# Pseudo Random Number Generation Using Hardware Implementation of Elementary Cellular Automata

by

William A. Schenck

A Thesis Submitted in Partial Fulfillment

for the Requirements for the Degree of

Master of Science in Engineering Technology

Middle Tennessee State University

December 2016

To my parents for their love and support.

ACKNOWLEDGEMENTS

I would like to thank Dr. Karim Salman for his invaluable guidance without which this project would have never happened.

## ABSTRACT

Stephen Wolfram suggested cellular automata may be a good candidate for generating suitable encrypted data. His work recommends that rule thirty was good enough for data encryption. The production of strong data encryption is the goal of anyone who desires the means of transmitting secure messages. Studies worldwide has generated numerous volumes of research. However, these studies are based on the use of computational machinery to produce these results. This project goes beyond the computational method of pseudo-random number generation. Through the use of VHDL, a working pseudo-random number generator can be designed and programmed into a FPGA device. A pseudo-random number generator (based around elementary cellular automation) has been implemented and it was installed into a FPGA device. This working device produced results that matched the computational methods of a similar pseudo-random number generator.

TABLE OF CONTENTS

| | Page |
|---|---|
| LIST OF TABLES…… | vii |
| LIST OF FIGURES…… | viii |
| LIST OF ABBREVIATIONS…… | x |
| INTRODUCTION…… | 1 |
| LITERATURE REVIEW…… | 6 |
| Early History…… | 6 |
| Earlier Research at MTSU…… | 7 |
| THEORY…… | 9 |
| PREPARATION AND DESIGN FLOW…… | 19 |
| GRAPHIC DESIGN IMPLEMENTATION…… | 27 |
| VHDL DESIGN IMPLEMENTATION…… | 35 |
| DATA COLLECTION AND TRIALS…… | 49 |
| FINDINGS AND RESULTS…… | 58 |
| CONCLUSIONS…… | 60 |
| REFERENCES…… | 61 |

APPENDICES…………………………………………………………………… 62

APPENDIX A: Taps Table for Up to 67 LSFR Bits…………………… 63

APPENDIX B: Pen and Paper Results for 4 and 8 Bits………………… 64

APPENDIX C: Program Flow…………………………………………… 66

APPENDIX D: The 16 Chaotic and Balanced Rules…………………… 68

APPENDIX E: Iterations and Quarters Table…………………………… 69

APPENDIX F: Resources Used Table…………………………………… 70

APPENDIX G: Results for ECA One Rule……………………………… 71

APPENDIX H: Results for Dual Rules…………………………………… 75

LIST OF TABLES

1. Taps Table for Up to 67 LSFR Bits……………………………………… 63

2. The 16 Chaotic and Balanced Rules……………………………………... 68

3. Iterations and Quarters Table……………………………………………. 69

4. Resources Used Table…………………………………………………… 70

5. Results for ECA One Rule………………………………………………  71

6. Dual Rule 31 Bits……………………………………………………….. 75

7. Dual Rule 127 Bits………………………………………………………  75

8. Dual Rule 113 Bits………………………………………………………  76

## LIST OF FIGURES

1. Linear-Shift Feedback Register System………………………………….. 2
2. Encryption/Decryption Message Flow…………………………………… 3
3. Basic Layout and Logic Flow……………………………………………. 4
4. Von Neumann and Moore Neighborhoods………………………………. 9
5. Cell Layout……………………………………………………………….. 10
6. 3-Bit Cellular Automata Basic Layout…………………………………… 11
7. Layout of an ECA System……………………………………………….. 12
8. Elementary Cellular Automation………………………………………… 13
9. Rule 254 Layout…………………………………………………………. 14
10. How Rules 15 and 30 Look……………………………………………… 14
11. Before and After Time Step Example of Rule 15……………………….. 15
12. Simplified Rule Patterns…………………………………………………. 16
13. Homogenous Class Using Rule 160……………………………………… 17
14. Periodic Class Using Rule 108…………………………………………… 17
15. Chaotic Class Using Rule 30…………………………………………….. 17
16. Complex Class Using Rule 110………………………………………….. 18
17. Screen Shot of the CA Tabs Program……………………………………. 21
18. Illustration of an Alternating Bits Layout Using Rules 30 and 45………. 22
19. Illustration of Alternating Rows Layout Using Rules 30 and 45………... 23
20. Simplified DE-2 board…………………………………………………… 25
21. Communication Flow Between DE-2 Board and PC (Simplified)………. 26

22. 3:8 Multiplexer………………………………………………………… 27

23. Simplified 3 Cell Design…………………………………………………. 28

24. Basic Cell Layout Using Rule 30………………………………………… 30

25. Quartus II Basic Cell Layout - Graphic Design File……………………... 31

26. Graphic Design - Four Bit Layout……………………………………...... 32

27. Graphic Design of an Eight Bit System Layout…………………………. 33

28. Simplified CA Bits Unit…………………………………………………. 36

29. Basic NIOS Processor (Simplified)……………………………………… 40

30. Basic NIOS system (Simplified)………………………………………… 41

31. Simplified Diagram of a 32-Bit 2-1 Multiplexer………………………… 43

32. 64-Bit Data Exchange……………………………………………………. 44

33. Simplified View of the Complete System……………………………….. 45

# LIST OF ABBREVIATIONS

| | |
|---|---|
| ASCII | American Standard Code for Information Interchange |
| CA | Cellular Automata |
| DOD | Department of Defense |
| ECA | Elementary Cellular Automata |
| FPGA | Field Programmable Gate Array |
| HDL | Hardware Description Language |
| IEEE | Institute of Electrical and Electronic Engineers |
| LFSR | Linear-Feedback Shift Register |
| LSB | Least Significant Bit |
| MSB | Most Significant Bit |
| MTSU | Middle Tennessee State University |
| USB | Universal Serial Bus |
| VHDL | Very High Speed Integrated Circuit Hardware Description Language |

# INTRODUCTION

Cryptography has been a time honored tradition of anyone trying to convey information (through various codes) to their allies. However, as history has proven, codes can be broken. The English cracked Germany’s ENIGMA code during World War II [1]. The United States also deciphered Japans secret messages [2]. This gave the Allies an advantage because they knew what the Axes were planning.

To encrypt messages require methods that the sender (and intended receiver) uses to convey sensitive information. The main method usually used is the use of pseudo-randomly generated numbers. The major difference between a true random number generator and the pseudo-random number generator is that the true random number generator is non-periodical. While a non-periodical random number generator has a desirable quality for things like lotteries and games, it is completely unusable for cryptology. If a true random number generator encrypted a message, it would be impossible to decrypt the message. In this case, a pseudo-random number generator has to be used. Because pseudo–random numbers are periodic, they are excellent for cryptology. This section will discuss a method used for transmission of secure messages momentarily, but first a method used at Middle Tennessee to generate random numbers.

At Middle Tennessee State University (MTSU), numerous projects are incorporated that require the use of randomly generated numbers. One such project is a dice game that the students have to implement. Usually, the students utilize a linear-feedback shift register system (LFSR) and (for course purposes) this is suitable for generating the random numbers needed for their projects. However, for Cryptography, this is not good enough for producing acceptable pseudo-random numbers (discussed later).

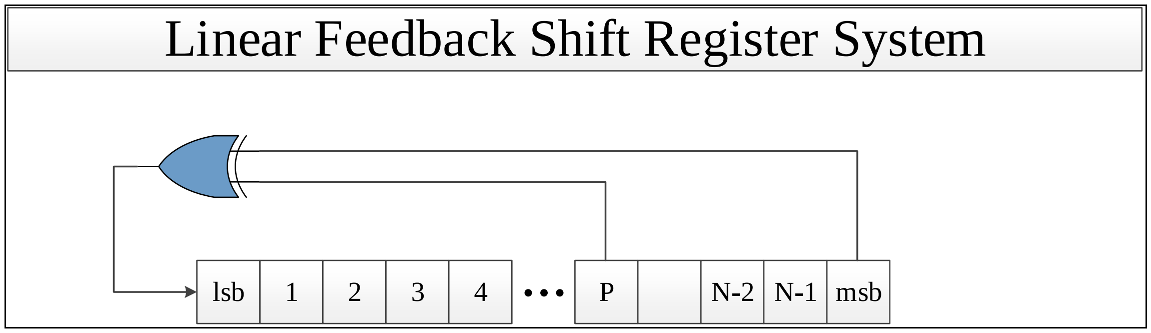


Figure1: Linear-Shift Feedback Register System.

The LFSR, in figure 1, uses an exclusive-or (XOR) gate(s) attached to specially selected output registers. The taps selected usually come from a LFSR taps table. Appendix A provides a small taps table that recommends the best register selection for up to sixty-seven bits [3]. For example, if a 31-bit system is desired, the selected taps would come from registers 31 (the MSB) and 28. The two registers have outputs that is applied to a XOR gate and the output of the XOR gate is determined by the value of these inputs. The output of the XOR gate(s) is fed directly back into the shift register. The individual registers also provide an output that is combined with other registers to produce a binary output. External logic circuitry converts this binary number into a usable number that the system displays to the user. This system is adequate for academia purposes. However, for real world applications (like cryptology), this system is not good enough.

Cryptology involves the communication of data that only the intended receiver can understand. As an example (figure 2), the sender creates a plain text version of the message they want to send to the receiver. The sender encrypts the message using an encryption key.

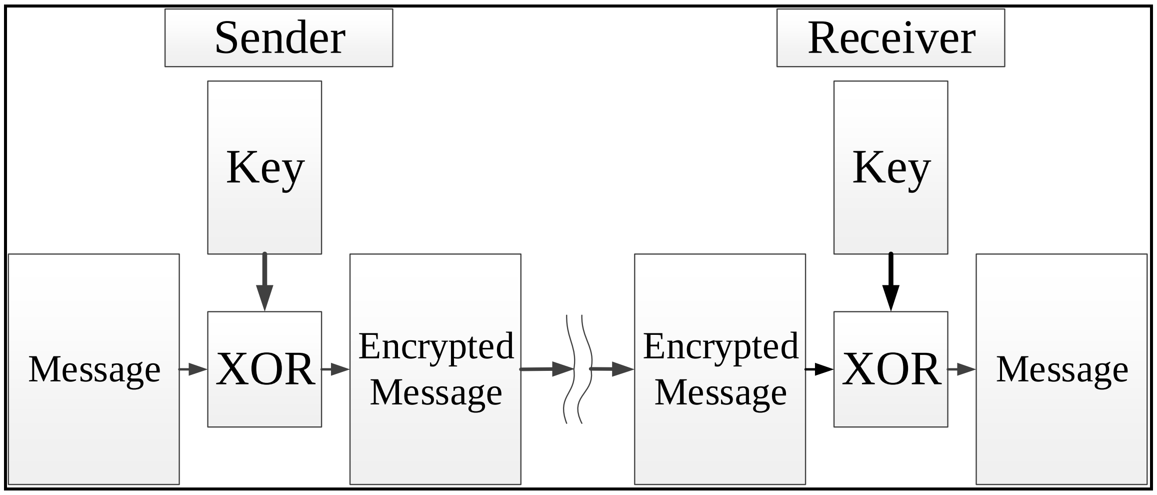


Figure 2: Encryption/Decryption Message Flow.

This message (in encrypted form) is delivered to the receiver. The receiver (through pre-arranged procedures) places the encrypted message through an identical encryption key and decrypts the message that can be read by the receiver. However, if an unauthorized individual somehow manages to intercept the encrypted message, the protection scheme, hopefully, will be difficult enough to discourage any attempts to try and decrypt the message.

There are several methods of producing pseudo-random numbers. While (presently) not used at MTSU, one such method is through the use of cellular automation. Cellular automation, through the usage of various rules, may provide acceptable pseudo-random numbers. Consider having several blocks lined up in a row as displayed in figure 3. The number is not important for this discussion. Like the LSFR, the outputs from these blocks provide the feedback to the other blocks. The inputs are applied to the logic and, through the use of the selected rule that is chosen, these cells will update to their new values.

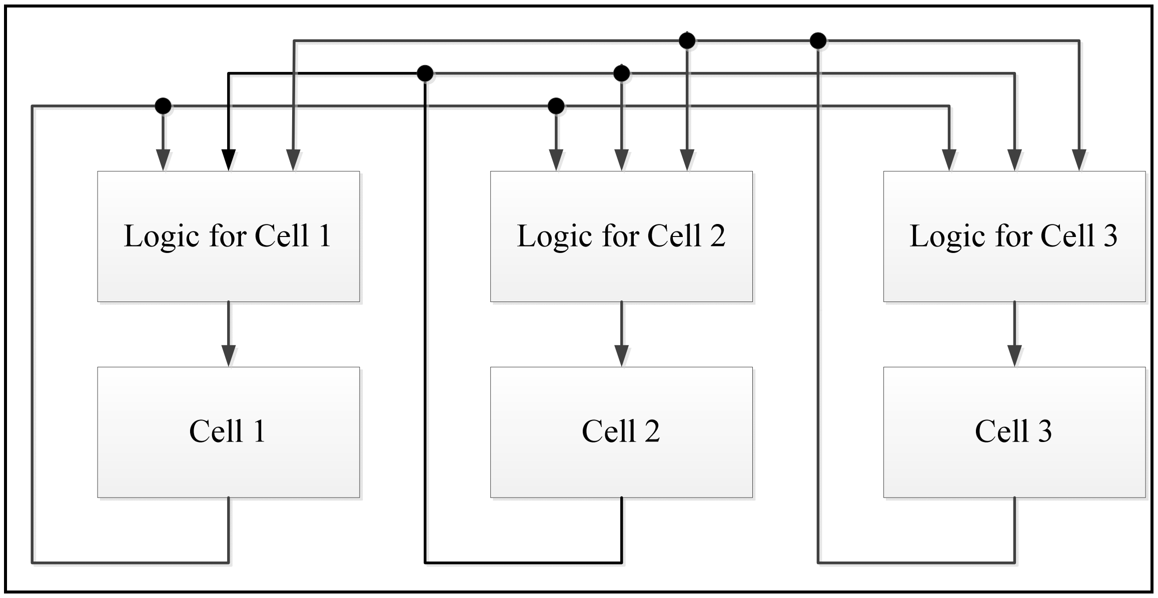


Figure 3: Basic Layout and Logic Flow.

The chosen rule is applied to this data to create the encryption key that is used to generate the cypher text of the encrypted message. Using the three cells in figure 3, as an example, if rule fifteen was selected, and the output of the cells, initially, is 1 0 0 (binary 4), then the output of the next cycle is 1 0 1 (5). The next cycle is 0 0 1 (1). Further progressions give us 0 1 1 (3), 0 1 0 (2), 1 1 0 (6), and back to 1 0 0 (4). However, if we have 0 0 0 (0) or 1 1 1 (7), we cycle between 0 and 7 indefinitely. Not starting off with binary 0 or 7, this 3-bit pseudo-random number generator could possibly be used for the dice game, as mentioned previously, since a six-sided die has only values one through six.

As such, cellular automata may hold promise as a pseudo-random number generator. This research will attempt to develop a suitable pseudo-random number generator, with strong encryption schemes, that can be programmed into a FPGA.

# LITERATURE REVIEW

## Early history

John Von Neumann originally proposes formal models of self-reproducing robots [4]. Suggestion by Stanislaw Ulam, Von Neumann theorizes two-dimensional mesh of finite state machines (called cells) interconnected with each other. Cells outputs change based on the conditions of neighboring cells [5].

Stephen Wolfram continues independent work on cellular automata. Publishes A New Kind of Science in 2002. A New Kind of Science details one and two dimensional CA and attempts to classify rules into four categories [6]. Wolfram suggests that CA rule 30 may be good for data encryption [7].

Further examination of additional past works provide only experiments and results of software generated projects. During this time, there has been nothing mentioned about the actual development of a hardware based system using Field Programmable Gate Array (FPGA) devices until 2006. As a result, the majority of this research had been provided by the earlier works of previous MTSU's students using software based techniques of pseudo-random number generation. Prior research by Asfaw Estub and Christopher Stocker provided initial methods used to determine the success/failure of Cellular Automata as an acceptable pseudo-random number generator.

In 2006, Students from the University of North Texas [8] develop a two-dimensional system. However, this system was developed primarily for Conway's Game of Life with CA using both the Von Newman and Moore neighborhoods. This project is centered on a two-dimensional system. This particular system does not meet the requirements of this project. However, for future projects involving multi-dimensional systems, these findings may prove beneficial.

## Earlier Research at MTSU

In 2007, Asfaw Estub [9] provided the initial research of this project. His reference to Stephan Wolfram's work in the use of cellular automata as a possible pseudo-random number generator was worth pursuing. However, Estub's work presented various boundary conditions for the generation of pseudo-random numbers using rule 30 exclusively. He designed the initial CA_TABS program using Microsoft's VISUAL BASIC©. His program confirmed Wolfram's work that rule 30 is suitable for strong encryption. Estub's conclusion was that further work in this study was needed. But, a one-dimensional cellular automata machine using rule 30 was worth pursuing.

In 2010 Christopher Stocker [10] continued Asfaw Estub's work. Stocker added additional capabilities to Estub's program. Stocker later rewrites CA_TABS using Microsoft's VISUAL C++©. The use of a C++ program increased the operational speed of the CA_TABS program. While the CA_TABS program originally written by Estub limited users to use rule 30 only, Stocker's work added the additional ability of allowing the user the choice of any rule from 1 to 255. Stocker's results found that the added complexities of the additional updates did not negatively affect randomness. But, for our purposes, this work derives the foundation for a one-dimensional cellular automata machine for both software and hardware.

This work is a continuance of Christopher Stocker’s work. Follow on with the design, and implementation, of a one-dimensional cellular automata machine that could be programmed into a FPGA device. This device was used to generate pseudo-random numbers that were compared with a software generated pseudo-random number generator. The hardware generated results had to match the software generated results. If these results matched, then this experiment was successful in developing a working device that could provide pseudo-random numbers that have strong encryption capabilities. The possibilities of generating better encrypted messages will be achieved.

## THEORY

Before the details of this research can be discussed, a proper definition of cellular automata, and how it applies to this project, is needed. One accepted definition is stated by Stephan Wolfram on mathworld - “A cellular automaton is a collection of (colored) cells on a grid of specified shape that evolves through a number of discrete time steps according to a set of rules based on the states of neighboring cells. The rules are then applied iteratively for as many time steps as desired.” [11] Cellular Automation, by this definition, can be very complex. To illustrate, there are two general neighborhoods – Von Neumann and Moore (fig 4).

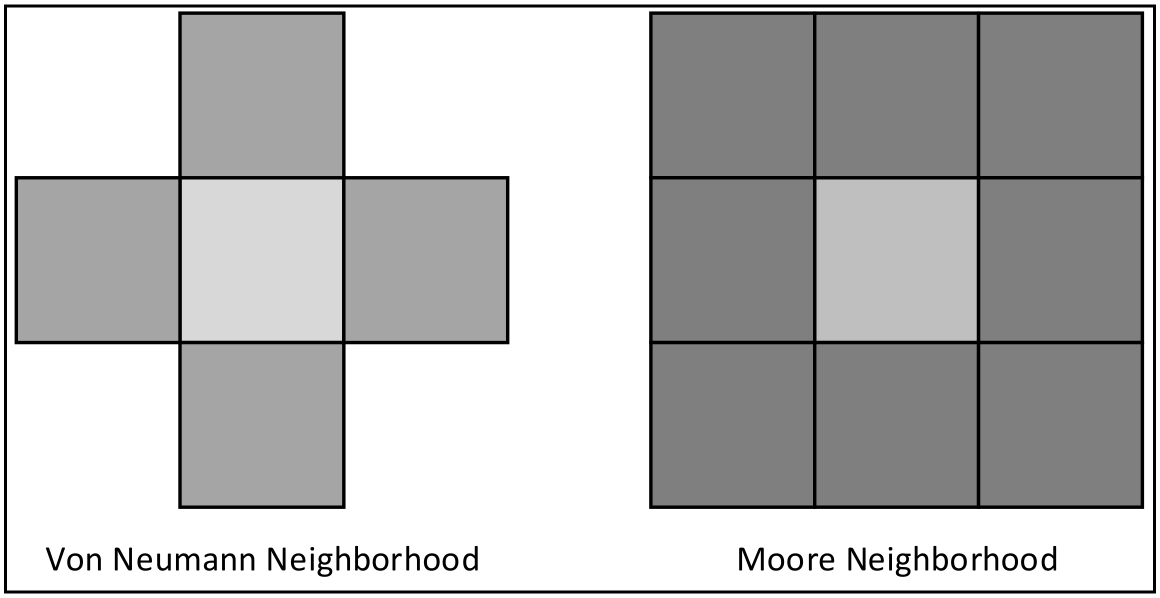


Figure 4: Von Neumann and Moore Neighborhoods

The Von Neumann cell has four neighbors (N, S E and W) plus itself. The Moore cell also has these neighbors and additionally NW, NE, SE, and SW cells (along with itself). For this research, this project will use the elementary cellular automata (ECA).

