A Boolean Cube to VHDL converter and its application to parallel CRC generation

Majid Hantoosh

Master’s Thesis in Electronic Design

School of Information and Communication Technology
Royal Institute of Technology (KTH)

Supervisor: Prof. Elena Dubrova
Co-Supervisor: Shohreh Sharif Mansouri
Examiner: Prof. Elena Dubrova

September 2011
Abstract

The primary outcome of this thesis is found in three contributions. First, we developed an automatic converter from the cube representation of incompletely specified multiple-output Boolean function, given in Espresso format, to VHDL. The converter is designed specifically for updating functions of Feedback Shift Register (FSRs) in the Galois configuration, namely it reads in the description of a combinatorial function and adds register stages to the appropriate positions. The converter can handle both, Linear and Non-Linear feedback Shift Registers.

The second contribution is modifying the automatic converter to design a tool which gives the user the opportunity to see the hardware characteristics of its circuit quickly from the espresso format of its design.

The third contribution is applying the resulting converter to evaluate the results of the CRC generation algorithm presented in [6]. We computed the hardware characteristics such as area, timing and power dissipation on most popular CRCs in ASIC and FPGA technologies.

Furthermore, we introduced a simple interface used to provide the user a good estimation of the power diagram during the executing time which is similar to probing the current of the circuit.
Acknowledgements

I would like to thank my supervisor professor Elena Dubrova with deepest gratitude for the great opportunity that she gave me to do my thesis at the KTH. I also would like to thank my co-supervisor Shoherh Sharif for here illuminating instructions and the consistent supporting.

I am greatly indebted to the all teaching stuff at the Med Sweden University who have instructed and helped me in the past two years.

Special thanks to my family for their loving considerations and great confidence on me all through these years.
# Table of Contents

Abstract ................................................................................................................................. 2

Acknowledgements ............................................................................................................... 3

List of Figures ...................................................................................................................... 6

List of Tables ........................................................................................................................ 7

List of Abbreviations .......................................................................................................... 7

Introduction .......................................................................................................................... 8

Chapter 1 ............................................................................................................................. 10

Preliminaries ....................................................................................................................... 10
  1.1 Boolean Functions ........................................................................................................ 10
    1.1.1 Truth Tables and Maps .......................................................................................... 11
    1.1.2 Cubes .................................................................................................................. 12
    1.1.3 Espresso format .................................................................................................... 12
    1.2 LFSR and NLFSR ..................................................................................................... 12
  1.3 Cyclic Redundancy Check (CRC) ................................................................................ 15
    1.3.1 CRC in Serial Implementation ............................................................................. 15
    1.3.2 CRC in Parallel Implementation .......................................................................... 17
    1.3.3 VHDL Code for Parallel CRC Implementation ..................................................... 21

Chapter 2 ........................................................................................................................... 23

The Boolean Cube to VHDL Converter ............................................................................... 23
  2.1 Introduction .................................................................................................................. 23
  2.2 Our automatic converting ............................................................................................ 25
  2.3 Result ........................................................................................................................... 28

Chapter 3 ........................................................................................................................... 29

ASIC Design ........................................................................................................................ 29
  3.1 Introduction to Synopsys Design Compiler .................................................................. 29
    3.1.1 Startup file .......................................................................................................... 30
    3.2 FSRs Hardware Characteristics ............................................................................... 30
    3.3 CRCs Hardware Characteristics .............................................................................. 33
    3.4 Result ....................................................................................................................... 35

Chapter 4 ............................................................................................................................ 38

Spice Model of LFSR Power Diagram ............................................................................... 38
  4.1 Calculation of Power per Cycle ................................................................................. 39
  4.2 Calculation of Switching Activity ............................................................................... 42
  4.3 Correlation ................................................................................................................... 45

Chapter 5 ............................................................................................................................ 48

CRC hardware characteristic in FPGA ............................................................................ 48
  5.1 Background ................................................................................................................ 48
  5.2 The hardware characteristics in Altera ................................................................. 49
5.3 The hardware characteristics in Xilinx .............................................................................. 50
   5.3.1 Area and Timing calculation ...................................................................................... 50
   5.3.2 Power calculation ..................................................................................................... 51
5.4 Results ............................................................................................................................... 52
   5.4.2 Result of Altera tool ................................................................................................. 52
   5.4.1 Result of Xilinx tool ................................................................................................. 53
Chapter 6 .................................................................................................................................. 55
Conclusion and future work ..................................................................................................... 55
   6.1 Conclusion .................................................................................................................... 55
   6.2 Future Work .................................................................................................................. 56
Appendix A- VHDL code for CRC-5 bit .................................................................................. 57
Appendix B- VHDL Testbench for the CRC-5 bit ................................................................... 58
Appendix C- The Boolean Cube to VHDL Converter ............................................................ 59
Appendix D- Combining of ASIC design Commands with the Converter ....................... 62
Appendix E- A modified VHDL of the LFSR that used in CRC-12 to calculate the number of switching .................................................................................................................................. 67
Appendix F- The Script of computation the hardware characteristics in Altera Quartus II.... 73
References .................................................................................................................................. 75
# List of Figures

| Figure (1. 1) | A three-dimensional Boolean space [1] edited | 10 |
| Figure (1. 2) | Truth table and K-map of an example of multiple-output functions | 11 |
| Figure (1. 3) | reducing the number of rows for the truth table [2] edited | 12 |
| Figure (1. 4) | espresso format (PLA) | 12 |
| Figure (1. 5) | Fibonacci implementation of LFSR [5] edited | 13 |
| Figure (1. 6) | An 8-bit Fibonacci LFSR | 13 |
| Figure (1. 7) | Galois implementation of LFSR [5] edited | 14 |
| Figure (1. 8) | An 8-bit Galois LFSR | 14 |
| Figure (1. 9) | NLFSR, (a) Fibonacci configuration, (b) Galois configuration | 14 |
| Figure (1. 10) | CRC generator in Serial implementation | 16 |
| Figure (1. 11) | CRC Calculation using Binary Division with 5-bit of input d | 16 |
| Figure (1. 12) | CRC-5 registers diagram | 16 |
| Figure (1. 13) | Generating a CRC with the degree of Parallelization t [6] | 19 |
| Figure (1. 14) | Generating a CRC with the degree of Parallelization t=n=5 | 19 |
| Figure (1. 15) | CRC Binary Division with 10-bit input d | 21 |
| Figure (1. 16) | Input data and CRC time diagram | 22 |

| Figure (2. 1) | A part of Espresso format for a 17-bit NLFSR | 23 |
| Figure (2. 2) | Espresso output file named out.pla | 24 |
| Figure (2. 3) | The Converter design flow | 26 |

| Figure (3. 1) | An Overview of the Synthesis tool | 29 |
| Figure (3. 2) | The converter combined with the ASIC calculations flow | 31 |
| Figure (3. 3) | System interface and Timing | 34 |
| Figure (3. 4) | Total Dynamic Power for different CRC | 36 |
| Figure (3. 5) | Timing calculation for different CRC | 37 |
| Figure (3. 6) | Area Calculation for different CRC | 37 |

| Figure (4. 1) | Power per cycle plot for LFSR-12 | 41 |
| Figure (4. 2) | Internal, Switching and Dynamic average power for CRC-12 bit | 41 |
| Figure (4. 3) | Max Net Switching Power is (8.275uW) at the time (4120 ns) | 42 |
| Figure (4. 4) | LFSR-12 simulation shown total number of switching for one period | 43 |
| Figure (4. 5) | The LFSR-12 simulation shows the Max numbers of switching per cycle | 43 |
| Figure (4. 6) | The Containing of the VCD file at (4120-4124) ns | 44 |
| Figure (4. 7) | Correlation result in Matlab | 46 |
| Figure (4. 8) | Number of switching per cycle for the LFSR-12 | 46 |
| Figure (4. 9) | Max number of switching at the time 4120-4124 ns | 47 |
List of Tables

Table (2.1) A comparison of the time required to produce the VHDL code manually and our converter ......................................................... 28
Table (3.1) Results of NLFSRs using TSMC 90 nm technology ........................................ 35
Table (3.2) Results of different type of CRCs using TSMC 90 nm technology ................. 35
Table (3.3) ........................................................................................................ 36
Table (4.1) ........................................................................................................ 42
Table (5.1) The XPE parameters .............................................................................. 51
Table (5.2) The hardware characteristics for the CRC presented in [6] ...................... 52
Table (5.3) The relation between the No. of XORs and the results ......................... 53
Table (5.4) The results for different type of CRC .................................................. 54
Table (5.5) Total number of XORs, power, area and timing for different CRC generators design ............................................................ 54

List of Abbreviations

ASIC Application Specific Integrated Circuit
CLB Configurable Logic Blocks
CRC Cyclic Redundancy Check
DOS Disk Operating System
FPGA Field Programmable Gate Array
GUI Graphic User Interface
HDL Hardware Description Language
K-Map Karnaugh Map
LFSR Linear Feedback Shift Register
LSB Least Significant Bit
LUT Look-up tables
MSB Most Significant Bit
PLA Programmable Logic Array
SAIF Switching Activity Interchange Format
TCL Tool Command Language
VCD Value Change Dump
XPA Xpower Analyzer
XPE Xpower Estimation
Introduction

Linear and Non-Linear Feedback Shift Registers FSRs in the Galois configuration can be given in a cube representation of incompletely specified multiple-output Boolean functions in an Espresso format. This format is a Programmable Logic array (PLA) that has a sequence of zeros, ones and "don’t care ". By using Espresso tool, this format can be converted to a list of minimized functions in terms of input and output variables in Espresso format. The minimized functions are the updating functions of FSRs. Writing the VHDL cod for these updating functions, which given in Espresso format, can consume time and effort especially when the functions are related to a high order of FSR. The functions are consist of hundred or even thousands of the special characters ( ! , & , | , = ). These characters should be changed to another forms (NOT, AND, OR, <=) respectively.

In this thesis, we developed an automatic converter as a Linux script code that achieves all the above tasks and produces the VHDL code and the testbench as well. It written in C-program and works in Linux environment, it can work on both linear and non-linear FSRs. We also modified this converter so that it is not can only generate the VHDL, but it can also use the generated VHDL code to compute the hardware characteristics such as area, timing and power in ASIC design.

Using ASIC design means that the designer need to achieve some tasks of calculation to meet the goal of the design. The area, timing and power are the most important particulars or characteristics that needed to be calculated before sending the design to the factory. To calculate these characteristics, we need the appropriate tool such as Design Compiler from Synopsys [24] and a simulation tool such as ModelSim from Mentor Graphics [25]. Both tools have the ability to perform the required tasks in text commands form without need to open any Graphic User Interface GUI of that tool. These commands integrated to the converter in order to have a tool that can calculate the hardware characteristics beside the ability of generating the VHDL code.

In this thesis, we computed the hardware characteristics and evaluated for the CRC generation algorithm presented in [6].

A Spice model of power diagram was introduced in this project, which is a simple interface used to simulate the spice characteristics of the chip power. This interface can be used to compute the power diagram of the chip during the operation and calculate the average power as well.
The containing of this thesis is organized as follows. Chapter 1 discusses the related background such as Boolean functions, Espresso format, Cube, Linear and Non-Linear FSR and CRC. Chapter 2 describes the developed Boolean converter. In chapter 3, we discussed the ASIC design, where we appended the commands of calculating the hardware characteristics to the converter. In this chapter we also calculate the area, timing and power for the CRCs generators that presented in [6] and given the result at the end of this chapter. Chapter 4 describes the Spice model of power diagram, and we discussed how the power consumption is related to the switching activity. In chapter 5, we computed the hardware characteristic for the CRCs generators presented in [6] based on FPGA implementation by using Xilinx and Altera Quartus tools. Chapter 6 concludes the thesis work and suggests the future work.
Chapter 1

Preliminaries

This chapter undertakes a review of some basic concepts that needed to be known as a background. This background is important to be understood before starting the main work of this thesis. The chapter has the following sections: Boolean functions, Truth table, K-Maps, Boolean Cube, Espresso format, LFSR & NLFSR and CRC.

1.1 Boolean Functions

A Boolean function is a function that consist of one or more inputs and one or more outputs of “0” or “1” or “-” (don’t care). A completely specified Boolean function is a mapping between Boolean spaces [1]. A mapping \( f : B^n \to B^m \) has an n-input and m-output function, where \( B = \{0,1\} \), \( B^n \) is the multi-dimensional space spanned by n binary-valued Boolean variables, i.e. \( B^n \) is the Cartesian product \( B \times B \times \ldots \times B \) [21]. It is also referred to as the n-dimensional cube. Figure (1.1) shows the three-dimensional Boolean space with the vector notation. This vector is used because the Boolean function can be seen as an array of m scalar functions over the same domain.

![Figure (1.1) A three-dimensional Boolean space [1] edited](image)

- \( x_1, x_2, x_3 \) are Variables
- \( x_1, x'_1, x_2, x'_2, x_3, x'_3 \) are literals

Boolean function can be represented by a formula defined as catenations [2] of: Parentheses ( ), variables \( x, y, z \), binary operations “+” (OR) and “.” (AND), unary operation negation “-”.

E.g. \( f(x_1, x_2, x_3) = (x_1 + x_2)' \cdot x_3 \)

If we have a function \( (x_1, x_2, x_3) = x'_1, x'_2, x_3 + x'_1 \cdot x_2 \cdot x_3 + x_1 \cdot x'_2 \cdot x_3 \), then we mark the corresponding corners (001, 011 and 101) in the figure (1.1).

When we have more than one output (m > 1), \( f : B^n \to \{0, 1, -\}^m \).

Where (-) denotes a don’t care condition.

0 = off set.

1 = on set.

- = dc set.

Boolean Functions can be represented in two fundamental approaches [2]. The first approach is by keeping representation canonical with respect to the function e.g. Truth tables, Karnaugh maps and Binary Decision Diagrams (BDDs). The second approach is by keep representation non-canonical e.g. Boolean expressions and logic circuits [2].
A Boolean function $f: \{0,1\}^n \rightarrow \{0,1\}$ has a canonical sum of product (SOP) form of following type [2]:

$$f(x_1, x_2, \ldots, x_n) = \sum_{i=0}^{2^n-1} \left[ \prod_{j=1}^{n} C_i \cdot x_1^{i_1} \cdot x_2^{i_2} \cdot \ldots \cdot x_n^{i_n} \right]$$

Where:

$C_i \in \{0,1\}$ is a constant,

$(i_1, i_2, \ldots, i_n)$ is the binary expansion of $i$,

$x_k^{i_k} = x$ if $i_k = 0$ and $x_k^{i_k} = \bar{x}$ if $i_k = 1$

If $f$ is a function from $\{0,1\}^n$ into $\{0,1, -\}$, assuming that $f^1$ is the care set or of the Boolean function and $f^-$ is the don’t care set is the set on which the Boolean function $f$ is not defined, $f^{1-}$ is the union of the care set and the don’t care set. The Boolean function is completely specified or total if the don’t care set is empty, else is incompletely specified [3].

If $f: B^n \rightarrow B \cup \{-\}$, then the Boolean function $f$ is incompletely specified Boolean function [21]. The Boolean function $f$ is called Multiple-output Boolean functions if:

$$f: B^n \rightarrow B^m, f: B^n \rightarrow (B \cup \{-\})^k$$

### 1.1.1 Truth Tables and Maps

Truth table is the simplest tabular form used to represent the Boolean functions, it consist of two parts according to the inputs and outputs of the function. Truth table is a table that lists all possible combinations of input values and indicating for each combination in the output values. In other words, the truth table lists all points in the Boolean input space with the corresponding values of the outputs. If we have a Boolean function with an $n$-variable, then the truth table will contain $2^n$ rows that give the value of the function for the corresponding values of the variables $x_1, \ldots, x_n$.

The truth table can be re-arranged in a rectangular n-dimensional Karnaugh map (K-map) as illustrated in the example shown in figure (1.2).

![Truth Table and Karnaugh Map](image)

**Figure (1.2)** Truth table and K-map of an example of multiple-output functions

The example in figure (1.2) is for a function of $f: \{0,1\}^3 \rightarrow \{0,1\}^2$. The function $f$ has more than one output function ($f_1$ and $f_2$), therefore it is called multiple-output functions. The truth table and K-map can be used only for very small function since both give a complete list of $2^n$ points in $B^n$. By using don’t care symbol (-) and removing the input combinations $(a_1, \ldots, a_n) \in B^n$, for which $f(a_1, \ldots, a_n) = 0$, then we can reduce the number of rows in the truth table to be as shown in figure (1.3). The new truth table is called Cube table which we will discuss in the next section.
1.1.2 Cubes

A Boolean function with n-variable can be seen as a set of point in an n-dimensional Boolean space. Any k-dimensional subspace is called Cube, where 0 ≤ k ≤ n [2], it is n-tuple containing “0”, “1” and “-”, while minterm is an n-tuple containing “0” and “1” only. A cubes with n=3 can contains 0 0 0, 1 - 1, 1 0 - , - - , etc. A Cube can be defined as the conjunction of literals or AND of a set of literal functions, or it can be seen as a product-term, e.g. the product term of the cube -10 is \( x_2 x'_3 \).

