL and Hold DATA transistor blocks



All the components in the NCL circuit design are made up of these threshold gates which could be realized using CMOS transistors. So each component, be it an NCL register, completion detection circuitry or combinational logic, is a collection of these threshold gates.

CHAPTER III

NCL CIRCUIT DESIGN WITH VHDL

An FPGA is a semiconductor device that can be programmed or configured any number of times using a schematic design or a source code in HDL (hardware description language) that describes the user's hardware design. The NCL circuit designs using HDLs could be used on these existing CAD tools for synchronous circuits [25]. In order for the FPGA to be programmed with NCL circuits, the VHDL (VHSIC (Very High Speed Integrated Circuit) Hardware Description Language) description of these circuits is necessary. The first part of this chapter gives details about FPGA. The design flow used in construction of these NCL circuits until they are programmed on the reconfigurable logic is explained in the next section followed by the internal details of each essential unit used for constructing NCL circuits. The fourth section details an example of a simple NCL circuit and its simulation using the software.

3.1 FPGA

The basic resources present in an FPGA are CLBs (configurable logic blocks which contain combinational logic and register resources), IOBs (input/output blocks and are the interface between FPGA and outside world), PIs (programmable interconnections), RAM blocks and other resources like three-state buffers, global clock buffers, boundary scan logic, dedicated multipliers, digital clock managers, etc. Figure 3.1 shows the basic resources present inside an FPGA.

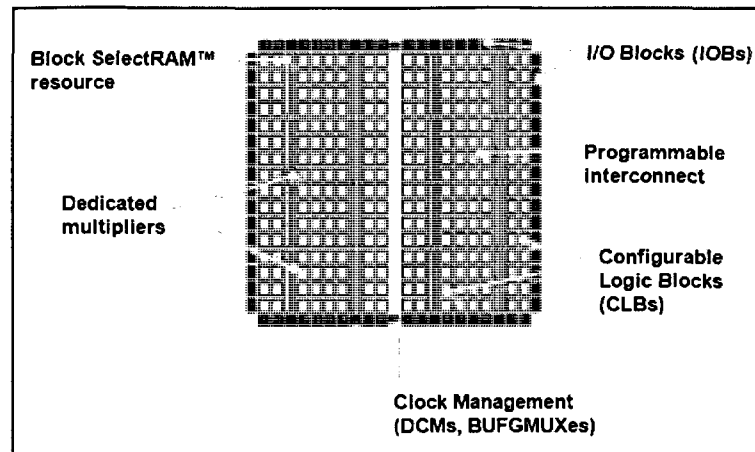


Figure 3.1 Basic resources of an FPGA [26]

The basic design steps to configure an FPGA are shown in figure 3.2. These are the design entry, design synthesis, design implementation, and device programming. Design verification, which includes both functional verification and timing verification, takes places at different points during the design flow.

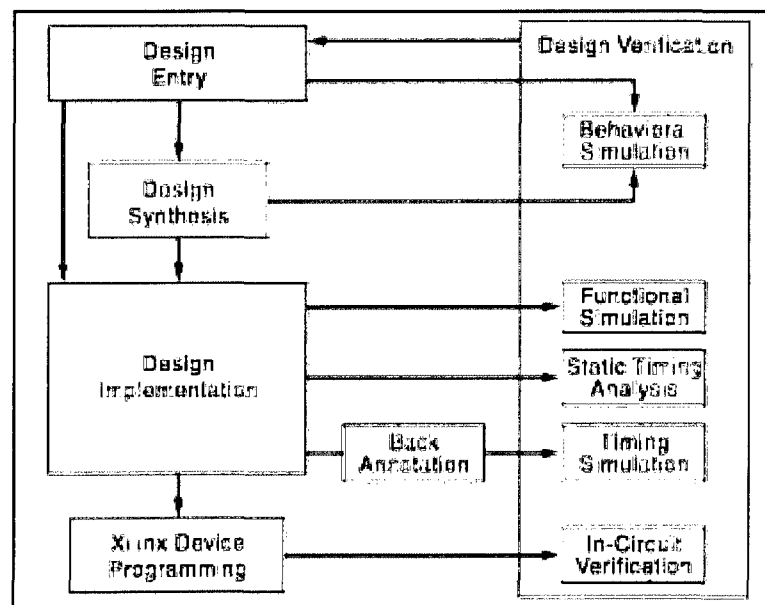


Figure 3.2 FPGA design flow [27]

Design entry is the first and foremost step in the design process to configure an FPGA. This step involves the creation of design files using schematic editor or HDL

(Hardware Description Language) and is referred to as RTL (Register Transfer Level). In RTL design, a circuit's behavior is defined in terms of the flow of signals (or transfer of data) between hardware registers, and the logical operations performed on those signals. Automatically creating lower level of logic abstraction from higher level of logic abstraction is what design synthesis is all about. In this design process an RTL description is usually converted to a gate-level description of the circuit by a logic synthesis tool. The next step is the design implementation. This step comprises of three steps; translation, mapping and place and routing. Merging multiple design files into a single netlist is called translation. Mapping is nothing but assigning a logic element to a physical element. Mapping logic onto the specific locations in the target FPGA chip, connecting the components and extracting timing data into reports is place and route. The design is verified at different levels in this process. Checking the syntax, functional simulation and timing simulation are some of the verification procedures. Once all the above steps are successfully performed there is the much awaited final step called programming or configuring the device. This involves creating a file that the FPGA can understand, for example, a .bit file in the case of Xilinx FPGAs or .sof file in for Altera FPGAs and downloading the file to the FPGA.

3.2 Design Flow used in the Thesis

Figure 3.3 illustrates the design flow used in the thesis starting from how the concept of dual-rail logic has been introduced into the design construction all the way through programming the FPGA with the constructed design. In order to configure the FPGA with NCL circuits, the design entry step must be performed using a HDL. VHDL

(VHSIC (Very High Speed Integrated Circuits) hardware description language) is the language used throughout the thesis.

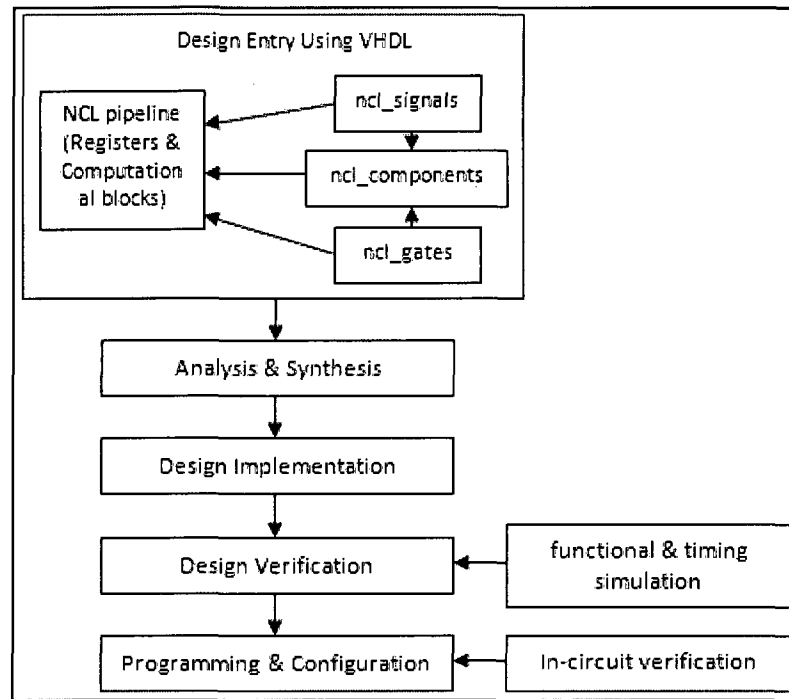


Figure 3.3 Design flow

The design entry step deals with the construction of NCL pipeline which includes NCL dual-rail registers and the NCL dual-rail computational blocks present between them. The way the NCL dual-rails circuits are constructed is clearly described in section 3.3. Once the circuits are designed, these circuits are undergone compilation using Quartus II software which include analysis & synthesis and design implementation. Now, the circuit needs to be verified. So, both functional simulation (no delays included) and timing simulation are performed on the circuit. Once the verification is done, the synthesized design needs to be programmed on an FPGA. Assign I/O pins of the FPGA to the inputs and outputs of the generated design and we are set to program the FPGA with this design. The “program device” option in the Quartus II leads to a window where

*.sof file is selected to be programmed on the Altera's Cyclone II FPGA present on the DE2 development and education board. Once the built circuit is configured on a FPGA, verification is done by providing inputs generated by components on board and outputs are extracted from the expansion headers of the board and viewed on a logic analyzer.

3.3 NCL Circuits in VHDL

In order to design NCL circuits in VHDL, several components need to be coded first. These components include creating a user-defined data-type called the "dual_rail_logic", creating generic n-bit NCL dual-rail registers and completion detection circuitry and threshold gates with hysteresis, etc., which are supposed to be building blocks of NCL circuits and aid in constructing any kind of NCL pipeline architecture. This section explains how these individual units are coded and how they are used in constructing NCL pipeline architectures.

3.3.1 Data Type called dual_rail_logic

VHDL doesn't contain a predefined data-type for the dual-rail logic signals. So, first a user-defined data-type called "dual_rail_logic" has been defined which comprises of two std_logic type signals: Rail0 and Rail1. The data-type also has its vector definition as dual_rail_logic_vector. The user-defined data type is defined in a package called "ncl_signals" and this package is placed in the work directory and will be accessed by all the further components using dual-rail logic signals. For instance, if a NCL dual-rail full-adder circuit is to be created, then the inputs, X, Y, and Ci to this computational block are dual_rail_logic signals which has two std_logic signals each as (X.Rail0 , X.Rail1) , (Y.Rail0, Y.Rail1) and (Ci.Rail0, Ci.Rail1). In the VHDL file used to describe the behavioral model of full-adder circuit, the "ncl_signals" package need to be mentioned as

“use.work.ncl_signals.all” and X,Y and Ci needs to be declared as dual_rail_logic instead of std_logic or bit, etc. Figure 3.4 gives the code which creates a user-defined data-type called the “dual_rail_logic”.

```

Library IEEE;
use IEEE.std_logic_1164.all;

package ncl_signals is

type dual_rail_logic is
  record
    RAIL1 : std_logic;
    RAIL0 : std_logic;
  end record;

type dual_rail_logic_vector is array (NATURAL range <>) of dual_rail_logic;

end ncl_signals;

```

Figure 3.4 dual_rail_logic data-type

3.3.2 Threshold Gates with Hysteresis in VHDL

The behavioral models of the required threshold gates with hysteresis are written using VHDL in a file called “ncl_gates” (refer to Appendix B for all the other threshold gates). A th_{22} gate is defined in VHDL and is shown in figure 3.5 as an example.

```

library ieee;
use ieee.std_logic_1164.all;

entity th22x0 is
  port(a: in std_logic;
       b: in std_logic;
       z: out std_logic );
end th22x0;

architecture archth22x0 of th22x0 is
begin
  th22x0: process(a, b)
  begin
    if (a= '1' and b= '1') then
      z <= '1';
    elsif (a= '0' and b= '0') then
      z <= '0';
    end if;
  end process;
end archth22x0;

```

Figure 3.5 Behavioral description of th_{22} in VHDL

The inputs to these threshold gates are individual dual-rail signals which could be either Rail0 or Rail1 according to the designed circuit and hence are std_logic signals. For example, the 1-bit dual-rail register in figure 2.4 has one input as Ki signal and the other input (d.rail0 for the top th_{22} and d.rail1 for the bottom th_{22}), one of the rails of 'd' which is a dual-rail signal. The "ncl_gates" file is also needs to be added to the project that is being constructed so that the components built in the current project can access these threshold gates.

3.3.3 NCL Dual-Rail Registers & Completion Detection Circuits in VHDL

NCL dual-rail registers and completion detection circuits are described in a VHDL file called "ncl_components". "ncl_components" file consists of design units like generic n-bit NCL dual-rail register and n-bit completion detection circuitry along with their internal components defined in it. The design units present in the "ncl_components" file make use of the "ncl_signals" package and threshold gates with hysteresis defined in the "ncl_gates" file. All these files including "ncl_signals" are presented in Appendix B for reference.

A single bit NCL register has two th_{22n} gates and a th_{12b} gate as explained in section 2.3.1. So the behavioral description of a single bit register in VHDL should contain the instances of the two gates as shown in figure 3.6. These instances are accessed from the "ncl_gates" file and since the inputs are dual_rail_logic signals, the "ncl_signals" package declaration needs to be done. An n-bit NCL register is generated by iteratively generating the same instance of 1-bit NCL register the input data length times as shown in figure 3.7.

```

entity ncl_register_D1 is
    generic(initial_value: integer := -4); -- 1=DATA1, 0=DATA0, -4=NULL
    port(D: in dual_rail_logic;
         ki: in std_logic;
         rst: in std_logic;
         Q: out dual_rail_logic;
         ko: out std_logic);
end ncl_register_D1;

architecture arch of ncl_register_d1 is
    signal Qbuf: dual_rail_logic;

    component th22nx0
        port (a, b, rst: IN std_logic;
              z: OUT std_logic);
    end component;

    component th22dx0
        port (a, b, rst: IN std_logic;
              z: OUT std_logic);
    end component;

    component th12bx0
        port (a, b: IN std_logic;
              zb: OUT std_logic);
    end component;

begin
    RstN: if initial_value = -4 generate
        R0: th22nx0 port map(D.rail0, ki, rst, Qbuf.rail0);

        R1: th22nx0 port map(D.rail1, ki, rst, Qbuf.rail1);
    end generate;

    Rst1: if initial_value = 1 generate
        R0: th22nx0 port map(D.rail0, ki, rst, Qbuf.rail0);

        R1: th22dx0 port map(D.rail1, ki, rst, Qbuf.rail1);
    end generate;

    Rst0: if initial_value = 0 generate
        R0: th22dx0 port map(D.rail0, ki, rst, Qbuf.rail0);
        R1: th22nx0 port map(D.rail1, ki, rst, Qbuf.rail1);
    end generate;

    Q <= Qbuf;

    COMP: th12bx0 port map(Qbuf.rail0, Qbuf.rail1, ko);
end;

```

Figure 3.6 1-bit NCL register in VHDL

```

gen_reg: for i in 0 to D'length-1 generate
    REGi: ncl_register_D1
        generic map(initial_value)
        port map(D(i), ki(i), rst, Q(i), ko(i));
end generate;

```

Figure 3.7 Creating n-bit register from 1-bit register

The only gates used for any completion detection circuitry are th_{22} , th_{33} , th_{44} . Based on the number of input signals, the number of logic levels must be calculated using $\log_4 n$. This component basically checks if all the inputs are 0s during DATA and 1s during NULL. The code used first calculates the number of logic levels. Then it checks if the number of inputs are multiples of four. If yes then the signals are assigned to th_{44} in sets of four. If there are any leftovers from the multiples of four, it checks if they are two or three. If the leftovers are three then, the code assigns them to th_{33} or if the leftovers are two then, th_{22} will be assigned to the signals. The same process repeats for each logic level. For the code of the completion detection circuitry refer to Appendix B.

3.3.4 Constructing Computational Blocks

<pre> entity exor is port(x : in dual_rail_logic; y : in dual_rail_logic; z : out dual_rail_logic); end exor; architecture Behavioral of exor is signal u1,u2 : std_logic; component th22x0 port(a: in std_logic; b: in std_logic; z: out std_logic); end component; component th23u2x0 is port(a: in std_logic; -- weight 2 b: in std_logic; c: in std_logic; z: out std_logic); end component; begin g1 : th22x0 port map(y.raill,x.raill,u1); g2 : th22x0 port map(y.raill0,x.raill,u2); g3 : th23u2x0 port map(u1,y.raill0,x.raill0,z.raill0); g4 : th23u2x0 port map(u2,y.raill,x.raill0,z.raill1); end Behavioral; </pre> <p style="text-align: center;">a)</p>	<pre> entity fulladder is port (a : in dual_rail_logic_vector(1 to 3); s : out dual_rail_logic_vector(1 to 2)); end fulladder; architecture behavioral of fulladder is signal c0,c1 : std_logic; component th23x0 is port(a: in std_logic; b: in std_logic; c: in std_logic; z: out std_logic); end component; component th34w2x0 is port(a: in std_logic; -- weight 2 b: in std_logic; c: in std_logic; d: in std_logic; z: out std_logic); end component; begin g1 : th23x0 port map(a(1).raill,a(2).raill,a(3).raill,c0); g2 : th23x0 port map(a(1).raill,a(2).raill,a(3).raill,c1); g3 : th34w2x0 port map(c1,a(1).raill0,a(2).raill0,a(3).raill0,s(2).raill0); g4 : th34w2x0 port map(c0,a(1).raill,a(2).raill,a(3).raill,s(2).raill); s(1).raill0<=c0;s(1).raill<=c1; end behavioral; </pre> <p style="text-align: center;">b)</p>
---	--

Figure 3.8 VHDL code for a) Exor gate b) Full-adder

One of the essential parts of an NCL pipeline is the computational block. A computational block is a combinational logic which performs some operations on the inputs generating outputs and is sandwiched between two NCL dual-rail registers. This thesis makes use of only two combinational circuits; the exor gate and the full-adder.

These computational blocks are constructed using the threshold gates with hysteresis and dual_rail_logic signals. The schematics of the NCL dual-rail exor gate and full-adder are provided in figure 2.6 and the VHDL behavioral descriptions are provided in the below figure 3.8.

3.4 Simulation of a Simple NCL Pipeline

This section explains in detail the procedure how a simple NCL pipeline is created and simulated using Quartus II software. The pipeline considered has two NCL dual-rail register with an NCL dual-rail exor gate as the computational unit between them as shown in figure 3.9.

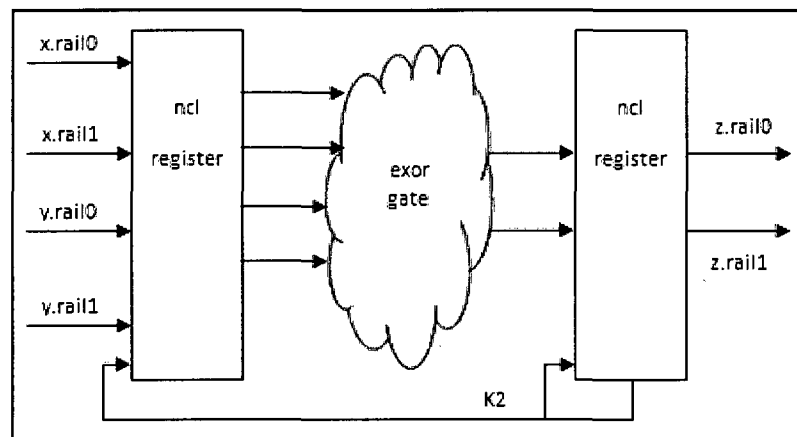


Figure 3.9 NCL pipeline with exor gate

The very first step is to create a 'New Project' using 'New Project Wizard' in Quartus II [28] (Quartus II tutorial provides all the information on how to compile the designs and program an FPGA). Select a 'New' from the 'file' menu and select 'VHDL file'. In the file write the following code shown in figure 3.10 and save it as 'exor.vhd'. In similar way add the other VHDL files; 'initreg.vhd', 'finalreg.vhd' and 'exor_dl.vhd' required for the 'exor.vhd' file and these files are presented in figure 3.11. 'initreg.vhd' has two components 'ncl_register_D' which is a generic n-bit register present in

'ncl_components' file mentioned earlier and 'th22x0' is th_{22} threshold gate which is acting as the completion detection unit for this register. The behavioral description of this gate is present in 'ncl_gates' file. 'finalreg.vhd' has one component which is 1-bit NCL register called ncl_component_D1 and is also described in 'ncl_components' file.

```

library ieee;
use ieee.std_logic_1164.all;
use work.ncl_signals.all;

entity exor is
  port ( x : in dual_rail_logic;
         y : in dual_rail_logic;
         rst : in std_logic;
         z : out dual_rail_logic);
end exor;

architecture behavioral of exor is
  signal m,n: dual_rail_logic_vector(1 to 2);
  signal zo: dual_rail_logic;
  signal k1,k2: std_logic;
  component initreg is
    port ( D : in dual_rail_logic_vector(1 to 2);
          ki : in std_logic;
          rst : in std_logic;
          Q : out dual_rail_logic_vector(1 to 2);
          ko : out std_logic);
  end component;
  component finalreg is
    port ( D : in dual_rail_logic;
          ki : in std_logic;
          rst : in std_logic;
          Q : out dual_rail_logic;
          ko : out std_logic);
  end component;
  component exor_d1
    port(ax : in dual_rail_logic;
         bx : in dual_rail_logic;
         cx : out dual_rail_logic);
  end component;
begin
  m(1)<=x; m(2)<=y;
  reg1 : initreg port map(m,k2,rst,n,k1);
  cb : exor_d1 port map(n(1),n(2),zo);
  reg2 : finalreg port map(zo,k2,rst,z,k2);
end behavioral;

```

Figure 3.10 VHDL code for the NCL pipeline with exor gate

```

Library IEEE;
Use IEEE.std_logic_1164.all;
Use work.ncl_signals.all;

entity initreg is
  port ( D : in dual_rail_logic_vector(1 to 2);
        ki : in std_logic;
        rst : in std_logic;
        Q : out dual_rail_logic_vector(1 to 2);
        ko : out std_logic);
end initreg;

architecture behavioral of initreg is
  signal ao : std_logic_vector(1 to 2);
  component ncl_register_D
    generic(width: integer;initial_value: integer);-- 1=DATA1,0=DATA0,-4=NULL
    port(D: in dual_rail_logic_vector(width-1 downto 0);
         ki: in std_logic;
         rst: in std_logic;
         Q: out dual_rail_logic_vector(width-1 downto 0);
         ko : out std_logic_vector(width-1 downto 0));
  end component;
  component th2x0
    port(a: in std_logic;
         b: in std_logic;
         z: out std_logic);
  end component;
begin
  regi : ncl_register_D generic map(width=>2,initial_value=>-4)
    port map(D,ki,rst,Q,ao);
  cdi : th2x0 port map(ao(2),ao(1),ko);
end behavioral;

```

Figure 3.11a) initreg.vhd

<pre> Library IEEE; use IEEE.std_logic_1164.all; use work.ncl_signals.all; entity exor_d1 is port(ax : in dual_rail_logic; bx : in dual_rail_logic; cx : out dual_rail_logic); end exor_d1; architecture Behavioral of exor_d1 is signal u1,u2 : std_logic; component th2x0 port(a: in std_logic; b: in std_logic; z: out std_logic); end component; component th23u2x0 is port(a: in std_logic; -- weight 2 b: in std_logic; c: in std_logic; z: out std_logic); end component; begin g1 : th2x0 port map(bx.ra11,ax.ra11,u1); g2 : th2x0 port map(bx.ra10,ax.ra11,u2); g3 : th23u2x0 port map(u1,bx.ra10,ax.ra10,cx.ra10); g4 : th23u2x0 port map(u2,bx.ra11,ax.ra10,cx.ra11); end Behavioral; </pre>	<pre> Library IEEE; Use IEEE.std_logic_1164.all; Use work.ncl_signals.all; entity finalreg is port (D : in dual_rail_logic; ki : in std_logic; rst : in std_logic; Q : out dual_rail_logic; ko : out std_logic); end finalreg; architecture behavioral of finalreg is component ncl_register_D1 generic(initial_value: integer);--1=DATA1,0=DATA0,-4=NULL port(D: in dual_rail_logic; ki: in std_logic; rst: in std_logic; Q: out dual_rail_logic; ko: out std_logic); end component; begin regi : ncl_register_D1 generic map(initial_value=>-4) port map(D,ki,rst,Q,ko); end behavioral; </pre>
b) exor_d1.vhd	c) finalreg.vhd

Figure 3.11b) exor_d1.vhd c) finalreg.vhd

finalreg.vhd doesn't have a separate completion detection unit because it is just a 1-bit register. The output signal from `th12bx0` acts as the completion detection signal. Since the internal component definitions are present in 'ncl_components' and 'ncl_gates' these files need to be added to the project by selecting 'add/remove files in project' in the 'project' menu. Since all these components are dual-rail logic, 'ncl_signals' files must also be added to the project. Once all the required files are present in the project folder, the project needs to be compiled by clicking on the 'compile design' option on the Quartus II software. Check for any syntax error or for any other errors during synthesis and implementation such as I/Os not sufficient for the design or design is too large to fit on to the device, etc.

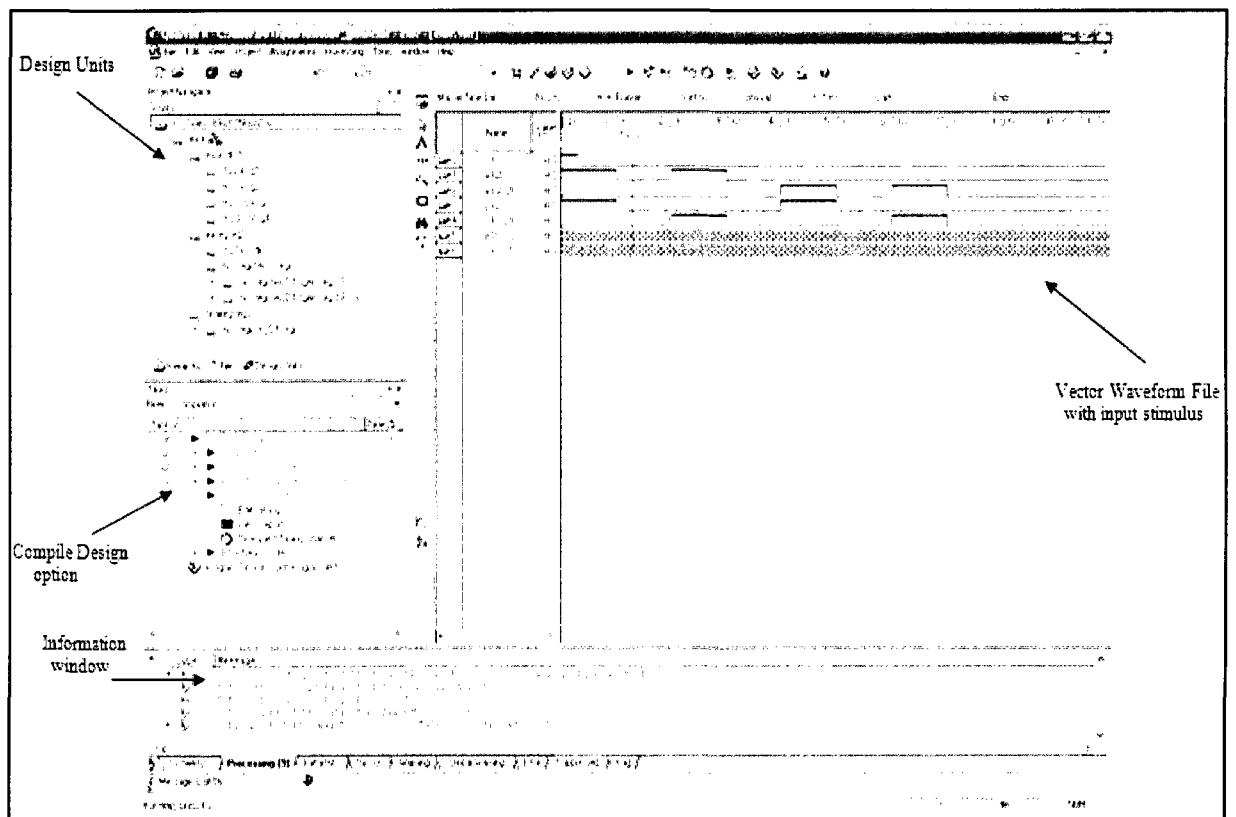


Figure 3.12 Quartus II Software Window

Once the above procedure is done generate a new 'vector waveform file' in order to provide simulation inputs. Add the input and output signals to the 'vector waveform file' and provide appropriate input signals as shown in figure 3.12.

Since the circuit used has dual-rail components, first reset or 'rst' as named in the design need to be asserted. Then the other inputs must be provided as consecutive DATA and NULL pairs. As inputs are provided in the 'vector waveform file' the design needs to be verified or simulated for these inputs. The functional simulation results after simulation is shown in figure 3.13.

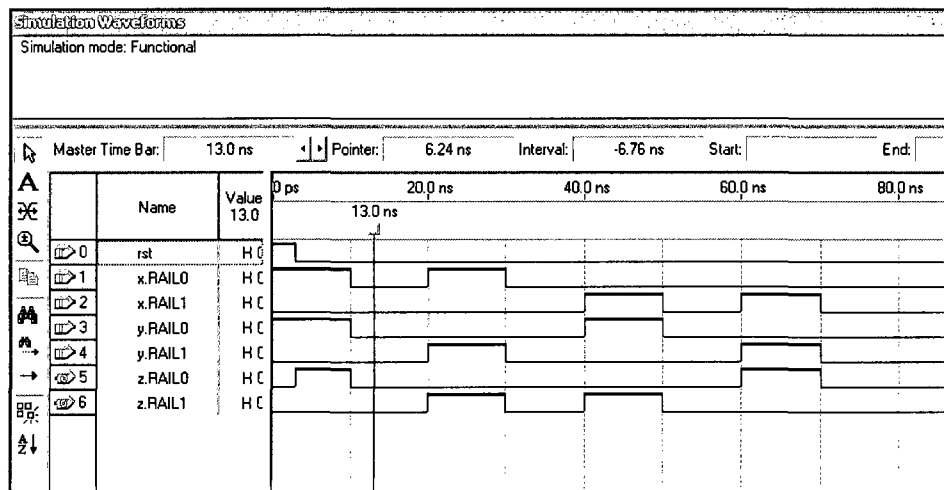


Figure 3.13 Simulation results of exor.vhd

CHAPTER IV

ASYNCHRONOUS DATA ENCRYPTION STANDARD ALGORITHM USING NCL

The symmetric property of DES algorithm provides an added advantage to implement DES using NCL dual-rail logic. It creates a scope for the pipelined architecture shown in figure 4.1 where the whole algorithm has 17 combinational logics embedded between NCL registers. The first round has plaintext and key as inputs and L1, R1 and C1, D1 as the outputs. The next round till 15 such rounds have the same combinational logic which takes L_{n-1} , R_{n-1} and C_{n-1} , D_{n-1} as inputs and generates L_n , R_n and C_n , D_n as outputs. The 16th round takes L15, R15 and CD15 as inputs and gives out L16 and R16 as outputs which are then permuted in the 17th round to generate the ciphertext output. Altogether, the number of registers present in the asynchronous DES pipeline is eighteen; initial register, NCL registers 1-15, register 16 and final register and the number of combinational logic circuits that are embedded between these eighteen registers are seventeen combinational circuits also called rounds in this case and they are, initial round, rounds 1-14, round 15 and final round. Also, each NCL register has its own completion detection circuitry along with it. The structure of the completion detection circuitry varies as the NCL register structure varies. The details of each component of the DES pipeline are mentioned in the following sections.