This is a simple, one-dimensional system. The elementary system holds cells in a singular, one-dimensional array. This is the Von Neumann design only using the E and W neighbors. To maintain consistency throughout this section, additional terminologies and definitions have to be addressed now. The cell is identified by the upper-case C. C is the index number for a finite CA of span $(K \in \mathbb{N},\ 0 \leq C\ \leq K)$. There is a super-script identified with lower-case t along with the sub-script designated with the lower-case letter k. The super-script t represents the time-step where t is at this time. t+1 signifies the next time step and t-1 represents the previous time step. The sub-script k defines the individual cell. For example, if there were six cells, then the cells (from left to right) k values are designated as 5 down to 0 and would look like figure 5.

| | | | | | |
|---|---|---|---|---|---|
| $C_5^t$ | $C_4^t$ | $C_3^t$ | $C_2^t$ | $C_1^t$ | $C_0^t$ |

Figure 5: Cell Layout

There are two requirements for the individual CA cell. First, its output must change to a value determined by the inputs that is compared with an established rule (explained later). Second, the output must maintain this state and only change when the next discrete time step is initiated. Logically, this is done by using a memory device such as a D-type flip-flop or a D-type latch.

The absolute minimum number of cells required for an ECA system is two. This is due to the fact that certain rules (like sixty) only require two cells to determine that bit's next output. However, three cells will be required for the remaining rules (like thirty). In fact, there can be any number of cells to work with and the higher the better. To clarify this statement, consider a three cell system. Three cells give $2^3 = 8$ . This means there are only eight combinations. On a four cell system, this gives $2^4 = 16$ combinations. If there are 5 cells, there are $2^5 = 32$ combinations.

To gain a better understanding of how a cell changes to its' next value, figure 6 outlines a general block diagram of a three cell system. The output of the main cell (C) is applied to one input of a logic unit that is going to determine the next output of this cell. The logic unit also requires additional inputs from Cell C's adjacent neighboring cells (C+1 and C-1). The logic unit acts on the outputs of these three cells and produces a binary number from zero (0 0 0) to seven (1 1 1). This number is compared to the established rule (explained later), and the output of the logic unit is returned to the input to update the cell at the next time step.

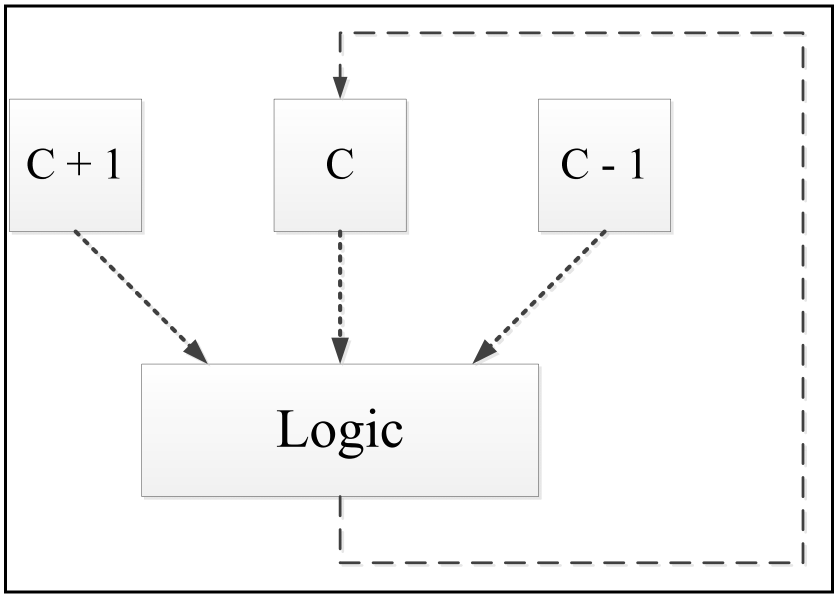


Figure 6: 3-Bit Cellular Automata Basic Layout

To see how the bits are arranged in a one-dimensional relationship, figure 7 details the layout of an ECA system.

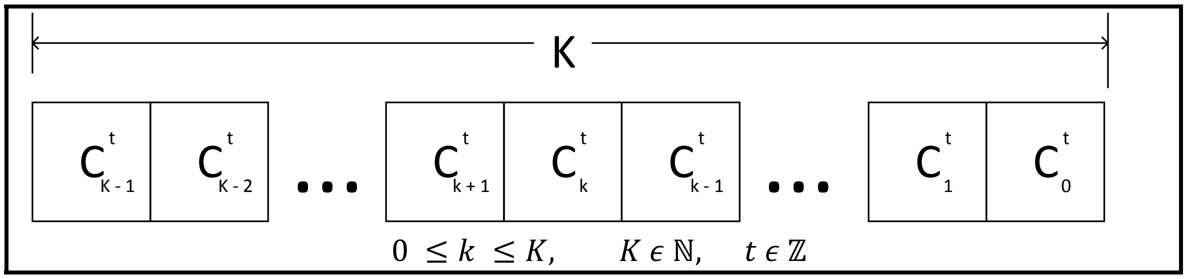


Fig 7: Layout of an ECA System

The number of bits involved is represented by the upper-case K. This is the total bits used for the experiments. Usually this will be a prime number. Extensive testing revealed that composite numbers (even or odd) tend to cause occurrences of short cycles. Because short cycles are undesirable for cryptology, prime numbers were chosen to avoid composite numbers.

Earlier, the minimum cell size stated was two (however, three is used to accommodate all of the rules) and the ceiling is theoretically infinite in number. But, the real limitation imposed on the maximum size is going to be governed by the hardware itself. Using an example of a system that is 32 bits in length, Figure 8 shows how additional cells are added to increase the possible number of outputs. The maximum possible outputs from a three-cell system return eight different values ($2^3 = 8$). The maximum possible outputs from a system that is 32 cells provide over 4 billion ($2^{32} = 4{,}294{,}976{,}296$) possible values. The maximum cell length is fixed by the number K. The individual bit (k) has two immediate neighbors (identified as k+1 and k-1). The output of this bit (k) is determined by the selected rule chosen and the binary number created by the bit patterns of the three cells (k+1, k, k-1). Every cell is built identically so all 32-bits will be interconnected with their neighboring cells. The exceptions are the two end cells. Cell K's cell-next input (k+1) must come from cell 0. Cell 0's cell-previous input (k-1) must come from cell K - 1. This interconnection scheme maintains a circular looping group of cells.

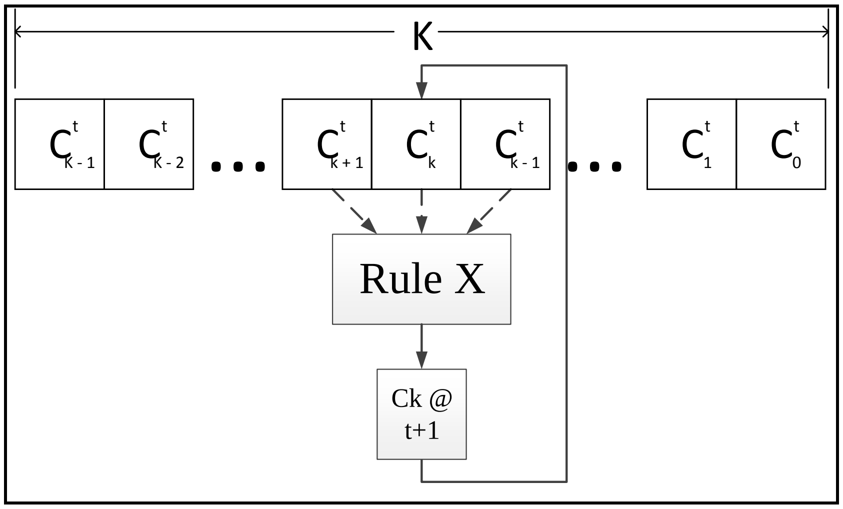


Figure 8: Elementary Cellular Automation

Consideration of a two, or three, dimensional system raises the complexity of these systems and, therefore, makes them non-elementary systems. However, for our purposes, these designs will go beyond the scope of what we are initially attempting. Future work along these lines are planned if this experiment proves successful.

A rule is a verbal representation of the desired output by means of a binary code. In Stephan Wolfram's work (A New Kind of Science), he uses several examples in chapter two to define how an output is generated. His example (using rule 254) in verbal form is: "A cell should be black in all cases where it or either of its neighbors were black on the step before." [Wolfram. P. 24] [6]. Figure 9 shows how rule 254 is laid out (in binary form) along with the accompanying diagram (taken from Wolfram p. 24) [6] that illustrates how the output will look after five iterations.

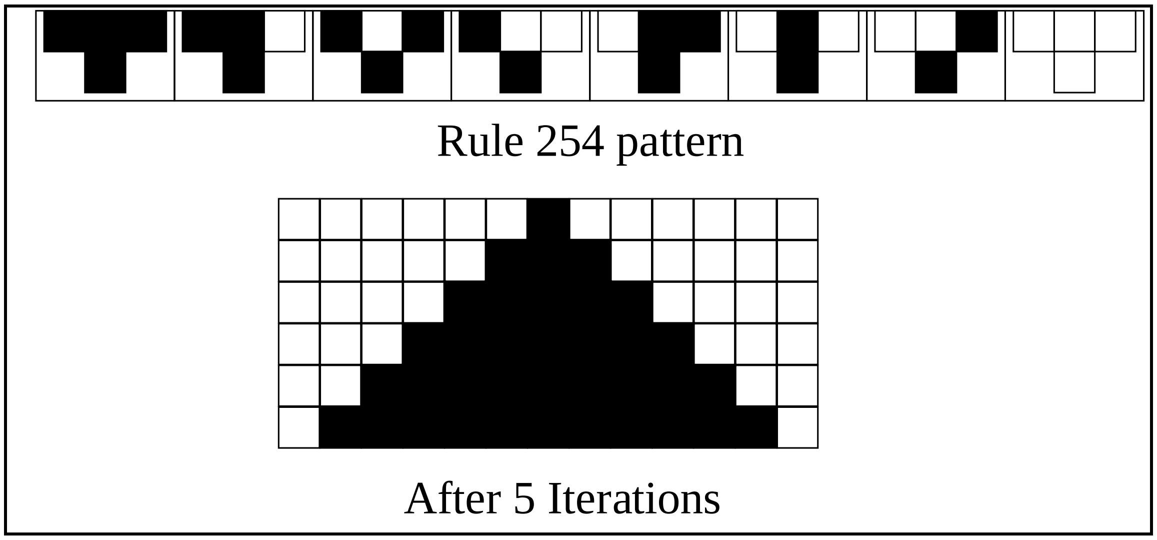
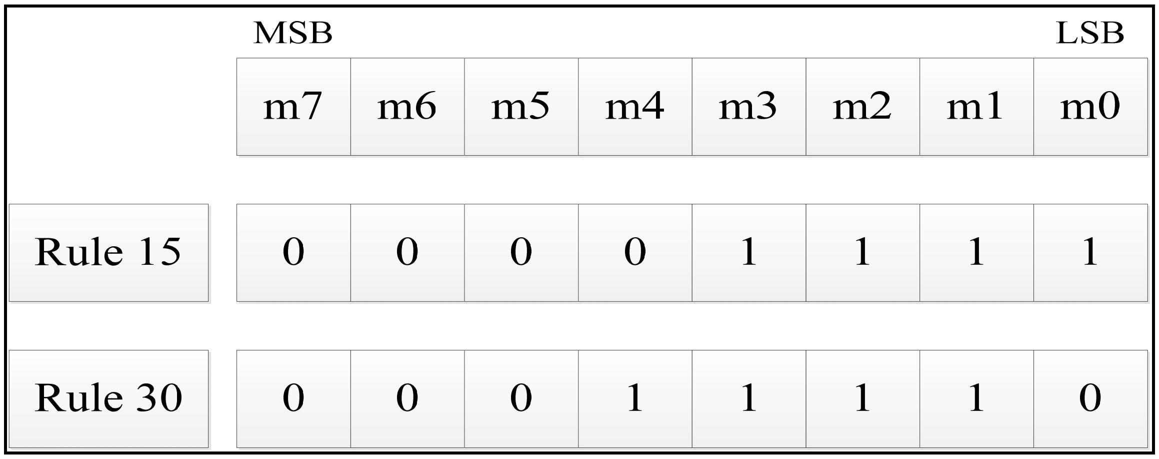


Figure 9: Rule 254 Layout

The binary eight-bit pattern of rule 254 equates to 1 1 1 1 1 1 1 0. By looking at figure 10, two examples of rules 15 and 30 are displayed.



| | MSB | | | | | | | LSB |
|---|---|---|---|---|---|---|---|---|
| | m7 | m6 | m5 | m4 | m3 | m2 | m1 | m0 |
| Rule 15 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| Rule 30 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 |

Figure 10: How Rules 15 and 30 Look

The min-terms (identified by the m-number in fig 10) also have a binary representation. These binary terms range from m0 (0 0 0) to m7 (1 1 1) and are the inputs that the rule requires to generate a new output that is returned to the cell. Using the example provided in the introduction, rule fifteen is chosen. This specifies the bit selection to be three (0 1 1), two (0, 1, 0), one (0, 0, 1), and zero (0, 0, 0) (Figure 11). If the initial condition of these three cells are C + 1 = 1, C = 0, C – 1 = 0 (Figure 11), then a binary value of four (as viewed by cell C) is generated. Figure 11 shows a before and after representation (along with rule 15) as viewed by the individual cells. Cell C+1's view would be a (binary) two and cell C-1's view would be a (binary) one. Figure 10's table reveals cell C's new output from m4 = 0 and cell C remains zero. Cell C+1 output (m2) is still one and C+1 remains one. However, cell C-1's output (m1) will update the output of this cell from zero to one. After the next time step, the updated output from all three cells would be C + 1 = 1, C = 0, C - 1 = 1 (binary 5) as displayed in figure 11.

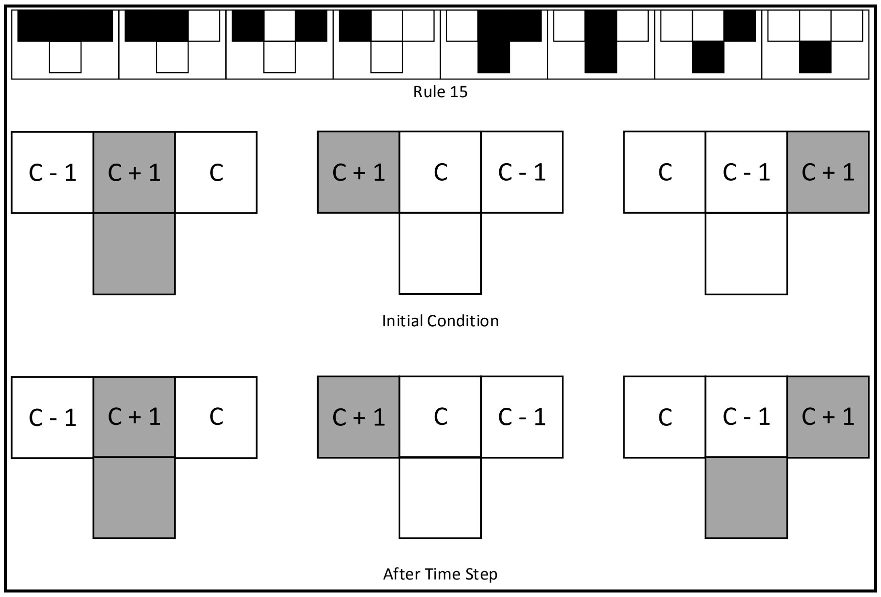
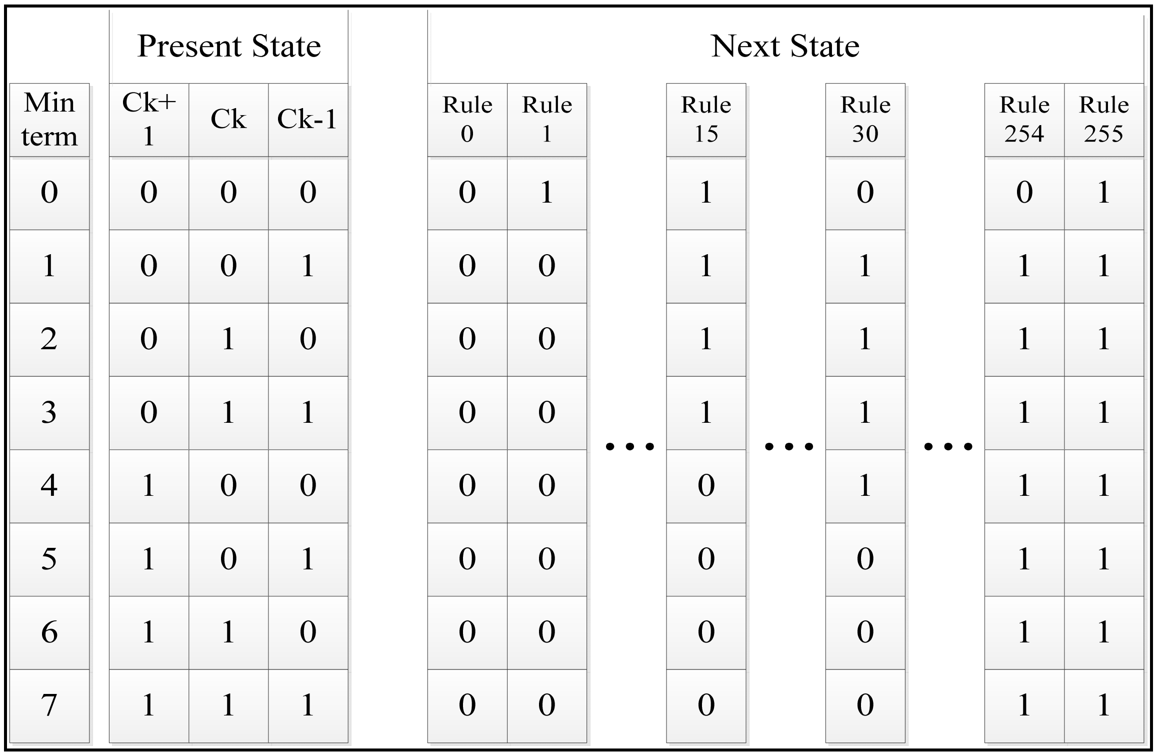


Fig 11: Before and After Time Step Example of Rule 15.