Boolean function can be represented by a sum of cubes or sum of products expression (SOP). E.g. if a set of cube such as \{1 - 0, - 0 -\} represented as on-set of f, where on-set is subset of \( B^n \) containing all minterms mapped to “1”, the sum of product expression for f will be \( x_1 x'_3 + x'_2 \).

1.1.3 Espresso format

Espresso format or PLA (Programmable Logic Array) is the format that used by Espresso tool. It contains the cube table arranged in a special way as shown in figure (1.4). This format is used by Espresso tool for minimization to generate the Boolean functions. Figure (1.4) shows an example of espresso format that has three inputs and one output, the espresso format can contain more than one outputs.

1.2 LFSR and NLFSR

Linear Feedback Shift Register LFSR is a string of bits which stored in a string of memory cells (Flip-Flop), and a clock pulse can advance the bits one space in that string [4]. LFSR shifts its contents into adjacent positions within the register (in case of the position on the end,
the shifting will be to the out of the register). It is called linear because the input bit to the register is a linear function of its previous state, or the input bit is a linear function of its internal states. The LFSR can be implemented with only XOR and XNOR combinations of the internal stages for the shift registers, both gates has a linear function.

In general, LFSR should be initialized with a value called seed. The shift register can have a limited number of state or period; however, the LFSR can have a very long period depending on the LFSR’s order and the feedback configurations.

In this thesis, we will use the same definition used in [6] for FSR architecture and we numbered the FSRs bits from left to right as n1, n-2, …, 0. The last bit of FSR is bit 0 and the first bit is n-1.

LFSR can be implemented in two approaches, Fibonacci implementation and Galois implementation.

**Fibonacci LFSR:**

In this type, the input for the shift register is a feedback of modulo-2 sum of the binary-weighted taps, where modulo-2 sum is performed using an exclusive-OR (XOR) [5].

Figure (1.5) shows a Fibonacci LFSR. The input side of the LFSR is the register \( x_{n-1} \) and the output is the register \( x_0 \). When weight of \( f \) is “1”, a feedback function will be considered, when the weight is (0) then there will be no feedback (no connection). Only two exceptions that \( f_0 \) and \( f_n \) should be always (1) and thus always connected directly.

In general, feedback functions are usually represented in Reed-Muller canonical form which can be described in two extensions: Fixed polarity where variables are allowed to appear in rather complemented or un-complemented form but not both according to the polarity vector [22]. Mixed polarity, variables are allowed to appear in both complemented and un-complemented form [22].

Figure (1.6) illustrates a simple 8-bit LFSR type Fibonacci. In this LFSR we have 4 taps on 0, 4, 5 and 6.

The feedback polynomial for this LFSR can be formulated as follow:

\[
f(x) = x^8 \oplus x^6 \oplus x^5 \oplus x^4 \oplus x^0 = x^8 \oplus x^6 \oplus x^5 \oplus x^4 \oplus 1
\]

As a set of tap: \([n, n-2, n-3, n-4, 0] = [8, 6, 5, 4, 0]\) where \( n = 8 \).

This feedback polynomial is for a maximal 8-bit LFSR. The number of state that we can get before the shift register repeated itself is \( 2^n - 1 = 2^8 - 1 = 255 \). In binary system, when
we have 8 bit, it means we have \(2^8\) different state starting from \((00000000)\) ending with \((11111111)\). However, the first case when all bits are zeros, the shift register will never change unless XNORs are used in the feedback function. Therefore, this state will be removed from the total state and \((-1)\) will added to the expression \(2^n\) to be \(2^n - 1\). Each power of the term in the equation represents to the corresponding tapped bit except \(x^8\) which is corresponds to the input to the first bit.

**Galois LFSR:**

It consists of a shift register where the content of which is modified at every step by a binary-weighted value of the output stage using modulo-2 operation [5].

![Galois implementation of LFSR](image)

Figure (1.7) shows a Galois LFSR, it is noted that the order of the Galois weights is opposite that of the Fibonacci weight in figure (1.5).

Both Fibonacci and Galois LFSR can produce the same sequence if they have identical feedback weights. However, to have identical sequences phase, the initial states of the two types must be different since zero bits offset relative to each other.

Figure (1.8) shows an example of 8-bit Galois LFSR, where \((\oplus)\) is represented to the XOR gate. The feedback polynomial for this LFSR can be formulated as follow:

\[
f = x^8 \oplus x^3 \oplus x^2 \oplus 1
\]

As a set of tap: \([n, n-5, n-6, 0] = [8, 3, 2, 0]\) where \(n = 8\).

![An 8-bit Galois LFSR](image)

Figure (1.8) An 8-bit Galois LFSR

Non-Linear Feedback Shift Register (NLFSR) is also FSR but the input bit is non linear function of its internal states. Figure (1.9) shows a NLFSR in Fibonacci and Galois configurations. NLFSR can be implemented with a simple and fast hardware architecture using XORs, XNORs and AND gates.

![NLFSR](image)

Figure (1.9) NLFSR, (a) Fibonacci configuration, (b) Galois configuration

Since \((\text{AND})\) gate is a non linear function, it caused the FSR to be a non linear FSR. Linear FSR is widely used in Cyclic Redundancy Check (CRC) which we will discuss it in the next section.
1.3 Cyclic Redundancy Check (CRC)

Cyclic Redundancy Check (CRC) is a technique used to detect the errors; CRC is the most popular among several error detection algorithms. It is widely used in digital data communication and other fields such as data compression, data storage etc. Different type of CRC algorithms found, every type has a predetermined generator polynomial $G(x)$ that is used to generate the CRC code.

The following are the common generator polynomials for CRC:

- CRC-8: $x^8 \oplus x^7 \oplus x^6 \oplus x^5 \oplus x^4 \oplus x^3 \oplus x^2 \oplus x^1 \oplus 1$
- CRC-10: $x^{10} \oplus x^9 \oplus x^5 \oplus x^4 \oplus x^2 \oplus 1$
- CRC-12: $x^{12} \oplus x^{11} \oplus x^3 \oplus x^2 \oplus x \oplus 1$
- CRC-CCITT: $x^{16} \oplus x^{12} \oplus x^5 \oplus 1$
- CRC-16: $x^{16} \oplus x^{15} \oplus x^{2} \oplus 1$
- CRC-32: $x^{32} \oplus x^{26} \oplus x^{23} \oplus x^{22} \oplus x^{16} \oplus x^{12} \oplus x^{11} \oplus x^{10} \oplus x^{8} \oplus x^{7} \oplus x^{4} \oplus x^{2} \oplus x \oplus 1$
- CRC-64-ISO: $x^{64} \oplus x^{4} \oplus x^{3} \oplus x \oplus 1$

CRC technique can be performed either in hardware or software. In software, the throughput can be very limited compared with hardware technique [7]. In hardware technique, the calculation of CRC can be easily performed by logical combinations of shift registers and XOR gates. The common method that use in serial (and parallel as well) CRC is the Linear Feedback Shift Register (LFSR). Every generator polynomials has highest order which is called as the degree of the generator polynomial, it is also indicate to the length of the CRC code.

For example, the highest order in the CRC-12 above is 12 which represents to the degree of the generator.

If we rewrite the CRC-12 above in term of the generator polynomial $G(x)$:

$G(x) = x^{12} \oplus x^{11} \oplus x^{3} \oplus x^{2} \oplus x \oplus 1$

The coefficient of $G(x)$ is:

$c_i = \{c_{12}, c_{11}, c_{10}, \ldots, c_1, c_0\} = \{110000000111\}$.

Where $i \in \{0,1, \ldots, n - 1\}$, $n$ is the degree of the generator, the number of coefficients $c$ will be $n + 1 = 13$. During the transmission, CRC code is appended to the end of the data message. If the data message is considered by the letter $d$ and the CRC code is denoted by $C$ with length $n$ (the generator polynomial degree). Then, overall transmitted data unit with CRC code can be written as below:

$T = \{dc\} = d \times 2^n + C$.

The transmitted data $T$ is exactly a multiple of generator $G(x)$. When there is no error occurs during the transmission of $T$, the received message must also be an exact multiple of the same generator $G(x)$. If the received message not a multiple of $G(x)$, an error must have occurred during the transmission.

1.3.1 CRC in Serial Implementation

In this implementation the data message is shifted in one bit per clock cycle, therefore we need one cycle per bit to process the data message. In Serial implementation, data processing
can be achieved in somewhat high clock rate but it suffers from the low data throughput for its serial input. Figure (1.10) illustrates the basic architecture for the serial Implementation of CRC using LFSR in Galois configuration.

\[ C_t = \{c_n, c_{n-1}, c_{n-2}, \ldots, c_2, c_1, c_0\} \]

The data message \( d \) entered from the left hand side. If \( c_i = 1 \) an XOR operation found in between the two registers. If \( c_i = 0 \) no XOR, only shift operation in between the two registers. If we consider a generator polynomial: \( G(x) = x^5 \oplus x^2 \oplus 1 \), and a data massage: \( d = \{11001\} \).

From \( G(x) \), the degree of the generator polynomial \( n = 5 \).

We can calculate the CRC for this data massage by using the polynomial long division as shown in figure (1.11).

\[
\text{Data Massage: } d = (11001) 
\]

\[
\text{Transmitted Data: } 1100101100 
\]

\[
\text{Figure (1.10) CRC generator in Serial implementation} 
\]

Since our design considers the input register is \( x_4 \) and the output register is \( x_0 \), we start with the LSB in the long division steps and the generated CRC is arranged in a reversed sequence as shown in figure (1.11). The length of the data message \( d \) is five bit and we appended five zeros, therefore we need 10 clock cycles to produce the CRC. Figure (1.12) shows the registers diagram for the example above.

\[
C_t = \{100101\} 
\]

\[
\text{Figure (1.12) CRC-5 registers diagram} 
\]

The next state of each register can be written as following:

\[
\begin{align*}
x'_4 &= (c_n \cdot x_0) \oplus d \\
x'_3 &= (c_{n-1} \cdot x_0) \oplus x_4 \\
x'_2 &= (c_{n-2} \cdot x_0) \oplus x_3 \\
x'_1 &= (c_{n-3} \cdot x_0) \oplus x_2 \\
x'_0 &= (c_{n-4} \cdot x_0) \oplus x_1
\end{align*} 
\]

\[
\begin{align*}
x'_4 &= x_0 \oplus d \\
x'_3 &= x_4 \\
x'_2 &= x_3 \\
x'_1 &= x_0 \oplus x_2 \\
x'_0 &= x_1
\end{align*} 
\]
Where: \(x\) is the current state, \(x'\) is the next state, \(d\) is the serial input of the data unit.

In general:
\[
\begin{align*}
x'_{n-1} &= (c_n \cdot x_0) \oplus d \\
x'_{n-2} &= (c_{n-1} \cdot x_0) \oplus x_{n-1} \\
x'_{n-3} &= (c_{n-2} \cdot x_0) \oplus x_{n-2} \\
&\quad \ddots \\
x'_1 &= (c_2 \cdot x_0) \oplus x_2 \\
x'_0 &= (c_1 \cdot x_0) \oplus x_1
\end{align*}
\]

Or we can write the above equation as follow:
\[
\begin{align*}
x'_{n-1} &= (c_n \cdot x_0) \oplus d \quad \text{----------- (1.1)} \\
x'_i &= (c_{i+1} \cdot x_0) \oplus x_{i+1} \quad ; \quad i = \{n-2, \ldots, 1, 0\} \quad \text{----------- (1.2)}
\end{align*}
\]

Equation (1.1) represents to the next state for the input register \((x'_{4})\), and equation (1.2) represents to the next states for the other registers \((x'_{3} \text{ to } x'_{0})\).

From equation (1.2), max value for \((i)\) is \(n-2\).

If \(c_{i+1} = 1\), we have \(x'_i = x_0 \oplus x_{i+1}\), which means there is an XOR operation between register \(x_i\) and register \(x_{i+1}\).

If \(c_{i+1} = 0\), we have \(x'_i = x_{i+1}\), which means there is only shift operation is performed and no XOR operation between these two registers.

Figure (1.11) illustrates the steps of calculating the CRC code. Since we have a data of 5 bit and we appended four zeros, then we need 10 clock cycle to get the CRC code. If we assume that the CRC code is \(k\)-bit long and there are \(t\)-bits in the data message, since we have here serial implementation of CRC where the data message is shifted in one bit per clock cycle. Therefore, the total number of cycles that required to performing the calculation of a CRC code will be \((k + t)\). It is clear that the data is processed one bit per clock cycle in total of \((k + t)\) cycles are needed to obtain a CRC code for a message with \(t\) bits.

1.3.2 CRC in Parallel Implementation

As we seen in serial implementation, we need one cycle per bit to process the data message. In parallel CRC implementation, data message processed in a way so that multiple input bits processed in parallel every clock cycle. Since parallel CRC used LFSR, then we can get the next state \(s_{t+1}\) from the current state \(s_t\) by a linear transformation [6]:
\[
s_{t+1} = s_t \cdot M
\]

Where:
\(s_{t+1}\) is the LFSR next state, \(s_t\) is the LFSR current state, and \(M\) is an \(n \times n\) connection matrix of the LFSR [6]. In the previous section we have used LFSR in Galois configuration; the connection matrix will be as follow:
Where \(c_n \ldots c_1\) are the coefficients of the generator polynomial of the LFSR \((c_0\) is ignored). We can write the feedback function \(f_{n-1-i}(x)\) of the LFSR which are represented in the columns of the Matrix \(M\) as follow [6]:

\[
f_{n-1-i}(x) = m_{0,i}x_{n-1} \oplus m_{1,i}x_{n-2} \oplus \ldots \oplus m_{n-1,i}x_0
\]

Where \(i \in \{0, 1, \ldots, n-1\}\), and \(m_{i,j}\) is the element of \(M\).

We can compute the state reachable \(s_t\) from any initial state \(s_0\) in \(t\) steps by taking the \(i\)th power of \(M\):

\[
s_t = s_0 \cdot M^t
\]

Thus, we do not need to go through the intermediate states when we want to compute \(s_t\).

By return to our example in the previous section 1.3.1, we can write the connection matrix \(M\) as follow:

\[
M = \begin{pmatrix}
0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 \\
1 & 0 & 0 & 0 & 1 \\
c_n & c_{n-1} & c_{n-2} & c_{n-3} & \ldots & c_1
\end{pmatrix}
\]

The feedback functions are as follows:

\[
\begin{align*}
f_4(x) &= x_0 \\
f_3(x) &= x_4 \\
f_2(x) &= x_3 \\
f_1(x) &= x_0 \oplus x_2 \\
f_0(x) &= x_1
\end{align*}
\]

The feedback functions above are identical to the functions of the next state that we got in the serial implementation of CRC in the previous section except we do not have data message \((d)\) here.

Since our generator polynomial is type 5th order, we can generate 5-bit of the output sequence per clock cycle by computing 5th power of \(M\) as follow:

\[
M^2 = M \times M = \begin{pmatrix}
0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & 1 \\
\end{pmatrix}
\]

\[
M^3 = M^2 \times M = \begin{pmatrix}
0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & 1 \\
1 & 0 & 1 & 0 & 1 \\
\end{pmatrix}
\]

18
Now we can write the feedback equations of the LFSR with the degree of parallelization 5

By using the matrix \( M^5 \) to be as the following:

\[
M^5 = M^3 \times M^2 = \begin{bmatrix}
1 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & 1 \\
1 & 0 & 1 & 1 & 0 \\
0 & 1 & 0 & 1 & 1 \\
1 & 0 & 1 & 1 & 1
\end{bmatrix}
\]

As a CRC, we can write the following equation [6],[20]:

\[ s_t = s_0 \cdot M^t \oplus D \]

Where \( D = (d_{t-1}, \ldots, d_4, d_0, 0, \ldots, 0) \) is an n-bit vector whose first \( t \) elements are the data [6], i.e. the data is entered in blocks of \( t \) bits into the first \( t \) bits of the LFSR as shown in figure (1.13).