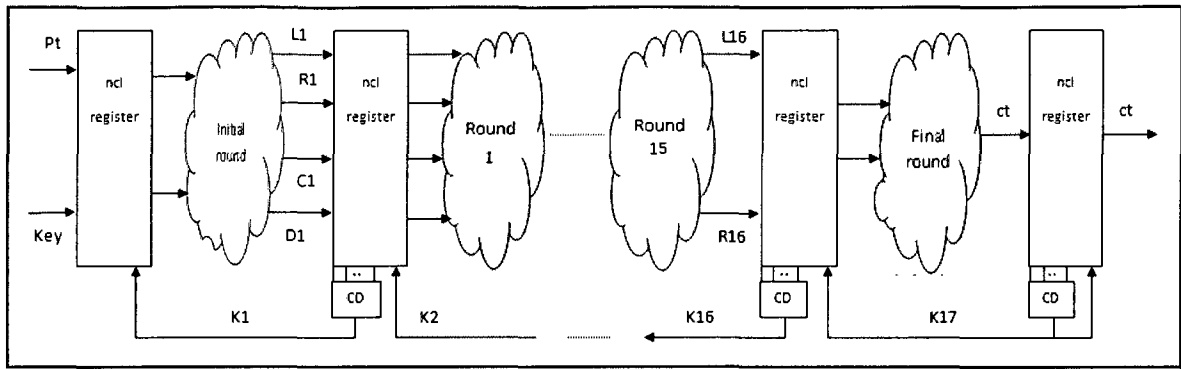


Figure 4.1 DES pipeline in NCL dual-rail logic

4.1 Initial Register

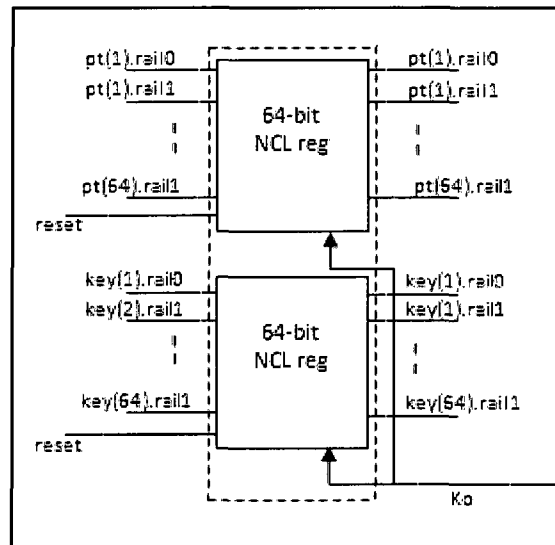


Figure 4.2 Initial Register

The starting stage of the DES pipeline in NCL dual-rail logic is the initial register. This register has plaintext of 64 bits and key of 64 bit dual-rail signals as inputs. Along with these it also has reset signal and K_i signal inputs similar to all the other registers. This initial register is different from the other registers in that it doesn't have a K_o signal since there is no register prior to it. This register will output all zeros when reset signal is logic 1. It allows the plaintext and key values to pass through it when K_i is logic 1. If K_i is logic 0 then the initial register will stop any flow of DATA and will be ready to pass

NULL through it. This register is the only register in the entire DES pipeline that doesn't have a completion detection circuitry. This circuitry has been eliminated to save logic on the FPGA. The structure of initial register is presented in figure 4.2.

4.2 Initial Round

DES pipeline in NCL dual-rail logic starts with an initial register, followed by an initial round. The operations that are performed on the outputs of the initial register are described in figure 4.3.

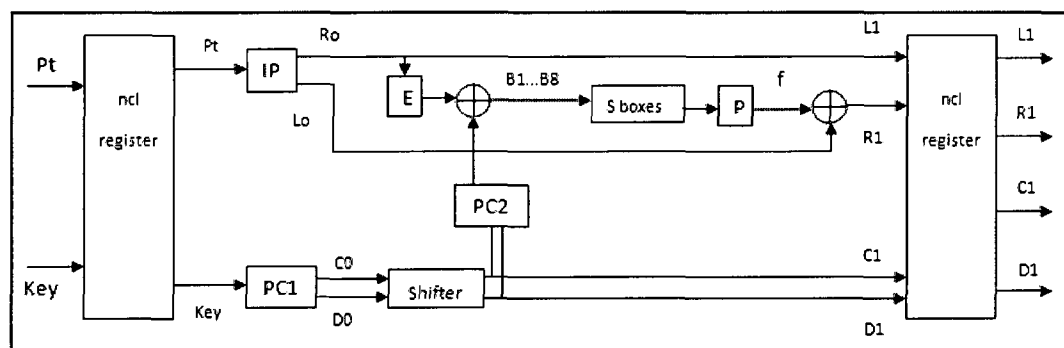


Figure 4.3 Initial round in the DES pipeline

As soon as the plaintext denoted as pt in figure 4.3 enters the initial round, an initial permutation IP is performed on it and is divided into L0 and R0. The key also undergoes permutation PC1 and is divided into C0 and D0. Shifter is basically used to left shift the bits in C0 and D0 which become C1 and D1 for the next stage and are used to form subkey in the present round. The output of the shifter is concatenated and another permutation PC2 is applied on it. The function E expands R0 from 32 bits to 48 bits as mentioned in the DES algorithm. The outputs of E and PC2 are exored bit by bit using the exor gate designed for NCL dual-rail logic. This is the only dual-rail combinational logic circuit used in the DES algorithm. All the permutations, expansions and left shifts are basically wiring and don't involve any logic function or operation. The output of the

exor gate goes through S-boxes which are constructed using if-else statements addressing all the possible combination of inputs. Eight S-boxes are written in VHDL and the structure of these is shown in figure 4.4 which addresses six-bits of inputs each producing four-bit outputs. The outputs of the S-boxes are combined and then a final permutation P is performed on the bits. The L0 output of IP is then exored with the P output and is fed to the next register as R1 input. The R0 output of the IP becomes the L1 input to the next stage.

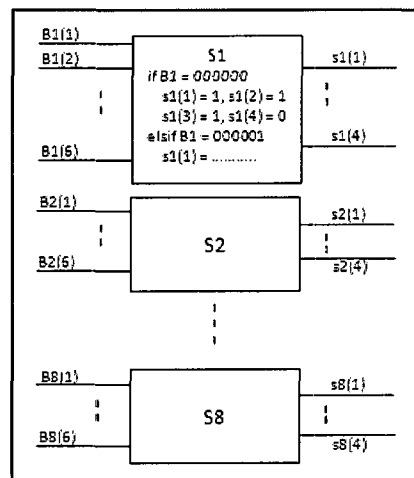


Figure 4.4 S-box inputs and outputs

4.3 NCL Registers 1-15

The outputs of the initial round are L1, R1, C1, and D1 which are fed to the next register which takes these signals as inputs. As mentioned earlier the whole DES pipeline consists of fourteen such rounds other than initial round which has the same structure and does similar operations. Due to this reason, the registers governing the rounds on both sides need to be similar, accepting same number of input signals and outputs the same. So NCL registers 1-15 allows Ln, Rn, Cn and Dn (L1, R1, C1, D1.....L15, R15, C15, D15) to pass through them. The completion detection circuits for all these registers have four internal circuits which take the Ko signals of each term like Ln, Rn, Cn and Dn. The

four outputs of the four completion detection circuits are fed to a th_{44} gate, the output of which is the Ko signals (acknowledge signal) for the entire register. The whole structure of the NCL register along with the completion detection circuits is depicted in figure 4.5.

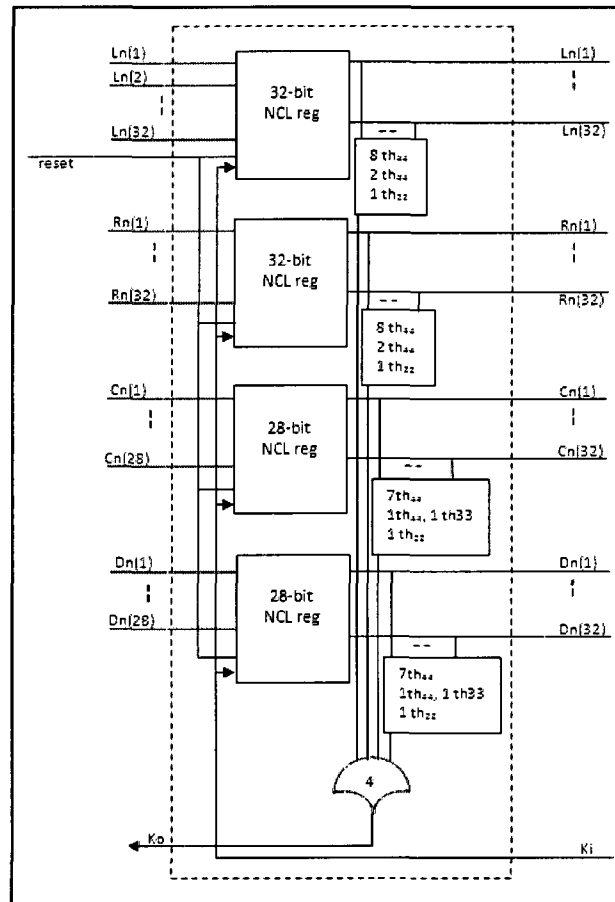


Figure 4.5 The inside view of NCL register 1-15

4.4 Rounds 1-14

Asynchronous DES pipeline has fourteen similar rounds which differ only by two functions to initial round. While the initial round has pt , key as inputs and had to permute its inputs, rounds 1-14 has L_{n-1} , R_{n-1} , C_{n-1} and D_{n-1} as inputs and the rounds don't require the initial permutation IP for pt and PC1 for key and is shown in figure 4.6. All the other functions are similar to initial round. Among these fourteen rounds, some of the rounds require a single bit left shift for the C_{n-1} and D_{n-1} inputs to form C_n and D_n while some

rounds require two bit left shifts. The number of shifts depends on the left shift table as mentioned in Appendix A.

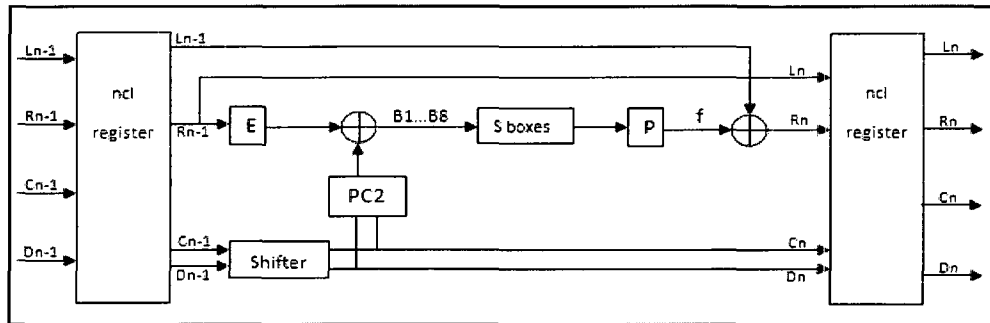


Figure 4.6 Internal structure of rounds 1-14

4.5 Round 15

Round 15 is in no way different to rounds 1-14 with respect to operations performed. But the difference is that round 15 has a slightly different output structure; while rounds 1-14 has L_{n-1} , R_{n-1} , C_{n-1} and D_{n-1} as inputs and L_n , R_n , C_n and D_n as outputs, round 15 has L_{n-1} , R_{n-1} , C_{n-1} and D_{n-1} as inputs but L_n and R_n alone as the outputs. The round need not send C_n and D_n to the next round as it doesn't need them. Because of this reason the register next to this round has a slightly different structure and is mentioned in the next section.

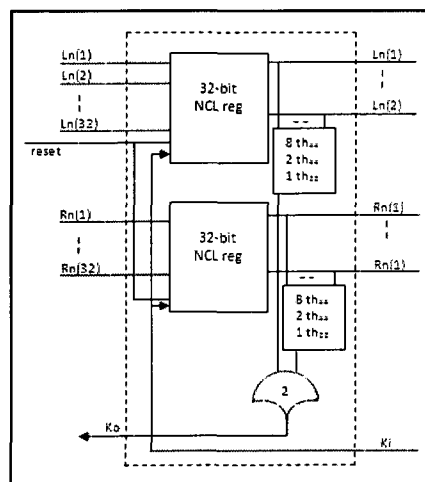


Figure 4.7 NCL register 17

4.6 NCL Register 17

As mentioned in the previous section, this register has L_n and R_n as inputs along with reset and K_i . So the structure of this register looks like the one in figure 4.7.

4.7 Final Round and Final Register

The final round takes the R_n input into L_n and L_n into R_n and permutes the combined result. This round doesn't involve any circuits and is entirely wiring. The output of this round is the cipher text. This output is fed to the final register. Hence, the completion detection block of the final register consists of sixteen th_{44} gates in the first logic level, four th_{44} gates in the second logic level and one th_{44} gate in the third logic level.

4.8 NCL DES Design on FPGAs

The results are in regard with implementation of the whole DES algorithm designed using NCL dual-rail logic, on FPGAs manufactured from different companies and on different FPGAs manufactured from the same company. Here four different FPGAs are selected from two different companies and the details are shown in table 4.1.

Table 4.1 Devices and companies

Device Family	Device Name	Company
Virtex 5	XC5VLX50T-3FF1136	Xilinx
Cyclone II	EP2C35F672C6	Altera
Cyclone II	EP2C70F672C6	Altera
Cyclone III	EP3C25F324C6	Altera

The whole DES algorithm was designed using NCL logic and was coded in VHDL language. The code was run using ISE 9.1i for Xilinx devices and Quartus II Web Edition (v8.1) for Altera devices. The design was simulated and then synthesized and tried to fit on different devices.

4.8.1 Xilinx Device

The Xilinx device used for comparison is Virtex 5 XC5VLX50T-3FF1136. Each configurable logic block (CLB) of virtex 5 has four slices. Each slice has four LUTs (look-up tables) and four registers. The results obtained for Xilinx Virtex 5 device are as follows. Table 4.2 shows some of the internal details of the above mentioned device.

Table 4.2 Virtex5 resources

CLBs	Block RAM	Embedded Multipliers	PLLs	I/O pins
7200	2160 Kb	48 (25 x 18)	6	480

The synthesis report shown in table 4.3 gives the details of the resources occupied by the whole algorithm on the FPGA.

Table 4.3 Resources used by DES algorithm (dual-rail logic) on Xilinx device

Resources	Used by the design	Available in the device	Percentage utilization
No. of slice registers	10899	28800	37%
No. of slice LUTs	30839	28800	107%
No. of bonded IOBs	369	480	76%

The percentage of the resources occupied in the Virtex5 device exceeds 100, which means that the design requires logic resources more than what the device currently have. The design was large for the device to be fit into.

4.8.2 Altera Devices

Three different Altera devices with different number of resources were considered and then compared with respect to the DES algorithm. Each LE (logic element) in the Altera devices contains a four input LUT, a programmable register and interconnects. Table 4.4 shows all the resources available in different Altera devices considered.

Table 4.4: Altera Device Resources

Resources	EP2C35F672C6	EP2C70F672C6	EP3C25F324C6
Logic elements	33216	68416	24624
Block RAM(Kb)	483	1152	594
Embedded Multipliers	35 (18 x 18)	150 (18 x 18)	66 (18 x 18)
PLLs	4	4	4
I/O pins	475	422	215

Among the three devices, EP2C70F672C6 has more number of logic elements. It also has sufficient number of I/O pins. Table 4.5 gives the detailed report on the resources available in the devices and the resources used by the design in respective FPGAs.

Table 4.5: Resources used by DES algorithm (dual-rail logic) on Altera devices

Resource Utilization	EP2C35F672C6		EP2C70F672C6		EP3C25F324C6	
	LEs	I/Os	LEs	I/Os	LEs	I/Os
Used by the design	56817	385	56816	385	56817	385
Available in the device	33216	475	68416	422	24624	216
Percentage utilization	171%	81%	83%	91%	230%	178%

Out of all the devices, Altera's EP2C70F672C6 was able to fit the entire asynchronous DES algorithm. It occupied 83% of its available logic elements and 91% of its available I/O pins. All the other devices need to be accommodated with more resources for the DES algorithm in NCL logic to be fit into. The Xilinx Virtex 5 would require atleast 1039 more slice LUTs for the asynchronous DES algorithm to be implemented on the FPGA although it has enough number of I/O pins. Among the three Altera devices, the Cyclone II devices have enough I/O pins since the design only requires 385 pins. The Cyclone II EP2C35F672C6 still requires 23659 logic elements and Cyclone III EP3C25F324C6 requires 32221 logic elements to accommodate the whole algorithm.

4.9 Improvements in Asynchronous DES Design

The asynchronous DES design using NCL dual-rail logic requires a lot of logic resources. Usually, asynchronous circuits occupy more logic resources compared to their synchronous counterparts as asynchronous circuits have additional circuits for handshaking protocols. Out of the four devices mentioned in the previous section, only one device could accommodate the entire design.

Asynchronous DES design consists of NCL registers, completion detection circuitry and combinational logic also called rounds in the entire design. Out of all the components, the S-boxes present in the rounds of the design consume most of the logic resources as it doesn't involve any logic but would require more logic elements to realize the functionality during mapping the design on FPGAs. Implementing S-boxes using RAM elements embedded in the FPGA is the efficient method to save a large amount of logic resources [7].

4.9.1 Design Modification to Utilize Internal RAM Elements

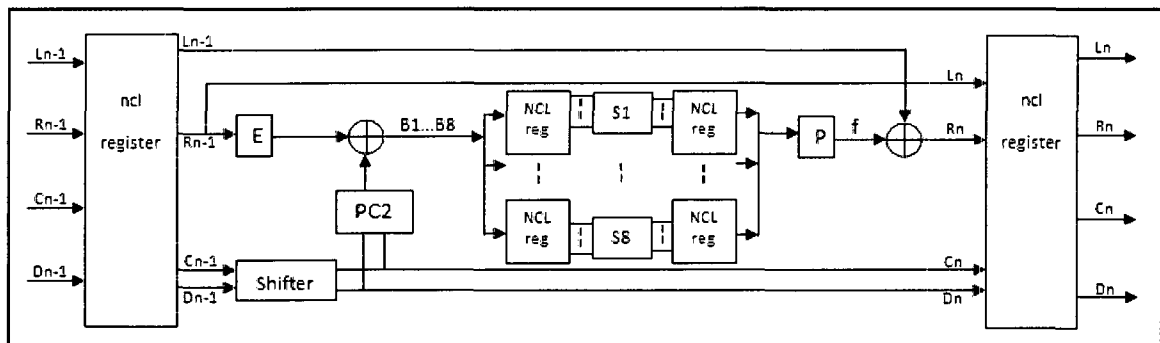


Figure 4.8 Asynchronous DES round with S-boxes as RAM elements

In order to reduce the amount of logic resources used by the design in the FPGA, the DES algorithm using NCL dual-rail logic need to be modified so as to utilize RAM elements. All the eight S-boxes need to be implemented using RAM elements. The

components operate using a clock signal. So the design needs to have a signal which can act as a clock signal to the RAM elements. After the inclusion of S-boxes as RAM elements, a single DES round would look like what is shown in figure 4.8.

The set of eight s-boxes along with two governing NCL registers for each S-box are added to the original asynchronous DES round. Only the S-boxes are realized using RAM elements. The rest of the design is realized using logic elements. So logic elements used to realize the S-boxes were replaced with RAM elements. The internal details of a single S-box is shown in figure 4.9.

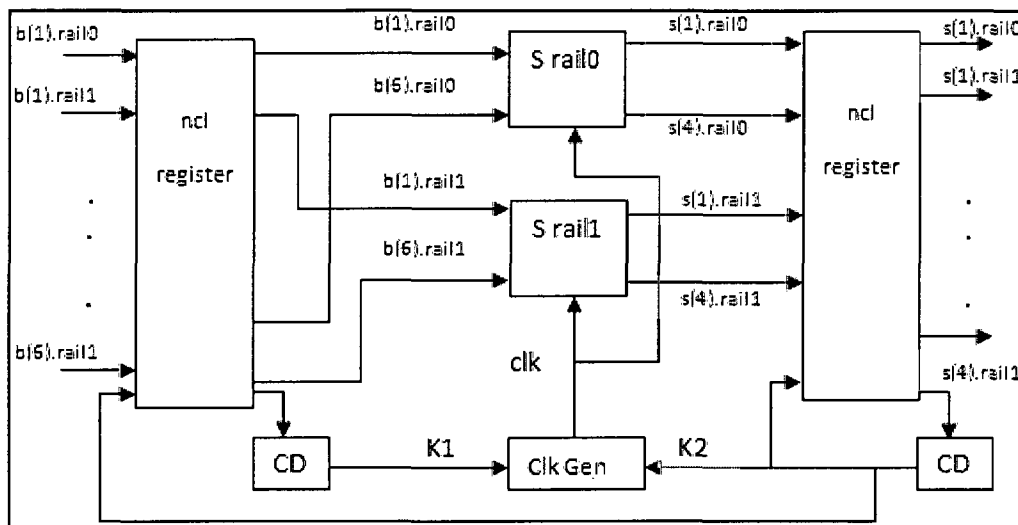


Figure 4.9 S-box as ROM.

As mentioned earlier each S-box is governed by two NCL dual-rail registers which will allow and stop the flow of data (DATA and NULL). In the design, two RAM elements are used; one for rail1 data and the other for rail0 data. The HDL used for the design is VHDL and Altera device is considered and ROM elements are generated using the Quartus II software. The RAM elements internal to FPGA are customized as ROM elements.

The design has two ROMs as two S-boxes; S-box rail1 and S-box rail0 which need a clock for their operation. The NCL registers with their completion detection circuits play a major role in the design. The six bit dual-rail data enters the entry side NCL register goes through the S-boxes splitting into rail1 signals and rail0 signals. The clock generator generates clock signal to the S-boxes according to K1 and K2 signals produced by entry side NCL register and exit side NCL register respectively. S-boxes realized as ROM elements generates output at the rising edge of the clock and this output goes through the exit side NCL register.

When reset is high, both the NCL registers output zeros. The completion detection circuits generate K1 and K2 as 1. The clock generator output is a 0. DATA enters the entry side register, K1 becomes 0, and the present value of K2 is 1 which would generate 1 as the clock signal. During this rising edge of the clock signal, ROM elements output the corresponding output values for the inputs. Now, the output signals pass through the exit side register thereby making K2 value to 0. When K1 is 0 and K2 is 0, the clock generator output will be a 0. When NULL enters the input register, K1 becomes a 1 and K2 value is still 0 and the output of the clock generator will be a 1. NULL enters ROM elements and generates NULL outputs. The output signals go through output register making K2 as 1 and hence generating 0 as the clock generators output. The clock generator circuit functions as an exor gate.

4.9.2 Resource Utilization with RAM Elements

The entire asynchronous DES algorithm including S-boxes as ROM elements is synthesized over some Altera FPGAs and the resource usage results are shown in Table 4.6.

Table 4.6 Resources used by DES algorithm with ROM on Altera devices

Resource Utilization	EP2C35F672C6		EP2C70F672C6		EP3C25F324C6	
	LEs	I/Os	LEs	I/Os	LEs	I/Os
Used by the design	40928	385	40927	385	40928	385
Available in the device	33216	475	68416	422	24624	216
Percentage utilization	123%	81%	60%	91%	166%	178%
Percentage improvement	48%	-	23%	-	64%	-

Table 4.6 gives the results of comparison between Altera FPGAs with asynchronous DES algorithm with and without ROM. The percentage improvement row shows the percentage comparison between the two designs. Again, Altera's EP2C70F672C6 is the only device which was able to contain the entire algorithm design. With the use of ROM, the device could save 23% of its logic elements. While the DES without ROM has only 17% of its logic resources left after the design, DES with ROM has 40% of its resources left which is a considerable improvement. Similarly comparing the two designs on EP2C35F672C6, the DES without ROM occupied 171% of the resources while the DES with ROM occupied only 123% which means there is an improvement of 48% in the logic element utilization. The device EP3C25F324C6 has an improvement of 64% which means another design with double the capacity to asynchronous DES algorithm could be accommodated.

4.9.3 Resource Comparison between Synchronous and Asynchronous Designs

Some of the implementations of synchronous DES algorithms on FPGAs have been cited in section 1.2. In this section the resource utilization comparison is made between synchronous and asynchronous circuits.

Due to the presence of additional circuitry for handshaking protocols such as the NCL registers and completion detection circuitry, asynchronous circuits are undoubtedly huge compared to their synchronous counterparts. Table 4.7 gives the resources utilized

by different synchronous DES designs on FPGAs as well as the resources used by asynchronous design created in the thesis. The resources utilized in Xilinx and Altera devices shows that asynchronous designs are 6-10 times bigger than the synchronous designs. So, these (asynchronous) circuits need to be designed such that they occupy the FPGA resources optimally just like the modification of the asynchronous DES algorithm incorporating RAM elements for the S-boxes. No asynchronous DES design has been implemented on FPGAs till date. The simulation results from this thesis provide a basic idea to design asynchronous DES with NCL dual-rail logic as well as the amount of logic elements required in the FPGAs for the design and will definitely be useful for implementation on FPGAs with asynchronous logic elements as mentioned by Smith [29-31] and others [32-33].

Table 4.7 FPGA resources used by different DES designs

Design by	Device Used	Resources Used	Data Rate(Mbps)
Wong et.al [6]	XC4020E	438 CLB slices	26.7
Kaps & Paar [34]	XC4028EX	741 CLB slices	402.7
McLooney, McCanny [8]	XCV1000	6446 CLB slices	3808
Patterson [9]	XCV150	1584 CLB slices	10752
Standaert et.al [35]	Virtex II Pro	250 CLB slices	1036
Xilinx [36]	XC2V1000	5036 LUTs	15100
Asynchronous DES(thesis)	XC5VLX50T	30839 LUTs	*
Arich et.al [7]	EP1K100FC484-3	5991 LEs	1054.24
Asynchronous DES(thesis)	EP2C70F672C6	56816 LEs	*
Asynchronous DES RAM(thesis)	EP2C70F672C6	40927LEs	*

* The asynchronous DES designed for the thesis generates DATA output for approximately every 360-380ns depending on the input values and the propagation path chosen by the inputs(to be studied in detail).

CHAPTER V

SOFT ERROR AND NCL CIRCUITS

While most of the researchers investigate the soft error issues in traditional synchronous circuits, little attention has been paid to asynchronous circuits. In fact, quasi delay insensitive (QDI) asynchronous circuits have a strong potential for soft error tolerance. The combination of handshaking protocol and dual-rail encoding in QDI circuits provide the circuits with a potential capability to detect and correct the soft errors. Besides single event upsets, particle strikes may cause other malfunctions on a chip: charges induced by particle strikes may slowly accumulate in the substrate of a chip. Those long term dose effects usually cause parameter shifts, in particular threshold voltages, which affect the timing of the system. QDI circuits are very robust to timing variations.

5.1 SEUs in Null Convention Logic

The analysis and estimation of the soft error rate have been extensively studied based on the three maskings which are logical masking, electrical masking and latching window masking [4][37][38]. This section explains the mechanism of soft errors in semiconductor circuits, how the generation and propagation affects the circuits and what kind of soft errors propagate through the NCL pipelines.

5.1.1 SEUs in Semiconductor Circuits

When a neutron particle strikes a CMOS transistor it generates a very high carrier concentration of electron-hole pairs [39] as it loses its energy in silicon with a rate, called stopping power (dE/dx) or linear energy transfer (LET). These electron-hole pairs are subject to drift, diffusion, and recombination. The ratio of the collected to the generated charge is called the collection efficiency. A higher voltage and a larger electric field in the depletion region result in a faster charge collection, creating a larger current transient at that node. An SEU, occurs when enough charge is collected in such a short time to reverse or flip the data of a gate output, memory cell, register, latch, or flip-flop. The transient current due to a particle strike can be modeled as [40]

$$I(t) = \frac{2Q}{T\sqrt{\pi}} \sqrt{\frac{t}{T}} \cdot \exp\left(-\frac{t}{T}\right) \quad (1)$$

where Q is the amount of collected charge, and T is a process technology-dependent time constant. Figure 5.1 shows the mechanism of soft errors in semiconductor circuits along with a transient current plotted for $Q=60fC$ and $T=20ps$.

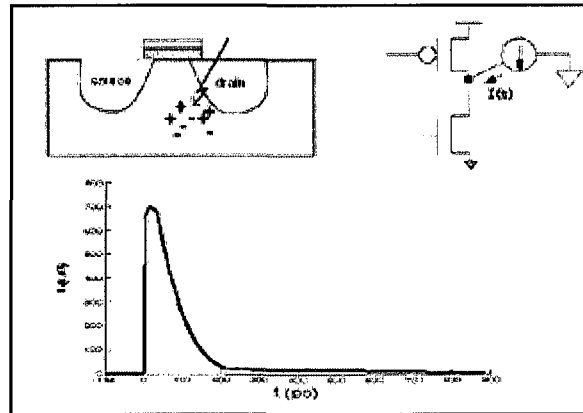


Figure 5.1 Mechanism of soft errors in semiconductor circuits.

Whether the current is injected into or removed from the node depends on the type of victim drain. For example, a current is injected into the node if a particle hit

occurs at a p-type drain, therefore momentarily increasing the node voltage. If the logic value of the node is 0 and the current is injected to the node, a 0-1-0 SEU may occur. Similarly, a 1-0-1 SEU may be generated if an n-type drain is hit.

5.1.2 Generation and Propagation of Soft Errors in NCL

The hysteresis behavior of Threshold Gates and the fact that their input data is encoded using dual-rail encoding makes asynchronous systems more susceptible to soft errors. The specific type of soft error depends on the input pattern, present output, and the location of the particle strike.

Theoretically, there are four types of soft errors that could be generated at the output of a threshold gate. Let us consider a th_{23} gate in figure 2.9 to demonstrate these errors.

1) *Positive glitch (PG)*: 0-1-0. The positive glitch could be generated when the input pattern of the th_{23} gate is $ABC=000$ and a strike occurs into any of $n1$, $n3$ or $n6$ n-type drains (removed current), as show in figure 5.2a.

2) *Negative glitch (NG)*: 1-0-1. The negative glitch could be generated when $ABC=011$ while the output is 1 and a strike occurs into any of $p3$ or $p9$ p-type drains (injected current), as shown in figure 5.2b.

3) *Positive fault transition (PFT)*: 0-1. The positive fault transition could be generated when $ABC=001$ while the output is 0 and a strike occurs into any of $n1$, $n3$, or $n6$ n-type drains (removed current), as shown in figure 5.2c.

4) *Negative fault transition (NFT)*: 1-0. The negative fault transition could be generated when $ABC=001$ while the output is 1 and a strike occurs into any of $p3$ or $p9$ p-type drains (injected current), as shown in figure 5.2d.

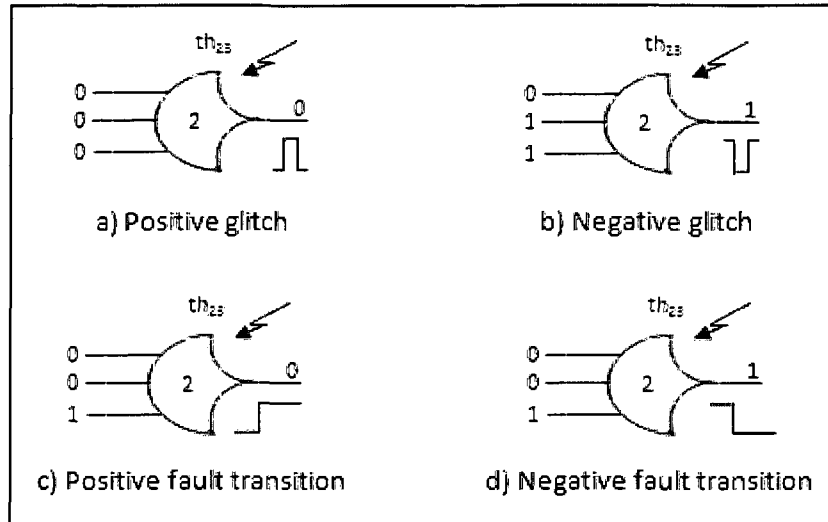


Figure 5.2 SEU generation in th_{23} gate.

Fortunately, only two types of SEUs are to be considered for NCL systems, Positive Glitch (PG) and Positive Fault Transition (PFT), because these are the only two possible SEUs that could flip the state of the node and propagate to the circuit output causing a soft error (malfunction). As for both Negative Glitch (NG) and Negative Fault Transition (NFT) they would only speed up the arrival of NULL if the strike happens after DATA arrival or they might not have any effect if the strike happens before DATA arrival as tested in [41]. The most sensitive node in a threshold gate for both PG and PFT is node S (in figure 2.9), where a soft error could cause either PG or PFT when a particle strikes any n-type drains of the NMOS transistors connected to node S.

5.2 Study of NCL Pipeline for an SEU

Unlike traditional synchronous circuits, there is no global clock in NCL circuits. The delivery of the computation results from one stage to the next stage is implemented by the handshaking scheme. Due to the hysteresis, a generated glitch SEU (0-1-0 or 1-0-1) will be either filtered or transformed into a fault transition (0-1 or 1-0) immediately by the following gate, and then the resulting fault transition (0-1 or 1-0) conditionally

propagates through the next gates. Therefore only fault transitions (0-1 or 1-0) may arrive at the output of computational block. SEUs on computational blocks have only been focused on throughout the thesis.