The ECA works on one of two hundred and fifty-six rules. This number comes from two facts. First, there are only two possible outputs (zero or one). Second, there have to be three cells that have only two outputs ($2^3$). This produces a result of ($2^{2^3} = 2^8 = 256$). Figure 12 shows how the individual rules, along with the cells, provide these outputs from the logic unit. The present state from cells C+1, C, and C-1 produces a binary number from zero to seven. The next states output is determined by the binary input and compared with the rule, and this result is returned to the input of cell C.



| | Present State | | | Next State | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Min term | Ck+1 | Ck | Ck-1 | Rule 0 | Rule 1 | | Rule 15 | | Rule 30 | | Rule 254 | Rule 255 |
| 0 | 0 | 0 | 0 | 0 | 1 | | 1 | | 0 | | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 | 0 | | 1 | | 1 | | 1 | 1 |
| 2 | 0 | 1 | 0 | 0 | 0 | | 1 | | 1 | | 1 | 1 |
| 3 | 0 | 1 | 1 | 0 | 0 | … | 1 | … | 1 | … | 1 | 1 |
| 4 | 1 | 0 | 0 | 0 | 0 | | 0 | | 1 | | 1 | 1 |
| 5 | 1 | 0 | 1 | 0 | 0 | | 0 | | 0 | | 1 | 1 |
| 6 | 1 | 1 | 0 | 0 | 0 | | 0 | | 0 | | 1 | 1 |
| 7 | 1 | 1 | 1 | 0 | 0 | | 0 | | 0 | | 1 | 1 |

Figure 12: Simplified Rule Patterns

Finally, Stephan Wolfram, through his research, found that the ECA rules (0 – 255) produced results that fell into one of four classes [12].

Class 1: (rules 0, 32, 160, and 250 are some examples) called the homogenous class, produces results that eventually die. Figure 13 provides a display of a homogenous class. The starting number eventually decays to zero and remains that way.

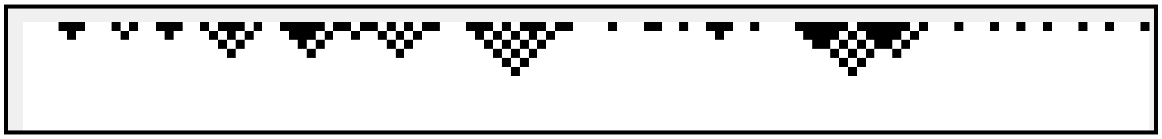


Figure 13: Homogenous Class Using Rule 160

Class 2: (rules 4, 108, 218, and 232 are some examples) called the periodic class, produces results that, after some initial value, will repeat as a pattern in space (horizontally), in time (vertically), or both. Figure 14 shows an example of a periodic class result. The pattern repeats indefinitely.

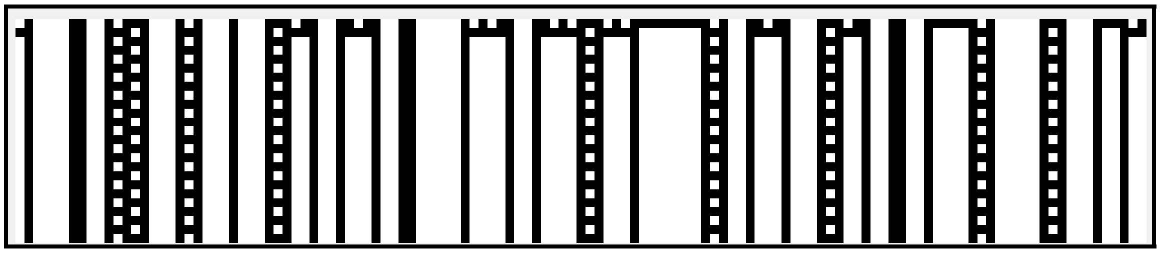


Figure 14: Periodic Class Using Rule 108

Class 3: (rules 30, 60, 90, and 150 are some examples) called the chaotic class, produce results in a chaotic pattern. Figure 15 displays an example of rule 30.

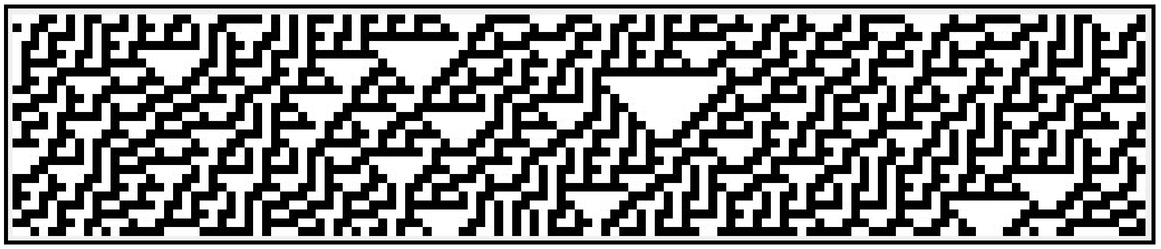


Figure 15: Chaotic Class Using Rule 30

Class 4: (rule 110 for example) called the complex class, produces patterns that will grow in a complicated way. Rule 110 is picture below (figure 16).

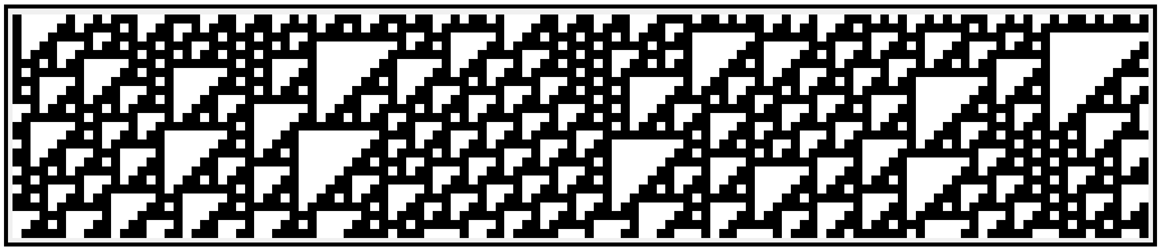


Figure 16: Complex Class Using Rule 110

In the introduction, the Linear-Feedback Shift Register system (LFSR) was used as an example. It was described as a poor candidate for pseudo-random number generation (for encryption). The reason this system is not suitable is that the LFSR (for example with 31 bits in length) can be expanded to twice its length (of 62 bits) and, algorithms like the Berlekamp-Massey, can determine which taps are in use. This knowledge gives unauthorized individuals the ability to decrypt and read the message. [13]

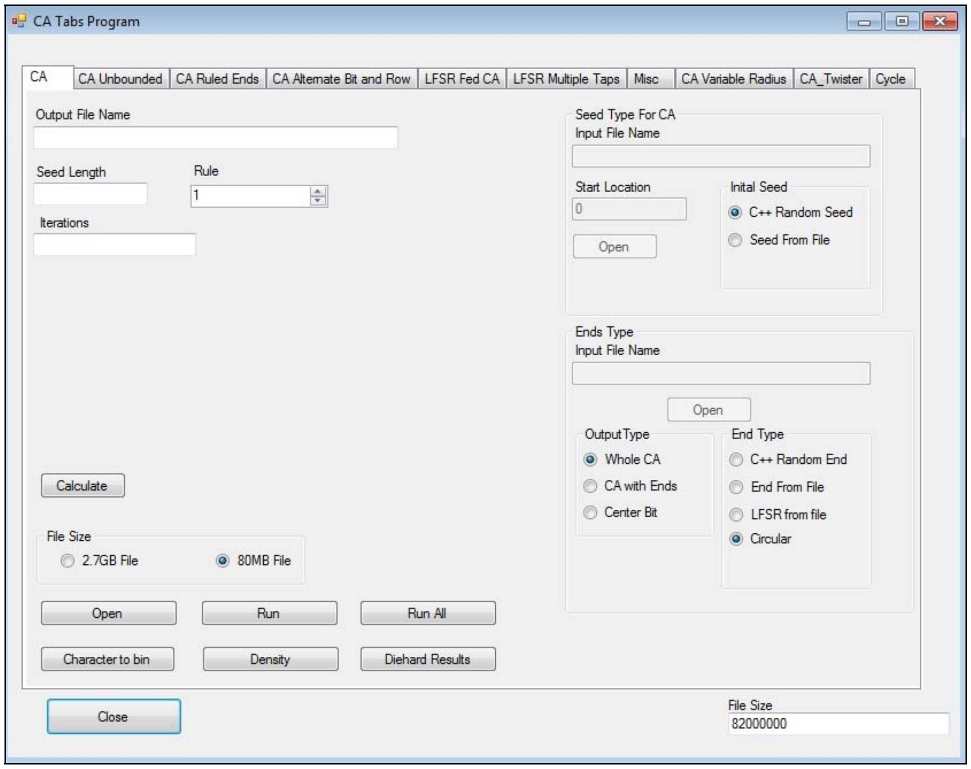
Previous studies revealed that the third class (chaotic) is the best choice for producing pseudo-random numbers that are effective enough for strong encryption. This experiment will attempt to create, in hardware (through the use of a FPGA device), a system that uses class three rules (of varying sizes) to generate suitable pseudo-random numbers with strong encryption.

# PREPARATION AND DESIGN FLOW

The development of a working prototype of a cellular automata device required a thorough understanding of the previous research conducted by Asfaw Estub [9] and Christopher Stocker [10]. Their development of the various CA_TABS programs allowed them to obtain valuable data using elementary cellular automata. CA_TABS (version 12) is a program created by MTSU Graduate Students for testing pseudo-random numbers through the use of elementary cellular automata. CA_TABS had shown that strong encryption could be developed with various chaotic rules (like rule 30 for instance). What had to be done was to determine if a working pseudo-random number generator could be developed that met the criteria of strong encryption in hardware. This meant implementing a working device that could generate pseudo-random numbers. How this was to be done had to satisfy the question - How would success be defined? For success, the results of this device had to match the results of an identical run created by the CA_TABS program.

In order to match the results of the CA_TABS program, a brief discussion of how the CA_TABS program works is necessary. CA_TABS works in one, two, or three phases. Initially, the user creates a filename that CA_TABS will create when it begins generating the pseudo-random numbers. The first phase starts with the generation of pseudo-random numbers. Once this process is complete, the second phase begins by converting this ASCII text file to a binary file. After the binary file is created, the third and final phase begins the diehard battery of tests. The diehard battery of tests is a series of tests developed by George Marsagila at Florida State University [14]. The results of these tests (if any) are saved to a file for review by the user.

When CA_TABS begins, a window is displayed for the user (figure 17). The main window holds ten tabs, one of which the user selects, for the desired test to perform. The first tab is the default tab and is the main “work horse” of the CA_TABS program. The two main tabs this research would use is tabs one (CA) and four (CA Alternate Bit and Row). The first tab provides the location for the user to enter the filename that CA_TABS will store the results to. The user also provides the seed length and rule to use. Normally, file sizes of 80MB is used so a convenient method to generate the proper number of iterations is provided. The 80MB file size is necessary because the Diehard Battery of Tests program requires a binary data file 10MB in length minimum. The pseudo-random number generators create the data of ones and zeros, but it is stored in a text file. The text file stores these values as ASCII ones and zeros which are eight bits in length for each character.



Output File Name
Seed Length
Rule: 1
Iterations
File Size: ( ) 2.7GB File [✓ 80MB File]
Seed Type For CA
Input File Name
Start Location: 0
Inital Seed: [✓ C++ Random Seed] ( ) Seed From File
Ends Type
Input File Name
Output Type: [✓ Whole CA] ( ) CA with Ends ( ) Center Bit
End Type: ( ) C++ Random End ( ) End From File ( ) LFSR from file [✓ Circular]
File Size: 82000000

Figure 17: Screen Shot of the CA Tabs Program

To ensure enough data is generated for the binary file, the 10MB binary data file is multiplied by eight. The upper-right quadrant (Figure 17) gives the option to choose a random seed number, or let the user provide a unique seed value. For this experiment, the seed is unique in that the MSB is set to one while the remaining bits are set to zero. The lower-right quadrant would not be used for this experiment and the values remain in their default position. The lower-left quadrant provides what functions are desired. The run button just creates a text file (the first phase only) with the parameters specified by the user. Usually the text file is an 80MB file. This file will be compared with the hardware generated file (also in ASCII). The Character to bin button (performs the second phase only) converts the specified file (supplied from the user) to a binary file. The hardware generated file would be converted to a binary file and the converted data would be ready for the Diehard Battery of Tests. The Diehard Results performs these tests and returns how many tests passed. Also, the p-values (if there were any) are returned. The Run All button has the program perform all three phases of generating an ASCII file of data, converting this file to a binary data file, and performing the diehard battery of tests.

The next tab (Alternate Bit and Row) would be utilized (time allowing) for alternating two rules for a trial. The primary difference between this tab and CA is the addition of the second rule select box. There are two methods of alternation: 1. Bits and 2. Rows.

The alternating bits (Figure 18) alternates the applied rule to each bit specified. Usually one rule is applied to all even-bits while the other rule is applied to all the odd-bits.

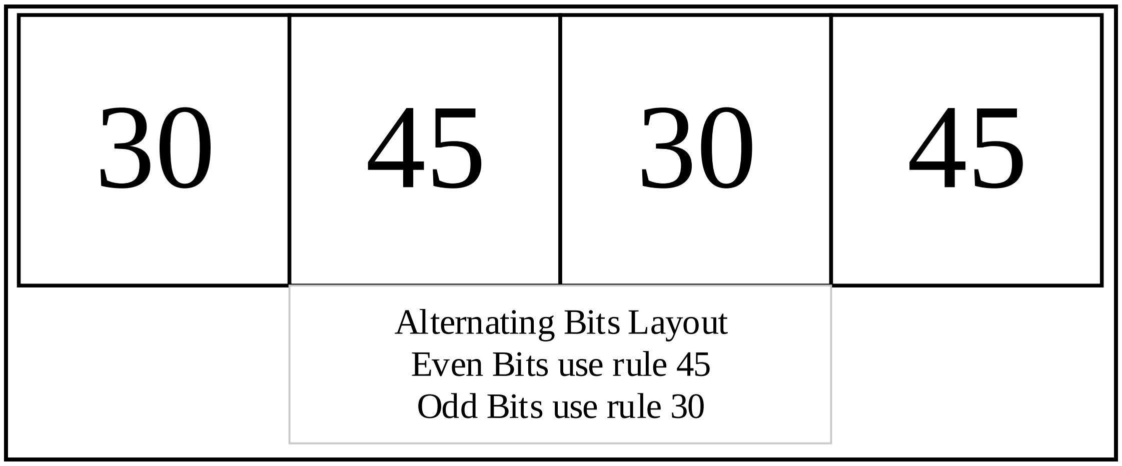
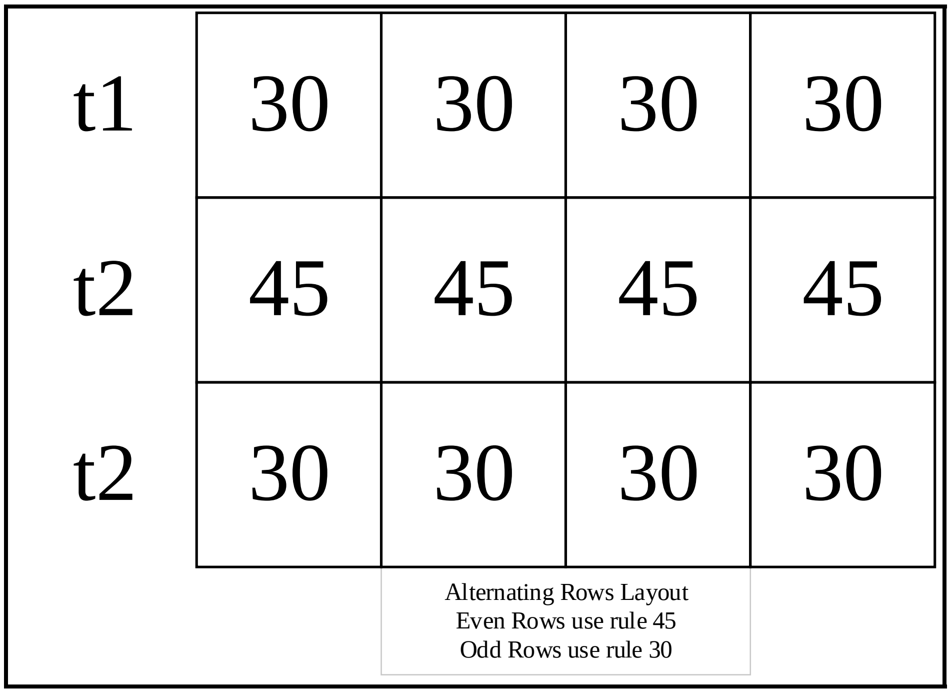


Figure 18: Illustration of an Alternating Bits Layout Using Rules 30 and 45

The alternating Row differs by using one rule for each bit during the even iterations and the second rule on the odd iterations (figure 19). During the first iteration (odd) rule 30 is selected, and applied to every bit. When the next iteration (even) occurs, rule 45 is applied to each bit.



| | | | | |
|---|---|---|---|---|
| t1 | 30 | 30 | 30 | 30 |
| t2 | 45 | 45 | 45 | 45 |
| t2 | 30 | 30 | 30 | 30 |

Alternating Rows Layout
Even Rows use rule 45
Odd Rows use rule 30

Figure 19: Illustration of Alternating Rows Layout Using Rules 30 and 45

To save resources and money, existing equipment already in use at MTSU would be utilized. The primary development board used was Altera©[1]'s DE2 board. This is the same board that is used for the digital labs. The DE2 board contains a field-programmable gate array (FPGA) device that is reprogrammable. The software necessary to utilize this board (commonly known as QUARTUS II) is also provided by Altera. The primary computers that were used operated on Windows©[2] XP. The LINUX OS would

[1] © 2010 Altera Corporation. All rights reserved. ALTERA, ARRIA, CYCLONE, HARDCOPY, MAX, MEGACORE, NIOS, QUARTUS and STRATIX are Reg. U.S. Pat. & Tm. Off. and or trademarks of Altera Corporation in the U.S. and other countries.

[2] ™ Windows is a registered trademark of Microsoft Corporation in the United States and other countries.

also work with this project, but the majority of our computers uses the XP operating system.

This project was done in three phases. The initial phase involved the use of graphically designed files. The second phase begins with the usage of a hardware descriptive language file. The final phase is the actual trial runs with data collection and results comparison.

Phase one was to use graphically designed files to begin the project. These files would determine if the project would be feasible or not. This phase would be restricted to development of the initial design and simulations. The maximum design size was limited to eight bits.

Phase two was to utilize hardware description files. These files were used for the actual trials generated for this project. Successful pre-trials (file smaller than 80MB) at this phase meant the project would continue to the actual full trials (generate file sizes of 80MB) and data collection.

The final phase was to conduct the full-size trials (80 MB files), collect and store the data, and compare this data with identical data generated by the CA_TABS program. Again the 80 MB file is the size the text file has to be when generated. The text is eight-bit ASCII and needs to be converted to a 10 MB data file before the Diehard Battery of Tests can be performed.

A simplified version of the DE2 board is illustrated in figure 20. For communications, the DE2's on board switches and LEDs was used. The user will determine the rule desired. The rule has to be a binary representation so the on-board switches are used to input this rule number to the processor. The LEDs were used to visually confirm the switch positions of the rule transmitted to the NIOS processor. The USB port on the DE2 board was the primary means of data transfer for storage from the DE2 board to the host PC. The DE2 board was designed to act like a USB device that had to be recognized by the XP OS, and the data sent from the DE2 board would be saved in a file on the PC.

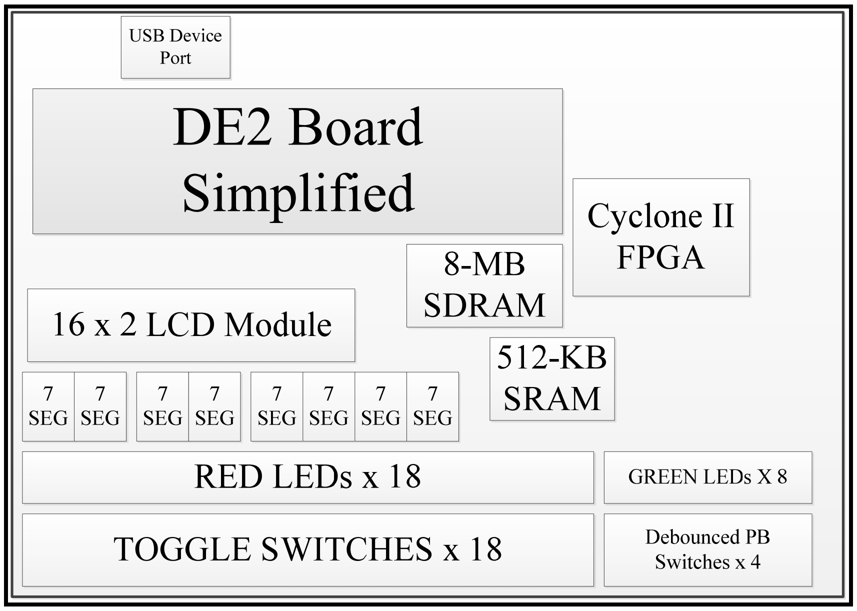


Figure 20: Simplified DE-2 Board

Both the pre (initial test file less than 80MB in length) and full trials (80 MB in length) required a custom processor that QUARTUS II provided. This processor (the NIOS II) is a LINUX based processor. It was to handle the random number generation, communication between the PC and the DE2 board, and results storage. One of the push buttons is used as a master reset to the NIOS processor (when installed) in the FPGA.