In our example above, \( t = n = 5 \) therefore a block of 5-bit of the data (d) will be added to the LFSR as following:

\[
\begin{align*}
f_4(x) &= x_0 \oplus x_2 \oplus x_4 \oplus d_4 \\
f_3(x) &= x_1 \oplus x_3 \\
f_2(x) &= x_0 \oplus x_2 \oplus d_2 \\
f_1(x) &= x_0 \oplus x_1 \oplus x_2 \oplus x_4 \oplus d_1 \\
f_0(x) &= x_0 \oplus x_1 \oplus x_3 \oplus d_0
\end{align*}
\]

The registers diagram for the CRC generator of our example will be as shown in figure (1.14).
To calculate the CRC in our example using the same data input \( d = \{11001\} \), which we used in the serial implementation in the previous section, we will use the previous equations in (1.3) in two steps:

- At the first step, we need to feed the data message to the LFSR registers in the first clock cycle. The LFSR will have the following values:
  \[
  \begin{align*}
  x_4 &= 1 \\
  x_3 &= 1 \\
  x_2 &= 0 \\
  x_1 &= 0 \\
  x_0 &= 1 \\
  \end{align*}
  \]
  \( \begin{array}{c}
  11001 \ (1^{st} \text{ clock})
  \end{array} \)

- Since we have 5th order of generator polynomial, we inter five zeros \( d_4 = 0, d_3 = 0, d_2 = 0, d_1 = 0, d_0 = 0 \) as a data message at the second clock cycle; the CRC will be as following:
  \[
  \begin{align*}
  f_4 &= 1 \oplus 0 \oplus 1 \oplus 0 = 0 \\
  f_3 &= 0 \oplus 1 \oplus 0 = 1 \\
  f_2 &= 1 \oplus 0 \oplus 0 = 1 \\
  f_1 &= 1 \oplus 0 \oplus 0 \oplus 1 \oplus 0 = 0 \\
  f_0 &= 1 \oplus 0 \oplus 1 \oplus 0 = 0 \\
  \end{align*}
  \]
  \( \text{CRC} = 01100 \ (2^{nd} \text{ clock}) \)

This is the same result that we got in the serial implementation of the CRC. Simply, the CRC is generated in only two clock cycle compared with 10 clock cycle in the serial implementation. Since \( (a) \oplus 0 = (a) \), we do not need to XOR the feedback equations with zero, this truth can speed up the computation processing as we will used in the next chapters of this thesis.

By using this equation, the next state of the registers \( x' \) can be calculated by the current states \( x \) recursively using a predetermined matrix \( M^f \) and the parallel input vector \( d \).

Equation (1.3) shows that we can feed the data message in a block of 5-bit, as an example, we consider now the length of the data message is 10-bit such as \( d = 1100100111 \), then we need three clock cycle to computing the CRC as following:

- Feeding the first 5-bit block to the LFSR:
  \[
  \begin{align*}
  x_4 &= 1 \\
  x_3 &= 1 \\
  x_2 &= 0 \\
  x_1 &= 0 \\
  x_0 &= 1 \\
  \end{align*}
  \]
  \( \begin{array}{c}
  11001 \ (1^{st} \text{ clock})
  \end{array} \)

- Feeding the second 5-bit block to the LFSR by using the equation (1.3):
  \[
  \begin{align*}
  d_4 &= 0, d_3 = 0, d_2 = 1, d_1 = 1, d_0 = 1 \\
  f_4 &= 1 \oplus 0 \oplus 1 \oplus 0 = 0 \\
  f_3 &= 0 \oplus 1 \oplus 0 = 1 \\
  f_2 &= 1 \oplus 0 \oplus 1 = 0 \\
  f_1 &= 1 \oplus 0 \oplus 0 \oplus 1 \oplus 1 = 1 \\
  f_0 &= 1 \oplus 0 \oplus 1 \oplus 1 = 1 \\
  \end{align*}
  \]
  \( = 01011 \ (2^{nd} \text{ clock}) \)

- As we mentioned above, we do not need to XOR with zeros as it will result the same values, so we calculate the CRC as in the following with considering the new values for the previous functions \( f_4 . . . f_0 \):
\[ f_4 = 1 \oplus 0 \oplus 0 = 1 \]
\[ f_3 = 1 \oplus 1 = 0 \]
\[ f_2 = 1 \oplus 0 = 1 \]
\[ f_1 = 1 \oplus 1 \oplus 0 \oplus 0 = 0 \]
\[ f_0 = 1 \oplus 1 \oplus 1 = 1 \]

This is also the same result that we calculated by using the long division as shown in figure (1.15).

In the above division, we reversed each block of 5-bit because we used \( x_4 \) as input register and \( x_0 \) as output register. From the above explaining, it is clear that we need three clock cycle to compute the CRC for a data message of 10-bit compared with 15 clock cycle in the serial implementation of CRC.

**Figure (1. 15) CRC Binary Division with 10-bit input \( d \).**

### 1.3.3 VHDL Code for Parallel CRC Implementation

CRC can be generated in hardware technique by using the updating functions to write the VHDL code. As we explained in section 1.3.2, two clock cycle needed to generate the CRC in case of the length of the data message that processed in parallel is equal to the length of the generator polynomial. Figure (1.16) shows the time diagram of generating the CRC, where every CRC generated needs two clock cycles.
By return to the previous example we have the following generator polynomial:
\[ G(x) = x^5 \oplus x^2 \oplus 1 \]

We have already computed the updating function to be as shown below:
\[
\begin{align*}
    f_4 &= x_0 \oplus x_2 \oplus x_4 \oplus d_4 \\
    f_3 &= x_1 \oplus x_3 \oplus d_3 \\
    f_2 &= x_0 \oplus x_2 \oplus d_2 \\
    f_1 &= x_0 \oplus x_1 \oplus x_2 \oplus x_4 \oplus d_1 \\
    f_0 &= x_0 \oplus x_1 \oplus x_3 \oplus d_0 
\end{align*}
\]

The completed VHDL code for these functions is shown in Appendix (A). As we explained in section 1.3.2, we do not need to add the data message \((d_4, \ldots, d_0)\) to these functions because it has zero value.

Appendix (B) shows the testbench for the VHDL code. The testbench can generate five bits of semi random input data message with a clock cycle of 4 ns.
Chapter 2

The Boolean Cube to VHDL Converter

The main goal of this thesis is designing an interface which gives the user the opportunity to see the hardware characteristics of its circuit quickly from the espresso format of its design. The tool goes automatically through all the steps of synthesis and characterization of the circuit.

We focus on the groups of users which do not have any knowledge about hardware design of a chip. From user point of view, he/she only need to define its design in espresso format. The circuits we supports in the first version of our tool are any combinational block containing AND, XOR, OR gates.

In this project, we focus on the FSRs; therefore the converter is designed for that purpose only.

The interface developed is an automatic converter tool written for Linux environment that used to generating a VHDL code from an Espresso format. The converter is a executable text file that has all the required Linux commands and C-code to produce the VHDL code.

The Espresso format used by this convertor describes the cube representation of incompletely specified multiple-Boolean function. The VHDL code generated by this convertor is designed specifically for updating functions of a Linear or Non-Linear Feedback Registers that arranged in Galois implementation. The converter generates the VHDL code and the testbench as well.

2.1 Introduction

Linear and Non-Linear FSRs can be given in an espresso format that has the cube representation of incompletely specified multiple-Boolean function, i.e. as a sequence of binary numbers of “0” and “1” and don’t care “-” in a PLA format as shown in the example illustrated in figure (2.1).

```
.l 17
.p 17
.lib x(16) x(15) x(14) x(13) x(12) x(11) x(10) x(9) x(8) x(7) x(6) x(5) x(4) x(3) x(2) x(1) x(0)
.ob f(16) f(15) f(14) f(13) f(12) f(11) f(10) f(9) f(8) f(7) f(6) f(5) f(4) f(3) f(2) f(1) f(0)
1------010-101- 00000010000010010
01------000-0-1- 00100110100000000
-1--0-1-1-0-----10- 000000000010100
-10-10--1-0--01-- 100100000010000
-0--0-0-1-01--1- 010000000001000
001-0--0------0-0- 010010001000000
.
.
.
.
@Figure (2.1) A part of Espresso format for a 17-bit NLFSR
```

Figure (2.1) illustrate a part of Cube representation for a 17-bit NLFSR in Espresso format. This NLFSR consists of 17-input $(x_{16} \ldots x_0)$ and 17-output $(f_{16} \ldots f_0)$.

Espresso tool is a two level logic minimize developed by University of California, Berkeley. This tool can be downloaded from [9]. With this tool we can convert the espresso format that
given in Cube representation to a number of functions according to the number of output as shown in the following Espresso command:

\[
\texttt{espresso -o eqntott input.pla > out.pla}
\]

When executing the above command, Espresso tool will manage the input file `input.pla` and creates the minimized results as a number of equations saved in separate file named `out.pla` as shown in figure (2.2). These equations represent to the updating functions of the NLFSR.

\[
f(16) = (x(15) &!x(14) &x(12) &!x(11) &x(8) &!x(6) &!x(3) &!x(2)) |
(x(15) &!x(10) &!x(9) &!x(8) &x(7) &x(5) &x(2) &x(1)) |
(x(14) &!x(13) &!x(10) &!x(8) &!x(6) &!x(5) &!x(3) &!x(1)) |
(!x(15) &!x(12) &!x(11) &!x(10) &!x(9) &!x(6) ........
\]

- ...
- ...

\[
f(0) = (x(12) &!x(11) &x(10) &!x(9) &!x(7) &!x(6) &!x(4) &!x(1)) |
(!x(13) &!x(11) &!x(10) &!x(8) &!x(7) &!x(4) &!x(2)) |
(x(15) &!x(14) &!x(10) &!x(8) &!x(5) &!x(3) &!x(2));
\]

Figure (2.2) Espresso output file named out.pla

The Espresso output file `output.pla` will has all the updating functions \(f_{16}, \ldots, f_0\), figure (2.2) shows the function of \(f_0\) and a part of the function \(f_{16}\) only.

**Converting from Espresso format to VHDL**

The VHDL code can be written for the functions shown in figure (2.2). Since the VHDL compilers cannot accept a character such as \&, !, |, Therefore these characters must be changed to AND, NOT, OR respectively. For the above example of NLFSR-17 bit, the total numbers of the characters found in the file `output.pla` are 29830, 16667, 4848 respectively which they need a lot of time to write the corresponding VHDL form. The numbers of these characters increase by increasing the order of LFSR or NLFSR. In NLFSR, the registers should be connected by the updating functions as shown below:

\[
\begin{align*}
X(16) &\leftarrow (x(15) \text{ and } \neg x(14) \text{ and } x(12) \text{ and } \neg x(11)) \ldots . \\
X(15) &\leftarrow (\neg x(15) \text{ and } x(11) \text{ and } \neg x(9)) \ldots . \\
\vdots & \hspace{5cm} \\
X(1) &\leftarrow (x(16) \text{ and } x(10) \text{ and } \neg x(7) \text{ and } x(6)) \ldots . \\
X(0) &\leftarrow (x(12) \text{ and } x(11) \text{ and } x(10) \text{ and } x(9)) \ldots . 
\end{align*}
\]

The other parts of the VHDL code should be written to be as shown below:

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity nlfsr is
port( clk : in std_logic;
      reset : in std_logic;
      output : out std_logic );
end entity nlfsr;

architecture aa of nlfsr is
begin
  signal x : std_logic_vector (16 downto 0);
  process (clk,reset)
  begin
    if (reset='1') then
      x <= "111111111111111111";
    elsif (clk'event and clk='1') then
      x(16) <= (x(15) and not x(14) and x(12) and not x(11) x(1)) ;
      x(15) <= (not x(15) and not x(11) and not x(9) and x(7));
    end if;
  end process;
end;
```

24
The VHDL code above doesn’t show the completed set of functions, only parts of some updating functions.

The test bench for the above code can be as follow:

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity nlfsr_tb is
end nlfsr_tb;
architecture aa of nlfsr_tb is
  component nlfsr
    port
      ( clk : in  std_logic;
      reset : in  std_logic;
      output : out std_logic    );
  end component;
  signal clk : std_logic := '1';
  signal reset : std_logic := '1';
  signal clk_en : std_logic := '1';
  signal output : std_logic ;
begin
  uut: nlfsr port map(
    clk,
    reset,
    output );
  process
  begin
    reset <= '0' after 4 ns;
    wait;
  end process;
  process (clk_en,clk)
  begin
    if (clk_en = '1') then
      clk <= not clk after 2 ns;
    end if;
  end process;
end architecture aa;
```

### 2.2 Our automatic converting

One of the excellent advantages of using Linux system is the ability of writing a script code that include all the commands in a single file and run this file in one double-click. That will save time and effort where there is no need to write every command in every time we need to run these commands.

The conversion between the espresso format and VHDL syntax is done with a C program. The C program converts the syntax of the combinational blocks in espresso formats to the corresponding format in VHDL.

We wrote a C-code that can manage the feedback function (updating functions) for the FSR and adding the necessary sequential blocks (flip-flops) in a synthesizable VHDL code.
The Espresso command is integrated with the C-code in one Linux script file as shown in figure (2.3)

![Converter Design Flow Diagram](image)

**Figure (2.3) The Converter design flow**

The converter is designed so that the inputs and the outputs bits in the espresso format must be sorted from n-1 to 0 and given the names x and f as shown in figure (2.1). Our converter can manage both the LFSR and NLFSR and producing the required VHDL code and its testbench. The code of our converter is attached in appendix (C).

Executing this file will generate the following files:

- **Out.pla**: This file has the updating functions.
- **C-code.c**: This file has the C-code.
- **nlfsr.vhd**: The VHDL code of the LFSR (or NLFSR).
- **nlfsr_tb.vhd**: The testbench of the above VHDL code.

This converter which is a text file can have any name. When running the converter, the Linux version of the Espresso tool and the file of the Espresso format that contains the cube representation (input.pla) both must be located in the same directory with the converter.

We can summarize the tasks of our converter which found in appendix (C) in three mains steps:

**Step 1**: Creating the FSR feedback functions in a file named *out.pla* by using the Espresso command: `$ espresso -o qntott input.pla > out.pla`. This step is specified in line number 8 on Appendix (C).

**Step 2**: Creating the C-code named *C_code.c* that contain all the commands needed to create the VHDL files. This step is specified in the lines 14 – 134.

**Step 3**: Compiling the file *C_code.c* in the Linux C-program compiler. This step is specified in the lines 137 – 139.

Now we will give some explaining about the C-code itself which is created in step two above as in the following points:

- The Linux commands in lines (1-11) are to check the existing of the espresso input file and its name. The name of the input file must be “input.pla”, if the user use a wrong name, a text line will appear on the Linux Terminal and will ask the user to change the input file name.
- The Linux command (`cat > C_code.c << EOF`) in the line (14) and (EOF) in the line (134) are to create a file named (C_code.c) that contains all the commands lines...
of the converter starting from line number (15) ending with line number (133).

- Since we need to convert the characters (&, !, |, =) to (AND, NOT, OR, <=) respectively, so we defined in line 23 a pointer (*x2find[ ] ) that indicates to the addresses where these characters are saved. Line (24) has the name of the pointer that indicates to the addresses of the new characters that needed to replace those in line (23).
- In line (36) the converter will open the file out.pla that has the feedback functions and will save it in the memory.
- Lines (47-55) will read the containing of the out.pla file character by character until reach the end of the file. During this task, the converter will calculate how many equal signs “=” there, these characters always indicate to how many bit (registers) that the LFSR consist of, or the order of the (N) LFSR. Line (56) used to close the out.pla file.
- On the lines (59-60) the file out.pla will be opened again and a new file called nlfsr.vhd will create.
- The traditional part of the vhdl code such as the library ieee , entity, architecture, ..etc are added to the output VHDL file using the C command (fprintf).
- Lines (69-75) are used to write the initial value for each register.
- In lines (81 - 103) have the required commands that will search the characters (&, !, |, =) and replace them with the required characters.
- Lines (112 – 127) are to write the testbench file (nlfsr_tb.vhd) using the C-command (fprintf).
- In Line 129 the converter will print the text line “VHDL code with the testbench has been created”.

All the commands of our converter are written in one text file, if we give a name for this file such as run.tcl, then, first we need to run following command:
chmod 755 run.tcl
This Linux command will convert the file (run.tcl) to an executable file. The option (755) means that everyone (not only the owner) can read, write and execute the file (run.tcl).
So, the file now is ready for running. To run this file (run.tcl) we only need to write the following Linux command:
./run.tcl
2.3 Result

We have tested the converter on NLFSRs of 12-bit and 17-bit used a computer with 2.2 GHz, Intel Core 2 Duo Processor. The result was as shown in table (2.1)

<table>
<thead>
<tr>
<th>Order of FSR</th>
<th>Time required to write the VHDL manually</th>
<th>Time required by our converter</th>
</tr>
</thead>
<tbody>
<tr>
<td>12-bit NLFSR</td>
<td>More than one hour</td>
<td>4 sec.</td>
</tr>
<tr>
<td>17-bit NLFSR</td>
<td>More than one hour</td>
<td>7 sec.</td>
</tr>
</tbody>
</table>

Table (2.1) A comparison of the time required to produce the VHDL code manually and our converter

From the table (2.1) and for NLFSR 17-bit, our converter needs only 7 second to create the VHDL code with the test bench compared with more than one hour of writing the same VHDL manually.

By using this converter we can get the VHDL cod with the testbench in several seconds. Then we can use the VHDL code in any other VHDL software such as ModelSim for simulation, or in Synopsys in ASIC design ..... etc. The VHDL cod and the testbench that created from this converter are same as the codes in section 2.1 (depending on the FSR order).
Chapter 3

ASIC Design

ASIC (Application Specific Integrated Circuit) is an integrated circuit (IC) designed for a particular use; it can have hundred million gates build as just simple gates for specific applications or as processors, memory and other large building blocks. Hardware description language (HDL) such as Verilog or VHDL is used to design the ASICs.