Due to the handshaking protocols used in the NCL methodology, all the possible circumstances during which the computational block is affected by the soft error must be studied in detail. A particle strike could affect the computational block any time during the 'request data'. The three possible strike timings that can generate a soft error are described below.

- 1) *Before computation completion:* This is the time during which the input register is requesting DATA and partial or complete DATA has already arrived at the input of the computational block until before the correct computation of outputs are done. A strike at this time could generate an error at the output of an input threshold gate leading to its propagation to the next gate and thereby generating a valid faulty output. If this soft error is not detected, it could pass on to the next stages. This situation is illustrated in figure 5.3b in terms of output signals.
- 2) *Exactly during computation completion:* A strike can happen on any of the internal components of the computational block when the input register is requesting data leading to a (1,1) output which is invalid in dual-rail logic as shown in figure 5.3c. The propagation of this invalid output to the next stage may cause a lot of computational errors at that stage.
- 3) *After computation completion:* The possibility of a strike after the computation of correct outputs cannot be ruled out. The next stage can even take this incorrect data if it is still requesting DATA. This incorrect output is shown in figure 5.3d.

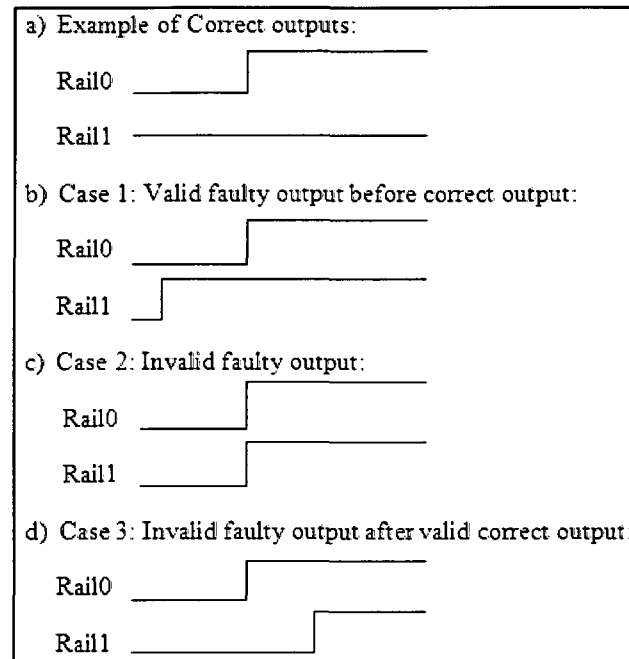


Figure 5.3 Different outputs during different strike timings

Any dual-rail NCL circuit designed to detect, eliminate and correct the soft errors must address the three different scenarios mentioned above and must be able to recompute the correct output.

5.3 Tackling Soft Errors using NCL Methodology

This section describes different methods by which soft errors can be handled so as to avoid them from disturbing the circuit's performance. The second part of this section focuses on different methods and circuits used for soft error hardening, detecting and correcting as well.

5.3.1 Soft Error Mitigation and Correction

Two ways in general, by which a designer can tackle soft errors are soft error mitigation and soft error correction. Soft error mitigation is a method in which a designer can attempt to minimize the rate of soft errors by judicious device design, choosing the

right semiconductor, package and substrate materials, and the right device geometry. Often, however, this is limited by the need to reduce device size and voltage, to increase operating speed and to reduce power dissipation. One technique that can be used to reduce the soft error rate in digital circuits is called radiation hardening. This involves increasing the capacitance at selected circuit nodes in order to increase its effective Q_{crit} value. This reduces the range of particle energies to which the logic value of the node can be upset. Radiation hardening is often accomplished by increasing the size of transistors which share a drain/source region at the node.

Soft error correction is a method where a designer chooses to accept that soft errors will occur, and design systems with appropriate error detection and correction to recover gracefully.

Typically, a semiconductor memory design might use forward error correction, incorporating redundant data into each word to create an error correcting code. Alternatively, roll-back error correction can be used, detecting the soft error with an error-detecting code such as parity, and rewriting correct data from another source. This technique is often used for write-through cache memories.

Soft errors in logic circuits are sometimes detected and corrected using the techniques of fault tolerant design. These often include the use of redundant circuitry or computation of data, and typically come at the cost of circuit area, decreased performance, and/or higher power consumption.

The concept of triple modular redundancy (TMR) can be employed to ensure very high soft-error reliability in logic circuits. In this technique, three identical copies of a circuit compute on the same data in parallel and outputs are fed into majority voting

logic, returning the value that occurred in at least two of three cases. In this way, the failure of one circuit due to soft error is discarded assuming the other two circuits operated correctly. In practice, however, few designers can afford the greater than 200% circuit area and power overhead required, so it is usually only selectively applied.

Another common concept to correct soft errors in logic circuits is temporal (or time) redundancy, in which one circuit operates on the same data multiple times and compares subsequent evaluations for consistency. This approach, however, often incurs performance overhead, area overhead (if copies of latches are used to store data), and power overhead, though is considerably more area-efficient than modular redundancy.

5.3.2 Soft Error Tolerant Schemes in NCL

Several attempts have been made in the past to tackle soft errors in NCL circuits. Some of the methods involve soft error hardening by carefully designing the threshold gates and some of them are designs to detect and correct the occurrence of a strike and are discussed below.

Monnet et.al proposed a metric, sensitive time to evaluate the sensitivity of asynchronous circuits to transient faults [42-43]. Jang et.al proposed several SEU-tolerant QDI circuit designs which cause the circuits to become three times larger and twice slower [44]. Peng et.al developed an efficient concurrent failure detection method for pipelined asynchronous circuits so that the asynchronous circuits halt in the presence of failure by single stuck at faults or single event upsets [45].

Casto, [46] proposed some techniques for preventing soft errors which include the use of Schmitt trigger in threshold gates, feedback transistor sizing and modification of NCL pipeline structures to prevent electrical and latch-window masking. An additional

self-feedback NCL register is inserted before the actual NCL register in the pipeline. This will reduce the amount of time during which a computational block's output will be affected due to soft error by blocking any incorrect outputs generated due to strike once the correct outputs pass through it. The occurrence of outputs as described in figure 6.3d can be completely eliminated using this method.

Kuang, et.al [47] concluded that increasing the feedback PMOS transistor size in the threshold gate can improve the robustness to particle strike and both single and double Schmitt triggers significantly increase the Q_{\max} without static soft error. Kuang, et.al [48] also proposed a modified NCL circuit which includes a self- feedback register that could eliminate most of the SEUs in the computational blocks. In another publication [49], they proposed a soft error detection and correction circuitry for any combinational logic.

Waleed, [41] proved that semi-static gates(in terms of transistors) could be used to construct soft error hardened asynchronous circuits and also proposed a circuit design that can detect, eliminate and correct the soft error.

CHAPTER VI

SOFT ERROR TOLERANT DESIGN USING FPGA

The previous designs that addressed the problem of soft errors are simulated using CADENCE software and are not tested in practical. This section explains in detail the design for soft error tolerance which has been synthesized using FPGA logic. FPGA provided the scope for testing and to demonstrate the design readily since it is reconfigurable logic device. Behavioral model designs are created and mapped onto the FPGA logic and tested for its functionality.

This chapter describes in detail the soft error tolerant circuit, the basic components involved in the design, inducing a strike into the designed circuit, testing the circuit using FPGA and analyzing the simulation and actual results.

6.1 Introduction

Unlike the existing designs that have their threshold gates designed using a set of NMOS and PMOS transistors generating the simulation results, the design in this thesis is designed to be made to work on FPGA to extract the actual outputs from the device. The soft error tolerant design using FPGA is designed in order to demonstrate the functionality of soft error tolerant designs. An idea need to be sought to imitate a particle strike and to induce the strike on to the computational block and to analyze the behavior of the circuit under the influence of these particle strikes. In the thesis the particle strike

The circuit shown in figure 6.1 has separate components each for detecting the occurrence of a soft error, for stopping the error flowing to the next stage and for correcting the faulty output due to the soft error, ensuring correct outputs. Apart from the NCL pipeline which consists of two registers with completion detection circuits that govern the inputs and outputs plus the computational block, it has a set of and gates between the initial register and the computational block, an inverter connected to the initial register's completion detection output, a soft error detection unit and few more and gates providing the control signals to the self-feedback inserted register and the final register.

Initially, the circuit needs to be reset. When 'reset' signal is applied, the NCL registers outputs all zeros. These zeros pass through the completion detection circuits giving '1' as the output indicating that the registers are ready to accept DATA. These zeros (NULL) pass through the and gates and the computational block and prepares then for the next incoming DATA. Now the 'K1' signal mentioned in figure 7.1 is logic 1 which goes through the inverter giving logic 0 for 'K1o' signal. This 'K1o' signal plays a crucial role in determining whether DATA can pass through the self-feedback register or not. The 'K1o' signal holding logic 0 goes to the input of the first and gate at the input of the self-feedback register, which is already having 'KRo' signal as it's another input. 'KRo' signal is the output of the completion detection circuit of the self-feedback register. When 'reset' signal is applied to the self-feedback register, 'KRo' holds logic 1, waiting for the DATA as done by the completion detection outputs of the other two registers in the pipeline. So, at the first and gate, 'KRo' is '1' and 'K1o' is '0' making the output signal 'Ka' hold logic 0. On the other hand, the SE detect unit which is nothing but a th_{22b} gate

had (0, 0) pass through it giving a logic 1 value to the 'se' signal. This 'se' signal along with the 'Ka' is fed to the inputs of the second and gate. 'se' is '1' and 'Ka' is '0', making 'KRi' input of the self-feedback register hold a logic 0 meaning the register is not ready to accept DATA. And finally the and gate at the final register accepts DATA only when 'se' is '1' and 'K3' is '1'. 'K3' is the output of the completion detection circuit of the following NCL register. For convenience, 'K3' is taken as 'K2' in the thesis since there is no next stage. 'K2' is the output of completion detection circuit of the final register. 'K2o' which is the output of the and gate at the input of the final register hold a logic 1 since 'se' is '1' and 'K2' is also '1'. 'se' also acts as the other input to the set of and gates present between the initial register and the computational unit.

Now, DATA enters the inputs. Since 'se' is '1', the exact DATA flows through the and gates and to the computational block. When complete DATA enters the inputs of the initial register, 'K1' becomes a '0' making 'K1o' logic 1. As 'KRo' is already '1', the first and gate at the self-feedback register opens. When there is no strike, 'se' holds a '1' value. With 'Ka' also '1' the second and gate will also be open making 'KRi' a '1' allowing the correct DATA to pass through the self-feedback register. If there is a particle strike 'se' acquires logic 0 turning off the 'KRi' and 'K2o' signals at the self-feedback and the final registers respectively, stopping the flow of the invalid DATA (1, 1). When 'se' is '0', the outputs of the set of and gates at the input of the computational block assumes all zeros which is NULL resetting the computational block. As NULL value pass through the computational unit making it ready for the same input values again, 'se' becomes a '1'. But 'KRi' is still '0'. So the self-feedback register allows NULL to pass through it making 'KRo' take '1'. Now 'KRi' become '1' ready for recomputed correct DATA. Here we have

to make sure that the DATA is still present at the input for recomputation. Once the correct DATA pass through the self-feedback register to the final register, it locks up further changes in the DATA values due to any further strikes.

Summarizing all the facts from the operation of the circuit, the importance of each component is described blow.

- 1) SE detect unit helps in detecting the particle strike by analyzing the outputs of the computational unit. Its gives a '0' output only when it detects invalid data (1, 1) on any of its dual rails.
- 2) Set of and gates are used to pass DATA and NULL through them as usually. When SE detect finds an error and sends a '0' to it these and gates reset the computational block.
- 3) The self-feedback register allows DATA to pass through it only when 'Ki' indicates complete DATA arrival. This register don't allow DATA in three different situations; when there is no complete DATA arrival at the input, when SE detect indicates a particle strike and after the passage of correct DATA through the register. These three situations are governed by the two and gates at the input of the register.
- 4) The and gate at the input of the final register allows DATA only when there is no strike and the next stage is ready to accept DATA.

6.2.2 Case Study: NCL Full-adder as Computational Block

The full-adder circuit is a dual-rail NCL circuit which has three dual-rail inputs and two dual-rail outputs. Since the full-adder has three inputs, the initial register mentioned in the above figure 6.1 is a 3-bit NCL dual-rail register and since the full-adder has two dual-rail outputs, the final register is a 2-bit dual-rail NCL register. The pictorial description of 1-bit NCL register with reset is shown in figure 2.4 and the

VHDL description of 1-bit and n-bit NCL dual-rail registers are done in figures 3.6 and 3.7 respectively.

The completion detection circuitry at the initial register has three inputs and hence is a th_{33} threshold gate and the completion detection circuitry at the final register is a th_{22} threshold gate as it has only two inputs. An n-bit completion detection block is represented in figure 2.5.

A 1-bit NCL full-adder functions just as the conventional synchronous 1-bit full-adder with the exception of having dual-rails for each I/O bit. The schematic of the NCL full-adder is shown in figure 2.6c and the VHDL description of it is shown in figure 3.8b. A sample truth table of the dual-rail 1-bit NCL full-adder is shown in table 6.1. When any of the inputs (C_i , X , or Y) is still Null value (0, 0), the output is not complete, either S or Co will still be Null. When any of the inputs is invalid value (1, 1) and all other inputs are Data, the resulted output will be incorrect (either S or Co).

Table 6.1 Truth Table of a 1-bit full adder with different states

CaseNo	Ci0	Ci1	X0	X1	Y0	Y1	S0	S1	Co0	Co1	State
1	0	0	0	0	0	0	0	0	0	0	NULL
2	0	0	1	0	0	1	0	0	0	0	Incomplete DATA
3	1	0	1	0	0	1	0	1	1	0	Complete DATA
4	1	1	1	0	0	1	1	1	1	1	Invalid DATA

The three different situations that are to be tested to prove that the circuit works as desired in tolerating soft errors are already mentioned in section 5.3. These are, a soft error happening before computation, a soft error happening exactly during output computation and a soft error happening after the output computation. These three scenarios are depicted using the timing of the inputs for the full-adder circuit in figure 6.2.

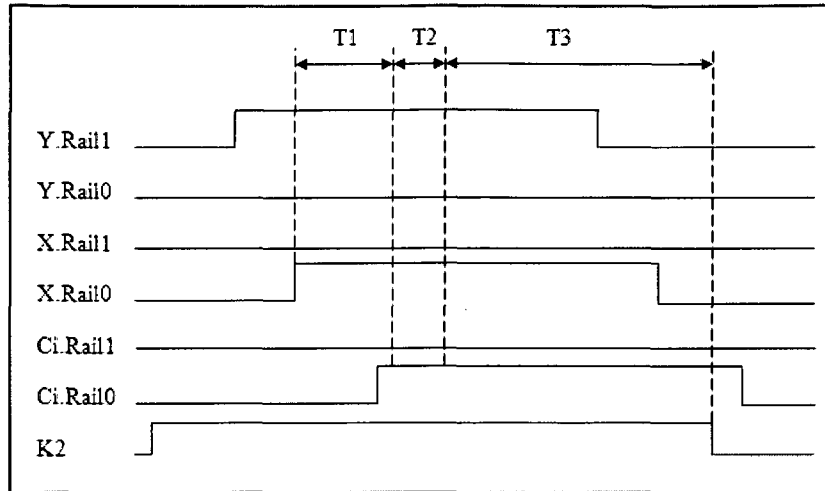


Figure 6.2 Three scenarios to be tested for a full-adder

The figure above is in particular to the full-adder circuit taken to be the computational block in this case. The full-adder has three inputs, each dual-rail and a particular pattern of inputs is selected as an example to clearly explain the threats of this circuit to a particle strike and how they are tackled. 'T1' in the figure is the time during which case 1 in figure 5.3 happens. The inputs are not complete. Due to the input completeness nature of the NCL full-adder, some or all of the outputs will not be complete. This means either S or Co will still be a (0, 0). Carefully looking at the number of inputs during 'T1', it indicates that already two inputs are available and waiting for the third input. At this time a strike on Ci.Rail0 can cause incorrect DATA computation which is Co.Rail1=1 instead of correct DATA being Co.Rail0=1 and Co.Rail1=0. In the absence of soft error detection circuit, this wrong DATA output caused by the strike could pass through the final register to the next stage. Or a (1, 1) is generated on Co dual-rails when Ci.Rail0 comes into the full-adder.

The second criterion 'T2' represents a strike happening exactly during the computation representing case 2 in figure 5.3. This time is after the arrival of complete

inputs and during the propagation of the inputs through the full-adder's internal components. In this case the output of the threshold gate is supposed to output a '0' but apparently becomes a '1' due to the particle strike at the same time when its other dual-rail signal is also a '1'. This condition may arise when the strike modified dual-rail signal (e.g., Co.Rail1=1) and its other dual-rail signal (e.g., Co.Rail0=1) coming from another threshold gate happens to propagate to the outputs at the same time. If this is not avoided, the (1, 1) which is an invalid DATA, propagates to the next stages.

The third case is represented by 'T3' which is when the correct output is calculated and a strike happens after the calculated output pass through the register, generating invalid (1, 1) at the outputs as shown in case 3 of figure 5.3. This situation must be avoided which otherwise would lead to the propagation of (1, 1) right after the correct DATA output in time.

6.2.3 Generating Inputs and Strike

Alpha particles and neutrons are the actual ones which affect the digital circuitry in practical. Due to the non-availability of neutron or alpha particle generator, a particle strike is mimicked using the software. This method also provides the flexibility to test the circuit during different particle strike timings.

Full-adder is the combinational circuit used for testing. The internal components of this full-adder consist of four threshold gates as shown in figure 6.3. As these threshold gates are actually behavioral models, a strike can only be applied at the inputs unlike the transistor built circuits where a strike can be applied to the most sensitive node (node S in figure 2.9) inside a threshold gate. The assumption here is that the strike applied at one of the input rails is equivalent to strike happening anywhere on the internal

circuit of that particular threshold gate present in the FPGA as long as it causes a fault propagation. This assumption is valid because the outputs are the ones which are tested for any faults.

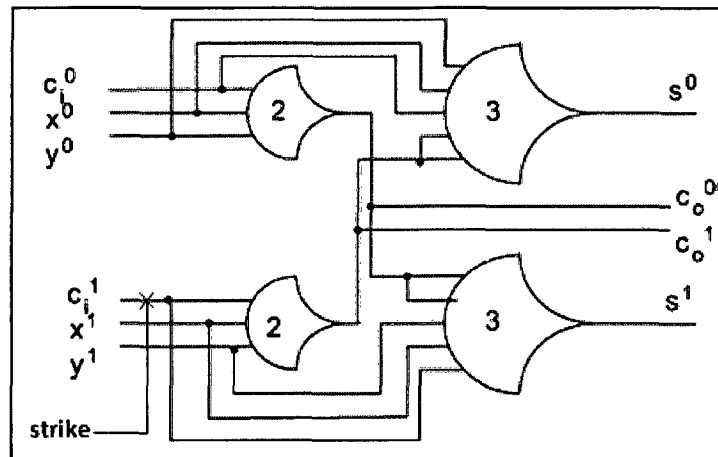


Figure 6.3 1-bit dual-rail full-adder with strike

The above figure suggests that all the three inputs, C_i , X and Y have equal priority due to symmetry. So, a strike on C_i has equal effects when compared to strike on X as well as Y . In this case study a particle strike is made on C_i .Rail1 input of the full-adder.

The full-adder circuit is designed in such a way that it accepts the strike on to the C_i .Rail1 input whenever the strike signal is 'logic1' assuming a strike happened. Otherwise the full-adder takes the C_i .Rail1 input from the initial register. The modified VHDL code for a full-adder accepting a particle strike is shown in figure 6.4. The behavioral model of the strike that is being induced on to the input rail of the full-adder is shown in figure 6.5.

```

Library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.nc1_signals.all;

entity fulladder is
    port ( a : in dual_rail_logic_vector(1 to 3);
          strk : in std_logic;
          s : out dual_rail_logic_vector(1 to 2)
        );
end fulladder;

architecture beh of fulladder is
    signal c0,c1,cstrk : std_logic;
    component strike is
        port( data : in std_logic;
              i : in std_logic;
              ipulse : out std_logic);
    end component;
    component th23x0 is
        port( a: in std_logic;
              b: in std_logic;
              c: in std_logic;
              z: out std_logic );
    end component;
    component th34w2x0 is
        port(a: in std_logic; -- weight 2
              b: in std_logic;
              c: in std_logic;
              d: in std_logic;
              z: out std_logic );
    end component;
begin
    g0 : strike port map(a(1).rail1,strk,cstrk);
    g1 : th23x0 port map(a(1).rail0,a(2).rail0,a(3).rail0,c0);
    g2 : th23x0 port map(cstrk,a(2).rail1,a(3).rail1,c1);
    s(1).rail0<=c0;s(1).rail1<=c1;
    g3 : th34w2x0 port map(c1,a(1).rail0,a(2).rail0,a(3).rail0,s(2).rail0);
    g4 : th34w2x0 port map(c0,cstrk,a(2).rail1,a(3).rail1,s(2).rail1);
end beh;

```

Figure 6.4 Modified full-adder incorporating strike

```

entity strike is
    port( data : in std_logic;
          i : in std_logic;
          ipulse : out std_logic);
end strike;

architecture beh of strike is
begin
    process(data,i)
    begin
        if i = '1' then
            ipulse <= i;
        else
            ipulse <= data;
        end if;
    end process;
end beh;

```

Figure 6.5 VHDL code for inducing a particle strike

The inputs and the particle strike have been generated from internal components present on the FPGA board. In order to test the circuit, a clock signal of 50MHz present

on the FPGA kit has been utilized. The clock output from the oscillator is connected to one of the pins of the FPGA. This has been taken as an advantage to generate the inputs to the full-adder with delay between each of them. The same clock has also been utilized to generate strikes at different timings. Figure 6.6 shows the special code used to generate the inputs and the strike.

```
entity signal_gen1 is
    port( clock : in std_logic;
          Di : out dual_rail_logic_vector(1 to 3);
          strk : out std_logic
        );
end signal_gen1;

architecture behavior of signal_gen1 is
begin
    incrementer: process is
        variable count_value: natural:=0;
        begin
            wait until clock = '1';
            count_value := (count_value+1) mod 16;
            Di(3).rail0<='0';Di(2).rail1<='0';Di(1).rail1<='0';

            case count_value is
                when 1 to 8 =>
                    Di(3).rail1 <='1';
                when others =>
                    Di(3).rail1 <='0';
            end case;
            case count_value is
                when 3 to 10 =>
                    Di(2).rail0 <='1';
                when others =>
                    Di(2).rail0 <='0';
            end case;
            case count_value is
                when 5 to 12 =>
                    Di(1).rail0 <='1';
                when others =>
                    Di(1).rail0 <='0';
            end case;
            case count_value is
                when 3 =>
                    strk <='1';
                when others =>
                    strk <='0';
            end case;
        end process incrementer;
    end behavior;
```

Figure 6.6 Inputs and strike generator

The code in figure 6.6 uses the 50MHz clock which has 20 ns time period to generate the three different inputs with delay. The basic idea is to count the number of clock pulses and making each input start at different clock pulse thereby creating a delay

between them. A strike is also generated in a similar fashion. The minimum width a strike can have using the code is 10 ns. A strike is made to appear at different times and at each possible time and the simulation and actual outputs are obtained.

The above code in figure 6.6 generates the input signals and strike as shown by the simulated results in figure 6.7.

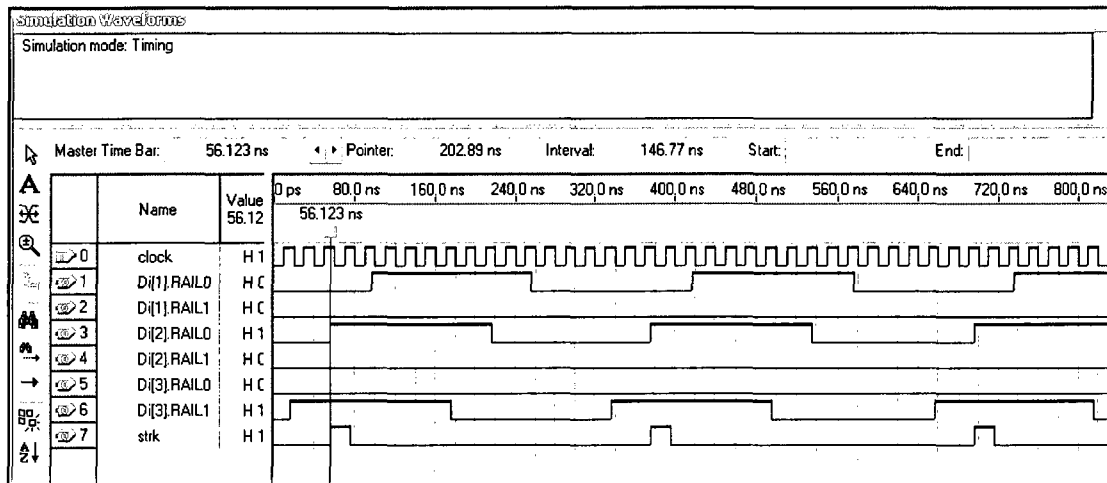


Figure 6.7 Simulation waveform of inputs and strike generator

6.2.4 Simulation Results

Simulation results are obtained from the whole soft error tolerant circuit with full-adder as the computational block, for both cases i.e., without a strike and with a strike. Along with the end outputs which are from the final register, the outputs from the full-adder are also viewed to notice how the outputs from the full-adder are filtered by the additional circuitry that deals with soft error. In the following figures, Qi signals are the outputs from the full-adder and Qo signals are the outputs from the final register. Qi(1) represents Co and Qi(2) represents S directly from the full-adder. Similarly, Qo(1) represents Co and Qo(2) represents S from the final register. Figure 6.8 represents the outputs of the circuit with full-adder when there is no strike. When there is no strike, the

circuit operates normally and doesn't need to reset the computational block and recompute the outputs. Due to this the outputs appear, for example, in this case at 101.75 ns.

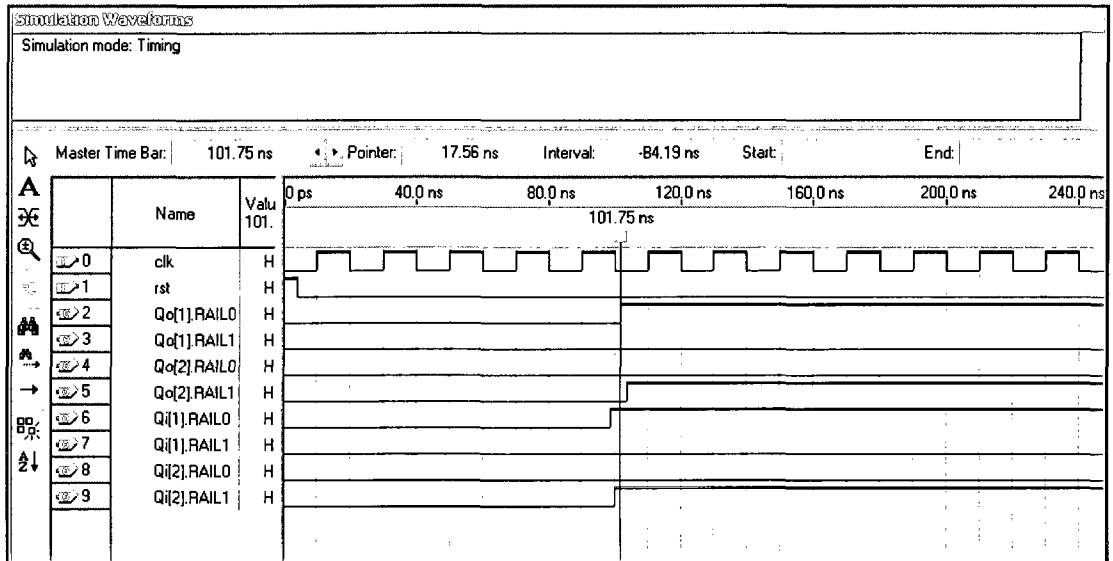


Figure 6.8 Simulation results without a strike.

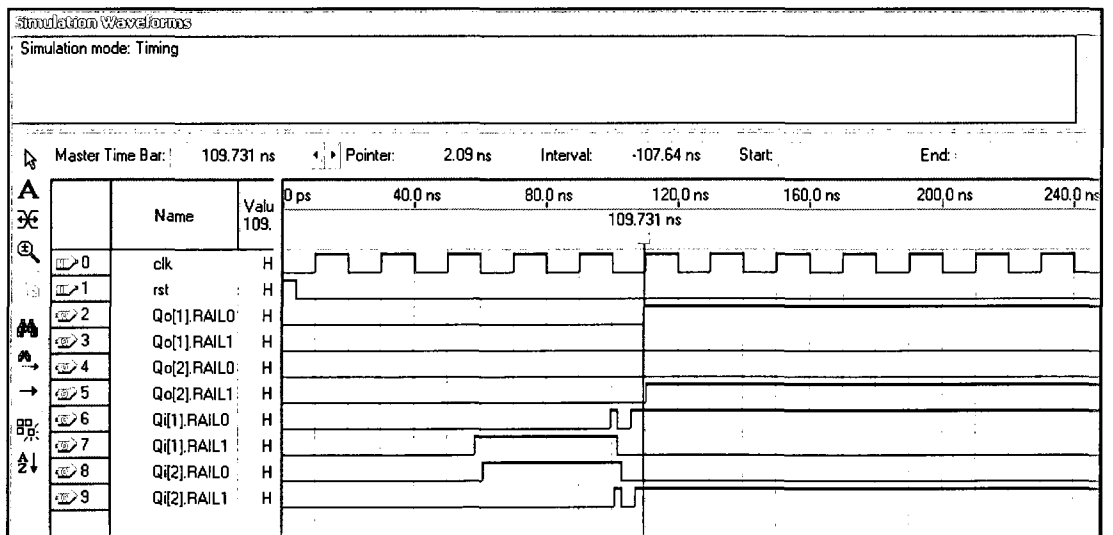


Figure 6.9 Simulation results during a strike at third clock cycle.

Figure 6.9 represents the outputs of the circuit when there is a strike happening at the third clock cycle. In this case, the output appears at a later time compared to the case without a strike. The time between 101.75 ns and 109.731 ns is the time during which the

circuit detects a (1, 1) on the output rails, resets the circuit and recomputes the outputs for the same inputs and hence the delay.