The primary communication means from QUARTUS II to the DE2 board comes from the USB port. This port will be used for the two way communications (and data transfer) between the NIOS processor and the PC (figure 21). The memory for the NIOS II processor comes from the on-board SDRAM. This device houses the NIOS processor, the necessary additional hardware, and the programming code that will run the NIOS processor. During initial development, the SRAM memory and LCD module provides visual feedback to the user troubleshooting information during failures or success if everything was correct.

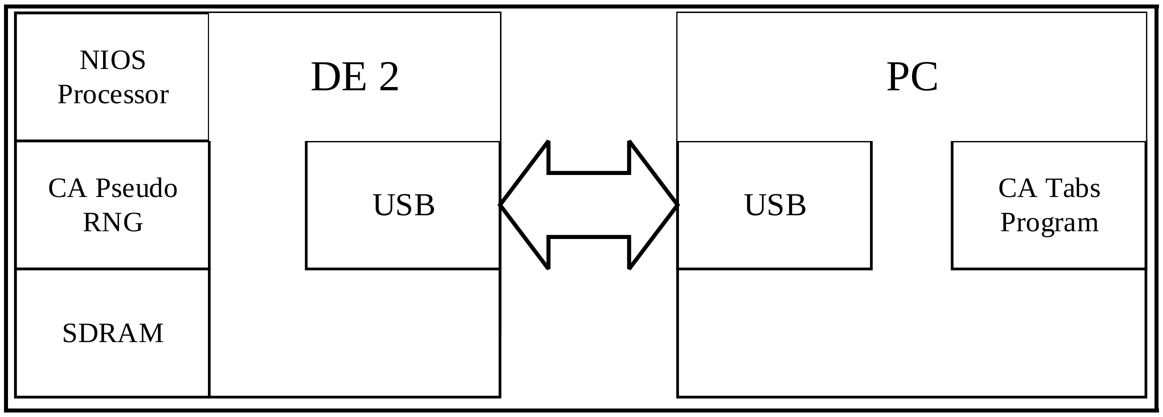


Figure 21: Communication Flow Between DE-2 Board and PC (Simplified)

The final phase will be data generated, data collection, and results comparison. The first part of the comparison was the comparison (bit by bit) of the last line of each run generated by both the hardware text file and CA_TABS text file. If these results matched, both files would be converted into binary data files and final comparisons would use Marsagila’s Diehard Battery of Tests [14]. The Diehard Battery of Tests was incorporated into the CA_TABS program and used for determination of strong pseudo-random numbers.

## GRAPHIC DESIGN IMPLEMENTATION

This phase required the creation of a cell along with the interconnection to its neighbors. The cell has two parts. The first is a combinational circuit that consists of logic units. The second is a storage (memory) device.

The first part consists primarily of combinational circuits. This circuit requires an input for the rule desired. This input is eight-bits wide to provide one of 256 rules chosen. The second input has to select the desired output from the rule provided. This input is three-bits wide and will determine the output of this unit. The device chosen for this task is a 3:8 multiplexer. By placing the rule on the eight-bit inputs, the three-bit select input will determine the output. Figure 22 illustrates a 3:8 multiplexer. QUARTUS II provides the user with a way to create unique multiplexers that will be programmed into the FPGA.

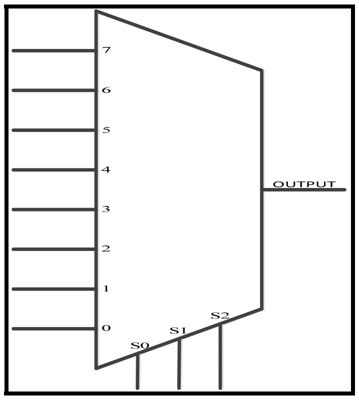


Figure 22: 3:8 Multiplexer

The multiplexer detailed in figure 22 highlights the two inputs and output. The inputs to the left of the multiplexer comes from the rule selected. This is an eight-bit number. Each individual bit applied to the input from the most significant bit (input seven) down to the least significant bit (applied to input zero). These eight bits form a binary number that ranges from zero to two hundred fifty-five. Using rule 30, as an example, the equivalent is 0 0 0 1 1 1 1 0. These inputs are assigned to the eight inputs of the multiplexer's eight-bit input. The second input is the return inputs from this and both neighbors' outputs. These inputs form a three-bit binary number from zero to seven. If, for example, a value returned was a binary four, the fourth input is selected and, what is on this input line (a one), will be the output from the multiplexer.

At a minimum, three multiplexers are required. How these three cells are interconnected is shown in figure 23.

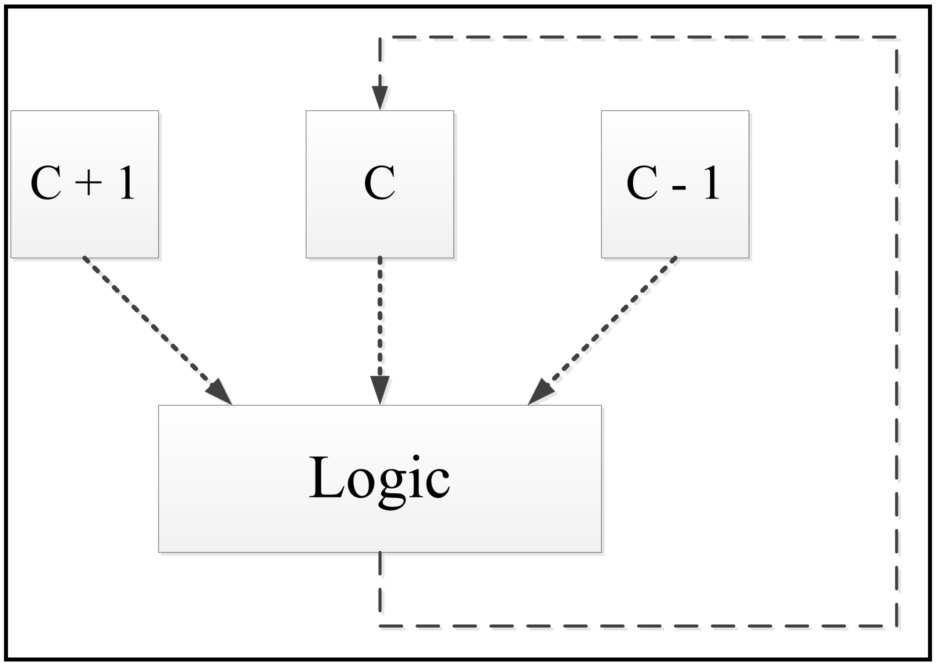


Figure 23: Simplified 3 Cell Design

Cell C's output is applied to a logic unit. This logic unit follows the rule (not shown) set forth by the user. The neighboring cells (C+1 and C-1) provide the remaining inputs to the unit. The three inputs form a binary number that the logic unit understands and generates the output based on this input. The output is returned to cell C and forms the new binary number.

Placing three of these multiplexers in a schematic file and interconnecting them can be done but, the logic unit will turn into an out of control free running oscillating device. This condition is caused by the feedback of the outputs directly into the inputs. These inputs are immediately acted upon and this output is again fed back into the logic unit. The speed of this free running device is determined by the propagation delays associated with the internal logic units. The second part (storage unit) prevents this condition from happening. This part is another logic unit configured to preserve its output until an update to change it is received from the controlling source. The D-type flip-flop is the device chosen for this task.

Figure 24 illustrates the complete cell. The storage unit is the D-type flip-flop. The inputs are the set, clear, clock (clk) and enable (ena). The set and clear inputs immediately forces the output to change to the required value regardless of the control signals. During normal operations, the output can only change during the clock cycle. The enable input allows the clock signal to update the output based on the input. The output from the multiplexer is applied to the input of the D-FF (labeled D). If the enable signal is activated, and the clock is cycled, the output from the multiplexer is stored in the flip-flop. The set (or clear) signal lets the user manually pre-set the output to a known condition (setting the seed). Setting the seed is the placement of all registers to a fixed, known condition.

With rule 30 as the illustrated example (figure 24), the outputs from the cell C+1 are placed on the ‘next’ input, C is fed back to the ‘this’ input, and C-1 is routed to the ‘prev’ input. These three inputs form a binary number that ranges from zero to seven and are usually referred to as the select lines. Whatever binary number is present at the select lines, the output from the multiplexer will be one of these eight values. This output is sent to the input of the D-FF. When the controlling signals command the D-FF to update to a new value, this value, at the input, is now stored in the D-FF. This value will remain until the controlling signals command the D-FF to change to its new value.

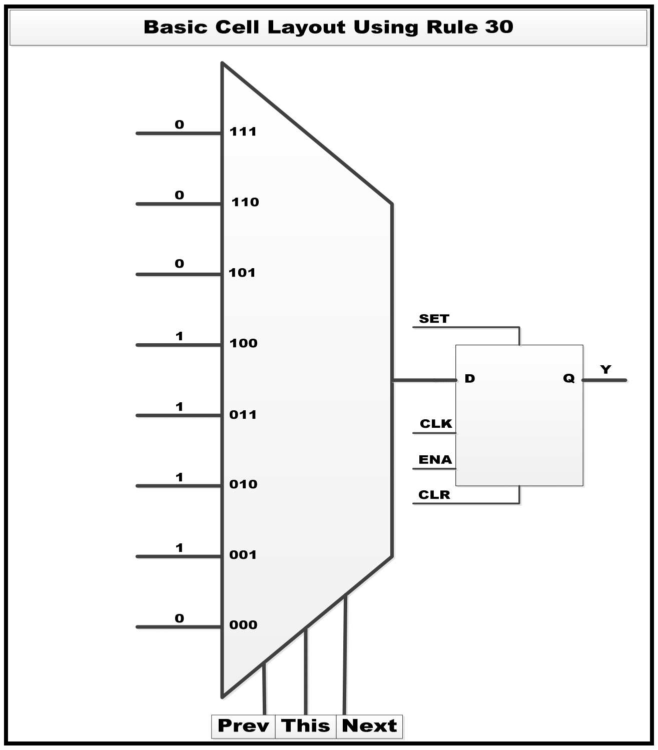


Figure 24: Basic Cell Layout Using Rule 30

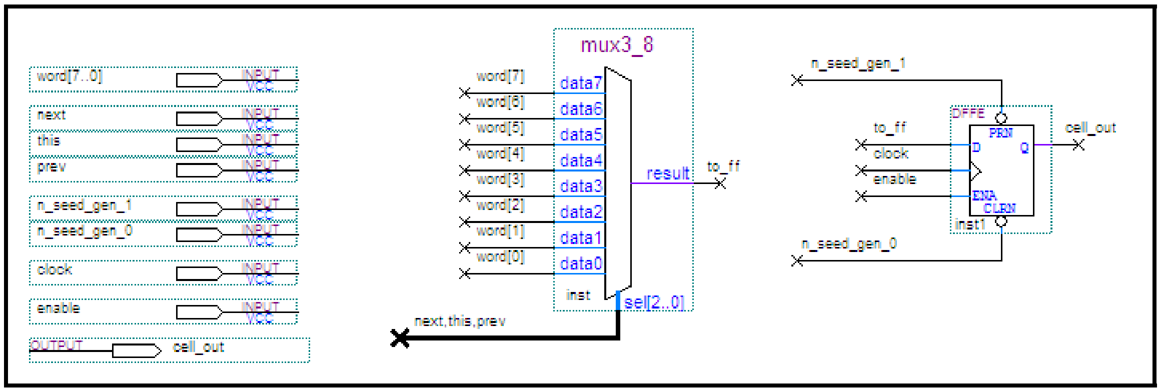


Figure 25: Quartus II Basic Cell Layout – Graphic Design File

Figure 25 is a graphical designed file created in QUARTUS II. This file becomes a block symbol file that can be reused in other files. The block symbol is a box symbol. This box symbol replaces the detailed circuit of figure 25. Using block symbols allows the user to create simplified diagrams and is a convenient method to place as many cells into a new project, and test them as shown in figure 26.

Four symbols are placed into a new project as shown in figure 26. Figure 25 illustrates the internal view of each Cell_Bit blocks and interconnects the blocks as required. Note: some connections do not require a connection and were left unconnected. Once this schematic was completed, simulations began. These simulations are critical and determine if this experiment is generating the correct results. Failure at this point would mean starting over and trying something new. To determine success, the pen and paper results in Appendix B are compared with the simulations. The results matched and a cell size of eight bits began.

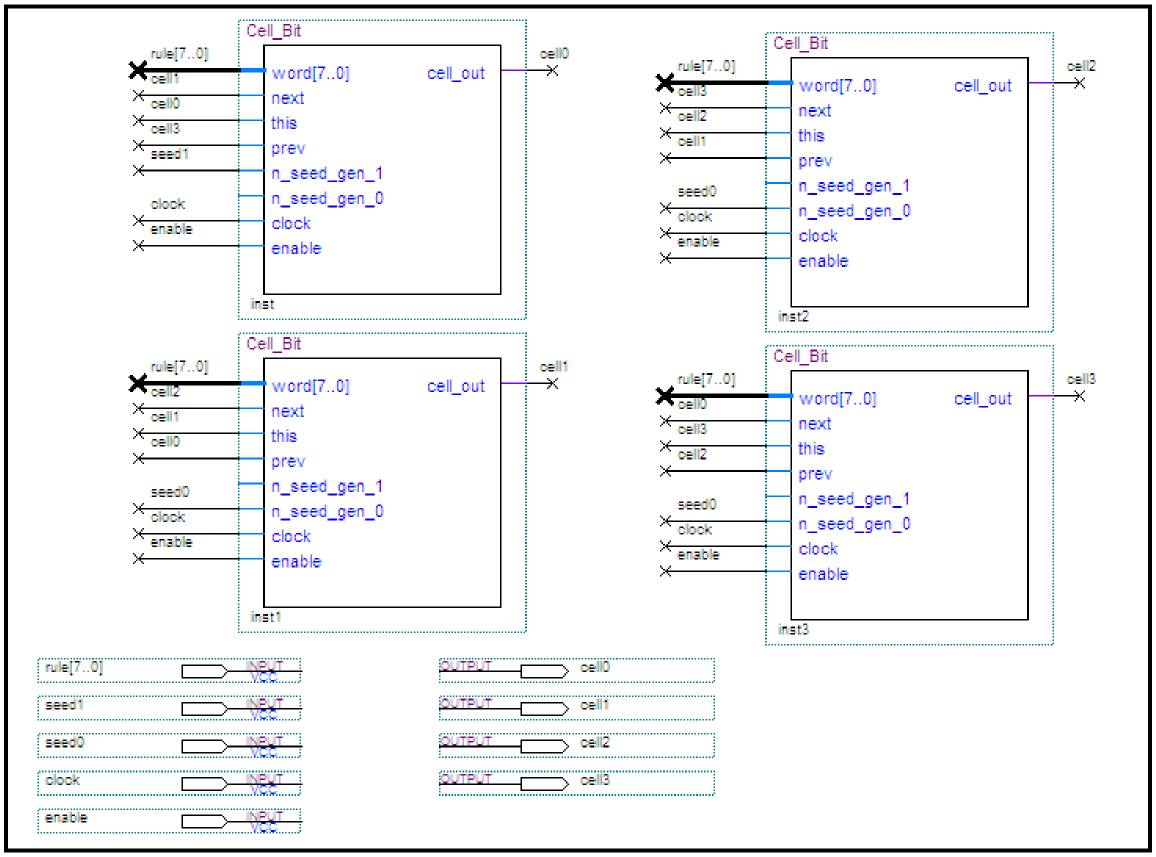


Figure 26: Graphic Design - Four Bit Layout

The eight-bit system was just as simple to create; following the same procedure as in the four bits design, only this time using eight bits (figure 27). Eight bits were placed in a new schematic file. It was immediately apparent how tedious interconnecting this design would be. Again, an incorrectly wired design results in failure. The simulations were checked against the pen and paper results in Appendix B, and they matched.

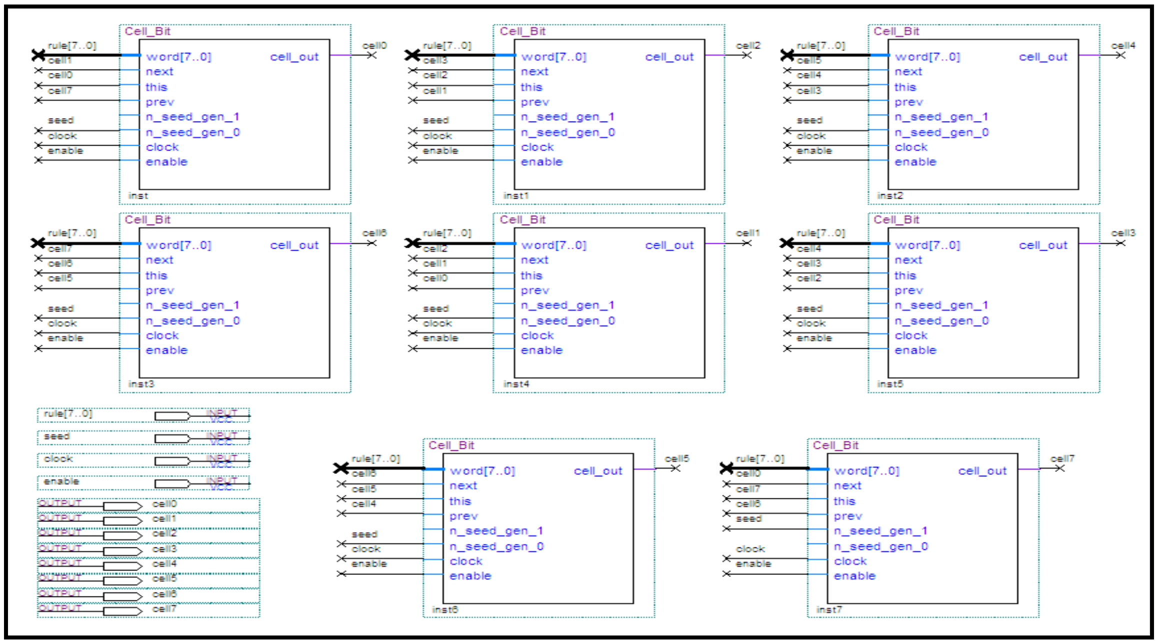


Figure 27: Graphic Design of an Eight Bit System Layout

If there was going to be a greater number of bits used, a better way was needed to do this. Fortunately, there were no errors made linking these eight bits. However, there was no guarantee that mistakes could not be made in future projects with differing bit sizes. Every effort was used to provide clear and descriptive naming conventions. This was to ensure interconnecting each bit would be as simple as possible. However, mistakes could have been made anyway.

Also, the four-bit system illustrated in figure 26 could be made into a block symbol project. Then an eight-bit system could get created by just placing two of these four-bit block into the drawing. Then this project could be converted into a symbol project. Then, new projects based on multiples of eight bits could be built.

Another problem with the eight-bit graphic designed system involved timing issues. During initial development, there were problems uploading the projects in the DE-2 board due to timing issues between the Cyclone II device and the SDRAM memory. To get around these issues, the on-board memory contained internally in the Cyclone II device was used. However, given the size and complexity of future projects, the on-board memory of 4k would not be enough. The DE2's SRAM was also limited with 512KB of memory. While the SRAM was suitable for the smaller projects, there would not be enough memory for the larger projects. The 8MB SDRAM would have to be used for the larger projects. The problem with the graphic designed files was the NIOS processor would not load properly onto the Cyclone II device if the SDRAM was used. The delay associated between the Cyclone II device and the SDRAM memory amounts to -3 Nano seconds. Attempts were made to vary the delay times ranging from +5 Nano Seconds to -5 Nano Seconds. Nothing worked. At this point, the graphic design was scrapped (because there was little choice) and continued research using a hardware description language (HDL) approach began.