Our converter that we discussed in the previous chapter is to generating a synthesizable VHDL file with the testbench. In this, chapter we will extend this converter so that it can generate the VHDL-code and the hardware characteristics of the FSRs in ASIC technology. To calculate the hardware characteristics such as area, timing and power we need to use a specific ASIC design tool called Design Compiler from Synopsys® [24], and ModelSim from Mentor Graphics® [25]. The tool ModelSim is used to simulating the design and generates the Value Change Dump (VCD) file which is required to calculate the power by Design Compiler. Extended the converter will be done by appending the commands of the Design Compiler and ModelSim to the converter as shown in Appendix (D).

In this chapter, we give an introduction to the Synopsys Design Compiler tool. In section 3.2, we explain the steps of calculating the hardware characteristics for the Linear and non-linear FSRs that produced by our converter. The ASIC design commands will be appended to the converter.

In this chapter, we will also calculate the hardware characteristic for the most popular CRCs generating algorithm presented in [6] and give the result in section 3.4.

3.1 Introduction to Synopsys Design Compiler

Design Compiler is a synthesis tool from Synopsys Inc, it is simply take the RTL, VHDL (or Verilog) and a standard cell library as input to produce a technology dependent gate-level-netlist as an output. Figure (3.1) shows an overview of the synthesis tool.

![Figure (3.1) An Overview of the Synthesis tool](image)

It translates the RTL description to components extracted from the Design Ware library and the Technology library. The Technology library contains the basic logic gates and Flip-Flops. The Design Ware library consists of more complex cells such as adders and comparators. Design Compiler performs many steps such as optimization of the high level RTL, technology independent optimizations and technology mapping to the available standard cells in the technology library (target library). Depending on the used constraints, the result will be gate-level-netlist. The constraints such as timing and environmental restrictions (power, area,
process etc.) are specified by the designer. Next, the design is ready for place and route step which is out of our project.

Design Compiler found in two version, Graphic mode (design_vision) which has Graphic User Interface (GUI), and non graphic mode (dc_shell) which has no GUI. In order to run all commands as script in the Design Compiler we need to use the non graphic mode. Using this mode will not invoke any graphic window and all commands can be executed as a text only.

3.1.1 Startup file

It is setup file used by Synopsys Design Compiler to initializing the design parameters and variables and declares the design libraries etc. When the Design Compiler is invoked, all the commands in this file will be read by the tool. The file must be present in the current working directory; the name of this file is (.synopsys_dc.setup). The dependent data included in this file are specified by the user. The containing of the startup file can be as shown in lines from 157 to 191 of Appendix (D).

In this file, we are using the library tcbn90 nm from TSMC. The file has the following important parameters that should be setup before starting use the tool [10]:

- **Search_path**
  It is a parameters used to specify all the paths that the tool should search when looking for a synthesis technology library for reference during synthesis.

- **Target_library**
  This parameter specifies the file that contains all the logic cells that should used for mapping during synthesis.

- **synthetic_library**
  This parameter specifies the synthetic or Design Ware libraries. These libraries are technology-independent, micro architecture-level design libraries.

- **Link_library**
  This parameter indicates to the library that contains information on the logic gates in the synthesis technology library.

- **Symbole_library**
  The library that contains the schematic (visual) information on the logic cells in the technology library will be indicated by this parameter. It is used to draw the design schematics.

3.2 FSRs Hardware Characteristics

**Method**

All the commands used by Design Compiler that needed to calculate the hardware characteristics such as area, timing and power can be written in a single text file or appended to the text file of our converter. The text file is an executable file works in Linux environment. Figure (3.2) shows the flow of our converter combines with the ASIC calculation commands for area, timing and power. This flow illustrated as commands in Appendix (D), i.e. Appendix (D) shows our Boolean cube to VHDL converter combined with a group of commands to compute area, timing and power consumption of FSRs using Design
Compiler and ModelSim tools. In chapter 2 we discussed our converter and how it generates the VHDL files (*nlfsr.vhd* and *nlfsr_tb.vhd*). These two files are needed to calculate area, timing and power.

In this section we will discuss the code in Appendix (D) which consists of two parts. The first part is related to the Cube Boolean converter where the VHDL files are created.

The second part is related to the ASIC design which have a groups of command divided into sections. We can explain these sections as bellow:

1- The section (Creating Directories) on the lines 150-151 used to create the directories in which the files of Design Compiler and ModelSim will save.

2- The next section of the code (Creating .synopsys_dc.setup File) on the lines 153-192 is to create the setup file (.synopsys_dc.setup) which discussed in section 3.1.1.

3- The section (Creating Constraints file) on the lines 194-216 is to create the file which has the commands that specify the design constraints, this file will be called later by the Design Compiler.

This command is to create a clock cycle named “clk” with a period of 4 ns (operating frequency is 250 MHz), 50/50 pulse ratio.

The following commands are used to specify the system interface:

```
set_load [load_of tcbn90gtc/INVD0/I] [get_ports output]
set_output_delay 0.5 [get_ports output] -clock clk
```

Practically, the design input and output ports should be connected to other devices. We can consider that the output design is connected a device acted as load named (INVD0). This device is NOT gate (inverter) with one input (I) and one output (ZN) as described in the

---

**Figure (3. 2)** The converter combined with the ASIC calculations flow
In the output ports, we have to specify a time to be enough for the signals to propagate from the output ports of our design to the next block.

In the linear or non-linear FSR there are no data input ports, so we will not add input ports, but we will do that when we use the code for the CRC generator design. In case there is data input port as we will see in the next section, usually, there is different in the time need by the signals to arrive to the input ports due to routing and other reasons, so we have to specify when signals arrive on the inputs ports (in the CRC design).

The following commands are specify the operating condition and the wire load model:

```tcl
set_operating_condition -lib tcbn90gtc NCCOM
set_wire_load_model -lib tcbn90gtc -name "TSMC16K_Lowk_Conservative"
set_wire_load_mode top
```

The above commands specified the operating environments which are as follow:
- Operating conditions such as voltage, temperature, RC tree and manufacturing process.
- Wire load models which specify the effect of the interconnects on the timing and area.
- System interface characteristics such as input driver, I/O loads and fanout loads.

In the first command above, we chose the default operating condition in the tcbn90gtc library (NCCOM) which is specify the temperature to 25°C and the voltage 1.0 volt. This library has other operating conditions such as WCCOM and LTCOM each one has different value for temperature, voltage etc.

In the second command, we chose the wire load model “TSMC16K_Lowk_Conservative”, other model can be used such “TSMC8K_Lowk_Conservative”, “TSMC16K_Lowk_Agressive” each one has its own specifications. The last commands above specify the wire load mode, which is to determine the wire load model used according to the hierarchy.

**4-** The section (Creating Synopsys_sdf.tcl File) on the lines 218-242 is to create a file named Synopsys_sdf.tcl. In this file, several tasks will be done, such as loading the VHDL design by analysis and elaborate the design, then applying the constraints which discussed in section (3) above. Both the logic-level and the gate-level optimizations will be performed on the design by compiling the design.

Then, we report the results of area, timing and the estimated power and other information about the design. The estimated power in this step is not accurate since it is calculated without using the testbench.

The design need to be saved in ddc, sdf and Verilog formats which will be needed later.

**5-** The section (Creating ModelSim do File) on the lines 244-256 is to create a file named ModelSim.do. This file is called Macro file will be used by ModelSim tool to simulate the design and create the VCD file. The Macro file is very useful to reduce repetitive work. Next, the tool will simulate the design using the test bench (nlfsr_tb) and the (sdf) file that we have already saved previously named (nlfsr_default.sdf).

The simulation time is depending on the type of the FSR, however, in the high order FSR such as 32 bit, VCD file can be very large. VCD file will be discussed in more details in chapter 4.
6- The section (Creating Synopsys.tcl File) on the lines 258-267 is to create a file named Synopsys.tcl. This file contains the commands that read the design again in ddc format. Design Compiler will read the Switching Activity Interchange Format (SWIF) file which will be discussed in chapter 4. Then we report the more accuracy power and save it in a file named (report_power_Final.txt):