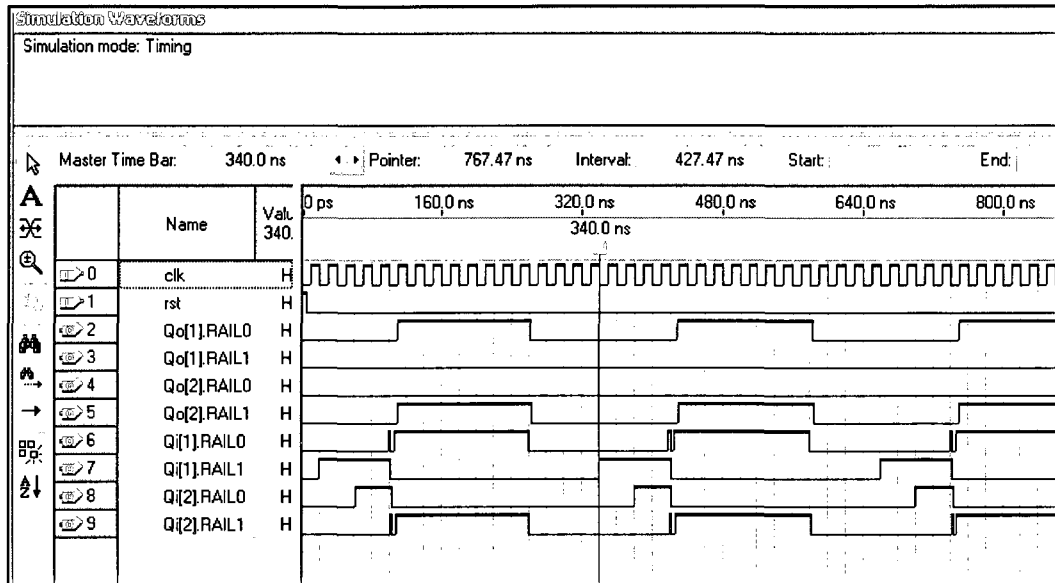


Figure 6.10 Simulation results when strike is placed at the first clock cycle

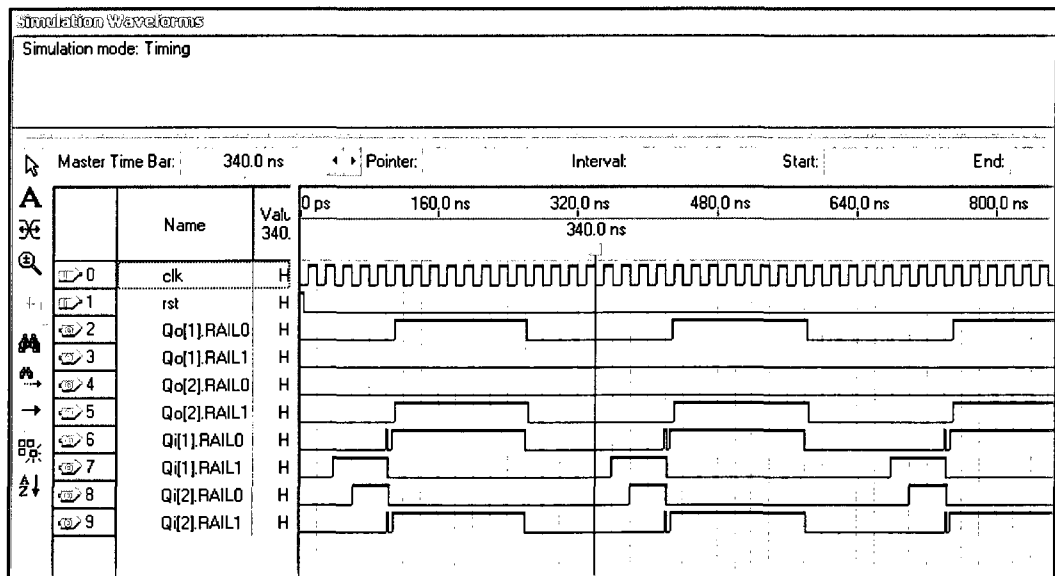


Figure 6.11 Simulation results when strike is placed at the second clock cycle

Figure 6.10 shows that the Co1 signal became a '1' due to the strike long before all the inputs arrive at the full-adder, but has not been passed through the final register. When the circuit detects the (1, 1) visible as glitches in the Qi, it resets the circuit due to

which all the outputs of the full-adder becomes zeros. This final correct output has been delivered to the final register.

The above two conditions come under the first scenario represented as 'T1' in figure 6.2 where only one input, Y.Rail1 is present and due to the strike the Co.Rail1 becomes a '1'. Later when X.Rail0 is asserted, S.Rail0 becomes a '1' which are incorrect outputs generated due to strike. Now, when the third input appears, the signals which should be actually asserted according to the full-adder circuit, that is Co.Rail0 and S.Rail1 becomes '1' showing the little glitches on the waveform editor.

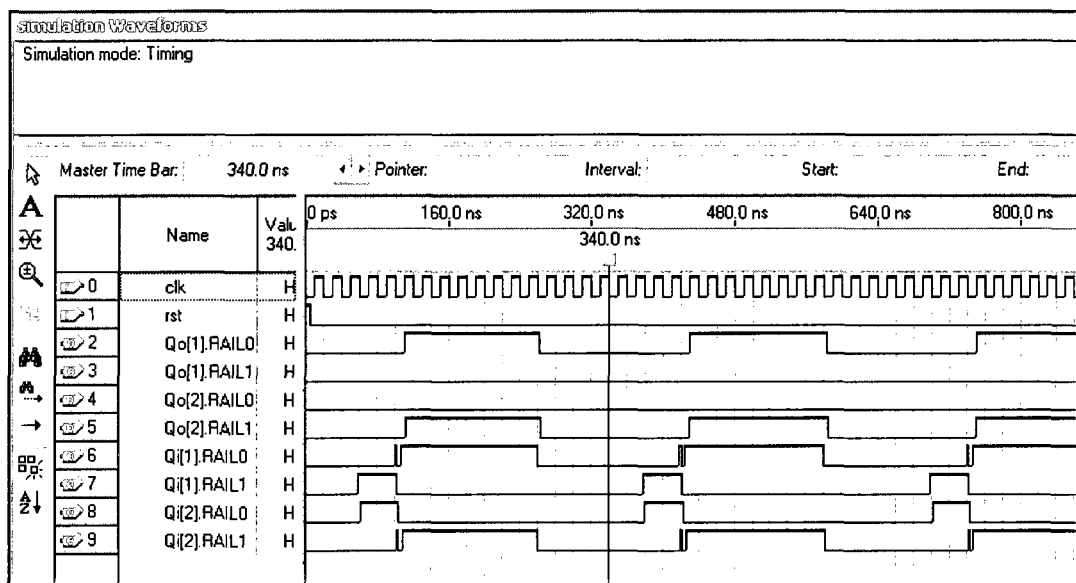


Figure 6.12 Simulation results when strike is placed at the third clock cycle

Figure 6.12 is the result of a strike appearing at the third clock cycle. The second input to the full-adder also starts at the third clock cycle. So by the time the strike makes Co.Rail1 take a '1', X.Rail0 is already '1' and hence the waveforms shows a lesser time difference between Co.Rail1 becoming '1' and S.Rail0 becoming a '1'. The simulation result clearly explains the occurrence of strike at the same time of the second input. The

little delay between Co.Rail1 and S.Rail0 is due to the propagation delay of the th_{34W2} gate which is taking both Co.Rail1 and X.Rail0 to generate S.Rail0 as '1'.

The figure 6.13 shown below is almost a similar situation to the previous figure when there are two inputs present and a strike is appearing. This still falls under 'T1' scenario of incomplete DATA and pre-computation.

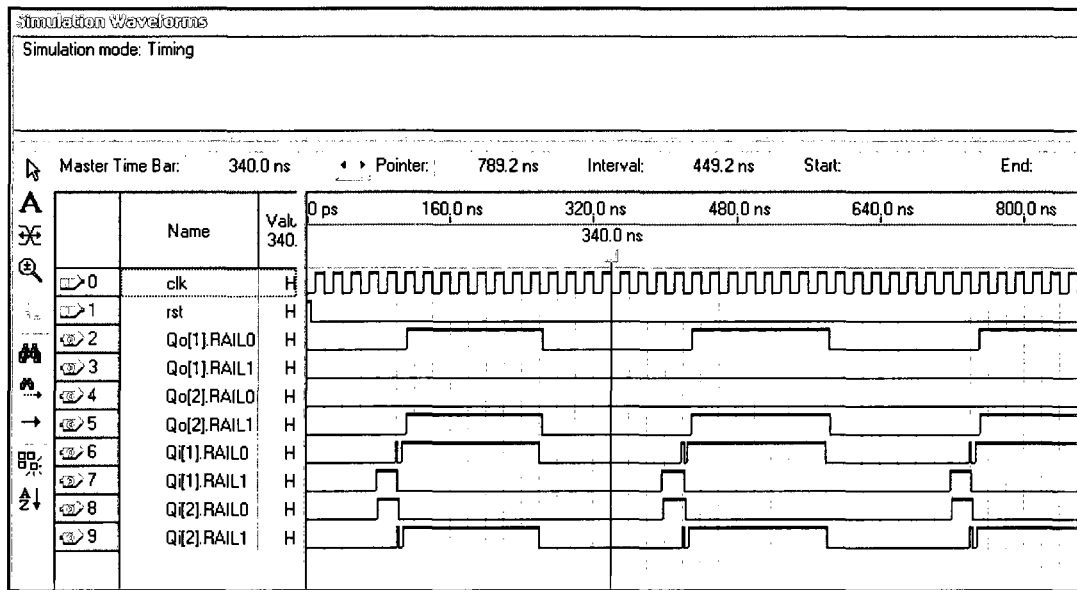


Figure 6.13 Simulation results when strike is placed at the fourth clock cycle

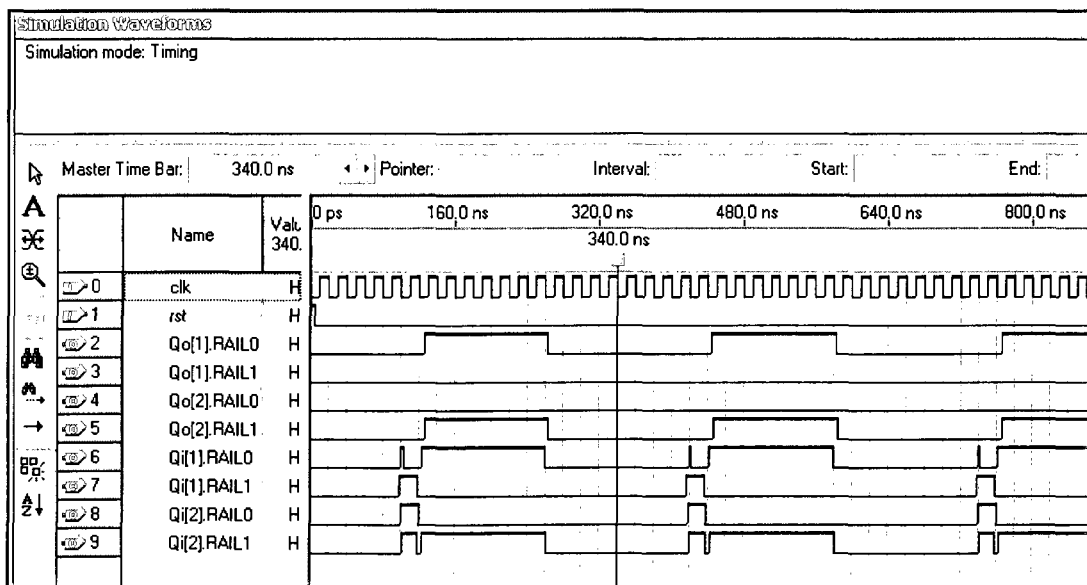


Figure 6.14 Simulation results when strike is placed at the fifth clock cycle

Figure 6.14 is a situation where all of the inputs are present and a strike happens. This comes under 'T2' scenario. Here all the outputs become '1' almost at the same timing. By the time the strike happens, Co.Rail0 is already '1' generated due to the complete inputs and this is still propagating through th_{34W2} threshold gate having S.Rail1 as the output. Due to the strike on Ci.Rail1, Co.Rail1 becomes a '1' and the SE detect detects (1, 1) and resets the circuit due to which Qi(1).Rail0 goes to a zero in figure 6.14. In the mean time the other threshold gates outputs the strike effected signals and then gets reset and now, all the outputs become NULL. As the DATA is still available at the inputs, recalculation is done generating the correct output which passes through the final register.

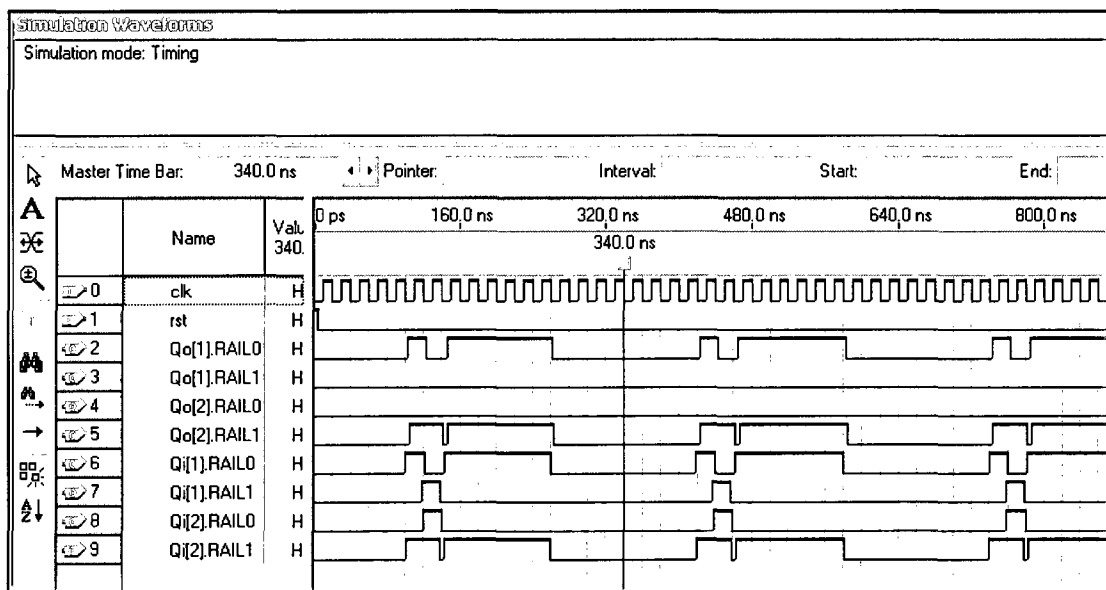


Figure 6.15 Simulation results when strike is placed at the sixth clock cycle

Figure 6.15 is the third scenario where the correct inputs are calculated and they pass through the self-feedback register and then a strike happens. During this case as the correct outputs have crossed the inserted register, it blocks from any further changes in the outputs and the register will be ready to accept NULL. But when a strike happens, as

the DATA is still present at the inputs, a (1, 1) appears on the outputs due to which resetting the circuit takes place internally. This is the reason why the output DATA is normal for sometime but becomes a NULL after sometime and gets back to the correct DATA again as shown in the figure above.

6.3 Experiments on FPGA Device

Now that we have the design on hand along with a provision for particle strike, it needs to be modeled practically to test its behavior. An FPGA provides the scope of creating the design onto its logic and be made available for testing. The design needs to be tested using hardware components, practically giving inputs and extracting the outputs. Once the design is simulated to check its behavior, the design is mapped on to a FPGA device for testing. The design flow is already mentioned in figure 3.3.

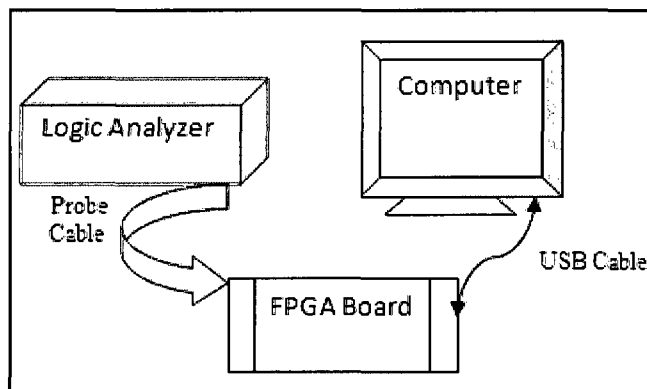


Figure 6.16 Experimental set up

The soft error tolerant design is first simulated and the simulation results are extracted and are presented in section 6.2.4. Then the design is synthesized, mapped and place and route procedure is done to put the design on the FPGA logic. Then pin assignment is done on the FPGA kit to provide inputs and to check the actual outputs. Some of the internal components of the FPGA were used to generate the high speed input signals and the procedure is described in section 6.2.3. The output pins of the FPGA are

connected to a logic analyzer and the outputs are viewed on the display and are provided in section 6.3.3. The experimental set up is shown in figure 6.16.

6.3.1 Experimental Set Up

This section explains about the equipments used for the project and the experimental set up. The entire experimental set up at the laboratory is shown in figure 6.17, consists of a PC (personal computer) with USB cable connected to the CPU, FPGA board whose expansion connectors are connected to probe cable of the logic analyzer.

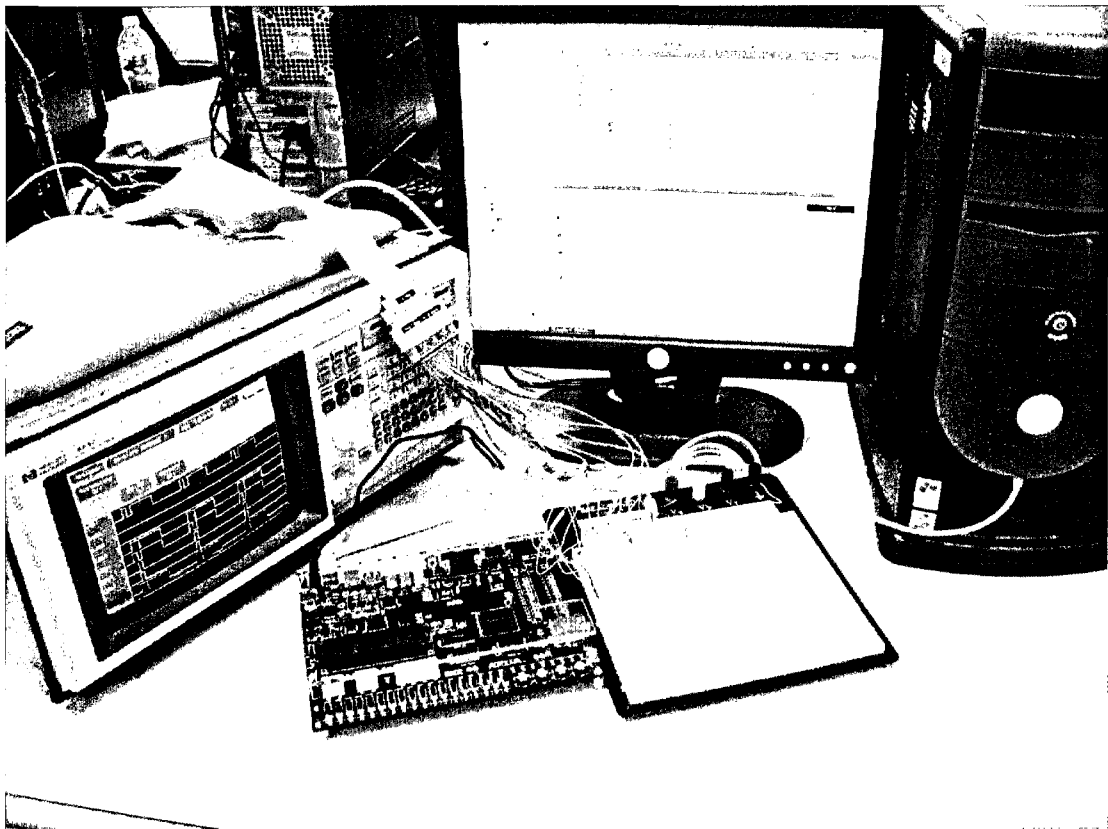


Figure 6.17 Laboratory experimental set up

A PC is a desktop computer used to download the Quartus II software, to write the essential code and simulating the code using the software. Most part of the project is performed on the PC starting from design entry, analysis, implementation, verification and till pin assignment using pin planner present in Quartus II. Altera's DE2 with

Cyclone II FPGA and Quartus II 8.1 Web Edition, the supporting software has been used for the thesis. The device manual is presented in [50]. Once the pin assignments are done, the FPGA device is connected to the PC via a download cable. USB 2.0 (type A to type B) cable is used for the device. The configuration file is sent from the PC to the FPGA via the USB cable. The inputs to the circuit present inside the FPGA are provided using one of the switches and the clock of 50MHz present on DE2 board. The outputs are extracted from the FPGA through expansion headers present on DE2 board on to a logic analyzer. HP 1663C Logic Analyzer is used in the experimental set up to view the outputs of the design present in the FPGA.

6.3.2 FPGA Board

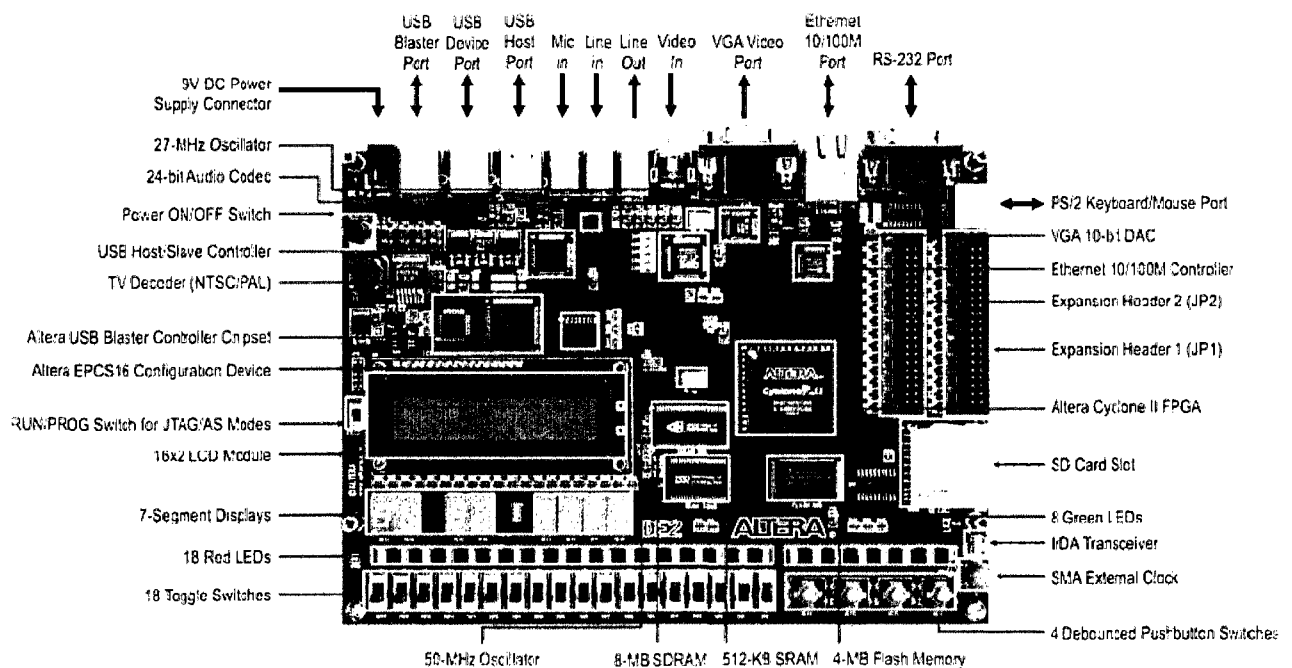


Figure 6.18 The DE2 board [50]

As mentioned earlier Altera's DE2 Development and Educational Board is used for the thesis. It has a wide variety of features and resources. But only few of them have

been used in the thesis. Figure 6.18 taken from DE2 user manual shows all the resources present on board.

The resources used in the above figure are the Altera Cyclone II FPGA, 50 MHz oscillator, one of the 18 toggle switches, Expansion Header JP2, USB Blaster Port, 9V DC Power supply connector and Power ON/OFF Switch. 50MHz clock is connected to PIN_N2 of the FPGA. 'rst' signal in the circuit which is the reset signal to the entire NCL dual-rail circuits is connected to PIN_N25 of the FPGA which is connected to the switch SW0 of the 18 toggle switches on board. PIN_K25, PIN_K26, PIN_M22 and PIN_M23 are the FPGA pins connected to IO_B0, IO_B1, IO_B2 and IO_B3 expansion header pins on JP2(GPIO_1). These pins are connected to Qo(1).Rail0 (Co0), Qo(1).Rail1 (Co1), Qo(2).Rail0 (S0) and Qo(2).Rail1 (S1) signals of the internal circuit, that are the outputs from the final register in the soft error tolerant design pipeline. Similarly, PIN_M19, PIN_M20, PIN_N20 and PIN_M21 are the FPGA pins connected to IO_B4, IO_B5, IO_B6 and IO_B7 expansion header pins on JP2(GPIO_1). These pins are connected to Qi(1).Rail0 (Co0), Qi(1).Rail1 (Co1), Qi(2).Rail0 (S0) and Qi(2).Rail1 (S1) signals of the internal circuit, that are the outputs from the full-adder in the soft error tolerant design pipeline.

6.3.3 Results

The actual outputs are from the FPGA using hardware components that generate the inputs and extract the outputs to the logic analyzer. The below are the images from the logic analyzer for strikes during different timings which resemble the simulation outputs. These actual outputs from the FPGA are very much similar to the simulation results except that the glitches present in the simulation waveform are not visible for the

actual waveform. The reason behind this could be the low resolution of the logic analyzer which could not properly capture the small glitches.

The first signal mentioned as Lab10 represents Qo(1).Rail0, Lab11 as Qo(1).Rail1, Lab12 as Qo(2).Rail0, Lab13 as Qo(2).Rail1, Lab14 as Qi(1).Rail0, Lab15 as Qi(1).Rail1, Lab16 as Qi(2).Rail0 and Lab17 is for Qi(2).Rail1. Qo is the output from the final register and Qi is the output of the full-adder. Q(1) represents Co which is carry and Q(2) represents S which is sum of the full-adder.