# VHDL DESIGN IMPLEMENTATION

The main question was, which HDL (Verilog or VHDL) would be chosen? VHDL was selected. VHDL stands for VHSIC (very high-speed integrated circuit) hardware description language. Around the 1980’s the U.S. Department of Defense (DOD) sponsored a program for describing the structure and function of integrated circuits (IC’s). In time, the DOD transferred this program to IEEE (Institute of Electrical and Electronics Engineers) [15] [16]. This choice (VHDL) was made because this was the only material initially available at MTSU. In prior basic digital courses, the textbook used employed VHDL as the programming language for the examples. The language seemed easy enough to follow and there were numerous documentation and examples available on the web. The final decision that forced the utilization of VHDL was the previously mentioned timing issues. Altera provided a tutorial for the NIOS processor. This tutorial used a VHDL generated file, along with the properly created clock unit, solved the -3 Nano second delay issue with the SDRAM.

As with any new language, there is a learning curve associated with VHDL. With numerous information (examples, lecture notes, and e-books) available on the web, a basic understanding of the language (syntax) developed. Initial cell development began.

Modularity was the main goal. This meant the same code was reused with other projects. The basic design would not differ from the graphic design files previously mentioned. VHDL can implement the multiplexer and flip-flops necessary. But a change in the procedure was needed. The main section had to be as generic as possible. This one-size fits all concept will save time when the number of bits are increased in future experiments. A change of one line in the code produced units of any bit-length desired. This also meant the design had to be broken into two parts – 1: up to 32 bits, and 2: Greater than 32 bits.

Up to 32 bits design would be simple enough. Because the NIOS processor is a 32-bit processor, modifications won’t be needed to connect the CA_Bits unit to the NIOS Processor. However, CA_Bits units greater than 32 bits will require modifications and shall be discussed later.

A simplified diagram of the CA_Bits unit is provided in figure 28. The clock (CLK) and enable (ENA) inputs are there to iterate the output (from Word Bits (X)). The seed input will set the initial condition of the output to a known condition. The SW inputs (eight total) are the (binary) rule number inputs.  Within this unit the two VHDL files are contained. The first sub-file contains the multiplexer and a D-type flip-flop. The second sub-file repeatedly calls the first sub-file and provides interconnection. Internally, this has the same look and operation as the graphic one-bit cell (figure 25). The operation is the same.

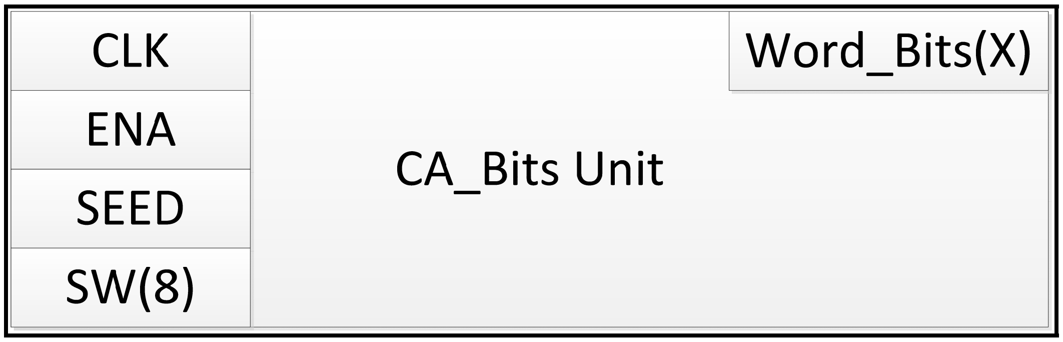


Figure 28: Simplified CA Bits Unit

Appendix I (is the VHDL representation of the CA_Bits unit) contains all the information that creates the self-contained file. The first section defines and creates the multiplexer and flip-flop. The entity of this section contains the inputs (clock, enable, cells (next, this, previous), clear, set, and the rule number) and output (q) of the multiplexer. To eliminate confusion, this first part is called CA_Bit. The second part is called CA_Bits.

The second part of the file immediately follows the first parts completion. This section (called CA_Bits) uses the first part to create the number of bits to experiment with. This section has its own unique I/O. The output (word_out) is a buffered I/O. This buffered I/O provides two-way data flow to this section. The set word sets the outputs to the seed value. For this experiment, the MSB is set to a logic one, while the remaining bits are set to a logic low. The clock and enable provide the external control signals to update the data. The second part repeatedly calls the first section each time it wants to create a new bit. After the new bit is created, the second part proceeds to interconnect this bit with the previously created bits. There are two unique bits that have to be handled separately; the first and last bits. These bits have to interconnect to each other, and this second part provides the specific instructions to complete this requirement. The remaining bits interconnection are identical and connect the same way with the creation of every new bit. Once this process is compiled, a unique unit is created with the proper number of bits to test. The generic statement in the code allows for changing the number of bits desired. This is a convenient method because only one line in the code needs to change. A four-bit unit was created. If this trial was successful, the results would match the four bits graphical designed trials. They did! The ability to create one file, generate the desired number of bits to test, simulate the results, and run the hardware version in real-time was achieved.

Next an eight-bit unit was developed and the results were compared with the graphic designed results. They matched. Because of these successes, the project could continue using the units of the desired bit lengths.

Previous work by MTSU graduate students suggested that prime numbers should provide the best results in generating pseudo-random numbers. Composite numbers tend to produce short cycles and these short cycles produce poor results. The prime numbers chosen for this experiment are 31, 61, 127, 223, 383, 479, 541, 607, 733, 827, 991, and 1021 (2039 was added later to determine the amount of resources used by the Cyclone II device previously mentioned). For simplicity, several different folders of each cell size was created. The modified cell size units were given unique filenames and saved into their respective folders. When the trials were ready to begin, all that was required was to load these projects into the DE2 board and run the test. The amount of resources was recorded whenever the cell count increased. This data would be used to approximate the maximum number of cells that could be used on the DE2 board. In the end, 24% on-board resources were used with the cell size of 1023 bits. 42% on-board resources were required with the cell size of 2039 bits. Appendix F lists the resources used for each cell size.

With the size of bits determined, what rules would be ones chosen? Cellular Automata has four classes (homogenous (I), periodic (II), chaotic (III), and complex

(IV)). The rules chosen need to be both chaotic (class III) and balanced. The chaotic class produces results in a chaotic pattern. To further clarify balanced rules, even though some rules fell into the chaotic (III) class, the rules need to be balanced. Balanced (for our purposes) holds an equal number of ones and zeroes in the rule number. For example, rule 30 (in binary) has an equal number of ones and zeroes (0 0 0 1 1 1 1 0). For the eight bits, there are four zeroes and four ones. Appendix D shows the rules that meet both the chaotic and balanced criteria (there are 16). There are other rules that are balanced, but they do not fall into the chaotic class (III). For example, rule 170 (1 0 1 0 1 0 1 0) is balanced but it falls into class 2 (Periodic). The total number of actual balanced rules is determined by the formula $\frac{8!}{4!*(8-4)!} = 70$. Rules: 30, 45, 90, and 150 met both conditions. With four DE2 boards available, individual tests were run simultaneously using these rules (30, 45, 90, and 150).

Individually, the CA_Bits unit will work on its own. However, control will be required when the CA_Bits unit operates. This will require the use of a processor. The processor chosen is the NIOS II processor. The NIOS II processor is Altera’s custom definable self-contained processor that embeds into a FPGA device like the Cyclone II chip installed on the DE2 board. The custom NIOS processor has to have only what is needed for this project. A simplified version of this design is shown in figure 29.

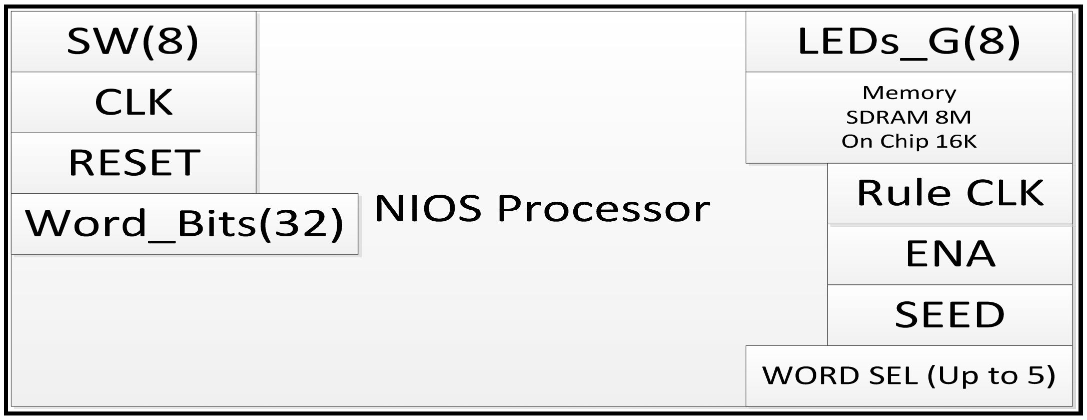


Figure 29: Basic NIOS Processor (Simplified)

Looking at figure 29, inputs are needed for the rule (SW (8), clock (CLK), system reset (RESET), and the word generated by the CA Bits unit (Word Bits (32)). The outputs (visual display of the rule switches (LEDs G (8)), individual connections to memory (Memory), a controllable clock signal (Rule CLK), a signal to set the CA Bits to a known condition (SEED), an enable signal to allow clock to update the output, and a special output to a multiplexer (WORD SEL) that will only be used with bits greater than 32. This special multiplexer and select unit will be discussed later in the section for greater than 32 bits.

Creation of a NIOS II Basic system requires the creation of a processor with external I/O ports. This creation process is done with Altera’s software (SOPC Builder) that is specifically designed to create customizable NIOS II processors and integrates with the user’s project. These ports, along with the logic required for data flow through these ports, has to be defined and generated. Figure 30 displays a simplified schematic and external interconnections of the NIOS II processor.

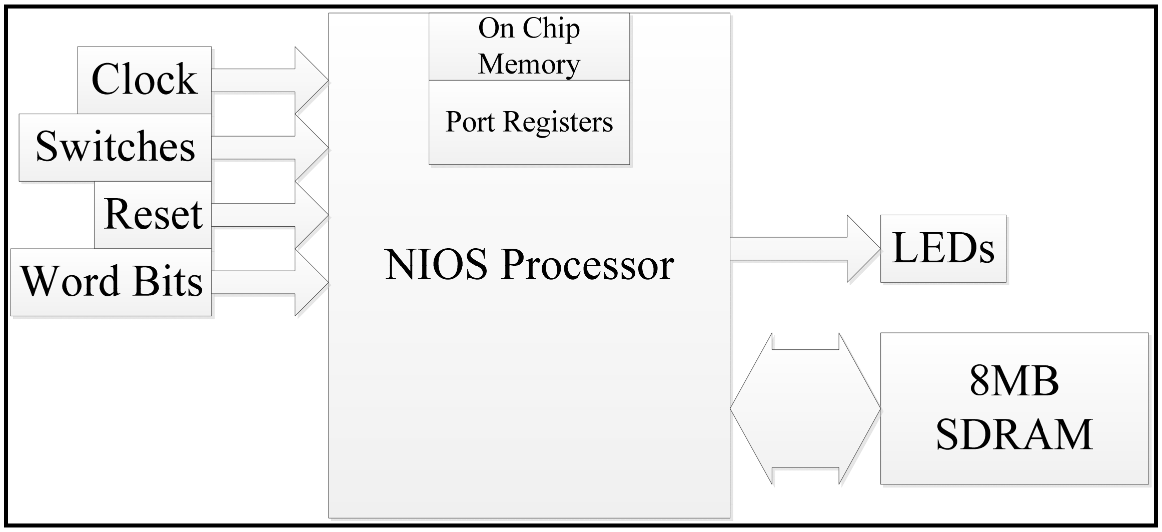


Figure 30: Basic NIOS System (Simplified)

Each processor required an extensive test to determine the operability of this processor. The toggle switches had to be read correctly, and the LEDs had to display properly. This required a program to be written and included when the system is loaded up onto the DE2 board. Testing this part required the use of another Altera program (NIOS IDE). This program allows the user to run, step-by-step, the C program developed by the user. This code (for this example) looks at the individual ports, sees any changes to the ports, and displays the appropriate results. To use this processor and the CA Bits unit together, a new project has to be created and the VHDL representation of these files has to be included.

With both modules installed, several programs (in C) were written to test the results. The main routine consists of clearing all displays, reading the switches, setting the green LEDs (to display the selected switches) accordingly, setting the seed pulse to one and back to zero (placing the CA_Bits unit to a known starting condition), displaying the four separate bits to four separate LEDs (this shows the output of the CA_Bits unit), setting the enable bit to one and setting the clock to one, then resetting the clock to zero (this should have iterated the CA_Bits unit), and finally resetting the enable bit back to zero.

These results were compared with the pen and paper results. The results were identical. This meant further experimentation could be used for this design.

The next test was trying to run a simplified test to include saving the data to the PC. Initially, the main problem was interfacing the ISB-1362 IC (USB HOST/DEVICE controller) that is integrated with the DE2 board. In time, some basic communication with the host PC was made. However, a program that runs on the host PC to communicate with the DE2 board could not be developed. Several months of trial and failures reduced this project to using the host file-system included with the tutorials provided by Altera. Using the host file-system meant that the programs could only run through the NIOS IDE in debug mode only. This killed the self-containment part (a self-contained, stand-alone unit) which was one of the requirements. It does not mean this experiment fails, but additional research would be needed to complete this requirement.

Before a full trial could begin, a CA_Bits module 31-bits in length was created. The NIOS processor required no modifications. A general program was written that would read the user selected rule to run, seed the CA_Bits unit, create and open a file, complete enough iterations that generates enough data (making a file 80MB in size), save this data line-by-line, and when done, close the file and properly terminate the program.

Running this experiment with 32 bits (or less) ran as designed, however, a problem was discovered when units greater than 32 bits in length were to be tested. The problem is that the NIOS processor is a 32-bit processor. As such, all data has to be 32 bits in length. The question was how to get data greater than 32 bits to fit into a 32-bit system? The solution to this problem was to use a specially designed multiplexer (Figure 31).

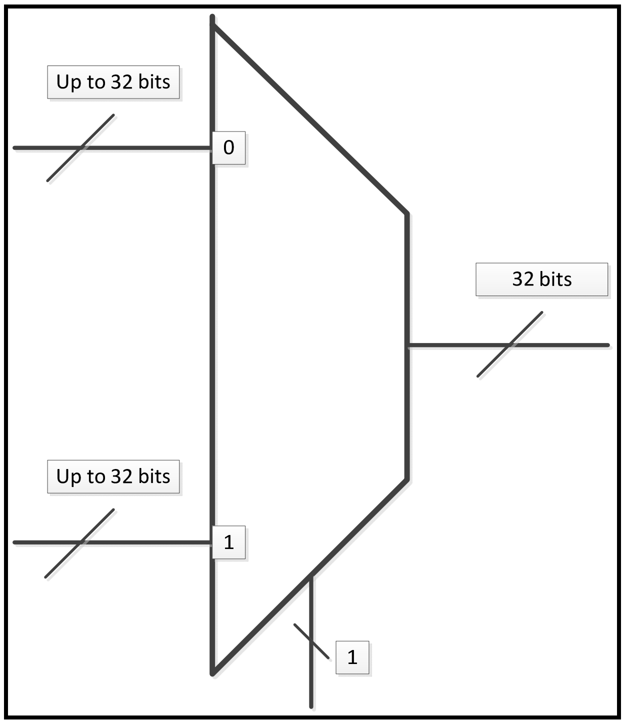


Figure 31: Simplified Diagram of a 32-Bit 2-1 Multiplexer

This multiplexer takes the entire data and reduces it into 32 bit chunks of data. Half of the data is applied to one input of the multiplexer. The other half is applied to the other input. The select input decides which half of the data is sent to the output of the multiplexer.

Figure 32 details a simplified version of a system that handles 64 bits of data.

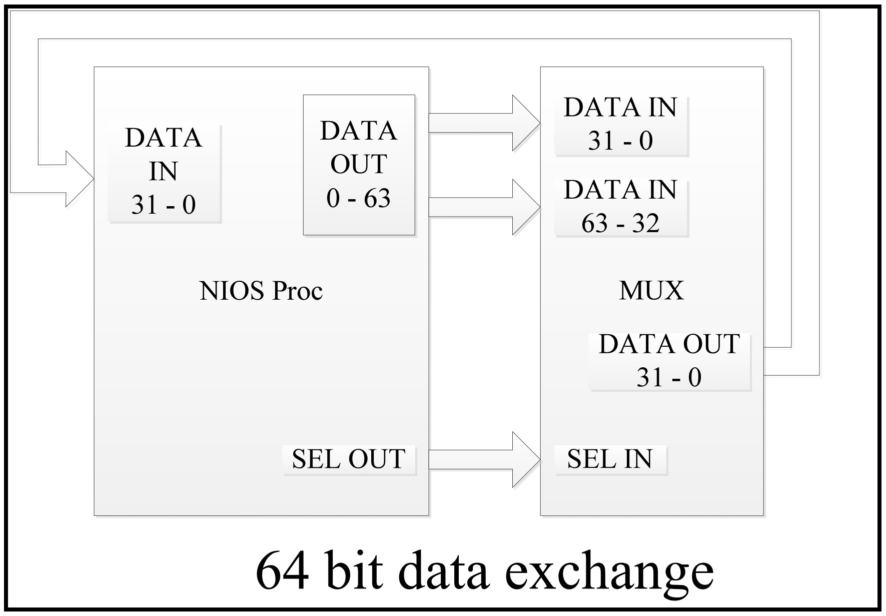


Figure 32: 64-Bit Data Exchange

The NIOS processor has an added output to this multiplexers select input. For simplicity, the CA_Bits unit is inside the NIOS processor. The data from the CA_Bits is sent out as 64 bits of data. To process this data, the returning data has to be returned 32 bits at a time. For this to happen, the select out chooses the high-order bits (63-32) and these bits are returned to the NIOS processor and handled. The select line then picks the low-order bits and the process is repeated.

The full version system (figure 33) has three major components: the NIOS processor, the CA_Bits unit, and the wide multiplexer. The NIOS processor controls the operation of the three components. The CA_Bits unit contains and iterates the data. The wide multiplexer separates the data stream into 32 bit chunks of data. Basic operations are similar to the smaller unit. The main difference is how the data will transfer to the PC.