7- In the sections (2-6) above, number of separated files are created, each file consist of a group of commands. In the section (Executing Section) on the lines 269-295, no file will be created; it has a number of commands that needed to complete the calculations of the design hardware characteristics as in the following:

```bash
dc_shell -f Synopsys_sdf.tcl
In this command Synopsys Design Compiler will perform all the content of the file (Synopsys_sdf.tcl). The option (-f) is to tell the Design Compiler that the executed command will be found in a file.
mkdir SIM
vlib SIM/WORK
The above commands are to create a directory named (SIM) in the same our project’s directory, and to create our design library in the directory (SIM/WORK). The following command is to map this library so that ModelSim can locate the design library.
vmap work SIM/WORK
The following command is to feed the file (nlfsr_netlist.v) to the ModelSim. This file has all the information about the design’s netlist in Verilog format.
vlog SYNOPS/NETLIST/nlfsr_netlist.v -work ./SIM/WORK
The next commands are to compile this Verilog file with the test bench file (nlfsr_tb) based on the library Verilog file (tcbn90g.v) as shown in the following command:
vcom ./nlfsr_tb.vhd -work ./SIM/WORK
vlog /afs/it.kth.se/pkg/synopsys/extra_libraries/standard_cell/TSMC/tcbn90g_110a/Front_End/verilog/tcbn90g_110a/tcbn90g.v -work ./SIM/WORK
vsim -c work.nlfsr_tb -do ModelSim.do
This command is to run the Macro file (ModelSim.do) in ModelSim to generating the VCD file.
vcd2saif -i ./SYNOPS/SAIF/myvcdfile.vcd -o ./SYNOPS/SAIF/mysaiffile
This command is to convert the VCD file to (saif) format since Design Compiler cannot work with the VCD file. Where the input file will be (myvcdfile.vcd) and the output file is (mysaiffile).
dc_shell -f Synopsys.tcl
In this command Synopsys Design Compiler will perform all the content of the file (Synopsys.tcl). The option (-f) is to tell the Design Compiler that the executed command will be found in a file.

3.3 CRCs Hardware Characteristics

In the previous section, we calculated area, timing and power for FSRs by using the VHDL codes that produced by our converter. In this section we will calculate area, timing and power
for CRC generators presented in [6]. As an example, the following feedback functions are for a parallel CRC of 12 bit (CRC-12) produced according to [6]:

\[
\begin{align*}
\text{f(11)} &= x(11) + x(10) + x(9) + x(8) + x(7) + x(6) + x(5) + x(4) + x(3) + x(0) + d(11) \\
\text{f(10)} &= x(11) + x(2) + x(0) + d(10) \\
\text{f(9)} &= x(11) + x(9) + x(8) + x(7) + x(6) + x(5) + x(4) + x(3) + x(1) + x(0) + d(9) \\
\text{f(8)} &= x(11) + x(9) + x(2) + d(8) \\
\text{f(7)} &= x(10) + x(8) + x(1) + d(7) \\
\text{f(6)} &= x(9) + x(7) + x(0) + d(6) \\
\text{f(5)} &= x(8) + x(6) + d(5) \\
\text{f(4)} &= x(7) + x(5) + d(4) \\
\text{f(3)} &= x(6) + x(4) + d(3) \\
\text{f(2)} &= x(5) + x(3) + d(2) \\
\text{f(1)} &= x(4) + x(2) + d(1) \\
\text{f(0)} &= x(11) + x(10) + x(8) + x(7) + x(6) + x(5) + x(4) + x(1) + x(0) + d(0)
\end{align*}
\]

Where \( d \) is a vector of 12 bit of the data input to the CRC-12, and (+) represent to XOR operation.

The above equations have the generator polynomial:

\[
x_{12} \oplus x_{11} \oplus x_3 \oplus x_2 \oplus x \oplus 1.
\]

The feedback function \( f(11…0) \) above represent to the value of the CRC after two clock cycle as we discussed in chapter 1 section 1.3.2. We append, to the data message, a number of zeros equal to the data length, i.e. \( d(11) \) to \( d(0) \) are equal to zero. Therefore, we can ignore the value of \( d(11) \) to \( d(0) \) where XOR with zero will give the same value.

The VHDL code and the testbench for CRC-12 can be written in the same method in Appendix (A) and (B) respectively.

The test bench we created will feed a (semi) random data message to the design. We suppose the name of the VHDL file is \( \text{nlfsr.vhd} \), and the testbench file is \( \text{nlfsr_tb.vhd} \). Both names are already given before to the output files for our Cube Boolean convertor. Therefore we can use the same code in Appendix (C) but with removing the commands related to the convertor. In another word, we can use the second part of that code which related to the ASIC design.

The FSRs design that generated by our convertor has no data input port, while the CRC circuit design has a data input \( (d) \). Therefore we need to add the following two commands to the section of the constraints of Appendix (C).

\[
\text{set_driving_cell -library tcbn90gtc -lib_cell INVD0 -pin ZN [get_ports d]} \\
\text{set_input_delay 0.5 [get_ports d] -clock clk}
\]

The first command is to adding input interface (INVDO) for the CRC design, this interface is an inverter which is the same inverter that used in the output port as shown in figure (3.3). The second command is to specify the signal time to propagate from the inverter device to the input port of our design.

![Figure (3.3) System interface and Timing](image-url)
3.4 Result

We calculated area, timing and power consumption for some NLFSR. The results are shown in table (3.1) below.

<table>
<thead>
<tr>
<th>NLFSR Types</th>
<th>Early Power (without test bench)</th>
<th>Final Power (with test bench)</th>
<th>Total</th>
<th>Critical Path Length (ns)</th>
<th>Gate Count</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cell Internal (uW)</td>
<td>Net Switch (uW)</td>
<td>Total Dynamic (uW)</td>
<td>Cell Leakage (uW)</td>
<td>Cell Internal (uW)</td>
</tr>
<tr>
<td>3-bit</td>
<td>18.39</td>
<td>2.24</td>
<td>20.63</td>
<td>0.175</td>
<td>22.98</td>
</tr>
<tr>
<td>4-bit</td>
<td>18.08</td>
<td>0.695</td>
<td>18.78</td>
<td>0.18</td>
<td>29.7</td>
</tr>
<tr>
<td>12-bit</td>
<td>141.03</td>
<td>356.6</td>
<td>497.6</td>
<td>12.64</td>
<td>258.64</td>
</tr>
</tbody>
</table>

Table (3.1) Results of NLFSRs using TSMC 90 nm technology, the power values are based on 250 MHz

We also calculated area, timing and power consumption for the CRC generator for the most popular CRCs which presented in [6] and we got the results as shown in table (3.2). Synopsys Design Compiler computes the design critical path delay by calculate the timing of the signal propagation between the registers. In another word, it calculates the time needed by the signal to propagate from the input registers passing through the logic gates and ending with the output registers. Therefore, if we use the VHDL code in Appendix (A) which is for CRC-5 generator, Design Compiler will compute the area and power but not the CPD because the input is not registered. If we register the input, the total area will be increased to be a double of its value. Since the CRC generators circuit usually used as a part of others modules, the internal inputs acquire registers from outputs of previous module. Therefore to make the Design Compiler calculate the CPD, we need to register the input by inserting the signal “x <= d” at the (line 16) of the VHDL in appendix (A) into the clocked process (line 24). By this way we created input register, logic gate and output register which are needed for calculation purpose only.

<table>
<thead>
<tr>
<th>CRC Types</th>
<th>Early Power (without test bench)</th>
<th>Final Power (with test bench)</th>
<th>Total</th>
<th>Critical Path Length (ns)</th>
<th>Gate Count</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cell Internal (uW)</td>
<td>Net Switch (uW)</td>
<td>Total Dynamic (uW)</td>
<td>Cell Leakage (uW)</td>
<td>Cell Internal (uW)</td>
</tr>
<tr>
<td>12</td>
<td>55.15</td>
<td>4.91</td>
<td>60.06</td>
<td>0.7</td>
<td>73.53</td>
</tr>
<tr>
<td>16</td>
<td>73.11</td>
<td>6.96</td>
<td>80.07</td>
<td>0.93</td>
<td>97.23</td>
</tr>
<tr>
<td>16-CCITT</td>
<td>82.60</td>
<td>8.9</td>
<td>91.49</td>
<td>1.17</td>
<td>103.13</td>
</tr>
<tr>
<td>32</td>
<td>258</td>
<td>51.69</td>
<td>309.63</td>
<td>5.10</td>
<td>281.27</td>
</tr>
<tr>
<td>64-ISO</td>
<td>303.15</td>
<td>31.5</td>
<td>334.65</td>
<td>4.39</td>
<td>282.39</td>
</tr>
<tr>
<td>64-ECMA-182</td>
<td>793.3</td>
<td>232.5</td>
<td>1026</td>
<td>17.6</td>
<td>703.47</td>
</tr>
</tbody>
</table>

Table (3.2) Results of different type of CRCs using TSMC 90 nm technology, the power values are based on 250 MHz

35
Gate count \(= \frac{\text{Total Area}}{\text{Area of smallest NAND gate}}\), where \(\text{Area of smallest NAND Gate} = 2.4192 \text{ nm}^2\). From the table (3.2), the power calculated by using the VCD file (Final Power) is more accurate than the early power estimation (Early Power). Using the VCD file, which need to use the test bench, will give the opportunity to calculate the power as if the circuit working in the real environment, therefore the result will be more accurate. We will discuss the VCD file in more details in chapter 4.

The area and power consumption in a design are increased by increasing the number of the gate count in that design which is mean more number of XOR gates. Therefore, increasing the number of the XORs in the RTL design will increased the area and power consumption as shown in table (3.3).

<table>
<thead>
<tr>
<th>CRC Types</th>
<th>Total number of the XORs</th>
<th>Dynamic Power</th>
<th>Area (nm²)</th>
<th>Max no. of the XORs found in a feedback equation</th>
<th>Critical Path Length (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>40</td>
<td>80.89</td>
<td>340</td>
<td>9</td>
<td>0.47</td>
</tr>
<tr>
<td>16</td>
<td>56</td>
<td>108</td>
<td>450</td>
<td>14</td>
<td>0.51</td>
</tr>
<tr>
<td>16-CCITT</td>
<td>72</td>
<td>116.73</td>
<td>543</td>
<td>7</td>
<td>0.48</td>
</tr>
<tr>
<td>32</td>
<td>420</td>
<td>367</td>
<td>2076</td>
<td>16</td>
<td>0.74</td>
</tr>
<tr>
<td>64-ISO</td>
<td>202</td>
<td>303.68</td>
<td>2008</td>
<td>6</td>
<td>0.43</td>
</tr>
<tr>
<td>64-ECMA-182</td>
<td>1970</td>
<td>949.29</td>
<td>6827</td>
<td>36</td>
<td>1.06</td>
</tr>
</tbody>
</table>

Table (3. 3)

CRC-64-ISO has 202 XORs while CRC-32 has 420 XORs, therefore CRC-32 has more area and more power consumption than CRC-64-ISO. The critical path delay is almost depending on the maximum number of the XOR gates that found in a feedback equation. CRC-64-ISO has less CPD than the others because it has less XORs as a maximum number of XORs that found in its feedback equations. CRC-ECMA-182 has the longest path signal propagation through its XORs. Therefore its CPD is the largest one.

Figure (3. 4) Total Dynamic Power for different CRC
Figures (3.4), (3.5) and (3.6) are showing the graphs of the power consumption, timing and area respectively for different type of parallel CRC design based on the result in the table (3.2).
Chapter 4

Spice Model of LFSR Power Diagram

Spice is a general purpose source used to simulate the analog circuits [26]. PSpice is an example of Spice tool, where the result can be saved as an ASCII text file that contains the information about the design such as current, voltage and the netlist representation of the circuit.

In this chapter, we didn’t use PSpice tool, but we designed a simple interface that simulates the spice characteristics of the chip power for the LFSR that used in the CRC generator circuit. With this interface, the user not only can see the average power but also he can see the power diagram of the chip during the operation.

The interface is a simple Linux script code used to calculate the power per cycle which is a good estimation with real power shown in circuit level tools such as PSpice and Virtuoso. The advantage of this script is giving the user a good estimation of the power diagram during the execution time which is similar to probing the current of the circuit.

The presented interface in this chapter can be used in some of the research projects done in our research group. For example, it is can be used to estimate the power of FSRs during power analysis attack presented in paper [8].

We also modified the VHDL code for the same LFSR to calculate the design switching activity (number of switching). We will discuss the relation between the power consumed every cycle and the switching activity and we will use the correlation to see how they are identical to each other.

Introduction

We have already used the tool of Synopsys Design Compiler to calculate area, timing and power consumption for different CRC generators. Power consumption in an ASIC design can be divided in two categories: static power and dynamic power [10].

1- Static Power
   This power is the power that consumed by the gates when it is not switching. Static power is produced by the current that flow through the transistors even when those transistors are turned off. This power is wasted energy which is not used for energy useful task.

2- Dynamic Power
   This power is consumed when the circuit performing some function. It occurs when the signals that go through the circuit devices change their logic state (switching activity) e.g from (0) to (1) or from (1) to (0). Dynamic power consists of two types: Switching Power and Internal Power.

   > Switching Power
     This power is consumed during charging and discharging the load capacitance at the cell output, where the interconnect (net) capacitance and the gate capacitance are charged and discharged. Switching power is increase when the switching activity increases, it affected by the operating frequency.
The operation of charging and discharging the internal cell capacitance will consume power that calls internal power. It also includes the short circuit power. When the circuit is active and there is a logic transition, the transistors type P and N are both ON simultaneously for a short time causing direct connection (short circuit) from $V_{dd}$ rail to ground rail.

### 4.1 Calculation of Power per Cycle

The CRC generator design is an application of the LFSRs, therefore we have a number of states depends on the order of the CRC generator and the type of the LFSR. The total number of these states is called *period*. The maximum possible period is $2^n - 1$, where n is the order of the LFSR. The previous expression has (-1) which is the case when all registers have “0” where this state cannot be achieved and the shift registers will never change its value. Therefore, this state must be avoided.

In general, LFSR can have a period less than the maximum possible period. For example, the 12 bit LFSR that used in the CRC generator (it is also used in paper [6]) has a number of states equal to $\left(\frac{2^{12}}{2} - 1\right)$, which is a half number of the maximum states. Therefore, this LFSR doesn’t have the maximum possible period. The total number of states for this LFSR is equal to $\left(\frac{2^{12}}{2} - 1\right) = 2047$ states, where the case when all bit equal to zero are avoided in each half. Therefore we removed one state (-1) from every half.

In chapter 3, we calculated the power consumption for different type of CRC generators that adopted from paper [6]. The power calculated represented to the average power during the simulation time. By using the same method, we can calculate the average power for the LFSR.

We will calculate the power consumption for the LFSR in every clock cycle. The average value for the powers per cycle should be identical to the average power that it can be calculated directly as we seen in chapter 3.

Appendix (E) shows the script code that used to calculate the power per cycle and plot the result as a power diagram in Matlab.

To calculate the power per cycle, we need to use the VCD file, this file records all the switching activity of the design for one period starting from the time 0ns and ending to the time $\left(\frac{2^n}{2} - 1\right) \times (cycle's\ length)$. The size of the VCD file can be very large depend on how long time we want to simulate the design and how often a net, pin or port went through either a 0 to 1 or a 1 to 0 transition.

Design Compiler tool can manage the format of type SAIF (Switching Activity Interchange Format) but not the VCD format, therefore, and as we did in the previous chapter, we
converted the VCD format to SAIF. SAIF file has all the information about the switching activity for the design during the simulation time.

The Linux script code in Appendix (E) can be explained as follow:

- The first step of calculating the power per cycle is by extracting the SAIF file per cycle from the VCD file, and uses each file in the Design Compiler to calculate the power and save the result in one file’s report. This step is illustrated in lines (173-191) of Appendix (E). The first part is to generate the SAIF files per clock cycle. The value of the variable (k) which start from 4ns and ended with the length of one period (period = 50) in a steps of 4 ns (in case of using a clock cycle of 4ns length) will be appended to the name of the new SAIF files (mysaiffiletiming) in line (184). The option (-time $i $j) of the command (vcd2saif) will specify the time in which the converting to the SAIF will be done. We can give any name for the above file such as (run12PowerCycle.tcl).

- The second part is to invoke the (dc_shell) and executing the commands of the file (Synopsys_PowerPerCycle.tcl) as shown in lines (158-171). In this file, Design Compiler will read every SAIF file (the switching activity in one cycle) alone and calculate the power per cycle. The results will be appended to each others in one report file.

- Design Compiler will calculate the power per cycle and will produce a file named (report_power_perCycle.txt) that has all the results of the power per cycle as shown in line (167).

- The report file will have \( \frac{2^n}{2} - 1 \) values; it takes some time to completing the report file depending on the length of the period. Since the file report also have others information such as the date, library etc, so we need to extract the results in order to plot them in Matlab as shown in lines (195-221). Line (196) used to recognize the required items regarding to power result and save them in another file. It is used to select all the lines in the file (report_power_perCycle.txt) that has the words (Cell Internal Power =) with its value and save them as a column in a temporary file named (Internall.log). Line (197) used to select only the value after the sign (=) and save these values in a new file named (Internal.log). Line (198) is to delete the temporary file (Internall.log). The same procedure will be applied on the other two items (Net Switching Power = and Total Dynamic Power =). Another group of commands used to print the time sequences as a column starting from (4ns) ending with (the length of one period), then collecting all the four columns in a new file named (out.dat) and delete all the temporary files.

The result can be plotted by using the the Matlab commands in the lines (225-244). The tool will invoke the Matlab to plot the power diagram.
Result
We have calculated the power per cycle for the LFSR-12 bit circuit for one period. By using
the same frequency that we have used in chapter 3, which was 250 MHz, the period length
will equal to 8188 ns. The result is plotted as shown in figure (4.1).

Figure (4.1) Power per cycle plot for LFSR-12, Max Dynamic power is 108.1uW at 4120 ns.
Max Net Switching power is 8.275uW at 4120 ns.

From figures (4.1) and (4.2c), we can see that the Maximum Dynamic power is (108.1uW)
occurrts at the time (4120 ns). Since the length of the clock cycle in this design is (4ns), the
first clock cycle will take the time (0-4ns) and the second one will take the time (4-8ns) and
so on. Therefore, the clock cycle at which we have the maximum Dynamic Power will take
the time (4120-4124ns). The average (mean) total dynamic power is (80.88uW).

Figure (4.2) Internal, Switching and Dynamic average power for CRC-12 bit

From figure (4.2b), the Maximum value of the Net Switching Power is (8.275uW). If we
extend the time of the maximum Net Switching Power (8.275uW) (the black signal at the
bottom) in figure (4.1), we will find that this value occurs at the time (4120-4124 ns) as
shown in figure (4.3).
Figure (4.3) Max Net Switching Power is (8.275uW) at the time (4120 ns)

Table (4.1) below illustrate the Internal, Switching and Dynamic power which calculated directly together with the values that we got in above. The results are almost same in the two approaches.

<table>
<thead>
<tr>
<th>The Direct Average Power (uW)</th>
<th>The Average Power calculated from the Power Per Cycle (uW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cell Internal Power</td>
<td>Cell Internal Power</td>
</tr>
<tr>
<td>76.52</td>
<td>75.99</td>
</tr>
</tbody>
</table>

From figure (4.1) and the result in above we conclude that the maximum dynamic power is very affected by the maximum switching power, where both occur at the same time.

### 4.2 Calculation of Switching Activity

As we discussed before, the power calculation of a design is much related to the how often a net, pin or port went through either a 0 to 1 or a 1 to 0 transition. This transition is called switching activity. If we know how many time a design switched between 0 and 1, then we can have an idea about the power consumed by the design. We can modify our design in the RTL level to calculate how many times that the registers are changed between “0” to “1” or “1” to “0”. Then, we can correlate these values with the Net Switching power values obtained by power per cycle.

To calculate how many times a LFSR circuit is switching between 0 and 1 in the RTL level, we need to modify our VHDL code of the LFSR design by adding some variable, signal and counters.

For the 12 bit LFSR design, we have modified the VHDL code to be as shown in appendix (F). Where, the variable (count_per_cycle) is to calculate the number of switching every
clock cycle, the variable (counter) is to calculate the total number of switching in one period \(\left(\frac{2^n - 1}{2}\right)\). The variable (counter_continue) is to calculate the number of switching during all the simulation time; it can be less or more than one period depending on the time of the simulation.

We simulate the modified design with a length of clock cycle equal to 4 ns (250 MHz), the total number of switching calculated for the registers of LFSR-12 bit in one period is: \(\text{counter} = 12284\). As shown in figure (4.4) below.

![Figure (4.4) LFSR-12 simulation shown total number of switching for one period](image)

If we check the number of switching per clock cycle during the simulation, we will find that the maximum value is (\(\text{count_per_cycle} = 12\)) occurs in the time (4120-4124) ns as shown in figure (4.5).

![Figure (4.5) The LFSR-12 simulation shows the Max numbers of switching per cycle](image)

From figure (4.5), at the time 4120 ns where the maximum number of switching is occurred, we have the following values:

At the time (4116-4120 ns) the LFSR registers (test) has the value (101010101010).

At the time (4120-4124 ns) the LFSR registers (test) has the value (010101010101).

If we compare the above two values, simply we will find that all bits in the LFSR registers at the time (4116-4120ns) will be changed into a new value either from “0” to “1” or from “1” to “0” at the time (4120-4124ns). Therefore, we have a number of changing (number of switching) equal to 12 which is the maximum value that can be occurs at one clock cycle in the registers of the LFSR design type 12 bit.
If we compare this result (the time 4120 to 4124 ns) in the above simulation with the result in figure (4.1) and figure (4.3), we will conclude that this time is the time in which the maximum Net Switching Power is occur.

Figure (4.6) gives more details about the switching activity in term of the VCD file.

Figure (4.6) The Containing of the VCD file at (4120-4124) ns. Left hand side: The containing of the VCD file for the time (4120 to 4124 ns) only, Right hand side: Description of each item in the left hand side

Each item in the left hand side of the figure (4.6) is recorded because its value is changed either from 0 to 1 or from 1 to 0. During the time (4120 to 4124 ns) all the LFSR contains are changed or switched to a new value.

The highlighted values on the left hand side of figure (4.6) represented to the register values, e.g. the value 0/ means that the register number 9 x(9) changes its value from “1” to “0”. If we rearrange the highlighted value in horizontal direction according to the sequence of the registers of the LFSR, from x(11) to x(0) as follow:

\[
\begin{array}{cccccccccccc}
0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
x(11) & x(10) & x(9) & x(8) & x(7) & x(6) & x(5) & x(4) & x(3) & x(2) & x(1) & x(0)
\end{array}
\]

It is obviously that the sequence in above is identical to the sequence of the same vector (n) at the time (4120-4124) in the figure (4.5).

Since we converted the VCD file to SAIF file, the contains of the SAIF file for the time (4120-4124 ns) will record the change of all register in a file as shown below: (shows a part of the SAIF file).
The above SAIF file shows annotations of some signals in the LFSR-12 bit, where:
- **T0**: the time that the signal had logic value ‘0’.
- **T1**: the time that the signal had logic value ‘1’.