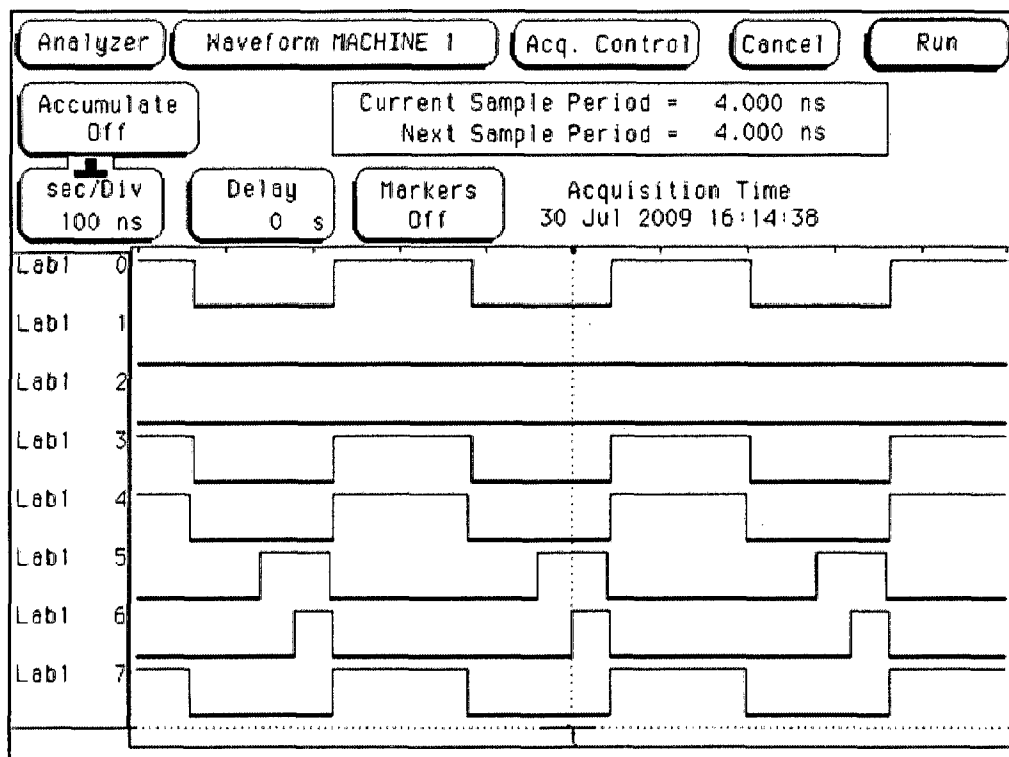


Figure 6.19 Actual results when strike is placed at the first clock cycle

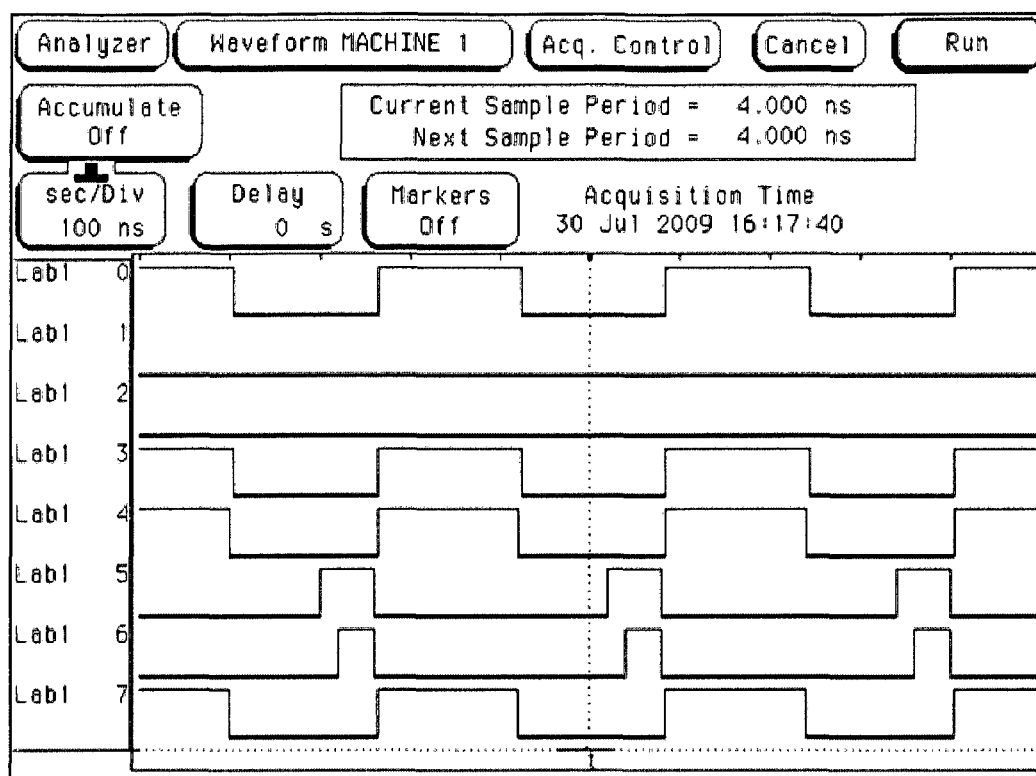


Figure 6.20 Actual results when strike is placed at the second clock cycle

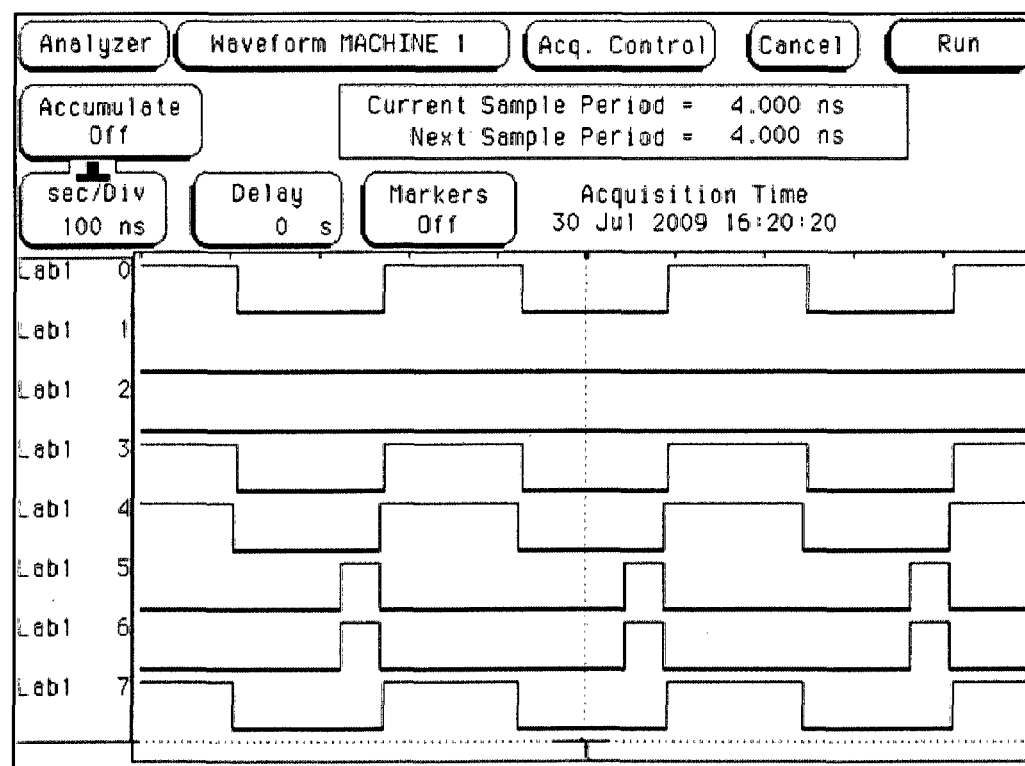


Figure 6.21 Actual results when strike is placed at the third clock cycle

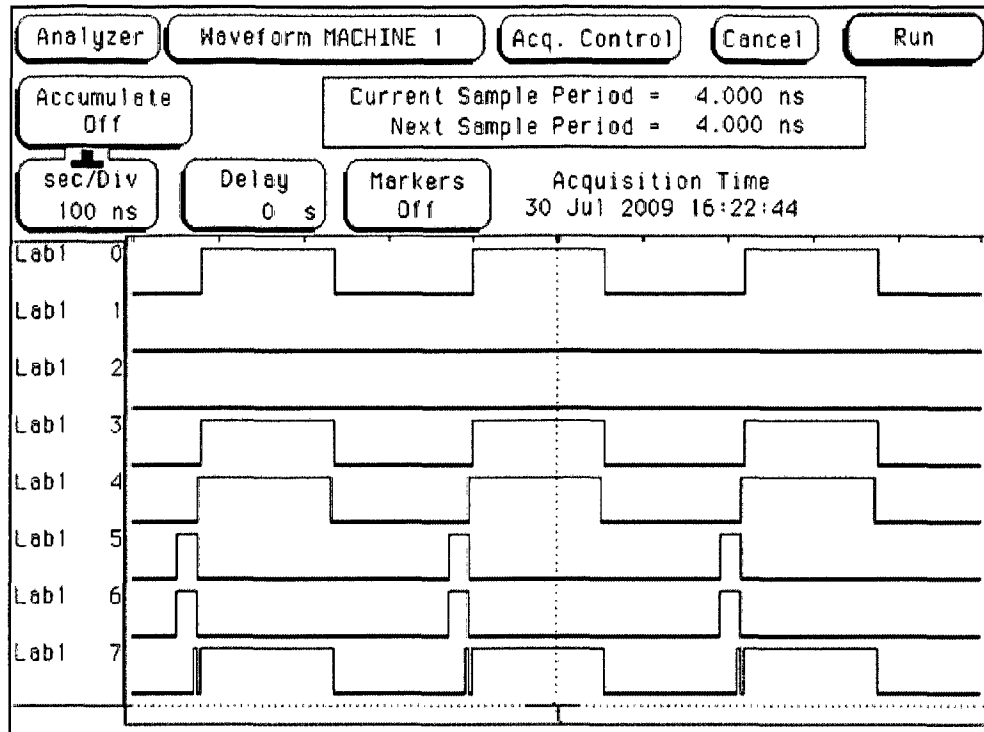


Figure 6.22 Actual results when strike is placed at the fourth clock cycle

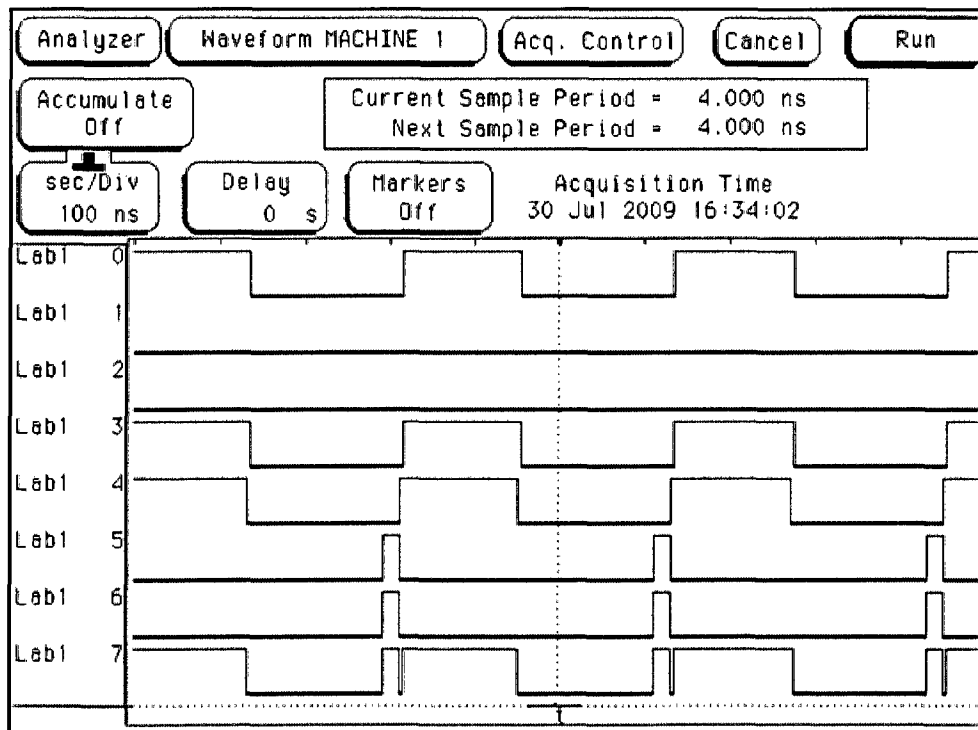


Figure 6.23 Actual results when strike is placed at the fifth clock cycle

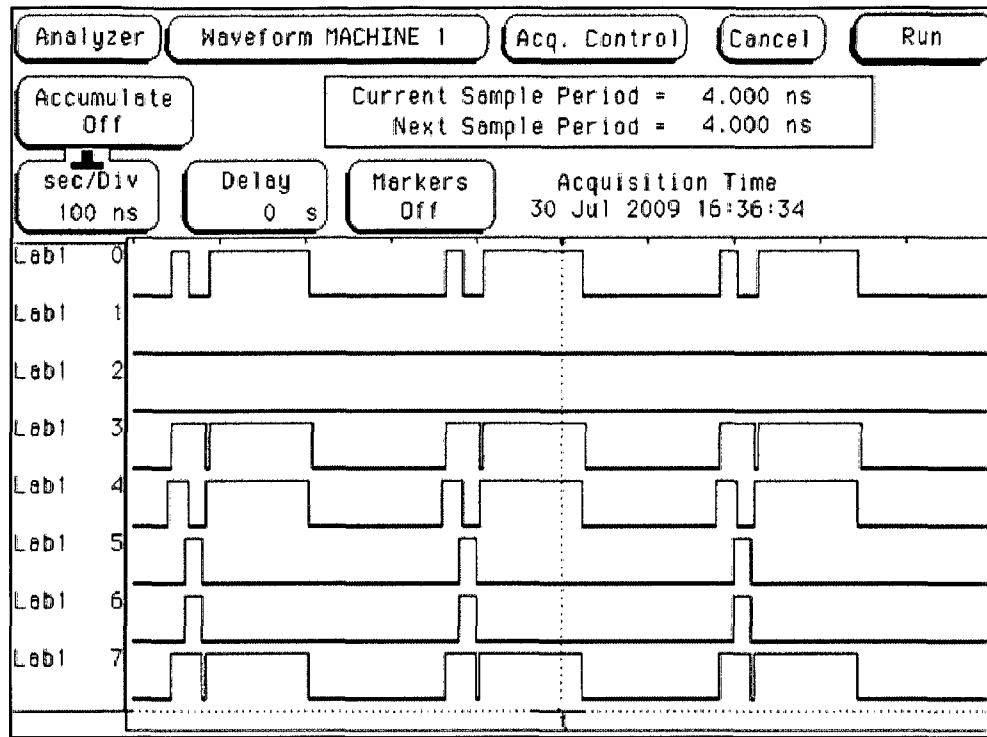


Figure 6.24 Actual results when strike is placed at the sixth clock cycle

CHAPTER VII

SOFT ERROR TOLERANT ASYNCHRONOUS DES DESIGN

In the thesis two asynchronous NCL designs are devised. One is the asynchronous NCL dual-rail logic DES algorithm and the other design is the soft error tolerant circuit for any NCL dual-rail circuit. This chapter explains how to implement the two circuits together on an FPGA.

7.1 Asynchronous DES with Soft Error Tolerance

The 17 stage pipelined asynchronous DES algorithm is added with the soft error detection and correction circuitry in one of the exor gates following the P box in figure 4.6 and the simulation and actual results are obtained from the FPGA. Altera's EP2C35F672C6 which is a cyclone II FPGA, is the device used. Since this device cannot accommodate the whole asynchronous DES design with dual-rail logic, only a single round is used. The modified asynchronous DES design with RAM elements as S-boxes has been used to test the soft error tolerant design's efficiency which successfully produced desired results representing the robustness of the soft error tolerant design.

A clock oscillator of 50MHz on board the FPGA has been used to generate inputs and strike in the similar manner as generated for testing the soft error tolerant design with full-adder as the computational block. The strike has been induced on to the exor gate at different timings and the results are collected. Figure 7.1 is the entire circuit implemented on the FPGA to show the working of soft error tolerant design on asynchronous DES.

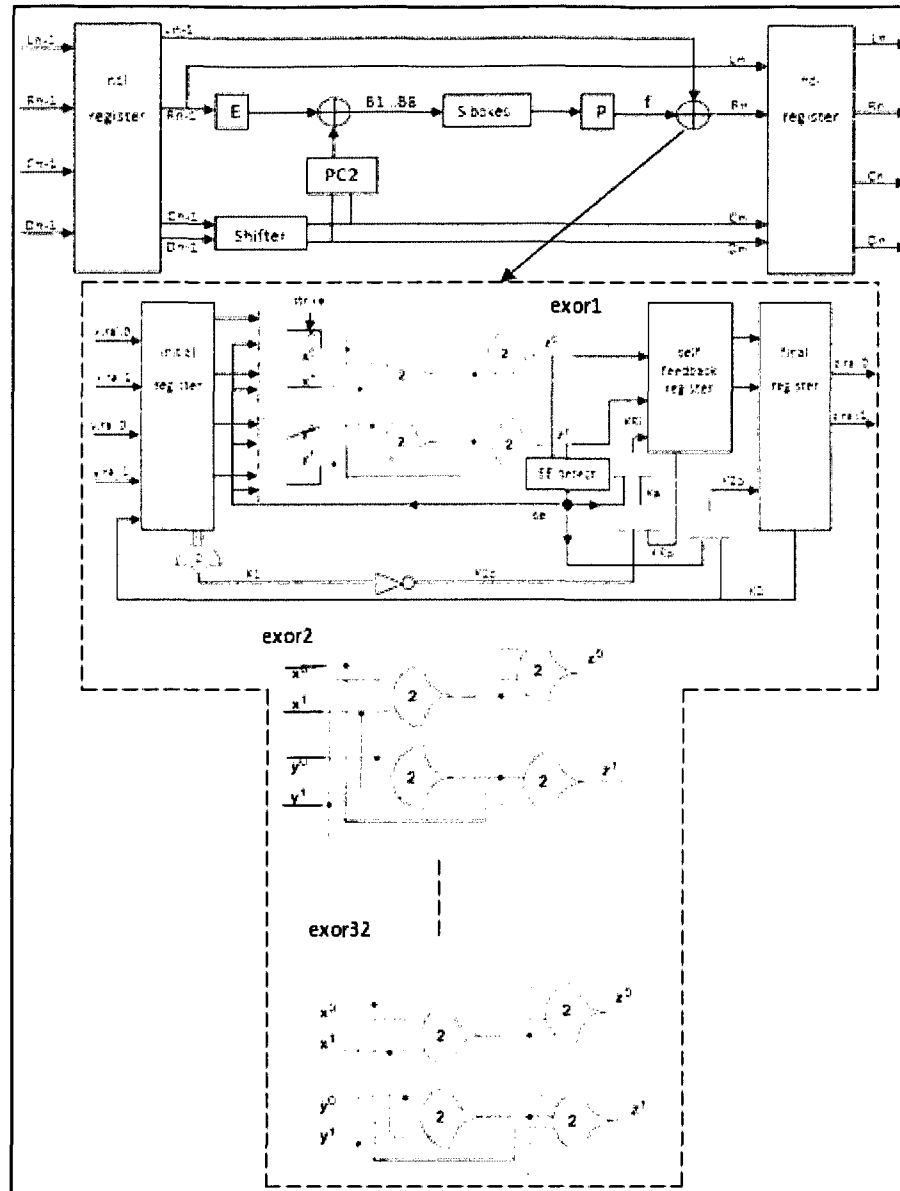


Figure 7.1 Asynchronous DES with embedded soft error tolerant circuit

The Xor gate following P permutation block has 32 exor gates as shown in the above figure. Soft error tolerant circuit is applied to only the first exor gate. The inputs to 'Round' are provided by the clock. To simplify the input generation all the inputs are given at a time instead of generating them with different delays between them due to the large number of inputs. The strike is given at different clock cycles of the 50MHz clock.

7.2 Results Obtained

In this section the result obtained from circuit shown in figure 7.1 are analyzed. The circuit worked as desired by detecting, correcting and eliminating the soft error.

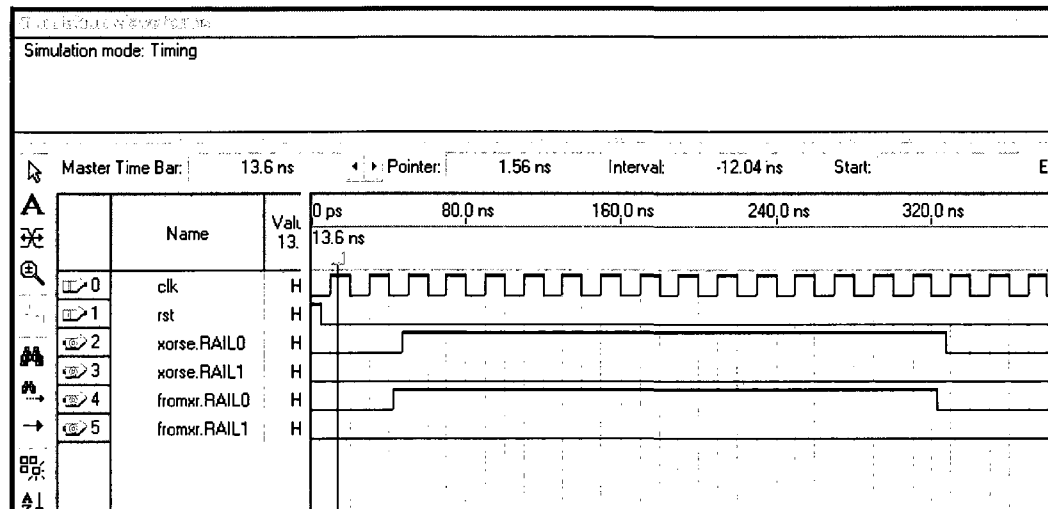


Figure 7.2 Simulation result for strike at first clock cycle.

‘xorse’ signal which is $R_n(1)$ is the output obtained from the final ncl-register of the ‘round’ while ‘fromxr’ signal is directly from the exor1 gate in figure 7.1. Figure 7.2 is the case when the inputs to the exor gate are still propagating when the strike happened and reaches the exor gate when there is no strike due to which there is no effect of soft error reflected in the waveform.

Figure 7.3 is the actual results obtained with similar condition in figure 7.2. Here, strike happens exactly during computation generating a (1,1) because of which the outputs of the exor1 gate are reset to zeros and then the output appears at the final register of the ‘round’. In figure 7.3, the first two signals Lab10 and Lab11 are xorse.Rail0 and xorse.Rail1 respectively while Lab12 and Lab13 are fromxr.Rail0 and fromxr.Rail1 respectively.

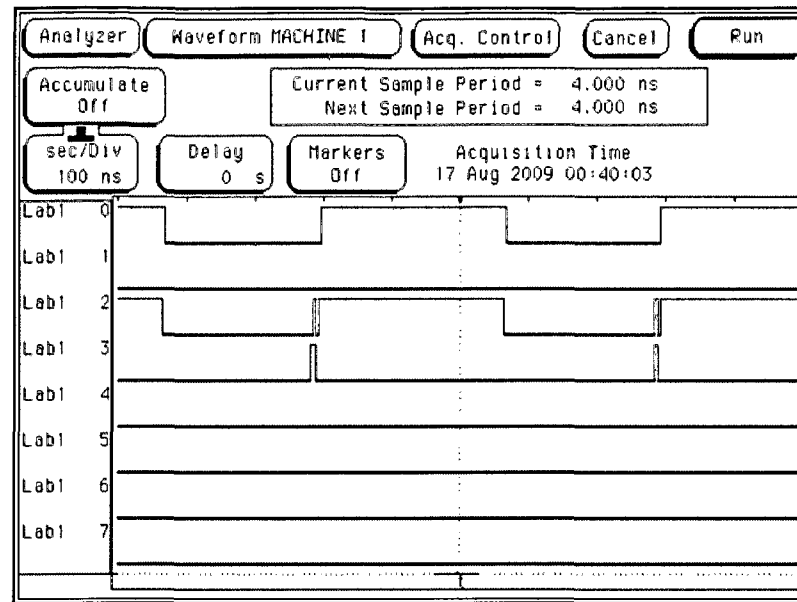


Figure 7.3 Actual results for strike at first clock cycle.

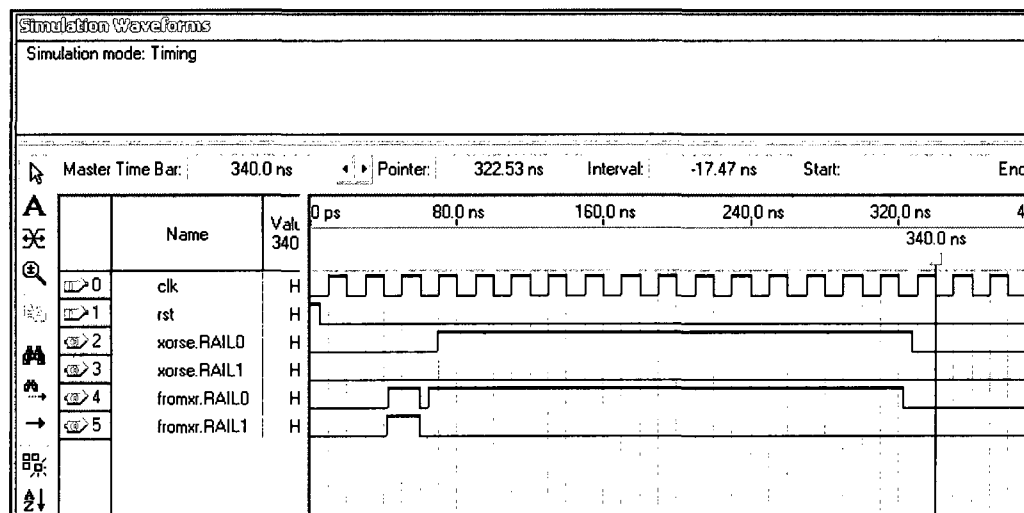


Figure 7.4 Simulation result for strike at second clock cycle.

This result is similar to figure 7.3 where a strike happened exactly during computation causing (1,1) and settling to good outputs after resetting circuit.(1,1) is stopped from passing through the final register.

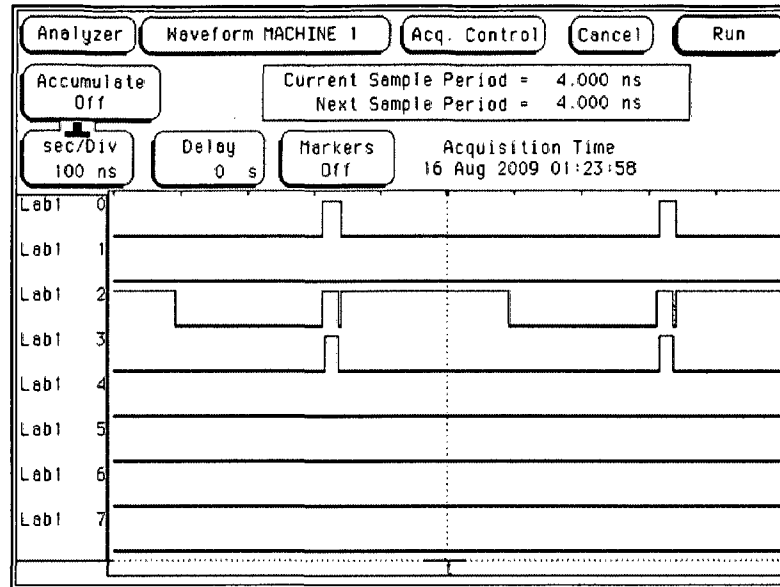


Figure 7.5 Actual result for strike at second clock cycle.

Here, the correct data passed through the self-feedback register in the soft error tolerant circuit and then a strike happened due to which (1,1) is generated, which is the case of a strike happening after computation completion unlike the simulation result. After this output go through the final register of the 'Round' followed by all the other signals of the DES round (L_n , R_n (except the first signal), C_n and D_n), the final register will stay in accepting NULL state since all the DATA passed through it. Now NULL is generated at the first signal of R_n (output of $exor1$) due to circuit reset by the soft error tolerant circuit. As the final register is in accepting NULL state it accepts the NULL signal at $R_n(1)$, while all the other signals $R_n(2..32)$ are still data because there is no NULL at the inputs of the 'Round'. This is the reason why recomputed DATA is not present at the $R_n(1)$ output at the final register.

Figure 7.6 and 7.7 are the results when the strike happens during third clock cycle. This is already the case of strike happening after computation completion allowing good DATA to pass through the register and blocking any further changes.

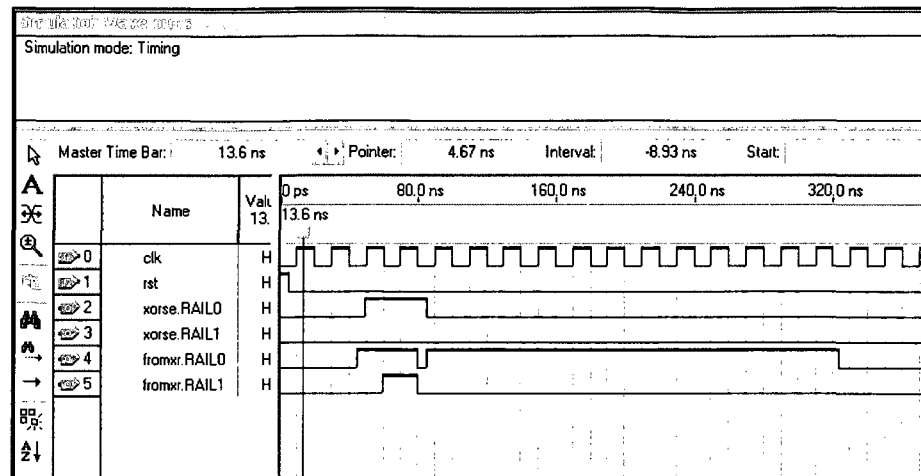


Figure 7.6 Simulation result for strike at third clock cycle.

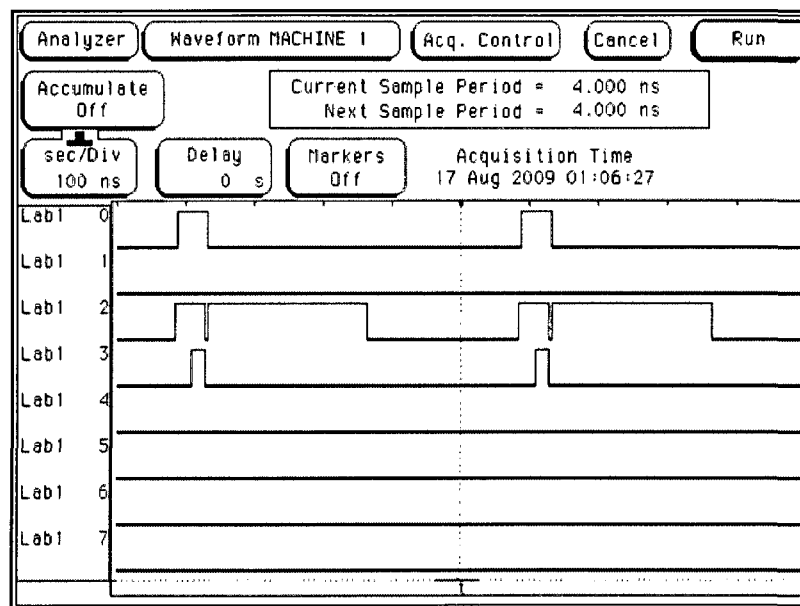


Figure 7.7 Actual result for strike at third clock cycle.

The reason behind the differences in the simulated results and actual results is due to the more propagation delay of simulated signals compared to actual signals in the FPGA due to which figure 7.2 has no effect on the waveforms while figure 7.3 has effect of soft error on the waveforms. Similarly in figure 7.4 the outputs appear at the final register only after resetting and recomputation of the soft error tolerant circuit while correct DATA already passed through the register in figure 7.5.

CHAPTER VIII

CONCLUSION AND FUTURE WORK

The present day digital era calls for cryptography as an inevitable concept in the day to day life whether it is storing data or transferring data. These cryptographic algorithms need a very high range of accuracy. The problem of soft errors cannot be overlooked upon. So circuits which could be able to tolerate these soft errors are as important for applications like security algorithms as these security algorithms are for digital data. Asynchronous dual-rail NCL methodology is a technique which is getting its popularity for various reasons. This methodology could be taken for advantage in creating the soft error tolerant models. So the application on which the soft error tolerant design needs to be implemented should also be an asynchronous design. In this thesis a basic yet powerful security algorithm has been considered. The NCL design of the DES algorithm is implemented on different FPGAs to figure out what kind of FPGA is suitable for such huge designs and to determine the amount of logic and hardware resources needed for the device. Apart from the implementation of the algorithm, some design techniques are used to save the amount of resources utilized by the design on an FPGA.

The second part of the thesis focuses on creating a soft error tolerant design that could be used for these kinds of security algorithms making them 100% accurate. A design which can tolerate the soft errors is simulated as well as practically implemented on the FPGA extracting the outputs in real-time with satisfactory results. This soft error

tolerant design is added to one exor gate in the DES round and the results are discussed.

The designs created in the thesis would help to do some future work on the asynchronous circuits. The asynchronous DES design in the thesis could be implemented on an FPGA and the performance parameters such as speed, power consumption, etc could be measured.

The soft error tolerant design could be embedded in the asynchronous DES algorithm [51] generated in the thesis and this collective design should be exposed to a natural particle strike environment [52] [43] and the design performance could be measured.

Another task that needs to be performed is to actually create the soft error tolerant design at the transistor level. And this circuit could be verified in practical.

Also, the generation and propagation of soft errors on FPGAs need to be studied and a technique to combat those could be devised.

REFERENCES

- [1] Data Encryption Standard – Wikipedia
http://en.wikipedia.org/wiki/Data_Encryption_Standard
- [2] Field-Programmable Gate Array - Wikipedia
http://en.wikipedia.org/wiki/Field-programmable_gate_array
- [3] S.Hauck, “Asynchronous Design Methodologies: An Overview,” *proceedings IEEE, Vol. 83, pp 69-93, Jan 1995.*
- [4] P. Shivakumar, et al., “Modeling the Impact of Device and Pipeline Scaling on the Soft Error Rate of Processor Elements,” in *Proc. Int. Conf. Dependable Syst. Netw.*, 2002.
- [5] U.S. Department of Commerce/National Institute of Standards and Technology, “Data Encryption Standard (DES),” *Federal Information Processing Standards Publication*, Reaffirmed 25, October 1999.
- [6] K. Wong, M. Wark and E. Dawson, “A Single-Chip FPGA Implementation of the Data Encryption Standard (DES) Algorithm,” *IEEE, Global Telecommunications Conference, Nov 1998.*
- [7] T.Arigh and M.Eleuldj, “Hardware Implementations of the Data Encryption Standard,” *IEEE Microelectronics, the 14th International Conference, Dec 2002.*
- [8] M.McLoone and J.V.McCanny, “High-performance FPGA Implementation of DES using a Novel Method for Implementing the Key Schedule,” *IEE Proc.-Circuits Devices Syst., Vol. 150, No. 5, Oct 2003.*
- [9] C.Patterson, Xilinx Inc, “High Performance DES Encryption in Virtex FPGAs using Jbits,” *IEEE Symposium on FP Custom Computing Machines, California, April 2000.*
- [10] I.E.Sutherland and J.Ebergen, “Computers without Clocks”, *Scientific American PP. 62-69, Aug 2002.*
- [11] International Technology Roadmap for Semiconductors (ITRS)
<http://www.itrs.net/Links/2005ITRS/Design2005.pdf>

- [12] N.Weste, D.Harris, "CMOS VLSI Design: A Circuits and Systems Perspective," 3rd Edition. Addison Wesley. 2005.
- [13] J.Wilkinson, et.al. "Cancer Radiotherapy Equipment as a cause of Soft Errors in Electronic Equipment," *IEEE Trans Device and Material Reliability*, Vol 5, No 3, PP-449-51, Apr 2005.
- [14] L.Franco, et.al. "SEUs on Commercial SRAM, Induces by Low Energy Neutron Produced at a Clinical Linac Facility," *RADECS proceedings*, Sept 2005.
- [15] Soft Error – Wikipedia
http://en.wikipedia.org/wiki/Soft_error#Soft_error_rate
- [16] C.L.Chen and M.Y.Hsiao, "Error-Correcting Codes for Semiconductor Memory Applications: A State-of-the art Review," *IBM J.Res.Develop.*, Vol. 28, 1984.
- [17] M.Zhang and N.R.Shanbhag, "Soft-Error-Rate-Analysis (SERA) Methodology," *IEEE Trans. on CAD of ICs and Systems*, Vol. 25, no. 10, pp 2140-2155, Oct 2006.
- [18] S.Mitra, et.al. "Logic Soft Errors: A Major Barrier to Robust Platform Design," *IEEE Int.Test Conf*, 2005.
- [19] K.M.Fant and S.A.Brandt "NULL Convention Logic: A Complete and Consistent Logic for Asynchronous Digital Circuit Synthesis," *Int. conf. Application Specific Systems, Architectures, Processors*, 1996, pp 261-273.
- [20] C.L.Seitz, "System Timing," in *Introduction to VLSI Systems: Addison-Wesley*, 1980, pp. 218-262.
- [21] G.E.Sobelman and K.Fant, "CMOS Circuit Design of Threshold Gates with Hysteresis," *Proceedings of the International Symposium on Circuits and Systems*, pp 61-64, 1998.
- [22] W.Kuang et.al., "Performance Analysis and Optimization of NCL Self-timed Rings," *IEE Proceedings Circuits, Device and Systems*, Vol. 150, no. 3, June 2003.
- [23] S.C.Smith et.al, "Delay-Insensitive Gate –Level Pipelining," *Elsevier's Integration, the VLSI Journal*, Vol. 30/2, pp103-131, Oct 2001.
- [24] J.S.Yuan and Weidong Kuang, "Teaching Asynchronous Design in Digital Integrated Circuit," *IEEE Trans on Education*, Vol. 47, No 3, Aug 2003.
- [25] Michiel Ligthart, et.al., "Asynchronous Design using Commercial HDL Synthesis Tools," *IEEE Intl. Symposium on Advanced Research in Asynchronous Circuits and Systems*, April 2000, pp 114-125.

[26] FPGA Basics

<http://www.cse.cuhk.edu.hk/~ceg5010/tuto/basic-fpga-arch-xilinx.ppt>

[27] Xilinx – FPGA Design Flow Overview

http://www.xilinx.com/itp/xilinx8/help/iseguide/html/ise_fpga_design_flow_overview.htm

[28] Altera Quartus II Design Entry & Synthesis

<http://www.altera.com/products/software/quartus-ii/subscription-edition/design-entry-synthesis/qts-des-ent-syn.html>

[29] S.C.Smith, “Design of an FPGA Logic Element for Implementing Asynchronous Null Convention Logic Circuits,” *IEEE Transaction on VLSI system*, vol.15, no.6, June 2007.

[30] S.Smith et.al, “Design and Implementation of FPGA Configuration Logic Block Using Asynchronous Semi-Static NCL Circuits,” *IEEE Region 5 Conference*, April 2008.

[31] S.Smith, “Design and Implementation of FPGA Configuration Logic Block Using Asynchronous Static NCL,” *IEEE Region 5 Conference*, April 2008.

[32] K.Meekins, et.al, “Delay Insensitive NCL Reconfigurable Logic,” *IEEE Aerospace Conference proceedings*, Vol. 4, 2002.