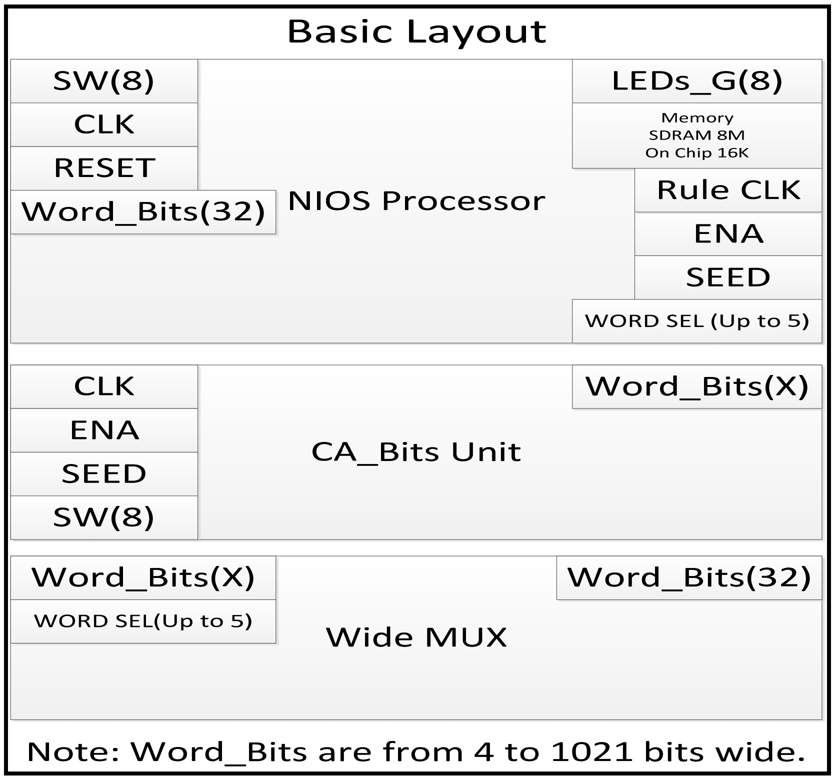


Figure 33: Simplified View of the Complete System

The NIOS processor holds the code for program execution. This code is the same as the smaller system except, additional instructions to control the wide mux. Again the code resets the processor and CA_Bits. Remember, the NIOS processor is a 32-bit system. In order to get the data to the PC, the data stream has to be no greater than 32 bits. The processor sets the iterate command and the CA_Bits unit makes one iteration of the data. This data is the full length bits. In order to separate these bits, the wide multiplexer has these bits sent to its input. The inputs are 32 bits wide each. Using the 64-bit example, the first 32 bits are applied to input group 1. The remaining bits are applied to group 2. Each group is 32 bits in length. The select line is either one or zero. The NIOS processor commands the wide multiplexer to send the first 32 bits of data by setting the select line to one. This data is returned to the NIOS processor and the processor transmits (bit-by-bit) this data to the PC to be saved. Once all these bits are saved, the processor sets the select line to zero. The remaining data is returned and this data is stored. After all 64 bits of data is transferred, the processor instructs the CA_Bits unit to iterate the CA Bits unit and the process repeats.

The CA_Bits component had to be an internally generated file. This component could be used externally. However, for the Cyclone II device; there is a limit to the amount of pins that could be used. The Cyclone II device has almost 400 external pins; therefore, the only components that could be built would be limited to 400 bits. Internally connectable devices, for all intents and purposes, have an unlimited amount of pins. These internal pins are really interconnecting wires. This interconnect capability provides the ability to make a unit virtually any size desired.

The wide multiplexer is the key for splitting up the data into 32 bit chunks. The choice of the prime numbers chosen is because of the wide multiplexer. The select lines are based on powers of two. If each group of inputs is 32 bits wide, special multiplexers just wide enough to hold all of the data could be made. The maximum numbers for additional select lines are calculated as: $2^1 = 64$ bits, $2^2 = 128$ bits, $2^3 = 288$ bits, $2^4 = 511$ bits, and $2^5 = 1024$ bits. As long as a multiplexer (large enough to handle the data) existed, data could be broken down into 32 bits of data that could be saved on file.

What the maximum number of bits that could realistically be used had to be determined. Ideally, there could be an unlimited number of bits, but the amount of resources available on the FPGA devices is going to limit how big the device could be made. During each compilation, the resources used was tabulated (Appendix F) (for each increased bit size) during the creation of each project. Initially 1023 bits was the stopping point. For the size of 1023 bits, the compiler used 24% of the Cyclone II's resources. If that was the case, then trying 2039 bits should use under 50% of the resources. One more project was generated with the size of this unit increased to 2039 bits. The compiler generated this unit using 42% of the resources. Therefore, an estimated maximum size (using the DE2's Cyclone II device) of 4096 bits could be created.

Initial trials were set to run 100 iterations of data and save these results. Next the CA_TABS program was also used to generate 100 iterations of data. The last lines of both data files was placed in a text file and a one-for-one-bit comparison of these last lines was performed. For this experiment's successful conclusion, these results must match. They did! The next phase of this experiment requires a complete run that generates an 80MB data file.

A final experiment was to the usage of two rules. This meant modifying the CA_Bits source code to include a second rule input. A final modification of the C source code had to ignore the read switch function and have the program insert the rules itself. Trial began using three K-sizes of 31, 113, and 129 bits. The four rules (30, 45, 60, and 90) in all possible combinations (ten) were determined and the 80MB data files were generated.

For encryption of data, the source code has to be modified by removing the hardwired latching signals to the cells. Initially, the MSB is set to one while the remaining bits are set to zero. This condition makes the system a pseudo-random number generator. The code needs to have a seed value for each cell to follow an external input from the user/program. The only requirement is that the seed value never starts out with all zeros. Since the seed will comprise ones and zeros, this condition will (in all likelihood) never happen.

## DATA COLLECTION AND TRIALS

The final runs have to generate files 80MB in size. Marsaglia's Diehard Battery of Tests of Randomness [14] requires a binary file size of 10MB. The PC stores characters (ones and zeros) in an eight-bit format. Therefore, the file needs to store eight-times the 10MB requirement (making the total file size 80MB). This requires enough iterations to generate enough data. The main loop counter is the number of iterations required to produce a file size greater than 80MB. To produce this number, the file size had to be divided by the number of bits. For example, to produce the proper iterations for a trial that is 31 bits wide is ($\frac{80,000,000}{31} = 258045.161$). As long as iterations greater than 258046 is used, trials completed with a file size greater than 80MB should be completed. Appendix E shows the minimum iterations table.

Appendix C details a (simplified) program flow the NIOS II processor uses during this experiment. The detailed operation of this file is as follows: The preprocessor routine defines and initializes (if necessary) the ports. It also declares data, constants, and function declarations. There is only one necessary include file and that was the iostream file. This file holds all the operations the compiler requires for I/O operations. The first declarations are the global declarations. The global declarations defined in this file details the pointers to the NIOS ports. In order to pass information into and out of the NIOS processor, the information must transfer through the NIOS ports. Access of these ports is controlled by the program through the use of pointers to these locations. The constants declared are: the main iterations, register size, MSB size, two registers to hold data, two masks for data comparison, and a file pointer for data transfer to the PC. The last things defined are the user created functions. These functions will do things like read in words, cycle a clock pulse, set seed … Additional information about these functions is explained in the following paragraphs.

The main function has the job of controlling the overall program flow. All the data that requires initialization has to be done first. Additional registers are created (and initialized) at the start of program execution. These registers are a loop counter, the rule number, a loop counter for rerunning loops and one for how many runs to perform (usually set to one). The filename of the file the PC is going to save is defined and created here. The data is stored as a string of characters, and, when called for, transfers this information to the PC. The last two items completed are calling the read switches function to get rule number selected by the user. The rule number is also saved in a special register for later use.

The main program loop begins and cycles until the program is finished. The first part of the loop prepares the filename. Confirmation of the filename is returned to the user in message form. File I/O operations begin. A new file is opened and a pointer to this file is created here. Failure to open this file pointer results in an error message. The next part displays a message notifying the user what rule number is set and calls individual functions to set the seed, reads in a word to the processor, displays the word, prints an end of line character and sends the data to the PC. Once this is done, another loop is entered. This loop calls functions that cycles a clock pulse, reads the new word, displays it, appends an end of line character, and sends the updated data to the PC.

Throughout this loop, the count is compared and messages are generated displaying the programs progress. If the main loop count reaches a number greater than the desired iteration, then a message is displayed showing the progress of the program in quarterly outputs ($\frac{1}{4}$, $\frac{1}{2}$ , and $\frac{3}{4}$ complete).

Immediately after exiting the loop, the program sends the final line of data to the PC, the final end of line character is generated, and sends the final character to the PC. The final part initiates the shutdown procedure (if no more runs are needed). After exiting this loop, a message states the program has finished and halts.

The user functions (8) are described below.

Read in a word. This function takes in all the data from the CA_Bits unit. Since this data is greater than 32 bits in length, this function stores the data in 32 bit chunks. For example, if there are 64 bits of data, this function stores these bits in two 32 bit registers.

Display the word. This function prepares the data for display. The function does a bit-by-bit comparison with a special mask. The registers are compared against this mask. If the result matches the mask, the display bit function is called to send a character one to the PC. If the result does not match the mask, then the display bit function is called to send a character zero to the PC.

Display bit function. This function sends a command to the PC to append the file with a one or zero. Additionally, this function could have this bit displayed on the monitor to see the results. Initially, this secondary function was used for debugging.

Cycle clock function. This function calls the set enable function, send a logic one to the clock port, then sends a logic zero to the clock port, and finally calls the clear enable function. This special clock pulse controls when the CA_Bits unit updates its data.

Read switch function. This function examines the port the user selectable switches (from the DE2 boards on board switches) are set to. The setting of these switches determines what the user defines the rule to be. The eight switches form an eight-bit binary value. This value is converted to the rule number and is stored in the rule number register for further use later.

Set seed function. This function transmits the signal to the CA_Bits unit to set the registers to a known condition. In this case, the MSB is set to one while the remaining bits are set to zero.

Set (and clear) enable functions. These function transmit a logic one (zero) to the CA_Bits unit for control of a controlled clock cycle generated by the cycle clock function.

A word of note about the clock pulse generated. The main clock pulses come from the phase lock loop generator created for the NIOS processor. However, there could be problems using this clock. Some instructions might require two or more clock cycles to execute that instruction.  The CA_Bits unit must iterate one time (and one time only) and ensure the data does not change until another iteration is requested. The machine language code required to just run this part is going to definitely be more than one clock cycle. Generating a manual clock pulse by software ensures that the CA_Bits unit will cycle only one time. The added enable bit is an additional safeguard that prevents possible spurious clock generation signals. In this case, the enable must be set and a clock pulse must be generated by the NIOS processor. Failure of either signal will result in no iterations of the CA_Bits unit.

To test each project, four DE2 boards were setup. Each board had one rule that it was going to run (30, 45, 90, and 150). Now the NIOS processor could be loaded, run the trial, obtain the results, and then prepare tomorrows trials to run the next day. For example, by running a test batch of 31 bits. Four DE2 boards would have a 31-bit program in them. Each DE2 board would run different rules however. One would run rule 30. The remaining boards would run rules 45, 90, and 150. The next day, repeat this process only using the next size (63 bits).

The biggest problem of transmitting all this data (regardless of length), is the time it took to transfer from the NIOS processor, through the USB port, and save on the PC. As of this writing, the USB standard is version two. This meant that data transfer is fairly slow. The transfer of data flows as – after the CA_Bits iterates, the data is returned to the NIOS processor. Then the processor, bit-by-bit, transmits the data through the USB port to the PC. The PC stores each bit in the file and waits for the next bit (or end of line character). The CA_Bits unit will keep iterating and generating new data until the process is complete. The complete process generates enough data to create an 80MB file. This is a slow process, but it was the only one that could be worked with. The initial time (using the DE2 board’s on-board 50MHz clock) for generating an 80MB file would have taken over 12 hours. Increasing the clock speed of the system to 100MHz, reduced the run-time to a little over 6 hours. Unfortunately, this posed a problem. When a trial using the prime number of 863 was attempted, somehow this particular unit did not want to run at 100MHz. Reduction to the prime number (827) produced a unit that would run at 100MHz. Every other prime number originally chosen ran at 100MHz.

In mid-June 2013, full trials began, starting with 31 bits. Trials continued until mid-July 2013. The final trial concluded with a test size of 2039 bits. All runs were completed with a clock speed of 100MHz and took just over 6 hours to complete. With the exception of 863 bits, all original prime number values selected, loaded, and ran completely (827 bit was the size that replaced 863 bits). The data was collected and transferred to a central location. These results would be compared with the CA_TABS program.

For a successful conclusion, the results generated have to match the results generated by the CA_TABS program. For confirmation, two tasks had to be completed. The first is a bit-by-bit comparison of the last lines of each run along with the CA_Tabs generated version. The last task is to compare the file’s results with each other using the Diehard Battery of Tests. The Diehard Battery of Test results were tabulated and transcribed to a spread sheet. These results will also have the CA_TABS programs Diehard Battery of Test results tabulated and stored in the same spread sheet. If the results do not match, the experiment fails and further testing need not be done.

The first task involves copying the last iteration (of the run) and pasting it into a text file. Then take the last iteration (of a CA_TABS generated run) and paste it directly below the first iteration. By placing the bits over each other, discrepancies in the data streams are easier to find. If all bits are the same, then the data streams match. Again, if the data streams did not match, the experiment failed. The results did match, the next phase of the experiment began.

The next phase of this experiment required that both of the software and hardware results were compared using the Diehard battery of tests. The first run was completed with the seed included in the data. The first trial (in software) was completed, and the results were both saved and placed into a spreadsheet table for comparison with the hardware trials later. The hardware trials were completed, and the results are stored in the central location along with the spreadsheet results (appendix G).

The very last thing was to repeat these runs, only this time, remove (absorb) the seed. Prior experimentation revealed that if the seed is not absorbed, the p values (along with the overall results) are not as good as with the seed absorbed. To remove the seed, the first K-size iterations have to be removed from the beginning of the file. For example, if the K-size was 31 bits, then the first 31 iterations had to be removed (using a text editor) from the file. This modified file was run using the Diehard Battery of Tests. The results (p value and overall) were also saved and placed in the spreadsheet for later comparison.

In June 2016 trials on dual rules began. Like the single rules, the results were collected. However, there was a problem with the CA_Tabs dual rule generation program. This meant that the results could not be compared with the CA_Tabs program. The data could still be tested with the Diehard Battery of Tests to determine if the hardware generated data produced strong encryption results.

In continuance of further testing, trials began using two rules. The source code was modified to provide the rules and ignore the single rule input by the external switches that the user entered manually, renaming of the filenames and placement into a special folder unique to this trial, and a specialized counter to alternate the rule based on even-odd iterations.

Initially, the modifications to the code worked. However, numerous problems cropped-up during run-times of the project. Specifically, the files were not saving correctly. The data would seed correctly, but the only data being generated with each iteration was zeros. The problem was the external switches were still being read. After removing the read switches function in the source, the correct data was saved into the file.

The 31-bit runs were completed and the data saved and stored. This data will have the diehard tests performed when all data had been collected. New runs were generated for a 127-bit machine. The results were saved and tested with the CA Tabs program and the results was placed into a spreadsheet for later comparison.

The final step was to remove (absorb) the seed. Initially, Quartus’s text editor had to be used. This provided line numbers for each line. For 31 bits, the first 31 lines had to be removed. For 127 bits, the first 127 lines had to be removed. While Quartus provided editing capabilities, a freeware text editor program (ConTEXT) was obtained. This text editor had line numbers built into the text file. This made removing the proper amount of line numbers easier.

With the seed removed, the results improved. However, no p-values were generated with any of the trials. After discussing these findings with the project advisor, another trial using 113 bits was recommended. Prior research suggested that this value should provide satisfactory results. All trials using 113 bits were conducted and the results were tabulated. The results are displayed in the appendix H also. These results are much better than the original trials (31 and 127 bits). P-values were generated.

## FINDINGS AND RESULTS

The first batch was completed with the seed included in the data. The first trial (in software) was completed, and the results were saved and placed into a spreadsheet table for comparison with the hardware trials later. The hardware trials were completed, and the results are stored in the central location along with the spreadsheet results (appendix G).

The very last thing was to repeat these test, only this time, remove (absorb) the seed. Prior experimentation revealed that if the seed is not absorbed, the p values (along with the overall results) are not as good as with the seed absorbed. The results (p value and overall) were also saved and placed in the spreadsheet for later comparison. Findings revealed that the absorbed seed produced better results.

When a trial using the prime number of 863 was attempted, somehow this particular unit did not want to run at 100MHz. Reduction of one prime number less (832) resulted with the same problem. One further reduction to the next lower prime number (827) produced a unit that would run at 100MHz. Every other prime number originally chosen ran at 100MHz.

The trials using dual rules were tabulated and the two initial bit sizes of 31 and 127 produced dismal results. Even with the seed removed, there was some improvement but the results were unsatisfactory. A third rule was chosen that should produce satisfactory results. The trial, for this size (113) produced acceptable results. The results were tabulated and stored in a spreadsheet. These results are detailed in Appendix H.

There was limited success with independent USB communication. The DE2 provides a USB controller chip that will become (depending on the programmer) a USB host or USB device. The tutorial (included) did not work with this version of QUARTUS (9.0). There were numerous problems trying to even get the board to recognize the ISP1362 controller device. Eventually, communication with the PC host controller was established. Due to the limited time left, this portion of the experiment was abandoned. Eventually, a different controller had to be used. This would establish communications and even have the PC save the data. However, this meant that the system had to run in debug mode with the NIOS IDE. Running this in the debug mode meant the project would not be self-contained. Despite this limitation, a system that met the criteria of generating pseudo random numbers was developed.

Another time consuming problem worth mentioning had to be overcome, creating a driver. The driver is (for all intents and purposes) a translator between the PC and the new hardware trying to connect to it. The driver tells the PC how to communicate with this piece of hardware. Even though Microsoft made available a driver development kit, it still took close to a year to get a (somewhat) working driver.

# CONCLUSIONS

A working elementary cellular automata device has been implemented, and can be installed into a FPGA device, provided that the FPGA is large enough for the system to be installed. For this project, the seed was fixed to set the MSB to one while the remaining bits were set to zero. This limitation was removed to allow the user to provide variable seeds that meets their requirements.

Additional work produced a second system that allowed the use of two rules. This system offered the user the added ability to alternate between two rules for each iteration.

This study lays the basic foundation for additional development of more complex cellular automata of two and three dimensional designs. Additional research pursuing the development of these devices will provide future systems with stronger encryption capabilities.

# REFERENCES

[1] Jennifer Wilcox. (2006). Solving the Enigma: History of the Cryptanalytic Bombe [On-line]. Available: https://www.nsa.gov/about/_files/cryptologic_heritage/publications/wwii/solving_enigma.pdf [Mar 30, 2015].

[2] Stephen P. Zammit. (2012). Behind the Victory: Midway and JN25 [On-line]. Available: http://www.cscss.org/featured/midway/Feature_Briefing-CSCSS_005_Midway_JN25.pdf [Mar 30, 2015].

[3] R. Ward and T. Molteno. (Oct 26, 2007). Table of Linear Feedback Shift Registers [On-line]. Available: http://courses.cse.tamu.edu/csce680/walker/lfsr_table.pdf [Sept 14, 2015].

[4] Palash Sarkar. *A Brief History of Cellular Automata.* ACM Computing Surveys, Vol32, No. 1, March 2000.

[5] Jarkko Kari. *Theory of cellular automata: A survey.* Theoretical Computer Science 334 (2005) 3-33, November 2004. [On-line], Available: www.sciencedirect.com [Sept 14, 2015].

[6] Stephen Wolfram. *A New Kind of Science*. Wolfram Media, Inc. May 14, 2002.

[7] Stephen Wolfram. *Random Sequence Generation by Cellular Automata.* Advances in Applied Mathematics 7, 123-169 (1986), Academic Press, Inc. 0196-8858/86.

[8] Cheryl A. Kincaid, Saraju P. Mohanty, Armin R. Mikler, Elias Kougianos, and Brandon Parker. "A High Performance ASIC for Cellular Automata (CA) Applications," in 9th International Conference on Information Technology (ICIT'06), 2006, IEEE, 0-7695-2635-7/06.

[9] Asfaw Beka Estub. "Random Number Generation Using Cellular Automation." M.S. thesis, Middle Tennessee State University, May 2010.

[10] Christopher Blake Stocker. "One Dimensional Cellular Automata Research Platform." M.S. thesis, Middle Tennessee State University, August 2010.

[11] Eric W. Weisstein. (2015). Cellular Automation. From MathWorld—A Wolfram web Resource. [On-line] Available: http://mathworld.wolfram.com/CellularAutomation.html [Sept 14, 2015].

[12] Stephen Wolfram. *Theory and Applications of Cellular Automata*. World Scientific Publishing Co. Ltd. Pages 485-557. 1986.

[13] Wikipedia. Linear Feedback Shift Register. [On-line] Available: http://en.wikipedia.org/wiki/Linear_feecback_shift_register [Sept 14, 2015].