- **TX**: the time that the signal had an unknown logic value ‘X’.
- **TC**: the number of ‘0’to’1’ and ‘1’ to ‘0’ transitions observed.
- **IG**: the number of ‘0’-‘X’-‘0’ and ‘1’-‘X’-‘1’ transitions observed.

The values of all TC item in this file are equal to 1 which means all the LFSR registers are changed to a new values.

### 4.3 Correlation

Correlation is a general term used to describe the relation between two (or more) variables. As we seen before, there is a strong relation between the number of switching activity and the power consumption. We can manage the data we plotted in figure (4.1) in section 4.1. This data has the information about the *Time*, *Cell Internal Power*, *Net Switching Power* and *Total Dynamic Power* calculated per clock cycle. Since we already calculated the number of switching in which the registers of the LFSR changed its value between “0” and “1” and vice versa, we can correlate these data with the power consumption per cycle.

The correlation can be done by using Matlab, the following Matlab code can perform the correlation between the power consumption per clock cycle and the number of registers switching per clock cycle:
close all
load Switching.dat
load Net_Switching_Power.dat
load Total_Dynamic_Power.dat
load Cell_Internal_Power.dat

Corr_Cell_Internal_Power = corr2 (Switching , Cell_Internal_Power);
Corr_Net_Switching_Power = corr2 (Switching , Net_Switching_Power);
Corr_Total_Dynamic_Power = corr2 (Switching , Total_Dynamic_Power);

Executing the above commands will produce the result on the Matlab Workspace as shown in figure (4.7).

![Workspace](image)

Figure (4.7) Correlation result in Matlab

From the figure (4.7) we can conclude the following result:
Correlation between switching activity and Cell Internal Power  = 0.9212 (92.12%).
Correlation between switching activity and Net Switching Power  = 0.6571 (65.71%).
Correlation between switching activity and Total Dynamic Power  = 0.9032 (90.32%).

Where:
(Total dynamic power = Cell Internal Power  +  Net Switching Power).

It is clear that the Total Dynamic power is very correlated (related) with the how many times the design is transition between either ‘0’ to ‘1’ or ‘1’ to ‘0’.

![Figure](image)

Figure (4.8) Number of switching per cycle for the LFSR-12 during one period in 250 MHz
Figure (4.8) shows the number of switching per clock cycle, we can see that the time position in which the maximum number of switching is occur.

![Bar Chart](image)

Figure (4.9) Max number of switching at the time 4120-4124 ns

Figure (4.9) shows the maximum number of switching at the time 4120-4124 ns.
Chapter 5

CRC hardware characteristic in FPGA

Introduction

In this chapter, we have computed the hardware characteristics for the same CRC generators which presented in [6] by using Altera Quartus tool [27] and ModelSim [25]. The computation is done by using a script without invoking the GUI of these tools in Linux environment. The script file does the compile, synthesis, fitting, assembly and timing analysis automatically.

At the end of this chapter we give the result of calculation using Altera Quartus tool. Furthermore, we give the result for the same CRC generators by using Xilinx tool [11], [12] which calculated by using GUI without script code.

5.1 Background

Field Programmable Gate Array (FPGA) is a very capable device in many applications. It has a number of logic blocks of semiconductor devices which can be programmed to perform a lot of applications from basic digital gate level to complex image processing algorithms. The implementation of a logic design with an FPGA can be described in the following general steps [11]:

1- By using a Hardware Description Language (HDL) such as VHDL or Verilog we can describe our design to meet the goal of the design.

2- Transform the HDL into a netlist by using logic synthesizer software, where netlist is a description of the various logic gates in the design and their interconnection way.

3- Mapping the logic gates and interconnections into the FPGA by using the implementation tools. The FPGA has many Configurable Logic Blocks (CLBs) that can be decomposed into look-up tables (LUTs) that perform logic operations. The netlist gates are collected into groups by the mapping tool, these groups fit into the LUTs. By using the Place and Route tool, these groups assigned to specific CLBs while opening or closing the switches in the routing matrices to connect them together.

4- After completing the implementation phase, the state of the switches in the routing matrices is extracted by a program to generate a bitstream. The bitstream is consisting of zeroes and ones that represented to the open or close switches.

5- Now the bitstream is ready to downloading into a physical FPGA chip. After completing the download process, the FPGA will perform the operations specified by the HDL code where the electronic switches in the FPGA will open or close in response to the binary bits in the bitstream.
In this chapter we will compute the hardware characteristics (area, timing and power) for the design of the CRC generator presented in [6].

5.2 The hardware characteristics in Altera

The Altera Quartus tool provides a complete multiplatform design environment that can be adapted to a specific design needs [27]. The commands that used by this tool to calculate the hardware characteristics can be written as a script in a text file. Using the script file will make the computation very easy even for the users who do not have a good knowledge in the hardware characteristics. We have set all the computation steps in a script file that can be executed in Linux environment as shown in appendix (G). This script file contains all the commands of Altera Quartus and ModelSim tools that needed to compute the hardware characteristics of a VHDL design. The ModelSim is used to simulate the netlist of the design and create the VCD file which is used to calculate the design power consumption.

In chapter 3 and during computation the hardware characteristics in ASIC design, Design Compiler cannot accept the VCD format directly. Therefore, we converted the VCD file format to SAIF format. In FPGA design, Altera Quartus can use the VCD format directly without converted it to another format.

After the running of the script file is completed, we will get the result as a number of reports which consist of the hardware characteristics such as area, timing and power consumption. In our script, we compute the power consumption by using the PowerPlay Power Analyzer tool which is included in Quartus II tool. The result of power calculation will indicate to the activity of the design over time which consists of the static power and dynamic power consumption in the design. Where the dynamic power is the power that consumed due to signal activity or toggling in the design and the static power is consumed regardless of design activity.

The Script file in Appendix (G) consists of three main sections:

- Creating the Quartus TCL file (line 1-38).
- Creating the ModelSim micro file (line 40-49).
- Executing (line 51-67)

We can explain each section briefly as follow:

1- The first section is to create the file that used by Quartus tool which has a group of Quartus commands work to set the tool to the required setting. In the following we have some of these commands:

- Lines 2-3 are Linux commands for writing contains of the file and gives it a name.
- Lines 5-6 are to load the packages of flow and report.
- Line 8 is to create a new project with overwrite the previous one if found.
- Lines 10,11 and 12 are to specify the family type, device and speed grade
- Line 13, is to specify the VHDL file name.
- Line 14, is to assign the design work frequency.
- Lines 15-20 are to specify the simulation tool and assigning the name of the unit under testing.
- Line 27 is to specify the type of fitter effort.
- Lines 28-33 are to prepare the tool for power calculation such as the name of the VCD file and its model name.
- Lines 35-36 for compiling, closing the project.
- Line 38 is a Linux command to close or ending the writing of the file.
2- In this section, a micro file of ModelSim will be created for simulation purpose. This file will be used by ModelSim for post simulation the design for a specific time and creating the VCD file that needed to compute the power consumption. This file is consist of the following four commands:
   - Line 44 is to simulate the (.sdo) file that created by Quartus tool with the testbench.
   - Line 45, in this command, ModelSim will load the TCL file that created by Quartus tool which has the command of creating the VCD file.
   - Lines 46-47 are to run the simulation for a specific time and closing the ModelSim tool respectively.

3- This is the executing section which has the commands that used to run the above two files in addition to others commands as follows:
   - Line 55 is to open the Quartus tool in shell mode without invoking the tool GUI and loading the Quartus file that created in the first section of our script file.
   - Lines 56-57, 59-60 and 62-63 are to create and mapping the libraries “altera”, “cycloneii” and “work” respectively. Where cycloneii is the library name for the family type Cyclone II.
   - Line 58 is compiling our design with altera primitive components and altera primitive libraries respectively in ModelSim.
   - Line 61 is to compiling the design with Cyclone II atoms and Cyclone components libraries.
   - Line 64 is to compile the (.vho) that created by the Quartus tool with testbench.
   - Line 65 is to simulate the design by using the micro file that created in section 2 of our script with the testbench to create the VCD file. Where the option (-c) is for simulation in ModelSim without invoke the tool GUI.
   - Line 67 is to run the PowerPlay Power Analyzer to compute the power consumption.

The reports of area and timing will be created directly after the Quartus tool compiles the design. The report of the power consumption will be generated later after simulate the design by the ModelSim and running the PowerPlay Power Analyzer tool.

In our script code, we target the CRC- generator design to the device of Cyclone II family. In order to use another family, we just change the name “cycloneii” to the required family. The old device families such as FLEX are not supported by PowerPlay Power Analyzer tool. Therefore, using the old version of Altera Quartus family devices in our script will not generate any power report, only area and timing reports will be generated.

5.3 The hardware characteristics in Xilinx

In this section, we will discuss briefly computation of the hardware characteristics using Xilinx tool in GUI, i.e. no script file will be used.

5.3.1 Area and Timing calculation

For area and timing computation, we used Xilinx ISE 13.1 tool. Before loading the VHDL code, family name and the target device should be specified. We have used Virtex-5 family and target the design to the xc5vlx30 device using the package FF324 with speed grade of (-3). Xilinx tool will convert the design into a logic circuit during the synthesize processing. The VHDL code will be read by the synthesizer which will transform it into a netlist of gates.
The synthesized process should be completed without problem. The netlist of gates will be translated, mapped, placed and routed into the logic resources of the FPGA.

The Translate process converts the netlist into a Xilinx specific format and annotates it with any design constraints that may specify by the user. The netlist is decomposed by the Map process and rearranged so that it fit nicely into the circuitry elements contained in the specified FPGA device. The Place & Route process which assigns the mapped elements to specific locations in the FPGA and sets the switches to route the logic signals between them. The tool now is finished the process of calculation for many information about the design. The area and timing information can be found in the synthesis report.

5.3.2 Power calculation

As we seen in chapter 3, the power calculation of the CRCs generator circuits that implemented on ASIC consists of tow type (Static power and Dynamic Power). In FPGA the total power consumption is the sum of two components:

- **Static power**: This power is result primarily from transistor leakage current in the device which is exists even when the transistor is logically OFF. The gate oxide and the source to drain are responsible on the leakage current.
- **Dynamic Power**: This power is due to the design activity and switching events in the core or I/O of the device, where the node capacitance, switching frequency and supply voltage are used to calculate the dynamic power.

In Xilinx software, there are two methods used to calculate the power: *Power Estimation Using XPE (Xpower Estimator)* and *Power Analysis Using Xilinx Power Analyzer (XPA)*.

5.3.2.1 Power Estimation Using XPE (Xpower Estimator)

In this approach, a spreadsheet is use to estimate the power used in the pre-design and pre-implementation stages of a project [12]. The power calculated by XPE can be not very accurate but it helps with architecture evaluation and device selection. XPE can also help the user to choose the appropriate power supply and the thermal management components. In this project we used XPE spreadsheet version 13.1from Xilinx. In the section 5.1 we synthesized the VHDL code of the CRCs generator designs. For every design, a file with the extension (.mrp) was created. This file should be imported to the XPE tool, which has the information that needed by the XPE to calculate the power of the design.

The XPE tool has some parameters that needed to be changed as in the table (5.1).

<table>
<thead>
<tr>
<th>Ambient Temperature ($^\circ$C)</th>
<th>AirFlow (FLM)</th>
<th>Heat sink</th>
<th>Frequency (MHz)</th>
<th>Toggle rate clock</th>
<th>The I/O toggle rate</th>
<th>Logic toggle rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>0</td>
<td>0</td>
<td>250</td>
<td>100%</td>
<td>75%</td>
<td>75%</td>
</tr>
</tbody>
</table>

*Table (5.1) The XPE parameters*

The considered the operation frequency same as the frequency that we used in ASIC design.
By preparing the above values, the tool is ready to calculate the estimated power consumption of the circuit design.

5.3.2.2 Power calculation Using Xilinx Power Analyzer (XPA)

The XPA tool from Xilinx is used to calculate the power in more accurate than XPE and does an analysis on real design data [12]. The simulation activity files (VCD or SAIF) that produced from a simulation tool such as ModelSim is used by XPA for accurate power analysis. To use the XPA, the design should be fully analyzed and placed & routed and generated an NCD output file (the physical constraints file).

The tool Xpower is integrated with Xilinx ISE software. The design file, physical constraints file (both are automatically generated during synthesis process in Xilinx ISE) and the VCD file should be loaded to the Xpower tool.

In the section 5.2.1 of XPE, the value of Ambient Temperature and LFM Airflow was 25c and 0 respectively which should be same values in the XPA.

As an example, the total power for the CRC-12 generator design is 0.430 watt compared with 0.415 watt in the XP Estimator; obviously both calculations are close to each other.

Power Analyzer tool (XPA) can give us more details about our design such as the toggle rate for each part of the design according to the VCD file.

5.4 Results

As we explained in the section 3.4 of chapter 3, Altera Quartus and Xilinx tools also needed the registers to compute the CPD. Therefore we need to modify the VHDL code as we explained previously in order to calculate the CPD in addition to the area and power consumption.

5.4.2 Result of Altera tool

In this section we calculated the Area and Timing for different CRCs based on the paper [6] in Altera Quartus II Version 8.1 web edition software.

<table>
<thead>
<tr>
<th>CRC Types</th>
<th>Family : FLEX10A</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(Area) LUTs</td>
</tr>
<tr>
<td>12</td>
<td>14</td>
</tr>
<tr>
<td>16</td>
<td>21</td>
</tr>
<tr>
<td>16-CCITT</td>
<td>23</td>
</tr>
<tr>
<td>32</td>
<td>103</td>
</tr>
<tr>
<td>64-ISO</td>
<td>69</td>
</tr>
<tr>
<td>64-ECMA-182</td>
<td>392</td>
</tr>
</tbody>
</table>

Table (5. 2) The hardware characteristics for the CRC presented in [6] using Altera’s FLEX10A, CPD is the Critical Path Delay.
The Altera family type FLEX10A is an old version of Altera’s families, and it can provide a minimum CPD of 6ns (166.67MHz).

Table (5.2) shows the results of different type of CRC implemented using Altera FLEX 10A, FPGA.

Table (5.3) shows that the design area is related to the total number of the XOR gates in the design, where more gates will occupy more area from the target device.

In the timing column, the CPD is related to the maximum number of the XOR gates found in a feedback equation. The signal propagation is increased by increasing the number of XORs in a feedback equation, this make the signal pass through a longer path, i.e. high CPD. Therefore, the CRC-64-ISO has the minimum length of the CPD compared with the others as shown in the table (5.3).

<table>
<thead>
<tr>
<th>Family : FLEX10A</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CRC Types</strong></td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>16</td>
</tr>
<tr>
<td>16-CCTT</td>
</tr>
<tr>
<td>32</td>
</tr>
<tr>
<td>64-ISO</td>
</tr>
<tr>
<td>64-ECMA-182</td>
</tr>
</tbody>
</table>

Table (5.3) The relation between the No. of XORs and the results

5.4.1 Result of Xilinx tool

Table (5.4) shows Power, Area and Timing for six common CRCs, these values is calculated based on Virtex-5 that target on the device XC5VLX30 using the package FF324 with speed grade of (-3).
The CRCs used in this table are based on the paper [6]. The table shows that the powers calculated by using XPE are very close to the power that calculated by using XPA. The Power consumption and area depend on the CRC feedback equations. Usually, the value of power and area are increased by increasing the total number of the XORs in the feedback equations as shown in table (5.5).

Table (5. 4) The results for different type of CRC based on Virtex5-XC5VLX30-FF324 and speed (-3), Powers calculation are based on a frequency of 250 MHz

<table>
<thead>
<tr>
<th>CRC Types</th>
<th>Total Power Out of 19200 (w)</th>
<th>Leakage Power (w)</th>
<th>Total Power Out of 19200 (w)</th>
<th>Leakage Power (w)</th>
<th>Max no. of the XORs found in a feedback equation</th>
<th>Critical Path Length (ns)</th>
<th>Fmax (MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>0.415</td>
<td>0.293</td>
<td>0.430</td>
<td>0.296</td>
<td>15</td>
<td>1.408</td>
<td>710.2</td>
</tr>
<tr>
<td>16</td>
<td>0.456</td>
<td>0.294</td>
<td>0.463</td>
<td>0.297</td>
<td>19</td>
<td>2.055</td>
<td>486.6</td>
</tr>
<tr>
<td>16-CCITT</td>
<td>0.457</td>
<td>0.294</td>
<td>0.466</td>
<td>0.297</td>
<td>20</td>
<td>1.408</td>
<td>710.2</td>
</tr>
<tr>
<td>32</td>
<td>0.622</td>
<td>0.299</td>
<td>0.647</td>
<td>0.302</td>
<td>98</td>
<td>2.856</td>
<td>350.1</td>
</tr>
<tr>
<td>64-ISO</td>
<td>0.951</td>
<td>0.307</td>
<td>0.813</td>
<td>0.308</td>
<td>65</td>
<td>1.798</td>
<td>566.2</td>
</tr>
<tr>
<td>64-ECMA-182</td>
<td>0.956</td>
<td>0.308</td>
<td>0.997</td>
<td>0.313</td>
<td>421</td>
<td>3.880</td>
<td>257.7</td>
</tr>
</tbody>
</table>

Table (5. 5) Total number of XORs, power, area and timing for different CRC generators design

The critical path delay is depending on the signal longest path propagation. It is also related to the maximum number of the XORs that found in a feedback equation, increasing this number will cause increasing in the CPD. Therefore, CRC-ECMA-182 has the longest CPD than the others because it has the largest number of XORS that can be found in one or more than one feedback equations. The CRC-ECMA-182 is also has the largest area and power consumption than the other because of the high number of the XORs in its design.

For the CRC-64, it has a CPD less than the CRC-16 in spite of it has a number of XORs larger than in CRC-16. This is because CRC-64-ISO has less maximum number of XORs that found in a feedback equation than CRC-16. However, the synthesis process and others parameters such as net also effected on the CPD.
Chapter 6

Conclusion and future work

6.1 Conclusion

In this project we developed an automatic converter that allowed us to directly produce the VHDL code from the cube representation of incompletely specified multiple-output Boolean function given in Espresso format. The converter can produce the VHDL in several seconds, while a time of minutes to hours can be needed to write the same VHDL manually. The converter uses the updating functions of FSRs in the Galois configuration, it handle both, LFSRs and NLFSRs as well.

In general, the time and effort needed to write the VHDL code manually is increased by increasing the order and the type of the FSRs, where the number of the feedback equations can be increased by increasing the FSRs order. Since we use Espresso tool to minimize the cube representation of the Boolean functions, the Espresso output format consist of a characters of (|, !, & , = ). When the FSRs order grows, many characters of this kind are present in the final equations. In order to keep the functionality for these characters in the VHDL synthesizer, they must be converted to the form (OR, NOT, AND, <=) respectively. The converter that we constructed in this project can performs all these tasks in a few seconds and without need a prior knowledge in VHDL code.

In this project, a walkthrough of hardware characteristics in ASIC design was also undertaken. We have used the power of using script commands in Synopsys Design Compiler and ModelSim in the calculations, and then we appended these commands to our converter to get a compatible tool for both creating the VHDL code and computing the hardware characteristics in ASIC design. This tool can be very helpful for the groups of users which do not have any knowledge about hardware design of a chip. The user only needs to define its design in espresso format. Calculating the hardware characteristics such as area, timing and power in a fast way will give the user or the developer a good opportunity to modifying the FSRs algorithms and recalculating these parameters in very short time.

We evaluated the hardware characteristics for CRCs presented in [6] in ASIC design. We discussed a simple interface that gives the user the opportunity to see the power diagram in real time without using the circuit level tools (such as PSpice). We also examined the relationship between the power consumption and the switching activity for one of the LFSR that used in the CRC generators circuit. Where, we proved that increasing the number of transition between '0' and '1' will increase the total power consumption in the LFSRs.

We computed the hardware characteristics for the parallel CRCs presented in [6] in FPGA by using Xilinx and Altera Quartus tools. We compare our calculation with a previous work for six common CRCs and we have got better results in area and sometime in the design critical path delay.
6.2 Future Work

The convertor we developed is designed specifically for the linear and non-linear FSRs in the Galois configuration. As a future work, the tool can be modified as follows:

- The tool can be extended to manage the FSRs in Fibonacci configuration in addition to the Galois configuration.
- The tool can be modified to manage other circuits rather than FSRs such as the normal combinational block or an application of the logic gates. The tool can potentially be updated to more complex designs.
- The input format we defined in our thesis is Espresso format, but the tool could potentially be updated to another similar format.
- The tool can be easily combined with the script commands of Altera Quartus tool to automatically calculate the design hardware characteristics in FPGA and ASIC as well.
- Develop another tool that can generate the VHDL code and the hardware characteristics of CRCs quickly from any polynomial generator.
Appendix A- VHDL code for CRC-5 bit

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity nlfsr is
  port( clk : in std_logic;
       reset : in std_logic;
       d   : in std_logic_vector (4 downto 0);
       output : out std_logic_vector (4 downto 0) );
end entity nlfsr;

architecture aa of nlfsr is
  signal x  : std_logic_vector (4 downto 0);
  begin
    x <= d;
    process (clk,reset,d)
    begin
      if  (reset='1') then
        output <= "00000";
      elsif (clk'event and clk='1') then
        x <= d;
        output(4)  <=  x(0)  xor x(2)  xor x(4)  ;
        output(3)  <=  x(1)  xor x(3)   ;
        output(2)  <=  x(0)  xor x(2)   ;
        output(1) <=  x(0) xor x(1) xor x(2) xor x(4)  ;
        output(0)  <=  x(0) xor x(1) xor x(3)  ;
      end if;
    end process;
  end architecture aa;
```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.all;
USE ieee.numeric_std.ALL;