[33] D.Fang et.al, “A High Performance Asynchronous FPGA: test results,” *IEEE Symposium on Field-programmable Custom Computing Machines*, pp 271-272, April 2005.

[34] J.P.Kaps and C.Paar, “Fast DES Implementation for FPGAs and its Application to a Universal Key-Search Machine”, *proc. 5th Annual workshop on Selected Areas in Cryptography*, Aug 1998, pp 234-247.

[35] F.X.Standaert, G.Rouvroy, J.J.Quisquater, “FPGA Implementations of the DES and Triple-DES Masked Against Power Analysis Attacks”, *IEEE Intl. Conference on Field Programmable Logic and Applications*, Aug 2006, pp 1-4.

[36] High Speed DES and Triple DES Encryptor/Decryptor.

http://www.xilinx.com/support/documentation/application_notes/xapp270.pdf

[37] Y.S.Dhillon, et.al, “ Analysis and Optimization of Nanometer CMOS Circuits for Soft-Error Tolerance,” *IEEE Trans. VLSI System*, Vol. 14, no.5, pp 514-524, May 2006.]

[38] T.Rejimon and S.Bhanja, “A Timing-Aware Probabilistic Model for Single-Event Upset Analysis,” *IEEE Trans. VLSI Systems*, Vol. 14, Oct 2006.

- [39] R.C.Baumann, "Radiation-Induced Soft Errors in Advanced Semiconductor Technologies," *IEEE Trans. Device and Material Reliability*, Vol. 5, no. 3, pp 305-316, Sept 2005.
- [40] P.Hazucha, et.al, "Impact of CMOS Technology Scaling on the Atmospheric Neutron Soft Error Rate," *IEEE Trans. on Nuclear Science*, Vol.47, no. 6, Dec 2000.
- [41] W.M.Ebrahim, "Asynchronous Circuit for Soft Error Tolerance," *Master's Thesis – The University of Texas-Pan American*, Dec 2008.
- [42] Y.Monnet, et.al., "Asynchronous Circuits Transient Faults Sensitivity Evaluation," *DAC 2005*, pp 863-868, June 2005.
- [43] Y.Monnet et.al, "Asynchronous Circuits Sensitivity to Fault Injection ," *Proc. 10th IEEE Intl. On-Line Testing Symp.* pp121-126, 2004.
- [44] W.Jang et.al., "SEU-Tolerant QDI Circuits," *Proceedings of the 11th Intl. Symp. on Asynchronous Circuits and Systems*, (ASYNC '05).
- [45] S.Peng and Rajit Manohar, "Efficient Failure Detection in Pipelined Asynchronous Circuits," *Proc. of the 20th IEEE Intl. Symposium on Defect and Fault Tolerance in VLSI Systems*, 2005.
- [46] I.M. Casto, "Self-Correction Structures and Robust Gate Design for Soft Error in Asynchronous Systems," *Master's Thesis – The University of Texas-Pan American*, Aug 2007.
- [47] Weidong Kuang, et.al. "Soft Error Hardening for Asynchronous Circuits," *22nd IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, pp273-281, Sept 2007.
- [48] Weidong Kuang, et.al. "Design Asynchronous Circuits for Soft Error Tolerance," *IEEE International Conference on Integrated Circuit Design and Technology*, May 2007.
- [49] Weidong Kuang, et.al. "Design of Asynchronous Circuits for High Soft Error Tolerance in Deep Sub micrometer CMOS Circuits," *IEEE Trans on VLSI Systems*, 2009.
- [50] Altera DE2 User Manual
ftp://ftp.altera.com/up/pub/Webdocs/DE2_UserManual.pdf
- [51] T.Beyrouthy et.al, "A Novel Asynchronous e-FPGA Architecture for Security Applications," *IEEE Intl. Conf. on Field Programmable Technology*, pp 369-372, Dec 2007.
- [52] V.Pouget, et.al, "Dynamic Testing of an SRAM based FPGA by Time Resolved Laser Fault Injection", *IEEE Symposium on On-Line Testing*, July 2008, pp 295-301.

APPENDIX A

APPENDIX A

DES Algorithm and Design Units in VHDL

Reference: <http://www.orlingrabbe.com/des.htm>

DES is a **block cipher**--meaning it operates on plaintext blocks of a given size (64-bits) and returns ciphertext blocks of the same size. Thus DES results in a **permutation** among the 2^{64} (read this as: "2 to the 64th power") possible arrangements of 64 bits, each of which may be either 0 or 1. Each block of 64 bits is divided into two blocks of 32 bits each, a left half block **L** and a right half block **R**. (This division is only used in certain operations.)

Example: Let **M** be the plain text message **M** = 0123456789ABCDEF, where **M** is in hexadecimal (base 16) format. Rewriting **M** in binary format, we get the 64-bit block of text:

M = 0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111

L = 0000 0001 0010 0011 0100 0101 0110 0111

R = 1000 1001 1010 1011 1100 1101 1110 1111

The first bit of **M** is "0". The last bit is "1". We read from left to right.

DES operates on the 64-bit blocks using key sizes of 56-bits. The keys are actually stored as being 64 bits long, but every 8th bit in the key is not used (i.e. bits numbered 8, 16, 24, 32, 40, 48, 56, and 64). However, we will nevertheless number the bits from 1 to 64, going left to right, in the following calculations. But, as you will see, the eight bits just mentioned get eliminated when we create subkeys.

Example: Let **K** be the hexadecimal key **K** = 133457799BBCDFF1. This gives us as the binary key (setting 1 = 0001, 3 = 0011, etc., and grouping together every eight bits, of which the last one in each group will be unused):

K = 00010011 00110100 01010111 01111001 10011011 10111100 11011111 11110001

The DES algorithm uses the following steps:

Step 1: Create 16 subkeys, each of which is 48-bits long.

The 64-bit key is permuted according to the following table, **PC-1**. Since the first entry in the table is "57", this means that the 57th bit of the original key **K** becomes the first bit of the permuted key **K+**. The 49th bit of the original key becomes the second bit of the permuted key. The 4th bit of the original key is the last bit of the permuted key. Note only 56 bits of the original key appear in the permuted key.

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

Example: From the original 64-bit key

K = 00010011 00110100 01010111 01111001 10011011 10111100 11011111 11110001

we get the 56-bit permutation

K+ = 1111000 0110011 0010101 0101111 0101010 1011001 1001111 0001111

Next, split this key into left and right halves, **C₀** and **D₀**, where each half has 28 bits.

Example: From the permuted key **K+**, we get

C₀ = 1111000 0110011 0010101 0101111

D₀ = 0101010 1011001 1001111 0001111

With C_0 and D_0 defined, we now create sixteen blocks C_n and D_n , $1 \leq n \leq 16$. Each pair of blocks C_n and D_n is formed from the previous pair C_{n-1} and D_{n-1} , respectively, for $n = 1, 2, \dots, 16$, using the following schedule of "left shifts" of the previous block. To do a left shift, move each bit one place to the left, except for the first bit, which is cycled to the end of the block.

Iteration Number	Number of Left Shifts
1	1
2	1
3	2
4	2
5	2
6	2
7	2
8	2
9	1
10	2
11	2
12	2
13	2
14	2
15	2
16	1

This means, for example, C_3 and D_3 are obtained from C_2 and D_2 , respectively, by two left shifts, and C_{16} and D_{16} are obtained from C_{15} and D_{15} , respectively, by one left shift. In all cases, by a single left shift is meant a rotation of the bits one place to the left, so that after one left shift the bits in the 28 positions are the bits that were previously in positions 2, 3, ..., 28, 1.

Example: From original pair C_0 and D_0 we obtain:

$C_0 = 1111000011001100101010101111$
 $D_0 = 0101010101100110011110001111$
 $C_1 = 1110000110011001010101011111$
 $D_1 = 1010101011001100111100011110$
 $C_2 = 1100001100110010101010111111$
 $D_2 = 0101010110011001111000111101$
 $C_3 = 0000110011001010101011111111$
 $D_3 = 0101011001100111100011110101$
 $C_4 = 0011001100101010101111111100$
 $D_4 = 0101100110011110001111010101$
 $C_5 = 1100110010101010111111110000$
 $D_5 = 0110011001111000111101010101$
 $C_6 = 0011001010101011111111000011$
 $D_6 = 1001100111100011110101010101$
 $C_7 = 1100101010101111111100001100$
 $D_7 = 0110011110001111010101010110$
 $C_8 = 0010101010111111110000110011$
 $D_8 = 1001111000111101010101011001$
 $C_9 = 0101010101111111100001100110$
 $D_9 = 0011110001111010101010110011$
 $C_{10} = 0101010111111110000110011001$
 $D_{10} = 1111000111101010101011001100$
 $C_{11} = 0101011111111000011001100101$
 $D_{11} = 1100011110101010101100110011$
 $C_{12} = 0101111111100001100110010101$
 $D_{12} = 0001111010101010110011001111$

$C_{13} = 0111111110000110011001010101$
 $D_{13} = 0111101010101011001100111100$
 $C_{14} = 1111111000011001100101010101$
 $D_{14} = 1110101010101100110011110001$
 $C_{15} = 1111100001100110010101010111$
 $D_{15} = 1010101010110011001111000111$
 $C_{16} = 1111000011001100101010101111$
 $D_{16} = 0101010101100110011110001111$

We now form the keys K_n , for $1 \leq n \leq 16$, by applying the following permutation table to each of the concatenated pairs $C_n D_n$. Each pair has 56 bits, but PC-2 only uses 48 of these.

PC-2					
14	17	11	24	1	5
3	28	15	6	21	10
23	19	12	4	26	8
16	7	27	20	13	2
41	52	31	37	47	55
30	40	51	45	33	48
44	49	39	56	34	53
46	42	50	36	29	32

Therefore, the first bit of K_n is the 14th bit of $C_n D_n$, the second bit the 17th, and so on, ending with the 48th bit of K_n being the 32th bit of $C_n D_n$.

Example: For the first key we have $C_1 D_1 = 1110000 1100110 0101010 1011111 1010101 0110011 0011110 0011110$

which, after we apply the permutation PC-2, becomes

$K_1 = 000110 110000 001011 101111 111111 000111 000001 110010$

For the other keys we have

$K_2 = 011110 011010 111011 011001 110110 111100 100111 100101$
 $K_3 = 010101 011111 110010 001010 010000 101100 111110 011001$
 $K_4 = 011100 101010 110111 010110 110110 110011 010100 011101$
 $K_5 = 011111 001110 110000 000111 111010 110101 001110 101000$
 $K_6 = 011000 111010 010100 111110 010100 000111 101100 101111$
 $K_7 = 111011 001000 010010 110111 111101 100001 100010 111100$
 $K_8 = 111101 111000 101000 111010 110000 010011 101111 111011$
 $K_9 = 111000 001101 101111 101011 111011 011110 011110 000001$
 $K_{10} = 101100 011111 001101 000111 101110 100100 011001 001111$
 $K_{11} = 001000 010101 111111 010011 110111 101101 001110 000110$
 $K_{12} = 011101 010111 000111 110101 100101 000110 011111 101001$
 $K_{13} = 100101 111100 010111 010001 111110 101011 101001 000001$
 $K_{14} = 010111 110100 001110 110111 111100 101110 011100 111010$
 $K_{15} = 101111 111001 000110 001101 001111 010011 111100 001010$
 $K_{16} = 110010 110011 110110 001011 000011 100001 011111 110101$

So much for the subkeys. Now we look at the message itself.

Step 2: Encode each 64-bit block of data.

There is an *initial permutation* IP of the 64 bits of the message data **M**. This rearranges the bits according to the following table, where the entries in the table show the new arrangement of the bits from their initial order. The 58th bit of **M** becomes the first bit of **IP**. The 50th bit of **M** becomes the second bit of **IP**. The 7th bit of **M** is the last bit of **IP**.

IP							
58	50	42	34	26	18	10	2
60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6
64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1
59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5
63	55	47	39	31	23	15	7

Example: Applying the initial permutation to the block of text **M**, given previously, we get

M = 0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111

IP = 1100 1100 0000 0000 1100 1100 1111 1111 1111 0000 1010 1010 1111 0000 1010 1010

Here the 58th bit of **M** is "1", which becomes the first bit of **IP**. The 50th bit of **M** is "1", which becomes the second bit of **IP**. The 7th bit of **M** is "0", which becomes the last bit of **IP**.

Next divide the permuted block **IP** into a left half **L₀** of 32 bits, and a right half **R₀** of 32 bits.

Example: From **IP**, we get **L₀** and **R₀**

L₀ = 1100 1100 0000 0000 1100 1100 1111 1111

R₀ = 1111 0000 1010 1010 1111 0000 1010 1010

We now proceed through 16 iterations, for $1 \leq n \leq 16$, using a function **f** which operates on two blocks--a data block of 32 bits and a key **K_n** of 48 bits--to produce a block of 32 bits. Let **+** denote XOR addition, (bit-by-bit addition modulo 2). Then for **n** going from 1 to 16 we calculate

$$L_n = R_{n-1}$$

$$R_n = L_{n-1} + f(R_{n-1}, K_n)$$

This results in a final block, for $n = 16$, of **L₁₆R₁₆**. That is, in each iteration, we take the right 32 bits of the previous result and make them the left 32 bits of the current step. For the right 32 bits in the current step, we XOR the left 32 bits of the previous step with the calculation **f**.

Example: For $n = 1$, we have

K₁ = 000110 110000 001011 101111 111111 000111 000001 110010

L₁ = **R₀** = 1111 0000 1010 1010 1111 0000 1010 1010

$$R_1 = L_0 + f(R_0, K_1)$$

It remains to explain how the function **f** works. To calculate **f**, we first expand each block **R_{n-1}** from 32 bits to 48 bits. This is done by using a selection table that repeats some of the bits in **R_{n-1}**. We'll call the use of this selection table the function **E**. Thus **E(R_{n-1})** has a 32 bit input block, and a 48 bit output block.

Let **E** be such that the 48 bits of its output, written as 8 blocks of 6 bits each, are obtained by selecting the bits in its inputs in order according to the following table:

E BIT-SELECTION TABLE

32	1	2	3	4	5
4	5	6	7	8	9
8	9	10	11	12	13
12	13	14	15	16	17
16	17	18	19	20	21
20	21	22	23	24	25
24	25	26	27	28	29
28	29	30	31	32	1

Thus the first three bits of **E(R_{n-1})** are the bits in positions 32, 1 and 2 of **R_{n-1}** while the last 2 bits of **E(R_{n-1})** are the bits in positions 32 and 1.

Example: We calculate **E(R₀)** from **R₀** as follows:

R₀ = 1111 0000 1010 1010 1111 0000 1010 1010

E(R₀) = 011110 100001 010101 010101 011110 100001 010101 010101

(Note that each block of 4 original bits has been expanded to a block of 6 output bits.)

Next in the **f** calculation, we XOR the output **E(R_{n-1})** with the key **K_n**:

$$K_n + E(R_{n-1}).$$

Example: For **K₁**, **E(R₀)**, we have

K₁ = 000110 110000 001011 101111 111111 000111 000001 110010

E(R₀) = 011110 100001 010101 010101 011110 100001 010101 010101

K₁ + E(R₀) = 011000 010001 011110 111010 100001 100110 010100 100111.

We have not yet finished calculating the function **f**. To this point we have expanded **R_{n-1}** from 32 bits to 48 bits, using the selection table, and XORed the result with the key **K_n**. We now have 48 bits, or eight groups of six bits. We now do something strange with each group of six bits: we use them as addresses in tables called "S boxes". Each group of six bits will give us an address in a different S box. Located at that address will be a 4 bit number. This 4 bit number will replace the original 6 bits. The net result is that the eight groups of 6 bits are transformed into eight groups of 4 bits (the 4-bit outputs from the S boxes) for 32 bits total.

Write the previous result, which is 48 bits, in the form:

$$K_n + E(R_{n-1}) = B_1 B_2 B_3 B_4 B_5 B_6 B_7 B_8,$$

where each B_i is a group of six bits. We now calculate

$$S_1(B_1)S_2(B_2)S_3(B_3)S_4(B_4)S_5(B_5)S_6(B_6)S_7(B_7)S_8(B_8)$$

where $S_i(B_i)$ refers to the output of the i -th S box.

To repeat, each of the functions S_1, S_2, \dots, S_8 , takes a 6-bit block as input and yields a 4-bit block as output.

The table to determine S_1 is shown and explained below:

S1																
Column Number																
Row No.	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
1	0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
2	4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
3	15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

If S_1 is the function defined in this table and B is a block of 6 bits, then $S_1(B)$ is determined as follows: The first and last bits of B represent in base 2 a number in the decimal range 0 to 3 (or binary 00 to 11). Let that number be i . The middle 4 bits of B represent in base 2 a number in the decimal range 0 to 15 (binary 0000 to 1111). Let that number be j . Look up in the table the number in the i -th row and j -th column. It is a number in the range 0 to 15 and is uniquely represented by a 4 bit block. That block is the output $S_1(B)$ of S_1 for the input B . For example, for input block $B = 011011$ the first bit is "0" and the last bit "1" giving 01 as the row. This is row 1. The middle four bits are "1101". This is the binary equivalent of decimal 13, so the column is column number 13. In row 1, column 13 appears 5. This determines the output; 5 is binary 0101, so that the output is 0101. Hence $S_1(011011) = 0101$.

The tables defining the functions S_1, \dots, S_8 are the following:

S1															
14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

S2															
15	1	8	14	6	11	3	4	9	7	2	13	12	0	5	10
3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	5
0	14	7	11	10	4	13	1	5	8	12	6	9	3	2	15
13	8	10	1	3	15	4	2	11	6	7	12	0	5	14	9

S3															
10	0	9	14	6	3	15	5	1	13	12	7	11	4	2	8
13	7	0	9	3	4	6	10	2	8	5	14	12	11	15	1
13	6	4	9	8	15	3	0	11	1	2	12	5	10	14	7
1	10	13	0	6	9	8	7	4	15	14	3	11	5	2	12

S4															
7	13	14	3	0	6	9	10	1	2	8	5	11	12	4	15
13	8	11	5	6	15	0	3	4	7	2	12	1	10	14	9
10	6	9	0	12	11	7	13	15	1	3	14	5	2	8	4
3	15	0	6	10	1	13	8	9	4	5	11	12	7	2	14

S5															
2	12	4	1	7	10	11	6	8	5	3	15	13	0	14	9
14	11	2	12	4	7	13	1	5	0	15	10	3	9	8	6
4	2	1	11	10	13	7	8	15	9	12	5	6	3	0	14
11	8	12	7	1	14	2	13	6	15	0	9	10	4	5	3

S6															
12	1	10	15	9	2	6	8	0	13	3	4	14	7	5	11
10	15	4	2	7	12	9	5	6	1	13	14	0	11	3	8
9	14	15	5	2	8	12	3	7	0	4	10	1	13	11	6
4	3	2	12	9	5	15	10	11	14	1	7	6	0	8	13

S7															
4	11	2	14	15	0	8	13	3	12	9	7	5	10	6	1
13	0	11	7	4	9	1	10	14	3	5	12	2	15	8	6
1	4	11	13	12	3	7	14	10	15	6	8	0	5	9	2
6	11	13	8	1	4	10	7	9	5	0	15	14	2	3	12

S8															
13	2	8	4	6	15	11	1	10	9	3	14	5	0	12	7
1	15	13	8	10	3	7	4	12	5	6	11	0	14	9	2
7	11	4	1	9	12	14	2	0	6	10	13	15	3	5	8
2	1	14	7	4	10	8	13	15	12	9	0	3	5	6	11

Example: For the first round, we obtain as the output of the eight S boxes:

$$K_1 + E(R_0) = 011000\ 010001\ 011110\ 111010\ 100001\ 100110\ 010100\ 100111.$$

$$S_1(B_1)S_2(B_2)S_3(B_3)S_4(B_4)S_5(B_5)S_6(B_6)S_7(B_7)S_8(B_8) = 0101\ 1100\ 1000\ 0010\ 1011\ 0101\ 1001\ 0111$$

The final stage in the calculation of f is to do a permutation P of the S-box output to obtain the final value of f :

$$f = P(S_1(B_1)S_2(B_2)...S_8(B_8))$$

The permutation P is defined in the following table. P yields a 32-bit output from a 32-bit input by permuting the bits of the input block.

P			
16	7	20	21
29	12	28	17
1	15	23	26
5	18	31	10
2	8	24	14
32	27	3	9
19	13	30	6
22	11	4	25

Example: From the output of the eight S boxes:

$$S_1(B_1)S_2(B_2)S_3(B_3)S_4(B_4)S_5(B_5)S_6(B_6)S_7(B_7)S_8(B_8) = 0101\ 1100\ 1000\ 0010\ 1011\ 0101\ 1001\ 0111$$

we get

$$f = 0010\ 0011\ 0100\ 1010\ 1010\ 1001\ 1011\ 1011$$

$$R_1 = L_0 + f(R_0, K_1)$$

$$= 1100\ 1100\ 0000\ 0000\ 1100\ 1100\ 1111\ 1111$$

$$+ 0010\ 0011\ 0100\ 1010\ 1010\ 1001\ 1011\ 1011$$

$$= 1110\ 1111\ 0100\ 1010\ 0110\ 0101\ 0100\ 0100$$

In the next round, we will have $L_2 = R_1$, which is the block we just calculated, and then we must calculate

$R_2 = L_1 + f(R_1, K_2)$, and so on for 16 rounds. At the end of the sixteenth round we have the blocks L_{16} and

R_{16} . We then *reverse* the order of the two blocks into the 64-bit block

$$R_{16}L_{16}$$

and apply a final permutation IP^{-1} as defined by the following table:

$$IP^{-1}$$

40	8	48	16	56	24	64	32
39	7	47	15	55	23	63	31
38	6	46	14	54	22	62	30
37	5	45	13	53	21	61	29
36	4	44	12	52	20	60	28
35	3	43	11	51	19	59	27

34	2	42	10	50	18	58	26
33	1	41	9	49	17	57	25

That is, the output of the algorithm has bit 40 of the preoutput block as its first bit, bit 8 as its second bit, and so on, until bit 25 of the preoutput block is the last bit of the output.

Example: If we process all 16 blocks using the method defined previously, we get, on the 16th round,

$L_{16} = 0100\ 0011\ 0100\ 0010\ 0011\ 0010\ 0011\ 0100$

$R_{16} = 0000\ 1010\ 0100\ 1100\ 1101\ 1001\ 1001\ 0101$

We reverse the order of these two blocks and apply the final permutation to

$R_{16}L_{16} = 00001010\ 01001100\ 11011001\ 10010101\ 01000011\ 01000010\ 00110010\ 00110100$

$IP^I = 10000101\ 11101000\ 00010011\ 01010100\ 00001111\ 00001010\ 10110100\ 00000101$

which in hexadecimal format is

85E813540F0AB405.

This is the encrypted form of $M = 0123456789ABCDEF$: namely, $C = 85E813540F0AB405$.

Decryption is simply the inverse of encryption, following the same steps as above, but reversing the order in which the subkeys are applied.

APPENDIX B

APPENDIX B

NCL Library

NCL_signals.vhd

```

Library IEEE;
use IEEE.std_logic_1164.all;

package ncl_signals is

type dual_rail_logic is
    record
        RAIL1 : std_logic;
        RAIL0 : std_logic;
    end record;

type dual_rail_logic_vector is array (NATURAL range <>) of dual_rail_logic;

end ncl_signals;

```

NCL_gates.vhd

```

-----
-- invx0
-----
library ieee;
use ieee.std_logic_1164.all;

entity invx0 is
    port(i: in std_logic;
         zb: out std_logic);
end invx0;

architecture archinvx0 of invx0 is
begin
    invx0: process(i)
    begin
        if i = '0' then
            zb <= '1';
        elsif i = '1' then
            zb <= '0';
        else
            zb <= not i;
        end if;
    end process;
end archinvx0;

-----
-- th12bx0
-----
library ieee;
use ieee.std_logic_1164.all;

entity th12bx0 is
    port(a: in std_logic;
         b: in std_logic;
         zb: out std_logic);
end th12bx0;

```

```

architecture archth12bx0 of th12bx0 is
begin
  th12bx0: process(a, b)
  begin
    if a = '0' and b = '0' then
      zb <= '1';
    elsif a = '1' or b = '1' then
      zb <= '0';
    -- else
      --zb <= a nor b;
    end if;
  end process;
end archth12bx0;

```

```

-----
-- th22dx0
-----

```

```

library ieee;
use ieee.std_logic_1164.all;

```

```

entity th22dx0 is
  port(a: in std_logic;
        b: in std_logic;
        rst: in std_logic;
        z: out std_logic );
end th22dx0;

```

```

architecture archth22dx0 of th22dx0 is
begin
  th22dx0: process(a, b, rst)
  begin
    if rst = '1' then -- reset
      z <= '1';
    elsif (a= '1' and b= '1') then
      z <= '1';
    elsif (a= '0' and b= '0') then
      z <= '0';
    end if;
  end process;
end archth22dx0;

```

```

-----
-- th22nx0
-----

```

```

library ieee;
use ieee.std_logic_1164.all;

```

```

entity th22nx0 is
  port(a: in std_logic;
        b: in std_logic;
        rst: in std_logic;
        z: out std_logic );
end th22nx0;

```

```

architecture archth22nx0 of th22nx0 is
begin
  th22nx0: process(a, b, rst)
  begin
    if rst = '1' then -- reset
      z <= '0';
    elsif (a= '1' and b= '1') then
      z <= '1';
    elsif (a= '0' and b= '0') then
      z <= '0';
    end if;
  end process;
end archth22nx0;

```

```

-----
-- th22x0
-----
library ieee;
use ieee.std_logic_1164.all;

entity th22x0 is
    port(a: in std_logic;
          b: in std_logic;
          z: out std_logic );
end th22x0;

architecture archth22x0 of th22x0 is
begin
    th22x0: process(a, b)
    begin
        if (a= '1' and b= '1') then
            z <= '1';
        elsif (a= '0' and b= '0') then
            z <= '0';
        end if;
    end process;
end archth22x0;

-----
-- th23x0
-----
library ieee;
use ieee.std_logic_1164.all;

entity th23x0 is
    port(a: in std_logic;
          b: in std_logic;
          c: in std_logic;
          z: out std_logic );
end th23x0;

architecture archth23x0 of th23x0 is
begin
    th23x0: process(a, b, c)
    begin
        if (a= '0' and b= '0' and c= '0') then
            z <= '0';
        elsif (a= '1' and b= '1') or (b= '1' and c= '1') or (c= '1' and a= '1') then
            z <= '1';
        end if;
    end process;
end archth23x0;

-----
-- th23w2x0
-----
library ieee;
use ieee.std_logic_1164.all;

entity th23w2x0 is
    port(a: in std_logic; -- weight 2
          b: in std_logic;
          c: in std_logic;
          z: out std_logic );
end th23w2x0;

architecture archth23w2x0 of th23w2x0 is
begin
    th23w2x0: process(a, b, c)
    begin
        if (a= '0' and b= '0' and c= '0') then
            z <= '0';
        elsif (a= '1' or (b= '1' and c= '1')) then

```

```

        z <= '1';
    end if; -- else NULL
end process;
end archth23w2x0;

-----
-- th33x0
-----
library ieee;
use ieee.std_logic_1164.all;

entity th33x0 is
    port(a: in std_logic;
          b: in std_logic;
          c: in std_logic;
          z: out std_logic );
end th33x0;

architecture archth33x0 of th33x0 is
begin
    th33x0: process(a, b, c)
    begin
        if (a= '1' and b= '1' and c= '1') then
            z <= '1';
        elsif (a= '0' and b= '0' and c= '0') then
            z <= '0';
        end if; -- else NULL
    end process;
end archth33x0;

-----
--th34w2x0
-----
library ieee;
use ieee.std_logic_1164.all;

entity th34w2x0 is
    port(a: in std_logic; -- weight 2
          b: in std_logic;
          c: in std_logic;
          d: in std_logic;
          z: out std_logic );
end th34w2x0;

architecture archth34w2x0 of th34w2x0 is
begin
    th34w2x0: process(a, b, c, d)
    begin
        if (a= '0' and b= '0' and c= '0' and d = '0') then
            z <= '0';
        elsif (a = '1' and b = '1')
            or (a = '1' and c = '1')
            or (a = '1' and d = '1')
            or (b = '1' and c = '1' and d = '1') then
            z <= '1';
        end if; -- else NULL
    end process;
end archth34w2x0;

-----
-- th44x0
-----
library ieee;
use ieee.std_logic_1164.all;

entity th44x0 is
    port(a: in std_logic;
          b: in std_logic;
          c: in std_logic;
          d: in std_logic;

```

```

        z: out std_logic );
end th44x0;

architecture archth44x0 of th44x0 is
begin
    th44x0: process(a, b, c, d)
    begin
        if (a= '1' and b= '1' and c= '1' and d= '1') then
            z <= '1';
        elsif (a= '0' and b= '0' and c= '0' and d= '0') then
            z <= '0';
        end if; -- else NULL
    end process;
end archth44x0;

```

NCL_components.vhd

-- Package used for Completion Component

```

Library IEEE;
use IEEE.std_logic_1164.all;

package tree_funcs is

function log_u(L: integer; R: integer) return integer; -- ceiling of Log base R of L
function level_number(width, level, base: integer) return integer; -- bits to be combined
on level of tree of width using base input gates

end tree_funcs;

package body tree_funcs is

function log_u(L: integer; R: integer) return integer is
variable temp: integer := 1;
variable level: integer := 0;
begin
    if L = 1 then
        return 0;
    end if;

    while temp < L loop
        temp := temp * R;
        level := level + 1;
    end loop;
    return level;
end;

function level_number(width, level, base: integer) return integer is
variable num: integer := width;
begin
    if level /= 0 then
        for i in 1 to level loop
            if (log_u((num / base) + (num rem base), base) + i) = log_u(width, base)
then
                num := (num / base) + (num rem base);
            else
                num := (num / base) + 1;
            end if;
        end loop;
    end if;
    return num;
end;

end tree_funcs;

-- Generic Completion Component

library ieee;
use ieee.std_logic_1164.all;
use work.tree_funcs.all;

```

```

entity comp is
    generic(width : integer);
    port(a: IN std_logic_vector(width-1 downto 0);
          ko: OUT std_logic);
end comp;

architecture arch of comp is

    type completion is array(log_u(width, 4) downto 0, width-1 downto 0) of std_logic;
    signal comp_array: completion;

    component th22x0
        port(a: in std_logic;
              b: in std_logic;
              z: out std_logic);
    end component;

    component th33x0
        port(a: in std_logic;
              b: in std_logic;
              c: in std_logic;
              z: out std_logic);
    end component;

    component th44x0
        port(a: in std_logic;
              b: in std_logic;
              c: in std_logic;
              d: in std_logic;
              z: out std_logic);
    end component;

begin
    RENAME: for i in 0 to width-1 generate
        comp_array(0, i) <= a(i);
    end generate;

    STRUCTURE: for k in 0 to log_u(width, 4)-1 generate
        begin
            NOT_LAST: if level_number(width, k, 4) > 4 generate
                begin
                    PRINCIPLE: for j in 0 to (level_number(width, k, 4) / 4)-1 generate
                        G4: th44x0
                        port map(comp_array(k, j*4), comp_array(k, j*4+1), comp_array(k, j*4+2), comp_array(k,
                        j*4+3), comp_array(k+1, j));
                    end generate;

                    LEFT_OVER_GATE: if log_u((level_number(width, k, 4) / 4) + (level_number(width, k, 4) rem
                    4), 4) + k + 1 /= log_u(width, 4) generate
                        begin
                            NEED22: if (level_number(width, k, 4) rem 4) = 2 generate
                                G2: th22x0
                                port map(comp_array(k, level_number(width, k, 4)-2), comp_array(k, level_number(width, k,
                                4)-1), comp_array(k+1, (level_number(width, k, 4) / 4)));
                            end generate

                            NEED33: if (level_number(width, k, 4) rem 4) = 3 generate
                                G3: th33x0
                                port map(comp_array(k, level_number(width, k, 4)-3), comp_array(k, level_number(width, k,
                                4)-2), comp_array(k, level_number(width, k, 4)-1), comp_array(k+1, (level_number(width,
                                k, 4) / 4)));
                            end generate;
                        end generate;

                    LEFT_OVER_SIGNALS: if (log_u((level_number(width, k, 4) / 4) + (level_number(width, k, 4)
                    rem 4), 4) + k + 1 = log_u(width, 4)) and ((level_number(width, k, 4) rem 4) /= 0)
                    generate
                        begin
                            RENAME_SIGNALS: for h in 0 to (level_number(width, k, 4) rem 4)-1 generate