[14] George Marsaglia. The Marsaglia Random Number CDROM including the Diehard Battery of Tests of Randomness. Available: http://stat.fsu.edu/pub/diehard/ [Sept 14, 2015]

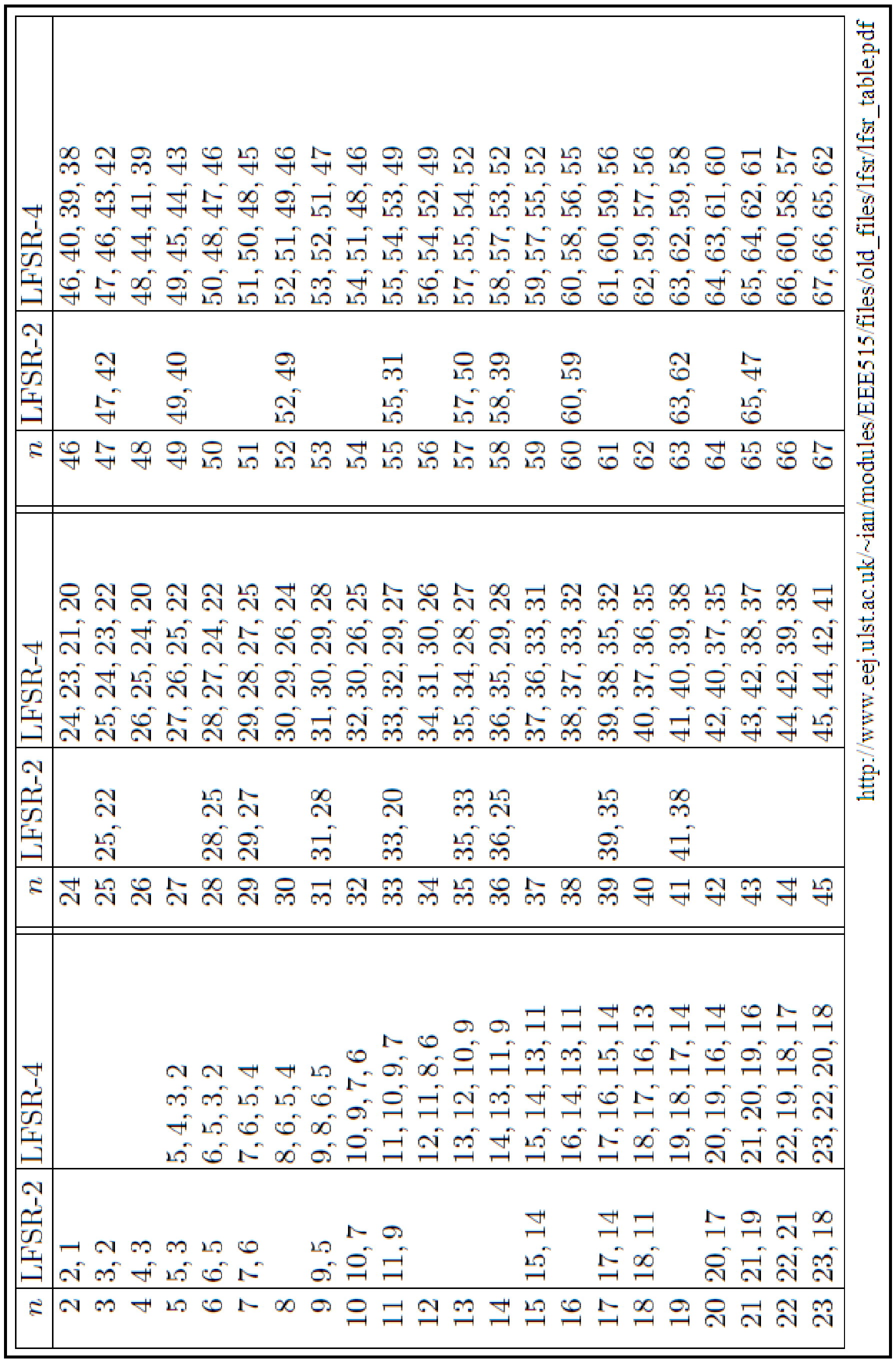
[15] Peter J. Asherton and Jim Lewis. *The Designer's Guide to VHDL (3rd)*. Morgan Kaufmann Publishers, 2008.

[16] Pong C. Chu. *RTL Hardware Design Using VHDL*. John Wiley & Sons, 2006.

# APPENDICES

## APPENDIX A

**Table 1: Taps Table for Up to 67 LSFR Bits**



| $n$ | LFSR-2 | LFSR-4 | $n$ | LFSR-2 | LFSR-4 | $n$ | LFSR-2 | LFSR-4 |
|---|---|---|---|---|---|---|---|---|
| 2 | 2, 1 | | 24 | | 24, 23, 21, 20 | 46 | | 46, 40, 39, 38 |
| 3 | 3, 2 | | 25 | 25, 22 | 25, 24, 23, 22 | 47 | 47, 42 | 47, 46, 43, 42 |
| 4 | 4, 3 | | 26 | | 26, 25, 24, 20 | 48 | | 48, 44, 41, 39 |
| 5 | 5, 3 | 5, 4, 3, 2 | 27 | | 27, 26, 25, 22 | 49 | 49, 40 | 49, 45, 44, 43 |
| 6 | 6, 5 | 6, 5, 3, 2 | 28 | 28, 25 | 28, 27, 24, 22 | 50 | | 50, 48, 47, 46 |
| 7 | 7, 6 | 7, 6, 5, 4 | 29 | 29, 27 | 29, 28, 27, 25 | 51 | | 51, 50, 48, 45 |
| 8 | | 8, 6, 5, 4 | 30 | | 30, 29, 26, 24 | 52 | 52, 49 | 52, 51, 49, 46 |
| 9 | 9, 5 | 9, 8, 6, 5 | 31 | 31, 28 | 31, 30, 29, 28 | 53 | | 53, 52, 51, 47 |
| 10 | 10, 7 | 10, 9, 7, 6 | 32 | | 32, 30, 26, 25 | 54 | | 54, 51, 48, 46 |
| 11 | 11, 9 | 11, 10, 9, 7 | 33 | 33, 20 | 33, 32, 29, 27 | 55 | 55, 31 | 55, 54, 53, 49 |
| 12 | | 12, 11, 8, 6 | 34 | | 34, 31, 30, 26 | 56 | | 56, 54, 52, 49 |
| 13 | | 13, 12, 10, 9 | 35 | 35, 33 | 35, 34, 28, 27 | 57 | 57, 50 | 57, 55, 54, 52 |
| 14 | | 14, 13, 11, 9 | 36 | 36, 25 | 36, 35, 29, 28 | 58 | 58, 39 | 58, 57, 53, 52 |
| 15 | 15, 14 | 15, 14, 13, 11 | 37 | | 37, 36, 33, 31 | 59 | | 59, 57, 55, 52 |
| 16 | | 16, 14, 13, 11 | 38 | | 38, 37, 33, 32 | 60 | 60, 59 | 60, 58, 56, 55 |
| 17 | 17, 14 | 17, 16, 15, 14 | 39 | 39, 35 | 39, 38, 35, 32 | 61 | | 61, 60, 59, 56 |
| 18 | 18, 11 | 18, 17, 16, 13 | 40 | | 40, 37, 36, 35 | 62 | | 62, 59, 57, 56 |
| 19 | | 19, 18, 17, 14 | 41 | 41, 38 | 41, 40, 39, 38 | 63 | 63, 62 | 63, 62, 59, 58 |
| 20 | 20, 17 | 20, 19, 16, 14 | 42 | | 42, 40, 37, 35 | 64 | | 64, 63, 61, 60 |
| 21 | 21, 19 | 21, 20, 19, 16 | 43 | | 43, 42, 38, 37 | 65 | 65, 47 | 65, 64, 62, 61 |
| 22 | 22, 21 | 22, 19, 18, 17 | 44 | | 44, 42, 39, 38 | 66 | | 66, 60, 58, 57 |
| 23 | 23, 18 | 23, 22, 20, 18 | 45 | | 45, 44, 42, 41 | 67 | | 67, 66, 65, 62 |

http://www.eej.ulst.ac.uk/~ian/modules/EEE515/files/old_files/lfsr/lfsr_table.pdf

## APPENDIX B

## **Pen and Paper Results for 4 Bits**

| For Rule 30 | | | | | |
|---|---|---|---|---|---|
| Iteration | Cell 3 | Cell 2 | Cell 1 | Cell 0 | |
| 0 | 0 | 0 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 | 1 | B |
| 2 | 0 | 0 | 1 | 0 | 2 |
| 3 | 0 | 1 | 1 | 1 | 7 |
| 4 | 0 | 1 | 0 | 0 | 4 |
| 5 | 1 | 1 | 1 | 0 | E |
| 6 | 1 | 0 | 0 | 0 | 8 |
| 7 | 1 | 1 | 0 | 1 | D |
| 8 | 0 | 0 | 0 | 1 | 1 |
| 9 | 1 | 0 | 1 | 1 | B |
| 10 | 0 | 0 | 1 | 0 | 2 |
| 11 | 0 | 1 | 1 | 1 | 7 |
| 12 | 0 | 1 | 0 | 0 | 4 |
| 13 | 1 | 1 | 1 | 0 | E |
| 14 | 1 | 0 | 0 | 0 | 8 |
| 15 | 1 | 1 | 0 | 1 | D |
| 16 | 0 | 0 | 0 | 1 | 1 |
| 17 | 1 | 0 | 1 | 1 | B |
| 18 | 0 | 0 | 1 | 0 | 2 |
| 19 | 0 | 1 | 1 | 1 | 7 |
| 20 | 0 | 1 | 0 | 0 | 4 |
| 21 | 1 | 1 | 1 | 0 | E |
| 22 | 1 | 0 | 0 | 0 | 8 |
| 23 | 1 | 1 | 0 | 1 | D |
| 24 | 0 | 0 | 0 | 1 | 1 |

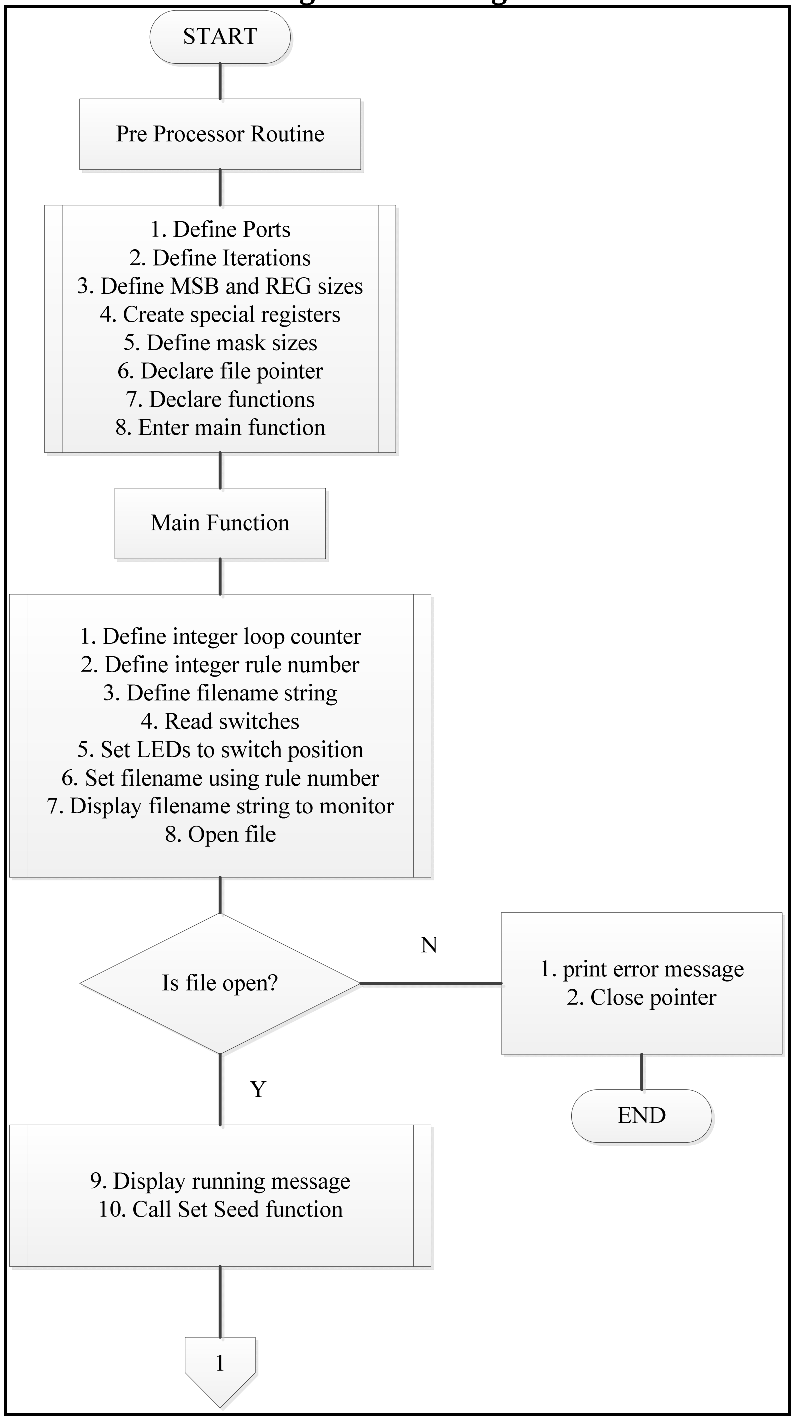
| Rule 30 | |
|---|---|
| 0 | 0 |
| 1 | 1 |
| 2 | 1 |
| 3 | 1 |
| 4 | 1 |
| 5 | 0 |
| 6 | 0 |
| 7 | 0 |

## Pen and Paper Results for 8 Bits

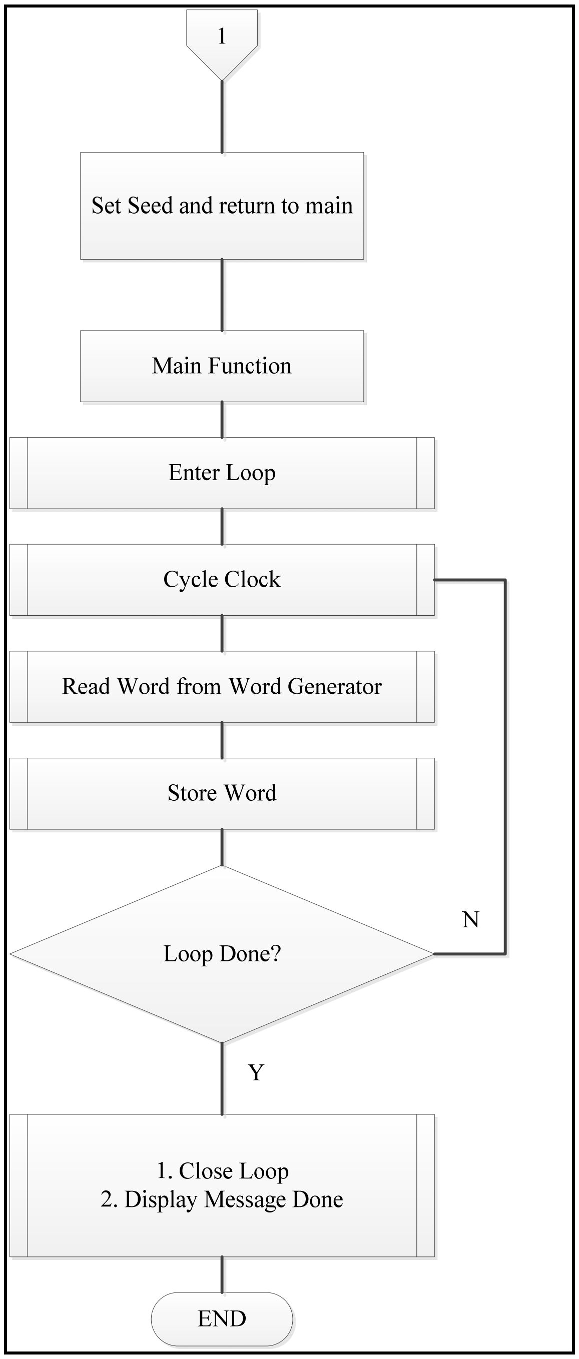
| For Rule 30 | | | | | | | | | | | Rule 30 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Iteration | Cell 7 | Cell 6 | Cell 5 | Cell 4 | Cell 3 | Cell 2 | Cell 1 | Cell 0 | | | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 80 | | 1 | 1 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | C1 | | 2 | 1 |
| 2 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 23 | | 3 | 1 |
| 3 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | F6 | | 4 | 1 |
| 4 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 84 | | 5 | 0 |
| 5 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | CF | | 6 | 0 |
| 6 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 38 | | 7 | 0 |
| 7 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 64 | | | |
| 8 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | DE | | | |
| 9 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 90 | | | |
| 10 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | F9 | | | |
| 11 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 07 | | | |
| 12 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 8C | | | |
| 13 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | DB | | | |
| 14 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 12 | | | |
| 15 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 3F | | | |
| 16 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | E0 | | | |
| 17 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 91 | | | |
| 18 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 7B | | | |
| 19 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 42 | | | |
| 20 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | E7 | | | |
| 21 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1C | | | |
| 22 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 32 | | | |
| 23 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 6F | | | |
| 24 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 48 | | | |
| 25 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | FC | | | |
| 26 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 83 | | | |
| 27 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 46 | | | |
| 28 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | ED | | | |
| 29 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 09 | | | |
| 30 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 9F | | | |
| 31 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 70 | | | |
| 32 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | C8 | | | |
| 33 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | BD | | | |
| 34 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 21 | | | |
| 35 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | F3 | | | |
| 36 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0E | | | |
| 37 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 19 | | | |
| 38 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | B7 | | | |
| 39 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 24 | | | |
| 40 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 7E | | | |
| 41 | Cycle repeats here | | | | | | | | C1 | | | |

# APPENDIX C

## Program Flow Page 1


START
Pre Processor Routine
1. Define Ports
2. Define Iterations
3. Define MSB and REG sizes
4. Create special registers
5. Define mask sizes
6. Declare file pointer
7. Declare functions
8. Enter main function
Main Function
1. Define integer loop counter
2. Define integer rule number
3. Define filename string
4. Read switches
5. Set LEDs to switch position
6. Set filename using rule number
7. Display filename string to monitor
8. Open file
Is file open?
N
1. print error message
2. Close pointer
END
Y
9. Display running message
10. Call Set Seed function
1

**Program Flow Page 2**



## APPENDIX D

**Table 2: The 16 Chaotic and Balanced Rules**

| Rule | MSB | 6 | 5 | 4 | 3 | 2 | 1 | LSB |
|---|---|---|---|---|---|---|---|---|
| 30 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 |
| 45 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 |
| 60 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 |
| 75 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 |
| 86 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 |
| 89 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 |
| 90 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 |
| 101 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 |
| 102 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| 105 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 |
| 135 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| 149 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 |
| 150 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 |
| 153 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 |
| 165 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| 195 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 |

These rules fall into Class 3 (Chaotic) and have an equal number of zeros and ones (Balanced).