ENTITY nlfsr_tb IS
END nlfsr_tb;

ARCHITECTURE aa OF nlfsr_tb IS

COMPONENT nlfsr

PORT(
    clk : IN  std_logic;
    reset : IN  std_logic;
    d      : IN std_logic_vector (4 downto 0);
    output  : out std_logic_vector (4 downto 0)  );
END COMPONENT;

--Inputs
signal clk : std_logic := '1';
signal reset : std_logic := '1';
signal clk_en : std_logic := '1';
signal d   : std_logic_vector (4 downto 0) := "00000";

--Outputs
signal output  : std_logic_vector (4 downto 0) := "00000";

BEGIN

uut: nlfsr PORT MAP (
    clk => clk,
    reset => reset,
    d => d,
    output => output  );

process
variable i : std_logic_vector (4 downto 0) := "00000";
variable j : std_logic_vector (4 downto 0) := "00000";
begin
reset <= '0' after 4 ns;
wait for 4 ns;

if (clk'event and clk='1') then
    i := i + "11";
    j := j + i;
    d <= j;
end if;
end process;

process (clk_en,clk)
begin
if (clk_en = '1') then
    clk <= not clk after 2 ns;
end if;
end process;
end architecture aa ;
Appendix C - The Boolean Cube to VHDL Converter

```c
#include <stdio.h>
#include <string.h>
#include<stdlib.h>

int main()
{
    char *x2find[] = {"f", "=", "!", "&", "|"};
    char *x2rep[] = {"x", "<=" , " not ", " and ", "or"};
    char *x[999];
    char ch;
    char espresso_file_name[255];
    int BitOrder = 0;
    int i;
    int j;
    FILE *fp1,*fp2,*fp3;
    FILE *fp1 = fopen(espresso_file_name, "r");
    if (fp1 == NULL)
    {
        printf("n File does not exist, please check file name!
        n File does not exist, please check file name!
        n File does not exist, please check file name!
        ");
        exit (1);
    }
    else
    {
        printf("OK, File exist now!
        ");
        print("n File does not exist, please check file name!
        n File does not exist, please check file name!
        n File does not exist, please check file name!
        ");
    }
    while(EOF!=(ch =fgetc(fp1)))
    {
        if (ch == *x2find[1]) // search "="
        {
            BitOrder = BitOrder +1;
        }
    }
    fclose(fp1);
    fp1 = fopen(espresso_file_name, "r");
    fp2 = fopen("nlfsr.vhd","w");

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity nlfsr is
  port(
    clk : in std_logic;
    reset : in std_logic;
    output : out std_logic
  );
end entity nlfsr;

architecture aa of nlfsr is
  signal x : std_logic_vector (%d downto 0);

begin
  process (clk,reset)
  begin
    if (reset='1') then
      x <= "";
    for (j=0 ; j<=(BitOrder -1) ; j++)
      fprintf(fp2,"1");
    fprintf(fp2,""");
    elsif (clk'event and clk='1') then
      fprintf(fp2,""");
    if (ch == *x2find[0])
      fputs(x2repl[0],fp2);
    else if (ch == *x2find[1])
      fputs(x2repl[1],fp2);
    else if (ch == *x2find[2])
      fputs(x2repl[2],fp2);
    else if (ch == *x2find[3])
      fputs(x2repl[3],fp2);
    else if (ch == *x2find[4])
      fputs(x2repl[4],fp2);
    else
      fputc(ch,fp2);
    end if;
    output <= x(0);
  end process;
end architecture aa;

signal clk : std_logic := '1';
signal reset : std_logic := '1';

fp3 = fopen("nlfsr_tb.vhd","w");

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity nlfsr_tb is
end nlfsr_tb;

architecture aa of nlfsr_tb is
  component nlfsr
    port(
      clk : in std_logic;
      reset : in std_logic;
      output : out std_logic
    );
  end component;

signal clk : std_logic := '1';
signal reset : std_logic := '1';

while(EOF!=(ch =fgetc(fp1)))
{//while(!feof(fp1))
  if (ch == *x2find[0])
    fputs(x2repl[0],fp2);
  else if (ch == *x2find[1])
    fputs(x2repl[1],fp2);
  else if (ch == *x2find[2])
    fputs(x2repl[2],fp2);
  else if (ch == *x2find[3])
    fputs(x2repl[3],fp2);
  else if (ch == *x2find[4])
    fputs(x2repl[4],fp2);
  else
    fputc(ch,fp2);
}

fprintf(fp2,"\n\noutput <= x(0);\nend if;\nend process;\nend architecture aa;\n")

fprintf(fp3,"library ieee;\nuse ieee.std_logic_1164.all;\nuse ieee.std_logic_arith.all;\nuse ieee.std_logic_unsigned.all;\n\nentity nlfsr_tb is\nend nlfsr_tb;\narchitectures aa of nlfsr_tb is\n\ncomponent nlfsr\n  port(
    clk : in std_logic;\n    reset : in std_logic;\n    output : out std_logic\n  );\nend component;\n\nsignal clk : std_logic := '1';\nsignal reset : std_logic := '1';\n\nfp3 = fopen("nlfsr_tb.vhd","w");

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity nlfsr_tb is
end nlfsr_tb;

architecture aa of nlfsr_tb is
  component nlfsr
    port(
      clk : in std_logic;
      reset : in std_logic;
      output : out std_logic
    );
  end component;

signal clk : std_logic := '1';
signal reset : std_logic := '1';

while(EOF!=(ch =fgetc(fp1)))
{//while(!feof(fp1))
  if (ch == *x2find[0])
    fputs(x2repl[0],fp2);
  else if (ch == *x2find[1])
    fputs(x2repl[1],fp2);
  else if (ch == *x2find[2])
    fputs(x2repl[2],fp2);
  else if (ch == *x2find[3])
    fputs(x2repl[3],fp2);
  else if (ch == *x2find[4])
    fputs(x2repl[4],fp2);
  else
    fputc(ch,fp2);
}

fprintf(fp2,"\n\noutput <= x(0);\nend if;\nend process;\nend architecture aa;\n")

fprintf(fp3,"library ieee;\nuse ieee.std_logic_1164.all;\nuse ieee.std_logic_arith.all;\nuse ieee.std_logic_unsigned.all;\n\nentity nlfsr_tb is\nend nlfsr_tb;\narchitectures aa of nlfsr_tb is\n\ncomponent nlfsr\n  port(
    clk : in std_logic;\n    reset : in std_logic;\n    output : out std_logic\n  );\nend component;\n\nsignal clk : std_logic := '1';\nsignal reset : std_logic := '1';\n\nfp3 = fopen("nlfsr_tb.vhd","w");
fprintf(fp3,"signal clk_en : std_logic := '1';
signal output : std_logic;
begin
  uut: nlfsr port map(clk, reset, output);
  process
  begin
    reset <= '0' after 4 ns;
    wait;
  end process;
end process;
begin
  process (clk_en,clk)
  begin
    if (clk_en = '1') then
      clk <= not clk after 2 ns;
    end if;
  end process;
end architecture aa;

fclose(fp1);
close(fp2);
close(fp3);
printf ("VHDL code with the testbench have been created

return 0;
EOF

#============executing C-code to produce nlfsr.vhd and nlfsr_tb.vhd============#
gcc C_code.c -o C_code
chmod 755 C_code
./C_code
Appendix D- Combining of ASIC design Commands with the Converter

```vhdl
#////////////////////////////////////// VHDL Creating

#==============================================================Check the existing file==============================================================#
FILE=input.pla
if [-f $FILE ];
 then
 echo "File $FILE exists"
 ./espresso -o eqntott input.pla > out.pla
else
 echo "File $FILE does not exists. Please, change your input file to:'input.pla'"
fi

#==============================================================Creating C_code.c File==============================================================#
cat > C_code.c << EOF
#include <stdio.h>
#include <string.h>
#include<stdlib.h>
int main()
{
char *x2find[]= {"f", "=", "!", "&", "|"};
char *x2rep[] = {"x", "=<", " not ", " and ", "or"};
char *x[999];
char ch;
char espresso_file_name[255];
int BitOrder = 0;
int i;
int j;
FILE *fp1,*fp2, *fp3;
//printf("Please enter an espresso file name (with extention): ");
//scanf("%s", espresso_file_name);
fp1 = fopen(espresso_file_name, "r");
if (fp1 == NULL)
{
 printf("\n File does not exist,please check file name!\n\n\n\n\n\nexit (1);
}
 else
{
 printf("OK, File exist now!\n\n\n\n\n\nexit (1);
}
 else
{
 printf("\n File does not exist,please check file name!\n\n\n\n\n\nexit (1);

 while(EOF!=(ch =fgetc(fp1)))
 { //while(!feof(fp1))
     { //ch = fgetc(fp1);
         if (ch == *x2find[1]) // search "="
         {
             BitOrder = BitOrder +1;
         }
     }
     fclose(fp1);
 }
//fp1 = fopen(espresso_file_name, "r");
```
fp1 = fopen("out.pla", "r");
fp2 = fopen("nlfsr.vhd", "w");

fprintf(fp2,"library ieee; use ieee.std_logic_1164.all; use ieee.std_logic_arith.all; use ieee.std_logic_unsigned.all; 

entity nlfsr is 
  port 
    clk : in std_logic; 
    reset : in std_logic; 
    output : out std_logic;
end entity nlfsr;

architecture aa of nlfsr is
    signal x : std_logic_vector (%d downto 0);

begin
    process (clk,reset)
    begin
        if (reset='1') then
            x <= "1";
        elsif (clk'event and clk='1') then
            if (ch == *x2find[0])
                fputs(x2repl[0],fp2);
            else if (ch == *x2find[1])
                fputs(x2repl[1],fp2);
            else if (ch == *x2find[2])
                fputs(x2repl[2],fp2);
            else if (ch == *x2find[3])
                fputs(x2repl[3],fp2);
            else if (ch == *x2find[4])
                fputs(x2repl[4],fp2);
            else
                fputc(ch,fp2);
            end if;
        end if;
    end process;
end architecture aa;

fp3 = fopen("nlfsr_tb.vhd", "w");

fprintf(fp3,"library ieee; use ieee.std_logic_1164.all; use ieee.std_logic_arith.all; use ieee.std_logic_unsigned.all; 

entity nlfsr_tb is 
end entity nlfsr_tb;

architecture aa of nlfsr_tb is
    component nlfsr 
      port 
        clk : in std_logic; 
        reset : in std_logic; 
        output : out std_logic;
    end component;

while(EOF!=(ch =fgetc(fp1)))
    if (ch == *x2find[0])
        fputs(x2repl[0],fp2);
    else if (ch == *x2find[1])
        fputs(x2repl[1],fp2);
    else if (ch == *x2find[2])
        fputs(x2repl[2],fp2);
    else if (ch == *x2find[3])
        fputs(x2repl[3],fp2);
    else if (ch == *x2find[4])
        fputs(x2repl[4],fp2);
    else
        fputc(ch,fp2);
    end if;
end while;
fprintf(fp3, "signal clk : std_logic := '1';
signal reset : std_logic := '1';
signal output : std_logic;
begin
    uut: nlfsr port map (clk, reset, output);
    process
        begin
            reset <= '0' after 4 ns;
            wait;
        end process;
    process (clk_en, clk)
        begin
            if (clk_en = '1') then
                clk <= not clk after 2 ns;
            end if;
        end process;
end architecture aa;
fclose(fp1);
fclose(fp2);
fclose(fp3);
printf ("VHDL code with the testbench have been created
"');
return 0;
EOF
#==========executing C-code to produce nlfsr.vhd and nlfsr_tb.vhd==========#
gcc C_code.c -o C_code
chmod 755 C_code
./C_code
#////////////////////////////////////// ASIC Calculations
mkdir SYNOPS
mkdir SYNOPS/DDC SYNOPS/NELIST SYNOPS/REPORTS SYNOPS/SAIF SYNOPS/SOURCE SYNOPS/WORK
#========================= Creating Directories ============#
#!/bin/bash
cat > .synopsys_dc.setup << EOF
set company "KTH"
set SynopsysHome [getenv "SYNOPSYS"]
set search_path ".";
$SynopsysHome/libraries/syn/
    $SynopsysHome/extra_libraries/standard_cell/Plessey/CLA/libs/
    $SynopsysHome/extra_libraries/standard_cell/NEC/CB10VX/syn/objects/
    $SynopsysHome/extra_libraries/gate_array/ChipExpress/syn/cx4001/
    $SynopsysHome/extra_libraries/gate_array/NEC/CMOSN5/syn/object/
    $SynopsysHome/extra_libraries/fpga/Altera/maxplus/alt_syn/flex10k/lib/
    $SynopsysHome/extra_libraries/fpga/Altera/quartus/dc/syn/apex20ke/lib/
    $SynopsysHome/extra_libraries/fpga/Altera/maxplus/alt_syn/max70000/lib/
    $SynopsysHome/extra_libraries/standard_cell/TSMC/tcbn90g_110a/.
    SRC .SYNOPS/SOURCES .SYNOPS/DDC/
set cache_read "/tmp";
set cache_write "/tmp";
set view_read_file_suffix
    "db sdb edif sedif vhd vhdl st script"
set view_analyze_file_suffix
    "v vhd vhdl"
set template_parameter_style
    "%d"; # Limits the length of comp. names
set link_path \${search_path}
set target_library "tcbn90gtc.db"
dw_foundation.sldb"
set symbol_library "tcbn90g.sdb"
set synthetic_library "standard.sldb
dw_foundation.sldb";
set link_library "* \${target_library}"
define_design_lib WORK -path ./SYNOPS/WORK EOF

#=======================Creating Constraints File ============================#
cat > constraints.tcl << EOF
create_clock -name "clk" -period 4 -waveform {0 2} [clk]
set_clock_uncertainty 0.1 clk
set_clock_latency 0.2 clk
set_clock_transition 0.1 clk
set_dont_touch_network clk

#set_dont_touch reset
#set_driving_cell
- library tcbn90gtc -lib_cell INVD0 -pin ZN [get_ports d]
set_load [load_of tcbn90gtc/INVD0/I] [get_ports output]

#set_input_delay 0.5 [get_ports d] -clock clk
set_output_delay 0.5 [get_ports output] -clock clk

#set_max_area 11000
set_operating_condition -lib tcbn90gtc NCCOM
set_wire_load_model -lib tcbn90gtc -name "TSMC16K_Lowk_Conservative"
set_wire_load_mode top
EOF

#========================Creating Synopsys_sdf.tcl File=========================
cat > Synopsys_sdf.tcl << EOF
analyze -library WORK -format vhdl \{ ./nlfsr.vhd}
elaborate NLFSR -architecture AA -library DEFAULT

#write -hierarchy -format ddc -output ./SYNOPS/DDC/nlfsr_elab.ddc
#Apply some constraints
source ./constraints.tcl
compile -write -hierarchy -format ddc -output ./SYNOPS/DDC/nlfsr_compwithconstr.ddc
report_qor > ./SYNOPS/REPORTS/report_qor_default.txt
report_constraint -all_violators > ./SYNOPS/REPORTS/report_Constraint.txt
report_timing > ./SYNOPS/REPORTS/report_timing_default.txt
report_reference > ./SYNOPS/REPORTS/report_reference_default.txt
report_power -analysis_effort high > ./SYNOPS/REPORTS/report_power_Early.txt
write_sdf ./SYNOPS/SOURCE/nlfsr_default.sdf

#change_names -rule verilog -hierarchy
write_file -format verilog -hierarchy -output ./SYNOPS/NETLIST/nlfsr_netlist.v
exit
# Creating ModelSim do File