```

```

comp_array(k+1, (level_number(width, k, 4) / 4)+h) <= comp_array(k, level_number(width,
k, 4)-1-h);
end generate;
end generate;
end generate;

LAST22: if level_number(width, k, 4) = 2 generate
G2F: th22x0
port map(comp_array(k, 0), comp_array(k, 1), ko);
end generate;

LAST33: if level_number(width, k, 4) = 3 generate
G3F: th33x0
port map(comp_array(k, 0), comp_array(k, 1), comp_array(k, 2), ko);
end generate;

LAST44: if level_number(width, k, 4) = 4 generate
G4F: th44x0
port map(comp_array(k, 0), comp_array(k, 1), comp_array(k, 2), comp_array(k, 3), ko);
end generate;
end generate;

end arch;

-- 1-bit Dual-Rail Register

use work.ncl_signals.all;
library ieee;
use ieee.std_logic_1164.all;

entity ncl_register_D1 is
generic(initial_value: integer := -4); -- 1=DATA1, 0=DATA0, -4=NULL
port(D: in dual_rail_logic;
ki: in std_logic;
rst: in std_logic;
Q: out dual_rail_logic;
ko: out std_logic);
end ncl_register_D1;

architecture arch of ncl_register_D1 is
signal Qbuf: dual_rail_logic;

component th22nx0
port (a, b, rst: IN std_logic;
z: OUT std_logic);
end component;

component th22dx0
port (a, b, rst: IN std_logic;
z: OUT std_logic);
end component;

component th12bx0
port (a, b: IN std_logic;
zb: OUT std_logic);
end component;

begin
RstN: if initial_value = -4 generate
R0: th22nx0
port map(D.rail0, ki, rst, Qbuf.rail0);

R1: th22nx0
port map(D.rail1, ki, rst, Qbuf.rail1);
end generate;

Rst1: if initial_value = 1 generate
R0: th22nx0
port map(D.rail0, ki, rst, Qbuf.rail0);

R1: th22dx0

```



```

        port map(D.rail1, ki, rst, Qbuf.rail1);
    end generate;

    Rst0: if initial_value = 0 generate
        R0: th22dx0
            port map(D.rail0, ki, rst, Qbuf.rail0);

        R1: th22nx0
            port map(D.rail1, ki, rst, Qbuf.rail1);
    end generate;

    Q <= Qbuf;

    COMP: th12bx0
        port map(Qbuf.rail0, Qbuf.rail1, ko);
end;

-- Generic Length Dual-Rail Register

use work.ncl_signals.all;
library ieee;
use ieee.std_logic_1164.all;

entity ncl_register_D is
    generic(width: integer;
        initial_value: integer := -4); -- 1=DATA1, 0=DATA0, -4=NULL
    port(D: in dual_rail_logic_vector(width-1 downto 0);
        ki: in std_logic;
        rst: in std_logic;
        Q: out dual_rail_logic_vector(width-1 downto 0);
        ko: out std_logic_vector(width-1 downto 0));
end ncl_register_D;

architecture gen of ncl_register_D is
    component ncl_register_D1
        generic(initial_value: integer := -4); -- 1=DATA1, 0=DATA0, -4=NULL
        port(D: in dual_rail_logic;
            ki: in std_logic;
            rst: in std_logic;
            Q: out dual_rail_logic;
            ko: out std_logic);
    end component;

    begin
        gen_reg: for i in 0 to D'length-1 generate
            REGi: ncl_register_D1
                generic map(initial_value)
                port map(D(i), ki, rst, Q(i), ko(i));
        end generate;
    end;

-- 1-bit initreg without comp det

use work.ncl_signals.all;
library ieee;
use ieee.std_logic_1164.all;

entity ncl_register_D11 is
    generic(initial_value: integer := -4); -- 1=DATA1, 0=DATA0, -4=NULL
    port(D: in dual_rail_logic;
        ki: in std_logic;
        rst: in std_logic;
        Q: out dual_rail_logic);
end ncl_register_D11;

architecture arch of ncl_register_D11 is
    component th22nx0
        port (a, b, rst: IN std_logic;
            z: OUT std_logic);
    end component;

```

```

end component;

component th22dx0
  port (a, b, rst: IN std_logic;
        z: OUT std_logic);
end component;

begin
  RstN: if initial_value = -4 generate
    R0: th22nx0
      port map(D.rail0, ki, rst, Q.rail0);

    R1: th22nx0
      port map(D.rail1, ki, rst, Q.rail1);
  end generate;

  Rst1: if initial_value = 1 generate
    R0: th22nx0
      port map(D.rail0, ki, rst, Q.rail0);

    R1: th22dx0
      port map(D.rail1, ki, rst, Q.rail1);
  end generate;

  Rst0: if initial_value = 0 generate
    R0: th22dx0
      port map(D.rail0, ki, rst, Q.rail0);

    R1: th22nx0
      port map(D.rail1, ki, rst, Q.rail1);
  end generate;

end;

-- Generic Length initial register

use work.ncl_signals.all;
library ieee;
use ieee.std_logic_1164.all;

entity ncl_reg_Dinit is
  generic(width: integer;
          initial_value: integer ); -- 1=DATA1, 0=DATA0, -4=NULL
  port(D: in dual_rail_logic_vector(width-1 downto 0);
        ki: in std_logic;
        rst: in std_logic;
        Q: out dual_rail_logic_vector(width-1 downto 0));
end ncl_reg_Dinit;

architecture gen of ncl_reg_Dinit is
  component ncl_register_D11
    generic(initial_value: integer := -4); -- 1=DATA1, 0=DATA0, -4=NULL
    port(D: in dual_rail_logic;
          ki: in std_logic;
          rst: in std_logic;
          Q: out dual_rail_logic);
  end component;

begin
  gen_reg: for i in 0 to D'length-1 generate
    REGi: ncl_register_D11
      generic map(initial_value)
      port map(D(i), ki, rst, Q(i));
  end generate;

end;

```

APPENDIX C

APPENDIX C

VHDL Files Used for the Thesis

Asynchronous DES Algorithm:

des1.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.ncl_signals.all;

entity des1 is
    Port ( pt : in  dual_rail_logic_vector (1 to 64);
          key : in  dual_rail_logic_vector (1 to 64);
          ct : out dual_rail_logic_vector (1 to 64);
          rst : in  STD_LOGIC);
end des1;

architecture Behavioral of des1 is
    signal pti,keyi,keyo,pto,cti : dual_rail_logic_vector(1 to 64);
    signal L1,L2,L3,L4,L5,L6,L7,L8,L9,L10,L11,L12,L13,L14,L15,L16,
          L1i,L2i,L3i,L4i,L5i,L6i,L7i,L8i,L9i,L10i,L11i,L12i,L13i,
          L14i,L15i,L16i,R1,R2,R3,R4,R5,R6,R7,R8,R9,R10,R11,R12,
          R13,R14,R15,R16,R1i,R2i,R3i,R4i,R5i,R6i,R7i,R8i,R9i,R10i,
          R11i,R12i,R13i,R14i,R15i,R16i : dual_rail_logic_vector(1 to 32);
    signal C1,C2,C3,C4,C5,C6,C7,C8,C9,C10,C11,C12,C13,C14,C15,C16,
          C1i,C2i,C3i,C4i,C5i,C6i,C7i,C8i,C9i,C10i,C11i,C12i,C13i,
          C14i,C15i,C16i,D1,D2,D3,D4,D5,D6,D7,D8,D9,D10,D11,D12,D13,
          D14,D15,D16,D1i,D2i,D3i,D4i,D5i,D6i,D7i,D8i,D9i,D10i,D11i,
          D12i,D13i,D14i,D15i,D16i : dual_rail_logic_vector(1 to 28);
    signal k1,k2,k3,k4,k5,k6,k7,k8,k9,k10,k11,k12,k13,k14,k15,k16,k17 : std_logic;
    component initreg
        port ( DT1 : in dual_rail_logic_vector(1 to 64);
              DT2 : in dual_rail_logic_vector(1 to 64);
              rst : in std_logic;
              ki : in std_logic;
              Q1 : out dual_rail_logic_vector(1 to 64);
              Q2 : out dual_rail_logic_vector(1 to 64));
    end component;
    component regandcompdet1to15 is
        Port ( DT1 : in dual_rail_logic_vector(1 to 32);
              DT2 : in dual_rail_logic_vector(1 to 32);
              DT3 : in dual_rail_logic_vector(1 to 28);
              DT4 : in dual_rail_logic_vector(1 to 28);
              rst : in  STD_LOGIC;
              ki : in  STD_LOGIC;
              Q1 : out  dual_rail_logic_vector(1 to 32);
              Q2 : out  dual_rail_logic_vector(1 to 32);
              Q3 : out  dual_rail_logic_vector(1 to 28);
              Q4 : out  dual_rail_logic_vector(1 to 28);
              ko : out  std_logic);
    end component;
    component regcompdet
        port ( DT1 : in dual_rail_logic_vector(1 to 32);
              DT2 : in dual_rail_logic_vector(1 to 32);
              rst : in std_logic;
              ki : in std_logic;

```

```

        Q1 : out dual_rail_logic_vector(1 to 32);
        Q2 : out dual_rail_logic_vector(1 to 32);
        ko : out std_logic);
end component;
component finalregandcompdet
    port ( D : in dual_rail_logic_vector(1 to 64);
          rst : in std_logic;
          ki : in std_logic;
          Q : out dual_rail_logic_vector(1 to 64);
          ko : out std_logic);
end component;
component initround
    port ( pt : in dual_rail_logic_vector(1 to 64);
          key : in dual_rail_logic_vector(1 to 64);
          L : out dual_rail_logic_vector(1 to 32);
          R : out dual_rail_logic_vector(1 to 32);
          C : out dual_rail_logic_vector(1 to 28);
          D : out dual_rail_logic_vector(1 to 28));
end component;
component roundlto14s1
    port ( Li : in dual_rail_logic_vector(1 to 32);
          Ri : in dual_rail_logic_vector(1 to 32);
          Ci : in dual_rail_logic_vector(1 to 28);
          Di : in dual_rail_logic_vector(1 to 28);
          Lo : out dual_rail_logic_vector(1 to 32);
          Ro : out dual_rail_logic_vector(1 to 32);
          Co : out dual_rail_logic_vector(1 to 28);
          Do : out dual_rail_logic_vector(1 to 28));
end component;
component roundlto14s2
    port ( Li : in dual_rail_logic_vector(1 to 32);
          Ri : in dual_rail_logic_vector(1 to 32);
          Ci : in dual_rail_logic_vector(1 to 28);
          Di : in dual_rail_logic_vector(1 to 28);
          Lo : out dual_rail_logic_vector(1 to 32);
          Ro : out dual_rail_logic_vector(1 to 32);
          Co : out dual_rail_logic_vector(1 to 28);
          Do : out dual_rail_logic_vector(1 to 28));
end component;
component round
    port ( Li : in dual_rail_logic_vector(1 to 32);
          Ri : in dual_rail_logic_vector(1 to 32);
          Ci : in dual_rail_logic_vector(1 to 28);
          Di : in dual_rail_logic_vector(1 to 28);
          Lo : out dual_rail_logic_vector(1 to 32);
          Ro : out dual_rail_logic_vector(1 to 32));
end component;
component finalround
    port ( L16 : in dual_rail_logic_vector(1 to 32);
          R16 : in dual_rail_logic_vector(1 to 32);
          ct : out dual_rail_logic_vector(1 to 64));
end component;

begin

pti <= pt; keyi <= key;
reg0 : initreg port map(pti,keyi,rst,k1,pto,keyo);
round0 : initround port map(pto,keyo,L1i,R1i,C1i,D1i);
reg1 : regandcompdetlto15 port map(L1i,R1i,C1i,D1i,rst,k2,L1,R1,C1,D1,k1);
round1 : roundlto14s1 port map(L1,R1,C1,D1,L2i,R2i,C2i,D2i);
reg2 : regandcompdetlto15 port map(L2i,R2i,C2i,D2i,rst,k3,L2,R2,C2,D2,k2);
round2 : roundlto14s2 port map(L2,R2,C2,D2,L3i,R3i,C3i,D3i);
reg3 : regandcompdetlto15 port map(L3i,R3i,C3i,D3i,rst,k4,L3,R3,C3,D3,k3);
round3 : roundlto14s2 port map(L3,R3,C3,D3,L4i,R4i,C4i,D4i);
reg4 : regandcompdetlto15 port map(L4i,R4i,C4i,D4i,rst,k5,L4,R4,C4,D4,k4);
round4 : roundlto14s2 port map(L4,R4,C4,D4,L5i,R5i,C5i,D5i);
reg5 : regandcompdetlto15 port map(L5i,R5i,C5i,D5i,rst,k6,L5,R5,C5,D5,k5);
round5 : roundlto14s2 port map(L5,R5,C5,D5,L6i,R6i,C6i,D6i);
reg6 : regandcompdetlto15 port map(L6i,R6i,C6i,D6i,rst,k7,L6,R6,C6,D6,k6);
round6 : roundlto14s2 port map(L6,R6,C6,D6,L7i,R7i,C7i,D7i);
reg7 : regandcompdetlto15 port map(L7i,R7i,C7i,D7i,rst,k8,L7,R7,C7,D7,k7);

```

```

round7 : roundlto14s2 port map(L7,R7,C7,D7,L8i,R8i,C8i,D8i);
reg8 : regandcompdet1to15 port map(L8i,R8i,C8i,D8i,rst,k8,L8,R8,C8,D8,k8);
round8 : roundlto14s1 port map(L8,R8,C8,D8,L9i,R9i,C9i,D9i);
reg9 : regandcompdet1to15 port map(L9i,R9i,C9i,D9i,rst,k10,L9,R9,C9,D9,k9);
round9 : roundlto14s2 port map(L9,R9,C9,D9,L10i,R10i,C10i,D10i);
reg10 : regandcompdet1to15 port map(L10i,R10i,C10i,D10i,rst,k11,L10,R10,C10,D10,k10);
round10 : roundlto14s2 port map(L10,R10,C10,D10,L11i,R11i,C11i,D11i);
reg11 : regandcompdet1to15 port map(L11i,R11i,C11i,D11i,rst,k12,L11,R11,C11,D11,k11);
round11 : roundlto14s2 port map(L11,R11,C11,D11,L12i,R12i,C12i,D12i);
reg12 : regandcompdet1to15 port map(L12i,R12i,C12i,D12i,rst,k13,L12,R12,C12,D12,k12);
round12 : roundlto14s2 port map(L12,R12,C12,D12,L13i,R13i,C13i,D13i);
reg13 : regandcompdet1to15 port map(L13i,R13i,C13i,D13i,rst,k14,L13,R13,C13,D13,k13);
round13 : roundlto14s2 port map(L13,R13,C13,D13,L14i,R14i,C14i,D14i);
reg14 : regandcompdet1to15 port map(L14i,R14i,C14i,D14i,rst,k15,L14,R14,C14,D14,k14);
round14 : roundlto14s2 port map(L14,R14,C14,D14,L15i,R15i,C15i,D15i);
reg15 : regandcompdet1to15 port map(L15i,R15i,C15i,D15i,rst,k16,L15,R15,C15,D15,k15);
round15 : round port map(L15,R15,C15,D15,L16i,R16i);
reg16 : regcompdet port map(L16i,R16i,rst,k17,L16,R16,k16);
round16 : finalround port map(R16,L16,cti);
reg17 : finalregandcompdet port map(cti,rst,k17,ct,k17);

```

```
end Behavioral;
```

initreg.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.ncl_signals.all;

entity initreg is
    Port ( DT1 : in  dual_rail_logic_vector(1 to 64);
          DT2 : in  dual_rail_logic_vector(1 to 64);
          rst : in  STD_LOGIC;
          ki : in  STD_LOGIC;
          Q1 : out dual_rail_logic_vector(1 to 64);
          Q2 : out dual_rail_logic_vector(1 to 64));
end initreg;

architecture Behavioral of initreg is

    component ncl_reg_Dinit
        generic(width: integer;
               initial_value: integer ); -- 1=DATA1, 0=DATA0, -4=NULL
        port( D: in dual_rail_logic_vector(width-1 downto 0);
              ki: in std_logic;
              rst: in std_logic;
              Q: out dual_rail_logic_vector(width-1 downto 0));
    end component;

    begin
        reg1 : ncl_reg_Dinit
            generic map(width => 64,initial_value => -4)
            port map(DT1,ki,rst,Q1);

        reg2 : ncl_reg_Dinit
            generic map(width => 64,initial_value => -4)
            port map(DT2,ki,rst,Q2);

    end Behavioral;

```

regandcompdet1to15.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.ncl_signals.all;

entity regandcompdet1to15 is
    Port ( DT1 : in dual_rail_logic_vector(1 to 32);

```

```

        DT2 : in dual_rail_logic_vector(1 to 32);
        DT3 : in dual_rail_logic_vector(1 to 28);
        DT4 : in dual_rail_logic_vector(1 to 28);
        rst : in STD_LOGIC;
        ki : in STD_LOGIC;
        Q1 : out dual_rail_logic_vector(1 to 32);
        Q2 : out dual_rail_logic_vector(1 to 32);
        Q3 : out dual_rail_logic_vector(1 to 28);
        Q4 : out dual_rail_logic_vector(1 to 28);
        ko : out std_logic);
end regandcompdet1to15;

architecture Behavioral of regandcompdet1to15 is
    signal ko1,ko2 : std_logic_vector(31 downto 0);
    signal ko3,ko4 : std_logic_vector(27 downto 0);
    signal ao1,ao2,ao3,ao4 : std_logic;
    component ncl_register_D
        generic(width : integer;initial_value: integer);
        port(D: in dual_rail_logic_vector(width-1 downto 0);
            ki: in std_logic;
            rst: in std_logic;
            Q: out dual_rail_logic_vector(width-1 downto 0);
            ko : out std_logic_vector(width-1 downto 0));
    end component;
    component comp
        generic(width : integer);
        port(a: IN std_logic_vector(width-1 downto 0);
            ko: OUT std_logic);
    end component;
    component th44x0
        port(a: in std_logic;
            b: in std_logic;
            c: in std_logic;
            d: in std_logic;
            z: out std_logic );
    end component;
begin
    reg1 : ncl_register_D
        generic map(width => 32,initial_value => -4)
        port map(DT1,ki,rst,Q1,ko1);
    cd1 : comp
        generic map(width => 32)
        port map(ko1,ao1);
    reg2 : ncl_register_D
        generic map(width => 32,initial_value => -4)
        port map(DT2,ki,rst,Q2,ko2);
    cd2 : comp
        generic map(width => 32)
        port map(ko2,ao2);
    reg3 : ncl_register_D
        generic map(width => 28,initial_value => -4)
        port map(DT3,ki,rst,Q3,ko3);
    cd3 : comp
        generic map(width => 28)
        port map(ko3,ao3);
    reg4 : ncl_register_D
        generic map(width => 28,initial_value => -4)
        port map(DT4,ki,rst,Q4,ko4);
    cd4 : comp
        generic map(width => 28)
        port map(ko4,ao4);
    g4 : th44x0 port map(ao1,ao2,ao3,ao4,ko);
end Behavioral;

```

regcompdet.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.ncl_signals.all;

```

```

entity regcompdet is
    port ( DT1 : in dual_rail_logic_vector(1 to 32);
          DT2 : in dual_rail_logic_vector(1 to 32);
          rst : in std_logic;
          ki : in std_logic;
          Q1 : out dual_rail_logic_vector(1 to 32);
          Q2 : out dual_rail_logic_vector(1 to 32);
          ko : out std_logic);
end regcompdet;

architecture Behavioral of regcompdet is
    signal kol,ko2 : std_logic_vector(31 downto 0);
    signal aol,ao2 : std_logic;
    component ncl_register_D
        generic(width : integer;initial_value: integer);
        port(D: in dual_rail_logic_vector(width-1 downto 0);
             ki: in std_logic;
             rst: in std_logic;
             Q: out dual_rail_logic_vector(width-1 downto 0);
             ko : out std_logic_vector(width-1 downto 0));
    end component;
    component comp
        generic(width : integer);
        port(a: IN std_logic_vector(width-1 downto 0);
             ko: OUT std_logic);
    end component;
    component th22x0
        port(a: in std_logic;
             b: in std_logic;
             z: out std_logic );
    end component;

    begin
    reg1 : ncl_register_D
        generic map(width => 32,initial_value => -4)
        port map(DT1,ki,rst,Q1,kol);
    cd1 : comp
        generic map(width => 32)
        port map(kol,aol);
    reg2 : ncl_register_D
        generic map(width => 32,initial_value => -4)
        port map(DT2,ki,rst,Q2,ko2);
    cd2 : comp
        generic map(width => 32)
        port map(ko2,ao2);
    g3 : th22x0 port map(aol,ao2,ko);
    end Behavioral;

```

finalregandcompdet.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.ncl_signals.all;

entity finalregandcompdet is
    port ( D : in dual_rail_logic_vector(1 to 64);
          rst : in std_logic;
          ki : in std_logic;
          Q : out dual_rail_logic_vector(1 to 64);
          ko : out std_logic);
end finalregandcompdet;

architecture Behavioral of finalregandcompdet is
    signal kol : std_logic_vector(63 downto 0);
    component ncl_register_D
        generic(width : integer;initial_value: integer);
        port(D: in dual_rail_logic_vector(width-1 downto 0);
             ki: in std_logic;

```



```

        rst: in std_logic;
        Q: out dual_rail_logic_vector(width-1 downto 0);
        ko : out std_logic_vector(width-1 downto 0));
end component;
component comp
    generic(width : integer);
    port(a: IN std_logic_vector(width-1 downto 0);
        ko: OUT std_logic);
end component;
begin
reg1 : ncl_register_D
    generic map(width => 64, initial_value => -4)
    port map(D, ki, rst, Q, ko1);
cd1 : comp
    generic map(width => 64)
    port map(ko1, ko);
end Behavioral;

```

initround.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.ncl_signals.all;

entity initround is
    Port ( pt : in dual_rail_logic_vector(1 to 64);
        key : in dual_rail_logic_vector(1 to 64);
        L : out dual_rail_logic_vector(1 to 32);
        R : out dual_rail_logic_vector(1 to 32);
        C : out dual_rail_logic_vector(1 to 28);
        D : out dual_rail_logic_vector(1 to 28));
end initround;

architecture Behavioral of initround is
    signal L0,R0,f,Ro : dual_rail_logic_vector(1 to 32);
    signal C0,D0,C1,D1 : dual_rail_logic_vector(1 to 28);
    signal ao,bo : dual_rail_logic_vector(1 to 48);
    signal e1,e2,e3,e4,e5,e6,e7,e8 : dual_rail_logic_vector(1 to 6);
    signal y1,y2,y3,y4,y5,y6,y7,y8 : dual_rail_logic_vector(1 to 4);
    component ip
        port( pt : in dual_rail_logic_vector(1 to 64);
            L0 : out dual_rail_logic_vector(1 to 32);
            R0 : out dual_rail_logic_vector(1 to 32));
    end component;
    component pcl
        port( key : in dual_rail_logic_vector(1 to 64);
            C0 : out dual_rail_logic_vector(1 to 28);
            D0 : out dual_rail_logic_vector(1 to 28));
    end component;
    component shifter1
        port ( C0 : in dual_rail_logic_vector(1 to 28);
            D0 : in dual_rail_logic_vector(1 to 28);
            C1 : out dual_rail_logic_vector(1 to 28);
            D1 : out dual_rail_logic_vector(1 to 28));
    end component;
    component expl
        port ( R0 : in dual_rail_logic_vector(1 to 32);
            bo : out dual_rail_logic_vector(1 to 48));
    end component;
    component pc2
        port( C1 : in dual_rail_logic_vector(1 to 28);
            D1 : in dual_rail_logic_vector(1 to 28);
            ao : out dual_rail_logic_vector(1 to 48));
    end component;
    component xord11
        port ( ao : in dual_rail_logic_vector(1 to 48);
            bo : in dual_rail_logic_vector(1 to 48);
            e1,e2,e3,e4,e5,e6,e7,e8 : out dual_rail_logic_vector(1 to 6));
    end component;

```

```

component s1
  port ( b : in dual_rail_logic_vector(1 to 6);
        s : out dual_rail_logic_vector(1 to 4));
end component;
component s2
  port ( b : in dual_rail_logic_vector(1 to 6);
        s : out dual_rail_logic_vector(1 to 4));
end component;
component s3
  port ( b : in dual_rail_logic_vector(1 to 6);
        s : out dual_rail_logic_vector(1 to 4));
end component;
component s4
  port ( b : in dual_rail_logic_vector(1 to 6);
        s : out dual_rail_logic_vector(1 to 4));
end component;
component s5
  port ( b : in dual_rail_logic_vector(1 to 6);
        s : out dual_rail_logic_vector(1 to 4));
end component;
component s6
  port ( b : in dual_rail_logic_vector(1 to 6);
        s : out dual_rail_logic_vector(1 to 4));
end component;
component s7
  port ( b : in dual_rail_logic_vector(1 to 6);
        s : out dual_rail_logic_vector(1 to 4));
end component;
component s8
  port ( b : in dual_rail_logic_vector(1 to 6);
        s : out dual_rail_logic_vector(1 to 4));
end component;
component fp
  port ( y1,y2,y3,y4,y5,y6,y7,y8 : in dual_rail_logic_vector(1 to 4);
        f : out dual_rail_logic_vector(1 to 32));
end component;
component xordl2
  port ( ao : in dual_rail_logic_vector(1 to 32);
        bo : in dual_rail_logic_vector(1 to 32);
        z : out dual_rail_logic_vector(1 to 32));
end component;

begin
comp1 : ip port map(pt,L0,R0);
L <= R0;
comp2 : pc1 port map(key,C0,D0);
comp3 : shifter1 port map(C0,D0,C1,D1);
C <= C1; D <= D1;
comp4 : exp1 port map(R0,bo);
comp5 : pc2 port map(C1,D1,ao);
comp6 : xordl1 port map(ao,bo,e1,e2,e3,e4,e5,e6,e7,e8);
comp7 : s1 port map(e1,y1);
comp8 : s2 port map(e2,y2);
comp9 : s3 port map(e3,y3);
comp10 : s4 port map(e4,y4);
comp11 : s5 port map(e5,y5);
comp12 : s6 port map(e6,y6);
comp13 : s7 port map(e7,y7);
comp14 : s8 port map(e8,y8);
comp15 : fp port map(y1,y2,y3,y4,y5,y6,y7,y8,f);
comp16 : xordl2 port map(f,L0,R);

end Behavioral;

ip.vhd
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.ncl_signals.all;

```

```

entity ip is
    port( pt : in dual_rail_logic_vector(1 to 64);
          L0 : out dual_rail_logic_vector(1 to 32);
          R0 : out dual_rail_logic_vector(1 to 32));
end ip;

architecture Behavioral of ip is

begin
    L0(1) <= pt(58); L0(2) <= pt(50); L0(3) <= pt(42); L0(4) <= pt(34); L0(5) <= pt(26);
    L0(6) <= pt(18); L0(7) <= pt(10); L0(8) <= pt(2); L0(9) <= pt(60); L0(10) <= pt(52);
    L0(11) <= pt(44); L0(12) <= pt(36); L0(13) <= pt(28); L0(14) <= pt(20);
    L0(15) <= pt(12); L0(16) <= pt(4); L0(17) <= pt(62); L0(18) <= pt(54);
    L0(19) <= pt(46); L0(20) <= pt(38); L0(21) <= pt(30); L0(22) <= pt(22);
    L0(23) <= pt(14); L0(24) <= pt(6); L0(25) <= pt(64); L0(26) <= pt(56);
    L0(27) <= pt(48); L0(28) <= pt(40); L0(29) <= pt(32); L0(30) <= pt(24);
    L0(31) <= pt(16); L0(32) <= pt(8);
    R0(1) <= pt(57); R0(2) <= pt(49); R0(3) <= pt(41); R0(4) <= pt(33); R0(5) <= pt(25);
    R0(6) <= pt(17); R0(7) <= pt(9); R0(8) <= pt(1); R0(9) <= pt(59); R0(10) <= pt(51);
    R0(11) <= pt(43); R0(12) <= pt(35); R0(13) <= pt(27); R0(14) <= pt(19);
    R0(15) <= pt(11); R0(16) <= pt(3); R0(17) <= pt(61); R0(18) <= pt(53);
    R0(19) <= pt(45); R0(20) <= pt(37); R0(21) <= pt(29); R0(22) <= pt(21);
    R0(23) <= pt(13); R0(24) <= pt(5); R0(25) <= pt(63); R0(26) <= pt(55);
    R0(27) <= pt(47); R0(28) <= pt(39); R0(29) <= pt(31); R0(30) <= pt(23);
    R0(31) <= pt(15); R0(32) <= pt(7);
end Behavioral;

```

PC1.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.ncl_signals.all;

entity pc1 is
    port( key : in dual_rail_logic_vector(1 to 64);
          C0 : out dual_rail_logic_vector(1 to 28);
          D0 : out dual_rail_logic_vector(1 to 28));
end pc1;

architecture Behavioral of pc1 is
    signal XX : dual_rail_logic_vector(1 to 56);
begin
    XX(1) <= key(57); XX(2) <= key(49); XX(3) <= key(41); XX(4) <= key(33);
    XX(5) <= key(25); XX(6) <= key(17); XX(7) <= key(9); XX(8) <= key(1);
    XX(9) <= key(58); XX(10) <= key(50); XX(11) <= key(42); XX(12) <= key(34);
    XX(13) <= key(26); XX(14) <= key(18); XX(15) <= key(10); XX(16) <= key(2);
    XX(17) <= key(59); XX(18) <= key(51); XX(19) <= key(43); XX(20) <= key(35);
    XX(21) <= key(27); XX(22) <= key(19); XX(23) <= key(11); XX(24) <= key(3);
    XX(25) <= key(60); XX(26) <= key(52); XX(27) <= key(44); XX(28) <= key(36);
    XX(29) <= key(63); XX(30) <= key(55); XX(31) <= key(47); XX(32) <= key(39);
    XX(33) <= key(31); XX(34) <= key(23); XX(35) <= key(15); XX(36) <= key(7);
    XX(37) <= key(62); XX(38) <= key(54); XX(39) <= key(46); XX(40) <= key(38);
    XX(41) <= key(30); XX(42) <= key(22); XX(43) <= key(14); XX(44) <= key(6);
    XX(45) <= key(61); XX(46) <= key(53); XX(47) <= key(45); XX(48) <= key(37);
    XX(49) <= key(29); XX(50) <= key(21); XX(51) <= key(13); XX(52) <= key(5);
    XX(53) <= key(28); XX(54) <= key(20); XX(55) <= key(12); XX(56) <= key(4);
    C0 <= XX(1 to 28); D0 <= XX(29 to 56);
end Behavioral;

```

shifter1.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.ncl_signals.all;

entity shifter1 is

```

```

    port ( C0 : in dual_rail_logic_vector(1 to 28);
          D0 : in dual_rail_logic_vector(1 to 28);
          C1 : out dual_rail_logic_vector(1 to 28);
          D1 : out dual_rail_logic_vector(1 to 28));
end shifter1;