## APPENDIX E

### Table 3: Iterations and Quarters Table

| Bits | Max Iter | Min Iter | Tot Iter | 1/4 | 1/2 | 3/4 | Expected Size | Actual Size | Difference | Was Iterations Enough |
|---|---|---|---|---|---|---|---|---|---|---|
| 31 | 2645161.29 | 2580645.161 | 2600000 | 650000 | 1300000 | 1950000 | 80600000 | 85800035 | 5200035 | Y |
| 61 | 1344262.295 | 1311475.41 | 1320000 | 330000 | 660000 | 990000 | 80520000 | 83160065 | 2640065 | Y |
| 127 | 645669.2913 | 629921.2598 | 640000 | 160000 | 320000 | 480000 | 81280000 | 82560131 | 1280131 | Y |
| 223 | 367713.0045 | 358744.3946 | 365000 | 91250 | 182500 | 273750 | 81395000 | 82125227 | 730227 | Y |
| 383 | 214099.2167 | 208877.2846 | 214000 | 53500 | 107000 | 160500 | 81962000 | 82390387 | 428387 | Y |
| 479 | 171189.9791 | 167014.6138 | 171000 | 42750 | 85500 | 128250 | 81909000 | 82251483 | 342483 | Y |
| 541 | 151571.1645 | 147874.3068 | 151000 | 37750 | 75500 | 113250 | 81691000 | 81993545 | 302545 | Y |
| 607 | 135090.6096 | 131795.7166 | 135000 | 33750 | 67500 | 101250 | 81945000 | 82215611 | 270611 | Y |
| 733 | 111869.0314 | 109140.5184 | 111869 | 27967 | 55934.5 | 83901.8 | 81999977 | 84224452 | 2224475 | Y |
| 827 | 99153.56711 | 96735.18742 | 99153 | 24788 | 49576.5 | 74364.8 | 81999531 | 82198668 | 199137 | Y |
| 991 | 82744.70232 | 80726.53885 | 82744 | 20686 | 41372 | 62058 | 81999304 | 82165787 | 166483 | Y |
| 1021 | 80313.41822 | 78354.55436 | 80313 | 20078 | 40156.5 | 60234.8 | 81999573 | 82161224 | 161651 | Y |
| 2039 | 40215.79205 | 39234.91908 | 40215 | 10054 | 20107.5 | 30161.3 | 81998385 | 82080858 | 82473 | Y |

## APPENDIX F

**Table 4: Resources Used Table**

| Bits | Total Logic Elements Used | Total Memory Resources Used |
|---|---|---|
| 31 | 2349 /33216 (7%) | 141312 of 483840 (29%) |
| 61 | 2484/33216 (7%) | 141312 of 483840 (29%) |
| 127 | 2837/33216 (9%) | 141312 of 483840 (29%) |
| 223 | 3478/33216 (10%) | 141312 of 483840 (29%) |
| 383 | 4400/33216 (13%) | 141312 of 483840 (29%) |
| 479 | 4946/33216 (15%) | 141312 of 483840 (29%) |
| 541 | 5314/33216 (16%) | 141312 of 483840 (29%) |
| 607 | 5723/33216 (17%) | 141312 of 483840 (29%) |
| 733 | 6448/33216 (19%) | 141312 of 483840 (29%) |
| 827* | 6955/33216 (21%) | 141312 of 483840 (29%) |
| 991 | 7897/33216 (24%) | 141312 of 483840 (29%) |
| 1021 | 8064/33216 (24%) | 141312 of 483840 (29%) |
| 2039 | 13863/33216 (42%) | 141312 of 483840 (29%) |

Total resources used for the DE-2 board by logic and memory used to generate project.

Note: bit size 863 was the original trial size. However, this size did not work and bit reductions were implemented until a new size was found that would work. The selected size that worked was 827 bits.

APPENDIX G

**Table 5: Results for ECA One Rule Page 1**

| Without Seed Removed | | | | | |
|---|---|---|---|---|---|
| Rule | 30 | | | | |
| | CA Tabs | CA Tabs | Hardware | Hardware | End of File |
| Bit Size | Overall | p val | Overall | p val | Compared |
| 31 | 32 out of 229 | 0 | 32 out of 229 | 0 | Good |
| 61 | 120 out of 229 | 0 | 120 out of 229 | 0 | Good |
| 127 | 123 out of 229 | 0 | 123 out of 229 | 0 | Good |
| 223 | 148 out of 229 | 0 | 148 out of 229 | 0 | Good |
| 383 | 153 out of 229 | 0 | 153 out of 229 | 0 | Good |
| 479 | 154 out of 229 | 0 | 154 out of 229 | 0 | Good |
| 541 | 168 out of 229 | 0 | 168 out of 229 | 0 | Good |
| 607 | 151 out of 229 | 0 | 151 out of 229 | 0 | Good |
| 733 | 156 out of 229 | 0 | 156 out of 229 | 0 | Good |
| 827 | 146 out of 229 | 0 | 146 out of 229 | 0 | Good |
| 991 | 96 out of 229 | 0 | 87 out of 229 | 0 | Good |
| 1021 | 87 out of 229 | 0 | 50 out of 229 | 0 | Good |

| With Seed Removed | | | | | |
|---|---|---|---|---|---|
| Rule | 30 | | | | |
| | CA Tabs | CA Tabs | Hardware | Hardware | End of File |
| Bit Size | Overall | p val | Overall | p val | Compared |
| 31 | 30 out ot 229 | 0 | 30 out of 229 | 0 | Good |
| 61 | 123 out of 229 | 0 | 123 out of 229 | 0 | Good |
| 127 | 127 out of 229 | 0 | 127 out of 229 | 0 | Good |
| 223 | 178 out of 229 | 0 | 178 out of 229 | 0 | Good |
| 383 | 211 out of 229 | 0 | 211 out of 229 | 0 | Good |
| 479 | 210 out of 229 | 0 | 210 out of 229 | 0 | Good |
| 541 | 228 out of 229 | 0.643535 | 228 out of 229 | 0.643535 | Good |
| 607 | 213 out of 229 | 0 | 213 out of 229 | 0 | Good |
| 733 | 228 out of 229 | 0.418168 | 228 out of 229 | 0.418168 | Good |
| 827 | 229 out of 229 | 0.789072 | 229 out of 229 | 0.789072 | Good |
| 991 | 217 out of 229 | 0 | 217 out of 229 | 0 | Good |
| 1021 | 229 out of 229 | 0.036327 | 229 out of 229 | 0.036327 | Good |

## Results for ECA One Rule Page 2

Without Seed Removed

| Rule | 45 | | | | |
|---|---|---|---|---|---|
| | CA Tabs | CA Tabs | Hardware | Hardware | End of File |
| Bit Size | Overall | p val | Overall | p val | Compared |
| 31 | 15 out of 229 | 0 | 15 out of 229 | 0 | Good |
| 61 | 117 out of 229 | 0 | 117 out of 229 | 0 | Good |
| 127 | 107 out of 229 | 0 | 107 out of 229 | 0 | Good |
| 223 | 140 out of 229 | 0 | 140 out of 229 | 0 | Good |
| 383 | 138 out of 229 | 0 | 138 out of 229 | 0 | Good |
| 479 | 135 out of 229 | 0 | 135 out of 229 | 0 | Good |
| 541 | 167 out of 229 | 0 | 167 out of 229 | 0 | Good |
| 607 | 127 out of 229 | 0 | 127 out of 229 | 0 | Good |
| 733 | 157 out of 229 | 0 | 157 out of 229 | 0 | Good |
| 827 | 130 out of 229 | 0 | 130 out of 229 | 0 | Good |
| 991 | 73 out of 229 | 0 | 97 out of 229 | 0 | Good |
| 1021 | 97 out of 229 | 0 | 48 out of 229 | 0 | Good |

With Seed Removed

| Rule | 45 | | | | |
|---|---|---|---|---|---|
| | CA Tabs | CA Tabs | Hardware | Hardware | End of File |
| Bit Size | Overall | p val | Overall | p val | Compared |
| 31 | 16 out of 229 | 0 | 16 out of 229 | 0 | Good |
| 61 | 120 out of 229 | 0 | 120 out of 229 | 0 | Good |
| 127 | 114 out of 229 | 0 | 114 out of 229 | 0 | Good |
| 223 | 166 out of 229 | 0 | 166 out of 229 | 0 | Good |
| 383 | 199 out of 229 | 0 | 199 out of 229 | 0 | Good |
| 479 | 202 out of 229 | 0 | 202 out of 229 | 0 | Good |
| 541 | 226 out of 229 | 0.006984 | 226 out of 229 | 0.006984 | Good |
| 607 | 201 out of 229 | 0 | 201 out of 229 | 0 | Good |
| 733 | 227 out of 229 | 0.021466 | 227 out of 229 | 0.012966 | Good |
| 827 | 228 out of 229 | 4.00E-06 | 228 out of 229 | 4.00E-06 | Good |
| 991 | 200 out of 229 | 0 | 200 out of 229 | 0 | Good |
| 1021 | 226 out of 229 | 0.05517 | 226 out of 229 | 0.05517 | Good |

## Results for ECA One Rule Page 3

| Without Seed Removed | | | | | |
|---|---|---|---|---|---|
| Rule | 90 | | | | |
| | CA Tabs | CA Tabs | Hardware | Hardware | End of File |
| Bit Size | Overall | p val | Overall | p val | Compared |
| 31 | Fails | Fails | Fails | Fails | Good |
| 61 | 33 out of 229 | 0 | 33 out of 229 | 0 | Good |
| 127 | Fails | Fails | Fails | Fails | Good |
| 223 | 5 out of 229 | 0 | 5 out of 229 | 0 | Good |
| 383 | Fails | Fails | Fails | Fails | Good |
| 479 | 0 out of 229 | 0 | 0 out of 229 | 0 | Good |
| 541 | 0 out of 229 | 0 | 0 out of 229 | 0 | Good |
| 607 | Fails | Fails | Fails | Fails | Good |
| 733 | Fails | Fails | Fails | Fails | Good |
| 827 | Fails | Fails | Fails | Fails | Good |
| 991 | Fails | Fails | Fails | Fails | Good |
| 1021 | Fails | Fails | Fails | Fails | Good |

| With Seed Removed | | | | | |
|---|---|---|---|---|---|
| Rule | 90 | | | | |
| | CA Tabs | CA Tabs | Hardware | Hardware | End of File |
| Bit Size | Overall | p val | Overall | p val | Compared |
| 31 | Fails | Fails | Fails | Fails | Good |
| 61 | 35 out of 229 | 0 | 35 out of 229 | 0 | Good |
| 127 | Fails | Fails | Fails | Fails | Good |
| 223 | 4 out of 229 | 0 | 4 out of 229 | 0 | Good |
| 383 | Fails | Fails | Fails | Fails | Good |
| 479 | 0 out of 229 | 0 | 0 out of 229 | 0 | Good |
| 541 | 0 out of 229 | 0 | 0 out of 229 | 0 | Good |
| 607 | Fails | Fails | Fails | Fails | Good |
| 733 | Fails | Fails | Fails | Fails | Good |
| 827 | Fails | Fails | Fails | Fails | Good |
| 991 | Fails | Fails | Fails | Fails | Good |
| 1021 | Fails | Fails | Fails | Fails | Good |

**Results for ECA One Rule Page 4**

Without Seed Removed

| Rule | 150 | | | | |
|---|---|---|---|---|---|
| | CA Tabs | CA Tabs | Hardware | Hardware | End of File |
| Bit Size | Overall | p val | Overall | p val | Compared |
| 31 | 0 out of 229 | 0 | 0 out of 229 | 0 | Good |
| 61 | 51 out of 229 | 0 | 51 out of 229 | 0 | Good |
| 127 | Fails | Fails | Fails | Fails | Good |
| 223 | 19 out of 229 | 0 | 19 out of 229 | 0 | Good |
| 383 | 0 out of 229 | 0 | 0 out of 229 | 0 | Good |
| 479 | 1 out of 229 | 0 | 1 out of 229 | 0 | Good |
| 541 | 2 out of 229 | 0 | 2 out of 229 | 0 | Good |
| 607 | 1 out of 229 | 0 | 1 out of 229 | 0 | Good |
| 733 | 0 out of 229 | 0 | 0 out of 229 | 0 | Good |
| 827 | 0 out of 229 | 0 | 0 out of 229 | 0 | Good |
| 991 | 0 out of 229 | 0 | 0 out of 229 | 0 | Good |
| 1021 | Fails | Fails | Fails | Fails | Good |

With Seed Removed

| Rule | 150 | | | | |
|---|---|---|---|---|---|
| | CA Tabs | CA Tabs | Hardware | Hardware | End of File |
| Bit Size | Overall | p val | Overall | p val | Compared |
| 31 | 0 out of 229 | 0 | 0 out of 229 | 0 | Good |
| 61 | 51 out of 229 | 0 | 51 out of 229 | 0 | Good |
| 127 | Fails | Fails | Fails | Fails | Good |
| 223 | 14 out of 229 | 0 | 14 out of 229 | 0 | Good |
| 383 | 0 out of 229 | 0 | 0 out of 229 | 0 | Good |
| 479 | 2 out of 229 | 0 | 2 out of 229 | 0 | Good |
| 541 | 0 out of 229 | 0 | 0 out of 229 | 0 | Good |
| 607 | 0 out of 229 | 0 | 0 out of 229 | 0 | Good |
| 733 | 0 out of 229 | 0 | 0 out of 229 | 0 | Good |
| 827 | 0 out of 229 | 0 | 0 out of 229 | 0 | Good |
| 991 | 0 out of 229 | 0 | 0 out of 229 | 0 | Good |
| 1021 | Fails | Fails | Fails | Fails | Good |

APPENDIX H

## Results for Dual Rules Page 1

**Table 6: Dual Rule 31 Bits**

| b31 | | | | | b31 | Seed Removed | | |
|---|---|---|---|---|---|---|---|---|
| Rule 1 | Rule 2 | #Passed out of 229 | Overall | | Rule 1 | Rule 2 | #Passed out of 229 | Overall |
| 30 | 45 | 42 | 0 | | 30 | 45 | 36 | 0 |
| 30 | 60 | 40 | 0 | | 30 | 60 | 42 | 0 |
| 30 | 90 | 43 | 0 | | 30 | 90 | 39 | 0 |
| 30 | 150 | 48 | 0 | | 30 | 150 | 49 | 0 |
| 45 | 60 | 39 | 0 | | 45 | 60 | 41 | 0 |
| 45 | 90 | 0 | 0 | | 45 | 90 | 0 | 0 |
| 45 | 150 | 62 | 0 | | 45 | 150 | 61 | 0 |
| 60 | 90 | Fails | Fails | | 60 | 90 | Fails | Fails |
| 60 | 150 | Fails | Fails | | 60 | 150 | 0 | 0 |
| 90 | 150 | 0 | 0 | | 90 | 150 | 0 | 0 |

**Table 7: Dual Rule 127 Bits**

| B127 | | | | | b127 | Seed Removed | | |
|---|---|---|---|---|---|---|---|---|
| Rule 1 | Rule 2 | #Passed out of 229 | Overall | | Rule 1 | Rule 2 | #Passed out of 229 | Overall |
| 30 | 45 | 129 | 0 | | 30 | 45 | 131 | 0 |
| 30 | 60 | 134 | 0 | | 30 | 60 | 138 | 0 |
| 30 | 90 | 135 | 0 | | 30 | 90 | 138 | 0 |
| 30 | 150 | 136 | 0 | | 30 | 150 | 140 | 0 |
| 45 | 60 | 136 | 0 | | 45 | 60 | 139 | 0 |
| 45 | 90 | Fails | Fails | | 45 | 90 | Fails | Fails |
| 45 | 150 | 137 | 0 | | 45 | 150 | 145 | 0 |
| 60 | 90 | Fails | Fails | | 60 | 90 | Fails | Fails |
| 60 | 150 | Fails | Fails | | 60 | 150 | Fails | Fails |
| 90 | 150 | 0 | 0 | | 90 | 150 | 0 | 0 |

## Results for Dual Rules Page 2

Table 8: Dual Rule 113 Bits

| b113 | | | | | b113 | Seed Removed | | |
|---|---|---|---|---|---|---|---|---|
| Rule 1 | Rule 2 | #Passed out of 229 | Overall | | Rule 1 | Rule 2 | #Passed out of 229 | Overall |
| 30 | 45 | 196 | 0 | | 30 | 45 | 226 | 0.001614 |
| 30 | 60 | 225 | 1.80E-05 | | 30 | 60 | 229 | 0.277377 |
| 30 | 90 | 225 | 0.149298 | | 30 | 90 | 229 | 0.208124 |
| 30 | 150 | 225 | 0.000364 | | 30 | 150 | 229 | 0.250332 |
| 45 | 60 | 226 | 0.035638 | | 45 | 60 | 228 | 0.553137 |
| 45 | 90 | 223 | 0 | | 45 | 90 | 229 | 0.271481 |
| 45 | 150 | 225 | 0.082326 | | 45 | 150 | 229 | 0.421195 |
| 60 | 90 | 1 | 0 | | 60 | 90 | 0 | 0 |
| 60 | 150 | 9 | 0 | | 60 | 150 | 8 | 0 |
| 90 | 150 | 42 | 0 | | 90 | 150 | 39 | 0 |

Appendix I

## VHDL Source Code of CA_Bits Unit

```
--CA_Bits.vhd
--Programmer William Schenck
--Middle Tennessee State University
--
-- Cellular Automata Logic Unit
--
-- This VHDL file defines a standard CA cell
-- and then uses it it the main entity. The
-- main entity creates the cell count defined
-- by the user in the generic clause. The cells
-- are then generated.
--
-- The rule input is set by the user. This is an
-- eight bit binary rule number.
-- Example: Rule 30 = 0X1E.
--
-- The set input sets the msb to 1 and all
-- others to 0.
--
-- To increase cell bit size, user must change
-- the generic number on line number 81 to the
-- desired number.
--
-- At present the generic number is set to 128
-- change the above number when you change the
-- generic number.

-- Define CA_Bit
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
entity CA_Bit is
port(
      clk, ena, prev_cell, this_cell, next_cell     :     in
      std_logic;
      clr, set    :     in    std_logic;
      rule  :     std_logic_vector(7 downto 0);
      q     :     out   std_logic
);
end CA_Bit;
architecture cell of CA_Bit is
      signal      sel   :     std_logic_vector(2 downto 0);
     signal      y     :     std_logic;
begin
     sel(2 downto 0) <=     next_cell & this_cell & prev_cell;

     process(rule, sel)
     begin
           case sel is
                 when "000" =>    y     <=    rule(0);
                 when "001" =>    y     <=    rule(1);
                 when "010" =>    y     <=    rule(2);
                 when "011" =>    y     <=    rule(3);
                 when "100" =>    y     <=    rule(4);
                 when "101" =>    y     <=    rule(5);
                 when "110" =>    y     <=    rule(6);
                 when "111" =>    y     <=    rule(7);
           end case;
     end process;

     process(clk, ena, set, clr, y)
     begin
           if(clr = '1') then
                 q     <=    '0';
           elsif(set = '1') then
                 q     <=    '1';
           elsif(clk'event and clk = '1') then
                 if(ena = '1') then
                       q     <=    y;
                 end if;
           end if;
     end process;
end cell;

-- Define CA_Bits
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
entity CA_Bits is
generic (cell_size     :     integer     :=    128); -- defines num of
bits
port(
           clock, enable, set_word      :     in    std_logic;
           rule_in     :     in    std_logic_vector(7 downto 0);
           word_out    :     buffer      std_logic_vector(cell_size -1
downto 0)
);
end CA_Bits;
architecture logic of CA_Bits is
component CA_Bit
```

```
port(
      clk, ena, prev_cell, this_cell, next_cell, clr, set :     in
      std_logic;
      rule  :     std_logic_vector(7 downto 0);
      q     :     out   std_logic
);
end component;
begin
cell_lsb:   CA_Bit
      port map(
            clk   =>    clock,
            ena   =>    enable,
            next_cell   =>    word_out(1),
            this_cell   =>    word_out(0),
            prev_cell   =>    word_out(cell_size -1),
            clr   =>    set_word,
            set   =>    '0',
            rule(7 downto 0) =>    rule_in(7 downto 0),
            q     =>    word_out(0)
      );

cells:
      for i in 1 to cell_size - 2 generate
      cell: CA_Bit port map (
            clk   =>    clock,
            ena   =>    enable,
            next_cell   =>    word_out(i +1),
            this_cell   =>    word_out(i),
            prev_cell   =>    word_out(i -1),
            clr   =>    set_word,
            set   =>    '0',
            rule(7 downto 0) =>    rule_in(7 downto 0),
            q     =>    word_out(i)
      );
      end generate;

cell_msb:   CA_Bit
      port map(
            clk   =>    clock,
            ena   =>    enable,
            next_cell   =>    word_out(0),
            this_cell   =>    word_out(cell_size -1),
            prev_cell   =>    word_out(cell_size -2),
            clr   =>    '0',
            set   =>    set_word,
            rule(7 downto 0) =>    rule_in(7 downto 0),
            q     =>    word_out(cell_size -1)
      );
end logic;
```