```bash
cat > ModelSim.do << EOF
vsim -sdfmax/nlfsr_tb/uut=./SYNOPS/SOURCE/nlfsr_default.sdf work.nlfsr_tb
run 4 ns
vcd file ./SYNOPS/SAIF/myvcdfile.vcd
vcd add -r /*
vcd add -r nlfsr_tb/*
run 16384 ns
vcd checkpoint
quit -f
EOF
```

# Creating Synopsys.tcl File

```bash
cat > Synopsys.tcl << EOF
read_file -format ddc ./SYNOPS/DDC/nlfsr_compwithconstr.ddc
read_saif -input SYNOPS/SAIF/mysaiffile -instance_name nlfsr_tb/uut
report_power
report_power -analysis_effort high > ./SYNOPS/REPORTS/report_power_Final.txt
exit
EOF
```

# Executing Section

```bash
dc_shell -f Synopsys_sdf.tcl
```

# Convert VCD to SAIF

```bash
vcd2saif -i ./SYNOPS/SAIF/myvcdfile.vcd -o ./SYNOPS/SAIF/mysaiffile
```

# Invoke Synopsys tool

```bash
dc_shell -f Synopsys.tcl
```
Appendix E- Spiee Model of LFSR Power Diagram

```bash
#ASIC Calculations

# Creating Directories

mkdir SYNOPS
mkdir SYNOPS/DDC SYNOPS/NETLIST SYNOPS/REPORTS SYNOPS/SAIF SYNOPS/SOURCE SYNOPS/WORK

# Creating .synopsys_dc.setup File

#!/bin/bash
cat > .synopsys_dc.setup << EOF
set company "KTH"
set SynopsysHome [getenv "SYNOPSYS"]
set search_path ".
 ${SynopsysHome}/libraries/syn
 ${SynopsysHome}/../extra_libraries/standard_cell/Plessey/CLA/libs/
 ${SynopsysHome}/../extra_libraries/standard_cell/NEC/CB10VX/syn/objects/
 ${SynopsysHome}/../extra_libraries/gate_array/ChipExpress/syn/cx4001/
 ${SynopsysHome}/../extra_libraries/gate_array/NEC/CMOSNS/syn/object/
 ${SynopsysHome}/../extra_libraries/gate_array/NEC/CMOSN5/syn/object/
 ${SynopsysHome}/../extra_libraries/fpga/Altera/maxplus/alt_syn/flex10k/lib/
 ${SynopsysHome}/../extra_libraries/fpga/Altera/quartus/dc/syn/apex20ke/lib/
 ${SynopsysHome}/../extra_libraries/fpga/Altera/maxplus/alt_syn/max7000/lib/
 ${SynopsysHome}/../extra_libraries/standard_cell/TSMC/tcbn90g_110a/Front_End/timing_power/tcbn90g_110a/
 ./SRC ./SYNOPS/SOURCE ./SYNOPS/DDC/
 set cache_read "/tmp";
 set cache_write "/tmp";
 set target_library "tcbn90gtc.db"
dw_foundation.sldb"
 set symbol_library "tcbn90g.sdb"
 set synthetic_library "standard.sldb dw_foundation.sldb";
 set link_library "* "${target_library}"
 define_design_lib WORK -path ./SYNOPS/WORK

EOF

# Creating Constraints File

create_clock -name "clk" -period 4 -waveform {0 2} {clk}
set_clock_uncertainty 0.1 clk
set_clock_latency 0.2 clk
set_clock_transition 0.1 clk
set_dont_touch_network clk

#set_dont_touch reset
```
#set_driving_cell -library tcbn90gtc -lib_cell INVD0 -pin ZN [get_ports d]
set_load [load_of tcbn90gtc/INVD0/1] [get_ports output]
#set_input_delay 0.5 [get_ports d] -clock clk
set_output_delay 0.5 [get_ports output] -clock clk
#set_max_area 11000
set_operating_condition -lib tcbn90gtc NCCOM
set_wire_load_model -lib tcbn90gtc -name "TSMC16K_Lowk_Conservative"
set_wire_load_mode top
EOF

#========================Creating Synopsys sdf.tcl File=======================#
cat > Synopsys_sdf.tcl << EOF
analyze -library WORK -format vhdl { ./nlfsr.vhd}
elaborate NLFSR -architecture AA -library DEFAULT
#write -hierarchy -format ddc -output ./SYNOPS/DDC/nlfsr_elab.ddc
#Apply some constraints
source ./constraints.tcl
compile
write -hierarchy -format ddc -output ./SYNOPS/DDC/nlfsr_compwithconstr.ddc
report_qor > ./SYNOPS/REPORTS/report_qor_default.txt
report_constraint -all_violators > ./SYNOPS/REPORTS/report_Constraint.txt
report_timing > ./SYNOPS/REPORTS/report_timing_default.txt
report_area > ./SYNOPS/REPORTS/report_area_default.txt
report_reference > ./SYNOPS/REPORTS/report_reference_default.txt
report_power -analysis_effort high > ./SYNOPS/REPORTS/report_power_Early.txt
write_sdf ./SYNOPS/SOURCE/nlfsr_default.sdf
#change_names -rule verilog -hierarchy
write_file -format verilog -hierarchy -output ./SYNOPS/NETLIST/nlfsr_netlist.v
exit
EOF

#=========================Creating ModelSim do File ==========================#
cat > ModelSim.do << EOF
vsim -sdfmax /nlfsr_tb/uut=./SYNOPS/SOURCE/nlfsr_default.sdf work.nlfsr_tb
run 4 ns
vcd file ./SYNOPS/SAIF/myvcdfile.vcd
#vcd add -r /*
vcd add -r nlfsr_tb/*
run 16384 ns
vcd checkpoint
quit -f
EOF

#===========================Creating Synopsys.tcl File==========================#
cat > Synopsys.tcl << EOF
read_file -format ddc ./SYNOPS/DDC/nlfsr_compwithconstr.ddc
read_saif -input SYNOPS/SAIF/mysaiffile -instance_name nlfsr_tb/uut
report_power
report_power -analysis_effort high > ./SYNOPS/REPORTS/report_power_Final.txt
exit
# run Synopsys to get the sdf and netlist files with some reports

dc_shell -f Synopsys_sdf.tcl

#=================Invoke Modelsim to generate netlist & SAIF=================
mkdir SIM
vlib SIM/WORK
vmap work SIM/WORK
vlog SYNOPS/NETLIST/nlfsr_netlist.v
 work ./SIM/WORK
vcom ./nlfsr_tb.vhd
 work ./SIM/WORK

#compile the get level netlist and the testbench with this verilog file
vlog
 /afs/it.kth.se/pkg/synopsys/extra_libraries/standard_cell/TSMC/tcbn90g_110a/Front_End/verilog/tcbn90g_110a/tcbn90g.v
 work ./SIM/WORK

vsim -c work.nlfsr_tb
do ModelSim.do

#=============================convert VCD to SAIF=============================#
vcd2saif -i ./SYNOPS/SAIF/myvcdfile.vcd
 o ./SYNOPS/SAIF/mysaiffile$1

#============================Invoke Synopsys tool==============================#
dc_shell -f Synopsys.tcl

#//////////////////////////////Calculating Power_Per_Cycle

#=====================Generating source file that used by the dc_shell===================#
cat > Synopsys_PowerPerCycle.tcl << EOF
 set r 4
 while { $r <= 50 } {
 echo "r is: [expr $r]"
 read_file -format ddc ./SYNOPS/DDC/nlfsr_compwithconstr.ddc
 read_saif -input SYNOPS/SAIF/mysaiffile$1 -instance_name nlfsr_tb/uut
 compile
 report_power
 report_power -analysis_effort low >> ./SYNOPS/REPORTS/report_power_perCycle.txt
 incr r 4
 }
 exit
 EOF

#==================Converting the VCD file to SAIF per Cycle==================#
#!/bin/bash
i=4
j=0
k=4
while [ $i -le 50 ]
do
 j=$((j+4))
 echo "i is $i"
 echo "j is $j"
vcd2saif -i ./SYNOPS/SAIF/mysaiffile$1 -time $i $j
EOF
# Invoke Synopsys tool

current_list=[$1 + 4 ]
dc_shell -f Synopsys_PowerPerCycle.tcl

#!/bin/bash
grep -o "Cell Internal Power =..........." ./SYNOPS/REPORTS/report_power_perCycle.txt >Internal1.log
grep -o ["^Cell Internal Power ="].." Internal1.log > Internal.log
rm Internal1.log

grep -o "Net Switching Power =........" ./SYNOPS/REPORTS/report_power_perCycle.txt >Switching1.log
grep -o ["^Net Switching Power ="].." Switching1.log > Switching.log
rm Switching1.log

grep -o "Total Dynamic Power =........" ./SYNOPS/REPORTS/report_power_perCycle.txt >Dynamic1.log
grep -o ["^Total Dynamic Power ="].." Dynamic1.log > Dynamic.log
rm Dynamic1.log

i=4
while [ $i -le 50 ]
do
  #echo "i is $i"
  echo $i >> timing.log
  i=$(($i + 4 ))
done
paste timing.log Internal.log Switching.log Dynamic.log > ./SYNOPS/REPORTS/For_Matlab.dat
paste timing.log Internal.log Switching.log Dynamic.log > ./SYNOPS/REPORTS/RESULT_TO_MATLAB/For_Matlab.dat
rm Internal.log
rm Switching.log
rm Dynamic.log
rm timing.log

# Creating the Matlab file

cat > ./SYNOPS/REPORTS/RESULT_TO_MATLAB/Plot_Result_in_Matlab.m << EOF
  close all
  load For_Matlab.dat
  plot(For_Matlab(:,1), For_Matlab(:,2), 'g');
  title('Power Analysis', 'FontSize', 14);
  hold on;
  plot(For_Matlab(:,1), For_Matlab(:,3), 'k-');
  hold on;
  plot(For_Matlab(:,1), For_Matlab(:,4), 'r-');
  hold on;
  %plot(For_Matlab(:,1), For_Matlab(:,5));
  %hold on;
  grid;
  xlabel('Time (ns)');
  ylabel('Power (uW)');
  legend('Internal', 'Switching', 'Dynamic', -1);
EOF

cd ./SYNOPS/REPORTS/RESULT_TO_MATLAB
matlab
Appendix F - A modified VHDL of the LFSR-12 to calculate the number of switching.

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity nlfsr is
   port( clk : in std_logic;
       reset : in std_logic;
       output  : out std_logic
   );
end entity nlfsr;

architecture aa of nlfsr is
   signal  x  : std_logic_vector (11 downto 0);
   signal  n  : std_logic_vector (11 downto 0);
begin
   process (clk,reset)
      variable counter : integer := 0;
      variable counter_continue : integer := 0;
      variable count   : integer := 0;
      variable count_per_cycle : integer := 0;
   begin
      if  (reset='1') then
         x <= "111111111111" ;
         n <= "111111111111" ;
      elsif (clk'event and clk='1') then
         x(11) <= x(0) ;
         x(10) <= x(11) xor x(0);  
         x(9)  <= x(10) ;
         x(8)  <= x(9) ;
         x(7)  <= x(8) ;
         x(6)  <= x(7) ;
         x(5)  <= x(6) ;
         x(4)  <= x(5) ;
         x(3)  <= x(4) ;
         x(2)  <= x(3) xor x(0);  
         x(1)  <= x(2) xor x(0);  
         x(0)  <= x(1) xor x(0);
         output <= x(0);
         n(0)  <= x(0);
         n(1)  <= x(1);
         n(2)  <= x(2);
         n(3)  <= x(3);
         n(4)  <= x(4);
         n(5)  <= x(5);
         n(6)  <= x(6);
         n(7)  <= x(7);
         n(8)  <= x(8);
         n(9)  <= x(9);
         n(10) <= x(10);
      end if;
   end process;
end;
n(11) <= x(11);

if (x = "111111111111") then
counter := 0;
end if;

if (x(0) xor n(0)) = '1' then
  count := count + 1;
end if;

if (x(1) xor n(1)) = '1' then
  count := count + 1;
end if;

if (x(2) xor n(2)) = '1' then
  count := count + 1;
end if;

if (x(3) xor n(3)) = '1' then
  count := count + 1;
end if;

if (x(4) xor n(4)) = '1' then
  count := count + 1;
end if;

if (x(5) xor n(5)) = '1' then
  count := count + 1;
end if;

if (x(6) xor n(6)) = '1' then
  count := count + 1;
end if;

if (x(7) xor n(7)) = '1' then
  count := count + 1;
end if;

if (x(8) xor n(8)) = '1' then
  count := count + 1;
end if;

if (x(9) xor n(9)) = '1' then
  count := count + 1;
end if;

if (x(10) xor n(10)) = '1' then
  count := count + 1;
end if;

if (x(11) xor n(11)) = '1' then
  count := count + 1;
end if;

count_per_cycle := count;
counter := counter + count;
counter_continue := counter + count;
count := 0;
end if;
end process;
end architecture aa;
Appendix G- The Script of computation the hardware characteristics in Altera Quartus II

```bash
#!/bin/bash
cat > nlfsr.tcl << EOF
load_package flow
load_package report
# Create the project and overwrite any settings
project_new nlfsr -overwrite

set_global_assignment -name FAMILY CYCLONEII
set_global_assignment -name DEVICE AUTO
set_global_assignment -name DEVICE_FILTER_SPEED_GRADE FASTEST
set_global_assignment -name VHDL_FILE nlfsr.vhd
set_global_assignment -name FMAX_REQUIREMENT "100 MHz"
set_global_assignment -name EDA_SIMULATION_TOOL "ModelSim (VHDL)"
set_global_assignment -name EDA_MAP_ILLEGAL_CHARACTERS ON
set_global_assignment -name EDA_OUTPUT_DATA_FORMAT VHDL
set_global_assignment -name EDA_WRITE_NODES_FOR_POWER_ESTIMATION ALL_NODES
set_global_assignment -name EDA_TEST_BENCH_DESIGN_INSTANCE_NAME uut
set_global_assignment -name EDA_VHDL_ARCH_NAME cont
set_global_assignment -name EDA_NATIVELINK_SIMULATION_TEST_BENCH nlfsr_tb.vhd
set_global_assignment -name EDA_TEST_BENCH_NAME nlfsr_tb.vhd
set_global_assignment -name EDA_TEST_BENCH_FILE nlfsr_tb.vhd
set_global_assignment -name FITTER_EFFORT "STANDARD FIT"
set_global_assignment -name EDA_TEST_BENCH_FILE nlfsr_tb.vhd
# compile the project
execute_flow -compile
project_close
EOF
```

```bash
#!/bin/bash
cat > simulation.do << EOF
vsim -sdfmax /nlfsr_tb/uut=./simulation/modelsim/nlfsr_vhd.sdo work.nlfsr_tb
do ./simulation/modelsim/nlfsr_dump_all_vcd_nodes.tcl
run 16384 ns
quit -f
EOF
```

```
quartus_sh -t nlfsr.tcl
```
vlib simulation/modelsim/altera
vmap altera simulation/modelsim/altera
vcom /afs/it.kth.se/pkg/quartus/9.0sp2/quartus/eda/sim_lib/altera_primitives_components.vhd 
/afs/it.kth.se/pkg/quartus/9.0sp2/quartus/eda/sim_lib/altera_primitives.vhd -work
./simulation/modelsim/altera
vlib simulation/modelsim/cycloneii
vmap cycloneii simulation/modelsim/cycloneii
vcom /afs/it.kth.se/pkg/quartus/9.0sp2/quartus/eda/sim_lib/cycloneii_atoms.vhd
/afs/it.kth.se/pkg/quartus/9.0sp2/quartus/eda/sim_lib/cycloneii_components.vhd -work
./simulation/modelsim/cycloneii
vlib simulation/modelsim/work
vmap work simulation/modelsim/work
vcom ./simulation/modelsim//nlfsr.vho ./nlfsr_tb.vhd -work ./simulation/modelsim/work
vsim -c work.nlfsr_tb -do simulation.do
quartus_pow --read_settings_files=on --write_settings_files=off nlfsr -c nlfsr
References


[8] Shohreh Sherif Mansouri and Elena Dubrova “A Countermeasure Against Power Analysis Attacks for FSR-Based Stream Ciphers”


[16] ModelSim, SE Command Reference, version 5.6d, published: 6/Aug/02


75


[26] OrCAD PSpice User’s Guide

[27] ALTERA, “Introduction to the Quarus II Software”, Version 10.0