```

architecture Behavioral of shifter1 is

```

begin
C1(1)<=C0(2);C1(2)<=C0(3);C1(3)<=C0(4);C1(4)<=C0(5);C1(5)<=C0(6);
C1(6)<=C0(7);C1(7)<=C0(8);C1(8)<=C0(9);C1(9)<=C0(10);C1(10)<=C0(11);
C1(11)<=C0(12);C1(12)<=C0(13);C1(13)<=C0(14);C1(14)<=C0(15);
C1(15)<=C0(16);C1(16)<=C0(17);C1(17)<=C0(18);C1(18)<=C0(19);
C1(19)<=C0(20);C1(20)<=C0(21);C1(21)<=C0(22);C1(22)<=C0(23);
C1(23)<=C0(24);C1(24)<=C0(25);C1(25)<=C0(26);C1(26)<=C0(27);
C1(27)<=C0(28);C1(28)<=C0(1);
D1(1)<=D0(2);D1(2)<=D0(3);D1(3)<=D0(4);D1(4)<= D0(5);D1(5)<=D0(6);
D1(6)<=D0(7);D1(7)<=D0(8);D1(8)<=D0(9);D1(9)<=D0(10);D1(10)<=D0(11);
D1(11)<=D0(12);D1(12)<=D0(13);D1(13)<=D0(14);D1(14)<=D0(15);
D1(15)<=D0(16);D1(16)<=D0(17);D1(17)<=D0(18);D1(18)<=D0(19);
D1(19)<=D0(20);D1(20)<=D0(21);D1(21)<=D0(22);D1(22)<=D0(23);
D1(23)<=D0(24);D1(24)<=D0(25);D1(25)<=D0(26);D1(26)<=D0(27);
D1(27)<=D0(28);D1(28)<=D0(1);
end Behavioral;

```

expl.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.ncl_signals.all;

entity expl is
    port ( R0 : in dual_rail_logic_vector(1 to 32);
          bo : out dual_rail_logic_vector(1 to 48));
end expl;

```

architecture Behavioral of expl is

```

begin
bo(1)<=R0(32);bo(2)<=R0(1);bo(3)<=R0(2);bo(4)<=R0(3);bo(5)<=R0(4);
bo(6)<=R0(5);bo(7)<=R0(4);bo(8)<=R0(5);bo(9)<=R0(6);bo(10)<=R0(7);
bo(11)<=R0(8);bo(12)<=R0(9);bo(13)<=R0(8);bo(14)<=R0(9);bo(15)<=R0(10);
bo(16)<=R0(11);bo(17)<=R0(12);bo(18)<=R0(13);bo(19)<=R0(12);
bo(20)<=R0(13);bo(21)<=R0(14);bo(22)<=R0(15);bo(23)<=R0(16);
bo(24)<=R0(17);bo(25)<=R0(16);bo(26)<=R0(17);bo(27)<=R0(18);
bo(28)<=R0(19);bo(29)<=R0(20);bo(30)<=R0(21);bo(31)<=R0(20);
bo(32)<=R0(21);bo(33)<=R0(22);bo(34)<=R0(23);bo(35)<=R0(24);
bo(36)<=R0(25);bo(37)<=R0(24);bo(38)<=R0(25);bo(39)<=R0(26);
bo(40)<=R0(27);bo(41)<=R0(28);bo(42)<=R0(29);bo(43)<=R0(28);
bo(44)<=R0(29);bo(45)<=R0(30);bo(46)<=R0(31);bo(47)<=R0(32);
bo(48)<=R0(1);
end Behavioral;

```

pc2.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.ncl_signals.all;

entity pc2 is
    port( C1 : in dual_rail_logic_vector(1 to 28);
          D1 : in dual_rail_logic_vector(1 to 28);
          ao : out dual_rail_logic_vector(1 to 48));
end pc2;

```

architecture Behavioral of pc2 is

```

signal YY : dual_rail_logic_vector(1 to 56);

```

```

begin
YY(1 to 28) <= C1(1 to 28); YY(29 to 56) <= D1(1 to 28);

ao(1)<=YY(14);ao(2)<=YY(17);ao(3)<=YY(11);ao(4)<=YY(24);ao(5)<=YY(1);
ao(6)<=YY(5);ao(7)<=YY(3);ao(8)<=YY(28);ao(9)<=YY(15);ao(10)<=YY(6);
ao(11)<=YY(21);ao(12)<=YY(10);ao(13)<=YY(23);ao(14)<=YY(19);
ao(15)<=YY(12);ao(16)<=YY(4);ao(17)<=YY(26);ao(18)<=YY(8);
ao(19)<=YY(16);ao(20)<=YY(7);ao(21)<=YY(27);ao(22)<=YY(20);
ao(23)<=YY(13);ao(24)<=YY(2);ao(25)<=YY(41);ao(26)<=YY(52);
ao(27)<=YY(31);ao(28)<=YY(37);ao(29)<=YY(47);ao(30)<=YY(55);
ao(31)<=YY(30);ao(32)<=YY(40);ao(33)<=YY(51);ao(34)<=YY(45);
ao(35)<=YY(33);ao(36)<=YY(48);ao(37)<=YY(44);ao(38)<=YY(49);
ao(39)<=YY(39);ao(40)<=YY(56);ao(41)<=YY(34);ao(42)<=YY(53);
ao(43)<=YY(46);ao(44)<=YY(42);ao(45)<=YY(50);ao(46)<=YY(36);
ao(47)<=YY(29);ao(48)<=YY(32);
end Behavioral;

```

xordl1.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.ncl_signals.all;

entity xordl1 is
    port( ao : in dual_rail_logic_vector(1 to 48);
          bo : in dual_rail_logic_vector(1 to 48);
          e1,e2,e3,e4,e5,e6,e7,e8 : out dual_rail_logic_vector(1 to 6));
end xordl1;

architecture Behavioral of xordl1 is
    signal e : dual_rail_logic_vector(1 to 48);
    component exor
        port( x : in dual_rail_logic;
              y : in dual_rail_logic;
              z : out dual_rail_logic);
    end component;
begin
    g1 : for i in 1 to 48 generate
        h1 : exor port map(ao(i),bo(i),e(i));
    end generate;
    e1(1 to 6)<=e(1 to 6);e2(1 to 6)<=e(7 to 12);
    e3(1 to 6)<=e(13 to 18);e4(1 to 6)<=e(19 to 24);
    e5(1 to 6)<=e(25 to 30);e6(1 to 6)<=e(31 to 36);
    e7(1 to 6)<=e(37 to 42);e8(1 to 6)<=e(43 to 48);
end Behavioral;

```

s1.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.ncl_signals.all;

entity s1 is
    port ( b : in dual_rail_logic_vector(1 to 6);
          s : out dual_rail_logic_vector(1 to 4));
end s1;

architecture Behavioral of s1 is
begin
    process(b)
    begin
        if b(1).rail0='0' and b(1).rail1='0' and b(2).rail0='0' and b(2).rail1='0' and
           b(3).rail0='0' and b(3).rail1='0' and b(4).rail0='0' and b(4).rail1='0' and
           b(5).rail0='0' and b(5).rail1='0' and b(6).rail0='0' and b(6).rail1='0' then -- for null values
            s(1).rail0 <= '0';s(2).rail0 <= '0';s(3).rail0 <= '0';s(4).rail0 <= '0';
            s(1).rail1 <= '0';s(2).rail1 <= '0';s(3).rail1 <= '0';s(4).rail1 <= '0';
        end if;
    end process;
end Behavioral;

```

[illegible]


```

        b(2).rail0='0' and b(3).rail0='1' and b(4).rail0='1' and b(5).rail0='0' then
            s(1).rail1 <= '1';s(2).rail1 <= '0';s(3).rail1 <= '1';s(4).rail1 <= '1';
            s(1).rail0 <= '0';s(2).rail0 <= '1';s(3).rail0 <= '0';s(4).rail0 <= '0';
        elsif b(2).rail1='1' and b(3).rail1='0' and b(4).rail1='1' and b(5).rail1='0' and--"1010"
            b(2).rail0='0' and b(3).rail0='1' and b(4).rail0='0' and b(5).rail0='1' then
                s(1).rail1 <= '0';s(2).rail1 <= '0';s(3).rail1 <= '1';s(4).rail1 <= '1';
                s(1).rail0 <= '1';s(2).rail0 <= '1';s(3).rail0 <= '0';s(4).rail0 <= '0';
            elsif b(2).rail1='1' and b(3).rail1='0' and b(4).rail1='1' and b(5).rail1='1' and--"1011"
                b(2).rail0='0' and b(3).rail0='1' and b(4).rail0='0' and b(5).rail0='0' then
                    s(1).rail1 <= '1';s(2).rail1 <= '1';s(3).rail1 <= '1';s(4).rail1 <= '0';
                    s(1).rail0 <= '0';s(2).rail0 <= '0';s(3).rail0 <= '0';s(4).rail0 <= '1';
            elsif b(2).rail1='1' and b(3).rail1='1' and b(4).rail1='0' and b(5).rail1='1' and--"1101"
                b(2).rail0='0' and b(3).rail0='0' and b(4).rail0='1' and b(5).rail0='0' then
                    s(1).rail1 <= '0';s(2).rail1 <= '0';s(3).rail1 <= '0';s(4).rail1 <= '0';
                    s(1).rail0 <= '1';s(2).rail0 <= '1';s(3).rail0 <= '1';s(4).rail0 <= '1';
            elsif b(2).rail1='1' and b(3).rail1='1' and b(4).rail1='1' and b(5).rail1='0' and--"1110"
                b(2).rail0='0' and b(3).rail0='0' and b(4).rail0='0' and b(5).rail0='1' then
                    s(1).rail1 <= '0';s(2).rail1 <= '1';s(3).rail1 <= '1';s(4).rail1 <= '0';
                    s(1).rail0 <= '1';s(2).rail0 <= '0';s(3).rail0 <= '0';s(4).rail0 <= '1';
            elsif b(2).rail1='1' and b(3).rail1='1' and b(4).rail1='1' and b(5).rail1='1' and--"1111"
                b(2).rail0='0' and b(3).rail0='0' and b(4).rail0='0' and b(5).rail0='0' then
                    s(1).rail1 <= '1';s(2).rail1 <= '1';s(3).rail1 <= '1';s(4).rail1 <= '1';
                    s(1).rail0 <= '0';s(2).rail0 <= '0';s(3).rail0 <= '1';s(4).rail0 <= '0';
            elsif b(2).rail1='1' and b(3).rail1='1' and b(4).rail1='0' and b(5).rail1='0' and--"1100"
                b(2).rail0='0' and b(3).rail0='0' and b(4).rail0='1' and b(5).rail0='1' then
                    s(1).rail1 <= '1';s(2).rail1 <= '0';s(3).rail1 <= '1';s(4).rail1 <= '0';
                    s(1).rail0 <= '0';s(2).rail0 <= '1';s(3).rail0 <= '0';s(4).rail0 <= '1';
            end if;
        end if;

end if;

end process;
end Behavioral;

```

fp.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.ncl_signals.all;

entity fp is
    port ( y1,y2,y3,y4,y5,y6,y7,y8 : in dual_rail_logic_vector(1 to 4);
           f : out dual_rail_logic_vector(1 to 32));
end fp;

architecture Behavioral of fp is
    signal ww : dual_rail_logic_vector(1 to 32);
begin

    ww(1 to 4)<=y1(1 to 4);ww(5 to 8)<=y2(1 to 4);ww(9 to 12)<=y3(1 to 4);
    ww(13 to 16)<=y4(1 to 4);ww(17 to 20)<=y5(1 to 4);
    ww(21 to 24)<=y6(1 to 4);ww(25 to 28)<=y7(1 to 4);
    ww(29 to 32)<=y8(1 to 4);

    f(1)<=ww(16);f(2)<=ww(7);f(3)<=ww(20);f(4)<=ww(21);f(5)<=ww(29);
    f(6)<=ww(12);f(7)<=ww(28);f(8)<=ww(17);f(9)<=ww(1);f(10)<=ww(15);
    f(11)<=ww(23);f(12)<=ww(26);f(13)<=ww(5);f(14)<=ww(18);f(15)<=ww(31);
    f(16)<=ww(10);f(17)<=ww(2);f(18)<=ww(8);f(19)<=ww(24);f(20)<=ww(14);
    f(21)<=ww(32);f(22)<=ww(27);f(23)<=ww(3);f(24)<=ww(9);f(25)<=ww(19);
    f(26)<=ww(13);f(27)<=ww(30);f(28)<=ww(6);f(29)<=ww(22);f(30)<=ww(11);
    f(31)<=ww(4);f(32)<=ww(25);
end Behavioral;

```

xordl2.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.ncl_signals.all;

entity xordl2 is
    port( ao : in dual_rail_logic_vector(1 to 32);
          bo : in dual_rail_logic_vector(1 to 32);

```

```

        z : out dual_rail_logic_vector(1 to 32));
end xordl2;

architecture Behavioral of xordl2 is
component exor
    port( x : in dual_rail_logic;
          y : in dual_rail_logic;
          z : out dual_rail_logic);
end component;
begin
    g2 : for i in 1 to 32 generate
        h2 : exor port map(ao(i),bo(i),z(i));
    end generate;

end Behavioral;

```

round1to14s1.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.ncl_signals.all;

entity round1to14s1 is
    port ( Li : in dual_rail_logic_vector(1 to 32);
          Ri : in dual_rail_logic_vector(1 to 32);
          Ci : in dual_rail_logic_vector(1 to 28);
          Di : in dual_rail_logic_vector(1 to 28);
          Lo : out dual_rail_logic_vector(1 to 32);
          Ro : out dual_rail_logic_vector(1 to 32);
          Co : out dual_rail_logic_vector(1 to 28);
          Do : out dual_rail_logic_vector(1 to 28));
end round1to14s1;

architecture Behavioral of round1to14s1 is
    signal L,R,f : dual_rail_logic_vector(1 to 32);
    signal C,D : dual_rail_logic_vector(1 to 28);
    signal ao,bo : dual_rail_logic_vector(1 to 48);
    signal e1,e2,e3,e4,e5,e6,e7,e8 : dual_rail_logic_vector(1 to 6);
    signal y1,y2,y3,y4,y5,y6,y7,y8 : dual_rail_logic_vector(1 to 4);
    component shifter1
        port ( C0 : in dual_rail_logic_vector(1 to 28);
              D0 : in dual_rail_logic_vector(1 to 28);
              C1 : out dual_rail_logic_vector(1 to 28);
              D1 : out dual_rail_logic_vector(1 to 28));
    end component;
    component exp1
        port ( R0 : in dual_rail_logic_vector(1 to 32);
              bo : out dual_rail_logic_vector(1 to 48));
    end component;
    component pc2
        port( C1 : in dual_rail_logic_vector(1 to 28);
              D1 : in dual_rail_logic_vector(1 to 28);
              ao : out dual_rail_logic_vector(1 to 48));
    end component;
    component xordl1
        port ( ao : in dual_rail_logic_vector(1 to 48);
              bo : in dual_rail_logic_vector(1 to 48);
              e1,e2,e3,e4,e5,e6,e7,e8 : out dual_rail_logic_vector(1 to 6));
    end component;
    component s1
        port ( b : in dual_rail_logic_vector(1 to 6);
              s : out dual_rail_logic_vector(1 to 4));
    end component;
    component s2
        port ( b : in dual_rail_logic_vector(1 to 6);
              s : out dual_rail_logic_vector(1 to 4));
    end component;
    component s3
        port ( b : in dual_rail_logic_vector(1 to 6);

```

```

        s : out dual_rail_logic_vector(1 to 4));
end component;
component s4
    port ( b : in dual_rail_logic_vector(1 to 6);
          s : out dual_rail_logic_vector(1 to 4));
end component;
component s5
    port ( b : in dual_rail_logic_vector(1 to 6);
          s : out dual_rail_logic_vector(1 to 4));
end component;
component s6
    port ( b : in dual_rail_logic_vector(1 to 6);
          s : out dual_rail_logic_vector(1 to 4));
end component;
component s7
    port ( b : in dual_rail_logic_vector(1 to 6);
          s : out dual_rail_logic_vector(1 to 4));
end component;
component s8
    port ( b : in dual_rail_logic_vector(1 to 6);
          s : out dual_rail_logic_vector(1 to 4));
end component;
component fp
    port ( y1,y2,y3,y4,y5,y6,y7,y8 : in dual_rail_logic_vector(1 to 4);
          f : out dual_rail_logic_vector(1 to 32));
end component;
component xordl2
    port ( ao : in dual_rail_logic_vector(1 to 32);
          bo : in dual_rail_logic_vector(1 to 32);
          z : out dual_rail_logic_vector(1 to 32));
end component;

begin
    comp1 : shifter1 port map(Ci,Di,C,D);
    Co <= C; Do <= D;
    L <= Li; R <= Ri;
    Lo <= R;
    comp2 : exp1 port map(R,bo);
    comp3 : pc2 port map(C,D,ao);
    comp4 : xordl1 port map(ao,bo,e1,e2,e3,e4,e5,e6,e7,e8);
    comp5 : s1 port map(e1,y1);
    comp6 : s2 port map(e2,y2);
    comp7 : s3 port map(e3,y3);
    comp8 : s4 port map(e4,y4);
    comp9 : s5 port map(e5,y5);
    comp10 : s6 port map(e6,y6);
    comp11 : s7 port map(e7,y7);
    comp12 : s8 port map(e8,y8);
    comp13 : fp port map(y1,y2,y3,y4,y5,y6,y7,y8,f);
    comp14 : xordl2 port map(f,L,Ro);

```

end Behavioral;

finalround.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.ncl_signals.all;

entity finalround is
    port ( L16 : in dual_rail_logic_vector(1 to 32);
          R16 : in dual_rail_logic_vector(1 to 32);
          ct : out dual_rail_logic_vector(1 to 64));
end finalround;

architecture Behavioral of finalround is
    signal uu : dual_rail_logic_vector(1 to 64);
begin
    -- process(uu)

```

```

-- begin
uu(1 to 32) <= L16; uu(33 to 64) <= R16;

ct(1)<=uu(40);ct(2)<=uu(8);ct(3)<=uu(48);ct(4)<=uu(16);
ct(5)<=uu(56);ct(6)<=uu(24);ct(7)<=uu(64);ct(8)<=uu(32);
ct(9)<=uu(39);ct(10)<=uu(7);ct(11)<=uu(47);ct(12)<=uu(15);
ct(13)<=uu(55);ct(14)<=uu(23);ct(15)<=uu(63);ct(16)<=uu(31);
ct(17)<=uu(38);ct(18)<=uu(6);ct(19)<=uu(46);ct(20)<=uu(14);
ct(21)<=uu(54);ct(22)<=uu(22);ct(23)<=uu(62);ct(24)<=uu(30);
ct(25)<=uu(37);ct(26)<=uu(5);ct(27)<=uu(45);ct(28)<=uu(13);
ct(29)<=uu(53);ct(30)<=uu(21);ct(31)<=uu(61);ct(32)<=uu(29);
ct(33)<=uu(36);ct(34)<=uu(4);ct(35)<=uu(44);ct(36)<=uu(12);
ct(37)<=uu(52);ct(38)<=uu(20);ct(39)<=uu(60);ct(40)<=uu(28);
ct(41)<=uu(35);ct(42)<=uu(3);ct(43)<=uu(43);ct(44)<=uu(11);
ct(45)<=uu(51);ct(46)<=uu(19);ct(47)<=uu(59);ct(48)<=uu(27);
ct(49)<=uu(34);ct(50)<=uu(2);ct(51)<=uu(42);ct(52)<=uu(10);
ct(53)<=uu(50);ct(54)<=uu(18);ct(55)<=uu(58);ct(56)<=uu(26);
ct(57)<=uu(33);ct(58)<=uu(1);ct(59)<=uu(41);ct(60)<=uu(9);
ct(61)<=uu(49);ct(62)<=uu(17);ct(63)<=uu(57);ct(64)<=uu(25);
-- end process;
end Behavioral;

```

VHDL Files for Soft Error Tolerant Designs

asynctfulladder.vhd

```

library ieee;
use ieee.std_logic_1164.all;
use work.ncl_signals.all;

entity asynctfulladder is
port( clk : in std_logic;
      rst : in std_logic;
      Qi : out dual_rail_logic_vector(1 to 2);
      Qo : out dual_rail_logic_vector(1 to 2));
end asynctfulladder;

architecture behavior of asynctfulladder is
signal D : dual_rail_logic_vector(1 to 3);
signal Ipul : std_logic;
component signal_gen1
  port( clock : in std_logic;
        Di : out dual_rail_logic_vector(1 to 3);
        strk : out std_logic);
end component;
component asynctfa
  port ( Di : in dual_rail_logic_vector(1 to 3);
        I : in std_logic;
        rst : in std_logic;
        Qi : out dual_rail_logic_vector(1 to 2);
        Qo : out dual_rail_logic_vector(1 to 2));
end component;
begin
c1 : signal_gen1 port map(clk,D,Ipul);
c2 : asynctfa port map(D,Ipul,rst,Qi,Qo);
end behavior;

```

signal_gen1.vhd

```

library ieee;
use ieee.std_logic_1164.all;
use work.ncl_signals.all;

entity signal_gen1 is
  port( clock : in std_logic;
        Di : out dual_rail_logic_vector(1 to 3);
        strk : out std_logic
      );
end signal_gen1;

```

```

architecture behavior of signal_gen1 is
begin
  incrementer: process is
    variable count_value: natural:=0;
    begin
      wait until clock = '1';
      count_value := (count_value+1) mod 16;
      Di(3).rail0<='0';Di(2).rail1<='0';Di(1).rail1<='0'; case count_value is
        when 1 to 8 =>
          Di(3).rail1 <='1';
        when others =>
          Di(3).rail1 <='0';
        end case;

    case count_value is
      when 3 to 10 =>
        Di(2).rail0 <='1';
      when others =>
        Di(2).rail0 <='0';
      end case;
    case count_value is
      when 5 to 12 =>
        Di(1).rail0 <='1';
      when others =>
        Di(1).rail0 <='0';
      end case;
    case count_value is
      when 3 =>
        strk <='1';
      when others =>
        strk <='0';
      end case;

  end process incrementer;

end behavior;

```

asyncfa.vhd

```

Library IEEE;
Use IEEE.std_logic_1164.all;
Use work.ncl_signals.all;

entity asyncfa is
  port ( Di : in dual_rail_logic_vector(1 to 3);
        I : in std_logic;
        rst : in std_logic;
        Qi : out dual_rail_logic_vector(1 to 2);
        Qo : out dual_rail_logic_vector(1 to 2));
end asyncfa;

architecture behavioral of asyncfa is
  signal se,kf,kro,ki,kri,kfo : std_logic;
  signal Qia,Qic : dual_rail_logic_vector(1 to 3);
  signal sc,Qins : dual_rail_logic_vector(1 to 2);
  component initreg is
    port ( D : in dual_rail_logic_vector(1 to 3);
          ki : in std_logic;
          rst : in std_logic;
          Q : out dual_rail_logic_vector(1 to 3);
          ko : out std_logic);
  end component;
  component finalreg is
    port ( D : in dual_rail_logic_vector(1 to 2);
          ki : in std_logic;
          rst : in std_logic;
          Q : out dual_rail_logic_vector(1 to 2);
          ko : out std_logic);
  end component;
  component fulladder is

```

```

    port ( a : in dual_rail_logic_vector(1 to 3);
          strk : in std_logic;
          s : out dual_rail_logic_vector(1 to 2));
end component;
component andgates is
    port ( data : in dual_rail_logic_vector(1 to 3);
          q : in std_logic;
          output : out dual_rail_logic_vector(1 to 3));
end component;
component rstcircuit
    port ( faout : in dual_rail_logic_vector(1 to 2);
          kinsout : in std_logic;
          kinitial : in std_logic;
          kinsin : out std_logic;
          q : out std_logic);
end component;
component agate
    port ( a : in std_logic;
          b : in std_logic;
          c : out std_logic);
end component;
begin
    reg1 : initreg port map(Di,kf,rst,Qia,ki);
    a1 : andgates port map(Qia,se,Qic);
    comb1 : fulladder port map(Qic,I,sc);
    Qi<=sc;
    a2 : rstcircuit port map(sc,kro,ki,kri,se);
    regi : finalreg port map(sc,kri,rst,Qins,kro);
    reg2 : finalreg port map(Qins,kfo,rst,Qo,kf);
    a3 : agate port map(kf,se,kfo);
end behavioral;

```

initreg.vhd

```

Library IEEE;
Use IEEE.std_logic_1164.all;
Use work.ncl_signals.all;

entity initreg is
    port ( D : in dual_rail_logic_vector(1 to 3);
          ki : in std_logic;
          rst : in std_logic;
          Q : out dual_rail_logic_vector(1 to 3);
          ko : out std_logic);
end initreg;

architecture behavioral of initreg is
    signal ao : std_logic_vector(1 to 3);
    component ncl_register_D
        generic(width : integer;initial_value: integer); -- 1=DATA1, 0=DATA0, -4=NULL
        port(D: in dual_rail_logic_vector(width-1 downto 0);
             ki: in std_logic;
             rst: in std_logic;
             Q: out dual_rail_logic_vector(width-1 downto 0);
             ko : out std_logic_vector(width-1 downto 0));
    end component;
    component th33x0
        port(a: in std_logic;
             b: in std_logic;
             c: in std_logic;
             z: out std_logic);
    end component;
    begin
        regi : ncl_register_D generic map(width=>3,initial_value=>-4)
            port map(D,ki,rst,Q,ao);
        cdi : th33x0 port map(ao(3),ao(1),ao(2),ko);
    end behavioral;

```

finalreg.vhd

```

Library IEEE;

```

```

Use IEEE.std_logic_1164.all;
Use work.ncl_signals.all;

entity finalreg is
  port ( D : in dual_rail_logic_vector(1 to 2);
        ki : in std_logic;
        rst : in std_logic;
        Q : out dual_rail_logic_vector(1 to 2);
        ko : out std_logic);
end finalreg;

architecture behavioral of finalreg is
  signal ao : std_logic_vector(1 to 2);
  component ncl_register_D
    generic(width : integer;initial_value: integer); -- 1=DATA1, 0=DATA0, -4=NULL
    port(D: in dual_rail_logic_vector(width-1 downto 0);
        ki: in std_logic;
        rst: in std_logic;
        Q: out dual_rail_logic_vector(width-1 downto 0);
        ko : out std_logic_vector(width-1 downto 0));
  end component;
  component th22x0
    port(a: in std_logic;
        b: in std_logic;
        z: out std_logic);
  end component;
begin
  regi : ncl_register_D generic map(width=>2,initial_value=>-4)
    port map(D,ki,rst,Q,ao);
  cdi : th22x0 port map(ao(2),ao(1),ko);
end behavioral;

```

fulladder.vhd

```

Library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.ncl_signals.all;

entity fulladder is
  port ( a : in dual_rail_logic_vector(1 to 3);
        strk : in std_logic;
        s : out dual_rail_logic_vector(1 to 2)
        );
end fulladder;

architecture beh of fulladder is
  signal c0,c1,cstrk : std_logic;
  component strike is
    port( data : in std_logic;
        i : in std_logic;
        ipulse : out std_logic);
  end component;
  component th23x0 is
    port( a : in std_logic;
        b: in std_logic;
        c: in std_logic;
        z: out std_logic );
  end component;
  component th34w2x0 is
    port(a: in std_logic; -- weight 2
        b: in std_logic;
        c: in std_logic;
        d: in std_logic;
        z: out std_logic );
  end component;
begin
  g0 : strike port map(a(1).rail1,strk,cstrk);
  g1 : th23x0 port map(a(1).rail0,a(2).rail0,a(3).rail0,c0);
  g2 : th23x0 port map(cstrk,a(2).rail1,a(3).rail1,c1);
  s(1).rail0<=c0;s(1).rail1<=c1;
  g3 : th34w2x0 port map(c1,a(1).rail0,a(2).rail0,a(3).rail0,s(2).rail0);

```

```

g4 : th34w2x0 port map(c0,cstrk,a(2).rail1,a(3).rail1,s(2).rail1);
end beh;

```

strike.vhd

```

library ieee;
use ieee.std_logic_1164.ALL;

entity strike is
    port( data : in std_logic;
          i    : in std_logic;
          ipulse : out std_logic);
end strike;

architecture beh of strike is
begin
    process(data,i)
    begin
        if i = '1' then
            ipulse <= i;
        else
            ipulse <= data;
        end if;
    end process;
end beh;

```

andgates.vhd

```

Library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.ncl_signals.all;

entity andgates is
    port ( data : in dual_rail_logic_vector(1 to 3);
          q    : in std_logic;
          output : out dual_rail_logic_vector(1 to 3)
        );
end andgates;

architecture beh of andgates is
begin
    process(data,q)
    begin
        for i in 1 to data'length loop
            output(i).rail0 <= data(i).rail0 and q;
            output(i).rail1 <= data(i).rail1 and q;
        end loop;
    end process;
end beh;

```

rstcircuit.vhd

```

Library IEEE;
Use IEEE.std_logic_1164.all;
Use work.ncl_signals.all;

entity rstcircuit is
    port ( faout : in dual_rail_logic_vector(1 to 2);
          kinsout : in std_logic;
          kinitial : in std_logic;
          kinsin : out std_logic;
          q : out std_logic);
end rstcircuit;

architecture behavioral of rstcircuit is
    signal qs,ka : std_logic;
    component sedetect is
        port( sc : in dual_rail_logic_vector(1 to 2);
              se : out std_logic);
    end component;
    component door1 is
        port( s1 : in std_logic;

```



```

        s2 : in std_logic;
        t  : out std_logic);
end component;
component door2 is
    port( m1 : in std_logic;
          m2 : in std_logic;
          n  : out std_logic);
end component;
begin
s1 : sedetect port map(faout,qs);
q<=qs;
a1 : door1 port map(kinitial,kinsout,ka);
a2 : door2 port map(ka,qs,kinsin);
end behavioral;

```

sedetect.vhd

```

Library IEEE;
Use IEEE.std_logic_1164.all;
Use work.ncl_signals.all;

entity sedetect is
    port( sc : in dual_rail_logic_vector(1 to 2);
          se : out std_logic);
end sedetect;
architecture beh of sedetect is
    signal det,det1,det2 : std_logic;
    component th22x0
        port ( a : in std_logic;
              b : in std_logic;
              z : out std_logic);
    end component;
    component th12bx0
        port ( a : in std_logic;
              b : in std_logic;
              zb : out std_logic);
    end component;
begin
    g1 : th22x0 port map(sc(1).rail0,sc(1).rail1,det1);
    g2 : th22x0 port map(sc(2).rail0,sc(2).rail1,det2);
    g3 : th12bx0 port map(det1,det2,se);
end beh;

```

door1.vhd

```

Library IEEE;
Use IEEE.std_logic_1164.all;

entity door1 is
    port( s1 : in std_logic;
          s2 : in std_logic;
          t  : out std_logic;
          -- k1 : out std_logic);--test
end door1;
architecture beh of door1 is
    signal si,ti : std_logic;
begin
    process(s1,s2)
    begin
        si <= not s1;
        t <= si and s2;
    end process;
end beh;

```

door2.vhd

```

Library IEEE;
Use IEEE.std_logic_1164.all;

entity door2 is
    port( m1 : in std_logic;

```

```

        m2 : in std_logic;
        n  : out std_logic);
end door2;
architecture beh of door2 is
begin
    process(m1,m2)
    begin
        n <= m1 and m2;
    end process;
end beh;

```

agate.vhd

```

library IEEE;
Use IEEE.std_logic_1164.all;

entity agate is
    port ( a : in std_logic;
          b : in std_logic;
          c : out std_logic);
end agate;

architecture behavioral of agate is
begin
    process(a,b)
    begin
        c<=a and b;
    end process;
end behavioral;

```

BIOGRAPHICAL SKETCH

Deepya Reddy Nalubolu was born in Nellore, Andhra Pradesh, India on May 19, 1984, the daughter of VenuGopal Reddy Nalubolu and Sowjanya Nalubolu. She completed her schooling from Space Central School, Sriharikota, India in 2001 and joined PBR Visvodaya Institute of Technology & Science affiliated to Jawaharlal Nehru Technological University (JNTU) for her Bachelor's in Technology (BTech). She graduated with a BTech degree in Electronics and Communications Engineering in April 2005 and later decided to work as a lecturer teaching electronic concepts from December 2006 till December 2007. She quit her job to come to USA for pursuing her Master's degree at UTPA, Edinburg, Texas in January 2007. During her graduate program at the Department of Electrical Engineering, she worked as a Teaching Assistant and as a Research Assistant. She is expected to graduate in August 2009.

Permanent Address: 1609 W Schunior st,

Apt #1704,

Edinburg, TX – 78541.

This thesis was typed by the Author.